KubeCon | CloudNativeCon

Europe 2023

KubeCon | CloudNativeCon

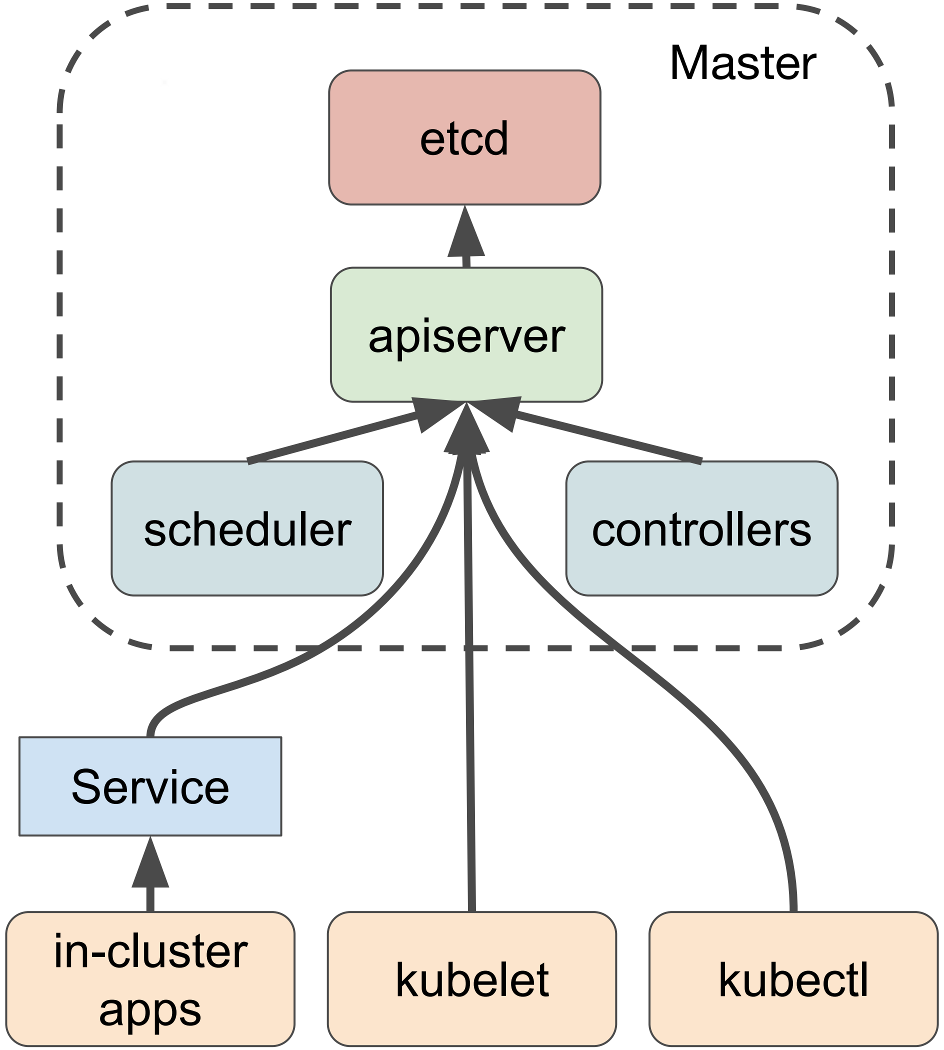
Europe 2023

Setting up Etcd with Kubernetes to Host Clusters with Thousands of Nodes

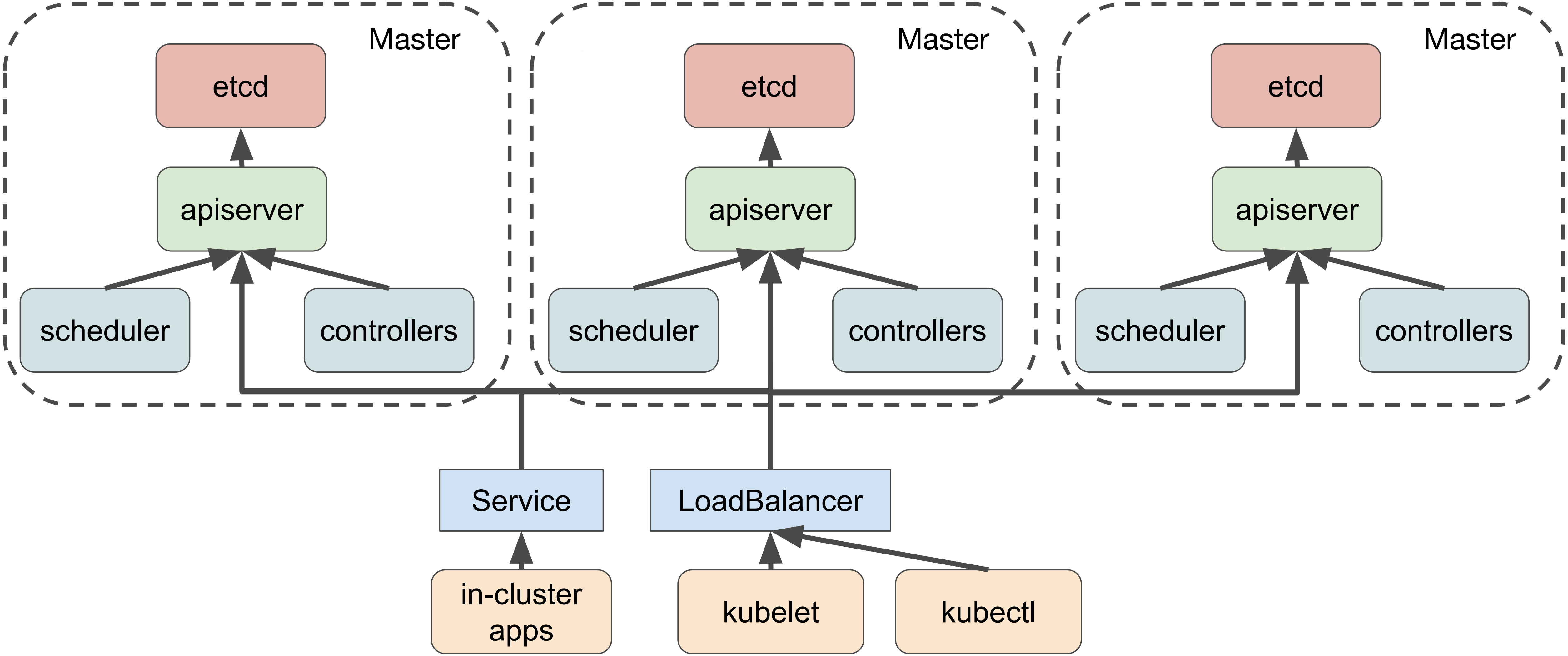
Marcel Zięba (Isovalent)
Laurent Bernaille (Datadog)

