The current Cloud Native Observability *dogma* is that metrics (and logs and traces) are "**not good enough**" and that this brave new world needs brave new Observability tools.

This is **false**.

Who is this guy

Tim Simmons

Sr. Engineer/Prometheus person

Observability Platforms

DigitalOcean

@timsimlol

Tim Simmons
@timsimlol

made it to May 4th without touching a coin this year...really makes you think....

1:29 PM · May 4, 2019 · Tweetbot for iOS

Prometheus at DigitalOcean

400 Prometheus servers

300M+ time series

2.5M+ samples/second

The current Cloud Native Observability *dogma* is that metrics (and logs and traces) are "**not good enough**" and that this brave new world needs brave new Observability tools.

This is **false**.

who is this guy?

WOW

The current Cloud Native Observability *dogma* is that metrics (and logs and traces) are “**not good enough**” and that this brave new world needs brave new Observability tools.

This is **false**.

HOT TAKE ALERT

The current Cloud Native Observability *dogma* is that metrics (and logs and traces) are “**not good enough**” and that this brave new world needs brave new Observability tools.

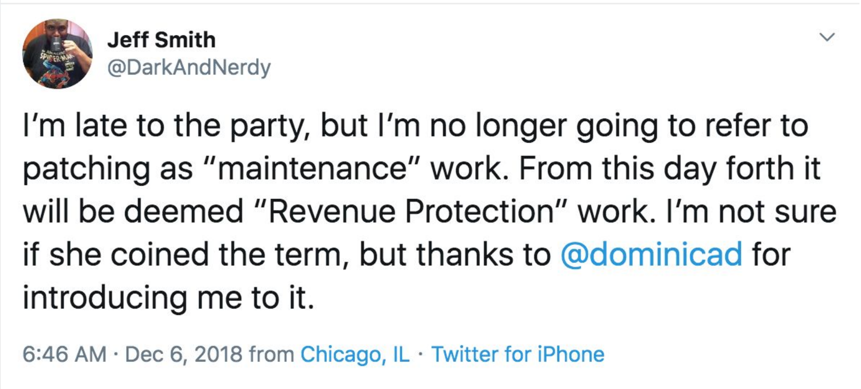
This is **false** for ~~everybody most of the time~~ everybody all the time if you understand the tools you’re using and have an inkling of the outcomes you’re looking for.

The thing that is “not good enough” is **us**.

“I don’t have **time** to learn these complicated tools”

These cOmPLiCaTeD tOolS tell you how your *complex* software is serving customers that are giving you **money**.

Isn’t that worth your **time** and **energy**? What is more important than **protecting revenue**?



Jeff Smith
@DarkAndNerdy

I’m late to the party, but I’m no longer going to refer to patching as “maintenance” work. From this day forth it will be deemed “Revenue Protection” work. I’m not sure if she coined the term, but thanks to @dominicad for introducing me to it.

6:46 AM · Dec 6, 2018 from Chicago, IL · Twitter for iPhone

https://twitter.com/DarkAndNerdy/status/1070690952472350720

The hardest problem in **observability** is convincing your leaders that it is worth the investment over new features.

You do this by investing **time** and **energy** into instrumenting your software.

The *best* way to do that is to understand the **tools** so you can produce the **best** outcomes.

What are we doing here today?

Put some respect on Observability’s name

Apply the mindset to some problems Prometheus can solve

Despite what you’ve potentially heard to the contrary

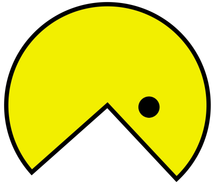
Metrics

How many/much/long?

Great for alerts

Apps make metrics

Prometheus eats metrics



```
# HELP http_request_duration_seconds request duration histogram
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.5"} 0
http_request_duration_seconds_bucket{le="1"} 1
http_request_duration_seconds_bucket{le="2"} 2
http_request_duration_seconds_bucket{le="3"} 3
http_request_duration_seconds_bucket{le="5"} 3
http_request_duration_seconds_bucket{le="+Inf"} 3
http_request_duration_seconds_sum 6
http_request_duration_seconds_count 3
```

Observability

Patterns need to be **thoughtfully** utilized

Central to software design

Have to understand the specific tools

Observability

“Beyond the three pillars”

Difficult without core understanding and experience

When you become deeply familiar, you can understand limitations beyond marketing literature

