Runtime Class

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

This page describes the RuntimeClass resource and runtime selection mechanism.

RuntimeClass is a feature for selecting the container runtime configuration. The container runtime configuration is used to run a Pod's containers.

Motivation

You can set a different RuntimeClass between different Pods to provide a balance of performance versus security. For example, if part of your workload deserves a high level of information security assurance, you might choose to schedule those Pods so that they run in a container runtime that uses hardware virtualization. You'd then benefit from the extra isolation of the alternative runtime, at the expense of some additional overhead.

You can also use RuntimeClass to run different Pods with the same container runtime but with different settings.

Setup

1. Configure the CRI implementation on nodes (runtime dependent)
2. Create the corresponding RuntimeClass resources

1. Configure the CRI implementation on nodes

The configurations available through RuntimeClass are Container Runtime Interface (CRI) implementation dependent. See the corresponding documentation (below) for your CRI implementation for how to configure.

Note:

RuntimeClass assumes a homogeneous node configuration across the cluster by default (which means that all nodes are configured the same way with respect to container runtimes). To support heterogeneous node configurations, see Scheduling below.

The configurations have a corresponding `handler` name, referenced by the RuntimeClass. The handler must be a valid DNS label name.

2. Create the corresponding RuntimeClass resources

The configurations setup in step 1 should each have an associated `handler` name, which identifies the configuration. For each handler, create a corresponding RuntimeClass object.

The RuntimeClass resource currently only has 2 significant fields: the RuntimeClass name (`metadata.name`) and the handler (`handler`). The object definition looks like this:

```
# RuntimeClass is defined in the node.k8s.io API group
apiVersion: node.k8s.io/v1kind: RuntimeClassmetadata:  # The name the RuntimeClass will be referenced by.  # RuntimeClass is a non
```

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

Note:

It is recommended that RuntimeClass write operations (create/update/patch/delete) be restricted to the cluster administrator. This is typically the default. See Authorization Overview for more details.

Usage

Once RuntimeClasses are configured for the cluster, you can specify a `runtimeClassName` in the Pod spec to use it. For example:

```
apiVersion: v1
kind: Podmetadata:  name: mypodspec:  runtimeClassName: myclass  # ...
```

This will instruct the kubelet to use the named RuntimeClass to run this pod. If the named RuntimeClass does not exist, or the CRI cannot run the corresponding handler, the pod will enter the `Failed` terminal phase. Look for a corresponding event for an error message.

If no `runtimeClassName` is specified, the default RuntimeHandler will be used, which is equivalent to the behavior when the RuntimeClass feature is disabled.

CRI Configuration

For more details on setting up CRI runtimes, see CRI installation.

containerd

Runtime handlers are configured through containerd's configuration at `/etc/containerd/config.toml`. Valid handlers are configured under the runtimes section:

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.${HANDLER_NAME}]
```

See containerd's config documentation for more details:

CRI-O

Runtime handlers are configured through CRI-O's configuration at `/etc/crio/crio.conf`. Valid handlers are configured under the crio.runtime table:

```
[crio.runtime.runtimes.${HANDLER_NAME}]
  runtime_path = "${PATH_TO_BINARY}"
```

See CRI-O's config documentation for more details.

Scheduling

FEATURE STATE: `Kubernetes v1.16 [beta]`

By specifying the `scheduling` field for a RuntimeClass, you can set constraints to ensure that Pods running with this RuntimeClass are scheduled to nodes that support it. If `scheduling` is not set, this RuntimeClass is assumed to be supported by all nodes.

To ensure pods land on nodes supporting a specific RuntimeClass, that set of nodes should have a common label which is then selected by the `runtimeclass.scheduling.nodeSelector` field. The RuntimeClass's nodeSelector is merged with the pod's nodeSelector in admission, effectively taking the intersection of the set of nodes selected by each. If there is a conflict, the pod will be rejected.

If the supported nodes are tainted to prevent other RuntimeClass pods from running on the node, you can add `tolerations` to the RuntimeClass. As with the `nodeSelector`, the tolerations are merged with the pod's tolerations in admission, effectively taking the union of the set of nodes tolerated by each.

To learn more about configuring the node selector and tolerations, see Assigning Pods to Nodes.

Pod Overhead

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

You can specify *overhead* resources that are associated with running a Pod. Declaring overhead allows the cluster (including the scheduler) to account for it when making decisions about Pods and resources.

Pod overhead is defined in RuntimeClass through the `overhead` field. Through the use of this field, you can specify the overhead of running pods utilizing this RuntimeClass and ensure these overheads are accounted for in Kubernetes.

What's next

- RuntimeClass Design
- RuntimeClass Scheduling Design
- Read about the Pod Overhead concept
- PodOverhead Feature Design

Admission Webhook Good Practices

Recommendations for designing and deploying admission webhooks in Kubernetes.

This page provides good practices and considerations when designing *admission webhooks* in Kubernetes. This information is intended for cluster operators who run admission webhook servers or third-party applications that modify or validate your API requests.

Before reading this page, ensure that you're familiar with the following concepts:

- Admission controllers
- Admission webhooks

Importance of good webhook design

Admission control occurs when any create, update, or delete request is sent to the Kubernetes API. Admission controllers intercept requests that match specific criteria that you define. These requests are then sent to mutating admission webhooks or validating admission webhooks. These webhooks are often written to ensure that specific fields in object specifications exist or have specific allowed values.

Webhooks are a powerful mechanism to extend the Kubernetes API. Badly-designed webhooks often result in workload disruptions because of how much control the webhooks have over objects in the cluster. Like other API extension mechanisms, webhooks are challenging to test at scale for compatibility with all of your workloads, other webhooks, add-ons, and plugins.

Additionally, with every release, Kubernetes adds or modifies the API with new features, feature promotions to beta or stable status, and deprecations. Even stable Kubernetes APIs are likely to change. For example, the `Pod` API changed in v1.29 to add the Sidecar containers feature. While it's rare for a Kubernetes object to enter a broken state because of a new Kubernetes API, webhooks that worked as expected with earlier versions of an API might not be able to reconcile more recent changes to that API. This can result in unexpected behavior after you upgrade your clusters to newer versions.

This page describes common webhook failure scenarios and how to avoid them by cautiously and thoughtfully designing and implementing your webhooks.

Identify whether you use admission webhooks

Even if you don't run your own admission webhooks, some third-party applications that you run in your clusters might use mutating or validating admission webhooks.

To check whether your cluster has any mutating admission webhooks, run the following command:

```
kubectl get mutatingwebhookconfigurations
```

The output lists any mutating admission controllers in the cluster.

To check whether your cluster has any validating admission webhooks, run the following command:

```
kubectl get validatingwebhookconfigurations
```

The output lists any validating admission controllers in the cluster.

Choose an admission control mechanism

Kubernetes includes multiple admission control and policy enforcement options. Knowing when to use a specific option can help you to improve latency and performance, reduce management overhead, and avoid issues during version upgrades. The following table describes the mechanisms that let you mutate or validate resources during admission:

Mutating and validating admission control in Kubernetes

Mechanism	Description	Use cases
Mutating admission webhook	Intercept API requests before admission and modify as needed using custom logic.	• Make critical modifications that must happen before resource admission. • Make complex modifications that require advanced logic, like calling external APIs.
Mutating admission policy	Intercept API requests before admission and modify as needed using Common Expression Language (CEL) expressions.	• Make critical modifications that must happen before resource admission. • Make simple modifications, such as adjusting labels or replica counts.
Validating admission webhook	Intercept API requests before admission and validate against complex policy declarations.	• Validate critical configurations before resource admission. • Enforce complex policy logic before admission.
Validating admission policy	Intercept API requests before admission and validate against CEL expressions.	• Validate critical configurations before resource admission. • Enforce policy logic using CEL expressions.

In general, use *webhook* admission control when you want an extensible way to declare or configure the logic. Use built-in CEL-based admission control when you want to declare simpler logic without the overhead of running a webhook server. The Kubernetes project recommends that you use CEL-based admission control when possible.

Use built-in validation and defaulting for CustomResourceDefinitions

If you use CustomResourceDefinitions, don't use admission webhooks to validate values in CustomResource specifications or to set default values for fields. Kubernetes lets you define validation rules and default field values when you create CustomResourceDefinitions.

To learn more, see the following resources:

- Validation rules
- Defaulting

Performance and latency

This section describes recommendations for improving performance and reducing latency. In summary, these are as follows:

- Consolidate webhooks and limit the number of API calls per webhook.
- Use audit logs to check for webhooks that repeatedly do the same action.
- Use load balancing for webhook availability.
- Set a small timeout value for each webhook.
- Consider cluster availability needs during webhook design.

Design admission webhooks for low latency

Mutating admission webhooks are called in sequence. Depending on the mutating webhook setup, some webhooks might be called multiple times. Every mutating webhook call adds latency to the admission process. This is unlike validating webhooks, which get called in parallel.

When designing your mutating webhooks, consider your latency requirements and tolerance. The more mutating webhooks there are in your cluster, the greater the chance of latency increases.

Consider the following to reduce latency:

- Consolidate webhooks that perform a similar mutation on different objects.
- Reduce the number of API calls made in the mutating webhook server logic.
- Limit the match conditions of each mutating webhook to reduce how many webhooks are triggered by a specific API request.
- Consolidate small webhooks into one server and configuration to help with ordering and organization.

Prevent loops caused by competing controllers

Consider any other components that run in your cluster that might conflict with the mutations that your webhook makes. For example, if your webhook adds a label that a different controller removes, your webhook gets called again. This leads to a loop.

To detect these loops, try the following:

1. Update your cluster audit policy to log audit events. Use the following parameters:

   - `level`: `RequestResponse`
   - `verbs`: `["patch"]`
   - `omitStages`: `RequestReceived`

   Set the audit rule to create events for the specific resources that your webhook mutates.

2. Check your audit events for webhooks being reinvoked multiple times with the same patch being applied to the same object, or for an object having a field updated and reverted multiple times.

Set a small timeout value

Admission webhooks should evaluate as quickly as possible (typically in milliseconds), since they add to API request latency. Use a small timeout for webhooks.

For details, see Timeouts.

Use a load balancer to ensure webhook availability

Admission webhooks should leverage some form of load-balancing to provide high availability and performance benefits. If a webhook is running within the cluster, you can run multiple webhook backends behind a Service of type `ClusterIP`.

Use a high-availability deployment model

Consider your cluster's availability requirements when designing your webhook. For example, during node downtime or zonal outages, Kubernetes marks Pods as `NotReady` to allow load balancers to reroute traffic to available zones and nodes. These updates to Pods might trigger your mutating webhooks. Depending on the number of affected Pods, the mutating webhook server has a risk of timing out or causing delays in Pod processing. As a result, traffic won't get rerouted as quickly as you need.

Consider situations like the preceding example when writing your webhooks. Exclude operations that are a result of Kubernetes responding to unavoidable incidents.

Request filtering

This section provides recommendations for filtering which requests trigger specific webhooks. In summary, these are as follows:

- Limit the webhook scope to avoid system components and read-only requests.
- Limit webhooks to specific namespaces.
- Use match conditions to perform fine-grained request filtering.
- Match all versions of an object.

Limit the scope of each webhook

Admission webhooks are only called when an API request matches the corresponding webhook configuration. Limit the scope of each webhook to reduce unnecessary calls to the webhook server. Consider the following scope limitations:

- Avoid matching objects in the `kube-system` namespace. If you run your own Pods in the `kube-system` namespace, use an `objectSelector` to avoid mutating a critical workload.
- Don't mutate node leases, which exist as Lease objects in the `kube-node-lease` system namespace. Mutating node leases might result in failed node upgrades. Only apply validation controls to Lease objects in this namespace if you're confident that the controls won't put your cluster at risk.
- Don't mutate TokenReview or SubjectAccessReview objects. These are always read-only requests. Modifying these objects might break your cluster.
- Limit each webhook to a specific namespace by using a `namespaceSelector`.

Filter for specific requests by using match conditions

Admission controllers support multiple fields that you can use to match requests that meet specific criteria. For example, you can use a `namespaceSelector` to filter for requests that target a specific namespace.

For more fine-grained request filtering, use the `matchConditions` field in your webhook configuration. This field lets you write multiple CEL expressions that must evaluate to `true` for a request to trigger your admission webhook. Using `matchConditions` might significantly reduce the number of calls to your webhook server.

For details, see Matching requests: `matchConditions`.

Match all versions of an API

By default, admission webhooks run on any API versions that affect a specified resource. The `matchPolicy` field in the webhook configuration controls this behavior. Specify a value of `Equivalent` in the `matchPolicy` field or omit the field to allow the webhook to run on any API version.

For details, see Matching requests: `matchPolicy`.

Mutation scope and field considerations

This section provides recommendations for the scope of mutations and any special considerations for object fields. In summary, these are as follows:

- Patch only the fields that you need to patch.
- Don't overwrite array values.
- Avoid side effects in mutations when possible.
- Avoid self-mutations.
- Fail open and validate the final state.
- Plan for future field updates in later versions.
- Prevent webhooks from self-triggering.
- Don't change immutable objects.

Patch only required fields

Admission webhook servers send HTTP responses to indicate what to do with a specific Kubernetes API request. This response is an AdmissionReview object. A mutating webhook can add specific fields to mutate before allowing admission by using the `patchType` field and the `patch` field in the response. Ensure that you only modify the fields that require a change.

For example, consider a mutating webhook that's configured to ensure that `web-server` Deployments have at least three replicas. When a request to create a Deployment object matches your webhook configuration, the webhook should only update the value in the `spec.replicas` field.

Don't overwrite array values

Fields in Kubernetes object specifications might include arrays. Some arrays contain key:value pairs (like the `envVar` field in a container specification), while other arrays are unkeyed (like the `readinessGates` field in a Pod specification). The order of values in an array field might matter in some situations. For example, the order of arguments in the `args` field of a container specification might affect the container.

Consider the following when modifying arrays:

- Whenever possible, use the `add` JSONPatch operation instead of `replace` to avoid accidentally replacing a required value.
- Treat arrays that don't use key:value pairs as sets.
- Ensure that the values in the field that you modify aren't required to be in a specific order.
- Don't overwrite existing key:value pairs unless absolutely necessary.
- Use caution when modifying label fields. An accidental modification might cause label selectors to break, resulting in unintended behavior.

Avoid side effects

Ensure that your webhooks operate only on the content of the AdmissionReview that's sent to them, and do not make out-of-band changes. These additional changes, called *side effects*, might cause conflicts during admission if they aren't reconciled properly. The `.webhooks[].sideEffects` field should be set to `None` if a webhook doesn't have any side effect.

If side effects are required during the admission evaluation, they must be suppressed when processing an AdmissionReview object with `dryRun` set to `true`, and the `.webhooks[].sideEffects` field should be set to `NoneOnDryRun`.

For details, see Side effects.

Avoid self-mutations

A webhook running inside the cluster might cause deadlocks for its own deployment if it is configured to intercept resources required to start its own Pods.

For example, a mutating admission webhook is configured to admit **create** Pod requests only if a certain label is set in the Pod (such as `env: prod`). The webhook server runs in a Deployment that doesn't set the `env` label.

When a node that runs the webhook server Pods becomes unhealthy, the webhook Deployment tries to reschedule the Pods to another node. However, the existing webhook server rejects the requests since the `env` label is unset. As a result, the migration cannot happen.

Exclude the namespace where your webhook is running with a `namespaceSelector`.

Avoid dependency loops

Dependency loops can occur in scenarios like the following:

- Two webhooks check each other's Pods. If both webhooks become unavailable at the same time, neither webhook can start.
- Your webhook intercepts cluster add-on components, such as networking plugins or storage plugins, that your webhook depends on. If both the webhook and the dependent add-on become unavailable, neither component can function.

To avoid these dependency loops, try the following:

- Use ValidatingAdmissionPolicies to avoid introducing dependencies.
- Prevent webhooks from validating or mutating other webhooks. Consider excluding specific namespaces from triggering your webhook.
- Prevent your webhooks from acting on dependent add-ons by using an `objectSelector`.

Fail open and validate the final state

Mutating admission webhooks support the `failurePolicy` configuration field. This field indicates whether the API server should admit or reject the request if the webhook fails. Webhook failures might occur because of timeouts or errors in the server logic.

By default, admission webhooks set the `failurePolicy` field to Fail. The API server rejects a request if the webhook fails. However, rejecting requests by default might result in compliant requests being rejected during webhook downtime.

Let your mutating webhooks "fail open" by setting the `failurePolicy` field to Ignore. Use a validating controller to check the state of requests to ensure that they comply with your policies.

This approach has the following benefits:

- Mutating webhook downtime doesn't affect compliant resources from deploying.
- Policy enforcement occurs during validating admission control.
- Mutating webhooks don't interfere with other controllers in the cluster.

Plan for future updates to fields

In general, design your webhooks under the assumption that Kubernetes APIs might change in a later version. Don't write a server that takes the stability of an API for granted. For example, the release of sidecar containers in Kubernetes added a `restartPolicy` field to the Pod API.

Prevent your webhook from triggering itself

Mutating webhooks that respond to a broad range of API requests might unintentionally trigger themselves. For example, consider a webhook that responds to all requests in the cluster. If you configure the webhook to create Event objects for every mutation, it'll respond to its own Event object creation requests.

To avoid this, consider setting a unique label in any resources that your webhook creates. Exclude this label from your webhook match conditions.

Don't change immutable objects

Some Kubernetes objects in the API server can't change. For example, when you deploy a static Pod, the kubelet on the node creates a mirror Pod in the API server to track the static Pod. However, changes to the mirror Pod don't propagate to the static Pod.

Don't attempt to mutate these objects during admission. All mirror Pods have the `kubernetes.io/config.mirror` annotation. To exclude mirror Pods while reducing the security risk of ignoring an annotation, allow static Pods to only run in specific namespaces.

Mutating webhook ordering and idempotence

This section provides recommendations for webhook order and designing idempotent webhooks. In summary, these are as follows:

- Don't rely on a specific order of execution.
- Validate mutations before admission.
- Check for mutations being overwritten by other controllers.
- Ensure that the set of mutating webhooks is idempotent, not just the individual webhooks.

Don't rely on mutating webhook invocation order

Mutating admission webhooks don't run in a consistent order. Various factors might change when a specific webhook is called. Don't rely on your webhook running at a specific point in the admission process. Other webhooks could still mutate your modified object.

The following recommendations might help to minimize the risk of unintended changes:

- Validate mutations before admission
- Use a reinvocation policy to observe changes to an object by other plugins and re-run the webhook as needed. For details, see Reinvocation policy.

Ensure that the mutating webhooks in your cluster are idempotent

Every mutating admission webhook should be *idempotent*. The webhook should be able to run on an object that it already modified without making additional changes beyond the original change.

Additionally, all of the mutating webhooks in your cluster should, as a collection, be idempotent. After the mutation phase of admission control ends, every individual mutating webhook should be able to run on an object without making additional changes to the object.

Depending on your environment, ensuring idempotence at scale might be challenging. The following recommendations might help:

- Use validating admission controllers to verify the final state of critical workloads.
- Test your deployments in a staging cluster to see if any objects get modified multiple times by the same webhook.
- Ensure that the scope of each mutating webhook is specific and limited.

The following examples show idempotent mutation logic:

1. For a **create** Pod request, set the field `.spec.securityContext.runAsNonRoot` of the Pod to true.

2. For a **create** Pod request, if the field `.spec.containers[].resources.limits` of a container is not set, set default resource limits.

3. For a **create** Pod request, inject a sidecar container with name `foo-sidecar` if no container with the name `foo-sidecar` already exists.

In these cases, the webhook can be safely reinvoked, or admit an object that already has the fields set.

The following examples show non-idempotent mutation logic:

1. For a **create** Pod request, inject a sidecar container with name `foo-sidecar` suffixed with the current timestamp (such as `foo-sidecar-19700101-000000`).

   Reinvoking the webhook can result in the same sidecar being injected multiple times to a Pod, each time with a different container name. Similarly, the webhook can inject duplicated containers if the sidecar already exists in a user-provided pod.

2. For a **create/update** Pod request, reject if the Pod has label `env` set, otherwise add an `env: prod` label to the Pod.

Reinvoking the webhook will result in the webhook failing on its own output.

3. For a **create** Pod request, append a sidecar container named `foo-sidecar` without checking whether a `foo-sidecar` container exists.

   Reinvoking the webhook will result in duplicated containers in the Pod, which makes the request invalid and rejected by the API server.

Mutation testing and validation

This section provides recommendations for testing your mutating webhooks and validating mutated objects. In summary, these are as follows:

- Test webhooks in staging environments.
- Avoid mutations that violate validations.
- Test minor version upgrades for regressions and conflicts.
- Validate mutated objects before admission.

Test webhooks in staging environments

Robust testing should be a core part of your release cycle for new or updated webhooks. If possible, test any changes to your cluster webhooks in a staging environment that closely resembles your production clusters. At the very least, consider using a tool like minikube or kind to create a small test cluster for webhook changes.

Ensure that mutations don't violate validations

Your mutating webhooks shouldn't break any of the validations that apply to an object before admission. For example, consider a mutating webhook that sets the default CPU request of a Pod to a specific value. If the CPU limit of that Pod is set to a lower value than the mutated request, the Pod fails admission.

Test every mutating webhook against the validations that run in your cluster.

Test minor version upgrades to ensure consistent behavior

Before upgrading your production clusters to a new minor version, test your webhooks and workloads in a staging environment. Compare the results to ensure that your webhooks continue to function as expected after the upgrade.

Additionally, use the following resources to stay informed about API changes:

- Kubernetes release notes
- Kubernetes blog

Validate mutations before admission

Mutating webhooks run to completion before any validating webhooks run. There is no stable order in which mutations are applied to objects. As a result, your mutations could get overwritten by a mutating webhook that runs at a later time.

Add a validating admission controller like a ValidatingAdmissionWebhook or a ValidatingAdmissionPolicy to your cluster to ensure that your mutations are still present. For example, consider a mutating webhook that inserts the `restartPolicy: Always` field to specific init containers to make them run as sidecar containers. You could run a validating webhook to ensure that those init containers retained the `restartPolicy: Always` configuration after all mutations were completed.

For details, see the following resources:

- Validating Admission Policy
- ValidatingAdmissionWebhooks

Mutating webhook deployment

This section provides recommendations for deploying your mutating admission webhooks. In summary, these are as follows:

- Gradually roll out the webhook configuration and monitor for issues by namespace.
- Limit access to edit the webhook configuration resources.
- Limit access to the namespace that runs the webhook server, if the server is in-cluster.

Install and enable a mutating webhook

When you're ready to deploy your mutating webhook to a cluster, use the following order of operations:

1. Install the webhook server and start it.
2. Set the `failurePolicy` field in the MutatingWebhookConfiguration manifest to Ignore. This lets you avoid disruptions caused by misconfigured webhooks.
3. Set the `namespaceSelector` field in the MutatingWebhookConfiguration manifest to a test namespace.
4. Deploy the MutatingWebhookConfiguration to your cluster.

Monitor the webhook in the test namespace to check for any issues, then roll the webhook out to other namespaces. If the webhook intercepts an API request that it wasn't meant to intercept, pause the rollout and adjust the scope of the webhook configuration.

Limit edit access to mutating webhooks

Mutating webhooks are powerful Kubernetes controllers. Use RBAC or another authorization mechanism to limit access to your webhook configurations and servers. For RBAC, ensure that the following access is only available to trusted entities:

- Verbs: **create**, **update**, **patch**, **delete**, **deletecollection**
- API group: `admissionregistration.k8s.io/v1`
- API kind: MutatingWebhookConfigurations

If your mutating webhook server runs in the cluster, limit access to create or modify any resources in that namespace.

Examples of good 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 projects are examples of "good" custom webhook server implementations. You can use them as a starting point when designing your own webhooks. Don't use these examples as-is; use them as a starting point and design your webhooks to run well in your specific environment.

- `cert-manager`
- Gatekeeper Open Policy Agent (OPA)

What's next

- Use webhooks for authentication and authorization
- Learn about MutatingAdmissionPolicies
- Learn about ValidatingAdmissionPolicies

Logging Architecture

Application logs can help you understand what is happening inside your application. The logs are particularly useful for debugging problems and monitoring cluster activity. Most modern applications have some kind of logging mechanism. Likewise, container engines are designed to support logging. The easiest and most adopted logging method for containerized applications is writing to standard output and standard error streams.

However, the native functionality provided by a container engine or runtime is usually not enough for a complete logging solution.

For example, you may want to access your application's logs if a container crashes, a pod gets evicted, or a node dies.

In a cluster, logs should have a separate storage and lifecycle independent of nodes, pods, or containers. This concept is called cluster-level logging.

Cluster-level logging architectures require a separate backend to store, analyze, and query logs. Kubernetes does not provide a native storage solution for log data. Instead, there are many logging solutions that integrate with Kubernetes. The following sections describe how to handle and store logs on nodes.

Pod and container logs

Kubernetes captures logs from each container in a running Pod.

This example uses a manifest for a `Pod` with a container that writes text to the standard output stream, once per second.

`debug/counter-pod.yaml` Copy debug/counter-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: counterspec:  containers:  - name: count    image: busybox:1.28    args: [/bin/sh, -c,           'i=0; 
```

To run this pod, use the following command:

```
kubectl apply -f https://k8s.io/examples/debug/counter-pod.yaml
```

The output is:

```
pod/counter created
```

To fetch the logs, use the `kubectl logs` command, as follows:

```
kubectl logs counter
```

The output is similar to:

```
0: Fri Apr  1 11:42:23 UTC 2022
1: Fri Apr  1 11:42:24 UTC 2022
2: Fri Apr  1 11:42:25 UTC 2022
```

You can use `kubectl logs --previous` to retrieve logs from a previous instantiation of a container. If your pod has multiple containers, specify which container's logs you want to access by appending a container name to the command, with a `-c` flag, like so:

```
kubectl logs counter -c count
```

Container log streams

FEATURE STATE: `Kubernetes v1.32 [alpha]` (enabled by default: false)

As an alpha feature, the kubelet can split out the logs from the two standard streams produced by a container: standard output and standard error. To use this behavior, you must enable the `PodLogsQuerySplitStreams` feature gate. With that feature gate enabled, Kubernetes 1.34 allows access to these log streams directly via the Pod API. You can fetch a specific stream by specifying the stream name (either `Stdout` or `Stderr`), using the `stream` query string. You must have access to read the `log` subresource of that Pod.

To demonstrate this feature, you can create a Pod that periodically writes text to both the standard output and error stream.

`debug/counter-pod-err.yaml` Copy debug/counter-pod-err.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: counter-errspec:  containers:  - name: count    image: busybox:1.28    args: [/bin/sh, -c,            'i
```

To run this pod, use the following command:

```
kubectl apply -f https://k8s.io/examples/debug/counter-pod-err.yaml
```

To fetch only the stderr log stream, you can run:

```
kubectl get --raw "/api/v1/namespaces/default/pods/counter-err/log?stream=Stderr"
```

See the `kubectl logs` documentation for more details.

How nodes handle container logs

Node level logging

A container runtime handles and redirects any output generated to a containerized application's `stdout` and `stderr` streams. Different container runtimes implement this in different ways; however, the integration with the kubelet is standardized as the *CRI logging format*.

By default, if a container restarts, the kubelet keeps one terminated container with its logs. If a pod is evicted from the node, all corresponding containers are also evicted, along with their logs.

The kubelet makes logs available to clients via a special feature of the Kubernetes API. The usual way to access this is by running `kubectl logs`.

Log rotation

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

The kubelet is responsible for rotating container logs and managing the logging directory structure. The kubelet sends this information to the container runtime (using CRI), and the runtime writes the container logs to the given location.

You can configure two kubelet configuration settings, `containerLogMaxSize` (default 10Mi) and `containerLogMaxFiles` (default 5), using the kubelet configuration file. These settings let you configure the maximum size for each log file and the maximum number of files allowed for each container respectively.

In order to perform an efficient log rotation in clusters where the volume of the logs generated by the workload is large, kubelet also provides a mechanism to tune how the logs are rotated in terms of how many concurrent log rotations can be performed and the interval at which the logs are monitored and rotated as required. You can configure two kubelet configuration settings, `containerLogMaxWorkers` and `containerLogMonitorInterval` using the kubelet configuration file.

When you run `kubectl logs` as in the basic logging example, the kubelet on the node handles the request and reads directly from the log file. The kubelet returns the content of the log file.

Note:

Only the contents of the latest log file are available through `kubectl logs`.

For example, if a Pod writes 40 MiB of logs and the kubelet rotates logs after 10 MiB, running `kubectl logs` returns at most 10MiB of data.

System component logs

There are two types of system components: those that typically run in a container, and those components directly involved in running containers. For example:

- The kubelet and container runtime do not run in containers. The kubelet runs your containers (grouped together in pods)
- The Kubernetes scheduler, controller manager, and API server run within pods (usually static Pods). The etcd component runs in the control plane, and most commonly also as a static pod. If your cluster uses kube-proxy, you typically run this as a `DaemonSet`.

Log locations

The way that the kubelet and container runtime write logs depends on the operating system that the node uses:

- Linux
- Windows

On Linux nodes that use systemd, the kubelet and container runtime write to journald by default. You use `journalctl` to read the systemd journal; for example: `journalctl -u kubelet`.

If systemd is not present, the kubelet and container runtime write to `.log` files in the `/var/log` directory. If you want to have logs written elsewhere, you can indirectly run the kubelet via a helper tool, `kube-log-runner`, and use that tool to redirect kubelet logs to a directory that you choose.

By default, kubelet directs your container runtime to write logs into directories within `/var/log/pods`.

For more information on `kube-log-runner`, read System Logs.

By default, the kubelet writes logs to files within the directory `C:\var\logs` (notice that this is not `C:\var\log`).

Although `C:\var\log` is the Kubernetes default location for these logs, several cluster deployment tools set up Windows nodes to log to `C:\var\log\kubelet` instead.

If you want to have logs written elsewhere, you can indirectly run the kubelet via a helper tool, `kube-log-runner`, and use that tool to redirect kubelet logs to a directory that you choose.

However, by default, kubelet directs your container runtime to write logs within the directory `C:\var\log\pods`.

For more information on `kube-log-runner`, read System Logs.

For Kubernetes cluster components that run in pods, these write to files inside the `/var/log` directory, bypassing the default logging mechanism (the components do not write to the systemd journal). You can use Kubernetes' storage mechanisms to map persistent storage into the container that runs the component.

Kubelet allows changing the pod logs directory from default `/var/log/pods` to a custom path. This adjustment can be made by configuring the `podLogsDir` parameter in the kubelet's configuration file.

Caution:

It's important to note that the default location `/var/log/pods` has been in use for an extended period and certain processes might implicitly assume this path. Therefore, altering this parameter must be approached with caution and at your own risk.

Another caveat to keep in mind is that the kubelet supports the location being on the same disk as `/var`. Otherwise, if the logs are on a separate filesystem from `/var`, then the kubelet will not track that filesystem's usage, potentially leading to issues if it fills up.

For details about etcd and its logs, view the etcd documentation. Again, you can use Kubernetes' storage mechanisms to map persistent storage into the container that runs the component.

Note:

If you deploy Kubernetes cluster components (such as the scheduler) to log to a volume shared from the parent node, you need to consider and ensure that those logs are rotated. **Kubernetes does not manage that log rotation**.

Your operating system may automatically implement some log rotation - for example, if you share the directory `/var/log` into a static Pod for a component, node-level log rotation treats a file in that directory the same as a file written by any component outside Kubernetes.

Some deploy tools account for that log rotation and automate it; others leave this as your responsibility.

Cluster-level logging architectures

While Kubernetes does not provide a native solution for cluster-level logging, there are several common approaches you can consider. Here are some options:

- Use a node-level logging agent that runs on every node.
- Include a dedicated sidecar container for logging in an application pod.
- Push logs directly to a backend from within an application.

Using a node logging agent

Using a node level logging agent

You can implement cluster-level logging by including a *node-level logging agent* on each node. The logging agent is a dedicated tool that exposes logs or pushes logs to a backend. Commonly, the logging agent is a container that has access to a directory with log files from all of the application containers on that node.

Because the logging agent must run on every node, it is recommended to run the agent as a `DaemonSet`.

Node-level logging creates only one agent per node and doesn't require any changes to the applications running on the node.

Containers write to stdout and stderr, but with no agreed format. A node-level agent collects these logs and forwards them for aggregation.

Using a sidecar container with the logging agent

You can use a sidecar container in one of the following ways:

- The sidecar container streams application logs to its own `stdout`.
- The sidecar container runs a logging agent, which is configured to pick up logs from an application container.

Streaming sidecar container

Sidecar container with a streaming container

By having your sidecar containers write to their own `stdout` and `stderr` streams, you can take advantage of the kubelet and the logging agent that already run on each node. The sidecar containers read logs from a file, a socket, or journald. Each sidecar container prints a log to its own `stdout` or `stderr` stream.

This approach allows you to separate several log streams from different parts of your application, some of which can lack support for writing to `stdout` or `stderr`. The logic behind redirecting logs is minimal, so it's not a significant overhead. Additionally, because `stdout` and `stderr` are handled by the kubelet, you can use built-in tools like `kubectl logs`.

For example, a pod runs a single container, and the container writes to two different log files using two different formats. Here's a manifest for the Pod:

admin/logging/two-files-counter-pod.yaml Copy admin/logging/two-files-counter-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: counterspec:  containers:  - name: count    image: busybox:1.28    args:    - /bin/sh    - -c    - >
```

It is not recommended to write log entries with different formats to the same log stream, even if you managed to redirect both components to the `stdout` stream of the container. Instead, you can create two sidecar containers. Each sidecar container could tail a particular log file from a shared volume and then redirect the logs to its own `stdout` stream.

