Prometheus

a practical workshop v1.1

Java Edition

Tomer Gabel

@ JCON Slovenia

May 2025

The Pillars of Observability

1. Logs
2. Metrics
3. Traces

The Pillars of Observability

1. ~~Logs~~
2. Metrics
3. ~~Traces~~

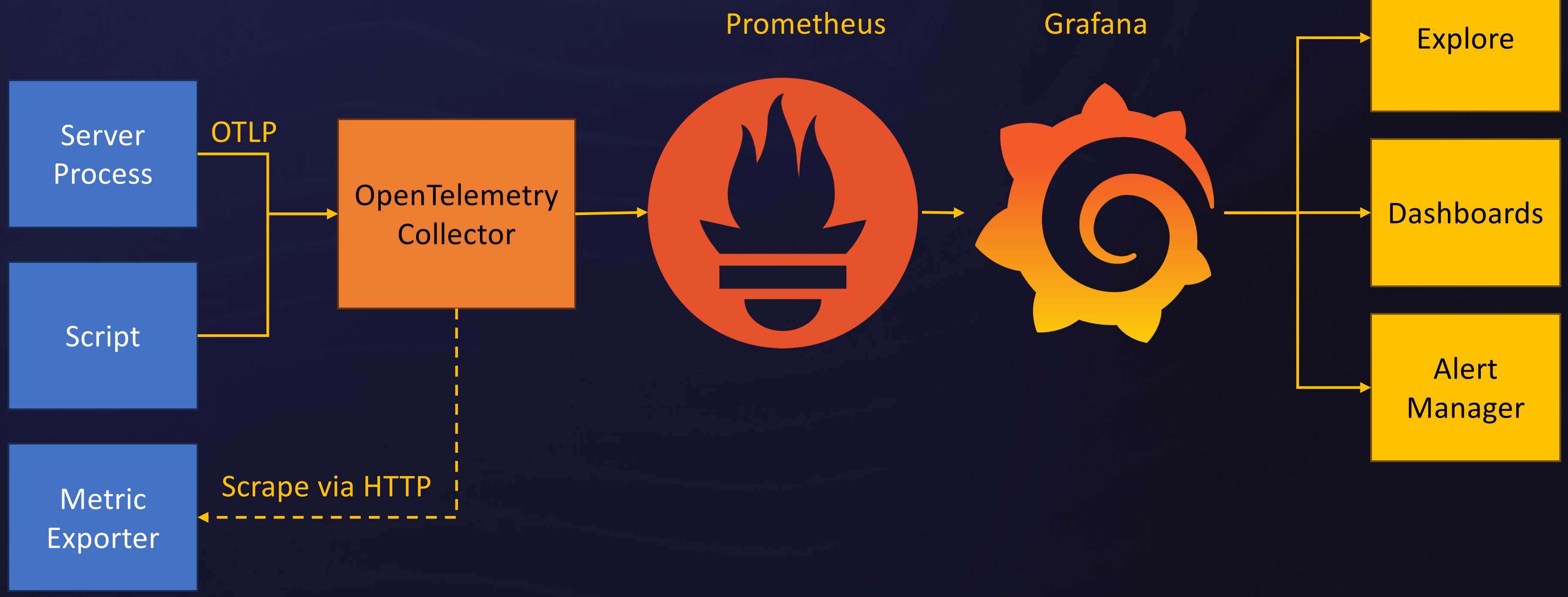
Our focus today.

Metrics:

Answer *quantitative* questions

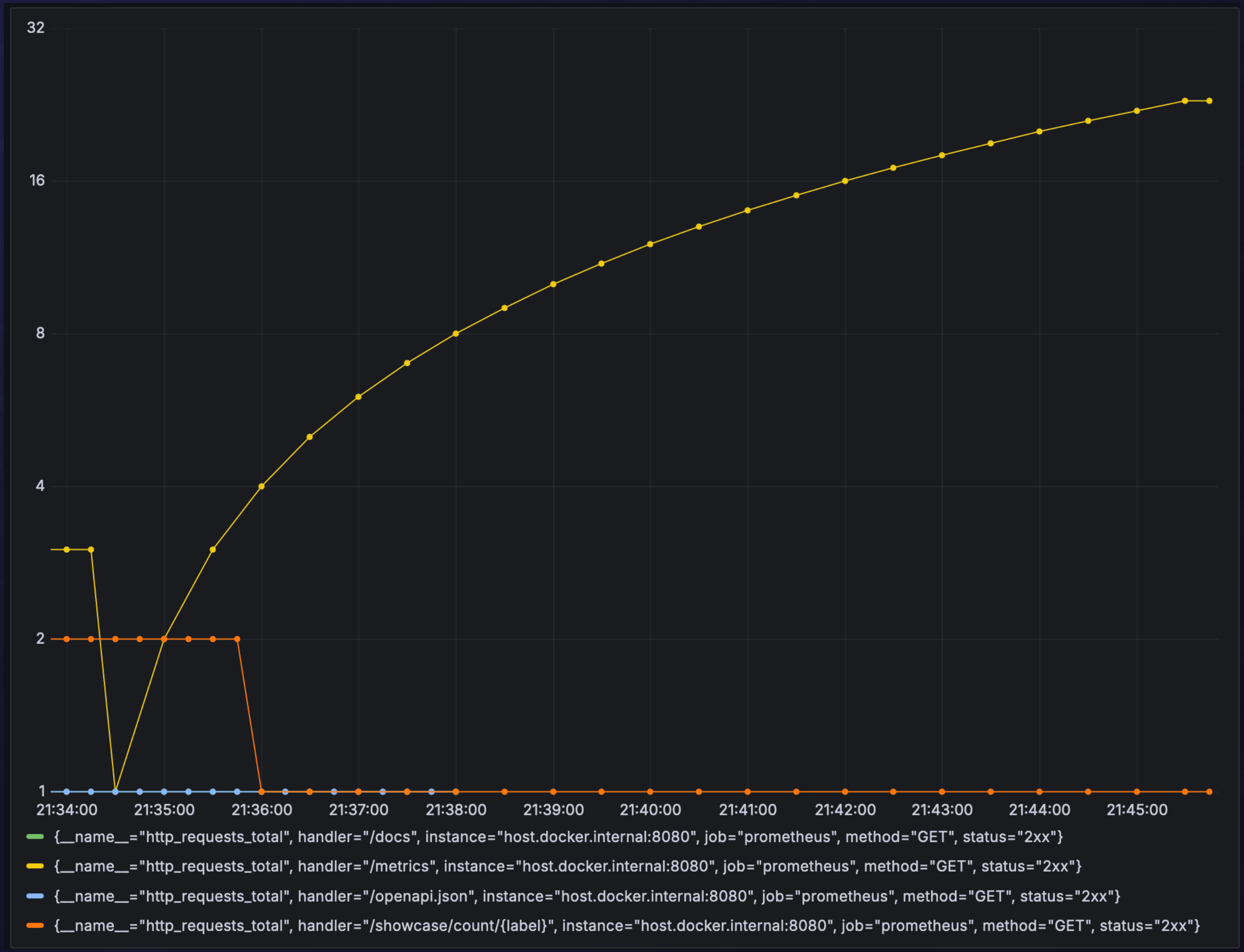
1. *How many* requests did I get?
2. *How long* did they take to process?
3. *How much* memory am I using?
4. *How much* free space is left on the disk?

Enter: Prometheus



Metrics 101

1. Name
2. Labels
3. Time
4. Value



🟩 {__name__="http_requests_total", handler="/docs", instance="host.docker.internal:8080", job="prometheus", method="GET", status="2xx"}

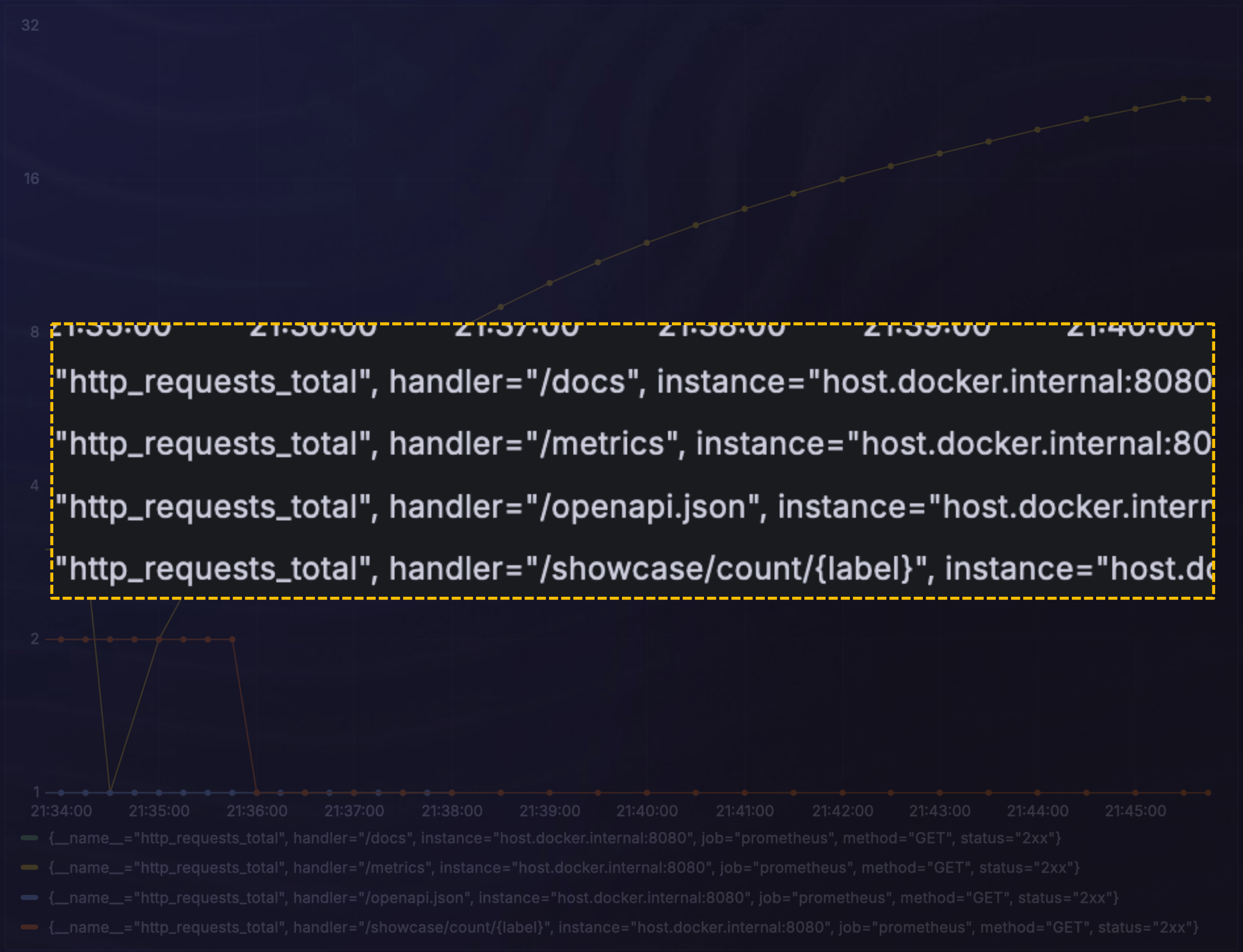
🟨 {__name__="http_requests_total", handler="/metrics", instance="host.docker.internal:8080", job="prometheus", method="GET", status="2xx"}

