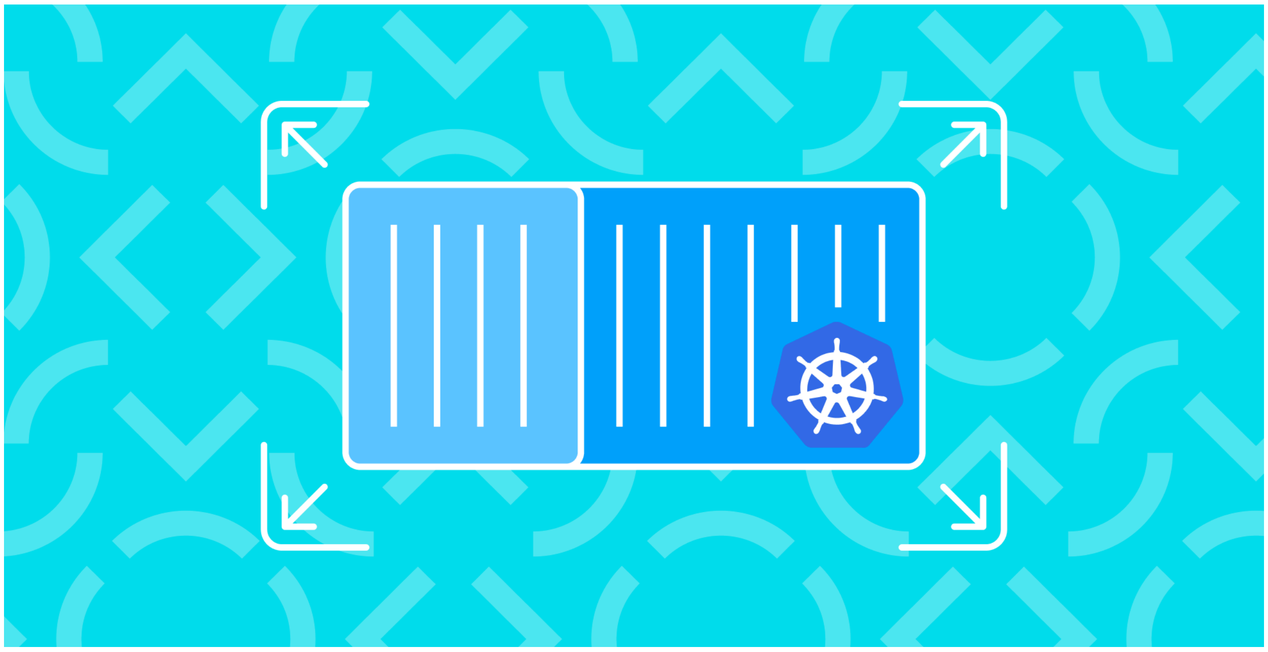
Kubernetes

In-place Pod resizing in Kubernetes: How it works and how to use it

12 November 2025
By Yury Losev, software engineer



You may well have been there yourself before: you carefully configured your Pods, setting the CPU and memory precisely, but then a sudden spike in traffic causes the application to start eating up a lot more memory than expected. This devolves into a bit of a gamble with the VPA: will it manage to pick the appropriate new resource values, since the `Recreate` mode sometimes tends to bring on unpleasant surprises? For instance, there's the possibility that it will recreate a Pod during a traffic surge, which under a load could overwhelm other components and lead to a cascading failure. Alternatively, you could just allocate spare resources upfront

In this article, I will address the issue of resource shortages, which can sometimes occur in containers. A solution to this issue was first introduced in Kubernetes 1.27: the in-place Pod resource update enables you to change the compute resources of running Pods without needing to restart them. In version 1.33, it was greatly enhanced and made generally available (that is, enabled by default!).

Let's dive into how it works, where it particularly comes in handy in real-world scenarios, and what limitations still exist.

