Treating Dashboards Like Code

Scott Kidder, Staff Software Engineer @ Mux

Grafanacon, February 26, 2019

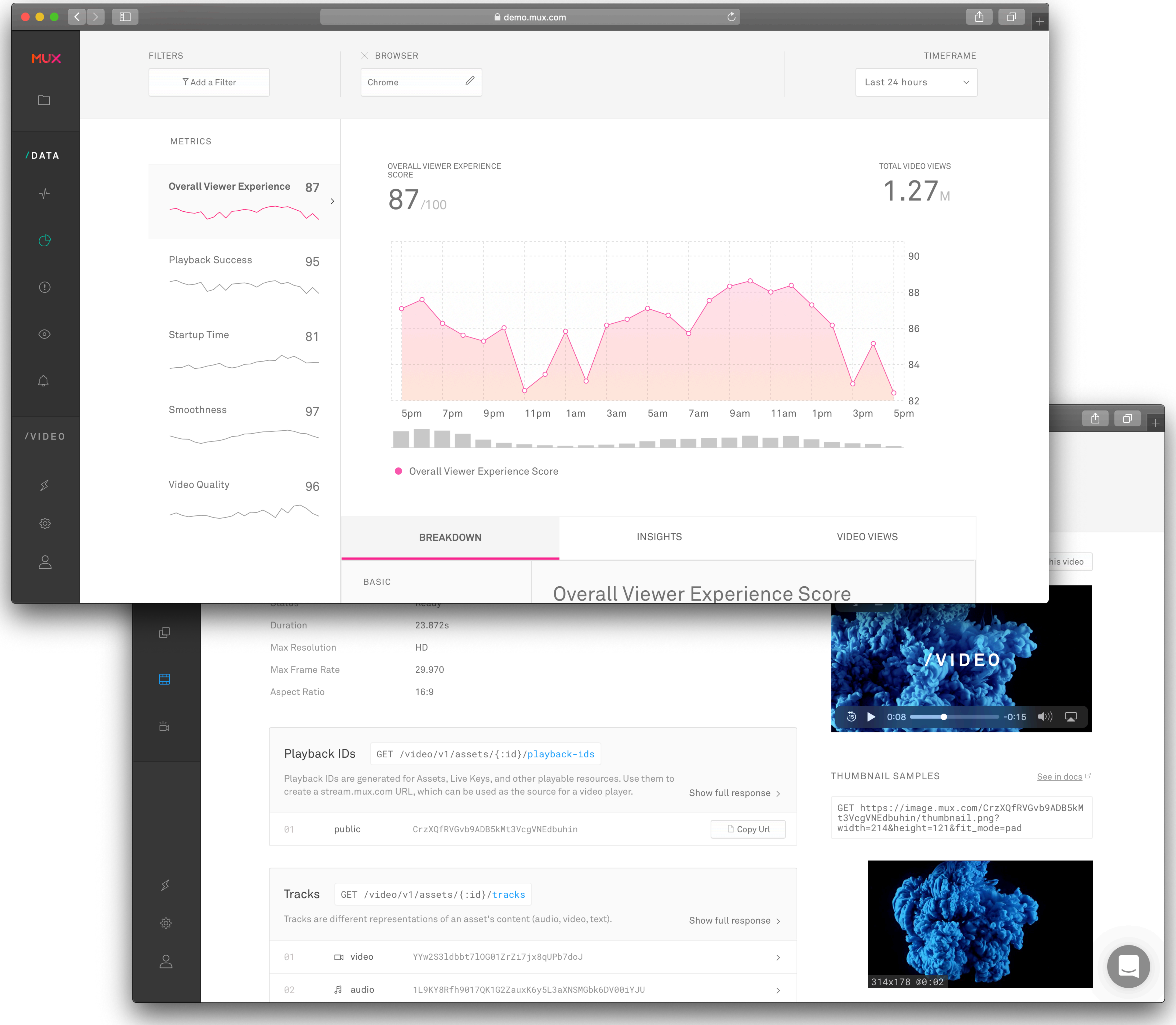
Agenda

- Background on Mux
- Monitoring for Mux Data
- Greenfield Monitoring Opportunities with Mux Video
- Goals for monitoring
- Questions

Background on Mux

What is Mux?

