Kubernetes at Datadog the *very* hard way

Laurent Bernaille and Rob Boll

DATADOG

Table of Contents

Background

Platform challenges

Shooting ourselves in the foot

Future plans

Background

New instance of Datadog

- Completely independent and isolated
- Launch on a different cloud provider
- Fresh start, leave legacy behind

Background

New platform requirements

- Small infra team, high leverage, low touch
- Support for multiple cloud providers
- Self service, API driven, automation friendly
- Meet our scale now, and years from now

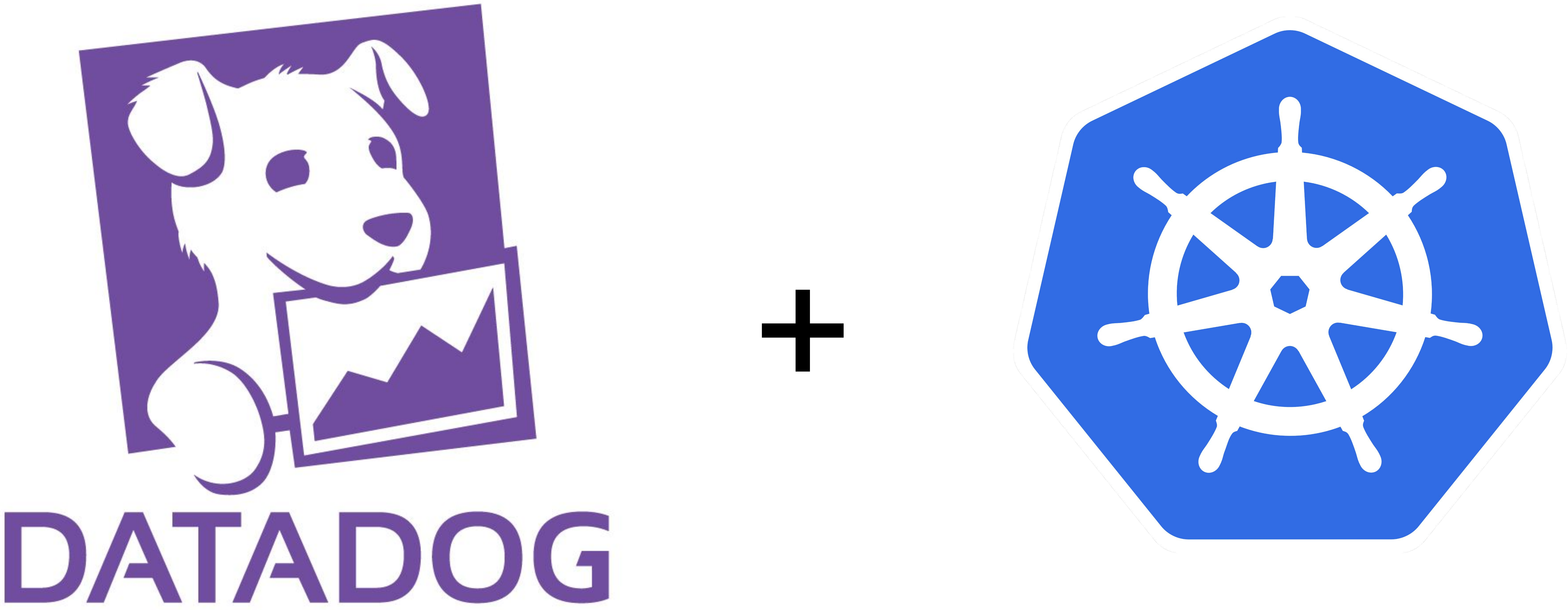
Why Kubernetes?

Kubernetes hits all the requirements

- Extensible & API driven (but *changing fast*)
- Large active community (but *immature*)
- Scalable architecture (but *unproven at scale*)
- Multiple cloud providers supported (*kind of*)

Why Kubernetes?

Dogfooding!



kubernetes_cluster:app3

Group 1150

kubernetes_cluster:app3

Metrics (group avg.)

kubernetes.memory.usage

kubernetes.memory.usage

🟩 3.45M ⬜ 3.76G 7

Hope is not an option

Platform challenges

Certificates

Runtime

Networking

Cloud integrations

Ecosystem

Scale

Certificates

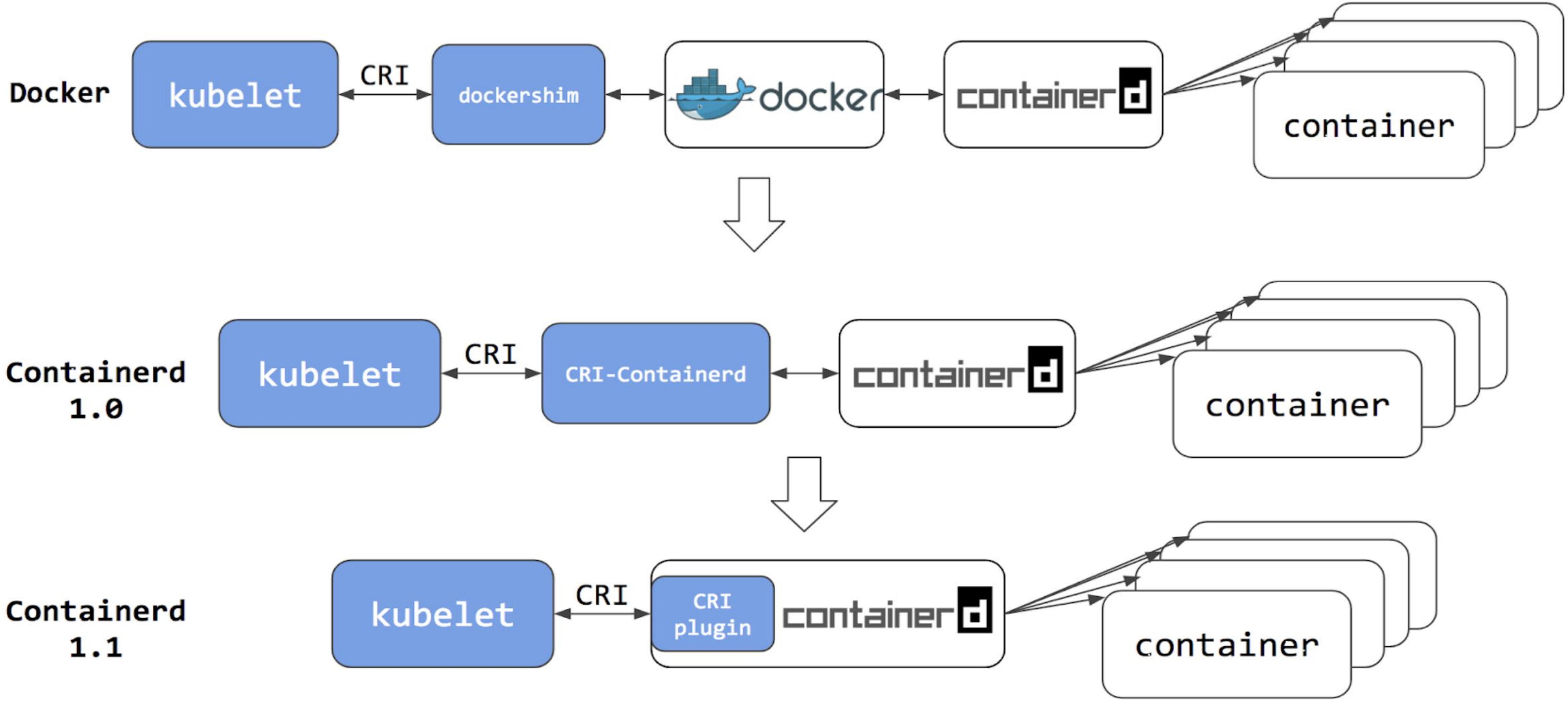
Setup

- Vault + TLS bootstrap
- Refresh certificates every 24h

The fun part

- etcd did not reload certs for client connections using IP addresses
- Kubernetes master components don’t reload certificates
- Flaky bootstraps (vault dependency)
- vault-agent and vault-sidekick

Runtime: containerd



Runtime: containerd

The good

- Lightweight
- Great development team easily accessible

The bad

- Not as battle-tested as docker
- Several small issues
- Many tools assume docker (Datadog agent used to)

The ugly

- Remaining issue: shim sometimes hang and require **kill -9**

Not containerd specific...

/home/kubernetes/bin/health-monitor.sh

Running on all GKE instances

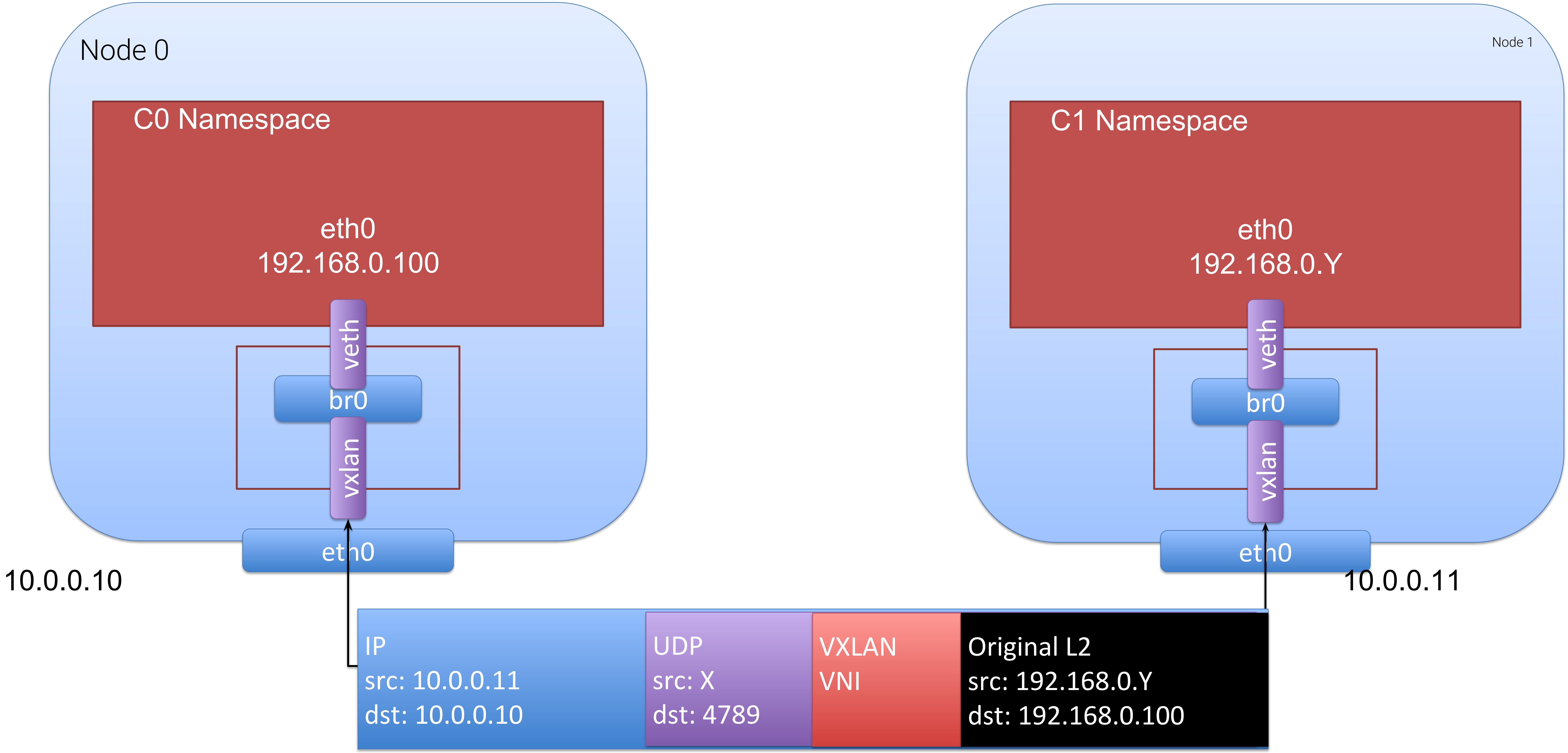
```
# We simply kill the process when there is a failure. Another systemd service will
# automatically restart the process.
function docker_monitoring {
  while [ 1 ]; do
    if ! timeout 60 docker ps > /dev/null; then
      echo "Docker daemon failed!"
      pkill docker
      # Wait for a while, as we don't want to kill it again before it is really up.
      sleep 120
    else
      sleep "${SLEEP_SECONDS}"
    fi
  done
}

function kubelet_monitoring {
  echo "Wait for 2 minutes for kubelet to be functional"
  # TODO(andyzheng0831): replace it with a more reliable method if possible.
  sleep 120
  local -r max_seconds=10
  local output=""
  while [ 1 ]; do
    if ! output=$(curl -m "${max_seconds}" -f -s -S http://127.0.0.1:10255/healthz 2>&1); then
      # Print the response and/or errors.
      echo $output
      echo "Kubelet is unhealthy!"
      pkill kubelet
      # Wait for a while, as we don't want to kill it again before it is really up.
      sleep 60
    else
      sleep "${SLEEP_SECONDS}"
    fi
  done
}
```

