Operating Within Normal Parameters: Monitoring Kubernetes

Elana Hashman
Two Sigma Investments, LP

SREcon 2019 Americas – Brooklyn, NY

@ehashdn :: #SREcon

Disclaimer

This document is being distributed for informational and educational purposes only and is not an offer to sell or the solicitation of an offer to buy any securities or other instruments. The information contained herein is not intended to provide, and should not be relied upon for, investment advice. The views expressed herein are not necessarily the views of Two Sigma Investments, LP or any of its affiliates (collectively, "Two Sigma"). Such views reflect the assumptions of the author(s) of the document and are subject to change without notice. The document may employ data derived from third-party sources. No representation is made by Two Sigma as to the accuracy of such information and the use of such information in no way implies an endorsement of the source of such information or its validity.

The copyrights and/or trademarks in some of the images, logos or other material used herein may be owned by entities other than Two Sigma. If so, such copyrights and/or trademarks are most likely owned by the entity that created the material and are used purely for identification and comment as fair use under international copyright and/or trademark laws. Use of such image, copyright or trademark does not imply any association with such organization (or endorsement of such organization) by Two Sigma, nor vice versa.

Outline

- A brief introduction to “observability”
- Service Level Objectives (SLOs), a measure of “normal”
- Collecting Kubernetes metrics: what’s available?
- **How-to:** A minimal FOSS monitoring stack for Kubernetes
- Debug some common problems using our metrics!

What is “observability?”

A fancy name to make monitoring more marketable?

Why are we even here?

To operate systems that
make users happy.

When **something goes wrong**

O*bservability* lets you answer
what, where, how, and ***why***

How do you agree on
something gone wrong?

Service Level Objectives, perhaps

Defining Service Level Objectives

- **Service Level Objectives (SLOs)** are a formal specification of what your team considers **normal** for a service
- SLOs cover areas (**availability**, **latency**, **capacity**, etc.) and specific targets for quality of service
- Areas and targets differ depending on circumstances
  - e.g. development vs. production

Defining Service Level Objectives

- Who are your users and how do they interact with your cluster?
  - Do you have an intermediary platform?
  - What are their performance expectations?
- What capacity and load are you expecting?
  - How many nodes per cluster and what size?
  - How many users? What is their average workload size?

Defining Service Level Objectives

- SLOs communicate your service expectations with users
- Some Kubernetes-specific examples:
  - **Availability:** Control plane has 99% monthly uptime
  - **Latency:** Valid Pods should start within 5s for p99
  - **Capacity:** Cluster accommodates 50 running Pods per user

Defining Service Level Objectives

- ▪ SLOs are flexible and context-dependent

✔ SLOs set customer expectations through a commitment to quality of service

✘ SLOs are not a measure of your team's ability to deliver 9's

Defining Service Level Objectives

- Can't commit to quality of service targets if you have no idea what your quality of service is
- Sample workloads provide data for performance tuning and iteration on SLOs
- Must include a monitoring stack *in every cluster* at launch
  - But how??

Case study: instrumenting Kubernetes

Collecting Kubernetes metrics

- What sources of metrics are available?
- How can metrics be analyzed, aggregated, and visualized?

What sources of metrics are available?

Timeseries	Value
`up{job="kube-apiserver",instance="api-1"}`	1

What sources of metrics are available?

Out-of-the-box metrics

- Most Kubernetes components export Prometheus metrics
  - etcd (`/metrics`)
  - API servers (`/metrics`)
  - Kubelets (`/api/v1/nodes/<node>/proxy/metrics`)
  - cadvisor (`/api/v1/nodes/<node>/proxy/metrics/cadvisor`)
  - Service endpoints (`/metrics` via cluster service discovery)

What sources of metrics are available?

Official Kubernetes metric exporters

- kubernetes/**kube-state-metrics** (stable)
  - Prometheus adapter for cluster state
- kubernetes-incubator/**metrics-server** (alpha)
  - Aggregates metrics from kubelets (**not** Prometheus format)
  - Provides programmatic access for autoscalers, `kubectl top`, etc.
- kubernetes-retired/**heapster** (deprecated)
  - Similar to metrics-server, used InfluxDB backend storage

What sources of metrics are available?

