Security For Windows Nodes

This page describes security considerations and best practices specific to the Windows operating system.

Protection for Secret data on nodes

On Windows, data from Secrets are written out in clear text onto the node's local storage (as compared to using tmpfs / in-memory filesystems on Linux). As a cluster operator, you should take both of the following additional measures:

1. Use file ACLs to secure the Secrets' file location.
2. Apply volume-level encryption using BitLocker.

Container users

RunAsUsername can be specified for Windows Pods or containers to execute the container processes as specific user. This is roughly equivalent to RunAsUser.

Windows containers offer two default user accounts, ContainerUser and ContainerAdministrator. The differences between these two user accounts are covered in When to use ContainerAdmin and ContainerUser user accounts within Microsoft's *Secure Windows containers* documentation.

Local users can be added to container images during the container build process.

Note:

- Nano Server based images run as `ContainerUser` by default
- Server Core based images run as `ContainerAdministrator` by default

Windows containers can also run as Active Directory identities by utilizing Group Managed Service Accounts

Pod-level security isolation

Linux-specific pod security context mechanisms (such as SELinux, AppArmor, Seccomp, or custom POSIX capabilities) are not supported on Windows nodes.

Privileged containers are not supported on Windows. Instead HostProcess containers can be used on Windows to perform many of the tasks performed by privileged containers on Linux.

Dynamic Volume Provisioning

Dynamic volume provisioning allows storage volumes to be created on-demand. Without dynamic provisioning, cluster administrators have to manually make calls to their cloud or storage provider to create new storage volumes, and then create `PersistentVolume` objects to represent them in Kubernetes. The dynamic provisioning feature eliminates the need for cluster administrators to pre-provision storage. Instead, it automatically provisions storage when users create `PersistentVolumeClaim` objects.

Background

The implementation of dynamic volume provisioning is based on the API object `StorageClass` from the API group `storage.k8s.io`. A cluster administrator can define as many `StorageClass` objects as needed, each specifying a *volume plugin* (aka *provisioner*) that provisions a volume and the set of parameters to pass to that provisioner when provisioning. A cluster administrator can define and expose multiple flavors of storage (from the same or different storage systems) within a cluster, each with a custom set of parameters. This design also ensures that end users don't have to worry about the complexity and nuances of how storage is provisioned, but still have the ability to select from multiple storage options.

For more details, see the Storage Classes concept.

Enabling Dynamic Provisioning

To enable dynamic provisioning, a cluster administrator needs to pre-create one or more StorageClass objects for users. StorageClass objects define which provisioner should be used and what parameters should be passed to that provisioner when dynamic provisioning is invoked. The name of a StorageClass object must be a valid DNS subdomain name.

The following manifest creates a storage class "slow" which provisions standard disk-like persistent disks.

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: slowprovisioner: kubernetes.io/gce-pdparameters:  type: pd-standard
```

The following manifest creates a storage class "fast" which provisions SSD-like persistent disks.

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: fastprovisioner: kubernetes.io/gce-pdparameters:  type: pd-ssd
```

Using Dynamic Provisioning

Users request dynamically provisioned storage by including a storage class in their `PersistentVolumeClaim`. Before Kubernetes v1.6, this was done via the `volume.beta.kubernetes.io/storage-class` annotation. However, this annotation is deprecated since v1.9. Users now can and should instead use the `storageClassName` field of the `PersistentVolumeClaim` object. The value of this field must match the name of a `StorageClass` configured by the administrator (see Enabling Dynamic Provisioning).

To select the "fast" storage class, for example, a user would create the following PersistentVolumeClaim:

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: claim1spec:  accessModes:    - ReadWriteOnce  storageClassName: fast  resources:    re
```

This claim results in an SSD-like Persistent Disk being automatically provisioned. When the claim is deleted, the volume is destroyed.

Defaulting Behavior

Dynamic provisioning can be enabled on a cluster such that all claims are dynamically provisioned if no storage class is specified. A cluster administrator can enable this behavior by:

- Marking one `StorageClass` object as *default*.
- Making sure that the `DefaultStorageClass` admission controller is enabled on the API server.

An administrator can mark a specific `StorageClass` as default by adding the `storageclass.kubernetes.io/is-default-class` annotation to it. When a default `StorageClass` exists in a cluster and a user creates a `PersistentVolumeClaim` with `storageClassName` unspecified, the `DefaultStorageClass` admission controller automatically adds the `storageClassName` field pointing to the default storage class.

Note that if you set the `storageclass.kubernetes.io/is-default-class` annotation to true on more than one StorageClass in your cluster, and you then create a `PersistentVolumeClaim` with no `storageClassName` set, Kubernetes uses the most recently created default StorageClass.

Topology Awareness

In Multi-Zone clusters, Pods can be spread across Zones in a Region. Single-Zone storage backends should be provisioned in the Zones where Pods are scheduled. This can be accomplished by setting the Volume Binding Mode.

Hardening Guide - Authentication Mechanisms

Information on authentication options in Kubernetes and their security properties.

Selecting the appropriate authentication mechanism(s) is a crucial aspect of securing your cluster. Kubernetes provides several built-in mechanisms, each with its own strengths and weaknesses that should be carefully considered when choosing the best authentication mechanism for your cluster.

In general, it is recommended to enable as few authentication mechanisms as possible to simplify user management and prevent cases where users retain access to a cluster that is no longer required.

It is important to note that Kubernetes does not have an in-built user database within the cluster. Instead, it takes user information from the configured authentication system and uses that to make authorization decisions. Therefore, to audit user access, you need to review credentials from every configured authentication source.

For production clusters with multiple users directly accessing the Kubernetes API, it is recommended to use external authentication sources such as OIDC. The internal authentication mechanisms, such as client certificates and service account tokens, described below, are not suitable for this use case.

X.509 client certificate authentication

Kubernetes leverages X.509 client certificate authentication for system components, such as when the kubelet authenticates to the API Server. While this mechanism can also be used for user authentication, it might not be suitable for production use due to several restrictions:

- Client certificates cannot be individually revoked. Once compromised, a certificate can be used by an attacker until it expires. To mitigate this risk, it is recommended to configure short lifetimes for user authentication credentials created using client certificates.
- If a certificate needs to be invalidated, the certificate authority must be re-keyed, which can introduce availability risks to the cluster.
- There is no permanent record of client certificates created in the cluster. Therefore, all issued certificates must be recorded if you need to keep track of them.
- Private keys used for client certificate authentication cannot be password-protected. Anyone who can read the file containing the key will be able to make use of it.
- Using client certificate authentication requires a direct connection from the client to the API server without any intervening TLS termination points, which can complicate network architectures.
- Group data is embedded in the `O` value of the client certificate, which means the user's group memberships cannot be changed for the lifetime of the certificate.

Static token file

Although Kubernetes allows you to load credentials from a static token file located on the control plane node disks, this approach is not recommended for production servers due to several reasons:

- Credentials are stored in clear text on control plane node disks, which can be a security risk.
- Changing any credential requires a restart of the API server process to take effect, which can impact availability.
- There is no mechanism available to allow users to rotate their credentials. To rotate a credential, a cluster administrator must modify the token on disk and distribute it to the users.
- There is no lockout mechanism available to prevent brute-force attacks.

Bootstrap tokens

Bootstrap tokens are used for joining nodes to clusters and are not recommended for user authentication due to several reasons:

- They have hard-coded group memberships that are not suitable for general use, making them unsuitable for authentication purposes.
- Manually generating bootstrap tokens can lead to weak tokens that can be guessed by an attacker, which can be a security risk.
- There is no lockout mechanism available to prevent brute-force attacks, making it easier for attackers to guess or crack the token.

ServiceAccount secret tokens

Service account secrets are available as an option to allow workloads running in the cluster to authenticate to the API server. In Kubernetes < 1.23, these were the default option, however, they are being replaced with TokenRequest API tokens. While these secrets could be used for user authentication, they are generally unsuitable for a number of reasons:

- They cannot be set with an expiry and will remain valid until the associated service account is deleted.
- The authentication tokens are visible to any cluster user who can read secrets in the namespace that they are defined in.
- Service accounts cannot be added to arbitrary groups complicating RBAC management where they are used.

TokenRequest API tokens

The TokenRequest API is a useful tool for generating short-lived credentials for service authentication to the API server or third-party systems. However, it is not generally recommended for user authentication as there is no revocation method available, and distributing credentials to users in a secure manner can be challenging.

When using TokenRequest tokens for service authentication, it is recommended to implement a short lifespan to reduce the impact of compromised tokens.

OpenID Connect token authentication

Kubernetes supports integrating external authentication services with the Kubernetes API using OpenID Connect (OIDC). There is a wide variety of software that can be used to integrate Kubernetes with an identity provider. However, when using OIDC authentication in Kubernetes, it is important to consider the following hardening measures:

- The software installed in the cluster to support OIDC authentication should be isolated from general workloads as it will run with high privileges.
- Some Kubernetes managed services are limited in the OIDC providers that can be used.
- As with TokenRequest tokens, OIDC tokens should have a short lifespan to reduce the impact of compromised tokens.

Webhook token authentication

Webhook token authentication is another option for integrating external authentication providers into Kubernetes. This mechanism allows for an authentication service, either running inside the cluster or externally, to be contacted for an authentication decision over a webhook. It is important to note that the suitability of this mechanism will likely depend on the software used for the authentication service, and there are some Kubernetes-specific considerations to take into account.

To configure Webhook authentication, access to control plane server filesystems is required. This means that it will not be possible with Managed Kubernetes unless the provider specifically makes it available. Additionally, any software installed in the cluster to support this access should be isolated from general workloads, as it will run with high privileges.

Authenticating proxy

Another option for integrating external authentication systems into Kubernetes is to use an authenticating proxy. With this mechanism, Kubernetes expects to receive requests from the proxy with specific header values set, indicating the username and group memberships to assign for authorization purposes. It is important to note that there are specific considerations to take into account when using this mechanism.

Firstly, securely configured TLS must be used between the proxy and Kubernetes API server to mitigate the risk of traffic interception or sniffing attacks. This ensures that the communication between the proxy and Kubernetes API server is secure.

Secondly, it is important to be aware that an attacker who is able to modify the headers of the request may be able to gain unauthorized access to Kubernetes resources. As such, it is important to ensure that the headers are properly secured and cannot be tampered with.

What's next

- User Authentication
- Authenticating with Bootstrap Tokens
- kubelet Authentication
- Authenticating with Service Account Tokens

Service Accounts

Learn about ServiceAccount objects in Kubernetes.

This page introduces the ServiceAccount object in Kubernetes, providing information about how service accounts work, use cases, limitations, alternatives, and links to resources for additional guidance.

What are service accounts?

A service account is a type of non-human account that, in Kubernetes, provides a distinct identity in a Kubernetes cluster. Application Pods, system components, and entities inside and outside the cluster can use a specific ServiceAccount's credentials to identify as that ServiceAccount. This identity is useful in various situations, including authenticating to the API server or implementing identity-based security policies.

Service accounts exist as ServiceAccount objects in the API server. Service accounts have the following properties:

- **Namespaced:** Each service account is bound to a Kubernetes namespace. Every namespace gets a `default` ServiceAccount upon creation.
- **Lightweight:** Service accounts exist in the cluster and are defined in the Kubernetes API. You can quickly create service accounts to enable specific tasks.
- **Portable:** A configuration bundle for a complex containerized workload might include service account definitions for the system's components. The lightweight nature of service accounts and the namespaced identities make the configurations portable.

Service accounts are different from user accounts, which are authenticated human users in the cluster. By default, user accounts don't exist in the Kubernetes API server; instead, the API server treats user identities as opaque data. You can authenticate as a user account using multiple methods. Some Kubernetes distributions might add custom extension APIs to represent user accounts in the API server.

Description	ServiceAccount	User or group
Location	Kubernetes API (ServiceAccount object)	External
Access control	Kubernetes RBAC or other authorization mechanisms	Kubernetes RBAC or other identity and access management mechanisms
Intended use	Workloads, automation	People

Default service accounts

When you create a cluster, Kubernetes automatically creates a ServiceAccount object named `default` for every namespace in your cluster. The `default` service accounts in each namespace get no permissions by default other than the default API discovery permissions that Kubernetes grants to all authenticated principals if role-based access control (RBAC) is enabled. If you delete the `default` ServiceAccount object in a namespace, the control plane replaces it with a new one.

If you deploy a Pod in a namespace, and you don't manually assign a ServiceAccount to the Pod, Kubernetes assigns the `default` ServiceAccount for that namespace to the Pod.

Use cases for Kubernetes service accounts

As a general guideline, you can use service accounts to provide identities in the following scenarios:

- Your Pods need to communicate with the Kubernetes API server, for example in situations such as the following:
  - Providing read-only access to sensitive information stored in Secrets.
  - Granting cross-namespace access, such as allowing a Pod in namespace `example` to read, list, and watch for Lease objects in the `kube-node-lease` namespace.
- Your Pods need to communicate with an external service. For example, a workload Pod requires an identity for a commercially available cloud API, and the commercial provider allows configuring a suitable trust relationship.
- Authenticating to a private image registry using an `imagePullSecret`.
- An external service needs to communicate with the Kubernetes API server. For example, authenticating to the cluster as part of a CI/CD pipeline.
- You use third-party security software in your cluster that relies on the ServiceAccount identity of different Pods to group those Pods into different contexts.

How to use service accounts

To use a Kubernetes service account, you do the following:

1. Create a ServiceAccount object using a Kubernetes client like `kubectl` or a manifest that defines the object.
2. Grant permissions to the ServiceAccount object using an authorization mechanism such as RBAC.
3. Assign the ServiceAccount object to Pods during Pod creation.

   If you're using the identity from an external service, retrieve the ServiceAccount token and use it from that service instead.

For instructions, refer to Configure Service Accounts for Pods.

Grant permissions to a ServiceAccount

You can use the built-in Kubernetes role-based access control (RBAC) mechanism to grant the minimum permissions required by each service account. You create a *role*, which grants access, and then *bind* the role to your ServiceAccount. RBAC lets you define a minimum set of permissions so that the service account permissions follow the principle of least privilege. Pods that use that service account don't get more permissions than are required to function correctly.

For instructions, refer to ServiceAccount permissions.

Cross-namespace access using a ServiceAccount

You can use RBAC to allow service accounts in one namespace to perform actions on resources in a different namespace in the cluster. For example, consider a scenario where you have a service account and Pod in the `dev` namespace and you want your Pod to see Jobs running in the `maintenance` namespace. You could create a Role object that grants permissions to list Job objects. Then, you'd create a RoleBinding object in the `maintenance` namespace to bind the Role to the ServiceAccount object. Now, Pods in the `dev` namespace can list Job objects in the `maintenance` namespace using that service account.

Assign a ServiceAccount to a Pod

To assign a ServiceAccount to a Pod, you set the `spec.serviceAccountName` field in the Pod specification. Kubernetes then automatically provides the credentials for that ServiceAccount to the Pod. In v1.22 and later, Kubernetes gets a short-lived, **automatically rotating** token using the `TokenRequest` API and mounts the token as a projected volume.

By default, Kubernetes provides the Pod with the credentials for an assigned ServiceAccount, whether that is the `default` ServiceAccount or a custom ServiceAccount that you specify.

To prevent Kubernetes from automatically injecting credentials for a specified ServiceAccount or the `default` ServiceAccount, set the `automountServiceAccountToken` field in your Pod specification to `false`.

In versions earlier than 1.22, Kubernetes provides a long-lived, static token to the Pod as a Secret.

Manually retrieve ServiceAccount credentials

If you need the credentials for a ServiceAccount to mount in a non-standard location, or for an audience that isn't the API server, use one of the following methods:

- TokenRequest API (recommended): Request a short-lived service account token from within your own *application code*. The token expires automatically and can rotate upon expiration. If you have a legacy application that is not aware of Kubernetes, you could use a sidecar container within the same pod to fetch these tokens and make them available to the application workload.
- Token Volume Projection (also recommended): In Kubernetes v1.20 and later, use the Pod specification to tell the kubelet to add the service account token to the Pod as a *projected volume*. Projected tokens expire automatically, and the kubelet rotates the token before it expires.
- Service Account Token Secrets (not recommended): You can mount service account tokens as Kubernetes Secrets in Pods. These tokens don't expire and don't rotate. In versions prior to v1.24, a permanent token was automatically created for each service account. This method is not recommended anymore, especially at scale, because of the risks associated with static, long-lived credentials. The LegacyServiceAccountTokenNoAutoGeneration feature gate (which was enabled by default from Kubernetes v1.24 to v1.26), prevented Kubernetes from automatically creating these tokens for ServiceAccounts. The feature gate is removed in v1.27, because it was elevated to GA status; you can still create indefinite service account tokens manually, but should take into account the security implications.

Note:

For applications running outside your Kubernetes cluster, you might be considering creating a long-lived ServiceAccount token that is stored in a Secret. This allows authentication, but the Kubernetes project recommends you avoid this approach. Long-lived bearer tokens represent a security risk as, once disclosed, the token can be misused. Instead, consider using an alternative. For example, your external application can authenticate using a well-protected private key `and` a certificate, or using a custom mechanism such as an authentication webhook that you implement yourself.

You can also use TokenRequest to obtain short-lived tokens for your external application.

Restricting access to Secrets (deprecated)

FEATURE STATE: `Kubernetes v1.32 [deprecated]`

Note:

`kubernetes.io/enforce-mountable-secrets` is deprecated since Kubernetes v1.32. Use separate namespaces to isolate access to mounted secrets.

Kubernetes provides an annotation called `kubernetes.io/enforce-mountable-secrets` that you can add to your ServiceAccounts. When this annotation is applied, the ServiceAccount's secrets can only be mounted on specified types of resources, enhancing the security posture of your cluster.

You can add the annotation to a ServiceAccount using a manifest:

```
apiVersion: v1
kind: ServiceAccountmetadata:  annotations:    kubernetes.io/enforce-mountable-secrets: "true"  name: my-serviceaccount  namespace
```

When this annotation is set to "true", the Kubernetes control plane ensures that the Secrets from this ServiceAccount are subject to certain mounting restrictions.

1. The name of each Secret that is mounted as a volume in a Pod must appear in the `secrets` field of the Pod's ServiceAccount.
2. The name of each Secret referenced using `envFrom` in a Pod must also appear in the `secrets` field of the Pod's ServiceAccount.
3. The name of each Secret referenced using `imagePullSecrets` in a Pod must also appear in the `secrets` field of the Pod's ServiceAccount.

By understanding and enforcing these restrictions, cluster administrators can maintain a tighter security profile and ensure that secrets are accessed only by the appropriate resources.

Authenticating service account credentials

ServiceAccounts use signed JSON Web Tokens (JWTs) to authenticate to the Kubernetes API server, and to any other system where a trust relationship exists. Depending on how the token was issued (either time-limited using a `TokenRequest` or using a legacy mechanism with a Secret), a ServiceAccount token might also have an expiry time, an audience, and a time after which the token *starts* being valid. When a client that is acting as a ServiceAccount tries to communicate with the Kubernetes API server, the client includes an `Authorization: Bearer <token>` header with the HTTP request. The API server checks the validity of that bearer token as follows:

1. Checks the token signature.
2. Checks whether the token has expired.
3. Checks whether object references in the token claims are currently valid.
4. Checks whether the token is currently valid.
5. Checks the audience claims.

The TokenRequest API produces *bound tokens* for a ServiceAccount. This binding is linked to the lifetime of the client, such as a Pod, that is acting as that ServiceAccount. See Token Volume Projection for an example of a bound pod service account token's JWT schema and payload.

For tokens issued using the `TokenRequest` API, the API server also checks that the specific object reference that is using the ServiceAccount still exists, matching by the unique ID of that object. For legacy tokens that are mounted as Secrets in Pods, the API server checks the token against the Secret.

For more information about the authentication process, refer to Authentication.

Authenticating service account credentials in your own code

If you have services of your own that need to validate Kubernetes service account credentials, you can use the following methods:

- TokenReview API (recommended)
- OIDC discovery

The Kubernetes project recommends that you use the TokenReview API, because this method invalidates tokens that are bound to API objects such as Secrets, ServiceAccounts, Pods or Nodes when those objects are deleted. For example, if you delete the Pod that contains a projected ServiceAccount token, the cluster invalidates that token immediately and a TokenReview immediately fails. If you use OIDC validation instead, your clients continue to treat the token as valid until the token reaches its expiration timestamp.

Your application should always define the audience that it accepts, and should check that the token's audiences match the audiences that the application expects. This helps to minimize the scope of the token so that it can only be used in your application and nowhere else.

Alternatives

- Issue your own tokens using another mechanism, and then use Webhook Token Authentication to validate bearer tokens using your own validation service.
- Provide your own identities to Pods.
  - Use the SPIFFE CSI driver plugin to provide SPIFFE SVIDs as X.509 certificate pairs to Pods.

    □ This item links to a third party project or product that is not part of Kubernetes itself. More information

  - Use a service mesh such as Istio to provide certificates to Pods.
- Authenticate from outside the cluster to the API server without using service account tokens:
  - Configure the API server to accept OpenID Connect (OIDC) tokens from your identity provider.
  - Use service accounts or user accounts created using an external Identity and Access Management (IAM) service, such as from a cloud provider, to authenticate to your cluster.
  - Use the CertificateSigningRequest API with client certificates.
- Configure the kubelet to retrieve credentials from an image registry.
- Use a Device Plugin to access a virtual Trusted Platform Module (TPM), which then allows authentication using a private key.

What's next

- Learn how to manage your ServiceAccounts as a cluster administrator.
- Learn how to assign a ServiceAccount to a Pod.
- Read the ServiceAccount API reference.

Volumes

Kubernetes *volumes* provide a way for containers in a pod to access and share data via the filesystem. There are different kinds of volume that you can use for different purposes, such as:

- populating a configuration file based on a ConfigMap or a Secret
- providing some temporary scratch space for a pod
- sharing a filesystem between two different containers in the same pod
- sharing a filesystem between two different pods (even if those Pods run on different nodes)
- durably storing data so that it stays available even if the Pod restarts or is replaced
- passing configuration information to an app running in a container, based on details of the Pod the container is in (for example: telling a sidecar container what namespace the Pod is running in)
- providing read-only access to data in a different container image

Data sharing can be between different local processes within a container, or between different containers, or between Pods.

Why volumes are important

- **Data persistence:** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped, the container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. After a crash, kubelet restarts the container with a clean state.

- **Shared storage:** Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to set up and access a shared filesystem across all of the containers.

The Kubernetes volume abstraction can help you to solve both of these problems.

Before you learn about volumes, PersistentVolumes and PersistentVolumeClaims, you should read up about Pods and make sure that you understand how Kubernetes uses Pods to run containers.

How volumes work

Kubernetes supports many types of volumes. A Pod can use any number of volume types simultaneously. Ephemeral volume types have a lifetime linked to a specific Pod, but persistent volumes exist beyond the lifetime of any individual pod. When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes. For any kind of volume in a given pod, data is preserved across container restarts.

At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

To use a volume, specify the volumes to provide for the Pod in `.spec.volumes` and declare where to mount those volumes into containers in `.spec.containers[*].volumeMounts`.

When a pod is launched, a process in the container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes are mounted at specified paths within the container filesystem. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses.

Volumes cannot mount within other volumes (but see Using subPath for a related mechanism). Also, a volume cannot contain a hard link to anything in a different volume.

Types of volumes

Kubernetes supports several types of volumes.

awsElasticBlockStore (deprecated)

In Kubernetes 1.34, all operations for the in-tree `awsElasticBlockStore` type are redirected to the `ebs.csi.aws.com` CSI driver.

The AWSElasticBlockStore in-tree storage driver was deprecated in the Kubernetes v1.19 release and then removed entirely in the v1.27 release.

The Kubernetes project suggests that you use the AWS EBS third party storage driver instead.

azureDisk (deprecated)

In Kubernetes 1.34, all operations for the in-tree `azureDisk` type are redirected to the `disk.csi.azure.com` CSI driver.

The AzureDisk in-tree storage driver was deprecated in the Kubernetes v1.19 release and then removed entirely in the v1.27 release.

The Kubernetes project suggests that you use the Azure Disk third party storage driver instead.

azureFile (deprecated)

In Kubernetes 1.34, all operations for the in-tree `azureFile` type are redirected to the `file.csi.azure.com` CSI driver.

The AzureFile in-tree storage driver was deprecated in the Kubernetes v1.21 release and then removed entirely in the v1.30 release.

The Kubernetes project suggests that you use the Azure File third party storage driver instead.

cephfs (removed)

Kubernetes 1.34 does not include a `cephfs` volume type.

The `cephfs` in-tree storage driver was deprecated in the Kubernetes v1.28 release and then removed entirely in the v1.31 release.

cinder (deprecated)

In Kubernetes 1.34, all operations for the in-tree `cinder` type are redirected to the `cinder.csi.openstack.org` CSI driver.

The OpenStack Cinder in-tree storage driver was deprecated in the Kubernetes v1.11 release and then removed entirely in the v1.26 release.

The Kubernetes project suggests that you use the OpenStack Cinder third party storage driver instead.

configMap

A ConfigMap provides a way to inject configuration data into pods. The data stored in a ConfigMap can be referenced in a volume of type `configMap` and then consumed by containerized applications running in a pod.

When referencing a ConfigMap, you provide the name of the ConfigMap in the volume. You can customize the path to use for a specific entry in the ConfigMap. The following configuration shows how to mount the `log-config` ConfigMap onto a Pod called `configmap-pod`:

```
apiVersion: v1
kind: Podmetadata:  name: configmap-podspec:  containers:    - name: test      image: busybox:1.28      command: ['sh', '-c', 'ech
```

The `log-config` ConfigMap is mounted as a volume, and all contents stored in its `log_level` entry are mounted into the Pod at path `/etc/config/log_level.conf`. Note that this path is derived from the volume's `mountPath` and the `path` keyed with `log_level`.

Note:

- You must create a ConfigMap before you can use it.
- A ConfigMap is always mounted as `readOnly`.
- A container using a ConfigMap as a `subPath` volume mount will not receive updates when the ConfigMap changes.
- Text data is exposed as files using the UTF-8 character encoding. For other character encodings, use `binaryData`.

downwardAPI

A `downwardAPI` volume makes downward API data available to applications. Within the volume, you can find the exposed data as read-only files in plain text format.

Note:

A container using the downward API as a `subPath` volume mount does not receive updates when field values change.

See Expose Pod Information to Containers Through Files to learn more.

emptyDir

For a Pod that defines an `emptyDir` volume, the volume is created when the Pod is assigned to a node. As the name says, the `emptyDir` volume is initially empty. All containers in the Pod can read and write the same files in the `emptyDir` volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the `emptyDir` is deleted permanently.

Note:

A container crashing does *not* remove a Pod from a node. The data in an `emptyDir` volume is safe across container crashes.

Some uses for an `emptyDir` are:

- scratch space, such as for a disk-based merge sort
- checkpointing a long computation for recovery from crashes
- holding files that a content-manager container fetches while a webserver container serves the data

The `emptyDir.medium` field controls where `emptyDir` volumes are stored. By default `emptyDir` volumes are stored on whatever medium that backs the node such as disk, SSD, or network storage, depending on your environment. If you set the `emptyDir.medium` field to `"Memory"`, Kubernetes mounts a tmpfs (RAM-backed filesystem) for you instead. While tmpfs is very fast, be aware that, unlike disks, files you write count against the memory limit of the container that wrote them.

A size limit can be specified for the default medium, which limits the capacity of the `emptyDir` volume. The storage is allocated from node ephemeral storage. If that is filled up from another source (for example, log files or image overlays), the `emptyDir` may run out of capacity before this limit. If no size is specified, memory-backed volumes are sized to node allocatable memory.

Caution:

Please check here for points to note in terms of resource management when using memory-backed `emptyDir`.

emptyDir configuration example

```
apiVersion: v1
kind: Podmetadata:   name: test-pdspec:   containers:   - image: registry.k8s.io/test-webserver    name: test-container    volumeMoun
```

emptyDir memory configuration example

```
apiVersion: v1
kind: Podmetadata:   name: test-pdspec:   containers:   - image: registry.k8s.io/test-webserver    name: test-container    volumeMoun
```

fc (fibre channel)

An `fc` volume type allows an existing fibre channel block storage volume to be mounted in a Pod. You can specify single or multiple target world wide names (WWNs) using the parameter `targetWWNs` in your Volume configuration. If multiple WWNs are specified, targetWWNs expect that those WWNs are from multi-path connections.

Note:

You must configure FC SAN Zoning to allocate and mask those LUNs (volumes) to the target WWNs beforehand so that Kubernetes hosts can access them.

gcePersistentDisk (deprecated)

In Kubernetes 1.34, all operations for the in-tree `gcePersistentDisk` type are redirected to the `pd.csi.storage.gke.io` CSI driver.

The `gcePersistentDisk` in-tree storage driver was deprecated in the Kubernetes v1.17 release and then removed entirely in the v1.28 release.

The Kubernetes project suggests that you use the Google Compute Engine Persistent Disk CSI third party storage driver instead.

gitRepo (deprecated)

Warning:

The gitRepo volume plugin is deprecated and is disabled by default.

To provision a Pod that has a Git repository mounted, you can mount an emptyDir volume into an init container that clones the repo using Git, then mount the EmptyDir into the Pod's container.

You can restrict the use of gitRepo volumes in your cluster using policies, such as ValidatingAdmissionPolicy. You can use the following Common Expression Language (CEL) expression as part of a policy to reject use of gitRepo volumes:

```
!has(object.spec.volumes) || !object.spec.volumes.exists(v, has(v.gitRepo))
```

You can use this deprecated storage plugin in your cluster if you explicitly enable the GitRepoVolumeDriver feature gate.

A gitRepo volume is an example of a volume plugin. This plugin mounts an empty directory and clones a git repository into this directory for your Pod to use.

Here is an example of a gitRepo volume:

```
apiVersion: v1
kind: Podmetadata:  name: serverspec:  containers:  - image: nginx    name: nginx    volumeMounts:    - mountPath: /mypath      na
```

glusterfs (removed)

Kubernetes 1.34 does not include a glusterfs volume type.

The GlusterFS in-tree storage driver was deprecated in the Kubernetes v1.25 release and then removed entirely in the v1.26 release.

hostPath

A hostPath volume mounts a file or directory from the host node's filesystem into your Pod. This is not something that most Pods will need, but it offers a powerful escape hatch for some applications.

Warning:

Using the hostPath volume type presents many security risks. If you can avoid using a hostPath volume, you should. For example, define a local PersistentVolume, and use that instead.

If you are restricting access to specific directories on the node using admission-time validation, that restriction is only effective when you additionally require that any mounts of that hostPath volume are **read only**. If you allow a read-write mount of any host path by an untrusted Pod, the containers in that Pod may be able to subvert the read-write host mount.

Take care when using hostPath volumes, whether these are mounted as read-only or as read-write, because:

- Access to the host filesystem can expose privileged system credentials (such as for the kubelet) or privileged APIs (such as the container runtime socket) that can be used for container escape or to attack other parts of the cluster.
- Pods with identical configuration (such as created from a PodTemplate) may behave differently on different nodes due to different files on the nodes.
- hostPath volume usage is not treated as ephemeral storage usage. You need to monitor the disk usage by yourself because excessive hostPath disk usage will lead to disk pressure on the node.

Some uses for a hostPath are:

- running a container that needs access to node-level system components (such as a container that transfers system logs to a central location, accessing those logs using a read-only mount of /var/log)
- making a configuration file stored on the host system available read-only to a static pod; unlike normal Pods, static Pods cannot access ConfigMaps

hostPath volume types

In addition to the required path property, you can optionally specify a type for a hostPath volume.

The available values for type are:

Value	Behavior
""	Empty string (default) is for backward compatibility, which means that no checks will be performed before mounting the hostPath volume.
DirectoryOrCreate	If nothing exists at the given path, an empty directory will be created there as needed with permission set to 0755, having the same group and ownership with Kubelet.
Directory	A directory must exist at the given path
FileOrCreate	If nothing exists at the given path, an empty file will be created there as needed with permission set to 0644, having the same group and ownership with Kubelet.
File	A file must exist at the given path
Socket	A UNIX socket must exist at the given path
CharDevice	*(Linux nodes only)* A character device must exist at the given path
BlockDevice	*(Linux nodes only)* A block device must exist at the given path

Caution:

The FileOrCreate mode does **not** create the parent directory of the file. If the parent directory of the mounted file does not exist, the pod fails to start. To ensure that this mode works, you can try to mount directories and files separately, as shown in the FileOrCreate example for hostPath.

Some files or directories created on the underlying hosts might only be accessible by root. You then either need to run your process as root in a privileged container or modify the file permissions on the host to read from or write to a `hostPath` volume.

hostPath configuration example

- Linux node
- Windows node

```
---# This manifest mounts /data/foo on the host as /foo inside the# single container that runs within the hostpath-example-linux P
```

```
---# This manifest mounts C:\Data\foo on the host as C:\foo, inside the# single container that runs within the hostpath-example-wi
```

hostPath FileOrCreate configuration example

The following manifest defines a Pod that mounts `/var/local/aaa` inside the single container in the Pod. If the node does not already have a path `/var/local/aaa`, the kubelet creates it as a directory and then mounts it into the Pod.

If `/var/local/aaa` already exists but is not a directory, the Pod fails. Additionally, the kubelet attempts to make a file named `/var/local/aaa/1.txt` inside that directory (as seen from the host); if something already exists at that path and isn't a regular file, the Pod fails.

Here's the example manifest:

```
apiVersion: v1
kind: Podmetadata:  name: test-webserverspec:  os: { name: linux }  nodeSelector:    kubernetes.io/os: linux  containers:  - name:
```

image

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

An `image` volume source represents an OCI object (a container image or artifact) which is available on the kubelet's host machine.

An example of using the `image` volume source is:

`pods/image-volumes.yaml` Copy pods/image-volumes.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: image-volumespec:  containers:  - name: shell    command: ["sleep", "infinity"]    image: debian    volu
```

The volume is resolved at pod startup depending on which `pullPolicy` value is provided:

`Always`
: the kubelet always attempts to pull the reference. If the pull fails, the kubelet sets the Pod to `Failed`.

`Never`
: the kubelet never pulls the reference and only uses a local image or artifact. The Pod becomes `Failed` if any layers of the image aren't already present locally, or if the manifest for that image isn't already cached.

`IfNotPresent`
: the kubelet pulls if the reference isn't already present on disk. The Pod becomes `Failed` if the reference isn't present and the pull fails.

The volume gets re-resolved if the pod gets deleted and recreated, which means that new remote content will become available on pod recreation. A failure to resolve or pull the image during pod startup will block containers from starting and may add significant latency. Failures will be retried using normal volume backoff and will be reported on the pod reason and message.

The types of objects that may be mounted by this volume are defined by the container runtime implementation on a host machine. At a minimum, they must include all valid types supported by the container image field. The OCI object gets mounted in a single directory (`spec.containers[*].volumeMounts.mountPath`) and will be mounted read-only. On Linux, the container runtime typically also mounts the volume with file execution blocked (`noexec`).

Besides that:

- `subPath` or `subPathExpr` mounts for containers (`spec.containers[*].volumeMounts.[subPath,subPathExpr]`) are only supported from Kubernetes v1.33.
- The field `spec.securityContext.fsGroupChangePolicy` has no effect on this volume type.
- The `AlwaysPullImages` Admission Controller does also work for this volume source like for container images.

The following fields are available for the `image` type:

`reference`
: Artifact reference to be used. For example, you could specify `registry.k8s.io/conformance:v1.34.0` to load the files from the Kubernetes conformance test image. Behaves in the same way as `pod.spec.containers[*].image`. Pull secrets will be assembled in the same way as for the container image by looking up node credentials, service account image pull secrets, and pod spec image pull secrets. This field is optional to allow higher level config management to default or override container images in workload controllers like Deployments and StatefulSets. More info about container images

`pullPolicy`
: Policy for pulling OCI objects. Possible values are: `Always`, `Never` or `IfNotPresent`. Defaults to `Always` if `:latest` tag is specified, or `IfNotPresent` otherwise.

See the *Use an Image Volume With a Pod* example for more details on how to use the volume source.

iscsi

An `iscsi` volume allows an existing iSCSI (SCSI over IP) volume to be mounted into your Pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an `iscsi` volume are preserved and the volume is merely unmounted. This means that an iscsi volume can be pre-populated with data, and that data can be shared between pods.

Note:

You must have your own iSCSI server running with the volume created before you can use it.

A feature of iSCSI is that it can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a volume with your dataset and then serve it in parallel from as many Pods as you need. Unfortunately, iSCSI volumes can only be mounted by a single consumer in read-write mode. Simultaneous writers are not allowed.

local

A `local` volume represents a mounted local storage device such as a disk, partition or directory.

Local volumes can only be used as a statically created PersistentVolume. Dynamic provisioning is not supported.

Compared to `hostPath` volumes, `local` volumes are used in a durable and portable manner without manually scheduling pods to nodes. The system is aware of the volume's node constraints by looking at the node affinity on the PersistentVolume.

However, `local` volumes are subject to the availability of the underlying node and are not suitable for all applications. If a node becomes unhealthy, then the `local` volume becomes inaccessible to the pod. The pod using this volume is unable to run. Applications using `local` volumes must be able to tolerate this reduced availability, as well as potential data loss, depending on the durability characteristics of the underlying disk.

The following example shows a PersistentVolume using a `local` volume and `nodeAffinity`:

```
apiVersion: v1
kind: PersistentVolumemetadata:   name: example-pvspec:   capacity:     storage: 100Gi   volumeMode: Filesystem   accessModes:   - ReadW
```

You must set a PersistentVolume `nodeAffinity` when using `local` volumes. The Kubernetes scheduler uses the PersistentVolume `nodeAffinity` to schedule these Pods to the correct node.

PersistentVolume `volumeMode` can be set to "Block" (instead of the default value "Filesystem") to expose the local volume as a raw block device.

When using local volumes, it is recommended to create a StorageClass with `volumeBindingMode` set to `WaitForFirstConsumer`. For more details, see the local StorageClass example. Delaying volume binding ensures that the PersistentVolumeClaim binding decision will also be evaluated with any other node constraints the Pod may have, such as node resource requirements, node selectors, Pod affinity, and Pod anti-affinity.

An external static provisioner can be run separately for improved management of the local volume lifecycle. Note that this provisioner does not support dynamic provisioning yet. For an example on how to run an external local provisioner, see the local volume provisioner user guide.

Note:

The local PersistentVolume requires manual cleanup and deletion by the user if the external static provisioner is not used to manage the volume lifecycle.

nfs

An `nfs` volume allows an existing NFS (Network File System) share to be mounted into a Pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an `nfs` volume are preserved and the volume is merely unmounted. This means that an NFS volume can be pre-populated with data, and that data can be shared between pods. NFS can be mounted by multiple writers simultaneously.

```
apiVersion: v1
kind: Podmetadata:   name: test-pdspec:   containers:   - image: registry.k8s.io/test-webserver     name: test-container     volumeMoun
```

Note:

You must have your own NFS server running with the share exported before you can use it.

Also note that you can't specify NFS mount options in a Pod spec. You can either set mount options server-side or use /etc/nfsmount.conf. You can also mount NFS volumes via PersistentVolumes which do allow you to set mount options.

persistentVolumeClaim

A `persistentVolumeClaim` volume is used to mount a PersistentVolume into a Pod. PersistentVolumeClaims are a way for users to "claim" durable storage (such as an iSCSI volume) without knowing the details of the particular cloud environment.

See the information about PersistentVolumes for more details.

portworxVolume (deprecated)

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

A `portworxVolume` is an elastic block storage layer that runs hyperconverged with Kubernetes. Portworx fingerprints storage in a server, tiers based on capabilities, and aggregates capacity across multiple servers. Portworx runs in-guest in virtual machines or on bare metal Linux nodes.

A `portworxVolume` can be dynamically created through Kubernetes or it can also be pre-provisioned and referenced inside a Pod. Here is an example Pod referencing a pre-provisioned Portworx volume:

```
apiVersion: v1
kind: Podmetadata:   name: test-portworx-volume-podspec:   containers:   - image: registry.k8s.io/test-webserver     name: test-contai
```

Note:

Make sure you have an existing PortworxVolume with name `pxvol` before using it in the Pod.

Portworx CSI migration

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

In Kubernetes 1.34, all operations for the in-tree Portworx volumes are redirected to the `pxd.portworx.com` Container Storage Interface (CSI) Driver by default.
Portworx CSI Driver must be installed on the cluster.

projected

A projected volume maps several existing volume sources into the same directory. For more details, see projected volumes.

rbd (removed)

Kubernetes 1.34 does not include a `rbd` volume type.

The Rados Block Device (RBD) in-tree storage driver and its csi migration support were deprecated in the Kubernetes v1.28 release and then removed entirely in the v1.31 release.

secret

A `secret` volume is used to pass sensitive information, such as passwords, to Pods. You can store secrets in the Kubernetes API and mount them as files for use by pods without coupling to Kubernetes directly. `secret` volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage.

Note:

- You must create a Secret in the Kubernetes API before you can use it.
- A Secret is always mounted as `readOnly`.
- A container using a Secret as a `subPath` volume mount will not receive Secret updates.

For more details, see Configuring Secrets.

vsphereVolume (deprecated)

In Kubernetes 1.34, all operations for the in-tree `vsphereVolume` type are redirected to the `csi.vsphere.vmware.com` CSI driver.

The `vsphereVolume` in-tree storage driver was deprecated in the Kubernetes v1.19 release and then removed entirely in the v1.30 release.

The Kubernetes project suggests that you use the vSphere CSI third party storage driver instead.

Using subPath

Sometimes, it is useful to share one volume for multiple uses in a single pod. The `volumeMounts[*].subPath` property specifies a sub-path inside the referenced volume instead of its root.

The following example shows how to configure a Pod with a LAMP stack (Linux Apache MySQL PHP) using a single, shared volume. This sample `subPath` configuration is not recommended for production use.

The PHP application's code and assets map to the volume's `html` folder and the MySQL database is stored in the volume's `mysql` folder. For example:

```
apiVersion: v1
kind: Podmetadata:  name: my-lamp-sitespec:    containers:    - name: mysql      image: mysql      env:      - name: MYSQL_ROOT_PA
```

Using subPath with expanded environment variables

FEATURE STATE: `Kubernetes v1.17 [stable]`

Use the `subPathExpr` field to construct `subPath` directory names from downward API environment variables. The `subPath` and `subPathExpr` properties are mutually exclusive.

In this example, a `Pod` uses `subPathExpr` to create a directory `pod1` within the `hostPath` volume `/var/log/pods`. The `hostPath` volume takes the `Pod` name from the `downwardAPI`. The host directory `/var/log/pods/pod1` is mounted at `/logs` in the container.

```
apiVersion: v1
kind: Podmetadata:  name: pod1spec:  containers:  - name: container1    env:    - name: POD_NAME      valueFrom:        fieldRef:
```

Resources

The storage medium (such as Disk or SSD) of an `emptyDir` volume is determined by the medium of the filesystem holding the kubelet root dir (typically `/var/lib/kubelet`). There is no limit on how much space an `emptyDir` or `hostPath` volume can consume, and no isolation between containers or pods.

To learn about requesting space using a resource specification, see how to manage resources.

Out-of-tree volume plugins

The out-of-tree volume plugins include Container Storage Interface (CSI), and also FlexVolume (which is deprecated). These plugins enable storage vendors to create custom storage plugins without adding their plugin source code to the Kubernetes repository.

Previously, all volume plugins were "in-tree". The "in-tree" plugins were built, linked, compiled, and shipped with the core Kubernetes binaries. This meant that adding a new storage system to Kubernetes (a volume plugin) required checking code into the core Kubernetes code repository.

Both CSI and FlexVolume allow volume plugins to be developed independently of the Kubernetes code base, and deployed (installed) on Kubernetes clusters as extensions.

For storage vendors looking to create an out-of-tree volume plugin, please refer to the volume plugin FAQ.

csi

Container Storage Interface (CSI) defines a standard interface for container orchestration systems (like Kubernetes) to expose arbitrary storage systems to their container workloads.

Please read the CSI design proposal for more information.

Note:

Support for CSI spec versions 0.2 and 0.3 is deprecated in Kubernetes v1.13 and will be removed in a future release.

Note:

CSI drivers may not be compatible across all Kubernetes releases. Please check the specific CSI driver's documentation for supported deployments steps for each Kubernetes release and a compatibility matrix.

Once a CSI-compatible volume driver is deployed on a Kubernetes cluster, users may use the `csi` volume type to attach or mount the volumes exposed by the CSI driver.

A `csi` volume can be used in a Pod in three different ways:

- through a reference to a PersistentVolumeClaim
- with a generic ephemeral volume
- with a CSI ephemeral volume if the driver supports that

The following fields are available to storage administrators to configure a CSI persistent volume:

- `driver`: A string value that specifies the name of the volume driver to use. This value must correspond to the value returned in the `GetPluginInfoResponse` by the CSI driver as defined in the CSI spec. It is used by Kubernetes to identify which CSI driver to call out to, and by CSI driver components to identify which PV objects belong to the CSI driver.
- `volumeHandle`: A string value that uniquely identifies the volume. This value must correspond to the value returned in the `volume.id` field of the `CreateVolumeResponse` by the CSI driver as defined in the CSI spec. The value is passed as `volume_id` in all calls to the CSI volume driver when referencing the volume.
- `readOnly`: An optional boolean value indicating whether the volume is to be "ControllerPublished" (attached) as read only. Default is false. This value is passed to the CSI driver via the `readonly` field in the `ControllerPublishVolumeRequest`.
- `fsType`: If the PV's `VolumeMode` is `Filesystem`, then this field may be used to specify the filesystem that should be used to mount the volume. If the volume has not been formatted and formatting is supported, this value will be used to format the volume. This value is passed to the CSI driver via the `VolumeCapability` field of `ControllerPublishVolumeRequest`, `NodeStageVolumeRequest`, and `NodePublishVolumeRequest`.
- `volumeAttributes`: A map of string to string that specifies static properties of a volume. This map must correspond to the map returned in the `volume.attributes` field of the `CreateVolumeResponse` by the CSI driver as defined in the CSI spec. The map is passed to the CSI driver via the `volume_context` field in the `ControllerPublishVolumeRequest`, `NodeStageVolumeRequest`, and `NodePublishVolumeRequest`.
- `controllerPublishSecretRef`: A reference to the secret object containing sensitive information to pass to the CSI driver to complete the CSI `ControllerPublishVolume` and `ControllerUnpublishVolume` calls. This field is optional, and may be empty if no secret is required. If the Secret contains more than one secret, all secrets are passed.
- `nodeExpandSecretRef`: A reference to the secret containing sensitive information to pass to the CSI driver to complete the CSI `NodeExpandVolume` call. This field is optional and may be empty if no secret is required. If the object contains more than one secret, all secrets are passed. When you have configured secret data for node-initiated volume expansion, the kubelet passes that data via the `NodeExpandVolume()` call to the CSI driver. All supported versions of Kubernetes offer the `nodeExpandSecretRef` field, and have it available by default. Kubernetes releases prior to v1.25 did not include this support.
- Enable the feature gate named `CSINodeExpandSecret` for each kube-apiserver and for the kubelet on every node. Since Kubernetes version 1.27, this feature has been enabled by default and no explicit enablement of the feature gate is required. You must also be using a CSI driver that supports or requires secret data during node-initiated storage resize operations.
- `nodePublishSecretRef`: A reference to the secret object containing sensitive information to pass to the CSI driver to complete the CSI `NodePublishVolume` call. This field is optional and may be empty if no secret is required. If the secret object contains more than one secret, all secrets are passed.
- `nodeStageSecretRef`: A reference to the secret object containing sensitive information to pass to the CSI driver to complete the CSI `NodeStageVolume` call. This field is optional and may be empty if no secret is required. If the Secret contains more than one secret, all secrets are passed.

CSI raw block volume support

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

Vendors with external CSI drivers can implement raw block volume support in Kubernetes workloads.

You can set up your PersistentVolume/PersistentVolumeClaim with raw block volume support as usual, without any CSI-specific changes.

CSI ephemeral volumes

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

You can directly configure CSI volumes within the Pod specification. Volumes specified in this way are ephemeral and do not persist across pod restarts. See Ephemeral Volumes for more information.

For more information on how to develop a CSI driver, refer to the kubernetes-csi documentation

Windows CSI proxy

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

CSI node plugins need to perform various privileged operations like scanning of disk devices and mounting of file systems. These operations differ for each host operating system. For Linux worker nodes, containerized CSI node plugins are typically deployed as privileged containers. For Windows worker nodes, privileged operations for containerized CSI node plugins is supported using csi-proxy, a community-managed, stand-alone binary that needs to be pre-installed on each Windows node.

For more details, refer to the deployment guide of the CSI plugin you wish to deploy.

Migrating to CSI drivers from in-tree plugins

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

The `CSIMigration` feature directs operations against existing in-tree plugins to corresponding CSI plugins (which are expected to be installed and configured). As a result, operators do not have to make any configuration changes to existing Storage Classes, PersistentVolumes or PersistentVolumeClaims (referring to in-tree plugins) when transitioning to a CSI driver that supersedes an in-tree plugin.

Note:

Existing PVs created by an in-tree volume plugin can still be used in the future without any configuration changes, even after the migration to CSI is completed for that volume type, and even after you upgrade to a version of Kubernetes that doesn't have compiled-in support for that kind of storage.

As part of that migration, you - or another cluster administrator - **must** have installed and configured the appropriate CSI driver for that storage. The core of Kubernetes does not install that software for you.

After that migration, you can also define new PVCs and PVs that refer to the legacy, built-in storage integrations. Provided you have the appropriate CSI driver installed and configured, the PV creation continues to work, even for brand new volumes. The actual storage management now happens through the CSI driver.

The operations and features that are supported include: provisioning/delete, attach/detach, mount/unmount and resizing of volumes.

In-tree plugins that support `CSIMigration` and have a corresponding CSI driver implemented are listed in Types of Volumes.

flexVolume (deprecated)

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

FlexVolume is an out-of-tree plugin interface that uses an exec-based model to interface with storage drivers. The FlexVolume driver binaries must be installed in a pre-defined volume plugin path on each node and in some cases the control plane nodes as well.

Pods interact with FlexVolume drivers through the `flexVolume` in-tree volume plugin.

The following FlexVolume plugins, deployed as PowerShell scripts on the host, support Windows nodes:

- SMB
- iSCSI

Note:

FlexVolume is deprecated. Using an out-of-tree CSI driver is the recommended way to integrate external storage with Kubernetes.

Maintainers of FlexVolume driver should implement a CSI Driver and help to migrate users of FlexVolume drivers to CSI. Users of FlexVolume should move their workloads to use the equivalent CSI Driver.

Mount propagation

Caution:

Mount propagation is a low-level feature that does not work consistently on all volume types. The Kubernetes project recommends only using mount propagation with `hostPath` or memory-backed `emptyDir` volumes. See Kubernetes issue #95049 for more context.

Mount propagation allows for sharing volumes mounted by a container to other containers in the same pod, or even to other pods on the same node.

Mount propagation of a volume is controlled by the `mountPropagation` field in `containers[*].volumeMounts`. Its values are:

- `None` - This volume mount will not receive any subsequent mounts that are mounted to this volume or any of its subdirectories by the host. In similar fashion, no mounts created by the container will be visible on the host. This is the default mode.

  This mode is equal to `rprivate` mount propagation as described in `mount(8)`

However, the CRI runtime may choose `rslave` mount propagation (i.e., `HostToContainer`) instead, when `rprivate` propagation is not applicable. cri-dockerd (Docker) is known to choose `rslave` mount propagation when the mount source contains the Docker daemon's root directory (`/var/lib/docker`).

- `HostToContainer` - This volume mount will receive all subsequent mounts that are mounted to this volume or any of its subdirectories.

  In other words, if the host mounts anything inside the volume mount, the container will see it mounted there.

  Similarly, if any Pod with `Bidirectional` mount propagation to the same volume mounts anything there, the container with `HostToContainer` mount propagation will see it.

  This mode is equal to `rslave` mount propagation as described in the [`mount(8)`]

- `Bidirectional` - This volume mount behaves the same the `HostToContainer` mount. In addition, all volume mounts created by the container will be propagated back to the host and to all containers of all pods that use the same volume.

  A typical use case for this mode is a Pod with a FlexVolume or CSI driver or a Pod that needs to mount something on the host using a `hostPath` volume.

  This mode is equal to `rshared` mount propagation as described in the [`mount(8)`]

  Warning:

  `Bidirectional` mount propagation can be dangerous. It can damage the host operating system and therefore it is allowed only in privileged containers. Familiarity with Linux kernel behavior is strongly recommended. In addition, any volume mounts created by containers in pods must be destroyed (unmounted) by the containers on termination.

Read-only mounts

A mount can be made read-only by setting the `.spec.containers[].volumeMounts[].readOnly` field to `true`. This does not make the volume itself read-only, but that specific container will not be able to write to it. Other containers in the Pod may mount the same volume as read-write.

On Linux, read-only mounts are not recursively read-only by default. For example, consider a Pod which mounts the hosts `/mnt` as a `hostPath` volume. If there is another filesystem mounted read-write on `/mnt/<SUBMOUNT>` (such as tmpfs, NFS, or USB storage), the volume mounted into the container(s) will also have a writeable `/mnt/<SUBMOUNT>`, even if the mount itself was specified as read-only.

Recursive read-only mounts

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

Recursive read-only mounts can be enabled by setting the `RecursiveReadOnlyMounts` feature gate for kubelet and kube-apiserver, and setting the `.spec.containers[].volumeMounts[].recursiveReadOnly` field for a pod.

The allowed values are:

- `Disabled` (default): no effect.

- `Enabled`: makes the mount recursively read-only. Needs all the following requirements to be satisfied:

  - `readOnly` is set to `true`
  - `mountPropagation` is unset, or, set to `None`
  - The host is running with Linux kernel v5.12 or later
  - The CRI-level container runtime supports recursive read-only mounts
  - The OCI-level container runtime supports recursive read-only mounts.

  It will fail if any of these is not true.

- `IfPossible`: attempts to apply `Enabled`, and falls back to `Disabled` if the feature is not supported by the kernel or the runtime class.

Example:

`storage/rro.yaml` Copy storage/rro.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: rrospec:  volumes:    - name: mnt      hostPath:        # tmpfs is mounted on /mnt/tmpfs        path: /m
```

When this property is recognized by kubelet and kube-apiserver, the `.status.containerStatuses[].volumeMounts[].recursiveReadOnly` field is set to either `Enabled` or `Disabled`.

Implementations

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.

The following container runtimes are known to support recursive read-only mounts.

CRI-level:

- containerd, since v2.0
- CRI-O, since v1.30

OCI-level:

- runc, since v1.1
- crun, since v1.8.6

What's next

Follow an example of deploying WordPress and MySQL with Persistent Volumes.

Gateway API

Gateway API is a family of API kinds that provide dynamic infrastructure provisioning and advanced traffic routing.

Make network services available by using an extensible, role-oriented, protocol-aware configuration mechanism. Gateway API is an add-on containing API kinds that provide dynamic infrastructure provisioning and advanced traffic routing.

Design principles

The following principles shaped the design and architecture of Gateway API:

- **Role-oriented:** Gateway API kinds are modeled after organizational roles that are responsible for managing Kubernetes service networking:
    - **Infrastructure Provider:** Manages infrastructure that allows multiple isolated clusters to serve multiple tenants, e.g. a cloud provider.
    - **Cluster Operator:** Manages clusters and is typically concerned with policies, network access, application permissions, etc.
    - **Application Developer:** Manages an application running in a cluster and is typically concerned with application-level configuration and Service composition.
- **Portable:** Gateway API specifications are defined as custom resources and are supported by many implementations.
- **Expressive:** Gateway API kinds support functionality for common traffic routing use cases such as header-based matching, traffic weighting, and others that were only possible in Ingress by using custom annotations.
- **Extensible:** Gateway allows for custom resources to be linked at various layers of the API. This makes granular customization possible at the appropriate places within the API structure.

Resource model

Gateway API has four stable API kinds:

- **GatewayClass:** Defines a set of gateways with common configuration and managed by a controller that implements the class.

- **Gateway:** Defines an instance of traffic handling infrastructure, such as cloud load balancer.

- **HTTPRoute:** Defines HTTP-specific rules for mapping traffic from a Gateway listener to a representation of backend network endpoints. These endpoints are often represented as a Service.

- **GRPCRoute:** Defines gRPC-specific rules for mapping traffic from a Gateway listener to a representation of backend network endpoints. These endpoints are often represented as a Service.

Gateway API is organized into different API kinds that have interdependent relationships to support the role-oriented nature of organizations. A Gateway object is associated with exactly one GatewayClass; the GatewayClass describes the gateway controller responsible for managing Gateways of this class. One or more route kinds such as HTTPRoute, are then associated to Gateways. A Gateway can filter the routes that may be attached to its `listeners`, forming a bidirectional trust model with routes.

The following figure illustrates the relationships of the three stable Gateway API kinds:

A figure illustrating the relationships of the three stable Gateway API kinds

GatewayClass

Gateways can be implemented by different controllers, often with different configurations. A Gateway must reference a GatewayClass that contains the name of the controller that implements the class.

A minimal GatewayClass example:

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClassmetadata:  name: example-classspec:  controllerName: example.com/gateway-controller
```

In this example, a controller that has implemented Gateway API is configured to manage GatewayClasses with the controller name `example.com/gateway-controller`. Gateways of this class will be managed by the implementation's controller.

See the GatewayClass reference for a full definition of this API kind.

Gateway

A Gateway describes an instance of traffic handling infrastructure. It defines a network endpoint that can be used for processing traffic, i.e. filtering, balancing, splitting, etc. for backends such as a Service. For example, a Gateway may represent a cloud load balancer or an in-cluster proxy server that is configured to accept HTTP traffic.

A minimal Gateway resource example:

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gatewaymetadata:  name: example-gatewayspec:  gatewayClassName: example-class  listeners:  - name: http    protocol: HTTP
```

In this example, an instance of traffic handling infrastructure is programmed to listen for HTTP traffic on port 80. Since the `addresses` field is unspecified, an address or hostname is assigned to the Gateway by the implementation's controller. This address is used as a network endpoint for processing traffic of backend network endpoints defined in routes.

See the Gateway reference for a full definition of this API kind.

HTTPRoute

The HTTPRoute kind specifies routing behavior of HTTP requests from a Gateway listener to backend network endpoints. For a Service backend, an implementation may represent the backend network endpoint as a Service IP or the backing EndpointSlices of the Service. An HTTPRoute represents configuration that is applied to the underlying Gateway implementation. For example, defining a new HTTPRoute may result in configuring additional traffic routes in a cloud load balancer or in-cluster proxy server.

A minimal HTTPRoute example:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoutemetadata:  name: example-httproutespec:  parentRefs:  - name: example-gateway  hostnames:  - "www.example.com"  rul
```

In this example, HTTP traffic from Gateway `example-gateway` with the Host: header set to `www.example.com` and the request path specified as `/login` will be routed to Service `example-svc` on port `8080`.

See the HTTPRoute reference for a full definition of this API kind.

GRPCRoute

The GRPCRoute kind specifies routing behavior of gRPC requests from a Gateway listener to backend network endpoints. For a Service backend, an implementation may represent the backend network endpoint as a Service IP or the backing EndpointSlices of the Service. A GRPCRoute represents configuration that is applied to the underlying Gateway implementation. For example, defining a new GRPCRoute may result in configuring additional traffic routes in a cloud load balancer or in-cluster proxy server.

Gateways supporting GRPCRoute are required to support HTTP/2 without an initial upgrade from HTTP/1, so gRPC traffic is guaranteed to flow properly.

A minimal GRPCRoute example:

```
apiVersion: gateway.networking.k8s.io/v1
kind: GRPCRoutemetadata:  name: example-grpcroutespec:  parentRefs:  - name: example-gateway  hostnames:  - "svc.example.com"  rul
```

In this example, gRPC traffic from Gateway `example-gateway` with the host set to `svc.example.com` will be directed to the service `example-svc` on port `50051` from the same namespace.

GRPCRoute allows matching specific gRPC services, as per the following example:

```
apiVersion: gateway.networking.k8s.io/v1
kind: GRPCRoutemetadata:  name: example-grpcroutespec:  parentRefs:  - name: example-gateway  hostnames:  - "svc.example.com"  rul
```

In this case, the GRPCRoute will match any traffic for svc.example.com and apply its routing rules to forward the traffic to the correct backend. Since there is only one match specified,only requests for the com.example.User.Login method to svc.example.com will be forwarded. RPCs of any other method` will not be matched by this Route.

See the GRPCRoute reference for a full definition of this API kind.

Request flow

Here is a simple example of HTTP traffic being routed to a Service by using a Gateway and an HTTPRoute:

A diagram that provides an example of HTTP traffic being routed to a Service by using a Gateway and an HTTPRoute

In this example, the request flow for a Gateway implemented as a reverse proxy is:

1. The client starts to prepare an HTTP request for the URL `http://www.example.com`
2. The client's DNS resolver queries for the destination name and learns a mapping to one or more IP addresses associated with the Gateway.
3. The client sends a request to the Gateway IP address; the reverse proxy receives the HTTP request and uses the Host: header to match a configuration that was derived from the Gateway and attached HTTPRoute.
4. Optionally, the reverse proxy can perform request header and/or path matching based on match rules of the HTTPRoute.
5. Optionally, the reverse proxy can modify the request; for example, to add or remove headers, based on filter rules of the HTTPRoute.
6. Lastly, the reverse proxy forwards the request to one or more backends.

Conformance

Gateway API covers a broad set of features and is widely implemented. This combination requires clear conformance definitions and tests to ensure that the API provides a consistent experience wherever it is used.

See the conformance documentation to understand details such as release channels, support levels, and running conformance tests.

Migrating from Ingress

Gateway API is the successor to the Ingress API. However, it does not include the Ingress kind. As a result, a one-time conversion from your existing Ingress resources to Gateway API resources is necessary.

Refer to the ingress migration guide for details on migrating Ingress resources to Gateway API resources.

What's next

Instead of Gateway API resources being natively implemented by Kubernetes, the specifications are defined as Custom Resources supported by a wide range of implementations. Install the Gateway API CRDs or follow the installation instructions of your selected implementation. After installing an implementation, use the Getting Started guide to help you quickly start working with Gateway API.

Note:

Make sure to review the documentation of your selected implementation to understand any caveats.

Refer to the API specification for additional details of all Gateway API kinds.

EndpointSlices

The EndpointSlice API is the mechanism that Kubernetes uses to let your Service scale to handle large numbers of backends, and allows the cluster to update its list of healthy backends efficiently.
FEATURE STATE: `Kubernetes v1.21 [stable]`
EndpointSlices track the IP addresses of backend endpoints. EndpointSlices are normally associated with a Service and the backend endpoints typically represent Pods.

EndpointSlice API

In Kubernetes, an EndpointSlice contains references to a set of network endpoints. The control plane automatically creates EndpointSlices for any Kubernetes Service that has a selector specified. These EndpointSlices include references to all the Pods that match the Service selector. EndpointSlices group network endpoints together by unique combinations of IP family, protocol, port number, and Service name. The name of a EndpointSlice object must be a valid DNS subdomain name.

As an example, here's a sample EndpointSlice object, that's owned by the `example` Kubernetes Service.

```
apiVersion: discovery.k8s.io/v1
kind: EndpointSlicemetadata:  name: example-abc  labels:    kubernetes.io/service-name: exampleaddressType: IPv4ports:  - name: ht
```

By default, the control plane creates and manages EndpointSlices to have no more than 100 endpoints each. You can configure this with the `--max-endpoints-per-slice` kube-controller-manager flag, up to a maximum of 1000.

EndpointSlices act as the source of truth for kube-proxy when it comes to how to route internal traffic.

Address types

EndpointSlices support two address types:

- IPv4
- IPv6

Each `EndpointSlice` object represents a specific IP address type. If you have a Service that is available via IPv4 and IPv6, there will be at least two `EndpointSlice` objects (one for IPv4, and one for IPv6).

Conditions

The EndpointSlice API stores conditions about endpoints that may be useful for consumers. The three conditions are `serving`, `terminating`, and `ready`.

Serving

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

The `serving` condition indicates that the endpoint is currently serving responses, and so it should be used as a target for Service traffic. For endpoints backed by a Pod, this maps to the Pod's `Ready` condition.

Terminating

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

The `terminating` condition indicates that the endpoint is terminating. For endpoints backed by a Pod, this condition is set when the Pod is first deleted (that is, when it receives a deletion timestamp, but most likely before the Pod's containers exit).

Service proxies will normally ignore endpoints that are `terminating`, but they may route traffic to endpoints that are both `serving` and `terminating` if all available endpoints are `terminating`. (This helps to ensure that no Service traffic is lost during rolling updates of the underlying Pods.)