Here's a manifest for a pod that has two sidecar containers:

admin/logging/two-files-counter-pod-streaming-sidecar.yaml Copy admin/logging/two-files-counter-pod-streaming-sidecar.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: counterspec:  containers:  - name: count    image: busybox:1.28    args:    - /bin/sh    - -c    - >
```

Now when you run this pod, you can access each log stream separately by running the following commands:

```
kubectl logs counter count-log-1
```

The output is similar to:

```
0: Fri Apr  1 11:42:26 UTC 2022
1: Fri Apr  1 11:42:27 UTC 2022
2: Fri Apr  1 11:42:28 UTC 2022
...
```

```
kubectl logs counter count-log-2
```

The output is similar to:

```
Fri Apr  1 11:42:29 UTC 2022 INFO 0
Fri Apr  1 11:42:30 UTC 2022 INFO 0
Fri Apr  1 11:42:31 UTC 2022 INFO 0
...
```

If you installed a node-level agent in your cluster, that agent picks up those log streams automatically without any further configuration. If you like, you can configure the agent to parse log lines depending on the source container.

Even for Pods that only have low CPU and memory usage (order of a couple of millicores for cpu and order of several megabytes for memory), writing logs to a file and then streaming them to `stdout` can double how much storage you need on the node. If you have an application that writes to a single file, it's recommended to set `/dev/stdout` as the destination rather than implement the streaming sidecar container approach.

Sidecar containers can also be used to rotate log files that cannot be rotated by the application itself. An example of this approach is a small container running `logrotate` periodically. However, it's more straightforward to use `stdout` and `stderr` directly, and leave rotation and retention policies to the kubelet.

Sidecar container with a logging agent

Sidecar container with a logging agent

If the node-level logging agent is not flexible enough for your situation, you can create a sidecar container with a separate logging agent that you have configured specifically to run with your application.

Note:

Using a logging agent in a sidecar container can lead to significant resource consumption. Moreover, you won't be able to access those logs using `kubectl logs` because they are not controlled by the kubelet.

Here are two example manifests that you can use to implement a sidecar container with a logging agent. The first manifest contains a `ConfigMap` to configure fluentd.

admin/logging/fluentd-sidecar-config.yaml Copy admin/logging/fluentd-sidecar-config.yaml to clipboard

```
apiVersion: v1
kind: ConfigMapmetadata:  name: fluentd-configdata:  fluentd.conf: |    <source>      type tail      format none      path /var/lo
```

Note:

In the sample configurations, you can replace fluentd with any logging agent, reading from any source inside an application container.

The second manifest describes a pod that has a sidecar container running fluentd. The pod mounts a volume where fluentd can pick up its configuration data.

admin/logging/two-files-counter-pod-agent-sidecar.yaml Copy admin/logging/two-files-counter-pod-agent-sidecar.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: counterspec:  containers:  - name: count    image: busybox:1.28    args:    - /bin/sh    - -c    - >
```

Exposing logs directly from the application

Exposing logs directly from the application

Cluster-logging that exposes or pushes logs directly from every application is outside the scope of Kubernetes.

What's next

- Read about Kubernetes system logs
- Learn about Traces For Kubernetes System Components
- Learn how to customise the termination message that Kubernetes records when a Pod fails

Swap memory management

Kubernetes can be configured to use swap memory on a node, allowing the kernel to free up physical memory by swapping out pages to backing storage. This is useful for multiple use-cases. For example, nodes running workloads that can benefit from using swap, such as those that have large memory footprints but only access a portion of that memory at any given time. It also helps prevent Pods from being terminated during memory pressure spikes, shields nodes from system-level memory spikes that might compromise its stability, allows for more flexible memory management on the node, and much more.

To learn about configuring swap in your cluster, read Configuring swap memory on Kubernetes nodes.

Operating system support

- Linux nodes support swap; you need to configure each node to enable it. By default, the kubelet will **not** start on a Linux node that has swap enabled.
- Windows nodes require swap space. By default, the kubelet does **not** start on a Windows node that has swap disabled.

How does it work?

There are a number of possible ways that one could envision swap use on a node. If kubelet is already running on a node, it would need to be restarted after swap is provisioned in order to identify it.

When kubelet starts on a node in which swap is provisioned and available (with the `failSwapOn: false` configuration), kubelet will:

- Be able to start on this swap-enabled node.
- Direct the Container Runtime Interface (CRI) implementation, often referred to as the container runtime, to allocate zero swap memory to Kubernetes workloads by default.

Swap configuration on a node is exposed to a cluster admin via the `memorySwap` in the KubeletConfiguration. As a cluster administrator, you can specify the node's behaviour in the presence of swap memory by setting `memorySwap.swapBehavior`.

Swap behaviors

You need to pick a swap behavior to use. Different nodes in your cluster can use different swap behaviors.

The swap behaviors you can choose for Linux nodes are:

`NoSwap` (default)
    Workloads running as Pods on this node do not and cannot use swap.
`LimitedSwap`
    Kubernetes workloads can utilize swap memory.

Note:

If you choose the NoSwap behavior, and you configure the kubelet to tolerate swap space (`failSwapOn: false`), then your workloads don't use any swap.

However, processes outside of Kubernetes-managed containers, such as systemi services (and even the kubelet itself!) **can** utilize swap.

You can read configuring swap memory on Kubernetes nodes to learn about enabling swap for your cluster.

Container runtime integration

The kubelet uses the container runtime API, and directs the container runtime to apply specific configuration (for example, in the cgroup v2 case, `memory.swap.max`) in a manner that will enable the desired swap configuration for a container. For runtimes that use control groups, or cgroups, the container runtime is then responsible for writing these settings to the container-level cgroup.

Observability for swap use

Node and container level metric statistics

Kubelet now collects node and container level metric statistics, which can be accessed at the `/metrics/resource` (which is used mainly by monitoring tools like Prometheus) and `/stats/summary` (which is used mainly by Autoscalers) kubelet HTTP endpoints. This allows clients who can directly request the kubelet to monitor swap usage and remaining swap memory when using `LimitedSwap`. Additionally, a `machine_swap_bytes` metric has been added to cadvisor to show the total physical swap capacity of the machine. See this page for more info.

For example, these `/metrics/resource` are supported:

- `node_swap_usage_bytes`: Current swap usage of the node in bytes.
- `container_swap_usage_bytes`: Current amount of the container swap usage in bytes.
- `container_swap_limit_bytes`: Current amount of the container swap limit in bytes.

Using `kubectl top --show-swap`

Querying metrics is valuable, but somewhat cumbersome, as these metrics are designed to be used by software rather than humans. In order to consume this data in a more user-friendly way, the `kubectl top` command has been extended to support swap metrics, using the `--show-swap` flag.

In order to receive information about swap usage on nodes, `kubectl top nodes --show-swap` can be used:

```
kubectl top nodes --show-swap
```

This will result in an output similar to:

```
NAME    CPU(cores)   CPU(%)   MEMORY(bytes)   MEMORY(%)   SWAP(bytes)    SWAP(%)
node1   1m           10%      2Mi             10%         1Mi            0%
node2   5m           10%      6Mi             10%         2Mi            0%
node3   3m           10%      4Mi             10%         <unknown>      <unknown>
```

In order to receive information about swap usage by pods, `kubectl top nodes --show-swap` can be used:

```
kubectl top pod -n kube-system --show-swap
```

This will result in an output similar to:

```
NAME                                      CPU(cores)   MEMORY(bytes)   SWAP(bytes)
coredns-58d5bc5cdb-5nbk4                  2m           19Mi            0Mi
coredns-58d5bc5cdb-jsh26                  3m           37Mi            0Mi
etcd-node01                               51m          143Mi           5Mi
kube-apiserver-node01                     98m          824Mi           16Mi
kube-controller-manager-node01            20m          135Mi           9Mi
kube-proxy-ffgs2                          1m           24Mi            0Mi
kube-proxy-fhvwx                          1m           39Mi            0Mi
kube-scheduler-node01                     13m          69Mi            0Mi
metrics-server-8598789fdb-d2kcj           5m           26Mi            0Mi
```

Nodes to report swap capacity as part of node status

A new node status field is now added, `node.status.nodeInfo.swap.capacity`, to report the swap capacity of a node.

As an example, the following command can be used to retrieve the swap capacity of the nodes in a cluster:

```
kubectl get nodes -o go-template='{{range .items}}{{.metadata.name}}: {{if .status.nodeInfo.swap.capacity}}{{.status.nodeInfo.swap
```

This will result in an output similar to:

```
node1: 21474836480
node2: 42949664768
node3: <unknown>
```

Note:

The `<unknown>` value indicates that the `.status.nodeInfo.swap.capacity` field is not set for that Node. This probably means that the node does not have swap provisioned, or less likely, that the kubelet is not able to determine the swap capacity of the node.

Swap discovery using Node Feature Discovery (NFD)

Node Feature Discovery is a Kubernetes addon for detecting hardware features and configuration. It can be utilized to discover which nodes are provisioned with swap.

As an example, to figure out which nodes are provisioned with swap, use the following command:

```
kubectl get nodes -o jsonpath='{range .items[?(@.metadata.labels.feature\.node\.kubernetes\.io/memory-swap)]}{.metadata.name}{"\t"
```

This will result in an output similar to:

```
k8s-worker1: true
k8s-worker2: true
k8s-worker3: false
```

In this example, swap is provisioned on nodes `k8s-worker1` and `k8s-worker2`, but not on `k8s-worker3`.

Risks and caveats

Caution:

It is deeply encouraged to encrypt the swap space. See Memory-backed volumes memory-backed volumes for more info.

Having swap available on a system reduces predictability. While swap can enhance performance by making more RAM available, swapping data back to memory is a heavy operation, sometimes slower by many orders of magnitude, which can cause unexpected performance regressions. Furthermore, swap changes a system's behaviour under memory pressure. Enabling swap increases the risk of noisy neighbors, where Pods that frequently use their RAM may cause other Pods to swap. In addition, since swap allows for greater memory usage for workloads in Kubernetes that cannot be predictably accounted for, and due to unexpected packing configurations, the scheduler currently does not account for swap memory usage. This heightens the risk of noisy neighbors.

The performance of a node with swap memory enabled depends on the underlying physical storage. When swap memory is in use, performance will be significantly worse in an I/O operations per second (IOPS) constrained environment, such as a cloud VM with I/O throttling, when compared to faster storage mediums like solid-state drives or NVMe. As swap might cause IO pressure, it is recommended to give a higher IO latency priority to system critical daemons. See the relevant section in the recommended practices section below.

Memory-backed volumes

On Linux nodes, memory-backed volumes (such as secret volume mounts, or emptyDir with medium: Memory) are implemented with a tmpfs filesystem. The contents of such volumes should remain in memory at all times, hence should not be swapped to disk. To ensure the contents of such volumes remain in memory, the noswap tmpfs option is being used.

The Linux kernel officially supports the noswap option from version 6.3 (more info can be found in Linux Kernel Version Requirements). However, the different distributions often choose to backport this mount option to older Linux versions as well.

In order to verify whether the node supports the noswap option, the kubelet will do the following:

- If the kernel's version is above 6.3 then the noswap option will be assumed to be supported.
- Otherwise, kubelet would try to mount a dummy tmpfs with the noswap option at startup. If kubelet fails with an error indicating of an unknown option, noswap will be assumed to not be supported, hence will not be used. A kubelet log entry will be emitted to warn the user about memory-backed volumes might swap to disk. If kubelet succeeds, the dummy tmpfs will be deleted and the noswap option will be used.
  - If the noswap option is not supported, kubelet will emit a warning log entry, then continue its execution.

See the section above with an example for setting unencrypted swap. However, handling encrypted swap is not within the scope of kubelet; rather, it is a general OS configuration concern and should be addressed at that level. It is the administrator's responsibility to provision encrypted swap to mitigate this risk.

Evictions

Configuring memory eviction thresholds for swap-enabled nodes can be tricky.

With swap being disabled, it is reasonable to configure kubelet's eviction thresholds to be a bit lower than the node's memory capacity. The rationale is that we want Kubernetes to start evicting Pods before the node runs out of memory and invokes the Out Of Memory (OOM) killer, since the OOM killer is not Kubernetes-aware, therefore does not consider things like QoS, pod priority, or other Kubernetes-specific factors.

With swap enabled, the situation is more complex. In Linux, the vm.min_free_kbytes parameter defines the memory threshold for the kernel to start aggressively reclaiming memory, which includes swapping out pages. If the kubelet's eviction thresholds are set in a way that eviction would take place before the kernel starts reclaiming memory, it could lead to workloads never being able to swap out during node memory pressure. However, setting the eviction thresholds too high could result in the node running out of memory and invoking the OOM killer, which is not ideal either.

To address this, it is recommended to set the kubelet's eviction thresholds to be slightly lower than the vm.min_free_kbytes value. This way, the node can start swapping before kubelet would start evicting Pods, allowing workloads to swap out unused data and preventing evictions from happening. On the other hand, since it is just slightly lower, kubelet is likely to start evicting Pods before the node runs out of memory, thus avoiding the OOM killer.

The value of vm.min_free_kbytes can be determined by running the following command on the node:

```
cat /proc/sys/vm/min_free_kbytes
```

Unutilized swap space

Under the LimitedSwap behavior, the amount of swap available to a Pod is determined automatically, based on the proportion of the memory requested relative to the node's total memory (For more details, see the section below).

This design means that usually there would be some portion of swap that will remain restricted for Kubernetes workloads. For example, since Kubernetes 1.34 does not permit swap use for Pods in the Guaranteed QoS class, the amount of swap that's proportional to the memory request for Guaranteed pods would remain unused by Kubernetes workloads.

This behavior carries some risk in a situation where many pods are not eligible for swapping. On the other hand, it effectively keeps some system-reserved amount of swap memory that can be used by processes outside of Kubernetes' scope, such as system daemons and even kubelet itself.

Good practice for using swap in a Kubernetes cluster

Disable swap for system-critical daemons

During the testing phase and based on user feedback, it was observed that the performance of system-critical daemons and services might degrade. This implies that system daemons, including the kubelet, could operate slower than usual. If this issue is encountered, it is advisable to configure the cgroup of the system slice to prevent swapping (i.e., set memory.swap.max=0).

Protect system-critical daemons for I/O latency

Swap can increase the I/O load on a node. When memory pressure causes the kernel to rapidly swap pages in and out, system-critical daemons and services that rely on I/O operations may experience performance degradation.

To mitigate this, it is recommended for systemd users to prioritize the system slice in terms of I/O latency. For non-systemd users, setting up a dedicated cgroup for system daemons and processes and prioritizing I/O latency in the same way is advised. This can be achieved by setting io.latency for the system slice, thereby granting it higher I/O priority. See cgroup's documentation for more info.

Swap and control plane nodes

The Kubernetes project recommends running control plane nodes without any swap space configured. The control plane primarily hosts Guaranteed QoS Pods, so swap can generally be disabled. The main concern is that swapping critical services on the control plane could negatively impact performance.

Use of a dedicated disk for swap

The Kubernetes project recommends using encrypted swap, whenever you run nodes with swap enabled. If swap resides on a partition or the root filesystem, workloads may interfere with system processes that need to write to disk. When they share the same disk, processes can overwhelm swap, disrupting the I/O of kubelet, container runtime, and systemd, which would impact other workloads. Since swap space is located on a disk, it is crucial to ensure the disk is fast enough for the intended use cases. Alternatively, one can configure I/O priorities between different mapped areas of a single backing device.

Swap-aware scheduling

Kubernetes 1.34 does not support allocating Pods to nodes in a way that accounts for swap memory usage. The scheduler typically uses *requests* for infrastructure resources to guide Pod placement, and Pods do not request swap space; they just request `memory`. This means that the scheduler does not consider swap memory when making scheduling decisions. While this is something we are actively working on, it is not yet implemented.

In order for administrators to ensure that Pods are not scheduled on nodes with swap memory unless they are specifically intended to use it, Administrators can taint nodes with swap available to protect against this problem. Taints will ensure that workloads which tolerate swap will not spill onto nodes without swap under load.

Selecting storage for optimal performance

The storage device designated for swap space is critical to maintaining system responsiveness during high memory usage. Rotational hard disk drives (HDDs) are ill-suited for this task as their mechanical nature introduces significant latency, leading to severe performance degradation and system thrashing. For modern performance needs, a device such as a Solid State Drive (SSD) is probably the appropriate choice for swap, as its low-latency electronic access minimizes the slowdown.

Swap behavior details

How is the swap limit being determined with LimitedSwap?

The configuration of swap memory, including its limitations, presents a significant challenge. Not only is it prone to misconfiguration, but as a system-level property, any misconfiguration could potentially compromise the entire node rather than just a specific workload. To mitigate this risk and ensure the health of the node, we have implemented Swap with automatic configuration of limitations.

With `LimitedSwap`, Pods that do not fall under the Burstable QoS classification (i.e. `BestEffort`/`Guaranteed` QoS Pods) are prohibited from utilizing swap memory. `BestEffort` QoS Pods exhibit unpredictable memory consumption patterns and lack information regarding their memory usage, making it difficult to determine a safe allocation of swap memory. Conversely, `Guaranteed` QoS Pods are typically employed for applications that rely on the precise allocation of resources specified by the workload, with memory being immediately available. To maintain the aforementioned security and node health guarantees, these Pods are not permitted to use swap memory when `LimitedSwap` is in effect. In addition, high-priority pods are not permitted to use swap in order to ensure the memory they consume always residents on disk, hence ready to use.

Prior to detailing the calculation of the swap limit, it is necessary to define the following terms:

- `nodeTotalMemory`: The total amount of physical memory available on the node.
- `totalPodsSwapAvailable`: The total amount of swap memory on the node that is available for use by Pods (some swap memory may be reserved for system use).
- `containerMemoryRequest`: The container's memory request.

Swap limitation is configured as:
(`containerMemoryRequest` / `nodeTotalMemory`) × `totalPodsSwapAvailable`

In other words, the amount of swap that a container is able to use is proportionate to its memory request, the node's total physical memory and the total amount of swap memory on the node that is available for use by Pods.

It is important to note that, for containers within Burstable QoS Pods, it is possible to opt-out of swap usage by specifying memory requests that are equal to memory limits. Containers configured in this manner will not have access to swap memory.

What's next

- To learn about managing swap on Linux nodes, read configuring swap memory on Kubernetes nodes.
- You can check out a blog post about Kubernetes and swap
- For background information, please see the original KEP, KEP-2400, and its design.

Traces For Kubernetes System Components

FEATURE STATE: `Kubernetes v1.27 [beta]`

System component traces record the latency of and relationships between operations in the cluster.

Kubernetes components emit traces using the OpenTelemetry Protocol with the gRPC exporter and can be collected and routed to tracing backends using an OpenTelemetry Collector.

Trace Collection

Kubernetes components have built-in gRPC exporters for OTLP to export traces, either with an OpenTelemetry Collector, or without an OpenTelemetry Collector.

For a complete guide to collecting traces and using the collector, see Getting Started with the OpenTelemetry Collector. However, there are a few things to note that are specific to Kubernetes components.

By default, Kubernetes components export traces using the grpc exporter for OTLP on the IANA OpenTelemetry port, 4317. As an example, if the collector is running as a sidecar to a Kubernetes component, the following receiver configuration will collect spans and log them to standard output:

```
receivers:
  otlp:
    protocols:
      grpc:
```

```
exporters:  # Replace this exporter with the exporter for your backend  exporters:    debug:      verbosity: detailedservice:  pip
```

To directly emit traces to a backend without utilizing a collector, specify the endpoint field in the Kubernetes tracing configuration file with the desired trace backend address. This method negates the need for a collector and simplifies the overall structure.

For trace backend header configuration, including authentication details, environment variables can be used with `OTEL_EXPORTER_OTLP_HEADERS`, see [OTLP Exporter Configuration]().

Additionally, for trace resource attribute configuration such as Kubernetes cluster name, namespace, Pod name, etc., environment variables can also be used with `OTEL_RESOURCE_ATTRIBUTES`, see [OTLP Kubernetes Resource]().

Component traces

kube-apiserver traces

The kube-apiserver generates spans for incoming HTTP requests, and for outgoing requests to webhooks, etcd, and re-entrant requests. It propagates the [W3C Trace Context]() with outgoing requests but does not make use of the trace context attached to incoming requests, as the kube-apiserver is often a public endpoint.

Enabling tracing in the kube-apiserver

To enable tracing, provide the kube-apiserver with a tracing configuration file with `--tracing-config-file=<path-to-config>`. This is an example config that records spans for 1 in 10000 requests, and uses the default OpenTelemetry endpoint:

```
apiVersion: apiserver.config.k8s.io/v1
kind: TracingConfiguration# default value#endpoint: localhost:4317samplingRatePerMillion: 100
```

For more information about the `TracingConfiguration` struct, see [API server config API (v1)]().

kubelet traces

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

The kubelet CRI interface and authenticated http servers are instrumented to generate trace spans. As with the apiserver, the endpoint and sampling rate are configurable. Trace context propagation is also configured. A parent span's sampling decision is always respected. A provided tracing configuration sampling rate will apply to spans without a parent. Enabled without a configured endpoint, the default OpenTelemetry Collector receiver address of "localhost:4317" is set.

Enabling tracing in the kubelet

To enable tracing, apply the [tracing configuration](). This is an example snippet of a kubelet config that records spans for 1 in 10000 requests, and uses the default OpenTelemetry endpoint:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationtracing:  # default value  #endpoint: localhost:4317  samplingRatePerMillion: 100
```

If the `samplingRatePerMillion` is set to one million (`1000000`), then every span will be sent to the exporter.

The kubelet in Kubernetes v1.34 collects spans from the garbage collection, pod synchronization routine as well as every gRPC method. The kubelet propagates trace context with gRPC requests so that container runtimes with trace instrumentation, such as CRI-O and containerd, can associate their exported spans with the trace context from the kubelet. The resulting traces will have parent-child links between kubelet and container runtime spans, providing helpful context when debugging node issues.

Please note that exporting spans always comes with a small performance overhead on the networking and CPU side, depending on the overall configuration of the system. If there is any issue like that in a cluster which is running with tracing enabled, then mitigate the problem by either reducing the `samplingRatePerMillion` or disabling tracing completely by removing the configuration.

Stability

Tracing instrumentation is still under active development, and may change in a variety of ways. This includes span names, attached attributes, instrumented endpoints, etc. Until this feature graduates to stable, there are no guarantees of backwards compatibility for tracing instrumentation.

What's next

- Read about [Getting Started with the OpenTelemetry Collector]()
- Read about [OTLP Exporter Configuration]()

Good practices for Dynamic Resource Allocation as a Cluster Admin

This page describes good practices when configuring a Kubernetes cluster utilizing Dynamic Resource Allocation (DRA). These instructions are for cluster administrators.

Separate permissions to DRA related APIs

DRA is orchestrated through a number of different APIs. Use authorization tools (like RBAC, or another solution) to control access to the right APIs depending on the persona of your user.

In general, DeviceClasses and ResourceSlices should be restricted to admins and the DRA drivers. Cluster operators that will be deploying Pods with claims will need access to ResourceClaim and ResourceClaimTemplate APIs; both of these APIs are namespace scoped.

DRA driver deployment and maintenance

DRA drivers are third-party applications that run on each node of your cluster to interface with the hardware of that node and Kubernetes' native DRA components. The installation procedure depends on the driver you choose, but is likely deployed as a DaemonSet to all or a selection of the nodes (using node selectors or similar mechanisms) in your cluster.

Use drivers with seamless upgrade if available

DRA drivers implement the `kubeletplugin` package interface. Your driver may support *seamless upgrades* by implementing a property of this interface that allows two versions of the same DRA driver to coexist for a short time. This is only available for kubelet versions 1.33 and above and may not be supported by your driver for heterogeneous clusters with attached nodes running older versions of Kubernetes - check your driver's documentation to be sure.

If seamless upgrades are available for your situation, consider using it to minimize scheduling delays when your driver updates.

If you cannot use seamless upgrades, during driver downtime for upgrades you may observe that:

- Pods cannot start unless the claims they depend on were already prepared for use.
- Cleanup after the last pod which used a claim gets delayed until the driver is available again. The pod is not marked as terminated. This prevents reusing the resources used by the pod for other pods.
- Running pods will continue to run.

Confirm your DRA driver exposes a liveness probe and utilize it

Your DRA driver likely implements a gRPC socket for healthchecks as part of DRA driver good practices. The easiest way to utilize this grpc socket is to configure it as a liveness probe for the DaemonSet deploying your DRA driver. Your driver's documentation or deployment tooling may already include this, but if you are building your configuration separately or not running your DRA driver as a Kubernetes pod, be sure that your orchestration tooling restarts the DRA driver on failed healthchecks to this grpc socket. Doing so will minimize any accidental downtime of the DRA driver and give it more opportunities to self heal, reducing scheduling delays or troubleshooting time.

When draining a node, drain the DRA driver as late as possible

The DRA driver is responsible for unpreparing any devices that were allocated to Pods, and if the DRA driver is drained before Pods with claims have been deleted, it will not be able to finalize its cleanup. If you implement custom drain logic for nodes, consider checking that there are no allocated/reserved ResourceClaim or ResourceClaimTemplates before terminating the DRA driver itself.

Monitor and tune components for higher load, especially in high scale environments

Control plane component kube-scheduler and the internal ResourceClaim controller orchestrated by the component kube-controller-manager do the heavy lifting during scheduling of Pods with claims based on metadata stored in the DRA APIs. Compared to non-DRA scheduled Pods, the number of API server calls, memory, and CPU utilization needed by these components is increased for Pods using DRA claims. In addition, node local components like the DRA driver and kubelet utilize DRA APIs to allocated the hardware request at Pod sandbox creation time. Especially in high scale environments where clusters have many nodes, and/or deploy many workloads that heavily utilize DRA defined resource claims, the cluster administrator should configure the relevant components to anticipate the increased load.

The effects of mistuned components can have direct or snowballing affects causing different symptoms during the Pod lifecycle. If the `kube-scheduler` component's QPS and burst configurations are too low, the scheduler might quickly identify a suitable node for a Pod but take longer to bind the Pod to that node. With DRA, during Pod scheduling, the QPS and Burst parameters in the client-go configuration within `kube-controller-manager` are critical.

The specific values to tune your cluster to depend on a variety of factors like number of nodes/pods, rate of pod creation, churn, even in non-DRA environments; see the SIG Scalability README on Kubernetes scalability thresholds for more information. In scale tests performed against a DRA enabled cluster with 100 nodes, involving 720 long-lived pods (90% saturation) and 80 churn pods (10% churn, 10 times), with a job creation QPS of 10, `kube-controller-manager` QPS could be set to as low as 75 and Burst to 150 to meet equivalent metric targets for non-DRA deployments. At this lower bound, it was observed that the client side rate limiter was triggered enough to protect the API server from explosive burst but was high enough that pod startup SLOs were not impacted. While this is a good starting point, you can get a better idea of how to tune the different components that have the biggest effect on DRA performance for your deployment by monitoring the following metrics. For more information on all the stable metrics in Kubernetes, see the Kubernetes Metrics Reference.

`kube-controller-manager` metrics

The following metrics look closely at the internal ResourceClaim controller managed by the `kube-controller-manager` component.

- Workqueue Add Rate: Monitor `sum(rate(workqueue_adds_total{name="resource_claim"}[5m]))` to gauge how quickly items are added to the ResourceClaim controller.
- Workqueue Depth: Track `sum(workqueue_depth{endpoint="kube-controller-manager", name="resource_claim"})` to identify any backlogs in the ResourceClaim controller.
- Workqueue Work Duration: Observe `histogram_quantile(0.99, sum(rate(workqueue_work_duration_seconds_bucket{name="resource_claim"}[5m])) by (le))` to understand the speed at which the ResourceClaim controller processes work.

If you are experiencing low Workqueue Add Rate, high Workqueue Depth, and/or high Workqueue Work Duration, this suggests the controller isn't performing optimally. Consider tuning parameters like QPS, burst, and CPU/memory configurations.

If you are experiencing high Workeqeue Add Rate, high Workqueue Depth, but reasonable Workqueue Work Duration, this indicates the controller is processing work, but concurrency might be insufficient. Concurrency is hardcoded in the controller, so as a cluster administrator, you can tune for this by reducing the pod creation QPS, so the add rate to the resource claim workqueue is more manageable.

`kube-scheduler` metrics

The following scheduler metrics are high level metrics aggregating performance across all Pods scheduled, not just those using DRA. It is important to note that the end-to-end metrics are ultimately influenced by the `kube-controller-manager`'s performance in creating ResourceClaims from ResourceClainTemplates in deployments that heavily use ResourceClainTemplates.

- Scheduler End-to-End Duration: Monitor `histogram_quantile(0.99, sum(increase(scheduler_pod_scheduling_sli_duration_seconds_bucket[5m])) by (le))`.
- Scheduler Algorithm Latency: Track `histogram_quantile(0.99, sum(increase(scheduler_scheduling_algorithm_duration_seconds_bucket[5m])) by (le))`.

`kubelet` metrics

When a Pod bound to a node must have a ResourceClaim satisfied, kubelet calls the `NodePrepareResources` and `NodeUnprepareResources` methods of the DRA driver. You can observe this behavior from the kubelet's point of view with the following metrics.

- Kubelet NodePrepareResources: Monitor `histogram_quantile(0.99, sum(rate(dra_operations_duration_seconds_bucket{operation_name="PrepareResources"}[5m])) by (le))`.
- Kubelet NodeUnprepareResources: Track `histogram_quantile(0.99, sum(rate(dra_operations_duration_seconds_bucket{operation_name="UnprepareResources"}[5m])) by (le))`.

DRA kubeletplugin operations

DRA drivers implement the `kubeletplugin` package interface which surfaces its own metric for the underlying gRPC operation `NodePrepareResources` and `NodeUnprepareResources`. You can observe this behavior from the point of view of the internal kubeletplugin with the following metrics.

- DRA kubeletplugin gRPC NodePrepareResources operation: Observe `histogram_quantile(0.99, sum(rate(dra_grpc_operations_duration_seconds_bucket{method_name=~".*NodePrepareResources"}[5m])) by (le))`.
- DRA kubeletplugin gRPC NodeUnprepareResources operation: Observe `histogram_quantile(0.99, sum(rate(dra_grpc_operations_duration_seconds_bucket{method_name=~".*NodeUnprepareResources"}[5m])) by (le))`.

What's next

- Learn more about DRA
- Read the Kubernetes Metrics Reference

System Logs

System component logs record events happening in cluster, which can be very useful for debugging. You can configure log verbosity to see more or less detail. Logs can be as coarse-grained as showing errors within a component, or as fine-grained as showing step-by-step traces of events (like HTTP access logs, pod state changes, controller actions, or scheduler decisions).

Warning:

In contrast to the command line flags described here, the *log output* itself does *not* fall under the Kubernetes API stability guarantees: individual log entries and their formatting may change from one release to the next!

Klog

klog is the Kubernetes logging library. klog generates log messages for the Kubernetes system components.

Kubernetes is in the process of simplifying logging in its components. The following klog command line flags are deprecated starting with Kubernetes v1.23 and removed in Kubernetes v1.26:

- `--add-dir-header`
- `--alsologtostderr`
- `--log-backtrace-at`
- `--log-dir`
- `--log-file`
- `--log-file-max-size`
- `--logtostderr`
- `--one-output`
- `--skip-headers`
- `--skip-log-headers`
- `--stderrthreshold`

Output will always be written to stderr, regardless of the output format. Output redirection is expected to be handled by the component which invokes a Kubernetes component. This can be a POSIX shell or a tool like systemd.

In some cases, for example a distroless container or a Windows system service, those options are not available. Then the `kube-log-runner` binary can be used as wrapper around a Kubernetes component to redirect output. A prebuilt binary is included in several Kubernetes base images under its traditional name as `/go-runner` and as `kube-log-runner` in server and node release archives.

This table shows how `kube-log-runner` invocations correspond to shell redirection:

Usage	POSIX shell (such as bash)	`kube-log-runner <options> <cmd>`
Merge stderr and stdout, write to stdout	`2>&1`	`kube-log-runner` (default behavior)

Usage	POSIX shell (such as bash)	`kube-log-runner <options> <cmd>`
Redirect both into log file	`1>>/tmp/log 2>&1`	`kube-log-runner -log-file=/tmp/log`
Copy into log file and to stdout	`2>&1 | tee -a /tmp/log`	`kube-log-runner -log-file=/tmp/log -also-stdout`
Redirect only stdout into log file	`>/tmp/log`	`kube-log-runner -log-file=/tmp/log -redirect-stderr=false`

Klog output

An example of the traditional klog native format:

```
I1025 00:15:15.525108       1 httplog.go:79] GET /api/v1/namespaces/kube-system/pods/metrics-server-v0.3.1-57c75779f-9p8wg: (1.512
```

The message string may contain line breaks:

```
I1025 00:15:15.525108       1 example.go:79] This is a message
which has a line break.
```

Structured Logging

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

Warning:

Migration to structured log messages is an ongoing process. Not all log messages are structured in this version. When parsing log files, you must also handle unstructured log messages.

Log formatting and value serialization are subject to change.

Structured logging introduces a uniform structure in log messages allowing for programmatic extraction of information. You can store and process structured logs with less effort and cost. The code which generates a log message determines whether it uses the traditional unstructured klog output or structured logging.

The default formatting of structured log messages is as text, with a format that is backward compatible with traditional klog:

```
<klog header> "<message>" <key1>="<value1>" <key2>="<value2>" ...
```

Example:

```
I1025 00:15:15.525108       1 controller_utils.go:116] "Pod status updated" pod="kube-system/kubedns" status="ready"
```

Strings are quoted. Other values are formatted with `%+v`, which may cause log messages to continue on the next line depending on the data.