Scaling Kubernetes control planes

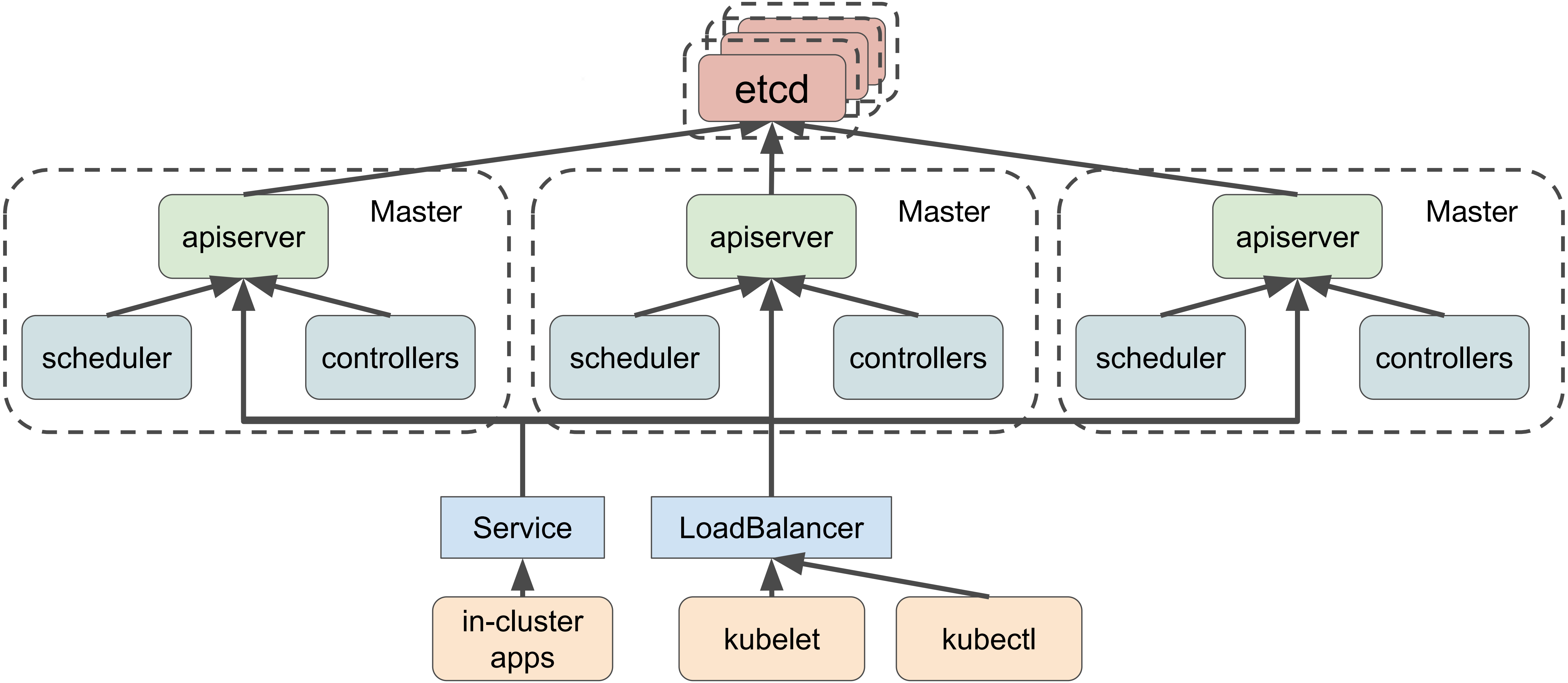
Kube 101 Control Plane



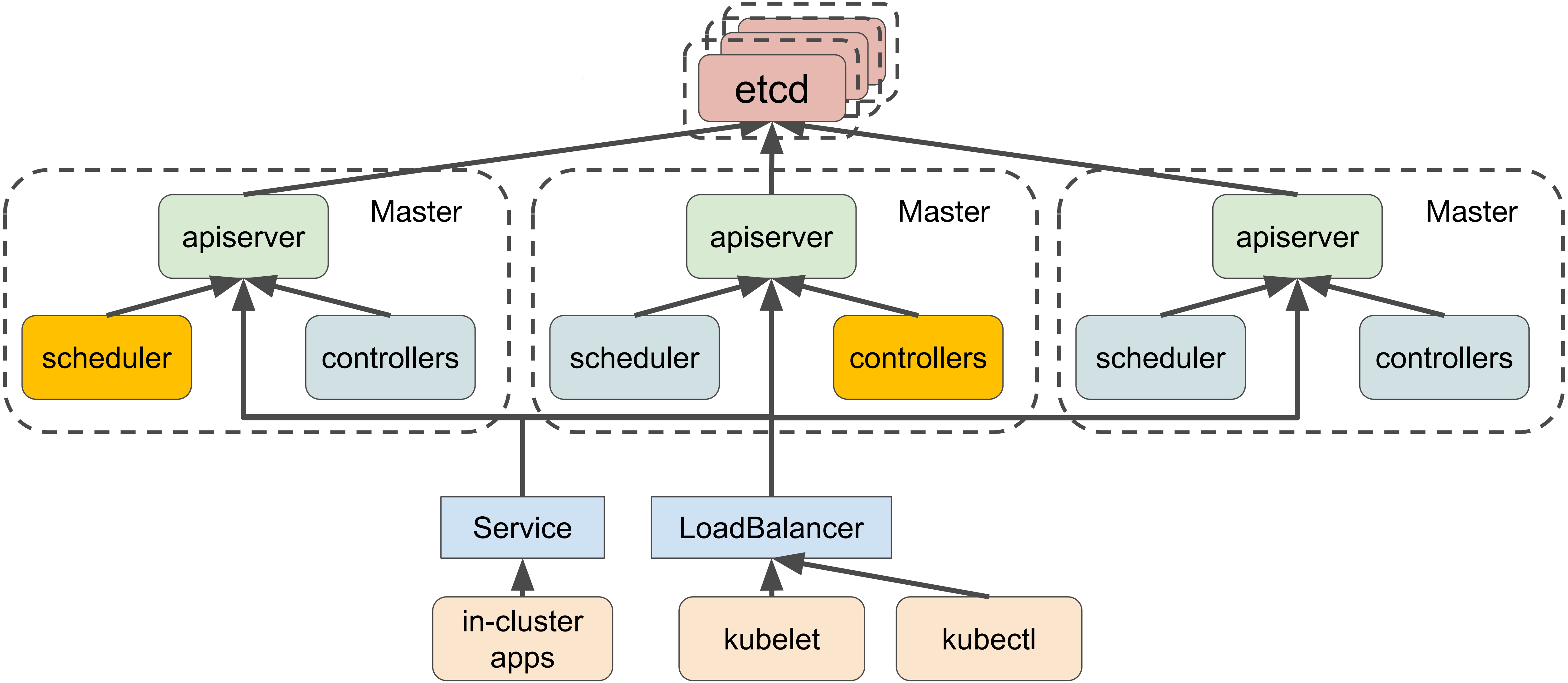
Making it resilient



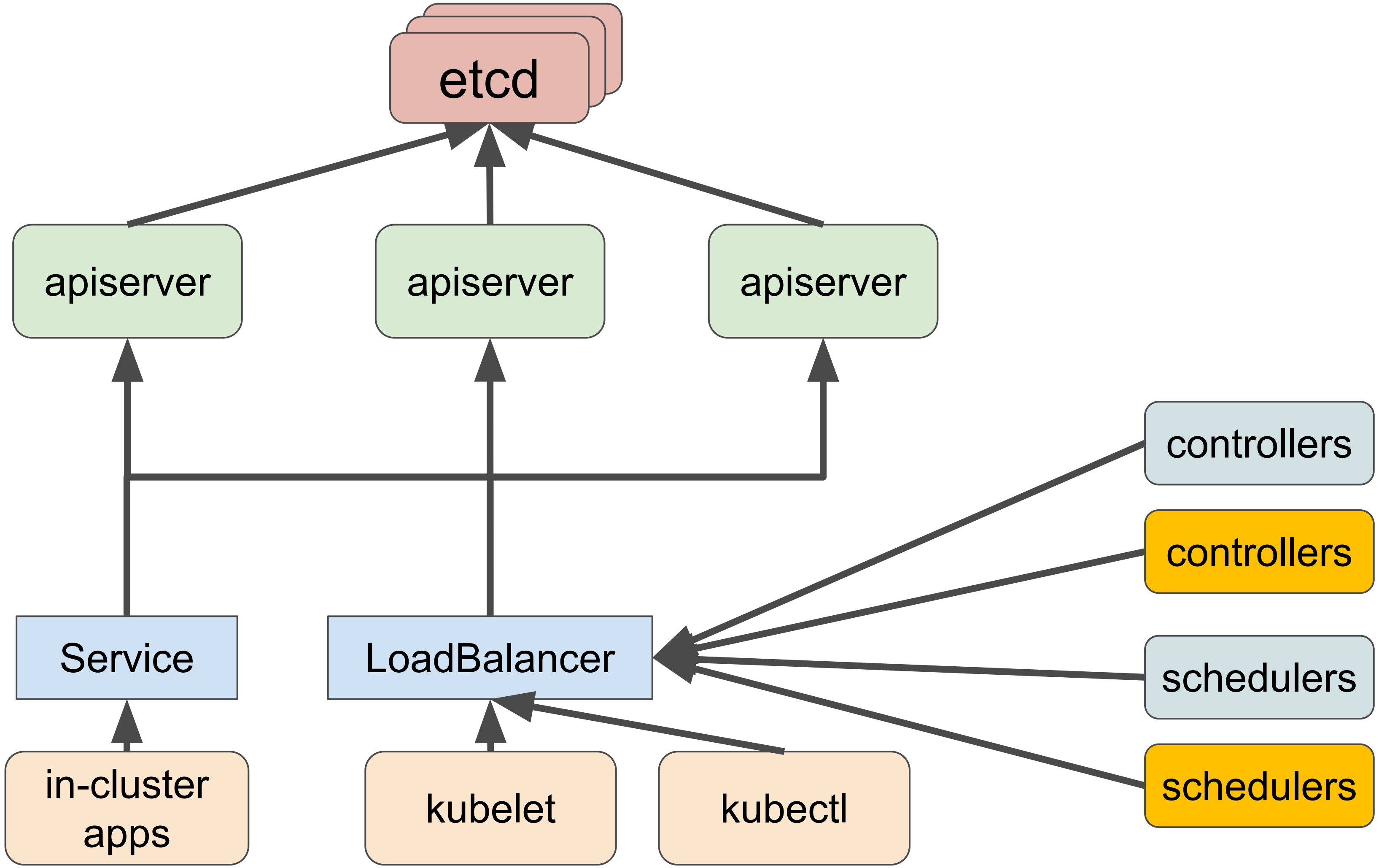
Separate etcd nodes



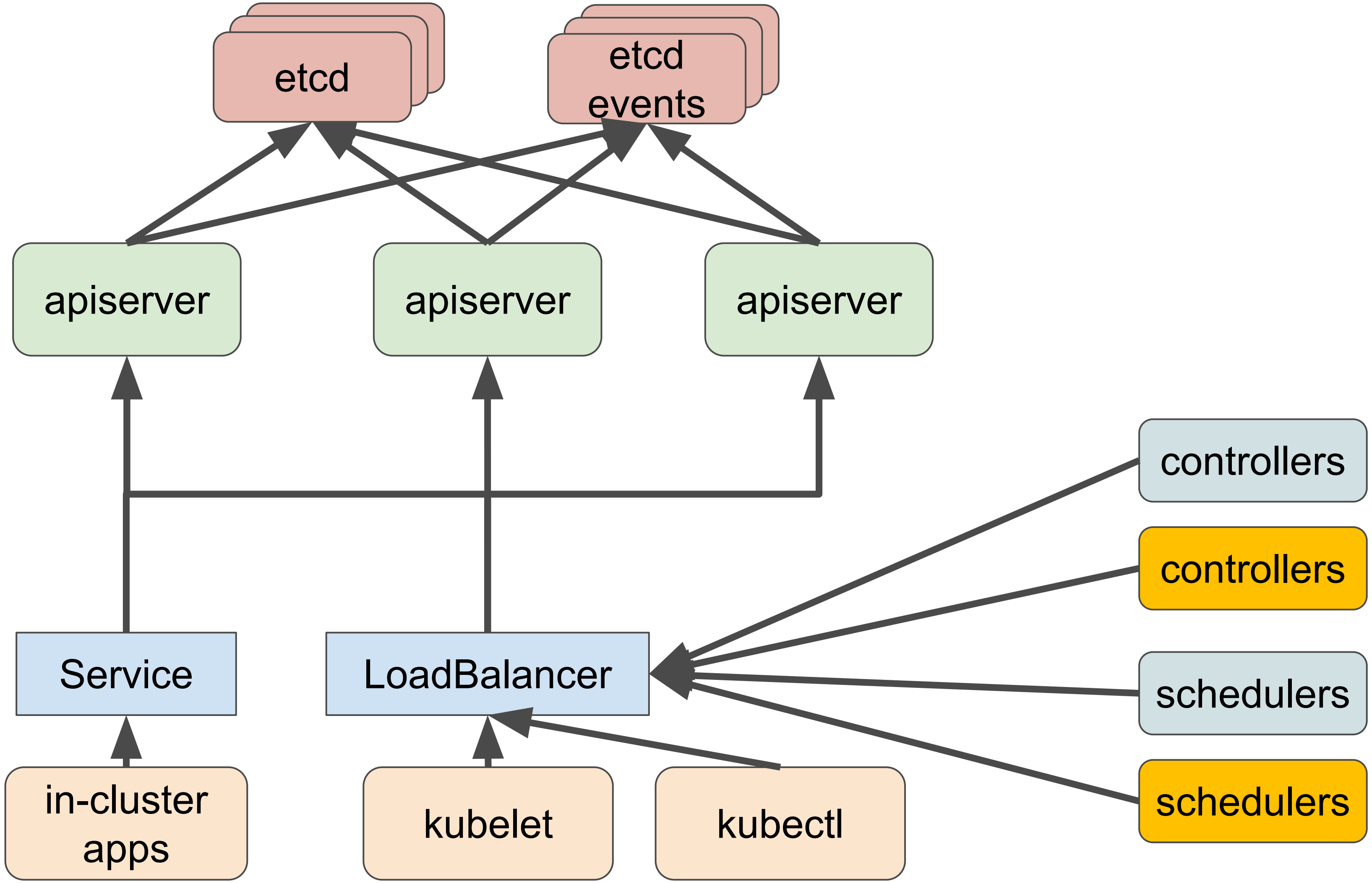
Single active Controller/scheduler



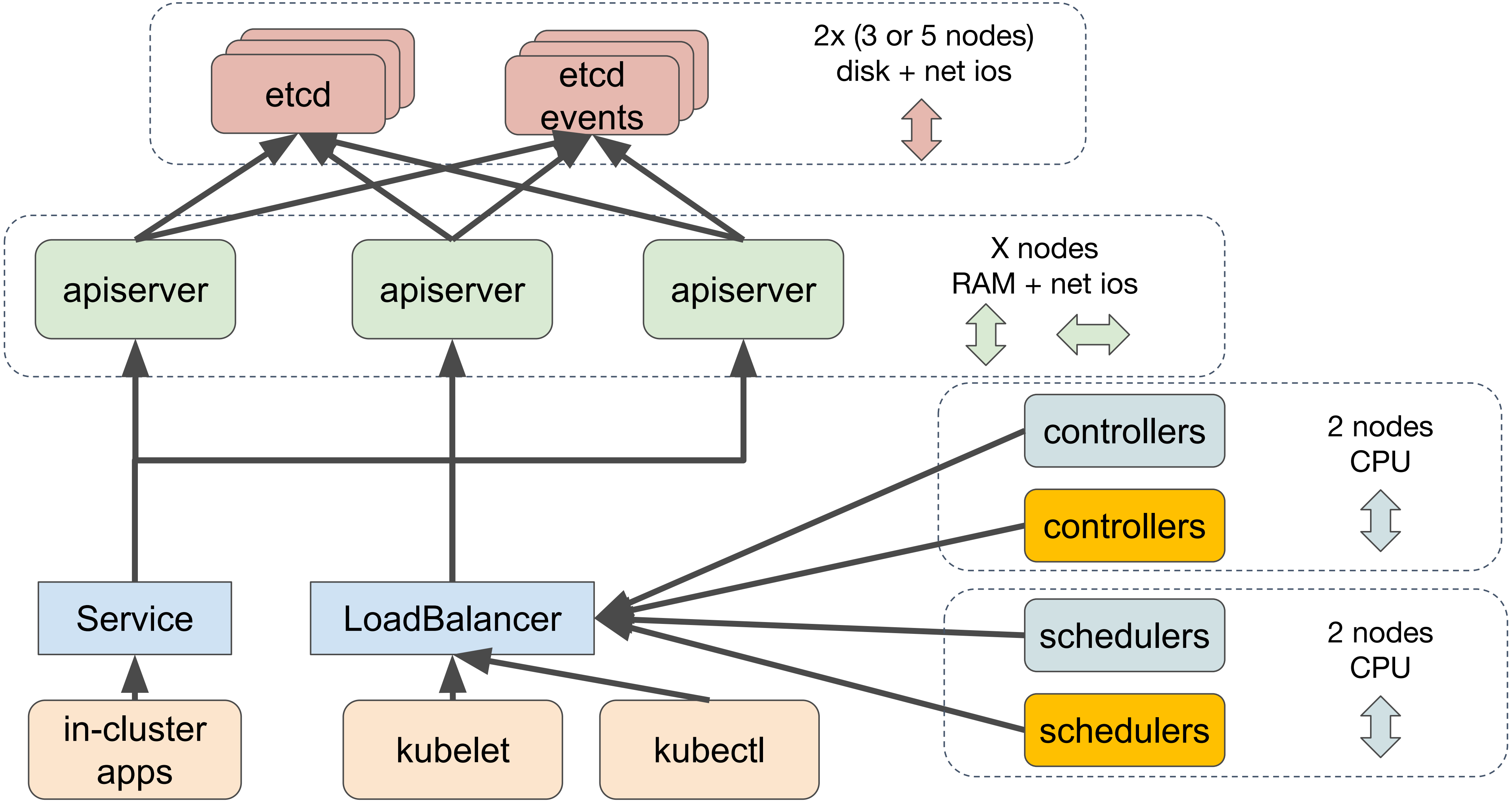
Split Controller and Scheduler



Split etcd


etcd
etcd events
apiserver
apiserver
apiserver
controllers
controllers
Service
LoadBalancer
schedulers
schedulers
in-cluster apps
kubelet
kubectl

Sizing the control plane



What if you use managed services?

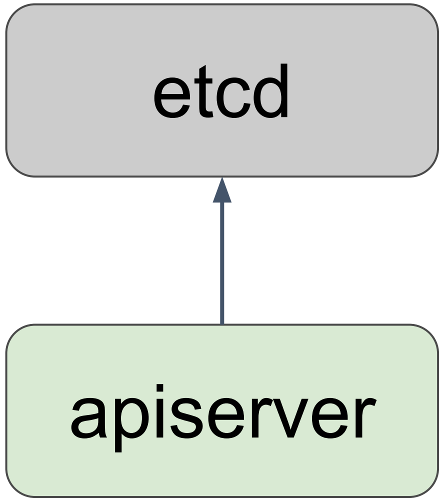
- Control
  - #nodes
  - #services
  - Churn of pods
- Reduce some of the traffic
  - Headless services
  - Immutable configmaps/secrets
  - Cilium with kvstore
- Make sure your operators/daemonsets behave “nicely”