Ready

The `ready` condition is essentially a shortcut for checking "`serving` and not `terminating`" (though it will also always be `true` for Services with `spec.publishNotReadyAddresses` set to `true`).

Topology information

Each endpoint within an EndpointSlice can contain relevant topology information. The topology information includes the location of the endpoint and information about the corresponding Node and zone. These are available in the following per endpoint fields on EndpointSlices:

- `nodeName` - The name of the Node this endpoint is on.
- `zone` - The zone this endpoint is in.

Management

Most often, the control plane (specifically, the endpoint slice controller) creates and manages EndpointSlice objects. There are a variety of other use cases for EndpointSlices, such as service mesh implementations, that could result in other entities or controllers managing additional sets of EndpointSlices.

To ensure that multiple entities can manage EndpointSlices without interfering with each other, Kubernetes defines the label `endpointslice.kubernetes.io/managed-by`, which indicates the entity managing an EndpointSlice. The endpoint slice controller sets `endpointslice-controller.k8s.io` as the value for this label on all EndpointSlices it manages. Other entities managing EndpointSlices should also set a unique value for this label.

Ownership

In most use cases, EndpointSlices are owned by the Service that the endpoint slice object tracks endpoints for. This ownership is indicated by an owner reference on each EndpointSlice as well as a `kubernetes.io/service-name` label that enables simple lookups of all EndpointSlices belonging to a Service.

Distribution of EndpointSlices

Each EndpointSlice has a set of ports that applies to all endpoints within the resource. When named ports are used for a Service, Pods may end up with different target port numbers for the same named port, requiring different EndpointSlices.

The control plane tries to fill EndpointSlices as full as possible, but does not actively rebalance them. The logic is fairly straightforward:

1. Iterate through existing EndpointSlices, remove endpoints that are no longer desired and update matching endpoints that have changed.
2. Iterate through EndpointSlices that have been modified in the first step and fill them up with any new endpoints needed.
3. If there's still new endpoints left to add, try to fit them into a previously unchanged slice and/or create new ones.

Importantly, the third step prioritizes limiting EndpointSlice updates over a perfectly full distribution of EndpointSlices. As an example, if there are 10 new endpoints to add and 2 EndpointSlices with room for 5 more endpoints each, this approach will create a new EndpointSlice instead of filling up the 2 existing EndpointSlices. In other words, a single EndpointSlice creation is preferable to multiple EndpointSlice updates.

With kube-proxy running on each Node and watching EndpointSlices, every change to an EndpointSlice becomes relatively expensive since it will be transmitted to every Node in the cluster. This approach is intended to limit the number of changes that need to be sent to every Node, even if it may result with multiple EndpointSlices that are not full.

In practice, this less than ideal distribution should be rare. Most changes processed by the EndpointSlice controller will be small enough to fit in an existing EndpointSlice, and if not, a new EndpointSlice is likely going to be necessary soon anyway. Rolling updates of Deployments also provide a natural repacking of EndpointSlices with all Pods and their corresponding endpoints getting replaced.

Duplicate endpoints

Due to the nature of EndpointSlice changes, endpoints may be represented in more than one EndpointSlice at the same time. This naturally occurs as changes to different EndpointSlice objects can arrive at the Kubernetes client watch / cache at different times.

Note:

Clients of the EndpointSlice API must iterate through all the existing EndpointSlices associated to a Service and build a complete list of unique network endpoints. It is important to mention that endpoints may be duplicated in different EndpointSlices.

You can find a reference implementation for how to perform this endpoint aggregation and deduplication as part of the `EndpointSliceCache` code within `kube-proxy`.

EndpointSlice mirroring

FEATURE STATE: `Kubernetes v1.33 [deprecated]`

The EndpointSlice API is a replacement for the older Endpoints API. To preserve compatibility with older controllers and user workloads that expect kube-proxy to route traffic based on Endpoints resources, the cluster's control plane mirrors most user-created Endpoints resources to corresponding EndpointSlices.

(However, this feature, like the rest of the Endpoints API, is deprecated. Users who manually specify endpoints for selectorless Services should do so by creating EndpointSlice resources directly, rather than by creating Endpoints resources and allowing them to be mirrored.)

The control plane mirrors Endpoints resources unless:

- the Endpoints resource has a `endpointslice.kubernetes.io/skip-mirror` label set to `true`.
- the Endpoints resource has a `control-plane.alpha.kubernetes.io/leader` annotation.
- the corresponding Service resource does not exist.
- the corresponding Service resource has a non-nil selector.

Individual Endpoints resources may translate into multiple EndpointSlices. This will occur if an Endpoints resource has multiple subsets or includes endpoints with multiple IP families (IPv4 and IPv6). A maximum of 1000 addresses per subset will be mirrored to EndpointSlices.

What's next

- Follow the Connecting Applications with Services tutorial
- Read the API reference for the EndpointSlice API
- Read the API reference for the Endpoints API

Multi-tenancy

This page provides an overview of available configuration options and best practices for cluster multi-tenancy.

Sharing clusters saves costs and simplifies administration. However, sharing clusters also presents challenges such as security, fairness, and managing *noisy neighbors*.

Clusters can be shared in many ways. In some cases, different applications may run in the same cluster. In other cases, multiple instances of the same application may run in the same cluster, one for each end user. All these types of sharing are frequently described using the umbrella term *multi-tenancy*.

While Kubernetes does not have first-class concepts of end users or tenants, it provides several features to help manage different tenancy requirements. These are discussed below.

Use cases

The first step to determining how to share your cluster is understanding your use case, so you can evaluate the patterns and tools available. In general, multi-tenancy in Kubernetes clusters falls into two broad categories, though many variations and hybrids are also possible.

Multiple teams

A common form of multi-tenancy is to share a cluster between multiple teams within an organization, each of whom may operate one or more workloads. These workloads frequently need to communicate with each other, and with other workloads located on the same or different clusters.

In this scenario, members of the teams often have direct access to Kubernetes resources via tools such as `kubectl`, or indirect access through GitOps controllers or other types of release automation tools. There is often some level of trust between members of different teams, but Kubernetes policies such as RBAC, quotas, and network policies are essential to safely and fairly share clusters.

Multiple customers

The other major form of multi-tenancy frequently involves a Software-as-a-Service (SaaS) vendor running multiple instances of a workload for customers. This business model is so strongly associated with this deployment style that many people call it "SaaS tenancy." However, a better term might be "multi-customer tenancy," since SaaS vendors may also use other deployment models, and this deployment model can also be used outside of SaaS.

In this scenario, the customers do not have access to the cluster; Kubernetes is invisible from their perspective and is only used by the vendor to manage the workloads. Cost optimization is frequently a critical concern, and Kubernetes policies are used to ensure that the workloads are strongly isolated from each other.

Terminology

Tenants

When discussing multi-tenancy in Kubernetes, there is no single definition for a "tenant". Rather, the definition of a tenant will vary depending on whether multi-team or multi-customer tenancy is being discussed.

In multi-team usage, a tenant is typically a team, where each team typically deploys a small number of workloads that scales with the complexity of the service. However, the definition of "team" may itself be fuzzy, as teams may be organized into higher-level divisions or subdivided into smaller teams.

By contrast, if each team deploys dedicated workloads for each new client, they are using a multi-customer model of tenancy. In this case, a "tenant" is simply a group of users who share a single workload. This may be as large as an entire company, or as small as a single team at that company.

In many cases, the same organization may use both definitions of "tenants" in different contexts. For example, a platform team may offer shared services such as security tools and databases to multiple internal “customers” and a SaaS vendor may also have multiple teams sharing a development cluster. Finally, hybrid architectures are also possible, such as a SaaS provider using a combination of per-customer workloads for sensitive data, combined with multi-tenant shared services.

A cluster showing coexisting tenancy models

Isolation

There are several ways to design and build multi-tenant solutions with Kubernetes. Each of these methods comes with its own set of tradeoffs that impact the isolation level, implementation effort, operational complexity, and cost of service.

A Kubernetes cluster consists of a control plane which runs Kubernetes software, and a data plane consisting of worker nodes where tenant workloads are executed as pods. Tenant isolation can be applied in both the control plane and the data plane based on organizational requirements.

The level of isolation offered is sometimes described using terms like “hard” multi-tenancy, which implies strong isolation, and “soft” multi-tenancy, which implies weaker isolation. In particular, "hard" multi-tenancy is often used to describe cases where the tenants do not trust each other, often from security and resource sharing perspectives (e.g. guarding against attacks such as data exfiltration or DoS). Since data planes typically have much larger attack surfaces, "hard" multi-tenancy often requires extra attention to isolating the data-plane, though control plane isolation also remains critical.

However, the terms "hard" and "soft" can often be confusing, as there is no single definition that will apply to all users. Rather, "hardness" or "softness" is better understood as a broad spectrum, with many different techniques that can be used to maintain different types of isolation in your clusters, based on your requirements.

In more extreme cases, it may be easier or necessary to forgo any cluster-level sharing at all and assign each tenant their dedicated cluster, possibly even running on dedicated hardware if VMs are not considered an adequate security boundary. This may be easier with managed Kubernetes clusters, where the overhead of creating and operating clusters is at least somewhat taken on by a cloud provider. The benefit of stronger tenant isolation must be evaluated against the cost and complexity of managing multiple clusters. The Multi-cluster SIG is responsible for addressing these types of use cases.

The remainder of this page focuses on isolation techniques used for shared Kubernetes clusters. However, even if you are considering dedicated clusters, it may be valuable to review these recommendations, as it will give you the flexibility to shift to shared clusters in the future if your needs or capabilities change.

Control plane isolation

Control plane isolation ensures that different tenants cannot access or affect each others' Kubernetes API resources.

Namespaces

In Kubernetes, a Namespace provides a mechanism for isolating groups of API resources within a single cluster. This isolation has two key dimensions:

1. Object names within a namespace can overlap with names in other namespaces, similar to files in folders. This allows tenants to name their resources without having to consider what other tenants are doing.

2. Many Kubernetes security policies are scoped to namespaces. For example, RBAC Roles and Network Policies are namespace-scoped resources. Using RBAC, Users and Service Accounts can be restricted to a namespace.

In a multi-tenant environment, a Namespace helps segment a tenant's workload into a logical and distinct management unit. In fact, a common practice is to isolate every workload in its own namespace, even if multiple workloads are operated by the same tenant. This ensures that each workload has its own identity and can be configured with an appropriate security policy.

The namespace isolation model requires configuration of several other Kubernetes resources, networking plugins, and adherence to security best practices to properly isolate tenant workloads. These considerations are discussed below.

Access controls

The most important type of isolation for the control plane is authorization. If teams or their workloads can access or modify each others' API resources, they can change or disable all other types of policies thereby negating any protection those policies may offer. As a result, it is critical to ensure that each tenant has the appropriate access to only the namespaces they need, and no more. This is known as the "Principle of Least Privilege."

Role-based access control (RBAC) is commonly used to enforce authorization in the Kubernetes control plane, for both users and workloads (service accounts). Roles and RoleBindings are Kubernetes objects that are used at a namespace level to enforce access control in your application; similar objects exist for authorizing access to cluster-level objects, though these are less useful for multi-tenant clusters.

In a multi-team environment, RBAC must be used to restrict tenants' access to the appropriate namespaces, and ensure that cluster-wide resources can only be accessed or modified by privileged users such as cluster administrators.

If a policy ends up granting a user more permissions than they need, this is likely a signal that the namespace containing the affected resources should be refactored into finer-grained namespaces. Namespace management tools may simplify the management of these finer-grained namespaces by applying common RBAC policies to different namespaces, while still allowing fine-grained policies where necessary.

Quotas

Kubernetes workloads consume node resources, like CPU and memory. In a multi-tenant environment, you can use Resource Quotas to manage resource usage of tenant workloads. For the multiple teams use case, where tenants have access to the Kubernetes API, you can use resource quotas to limit the number of API resources (for example: the number of Pods, or the number of ConfigMaps) that a tenant can create. Limits on object count ensure fairness and aim to avoid *noisy neighbor* issues from affecting other tenants that share a control plane.

Resource quotas are namespaced objects. By mapping tenants to namespaces, cluster admins can use quotas to ensure that a tenant cannot monopolize a cluster's resources or overwhelm its control plane. Namespace management tools simplify the administration of quotas. In addition, while Kubernetes quotas only apply within a single namespace, some namespace management tools allow groups of namespaces to share quotas, giving administrators far more flexibility with less effort than built-in quotas.

Quotas prevent a single tenant from consuming greater than their allocated share of resources hence minimizing the “noisy neighbor” issue, where one tenant negatively impacts the performance of other tenants' workloads.

When you apply a quota to namespace, Kubernetes requires you to also specify resource requests and limits for each container. Limits are the upper bound for the amount of resources that a container can consume. Containers that attempt to consume resources that exceed the configured limits will either be throttled or killed, based on the resource type. When resource requests are set lower than limits, each container is guaranteed the requested amount but there may still be some potential for impact across workloads.

Quotas cannot protect against all kinds of resource sharing, such as network traffic. Node isolation (described below) may be a better solution for this problem.

Data Plane Isolation

Data plane isolation ensures that pods and workloads for different tenants are sufficiently isolated.

Network isolation

By default, all pods in a Kubernetes cluster are allowed to communicate with each other, and all network traffic is unencrypted. This can lead to security vulnerabilities where traffic is accidentally or maliciously sent to an unintended destination, or is intercepted by a workload on a compromised node.

Pod-to-pod communication can be controlled using Network Policies, which restrict communication between pods using namespace labels or IP address ranges. In a multi-tenant environment where strict network isolation between tenants is required, starting with a default policy that denies communication between pods is recommended with another rule that allows all pods to query the DNS server for name resolution. With such a default policy in place, you can begin adding more permissive rules that allow for communication within a namespace. It is also recommended not to use empty label selector '{}' for namespaceSelector field in network policy definition, in case traffic need to be allowed between namespaces. This scheme can be further refined as required. Note that this only applies to pods within a single control plane; pods that belong to different virtual control planes cannot talk to each other via Kubernetes networking.

Namespace management tools may simplify the creation of default or common network policies. In addition, some of these tools allow you to enforce a consistent set of namespace labels across your cluster, ensuring that they are a trusted basis for your policies.

Warning:

Network policies require a CNI plugin that supports the implementation of network policies. Otherwise, NetworkPolicy resources will be ignored.

More advanced network isolation may be provided by service meshes, which provide OSI Layer 7 policies based on workload identity, in addition to namespaces. These higher-level policies can make it easier to manage namespace-based multi-tenancy, especially when multiple namespaces are dedicated to a single tenant. They frequently also offer encryption using mutual TLS, protecting your data even in the presence of a compromised node, and work across dedicated or virtual clusters. However, they can be significantly more complex to manage and may not be appropriate for all users.

Storage isolation

Kubernetes offers several types of volumes that can be used as persistent storage for workloads. For security and data-isolation, dynamic volume provisioning is recommended and volume types that use node resources should be avoided.

StorageClasses allow you to describe custom "classes" of storage offered by your cluster, based on quality-of-service levels, backup policies, or custom policies determined by the cluster administrators.

Pods can request storage using a PersistentVolumeClaim. A PersistentVolumeClaim is a namespaced resource, which enables isolating portions of the storage system and dedicating it to tenants within the shared Kubernetes cluster. However, it is important to note that a PersistentVolume is a cluster-wide resource and has a lifecycle independent of workloads and namespaces.

For example, you can configure a separate StorageClass for each tenant and use this to strengthen isolation. If a StorageClass is shared, you should set a reclaim policy of `Delete` to ensure that a PersistentVolume cannot be reused across different namespaces.

Sandboxing containers

Kubernetes pods are composed of one or more containers that execute on worker nodes. Containers utilize OS-level virtualization and hence offer a weaker isolation boundary than virtual machines that utilize hardware-based virtualization.

In a shared environment, unpatched vulnerabilities in the application and system layers can be exploited by attackers for container breakouts and remote code execution that allow access to host resources. In some applications, like a Content Management System (CMS), customers may be allowed the ability to upload and execute untrusted scripts or code. In either case, mechanisms to further isolate and protect workloads using strong isolation are desirable.

Sandboxing provides a way to isolate workloads running in a shared cluster. It typically involves running each pod in a separate execution environment such as a virtual machine or a userspace kernel. Sandboxing is often recommended when you are running untrusted code, where workloads are assumed to be malicious. Part of the reason this type of isolation is necessary is because containers are processes running on a shared kernel; they mount file systems like `/sys` and `/proc` from the underlying host, making them less secure than an application that runs on a virtual machine which has its own kernel. While controls such as seccomp, AppArmor, and SELinux can be used to strengthen the security of containers, it is hard to apply a universal set of rules to all workloads running in a shared cluster. Running workloads in a sandbox environment helps to insulate the host from container escapes, where an attacker exploits a vulnerability to gain access to the host system and all the processes/files running on that host.

Virtual machines and userspace kernels are two popular approaches to sandboxing.

Node Isolation

Node isolation is another technique that you can use to isolate tenant workloads from each other. With node isolation, a set of nodes is dedicated to running pods from a particular tenant and co-mingling of tenant pods is prohibited. This configuration reduces the noisy tenant issue, as all pods running on a node will belong to a single tenant. The risk of information disclosure is slightly lower with node isolation because an attacker that manages to escape from a container will only have access to the containers and volumes mounted to that node.

Although workloads from different tenants are running on different nodes, it is important to be aware that the kubelet and (unless using virtual control planes) the API service are still shared services. A skilled attacker could use the permissions assigned to the kubelet or other pods running on the node to move laterally within the cluster and gain access to tenant workloads running on other nodes. If this is a major concern, consider implementing compensating controls such as seccomp, AppArmor or SELinux or explore using sandboxed containers or creating separate clusters for each tenant.

Node isolation is a little easier to reason about from a billing standpoint than sandboxing containers since you can charge back per node rather than per pod. It also has fewer compatibility and performance issues and may be easier to implement than sandboxing containers. For example, nodes for each tenant can be configured with taints so that only pods with the corresponding toleration can run on them. A mutating webhook could then be used to automatically add tolerations and node affinities to pods deployed into tenant namespaces so that they run on a specific set of nodes designated for that tenant.

Node isolation can be implemented using pod node selectors.

Additional Considerations

This section discusses other Kubernetes constructs and patterns that are relevant for multi-tenancy.

API Priority and Fairness

API priority and fairness is a Kubernetes feature that allows you to assign a priority to certain pods running within the cluster. When an application calls the Kubernetes API, the API server evaluates the priority assigned to pod. Calls from pods with higher priority are fulfilled before those with a lower priority. When contention is high, lower priority calls can be queued until the server is less busy or you can reject the requests.

Using API priority and fairness will not be very common in SaaS environments unless you are allowing customers to run applications that interface with the Kubernetes API, for example, a controller.

Quality-of-Service (QoS)

When you're running a SaaS application, you may want the ability to offer different Quality-of-Service (QoS) tiers of service to different tenants. For example, you may have freemium service that comes with fewer performance guarantees and features and a for-fee service tier with specific performance guarantees. Fortunately, there are several Kubernetes constructs that can help you accomplish this within a shared cluster, including network QoS, storage classes, and pod priority and preemption. The idea with each of these is to provide tenants with the quality of service that they paid for. Let's start by looking at networking QoS.

Typically, all pods on a node share a network interface. Without network QoS, some pods may consume an unfair share of the available bandwidth at the expense of other pods. The Kubernetes bandwidth plugin creates an extended resource for networking that allows you to use Kubernetes resources constructs, i.e. requests/limits, to apply rate limits to pods by using Linux tc queues. Be aware that the plugin is considered experimental as per the Network Plugins documentation and should be thoroughly tested before use in production environments.

For storage QoS, you will likely want to create different storage classes or profiles with different performance characteristics. Each storage profile can be associated with a different tier of service that is optimized for different workloads such IO, redundancy, or throughput. Additional logic might be necessary to allow the tenant to associate the appropriate storage profile with their workload.

Finally, there's pod priority and preemption where you can assign priority values to pods. When scheduling pods, the scheduler will try evicting pods with lower priority when there are insufficient resources to schedule pods that are assigned a higher priority. If you have a use case where tenants have different service tiers in a shared cluster e.g. free and paid, you may want to give higher priority to certain tiers using this feature.

DNS

Kubernetes clusters include a Domain Name System (DNS) service to provide translations from names to IP addresses, for all Services and Pods. By default, the Kubernetes DNS service allows lookups across all namespaces in the cluster.

In multi-tenant environments where tenants can access pods and other Kubernetes resources, or where stronger isolation is required, it may be necessary to prevent pods from looking up services in other Namespaces. You can restrict cross-namespace DNS lookups by configuring security rules for the DNS service. For example, CoreDNS (the default DNS service for Kubernetes) can leverage Kubernetes metadata to restrict queries to Pods and Services within a namespace. For more information, read an example of configuring this within the CoreDNS documentation.

When a Virtual Control Plane per tenant model is used, a DNS service must be configured per tenant or a multi-tenant DNS service must be used. Here is an example of a customized version of CoreDNS that supports multiple tenants.

Operators

Operators are Kubernetes controllers that manage applications. Operators can simplify the management of multiple instances of an application, like a database service, which makes them a common building block in the multi-consumer (SaaS) multi-tenancy use case.

Operators used in a multi-tenant environment should follow a stricter set of guidelines. Specifically, the Operator should:

- Support creating resources within different tenant namespaces, rather than just in the namespace in which the Operator is deployed.
- Ensure that the Pods are configured with resource requests and limits, to ensure scheduling and fairness.
- Support configuration of Pods for data-plane isolation techniques such as node isolation and sandboxed containers.

Implementations

There are two primary ways to share a Kubernetes cluster for multi-tenancy: using Namespaces (that is, a Namespace per tenant) or by virtualizing the control plane (that is, virtual control plane per tenant).

In both cases, data plane isolation, and management of additional considerations such as API Priority and Fairness, is also recommended.

Namespace isolation is well-supported by Kubernetes, has a negligible resource cost, and provides mechanisms to allow tenants to interact appropriately, such as by allowing service-to-service communication. However, it can be difficult to configure, and doesn't apply to Kubernetes resources that can't be namespaced, such as Custom Resource Definitions, Storage Classes, and Webhooks.

Control plane virtualization allows for isolation of non-namespaced resources at the cost of somewhat higher resource usage and more difficult cross-tenant sharing. It is a good option when namespace isolation is insufficient but dedicated clusters are undesirable, due to the high cost of maintaining them (especially on-prem) or due to their higher overhead and lack of resource sharing. However, even within a virtualized control plane, you will likely see benefits by using namespaces as well.

The two options are discussed in more detail in the following sections.

Namespace per tenant

As previously mentioned, you should consider isolating each workload in its own namespace, even if you are using dedicated clusters or virtualized control planes. This ensures that each workload only has access to its own resources, such as ConfigMaps and Secrets, and allows you to tailor dedicated security policies for each workload. In addition, it is a best practice to give each namespace names that are unique across your entire fleet (that is, even if they are in separate clusters), as this gives you the flexibility to switch between dedicated and shared clusters in the future, or to use multi-cluster tooling such as service meshes.

Conversely, there are also advantages to assigning namespaces at the tenant level, not just the workload level, since there are often policies that apply to all workloads owned by a single tenant. However, this raises its own problems. Firstly, this makes it difficult or impossible to customize policies to individual workloads, and secondly, it may be challenging to come up with a single level of "tenancy" that should be given a namespace. For example, an organization may have divisions, teams, and subteams - which should be assigned a namespace?

One possible approach is to organize your namespaces into hierarchies, and share certain policies and resources between them. This could include managing namespace labels, namespace lifecycles, delegated access, and shared resource quotas across related namespaces. These capabilities can be useful in both multi-team and multi-customer scenarios.

Virtual control plane per tenant

Another form of control-plane isolation is to use Kubernetes extensions to provide each tenant a virtual control-plane that enables segmentation of cluster-wide API resources. Data plane isolation techniques can be used with this model to securely manage worker nodes across tenants.

The virtual control plane based multi-tenancy model extends namespace-based multi-tenancy by providing each tenant with dedicated control plane components, and hence complete control over cluster-wide resources and add-on services. Worker nodes are shared across all tenants, and are managed by a Kubernetes cluster that is normally inaccessible to tenants. This cluster is often referred to as a *super-cluster* (or sometimes as a *host-cluster*). Since a tenant's control-plane is not directly associated with underlying compute resources it is referred to as a *virtual control plane*.

A virtual control plane typically consists of the Kubernetes API server, the controller manager, and the etcd data store. It interacts with the super cluster via a metadata synchronization controller which coordinates changes across tenant control planes and the control plane of the super-cluster.

By using per-tenant dedicated control planes, most of the isolation problems due to sharing one API server among all tenants are solved. Examples include noisy neighbors in the control plane, uncontrollable blast radius of policy misconfigurations, and conflicts between cluster scope objects such as webhooks and CRDs. Hence, the virtual control plane model is particularly suitable for cases where each tenant requires access to a Kubernetes API server and expects the full cluster manageability.

The improved isolation comes at the cost of running and maintaining an individual virtual control plane per tenant. In addition, per-tenant control planes do not solve isolation problems in the data plane, such as node-level noisy neighbors or security threats. These must still be addressed separately.

Projected Volumes

This document describes *projected volumes* in Kubernetes. Familiarity with volumes is suggested.

Introduction

A `projected` volume maps several existing volume sources into the same directory.

Currently, the following types of volume sources can be projected:

- `secret`
- `downwardAPI`
- `configMap`
- `serviceAccountToken`
- `clusterTrustBundle`
- `podCertificate`

All sources are required to be in the same namespace as the Pod. For more details, see the all-in-one volume design document.

Example configuration with a secret, a downwardAPI, and a configMap

`pods/storage/projected-secret-downwardapi-configmap.yaml` Copy pods/storage/projected-secret-downwardapi-configmap.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: volume-testspec:  containers:  - name: container-test    image: busybox:1.28    command: ["sleep", "3600
```

Example configuration: secrets with a non-default permission mode set

`pods/storage/projected-secrets-nondefault-permission-mode.yaml`
Copy pods/storage/projected-secrets-nondefault-permission-mode.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: volume-testspec:  containers:  - name: container-test    image: busybox:1.28    command: ["sleep", "3600
```

Each projected volume source is listed in the spec under `sources`. The parameters are nearly the same with two exceptions:

- For secrets, the `secretName` field has been changed to `name` to be consistent with ConfigMap naming.
- The `defaultMode` can only be specified at the projected level and not for each volume source. However, as illustrated above, you can explicitly set the `mode` for each individual projection.

serviceAccountToken projected volumes

You can inject the token for the current service account into a Pod at a specified path. For example:

`pods/storage/projected-service-account-token.yaml` Copy pods/storage/projected-service-account-token.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: sa-token-testspec:  containers:  - name: container-test    image: busybox:1.28    command: ["sleep", "36
```

The example Pod has a projected volume containing the injected service account token. Containers in this Pod can use that token to access the Kubernetes API server, authenticating with the identity of [the pod's ServiceAccount](). The `audience` field contains the intended audience of the token. A recipient of the token must identify itself with an identifier specified in the audience of the token, and otherwise should reject the token. This field is optional and it defaults to the identifier of the API server.

The `expirationSeconds` is the expected duration of validity of the service account token. It defaults to 1 hour and must be at least 10 minutes (600 seconds). An administrator can also limit its maximum value by specifying the `--service-account-max-token-expiration` option for the API server. The `path` field specifies a relative path to the mount point of the projected volume.

Note:

A container using a projected volume source as a [`subPath`]() volume mount will not receive updates for those volume sources.

clusterTrustBundle projected volumes

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

Note:

To use this feature in Kubernetes 1.34, you must enable support for ClusterTrustBundle objects with the `ClusterTrustBundle` [feature gate]() and `--runtime-config=certificates.k8s.io/v1beta1/clustertrustbundles=true` kube-apiserver flag, then enable the `ClusterTrustBundleProjection` feature gate.

The `clusterTrustBundle` projected volume source injects the contents of one or more [ClusterTrustBundle]() objects as an automatically-updating file in the container filesystem.

ClusterTrustBundles can be selected either by [name]() or by [signer name]().

To select by name, use the `name` field to designate a single ClusterTrustBundle object.

To select by signer name, use the `signerName` field (and optionally the `labelSelector` field) to designate a set of ClusterTrustBundle objects that use the given signer name. If `labelSelector` is not present, then all ClusterTrustBundles for that signer are selected.

The kubelet deduplicates the certificates in the selected ClusterTrustBundle objects, normalizes the PEM representations (discarding comments and headers), reorders the certificates, and writes them into the file named by `path`. As the set of selected ClusterTrustBundles or their content changes, kubelet keeps the file up-to-date.

By default, the kubelet will prevent the pod from starting if the named ClusterTrustBundle is not found, or if `signerName` / `labelSelector` do not match any ClusterTrustBundles. If this behavior is not what you want, then set the `optional` field to `true`, and the pod will start up with an empty file at `path`.

[`pods/storage/projected-clustertrustbundle.yaml`]() Copy pods/storage/projected-clustertrustbundle.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: sa-ctb-name-testspec:  containers:  - name: container-test    image: busybox    command: ["sleep", "3600
```

podCertificate projected volumes

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

Note:

In Kubernetes 1.34, you must enable support for Pod Certificates using the `PodCertificateRequest` [feature gate]() and the `--runtime-config=certificates.k8s.io/v1alpha1/podcertificaterequests=true` kube-apiserver flag.

The `podCertificate` projected volumes source securely provisions a private key and X.509 certificate chain for pod to use as client or server credentials. Kubelet will then handle refreshing the private key and certificate chain when they get close to expiration. The application just has to make sure that it reloads the file promptly when it changes, with a mechanism like `inotify` or polling.

Each `podCertificate` projection supports the following configuration fields:

- `signerName`: The [signer]() you want to issue the certificate. Note that signers may have their own access requirements, and may refuse to issue certificates to your pod.
- `keyType`: The type of private key that should be generated. Valid values are `ED25519`, `ECDSAP256`, `ECDSAP384`, `ECDSAP521`, `RSA3072`, and `RSA4096`.
- `maxExpirationSeconds`: The maximum lifetime you will accept for the certificate issued to the pod. If not set, will be defaulted to `86400` (24 hours). Must be at least `3600` (1 hour), and at most `7862400` (91 days). Kubernetes built-in signers are restricted to a max lifetime of `86400` (1 day). The signer is allowed to issue a certificate with a lifetime shorter than what you've specified.
- `credentialBundlePath`: Relative path within the projection where the credential bundle should be written. The credential bundle is a PEM-formatted file, where the first block is a "PRIVATE KEY" block that contains a PKCS#8-serialized private key, and the remaining blocks are "CERTIFICATE" blocks that comprise the certificate chain (leaf certificate and any intermediates).
- `keyPath` and `certificateChainPath`: Separate paths where Kubelet should write *just* the private key or certificate chain.

Note:

Most applications should prefer using `credentialBundlePath` unless they need the key and certificates in separate files for compatibility reasons. Kubelet uses an atomic writing strategy based on symlinks to make sure that when you open the files it projects, you read either the old content or the new content. However, if you read the key and certificate chain from separate files, Kubelet may rotate the credentials after your first read and before your second read, resulting in your application loading a mismatched key and certificate.
[`pods/storage/projected-podcertificate.yaml`]() Copy pods/storage/projected-podcertificate.yaml to clipboard

```
# Sample Pod spec that uses a podCertificate projection to request an ED25519
# private key, a certificate from the `coolcert.example.com/foo` signer, and# write the results to `/var/run/my-x509-credentials/c
```

SecurityContext interactions

The proposal for file permission handling in projected service account volume enhancement introduced the projected files having the correct owner permissions set.

Linux

In Linux pods that have a projected volume and `RunAsUser` set in the Pod `SecurityContext`, the projected files have the correct ownership set including container user ownership.

When all containers in a pod have the same `runAsUser` set in their `PodSecurityContext` or container `SecurityContext`, then the kubelet ensures that the contents of the `serviceAccountToken` volume are owned by that user, and the token file has its permission mode set to `0600`.

Note:

Ephemeral containers added to a Pod after it is created do *not* change volume permissions that were set when the pod was created.

If a Pod's `serviceAccountToken` volume permissions were set to `0600` because all other containers in the Pod have the same `runAsUser`, ephemeral containers must use the same `runAsUser` to be able to read the token.

Windows

In Windows pods that have a projected volume and `RunAsUsername` set in the Pod `SecurityContext`, the ownership is not enforced due to the way user accounts are managed in Windows. Windows stores and manages local user and group accounts in a database file called Security Account Manager (SAM). Each container maintains its own instance of the SAM database, to which the host has no visibility into while the container is running. Windows containers are designed to run the user mode portion of the OS in isolation from the host, hence the maintenance of a virtual SAM database. As a result, the kubelet running on the host does not have the ability to dynamically configure host file ownership for virtualized container accounts. It is recommended that if files on the host machine are to be shared with the container then they should be placed into their own volume mount outside of `C:\`.

By default, the projected files will have the following ownership as shown for an example projected volume file:

```
PS C:\> Get-Acl C:\var\run\secrets\kubernetes.io\serviceaccount\..2021_08_31_22_22_18.318230061\ca.crt | Format-List

Path   : Microsoft.PowerShell.Core\FileSystem::C:\var\run\secrets\kubernetes.io\serviceaccount\..2021_08_31_22_22_18.318230061\ca.
Owner  : BUILTIN\Administrators
Group  : NT AUTHORITY\SYSTEM
Access : NT AUTHORITY\SYSTEM Allow  FullControl
         BUILTIN\Administrators Allow  FullControl
         BUILTIN\Users Allow  ReadAndExecute, Synchronize
Audit  :
Sddl   : O:BAG:SYD:AI(A;ID;FA;;;SY)(A;ID;FA;;;BA)(A;ID;0x1200a9;;;BU)
```

This implies all administrator users like `ContainerAdministrator` will have read, write and execute access while, non-administrator users will have read and execute access.

Note:

In general, granting the container access to the host is discouraged as it can open the door for potential security exploits.

Creating a Windows Pod with `RunAsUser` in it's `SecurityContext` will result in the Pod being stuck at `ContainerCreating` forever. So it is advised to not use the Linux only `RunAsUser` option with Windows Pods.

Volume Attributes Classes

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

This page assumes that you are familiar with StorageClasses, volumes and PersistentVolumes in Kubernetes.

A VolumeAttributesClass provides a way for administrators to describe the mutable "classes" of storage they offer. Different classes might map to different quality-of-service levels. Kubernetes itself is un-opinionated about what these classes represent.

This feature is generally available (GA) as of version 1.34, and users have the option to disable it.

You can also only use VolumeAttributesClasses with storage backed by Container Storage Interface, and only where the relevant CSI driver implements the `ModifyVolume` API.

The VolumeAttributesClass API

Each VolumeAttributesClass contains the `driverName` and `parameters`, which are used when a PersistentVolume (PV) belonging to the class needs to be dynamically provisioned or modified.

The name of a VolumeAttributesClass object is significant and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating VolumeAttributesClass objects. While the name of a VolumeAttributesClass object in a `PersistentVolumeClaim` is mutable, the parameters in an existing class are immutable.

```
apiVersion: storage.k8s.io/v1
kind: VolumeAttributesClassmetadata:  name: silverdriverName: pd.csi.storage.gke.ioparameters:  provisioned-iops: "3000"  provisio
```

Provisioner

Each VolumeAttributesClass has a provisioner that determines what volume plugin is used for provisioning PVs. The field `driverName` must be specified.

The feature support for VolumeAttributesClass is implemented in kubernetes-csi/external-provisioner.

You are not restricted to specifying the kubernetes-csi/external-provisioner. You can also run and specify external provisioners, which are independent programs that follow a specification defined by Kubernetes. Authors of external provisioners have full discretion over where their code lives, how the provisioner is shipped, how it needs to be run, what volume plugin it uses, etc.

To understand how the provisioner works with VolumeAttributesClass, refer to the CSI external-provisioner documentation.

Resizer

Each VolumeAttributesClass has a resizer that determines what volume plugin is used for modifying PVs. The field `driverName` must be specified.

The modifying volume feature support for VolumeAttributesClass is implemented in kubernetes-csi/external-resizer.

For example, an existing PersistentVolumeClaim is using a VolumeAttributesClass named silver:

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: test-pv-claimspec:  …  volumeAttributesClassName: silver  …
```

A new VolumeAttributesClass gold is available in the cluster:

```
apiVersion: storage.k8s.io/v1
kind: VolumeAttributesClassmetadata:  name: golddriverName: pd.csi.storage.gke.ioparameters:  iops: "4000"  throughput: "60"
```

The end user can update the PVC with the new VolumeAttributesClass gold and apply:

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: test-pv-claimspec:  …  volumeAttributesClassName: gold  …
```

To understand how the resizer works with VolumeAttributesClass, refer to the CSI external-resizer documentation.

Parameters

VolumeAttributeClasses have parameters that describe volumes belonging to them. Different parameters may be accepted depending on the provisioner or the resizer. For example, the value `4000`, for the parameter `iops`, and the parameter `throughput` are specific to GCE PD. When a parameter is omitted, the default is used at volume provisioning. If a user applies the PVC with a different VolumeAttributesClass with omitted parameters, the default value of the parameters may be used depending on the CSI driver implementation. Please refer to the related CSI driver documentation for more details.

There can be at most 512 parameters defined for a VolumeAttributesClass. The total length of the parameters object including its keys and values cannot exceed 256 KiB.

Storage

Ways to provide both long-term and temporary storage to Pods in your cluster.

Assigning Pods to Nodes

You can constrain a Pod so that it is *restricted* to run on particular node(s), or to *prefer* to run on particular nodes. There are several ways to do this and the recommended approaches all use label selectors to facilitate the selection. Often, you do not need to set any such constraints; the scheduler will automatically do a reasonable placement (for example, spreading your Pods across nodes so as not place Pods on a node with insufficient free resources). However, there are some circumstances where you may want to control which node the Pod deploys to, for example, to ensure that a Pod ends up on a node with an SSD attached to it, or to co-locate Pods from two different services that communicate a lot into the same availability zone.

You can use any of the following methods to choose where Kubernetes schedules specific Pods:

- nodeSelector field matching against node labels
- Affinity and anti-affinity
- nodeName field
- Pod topology spread constraints

Node labels

Like many other Kubernetes objects, nodes have labels. You can attach labels manually. Kubernetes also populates a standard set of labels on all nodes in a cluster.

Note:

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

Node isolation/restriction

Adding labels to nodes allows you to target Pods for scheduling on specific nodes or groups of nodes. You can use this functionality to ensure that specific Pods only run on nodes with certain isolation, security, or regulatory properties.

If you use labels for node isolation, choose label keys that the kubelet cannot modify. This prevents a compromised node from setting those labels on itself so that the scheduler schedules workloads onto the compromised node.

The `NodeRestriction` admission plugin prevents the kubelet from setting or modifying labels with a `node-restriction.kubernetes.io/` prefix.

To make use of that label prefix for node isolation:

1. Ensure you are using the Node authorizer and have *enabled* the `NodeRestriction` admission plugin.
2. Add labels with the `node-restriction.kubernetes.io/` prefix to your nodes, and use those labels in your node selectors. For example, `example.com.node-restriction.kubernetes.io/fips=true` or `example.com.node-restriction.kubernetes.io/pci-dss=true`.

nodeSelector

`nodeSelector` is the simplest recommended form of node selection constraint. You can add the `nodeSelector` field to your Pod specification and specify the node labels you want the target node to have. Kubernetes only schedules the Pod onto nodes that have each of the labels you specify.

See Assign Pods to Nodes for more information.

Affinity and anti-affinity

`nodeSelector` is the simplest way to constrain Pods to nodes with specific labels. Affinity and anti-affinity expand the types of constraints you can define. Some of the benefits of affinity and anti-affinity include:

- The affinity/anti-affinity language is more expressive. `nodeSelector` only selects nodes with all the specified labels. Affinity/anti-affinity gives you more control over the selection logic.
- You can indicate that a rule is *soft* or *preferred*, so that the scheduler still schedules the Pod even if it can't find a matching node.
- You can constrain a Pod using labels on other Pods running on the node (or other topological domain), instead of just node labels, which allows you to define rules for which Pods can be co-located on a node.

The affinity feature consists of two types of affinity:

- *Node affinity* functions like the `nodeSelector` field but is more expressive and allows you to specify soft rules.
- *Inter-pod affinity/anti-affinity* allows you to constrain Pods against labels on other Pods.

Node affinity

Node affinity is conceptually similar to `nodeSelector`, allowing you to constrain which nodes your Pod can be scheduled on based on node labels. There are two types of node affinity:

- `requiredDuringSchedulingIgnoredDuringExecution`: The scheduler can't schedule the Pod unless the rule is met. This functions like `nodeSelector`, but with a more expressive syntax.
- `preferredDuringSchedulingIgnoredDuringExecution`: The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the Pod.

Note:

In the preceding types, `IgnoredDuringExecution` means that if the node labels change after Kubernetes schedules the Pod, the Pod continues to run.

You can specify node affinities using the `.spec.affinity.nodeAffinity` field in your Pod spec.

For example, consider the following Pod spec:

pods/pod-with-node-affinity.yaml Copy pods/pod-with-node-affinity.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: with-node-affinityspec:  affinity:    nodeAffinity:      requiredDuringSchedulingIgnoredDuringExecution:
```

In this example, the following rules apply:

- The node *must* have a label with the key `topology.kubernetes.io/zone` and the value of that label *must* be either `antarctica-east1` or `antarctica-west1`.
- The node *preferably* has a label with the key `another-node-label-key` and the value `another-node-label-value`.

You can use the `operator` field to specify a logical operator for Kubernetes to use when interpreting the rules. You can use `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt` and `Lt`.

Read Operators to learn more about how these work.

`NotIn` and `DoesNotExist` allow you to define node anti-affinity behavior. Alternatively, you can use node taints to repel Pods from specific nodes.

Note:

If you specify both `nodeSelector` and `nodeAffinity`, *both* must be satisfied for the Pod to be scheduled onto a node.

If you specify multiple terms in `nodeSelectorTerms` associated with `nodeAffinity` types, then the Pod can be scheduled onto a node if one of the specified terms can be satisfied (terms are ORed).

If you specify multiple expressions in a single `matchExpressions` field associated with a term in `nodeSelectorTerms`, then the Pod can be scheduled onto a node only if all the expressions are satisfied (expressions are ANDed).

See Assign Pods to Nodes using Node Affinity for more information.

Node affinity weight

You can specify a `weight` between 1 and 100 for each instance of the `preferredDuringSchedulingIgnoredDuringExecution` affinity type. When the scheduler finds nodes that meet all the other scheduling requirements of the Pod, the scheduler iterates through every preferred rule that the node satisfies and adds the value of the `weight` for that expression to a sum.

The final sum is added to the score of other priority functions for the node. Nodes with the highest total score are prioritized when the scheduler makes a scheduling decision for the Pod.

For example, consider the following Pod spec:

pods/pod-with-affinity-preferred-weight.yaml Copy pods/pod-with-affinity-preferred-weight.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: with-affinity-preferred-weightspec:  affinity:    nodeAffinity:      requiredDuringSchedulingIgnoredDuri
```

If there are two possible nodes that match the `preferredDuringSchedulingIgnoredDuringExecution` rule, one with the `label-1:key-1` label and another with the `label-2:key-2` label, the scheduler considers the `weight` of each node and adds the weight to the other scores for that node, and schedules the Pod onto the node with the highest final score.

Note:

If you want Kubernetes to successfully schedule the Pods in this example, you must have existing nodes with the `kubernetes.io/os=linux` label.

Node affinity per scheduling profile

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

When configuring multiple scheduling profiles, you can associate a profile with a node affinity, which is useful if a profile only applies to a specific set of nodes. To do so, add an `addedAffinity` to the `args` field of the `NodeAffinity` plugin in the scheduler configuration. For example:

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfigurationprofiles:  - schedulerName: default-scheduler  - schedulerName: foo-scheduler    pluginConfig:
```

The `addedAffinity` is applied to all Pods that set `.spec.schedulerName` to `foo-scheduler`, in addition to the NodeAffinity specified in the PodSpec. That is, in order to match the Pod, nodes need to satisfy `addedAffinity` and the Pod's `.spec.NodeAffinity`.

Since the `addedAffinity` is not visible to end users, its behavior might be unexpected to them. Use node labels that have a clear correlation to the scheduler profile name.

Note:

The DaemonSet controller, which creates Pods for DaemonSets, does not support scheduling profiles. When the DaemonSet controller creates Pods, the default Kubernetes scheduler places those Pods and honors any `nodeAffinity` rules in the DaemonSet controller.

Inter-pod affinity and anti-affinity

Inter-pod affinity and anti-affinity allow you to constrain which nodes your Pods can be scheduled on based on the labels of Pods already running on that node, instead of the node labels.

Types of Inter-pod Affinity and Anti-affinity

Inter-pod affinity and anti-affinity take the form "this Pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more Pods that meet rule Y", where X is a topology domain like node, rack, cloud provider zone or region, or similar and Y is the rule Kubernetes tries to satisfy.

You express these rules (Y) as label selectors with an optional associated list of namespaces. Pods are namespaced objects in Kubernetes, so Pod labels also implicitly have namespaces. Any label selectors for Pod labels should specify the namespaces in which Kubernetes should look for those labels.

You express the topology domain (X) using a `topologyKey`, which is the key for the node label that the system uses to denote the domain. For examples, see Well-Known Labels, Annotations and Taints.

Note:

Inter-pod affinity and anti-affinity require substantial amounts of processing which can slow down scheduling in large clusters significantly. We do not recommend using them in clusters larger than several hundred nodes.

Note:

Pod anti-affinity requires nodes to be consistently labeled, in other words, every node in the cluster must have an appropriate label matching `topologyKey`. If some or all nodes are missing the specified `topologyKey` label, it can lead to unintended behavior.

Similar to node affinity are two types of Pod affinity and anti-affinity as follows:

- `requiredDuringSchedulingIgnoredDuringExecution`
- `preferredDuringSchedulingIgnoredDuringExecution`

For example, you could use `requiredDuringSchedulingIgnoredDuringExecution` affinity to tell the scheduler to co-locate Pods of two services in the same cloud provider zone because they communicate with each other a lot. Similarly, you could use `preferredDuringSchedulingIgnoredDuringExecution` anti-affinity to spread Pods from a service across multiple cloud provider zones.

To use inter-pod affinity, use the `affinity.podAffinity` field in the Pod spec. For inter-pod anti-affinity, use the `affinity.podAntiAffinity` field in the Pod spec.

Scheduling Behavior

When scheduling a new Pod, the Kubernetes scheduler evaluates the Pod's affinity/anti-affinity rules in the context of the current cluster state:

1. Hard Constraints (Node Filtering):
    - `podAffinity.requiredDuringSchedulingIgnoredDuringExecution` and `podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution`:
        - The scheduler ensures the new Pod is assigned to nodes that satisfy these required affinity and anti-affinity rules based on existing Pods.
2. Soft Constraints (Scoring):
    - `podAffinity.preferredDuringSchedulingIgnoredDuringExecution` and `podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution`:
        - The scheduler scores nodes based on how well they meet these preferred affinity and anti-affinity rules to optimize Pod placement.
3. Ignored Fields:
    - Existing Pods' `podAffinity.preferredDuringSchedulingIgnoredDuringExecution`:
        - These preferred affinity rules are not considered during the scheduling decision for new Pods.
    - Existing Pods' `podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution`:
        - Similarly, preferred anti-affinity rules of existing Pods are ignored during scheduling.

Scheduling a Group of Pods with Inter-pod Affinity to Themselves

If the current Pod being scheduled is the first in a series that have affinity to themselves, it is allowed to be scheduled if it passes all other affinity checks. This is determined by verifying that no other Pod in the cluster matches the namespace and selector of this Pod, that the Pod matches its own terms, and the chosen node matches all requested topologies. This ensures that there will not be a deadlock even if all the Pods have inter-pod affinity specified.

Pod Affinity Example

Consider the following Pod spec:

`pods/pod-with-pod-affinity.yaml` Copy pods/pod-with-pod-affinity.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: with-pod-affinityspec:  affinity:    podAffinity:      requiredDuringSchedulingIgnoredDuringExecution:
```

This example defines one Pod affinity rule and one Pod anti-affinity rule. The Pod affinity rule uses the "hard" `requiredDuringSchedulingIgnoredDuringExecution`, while the anti-affinity rule uses the "soft" `preferredDuringSchedulingIgnoredDuringExecution`.

The affinity rule specifies that the scheduler is allowed to place the example Pod on a node only if that node belongs to a specific zone where other Pods have been labeled with `security=S1`. For instance, if we have a cluster with a designated zone, let's call it "Zone V," consisting of nodes labeled with `topology.kubernetes.io/zone=V`, the scheduler can assign the Pod to any node within Zone V, as long as there is at least one Pod within Zone V already labeled with `security=S1`. Conversely, if there are no Pods with `security=S1` labels in Zone V, the scheduler will not assign the example Pod to any node in that zone.

The anti-affinity rule specifies that the scheduler should try to avoid scheduling the Pod on a node if that node belongs to a specific zone where other Pods have been labeled with `security=S2`. For instance, if we have a cluster with a designated zone, let's call it "Zone R," consisting of nodes labeled with topology.kubernetes.io/zone=R, the scheduler should avoid assigning the Pod to any node within Zone R, as long as there is at least one Pod within Zone R already labeled with security=S2. Conversely, the anti-affinity rule does not impact scheduling into Zone R if there are no Pods with security=S2 labels.

To get yourself more familiar with the examples of Pod affinity and anti-affinity, refer to the design proposal.

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

Read Operators to learn more about how these work.

In principle, the topologyKey can be any allowed label key with the following exceptions for performance and security reasons:

- For Pod affinity and anti-affinity, an empty topologyKey field is not allowed in both requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution.
- For requiredDuringSchedulingIgnoredDuringExecution Pod anti-affinity rules, the admission controller LimitPodHardAntiAffinityTopology limits topologyKey to kubernetes.io/hostname. You can modify or disable the admission controller if you want to allow custom topologies.

In addition to labelSelector and topologyKey, you can optionally specify a list of namespaces which the labelSelector should match against using the namespaces field at the same level as labelSelector and topologyKey. If omitted or empty, namespaces defaults to the namespace of the Pod where the affinity/anti-affinity definition appears.

Namespace Selector

FEATURE STATE: Kubernetes v1.24 [stable]

You can also select matching namespaces using namespaceSelector, which is a label query over the set of namespaces. The affinity term is applied to namespaces selected by both namespaceSelector and the namespaces field. Note that an empty namespaceSelector ({}) matches all namespaces, while a null or empty namespaces list and null namespaceSelector matches the namespace of the Pod where the rule is defined.

matchLabelKeys

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

Note:

The matchLabelKeys field is a beta-level field and is enabled by default in Kubernetes 1.34. When you want to disable it, you have to disable it explicitly via the MatchLabelKeysInPodAffinity feature gate.

Kubernetes includes an optional matchLabelKeys field for Pod affinity or anti-affinity. The field specifies keys for the labels that should match with the incoming Pod's labels, when satisfying the Pod (anti)affinity.

The keys are used to look up values from the Pod labels; those key-value labels are combined (using AND) with the match restrictions defined using the labelSelector field. The combined filtering selects the set of existing Pods that will be taken into Pod (anti)affinity calculation.

Caution:

It's not recommended to use matchLabelKeys with labels that might be updated directly on pods. Even if you edit the pod's label that is specified at matchLabelKeys **directly**, (that is, not via a deployment), kube-apiserver doesn't reflect the label update onto the merged labelSelector.

A common use case is to use matchLabelKeys with pod-template-hash (set on Pods managed as part of a Deployment, where the value is unique for each revision). Using pod-template-hash in matchLabelKeys allows you to target the Pods that belong to the same revision as the incoming Pod, so that a rolling upgrade won't break affinity.

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: application-server...spec:  template:    spec:      affinity:        podAffinity:          requir
```

mismatchLabelKeys

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

Note:

The mismatchLabelKeys field is a beta-level field and is enabled by default in Kubernetes 1.34. When you want to disable it, you have to disable it explicitly via the MatchLabelKeysInPodAffinity feature gate.

Kubernetes includes an optional mismatchLabelKeys field for Pod affinity or anti-affinity. The field specifies keys for the labels that should not match with the incoming Pod's labels, when satisfying the Pod (anti)affinity.

Caution:

It's not recommended to use mismatchLabelKeys with labels that might be updated directly on pods. Even if you edit the pod's label that is specified at mismatchLabelKeys **directly**, (that is, not via a deployment), kube-apiserver doesn't reflect the label update onto the merged labelSelector.

One example use case is to ensure Pods go to the topology domain (node, zone, etc) where only Pods from the same tenant or team are scheduled in. In other words, you want to avoid running Pods from two different tenants on the same topology domain at the same time.

```
apiVersion: v1
kind: Podmetadata:  labels:    # Assume that all relevant Pods have a "tenant" label set    tenant: tenant-a...spec:  affinity:
```

More practical use-cases

Inter-pod affinity and anti-affinity can be even more useful when they are used with higher level collections such as ReplicaSets, StatefulSets, Deployments, etc. These rules allow you to configure that a set of workloads should be co-located in the same defined topology; for example, preferring to place two related Pods onto the same node.

For example: imagine a three-node cluster. You use the cluster to run a web application and also an in-memory cache (such as Redis). For this example, also assume that latency between the web application and the memory cache should be as low as is practical. You could use inter-pod affinity and anti-affinity to co-locate the web servers with the cache as much as possible.

In the following example Deployment for the Redis cache, the replicas get the label `app=store`. The `podAntiAffinity` rule tells the scheduler to avoid placing multiple replicas with the `app=store` label on a single node. This creates each cache in a separate node.

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: redis-cachespec:  selector:    matchLabels:      app: store  replicas: 3  template:    metadata:
```

The following example Deployment for the web servers creates replicas with the label `app=web-store`. The Pod affinity rule tells the scheduler to place each replica on a node that has a Pod with the label `app=store`. The Pod anti-affinity rule tells the scheduler never to place multiple `app=web-store` servers on a single node.

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: web-serverspec:  selector:    matchLabels:      app: web-store  replicas: 3  template:    metadat
```

Creating the two preceding Deployments results in the following cluster layout, where each web server is co-located with a cache, on three separate nodes.

node-1	node-2	node-3
webserver-1	*webserver-2*	*webserver-3*
cache-1	*cache-2*	*cache-3*

The overall effect is that each cache instance is likely to be accessed by a single client that is running on the same node. This approach aims to minimize both skew (imbalanced load) and latency.

You might have other reasons to use Pod anti-affinity. See the ZooKeeper tutorial for an example of a StatefulSet configured with anti-affinity for high availability, using the same technique as this example.

nodeName

`nodeName` is a more direct form of node selection than affinity or `nodeSelector`. `nodeName` is a field in the Pod spec. If the `nodeName` field is not empty, the scheduler ignores the Pod and the kubelet on the named node tries to place the Pod on that node. Using `nodeName` overrules using `nodeSelector` or affinity and anti-affinity rules.

Some of the limitations of using `nodeName` to select nodes are:

- If the named node does not exist, the Pod will not run, and in some cases may be automatically deleted.
- If the named node does not have the resources to accommodate the Pod, the Pod will fail and its reason will indicate why, for example OutOfmemory or OutOfcpu.
- Node names in cloud environments are not always predictable or stable.

Warning:

`nodeName` is intended for use by custom schedulers or advanced use cases where you need to bypass any configured schedulers. Bypassing the schedulers might lead to failed Pods if the assigned Nodes get oversubscribed. You can use node affinity or the `nodeSelector` field to assign a Pod to a specific Node without bypassing the schedulers.

Here is an example of a Pod spec using the `nodeName` field:

```
apiVersion: v1
kind: Podmetadata:  name: nginxspec:  containers:  - name: nginx    image: nginx  nodeName: kube-01
```

The above Pod will only run on the node `kube-01`.

nominatedNodeName

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

`nominatedNodeName` can be used for external components to nominate node for a pending pod. This nomination is best effort: it might be ignored if the scheduler determines the pod cannot go to a nominated node.

Also, this field can be (over)written by the scheduler:

- If the scheduler finds a node to nominate via the preemption.
- If the scheduler decides where the pod is going, and move it to the binding cycle.
    - Note that, in this case, `nominatedNodeName` is put only when the pod has to go through `WaitOnPermit` or `PreBind` extension points.

Here is an example of a Pod status using the `nominatedNodeName` field:

```
apiVersion: v1
kind: Podmetadata:  name: nginx...status:  nominatedNodeName: kube-01
```

Pod topology spread constraints

You can use *topology spread constraints* to control how Pods are spread across your cluster among failure-domains such as regions, zones, nodes, or among any other topology domains that you define. You might do this to improve performance, expected availability, or overall utilization.

Read Pod topology spread constraints to learn more about how these work.

Operators

The following are all the logical operators that you can use in the `operator` field for `nodeAffinity` and `podAffinity` mentioned above.

Operator	Behavior
`In`	The label value is present in the supplied set of strings
`NotIn`	The label value is not contained in the supplied set of strings
`Exists`	A label with this key exists on the object
`DoesNotExist`	No label with this key exists on the object

The following operators can only be used with `nodeAffinity`.

Operator	Behavior
`Gt`	The field value will be parsed as an integer, and that integer is less than the integer that results from parsing the value of a label named by this selector
`Lt`	The field value will be parsed as an integer, and that integer is greater than the integer that results from parsing the value of a label named by this selector

Note:

`Gt` and `Lt` operators will not work with non-integer values. If the given value doesn't parse as an integer, the Pod will fail to get scheduled. Also, `Gt` and `Lt` are not available for `podAffinity`.

What's next

- Read more about taints and tolerations.
- Read the design docs for node affinity and for inter-pod affinity/anti-affinity.
- Learn about how the topology manager takes part in node-level resource allocation decisions.
- Learn how to use nodeSelector.
- Learn how to use affinity and anti-affinity.

Hardening Guide - Scheduler Configuration

Information about how to make the Kubernetes scheduler more secure.

The Kubernetes scheduler is one of the critical components of the control plane.

This document covers how to improve the security posture of the Scheduler.

A misconfigured scheduler can have security implications. Such a scheduler can target specific nodes and evict the workloads or applications that are sharing the node and its resources. This can aid an attacker with a Yo-Yo attack: an attack on a vulnerable autoscaler.

kube-scheduler configuration

Scheduler authentication & authorization command line options

When setting up authentication configuration, it should be made sure that kube-scheduler's authentication remains consistent with kube-api-server's authentication. If any request has missing authentication headers, the authentication should happen through the kube-api-server allowing all authentication to be consistent in the cluster.

- `authentication-kubeconfig`: Make sure to provide a proper kubeconfig so that the scheduler can retrieve authentication configuration options from the API Server. This kubeconfig file should be protected with strict file permissions.
- `authentication-tolerate-lookup-failure`: Set this to `false` to make sure the scheduler *always* looks up its authentication configuration from the API server.
- `authentication-skip-lookup`: Set this to `false` to make sure the scheduler *always* looks up its authentication configuration from the API server.
- `authorization-always-allow-paths`: These paths should respond with data that is appropriate for anonymous authorization. Defaults to `/healthz,/readyz,/livez`.
- `profiling`: Set to `false` to disable the profiling endpoints which are provide debugging information but which should not be enabled on production clusters as they present a risk of denial of service or information leakage. The `--profiling` argument is deprecated and can now be provided through the KubeScheduler DebuggingConfiguration. Profiling can be disabled through the kube-scheduler config by setting `enableProfiling` to `false`.
- `requestheader-client-ca-file`: Avoid passing this argument.

Scheduler networking command line options

- `bind-address`: In most cases, the kube-scheduler does not need to be externally accessible. Setting the bind address to `localhost` is a secure practice.
- `permit-address-sharing`: Set this to `false` to disable connection sharing through `SO_REUSEADDR`. `SO_REUSEADDR` can lead to reuse of terminated connections that are in `TIME_WAIT` state.
- `permit-port-sharing`: Default `false`. Use the default unless you are confident you understand the security implications.

Scheduler TLS command line options

- `tls-cipher-suites`: Always provide a list of preferred cipher suites. This ensures encryption never happens with insecure cipher suites.

Scheduling configurations for custom schedulers

When using custom schedulers based on the Kubernetes scheduling code, cluster administrators need to be careful with plugins that use the `queueSort`, `prefilter`, `filter`, or `permit` extension points. These extension points control various stages of a scheduling process, and the wrong configuration can impact the kube-scheduler's behavior in your cluster.

Key considerations

- Exactly one plugin that uses the `queueSort` extension point can be enabled at a time. Any plugins that use `queueSort` should be scrutinized.
- Plugins that implement the `prefilter` or `filter` extension point can potentially mark all nodes as unschedulable. This can bring scheduling of new pods to a halt.
- Plugins that implement the `permit` extension point can prevent or delay the binding of a Pod. Such plugins should be thoroughly reviewed by the cluster administrator.

When using a plugin that is not one of the default plugins, consider disabling the `queueSort`, `filter` and `permit` extension points as follows:

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfigurationprofiles:  - schedulerName: my-scheduler    plugins:      # Disable specific plugins for different
```

This creates a scheduler profile `my-custom-scheduler`. Whenever the `.spec` of a Pod does not have a value for `.spec.schedulerName`, the kube-scheduler runs for that Pod, using its main configuration, and default plugins. If you define a Pod with `.spec.schedulerName` set to `my-custom-scheduler`, the kube-scheduler runs but with a custom configuration; in that custom configuration, the `queueSort`, `filter` and `permit` extension points are disabled. If you use this KubeSchedulerConfiguration, and don't run any custom scheduler, and you then define a Pod with `.spec.schedulerName` set to `nonexistent-scheduler` (or any other scheduler name that doesn't exist in your cluster), no events would be generated for a pod.

Disallow labeling nodes

A cluster administrator should ensure that cluster users cannot label the nodes. A malicious actor can use `nodeSelector` to schedule workloads on nodes where those workloads should not be present.

Security Checklist

Baseline checklist for ensuring security in Kubernetes clusters.

This checklist aims at providing a basic list of guidance with links to more comprehensive documentation on each topic. It does not claim to be exhaustive and is meant to evolve.

On how to read and use this document:

- The order of topics does not reflect an order of priority.
- Some checklist items are detailed in the paragraph below the list of each section.

Caution:

Checklists are **not** sufficient for attaining a good security posture on their own. A good security posture requires constant attention and improvement, but a checklist can be the first step on the never-ending journey towards security preparedness. Some of the recommendations in this checklist may be too restrictive or too lax for your specific security needs. Since Kubernetes security is not "one size fits all", each category of checklist items should be evaluated on its merits.

Authentication & Authorization

- □ `system:masters` group is not used for user or component authentication after bootstrapping.
- □ The kube-controller-manager is running with `--use-service-account-credentials` enabled.
- □ The root certificate is protected (either an offline CA, or a managed online CA with effective access controls).
- □ Intermediate and leaf certificates have an expiry date no more than 3 years in the future.
- □ A process exists for periodic access review, and reviews occur no more than 24 months apart.
- □ The Role Based Access Control Good Practices are followed for guidance related to authentication and authorization.

After bootstrapping, neither users nor components should authenticate to the Kubernetes API as `system:masters`. Similarly, running all of kube-controller-manager as `system:masters` should be avoided. In fact, `system:masters` should only be used as a break-glass mechanism, as opposed to an admin user.

Network security

- □ CNI plugins in use support network policies.
- □ Ingress and egress network policies are applied to all workloads in the cluster.
- □ Default network policies within each namespace, selecting all pods, denying everything, are in place.
- □ If appropriate, a service mesh is used to encrypt all communications inside of the cluster.
- □ The Kubernetes API, kubelet API and etcd are not exposed publicly on Internet.
- □ Access from the workloads to the cloud metadata API is filtered.
- □ Use of LoadBalancer and ExternalIPs is restricted.

A number of Container Network Interface (CNI) plugins plugins provide the functionality to restrict network resources that pods may communicate with. This is most commonly done through Network Policies which provide a namespaced resource to define rules. Default network policies that block all egress and ingress, in each namespace, selecting all pods, can be useful to adopt an allow list approach to ensure that no workloads are missed.

Not all CNI plugins provide encryption in transit. If the chosen plugin lacks this feature, an alternative solution could be to use a service mesh to provide that functionality.

The etcd datastore of the control plane should have controls to limit access and not be publicly exposed on the Internet. Furthermore, mutual TLS (mTLS) should be used to communicate securely with it. The certificate authority for this should be unique to etcd.

External Internet access to the Kubernetes API server should be restricted to not expose the API publicly. Be careful, as many managed Kubernetes distributions are publicly exposing the API server by default. You can then use a bastion host to access the server.

The kubelet API access should be restricted and not exposed publicly, the default authentication and authorization settings, when no configuration file specified with the `--config` flag, are overly permissive.

