How the OOM-Killer Deleted my Namespace (and Other Kubernetes Tales)

Laurent Bernaille

Datadog

Over 400 integrations
Over 1,500 employees
Over 12,000 customers
Runs on millions of hosts
Trillions of data points per day

10000s hosts in our infra
10s of k8s clusters with 100-4000 nodes
Multi-cloud
Very fast growth

Context

- We have shared a few stories already
  - Kubernetes the Very Hard Way
  - 10 Ways to Shoot Yourself in the Foot with Kubernetes
  - Kubernetes DNS horror stories

- The scale of our Kubernetes infra keeps growing
  - New scalability issues
  - But also, much more complex problems

How The OOM-killer Deleted my Namespace

Symptoms

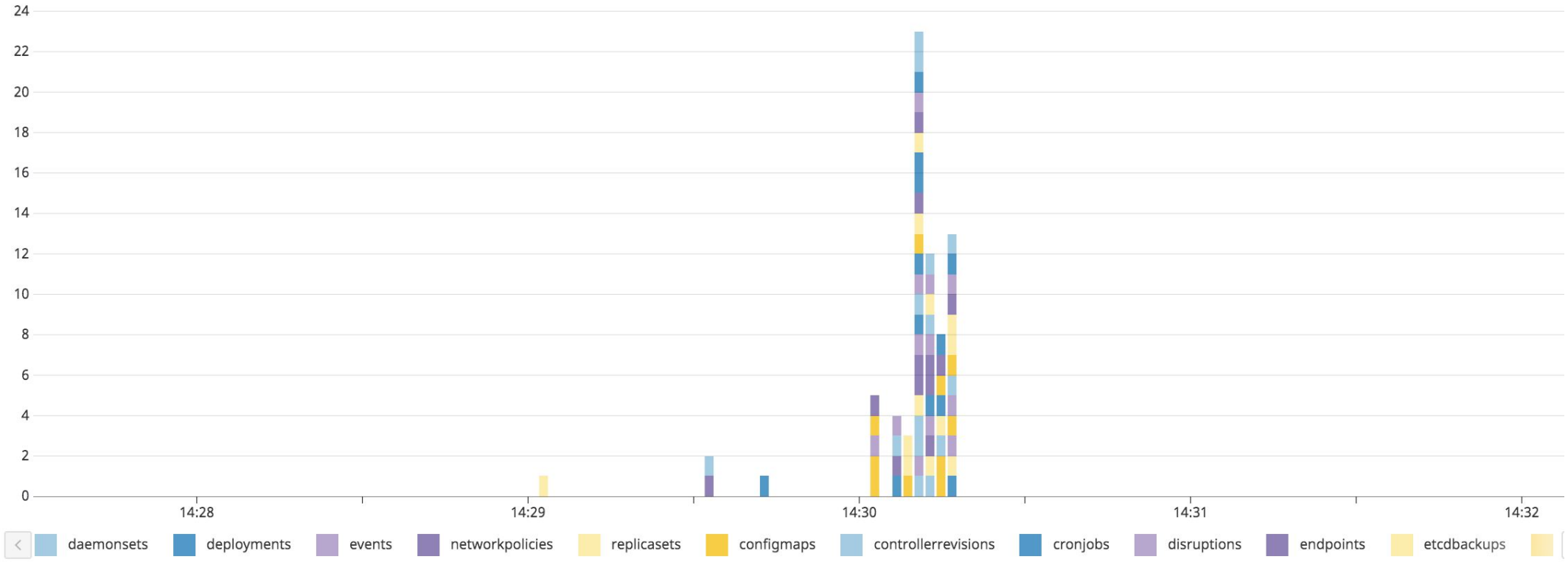
"My namespace and everything in it are **completely gone**"

Investigation

KubeCon CloudNativeCon North America 2020 Virtual

Feb 17 14:30: Namespace and workloads deleted

DeleteCollection calls by resource (audit logs)



Investigation

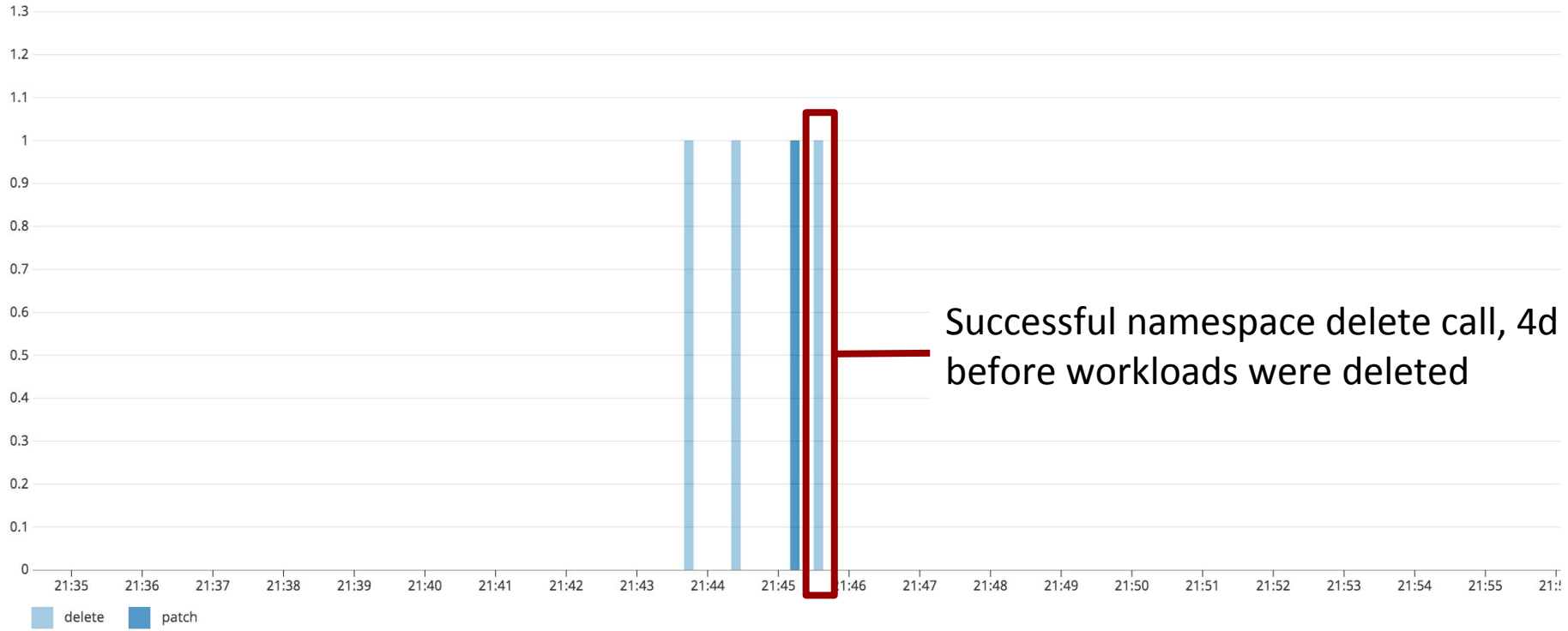
Audit logs

- Delete calls for all resources in the namespace
- **No delete call for the namespace resource**

Why was the namespace deleted??

Deletion call, 4d before

Audit logs for the namespace

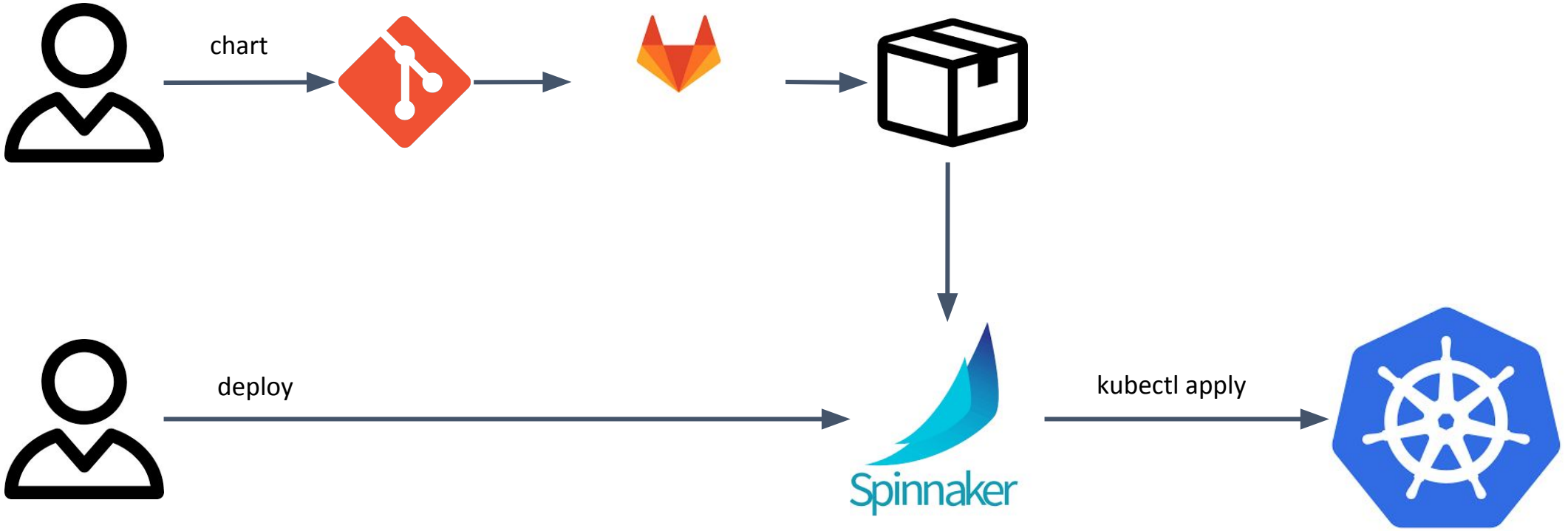


Feb 13th (resource deletion happened on Feb 17th)

