Frequently Asked Questions

Older versions

The answers in this FAQ apply to the newest (HEAD) version of Cluster Autoscaler. If you're using an older version of CA please refer to corresponding version of this document:

- Cluster Autoscaler 0.5.X
- Cluster Autoscaler 0.6.X

Table of Contents

- Basics
  - What is Cluster Autoscaler?
  - When does Cluster Autoscaler change the size of a cluster?
  - What types of pods can prevent CA from removing a node?
  - Which version on Cluster Autoscaler should I use in my cluster?
  - Is Cluster Autoscaler an Alpha, Beta or GA product?
  - What are the Service Level Objectives for Cluster Autoscaler?
  - How does Horizontal Pod Autoscaler work with Cluster Autoscaler?
  - What are the key best practices for running Cluster Autoscaler?
  - Should I use a CPU-usage-based node autoscaler with Kubernetes?
  - How is Cluster Autoscaler different from CPU-usage-based node autoscalers?
  - Is Cluster Autoscaler compatible with CPU-usage-based node autoscalers?
  - How does Cluster Autoscaler work with Pod Priority and Preemption?
  - How does Cluster Autoscaler remove nodes?
  - How does Cluster Autoscaler treat nodes with status/startup/ignore taints?
- How to?
  - I'm running cluster with nodes in multiple zones for HA purposes. Is that supported by Cluster Autoscaler?
  - How can I monitor Cluster Autoscaler?
  - How can I increase the information that the CA is logging?
  - How can I change the log format that the CA outputs?
  - How can I see all the events from Cluster Autoscaler?
  - How can I scale my cluster to just 1 node?
  - How can I scale a node group to 0?
  - How can I prevent Cluster Autoscaler from scaling down a particular node?
  - How can I prevent Cluster Autoscaler from scaling down non-empty nodes?
  - How can I modify Cluster Autoscaler reaction time?

  - How can I configure overprovisioning with Cluster Autoscaler?
  - How can I enable/disable eviction for a specific DaemonSet
  - How can I enable Cluster Autoscaler to scale up when Node's max volume count is exceeded (CSI migration enabled)?
  - How can I use ProvisioningRequest to run batch workloads?
- Internals
  - Are all of the mentioned heuristics and timings final?
  - How does scale-up work?
  - How does scale-down work?
  - Does CA work with PodDisruptionBudget in scale-down?
  - Does CA respect GracefulTermination in scale-down?
  - How does CA deal with unready nodes?
  - How fast is Cluster Autoscaler?
  - How fast is HPA when combined with CA?
  - Where can I find the designs of the upcoming features?
  - What are Expanders?
  - Does CA respect node affinity when selecting node groups to scale up?
  - What are the parameters to CA?
- Troubleshooting
  - I have a couple of nodes with low utilization, but they are not scaled down. Why?
  - How to set PDBs to enable CA to move kube-system pods?
  - I have a couple of pending pods, but there was no scale-up?
  - CA doesn't work, but it used to work yesterday. Why?
  - How can I check what is going on in CA ?
  - What events are emitted by CA?
  - My cluster is below minimum / above maximum number of nodes, but CA did not fix that! Why?
  - What happens in scale-up when I have no more quota in the cloud provider?
- Developer
  - What go version should be used to compile CA?
  - How can I run e2e tests?
  - How should I test my code before submitting PR?
  - How can I update CA dependencies (particularly k8s.io/kubernetes)?

Basics

What is Cluster Autoscaler?

Cluster Autoscaler is a standalone program that adjusts the size of a Kubernetes cluster to meet the current needs.

When does Cluster Autoscaler change the size of a cluster?

Cluster Autoscaler increases the size of the cluster when:

- there are pods that failed to schedule on any of the current nodes due to insufficient resources.
- adding a node similar to the nodes currently present in the cluster would help.

Cluster Autoscaler decreases the size of the cluster when some nodes are consistently unneeded for a significant amount of time. A node is unneeded when it has low utilization and all of its important pods can be moved elsewhere.

What types of pods can prevent CA from removing a node?

- Pods with restrictive PodDisruptionBudget.
- Kube-system pods that:
  - are not run on the node by default, *
  - don't have a pod disruption budget set or their PDB is too restrictive (since CA 0.6).
- Pods that are not backed by a controller object (so not created by deployment, replica set, job, stateful set etc). *
- Pods with local storage **. *
  - unless the pod has the following annotation set:
    `"cluster-autoscaler.kubernetes.io/safe-to-evict-local-volumes": "volume-1,volume-2,`
    and all of the pod's local volumes are listed in the annotation value.
- Pods that cannot be moved elsewhere due to scheduling constraints. CA simulates kube-scheduler behavior, and if there's no other node where a given pod can schedule, the pod's node won't be scaled down.
  - This can be particularly visible if a given workloads' pods are configured to only fit one pod per node on some subset of nodes. Such pods will always block CA from scaling down their nodes, because all other valid nodes are either taken by another pod, or empty (and CA prefers scaling down empty nodes).
  - Examples of scenarios where scheduling constraints prevent CA from deleting a node:
    * No other node has enough resources to satisfy a pod's request
    * No other node has available ports to satisfy a pod's `hostPort` configuration.
    * No other node with enough resources has the labels required by a pod's node selector
- Pods that have the following annotation set:

```
"cluster-autoscaler.kubernetes.io/safe-to-evict": "false"
```

*Unless the pod has the following annotation (supported in CA 1.0.3 or later):

```
"cluster-autoscaler.kubernetes.io/safe-to-evict": "true"
```

Or you have overridden this behaviour with one of the relevant flags. See below for more information on these flags.

**Local storage in this case considers a Volume configured with properties making it a local Volume, such as the following examples:

- `hostPath`
- `emptyDir` which does **not** use "Memory" for its `emptyDir.medium` field

ConfigMaps, Secrets, Projected volumes and emptyDir with `medium=Memory` are not considered local storage.

Which version on Cluster Autoscaler should I use in my cluster?

See Cluster Autoscaler Releases.

Is Cluster Autoscaler an Alpha, Beta or GA product?

Since version 1.0.0 we consider CA as GA. It means that:

- We have enough confidence that it does what it is expected to do. Each commit goes through a big suite of unit tests with more than 75% coverage (on average). We have a series of e2e tests that validate that CA works well on GCE and GKE. Due to the missing testing infrastructure, AWS (or any other cloud provider) compatibility tests are not the part of the standard development or release procedure. However there is a number of AWS users who run CA in their production environment and submit new code, patches and bug reports.
- It was tested that CA scales well. CA should handle up to 1000 nodes running 30 pods each. Our testing procedure is described here.
- Most of the pain-points reported by the users (like too short graceful termination support) were fixed, however some of the less critical feature requests are yet to be implemented.
- CA has decent monitoring, logging and eventing.
- CA tries to handle most of the error situations in the cluster (like cloud provider stockouts, broken nodes, etc). The cases handled can however vary from cloudprovider to cloudprovider.
- CA developers are committed to maintaining and supporting CA in the foreseeable future.

All of the previous versions (earlier than 1.0.0) are considered beta.

What are the Service Level Objectives for Cluster Autoscaler?

The main purpose of Cluster Autoscaler is to get pending pods a place to run. Cluster Autoscaler periodically checks whether there are any pending pods and increases the size of the cluster if it makes sense and if the scaled up cluster is still within the user-provided constraints. The time of new node provisioning

doesn't depend on CA, but rather on the cloud provider and other Kubernetes components.

So, the main SLO for CA would be expressed in the latency time measured from the time a pod is marked as unschedulable (by K8S scheduler) to the time CA issues scale-up request to the cloud provider (assuming that happens). During our scalability tests (described here) we aimed at max 20sec latency, even in the big clusters. We reach these goals on GCE on our test cases, however in practice, the performance may differ. Hence, users should expect:

- No more than 30 sec latency on small clusters (less than 100 nodes with up to 30 pods each), with the average latency of about 5 sec.
- No more than 60 sec latency on big clusters (100 to 1000 nodes), with average latency of about 15 sec.

Please note that the above performance can be achieved only if NO pod affinity and anti-affinity is used on any of the pods. Unfortunately, the current implementation of the affinity predicate in scheduler is about 3 orders of magnitude slower than for all other predicates combined, and it makes CA hardly usable on big clusters.

It is also important to request full 1 core (or make it available) for CA pod in a bigger clusters. Putting CA on an overloaded node would not allow to reach the declared performance.

We didn't run any performance tests on clusters bigger than 1000 nodes, and supporting them was not a goal for 1.0.

More SLOs may be defined in the future.

How does Horizontal Pod Autoscaler work with Cluster Autoscaler?

Horizontal Pod Autoscaler changes the deployment's or replicaset's number of replicas based on the current CPU load. If the load increases, HPA will create new replicas, for which there may or may not be enough space in the cluster. If there are not enough resources, CA will try to bring up some nodes, so that the HPA-created pods have a place to run. If the load decreases, HPA will stop some of the replicas. As a result, some nodes may become underutilized or completely empty, and then CA will terminate such unneeded nodes.

What are the key best practices for running Cluster Autoscaler?

- Do not modify the nodes belonging to autoscaled node groups directly. All nodes within the same node group should have the same capacity, labels and system pods running on them.
- Specify requests for your pods.
- Use PodDisruptionBudgets to prevent pods from being deleted too abruptly (if needed).

- Check if your cloud provider's quota is big enough before specifying min/max settings for your node pools.
- Do not run any additional node group autoscalers (especially those from your cloud provider).