```
I1025 00:15:15.525108       1 example.go:116] "Example" data="This is text with a line break\nand \"quotation marks\"." someInt=1
second line.}
```

Contextual Logging

FEATURE STATE: `Kubernetes v1.30 [beta]`

Contextual logging builds on top of structured logging. It is primarily about how developers use logging calls: code based on that concept is more flexible and supports additional use cases as described in the Contextual Logging KEP.

If developers use additional functions like `WithValues` or `WithName` in their components, then log entries contain additional information that gets passed into functions by their caller.

For Kubernetes 1.34, this is gated behind the `ContextualLogging` feature gate and is enabled by default. The infrastructure for this was added in 1.24 without modifying components. The `component-base/logs/example` command demonstrates how to use the new logging calls and how a component behaves that supports contextual logging.

```
$ cd $GOPATH/src/k8s.io/kubernetes/staging/src/k8s.io/component-base/logs/example/cmd/
$ go run . --help
...
      --feature-gates mapStringBool  A set of key=value pairs that describe feature gates for alpha/experimental features. Options
                                     AllAlpha=true|false (ALPHA - default=false)
                                     AllBeta=true|false (BETA - default=false)
                                     ContextualLogging=true|false (BETA - default=true)
$ go run . --feature-gates ContextualLogging=true...I0222 15:13:31.645988  197901 example.go:54] "runtime" logger="example.myname"
```

The `logger` key and `foo="bar"` were added by the caller of the function which logs the `runtime` message and `duration="1m0s"` value, without having to modify that function.

With contextual logging disable, `WithValues` and `WithName` do nothing and log calls go through the global klog logger. Therefore this additional information is not in the log output anymore:

```
$ go run . --feature-gates ContextualLogging=false
...
I0222 15:14:40.497333  198174 example.go:54] "runtime" duration="1m0s"
I0222 15:14:40.497346  198174 example.go:55] "another runtime" duration="1h0m0s" duration="1m0s"
```

JSON log format

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

Warning:

JSON output does not support many standard klog flags. For list of unsupported klog flags, see the Command line tool reference.

Not all logs are guaranteed to be written in JSON format (for example, during process start). If you intend to parse logs, make sure you can handle log lines that are not JSON as well.

Field names and JSON serialization are subject to change.

The `--logging-format=json` flag changes the format of logs from klog native format to JSON format. Example of JSON log format (pretty printed):

```
{
   "ts": 1580306777.04728,
   "v": 4,
   "msg": "Pod status updated",
   "pod":{
      "name": "nginx-1",
      "namespace": "default"
   },
   "status": "ready"
}
```

Keys with special meaning:

- `ts` - timestamp as Unix time (required, float)
- `v` - verbosity (only for info and not for error messages, int)
- `err` - error string (optional, string)
- `msg` - message (required, string)

List of components currently supporting JSON format:

- kube-controller-manager
- kube-apiserver
- kube-scheduler
- kubelet

Log verbosity level

The `-v` flag controls log verbosity. Increasing the value increases the number of logged events. Decreasing the value decreases the number of logged events. Increasing verbosity settings logs increasingly less severe events. A verbosity setting of 0 logs only critical events.

Log location

There are two types of system components: those that run in a container and those that do not run in a container. For example:

- The Kubernetes scheduler and kube-proxy run in a container.
- The kubelet and container runtime do not run in containers.

On machines with systemd, the kubelet and container runtime write to journald. Otherwise, they write to `.log` files in the `/var/log` directory. System components inside containers always write to `.log` files in the `/var/log` directory, bypassing the default logging mechanism. Similar to the container logs, you should rotate system component logs in the `/var/log` directory. In Kubernetes clusters created by the `kube-up.sh` script, log rotation is configured by the `logrotate` tool. The `logrotate` tool rotates logs daily, or once the log size is greater than 100MB.

Log query

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

To help with debugging issues on nodes, Kubernetes v1.27 introduced a feature that allows viewing logs of services running on the node. To use the feature, ensure that the `NodeLogQuery` feature gate is enabled for that node, and that the kubelet configuration options `enableSystemLogHandler` and `enableSystemLogQuery` are both set to true. On Linux the assumption is that service logs are available via journald. On Windows the assumption is that service logs are available in the application log provider. On both operating systems, logs are also available by reading files within `/var/log/`.

Provided you are authorized to interact with node objects, you can try out this feature on all your nodes or just a subset. Here is an example to retrieve the kubelet service logs from a node:

```
# Fetch kubelet logs from a node named node-1.example
kubectl get --raw "/api/v1/nodes/node-1.example/proxy/logs/?query=kubelet"
```

You can also fetch files, provided that the files are in a directory that the kubelet allows for log fetches. For example, you can fetch a log from `/var/log` on a Linux node:

```
kubectl get --raw "/api/v1/nodes/<insert-node-name-here>/proxy/logs/?query=/<insert-log-file-name-here>"
```

The kubelet uses heuristics to retrieve logs. This helps if you are not aware whether a given system service is writing logs to the operating system's native logger like journald or to a log file in `/var/log/`. The heuristics first checks the native logger and if that is not available attempts to retrieve the first logs from `/var/log/<servicename>` or `/var/log/<servicename>.log` or `/var/log/<servicename>/<servicename>.log`.

The complete list of options that can be used are:

Option	Description
`boot`	boot show messages from a specific system boot
`pattern`	pattern filters log entries by the provided PERL-compatible regular expression
`query`	query specifies services(s) or files from which to return logs (required)
`sinceTime`	an RFC3339 timestamp from which to show logs (inclusive)

Option	Description
`untilTime`	an RFC3339 timestamp until which to show logs (inclusive)
`tailLines`	specify how many lines from the end of the log to retrieve; the default is to fetch the whole log

Example of a more complex query:

```
# Fetch kubelet logs from a node named node-1.example that have the word "error"
kubectl get --raw "/api/v1/nodes/node-1.example/proxy/logs/?query=kubelet&pattern=error"
```

What's next

- Read about the Kubernetes Logging Architecture
- Read about Structured Logging
- Read about Contextual Logging
- Read about deprecation of klog flags
- Read about the Conventions for logging severity
- Read about Log Query

Resource Management for Windows nodes

This page outlines the differences in how resources are managed between Linux and Windows.

On Linux nodes, cgroups are used as a pod boundary for resource control. Containers are created within that boundary for network, process and file system isolation. The Linux cgroup APIs can be used to gather CPU, I/O, and memory use statistics.

In contrast, Windows uses a *job object* per container with a system namespace filter to contain all processes in a container and provide logical isolation from the host. (Job objects are a Windows process isolation mechanism and are different from what Kubernetes refers to as a Job).

There is no way to run a Windows container without the namespace filtering in place. This means that system privileges cannot be asserted in the context of the host, and thus privileged containers are not available on Windows. Containers cannot assume an identity from the host because the Security Account Manager (SAM) is separate.

Memory management

Windows does not have an out-of-memory process killer as Linux does. Windows always treats all user-mode memory allocations as virtual, and pagefiles are mandatory.

Windows nodes do not overcommit memory for processes. The net effect is that Windows won't reach out of memory conditions the same way Linux does, and processes page to disk instead of being subject to out of memory (OOM) termination. If memory is over-provisioned and all physical memory is exhausted, then paging can slow down performance.

CPU management

Windows can limit the amount of CPU time allocated for different processes but cannot guarantee a minimum amount of CPU time.

On Windows, the kubelet supports a command-line flag to set the scheduling priority of the kubelet process: `--windows-priorityclass`. This flag allows the kubelet process to get more CPU time slices when compared to other processes running on the Windows host. More information on the allowable values and their meaning is available at Windows Priority Classes. To ensure that running Pods do not starve the kubelet of CPU cycles, set this flag to `ABOVE_NORMAL_PRIORITY_CLASS` or above.

Resource reservation

To account for memory and CPU used by the operating system, the container runtime, and by Kubernetes host processes such as the kubelet, you can (and should) reserve memory and CPU resources with the `--kube-reserved` and/or `--system-reserved` kubelet flags. On Windows these values are only used to calculate the node's allocatable resources.

Caution:

As you deploy workloads, set resource memory and CPU limits on containers. This also subtracts from `NodeAllocatable` and helps the cluster-wide scheduler in determining which pods to place on which nodes.

Scheduling pods without limits may over-provision the Windows nodes and in extreme cases can cause the nodes to become unhealthy.

On Windows, a good practice is to reserve at least 2GiB of memory.

To determine how much CPU to reserve, identify the maximum pod density for each node and monitor the CPU usage of the system services running there, then choose a value that meets your workload needs.

Images

A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well-defined assumptions about their runtime environment.

You typically create a container image of your application and push it to a registry before referring to it in a Pod.

This page provides an outline of the container image concept.

Note:

If you are looking for the container images for a Kubernetes release (such as v1.34, the latest minor release), visit Download Kubernetes.

Image names

Container images are usually given a name such as `pause`, `example/mycontainer`, or `kube-apiserver`. Images can also include a registry hostname; for example: `fictional.registry.example/imagename`, and possibly a port number as well; for example: `fictional.registry.example:10443/imagename`.

If you don't specify a registry hostname, Kubernetes assumes that you mean the Docker public registry. You can change this behavior by setting a default image registry in the container runtime configuration.

After the image name part you can add a *tag* or *digest* (in the same way you would when using with commands like `docker` or `podman`). Tags let you identify different versions of the same series of images. Digests are a unique identifier for a specific version of an image. Digests are hashes of the image's content, and are immutable. Tags can be moved to point to different images, but digests are fixed.

Image tags consist of lowercase and uppercase letters, digits, underscores (_), periods (`.`), and dashes (`-`). A tag can be up to 128 characters long, and must conform to the following regex pattern: `[a-zA-Z0-9_][a-zA-Z0-9._-]{0,127}`. You can read more about it and find the validation regex in the OCI Distribution Specification. If you don't specify a tag, Kubernetes assumes you mean the tag `latest`.

Image digests consists of a hash algorithm (such as `sha256`) and a hash value. For example: `sha256:1ff6c18fbef2045af6b9c16bf034cc421a29027b800e4f9b68ae9b1cb3e9ae07`. You can find more information about the digest format in the OCI Image Specification.

Some image name examples that Kubernetes can use are:

- `busybox` — Image name only, no tag or digest. Kubernetes will use the Docker public registry and latest tag. Equivalent to `docker.io/library/busybox:latest`.
- `busybox:1.32.0` — Image name with tag. Kubernetes will use the Docker public registry. Equivalent to `docker.io/library/busybox:1.32.0`.
- `registry.k8s.io/pause:latest` — Image name with a custom registry and latest tag.
- `registry.k8s.io/pause:3.5` — Image name with a custom registry and non-latest tag.
- `registry.k8s.io/pause@sha256:1ff6c18fbef2045af6b9c16bf034cc421a29027b800e4f9b68ae9b1cb3e9ae07` — Image name with digest.
- `registry.k8s.io/pause:3.5@sha256:1ff6c18fbef2045af6b9c16bf034cc421a29027b800e4f9b68ae9b1cb3e9ae07` — Image name with tag and digest. Only the digest will be used for pulling.

Updating images

When you first create a Deployment, StatefulSet, Pod, or other object that includes a PodTemplate, and a pull policy was not explicitly specified, then by default the pull policy of all containers in that Pod will be set to `IfNotPresent`. This policy causes the kubelet to skip pulling an image if it already exists.

Image pull policy

The `imagePullPolicy` for a container and the tag of the image both affect *when* the kubelet attempts to pull (download) the specified image.

Here's a list of the values you can set for `imagePullPolicy` and the effects these values have:

`IfNotPresent`
: the image is pulled only if it is not already present locally.

`Always`
: every time the kubelet launches a container, the kubelet queries the container image registry to resolve the name to an image digest. If the kubelet has a container image with that exact digest cached locally, the kubelet uses its cached image; otherwise, the kubelet pulls the image with the resolved digest, and uses that image to launch the container.

`Never`
: the kubelet does not try fetching the image. If the image is somehow already present locally, the kubelet attempts to start the container; otherwise, startup fails. See pre-pulled images for more details.

The caching semantics of the underlying image provider make even `imagePullPolicy: Always` efficient, as long as the registry is reliably accessible. Your container runtime can notice that the image layers already exist on the node so that they don't need to be downloaded again.

Note:

You should avoid using the `:latest` tag when deploying containers in production as it is harder to track which version of the image is running and more difficult to roll back properly.

Instead, specify a meaningful tag such as `v1.42.0` and/or a digest.

To make sure the Pod always uses the same version of a container image, you can specify the image's digest; replace `<image-name>:<tag>` with `<image-name>@<digest>` (for example, `image@sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2`).

When using image tags, if the image registry were to change the code that the tag on that image represents, you might end up with a mix of Pods running the old and new code. An image digest uniquely identifies a specific version of the image, so Kubernetes runs the same code every time it starts a container with that image name and digest specified. Specifying an image by digest pins the code that you run so that a change at the registry cannot lead to that mix of versions.

There are third-party admission controllers that mutate Pods (and PodTemplates) when they are created, so that the running workload is defined based on an image digest rather than a tag. That might be useful if you want to make sure that your entire workload is running the same code no matter what tag changes happen at the registry.

Default image pull policy

When you (or a controller) submit a new Pod to the API server, your cluster sets the `imagePullPolicy` field when specific conditions are met:

- if you omit the `imagePullPolicy` field, and you specify the digest for the container image, the `imagePullPolicy` is automatically set to `IfNotPresent`.
- if you omit the `imagePullPolicy` field, and the tag for the container image is `:latest`, `imagePullPolicy` is automatically set to `Always`.
- if you omit the `imagePullPolicy` field, and you don't specify the tag for the container image, `imagePullPolicy` is automatically set to `Always`.
- if you omit the `imagePullPolicy` field, and you specify a tag for the container image that isn't `:latest`, the `imagePullPolicy` is automatically set to `IfNotPresent`.

Note:

The value of `imagePullPolicy` of the container is always set when the object is first *created*, and is not updated if the image's tag or digest later changes.

For example, if you create a Deployment with an image whose tag is *not* `:latest`, and later update that Deployment's image to a `:latest` tag, the `imagePullPolicy` field will *not* change to `Always`. You must manually change the pull policy of any object after its initial creation.

Required image pull

If you would like to always force a pull, you can do one of the following:

- Set the `imagePullPolicy` of the container to `Always`.
- Omit the `imagePullPolicy` and use `:latest` as the tag for the image to use; Kubernetes will set the policy to `Always` when you submit the Pod.
- Omit the `imagePullPolicy` and the tag for the image to use; Kubernetes will set the policy to `Always` when you submit the Pod.
- Enable the AlwaysPullImages admission controller.

ImagePullBackOff

When a kubelet starts creating containers for a Pod using a container runtime, it might be possible the container is in Waiting state because of `ImagePullBackOff`.

The status `ImagePullBackOff` means that a container could not start because Kubernetes could not pull a container image (for reasons such as invalid image name, or pulling from a private registry without `imagePullSecret`). The `BackOff` part indicates that Kubernetes will keep trying to pull the image, with an increasing back-off delay.

Kubernetes raises the delay between each attempt until it reaches a compiled-in limit, which is 300 seconds (5 minutes).

Image pull per runtime class

FEATURE STATE: `Kubernetes v1.29 [alpha]` (enabled by default: false)
Kubernetes includes alpha support for performing image pulls based on the RuntimeClass of a Pod.

If you enable the `RuntimeClassInImageCriApi` feature gate, the kubelet references container images by a tuple of image name and runtime handler rather than just the image name or digest. Your container runtime may adapt its behavior based on the selected runtime handler. Pulling images based on runtime class is useful for VM-based containers, such as Windows Hyper-V containers.

Serial and parallel image pulls

By default, the kubelet pulls images serially. In other words, the kubelet sends only one image pull request to the image service at a time. Other image pull requests have to wait until the one being processed is complete.

Nodes make image pull decisions in isolation. Even when you use serialized image pulls, two different nodes can pull the same image in parallel.

If you would like to enable parallel image pulls, you can set the field `serializeImagePulls` to false in the kubelet configuration. With `serializeImagePulls` set to false, image pull requests will be sent to the image service immediately, and multiple images will be pulled at the same time.

When enabling parallel image pulls, ensure that the image service of your container runtime can handle parallel image pulls.

The kubelet never pulls multiple images in parallel on behalf of one Pod. For example, if you have a Pod that has an init container and an application container, the image pulls for the two containers will not be parallelized. However, if you have two Pods that use different images, and the parallel image pull feature is enabled, the kubelet will pull the images in parallel on behalf of the two different Pods.

Maximum parallel image pulls

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

When `serializeImagePulls` is set to false, the kubelet defaults to no limit on the maximum number of images being pulled at the same time. If you would like to limit the number of parallel image pulls, you can set the field `maxParallelImagePulls` in the kubelet configuration. With `maxParallelImagePulls` set to *n*, only *n* images can be pulled at the same time, and any image pull beyond *n* will have to wait until at least one ongoing image pull is complete.

Limiting the number of parallel image pulls prevents image pulling from consuming too much network bandwidth or disk I/O, when parallel image pulling is enabled.

You can set `maxParallelImagePulls` to a positive number that is greater than or equal to 1. If you set `maxParallelImagePulls` to be greater than or equal to 2, you must set `serializeImagePulls` to false. The kubelet will fail to start with an invalid `maxParallelImagePulls` setting.

Multi-architecture images with image indexes

As well as providing binary images, a container registry can also serve a container image index. An image index can point to multiple image manifests for architecture-specific versions of a container. The idea is that you can have a name for an image (for example: `pause`, `example/mycontainer`, `kube-apiserver`) and allow different systems to fetch the right binary image for the machine architecture they are using.

The Kubernetes project typically creates container images for its releases with names that include the suffix `-$(ARCH)`. For backward compatibility, generate older images with suffixes. For instance, an image named as `pause` would be a multi-architecture image containing manifests for all supported architectures, while `pause-amd64` would be a backward-compatible version for older configurations, or for YAML files with hardcoded image names containing suffixes.

Using a private registry

Private registries may require authentication to be able to discover and/or pull images from them. Credentials can be provided in several ways:

- Specifying `imagePullSecrets` when you define a Pod

  Only Pods which provide their own keys can access the private registry.

- Configuring Nodes to Authenticate to a Private Registry

  - All Pods can read any configured private registries.
  - Requires node configuration by cluster administrator.

- Using a *kubelet credential provider* plugin to dynamically fetch credentials for private registries

  The kubelet can be configured to use credential provider exec plugin for the respective private registry.

- Pre-pulled Images

  - All Pods can use any images cached on a node.
  - Requires root access to all nodes to set up.

- Vendor-specific or local extensions

  If you're using a custom node configuration, you (or your cloud provider) can implement your mechanism for authenticating the node to the container registry.

These options are explained in more detail below.

Specifying `imagePullSecrets` on a Pod

Note:

This is the recommended approach to run containers based on images in private registries.

Kubernetes supports specifying container image registry keys on a Pod. All `imagePullSecrets` must be Secrets that exist in the same Namespace as the Pod. These Secrets must be of type `kubernetes.io/dockercfg` or `kubernetes.io/dockerconfigjson`.

Configuring nodes to authenticate to a private registry

Specific instructions for setting credentials depends on the container runtime and registry you chose to use. You should refer to your solution's documentation for the most accurate information.

For an example of configuring a private container image registry, see the Pull an Image from a Private Registry task. That example uses a private registry in Docker Hub.

Kubelet credential provider for authenticated image pulls

You can configure the kubelet to invoke a plugin binary to dynamically fetch registry credentials for a container image. This is the most robust and versatile way to fetch credentials for private registries, but also requires kubelet-level configuration to enable.

This technique can be especially useful for running static Pods that require container images hosted in a private registry. Using a ServiceAccount or a Secret to provide private registry credentials is not possible in the specification of a static Pod, because it *cannot* have references to other API resources in its specification.

See Configure a kubelet image credential provider for more details.

Interpretation of config.json

The interpretation of `config.json` varies between the original Docker implementation and the Kubernetes interpretation. In Docker, the `auths` keys can only specify root URLs, whereas Kubernetes allows glob URLs as well as prefix-matched paths. The only limitation is that glob patterns (`*`) have to include the dot (`.`) for each subdomain. The amount of matched subdomains has to be equal to the amount of glob patterns (`*.`), for example:

- `*.kubernetes.io` will *not* match `kubernetes.io`, but will match `abc.kubernetes.io`.
- `*.*.kubernetes.io` will *not* match `abc.kubernetes.io`, but will match `abc.def.kubernetes.io`.
- `prefix.*.io` will match `prefix.kubernetes.io`.
- `*-good.kubernetes.io` will match `prefix-good.kubernetes.io`.

This means that a `config.json` like this is valid:

```
{
    "auths": {
        "my-registry.example/images": { "auth": "…" },
        "*.my-registry.example/images": { "auth": "…" }
    }
}
```

Image pull operations pass the credentials to the CRI container runtime for every valid pattern. For example, the following container image names would match successfully:

- `my-registry.example/images`
- `my-registry.example/images/my-image`
- `my-registry.example/images/another-image`
- `sub.my-registry.example/images/my-image`

However, these container image names would *not* match:

- `a.sub.my-registry.example/images/my-image`
- `a.b.sub.my-registry.example/images/my-image`

The kubelet performs image pulls sequentially for every found credential. This means that multiple entries in `config.json` for different paths are possible, too:

```
{
    "auths": {
        "my-registry.example/images": {
            "auth": "…"
        },
        "my-registry.example/images/subpath": {
            "auth": "…"
        }
    }
}
```

If now a container specifies an image `my-registry.example/images/subpath/my-image` to be pulled, then the kubelet will try to download it using both authentication sources if one of them fails.

Pre-pulled images

Note:

This approach is suitable if you can control node configuration. It will not work reliably if your cloud provider manages nodes and replaces them automatically.

By default, the kubelet tries to pull each image from the specified registry. However, if the `imagePullPolicy` property of the container is set to `IfNotPresent` or `Never`, then a local image is used (preferentially or exclusively, respectively).

If you want to rely on pre-pulled images as a substitute for registry authentication, you must ensure all nodes in the cluster have the same pre-pulled images.

This can be used to preload certain images for speed or as an alternative to authenticating to a private registry.

Similar to the usage of the kubelet credential provider, pre-pulled images are also suitable for launching static Pods that depend on images hosted in a private registry.

Note:

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

Access to pre-pulled images may be authorized according to image pull credential verification.

Ensure image pull credential verification

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

If the `KubeletEnsureSecretPulledImages` feature gate is enabled for your cluster, Kubernetes will validate image credentials for every image that requires credentials to be pulled, even if that image is already present on the node. This validation ensures that images in a Pod request which have not been successfully pulled with the provided credentials must re-pull the images from the registry. Additionally, image pulls that re-use the same credentials which previously resulted in a successful image pull will not need to re-pull from the registry and are instead validated locally without accessing the registry (provided the image is available locally). This is controlled by the`imagePullCredentialsVerificationPolicy` field in the Kubelet configuration.

This configuration controls when image pull credentials must be verified if the image is already present on the node:

- `NeverVerify`: Mimics the behavior of having this feature gate disabled. If the image is present locally, image pull credentials are not verified.
- `NeverVerifyPreloadedImages`: Images pulled outside the kubelet are not verified, but all other images will have their credentials verified. This is the default behavior.
- `NeverVerifyAllowListedImages`: Images pulled outside the kubelet and mentioned within the `preloadedImagesVerificationAllowlist` specified in the kubelet config are not verified.
- `AlwaysVerify`: All images will have their credentials verified before they can be used.

This verification applies to pre-pulled images, images pulled using node-wide secrets, and images pulled using Pod-level secrets.

Note:

In the case of credential rotation, the credentials previously used to pull the image will continue to verify without the need to access the registry. New or rotated credentials will require the image to be re-pulled from the registry.

Creating a Secret with a Docker config

You need to know the username, registry password and client email address for authenticating to the registry, as well as its hostname. Run the following command, substituting placeholders with the appropriate values:

```
kubectl create secret docker-registry <name> \
  --docker-server=<docker-registry-server> \  --docker-username=<docker-user> \  --docker-password=<docker-password> \  --docker-e
```

If you already have a Docker credentials file then, rather than using the above command, you can import the credentials file as a Kubernetes Secret. Create a Secret based on existing Docker credentials explains how to set this up.

This is particularly useful if you are using multiple private container registries, as `kubectl create secret docker-registry` creates a Secret that only works with a single private registry.

Note:

Pods can only reference image pull secrets in their own namespace, so this process needs to be done one time per namespace.

Referring to `imagePullSecrets` on a Pod

Now, you can create pods which reference that secret by adding the `imagePullSecrets` section to a Pod definition. Each item in the `imagePullSecrets` array can only reference one Secret in the same namespace.

For example:

```
cat <<EOF > pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: foo
  namespace: awesomeapps
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
EOF

cat <<EOF >> ./kustomization.yaml
resources:
- pod.yaml
EOF
```

This needs to be done for each Pod that is using a private registry.

However, you can automate this process by specifying the `imagePullSecrets` section in a ServiceAccount resource. See Add ImagePullSecrets to a Service Account for detailed instructions.

You can use this in conjunction with a per-node `.docker/config.json`. The credentials will be merged.

Use cases

There are a number of solutions for configuring private registries. Here are some common use cases and suggested solutions.

1. Cluster running only non-proprietary (e.g. open-source) images. No need to hide images.
    - Use public images from a public registry
        - No configuration required.
        - Some cloud providers automatically cache or mirror public images, which improves availability and reduces the time to pull images.
2. Cluster running some proprietary images which should be hidden to those outside the company, but visible to all cluster users.
    - Use a hosted private registry
        - Manual configuration may be required on the nodes that need to access to private registry.
    - Or, run an internal private registry behind your firewall with open read access.
        - No Kubernetes configuration is required.
    - Use a hosted container image registry service that controls image access
        - It will work better with Node autoscaling than manual node configuration.
    - Or, on a cluster where changing the node configuration is inconvenient, use `imagePullSecrets`.
3. Cluster with proprietary images, a few of which require stricter access control.
    - Ensure AlwaysPullImages admission controller is active. Otherwise, all Pods potentially have access to all images.
    - Move sensitive data into a Secret resource, instead of packaging it in an image.
4. A multi-tenant cluster where each tenant needs own private registry.
    - Ensure AlwaysPullImages admission controller is active. Otherwise, all Pods of all tenants potentially have access to all images.
    - Run a private registry with authorization required.
    - Generate registry credentials for each tenant, store into a Secret, and propagate the Secret to every tenant namespace.
    - The tenant then adds that Secret to `imagePullSecrets` of each namespace.

If you need access to multiple registries, you can create one Secret per registry.

Legacy built-in kubelet credential provider

In older versions of Kubernetes, the kubelet had a direct integration with cloud provider credentials. This provided the ability to dynamically fetch credentials for image registries.

There were three built-in implementations of the kubelet credential provider integration: ACR (Azure Container Registry), ECR (Elastic Container Registry), and GCR (Google Container Registry).

Starting with version 1.26 of Kubernetes, the legacy mechanism has been removed, so you would need to either:

- configure a kubelet image credential provider on each node; or
- specify image pull credentials using `imagePullSecrets` and at least one Secret.

What's next

- Read the OCI Image Manifest Specification.
- Learn about container image garbage collection.
- Learn more about pulling an Image from a Private Registry.

API Priority and Fairness

FEATURE STATE: `Kubernetes v1.29 [stable]`

Controlling the behavior of the Kubernetes API server in an overload situation is a key task for cluster administrators. The kube-apiserver has some controls available (i.e. the `--max-requests-inflight` and `--max-mutating-requests-inflight` command-line flags) to limit the amount of outstanding work that will be accepted, preventing a flood of inbound requests from overloading and potentially crashing the API server, but these flags are not enough to ensure that the most important requests get through in a period of high traffic.

The API Priority and Fairness feature (APF) is an alternative that improves upon aforementioned max-inflight limitations. APF classifies and isolates requests in a more fine-grained way. It also introduces a limited amount of queuing, so that no requests are rejected in cases of very brief bursts. Requests are dispatched from queues using a fair queuing technique so that, for example, a poorly-behaved controller need not starve others (even at the same priority level).

This feature is designed to work well with standard controllers, which use informers and react to failures of API requests with exponential back-off, and other clients that also work this way.

Caution:

Some requests classified as "long-running"—such as remote command execution or log tailing—are not subject to the API Priority and Fairness filter. This is also true for the `--max-requests-inflight` flag without the API Priority and Fairness feature enabled. API Priority and Fairness *does* apply to **watch** requests. When API Priority and Fairness is disabled, **watch** requests are not subject to the `--max-requests-inflight` limit.

Enabling/Disabling API Priority and Fairness

The API Priority and Fairness feature is controlled by a command-line flag and is enabled by default. See Options for a general explanation of the available kube-apiserver command-line options and how to enable and disable them. The name of the command-line option for APF is "--enable-priority-and-fairness". This feature also involves an API Group with: (a) a stable `v1` version, introduced in 1.29, and enabled by default (b) a `v1beta3` version, enabled by default, and deprecated in v1.29. You can disable the API group beta version `v1beta3` by adding the following command-line flags to your `kube-apiserver` invocation:

```
kube-apiserver \
--runtime-config=flowcontrol.apiserver.k8s.io/v1beta3=false \ # …and other flags as usual
```

The command-line flag `--enable-priority-and-fairness=false` will disable the API Priority and Fairness feature.

Recursive server scenarios

API Priority and Fairness must be used carefully in recursive server scenarios. These are scenarios in which some server A, while serving a request, issues a subsidiary request to some server B. Perhaps server B might even make a further subsidiary call back to server A. In situations where Priority and Fairness control is applied to both the original request and some subsidiary ones(s), no matter how deep in the recursion, there is a danger of priority inversions and/or deadlocks.

One example of recursion is when the `kube-apiserver` issues an admission webhook call to server B, and while serving that call, server B makes a further subsidiary request back to the `kube-apiserver`. Another example of recursion is when an `APIService` object directs the `kube-apiserver` to delegate requests about a certain API group to a custom external server B (this is one of the things called "aggregation").

When the original request is known to belong to a certain priority level, and the subsidiary controlled requests are classified to higher priority levels, this is one possible solution. When the original requests can belong to any priority level, the subsidiary controlled requests have to be exempt from Priority and Fairness limitation. One way to do that is with the objects that configure classification and handling, discussed below. Another way is to disable Priority and Fairness on server B entirely, using the techniques discussed above. A third way, which is the simplest to use when server B is not `kube-apiserver`, is to build server B with Priority and Fairness disabled in the code.

Concepts

There are several distinct features involved in the API Priority and Fairness feature. Incoming requests are classified by attributes of the request using *FlowSchemas*, and assigned to priority levels. Priority levels add a degree of isolation by maintaining separate concurrency limits, so that requests assigned to different priority levels cannot starve each other. Within a priority level, a fair-queuing algorithm prevents requests from different *flows* from starving each other, and allows for requests to be queued to prevent bursty traffic from causing failed requests when the average load is acceptably low.

Priority Levels

Without APF enabled, overall concurrency in the API server is limited by the `kube-apiserver` flags `--max-requests-inflight` and `--max-mutating-requests-inflight`. With APF enabled, the concurrency limits defined by these flags are summed and then the sum is divided up among a configurable set of *priority levels*. Each incoming request is assigned to a single priority level, and each priority level will only dispatch as many concurrent requests as its particular limit allows.

The default configuration, for example, includes separate priority levels for leader-election requests, requests from built-in controllers, and requests from Pods. This means that an ill-behaved Pod that floods the API server with requests cannot prevent leader election or actions by the built-in controllers from succeeding.

The concurrency limits of the priority levels are periodically adjusted, allowing under-utilized priority levels to temporarily lend concurrency to heavily-utilized levels. These limits are based on nominal limits and bounds on how much concurrency a priority level may lend and how much it may borrow, all derived from the configuration objects mentioned below.

Seats Occupied by a Request

The above description of concurrency management is the baseline story. Requests have different durations but are counted equally at any given moment when comparing against a priority level's concurrency limit. In the baseline story, each request occupies one unit of concurrency. The word "seat" is used to mean one unit of concurrency, inspired by the way each passenger on a train or aircraft takes up one of the fixed supply of seats.

But some requests take up more than one seat. Some of these are **list** requests that the server estimates will return a large number of objects. These have been found to put an exceptionally heavy burden on the server. For this reason, the server estimates the number of objects that will be returned and considers the request to take a number of seats that is proportional to that estimated number.

Execution time tweaks for watch requests

API Priority and Fairness manages **watch** requests, but this involves a couple more excursions from the baseline behavior. The first concerns how long a **watch** request is considered to occupy its seat. Depending on request parameters, the response to a **watch** request may or may not begin with **create** notifications for all the relevant pre-existing objects. API Priority and Fairness considers a **watch** request to be done with its seat once that initial burst of notifications, if any, is over.

The normal notifications are sent in a concurrent burst to all relevant **watch** response streams whenever the server is notified of an object create/update/delete. To account for this work, API Priority and Fairness considers every write request to spend some additional time occupying seats after the actual writing is done. The server estimates the number of notifications to be sent and adjusts the write request's number of seats and seat occupancy time to include this extra work.

Queuing

Even within a priority level there may be a large number of distinct sources of traffic. In an overload situation, it is valuable to prevent one stream of requests from starving others (in particular, in the relatively common case of a single buggy client flooding the kube-apiserver with requests, that buggy client would ideally not have much measurable impact on other clients at all). This is handled by use of a fair-queuing algorithm to process requests that are assigned the same priority level. Each request is assigned to a *flow*, identified by the name of the matching FlowSchema plus a *flow distinguisher* — which is either the requesting user, the target resource's namespace, or nothing — and the system attempts to give approximately equal weight to requests in different flows of the same priority level. To enable distinct handling of distinct instances, controllers that have many instances should authenticate with distinct usernames

After classifying a request into a flow, the API Priority and Fairness feature then may assign the request to a queue. This assignment uses a technique known as shuffle sharding, which makes relatively efficient use of queues to insulate low-intensity flows from high-intensity flows.

The details of the queuing algorithm are tunable for each priority level, and allow administrators to trade off memory use, fairness (the property that independent flows will all make progress when total traffic exceeds capacity), tolerance for bursty traffic, and the added latency induced by queuing.

Exempt requests

Some requests are considered sufficiently important that they are not subject to any of the limitations imposed by this feature. These exemptions prevent an improperly-configured flow control configuration from totally disabling an API server.

Resources

The flow control API involves two kinds of resources. PriorityLevelConfigurations define the available priority levels, the share of the available concurrency budget that each can handle, and allow for fine-tuning queuing behavior. FlowSchemas are used to classify individual inbound requests, matching each to a single PriorityLevelConfiguration.

PriorityLevelConfiguration

A PriorityLevelConfiguration represents a single priority level. Each PriorityLevelConfiguration has an independent limit on the number of outstanding requests, and limitations on the number of queued requests.

The nominal concurrency limit for a PriorityLevelConfiguration is not specified in an absolute number of seats, but rather in "nominal concurrency shares." The total concurrency limit for the API Server is distributed among the existing PriorityLevelConfigurations in proportion to these shares, to give each level its nominal limit in terms of seats. This allows a cluster administrator to scale up or down the total amount of traffic to a server by restarting `kube-apiserver` with a different value for `--max-requests-inflight` (or `--max-mutating-requests-inflight`), and all PriorityLevelConfigurations will see their maximum allowed concurrency go up (or down) by the same fraction.

Caution:

In the versions before `v1beta3` the relevant PriorityLevelConfiguration field is named "assured concurrency shares" rather than "nominal concurrency shares". Also, in Kubernetes release 1.25 and earlier there were no periodic adjustments: the nominal/assured limits were always applied without adjustment.

The bounds on how much concurrency a priority level may lend and how much it may borrow are expressed in the PriorityLevelConfiguration as percentages of the level's nominal limit. These are resolved to absolute numbers of seats by multiplying with the nominal limit / 100.0 and rounding. The dynamically adjusted concurrency limit of a priority level is constrained to lie between (a) a lower bound of its nominal limit minus its lendable seats and (b) an upper bound of its nominal limit plus the seats it may borrow. At each adjustment the dynamic limits are derived by each priority level reclaiming any lent seats for which demand recently appeared and then jointly fairly responding to the recent seat demand on the priority levels, within the bounds just described.

Caution:

With the Priority and Fairness feature enabled, the total concurrency limit for the server is set to the sum of `--max-requests-inflight` and `--max-mutating-requests-inflight`. There is no longer any distinction made between mutating and non-mutating requests; if you want to treat them separately for a given resource, make separate FlowSchemas that match the mutating and non-mutating verbs respectively.

When the volume of inbound requests assigned to a single PriorityLevelConfiguration is more than its permitted concurrency level, the `type` field of its specification determines what will happen to extra requests. A type of `Reject` means that excess traffic will immediately be rejected with an HTTP 429 (Too Many Requests) error. A type of `Queue` means that requests above the threshold will be queued, with the shuffle sharding and fair queuing techniques used to balance progress between request flows.

The queuing configuration allows tuning the fair queuing algorithm for a priority level. Details of the algorithm can be read in the enhancement proposal, but in short:

- Increasing `queues` reduces the rate of collisions between different flows, at the cost of increased memory usage. A value of 1 here effectively disables the fair-queuing logic, but still allows requests to be queued.

- Increasing `queueLengthLimit` allows larger bursts of traffic to be sustained without dropping any requests, at the cost of increased latency and memory usage.

- Changing `handSize` allows you to adjust the probability of collisions between different flows and the overall concurrency available to a single flow in an overload situation.

  Note:

  A larger `handSize` makes it less likely for two individual flows to collide (and therefore for one to be able to starve the other), but more likely that a small number of flows can dominate the apiserver. A larger `handSize` also potentially increases the amount of latency that a single high-traffic flow can cause. The maximum number of queued requests possible from a single flow is `handSize * queueLengthLimit`.

Following is a table showing an interesting collection of shuffle sharding configurations, showing for each the probability that a given mouse (low-intensity flow) is squished by the elephants (high-intensity flows) for an illustrative collection of numbers of elephants. See https://play.golang.org/p/Gi0PLgVHiUg , which computes this table.

HandSize	Queues	1 elephant	4 elephants	16 elephants
12	32	4.428838398950118e-09	0.11431348830099144	0.9935089607656024
10	32	1.550093439632541e-08	0.0626479840223545	0.9753101519027554
10	64	6.601827268370426e-12	0.00045571320990370776	0.49999929150089345
9	64	3.6310049976037345e-11	0.00045501212304112273	0.4282314876454858
8	64	2.25929199850899e-10	0.0004886697053040446	0.35935114681123076
8	128	6.994461389026097e-13	3.4055790161620863e-06	0.02746173137155063
7	128	1.0579122850901972e-11	6.960839379258192e-06	0.02406157386340147
7	256	7.597695465552631e-14	6.728547142019406e-08	0.0006709661542533682
6	256	2.7134626662687968e-12	2.9516464018476436e-07	0.0008895654642000348
6	512	4.116062922897309e-14	4.982983350480894e-09	2.26025764343413e-05
6	1024	6.337324016514285e-16	8.09060164312957e-11	4.517408062903668e-07

FlowSchema

A FlowSchema matches some inbound requests and assigns them to a priority level. Every inbound request is tested against FlowSchemas, starting with those with the numerically lowest `matchingPrecedence` and working upward. The first match wins.

Caution:

Only the first matching FlowSchema for a given request matters. If multiple FlowSchemas match a single inbound request, it will be assigned based on the one with the highest `matchingPrecedence`. If multiple FlowSchemas with equal `matchingPrecedence` match the same request, the one with lexicographically smaller `name` will win, but it's better not to rely on this, and instead to ensure that no two FlowSchemas have the same `matchingPrecedence`.

A FlowSchema matches a given request if at least one of its `rules` matches. A rule matches if at least one of its `subjects` *and* at least one of its `resourceRules` or `nonResourceRules` (depending on whether the incoming request is for a resource or non-resource URL) match the request.

For the `name` field in subjects, and the `verbs`, `apiGroups`, `resources`, `namespaces`, and `nonResourceURLs` fields of resource and non-resource rules, the wildcard `*` may be specified to match all values for the given field, effectively removing it from consideration.

A FlowSchema's `distinguisherMethod.type` determines how requests matching that schema will be separated into flows. It may be `ByUser`, in which one requesting user will not be able to starve other users of capacity; `ByNamespace`, in which requests for resources in one namespace will not be able to starve requests for resources in other namespaces of capacity; or blank (or `distinguisherMethod` may be omitted entirely), in which all requests matched by this FlowSchema will be considered part of a single flow. The correct choice for a given FlowSchema depends on the resource and your particular environment.

Defaults

Each kube-apiserver maintains two sorts of APF configuration objects: mandatory and suggested.

Mandatory Configuration Objects

The four mandatory configuration objects reflect fixed built-in guardrail behavior. This is behavior that the servers have before those objects exist, and when those objects exist their specs reflect this behavior. The four mandatory objects are as follows.

- The mandatory `exempt` priority level is used for requests that are not subject to flow control at all: they will always be dispatched immediately. The mandatory `exempt` FlowSchema classifies all requests from the `system:masters` group into this priority level. You may define other FlowSchemas that direct other requests to this priority level, if appropriate.

- The mandatory `catch-all` priority level is used in combination with the mandatory `catch-all` FlowSchema to make sure that every request gets some kind of classification. Typically you should not rely on this catch-all configuration, and should create your own catch-all FlowSchema and PriorityLevelConfiguration (or use the suggested `global-default` priority level that is installed by default) as appropriate. Because it is not expected to be used normally, the mandatory `catch-all` priority level has a very small concurrency share and does not queue requests.

Suggested Configuration Objects

The suggested FlowSchemas and PriorityLevelConfigurations constitute a reasonable default configuration. You can modify these and/or create additional configuration objects if you want. If your cluster is likely to experience heavy load then you should consider what configuration will work best.

The suggested configuration groups requests into six priority levels:

- The `node-high` priority level is for health updates from nodes.

- The `system` priority level is for non-health requests from the `system:nodes` group, i.e. Kubelets, which must be able to contact the API server in order for workloads to be able to schedule on them.

- The `leader-election` priority level is for leader election requests from built-in controllers (in particular, requests for `endpoints`, `configmaps`, or `leases` coming from the `system:kube-controller-manager` or `system:kube-scheduler` users and service accounts in the `kube-system` namespace). These are important to isolate from other traffic because failures in leader election cause their controllers to fail and restart, which in turn causes more expensive traffic as the new controllers sync their informers.

- The `workload-high` priority level is for other requests from built-in controllers.

- The `workload-low` priority level is for requests from any other service account, which will typically include all requests from controllers running in Pods.

- The `global-default` priority level handles all other traffic, e.g. interactive `kubectl` commands run by nonprivileged users.

The suggested FlowSchemas serve to steer requests into the above priority levels, and are not enumerated here.

Maintenance of the Mandatory and Suggested Configuration Objects

Each `kube-apiserver` independently maintains the mandatory and suggested configuration objects, using initial and periodic behavior. Thus, in a situation with a mixture of servers of different versions there may be thrashing as long as different servers have different opinions of the proper content of these objects.

Each `kube-apiserver` makes an initial maintenance pass over the mandatory and suggested configuration objects, and after that does periodic maintenance (once per minute) of those objects.

For the mandatory configuration objects, maintenance consists of ensuring that the object exists and, if it does, has the proper spec. The server refuses to allow a creation or update with a spec that is inconsistent with the server's guardrail behavior.

Maintenance of suggested configuration objects is designed to allow their specs to be overridden. Deletion, on the other hand, is not respected: maintenance will restore the object. If you do not want a suggested configuration object then you need to keep it around but set its spec to have minimal consequences. Maintenance of suggested objects is also designed to support automatic migration when a new version of the `kube-apiserver` is rolled out, albeit potentially with thrashing while there is a mixed population of servers.

Maintenance of a suggested configuration object consists of creating it --- with the server's suggested spec --- if the object does not exist. OTOH, if the object already exists, maintenance behavior depends on whether the `kube-apiservers` or the users control the object. In the former case, the server ensures that the object's spec is what the server suggests; in the latter case, the spec is left alone.

The question of who controls the object is answered by first looking for an annotation with key `apf.kubernetes.io/autoupdate-spec`. If there is such an annotation and its value is `true` then the kube-apiservers control the object. If there is such an annotation and its value is `false` then the users control the object. If neither of those conditions holds then the `metadata.generation` of the object is consulted. If that is 1 then the kube-apiservers control the object. Otherwise the users control the object. These rules were introduced in release 1.22 and their consideration of `metadata.generation` is for the sake of migration from the simpler earlier behavior. Users who wish to control a suggested configuration object should set its `apf.kubernetes.io/autoupdate-spec` annotation to `false`.

Maintenance of a mandatory or suggested configuration object also includes ensuring that it has an `apf.kubernetes.io/autoupdate-spec` annotation that accurately reflects whether the kube-apiservers control the object.

Maintenance also includes deleting objects that are neither mandatory nor suggested but are annotated `apf.kubernetes.io/autoupdate-spec=true`.

Health check concurrency exemption

The suggested configuration gives no special treatment to the health check requests on kube-apiservers from their local kubelets --- which tend to use the secured port but supply no credentials. With the suggested config, these requests get assigned to the `global-default` FlowSchema and the corresponding `global-default` priority level, where other traffic can crowd them out.

If you add the following additional FlowSchema, this exempts those requests from rate limiting.

Caution:

Making this change also allows any hostile party to then send health-check requests that match this FlowSchema, at any volume they like. If you have a web traffic filter or similar external security mechanism to protect your cluster's API server from general internet traffic, you can configure rules to block any health check requests that originate from outside your cluster.
priority-and-fairness/health-for-strangers.yaml Copy priority-and-fairness/health-for-strangers.yaml to clipboard

```
apiVersion: flowcontrol.apiserver.k8s.io/v1
kind: FlowSchemametadata:  name: health-for-strangersspec:  matchingPrecedence: 1000  priorityLevelConfiguration:    name: exempt
```

Observability

Metrics

Note:

In versions of Kubernetes before v1.20, the labels `flow_schema` and `priority_level` were inconsistently named `flowSchema` and `priorityLevel`, respectively. If you're running Kubernetes versions v1.19 and earlier, you should refer to the documentation for your version.

When you enable the API Priority and Fairness feature, the kube-apiserver exports additional metrics. Monitoring these can help you determine whether your configuration is inappropriately throttling important traffic, or find poorly-behaved workloads that may be harming system health.

Maturity level BETA

- `apiserver_flowcontrol_rejected_requests_total` is a counter vector (cumulative since server start) of requests that were rejected, broken down by the labels `flow_schema` (indicating the one that matched the request), `priority_level` (indicating the one to which the request was assigned), and `reason`. The `reason` label will be one of the following values:

    - `queue-full`, indicating that too many requests were already queued.
    - `concurrency-limit`, indicating that the PriorityLevelConfiguration is configured to reject rather than queue excess requests.
    - `time-out`, indicating that the request was still in the queue when its queuing time limit expired.
    - `cancelled`, indicating that the request is not purge locked and has been ejected from the queue.

- `apiserver_flowcontrol_dispatched_requests_total` is a counter vector (cumulative since server start) of requests that began executing, broken down by `flow_schema` and `priority_level`.

- `apiserver_flowcontrol_current_inqueue_requests` is a gauge vector holding the instantaneous number of queued (not executing) requests, broken down by `priority_level` and `flow_schema`.

- `apiserver_flowcontrol_current_executing_requests` is a gauge vector holding the instantaneous number of executing (not waiting in a queue) requests, broken down by `priority_level` and `flow_schema`.

- `apiserver_flowcontrol_current_executing_seats` is a gauge vector holding the instantaneous number of occupied seats, broken down by `priority_level` and `flow_schema`.

- `apiserver_flowcontrol_request_wait_duration_seconds` is a histogram vector of how long requests spent queued, broken down by the labels `flow_schema`, `priority_level`, and `execute`. The `execute` label indicates whether the request has started executing.

    Note:

    Since each FlowSchema always assigns requests to a single PriorityLevelConfiguration, you can add the histograms for all the FlowSchemas for one priority level to get the effective histogram for requests assigned to that priority level.

- `apiserver_flowcontrol_nominal_limit_seats` is a gauge vector holding each priority level's nominal concurrency limit, computed from the API server's total concurrency limit and the priority level's configured nominal concurrency shares.

Maturity level ALPHA

- `apiserver_current_inqueue_requests` is a gauge vector of recent high water marks of the number of queued requests, grouped by a label named `request_kind` whose value is `mutating` or `readOnly`. These high water marks describe the largest number seen in the one second window most recently completed. These complement the older `apiserver_current_inflight_requests` gauge vector that holds the last window's high water mark of number of requests actively being served.

- `apiserver_current_inqueue_seats` is a gauge vector of the sum over queued requests of the largest number of seats each will occupy, grouped by labels named `flow_schema` and `priority_level`.

- `apiserver_flowcontrol_read_vs_write_current_requests` is a histogram vector of observations, made at the end of every nanosecond, of the number of requests broken down by the labels `phase` (which takes on the values `waiting` and `executing`) and `request_kind` (which takes on the values `mutating` and `readOnly`). Each observed value is a ratio, between 0 and 1, of the number of requests divided by the corresponding limit on the number of requests (queue volume limit for waiting and concurrency limit for executing).

- `apiserver_flowcontrol_request_concurrency_in_use` is a gauge vector holding the instantaneous number of occupied seats, broken down by `priority_level` and `flow_schema`.

- `apiserver_flowcontrol_priority_level_request_utilization` is a histogram vector of observations, made at the end of each nanosecond, of the number of requests broken down by the labels `phase` (which takes on the values `waiting` and `executing`) and `priority_level`. Each observed value is a ratio, between 0 and 1, of a number of requests divided by the corresponding limit on the number of requests (queue volume limit for waiting and concurrency limit for executing).

- `apiserver_flowcontrol_priority_level_seat_utilization` is a histogram vector of observations, made at the end of each nanosecond, of the utilization of a priority level's concurrency limit, broken down by `priority_level`. This utilization is the fraction (number of seats occupied) / (concurrency limit). This metric considers all stages of execution (both normal and the extra delay at the end of a write to cover for the corresponding notification work) of all requests except WATCHes; for those it considers only the initial stage that delivers notifications of pre-existing objects. Each histogram in the vector is also labeled with `phase: executing` (there is no seat limit for the waiting phase).

- `apiserver_flowcontrol_request_queue_length_after_enqueue` is a histogram vector of queue lengths for the queues, broken down by `priority_level` and `flow_schema`, as sampled by the enqueued requests. Each request that gets queued contributes one sample to its histogram, reporting the length of the queue immediately after the request was added. Note that this produces different statistics than an unbiased survey would.

  Note:

  An outlier value in a histogram here means it is likely that a single flow (i.e., requests by one user or for one namespace, depending on configuration) is flooding the API server, and being throttled. By contrast, if one priority level's histogram shows that all queues for that priority level are longer than those for other priority levels, it may be appropriate to increase that PriorityLevelConfiguration's concurrency shares.

- `apiserver_flowcontrol_request_concurrency_limit` is the same as `apiserver_flowcontrol_nominal_limit_seats`. Before the introduction of concurrency borrowing between priority levels, this was always equal to `apiserver_flowcontrol_current_limit_seats` (which did not exist as a distinct metric).

- `apiserver_flowcontrol_lower_limit_seats` is a gauge vector holding the lower bound on each priority level's dynamic concurrency limit.

- `apiserver_flowcontrol_upper_limit_seats` is a gauge vector holding the upper bound on each priority level's dynamic concurrency limit.

- `apiserver_flowcontrol_demand_seats` is a histogram vector counting observations, at the end of every nanosecond, of each priority level's ratio of (seat demand) / (nominal concurrency limit). A priority level's seat demand is the sum, over both queued requests and those in the initial phase of execution, of the maximum of the number of seats occupied in the request's initial and final execution phases.

- `apiserver_flowcontrol_demand_seats_high_watermark` is a gauge vector holding, for each priority level, the maximum seat demand seen during the last concurrency borrowing adjustment period.

- `apiserver_flowcontrol_demand_seats_average` is a gauge vector holding, for each priority level, the time-weighted average seat demand seen during the last concurrency borrowing adjustment period.

- `apiserver_flowcontrol_demand_seats_stdev` is a gauge vector holding, for each priority level, the time-weighted population standard deviation of seat demand seen during the last concurrency borrowing adjustment period.

- `apiserver_flowcontrol_demand_seats_smoothed` is a gauge vector holding, for each priority level, the smoothed enveloped seat demand determined at the last concurrency adjustment.

- `apiserver_flowcontrol_target_seats` is a gauge vector holding, for each priority level, the concurrency target going into the borrowing allocation problem.

- `apiserver_flowcontrol_seat_fair_frac` is a gauge holding the fair allocation fraction determined in the last borrowing adjustment.

- `apiserver_flowcontrol_current_limit_seats` is a gauge vector holding, for each priority level, the dynamic concurrency limit derived in the last adjustment.

- `apiserver_flowcontrol_request_execution_seconds` is a histogram vector of how long requests took to actually execute, broken down by `flow_schema` and `priority_level`.

- `apiserver_flowcontrol_watch_count_samples` is a histogram vector of the number of active WATCH requests relevant to a given write, broken down by `flow_schema` and `priority_level`.

- `apiserver_flowcontrol_work_estimated_seats` is a histogram vector of the number of estimated seats (maximum of initial and final stage of execution) associated with requests, broken down by `flow_schema` and `priority_level`.

- `apiserver_flowcontrol_request_dispatch_no_accommodation_total` is a counter vector of the number of events that in principle could have led to a request being dispatched but did not, due to lack of available concurrency, broken down by `flow_schema` and `priority_level`.

- `apiserver_flowcontrol_epoch_advance_total` is a counter vector of the number of attempts to jump a priority level's progress meter backward to avoid numeric overflow, grouped by `priority_level` and `success`.

Good practices for using API Priority and Fairness

When a given priority level exceeds its permitted concurrency, requests can experience increased latency or be dropped with an HTTP 429 (Too Many Requests) error. To prevent these side effects of APF, you can modify your workload or tweak your APF settings to ensure there are sufficient seats available to serve your requests.

To detect whether requests are being rejected due to APF, check the following metrics:

- apiserver_flowcontrol_rejected_requests_total: the total number of requests rejected per FlowSchema and PriorityLevelConfiguration.
- apiserver_flowcontrol_current_inqueue_requests: the current number of requests queued per FlowSchema and PriorityLevelConfiguration.
- apiserver_flowcontrol_request_wait_duration_seconds: the latency added to requests waiting in queues.
- apiserver_flowcontrol_priority_level_seat_utilization: the seat utilization per PriorityLevelConfiguration.

Workload modifications

To prevent requests from queuing and adding latency or being dropped due to APF, you can optimize your requests by:

- Reducing the rate at which requests are executed. A fewer number of requests over a fixed period will result in a fewer number of seats being needed at a given time.

- Avoid issuing a large number of expensive requests concurrently. Requests can be optimized to use fewer seats or have lower latency so that these requests hold those seats for a shorter duration. List requests can occupy more than 1 seat depending on the number of objects fetched during the request. Restricting the number of objects retrieved in a list request, for example by using pagination, will use less total seats over a shorter period. Furthermore, replacing list requests with watch requests will require lower total concurrency shares as watch requests only occupy 1 seat during its initial burst of notifications. If using streaming lists in versions 1.27 and later, watch requests will occupy the same number of seats as a list request for its initial burst of notifications because the entire state of the collection has to be streamed. Note that in both cases, a watch request will not hold any seats after this initial phase.

Keep in mind that queuing or rejected requests from APF could be induced by either an increase in the number of requests or an increase in latency for existing requests. For example, if requests that normally take 1s to execute start taking 60s, it is possible that APF will start rejecting requests because requests are occupying seats for a longer duration than normal due to this increase in latency. If APF starts rejecting requests across multiple priority levels without a significant change in workload, it is possible there is an underlying issue with control plane performance rather than the workload or APF settings.

Priority and fairness settings

You can also modify the default FlowSchema and PriorityLevelConfiguration objects or create new objects of these types to better accommodate your workload.

APF settings can be modified to:

- Give more seats to high priority requests.
- Isolate non-essential or expensive requests that would starve a concurrency level if it was shared with other flows.

Give more seats to high priority requests

1. If possible, the number of seats available across all priority levels for a particular `kube-apiserver` can be increased by increasing the values for the `max-requests-inflight` and `max-mutating-requests-inflight` flags. Alternatively, horizontally scaling the number of `kube-apiserver` instances will increase the total concurrency per priority level across the cluster assuming there is sufficient load balancing of requests.
2. You can create a new FlowSchema which references a PriorityLevelConfiguration with a larger concurrency level. This new PriorityLevelConfiguration could be an existing level or a new level with its own set of nominal concurrency shares. For example, a new FlowSchema could be introduced to change the PriorityLevelConfiguration for your requests from global-default to workload-low to increase the number of seats available to your user. Creating a new PriorityLevelConfiguration will reduce the number of seats designated for existing levels. Recall that editing a default FlowSchema or PriorityLevelConfiguration will require setting the `apf.kubernetes.io/autoupdate-spec` annotation to false.
3. You can also increase the NominalConcurrencyShares for the PriorityLevelConfiguration which is serving your high priority requests. Alternatively, for versions 1.26 and later, you can increase the LendablePercent for competing priority levels so that the given priority level has a higher pool of seats it can borrow.

Isolate non-essential requests from starving other flows

For request isolation, you can create a FlowSchema whose subject matches the user making these requests or create a FlowSchema that matches what the request is (corresponding to the resourceRules). Next, you can map this FlowSchema to a PriorityLevelConfiguration with a low share of seats.

For example, suppose list event requests from Pods running in the default namespace are using 10 seats each and execute for 1 minute. To prevent these expensive requests from impacting requests from other Pods using the existing service-accounts FlowSchema, you can apply the following FlowSchema to isolate these list calls from other requests.

Example FlowSchema object to isolate list event requests:

`priority-and-fairness/list-events-default-service-account.yaml`
Copy priority-and-fairness/list-events-default-service-account.yaml to clipboard