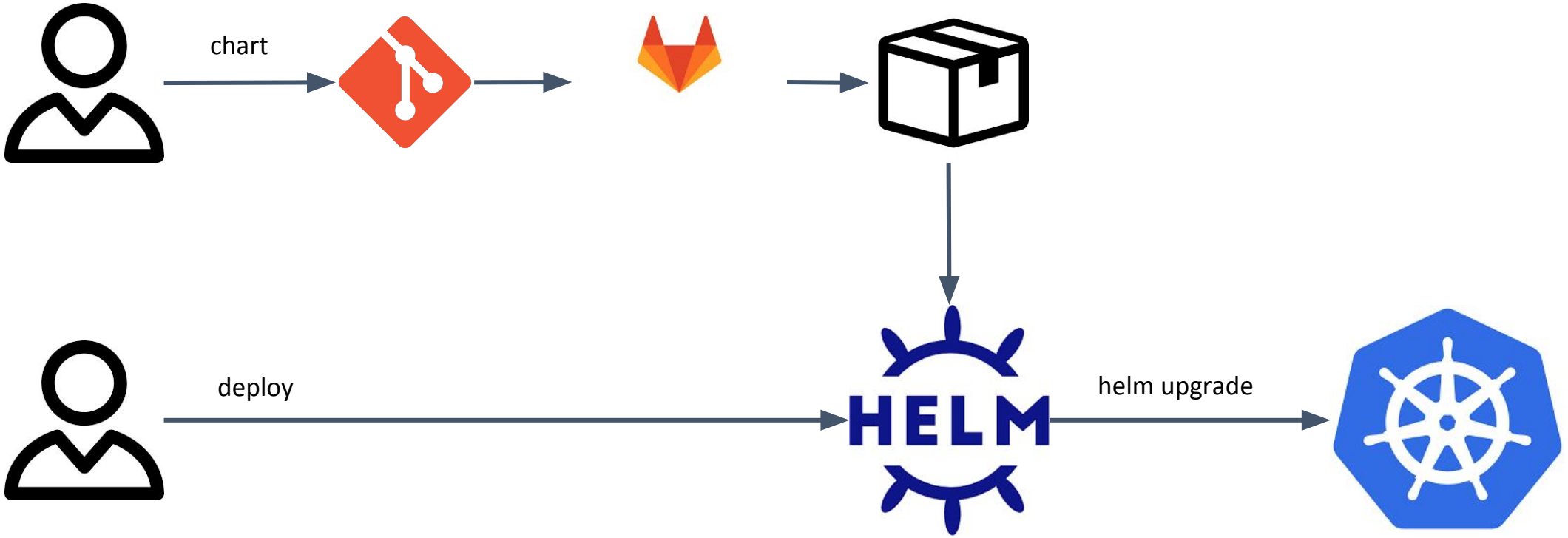
Engineer did not delete the namespace

But engineer was migrating to new deployment method

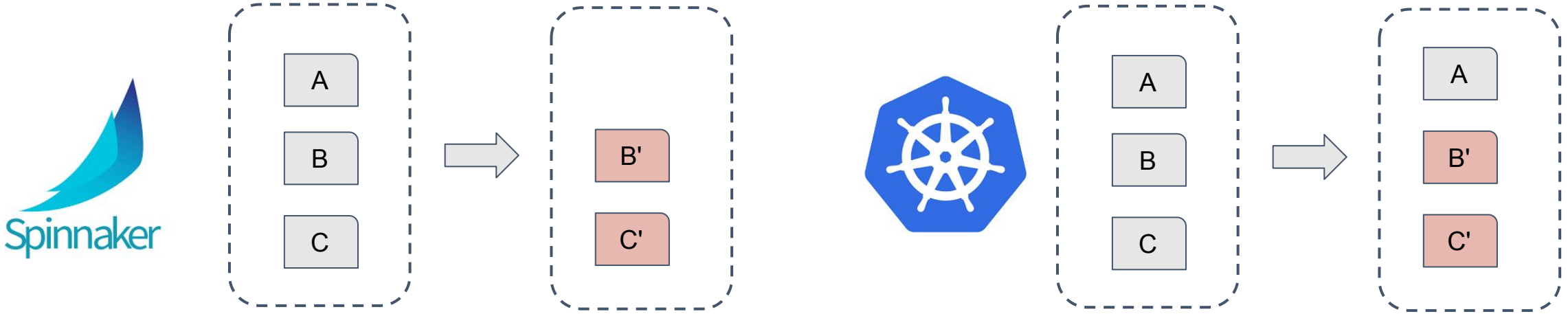
Spinnaker deploys (v1)



Helm 3 deploys (v2)

