

 ace2cc6431 ▾

...

[enhancements](#) / [keys](#) / [sig-node](#) / [753-sidecar-containers](#) / [README.md](#)




SergeyKanzhelev applied the feedback

 **History**

 2 contributors



 1541 lines (1189 sloc) | 70.4 KB

...

KEP-753: Sidecar containers

- [Release Signoff Checklist](#)
- [Summary](#)
- [Motivation](#)
 - [Problems: jobs with sidecar containers](#)
 - [Problems: log forwarding and metrics sidecar](#)
 - [Problems: service mesh](#)
 - [Problems: configuration / secrets](#)
 - [Goals](#)
 - [Non-Goals](#)
- [Proposal](#)
 - [Naming](#)
 - [Collection name](#)
 - [Reuse of restartPolicy field and enum](#)
 - [Use Always vs. New enum value](#)
 - [Risks and Mitigations](#)
 - [Scenario 1. User decides to use sidecars as a way to run regular containers](#)
 - [Scenario 1.a](#)
 - [Scenario 2. Balloon sidecars](#)
 - [Scenario 3. Long initialization tasks running in-parallel](#)
 - [Scenario 4. Sidecar that never becomes ready](#)

- Scenario 5. Intentional failing or terminating sidecars
 - Scenario 6. Keeping a sidecar alive to keep consuming cycles on termination
 - Scenario 7. Risk of porting existing sidecars to the new mechanism naively
- Design Details
 - Backward compatibility
 - Resources calculation for scheduling and pod admission
 - Exposing Pod Resource requirements
 - Goals of exposing the Pod.TotalResourcesRequested field
 - Implementation details
 - Notes for implementation
 - Resources calculation and Pod QoS evaluation
 - Topology and CPU managers
 - Termination of containers
 - Other
 - Test Plan
 - Prerequisite testing updates
 - Unit tests
 - Integration tests
 - e2e tests
 - Ready state of a sidecar container is properly used to create/delete endpoints
 - Pod lifecycle scenarios without sidecar containers
 - Pod lifecycle scenarios with sidecar containers
 - Kubelet restart test cases
 - API server is down: failure to update containers status during initialization
 - Resource usage testing
 - Upgrade/downgrade testing
 - Graduation Criteria
 - Alpha
 - Beta
 - Upgrade / Downgrade Strategy
 - Upgrade strategy
 - Downgrade strategy
 - Version Skew Strategy
- Production Readiness Review Questionnaire
 - Feature Enablement and Rollback
 - Rollout, Upgrade and Rollback Planning

- [Monitoring Requirements](#)
- [Dependencies](#)
- [Scalability](#)
- [Troubleshooting](#)
- [Implementation History](#)
- [Drawbacks](#)
- [Future use of restartPolicy field](#)
- [Alternatives](#)
 - [Pod startup completed condition](#)
 - [Readiness probes](#)
 - [Startup probes](#)
 - [postStart hook](#)
 - [Alternative 1. Sidecar containers as a separate collection](#)
 - [Alternative 2. DependOn semantic between containers](#)
 - [Alternative 3. Phases](#)
 - [Alternative 4. TerminatePod on container completion](#)
 - [Alternative 5. Injection of sidecar containers thru the "external" object](#)
- [Infrastructure Needed \(Optional\)](#)

Release Signoff Checklist

Items marked with (R) are required *prior to targeting to a milestone / release*.

- ☐ (R) Enhancement issue in release milestone, which links to KEP dir in [kubernetes/enhancements](#) (not the initial KEP PR)
- ☐ (R) KEP approvers have approved the KEP status as `implementable`
- ☐ (R) Design details are appropriately documented
- ☐ (R) Test plan is in place, giving consideration to SIG Architecture and SIG Testing input (including test refactors)
 - ☐ e2e Tests for all Beta API Operations (endpoints)
 - ☐ (R) Ensure GA e2e tests meet requirements for [Conformance Tests](#)
 - ☐ (R) Minimum Two Week Window for GA e2e tests to prove flake free
- ☐ (R) Graduation criteria is in place
 - ☐ (R) [all GA Endpoints](#) must be hit by [Conformance Tests](#)
- ☐ (R) Production readiness review completed
- ☐ (R) Production readiness review approved
- ☐ "Implementation History" section is up-to-date for milestone
- ☐ User-facing documentation has been created in [kubernetes/website](#), for publication to [kubernetes.io](#)

- ❑ Supporting documentation—e.g., additional design documents, links to mailing list discussions/SIG meetings, relevant PRs/issues, release notes

Summary

Sidecar containers are a new type of containers that start among the Init containers, run through the lifecycle of the Pod and don't block pod termination. Kubelet makes a best effort to keep them alive and running while other containers are running.

Motivation

The concept of sidecar containers has been around since the early days of Kubernetes. A clear example is [this Kubernetes blog post](#) from 2015 mentioning the sidecar pattern.

Over the years the sidecar pattern has become more common in applications, gained popularity and the use cases are getting more diverse. The current Kubernetes primitives handle that well, but they fall short for several use cases and force weird work-arounds in the applications.

The next sections expand on what the current problems are. But, to give more context, it is important to highlight that some companies are already using a fork of Kubernetes with this sidecar functionality added (not all implementations are the same, but more than one company has a fork for this).

Problems: jobs with sidecar containers

Imagine you have a Job with two containers: one which does the main processing of the job and the other is just a sidecar facilitating it. This sidecar could be a service mesh, a metrics gathering statsd server, etc.

When the main processing finishes, the pod won't terminate until the sidecar container finishes too. This is problematic for sidecar containers that run continuously.