Changing Pod's requests and limits without restarting it

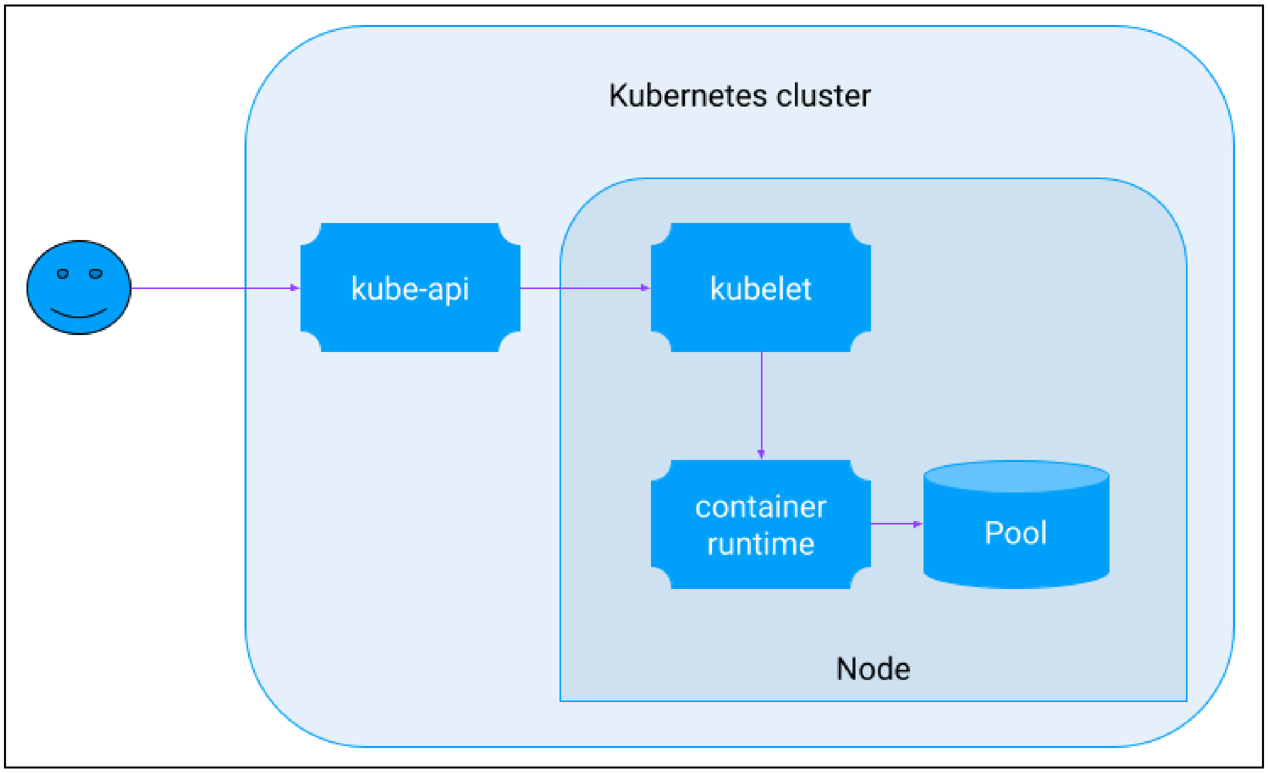
Thanks to KEP-1287, you can now change the `resources.requests` and `resources.limits` fields in a Pod's spec — which used to be immutable — without having to restart the Pod. This means Pod can now be resized "on the fly" or "in place".

How does it work? Here is a step-by-step breakdown of the process:

1. You modify the container's resources (CPU or memory) in the Pod manifest. The changes are then pushed to the Kubernetes API (kube-api).
2. The kube-api passes the updated Pod spec to the node.
3. On the node, kubelet quickly checks if there are enough resources. It means the sum of all node resources available for containers must be greater than or equal to the resources requested.

   If this condition is true, the changes are applied.

   If not, the Pod is marked as `PodResizePending`.
4. Following the check, kubelet uses the Container Runtime Interface (CRI) to inform the container runtime (such as containerd, CRI-O, or Podman): "This container needs more/less memory."
5. The container runtime automatically adjusts the container's cgroups, increasing or decreasing the allocated resources without restarting the container. This process is asynchronous and non-blocking, allowing kubelet to continue on with other tasks.



