Service Level Objectives

In Practice And At Scale

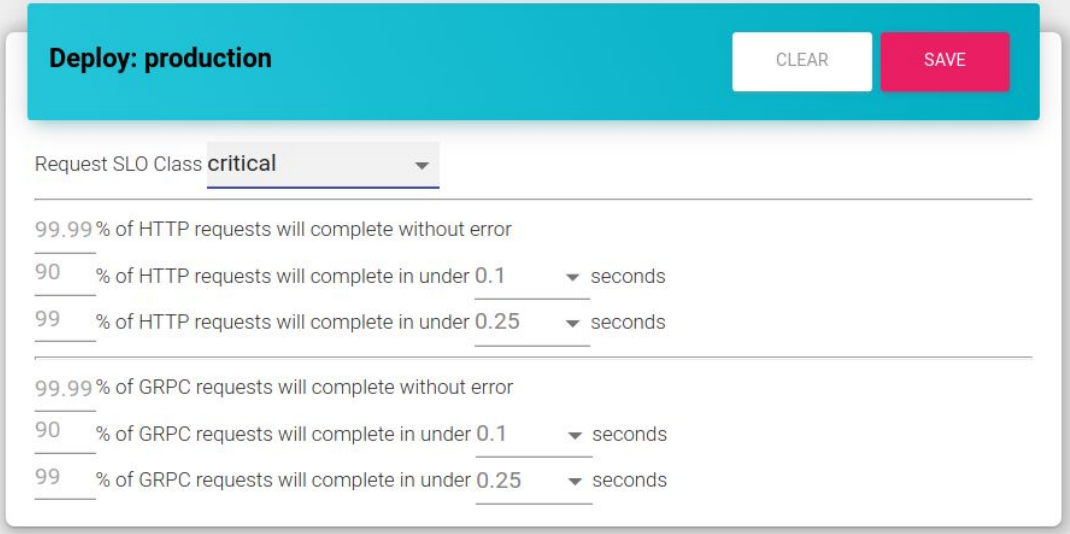
Carson Anderson
Weave

 @carsonoid

 @carson_ops

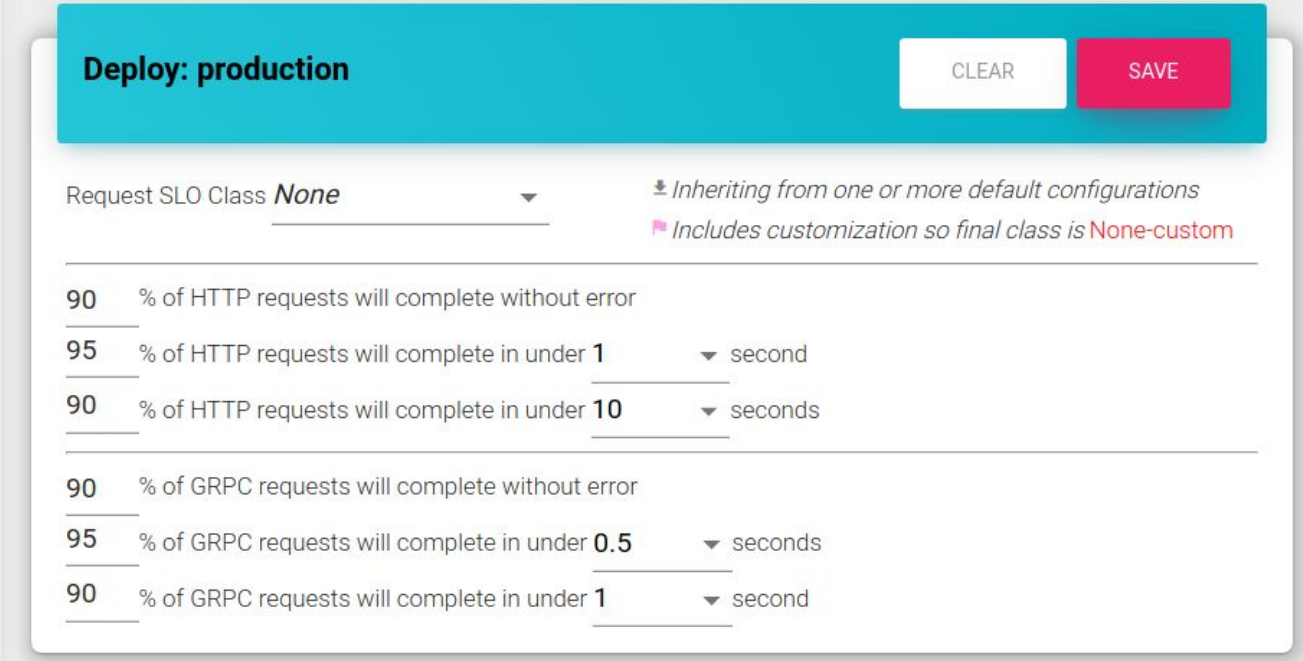
SLOs at Weave

Simple, Pre-Defined Objectives



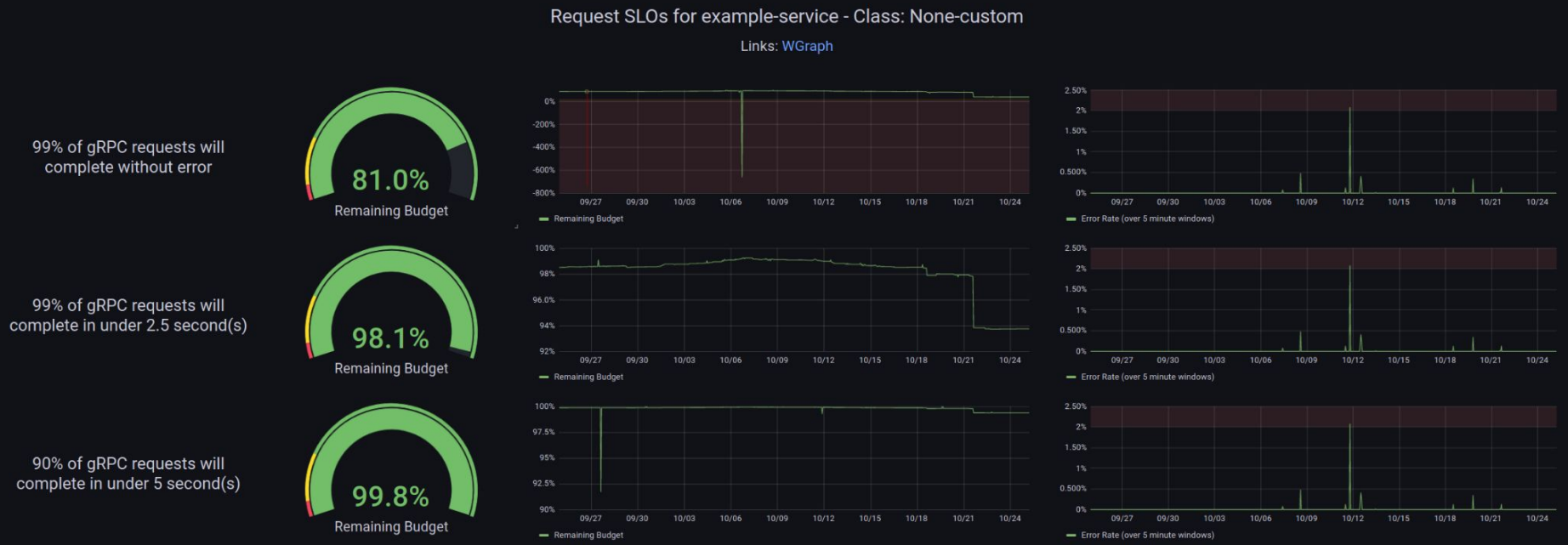

```
1 deploy:
2   production:
3     slo:
4       requestClass: critical
5
```

Fully Customized Objectives




```
deploy:
  production:
    slo:
      requestSpec:
        grpc:
          availability:
            percent: "90"
          latency:
            tier1:
              percent: "90"
              seconds: "1"
            tier2:
              percent: "95"
              seconds: "0.5"
        http:
          availability:
            percent: "90"
          latency:
            tier1:
              percent: "90"
              seconds: "10"
            tier2:
              percent: "95"
              seconds: "1"
```

