Pods

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

As well as application containers, a Pod can contain init containers that run during Pod startup. You can also inject ephemeral containers for debugging if your cluster offers this.

What is a Pod?

Note: While Kubernetes supports more container runtimes than just Docker, Docker is the most commonly known runtime, and it helps to describe Pods using some terminology from Docker.

The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a Docker container. Within a Pod's context, the individual applications may have further sub-isolations applied.

In terms of Docker concepts, a Pod is similar to a group of Docker containers with shared namespaces and shared filesystem volumes.

Using Pods

Usually you don't need to create Pods directly, even singleton Pods. Instead, create them using workload resources such as Deployment or Job. If your Pods need to track state, consider the StatefulSet resource.

Pods in a Kubernetes cluster are used in two main ways:

- **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.

- **Pods that run multiple containers that need to work together**. A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service —for example, one container serving data stored in a shared volume to the public, while a separate *sidecar* container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.

  Note: Grouping multiple co-located and co-managed containers in a single Pod is a relatively advanced use case. You should use this pattern only in specific instances in which your containers are tightly coupled.

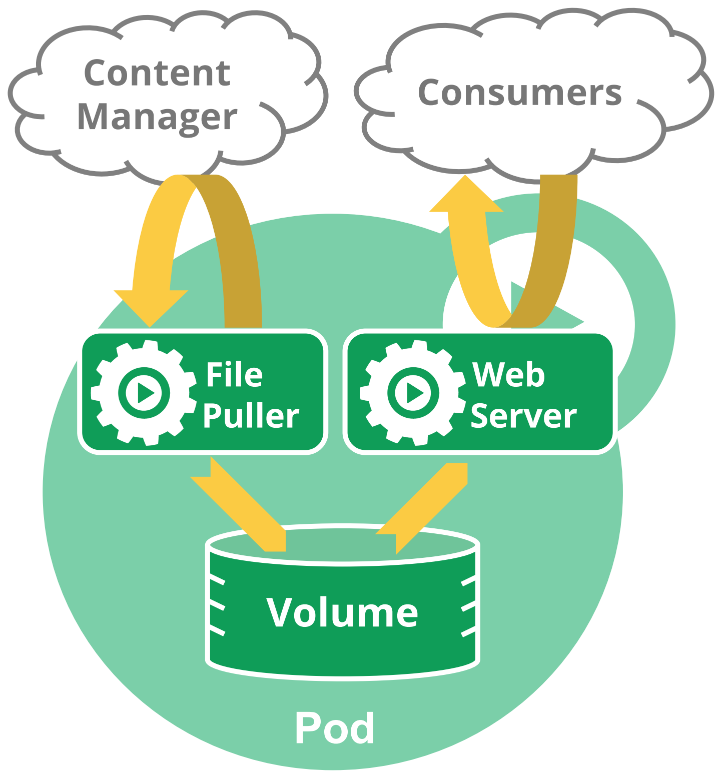
Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (to provide more overall resources by running more instances), you should use multiple Pods, one for each instance. In Kubernetes, this is typically referred to as *replication*. Replicated Pods are usually created and managed as a group by a workload resource and its controller.

See Pods and controllers for more information on how Kubernetes uses workload resources, and their controllers, to implement application scaling and auto-healing.

How Pods manage multiple containers

Pods are designed to support multiple cooperating processes (as containers) that form a cohesive unit of service. The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated.

For example, you might have a container that acts as a web server for files in a shared volume, and a separate "sidecar" container that updates those files from a remote source, as in the following diagram:



Some Pods have init containers as well as app containers. Init containers run and complete before the app containers are started.

Pods natively provide two kinds of shared resources for their constituent containers: networking and storage.

Working with Pods

You'll rarely create individual Pods directly in Kubernetes—even singleton Pods. This is because Pods are designed as relatively ephemeral, disposable entities. When a Pod gets created (directly by you, or indirectly by a controller), the new Pod is scheduled to run on a Node in your cluster. The Pod remains on that node until the Pod finishes execution, the Pod object is deleted, the Pod is *evicted* for lack of resources, or the node fails.

> **Note:** Restarting a container in a Pod should not be confused with restarting a Pod. A Pod is not a process, but an environment for running container(s). A Pod persists until it is deleted.

When you create the manifest for a Pod object, make sure the name specified is a valid DNS subdomain name.

Pods and controllers

You can use workload resources to create and manage multiple Pods for you. A controller for the resource handles replication and rollout and automatic healing in case of Pod failure. For example, if a Node fails, a controller notices that Pods on that Node have stopped working and creates a replacement Pod. The scheduler places the replacement Pod onto a healthy Node.

Here are some examples of workload resources that manage one or more Pods:

- Deployment
- StatefulSet
- DaemonSet

Pod templates

Controllers for workload resources create Pods from a *pod template* and manage those Pods on your behalf.

PodTemplates are specifications for creating Pods, and are included in workload resources such as Deployments, Jobs, and DaemonSets.

Each controller for a workload resource uses the `PodTemplate` inside the workload object to make actual Pods. The `PodTemplate` is part of the desired state of whatever workload resource you used to run your app.