If a cloud provider is used for hosting Kubernetes, the access from pods to the cloud metadata API `169.254.169.254` should also be restricted or blocked if not needed because it may leak information.

For restricted LoadBalancer and ExternalIPs use, see CVE-2020-8554: Man in the middle using LoadBalancer or ExternalIPs and the DenyServiceExternalIPs admission controller for further information.

Pod security

- □ RBAC rights to `create`, `update`, `patch`, `delete` workloads is only granted if necessary.
- □ Appropriate Pod Security Standards policy is applied for all namespaces and enforced.
- □ Memory limit is set for the workloads with a limit equal or inferior to the request.
- □ CPU limit might be set on sensitive workloads.
- □ For nodes that support it, Seccomp is enabled with appropriate syscalls profile for programs.
- □ For nodes that support it, AppArmor or SELinux is enabled with appropriate profile for programs.

RBAC authorization is crucial but cannot be granular enough to have authorization on the Pods' resources (or on any resource that manages Pods). The only granularity is the API verbs on the resource itself, for example, `create` on Pods. Without additional admission, the authorization to create these resources allows direct unrestricted access to the schedulable nodes of a cluster.

The Pod Security Standards define three different policies, privileged, baseline and restricted that limit how fields can be set in the `PodSpec` regarding security. These standards can be enforced at the namespace level with the new Pod Security admission, enabled by default, or by third-party admission webhook. Please note that, contrary to the removed PodSecurityPolicy admission it replaces, Pod Security admission can be easily combined with admission webhooks and external services.

Pod Security admission `restricted` policy, the most restrictive policy of the Pod Security Standards set, can operate in several modes, `warn`, `audit` or `enforce` to gradually apply the most appropriate security context according to security best practices. Nevertheless, pods' security context should be separately investigated to limit the privileges and access pods may have on top of the predefined security standards, for specific use cases.

For a hands-on tutorial on Pod Security, see the blog post Kubernetes 1.23: Pod Security Graduates to Beta.

Memory and CPU limits should be set in order to restrict the memory and CPU resources a pod can consume on a node, and therefore prevent potential DoS attacks from malicious or breached workloads. Such policy can be enforced by an admission controller. Please note that CPU limits will throttle usage and thus can have unintended effects on auto-scaling features or efficiency i.e. running the process in best effort with the CPU resource available.

Caution:

Memory limit superior to request can expose the whole node to OOM issues.

Enabling Seccomp

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 can improve the security of your workloads by reducing the Linux kernel syscall attack surface available inside containers. The seccomp filter mode leverages BPF to create an allow or deny list of specific syscalls, named profiles.

Since Kubernetes 1.27, you can enable the use of `RuntimeDefault` as the default seccomp profile for all workloads. A security tutorial is available on this topic. In addition, the Kubernetes Security Profiles Operator is a project that facilitates the management and use of seccomp in clusters.

Note:

Seccomp is only available on Linux nodes.

Enabling AppArmor or SELinux

AppArmor

AppArmor is a Linux kernel security module that can provide an easy way to implement Mandatory Access Control (MAC) and better auditing through system logs. A default AppArmor profile is enforced on nodes that support it, or a custom profile can be configured. Like seccomp, AppArmor is also configured through profiles, where each profile is either running in enforcing mode, which blocks access to disallowed resources or complain mode, which only reports violations. AppArmor profiles are enforced on a per-container basis, with an annotation, allowing for processes to gain just the right privileges.

Note:

AppArmor is only available on Linux nodes, and enabled in some Linux distributions.

SELinux

SELinux is also a Linux kernel security module that can provide a mechanism for supporting access control security policies, including Mandatory Access Controls (MAC). SELinux labels can be assigned to containers or pods via their `securityContext` section.

Note:

SELinux is only available on Linux nodes, and enabled in some Linux distributions.

Logs and auditing

- □ Audit logs, if enabled, are protected from general access.

Pod placement

- □ Pod placement is done in accordance with the tiers of sensitivity of the application.
- □ Sensitive applications are running isolated on nodes or with specific sandboxed runtimes.

Pods that are on different tiers of sensitivity, for example, an application pod and the Kubernetes API server, should be deployed onto separate nodes. The purpose of node isolation is to prevent an application container breakout to directly providing access to applications with higher level of sensitivity to easily pivot within the cluster. This separation should be enforced to prevent pods accidentally being deployed onto the same node. This could be enforced with the following features:

Node Selectors
: Key-value pairs, as part of the pod specification, that specify which nodes to deploy onto. These can be enforced at the namespace and cluster level with the PodNodeSelector admission controller.

PodTolerationRestriction
: An admission controller that allows administrators to restrict permitted tolerations within a namespace. Pods within a namespace may only utilize the tolerations specified on the namespace object annotation keys that provide a set of default and allowed tolerations.

RuntimeClass
: RuntimeClass is a feature for selecting the container runtime configuration. The container runtime configuration is used to run a Pod's containers and can provide more or less isolation from the host at the cost of performance overhead.

Secrets

- □ ConfigMaps are not used to hold confidential data.
- □ Encryption at rest is configured for the Secret API.
- □ If appropriate, a mechanism to inject secrets stored in third-party storage is deployed and available.
- □ Service account tokens are not mounted in pods that don't require them.
- □ Bound service account token volume is in-use instead of non-expiring tokens.

Secrets required for pods should be stored within Kubernetes Secrets as opposed to alternatives such as ConfigMap. Secret resources stored within etcd should be encrypted at rest.

Pods needing secrets should have these automatically mounted through volumes, preferably stored in memory like with the `emptyDir.medium` option. Mechanism can be used to also inject secrets from third-party storages as volume, like the Secrets Store CSI Driver. This should be done preferentially as compared to providing the pods service account RBAC access to secrets. This would allow adding secrets into the pod as environment variables or files. Please note that the environment variable method might be more prone to leakage due to crash dumps in logs and the non-confidential nature of environment variable in Linux, as opposed to the permission mechanism on files.

Service account tokens should not be mounted into pods that do not require them. This can be configured by setting `automountServiceAccountToken` to `false` either within the service account to apply throughout the namespace or specifically for a pod. For Kubernetes v1.22 and above, use Bound Service Accounts for time-bound service account credentials.

Images

- □ Minimize unnecessary content in container images.
- □ Container images are configured to be run as unprivileged user.
- □ References to container images are made by sha256 digests (rather than tags) or the provenance of the image is validated by verifying the image's digital signature at deploy time via admission control.
- □ Container images are regularly scanned during creation and in deployment, and known vulnerable software is patched.

Container image should contain the bare minimum to run the program they package. Preferably, only the program and its dependencies, building the image from the minimal possible base. In particular, image used in production should not contain shells or debugging utilities, as an ephemeral debug container can be used for troubleshooting.

Build images to directly start with an unprivileged user by using the `USER` instruction in Dockerfile. The Security Context allows a container image to be started with a specific user and group with `runAsUser` and `runAsGroup`, even if not specified in the image manifest. However, the file permissions in the image layers might make it impossible to just start the process with a new unprivileged user without image modification.

Avoid using image tags to reference an image, especially the `latest` tag, the image behind a tag can be easily modified in a registry. Prefer using the complete `sha256` digest which is unique to the image manifest. This policy can be enforced via an ImagePolicyWebhook. Image signatures can also be automatically verified with an admission controller at deploy time to validate their authenticity and integrity.

Scanning a container image can prevent critical vulnerabilities from being deployed to the cluster alongside the container image. Image scanning should be completed before deploying a container image to a cluster and is usually done as part of the deployment process in a CI/CD pipeline. The purpose of an image scan is to obtain information about possible vulnerabilities and their prevention in the container image, such as a Common Vulnerability Scoring

System (CVSS) score. If the result of the image scans is combined with the pipeline compliance rules, only properly patched container images will end up in Production.

Admission controllers

- □ An appropriate selection of admission controllers is enabled.
- □ A pod security policy is enforced by the Pod Security Admission or/and a webhook admission controller.
- □ The admission chain plugins and webhooks are securely configured.

Admission controllers can help improve the security of the cluster. However, they can present risks themselves as they extend the API server and should be properly secured.

The following lists present a number of admission controllers that could be considered to enhance the security posture of your cluster and application. It includes controllers that may be referenced in other parts of this document.

This first group of admission controllers includes plugins enabled by default, consider to leave them enabled unless you know what you are doing:

`CertificateApproval`
Performs additional authorization checks to ensure the approving user has permission to approve certificate request.

`CertificateSigning`
Performs additional authorization checks to ensure the signing user has permission to sign certificate requests.

`CertificateSubjectRestriction`
Rejects any certificate request that specifies a 'group' (or 'organization attribute') of `system:masters`.

`LimitRanger`
Enforces the LimitRange API constraints.

`MutatingAdmissionWebhook`
Allows the use of custom controllers through webhooks, these controllers may mutate requests that they review.

`PodSecurity`
Replacement for Pod Security Policy, restricts security contexts of deployed Pods.

`ResourceQuota`
Enforces resource quotas to prevent over-usage of resources.

`ValidatingAdmissionWebhook`
Allows the use of custom controllers through webhooks, these controllers do not mutate requests that it reviews.

The second group includes plugins that are not enabled by default but are in general availability state and are recommended to improve your security posture:

`DenyServiceExternalIPs`
Rejects all net-new usage of the `Service.spec.externalIPs` field. This is a mitigation for CVE-2020-8554: Man in the middle using LoadBalancer or ExternalIPs.

`NodeRestriction`
Restricts kubelet's permissions to only modify the pods API resources they own or the node API resource that represent themselves. It also prevents kubelet from using the `node-restriction.kubernetes.io/` annotation, which can be used by an attacker with access to the kubelet's credentials to influence pod placement to the controlled node.

The third group includes plugins that are not enabled by default but could be considered for certain use cases:

`AlwaysPullImages`
Enforces the usage of the latest version of a tagged image and ensures that the deployer has permissions to use the image.

`ImagePolicyWebhook`
Allows enforcing additional controls for images through webhooks.

What's next

- Privilege escalation via Pod creation warns you about a specific access control risk; check how you're managing that threat.
  - If you use Kubernetes RBAC, read RBAC Good Practices for further information on authorization.
- Securing a Cluster for information on protecting a cluster from accidental or malicious access.
- Cluster Multi-tenancy guide for configuration options recommendations and best practices on multi-tenancy.
- Blog post "A Closer Look at NSA/CISA Kubernetes Hardening Guidance" for complementary resource on hardening Kubernetes clusters.

Security For Linux Nodes

This page describes security considerations and best practices specific to the Linux operating system.

Protection for Secret data on nodes

On Linux nodes, memory-backed volumes (such as `secret` volume mounts, or `emptyDir` with `medium: Memory`) are implemented with a `tmpfs` filesystem.

If you have swap configured and use an older Linux kernel (or a current kernel and an unsupported configuration of Kubernetes), **memory** backed volumes can have data written to persistent storage.

The Linux kernel officially supports the `noswap` option from version 6.3, therefore it is recommended the used kernel version is 6.3 or later, or supports the `noswap` option via a backport, if swap is enabled on the node.

Read swap memory management for more info.

CSI Volume Cloning

This document describes the concept of cloning existing CSI Volumes in Kubernetes. Familiarity with Volumes is suggested.

Introduction

The CSI Volume Cloning feature adds support for specifying existing PVCs in the `dataSource` field to indicate a user would like to clone a Volume.

A Clone is defined as a duplicate of an existing Kubernetes Volume that can be consumed as any standard Volume would be. The only difference is that upon provisioning, rather than creating a "new" empty Volume, the back end device creates an exact duplicate of the specified Volume.

The implementation of cloning, from the perspective of the Kubernetes API, adds the ability to specify an existing PVC as a dataSource during new PVC creation. The source PVC must be bound and available (not in use).

Users need to be aware of the following when using this feature:

- Cloning support (`VolumePVCDataSource`) is only available for CSI drivers.
- Cloning support is only available for dynamic provisioners.
- CSI drivers may or may not have implemented the volume cloning functionality.
- You can only clone a PVC when it exists in the same namespace as the destination PVC (source and destination must be in the same namespace).
- Cloning is supported with a different Storage Class.
    - Destination volume can be the same or a different storage class as the source.
    - Default storage class can be used and storageClassName omitted in the spec.
- Cloning can only be performed between two volumes that use the same VolumeMode setting (if you request a block mode volume, the source MUST also be block mode)

Provisioning

Clones are provisioned like any other PVC with the exception of adding a dataSource that references an existing PVC in the same namespace.

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:    name: clone-of-pvc-1    namespace: mynsspec:  accessModes:  - ReadWriteOnce  storageClassN
```

Note:

You must specify a capacity value for `spec.resources.requests.storage`, and the value you specify must be the same or larger than the capacity of the source volume.

The result is a new PVC with the name `clone-of-pvc-1` that has the exact same content as the specified source `pvc-1`.

Usage

Upon availability of the new PVC, the cloned PVC is consumed the same as other PVC. It's also expected at this point that the newly created PVC is an independent object. It can be consumed, cloned, snapshotted, or deleted independently and without consideration for it's original dataSource PVC. This also implies that the source is not linked in any way to the newly created clone, it may also be modified or deleted without affecting the newly created clone.

Volume Snapshots

In Kubernetes, a *VolumeSnapshot* represents a snapshot of a volume on a storage system. This document assumes that you are already familiar with Kubernetes persistent volumes.

Introduction

Similar to how API resources `PersistentVolume` and `PersistentVolumeClaim` are used to provision volumes for users and administrators, `VolumeSnapshotContent` and `VolumeSnapshot` API resources are provided to create volume snapshots for users and administrators.

A `VolumeSnapshotContent` is a snapshot taken from a volume in the cluster that has been provisioned by an administrator. It is a resource in the cluster just like a PersistentVolume is a cluster resource.

A `VolumeSnapshot` is a request for snapshot of a volume by a user. It is similar to a PersistentVolumeClaim.

`VolumeSnapshotClass` allows you to specify different attributes belonging to a `VolumeSnapshot`. These attributes may differ among snapshots taken from the same volume on the storage system and therefore cannot be expressed by using the same `StorageClass` of a `PersistentVolumeClaim`.

Volume snapshots provide Kubernetes users with a standardized way to copy a volume's contents at a particular point in time without creating an entirely new volume. This functionality enables, for example, database administrators to backup databases before performing edit or delete modifications.

Users need to be aware of the following when using this feature:

- API Objects `VolumeSnapshot`, `VolumeSnapshotContent`, and `VolumeSnapshotClass` are CRDs, not part of the core API.
- `VolumeSnapshot` support is only available for CSI drivers.
- As part of the deployment process of `VolumeSnapshot`, the Kubernetes team provides a snapshot controller to be deployed into the control plane, and a sidecar helper container called csi-snapshotter to be deployed together with the CSI driver. The snapshot controller watches `VolumeSnapshot` and `VolumeSnapshotContent` objects and is responsible for the creation and deletion of `VolumeSnapshotContent` object. The sidecar csi-snapshotter watches `VolumeSnapshotContent` objects and triggers `CreateSnapshot` and `DeleteSnapshot` operations against a CSI endpoint.
- There is also a validating webhook server which provides tightened validation on snapshot objects. This should be installed by the Kubernetes distros along with the snapshot controller and CRDs, not CSI drivers. It should be installed in all Kubernetes clusters that has the snapshot feature enabled.
- CSI drivers may or may not have implemented the volume snapshot functionality. The CSI drivers that have provided support for volume snapshot will likely use the csi-snapshotter. See CSI Driver documentation for details.

- The CRDs and snapshot controller installations are the responsibility of the Kubernetes distribution.

For advanced use cases, such as creating group snapshots of multiple volumes, see the external CSI Volume Group Snapshot documentation.

Lifecycle of a volume snapshot and volume snapshot content

`VolumeSnapshotContents` are resources in the cluster. `VolumeSnapshots` are requests for those resources. The interaction between `VolumeSnapshotContents` and `VolumeSnapshots` follow this lifecycle:

Provisioning Volume Snapshot

There are two ways snapshots may be provisioned: pre-provisioned or dynamically provisioned.

Pre-provisioned

A cluster administrator creates a number of `VolumeSnapshotContents`. They carry the details of the real volume snapshot on the storage system which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

Dynamic

Instead of using a pre-existing snapshot, you can request that a snapshot to be dynamically taken from a PersistentVolumeClaim. The VolumeSnapshotClass specifies storage provider-specific parameters to use when taking a snapshot.

Binding

The snapshot controller handles the binding of a `VolumeSnapshot` object with an appropriate `VolumeSnapshotContent` object, in both pre-provisioned and dynamically provisioned scenarios. The binding is a one-to-one mapping.

In the case of pre-provisioned binding, the VolumeSnapshot will remain unbound until the requested VolumeSnapshotContent object is created.

Persistent Volume Claim as Snapshot Source Protection

The purpose of this protection is to ensure that in-use PersistentVolumeClaim API objects are not removed from the system while a snapshot is being taken from it (as this may result in data loss).

While a snapshot is being taken of a PersistentVolumeClaim, that PersistentVolumeClaim is in-use. If you delete a PersistentVolumeClaim API object in active use as a snapshot source, the PersistentVolumeClaim object is not removed immediately. Instead, removal of the PersistentVolumeClaim object is postponed until the snapshot is readyToUse or aborted.

Delete

Deletion is triggered by deleting the `VolumeSnapshot` object, and the `DeletionPolicy` will be followed. If the `DeletionPolicy` is `Delete`, then the underlying storage snapshot will be deleted along with the `VolumeSnapshotContent` object. If the `DeletionPolicy` is `Retain`, then both the underlying snapshot and `VolumeSnapshotContent` remain.

VolumeSnapshots

Each VolumeSnapshot contains a spec and a status.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotmetadata:  name: new-snapshot-testspec:  volumeSnapshotClassName: csi-hostpath-snapclass  source:    persisten
```

`persistentVolumeClaimName` is the name of the PersistentVolumeClaim data source for the snapshot. This field is required for dynamically provisioning a snapshot.

A volume snapshot can request a particular class by specifying the name of a VolumeSnapshotClass using the attribute `volumeSnapshotClassName`. If nothing is set, then the default class is used if available.

For pre-provisioned snapshots, you need to specify a `volumeSnapshotContentName` as the source for the snapshot as shown in the following example. The `volumeSnapshotContentName` source field is required for pre-provisioned snapshots.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotmetadata:  name: test-snapshotspec:  source:    volumeSnapshotContentName: test-content
```

Volume Snapshot Contents

Each VolumeSnapshotContent contains a spec and status. In dynamic provisioning, the snapshot common controller creates `VolumeSnapshotContent` objects. Here is an example:

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContentmetadata:  name: snapcontent-72d9a349-aacd-42d2-a240-d775650d2455spec:  deletionPolicy: Delete  driver:
```

`volumeHandle` is the unique identifier of the volume created on the storage backend and returned by the CSI driver during the volume creation. This field is required for dynamically provisioning a snapshot. It specifies the volume source of the snapshot.

For pre-provisioned snapshots, you (as cluster administrator) are responsible for creating the `VolumeSnapshotContent` object as follows.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContentmetadata:  name: new-snapshot-content-testspec:  deletionPolicy: Delete  driver: hostpath.csi.k8s.io  s
```

`snapshotHandle` is the unique identifier of the volume snapshot created on the storage backend. This field is required for the pre-provisioned snapshots. It specifies the CSI snapshot id on the storage system that this `VolumeSnapshotContent` represents.

`sourceVolumeMode` is the mode of the volume whose snapshot is taken. The value of the `sourceVolumeMode` field can be either `Filesystem` or `Block`. If the source volume mode is not specified, Kubernetes treats the snapshot as if the source volume's mode is unknown.

`volumeSnapshotRef` is the reference of the corresponding `VolumeSnapshot`. Note that when the `VolumeSnapshotContent` is being created as a pre-provisioned snapshot, the `VolumeSnapshot` referenced in `volumeSnapshotRef` might not exist yet.

Converting the volume mode of a Snapshot

If the `VolumeSnapshots` API installed on your cluster supports the `sourceVolumeMode` field, then the API has the capability to prevent unauthorized users from converting the mode of a volume.

To check if your cluster has capability for this feature, run the following command:

```
$ kubectl get crd volumesnapshotcontent -o yaml
```

If you want to allow users to create a `PersistentVolumeClaim` from an existing `VolumeSnapshot`, but with a different volume mode than the source, the annotation `snapshot.storage.kubernetes.io/allow-volume-mode-change: "true"`needs to be added to the `VolumeSnapshotContent` that corresponds to the `VolumeSnapshot`.

For pre-provisioned snapshots, `spec.sourceVolumeMode` needs to be populated by the cluster administrator.

An example `VolumeSnapshotContent` resource with this feature enabled would look like:

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContentmetadata:  name: new-snapshot-content-test  annotations:    - snapshot.storage.kubernetes.io/allow-volu
```

Provisioning Volumes from Snapshots

You can provision a new volume, pre-populated with data from a snapshot, by using the *dataSource* field in the `PersistentVolumeClaim` object.

For more details, see Volume Snapshot and Restore Volume from Snapshot.

Scheduling, Preemption and Eviction

In Kubernetes, scheduling refers to making sure that Pods are matched to Nodes so that the kubelet can run them. Preemption is the process of terminating Pods with lower Priority so that Pods with higher Priority can schedule on Nodes. Eviction is the process of terminating one or more Pods on Nodes.

Scheduling

- Kubernetes Scheduler
- Assigning Pods to Nodes
- Pod Overhead
- Pod Topology Spread Constraints
- Taints and Tolerations
- Scheduling Framework
- Dynamic Resource Allocation
- Scheduler Performance Tuning
- Resource Bin Packing for Extended Resources
- Pod Scheduling Readiness
- Descheduler

Pod Disruption

Pod disruption is the process by which Pods on Nodes are terminated either voluntarily or involuntarily.

Voluntary disruptions are started intentionally by application owners or cluster administrators. Involuntary disruptions are unintentional and can be triggered by unavoidable issues like Nodes running out of resources, or by accidental deletions.

- Pod Priority and Preemption
- Node-pressure Eviction
- API-initiated Eviction

Service ClusterIP allocation

In Kubernetes, Services are an abstract way to expose an application running on a set of Pods. Services can have a cluster-scoped virtual IP address (using a Service of `type: ClusterIP`). Clients can connect using that virtual IP address, and Kubernetes then load-balances traffic to that Service across the different backing Pods.

How Service ClusterIPs are allocated?

When Kubernetes needs to assign a virtual IP address for a Service, that assignment happens one of two ways:

dynamically
: the cluster's control plane automatically picks a free IP address from within the configured IP range for `type: ClusterIP` Services.

statically
: you specify an IP address of your choice, from within the configured IP range for Services.

Across your whole cluster, every Service `ClusterIP` must be unique. Trying to create a Service with a specific `ClusterIP` that has already been allocated will return an error.

Why do you need to reserve Service Cluster IPs?

Sometimes you may want to have Services running in well-known IP addresses, so other components and users in the cluster can use them.

The best example is the DNS Service for the cluster. As a soft convention, some Kubernetes installers assign the 10th IP address from the Service IP range to the DNS service. Assuming you configured your cluster with Service IP range 10.96.0.0/16 and you want your DNS Service IP to be 10.96.0.10, you'd have to create a Service like this:

```
apiVersion: v1
kind: Servicemetadata:  labels:    k8s-app: kube-dns    kubernetes.io/cluster-service: "true"    kubernetes.io/name: CoreDNS  name
```

But, as it was explained before, the IP address 10.96.0.10 has not been reserved. If other Services are created before or in parallel with dynamic allocation, there is a chance they can allocate this IP. Hence, you will not be able to create the DNS Service because it will fail with a conflict error.

How can you avoid Service ClusterIP conflicts?

The allocation strategy implemented in Kubernetes to allocate ClusterIPs to Services reduces the risk of collision.

The `ClusterIP` range is divided, based on the formula `min(max(16, cidrSize / 16), 256)`, described as *never less than 16 or more than 256 with a graduated step between them*.

Dynamic IP assignment uses the upper band by default, once this has been exhausted it will use the lower range. This will allow users to use static allocations on the lower band with a low risk of collision.

Examples

Example 1

This example uses the IP address range: 10.96.0.0/24 (CIDR notation) for the IP addresses of Services.

Range Size: 2^8 - 2 = 254
Band Offset: `min(max(16, 256/16), 256)` = `min(16, 256)` = 16
Static band start: 10.96.0.1
Static band end: 10.96.0.16
Range end: 10.96.0.254

> pie showData title 10.96.0.0/24 "Static" : 16 "Dynamic" : 238

Example 2

This example uses the IP address range: 10.96.0.0/20 (CIDR notation) for the IP addresses of Services.

Range Size: 2^{12} - 2 = 4094
Band Offset: `min(max(16, 4096/16), 256)` = `min(256, 256)` = 256
Static band start: 10.96.0.1
Static band end: 10.96.1.0
Range end: 10.96.15.254

> pie showData title 10.96.0.0/20 "Static" : 256 "Dynamic" : 3838

Example 3

This example uses the IP address range: 10.96.0.0/16 (CIDR notation) for the IP addresses of Services.

Range Size: 2^{16} - 2 = 65534
Band Offset: `min(max(16, 65536/16), 256)` = `min(4096, 256)` = 256
Static band start: 10.96.0.1
Static band ends: 10.96.1.0
Range end: 10.96.255.254

> pie showData title 10.96.0.0/16 "Static" : 256 "Dynamic" : 65278

What's next

- Read about Service External Traffic Policy
- Read about Connecting Applications with Services
- Read about Services

Scheduling Framework

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

The *scheduling framework* is a pluggable architecture for the Kubernetes scheduler. It consists of a set of "plugin" APIs that are compiled directly into the scheduler. These APIs allow most scheduling features to be implemented as plugins, while keeping the scheduling "core" lightweight and maintainable. Refer to the design proposal of the scheduling framework for more technical information on the design of the framework.

Framework workflow

The Scheduling Framework defines a few extension points. Scheduler plugins register to be invoked at one or more extension points. Some of these plugins can change the scheduling decisions and some are informational only.

Each attempt to schedule one Pod is split into two phases, the **scheduling cycle** and the **binding cycle**.

Scheduling cycle & binding cycle

The scheduling cycle selects a node for the Pod, and the binding cycle applies that decision to the cluster. Together, a scheduling cycle and binding cycle are referred to as a "scheduling context".

Scheduling cycles are run serially, while binding cycles may run concurrently.

A scheduling or binding cycle can be aborted if the Pod is determined to be unschedulable or if there is an internal error. The Pod will be returned to the queue and retried.

Interfaces

The following picture shows the scheduling context of a Pod and the interfaces that the scheduling framework exposes.

One plugin may implement multiple interfaces to perform more complex or stateful tasks.

Some interfaces match the scheduler extension points which can be configured through Scheduler Configuration.

Scheduling framework extension points

PreEnqueue

These plugins are called prior to adding Pods to the internal active queue, where Pods are marked as ready for scheduling.

Only when all PreEnqueue plugins return `Success`, the Pod is allowed to enter the active queue. Otherwise, it's placed in the internal unschedulable Pods list, and doesn't get an `Unschedulable` condition.

For more details about how internal scheduler queues work, read Scheduling queue in kube-scheduler.

EnqueueExtension

EnqueueExtension is the interface where the plugin can control whether to retry scheduling of Pods rejected by this plugin, based on changes in the cluster. Plugins that implement PreEnqueue, PreFilter, Filter, Reserve or Permit should implement this interface.

QueueingHint

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

QueueingHint is a callback function for deciding whether a Pod can be requeued to the active queue or backoff queue. It's executed every time a certain kind of event or change happens in the cluster. When the QueueingHint finds that the event might make the Pod schedulable, the Pod is put into the active queue or the backoff queue so that the scheduler will retry the scheduling of the Pod.

QueueSort

These plugins are used to sort Pods in the scheduling queue. A queue sort plugin essentially provides a `Less(Pod1, Pod2)` function. Only one queue sort plugin may be enabled at a time.

PreFilter

These plugins are used to pre-process info about the Pod, or to check certain conditions that the cluster or the Pod must meet. If a PreFilter plugin returns an error, the scheduling cycle is aborted.

Filter

These plugins are used to filter out nodes that cannot run the Pod. For each node, the scheduler will call filter plugins in their configured order. If any filter plugin marks the node as infeasible, the remaining plugins will not be called for that node. Nodes may be evaluated concurrently.

PostFilter

These plugins are called after the Filter phase, but only when no feasible nodes were found for the pod. Plugins are called in their configured order. If any postFilter plugin marks the node as `Schedulable`, the remaining plugins will not be called. A typical PostFilter implementation is preemption, which tries to make the pod schedulable by preempting other Pods.

PreScore

These plugins are used to perform "pre-scoring" work, which generates a sharable state for Score plugins to use. If a PreScore plugin returns an error, the scheduling cycle is aborted.

Score

These plugins are used to rank nodes that have passed the filtering phase. The scheduler will call each scoring plugin for each node. There will be a well defined range of integers representing the minimum and maximum scores. After the [NormalizeScore] phase, the scheduler will combine node scores from all plugins according to the configured plugin weights.

Capacity scoring

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

The feature gate `VolumeCapacityPriority` was used in v1.32 to support storage that are statically provisioned. Starting from v1.33, the new feature gate `StorageCapacityScoring` replaces the old `VolumeCapacityPriority` gate with added support to dynamically provisioned storage. When `StorageCapacityScoring` is enabled, the VolumeBinding plugin in the kube-scheduler is extended to score Nodes based on the storage capacity on each of them. This feature is applicable to CSI volumes that supported [Storage Capacity], including local storage backed by a CSI driver.

NormalizeScore

These plugins are used to modify scores before the scheduler computes a final ranking of Nodes. A plugin that registers for this extension point will be called with the [Score] results from the same plugin. This is called once per plugin per scheduling cycle.

For example, suppose a plugin `BlinkingLightScorer` ranks Nodes based on how many blinking lights they have.

```
func ScoreNode(_ *v1.pod, n *v1.Node) (int, error) {
    return getBlinkingLightCount(n)
}
```

However, the maximum count of blinking lights may be small compared to `NodeScoreMax`. To fix this, `BlinkingLightScorer` should also register for this extension point.

```
func NormalizeScores(scores map[string]int) {
    highest := 0
    for _, score := range scores {
        highest = max(highest, score)
    }
    for node, score := range scores {
        scores[node] = score*NodeScoreMax/highest
    }
}
```

If any NormalizeScore plugin returns an error, the scheduling cycle is aborted.

Note:

Plugins wishing to perform "pre-reserve" work should use the NormalizeScore extension point.

Reserve

A plugin that implements the Reserve interface has two methods, namely `Reserve` and `Unreserve`, that back two informational scheduling phases called Reserve and Unreserve, respectively. Plugins which maintain runtime state (aka "stateful plugins") should use these phases to be notified by the scheduler when resources on a node are being reserved and unreserved for a given Pod.

The Reserve phase happens before the scheduler actually binds a Pod to its designated node. It exists to prevent race conditions while the scheduler waits for the bind to succeed. The `Reserve` method of each Reserve plugin may succeed or fail; if one `Reserve` method call fails, subsequent plugins are not executed and the Reserve phase is considered to have failed. If the `Reserve` method of all plugins succeed, the Reserve phase is considered to be successful and the rest of the scheduling cycle and the binding cycle are executed.

The Unreserve phase is triggered if the Reserve phase or a later phase fails. When this happens, the `Unreserve` method of **all** Reserve plugins will be executed in the reverse order of `Reserve` method calls. This phase exists to clean up the state associated with the reserved Pod.

Caution:

The implementation of the `Unreserve` method in Reserve plugins must be idempotent and may not fail.

Permit

Permit plugins are invoked at the end of the scheduling cycle for each Pod, to prevent or delay the binding to the candidate node. A permit plugin can do one of the three things:

1. **approve**
   Once all Permit plugins approve a Pod, it is sent for binding.

2. **deny**
   If any Permit plugin denies a Pod, it is returned to the scheduling queue. This will trigger the Unreserve phase in Reserve plugins.

3. **wait** (with a timeout)
   If a Permit plugin returns "wait", then the Pod is kept in an internal "waiting" Pods list, and the binding cycle of this Pod starts but directly blocks until it gets approved. If a timeout occurs, **wait** becomes **deny** and the Pod is returned to the scheduling queue, triggering the Unreserve phase in Reserve plugins.

Note:

While any plugin can access the list of "waiting" Pods and approve them (see `FrameworkHandle`), we expect only the permit plugins to approve binding of reserved Pods that are in "waiting" state. Once a Pod is approved, it is sent to the PreBind phase.

PreBind

These plugins are used to perform any work required before a Pod is bound. For example, a pre-bind plugin may provision a network volume and mount it on the target node before allowing the Pod to run there.

If any PreBind plugin returns an error, the Pod is rejected and returned to the scheduling queue.

Bind

These plugins are used to bind a Pod to a Node. Bind plugins will not be called until all PreBind plugins have completed. Each bind plugin is called in the configured order. A bind plugin may choose whether or not to handle the given Pod. If a bind plugin chooses to handle a Pod, **the remaining bind plugins are skipped**.

PostBind

This is an informational interface. Post-bind plugins are called after a Pod is successfully bound. This is the end of a binding cycle, and can be used to clean up associated resources.

Plugin API

There are two steps to the plugin API. First, plugins must register and get configured, then they use the extension point interfaces. Extension point interfaces have the following form.

```
type Plugin interface {
    Name() string
}

type QueueSortPlugin interface {
    Plugin
    Less(*v1.pod, *v1.pod) bool
}

type PreFilterPlugin interface {
    Plugin
    PreFilter(context.Context, *framework.CycleState, *v1.pod) error
}

// ...
```

Plugin configuration

You can enable or disable plugins in the scheduler configuration. If you are using Kubernetes v1.18 or later, most scheduling plugins are in use and enabled by default.

In addition to default plugins, you can also implement your own scheduling plugins and get them configured along with default plugins. You can visit scheduler-plugins for more details.

If you are using Kubernetes v1.18 or later, you can configure a set of plugins as a scheduler profile and then define multiple profiles to fit various kinds of workload. Learn more at multiple profiles.

Linux kernel security constraints for Pods and containers

Overview of Linux kernel security modules and constraints that you can use to harden your Pods and containers.

This page describes some of the security features that are built into the Linux kernel that you can use in your Kubernetes workloads. To learn how to apply these features to your Pods and containers, refer to Configure a SecurityContext for a Pod or Container. You should already be familiar with Linux and with the basics of Kubernetes workloads.

Run workloads without root privileges

When you deploy a workload in Kubernetes, use the Pod specification to restrict that workload from running as the root user on the node. You can use the Pod `securityContext` to define the specific Linux user and group for the processes in the Pod, and explicitly restrict containers from running as root users. Setting these values in the Pod manifest takes precedence over similar values in the container image, which is especially useful if you're running images that you don't own.

Caution:

Ensure that the user or group that you assign to the workload has the permissions required for the application to function correctly. Changing the user or group to one that doesn't have the correct permissions could lead to file access issues or failed operations.

Configuring the kernel security features on this page provides fine-grained control over the actions that processes in your cluster can take, but managing these configurations can be challenging at scale. Running containers as non-root, or in user namespaces if you need root privileges, helps to reduce the chance that you'll need to enforce your configured kernel security capabilities.

Security features in the Linux kernel

Kubernetes lets you configure and use Linux kernel features to improve isolation and harden your containerized workloads. Common features include the following:

- **Secure computing mode (seccomp)**: Filter which system calls a process can make
- **AppArmor**: Restrict the access privileges of individual programs
- **Security Enhanced Linux (SELinux)**: Assign security labels to objects for more manageable security policy enforcement

To configure settings for one of these features, the operating system that you choose for your nodes must enable the feature in the kernel. For example, Ubuntu 7.10 and later enable AppArmor by default. To learn whether your OS enables a specific feature, consult the OS documentation.

You use the `securityContext` field in your Pod specification to define the constraints that apply to those processes. The `securityContext` field also supports other security settings, such as specific Linux capabilities or file access permissions using UIDs and GIDs. To learn more, refer to Configure a SecurityContext for a Pod or Container.

seccomp

Some of your workloads might need privileges to perform specific actions as the root user on your node's host machine. Linux uses *capabilities* to divide the available privileges into categories, so that processes can get the privileges required to perform specific actions without being granted all privileges. Each capability has a set of system calls (syscalls) that a process can make. seccomp lets you restrict these individual syscalls. It can be used to sandbox the privileges of a process, restricting the calls it is able to make from userspace into the kernel.

In Kubernetes, you use a *container runtime* on each node to run your containers. Example runtimes include CRI-O, Docker, or containerd. Each runtime allows only a subset of Linux capabilities by default. You can further limit the allowed syscalls individually by using a seccomp profile. Container runtimes usually include a default seccomp profile. Kubernetes lets you automatically apply seccomp profiles loaded onto a node to your Pods and containers.

Note:

Kubernetes also has the `allowPrivilegeEscalation` setting for Pods and containers. When set to `false`, this prevents processes from gaining new capabilities and restricts unprivileged users from changing the applied seccomp profile to a more permissive profile.

To learn how to implement seccomp in Kubernetes, refer to Restrict a Container's Syscalls with seccomp or the Seccomp node reference

To learn more about seccomp, see Seccomp BPF in the Linux kernel documentation.

Considerations for seccomp

seccomp is a low-level security configuration that you should only configure yourself if you require fine-grained control over Linux syscalls. Using seccomp, especially at scale, has the following risks:

- Configurations might break during application updates
- Attackers can still use allowed syscalls to exploit vulnerabilities
- Profile management for individual applications becomes challenging at scale

Recommendation: Use the default seccomp profile that's bundled with your container runtime. If you need a more isolated environment, consider using a sandbox, such as gVisor. Sandboxes solve the preceding risks with custom seccomp profiles, but require more compute resources on your nodes and might have compatibility issues with GPUs and other specialized hardware.

AppArmor and SELinux: policy-based mandatory access control

You can use Linux policy-based mandatory access control (MAC) mechanisms, such as AppArmor and SELinux, to harden your Kubernetes workloads.

AppArmor

AppArmor is a Linux kernel security module that supplements the standard Linux user and group based permissions to confine programs to a limited set of resources. AppArmor can be configured for any application to reduce its potential attack surface and provide greater in-depth defense. It is configured through profiles tuned to allow the access needed by a specific program or container, such as Linux capabilities, network access, and file permissions. Each profile can be run in either enforcing mode, which blocks access to disallowed resources, or complain mode, which only reports violations.

AppArmor can help you to run a more secure deployment by restricting what containers are allowed to do, and/or provide better auditing through system logs. The container runtime that you use might ship with a default AppArmor profile, or you can use a custom profile.

To learn how to use AppArmor in Kubernetes, refer to Restrict a Container's Access to Resources with AppArmor.

SELinux

SELinux is a Linux kernel security module that lets you restrict the access that a specific *subject*, such as a process, has to the files on your system. You define security policies that apply to subjects that have specific SELinux labels. When a process that has an SELinux label attempts to access a file, the SELinux server checks whether that process' security policy allows the access and makes an authorization decision.

In Kubernetes, you can set an SELinux label in the `securityContext` field of your manifest. The specified labels are assigned to those processes. If you have configured security policies that affect those labels, the host OS kernel enforces these policies.

To learn how to use SELinux in Kubernetes, refer to [Assign SELinux labels to a container](#).

Differences between AppArmor and SELinux

The operating system on your Linux nodes usually includes one of either AppArmor or SELinux. Both mechanisms provide similar types of protection, but have differences such as the following:

- **Configuration**: AppArmor uses profiles to define access to resources. SELinux uses policies that apply to specific labels.
- **Policy application**: In AppArmor, you define resources using file paths. SELinux uses the index node (inode) of a resource to identify the resource.

Summary of features

The following table describes the use cases and scope of each security control. You can use all of these controls together to build a more hardened system.

Summary of Linux kernel security features

Security feature	Description	How to use	Example
seccomp	Restrict individual kernel calls in the userspace. Reduces the likelihood that a vulnerability that uses a restricted syscall would compromise the system.	Specify a loaded seccomp profile in the Pod or container specification to apply its constraints to the processes in the Pod.	Reject the `unshare` syscall, which was used in [CVE-2022-0185](#).
AppArmor	Restrict program access to specific resources. Reduces the attack surface of the program. Improves audit logging.	Specify a loaded AppArmor profile in the container specification.	Restrict a read-only program from writing to any file path in the system.
SELinux	Restrict access to resources such as files, applications, ports, and processes using labels and security policies.	Specify access restrictions for specific labels. Tag processes with those labels to enforce the access restrictions related to the label.	Restrict a container from accessing files outside its own filesystem.

Note:

Mechanisms like AppArmor and SELinux can provide protection that extends beyond the container. For example, you can use SELinux to help mitigate [CVE-2019-5736](#).

Considerations for managing custom configurations

seccomp, AppArmor, and SELinux usually have a default configuration that offers basic protections. You can also create custom profiles and policies that meet the requirements of your workloads. Managing and distributing these custom configurations at scale might be challenging, especially if you use all three features together. To help you to manage these configurations at scale, use a tool like the [Kubernetes Security Profiles Operator](#).

Kernel-level security features and privileged containers

Kubernetes lets you specify that some trusted containers can run in *privileged* mode. Any container in a Pod can run in privileged mode to use operating system administrative capabilities that would otherwise be inaccessible. This is available for both Windows and Linux.

Privileged containers explicitly override some of the Linux kernel constraints that you might use in your workloads, as follows:

- **seccomp**: Privileged containers run as the `Unconfined` seccomp profile, overriding any seccomp profile that you specified in your manifest.
- **AppArmor**: Privileged containers ignore any applied AppArmor profiles.
- **SELinux**: Privileged containers run as the `unconfined_t` domain.

Privileged containers

Any container in a Pod can enable *Privileged mode* if you set the `privileged: true` field in the [`securityContext`](#) field for the container. Privileged containers override or undo many other hardening settings such as the applied seccomp profile, AppArmor profile, or SELinux constraints. Privileged containers are given all Linux capabilities, including capabilities that they don't require. For example, a root user in a privileged container might be able to use the `CAP_SYS_ADMIN` and `CAP_NET_ADMIN` capabilities on the node, bypassing the runtime seccomp configuration and other restrictions.

In most cases, you should avoid using privileged containers, and instead grant the specific capabilities required by your container using the `capabilities` field in the `securityContext` field. Only use privileged mode if you have a capability that you can't grant with the securityContext. This is useful for containers that want to use operating system administrative capabilities such as manipulating the network stack or accessing hardware devices.

In Kubernetes version 1.26 and later, you can also run Windows containers in a similarly privileged mode by setting the `windowsOptions.hostProcess` flag on the security context of the Pod spec. For details and instructions, see [Create a Windows HostProcess Pod](#).

Recommendations and best practices

- Before configuring kernel-level security capabilities, you should consider implementing network-level isolation. For more information, read the [Security Checklist](#).
- Unless necessary, run Linux workloads as non-root by setting specific user and group IDs in your Pod manifest and by specifying `runAsNonRoot: true`.

Additionally, you can run workloads in user namespaces by setting `hostUsers: false` in your Pod manifest. This lets you run containers as root users in the user namespace, but as non-root users in the host namespace on the node. This is still in early stages of development and might not have the level of support that you need. For instructions, refer to [Use a User Namespace With a Pod](#).

What's next

- Learn how to use AppArmor
- Learn how to use seccomp
- Learn how to use SELinux
- Seccomp Node Reference

IPv4/IPv6 dual-stack

Kubernetes lets you configure single-stack IPv4 networking, single-stack IPv6 networking, or dual stack networking with both network families active. This page explains how.
FEATURE STATE: `Kubernetes v1.23 [stable]`

IPv4/IPv6 dual-stack networking enables the allocation of both IPv4 and IPv6 addresses to Pods and Services.

IPv4/IPv6 dual-stack networking is enabled by default for your Kubernetes cluster starting in 1.21, allowing the simultaneous assignment of both IPv4 and IPv6 addresses.

Supported Features

IPv4/IPv6 dual-stack on your Kubernetes cluster provides the following features:

- Dual-stack Pod networking (a single IPv4 and IPv6 address assignment per Pod)
- IPv4 and IPv6 enabled Services
- Pod off-cluster egress routing (eg. the Internet) via both IPv4 and IPv6 interfaces

Prerequisites

The following prerequisites are needed in order to utilize IPv4/IPv6 dual-stack Kubernetes clusters:

- Kubernetes 1.20 or later

  For information about using dual-stack services with earlier Kubernetes versions, refer to the documentation for that version of Kubernetes.

- Provider support for dual-stack networking (Cloud provider or otherwise must be able to provide Kubernetes nodes with routable IPv4/IPv6 network interfaces)

- A network plugin that supports dual-stack networking.

Configure IPv4/IPv6 dual-stack

To configure IPv4/IPv6 dual-stack, set dual-stack cluster network assignments:

- kube-apiserver:
  - `--service-cluster-ip-range=<IPv4 CIDR>,<IPv6 CIDR>`
- kube-controller-manager:
  - `--cluster-cidr=<IPv4 CIDR>,<IPv6 CIDR>`
  - `--service-cluster-ip-range=<IPv4 CIDR>,<IPv6 CIDR>`
  - `--node-cidr-mask-size-ipv4|--node-cidr-mask-size-ipv6` defaults to /24 for IPv4 and /64 for IPv6
- kube-proxy:
  - `--cluster-cidr=<IPv4 CIDR>,<IPv6 CIDR>`
- kubelet:
  - `--node-ip=<IPv4 IP>,<IPv6 IP>`
    - This option is required for bare metal dual-stack nodes (nodes that do not define a cloud provider with the `--cloud-provider` flag). If you are using a cloud provider and choose to override the node IPs chosen by the cloud provider, set the `--node-ip` option.
    - (The legacy built-in cloud providers do not support dual-stack `--node-ip`.)

Note:

An example of an IPv4 CIDR: `10.244.0.0/16` (though you would supply your own address range)

An example of an IPv6 CIDR: `fdXY:IJKL:MNOP:15::/64` (this shows the format but is not a valid address - see RFC 4193)

Services

You can create Services which can use IPv4, IPv6, or both.

The address family of a Service defaults to the address family of the first service cluster IP range (configured via the `--service-cluster-ip-range` flag to the kube-apiserver).

When you define a Service you can optionally configure it as dual stack. To specify the behavior you want, you set the `.spec.ipFamilyPolicy` field to one of the following values:

- `SingleStack`: Single-stack service. The control plane allocates a cluster IP for the Service, using the first configured service cluster IP range.
- `PreferDualStack`: Allocates both IPv4 and IPv6 cluster IPs for the Service when dual-stack is enabled. If dual-stack is not enabled or supported, it falls back to single-stack behavior.

- `RequireDualStack`: Allocates Service `.spec.clusterIPs` from both IPv4 and IPv6 address ranges when dual-stack is enabled. If dual-stack is not enabled or supported, the Service API object creation fails.
    - Selects the `.spec.clusterIP` from the list of `.spec.clusterIPs` based on the address family of the first element in the `.spec.ipFamilies` array.

If you would like to define which IP family to use for single stack or define the order of IP families for dual-stack, you can choose the address families by setting an optional field, `.spec.ipFamilies`, on the Service.

Note:

The `.spec.ipFamilies` field is conditionally mutable: you can add or remove a secondary IP address family, but you cannot change the primary IP address family of an existing Service.

You can set `.spec.ipFamilies` to any of the following array values:

- `["IPv4"]`
- `["IPv6"]`
- `["IPv4","IPv6"]` (dual stack)
- `["IPv6","IPv4"]` (dual stack)

The first family you list is used for the legacy `.spec.clusterIP` field.

Dual-stack Service configuration scenarios

These examples demonstrate the behavior of various dual-stack Service configuration scenarios.

Dual-stack options on new Services

1. This Service specification does not explicitly define `.spec.ipFamilyPolicy`. When you create this Service, Kubernetes assigns a cluster IP for the Service from the first configured `service-cluster-ip-range` and sets the `.spec.ipFamilyPolicy` to `SingleStack`. (Services without selectors and headless Services with selectors will behave in this same way.)

    service/networking/dual-stack-default-svc.yaml Copy service/networking/dual-stack-default-svc.yaml to clipboard

    ```
    apiVersion: v1
    kind: Servicemetadata:  name: my-service  labels:    app.kubernetes.io/name: MyAppspec:  selector:    app.kubernetes.io/name:
    ```

2. This Service specification explicitly defines `PreferDualStack` in `.spec.ipFamilyPolicy`. When you create this Service on a dual-stack cluster, Kubernetes assigns both IPv4 and IPv6 addresses for the service. The control plane updates the `.spec` for the Service to record the IP address assignments. The field `.spec.clusterIPs` is the primary field, and contains both assigned IP addresses; `.spec.clusterIP` is a secondary field with its value calculated from `.spec.clusterIPs`.

    - For the `.spec.clusterIP` field, the control plane records the IP address that is from the same address family as the first service cluster IP range.
    - On a single-stack cluster, the `.spec.clusterIPs` and `.spec.clusterIP` fields both only list one address.
    - On a cluster with dual-stack enabled, specifying `RequireDualStack` in `.spec.ipFamilyPolicy` behaves the same as `PreferDualStack`.

    service/networking/dual-stack-preferred-svc.yaml Copy service/networking/dual-stack-preferred-svc.yaml to clipboard

    ```
    apiVersion: v1
    kind: Servicemetadata:  name: my-service  labels:    app.kubernetes.io/name: MyAppspec:  ipFamilyPolicy: PreferDualStack  sel
    ```

3. This Service specification explicitly defines `IPv6` and `IPv4` in `.spec.ipFamilies` as well as defining `PreferDualStack` in `.spec.ipFamilyPolicy`. When Kubernetes assigns an IPv6 and IPv4 address in `.spec.clusterIPs`, `.spec.clusterIP` is set to the IPv6 address because that is the first element in the `.spec.clusterIPs` array, overriding the default.

    service/networking/dual-stack-preferred-ipfamilies-svc.yaml
    Copy service/networking/dual-stack-preferred-ipfamilies-svc.yaml to clipboard

    ```
    apiVersion: v1
    kind: Servicemetadata:  name: my-service  labels:    app.kubernetes.io/name: MyAppspec:  ipFamilyPolicy: PreferDualStack  ipF
    ```

Dual-stack defaults on existing Services

These examples demonstrate the default behavior when dual-stack is newly enabled on a cluster where Services already exist. (Upgrading an existing cluster to 1.21 or beyond will enable dual-stack.)

1. When dual-stack is enabled on a cluster, existing Services (whether `IPv4` or `IPv6`) are configured by the control plane to set `.spec.ipFamilyPolicy` to `SingleStack` and set `.spec.ipFamilies` to the address family of the existing Service. The existing Service cluster IP will be stored in `.spec.clusterIPs`.

    service/networking/dual-stack-default-svc.yaml Copy service/networking/dual-stack-default-svc.yaml to clipboard

    ```
    apiVersion: v1
    kind: Servicemetadata:  name: my-service  labels:    app.kubernetes.io/name: MyAppspec:  selector:    app.kubernetes.io/name:
    ```

    You can validate this behavior by using kubectl to inspect an existing service.

    ```
    kubectl get svc my-service -o yaml
    ```

    ```
    apiVersion: v1
    kind: Servicemetadata:  labels:    app.kubernetes.io/name: MyApp  name: my-servicespec:  clusterIP: 10.0.197.123  clusterIPs:
    ```

2. When dual-stack is enabled on a cluster, existing headless Services with selectors are configured by the control plane to set `.spec.ipFamilyPolicy` to `SingleStack` and set `.spec.ipFamilies` to the address family of the first service cluster IP range (configured via the `--service-cluster-ip-range` flag to the kube-apiserver) even though `.spec.clusterIP` is set to `None`.

service/networking/dual-stack-default-svc.yaml Copy service/networking/dual-stack-default-svc.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: my-service  labels:    app.kubernetes.io/name: MyAppspec:  selector:    app.kubernetes.io/name:
```

You can validate this behavior by using kubectl to inspect an existing headless service with selectors.

```
kubectl get svc my-service -o yaml
```

```
apiVersion: v1
kind: Servicemetadata:  labels:    app.kubernetes.io/name: MyApp  name: my-servicespec:  clusterIP: None  clusterIPs:  - None
```

Switching Services between single-stack and dual-stack

Services can be changed from single-stack to dual-stack and from dual-stack to single-stack.

1. To change a Service from single-stack to dual-stack, change `.spec.ipFamilyPolicy` from `SingleStack` to `PreferDualStack` or `RequireDualStack` as desired. When you change this Service from single-stack to dual-stack, Kubernetes assigns the missing address family so that the Service now has IPv4 and IPv6 addresses.

   Edit the Service specification updating the `.spec.ipFamilyPolicy` from `SingleStack` to `PreferDualStack`.

   Before:

   ```
   spec:
     ipFamilyPolicy: SingleStack
   ```

   After:

   ```
   spec:
     ipFamilyPolicy: PreferDualStack
   ```

2. To change a Service from dual-stack to single-stack, change `.spec.ipFamilyPolicy` from `PreferDualStack` or `RequireDualStack` to `SingleStack`. When you change this Service from dual-stack to single-stack, Kubernetes retains only the first element in the `.spec.clusterIPs` array, and sets `.spec.clusterIP` to that IP address and sets `.spec.ipFamilies` to the address family of `.spec.clusterIPs`.

Headless Services without selector

For Headless Services without selectors and without `.spec.ipFamilyPolicy` explicitly set, the `.spec.ipFamilyPolicy` field defaults to `RequireDualStack`.

Service type LoadBalancer

To provision a dual-stack load balancer for your Service:

- Set the `.spec.type` field to `LoadBalancer`
- Set `.spec.ipFamilyPolicy` field to `PreferDualStack` or `RequireDualStack`

Note:

To use a dual-stack `LoadBalancer` type Service, your cloud provider must support IPv4 and IPv6 load balancers.

Egress traffic

If you want to enable egress traffic in order to reach off-cluster destinations (eg. the public Internet) from a Pod that uses non-publicly routable IPv6 addresses, you need to enable the Pod to use a publicly routed IPv6 address via a mechanism such as transparent proxying or IP masquerading. The ip-masq-agent project supports IP masquerading on dual-stack clusters.

Note:

Ensure your CNI provider supports IPv6.

Windows support

Kubernetes on Windows does not support single-stack "IPv6-only" networking. However, dual-stack IPv4/IPv6 networking for pods and nodes with single-family services is supported.

You can use IPv4/IPv6 dual-stack networking with `l2bridge` networks.

Note:

Overlay (VXLAN) networks on Windows **do not** support dual-stack networking.

You can read more about the different network modes for Windows within the Networking on Windows topic.

What's next

- Validate IPv4/IPv6 dual-stack networking
- Enable dual-stack networking using kubeadm

Networking on Windows

Kubernetes supports running nodes on either Linux or Windows. You can mix both kinds of node within a single cluster. This page provides an overview to networking specific to the Windows operating system.

Container networking on Windows

Networking for Windows containers is exposed through CNI plugins. Windows containers function similarly to virtual machines in regards to networking. Each container has a virtual network adapter (vNIC) which is connected to a Hyper-V virtual switch (vSwitch). The Host Networking Service (HNS) and the Host Compute Service (HCS) work together to create containers and attach container vNICs to networks. HCS is responsible for the management of containers whereas HNS is responsible for the management of networking resources such as:

- Virtual networks (including creation of vSwitches)
- Endpoints / vNICs
- Namespaces
- Policies including packet encapsulations, load-balancing rules, ACLs, and NAT rules.

The Windows HNS and vSwitch implement namespacing and can create virtual NICs as needed for a pod or container. However, many configurations such as DNS, routes, and metrics are stored in the Windows registry database rather than as files inside `/etc`, which is how Linux stores those configurations. The Windows registry for the container is separate from that of the host, so concepts like mapping `/etc/resolv.conf` from the host into a container don't have the same effect they would on Linux. These must be configured using Windows APIs run in the context of that container. Therefore CNI implementations need to call the HNS instead of relying on file mappings to pass network details into the pod or container.

Network modes

Windows supports five different networking drivers/modes: L2bridge, L2tunnel, Overlay (Beta), Transparent, and NAT. In a heterogeneous cluster with Windows and Linux worker nodes, you need to select a networking solution that is compatible on both Windows and Linux. The following table lists the out-of-tree plugins are supported on Windows, with recommendations on when to use each CNI:

Network Driver	Description	Container Packet Modifications	Network Plugins	Network Plugin Characteristics
L2bridge	Containers are attached to an external vSwitch. Containers are attached to the underlay network, although the physical network doesn't need to learn the container MACs because they are rewritten on ingress/egress.	MAC is rewritten to host MAC, IP may be rewritten to host IP using HNS OutboundNAT policy.	win-bridge, Azure-CNI, Flannel host-gateway uses win-bridge	win-bridge uses L2bridge network mode, connects containers to the underlay of hosts, offering best performance. Requires user-defined routes (UDR) for inter-node connectivity.
L2Tunnel	This is a special case of l2bridge, but only used on Azure. All packets are sent to the virtualization host where SDN policy is applied.	MAC rewritten, IP visible on the underlay network	Azure-CNI	Azure-CNI allows integration of containers with Azure vNET, and allows them to leverage the set of capabilities that Azure Virtual Network provides. For example, securely connect to Azure services or use Azure NSGs. See azure-cni for some examples
Overlay	Containers are given a vNIC connected to an external vSwitch. Each overlay network gets its own IP subnet, defined by a custom IP prefix.The overlay network driver uses VXLAN encapsulation.	Encapsulated with an outer header.	win-overlay, Flannel VXLAN (uses win-overlay)	win-overlay should be used when virtual container networks are desired to be isolated from underlay of hosts (e.g. for security reasons). Allows for IPs to be re-used for different overlay networks (which have different VNID tags) if you are restricted on IPs in your datacenter. This option requires KB4489899 on Windows Server 2019.
Transparent (special use case for ovn-kubernetes)	Requires an external vSwitch. Containers are attached to an external vSwitch which enables intra-pod communication via logical networks (logical switches and routers).	Packet is encapsulated either via GENEVE or STT tunneling to reach pods which are not on the same host. Packets are forwarded or dropped via the tunnel metadata information supplied by the ovn network controller. NAT is done for north-south communication.	ovn-kubernetes	Deploy via ansible. Distributed ACLs can be applied via Kubernetes policies. IPAM support. Load-balancing can be achieved without kube-proxy. NATing is done without using iptables/netsh.
NAT (*not used in Kubernetes*)	Containers are given a vNIC connected to an internal vSwitch. DNS/DHCP is provided using an internal component called WinNAT	MAC and IP is rewritten to host MAC/IP.	nat	Included here for completeness

As outlined above, the Flannel CNI plugin is also supported on Windows via the VXLAN network backend (**Beta support** ; delegates to win-overlay) and host-gateway network backend (stable support; delegates to win-bridge).

This plugin supports delegating to one of the reference CNI plugins (win-overlay, win-bridge), to work in conjunction with Flannel daemon on Windows (Flanneld) for automatic node subnet lease assignment and HNS network creation. This plugin reads in its own configuration file (cni.conf), and aggregates it with the environment variables from the FlannelD generated subnet.env file. It then delegates to one of the reference CNI plugins for network plumbing, and sends the correct configuration containing the node-assigned subnet to the IPAM plugin (for example: `host-local`).

For Node, Pod, and Service objects, the following network flows are supported for TCP/UDP traffic:

- Pod → Pod (IP)

- Pod → Pod (Name)
- Pod → Service (Cluster IP)
- Pod → Service (PQDN, but only if there are no ".")
- Pod → Service (FQDN)
- Pod → external (IP)
- Pod → external (DNS)
- Node → Pod
- Pod → Node

IP address management (IPAM)

The following IPAM options are supported on Windows:

- host-local
- azure-vnet-ipam (for azure-cni only)
- Windows Server IPAM (fallback option if no IPAM is set)

Direct Server Return (DSR)

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

Load balancing mode where the IP address fixups and the LBNAT occurs at the container vSwitch port directly; service traffic arrives with the source IP set as the originating pod IP. This provides performance optimizations by allowing the return traffic routed through load balancers to bypass the load balancer and respond directly to the client; reducing load on the load balancer and also reducing overall latency. For more information, read Direct Server Return (DSR) in a nutshell.

Load balancing and Services

A Kubernetes Service is an abstraction that defines a logical set of Pods and a means to access them over a network. In a cluster that includes Windows nodes, you can use the following types of Service:

- `NodePort`
- `ClusterIP`
- `LoadBalancer`
- `ExternalName`

Windows container networking differs in some important ways from Linux networking. The Microsoft documentation for Windows Container Networking provides additional details and background.

On Windows, you can use the following settings to configure Services and load balancing behavior:

Feature	Description	Minimum Supported Windows OS build	How to enable
Session affinity	Ensures that connections from a particular client are passed to the same Pod each time.	Windows Server 2022	Set `service.spec.sessionAffinity` to "ClientIP"
Direct Server Return (DSR)	See DSR notes above.	Windows Server 2019	Set the following command line argument (assuming version 1.34): `--enable-dsr=true`
Preserve-Destination	Skips DNAT of service traffic, thereby preserving the virtual IP of the target service in packets reaching the backend Pod. Also disables node-node forwarding.	Windows Server, version 1903	Set `"preserve-destination": "true"` in service annotations and enable DSR in kube-proxy.
IPv4/IPv6 dual-stack networking	Native IPv4-to-IPv4 in parallel with IPv6-to-IPv6 communications to, from, and within a cluster	Windows Server 2019	See IPv4/IPv6 dual-stack
Client IP preservation	Ensures that source IP of incoming ingress traffic gets preserved. Also disables node-node forwarding.	Windows Server 2019	Set `service.spec.externalTrafficPolicy` to "Local" and enable DSR in kube-proxy

Limitations

The following networking functionality is *not* supported on Windows nodes:

- Host networking mode
- Local NodePort access from the node itself (works for other nodes or external clients)
- More than 64 backend pods (or unique destination addresses) for a single Service
- IPv6 communication between Windows pods connected to overlay networks
- Local Traffic Policy in non-DSR mode
- Outbound communication using the ICMP protocol via the `win-overlay`, `win-bridge`, or using the Azure-CNI plugin. Specifically, the Windows data plane (VFP) doesn't support ICMP packet transpositions, and this means:
    - ICMP packets directed to destinations within the same network (such as pod to pod communication via ping) work as expected;
    - TCP/UDP packets work as expected;
    - ICMP packets directed to pass through a remote network (e.g. pod to external internet communication via ping) cannot be transposed and thus will not be routed back to their source;
    - Since TCP/UDP packets can still be transposed, you can substitute `ping <destination>` with `curl <destination>` when debugging connectivity with the outside world.

Other limitations:

- Windows reference network plugins win-bridge and win-overlay do not implement CNI spec v0.4.0, due to a missing `CHECK` implementation.

- The Flannel VXLAN CNI plugin has the following limitations on Windows:
  - Node-pod connectivity is only possible for local pods with Flannel v0.12.0 (or higher).
  - Flannel is restricted to using VNI 4096 and UDP port 4789. See the official Flannel VXLAN backend docs for more details on these parameters.

Controlling Access to the Kubernetes API

This page provides an overview of controlling access to the Kubernetes API.

Users access the Kubernetes API using `kubectl`, client libraries, or by making REST requests. Both human users and Kubernetes service accounts can be authorized for API access. When a request reaches the API, it goes through several stages, illustrated in the following diagram:

Diagram of request handling steps for Kubernetes API request

Transport security

By default, the Kubernetes API server listens on port 6443 on the first non-localhost network interface, protected by TLS. In a typical production Kubernetes cluster, the API serves on port 443. The port can be changed with the `--secure-port`, and the listening IP address with the `--bind-address` flag.

The API server presents a certificate. This certificate may be signed using a private certificate authority (CA), or based on a public key infrastructure linked to a generally recognized CA. The certificate and corresponding private key can be set by using the `--tls-cert-file` and `--tls-private-key-file` flags.

If your cluster uses a private certificate authority, you need a copy of that CA certificate configured into your `~/.kube/config` on the client, so that you can trust the connection and be confident it was not intercepted.

Your client can present a TLS client certificate at this stage.

Authentication

Once TLS is established, the HTTP request moves to the Authentication step. This is shown as step **1** in the diagram. The cluster creation script or cluster admin configures the API server to run one or more Authenticator modules. Authenticators are described in more detail in Authentication.

The input to the authentication step is the entire HTTP request; however, it typically examines the headers and/or client certificate.

Authentication modules include client certificates, password, and plain tokens, bootstrap tokens, and JSON Web Tokens (used for service accounts).

Multiple authentication modules can be specified, in which case each one is tried in sequence, until one of them succeeds.

If the request cannot be authenticated, it is rejected with HTTP status code 401. Otherwise, the user is authenticated as a specific `username`, and the user name is available to subsequent steps to use in their decisions. Some authenticators also provide the group memberships of the user, while other authenticators do not.

While Kubernetes uses usernames for access control decisions and in request logging, it does not have a `User` object nor does it store usernames or other information about users in its API.

Authorization

After the request is authenticated as coming from a specific user, the request must be authorized. This is shown as step **2** in the diagram.

A request must include the username of the requester, the requested action, and the object affected by the action. The request is authorized if an existing policy declares that the user has permissions to complete the requested action.

For example, if Bob has the policy below, then he can read pods only in the namespace `projectCaribou`:

```
{
    "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
    "kind": "Policy",
    "spec": {
        "user": "bob",
        "namespace": "projectCaribou",
        "resource": "pods",
        "readonly": true
    }
}
```

If Bob makes the following request, the request is authorized because he is allowed to read objects in the `projectCaribou` namespace:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "projectCaribou",
      "verb": "get",
      "group": "unicorn.example.org",
      "resource": "pods"
    }
  }
}
```

If Bob makes a request to write (`create` or `update`) to the objects in the `projectCaribou` namespace, his authorization is denied. If Bob makes a request to read (`get`) objects in a different namespace such as `projectFish`, then his authorization is denied.