Generated Dashboards



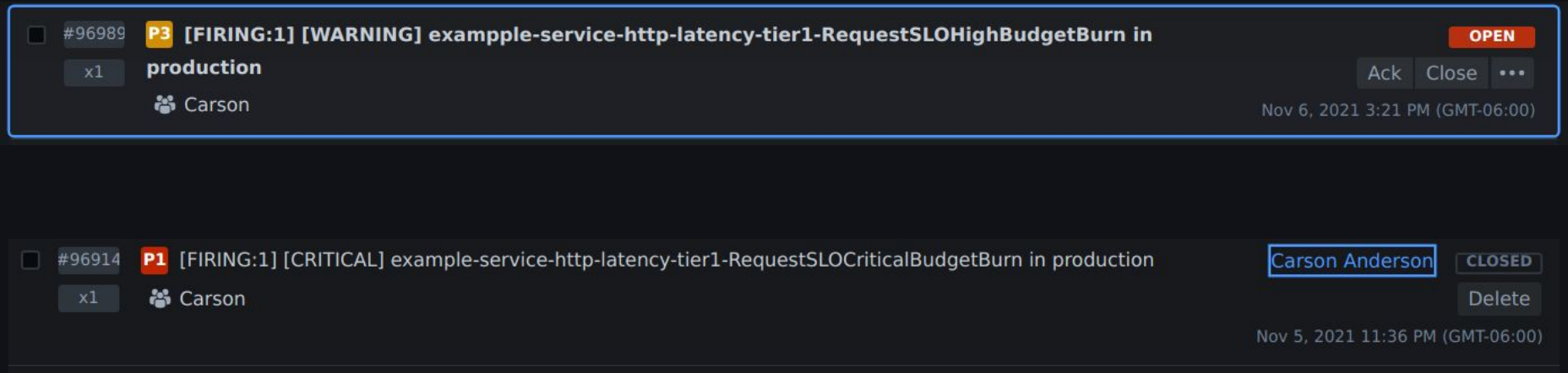
Generated Dashboards - Pt. 2

99% of HTTP requests will complete without error

99% of HTTP requests will complete in under 2.5 second(s)

90% of HTTP requests will complete in under 5 second(s)

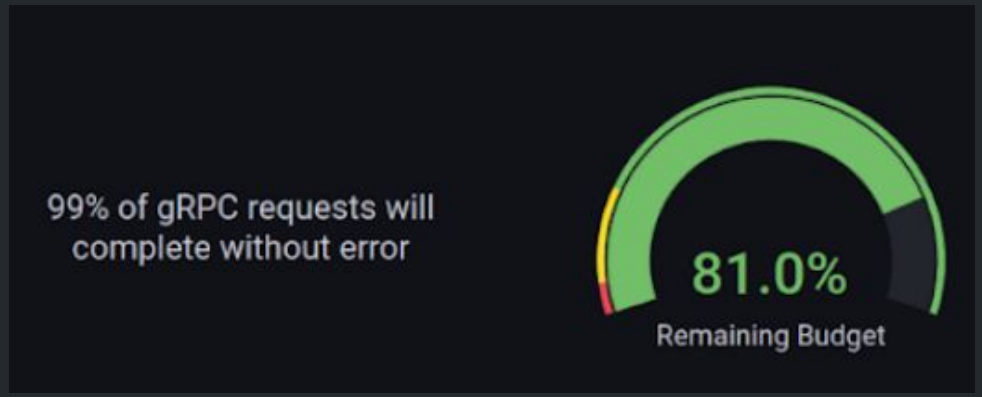
Generated Alerts



So How Do I Get There?

- Calculating budgets
- Creating great alerts
- Handle scale
- Handle cost

Calculating An SLO Budget
The Math



How to calculate an SLO

- Decide on an SLO window
  - Google SRE book: 30d
  - Weave: 28d (4 weeks)
- Calculate the number of objective "fails" allowed
  - Availability failure == response returns a server-side error
    - Ex: 500
  - Latency failure == response time < objective time
    - Ex: 2 seconds to respond
- Calculate the final budget % remaining
  - Based on total & budged

Calculating Budgeted Errors

Total * .1 = Budgeted

1- (Objective/100)
Ex: 1- (90/100)

50 * .1 = 5

Calculating the SLO %

$$\frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}} = \text{Budget}$$

$$\frac{5 - 0}{5} = 100\%$$

Basic SLO Budget Calculations

Total * .1 = Budgeted

1- (Objective/100)
Ex: 1- (90/100)

$$\frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}} = \text{Budget}$$

+50 Good Requests

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

Budgeted: 5 = Total: 50 * .1

$$\frac{5 - 0}{5} = 100\%$$

+1 Bad Request

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

Budgeted: 5 = Total: 51 * .1

$$100\% = \frac{5 - 0}{5} \Rightarrow 80\% = \frac{5 - 1}{5}$$

+3 Bad Requests

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

Budgeted: 5 = Total: 54 * .1

$$80\% = \frac{5 - 1}{5} \Rightarrow 20\% = \frac{5 - 4}{5}$$

+1 **Good** Request

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

Budgeted: 5 = Total: 55 * .1

$$20\% = \frac{5 - 4}{5} \Rightarrow 20\% = \frac{5 - 4}{5}$$

+5 **Good** Requests

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

Budgeted: 6 = Total: 60 * .1

$$20\% = \frac{5 - 4}{5} \Rightarrow 33\% = \frac{6 - 4}{6}$$

Recover With Time

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

28d

$$25 * .1 = \text{Budgeted:2}$$

Recover With Time

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

28d

$$\frac{2 - 4}{2} = -100\%$$

Recover With Time

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

28d

$$-100\% = \frac{2 - 4}{2} \Rightarrow 0\% = \frac{2 - 2}{2}$$

Key Takeaways

- Budgets can burn FAST
- Two ways to recover:
  - Time
  - Handle more "Good" request, a lot of them

Availability SLO Recording Rules Prometheus Specifics

Prometheus Metric Crash-Course - counters

```
# HELP http_timer_count The total number of HTTP requests.
# TYPE http_timer_count counter
http_timer_count{app="api", code="200", method="GET"} 1
http_timer_count{app="api", code="200", method="POST"} 10
http_timer_count{app="api", code="200", method="PATCH"} 2
http_timer_count{app="frontend", code="200", method="GET"} 110
http_timer_count{app="frontend", code="404", method="GET"} 23
http_timer_count{app="frontend", code="500", method="GET"} 2
```

Example Latency Histogram in Prometheus

```
# HELP http_timer_bucket A histogram of request duration
# TYPE http_timer_bucket histogram
http_timer_bucket{app="api", code="200", method="GET", le="+Inf"}
...
```

Example Latency Histogram in Prometheus

New Requests

- 1s latency

Bucket Increments

```
le: .1   =
le: .25  =
le: 1    = X
le: 5    = X
le: +Inf = X
```

Example Latency Histogram in Prometheus

New Requests

- 1s latency
- 5s latency

Bucket Increments

```
le: .1   =
le: .25  =
le: 1    = X
le: 5    = XX
le: +Inf = XX
```

Example Latency Histogram in Prometheus

New Requests

- 1s latency
- 5s latency
- .1s latency

Bucket Increments

```
le: .1   = X
le: .25  = X
le: 1    = XX
le: 5    = XXX
le: +Inf = XXX
```

Prometheus Counter Reminder

```
# HELP http_timer_count The total number of HTTP requests.
# TYPE http_timer_count counter
http_timer_count{app="api", code="200", method="GET"} 1
http_timer_count{app="api", code="200", method="POST"} 10
http_timer_count{app="api", code="200", method="PATCH"} 2
http_timer_count{app="frontend", code="200", method="GET"} 110
http_timer_count{app="frontend", code="404", method="GET"} 23
http_timer_count{app="frontend", code="500", method="GET"} 2
```

Prometheus Metric Crash-Course - histograms

```
# HELP http_timer_bucket A histogram of request duration
# TYPE http_timer_bucket histogram
http_timer_bucket{app="api", code="200", method="GET", l e="+Inf"} 1
http_timer_bucket{app="api", code="200", method="GET", l e="5"} 1
http_timer_bucket{app="api", code="200", method="POST",  le="+Inf"} 10
http_timer_bucket{app="api", code="200", method="POST",  le="5"} 7
http_timer_bucket{app="api", code="200", method="POST",  le="1"} 5
http_timer_bucket{app="api", code="200", method="POST",  le=".25"} 5
http_timer_bucket{app="api", code="200", method="POST",  le=".1"} 5
http_timer_bucket{app="api", code="200", method="PATCH", l e="+Inf"} 2
http_timer_bucket{app="frontend", code="200", method="GET",  le="+Inf"} 110
http_timer_bucket{app="frontend", code="200", method="GET",  le="5"} 101
http_timer_bucket{app="frontend", code="200", method="GET",  le="1"} 100
http_timer_bucket{app="frontend", code="200", method="GET",  le=".25"} 97
http_timer_bucket{app="frontend", code="200", method="GET",  le=".1"} 96
http_timer_bucket{app="frontend", code="404", method="GET" , le="+Inf"} 23
...
```

FYI there is some duplication

```
# sum all variants of the count
sum(http_timer_count{app="frontend"})                = 135

# sum all variants of the catch-all bucket
sum(http_timer_bucket{app="frontend", le="+Inf"}) = 135

# some tooling sets a dedicated "bucket_count" metric
http_timer_bucket_count{app="frontend"}              = 135
```

Which Metrics to use?