The sample below is a manifest for a simple Job with a `template` that starts one container. The container in that Pod prints a message then pauses.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # This is the pod template
    spec:
      containers:
      - name: hello
        image: busybox
        command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
      restartPolicy: OnFailure
    # The pod template ends here
```

Modifying the pod template or switching to a new pod template has no direct effect on the Pods that already exist. If you change the pod template for a workload resource, that resource needs to create replacement Pods that use the updated template.

For example, the StatefulSet controller ensures that the running Pods match the current pod template for each StatefulSet object. If you edit the StatefulSet to change its pod template, the StatefulSet starts to create new Pods based on the updated template. Eventually, all of the old Pods are replaced with new Pods, and the update is complete.

Each workload resource implements its own rules for handling changes to the Pod template. If you want to read more about StatefulSet specifically, read Update strategy in the StatefulSet Basics tutorial.

On Nodes, the kubelet does not directly observe or manage any of the details around pod templates and updates; those details are abstracted away. That abstraction and separation of concerns simplifies system semantics, and makes it feasible to extend the cluster's behavior without changing existing code.

Pod update and replacement

As mentioned in the previous section, when the Pod template for a workload resource is changed, the controller creates new Pods based on the updated template instead of updating or patching the existing Pods.

Kubernetes doesn't prevent you from managing Pods directly. It is possible to update some fields of a running Pod, in place. However, Pod update operations like `patch`, and `replace` have some limitations:

- Most of the metadata about a Pod is immutable. For example, you cannot change the `namespace`, `name`, `uid`, or `creationTimestamp` fields; the `generation` field is unique. It only accepts updates that increment the field's current value.
- If the `metadata.deletionTimestamp` is set, no new entry can be added to the `metadata.finalizers` list.

- Pod updates may not change fields other than `spec.containers[*].image`, `spec.initContainers[*].image`, `spec.activeDeadlineSeconds` or `spec.tolerations`. For `spec.tolerations`, you can only add new entries.
- When updating the `spec.activeDeadlineSeconds` field, two types of updates are allowed:
  1. setting the unassigned field to a positive number;
  2. updating the field from a positive number to a smaller, non-negative number.

Resource sharing and communication

Pods enable data sharing and communication among their constituent containers.

Storage in Pods

A Pod can specify a set of shared storage volumes. All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted. See Storage for more information on how Kubernetes implements shared storage and makes it available to Pods.

Pod networking

Each Pod is assigned a unique IP address for each address family. Every container in a Pod shares the network namespace, including the IP address and network ports. Inside a Pod (and **only** then), the containers that belong to the Pod can communicate with one another using `localhost`. When containers in a Pod communicate with entities *outside the Pod*, they must coordinate how they use the shared network resources (such as ports). Within a Pod, containers share an IP address and port space, and can find each other via `localhost`. The containers in a Pod can also communicate with each other using standard inter-process communications like SystemV semaphores or POSIX shared memory. Containers in different Pods have distinct IP addresses and can not communicate by IPC without special configuration. Containers that want to interact with a container running in a different Pod can use IP networking to communicate.

Containers within the Pod see the system hostname as being the same as the configured `name` for the Pod. There's more about this in the networking section.

Privileged mode for containers

In Linux, any container in a Pod can enable privileged mode using the `privileged` (Linux) flag on the security context of the container spec. This is useful for containers that want to use operating system administrative capabilities such as manipulating the network stack or accessing hardware devices.

If your cluster has the `WindowsHostProcessContainers` feature enabled, you can create a Windows HostProcess pod by setting the `windowsOptions.hostProcess` flag on the security context of the pod spec. All containers in these pods must run as Windows HostProcess containers. HostProcess pods run directly on the host and can also be used to perform administrative tasks as is done with Linux privileged containers.

Note: Your container runtime must support the concept of a privileged container for this setting to be relevant.

Static Pods

Static Pods are managed directly by the kubelet daemon on a specific node, without the API server observing them. Whereas most Pods are managed by the control plane (for example, a Deployment), for static Pods, the kubelet directly supervises each static Pod (and restarts it if it fails).

Static Pods are always bound to one Kubelet on a specific node. The main use for static Pods is to run a self-hosted control plane: in other words, using the kubelet to supervise the individual control plane components.

The kubelet automatically tries to create a mirror Pod on the Kubernetes API server for each static Pod. This means that the Pods running on a node are visible on the API server, but cannot be controlled from there.

Note: The `spec` of a static Pod cannot refer to other API objects (e.g., ServiceAccount, ConfigMap, Secret, etc).

Container probes

A *probe* is a diagnostic performed periodically by the kubelet on a container. To perform a diagnostic, the kubelet can invoke different actions:

- `ExecAction` (performed with the help of the container runtime)
- `TCPSocketAction` (checked directly by the kubelet)
- `HTTPGetAction` (checked directly by the kubelet)

You can read more about probes in the Pod Lifecycle documentation.

What's next

- Learn about the lifecycle of a Pod.
- Learn about RuntimeClass and how you can use it to configure different Pods with different container runtime configurations.
- Read about Pod topology spread constraints.
- Read about PodDisruptionBudget and how you can use it to manage application availability during disruptions.
- Pod is a top-level resource in the Kubernetes REST API. The Pod object definition describes the object in detail.
- The Distributed System Toolkit: Patterns for Composite Containers explains common layouts for Pods with more than one container.

To understand the context for why Kubernetes wraps a common Pod API in other resources (such as StatefulSets or Deployments), you can read about the prior art, including:

- Aurora
- Borg
- Marathon
- Omega
- Tupperware.

1 - Pod Lifecycle

This page describes the lifecycle of a Pod. Pods follow a defined lifecycle, starting in the `Pending` phase, moving through `Running` if at least one of its primary containers starts OK, and then through either the `Succeeded` or `Failed` phases depending on whether any container in the Pod terminated in failure.

Whilst a Pod is running, the kubelet is able to restart containers to handle some kind of faults. Within a Pod, Kubernetes tracks different container states and determines what action to take to make the Pod healthy again.