KubeCon | CloudNativeCon

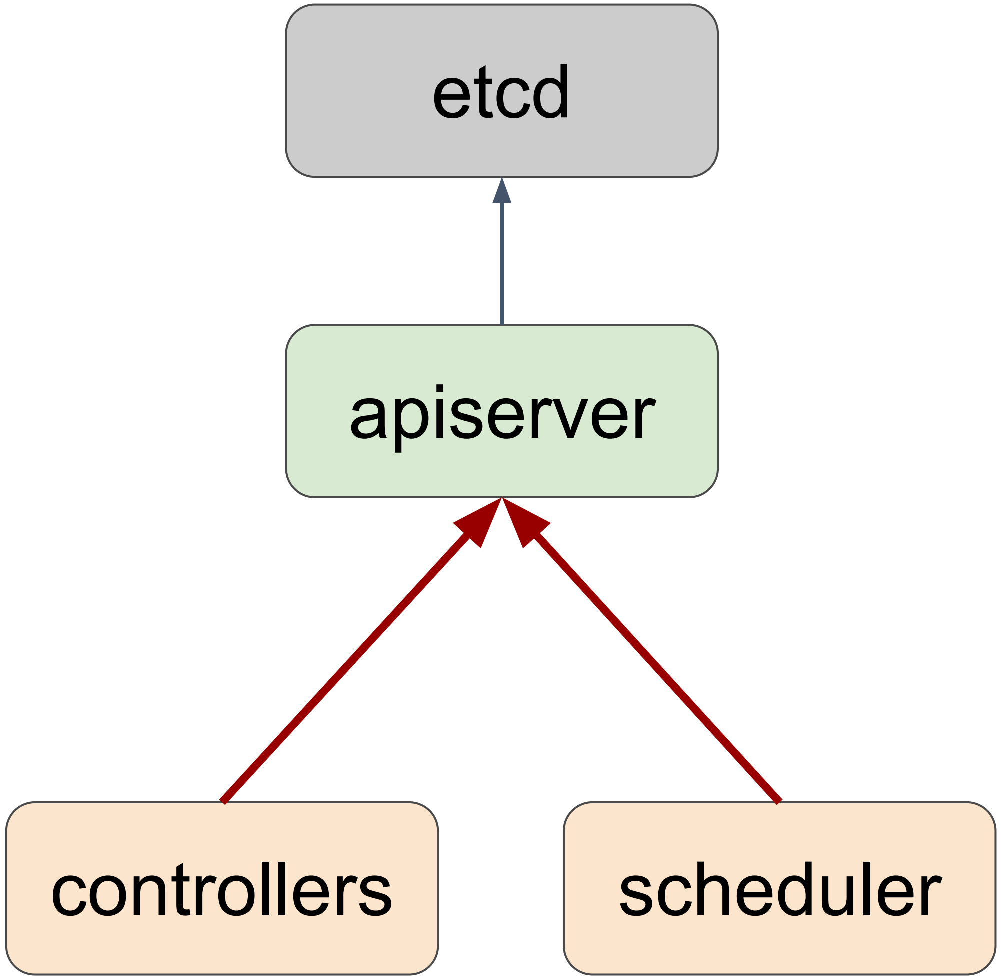
Europe 2023

Understanding the Kubernetes API

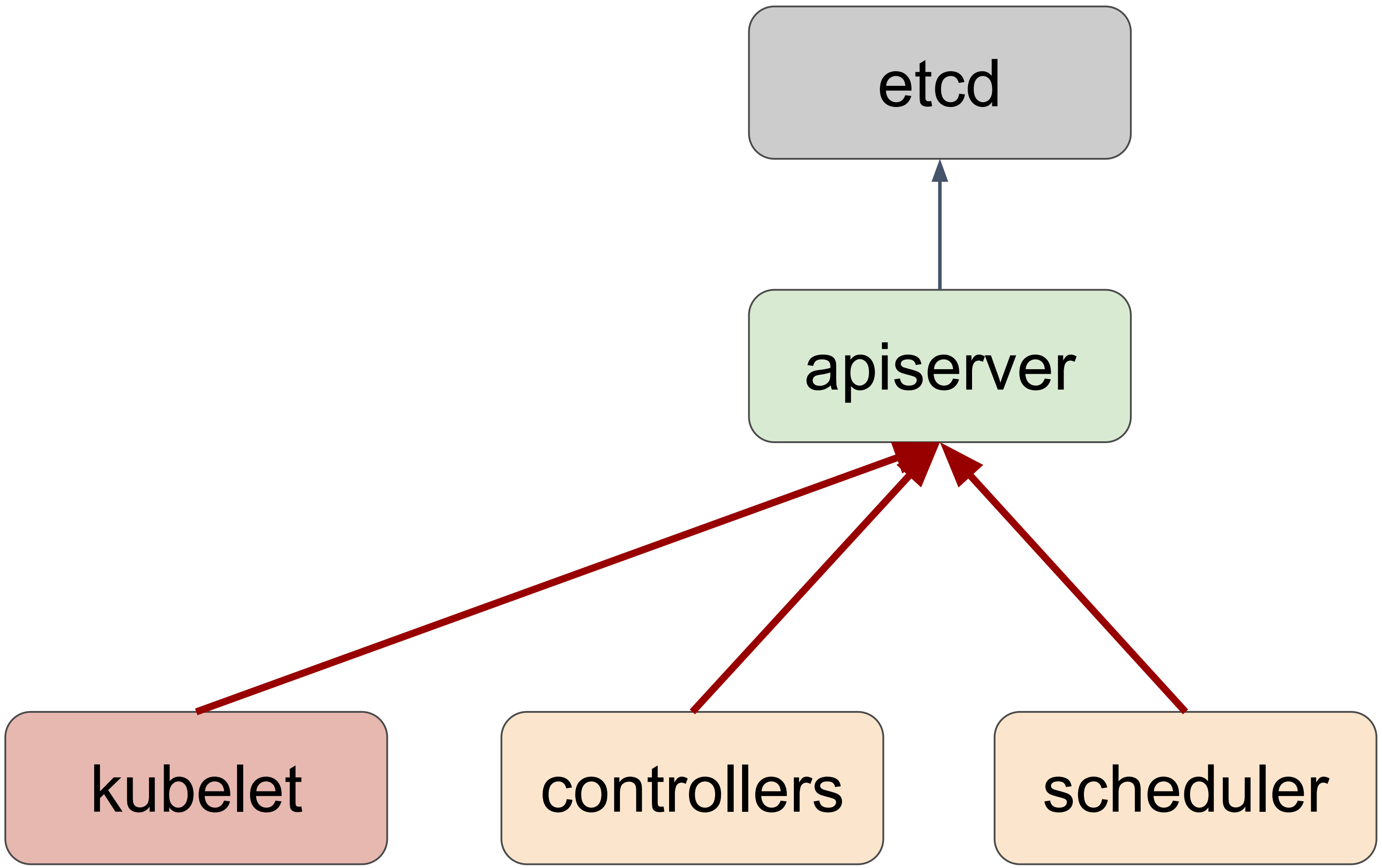
Calls to the apiservers



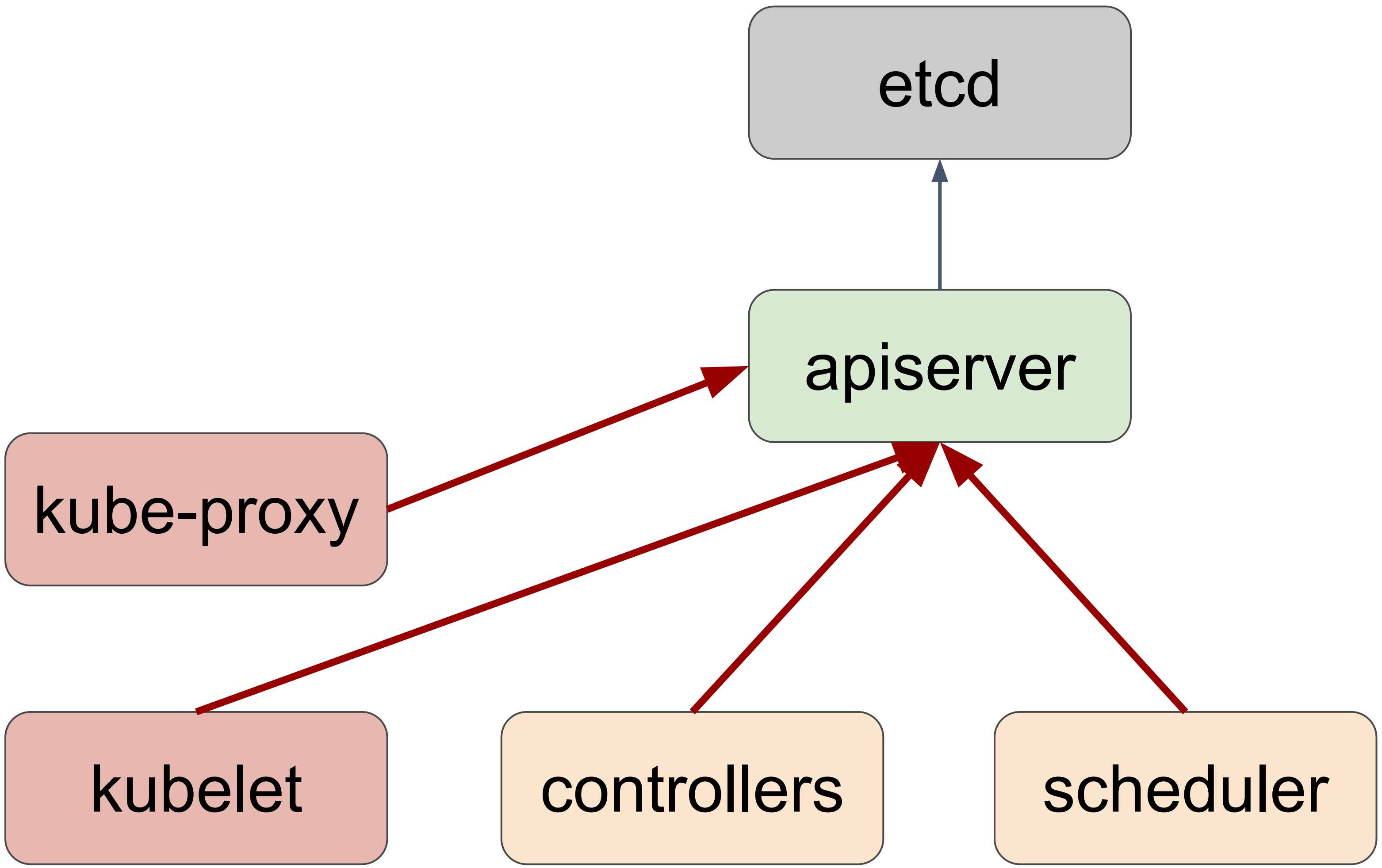
Control plane



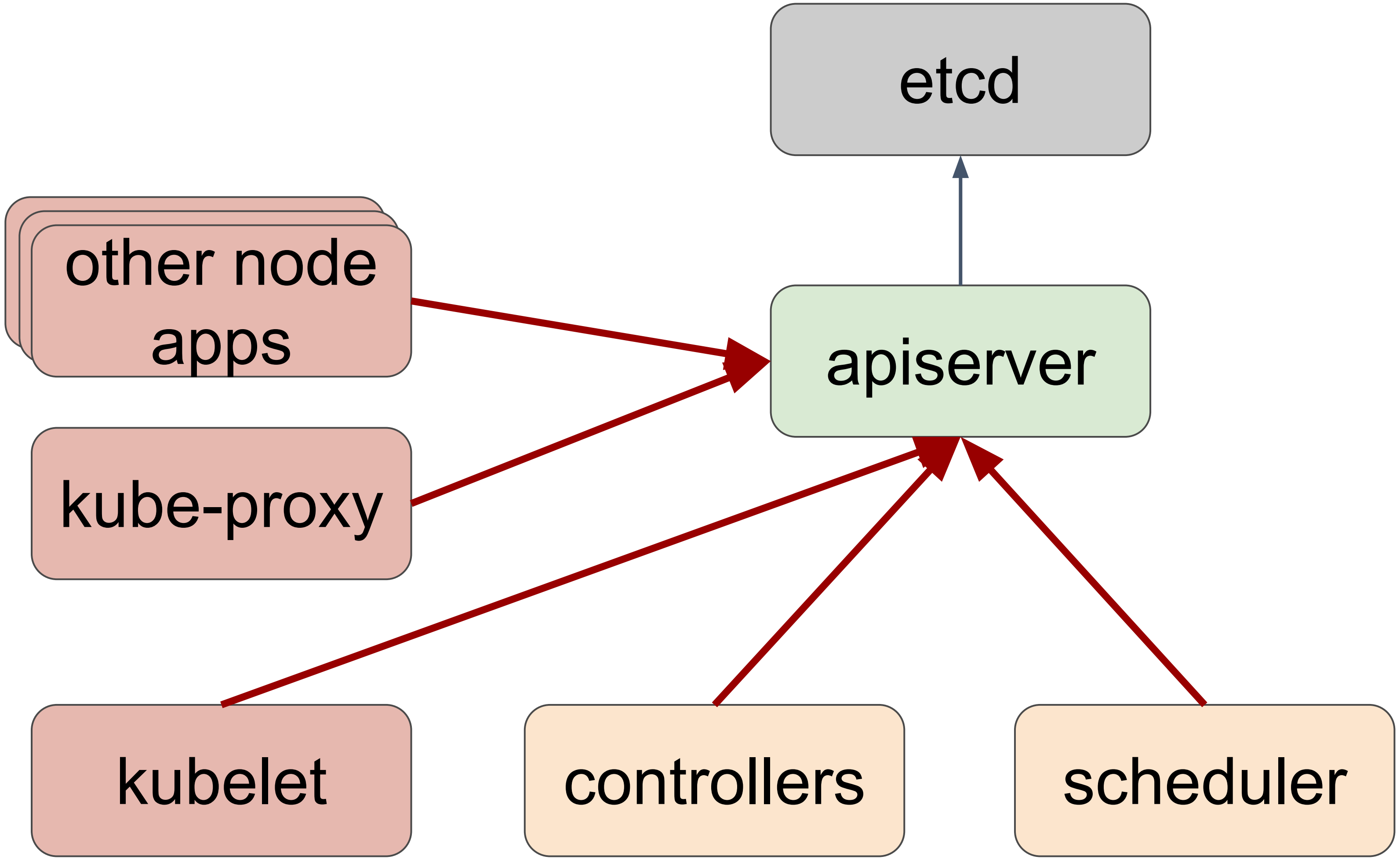
Kubelet



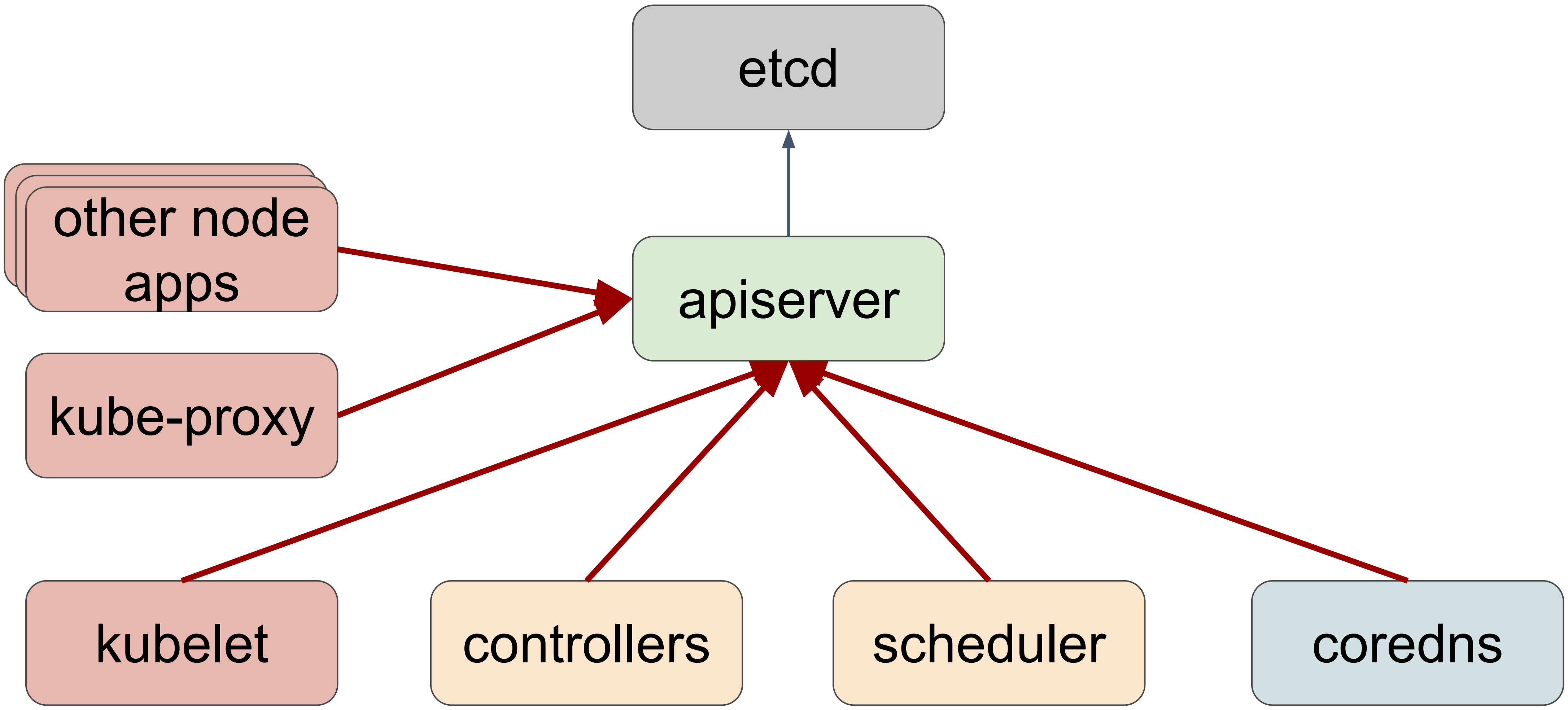
DaemonSet: kube-proxy



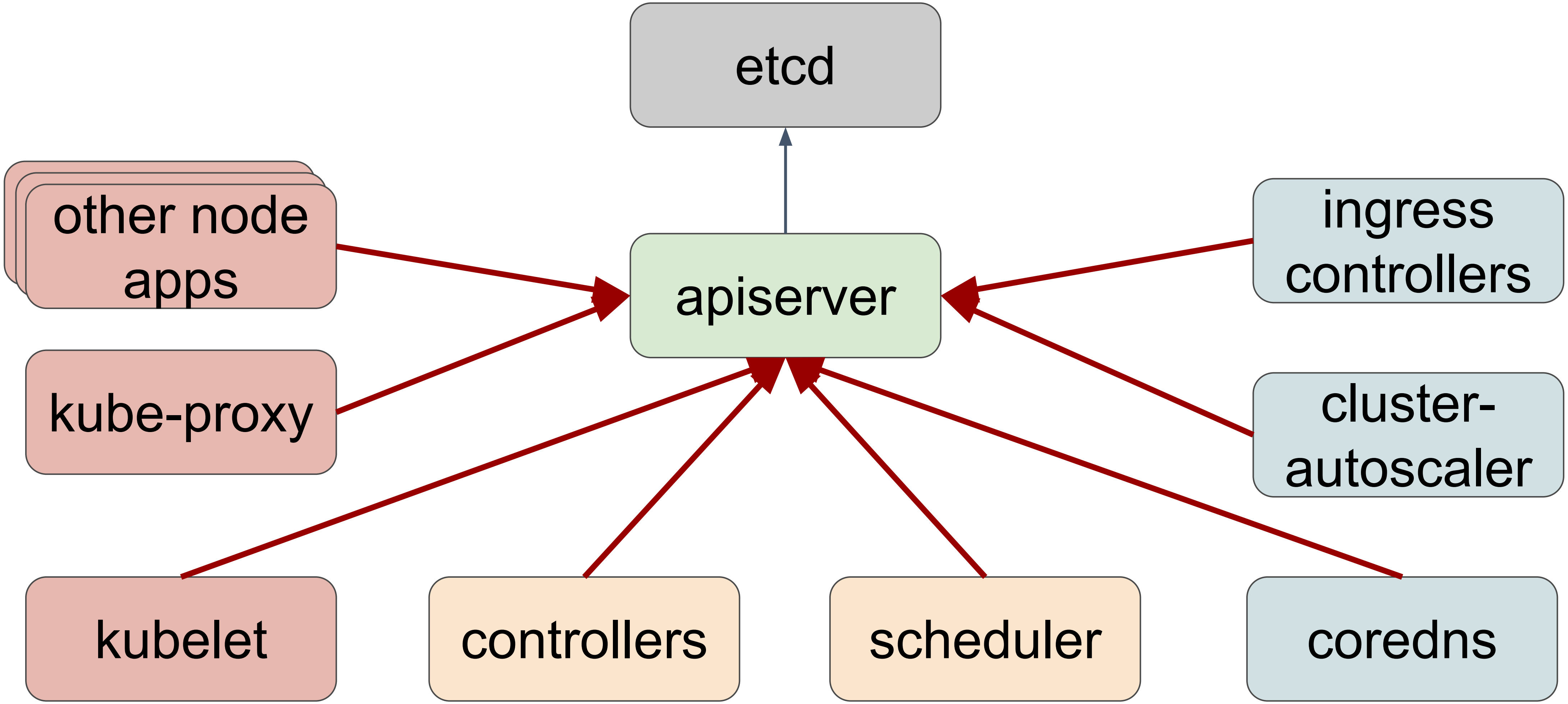
Other DaemonSets (cni, etc.)