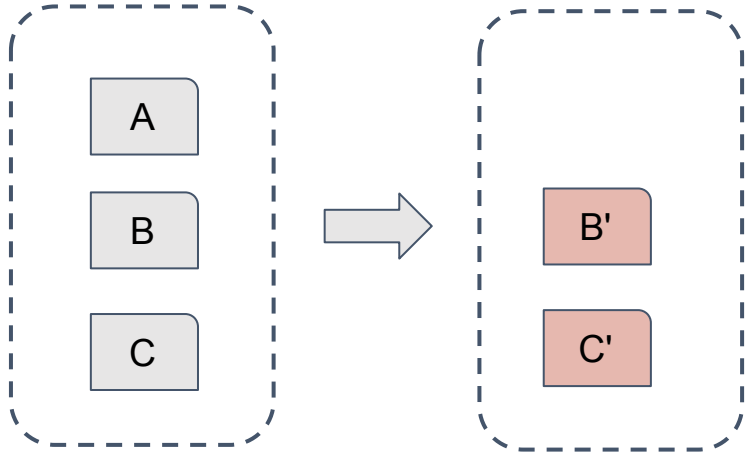


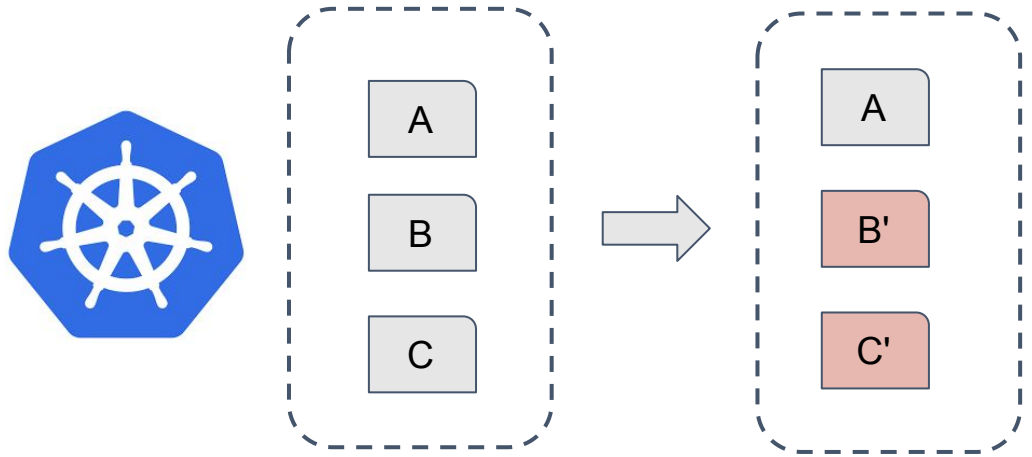
Big difference



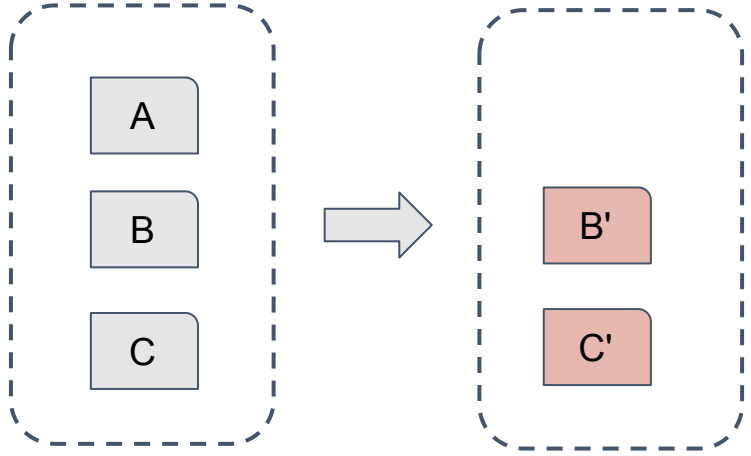
Big difference

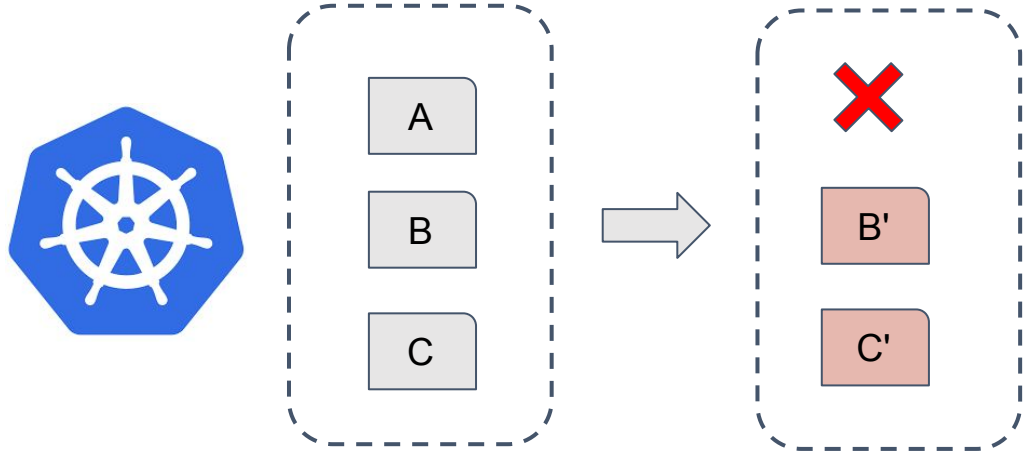
Spinnaker



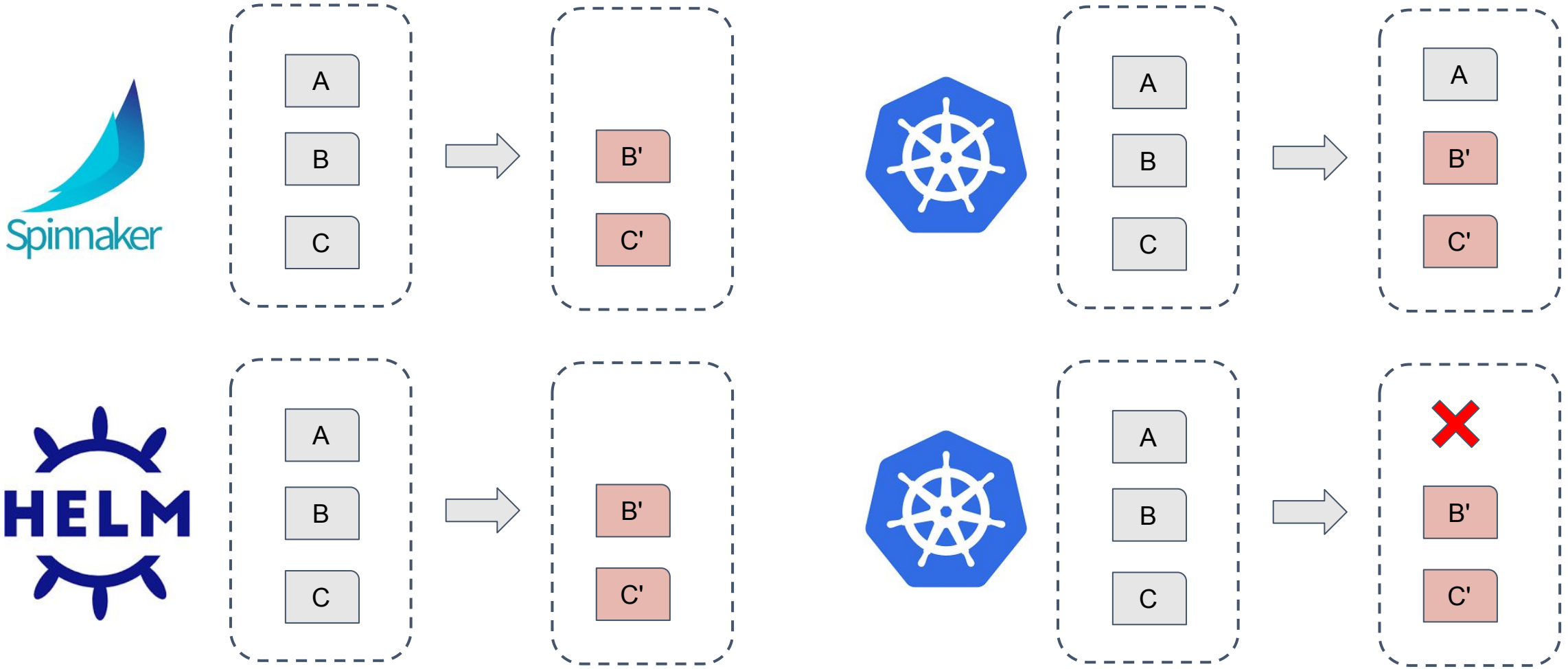


HELM





Big difference



v2: removing a resource from the chart deletes it from the cluster

What happened?

1- Chart refactoring: move namespace out of it

2- Deployment: deletes namespace

But why did it take 4 days ?

Namespace Controller


```
497  // deleteAllContent will use the dynamic client to delete each resource identified in groupVersionResources.
498  // It returns an estimate of the time remaining before the remaining resources are deleted.
499  // If estimate > 0, not all resources are guaranteed to be gone.
500  func (d *namespacedResourcesDeleter) deleteAllContent(
501          namespace string, namespaceDeletedAt metav1.Time) (int64, error) {
502          estimate := int64(0)
503          glog.V(4).Infof("namespace controller - deleteAllContent - namespace: %s", namespace)
504          resources, err := d.discoverResourcesFn()
505          if err != nil {
506                  return estimate, err
507          }
508          // TODO(sttts): get rid of opCache and pass the verbs (especially "deletecollection") down into the deleter
509          deletableResources := discovery.FilteredBy(discovery.SupportsAllVerbs{Verbs: []string{"delete"}}, resources)
510          groupVersionResources, err := discovery.GroupVersionResources(deletableResources)
511          if err != nil {
512                  return estimate, err
513          }
514          var errs []error
515          for gvr := range groupVersionResources {
516                  gvrEstimate, err := d.deleteAllContentForGroupVersionResource(gvr, namespace, namespaceDeletedAt)
517                  if err != nil {
518                          // If there is an error, hold on to it but proceed with all the remaining
519                          // groupVersionResources.
520                          errs = append(errs, err)
521                  }
522                  if gvrEstimate > estimate {
523                          estimate = gvrEstimate
524                  }
525          }
```

Discover all API resources

Delete all resources in namespace

Namespace Controller


```
deletableResources := discovery.FilteredBy(discovery.SupportsAllVerbs{Verbs: []string{"delete"}}, resources)
groupVersionResources, err := discovery.GroupVersionResources(deletableResources)
if err != nil {
        return estimate, err
}
```

=> If discovery fails, namespace content is not deleted

Kubernetes 1.14+: Delete as many resources as possible
Kubernetes 1.16+: Add Conditions to namespaces to surface errors

Namespace Controller logs

Controller-manager contained many occurrences of

```
unable to retrieve the complete list of server APIs: metrics.k8s.io/v1beta1:
the server is currently unable to handle the request
```

No context so hard to link with the incident but

- This is the error from the Discovery call
- We have a likely cause: **metrics.k8s.io/v1beta1**

metrics-server

- Additional controller providing node/pod metrics
- Registers an ApiService
  - Additional Kubernetes APIs
  - Managed by the external controller

```yaml
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  name: v1beta1.metrics.k8s.io
spec:
  group: metrics.k8s.io
  version: v1beta1

  service:
    name: metrics-server
    namespace: kube-system
```

Additional APIs (group and version)

Where should the apiserver proxy calls for this APIs

Events so far

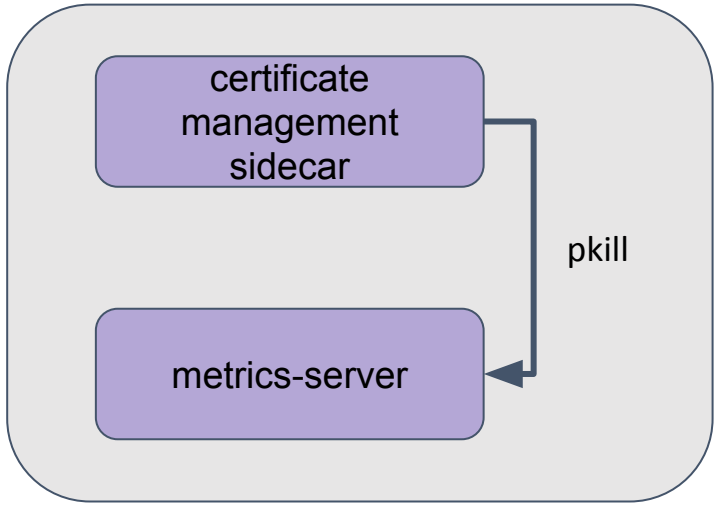
- Feb 13th: namespace deletion call
- Feb 13-17th
  - Namespace controller can't discover metrics APIs
  - Namespace content is not deleted
  - At each sync, controller retries and fails
- Feb 17th
  - ?
  - Namespace controller is unblocked
  - Everything in the namespace is deleted

Events so far

- Feb 13th: error leads to namespace deletion call
- Feb 13-17th
  - Namespace controller can't discover metrics APIs
  - Namespace content is not deleted
  - At each sync, controller retries and fails
- Feb 17th
  - OOM-killer kills metrics-server and it is restarted
  - Metrics-server is now working fine
  - Namespace controller is unblocked
  - Everything in the namespace is deleted

Why was metrics-server failing?