Even more metrics from Prometheus exporters!

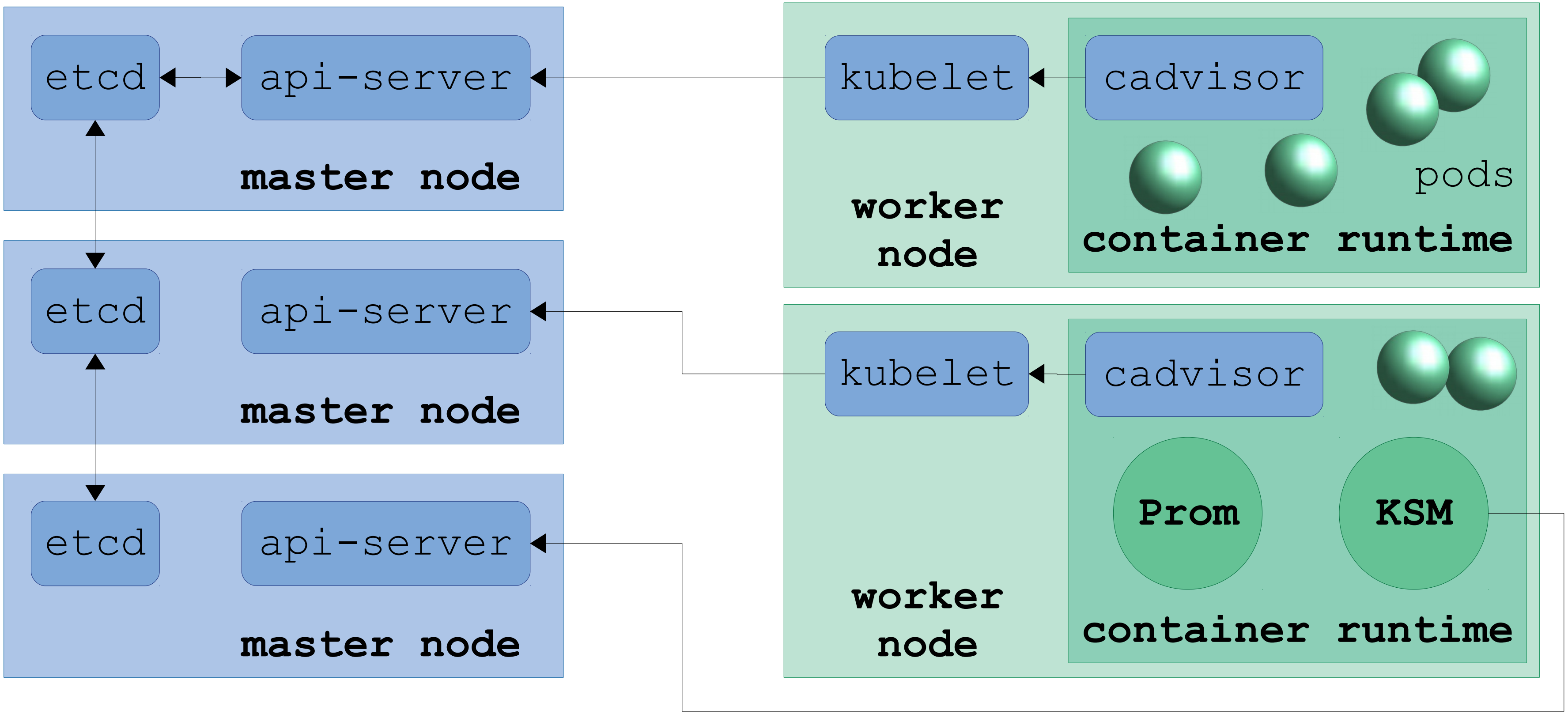
- prometheus/**node_exporter**
  - System metrics for your Kubernetes Nodes
- prometheus/**blackbox_exporter**
  - Probes arbitrary endpoints via HTTP, HTTPS, DNS, TCP, or ICMP
- Write your own
- Many other open source options

What types of metrics are available?