Should I use a CPU-usage-based node autoscaler with Kubernetes?

No.

How is Cluster Autoscaler different from CPU-usage-based node autoscalers?

Cluster Autoscaler makes sure that all pods in the cluster have a place to run, no matter if there is any CPU load or not. Moreover, it tries to ensure that there are no unneeded nodes in the cluster.

CPU-usage-based (or any metric-based) cluster/node group autoscalers don't care about pods when scaling up and down. As a result, they may add a node that will not have any pods, or remove a node that has some system-critical pods on it, like kube-dns. Usage of these autoscalers with Kubernetes is discouraged.

Is Cluster Autoscaler compatible with CPU-usage-based node autoscalers?

No. CPU-based (or any metric-based) cluster/node group autoscalers, like GCE Instance Group Autoscaler, are NOT compatible with CA. They are also not particularly suited to use with Kubernetes in general.

How does Cluster Autoscaler work with Pod Priority and Preemption?

Since version 1.1 (to be shipped with Kubernetes 1.9), CA takes pod priorities into account.

Pod Priority and Preemption feature enables scheduling pods based on priorities if there is not enough resources. On the other hand, Cluster Autoscaler makes sure that there is enough resources to run all pods. In order to allow users to schedule "best-effort" pods, which shouldn't trigger Cluster Autoscaler actions, but only run when there are spare resources available, we introduced priority cutoff to Cluster Autoscaler.

Pods with priority lower than this cutoff:

- don't trigger scale-ups - no new node is added in order to run them,
- don't prevent scale-downs - nodes running such pods can be terminated.

Nothing changes for pods with priority greater or equal to cutoff, and pods without priority.

Default priority cutoff is -10 (since version 1.12, was 0 before that). It can be changed using `--expendable-pods-priority-cutoff` flag, but we discourage

it. Cluster Autoscaler also doesn't trigger scale-up if an unschedulable pod is already waiting for a lower priority pod preemption.

Older versions of CA won't take priorities into account.

More about Pod Priority and Preemption:

- Priority in Kubernetes API,
- Pod Preemption in Kubernetes,
- Pod Priority and Preemption tutorial.

How does Cluster Autoscaler remove nodes?

Cluster Autoscaler terminates the underlying instance in a cloud-provider-dependent manner.

It does *not* delete the Node object from Kubernetes. Cleaning up Node objects corresponding to terminated instances is the responsibility of the cloud node controller, which can run as part of kube-controller-manager or cloud-controller-manager.

How does Cluster Autoscaler treat nodes with status/startup/ignore taints?

Startup taints

Startup taints are meant to be used when there is an operation that has to complete before any pods can run on the node, e.g. drivers installation.

Cluster Autoscaler treats nodes tainted with `startup taints` as unready, but taken into account during scale up logic, assuming they will become ready shortly.

However, if the substantial number of nodes are tainted with `startup taints` (and therefore unready) for an extended period of time the Cluster Autoscaler might stop working as it might assume the cluster is broken and should not be scaled (creating new nodes doesn't help as they don't become ready).

Startup taints are defined as:

- all taints with the prefix `startup-taint.cluster-autoscaler.kubernetes.io/`,
- all taints defined using `--startup-taint` flag.

Status taints

Status taints are meant to be used when a given node should not be used to run pods for the time being.

Cluster Autoscaler internally treats nodes tainted with `status taints` as ready, but filtered out during scale up logic.

This means that even though the node is ready, no pods should run there as long as the node is tainted and if necessary a scale-up should occur.

Status taints are defined as:

- all taints with the prefix `status-taint.cluster-autoscaler.kubernetes.io/`,
- all taints defined using `--status-taint` flag.

Ignore taints

Ignore taints are now deprecated and treated as startup taints.

Ignore taints are defined as:

- all taints with the prefix `ignore-taint.cluster-autoscaler.kubernetes.io/`,
- all taints defined using `--ignore-taint` flag.

How to?

I'm running cluster with nodes in multiple zones for HA purposes. Is that supported by Cluster Autoscaler?

CA 0.6 introduced `--balance-similar-node-groups` flag to support this use case. If you set the flag to true, CA will automatically identify node groups with the same instance type and the same set of labels (except for automatically added zone label) and try to keep the sizes of those node groups balanced.

This does not guarantee similar node groups will have exactly the same sizes:

- Currently the balancing is only done at scale-up. Cluster Autoscaler will still scale down underutilized nodes regardless of the relative sizes of underlying node groups. We plan to take balancing into account in scale-down in the future.
- Cluster Autoscaler will only add as many nodes as required to run all existing pods. If the number of nodes is not divisible by the number of balanced node groups, some groups will get 1 more node than others.
- Cluster Autoscaler will only balance between node groups that can support the same set of pending pods. If you run pods that can only go to a single node group (for example due to nodeSelector on zone label) CA will only add nodes to this particular node group.

You can opt-out a node group from being automatically balanced with other node groups using the same instance type by giving it any custom label.

How can I monitor Cluster Autoscaler?

Cluster Autoscaler provides metrics and livenessProbe endpoints. By default they're available on port 8085 (configurable with `--address` flag), respectively

under `/metrics` and `/health-check`.

Metrics are provided in Prometheus format and their detailed description is available here.

How can I see all events from Cluster Autoscaler?

By default, the Cluster Autoscaler will deduplicate similar events that occur within a 5 minute window. This is done to improve scalability performance where many similar events might be triggered in a short timespan, such as when there are too many unscheduled pods.

In some cases, such as for debugging or when scalability of events is not an issue, you might want to see all the events coming from the Cluster Autoscaler. In these scenarios you should use the `--record-duplicated-events` command line flag.

How can I scale my cluster to just 1 node?

Prior to version 0.6, Cluster Autoscaler was not touching nodes that were running important kube-system pods like DNS, Metrics Server, Dashboard, etc. If these pods landed on different nodes, CA could not scale the cluster down and the user could end up with a completely empty 3 node cluster. In 0.6, we added an option to tell CA that some system pods can be moved around. If the user configures a PodDisruptionBudget for the kube-system pod, then the default strategy of not touching the node running this pod is overridden with PDB settings. So, to enable kube-system pods migration, one should set minAvailable to 0 (or <= N if there are N+1 pod replicas.) See also I have a couple of nodes with low utilization, but they are not scaled down. Why?

How can I scale a node group to 0?

From CA 0.6 for GCE/GKE and CA 0.6.1 for AWS, it is possible to scale a node group to 0 (and obviously from 0), assuming that all scale-down conditions are met.

For AWS, if you are using `nodeSelector`, you need to tag the ASG with a node-template key `"k8s.io/cluster-autoscaler/node-template/label/"`.

For example, for a node label of `foo=bar`, you would tag the ASG with:

```
{
    "ResourceType": "auto-scaling-group",
    "ResourceId": "foo.example.com",
    "PropagateAtLaunch": true,
    "Value": "bar",
    "Key": "k8s.io/cluster-autoscaler/node-template/label/foo"
}
```

How can I prevent Cluster Autoscaler from scaling down a particular node?

From CA 1.0, node will be excluded from scale-down if it has the annotation preventing scale-down:

```
"cluster-autoscaler.kubernetes.io/scale-down-disabled": "true"
```

It can be added to (or removed from) a node using kubectl:

```
kubectl annotate node <nodename> cluster-autoscaler.kubernetes.io/scale-down-disabled=true
```

How can I prevent Cluster Autoscaler from scaling down non-empty nodes?

CA might scale down non-empty nodes with utilization below a threshold (configurable with `--scale-down-utilization-threshold` flag).

To prevent this behavior, set the utilization threshold to `0`.

How can I modify Cluster Autoscaler reaction time?

There are multiple flags which can be used to configure scale up and scale down delays.

In some environments, you may wish to give the k8s scheduler a bit more time to schedule a pod than the CA's scan-interval. One way to do this is by setting `--new-pod-scale-up-delay`, which causes the CA to ignore unschedulable pods until they are a certain "age", regardless of the scan-interval. This setting can be overridden per pod through `cluster-autoscaler.kubernetes.io/pod-scale-up-delay` annotation. If k8s has not scheduled them by the end of that delay, then they may be considered by the CA for a possible scale-up.

```
"cluster-autoscaler.kubernetes.io/pod-scale-up-delay": "600s"
```

Scaling down of unneeded nodes can be configured by setting `--scale-down-unneeded-time`. Increasing value will make nodes stay up longer, waiting for pods to be scheduled while decreasing value will make nodes be deleted sooner.

How can I configure overprovisioning with Cluster Autoscaler?

Below solution works since version 1.1 (to be shipped with Kubernetes 1.9).

Overprovisioning can be configured using deployment running pause pods with very low assigned priority (see Priority Preemption) which keeps resources that can be used by other pods. If there is not enough resources then pause pods are preempted and new pods take their place. Next pause pods become unschedulable and force CA to scale up the cluster.

The size of overprovisioned resources can be controlled by changing the size of pause pods and the number of replicas. This way you can configure static size

of overprovisioning resources (i.e. 2 additional cores). If we want to configure dynamic size (i.e. 20% of resources in the cluster) then we need to use Horizontal Cluster Proportional Autoscaler which will change number of pause pods depending on the size of the cluster. It will increase the number of replicas when cluster grows and decrease the number of replicas if cluster shrinks.

Configuration of dynamic overprovisioning:

1. (For 1.10, and below) Enable priority preemption in your cluster.