Metrics-server setup



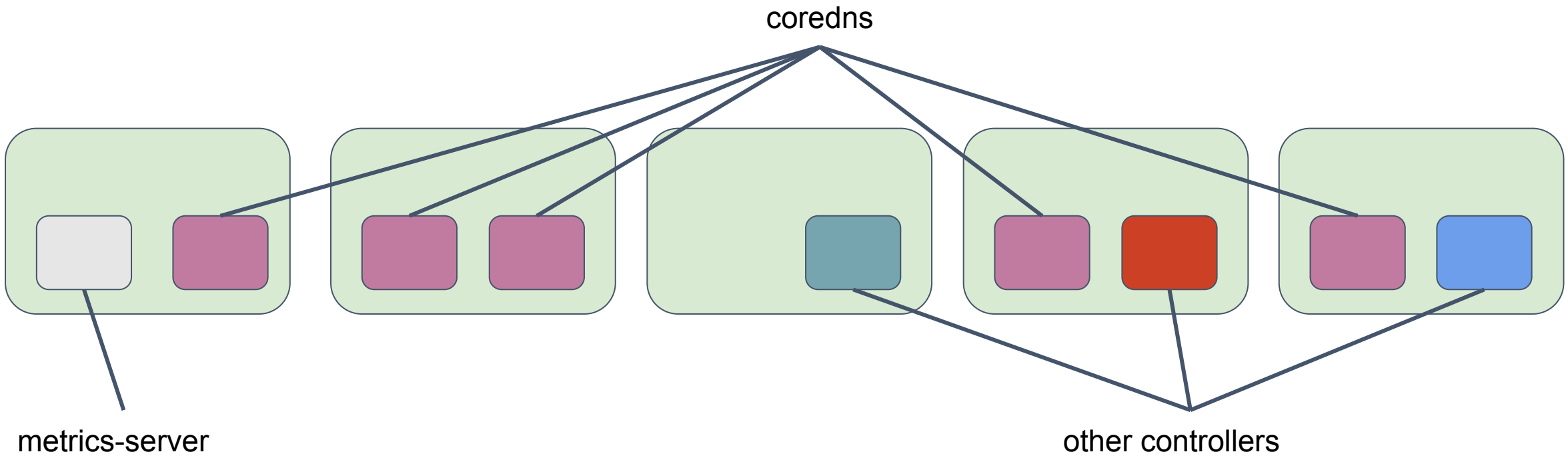
Pod with two containers

- metrics-server
- sidecar: rotate server certificate, and signal metrics-server to reload

Requires shared pid namespace

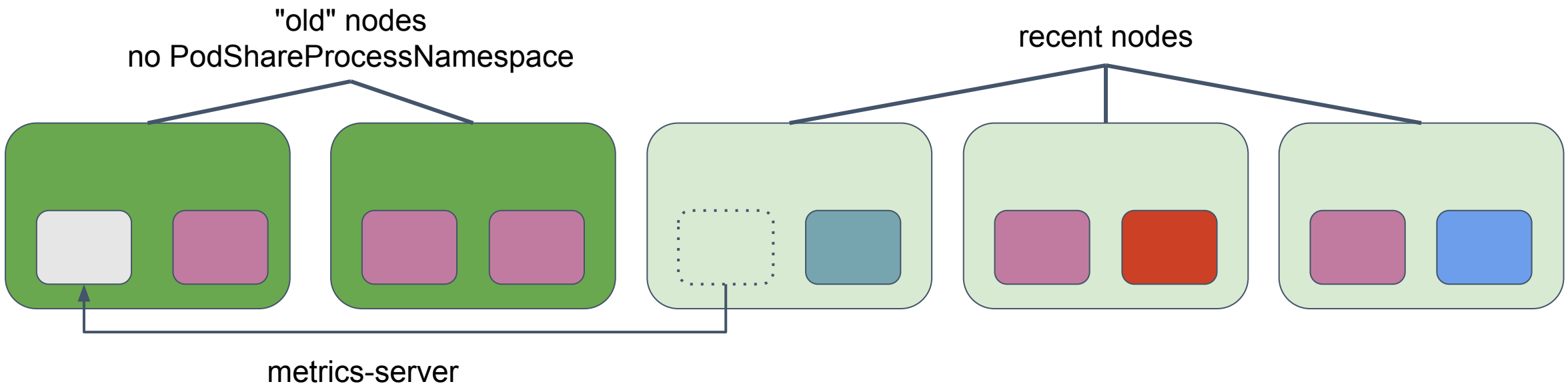
- shareProcessNamespace (pod spec)
- PodShareProcessNamespace (feature gate on Kubelet and Apiserver)

Metrics-server deployment



Deployed on a pool of nodes dedicated to core controllers ("system")

Metrics-server deployment



Subset of nodes using an old image without PodShareProcessNamespace

- **metrics-server: fine until it was rescheduled on an old node**
- **At this point, server certificate is no longer reloaded**

Full chain of events

- **Before Feb 13th**
  - **metrics-server scheduled on a "bad" node**
  - **controller discovery calls fail**
- Feb 13th: error leads to namespace deletion call
- Feb 13-17th
  - Namespace controller can't discover metrics APIs
  - Namespace content is not deleted
  - At each sync, controller retries and fails
- Feb 17th
  - OOM-killer kills metrics-server and it is restarted
  - **Metrics-server is now working fine (certificate up to date)**
  - Namespace controller is unblocked
  - Everything in the namespace is deleted

Key take-away

- Apiservice extensions are great but can impact your cluster
  - If the API is not used, they can be down for days without impact
  - But some controllers need to discover all apis and will fail

- In addition, Apiservice extensions security is hard
  - If you are curious, go and see

> PKI the Wrong Way: Simple TLS Mistakes and Surprising Consequences - Tabitha Sable, Datadog

New node image does not work and it's Weird...

Context

- We build images for our Kubernetes nodes
- A small change pushed
- Nodes become NotReady after a few days
- Change completely unrelated to kubelet / runtime
- We can't promote to Prod

Investigation

Node description

```
Conditions:
  Type             Status  Reason                       Message
  ----             ------  ------                       -------
  OutOfDisk        False   KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure   False   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure     False   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            False   KubeletNotReady              container runtime is down
```

Runtime is down?

KubeCon CloudNativeCon North America 2020 Virtual

Let's look at containerd

```
$ crictl pods
POD ID          STATE   NAME                              NAMESPACE
681bb3aec7fba   Ready   local-volume-provisioner-jtwzf    local-volume-provisioner
53f122f351da2   Ready   datadog-agent-qctt2-8ljgp         datadog-agent
c1edb4f19713c   Ready   localusers-z26n9                  prod-platform
01cf5cafd6bc9   Ready   node-monitoring-ws2xf             node-monitoring
39f01bdaade86   Ready   kube2iam-zzdt6                    kube2iam
3fcf520680a63   Ready   kube-proxy-sr7tn                  kube-system
5ecd0966634f1   Ready   node-local-dns-m2zbp              coredns
```

Looks completely fine

Kubelet logs

Something is definitely wrong

```
remote_runtime.go:434] Status from runtime service failed: rpc error: code = DeadlineExceeded desc =
context deadline exceeded

kubelet.go:2130] Container runtime sanity check failed: rpc error: code = DeadlineExceeded desc =
context deadline exceeded

kubelet.go:1803] skipping pod synchronization - [container runtime is down]
```

"Status from runtime service failed"

- ➢ Is there a CRI call for this?

```
// Status returns the status of the runtime.
rpc Status(StatusRequest) returns (StatusResponse) {}
```

Testing with crictl

- crictl has no "status" subcommand
- but "crictl info" looks close, let's check

```
// Info sends a StatusRequest to the server, and parses the returned StatusResponse.
func Info(cliContext *cli.Context, client pb.RuntimeServiceClient) error {
        request := &pb.StatusRequest{Verbose: !cliContext.Bool("quiet")}
        logrus.Debugf("StatusRequest: %v", request)
        r, err := client.Status(context.Background(), request)
```

Crictl info


```
$ crictl info
^C
```

- Hangs indefinitely
- But what does Status do?
- Almost nothing, except this

```
44          // Check the status of the cni initialization
45          if err := c.netPlugin.Status(); err != nil {
46                  networkCondition.Status = false
47                  networkCondition.Reason = networkNotReadyReason
48                  networkCondition.Message = fmt.Sprintf("Network plugin returns error: %v", err)
49          }
```

CNI status


```
124  // Status returns the status of CNI initialization.
125  func (c *libcni) Status() error {
126          c.RLock()
127          defer c.RUnlock()
128          if len(c.networks) < c.networkCount {
129                  return ErrCNINotInitialized
130          }
131          return nil
132  }
```

- Really not much here right?
- But we have calls that hang and a lock
- Could this be related?