Changing resources in the Pod specification

When in-place resizing comes in handy

Let's take a look at some real-world scenarios where the new feature is a perfect fit:

1. **Databases**. Imagine a PostgreSQL instance suddenly requiring more memory to process a new report. Previously, you had no choice but to restart the Pod, causing downtime and dropped connections. Now, you can increase resources on the fly without suffering any interruption, and your users won't notice a thing.
2. **Stateless web applications**. Go and Python applications can utilize extra CPU and memory with ease, without requiring a restart. This is a perfect dynamic scaling situation for handling abrupt traffic spikes. However, for Node.js applications, it's not enough simply to increase the Pod's memory limits, as changing the `max-old-space-size` parameter is required. Similarly, for JVM applications, the `-Xmx` parameter (maximum heap size) is typically set at startup. While you can adjust CPU and non-heap memory on the fly, a restart is still required for the Java application to reap the full benefits of the new memory limits. Unfortunately, an ideal solution has not yet been crafted.
3. **ML services**. TensorFlow-based services that suddenly require resources for larger models or higher-volume requests can now be bolstered by a resource boost without having to restart.

4. **Sidecar proxies in service meshes**. You can tweak Envoy proxies in Istio and other service meshes to handle traffic spikes without dealing with glitches or messing with the main app. This is a cool feature to have, especially considering how unpredictable traffic can be.

Sounds good? Enough with the theory; let's move ahead to the practical application and see how it works.

Using in-place Pod resizing

There's a bit of a difference between Kubernetes 1.27–1.32 and the latest K8s versions (1.33+). To see what's going on and why, let's start by experimenting with version 1.27.

Updating Pod resources in Kubernetes v1.27

Kubernetes 1.27 introduced the `InPlacePodVerticalScaling` feature gate, which lets you modify the `.spec.containers[*].resources` field for Pods. However, in order for this feature to work, you must enable it not only for the control plane components (apiserver, controller-manager, scheduler) but also for the kubelet. Once the `InPlacePodVerticalScaling` gate is enabled, the kubelet will start exposing stats for allocated resources in the container's status. E.g.:

```
status:
 containerStatuses:
 - allocatedResources:
     cpu: 150m
     memory: 128Mi
   containerID: containerd://foobar
   image: docker.io/library/ubuntu:22.04
   imageID: docker.io/library/ubuntu@...
   name: pod-watcher
   ready: true
   resources:
     limits:
       cpu: 150m
       memory: 128Mi
     requests:
       cpu: 150m
       memory: 128Mi
   restartCount: 0
```

This is where things start to get interesting. After specifying these flags, you can change not only the image, tolerations, and `terminationGracePeriodSeconds` fields of a Pod (as is normally the case), but you can also **alter the `resources` field**:

```
The Pod "resize-pod" is invalid: spec: Forbidden: pod updates may not change
fields other than
`spec.containers[*].image`,`spec.initContainers[*].image`,`spec.activeDeadli
neSeconds`,`spec.tolerations` (only additions to existing
tolerations),`spec.terminationGracePeriodSeconds` (allow it to be set to 1
if it was previously negative),`spec.containers[*].resources` (for
CPU/memory only)
```