```
apiVersion: flowcontrol.apiserver.k8s.io/v1
kind: FlowSchemametadata:  name: list-events-default-service-accountspec:  distinguisherMethod:    type: ByUser  matchingPrecedenc
```

- This FlowSchema captures all list event calls made by the default service account in the default namespace. The matching precedence 8000 is lower than the value of 9000 used by the existing service-accounts FlowSchema so these list event calls will match list-events-default-service-account rather than service-accounts.
- The catch-all PriorityLevelConfiguration is used to isolate these requests. The catch-all priority level has a very small concurrency share and does not queue requests.

What's next

- You can visit flow control reference doc to learn more about troubleshooting.
- For background information on design details for API priority and fairness, see the enhancement proposal.
- You can make suggestions and feature requests via SIG API Machinery or the feature's slack channel.

Container Environment

This page describes the resources available to Containers in the Container environment.

Container environment

The Kubernetes Container environment provides several important resources to Containers:

- A filesystem, which is a combination of an image and one or more volumes.
- Information about the Container itself.
- Information about other objects in the cluster.

Container information

The *hostname* of a Container is the name of the Pod in which the Container is running. It is available through the `hostname` command or the `gethostname` function call in libc.

The Pod name and namespace are available as environment variables through the downward API.

User defined environment variables from the Pod definition are also available to the Container, as are any environment variables specified statically in the container image.

Cluster information

A list of all services that were running when a Container was created is available to that Container as environment variables. This list is limited to services within the same namespace as the new Container's Pod and Kubernetes control plane services.

For a service named *foo* that maps to a Container named *bar*, the following variables are defined:

```
FOO_SERVICE_HOST=<the host the service is running on>
FOO_SERVICE_PORT=<the port the service is running on>
```

Services have dedicated IP addresses and are available to the Container via DNS, if DNS addon is enabled.

What's next

- Learn more about Container lifecycle hooks.
- Get hands-on experience attaching handlers to Container lifecycle events.

Observability

Understand how to gain end-to-end visibility of a Kubernetes cluster through the collection of metrics, logs, and traces.

In Kubernetes, observability is the process of collecting and analyzing metrics, logs, and traces—often referred to as the three pillars of observability—in order to obtain a better understanding of the internal state, performance, and health of the cluster.

Kubernetes control plane components, as well as many add-ons, generate and emit these signals. By aggregating and correlating them, you can gain a unified picture of the control plane, add-ons, and applications across the cluster.

Figure 1 outlines how cluster components emit the three primary signal types.

> flowchart LR A[Cluster components] --> M[Metrics pipeline] A --> L[Log pipeline] A --> T[Trace pipeline] M --> S[(Storage and analysis)] L --> S T --> S S --> O[Operators and automation]

Figure 1. High-level signals emitted by cluster components and their consumers.

Metrics

Kubernetes components emit metrics in Prometheus format from their `/metrics` endpoints, including:

- kube-controller-manager
- kube-proxy
- kube-apiserver
- kube-scheduler
- kubelet

The kubelet also exposes metrics at `/metrics/cadvisor`, `/metrics/resource`, and `/metrics/probes`, and add-ons such as kube-state-metrics enrich those control plane signals with Kubernetes object status.

A typical Kubernetes metrics pipeline periodically scrapes these endpoints and stores the samples in a time series database (for example with Prometheus).

See the system metrics guide for details and configuration options.

Figure 2 outlines a common Kubernetes metrics pipeline.

> flowchart LR C[Cluster components] --> P[Prometheus scraper] P --> TS[(Time series storage)] TS --> D[Dashboards and alerts] TS --> A[Automated actions]

Figure 2. Components of a typical Kubernetes metrics pipeline.

For multi-cluster or multi-cloud visibility, distributed time series databases (for example Thanos or Cortex) can complement Prometheus.

See Common observability tools - metrics tools for metrics scrapers and time series databases.

See Also

- System metrics for Kubernetes components
- Resource usage monitoring with metrics-server
- kube-state-metrics concept
- Resource metrics pipeline overview

Logs

Logs provide a chronological record of events inside applications, Kubernetes system components, and security-related activities such as audit logging.

Container runtimes capture a containerized application’s output from standard output (`stdout`) and standard error (`stderr`) streams. While runtimes implement this differently, the integration with the kubelet is standardized through the *CRI logging format*, and the kubelet makes these logs available through `kubectl logs`.

Node-level logging

Figure 3a. Node-level logging architecture.

System component logs capture events from the cluster and are often useful for debugging and troubleshooting. These components are classified in two different ways: those that run in a container and those that do not. For example, the `kube-scheduler` and `kube-proxy` usually run in containers, whereas the `kubelet` and the container runtime run directly on the host.

- On machines with `systemd`, the kubelet and container runtime write to journald. Otherwise, they write to `.log` files in the `/var/log` directory.
- System components that run inside containers always write to `.log` files in `/var/log`, bypassing the default container logging mechanism.

System component and container logs stored under `/var/log` require log rotation to prevent uncontrolled growth. Some cluster provisioning scripts install log rotation by default; verify your environment and adjust as needed. See the system logs reference for details on locations, formats, and configuration options.

Most clusters run a node-level logging agent (for example, Fluent Bit or Fluentd) that tails these files and forwards entries to a central log store. The logging architecture guidance explains how to design such pipelines, apply retention, and log flows to backends.

Figure 3 outlines a common log aggregation pipeline.

> flowchart LR subgraph Sources A[Application stdout / stderr] B[Control plane logs] C[Audit records] end A --> N[Node log agent] B --> N C --> N N --> L[Central log store] L --> Q[Dashboards, alerting, SIEM]

Figure 3. Components of a typical Kubernetes logs pipeline.

See Common observability tools - logging tools for logging agents and central log stores.

See Also

- Logging architecture
- System logs
- Logging tasks and tutorials
- Configure audit logging

Traces

Traces capture how requests moves across Kubernetes components and applications, linking latency, timing and relationships between operations.By collecting traces, you can visualize end-to-end request flow, diagnose performance issues, and identify bottlenecks or unexpected interactions in the control plane, add-ons, or applications.

Kubernetes 1.34 can export spans over the OpenTelemetry Protocol (OTLP), either directly via built-in gRPC exporters or by forwarding them through an OpenTelemetry Collector.

The OpenTelemetry Collector receives spans from components and applications, processes them (for example by applying sampling or redaction), and forwards them to a tracing backend for storage and analysis.

Figure 4 outlines a typical distributed tracing pipeline.

> flowchart LR subgraph Sources A[Control plane spans] B[Application spans] end A --> X[OTLP exporter] B --> X X --> COL[OpenTelemetry Collector] COL --> TS[(Tracing backend)] TS --> V[Visualization and analysis]

Figure 4. Components of a typical Kubernetes traces pipeline.

See Common observability tools - tracing tools for tracing collectors and backends.

See Also

- System traces for Kubernetes components
- OpenTelemetry Collector getting started guide
- Monitoring and tracing tasks

Common observability tools

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.

Note: This section links to third-party projects that provide observability capabilities 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.

Metrics tools

- Cortex offers horizontally scalable, long-term Prometheus storage.
- Grafana Mimir is a Grafana Labs project that provides multi-tenant, horizontally scalable Prometheus-compatible storage.
- Prometheus is the monitoring system that scrapes and stores metrics from Kubernetes components.
- Thanos extends Prometheus with global querying, downsampling, and object storage support.

Logging tools

- Elasticsearch delivers distributed log indexing and search.
- Fluent Bit collects and forwards container and node logs with a low resource footprint.
- Fluentd routes and transforms logs to multiple destinations.
- Grafana Loki stores logs in a Prometheus-inspired, label-based format.
- OpenSearch provides open source log indexing and search compatible with Elasticsearch APIs.

Tracing tools

- Grafana Tempo offers scalable, low-cost distributed tracing storage.
- Jaeger captures and visualizes distributed traces for microservices.
- OpenTelemetry Collector receives, processes, and exports telemetry data including traces.
- Zipkin provides distributed tracing collection and visualization.

What's next

- Learn how to collect resource usage metrics with metrics-server
- Explore logging tasks and tutorials
- Follow the monitoring and tracing task guides
- Review the system metrics guide for component endpoints and stability
- Review the common observability tools section for vetted third-party options

Configuration Best Practices

This document highlights and consolidates configuration best practices that are introduced throughout the user guide, Getting Started documentation, and examples.

This is a living document. If you think of something that is not on this list but might be useful to others, please don't hesitate to file an issue or submit a PR.

General Configuration Tips

- When defining configurations, specify the latest stable API version.

- Configuration files should be stored in version control before being pushed to the cluster. This allows you to quickly roll back a configuration change if necessary. It also aids cluster re-creation and restoration.

- Write your configuration files using YAML rather than JSON. Though these formats can be used interchangeably in almost all scenarios, YAML tends to be more user-friendly.

- Group related objects into a single file whenever it makes sense. One file is often easier to manage than several. See the guestbook-all-in-one.yaml file as an example of this syntax.

- Note also that many `kubectl` commands can be called on a directory. For example, you can call `kubectl apply` on a directory of config files.

- Don't specify default values unnecessarily: simple, minimal configuration will make errors less likely.

- Put object descriptions in annotations, to allow better introspection.

Note:

There is a breaking change introduced in the YAML 1.2 boolean values specification with respect to YAML 1.1. This is a known issue in Kubernetes. YAML 1.2 only recognizes **true** and **false** as valid booleans, while YAML 1.1 also accepts **yes**, **no**, **on**, and **off** as booleans. However, Kubernetes uses YAML parsers that are mostly compatible with YAML 1.1, which means that using **yes** or **no** instead of **true** or **false** in a YAML manifest may cause unexpected errors or behaviors. To avoid this issue, it is recommended to always use **true** or **false** for boolean values in YAML manifests, and to quote any strings that may be confused with booleans, such as **"yes"** or **"no"**.

Besides booleans, there are additional specifications changes between YAML versions. Please refer to the YAML Specification Changes documentation for a comprehensive list.

"Naked" Pods versus ReplicaSets, Deployments, and Jobs

- Don't use naked Pods (that is, Pods not bound to a ReplicaSet or Deployment) if you can avoid it. Naked Pods will not be rescheduled in the event of a node failure.

  A Deployment, which both creates a ReplicaSet to ensure that the desired number of Pods is always available, and specifies a strategy to replace Pods (such as RollingUpdate), is almost always preferable to creating Pods directly, except for some explicit `restartPolicy: Never` scenarios. A Job may also be appropriate.

Services

- Create a Service before its corresponding backend workloads (Deployments or ReplicaSets), and before any workloads that need to access it. When Kubernetes starts a container, it provides environment variables pointing to all the Services which were running when the container was started. For example, if a Service named `foo` exists, all containers will get the following variables in their initial environment:

  ```
  FOO_SERVICE_HOST=<the host the Service is running on>
  FOO_SERVICE_PORT=<the port the Service is running on>
  ```

  This does imply an ordering requirement - any `Service` that a `Pod` wants to access must be created before the `Pod` itself, or else the environment variables will not be populated. DNS does not have this restriction.

- An optional (though strongly recommended) cluster add-on is a DNS server. The DNS server watches the Kubernetes API for new `Services` and creates a set of DNS records for each. If DNS has been enabled throughout the cluster then all `Pods` should be able to do name resolution of `Services` automatically.

- Don't specify a `hostPort` for a Pod unless it is absolutely necessary. When you bind a Pod to a `hostPort`, it limits the number of places the Pod can be scheduled, because each <`hostIP`, `hostPort`, `protocol`> combination must be unique. If you don't specify the `hostIP` and `protocol` explicitly, Kubernetes will use `0.0.0.0` as the default `hostIP` and `TCP` as the default `protocol`.

  If you only need access to the port for debugging purposes, you can use the apiserver proxy or `kubectl port-forward`.

  If you explicitly need to expose a Pod's port on the node, consider using a NodePort Service before resorting to `hostPort`.

- Avoid using `hostNetwork`, for the same reasons as `hostPort`.

- Use headless Services (which have a `ClusterIP` of `None`) for service discovery when you don't need `kube-proxy` load balancing.

Using Labels

- Define and use labels that identify **semantic attributes** of your application or Deployment, such as `{ app.kubernetes.io/name: MyApp, tier: frontend, phase: test, deployment: v3 }`. You can use these labels to select the appropriate Pods for other resources; for example, a Service that selects all `tier: frontend` Pods, or all `phase: test` components of `app.kubernetes.io/name: MyApp`. See the guestbook app for examples of this approach.

  A Service can be made to span multiple Deployments by omitting release-specific labels from its selector. When you need to update a running service without downtime, use a Deployment.

  A desired state of an object is described by a Deployment, and if changes to that spec are *applied*, the deployment controller changes the actual state to the desired state at a controlled rate.

- Use the Kubernetes common labels for common use cases. These standardized labels enrich the metadata in a way that allows tools, including `kubectl` and dashboard, to work in an interoperable way.

- You can manipulate labels for debugging. Because Kubernetes controllers (such as ReplicaSet) and Services match to Pods using selector labels, removing the relevant labels from a Pod will stop it from being considered by a controller or from being served traffic by a Service. If you remove the labels of an existing Pod, its controller will create a new Pod to take its place. This is a useful way to debug a previously "live" Pod in a "quarantine" environment. To interactively remove or add labels, use `kubectl label`.

Using kubectl

- Use `kubectl apply -f <directory>`. This looks for Kubernetes configuration in all `.yaml`, `.yml`, and `.json` files in `<directory>` and passes it to `apply`.

- Use label selectors for `get` and `delete` operations instead of specific object names. See the sections on label selectors and using labels effectively.

- Use `kubectl create deployment` and `kubectl expose` to quickly create single-container Deployments and Services. See Use a Service to Access an Application in a Cluster for an example.

Installing Addons

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.

Add-ons extend the functionality of Kubernetes.

This page lists some of the available add-ons and links to their respective installation instructions. The list does not try to be exhaustive.

Networking and Network Policy

- ACI provides integrated container networking and network security with Cisco ACI.
- Antrea operates at Layer 3/4 to provide networking and security services for Kubernetes, leveraging Open vSwitch as the networking data plane. Antrea is a CNCF project at the Sandbox level.
- Calico is a networking and network policy provider. Calico supports a flexible set of networking options so you can choose the most efficient option for your situation, including non-overlay and overlay networks, with or without BGP. Calico uses the same engine to enforce network policy for hosts, pods, and (if using Istio & Envoy) applications at the service mesh layer.
- Canal unites Flannel and Calico, providing networking and network policy.
- Cilium is a networking, observability, and security solution with an eBPF-based data plane. Cilium provides a simple flat Layer 3 network with the ability to span multiple clusters in either a native routing or overlay/encapsulation mode, and can enforce network policies on L3-L7 using an identity-based security model that is decoupled from network addressing. Cilium can act as a replacement for kube-proxy; it also offers additional, opt-in observability and security features. Cilium is a CNCF project at the Graduated level.

- CNI-Genie enables Kubernetes to seamlessly connect to a choice of CNI plugins, such as Calico, Canal, Flannel, or Weave. CNI-Genie is a CNCF project at the Sandbox level.
- Contiv provides configurable networking (native L3 using BGP, overlay using vxlan, classic L2, and Cisco-SDN/ACI) for various use cases and a rich policy framework. Contiv project is fully open sourced. The installer provides both kubeadm and non-kubeadm based installation options.
- Contrail, based on Tungsten Fabric, is an open source, multi-cloud network virtualization and policy management platform. Contrail and Tungsten Fabric are integrated with orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provide isolation modes for virtual machines, containers/pods and bare metal workloads.
- Flannel is an overlay network provider that can be used with Kubernetes.
- Gateway API is an open source project managed by the SIG Network community and provides an expressive, extensible, and role-oriented API for modeling service networking.
- Knitter is a plugin to support multiple network interfaces in a Kubernetes pod.
- Multus is a Multi plugin for multiple network support in Kubernetes to support all CNI plugins (e.g. Calico, Cilium, Contiv, Flannel), in addition to SRIOV, DPDK, OVS-DPDK and VPP based workloads in Kubernetes.
- OVN-Kubernetes is a networking provider for Kubernetes based on OVN (Open Virtual Network), a virtual networking implementation that came out of the Open vSwitch (OVS) project. OVN-Kubernetes provides an overlay based networking implementation for Kubernetes, including an OVS based implementation of load balancing and network policy.
- Nodus is an OVN based CNI controller plugin to provide cloud native based Service function chaining(SFC).
- NSX-T Container Plug-in (NCP) provides integration between VMware NSX-T and container orchestrators such as Kubernetes, as well as integration between NSX-T and container-based CaaS/PaaS platforms such as Pivotal Container Service (PKS) and OpenShift.
- Nuage is an SDN platform that provides policy-based networking between Kubernetes Pods and non-Kubernetes environments with visibility and security monitoring.
- Romana is a Layer 3 networking solution for pod networks that also supports the NetworkPolicy API.
- Spiderpool is an underlay and RDMA networking solution for Kubernetes. Spiderpool is supported on bare metal, virtual machines, and public cloud environments.
- Terway is a suite of CNI plugins based on AlibabaCloud's VPC and ECS network products. It provides native VPC networking and network policies in AlibabaCloud environments.
- Weave Net provides networking and network policy, will carry on working on both sides of a network partition, and does not require an external database.

Service Discovery

- CoreDNS is a flexible, extensible DNS server which can be installed as the in-cluster DNS for pods.

Visualization & Control

- Dashboard is a dashboard web interface for Kubernetes.

Infrastructure

- KubeVirt is an add-on to run virtual machines on Kubernetes. Usually run on bare-metal clusters.
- The node problem detector runs on Linux nodes and reports system issues as either Events or Node conditions.

Instrumentation

- kube-state-metrics

Legacy Add-ons

There are several other add-ons documented in the deprecated cluster/addons directory.

Well-maintained ones should be linked to here. PRs welcome!

Coordinated Leader Election

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

Kubernetes 1.34 includes a beta feature that allows control plane components to deterministically select a leader via *coordinated leader election*. This is useful to satisfy Kubernetes version skew constraints during cluster upgrades. Currently, the only builtin selection strategy is `OldestEmulationVersion`, preferring the leader with the lowest emulation version, followed by binary version, followed by creation timestamp.

Enabling coordinated leader election

Ensure that `CoordinatedLeaderElection` feature gate is enabled when you start the API Server: and that the `coordination.k8s.io/v1beta1` API group is enabled.

This can be done by setting flags `--feature-gates="CoordinatedLeaderElection=true"` and `--runtime-config="coordination.k8s.io/v1beta1=true"`.

Component configuration

Provided that you have enabled the `CoordinatedLeaderElection` feature gate *and* have the `coordination.k8s.io/v1beta1` API group enabled, compatible control plane components automatically use the LeaseCandidate and Lease APIs to elect a leader as needed.

For Kubernetes 1.34, two control plane components
(kube-controller-manager and kube-scheduler) automatically use coordinated
leader election when the feature gate and API group are enabled.

Certificates

To learn how to generate certificates for your cluster, see Certificates.

Metrics For Kubernetes System Components

System component metrics can give a better look into what is happening inside them. Metrics are particularly useful for building dashboards and alerts.

Kubernetes components emit metrics in Prometheus format. This format is structured plain text, designed so that people and machines can both read it.

Metrics in Kubernetes

In most cases metrics are available on `/metrics` endpoint of the HTTP server. For components that don't expose endpoint by default, it can be enabled using `--bind-address` flag.

Examples of those components:

- kube-controller-manager
- kube-proxy
- kube-apiserver
- kube-scheduler
- kubelet

In a production environment you may want to configure Prometheus Server or some other metrics scraper to periodically gather these metrics and make them available in some kind of time series database.

Note that kubelet also exposes metrics in `/metrics/cadvisor`, `/metrics/resource` and `/metrics/probes` endpoints. Those metrics do not have the same lifecycle.

If your cluster uses RBAC, reading metrics requires authorization via a user, group or ServiceAccount with a ClusterRole that allows accessing `/metrics`. For example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: prometheusrules:  - nonResourceURLs:      - "/metrics"    verbs:      - get
```

Metric lifecycle

Alpha metric → Beta metric → Stable metric → Deprecated metric → Hidden metric → Deleted metric

Alpha metrics have no stability guarantees. These metrics can be modified or deleted at any time.

Beta metrics observe a looser API contract than its stable counterparts. No labels can be removed from beta metrics during their lifetime, however, labels can be added while the metric is in the beta stage.

Stable metrics are guaranteed to not change. This means:

- A stable metric without a deprecated signature will not be deleted or renamed
- A stable metric's type will not be modified

Deprecated metrics are slated for deletion, but are still available for use. These metrics include an annotation about the version in which they became deprecated.

For example:

- Before deprecation

  ```
  # HELP some_counter this counts things
  # TYPE some_counter counter
  some_counter 0
  ```

- After deprecation

  ```
  # HELP some_counter (Deprecated since 1.15.0) this counts things
  # TYPE some_counter counter
  some_counter 0
  ```

Hidden metrics are no longer published for scraping, but are still available for use. A deprecated metric becomes a hidden metric after a period of time, based on its stability level:

- **STABLE** metrics become hidden after a minimum of 3 releases or 9 months, whichever is longer.
- **BETA** metrics become hidden after a minimum of 1 release or 4 months, whichever is longer.
- **ALPHA** metrics can be hidden or removed in the same release in which they are deprecated.

To use a hidden metric, you must enable it. For more details, refer to the Show hidden metrics section.

Deleted metrics are no longer published and cannot be used.

Show hidden metrics

As described above, admins can enable hidden metrics through a command-line flag on a specific binary. This intends to be used as an escape hatch for admins if they missed the migration of the metrics deprecated in the last release.

The flag `show-hidden-metrics-for-version` takes a version for which you want to show metrics deprecated in that release. The version is expressed as x.y, where x is the major version, y is the minor version. The patch version is not needed even though a metrics can be deprecated in a patch release, the reason for that is the metrics deprecation policy runs against the minor release.

The flag can only take the previous minor version as its value. If you want to show all metrics hidden in the previous release, you can set the `show-hidden-metrics-for-version` flag to the previous version. Using a version that is too old is not allowed because it violates the metrics deprecation policy.

For example, let's assume metric `A` is deprecated in `1.29`. The version in which metric `A` becomes hidden depends on its stability level:

- If metric `A` is **ALPHA**, it could be hidden in `1.29`.
- If metric `A` is **BETA**, it will be hidden in `1.30` at the earliest. If you are upgrading to `1.30` and still need `A`, you must use the command-line flag `--show-hidden-metrics-for-version=1.29`.
- If metric `A` is **STABLE**, it will be hidden in `1.32` at the earliest. If you are upgrading to `1.32` and still need `A`, you must use the command-line flag `--show-hidden-metrics-for-version=1.31`.

Component metrics

kube-controller-manager metrics

Controller manager metrics provide important insight into the performance and health of the controller manager. These metrics include common Go language runtime metrics such as go_routine count and controller specific metrics such as etcd request latencies or Cloudprovider (AWS, GCE, OpenStack) API latencies that can be used to gauge the health of a cluster.

Starting from Kubernetes 1.7, detailed Cloudprovider metrics are available for storage operations for GCE, AWS, Vsphere and OpenStack. These metrics can be used to monitor health of persistent volume operations.

For example, for GCE these metrics are called:

```
cloudprovider_gce_api_request_duration_seconds { request = "instance_list"}
cloudprovider_gce_api_request_duration_seconds { request = "disk_insert"}
cloudprovider_gce_api_request_duration_seconds { request = "disk_delete"}
cloudprovider_gce_api_request_duration_seconds { request = "attach_disk"}
cloudprovider_gce_api_request_duration_seconds { request = "detach_disk"}
cloudprovider_gce_api_request_duration_seconds { request = "list_disk"}
```

kube-scheduler metrics

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

The scheduler exposes optional metrics that reports the requested resources and the desired limits of all running pods. These metrics can be used to build capacity planning dashboards, assess current or historical scheduling limits, quickly identify workloads that cannot schedule due to lack of resources, and compare actual usage to the pod's request.

The kube-scheduler identifies the resource requests and limits configured for each Pod; when either a request or limit is non-zero, the kube-scheduler reports a metrics timeseries. The time series is labelled by:

- namespace
- pod name
- the node where the pod is scheduled or an empty string if not yet scheduled
- priority
- the assigned scheduler for that pod
- the name of the resource (for example, `cpu`)
- the unit of the resource if known (for example, `cores`)

Once a pod reaches completion (has a `restartPolicy` of `Never` or `OnFailure` and is in the `Succeeded` or `Failed` pod phase, or has been deleted and all containers have a terminated state) the series is no longer reported since the scheduler is now free to schedule other pods to run. The two metrics are called `kube_pod_resource_request` and `kube_pod_resource_limit`.

The metrics are exposed at the HTTP endpoint `/metrics/resources`. They require authorization for the `/metrics/resources` endpoint, usually granted by a ClusterRole with the `get` verb for the `/metrics/resources` non-resource URL.

On Kubernetes 1.21 you must use the `--show-hidden-metrics-for-version=1.20` flag to expose these alpha stability metrics.

kubelet Pressure Stall Information (PSI) metrics

FEATURE STATE: `Kubernetes v1.34 [beta]`

As a beta feature, Kubernetes lets you configure kubelet to collect Linux kernel Pressure Stall Information (PSI) for CPU, memory and I/O usage. The information is collected at node, pod and container level. The metrics are exposed at the `/metrics/cadvisor` endpoint with the following names:

```
container_pressure_cpu_stalled_seconds_total
container_pressure_cpu_waiting_seconds_total
container_pressure_memory_stalled_seconds_total
container_pressure_memory_waiting_seconds_total
```

```
container_pressure_io_stalled_seconds_total
container_pressure_io_waiting_seconds_total
```

This feature is enabled by default, by setting the `KubeletPSI` feature gate. The information is also exposed in the Summary API.

You can learn how to interpret the PSI metrics in Understand PSI Metrics.

Requirements

Pressure Stall Information requires:

- Linux kernel versions 4.20 or later.
- cgroup v2

Disabling metrics

You can explicitly turn off metrics via command line flag `--disabled-metrics`. This may be desired if, for example, a metric is causing a performance problem. The input is a list of disabled metrics (i.e. `--disabled-metrics=metric1,metric2`).

Metric cardinality enforcement

Metrics with unbounded dimensions could cause memory issues in the components they instrument. To limit resource use, you can use the `--allow-metric-labels` command line option to dynamically configure an allow-list of label values for a metric.

In alpha stage, the flag can only take in a series of mappings as metric label allow-list. Each mapping is of the format `<metric_name>,<label_name>=<allowed_labels>` where `<allowed_labels>` is a comma-separated list of acceptable label names.

The overall format looks like:

```
--allow-metric-labels <metric_name>,<label_name>='<allow_value1>, <allow_value2>...', <metric_name2>,<label_name>='<allow_value1>,
```

Here is an example:

```
--allow-metric-labels number_count_metric,odd_number='1,3,5', number_count_metric,even_number='2,4,6', date_gauge_metric,weekend='
```

In addition to specifying this from the CLI, this can also be done within a configuration file. You can specify the path to that configuration file using the `--allow-metric-labels-manifest` command line argument to a component. Here's an example of the contents of that configuration file:

```
"metric1,label2": "v1,v2,v3"
"metric2,label1": "v1,v2,v3"
```

Additionally, the `cardinality_enforcement_unexpected_categorizations_total` meta-metric records the count of unexpected categorizations during cardinality enforcement, that is, whenever a label value is encountered that is not allowed with respect to the allow-list constraints.

What's next

- Read about the Prometheus text format for metrics
- See the list of stable Kubernetes metrics
- Read about the Kubernetes deprecation policy

Secrets

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image. Using a Secret means that you don't need to include confidential data in your application code.

Because Secrets can be created independently of the Pods that use them, there is less risk of the Secret (and its data) being exposed during the workflow of creating, viewing, and editing Pods. Kubernetes, and applications that run in your cluster, can also take additional precautions with Secrets, such as avoiding writing sensitive data to nonvolatile storage.

Secrets are similar to ConfigMaps but are specifically intended to hold confidential data.

Caution:

Kubernetes Secrets are, by default, stored unencrypted in the API server's underlying data store (etcd). Anyone with API access can retrieve or modify a Secret, and so can anyone with access to etcd. Additionally, anyone who is authorized to create a Pod in a namespace can use that access to read any Secret in that namespace; this includes indirect access such as the ability to create a Deployment.

In order to safely use Secrets, take at least the following steps:

1. Enable Encryption at Rest for Secrets.
2. Enable or configure RBAC rules with least-privilege access to Secrets.
3. Restrict Secret access to specific containers.
4. Consider using external Secret store providers.

For more guidelines to manage and improve the security of your Secrets, refer to Good practices for Kubernetes Secrets.

See Information security for Secrets for more details.

Uses for Secrets

You can use Secrets for purposes such as the following:

- Set environment variables for a container.
- Provide credentials such as SSH keys or passwords to Pods.
- Allow the kubelet to pull container images from private registries.

The Kubernetes control plane also uses Secrets; for example, bootstrap token Secrets are a mechanism to help automate node registration.

Use case: dotfiles in a secret volume

You can make your data "hidden" by defining a key that begins with a dot. This key represents a dotfile or "hidden" file. For example, when the following Secret is mounted into a volume, `secret-volume`, the volume will contain a single file, called `.secret-file`, and the `dotfile-test-container` will have this file present at the path `/etc/secret-volume/.secret-file`.

Note:

Files beginning with dot characters are hidden from the output of `ls -l`; you must use `ls -la` to see them when listing directory contents.
`secret/dotfile-secret.yaml` Copy secret/dotfile-secret.yaml to clipboard

```
apiVersion: v1
kind: Secretmetadata:  name: dotfile-secretdata:  .secret-file: dmFsdWUtMg0KDQo=---apiVersion: v1kind: Podmetadata:  name: secret-
```

Use case: Secret visible to one container in a Pod

Consider a program that needs to handle HTTP requests, do some complex business logic, and then sign some messages with an HMAC. Because it has complex application logic, there might be an unnoticed remote file reading exploit in the server, which could expose the private key to an attacker.

This could be divided into two processes in two containers: a frontend container which handles user interaction and business logic, but which cannot see the private key; and a signer container that can see the private key, and responds to simple signing requests from the frontend (for example, over localhost networking).

With this partitioned approach, an attacker now has to trick the application server into doing something rather arbitrary, which may be harder than getting it to read a file.

Alternatives to Secrets

Rather than using a Secret to protect confidential data, you can pick from alternatives.

Here are some of your options:

- If your cloud-native component needs to authenticate to another application that you know is running within the same Kubernetes cluster, you can use a ServiceAccount and its tokens to identify your client.
- There are third-party tools that you can run, either within or outside your cluster, that manage sensitive data. For example, a service that Pods access over HTTPS, that reveals a Secret if the client correctly authenticates (for example, with a ServiceAccount token).
- For authentication, you can implement a custom signer for X.509 certificates, and use CertificateSigningRequests to let that custom signer issue certificates to Pods that need them.
- You can use a device plugin to expose node-local encryption hardware to a specific Pod. For example, you can schedule trusted Pods onto nodes that provide a Trusted Platform Module, configured out-of-band.

You can also combine two or more of those options, including the option to use Secret objects themselves.

For example: implement (or deploy) an operator that fetches short-lived session tokens from an external service, and then creates Secrets based on those short-lived session tokens. Pods running in your cluster can make use of the session tokens, and operator ensures they are valid. This separation means that you can run Pods that are unaware of the exact mechanisms for issuing and refreshing those session tokens.