In the Kubernetes API, Pods have both a specification and an actual status. The status for a Pod object consists of a set of Pod conditions. You can also inject custom readiness information into the condition data for a Pod, if that is useful to your application.

Pods are only scheduled once in their lifetime. Once a Pod is scheduled (assigned) to a Node, the Pod runs on that Node until it stops or is terminated.

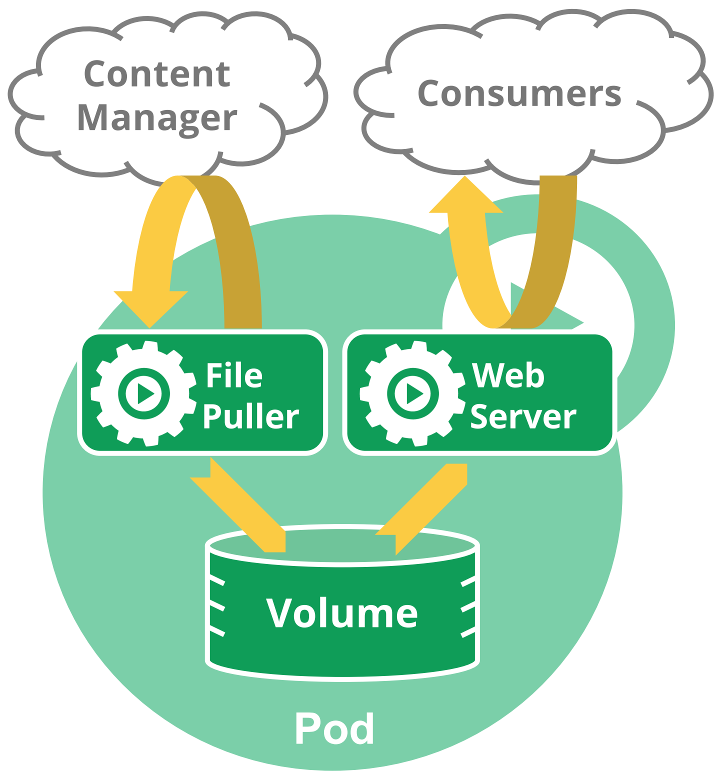
Pod lifetime

Like individual application containers, Pods are considered to be relatively ephemeral (rather than durable) entities. Pods are created, assigned a unique ID (UID), and scheduled to nodes where they remain until termination (according to restart policy) or deletion. If a Node dies, the Pods scheduled to that node are scheduled for deletion after a timeout period.

Pods do not, by themselves, self-heal. If a Pod is scheduled to a node that then fails, the Pod is deleted; likewise, a Pod won't survive an eviction due to a lack of resources or Node maintenance. Kubernetes uses a higher-level abstraction, called a controller, that handles the work of managing the relatively disposable Pod instances.

A given Pod (as defined by a UID) is never "rescheduled" to a different node; instead, that Pod can be replaced by a new, near-identical Pod, with even the same name if desired, but with a different UID.

When something is said to have the same lifetime as a Pod, such as a volume, that means that the thing exists as long as that specific Pod (with that exact UID) exists. If that Pod is deleted for any reason, and even if an identical replacement is created, the related thing (a volume, in this example) is also destroyed and created anew.



Pod diagram

A multi-container Pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers.

Pod phase

A Pod's `status` field is a PodStatus object, which has a `phase` field.

The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle. The phase is not intended to be a comprehensive rollup of observations of container or Pod state, nor is it intended to be a comprehensive state machine.

The number and meanings of Pod phase values are tightly guarded. Other than what is documented here, nothing should be assumed about Pods that have a given `phase` value.

Here are the possible values for `phase`:

Value	Description
`Pending`	The Pod has been accepted by the Kubernetes cluster, but one or more of the containers has not been set up and made ready to run. This includes time a Pod spends waiting to be scheduled as well as the time spent downloading container images over the network.
`Running`	The Pod has been bound to a node, and all of the containers have been created. At least one container is still running, or is in the process of starting or restarting.
`Succeeded`	All containers in the Pod have terminated in success, and will not be restarted.
`Failed`	All containers in the Pod have terminated, and at least one container has terminated in failure. That is, the container either exited with non-zero status or was terminated by the system.
`Unknown`	For some reason the state of the Pod could not be obtained. This phase typically occurs due to an error in communicating with the node where the Pod should be running.

Note: When a Pod is being deleted, it is shown as `Terminating` by some kubectl commands. This `Terminating` status is not one of the Pod phases. A Pod is granted a term to terminate gracefully, which defaults to 30 seconds. You can use the flag `--force` to terminate a Pod by force.

If a node dies or is disconnected from the rest of the cluster, Kubernetes applies a policy for setting the `phase` of all Pods on the lost node to Failed.

Container states

As well as the phase of the Pod overall, Kubernetes tracks the state of each container inside a Pod. You can use container lifecycle hooks to trigger events to run at certain points in a container's lifecycle.

Once the scheduler assigns a Pod to a Node, the kubelet starts creating containers for that Pod using a container runtime. There are three possible container states: `Waiting`, `Running`, and `Terminated`.

To check the state of a Pod's containers, you can use `kubectl describe pod <name-of-pod>`. The output shows the state for each container within that Pod.

Each state has a specific meaning:

`Waiting`

If a container is not in either the `Running` or `Terminated` state, it is `Waiting`. A container in the `Waiting` state is still running the operations it requires in order to complete start up: for example, pulling the container image from a container image registry, or applying Secret data.

When you use `kubectl` to query a Pod with a container that is `Waiting`, you also see a Reason field to summarize why the container is in that state.

`Running`

The `Running` status indicates that a container is executing without issues. If there was a `postStart` hook configured, it has already executed and finished. When you use `kubectl` to query a Pod with a container that is `Running`, you also see information about when the container entered the `Running` state.

`Terminated`

A container in the `Terminated` state began execution and then either ran to completion or failed for some reason. When you use `kubectl` to query a Pod with a container that is `Terminated`, you see a reason, an exit code, and the start and finish time for that container's period of execution.

If a container has a `preStop` hook configured, that runs before the container enters the `Terminated` state.

Container restart policy

The `spec` of a Pod has a `restartPolicy` field with possible values Always, OnFailure, and Never. The default value is Always.

The `restartPolicy` applies to all containers in the Pod. `restartPolicy` only refers to restarts of the containers by the kubelet on the same node. After containers in a Pod exit, the kubelet restarts them with an exponential back-off delay (10s, 20s, 40s, …), that is capped at five minutes. Once a container has executed for 10 minutes without any problems, the kubelet resets the restart backoff timer for that container.

Pod conditions

A Pod has a PodStatus, which has an array of PodConditions through which the Pod has or has not passed:

- `PodScheduled`: the Pod has been scheduled to a node.
- `ContainersReady`: all containers in the Pod are ready.
- `Initialized`: all init containers have started successfully.
- `Ready`: the Pod is able to serve requests and should be added to the load balancing pools of all matching Services.

Field name	**Description**
`type`	Name of this Pod condition.
`status`	Indicates whether that condition is applicable, with possible values "`True`", "`False`", or "`Unknown`".
`lastProbeTime`	Timestamp of when the Pod condition was last probed.
`lastTransitionTime`	Timestamp for when the Pod last transitioned from one status to another.
`reason`	Machine-readable, UpperCamelCase text indicating the reason for the condition's last transition.

Field name	Description
`message`	Human-readable message indicating details about the last status transition.

Pod readiness

FEATURE STATE: `Kubernetes v1.14 [stable]`

Your application can inject extra feedback or signals into PodStatus: *Pod readiness*. To use this, set `readinessGates` in the Pod's `spec` to specify a list of additional conditions that the kubelet evaluates for Pod readiness.

Readiness gates are determined by the current state of `status.condition` fields for the Pod. If Kubernetes cannot find such a condition in the `status.conditions` field of a Pod, the status of the condition is defaulted to "`False`".

Here is an example:

```
kind: Pod
...
spec:
  readinessGates:
    - conditionType: "www.example.com/feature-1"
status:
  conditions:
    - type: Ready                              # a built in PodCondition
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2018-01-01T00:00:00Z
    - type: "www.example.com/feature-1"        # an extra PodCondition
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2018-01-01T00:00:00Z
  containerStatuses:
    - containerID: docker://abcd...
      ready: true
...
```

The Pod conditions you add must have names that meet the Kubernetes label key format.

Status for Pod readiness

The `kubectl patch` command does not support patching object status. To set these `status.conditions` for the pod, applications and operators should use the `PATCH` action. You can use a Kubernetes client library to write code that sets custom Pod conditions for Pod readiness.

For a Pod that uses custom conditions, that Pod is evaluated to be ready **only** when both the following statements apply:

- All containers in the Pod are ready.
- All conditions specified in `readinessGates` are `True`.

When a Pod's containers are Ready but at least one custom condition is missing or `False`, the kubelet sets the Pod's condition to `ContainersReady`.

Container probes

A Probe is a diagnostic performed periodically by the kubelet on a Container. To perform a diagnostic, the kubelet calls a Handler implemented by the container. There are three types of handlers:

- ExecAction: Executes a specified command inside the container. The diagnostic is considered successful if the command exits with a status code of 0.
- TCPSocketAction: Performs a TCP check against the Pod's IP address on a specified port. The diagnostic is considered successful if the port is open.
- HTTPGetAction: Performs an HTTP `GET` request against the Pod's IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

Each probe has one of three results:

- `Success` : The container passed the diagnostic.
- `Failure` : The container failed the diagnostic.
- `Unknown` : The diagnostic failed, so no action should be taken.

The kubelet can optionally perform and react to three kinds of probes on running containers:

- `livenessProbe` : Indicates whether the container is running. If the liveness probe fails, the kubelet kills the container, and the container is subjected to its restart policy. If a Container does not provide a liveness probe, the default state is `Success` .
- `readinessProbe` : Indicates whether the container is ready to respond to requests. If the readiness probe fails, the endpoints controller removes the Pod's IP address from the endpoints of all Services that match the Pod. The default state of readiness before the initial delay is `Failure` . If a Container does not provide a readiness probe, the default state is `Success` .
- `startupProbe` : Indicates whether the application within the container is started. All other probes are disabled if a startup probe is provided, until it succeeds. If the startup probe fails, the kubelet kills the container, and the container is subjected to its restart policy. If a Container does not provide a startup probe, the default state is `Success` .

For more information about how to set up a liveness, readiness, or startup probe, see Configure Liveness, Readiness and Startup Probes.

When should you use a liveness probe?

FEATURE STATE: `Kubernetes v1.0 [stable]`

If the process in your container is able to crash on its own whenever it encounters an issue or becomes unhealthy, you do not necessarily need a liveness probe; the kubelet will automatically perform the correct action in accordance with the Pod's `restartPolicy` .

If you'd like your container to be killed and restarted if a probe fails, then specify a liveness probe, and specify a `restartPolicy` of Always or OnFailure.

When should you use a readiness probe?

FEATURE STATE: `Kubernetes v1.0 [stable]`

If you'd like to start sending traffic to a Pod only when a probe succeeds, specify a readiness probe. In this case, the readiness probe might be the same as the liveness probe, but the existence of the readiness probe in the spec means that the Pod will start without receiving any traffic and only start receiving traffic after the probe starts succeeding.

If you want your container to be able to take itself down for maintenance, you can specify a readiness probe that checks an endpoint specific to readiness that is different from the liveness probe.

If your app has a strict dependency on back-end services, you can implement both a liveness and a readiness probe. The liveness probe passes when the app itself is healthy, but the readiness probe additionally checks that each required back-end service is available. This helps you avoid directing traffic to Pods that can only respond with error messages.

If your container needs to work on loading large data, configuration files, or migrations during startup, you can use a startup probe. However, if you want to detect the difference between an app that has failed and an app that is still processing its startup data, you might prefer a readiness probe.

> **Note:** If you want to be able to drain requests when the Pod is deleted, you do not necessarily need a readiness probe; on deletion, the Pod automatically puts itself into an unready state regardless of whether the readiness probe exists. The Pod remains in the unready state while it waits for the containers in the Pod to stop.

When should you use a startup probe?

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

Startup probes are useful for Pods that have containers that take a long time to come into service. Rather than set a long liveness interval, you can configure a separate configuration for probing the container as it starts up, allowing a time longer than the liveness interval would allow.

If your container usually starts in more than `initialDelaySeconds + failureThreshold × periodSeconds` , you should specify a startup probe that checks the same endpoint as the liveness probe. The default for `periodSeconds` is 10s. You should then set its `failureThreshold` high enough to allow the container to start, without changing the default values of the liveness probe. This helps to protect against deadlocks.

Termination of Pods

Because Pods represent processes running on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed (rather than being abruptly stopped with a `KILL` signal and having no chance to clean up).

The design aim is for you to be able to request deletion and know when processes terminate, but also be able to ensure that deletes eventually complete. When you request deletion of a Pod, the cluster records and tracks the intended grace period before the Pod is allowed to be forcefully killed. With that forceful shutdown tracking in place, the kubelet attempts graceful shutdown.

Typically, the container runtime sends a TERM signal to the main process in each container. Many container runtimes respect the `STOPSIGNAL` value defined in the container image and send this instead of TERM. Once the grace period has expired, the KILL signal is sent to any remaining processes, and the Pod is then deleted from the API Server. If the kubelet or the container runtime's management service is restarted while waiting for processes to terminate, the cluster retries from the start including the full original grace period.

An example flow:

1. You use the `kubectl` tool to manually delete a specific Pod, with the default grace period (30 seconds).
2. The Pod in the API server is updated with the time beyond which the Pod is considered "dead" along with the grace period. If you use `kubectl describe` to check on the Pod you're deleting, that Pod shows up as "Terminating". On the node where the Pod is running: as soon as the kubelet sees that a Pod has been marked as terminating (a graceful shutdown duration has been set), the kubelet begins the local Pod shutdown process.
    1. If one of the Pod's containers has defined a `preStop` hook, the kubelet runs that hook inside of the container. If the `preStop` hook is still running after the grace period expires, the kubelet requests a small, one-off grace period extension of 2 seconds.

        > **Note:** If the `preStop` hook needs longer to complete than the default grace period allows, you must modify `terminationGracePeriodSeconds` to suit this.

    2. The kubelet triggers the container runtime to send a TERM signal to process 1 inside each container.

> **Note:** The containers in the Pod receive the TERM signal at different times and in an arbitrary order. If the order of shutdowns matters, consider using a `preStop` hook to synchronize.

3. At the same time as the kubelet is starting graceful shutdown, the control plane removes that shutting-down Pod from Endpoints (and, if enabled, EndpointSlice) objects where these represent a Service with a configured selector. ReplicaSets and other workload resources no longer treat the shutting-down Pod as a valid, in-service replica. Pods that shut down slowly cannot continue to serve traffic as load balancers (like the service proxy) remove the Pod from the list of endpoints as soon as the termination grace period *begins*.
4. When the grace period expires, the kubelet triggers forcible shutdown. The container runtime sends `SIGKILL` to any processes still running in any container in the Pod. The kubelet also cleans up a hidden `pause` container if that container runtime uses one.
5. The kubelet triggers forcible removal of Pod object from the API server, by setting grace period to 0 (immediate deletion).
6. The API server deletes the Pod's API object, which is then no longer visible from any client.

Forced Pod termination

> **Caution:** Forced deletions can be potentially disruptive for some workloads and their Pods.

By default, all deletes are graceful within 30 seconds. The `kubectl delete` command supports the `--grace-period=<seconds>` option which allows you to override the default and specify your own value.

Setting the grace period to `0` forcibly and immediately deletes the Pod from the API server. If the pod was still running on a node, that forcible deletion triggers the kubelet to begin immediate cleanup.

> **Note:** You must specify an additional flag `--force` along with `--grace-period=0` in order to perform force deletions.

When a force deletion is performed, the API server does not wait for confirmation from the kubelet that the Pod has been terminated on the node it was running on. It removes the Pod in the API immediately so a new Pod can be created with the same name. On the node, Pods that are set to terminate immediately will still be given a small grace period before being force killed.

If you need to force-delete Pods that are part of a StatefulSet, refer to the task documentation for deleting Pods from a StatefulSet.

Garbage collection of failed Pods

For failed Pods, the API objects remain in the cluster's API until a human or controller process explicitly removes them.

The control plane cleans up terminated Pods (with a phase of `Succeeded` or `Failed`), when the number of Pods exceeds the configured threshold (determined by `terminated-pod-gc-threshold` in the kube-controller-manager). This avoids a resource leak as Pods are created and terminated over time.

What's next

- Get hands-on experience attaching handlers to Container lifecycle events.
- Get hands-on experience configuring Liveness, Readiness and Startup Probes.
- Learn more about container lifecycle hooks.
- For detailed information about Pod / Container status in the API, see PodStatus and ContainerStatus.

2 - Init Containers

This page provides an overview of init containers: specialized containers that run before app containers in a Pod. Init containers can contain utilities or setup scripts not present in an app image.

You can specify init containers in the Pod specification alongside the `containers` array (which describes app containers).

Understanding init containers

A Pod can have multiple containers running apps within it, but it can also have one or more init containers, which are run before the app containers are started.

Init containers are exactly like regular containers, except:

- Init containers always run to completion.
- Each init container must complete successfully before the next one starts.

If a Pod's init container fails, the kubelet repeatedly restarts that init container until it succeeds. However, if the Pod has a `restartPolicy` of Never, and an init container fails during startup of that Pod, Kubernetes treats the overall Pod as failed.

To specify an init container for a Pod, add the `initContainers` field into the Pod specification, as an array of `container` items (similar to the app `containers` field and its contents). See Container in the API reference for more details.

The status of the init containers is returned in `.status.initContainerStatuses` field as an array of the container statuses (similar to the `.status.containerStatuses` field).

Differences from regular containers

Init containers support all the fields and features of app containers, including resource limits, volumes, and security settings. However, the resource requests and limits for an init container are handled differently, as documented in Resources.

Also, init containers do not support `lifecycle`, `livenessProbe`, `readinessProbe`, or `startupProbe` because they must run to completion before the Pod can be ready.

If you specify multiple init containers for a Pod, kubelet runs each init container sequentially. Each init container must succeed before the next can run. When all of the init containers have run to completion, kubelet initializes the application containers for the Pod and runs them as usual.

Using init containers

Because init containers have separate images from app containers, they have some advantages for start-up related code:

- Init containers can contain utilities or custom code for setup that are not present in an app image. For example, there is no need to make an image `FROM` another image just to use a tool like `sed`, `awk`, `python`, or `dig` during setup.
- The application image builder and deployer roles can work independently without the need to jointly build a single app image.
- Init containers can run with a different view of the filesystem than app containers in the same Pod. Consequently, they can be given access to Secrets that app containers cannot access.
- Because init containers run to completion before any app containers start, init containers offer a mechanism to block or delay app container startup until a set of preconditions are met. Once preconditions are met, all of the app containers in a Pod can start in parallel.

- Init containers can securely run utilities or custom code that would otherwise make an app container image less secure. By keeping unnecessary tools separate you can limit the attack surface of your app container image.

Examples

Here are some ideas for how to use init containers:

- Wait for a Service to be created, using a shell one-line command like:

```
for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; done; exit 1
```

- Register this Pod with a remote server from the downward API with a command like:

```
curl -X POST http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/register
```

- Wait for some time before starting the app container with a command like

```
sleep 60
```

- Clone a Git repository into a Volume
- Place values into a configuration file and run a template tool to dynamically generate a configuration file for the main app container. For example, place the `POD_IP` value in a configuration and generate the main app configuration file using Jinja.

Init containers in use

This example defines a simple Pod that has two init containers. The first waits for `myservice`, and the second waits for `mydb`. Once both init containers complete, the Pod runs the app container from its `spec` section.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup myservice.$(cat /var/run/secrets/kubernete
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup mydb.$(cat /var/run/secrets/kubernetes.io/
```

You can start this Pod by running:

```
kubectl apply -f myapp.yaml
```

The output is similar to this:

```
pod/myapp-pod created
```

And check on its status with:

```
kubectl get -f myapp.yaml
```

The output is similar to this:

```
NAME        READY     STATUS     RESTARTS   AGE
myapp-pod   0/1       Init:0/2   0          6m
```

or for more details:

```
kubectl describe -f myapp.yaml
```

The output is similar to this:

```
Name:          myapp-pod
Namespace:     default
[...]
Labels:        app=myapp
Status:        Pending
[...]
Init Containers:
  init-myservice:
[...]
    State:         Running
[...]
  init-mydb:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Containers:
  myapp-container:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Events:
  FirstSeen    LastSeen    Count    From                      SubObjectPath
  ---------    --------    -----    ----                      -------------
  16s          16s         1        {default-scheduler }
  16s          16s         1        {kubelet 172.17.4.201}    spec.initContainers{ir
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{ir
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{ir
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{ir
```

To see logs for the init containers in this Pod, run:

```
kubectl logs myapp-pod -c init-myservice # Inspect the first init container
kubectl logs myapp-pod -c init-mydb      # Inspect the second init container
```

At this point, those init containers will be waiting to discover Services named `mydb` and `myservice`.

Here's a configuration you can use to make those Services appear:

```
---
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
---
apiVersion: v1
kind: Service
metadata:
  name: mydb
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9377
```

To create the `mydb` and `myservice` services:

```
kubectl apply -f services.yaml
```

The output is similar to this:

```
service/myservice created
service/mydb created
```

You'll then see that those init containers complete, and that the `myapp-pod` Pod moves into the Running state:

```
kubectl get -f myapp.yaml
```

The output is similar to this:

```
NAME        READY     STATUS    RESTARTS   AGE
myapp-pod   1/1       Running   0          9m
```

This simple example should provide some inspiration for you to create your own init containers. What's next contains a link to a more detailed example.

Detailed behavior

During Pod startup, the kubelet delays running init containers until the networking and storage are ready. Then the kubelet runs the Pod's init containers in the order they appear in the Pod's spec.

Each init container must exit successfully before the next container starts. If a container fails to start due to the runtime or exits with failure, it is retried according to the Pod `restartPolicy` . However, if the Pod `restartPolicy` is set to Always, the init containers use `restartPolicy` OnFailure.

A Pod cannot be `Ready` until all init containers have succeeded. The ports on an init container are not aggregated under a Service. A Pod that is initializing is in the `Pending` state but should have a condition `Initialized` set to false.

If the Pod restarts, or is restarted, all init containers must execute again.

Changes to the init container spec are limited to the container image field. Altering an init container image field is equivalent to restarting the Pod.

Because init containers can be restarted, retried, or re-executed, init container code should be idempotent. In particular, code that writes to files on `EmptyDirs` should be prepared for the possibility that an output file already exists.

Init containers have all of the fields of an app container. However, Kubernetes prohibits `readinessProbe` from being used because init containers cannot define readiness distinct from completion. This is enforced during validation.

Use `activeDeadlineSeconds` on the Pod to prevent init containers from failing forever. The active deadline includes init containers. However it is recommended to use `activeDeadlineSeconds` if user deploy their application as a Job, because `activeDeadlineSeconds` has an effect even after initContainer finished. The Pod which is already running correctly would be killed by `activeDeadlineSeconds` if you set.

The name of each app and init container in a Pod must be unique; a validation error is thrown for any container sharing a name with another.

Resources

Given the ordering and execution for init containers, the following rules for resource usage apply:

- The highest of any particular resource request or limit defined on all init containers is the *effective init request/limit*. If any resource has no resource limit specified this is considered as the highest limit.
- The Pod's *effective request/limit* for a resource is the higher of:
  - the sum of all app containers request/limit for a resource
  - the effective init request/limit for a resource
- Scheduling is done based on effective requests/limits, which means init containers can reserve resources for initialization that are not used during the life of the Pod.
- The QoS (quality of service) tier of the Pod's *effective QoS tier* is the QoS tier for init containers and app containers alike.

Quota and limits are applied based on the effective Pod request and limit.

Pod level control groups (cgroups) are based on the effective Pod request and limit, the same as the scheduler.

Pod restart reasons

A Pod can restart, causing re-execution of init containers, for the following reasons:

- The Pod infrastructure container is restarted. This is uncommon and would have to be done by someone with root access to nodes.
- All containers in a Pod are terminated while `restartPolicy` is set to Always, forcing a restart, and the init container completion record has been lost due to garbage collection.

The Pod will not be restarted when the init container image is changed, or the init container completion record has been lost due to garbage collection. This applies for Kubernetes v1.20 and later. If you are using an earlier version of Kubernetes, consult the documentation for the version you are using.

What's next

- Read about creating a Pod that has an init container

- Learn how to debug init containers

3 - Pod Topology Spread Constraints

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

You can use *topology spread constraints* to control how Pods are spread across your cluster among failure-domains such as regions, zones, nodes, and other user-defined topology domains. This can help to achieve high availability as well as efficient resource utilization.

Note: In versions of Kubernetes before v1.18, you must enable the `EvenPodsSpread` feature gate on the API server and the scheduler in order to use Pod topology spread constraints.

Prerequisites

Node Labels

Topology spread constraints rely on node labels to identify the topology domain(s) that each Node is in. For example, a Node might have labels: `node=node1,zone=us-east-1a,region=us-east-1`

Suppose you have a 4-node cluster with the following labels:

```
NAME    STATUS   ROLES    AGE     VERSION   LABELS
node1   Ready    <none>   4m26s   v1.16.0   node=node1,zone=zoneA
node2   Ready    <none>   3m58s   v1.16.0   node=node2,zone=zoneA
node3   Ready    <none>   3m17s   v1.16.0   node=node3,zone=zoneB
node4   Ready    <none>   2m43s   v1.16.0   node=node4,zone=zoneB
```

Then the cluster is logically viewed as below:

graph TB subgraph "zoneB" n3(Node3) n4(Node4) end subgraph "zoneA" n1(Node1) n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4 k8s; class zoneA,zoneB cluster;

Instead of manually applying labels, you can also reuse the well-known labels that are created and populated automatically on most clusters.