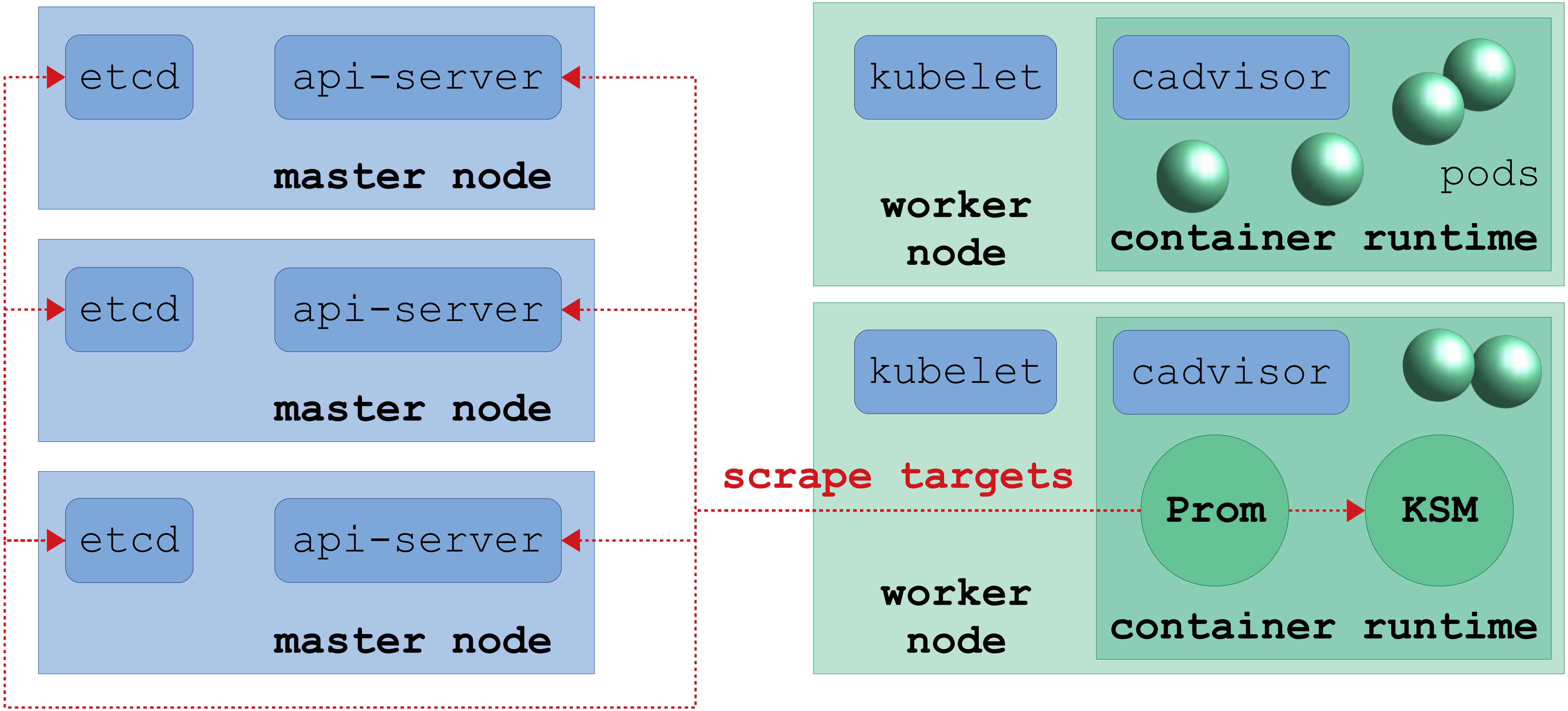
- Container CPU, memory, network utilization: **cadvisor**
- General Pod info: **kube-state-metrics**
- Node performance info: **node_exporter**
- General cluster info: **many sources**
- Control plane info: **etcd, API servers**
  - Sample metric queries → *see talk resources*

How-to: Let's deploy this!