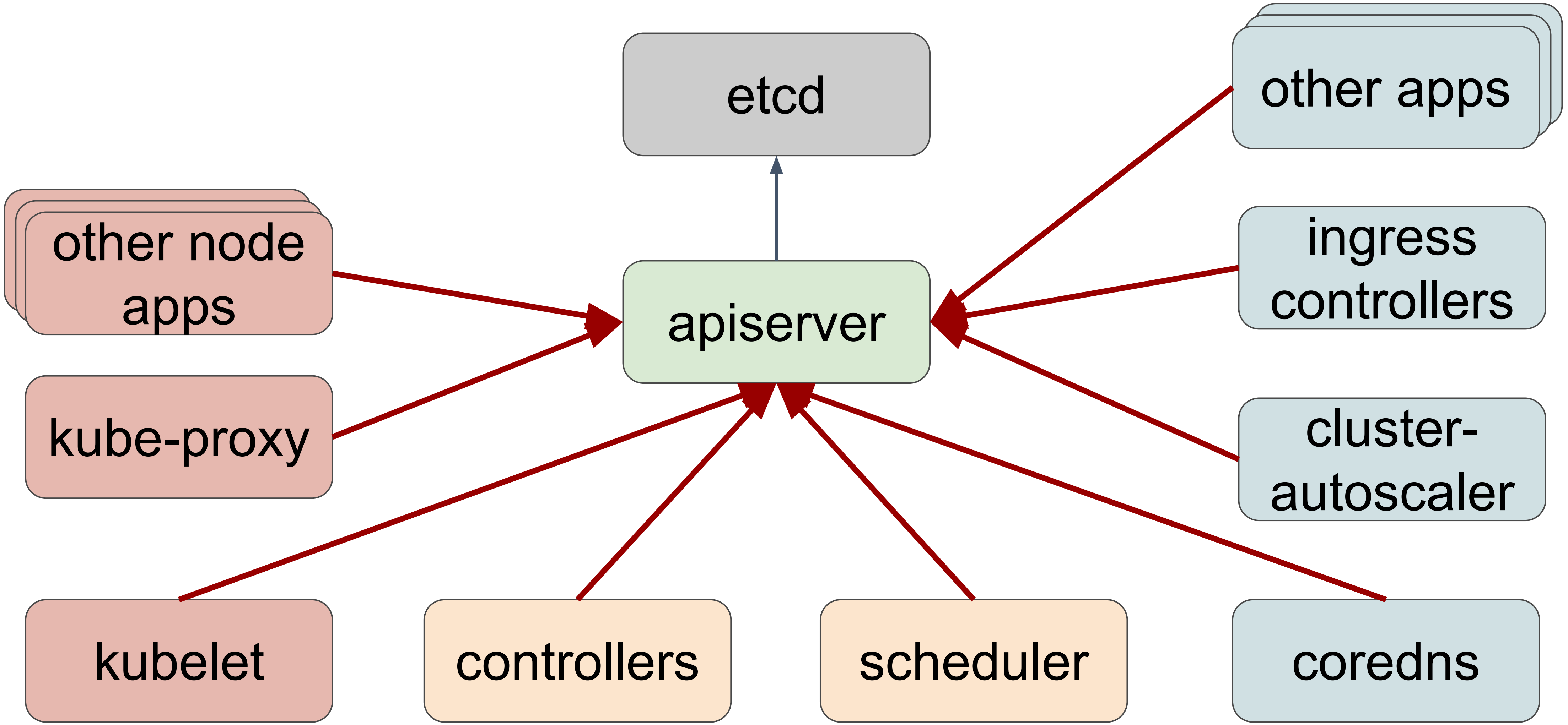
Cluster services: DNS



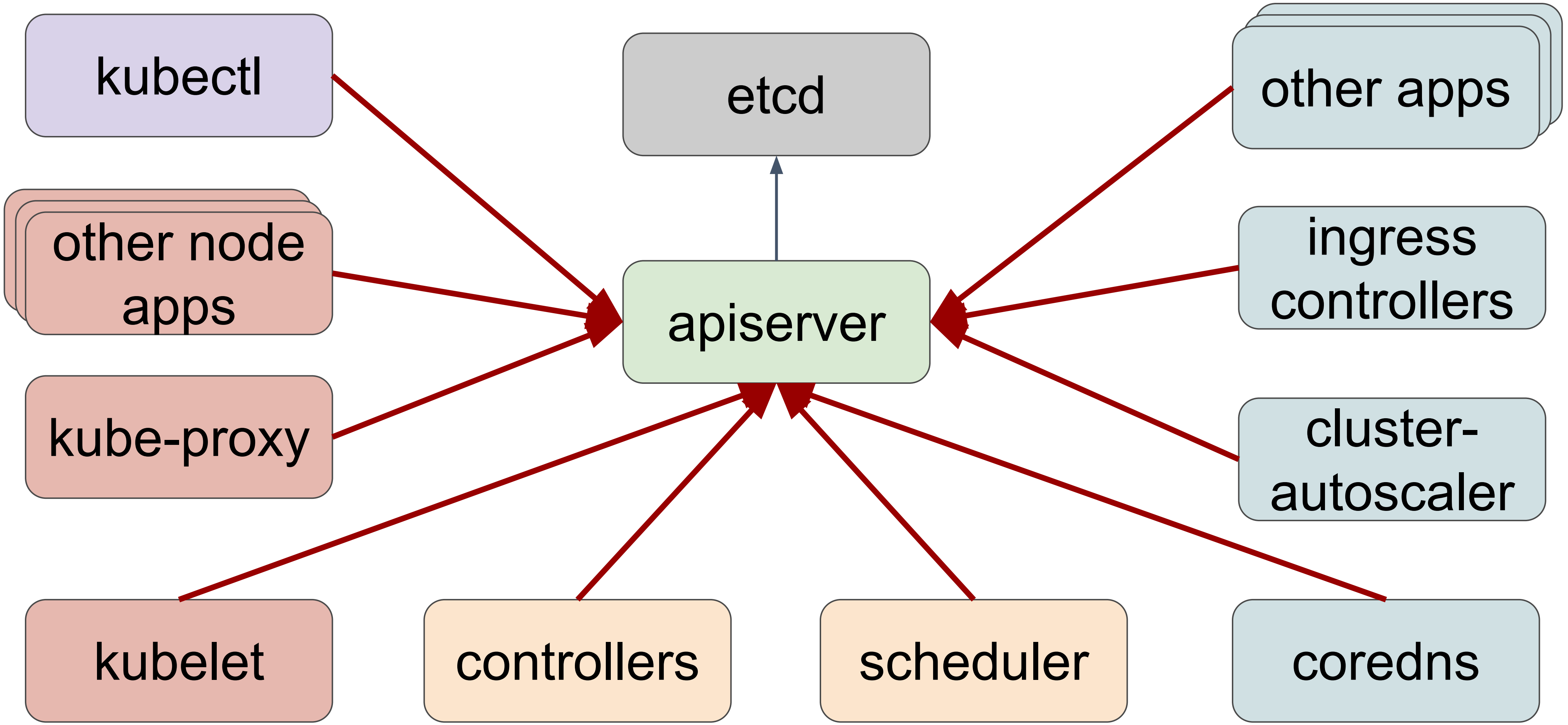
Other cluster services



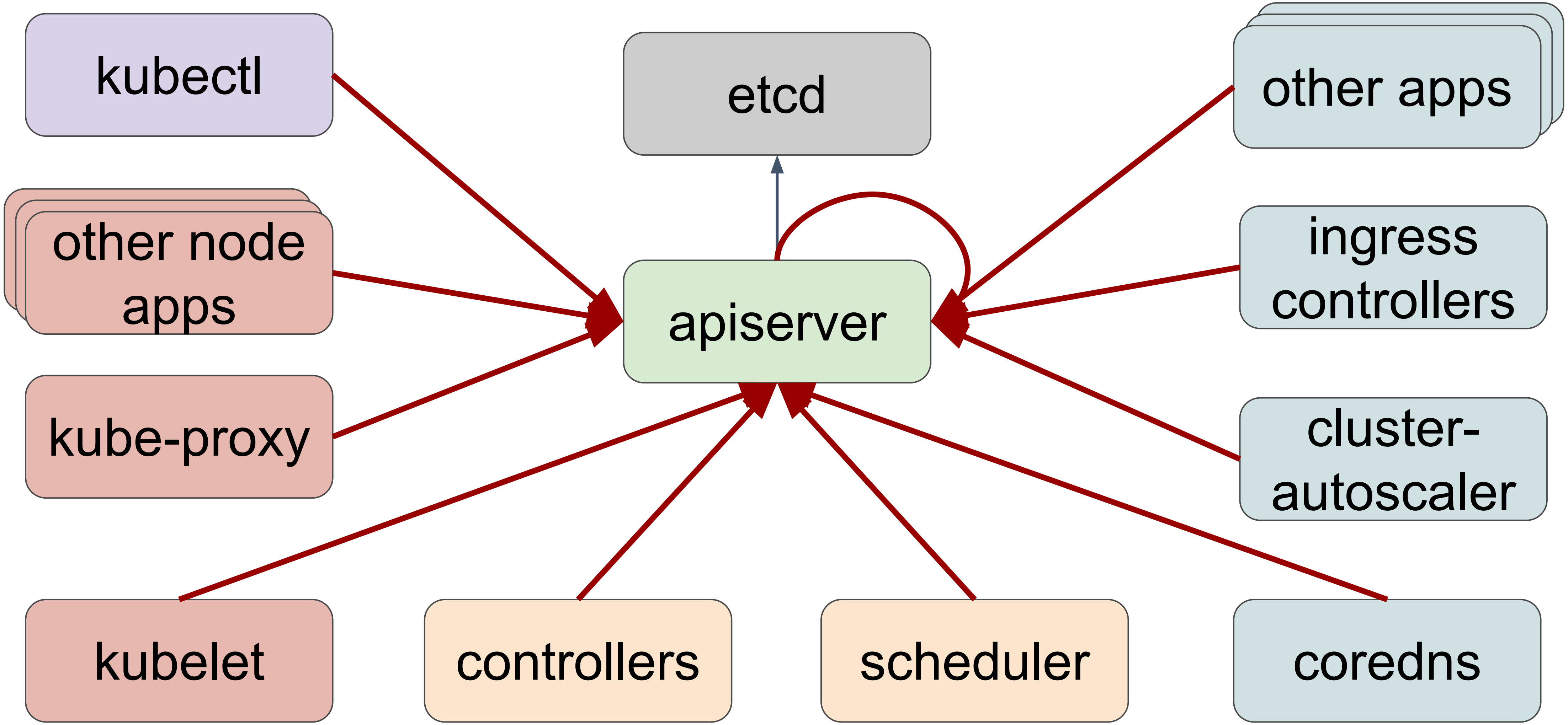
Probably several other applications



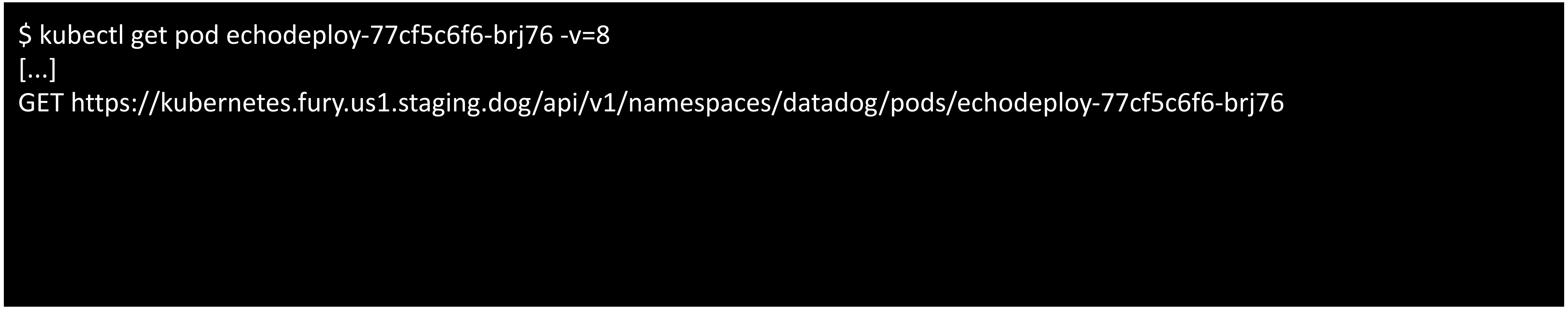
And users, of course



And, surprise, the apiserver itself

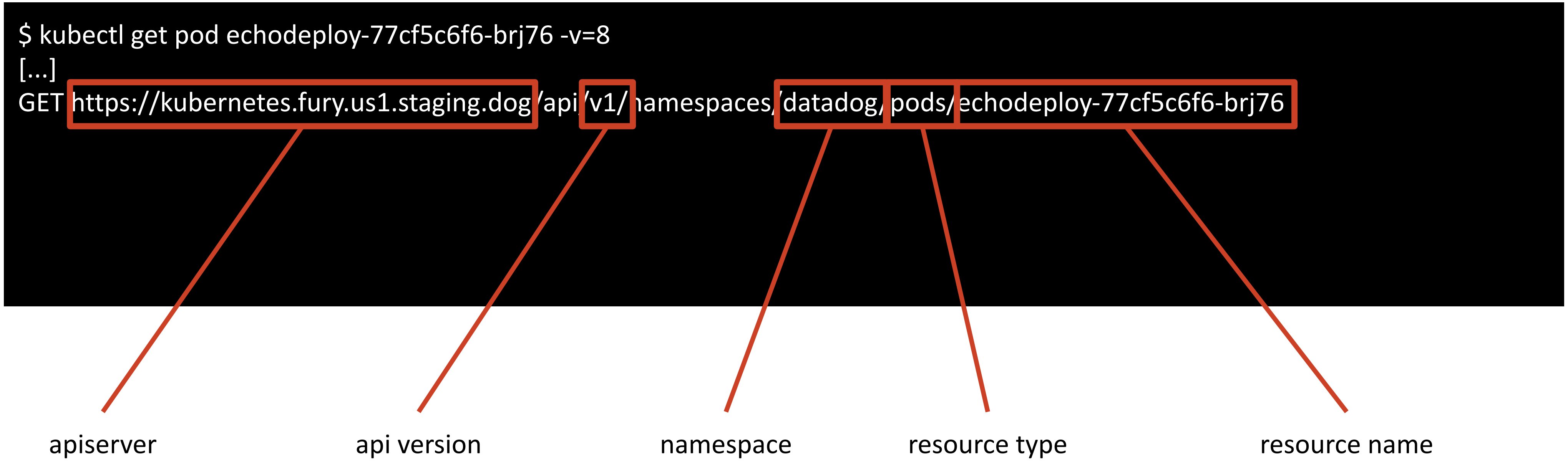


What happens when you kubectl?