Kubernetes authorization requires that you use common REST attributes to interact with existing organization-wide or cloud-provider-wide access control systems. It is important to use REST formatting because these control systems might interact with other APIs besides the Kubernetes API.

Kubernetes supports multiple authorization modules, such as ABAC mode, RBAC Mode, and Webhook mode. When an administrator creates a cluster, they configure the authorization modules that should be used in the API server. If more than one authorization modules are configured, Kubernetes checks each module, and if any module authorizes the request, then the request can proceed. If all of the modules deny the request, then the request is denied (HTTP status code 403).

To learn more about Kubernetes authorization, including details about creating policies using the supported authorization modules, see Authorization.

Admission control

Admission Control modules are software modules that can modify or reject requests. In addition to the attributes available to Authorization modules, Admission Control modules can access the contents of the object that is being created or modified.

Admission controllers act on requests that create, modify, delete, or connect to (proxy) an object. Admission controllers do not act on requests that merely read objects. When multiple admission controllers are configured, they are called in order.

This is shown as step **3** in the diagram.

Unlike Authentication and Authorization modules, if any admission controller module rejects, then the request is immediately rejected.

In addition to rejecting objects, admission controllers can also set complex defaults for fields.

The available Admission Control modules are described in Admission Controllers.

Once a request passes all admission controllers, it is validated using the validation routines for the corresponding API object, and then written to the object store (shown as step **4**).

Auditing

Kubernetes auditing provides a security-relevant, chronological set of records documenting the sequence of actions in a cluster. The cluster audits the activities generated by users, by applications that use the Kubernetes API, and by the control plane itself.

For more information, see Auditing.

What's next

Read more documentation on authentication, authorization and API access control:

- Authenticating
  - Authenticating with Bootstrap Tokens
- Admission Controllers
  - Dynamic Admission Control
- Authorization
  - Role Based Access Control
  - Attribute Based Access Control
  - Node Authorization
  - Webhook Authorization
- Certificate Signing Requests
  - including CSR approval and certificate signing
- Service accounts
  - Developer guide
  - Administration

You can learn about:

- how Pods can use Secrets to obtain API credentials.

Node-pressure Eviction

Node-pressure eviction is the process by which the kubelet proactively terminates pods to reclaim resource on nodes.

The kubelet monitors resources like memory, disk space, and filesystem inodes on your cluster's nodes. When one or more of these resources reach specific consumption levels, the kubelet can proactively fail one or more pods on the node to reclaim resources and prevent starvation.

During a node-pressure eviction, the kubelet sets the phase for the selected pods to `Failed`, and terminates the Pod.

Node-pressure eviction is not the same as API-initiated eviction.

The kubelet does not respect your configured PodDisruptionBudget or the pod's `terminationGracePeriodSeconds`. If you use soft eviction thresholds, the kubelet respects your configured `eviction-max-pod-grace-period`. If you use hard eviction thresholds, the kubelet uses a `0s` grace period (immediate shutdown) for termination.

Self healing behavior

The kubelet attempts to reclaim node-level resources before it terminates end-user pods. For example, it removes unused container images when disk resources are starved.

If the pods are managed by a workload management object (such as StatefulSet or Deployment) that replaces failed pods, the control plane (`kube-controller-manager`) creates new pods in place of the evicted pods.

Self healing for static pods

If you are running a static pod on a node that is under resource pressure, the kubelet may evict that static Pod. The kubelet then tries to create a replacement, because static Pods always represent an intent to run a Pod on that node.

The kubelet takes the *priority* of the static pod into account when creating a replacement. If the static pod manifest specifies a low priority, and there are higher-priority Pods defined within the cluster's control plane, and the node is under resource pressure, the kubelet may not be able to make room for that static pod. The kubelet continues to attempt to run all static pods even when there is resource pressure on a node.

Eviction signals and thresholds

The kubelet uses various parameters to make eviction decisions, like the following:

- Eviction signals
- Eviction thresholds
- Monitoring intervals

Eviction signals

Eviction signals are the current state of a particular resource at a specific point in time. The kubelet uses eviction signals to make eviction decisions by comparing the signals to eviction thresholds, which are the minimum amount of the resource that should be available on the node.

The kubelet uses the following eviction signals:

Eviction Signal	Description	Linux Only
`memory.available`	`memory.available` := `node.status.capacity[memory]` - `node.stats.memory.workingSet`	
`nodefs.available`	`nodefs.available` := `node.stats.fs.available`	
`nodefs.inodesFree`	`nodefs.inodesFree` := `node.stats.fs.inodesFree`	•
`imagefs.available`	`imagefs.available` := `node.stats.runtime.imagefs.available`	
`imagefs.inodesFree`	`imagefs.inodesFree` := `node.stats.runtime.imagefs.inodesFree`	•
`containerfs.available`	`containerfs.available` := `node.stats.runtime.containerfs.available`	
`containerfs.inodesFree`	`containerfs.inodesFree` := `node.stats.runtime.containerfs.inodesFree`	•
`pid.available`	`pid.available` := `node.stats.rlimit.maxpid` - `node.stats.rlimit.curproc`	•

In this table, the **Description** column shows how kubelet gets the value of the signal. Each signal supports either a percentage or a literal value. The kubelet calculates the percentage value relative to the total capacity associated with the signal.

Memory signals

On Linux nodes, the value for `memory.available` is derived from the cgroupfs instead of tools like `free -m`. This is important because `free -m` does not work in a container, and if users use the node allocatable feature, out of resource decisions are made local to the end user Pod part of the cgroup hierarchy as well as the root node. This script or cgroupv2 script reproduces the same set of steps that the kubelet performs to calculate `memory.available`. The kubelet excludes inactive_file (the number of bytes of file-backed memory on the inactive LRU list) from its calculation, as it assumes that memory is reclaimable under pressure.

On Windows nodes, the value for `memory.available` is derived from the node's global memory commit levels (queried through the `GetPerformanceInfo()` system call) by subtracting the node's global `CommitTotal` from the node's `CommitLimit`. Please note that `CommitLimit` can change if the node's page-file size changes!

Filesystem signals

The kubelet recognizes three specific filesystem identifiers that can be used with eviction signals (`<identifier>.inodesFree` or `<identifier>.available`):

1. `nodefs`: The node's main filesystem, used for local disk volumes, emptyDir volumes not backed by memory, log storage, ephemeral storage, and more. For example, `nodefs` contains `/var/lib/kubelet`.

2. `imagefs`: An optional filesystem that container runtimes can use to store container images (which are the read-only layers) and container writable layers.

3. `containerfs`: An optional filesystem that container runtime can use to store the writeable layers. Similar to the main filesystem (see `nodefs`), it's used to store local disk volumes, emptyDir volumes not backed by memory, log storage, and ephemeral storage, except for the container images. When `containerfs` is used, the `imagefs` filesystem can be split to only store images (read-only layers) and nothing else.

Note:

FEATURE STATE: `Kubernetes v1.31 [beta]` (enabled by default: true)

The *split image filesystem* feature, which enables support for the `containerfs` filesystem, adds several new eviction signals, thresholds and metrics. To use `containerfs`, the Kubernetes release v1.34 requires the `KubeletSeparateDiskGC` feature gate to be enabled. Currently, only CRI-O (v1.29 or higher) offers the `containerfs` filesystem support.

As such, kubelet generally allows three options for container filesystems:

- Everything is on the single `nodefs`, also referred to as "rootfs" or simply "root", and there is no dedicated image filesystem.
- Container storage (see `nodefs`) is on a dedicated disk, and `imagefs` (writable and read-only layers) is separate from the root filesystem. This is often referred to as "split disk" (or "separate disk") filesystem.
- Container filesystem `containerfs` (same as `nodefs` plus writable layers) is on root and the container images (read-only layers) are stored on separate `imagefs`. This is often referred to as "split image" filesystem.

The kubelet will attempt to auto-discover these filesystems with their current configuration directly from the underlying container runtime and will ignore other local node filesystems.

The kubelet does not support other container filesystems or storage configurations, and it does not currently support multiple filesystems for images and containers.

Deprecated kubelet garbage collection features

Some kubelet garbage collection features are deprecated in favor of eviction:

Existing Flag	Rationale
`--maximum-dead-containers`	deprecated once old logs are stored outside of container's context
`--maximum-dead-containers-per-container`	deprecated once old logs are stored outside of container's context
`--minimum-container-ttl-duration`	deprecated once old logs are stored outside of container's context

Eviction thresholds

You can specify custom eviction thresholds for the kubelet to use when it makes eviction decisions. You can configure soft and hard eviction thresholds.

Eviction thresholds have the form `[eviction-signal][operator][quantity]`, where:

- `eviction-signal` is the eviction signal to use.
- `operator` is the relational operator you want, such as `<` (less than).
- `quantity` is the eviction threshold amount, such as `1Gi`. The value of `quantity` must match the quantity representation used by Kubernetes. You can use either literal values or percentages (`%`).

For example, if a node has 10GiB of total memory and you want trigger eviction if the available memory falls below 1GiB, you can define the eviction threshold as either `memory.available<10%` or `memory.available<1Gi` (you cannot use both).

Soft eviction thresholds

A soft eviction threshold pairs an eviction threshold with a required administrator-specified grace period. The kubelet does not evict pods until the grace period is exceeded. The kubelet returns an error on startup if you do not specify a grace period.

You can specify both a soft eviction threshold grace period and a maximum allowed pod termination grace period for kubelet to use during evictions. If you specify a maximum allowed grace period and the soft eviction threshold is met, the kubelet uses the lesser of the two grace periods. If you do not specify a maximum allowed grace period, the kubelet kills evicted pods immediately without graceful termination.

You can use the following flags to configure soft eviction thresholds:

- `eviction-soft`: A set of eviction thresholds like `memory.available<1.5Gi` that can trigger pod eviction if held over the specified grace period.
- `eviction-soft-grace-period`: A set of eviction grace periods like `memory.available=1m30s` that define how long a soft eviction threshold must hold before triggering a Pod eviction.
- `eviction-max-pod-grace-period`: The maximum allowed grace period (in seconds) to use when terminating pods in response to a soft eviction threshold being met.

Hard eviction thresholds

A hard eviction threshold has no grace period. When a hard eviction threshold is met, the kubelet kills pods immediately without graceful termination to reclaim the starved resource.

You can use the `eviction-hard` flag to configure a set of hard eviction thresholds like `memory.available<1Gi`.

The kubelet has the following default hard eviction thresholds:

- `memory.available<100Mi` (Linux nodes)
- `memory.available<500Mi` (Windows nodes)
- `nodefs.available<10%`
- `imagefs.available<15%`
- `nodefs.inodesFree<5%` (Linux nodes)
- `imagefs.inodesFree<5%` (Linux nodes)

These default values of hard eviction thresholds will only be set if none of the parameters is changed. If you change the value of any parameter, then the values of other parameters will not be inherited as the default values and will be set to zero. In order to provide custom values, you should provide all the thresholds respectively. You can also set the kubelet config MergeDefaultEvictionSettings to true in the kubelet configuration file. If set to true and any parameter is changed, then the other parameters will inherit their default values instead of 0.

The `containerfs.available` and `containerfs.inodesFree` (Linux nodes) default eviction thresholds will be set as follows:

- If a single filesystem is used for everything, then `containerfs` thresholds are set the same as `nodefs`.

- If separate filesystems are configured for both images and containers, then `containerfs` thresholds are set the same as `imagefs`.

Setting custom overrides for thresholds related to `containersfs` is currently not supported, and a warning will be issued if an attempt to do so is made; any provided custom values will, as such, be ignored.

Eviction monitoring interval

The kubelet evaluates eviction thresholds based on its configured `housekeeping-interval`, which defaults to `10s`.

Node conditions

The kubelet reports node conditions to reflect that the node is under pressure because hard or soft eviction threshold is met, independent of configured grace periods.

The kubelet maps eviction signals to node conditions as follows:

Node Condition	Eviction Signal	Description
`MemoryPressure`	`memory.available`	Available memory on the node has satisfied an eviction threshold
`DiskPressure`	`nodefs.available`, `nodefs.inodesFree`, `imagefs.available`, `imagefs.inodesFree`, `containerfs.available`, or `containerfs.inodesFree`	Available disk space and inodes on either the node's root filesystem, image filesystem, or container filesystem has satisfied an eviction threshold
`PIDPressure`	`pid.available`	Available processes identifiers on the (Linux) node has fallen below an eviction threshold

The control plane also maps these node conditions to taints.

The kubelet updates the node conditions based on the configured `--node-status-update-frequency`, which defaults to `10s`.

Node condition oscillation

In some cases, nodes oscillate above and below soft eviction thresholds without holding for the defined grace periods. This causes the reported node condition to constantly switch between `true` and `false`, leading to bad eviction decisions.

To protect against oscillation, you can use the `eviction-pressure-transition-period` flag, which controls how long the kubelet must wait before transitioning a node condition to a different state. The transition period has a default value of `5m`.

Reclaiming node level resources

The kubelet tries to reclaim node-level resources before it evicts end-user pods.

When a `DiskPressure` node condition is reported, the kubelet reclaims node-level resources based on the filesystems on the node.

Without `imagefs` or `containerfs`

If the node only has a `nodefs` filesystem that meets eviction thresholds, the kubelet frees up disk space in the following order:

1. Garbage collect dead pods and containers.
2. Delete unused images.

With `imagefs`

If the node has a dedicated `imagefs` filesystem for container runtimes to use, the kubelet does the following:

- If the `nodefs` filesystem meets the eviction thresholds, the kubelet garbage collects dead pods and containers.
- If the `imagefs` filesystem meets the eviction thresholds, the kubelet deletes all unused images.

With `imagefs` and `containerfs`

If the node has a dedicated `containerfs` alongside the `imagefs` filesystem configured for the container runtimes to use, then kubelet will attempt to reclaim resources as follows:

- If the `containerfs` filesystem meets the eviction thresholds, the kubelet garbage collects dead pods and containers.
- If the `imagefs` filesystem meets the eviction thresholds, the kubelet deletes all unused images.

Pod selection for kubelet eviction

If the kubelet's attempts to reclaim node-level resources don't bring the eviction signal below the threshold, the kubelet begins to evict end-user pods.

The kubelet uses the following parameters to determine the pod eviction order:

1. Whether the pod's resource usage exceeds requests
2. Pod Priority
3. The pod's resource usage relative to requests

As a result, kubelet ranks and evicts pods in the following order:

1. `BestEffort` or `Burstable` pods where the usage exceeds requests. These pods are evicted based on their Priority and then by how much their usage level exceeds the request.
2. `Guaranteed` pods and `Burstable` pods where the usage is less than requests are evicted last, based on their Priority.

Note:

The kubelet does not use the pod's QoS class to determine the eviction order. You can use the QoS class to estimate the most likely pod eviction order when reclaiming resources like memory. QoS classification does not apply to EphemeralStorage requests, so the above scenario will not apply if the node is, for example, under `DiskPressure`.

`Guaranteed` pods are guaranteed only when requests and limits are specified for all the containers and they are equal. These pods will never be evicted because of another pod's resource consumption. If a system daemon (such as `kubelet` and `journald`) is consuming more resources than were reserved via `system-reserved` or `kube-reserved` allocations, and the node only has `Guaranteed` or `Burstable` pods using less resources than requests left on it, then the kubelet must choose to evict one of these pods to preserve node stability and to limit the impact of resource starvation on other pods. In this case, it will choose to evict pods of lowest Priority first.

If you are running a static pod and want to avoid having it evicted under resource pressure, set the `priority` field for that Pod directly. Static pods do not support the `priorityClassName` field.

When the kubelet evicts pods in response to inode or process ID starvation, it uses the Pods' relative priority to determine the eviction order, because inodes and PIDs have no requests.

The kubelet sorts pods differently based on whether the node has a dedicated `imagefs` or `containerfs` filesystem:

Without `imagefs` or `containerfs` (`nodefs` and `imagefs` use the same filesystem)

- If `nodefs` triggers evictions, the kubelet sorts pods based on their total disk usage (`local volumes + logs and a writable layer of all containers`).

With `imagefs` (`nodefs` and `imagefs` filesystems are separate)

- If `nodefs` triggers evictions, the kubelet sorts pods based on `nodefs` usage (`local volumes + logs of all containers`).
- If `imagefs` triggers evictions, the kubelet sorts pods based on the writable layer usage of all containers.

With `imagesfs` and `containerfs` (`imagefs` and `containerfs` have been split)

- If `containerfs` triggers evictions, the kubelet sorts pods based on `containerfs` usage (`local volumes + logs and a writable layer of all containers`).
- If `imagefs` triggers evictions, the kubelet sorts pods based on the `storage of images` rank, which represents the disk usage of a given image.

Minimum eviction reclaim

Note:

As of Kubernetes v1.34, you cannot set a custom value for the `containerfs.available` metric. The configuration for this specific metric will be set automatically to reflect values set for either the `nodefs` or `imagefs`, depending on the configuration.

In some cases, pod eviction only reclaims a small amount of the starved resource. This can lead to the kubelet repeatedly hitting the configured eviction thresholds and triggering multiple evictions.

You can use the `--eviction-minimum-reclaim` flag or a kubelet config file to configure a minimum reclaim amount for each resource. When the kubelet notices that a resource is starved, it continues to reclaim that resource until it reclaims the quantity you specify.

For example, the following configuration sets minimum reclaim amounts:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationevictionHard:  memory.available: "500Mi"  nodefs.available: "1Gi"  imagefs.available: "100Gi"evictionMin
```

In this example, if the `nodefs.available` signal meets the eviction threshold, the kubelet reclaims the resource until the signal reaches the threshold of 1GiB, and then continues to reclaim the minimum amount of 500MiB, until the available nodefs storage value reaches 1.5GiB.

Similarly, the kubelet tries to reclaim the `imagefs` resource until the `imagefs.available` value reaches `102Gi`, representing 102 GiB of available container image storage. If the amount of storage that the kubelet could reclaim is less than 2GiB, the kubelet doesn't reclaim anything.

The default `eviction-minimum-reclaim` is `0` for all resources.

Node out of memory behavior

If the node experiences an *out of memory* (OOM) event prior to the kubelet being able to reclaim memory, the node depends on the oom_killer to respond.

The kubelet sets an `oom_score_adj` value for each container based on the QoS for the pod.

Quality of Service	`oom_score_adj`
`Guaranteed`	-997
`BestEffort`	1000
`Burstable`	*min(max(2, 1000 - (1000 × memoryRequestBytes) / machineMemoryCapacityBytes), 999)*

Note:

The kubelet also sets an `oom_score_adj` value of `-997` for any containers in Pods that have `system-node-critical` Priority.

If the kubelet can't reclaim memory before a node experiences OOM, the `oom_killer` calculates an `oom_score` based on the percentage of memory it's using on the node, and then adds the `oom_score_adj` to get an effective `oom_score` for each container. It then kills the container with the highest score.

This means that containers in low QoS pods that consume a large amount of memory relative to their scheduling requests are killed first.

Unlike pod eviction, if a container is OOM killed, the kubelet can restart it based on its `restartPolicy`.

Good practices

The following sections describe good practice for eviction configuration.

Schedulable resources and eviction policies

When you configure the kubelet with an eviction policy, you should make sure that the scheduler will not schedule pods if they will trigger eviction because they immediately induce memory pressure.

Consider the following scenario:

- Node memory capacity: 10GiB
- Operator wants to reserve 10% of memory capacity for system daemons (kernel, `kubelet`, etc.)
- Operator wants to evict Pods at 95% memory utilization to reduce incidence of system OOM.

For this to work, the kubelet is launched as follows:

```
--eviction-hard=memory.available<500Mi
--system-reserved=memory=1.5Gi
```

In this configuration, the `--system-reserved` flag reserves 1.5GiB of memory for the system, which is `10% of the total memory + the eviction threshold amount`.

The node can reach the eviction threshold if a pod is using more than its request, or if the system is using more than 1GiB of memory, which makes the `memory.available` signal fall below 500MiB and triggers the threshold.

DaemonSets and node-pressure eviction

Pod priority is a major factor in making eviction decisions. If you do not want the kubelet to evict pods that belong to a DaemonSet, give those pods a high enough priority by specifying a suitable `priorityClassName` in the pod spec. You can also use a lower priority, or the default, to only allow pods from that DaemonSet to run when there are enough resources.

Known issues

The following sections describe known issues related to out of resource handling.

kubelet may not observe memory pressure right away

By default, the kubelet polls cAdvisor to collect memory usage stats at a regular interval. If memory usage increases within that window rapidly, the kubelet may not observe `MemoryPressure` fast enough, and the OOM killer will still be invoked.

You can use the `--kernel-memcg-notification` flag to enable the `memcg` notification API on the kubelet to get notified immediately when a threshold is crossed.

If you are not trying to achieve extreme utilization, but a sensible measure of overcommit, a viable workaround for this issue is to use the `--kube-reserved` and `--system-reserved` flags to allocate memory for the system.

active_file memory is not considered as available memory

On Linux, the kernel tracks the number of bytes of file-backed memory on active least recently used (LRU) list as the `active_file` statistic. The kubelet treats `active_file` memory areas as not reclaimable. For workloads that make intensive use of block-backed local storage, including ephemeral local storage, kernel-level caches of file and block data means that many recently accessed cache pages are likely to be counted as `active_file`. If enough of these kernel block buffers are on the active LRU list, the kubelet is liable to observe this as high resource use and taint the node as experiencing memory pressure - triggering pod eviction.

For more details, see https://github.com/kubernetes/kubernetes/issues/43916

You can work around that behavior by setting the memory limit and memory request the same for containers likely to perform intensive I/O activity. You will need to estimate or measure an optimal memory limit value for that container.

What's next

- Learn about API-initiated Eviction
- Learn about Pod Priority and Preemption
- Learn about PodDisruptionBudgets
- Learn about Quality of Service (QoS)
- Check out the Eviction API

Services, Load Balancing, and Networking

Concepts and resources behind networking in Kubernetes.

The Kubernetes network model

The Kubernetes network model is built out of several pieces:

- Each pod in a cluster gets its own unique cluster-wide IP address.
  - A pod has its own private network namespace which is shared by all of the containers within the pod. Processes running in different containers in the same pod can communicate with each other over `localhost`.
- The *pod network* (also called a cluster network) handles communication between pods. It ensures that (barring intentional network segmentation):
  - All pods can communicate with all other pods, whether they are on the same node or on different nodes. Pods can communicate with each other directly, without the use of proxies or address translation (NAT).

    On Windows, this rule does not apply to host-network pods.
  - Agents on a node (such as system daemons, or kubelet) can communicate with all pods on that node.
- The Service API lets you provide a stable (long lived) IP address or hostname for a service implemented by one or more backend pods, where the individual pods making up the service can change over time.
  - Kubernetes automatically manages EndpointSlice objects to provide information about the pods currently backing a Service.
  - A service proxy implementation monitors the set of Service and EndpointSlice objects, and programs the data plane to route service traffic to its backends, by using operating system or cloud provider APIs to intercept or rewrite packets.
- The Gateway API (or its predecessor, Ingress) allows you to make Services accessible to clients that are outside the cluster.
  - A simpler, but less-configurable, mechanism for cluster ingress is available via the Service API's `type: LoadBalancer`, when using a supported Cloud Provider.
- NetworkPolicy is a built-in Kubernetes API that allows you to control traffic between pods, or between pods and the outside world.

In older container systems, there was no automatic connectivity between containers on different hosts, and so it was often necessary to explicitly create links between containers, or to map container ports to host ports to make them reachable by containers on other hosts. This is not needed in Kubernetes; Kubernetes's model is that pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Only a few parts of this model are implemented by Kubernetes itself. For the other parts, Kubernetes defines the APIs, but the corresponding functionality is provided by external components, some of which are optional:

- Pod network namespace setup is handled by system-level software implementing the Container Runtime Interface.
- The pod network itself is managed by a pod network implementation. On Linux, most container runtimes use the Container Networking Interface (CNI) to interact with the pod network implementation, so these implementations are often called *CNI plugins*.
- Kubernetes provides a default implementation of service proxying, called kube-proxy, but some pod network implementations instead use their own service proxy that is more tightly integrated with the rest of the implementation.
- NetworkPolicy is generally also implemented by the pod network implementation. (Some simpler pod network implementations don't implement NetworkPolicy, or an administrator may choose to configure the pod network without NetworkPolicy support. In these cases, the API will still be present, but it will have no effect.)
- There are many implementations of the Gateway API, some of which are specific to particular cloud environments, some more focused on "bare metal" environments, and others more generic.

What's next

The Connecting Applications with Services tutorial lets you learn about Services and Kubernetes networking with a hands-on example.

Cluster Networking explains how to set up networking for your cluster, and also provides an overview of the technologies involved.

Service

Expose an application running in your cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.

Ingress

Make your HTTP (or HTTPS) network service available using a protocol-aware configuration mechanism, that understands web concepts like URIs, hostnames, paths, and more. The Ingress concept lets you map traffic to different backends based on rules you define via the Kubernetes API.

Ingress Controllers

In order for an Ingress to work in your cluster, there must be an *ingress controller* running. You need to select at least one ingress controller and make sure it is set up in your cluster. This page lists common ingress controllers that you can deploy.

Gateway API

Gateway API is a family of API kinds that provide dynamic infrastructure provisioning and advanced traffic routing.

EndpointSlices

The EndpointSlice API is the mechanism that Kubernetes uses to let your Service scale to handle large numbers of backends, and allows the cluster to update its list of healthy backends efficiently.

Network Policies

If you want to control traffic flow at the IP address or port level (OSI layer 3 or 4), NetworkPolicies allow you to specify rules for traffic flow within your cluster, and also between Pods and the outside world. Your cluster must use a network plugin that supports NetworkPolicy enforcement.

DNS for Services and Pods

Your workload can discover Services within your cluster using DNS; this page explains how that works.

IPv4/IPv6 dual-stack

Kubernetes lets you configure single-stack IPv4 networking, single-stack IPv6 networking, or dual stack networking with both network families active. This page explains how.

Topology Aware Routing

Topology Aware Routing provides a mechanism to help keep network traffic within the zone where it originated. Preferring same-zone traffic between Pods in your cluster can help with reliability, performance (network latency and throughput), or cost.

Networking on Windows

Service ClusterIP allocation

Service Internal Traffic Policy

If two Pods in your cluster want to communicate, and both Pods are actually running on the same node, use *Service Internal Traffic Policy* to keep network traffic within that node. Avoiding a round trip via the cluster network can help with reliability, performance (network latency and throughput), or cost.

Storage Capacity

Storage capacity is limited and may vary depending on the node on which a pod runs: network-attached storage might not be accessible by all nodes, or storage is local to a node to begin with.

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

This page describes how Kubernetes keeps track of storage capacity and how the scheduler uses that information to schedule Pods onto nodes that have access to enough storage capacity for the remaining missing volumes. Without storage capacity tracking, the scheduler may choose a node that doesn't have enough capacity to provision a volume and multiple scheduling retries will be needed.

Before you begin

Kubernetes v1.34 includes cluster-level API support for storage capacity tracking. To use this you must also be using a CSI driver that supports capacity tracking. Consult the documentation for the CSI drivers that you use to find out whether this support is available and, if so, how to use it. If you are not running Kubernetes v1.34, check the documentation for that version of Kubernetes.

API

There are two API extensions for this feature:

- CSIStorageCapacity objects: these get produced by a CSI driver in the namespace where the driver is installed. Each object contains capacity information for one storage class and defines which nodes have access to that storage.
- The `CSIDriverSpec.StorageCapacity` field: when set to `true`, the Kubernetes scheduler will consider storage capacity for volumes that use the CSI driver.

Scheduling

Storage capacity information is used by the Kubernetes scheduler if:

- a Pod uses a volume that has not been created yet,
- that volume uses a StorageClass which references a CSI driver and uses `WaitForFirstConsumer` volume binding mode, and
- the `CSIDriver` object for the driver has `StorageCapacity` set to true.

In that case, the scheduler only considers nodes for the Pod which have enough storage available to them. This check is very simplistic and only compares the size of the volume against the capacity listed in `CSIStorageCapacity` objects with a topology that includes the node.

For volumes with `Immediate` volume binding mode, the storage driver decides where to create the volume, independently of Pods that will use the volume. The scheduler then schedules Pods onto nodes where the volume is available after the volume has been created.

For CSI ephemeral volumes, scheduling always happens without considering storage capacity. This is based on the assumption that this volume type is only used by special CSI drivers which are local to a node and do not need significant resources there.

Rescheduling

When a node has been selected for a Pod with `WaitForFirstConsumer` volumes, that decision is still tentative. The next step is that the CSI storage driver gets asked to create the volume with a hint that the volume is supposed to be available on the selected node.

Because Kubernetes might have chosen a node based on out-dated capacity information, it is possible that the volume cannot really be created. The node selection is then reset and the Kubernetes scheduler tries again to find a node for the Pod.

Limitations

Storage capacity tracking increases the chance that scheduling works on the first try, but cannot guarantee this because the scheduler has to decide based on potentially out-dated information. Usually, the same retry mechanism as for scheduling without any storage capacity information handles scheduling failures.

One situation where scheduling can fail permanently is when a Pod uses multiple volumes: one volume might have been created already in a topology segment which then does not have enough capacity left for another volume. Manual intervention is necessary to recover from this, for example by increasing capacity or deleting the volume that was already created.

What's next

- For more information on the design, see the Storage Capacity Constraints for Pod Scheduling KEP.

Kubernetes Scheduler

In Kubernetes, *scheduling* refers to making sure that Pods are matched to Nodes so that Kubelet can run them.

Scheduling overview

A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below.

If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling.

kube-scheduler

kube-scheduler is the default scheduler for Kubernetes and runs as part of the control plane. kube-scheduler is designed so that, if you want and need to, you can write your own scheduling component and use that instead.

Kube-scheduler selects an optimal node to run newly created or not yet scheduled (unscheduled) pods. Since containers in pods - and pods themselves - can have different requirements, the scheduler filters out any nodes that don't meet a Pod's specific scheduling needs. Alternatively, the API lets you specify a node for a Pod when you create it, but this is unusual and is only done in special cases.

In a cluster, Nodes that meet the scheduling requirements for a Pod are called *feasible* nodes. If none of the nodes are suitable, the pod remains unscheduled until the scheduler is able to place it.

The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes and picks a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called *binding*.

Factors that need to be 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 so on.

Node selection in kube-scheduler

kube-scheduler selects a node for the pod in a 2-step operation:

1. Filtering
2. Scoring

The *filtering* step finds the set of Nodes where it's feasible to schedule the Pod. For example, the PodFitsResources filter checks whether a candidate Node has enough available resources to meet a Pod's specific resource requests. After this step, the node list contains any suitable Nodes; often, there will be more than one. If the list is empty, that Pod isn't (yet) schedulable.

In the *scoring* step, the scheduler ranks the remaining nodes to choose the most suitable Pod placement. The scheduler assigns a score to each Node that survived filtering, basing this score on the active scoring rules.

Finally, kube-scheduler assigns the Pod to the Node with the highest ranking. If there is more than one node with equal scores, kube-scheduler selects one of these at random.

There are two supported ways to configure the filtering and scoring behavior of the scheduler:

1. Scheduling Policies allow you to configure *Predicates* for filtering and *Priorities* for scoring.
2. Scheduling Profiles allow you to configure Plugins that implement different scheduling stages, including: `QueueSort`, `Filter`, `Score`, `Bind`, `Reserve`, `Permit`, and others. You can also configure the kube-scheduler to run different profiles.

What's next

- Read about scheduler performance tuning
- Read about Pod topology spread constraints
- Read the reference documentation for kube-scheduler
- Read the kube-scheduler config (v1) reference
- Learn about configuring multiple schedulers
- Learn about topology management policies
- Learn about Pod Overhead
- Learn about scheduling of Pods that use volumes in:
  - Volume Topology Support
  - Storage Capacity Tracking
  - Node-specific Volume Limits

Node-specific Volume Limits

This page describes the maximum number of volumes that can be attached to a Node for various cloud providers.

Cloud providers like Google, Amazon, and Microsoft typically have a limit on how many volumes can be attached to a Node. It is important for Kubernetes to respect those limits. Otherwise, Pods scheduled on a Node could get stuck waiting for volumes to attach.

Kubernetes default limits

The Kubernetes scheduler has default limits on the number of volumes that can be attached to a Node:

Cloud service	Maximum volumes per Node
Amazon Elastic Block Store (EBS)	39
Google Persistent Disk	16
Microsoft Azure Disk Storage	16

Dynamic volume limits

FEATURE STATE: `Kubernetes v1.17 [stable]`

Dynamic volume limits are supported for following volume types.

- Amazon EBS
- Google Persistent Disk
- Azure Disk
- CSI

For volumes managed by in-tree volume plugins, Kubernetes automatically determines the Node type and enforces the appropriate maximum number of volumes for the node. For example:

- On Google Compute Engine, up to 127 volumes can be attached to a node, depending on the node type.

- For Amazon EBS disks on M5,C5,R5,T3 and Z1D instance types, Kubernetes allows only 25 volumes to be attached to a Node. For other instance types on Amazon Elastic Compute Cloud (EC2), Kubernetes allows 39 volumes to be attached to a Node.

- On Azure, up to 64 disks can be attached to a node, depending on the node type. For more details, refer to Sizes for virtual machines in Azure.

- If a CSI storage driver advertises a maximum number of volumes for a Node (using `NodeGetInfo`), the kube-scheduler honors that limit. Refer to the CSI specifications for details.

- For volumes managed by in-tree plugins that have been migrated to a CSI driver, the maximum number of volumes will be the one reported by the CSI driver.

Mutable CSI Node Allocatable Count

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

CSI drivers can dynamically adjust the maximum number of volumes that can be attached to a Node at runtime. This enhances scheduling accuracy and reduces pod scheduling failures due to changes in resource availability.

To use this feature, you must enable the `MutableCSINodeAllocatableCount` feature gate on the following components:

- `kube-apiserver`
- `kubelet`

Periodic Updates

When enabled, CSI drivers can request periodic updates to their volume limits by setting the `nodeAllocatableUpdatePeriodSeconds` field in the `CSIDriver` specification. For example:

```
apiVersion: storage.k8s.io/v1
kind: CSIDrivermetadata:  name: hostpath.csi.k8s.iospec:  nodeAllocatableUpdatePeriodSeconds: 60
```

Kubelet will periodically call the corresponding CSI driver's `NodeGetInfo` endpoint to refresh the maximum number of attachable volumes, using the interval specified in `nodeAllocatableUpdatePeriodSeconds`. The minimum allowed value for this field is 10 seconds.

If a volume attachment operation fails with a `ResourceExhausted` error (gRPC code 8), Kubernetes triggers an immediate update to the allocatable volume count for that Node. Additionally, kubelet marks affected pods as Failed, allowing their controllers to handle recreation. This prevents pods from getting stuck indefinitely in the `ContainerCreating` state.

Ephemeral Volumes

This document describes *ephemeral volumes* in Kubernetes. Familiarity with volumes is suggested, in particular PersistentVolumeClaim and PersistentVolume.

Some applications need additional storage but don't care whether that data is stored persistently across restarts. For example, caching services are often limited by memory size and can move infrequently used data into storage that is slower than memory with little impact on overall performance.

Other applications expect some read-only input data to be present in files, like configuration data or secret keys.

Ephemeral volumes are designed for these use cases. Because volumes follow the Pod's lifetime and get created and deleted along with the Pod, Pods can be stopped and restarted without being limited to where some persistent volume is available.

Ephemeral volumes are specified *inline* in the Pod spec, which simplifies application deployment and management.

Types of ephemeral volumes

Kubernetes supports several different kinds of ephemeral volumes for different purposes:

- emptyDir: empty at Pod startup, with storage coming locally from the kubelet base directory (usually the root disk) or RAM
- configMap, downwardAPI, secret: inject different kinds of Kubernetes data into a Pod
- image: allows mounting container image files or artifacts, directly to a Pod.
- CSI ephemeral volumes: similar to the previous volume kinds, but provided by special CSI drivers which specifically support this feature
- generic ephemeral volumes, which can be provided by all storage drivers that also support persistent volumes

`emptyDir`, `configMap`, `downwardAPI`, `secret` are provided as local ephemeral storage. They are managed by kubelet on each node.

CSI ephemeral volumes *must* be provided by third-party CSI storage drivers.

Generic ephemeral volumes *can* be provided by third-party CSI storage drivers, but also by any other storage driver that supports dynamic provisioning. Some CSI drivers are written specifically for CSI ephemeral volumes and do not support dynamic provisioning: those then cannot be used for generic ephemeral volumes.

The advantage of using third-party drivers is that they can offer functionality that Kubernetes itself does not support, for example storage with different performance characteristics than the disk that is managed by kubelet, or injecting different data.

CSI ephemeral volumes

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

Note:

CSI ephemeral volumes are only supported by a subset of CSI drivers. The Kubernetes CSI Drivers list shows which drivers support ephemeral volumes.

Conceptually, CSI ephemeral volumes are similar to `configMap`, `downwardAPI` and `secret` volume types: the storage is managed locally on each node and is created together with other local resources after a Pod has been scheduled onto a node. Kubernetes has no concept of rescheduling Pods anymore at this stage. Volume creation has to be unlikely to fail, otherwise Pod startup gets stuck. In particular, storage capacity aware Pod scheduling is *not* supported for these volumes. They are currently also not covered by the storage resource usage limits of a Pod, because that is something that kubelet can only enforce for storage that it manages itself.

Here's an example manifest for a Pod that uses CSI ephemeral storage:

```
kind: Pod
apiVersion: v1metadata:  name: my-csi-appspec:  containers:    - name: my-frontend      image: busybox:1.28      volumeMounts:
```

The `volumeAttributes` determine what volume is prepared by the driver. These attributes are specific to each driver and not standardized. See the documentation of each CSI driver for further instructions.

CSI driver restrictions

CSI ephemeral volumes allow users to provide `volumeAttributes` directly to the CSI driver as part of the Pod spec. A CSI driver allowing `volumeAttributes` that are typically restricted to administrators is NOT suitable for use in an inline ephemeral volume. For example, parameters that are normally defined in the StorageClass should not be exposed to users through the use of inline ephemeral volumes.

Cluster administrators who need to restrict the CSI drivers that are allowed to be used as inline volumes within a Pod spec may do so by:

- Removing `Ephemeral` from `volumeLifecycleModes` in the CSIDriver spec, which prevents the driver from being used as an inline ephemeral volume.

- Using an admission webhook to restrict how this driver is used.

Generic ephemeral volumes

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

Generic ephemeral volumes are similar to `emptyDir` volumes in the sense that they provide a per-pod directory for scratch data that is usually empty after provisioning. But they may also have additional features:

- Storage can be local or network-attached.
- Volumes can have a fixed size that Pods are not able to exceed.
- Volumes may have some initial data, depending on the driver and parameters.
- Typical operations on volumes are supported assuming that the driver supports them, including snapshotting, cloning, resizing, and storage capacity tracking.

Example:

```
kind: Pod
apiVersion: v1metadata:  name: my-appspec:  containers:    - name: my-frontend      image: busybox:1.28      volumeMounts:      - 
```

Lifecycle and PersistentVolumeClaim

The key design idea is that the parameters for a volume claim are allowed inside a volume source of the Pod. Labels, annotations and the whole set of fields for a PersistentVolumeClaim are supported. When such a Pod gets created, the ephemeral volume controller then creates an actual PersistentVolumeClaim object in the same namespace as the Pod and ensures that the PersistentVolumeClaim gets deleted when the Pod gets deleted.

That triggers volume binding and/or provisioning, either immediately if the StorageClass uses immediate volume binding or when the Pod is tentatively scheduled onto a node (`WaitForFirstConsumer` volume binding mode). The latter is recommended for generic ephemeral volumes because then the scheduler is free to choose a suitable node for the Pod. With immediate binding, the scheduler is forced to select a node that has access to the volume once it is available.

In terms of resource ownership, a Pod that has generic ephemeral storage is the owner of the PersistentVolumeClaim(s) that provide that ephemeral storage. When the Pod is deleted, the Kubernetes garbage collector deletes the PVC, which then usually triggers deletion of the volume because the default reclaim policy of storage classes is to delete volumes. You can create quasi-ephemeral local storage using a StorageClass with a reclaim policy of `retain`: the storage outlives the Pod, and in this case you need to ensure that volume clean up happens separately.

While these PVCs exist, they can be used like any other PVC. In particular, they can be referenced as data source in volume cloning or snapshotting. The PVC object also holds the current status of the volume.

PersistentVolumeClaim naming

Naming of the automatically created PVCs is deterministic: the name is a combination of the Pod name and volume name, with a hyphen (`-`) in the middle. In the example above, the PVC name will be `my-app-scratch-volume`. This deterministic naming makes it easier to interact with the PVC because one does not have to search for it once the Pod name and volume name are known.

The deterministic naming also introduces a potential conflict between different Pods (a Pod "pod-a" with volume "scratch" and another Pod with name "pod" and volume "a-scratch" both end up with the same PVC name "pod-a-scratch") and between Pods and manually created PVCs.

Such conflicts are detected: a PVC is only used for an ephemeral volume if it was created for the Pod. This check is based on the ownership relationship. An existing PVC is not overwritten or modified. But this does not resolve the conflict because without the right PVC, the Pod cannot start.

Caution:

Take care when naming Pods and volumes inside the same namespace, so that these conflicts can't occur.

Security

Using generic ephemeral volumes allows users to create PVCs indirectly if they can create Pods, even if they do not have permission to create PVCs directly. Cluster administrators must be aware of this. If this does not fit their security model, they should use an admission webhook that rejects objects like Pods that have a generic ephemeral volume.

The normal namespace quota for PVCs still applies, so even if users are allowed to use this new mechanism, they cannot use it to circumvent other policies.

What's next

Ephemeral volumes managed by kubelet

See local ephemeral storage.

CSI ephemeral volumes

- For more information on the design, see the Ephemeral Inline CSI volumes KEP.
- For more information on further development of this feature, see the enhancement tracking issue #596.

Generic ephemeral volumes

- For more information on the design, see the Generic ephemeral inline volumes KEP.

Application Security Checklist

Baseline guidelines around ensuring application security on Kubernetes, aimed at application developers

This checklist aims to provide basic guidelines on securing applications running in Kubernetes from a developer's perspective. This list is not meant to be exhaustive and is intended to evolve over time.

On how to read and use this document:

- The order of topics does not reflect an order of priority.
- Some checklist items are detailed in the paragraph below the list of each section.
- This checklist assumes that a `developer` is a Kubernetes cluster user who interacts with namespaced scope objects.

Caution:

Checklists are **not** sufficient for attaining a good security posture on their own. A good security posture requires constant attention and improvement, but a checklist can be the first step on the never-ending journey towards security preparedness. Some recommendations in this checklist may be too restrictive or too lax for your specific security needs. Since Kubernetes security is not "one size fits all", each category of checklist items should be evaluated on its merits.

Base security hardening

The following checklist provides base security hardening recommendations that would apply to most applications deploying to Kubernetes.

Application design

- □ Follow the right security principles when designing applications.
- □ Application configured with appropriate QoS class through resource request and limits.
  - □ Memory limit is set for the workloads with a limit equal to or greater than the request.
  - □ CPU limit might be set on sensitive workloads.

Service account

- □ Avoid using the `default` ServiceAccount. Instead, create ServiceAccounts for each workload or microservice.
- □ `automountServiceAccountToken` should be set to `false` unless the pod specifically requires access to the Kubernetes API to operate.

Pod-level `securityContext` recommendations

- □ Set `runAsNonRoot: true`.
- □ Configure the container to execute as a less privileged user (for example, using `runAsUser` and `runAsGroup`), and configure appropriate permissions on files or directories inside the container image.
- □ Optionally add a supplementary group with `fsGroup` to access persistent volumes.
- □ The application deploys into a namespace that enforces an appropriate Pod security standard. If you cannot control this enforcement for the cluster(s) where the application is deployed, take this into account either through documentation or additional defense in depth.

Container-level `securityContext` recommendations

- □ Disable privilege escalations using `allowPrivilegeEscalation: false`.
- □ Configure the root filesystem to be read-only with `readOnlyRootFilesystem: true`.
- □ Avoid running privileged containers (set `privileged: false`).
- □ Drop all capabilities from the containers and add back only specific ones that are needed for operation of the container.

Role Based Access Control (RBAC)

- □ Permissions such as **create**, **patch**, **update** and **delete** should be only granted if necessary.
- □ Avoid creating RBAC permissions to create or update roles which can lead to privilege escalation.
- □ Review bindings for the `system:unauthenticated` group and remove them where possible, as this gives access to anyone who can contact the API server at a network level.

The **create**, **update** and **delete** verbs should be permitted judiciously. The **patch** verb if allowed on a Namespace can allow users to update labels on the namespace or deployments which can increase the attack surface.

For sensitive workloads, consider providing a recommended ValidatingAdmissionPolicy that further restricts the permitted write actions.

Image security

- □ Using an image scanning tool to scan an image before deploying containers in the Kubernetes cluster.
- □ Use container signing to validate the container image signature before deploying to the Kubernetes cluster.

Network policies

- □ Configure NetworkPolicies to only allow expected ingress and egress traffic from the pods.

Make sure that your cluster provides and enforces NetworkPolicy. If you are writing an application that users will deploy to different clusters, consider whether you can assume that NetworkPolicy is available and enforced.

Advanced security hardening

This section of this guide covers some advanced security hardening points which might be valuable based on different Kubernetes environment setup.

Linux container security

Configure Security Context for the pod-container.

- □ Set the Seccomp Profile for a Container.
- □ Restrict a Container's Access to Resources with AppArmor.
- □ Assign SELinux Labels to a Container.

Runtime classes

- □ Configure appropriate runtime classes for containers.

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.

Some containers may require a different isolation level from what is provided by the default runtime of the cluster. `runtimeClassName` can be used in a podspec to define a different runtime class.

For sensitive workloads consider using kernel emulation tools like gVisor, or virtualized isolation using a mechanism such as kata-containers.

In high trust environments, consider using confidential virtual machines to improve cluster security even further.

Windows Storage

This page provides an storage overview specific to the Windows operating system.

Persistent storage

Windows has a layered filesystem driver to mount container layers and create a copy filesystem based on NTFS. All file paths in the container are resolved only within the context of that container.

- With Docker, volume mounts can only target a directory in the container, and not an individual file. This limitation does not apply to containerd.
- Volume mounts cannot project files or directories back to the host filesystem.
- Read-only filesystems are not supported because write access is always required for the Windows registry and SAM database. However, read-only volumes are supported.
- Volume user-masks and permissions are not available. Because the SAM is not shared between the host & container, there's no mapping between them. All permissions are resolved within the context of the container.

As a result, the following storage functionality is not supported on Windows nodes:

- Volume subpath mounts: only the entire volume can be mounted in a Windows container
- Subpath volume mounting for Secrets
- Host mount projection
- Read-only root filesystem (mapped volumes still support `readOnly`)
- Block device mapping
- Memory as the storage medium (for example, `emptyDir.medium` set to `Memory`)
- File system features like uid/gid; per-user Linux filesystem permissions
- Setting secret permissions with DefaultMode (due to UID/GID dependency)
- NFS based storage/volume support
- Expanding the mounted volume (resizefs)

Kubernetes volumes enable complex applications, with data persistence and Pod volume sharing requirements, to be deployed on Kubernetes. Management of persistent volumes associated with a specific storage back-end or protocol includes actions such as provisioning/de-provisioning/resizing of volumes, attaching/detaching a volume to/from a Kubernetes node and mounting/dismounting a volume to/from individual containers in a pod that needs to persist data.

Volume management components are shipped as Kubernetes volume plugin. The following broad classes of Kubernetes volume plugins are supported on Windows:

- `FlexVolume plugins`
  - Please note that FlexVolumes have been deprecated as of 1.23
- `CSI Plugins`

In-tree volume plugins

The following in-tree plugins support persistent storage on Windows nodes:

- `azureFile`
- `vsphereVolume`

Pod Scheduling Readiness

FEATURE STATE: `Kubernetes v1.30 [stable]`

Pods were considered ready for scheduling once created. Kubernetes scheduler does its due diligence to find nodes to place all pending Pods. However, in a real-world case, some Pods may stay in a "miss-essential-resources" state for a long period. These Pods actually churn the scheduler (and downstream integrators like Cluster AutoScaler) in an unnecessary manner.

By specifying/removing a Pod's `.spec.schedulingGates`, you can control when a Pod is ready to be considered for scheduling.

Configuring Pod schedulingGates

The `schedulingGates` field contains a list of strings, and each string literal is perceived as a criteria that Pod should be satisfied before considered schedulable. This field can be initialized only when a Pod is created (either by the client, or mutated during admission). After creation, each schedulingGate can be removed in arbitrary order, but addition of a new scheduling gate is disallowed.

pod-scheduling-gates-diagram

Figure. Pod SchedulingGates

Usage example

To mark a Pod not-ready for scheduling, you can create it with one or more scheduling gates like this:

`pods/pod-with-scheduling-gates.yaml` Copy pods/pod-with-scheduling-gates.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: test-podspec:  schedulingGates:  - name: example.com/foo  - name: example.com/bar  containers:  - name:
```

After the Pod's creation, you can check its state using:

```
kubectl get pod test-pod
```

The output reveals it's in `SchedulingGated` state:

```
NAME       READY   STATUS            RESTARTS   AGE
test-pod   0/1     SchedulingGated   0          7s
```

You can also check its `schedulingGates` field by running:

```
kubectl get pod test-pod -o jsonpath='{.spec.schedulingGates}'
```

The output is:

```
[{"name":"example.com/foo"},{"name":"example.com/bar"}]
```

To inform scheduler this Pod is ready for scheduling, you can remove its `schedulingGates` entirely by reapplying a modified manifest:

`pods/pod-without-scheduling-gates.yaml` Copy pods/pod-without-scheduling-gates.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: test-podspec:  containers:  - name: pause    image: registry.k8s.io/pause:3.6
```

You can check if the `schedulingGates` is cleared by running:

```
kubectl get pod test-pod -o jsonpath='{.spec.schedulingGates}'
```

The output is expected to be empty. And you can check its latest status by running:

```
kubectl get pod test-pod -o wide
```

Given the test-pod doesn't request any CPU/memory resources, it's expected that this Pod's state get transited from previous `SchedulingGated` to `Running`:

```
NAME       READY   STATUS    RESTARTS   AGE   IP         NODE
test-pod   1/1     Running   0          15s   10.0.0.4   node-2
```

Observability

The metric `scheduler_pending_pods` comes with a new label `"gated"` to distinguish whether a Pod has been tried scheduling but claimed as unschedulable, or explicitly marked as not ready for scheduling. You can use `scheduler_pending_pods{queue="gated"}` to check the metric result.

Mutable Pod scheduling directives

You can mutate scheduling directives of Pods while they have scheduling gates, with certain constraints. At a high level, you can only tighten the scheduling directives of a Pod. In other words, the updated directives would cause the Pods to only be able to be scheduled on a subset of the nodes that it would previously match. More concretely, the rules for updating a Pod's scheduling directives are as follows:

1. For `.spec.nodeSelector`, only additions are allowed. If absent, it will be allowed to be set.

2. For `spec.affinity.nodeAffinity`, if nil, then setting anything is allowed.

3. If `NodeSelectorTerms` was empty, it will be allowed to be set. If not empty, then only additions of `NodeSelectorRequirements` to `matchExpressions` or `fieldExpressions` are allowed, and no changes to existing `matchExpressions` and `fieldExpressions` will be allowed. This is because the terms in `.requiredDuringSchedulingIgnoredDuringExecution.NodeSelectorTerms`, are ORed while the expressions in `nodeSelectorTerms[].matchExpressions` and `nodeSelectorTerms[].fieldExpressions` are ANDed.

4. For `.preferredDuringSchedulingIgnoredDuringExecution`, all updates are allowed. This is because preferred terms are not authoritative, and so policy controllers don't validate those terms.

What's next

- Read the PodSchedulingReadiness KEP for more details

Cloud Native Security and Kubernetes

Concepts for keeping your cloud-native workload secure.

Kubernetes is based on a cloud-native architecture, and draws on advice from the CNCF about good practice for cloud native information security.

Read on through this page for an overview of how Kubernetes is designed to help you deploy a secure cloud native platform.

Cloud native information security

The CNCF white paper on cloud native security defines security controls and practices that are appropriate to different *lifecycle phases*.

Develop lifecycle phase

- Ensure the integrity of development environments.
- Design applications following good practice for information security, appropriate for your context.
- Consider end user security as part of solution design.

To achieve this, you can:

1. Adopt an architecture, such as zero trust, that minimizes attack surfaces, even for internal threats.
2. Define a code review process that considers security concerns.
3. Build a *threat model* of your system or application that identifies trust boundaries. Use that to model to identify risks and to help find ways to treat those risks.
4. Incorporate advanced security automation, such as *fuzzing* and security chaos engineering, where it's justified.

Distribute lifecycle phase

- Ensure the security of the supply chain for container images you execute.
- Ensure the security of the supply chain for the cluster and other components that execute your application. An example of another component might be an external database that your cloud-native application uses for persistence.

To achieve this, you can:

1. Scan container images and other artifacts for known vulnerabilities.
2. Ensure that software distribution uses encryption in transit, with a chain of trust for the software source.
3. Adopt and follow processes to update dependencies when updates are available, especially in response to security announcements.
4. Use validation mechanisms such as digital certificates for supply chain assurance.
5. Subscribe to feeds and other mechanisms to alert you to security risks.
6. Restrict access to artifacts. Place container images in a private registry that only allows authorized clients to pull images.

Deploy lifecycle phase

Ensure appropriate restrictions on what can be deployed, who can deploy it, and where it can be deployed to. You can enforce measures from the *distribute* phase, such as verifying the cryptographic identity of container image artifacts.

You can deploy different applications and cluster components into different namespaces. Containers themselves, and namespaces, both provide isolation mechanisms that are relevant to information security.

When you deploy Kubernetes, you also set the foundation for your applications' runtime environment: a Kubernetes cluster (or multiple clusters). That IT infrastructure must provide the security guarantees that higher layers expect.

Runtime lifecycle phase

The Runtime phase comprises three critical areas: access, compute, and storage.

Runtime protection: access

The Kubernetes API is what makes your cluster work. Protecting this API is key to providing effective cluster security.

Other pages in the Kubernetes documentation have more detail about how to set up specific aspects of access control. The security checklist has a set of suggested basic checks for your cluster.

Beyond that, securing your cluster means implementing effective authentication and authorization for API access. Use ServiceAccounts to provide and manage security identities for workloads and cluster components.

Kubernetes uses TLS to protect API traffic; make sure to deploy the cluster using TLS (including for traffic between nodes and the control plane), and protect the encryption keys. If you use Kubernetes' own API for CertificateSigningRequests, pay special attention to restricting misuse there.

Runtime protection: compute

Containers provide two things: isolation between different applications, and a mechanism to combine those isolated applications to run on the same host computer. Those two aspects, isolation and aggregation, mean that runtime security involves identifying trade-offs and finding an appropriate balance.

Kubernetes relies on a container runtime to actually set up and run containers. The Kubernetes project does not recommend a specific container runtime and you should make sure that the runtime(s) that you choose meet your information security needs.

To protect your compute at runtime, you can:

1. Enforce Pod security standards for applications, to help ensure they run with only the necessary privileges.

2. Run a specialized operating system on your nodes that is designed specifically for running containerized workloads. This is typically based on a read-only operating system (*immutable image*) that provides only the services essential for running containers.

   Container-specific operating systems help to isolate system components and present a reduced attack surface in case of a container escape.

3. Define ResourceQuotas to fairly allocate shared resources, and use mechanisms such as LimitRanges to ensure that Pods specify their resource requirements.

4. Partition workloads across different nodes. Use node isolation mechanisms, either from Kubernetes itself or from the ecosystem, to ensure that Pods with different trust contexts are run on separate sets of nodes.

5. Use a container runtime that provides security restrictions.

6. On Linux nodes, use a Linux security module such as AppArmor or seccomp.

Runtime protection: storage

To protect storage for your cluster and the applications that run there, you can:

1. Integrate your cluster with an external storage plugin that provides encryption at rest for volumes.
2. Enable encryption at rest for API objects.
3. Protect data durability using backups. Verify that you can restore these, whenever you need to.
4. Authenticate connections between cluster nodes and any network storage they rely upon.
5. Implement data encryption within your own application.

For encryption keys, generating these within specialized hardware provides the best protection against disclosure risks. A *hardware security module* can let you perform cryptographic operations without allowing the security key to be copied elsewhere.

Networking and security

You should also consider network security measures, such as NetworkPolicy or a service mesh. Some network plugins for Kubernetes provide encryption for your cluster network, using technologies such as a virtual private network (VPN) overlay. By design, Kubernetes lets you use your own networking plugin for your cluster (if you use managed Kubernetes, the person or organization managing your cluster may have chosen a network plugin for you).

The network plugin you choose and the way you integrate it can have a strong impact on the security of information in transit.

Observability and runtime security

Kubernetes lets you extend your cluster with extra tooling. You can set up third party solutions to help you monitor or troubleshoot your applications and the clusters they are running. You also get some basic observability features built in to Kubernetes itself. Your code running in containers can generate logs, publish metrics or provide other observability data; at deploy time, you need to make sure your cluster provides an appropriate level of protection there.

If you set up a metrics dashboard or something similar, review the chain of components that populate data into that dashboard, as well as the dashboard itself. Make sure that the whole chain is designed with enough resilience and enough integrity protection that you can rely on it even during an incident where your cluster might be degraded.

Where appropriate, deploy security measures below the level of Kubernetes itself, such as cryptographically measured boot, or authenticated distribution of time (which helps ensure the fidelity of logs and audit records).

For a high assurance environment, deploy cryptographic protections to ensure that logs are both tamper-proof and confidential.

What's next

Cloud native security

- CNCF white paper on cloud native security.
- CNCF white paper on good practices for securing a software supply chain.
- Fixing the Kubernetes clusterf**k: Understanding security from the kernel up (FOSDEM 2020)
- Kubernetes Security Best Practices (Kubernetes Forum Seoul 2019)
- Towards Measured Boot Out of the Box (Linux Security Summit 2016)

Kubernetes and information security

- Kubernetes security
- Securing your cluster
- Data encryption in transit for the control plane
- Data encryption at rest
- Secrets in Kubernetes

- Controlling Access to the Kubernetes API
- Network policies for Pods
- Pod security standards
- RuntimeClasses

Security

Concepts for keeping your cloud-native workload secure.

This section of the Kubernetes documentation aims to help you learn to run workloads more securely, and about the essential aspects of keeping a Kubernetes cluster secure.

Kubernetes is based on a cloud-native architecture, and draws on advice from the CNCF about good practice for cloud native information security.

Read Cloud Native Security and Kubernetes for the broader context about how to secure your cluster and the applications that you're running on it.

Kubernetes security mechanisms

Kubernetes includes several APIs and security controls, as well as ways to define policies that can form part of how you manage information security.

Control plane protection

A key security mechanism for any Kubernetes cluster is to control access to the Kubernetes API.

Kubernetes expects you to configure and use TLS to provide data encryption in transit within the control plane, and between the control plane and its clients. You can also enable encryption at rest for the data stored within Kubernetes control plane; this is separate from using encryption at rest for your own workloads' data, which might also be a good idea.

Secrets

The Secret API provides basic protection for configuration values that require confidentiality.

Workload protection

Enforce Pod security standards to ensure that Pods and their containers are isolated appropriately. You can also use RuntimeClasses to define custom isolation if you need it.

Network policies let you control network traffic between Pods, or between Pods and the network outside your cluster.

You can deploy security controls from the wider ecosystem to implement preventative or detective controls around Pods, their containers, and the images that run in them.

Admission control

Admission controllers are plugins that intercept Kubernetes API requests and can validate or mutate the requests based on specific fields in the request. Thoughtfully designing these controllers helps to avoid unintended disruptions as Kubernetes APIs change across version updates. For design considerations, see Admission Webhook Good Practices.

Auditing

Kubernetes audit logging provides a security-relevant, chronological set of records documenting the sequence of actions in a cluster. The cluster audits the activities generated by users, by applications that use the Kubernetes API, and by the control plane itself.

Cloud provider security

Note: Items on this page refer to vendors external to Kubernetes. The Kubernetes project authors aren't responsible for those third-party products or projects. To add a vendor, product or project to this list, read the content guide before submitting a change. More information.

If you are running a Kubernetes cluster on your own hardware or a different cloud provider, consult your documentation for security best practices. Here are links to some of the popular cloud providers' security documentation:

IaaS Provider	Link
Alibaba Cloud	https://www.alibabacloud.com/trust-center
Amazon Web Services	https://aws.amazon.com/security
Google Cloud Platform	https://cloud.google.com/security
Huawei Cloud	https://www.huaweicloud.com/intl/en-us/securecenter/overallsafety
IBM Cloud	https://www.ibm.com/cloud/security
Microsoft Azure	https://docs.microsoft.com/en-us/azure/security/azure-security
Oracle Cloud Infrastructure	https://www.oracle.com/security
Tencent Cloud	https://www.tencentcloud.com/solutions/data-security-and-information-protection
VMware vSphere	https://www.vmware.com/solutions/security/hardening-guides

Policies

You can define security policies using Kubernetes-native mechanisms, such as NetworkPolicy (declarative control over network packet filtering) or ValidatingAdmissionPolicy (declarative restrictions on what changes someone can make using the Kubernetes API).

However, you can also rely on policy implementations from the wider ecosystem around Kubernetes. Kubernetes provides extension mechanisms to let those ecosystem projects implement their own policy controls on source code review, container image approval, API access controls, networking, and more.

For more information about policy mechanisms and Kubernetes, read Policies.

What's next

Learn about related Kubernetes security topics:

- Securing your cluster
- Known vulnerabilities in Kubernetes (and links to further information)
- Data encryption in transit for the control plane
- Data encryption at rest
- Controlling Access to the Kubernetes API
- Network policies for Pods
- Secrets in Kubernetes
- Pod security standards
- RuntimeClasses

Learn the context:

- Cloud Native Security and Kubernetes

Get certified:

- Certified Kubernetes Security Specialist certification and official training course.

Read more in this section:

- Pod Security Standards
- Pod Security Admission
- Service Accounts
- Pod Security Policies
- Security For Linux Nodes
- Security For Windows Nodes
- Controlling Access to the Kubernetes API
- Role Based Access Control Good Practices
- Good practices for Kubernetes Secrets
- Multi-tenancy
- Hardening Guide - Authentication Mechanisms
- Hardening Guide - Scheduler Configuration
- Kubernetes API Server Bypass Risks
- Linux kernel security constraints for Pods and containers
- Security Checklist
- Application Security Checklist

Storage Classes

This document describes the concept of a StorageClass in Kubernetes. Familiarity with volumes and persistent volumes is suggested.

A StorageClass provides a way for administrators to describe the *classes* of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. Kubernetes itself is unopinionated about what classes represent.

The Kubernetes concept of a storage class is similar to "profiles" in some other storage system designs.

StorageClass objects

Each StorageClass contains the fields `provisioner`, `parameters`, and `reclaimPolicy`, which are used when a PersistentVolume belonging to the class needs to be dynamically provisioned to satisfy a PersistentVolumeClaim (PVC).

The name of a StorageClass object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating StorageClass objects.

As an administrator, you can specify a default StorageClass that applies to any PVCs that don't request a specific class. For more details, see the PersistentVolumeClaim concept.

Here's an example of a StorageClass:

`storage/storageclass-low-latency.yaml` Copy storage/storageclass-low-latency.yaml to clipboard

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: low-latency  annotations:    storageclass.kubernetes.io/is-default-class: "false"provisioner: c
```

Default StorageClass

You can mark a StorageClass as the default for your cluster. For instructions on setting the default StorageClass, see Change the default StorageClass.