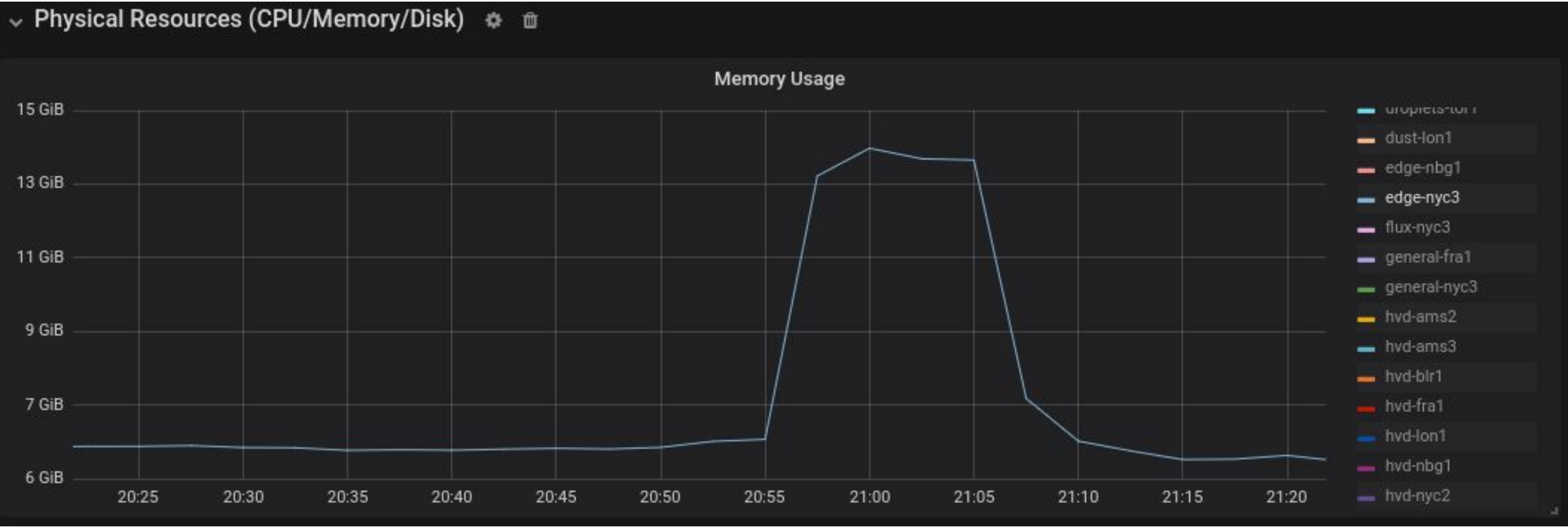
“Doing Things Prometheus Can’t Do with Prometheus”