Let’s deploy the following Pod to see if the new mechanism works:

```
apiVersion: v1
kind: Pod
metadata:
 name: resize-pod
spec:
 containers:
 - name: pod-watcher
   image: ubuntu:22.04
   command:
   - "/bin/bash"
   - "-c"
   - |
     apt-get update && apt-get install -y procps bc
     echo "--- Pod started: $(date)"

     # Get current container resource limits
     get_cpu_limit() {
       if [ -f /sys/fs/cgroup/cpu.max ]; then
```

```
    # cgroup v2
    local cpu_data=$(cat /sys/fs/cgroup/cpu.max)
    local quota=$(echo $cpu_data | awk '{print $1}')
    local period=$(echo $cpu_data | awk '{print $2}')

    if [ "$quota" = "max" ]; then
      echo "unlimited"
    else
      echo "$(echo "scale=3; $quota / $period" | bc) cores"
    fi
  else
    # cgroup v1
    local quota=$(cat /sys/fs/cgroup/cpu/cpu.cfs_quota_us)
    local period=$(cat /sys/fs/cgroup/cpu/cpu.cfs_period_us)

    if [ "$quota" = "-1" ]; then
      echo "unlimited"
    else
      echo "$(echo "scale=3; $quota / $period" | bc) cores"
    fi
  fi
}

get_memory_limit() {
  if [ -f /sys/fs/cgroup/memory.max ]; then
    # cgroup v2
    local mem=$(cat /sys/fs/cgroup/memory.max)
    if [ "$mem" = "max" ]; then
      echo "unlimited"
    else
      echo "$((mem / 1024 / 1024)) MiB"
    fi
  else
    # cgroup v1
    local mem=$(cat /sys/fs/cgroup/memory/memory.limit_in_bytes)
    echo "$((mem / 1024 / 1024)) MiB"
  fi
```

```
        }

        # Repeat printing the resource info every 5 sec
        while true; do
          echo "----- Resource check: $(date)"
          echo "CPU limit: $(get_cpu_limit)"
          echo "Memory limit: $(get_memory_limit)"
          echo "Available memory: $(free -h | grep Mem | awk '{print $7}')"
          sleep 5
        done
      resizePolicy:
      - resourceName: cpu
        restartPolicy: NotRequired
      - resourceName: memory
        restartPolicy: NotRequired
      resources:
        requests:
          memory: "128Mi"
          cpu: "100m"
        limits:
          memory: "128Mi"
          cpu: "100m"
```

Note the `resources` section of this manifest: we're creating a Pod that has its requests and limits set.

Now, let's try to modify the Pod's resources:

```
kubectl patch pod resize-pod --patch \
 '{"spec":{"containers":[{"name":"pod-watcher", "resources":{"requests":
{"cpu":"300m"}, "limits":{"cpu":"300m"}}}]}}'

pod/resize-pod patched
```

Our command went through just fine, and the Pod's resources got updated. Looks like everything is working, right? Well, not so fast – the following command will faidddddd:

```
kubectl patch pod resize-pod --patch \
 '{"spec":{"containers":[{"name":"pod-watcher", "resources":{"requests":
{"cpu":"250m"}, "limits":{"cpu":"300m"}}}]}}'

The Pod "resize-pod" is invalid: metadata: Invalid value: "Guaranteed": Pod
QoS is immutable
```

So why is that? Well, when a Pod is updated, Kubernetes automatically fills some of its fields. Since the initial requests and limits were identical, the Pod's QoS class was set to `Guaranteed`. After we made the edit, they were different, so the QoS needs to switch to `Burstable`. The problem is that **the QoS class field is immutable**, so you can't change it.

However, by taking some preparatory steps — such as defining the usage profile and ensuring the necessary runtime guarantees — you can modify the container's resources. For example:

```
resources:
  requests:
    cpu: 100m
    memory: 128Mi
  limits:
    cpu: 300m
    memory: 256Mi
```

Let's say we have our resources set, but it turns out they don't cover the app's actual needs, so we decide to scale everything up proportionally:

```
kubectl patch pod resize-pod --patch \
 '{"spec":{"containers":[{"name":"pod-watcher", "resources":{"requests":
{"cpu":"200m", "memory": "256Mi"}, "limits":{"cpu":"600m", "memory":
"1Gi"}}}]}}'

pod/resize-pod patched
```

… or we can just raise the limits to avoid getting OOM-killed:

```
kubectl patch pod resize-pod \
 --type='json' \
 -p='[
   {
     "op": "add",
     "path": "/spec/containers/0/resources/limits/memory",
     "value": "1Gi"
   },
   {
     "op": "add",
     "path": "/spec/containers/0/resources/limits/cpu",
     "value": "1000m"
   }
 ]'

pod/resize-pod patched
```

