Writing a Kubernetes Cluster Autoscaler Provider with externalgrpc

teaching the autoscaler to speak cloudscale

Marco De Luca · **mdnix.io**

$whoami

- Software/Infra Engineer at **VSHN**
- Kubernetes platforms, developer tooling, 3 AM pages
- Writes at **mdnix.io** - infrastructure, Linux, Kubernetes
- Off-hours: Alps with a Mamiya 7, or playing guitar

Agenda

- How the **Cluster Autoscaler** works
- **externalgrpc** + the SIG refactor proposal
- Building a provider
- **Live demo**
- **Gotchas** & takeaways
- Questions

The autoscaler knows *when*, not *how*

The **Cluster Autoscaler** decides:

- **When** to scale up - pending pods that won't fit anywhere
- **When** to scale down - nodes idle and re-schedulable
- **How** to make a VM - *not its job*

That's delegated to a **cloud provider** plugin.

Upstream in-tree providers: AWS · GCP · Azure · DigitalOcean · OVH · Hetzner · Equinix · …

*Next: how does the autoscaler actually decide **when** ?*

Every ten seconds, the autoscaler asks itself...

1. Any **unschedulable** pods?
2. Would they fit on a node from group X? *(scheduler simulator)*
3. If yes → `NodeGroupIncreaseSize(X, delta)`
4. Anyone **idle** for >10 min, pods reschedulable elsewhere?
5. If yes → `NodeGroupDeleteNodes(X, [nodes])`

The simulator is why the provider needs `NodeGroupTemplateNodeInfo` - more on that later.

Entry point: `StaticAutoscaler.RunOnce()` in `core/static_autoscaler.go`. Single-threaded. Leader-elected.

How CA actually picks a group

The 10s loop hides a lot. When CA decides to scale up, it picks **which** group through:

Component	Job
Estimator	How many nodes of group X to fit these pods? *(binpacking, default)*
Expander	Multiple groups fit - which one wins? *(random · most-pods · least-waste · price · priority · grpc)*
Upcoming nodes	Tracks scale-ups already in-flight so CA doesn't double-scale
DaemonSet accounting	Subtracts DS pods from each candidate's allocatable *before* binpacking
Backoff	Group with a recent failed create → skipped ~10min initial, doubles, caps ~1h *(stockout, quota)*

The provider sees **none of this**. We just get `IncreaseSize(group=X, delta=N)`.

But the *quality* of that decision rests on `TemplateNodeInfo` - wrong allocatable poisons estimator + expander every loop.

The simulator *is* the scheduler

CA doesn't reimplement scheduling. It **imports** `k8s.io/kubernetes/pkg/scheduler/framework` and runs the real Filter plugins:

- `NodeResourcesFit` · `NodeAffinity` · `NodeUnschedulable`
- `PodTopologySpread` · `InterPodAffinity` · `TaintToleration`
- `VolumeBinding` · `NodeVolumeLimits` · `NodePorts`

Run against a `ClusterSnapshot` - default impl is `DeltaClusterSnapshotStore`. Copy-on-write. **O(1)** `Fork` / `Revert`.

That's how scale-down can simulate removing *every* candidate node - hundreds of forks per loop - without deep-copying the cluster.

Same plugins kube-scheduler runs at the Filter extension point. Same verdicts.

What if your cloud isn't on the list?

That cloud-provider plugin we talked about on slide 4 - well, `kubernetes/autoscaler` is a **monolith**.

- Core CA logic
- 30+ cloud provider implementations
- Everyone vendored in-tree, one coupled release cycle

From SIG Autoscaling's own refactor proposal (autoscaler#9264, Feb 2026) - three bullets paraphrased:

> *Dependency hell · release coupling · many cloud providers not owned by any SIG-Autoscaling maintainer.*

Three options to add a new cloud:

Approach	Effort	Maintenance
In-tree PR to `k/autoscaler`	High	Rebase hell, ride upstream releases
Fork the autoscaler	High	Own CA forever (GKE, AKS, DataDog have done this)
`externalgrpc`	Low	Standalone binary, your own release cycle

Where it's heading: the split

autoscaler#9264 - opened **Feb 2026**, lazy-consensus period invoked in March. GKE-led proposal, buy-in from Microsoft, Red Hat, and SIG Autoscaling reviewers.