There is no simple way to handle this on Kubernetes today. There are work-arounds for this problem, most of them consist of some form of coupling between the containers to add some logic where a container that finishes communicates it so other containers can react. But it gets tricky when you have more than one sidecar container or want to auto-inject sidecars.

The sidecar will also not be restarted for jobs with `restartPolicy:Never` when it was OOM killed, which may render the pod unusable if the sidecar provided secure communication to other containers. The issue gets complicated by the fact that sidecar containers typically have smaller request, making them the first target as OOM score adjustment uses the request as a main input for calculation.

Problems: log forwarding and metrics sidecar

A log forwarding sidecar should start before several other containers, to simplify getting logs from the startup of other applications and from the Init containers. Let's call "main" container the app that will log and "logging" container the sidecar that will facilitate it.

If the logging container starts after the main app, special logic needs to be implemented to gather logs from the main app. Furthermore, if the logging container is not yet started and the main app crashes on startup, those logs are more likely to be lost (depends if logs go to a shared volume or over the network on localhost, etc.). While you can modify your application to handle this scenario during startup (as it is probably the change you need to do to handle sidecar crashes), for shutdown this approach won't work.

On shutdown the ordering behavior is arguably more important: if the logging container is stopped first, logs for other containers are lost. No matter if those containers queue them and retry to send them to the logging container, or if they are persisted to a shared volume. The logging container is already killed and will not be restarted, as the pod is shutting down. In these cases, logs are lost.

The same things regarding startup and shutdown apply for a metrics container.

Problems: service mesh

Service mesh presents a similar problem: you want the service mesh container to be running and ready before other containers start, so that any inbound/outbound connections that a container can initiate go through the service mesh.

A similar problem happens for shutdown: if the service mesh container is terminated prior to the other containers, outgoing traffic from other apps will be blackholed or not use the service mesh.

However, as none of these are possible to guarantee, most service meshes (like Linkerd and Istio), use fragile and platform specific workarounds to have the basic functionality. For example, for termination, projects like [kubexit](#) or [custom solutions](#) are used.

Some service meshes depend on secret (like a certificate) downloaded by other init container to establish secure communication between services. This makes the problem of ordering of sidecar and init containers harder.

Another complication is between log forwarding and service mesh sidecars. Service mesh sidecars would provide the networking while log forwarding needs to be active to upload logs. Startup and tear down sequence of those may be a complicated problem.

Problems: configuration / secrets

Some pods use init containers to pull down configuration/secrets and update them before the main container gets it. Then use sidecars to continue to watch for changes and perform the updates and push to the main container. This requires two separate code paths today. Perhaps the same sidecar could be used for both cases.

Goals

This proposal aims to:

- make containers implementing the sidecar pattern first class citizens inside a Pod
- solve a Job completion issue when sidecars should run continuously
- allow mixing initContainers and sidecars for a choreographed startup sequence
- allow easy injection of sidecar containers in any Pod
- (to be evaluated after alpha) allow to implement sidecar containers that will guaranteed to be running longer than regular containers

Non-Goals

This proposal doesn't aim to:

- support arbitrary dependencies graphs between containers
- act as a security control to enforce that pod containers only run while the sidecar is healthy. Restart of sidecar containers is a best effort
- allow to enforce security boundaries to sidecar containers different that other containers. For example, allow Istio to run privileged to configure ip tables and disable other containers from doing so
- (alpha) support containers termination ordering

Proposal

The proposal is to introduce a `restartPolicy` field to init containers and use it to indicate that an init container is a sidecar container. Kubelet will start init containers with `restartPolicy=Always` in the order with other init containers, but instead of waiting for its completion, it will wait for the container startup completion.

The condition for startup completion will be that the startup probe succeeded (or if no startup probe defined) and `postStart` handler completed. This condition is represented with the field `started` of `ContainerStatus` type. See the section "[Pod startup completed condition](#)" for considerations on picking this signal.

As part of the KEP, init containers and regular containers will be split into two different types. The field `restartPolicy` will only be introduced on init containers. The only supported value proposed in this KEP is `Always`. No other values will be defined as part of this KEP. Moreover, the field will be nullable so the default value will be "no value".

Other values for `restartPolicy` of containers will not be accepted and containers will follow the logic is currently implemented (documented [here](#) and more details can be found in the section "[Future use of restartPolicy field](#)").

Sidecar containers will not block Pod completion - if all regular containers complete, sidecar containers will be terminated.

During the sidecar startup stage the restart behavior will be similar to init containers. If Pod `restartPolicy` is `Never`, sidecar container failed during startup will NOT be restarted and the whole Pod will fail. If Pod `restartPolicy` is `Always` or `OnFailure`, it will be restarted.

Once sidecar container is started (`postStart` completed and startup probe succeeded), this containers will be restarted even when the Pod `restartPolicy` is `Never` or `OnFailure`. Furthermore, sidecar containers will be restarted even during Pod termination.

In order to minimize OOM kills of sidecar containers, the OOM adjustment for these containers will match or exceed the OOM score adjustment of regular containers in the Pod. This intent to solve the issue [kubernetes/kubernetes#111356](#)

As part of this KEP we also will be enabling for sidecar containers (those will not be enabled for other init containers):

- `PostStart` and `PreStop` lifecycle handlers for sidecar containers
- All probes (startup, readiness, liveness)
- Readiness probes of sidecars will contribute to determine the whole Pod readiness.

```
kind: Pod
spec:
  initContainers:
    - name: vault-agent
      image: hashicorp/vault:1.12.1
    - name: istio-proxy
      image: istio/proxyv2:1.16.0
      args: ["proxy", "sidecar"]
      restartPolicy: Always
  containers:
    ...
```

Naming

This section explains the motivation for naming, assuming that sidecar containers and init containers belong to the same collection and distinguished using a field. Other alternatives are outlined in other section.