Sometimes, **JSON patch** is just easier, especially when you don't feel like typing the container name. Thus, here is what we ended up with:

```
kubectl get pods resize-pod -o yaml

 containerStatuses:
 - allocatedResources:
     cpu: 100m
     memory: 128Mi
   containerID: containerd://foobar
   image: docker.io/library/ubuntu:22.04
   imageID: docker.io/library/ubuntu@...
   name: pod-watcher
   ready: true
   resources:
     limits:
       cpu: 1000m
       memory: 1Gi
     requests:
       cpu: 100m
       memory: 128Mi
   restartCount: 0
```

The Pod **resize happens asynchronously**, which means the new values don't come into effect right away. Kubernetes 1.27 lacks a specific field to track the status of this operation. So, you can either run the command again in about 30 seconds or use watch mode:

```
kubectl get pods resize-pod -o yaml -w
```

As you can see, the container's resources have been successfully changed while the `restartCount` remains at 0, meaning there were no container restarts.

You can also see that the Pod's status has changed from Kubernetes' point of view. The key here is **to see if CRI has actually altered the Pod's cgroups.** To check if that indeed occurred, we ran a Pod that logs its current cgroup values. Let's take a look at them:

```
kubectl logs resize-pod --tail=4
----- Resource check: Tue Oct 22 13:21:10 UTC 2025
CPU limit: 1.000 cores
Memory limit: 1 GiB
Available memory: 3.9Gi
```

You may need to check the whole log history to see when the change took place:

```
kubectl logs resize-demo --tail=-1
...
----- Resource check: Tue Oct 22 13:21:05 UTC 2025
CPU limit: .100 cores
Memory limit: 128 MiB
Available memory: 3.9Gi
----- Resource check: Tue Oct 22 13:21:10 UTC 2025
CPU limit: 1.000 cores
```

```
Memory limit: 1 GiB
Available memory: 3.9Gi
```

Yep, it looks like containerd has modified resources, and our container is running with the new limits.

Resize policy

You may have noticed the following section in the specification:

```
resizePolicy:
  - resourceName: cpu
    restartPolicy: NotRequired
  - resourceName: memory
    restartPolicy: NotRequired
```

This new field pops up when the `InPlacePodVerticalScaling` feature flag is enabled. It specifies how a container should behave when its resources are resized. By default, the `restartPolicy` is set to `NotRequired`, so the container does not need to be restarted when its resource limits are changed.

However, some applications require a restart when their limits are updated, such as a JVM with the `-Xmx` option. For such containers, you can set `restartPolicy: RestartContainer`. This presents you with a means to flexibly configure your applications. For example, a sidecar running an Envoy proxy can be updated without having to restart, whereas a “heavy” Java application can be configured to restart at all times, if necessary.

For Pods like these, you’ll see the number of restarts when you list them:

```
kubectl get pods resize-pod
NAME          READY   STATUS    RESTARTS      AGE
resize-pod    1/1     Running   1 (32s ago)   3m56s
```

It’ll also be reflected in the container status:

```
    resources:
      limits:
        cpu: 200m
        memory: 512Mi
      requests:
        cpu: 100m
        memory: 128Mi
    restartCount: 1
```

Updating Pod resources in Kubernetes v1.33+

Practising Pod resizing in newer Kubernetes versions will be almost the same. However, you need to consider some important changes introduced to this feature in v1.33:

1. The `InPlacePodVerticalScaling` feature flag is now enabled by default.
2. A new `/resize` subresource has been added (available starting with kubectl 1.32), which allows you to change Pod resources without messing up the other fields in its spec.
3. The Pod's status now includes new states to show what's happening with the resize:

   `type: PodResizePending` — kubelet can't satisfy the resource change request right away. You can check the message field for additional details and reasons:

   - `reason: Infeasible` — the requested change isn't possible on the current node (e. g., more resources have been requested than are available);
   - `reason: Deferred` — the change can't be implemented now, but it might be possible later (for example, after another Pod is removed). The kubelet will keep retrying the resize.

   `type: PodResizeInProgress` — kubelet has accepted the request and assigned the resources, but the changes are still being applied. This process usually does not take long but can sometimes drag on, depending on the resource type and the runtime behavior. If anything goes wrong, you'll see it in the message field (with `reason: Error`).