<= Restart docker if docker ps hangs 60s

<= Restart kubelet if healthz takes 10+s

Networking: Overlays



Node 0

C0 Namespace

eth0
192.168.0.100

veth

br0

vxlan

eth0

10.0.0.10

Node 1

C1 Namespace

eth0
192.168.0.Y

veth

br0

vxlan

eth0

10.0.0.11

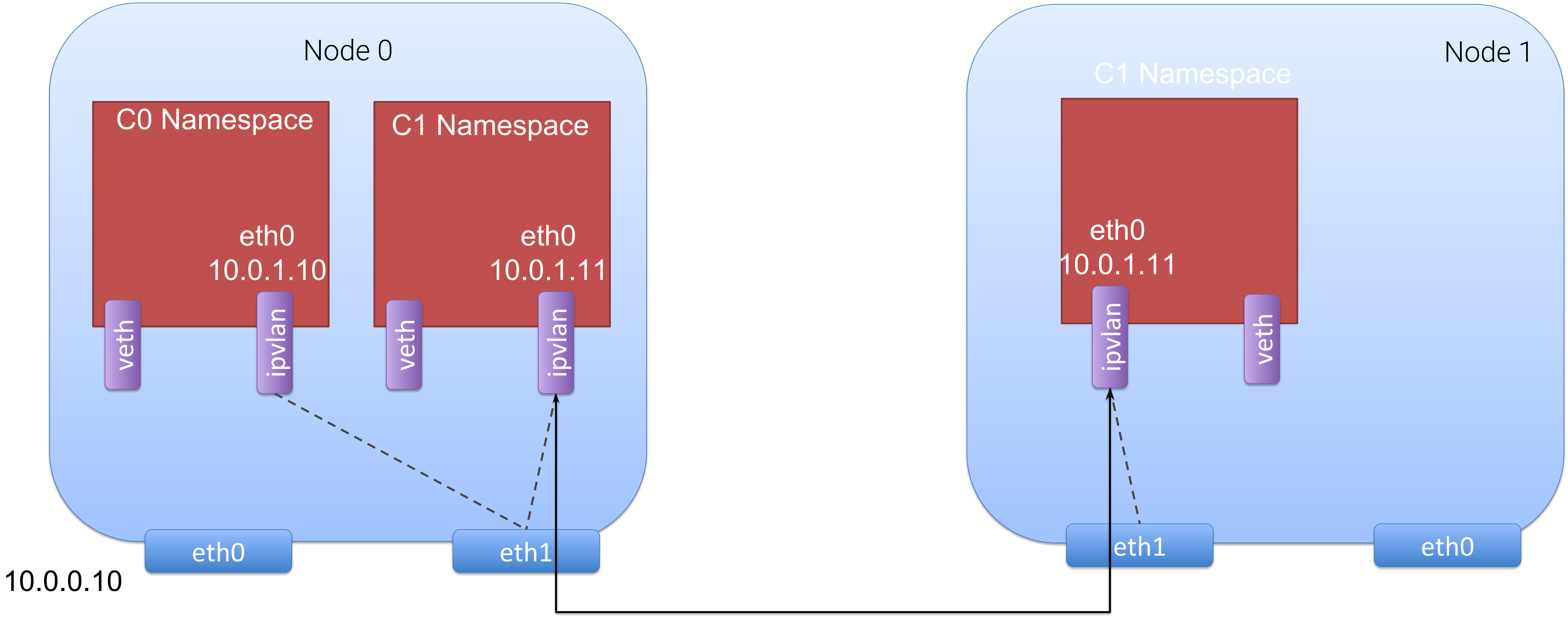
IP
src: 10.0.0.11
dst: 10.0.0.10

UDP
src: X
dst: 4789

VXLAN
VNI

Original L2
src: 192.168.0.Y
dst: 192.168.0.100

Networking: Native pod routing



Networking: Native pod routing

Objectives

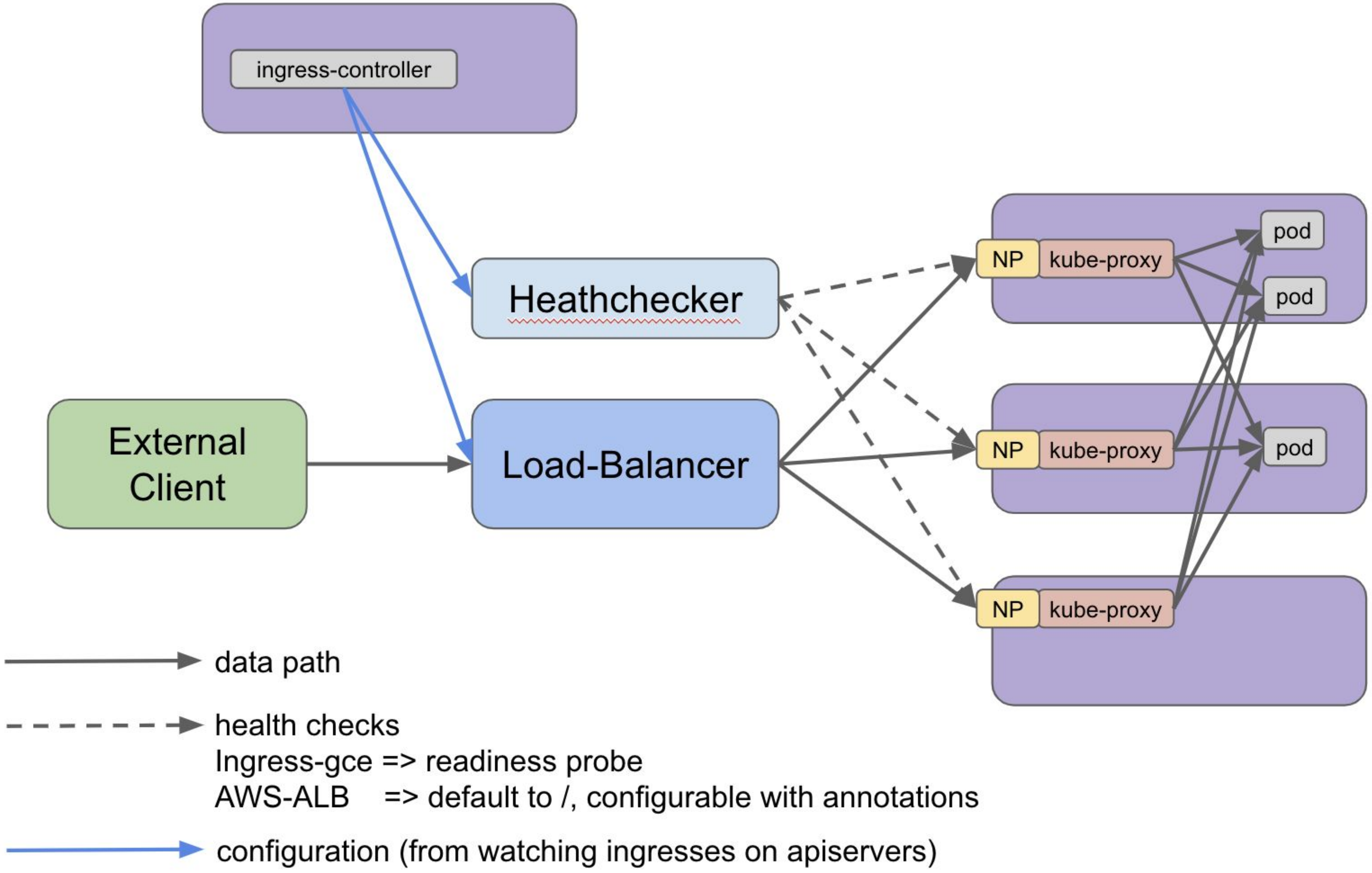
- Avoid overlays
- Avoid bridges (PTP or IPVLAN)
- Route from non kubernetes hosts/between clusters

Challenges

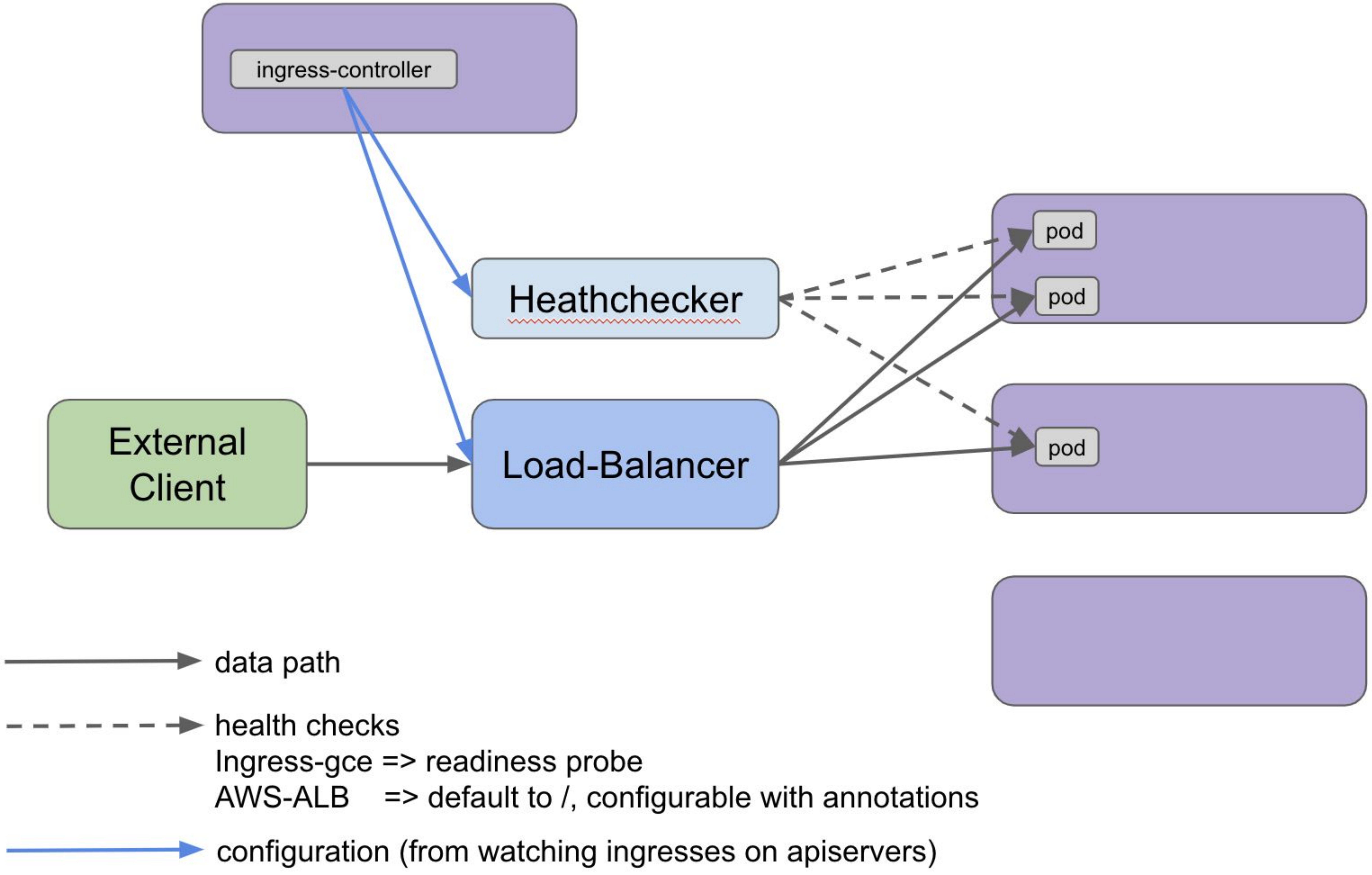
- Beta features
- Young CNI plugins
  - Bugs (Nodeports, inconsistent metadata)
  - Much more complicated to debug