A minimal monitoring stack for Kubernetes

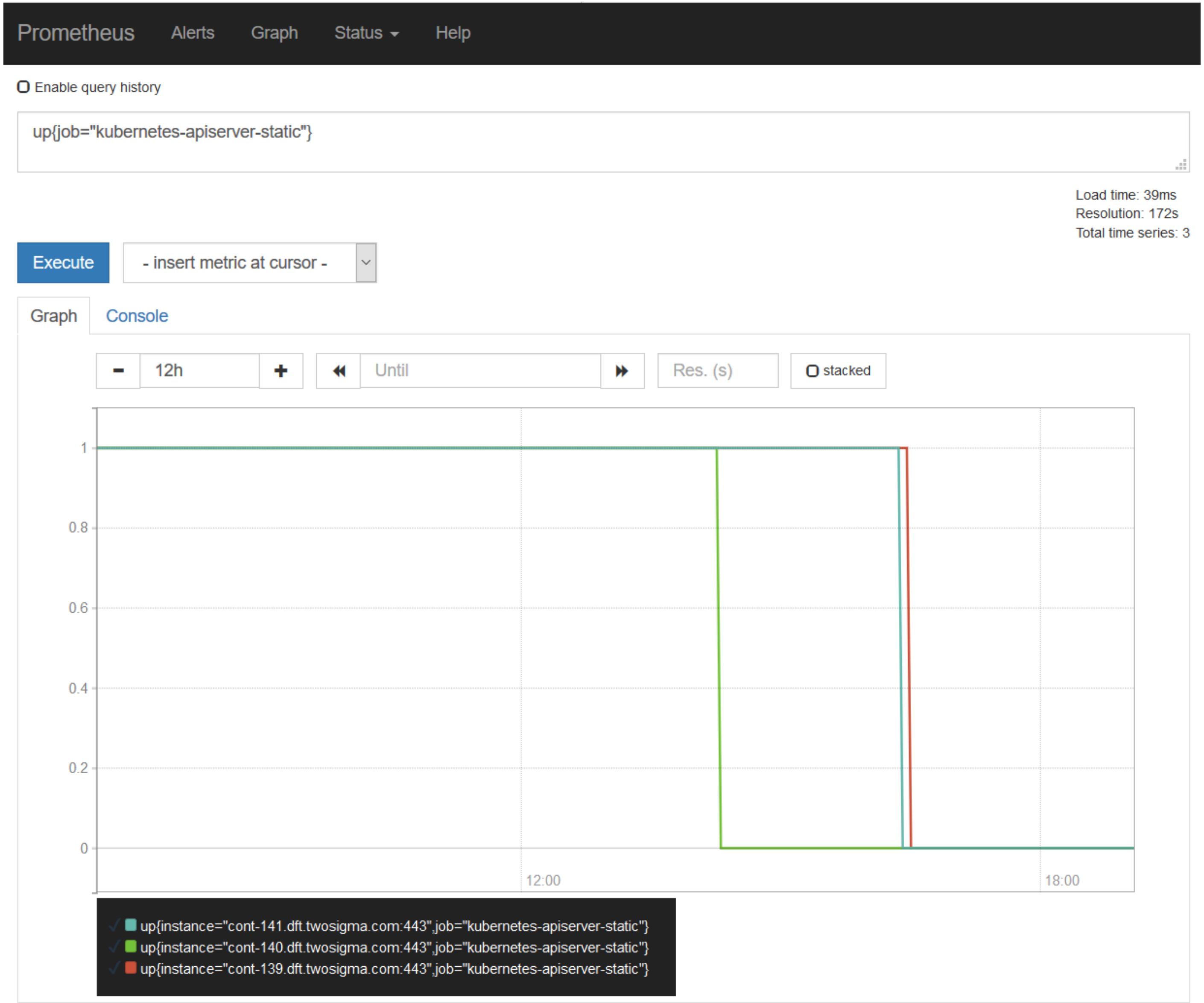


A minimal monitoring stack for Kubernetes



Run your monitoring stack on Kubernetes!