Collection name

For this KEP it is important to have sidecar containers be defined among other init containers to be able to express the initialization order of containers. The name `initContainers` is not a good fit for sidecar containers as they typically do more than initialization. The better name can be “infrastructure” containers. The current idea is to implement sidecars as a part of `initContainers` and if this introduces too much trouble, the new collection name may replace the old collection name in future.

Alternative is to introduce a new collection `infrastructureContainers` that replaces semantically `initContainers` and deprecate the `initContainers`. This collection will allow both - init containers and sidecar containers. Decision on this alternative can be postponed to after alpha implementation.

Another *alternative* is to instead of containers, insert placeholders to the `initContainers` collection. Containers themselves are defined in other collection. This option will likely confuse end users more than will help.

Reuse of `restartPolicy` field and enum

The per-container restart policy was a long standing request from the community. Implementing per-container restart policy introduces a set of challenging problems for the pod lifecycle. For example, the state keeping for containers which has already run to completion. Introducing sidecar containers is another scenario where this field can be semantically used.

Introducing this field on containers opens up opportunities to implement those long-standing requests from the community.

Alternative is to introduce a new field: `ambient: true` on all containers. This property will make containers be restarted all the time, and will not block the pod completion.

- Pros: detaching sidecar proposal from the per-container `restartPolicy` proposal
- Cons: this field is a new concept that will be introduced for the same property that is typically controlled by `restartPolicy`.

Use Always vs. New enum value

The semantic of an `Always` enum value is very close to what sidecars require. This is why reusing `Always` to represent sidecars makes a lot of sense for Init containers.

There are a few pros and cons to reuse `Always` as the value instead of introducing a new enum value `UntilPodTermination` / `UntilPodShutdown` / `WithPod` / `Ambient` for `restartPolicy` on containers.

Pros:

- Allows the same enum values to be used for the `restartPolicy` of both pods and containers, but semantics that are mostly the same.
- Less

Cons:

- There are slight differences between the semantics of `Always` for containers and `Always` for pods. The main difference is that a `initContainer` with `Always` will be terminated when the pod terminates and has no influence over the pod lifecycle. Also, for Pods, the default `restartPolicy` is [documented](#) as `Always` but for `initContainers` it will default to `onFailure`. We believe this can be addressed in the documentation of the existing enum fields.

When in future we may support `Always` on regular containers, there will be interesting case of `Always` having a meaning of non-blocking container for the Pod with `restartPolicy == Never` or `onFailure`. This is easy to explain - Pod lifecycle is controlled by containers with the matching `restartPolicy`. But it may be slightly confusing and needs to be carefully reviewed at the time this feature will be considered.

Risks and Mitigations

The following is the list of hypothetical scenarios how users may decide to abuse the feature or use it in a non-designed way. This is an exercise to understand potential side effects from implementing the feature. We are looking for side effects like abuse of the functionality to implement something error-prone that sidecars were not designed for. Or causing issues like noisy neighbors, etc.

Scenario 1. User decides to use sidecars as a way to run regular containers

At least one regular container is required by the Pod. Users may decide to run all other containers as sidecars and have a single container with the Sleep logic inside with timeout or waiting for some cancellation signal from custom job orchestration portal. This way users may implement execution with a time limit defined by that logic. This is something that users cannot express today “declaratively” and will be possible using new sidecar containers. One may imagine a scenario when a third party jobs orchestration tool converts all containers to sidecar containers programmatically and injects a single job orchestration container into the Pod. This can be seen as an abuse of the concept, and may lead to issues when pods terminate unexpectedly or jobs meant to be run as regular containers start being run as sidecars being restarted on Completion. However we do not see a major problem with it as this use of sidecar containers will unlikely lead to “unexpected” behavior - all side effects are quite clear.

Another reason for users to implement regular containers as sidecars would be to use any special properties kubelet may apply to sidecar containers. For example, if restart backoff timeouts will be minimized for sidecar containers (which was proposed and rejected), users may decide to use this feature for regular containers by running those as sidecars. Current proposal as written doesn’t introduce any special properties for sidecars that users may start abusing. Special OOM score adjustment will unlikely be useful as this kind of abuse will likely be needed for bigger containers that does not have the problem of OOM score adjustment to be too low.

Scenario 1.a

User decides to run regular container as a sidecar to simply guarantee the startup sequence of the containers. Users can already implement the startup sequencing by using blocking nature of `postStart` hook. So using the sidecar containers instead is not adding any benefits.

As for injecting sidecars into Pods with regular containers run as sidecars - webhooks will likely inject sidecar containers to be first, so the risk is minimal here as well.

Scenario 2. Balloon sidecars

Users may decide to start the “large request” sidecar containers early to pre-allocate resources for other containers. The same time asking for less resources for a regular containers hoping to reuse what is allocated by the “balloon” sidecar. This was previously impossible to pre-allocate resources like CPU before any init containers run and may be more critical when resource managers like CPU or topology managers are being used. If this pattern have any benefits, people may be incentivized to abuse sidecars this way. However it’s unclear if this has any benefits for users at all.

Scenario 3. Long initialization tasks running in-parallel

Users may decide to implement long initialization tasks that will run in-parallel with other initialization tasks. Users may also decide to run new type of initialization tasks like image preloading for workload containers from the sidecar container, which will make it run in parallel with the Initialization tasks. This is impossible to do today as only a single Init container is run at a time today. For containers with lengthy initialization this pattern may be abused and can lead to the pattern of sidecars synchronization when the first workload container will wait for all Init sidecar containers to complete. This pattern may lead to race conditions and be error prone.