When a PVC does not specify a `storageClassName`, the default StorageClass is used.

If you set the `storageclass.kubernetes.io/is-default-class` annotation to true on more than one StorageClass in your cluster, and you then create a PersistentVolumeClaim with no `storageClassName` set, Kubernetes uses the most recently created default StorageClass.

Note:

You should try to only have one StorageClass in your cluster that is marked as the default. The reason that Kubernetes allows you to have multiple default StorageClasses is to allow for seamless migration.

You can create a PersistentVolumeClaim without specifying a `storageClassName` for the new PVC, and you can do so even when no default StorageClass exists in your cluster. In this case, the new PVC creates as you defined it, and the `storageClassName` of that PVC remains unset until a default becomes available.

You can have a cluster without any default StorageClass. If you don't mark any StorageClass as default (and one hasn't been set for you by, for example, a cloud provider), then Kubernetes cannot apply that defaulting for PersistentVolumeClaims that need it.

If or when a default StorageClass becomes available, the control plane identifies any existing PVCs without `storageClassName`. For the PVCs that either have an empty value for `storageClassName` or do not have this key, the control plane then updates those PVCs to set `storageClassName` to match the new default StorageClass. If you have an existing PVC where the `storageClassName` is `""`, and you configure a default StorageClass, then this PVC will not get updated.

In order to keep binding to PVs with `storageClassName` set to `""` (while a default StorageClass is present), you need to set the `storageClassName` of the associated PVC to `""`.

Provisioner

Each StorageClass has a provisioner that determines what volume plugin is used for provisioning PVs. This field must be specified.

Volume Plugin	Internal Provisioner	Config Example
AzureFile	✓	Azure File
CephFS	-	-
FC	-	-
FlexVolume	-	-
iSCSI	-	-
Local	-	Local
NFS	-	NFS
PortworxVolume	✓	Portworx Volume
RBD	-	Ceph RBD
VsphereVolume	✓	vSphere

You are not restricted to specifying the "internal" provisioners listed here (whose names are prefixed with "kubernetes.io" and shipped alongside Kubernetes). You can also run and specify external provisioners, which are independent programs that follow a specification defined by Kubernetes. Authors of external provisioners have full discretion over where their code lives, how the provisioner is shipped, how it needs to be run, what volume plugin it uses (including Flex), etc. The repository kubernetes-sigs/sig-storage-lib-external-provisioner houses a library for writing external provisioners that implements the bulk of the specification. Some external provisioners are listed under the repository kubernetes-sigs/sig-storage-lib-external-provisioner.

For example, NFS doesn't provide an internal provisioner, but an external provisioner can be used. There are also cases when 3rd party storage vendors provide their own external provisioner.

Reclaim policy

PersistentVolumes that are dynamically created by a StorageClass will have the reclaim policy specified in the `reclaimPolicy` field of the class, which can be either `Delete` or `Retain`. If no `reclaimPolicy` is specified when a StorageClass object is created, it will default to `Delete`.

PersistentVolumes that are created manually and managed via a StorageClass will have whatever reclaim policy they were assigned at creation.

Volume expansion

PersistentVolumes can be configured to be expandable. This allows you to resize the volume by editing the corresponding PVC object, requesting a new larger amount of storage.

The following types of volumes support volume expansion, when the underlying StorageClass has the field `allowVolumeExpansion` set to true.

Volume type	Required Kubernetes version for volume expansion
Azure File	1.11
CSI	1.24
FlexVolume	1.13
Portworx	1.11
rbd	1.11

Note:

You can only use the volume expansion feature to grow a Volume, not to shrink it.

Mount options

PersistentVolumes that are dynamically created by a StorageClass will have the mount options specified in the `mountOptions` field of the class.

If the volume plugin does not support mount options but mount options are specified, provisioning will fail. Mount options are **not** validated on either the class or PV. If a mount option is invalid, the PV mount fails.

Volume binding mode

The `volumeBindingMode` field controls when volume binding and dynamic provisioning should occur. When unset, `Immediate` mode is used by default.

The `Immediate` mode indicates that volume binding and dynamic provisioning occurs once the PersistentVolumeClaim is created. For storage backends that are topology-constrained and not globally accessible from all Nodes in the cluster, PersistentVolumes will be bound or provisioned without knowledge of the Pod's scheduling requirements. This may result in unschedulable Pods.

A cluster administrator can address this issue by specifying the `WaitForFirstConsumer` mode which will delay the binding and provisioning of a PersistentVolume until a Pod using the PersistentVolumeClaim is created. PersistentVolumes will be selected or provisioned conforming to the topology that is specified by the Pod's scheduling constraints. These include, but are not limited to, resource requirements, node selectors, pod affinity and anti-affinity, and taints and tolerations.

The following plugins support `WaitForFirstConsumer` with dynamic provisioning:

- CSI volumes, provided that the specific CSI driver supports this

The following plugins support `WaitForFirstConsumer` with pre-created PersistentVolume binding:

- CSI volumes, provided that the specific CSI driver supports this
- `local`

Note:

If you choose to use `WaitForFirstConsumer`, do not use `nodeName` in the Pod spec to specify node affinity. If `nodeName` is used in this case, the scheduler will be bypassed and PVC will remain in `pending` state.

Instead, you can use node selector for `kubernetes.io/hostname`:

`storage/storageclass/pod-volume-binding.yaml` Copy storage/storageclass/pod-volume-binding.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: task-pv-podspec:  nodeSelector:    kubernetes.io/hostname: kube-01  volumes:    - name: task-pv-storage
```

Allowed topologies

When a cluster operator specifies the `WaitForFirstConsumer` volume binding mode, it is no longer necessary to restrict provisioning to specific topologies in most situations. However, if still required, `allowedTopologies` can be specified.

This example demonstrates how to restrict the topology of provisioned volumes to specific zones and should be used as a replacement for the `zone` and `zones` parameters for the supported plugins.

`storage/storageclass/storageclass-topology.yaml` Copy storage/storageclass/storageclass-topology.yaml to clipboard

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: standardprovisioner:  example.com/exampleparameters:  type: pd-standardvolumeBindingMode: WaitF
```

Parameters

StorageClasses have parameters that describe volumes belonging to the storage class. Different parameters may be accepted depending on the `provisioner`. When a parameter is omitted, some default is used.

There can be at most 512 parameters defined for a StorageClass. The total length of the parameters object including its keys and values cannot exceed 256 KiB.

AWS EBS

Kubernetes 1.34 does not include a `awsElasticBlockStore` volume type.

The AWSElasticBlockStore in-tree storage driver was deprecated in the Kubernetes v1.19 release and then removed entirely in the v1.27 release.

The Kubernetes project suggests that you use the AWS EBS out-of-tree storage driver instead.

Here is an example StorageClass for the AWS EBS CSI driver:

`storage/storageclass/storageclass-aws-ebs.yaml` Copy storage/storageclass/storageclass-aws-ebs.yaml to clipboard

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: ebs-scprovisioner: ebs.csi.aws.comvolumeBindingMode: WaitForFirstConsumerparameters:  csi.stora
```

`tagSpecification`: Tags with this prefix are applied to dynamically provisioned EBS volumes.

AWS EFS

To configure AWS EFS storage, you can use the out-of-tree AWS_EFS_CSI_DRIVER.

`storage/storageclass/storageclass-aws-efs.yaml` Copy storage/storageclass/storageclass-aws-efs.yaml to clipboard

```
kind: StorageClass
apiVersion: storage.k8s.io/v1metadata:  name: efs-scprovisioner: efs.csi.aws.comparameters:  provisioningMode: efs-ap  fileSystemI
```

- `provisioningMode`: The type of volume to be provisioned by Amazon EFS. Currently, only access point based provisioning is supported (`efs-ap`).
- `fileSystemId`: The file system under which the access point is created.
- `directoryPerms`: The directory permissions of the root directory created by the access point.

For more details, refer to the AWS_EFS_CSI_Driver Dynamic Provisioning documentation.

NFS

To configure NFS storage, you can use the in-tree driver or the NFS CSI driver for Kubernetes (recommended).

`storage/storageclass/storageclass-nfs.yaml` Copy storage/storageclass/storageclass-nfs.yaml to clipboard

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: example-nfsprovisioner: example.com/external-nfsparameters:  server: nfs-server.example.com  pa
```

- `server`: Server is the hostname or IP address of the NFS server.
- `path`: Path that is exported by the NFS server.
- `readOnly`: A flag indicating whether the storage will be mounted as read only (default false).

Kubernetes doesn't include an internal NFS provisioner. You need to use an external provisioner to create a StorageClass for NFS. Here are some examples:

- NFS Ganesha server and external provisioner
- NFS subdir external provisioner

vSphere

There are two types of provisioners for vSphere storage classes:

- CSI provisioner: `csi.vsphere.vmware.com`
- vCP provisioner: `kubernetes.io/vsphere-volume`

In-tree provisioners are deprecated. For more information on the CSI provisioner, see Kubernetes vSphere CSI Driver and vSphereVolume CSI migration.

CSI Provisioner

The vSphere CSI StorageClass provisioner works with Tanzu Kubernetes clusters. For an example, refer to the vSphere CSI repository.

vCP Provisioner

The following examples use the VMware Cloud Provider (vCP) StorageClass provisioner.

1. Create a StorageClass with a user specified disk format.

   ```
   apiVersion: storage.k8s.io/v1
   kind: StorageClassmetadata:  name: fastprovisioner: kubernetes.io/vsphere-volumeparameters:  diskformat: zeroedthick
   ```

   `diskformat`: `thin`, `zeroedthick` and `eagerzeroedthick`. Default: `"thin"`.

2. Create a StorageClass with a disk format on a user specified datastore.

   ```
   apiVersion: storage.k8s.io/v1
   kind: StorageClassmetadata:  name: fastprovisioner: kubernetes.io/vsphere-volumeparameters:  diskformat: zeroedthick  datasto
   ```

   `datastore`: The user can also specify the datastore in the StorageClass. The volume will be created on the datastore specified in the StorageClass, which in this case is `VSANDatastore`. This field is optional. If the datastore is not specified, then the volume will be created on the datastore specified in the vSphere config file used to initialize the vSphere Cloud Provider.

3. Storage Policy Management inside kubernetes

   - Using existing vCenter SPBM policy

     One of the most important features of vSphere for Storage Management is policy based Management. Storage Policy Based Management (SPBM) is a storage policy framework that provides a single unified control plane across a broad range of data services and storage solutions. SPBM enables vSphere administrators to overcome upfront storage provisioning challenges, such as capacity planning, differentiated service levels and managing capacity headroom.

     The SPBM policies can be specified in the StorageClass using the `storagePolicyName` parameter.

   - Virtual SAN policy support inside Kubernetes

     Vsphere Infrastructure (VI) Admins will have the ability to specify custom Virtual SAN Storage Capabilities during dynamic volume provisioning. You can now define storage requirements, such as performance and availability, in the form of storage capabilities during dynamic volume provisioning. The storage capability requirements are converted into a Virtual SAN policy which are then pushed down to the Virtual SAN layer when a persistent volume (virtual disk) is being created. The virtual disk is distributed across the Virtual SAN datastore to meet the requirements.

You can see Storage Policy Based Management for dynamic provisioning of volumes for more details on how to use storage policies for persistent volumes management.

Ceph RBD (deprecated)

Note:

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

This internal provisioner of Ceph RBD is deprecated. Please use CephFS RBD CSI driver.

storage/storageclass/storageclass-ceph-rbd.yaml Copy storage/storageclass/storageclass-ceph-rbd.yaml to clipboard

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: fastprovisioner: kubernetes.io/rbd # This provisioner is deprecatedparameters:  monitors: 198.1
```

- `monitors`: Ceph monitors, comma delimited. This parameter is required.
- `adminId`: Ceph client ID that is capable of creating images in the pool. Default is "admin".
- `adminSecretName`: Secret Name for `adminId`. This parameter is required. The provided secret must have type "kubernetes.io/rbd".
- `adminSecretNamespace`: The namespace for `adminSecretName`. Default is "default".
- `pool`: Ceph RBD pool. Default is "rbd".
- `userId`: Ceph client ID that is used to map the RBD image. Default is the same as `adminId`.
- `userSecretName`: The name of Ceph Secret for `userId` to map RBD image. It must exist in the same namespace as PVCs. This parameter is required. The provided secret must have type "kubernetes.io/rbd", for example created in this way:

  ```
  kubectl create secret generic ceph-secret --type="kubernetes.io/rbd" \
    --from-literal=key='QVFEQ1pMdFhPUnQrSmhBQUFYaERWNHJsZ3BsMmNjcDR6RFZST0E9PQ==' \  --namespace=kube-system
  ```

- `userSecretNamespace`: The namespace for `userSecretName`.
- `fsType`: fsType that is supported by kubernetes. Default: `"ext4"`.
- `imageFormat`: Ceph RBD image format, "1" or "2". Default is "2".
- `imageFeatures`: This parameter is optional and should only be used if you set `imageFormat` to "2". Currently supported features are `layering` only. Default is "", and no features are turned on.

Azure Disk

Kubernetes 1.34 does not include a `azureDisk` volume type.

The `azureDisk` in-tree storage driver was deprecated in the Kubernetes v1.19 release and then removed entirely in the v1.27 release.

The Kubernetes project suggests that you use the Azure Disk third party storage driver instead.

Azure File (deprecated)

storage/storageclass/storageclass-azure-file.yaml Copy storage/storageclass/storageclass-azure-file.yaml to clipboard

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: azurefileprovisioner: kubernetes.io/azure-fileparameters:  skuName: Standard_LRS  location: eas
```

- `skuName`: Azure storage account SKU tier. Default is empty.
- `location`: Azure storage account location. Default is empty.
- `storageAccount`: Azure storage account name. Default is empty. If a storage account is not provided, all storage accounts associated with the resource group are searched to find one that matches `skuName` and `location`. If a storage account is provided, it must reside in the same resource group as the cluster, and `skuName` and `location` are ignored.
- `secretNamespace`: the namespace of the secret that contains the Azure Storage Account Name and Key. Default is the same as the Pod.
- `secretName`: the name of the secret that contains the Azure Storage Account Name and Key. Default is `azure-storage-account-<accountName>-secret`
- `readOnly`: a flag indicating whether the storage will be mounted as read only. Defaults to false which means a read/write mount. This setting will impact the `ReadOnly` setting in VolumeMounts as well.

During storage provisioning, a secret named by `secretName` is created for the mounting credentials. If the cluster has enabled both RBAC and Controller Roles, add the `create` permission of resource `secret` for clusterrole `system:controller:persistent-volume-binder`.

In a multi-tenancy context, it is strongly recommended to set the value for `secretNamespace` explicitly, otherwise the storage account credentials may be read by other users.

Portworx volume (deprecated)

storage/storageclass/storageclass-portworx-volume.yaml Copy storage/storageclass/storageclass-portworx-volume.yaml to clipboard

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: portworx-io-priority-highprovisioner: kubernetes.io/portworx-volume # This provisioner is depre
```

- `fs`: filesystem to be laid out: `none/xfs/ext4` (default: `ext4`).
- `block_size`: block size in Kbytes (default: `32`).

- `repl`: number of synchronous replicas to be provided in the form of replication factor `1..3` (default: `1`) A string is expected here i.e. `"1"` and not `1`.
- `priority_io`: determines whether the volume will be created from higher performance or a lower priority storage `high/medium/low` (default: `low`).
- `snap_interval`: clock/time interval in minutes for when to trigger snapshots. Snapshots are incremental based on difference with the prior snapshot, 0 disables snaps (default: `0`). A string is expected here i.e. `"70"` and not `70`.
- `aggregation_level`: specifies the number of chunks the volume would be distributed into, 0 indicates a non-aggregated volume (default: `0`). A string is expected here i.e. `"0"` and not `0`
- `ephemeral`: specifies whether the volume should be cleaned-up after unmount or should be persistent. `emptyDir` use case can set this value to true and `persistent volumes` use case such as for databases like Cassandra should set to false, `true/false` (default `false`). A string is expected here i.e. `"true"` and not `true`.

Local

storage/storageclass/storageclass-local.yaml Copy storage/storageclass/storageclass-local.yaml to clipboard

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: local-storageprovisioner: kubernetes.io/no-provisioner # indicates that this StorageClass does
```

Local volumes do not support dynamic provisioning in Kubernetes 1.34; however a StorageClass should still be created to delay volume binding until a Pod is actually scheduled to the appropriate node. This is specified by the `WaitForFirstConsumer` volume binding mode.

Delaying volume binding allows the scheduler to consider all of a Pod's scheduling constraints when choosing an appropriate PersistentVolume for a PersistentVolumeClaim.

Pod Security Policies

Removed feature

PodSecurityPolicy was deprecated in Kubernetes v1.21, and removed from Kubernetes in v1.25.

Instead of using PodSecurityPolicy, you can enforce similar restrictions on Pods using either or both:

- Pod Security Admission
- a 3rd party admission plugin, that you deploy and configure yourself

For a migration guide, see Migrate from PodSecurityPolicy to the Built-In PodSecurity Admission Controller. For more information on the removal of this API, see PodSecurityPolicy Deprecation: Past, Present, and Future.

If you are not running Kubernetes v1.34, check the documentation for your version of Kubernetes.

Service Internal Traffic Policy

If two Pods in your cluster want to communicate, and both Pods are actually running on the same node, use *Service Internal Traffic Policy* to keep network traffic within that node. Avoiding a round trip via the cluster network can help with reliability, performance (network latency and throughput), or cost. FEATURE STATE: `Kubernetes v1.26 [stable]`

Service Internal Traffic Policy enables internal traffic restrictions to only route internal traffic to endpoints within the node the traffic originated from. The "internal" traffic here refers to traffic originated from Pods in the current cluster. This can help to reduce costs and improve performance.

Using Service Internal Traffic Policy

You can enable the internal-only traffic policy for a Service, by setting its `.spec.internalTrafficPolicy` to `Local`. This tells kube-proxy to only use node local endpoints for cluster internal traffic.

Note:

For pods on nodes with no endpoints for a given Service, the Service behaves as if it has zero endpoints (for Pods on this node) even if the service does have endpoints on other nodes.

The following example shows what a Service looks like when you set `.spec.internalTrafficPolicy` to `Local`:

```
apiVersion: v1
kind: Servicemetadata:  name: my-servicespec:  selector:    app.kubernetes.io/name: MyApp  ports:    - protocol: TCP      port: 80
```

How it works

The kube-proxy filters the endpoints it routes to based on the `spec.internalTrafficPolicy` setting. When it's set to `Local`, only node local endpoints are considered. When it's `Cluster` (the default), or is not set, Kubernetes considers all endpoints.

What's next

- Read about Topology Aware Routing
- Read about Service External Traffic Policy
- Follow the Connecting Applications with Services tutorial

DNS for Services and Pods

Your workload can discover Services within your cluster using DNS; this page explains how that works.

Kubernetes creates DNS records for Services and Pods. You can contact Services with consistent DNS names instead of IP addresses.

Kubernetes publishes information about Pods and Services which is used to program DNS. kubelet configures Pods' DNS so that running containers can look up Services by name rather than IP.

Services defined in the cluster are assigned DNS names. By default, a client Pod's DNS search list includes the Pod's own namespace and the cluster's default domain.

Namespaces of Services

A DNS query may return different results based on the namespace of the Pod making it. DNS queries that don't specify a namespace are limited to the Pod's namespace. Access Services in other namespaces by specifying it in the DNS query.

For example, consider a Pod in a `test` namespace. A `data` Service is in the `prod` namespace.

A query for `data` returns no results, because it uses the Pod's `test` namespace.

A query for `data.prod` returns the intended result, because it specifies the namespace.

DNS queries may be expanded using the Pod's `/etc/resolv.conf`. kubelet configures this file for each Pod. For example, a query for just `data` may be expanded to `data.test.svc.cluster.local`. The values of the `search` option are used to expand queries. To learn more about DNS queries, see the `resolv.conf` manual page.

```
nameserver 10.32.0.10
search <namespace>.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

In summary, a Pod in the *test* namespace can successfully resolve either `data.prod` or `data.prod.svc.cluster.local`.

DNS Records

What objects get DNS records?

1. Services
2. Pods

The following sections detail the supported DNS record types and layout that is supported. Any other layout or names or queries that happen to work are considered implementation details and are subject to change without warning. For more up-to-date specification, see Kubernetes DNS-Based Service Discovery.

Services

A/AAAA records

"Normal" (not headless) Services are assigned DNS A and/or AAAA records, depending on the IP family or families of the Service, with a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. This resolves to the cluster IP of the Service.

Headless Services (without a cluster IP) are also assigned DNS A and/or AAAA records, with a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. Unlike normal Services, this resolves to the set of IPs of all of the Pods selected by the Service. Clients are expected to consume the set or else use standard round-robin selection from the set.

SRV records

SRV Records are created for named ports that are part of normal or headless services.

- For each named port, the SRV record has the form `_port-name._port-protocol.my-svc.my-namespace.svc.cluster-domain.example`.
- For a regular Service, this resolves to the port number and the domain name: `my-svc.my-namespace.svc.cluster-domain.example`.
- For a headless Service, this resolves to multiple answers, one for each Pod that is backing the Service, and contains the port number and the domain name of the Pod of the form `hostname.my-svc.my-namespace.svc.cluster-domain.example`.

Pods

A/AAAA records

Kube-DNS versions, prior to the implementation of the DNS specification, had the following DNS resolution:

`<pod-IPv4-address>.<namespace>.pod.<cluster-domain>`

For example, if a Pod in the `default` namespace has the IP address 172.17.0.3, and the domain name for your cluster is `cluster.local`, then the Pod has a DNS name:

`172-17-0-3.default.pod.cluster.local`

Some cluster DNS mechanisms, like CoreDNS, also provide `A` records for:

`<pod-ipv4-address>.<service-name>.<my-namespace>.svc.<cluster-domain.example>`

For example, if a Pod in the `cafe` namespace has the IP address 172.17.0.3, is an endpoint of a Service named `barista`, and the domain name for your cluster is `cluster.local`, then the Pod would have this service-scoped DNS `A` record.

```
172-17-0-3.barista.cafe.svc.cluster.local
```

Pod's hostname and subdomain fields

Currently when a Pod is created, its hostname (as observed from within the Pod) is the Pod's `metadata.name` value.

The Pod spec has an optional `hostname` field, which can be used to specify a different hostname. When specified, it takes precedence over the Pod's name to be the hostname of the Pod (again, as observed from within the Pod). For example, given a Pod with `spec.hostname` set to `"my-host"`, the Pod will have its hostname set to `"my-host"`.

The Pod spec also has an optional `subdomain` field which can be used to indicate that the pod is part of sub-group of the namespace. For example, a Pod with `spec.hostname` set to `"foo"`, and `spec.subdomain` set to `"bar"`, in namespace `"my-namespace"`, will have its hostname set to `"foo"` and its fully qualified domain name (FQDN) set to `"foo.bar.my-namespace.svc.cluster.local"` (once more, as observed from within the Pod).

If there exists a headless Service in the same namespace as the Pod, with the same name as the subdomain, the cluster's DNS Server also returns A and/or AAAA records for the Pod's fully qualified hostname.

Example:

```
apiVersion: v1
kind: Servicemetadata:  name: busybox-subdomainspec:  selector:    name: busybox  clusterIP: None  ports:  - name: foo # name is n
```

Given the above Service `"busybox-subdomain"` and the Pods which set `spec.subdomain` to `"busybox-subdomain"`, the first Pod will see its own FQDN as `"busybox-1.busybox-subdomain.my-namespace.svc.cluster-domain.example"`. DNS serves A and/or AAAA records at that name, pointing to the Pod's IP. Both Pods `"busybox1"` and `"busybox2"` will have their own address records.

An EndpointSlice can specify the DNS hostname for any endpoint addresses, along with its IP.

Note:

A and AAAA records are not created for Pod names since `hostname` is missing for the Pod. A Pod with no `hostname` but with `subdomain` will only create the A or AAAA record for the headless Service (`busybox-subdomain.my-namespace.svc.cluster-domain.example`), pointing to the Pods' IP addresses. Also, the Pod needs to be ready in order to have a record unless `publishNotReadyAddresses=True` is set on the Service.

Pod's setHostnameAsFQDN field

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

When a Pod is configured to have fully qualified domain name (FQDN), its hostname is the short hostname. For example, if you have a Pod with the fully qualified domain name `busybox-1.busybox-subdomain.my-namespace.svc.cluster-domain.example`, then by default the `hostname` command inside that Pod returns `busybox-1` and the `hostname --fqdn` command returns the FQDN.

When you set `setHostnameAsFQDN: true` in the Pod spec, the kubelet writes the Pod's FQDN into the hostname for that Pod's namespace. In this case, both `hostname` and `hostname --fqdn` return the Pod's FQDN.

Note:

In Linux, the hostname field of the kernel (the `nodename` field of `struct utsname`) is limited to 64 characters.

If a Pod enables this feature and its FQDN is longer than 64 character, it will fail to start. The Pod will remain in `Pending` status (`ContainerCreating` as seen by `kubectl`) generating error events, such as Failed to construct FQDN from Pod hostname and cluster domain, FQDN `long-FQDN` is too long (64 characters is the max, 70 characters requested). One way of improving user experience for this scenario is to create an admission webhook controller to control FQDN size when users create top level objects, for example, Deployment.

Pod's DNS Policy

DNS policies can be set on a per-Pod basis. Currently Kubernetes supports the following Pod-specific DNS policies. These policies are specified in the `dnsPolicy` field of a Pod Spec.

- "`Default`": The Pod inherits the name resolution configuration from the node that the Pods run on. See related discussion for more details.
- "`ClusterFirst`": Any DNS query that does not match the configured cluster domain suffix, such as "`www.kubernetes.io`", is forwarded to an upstream nameserver by the DNS server. Cluster administrators may have extra stub-domain and upstream DNS servers configured. See related discussion for details on how DNS queries are handled in those cases.
- "`ClusterFirstWithHostNet`": For Pods running with hostNetwork, you should explicitly set its DNS policy to "`ClusterFirstWithHostNet`". Otherwise, Pods running with hostNetwork and `"ClusterFirst"` will fallback to the behavior of the `"Default"` policy.

  Note:

  This is not supported on Windows. See below for details.
- "`None`": It allows a Pod to ignore DNS settings from the Kubernetes environment. All DNS settings are supposed to be provided using the `dnsConfig` field in the Pod Spec. See Pod's DNS config subsection below.

Note:

"Default" is not the default DNS policy. If `dnsPolicy` is not explicitly specified, then "ClusterFirst" is used.

The example below shows a Pod with its DNS policy set to "`ClusterFirstWithHostNet`" because it has `hostNetwork` set to `true`.

```
apiVersion: v1
kind: Podmetadata:  name: busybox  namespace: defaultspec:  containers:  - image: busybox:1.28    command:      - sleep      - "36
```

Pod's DNS Config

FEATURE STATE: `Kubernetes v1.14 [stable]`

Pod's DNS Config allows users more control on the DNS settings for a Pod.

The `dnsConfig` field is optional and it can work with any `dnsPolicy` settings. However, when a Pod's `dnsPolicy` is set to "`None`", the `dnsConfig` field has to be specified.

Below are the properties a user can specify in the `dnsConfig` field:

- `nameservers`: a list of IP addresses that will be used as DNS servers for the Pod. There can be at most 3 IP addresses specified. When the Pod's `dnsPolicy` is set to "`None`", the list must contain at least one IP address, otherwise this property is optional. The servers listed will be combined to the base nameservers generated from the specified DNS policy with duplicate addresses removed.
- `searches`: a list of DNS search domains for hostname lookup in the Pod. This property is optional. When specified, the provided list will be merged into the base search domain names generated from the chosen DNS policy. Duplicate domain names are removed. Kubernetes allows up to 32 search domains.
- `options`: an optional list of objects where each object may have a `name` property (required) and a `value` property (optional). The contents in this property will be merged to the options generated from the specified DNS policy. Duplicate entries are removed.

The following is an example Pod with custom DNS settings:

`service/networking/custom-dns.yaml` Copy service/networking/custom-dns.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  namespace: default  name: dns-examplespec:  containers:    - name: test      image: nginx  dnsPolicy: "None"  
```

When the Pod above is created, the container `test` gets the following contents in its `/etc/resolv.conf` file:

```
nameserver 192.0.2.1
search ns1.svc.cluster-domain.example my.dns.search.suffix
options ndots:2 edns0
```

For IPv6 setup, search path and name server should be set up like this:

```
kubectl exec -it dns-example -- cat /etc/resolv.conf
```

The output is similar to this:

```
nameserver 2001:db8:30::a
search default.svc.cluster-domain.example svc.cluster-domain.example cluster-domain.example
options ndots:5
```

DNS search domain list limits

FEATURE STATE: `Kubernetes 1.28 [stable]`

Kubernetes itself does not limit the DNS Config until the length of the search domain list exceeds 32 or the total length of all search domains exceeds 2048. This limit applies to the node's resolver configuration file, the Pod's DNS Config, and the merged DNS Config respectively.

Note:

Some container runtimes of earlier versions may have their own restrictions on the number of DNS search domains. Depending on the container runtime environment, the pods with a large number of DNS search domains may get stuck in the pending state.

It is known that containerd v1.5.5 or earlier and CRI-O v1.21 or earlier have this problem.

DNS resolution on Windows nodes

- `ClusterFirstWithHostNet` is not supported for Pods that run on Windows nodes. Windows treats all names with a `.` as a FQDN and skips FQDN resolution.
- On Windows, there are multiple DNS resolvers that can be used. As these come with slightly different behaviors, using the `Resolve-DNSName` powershell cmdlet for name query resolutions is recommended.
- On Linux, you have a DNS suffix list, which is used after resolution of a name as fully qualified has failed. On Windows, you can only have 1 DNS suffix, which is the DNS suffix associated with that Pod's namespace (example: `mydns.svc.cluster.local`). Windows can resolve FQDNs, Services, or network name which can be resolved with this single suffix. For example, a Pod spawned in the `default` namespace, will have the DNS suffix `default.svc.cluster.local`. Inside a Windows Pod, you can resolve both `kubernetes.default.svc.cluster.local` and `kubernetes`, but not the partially qualified names (`kubernetes.default` or `kubernetes.default.svc`).

What's next

For guidance on administering DNS configurations, check Configure DNS Service.

Pod Overhead

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

When you run a Pod on a Node, the Pod itself takes an amount of system resources. These resources are additional to the resources needed to run the container(s) inside the Pod. In Kubernetes, *Pod Overhead* is a way to account for the resources consumed by the Pod infrastructure on top of the container requests & limits.

In Kubernetes, the Pod's overhead is set at admission time according to the overhead associated with the Pod's RuntimeClass.

A pod's overhead is considered in addition to the sum of container resource requests when scheduling a Pod. Similarly, the kubelet will include the Pod overhead when sizing the Pod cgroup, and when carrying out Pod eviction ranking.

Configuring Pod overhead

You need to make sure a `RuntimeClass` is utilized which defines the `overhead` field.

Usage example

To work with Pod overhead, you need a RuntimeClass that defines the `overhead` field. As an example, you could use the following RuntimeClass definition with a virtualization container runtime (in this example, Kata Containers combined with the Firecracker virtual machine monitor) that uses around 120MiB per Pod for the virtual machine and the guest OS:

```
# You need to change this example to match the actual runtime name, and per-Pod
# resource overhead, that the container runtime is adding in your cluster.apiVersion: node.k8s.io/v1kind: RuntimeClassmetadata:  na
```

Workloads which are created which specify the `kata-fc` RuntimeClass handler will take the memory and cpu overheads into account for resource quota calculations, node scheduling, as well as Pod cgroup sizing.

Consider running the given example workload, test-pod:

```
apiVersion: v1
kind: Podmetadata:  name: test-podspec:  runtimeClassName: kata-fc  containers:  - name: busybox-ctr    image: busybox:1.28    std
```

Note:

If only `limits` are specified in the pod definition, kubelet will deduce `requests` from those limits and set them to be the same as the defined `limits`.

At admission time the RuntimeClass admission controller updates the workload's PodSpec to include the `overhead` as described in the RuntimeClass. If the PodSpec already has this field defined, the Pod will be rejected. In the given example, since only the RuntimeClass name is specified, the admission controller mutates the Pod to include an `overhead`.

After the RuntimeClass admission controller has made modifications, you can check the updated Pod overhead value:

```
kubectl get pod test-pod -o jsonpath='{.spec.overhead}'
```

The output is:

```
map[cpu:250m memory:120Mi]
```

If a ResourceQuota is defined, the sum of container requests as well as the `overhead` field are counted.

When the kube-scheduler is deciding which node should run a new Pod, the scheduler considers that Pod's `overhead` as well as the sum of container requests for that Pod. For this example, the scheduler adds the requests and the overhead, then looks for a node that has 2.25 CPU and 320 MiB of memory available.

Once a Pod is scheduled to a node, the kubelet on that node creates a new cgroup for the Pod. It is within this pod that the underlying container runtime will create containers.

If the resource has a limit defined for each container (Guaranteed QoS or Burstable QoS with limits defined), the kubelet will set an upper limit for the pod cgroup associated with that resource (cpu.cfs_quota_us for CPU and memory.limit_in_bytes memory). This upper limit is based on the sum of the container limits plus the `overhead` defined in the PodSpec.

For CPU, if the Pod is Guaranteed or Burstable QoS, the kubelet will set `cpu.shares` based on the sum of container requests plus the `overhead` defined in the PodSpec.

Looking at our example, verify the container requests for the workload:

```
kubectl get pod test-pod -o jsonpath='{.spec.containers[*].resources.limits}'
```

The total container requests are 2000m CPU and 200MiB of memory:

```
map[cpu: 500m memory:100Mi] map[cpu:1500m memory:100Mi]
```

Check this against what is observed by the node:

```
kubectl describe node | grep test-pod -B2
```

The output shows requests for 2250m CPU, and for 320MiB of memory. The requests include Pod overhead:

```
  Namespace     Name        CPU Requests  CPU Limits   Memory Requests  Memory Limits  AGE
  ---------     ----        ------------  ----------   ---------------  -------------  ---
  default       test-pod    2250m (56%)   2250m (56%)  320Mi (1%)       320Mi (1%)     36m
```

Verify Pod cgroup limits

Check the Pod's memory cgroups on the node where the workload is running. In the following example, [crictl](#) is used on the node, which provides a CLI for CRI-compatible container runtimes. This is an advanced example to show Pod overhead behavior, and it is not expected that users should need to check cgroups directly on the node.

First, on the particular node, determine the Pod identifier:

```
# Run this on the node where the Pod is scheduled
POD_ID="$(sudo crictl pods --name test-pod -q)"
```

From this, you can determine the cgroup path for the Pod:

```
# Run this on the node where the Pod is scheduled
sudo crictl inspectp -o=json $POD_ID | grep cgroupsPath
```

The resulting cgroup path includes the Pod's `pause` container. The Pod level cgroup is one directory above.

```
  "cgroupsPath": "/kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2/7ccf55aee35dd16aca4189c952d83487297f3cd760f1bbf09620e206e7d0c2
```

In this specific case, the pod cgroup path is `kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2`. Verify the Pod level cgroup setting for memory:

```
# Run this on the node where the Pod is scheduled.
# Also, change the name of the cgroup to match the cgroup allocated for your pod.
 cat /sys/fs/cgroup/memory/kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2/memory.limit_in_bytes
```

This is 320 MiB, as expected:

```
335544320
```

Observability

Some `kube_pod_overhead_*` metrics are available in [kube-state-metrics](#) to help identify when Pod overhead is being utilized and to help observe stability of workloads running with a defined overhead.

What's next

- Learn more about [RuntimeClass](#)
- Read the [PodOverhead Design](#) enhancement proposal for extra context

---

Pod Topology Spread Constraints

You can use *topology spread constraints* to control how [Pods](#) are spread across your cluster among failure-domains such as regions, zones, nodes, and other user-defined topology domains. This can help to achieve high availability as well as efficient resource utilization.

You can set [cluster-level constraints](#) as a default, or configure topology spread constraints for individual workloads.

Motivation

Imagine that you have a cluster of up to twenty nodes, and you want to run a [workload](#) that automatically scales how many replicas it uses. There could be as few as two Pods or as many as fifteen. When there are only two Pods, you'd prefer not to have both of those Pods run on the same node: you would run the risk that a single node failure takes your workload offline.

In addition to this basic usage, there are some advanced usage examples that enable your workloads to benefit on high availability and cluster utilization.

As you scale up and run more Pods, a different concern becomes important. Imagine that you have three nodes running five Pods each. The nodes have enough capacity to run that many replicas; however, the clients that interact with this workload are split across three different datacenters (or infrastructure zones). Now you have less concern about a single node failure, but you notice that latency is higher than you'd like, and you are paying for network costs associated with sending network traffic between the different zones.

You decide that under normal operation you'd prefer to have a similar number of replicas [scheduled](#) into each infrastructure zone, and you'd like the cluster to self-heal in the case that there is a problem.

Pod topology spread constraints offer you a declarative way to configure that.

`topologySpreadConstraints` field

The Pod API includes a field, `spec.topologySpreadConstraints`. The usage of this field looks like the following:

```
---
apiVersion: v1kind: Podmetadata:  name: example-podspec:  # Configure a topology spread constraint  topologySpreadConstraints:
```

Note:

There can only be one `topologySpreadConstraint` for a given `topologyKey` and `whenUnsatisfiable` value. For example, if you have defined a `topologySpreadConstraint` that uses the `topologyKey` "kubernetes.io/hostname" and `whenUnsatisfiable` value "DoNotSchedule", you can only add another `topologySpreadConstraint` for the `topologyKey` "kubernetes.io/hostname" if you use a different `whenUnsatisfiable` value.

You can read more about this field by running `kubectl explain Pod.spec.topologySpreadConstraints` or refer to the [scheduling](#) section of the API reference for Pod.

Spread constraint definition

You can define one or multiple `topologySpreadConstraints` entries to instruct the kube-scheduler how to place each incoming Pod in relation to the existing Pods across your cluster. Those fields are:

- **maxSkew** describes the degree to which Pods may be unevenly distributed. You must specify this field and the number must be greater than zero. Its semantics differ according to the value of `whenUnsatisfiable`:

  - if you select `whenUnsatisfiable: DoNotSchedule`, then `maxSkew` defines the maximum permitted difference between the number of matching pods in the target topology and the *global minimum* (the minimum number of matching pods in an eligible domain or zero if the number of eligible domains is less than MinDomains). For example, if you have 3 zones with 2, 2 and 1 matching pods respectively, `MaxSkew` is set to 1 then the global minimum is 1.
  - if you select `whenUnsatisfiable: ScheduleAnyway`, the scheduler gives higher precedence to topologies that would help reduce the skew.

- **minDomains** indicates a minimum number of eligible domains. This field is optional. A domain is a particular instance of a topology. An eligible domain is a domain whose nodes match the node selector.

  Note:

  Before Kubernetes v1.30, the `minDomains` field was only available if the `MinDomainsInPodTopologySpread` feature gate was enabled (default since v1.28). In older Kubernetes clusters it might be explicitly disabled or the field might not be available.

  - The value of `minDomains` must be greater than 0, when specified. You can only specify `minDomains` in conjunction with `whenUnsatisfiable: DoNotSchedule`.
  - When the number of eligible domains with match topology keys is less than `minDomains`, Pod topology spread treats global minimum as 0, and then the calculation of `skew` is performed. The global minimum is the minimum number of matching Pods in an eligible domain, or zero if the number of eligible domains is less than `minDomains`.
  - When the number of eligible domains with matching topology keys equals or is greater than `minDomains`, this value has no effect on scheduling.
  - If you do not specify `minDomains`, the constraint behaves as if `minDomains` is 1.

- **topologyKey** is the key of node labels. Nodes that have a label with this key and identical values are considered to be in the same topology. We call each instance of a topology (in other words, a <key, value> pair) a domain. The scheduler will try to put a balanced number of pods into each domain. Also, we define an eligible domain as a domain whose nodes meet the requirements of nodeAffinityPolicy and nodeTaintsPolicy.

- **whenUnsatisfiable** indicates how to deal with a Pod if it doesn't satisfy the spread constraint:

  - `DoNotSchedule` (default) tells the scheduler not to schedule it.
  - `ScheduleAnyway` tells the scheduler to still schedule it while prioritizing nodes that minimize the skew.

- **labelSelector** is used to find matching Pods. Pods that match this label selector are counted to determine the number of Pods in their corresponding topology domain. See Label Selectors for more details.

- **matchLabelKeys** is a list of pod label keys to select the group of pods over which the spreading skew will be calculated. At a pod creation, the kube-apiserver uses those keys to lookup values from the incoming pod labels, and those key-value labels will be merged with any existing `labelSelector`. The same key is forbidden to exist in both `matchLabelKeys` and `labelSelector`. `matchLabelKeys` cannot be set when `labelSelector` isn't set. Keys that don't exist in the pod labels will be ignored. A null or empty list means only match against the `labelSelector`.

  Caution:

  It's not recommended to use `matchLabelKeys` with labels that might be updated directly on pods. Even if you edit the pod's label that is specified at `matchLabelKeys` **directly**, (that is, you edit the Pod and not a Deployment), kube-apiserver doesn't reflect the label update onto the merged `labelSelector`.

  With `matchLabelKeys`, you don't need to update the `pod.spec` between different revisions. The controller/operator just needs to set different values to the same label key for different revisions. For example, if you are configuring a Deployment, you can use the label keyed with pod-template-hash, which is added automatically by the Deployment controller, to distinguish between different revisions in a single Deployment.

  ```
  topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: kubernetes.io/hostname
        whenUnsatisfiable: DoNotSchedule
        labelSelector:
          matchLabels:
            app: foo
        matchLabelKeys:
          - pod-template-hash
  ```

  Note:

  The `matchLabelKeys` field is a beta-level field and enabled by default in 1.27. You can disable it by disabling the `MatchLabelKeysInPodTopologySpread` feature gate.

  Before v1.34, `matchLabelKeys` was handled implicitly. Since v1.34, key-value labels corresponding to `matchLabelKeys` are explicitly merged into `labelSelector`. You can disable it and revert to the previous behavior by disabling the `MatchLabelKeysInPodTopologySpreadSelectorMerge` feature gate of kube-apiserver.

- **nodeAffinityPolicy** indicates how we will treat Pod's nodeAffinity/nodeSelector when calculating pod topology spread skew. Options are:

  - Honor: only nodes matching nodeAffinity/nodeSelector are included in the calculations.
  - Ignore: nodeAffinity/nodeSelector are ignored. All nodes are included in the calculations.

  If this value is null, the behavior is equivalent to the Honor policy.

Note:

The `nodeAffinityPolicy` became beta in 1.26 and graduated to GA in 1.33. It's enabled by default in beta, you can disable it by disabling the `NodeInclusionPolicyInPodTopologySpread` feature gate.

- **nodeTaintsPolicy** indicates how we will treat node taints when calculating pod topology spread skew. Options are:

  - Honor: nodes without taints, along with tainted nodes for which the incoming pod has a toleration, are included.
  - Ignore: node taints are ignored. All nodes are included.

  If this value is null, the behavior is equivalent to the Ignore policy.

  Note:

  The `nodeTaintsPolicy` became beta in 1.26 and graduated to GA in 1.33. It's enabled by default in beta, you can disable it by disabling the `NodeInclusionPolicyInPodTopologySpread` feature gate.

When a Pod defines more than one `topologySpreadConstraint`, those constraints are combined using a logical AND operation: the kube-scheduler looks for a node for the incoming Pod that satisfies all the configured constraints.

Node labels

Topology spread constraints rely on node labels to identify the topology domain(s) that each node is in. For example, a node might have labels:

```
  region: us-east-1
  zone: us-east-1a
```

Note:

For brevity, this example doesn't use the well-known label keys `topology.kubernetes.io/zone` and `topology.kubernetes.io/region`. However, those registered label keys are nonetheless recommended rather than the private (unqualified) label keys `region` and `zone` that are used here.

You can't make a reliable assumption about the meaning of a private label key between different contexts.

Suppose you have a 4-node cluster with the following labels:

```
NAME    STATUS   ROLES    AGE     VERSION   LABELS
node1   Ready    <none>   4m26s   v1.16.0   node=node1,zone=zoneA
node2   Ready    <none>   3m58s   v1.16.0   node=node2,zone=zoneA
node3   Ready    <none>   3m17s   v1.16.0   node=node3,zone=zoneB
node4   Ready    <none>   2m43s   v1.16.0   node=node4,zone=zoneB
```

Then the cluster is logically viewed as below:

> graph TB subgraph "zoneB" n3(Node3) n4(Node4) end subgraph "zoneA" n1(Node1) n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4 k8s; class zoneA,zoneB cluster;

Consistency

You should set the same Pod topology spread constraints on all pods in a group.

Usually, if you are using a workload controller such as a Deployment, the pod template takes care of this for you. If you mix different spread constraints then Kubernetes follows the API definition of the field; however, the behavior is more likely to become confusing and troubleshooting is less straightforward.

You need a mechanism to ensure that all the nodes in a topology domain (such as a cloud provider region) are labeled consistently. To avoid you needing to manually label nodes, most clusters automatically populate well-known labels such as `kubernetes.io/hostname`. Check whether your cluster supports this.

Topology spread constraint examples

Example: one topology spread constraint

Suppose you have a 4-node cluster where 3 Pods labeled `foo: bar` are located in node1, node2 and node3 respectively:

> graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class zoneA,zoneB cluster;

If you want an incoming Pod to be evenly spread with existing Pods across zones, you can use a manifest similar to:

`pods/topology-spread-constraints/one-constraint.yaml` Copy pods/topology-spread-constraints/one-constraint.yaml to clipboard

```
kind: Pod
apiVersion: v1metadata:  name: mypod  labels:    foo: barspec:  topologySpreadConstraints:  - maxSkew: 1    topologyKey: zone    wl
```

From that manifest, `topologyKey: zone` implies the even distribution will only be applied to nodes that are labeled `zone: <any value>` (nodes that don't have a `zone` label are skipped). The field `whenUnsatisfiable: DoNotSchedule` tells the scheduler to let the incoming Pod stay pending if the scheduler can't find a way to satisfy the constraint.

If the scheduler placed this incoming Pod into zone `A`, the distribution of Pods would become `[3, 1]`. That means the actual skew is then 2 (calculated as `3 - 1`), which violates `maxSkew: 1`. To satisfy the constraints and context for this example, the incoming Pod can only be placed onto a node in zone `B`:

> graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) p4(mypod) --> n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;

OR

> graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) p4(mypod) --> n3 n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;

You can tweak the Pod spec to meet various kinds of requirements:

- Change `maxSkew` to a bigger value - such as `2` - so that the incoming Pod can be placed into zone `A` as well.
- Change `topologyKey` to `node` so as to distribute the Pods evenly across nodes instead of zones. In the above example, if `maxSkew` remains `1`, the incoming Pod can only be placed onto the node `node4`.
- Change `whenUnsatisfiable: DoNotSchedule` to `whenUnsatisfiable: ScheduleAnyway` to ensure the incoming Pod to be always schedulable (suppose other scheduling APIs are satisfied). However, it's preferred to be placed into the topology domain which has fewer matching Pods. (Be aware that this preference is jointly normalized with other internal scheduling priorities such as resource usage ratio).

Example: multiple topology spread constraints

This builds upon the previous example. Suppose you have a 4-node cluster where 3 existing Pods labeled `foo: bar` are located on node1, node2 and node3 respectively:

> graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;

You can combine two topology spread constraints to control the spread of Pods both by node and by zone:

`pods/topology-spread-constraints/two-constraints.yaml` Copy pods/topology-spread-constraints/two-constraints.yaml to clipboard

```
kind: Pod
apiVersion: v1metadata:   name: mypod   labels:     foo: barspec:   topologySpreadConstraints:   - maxSkew: 1    topologyKey: zone    w
```

In this case, to match the first constraint, the incoming Pod can only be placed onto nodes in zone `B`; while in terms of the second constraint, the incoming Pod can only be scheduled to the node `node4`. The scheduler only considers options that satisfy all defined constraints, so the only valid placement is onto node `node4`.

Example: conflicting topology spread constraints

Multiple constraints can lead to conflicts. Suppose you have a 3-node cluster across 2 zones:

> graph BT subgraph "zoneB" p4(Pod) --> n3(Node3) p5(Pod) --> n3 end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n1 p3(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3,p4,p5 k8s; class zoneA,zoneB cluster;

If you were to apply `two-constraints.yaml` (the manifest from the previous example) to **this** cluster, you would see that the Pod `mypod` stays in the `Pending` state. This happens because: to satisfy the first constraint, the Pod `mypod` can only be placed into zone `B`; while in terms of the second constraint, the Pod `mypod` can only schedule to node `node2`. The intersection of the two constraints returns an empty set, and the scheduler cannot place the Pod.

To overcome this situation, you can either increase the value of `maxSkew` or modify one of the constraints to use `whenUnsatisfiable: ScheduleAnyway`. Depending on circumstances, you might also decide to delete an existing Pod manually - for example, if you are troubleshooting why a bug-fix rollout is not making progress.

Interaction with node affinity and node selectors

The scheduler will skip the non-matching nodes from the skew calculations if the incoming Pod has `spec.nodeSelector` or `spec.affinity.nodeAffinity` defined.

Example: topology spread constraints with node affinity

Suppose you have a 5-node cluster ranging across zones A to C:

> graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;
>
> graph BT subgraph "zoneC" n5(Node5) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n5 k8s; class zoneC cluster;

and you know that zone `C` must be excluded. In this case, you can compose a manifest as below, so that Pod `mypod` will be placed into zone `B` instead of zone `C`. Similarly, Kubernetes also respects `spec.nodeSelector`.

`pods/topology-spread-constraints/one-constraint-with-nodeaffinity.yaml`
Copy pods/topology-spread-constraints/one-constraint-with-nodeaffinity.yaml to clipboard

```
kind: Pod
apiVersion: v1metadata:   name: mypod   labels:     foo: barspec:   topologySpreadConstraints:   - maxSkew: 1    topologyKey: zone    w
```

Implicit conventions

There are some implicit conventions worth noting here:

- Only the Pods holding the same namespace as the incoming Pod can be matching candidates.
- The scheduler only considers nodes that have all `topologySpreadConstraints[*].topologyKey` present at the same time. Nodes missing any of these `topologyKeys` are bypassed. This implies that:
  1. any Pods located on those bypassed nodes do not impact `maxSkew` calculation - in the above example, suppose the node `node1` does not have a label "zone", then the 2 Pods will be disregarded, hence the incoming Pod will be scheduled into zone `A`.
  2. the incoming Pod has no chances to be scheduled onto this kind of nodes - in the above example, suppose a node `node5` has the **mistyped** label `zone-typo: zoneC` (and no `zone` label set). After node `node5` joins the cluster, it will be bypassed and Pods for this workload aren't scheduled there.
- Be aware of what will happen if the incoming Pod's `topologySpreadConstraints[*].labelSelector` doesn't match its own labels. In the above example, if you remove the incoming Pod's labels, it can still be placed onto nodes in zone `B`, since the constraints are still satisfied. However, after that placement, the degree of imbalance of the cluster remains unchanged - it's still zone `A` having 2 Pods labeled as `foo: bar`, and zone `B` having 1 Pod labeled as `foo: bar`. If this is not what you expect, update the workload's `topologySpreadConstraints[*].labelSelector` to match the labels in the pod template.

Cluster-level default constraints

It is possible to set default topology spread constraints for a cluster. Default topology spread constraints are applied to a Pod if, and only if:

- It doesn't define any constraints in its `.spec.topologySpreadConstraints`.
- It belongs to a Service, ReplicaSet, StatefulSet or ReplicationController.

Default constraints can be set as part of the `PodTopologySpread` plugin arguments in a scheduling profile. The constraints are specified with the same API above, except that `labelSelector` must be empty. The selectors are calculated from the Services, ReplicaSets, StatefulSets or ReplicationControllers that the Pod belongs to.

An example configuration might look like follows:

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfigurationprofiles:  - schedulerName: default-scheduler    pluginConfig:      - name: PodTopologySpread
```

Built-in default constraints

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

If you don't configure any cluster-level default constraints for pod topology spreading, then kube-scheduler acts as if you specified the following default topology constraints:

```
defaultConstraints:
  - maxSkew: 3
    topologyKey: "kubernetes.io/hostname"
    whenUnsatisfiable: ScheduleAnyway
  - maxSkew: 5
    topologyKey: "topology.kubernetes.io/zone"
    whenUnsatisfiable: ScheduleAnyway
```

Also, the legacy `SelectorSpread` plugin, which provides an equivalent behavior, is disabled by default.

Note:

The `PodTopologySpread` plugin does not score the nodes that don't have the topology keys specified in the spreading constraints. This might result in a different default behavior compared to the legacy `SelectorSpread` plugin when using the default topology constraints.

If your nodes are not expected to have **both** `kubernetes.io/hostname` and `topology.kubernetes.io/zone` labels set, define your own constraints instead of using the Kubernetes defaults.

If you don't want to use the default Pod spreading constraints for your cluster, you can disable those defaults by setting `defaultingType` to `List` and leaving empty `defaultConstraints` in the `PodTopologySpread` plugin configuration:

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfigurationprofiles:  - schedulerName: default-scheduler    pluginConfig:      - name: PodTopologySpread
```

Comparison with podAffinity and podAntiAffinity

In Kubernetes, inter-Pod affinity and anti-affinity control how Pods are scheduled in relation to one another - either more packed or more scattered.

`podAffinity`
: attracts Pods; you can try to pack any number of Pods into qualifying topology domain(s).

`podAntiAffinity`
: repels Pods. If you set this to `requiredDuringSchedulingIgnoredDuringExecution` mode then only a single Pod can be scheduled into a single topology domain; if you choose `preferredDuringSchedulingIgnoredDuringExecution` then you lose the ability to enforce the constraint.

For finer control, you can specify topology spread constraints to distribute Pods across different topology domains - to achieve either high availability or cost-saving. This can also help on rolling update workloads and scaling out replicas smoothly.

For more context, see the Motivation section of the enhancement proposal about Pod topology spread constraints.

Known limitations

- There's no guarantee that the constraints remain satisfied when Pods are removed. For example, scaling down a Deployment may result in imbalanced Pods distribution.

  You can use a tool such as the Descheduler to rebalance the Pods distribution.

- Pods matched on tainted nodes are respected. See Issue 80921.

- The scheduler doesn't have prior knowledge of all the zones or other topology domains that a cluster has. They are determined from the existing nodes in the cluster. This could lead to a problem in autoscaled clusters, when a node pool (or node group) is scaled to zero nodes, and you're expecting the cluster to scale up, because, in this case, those topology domains won't be considered until there is at least one node in them.

  You can work around this by using a Node autoscaler that is aware of Pod topology spread constraints and is also aware of the overall set of topology domains.

- Pods that don't match their own labelSelector create "ghost pods". If a pod's labels don't match the `labelSelector` in its topology spread constraint, the pod won't count itself in spread calculations. This means:

  - Multiple such pods can just accumulate on the same topology (until matching pods are newly created/deleted) because those pod's schedule don't change a spreading calculation result.
  - The spreading constraint works in an unintended way, most likely not matching your expectations

  Ensure your pod's labels match the `labelSelector` in your spread constraints. Typically, a pod should match its own topology spread constraint selector.

What's next

- The blog article Introducing PodTopologySpread explains `maxSkew` in some detail, as well as covering some advanced usage examples.
- Read the scheduling section of the API reference for Pod.

Network Policies

If you want to control traffic flow at the IP address or port level (OSI layer 3 or 4), NetworkPolicies allow you to specify rules for traffic flow within your cluster, and also between Pods and the outside world. Your cluster must use a network plugin that supports NetworkPolicy enforcement.

If you want to control traffic flow at the IP address or port level for TCP, UDP, and SCTP protocols, then you might consider using Kubernetes NetworkPolicies for particular applications in your cluster. NetworkPolicies are an application-centric construct which allow you to specify how a pod is allowed to communicate with various network "entities" (we use the word "entity" here to avoid overloading the more common terms such as "endpoints" and "services", which have specific Kubernetes connotations) over the network. NetworkPolicies apply to a connection with a pod on one or both ends, and are not relevant to other connections.

The entities that a Pod can communicate with are identified through a combination of the following three identifiers:

1. Other pods that are allowed (exception: a pod cannot block access to itself)
2. Namespaces that are allowed
3. IP blocks (exception: traffic to and from the node where a Pod is running is always allowed, regardless of the IP address of the Pod or the node)

When defining a pod- or namespace-based NetworkPolicy, you use a selector to specify what traffic is allowed to and from the Pod(s) that match the selector.

Meanwhile, when IP-based NetworkPolicies are created, we define policies based on IP blocks (CIDR ranges).

Prerequisites

Network policies are implemented by the network plugin. To use network policies, you must be using a networking solution which supports NetworkPolicy. Creating a NetworkPolicy resource without a controller that implements it will have no effect.

The two sorts of pod isolation

There are two sorts of isolation for a pod: isolation for egress, and isolation for ingress. They concern what connections may be established. "Isolation" here is not absolute, rather it means "some restrictions apply". The alternative, "non-isolated for $direction", means that no restrictions apply in the stated direction. The two sorts of isolation (or not) are declared independently, and are both relevant for a connection from one pod to another.

By default, a pod is non-isolated for egress; all outbound connections are allowed. A pod is isolated for egress if there is any NetworkPolicy that both selects the pod and has "Egress" in its `policyTypes`; we say that such a policy applies to the pod for egress. When a pod is isolated for egress, the only allowed connections from the pod are those allowed by the `egress` list of some NetworkPolicy that applies to the pod for egress. Reply traffic for those allowed connections will also be implicitly allowed. The effects of those `egress` lists combine additively.

By default, a pod is non-isolated for ingress; all inbound connections are allowed. A pod is isolated for ingress if there is any NetworkPolicy that both selects the pod and has "Ingress" in its `policyTypes`; we say that such a policy applies to the pod for ingress. When a pod is isolated for ingress, the only allowed connections into the pod are those from the pod's node and those allowed by the `ingress` list of some NetworkPolicy that applies to the pod for ingress. Reply traffic for those allowed connections will also be implicitly allowed. The effects of those `ingress` lists combine additively.

Network policies do not conflict; they are additive. If any policy or policies apply to a given pod for a given direction, the connections allowed in that direction from that pod is the union of what the applicable policies allow. Thus, order of evaluation does not affect the policy result.

For a connection from a source pod to a destination pod to be allowed, both the egress policy on the source pod and the ingress policy on the destination pod need to allow the connection. If either side does not allow the connection, it will not happen.

The NetworkPolicy resource

See the NetworkPolicy reference for a full definition of the resource.

An example NetworkPolicy might look like this:

service/networking/networkpolicy.yaml Copy service/networking/networkpolicy.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicymetadata:  name: test-network-policy  namespace: defaultspec:  podSelector:    matchLabels:      role: db  poli
```

Note:

POSTing this to the API server for your cluster will have no effect unless your chosen networking solution supports network policy.

Mandatory Fields: As with all other Kubernetes config, a NetworkPolicy needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see Configure a Pod to Use a ConfigMap, and Object Management.

spec: NetworkPolicy spec has all the information needed to define a particular network policy in the given namespace.

podSelector: Each NetworkPolicy includes a `podSelector` which selects the grouping of pods to which the policy applies. The example policy selects pods with the label "role=db". An empty `podSelector` selects all pods in the namespace.

policyTypes: Each NetworkPolicy includes a `policyTypes` list which may include either `Ingress`, `Egress`, or both. The `policyTypes` field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no `policyTypes` are specified on a NetworkPolicy then by default `Ingress` will always be set and `Egress` will be set if the NetworkPolicy has any egress rules.

ingress: Each NetworkPolicy may include a list of allowed `ingress` rules. Each rule allows traffic which matches both the `from` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port, from one of three sources, the first specified via an `ipBlock`, the second via a `namespaceSelector` and the third via a `podSelector`.

egress: Each NetworkPolicy may include a list of allowed `egress` rules. Each rule allows traffic which matches both the `to` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port to any destination in `10.0.0.0/24`.

So, the example NetworkPolicy:

1. isolates `role=db` pods in the `default` namespace for both ingress and egress traffic (if they weren't already isolated)
2. (Ingress rules) allows connections to all pods in the `default` namespace with the label `role=db` on TCP port 6379 from:
   - any pod in the `default` namespace with the label `role=frontend`
   - any pod in a namespace with the label `project=myproject`
   - IP addresses in the ranges `172.17.0.0`–`172.17.0.255` and `172.17.2.0`–`172.17.255.255` (ie, all of `172.17.0.0/16` except `172.17.1.0/24`)
3. (Egress rules) allows connections from any pod in the `default` namespace with the label `role=db` to CIDR `10.0.0.0/24` on TCP port 5978

See the Declare Network Policy walkthrough for further examples.

Behavior of `to` and `from` selectors

There are four kinds of selectors that can be specified in an `ingress` `from` section or `egress` `to` section:

podSelector: This selects particular Pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations.

namespaceSelector: This selects particular namespaces for which all Pods should be allowed as ingress sources or egress destinations.

namespaceSelector *and* **podSelector**: A single `to`/`from` entry that specifies both `namespaceSelector` and `podSelector` selects particular Pods within particular namespaces. Be careful to use correct YAML syntax. For example:

```
  ...
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          user: alice
      podSelector:
        matchLabels:
          role: client
  ...
```

This policy contains a single `from` element allowing connections from Pods with the label `role=client` in namespaces with the label `user=alice`. But the following policy is different:

```
  ...
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          user: alice
    - podSelector:
        matchLabels:
          role: client
  ...
```

It contains two elements in the `from` array, and allows connections from Pods in the local Namespace with the label `role=client`, *or* from any Pod in any namespace with the label `user=alice`.

When in doubt, use `kubectl describe` to see how Kubernetes has interpreted the policy.

ipBlock: This selects particular IP CIDR ranges to allow as ingress sources or egress destinations. These should be cluster-external IPs, since Pod IPs are ephemeral and unpredictable.

Cluster ingress and egress mechanisms often require rewriting the source or destination IP of packets. In cases where this happens, it is not defined whether this happens before or after NetworkPolicy processing, and the behavior may be different for different combinations of network plugin, cloud provider, `Service` implementation, etc.

In the case of ingress, this means that in some cases you may be able to filter incoming packets based on the actual original source IP, while in other cases, the "source IP" that the NetworkPolicy acts on may be the IP of a `LoadBalancer` or of the Pod's node, etc.

For egress, this means that connections from pods to `Service` IPs that get rewritten to cluster-external IPs may or may not be subject to `ipBlock`-based policies.

Default policies

By default, if no policies exist in a namespace, then all ingress and egress traffic is allowed to and from pods in that namespace. The following examples let you change the default behavior in that namespace.

Default deny all ingress traffic

You can create a "default" ingress isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any ingress traffic to those pods.

`service/networking/network-policy-default-deny-ingress.yaml` Copy service/networking/network-policy-default-deny-ingress.yaml to clipboard

```
---
apiVersion: networking.k8s.io/v1kind: NetworkPolicymetadata:  name: default-deny-ingressspec:  podSelector: {}  policyTypes:  - In
```

This ensures that even pods that aren't selected by any other NetworkPolicy will still be isolated for ingress. This policy does not affect isolation for egress from any pod.

Allow all ingress traffic

If you want to allow all incoming connections to all pods in a namespace, you can create a policy that explicitly allows that.

`service/networking/network-policy-allow-all-ingress.yaml` Copy service/networking/network-policy-allow-all-ingress.yaml to clipboard

```
---
apiVersion: networking.k8s.io/v1kind: NetworkPolicymetadata:  name: allow-all-ingressspec:  podSelector: {}  ingress:  - {}  polic
```

With this policy in place, no additional policy or policies can cause any incoming connection to those pods to be denied. This policy has no effect on isolation for egress from any pod.

Default deny all egress traffic

You can create a "default" egress isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any egress traffic from those pods.

`service/networking/network-policy-default-deny-egress.yaml` Copy service/networking/network-policy-default-deny-egress.yaml to clipboard

```
---
apiVersion: networking.k8s.io/v1kind: NetworkPolicymetadata:  name: default-deny-egressspec:  podSelector: {}  policyTypes:  - Egr
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed egress traffic. This policy does not change the ingress isolation behavior of any pod.

Allow all egress traffic

If you want to allow all connections from all pods in a namespace, you can create a policy that explicitly allows all outgoing connections from pods in that namespace.

`service/networking/network-policy-allow-all-egress.yaml` Copy service/networking/network-policy-allow-all-egress.yaml to clipboard

```
---
apiVersion: networking.k8s.io/v1kind: NetworkPolicymetadata:  name: allow-all-egressspec:  podSelector: {}  egress:  - {}  policyT
```

With this policy in place, no additional policy or policies can cause any outgoing connection from those pods to be denied. This policy has no effect on isolation for ingress to any pod.