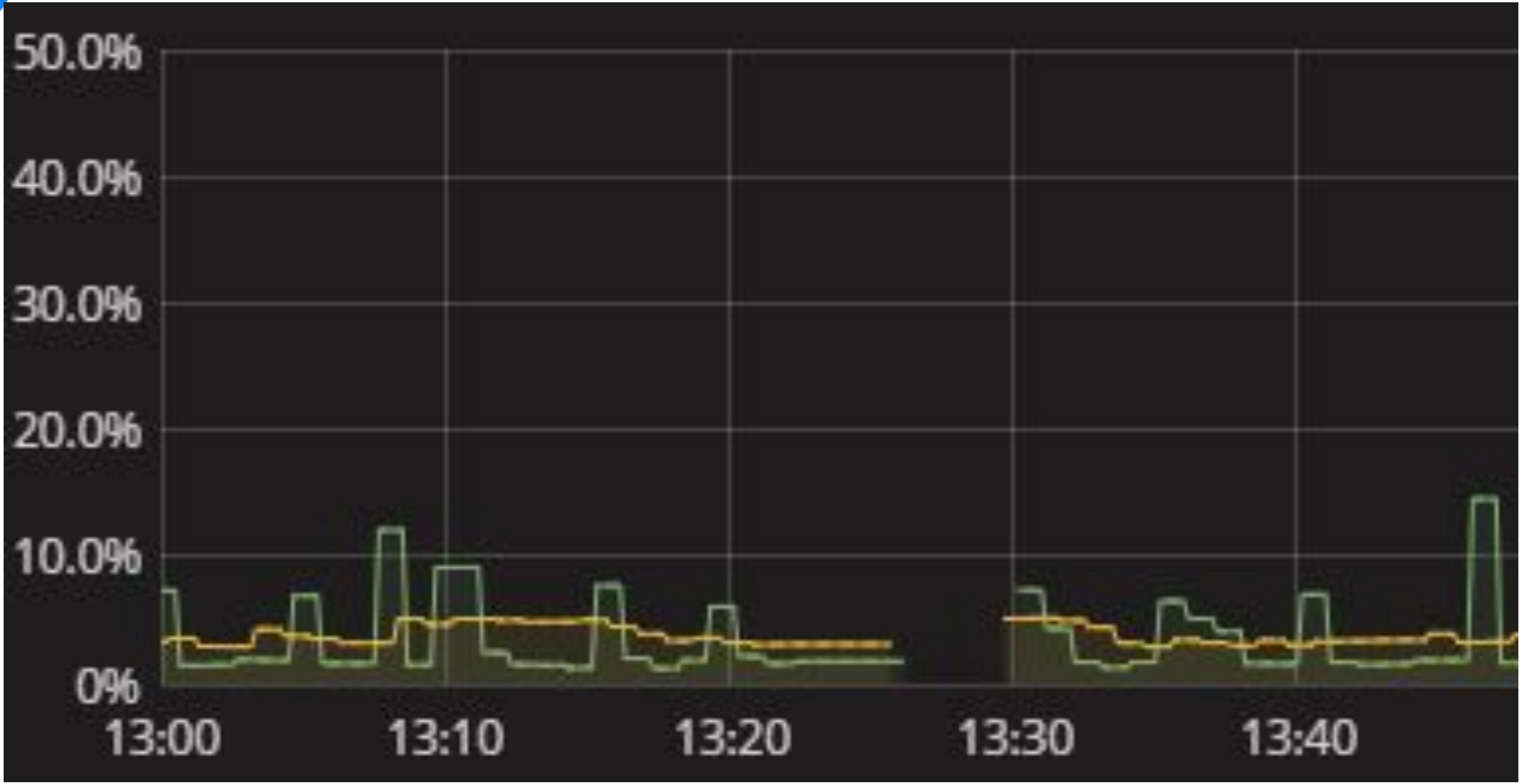
High Availability Prometheus

Prometheus is not distributed

Sometimes the network breaks

Sometimes queries make Prometheus sad :(


Physical Resources (CPU/Memory/Disk)
Memory Usage
15 GiB
13 GiB
11 GiB
9 GiB
7 GiB
6 GiB
20:25
20:30
20:35
20:40
20:45
20:50
20:55
21:00
21:05
21:10
21:15
21:20
droplets-lon1
dust-lon1
edge-nbg1
edge-nyc3
flux-nyc3
general-fra1
general-nyc3
hvd-ams2
hvd-ams3
hvd-blr1
hvd-fra1
hvd-lon1
hvd-nbg1
hvd-nyc2


50.0%
40.0%
30.0%
20.0%
10.0%
0%
13:00
13:10
13:20
13:30
13:40

High Availability Prometheus

Deploy replicas and protect Prometheus

Smart proxy (NGINX, HAProxy)

Intelligently switch instances for queries

Get some extra features for free

High Availability “Prometheus”

thanos-query, promxy fan out to fill gaps

Decreased query performance

Big queries fanning out

Operational overhead

Cardinality

Every **permutation of labels** in Prometheus creates a new time series

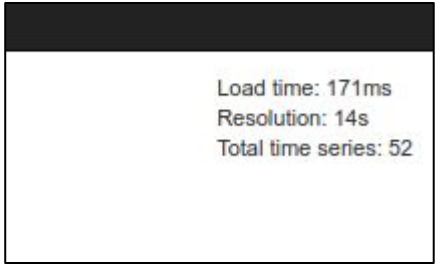
Individual queries should use **hundreds not thousands** of time series **(at most)**

Queries that **operate on** thousands of time series will overload Prometheus

Work out your query in the **Console** before graphing

Avoid **high cardinality** labels*

*unless you *really* know what you’re doing



Load time: 171ms
Resolution: 14s
Total time series: 52

query: requests_total{path=~”(/status|/)”, method=~”(GET|POST)”}

```
{__name__="requests_total", path="/status", method="GET", instance=”10.0.0.1:80”}
{__name__="requests_total", path="/status", method="POST", instance=”10.0.0.3:80”}
{__name__="requests_total", path="/", method="GET", instance=”10.0.0.2:80”}
```

2000
timeseries

Effective Cardinality - protec

Leave resource headroom

`--query.max-samples`

`--query.max-concurrency`

Shard on logical boundaries or federate

Effective Cardinality - attac

Use aggregation, query step size, and time windows carefully

You must understand your query pattern, the expense, and resource accordingly

It’s hard, but deal with it

Effective Cardinality - Example

HTTP service latency per endpoint with 1000s of endpoints

Create a metric with a label per endpoint and only query individuals

Create a *separate* metric for global, semi-global latency

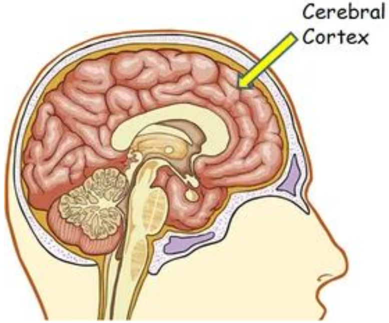
Separate questions sometimes deserve separate metrics, more metrics isn't **always** bad

Long-term Metrics Storage

Prometheus 2.8 released disk-backed retention

Keep `storage.tsdb.max-block-duration` down to ~3d-7d

I did a query over 250d, ~500 timeseries in ~10s.
+50% RAM usage for a couple hours.



Long-term Metrics Storage

New solutions in the space — Thanos, M3, Cortex, etc

This is a new animal — operational overhead

LTS is something everyone “wants” but is rarely used

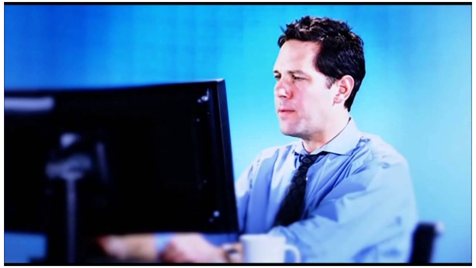
Mindful data and vanilla Prometheus could be all you need