Today similar behavior can be achieved by running initialization tasks as regular containers, with the special container that blocks workload from execution using synchronization logic in the `PostStart` handler. Sidecars support makes implementation of this feature easier.

There is not much risk, however, even if the user abuses this pattern by converting all init containers in sidecars. Whenever this pattern is useful, the user will most likely need to spend time understanding the logic of different Init Containers and making sure it is not conflicting. This pattern also likely prohibited for large scale customers because sidecar-based initialization will cost resources even after initialization is complete.

Scenario 4. Sidecar that never becomes ready

This is the failure mode when the sidecar never becomes ready. This is functionally equivalent to the init container that never completes and not allowing users to implement any other ways to abuse kubelet.

Scenario 5. Intentional failing or terminating sidecars

Users may implement a sidecar container that intentionally crashes or terminates often. This scenario functionally similar to a regular container restarts often so not much additional overhead or side effects will be created.

Scenario 6. Keeping a sidecar alive to keep consuming cycles on termination

On a multitenant cluster, you may have time limits placed on jobs to enable fair usage across tenants. If a sidecar can prevent termination indefinitely, it could be used to perform computation outside the allowed limit.

Scenario 7. Risk of porting existing sidecars to the new mechanism naively

There is risk associated with people moving sidecars as implemented today to use the new pattern. We didn't receive any feedback on potential downsides. One scenarios that may be affected is if sidecars decided to terminate itself and kubelet keeps trying to restart it as one of the main containers are still being terminated. Based on current patterns for sidecar containers, this is not likely the problem.

Another potential problem may be that sidecars will wait for some condition to mark itself “started” that cannot be met with the new pattern. For example, wait for other containers to fully start. As sidecar will not be marked as startup completed, other init containers will not run and Pod will be stuck on initialization. For example, Knative's sidecar does aggressive probing of the user's container to ensure they're ready prior to marking the sidecar ready itself. This prevents the Pod from being included K8s ready endpoints. See the section "[Pod startup completed condition](#)" for more details.

Switching to the new sidecars approach will slow down Pod start for Istio. Istio today is not blocking other containers to start during its initialization. With the switch to the new model, the separation of Initialization stage and main containers running stage will be more explicit and many implementations will likely wait for sidecar initialization, effectively slowing down Pod startup comparing to current approach. This can be eliminated by slight redesign of a sidecars.

Design Details

Backward compatibility

The new field means that any Pod that is not using this field will behave the same way as before.

The new field will only work with the proper control plane and kubelet. Upgrade and downgrade scenarios will be covered in the further sections.

Outside of Kubernetes-controlled code, there might be third party controllers or existing containers relying on the current behavior of init containers. Behaviors they can rely on:

- Assuming nothing is running in the Pod when init container starts. For instance taking PID of the process assuming there are no sidecars.
- Incorrectly calculating the resource usage of a Pod not taking the new type of containers into account (e.g. some grafana dashboards may need modification)
- Stripping the restartPolicy:Always from init container as an unknown field rendering Pod unfunctional as it will not pass the init stage after this
- OPA rules may reject the Pod with new unknown fields because of failure to parse the new field

These potential incompatibilities will be documented.

Resources calculation for scheduling and pod admission

When calculating whether Pod will fit the Node, resource limits and requests are being calculated.

Resources calculation will change for Pod with sidecar containers. Today resources are calculated as a maximum between the maximum use of an InitContainer and Sum of all regular containers:

$$\text{Max} (\text{Max}(\text{initContainers}), \text{Sum}(\text{Containers}))$$

With the sidecar containers the formula will change.

Easiest formula would be to assume they are running the duration of the init stage as well as regular containers.

$$\text{Max} (\text{Max}(\text{nonSidecarInitContainers}) + \text{Sum}(\text{Sidecar Containers}), \text{Sum}(\text{Sidecar Containers}) + \text{Sum}(\text{Containers}))$$

However the true calculations will be different as all init containers that completed before the first sidecar containers will not need to account for any sidecar containers for the maximum value calculation.

So the formula, assuming the function:

$$\text{InitContainerUse}(i) = \text{Sum}(\text{sidecar containers with index} < i) + \text{InitContainer}(i)$$

Is this:

$$\text{Max} (\text{Max}(\text{each InitContainerUse}) , \text{Sum}(\text{Sidecar Containers}) + \text{Sum}(\text{Containers}))$$

There is also a [Pod overhead](#) that is being added to the resource usage. This section assumes it will be added by kubelet independently from this formulae computation.

Exposing Pod Resource requirements

It's currently not straightforward for users to know the effective resource requirements for a pod. The formula for this is:

$$\text{Max} (\text{Max}(\text{initContainers}), \text{Sum}(\text{Containers})) + \text{pod overhead} .$$

This is derived from the fact that init containers run serially and to completion before non-init containers. The effective request for each resource is then the maximum of the largest request for a resource in any init container, and the sum of that resources across all non-init containers.

The introduction of in place pod updates of resource requirement in [KEP#1287](#) further complicates effective resource requirement calculation as

`Pod.Spec.Containers[i].Resources` becomes a desired state field and may not represent the actual resources in use. The KEP notes that:

Schedulers should use the larger of `Spec.Containers[i].Resources` and `Status.ContainerStatuses[i].ResourcesAllocated` when considering available space on a node.