- Credentials for scraping are way easier to manage
  - Grant a ServiceAccount granular permissions!
  - ServiceAccount tokens get automatically rotated!
- Kubernetes abstractions and architecture are powerful
  - Built-in service discovery for scraping!
  - Kubernetes Deployments keep your Pods alive!
  - Data plane is resilient to control plane failures!



```
up{job="kubernetes-apiserver-static"}
```

Load time: 39ms
Resolution: 172s
Total time series: 3

✓ 🟦 up{instance="cont-141.dft.twosigma.com:443",job="kubernetes-apiserver-static"}
✓ 🟩 up{instance="cont-140.dft.twosigma.com:443",job="kubernetes-apiserver-static"}
✓ 🟥 up{instance="cont-139.dft.twosigma.com:443",job="kubernetes-apiserver-static"}

Let’s not worry about high availability!

- High availability is not as simple as “run two replicas”
  - Two Prometheus replicas doubles (high) scrape load
  - Prometheus replicas are stateful, with subtly different state
- kube-state-metrics is stateless, so why not?
  - Prometheus counters monotonically increase but differ between replicas
  - You could scrape all of them simultaneously and deduplicate client-side?
- >:(

It’s okay for Prometheus to not be a panacea

- Set up backup monitoring jobs
  - Run them off-cluster
  - Kubernetes’ scheduling gives us 99% uptime for ~free
  - Alert when Prometheus or KSM has extended downtime
- This architecture avoids data integrity issues and deployment complexity, for way less work

Metric analysis, aggregation, visualization

- **Prometheus** query language (PromQL) powers metric analysis and aggregation; Prometheus UI for visualizations
- **Grafana** accepts Prometheus data sources for dashboards
- Can perform arbitrary processing on metrics in **JSON format**
  - *Prometheus format JSON:* use Prometheus query API
  - *Metrics API format JSON or gRPC:* use Metrics Server API

How can we use this data for debugging?

Service Degradation: Node is down

- **Obvious:** Prometheus scrape job is down

```
up{job="kube-nodes"} != 1
```

- **Less obvious:** Grey failure indicated by unusually slow scrape time

```
scrape_duration_seconds{job="kube-nodes"} > 2
```

Service Degradation: Customer can’t launch Pods

- **Obvious:** Customer has hit their quota limit

```
sum(kube_resourcequota{namespace="foo",resource="cpu",type="used"})
 /  kube_resourcequota{namespace="foo",resource="cpu",type="hard"}
 > 0.95
```