Containerd goroutine dump

Blocked goroutines?

```
containerd[737]: goroutine 107298383 [semacquire, 2 minutes]:
containerd[737]: goroutine 107290532 [semacquire, 4 minutes]:
containerd[737]: goroutine 107282673 [semacquire, 6 minutes]:
containerd[737]: goroutine 107274360 [semacquire, 8 minutes]:
containerd[737]: goroutine 107266554 [semacquire, 10 minutes]:
```

The runtime error from the kubelet also happens every 2mn

Goroutine details match exactly the codepath we looked at

```
containerd[737]: goroutine 107298383 [semacquire, 2 minutes]:
...
containerd[737]: .../containerd/vendor/github.com/containerd/go-cni.(*libcni).Status()
containerd[737]:     .../containerd/vendor/github.com/containerd/go-cni/cni.go:126
containerd[737]: .../containerd/vendor/github.com/containerd/cri/pkg/server.(*criService).Status(...)
containerd[737]:     .../containerd/containerd/vendor/github.com/containerd/cri/pkg/server/status.go:45
```

Seems CNI related

Let's bypass containerd/kubelet

```
$ ip netns add debug
$ CNI_PATH=/opt/cni/bin cnitool add pod-network /var/run/netns/debug
```

Works great and we even have connectivity

```
$ ip netns exec debug ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=111 time=2.41 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=111 time=1.70 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=111 time=1.76 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=111 time=1.72 ms
```

What about Delete?


```
$ CNI_PATH=/opt/cni/bin cnitool del pod-network /var/run/netns/debug
^C
```

- hangs indefinitely
- and we can even find the original process still running

```
ps aux | grep cni
root       367  0.0  0.1 410280  8004 ?        Sl   15:37   0:00
/opt/cni/bin/cni-ipvlan-vpc-k8s-unnumbered-ptp
```

CNI plugin

- On this cluster we use the Lyft CNI plugin
- The delete calls fails for cni-ipvlan-vpc-k8s-unnumbered-ptp
- After a few tests, we tracked the problem to this call

```
vethPeerIndex, _ = netlink.VethPeerIndex(&netlink.Veth{LinkAttrs: *vethIface.Attrs()})
```

- Weird, this is in the netlink library!

The root cause

link, veth: fix stack corruption from retrieving peer index #498

Merged **vishvananda** merged 2 commits into `vishvananda:master` from `borkmann:pr/netlink_vpeer_stack_corruption` on Nov 13, 2019

Conversation 2 | Commits 2 | Checks 0 | Files changed 2

borkmann commented on Nov 6, 2019 — Contributor

For 4.20 and newer kernels VethPeerIndex() causes a stack corruption as the kernel is copying more data to golang user space than originally expected.

See commits for detailed description.

Many thanks to Daniel Borkmann!

Summary

- Packer configured to use the latest Ubuntu 18.04
- The default kernel changed from 4.15 to 5.4
- Netlink library had a bug for kernels 4.20+
- Bug only happened on pod deletion

Status

- Containerd
  - Status calls trigger config file reload, which acquires a RWLock on the config mutex
  - CNI Add/Del calls acquire a RLock on the mutex
  - If CNI Add/Del hangs, Status calls block
  - Does not impact containerd 1.4+ (separate fsnotify.Watcher to reload config)
- cni-ipvlan-vpc-k8s ("Lyft" CNI plugin)
  - Bug in Netlink library with kernel 4.20+
  - Does not impact cni-ipvlan-vpc-k8s 0.6.2+ (different function used)

Key take-away

Nodes are not abstract compute resources

- Kernel
- Distribution
- Hardware

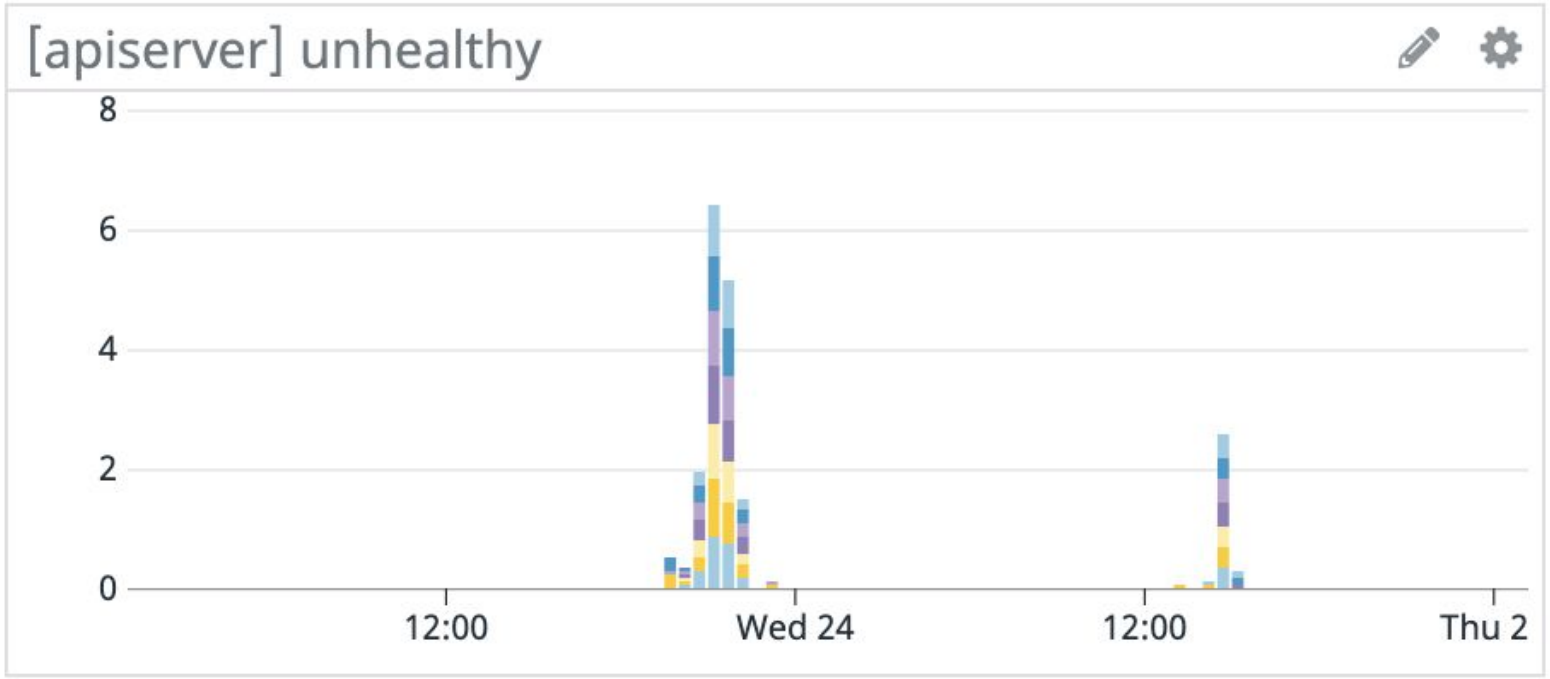
Error messages can be misleading

- For CNI errors, runtime usually reports a clear error
- Here we had nothing but a timeout

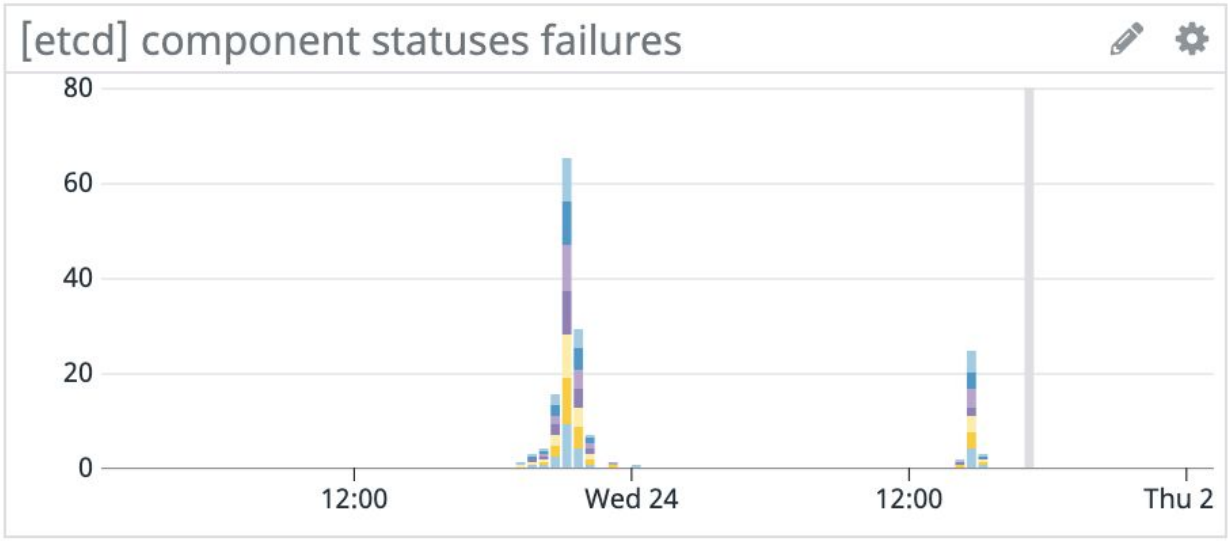
How a single app can overwhelm the control plane