Types of Secret

When creating a Secret, you can specify its type using the `type` field of the Secret resource, or certain equivalent `kubectl` command line flags (if available). The Secret type is used to facilitate programmatic handling of the Secret data.

Kubernetes provides several built-in types for some common usage scenarios. These types vary in terms of the validations performed and the constraints Kubernetes imposes on them.

Built-in Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	ServiceAccount token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

You can define and use your own Secret type by assigning a non-empty string as the `type` value for a Secret object (an empty string is treated as an `Opaque` type).

Kubernetes doesn't impose any constraints on the type name. However, if you are using one of the built-in types, you must meet all the requirements defined for that type.

If you are defining a type of Secret that's for public use, follow the convention and structure the Secret type to have your domain name before the name, separated by a `/`. For example: `cloud-hosting.example.net/cloud-api-credentials`.

Opaque Secrets

`Opaque` is the default Secret type if you don't explicitly specify a type in a Secret manifest. When you create a Secret using `kubectl`, you must use the `generic` subcommand to indicate an `Opaque` Secret type. For example, the following command creates an empty Secret of type `Opaque`:

```
kubectl create secret generic empty-secret
kubectl get secret empty-secret
```

The output looks like:

```
NAME           TYPE     DATA   AGE
empty-secret   Opaque   0      2m6s
```

The `DATA` column shows the number of data items stored in the Secret. In this case, `0` means you have created an empty Secret.

ServiceAccount token Secrets

A `kubernetes.io/service-account-token` type of Secret is used to store a token credential that identifies a ServiceAccount. This is a legacy mechanism that provides long-lived ServiceAccount credentials to Pods.

In Kubernetes v1.22 and later, the recommended approach is to obtain a short-lived, automatically rotating ServiceAccount token by using the `TokenRequest` API instead. You can get these short-lived tokens using the following methods:

- Call the `TokenRequest` API either directly or by using an API client like `kubectl`. For example, you can use the `kubectl create token` command.
- Request a mounted token in a projected volume in your Pod manifest. Kubernetes creates the token and mounts it in the Pod. The token is automatically invalidated when the Pod that it's mounted in is deleted. For details, see Launch a Pod using service account token projection.

Note:

You should only create a ServiceAccount token Secret if you can't use the `TokenRequest` API to obtain a token, and the security exposure of persisting a non-expiring token credential in a readable API object is acceptable to you. For instructions, see Manually create a long-lived API token for a ServiceAccount.

When using this Secret type, you need to ensure that the `kubernetes.io/service-account.name` annotation is set to an existing ServiceAccount name. If you are creating both the ServiceAccount and the Secret objects, you should create the ServiceAccount object first.

After the Secret is created, a Kubernetes controller fills in some other fields such as the `kubernetes.io/service-account.uid` annotation, and the `token` key in the `data` field, which is populated with an authentication token.

The following example configuration declares a ServiceAccount token Secret:

`secret/serviceaccount-token-secret.yaml` Copy secret/serviceaccount-token-secret.yaml to clipboard

```
apiVersion: v1
kind: Secretmetadata:  name: secret-sa-sample  annotations:    kubernetes.io/service-account.name: "sa-name"type: kubernetes.io/se
```

After creating the Secret, wait for Kubernetes to populate the `token` key in the `data` field.

See the ServiceAccount documentation for more information on how ServiceAccounts work. You can also check the `automountServiceAccountToken` field and the `serviceAccountName` field of the `Pod` for information on referencing ServiceAccount credentials from within Pods.

Docker config Secrets

If you are creating a Secret to store credentials for accessing a container image registry, you must use one of the following `type` values for that Secret:

- `kubernetes.io/dockercfg`: store a serialized `~/.dockercfg` which is the legacy format for configuring Docker command line. The Secret `data` field contains a `.dockercfg` key whose value is the content of a base64 encoded `~/.dockercfg` file.
- `kubernetes.io/dockerconfigjson`: store a serialized JSON that follows the same format rules as the `~/.docker/config.json` file, which is a new format for `~/.dockercfg`. The Secret `data` field must contain a `.dockerconfigjson` key for which the value is the content of a base64 encoded `~/.docker/config.json` file.

Below is an example for a `kubernetes.io/dockercfg` type of Secret:

`secret/dockercfg-secret.yaml` Copy secret/dockercfg-secret.yaml to clipboard

```
apiVersion: v1
kind: Secretmetadata:  name: secret-dockercfgtype: kubernetes.io/dockercfgdata:  .dockercfg: |    eyJhdXRocyI6eyJodHRwczovL2V4YW1w
```

Note:

If you do not want to perform the base64 encoding, you can choose to use the `stringData` field instead.

When you create Docker config Secrets using a manifest, the API server checks whether the expected key exists in the `data` field, and it verifies if the value provided can be parsed as a valid JSON. The API server doesn't validate if the JSON actually is a Docker config file.

You can also use `kubectl` to create a Secret for accessing a container registry, such as when you don't have a Docker configuration file:

```
kubectl create secret docker-registry secret-tiger-docker \
  --docker-email=tiger@acme.example \
  --docker-username=tiger \  --docker-password=pass1234 \  --docker-server=my-registry.example:5000
```

This command creates a Secret of type `kubernetes.io/dockerconfigjson`.

Retrieve the `.data.dockerconfigjson` field from that new Secret and decode the data:

```
kubectl get secret secret-tiger-docker -o jsonpath='{.data.*}' | base64 -d
```

The output is equivalent to the following JSON document (which is also a valid Docker configuration file):

```
{
  "auths": {
    "my-registry.example:5000": {
      "username": "tiger",
      "password": "pass1234",
      "email": "tiger@acme.example",
      "auth": "dGlnZXI6cGFzczEyMzQ="
    }
  }
}
```

Caution:

The `auth` value there is base64 encoded; it is obscured but not secret. Anyone who can read that Secret can learn the registry access bearer token.

It is suggested to use credential providers to dynamically and securely provide pull secrets on-demand.

Basic authentication Secret

The `kubernetes.io/basic-auth` type is provided for storing credentials needed for basic authentication. When using this Secret type, the `data` field of the Secret must contain one of the following two keys:

- `username`: the user name for authentication
- `password`: the password or token for authentication

Both values for the above two keys are base64 encoded strings. You can alternatively provide the clear text content using the `stringData` field in the Secret manifest.

The following manifest is an example of a basic authentication Secret:

secret/basicauth-secret.yaml Copy secret/basicauth-secret.yaml to clipboard

```
apiVersion: v1
kind: Secretmetadata:  name: secret-basic-authtype: kubernetes.io/basic-authstringData:  username: admin # required field for kube
```

Note:

The `stringData` field for a Secret does not work well with server-side apply.

The basic authentication Secret type is provided only for convenience. You can create an `Opaque` type for credentials used for basic authentication. However, using the defined and public Secret type (`kubernetes.io/basic-auth`) helps other people to understand the purpose of your Secret, and sets a convention for what key names to expect.

SSH authentication Secrets

The builtin type `kubernetes.io/ssh-auth` is provided for storing data used in SSH authentication. When using this Secret type, you will have to specify a `ssh-privatekey` key-value pair in the `data` (or `stringData`) field as the SSH credential to use.

The following manifest is an example of a Secret used for SSH public/private key authentication:

secret/ssh-auth-secret.yaml Copy secret/ssh-auth-secret.yaml to clipboard

```
apiVersion: v1
kind: Secretmetadata:  name: secret-ssh-authtype: kubernetes.io/ssh-authdata:  # the data is abbreviated in this example  ssh-priv
```

The SSH authentication Secret type is provided only for convenience. You can create an `Opaque` type for credentials used for SSH authentication. However, using the defined and public Secret type (`kubernetes.io/ssh-auth`) helps other people to understand the purpose of your Secret, and sets a convention for what key names to expect. The Kubernetes API verifies that the required keys are set for a Secret of this type.

Caution:

SSH private keys do not establish trusted communication between an SSH client and host server on their own. A secondary means of establishing trust is needed to mitigate "man in the middle" attacks, such as a `known_hosts` file added to a ConfigMap.

TLS Secrets

The `kubernetes.io/tls` Secret type is for storing a certificate and its associated key that are typically used for TLS.

One common use for TLS Secrets is to configure encryption in transit for an Ingress, but you can also use it with other resources or directly in your workload. When using this type of Secret, the `tls.key` and the `tls.crt` key must be provided in the `data` (or `stringData`) field of the Secret configuration, although the API server doesn't actually validate the values for each key.

As an alternative to using `stringData`, you can use the `data` field to provide the base64 encoded certificate and private key. For details, see [Constraints on Secret names and data](#).

The following YAML contains an example config for a TLS Secret:

[secret/tls-auth-secret.yaml](#) Copy secret/tls-auth-secret.yaml to clipboard

```
apiVersion: v1
kind: Secretmetadata:  name: secret-tlstype: kubernetes.io/tlsdata:  # values are base64 encoded, which obscures them but does NOT
```

The TLS Secret type is provided only for convenience. You can create an `Opaque` type for credentials used for TLS authentication. However, using the defined and public Secret type (`kubernetes.io/tls`) helps ensure the consistency of Secret format in your project. The API server verifies if the required keys are set for a Secret of this type.

To create a TLS Secret using `kubectl`, use the `tls` subcommand:

```
kubectl create secret tls my-tls-secret \
  --cert=path/to/cert/file \  --key=path/to/key/file
```

The public/private key pair must exist before hand. The public key certificate for `--cert` must be .PEM encoded and must match the given private key for `--key`.

Bootstrap token Secrets

The `bootstrap.kubernetes.io/token` Secret type is for tokens used during the node bootstrap process. It stores tokens used to sign well-known ConfigMaps.

A bootstrap token Secret is usually created in the `kube-system` namespace and named in the form `bootstrap-token-<token-id>` where `<token-id>` is a 6 character string of the token ID.

As a Kubernetes manifest, a bootstrap token Secret might look like the following:

[secret/bootstrap-token-secret-base64.yaml](#) Copy secret/bootstrap-token-secret-base64.yaml to clipboard

```
apiVersion: v1
kind: Secretmetadata:  name: bootstrap-token-5emitj  namespace: kube-systemtype: bootstrap.kubernetes.io/tokendata:  auth-extra-gr
```

A bootstrap token Secret has the following keys specified under `data`:

- `token-id`: A random 6 character string as the token identifier. Required.
- `token-secret`: A random 16 character string as the actual token Secret. Required.
- `description`: A human-readable string that describes what the token is used for. Optional.
- `expiration`: An absolute UTC time using [RFC3339](#) specifying when the token should be expired. Optional.
- `usage-bootstrap-<usage>`: A boolean flag indicating additional usage for the bootstrap token.
- `auth-extra-groups`: A comma-separated list of group names that will be authenticated as in addition to the `system:bootstrappers` group.

You can alternatively provide the values in the `stringData` field of the Secret without base64 encoding them:

[secret/bootstrap-token-secret-literal.yaml](#) Copy secret/bootstrap-token-secret-literal.yaml to clipboard

```
apiVersion: v1
kind: Secretmetadata:  # Note how the Secret is named  name: bootstrap-token-5emitj  # A bootstrap token Secret usually resides in
```

Note:

The `stringData` field for a Secret does not work well with server-side apply.

Working with Secrets

Creating a Secret

There are several options to create a Secret:

- [Use `kubectl`](#)
- [Use a configuration file](#)
- [Use the Kustomize tool](#)

Constraints on Secret names and data

The name of a Secret object must be a valid [DNS subdomain name](#).

You can specify the `data` and/or the `stringData` field when creating a configuration file for a Secret. The `data` and the `stringData` fields are optional. The values for all keys in the `data` field have to be base64-encoded strings. If the conversion to base64 string is not desirable, you can choose to specify the `stringData` field instead, which accepts arbitrary strings as values.

The keys of `data` and `stringData` must consist of alphanumeric characters, `-`, `_` or `.`. All key-value pairs in the `stringData` field are internally merged into the `data` field. If a key appears in both the `data` and the `stringData` field, the value specified in the `stringData` field takes precedence.

Size limit

Individual Secrets are limited to 1MiB in size. This is to discourage creation of very large Secrets that could exhaust the API server and kubelet memory. However, creation of many smaller Secrets could also exhaust memory. You can use a [resource quota](#) to limit the number of Secrets (or other resources) in a namespace.

Editing a Secret

You can edit an existing Secret unless it is immutable. To edit a Secret, use one of the following methods:

- Use `kubectl`
- Use a configuration file

You can also edit the data in a Secret using the Kustomize tool. However, this method creates a new `Secret` object with the edited data.

Depending on how you created the Secret, as well as how the Secret is used in your Pods, updates to existing `Secret` objects are propagated automatically to Pods that use the data. For more information, refer to Using Secrets as files from a Pod section.

Using a Secret

Secrets can be mounted as data volumes or exposed as environment variables to be used by a container in a Pod. Secrets can also be used by other parts of the system, without being directly exposed to the Pod. For example, Secrets can hold credentials that other parts of the system should use to interact with external systems on your behalf.

Secret volume sources are validated to ensure that the specified object reference actually points to an object of type Secret. Therefore, a Secret needs to be created before any Pods that depend on it.

If the Secret cannot be fetched (perhaps because it does not exist, or due to a temporary lack of connection to the API server) the kubelet periodically retries running that Pod. The kubelet also reports an Event for that Pod, including details of the problem fetching the Secret.

Optional Secrets

When you reference a Secret in a Pod, you can mark the Secret as *optional*, such as in the following example. If an optional Secret doesn't exist, Kubernetes ignores it.

`secret/optional-secret.yaml` Copy secret/optional-secret.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: mypodspec:  containers:  - name: mypod    image: redis    volumeMounts:    - name: foo      mountPath: "
```

By default, Secrets are required. None of a Pod's containers will start until all non-optional Secrets are available.

If a Pod references a specific key in a non-optional Secret and that Secret does exist, but is missing the named key, the Pod fails during startup.

Using Secrets as files from a Pod

If you want to access data from a Secret in a Pod, one way to do that is to have Kubernetes make the value of that Secret be available as a file inside the filesystem of one or more of the Pod's containers.

For instructions, refer to Create a Pod that has access to the secret data through a Volume.

When a volume contains data from a Secret, and that Secret is updated, Kubernetes tracks this and updates the data in the volume, using an eventually-consistent approach.

Note:

A container using a Secret as a subPath volume mount does not receive automated Secret updates.

The kubelet keeps a cache of the current keys and values for the Secrets that are used in volumes for pods on that node. You can configure the way that the kubelet detects changes from the cached values. The `configMapAndSecretChangeDetectionStrategy` field in the kubelet configuration controls which strategy the kubelet uses. The default strategy is `Watch`.

Updates to Secrets can be either propagated by an API watch mechanism (the default), based on a cache with a defined time-to-live, or polled from the cluster API server on each kubelet synchronisation loop.

As a result, the total delay from the moment when the Secret is updated to the moment when new keys are projected to the Pod can be as long as the kubelet sync period + cache propagation delay, where the cache propagation delay depends on the chosen cache type (following the same order listed in the previous paragraph, these are: watch propagation delay, the configured cache TTL, or zero for direct polling).

Using Secrets as environment variables

To use a Secret in an environment variable in a Pod:

1. For each container in your Pod specification, add an environment variable for each Secret key that you want to use to the `env[].valueFrom.secretKeyRef` field.
2. Modify your image and/or command line so that the program looks for values in the specified environment variables.

For instructions, refer to Define container environment variables using Secret data.

It's important to note that the range of characters allowed for environment variable names in pods is restricted. If any keys do not meet the rules, those keys are not made available to your container, though the Pod is allowed to start.

Container image pull Secrets

If you want to fetch container images from a private repository, you need a way for the kubelet on each node to authenticate to that repository. You can configure *image pull Secrets* to make this possible. These Secrets are configured at the Pod level.

Using imagePullSecrets

The `imagePullSecrets` field is a list of references to Secrets in the same namespace. You can use an `imagePullSecrets` to pass a Secret that contains a Docker (or other) image registry password to the kubelet. The kubelet uses this information to pull a private image on behalf of your Pod. See the PodSpec API for more information about the `imagePullSecrets` field.

Manually specifying an imagePullSecret

You can learn how to specify `imagePullSecrets` from the container images documentation.

Arranging for imagePullSecrets to be automatically attached

You can manually create `imagePullSecrets`, and reference these from a ServiceAccount. Any Pods created with that ServiceAccount or created with that ServiceAccount by default, will get their `imagePullSecrets` field set to that of the service account. See Add ImagePullSecrets to a service account for a detailed explanation of that process.

Using Secrets with static Pods

You cannot use ConfigMaps or Secrets with static Pods.

Immutable Secrets

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

Kubernetes lets you mark specific Secrets (and ConfigMaps) as *immutable*. Preventing changes to the data of an existing Secret has the following benefits:

- protects you from accidental (or unwanted) updates that could cause applications outages
- (for clusters that extensively use Secrets - at least tens of thousands of unique Secret to Pod mounts), switching to immutable Secrets improves the performance of your cluster by significantly reducing load on kube-apiserver. The kubelet does not need to maintain a [watch] on any Secrets that are marked as immutable.

Marking a Secret as immutable

You can create an immutable Secret by setting the `immutable` field to `true`. For example,

```
apiVersion: v1
kind: Secretmetadata: ...data: ...immutable: true
```

You can also update any existing mutable Secret to make it immutable.

Note:

Once a Secret or ConfigMap is marked as immutable, it is *not* possible to revert this change nor to mutate the contents of the `data` field. You can only delete and recreate the Secret. Existing Pods maintain a mount point to the deleted Secret - it is recommended to recreate these pods.

Information security for Secrets

Although ConfigMap and Secret work similarly, Kubernetes applies some additional protection for Secret objects.

Secrets often hold values that span a spectrum of importance, many of which can cause escalations within Kubernetes (e.g. service account tokens) and to external systems. Even if an individual app can reason about the power of the Secrets it expects to interact with, other apps within the same namespace can render those assumptions invalid.

A Secret is only sent to a node if a Pod on that node requires it. For mounting Secrets into Pods, the kubelet stores a copy of the data into a `tmpfs` so that the confidential data is not written to durable storage. Once the Pod that depends on the Secret is deleted, the kubelet deletes its local copy of the confidential data from the Secret.

There may be several containers in a Pod. By default, containers you define only have access to the default ServiceAccount and its related Secret. You must explicitly define environment variables or map a volume into a container in order to provide access to any other Secret.

There may be Secrets for several Pods on the same node. However, only the Secrets that a Pod requests are potentially visible within its containers. Therefore, one Pod does not have access to the Secrets of another Pod.

Configure least-privilege access to Secrets

To enhance the security measures around Secrets, use separate namespaces to isolate access to mounted secrets.

Warning:

Any containers that run with `privileged: true` on a node can access all Secrets used on that node.

What's next

- For guidelines to manage and improve the security of your Secrets, refer to Good practices for Kubernetes Secrets.

- Learn how to manage Secrets using `kubectl`
- Learn how to manage Secrets using config file
- Learn how to manage Secrets using kustomize
- Read the API reference for `Secret`

Resource Management for Pods and Containers

When you specify a Pod, you can optionally specify how much of each resource a container needs. The most common resources to specify are CPU and memory (RAM); there are others.

When you specify the resource *request* for containers in a Pod, the kube-scheduler uses this information to decide which node to place the Pod on. When you specify a resource *limit* for a container, the kubelet enforces those limits so that the running container is not allowed to use more of that resource than the limit you set. The kubelet also reserves at least the *request* amount of that system resource specifically for that container to use.

Requests and limits

If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its `request` for that resource specifies.

For example, if you set a `memory` request of 256 MiB for a container, and that container is in a Pod scheduled to a Node with 8GiB of memory and no other Pods, then the container can try to use more RAM.

Limits are a different story. Both `cpu` and `memory` limits are applied by the kubelet (and container runtime), and are ultimately enforced by the kernel. On Linux nodes, the Linux kernel enforces limits with cgroups. The behavior of `cpu` and `memory` limit enforcement is slightly different.

`cpu` limits are enforced by CPU throttling. When a container approaches its `cpu` limit, the kernel will restrict access to the CPU corresponding to the container's limit. Thus, a `cpu` limit is a hard limit the kernel enforces. Containers may not use more CPU than is specified in their `cpu` limit.

`memory` limits are enforced by the kernel with out of memory (OOM) kills. When a container uses more than its `memory` limit, the kernel may terminate it. However, terminations only happen when the kernel detects memory pressure. Thus, a container that over allocates memory may not be immediately killed. This means `memory` limits are enforced reactively. A container may use more memory than its `memory` limit, but if it does, it may get killed.

Note:

There is an alpha feature `MemoryQoS` which attempts to add more preemptive limit enforcement for memory (as opposed to reactive enforcement by the OOM killer). However, this effort is stalled due to a potential livelock situation a memory hungry can cause.

Note:

If you specify a limit for a resource, but do not specify any request, and no admission-time mechanism has applied a default request for that resource, then Kubernetes copies the limit you specified and uses it as the requested value for the resource.

Resource types

CPU and *memory* are each a *resource type*. A resource type has a base unit. CPU represents compute processing and is specified in units of Kubernetes CPUs. Memory is specified in units of bytes. For Linux workloads, you can specify *huge page* resources. Huge pages are a Linux-specific feature where the node kernel allocates blocks of memory that are much larger than the default page size.

For example, on a system where the default page size is 4KiB, you could specify a limit, `hugepages-2Mi: 80Mi`. If the container tries allocating over 40 2MiB huge pages (a total of 80 MiB), that allocation fails.

Note:

You cannot overcommit `hugepages-*` resources. This is different from the `memory` and `cpu` resources.

CPU and memory are collectively referred to as *compute resources*, or *resources*. Compute resources are measurable quantities that can be requested, allocated, and consumed. They are distinct from API resources. API resources, such as Pods and Services are objects that can be read and modified through the Kubernetes API server.

Resource requests and limits of Pod and container

For each container, you can specify resource limits and requests, including the following:

- `spec.containers[].resources.limits.cpu`
- `spec.containers[].resources.limits.memory`
- `spec.containers[].resources.limits.hugepages-<size>`
- `spec.containers[].resources.requests.cpu`
- `spec.containers[].resources.requests.memory`
- `spec.containers[].resources.requests.hugepages-<size>`

Although you can only specify requests and limits for individual containers, it is also useful to think about the overall resource requests and limits for a Pod. For a particular resource, a *Pod resource request/limit* is the sum of the resource requests/limits of that type for each container in the Pod.

Pod-level resource specification

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

Provided your cluster has the `PodLevelResources` feature gate enabled, you can specify resource requests and limits at the Pod level. At the Pod level, Kubernetes 1.34 only supports resource requests or limits for specific resource types: `cpu` and / or `memory` and / or `hugepages`. With this feature, Kubernetes allows you to declare an overall resource budget for the Pod, which is especially helpful when dealing with a large number of containers where it can be difficult to accurately gauge individual resource needs. Additionally, it enables containers within a Pod to share idle resources with each other, improving resource utilization.

For a Pod, you can specify resource limits and requests for CPU and memory by including the following:

- `spec.resources.limits.cpu`
- `spec.resources.limits.memory`
- `spec.resources.limits.hugepages-<size>`
- `spec.resources.requests.cpu`
- `spec.resources.requests.memory`
- `spec.resources.requests.hugepages-<size>`

Resource units in Kubernetes

CPU resource units

Limits and requests for CPU resources are measured in *cpu* units. In Kubernetes, 1 CPU unit is equivalent to **1 physical CPU core**, or **1 virtual core**, depending on whether the node is a physical host or a virtual machine running inside a physical machine.

Fractional requests are allowed. When you define a container with `spec.containers[].resources.requests.cpu` set to `0.5`, you are requesting half as much CPU time compared to if you asked for `1.0` CPU. For CPU resource units, the quantity expression `0.1` is equivalent to the expression `100m`, which can be read as "one hundred millicpu". Some people say "one hundred millicores", and this is understood to mean the same thing.

CPU resource is always specified as an absolute amount of resource, never as a relative amount. For example, `500m` CPU represents the roughly same amount of computing power whether that container runs on a single-core, dual-core, or 48-core machine.

Note:

Kubernetes doesn't allow you to specify CPU resources with a precision finer than `1m` or `0.001` CPU. To avoid accidentally using an invalid CPU quantity, it's useful to specify CPU units using the milliCPU form instead of the decimal form when using less than 1 CPU unit.

For example, you have a Pod that uses `5m` or `0.005` CPU and would like to decrease its CPU resources. By using the decimal form, it's harder to spot that `0.0005` CPU is an invalid value, while by using the milliCPU form, it's easier to spot that `0.5m` is an invalid value.

Memory resource units

Limits and requests for `memory` are measured in bytes. You can express memory as a plain integer or as a fixed-point number using one of these quantity suffixes: E, P, T, G, M, k. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent roughly the same value:

```
128974848, 129e6, 129M,  128974848000m, 123Mi
```

Pay attention to the case of the suffixes. If you request `400m` of memory, this is a request for 0.4 bytes. Someone who types that probably meant to ask for 400 mebibytes (`400Mi`) or 400 megabytes (`400M`).

Container resources example

The following Pod has two containers. Both containers are defined with a request for 0.25 CPU and 64MiB (2^{26} bytes) of memory. Each container has a limit of 0.5 CPU and 128MiB of memory. You can say the Pod has a request of 0.5 CPU and 128 MiB of memory, and a limit of 1 CPU and 256MiB of memory.

```
---
apiVersion: v1kind: Podmetadata:  name: frontendspec:  containers:  - name: app    image: images.my-company.example/app:v4    reso
```

Pod resources example

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

This feature can be enabled by setting the `PodLevelResources` feature gate. The following Pod has an explicit request of 1 CPU and 100 MiB of memory, and an explicit limit of 1 CPU and 200 MiB of memory. The `pod-resources-demo-ctr-1` container has explicit requests and limits set. However, the `pod-resources-demo-ctr-2` container will simply share the resources available within the Pod resource boundaries, as it does not have explicit requests and limits set.

`pods/resource/pod-level-resources.yaml` Copy pods/resource/pod-level-resources.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: pod-resources-demo  namespace: pod-resources-examplespec:  resources:    limits:      cpu: "1"      memo
```

How Pods with resource requests are scheduled

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum capacity for each of the resource types: the amount of CPU and memory it can provide for Pods. The scheduler ensures that, for each resource type, the sum of the resource requests of the scheduled containers is less than the capacity of the node. Note that although actual memory or CPU resource usage on nodes is very low, the scheduler still refuses to place a Pod on a node if the capacity check fails. This protects against a resource shortage on a node when resource usage later increases, for example, during a daily peak in request rate.

How Kubernetes applies resource requests and limits

When the kubelet starts a container as part of a Pod, the kubelet passes that container's requests and limits for memory and CPU to the container runtime.

On Linux, the container runtime typically configures kernel cgroups that apply and enforce the limits you defined.

- The CPU limit defines a hard ceiling on how much CPU time the container can use. During each scheduling interval (time slice), the Linux kernel checks to see if this limit is exceeded; if so, the kernel waits before allowing that cgroup to resume execution.
- The CPU request typically defines a weighting. If several different containers (cgroups) want to run on a contended system, workloads with larger CPU requests are allocated more CPU time than workloads with small requests.
- The memory request is mainly used during (Kubernetes) Pod scheduling. On a node that uses cgroups v2, the container runtime might use the memory request as a hint to set `memory.min` and `memory.low`.
- The memory limit defines a memory limit for that cgroup. If the container tries to allocate more memory than this limit, the Linux kernel out-of-memory subsystem activates and, typically, intervenes by stopping one of the processes in the container that tried to allocate memory. If that process is the container's PID 1, and the container is marked as restartable, Kubernetes restarts the container.
- The memory limit for the Pod or container can also apply to pages in memory backed volumes, such as an `emptyDir`. The kubelet tracks `tmpfs` emptyDir volumes as container memory use, rather than as local ephemeral storage.   When using memory backed `emptyDir`, be sure to check the notes below.

If a container exceeds its memory request and the node that it runs on becomes short of memory overall, it is likely that the Pod the container belongs to will be evicted.

A container might or might not be allowed to exceed its CPU limit for extended periods of time. However, container runtimes don't terminate Pods or containers for excessive CPU usage.

To determine whether a container cannot be scheduled or is being killed due to resource limits, see the Troubleshooting section.

Monitoring compute & memory resource usage

The kubelet reports the resource usage of a Pod as part of the Pod `status`.

If optional tools for monitoring are available in your cluster, then Pod resource usage can be retrieved either from the Metrics API directly or from your monitoring tools.

Considerations for memory backed `emptyDir` volumes

Caution:

If you do not specify a `sizeLimit` for an `emptyDir` volume, that volume may consume up to that pod's memory limit (`Pod.spec.containers[].resources.limits.memory`). If you do not set a memory limit, the pod has no upper bound on memory consumption, and can consume all available memory on the node. Kubernetes schedules pods based on resource requests (`Pod.spec.containers[].resources.requests`) and will not consider memory usage above the request when deciding if another pod can fit on a given node. This can result in a denial of service and cause the OS to do out-of-memory (OOM) handling. It is possible to create any number of `emptyDir`s that could potentially consume all available memory on the node, making OOM more likely.

From the perspective of memory management, there are some similarities between when a process uses memory as a work area and when using memory-backed `emptyDir`. But when using memory as a volume, like memory-backed `emptyDir`, there are additional points below that you should be careful of:

- Files stored on a memory-backed volume are almost entirely managed by the user application. Unlike when used as a work area for a process, you can not rely on things like language-level garbage collection.
- The purpose of writing files to a volume is to save data or pass it between applications. Neither Kubernetes nor the OS may automatically delete files from a volume, so memory used by those files can not be reclaimed when the system or the pod are under memory pressure.
- A memory-backed `emptyDir` is useful because of its performance, but memory is generally much smaller in size and much higher in cost than other storage media, such as disks or SSDs. Using large amounts of memory for `emptyDir` volumes may affect the normal operation of your pod or of the whole node, so should be used carefully.

If you are administering a cluster or namespace, you can also set ResourceQuota that limits memory use; you may also want to define a LimitRange for additional enforcement. If you specify a `spec.containers[].resources.limits.memory` for each Pod, then the maximum size of an `emptyDir` volume will be the pod's memory limit.

As an alternative, a cluster administrator can enforce size limits for `emptyDir` volumes in new Pods using a policy mechanism such as ValidationAdmissionPolicy.

Local ephemeral storage

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

Nodes have local ephemeral storage, backed by locally-attached writeable devices or, sometimes, by RAM. "Ephemeral" means that there is no long-term guarantee about durability.

Pods use ephemeral local storage for scratch space, caching, and for logs. The kubelet can provide scratch space to Pods using local ephemeral storage to mount `emptyDir` volumes into containers.

The kubelet also uses this kind of storage to hold node-level container logs, container images, and the writable layers of running containers.

Caution:

If a node fails, the data in its ephemeral storage can be lost. Your applications cannot expect any performance SLAs (disk IOPS for example) from local ephemeral storage.

Note:

To make the resource quota work on ephemeral-storage, two things need to be done:

- An admin sets the resource quota for ephemeral-storage in a namespace.
- A user needs to specify limits for the ephemeral-storage resource in the Pod spec.

If the user doesn't specify the ephemeral-storage resource limit in the Pod spec, the resource quota is not enforced on ephemeral-storage.

Kubernetes lets you track, reserve and limit the amount of ephemeral local storage a Pod can consume.