Repo	Role
`kubernetes-sigs/cluster-autoscaler` *(new)*	Pure library. Core logic + scale decisions + test providers + **externalgrpc**.
`cluster-autoscaler-provider-X` *(new, per cloud)*	Vendors core, implements `CloudProvider` interface. Karpenter pattern.
`kubernetes/autoscaler` *(current)*	Temporary back-compat home. Eventually shrinks.

The proposal **explicitly retains externalgrpc** in the new core alongside the test providers. Every other cloud moves out - as a library consumer, or as an externalgrpc client.

> *"Urge providers to maintain their own projects in repositories that they are 100% responsible for." - Jack Francis, Azure provider maintainer (Microsoft)*

Two integration models in the new world: library (compile-time, Karpenter pattern) or **externalgrpc** (wire-time, language-agnostic). For now, externalgrpc is the only realistic path for an outsider - and post-refactor, it stays as the process-isolated escape hatch.

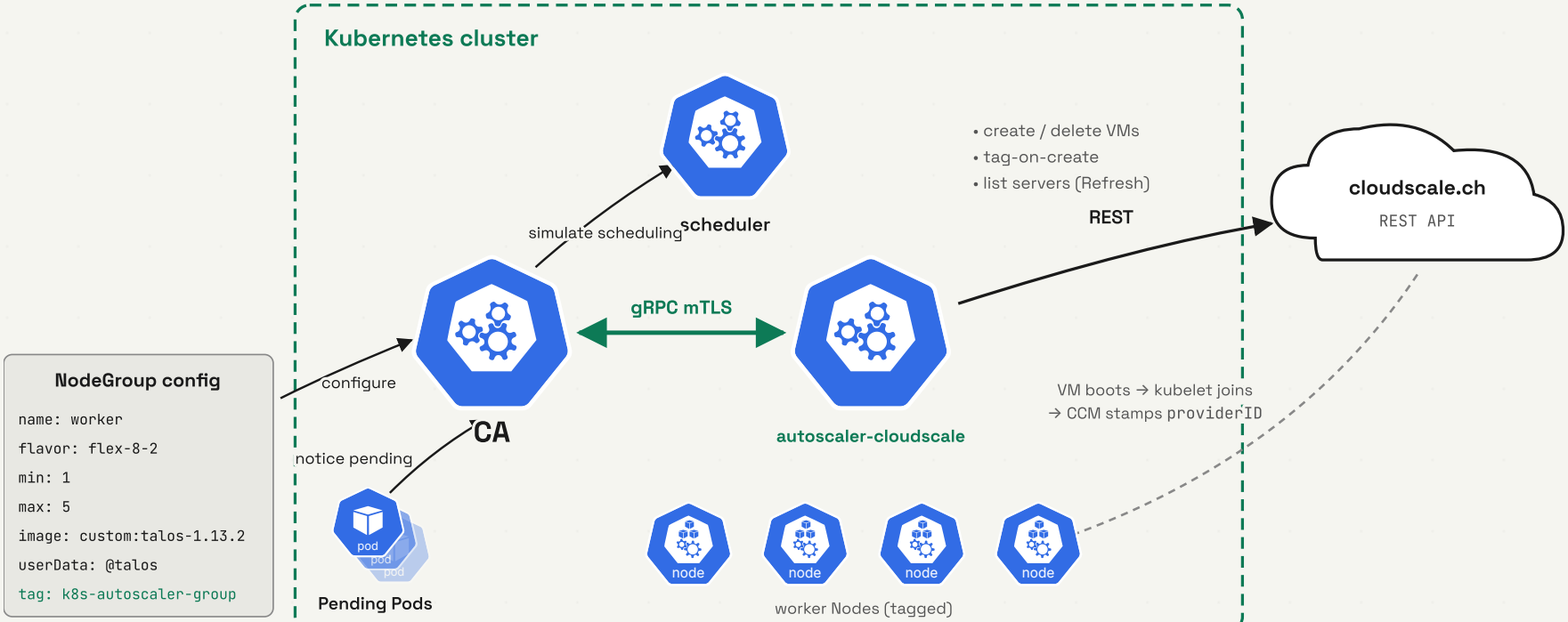
Why I went with externalgrpc

- No upstream provider for this cloud
- No official channel as an outsider to either project
- **In-tree**: PR upstream, ride their release train, hope SIG reviews changes
- **Fork**: maintain my own CA build forever
- `externalgrpc`: ship a binary, own my release, mTLS, language-agnostic

It was the only path that didn't require me to either become a CA maintainer or fork a 100k-line monolith.

Turns out externalgrpc is one of the two paths SIG Autoscaling keeps in the proposed split.

Architecture, at a glance



The contract: six RPCs and a tag

RPC	What it does
`Refresh`	Sync target sizes with the cloud
`NodeGroups`	List configured groups
`NodeGroupForNode`	Map a `v1.Node` → group
`NodeGroupTemplateNodeInfo`	What would a new node look like?
`NodeGroupIncreaseSize`	Create N servers in a group
`NodeGroupDeleteNodes`	Delete specific servers

The proto has **15 RPCs**. Six do the heavy work, three more are bookkeeping (`NodeGroupNodes`, `TargetSize`, `DecreaseTargetSize`), the rest are stubs returning `codes.Unimplemented` - but they **must exist**.