For GCE, it can be done by exporting following env variables before executing kube-up (more details here):

```
export KUBE_RUNTIME_CONFIG=scheduling.k8s.io/v1alpha1=true
export ENABLE_POD_PRIORITY=true
```

For AWS using kops, see this issue.

2. Define priority class for overprovisioning pods. Priority -10 will be reserved for overprovisioning pods as it is the lowest priority that triggers scaling clusters. Other pods need to use priority 0 or higher in order to be able to preempt overprovisioning pods. You can use following definitions.

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: overprovisioning
value: -10
globalDefault: false
description: "Priority class used by overprovisioning."
```

3. Create service account that will be used by Horizontal Cluster Proportional Autoscaler which needs specific roles. More details here

4. Create deployments that will reserve resources. “overprovisioning” deployment will reserve resources and “overprovisioning-autoscaler” deployment will change the size of reserved resources. You can use following definitions (you need to change service account for “overprovisioning-autoscaler” deployment to the one created in the previous step):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: overprovisioning
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: overprovisioning
  template:
```

```yaml
    metadata:
      labels:
        run: overprovisioning
    spec:
      priorityClassName: overprovisioning
      terminationGracePeriodSeconds: 0
      containers:
      - name: reserve-resources
        image: registry.k8s.io/pause:3.9
        resources:
          requests:
            cpu: "200m"

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: overprovisioning-autoscaler
  namespace: default
  labels:
    app: overprovisioning-autoscaler
spec:
  selector:
    matchLabels:
      app: overprovisioning-autoscaler
  replicas: 1
  template:
    metadata:
      labels:
        app: overprovisioning-autoscaler
    spec:
      containers:
        - image: registry.k8s.io/cluster-proportional-autoscaler-amd64:1.8.1
          name: autoscaler
          command:
            - /cluster-proportional-autoscaler
            - --namespace=default
            - --configmap=overprovisioning-autoscaler
            - --default-params={"linear":{"coresPerReplica":1}}
            - --target=deployment/overprovisioning
            - --logtostderr=true
            - --v=2
      serviceAccountName: cluster-proportional-autoscaler-service-account
```

How can I enable/disable eviction for a specific DaemonSet

Cluster Autoscaler will evict DaemonSets based on its configuration, which is common for the entire cluster. It is possible, however, to specify the desired behavior on a per pod basis. All DaemonSet pods will be evicted when they have the following annotation.

```
"cluster-autoscaler.kubernetes.io/enable-ds-eviction": "true"
```

It is also possible to disable DaemonSet pods eviction expicitly:

```
"cluster-autoscaler.kubernetes.io/enable-ds-eviction": "false"
```

Note that this annotation needs to be specified on DaemonSet pods, not the DaemonSet object itself. In order to do that for all DaemonSet pods, it is sufficient to modify the pod spec in the DaemonSet object.

This annotation has no effect on pods that are not a part of any DaemonSet.

How can I enable Cluster Autoscaler to scale up when Node's max volume count is exceeded (CSI migration enabled)?

Kubernetes scheduler will fail to schedule a Pod to a Node if the Node's max volume count is exceeded. In such case to enable Cluster Autoscaler to scale up in a Kubernetes cluster with CSI migration enabled, the appropriate CSI related feature gates have to be specified for the Cluster Autoscaler (if the corresponding feature gates are not enabled by default).

For example:

```
--feature-gates=CSIMigration=true,CSIMigration{Provdider}=true,InTreePlugin{Provider}Unregis
```

For a complete list of the feature gates and their default values per Kubernetes versions, refer to the Feature Gates documentation.

How can I use ProvisioningRequest to run batch workloads

Provisioning Request (abbr. ProvReq) is a new namespaced Custom Resource that aims to allow users to ask CA for capacity for groups of pods. For a detailed explanation of the ProvisioningRequest API, please refer to the original proposal.

Enabling ProvisioningRequest Support

1. **Cluster Autoscaler Version**: Ensure you are using Cluster Autoscaler version 1.30.1 or later.
2. **Feature Flag**: Enable ProvisioningRequest support by setting the following flag in your Cluster Autoscaler configuration: `--enable-provisioning-requests=true`.
3. **Content Type**: Set API content type flag to application/json in your Cluster Autoscaler configuration: `--kube-api-content-type application/json`.

4. **RBAC permissions**: Ensure your cluster-autoscaler pod has the necessary permissions to interact with ProvisioningRequests and PodTemplates:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-autoscaler-provisioning
rules:
  - apiGroups:
    - "autoscaling.x-k8s.io"
    resources:
    - provisioningrequests
    - provisioningrequests/status
    verbs: ["watch", "list", "get", "create", "update", "patch", "delete"]
  - apiGroups: [""]
    resources: ["podtemplates"]
    verbs: ["watch", "list", "get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-autoscaler-provisioning-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-autoscaler-provisioning
subjects:
- kind: ServiceAccount
  name: cluster-autoscaler
  namespace: kube-system
```

5. Deploy the ProvisioningRequest CRD

Supported ProvisioningClasses Currently, ClusterAutoscaler supports following ProvisioningClasses:

- `check-capacity.autoscaling.x-k8s.io`. When using this class, Cluster Autoscaler performs following actions:
  - **Capacity Check**: Determines if sufficient capacity exists in the cluster to fulfill the ProvisioningRequest.
  - **Reservation from other ProvReqs** (if capacity is available): Reserves this capacity for the ProvisioningRequest for 10 minutes, preventing other ProvReqs from using it.
  - **Condition Updates**: Adds a Accepted=True condition when ProvReq is accepted by ClusterAutoscaler and ClusterAutoscaler will

check capacity for this ProvReq. Adds a Provisioned=True condition to the ProvReq if capacity is available. Adds a BookingExpired=True condition when the 10-minute reservation period expires.

Since Cluster Autoscaler version 1.33, it is possible to configure the autoscaler to process only subset of check capacity ProvisioningRequests and ignore the rest. It should be done with caution by specifying `--check-capacity-processor-instance=<name>` flag. Then, ProvReq Parameters map should contain a key "processorInstance" with a value equal to the configured instance name.

This allows to run two Cluster Autoscalers in the cluster, but the second instance (likely this with configured instance name) **should only** handle check capacity ProvisioningRequests and not overlap node groups with the main instance. It is responsibility of the user to ensure the capacity checks are not overlapping. Best-effort atomic ProvisioningRequests processing is disabled in the instance that has this flag set.

For backwards compatibility, it is possible to differentiate the ProvReqs by prefixing provisioningClassName with the instance name, but it is **not recommended** and will be removed in CA 1.35.

- `best-effort-atomic-scale-up.autoscaling.x-k8s.io` (supported from Cluster Autoscaler version 1.30.2 or later). When using this class, Cluster Autoscaler performs following actions:
  - **Capacity Check**: Check which pods could be scheduled on existing capacity.
  - **ScaleUp Request**: Evaluates if scaling up a node group could fulfill all remaining requirements of the ProvisioningRequest. The scale-up request will use the AtomicIncreaseSize method if a given cloud provider supports it. Note that the ScaleUp result depends on the cloud provider's implementation of the AtomicIncreaseSize method. If the method is not implemented, the scale-up request will try to increase the node group atomically but doesn't guarantee atomicity.
  - **Reservation from other ProvReqs (if scale up request succeeded)**: Reserves this capacity for the ProvisioningRequest for 10 minutes, preventing other ProvReqs from using it.
  - **Condition Updates**:
    - Adds a Accepted=True condition when ProvReq is accepted by ClusterAutoscaler.
    - Adds a Provisioned=True condition to the ProvReq if the node group scale up request is successful.
    - Adds a BookingExpired=True condition when the 10-minute reservation period expires.

Note: make sure you setup –max-nodes-per-scaleup flag correctly. By default –max-nodes-per-scaleup=1000, so any scale up that require more than 1000 nodes will be rejected.

Example Usage Deploy the first 2 resources, observe the request being Approved and Provisioned, then deploy the Deployment and observe the Deployment using up the Request.

```yaml
apiVersion: v1
kind: PodTemplate
metadata:
  name: template
template:
  spec:
    containers:
    - image: ubuntu
      name: default
      resources:
        requests:
          cpu: "1"
          memory: 600Mi

---
apiVersion: autoscaling.x-k8s.io/v1
kind: ProvisioningRequest
metadata:
  name: provider
spec:
  provisioningClassName: "best-effort-atomic-scale-up.autoscaling.x-k8s.io"
  podSets:
  - podTemplateRef:
      name: cluster-autoscaler
    count: 10

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: consumer
  annotations:
    autoscaling.x-k8s.io/consume-provisioning-request: provider
spec:
  replicas: 10
  selector:
    matchLabels:
      app: consumer
```