🟦 {__name__="http_requests_total", handler="/openapi.json", instance="host.docker.internal:8080", job="prometheus", method="GET", status="2xx"}

🟧 {__name__="http_requests_total", handler="/showcase/count/{label}", instance="host.docker.internal:8080", job="prometheus", method="GET", status="2xx"}

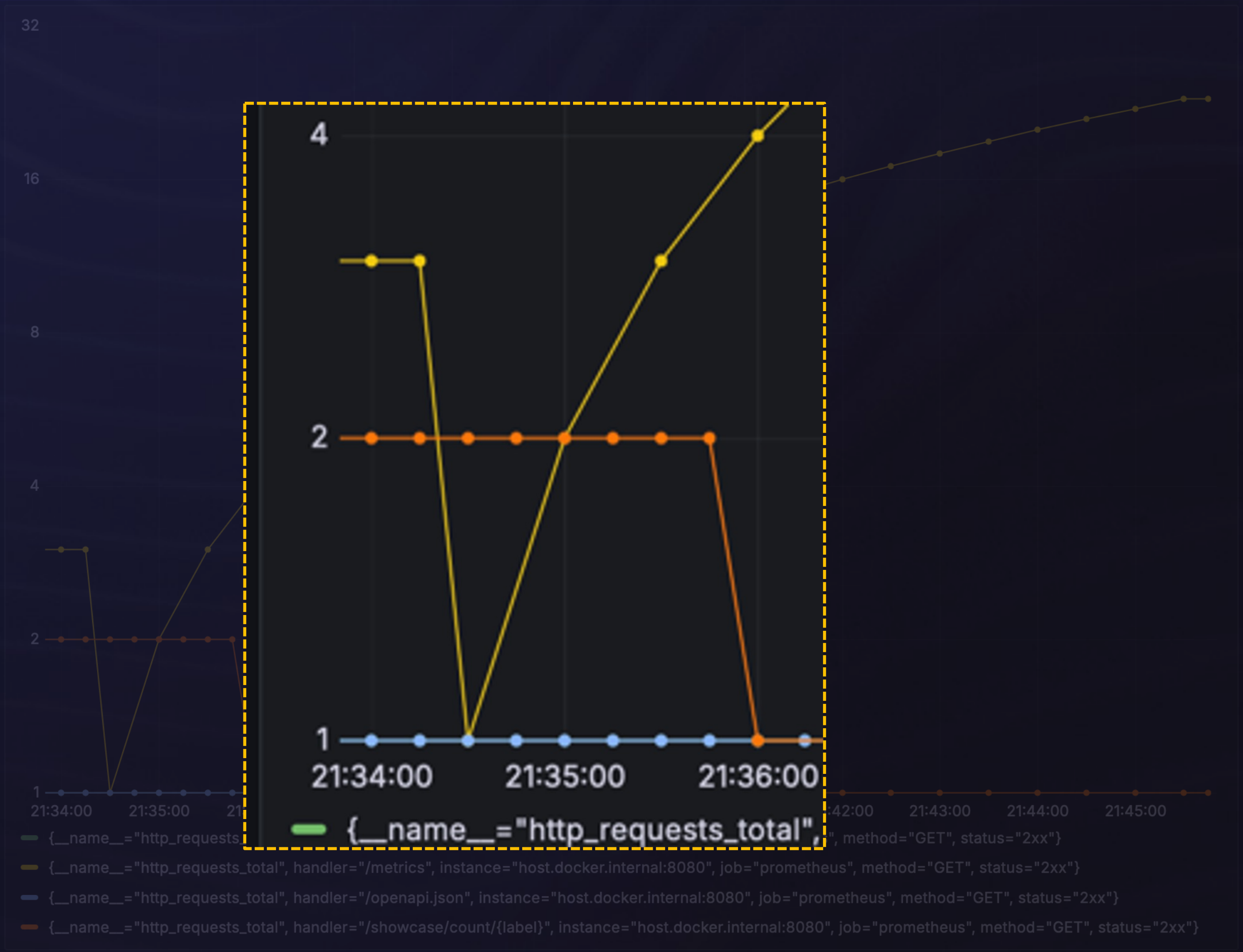
Metrics 101

1. Name
2. Labels
3. Time
4. Value



Metrics 101

1. Name
2. Labels
3. Time
4. Value



A word on cardinality

1. Prometheus is analogous to a *flat database*

   - Partitioned by *time*

   - Labels are like *indexed columns*

2. The DB size is therefore:

   - Timeframe * # metrics * # *label values*

Prometheus metric types: Counter

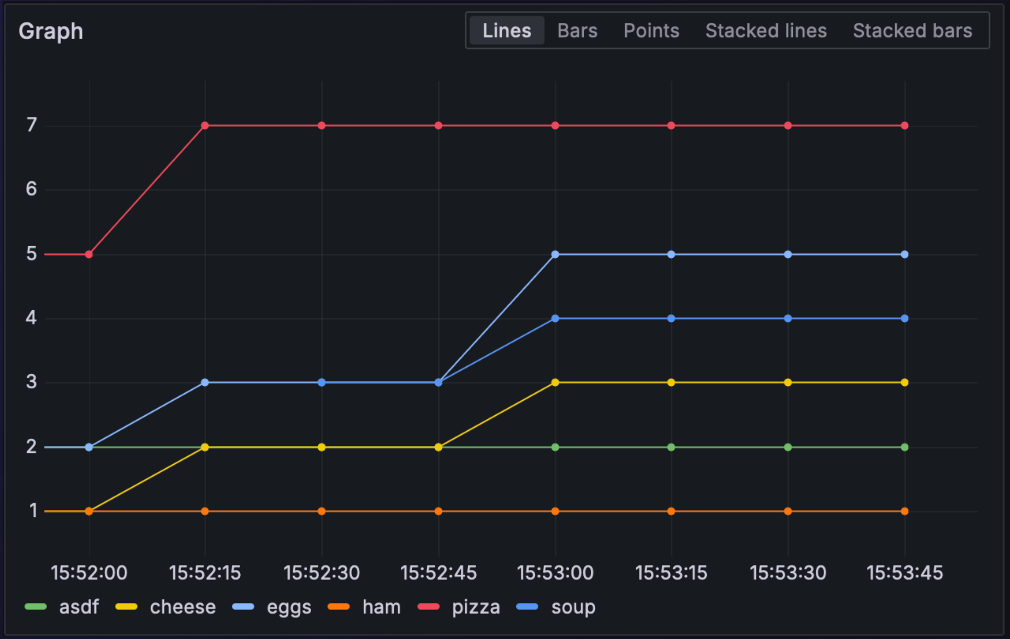
1. As the name implies, a metric that *counts*
2. Can go up or down by any amount
3. *Stateful* – value managed by *Prometheus*