`grpc_timer_count & http_timer_count`

- total count of requests

`grpc_timer_bucket & http_timer_bucket`

- histogram of requests, bucketed by time

`grpc_timer_bucket_count & http_timer_bucket_count`

- total number of requests through histogram

Key Takeaways

- Histograms in Prometheus are **cumulative**
  - ALL requests will always get counted in "+Inf"
- Weave tooling: `` `http_timer_bucket` `` has everything
  - No need for any `` `*_count` `` metrics
  - Can dramatically cut total # of series in

grpc_timer_bucket{"+Inf"} == grpc_timer_count

Availability SLO Recording Rules … The Easy Way

Instantaneous Error Budget Rule For All Apps

```
(
   ( sum(http_timer_bucket{le="+Inf"}) by (app) * .1 )
   -
   (
     ( sum(http_timer_bucket{le="+Inf"}) by (app) )
     -
     ( sum(http_timer_bucket{le="+Inf",code!~"5.."}) by (app) )
   )
 )
/
( sum (http_timer_bucket{le="+Inf"}) by (app) * .1 )
```

28d Error Budget Rule For All Apps

```
(
  ( sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app) * .1 )
  -
  (
    ( sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app) )
    -
    ( sum(increase(http_timer_bucket{le="+Inf",code!~"5.."}[28d])) by (app) )
  )
)
/
( sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app) * .1 )
```

Lots of Duplication, Compute, Confusion

```
(
  ( sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app) * .1 )
  -
  (
    ( sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app) )
    -
    ( sum(increase(http_timer_bucket{le="+Inf",code!~"5.."}[28d])) by (app) )
  )
)
/
( sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app) * .1 )
```

⚠️ Important Prometheus Recording Behavior! ⚠️

- Prometheus supports Rules
  - alert rules - used to create alerts (duh)
  - record rules - run queries and save results
- Rules are provided in groups
- Rules are evaluated periodically
- Groups are processed in a random order
- **Rules within a group are always processed in order**

Example Availability Prometheus Rules

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

```
groups:
  - name: slo:example:budget_all_in_one
    rules:
    - record: slo:total
      expr: sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app)
    - record: slo:budgeted
      expr: slo:total * .1
    - record: slo:occured
      expr: |
        slo:total -
        sum(increase(http_timer_bucket{le="+Inf",code!~"5.."}[28d])) by (app)
    - record: slo:budget
      expr: (slo:budgeted - slo:occured) / slo:budgeted
```

Old rule for comparison

```
(
  ( sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app) * .1 )
  -
  (
    ( sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app) )
    -
    ( sum(increase(http_timer_bucket{le="+Inf",code!~"5.."}[28d])) by (app) )
  )
)
/
( sum(increase(http_timer_bucket{le="+Inf"}[28d])) by (app) * .1 )
```

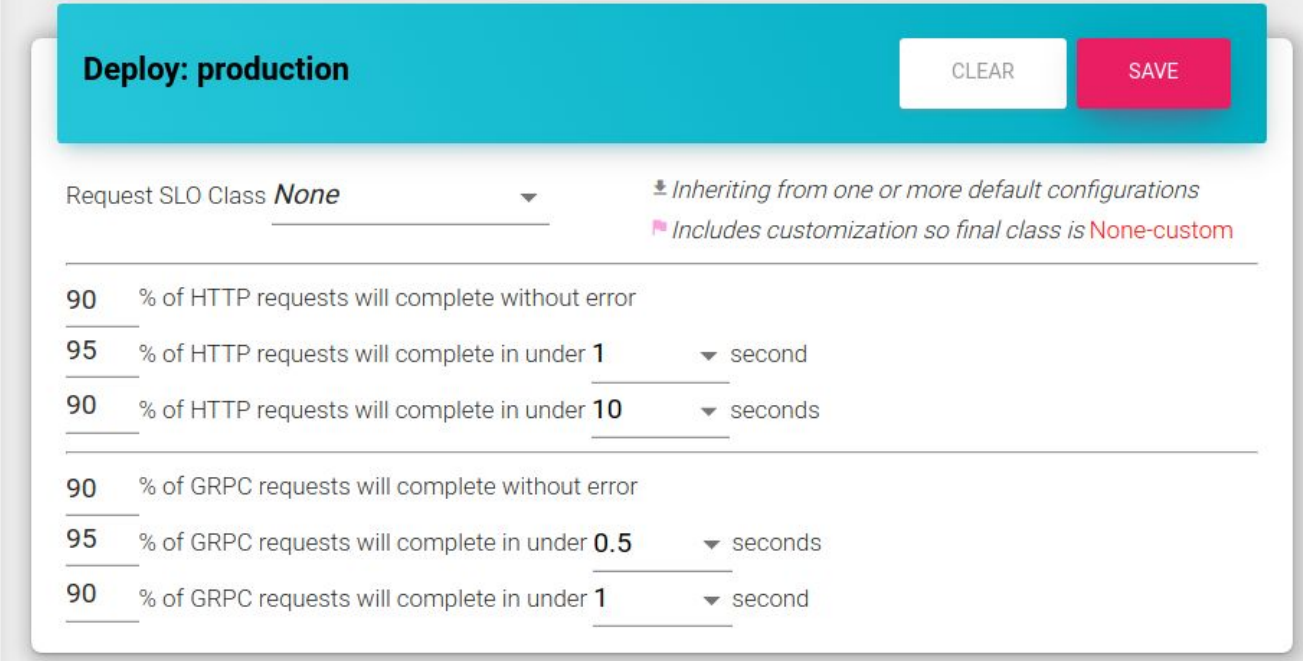
There are still problems!

Problems With All-In-One Rules

- All Apps have to have the same SLO objective
- Calculations for long SLO periods can be too large for Prometheus to calculate
  - "error executing query: query processing would load too many samples into memory in query execution"

Real, Auto-Generated Availability SLO Recording Rules

Remember This?




```
deploy:
  production:
    slo:
      requestSpec:
        grpc:
          availability:
            percent: "90"
          latency:
            tier1:
              percent: "90"
              seconds: "1"
            tier2:
              percent: "95"
              seconds: "0.5"
        http:
          availability:
            percent: "90"
          latency:
            tier1:
              percent: "90"
              seconds: "10"
            tier2:
              percent: "95"
              seconds: "1"
```

Calculate the Total # of requests for the budget

```
- record: 'slo:example_service:request:http:availability:budget_total
  expr: |
    sum(increase(http_timer_bucket{app="example-service",le="+Inf"}[28d]))
```

Calculate the total # of requests allowed to fail

```
- record: 'slo:example_service:request:http:availability:budget_allowed'
  expr: |
    (slo:example_service:request:http:availability:tier1:budget_total * .01)
```

Objective:
"99% Must Not Fail"

Calculate the total # of requests to actually fail

```
- record: 'slo:example_service:request:http:availability:budget_violations'
  expr: >
    (slo:example_service:request:http:availability:tier1:budget_total -
    sum(increase(
         http_timer_bucket{app="example-service",code!~"5..",le="+Inf"}[28d]
       ))
    )
```

Calculate Remaining Budget

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

```
# Calculate budget as: Budgeted - Actual / Budgeted
- record: 'slo:example_service:request:http:availability:error_budget'
  expr: >
    (
     slo:example_service:request:http:availability:budget_allowed -
     slo:example_service:request:http:availability:budget_violations
    ) /
    slo:example_service:request:http:availability:budget_allowed
```

Final Rules - Availability

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

```
# Calculate the Total # of requests within a timeframe that count towards the budget
- record: 'slo:example_service:request:http:availability:budget_total'
  expr: |
    sum(increase(http_timer_bucket{app="example-service",le="+Inf"}[28d]))

# Calculate the total # of requests within a timeframeBudgeted to fail
- record: 'slo:example_service:request:http:availability:budget_allowed'
  expr: |
    (slo:example_service:request:http:availability:budget_total * .01)

# Calculate the total # of requests within a timeframe thatActually failed
- record: 'slo:example_service:request:http:availability:budget_violations'
  expr: >
    (slo:example_service:request:http:availability:budget_total -
    sum(increase(http_timer_bucket{app="example-service",code!~"5..",le="+Inf"}[28d])))

# Calculate budget as: Budgeted - Actual / Budgeted
- record: 'slo:example_service:request:http:availability:error_budget'
  expr: >
    (slo:example_service:request:http:availability:budget_allowed -
    slo:example_service:request:http:availability:budget_violations) /
    slo:example_service:request:http:availability:budget_allowed
```