➢ Good relationship with developers of the Lyft plugins

Networking: ingresses, default


ingress-controller
Heathchecker
External Client
Load-Balancer
NP
kube-proxy
pod
pod
NP
kube-proxy
pod
NP
kube-proxy
data path
health checks
Ingress-gce => readiness probe
AWS-ALB => default to /, configurable with annotations
configuration (from watching ingresses on apiservers)

Ingresses: native pod routing



Ingress-gce => readiness probe
AWS-ALB => default to /, configurable with annotations

Ingresses: native pod routing

More efficient

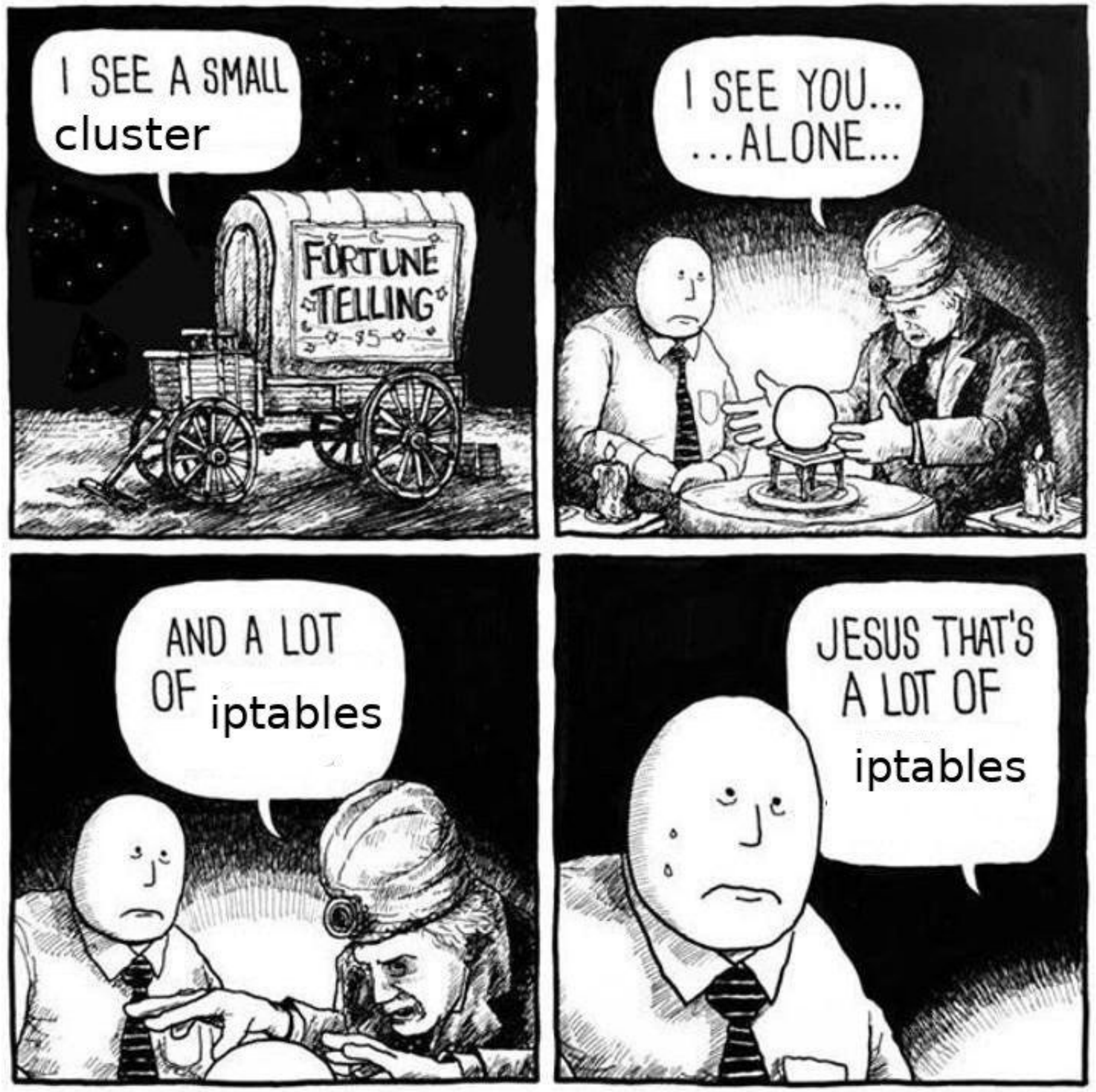
- No need to add all nodes to load-balancers
- Simpler data path

Not that simple

- Very recent
- A few bugs (instable cloud provider features, single CIDR VPC)

➢ Getting fixes upstream was quite easy

Networking: kube-proxy



IPVS instead of iptables

Sounded too good to be true

- Faster (traffic and refresh)
- Cleaner (almost no iptables rules)

Was too good to be true

- Unable to access services from the host
- Regression in 1.11 breaking ExternalTrafficPolicy: Local
- No localhost:nodeport
- No graceful termination

➢ Very good relationship with the Huawei team

➢ Almost everything has been fixed, working great so far

Networking: IPV6 and DNS

“Sometimes my DNS queries take more than 5s/time out”
“Yeah right”
[narrator]: “Well *actually*...”

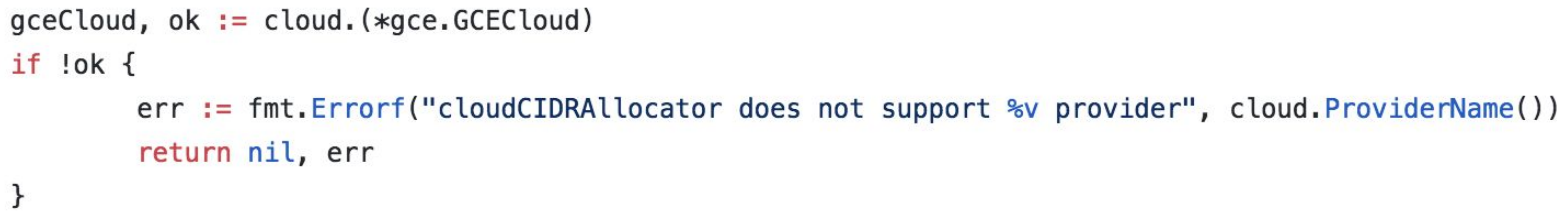
- ➢ Race condition in the conntrack code
- ➢ We disabled IPV6 in the kernel
- ➢ Also had to disable native Go resolution

Cloud integrations

Many small edge-cases

- Different Load-balancer behaviors
- Magically disappearing instances (zones / instance state)
- Some “standard” controller config like “cidr-allocator-type”

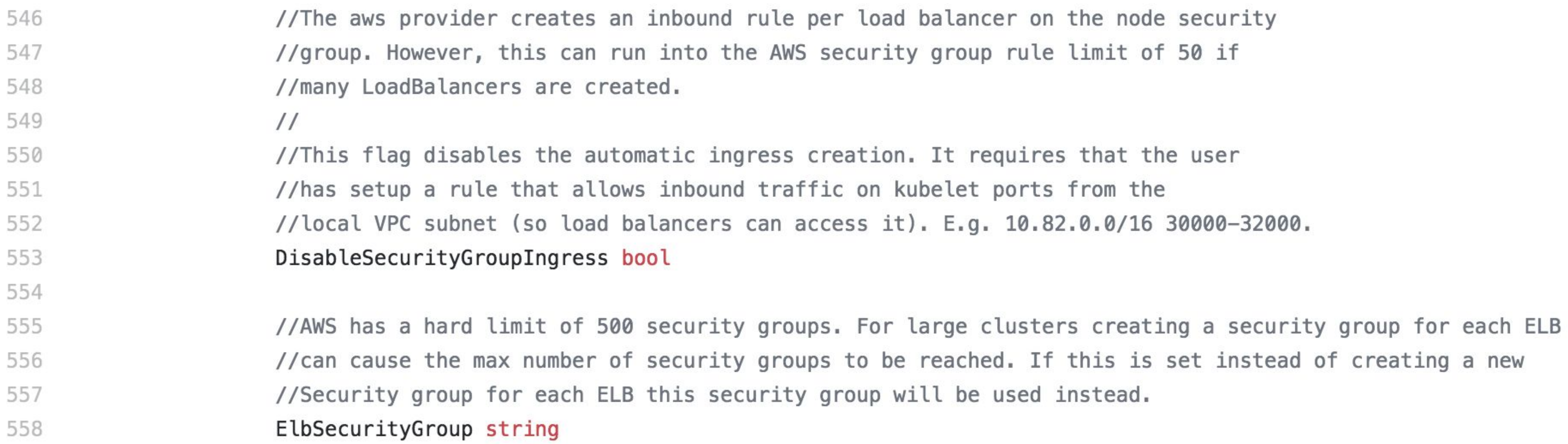
controller/nodeipam/ipam/cloud_cidr_allocator.go



```
gceCloud, ok := cloud.(*gce.GCECloud)
if !ok {
	err := fmt.Errorf("cloudCIDRAllocator does not support %v provider", cloud.ProviderName())
	return nil, err
}
```

(almost) No doc

providers/aws/aws.go



```
//The aws provider creates an inbound rule per load balancer on the node security
//group. However, this can run into the AWS security group rule limit of 50 if
//many LoadBalancers are created.
//
//This flag disables the automatic ingress creation. It requires that the user
//has setup a rule that allows inbound traffic on kubelet ports from the
//local VPC subnet (so load balancers can access it). E.g. 10.82.0.0/16 30000-32000.
DisableSecurityGroupIngress bool

//AWS has a hard limit of 500 security groups. For large clusters creating a security group for each ELB
//can cause the max number of security groups to be reached. If this is set instead of creating a new
//Security group for each ELB this security group will be used instead.
ElbSecurityGroup string
```