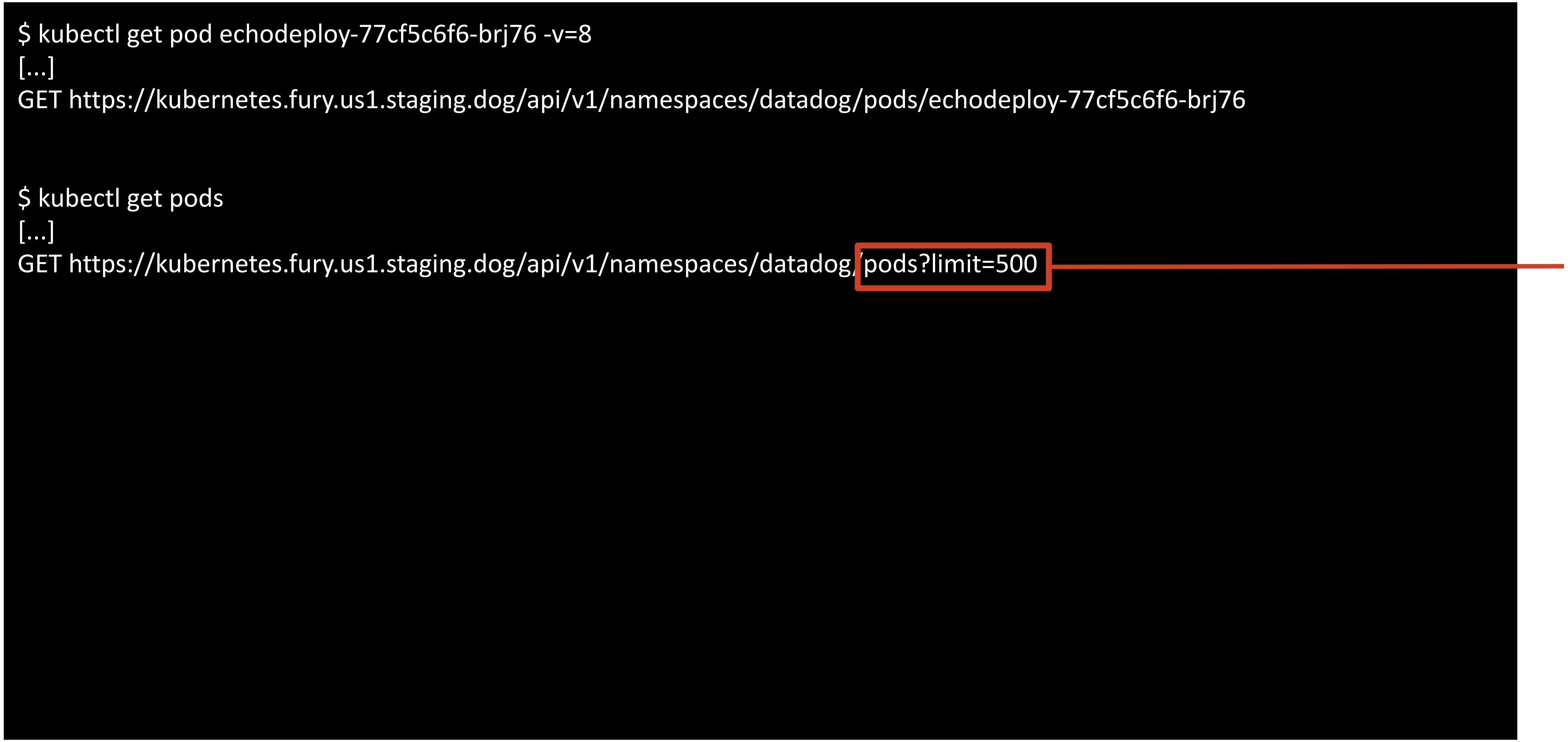
```
$ kubectl get pod echodeploy-77cf5c6f6-brj76 -v=8
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods/echodeploy-77cf5c6f6-brj76
```

Let’s look at details



```
$ kubectl get pod echodeploy-77cf5c6f6-brj76 -v=8
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods/echodeploy-77cf5c6f6-brj76
```

A few more GET examples

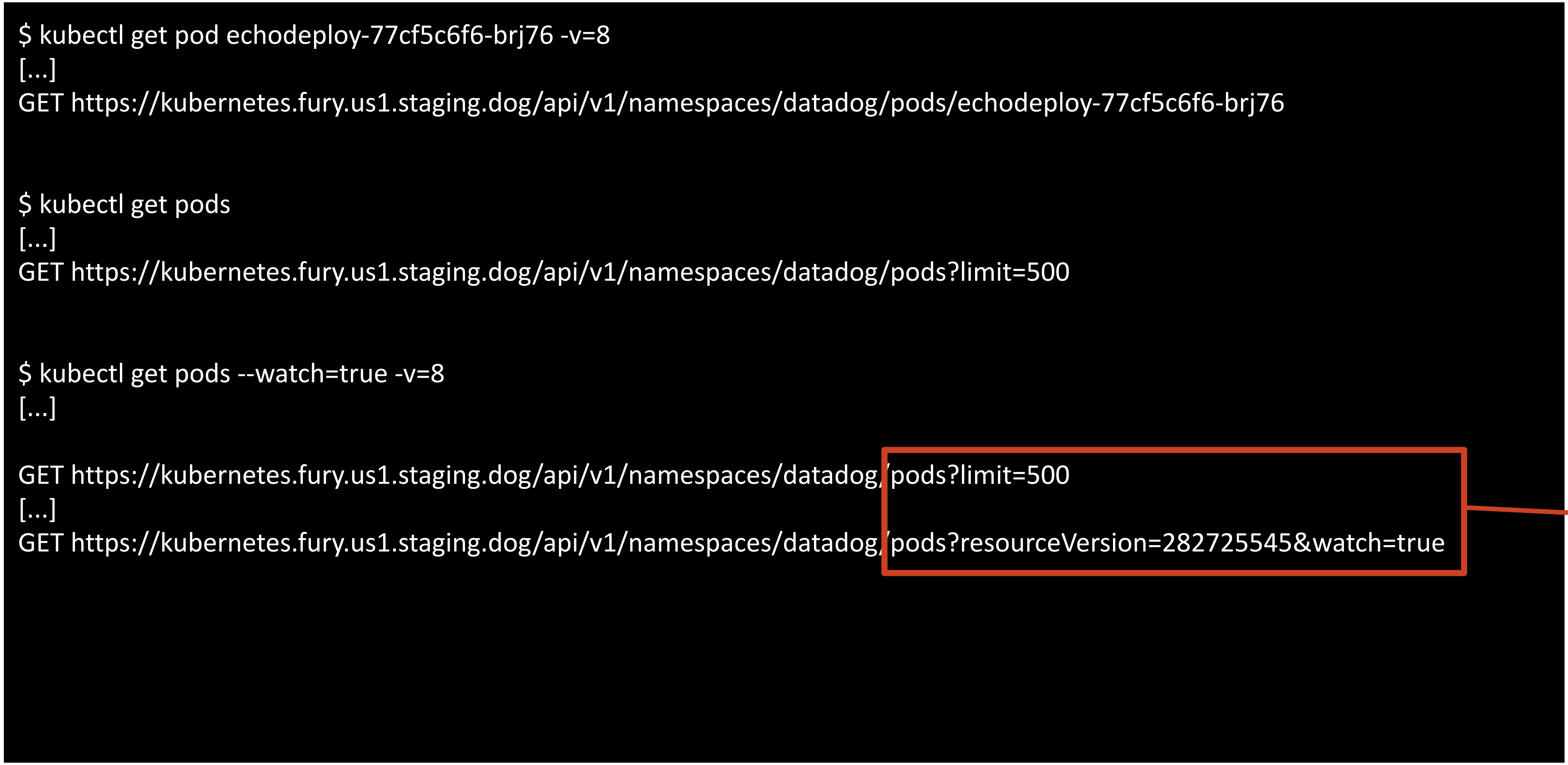


```
$ kubectl get pod echodeploy-77cf5c6f6-brj76 -v=8
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods/echodeploy-77cf5c6f6-brj76


$ kubectl get pods
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods?limit=500
```

List
(paginated)

A few more GET examples



```
$ kubectl get pod echodeploy-77cf5c6f6-brj76 -v=8
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods/echodeploy-77cf5c6f6-brj76


$ kubectl get pods
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods?limit=500


$ kubectl get pods --watch=true -v=8
[...]

GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods?limit=500
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods?resourceVersion=282725545&watch=true
```

List &
Watch

Describe resource

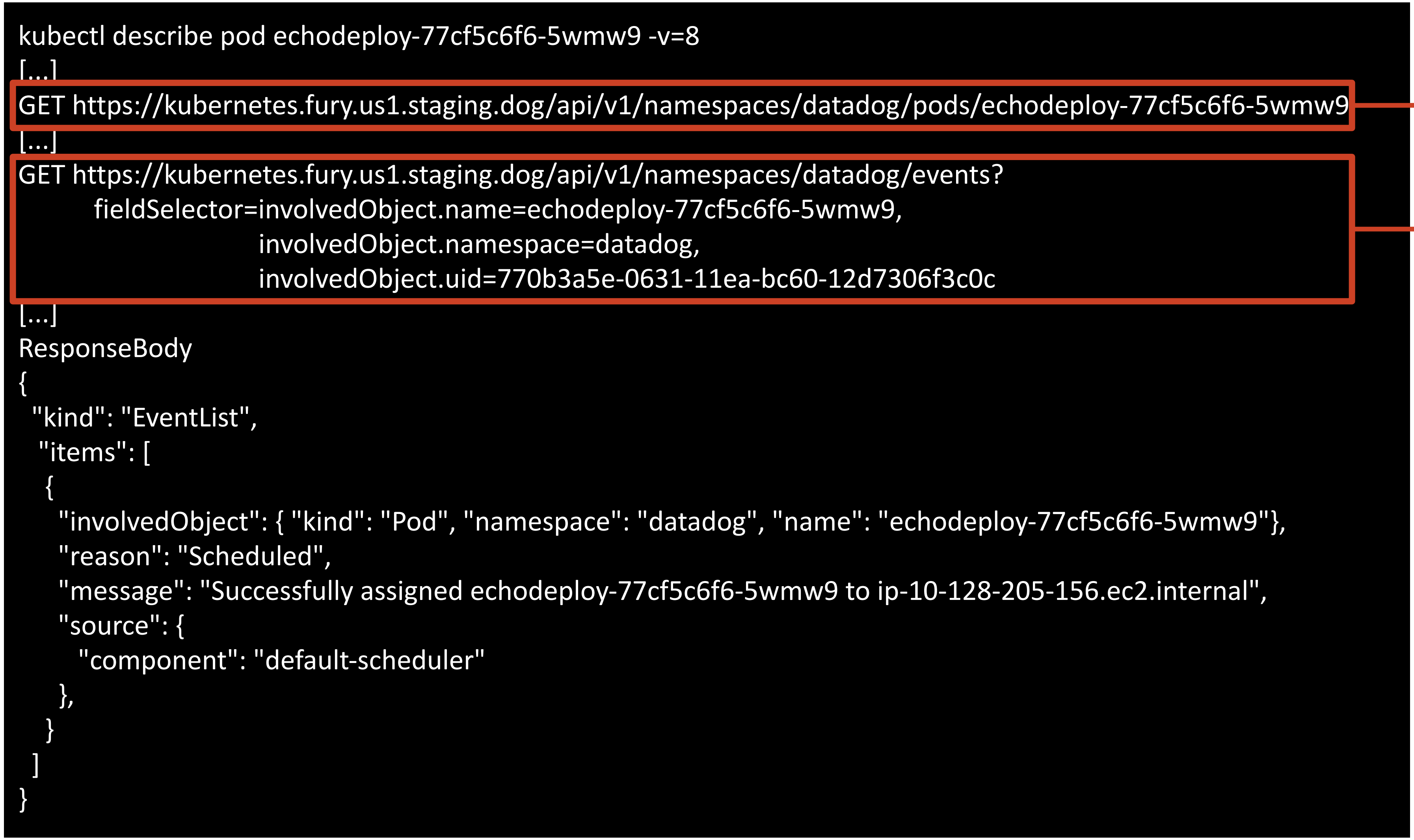
```
kubectl describe pod echodeploy-77cf5c6f6-5wmw9 -v=8
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods/echodeploy-77cf5c6f6-5wmw9
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/events?
        fieldSelector=involvedObject.name=echodeploy-77cf5c6f6-5wmw9,
                      involvedObject.namespace=datadog,
                      involvedObject.uid=770b3a5e-0631-11ea-bc60-12d7306f3c0c
[...]
ResponseBody
{
 "kind": "EventList",
  "items": [
   {
    "involvedObject": { "kind": "Pod", "namespace": "datadog", "name": "echodeploy-77cf5c6f6-5wmw9"},
    "reason": "Scheduled",
    "message": "Successfully assigned echodeploy-77cf5c6f6-5wmw9 to ip-10-128-205-156.ec2.internal",
    "source": {
      "component": "default-scheduler"
    },
   }
 ]
}
```

Describe resource

```
kubectl describe pod echodeploy-77cf5c6f6-5wmw9 -v=8
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods/echodeploy-77cf5c6f6-5wmw9
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/events?
        fieldSelector=involvedObject.name=echodeploy-77cf5c6f6-5wmw9,
                          involvedObject.namespace=datadog,
                          involvedObject.uid=770b3a5e-0631-11ea-bc60-12d7306f3c0c
[...]
ResponseBody
{
 "kind": "EventList",
  "items": [
   {
    "involvedObject": { "kind": "Pod", "namespace": "datadog", "name": "echodeploy-77cf5c6f6-5wmw9"},
    "reason": "Scheduled",
    "message": "Successfully assigned echodeploy-77cf5c6f6-5wmw9 to ip-10-128-205-156.ec2.internal",
    "source": {
      "component": "default-scheduler"
    },
   }
 ]
}
```