Real Latency SLO Recording Rules

Final Rules - Latency

$$\text{Budget} = \frac{\text{Budgeted} - \text{Actual}}{\text{Budgeted}}$$

```
# Calculate the Total # of requests within a timeframe that count toward
- record: 'slo:example_service:request:http:latency:tier1:budget_
  expr: |
    sum(increase(http_timer_bucket{app="example-service",le="+Inf"}[28d]))

# Calculate the total # of requests within a timeframe Budgeted to fail
- record: 'slo:example_service:request:http:latency:tier1:budget_allowed'
  expr: |
    (slo:example_service:request:http:latency:tier1:budget_total * .01)

# Calculate the total # of requests within a timeframe that Actually failed
- record: 'slo:example_service:request:http:latency:tier1:budget_violations'
  expr: >
    (slo:example_service:request:http:latency:tier1:budget_total -
    sum(increase(http_timer_bucket{app="example-service",le="1"}[28d])))

# Calculate budget as: Budgeted - Actual / Budgeted
- record: 'slo:example_service:request:http:latency:tier1:error_budget' # <- The actual budget
  expr: >
    (slo:example_service:request:http:latency:tier1:budget_allowed -
    slo:example_service:request:http:latency:tier1:budget_violations) /
    slo:example_service:request:http:latency:tier1:budget_allowed
```

Change rule names

Weave has two "tiers" of latency:

- tier 1 = stretch goal
- tier 2 = required

99% must complete...

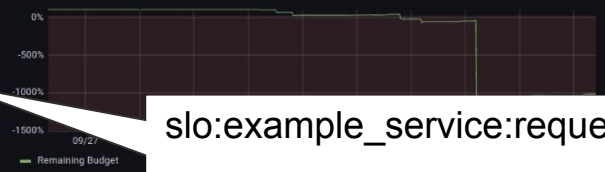
...in 1 second or less

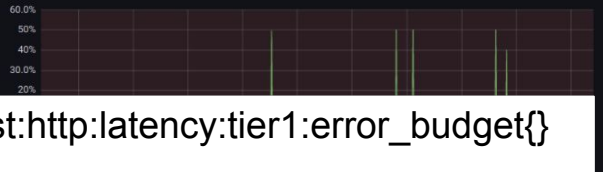
Dashboard Queries Get To Be Simple!

slo:example_service:request:http:availability:error_budget{}

99% of HTTP requests will complete without error

99% of HTTP requests will complete in under 2.5 second(s)





slo:example_service:request:http:latency:tier1:error_budget{}

slo:example_service:request:http:latency:tier2:error_budget{}

90% of HTTP requests will complete in under 5 second(s)

SLO Alerting

Naive SLO Alert

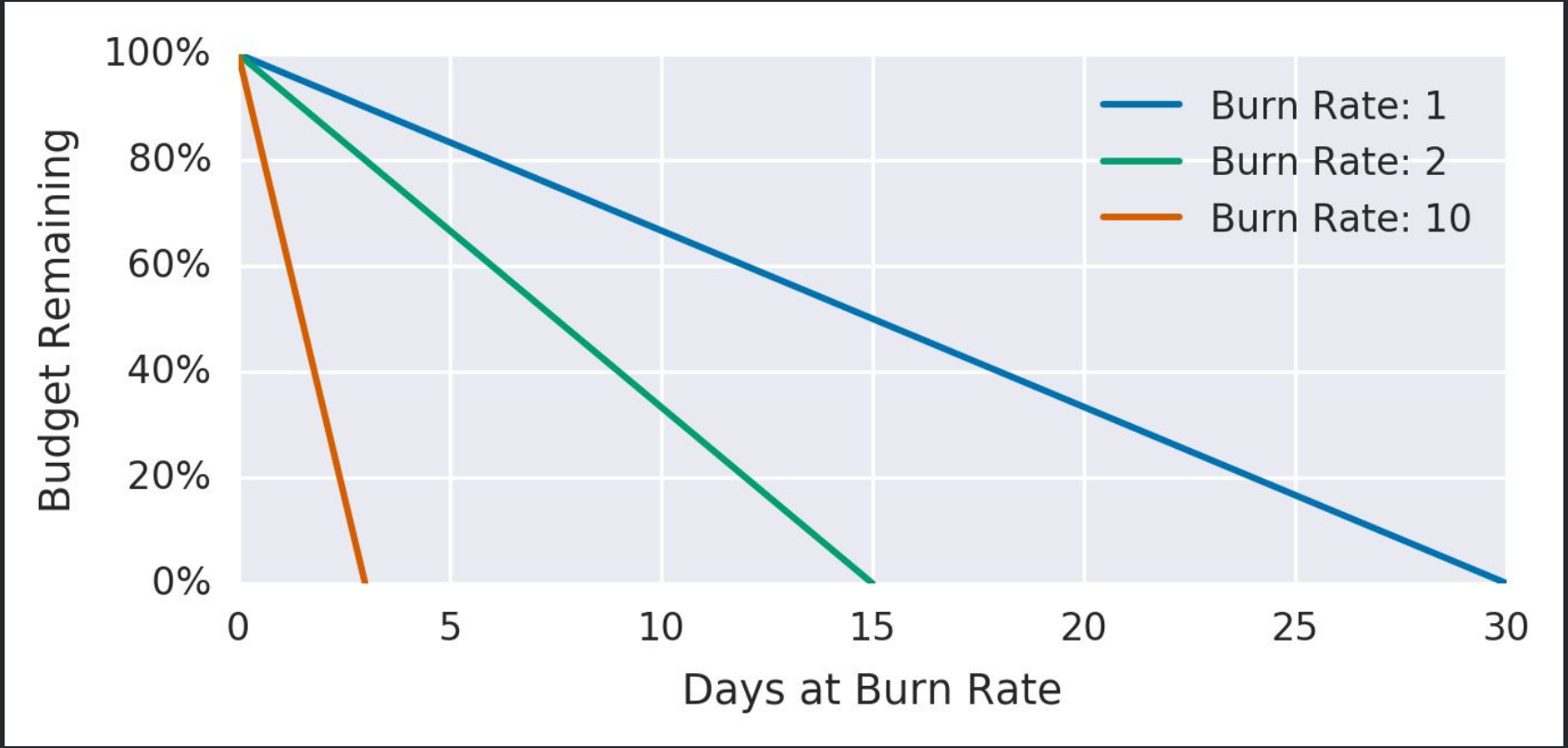
```
- alert: example_service:LowHTTPAvailabilityBudget
  expr: |
    slo:example_service:request:http:availability:error_budget < .5
```

Alert Periods

Goal: Detect Fast Drops of High %

Goal: Detect Slow Burn Over Time

Great Alerts Use "Burn Rate"



Calculate A Burn Rate

$$\frac{\text{error\_rate}}{\text{request\_rate}} = \text{burn\_rate}$$

Calculate A Burn Rate In Prometheus

$$\frac{\text{request\_rate} - \text{good\_rate}}{\text{request\_rate}} = \text{burn\_rate}$$

Availability Burn Rate Recording Rule 5m Example

```
- expr: |
    (
      sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[5m]))
      -

sum(rate(http_timer_bucket{app="example-app",code!~"5..",le="+Inf"}[5m]))
    )
    /
    sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[5m]))
  record: 'slo:example_app:request:http:availability:rate_5m'
```

Availability Burn Rate Recording Rules

```
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[5m])) -
    sum(rate(http_timer_bucket{app="example-app",code!~"5..",le="+Inf"}[5m])))
    / sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[5m]))
  record: 'slo:example_app:request:http:availability:rate_5m'
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[30m])) -
    sum(rate(http_timer_bucket{app="example-app",code!~"5..",le="+Inf"}[30m])))
    / sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[30m]))
  record: 'slo:example_app:request:http:availability:rate_30m'
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[1h])) -
    sum(rate(http_timer_bucket{app="example-app",code!~"5..",le="+Inf"}[1h])))
    / sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[1h]))
  record: 'slo:example_app:request:http:availability:rate_1h'
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[6h])) -
    sum(rate(http_timer_bucket{app="example-app",code!~"5..",le="+Inf"}[6h])))
    / sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[6h]))
  record: 'slo:example_app:request:http:availability:rate_6h'
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[3d])) -
    sum(rate(http_timer_bucket{app="example-app",code!~"5..",le="+Inf"}[3d])))
    / sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[3d]))
  record: 'slo:example_app:request:http:availability:rate_3d'
```