```yaml
template:
  metadata:
    labels:
      app: consumer
  spec:
    containers:
    - name: default
      image: ubuntu
      resources:
        requests:
          cpu: "1"
          memory: 600Mi
      args: ["sleep"]
```

How can I tune Cluster Autoscaler's performance for processing ProvisioningRequests?

Cluster Autoscaler can be run in batch processing mode for CheckCapacity ProvisioningRequests. In this mode, Cluster Autoscaler processes multiple CheckCapacity ProvisioningRequests in a single iteration. This mode is useful for scenarios where a large number of CheckCapacity ProvisioningRequests need to be processed.

However, enabling batch processing for CheckCapacity ProvisioningRequests can adversely affect the performance of processing other types of ProvisioningRequests and incoming pods since iterations where CheckCapacity ProvisioningRequests are processed will take longer and scale-ups for other types of ProvisioningRequests and incoming pods will not be processed during that time.

Enabling Batch Processing

1. **Cluster Autoscaler Version**: Ensure you are using Cluster Autoscaler version 1.32.0 or later.
2. **Feature Flag**: Batch processing is disabled by default, it can be enabled by setting the following flag in your Cluster Autoscaler configuration: `--check-capacity-batch-processing=true`.
3. **Batch Size**: Set the maximum number of CheckCapacity ProvisioningRequests to process in a single iteration by setting the following flag in your Cluster Autoscaler configuration: `--check-capacity-provisioning-request-max-batch-size=<batch-`
   The default value is 10.
4. **Batch Timebox**: Set the maximum time in seconds that Cluster Autoscaler will spend processing CheckCapacity ProvisioningRequests in a single iteration by setting the following flag in your Cluster Autoscaler configuration: `--check-capacity-provisioning-request-batch-timebox=<timebox>`.
   The default value is 10s.

Internals

Are all of the mentioned heuristics and timings final?

No. We reserve the right to update them in the future if needed.

How does scale-up work?

Scale-up creates a watch on the API server looking for all pods. It checks for any unschedulable pods every 10 seconds (configurable by `--scan-interval` flag). A pod is unschedulable when the Kubernetes scheduler is unable to find a node that can accommodate the pod. For example, a pod can request more CPU that is available on any of the cluster nodes. Unschedulable pods are recognized by their PodCondition. Whenever a Kubernetes scheduler fails to find a place to run a pod, it sets "schedulable" PodCondition to false and reason to "unschedulable". If there are any items in the unschedulable pods list, Cluster Autoscaler tries to find a new place to run them.

It is assumed that the underlying cluster is run on top of some kind of node groups. Inside a node group, all machines have identical capacity and have the same set of assigned labels. Thus, increasing a size of a node group will create a new machine that will be similar to these already in the cluster - they will just not have any user-created pods running (but will have all pods run from the node manifest and daemon sets.)

Based on the above assumption, Cluster Autoscaler creates template nodes for each of the node groups and checks if any of the unschedulable pods would fit on a new node. While it may sound similar to what the real scheduler does, it is currently quite simplified and may require multiple iterations before all of the pods are eventually scheduled. If there are multiple node groups that, if increased, would help with getting some pods running, different strategies can be selected for choosing which node group is increased. Check What are Expanders? section to learn more about strategies.

It may take some time before the created nodes appear in Kubernetes. It almost entirely depends on the cloud provider and the speed of node provisioning, including the TLS bootstrapping process. Cluster Autoscaler expects requested nodes to appear within 15 minutes (configured by `--max-node-provision-time` flag.) After this time, if they are still unregistered, it stops considering them in simulations and may attempt to scale up a different group if the pods are still pending. It will also attempt to remove any nodes left unregistered after this time.

> Note: Cluster Autoscaler is **not** responsible for behaviour and registration to the cluster of the new nodes it creates. The responsibility of registering the new nodes into your cluster lies with the cluster

provisioning tooling you use. Example: If you use kubeadm to provision your cluster, it is up to you to automatically execute `kubeadm join` at boot time via some script.

How does scale-down work?

Every 10 seconds (configurable by `--scan-interval` flag), if no scale-up is needed, Cluster Autoscaler checks which nodes are unneeded. A node is considered for removal when **all** below conditions hold:

- The sum of cpu requests and sum of memory requests of all pods running on this node (DaemonSet pods and Mirror pods are included by default but this is configurable with `--ignore-daemonsets-utilization` and `--ignore-mirror-pods-utilization` flags) are smaller than 50% of the node's allocatable. (Before 1.1.0, node capacity was used instead of allocatable.) Utilization threshold can be configured using `--scale-down-utilization-threshold` flag.
- All pods running on the node (except these that run on all nodes by default, like manifest-run pods or pods created by daemonsets) can be moved to other nodes. See What types of pods can prevent CA from removing a node? section for more details on what pods don't fulfill this condition, even if there is space for them elsewhere. While checking this condition, the new locations of all movable pods are memorized. With that, Cluster Autoscaler knows where each pod can be moved, and which nodes depend on which other nodes in terms of pod migration. Of course, it may happen that eventually the scheduler will place the pods somewhere else.
- It doesn't have scale-down disabled annotation (see How can I prevent Cluster Autoscaler from scaling down a particular node?)

If a node is unneeded for more than 10 minutes, it will be terminated. (This time can be configured by flags - please see I have a couple of nodes with low utilization, but they are not scaled down. Why? section for a more detailed explanation.) Cluster Autoscaler terminates one non-empty node at a time to reduce the risk of creating new unschedulable pods. The next node may possibly be terminated just after the first one, if it was also unneeded for more than 10 min and didn't rely on the same nodes in simulation (see below example scenario), but not together. Empty nodes, on the other hand, can be terminated in bulk, up to 10 nodes at a time (configurable by `--max-empty-bulk-delete` flag.)

What happens when a non-empty node is terminated? As mentioned above, all pods should be migrated elsewhere. Cluster Autoscaler does this by evicting them and tainting the node, so they aren't scheduled there again.

DaemonSet pods may also be evicted. This can be configured separately for empty (i.e. containing only DaemonSet pods) and non-empty nodes with `--daemonset-eviction-for-empty-nodes` and `--daemonset-eviction-for-occupied-nodes`

flags, respectively. Note that the default behavior is different on each flag: by default DaemonSet pods eviction will happen only on occupied nodes. Individual DaemonSet pods can also explicitly choose to be evicted (or not). See How can I enable/disable eviction for a specific DaemonSet for more details.

Example scenario:

Nodes A, B, C, X, Y. A, B, C are below utilization threshold. In simulation, pods from A fit on X, pods from B fit on X, and pods from C fit on Y.

Node A was terminated. OK, but what about B and C, which were also eligible for deletion? Well, it depends.

Pods from B may no longer fit on X after pods from A were moved there. Cluster Autoscaler has to find place for them somewhere else, and it is not sure that if A had been terminated much earlier than B, there would always have been a place for them. So the condition of having been unneeded for 10 min may not be true for B anymore.

But for node C, it's still true as long as nothing happened to Y. So C can be terminated immediately after A, but B may not.

Cluster Autoscaler does all of this accounting based on the simulations and memorized new pod location. They may not always be precise (pods can be scheduled elsewhere in the end), but it seems to be a good heuristic so far.

Does CA work with PodDisruptionBudget in scale-down?

From 0.5 CA (K8S 1.6) respects PDBs. Before starting to terminate a node, CA makes sure that PodDisruptionBudgets for pods scheduled there allow for removing at least one replica. Then it deletes all pods from a node through the pod eviction API, retrying, if needed, for up to 2 min. During that time other CA activity is stopped. If one of the evictions fails, the node is saved and it is not terminated, but another attempt to terminate it may be conducted in the near future.

Does CA respect GracefulTermination in scale-down?

CA, from version 1.0, gives pods at most 10 minutes graceful termination time by default (configurable via `--max-graceful-termination-sec`). If the pod is not stopped within these 10 min then the node is terminated anyway. Earlier versions of CA gave 1 minute or didn't respect graceful termination at all.

How does CA deal with unready nodes?

From 0.5 CA (K8S 1.6) continues to work even if some nodes are unavailable. The default number of tolerated unready nodes in CA 1.2.1 or earlier is 33% of total nodes in the cluster or up to 3 nodes, whichever is higher. For CA 1.2.2 and later, it's 45% or 3 nodes. This is configurable by `--max-total-unready-percentage`

and `--ok-total-unready-count` flags. Once there are more unready nodes in the cluster, CA stops all operations until the situation improves. If there are fewer unready nodes, but they are concentrated in a particular node group, then this node group may be excluded from future scale-ups.

How fast is Cluster Autoscaler?

By default, scale-up is considered up to 10 seconds after pod is marked as unschedulable, and scale-down 10 minutes after a node becomes unneeded. Read this section to see how you can modify this behaviour.

Assuming default settings, SLOs described here apply.

How fast is HPA when combined with CA?

When HPA is combined with CA, the total time from increased load to new pods running is determined by three major factors:

- HPA reaction time,
- CA reaction time,
- node provisioning time.

By default, pods' CPU usage is scraped by kubelet every 10 seconds, and it is obtained from kubelet by Metrics Server every 1 minute. HPA checks CPU load metrics in Metrics Server every 30 seconds. However, after changing the number of replicas, HPA backs off for 3 minutes before taking further action. So it can be up to 3 minutes before pods are added or deleted, but usually it's closer to 1 minute.

CA should react as fast as described here, regardless of whether it was HPA or the user that modified the number of replicas. For scale-up, we expect it to be less than 30 seconds in most cases.

Node provisioning time depends mostly on cloud provider. In our experience, on GCE it usually takes 3 to 4 minutes from CA request to when pods can be scheduled on newly created nodes.

Total time is a sum of those steps, and it's usually about 5 minutes. Please note that CA is the least significant factor here.

On the other hand, for scale-down CA is usually the most significant factor, as it doesn't attempt to remove nodes immediately, but only after they've been unneeded for a certain time.

Where can I find the designs of the upcoming features?

CA team follows the generic Kubernetes process and submits design proposals HERE before starting any significant effort. Some of the not-yet-fully-approved proposals may be hidden among PRs.

What are Expanders?

When Cluster Autoscaler identifies that it needs to scale up a cluster due to unschedulable pods, it increases the number of nodes in some node group. When there is one node group, this strategy is trivial. When there is more than one node group, it has to decide which to expand.

Expanders provide different strategies for selecting the node group to which new nodes will be added.

Expanders can be selected by passing the name to the `--expander` flag, i.e. `./cluster-autoscaler --expander=random`.

Currently Cluster Autoscaler has 5 expanders:

- `random` - should be used when you don't have a particular need for the node groups to scale differently.
- `most-pods` - selects the node group that would be able to schedule the most pods when scaling up. This is useful when you are using nodeSelector to make sure certain pods land on certain nodes. Note that this won't cause the autoscaler to select bigger nodes vs. smaller, as it can add multiple smaller nodes at once.
- `least-waste` - this is the default expander, selects the node group that will have the least idle CPU (if tied, unused memory) after scale-up. This is useful when you have different classes of nodes, for example, high CPU or high memory nodes, and only want to expand those when there are pending pods that need a lot of those resources.
- `least-nodes` - selects the node group that will use the least number of nodes after scale-up. This is useful when you want to minimize the number of nodes in the cluster and instead opt for fewer larger nodes. Useful when chained with the `most-pods` expander before it to ensure that the node group selected can fit the most pods on the fewest nodes.
- `price` - select the node group that will cost the least and, at the same time, whose machines would match the cluster size. This expander is described in more details HERE. Currently it works only for GCE, GKE and Equinix Metal (patches welcome.)
- `priority` - selects the node group that has the highest priority assigned by the user. It's configuration is described in more details here

From 1.23.0 onwards, multiple expanders may be passed, i.e. `.cluster-autoscaler --expander=priority,least-waste`

This will cause the `least-waste` expander to be used as a fallback in the event that the priority expander selects multiple node groups. In general, a list of expanders can be used, where the output of one is passed to the next and the

final decision by randomly selecting one. An expander must not appear in the list more than once.

Does CA respect node affinity when selecting node groups to scale up?

CA respects `nodeSelector` and `requiredDuringSchedulingIgnoredDuringExecution` in nodeAffinity given that you have labelled your node groups accordingly. If there is a pod that cannot be scheduled with either `nodeSelector` or `requiredDuringSchedulingIgnoredDuringExecution` specified, CA will only consider node groups that satisfy those requirements for expansion.

However, CA does not consider "soft" constraints like `preferredDuringSchedulingIgnoredDuringExecution` when selecting node groups. That means that if CA has two or more node groups available for expansion, it will not use soft constraints to pick one node group over another.

What are the parameters to CA?

The following startup parameters are supported for cluster autoscaler:

Parameter	Description	Default
`add-dir-header`	If true, adds the file directory to the header of the log messages	
`address`	The address to expose prometheus metrics.	":8085"
`alsologtostderr`	log to standard error as well as files (no effect when -logtostderr=true)	
`async-node-groups`	Whether clusterautoscaler creates and deletes node groups asynchronously. Experimental: requires cloud provider supporting async node group operations, enable at your own risk.	
`aws-use-static-instance-list`	Should CA fetch instance types in runtime or use a static list. AWS only	

Parameter	Description	Default
`balance-similar-node-groups`	Detect similar node groups and balance the number of nodes between them	
`balancing-ignore-label`	Specifies a label to ignore in addition to the basic and cloud-provider set of labels when comparing if two node groups are similar	[]
`balancing-label`	Specifies a label to use for comparing if two node groups are similar, rather than the built in heuristics. Setting this flag disables all other comparison logic, and cannot be combined with –balancing-ignore-label.	[]
`bulk-mig-instances-listing-enabled`	Fetch GCE mig instances in bulk instead of per mig	false
`bypassed-scheduler-names`	Names of schedulers to bypass. If set to non-empty value, CA will not wait for pods to reach a certain age before triggering a scale-up.	
`check-capacity-batch-processing`	Whether to enable batch processing for check capacity requests.	false

Parameter	Description	Default
`check-capacity-processor-instance`	Name of the processor instance. Only ProvisioningRequests that define this name in their parameters with the key "processorInstance" will be processed by this CA instance. It only refers to check capacity ProvisioningRequests, but if not empty, best-effort atomic ProvisioningRequests processing is disabled in this instance. Not recommended: Until CA 1.35, ProvisioningRequests with this name as prefix in their class will be also processed.	
`check-capacity-provisioning-request-batch-timeout`	Maximum time to process a batch of provisioning requests.	10s
`check-capacity-provisioning-request-max-batch-size`	Maximum number of provisioning requests to process in a single batch.	10
`cloud-config`	The path to the cloud provider configuration file. Empty string for no configuration file.	
`cloud-provider`	Cloud provider type. Available values: [aws,azure,gce,alicloud,cherryservers,cloudstack,baiducloud,magnum,digitalocean,exo	"gce"
`cloud-provider-gce-l7lb-src-cidrs`	CIDRs opened in GCE firewall for L7 LB traffic proxy & health checks	130.211.0.0/22,35.191.0.0/16
`cloud-provider-gce-lb-src-cidrs`	CIDRs opened in GCE firewall for L4 LB traffic proxy & health checks	130.211.0.0/22,209.85.152.0/22,209.85.204.0/22,35.191.0.
`cluster-name`	Autoscaled cluster name, if available	

Parameter	Description	Default
`cluster-snapshot-parallelism`	Maximum parallelism of cluster snapshot creation.	16
`clusterapi-cloud-config-authoritative`	Treat the cloud-config flag authoritatively (do not fallback to using kubeconfig flag). ClusterAPI only	
`cordon-node-before-terminating`	Should CA cordon nodes before terminating during downscale process	
`cores-total`	Minimum and maximum number of cores in cluster, in the format :. Cluster autoscaler will not scale the cluster beyond these numbers.	“0:320000”
`daemonset-eviction-for-empty-nodes`	DaemonSet pods will be gracefully terminated from empty nodes	
`daemonset-eviction-for-occupied-nodes`	DaemonSet pods will be gracefully terminated from non-empty nodes	true
`debugging-snapshot-enabled`	Whether the debugging snapshot of cluster autoscaler feature is enabled	

Parameter	Description	Default
`drain-priority-config`	List of ‘,’ separated pairs (priority:terminationGracePeriodSeconds) of integers separated by ‘:’ enables priority evictor. Priority evictor groups pods into priority groups based on pod priority and evict pods in the ascending order of group priorities–max-graceful-termination-sec flag should not be set when this flag is set. Not setting this flag will use unordered evictor by default.Priority evictor reuses the concepts of drain logic in kubelet(https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/2712-pod-priority-based-graceful-node-shutdown#migration-from-the-node-graceful-shutdown-feature).Eg. flag usage: ‘10000:20,1000:100,0:60’	