- Mux Data: Analytics for Video (2016)
- Mux Video: API for Video (2018)
- Mux Video makes it easy to publish video with a REST API call
- Optimal video encoding settings chosen automatically
- Deployments in AWS and Google Cloud



demo.mux.com

FILTERS: Add a Filter — BROWSER: Chrome — TIMEFRAME: Last 24 hours

METRICS

Overall Viewer Experience	87
Playback Success	95
Startup Time	81
Smoothness	97
Video Quality	96

OVERALL VIEWER EXPERIENCE SCORE 87 /100

TOTAL VIDEO VIEWS 1.27 M

Overall Viewer Experience Score

Duration	23.872s
Max Resolution	HD
Max Frame Rate	29.970
Aspect Ratio	16:9

Playback IDs `GET /video/v1/assets/{:id}/playback-ids`

Playback IDs are generated for Assets, Live Keys, and other playable resources. Use them to create a stream.mux.com URL, which can be used as the source for a video player.

Show full response ›

01	public	CrzXQfRVGvb9ADB5kMt3VcgVNEdbuhin	Copy Url

Tracks `GET /video/v1/assets/{:id}/tracks`

Tracks are different representations of an asset's content (audio, video, text).

Show full response ›

01	video	YYw2S3ldbbt7lOG01ZrZi7jx8qUPb7doJ
02	audio	1L9KY8Rfh9017QK1G2ZauxK6y5L3aXNSMGbk6DV00iYJU

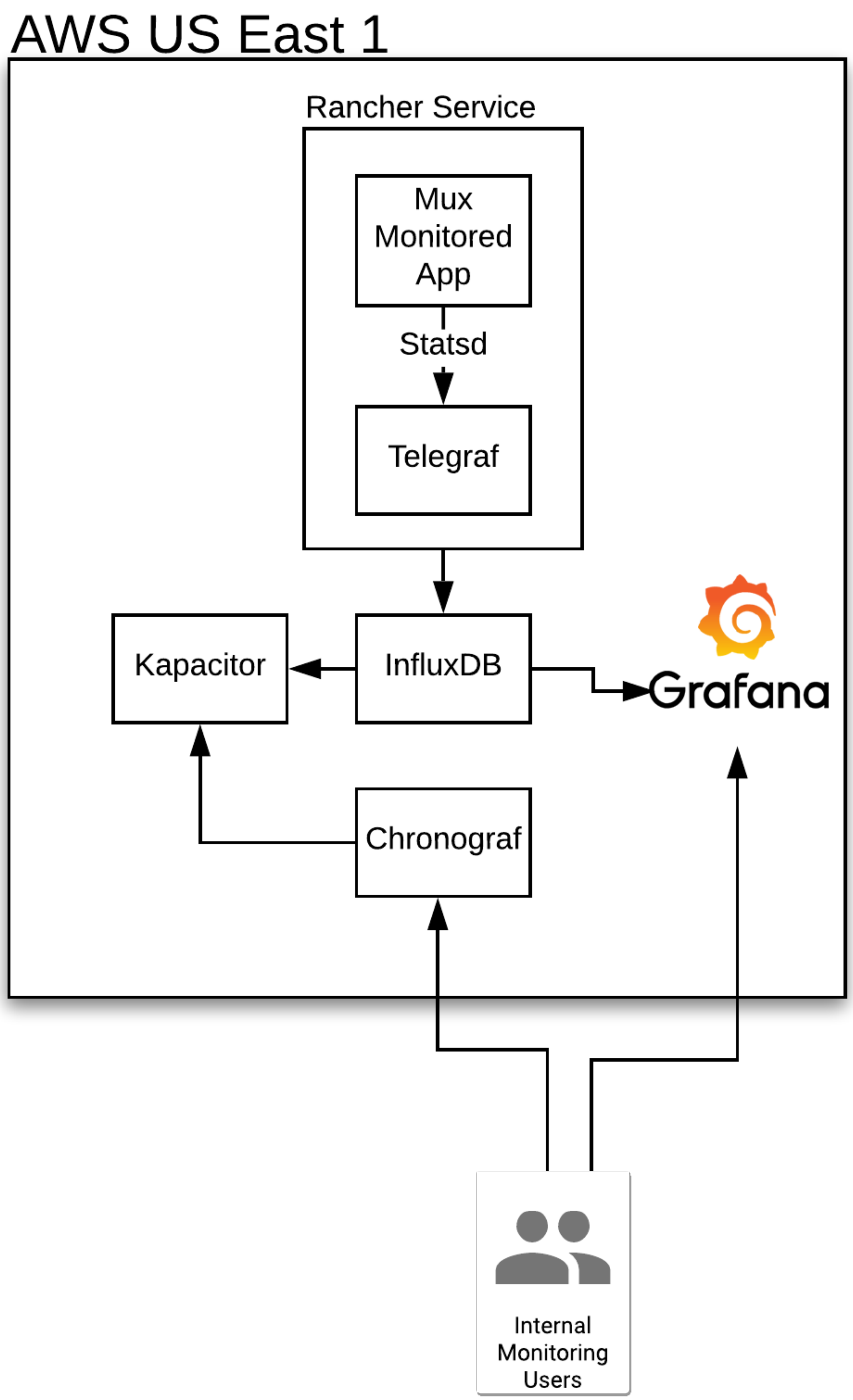
THUMBNAIL SAMPLES — See in docs

```
GET https://image.mux.com/CrzXQfRVGvb9ADB5kM
t3VcgVNEdbuhin/thumbnail.png?
width=214&height=121&fit_mode=pad
```