4. Examples:
   - # of server requests (by method, path, …)
   - # of events ("times a user logged in")

Counting the Java way

```
OpenTelemetry otel = …;

var meter = otel.getMeter("controller");
var counter = meter
    .counterBuilder("my_count")
    .setDescription("Event count")
    .build();

counter.add(
  1,
  Attributes.of(stringKey("my_label"), "value")
);
```



Prometheus metric types: Gauge

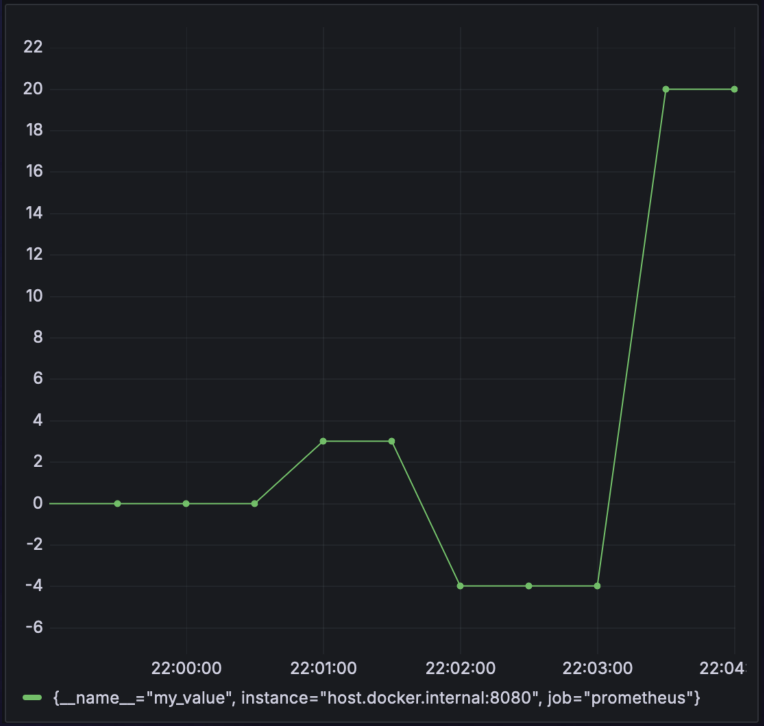
1. A metric that is sampled *on demand*
2. Each sample is *independent*
3. Common examples:
   - CPU core temperature in ℃
   - Free space on /dev/sda2

Gauging the Java way

```
OpenTelemetry otel = …;

var meter = otel.getMeter("controller");
var gauge = meter
    .gaugeBuilder("my_value")
    .setDescription("Some stateful value")
    .ofLongs()
    .build();

gauge.set(15);
```