`dynamic-node-delete-delay-after-taint-enabled`	Enable/disable dynamic adjustment of NodeDeleteDelayAfterTaint based of the latency between CA and api-server	
`emit-per-nodegroup-metrics`	If true, emit per node group metrics.	
`enable-dynamic-resource-allocation`	Whether logic for handling DRA (Dynamic Resource Allocation) objects is enabled.	
`enable-proactive-scaleup`	Whether to enable/disable proactive scale-ups, defaults to false	

Parameter	Description	Default
`enable-provisioning-requests`	Whether the clusterautoscaler will be handling the ProvisioningRequest CRs.	
`enforce-node-group-min-size`	Should CA scale up the node group to the configured min size if needed.	
`estimator`	Type of resource estimator to be used in scale up. Available values: [binpacking]	“binpacking”
`expander`	Type of node group expander to be used in scale up. Available values: [random,most-pods,least-waste,price,priority,grpc]. Specifying multiple values separated by commas will call the expanders in succession until there is only one option remaining. Ties still existing after this process are broken randomly.	“least-waste”
`expendable-pods-priority-cutoff`	Pods with priority below cutoff will be expendable. They can be killed without any consideration during scale down and they don’t cause scale up. Pods with null priority (PodPriority disabled) are non expendable.	-10
`feature-gates`	A set of key=value pairs that describe feature gates for alpha/experimental features. Options are:	

Parameter	Description	Default
`force-delete-unregistered-nodes`	Whether to enable force deletion of long unregistered nodes, regardless of the min size of the node group the belong to.	
`force-ds`	Blocks scale-up of node groups too small for all suitable Daemon Sets pods.	
`frequent-loops-enabled`	Whether clusterautoscaler triggers new iterations more frequently when it's needed	
`gce-concurrent-refreshes`	Maximum number of concurrent refreshes per cloud object type.	1
`gce-expander-ephemeral-storage-support`	Whether scale-up takes ephemeral storage resources into account for GCE cloud provider (Deprecated, to be removed in 1.30+)	true
`gce-mig-instances-min-refresh-wait-time`	The minimum time which needs to pass before GCE MIG instances from a given MIG can be refreshed.	5s
`gpu-total`	Minimum and maximum number of different GPUs in cluster, in the format ::. Cluster autoscaler will not scale the cluster beyond these numbers. Can be passed multiple times. CURRENTLY THIS FLAG ONLY WORKS ON GKE.	[]
`grpc-expander-cert`	Path to cert used by gRPC server over TLS	
`grpc-expander-url`	URL to reach gRPC expander server.	

Parameter	Description	Default
`ignore-daemonsets-utilization`	Should CA ignore DaemonSet pods when calculating resource utilization for scaling down	
`ignore-mirror-pods-utilization`	Should CA ignore Mirror pods when calculating resource utilization for scaling down	
`ignore-taint`	Specifies a taint to ignore in node templates when considering to scale a node group (Deprecated, use startup-taints instead)	[]
`initial-node-group-backoff-duration`	initialNodeGroupBackoffDuration is the duration of first backoff after a new node failed to start.	5m0s
`kube-api-content-type`	Content type of requests sent to apiserver.	“application/vnd.kubernetes.protobuf”
`kube-client-burst`	Burst value for kubernetes client.	10
`kube-client-qps`	QPS value for kubernetes client.	5
`kubeconfig`	Path to kubeconfig file with authorization and master location information.	
`kubernetes`	Kubernetes master location. Leave blank for default	
`leader-elect`	Start a leader election client and gain leadership before executing the main loop. Enable this when running replicated components for high availability.	true

Parameter	Description	Default
`leader-elect-lease-duration`	The duration that non-leader candidates will wait after observing a leadership renewal until attempting to acquire leadership of a led but unrenewed leader slot. This is effectively the maximum duration that a leader can be stopped before it is replaced by another candidate. This is only applicable if leader election is enabled.	15s
`leader-elect-renew-deadline`	The interval between attempts by the acting master to renew a leadership slot before it stops leading. This must be less than the lease duration. This is only applicable if leader election is enabled.	10s
`leader-elect-resource-lock`	The type of resource object that is used for locking during leader election. Supported options are 'leases'.	"leases"
`leader-elect-resource-name`	The name of resource object that is used for locking during leader election.	"cluster-autoscaler"
`leader-elect-resource-namespace`	The namespace of resource object that is used for locking during leader election.	
`leader-elect-retry-period`	The duration the clients should wait between attempting acquisition and renewal of a leadership. This is only applicable if leader election is enabled.	2s

Parameter	Description	Default
`log-backtrace-at`	when logging hits line file:N, emit a stack trace	:0
`log-dir`	If non-empty, write log files in this directory (no effect when -logtostderr=true)	
`log-file`	If non-empty, use this log file (no effect when -logtostderr=true)	
`log-file-max-size`	Defines the maximum size a log file can grow to (no effect when -logtostderr=true). Unit is megabytes. If the value is 0, the maximum file size is unlimited.	1800
`log-flush-frequency`	Maximum number of seconds between log flushes	5s
`log-json-info-buffer-size`	[Alpha] In JSON format with split output streams, the info messages can be buffered for a while to increase performance. The default value of zero bytes disables buffering. The size can be specified as number of bytes (512), multiples of 1000 (1K), multiples of 1024 (2Ki), or powers of those (3M, 4G, 5Mi, 6Gi). Enable the LoggingAlphaOptions feature gate to use this.	
`log-json-split-stream`	[Alpha] In JSON format, write error messages to stderr and info messages to stdout. The default is to write a single stream to stdout. Enable the LoggingAlphaOptions feature gate to use this.	

Parameter	Description	Default
`log-text-info-buffer-size`	[Alpha] In text format with split output streams, the info messages can be buffered for a while to increase performance. The default value of zero bytes disables buffering. The size can be specified as number of bytes (512), multiples of 1000 (1K), multiples of 1024 (2Ki), or powers of those (3M, 4G, 5Mi, 6Gi). Enable the LoggingAlphaOptions feature gate to use this.	
`log-text-split-stream`	[Alpha] In text format, write error messages to stderr and info messages to stdout. The default is to write a single stream to stdout. Enable the LoggingAlphaOptions feature gate to use this.	
`logging-format`	Sets the log format. Permitted formats: “json” (gated by LoggingBetaOptions), “text”.	“text”
`logtostderr`	log to standard error instead of files	true
`max-allocatable-difference-ratio`	Maximum difference in allocatable resources between two similar node groups to be considered for balancing. Value is a ratio of the smaller node group’s allocatable resource.	0.05

Parameter	Description	Default
`max-autoprovisioned-node-group-count`	The maximum number of autoprovisioned groups in the cluster.This flag is deprecated and will be removed in future releases.	15
`max-binpacking-time`	Maximum time spend on binpacking for a single scale-up. If binpacking is limited by this, scale-up will continue with the already calculated scale-up options.	5m0s
`max-bulk-soft-taint-count`	Maximum number of nodes that can be tainted/untainted PreferNoSchedule at the same time. Set to 0 to turn off such tainting.	10
`max-bulk-soft-taint-time`	Maximum duration of tainting/untainting nodes as PreferNoSchedule at the same time.	3s
`max-drain-parallelism`	Maximum number of nodes needing drain, that can be drained and deleted in parallel.	1
`max-empty-bulk-delete`	Maximum number of empty nodes that can be deleted at the same time. DEPRECATED: Use –max-scale-down-parallelism instead.	10
`max-failing-time`	Maximum time from last recorded successful autoscaler run before automatic restart	15m0s

Parameter	Description	Default
`max-free-difference-ratio`	Maximum difference in free resources between two similar node groups to be considered for balancing. Value is a ratio of the smaller node group's free resource.	0.05
`max-graceful-termination-sec`	Maximum number of seconds CA waits for pod termination when trying to scale down a node. This flag is mutually exclusion with drain-priority-config flag which allows more configuration options.	600
`max-inactivity`	Maximum time from last recorded autoscaler activity before automatic restart	10m0s
`max-node-group-backoff-duration`	Maximum NodeGroupBackoffDuration is the maximum backoff duration for a NodeGroup after new nodes failed to start.	30m0s
`max-node-provision-time`	The default maximum time CA waits for node to be provisioned - the value can be overridden per node group	15m0s
`max-nodegroup-binpacking-duration`	Maximum time that will be spent in binpacking simulation for each NodeGroup.	10s
`max-nodes-per-scaleup`	Max nodes added in a single scale-up. This is intended strictly for optimizing CA algorithm latency and not a tool to rate-limit scale-up throughput.	1000

Parameter	Description	Default
`max-nodes-total`	Maximum number of nodes in all node groups. Cluster autoscaler will not grow the cluster beyond this number.	
`max-pod-eviction-time`	Maximum time CA tries to evict a pod before giving up	2m0s
`max-scale-down-parallelism`	Maximum number of nodes (both empty and needing drain) that can be deleted in parallel.	10
`max-total-unready-percentage`	Maximum percentage of unready nodes in the cluster. After this is exceeded, CA halts operations	45
`memory-difference-ratio`	Maximum difference in memory capacity between two similar node groups to be considered for balancing. Value is a ratio of the smaller node group's memory capacity.	0.015
`memory-total`	Minimum and maximum number of gigabytes of memory in cluster, in the format :. Cluster autoscaler will not scale the cluster beyond these numbers.	"0:6400000"
`min-replica-count`	Minimum number or replicas that a replica set or replication controller should have to allow their pods deletion in scale down	
`namespace`	Namespace in which cluster-autoscaler run.	"kube-system"

Parameter	Description	Default
`new-pod-scale-up-delay`	Pods less than this old will not be considered for scale-up. Can be increased for individual pods through annotation 'cluster-autoscaler.kubernetes.io/pod-scale-up-delay'.	0s
`node-autoprovisioning-enabled`	Should CA autoprovision node groups when needed.This flag is deprecated and will be removed in future releases.	false
`node-delete-delay-after-taint`	How long to wait before deleting a node after tainting it	5s
`node-deletion-batcher-interval`	How long CA ScaleDown gather nodes to delete them in batch.	0s
`node-deletion-delay-timeout`	Maximum time CA waits for removing delay-deletion.cluster-autoscaler.kubernetes.io/ annotations before deleting the node.	2m0s

Parameter	Description	Default
`node-group-auto-discovery`	<discoverer>:[[=]] One or more definition(s) of node group auto-discovery. A definition is expressed :[[=]]. The `aws`, `gce`, and `azure` cloud providers are currently supported. AWS matches by ASG tags, e.g. `asg:tag=tagKey,anotherTagKey`. GCE matches by IG name prefix, and requires you to specify min and max nodes per IG, e.g. `mig:namePrefix=pfx,min=0,max=10` Azure matches by VMSS tags, similar to AWS. And you can optionally specify a default min and max size, e.g. `label:tag=tagKey,anotherTagKey=bar,min=0,max=600`. Can be used multiple times.	[]
`node-group-backoff-reset-timeout`	Node Group Backoff Reset Timeout is the time after last failed scale-up when the backoff duration is reset.	3h0m0s
`node-info-cache-expire-time`	Node Info cache expire time for each item. Default value is 10 years.	87600h0m0s
`nodes`	sets min,max size and other configuration data for a node group in a format accepted by cloud provider. Can be used multiple times. Format: ::<other...>	[]
`ok-total-unready-count`	Number of allowed unready nodes, irrespective of max-total-unready-percentage	3

Parameter	Description	Default
`one-output`	If true, only write logs to their native severity level (vs also writing to each lower severity level; no effect when -logtostderr=true)	
`parallel-scale-up`	Whether to allow parallel node groups scale up. Experimental: may not work on some cloud providers, enable at your own risk.	
`pod-injection-limit`	Limits total number of pods while injecting fake pods. If unschedulable pods already exceeds the limit, pod injection is disabled but pods are not truncated.	5000
`profiling`	Is debug/pprof endpoint enabled	
`provisioning-request-initial-backoff-time`	Initial backoff time for ProvisioningRequest retry after failed ScaleUp.	1m0s
`provisioning-request-max-backoff-cache-size`	Max backoff cache size for ProvisioningRequest cache size used for retry backoff mechanism.	1000
`provisioning-request-max-backoff-time`	Max backoff time for ProvisioningRequest retry after failed ScaleUp.	10m0s
`record-duplicated-events`	enable duplication of similar events within a 5 minute window.	
`regional`	Cluster is regional.	

Parameter	Description	Default
`scale-down-candidates-pool-min-count`	Minimum number of nodes that are considered as additional non empty candidatesfor scale down when some candidates from previous iteration are no longer valid.When calculating the pool size for additional candidates we takemax(#nodes * scale-down-candidates-pool-ratio, scale-down-candidates-pool-min-count).	50
`scale-down-candidates-pool-ratio`	A ratio of nodes that are considered as additional non empty candidates forscale down when some candidates from previous iteration are no longer valid.Lower value means better CA responsiveness but possible slower scale down latency.Higher value can affect CA performance with big clusters (hundreds of nodes).Set to 1.0 to turn this heuristics off - CA will take all nodes as additional candidates.	0.1
`scale-down-delay-after-add`	How long after scale up that scale down evaluation resumes	10m0s
`scale-down-delay-after-delete`	How long after node deletion that scale down evaluation resumes, defaults to scanInterval	0s
`scale-down-delay-after-failure`	How long after scale down failure that scale down evaluation resumes	3m0s

Parameter	Description	Default
`scale-down-delay-type-local`	Should –scale-down-delay-after-* flags be applied locally per nodegroup or globally across all nodegroups	
`scale-down-enabled`	Should CA scale down the cluster	true
`scale-down-gpu-utilization-threshold`	Sum of gpu requests of all pods running on the node divided by node's allocatable resource, below which a node can be considered for scale down.Utilization calculation only cares about gpu resource for accelerator node. cpu and memory utilization will be ignored.	0.5
`scale-down-non-empty-candidates-count`	Maximum number of non empty nodes considered in one iteration as candidates for scale down with drain.Lower value means better CA responsiveness but possible slower scale down latency.Higher value can affect CA performance with big clusters (hundreds of nodes).Set to non positive value to turn this heuristic off - CA will not limit the number of nodes it considers.	30
`scale-down-simulation-timeout`	How long should we run scale down simulation.	30s
`scale-down-unneeded-time`	How long a node should be unneeded before it is eligible for scale down	10m0s

Parameter	Description	Default
`scale-down-unready-enabled`	Should CA scale down unready nodes of the cluster	true
`scale-down-unready-time`	How long an unready node should be unneeded before it is eligible for scale down	20m0s
`scale-down-utilization-threshold`	The maximum value between the sum of cpu requests and sum of memory requests of all pods running on the node divided by node's corresponding allocatable resource, below which a node can be considered for scale down	0.5
`scale-up-from-zero`	Should CA scale up when there are 0 ready nodes.	true
`scan-interval`	How often cluster is reevaluated for scale up or down	10s
`scheduler-config-file`	scheduler-config allows changing configuration of in-tree scheduler plugins acting on PreFilter and Filter extension points	
`skip-headers`	If true, avoid header prefixes in the log messages	
`skip-log-headers`	If true, avoid headers when opening log files (no effect when -logtostderr=true)	
`skip-nodes-with-custom-controller-pods`	If true cluster autoscaler will never delete nodes with pods owned by custom controllers	true

Parameter	Description	Default
skip-nodes-with-local-storage	If true cluster autoscaler will never delete nodes with pods with local storage, e.g. EmptyDir or HostPath	true
skip-nodes-with-system-pods	If true cluster autoscaler will never delete nodes with pods from kube-system (except for DaemonSet or mirror pods)	true
startup-taint	Specifies a taint to ignore in node templates when considering to scale a node group (Equivalent to ignore-taint)	[]
status-config-map-name	Status configmap name	“cluster-autoscaler-status”
status-taint	Specifies a taint to ignore in node templates when considering to scale a node group but nodes will not be treated as unready	[]
stderrthreshold	logs at or above this threshold go to stderr when writing to files and stderr (no effect when -logtostderr=true or -alsologtostderr=true)	2
unremovable-node-recheck-timeout	The timeout before we check again a node that couldn’t be removed before	5m0s
user-agent	User agent used for HTTP calls.	“cluster-autoscaler”
v	number for the log level verbosity	
vmodule	comma-separated list of pattern=N settings for file-filtered logging (only works for text log format)	

Parameter	Description	Default
`write-status-configmap`	Should CA write status information to a configmap	true

Troubleshooting

I have a couple of nodes with low utilization, but they are not scaled down. Why?

CA doesn't remove underutilized nodes if they are running pods that it shouldn't evict. Other possible reasons for not scaling down:

- the node group already has the minimum size,
- node has the scale-down disabled annotation (see How can I prevent Cluster Autoscaler from scaling down a particular node?)
- node was unneeded for less than 10 minutes (configurable by `--scale-down-unneeded-time` flag),
- there was a scale-up in the last 10 min (configurable by `--scale-down-delay-after-add` flag),
- there was a failed scale-down for this group in the last 3 minutes (configurable by `--scale-down-delay-after-failure` flag),
- there was a failed attempt to remove this particular node, in which case Cluster Autoscaler will wait for extra 5 minutes before considering it for removal again,
- using large custom value for `--scale-down-delay-after-delete` or `--scan-interval`, which delays CA action.
- make sure `--scale-down-enabled` parameter in command is not set to false

How to set PDBs to enable CA to move kube-system pods?

By default, kube-system pods prevent CA from removing nodes on which they are running. Users can manually add PDBs for the kube-system pods that can be safely rescheduled elsewhere:

```
kubectl create poddisruptionbudget <pdb name> --namespace=kube-system --selector app=<app na
```

Here's how to do it for some common pods:

- kube-dns can safely be rescheduled as long as there are supposed to be at least 2 of these pods. In 1.7, this will always be the case. For 1.6

and earlier, edit kube-dns-autoscaler config map as described here, adding preventSinglePointFailure parameter. For example:

```
linear:'{"coresPerReplica":256,"nodesPerReplica":16,"preventSinglePointFailure":true}'
```

- Metrics Server is best left alone, as restarting it causes the loss of metrics for >1 minute, as well as metrics in dashboard from the last 15 minutes. Metrics Server downtime also means effective HPA downtime as it relies on metrics. Add PDB for it only if you're sure you don't mind.

I have a couple of pending pods, but there was no scale-up?

CA doesn't add nodes to the cluster if it wouldn't make a pod schedulable. It will only consider adding nodes to node groups for which it was configured. So one of the reasons it doesn't scale up the cluster may be that the pod has too large (e.g. 100 CPUs), or too specific requests (like node selector), and wouldn't fit on any of the available node types. Another possible reason is that all suitable node groups are already at their maximum size.

If the pending pods are in a stateful set and the cluster spans multiple zones, CA may not be able to scale up the cluster, even if it has not yet reached the upper scaling limit in all zones. Stateful set pods require an associated Persistent Volume (PV), which is created before scheduling the pod and CA has no way of influencing the zone choice. The pending pod has a strict constraint to be scheduled in the same zone that the PV is in, so if it is a zone that has already reached the upper scaling limit, CA will not be able to perform a scale-up, even if there are other zones in which nodes could be added. This will manifest itself by following events on the pod:

```
Events:
  Type     Reason             Age   From                Message
  ----     ------             ----  -------             -------
  Normal   NotTriggerScaleUp  ..    cluster-autoscaler  pod didn't trigger scale-up (it woul
  Warning  FailedScheduling   ..    default-scheduler   No nodes are available that match al
```

This limitation was solved with volume topological scheduling introduced as beta in Kubernetes 1.11 and planned for GA in 1.13. To allow CA to take advantage of topological scheduling, use separate node groups per zone. This way CA knows exactly which node group will create nodes in the required zone rather than relying on the cloud provider choosing a zone for a new node in a multi-zone node group. When using separate node groups per zone, the `--balance-similar-node-groups` flag will keep nodes balanced across zones for workloads that don't require topological scheduling.

CA doesn't work, but it used to work yesterday. Why?

Most likely it's due to a problem with the cluster. Steps to debug:

- Check if cluster autoscaler is up and running. In version 0.5 and later, it periodically publishes the kube-system/cluster-autoscaler-status config map. Check last update time annotation. It should be no more than 3 min (usually 10 sec old).
- Check in the above config map if cluster and node groups are in the healthy state. If not, check if there are unready nodes. If some nodes appear unready despite being Ready in the Node object, check `resourceUnready` count. If there are any nodes marked as `resourceUnready`, it is most likely a problem with the device driver failing to install a new resource (e.g. GPU). `resourceUnready` count is only available in CA version 1.24 and later.

If both the cluster and CA appear healthy:

- If you expect some nodes to be terminated, but they are not terminated for a long time, check I have a couple of nodes with low utilization, but they are not scaled down. Why? section.
- If you expect some nodes to be added to make space for pending pods, but they are not added for a long time, check I have a couple of pending pods, but there was no scale-up? section.
- If you have access to the control plane (previously referred to as master) machine, check Cluster Autoscaler logs in `/var/log/cluster-autoscaler.log`. Cluster Autoscaler logs a lot of useful information, including why it considers a pod unremovable or what was its scale-up plan.
- Check events added by CA to the pod object.
- Check events on the kube-system/cluster-autoscaler-status config map.
- If you see failed attempts to add nodes, check if you have sufficient quota on your cloud provider side. If VMs are created, but nodes fail to register, it may be a symptom of networking issues.

How can I check what is going on in CA ?

There are three options:

- Logs on the control plane (previously referred to as master) nodes, in `/var/log/cluster-autoscaler.log`.
- Cluster Autoscaler 0.5 and later publishes kube-system/cluster-autoscaler-status config map. To see it, run `kubectl get configmap cluster-autoscaler-status -n kube-system -o yaml`.
- Events:
  - on pods (particularly those that cannot be scheduled, or on underutilized nodes),
  - on nodes,
  - on kube-system/cluster-autoscaler-status config map.

How can I increase the information that the CA is logging?

By default, the Cluster Autoscaler will be conservative about the log messages that it emits. This is primarily due to performance degradations in scenarios where clusters have a large number of nodes (> 100). In these cases excess log messages will lead to the log storage filling more quickly, and in some cases (eg clusters with >1000 nodes) the processing performance of the Cluster Autoscaler can be impacted.

The `--v` flag controls how verbose the Cluster Autoscaler will be when running. In most cases using a value of `--v=0` or `--v=1` will be sufficient to monitor its activity. If you would like to have more information, especially about the scaling decisions made by the Cluster Autoscaler, then setting a value of `--v=4` is recommended. If you are debugging connection issues between the Cluster Autoscaler and the Kubernetes API server, or infrastructure endpoints, then setting a value of `--v=9` will show all the individual HTTP calls made. Be aware that using verbosity levels higher than `--v=1` will generate an increased amount of logs, prepare your deployments and storage accordingly.

How Can I change the log format that the CA outputs?

There are 2 log format options, `text` and `json`. By default (`text`), the Cluster Autoscaler will output logs in the klog native format.