Spread Constraints for Pods

API

The API field `pod.spec.topologySpreadConstraints` is defined as below:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  topologySpreadConstraints:
    - maxSkew: <integer>
      topologyKey: <string>
      whenUnsatisfiable: <string>
      labelSelector: <object>
```

You can define one or multiple `topologySpreadConstraint` to instruct the kube-scheduler how to place each incoming Pod in relation to the existing Pods across your cluster. The fields are:

- **maxSkew** describes the degree to which Pods may be unevenly distributed. It must be greater than zero. Its semantics differs according to the value of `whenUnsatisfiable` :
  - when `whenUnsatisfiable` equals to "DoNotSchedule", `maxSkew` is the maximum permitted difference between the number of matching pods in the target topology and the global minimum (the minimum number of pods that match the label selector in a topology domain. For example, if you have 3 zones with 0, 2 and 3 matching pods respectively, The global minimum is 0).
  - when `whenUnsatisfiable` equals to "ScheduleAnyway", scheduler gives higher precedence to topologies that would help reduce the skew.
- **topologyKey** is the key of node labels. If two Nodes are labelled with this key and have identical values for that label, the scheduler treats both Nodes as being in the same topology. The scheduler tries to place a balanced number of Pods into each topology domain.
- **whenUnsatisfiable** indicates how to deal with a Pod if it doesn't satisfy the spread constraint:
  - `DoNotSchedule` (default) tells the scheduler not to schedule it.
  - `ScheduleAnyway` tells the scheduler to still schedule it while prioritizing nodes that minimize the skew.
- **labelSelector** is used to find matching Pods. Pods that match this label selector are counted to determine the number of Pods in their corresponding topology domain. See Label Selectors for more details.

When a Pod defines more than one `topologySpreadConstraint` , those constraints are ANDed: The kube-scheduler looks for a node for the incoming Pod that satisfies all the constraints.

You can read more about this field by running `kubectl explain Pod.spec.topologySpreadConstraints` .

Example: One TopologySpreadConstraint

Suppose you have a 4-node cluster where 3 Pods labeled `foo:bar` are located in node1, node2 and node3 respectively:

```
graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) n4(Node4) end subgraph
"zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain
fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s
fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster
fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class
n1,n2,n3,n4,p1,p2,p3 k8s; class zoneA,zoneB cluster;
```

If we want an incoming Pod to be evenly spread with existing Pods across zones, the spec can be given as:

pods/topology-spread-constraints/one-constraint.yaml

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: mypod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - name: pause
    image: k8s.gcr.io/pause:3.1
```

`topologyKey: zone` implies the even distribution will only be applied to the nodes which have label pair "zone:<any value>" present. `whenUnsatisfiable: DoNotSchedule` tells the scheduler to let it stay pending if the incoming Pod can't satisfy the constraint.

If the scheduler placed this incoming Pod into "zoneA", the Pods distribution would become [3, 1], hence the actual skew is 2 (3 - 1) - which violates `maxSkew: 1`. In this example, the incoming Pod can only be placed onto "zoneB":

```
graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) p4(mypod) --> n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;
```

OR

```
graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) p4(mypod) --> n3 n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;
```

You can tweak the Pod spec to meet various kinds of requirements:

- Change `maxSkew` to a bigger value like "2" so that the incoming Pod can be placed onto "zoneA" as well.
- Change `topologyKey` to "node" so as to distribute the Pods evenly across nodes instead of zones. In the above example, if `maxSkew` remains "1", the incoming Pod can only be placed onto "node4".
- Change `whenUnsatisfiable: DoNotSchedule` to `whenUnsatisfiable: ScheduleAnyway` to ensure the incoming Pod to be always schedulable (suppose other scheduling APIs are satisfied). However, it's preferred to be placed onto the topology domain which has fewer matching Pods. (Be aware that this preferability is jointly normalized with other internal scheduling priorities like resource usage ratio, etc.)

Example: Multiple TopologySpreadConstraints

This builds upon the previous example. Suppose you have a 4-node cluster where 3 Pods labeled `foo:bar` are located in node1, node2 and node3 respectively:

```
graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;
```

You can use 2 TopologySpreadConstraints to control the Pods spreading on both zone and node:

pods/topology-spread-constraints/two-constraints.yaml

```yaml
kind: Pod
apiVersion: v1
```

```
metadata:
  name: mypod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  - maxSkew: 1
    topologyKey: node
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - name: pause
    image: k8s.gcr.io/pause:3.1
```

In this case, to match the first constraint, the incoming Pod can only be placed onto "zoneB"; while in terms of the second constraint, the incoming Pod can only be placed onto "node4". Then the results of 2 constraints are ANDed, so the only viable option is to place on "node4".

Multiple constraints can lead to conflicts. Suppose you have a 3-node cluster across 2 zones:

```
graph BT subgraph "zoneB" p4(Pod) --> n3(Node3) p5(Pod) --> n3 end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n1 p3(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3,p4,p5 k8s; class zoneA,zoneB cluster;
```

If you apply "two-constraints.yaml" to this cluster, you will notice "mypod" stays in `Pending` state. This is because: to satisfy the first constraint, "mypod" can only be put to "zoneB"; while in terms of the second constraint, "mypod" can only put to "node2". Then a joint result of "zoneB" and "node2" returns nothing.

To overcome this situation, you can either increase the `maxSkew` or modify one of the constraints to use `whenUnsatisfiable: ScheduleAnyway`.

Conventions

There are some implicit conventions worth noting here:

- Only the Pods holding the same namespace as the incoming Pod can be matching candidates.
- Nodes without `topologySpreadConstraints[*].topologyKey` present will be bypassed. It implies that:
    1. the Pods located on those nodes do not impact `maxSkew` calculation - in the above example, suppose "node1" does not have label "zone", then the 2 Pods will be disregarded, hence the incoming Pod will be scheduled into "zoneA".
    2. the incoming Pod has no chances to be scheduled onto this kind of nodes - in the above example, suppose a "node5" carrying label `{zone-typo: zoneC}` joins the cluster, it will be bypassed due to the absence of label key "zone".
- Be aware of what will happen if the incomingPod's `topologySpreadConstraints[*].labelSelector` doesn't match its own labels. In the above example, if we remove the incoming Pod's labels, it can still be placed onto "zoneB" since

the constraints are still satisfied. However, after the placement, the degree of imbalance of the cluster remains unchanged - it's still zoneA having 2 Pods which hold label {foo:bar}, and zoneB having 1 Pod which holds label {foo:bar}. So if this is not what you expect, we recommend the workload's `topologySpreadConstraints[*].labelSelector` to match its own labels.

- If the incoming Pod has `spec.nodeSelector` or `spec.affinity.nodeAffinity` defined, nodes not matching them will be bypassed.