🟩 {__name__="my_value", instance="host.docker.internal:8080", job="prometheus"}

Prometheus metric types: Histograms

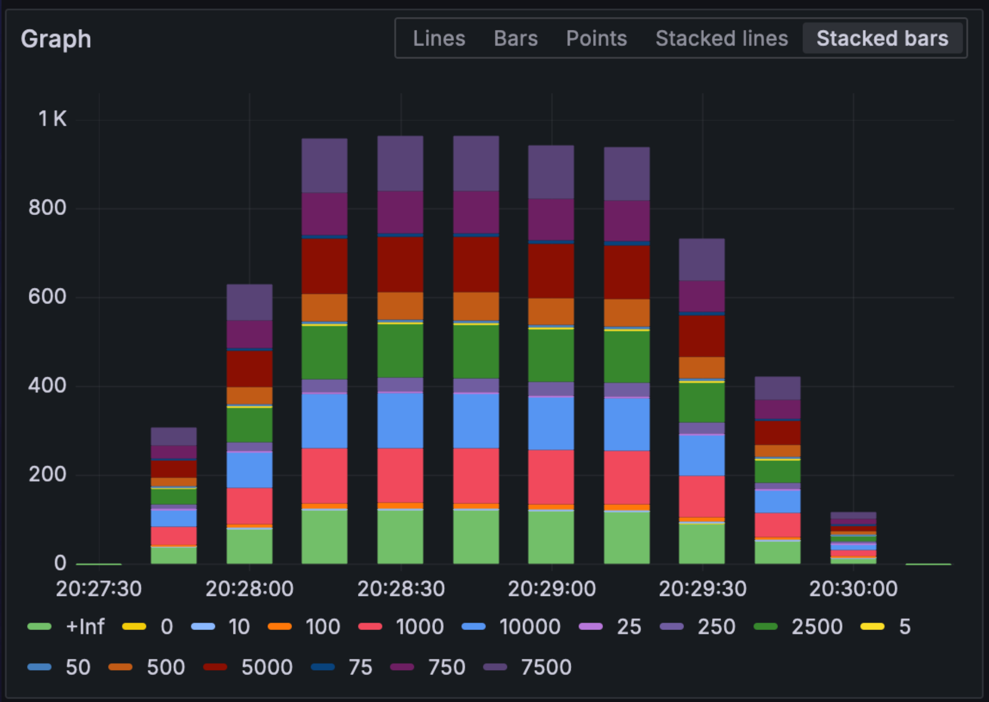
1. Some metrics *cannot be represented* with one value
   - "*How long* do requests to /login take?"
   - Do you mean average? Mean? 90th percentile?
2. These values prdoduce a *distribution*
3. Managed with buckets. Latency, for example:
   - 0-10ms, 10-100ms, 100-1000ms, ...

Histograms the Java way

```
OpenTelemetry otel = …;

var meter = otel.getMeter("controller");
var histogram = meter
    .histogramBuilder("my_duration")
    .setDescription("Event duration")
    .setUnit("ms")
    .ofLongs()
    .build();

histogram.record(3400);
```



Not all histograms are made equal

Latencies vary wildly (ms vs second), as do *sizes* (payload in KB, disk in GB).

Views to the rescue!

```
var buckets = List.of(0.0, 500.0, 1000.0, 10000.0);
var meterProvider = SdkMeterProvider.builder()
        .registerView(
            InstrumentSelector.builder().setName("my_duration").build(),
            View.builder()
                .setAggregation(Aggregation.explicitBucketHistogram(buckets))
                .build()
        ).build();
```

Lab Time

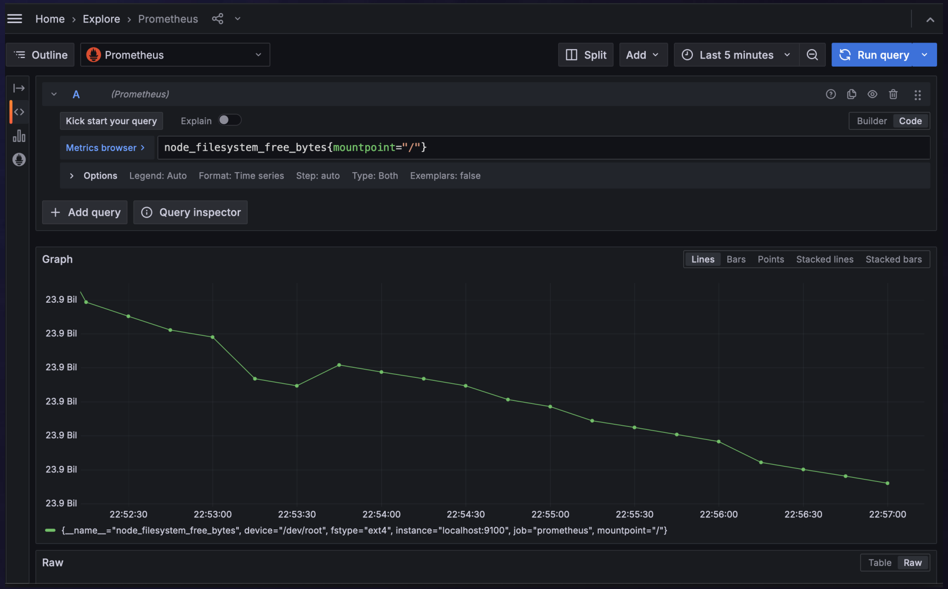
Showcase

1. *Familiarize yourself* with the lab setup
2. *Get handsy* with Grafana
3. Open lab-showcase from the *class materials*
4. You have *30 minutes* to explore!