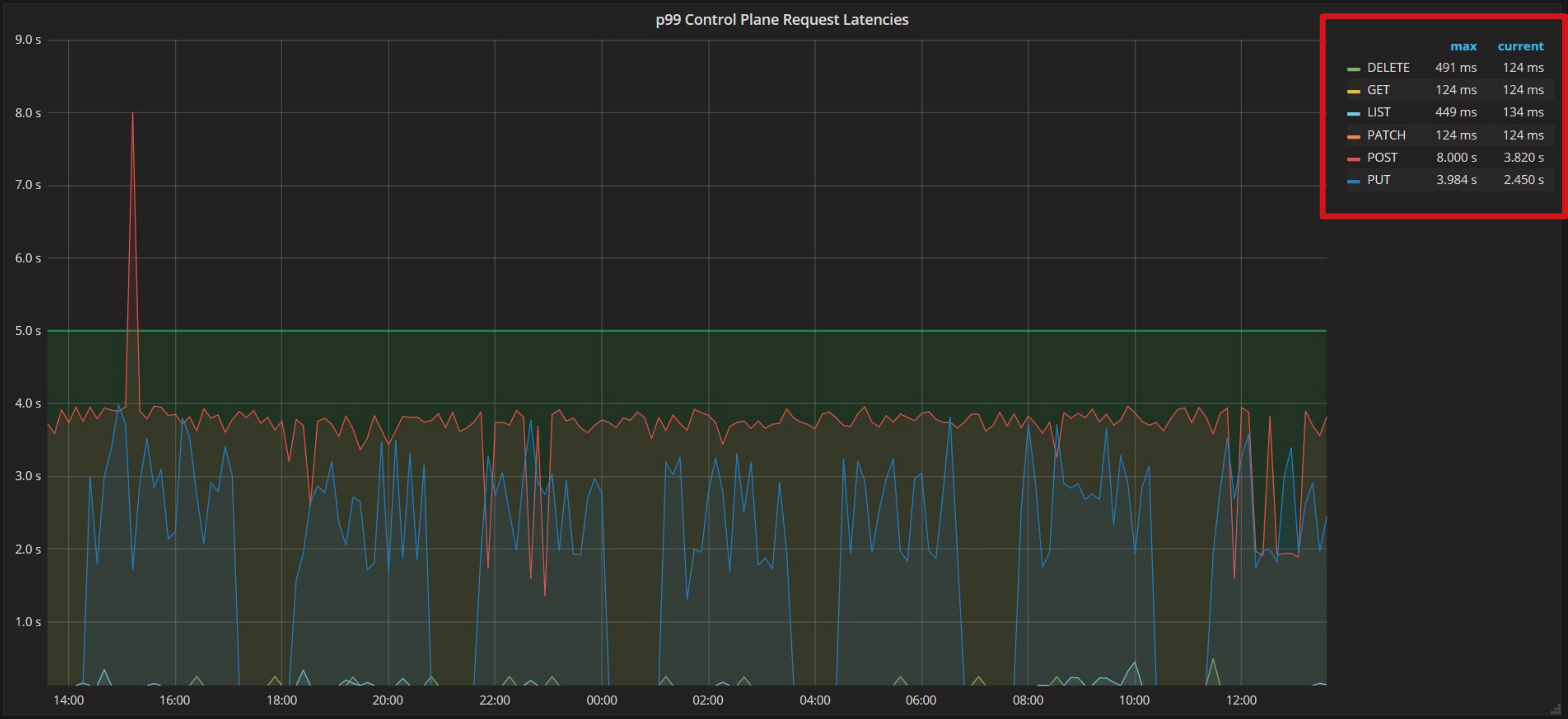
- **Less obvious:** Customer has overprovisioned their workloads

```
sum(container_cpu_usage_seconds_total:rate1m{namespace="foo"})
 /  kube_resourcequota{namespace="foo",resource="cpu",type="hard"}
 < 0.35
```