Monitoring for Mux Data

For a Moment, Let’s Return to a Simpler Time

- Mux has used Grafana since inception (early 2016)
- Single deployment of Rancher container orchestration system in AWS
- Supported Mux Data, our only product at the time
- Single Grafana instance for all dashboards
- Single InfluxDB instance for application metrics



AWS US East 1

Rancher Service

Mux Monitored App

Statsd

Telegraf

Kapacitor

InfluxDB

Grafana

Chronograf

Internal Monitoring Users

But in many ways, things were more difficult…

Problems began to surface

- Management of alerting rules was performed in Chronograf
- Ran a second visualization tool just to administer alerting rules
- No versioned history of alerting rules
- Rules were often disabled during a deploy or maintenance, and then people would forget to re-enable them, leading to undetected incidents
- Unclear why alerts were disabled, and whether it's safe to delete

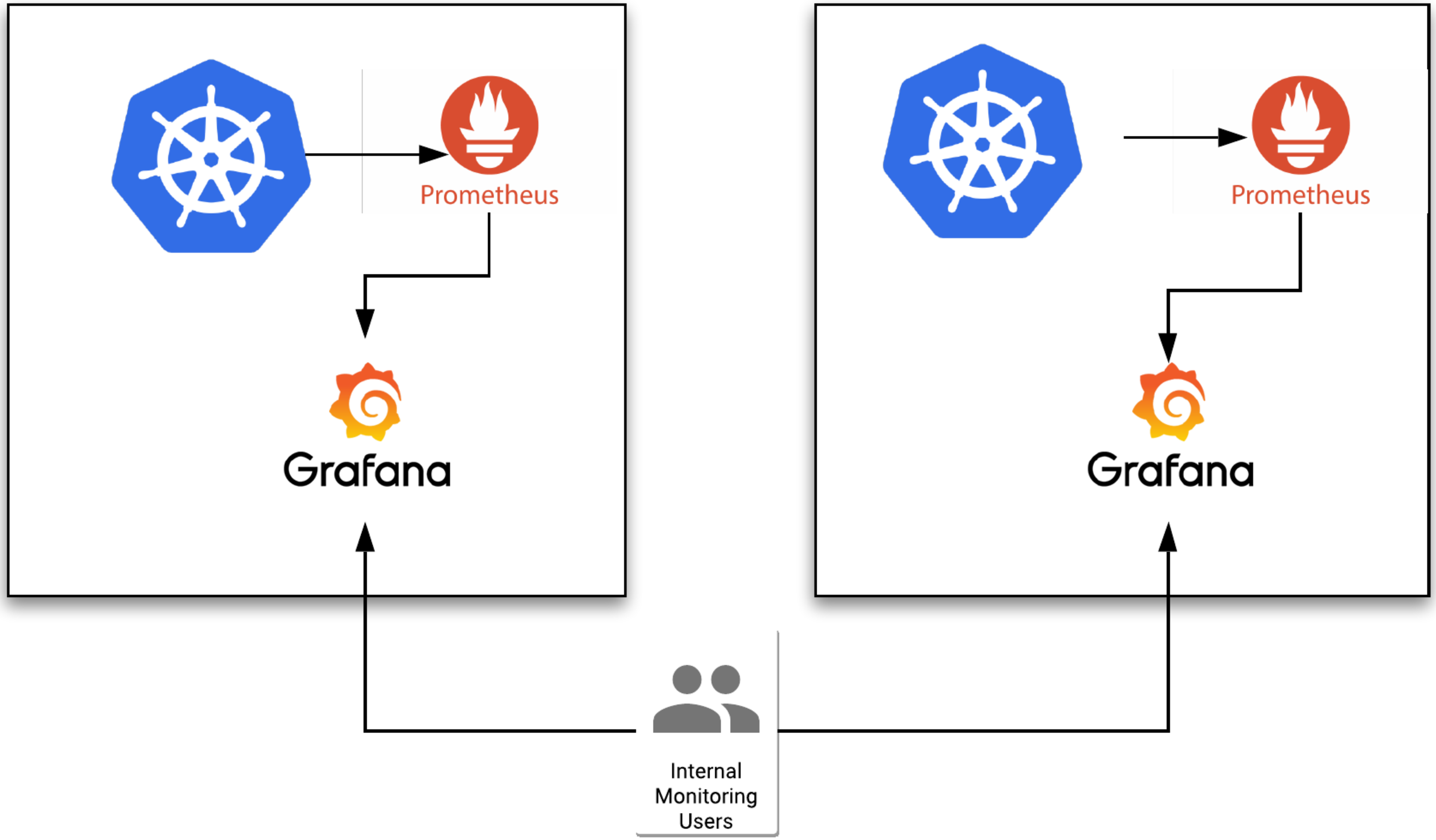
Greenfield Monitoring Opportunities with Mux Video