Part 2: PromQL

PromQL: Fear, Terror and Ruthless Efficiency

1. Extensive, powerful
2. … not entirely trivial



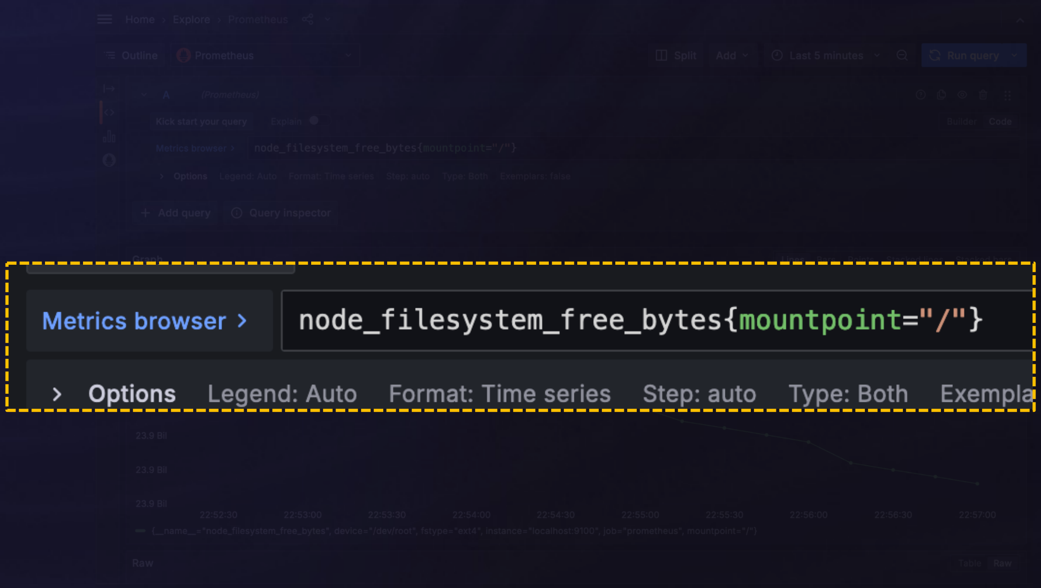
```
node_filesystem_free_bytes{mountpoint="/"}
```

Options Legend: Auto Format: Time series Step: auto Type: Both Exemplars: false

{__name__="node_filesystem_free_bytes", device="/dev/root", fstype="ext4", instance="localhost:9100", job="prometheus", mountpoint="/"}

PromQL: Fear, Terror and Ruthless Efficiency

1. Extensive, powerful
2. … not entirely trivial
3. Basically:
   - A *query*
   - A *timeframe*



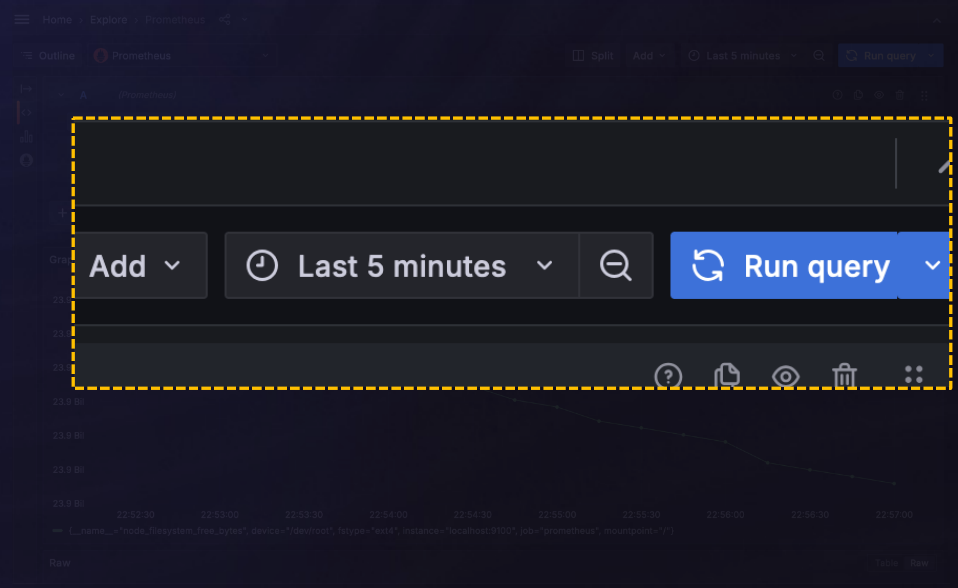
```
node_filesystem_free_bytes{mountpoint="/"}
```

› Options   Legend: Auto   Format: Time series   Step: auto   Type: Both   Exempla