Configurations for local ephemeral storage

Kubernetes supports two ways to configure local ephemeral storage on a node:

- Single filesystem
- Two filesystems

In this configuration, you place all different kinds of ephemeral local data (`emptyDir` volumes, writeable layers, container images, logs) into one filesystem. The most effective way to configure the kubelet means dedicating this filesystem to Kubernetes (kubelet) data.

The kubelet also writes node-level container logs and treats these similarly to ephemeral local storage.

The kubelet writes logs to files inside its configured log directory (`/var/log` by default); and has a base directory for other locally stored data (`/var/lib/kubelet` by default).

Typically, both `/var/lib/kubelet` and `/var/log` are on the system root filesystem, and the kubelet is designed with that layout in mind.

Your node can have as many other filesystems, not used for Kubernetes, as you like.

You have a filesystem on the node that you're using for ephemeral data that comes from running Pods: logs, and `emptyDir` volumes. You can use this filesystem for other data (for example: system logs not related to Kubernetes); it can even be the root filesystem.

The kubelet also writes node-level container logs into the first filesystem, and treats these similarly to ephemeral local storage.

You also use a separate filesystem, backed by a different logical storage device. In this configuration, the directory where you tell the kubelet to place container image layers and writeable layers is on this second filesystem.

The first filesystem does not hold any image layers or writeable layers.

Your node can have as many other filesystems, not used for Kubernetes, as you like.

The kubelet can measure how much local storage it is using. It does this provided that you have set up the node using one of the supported configurations for local ephemeral storage.

If you have a different configuration, then the kubelet does not apply resource limits for ephemeral local storage.

Note:

The kubelet tracks `tmpfs` emptyDir volumes as container memory use, rather than as local ephemeral storage.

Note:

The kubelet will only track the root filesystem for ephemeral storage. OS layouts that mount a separate disk to `/var/lib/kubelet` or `/var/lib/containers` will not report ephemeral storage correctly.

Setting requests and limits for local ephemeral storage

You can specify `ephemeral-storage` for managing local ephemeral storage. Each container of a Pod can specify either or both of the following:

- `spec.containers[].resources.limits.ephemeral-storage`
- `spec.containers[].resources.requests.ephemeral-storage`

Limits and requests for `ephemeral-storage` are measured in byte quantities. You can express storage as a plain integer or as a fixed-point number using one of these suffixes: E, P, T, G, M, k. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following quantities all represent roughly the same value:

- `128974848`
- `129e6`
- `129M`
- `123Mi`

Pay attention to the case of the suffixes. If you request `400m` of ephemeral-storage, this is a request for 0.4 bytes. Someone who types that probably meant to ask for 400 mebibytes (`400Mi`) or 400 megabytes (`400M`).

In the following example, the Pod has two containers. Each container has a request of 2GiB of local ephemeral storage. Each container has a limit of 4GiB of local ephemeral storage. Therefore, the Pod has a request of 4GiB of local ephemeral storage, and a limit of 8GiB of local ephemeral storage. 500Mi of that limit could be consumed by the `emptyDir` volume.

```
apiVersion: v1
kind: Podmetadata:  name: frontendspec:  containers:  - name: app    image: images.my-company.example/app:v4    resources:      re
```

How Pods with ephemeral-storage requests are scheduled

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum amount of local ephemeral storage it can provide for Pods. For more information, see Node Allocatable.

The scheduler ensures that the sum of the resource requests of the scheduled containers is less than the capacity of the node.

Ephemeral storage consumption management

If the kubelet is managing local ephemeral storage as a resource, then the kubelet measures storage use in:

- `emptyDir` volumes, except *tmpfs* `emptyDir` volumes
- directories holding node-level logs
- writeable container layers

If a Pod is using more ephemeral storage than you allow it to, the kubelet sets an eviction signal that triggers Pod eviction.

For container-level isolation, if a container's writable layer and log usage exceeds its storage limit, the kubelet marks the Pod for eviction.

For pod-level isolation the kubelet works out an overall Pod storage limit by summing the limits for the containers in that Pod. In this case, if the sum of the local ephemeral storage usage from all containers and also the Pod's `emptyDir` volumes exceeds the overall Pod storage limit, then the kubelet also marks the Pod for eviction.

Caution:

If the kubelet is not measuring local ephemeral storage, then a Pod that exceeds its local storage limit will not be evicted for breaching local storage resource limits.

However, if the filesystem space for writeable container layers, node-level logs, or `emptyDir` volumes falls low, the node taints itself as short on local storage and this taint triggers eviction for any Pods that don't specifically tolerate the taint.

See the supported configurations for ephemeral local storage.

The kubelet supports different ways to measure Pod storage use:

- Periodic scanning
- Filesystem project quota

The kubelet performs regular, scheduled checks that scan each `emptyDir` volume, container log directory, and writeable container layer.

The scan measures how much space is used.

Note:

In this mode, the kubelet does not track open file descriptors for deleted files.

If you (or a container) create a file inside an `emptyDir` volume, something then opens that file, and you delete the file while it is still open, then the inode for the deleted file stays until you close that file but the kubelet does not categorize the space as in use.

FEATURE STATE: `Kubernetes v1.31 [beta]` (enabled by default: false)

Project quotas are an operating-system level feature for managing storage use on filesystems. With Kubernetes, you can enable project quotas for monitoring storage use. Make sure that the filesystem backing the `emptyDir` volumes, on the node, provides project quota support. For example, XFS and ext4fs offer project quotas.

Note:

Project quotas let you monitor storage use; they do not enforce limits.

Kubernetes uses project IDs starting from `1048576`. The IDs in use are registered in `/etc/projects` and `/etc/projid`. If project IDs in this range are used for other purposes on the system, those project IDs must be registered in `/etc/projects` and `/etc/projid` so that Kubernetes does not use them.

Quotas are faster and more accurate than directory scanning. When a directory is assigned to a project, all files created under a directory are created in that project, and the kernel merely has to keep track of how many blocks are in use by files in that project. If a file is created and deleted, but has an open file descriptor, it continues to consume space. Quota tracking records that space accurately whereas directory scans overlook the storage used by deleted files.

To use quotas to track a pod's resource usage, the pod must be in a user namespace. Within user namespaces, the kernel restricts changes to projectIDs on the filesystem, ensuring the reliability of storage metrics calculated by quotas.

If you want to use project quotas, you should:

- Enable the `LocalStorageCapacityIsolationFSQuotaMonitoring=true` feature gate using the `featureGates` field in the kubelet configuration.

- Ensure the `UserNamespacesSupport` feature gate is enabled, and that the kernel, CRI implementation and OCI runtime support user namespaces.

- Ensure that the root filesystem (or optional runtime filesystem) has project quotas enabled. All XFS filesystems support project quotas. For ext4 filesystems, you need to enable the project quota tracking feature while the filesystem is not mounted.

  ```
  # For ext4, with /dev/block-device not mounted
  sudo tune2fs -O project -Q prjquota /dev/block-device
  ```

- Ensure that the root filesystem (or optional runtime filesystem) is mounted with project quotas enabled. For both XFS and ext4fs, the mount option is named `prjquota`.

If you don't want to use project quotas, you should:

- Disable the `LocalStorageCapacityIsolationFSQuotaMonitoring` feature gate using the `featureGates` field in the kubelet configuration.

Extended resources

Extended resources are fully-qualified resource names outside the `kubernetes.io` domain. They allow cluster operators to advertise and users to consume the non-Kubernetes-built-in resources.

There are two steps required to use Extended Resources. First, the cluster operator must advertise an Extended Resource. Second, users must request the Extended Resource in Pods.

Managing extended resources

Node-level extended resources

Node-level extended resources are tied to nodes.

Device plugin managed resources

See Device Plugin for how to advertise device plugin managed resources on each node.

Other resources

To advertise a new node-level extended resource, the cluster operator can submit a `PATCH` HTTP request to the API server to specify the available quantity in the `status.capacity` for a node in the cluster. After this operation, the node's `status.capacity` will include a new resource. The `status.allocatable` field is updated automatically with the new resource asynchronously by the kubelet.

Because the scheduler uses the node's `status.allocatable` value when evaluating Pod fitness, the scheduler only takes account of the new value after that asynchronous update. There may be a short delay between patching the node capacity with a new resource and the time when the first Pod that requests the resource can be scheduled on that node.

Example:

Here is an example showing how to use `curl` to form an HTTP request that advertises five "example.com/foo" resources on node `k8s-node-1` whose master is `k8s-master`.

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \--data '[{"op": "add", "path": "/status/capacity/example.com~1foo", "value": "5"}]' \http://k8s-master:8080/api/v
```

Note:

In the preceding request, `~1` is the encoding for the character `/` in the patch path. The operation path value in JSON-Patch is interpreted as a JSON-Pointer. For more details, see IETF RFC 6901, section 3.

Cluster-level extended resources

Cluster-level extended resources are not tied to nodes. They are usually managed by scheduler extenders, which handle the resource consumption and resource quota.

You can specify the extended resources that are handled by scheduler extenders in scheduler configuration

Example:

The following configuration for a scheduler policy indicates that the cluster-level extended resource "example.com/foo" is handled by the scheduler extender.

- The scheduler sends a Pod to the scheduler extender only if the Pod requests "example.com/foo".
- The `ignoredByScheduler` field specifies that the scheduler does not check the "example.com/foo" resource in its `PodFitsResources` predicate.

```
{
  "kind": "Policy",
  "apiVersion": "v1",
  "extenders": [
    {
      "urlPrefix":"<extender-endpoint>",
      "bindVerb": "bind",
      "managedResources": [
        {
          "name": "example.com/foo",
          "ignoredByScheduler": true
        }
      ]
    }
  ]
}
```

Extended resources allocation by DRA

Extended resources allocation by DRA allows cluster administrators to specify an `extendedResourceName` in DeviceClass, then the devices matching the DeviceClass can be requested from a pod's extended resource requests. Read more about Extended Resource allocation by DRA.

Consuming extended resources

Users can consume extended resources in Pod specs like CPU and memory. The scheduler takes care of the resource accounting so that no more than the available amount is simultaneously allocated to Pods.

The API server restricts quantities of extended resources to whole numbers. Examples of *valid* quantities are `3`, `3000m` and `3Ki`. Examples of *invalid* quantities are `0.5` and `1500m` (because `1500m` would result in `1.5`).

Note:

Extended resources replace Opaque Integer Resources. Users can use any domain name prefix other than `kubernetes.io` which is reserved.

To consume an extended resource in a Pod, include the resource name as a key in the `spec.containers[].resources.limits` map in the container spec.

Note:

Extended resources cannot be overcommitted, so request and limit must be equal if both are present in a container spec.

A Pod is scheduled only if all of the resource requests are satisfied, including CPU, memory and any extended resources. The Pod remains in the `PENDING` state as long as the resource request cannot be satisfied.

Example:

The Pod below requests 2 CPUs and 1 "example.com/foo" (an extended resource).

```
apiVersion: v1
kind: Podmetadata:  name: my-podspec:  containers:  - name: my-container    image: myimage    resources:      requests:        cpu
```

PID limiting

Process ID (PID) limits allow for the configuration of a kubelet to limit the number of PIDs that a given Pod can consume. See PID Limiting for information.

Troubleshooting

My Pods are pending with event message `FailedScheduling`

If the scheduler cannot find any node where a Pod can fit, the Pod remains unscheduled until a place can be found. An Event is produced each time the scheduler fails to find a place for the Pod. You can use `kubectl` to view the events for a Pod; for example:

```
kubectl describe pod frontend | grep -A 9999999999 Events
```

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  23s   default-scheduler  0/42 nodes available: insufficient cpu
```

In the preceding example, the Pod named "frontend" fails to be scheduled due to insufficient CPU resource on any node. Similar error messages can also suggest failure due to insufficient memory (PodExceedsFreeMemory). In general, if a Pod is pending with a message of this type, there are several things to try:

- Add more nodes to the cluster.
- Terminate unneeded Pods to make room for pending Pods.
- Check that the Pod is not larger than all the nodes. For example, if all the nodes have a capacity of `cpu: 1`, then a Pod with a request of `cpu: 1.1` will never be scheduled.
- Check for node taints. If most of your nodes are tainted, and the new Pod does not tolerate that taint, the scheduler only considers placements onto the remaining nodes that don't have that taint.

You can check node capacities and amounts allocated with the `kubectl describe nodes` command. For example:

```
kubectl describe nodes e2e-test-node-pool-4lw4
```

```
Name:            e2e-test-node-pool-4lw4
[ ... lines removed for clarity ...]
Capacity:
 cpu:                               2
 memory:                            7679792Ki
 pods:                              110
Allocatable:
 cpu:                               1800m
 memory:                            7474992Ki
 pods:                              110
[ ... lines removed for clarity ...]
Non-terminated Pods:        (5 in total)
  Namespace    Name                                  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  ---------    ----                                  ------------  ----------  ---------------  -------------
  kube-system  fluentd-gcp-v1.38-28bv1               100m (5%)     0 (0%)      200Mi (2%)       200Mi (2%)
  kube-system  kube-dns-3297075139-61lj3             260m (13%)    0 (0%)      100Mi (1%)       170Mi (2%)
  kube-system  kube-proxy-e2e-test-...               100m (5%)     0 (0%)      0 (0%)           0 (0%)
  kube-system  monitoring-influxdb-grafana-v4-z1m12  200m (10%)    200m (10%)  600Mi (8%)       600Mi (8%)
  kube-system  node-problem-detector-v0.1-fj7m3      20m (1%)      200m (10%)  20Mi (0%)        100Mi (1%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests    CPU Limits    Memory Requests    Memory Limits
  ------------    ----------    ---------------    -------------
  680m (34%)      400m (20%)    920Mi (11%)        1070Mi (13%)
```

In the preceding output, you can see that if a Pod requests more than 1.120 CPUs or more than 6.23Gi of memory, that Pod will not fit on the node.

By looking at the “Pods” section, you can see which Pods are taking up space on the node.

The amount of resources available to Pods is less than the node capacity because system daemons use a portion of the available resources. Within the Kubernetes API, each Node has a `.status.allocatable` field (see NodeStatus for details).

The `.status.allocatable` field describes the amount of resources that are available to Pods on that node (for example: 15 virtual CPUs and 7538 MiB of memory). For more information on node allocatable resources in Kubernetes, see Reserve Compute Resources for System Daemons.

You can configure resource quotas to limit the total amount of resources that a namespace can consume. Kubernetes enforces quotas for objects in particular namespace when there is a ResourceQuota in that namespace. For example, if you assign specific namespaces to different teams, you can add ResourceQuotas into those namespaces. Setting resource quotas helps to prevent one team from using so much of any resource that this over-use affects other teams.

You should also consider what access you grant to that namespace: **full** write access to a namespace allows someone with that access to remove any resource, including a configured ResourceQuota.

My container is terminated

Your container might get terminated because it is resource-starved. To check whether a container is being killed because it is hitting a resource limit, call `kubectl describe pod` on the Pod of interest:

```
kubectl describe pod simmemleak-hra99
```

The output is similar to:

```
Name:                           simmemleak-hra99
Namespace:                      default
Image(s):                       saadali/simmemleak
Node:                           kubernetes-node-tf0f/10.240.216.66
Labels:                         name=simmemleak
Status:                         Running
Reason:
Message:
IP:                             10.244.2.75
Containers:
  simmemleak:
    Image:  saadali/simmemleak:latest
    Limits:
      cpu:          100m
      memory:       50Mi
    State:          Running
      Started:      Tue, 07 Jul 2019 12:54:41 -0700
    Last State:     Terminated
      Reason:       OOMKilled
      Exit Code:    137
      Started:      Fri, 07 Jul 2019 12:54:30 -0700
      Finished:     Fri, 07 Jul 2019 12:54:33 -0700
    Ready:          False
    Restart Count:  5
Conditions:
  Type      Status
  Ready     False
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  42s   default-scheduler  Successfully assigned simmemleak-hra99 to kubernetes-node-tf0f
  Normal  Pulled     41s   kubelet            Container image "saadali/simmemleak:latest" already present on machine
  Normal  Created    41s   kubelet            Created container simmemleak
  Normal  Started    40s   kubelet            Started container simmemleak
  Normal  Killing    32s   kubelet            Killing container with id ead3fb35-5cf5-44ed-9ae1-488115be66c6: Need to kill Pod
```

In the preceding example, the `Restart Count: 5` indicates that the `simmemleak` container in the Pod was terminated and restarted five times (so far). The `OOMKilled` reason shows that the container tried to use more memory than its limit.

Your next step might be to check the application code for a memory leak. If you find that the application is behaving how you expect, consider setting a higher memory limit (and possibly request) for that container.

What's next

- Get hands-on experience assigning Memory resources to containers and Pods.
- Get hands-on experience assigning CPU resources to containers and Pods.
- Read how the API reference defines a container and its resource requirements
- Read about project quotas in XFS
- Read more about the kube-scheduler configuration reference (v1)
- Read more about Quality of Service classes for Pods
- Read more about Extended Resource allocation by DRA

Cluster Networking

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems to address:

1. Highly-coupled container-to-container communications: this is solved by Pods and `localhost` communications.
2. Pod-to-Pod communications: this is the primary focus of this document.
3. Pod-to-Service communications: this is covered by Services.
4. External-to-Service communications: this is also covered by Services.

Kubernetes is all about sharing machines among applications. Typically, sharing machines requires ensuring that two applications do not try to use the same ports. Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control.

Dynamic port allocation brings a lot of complications to the system - every application has to take ports as flags, the API servers have to know how to insert dynamic port numbers into configuration blocks, services have to know how to find each other, etc. Rather than deal with this, Kubernetes takes a different approach.

To learn about the Kubernetes networking model, see here.

Kubernetes IP address ranges

Kubernetes clusters require to allocate non-overlapping IP addresses for Pods, Services and Nodes, from a range of available addresses configured in the following components:

- The network plugin is configured to assign IP addresses to Pods.
- The kube-apiserver is configured to assign IP addresses to Services.
- The kubelet or the cloud-controller-manager is configured to assign IP addresses to Nodes.

A figure illustrating the different network ranges in a kubernetes cluster

Cluster networking types

Kubernetes clusters, attending to the IP families configured, can be categorized into:

- IPv4 only: The network plugin, kube-apiserver and kubelet/cloud-controller-manager are configured to assign only IPv4 addresses.
- IPv6 only: The network plugin, kube-apiserver and kubelet/cloud-controller-manager are configured to assign only IPv6 addresses.
- IPv4/IPv6 or IPv6/IPv4 dual-stack:
    - The network plugin is configured to assign IPv4 and IPv6 addresses.
    - The kube-apiserver is configured to assign IPv4 and IPv6 addresses.
    - The kubelet or cloud-controller-manager is configured to assign IPv4 and IPv6 address.
    - All components must agree on the configured primary IP family.

Kubernetes clusters only consider the IP families present on the Pods, Services and Nodes objects, independently of the existing IPs of the represented objects. Per example, a server or a pod can have multiple IP addresses on its interfaces, but only the IP addresses in `node.status.addresses` or `pod.status.ips` are considered for implementing the Kubernetes network model and defining the type of the cluster.

How to implement the Kubernetes network model

The network model is implemented by the container runtime on each node. The most common container runtimes use Container Network Interface (CNI) plugins to manage their network and security capabilities. Many different CNI plugins exist from many different vendors. Some of these provide only basic features of adding and removing network interfaces, while others provide more sophisticated solutions, such as integration with other container orchestration systems, running multiple CNI plugins, advanced IPAM features etc.

See this page for a non-exhaustive list of networking addons supported by Kubernetes.

What's next

The early design of the networking model and its rationale are described in more detail in the networking design document. For future plans and some on-going efforts that aim to improve Kubernetes networking, please refer to the SIG-Network KEPs.

ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

Caution:

ConfigMap does not provide secrecy or encryption. If the data you want to store are confidential, use a Secret rather than a ConfigMap, or use additional (third party) tools to keep your data private.

Motivation

Use a ConfigMap for setting configuration data separately from application code.

For example, imagine that you are developing an application that you can run on your own computer (for development) and in the cloud (to handle real traffic). You write the code to look in an environment variable named `DATABASE_HOST`. Locally, you set that variable to `localhost`. In the cloud, you set it to refer to a Kubernetes Service that exposes the database component to your cluster. This lets you fetch a container image running in the cloud and debug the exact same code locally if needed.

Note:

A ConfigMap is not designed to hold large chunks of data. The data stored in a ConfigMap cannot exceed 1 MiB. If you need to store settings that are larger than this limit, you may want to consider mounting a volume or use a separate database or file service.

ConfigMap object

A ConfigMap is an API object that lets you store configuration for other objects to use. Unlike most Kubernetes objects that have a `spec`, a ConfigMap has `data` and `binaryData` fields. These fields accept key-value pairs as their values. Both the `data` field and the `binaryData` are optional. The `data` field is designed to contain UTF-8 strings while the `binaryData` field is designed to contain binary data as base64-encoded strings.

The name of a ConfigMap must be a valid DNS subdomain name.

Each key under the `data` or the `binaryData` field must consist of alphanumeric characters, `-`, `_` or `.`. The keys stored in `data` must not overlap with the keys in the `binaryData` field.

Starting from v1.19, you can add an `immutable` field to a ConfigMap definition to create an immutable ConfigMap.

ConfigMaps and Pods

You can write a Pod `spec` that refers to a ConfigMap and configures the container(s) in that Pod based on the data in the ConfigMap. The Pod and the ConfigMap must be in the same namespace.

Note:

The `spec` of a static Pod cannot refer to a ConfigMap or any other API objects.

Here's an example ConfigMap that has some keys with single values, and other keys where the value looks like a fragment of a configuration format.

```
apiVersion: v1
kind: ConfigMapmetadata:  name: game-demodata:  # property-like keys; each key maps to a simple value  player_initial_lives: "3"
```

There are four different ways that you can use a ConfigMap to configure a container inside a Pod:

1. Inside a container command and args
2. Environment variables for a container
3. Add a file in read-only volume, for the application to read
4. Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

These different methods lend themselves to different ways of modeling the data being consumed. For the first three methods, the kubelet uses the data from the ConfigMap when it launches container(s) for a Pod.

The fourth method means you have to write code to read the ConfigMap and its data. However, because you're using the Kubernetes API directly, your application can subscribe to get updates whenever the ConfigMap changes, and react when that happens. By accessing the Kubernetes API directly, this technique also lets you access a ConfigMap in a different namespace.

Here's an example Pod that uses values from `game-demo` to configure a Pod:

`configmap/configure-pod.yaml` Copy configmap/configure-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: configmap-demo-podspec:  containers:    - name: demo      image: alpine      command: ["sleep", "3600"]
```

A ConfigMap doesn't differentiate between single line property values and multi-line file-like values. What matters is how Pods and other objects consume those values.

For this example, defining a volume and mounting it inside the `demo` container as `/config` creates two files, `/config/game.properties` and `/config/user-interface.properties`, even though there are four keys in the ConfigMap. This is because the Pod definition specifies an `items` array in the `volumes` section. If you omit the `items` array entirely, every key in the ConfigMap becomes a file with the same name as the key, and you get 4 files.

Using ConfigMaps

ConfigMaps can be mounted as data volumes. ConfigMaps can also be used by other parts of the system, without being directly exposed to the Pod. For example, ConfigMaps can hold data that other parts of the system should use for configuration.

The most common way to use ConfigMaps is to configure settings for containers running in a Pod in the same namespace. You can also use a ConfigMap separately.

For example, you might encounter addons or operators that adjust their behavior based on a ConfigMap.

Using ConfigMaps as files from a Pod

To consume a ConfigMap in a volume in a Pod:

1. Create a ConfigMap or use an existing one. Multiple Pods can reference the same ConfigMap.
2. Modify your Pod definition to add a volume under `.spec.volumes[]`. Name the volume anything, and have a `.spec.volumes[].configMap.name` field set to reference your ConfigMap object.

3. Add a `.spec.containers[].volumeMounts[]` to each container that needs the ConfigMap. Specify `.spec.containers[].volumeMounts[].readOnly = true` and `.spec.containers[].volumeMounts[].mountPath` to an unused directory name where you would like the ConfigMap to appear.
4. Modify your image or command line so that the program looks for files in that directory. Each key in the ConfigMap `data` map becomes the filename under `mountPath`.

This is an example of a Pod that mounts a ConfigMap in a volume:

```
apiVersion: v1
kind: Podmetadata:  name: mypodspec:  containers:  - name: mypod    image: redis    volumeMounts:    - name: foo      mountPath: "
```

Each ConfigMap you want to use needs to be referred to in `.spec.volumes`.

If there are multiple containers in the Pod, then each container needs its own `volumeMounts` block, but only one `.spec.volumes` is needed per ConfigMap.

Mounted ConfigMaps are updated automatically

When a ConfigMap currently consumed in a volume is updated, projected keys are eventually updated as well. The kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the ConfigMap. The type of the cache is configurable using the `configMapAndSecretChangeDetectionStrategy` field in the KubeletConfiguration struct. A ConfigMap can be either propagated by watch (default), ttl-based, or by redirecting all requests directly to the API server. As a result, the total delay from the moment when the ConfigMap is updated to the moment when new keys are projected to the Pod can be as long as the kubelet sync period + cache propagation delay, where the cache propagation delay depends on the chosen cache type (it equals to watch propagation delay, ttl of cache, or zero correspondingly).

ConfigMaps consumed as environment variables are not updated automatically and require a pod restart.

Note:

A container using a ConfigMap as a subPath volume mount will not receive ConfigMap updates.

Using Configmaps as environment variables

To use a Configmap in an environment variable in a Pod:

1. For each container in your Pod specification, add an environment variable for each Configmap key that you want to use to the `env[].valueFrom.configMapKeyRef` field.
2. Modify your image and/or command line so that the program looks for values in the specified environment variables.

This is an example of defining a ConfigMap as a pod environment variable:

The following ConfigMap (myconfigmap.yaml) stores two properties: username and access_level:

```
apiVersion: v1
kind: ConfigMapmetadata:  name: myconfigmapdata:  username: k8s-admin  access_level: "1"
```

The following command will create the ConfigMap object:

```
kubectl apply -f myconfigmap.yaml
```

The following Pod consumes the content of the ConfigMap as environment variables:

configmap/env-configmap.yaml Copy configmap/env-configmap.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: env-configmapspec:  containers:    - name: app      command: ["/bin/sh", "-c", "printenv"]      image: b
```

The `envFrom` field instructs Kubernetes to create environment variables from the sources nested within it. The inner `configMapRef` refers to a ConfigMap by its name and selects all its key-value pairs. Add the Pod to your cluster, then retrieve its logs to see the output from the printenv command. This should confirm that the two key-value pairs from the ConfigMap have been set as environment variables:

```
kubectl apply -f env-configmap.yaml
```

```
kubectl logs pod/ env-configmap
```

The output is similar to this:

```
...
username: "k8s-admin"
access_level: "1"
...
```

Sometimes a Pod won't require access to all the values in a ConfigMap. For example, you could have another Pod which only uses the username value from the ConfigMap. For this use case, you can use the `env.valueFrom` syntax instead, which lets you select individual keys in a ConfigMap. The name of the environment variable can also be different from the key within the ConfigMap. For example:

```
apiVersion: v1
kind: Podmetadata:  name: env-configmapspec:  containers:  - name: envars-test-container    image: nginx    env:    - name: CONFIG
```

In the Pod created from this manifest, you will see that the environment variable `CONFIGMAP_USERNAME` is set to the value of the `username` value from the ConfigMap. Other keys from the ConfigMap data are not copied into the environment.

It's important to note that the range of characters allowed for environment variable names in pods is restricted. If any keys do not meet the rules, those keys are not made available to your container, though the Pod is allowed to start.

Immutable ConfigMaps

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

The Kubernetes feature *Immutable Secrets and ConfigMaps* provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use ConfigMaps (at least tens of thousands of unique ConfigMap to Pod mounts), preventing changes to their data has the following advantages:

- protects you from accidental (or unwanted) updates that could cause applications outages
- improves performance of your cluster by significantly reducing load on kube-apiserver, by closing watches for ConfigMaps marked as immutable.

You can create an immutable ConfigMap by setting the `immutable` field to `true`. For example:

```
apiVersion: v1
kind: ConfigMapmetadata:  ...data:  ...immutable: true
```

Once a ConfigMap is marked as immutable, it is *not* possible to revert this change nor to mutate the contents of the `data` or the `binaryData` field. You can only delete and recreate the ConfigMap. Because existing Pods maintain a mount point to the deleted ConfigMap, it is recommended to recreate these pods.

What's next

- Read about Secrets.
- Read Configure a Pod to Use a ConfigMap.
- Read about changing a ConfigMap (or any other Kubernetes object)
- Read The Twelve-Factor App to understand the motivation for separating code from configuration.

Container Lifecycle Hooks

This page describes how kubelet managed Containers can use the Container lifecycle hook framework to run code triggered by events during their management lifecycle.

Overview

Analogous to many programming language frameworks that have component lifecycle hooks, such as Angular, Kubernetes provides Containers with lifecycle hooks. The hooks enable Containers to be aware of events in their management lifecycle and run code implemented in a handler when the corresponding lifecycle hook is executed.

Container hooks

There are two hooks that are exposed to Containers:

`PostStart`

This hook is executed immediately after a container is created. However, there is no guarantee that the hook will execute before the container ENTRYPOINT. No parameters are passed to the handler.

`PreStop`

This hook is called immediately before a container is terminated due to an API request or management event such as a liveness/startup probe failure, preemption, resource contention and others. A call to the `PreStop` hook fails if the container is already in a terminated or completed state and the hook must complete before the TERM signal to stop the container can be sent. The Pod's termination grace period countdown begins before the `PreStop` hook is executed, so regardless of the outcome of the handler, the container will eventually terminate within the Pod's termination grace period. No parameters are passed to the handler.

A more detailed description of the termination behavior can be found in Termination of Pods.

`StopSignal`

The StopSignal lifecycle can be used to define a stop signal which would be sent to the container when it is stopped. If you set this, it overrides any `STOPSIGNAL` instruction defined within the container image.

A more detailed description of termination behaviour with custom stop signals can be found in Stop Signals.

Hook handler implementations

Containers can access a hook by implementing and registering a handler for that hook. There are three types of hook handlers that can be implemented for Containers:

- Exec - Executes a specific command, such as `pre-stop.sh`, inside the cgroups and namespaces of the Container. Resources consumed by the command are counted against the Container.
- HTTP - Executes an HTTP request against a specific endpoint on the Container.
- Sleep - Pauses the container for a specified duration.

Hook handler execution

When a Container lifecycle management hook is called, the Kubernetes management system executes the handler according to the hook action, `httpGet`, `tcpSocket` (deprecated) and `sleep` are executed by the kubelet process, and `exec` is executed in the container.

The `PostStart` hook handler call is initiated when a container is created, meaning the container ENTRYPOINT and the `PostStart` hook are triggered simultaneously. However, if the `PostStart` hook takes too long to execute or if it hangs, it can prevent the container from transitioning to a `running` state.

`PreStop` hooks are not executed asynchronously from the signal to stop the Container; the hook must complete its execution before the TERM signal can be sent. If a `PreStop` hook hangs during execution, the Pod's phase will be `Terminating` and remain there until the Pod is killed after its `terminationGracePeriodSeconds` expires. This grace period applies to the total time it takes for both the `PreStop` hook to execute and for the Container to stop normally. If, for example, `terminationGracePeriodSeconds` is 60, and the hook takes 55 seconds to complete, and the Container takes 10 seconds to stop normally after receiving the signal, then the Container will be killed before it can stop normally, since `terminationGracePeriodSeconds` is less than the total time (55+10) it takes for these two things to happen.