Mux Video Development

- Late 2017 we began developing Mux Video
- We had already run some proof-of-concept Kubernetes cluster with Mux Data
- Decided to run all services in Kubernetes and monitor with Prometheus and Grafana

GCE US East 1

GCE US East 4



Prometheus

Grafana

Prometheus

Grafana

Internal
Monitoring
Users

Goals for Monitoring

Goals

1. Easily configure which services are scraped by Prometheus
2. Run policy checks on alert rules with each build
3. Store the dashboards and alert rules alongside code
4. Automatically deploy dashboards and alert rules to Kubernetes clusters each time we ship code

Goal #1

Easily configure which services are scraped by Prometheus

Prometheus Monitoring in Kubernetes

- Using the Prometheus Operator to configure Prometheus and Alertmanager

- https://github.com/coreos/prometheus-operator

- Uses Kubernetes label metadata to target which services to scrape and on which port

Prometheus: Kubernetes Service Monitor

```
apiVersion:
monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: core-servers
  namespace: monitoring
  labels:
    k8s-app: core-servers
spec:
  jobLabel: core-servers
  namespaceSelector:
    any: true
  selector:
    matchLabels:
      monitoring: core
  endpoints:
  - port: metrics
    interval: 10s
    honorLabels: true
```

1) Examine services in all Kubernetes namespaces

2) Match on services with a “monitoring: core” label

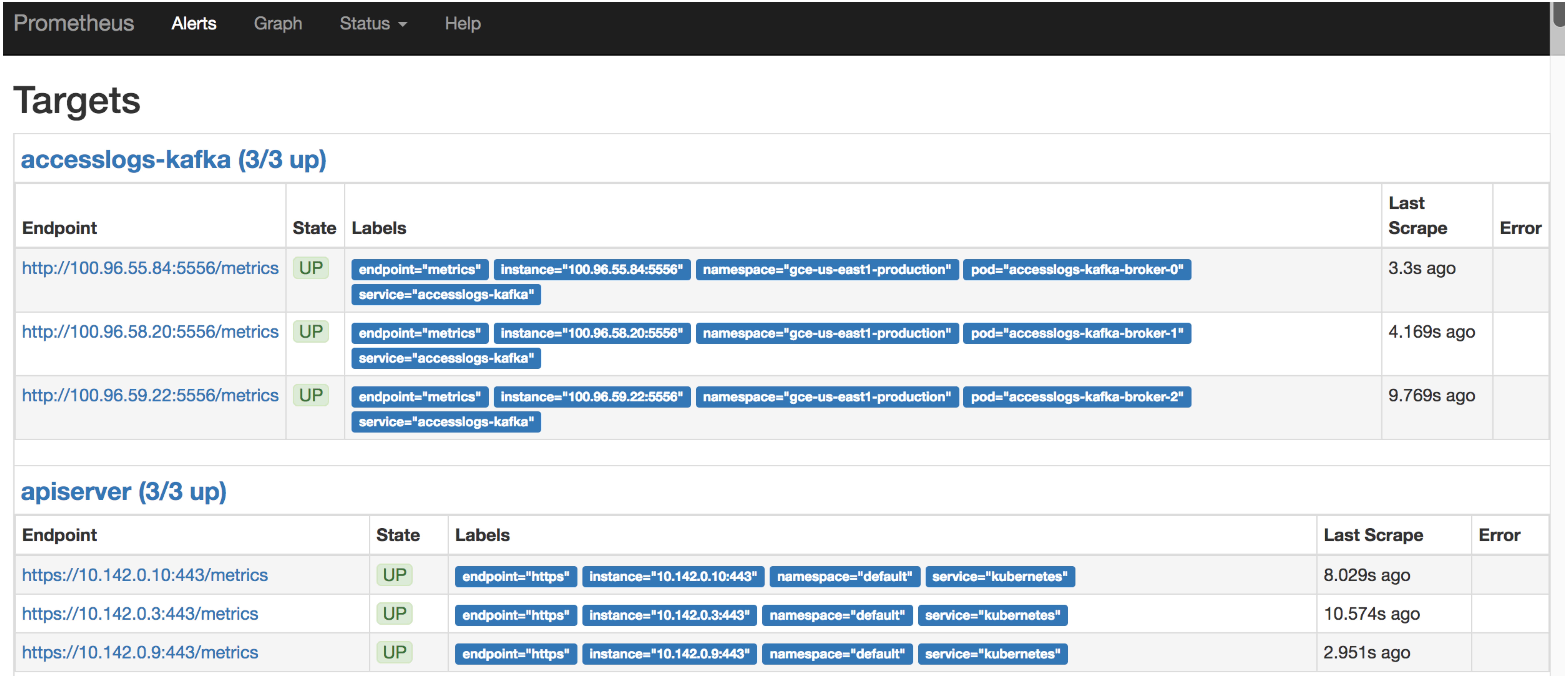
3) Scrape whatever port is named “metrics”