Get resource

Describe resource

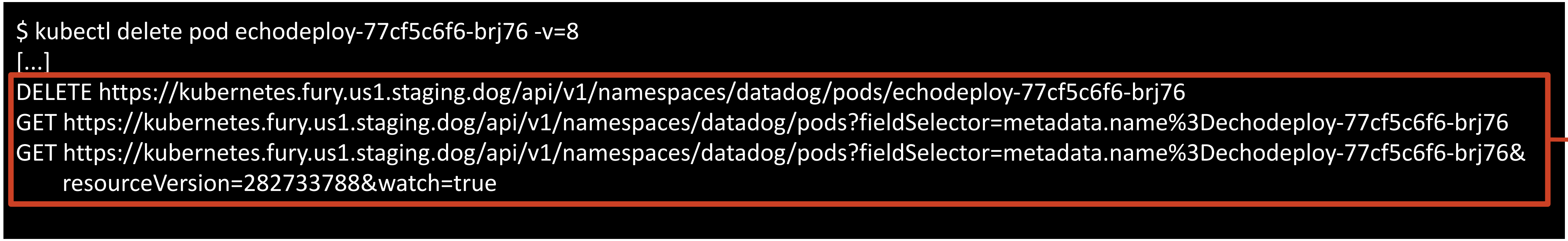


```
kubectl describe pod echodeploy-77cf5c6f6-5wmw9 -v=8
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods/echodeploy-77cf5c6f6-5wmw9
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/events?
        fieldSelector=involvedObject.name=echodeploy-77cf5c6f6-5wmw9,
                         involvedObject.namespace=datadog,
                         involvedObject.uid=770b3a5e-0631-11ea-bc60-12d7306f3c0c
[...]
ResponseBody
{
 "kind": "EventList",
  "items": [
   {
    "involvedObject": { "kind": "Pod", "namespace": "datadog", "name": "echodeploy-77cf5c6f6-5wmw9"},
    "reason": "Scheduled",
    "message": "Successfully assigned echodeploy-77cf5c6f6-5wmw9 to ip-10-128-205-156.ec2.internal",
    "source": {
      "component": "default-scheduler"
    },
   }
 ]
}
```

Get resource

Get **events** associated with resource

What about deletes?



```
$ kubectl delete pod echodeploy-77cf5c6f6-brj76 -v=8
[...]
DELETE https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods/echodeploy-77cf5c6f6-brj76
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods?fieldSelector=metadata.name%3Dechodeploy-77cf5c6f6-brj76
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods?fieldSelector=metadata.name%3Dechodeploy-77cf5c6f6-brj76&
      resourceVersion=282733788&watch=true
```

Delete
+
List &
Watch

Takeaways

- A lot of components are making calls
  - Control plane: controllers, scheduler
  - Node daemons: kubelet, kube-proxy
  - Other controllers: autoscaler, ingress

- “Simple” user ops translate to **many** API calls

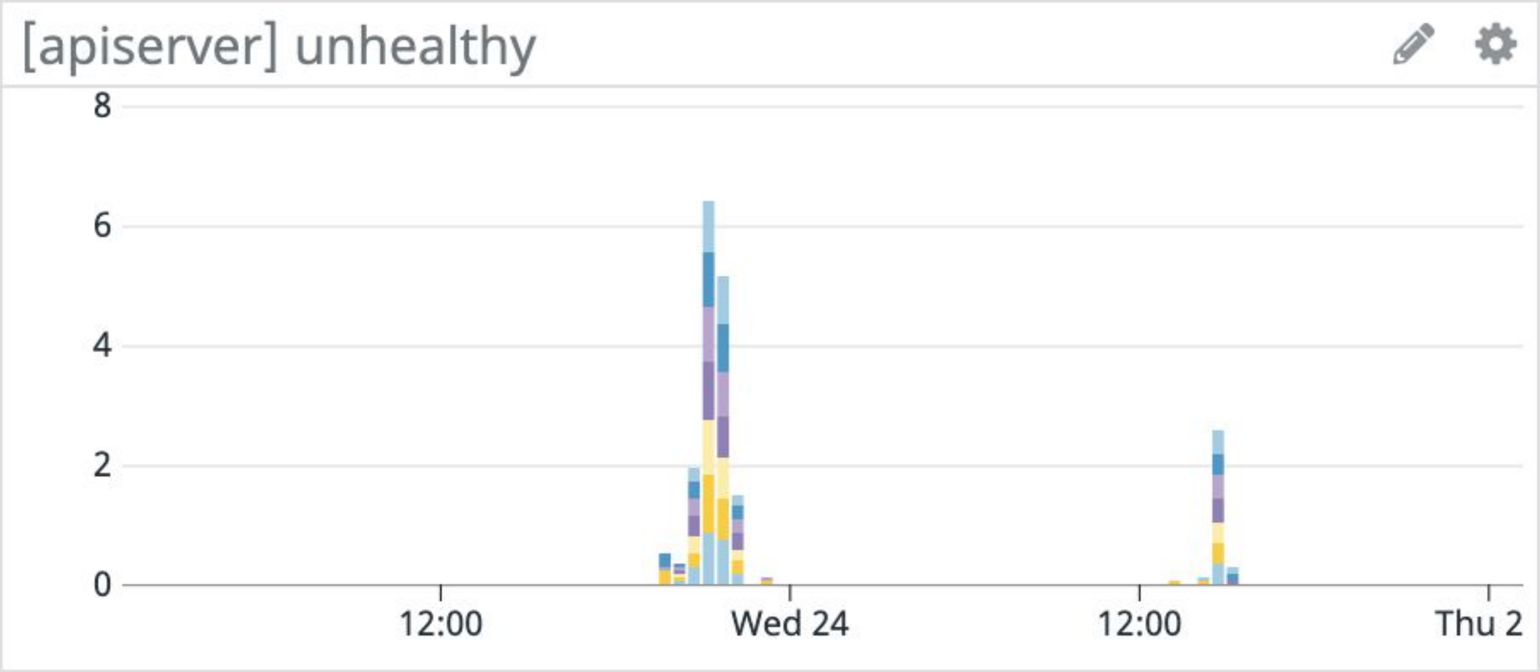
KubeCon | CloudNativeCon

Europe 2023

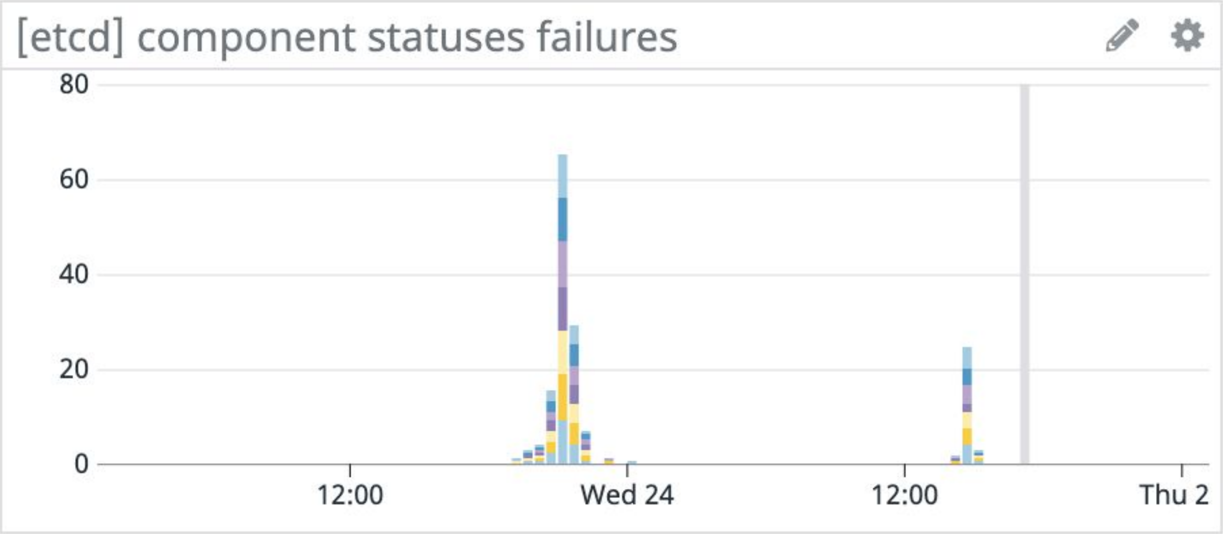
How a single app can overwhelm the control plane

Context

- Users report connectivity issues with a cluster
- Apiservers are not doing well

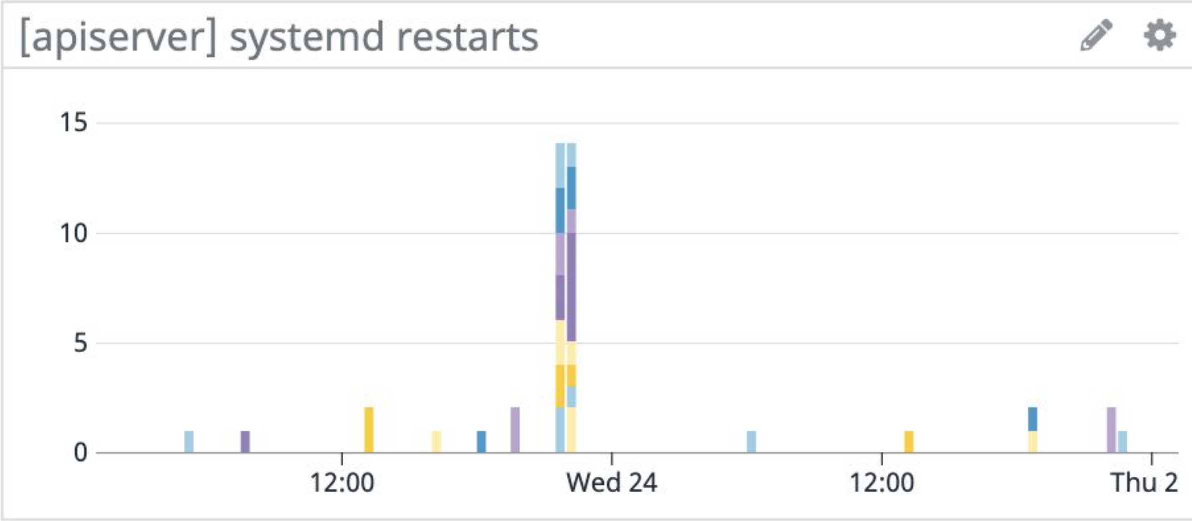


What's with the Apiservers?



[etcd] component statuses failures

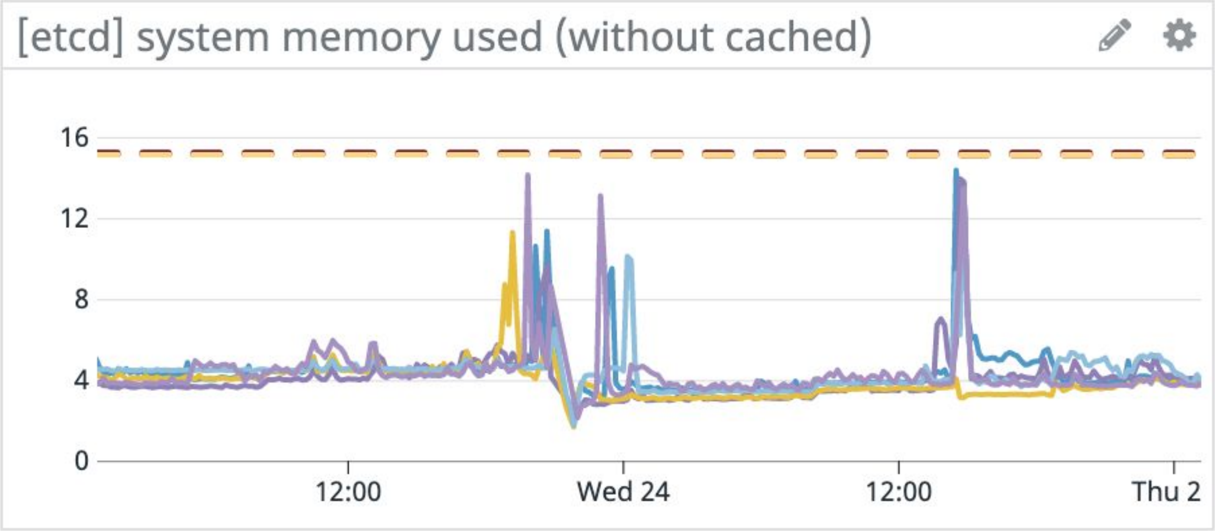
Apiservers can't reach etcd



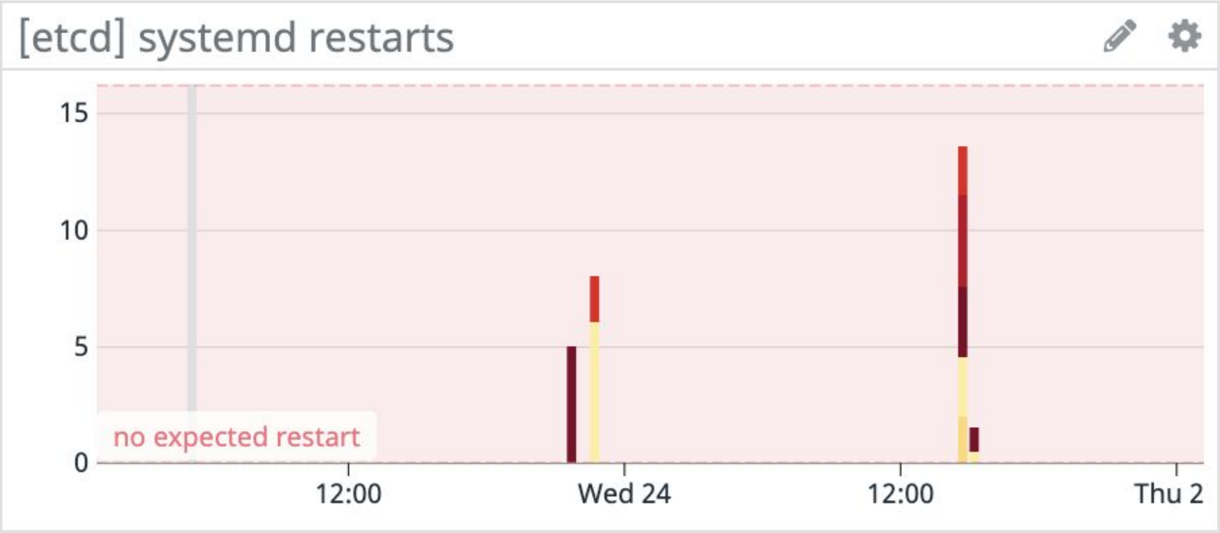
[apiserver] systemd restarts

Apiservers crash/restart

Etcd?