Ecosystem

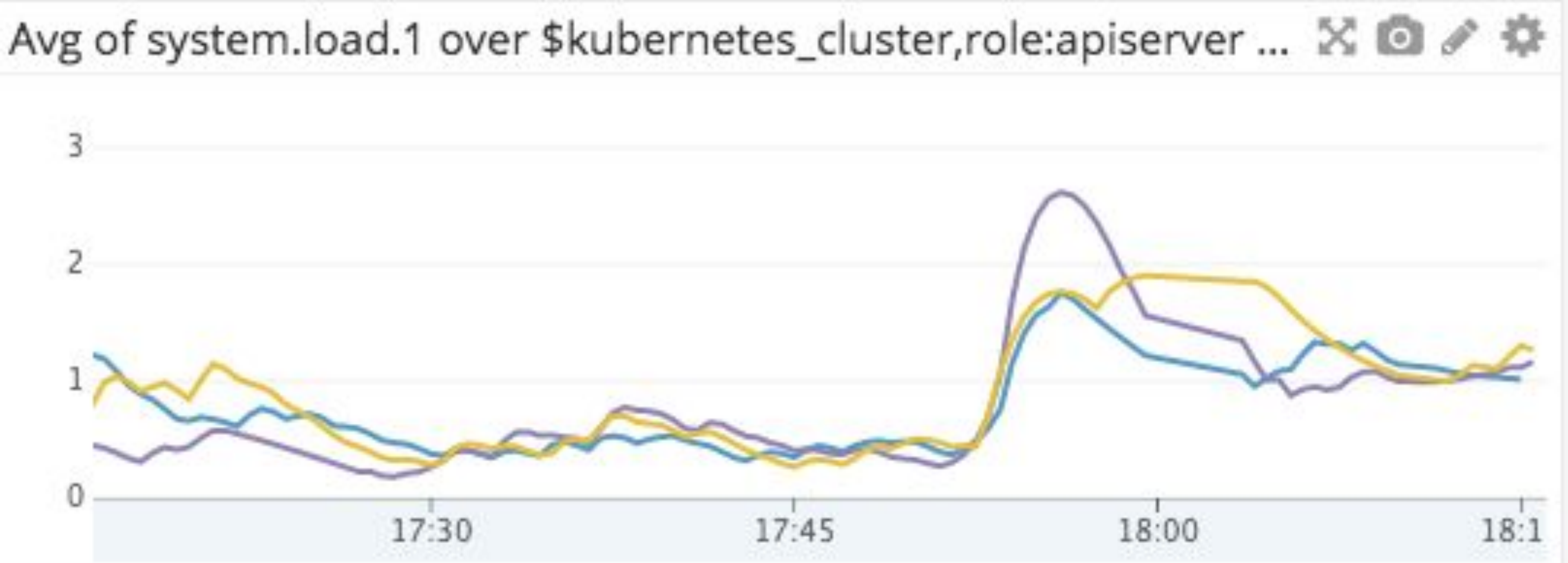
Good news

- Very dynamic
- Usually easy to get PRs merged

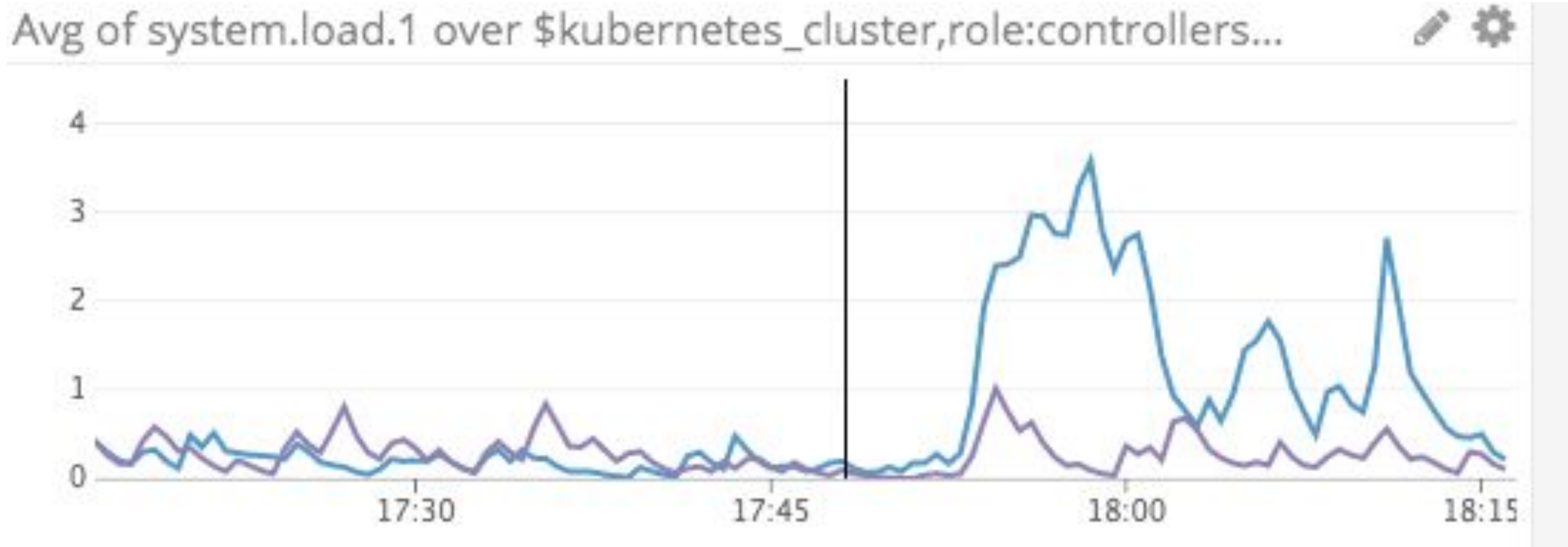
Bad news

- Not heavily tested / Limited to basic setup
- (very) Limited doc: localVolumeProvisioner and mount paths
- **Almost never tested on large clusters**
  - cluster-autoscaler doesn’t work with more than 50 ASGs
  - we abandoned kubed when its memory usage reached 10+GB
  - metrics-server: doesn’t start with a single NotReady node
  - kube-state-metrics: pod/node collector generates ~100MB payloads
  - voyager: map sort bug lead to continuous pod recreation

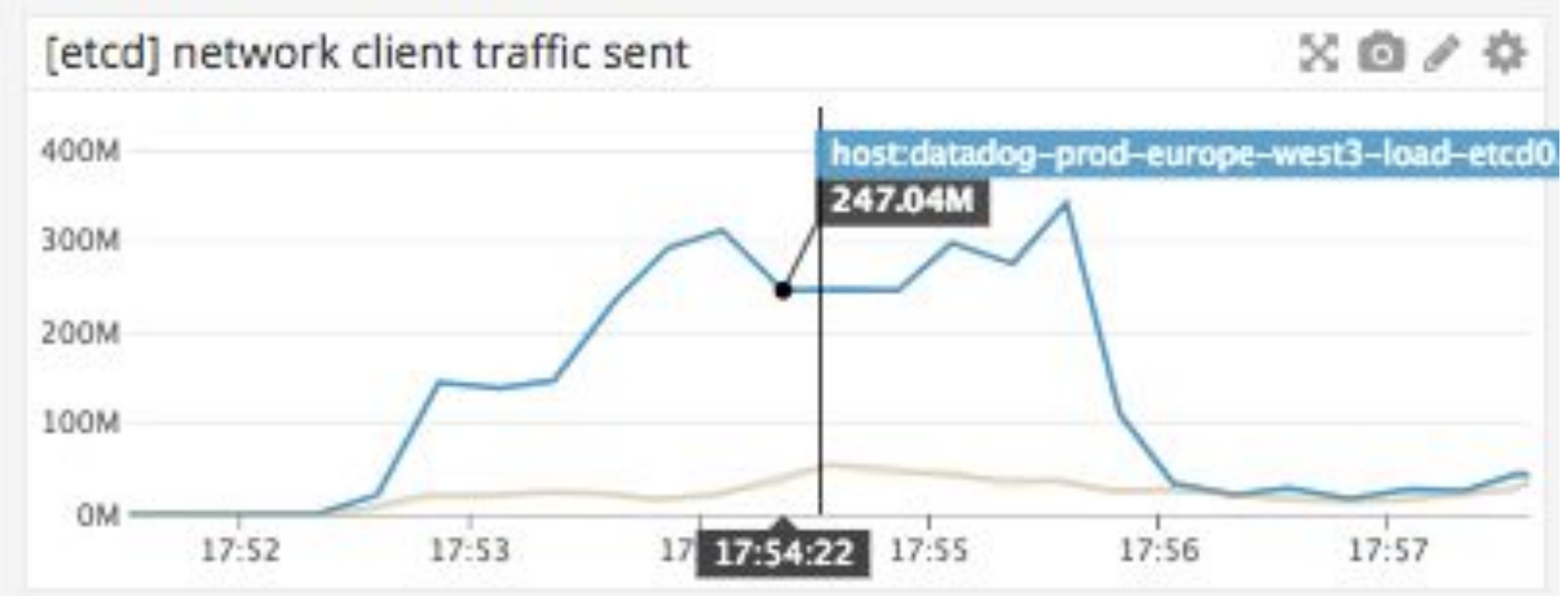
Scaling nodes: 100 => 1000



API server : High load but ok
Careful with File descriptors, CPU, Memory
TargetRAM helped a lot (avoid OOM)



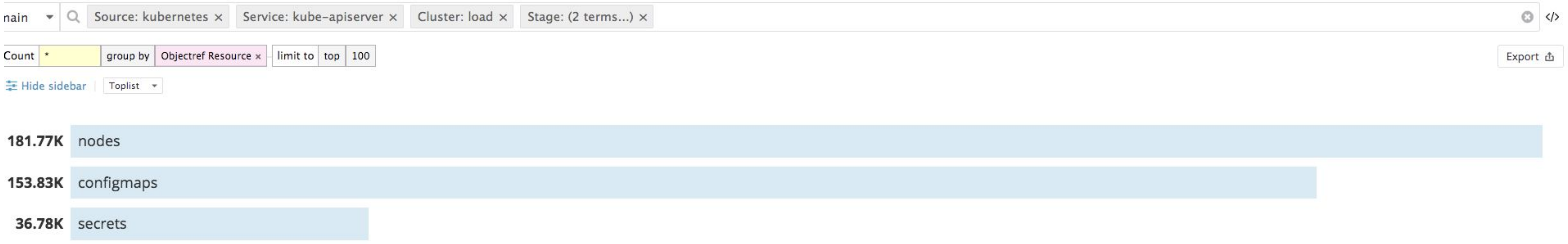
Controller/Scheduler with high load too
Competing for CPU => split
Thinking about splitting controllers but this is hard/impossible



etcd imbalance => shuffle etcd endpoints on API servers
works pretty well, alternatives were a bit scary (gRPC proxy)

Scaling nodes: 100 => 1000

Routes on API servers (registration + daemonsets)



nain ▾ Source: kubernetes × Service: kube-apiserver × Cluster: load × Stage: (2 terms...) ×