Long-term Metrics Storage

Mindful data and vanilla Prometheus could be all you need

Block storage or a fat disk

Separate “long term” server



“Machine Learning” on Metrics

ML/Anomaly Detection is a common idea with metrics

Personally I’ve heard lots of promises, not much results

PromQL has enough features for you to impress your boss

“Machine Learning” on Metrics

Show me average worker latencies over the last 5m greater than…

The average of all worker latencies

Plus two standard deviations

That are also more than 20% higher than the average over the last 5m to eliminate false positives

https://prometheus.io/blog/2015/06/18/practical-anomaly-detection/

```
(
  instance:latency_seconds:mean5m
  > on (job) group_left()
    (
      avg by (job)
        (instance:latency_seconds:mean5m)
      + on (job)
        2 * stddev by (job)
        (instance:latency_seconds:mean5m)
    )
)
> on (job) group_left()
1.2 * avg by (job)
  (instance:latency_seconds:mean5m)
```

“Machine Learning” on Metrics

```
instance:latency_seconds:mean5m
>
(1.2 * avg_over_time(
    instance:latency_seconds:mean5m[24h] offset 5m)
)
```

Compare the last few minutes to a longer time period average to find if the last few minutes are an outlier.

“Machine Learning” on Metrics

`predict_linear` catches problems before they happen

Predict the future value of a timeseries based on past

```
predict_linear(node_filesystem_free{job="node"}[1h], 4 * 3600) < 0
```

Good for any sort of “saturation” metric. Capacity, fullness, etc

Measuring Everything with Prometheus

Exporters are apps that expose Prometheus metrics

They exist for lots of things. https://prometheus.io/docs/instrumenting/writing_exporters/

Standardizing on metrics >>>>>> bespoke bash scripts invoked by your monitoring system

Exporters - The “gotcha”

Often when you create a metric in a Prometheus language client, it stays in /metrics *forever*

Labels with ephemeral values stay around

Pattern of creating metrics at scrape-time called “ConstMetrics” in Go lib. **Use this when you’re exporting metrics from a separate source of truth.**

Tim Simmons
@timsimlol

@PrometheusIO do you like my alertmanager routing tree?

Alerts

Create team/service level alerting receivers

Abstract this and generate the config

Try not to deviate from the existing patterns

Read the literature — actionable, contextual alerts

https://juliusv.com/promslack/

Observability Culture with SLOs

https://landing.google.com/sre/sre-book/chapters/service-level-objectives/

tl;dr: define performance experience, measure it, maintain it

Quantifying user experience is hard, being outcomes-oriented

Prometheus can help you implement an error budgets system

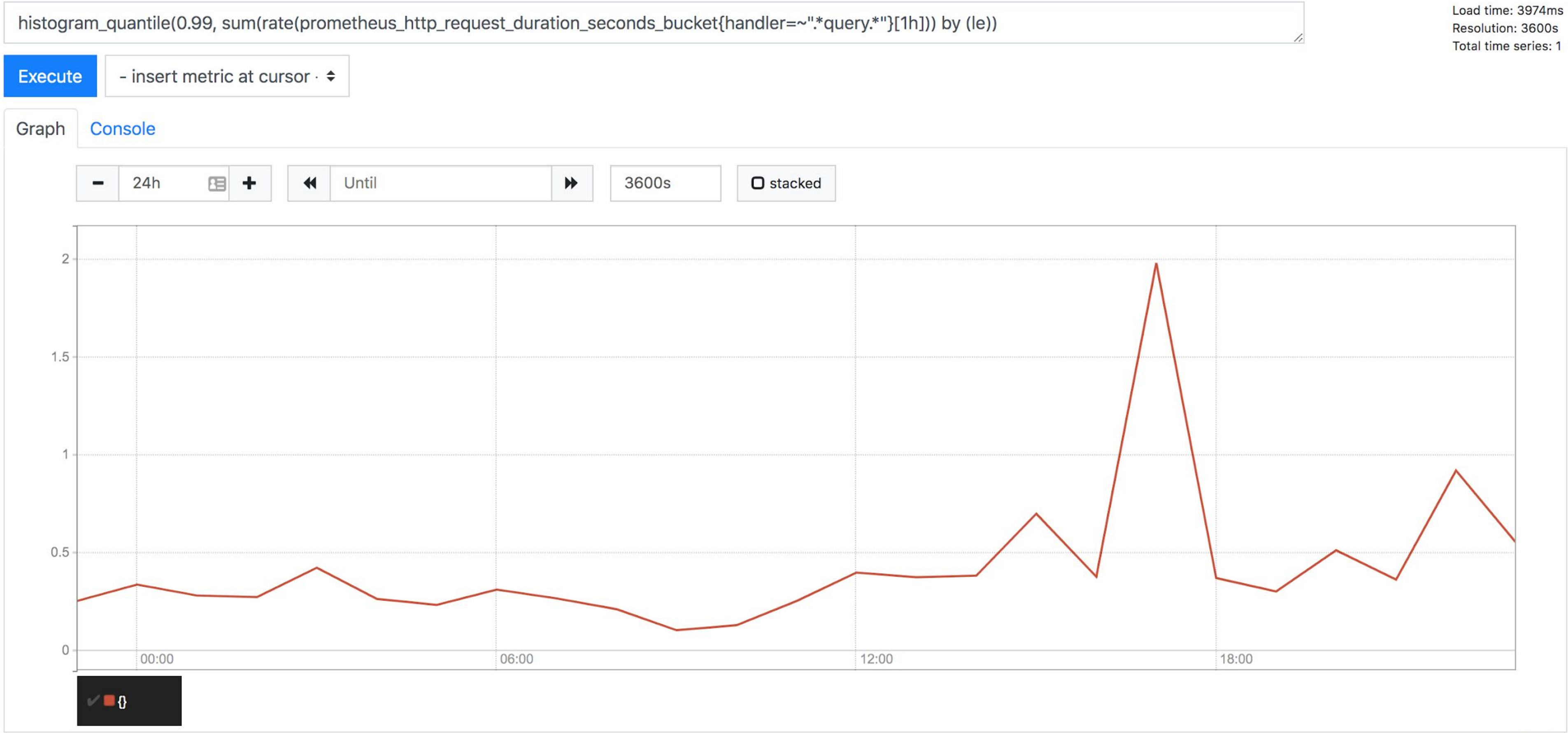
Observability Culture with SLOs - Error Budgets