Default deny all ingress and all egress traffic

You can create a "default" policy for a namespace which prevents all ingress AND egress traffic by creating the following NetworkPolicy in that namespace.

`service/networking/network-policy-default-deny-all.yaml` Copy service/networking/network-policy-default-deny-all.yaml to clipboard

```
---
apiVersion: networking.k8s.io/v1kind: NetworkPolicymetadata:  name: default-deny-allspec:  podSelector: {}  policyTypes:  - Ingres
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed ingress or egress traffic.

Network traffic filtering

NetworkPolicy is defined for layer 4 connections (TCP, UDP, and optionally SCTP). For all the other protocols, the behaviour may vary across network plugins.

Note:

You must be using a CNI plugin that supports SCTP protocol NetworkPolicies.

When a `deny all` network policy is defined, it is only guaranteed to deny TCP, UDP and SCTP connections. For other protocols, such as ARP or ICMP, the behaviour is undefined. The same applies to allow rules: when a specific pod is allowed as ingress source or egress destination, it is undefined what happens with (for example) ICMP packets. Protocols such as ICMP may be allowed by some network plugins and denied by others.

Targeting a range of ports

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

When writing a NetworkPolicy, you can target a range of ports instead of a single port.

This is achievable with the usage of the `endPort` field, as the following example:

`service/networking/networkpolicy-multiport-egress.yaml` Copy service/networking/networkpolicy-multiport-egress.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicymetadata:  name: multi-port-egress  namespace: defaultspec:  podSelector:    matchLabels:      role: db  policy
```

The above rule allows any Pod with label `role=db` on the namespace `default` to communicate with any IP within the range `10.0.0.0/24` over TCP, provided that the target port is between the range 32000 and 32768.

The following restrictions apply when using this field:

- The `endPort` field must be equal to or greater than the `port` field.
- `endPort` can only be defined if `port` is also defined.
- Both ports must be numeric.

Note:

Your cluster must be using a CNI plugin that supports the `endPort` field in NetworkPolicy specifications. If your network plugin does not support the `endPort` field and you specify a NetworkPolicy with that, the policy will be applied only for the single `port` field.

Targeting multiple namespaces by label

In this scenario, your `Egress` NetworkPolicy targets more than one namespace using their label names. For this to work, you need to label the target namespaces. For example:

```
kubectl label namespace frontend namespace=frontend
kubectl label namespace backend namespace=backend
```

Add the labels under `namespaceSelector` in your NetworkPolicy document. For example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicymetadata:  name: egress-namespacesspec:  podSelector:    matchLabels:      app: myapp  policyTypes:  - Egress
```

Note:

It is not possible to directly specify the name of the namespaces in a NetworkPolicy. You must use a `namespaceSelector` with `matchLabels` or `matchExpressions` to select the namespaces based on their labels.

Targeting a Namespace by its name

The Kubernetes control plane sets an immutable label `kubernetes.io/metadata.name` on all namespaces, the value of the label is the namespace name.

While NetworkPolicy cannot target a namespace by its name with some object field, you can use the standardized label to target a specific namespace.

Pod lifecycle

Note:

The following applies to clusters with a conformant networking plugin and a conformant implementation of NetworkPolicy.

When a new NetworkPolicy object is created, it may take some time for a network plugin to handle the new object. If a pod that is affected by a NetworkPolicy is created before the network plugin has completed NetworkPolicy handling, that pod may be started unprotected, and isolation rules will be applied when the NetworkPolicy handling is completed.

Once the NetworkPolicy is handled by a network plugin,

1. All newly created pods affected by a given NetworkPolicy will be isolated before they are started. Implementations of NetworkPolicy must ensure that filtering is effective throughout the Pod lifecycle, even from the very first instant that any container in that Pod is started. Because they are applied at Pod level, NetworkPolicies apply equally to init containers, sidecar containers, and regular containers.

2. Allow rules will be applied eventually after the isolation rules (or may be applied at the same time). In the worst case, a newly created pod may have no network connectivity at all when it is first started, if isolation rules were already applied, but no allow rules were applied yet.

Every created NetworkPolicy will be handled by a network plugin eventually, but there is no way to tell from the Kubernetes API when exactly that happens.

Therefore, pods must be resilient against being started up with different network connectivity than expected. If you need to make sure the pod can reach certain destinations before being started, you can use an init container to wait for those destinations to be reachable before kubelet starts the app containers.

Every NetworkPolicy will be applied to all selected pods eventually. Because the network plugin may implement NetworkPolicy in a distributed manner, it is possible that pods may see a slightly inconsistent view of network policies when the pod is first created, or when pods or policies change. For example, a newly-created pod that is supposed to be able to reach both Pod A on Node 1 and Pod B on Node 2 may find that it can reach Pod A immediately, but cannot reach Pod B until a few seconds later.

NetworkPolicy and `hostNetwork` pods

NetworkPolicy behaviour for `hostNetwork` pods is undefined, but it should be limited to 2 possibilities:

- The network plugin can distinguish `hostNetwork` pod traffic from all other traffic (including being able to distinguish traffic from different `hostNetwork` pods on the same node), and will apply NetworkPolicy to `hostNetwork` pods just like it does to pod-network pods.
- The network plugin cannot properly distinguish `hostNetwork` pod traffic, and so it ignores `hostNetwork` pods when matching `podSelector` and `namespaceSelector`. Traffic to/from `hostNetwork` pods is treated the same as all other traffic to/from the node IP. (This is the most common implementation.)

This applies when

1. a `hostNetwork` pod is selected by `spec.podSelector`.

```
...
spec:
  podSelector:
    matchLabels:
      role: client
...
```

2. a `hostNetwork` pod is selected by a `podSelector` or `namespaceSelector` in an `ingress` or `egress` rule.

```
...
ingress:
  - from:
    - podSelector:
        matchLabels:
          role: client
...
```

At the same time, since `hostNetwork` pods have the same IP addresses as the nodes they reside on, their connections will be treated as node connections. For example, you can allow traffic from a `hostNetwork` Pod using an `ipBlock` rule.

What you can't do with network policies (at least, not yet)

As of Kubernetes 1.34, the following functionality does not exist in the NetworkPolicy API, but you might be able to implement workarounds using Operating System components (such as SELinux, OpenVSwitch, IPTables, and so on) or Layer 7 technologies (Ingress controllers, Service Mesh implementations) or admission controllers. In case you are new to network security in Kubernetes, its worth noting that the following User Stories cannot (yet) be implemented using the NetworkPolicy API.

- Forcing internal cluster traffic to go through a common gateway (this might be best served with a service mesh or other proxy).
- Anything TLS related (use a service mesh or ingress controller for this).
- Node specific policies (you can use CIDR notation for these, but you cannot target nodes by their Kubernetes identities specifically).
- Targeting of services by name (you can, however, target pods or namespaces by their labels, which is often a viable workaround).
- Creation or management of "Policy requests" that are fulfilled by a third party.
- Default policies which are applied to all namespaces or pods (there are some third party Kubernetes distributions and projects which can do this).
- Advanced policy querying and reachability tooling.
- The ability to log network security events (for example connections that are blocked or accepted).
- The ability to explicitly deny policies (currently the model for NetworkPolicies are deny by default, with only the ability to add allow rules).
- The ability to prevent loopback or incoming host traffic (Pods cannot currently block localhost access, nor do they have the ability to block access from their resident node).

NetworkPolicy's impact on existing connections

When the set of NetworkPolicies that applies to an existing connection changes - this could happen either due to a change in NetworkPolicies or if the relevant labels of the namespaces/pods selected by the policy (both subject and peers) are changed in the middle of an existing connection - it is implementation defined as to whether the change will take effect for that existing connection or not. Example: A policy is created that leads to denying a previously allowed connection, the underlying network plugin implementation is responsible for defining if that new policy will close the existing connections or not. It is recommended not to modify policies/pods/namespaces in ways that might affect existing connections.

What's next

- See the Declare Network Policy walkthrough for further examples.

- See more recipes for common scenarios enabled by the NetworkPolicy resource.

Taints and Tolerations

Node affinity is a property of Pods that *attracts* them to a set of nodes (either as a preference or a hard requirement). *Taints* are the opposite -- they allow a node to repel a set of pods.

Tolerations are applied to pods. Tolerations allow the scheduler to schedule pods with matching taints. Tolerations allow scheduling but don't guarantee scheduling: the scheduler also evaluates other parameters as part of its function.

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

Concepts

You add a taint to a node using kubectl taint. For example,

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

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

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

```
kubectl taint nodes node1 key1=value1:NoSchedule-
```

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

```
tolerations:
- key: "key1"  operator: "Equal"  value: "value1"  effect: "NoSchedule"
```

```
tolerations:
- key: "key1"  operator: "Exists"  effect: "NoSchedule"
```

The default Kubernetes scheduler takes taints and tolerations into account when selecting a node to run a particular Pod. However, if you manually specify the `.spec.nodeName` for a Pod, that action bypasses the scheduler; the Pod is then bound onto the node where you assigned it, even if there are `NoSchedule` taints on that node that you selected. If this happens and the node also has a `NoExecute` taint set, the kubelet will eject the Pod unless there is an appropriate tolerance set.

Here's an example of a pod that has some tolerations defined:

`pods/pod-with-toleration.yaml` Copy pods/pod-with-toleration.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: nginx  labels:    env: testspec:  containers:  - name: nginx    image: nginx    imagePullPolicy: IfNotPr
```

The default value for `operator` is `Equal`.

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

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

Note:

There are two special cases:

If the `key` is empty, then the `operator` must be `Exists`, which matches all keys and values. Note that the `effect` still needs to be matched at the same time.

An empty `effect` matches all effects with key `key1`.

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

The allowed values for the `effect` field are:

`NoExecute`

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

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

`NoSchedule`

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

`PreferNoSchedule`

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

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

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

For example, imagine you taint a node like this

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

And a pod has two tolerations:

```
tolerations:
- key: "key1"  operator: "Equal"  value: "value1"  effect: "NoSchedule"- key: "key1"  operator: "Equal"  value: "value1"  effect:
```

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

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

```
tolerations:
- key: "key1"  operator: "Equal"  value: "value1"  effect: "NoExecute"  tolerationSeconds: 3600
```

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

Example Use Cases

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

- **Dedicated Nodes**: If you want to dedicate a set of nodes for exclusive use by a particular set of users, you can add a taint to those nodes (say, `kubectl taint nodes nodename dedicated=groupName:NoSchedule`) and then add a corresponding toleration to their pods (this would be done most easily by writing a custom admission controller). The pods with the tolerations will then be allowed to use the tainted (dedicated) nodes as well as any other nodes in the cluster. If you want to dedicate the nodes to them *and* ensure they *only* use the dedicated nodes, then you should additionally add a label similar to the taint to the same set of nodes (e.g. `dedicated=groupName`), and the admission controller should additionally add a node affinity to require that the pods can only schedule onto nodes labeled with `dedicated=groupName`.

- **Nodes with Special Hardware**: In a cluster where a small subset of nodes have specialized hardware (for example GPUs), it is desirable to keep pods that don't need the specialized hardware off of those nodes, thus leaving room for later-arriving pods that do need the specialized hardware. This can be done by tainting the nodes that have the specialized hardware (e.g. `kubectl taint nodes nodename special=true:NoSchedule` or `kubectl taint nodes nodename special=true:PreferNoSchedule`) and adding a corresponding toleration to pods that use the special hardware. As in the dedicated nodes use case, it is probably easiest to apply the tolerations using a custom admission controller. For example, it is recommended to use Extended Resources to represent the special hardware, taint your special hardware nodes with the extended resource name and run the ExtendedResourceToleration admission controller. Now, because the nodes are tainted, no pods without the toleration will schedule on them. But when you submit a pod that requests the extended resource, the `ExtendedResourceToleration` admission controller will automatically add the correct toleration to the pod and that pod will schedule on the special hardware nodes. This will make sure that these special hardware nodes are dedicated for pods requesting such hardware and you don't have to manually add tolerations to your pods.

- **Taint based Evictions**: A per-pod-configurable eviction behavior when there are node problems, which is described in the next section.

Taint based Evictions

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

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

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

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

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

Note:

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

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

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

```
tolerations:
- key: "node.kubernetes.io/unreachable"  operator: "Exists"  effect: "NoExecute"  tolerationSeconds: 6000
```

Note:

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

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

DaemonSet pods are created with `NoExecute` tolerations for the following taints with no `tolerationSeconds`:

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

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

Note:

The node controller was responsible for adding taints to nodes and evicting pods. But after 1.29, the taint-based eviction implementation has been moved out of node controller into a separate, and independent component called taint-eviction-controller. Users can optionally disable taint-based eviction by setting `--controllers=-taint-eviction-controller` in kube-controller-manager.

Taint Nodes by Condition

The control plane, using the node controller, automatically creates taints with a `NoSchedule` effect for node conditions.

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

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

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

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

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

Device taints and tolerations

Instead of tainting entire nodes, administrators can also taint individual devices when the cluster uses dynamic resource allocation to manage special hardware. The advantage is that tainting can be targeted towards exactly the hardware that is faulty or needs maintenance. Tolerations are also supported and can be specified when requesting devices. Like taints they apply to all pods which share the same allocated device.

What's next

- Read about Node-pressure Eviction and how you can configure it
- Read about Pod Priority
- Read about device taints and tolerations

Pod Security Standards

A detailed look at the different policy levels defined in the Pod Security Standards.

The Pod Security Standards define three different *policies* to broadly cover the security spectrum. These policies are *cumulative* and range from highly-permissive to highly-restrictive. This guide outlines the requirements of each policy.

Profile	Description
Privileged	Unrestricted policy, providing the widest possible level of permissions. This policy allows for known privilege escalations.
Baseline	Minimally restrictive policy which prevents known privilege escalations. Allows the default (minimally specified) Pod configuration.
Restricted	Heavily restricted policy, following current Pod hardening best practices.

Profile Details

Privileged

The *Privileged* policy is purposely-open, and entirely unrestricted. This type of policy is typically aimed at system- and infrastructure-level workloads managed by privileged, trusted users.

The Privileged policy is defined by an absence of restrictions. If you define a Pod where the Privileged security policy applies, the Pod you define is able to bypass typical container isolation mechanisms. For example, you can define a Pod that has access to the node's host network.

Baseline

The *Baseline* policy is aimed at ease of adoption for common containerized workloads while preventing known privilege escalations. This policy is targeted at application operators and developers of non-critical applications. The following listed controls should be enforced/disallowed:

Note:

In this table, wildcards (`*`) indicate all elements in a list. For example, `spec.containers[*].securityContext` refers to the Security Context object for *all defined containers*. If any of the listed containers fails to meet the requirements, the entire pod will fail validation.

Control	Policy
HostProcess	Windows Pods offer the ability to run HostProcess containers which enables privileged access to the Windows host machine. Privileged access to the host is disallowed in the Baseline policy. FEATURE STATE: `Kubernetes v1.26 [stable]` **Restricted Fields** • `spec.securityContext.windowsOptions.hostProcess` • `spec.containers[*].securityContext.windowsOptions.hostProcess` • `spec.initContainers[*].securityContext.windowsOptions.hostProcess` • `spec.ephemeralContainers[*].securityContext.windowsOptions.hostProcess` **Allowed Values** • Undefined/nil • `false`
Host Namespaces	Sharing the host namespaces must be disallowed. **Restricted Fields** • `spec.hostNetwork` • `spec.hostPID` • `spec.hostIPC` **Allowed Values** • Undefined/nil • `false`
Privileged Containers	Privileged Pods disable most security mechanisms and must be disallowed. **Restricted Fields** • `spec.containers[*].securityContext.privileged` • `spec.initContainers[*].securityContext.privileged` • `spec.ephemeralContainers[*].securityContext.privileged` **Allowed Values** • Undefined/nil • `false`
Capabilities	Adding additional capabilities beyond those listed below must be disallowed. **Restricted Fields** • `spec.containers[*].securityContext.capabilities.add` • `spec.initContainers[*].securityContext.capabilities.add` • `spec.ephemeralContainers[*].securityContext.capabilities.add` **Allowed Values** • Undefined/nil • `AUDIT_WRITE` • `CHOWN` • `DAC_OVERRIDE` • `FOWNER` • `FSETID` • `KILL` • `MKNOD` • `NET_BIND_SERVICE` • `SETFCAP`

	- `SETGID` - `SETPCAP` - `SETUID` - `SYS_CHROOT`
HostPath Volumes	HostPath volumes must be forbidden. **Restricted Fields** - `spec.volumes[*].hostPath` **Allowed Values** - Undefined/nil
Host Ports	HostPorts should be disallowed entirely (recommended) or restricted to a known list **Restricted Fields** - `spec.containers[*].ports[*].hostPort` - `spec.initContainers[*].ports[*].hostPort` - `spec.ephemeralContainers[*].ports[*].hostPort` **Allowed Values** - Undefined/nil - Known list (not supported by the built-in Pod Security Admission controller) - `0`
Host Probes / Lifecycle Hooks (v1.34+)	The Host field in probes and lifecycle hooks must be disallowed. **Restricted Fields** - `spec.containers[*].livenessProbe.httpGet.host` - `spec.containers[*].readinessProbe.httpGet.host` - `spec.containers[*].startupProbe.httpGet.host` - `spec.containers[*].livenessProbe.tcpSocket.host` - `spec.containers[*].readinessProbe.tcpSocket.host` - `spec.containers[*].startupProbe.tcpSocket.host` - `spec.containers[*].lifecycle.postStart.tcpSocket.host` - `spec.containers[*].lifecycle.preStop.tcpSocket.host` - `spec.containers[*].lifecycle.postStart.httpGet.host` - `spec.containers[*].lifecycle.preStop.httpGet.host` - `spec.initContainers[*].livenessProbe.httpGet.host` - `spec.initContainers[*].readinessProbe.httpGet.host` - `spec.initContainers[*].startupProbe.httpGet.host` - `spec.initContainers[*].livenessProbe.tcpSocket.host` - `spec.initContainers[*].readinessProbe.tcpSocket.host` - `spec.initContainers[*].startupProbe.tcpSocket.host` - `spec.initContainers[*].lifecycle.postStart.tcpSocket.host` - `spec.initContainers[*].lifecycle.preStop.tcpSocket.host` - `spec.initContainers[*].lifecycle.postStart.httpGet.host` - `spec.initContainers[*].lifecycle.preStop.httpGet.host` **Allowed Values** - Undefined/nil - `""`
AppArmor	On supported hosts, the `RuntimeDefault` AppArmor profile is applied by default. The baseline policy should prevent overriding or disabling the default AppArmor profile, or restrict overrides to an allowed set of profiles. **Restricted Fields** - `spec.securityContext.appArmorProfile.type` - `spec.containers[*].securityContext.appArmorProfile.type` - `spec.initContainers[*].securityContext.appArmorProfile.type` - `spec.ephemeralContainers[*].securityContext.appArmorProfile.type` **Allowed Values** - Undefined/nil - `RuntimeDefault` - `Localhost` --- - `metadata.annotations["container.apparmor.security.beta.kubernetes.io/*"]` **Allowed Values** - Undefined/nil - `runtime/default`

	• `localhost/*`
SELinux	Setting the SELinux type is restricted, and setting a custom SELinux user or role option is forbidden. **Restricted Fields** • `spec.securityContext.seLinuxOptions.type` • `spec.containers[*].securityContext.seLinuxOptions.type` • `spec.initContainers[*].securityContext.seLinuxOptions.type` • `spec.ephemeralContainers[*].securityContext.seLinuxOptions.type` **Allowed Values** • Undefined/"" • `container_t` • `container_init_t` • `container_kvm_t` • `container_engine_t` (since Kubernetes 1.31) --- **Restricted Fields** • `spec.securityContext.seLinuxOptions.user` • `spec.containers[*].securityContext.seLinuxOptions.user` • `spec.initContainers[*].securityContext.seLinuxOptions.user` • `spec.ephemeralContainers[*].securityContext.seLinuxOptions.user` • `spec.securityContext.seLinuxOptions.role` • `spec.containers[*].securityContext.seLinuxOptions.role` • `spec.initContainers[*].securityContext.seLinuxOptions.role` • `spec.ephemeralContainers[*].securityContext.seLinuxOptions.role` **Allowed Values** • Undefined/""
`/proc` Mount Type	The default `/proc` masks are set up to reduce attack surface, and should be required. **Restricted Fields** • `spec.containers[*].securityContext.procMount` • `spec.initContainers[*].securityContext.procMount` • `spec.ephemeralContainers[*].securityContext.procMount` **Allowed Values** • Undefined/nil • `Default`
Seccomp	Seccomp profile must not be explicitly set to `Unconfined`. **Restricted Fields** • `spec.securityContext.seccompProfile.type` • `spec.containers[*].securityContext.seccompProfile.type` • `spec.initContainers[*].securityContext.seccompProfile.type` • `spec.ephemeralContainers[*].securityContext.seccompProfile.type` **Allowed Values** • Undefined/nil • `RuntimeDefault` • `Localhost`
Sysctls	Sysctls can disable security mechanisms or affect all containers on a host, and should be disallowed except for an allowed "safe" subset. A sysctl is considered safe if it is namespaced in the container or the Pod, and it is isolated from other Pods or processes on the same Node. **Restricted Fields** • `spec.securityContext.sysctls[*].name` **Allowed Values** • Undefined/nil • `kernel.shm_rmid_forced` • `net.ipv4.ip_local_port_range` • `net.ipv4.ip_unprivileged_port_start` • `net.ipv4.tcp_syncookies` • `net.ipv4.ping_group_range` • `net.ipv4.ip_local_reserved_ports` (since Kubernetes 1.27) • `net.ipv4.tcp_keepalive_time` (since Kubernetes 1.29) • `net.ipv4.tcp_fin_timeout` (since Kubernetes 1.29) • `net.ipv4.tcp_keepalive_intvl` (since Kubernetes 1.29)

- `net.ipv4.tcp_keepalive_probes` (since Kubernetes 1.29)

Restricted

The *Restricted* policy is aimed at enforcing current Pod hardening best practices, at the expense of some compatibility. It is targeted at operators and developers of security-critical applications, as well as lower-trust users. The following listed controls should be enforced/disallowed:

Note:

In this table, wildcards (`*`) indicate all elements in a list. For example, `spec.containers[*].securityContext` refers to the Security Context object for *all defined containers*. If any of the listed containers fails to meet the requirements, the entire pod will fail validation.

Control	Policy
Everything from the Baseline policy	
Volume Types	The Restricted policy only permits the following volume types. **Restricted Fields** • `spec.volumes[*]` **Allowed Values** Every item in the `spec.volumes[*]` list must set one of the following fields to a non-null value: • `spec.volumes[*].configMap` • `spec.volumes[*].csi` • `spec.volumes[*].downwardAPI` • `spec.volumes[*].emptyDir` • `spec.volumes[*].ephemeral` • `spec.volumes[*].persistentVolumeClaim` • `spec.volumes[*].projected` • `spec.volumes[*].secret`
Privilege Escalation (v1.8+)	Privilege escalation (such as via set-user-ID or set-group-ID file mode) should not be allowed. *This is Linux only policy in v1.25+* `(spec.os.name != windows)` **Restricted Fields** • `spec.containers[*].securityContext.allowPrivilegeEscalation` • `spec.initContainers[*].securityContext.allowPrivilegeEscalation` • `spec.ephemeralContainers[*].securityContext.allowPrivilegeEscalation` **Allowed Values** • `false`
Running as Non-root	Containers must be required to run as non-root users. **Restricted Fields** • `spec.securityContext.runAsNonRoot` • `spec.containers[*].securityContext.runAsNonRoot` • `spec.initContainers[*].securityContext.runAsNonRoot` • `spec.ephemeralContainers[*].securityContext.runAsNonRoot` **Allowed Values** • `true` The container fields may be undefined/`nil` if the pod-level `spec.securityContext.runAsNonRoot` is set to `true`.
Running as Non-root user (v1.23+)	Containers must not set `runAsUser` to 0 **Restricted Fields** • `spec.securityContext.runAsUser` • `spec.containers[*].securityContext.runAsUser` • `spec.initContainers[*].securityContext.runAsUser` • `spec.ephemeralContainers[*].securityContext.runAsUser` **Allowed Values** • any non-zero value • `undefined/null`
Seccomp (v1.19+)	Seccomp profile must be explicitly set to one of the allowed values. Both the `Unconfined` profile and the *absence* of a profile are prohibited. *This is Linux only policy in v1.25+* `(spec.os.name != windows)` **Restricted Fields** • `spec.securityContext.seccompProfile.type` • `spec.containers[*].securityContext.seccompProfile.type`

	- `spec.initContainers[*].securityContext.seccompProfile.type` - `spec.ephemeralContainers[*].securityContext.seccompProfile.type` **Allowed Values** - `RuntimeDefault` - `Localhost` The container fields may be undefined/`nil` if the pod-level `spec.securityContext.seccompProfile.type` field is set appropriately. Conversely, the pod-level field may be undefined/`nil` if _all_ container- level fields are set.
Capabilities (v1.22+)	Containers must drop `ALL` capabilities, and are only permitted to add back the `NET_BIND_SERVICE` capability. *This is Linux only policy in v1.25+ `(.spec.os.name != "windows")`* **Restricted Fields** - `spec.containers[*].securityContext.capabilities.drop` - `spec.initContainers[*].securityContext.capabilities.drop` - `spec.ephemeralContainers[*].securityContext.capabilities.drop` **Allowed Values** - Any list of capabilities that includes `ALL` --- **Restricted Fields** - `spec.containers[*].securityContext.capabilities.add` - `spec.initContainers[*].securityContext.capabilities.add` - `spec.ephemeralContainers[*].securityContext.capabilities.add` **Allowed Values** - Undefined/nil - `NET_BIND_SERVICE`

Policy Instantiation

Decoupling policy definition from policy instantiation allows for a common understanding and consistent language of policies across clusters, independent of the underlying enforcement mechanism.

As mechanisms mature, they will be defined below on a per-policy basis. The methods of enforcement of individual policies are not defined here.

Pod Security Admission Controller

- Privileged namespace
- Baseline namespace
- Restricted namespace

Alternatives

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.

Other alternatives for enforcing policies are being developed in the Kubernetes ecosystem, such as:

- Kubewarden
- Kyverno
- OPA Gatekeeper

Pod OS field

Kubernetes lets you use nodes that run either Linux or Windows. You can mix both kinds of node in one cluster. Windows in Kubernetes has some limitations and differentiators from Linux-based workloads. Specifically, many of the Pod `securityContext` fields have no effect on Windows.

Note:

Kubelets prior to v1.24 don't enforce the pod OS field, and if a cluster has nodes on versions earlier than v1.24 the Restricted policies should be pinned to a version prior to v1.25.

Restricted Pod Security Standard changes

Another important change, made in Kubernetes v1.25 is that the *Restricted* policy has been updated to use the `pod.spec.os.name` field. Based on the OS name, certain policies that are specific to a particular OS can be relaxed for the other OS.

OS-specific policy controls

Restrictions on the following controls are only required if `.spec.os.name` is not `windows`:

- Privilege Escalation

- Seccomp
- Linux Capabilities

User namespaces

User Namespaces are a Linux-only feature to run workloads with increased isolation. How they work together with Pod Security Standards is described in the documentation for Pods that use user namespaces.

FAQ

Why isn't there a profile between Privileged and Baseline?

The three profiles defined here have a clear linear progression from most secure (Restricted) to least secure (Privileged), and cover a broad set of workloads. Privileges required above the Baseline policy are typically very application specific, so we do not offer a standard profile in this niche. This is not to say that the privileged profile should always be used in this case, but that policies in this space need to be defined on a case-by-case basis.

SIG Auth may reconsider this position in the future, should a clear need for other profiles arise.

What's the difference between a security profile and a security context?

Security Contexts configure Pods and Containers at runtime. Security contexts are defined as part of the Pod and container specifications in the Pod manifest, and represent parameters to the container runtime.

Security profiles are control plane mechanisms to enforce specific settings in the Security Context, as well as other related parameters outside the Security Context. As of July 2021, Pod Security Policies are deprecated in favor of the built-in Pod Security Admission Controller.

What about sandboxed Pods?

There is currently no API standard that controls whether a Pod is considered sandboxed or not. Sandbox Pods may be identified by the use of a sandboxed runtime (such as gVisor or Kata Containers), but there is no standard definition of what a sandboxed runtime is.

The protections necessary for sandboxed workloads can differ from others. For example, the need to restrict privileged permissions is lessened when the workload is isolated from the underlying kernel. This allows for workloads requiring heightened permissions to still be isolated.

Additionally, the protection of sandboxed workloads is highly dependent on the method of sandboxing. As such, no single recommended profile is recommended for all sandboxed workloads.

API-initiated Eviction

API-initiated eviction is the process by which you use the Eviction API to create an `Eviction` object that triggers graceful pod termination.

You can request eviction by calling the Eviction API directly, or programmatically using a client of the API server, like the `kubectl drain` command. This creates an `Eviction` object, which causes the API server to terminate the Pod.

API-initiated evictions respect your configured `PodDisruptionBudgets` and `terminationGracePeriodSeconds`.

Using the API to create an Eviction object for a Pod is like performing a policy-controlled `DELETE` operation on the Pod.

Calling the Eviction API

You can use a Kubernetes language client to access the Kubernetes API and create an `Eviction` object. To do this, you POST the attempted operation, similar to the following example:

- policy/v1
- policy/v1beta1

Note:

`policy/v1` Eviction is available in v1.22+. Use `policy/v1beta1` with prior releases.

```
{
  "apiVersion": "policy/v1",
  "kind": "Eviction",
  "metadata": {
    "name": "quux",
    "namespace": "default"
  }
}
```

Note:

Deprecated in v1.22 in favor of `policy/v1`

```
{
  "apiVersion": "policy/v1beta1",
  "kind": "Eviction",
  "metadata": {
    "name": "quux",
    "namespace": "default"
  }
}
```

Alternatively, you can attempt an eviction operation by accessing the API using `curl` or `wget`, similar to the following example:

```
curl -v -H 'Content-type: application/json' https://your-cluster-api-endpoint.example/api/v1/namespaces/default/pods/quux/eviction
```

How API-initiated eviction works

When you request an eviction using the API, the API server performs admission checks and responds in one of the following ways:

- `200 OK`: the eviction is allowed, the `Eviction` subresource is created, and the Pod is deleted, similar to sending a `DELETE` request to the Pod URL.
- `429 Too Many Requests`: the eviction is not currently allowed because of the configured PodDisruptionBudget. You may be able to attempt the eviction again later. You might also see this response because of API rate limiting.
- `500 Internal Server Error`: the eviction is not allowed because there is a misconfiguration, like if multiple PodDisruptionBudgets reference the same Pod.

If the Pod you want to evict isn't part of a workload that has a PodDisruptionBudget, the API server always returns `200 OK` and allows the eviction.

If the API server allows the eviction, the Pod is deleted as follows:

1. The `Pod` resource in the API server is updated with a deletion timestamp, after which the API server considers the `Pod` resource to be terminated. The `Pod` resource is also marked with the configured grace period.
2. The kubelet on the node where the local Pod is running notices that the `Pod` resource is marked for termination and starts to gracefully shut down the local Pod.
3. While the kubelet is shutting the Pod down, the control plane removes the Pod from EndpointSlice objects. As a result, controllers no longer consider the Pod as a valid object.
4. After the grace period for the Pod expires, the kubelet forcefully terminates the local Pod.
5. The kubelet tells the API server to remove the `Pod` resource.
6. The API server deletes the `Pod` resource.

Troubleshooting stuck evictions

In some cases, your applications may enter a broken state, where the Eviction API will only return `429` or `500` responses until you intervene. This can happen if, for example, a ReplicaSet creates pods for your application but new pods do not enter a `Ready` state. You may also notice this behavior in cases where the last evicted Pod had a long termination grace period.

If you notice stuck evictions, try one of the following solutions:

- Abort or pause the automated operation causing the issue. Investigate the stuck application before you restart the operation.
- Wait a while, then directly delete the Pod from your cluster control plane instead of using the Eviction API.

What's next

- Learn how to protect your applications with a Pod Disruption Budget.
- Learn about Node-pressure Eviction.
- Learn about Pod Priority and Preemption.

Persistent Volumes

This document describes *persistent volumes* in Kubernetes. Familiarity with volumes, StorageClasses and VolumeAttributesClasses is suggested.

Introduction

Managing storage is a distinct problem from managing compute instances. The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, we introduce two new API resources: PersistentVolume and PersistentVolumeClaim.

A *PersistentVolume* (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A *PersistentVolumeClaim* (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany, ReadWriteMany, or ReadWriteOncePod, see AccessModes).

While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems. Cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than size and access modes, without exposing users to the details of how those volumes are implemented. For these needs, there is the *StorageClass* resource.

See the detailed walkthrough with working examples.

Lifecycle of a volume and claim

PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs follows this lifecycle:

Provisioning

There are two ways PVs may be provisioned: statically or dynamically.

Static

A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

Dynamic

When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on StorageClasses: the PVC must request a storage class and the administrator must have created and configured that class for dynamic provisioning to occur. Claims that request the class `""` effectively disable dynamic provisioning for themselves.

To enable dynamic storage provisioning based on storage class, the cluster administrator needs to enable the `DefaultStorageClass` admission controller on the API server. This can be done, for example, by ensuring that `DefaultStorageClass` is among the comma-delimited, ordered list of values for the `--enable-admission-plugins` flag of the API server component. For more information on API server command-line flags, check kube-apiserver documentation.

Binding

A user creates, or in the case of dynamic provisioning, has already created, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes. A control loop in the control plane watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC. Otherwise, the user will always get at least what they asked for, but the volume may be in excess of what was requested. Once bound, PersistentVolumeClaim binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping, using a ClaimRef which is a bi-directional binding between the PersistentVolume and the PersistentVolumeClaim.

Claims will remain unbound indefinitely if a matching volume does not exist. Claims will be bound as matching volumes become available. For example, a cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod. For volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a Pod.

Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Users schedule Pods and access their claimed PVs by including a `persistentVolumeClaim` section in a Pod's `volumes` block. See Claims As Volumes for more details on this.

Storage Object in Use Protection

The purpose of the Storage Object in Use Protection feature is to ensure that PersistentVolumeClaims (PVCs) in active use by a Pod and PersistentVolume (PVs) that are bound to PVCs are not removed from the system, as this may result in data loss.

Note:

PVC is in active use by a Pod when a Pod object exists that is using the PVC.

If a user deletes a PVC in active use by a Pod, the PVC is not removed immediately. PVC removal is postponed until the PVC is no longer actively used by any Pods. Also, if an admin deletes a PV that is bound to a PVC, the PV is not removed immediately. PV removal is postponed until the PV is no longer bound to a PVC.

You can see that a PVC is protected when the PVC's status is `Terminating` and the `Finalizers` list includes `kubernetes.io/pvc-protection`:

```
kubectl describe pvc hostpath
Name:          hostpath
Namespace:     default
StorageClass:  example-hostpath
Status:        Terminating
Volume:
Labels:        <none>
Annotations:   volume.beta.kubernetes.io/storage-class=example-hostpath
               volume.beta.kubernetes.io/storage-provisioner=example.com/hostpath
Finalizers:    [kubernetes.io/pvc-protection]
...
```

You can see that a PV is protected when the PV's status is `Terminating` and the `Finalizers` list includes `kubernetes.io/pv-protection` too:

```
kubectl describe pv task-pv-volume
Name:            task-pv-volume
Labels:          type=local
Annotations:     <none>
Finalizers:      [kubernetes.io/pv-protection]
StorageClass:    standard
Status:          Terminating
Claim:
Reclaim Policy:  Delete
Access Modes:    RWO
Capacity:        1Gi
Message:
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /tmp/data
    HostPathType:
```

```
Events:            <none>
```

Reclaiming

When a user is done with their volume, they can delete the PVC objects from the API that allows reclamation of the resource. The reclaim policy for a PersistentVolume tells the cluster what to do with the volume after it has been released of its claim. Currently, volumes can either be Retained, Recycled, or Deleted.

Retain

The `Retain` reclaim policy allows for manual reclamation of the resource. When the PersistentVolumeClaim is deleted, the PersistentVolume still exists and the volume is considered "released". But it is not yet available for another claim because the previous claimant's data remains on the volume. An administrator can manually reclaim the volume with the following steps.

1. Delete the PersistentVolume. The associated storage asset in external infrastructure still exists after the PV is deleted.
2. Manually clean up the data on the associated storage asset accordingly.
3. Manually delete the associated storage asset.

If you want to reuse the same storage asset, create a new PersistentVolume with the same storage asset definition.

Delete

For volume plugins that support the `Delete` reclaim policy, deletion removes both the PersistentVolume object from Kubernetes, as well as the associated storage asset in the external infrastructure. Volumes that were dynamically provisioned inherit the reclaim policy of their StorageClass, which defaults to `Delete`. The administrator should configure the StorageClass according to users' expectations; otherwise, the PV must be edited or patched after it is created. See Change the Reclaim Policy of a PersistentVolume.

Recycle

Warning:

The `Recycle` reclaim policy is deprecated. Instead, the recommended approach is to use dynamic provisioning.

If supported by the underlying volume plugin, the `Recycle` reclaim policy performs a basic scrub (`rm -rf /thevolume/*`) on the volume and makes it available again for a new claim.

However, an administrator can configure a custom recycler Pod template using the Kubernetes controller manager command line arguments as described in the reference. The custom recycler Pod template must contain a `volumes` specification, as shown in the example below:

```
apiVersion: v1
kind: Podmetadata:  name: pv-recycler  namespace: defaultspec:  restartPolicy: Never  volumes:  - name: vol    hostPath:      path
```

However, the particular path specified in the custom recycler Pod template in the `volumes` part is replaced with the particular path of the volume that is being recycled.

PersistentVolume deletion protection finalizer

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

Finalizers can be added on a PersistentVolume to ensure that PersistentVolumes having `Delete` reclaim policy are deleted only after the backing storage are deleted.

The finalizer `external-provisioner.volume.kubernetes.io/finalizer`(introduced in v1.31) is added to both dynamically provisioned and statically provisioned CSI volumes.

The finalizer `kubernetes.io/pv-controller`(introduced in v1.31) is added to dynamically provisioned in-tree plugin volumes and skipped for statically provisioned in-tree plugin volumes.

The following is an example of dynamically provisioned in-tree plugin volume:

```
kubectl describe pv pvc-74a498d6-3929-47e8-8c02-078c1ece4d78
Name:            pvc-74a498d6-3929-47e8-8c02-078c1ece4d78
Labels:          <none>
Annotations:     kubernetes.io/createdby: vsphere-volume-dynamic-provisioner
                 pv.kubernetes.io/bound-by-controller: yes
                 pv.kubernetes.io/provisioned-by: kubernetes.io/vsphere-volume
Finalizers:      [kubernetes.io/pv-protection kubernetes.io/pv-controller]
StorageClass:    vcp-sc
Status:          Bound
Claim:           default/vcp-pvc-1
Reclaim Policy:  Delete
Access Modes:    RWO
VolumeMode:      Filesystem
Capacity:        1Gi
Node Affinity:   <none>
Message:
Source:
    Type:               vSphereVolume (a Persistent Disk resource in vSphere)
    VolumePath:         [vsanDatastore] d49c4a62-166f-ce12-c464-020077ba5d46/kubernetes-dynamic-pvc-74a498d6-3929-47e8-8c02-078c1e
    FSType:             ext4
    StoragePolicyName:  vSAN Default Storage Policy
Events:                 <none>
```

The finalizer `external-provisioner.volume.kubernetes.io/finalizer` is added for CSI volumes. The following is an example:

```
Name:            pvc-2f0bab97-85a8-4552-8044-eb8be45cf48d
Labels:          <none>
Annotations:     pv.kubernetes.io/provisioned-by: csi.vsphere.vmware.com
Finalizers:      [kubernetes.io/pv-protection external-provisioner.volume.kubernetes.io/finalizer]
StorageClass:    fast
Status:          Bound
Claim:           demo-app/nginx-logs
Reclaim Policy:  Delete
Access Modes:    RWO
VolumeMode:      Filesystem
Capacity:        200Mi
Node Affinity:   <none>
Message:
Source:
    Type:              CSI (a Container Storage Interface (CSI) volume source)
    Driver:            csi.vsphere.vmware.com
    FSType:            ext4
    VolumeHandle:      44830fa8-79b4-406b-8b58-621ba25353fd
    ReadOnly:          false
    VolumeAttributes:      storage.kubernetes.io/csiProvisionerIdentity=1648442357185-8081-csi.vsphere.vmware.com
                           type=vSphere CNS Block Volume
Events:                <none>
```

When the `CSIMigration{provider}` feature flag is enabled for a specific in-tree volume plugin, the `kubernetes.io/pv-controller` finalizer is replaced by the `external-provisioner.volume.kubernetes.io/finalizer` finalizer.

The finalizers ensure that the PV object is removed only after the volume is deleted from the storage backend provided the reclaim policy of the PV is `Delete`. This also ensures that the volume is deleted from storage backend irrespective of the order of deletion of PV and PVC.

Reserving a PersistentVolume

The control plane can bind PersistentVolumeClaims to matching PersistentVolumes in the cluster. However, if you want a PVC to bind to a specific PV, you need to pre-bind them.

By specifying a PersistentVolume in a PersistentVolumeClaim, you declare a binding between that specific PV and PVC. If the PersistentVolume exists and has not reserved PersistentVolumeClaims through its `claimRef` field, then the PersistentVolume and PersistentVolumeClaim will be bound.

The binding happens regardless of some volume matching criteria, including node affinity. The control plane still checks that storage class, access modes, and requested storage size are valid.

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: foo-pvc  namespace: foospec:  storageClassName: "" # Empty string must be explicitly s
```

This method does not guarantee any binding privileges to the PersistentVolume. If other PersistentVolumeClaims could use the PV that you specify, you first need to reserve that storage volume. Specify the relevant PersistentVolumeClaim in the `claimRef` field of the PV so that other PVCs can not bind to it.

```
apiVersion: v1
kind: PersistentVolumemetadata:  name: foo-pvspec:  storageClassName: ""  claimRef:    name: foo-pvc    namespace: foo  ...
```

This is useful if you want to consume PersistentVolumes that have their `persistentVolumeReclaimPolicy` set to `Retain`, including cases where you are reusing an existing PV.

Expanding Persistent Volumes Claims

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

Support for expanding PersistentVolumeClaims (PVCs) is enabled by default. You can expand the following types of volumes:

- csi (including some CSI migrated volme types)
- flexVolume (deprecated)
- portworxVolume (deprecated)

You can only expand a PVC if its storage class's `allowVolumeExpansion` field is set to true.

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: example-vol-defaultprovisioner: vendor-name.example/magicstorageparameters:  resturl: "http://1
```

To request a larger volume for a PVC, edit the PVC object and specify a larger size. This triggers expansion of the volume that backs the underlying PersistentVolume. A new PersistentVolume is never created to satisfy the claim. Instead, an existing volume is resized.

Warning:

Directly editing the size of a PersistentVolume can prevent an automatic resize of that volume. If you edit the capacity of a PersistentVolume, and then edit the `.spec` of a matching PersistentVolumeClaim to make the size of the PersistentVolumeClaim match the PersistentVolume, then no storage resize happens. The Kubernetes control plane will see that the desired state of both resources matches, conclude that the backing volume size has been manually increased and that no resize is necessary.

CSI Volume expansion

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

Support for expanding CSI volumes is enabled by default but it also requires a specific CSI driver to support volume expansion. Refer to documentation of the specific CSI driver for more information.

Resizing a volume containing a file system

You can only resize volumes containing a file system if the file system is XFS, Ext3, or Ext4.

When a volume contains a file system, the file system is only resized when a new Pod is using the PersistentVolumeClaim in `ReadWrite` mode. File system expansion is either done when a Pod is starting up or when a Pod is running and the underlying file system supports online expansion.

FlexVolumes (deprecated since Kubernetes v1.23) allow resize if the driver is configured with the `RequiresFSResize` capability to `true`. The FlexVolume can be resized on Pod restart.

Resizing an in-use PersistentVolumeClaim

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

In this case, you don't need to delete and recreate a Pod or deployment that is using an existing PVC. Any in-use PVC automatically becomes available to its Pod as soon as its file system has been expanded. This feature has no effect on PVCs that are not in use by a Pod or deployment. You must create a Pod that uses the PVC before the expansion can complete.

Similar to other volume types - FlexVolume volumes can also be expanded when in-use by a Pod.

Note:

FlexVolume resize is possible only when the underlying driver supports resize.

Recovering from Failure when Expanding Volumes

If a user specifies a new size that is too big to be satisfied by underlying storage system, expansion of PVC will be continuously retried until user or cluster administrator takes some action. This can be undesirable and hence Kubernetes provides following methods of recovering from such failures.

- Manually with Cluster Administrator access
- By requesting expansion to smaller size

If expanding underlying storage fails, the cluster administrator can manually recover the Persistent Volume Claim (PVC) state and cancel the resize requests. Otherwise, the resize requests are continuously retried by the controller without administrator intervention.

1. Mark the PersistentVolume(PV) that is bound to the PersistentVolumeClaim(PVC) with `Retain` reclaim policy.
2. Delete the PVC. Since PV has `Retain` reclaim policy - we will not lose any data when we recreate the PVC.
3. Delete the `claimRef` entry from PV specs, so as new PVC can bind to it. This should make the PV `Available`.
4. Re-create the PVC with smaller size than PV and set `volumeName` field of the PVC to the name of the PV. This should bind new PVC to existing PV.
5. Don't forget to restore the reclaim policy of the PV.

If expansion has failed for a PVC, you can retry expansion with a smaller size than the previously requested value. To request a new expansion attempt with a smaller proposed size, edit `.spec.resources` for that PVC and choose a value that is less than the value you previously tried. This is useful if expansion to a higher value did not succeed because of capacity constraint. If that has happened, or you suspect that it might have, you can retry expansion by specifying a size that is within the capacity limits of underlying storage provider. You can monitor status of resize operation by watching `.status.allocatedResourceStatuses` and events on the PVC.

Note that, although you can specify a lower amount of storage than what was requested previously, the new value must still be higher than `.status.capacity`. Kubernetes does not support shrinking a PVC to less than its current size.

Types of Persistent Volumes

PersistentVolume types are implemented as plugins. Kubernetes currently supports the following plugins:

- `csi` - Container Storage Interface (CSI)
- `fc` - Fibre Channel (FC) storage
- `hostPath` - HostPath volume (for single node testing only; WILL NOT WORK in a multi-node cluster; consider using `local` volume instead)
- `iscsi` - iSCSI (SCSI over IP) storage
- `local` - local storage devices mounted on nodes.
- `nfs` - Network File System (NFS) storage

The following types of PersistentVolume are deprecated but still available. If you are using these volume types except for `flexVolume`, `cephfs` and `rbd`, please install corresponding CSI drivers.

- `awsElasticBlockStore` - AWS Elastic Block Store (EBS) (**migration on by default** starting v1.23)
- `azureDisk` - Azure Disk (**migration on by default** starting v1.23)
- `azureFile` - Azure File (**migration on by default** starting v1.24)
- `cinder` - Cinder (OpenStack block storage) (**migration on by default** starting v1.21)
- `flexVolume` - FlexVolume (**deprecated** starting v1.23, no migration plan and no plan to remove support)
- `gcePersistentDisk` - GCE Persistent Disk (**migration on by default** starting v1.23)
- `portworxVolume` - Portworx volume (**migration on by default** starting v1.31)
- `vsphereVolume` - vSphere VMDK volume (**migration on by default** starting v1.25)

Older versions of Kubernetes also supported the following in-tree PersistentVolume types:

- `cephfs` (**not available** starting v1.31)
- `flocker` - Flocker storage. (**not available** starting v1.25)
- `glusterfs` - GlusterFS storage. (**not available** starting v1.26)
- `photonPersistentDisk` - Photon controller persistent disk. (**not available** starting v1.15)
- `quobyte` - Quobyte volume. (**not available** starting v1.25)
- `rbd` - Rados Block Device (RBD) volume (**not available** starting v1.31)
- `scaleIO` - ScaleIO volume. (**not available** starting v1.21)

- `storageos` - StorageOS volume. (**not available** starting v1.25)

Persistent Volumes

Each PV contains a spec and status, which is the specification and status of the volume. The name of a PersistentVolume object must be a valid DNS subdomain name.

```
apiVersion: v1
kind: PersistentVolumemetadata:  name: pv0003spec:  capacity:    storage: 5Gi  volumeMode: Filesystem  accessModes:    - ReadWrite
```

Note:

Helper programs relating to the volume type may be required for consumption of a PersistentVolume within a cluster. In this example, the PersistentVolume is of type NFS and the helper program /sbin/mount.nfs is required to support the mounting of NFS filesystems.

Capacity

Generally, a PV will have a specific storage capacity. This is set using the PV's `capacity` attribute which is a Quantity value.

Currently, storage size is the only resource that can be set or requested. Future attributes may include IOPS, throughput, etc.

Volume Mode

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

Kubernetes supports two `volumeModes` of PersistentVolumes: `Filesystem` and `Block`.

`volumeMode` is an optional API parameter. `Filesystem` is the default mode used when `volumeMode` parameter is omitted.

A volume with `volumeMode: Filesystem` is *mounted* into Pods into a directory. If the volume is backed by a block device and the device is empty, Kubernetes creates a filesystem on the device before mounting it for the first time.

You can set the value of `volumeMode` to `Block` to use a volume as a raw block device. Such volume is presented into a Pod as a block device, without any filesystem on it. This mode is useful to provide a Pod the fastest possible way to access a volume, without any filesystem layer between the Pod and the volume. On the other hand, the application running in the Pod must know how to handle a raw block device. See Raw Block Volume Support for an example on how to use a volume with `volumeMode: Block` in a Pod.

Access Modes

A PersistentVolume can be mounted on a host in any way supported by the resource provider. As shown in the table below, providers will have different capabilities and each PV's access modes are set to the specific modes supported by that particular volume. For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. Each PV gets its own set of access modes describing that specific PV's capabilities.

The access modes are:

`ReadWriteOnce`
: the volume can be mounted as read-write by a single node. ReadWriteOnce access mode still can allow multiple pods to access (read from or write to) that volume when the pods are running on the same node. For single pod access, please see ReadWriteOncePod.

`ReadOnlyMany`
: the volume can be mounted as read-only by many nodes.

`ReadWriteMany`
: the volume can be mounted as read-write by many nodes.

`ReadWriteOncePod`
: FEATURE STATE: `Kubernetes v1.29 [stable]`
the volume can be mounted as read-write by a single Pod. Use ReadWriteOncePod access mode if you want to ensure that only one pod across the whole cluster can read that PVC or write to it.

Note:

The `ReadWriteOncePod` access mode is only supported for CSI volumes and Kubernetes version 1.22+. To use this feature you will need to update the following CSI sidecars to these versions or greater:

- csi-provisioner:v3.0.0+
- csi-attacher:v3.3.0+
- csi-resizer:v1.3.0+

In the CLI, the access modes are abbreviated to:

- RWO - ReadWriteOnce
- ROX - ReadOnlyMany
- RWX - ReadWriteMany
- RWOP - ReadWriteOncePod

Note:

Kubernetes uses volume access modes to match PersistentVolumeClaims and PersistentVolumes. In some cases, the volume access modes also constrain where the PersistentVolume can be mounted. Volume access modes do **not** enforce write protection once the storage has been mounted. Even if the access modes are specified as ReadWriteOnce, ReadOnlyMany, or ReadWriteMany, they don't set any constraints on the volume. For example, even if a

PersistentVolume is created as ReadOnlyMany, it is no guarantee that it will be read-only. If the access modes are specified as ReadWriteOncePod, the volume is constrained and can be mounted on only a single Pod.

> **Important!** A volume can only be mounted using one access mode at a time, even if it supports many.

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany	ReadWriteOncePod
AzureFile	✓	✓	✓	-
CephFS	✓	✓	✓	-
CSI	depends on the driver	depends on the driver	depends on the driver	depends on the driver
FC	✓	✓	-	-
FlexVolume	✓	✓	depends on the driver	-
HostPath	✓	-	-	-
iSCSI	✓	✓	-	-
NFS	✓	✓	✓	-
RBD	✓	✓	-	-
VsphereVolume	✓	-	- (works when Pods are collocated)	-
PortworxVolume	✓	-	✓	-

Class

A PV can have a class, which is specified by setting the `storageClassName` attribute to the name of a StorageClass. A PV of a particular class can only be bound to PVCs requesting that class. A PV with no `storageClassName` has no class and can only be bound to PVCs that request no particular class.

In the past, the annotation `volume.beta.kubernetes.io/storage-class` was used instead of the `storageClassName` attribute. This annotation is still working; however, it will become fully deprecated in a future Kubernetes release.

Reclaim Policy

Current reclaim policies are:

- Retain -- manual reclamation
- Recycle -- basic scrub (`rm -rf /thevolume/*`)
- Delete -- delete the volume

For Kubernetes 1.34, only `nfs` and `hostPath` volume types support recycling.

Mount Options

A Kubernetes administrator can specify additional mount options for when a Persistent Volume is mounted on a node.

Note:

Not all Persistent Volume types support mount options.

The following volume types support mount options:

- `csi` (including CSI migrated volume types)
- `iscsi`
- `nfs`

Mount options are not validated. If a mount option is invalid, the mount fails.

In the past, the annotation `volume.beta.kubernetes.io/mount-options` was used instead of the `mountOptions` attribute. This annotation is still working; however, it will become fully deprecated in a future Kubernetes release.

Node Affinity

Note:

For most volume types, you do not need to set this field. You need to explicitly set this for local volumes.

A PV can specify node affinity to define constraints that limit what nodes this volume can be accessed from. Pods that use a PV will only be scheduled to nodes that are selected by the node affinity. To specify node affinity, set `nodeAffinity` in the `.spec` of a PV. The PersistentVolume API reference has more details on this field.

Phase

A PersistentVolume will be in one of the following phases:

`Available`
: a free resource that is not yet bound to a claim

`Bound`
: the volume is bound to a claim

`Released`
: the claim has been deleted, but the associated storage resource is not yet reclaimed by the cluster

`Failed`
: the volume has failed its (automated) reclamation

You can see the name of the PVC bound to the PV using `kubectl describe persistentvolume <name>`.

Phase transition timestamp

FEATURE STATE: `Kubernetes v1.31 [stable]` (enabled by default: true)

The `.status` field for a PersistentVolume can include an alpha `lastPhaseTransitionTime` field. This field records the timestamp of when the volume last transitioned its phase. For newly created volumes the phase is set to `Pending` and `lastPhaseTransitionTime` is set to the current time.

PersistentVolumeClaims

Each PVC contains a spec and status, which is the specification and status of the claim. The name of a PersistentVolumeClaim object must be a valid DNS subdomain name.

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: myclaimspec:  accessModes:    - ReadWriteOnce  volumeMode: Filesystem  resources:    r
```

Access Modes

Claims use the same conventions as volumes when requesting storage with specific access modes.

Volume Modes

Claims use the same convention as volumes to indicate the consumption of the volume as either a filesystem or block device.

Volume Name

Claims can use the `volumeName` field to explicitly bind to a specific PersistentVolume. You can also leave `volumeName` unset, indicating that you'd like Kubernetes to set up a new PersistentVolume that matches the claim. If the specified PV is already bound to another PVC, the binding will be stuck in a pending state.

Resources

Claims, like Pods, can request specific quantities of a resource. In this case, the request is for storage. The same resource model applies to both volumes and claims.

Note:

For `Filesystem` volumes, the storage request refers to the "outer" volume size (i.e. the allocated size from the storage backend). This means that the writeable size may be slightly lower for providers that build a filesystem on top of a block device, due to filesystem overhead. This is especially visible with XFS, where many metadata features are enabled by default.

Selector

Claims can specify a label selector to further filter the set of volumes. Only the volumes whose labels match the selector can be bound to the claim. The selector can consist of two fields:

- `matchLabels` - the volume must have a label with this value
- `matchExpressions` - a list of requirements made by specifying key, list of values, and operator that relates the key and values. Valid operators include `In`, `NotIn`, `Exists`, and `DoesNotExist`.

All of the requirements, from both `matchLabels` and `matchExpressions`, are ANDed together – they must all be satisfied in order to match.

Class

A claim can request a particular class by specifying the name of a StorageClass using the attribute `storageClassName`. Only PVs of the requested class, ones with the same `storageClassName` as the PVC, can be bound to the PVC.

PVCs don't necessarily have to request a class. A PVC with its `storageClassName` set equal to `""` is always interpreted to be requesting a PV with no class, so it can only be bound to PVs with no class (no annotation or one set equal to `""`). A PVC with no `storageClassName` is not quite the same and is treated differently by the cluster, depending on whether the `DefaultStorageClass` admission plugin is turned on.

- If the admission plugin is turned on, the administrator may specify a default StorageClass. All PVCs that have no `storageClassName` can be bound only to PVs of that default. Specifying a default StorageClass is done by setting the annotation `storageclass.kubernetes.io/is-default-class` equal to `true` in a StorageClass object. If the administrator does not specify a default, the cluster responds to PVC creation as if the admission plugin were turned off. If more than one default StorageClass is specified, the newest default is used when the PVC is dynamically provisioned.
- If the admission plugin is turned off, there is no notion of a default StorageClass. All PVCs that have `storageClassName` set to `""` can be bound only to PVs that have `storageClassName` also set to `""`. However, PVCs with missing `storageClassName` can be updated later once default StorageClass becomes available. If the PVC gets updated it will no longer bind to PVs that have `storageClassName` also set to `""`.

See retroactive default StorageClass assignment for more details.

Depending on installation method, a default StorageClass may be deployed to a Kubernetes cluster by addon manager during installation.

When a PVC specifies a `selector` in addition to requesting a StorageClass, the requirements are ANDed together: only a PV of the requested class and with the requested labels may be bound to the PVC.

Note:

Currently, a PVC with a non-empty `selector` can't have a PV dynamically provisioned for it.

In the past, the annotation `volume.beta.kubernetes.io/storage-class` was used instead of `storageClassName` attribute. This annotation is still working; however, it won't be supported in a future Kubernetes release.

Retroactive default StorageClass assignment

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

You can create a PersistentVolumeClaim without specifying a `storageClassName` for the new PVC, and you can do so even when no default StorageClass exists in your cluster. In this case, the new PVC creates as you defined it, and the `storageClassName` of that PVC remains unset until default becomes available.

When a default StorageClass becomes available, the control plane identifies any existing PVCs without `storageClassName`. For the PVCs that either have an empty value for `storageClassName` or do not have this key, the control plane then updates those PVCs to set `storageClassName` to match the new default StorageClass. If you have an existing PVC where the `storageClassName` is `""`, and you configure a default StorageClass, then this PVC will not get updated.

In order to keep binding to PVs with `storageClassName` set to `""` (while a default StorageClass is present), you need to set the `storageClassName` of the associated PVC to `""`.

This behavior helps administrators change default StorageClass by removing the old one first and then creating or setting another one. This brief window while there is no default causes PVCs without `storageClassName` created at that time to not have any default, but due to the retroactive default StorageClass assignment this way of changing defaults is safe.

Claims As Volumes

Pods access storage by using the claim as a volume. Claims must exist in the same namespace as the Pod using the claim. The cluster finds the claim in the Pod's namespace and uses it to get the PersistentVolume backing the claim. The volume is then mounted to the host and into the Pod.

```
apiVersion: v1
kind: Podmetadata:  name: mypodspec:  containers:    - name: myfrontend      image: nginx      volumeMounts:      - mountPath: "/v
```

A Note on Namespaces

PersistentVolumes binds are exclusive, and since PersistentVolumeClaims are namespaced objects, mounting claims with "Many" modes (`ROX`, `RWX`) is only possible within one namespace.

PersistentVolumes typed `hostPath`

A `hostPath` PersistentVolume uses a file or directory on the Node to emulate network-attached storage. See an example of `hostPath` typed volume.

Raw Block Volume Support

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

The following volume plugins support raw block volumes, including dynamic provisioning where applicable:

- CSI (including some CSI migrated volume types)
- FC (Fibre Channel)
- iSCSI
- Local volume

PersistentVolume using a Raw Block Volume

```
apiVersion: v1
kind: PersistentVolumemetadata:  name: block-pvspec:  capacity:    storage: 10Gi  accessModes:    - ReadWriteOnce  volumeMode: Blo
```

PersistentVolumeClaim requesting a Raw Block Volume

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: block-pvcspec:  accessModes:    - ReadWriteOnce  volumeMode: Block  resources:    requ
```

Pod specification adding Raw Block Device path in container

```
apiVersion: v1
kind: Podmetadata:  name: pod-with-block-volumespec:  containers:    - name: fc-container      image: fedora:26      command: ["/b
```

Note:

When adding a raw block device for a Pod, you specify the device path in the container instead of a mount path.

Binding Block Volumes

If a user requests a raw block volume by indicating this using the `volumeMode` field in the PersistentVolumeClaim spec, the binding rules differ slightly from previous releases that didn't consider this mode as part of the spec. Listed is a table of possible combinations the user and admin might specify for requesting a raw block device. The table indicates if the volume will be bound or not given the combinations: Volume binding matrix for statically provisioned volumes:

PV volumeMode	PVC volumeMode	Result
unspecified	unspecified	BIND
unspecified	Block	NO BIND
unspecified	Filesystem	BIND
Block	unspecified	NO BIND
Block	Block	BIND
Block	Filesystem	NO BIND
Filesystem	Filesystem	BIND
Filesystem	Block	NO BIND
Filesystem	unspecified	BIND

Note:

Only statically provisioned volumes are supported for alpha release. Administrators should take care to consider these values when working with raw block devices.

Volume Snapshot and Restore Volume from Snapshot Support

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

Volume snapshots only support the out-of-tree CSI volume plugins. For details, see Volume Snapshots. In-tree volume plugins are deprecated. You can read about the deprecated volume plugins in the Volume Plugin FAQ.

Create a PersistentVolumeClaim from a Volume Snapshot

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: restore-pvcspec:  storageClassName: csi-hostpath-sc  dataSource:    name: new-snapshot
```

Volume Cloning

Volume Cloning only available for CSI volume plugins.

Create PersistentVolumeClaim from an existing PVC

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: cloned-pvcspec:  storageClassName: my-csi-plugin  dataSource:    name: existing-src-pv
```

Volume populators and data sources

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

Kubernetes supports custom volume populators. To use custom volume populators, you must enable the `AnyVolumeDataSource` feature gate for the kube-apiserver and kube-controller-manager.

Volume populators take advantage of a PVC spec field called `dataSourceRef`. Unlike the `dataSource` field, which can only contain either a reference to another PersistentVolumeClaim or to a VolumeSnapshot, the `dataSourceRef` field can contain a reference to any object in the same namespace, except for core objects other than PVCs. For clusters that have the feature gate enabled, use of the `dataSourceRef` is preferred over `dataSource`.

Cross namespace data sources

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

Kubernetes supports cross namespace volume data sources. To use cross namespace volume data sources, you must enable the `AnyVolumeDataSource` and `CrossNamespaceVolumeDataSource` feature gates for the kube-apiserver and kube-controller-manager. Also, you must enable the `CrossNamespaceVolumeDataSource` feature gate for the csi-provisioner.

Enabling the `CrossNamespaceVolumeDataSource` feature gate allows you to specify a namespace in the dataSourceRef field.

Note:

When you specify a namespace for a volume data source, Kubernetes checks for a ReferenceGrant in the other namespace before accepting the reference. ReferenceGrant is part of the `gateway.networking.k8s.io` extension APIs. See ReferenceGrant in the Gateway API documentation for details. This means that you must extend your Kubernetes cluster with at least ReferenceGrant from the Gateway API before you can use this mechanism.

Data source references

The `dataSourceRef` field behaves almost the same as the `dataSource` field. If one is specified while the other is not, the API server will give both fields the same value. Neither field can be changed after creation, and attempting to specify different values for the two fields will result in a validation error. Therefore the two fields will always have the same contents.

There are two differences between the `dataSourceRef` field and the `dataSource` field that users should be aware of:

- The `dataSource` field ignores invalid values (as if the field was blank) while the `dataSourceRef` field never ignores values and will cause an error if an invalid value is used. Invalid values are any core object (objects with no apiGroup) except for PVCs.
- The `dataSourceRef` field may contain different types of objects, while the `dataSource` field only allows PVCs and VolumeSnapshots.

When the `CrossNamespaceVolumeDataSource` feature is enabled, there are additional differences:

- The `dataSource` field only allows local objects, while the `dataSourceRef` field allows objects in any namespaces.
- When namespace is specified, `dataSource` and `dataSourceRef` are not synced.

Users should always use `dataSourceRef` on clusters that have the feature gate enabled, and fall back to `dataSource` on clusters that do not. It is not necessary to look at both fields under any circumstance. The duplicated values with slightly different semantics exist only for backwards compatibility. In particular, a mixture of older and newer controllers are able to interoperate because the fields are the same.