[etcd] system memory used (without cached)

Etcd memory usage is spiking

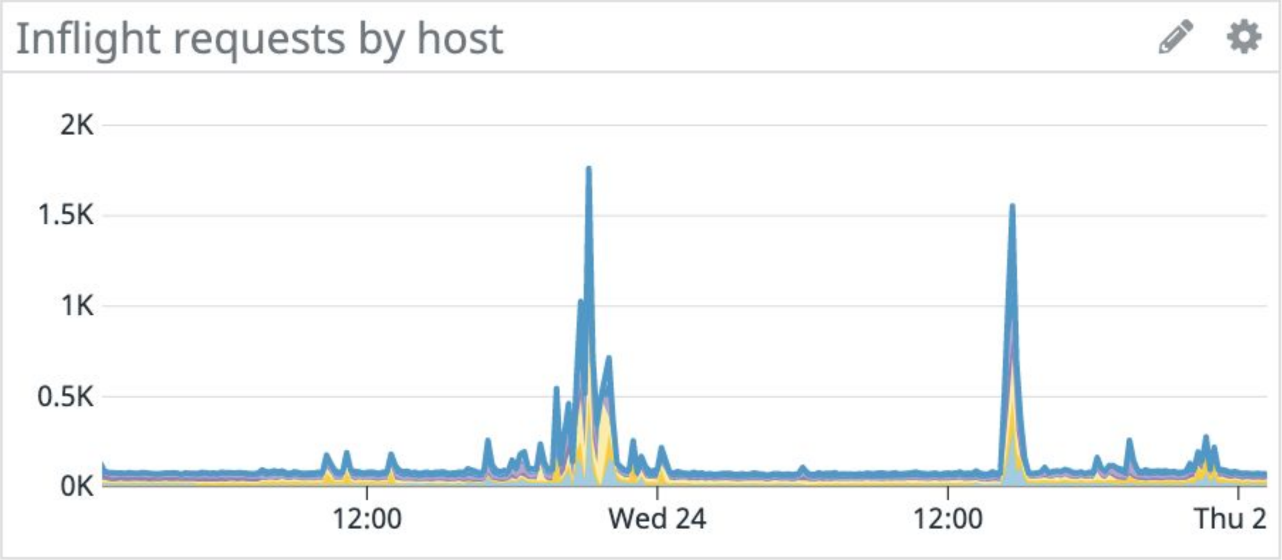

[etcd] systemd restarts

Etcd is oom-killed

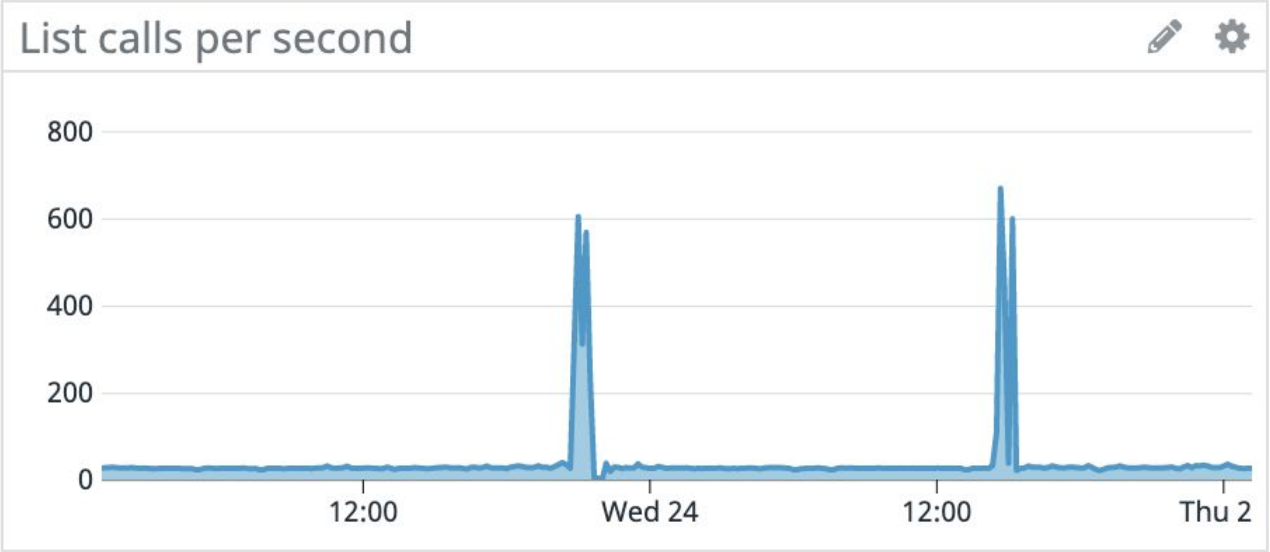
What we know

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

Apiserver requests



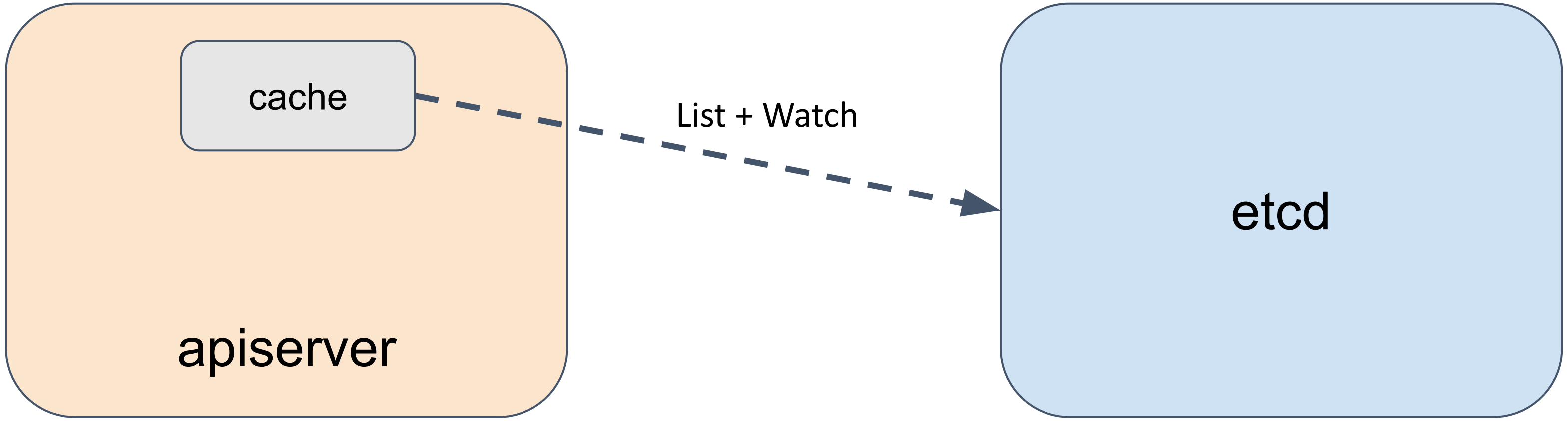
Spikes in inflight requests



Spikes in list calls
(very expensive)

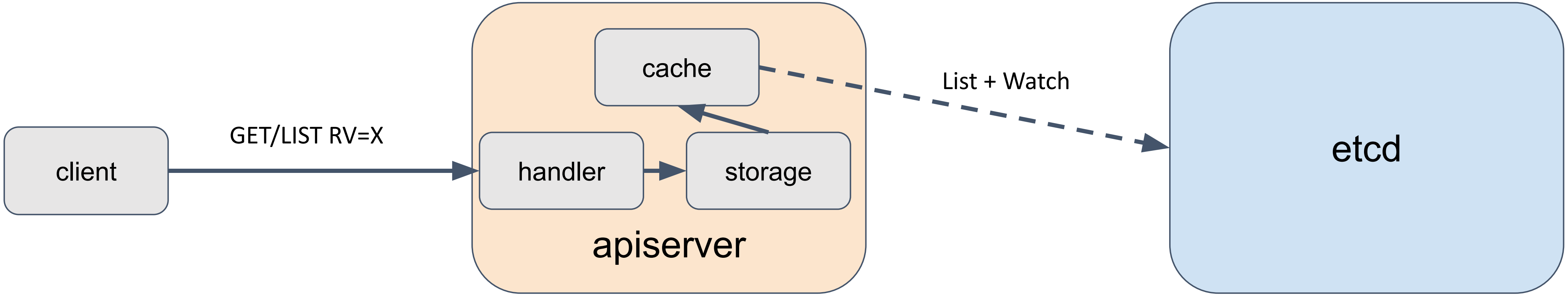
Why are list calls expensive?

Understanding Apiserver caching



Why are list calls expensive?

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**

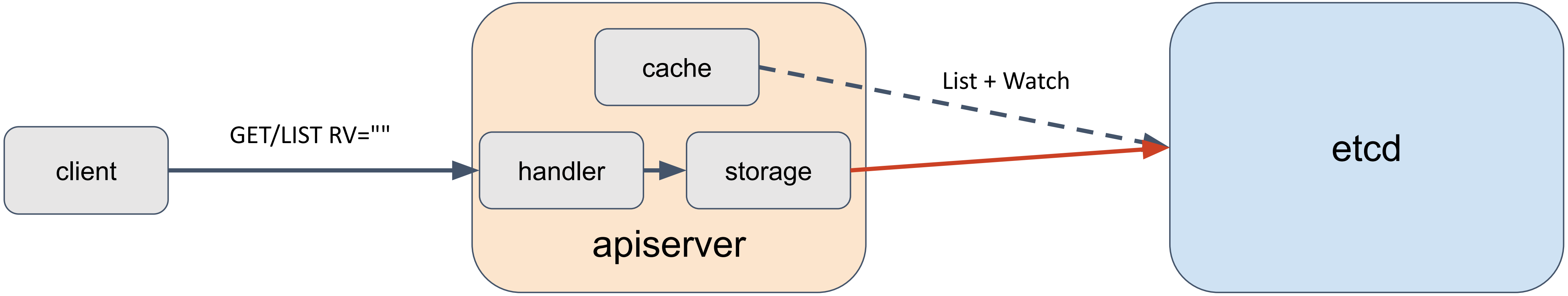
  else **Error**: **"Too large resource version"**

  ➢ *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**

Remember Describe?

```
kubectl describe pod echodeploy-77cf5c6f6-5wmw9 -v=8
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/pods/echodeploy-77cf5c6f6-5wmw9
[...]
GET https://kubernetes.fury.us1.staging.dog/api/v1/namespaces/datadog/events?
        fieldSelector=involvedObject.name=echodeploy-77cf5c6f6-5wmw9,
                          involvedObject.namespace=datadog,
                          involvedObject.uid=770b3a5e-0631-11ea-bc60-12d7306f3c0c
[...]
ResponseBody
{
 "kind": "EventList",
  "items": [
   {
    "involvedObject": { "kind": "Pod", "namespace": "datadog", "name": "echodeploy-77cf5c6f6-5wmw9"},
    "reason": "Scheduled",
    "message": "Successfully assigned echodeploy-77cf5c6f6-5wmw9 to ip-10-128-205-156.ec2.internal",
    "source": {
      "component": "default-scheduler"
    },
   }
 ]
}
```