Prometheus: Monitored Service

```
apiVersion: v1
kind: Service
metadata:
  name: kafka
  namespace: default
  labels:
    app: kafka
    monitoring: core
```

1) Simply add the “monitoring: core” label to a server

Services Scraped



Targets

accesslogs-kafka (3/3 up)

Endpoint	State	Labels	Last Scrape	Error
http://100.96.55.84:5556/metrics	UP	endpoint="metrics" instance="100.96.55.84:5556" namespace="gce-us-east1-production" pod="accesslogs-kafka-broker-0" service="accesslogs-kafka"	3.3s ago	
http://100.96.58.20:5556/metrics	UP	endpoint="metrics" instance="100.96.58.20:5556" namespace="gce-us-east1-production" pod="accesslogs-kafka-broker-1" service="accesslogs-kafka"	4.169s ago	
http://100.96.59.22:5556/metrics	UP	endpoint="metrics" instance="100.96.59.22:5556" namespace="gce-us-east1-production" pod="accesslogs-kafka-broker-2" service="accesslogs-kafka"	9.769s ago	

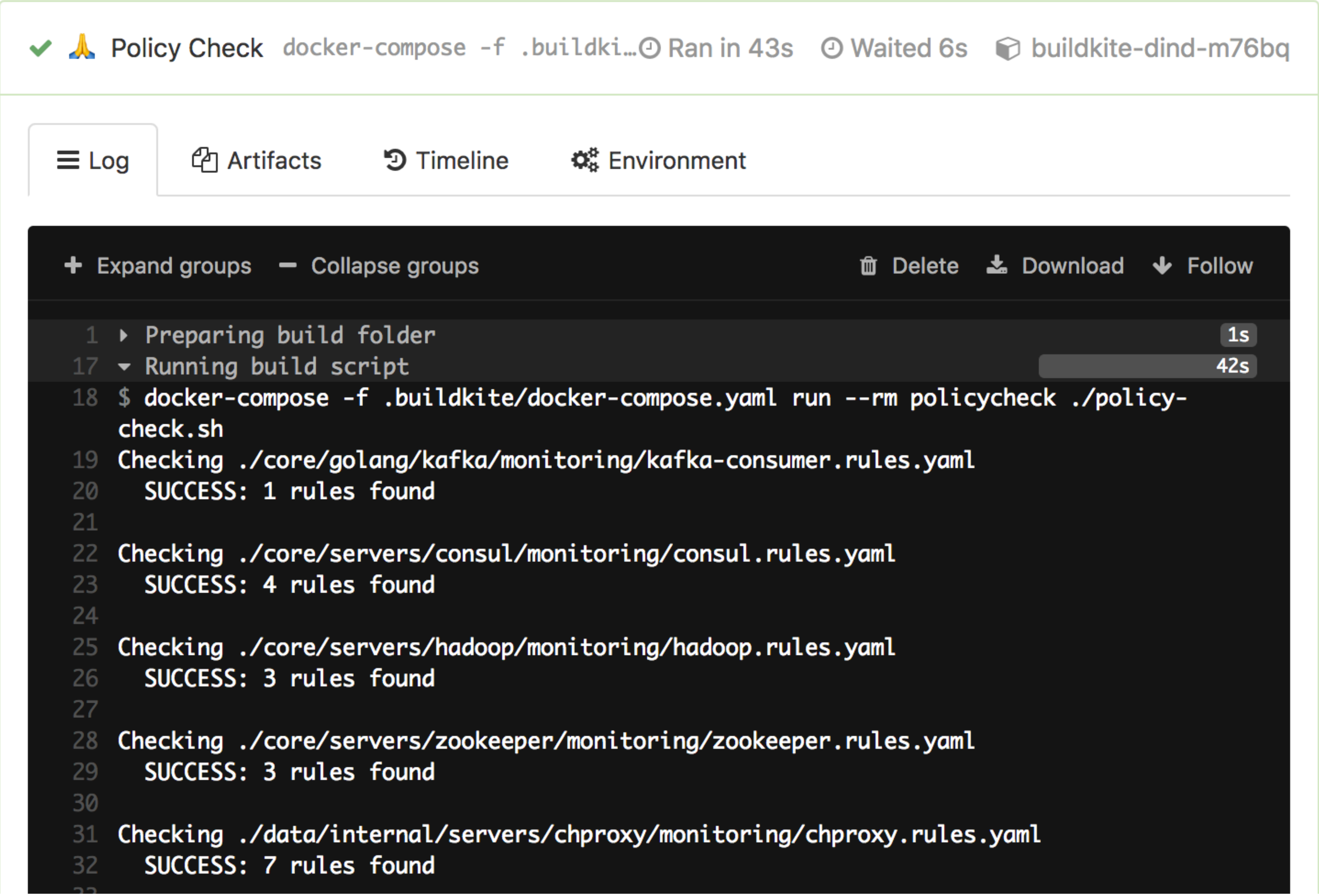
apiserver (3/3 up)

Endpoint	State	Labels	Last Scrape	Error
https://10.142.0.10:443/metrics	UP	endpoint="https" instance="10.142.0.10:443" namespace="default" service="kubernetes"	8.029s ago	
https://10.142.0.3:443/metrics	UP	endpoint="https" instance="10.142.0.3:443" namespace="default" service="kubernetes"	10.574s ago	
https://10.142.0.9:443/metrics	UP	endpoint="https" instance="10.142.0.9:443" namespace="default" service="kubernetes"	2.951s ago	