Using volume populators

Volume populators are controllers that can create non-empty volumes, where the contents of the volume are determined by a Custom Resource. Users create a populated volume by referring to a Custom Resource using the `dataSourceRef` field:

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: populated-pvcspec:  dataSourceRef:    name: example-name    kind: ExampleDataSource
```

Because volume populators are external components, attempts to create a PVC that uses one can fail if not all the correct components are installed. External controllers should generate events on the PVC to provide feedback on the status of the creation, including warnings if the PVC cannot be created due to some missing component.

You can install the alpha volume data source validator controller into your cluster. That controller generates warning Events on a PVC in the case that no populator is registered to handle that kind of data source. When a suitable populator is installed for a PVC, it's the responsibility of that populator controller to report Events that relate to volume creation and issues during the process.

Using a cross-namespace volume data source

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

Create a ReferenceGrant to allow the namespace owner to accept the reference. You define a populated volume by specifying a cross namespace volume data source using the `dataSourceRef` field. You must already have a valid ReferenceGrant in the source namespace:

```
apiVersion: gateway.networking.k8s.io/v1beta1
kind: ReferenceGrantmetadata:  name: allow-ns1-pvc  namespace: defaultspec:  from:  - group: ""    kind: PersistentVolumeClaim
```

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: foo-pvc  namespace: ns1spec:  storageClassName: example  accessModes:  - ReadWriteOnce
```

Writing Portable Configuration

If you're writing configuration templates or examples that run on a wide range of clusters and need persistent storage, it is recommended that you use the following pattern:

- Include PersistentVolumeClaim objects in your bundle of config (alongside Deployments, ConfigMaps, etc).
- Do not include PersistentVolume objects in the config, since the user instantiating the config may not have permission to create PersistentVolumes.
- Give the user the option of providing a storage class name when instantiating the template.
    - If the user provides a storage class name, put that value into the `persistentVolumeClaim.storageClassName` field. This will cause the PVC to match the right storage class if the cluster has StorageClasses enabled by the admin.
    - If the user does not provide a storage class name, leave the `persistentVolumeClaim.storageClassName` field as nil. This will cause a PV to be automatically provisioned for the user with the default StorageClass in the cluster. Many cluster environments have a default StorageClass installed, or administrators can create their own default StorageClass.
- In your tooling, watch for PVCs that are not getting bound after some time and surface this to the user, as this may indicate that the cluster has no dynamic storage support (in which case the user should create a matching PV) or the cluster has no storage system (in which case the user cannot deploy config requiring PVCs).

What's next

- Learn more about Creating a PersistentVolume.
- Learn more about Creating a PersistentVolumeClaim.
- Read the Persistent Storage design document.

API references

Read about the APIs described in this page:

- `PersistentVolume`
- `PersistentVolumeClaim`

Scheduler Performance Tuning

FEATURE STATE: `Kubernetes v1.14 [beta]`

kube-scheduler is the Kubernetes default scheduler. It is responsible for placement of Pods on Nodes in a cluster.

Nodes in a cluster that meet the scheduling requirements of a Pod are called *feasible* Nodes for the Pod. The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes, picking a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called *Binding*.

This page explains performance tuning optimizations that are relevant for large Kubernetes clusters.

In large clusters, you can tune the scheduler's behaviour balancing scheduling outcomes between latency (new Pods are placed quickly) and accuracy (the scheduler rarely makes poor placement decisions).

You configure this tuning setting via kube-scheduler setting `percentageOfNodesToScore`. This KubeSchedulerConfiguration setting determines a threshold for scheduling nodes in your cluster.

Setting the threshold

The `percentageOfNodesToScore` option accepts whole numeric values between 0 and 100. The value 0 is a special number which indicates that the kube-scheduler should use its compiled-in default. If you set `percentageOfNodesToScore` above 100, kube-scheduler acts as if you had set a value of 100.

To change the value, edit the kube-scheduler configuration file and then restart the scheduler. In many cases, the configuration file can be found at `/etc/kubernetes/config/kube-scheduler.yaml`.

After you have made this change, you can run

```
kubectl get pods -n kube-system | grep kube-scheduler
```

to verify that the kube-scheduler component is healthy.

Node scoring threshold

To improve scheduling performance, the kube-scheduler can stop looking for feasible nodes once it has found enough of them. In large clusters, this saves time compared to a naive approach that would consider every node.

You specify a threshold for how many nodes are enough, as a whole number percentage of all the nodes in your cluster. The kube-scheduler converts this into an integer number of nodes. During scheduling, if the kube-scheduler has identified enough feasible nodes to exceed the configured percentage, the kube-scheduler stops searching for more feasible nodes and moves on to the scoring phase.

How the scheduler iterates over Nodes describes the process in detail.

Default threshold

If you don't specify a threshold, Kubernetes calculates a figure using a linear formula that yields 50% for a 100-node cluster and yields 10% for a 5000-node cluster. The lower bound for the automatic value is 5%.

This means that the kube-scheduler always scores at least 5% of your cluster no matter how large the cluster is, unless you have explicitly set `percentageOfNodesToScore` to be smaller than 5.

If you want the scheduler to score all nodes in your cluster, set `percentageOfNodesToScore` to 100.

Example

Below is an example configuration that sets `percentageOfNodesToScore` to 50%.

```
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfigurationalgorithmSource:  provider: DefaultProvider...percentageOfNodesToScore: 50
```

Tuning percentageOfNodesToScore

`percentageOfNodesToScore` must be a value between 1 and 100 with the default value being calculated based on the cluster size. There is also a hardcoded minimum value of 100 nodes.

Note:

In clusters with less than 100 feasible nodes, the scheduler still checks all the nodes because there are not enough feasible nodes to stop the scheduler's search early.

In a small cluster, if you set a low value for `percentageOfNodesToScore`, your change will have no or little effect, for a similar reason.

If your cluster has several hundred Nodes or fewer, leave this configuration option at its default value. Making changes is unlikely to improve the scheduler's performance significantly.

An important detail to consider when setting this value is that when a smaller number of nodes in a cluster are checked for feasibility, some nodes are not sent to be scored for a given Pod. As a result, a Node which could possibly score a higher value for running the given Pod might not even be passed to the scoring phase. This would result in a less than ideal placement of the Pod.

You should avoid setting `percentageOfNodesToScore` very low so that kube-scheduler does not make frequent, poor Pod placement decisions. Avoid setting the percentage to anything below 10%, unless the scheduler's throughput is critical for your application and the score of nodes is not important. In other words, you prefer to run the Pod on any Node as long as it is feasible.

How the scheduler iterates over Nodes

This section is intended for those who want to understand the internal details of this feature.

In order to give all the Nodes in a cluster a fair chance of being considered for running Pods, the scheduler iterates over the nodes in a round robin fashion. You can imagine that Nodes are in an array. The scheduler starts from the start of the array and checks feasibility of the nodes until it finds enough Nodes as specified by `percentageOfNodesToScore`. For the next Pod, the scheduler continues from the point in the Node array that it stopped at when checking feasibility of Nodes for the previous Pod.

If Nodes are in multiple zones, the scheduler iterates over Nodes in various zones to ensure that Nodes from different zones are considered in the feasibility checks. As an example, consider six nodes in two zones:

```
Zone 1: Node 1, Node 2, Node 3, Node 4
Zone 2: Node 5, Node 6
```

The Scheduler evaluates feasibility of the nodes in this order:

```
Node 1, Node 5, Node 2, Node 6, Node 3, Node 4
```

After going over all the Nodes, it goes back to Node 1.

What's next

- Check the kube-scheduler configuration reference (v1)

Dynamic Resource Allocation

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

This page describes *dynamic resource allocation (DRA)* in Kubernetes.

About DRA

DRA is a Kubernetes feature that lets you request and share resources among Pods. These resources are often attached devices like hardware accelerators.

With DRA, device drivers and cluster admins define device *classes* that are available to *claim* in workloads. Kubernetes allocates matching devices to specific claims and places the corresponding Pods on nodes that can access the allocated devices.

Allocating resources with DRA is a similar experience to dynamic volume provisioning, in which you use PersistentVolumeClaims to claim storage capacity from storage classes and request the claimed capacity in your Pods.

Benefits of DRA

DRA provides a flexible way to categorize, request, and use devices in your cluster. Using DRA provides benefits like the following:

- **Flexible device filtering**: use common expression language (CEL) to perform fine-grained filtering for specific device attributes.
- **Device sharing**: share the same resource with multiple containers or Pods by referencing the corresponding resource claim.
- **Centralized device categorization**: device drivers and cluster admins can use device classes to provide app operators with hardware categories that are optimized for various use cases. For example, you can create a cost-optimized device class for general-purpose workloads, and a high-performance device class for critical jobs.
- **Simplified Pod requests**: with DRA, app operators don't need to specify device quantities in Pod resource requests. Instead, the Pod references a resource claim, and the device configuration in that claim applies to the Pod.

These benefits provide significant improvements in the device allocation workflow when compared to device plugins, which require per-container device requests, don't support device sharing, and don't support expression-based device filtering.

Types of DRA users

The workflow of using DRA to allocate devices involves the following types of users:

- **Device owner**: responsible for devices. Device owners might be commercial vendors, the cluster operator, or another entity. To use DRA, devices must have DRA-compatible drivers that do the following:
    - Create ResourceSlices that provide Kubernetes with information about nodes and resources.
    - Update ResourceSlices when resource capacity in the cluster changes.
    - Optionally, create DeviceClasses that workload operators can use to claim devices.
- **Cluster admin**: responsible for configuring clusters and nodes, attaching devices, installing drivers, and similar tasks. To use DRA, cluster admins do the following:
    - Attach devices to nodes.
    - Install device drivers that support DRA.
    - Optionally, create DeviceClasses that workload operators can use to claim devices.
- **Workload operator**: responsible for deploying and managing workloads in the cluster. To use DRA to allocate devices to Pods, workload operators do the following:
    - Create ResourceClaims or ResourceClaimTemplates to request specific configurations within DeviceClasses.
    - Deploy workloads that use specific ResourceClaims or ResourceClaimTemplates.

DRA terminology

DRA uses the following Kubernetes API kinds to provide the core allocation functionality. All of these API kinds are included in the `resource.k8s.io/v1` API group.

DeviceClass

Defines a category of devices that can be claimed and how to select specific device attributes in claims. The DeviceClass parameters can match zero or more devices in ResourceSlices. To claim devices from a DeviceClass, ResourceClaims select specific device attributes.

ResourceClaim

Describes a request for access to attached resources, such as devices, in the cluster. ResourceClaims provide Pods with access to a specific resource. ResourceClaims can be created by workload operators or generated by Kubernetes based on a ResourceClaimTemplate.

ResourceClaimTemplate

Defines a template that Kubernetes uses to create per-Pod ResourceClaims for a workload. ResourceClaimTemplates provide Pods with access to separate, similar resources. Each ResourceClaim that Kubernetes generates from the template is bound to a specific Pod. When the Pod terminates, Kubernetes deletes the corresponding ResourceClaim.

ResourceSlice

Represents one or more resources that are attached to nodes, such as devices. Drivers create and manage ResourceSlices in the cluster. When a ResourceClaim is created and used in a Pod, Kubernetes uses ResourceSlices to find nodes that have access to the claimed resources. Kubernetes allocates resources to the ResourceClaim and schedules the Pod onto a node that can access the resources.

DeviceClass

A DeviceClass lets cluster admins or device drivers define categories of devices in the cluster. DeviceClasses tell operators what devices they can request and how they can request those devices. You can use common expression language (CEL) to select devices based on specific attributes. A ResourceClaim that references the DeviceClass can then request specific configurations within the DeviceClass.

To create a DeviceClass, see Set Up DRA in a Cluster.

ResourceClaims and ResourceClaimTemplates

A ResourceClaim defines the resources that a workload needs. Every ResourceClaim has *requests* that reference a DeviceClass and select devices from that DeviceClass. ResourceClaims can also use *selectors* to filter for devices that meet specific requirements, and can use *constraints* to limit the devices that can satisfy a request. ResourceClaims can be created by workload operators or can be generated by Kubernetes based on a ResourceClaimTemplate. A ResourceClaimTemplate defines a template that Kubernetes can use to auto-generate ResourceClaims for Pods.

Use cases for ResourceClaims and ResourceClaimTemplates

The method that you use depends on your requirements, as follows:

- **ResourceClaim**: you want multiple Pods to share access to specific devices. You manually manage the lifecycle of ResourceClaims that you create.
- **ResourceClaimTemplate**: you want Pods to have independent access to separate, similarly-configured devices. Kubernetes generates ResourceClaims from the specification in the ResourceClaimTemplate. The lifetime of each generated ResourceClaim is bound to the lifetime of the corresponding Pod.

When you define a workload, you can use Common Expression Language (CEL) to filter for specific device attributes or capacity. The available parameters for filtering depend on the device and the drivers.

If you directly reference a specific ResourceClaim in a Pod, that ResourceClaim must already exist in the same namespace as the Pod. If the ResourceClaim doesn't exist in the namespace, the Pod won't schedule. This behavior is similar to how a PersistentVolumeClaim must exist in the same namespace as a Pod that references it.

You can reference an auto-generated ResourceClaim in a Pod, but this isn't recommended because auto-generated ResourceClaims are bound to the lifetime of the Pod that triggered the generation.

To learn how to claim resources using one of these methods, see Allocate Devices to Workloads with DRA.

Prioritized list

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

You can provide a prioritized list of subrequests for requests in a ResourceClaim or ResourceClaimTemplate. The scheduler will then select the first subrequest that can be allocated. This allows users to specify alternative devices that can be used by the workload if the primary choice is not available.

In the example below, the ResourceClaimTemplate requested a device with the color black and the size large. If a device with those attributes is not available, the pod cannot be scheduled. With the prioritized list feature, a second alternative can be specified, which requests two devices with the color white and size small. The large black device will be allocated if it is available. If it is not, but two small white devices are available, the pod will still be able to run.

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaimTemplatemetadata:  name: prioritized-list-claim-templatespec:  spec:    devices:      requests:      - name: re
```

The decision is made on a per-Pod basis, so if the Pod is a member of a ReplicaSet or similar grouping, you cannot rely on all the members of the group having the same subrequest chosen. Your workload must be able to accommodate this.

Prioritized lists is a *beta feature* and is enabled by default with the `DRAPrioritizedList` feature gate in the kube-apiserver and kube-scheduler.

ResourceSlice

Each ResourceSlice represents one or more devices in a pool. The pool is managed by a device driver, which creates and manages ResourceSlices. The resources in a pool might be represented by a single ResourceSlice or span multiple ResourceSlices.

ResourceSlices provide useful information to device users and to the scheduler, and are crucial for dynamic resource allocation. Every ResourceSlice must include the following information:

- **Resource pool**: a group of one or more resources that the driver manages. The pool can span more than one ResourceSlice. Changes to the resources in a pool must be propagated across all of the ResourceSlices in that pool. The device driver that manages the pool is responsible for ensuring that this propagation happens.

- **Devices**: devices in the managed pool. A ResourceSlice can list every device in a pool or a subset of the devices in a pool. The ResourceSlice defines device information like attributes, versions, and capacity. Device users can select devices for allocation by filtering for device information in ResourceClaims or in DeviceClasses.
- **Nodes**: the nodes that can access the resources. Drivers can choose which nodes can access the resources, whether that's all of the nodes in the cluster, a single named node, or nodes that have specific node labels.

Drivers use a controller to reconcile ResourceSlices in the cluster with the information that the driver has to publish. This controller overwrites any manual changes, such as cluster users creating or modifying ResourceSlices.

Consider the following example ResourceSlice:

```
apiVersion: resource.k8s.io/v1
kind: ResourceSlicemetadata:  name: cat-slicespec:  driver: "resource-driver.example.com"  pool:    generation: 1    name: "black-
```

This ResourceSlice is managed by the `resource-driver.example.com` driver in the `black-cat-pool` pool. The `allNodes: true` field indicates that any node in the cluster can access the devices. There's one device in the ResourceSlice, named `large-black-cat`, with the following attributes:

- `color`: `black`
- `size`: `large`
- `cat`: `true`

A DeviceClass could select this ResourceSlice by using these attributes, and a ResourceClaim could filter for specific devices in that DeviceClass.

How resource allocation with DRA works

The following sections describe the workflow for the various types of DRA users and for the Kubernetes system during dynamic resource allocation.

Workflow for users

1. **Driver creation**: device owners or third-party entities create drivers that can create and manage ResourceSlices in the cluster. These drivers optionally also create DeviceClasses that define a category of devices and how to request them.
2. **Cluster configuration**: cluster admins create clusters, attach devices to nodes, and install the DRA device drivers. Cluster admins optionally create DeviceClasses that define categories of devices and how to request them.
3. **Resource claims**: workload operators create ResourceClaimTemplates or ResourceClaims that request specific device configurations within a DeviceClass. In the same step, workload operators modify their Kubernetes manifests to request those ResourceClaimTemplates or ResourceClaims.

Workflow for Kubernetes

1. **ResourceSlice creation**: drivers in the cluster create ResourceSlices that represent one or more devices in a managed pool of similar devices.

2. **Workload creation**: the cluster control plane checks new workloads for references to ResourceClaimTemplates or to specific ResourceClaims.

   - If the workload uses a ResourceClaimTemplate, a controller named the `resourceclaim-controller` generates ResourceClaims for every Pod in the workload.
   - If the workload uses a specific ResourceClaim, Kubernetes checks whether that ResourceClaim exists in the cluster. If the ResourceClaim doesn't exist, the Pods won't deploy.

3. **ResourceSlice filtering**: for every Pod, Kubernetes checks the ResourceSlices in the cluster to find a device that satisfies all of the following criteria:

   - The nodes that can access the resources are eligible to run the Pod.
   - The ResourceSlice has unallocated resources that match the requirements of the Pod's ResourceClaim.

4. **Resource allocation**: after finding an eligible ResourceSlice for a Pod's ResourceClaim, the Kubernetes scheduler updates the ResourceClaim with the allocation details.

5. **Pod scheduling**: when resource allocation is complete, the scheduler places the Pod on a node that can access the allocated resource. The device driver and the kubelet on that node configure the device and the Pod's access to the device.

Observability of dynamic resources

You can check the status of dynamically allocated resources by using any of the following methods:

- kubelet device metrics
- ResourceClaim status
- Device health monitoring

kubelet device metrics

The `PodResourcesLister` kubelet gRPC service lets you monitor in-use devices. The `DynamicResource` message provides information that's specific to dynamic resource allocation, such as the device name and the claim name. For details, see Monitoring device plugin resources.

ResourceClaim device status

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

DRA drivers can report driver-specific device status data for each allocated device in the `status.devices` field of a ResourceClaim. For example, the driver might list the IP addresses that are assigned to a network interface device.

The accuracy of the information that a driver adds to a ResourceClaim `status.devices` field depends on the driver. Evaluate drivers to decide whether you can rely on this field as the only source of device information.

If you disable the `DRAResourceClaimDeviceStatus` feature gate, the `status.devices` field automatically gets cleared when storing the ResourceClaim. A ResourceClaim device status is supported when it is possible, from a DRA driver, to update an existing ResourceClaim where the `status.devices` field is set.

For details about the `status.devices` field, see the ResourceClaim API reference.

Device Health Monitoring

FEATURE STATE: `Kubernetes v1.31 [alpha]` (enabled by default: false)

As an alpha feature, Kubernetes provides a mechanism for monitoring and reporting the health of dynamically allocated infrastructure resources. For stateful applications running on specialized hardware, it is critical to know when a device has failed or become unhealthy. It is also helpful to find out if the device recovers.

To enable this functionality, the `ResourceHealthStatus` feature gate must be enabled, and the DRA driver must implement the `DRAResourceHealth` gRPC service.

When a DRA driver detects that an allocated device has become unhealthy, it reports this status back to the kubelet. This health information is then exposed directly in the Pod's status. The kubelet populates the `allocatedResourcesStatus` field in the status of each container, detailing the health of each device assigned to that container.

This provides crucial visibility for users and controllers to react to hardware failures. For a Pod that is failing, you can inspect this status to determine if the failure was related to an unhealthy device.

Pre-scheduled Pods

When you - or another API client - create a Pod with `spec.nodeName` already set, the scheduler gets bypassed. If some ResourceClaim needed by that Pod does not exist yet, is not allocated or not reserved for the Pod, then the kubelet will fail to run the Pod and re-check periodically because those requirements might still get fulfilled later.

Such a situation can also arise when support for dynamic resource allocation was not enabled in the scheduler at the time when the Pod got scheduled (version skew, configuration, feature gate, etc.). kube-controller-manager detects this and tries to make the Pod runnable by reserving the required ResourceClaims. However, this only works if those were allocated by the scheduler for some other pod.

It is better to avoid bypassing the scheduler because a Pod that is assigned to a node blocks normal resources (RAM, CPU) that then cannot be used for other Pods while the Pod is stuck. To make a Pod run on a specific node while still going through the normal scheduling flow, create the Pod with a node selector that exactly matches the desired node:

```
apiVersion: v1
kind: Podmetadata:  name: pod-with-catsspec:  nodeSelector:    kubernetes.io/hostname: name-of-the-intended-node  ...
```

You may also be able to mutate the incoming Pod, at admission time, to unset the `.spec.nodeName` field and to use a node selector instead.

DRA beta features

The following sections describe DRA features that are available in the Beta feature stage. For more information, see Set up DRA in the cluster.

Admin access

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

You can mark a request in a ResourceClaim or ResourceClaimTemplate as having privileged features for maintenance and troubleshooting tasks. A request with admin access grants access to in-use devices and may enable additional permissions when making the device available in a container:

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaimTemplatemetadata:  name: large-black-cat-claim-templatespec:  spec:    devices:      requests:      - name: req
```

If this feature is disabled, the `adminAccess` field will be removed automatically when creating such a ResourceClaim.

Admin access is a privileged mode and should not be granted to regular users in multi-tenant clusters. Starting with Kubernetes v1.33, only users authorized to create ResourceClaim or ResourceClaimTemplate objects in namespaces labeled with `resource.k8s.io/admin-access: "true"` (case-sensitive) can use the `adminAccess` field. This ensures that non-admin users cannot misuse the feature. Starting with Kubernetes v1.34, this label has been updated to `resource.kubernetes.io/admin-access: "true"`.

DRA alpha features

The following sections describe DRA features that are available in the Alpha feature stage. To use any of these features, you must also set up DRA in your clusters by enabling the DynamicResourceAllocation feature gate and the DRA API groups. For more information, see Set up DRA in the cluster.

Extended resource allocation by DRA

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

You can provide an extended resource name for a DeviceClass. The scheduler will then select the devices matching the class for the extended resource requests. This allows users to continue using extended resource requests in a pod to request either extended resources provided by device plugin, or DRA devices. The same extended resource can be provided either by device plugin, or DRA on one single cluster node. The same extended resource can be provided by device plugin on some nodes, and DRA on other nodes in the same cluster.

In the example below, the DeviceClass is given an extendedResourceName `example.com/gpu`. If a pod requested for the extended resource `example.com/gpu: 2`, it can be scheduled to a node with two or more devices matching the DeviceClass.

```
apiVersion: resource.k8s.io/v1
kind: DeviceClassmetadata:  name: gpu.example.comspec:  selectors:  - cel:      expression: device.driver == 'gpu.example.com' &&
```

In addition, users can use a special extended resource to allocate devices without having to explicitly create a ResourceClaim. Using the extended resource name prefix `deviceclass.resource.kubernetes.io/` and the DeviceClass name. This works for any DeviceClass, even if it does not specify the an extended resource name. The resulting ResourceClaim will contain a request for an `ExactCount` of the specified number of devices of that DeviceClass.

Extended resource allocation by DRA is an *alpha feature* and only enabled when the `DRAExtendedResource` feature gate is enabled in the kube-apiserver, kube-scheduler, and kubelet.

Partitionable devices

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

Devices represented in DRA don't necessarily have to be a single unit connected to a single machine, but can also be a logical device comprised of multiple devices connected to multiple machines. These devices might consume overlapping resources of the underlying phyical devices, meaning that when one logical device is allocated other devices will no longer be available.

In the ResourceSlice API, this is represented as a list of named CounterSets, each of which contains a set of named counters. The counters represent the resources available on the physical device that are used by the logical devices advertised through DRA.

Logical devices can specify the ConsumesCounters list. Each entry contains a reference to a CounterSet and a set of named counters with the amounts they will consume. So for a device to be allocatable, the referenced counter sets must have sufficient quantity for the counters referenced by the device.

Here is an example of two devices, each consuming 6Gi of memory from the a shared counter with 8Gi of memory. Thus, only one of the devices can be allocated at any point in time. The scheduler handles this and it is transparent to the consumer as the ResourceClaim API is not affected.

```
kind: ResourceSlice
apiVersion: resource.k8s.io/v1metadata:  name: resourceslicespec:  nodeName: worker-1  pool:    name: pool    generation: 1    res
```

Partitionable devices is an *alpha feature* and only enabled when the `DRAPartitionableDevices` feature gate is enabled in the kube-apiserver and kube-scheduler.

Consumable capacity

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

The consumable capacity feature allows the same devices to be consumed by multiple independent ResourceClaims, with the Kubernetes scheduler managing how much of the device's capacity is used up by each claim. This is analogous to how Pods can share the resources on a Node; ResourceClaims can share the resources on a Device.

The device driver can set `allowMultipleAllocations` field added in `.spec.devices` of `ResourceSlice` to allow allocating that device to multiple independent ResourceClaims or to multiple requests within a ResourceClaim.

Users can set `capacity` field added in `spec.devices.requests` of `ResourceClaim` to specify the device resource requirements for each allocation.

For the device that allows multiple allocations, the requested capacity is drawn from — or consumed from — its total capacity, a concept known as **consumable capacity**. Then, the scheduler ensures that the aggregate consumed capacity across all claims does not exceed the device's overall capacity. Furthermore, driver authors can use the `requestPolicy` constraints on individual device capacities to control how those capacities are consumed. For example, the driver author can specify that a given capacity is only consumed in increments of 1Gi.

Here is an example of a network device which allows multiple allocations and contains a consumable bandwidth capacity.

```
kind: ResourceSlice
apiVersion: resource.k8s.io/v1metadata:  name: resourceslicespec:  nodeName: worker-1  pool:    name: pool    generation: 1    res
```

The consumable capacity can be requested as shown in the below example.

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaimTemplatemetadata:  name: bandwidth-claim-templatespec:  spec:    devices:      requests:      - name: req-0
```

The allocation result will include the consumed capacity and the identifier of the share.

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaim...status:  allocation:    devices:      results:      - consumedCapacity:          bandwidth: 1G        device
```

In this example, a multiply-allocatable device was chosen. However, any `resource.example.com` device with at least the requested 1G bandwidth could have met the requirement. If a non-multiply-allocatable device were chosen, the allocation would have resulted in the entire device. To force the use of a only multiply-allocatable devices, you can use the CEL criteria `device.allowMultipleAllocations == true`.

Device taints and tolerations

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

Device taints are similar to node taints: a taint has a string key, a string value, and an effect. The effect is applied to the ResourceClaim which is using a tainted device and to all Pods referencing that ResourceClaim. The "NoSchedule" effect prevents scheduling those Pods. Tainted devices are ignored when trying to allocate a ResourceClaim because using them would prevent scheduling of Pods.

The "NoExecute" effect implies "NoSchedule" and in addition causes eviction of all Pods which have been scheduled already. This eviction is implemented in the device taint eviction controller in kube-controller-manager by deleting affected Pods.

ResourceClaims can tolerate taints. If a taint is tolerated, its effect does not apply. An empty toleration matches all taints. A toleration can be limited to certain effects and/or match certain key/value pairs. A toleration can check that a certain key exists, regardless which value it has, or it can check for specific values of a key. For more information on this matching see the node taint concepts.

Eviction can be delayed by tolerating a taint for a certain duration. That delay starts at the time when a taint gets added to a device, which is recorded in a field of the taint.

Taints apply as described above also to ResourceClaims allocating "all" devices on a node. All devices must be untainted or all of their taints must be tolerated. Allocating a device with admin access (described above) is not exempt either. An admin using that mode must explicitly tolerate all taints to access tainted devices.

Device taints and tolerations is an *alpha feature* and only enabled when the `DRADeviceTaints` feature gate is enabled in the kube-apiserver, kube-controller-manager and kube-scheduler. To use DeviceTaintRules, the `resource.k8s.io/v1alpha3` API version must be enabled.

You can add taints to devices in the following ways, by using the DeviceTaintRule API kind.

Taints set by the driver

A DRA driver can add taints to the device information that it publishes in ResourceSlices. Consult the documentation of a DRA driver to learn whether the driver uses taints and what their keys and values are.

Taints set by an admin

An admin or a control plane component can taint devices without having to tell the DRA driver to include taints in its device information in ResourceSlices. They do that by creating DeviceTaintRules. Each DeviceTaintRule adds one taint to devices which match the device selector. Without such a selector, no devices are tainted. This makes it harder to accidentally evict all pods using ResourceClaims when leaving out the selector by mistake.

Devices can be selected by giving the name of a DeviceClass, driver, pool, and/or device. The DeviceClass selects all devices that are selected by the selectors in that DeviceClass. With just the driver name, an admin can taint all devices managed by that driver, for example while doing some kind of maintenance of that driver across the entire cluster. Adding a pool name can limit the taint to a single node, if the driver manages node-local devices.

Finally, adding the device name can select one specific device. The device name and pool name can also be used alone, if desired. For example, drivers for node-local devices are encouraged to use the node name as their pool name. Then tainting with that pool name automatically taints all devices on a node.

Drivers might use stable names like "gpu-0" that hide which specific device is currently assigned to that name. To support tainting a specific hardware instance, CEL selectors can be used in a DeviceTaintRule to match a vendor-specific unique ID attribute, if the driver supports one for its hardware.

The taint applies as long as the DeviceTaintRule exists. It can be modified and and removed at any time. Here is one example of a DeviceTaintRule for a fictional DRA driver:

```
apiVersion: resource.k8s.io/v1alpha3
kind: DeviceTaintRulemetadata:  name: examplespec:  # The entire hardware installation for this  # particular driver is broken.  #
```

Device Binding Conditions

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

Device Binding Conditions allow the Kubernetes scheduler to delay Pod binding until external resources, such as fabric-attached GPUs or reprogrammable FPGAs, are confirmed to be ready.

This waiting behavior is implemented in the PreBind phase of the scheduling framework. During this phase, the scheduler checks whether all required device conditions are satisfied before proceeding with binding.

This improves scheduling reliability by avoiding premature binding and enables coordination with external device controllers.

To use this feature, device drivers (typically managed by driver owners) must publish the following fields in the `Device` section of a `ResourceSlice`. Cluster administrators must enable the `DRADeviceBindingConditions` and `DRAResourceClaimDeviceStatus` feature gates for the scheduler to honor these fields.

- `bindingConditions`: A list of condition types that must be set to True in the status.conditions field of the associated ResourceClaim before the Pod can be bound. These typically represent readiness signals such as "DeviceAttached" or "DeviceInitialized".
- `bindingFailureConditions`: A list of condition types that, if set to True in status.conditions field of the associated ResourceClaim, indicate a failure state. If any of these conditions are True, the scheduler will abort binding and reschedule the Pod.
- `bindsToNode`: if set to `true`, the scheduler records the selected node name in the `status.allocation.nodeSelector` field of the ResourceClaim. This does not affect the Pod's `spec.nodeSelector`. Instead, it sets a node selector inside the ResourceClaim, which external controllers can use to perform node-specific operations such as device attachment or preparation.

All condition types listed in bindingConditions and bindingFailureConditions are evaluated from the `status.conditions` field of the ResourceClaim. External controllers are responsible for updating these conditions using standard Kubernetes condition semantics (`type`, `status`, `reason`, `message`, `lastTransitionTime`).

The scheduler waits up to **600 seconds** for all `bindingConditions` to become `True`. If the timeout is reached or any `bindingFailureConditions` are `True`, the scheduler clears the allocation and reschedules the Pod.

```
apiVersion: resource.k8s.io/v1
kind: ResourceSlicemetadata:  name: gpu-slicespec:  driver: dra.example.com  nodeSelector:    nodeSelectorTerms:    - matchExpress
```

This example ResourceSlice has the following properties:

- The ResourceSlice targets nodes labeled with `accelerator-type=high-performance`, so that the scheduler uses only a specific set of eligible nodes.
- The scheduler selects one node from the selected group (for example, `node-3`) and sets the `status.allocation.nodeSelector` field in the ResourceClaim to that node name.
- The `dra.example.com/is-prepared` binding condition indicates that the device `gpu-1` must be prepared (the `is-prepared` condition has a status of `True`) before binding.
- If the `gpu-1` device preparation fails (the `preparing-failed` condition has a status of `True`), the scheduler aborts binding.

- The scheduler waits up to 600 seconds for the device to become ready.
- External controllers can use the node selector in the ResourceClaim to perform node-specific setup on the selected node.

What's next

- Set Up DRA in a Cluster
- Allocate devices to workloads using DRA
- For more information on the design, see the Dynamic Resource Allocation with Structured Parameters KEP.

Volume Snapshot Classes

This document describes the concept of VolumeSnapshotClass in Kubernetes. Familiarity with volume snapshots and storage classes is suggested.

Introduction

Just like StorageClass provides a way for administrators to describe the "classes" of storage they offer when provisioning a volume, VolumeSnapshotClass provides a way to describe the "classes" of storage when provisioning a volume snapshot.

The VolumeSnapshotClass Resource

Each VolumeSnapshotClass contains the fields `driver`, `deletionPolicy`, and `parameters`, which are used when a VolumeSnapshot belonging to the class needs to be dynamically provisioned.

The name of a VolumeSnapshotClass object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating VolumeSnapshotClass objects, and the objects cannot be updated once they are created.

Note:

Installation of the CRDs is the responsibility of the Kubernetes distribution. Without the required CRDs present, the creation of a VolumeSnapshotClass fails.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClassmetadata:  name: csi-hostpath-snapclassdriver: hostpath.csi.k8s.iodeletionPolicy: Deleteparameters:
```

Administrators can specify a default VolumeSnapshotClass for VolumeSnapshots that don't request any particular class to bind to by adding the `snapshot.storage.kubernetes.io/is-default-class: "true"` annotation:

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClassmetadata:  name: csi-hostpath-snapclass  annotations:    snapshot.storage.kubernetes.io/is-default-class:
```

If multiple CSI drivers exist, a default VolumeSnapshotClass can be specified for each of them.

VolumeSnapshotClass dependencies

When you create a VolumeSnapshot without specifying a VolumeSnapshotClass, Kubernetes automatically selects a default VolumeSnapshotClass that has a CSI driver matching the CSI driver of the PVC’s StorageClass.

This behavior allows multiple default VolumeSnapshotClass objects to coexist in a cluster, as long as each one is associated with a unique CSI driver.

Always ensure that there is only one default VolumeSnapshotClass for each CSI driver. If multiple default VolumeSnapshotClass objects are created using the same CSI driver, a VolumeSnapshot creation will fail because Kubernetes cannot determine which one to use.

Driver

Volume snapshot classes have a driver that determines what CSI volume plugin is used for provisioning VolumeSnapshots. This field must be specified.

DeletionPolicy

Volume snapshot classes have a deletionPolicy. It enables you to configure what happens to a VolumeSnapshotContent when the VolumeSnapshot object it is bound to is to be deleted. The deletionPolicy of a volume snapshot class can either be `Retain` or `Delete`. This field must be specified.

If the deletionPolicy is `Delete`, then the underlying storage snapshot will be deleted along with the VolumeSnapshotContent object. If the deletionPolicy is `Retain`, then both the underlying snapshot and VolumeSnapshotContent remain.

Parameters

Volume snapshot classes have parameters that describe volume snapshots belonging to the volume snapshot class. Different parameters may be accepted depending on the `driver`.

Resource Bin Packing

In the scheduling-plugin `NodeResourcesFit` of kube-scheduler, there are two scoring strategies that support the bin packing of resources: `MostAllocated` and `RequestedToCapacityRatio`.

Enabling bin packing using MostAllocated strategy

The `MostAllocated` strategy scores the nodes based on the utilization of resources, favoring the ones with higher allocation. For each resource type, you can set a weight to modify its influence in the node score.

To set the `MostAllocated` strategy for the `NodeResourcesFit` plugin, use a scheduler configuration similar to the following:

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfigurationprofiles:- pluginConfig:  - args:      scoringStrategy:        resources:         - name: cpu
```

To learn more about other parameters and their default configuration, see the API documentation for `NodeResourcesFitArgs`.

Enabling bin packing using RequestedToCapacityRatio

The `RequestedToCapacityRatio` strategy allows the users to specify the resources along with weights for each resource to score nodes based on the request to capacity ratio. This allows users to bin pack extended resources by using appropriate parameters to improve the utilization of scarce resources in large clusters. It favors nodes according to a configured function of the allocated resources. The behavior of the `RequestedToCapacityRatio` in the `NodeResourcesFit` score function can be controlled by the scoringStrategy field. Within the `scoringStrategy` field, you can configure two parameters: `requestedToCapacityRatio` and `resources`. The `shape` in the `requestedToCapacityRatio` parameter allows the user to tune the function as least requested or most requested based on `utilization` and `score` values. The `resources` parameter comprises both the `name` of the resource to be considered during scoring and its corresponding `weight`, which specifies the weight of each resource.

Below is an example configuration that sets the bin packing behavior for extended resources `intel.com/foo` and `intel.com/bar` using the `requestedToCapacityRatio` field.

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfigurationprofiles:- pluginConfig:  - args:      scoringStrategy:        resources:         - name: intel.com
```

Referencing the `KubeSchedulerConfiguration` file with the kube-scheduler flag `--config=/path/to/config/file` will pass the configuration to the scheduler.

To learn more about other parameters and their default configuration, see the API documentation for `NodeResourcesFitArgs`.

Tuning the score function

`shape` is used to specify the behavior of the `RequestedToCapacityRatio` function.

```
shape:
  - utilization: 0
    score: 0
  - utilization: 100
    score: 10
```

The above arguments give the node a `score` of 0 if `utilization` is 0% and 10 for `utilization` 100%, thus enabling bin packing behavior. To enable least requested the score value must be reversed as follows.

```
shape:
  - utilization: 0
    score: 10
  - utilization: 100
    score: 0
```

`resources` is an optional parameter which defaults to:

```
resources:
  - name: cpu
    weight: 1
  - name: memory
    weight: 1
```

It can be used to add extended resources as follows:

```
resources:
  - name: intel.com/foo
    weight: 5
  - name: cpu
    weight: 3
  - name: memory
    weight: 1
```

The `weight` parameter is optional and is set to 1 if not specified. Also, the `weight` cannot be set to a negative value.

Node scoring for capacity allocation

This section is intended for those who want to understand the internal details of this feature. Below is an example of how the node score is calculated for a given set of values.

Requested resources:

```
intel.com/foo : 2
memory: 256MB
cpu: 2
```

Resource weights:

```
intel.com/foo : 5
memory: 1
cpu: 3
```

FunctionShapePoint $\{\{0, 0\}, \{100, 10\}\}$

Node 1 spec:

```
Available:
  intel.com/foo: 4
  memory: 1 GB
  cpu: 8

Used:
  intel.com/foo: 1
  memory: 256MB
  cpu: 1
```

Node score:

```
intel.com/foo  = resourceScoringFunction((2+1),4)
               = (100 - ((4-3)*100/4))
               = (100 - 25)
               = 75                       # requested + used = 75% * available
               = rawScoringFunction(75)
               = 7                        # floor(75/10)

memory         = resourceScoringFunction((256+256),1024)
               = (100 -((1024-512)*100/1024))
               = 50                       # requested + used = 50% * available
               = rawScoringFunction(50)
               = 5                        # floor(50/10)

cpu            = resourceScoringFunction((2+1),8)
               = (100 -((8-3)*100/8))
               = 37.5                     # requested + used = 37.5% * available
               = rawScoringFunction(37.5)
               = 3                        # floor(37.5/10)

NodeScore   =  ((7 * 5) + (5 * 1) + (3 * 3)) / (5 + 1 + 3)
            =  5
```

Node 2 spec:

```
Available:
  intel.com/foo: 8
  memory: 1GB
  cpu: 8
Used:
  intel.com/foo: 2
  memory: 512MB
  cpu: 6
```

Node score:

```
intel.com/foo  = resourceScoringFunction((2+2),8)
               =  (100 - ((8-4)*100/8)
               =  (100 - 50)
               =  50
               =  rawScoringFunction(50)
               = 5

memory         = resourceScoringFunction((256+512),1024)
               = (100 -((1024-768)*100/1024))
               = 75
               = rawScoringFunction(75)
               = 7

cpu            = resourceScoringFunction((2+6),8)
               = (100 -((8-8)*100/8))
               = 100
               = rawScoringFunction(100)
               = 10

NodeScore   =  ((5 * 5) + (7 * 1) + (10 * 3)) / (5 + 1 + 3)
            =  7
```

What's next

- Read more about the scheduling framework
- Read more about scheduler configuration

Pod Priority and Preemption

FEATURE STATE: `Kubernetes v1.14 [stable]`

Pods can have *priority*. Priority indicates the importance of a Pod relative to other Pods. If a Pod cannot be scheduled, the scheduler tries to preempt (evict) lower priority Pods to make scheduling of the pending Pod possible.

Warning:

In a cluster where not all users are trusted, a malicious user could create Pods at the highest possible priorities, causing other Pods to be evicted/not get scheduled. An administrator can use ResourceQuota to prevent users from creating pods at high priorities.

See limit Priority Class consumption by default for details.

How to use priority and preemption

To use priority and preemption:

1. Add one or more PriorityClasses.
2. Create Pods with`priorityClassName` set to one of the added PriorityClasses. Of course you do not need to create the Pods directly; normally you would add `priorityClassName` to the Pod template of a collection object like a Deployment.

Keep reading for more information about these steps.

Note:

Kubernetes already ships with two PriorityClasses: `system-cluster-critical` and `system-node-critical`. These are common classes and are used to ensure that critical components are always scheduled first.

PriorityClass

A PriorityClass is a non-namespaced object that defines a mapping from a priority class name to the integer value of the priority. The name is specified in the `name` field of the PriorityClass object's metadata. The value is specified in the required `value` field. The higher the value, the higher the priority. The name of a PriorityClass object must be a valid DNS subdomain name, and it cannot be prefixed with `system-`.

A PriorityClass object can have any 32-bit integer value smaller than or equal to 1 billion. This means that the range of values for a PriorityClass object is from -2147483648 to 1000000000 inclusive. Larger numbers are reserved for built-in PriorityClasses that represent critical system Pods. A cluster admin should create one PriorityClass object for each such mapping that they want.

PriorityClass also has two optional fields: `globalDefault` and `description`. The `globalDefault` field indicates that the value of this PriorityClass should be used for Pods without a `priorityClassName`. Only one PriorityClass with `globalDefault` set to true can exist in the system. If there is no PriorityClass with `globalDefault` set, the priority of Pods with no `priorityClassName` is zero.

The `description` field is an arbitrary string. It is meant to tell users of the cluster when they should use this PriorityClass.

Notes about PodPriority and existing clusters

- If you upgrade an existing cluster without this feature, the priority of your existing Pods is effectively zero.
- Addition of a PriorityClass with `globalDefault` set to `true` does not change the priorities of existing Pods. The value of such a PriorityClass is used only for Pods created after the PriorityClass is added.
- If you delete a PriorityClass, existing Pods that use the name of the deleted PriorityClass remain unchanged, but you cannot create more Pods that use the name of the deleted PriorityClass.

Example PriorityClass

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClassmetadata:  name: high-priorityvalue: 1000000globalDefault: falsedescription: "This priority class should be use
```

Non-preempting PriorityClass

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

Pods with `preemptionPolicy: Never` will be placed in the scheduling queue ahead of lower-priority pods, but they cannot preempt other pods. A non-preempting pod waiting to be scheduled will stay in the scheduling queue, until sufficient resources are free, and it can be scheduled. Non-preempting pods, like other pods, are subject to scheduler back-off. This means that if the scheduler tries these pods and they cannot be scheduled, they will be retried with lower frequency, allowing other pods with lower priority to be scheduled before them.

Non-preempting pods may still be preempted by other, high-priority pods.

`preemptionPolicy` defaults to `PreemptLowerPriority`, which will allow pods of that PriorityClass to preempt lower-priority pods (as is existing default behavior). If `preemptionPolicy` is set to `Never`, pods in that PriorityClass will be non-preempting.

An example use case is for data science workloads. A user may submit a job that they want to be prioritized above other workloads, but do not wish to discard existing work by preempting running pods. The high priority job with `preemptionPolicy: Never` will be scheduled ahead of other queued pods, as soon as sufficient cluster resources "naturally" become free.

Example Non-preempting PriorityClass

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClassmetadata:  name: high-priority-nonpreemptingvalue: 1000000preemptionPolicy: NeverglobalDefault: falsedescriptio
```

Pod priority

After you have one or more PriorityClasses, you can create Pods that specify one of those PriorityClass names in their specifications. The priority admission controller uses the `priorityClassName` field and populates the integer value of the priority. If the priority class is not found, the Pod is rejected.

The following YAML is an example of a Pod configuration that uses the PriorityClass created in the preceding example. The priority admission controller checks the specification and resolves the priority of the Pod to 1000000.

```
apiVersion: v1
kind: Podmetadata:  name: nginx  labels:    env: testspec:  containers:  - name: nginx    image: nginx    imagePullPolicy: IfNotPr
```

Effect of Pod priority on scheduling order

When Pod priority is enabled, the scheduler orders pending Pods by their priority and a pending Pod is placed ahead of other pending Pods with lower priority in the scheduling queue. As a result, the higher priority Pod may be scheduled sooner than Pods with lower priority if its scheduling requirements are met. If such Pod cannot be scheduled, the scheduler will continue and try to schedule other lower priority Pods.

Preemption

When Pods are created, they go to a queue and wait to be scheduled. The scheduler picks a Pod from the queue and tries to schedule it on a Node. If no Node is found that satisfies all the specified requirements of the Pod, preemption logic is triggered for the pending Pod. Let's call the pending Pod P. Preemption logic tries to find a Node where removal of one or more Pods with lower priority than P would enable P to be scheduled on that Node. If such a Node is found, one or more lower priority Pods get evicted from the Node. After the Pods are gone, P can be scheduled on the Node.

User exposed information

When Pod P preempts one or more Pods on Node N, `nominatedNodeName` field of Pod P's status is set to the name of Node N. This field helps the scheduler track resources reserved for Pod P and also gives users information about preemptions in their clusters.

Please note that Pod P is not necessarily scheduled to the "nominated Node". The scheduler always tries the "nominated Node" before iterating over any other nodes. After victim Pods are preempted, they get their graceful termination period. If another node becomes available while scheduler is waiting for the victim Pods to terminate, scheduler may use the other node to schedule Pod P. As a result `nominatedNodeName` and `nodeName` of Pod spec are not always the same. Also, if the scheduler preempts Pods on Node N, but then a higher priority Pod than Pod P arrives, the scheduler may give Node N to the new higher priority Pod. In such a case, scheduler clears `nominatedNodeName` of Pod P. By doing this, scheduler makes Pod P eligible to preempt Pods on another Node.

Limitations of preemption

Graceful termination of preemption victims

When Pods are preempted, the victims get their graceful termination period. They have that much time to finish their work and exit. If they don't, they are killed. This graceful termination period creates a time gap between the point that the scheduler preempts Pods and the time when the pending Pod (P) can be scheduled on the Node (N). In the meantime, the scheduler keeps scheduling other pending Pods. As victims exit or get terminated, the scheduler tries to schedule Pods in the pending queue. Therefore, there is usually a time gap between the point that scheduler preempts victims and the time that Pod P is scheduled. In order to minimize this gap, one can set graceful termination period of lower priority Pods to zero or a small number.

PodDisruptionBudget is supported, but not guaranteed

A PodDisruptionBudget (PDB) allows application owners to limit the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. Kubernetes supports PDB when preempting Pods, but respecting PDB is best effort. The scheduler tries to find victims whose PDB are not violated by preemption, but if no such victims are found, preemption will still happen, and lower priority Pods will be removed despite their PDBs being violated.

Inter-Pod affinity on lower-priority Pods

A Node is considered for preemption only when the answer to this question is yes: "If all the Pods with lower priority than the pending Pod are removed from the Node, can the pending Pod be scheduled on the Node?"

Note:

Preemption does not necessarily remove all lower-priority Pods. If the pending Pod can be scheduled by removing fewer than all lower-priority Pods, then only a portion of the lower-priority Pods are removed. Even so, the answer to the preceding question must be yes. If the answer is no, the Node is not considered for preemption.

If a pending Pod has inter-pod affinity to one or more of the lower-priority Pods on the Node, the inter-Pod affinity rule cannot be satisfied in the absence of those lower-priority Pods. In this case, the scheduler does not preempt any Pods on the Node. Instead, it looks for another Node. The scheduler might find a suitable Node or it might not. There is no guarantee that the pending Pod can be scheduled.

Our recommended solution for this problem is to create inter-Pod affinity only towards equal or higher priority Pods.

Cross node preemption

Suppose a Node N is being considered for preemption so that a pending Pod P can be scheduled on N. P might become feasible on N only if a Pod on another Node is preempted. Here's an example:

- Pod P is being considered for Node N.
- Pod Q is running on another Node in the same Zone as Node N.
- Pod P has Zone-wide anti-affinity with Pod Q (`topologyKey: topology.kubernetes.io/zone`).
- There are no other cases of anti-affinity between Pod P and other Pods in the Zone.
- In order to schedule Pod P on Node N, Pod Q can be preempted, but scheduler does not perform cross-node preemption. So, Pod P will be deemed unschedulable on Node N.

If Pod Q were removed from its Node, the Pod anti-affinity violation would be gone, and Pod P could possibly be scheduled on Node N.

We may consider adding cross Node preemption in future versions if there is enough demand and if we find an algorithm with reasonable performance.

Troubleshooting

Pod priority and preemption can have unwanted side effects. Here are some examples of potential problems and ways to deal with them.

Pods are preempted unnecessarily

Preemption removes existing Pods from a cluster under resource pressure to make room for higher priority pending Pods. If you give high priorities to certain Pods by mistake, these unintentionally high priority Pods may cause preemption in your cluster. Pod priority is specified by setting the `priorityClassName` field in the Pod's specification. The integer value for priority is then resolved and populated to the `priority` field of `podSpec`.

To address the problem, you can change the `priorityClassName` for those Pods to use lower priority classes, or leave that field empty. An empty `priorityClassName` is resolved to zero by default.

When a Pod is preempted, there will be events recorded for the preempted Pod. Preemption should happen only when a cluster does not have enough resources for a Pod. In such cases, preemption happens only when the priority of the pending Pod (preemptor) is higher than the victim Pods. Preemption must not happen when there is no pending Pod, or when the pending Pods have equal or lower priority than the victims. If preemption happens in such scenarios, please file an issue.

Pods are preempted, but the preemptor is not scheduled

When pods are preempted, they receive their requested graceful termination period, which is by default 30 seconds. If the victim Pods do not terminate within this period, they are forcibly terminated. Once all the victims go away, the preemptor Pod can be scheduled.

While the preemptor Pod is waiting for the victims to go away, a higher priority Pod may be created that fits on the same Node. In this case, the scheduler will schedule the higher priority Pod instead of the preemptor.

This is expected behavior: the Pod with the higher priority should take the place of a Pod with a lower priority.

Higher priority Pods are preempted before lower priority pods

The scheduler tries to find nodes that can run a pending Pod. If no node is found, the scheduler tries to remove Pods with lower priority from an arbitrary node in order to make room for the pending pod. If a node with low priority Pods is not feasible to run the pending Pod, the scheduler may choose another node with higher priority Pods (compared to the Pods on the other node) for preemption. The victims must still have lower priority than the preemptor Pod.

When there are multiple nodes available for preemption, the scheduler tries to choose the node with a set of Pods with lowest priority. However, if such Pods have PodDisruptionBudget that would be violated if they are preempted then the scheduler may choose another node with higher priority Pods.

When multiple nodes exist for preemption and none of the above scenarios apply, the scheduler chooses a node with the lowest priority.

Interactions between Pod priority and quality of service

Pod priority and QoS class are two orthogonal features with few interactions and no default restrictions on setting the priority of a Pod based on its QoS classes. The scheduler's preemption logic does not consider QoS when choosing preemption targets. Preemption considers Pod priority and attempts to choose a set of targets with the lowest priority. Higher-priority Pods are considered for preemption only if the removal of the lowest priority Pods is not sufficient to allow the scheduler to schedule the preemptor Pod, or if the lowest priority Pods are protected by `PodDisruptionBudget`.

The kubelet uses Priority to determine pod order for node-pressure eviction. You can use the QoS class to estimate the order in which pods are most likely to get evicted. The kubelet ranks pods for eviction based on the following factors:

1. Whether the starved resource usage exceeds requests
2. Pod Priority
3. Amount of resource usage relative to requests

See Pod selection for kubelet eviction for more details.

kubelet node-pressure eviction does not evict Pods when their usage does not exceed their requests. If a Pod with lower priority is not exceeding its requests, it won't be evicted. Another Pod with higher priority that exceeds its requests may be evicted.

What's next

- Read about using ResourceQuotas in connection with PriorityClasses: limit Priority Class consumption by default
- Learn about Pod Disruption
- Learn about API-initiated Eviction
- Learn about Node-pressure Eviction

Volume Health Monitoring

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

CSI volume health monitoring allows CSI Drivers to detect abnormal volume conditions from the underlying storage systems and report them as events on PVCs or Pods.

Volume health monitoring

Kubernetes *volume health monitoring* is part of how Kubernetes implements the Container Storage Interface (CSI). Volume health monitoring feature is implemented in two components: an External Health Monitor controller, and the kubelet.

If a CSI Driver supports Volume Health Monitoring feature from the controller side, an event will be reported on the related PersistentVolumeClaim (PVC) when an abnormal volume condition is detected on a CSI volume.

The External Health Monitor controller also watches for node failure events. You can enable node failure monitoring by setting the `enable-node-watcher` flag to true. When the external health monitor detects a node failure event, the controller reports an Event will be reported on the PVC to indicate that pods using this PVC are on a failed node.

If a CSI Driver supports Volume Health Monitoring feature from the node side, an Event will be reported on every Pod using the PVC when an abnormal volume condition is detected on a CSI volume. In addition, Volume Health information is exposed as Kubelet VolumeStats metrics. A new metric kubelet_volume_stats_health_status_abnormal is added. This metric includes two labels: `namespace` and `persistentvolumeclaim`. The count is either 1 or 0. 1 indicates the volume is unhealthy, 0 indicates volume is healthy. For more information, please check KEP.

Note:

You need to enable the `CSIVolumeHealth` feature gate to use this feature from the node side.

What's next

See the CSI driver documentation to find out which CSI drivers have implemented this feature.

Role Based Access Control Good Practices

Principles and practices for good RBAC design for cluster operators.

Kubernetes RBAC is a key security control to ensure that cluster users and workloads have only the access to resources required to execute their roles. It is important to ensure that, when designing permissions for cluster users, the cluster administrator understands the areas where privilege escalation could occur, to reduce the risk of excessive access leading to security incidents.

The good practices laid out here should be read in conjunction with the general RBAC documentation.

General good practice

Least privilege

Ideally, minimal RBAC rights should be assigned to users and service accounts. Only permissions explicitly required for their operation should be used. While each cluster will be different, some general rules that can be applied are :

- Assign permissions at the namespace level where possible. Use RoleBindings as opposed to ClusterRoleBindings to give users rights only within a specific namespace.
- Avoid providing wildcard permissions when possible, especially to all resources. As Kubernetes is an extensible system, providing wildcard access gives rights not just to all object types that currently exist in the cluster, but also to all object types which are created in the future.
- Administrators should not use `cluster-admin` accounts except where specifically needed. Providing a low privileged account with impersonation rights can avoid accidental modification of cluster resources.
- Avoid adding users to the `system:masters` group. Any user who is a member of this group bypasses all RBAC rights checks and will always have unrestricted superuser access, which cannot be revoked by removing RoleBindings or ClusterRoleBindings. As an aside, if a cluster is using an authorization webhook, membership of this group also bypasses that webhook (requests from users who are members of that group are never sent to the webhook)

Minimize distribution of privileged tokens

Ideally, pods shouldn't be assigned service accounts that have been granted powerful permissions (for example, any of the rights listed under privilege escalation risks). In cases where a workload requires powerful permissions, consider the following practices:

- Limit the number of nodes running powerful pods. Ensure that any DaemonSets you run are necessary and are run with least privilege to limit the blast radius of container escapes.
- Avoid running powerful pods alongside untrusted or publicly-exposed ones. Consider using Taints and Toleration, NodeAffinity, or PodAntiAffinity to ensure pods don't run alongside untrusted or less-trusted Pods. Pay special attention to situations where less-trustworthy Pods are not meeting the **Restricted** Pod Security Standard.

Hardening

Kubernetes defaults to providing access which may not be required in every cluster. Reviewing the RBAC rights provided by default can provide opportunities for security hardening. In general, changes should not be made to rights provided to `system:` accounts some options to harden cluster rights exist:

- Review bindings for the `system:unauthenticated` group and remove them where possible, as this gives access to anyone who can contact the API server at a network level.
- Avoid the default auto-mounting of service account tokens by setting `automountServiceAccountToken: false`. For more details, see using default service account token. Setting this value for a Pod will overwrite the service account setting, workloads which require service account tokens can still mount them.

Periodic review

It is vital to periodically review the Kubernetes RBAC settings for redundant entries and possible privilege escalations. If an attacker is able to create a user account with the same name as a deleted user, they can automatically inherit all the rights of the deleted user, especially the rights assigned to that user.

Kubernetes RBAC - privilege escalation risks

Within Kubernetes RBAC there are a number of privileges which, if granted, can allow a user or a service account to escalate their privileges in the cluster or affect systems outside the cluster.

This section is intended to provide visibility of the areas where cluster operators should take care, to ensure that they do not inadvertently allow for more access to clusters than intended.

Listing secrets

It is generally clear that allowing `get` access on Secrets will allow a user to read their contents. It is also important to note that `list` and `watch` access also effectively allow for users to reveal the Secret contents. For example, when a List response is returned (for example, via `kubectl get secrets -A -o yaml`), the response includes the contents of all Secrets.

Workload creation

Permission to create workloads (either Pods, or workload resources that manage Pods) in a namespace implicitly grants access to many other resources in that namespace, such as Secrets, ConfigMaps, and PersistentVolumes that can be mounted in Pods. Additionally, since Pods can run as any ServiceAccount, granting permission to create workloads also implicitly grants the API access levels of any service account in that namespace.

Users who can run privileged Pods can use that access to gain node access and potentially to further elevate their privileges. Where you do not fully trust a user or other principal with the ability to create suitably secure and isolated Pods, you should enforce either the **Baseline** or **Restricted** Pod Security Standard. You can use Pod Security admission or other (third party) mechanisms to implement that enforcement.

For these reasons, namespaces should be used to separate resources requiring different levels of trust or tenancy. It is still considered best practice to follow least privilege principles and assign the minimum set of permissions, but boundaries within a namespace should be considered weak.

Persistent volume creation

If someone - or some application - is allowed to create arbitrary PersistentVolumes, that access includes the creation of `hostPath` volumes, which then means that a Pod would get access to the underlying host filesystem(s) on the associated node. Granting that ability is a security risk.

There are many ways a container with unrestricted access to the host filesystem can escalate privileges, including reading data from other containers, and abusing the credentials of system services, such as Kubelet.

You should only allow access to create PersistentVolume objects for:

- Users (cluster operators) that need this access for their work, and who you trust.
- The Kubernetes control plane components which creates PersistentVolumes based on PersistentVolumeClaims that are configured for automatic provisioning. This is usually setup by the Kubernetes provider or by the operator when installing a CSI driver.

Where access to persistent storage is required trusted administrators should create PersistentVolumes, and constrained users should use PersistentVolumeClaims to access that storage.

Access to `proxy` subresource of Nodes

Users with access to the proxy sub-resource of node objects have rights to the Kubelet API, which allows for command execution on every pod on the node(s) to which they have rights. This access bypasses audit logging and admission control, so care should be taken before granting rights to this resource.

Escalate verb

Generally, the RBAC system prevents users from creating clusterroles with more rights than the user possesses. The exception to this is the `escalate` verb. As noted in the RBAC documentation, users with this right can effectively escalate their privileges.

Bind verb

Similar to the `escalate` verb, granting users this right allows for the bypass of Kubernetes in-built protections against privilege escalation, allowing users to create bindings to roles with rights they do not already have.

Impersonate verb

This verb allows users to impersonate and gain the rights of other users in the cluster. Care should be taken when granting it, to ensure that excessive permissions cannot be gained via one of the impersonated accounts.

CSRs and certificate issuing

The CSR API allows for users with `create` rights to CSRs and `update` rights on `certificatesigningrequests/approval` where the signer is `kubernetes.io/kube-apiserver-client` to create new client certificates which allow users to authenticate to the cluster. Those client certificates can have arbitrary names including duplicates of Kubernetes system components. This will effectively allow for privilege escalation.

Token request

Users with `create` rights on `serviceaccounts/token` can create TokenRequests to issue tokens for existing service accounts.

Control admission webhooks

Users with control over `validatingwebhookconfigurations` or `mutatingwebhookconfigurations` can control webhooks that can read any object admitted to the cluster, and in the case of mutating webhooks, also mutate admitted objects.

Namespace modification

Users who can perform **patch** operations on Namespace objects (through a namespaced RoleBinding to a Role with that access) can modify labels on that namespace. In clusters where Pod Security Admission is used, this may allow a user to configure the namespace for a more permissive policy than intended by the administrators. For clusters where NetworkPolicy is used, users may be set labels that indirectly allow access to services that an administrator did not intend to allow.

Kubernetes RBAC - denial of service risks

Object creation denial-of-service

Users who have rights to create objects in a cluster may be able to create sufficient large objects to create a denial of service condition either based on the size or number of objects, as discussed in etcd used by Kubernetes is vulnerable to OOM attack. This may be specifically relevant in multi-tenant clusters if semi-trusted or untrusted users are allowed limited access to a system.

One option for mitigation of this issue would be to use resource quotas to limit the quantity of objects which can be created.

What's next

- To learn more about RBAC, see the RBAC documentation.

Good practices for Kubernetes Secrets

Principles and practices for good Secret management for cluster administrators and application developers.

In Kubernetes, a Secret is an object that stores sensitive information, such as passwords, OAuth tokens, and SSH keys.

Secrets give you more control over how sensitive information is used and reduces the risk of accidental exposure. Secret values are encoded as base64 strings and are stored unencrypted by default, but can be configured to be encrypted at rest.

A Pod can reference the Secret in a variety of ways, such as in a volume mount or as an environment variable. Secrets are designed for confidential data and ConfigMaps are designed for non-confidential data.

The following good practices are intended for both cluster administrators and application developers. Use these guidelines to improve the security of your sensitive information in Secret objects, as well as to more effectively manage your Secrets.

Cluster administrators

This section provides good practices that cluster administrators can use to improve the security of confidential information in the cluster.

Configure encryption at rest

By default, Secret objects are stored unencrypted in etcd. You should configure encryption of your Secret data in `etcd`. For instructions, refer to Encrypt Secret Data at Rest.

Configure least-privilege access to Secrets

When planning your access control mechanism, such as Kubernetes Role-based Access Control (RBAC), consider the following guidelines for access to `Secret` objects. You should also follow the other guidelines in RBAC good practices.

- **Components**: Restrict `watch` or `list` access to only the most privileged, system-level components. Only grant `get` access for Secrets if the component's normal behavior requires it.
- **Humans**: Restrict `get`, `watch`, or `list` access to Secrets. Only allow cluster administrators to access `etcd`. This includes read-only access. For more complex access control, such as restricting access to Secrets with specific annotations, consider using third-party authorization mechanisms.

Caution:

Granting `list` access to Secrets implicitly lets the subject fetch the contents of the Secrets.

A user who can create a Pod that uses a Secret can also see the value of that Secret. Even if cluster policies do not allow a user to read the Secret directly, the same user could have access to run a Pod that then exposes the Secret. You can detect or limit the impact caused by Secret data being exposed, either intentionally or unintentionally, by a user with this access. Some recommendations include:

- Use short-lived Secrets
- Implement audit rules that alert on specific events, such as concurrent reading of multiple Secrets by a single user

Restrict Access for Secrets

Use separate namespaces to isolate access to mounted secrets.

Improve etcd management policies

Consider wiping or shredding the durable storage used by `etcd` once it is no longer in use.

If there are multiple `etcd` instances, configure encrypted SSL/TLS communication between the instances to protect the Secret data in transit.

Configure access to external Secrets

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.