```
I0823 17:15:11.472183   29944 main.go:569] Cluster Autoscaler 1.28.0-beta.0
```

Alternatively, adding the flag `--logging-format=json` changes the log output to json.

```
{"ts":1692825334994.433,"caller":"cluster-autoscaler/main.go:569","msg":"Cluster Autoscaler
```

What events are emitted by CA?

Whenever Cluster Autoscaler adds or removes nodes it will create events describing this action. It will also create events for some serious errors. Below is the non-exhaustive list of events emitted by CA (new events may be added in future):

- on kube-system/cluster-autoscaler-status config map:
  - ScaledUpGroup - CA increased the size of node group, gives both old and new group size.
  - ScaleDownEmpty - CA removed a node with no pods running on it (except system pods found on all nodes).
  - ScaleDown - CA decided to remove a node with some pods running on it. Event includes names of all pods that will be rescheduled to drain the node.
- on nodes:
  - ScaleDown - CA is scaling down the node. Multiple ScaleDown events may be recorded on the node, describing status of scale-down

operation.
  - ScaleDownFailed - CA tried to remove the node, but failed. The event includes error message.
- on pods:
  - TriggeredScaleUp - CA decided to scale up cluster to make place for this pod.
  - NotTriggerScaleUp - CA couldn't find node group that can be scaled up to make this pod schedulable.
  - ScaleDown - CA will try to evict this pod as part of draining the node.

Example event:

```
$ kubectl describe pods memory-reservation-73rl0 --namespace e2e-tests-autoscaling-kncnx
Name:   memory-reservation-73rl0

...

Events:
  FirstSeen LastSeen    Count   From            SubObjectPath   Type        Reason
  --------- --------    -----   ----            -------------   --------    ------
  1m        1m      1   cluster-autoscaler          Normal      TriggeredScaleUp    pod trig
```

My cluster is below minimum / above maximum number of nodes, but CA did not fix that! Why?

Cluster Autoscaler will not scale the cluster beyond these limits, but some other external factors could make this happen. Here are some common scenarios.

- Existing nodes were deleted from K8s and the cloud provider, which could cause the cluster fell below the minimum number of nodes.
- New nodes were added directly to the cloud provider, which could cause the cluster exceeded the maximum number of nodes.
- Cluster Autoscaler was turned on in the middle of the cluster lifecycle, and the initial number of nodes might beyond these limits.

By default, Cluster Autoscaler does not enforce the node group size. If your cluster is below the minimum number of nodes configured for CA, it will be scaled up *only* in presence of unschedulable pods. On the other hand, if your cluster is above the maximum number of nodes configured for CA, it will be scaled down *only* if it has unneeded nodes.

Starting with CA 1.26.0, a new flag `--enforce-node-group-min-size` was introduced to enforce the node group minimum size. For node groups with fewer nodes than the configuration, CA will scale them up to the minimum number of nodes. To enable this feature, please set it to `true` in the command.

What happens in scale-up when I have no more quota in the cloud provider?

Cluster Autoscaler will periodically try to increase the cluster and, once failed, move back to the previous size until the quota arrives or the scale-up-triggering pods are removed.

From version 0.6.2, Cluster Autoscaler backs off from scaling up a node group after failure. Depending on how long scale-ups have been failing, it may wait up to 30 minutes before next attempt.

Developer

What go version should be used to compile CA?

Cluster Autoscaler generally tries to use the same go version that is used by embedded Kubernetes code. For example CA 1.21 will use the same go version as Kubernetes 1.21. Only the officially used go version is supported and CA may not compile using other versions.

The source of truth for the used go version is builder/Dockerfile.

Warning: do NOT rely on go version specified in go.mod file. It is only meant to control go mod behavior and is not indicative of the go version actually used by CA. In particular go 1.17 changes go mod behavior in a way that is incompatible with existing Kubernetes tooling. Following Kubernetes example we have decided to pin version specified in go.mod to 1.16 for now (even though both Kubernetes and CA no longer compile using go 1.16).

How can I run e2e tests?

1. Set up environment and build e2e.go as described in the Kubernetes docs.
2. Set up the following env variables:

   ```
   export KUBE_AUTOSCALER_MIN_NODES=3
   export KUBE_AUTOSCALER_MAX_NODES=6
   export KUBE_ENABLE_CLUSTER_AUTOSCALER=true
   export KUBE_AUTOSCALER_ENABLE_SCALE_DOWN=true
   ```

   This is the minimum number of nodes required for all e2e tests to pass. The tests should also pass if you set higher maximum nodes limit.
3. Run `go run hack/e2e.go -- --verbose-commands --up` to bring up your cluster.
4. SSH to the control plane (previously referred to as master) node and edit `/etc/kubernetes/manifests/cluster-autoscaler.manifest` (you will need sudo for this).

- If you want to test your custom changes set `image` to point at your own CA image.
- Make sure `--scale-down-enabled` parameter in `command` is set to `true`.

5. Run CA tests with:

   ```
   go run hack/e2e.go -- --verbose-commands --test --test_args="--ginkgo.focus=\[Feature:C
   ```

   It will take >1 hour to run the full suite. You may want to redirect output to file, as there will be plenty of it.

   Test runner may be missing default credentials. On GCE they can be provided with:

   ```
   gcloud beta auth application-default login
   ```

A few tests are specific to GKE and will be skipped if you're running on a different provider.

Please open an issue if you find a failing or flaky test (a PR will be even more welcome).

How should I test my code before submitting PR?

This answer only applies to pull requests containing non-trivial code changes.

Unfortunately we can't automatically run e2e tests on every pull request yet, so for now we need to follow a few manual steps to test that PR doesn't break basic Cluster Autoscaler functionality. We don't require you to follow this whole process for trivial bugfixes or minor changes that don't affect main loop. Just use common sense to decide what is and what isn't required for your change.

To test your PR:

1. Run Cluster Autoscaler e2e tests if you can. We are running our e2e tests on GCE and we can't guarantee the tests are passing on every cloud provider.
2. If you can't run e2e we ask you to do a following manual test at the minimum, using Cluster-Autoscaler image containing your changes and using configuration required to activate them:

   i. Create a deployment. Scale it up, so that some pods don't fit onto existing nodes. Wait for new nodes to be added by Cluster Autoscaler and confirm all pods have been scheduled successfully.
   ii. Scale the deployment down to a single replica and confirm that the cluster scales down.

3. Run a manual test following the basic use case of your change. Confirm that nodes are added or removed as expected. Once again, we ask you to use common sense to decide what needs to be tested.

4. Describe your testing in PR description or in a separate comment on your PR (example: https://github.com/kubernetes/autoscaler/pull/74#issuecomment-302434795).

We are aware that this process is tedious and we will work to improve it.

How can I update CA dependencies (particularly k8s.io/kubernetes)?

Cluster Autoscaler imports a huge chunk of internal k8s code as it calls out to scheduler implementation. Therefore we want to keep set of libraries used in CA as close to one used by k8s, to avoid unexpected problems coming from version incompatibilities.

To sync the repositories' vendored k8s libraries, we have a script that takes a released version of k8s and updates the `replace` directives of each k8s sub-library. It can be used with custom kubernetes fork, by default it uses `git@github.com:kubernetes/kubernetes.git`.

Example execution looks like this:

```
./hack/update-deps.sh v1.30.2 v1.30.2 git@github.com:kubernetes/kubernetes.git
```

The first of two versions denotes k8s dependency of Cluster Autoscaler, the second one refers to the apis/ submodule.

If you need to update vendor to an unreleased commit of Kubernetes, you can use the breakglass script:

```
./hack/submodule-k8s.sh <k8s commit sha> git@github.com:kubernetes/kubernetes.git
```