Latency Burn Rate Recording Rule 5m Example

```
- expr: |
    (
      sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[5m]))
      -
      sum(rate(http_timer_bucket{app="example-app",le="5"}[5m]))
    )
    /
    sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[5m]))
  record: 'slo:example_app:request:http:latency:tier1:rate_5m'
```

Remove `code` and add a specific `le`

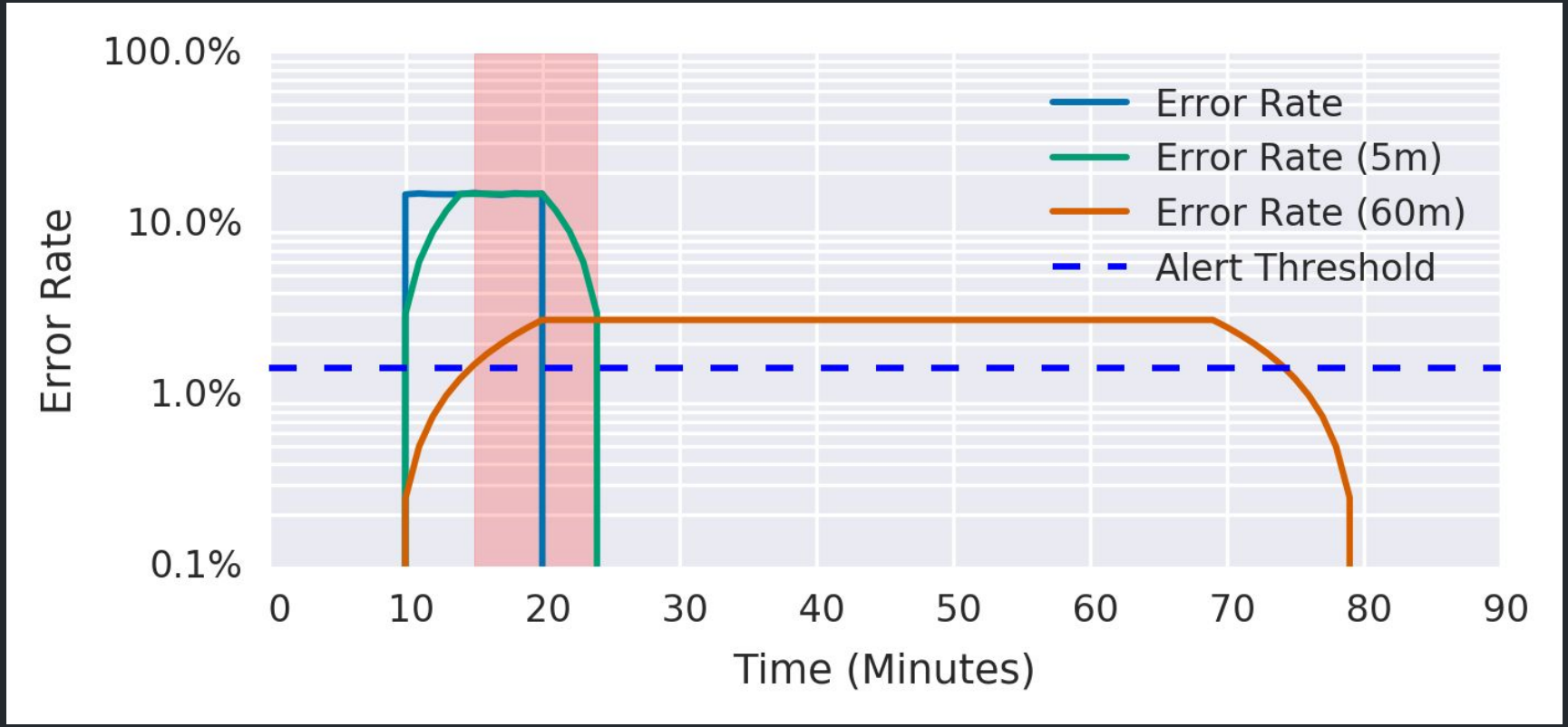
`le` **MUST** match a bucket reported in metrics

Latency Burn Rate Recording Rules

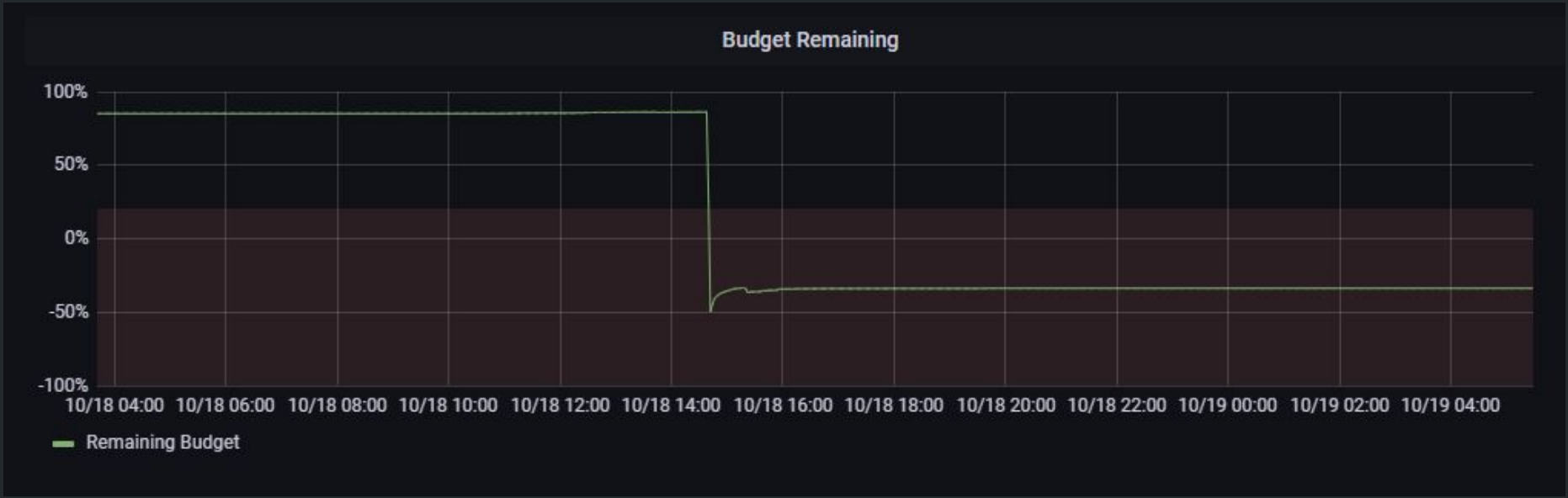
```
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[5m])) -
    sum(rate(http_timer_bucket{app="example-app",le="5"}[5m]))) /
    sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[5m]))
  record: 'slo:example_app:request:http:latency:tier1:rate_5m'
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[30m])) -
    sum(rate(http_timer_bucket{app="example-app",le="5"}[30m]))) /
    sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[30m]))
  record: 'slo:example_app:request:http:latency:tier1:rate_30m'
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[1h])) -
    sum(rate(http_timer_bucket{app="example-app",le="5"}[1h]))) /
    sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[1h]))
  record: 'slo:example_app:request:http:latency:tier1:rate_1h'
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[6h])) -
    sum(rate(http_timer_bucket{app="example-app",le="5"}[6h]))) /
    sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[6h]))
  record: 'slo:example_app:request:http:latency:tier1:rate_6h'
- expr: >
    (sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[3d])) -
    sum(rate(http_timer_bucket{app="example-app",le="5"}[3d]))) /
    sum(rate(http_timer_bucket{app="example-app",le="+Inf"}[3d]))
  record: 'slo:example_app:request:http:latency:tier1:rate_3d'
```

Now we can build smarter alerts

```
- alert: CriticalBudgetBurn
  labels:
    severity: page
  expr: |
    (
      (
        slo:example_app:request:http:latency:tier1:rate_1h > (13.44 * .01)
        and
        slo:example_app:request:http:latency:tier1:rate_5m > (13.44 * .01)
      )
      or
      (
        slo:example_app:request:http:latency:tier1:rate_6h > (5.6 * .01)
        and
        slo:example_app:request:http:latency:tier1:rate_30m > (5.6 * .01)
      )
    )
```


100.0%
10.0%
1.0%
0.1%
Error Rate
0
10
20
30
40
50
60
70
80
90
Time (Minutes)
Error Rate
Error Rate (5m)
Error Rate (60m)
Alert Threshold