We can introduce `containerUse` to represent this value:

```
ContainerUse(i) = Max(Spec.Containers[i].Resources,  
Status.ContainerStatuses[i].ResourcesAllocated)
```

In the absence of KEP 1287, or if the feature is disabled, `containerUse` is simply:

```
ContainerUse(i) = Spec.Containers[i].Resources
```

The sidecar KEP also changes that calculation to be more complicated as sidecar containers are init containers that do not terminate. Since init containers start in order, sidecar resource usage needs to be summed into those init containers that start after the sidecar. Defining `InitContainerUse` as:

```
InitContainerUse(i) = Sum(sidecar containers with index < i) +  
Max(Spec.InitContainers[i].Resources,  
Status.InitContainerStatuses[i].ResourcesAllocated)
```

allows representing the new formula for a pods resource usage

```
Max ( Max( each InitContainerUse ) , Sum(Sidecar Containers) + Sum(each  
ContainerUse) ) + pod overhead
```

Even now, users still sometimes find how a pod's effective resource requirements are calculated confusing or are just unaware of the formula. The mitigating quality to this is that init container resource requests are usually lower than the sum of non-init container resource requests, and can be ignored by users in those cases. Software that requires accurate pod resource requirement information (e.g. kube-scheduler, kubelet, autoscalers) don't have that luxury. It is too much to ask of users to perform this even more complex calculation simply to know the amount of free capacity they need for a given resource to allow a pod to schedule.

Goals of exposing the `Pod.TotalResourcesRequested` field

- Allow Kubernetes users to quickly identify the effective resource requirements for a pending or scheduled pod directly via observing the pod status.
- Ability to cluster autoscaler, karpenter etc to collect a bunch of resource unavailable/unschedulable pods and easily sum up their `TotalResourcesRequested` to add nodes
- Allow consuming the Pod Requested Resources via metrics:

- Make sure `kube_pod_resource_requests` formula is up to date
- Consider exposing Pod Requirements as a Pod state metric via <https://github.com/kubernetes/kube-state-metrics/blob/main/docs/pod-metrics.md>
- Provide a well documented, reusable, exported function to be used to calculate the effective resource requirements for a `v1.Pod` struct.
- Eliminate duplication of the pod resource requirement calculation within `kubelet` and `kube-scheduler`.

Implementation details

We propose making two changes to satisfy the two primary consumers of the effective pod resource requests, users and other Kubernetes ecosystem software components.

The first change is to add a field to `PodStatus` to represent the effective resource requirements for the pod. The field is named `TotalResourcesRequested`. This field allows users to inspect running and pending pods to quickly determine their effective resource requirements. The field will be first populated by scheduler in the `updatePod`.

The updating of this field would occur the existing `generateAPIPodStatus` method.

```
// TotalResourcesRequested is the effective resource requirements for
// this pod, taking into consideration init containers, sidecars, containers
// and pod overhead.
// More info: https://kubernetes.io/docs/concepts/configuration/manage-
// resources-containers/
// +optional
TotalResourcesRequested ResourceList
`json:"totalResourcesRequested,omitempty"
protobuf:"bytes,8,opt,name=totalResourcesRequested"
```

The second change is to update the `PodRequestsAndLimitsReuse` function to support the new calculation and, if possible, re-use this functionality the other places that pod resource requests are needed (e.g. `kube-scheduler`, `kubelet`). This ensures that components within Kubernetes have an identical computation for effective resource requirements and will reduce maintenance effort. Currently this function is only used for the metrics `kube_pod_resource_request` and `kube_pod_resource_limit` that are exposed by `kube-scheduler` which align with the values that will also now be reported on the pod status.

A correct, exported function is particularly useful for other Kubernetes ecosystem components that need to know the resource requirements for pods that don't exist yet. For example, an autoscaler needs to know what the resource requirements will be for DaemonSet pods when they are created to incorporate them into its calculations if it supports scale to zero.

Notes for implementation

This change could be made in a phased manner:

- Refactor to use the `PodRequestsAndLimitsReuse` function in all situations where pod resource requests are needed.
- Add the new `TotalResourcesRequested` field on `PodStatus` and modify `kubelet` & `kube-scheduler` to update the field.

These two changes are independent of the sidecar and in-place resource update KEPs. The first change doesn't present any user visible change, and if implemented, will in a small way reduce the effort for both of those KEPs by providing a single place to update the pod resource calculation.

Resources calculation and Pod QoS evaluation

Sidecar containers will be used for Pod QoS calculation as all other containers.

The logic in `GetPodQoS` not likely will need changes, but needs to be tested with the sidecar containers.

Topology and CPU managers

`NodeResourcesFit scheduler plugin` will need to be updated take sidecar container resource request into consideration.

Preliminary code analysis didn't expose any issues introducing sidecar containers. The biggest question is resources reuse for sidecar containers and other init containers, especially in cases of single NUMA node requirements and such. This may be non-trivial. The decision on this is not blocking the KEP though.

From the code, it appears that init containers are treated exactly like application containers so we don't need to change anything from resource management point of view. I found references in the code where all the containers (init containers and application containers) were coalesced before resources (CPUs, memory and devices) are allocated to them. Here are a few examples:

- Container Manager: https://github.com/kubernetes/kubernetes/blob/release-1.26/pkg/kubelet/cm/container_manager_linux.go#L708
- CPU Manager: https://github.com/kubernetes/kubernetes/blob/release-1.26/pkg/kubelet/cm/cpumanager/policy_static.go#L490

- Memory Manager: https://github.com/kubernetes/kubernetes/blob/release-1.26/pkg/kubelet/cm/memorymanager/policy_static.go#L372
- Topology Manager:
 - <https://github.com/kubernetes/kubernetes/blob/release-1.26/pkg/kubelet/cm/topologymanager/scope.go#L137>
 - https://github.com/kubernetes/kubernetes/blob/release-1.26/pkg/kubelet/cm/topologymanager/scope_container.go#L52
 - https://github.com/kubernetes/kubernetes/blob/release-1.26/pkg/kubelet/cm/topologymanager/scope_pod.go#L58

Termination of containers

In alpha sidecar containers will be terminated as regular containers. No special or additional signals will be supported.