Group membership: tag `k8s-autoscaler-group=<name>`.

Refresh: trust the cloud, not yourself

```
servers, err := c.api.Servers.List(ctx,
    cloudscale.WithTagFilter(cloudscale.TagMap{
        "k8s-cluster": c.clusterTag,
    }))

byUUID := make(map[string]*cloudscale.Server, len(servers))
for i := range servers {
    byUUID[servers[i].UUID] = &servers[i]
}

c.mu.Lock()
c.serversByUUID = byUUID
c.mu.Unlock()
```

One API call per loop. Tag filter scopes us to *this* cluster.

> *The cloud is the source of truth. Drift happens - manual deletes, failed creates, partial outages. Reconcile every Refresh.*

Lock-ordering gotcha (caught in tests): two mutexes - cache (`c.mu`) and per-node-group (`ng.mu`). Must be acquired **cache first**, else `Refresh` vs `DecreaseTargetSize` deadlocks under load.

Scale-from-zero needs a lie

How does the autoscaler decide a pod fits on a node group that currently has **zero** nodes?

It can't ask the kubelet - there isn't one.

So it asks the provider for a **fake** `v1.Node` and runs the scheduler simulator against it.

`NodeGroupTemplateNodeInfo` is that lie.

Get it wrong → autoscaler thinks pods don't fit → **scale-from-zero silently never triggers**.

...and that's only *one* of the things this RPC powers. The other one's next.

TemplateNodeInfo's other job: phantom nodes

After `IncreaseSize`, VMs take 60–120s to register. CA's next loop fires in 10s. **It can't wait.**

So CA **injects fake** `v1.Node` **objects** built from your `TemplateNodeInfo` into the next snapshot. Annotated `cluster-autoscaler.k8s.io/upcoming-node` - *these are the "upcoming nodes" from slide 6.*

→ Simulator sees pending pods as already-schedulable on phantoms → No duplicate scale-up while real VMs boot → When real nodes register, phantoms vanish

The implication: your `TemplateNodeInfo` feeds **two** things:

1. Scale-from-zero - previous slide
2. Phantom-node injection - this one

Wrong template → CA either **double-provisions** *or* **stalls**. Both silent.