Real World Example




Budget Remaining
100%
50%
0%
-50%
-100%
10/18 04:00
10/18 06:00
10/18 08:00
10/18 10:00
10/18 12:00
10/18 14:00
10/18 16:00
10/18 18:00
10/18 20:00
10/18 22:00
10/19 00:00
10/19 02:00
10/19 04:00
Remaining Budget
Burn Rates
100%
80%
60.0%
40%
20%
0%
10/18 04:00
10/18 06:00
10/18 08:00
10/18 10:00
10/18 12:00
10/18 14:00
10/18 16:00
10/18 18:00
10/18 20:00
10/18 22:00
10/19 00:00
10/19 02:00
10/19 04:00
1h rate
30m rate
3d rate
5m rate
6h rate


Burn Rates
100%
80%
60.0%
40%
20%
0%
10/18 04:00
10/18 06:00
10/18 08:00
10/18 10:00
10/18 12:00
10/18 14:00
10/18 16:00
10/18 18:00
10/18 20:00
10/18 22:00
10/19 00:00
10/19 02:00
10/19 04:00
1h rate
30m rate
3d rate
5m rate
6h rate
Burn Rates w/Alert Overlay
100%
10/18 14:00
10/18 16:00
10/18 18:00
10/18 20:00
10/18 22:00
10/19 00:00
10/19 02:00
10/19 04:00
alert-6h
alert-30m
alert-1h
alert-5m
alert-3d

100.0%
10.0%
1.0%
0.1%
Error Rate
Error Rate
Error Rate (5m)
Error Rate (60m)
Alert Threshold
0
10
20
30
40
50
60
70
80
90
Time (Minutes)

Alerting Breakdown

SLO Alerting Advice: Follow the SRE Book

- Use a 30d window
- Use the example rate windows (5m,30m,1h,6h,3d)
- Use the example alert rules

https://sre.google/workbook/alerting-on-slos/

Tone Shift!

Let's Talk K8s and Cloud Native

What about the Dashboards?

Solution: Grafana Operator

- The grafana-operator has support for a dashboard CRD!

Unfortunately, it doesn't work work for grafana cloud

- We used the operator-sdk to build a custom operator at Weave which could
  - Take a sample dashboard JSON
  - Do some string substitutions
  - Apply to Grafana Cloud

Request SLOs for __app__ - Class: __class__

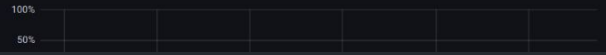
Links: WGraph

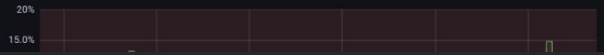
99.9% of gRPC requests will complete without error

99% of gRPC requests will complete in under 5 second(s)

90% of gRPC requests will complete in under 1 second(s)

99.9% of HTTP requests will complete without error

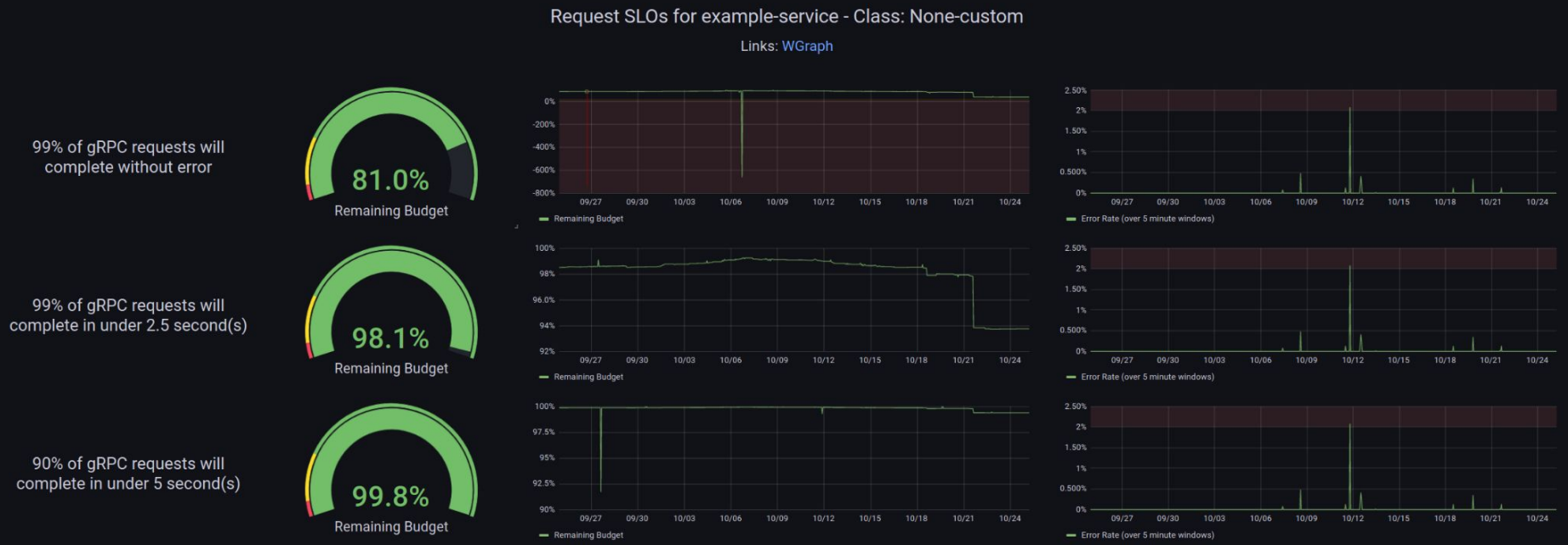




Example Dashboard YAML

```
apiVersion: custom.weave/v1beta1
kind: Dashboard
metadata:
  name: dashboard-sample
spec:
  grafana: local
  title: test-dash
  json: |
    {
        "title": "__app__ SLOs",
        "panels": [
         ...
        "timezone": "",
    }
 jsonReplacements:
    - find: __app__
      replace: example-app
```

Generated Dashboards



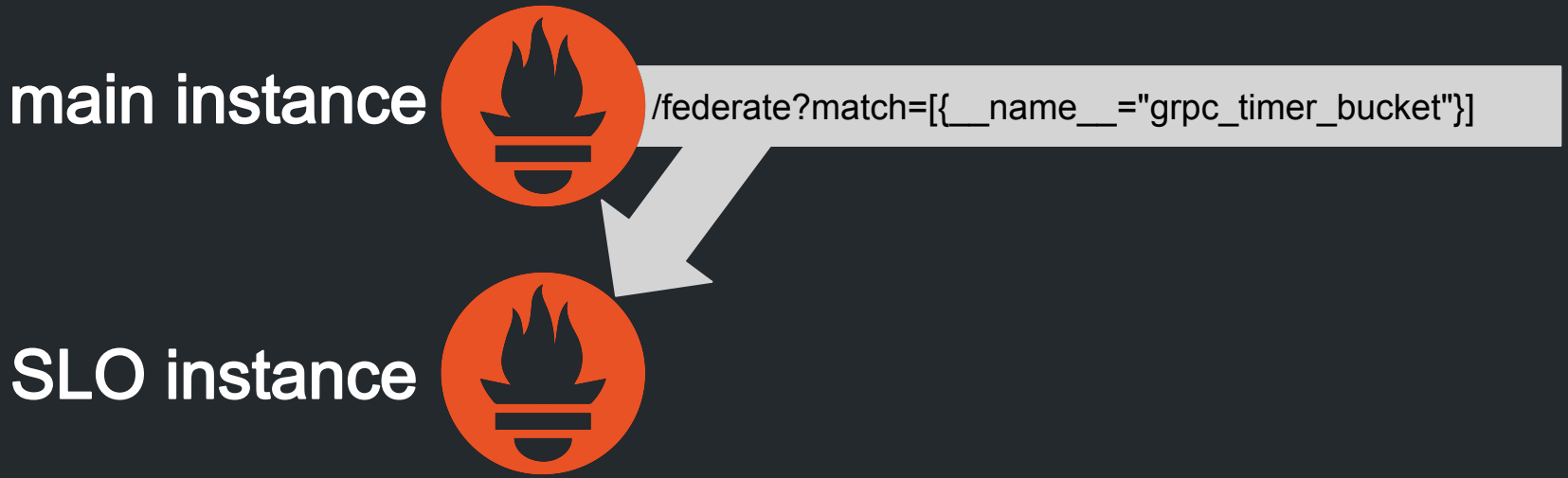
The Scale Issue

Prometheus Scaling Concerns

- The "main" Prometheus is huge and **busy**
- Our SLO is for 28 days but our main Prometheus is 14 days only
- SLO calculations can add load

Solution: Federation!

- Prometheus exposes a "federate" endpoint
  - Takes zero or more match expressions to filter returned results
- Essentially: Scrape Prometheus metrics from Prometheus



prometheus-operator to the rescue!