Get resource

Get **events** associated with resource

RV="" => no cache

=> get namespace events from etcd

=> Filter on apiserver

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**

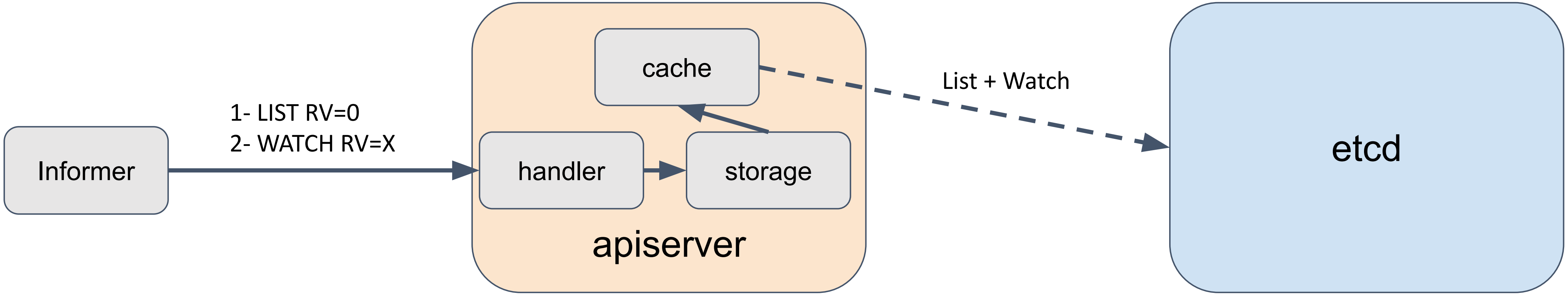
```
curl 'https://cluster.dog/api/v1/pods?labelSelector=app=A'
real    0m3.658s    <== get all pods (30k) in etcd, filter on apiserver

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

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**

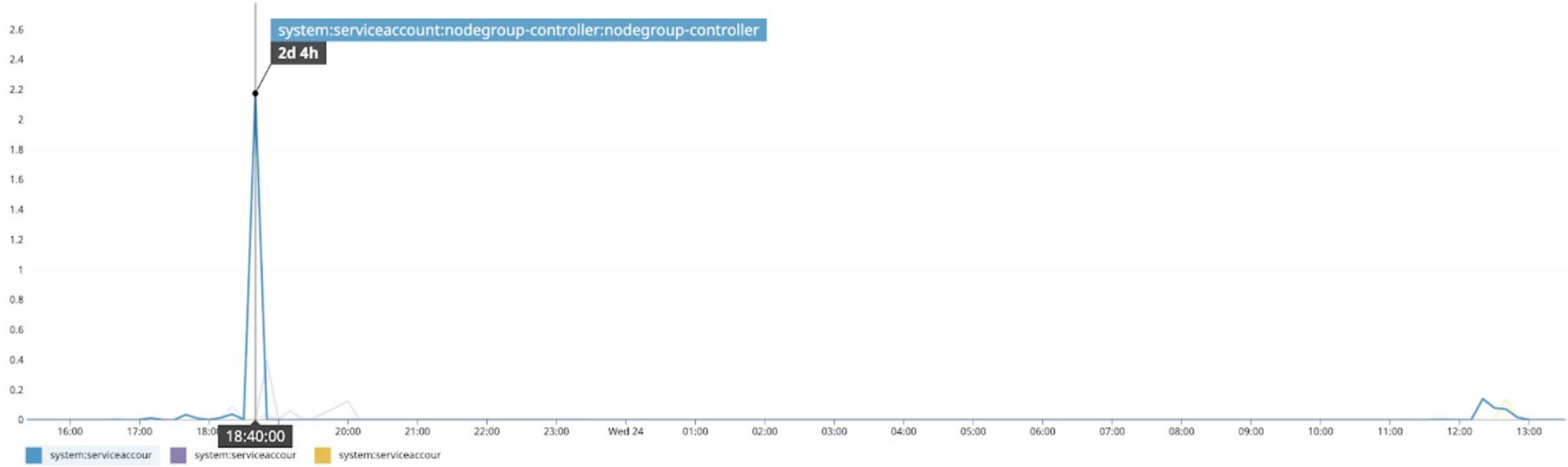
- kubectl get uses LIST with RV=""
- kubectl get could allow setting RV=0
  - Much faster: better user experience
  - Much better for etcd and apiservers
  - Trade-off: small inconsistency window

Back to the incident

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

Audit logs

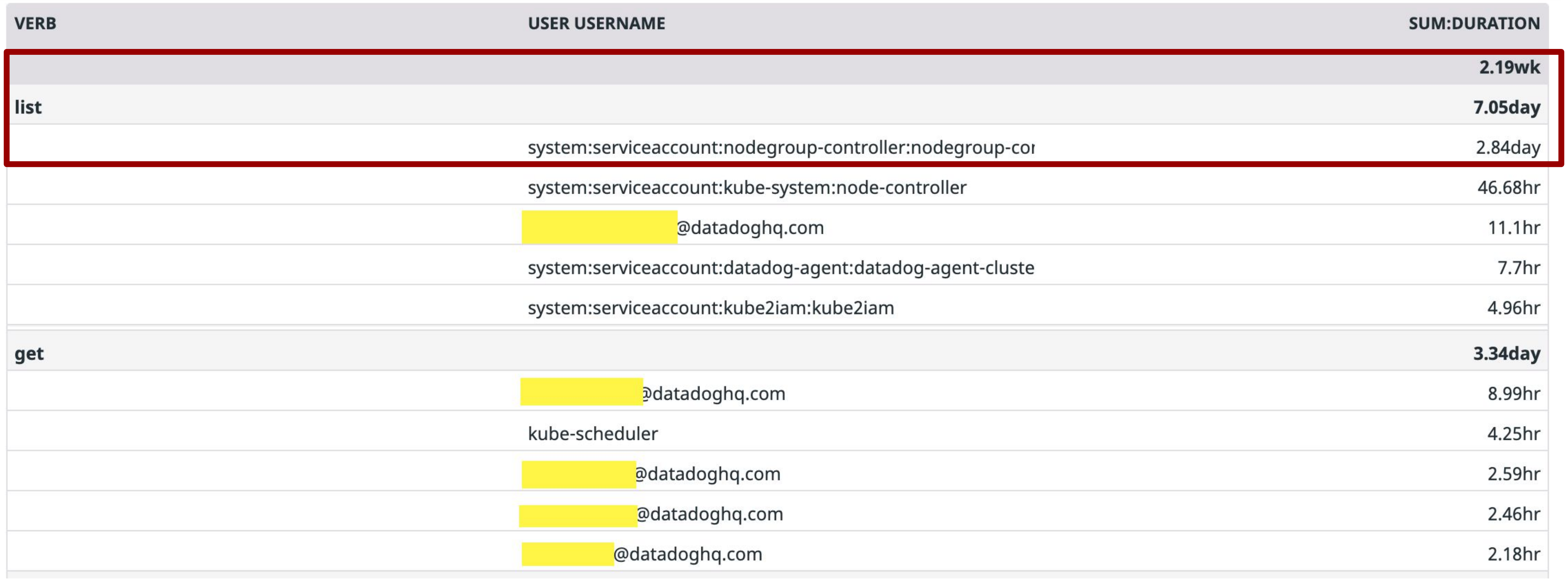
Cumulated query time by user for list calls



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)

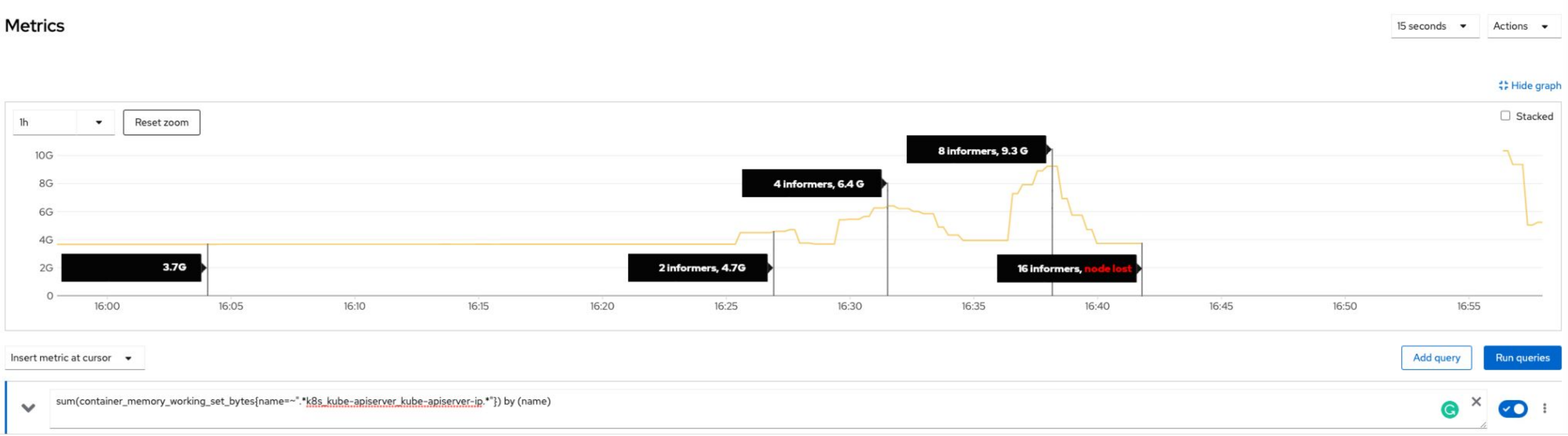
- Audit logs are extremely useful
  - Who did what when?
  - Which users are responsible for processing time

What Kubernetes community does to solve these issues

Streaming lists

- KEP-3157
- Alpha in 1.27
- Before: apiserver holds list response in memory
- Now: apiserver streams list response, significantly reducing memory usage

Streaming lists



3.7G

2 informers, 4.7G

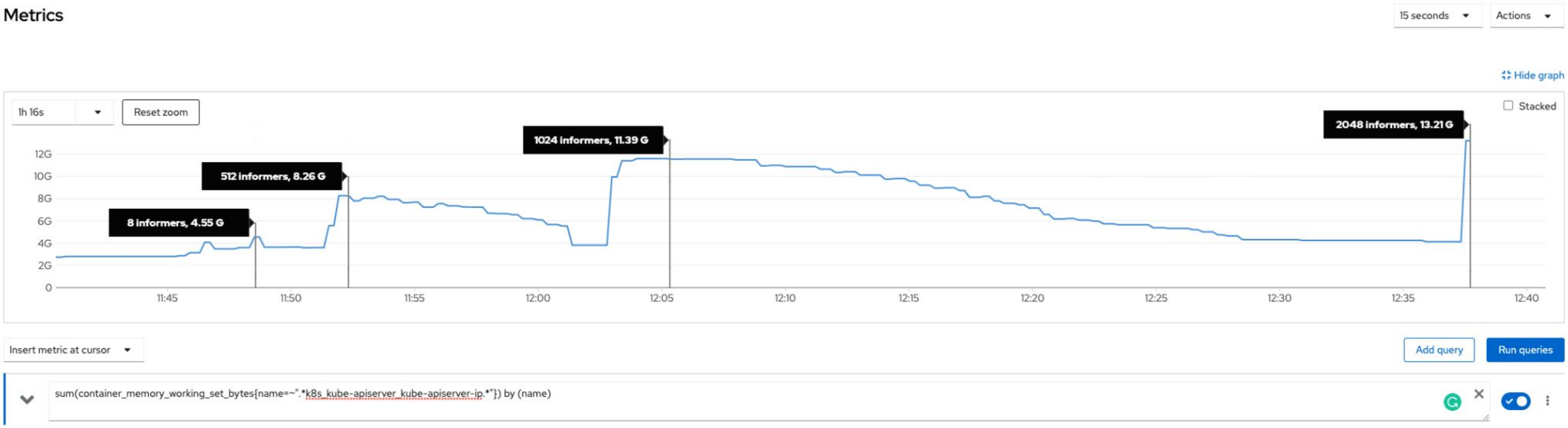
4 informers, 6.4 G

8 informers, 9.3 G

16 informers, node lost

```
sum(container_memory_working_set_bytes{name=~".*k8s_kube-apiserver_kube-apiserver-ip.*"}) by (name)
```

Streaming lists



Metrics

8 informers, 4.55 G

512 informers, 8.26 G

1024 informers, 11.39 G

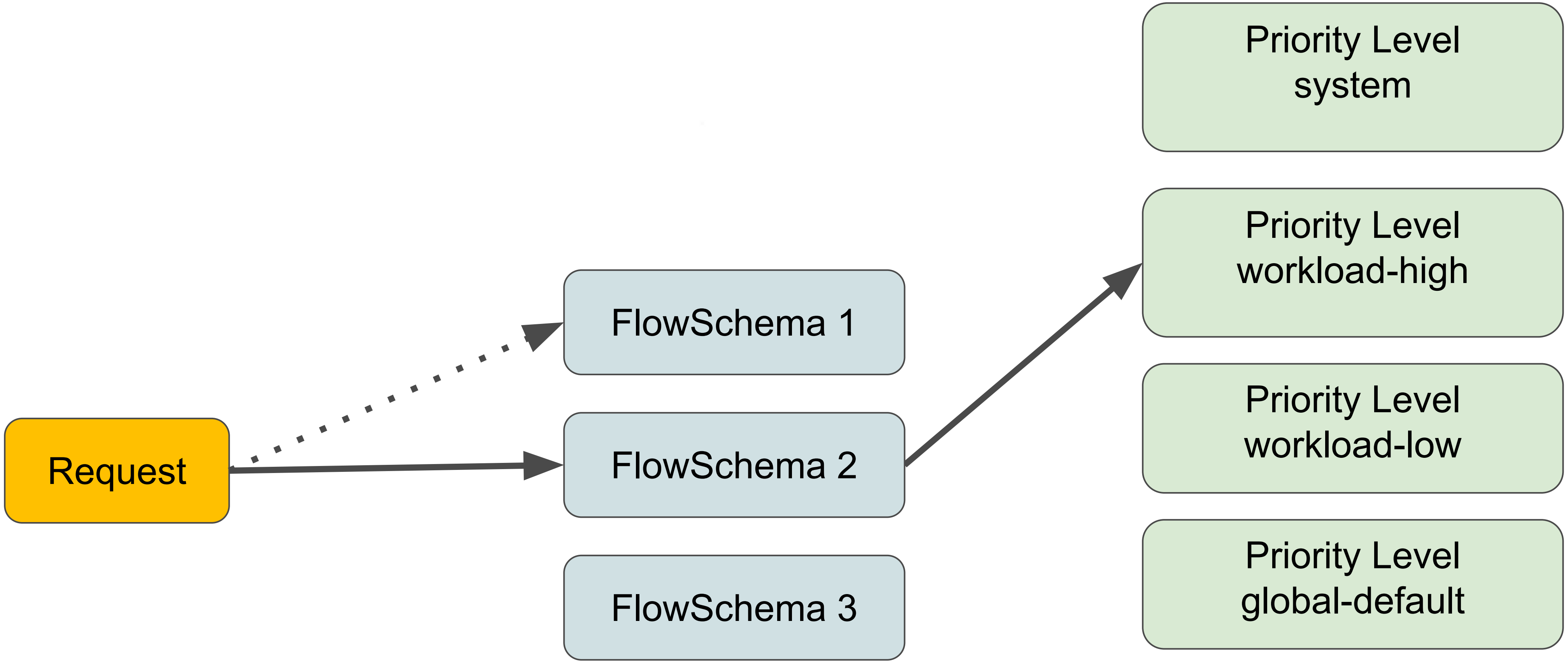
2048 informers, 13.21 G

```
sum(container_memory_working_set_bytes{name=~".*k8s_kube-apiserver_kube-apiserver-ip.*"}) by (name)
```

API Priority and Fairness

- KEP-1040
- Beta since 1.20
- Overload protection for apiserver
- Limits amount of concurrently executed requests
- Ensures fair distribution of throughput across different users within Priority Level

API Priority and Fairness



Request

FlowSchema 1

FlowSchema 2

FlowSchema 3

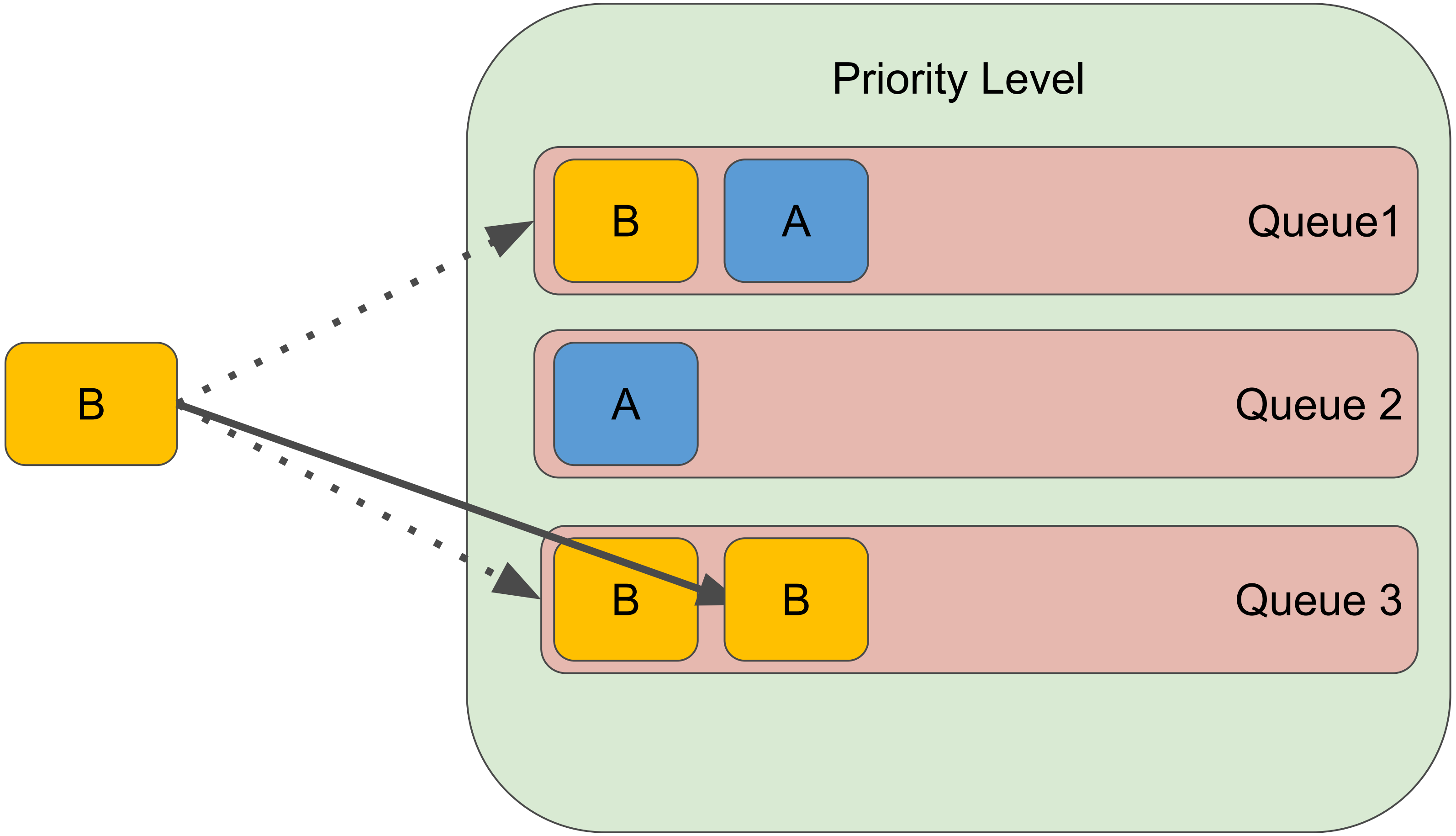
Priority Level system

Priority Level workload-high

Priority Level workload-low

Priority Level global-default

API Priority and Fairness



APF request weights

API requests get different weights:

- Weight can range from 1 to 10
- Simple get request consumes 1 concurrency share
- List requests can consume up to 10 concurrency share based on number of listed elements
- Mutating request can consume up to 10 concurrency shares based on number of watches.

APF default configuration

- Exempt (healthchecks, apiserver loopback requests)
- System (kubelet)
- Leader-election - kube-controller-manager, scheduler, kube-system service accounts
- Workload-high - kube-controller-manager, scheduler
- Workload-low - all service accounts
- Global-default – all others

Example Priority Level Configuration

Workload-high Priority Level:

- 40 Nominal Concurrency Shares
- 128 Queues
- 6 Hand Size
- 50 Queue Limit
- … and 50% of nominal concurrency shares are lendable

APF use case #1

- Misbehaving controller/daemonset etc
- Create new PriorityLevel with small concurrency
- Redirect all request of this component to newly created Priority Level

APF use case #2

- High churn of events
- Create new Priority Level with limited concurrency
- Redirect all event related requests (get/list/watch/create etc) to new Priority Level from all components

Conclusion

Key takeaways

- Running large clusters is still challenging
- Many improvements from the community
- Defaults are not always enough
- Avoid List calls as much as possible

Thank you! Questions?

Please scan the QR Code above
to leave feedback on this session