tl;dr: define performance experience, measure it, maintain it

Once measured, set a target and maintain it

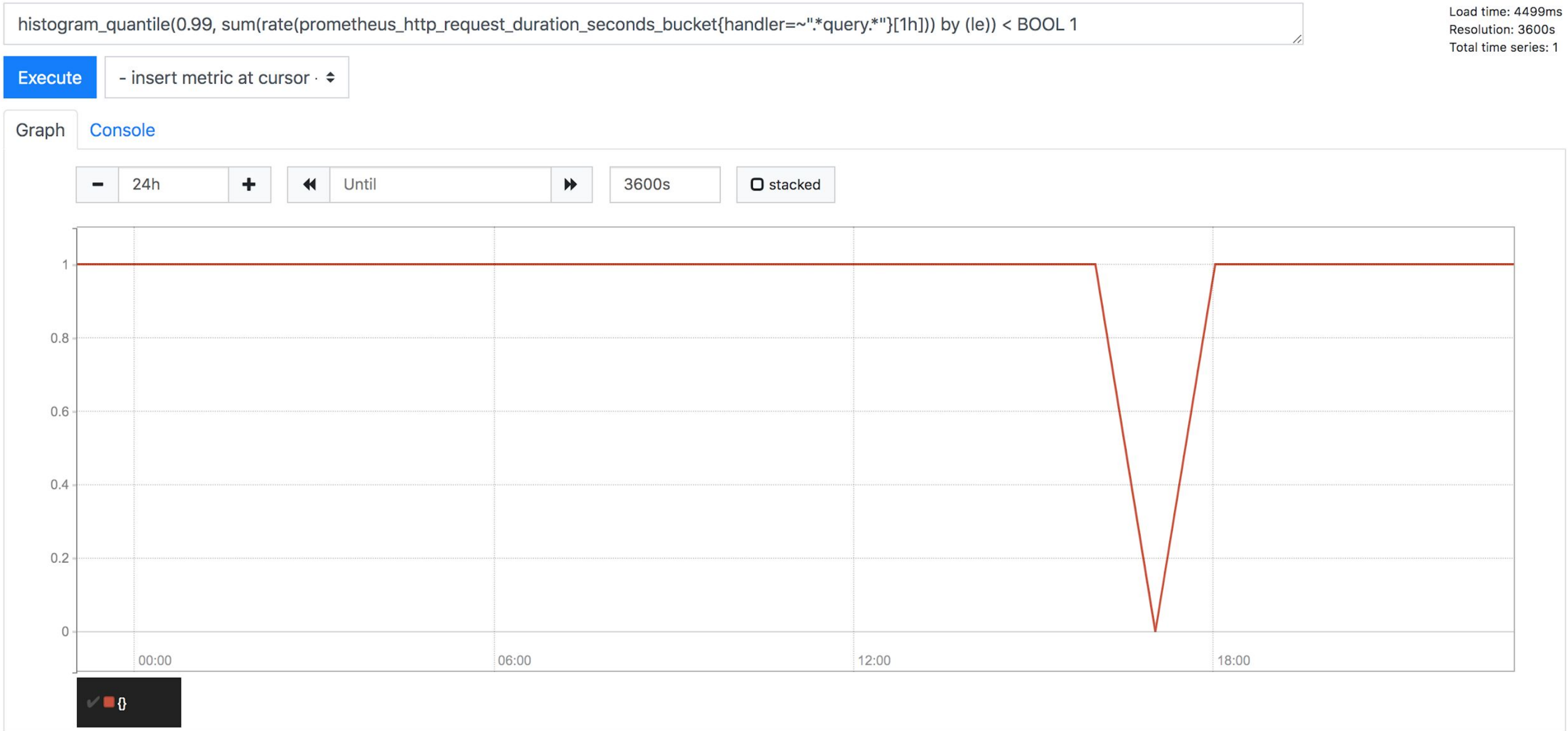
Prometheus can help you implement a system