PromQL: Fear, Terror and Ruthless Efficiency

1. Extensive, powerful
2. … not entirely trivial
3. Basically:

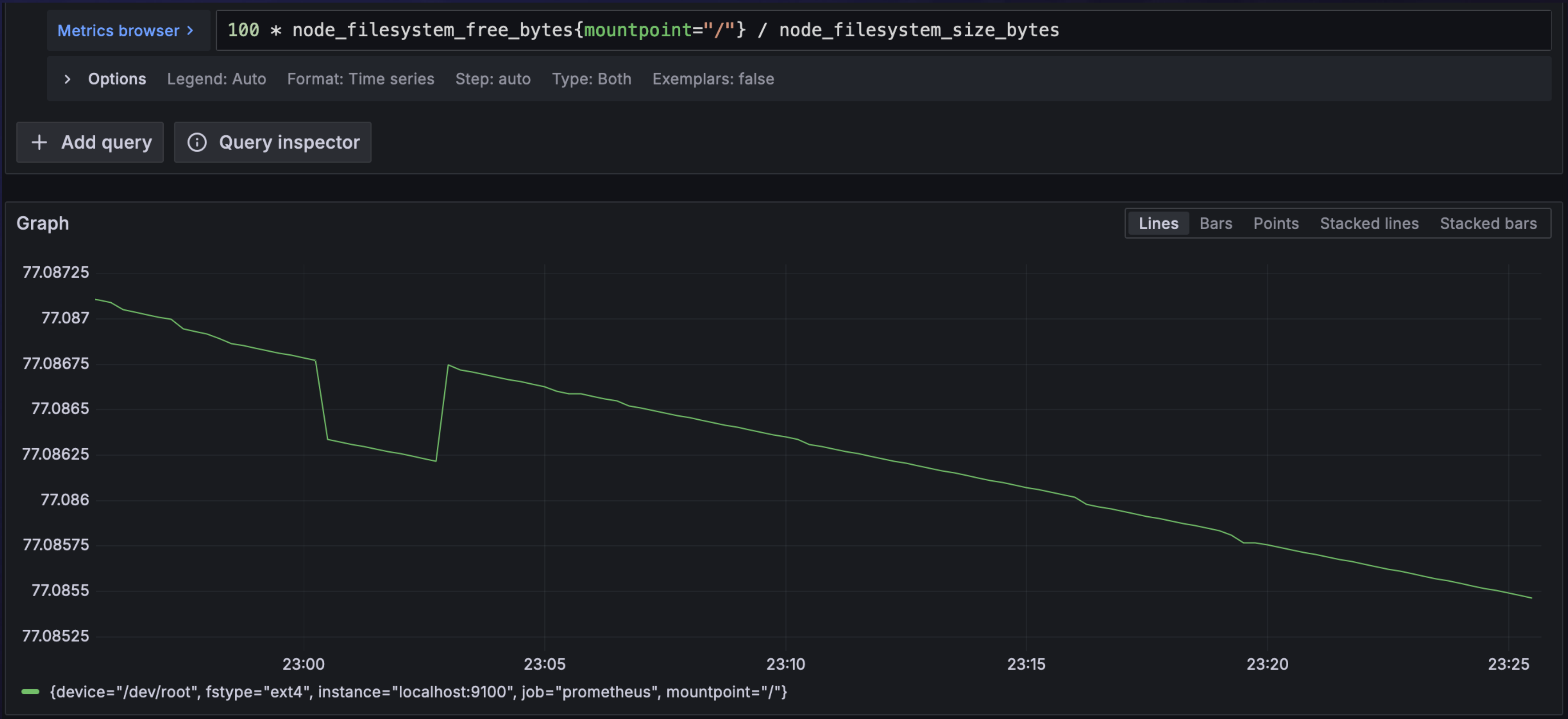
   - A *query*
   - A *timeframe*



PromQL: Maths!

1. [Metric, labels, time] yields one value
2. Some observations require *multiple* values
3. Expressed with basic arithmetic:
   - % free disk space = *free* disk bytes/*total* disk bytes
   - % CPU = *CPU used seconds* / (# seconds * # cores)

PromQL: Maths



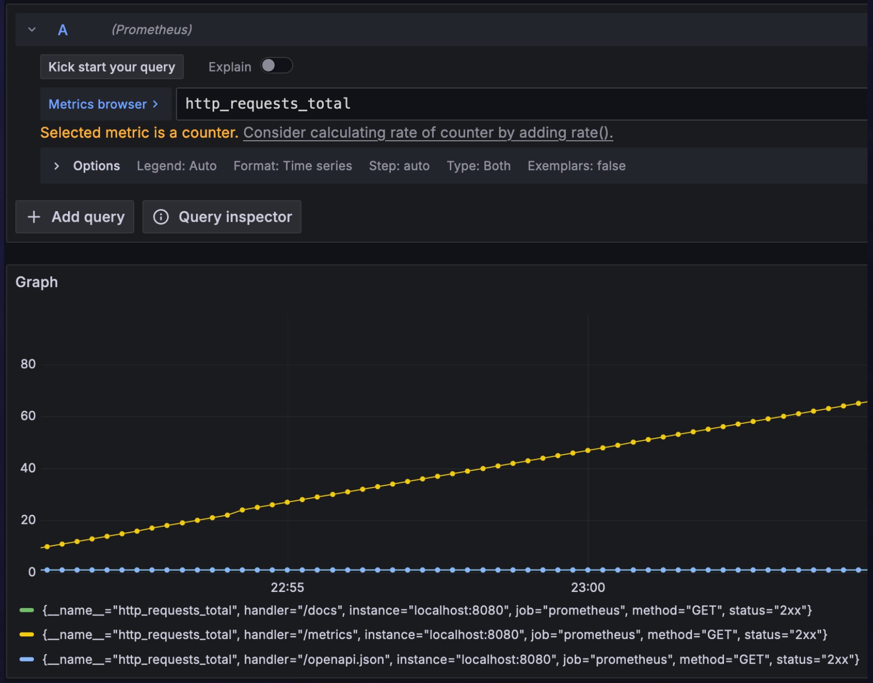
```
100 * node_filesystem_free_bytes{mountpoint="/"} / node_filesystem_size_bytes
```

Options Legend: Auto Format: Time series Step: auto Type: Both Exemplars: false

🟩 {device="/dev/root", fstype="ext4", instance="localhost:9100", job="prometheus", mountpoint="/"}

PromQL: Rates

1. Counters are very useful
2. Can’t use them *naïvely*



```
http_requests_total
```

Selected metric is a counter. Consider calculating rate of counter by adding rate().

