Dynamic Containers Proposal [DRAFT]

[PUBLIC]
Dynamic Containers

Tim Allclair · Apr 30, 2026

THIS IS OUT OF DATE
Please refere to the KEP for the latest:
https://github.com/kubernetes/enhancements/pull/6169

Overview

Dynamic Containers (a.k.a. Dynamic Pods) is a feature that lets *main* containers be added and removed from a pod while it's running.

Upstream issue: https://github.com/kubernetes/enhancements/issues/5972

Note: An earlier version of this proposal included a "non-terminal pod" concept, but I'm extracting that to a separate proposal, to be (or not be) pursued as a separate KEP. See Non-Terminal Pods Proposal [DRAFT].

Goals

- Enable high-churn addition and removal of containers
- Enable low-latency workload startup, targeting < 100ms of Kubelet overhead (excluding latency of CRI operations)

Use Cases

Hierarchical Scheduling

Dynamic Pods can be used as part of a **two-tier scheduling architecture** that allows Kubernetes to act as the macro-scheduler while frameworks handle micro-scheduling:

- **L1 (Kubernetes):** Orchestrates global placement (e.g., PodGroups) and enforces a secure "Resource Envelope".
- **L2 (Framework CP):** Specialized CPs perform fine-grained L2 scheduling *inside* that envelope—partitioning CPU/Memory and managing sub-cgroups in real-time.

See [External] Enhancing Kubernetes for Dynamic Batch Workloads for more details.

In particular, this enables use cases where the L2 control plane can start workloads with much lower latency than it takes to create, schedule and start a new pod.

TBD: Enabling Pod Restore

See [PUBLIC] Proposal: Pod Restore MVP

Proposal

Limitations

To scope down the proposal for the initial release, the following limitations are put in place. These are not inherent requirements for the feature, and can be reevaluated as new additive features in the future.

- Only main containers can be added or removed (not init containers)
- The pod must be in an initialized state (i.e. all init containers have completed) before any dynamic container changes can be made.
- This proposal does not allow for general container mutation. A corollary is that a container with the same name can only be added after the previous container with that name is *completely* removed (see discussion of termination below)
- Container resources cannot change the QOS class of the pod (in other words, containers added to a best-effort pod cannot specify resource requirements).
- Containers can only mount volumes or name ResourceClaims already mounted/claimed by the pod.
- Resize limitations apply: Windows is not supported, swap enabled pods are not supported
- Because container resources are treated as a resize, added or removed Containers can only request resources that are supported by in-place resize.
- A pod must have at least one main container. See Non-Terminal Pods Proposal [DRAFT] for an option to relax this constraint.

- Privileged containers cannot be added.

API

Containers are added and removed via an update on the pod resource.

- Multiple containers can be added and removed in a single request.
- Standard pod validation applies to the updated pod.
- Additional validation applied to cover the limitations listed above.
    - The container name must be unique among all containers AND all container statuses
- `.spec.containers` must be non-empty (i.e. cannot remove the last container)
- Container changes cannot be made once a DeletionTimestamp is set on the pod

Container Status

Containers that have been added but not allocated will generate a ContainerStatus with the `Waiting` state, and the reason `Unallocated`.

Containers that have been removed from the allocation will remain in the `Running` state until terminated. After termination, the container status will remain until garbage collected (see below).

Allocation

Newly added containers go through an allocation step that works exactly the same as in-place pod resize. In other words, adding a new container with additional resources is considered a pod resize, and can put the pod into a deferred (or even infeasible) state.

Allocation is an atomic operation, so if multiple containers are added and removed together, the changes will only be allocated if ALL of the changes can be allocated.

Allocation will save (and checkpoint) the full container spec for any allocated containers (see Image update allocation for an additional implicitation of this change).

Once allocated, the container changes will be made during the next pod sync. `UpdatePodFromAllocation` will modify the pod spec to reflect the allocated containers. Adding containers is straightforward: the Kubelet sees that the new container is not running and adds it. Removal is more complicated.

Allocation will be halted once a pod enters a terminating state.

Container Termination

When a running container is removed from a pod (and the changes are allocated), the Kubelet must terminate the container. `computePodActions` will be updated to identify containers that are running but not part of the PodSpec and queue them up for a `killContainer` action.

`kuberuntime_manager.killContainer` already supports reading the required container information from the container runtime annotations, so we do not need to store the container spec anywhere else. The operation also respects the grace period.

Once a removed container is terminated, its ContainerStatus will be kept in the pod status. The Kubelet will preserve removed container statuses in the API when it updates the pod status. Garbage collection will be applied to removed container statuses.

1. Keep up to N (maybe 10?) removed container statuses: After more than N removed container statuses have accrued, delete the oldest status.
2. Enhance `containerGC` to clear the container status when the actual container object is garbage collected.

Logs from removed containers will be managed by the existing container garbage collection.

In-Place Pod Resize Changes

This proposal makes a few changes to how in-place pod resize works.

Resize via pod update

We will now allow resizing a pod through update of the main `pods` resource. The `resize` subresource will still exist so that permission to resize *without* permission to modify running code can be granted, but resources will now be mutable through edit/update of the `pods` resource. The same validation rules apply.