Service Degradation: API Server is slow

- **Obvious:** API server calls are slow

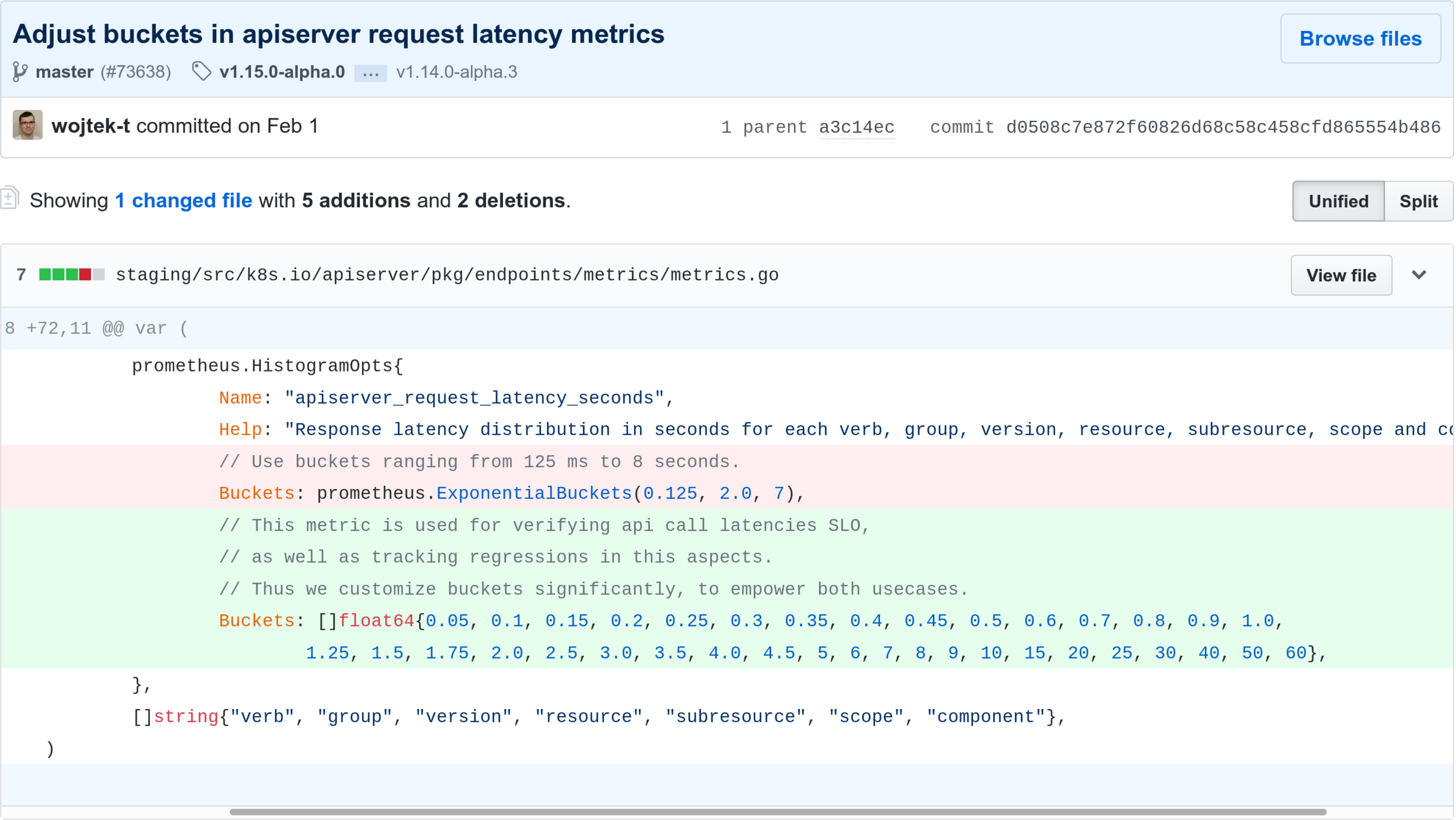
```
histogram_quantile(
  0.99,
  sum(rate(apiserver_request_latencies_bucket[1m]))
    by (le, verb)
)
```



	max	current
🟩 DELETE	491 ms	124 ms
🟨 GET	124 ms	124 ms
🟦 LIST	449 ms	134 ms
🟧 PATCH	124 ms	124 ms
🟥 POST	8.000 s	3.820 s
🟦 PUT	3.984 s	2.450 s

Service Degradation: API Server is slow

- **Less obvious:** API server metrics cap out between 125ms–8s because of default bucketing???



Adjust buckets in apiserver request latency metrics

master (#73638) v1.15.0-alpha.0 ... v1.14.0-alpha.3

wojtek-t committed on Feb 1

1 parent a3c14ec commit d0508c7e872f60826d68c58c458cfd865554b486

Showing **1 changed file** with **5 additions** and **2 deletions**.

7 🟩🟩🟩🟥⬜ staging/src/k8s.io/apiserver/pkg/endpoints/metrics/metrics.go

```
8 +72,11 @@ var (
		prometheus.HistogramOpts{
			Name: "apiserver_request_latency_seconds",
			Help: "Response latency distribution in seconds for each verb, group, version, resource, subresource, scope and co
			// Use buckets ranging from 125 ms to 8 seconds.
			Buckets: prometheus.ExponentialBuckets(0.125, 2.0, 7),
			// This metric is used for verifying api call latencies SLO,
			// as well as tracking regressions in this aspects.
			// Thus we customize buckets significantly, to empower both usecases.
			Buckets: []float64{0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0,
				1.25, 1.5, 1.75, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 40, 50, 60},
		},
		[]string{"verb", "group", "version", "resource", "subresource", "scope", "component"},
	)
```

Recap

- We learned how to select Service Level Objectives
- We explored FOSS monitoring solutions for Kubernetes
- We built a minimal monitoring stack
- We used it to debug some production issues
- **Try it for yourself:** check out the sample code on GitHub

Questions?

Thanks to:

Two Sigma Investments, LP
Liz Fong-Jones, Frederic Branczyk

Talk resources: https://hashman.ca/srecon-2019