Context

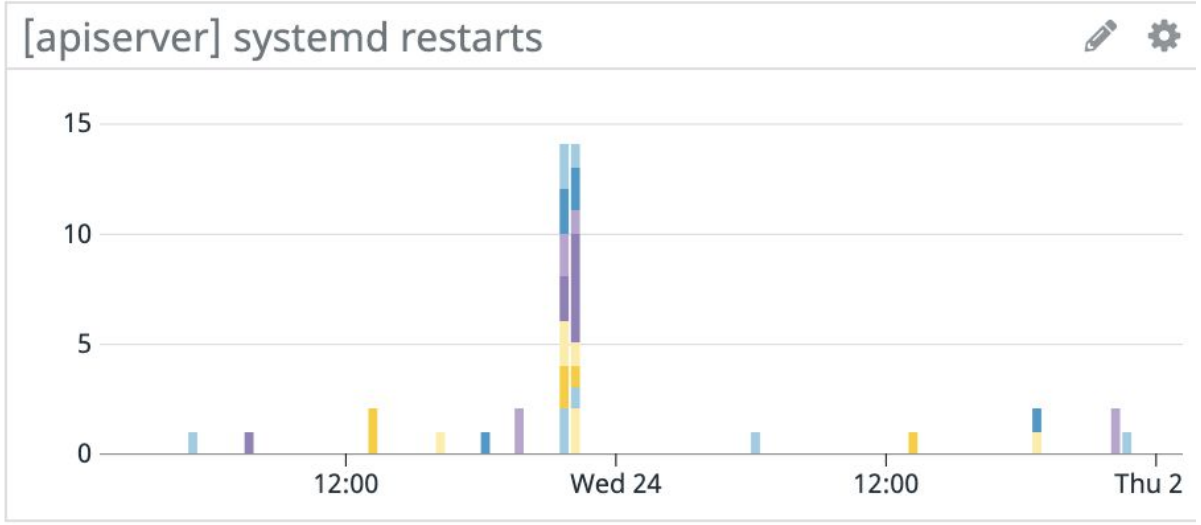
- Users report connectivity issue to a cluster
- Apiservers are not doing well



What's with the Apiservers?

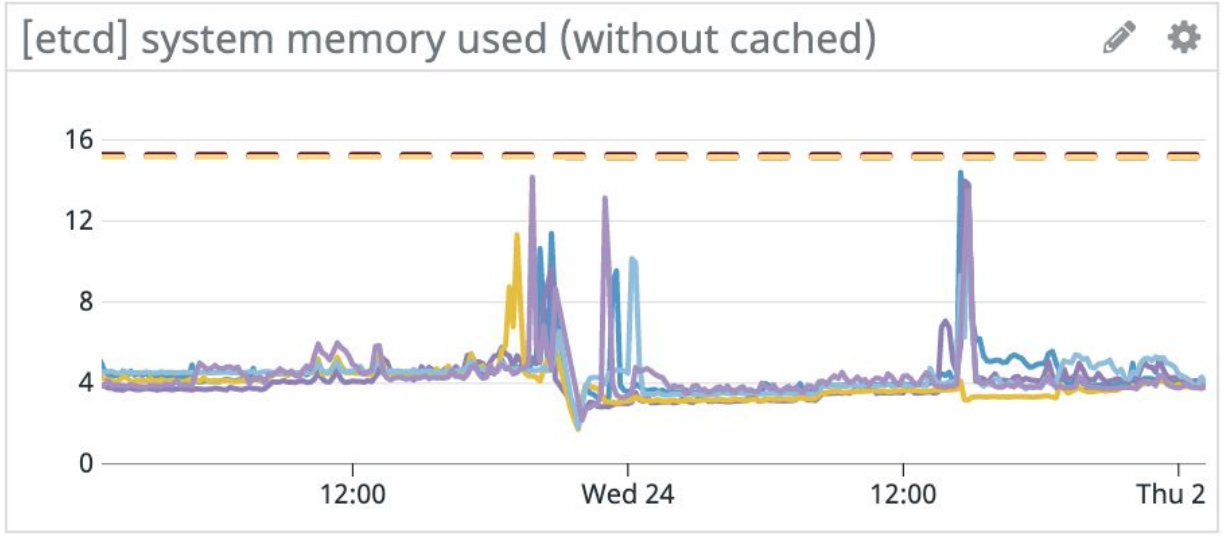


Apiservers can't reach etcd

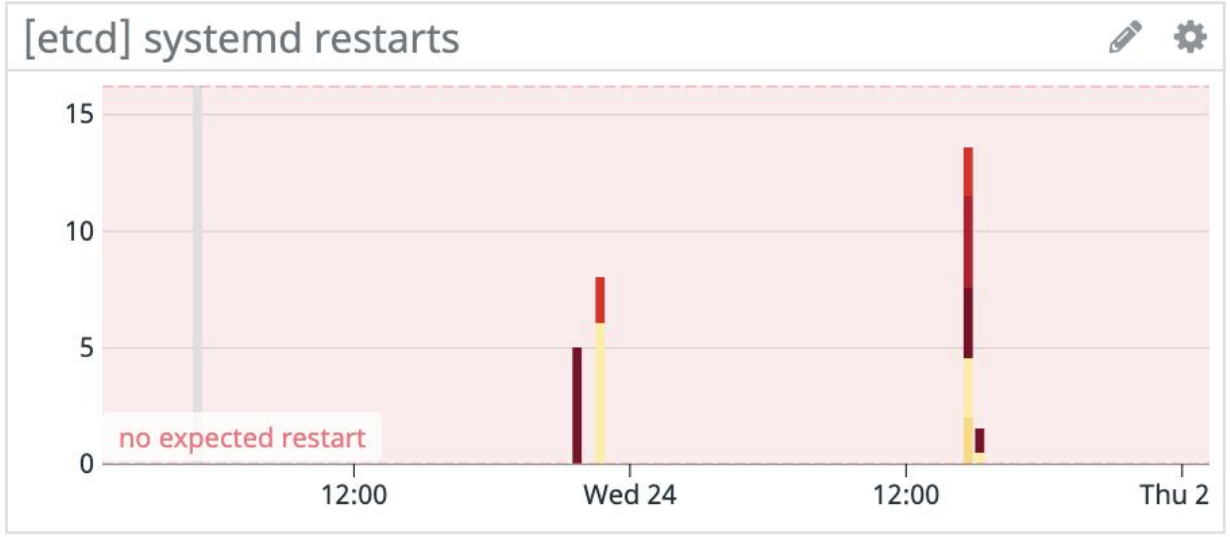


Apiservers crash/restart

Etcd?