Count * group by Objectref Resource × limit to top 100

181.77K nodes

153.83K configmaps

36.78K secrets

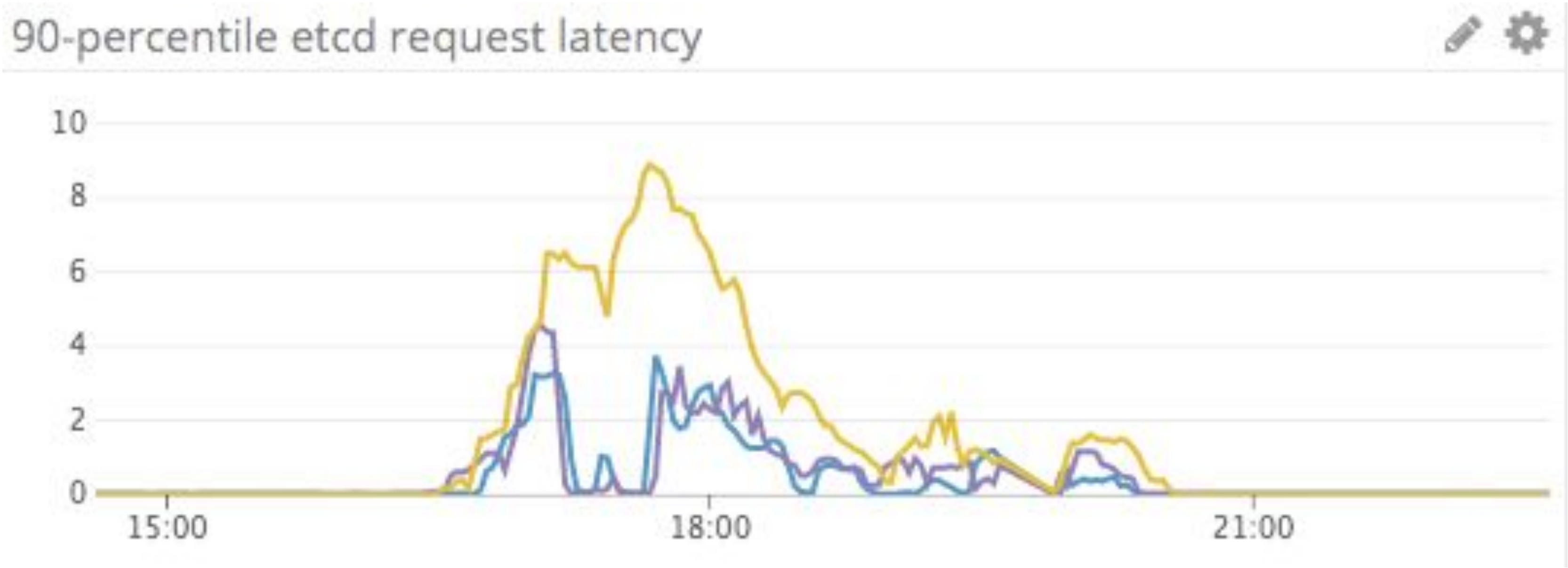
CoreDNS issues

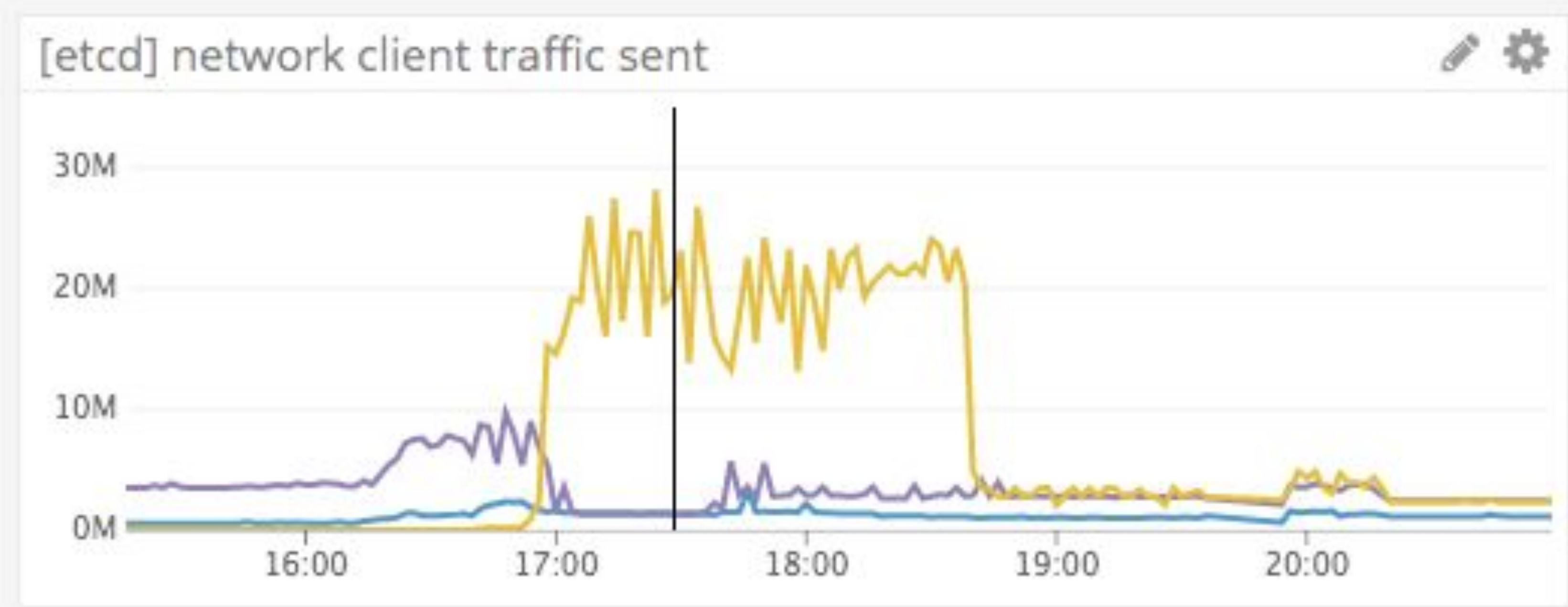
- Not enough nodes for coredns pods: "nodesPerReplica":16
- Memory limits leading to OOMkills (mem usage with “pods: verified”)

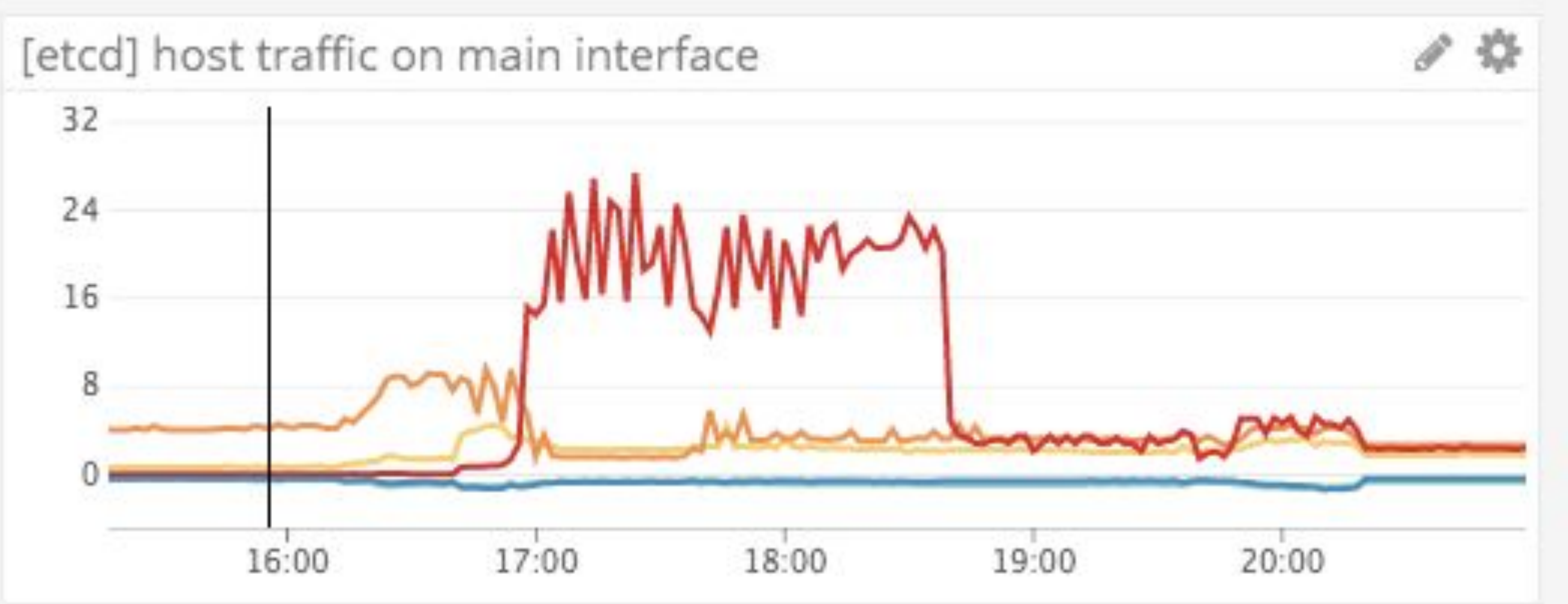
```
NAME                        READY    STATUS              RESTARTS   AGE
coredns-7b4d675999-22d4q    0/1      CrashLoopBackOff    40         6h
coredns-7b4d675999-2lt5w    0/1      ImagePullBackOff    135        17h
coredns-7b4d675999-45s94    0/1      CrashLoopBackOff    41         6h
coredns-7b4d675999-4dfbt    0/1      CrashLoopBackOff    140        17h
```

Scale: create 200 deployments

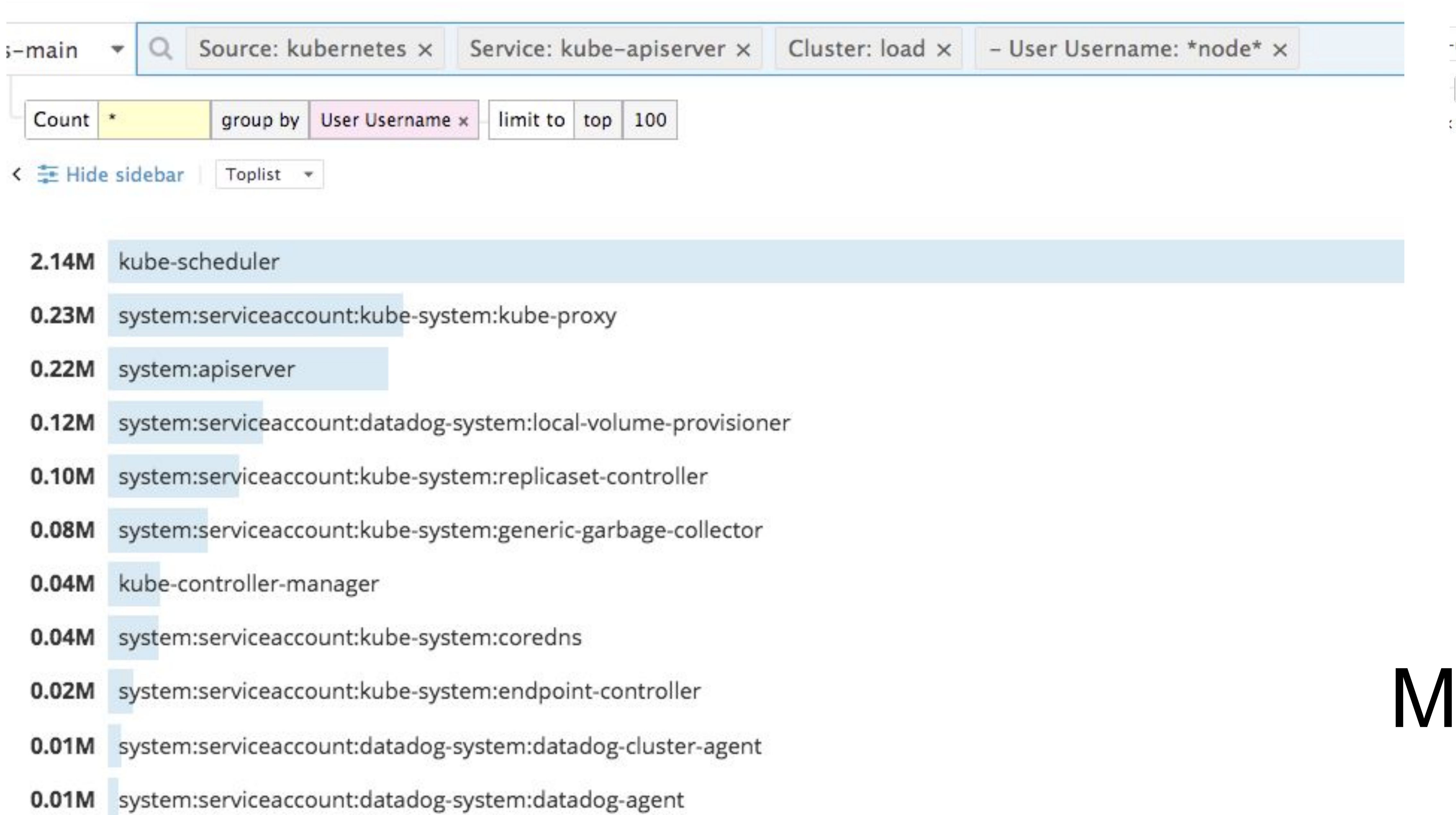
Very hard on all components (apiservers, controller, scheduler, etcd)







Maxed-out the scheduler QPS (too many tunables)

