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 v1.35, 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](disabled by default)

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 v1.35 [stable]

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 v1.35 [beta](enabled by default)
>
> Access to pre-pulled images may be authorized according to image pull credential verification.

Ensure image pull credential verification

ⓘ **FEATURE STATE:** Kubernetes v1.35 [beta](enabled by default)

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.

Enabling `KubeletEnsureSecretPulledImages` for the first time

When the `KubeletEnsureSecretPulledImages` gets enabled for the first time, either by a kubelet upgrade or by explicitly enabling the feature, if a kubelet is able to access any images at that time, these will all be considered pre-pulled. This happens because in this case the kubelet has no records about the images being pulled. The kubelet will only be able to start making image pull records as any image gets pulled for the first time.

If this is a concern, it is advised to clean up nodes of all images that should not be considered pre-pulled before enabling the feature.

Note that removing the directory holding the image pulled records will have the same effect on kubelet restart, particularly the images currently cached in the nodes by the container runtime will all be considered pre-pulled.

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-email=<docker-email>
```

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.

Feedback

Was this page helpful?

Yes No

Last modified November 18, 2025 at 10:38 AM PST: images: describe enabling KubeletEnsureSecretPulledImages for the first time (c43957b9ec)