Etcd memory usage is spiking

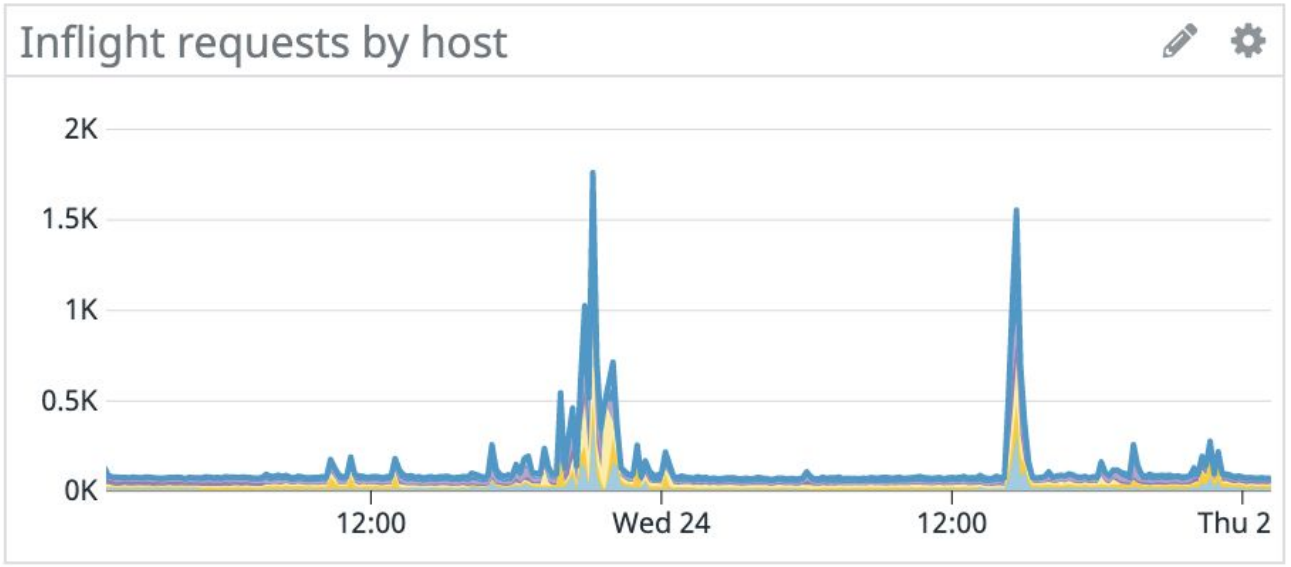


Etcd is oom-killed

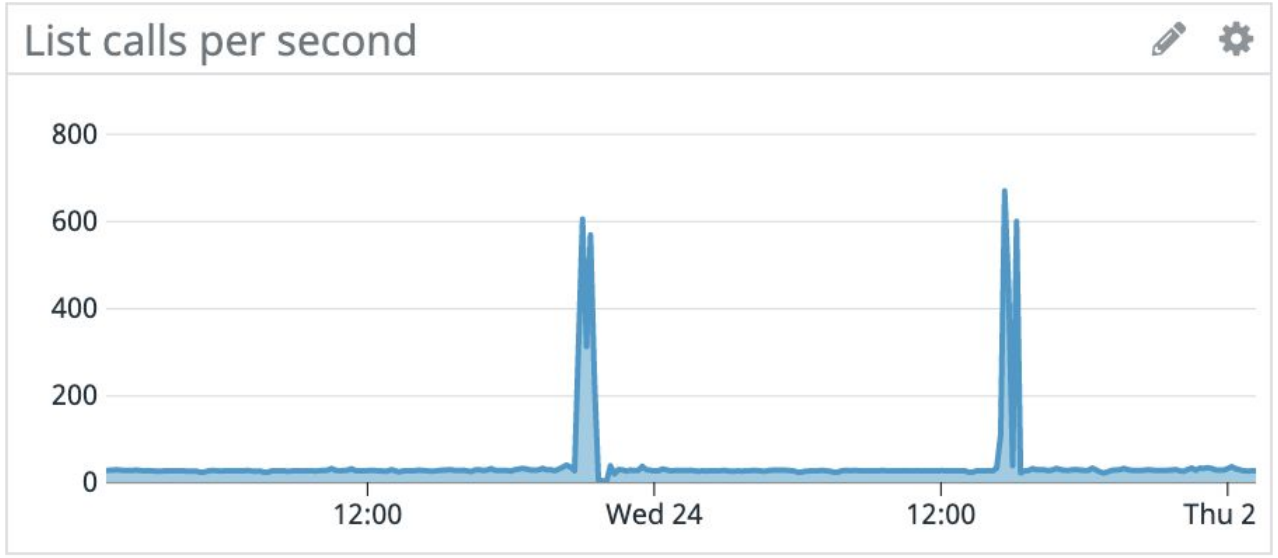
What we know

- Cluster size has not significantly changed
- The control plane has not been updated
- So it is very likely related to api calls

Apiserver requests



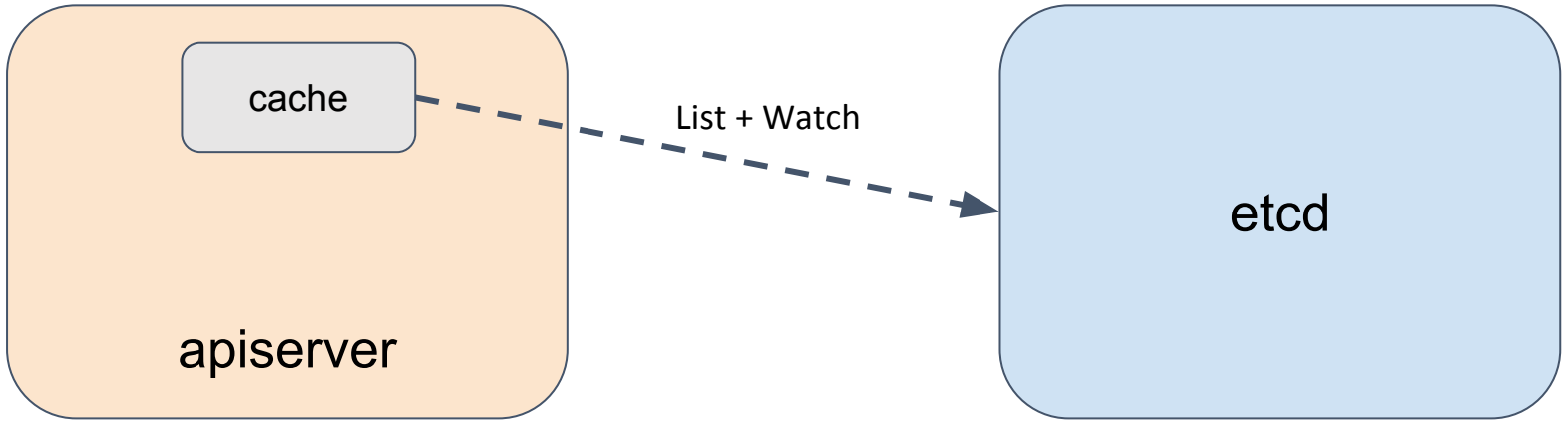
Spikes in inflight requests



Spikes in list calls
(very expensive)

Why are list calls expensive?

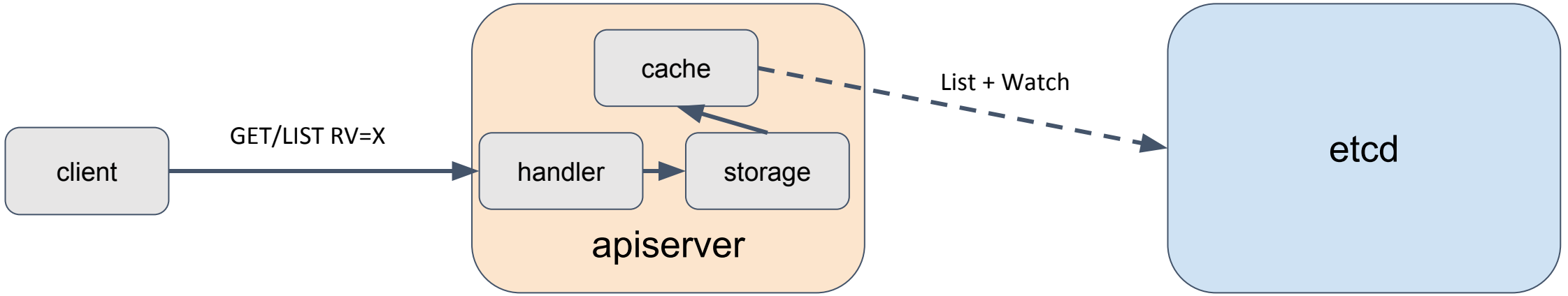
Understanding Apiserver caching



Why are list calls expensive?

KubeCon CloudNativeCon North America 2020 Virtual

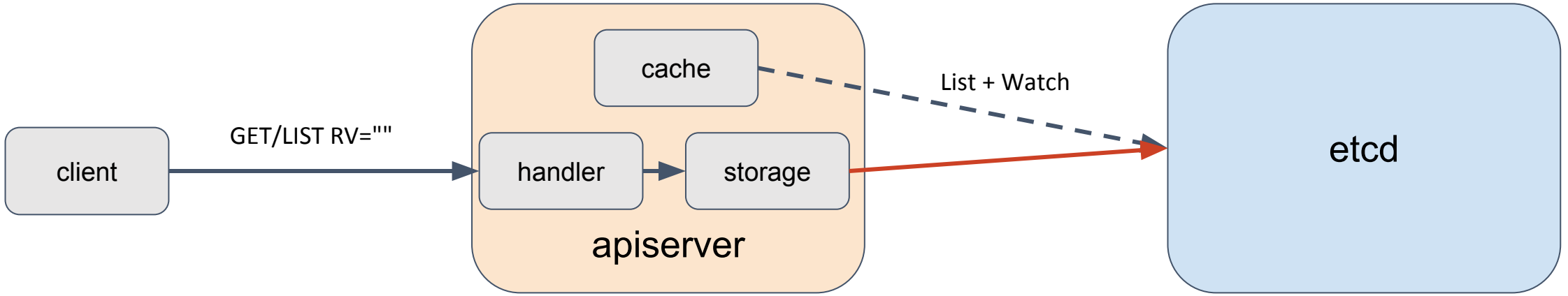
What happens for GET/LIST calls?



- Resources have versions (ResourceVersion, RV)
- For GET/LIST with RV=X
  If **cachedVersion >= X**, return **cachedVersion**
  else wait up to **3s**, if **cachedVersion>=X** return **cachedVersion**
  else **Error**: **"Too large resource version"**
  - ➢ *RV=X: "at least as fresh as X"*
  - ➢ *RV=0: current version in cache*

Why are list calls expensive?