You can use third-party Secrets store providers to keep your confidential data outside your cluster and then configure Pods to access that information. The Kubernetes Secrets Store CSI Driver is a DaemonSet that lets the kubelet retrieve Secrets from external stores, and mount the Secrets as a volume into specific Pods that you authorize to access the data.

For a list of supported providers, refer to Providers for the Secret Store CSI Driver.

Good practices for using swap memory

For best practices for setting swap memory for Linux nodes, please refer to swap memory management.

Developers

This section provides good practices for developers to use to improve the security of confidential data when building and deploying Kubernetes resources.

Restrict Secret access to specific containers

If you are defining multiple containers in a Pod, and only one of those containers needs access to a Secret, define the volume mount or environment variable configuration so that the other containers do not have access to that Secret.

Protect Secret data after reading

Applications still need to protect the value of confidential information after reading it from an environment variable or volume. For example, your application must avoid logging the secret data in the clear or transmitting it to an untrusted party.

Avoid sharing Secret manifests

If you configure a Secret through a manifest, with the secret data encoded as base64, sharing this file or checking it in to a source repository means the secret is available to everyone who can read the manifest.

Caution:

Base64 encoding is *not* an encryption method, it provides no additional confidentiality over plain text.

Ingress Controllers

In order for an Ingress to work in your cluster, there must be an *ingress controller* running. You need to select at least one ingress controller and make sure it is set up in your cluster. This page lists common ingress controllers that you can deploy.

In order for the Ingress resource to work, the cluster must have an ingress controller running.

Unlike other types of controllers which run as part of the `kube-controller-manager` binary, Ingress controllers are not started automatically with a cluster. Use this page to choose the ingress controller implementation that best fits your cluster.

Kubernetes as a project supports and maintains AWS, GCE, and nginx ingress controllers.

Additional controllers

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.

- AKS Application Gateway Ingress Controller is an ingress controller that configures the Azure Application Gateway.
- Alibaba Cloud MSE Ingress is an ingress controller that configures the Alibaba Cloud Native Gateway, which is also the commercial version of Higress.
- Apache APISIX ingress controller is an Apache APISIX-based ingress controller.
- Avi Kubernetes Operator provides L4-L7 load-balancing using VMware NSX Advanced Load Balancer.
- BFE Ingress Controller is a BFE-based ingress controller.
- Cilium Ingress Controller is an ingress controller powered by Cilium.
- The Citrix ingress controller works with Citrix Application Delivery Controller.
- Contour is an Envoy based ingress controller.
- Emissary-Ingress API Gateway is an Envoy-based ingress controller.
- EnRoute is an Envoy based API gateway that can run as an ingress controller.
- Easegress IngressController is an Easegress based API gateway that can run as an ingress controller.
- F5 BIG-IP Container Ingress Services for Kubernetes lets you use an Ingress to configure F5 BIG-IP virtual servers.
- FortiADC Ingress Controller support the Kubernetes Ingress resources and allows you to manage FortiADC objects from Kubernetes

- Gloo is an open-source ingress controller based on Envoy, which offers API gateway functionality.
- HAProxy Ingress is an ingress controller for HAProxy.
- Higress is an Envoy based API gateway that can run as an ingress controller.
- The HAProxy Ingress Controller for Kubernetes is also an ingress controller for HAProxy.
- Istio Ingress is an Istio based ingress controller.
- The Kong Ingress Controller for Kubernetes is an ingress controller driving Kong Gateway.
- Kusk Gateway is an OpenAPI-driven ingress controller based on Envoy.
- The NGINX Ingress Controller for Kubernetes works with the NGINX webserver (as a proxy).
- The ngrok Kubernetes Ingress Controller is an open source controller for adding secure public access to your K8s services using the ngrok platform.
- The OCI Native Ingress Controller is an Ingress controller for Oracle Cloud Infrastructure which allows you to manage the OCI Load Balancer.
- OpenNJet Ingress Controller is a OpenNJet-based ingress controller.
- The Pomerium Ingress Controller is based on Pomerium, which offers context-aware access policy.
- Skipper HTTP router and reverse proxy for service composition, including use cases like Kubernetes Ingress, designed as a library to build your custom proxy.
- The Traefik Kubernetes Ingress provider is an ingress controller for the Traefik proxy.
- Tyk Operator extends Ingress with Custom Resources to bring API Management capabilities to Ingress. Tyk Operator works with the Open Source Tyk Gateway & Tyk Cloud control plane.
- Voyager is an ingress controller for HAProxy.
- Wallarm Ingress Controller is an Ingress Controller that provides WAAP (WAF) and API Security capabilities.

Using multiple Ingress controllers

You may deploy any number of ingress controllers using ingress class within a cluster. Note the `.metadata.name` of your ingress class resource. When you create an ingress you would need that name to specify the `ingressClassName` field on your Ingress object (refer to IngressSpec v1 reference). `ingressClassName` is a replacement of the older annotation method.

If you do not specify an IngressClass for an Ingress, and your cluster has exactly one IngressClass marked as default, then Kubernetes applies the cluster's default IngressClass to the Ingress. You mark an IngressClass as default by setting the `ingressclass.kubernetes.io/is-default-class` annotation on that IngressClass, with the string value `"true"`.

Ideally, all ingress controllers should fulfill this specification, but the various ingress controllers operate slightly differently.

Note:

Make sure you review your ingress controller's documentation to understand the caveats of choosing it.

What's next

- Learn more about Ingress.

Service

Expose an application running in your cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.

In Kubernetes, a Service is a method for exposing a network application that is running as one or more Pods in your cluster.

A key aim of Services in Kubernetes is that you don't need to modify your existing application to use an unfamiliar service discovery mechanism. You can run code in Pods, whether this is a code designed for a cloud-native world, or an older app you've containerized. You use a Service to make that set of Pods available on the network so that clients can interact with it.

If you use a Deployment to run your app, that Deployment can create and destroy Pods dynamically. From one moment to the next, you don't know how many of those Pods are working and healthy; you might not even know what those healthy Pods are named. Kubernetes Pods are created and destroyed to match the desired state of your cluster. Pods are ephemeral resources (you should not expect that an individual Pod is reliable and durable).

Each Pod gets its own IP address (Kubernetes expects network plugins to ensure this). For a given Deployment in your cluster, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.

This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

Enter *Services*.

Services in Kubernetes

The Service API, part of Kubernetes, is an abstraction to help you expose groups of Pods over a network. Each Service object defines a logical set of endpoints (usually these endpoints are Pods) along with a policy about how to make those pods accessible.

For example, consider a stateless image-processing backend which is running with 3 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.

The Service abstraction enables this decoupling.

The set of Pods targeted by a Service is usually determined by a selector that you define. To learn about other ways to define Service endpoints, see Services *without* selectors.

If your workload speaks HTTP, you might choose to use an Ingress to control how web traffic reaches that workload. Ingress is not a Service type, but it acts as the entry point for your cluster. An Ingress lets you consolidate your routing rules into a single resource, so that you can expose multiple components of your workload, running separately in your cluster, behind a single listener.

The Gateway API for Kubernetes provides extra capabilities beyond Ingress and Service. You can add Gateway to your cluster - it is a family of extension APIs, implemented using CustomResourceDefinitions - and then use these to configure access to network services that are running in your cluster.

Cloud-native service discovery

If you're able to use Kubernetes APIs for service discovery in your application, you can query the API server for matching EndpointSlices. Kubernetes updates the EndpointSlices for a Service whenever the set of Pods in a Service changes.

For non-native applications, Kubernetes offers ways to place a network port or load balancer in between your application and the backend Pods.

Either way, your workload can use these service discovery mechanisms to find the target it wants to connect to.

Defining a Service

A Service is an object (the same way that a Pod or a ConfigMap is an object). You can create, view or modify Service definitions using the Kubernetes API. Usually you use a tool such as `kubectl` to make those API calls for you.

For example, suppose you have a set of Pods that each listen on TCP port 9376 and are labelled as `app.kubernetes.io/name=MyApp`. You can define a Service to publish that TCP listener:

`service/simple-service.yaml` Copy service/simple-service.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: my-servicespec:  selector:    app.kubernetes.io/name: MyApp  ports:    - protocol: TCP      port: 80
```

Applying this manifest creates a new Service named "my-service" with the default ClusterIP service type. The Service targets TCP port 9376 on any Pod with the `app.kubernetes.io/name: MyApp` label.

Kubernetes assigns this Service an IP address (the *cluster IP*), that is used by the virtual IP address mechanism. For more details on that mechanism, read Virtual IPs and Service Proxies.

The controller for that Service continuously scans for Pods that match its selector, and then makes any necessary updates to the set of EndpointSlices for the Service.

The name of a Service object must be a valid RFC 1035 label name.

Note:

A Service can map *any* incoming `port` to a `targetPort`. By default and for convenience, the `targetPort` is set to the same value as the `port` field.

Relaxed naming requirements for Service objects

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

The `RelaxedServiceNameValidation` feature gate allows Service object names to start with a digit. When this feature gate is enabled, Service object names must be valid RFC 1123 label names.

Port definitions

Port definitions in Pods have names, and you can reference these names in the `targetPort` attribute of a Service. For example, we can bind the `targetPort` of the Service to the Pod port in the following way:

```
apiVersion: v1
kind: Podmetadata:  name: nginx  labels:    app.kubernetes.io/name: proxyspec:  containers:  - name: nginx    image: nginx:stable
```

This works even if there is a mixture of Pods in the Service using a single configured name, with the same network protocol available via different port numbers. This offers a lot of flexibility for deploying and evolving your Services. For example, you can change the port numbers that Pods expose in the next version of your backend software, without breaking clients.

The default protocol for Services is TCP; you can also use any other supported protocol.

Because many Services need to expose more than one port, Kubernetes supports multiple port definitions for a single Service. Each port definition can have the same `protocol`, or a different one.

Services without selectors

Services most commonly abstract access to Kubernetes Pods thanks to the selector, but when used with a corresponding set of EndpointSlices objects and without a selector, the Service can abstract other kinds of backends, including ones that run outside the cluster.

For example:

- You want to have an external database cluster in production, but in your test environment you use your own databases.
- You want to point your Service to a Service in a different Namespace or on another cluster.
- You are migrating a workload to Kubernetes. While evaluating the approach, you run only a portion of your backends in Kubernetes.

In any of these scenarios you can define a Service *without* specifying a selector to match Pods. For example:

```
apiVersion: v1
kind: Servicemetadata:  name: my-servicespec:  ports:    - name: http      protocol: TCP      port: 80      targetPort: 9376
```

Because this Service has no selector, the corresponding EndpointSlice objects are not created automatically. You can map the Service to the network address and port where it's running, by adding an EndpointSlice object manually. For example:

```
apiVersion: discovery.k8s.io/v1
kind: EndpointSlicemetadata:  name: my-service-1 # by convention, use the name of the Service                   # as a prefix fo
```

Custom EndpointSlices

When you create an EndpointSlice object for a Service, you can use any name for the EndpointSlice. Each EndpointSlice in a namespace must have a unique name. You link an EndpointSlice to a Service by setting the `kubernetes.io/service-name` label on that EndpointSlice.

Note:

The endpoint IPs *must not* be: loopback (127.0.0.0/8 for IPv4, ::1/128 for IPv6), or link-local (169.254.0.0/16 and 224.0.0.0/24 for IPv4, fe80::/64 for IPv6).

The endpoint IP addresses cannot be the cluster IPs of other Kubernetes Services, because kube-proxy doesn't support virtual IPs as a destination.

For an EndpointSlice that you create yourself, or in your own code, you should also pick a value to use for the label `endpointslice.kubernetes.io/managed-by`. If you create your own controller code to manage EndpointSlices, consider using a value similar to `"my-domain.example/name-of-controller"`. If you are using a third party tool, use the name of the tool in all-lowercase and change spaces and other punctuation to dashes (`-`). If people are directly using a tool such as `kubectl` to manage EndpointSlices, use a name that describes this manual management, such as `"staff"` or `"cluster-admins"`. You should avoid using the reserved value `"controller"`, which identifies EndpointSlices managed by Kubernetes' own control plane.

Accessing a Service without a selector

Accessing a Service without a selector works the same as if it had a selector. In the example for a Service without a selector, traffic is routed to one of the two endpoints defined in the EndpointSlice manifest: a TCP connection to 10.1.2.3 or 10.4.5.6, on port 9376.

Note:

The Kubernetes API server does not allow proxying to endpoints that are not mapped to pods. Actions such as `kubectl port-forward service/<service-name> forwardedPort:servicePort` where the service has no selector will fail due to this constraint. This prevents the Kubernetes API server from being used as a proxy to endpoints the caller may not be authorized to access.

An `ExternalName` Service is a special case of Service that does not have selectors and uses DNS names instead. For more information, see the ExternalName section.

EndpointSlices

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

EndpointSlices are objects that represent a subset (a *slice*) of the backing network endpoints for a Service.

Your Kubernetes cluster tracks how many endpoints each EndpointSlice represents. If there are so many endpoints for a Service that a threshold is reached, then Kubernetes adds another empty EndpointSlice and stores new endpoint information there. By default, Kubernetes makes a new EndpointSlice once the existing EndpointSlices all contain at least 100 endpoints. Kubernetes does not make the new EndpointSlice until an extra endpoint needs to be added.

See EndpointSlices for more information about this API.

Endpoints (deprecated)

FEATURE STATE: `Kubernetes v1.33 [deprecated]`

The EndpointSlice API is the evolution of the older Endpoints API. The deprecated Endpoints API has several problems relative to EndpointSlice:

- It does not support dual-stack clusters.
- It does not contain information needed to support newer features, such as trafficDistribution.
- It will truncate the list of endpoints if it is too long to fit in a single object.

Because of this, it is recommended that all clients use the EndpointSlice API rather than Endpoints.

Over-capacity endpoints

Kubernetes limits the number of endpoints that can fit in a single Endpoints object. When there are over 1000 backing endpoints for a Service, Kubernetes truncates the data in the Endpoints object. Because a Service can be linked with more than one EndpointSlice, the 1000 backing endpoint limit only affects the legacy Endpoints API.

In that case, Kubernetes selects at most 1000 possible backend endpoints to store into the Endpoints object, and sets an annotation on the Endpoints: `endpoints.kubernetes.io/over-capacity: truncated`. The control plane also removes that annotation if the number of backend Pods drops below 1000.

Traffic is still sent to backends, but any load balancing mechanism that relies on the legacy Endpoints API only sends traffic to at most 1000 of the available backing endpoints.

The same API limit means that you cannot manually update an Endpoints to have more than 1000 endpoints.

Application protocol

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

The `appProtocol` field provides a way to specify an application protocol for each Service port. This is used as a hint for implementations to offer richer behavior for protocols that they understand. The value of this field is mirrored by the corresponding Endpoints and EndpointSlice objects.

This field follows standard Kubernetes label syntax. Valid values are one of:

- [IANA standard service names](#).
- Implementation-defined prefixed names such as `mycompany.com/my-custom-protocol`.
- Kubernetes-defined prefixed names:

Protocol	Description
`kubernetes.io/h2c`	HTTP/2 over cleartext as described in RFC 7540
`kubernetes.io/ws`	WebSocket over cleartext as described in RFC 6455
`kubernetes.io/wss`	WebSocket over TLS as described in RFC 6455

Multi-port Services

For some Services, you need to expose more than one port. Kubernetes lets you configure multiple port definitions on a Service object. When using multiple ports for a Service, you must give all of your ports names so that these are unambiguous. For example:

```
apiVersion: v1
kind: Servicemetadata:  name: my-servicespec:  selector:    app.kubernetes.io/name: MyApp  ports:    - name: http      protocol: T
```

Note:

As with Kubernetes names in general, names for ports must only contain lowercase alphanumeric characters and `-`. Port names must also start and end with an alphanumeric character.

For example, the names `123-abc` and `web` are valid, but `123_abc` and `-web` are not.

Service type

For some parts of your application (for example, frontends) you may want to expose a Service onto an external IP address, one that's accessible from outside of your cluster.

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

The available `type` values and their behaviors are:

`ClusterIP`
: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default that is used if you don't explicitly specify a `type` for a Service. You can expose the Service to the public internet using an Ingress or a Gateway.

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

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

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

The `type` field in the Service API is designed as nested functionality - each level adds to the previous. However there is an exception to this nested design. You can define a `LoadBalancer` Service by disabling the load balancer `NodePort` allocation.

`type: ClusterIP`

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

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

If you define a Service that has the `.spec.clusterIP` set to `"None"` then Kubernetes does not assign an IP address. See headless Services for more information.

Choosing your own IP address

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

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

Read avoiding collisions to learn how Kubernetes helps reduce the risk and impact of two different Services both trying to use the same IP address.

`type: NodePort`

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

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

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

Choosing your own port

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

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

```
apiVersion: v1
kind: Servicemetadata:  name: my-servicespec:  type: NodePort  selector:    app.kubernetes.io/name: MyApp  ports:    - port: 80
```

Reserve Nodeport ranges to avoid collisions

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

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

Custom IP address configuration for `type: NodePort` Services

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

If you want to specify particular IP address(es) to proxy the port, you can set the `--nodeport-addresses` flag for kube-proxy or the equivalent `nodePortAddresses` field of the kube-proxy configuration file to particular IP block(s).

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

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

Note:

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

`type: LoadBalancer`

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

```
apiVersion: v1
kind: Servicemetadata:  name: my-servicespec:  selector:    app.kubernetes.io/name: MyApp  ports:    - protocol: TCP      port: 80
```

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

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

You can configure a load balanced Service to omit assigning a node port, provided that the cloud provider implementation supports this.

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

Note:

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

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

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

If you are writing code for a load balancer integration with Kubernetes, avoid using this field. You can integrate with Gateway rather than Service, or you can define your own (provider specific) annotations on the Service that specify the equivalent detail.

Node liveness impact on load balancer traffic

Load balancer health checks are critical to modern applications. They are used to determine which server (virtual machine, or IP address) the load balancer should dispatch traffic to. The Kubernetes APIs do not define how health checks have to be implemented for Kubernetes managed load balancers, instead it's the cloud providers (and the people implementing integration code) who decide on the behavior. Load balancer health checks are extensively used within the context of supporting the `externalTrafficPolicy` field for Services.

Load balancers with mixed protocol types

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

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

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

Note:

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

Disabling load balancer NodePort allocation

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

You can optionally disable node port allocation for a Service of `type: LoadBalancer`, by setting the field `spec.allocateLoadBalancerNodePorts` to `false`. This should only be used for load balancer implementations that route traffic directly to pods as opposed to using node ports. By default, `spec.allocateLoadBalancerNodePorts` is `true` and type LoadBalancer Services will continue to allocate node ports. If `spec.allocateLoadBalancerNodePorts` is set to `false` on an existing Service with allocated node ports, those node ports will **not** be de-allocated automatically. You must explicitly remove the `nodePorts` entry in every Service port to de-allocate those node ports.

Specifying class of load balancer implementation

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

For a Service with `type` set to `LoadBalancer`, the `.spec.loadBalancerClass` field enables you to use a load balancer implementation other than the cloud provider default.

By default, `.spec.loadBalancerClass` is not set and a `LoadBalancer` type of Service uses the cloud provider's default load balancer implementation if the cluster is configured with a cloud provider using the `--cloud-provider` component flag.

If you specify `.spec.loadBalancerClass`, it is assumed that a load balancer implementation that matches the specified class is watching for Services. Any default load balancer implementation (for example, the one provided by the cloud provider) will ignore Services that have this field set. `spec.loadBalancerClass` can be set on a Service of type `LoadBalancer` only. Once set, it cannot be changed. The value of `spec.loadBalancerClass` must be a label-style identifier, with an optional prefix such as "`internal-vip`" or "`example.com/internal-vip`". Unprefixed names are reserved for end-users.

Load balancer IP address mode

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

For a Service of `type: LoadBalancer`, a controller can set `.status.loadBalancer.ingress.ipMode`. The `.status.loadBalancer.ingress.ipMode` specifies how the load-balancer IP behaves. It may be specified only when the `.status.loadBalancer.ingress.ip` field is also specified.

There are two possible values for `.status.loadBalancer.ingress.ipMode`: "VIP" and "Proxy". The default value is "VIP" meaning that traffic is delivered to the node with the destination set to the load-balancer's IP and port. There are two cases when setting this to "Proxy", depending on how the load-balancer from the cloud provider delivers the traffics:

- If the traffic is delivered to the node then DNATed to the pod, the destination would be set to the node's IP and node port;
- If the traffic is delivered directly to the pod, the destination would be set to the pod's IP and port.

Service implementations may use this information to adjust traffic routing.

Internal load balancer

In a mixed environment it is sometimes necessary to route traffic from Services inside the same (virtual) network address block.

In a split-horizon DNS environment you would need two Services to be able to route both external and internal traffic to your endpoints.

To set an internal load balancer, add one of the following annotations to your Service depending on the cloud service provider you're using:

- Default
- GCP
- AWS
- Azure
- IBM Cloud
- OpenStack

- Baidu Cloud
- Tencent Cloud
- Alibaba Cloud
- OCI

Select one of the tabs.

```
metadata:
  name: my-service
  annotations:
    networking.gke.io/load-balancer-type: "Internal"
```

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-scheme: "internal"
```

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
```

```
metadata:
  name: my-service
  annotations:
    service.kubernetes.io/ibm-load-balancer-cloud-provider-ip-type: "private"
```

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/openstack-internal-load-balancer: "true"
```

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/cce-load-balancer-internal-vpc: "true"
```

```
metadata:
  annotations:
    service.kubernetes.io/qcloud-loadbalancer-internal-subnetid: subnet-xxxxx
```

```
metadata:
  annotations:
    service.beta.kubernetes.io/alibaba-cloud-loadbalancer-address-type: "intranet"
```

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/oci-load-balancer-internal: true
```

`type: ExternalName`

Services of type ExternalName map a Service to a DNS name, not to a typical selector such as `my-service` or `cassandra`. You specify these Services with the `spec.externalName` parameter.

This Service definition, for example, maps the `my-service` Service in the `prod` namespace to `my.database.example.com`:

```
apiVersion: v1
kind: Servicemetadata:  name: my-service  namespace: prodspec:  type: ExternalName  externalName: my.database.example.com
```

Note:

A Service of `type: ExternalName` accepts an IPv4 address string, but treats that string as a DNS name comprised of digits, not as an IP address (the internet does not however allow such names in DNS). Services with external names that resemble IPv4 addresses are not resolved by DNS servers.

If you want to map a Service directly to a specific IP address, consider using headless Services.

When looking up the host `my-service.prod.svc.cluster.local`, the cluster DNS Service returns a `CNAME` record with the value `my.database.example.com`. Accessing `my-service` works in the same way as other Services but with the crucial difference that redirection happens at the DNS level rather than via proxying or forwarding. Should you later decide to move your database into your cluster, you can start its Pods, add appropriate selectors or endpoints, and change the Service's `type`.

Caution:

You may have trouble using ExternalName for some common protocols, including HTTP and HTTPS. If you use ExternalName then the hostname used by clients inside your cluster is different from the name that the ExternalName references.

For protocols that use hostnames this difference may lead to errors or unexpected responses. HTTP requests will have a `Host:` header that the origin server does not recognize; TLS servers will not be able to provide a certificate matching the hostname that the client connected to.

Headless Services

Sometimes you don't need load-balancing and a single Service IP. In this case, you can create what are termed *headless Services*, by explicitly specifying `"None"` for the cluster IP address (`.spec.clusterIP`).

You can use a headless Service to interface with other service discovery mechanisms, without being tied to Kubernetes' implementation.

For headless Services, a cluster IP is not allocated, kube-proxy does not handle these Services, and there is no load balancing or proxying done by the platform for them.

A headless Service allows a client to connect to whichever Pod it prefers, directly. Services that are headless don't configure routes and packet forwarding using virtual IP addresses and proxies; instead, headless Services report the endpoint IP addresses of the individual pods via internal DNS records, served through the cluster's DNS service. To define a headless Service, you make a Service with `.spec.type` set to ClusterIP (which is also the default for `type`), and you additionally set `.spec.clusterIP` to None.

The string value None is a special case and is not the same as leaving the `.spec.clusterIP` field unset.

How DNS is automatically configured depends on whether the Service has selectors defined:

With selectors

For headless Services that define selectors, the endpoints controller creates EndpointSlices in the Kubernetes API, and modifies the DNS configuration to return A or AAAA records (IPv4 or IPv6 addresses) that point directly to the Pods backing the Service.

Without selectors

For headless Services that do not define selectors, the control plane does not create EndpointSlice objects. However, the DNS system looks for and configures either:

- DNS CNAME records for `type: ExternalName` Services.
- DNS A / AAAA records for all IP addresses of the Service's ready endpoints, for all Service types other than `ExternalName`.
  - For IPv4 endpoints, the DNS system creates A records.
  - For IPv6 endpoints, the DNS system creates AAAA records.

When you define a headless Service without a selector, the `port` must match the `targetPort`.

Discovering services

For clients running inside your cluster, Kubernetes supports two primary modes of finding a Service: environment variables and DNS.

Environment variables

When a Pod is run on a Node, the kubelet adds a set of environment variables for each active Service. It adds `{SVCNAME}_SERVICE_HOST` and `{SVCNAME}_SERVICE_PORT` variables, where the Service name is upper-cased and dashes are converted to underscores.

For example, the Service `redis-primary` which exposes TCP port 6379 and has been allocated cluster IP address 10.0.0.11, produces the following environment variables:

```
REDIS_PRIMARY_SERVICE_HOST=10.0.0.11
REDIS_PRIMARY_SERVICE_PORT=6379
REDIS_PRIMARY_PORT=tcp://10.0.0.11:6379
REDIS_PRIMARY_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_PRIMARY_PORT_6379_TCP_PROTO=tcp
REDIS_PRIMARY_PORT_6379_TCP_PORT=6379
REDIS_PRIMARY_PORT_6379_TCP_ADDR=10.0.0.11
```

Note:

When you have a Pod that needs to access a Service, and you are using the environment variable method to publish the port and cluster IP to the client Pods, you must create the Service *before* the client Pods come into existence. Otherwise, those client Pods won't have their environment variables populated.

If you only use DNS to discover the cluster IP for a Service, you don't need to worry about this ordering issue.

Kubernetes also supports and provides variables that are compatible with Docker Engine's "*legacy container links*" feature. You can read `makeLinkVariables` to see how this is implemented in Kubernetes.

DNS

You can (and almost always should) set up a DNS service for your Kubernetes cluster using an add-on.

A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one. If DNS has been enabled throughout your cluster then all Pods should automatically be able to resolve Services by their DNS name.

For example, if you have a Service called `my-service` in a Kubernetes namespace `my-ns`, the control plane and the DNS Service acting together create a DNS record for `my-service.my-ns`. Pods in the `my-ns` namespace should be able to find the service by doing a name lookup for `my-service` (`my-service.my-ns` would also work).

Pods in other namespaces must qualify the name as `my-service.my-ns`. These names will resolve to the cluster IP assigned for the Service.

Kubernetes also supports DNS SRV (Service) records for named ports. If the `my-service.my-ns` Service has a port named `http` with the protocol set to `TCP`, you can do a DNS SRV query for `_http._tcp.my-service.my-ns` to discover the port number for `http`, as well as the IP address.

The Kubernetes DNS server is the only way to access `ExternalName` Services. You can find more information about `ExternalName` resolution in DNS for Services and Pods.

Virtual IP addressing mechanism

Read Virtual IPs and Service Proxies explains the mechanism Kubernetes provides to expose a Service with a virtual IP address.

Traffic policies

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

See Traffic Policies for more details.

Traffic distribution

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

The `.spec.trafficDistribution` field provides another way to influence traffic routing within a Kubernetes Service. While traffic policies focus on strict semantic guarantees, traffic distribution allows you to express *preferences* (such as routing to topologically closer endpoints). This can help optimize for performance, cost, or reliability. In Kubernetes 1.34, the following field value is supported:

`PreferClose`
: Indicates a preference for routing traffic to endpoints that are in the same zone as the 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 is an alias for `PreferClose` that is clearer about the intended semantics.

`PreferSameNode`
: Indicates a preference for routing traffic to endpoints that are on the same node as the client.

If the field is not set, the implementation will apply its default routing strategy.

See Traffic Distribution for more details

Session stickiness

If you want to make sure that connections from a particular client are passed to the same Pod each time, you can configure session affinity based on the client's IP address. Read session affinity to learn more.

External IPs

If there are external IPs that route to one or more cluster nodes, Kubernetes Services can be exposed on those `externalIPs`. When network traffic arrives into the cluster, with the external IP (as destination IP) and the port matching that Service, rules and routes that Kubernetes has configured ensure that the traffic is routed to one of the endpoints for that Service.

When you define a Service, you can specify `externalIPs` for any service type. In the example below, the Service named `"my-service"` can be accessed by clients using TCP, on `"198.51.100.32:80"` (calculated from `.spec.externalIPs[]` and `.spec.ports[].port`).

```
apiVersion: v1
kind: Servicemetadata:  name: my-servicespec:  selector:    app.kubernetes.io/name: MyApp  ports:    - name: http      protocol: T
```

Note:

Kubernetes does not manage allocation of `externalIPs`; these are the responsibility of the cluster administrator.

API Object

Service is a top-level resource in the Kubernetes REST API. You can find more details about the Service API object.

What's next

Learn more about Services and how they fit into Kubernetes:

- Follow the Connecting Applications with Services tutorial.
- Read about Ingress, which exposes HTTP and HTTPS routes from outside the cluster to Services within your cluster.
- Read about Gateway, an extension to Kubernetes that provides more flexibility than Ingress.

For more context, read the following:

- Virtual IPs and Service Proxies
- EndpointSlices
- Service API reference
- EndpointSlice API reference
- Endpoint API reference (legacy)

Topology Aware Routing

Topology Aware Routing provides a mechanism to help keep network traffic within the zone where it originated. Preferring same-zone traffic between Pods in your cluster can help with reliability, performance (network latency and throughput), or cost.
FEATURE STATE: `Kubernetes v1.23 [beta]`

Note:

Prior to Kubernetes 1.27, this feature was known as *Topology Aware Hints*.

Topology Aware Routing adjusts routing behavior to prefer keeping traffic in the zone it originated from. In some cases this can help reduce costs or improve network performance.

Motivation

Kubernetes clusters are increasingly deployed in multi-zone environments. *Topology Aware Routing* provides a mechanism to help keep traffic within the zone it originated from. When calculating the endpoints for a Service, the EndpointSlice controller considers the topology (region and zone) of each endpoint and populates the hints field to allocate it to a zone. Cluster components such as kube-proxy can then consume those hints, and use them to influence how the traffic is routed (favoring topologically closer endpoints).

Enabling Topology Aware Routing

Note:

Prior to Kubernetes 1.27, this behavior was controlled using the `service.kubernetes.io/topology-aware-hints` annotation.

You can enable Topology Aware Routing for a Service by setting the `service.kubernetes.io/topology-mode` annotation to `Auto`. When there are enough endpoints available in each zone, Topology Hints will be populated on EndpointSlices to allocate individual endpoints to specific zones, resulting in traffic being routed closer to where it originated from.

When it works best

This feature works best when:

1. Incoming traffic is evenly distributed

If a large proportion of traffic is originating from a single zone, that traffic could overload the subset of endpoints that have been allocated to that zone. This feature is not recommended when incoming traffic is expected to originate from a single zone.

2. The Service has 3 or more endpoints per zone

In a three zone cluster, this means 9 or more endpoints. If there are fewer than 3 endpoints per zone, there is a high (≈50%) probability that the EndpointSlice controller will not be able to allocate endpoints evenly and instead will fall back to the default cluster-wide routing approach.

How It Works

The "Auto" heuristic attempts to proportionally allocate a number of endpoints to each zone. Note that this heuristic works best for Services that have a significant number of endpoints.

EndpointSlice controller

The EndpointSlice controller is responsible for setting hints on EndpointSlices when this heuristic is enabled. The controller allocates a proportional amount of endpoints to each zone. This proportion is based on the allocatable CPU cores for nodes running in that zone. For example, if one zone had 2 CPU cores and another zone only had 1 CPU core, the controller would allocate twice as many endpoints to the zone with 2 CPU cores.

The following example shows what an EndpointSlice looks like when hints have been populated:

```
apiVersion: discovery.k8s.io/v1
kind: EndpointSlicemetadata:  name: example-hints  labels:    kubernetes.io/service-name: example-svcaddressType: IPv4ports:  - na
```

kube-proxy

The kube-proxy component filters the endpoints it routes to based on the hints set by the EndpointSlice controller. In most cases, this means that the kube-proxy is able to route traffic to endpoints in the same zone. Sometimes the controller allocates endpoints from a different zone to ensure more even distribution of endpoints between zones. This would result in some traffic being routed to other zones.

Safeguards

The Kubernetes control plane and the kube-proxy on each node apply some safeguard rules before using Topology Aware Hints. If these don't check out, the kube-proxy selects endpoints from anywhere in your cluster, regardless of the zone.

1. **Insufficient number of endpoints:** If there are less endpoints than zones in a cluster, the controller will not assign any hints.
2. **Impossible to achieve balanced allocation:** In some cases, it will be impossible to achieve a balanced allocation of endpoints among zones. For example, if zone-a is twice as large as zone-b, but there are only 2 endpoints, an endpoint allocated to zone-a may receive twice as much traffic as zone-b. The controller does not assign hints if it can't get this "expected overload" value below an acceptable threshold for each zone. Importantly this is not based on real-time feedback. It is still possible for individual endpoints to become overloaded.

3. **One or more Nodes has insufficient information:** If any node does not have a `topology.kubernetes.io/zone` label or is not reporting a value for allocatable CPU, the control plane does not set any topology-aware endpoint hints and so kube-proxy does not filter endpoints by zone.
4. **One or more endpoints does not have a zone hint:** When this happens, the kube-proxy assumes that a transition from or to Topology Aware Hints is underway. Filtering endpoints for a Service in this state would be dangerous so the kube-proxy falls back to using all endpoints.
5. **A zone is not represented in hints:** If the kube-proxy is unable to find at least one endpoint with a hint targeting the zone it is running in, it falls back to using endpoints from all zones. This is most likely to happen as you add a new zone into your existing cluster.

Constraints

- Topology Aware Hints are not used when `internalTrafficPolicy` is set to `Local` on a Service. It is possible to use both features in the same cluster on different Services, just not on the same Service.
- This approach will not work well for Services that have a large proportion of traffic originating from a subset of zones. Instead this assumes that incoming traffic will be roughly proportional to the capacity of the Nodes in each zone.
- The EndpointSlice controller ignores unready nodes as it calculates the proportions of each zone. This could have unintended consequences if a large portion of nodes are unready.
- The EndpointSlice controller ignores nodes with the `node-role.kubernetes.io/control-plane` or `node-role.kubernetes.io/master` label set. This could be problematic if workloads are also running on those nodes.
- The EndpointSlice controller does not take into account tolerations when deploying or calculating the proportions of each zone. If the Pods backing a Service are limited to a subset of Nodes in the cluster, this will not be taken into account.
- This may not work well with autoscaling. For example, if a lot of traffic is originating from a single zone, only the endpoints allocated to that zone will be handling that traffic. That could result in Horizontal Pod Autoscaler either not picking up on this event, or newly added pods starting in a different zone.

Custom heuristics

Kubernetes is deployed in many different ways, there is no single heuristic for allocating endpoints to zones will work for every use case. A key goal of this feature is to enable custom heuristics to be developed if the built in heuristic does not work for your use case. The first steps to enable custom heuristics were included in the 1.27 release. This is a limited implementation that may not yet cover some relevant and plausible situations.

What's next

- Follow the Connecting Applications with Services tutorial
- Learn about the trafficDistribution field, which is closely related to the `service.kubernetes.io/topology-mode` annotation and provides flexible options for traffic routing within Kubernetes.

Ingress

Make your HTTP (or HTTPS) network service available using a protocol-aware configuration mechanism, that understands web concepts like URIs, hostnames, paths, and more. The Ingress concept lets you map traffic to different backends based on rules you define via the Kubernetes API.

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

An API object that manages external access to the services in a cluster, typically HTTP.

Ingress may provide load balancing, SSL termination and name-based virtual hosting.

Note:

Ingress is frozen. New features are being added to the Gateway API.

Terminology

For clarity, this guide defines the following terms:

- Node: A worker machine in Kubernetes, part of a cluster.
- Cluster: A set of Nodes that run containerized applications managed by Kubernetes. For this example, and in most common Kubernetes deployments, nodes in the cluster are not part of the public internet.
- Edge router: A router that enforces the firewall policy for your cluster. This could be a gateway managed by a cloud provider or a physical piece of hardware.
- Cluster network: A set of links, logical or physical, that facilitate communication within a cluster according to the Kubernetes networking model.
- Service: A Kubernetes Service that identifies a set of Pods using label selectors. Unless mentioned otherwise, Services are assumed to have virtual IPs only routable within the cluster network.

What is Ingress?

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

Here is a simple example where an Ingress sends all its traffic to one Service:

ingress-diagram

Figure. Ingress

An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. An Ingress controller is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer.

Prerequisites

You must have an Ingress controller to satisfy an Ingress. Only creating an Ingress resource has no effect.

You may need to deploy an Ingress controller such as ingress-nginx. You can choose from a number of Ingress controllers.

Ideally, all Ingress controllers should fit the reference specification. In reality, the various Ingress controllers operate slightly differently.

Note:

Make sure you review your Ingress controller's documentation to understand the caveats of choosing it.

The Ingress resource

A minimal Ingress resource example:

`service/networking/minimal-ingress.yaml` Copy service/networking/minimal-ingress.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: Ingressmetadata:  name: minimal-ingress  annotations:    nginx.ingress.kubernetes.io/rewrite-target: /spec:  ingressClassNam
```

An Ingress needs `apiVersion`, `kind`, `metadata` and `spec` fields. The name of an Ingress object must be a valid DNS subdomain name. For general information about working with config files, see deploying applications, configuring containers, managing resources. Ingress frequently uses annotations to configure some options depending on the Ingress controller, an example of which is the rewrite-target annotation. Different Ingress controllers support different annotations. Review the documentation for your choice of Ingress controller to learn which annotations are supported.

The Ingress spec has all the information needed to configure a load balancer or proxy server. Most importantly, it contains a list of rules matched against all incoming requests. Ingress resource only supports rules for directing HTTP(S) traffic.

If the `ingressClassName` is omitted, a default Ingress class should be defined.

There are some ingress controllers, that work without the definition of a default `IngressClass`. For example, the Ingress-NGINX controller can be configured with a flag `--watch-ingress-without-class`. It is recommended though, to specify the default `IngressClass` as shown below.

Ingress rules

Each HTTP rule contains the following information:

- An optional host. In this example, no host is specified, so the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided (for example, foo.bar.com), the rules apply to that host.
- A list of paths (for example, `/testpath`), each of which has an associated backend defined with a `service.name` and a `service.port.name` or `service.port.number`. Both the host and path must match the content of an incoming request before the load balancer directs traffic to the referenced Service.
- A backend is a combination of Service and port names as described in the Service doc or a custom resource backend by way of a CRD. HTTP (and HTTPS) requests to the Ingress that match the host and path of the rule are sent to the listed backend.

A `defaultBackend` is often configured in an Ingress controller to service any requests that do not match a path in the spec.

DefaultBackend

An Ingress with no rules sends all traffic to a single default backend and `.spec.defaultBackend` is the backend that should handle requests in that case. The `defaultBackend` is conventionally a configuration option of the Ingress controller and is not specified in your Ingress resources. If no `.spec.rules` are specified, `.spec.defaultBackend` must be specified. If `defaultBackend` is not set, the handling of requests that do not match any of the rules will be up to the ingress controller (consult the documentation for your ingress controller to find out how it handles this case).

If none of the hosts or paths match the HTTP request in the Ingress objects, the traffic is routed to your default backend.

Resource backends

A `Resource` backend is an ObjectRef to another Kubernetes resource within the same namespace as the Ingress object. A `Resource` is a mutually exclusive setting with Service, and will fail validation if both are specified. A common usage for a `Resource` backend is to ingress data to an object storage backend with static assets.

`service/networking/ingress-resource-backend.yaml` Copy service/networking/ingress-resource-backend.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: Ingressmetadata:  name: ingress-resource-backendspec:  defaultBackend:    resource:      apiGroup: k8s.example.com      kind
```

After creating the Ingress above, you can view it with the following command:

```
kubectl describe ingress ingress-resource-backend
```

```
Name:             ingress-resource-backend
Namespace:        default
Address:
Default backend:  APIGroup: k8s.example.com, Kind: StorageBucket, Name: static-assets
Rules:
  Host        Path  Backends
  ----        ----  --------
  *
              /icons   APIGroup: k8s.example.com, Kind: StorageBucket, Name: icon-assets
Annotations:  <none>
Events:       <none>
```

Path types

Each path in an Ingress is required to have a corresponding path type. Paths that do not include an explicit `pathType` will fail validation. There are three supported path types:

- `ImplementationSpecific`: With this path type, matching is up to the IngressClass. Implementations can treat this as a separate `pathType` or treat it identically to `Prefix` or `Exact` path types.

- `Exact`: Matches the URL path exactly and with case sensitivity.

- `Prefix`: Matches based on a URL path prefix split by `/`. Matching is case sensitive and done on a path element by element basis. A path element refers to the list of labels in the path split by the `/` separator. A request is a match for path *p* if every *p* is an element-wise prefix of *p* of the request path.

  Note:

  If the last element of the path is a substring of the last element in request path, it is not a match (for example: `/foo/bar` matches `/foo/bar/baz`, but does not match `/foo/barbaz`).

Examples

Kind	Path(s)	Request path(s)	Matches?
Prefix	`/`	(all paths)	Yes
Exact	`/foo`	`/foo`	Yes
Exact	`/foo`	`/bar`	No
Exact	`/foo`	`/foo/`	No
Exact	`/foo/`	`/foo`	No
Prefix	`/foo`	`/foo`, `/foo/`	Yes
Prefix	`/foo/`	`/foo`, `/foo/`	Yes
Prefix	`/aaa/bb`	`/aaa/bbb`	No
Prefix	`/aaa/bbb`	`/aaa/bbb`	Yes
Prefix	`/aaa/bbb/`	`/aaa/bbb`	Yes, ignores trailing slash
Prefix	`/aaa/bbb`	`/aaa/bbb/`	Yes, matches trailing slash
Prefix	`/aaa/bbb`	`/aaa/bbb/ccc`	Yes, matches subpath
Prefix	`/aaa/bbb`	`/aaa/bbbxyz`	No, does not match string prefix
Prefix	`/`, `/aaa`	`/aaa/ccc`	Yes, matches `/aaa` prefix
Prefix	`/`, `/aaa`, `/aaa/bbb`	`/aaa/bbb`	Yes, matches `/aaa/bbb` prefix
Prefix	`/`, `/aaa`, `/aaa/bbb`	`/ccc`	Yes, matches `/` prefix
Prefix	`/aaa`	`/ccc`	No, uses default backend
Mixed	`/foo` (Prefix), `/foo` (Exact)	`/foo`	Yes, prefers Exact

Multiple matches

In some cases, multiple paths within an Ingress will match a request. In those cases precedence will be given first to the longest matching path. If two paths are still equally matched, precedence will be given to paths with an exact path type over prefix path type.

Hostname wildcards

Hosts can be precise matches (for example “`foo.bar.com`”) or a wildcard (for example “`*.foo.com`”). Precise matches require that the HTTP `host` header matches the `host` field. Wildcard matches require the HTTP `host` header is equal to the suffix of the wildcard rule.

Host	Host header	Match?
`*.foo.com`	`bar.foo.com`	Matches based on shared suffix
`*.foo.com`	`baz.bar.foo.com`	No match, wildcard only covers a single DNS label
`*.foo.com`	`foo.com`	No match, wildcard only covers a single DNS label

service/networking/ingress-wildcard-host.yaml Copy service/networking/ingress-wildcard-host.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: Ingressmetadata:  name: ingress-wildcard-hostspec:  rules:  - host: "foo.bar.com"    http:      paths:      - pathType: Pref
```

Ingress class

Ingresses can be implemented by different controllers, often with different configuration. Each Ingress should specify a class, a reference to an IngressClass resource that contains additional configuration including the name of the controller that should implement the class.

service/networking/external-lb.yaml Copy service/networking/external-lb.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: IngressClassmetadata:  name: external-lbspec:  controller: example.com/ingress-controller  parameters:    apiGroup: k8s.exam
```

The `.spec.parameters` field of an IngressClass lets you reference another resource that provides configuration related to that IngressClass.

The specific type of parameters to use depends on the ingress controller that you specify in the `.spec.controller` field of the IngressClass.

IngressClass scope

Depending on your ingress controller, you may be able to use parameters that you set cluster-wide, or just for one namespace.

- Cluster
- Namespaced

The default scope for IngressClass parameters is cluster-wide.

If you set the `.spec.parameters` field and don't set `.spec.parameters.scope`, or if you set `.spec.parameters.scope` to `Cluster`, then the IngressClass refers to a cluster-scoped resource. The `kind` (in combination the `apiGroup`) of the parameters refers to a cluster-scoped API (possibly a custom resource), and the `name` of the parameters identifies a specific cluster scoped resource for that API.

For example:

```
---
apiVersion: networking.k8s.io/v1kind: IngressClassmetadata:  name: external-lb-1spec:  controller: example.com/ingress-controller
```

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

If you set the `.spec.parameters` field and set `.spec.parameters.scope` to `Namespace`, then the IngressClass refers to a namespaced-scoped resource. You must also set the `namespace` field within `.spec.parameters` to the namespace that contains the parameters you want to use.

The `kind` (in combination the `apiGroup`) of the parameters refers to a namespaced API (for example: ConfigMap), and the `name` of the parameters identifies a specific resource in the namespace you specified in `namespace`.

Namespace-scoped parameters help the cluster operator delegate control over the configuration (for example: load balancer settings, API gateway definition) that is used for a workload. If you used a cluster-scoped parameter then either:

- the cluster operator team needs to approve a different team's changes every time there's a new configuration change being applied.
- the cluster operator must define specific access controls, such as RBAC roles and bindings, that let the application team make changes to the cluster-scoped parameters resource.

The IngressClass API itself is always cluster-scoped.

Here is an example of an IngressClass that refers to parameters that are namespaced:

```
---
apiVersion: networking.k8s.io/v1kind: IngressClassmetadata:  name: external-lb-2spec:  controller: example.com/ingress-controller
```

Deprecated annotation

Before the IngressClass resource and `ingressClassName` field were added in Kubernetes 1.18, Ingress classes were specified with a `kubernetes.io/ingress.class` annotation on the Ingress. This annotation was never formally defined, but was widely supported by Ingress controllers.

The newer `ingressClassName` field on Ingresses is a replacement for that annotation, but is not a direct equivalent. While the annotation was generally used to reference the name of the Ingress controller that should implement the Ingress, the field is a reference to an IngressClass resource that contains additional Ingress configuration, including the name of the Ingress controller.

Default IngressClass

You can mark a particular IngressClass as default for your cluster. Setting the `ingressclass.kubernetes.io/is-default-class` annotation to `true` on an IngressClass resource will ensure that new Ingresses without an `ingressClassName` field specified will be assigned this default IngressClass.

Caution:

If you have more than one IngressClass marked as the default for your cluster, the admission controller prevents creating new Ingress objects that don't have an `ingressClassName` specified. You can resolve this by ensuring that at most 1 IngressClass is marked as default in your cluster.

There are some ingress controllers, that work without the definition of a default `IngressClass`. For example, the Ingress-NGINX controller can be configured with a flag `--watch-ingress-without-class`. It is recommended though, to specify the default `IngressClass`:

service/networking/default-ingressclass.yaml Copy service/networking/default-ingressclass.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: IngressClassmetadata:  labels:    app.kubernetes.io/component: controller  name: nginx-example  annotations:    ingressclass
```

Types of Ingress

Ingress backed by a single Service

There are existing Kubernetes concepts that allow you to expose a single Service (see alternatives). You can also do this with an Ingress by specifying a *default backend* with no rules.

service/networking/test-ingress.yaml Copy service/networking/test-ingress.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: Ingressmetadata:  name: test-ingressspec:  defaultBackend:    service:      name: test      port:        number: 80
```

If you create it using `kubectl apply -f` you should be able to view the state of the Ingress you added:

```
kubectl get ingress test-ingress
```

```
NAME           CLASS         HOSTS   ADDRESS         PORTS   AGE
test-ingress   external-lb   *       203.0.113.123   80      59s
```

Where `203.0.113.123` is the IP allocated by the Ingress controller to satisfy this Ingress.

Note:

Ingress controllers and load balancers may take a minute or two to allocate an IP address. Until that time, you often see the address listed as `<pending>`.

Simple fanout

A fanout configuration routes traffic from a single IP address to more than one Service, based on the HTTP URI being requested. An Ingress allows you to keep the number of load balancers down to a minimum. For example, a setup like:

ingress-fanout-diagram

Figure. Ingress Fan Out

It would require an Ingress such as:

service/networking/simple-fanout-example.yaml Copy service/networking/simple-fanout-example.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: Ingressmetadata:  name: simple-fanout-examplespec:  rules:  - host: foo.bar.com    http:      paths:      - path: /foo
```

When you create the Ingress with `kubectl apply -f`:

```
kubectl describe ingress simple-fanout-example
```

```
Name:             simple-fanout-example
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host         Path  Backends
  ----         ----  --------
  foo.bar.com
               /foo   service1:4200 (10.8.0.90:4200)
               /bar   service2:8080 (10.8.0.91:8080)
Events:
  Type     Reason  Age                From                     Message
  ----     ------  ----               ----                     -------
  Normal   ADD     22s                loadbalancer-controller  default/test
```

The Ingress controller provisions an implementation-specific load balancer that satisfies the Ingress, as long as the Services (`service1`, `service2`) exist. When it has done so, you can see the address of the load balancer at the Address field.

Note:

Depending on the Ingress controller you are using, you may need to create a default-http-backend Service.

Name based virtual hosting

Name-based virtual hosts support routing HTTP traffic to multiple host names at the same IP address.

ingress-namebase-diagram

Figure. Ingress Name Based Virtual hosting

The following Ingress tells the backing load balancer to route requests based on the Host header.

service/networking/name-virtual-host-ingress.yaml Copy service/networking/name-virtual-host-ingress.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: Ingressmetadata:  name: name-virtual-host-ingressspec:  rules:  - host: foo.bar.com    http:      paths:      - pathType: Pr
```

If you create an Ingress resource without any hosts defined in the rules, then any web traffic to the IP address of your Ingress controller can be matched without a name based virtual host being required.

For example, the following Ingress routes traffic requested for `first.bar.com` to `service1`, `second.bar.com` to `service2`, and any traffic whose request host header doesn't match `first.bar.com` and `second.bar.com` to `service3`.

service/networking/name-virtual-host-ingress-no-third-host.yaml
Copy service/networking/name-virtual-host-ingress-no-third-host.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: Ingressmetadata:  name: name-virtual-host-ingress-no-third-hostspec:  rules:  - host: first.bar.com    http:      paths:
```

TLS

You can secure an Ingress by specifying a Secret that contains a TLS private key and certificate. The Ingress resource only supports a single TLS port, 443, and assumes TLS termination at the ingress point (traffic to the Service and its Pods is in plaintext). If the TLS configuration section in an Ingress specifies different hosts, they are multiplexed on the same port according to the hostname specified through the SNI TLS extension (provided the Ingress controller supports SNI). The TLS secret must contain keys named `tls.crt` and `tls.key` that contain the certificate and private key to use for TLS. For example:

```
apiVersion: v1
kind: Secretmetadata:  name: testsecret-tls  namespace: defaultdata:  tls.crt: base64 encoded cert  tls.key: base64 encoded keytyp
```

Referencing this secret in an Ingress tells the Ingress controller to secure the channel from the client to the load balancer using TLS. You need to make sure the TLS secret you created came from a certificate that contains a Common Name (CN), also known as a Fully Qualified Domain Name (FQDN) for `https-example.foo.com`.

Note:

Keep in mind that TLS will not work on the default rule because the certificates would have to be issued for all the possible sub-domains. Therefore, `hosts` in the `tls` section need to explicitly match the `host` in the `rules` section.
`service/networking/tls-example-ingress.yaml` Copy service/networking/tls-example-ingress.yaml to clipboard

```
apiVersion: networking.k8s.io/v1
kind: Ingressmetadata:  name: tls-example-ingressspec:  tls:  - hosts:      - https-example.foo.com    secretName: testsecret-tls
```

Note:

There is a gap between TLS features supported by various Ingress controllers. Please refer to documentation on nginx, GCE, or any other platform specific Ingress controller to understand how TLS works in your environment.

Load balancing

An Ingress controller is bootstrapped with some load balancing policy settings that it applies to all Ingress, such as the load balancing algorithm, backend weight scheme, and others. More advanced load balancing concepts (e.g. persistent sessions, dynamic weights) are not yet exposed through the Ingress. You can instead get these features through the load balancer used for a Service.

It's also worth noting that even though health checks are not exposed directly through the Ingress, there exist parallel concepts in Kubernetes such as readiness probes that allow you to achieve the same end result. Please review the controller specific documentation to see how they handle health checks (for example: nginx, or GCE).

Updating an Ingress

To update an existing Ingress to add a new Host, you can update it by editing the resource:

```
kubectl describe ingress test
```

```
Name:             test
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host         Path  Backends
  ----         ----  --------
  foo.bar.com
               /foo   service1:80 (10.8.0.90:80)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target:  /
Events:
  Type     Reason  Age                From                     Message
  ----     ------  ----               ----                     -------
  Normal   ADD     35s                loadbalancer-controller  default/test
```

```
kubectl edit ingress test
```

This pops up an editor with the existing configuration in YAML format. Modify it to include the new Host:

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          service:
            name: service1
            port:
              number: 80
        path: /foo
        pathType: Prefix
  - host: bar.baz.com
    http:
      paths:
      - backend:
          service:
            name: service2
            port:
              number: 80
        path: /foo
        pathType: Prefix
..
```

After you save your changes, kubectl updates the resource in the API server, which tells the Ingress controller to reconfigure the load balancer.

Verify this:

```
kubectl describe ingress test
```

```
Name:             test
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host         Path  Backends
  ----         ----  --------
  foo.bar.com
               /foo   service1:80 (10.8.0.90:80)
  bar.baz.com
               /foo   service2:80 (10.8.0.91:80)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target:  /
Events:
  Type     Reason  Age                From                     Message
  ----     ------  ----               ----                     -------
  Normal   ADD     45s                loadbalancer-controller  default/test
```

You can achieve the same outcome by invoking `kubectl replace -f` on a modified Ingress YAML file.

Failing across availability zones

Techniques for spreading traffic across failure domains differ between cloud providers. Please check the documentation of the relevant Ingress controller for details.

Alternatives

You can expose a Service in multiple ways that don't directly involve the Ingress resource:

- Use Service.Type=LoadBalancer
- Use Service.Type=NodePort

What's next

- Learn about the Ingress API
- Learn about Ingress controllers

Kubernetes API Server Bypass Risks

Security architecture information relating to the API server and other components

The Kubernetes API server is the main point of entry to a cluster for external parties (users and services) interacting with it.

As part of this role, the API server has several key built-in security controls, such as audit logging and admission controllers. However, there are ways to modify the configuration or content of the cluster that bypass these controls.

This page describes the ways in which the security controls built into the Kubernetes API server can be bypassed, so that cluster operators and security architects can ensure that these bypasses are appropriately restricted.

Static Pods

The kubelet on each node loads and directly manages any manifests that are stored in a named directory or fetched from a specific URL as *static Pods* in your cluster. The API server doesn't manage these static Pods. An attacker with write access to this location could modify the configuration of static pods loaded from that source, or could introduce new static Pods.

Static Pods are restricted from accessing other objects in the Kubernetes API. For example, you can't configure a static Pod to mount a Secret from the cluster. However, these Pods can take other security sensitive actions, such as using `hostPath` mounts from the underlying node.

By default, the kubelet creates a mirror pod so that the static Pods are visible in the Kubernetes API. However, if the attacker uses an invalid namespace name when creating the Pod, it will not be visible in the Kubernetes API and can only be discovered by tooling that has access to the affected host(s).

If a static Pod fails admission control, the kubelet won't register the Pod with the API server. However, the Pod still runs on the node. For more information, refer to kubeadm issue #1541.

Mitigations

- Only enable the kubelet static Pod manifest functionality if required by the node.
- If a node uses the static Pod functionality, restrict filesystem access to the static Pod manifest directory or URL to users who need the access.
- Restrict access to kubelet configuration parameters and files to prevent an attacker setting a static Pod path or URL.
- Regularly audit and centrally report all access to directories or web storage locations that host static Pod manifests and kubelet configuration files.

The kubelet API

The kubelet provides an HTTP API that is typically exposed on TCP port 10250 on cluster worker nodes. The API might also be exposed on control plane nodes depending on the Kubernetes distribution in use. Direct access to the API allows for disclosure of information about the pods running on a node, the logs from those pods, and execution of commands in every container running on the node.

When Kubernetes cluster users have RBAC access to `Node` object sub-resources, that access serves as authorization to interact with the kubelet API. The exact access depends on which sub-resource access has been granted, as detailed in kubelet authorization.

Direct access to the kubelet API is not subject to admission control and is not logged by Kubernetes audit logging. An attacker with direct access to this API may be able to bypass controls that detect or prevent certain actions.

The kubelet API can be configured to authenticate requests in a number of ways. By default, the kubelet configuration allows anonymous access. Most Kubernetes providers change the default to use webhook and certificate authentication. This lets the control plane ensure that the caller is authorized to access the `nodes` API resource or sub-resources. The default anonymous access doesn't make this assertion with the control plane.

Mitigations

- Restrict access to sub-resources of the `nodes` API object using mechanisms such as RBAC. Only grant this access when required, such as by monitoring services.
- Restrict access to the kubelet port. Only allow specified and trusted IP address ranges to access the port.
- Ensure that kubelet authentication. is set to webhook or certificate mode.
- Ensure that the unauthenticated "read-only" Kubelet port is not enabled on the cluster.

The etcd API

Kubernetes clusters use etcd as a datastore. The `etcd` service listens on TCP port 2379. The only clients that need access are the Kubernetes API server and any backup tooling that you use. Direct access to this API allows for disclosure or modification of any data held in the cluster.

Access to the etcd API is typically managed by client certificate authentication. Any certificate issued by a certificate authority that etcd trusts allows full access to the data stored inside etcd.

Direct access to etcd is not subject to Kubernetes admission control and is not logged by Kubernetes audit logging. An attacker who has read access to the API server's etcd client certificate private key (or can create a new trusted client certificate) can gain cluster admin rights by accessing cluster secrets or modifying access rules. Even without elevating their Kubernetes RBAC privileges, an attacker who can modify etcd can retrieve any API object or create new workloads inside the cluster.

Many Kubernetes providers configure etcd to use mutual TLS (both client and server verify each other's certificate for authentication). There is no widely accepted implementation of authorization for the etcd API, although the feature exists. Since there is no authorization model, any certificate with client access to etcd can be used to gain full access to etcd. Typically, etcd client certificates that are only used for health checking can also grant full read and write access.

Mitigations

- Ensure that the certificate authority trusted by etcd is used only for the purposes of authentication to that service.
- Control access to the private key for the etcd server certificate, and to the API server's client certificate and key.
- Consider restricting access to the etcd port at a network level, to only allow access from specified and trusted IP address ranges.

Container runtime socket

On each node in a Kubernetes cluster, access to interact with containers is controlled by the container runtime (or runtimes, if you have configured more than one). Typically, the container runtime exposes a Unix socket that the kubelet can access. An attacker with access to this socket can launch new containers or interact with running containers.

At the cluster level, the impact of this access depends on whether the containers that run on the compromised node have access to Secrets or other confidential data that an attacker could use to escalate privileges to other worker nodes or to control plane components.

Mitigations

- Ensure that you tightly control filesystem access to container runtime sockets. When possible, restrict this access to the `root` user.
- Isolate the kubelet from other components running on the node, using mechanisms such as Linux kernel namespaces.
- Ensure that you restrict or forbid the use of `hostPath` mounts that include the container runtime socket, either directly or by mounting a parent directory. Also `hostPath` mounts must be set as read-only to mitigate risks of attackers bypassing directory restrictions.
- Restrict user access to nodes, and especially restrict superuser access to nodes.

Pod Security Admission

An overview of the Pod Security Admission Controller, which can enforce the Pod Security Standards.
FEATURE STATE: `Kubernetes v1.25 [stable]`

The Kubernetes Pod Security Standards define different isolation levels for Pods. These standards let you define how you want to restrict the behavior of pods in a clear, consistent fashion.

Kubernetes offers a built-in *Pod Security* admission controller to enforce the Pod Security Standards. Pod security restrictions are applied at the namespace level when pods are created.

Built-in Pod Security admission enforcement

This page is part of the documentation for Kubernetes v1.34. If you are running a different version of Kubernetes, consult the documentation for that release.

Pod Security levels

Pod Security admission places requirements on a Pod's Security Context and other related fields according to the three levels defined by the Pod Security Standards: `privileged`, `baseline`, and `restricted`. Refer to the Pod Security Standards page for an in-depth look at those requirements.

Pod Security Admission labels for namespaces

Once the feature is enabled or the webhook is installed, you can configure namespaces to define the admission control mode you want to use for pod security in each namespace. Kubernetes defines a set of labels that you can set to define which of the predefined Pod Security Standard levels you want to use for a namespace. The label you select defines what action the control plane takes if a potential violation is detected:

Mode	Description
enforce	Policy violations will cause the pod to be rejected.
audit	Policy violations will trigger the addition of an audit annotation to the event recorded in the audit log, but are otherwise allowed.
warn	Policy violations will trigger a user-facing warning, but are otherwise allowed.

A namespace can configure any or all modes, or even set a different level for different modes.

For each mode, there are two labels that determine the policy used:

```
# The per-mode level label indicates which policy level to apply for the mode.
## MODE must be one of `enforce`, `audit`, or `warn`.# LEVEL must be one of `privileged`, `baseline`, or `restricted`.pod-security
```

Check out Enforce Pod Security Standards with Namespace Labels to see example usage.

Workload resources and Pod templates

Pods are often created indirectly, by creating a workload object such as a Deployment or Job. The workload object defines a *Pod template* and a controller for the workload resource creates Pods based on that template. To help catch violations early, both the audit and warning modes are applied to the workload resources. However, enforce mode is **not** applied to workload resources, only to the resulting pod objects.

Exemptions

You can define *exemptions* from pod security enforcement in order to allow the creation of pods that would have otherwise been prohibited due to the policy associated with a given namespace. Exemptions can be statically configured in the Admission Controller configuration.

Exemptions must be explicitly enumerated. Requests meeting exemption criteria are *ignored* by the Admission Controller (all `enforce`, `audit` and `warn` behaviors are skipped). Exemption dimensions include:

- **Usernames:** requests from users with an exempt authenticated (or impersonated) username are ignored.
- **RuntimeClassNames:** pods and workload resources specifying an exempt runtime class name are ignored.
- **Namespaces:** pods and workload resources in an exempt namespace are ignored.

Caution:

Most pods are created by a controller in response to a workload resource, meaning that exempting an end user will only exempt them from enforcement when creating pods directly, but not when creating a workload resource. Controller service accounts (such as `system:serviceaccount:kube-system:replicaset-controller`) should generally not be exempted, as doing so would implicitly exempt any user that can create the corresponding workload resource.

Updates to the following pod fields are exempt from policy checks, meaning that if a pod update request only changes these fields, it will not be denied even if the pod is in violation of the current policy level:

- Any metadata updates **except** changes to the seccomp or AppArmor annotations:
  - `seccomp.security.alpha.kubernetes.io/pod` (deprecated)
  - `container.seccomp.security.alpha.kubernetes.io/*` (deprecated)
  - `container.apparmor.security.beta.kubernetes.io/*` (deprecated)
- Valid updates to `.spec.activeDeadlineSeconds`
- Valid updates to `.spec.tolerations`

Metrics

Here are the Prometheus metrics exposed by kube-apiserver:

- `pod_security_errors_total`: This metric indicates the number of errors preventing normal evaluation. Non-fatal errors may result in the latest restricted profile being used for enforcement.
- `pod_security_evaluations_total`: This metric indicates the number of policy evaluations that have occurred, not counting ignored or exempt requests during exporting.
- `pod_security_exemptions_total`: This metric indicates the number of exempt requests, not counting ignored or out of scope requests.

What's next

- Pod Security Standards
- Enforcing Pod Security Standards
- Enforce Pod Security Standards by Configuring the Built-in Admission Controller
- Enforce Pod Security Standards with Namespace Labels

If you are running an older version of Kubernetes and want to upgrade to a version of Kubernetes that does not include PodSecurityPolicies, read migrate from PodSecurityPolicy to the Built-In PodSecurity Admission Controller.