Goal #2

Run policy checks on alert rules with each build

Prometheus: Automated Policy Check



```
▸ Preparing build folder
▾ Running build script
$ docker-compose -f .buildkite/docker-compose.yaml run --rm policycheck ./policy-
check.sh
Checking ./core/golang/kafka/monitoring/kafka-consumer.rules.yaml
  SUCCESS: 1 rules found

Checking ./core/servers/consul/monitoring/consul.rules.yaml
  SUCCESS: 4 rules found

Checking ./core/servers/hadoop/monitoring/hadoop.rules.yaml
  SUCCESS: 3 rules found

Checking ./core/servers/zookeeper/monitoring/zookeeper.rules.yaml
  SUCCESS: 3 rules found

Checking ./data/internal/servers/chproxy/monitoring/chproxy.rules.yaml
  SUCCESS: 7 rules found
```

Prometheus: Automated Policy Check

1) Use promtool to validate alert rules files

2) Verify that all files end with a new-line to allow for concatenation

```
MONITORING_YAML_FILES=$(find $searchdir -type f -name
"*.rules.yaml" | sort)
for f in $MONITORING_YAML_FILES; do
  promtool check rules $f
  rc=$?
  if [[ $rc != 0 ]]; then
    echo "$f is not a valid Prometheus alert rule
YAML file."
    echo ""
    ERRORS="yes"
  fi
  LAST_CHAR=$(cat $f | tr '\n' '#' | tail -c 1)
  if [[ $LAST_CHAR != "#" ]]; then
    echo "$f does not end with a new line."
    echo ""
    ERRORS="yes"
  fi
done
```