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus-slo
  labels:
    prometheus: prometheus-slo
spec:
  replicas: 1
  retention: 28d
  serviceAccountName: prometheus-slo
  serviceMonitorSelector:
    matchLabels:
      scraper: prometheus-slo
```

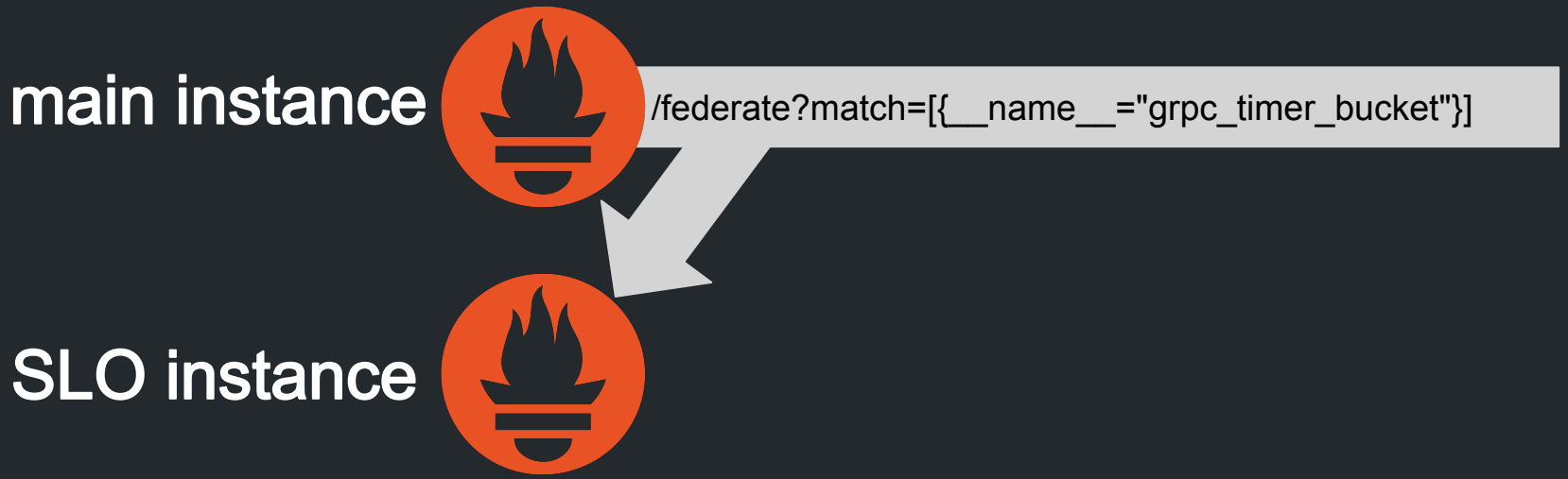
prometheus-operator to the rescue!

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: prometheus-slo-metrics
  labels:
    scraper: prometheus-slo
spec:
  jobLabel: prometheus-export-slo-rules
  namespaceSelector:
    matchNames:
      - prometheus
  selector:
    matchLabels:
      prometheus: main
  podMetricsEndpoints:
    - honorLabels: true
      interval: 30s
      params:
        'match[]':
          - '{__name__=~"grpc_timer_bucket"}'
          - '{__name__=~"http_timer_bucket"}'
      path: /federate
      targetPort: 9090
```

The Retention Issue

SLO Retention Concerns

- Management wants **2 years** of slo budget history
- Increasing even the slo Prometheus to 2y retention keeps **all** metrics forever



Add Another Layer!

main instance

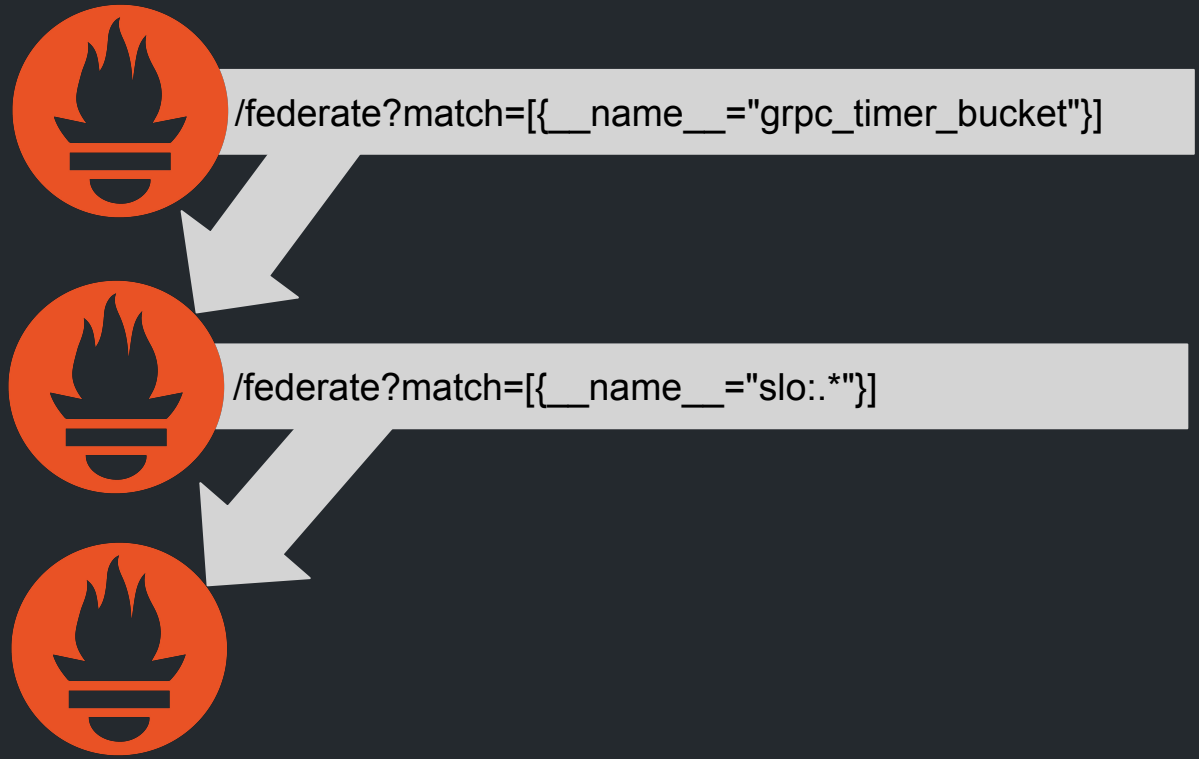
- 14d retention
- all metrics for cluster
- no calculations required

SLO instance

- 28d retention
- just slo metrics + calculations

export instance

- 2y
- just slo recording rules



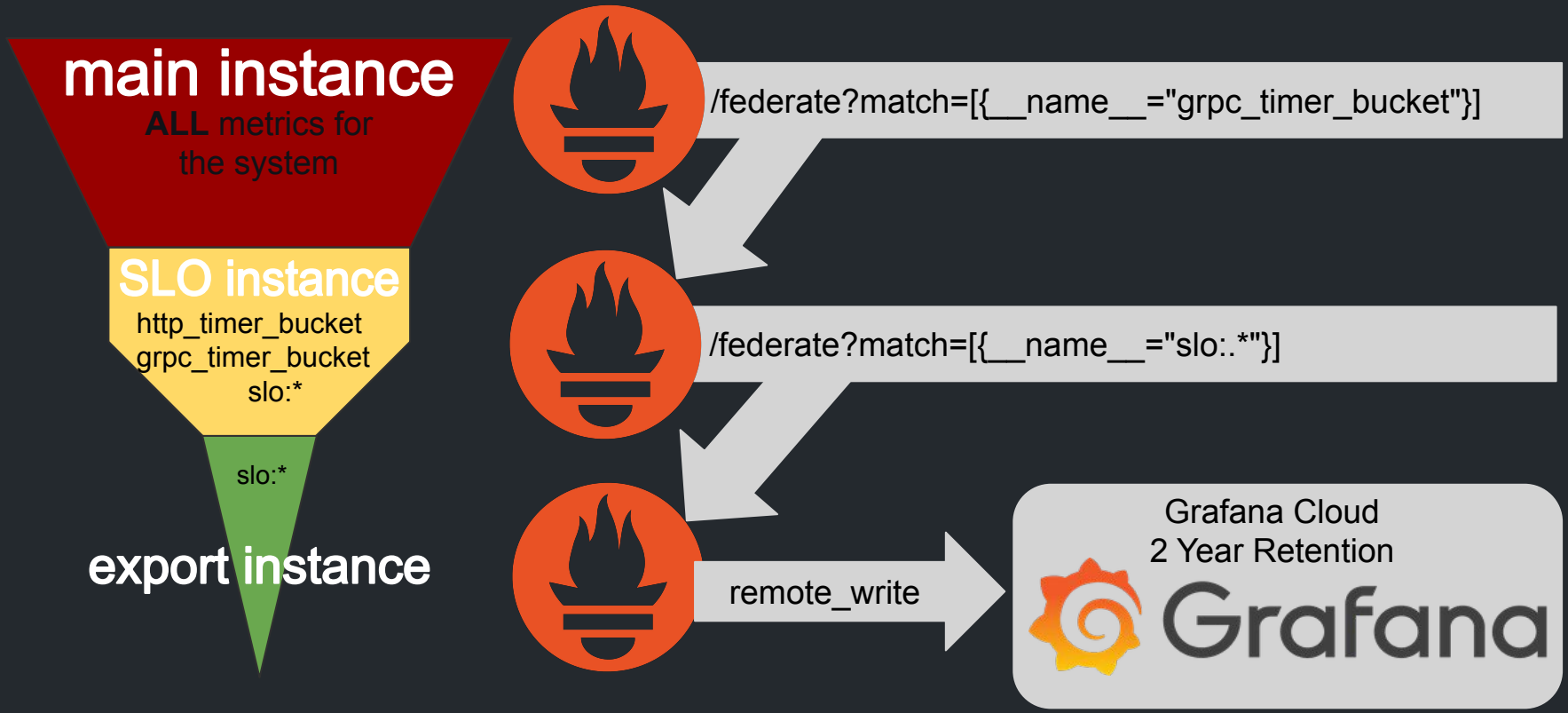
prometheus-operator to the rescue!