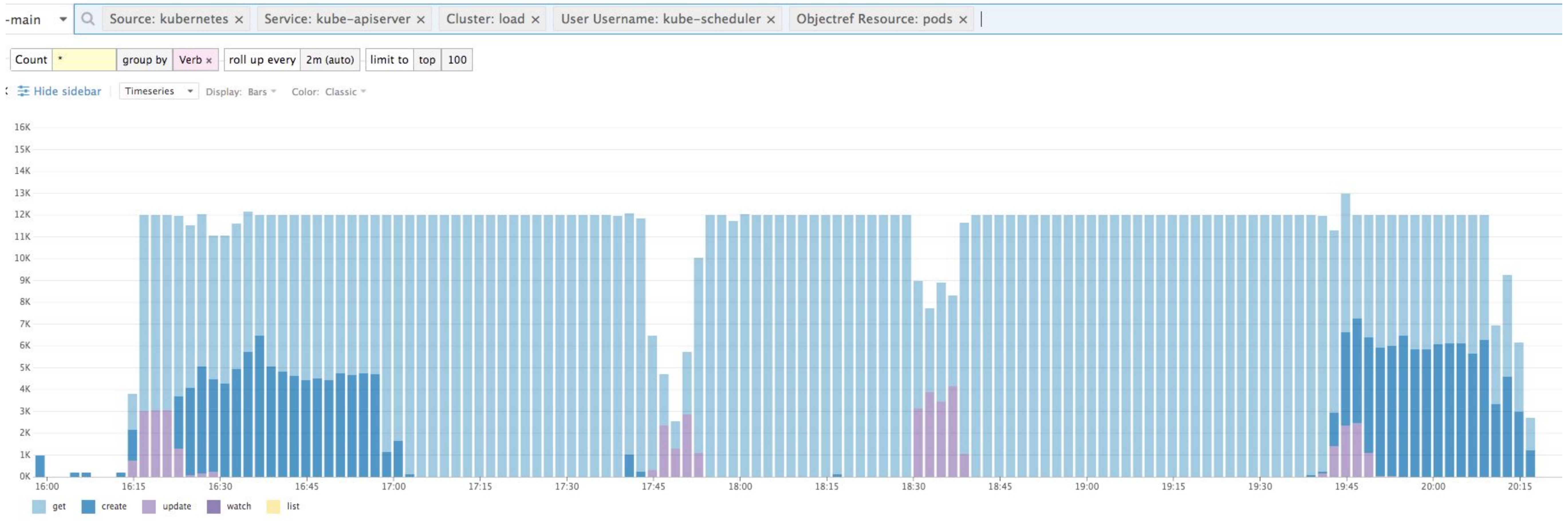


-main ▾ Source: kubernetes × Service: kube-apiserver × Cluster: load × – User Username: *node* ×

Count * group by User Username × limit to top 100

Hide sidebar Toplist ▾

2.14M	kube-scheduler
0.23M	system:serviceaccount:kube-system:kube-proxy
0.22M	system:apiserver
0.12M	system:serviceaccount:datadog-system:local-volume-provisioner
0.10M	system:serviceaccount:kube-system:replicaset-controller
0.08M	system:serviceaccount:kube-system:generic-garbage-collector
0.04M	kube-controller-manager
0.04M	system:serviceaccount:kube-system:coredns
0.02M	system:serviceaccount:kube-system:endpoint-controller
0.01M	system:serviceaccount:datadog-system:datadog-cluster-agent
0.01M	system:serviceaccount:datadog-system:datadog-agent



-main ▾ Source: kubernetes × Service: kube-apiserver × Cluster: load × User Username: kube-scheduler × Objectref Resource: pods ×

Count * group by Verb × roll up every 2m (auto) limit to top 100

Hide sidebar Timeseries ▾ Display: Bars ▾ Color: Classic ▾

Main issues: apiserver sizing and traffic imbalance

Footguns

DaemonSets

StatefulSets

Cargo culting

Zombies

Containers, not VMs

Rolling updates

OOM

InitContainers

Native resources / external config

Manual changes

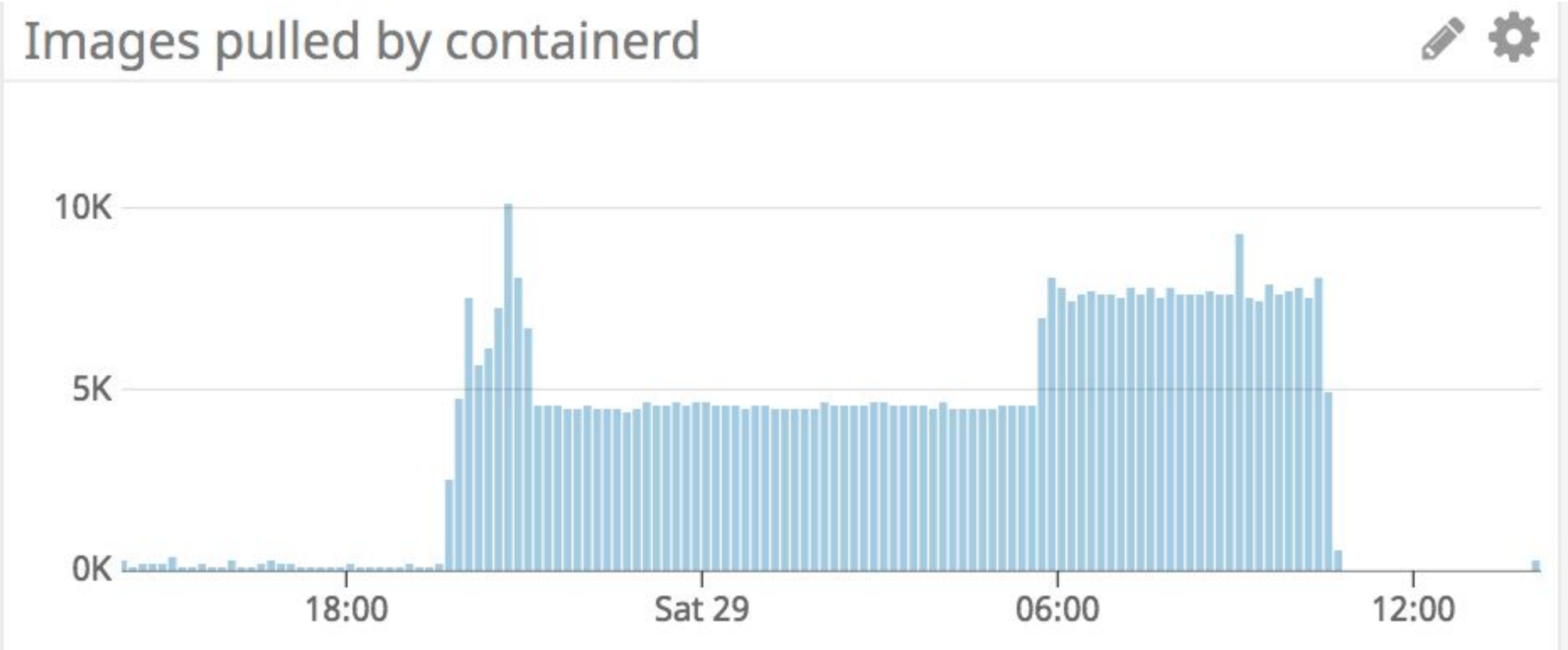
Daemonsets

High DDOS risk

- APIservers, vault
- Cloud provider API rate-limits
- Very slow or very dangerous

Pod scheduling

- Stuck rollout
- 1.12 scheduler alpha...



Images pulled by containerd

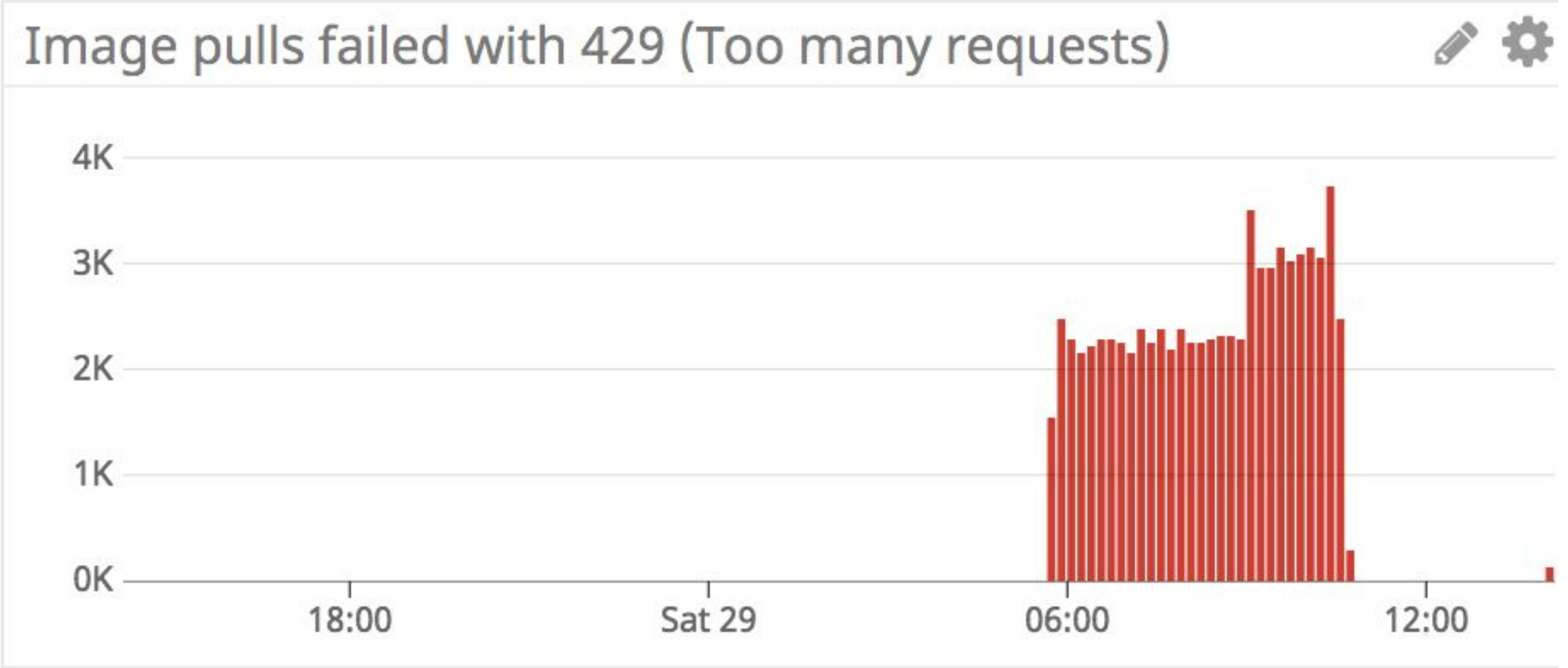


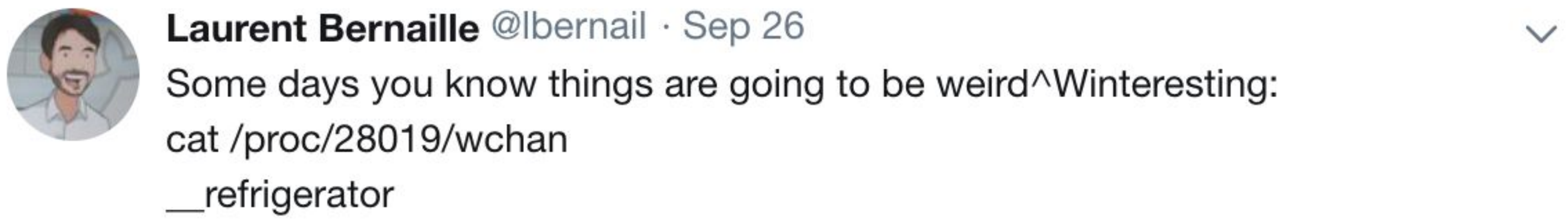
Image pulls failed with 429 (Too many requests)

app broke due to permission change
container in restart loop
ImagePullPolicy: Always