Lying convincingly

```
node := &v1.Node{
    ObjectMeta: metav1.ObjectMeta{Labels: nodeGroup.Labels},
    Spec:       v1.NodeSpec{Taints: nodeGroup.Taints},
    Status: v1.NodeStatus{
        Capacity: v1.ResourceList{
            v1.ResourceCPU:              flavor.VCPUs,         // 8 * 1000m
            v1.ResourceMemory:           flavor.Memory,        // 16 GiB
            v1.ResourceEphemeralStorage: nodeGroup.VolumeSize, // 100 GiB
            v1.ResourcePods:             resource.MustParse("110"),
        },
        Allocatable: subtractReserved(capacity),
    },
}
```

```
Allocatable = Capacity - systemReserved - evictionHard
CPU:    vcpus*1000m - 50m
Memory: mem - 384Mi (sys) - 100Mi (eviction)
Disk:   gb*1Gi - 256Mi (sys) - 10% (eviction)
```

Numbers must match real **kubelet flags**. Mismatch → real node smaller than simulated → pod stays Pending. **Silent.**

Lying convincingly - the gotchas

Two failure modes hidden in that one struct:

1. `ResourcePods` **defaults to 0.** Scale-from-zero just… never triggered. No error, no log line. CA's scheduler sees `pods: 0`, decides nothing fits, never calls `IncreaseSize`. Found by grepping CA source for `ResourcePods`.
2. **The response is proto-marshaled** `v1.Node` **bytes - not JSON.** Figured it out by reading CA's externalgrpc client code. *(CA 1.35 added a doc comment - #8746)*

Remember: the same numbers feed phantom injection. **One mistake, two silent failures.**

SIG Autoscaling is redesigning this API - autoscaler#7799. Until then: do the math, marshal the bytes.

IncreaseSize: what if half your boots fail?

Naive: `for i < n { create() }`, bail on first error. Reality: cloud APIs fail partially.

```
Request: IncreaseSize(delta=5)

targetSize: 3 → 8                  (optimistic bump, under lock)
create 5 servers concurrently      (semaphore: max 10 in-flight)
  ✓ ✓ ✓ × ×                         (2 fail)
targetSize: 8 → 6                  (rollback by failures)
```

Why bump first? If we bumped after, CA might call `IncreaseSize` again during the slow API calls.

Mirror image of scale-down: there we delete first, **decrement on success**, clamp at `minSize`.

VM created but never joins? Provider doesn't watch boot status - that's CA's job. CA waits `MaxNodeProvisionDuration` (default ~15 min), then calls `NodeGroupDeleteNodes` with the zombie's providerID. We delete. Cloud is clean.

Subtle gotcha: tag the server **inside the create request**, not in a follow-up `PATCH`. If the process crashes between create and tag, you orphan a VM that no future `Refresh` recognizes - it has no `k8s-autoscaler-group` tag, so we don't see it, but it's billing you. Most cloud APIs let you tag at creation. Use it.

After create: bootstrapping the Node

Provider's job ends at "VM created." Then:

1. cloud boots VM with our `userData` *(opaque bytes - we pass-through)*
2. OS bootstraps itself - **not our problem**
3. `kubelet` starts, joins the API server, Node appears
4. CCM stamps `spec.providerID: cloudscale://<uuid>`
5. CA bounds the wait: `MaxNodeProvisionDuration` *(default 15 min)*

`userData` shape depends on the OS - Talos machineconfig (what I run), cloud-init for Ubuntu/Debian/Flatcar, Ignition for Fedora CoreOS.

NodeGroupForNode: whose node is this?

Before we can scale down, the CA asks us: which group does this `v1.Node` belong to?