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus-export
  labels:
    prometheus: prometheus-export
spec:
  replicas: 1
  retention: 2y
  serviceAccountName: prometheus-export
  serviceMonitorSelector:
    matchLabels:
      scraper: prometheus-export
```

prometheus-operator to the rescue!

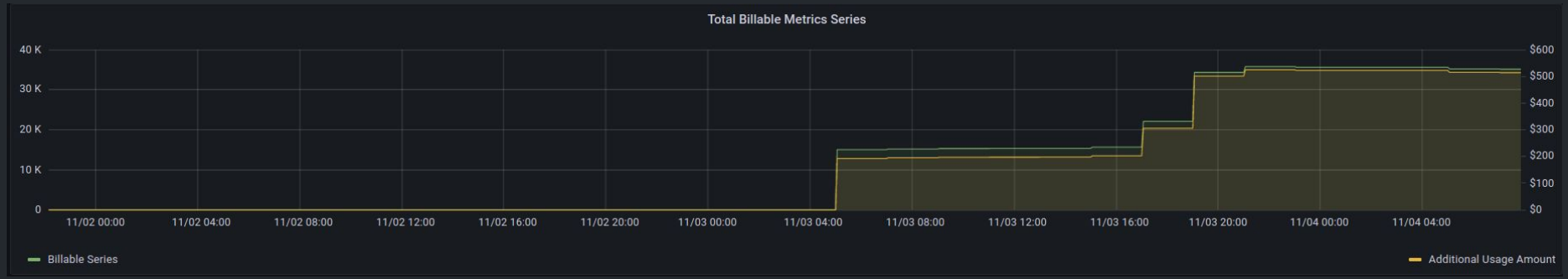
```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  labels:
    scraper: prometheus-export
  name: prometheus-export-slo-rules
spec:
  jobLabel: prometheus-export-slo-rules
  selector:
    matchLabels:
      prometheus: slo
  namespaceSelector:
    matchNames:
      - prometheus-slo
  podMetricsEndpoints:
    - honorLabels: true
      interval: 30s
      params:
        'match[]':
          - '{__name__=~"slo:.*"}'
      path: /federate
      targetPort: 9090
```

Prometheus Federation Series Count Funnel

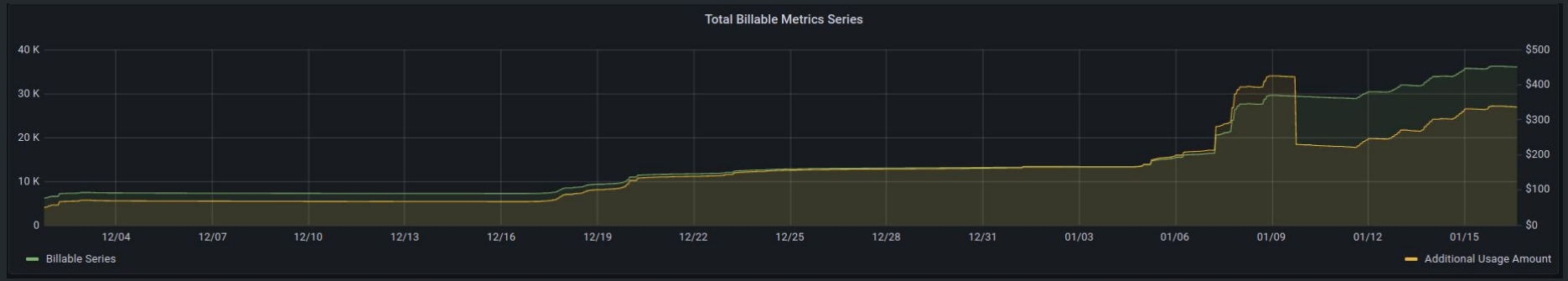


Grafana Cloud Billing Impact

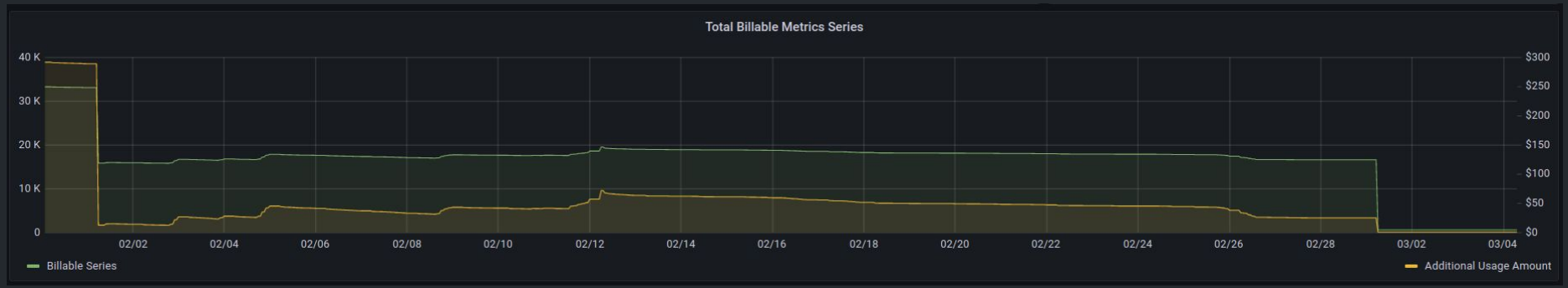
Ship ALL metrics



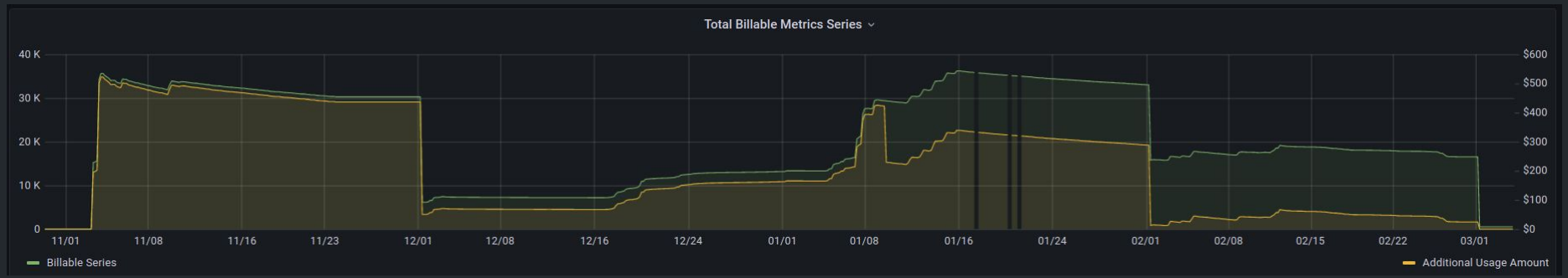
Ship Recording Rules



Ship only slo:* rules



Full Billing Timeline



Final Advice

- Standardize your metrics
- Pick 1 or 2 objectives to start with
- Use a 30d window if you can
- Automate & Generate!

Thank You!

SLOs In Practice

Carson Anderson

Weave

 @carsonoid

 @carson_ops