If either a `PostStart` or `PreStop` hook fails, it kills the Container.

Users should make their hook handlers as lightweight as possible. There are cases, however, when long running commands make sense, such as when saving state prior to stopping a Container.

Hook delivery guarantees

Hook delivery is intended to be *at least once*, which means that a hook may be called multiple times for any given event, such as for `PostStart` or `PreStop`. It is up to the hook implementation to handle this correctly.

Generally, only single deliveries are made. If, for example, an HTTP hook receiver is down and is unable to take traffic, there is no attempt to resend. In some rare cases, however, double delivery may occur. For instance, if a kubelet restarts in the middle of sending a hook, the hook might be resent after the kubelet comes back up.

Debugging Hook handlers

The logs for a Hook handler are not exposed in Pod events. If a handler fails for some reason, it broadcasts an event. For `PostStart`, this is the `FailedPostStartHook` event, and for `PreStop`, this is the `FailedPreStopHook` event. To generate a failed `FailedPostStartHook` event yourself, modify the lifecycle-events.yaml file to change the postStart command to "badcommand" and apply it. Here is some example output of the resulting events you see from running `kubectl describe pod lifecycle-demo`:

```
Events:
  Type     Reason               Age                From               Message
  ----     ------               ----               ----               -------
  Normal   Scheduled            7s                 default-scheduler  Successfully assigned default/lifecycle-demo to ip-XXX-XXX-XX-X
  Normal   Pulled               6s                 kubelet            Successfully pulled image "nginx" in 229.604315ms
  Normal   Pulling              4s (x2 over 6s)    kubelet            Pulling image "nginx"
  Normal   Created              4s (x2 over 5s)    kubelet            Created container lifecycle-demo-container
  Normal   Started              4s (x2 over 5s)    kubelet            Started container lifecycle-demo-container
  Warning  FailedPostStartHook  4s (x2 over 5s)    kubelet            Exec lifecycle hook ([badcommand]) for Container "lifecycle-dem
  Normal   Killing              4s (x2 over 5s)    kubelet            FailedPostStartHook
  Normal   Pulled               4s                 kubelet            Successfully pulled image "nginx" in 215.66395ms
  Warning  BackOff              2s (x2 over 3s)    kubelet            Back-off restarting failed container
```

What's next

- Learn more about the Container environment.
- Get hands-on experience attaching handlers to Container lifecycle events.

Containers

Technology for packaging an application along with its runtime dependencies.

This page will discuss containers and container images, as well as their use in operations and solution development.

The word *container* is an overloaded term. Whenever you use the word, check whether your audience uses the same definition.

Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it.

Containers decouple applications from the underlying host infrastructure. This makes deployment easier in different cloud or OS environments.

Each node in a Kubernetes cluster runs the containers that form the Pods assigned to that node. Containers in a Pod are co-located and co-scheduled to run on the same node.

Container images

A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings.

Containers are intended to be stateless and immutable: you should not change the code of a container that is already running. If you have a containerized application and want to make changes, the correct process is to build a new image that includes the change, then recreate the container to start from the updated image.

Container runtimes

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

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

Usually, you can allow your cluster to pick the default container runtime for a Pod. If you need to use more than one container runtime in your cluster, you can specify the RuntimeClass for a Pod to make sure that Kubernetes runs those containers using a particular container runtime.

You can also use RuntimeClass to run different Pods with the same container runtime but with different settings.

Container Environment

Container Lifecycle Hooks

Container Runtime Interface (CRI)

Metrics for Kubernetes Object States

kube-state-metrics, an add-on agent to generate and expose cluster-level metrics.

The state of Kubernetes objects in the Kubernetes API can be exposed as metrics. An add-on agent called kube-state-metrics can connect to the Kubernetes API server and expose a HTTP endpoint with metrics generated from the state of individual objects in the cluster. It exposes various information about the state of objects like labels and annotations, startup and termination times, status or the phase the object currently is in. For example, containers running in pods create a `kube_pod_container_info` metric. This includes the name of the container, the name of the pod it is part of, the namespace the pod is running in, the name of the container image, the ID of the image, the image name from the spec of the container, the ID of the running container and the ID of the pod as labels.

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

An external component that is able and capable to scrape the endpoint of kube-state-metrics (for example via Prometheus) can now be used to enable the following use cases.

Example: using metrics from kube-state-metrics to query the cluster state

Metric series generated by kube-state-metrics are helpful to gather further insights into the cluster, as they can be used for querying.

If you use Prometheus or another tool that uses the same query language, the following PromQL query returns the number of pods that are not ready:

```
count(kube_pod_status_ready{condition="false"}) by (namespace, pod)
```

Example: alerting based on from kube-state-metrics

Metrics generated from kube-state-metrics also allow for alerting on issues in the cluster.

If you use Prometheus or a similar tool that uses the same alert rule language, the following alert will fire if there are pods that have been in a `Terminating` state for more than 5 minutes:

```
groups:
- name: Pod state   rules:   - alert: PodsBlockedInTerminatingState    expr: count(kube_pod_deletion_timestamp) by (namespace, pod)
```

Node Shutdowns

In a Kubernetes cluster, a node can be shut down in a planned graceful way or unexpectedly because of reasons such as a power outage or something else external. A node shutdown could lead to workload failure if the node is not drained before the shutdown. A node shutdown can be either **graceful** or **non-graceful**.

Graceful node shutdown

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

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

Enabling graceful node shutdown

- Linux
- Windows

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

On Linux, the graceful node shutdown feature is controlled with the `GracefulNodeShutdown` feature gate which is enabled by default in 1.21.

Note:

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

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

On Windows, the graceful node shutdown feature is controlled with the `WindowsGracefulNodeShutdown` feature gate which is introduced in 1.32 as an alpha feature. In Kubernetes 1.34 the feature is Beta and is enabled by default.

Note:

The Windows graceful node shutdown feature depends on kubelet running as a Windows service, it will then have a registered service control handler to delay the preshutdown event with a given duration.

Windows graceful node shutdown can not be cancelled.

If kubelet is not running as a Windows service, it will not be able to set and monitor the Preshutdown event, the node will have to go through the Non-Graceful Node Shutdown procedure mentioned above.

In the case where the Windows graceful node shutdown feature is enabled, but the kubelet is not running as a Windows service, the kubelet will continue running instead of failing. However, it will log an error indicating that it needs to be run as a Windows service.

Configuring graceful node shutdown

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

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

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

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

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

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

The graceful node shutdown feature is configured with two `KubeletConfiguration` options:

- `shutdownGracePeriod`:

  Specifies the total duration that the node should delay the shutdown by. This is the total grace period for pod termination for both regular and critical pods.

- `shutdownGracePeriodCriticalPods`:

  Specifies the duration used to terminate critical pods during a node shutdown. This value should be less than `shutdownGracePeriod`.

Note:

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

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

Note:

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

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

Pod Priority based graceful node shutdown

FEATURE STATE: `Kubernetes v1.24 [beta]` (enabled by default: true)

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

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

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

Assuming the following custom pod priority classes in a cluster,

Pod priority class name	Pod priority class value
`custom-class-a`	100000

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

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

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

The corresponding kubelet config YAML configuration would be:

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

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

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

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

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

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

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

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

Note:

The ability to take Pod priority into account during graceful node shutdown was introduced as an Alpha feature in Kubernetes v1.23. In Kubernetes 1.34 the feature is Beta and is enabled by default.

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

Non-graceful node shutdown handling

FEATURE STATE: `Kubernetes v1.28 [stable]` (enabled by default: true)

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

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

To mitigate the above situation, a user can manually add the taint `node.kubernetes.io/out-of-service` with either `NoExecute` or `NoSchedule` effect to a Node marking it out-of-service. If a Node is marked out-of-service with this taint, the pods on the node will be forcefully deleted if there are no matching tolerations on it and volume detach operations for the pods terminating on the node will happen immediately. This allows the Pods on the out-of-service node to recover quickly on a different node.

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

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

Note:

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

Forced storage detach on timeout

In any situation where a pod deletion has not succeeded for 6 minutes, kubernetes will force detach volumes being unmounted if the node is unhealthy at that instant. Any workload still running on the node that uses a force-detached volume will cause a violation of the CSI specification, which states that `ControllerUnpublishVolume` "**must** be called after all `NodeUnstageVolume` and `NodeUnpublishVolume` on the volume are called and succeed". In such circumstances, volumes on the node in question might encounter data corruption.

The forced storage detach behaviour is optional; users might opt to use the "Non-graceful node shutdown" feature instead.

Force storage detach on timeout can be disabled by setting the `disable-force-detach-on-timeout` config field in `kube-controller-manager`. Disabling the force detach on timeout feature means that a volume that is hosted on a node that is unhealthy for more than 6 minutes will not have its associated VolumeAttachment deleted.

After this setting has been applied, unhealthy pods still attached to volumes must be recovered via the Non-Graceful Node Shutdown procedure mentioned above.

Note:

- Caution must be taken while using the Non-Graceful Node Shutdown procedure.
- Deviation from the steps documented above can result in data corruption.

What's next

Learn more about the following:

- Blog: Non-Graceful Node Shutdown.
- Cluster Architecture: Nodes.

Liveness, Readiness, and Startup Probes

Kubernetes has various types of probes:

- Liveness probe
- Readiness probe
- Startup probe

Liveness probe

Liveness probes determine when to restart a container. For example, liveness probes could catch a deadlock when an application is running but unable to make progress.

If a container fails its liveness probe repeatedly, the kubelet restarts the container.

Liveness probes do not wait for readiness probes to succeed. If you want to wait before executing a liveness probe, you can either define `initialDelaySeconds` or use a startup probe.

Readiness probe

Readiness probes determine when a container is ready to accept traffic. This is useful when waiting for an application to perform time-consuming initial tasks that depend on its backing services; for example: establishing network connections, loading files, and warming caches. Readiness probes can also be useful later in the container’s lifecycle, for example, when recovering from temporary faults or overloads.

If the readiness probe returns a failed state, Kubernetes removes the pod from all matching service endpoints.

Readiness probes run on the container during its whole lifecycle.

Startup probe

A startup probe verifies whether the application within a container is started. This can be used to adopt liveness checks on slow starting containers, avoiding them getting killed by the kubelet before they are up and running.

If such a probe is configured, it disables liveness and readiness checks until it succeeds.

This type of probe is only executed at startup, unlike liveness and readiness probes, which are run periodically.

- Read more about the Configure Liveness, Readiness and Startup Probes.

Configuration

Resources that Kubernetes provides for configuring Pods.

Container Runtime Interface (CRI)

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

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

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

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

The API

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

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

For Kubernetes v1.26 and later, the kubelet requires that the container runtime supports the `v1` CRI API. If a container runtime does not support the `v1` API, the kubelet will not register the node.

Upgrading

When upgrading the Kubernetes version on a node, the kubelet restarts. If the container runtime does not support the `v1` CRI API, the kubelet will fail to register and report an error. If a gRPC re-dial is required because the container runtime has been upgraded, the runtime must support the `v1` CRI API for the connection to succeed. This might require a restart of the kubelet after the container runtime is correctly configured.

What's next

- Learn more about the CRI protocol definition

Node Autoscaling

Automatically provision and consolidate the Nodes in your cluster to adapt to demand and optimize cost.

In order to run workloads in your cluster, you need Nodes. Nodes in your cluster can be *autoscaled* - dynamically *provisioned*, or *consolidated* to provide needed capacity while optimizing cost. Autoscaling is performed by Node *autoscalers*.

Node provisioning

If there are Pods in a cluster that can't be scheduled on existing Nodes, new Nodes can be automatically added to the cluster—*provisioned*—to accommodate the Pods. This is especially useful if the number of Pods changes over time, for example as a result of combining horizontal workload with Node autoscaling.

Autoscalers provision the Nodes by creating and deleting cloud provider resources backing them. Most commonly, the resources backing the Nodes are Virtual Machines.

The main goal of provisioning is to make all Pods schedulable. This goal is not always attainable because of various limitations, including reaching configured provisioning limits, provisioning configuration not being compatible with a particular set of pods, or the lack of cloud provider capacity. While provisioning, Node autoscalers often try to achieve additional goals (for example minimizing the cost of the provisioned Nodes or balancing the number of Nodes between failure domains).

There are two main inputs to a Node autoscaler when determining Nodes to provision—Pod scheduling constraints, and Node constraints imposed by autoscaler configuration.

Autoscaler configuration may also include other Node provisioning triggers (for example the number of Nodes falling below a configured minimum limit).

Note:

Provisioning was formerly known as *scale-up* in Cluster Autoscaler.

Pod scheduling constraints

Pods can express scheduling constraints to impose limitations on the kind of Nodes they can be scheduled on. Node autoscalers take these constraints into account to ensure that the pending Pods can be scheduled on the provisioned Nodes.

The most common kind of scheduling constraints are the resource requests specified by Pod containers. Autoscalers will make sure that the provisioned Nodes have enough resources to satisfy the requests. However, they don't directly take into account the real resource usage of the Pods after they start running. In order to autoscale Nodes based on actual workload resource usage, you can combine horizontal workload autoscaling with Node autoscaling.

Other common Pod scheduling constraints include Node affinity, inter-Pod affinity, or a requirement for a particular storage volume.

Node constraints imposed by autoscaler configuration

The specifics of the provisioned Nodes (for example the amount of resources, the presence of a given label) depend on autoscaler configuration. Autoscalers can either choose them from a pre-defined set of Node configurations, or use auto-provisioning.

Auto-provisioning

Node auto-provisioning is a mode of provisioning in which a user doesn't have to fully configure the specifics of the Nodes that can be provisioned. Instead, the autoscaler dynamically chooses the Node configuration based on the pending Pods it's reacting to, as well as pre-configured constraints (for example, the minimum amount of resources or the need for a given label).

Node consolidation

The main consideration when running a cluster is ensuring that all schedulable pods are running, whilst keeping the cost of the cluster as low as possible. To achieve this, the Pods' resource requests should utilize as much of the Nodes' resources as possible. From this perspective, the overall Node utilization in a cluster can be used as a proxy for how cost-effective the cluster is.

Note:

Correctly setting the resource requests of your Pods is as important to the overall cost-effectiveness of a cluster as optimizing Node utilization. Combining Node autoscaling with vertical workload autoscaling can help you achieve this.

Nodes in your cluster can be automatically *consolidated* in order to improve the overall Node utilization, and in turn the cost-effectiveness of the cluster. Consolidation happens through removing a set of underutilized Nodes from the cluster. Optionally, a different set of Nodes can be provisioned to replace them.

Consolidation, like provisioning, only considers Pod resource requests and not real resource usage when making decisions.

For the purpose of consolidation, a Node is considered *empty* if it only has DaemonSet and static Pods running on it. Removing empty Nodes during consolidation is more straightforward than non-empty ones, and autoscalers often have optimizations designed specifically for consolidating empty Nodes.

Removing non-empty Nodes during consolidation is disruptive—the Pods running on them are terminated, and possibly have to be recreated (for example by a Deployment). However, all such recreated Pods should be able to schedule on existing Nodes in the cluster, or the replacement Nodes provisioned as part of consolidation. **No Pods should normally become pending as a result of consolidation.**

Note:

Autoscalers predict how a recreated Pod will likely be scheduled after a Node is provisioned or consolidated, but they don't control the actual scheduling. Because of this, some Pods might become pending as a result of consolidation - if for example a completely new Pod appears while consolidation is being performed.

Autoscaler configuration may also enable triggering consolidation by other conditions (for example, the time elapsed since a Node was created), in order to optimize different properties (for example, the maximum lifespan of Nodes in a cluster).

The details of how consolidation is performed depend on the configuration of a given autoscaler.

Note:

Consolidation was formerly known as *scale-down* in Cluster Autoscaler.

Autoscalers

The functionalities described in previous sections are provided by Node *autoscalers*. In addition to the Kubernetes API, autoscalers also need to interact with cloud provider APIs to provision and consolidate Nodes. This means that they need to be explicitly integrated with each supported cloud provider. The performance and feature set of a given autoscaler can differ between cloud provider integrations.

```
graph TD na[Node autoscaler] k8s[Kubernetes] cp[Cloud Provider] k8s --> |get Pods/Nodes|na na --> |drain Nodes|k8s na --> |create/remove
resources backing Nodes|cp cp --> |get resources backing Nodes|na classDef white_on_blue fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff;
classDef blue_on_white fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class na blue_on_white; class k8s,cp white_on_blue;
```

Autoscaler implementations

Cluster Autoscaler and Karpenter are the two Node autoscalers currently sponsored by SIG Autoscaling.

From the perspective of a cluster user, both autoscalers should provide a similar Node autoscaling experience. Both will provision new Nodes for unschedulable Pods, and both will consolidate the Nodes that are no longer optimally utilized.

Different autoscalers may also provide features outside the Node autoscaling scope described on this page, and those additional features may differ between them.

Consult the sections below, and the linked documentation for the individual autoscalers to decide which autoscaler fits your use case better.

Cluster Autoscaler

Cluster Autoscaler adds or removes Nodes to pre-configured *Node groups*. Node groups generally map to some sort of cloud provider resource group (most commonly a Virtual Machine group). A single instance of Cluster Autoscaler can simultaneously manage multiple Node groups. When provisioning, Cluster Autoscaler will add Nodes to the group that best fits the requests of pending Pods. When consolidating, Cluster Autoscaler always selects specific Nodes to remove, as opposed to just resizing the underlying cloud provider resource group.

Additional context:

- Documentation overview
- Cloud provider integrations
- Cluster Autoscaler FAQ
- Contact

Karpenter

Karpenter auto-provisions Nodes based on NodePool configurations provided by the cluster operator. Karpenter handles all aspects of node lifecycle, not just autoscaling. This includes automatically refreshing Nodes once they reach a certain lifetime, and auto-upgrading Nodes when new worker Node images are released. It works directly with individual cloud provider resources (most commonly individual Virtual Machines), and doesn't rely on cloud provider resource groups.

Additional context:

- Documentation
- Cloud provider integrations
- Karpenter FAQ
- Contact

Implementation comparison

Main differences between Cluster Autoscaler and Karpenter:

- Cluster Autoscaler provides features related to just Node autoscaling. Karpenter has a wider scope, and also provides features intended for managing Node lifecycle altogether (for example, utilizing disruption to auto-recreate Nodes once they reach a certain lifetime, or auto-upgrade them to new versions).
- Cluster Autoscaler doesn't support auto-provisioning, the Node groups it can provision from have to be pre-configured. Karpenter supports auto-provisioning, so the user only has to configure a set of constraints for the provisioned Nodes, instead of fully configuring homogenous groups.
- Cluster Autoscaler provides cloud provider integrations directly, which means that they're a part of the Kubernetes project. For Karpenter, the Kubernetes project publishes Karpenter as a library that cloud providers can integrate with to build a Node autoscaler.
- Cluster Autoscaler provides integrations with numerous cloud providers, including smaller and less popular providers. There are fewer cloud providers that integrate with Karpenter, including AWS, and Azure.

Combine workload and Node autoscaling

Horizontal workload autoscaling

Node autoscaling usually works in response to Pods—it provisions new Nodes to accommodate unschedulable Pods, and then consolidates the Nodes once they're no longer needed.

Horizontal workload autoscaling automatically scales the number of workload replicas to maintain a desired average resource utilization across the replicas. In other words, it automatically creates new Pods in response to application load, and then removes the Pods once the load decreases.

You can use Node autoscaling together with horizontal workload autoscaling to autoscale the Nodes in your cluster based on the average real resource utilization of your Pods.

If the application load increases, the average utilization of its Pods should also increase, prompting workload autoscaling to create new Pods. Node autoscaling should then provision new Nodes to accommodate the new Pods.

Once the application load decreases, workload autoscaling should remove unnecessary Pods. Node autoscaling should, in turn, consolidate the Nodes that are no longer needed.

If configured correctly, this pattern ensures that your application always has the Node capacity to handle load spikes if needed, but you don't have to pay for the capacity when it's not needed.

Vertical workload autoscaling

When using Node autoscaling, it's important to set Pod resource requests correctly. If the requests of a given Pod are too low, provisioning a new Node for it might not help the Pod actually run. If the requests of a given Pod are too high, it might incorrectly prevent consolidating its Node.

Vertical workload autoscaling automatically adjusts the resource requests of your Pods based on their historical resource usage.

You can use Node autoscaling together with vertical workload autoscaling in order to adjust the resource requests of your Pods while preserving Node autoscaling capabilities in your cluster.

Caution:

When using Node autoscaling, it's not recommended to set up vertical workload autoscaling for DaemonSet Pods. Autoscalers have to predict what DaemonSet Pods on a new Node will look like in order to predict available Node resources. Vertical workload autoscaling might make these predictions unreliable, leading to incorrect scaling decisions.

Related components

This section describes components providing functionality related to Node autoscaling.

Descheduler

The descheduler is a component providing Node consolidation functionality based on custom policies, as well as other features related to optimizing Nodes and Pods (for example deleting frequently restarting Pods).

Workload autoscalers based on cluster size

Cluster Proportional Autoscaler and Cluster Proportional Vertical Autoscaler provide horizontal, and vertical workload autoscaling based on the number of Nodes in the cluster. You can read more in autoscaling based on cluster size.

What's next

- Read about workload-level autoscaling

Compatibility Version For Kubernetes Control Plane Components

Since release v1.32, we introduced configurable version compatibility and emulation options to Kubernetes control plane components to make upgrades safer by providing more control and increasing the granularity of steps available to cluster administrators.

Emulated Version

The emulation option is set by the `--emulated-version` flag of control plane components. It allows the component to emulate the behavior (APIs, features, ...) of an earlier version of Kubernetes.

When used, the capabilities available will match the emulated version:

- Any capabilities present in the binary version that were introduced after the emulation version will be unavailable.
- Any capabilities removed after the emulation version will be available.

This enables a binary from a particular Kubernetes release to emulate the behavior of a previous version with sufficient fidelity that interoperability with other system components can be defined in terms of the emulated version.

The `--emulated-version` must be <= `binaryVersion`. See the help message of the `--emulated-version` flag for supported range of emulated versions.

Organizing Cluster Access Using kubeconfig Files

Use kubeconfig files to organize information about clusters, users, namespaces, and authentication mechanisms. The `kubectl` command-line tool uses kubeconfig files to find the information it needs to choose a cluster and communicate with the API server of a cluster.

Note:

A file that is used to configure access to clusters is called a *kubeconfig file*. This is a generic way of referring to configuration files. It does not mean that there is a file named `kubeconfig`.

Warning:

Only use kubeconfig files from trusted sources. Using a specially-crafted kubeconfig file could result in malicious code execution or file exposure. If you must use an untrusted kubeconfig file, inspect it carefully first, much as you would a shell script.

By default, `kubectl` looks for a file named `config` in the `$HOME/.kube` directory. You can specify other kubeconfig files by setting the `KUBECONFIG` environment variable or by setting the `--kubeconfig` flag.

For step-by-step instructions on creating and specifying kubeconfig files, see Configure Access to Multiple Clusters.

Supporting multiple clusters, users, and authentication mechanisms

Suppose you have several clusters, and your users and components authenticate in a variety of ways. For example:

- A running kubelet might authenticate using certificates.

- A user might authenticate using tokens.
- Administrators might have sets of certificates that they provide to individual users.

With kubeconfig files, you can organize your clusters, users, and namespaces. You can also define contexts to quickly and easily switch between clusters and namespaces.

Context

A *context* element in a kubeconfig file is used to group access parameters under a convenient name. Each context has three parameters: cluster, namespace, and user. By default, the `kubectl` command-line tool uses parameters from the *current context* to communicate with the cluster.

To choose the current context:

```
kubectl config use-context
```

The KUBECONFIG environment variable

The `KUBECONFIG` environment variable holds a list of kubeconfig files. For Linux and Mac, the list is colon-delimited. For Windows, the list is semicolon-delimited. The `KUBECONFIG` environment variable is not required. If the `KUBECONFIG` environment variable doesn't exist, `kubectl` uses the default kubeconfig file, `$HOME/.kube/config`.

If the `KUBECONFIG` environment variable does exist, `kubectl` uses an effective configuration that is the result of merging the files listed in the `KUBECONFIG` environment variable.

Merging kubeconfig files

To see your configuration, enter this command:

```
kubectl config view
```

As described previously, the output might be from a single kubeconfig file, or it might be the result of merging several kubeconfig files.

Here are the rules that `kubectl` uses when it merges kubeconfig files:

1. If the `--kubeconfig` flag is set, use only the specified file. Do not merge. Only one instance of this flag is allowed.

   Otherwise, if the `KUBECONFIG` environment variable is set, use it as a list of files that should be merged. Merge the files listed in the `KUBECONFIG` environment variable according to these rules:

   - Ignore empty filenames.
   - Produce errors for files with content that cannot be deserialized.
   - The first file to set a particular value or map key wins.
   - Never change the value or map key. Example: Preserve the context of the first file to set `current-context`. Example: If two files specify a `red-user`, use only values from the first file's `red-user`. Even if the second file has non-conflicting entries under `red-user`, discard them.

   For an example of setting the `KUBECONFIG` environment variable, see [Setting the KUBECONFIG environment variable](#).

   Otherwise, use the default kubeconfig file, `$HOME/.kube/config`, with no merging.

2. Determine the context to use based on the first hit in this chain:

   1. Use the `--context` command-line flag if it exists.
   2. Use the `current-context` from the merged kubeconfig files.

   An empty context is allowed at this point.

3. Determine the cluster and user. At this point, there might or might not be a context. Determine the cluster and user based on the first hit in this chain, which is run twice: once for user and once for cluster:

   1. Use a command-line flag if it exists: `--user` or `--cluster`.
   2. If the context is non-empty, take the user or cluster from the context.

   The user and cluster can be empty at this point.

4. Determine the actual cluster information to use. At this point, there might or might not be cluster information. Build each piece of the cluster information based on this chain; the first hit wins:

   1. Use command line flags if they exist: `--server`, `--certificate-authority`, `--insecure-skip-tls-verify`.
   2. If any cluster information attributes exist from the merged kubeconfig files, use them.
   3. If there is no server location, fail.

5. Determine the actual user information to use. Build user information using the same rules as cluster information, except allow only one authentication technique per user:

   1. Use command line flags if they exist: `--client-certificate`, `--client-key`, `--username`, `--password`, `--token`.
   2. Use the `user` fields from the merged kubeconfig files.
   3. If there are two conflicting techniques, fail.

6. For any information still missing, use default values and potentially prompt for authentication information.

File references

File and path references in a kubeconfig file are relative to the location of the kubeconfig file. File references on the command line are relative to the current working directory. In `$HOME/.kube/config`, relative paths are stored relatively, and absolute paths are stored absolutely.

Proxy

You can configure `kubectl` to use a proxy per cluster using `proxy-url` in your kubeconfig file, like this:

```
apiVersion: v1
kind: Configclusters:- cluster:    proxy-url: http://proxy.example.org:3128    server: https://k8s.example.org/k8s/clusters/c-xxyy
```

What's next

- Configure Access to Multiple Clusters
- `kubectl config`

Proxies in Kubernetes

This page explains proxies used with Kubernetes.

Proxies

There are several different proxies you may encounter when using Kubernetes:

1. The kubectl proxy:
   - runs on a user's desktop or in a pod
   - proxies from a localhost address to the Kubernetes apiserver
   - client to proxy uses HTTP
   - proxy to apiserver uses HTTPS
   - locates apiserver
   - adds authentication headers
2. The apiserver proxy:
   - is a bastion built into the apiserver
   - connects a user outside of the cluster to cluster IPs which otherwise might not be reachable
   - runs in the apiserver processes
   - client to proxy uses HTTPS (or http if apiserver so configured)
   - proxy to target may use HTTP or HTTPS as chosen by proxy using available information
   - can be used to reach a Node, Pod, or Service
   - does load balancing when used to reach a Service
3. The kube proxy:
   - runs on each node
   - proxies UDP, TCP and SCTP
   - does not understand HTTP
   - provides load balancing
   - is only used to reach services
4. A Proxy/Load-balancer in front of apiserver(s):
   - existence and implementation varies from cluster to cluster (e.g. nginx)
   - sits between all clients and one or more apiservers
   - acts as load balancer if there are several apiservers.
5. Cloud Load Balancers on external services:
   - are provided by some cloud providers (e.g. AWS ELB, Google Cloud Load Balancer)
   - are created automatically when the Kubernetes service has type `LoadBalancer`
   - usually supports UDP/TCP only
   - SCTP support is up to the load balancer implementation of the cloud provider
   - implementation varies by cloud provider.

Kubernetes users will typically not need to worry about anything other than the first two types. The cluster admin will typically ensure that the latter types are set up correctly.

Requesting redirects

Proxies have replaced redirect capabilities. Redirects have been deprecated.

Cluster Administration

Lower-level detail relevant to creating or administering a Kubernetes cluster.

The cluster administration overview is for anyone creating or administering a Kubernetes cluster. It assumes some familiarity with core Kubernetes concepts.

Planning a cluster

See the guides in Setup for examples of how to plan, set up, and configure Kubernetes clusters. The solutions listed in this article are called *distros*.

Note:

Not all distros are actively maintained. Choose distros which have been tested with a recent version of Kubernetes.

Before choosing a guide, here are some considerations:

- Do you want to try out Kubernetes on your computer, or do you want to build a high-availability, multi-node cluster? Choose distros best suited for your needs.
- Will you be using **a hosted Kubernetes cluster**, such as Google Kubernetes Engine, or **hosting your own cluster**?
- Will your cluster be **on-premises**, or **in the cloud (IaaS)**? Kubernetes does not directly support hybrid clusters. Instead, you can set up multiple clusters.
- **If you are configuring Kubernetes on-premises**, consider which networking model fits best.
- Will you be running Kubernetes on **"bare metal" hardware** or on **virtual machines (VMs)**?
- Do you **want to run a cluster**, or do you expect to do **active development of Kubernetes project code**? If the latter, choose an actively-developed distro. Some distros only use binary releases, but offer a greater variety of choices.
- Familiarize yourself with the components needed to run a cluster.

Managing a cluster

- Learn how to manage nodes.

    - Read about Node autoscaling.

- Learn how to set up and manage the resource quota for shared clusters.

Securing a cluster

- Generate Certificates describes the steps to generate certificates using different tool chains.

- Kubernetes Container Environment describes the environment for Kubelet managed containers on a Kubernetes node.

- Controlling Access to the Kubernetes API describes how Kubernetes implements access control for its own API.

- Authenticating explains authentication in Kubernetes, including the various authentication options.

- Authorization is separate from authentication, and controls how HTTP calls are handled.

- Using Admission Controllers explains plug-ins which intercepts requests to the Kubernetes API server after authentication and authorization.

- Admission Webhook Good Practices provides good practices and considerations when designing mutating admission webhooks and validating admission webhooks.

- Using Sysctls in a Kubernetes Cluster describes to an administrator how to use the `sysctl` command-line tool to set kernel parameters .

- Auditing describes how to interact with Kubernetes' audit logs.

Securing the kubelet

- Control Plane-Node communication
- TLS bootstrapping
- Kubelet authentication/authorization

Optional Cluster Services

- DNS Integration describes how to resolve a DNS name directly to a Kubernetes service.

- Logging and Monitoring Cluster Activity explains how logging in Kubernetes works and how to implement it.