We will collect feedback from alpha implementation and plan to improve termination in Beta. When Pods with sidecars are terminated:

- Sidecars should not begin termination until all primary containers have terminated.
 - Implicit in this is that sidecars should continue to be restarted until all primary containers have terminated.
- Sidecars should terminate serially and in reverse order. I.e. the first sidecar to initialize should be the last sidecar to terminate.

To address these needs, before promoting this enhancement to Beta, we introduce additional termination grace period fields to manage termination duration ([draft proposal](#)) and leverage these fields to add reverse order termination of sidecar containers after the primary containers terminate.

Other

This behavior needs to be adjusted: [kubernetes#3676](#)

Test Plan

[] I/we understand the owners of the involved components may require updates to existing tests to make this code solid enough prior to committing the changes necessary to implement this enhancement.

Prerequisite testing updates

Unit tests

- `<package> : <date> - <test coverage>`

There will be many packages touched in process. A few that easy to identify by areas of a change:

Admission:

- k8s.io/kubernetes/pkg/kubelet/lifecycle : 61.7

Enable probes for sidecar containers

- k8s.io/kubernetes/pkg/kubelet/prober : 02/07/2023 - 79.9

Include sidecars into QoS decision:

- k8s.io/kubernetes/pkg/kubelet/qos : 02/07/2023 - 100

Include sidecar in resources calculation and policy decisions:

- k8s.io/kubernetes/pkg/kubelet/cm/topologymanager : 02/07/2023 - 93.2
- k8s.io/kubernetes/pkg/kubelet/cm/memorymanager : 02/07/2023 - 81.2
- k8s.io/kubernetes/pkg/kubelet/cm/cpumanager : 02/07/2023 - 86.4

Update OOM score adjustment:

- k8s.io/kubernetes/pkg/kubelet/oom : 02/07/2023 - 57.1

Integration tests

The feature will be covered with unit tests and e2e tests.

e2e tests

Ready state of a sidecar container is properly used to create/delete endpoints

The sidecar container can expose ports and can be used to handle external traffic.

- Pod with sidecar container exposing the port can receive Service traffic to this Port
- Pod with sidecar container exposing the port with readiness probe marks the Endpoint not ready when probe fails and switched back on when readiness probe succeed
- Pod with sidecar container exposing the port can receive Service traffic to this Port during Pod termination (during graceful termination period)

Pod lifecycle scenarios without sidecar containers

TBD: describe test cases that can be affected by introducing sidecar containers.

Pod lifecycle scenarios with sidecar containers

TBD: describe test cases to test with sidecar containers.

Kubelet restart test cases

TBD: describe test cases on how Pod with sidecar containers behaves when kubelet was restarted

API server is down: failure to update containers status during initialization

From @rata. Interesting test cases where the API server is down and kubelet wants to start a pod with sidecars. That was problematic in the previous sidecar KEP iterations.

Let say the API server is up and a pod is being started by the kubelet

- The kubelet still needs to start 3 sidecars in the pod
- The API server crashes (it is not restarted yet)
- The kubelet continues to try to start the sidecars
- The pod is started correctly
- The API server becomes reachable again

I think testing this scenario works is important in early phases (alpha), as in the past that proved to be tricky. The kubelet is authoritative on which containers are started and sends it to the API server, but I hit some bugs in the past where if the API server was down, we couldn't read that a new container was ready from the kubelet. The code to do it was there even if API server was down, but something was not working and I didn't debug it. And the end result was that the pod startup didn't finish until the API server was up again, as that is when we realized from the kubelet that the sidecar was ready.

I think this code has changed since 1.17 when I tested this, but I don't know if this issue is fixed. It is a non-trivial scenario that, if it happens to need more serious code changes in the kubelet to handle it correctly, it will be good to know in early stages of the KEP IMHO.

Resource usage testing

1. Validate that the pod overhead will be accounted for when scheduling a Pod with sidecar container.
2. Validate that LimitRanger will apply defaults and consider limits for the Pod with the sidecar containers.

Upgrade/downgrade testing

1. Kubelet and control plane reject the Pod with the new field if the feature gate is disabled.
2. kubelet and control plane reject the Pod with the new field if the feature gate was disabled AFTER the Pod with the new field was added.

Graduation Criteria

Alpha

- Feature implemented behind a feature flag
- Initial e2e tests completed and enabled
- E2e testing of the existing scenarios with and without the feature gate turned on

Beta

- Implement proper termination ordering.
- Provide defaults for `restartPolicy` field on init containers, `nil` is not an ideal default long term.
- Allow to apply security policies on all containers in `initContainers` collection. Example may be disabling `kubectl exec` on containers in `initContainers` collection.

Upgrade / Downgrade Strategy

Upgrade strategy

Existing sidecars (implemented as regular containers) will still work as intended, as in the past we don't recognize the new field `restartPolicy` today.

Upgrade will not change any current behaviors.

Downgrade strategy

First, there will be no effect on any workload that doesn't use a new field. Any combination of feature gate enabled/disabled or version skew will work as usual for that workload.

So when the new functionality wasn't yet used, downgrade will not be affected.

Due to the new field added to `initContainers` to turn them into sidecars, downgrading to the version without this feature will make all Pods using this flag `unschedulable`. New Pods will be rejected by the control plane and all kubelets. Pods that has already been created will not be rejected by control plane, but once reaching the kubelet, that has this feature disabled or which is old, kubelet will reject the Pod on unmarshalling.