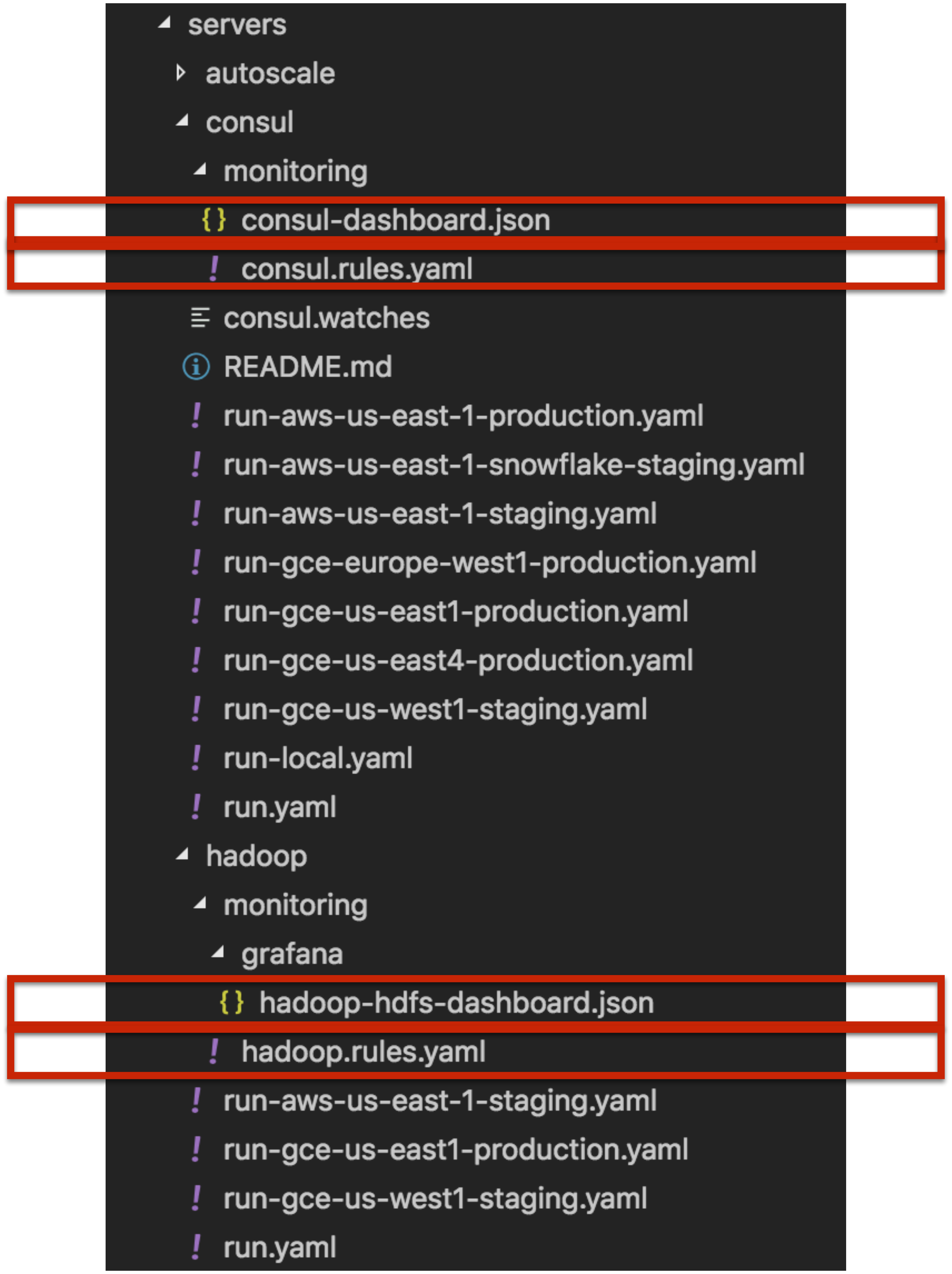
Goal #3

Store the dashboards and alert rules alongside code

Code Organization

1) Dashboards are named “*-dashboard.json”, and stored in a “monitoring/grafana” directory for the associated component

2) Alert rules are named “*.rules.yaml” and kept in a “monitoring” directory



```
servers
  autoscale
  consul
    monitoring
      consul-dashboard.json
      consul.rules.yaml
    consul.watches
    README.md
    run-aws-us-east-1-production.yaml
    run-aws-us-east-1-snowflake-staging.yaml
    run-aws-us-east-1-staging.yaml
    run-gce-europe-west1-production.yaml
    run-gce-us-east1-production.yaml
    run-gce-us-east4-production.yaml
    run-gce-us-west1-staging.yaml
    run-local.yaml
    run.yaml
  hadoop
    monitoring
      grafana
        hadoop-hdfs-dashboard.json
      hadoop.rules.yaml
    run-aws-us-east-1-staging.yaml
    run-gce-us-east1-production.yaml
    run-gce-us-west1-staging.yaml
    run.yaml
```