Stateful sets

Persistent volumes

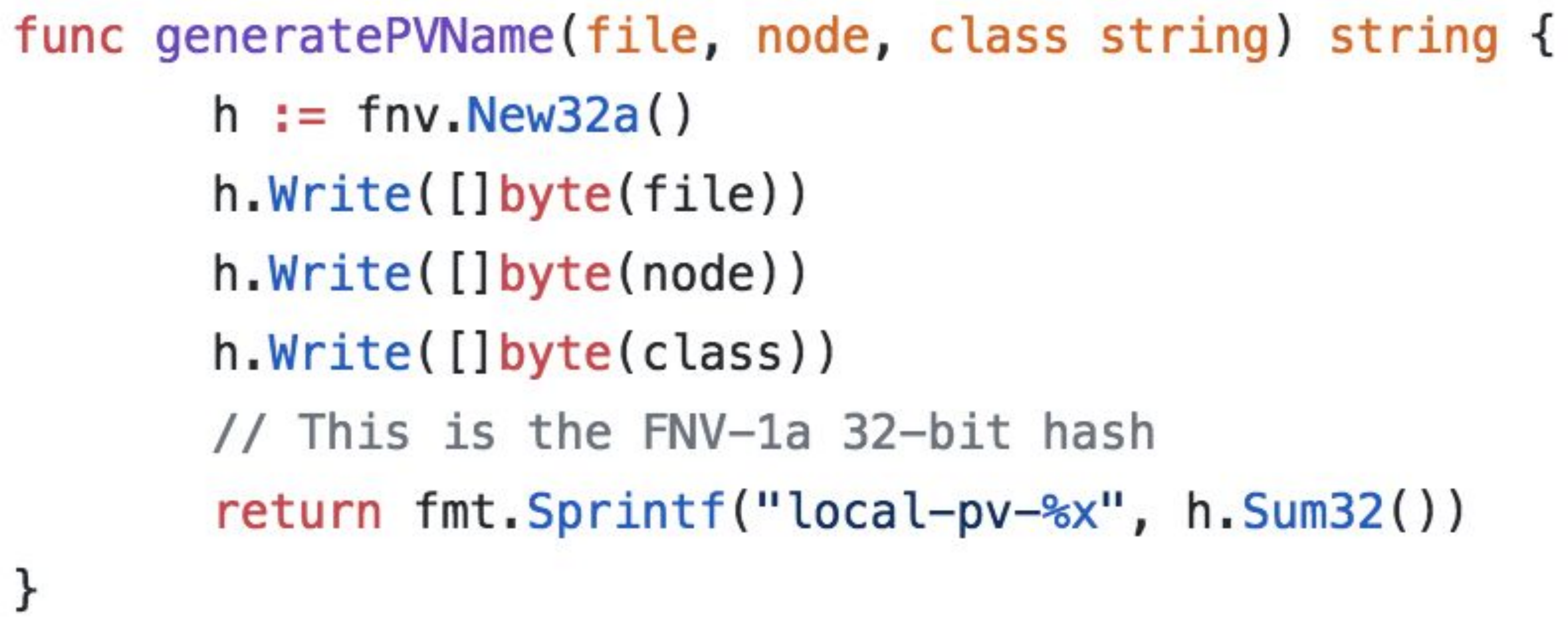
- Local: Cloud provider disk errors
- Local: New node with same name
- EBS scheduler



Laurent Bernaille @lbernail · Sep 26

Some days you know things are going to be weird^Winteresting:
cat /proc/28019/wchan
__refrigerator

localvolumeprovision: discovery.go



```
func generatePVName(file, node, class string) string {
	h := fnv.New32a()
	h.Write([]byte(file))
	h.Write([]byte(node))
	h.Write([]byte(class))
	// This is the FNV-1a 32-bit hash
	return fmt.Sprintf("local-pv-%x", h.Sum32())
}
```

Stateful sets

Scheduling tricks

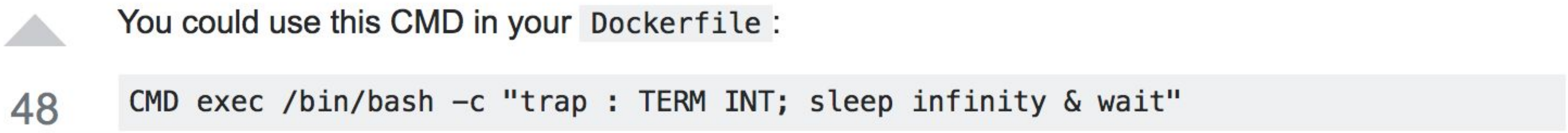
```
kubectl get sts myapp
NAME                               DESIRED   CURRENT   AGE
myapp                              5         4         5d

kubectl get pods -lapp=myapp
NAME                       READY     STATUS              RESTARTS   AGE
myapp-0                    1/1       Running             0          5d
myapp-1                    1/1       Running             10         6m
myapp-2                    1/1       CrashloopBackoff    10         6m
[?]
myapp-4                    1/1       Running             0          5d
```

Cargo culting

How can I keep container running on Kubernetes?

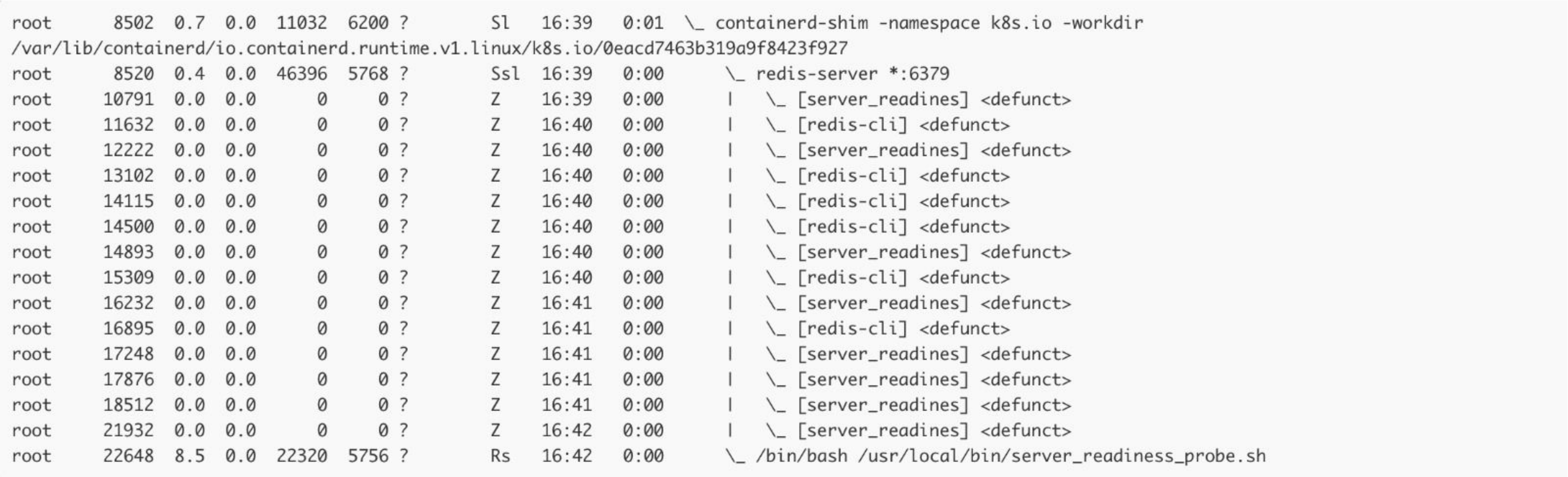
https://stackoverflow.com/questions/31870222/**how-can-i-keep-container-running-on-kubernetes**



You could use this CMD in your `Dockerfile`:

```
CMD exec /bin/bash -c "trap : TERM INT; sleep infinity & wait"
```

Zombies



```
root      8502  0.7  0.0  11032  6200 ?        Sl   16:39   0:01  \_ containerd-shim -namespace k8s.io -workdir
/var/lib/containerd/io.containerd.runtime.v1.linux/k8s.io/0eacd7463b319a9f8423f927
root      8520  0.4  0.0  46396  5768 ?        Ssl  16:39   0:00      \_ redis-server *:6379
root     10791  0.0  0.0      0     0 ?        Z    16:39   0:00      |   \_ [server_readines] <defunct>
root     11632  0.0  0.0      0     0 ?        Z    16:40   0:00      |   \_ [redis-cli] <defunct>
root     12222  0.0  0.0      0     0 ?        Z    16:40   0:00      |   \_ [server_readines] <defunct>
root     13102  0.0  0.0      0     0 ?        Z    16:40   0:00      |   \_ [redis-cli] <defunct>
root     14115  0.0  0.0      0     0 ?        Z    16:40   0:00      |   \_ [redis-cli] <defunct>
root     14500  0.0  0.0      0     0 ?        Z    16:40   0:00      |   \_ [redis-cli] <defunct>
root     14893  0.0  0.0      0     0 ?        Z    16:40   0:00      |   \_ [server_readines] <defunct>
root     15309  0.0  0.0      0     0 ?        Z    16:40   0:00      |   \_ [redis-cli] <defunct>
root     16232  0.0  0.0      0     0 ?        Z    16:41   0:00      |   \_ [server_readines] <defunct>
root     16895  0.0  0.0      0     0 ?        Z    16:41   0:00      |   \_ [redis-cli] <defunct>
root     17248  0.0  0.0      0     0 ?        Z    16:41   0:00      |   \_ [server_readines] <defunct>
root     17876  0.0  0.0      0     0 ?        Z    16:41   0:00      |   \_ [server_readines] <defunct>
root     18512  0.0  0.0      0     0 ?        Z    16:41   0:00      |   \_ [server_readines] <defunct>
root     21932  0.0  0.0      0     0 ?        Z    16:42   0:00      |   \_ [server_readines] <defunct>
root     22648  8.5  0.0  22320  5756 ?        Rs   16:42   0:00      \_ /bin/bash /usr/local/bin/server_readiness_probe.sh
```

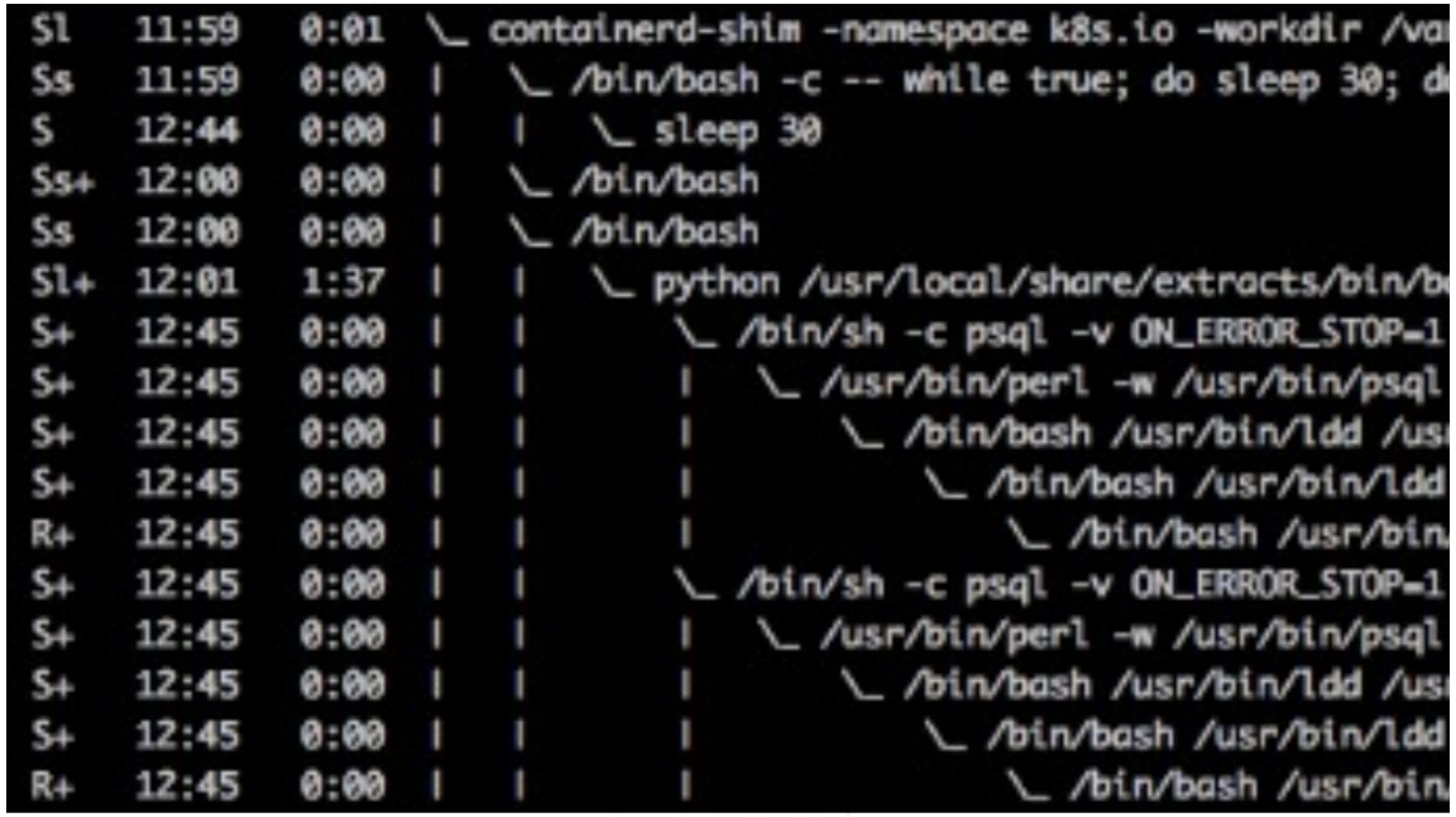
```
ps auxf | grep -c defunct
16018
```

```
readinessProbe:
  exec:
  command: [server_readiness_probe.sh]
  timeoutSeconds: 1
```