SLI: 99.999 percent of requests that complete in < 1 second
SLO: 99%
Error Budget: 1 percent
Human speak: 99 percent of the time, almost all requests should be completing in <1s



```
histogram_quantile(0.99, sum(rate(prometheus_http_request_duration_seconds_bucket{handler=~".*query.*"}[1h])) by (le))
```

Load time: 3974ms
Resolution: 3600s
Total time series: 1



```
histogram_quantile(0.99, sum(rate(prometheus_http_request_duration_seconds_bucket{handler=~".*query.*"}[1h])) by (le)) < BOOL 1
```

Load time: 4499ms
Resolution: 3600s
Total time series: 1

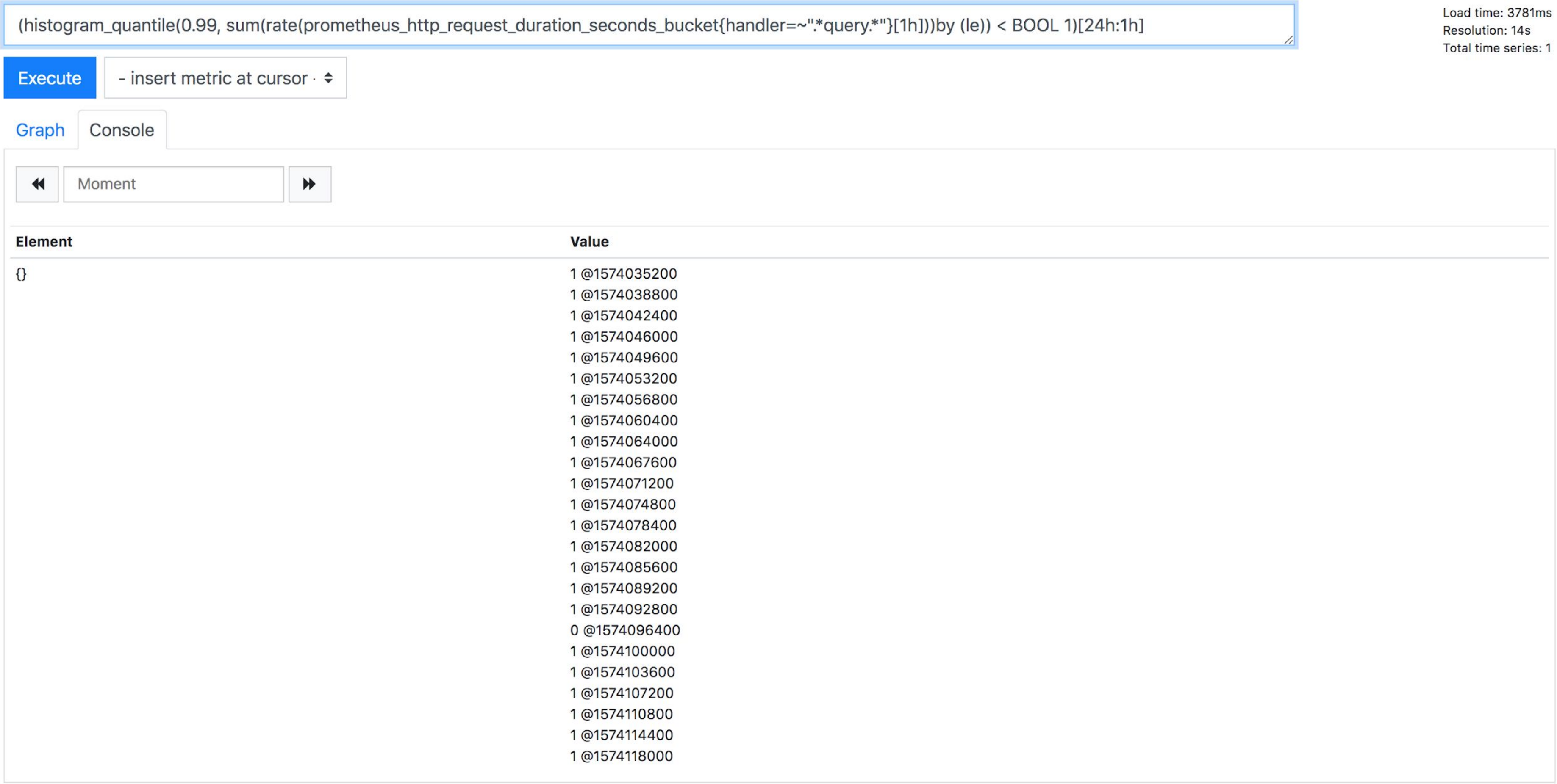
24h

Until

3600s

□ stacked

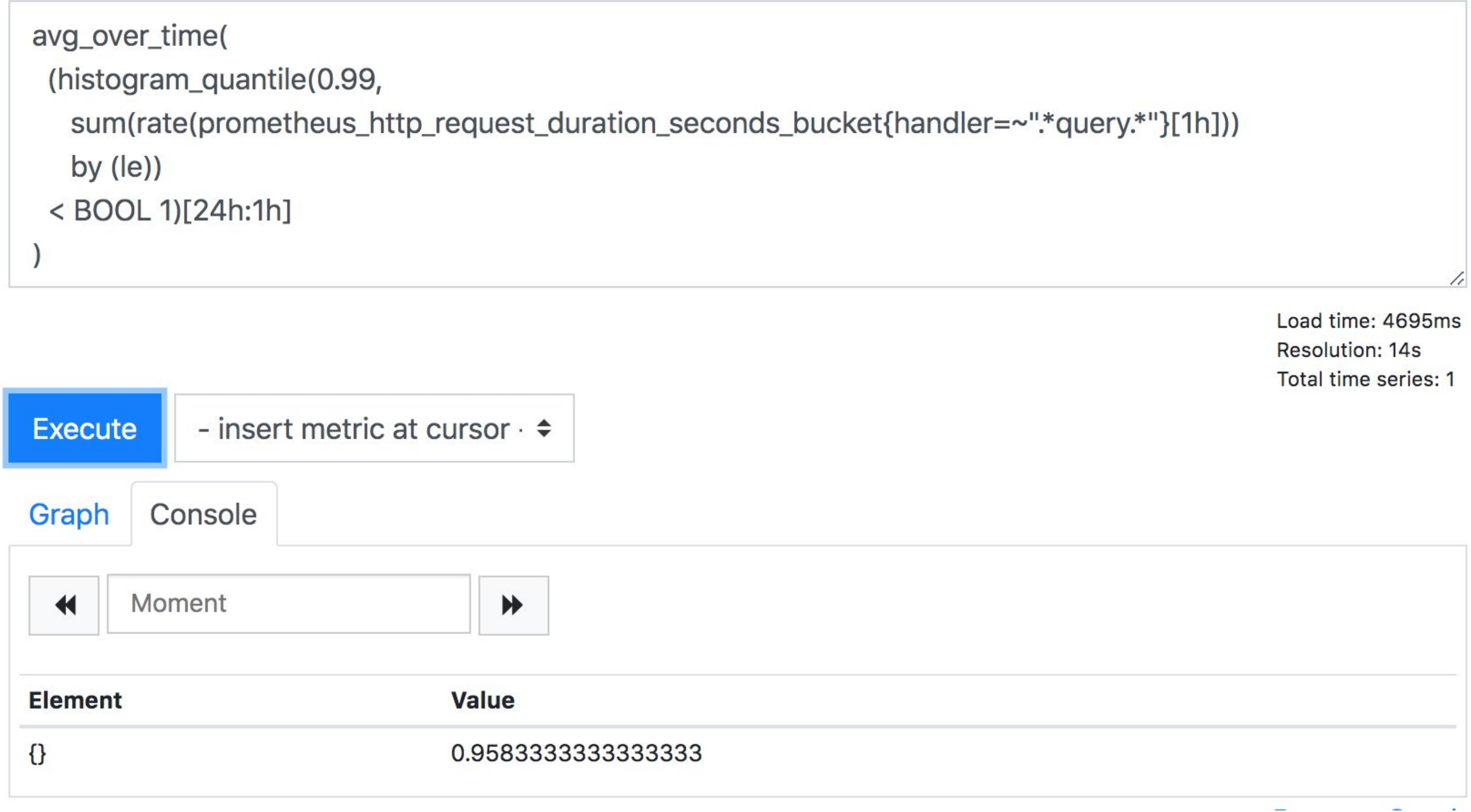
✓ 🟥 {}



```
(histogram_quantile(0.99, sum(rate(prometheus_http_request_duration_seconds_bucket{handler=~".*query.*"}[1h]))by (le)) < BOOL 1)[24h:1h]
```