To make things more convenient for us as Kubernetes operators, resource resizing is now free from the strict limitations it used to have, like those tied to QoS classes or `resizePolicy` errors.

Changes for the in-place updates in Kubernetes 1.34

Finally, the most recent K8s v1.34, released in August 2025, brought even more updates to the feature. They are the following:

1. Now, the in-place Pod update mechanism supports decreasing memory limits with a `NotRequired` resize restart policy. Before resizing the limit, kubelet will perform a best-effort check: if the current memory consumption exceeds the new limits, the resize will be blocked and the status will remain *"In Progress"*.
2. A separate `InPlacePodVerticalScalingExclusiveMemory` (default is `false`) feature gate was introduced to enable/disable support for the static memory policy. It will regulate memory resizing for Pods with `Guaranteed` QoS.
3. kubelet has been expanded with the following metrics:

   `kubelet_container_requested_resizes_total` — the total number of resize attempts. It is counted at the container level. Updating a Pod with changes to multiple containers at once will be counted as multiple resize attempts.

   `kubelet_pod_resize_duration_seconds` — the duration of the `doPodResizeAction` function responsible for performing the resize.

   `kubelet_pod_infeasible_resizes_total` — the total number of resizes that were rejected by the kubelet as infeasible.

   `kubelet_pod_pending_resizes` — the current number of Pods in the `Pending` state.

   `kubelet_pod_in_progress_resizes` — the total number of resize requests that kubelet marks as in progress. This means that resources for them have been allocated but not yet activated.

   `kubelet_pod_pod_deferred_resize_accepted_total` — the total number of resize requests that kubelet first marked as deferred but later accepted for execution.
4. The logic implementing retries of deferred resizes has been optimized. In the scenario that several deferred resizes are activated at once, retries are performed in the following order:

Retries for the Pods with a higher Priority (determined based on `PriorityClass`) are attempted first.

For Pods with the same `Priority` level, `Guaranteed` Pods are resized first, followed by the `Burstable` ones.

If all other parameters are the same, Pods that have been in the `Deferred` state for longer will be resized first.

Still, some limitations persist.

In-place Pod resizing limitations

At the time of writing this article, they are:

- If a Pod is using swap, you can't change its memory on the fly — the container needs to be restarted.
- It doesn't work on Windows nodes.
- You can only change the CPU and memory.
- The Pod's QoS class does not change. Regardless of how you modify the parameters, the QoS class will remain the same as it was at startup. That does not block resource updates, but changing the QoS class is not yet possible.
- You cannot restart init and ephemeral containers.
- You cannot remove requests or limits completely. You can only change their values.
- The `resizePolicy` field can't be changed once the Pod has been created.

A note on VPA. Starting with Kubernetes v1.33, Vertical Pod Autoscaler (VPA) version 1.4+ is supported with the new `mode: InPlaceOrRecreate`. This mode allows for applying resource recommendations on the fly. It uses the `resize` subresource, which is why you cannot enable it in older versions. You could probably patch them, but VPA still hits the same limitations mentioned earlier and can't correctly update the Pods.

Conclusion

Updating container resources for Kubernetes Pods on the fly is a giant leap forward for rendering K8s clusters more flexible and reliable. Now, you can react to load changes promptly and stop worrying about downtime or cascading failures from Pods being recreated. Sure, the technology still has its limits, and it doesn't cover every scenario perfectly. But the fact that live Pod resizing is now enabled by default shows how quickly things are evolving.

P.S. According to the GitHub issue, KEP-1287 is planned to become stable in Kubernetes v1.35, which is scheduled for 17th December 2025. However, depending on how the development progresses and on the user's feedback, this graduation might be postponed. Thus, if you're already using the feature, feel free to share your experience.

Kubernetes snapshots: What are they and how to use them?

Kubernetes storage backups

19 November 2021

Running MongoDB in Kubernetes: An overview of existing solutions

Kubernetes databases Kubernetes operators

1 April 2022

Dhall configuration language as another way to write manifests for Kubernetes

Kubernetes YAML

9 REVIEWS

Back up