What about GET/LIST without a resourceVersion?



- If RV is not set, apiserver performs a Quorum read against etcd (for consistency)
- This is the behavior of
  - kubectl get
  - client.CoreV1().Pods("").List() with default options (client-go)

Illustration


```
time curl 'https://cluster.dog/api/v1/pods'

real   0m4.631s


time curl 'https://cluster.dog/api/v1/pods?resourceVersion=0'

real   0m1.816s
```

- Test on a large cluster with more than 30k pods
- Using table view ("application/json;**as=Table**;v=v1beta1;g=meta.k8s.io, application/json")
  - Only ~25MB of data to minimize transfer time (full JSON: ~1GB)

What about label filters?


```
time curl 'https://cluster.dog/api/v1/pods?labelSelector=app=A'

real   0m3.658s


time curl 'https://cluster.dog/api/v1/pods?labelSelector=app=A&resourceVersion=0'

real   0m0.079s
```

- Call with RV="" is slightly faster (less data to send)
- Call with RV=0 is much much faster
  > Filtering is performed on cached data

- **When RV="", all pods are still retrieved from etcd and then filtered on apiservers**

Why not filter in etcd?

etcd key structure

/registry/{resource type}/{namespace)/{resource name}

So we can ask etcd for:

- a specific resource
- all resources of a type in a namespace
- all resources of a type
- **no other filtering / indexing**

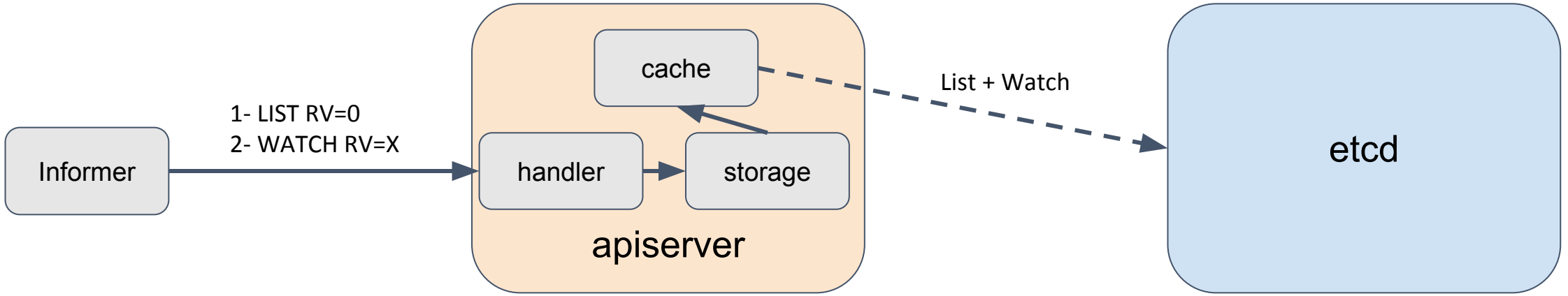
```
curl 'https://cluster.dog/api/v1/pods?labelSelector=app=A'
real    0m3.658s    <== get all pods (30k) in etcd, filter on apiserver

curl 'https://cluster.dog/api/v1/namespaces/datadog/pods?labelSelector=app=A'
real    0m0.188s    <== get pods from datadog namespace (1000) in etcd, filter on apiserver

curl 'https://cluster.dog/api/v1/namespaces/datadog/pods?labelSelector=app=A&resourceVersion=0'
real    0m0.058s    <== get pods from datadog namespace in apiserver cache, filter
```

Informers instead of List

How do informers work?



- Informers are much better because
  - They maintain a local cache, updated on changes
  - They start with a LIST using RV=0 to get data from the cache
- Edge case
  - Disconnections: can trigger a new LIST with RV="" (to avoid getting back in time)
  - Kubernetes 1.20 will have EfficientWatchResumption (alpha)

https://github.com/kubernetes/enhancements/tree/master/keps/sig-api-machinery/1904-efficient-watch-resumption

Summary

- LIST calls go to etcd by default and can have a huge impact
- LIST calls with label filters still retrieve everything from etcd
- **Avoid LIST, use Informers**

- kubectl get uses LIST with RV=""
- kubectl get could allow setting RV=0
  - Much faster: better user experience
  - Much better for etcd and apiservers
  - Trade-off: small inconsistency window

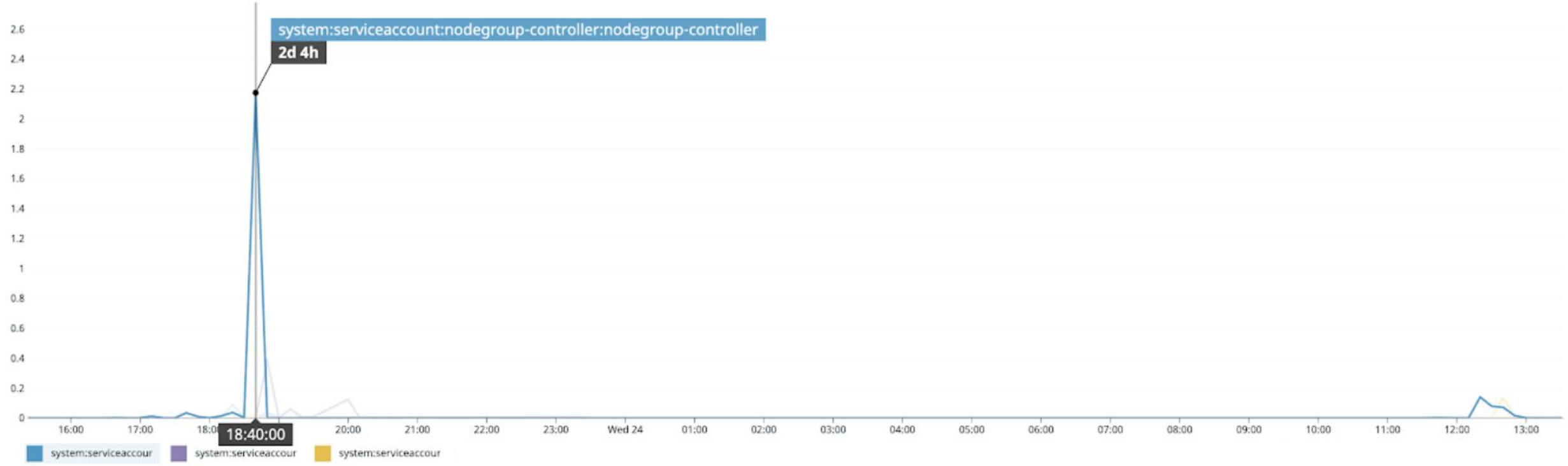
Many, many thanks to Wojtek-t for his help getting all this right

Back to the incident

- We know the problem comes from LIST calls
- What application is issuing these calls?

Audit logs

Cumulated query time by user for list calls



A single user accounts for 2+ days of query time over 20 minutes!

Audit logs

Cumulated query time by user for list/get calls over a week

VERB	USER USERNAME	SUM:DURATION
		2.19wk
list		**7.05day**
	system:serviceaccount:nodegroup-controller:nodegroup-cor	2.84day
	system:serviceaccount:kube-system:node-controller	46.68hr
	[redacted]@datadoghq.com	11.1hr
	system:serviceaccount:datadog-agent:datadog-agent-cluste	7.7hr
	system:serviceaccount:kube2iam:kube2iam	4.96hr
get		**3.34day**
	[redacted]@datadoghq.com	8.99hr
	kube-scheduler	4.25hr
	[redacted]@datadoghq.com	2.59hr
	[redacted]@datadoghq.com	2.46hr
	[redacted]@datadoghq.com	2.18hr

Nodegroup controller?

- An in-house controller to manage pools of nodes
- Used extensively for 2 years
- *But* recent upgrade deployed: deletion protection
  - Check if pods are running on pool of nodes
  - Deny nodegroup deletion if it is the case

How did it work?

On nodegroup delete

1. List all nodes in this nodegroup based on labels
   => Some groups have 100+ nodes

2. List all pods on each node filtering on bound node
   => List all pods (30k) for node
   => Performed in parallel for all nodes in the group
   => The bigger the nodegroup, the worse the impact

All LIST calls here retrieve full node/pod list from etcd

What we learned

- List calls are very dangerous
  - The volume of data can be very large
  - Filtering happens on the apiserver
  - Use Informers (whenever possible)

- Audit logs are extremely useful
  - Who did what when?
  - Which users are responsible for processing time

Conclusion

Conclusion

- Apiservice extensions are powerful but can harm your cluster
- Nodes are not abstract compute resources
- For large (1000+ nodes) clusters
  - Understanding Apiserver caching is important
  - Client interactions with Apiservers are critical
  - Avoid LIST calls

Thank you

We're hiring!
https://www.datadoghq.com/careers/

laurent@datadoghq.com
@lbernail
@lbernail

DATADOG


KEEP CLOUD NATIVE
EVERYWHERE
KubeCon
CloudNativeCon
North America 2020
Virtual