Note, we tested kubelet behavior. For the control plane we may need to implement a new logic to reject such Pods when feature gate got turned off. See [Upgrade/downgrade testing](#) section.

Workloads will have to be deleted and recreated with the old way of handling sidecars. Once there is no more Pods using sidecars, node can be downgraded without side effects.

If downgrade happening from the version with the feature enabled to the previous version that has this feature support, but feature gate is disabled, kubelet and/or control plane will reject these Pods.

Note, downgrade requires node drain. So we will not support scenarios when Pod already running on the node will need to be handled by the restarted kubelet that doesn't know about the sidecar containers.

Version Skew Strategy

Version skew is possible between the control plane and worker nodes as both should be aware of the new field used to flag sidecars inside `initContainers`.

Therefore all cluster nodes, including control plane nodes, must be upgraded before the user can deploy sidecars using the new syntax.

Also, since the feature flag applies to both kubelet and the control plane, similarly all cluster nodes need to have it enabled before deploying Pods with sidecars.

For the scenarios when the feature gate is disabled on control plane, but not disabled on kubelet, users will not be able to schedule Pods with the new field.

For the scenarios when the feature gate is disabled on kubelet, but enabled on control plane, users will be able to create these Pods, but kubelet will reject those.

Production Readiness Review Questionnaire

Feature Enablement and Rollback

How can this feature be enabled / disabled in a live cluster?

- ☒ Feature gate (also fill in values in `kep.yaml`)
 - Feature gate name: `SidecarContainers`
 - Components depending on the feature gate:
 - kubelet
 - kube-apiserver
 - kube-controller-manager
 - kube-scheduler

Does enabling the feature change any default behavior?

No.

Can the feature be disabled once it has been enabled (i.e. can we roll back the enablement)?

Yes. Pods that had sidecars will need to be deleted and recreated without them.

The feature gate disablement will require the kubelet restart. When kubelet will start, it will fail to reconcile the Pod with the new fields and will clean up all running containers.

If version downgrade is involved, the node must be drained. All Pods with the new field will not be accepted by kubelet once feature was disabled.

What happens if we reenable the feature if it was previously rolled back?

If Pods were in Pending State rejected by kubelet due to "unknown" field to be scheduled, they may become scheduleable again and will work as expected.

Are there any tests for feature enablement/disablement?

See [Upgrade/downgrade testing](#).

Rollout, Upgrade and Rollback Planning

How can a rollout or rollback fail? Can it impact already running workloads?

What specific metrics should inform a rollback?

Were upgrade and rollback tested? Was the upgrade->downgrade->upgrade path tested?

Is the rollout accompanied by any deprecations and/or removals of features, APIs, fields of API types, flags, etc.?

Monitoring Requirements

How can an operator determine if the feature is in use by workloads?

How can someone using this feature know that it is working for their instance?

- ☐ Events
 - Event Reason:
- ☐ API .status
 - Condition name:
 - Other field:
- ☐ Other (treat as last resort)
 - Details:

What are the reasonable SLOs (Service Level Objectives) for the enhancement?

What are the SLIs (Service Level Indicators) an operator can use to determine the health of the service?

- ☐ Metrics
 - Metric name:
 - [Optional] Aggregation method:
 - Components exposing the metric:
- ☐ Other (treat as last resort)
 - Details:

Are there any missing metrics that would be useful to have to improve observability of this feature?

Dependencies

Does this feature depend on any specific services running in the cluster?

Scalability

Will enabling / using this feature result in any new API calls?

Will enabling / using this feature result in introducing new API types?

Will enabling / using this feature result in any new calls to the cloud provider?

Will enabling / using this feature result in increasing size or count of the existing API objects?

Will enabling / using this feature result in increasing time taken by any operations covered by existing SLIs/SLOs?

Will enabling / using this feature result in non-negligible increase of resource usage (CPU, RAM, disk, IO, ...) in any components?

Troubleshooting

How does this feature react if the API server and/or etcd is unavailable?

What are other known failure modes?

What steps should be taken if SLOs are not being met to determine the problem?

Implementation History

Drawbacks

Future use of restartPolicy field

In the Alpha version, no other values beyond Always will be acceptable. Going forward the following semantic may be supported for other values of restartPolicy field of init containers. The restartPolicy may be set to one of: Never , OnFailure and Always . The meaning of these values is similar to the Pod 'restartPolicy' flag:

- Never indicates that if the container returns a non-zero exit code, that pod initialization has failed. This value can be supported for Pods with Never and OnFailure restartPolicy indicating the critical Initialization step. Supporting this value on Pods with the restartPolicy Always is not planned as it changes the semantic of this policy and may affect how deployments control the # of Pods in this deployment.
- OnFailure indicates the init container will be restarted until it completes successfully. This value may be used for Pods with the restartPolicy Never , indicating that the Initialization step is flaky and should be retried.
- Always indicates that the container will be considered initialized the moment the startup probe indicates it is started and will be restarted if it exits for any reason until the pod terminates.
- (If the initContainer's restartPolicy is unset, the pod's restartPolicy is used to determine the init container restart policy. For Always, OnFailure is used, for OnFailure - OnFailure, for Never - Never).

Since init container and regular containers share the schema, restartPolicy field will be available for regular containers as well. From scenarios perspective, the first set of scenarios would be for containers that can only override the restartPolicy to “higher” value. From Never to OnFailure or Always , from OnFailure to Always . An override from Always to Never will not be allowed. This will eliminate situations when kubelet restarts and cannot decide whether the container overridden value to Never will need to be restarted or not for the Pod with the restart policy Always . It is also likely that we will require at least one container to have a restartPolicy matching the Pod restartPolicy . We can re-evaluate this in future if scenarios will arise.