Load time: 3781ms
Resolution: 14s
Total time series: 1

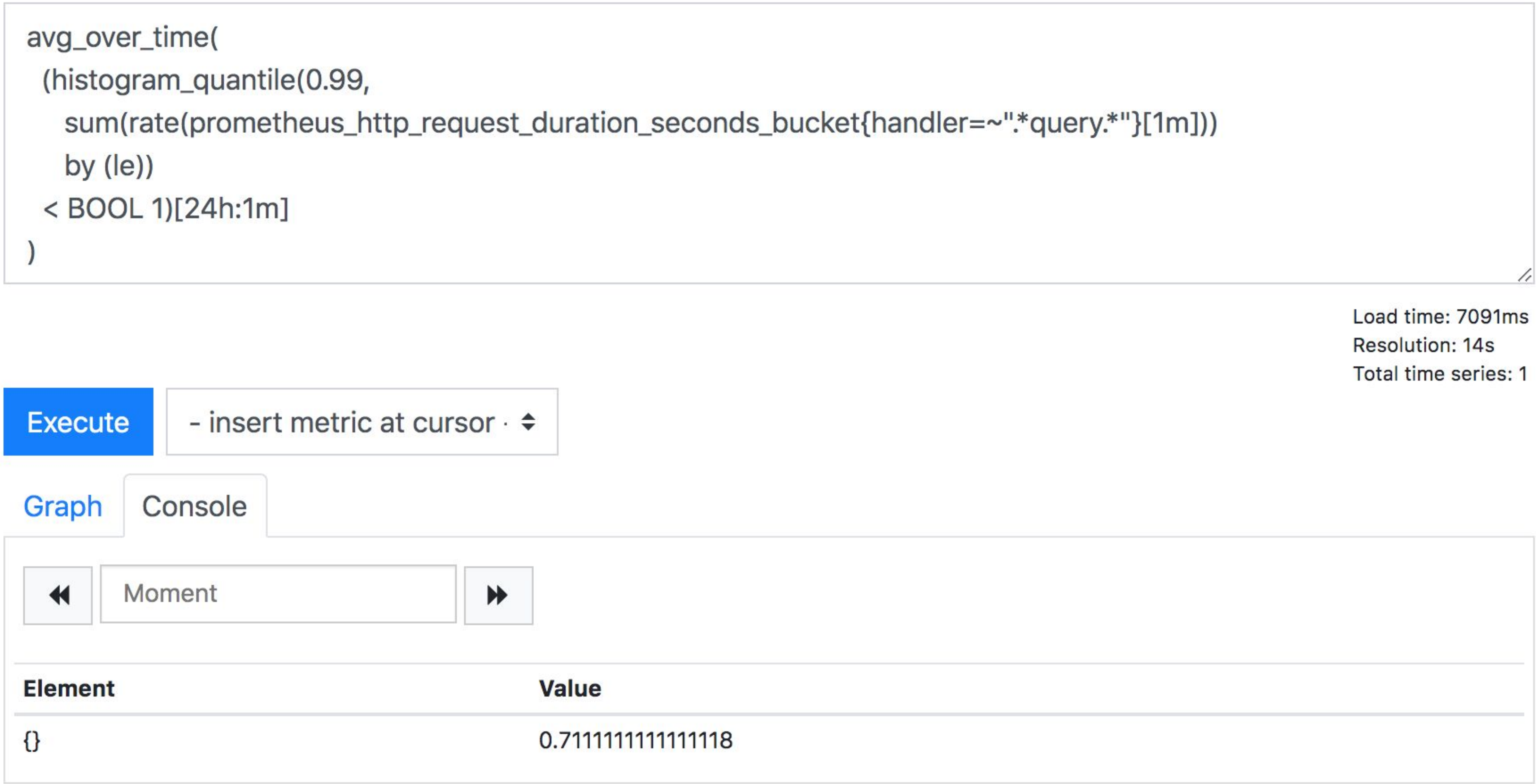
```
Element                 Value
{}                      1 @1574035200
                        1 @1574038800
                        1 @1574042400
                        1 @1574046000
                        1 @1574049600
                        1 @1574053200
                        1 @1574056800
                        1 @1574060400
                        1 @1574064000
                        1 @1574067600
                        1 @1574071200
                        1 @1574074800
                        1 @1574078400
                        1 @1574082000
                        1 @1574085600
                        1 @1574089200
                        1 @1574092800
                        0 @1574096400
                        1 @1574100000
                        1 @1574103600
                        1 @1574107200
                        1 @1574110800
                        1 @1574114400
                        1 @1574118000
```



```
avg_over_time(
 (histogram_quantile(0.99,
   sum(rate(prometheus_http_request_duration_seconds_bucket{handler=~".*query.*"}[1h]))
   by (le))
 < BOOL 1)[24h:1h]
)
```

Load time: 4695ms
Resolution: 14s
Total time series: 1

Element	Value
{}	0.9583333333333333



```
avg_over_time(
 (histogram_quantile(0.99,
   sum(rate(prometheus_http_request_duration_seconds_bucket{handler=~".*query.*"}[1m]))
   by (le))
 < BOOL 1)[24h:1m]
)
```

Load time: 7091ms
Resolution: 14s
Total time series: 1

Element	Value
{}	0.7111111111111118

Observability Culture with SLOs

Use a histogram for a more exact method of latency calculation (in this house we do not tolerate histogram estimation errors)

(rate extrapolation errors are chill)

Only problem is if you don't have requests for a time

```
(
  (
    sum(
     rate(
      latency_histogram{handler="gql", le="1"}[5m]
     )
    ) > 0
  )
  /
  (
    sum(
      rate(
        latency_histogram{handler="gql", le="+Inf"}[5m]
      )
    ) > 0
  )
) > BOOL .99
or
(
  1 +
  0 * sum(latency_histogram{handler="gql", le="+Inf"})
)
```

Observability Culture with SLOs

Sometimes you don’t have constant requests, so you need to sanitize your boolean to have “no requests” ⇒ 1

Percentage of time over the last 7d where 99% of requests completed under 1s

```
avg_over_time((
  (
    sum(
     rate(
      latency_histogram{handler="gql", le="1"}[5m]
     )
    ) > 0
  )
  /
  (
    sum(
      rate(
        latency_histogram{handler="gql", le="+Inf"}[5m]
      )
    ) > 0
  )
) > BOOL .99
or
(
  1 +
  0 * sum(latency_histogram{handler="gql", le="+Inf"})
))[7d:5m])
```

we did it everyone, this is the last slide

doing Observability *well* is hard but well worth it

requires investment, time, and energy

quality of input -> quality of output

pick some tools and learn them well, the pillars are great

Prometheus is very good