Image update allocation

Today, image update and resource allocation are totally separate. This means that if I resize a container, and the resize is deferred, and I also change the container image, the Kubelet will restart the container with the new image even while the resize is still deferred. This violates the atomic allocation principle.

Starting with the enablement of the `DynamicContainers` feature, container image update will also be gated by the atomic allocation step. This means that in the above scenario, the Kubelet would *not* restart the container with the new image until the resize (and any other changes) can be allocated.

Design Details

Security

- TODO: admission considerations

Privileged containers cannot be added.

Implementation Details

Probe Manager

- Currently: sets up container probes when the pod is added
- Proposed: Add/start container probes when starting a container, and remove/stop probes when a container is terminated.
- Note: adding a new container with a readiness probe will cause the pod to become unready, until that container is ready.

Topology Managers

Depends on `InPlacePodVerticalScalingExclusiveCPUs` / `InPlacePodVerticalScalingExclusiveMemory`. Passing the allocation step involves running the pod through admission again, which is where the CPU & memory allocation is managed. No updates should be needed for container additions. Whether changes are needed for remove depends on how resource downsizing is implemented for the above features.

Allocation Manager Admission handling

- (discussed above) Treat new containers as an allocation step, similar to resize

Admission Handlers

Need to handle updates to the pod.

- PodResizeValidator
- ~~Quota~~
- LimitRanger
- ~~NodeDeclaredFeatures~~

Risks

Third-Party Controllers that assume containers are static

TODO

Container Status update API load

TODO: high churn of containers will lead to high-churn of pod status updates (adding a new short-lived container will generate a minimum of 2-3 pod status updates).

Changes to Resize behavior

TODO

Alternatives Considered

Optimized Pods

The primary problem this proposal aims to address is running a (potentially very short lived) workload with minimal latency and overhead. Can we just optimize pod startup and overhead instead?

Even if we can optimize pod startup to within our target, this is still not accounting for potential workload initialization costs. In other words, with Dynamic Containers I can create a pod that runs several init containers to prepare for the workload, such as populating a volume, prepulling a model, setting up a database sidecar, or performing a credential exchange. No matter how fast we get minimal pod startup, we cannot optimize away these factors. Solving for this additional initialization work would require complicated cross-pod coordination.

In local tests I can start a minimal (pause image, no volumes / AutomountServiceAccountToken=false) pod in about 800ms,

Ephemeral Containers

Mutable Container Images

Non-Terminal Pods Proposal [DRAFT]

[PUBLIC]
Non-Terminal Pods

Overview

Non-Terminal Pods proposes a new "non-terminal" pod API option (`NonTerminating bool` on the PodSpec), which keeps the pod alive and active even if it does not have any running containers. This proposal builds on Dynamic Containers to enable place-holder pods that are ready to dynamically added containers, but are not actively running any workloads.

Proposal

Non-terminal API

TODO: Extract the non-terminal proposal to a separate KEP.

This section is a placeholder. The exact API is TBD.

A new `NonTerminal bool` is added to the pod spec. If a pod is non-terminal, then the pod is not marked as `Failure` or `Success` even if all of the containers in the pod have terminated. Additionally, it is allowed for a non-terminal pod to have NO containers, even on creation.

The state of a non-terminal pod with no running containers is:

- Not ready
- Initialized
- PodPhase = Running

Once termination is triggered by something outside of the container lifecycle, such as setting the pod deletion timestamp, the Kubelet will immediately start ignoring the non-terminal bit. This means that a non-terminal pod can transition to a success state if all containers exited successfully.

Alternatives Considered

Pause Container

This feature can be emulated by simply including a pause container in the pod spec, and leaving it running. As long as the pause container is not removed, the pod will remain alive.

Advantages of Pause approach:

- No changes to pod lifecycle:
    - No need for a new pod phase
    - No risk to external controllers that assume `Containers` can't be empty

Disadvantages:

- Continuous (but minimal) overhead of running a pause container

Authz & Admission Considerations

Dynamic Pods
Authz & Admission Considerations

Goals

1. Existing users (non super-privileged) don't have permissions to make dynamic pod updates.
2. Existing admission webhooks enforce policies on dynamically added containers.
3. Existing VAPs & MAPs enforce policies on dynamically added containers.
4. [TBD] Make dynamic mutability (or immutability) a fixed creation-time declaration.

Current Proposal

https://github.com/kubernetes/enhancements/pull/6169

- Dynamic Pod updates are allowed through a new subresource, `/dynamic` (name is open to suggestions).
- New containers are not allowed to escalate security context permissions beyond what is already allowed.

Issues

- Existing admission enforcement mechanisms need to be updated to intercept the new resource.
- Security context escalation treats each permission as an independent variable.

Brainstorming

- Trigger a dry-run create request for the pod.
    - Issues with false positives on rejections, e.g. with quota-like controllers
- Also send an UPDATE admission request on the pod's root resource to admission controls that don't intercept the new subresource
- Allow dynamic mutations through the pod resource, but require an additional permission for dynamic mutations (no new subresource).
- Forbid dynamic pod updates if there are any pod-intercepting admission controls that don't handle the new endpoint (or explicitly declare that they don't need to).
    - Scoped to webhooks that would intercept the pod create or update (in other words, ignore webhooks that filter out the namespace where the dynamic update is happening)