Takeway

- Careful with exec-based probes
- Use tini as pid 1 (or shared pid namespace)

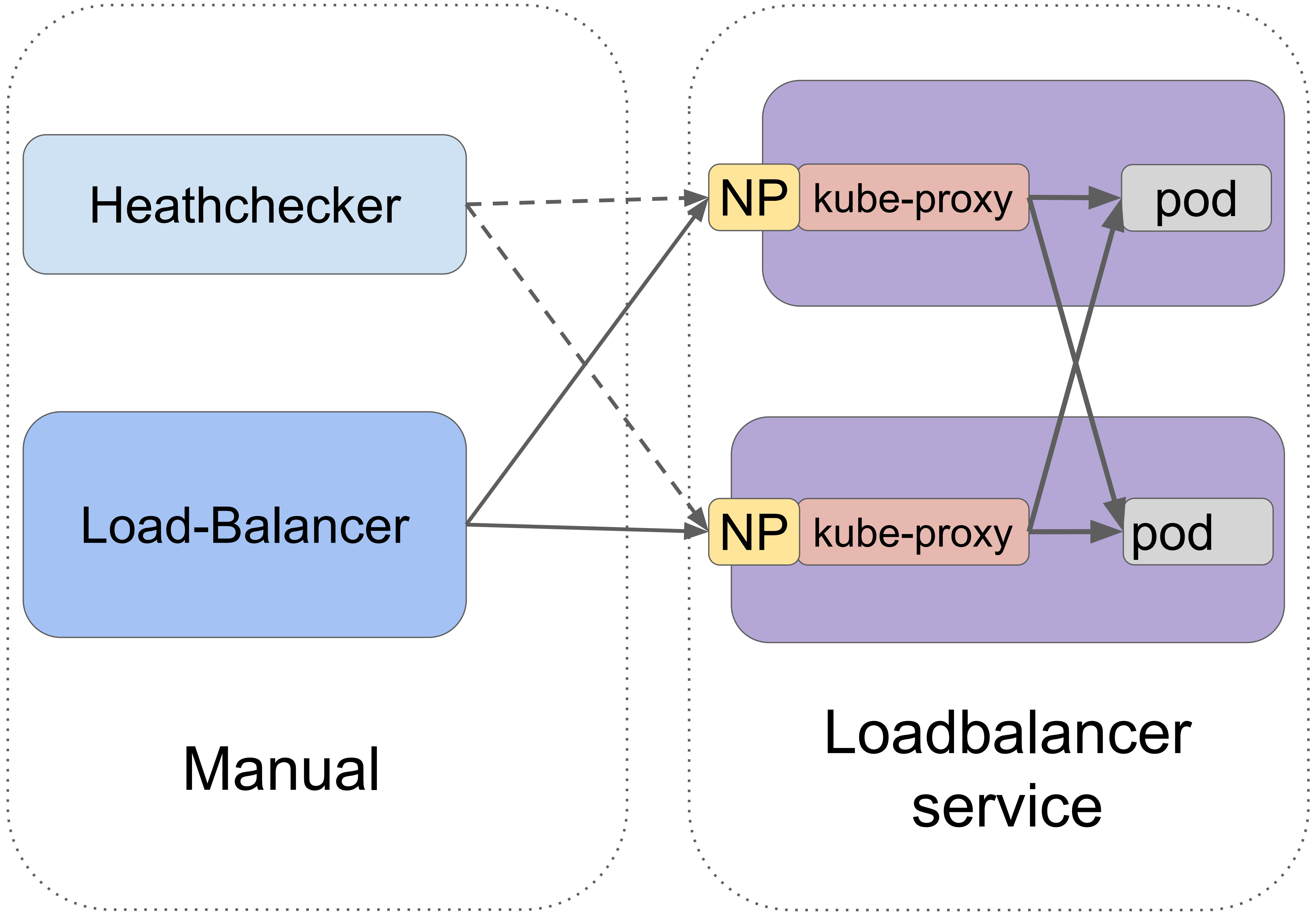
Containers, not VMs



```
Sl    11:59   0:01  \_ containerd-shim -namespace k8s.io -workdir /va
Ss    11:59   0:00  |   \_ /bin/bash -c -- while true; do sleep 30; d
S     12:44   0:00  |   |   \_ sleep 30
Ss+   12:00   0:00  |   \_ /bin/bash
Ss    12:00   0:00  |   \_ /bin/bash
Sl+   12:01   1:37  |   |   \_ python /usr/local/share/extracts/bin/b
S+    12:45   0:00  |   |       \_ /bin/sh -c psql -v ON_ERROR_STOP=1
S+    12:45   0:00  |   |       |   \_ /usr/bin/perl -w /usr/bin/psql
S+    12:45   0:00  |   |       |       \_ /bin/bash /usr/bin/ldd /us
S+    12:45   0:00  |   |       |           \_ /bin/bash /usr/bin/ldd
R+    12:45   0:00  |   |       |               \_ /bin/bash /usr/bin
S+    12:45   0:00  |   |       \_ /bin/sh -c psql -v ON_ERROR_STOP=1
S+    12:45   0:00  |   |       |   \_ /usr/bin/perl -w /usr/bin/psql
S+    12:45   0:00  |   |       |       \_ /bin/bash /usr/bin/ldd /us
S+    12:45   0:00  |   |       |           \_ /bin/bash /usr/bin/ldd
R+    12:45   0:00  |   |       |               \_ /bin/bash /usr/bin
```

Complex process trees and many open files is very hard on the runtime

Native resources / external config



Synchronisation

- NodePort
- External IP

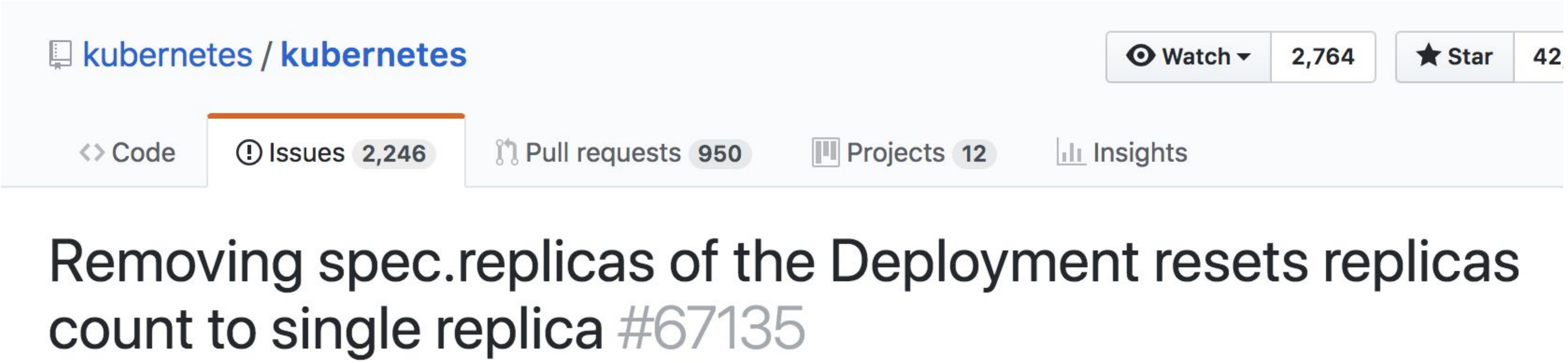
Issues

- Keeping track: Load-balancer External IP re-assigned to node
- Port conflict on Internal Load-Balancer (port 443, **broke apiservers**…)

Rolling updates

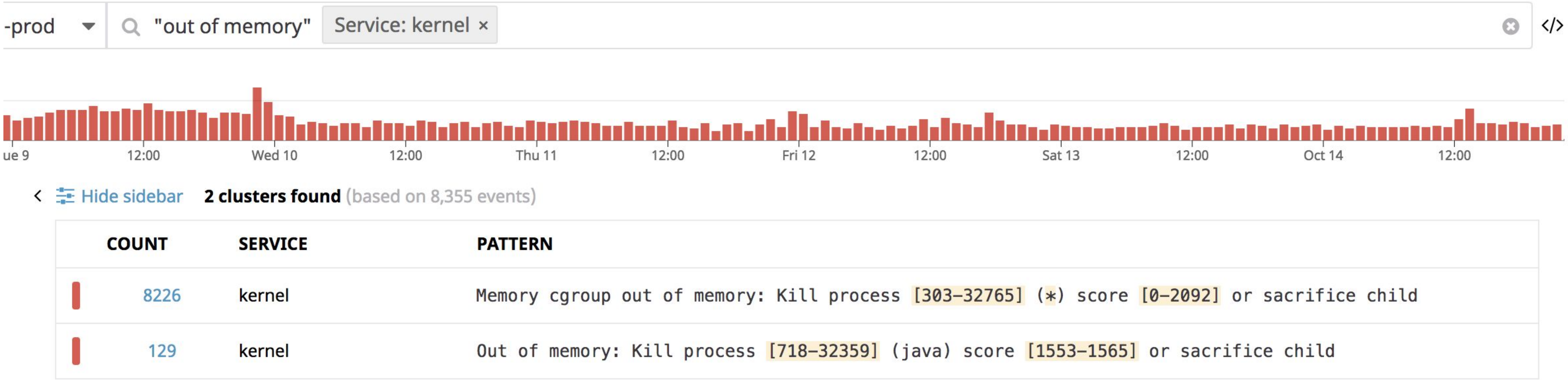
spec.replicas and hpa

- Replicas override hpa settings
- Removing replicas is not enough
- Recommended solution: edit the “last-applied” deployment first



Removing spec.replicas of the Deployment resets replicas count to single replica #67135

OOM Killer



2 clusters found (based on 8,355 events)

	COUNT	SERVICE	PATTERN
🟥	8226	kernel	`Memory cgroup out of memory: Kill process [303-32765] (*) score [0-2092] or sacrifice child`
🟥	129	kernel	`Out of memory: Kill process [718-32359] (java) score [1553-1565] or sacrifice child`

Limits too low will trigger "cgroup oom"

Requests too low (*or 0*) will trigger "system oom"

Not a surprise, but do spend some effort on sizing

InitContainers

Requests/Limits

- Pod Resources = MAX(MAX(initContainers), sum(containers))
- LimitRanger also applies to InitContainers

Inconsistent behavior on container restarts

- *Usually* not restarted
- *Except* when Kubernetes no longer knows their exit status

Manual changes

Untracked kubectl changes

```
kubectl apply / edit
```

Partial chart apply

```
kubectl apply deploy
  => checksum/config_templates: 840ae1e0b4b2f7b5033edd34fd9eb88b55dc914adca5c
```

^ Hash for the updated config map, but not deployed

Chart deletion

```
kubectl delete -f <dir>
```

Contained namespace

Future Plans

Control plane isolation

- “Meta” cluster for control planes

Better load balancing

- Avoid imbalanced API servers
- Avoid imbalanced etcd

Container-Optimized images

- In-place upgrade for data stores

Custom controllers

- We already have a few but will build new ones

Conclusion

The Kubernetes Control plane is very complex

- ➢ Crazy number of options/tunables
- ➢ Low-level components are great but some bugs remain
- ➢ The ecosystem is very young
- ➢ Reading the code is not optional (and fixing part of it)

Account for culture change and training

- ➢ Kubernetes resources don't look that complicated
- ➢ Many, many edge-cases and pitfalls

Instrument the audit logs

- ➢ Very high value to debug performance issues
- ➢ Also helps understand history of interactions with a specific resource
- ➢ Expensive (very verbose: we are at 1000+ logs/s) but game changing for us

Thank you

Also, We're hiring

DATADOG