  Suppose you have a 5-node cluster ranging from zoneA to zoneC:

  ```
  graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) n4(Node4) end
  subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end
  classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef
  k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster
  fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class
  n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;
  ```

  ```
  graph BT subgraph "zoneC" n5(Node5) end classDef plain
  fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s
  fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster
  fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n5 k8s; class
  zoneC cluster;
  ```

  and you know that "zoneC" must be excluded. In this case, you can compose the yaml as below, so that "mypod" will be placed onto "zoneB" instead of "zoneC". Similarly `spec.nodeSelector` is also respected.

  pods/topology-spread-constraints/one-constraint-with-nodeaffinity.yaml

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: mypod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: zone
            operator: NotIn
            values:
            - zoneC
  containers:
  - name: pause
    image: k8s.gcr.io/pause:3.1
```

Cluster-level default constraints

It is possible to set default topology spread constraints for a cluster. Default topology spread constraints are applied to a Pod if, and only if:

- It doesn't define any constraints in its `.spec.topologySpreadConstraints`.
- It belongs to a service, replication controller, replica set or stateful set.

Default constraints can be set as part of the `PodTopologySpread` plugin args in a scheduling profile. The constraints are specified with the same API above, except that `labelSelector` must be empty. The selectors are calculated from the services, replication controllers, replica sets or stateful sets that the Pod belongs to.

An example configuration might look like follows:

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration

profiles:
  - pluginConfig:
      - name: PodTopologySpread
        args:
          defaultConstraints:
            - maxSkew: 1
              topologyKey: topology.kubernetes.io/zone
              whenUnsatisfiable: ScheduleAnyway
          defaultingType: List
```

Note: The score produced by default scheduling constraints might conflict with the score produced by the `SelectorSpread` plugin. It is recommended that you disable this plugin in the scheduling profile when using default constraints for `PodTopologySpread`.

Internal default constraints

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

With the `DefaultPodTopologySpread` feature gate, enabled by default, the legacy `SelectorSpread` plugin is disabled. kube-scheduler uses the following default topology constraints for the `PodTopologySpread` plugin configuration:

```
defaultConstraints:
  - maxSkew: 3
    topologyKey: "kubernetes.io/hostname"
    whenUnsatisfiable: ScheduleAnyway
  - maxSkew: 5
    topologyKey: "topology.kubernetes.io/zone"
    whenUnsatisfiable: ScheduleAnyway
```

Also, the legacy `SelectorSpread` plugin, which provides an equivalent behavior, is disabled.

Note:
If your nodes are not expected to have **both** `kubernetes.io/hostname` and `topology.kubernetes.io/zone` labels set, define your own constraints instead of using the Kubernetes defaults.

The `PodTopologySpread` plugin does not score the nodes that don't have the topology keys specified in the spreading constraints.

If you don't want to use the default Pod spreading constraints for your cluster, you can disable those defaults by setting `defaultingType` to `List` and leaving empty `defaultConstraints` in the `PodTopologySpread` plugin configuration:

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration
```

```yaml
profiles:
  - pluginConfig:
      - name: PodTopologySpread
        args:
          defaultConstraints: []
          defaultingType: List
```

Comparison with PodAffinity/PodAntiAffinity

In Kubernetes, directives related to "Affinity" control how Pods are scheduled - more packed or more scattered.

- For `PodAffinity` , you can try to pack any number of Pods into qualifying topology domain(s)
- For `PodAntiAffinity` , only one Pod can be scheduled into a single topology domain.

For finer control, you can specify topology spread constraints to distribute Pods across different topology domains - to achieve either high availability or cost-saving. This can also help on rolling update workloads and scaling out replicas smoothly. See Motivation for more details.

Known Limitations

- There's no guarantee that the constraints remain satisfied when Pods are removed. For example, scaling down a Deployment may result in imbalanced Pods distribution. You can use Descheduler to rebalance the Pods distribution.
- Pods matched on tainted nodes are respected. See Issue 80921

What's next

- Blog: Introducing PodTopologySpread explains `maxSkew` in details, as well as bringing up some advanced usage examples.

4 - Disruptions

This guide is for application owners who want to build highly available applications, and thus need to understand what types of disruptions can happen to Pods.

It is also for cluster administrators who want to perform automated cluster actions, like upgrading and autoscaling clusters.

Voluntary and involuntary disruptions

Pods do not disappear until someone (a person or a controller) destroys them, or there is an unavoidable hardware or system software error.

We call these unavoidable cases *involuntary disruptions* to an application. Examples are:

- a hardware failure of the physical machine backing the node
- cluster administrator deletes VM (instance) by mistake
- cloud provider or hypervisor failure makes VM disappear
- a kernel panic
- the node disappears from the cluster due to cluster network partition
- eviction of a pod due to the node being out-of-resources.

Except for the out-of-resources condition, all these conditions should be familiar to most users; they are not specific to Kubernetes.

We call other cases *voluntary disruptions*. These include both actions initiated by the application owner and those initiated by a Cluster Administrator. Typical application owner actions include:

- deleting the deployment or other controller that manages the pod
- updating a deployment's pod template causing a restart
- directly deleting a pod (e.g. by accident)

Cluster administrator actions include:

- Draining a node for repair or upgrade.
- Draining a node from a cluster to scale the cluster down (learn about Cluster Autoscaling).
- Removing a pod from a node to permit something else to fit on that node.

These actions might be taken directly by the cluster administrator, or by automation run by the cluster administrator, or by your cluster hosting provider.

Ask your cluster administrator or consult your cloud provider or distribution documentation to determine if any sources of voluntary disruptions are enabled for your cluster. If none are enabled, you can skip creating Pod Disruption Budgets.

Caution: Not all voluntary disruptions are constrained by Pod Disruption Budgets. For example, deleting deployments or pods bypasses Pod Disruption Budgets.

Dealing with disruptions

Here are some ways to mitigate involuntary disruptions:

- Ensure your pod requests the resources it needs.
- Replicate your application if you need higher availability. (Learn about running replicated stateless and stateful applications.)
- For even higher availability when running replicated applications, spread applications across racks (using anti-affinity) or across zones (if using a multi-zone cluster.)

The frequency of voluntary disruptions varies. On a basic Kubernetes cluster, there are no automated voluntary disruptions (only user-triggered ones). However, your cluster administrator or hosting provider may run some additional services which cause voluntary disruptions. For example, rolling out node software updates can cause voluntary disruptions. Also, some implementations of cluster (node) autoscaling may cause voluntary disruptions to defragment and compact nodes. Your cluster administrator or hosting provider should have documented what level of voluntary disruptions, if any, to expect. Certain configuration options, such as using PriorityClasses in your pod spec can also cause voluntary (and involuntary) disruptions.

Pod disruption budgets

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

Kubernetes offers features to help you run highly available applications even when you introduce frequent voluntary disruptions.

As an application owner, you can create a PodDisruptionBudget (PDB) for each application. A PDB limits the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. For example, a quorum-based application would like to ensure that the number of replicas running is never brought below the number needed for a quorum. A web front end might want to ensure that the number of replicas serving load never falls below a certain percentage of the total.

Cluster managers and hosting providers should use tools which respect PodDisruptionBudgets by calling the Eviction API instead of directly deleting pods or deployments.

For example, the `kubectl drain` subcommand lets you mark a node as going out of service. When you run `kubectl drain`, the tool tries to evict all of the Pods on the Node you're taking out of service. The eviction request that `kubectl` submits on your behalf may be temporarily rejected, so the tool periodically retries all failed requests until all Pods on the target node are terminated, or until a configurable timeout is reached.

A PDB specifies the number of replicas that an application can tolerate having, relative to how many it is intended to have. For example, a Deployment which has a `.spec.replicas: 5` is supposed to have 5 pods at any given time. If its PDB allows for there to be 4 at a time, then the Eviction API will allow voluntary disruption of one (but not two) pods at a time.

The group of pods that comprise the application is specified using a label selector, the same as the one used by the application's controller (deployment, stateful-set, etc).

The "intended" number of pods is computed from the `.spec.replicas` of the workload resource that is managing those pods. The control plane discovers the owning workload resource by examining the `.metadata.ownerReferences` of the Pod.

PDBs cannot prevent involuntary disruptions from occurring, but they do count against the budget.

Pods which are deleted or unavailable due to a rolling upgrade to an application do count against the disruption budget, but workload resources (such as Deployment and StatefulSet) are not limited by PDBs when doing rolling upgrades. Instead, the handling of failures during application updates is configured in the spec for the specific workload resource.

When a pod is evicted using the eviction API, it is gracefully terminated, honoring the `terminationGracePeriodSeconds` setting in its PodSpec.

PodDisruptionBudget example

Consider a cluster with 3 nodes, `node-1` through `node-3`. The cluster is running several applications. One of them has 3 replicas initially called `pod-a`, `pod-b`, and `pod-c`. Another, unrelated pod without a PDB, called `pod-x`, is also shown. Initially, the pods are laid out as follows:

node-1	node-2	node-3
pod-a *available*	pod-b *available*	pod-c *available*

node-1	node-2	node-3
pod-x *available*		

All 3 pods are part of a deployment, and they collectively have a PDB which requires there be at least 2 of the 3 pods to be available at all times.

For example, assume the cluster administrator wants to reboot into a new kernel version to fix a bug in the kernel. The cluster administrator first tries to drain `node-1` using the `kubectl drain` command. That tool tries to evict `pod-a` and `pod-x` . This succeeds immediately. Both pods go into the `terminating` state at the same time. This puts the cluster in this state:

node-1 *draining*	node-2	node-3
pod-a *terminating*	pod-b *available*	pod-c *available*
pod-x *terminating*		

The deployment notices that one of the pods is terminating, so it creates a replacement called `pod-d` . Since `node-1` is cordoned, it lands on another node. Something has also created `pod-y` as a replacement for `pod-x` .

(Note: for a StatefulSet, `pod-a` , which would be called something like `pod-0` , would need to terminate completely before its replacement, which is also called `pod-0` but has a different UID, could be created. Otherwise, the example applies to a StatefulSet as well.)

Now the cluster is in this state:

node-1 *draining*	node-2	node-3
pod-a *terminating*	pod-b *available*	pod-c *available*
pod-x *terminating*	pod-d *starting*	pod-y

At some point, the pods terminate, and the cluster looks like this:

node-1 *drained*	node-2	node-3
	pod-b *available*	pod-c *available*
	pod-d *starting*	pod-y

At this point, if an impatient cluster administrator tries to drain `node-2` or `node-3` , the drain command will block, because there are only 2 available pods for the deployment, and its PDB requires at least 2. After some time passes, `pod-d` becomes available.

The cluster state now looks like this:

node-1 *drained*	node-2	node-3
	pod-b *available*	pod-c *available*
	pod-d *available*	pod-y

Now, the cluster administrator tries to drain `node-2` . The drain command will try to evict the two pods in some order, say `pod-b` first and then `pod-d` . It will succeed at evicting `pod-b` . But, when it tries to evict `pod-d` , it will be refused because that would leave only one pod available for the deployment.

The deployment creates a replacement for `pod-b` called `pod-e`. Because there are not enough resources in the cluster to schedule `pod-e` the drain will again block. The cluster may end up in this state:

node-1 *drained*	**node-2**	**node-3**	***no node***
	pod-b *terminating*	pod-c *available*	pod-e *pending*
	pod-d *available*	pod-y	

At this point, the cluster administrator needs to add a node back to the cluster to proceed with the upgrade.

You can see how Kubernetes varies the rate at which disruptions can happen, according to:

- how many replicas an application needs
- how long it takes to gracefully shutdown an instance
- how long it takes a new instance to start up
- the type of controller
- the cluster's resource capacity

Separating Cluster Owner and Application Owner Roles

Often, it is useful to think of the Cluster Manager and Application Owner as separate roles with limited knowledge of each other. This separation of responsibilities may make sense in these scenarios:

- when there are many application teams sharing a Kubernetes cluster, and there is natural specialization of roles
- when third-party tools or services are used to automate cluster management

Pod Disruption Budgets support this separation of roles by providing an interface between the roles.

If you do not have such a separation of responsibilities in your organization, you may not need to use Pod Disruption Budgets.

How to perform Disruptive Actions on your Cluster

If you are a Cluster Administrator, and you need to perform a disruptive action on all the nodes in your cluster, such as a node or system software upgrade, here are some options:

- Accept downtime during the upgrade.
- Failover to another complete replica cluster.
  - No downtime, but may be costly both for the duplicated nodes and for human effort to orchestrate the switchover.
- Write disruption tolerant applications and use PDBs.
  - No downtime.
  - Minimal resource duplication.
  - Allows more automation of cluster administration.
  - Writing disruption-tolerant applications is tricky, but the work to tolerate voluntary disruptions largely overlaps with work to support autoscaling and tolerating involuntary disruptions.

What's next

- Follow steps to protect your application by configuring a Pod Disruption Budget.
- Learn more about draining nodes
- Learn about updating a deployment including steps to maintain its availability during the rollout.

5 - Ephemeral Containers

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

This page provides an overview of ephemeral containers: a special type of container that runs temporarily in an existing Pod to accomplish user-initiated actions such as troubleshooting. You use ephemeral containers to inspect services rather than to build applications.

Warning: Ephemeral containers are in alpha state and are not suitable for production clusters. In accordance with the Kubernetes Deprecation Policy, this alpha feature could change significantly in the future or be removed entirely.

Understanding ephemeral containers

Pods are the fundamental building block of Kubernetes applications. Since Pods are intended to be disposable and replaceable, you cannot add a container to a Pod once it has been created. Instead, you usually delete and replace Pods in a controlled fashion using deployments.

Sometimes it's necessary to inspect the state of an existing Pod, however, for example to troubleshoot a hard-to-reproduce bug. In these cases you can run an ephemeral container in an existing Pod to inspect its state and run arbitrary commands.

What is an ephemeral container?

Ephemeral containers differ from other containers in that they lack guarantees for resources or execution, and they will never be automatically restarted, so they are not appropriate for building applications. Ephemeral containers are described using the same `ContainerSpec` as regular containers, but many fields are incompatible and disallowed for ephemeral containers.

- Ephemeral containers may not have ports, so fields such as `ports`, `livenessProbe`, `readinessProbe` are disallowed.
- Pod resource allocations are immutable, so setting `resources` is disallowed.
- For a complete list of allowed fields, see the EphemeralContainer reference documentation.

Ephemeral containers are created using a special `ephemeralcontainers` handler in the API rather than by adding them directly to `pod.spec`, so it's not possible to add an ephemeral container using `kubectl edit`.

Like regular containers, you may not change or remove an ephemeral container after you have added it to a Pod.

Uses for ephemeral containers

Ephemeral containers are useful for interactive troubleshooting when `kubectl exec` is insufficient because a container has crashed or a container image doesn't include debugging utilities.

In particular, distroless images enable you to deploy minimal container images that reduce attack surface and exposure to bugs and vulnerabilities. Since distroless images do not include a shell or any debugging utilities, it's difficult to troubleshoot distroless images using `kubectl exec` alone.

When using ephemeral containers, it's helpful to enable process namespace sharing so you can view processes in other containers.

What's next

- Learn how to debug pods using ephemeral containers.