The CCM stamps every Node:

```
spec:
  providerID: cloudscale://abc-123-def
```

The chain:

`providerID` → strip prefix → server UUID → **O(1) cache lookup** → read `k8s-autoscaler-group` tag → node group

```
uuid, ok := strings.CutPrefix(providerID, "cloudscale://")
server := c.serversByUUID[uuid]   // O(1) under RLock
group  := server.Tags["k8s-autoscaler-group"]
```

No CCM, no mapping. Hard prerequisite.

Design doc warns verbatim: *"extensively used by CA and can cause performance degradations on large clusters."* 500 nodes × 6 loops/min = **3000 RPCs/min** if you don't cache. The map above is that cache.

Gotcha: the `providerID` format isn't formalized anywhere upstream. Every CCM picks its own - `aws:///us-east-1a/i-abc`, `gce://project/zone/instance`, `cloudscale://<uuid>`. You match what your CCM emits, byte-for-byte. **Read the CCM source**; the format isn't a spec.

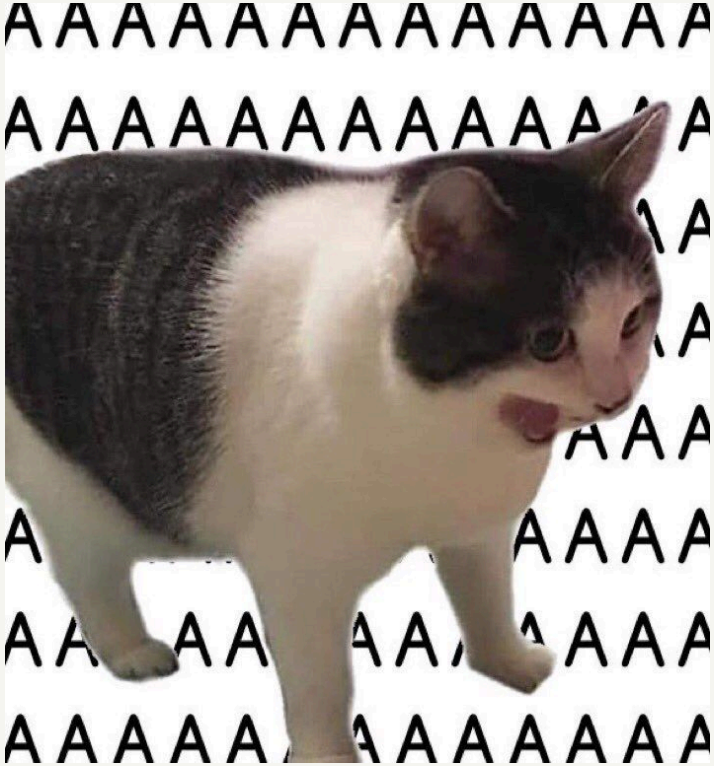
Building one for your cloud

If you're sitting here with a cloud that's not in the upstream tree:

1. **Implement six RPCs** - Refresh, NodeGroups, NodeGroupForNode, TemplateNodeInfo, IncreaseSize, DeleteNodes. Stub the other nine as `Unimplemented`.
2. **Pick one tag** for group membership.
3. **Make sure your cloud has a CCM** - `providerID` is the bridge. No CCM, no scale-down.
4. **Decide your `userData` shape** - Talos, cloud-init, Ignition. Bytes through. You own the bootstrap.
5. **Get `TemplateNodeInfo` right** - match kubelet `--system-reserved` + `--eviction-hard`. Set `ResourcePods` explicitly. Remember: this RPC feeds both scale-from-zero *and* phantom injection.
6. **mTLS the gRPC channel** - cert-manager + self-signed Issuer chain. ~5 min of YAML.

The CA does the orchestration. You write under 2,000 lines of cloud-specific glue.

The dreaded live demo



Links

- **Code** - https://github.com/kubeterm-sh/autoscaler-cloudscale
- **Blog** - https://mdnix.io
- **SIG refactor** - autoscaler#9264 (the library split)
- **TemplateNodeInfo redesign** - autoscaler#7799

Questions?

Thanks for listening!