Goal #4

Automatically deploy dashboards and alert rules to Kubernetes clusters each time we ship code

Automatic Deployment of Dashboards and Alert Rules

- Our Buildkite builds automatically generate Kubernetes manifest for servers across all target environments
- Also generate Kubernetes ConfigMaps with Grafana dashboards and Prometheus alert rules
- Buildkite deploy plan applies Kubernetes manifests and ConfigMaps to each Kubernetes cluster
- Grafana and Prometheus ConfigMaps automatically reloaded

Gather Alerting Rules

```
#!/bin/bash

set -e
RULES_DIR=$1
mkdir -p $OUTPUT_DIR
rm -r $OUTPUT_DIR/* || true

for searchdir in "${@:2}"; do
      RULES_FILES=$(find $searchdir
-type f -name "*.rules.yaml")
      for file in $RULES_FILES; do
            cp $file $RULES_DIR
      done
done
```

1) Find all alert rules files conforming to naming pattern

Generate Kubernetes ConfigMap with Alert Rules

1) Begin rendering a Kubernetes ConfigMap manifest

2) Concatenate contents of each alert rule file to the ConfigMap

```
set -e
NAMESPACE=$1
OUTPUT_FILE=$2
RULES_DIR=$3

mkdir -p $(dirname $OUTPUT_FILE)

cat <<-EOF > $OUTPUT_FILE
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-k8s-rules
  namespace: $NAMESPACE
  labels:
    role: prometheus-rulefiles
    prometheus: k8s
data:
EOF

for f in $(find $RULES_DIR -type f -name
"*.rules.yaml")
do
  echo "  $(basename $f): |+" >> $OUTPUT_FILE
  cat $f | sed "s/^/    /g" >> $OUTPUT_FILE
done
```

Automatic Deployment of Prometheus Alert Rules

- Prometheus Operator includes a config reloader that monitors the ConfigMap for changes

- Sends web hook to Prometheus instructing it to reload its config

Gather Grafana Dashboards and Datasources

1) Find all Grafana dashboard and datasource files conforming to naming pattern

```
#!/bin/bash
set -e



OUTPUT_DIR=$1
mkdir -p $OUTPUT_DIR
rm -r $OUTPUT_DIR/* || true

for searchdir in "${@:2}"; do
      DASHBOARD_FILES=$(find $searchdir -type
f -name "*-dashboard.json" -o -name "*-
datasource.json" | sort)
      for file in $DASHBOARD_FILES; do
        echo "FILE: $file"
            cp $file $OUTPUT_DIR
        done
done
```

Render ConfigMap with Grafana Dashboards

Have been using the `grafana-dashboards-configmap-generator` script at https://github.com/eedugon/grafana-dashboards-configmap-generator

```
  monitoring/grafana/grafana-dashboards-configmap-generator/bin/grafana_dashboards_generate.sh \
    -n ${MONITORING_NAMESPACE} \
    -s 200000 \
    -o run/k8s ${NAMESPACE}/monitoring/grafana/config-map.yaml \
    -g run/k8s/${NAMESPACE}/monitoring/grafana/run.yaml \
    -i run/monitoring/grafana \
    --hostname ${GRAFANA_HOSTNAME}
```

Grafana Watcher to reload Dashboards

```
- name: grafana-watcher
  image: quay.io/coreos/grafana-watcher:v0.0.8
  args:
    - '--watch-dir=/var/grafana-dashboards-0'
    - '--grafana-url=http://localhost:3000'
  env:
  - name: GRAFANA_USER
    valueFrom:
      secretKeyRef:
        name: grafana-credentials
        key: user
  - name: GRAFANA_PASSWORD
    valueFrom:
      secretKeyRef:
        name: grafana-credentials
        key: password
  volumeMounts:
  - name: grafana-dashboards-0
    mountPath: /var/grafana-dashboards-0
volumes:
- name: grafana-storage
  emptyDir: {}
- name: grafana-dashboards-0
  configMap:
    name: grafana-dashboards-0
```

1) Use ‘grafana-watcher’ container to reload Grafana dashboards supplied in ConfigMap volume

Next Steps

- Replace ‘grafana-watcher’ pod with Grafana provider config that automatically reloads dashboards from ConfigMap volume path
- Control over which dashboards are deployed; some Grafana instances have dashboards that are unused or point to non-existent servers

Credit to the Mux Team

Adam Brown

Matt Ward

Thank You!