Note, by implementing these scenarios, the meaning of `restartPolicy:Always` for containers on a Pod with the `restartPolicy:Never`, will not exactly be the meaning of the `restartPolicy:Always` for the Pod with the `restartPolicy:Always`. For the first case, containers will not block the Pod termination. For the second, they will. In other words, the container with `restartPolicy:Always` will effectively become a sidecar container similar to init container with the same field value.

Additional considerations can be found in the older document that was a previous attempt to implement the sidecar containers:

https://docs.google.com/document/d/1gX_SOZXCNMCIe9d8CkekiT0GU9htH_my8OB7HSm3fM8/edit#heading=h.71o8fcesvgba.

Alternatives

Pod startup completed condition

As part of this KEP, we needed to decide on a condition that Pod needs to meet to be recognized as "started". This will be used to sequence init containers properly and give enough time for sidecar container to complete its initialization.

Readiness probes

The easiest signal we have today is a Container Ready state. Semantically this signal means that the container is fully prepared to handle traffic.

The risk of using of Ready state as a startup completed condition, is:

- today the ready state is used to control traffic, but not for container lifecycle (as oppose to startup and liveness probes). If Container's readiness probe depends on external services, and those services are not yet available or depend on other containers in the Pod, the sidecar container will never be started.
- it may be harder semantically to understand what does the not Ready Sidecar container mean if it became not ready after it has started, while other Init containers are being run.
- the scenario when sidecar container is started, but wants to delay the readiness for later would not be possible to implement.

Startup probes

The second possible signal we can use is Startup probes. Kubelet will be waiting for startup probe to succeed to mark the container as started. Readiness probes will only start when Pod's initialization stage will complete and regular containers will start. There is a `ContainerStatus` field `started` indicating this state.

postStart hook

The easiest approach would be to simply wait for `postStart` hook to complete. This is already implemented for regular containers - they are started sequentially with the `postStart` hook being a blocking call.

Using `postStart` has some debugging challenges in how logs are reported (they aren't) and the status is displayed. If this option will be used, we should consider improving this situation.

With `postStart` simple sidecars will start faster than if any probes are used. However it brings additional challenges for sidecar authors as majority of logic may be duplicated and needs to be authored, e.g. a shell binary + a loop script to retry `curl`.

Alternative 1. Sidecar containers as a separate collection

This alternative was rejected for the following reasons:

- it introduces a whole new collection of containers, which is a big change doesn't allow different startup sequencing - before or after init containers. Before or after a specific init container. Making a single decision for the entire collection of containers and all scenarios was not possible.
- A single semantic for the entire collection of containers may make it harder to introduce new properties to containers in this collection without changing its semantic. It was preferable to introduce new properties representing specific behaviors to the existing containers
- Each time a new collection of containers is added to Pod, it increases the size of the OpenAPI schema of pods and all resources containing PodTemplate significantly, which has negative downstream implications on systems consuming OpenAPI schemas and on CRDs that embed PodTemplate.

Alternative 2. DependOn semantic between containers

This alternative is trying to replicate systemd dependencies. The alternative was rejected for the following reasons:

- Universal sidecars injection is not possible for such a system.
- There were lack of scenarios requiring more than general containers sequencing and two stages - Init and Main execution.

Alternative 3. Phases

This alternative introduces semantically meaningful phases like Network unavailable, Network configured, Storage configured, etc. and fit containers in these phases. This alternative was rejected because it was hard to impossible to fit all containers and scenarios into predefined buckets like this. And the opposite was also true - many scenarios didn't need this level of complexity and separation.

Alternative 4. TerminatePod on container completion

This alternative ([kubernetes#3582](#)) was one of the attempts to scope a problem to the absolute minimum needed to solve Jobs with sidecars termination problems. The idea is to implement a specific property on a container that will terminate the whole Pod when a specific container exits. This idea was rejected:

- Automatic injection is hard
- Only a subset of scenarios is being solved and introduction of built-in sidecar containers support will disregard most of the work
- There is a desire to allow to distinguish sidecar containers from other containers

Alternative 5. Injection of sidecar containers thru the "external" object

The idea of this alternative is to allow configuring some containers for the Pod using another object, so kubelet will combine a resulting Pod definition as a multi-part object. This alternative offers great properties of security boundaries between objects allowing different permissions sets to different containers.

Scenario this and similar alternative may cover are:

More effectively separate access to privileged / secure resources between containers. Filesystem resources, secrets, or more privileged process behaviors can be separated into a container that has higher access and is available before the regular serving/job container and until it has completed.

This is somewhat part of the service mesh use case (especially those that perform privileged iptables manipulation and hide access to secrets).

Other scenarios:

- Run an agent that exchanges pod identity information to gain a time or space limited security credential (such as an SSL cert, remote token, whatever) alongside a web serving container that has no access to the pod identity credential, and thus can only expose the bound credential if compromised.
- Start a FUSE daemon in a sidecar that mounts an object storage bucket into one of the shared pod emptydir volumes - have another container running with much lower privileges that can use that
- Run a container image build with privileges in a sidecar container that responds to commands on a unix socket shared in a pod emptydir and takes commands from an unprivileged container to snapshot the state of the container.

However this alternative was rejected as the change introduced is beyond the scope of the sidecar KEP and will be needed for many more scenarios. Implementing sidecar containers as proposed in this KEP does not make this alternative harder to implement in future.

A little more context can be found in this document:

<https://docs.google.com/document/d/10xqIf19HPPUa58GXtKGO7D1LGX3t4wfmqjJUN4PjfSA/edit#heading=h.xdsy4mbwba7q>

Infrastructure Needed (Optional)

[Give feedback](#)