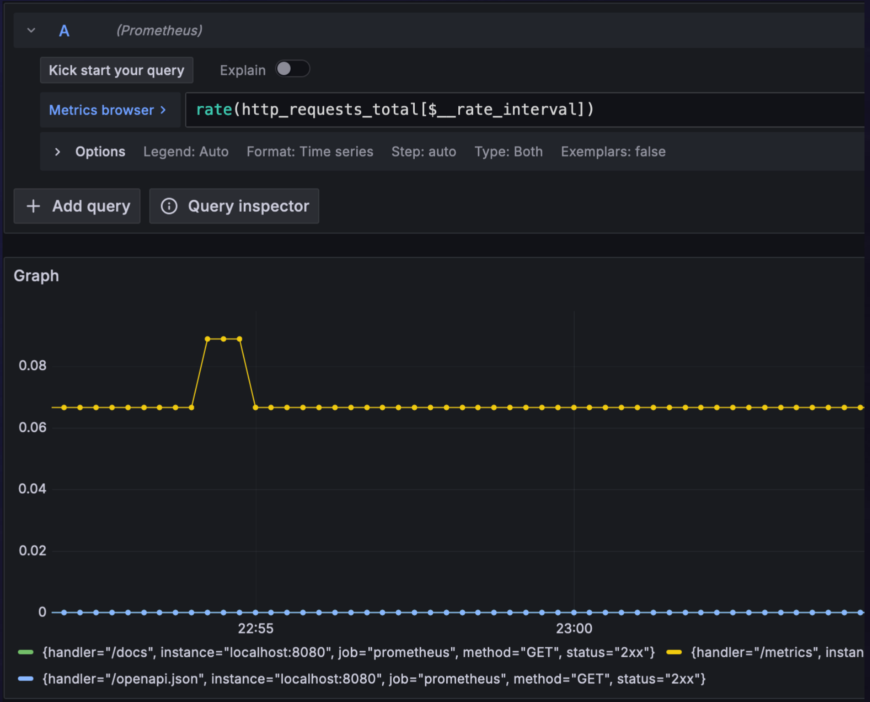
› Options Legend: Auto Format: Time series Step: auto Type: Both Exemplars: false

🟩 {__name__="http_requests_total", handler="/docs", instance="localhost:8080", job="prometheus", method="GET", status="2xx"}
🟨 {__name__="http_requests_total", handler="/metrics", instance="localhost:8080", job="prometheus", method="GET", status="2xx"}
🟦 {__name__="http_requests_total", handler="/openapi.json", instance="localhost:8080", job="prometheus", method="GET", status="2xx"}

PromQL: Rates

1. Counters are very useful
2. Can't use them *naïvely*
3. We need a *rate derivation*
4. Magic incantation for the win:

```
rate(metric{…}[$__rate_interval])
```



A (Prometheus)

Kick start your query · Explain

Metrics browser >

```
rate(http_requests_total[$__rate_interval])
```

Options · Legend: Auto · Format: Time series · Step: auto · Type: Both · Exemplars: false

Add query · Query inspector

Graph

🟩 {handler="/docs", instance="localhost:8080", job="prometheus", method="GET", status="2xx"} 🟨 {handler="/metrics", instan
🟦 {handler="/openapi.json", instance="localhost:8080", job="prometheus", method="GET", status="2xx"}

PromQL: Aggregations

1. Consider *http_server_duration_milliseconds_count*
   - A simple counter
   - Potentially many services, paths
2. Suppose you want the grand total?

```
sum(http_server_duration_milliseconds_count)
```

PromQL: Aggregations

1. Consider *http_server_duration_milliseconds_count*
   - A simple counter
   - Potentially many services, paths
2. How about the 5 most used paths?

```
topk(5, http_server_duration_milliseconds_count)
```

PromQL: Aggregations

1. What are the top 3 most time-consuming endpoints?

   ```
   topk(3, http_server_duration_milliseconds_sum)
   ```

2. How many processes run which Python version?

   ```
   count by (major) (python_info)
   ```

3. What is the highest rate of pagefaults recorded?

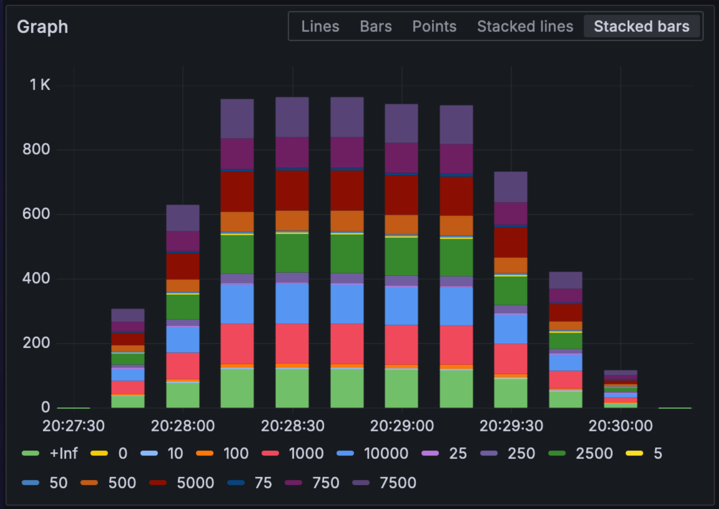
   ```
   max(rate(node_vmstat_pgfault[$__rate_interval]))
   ```

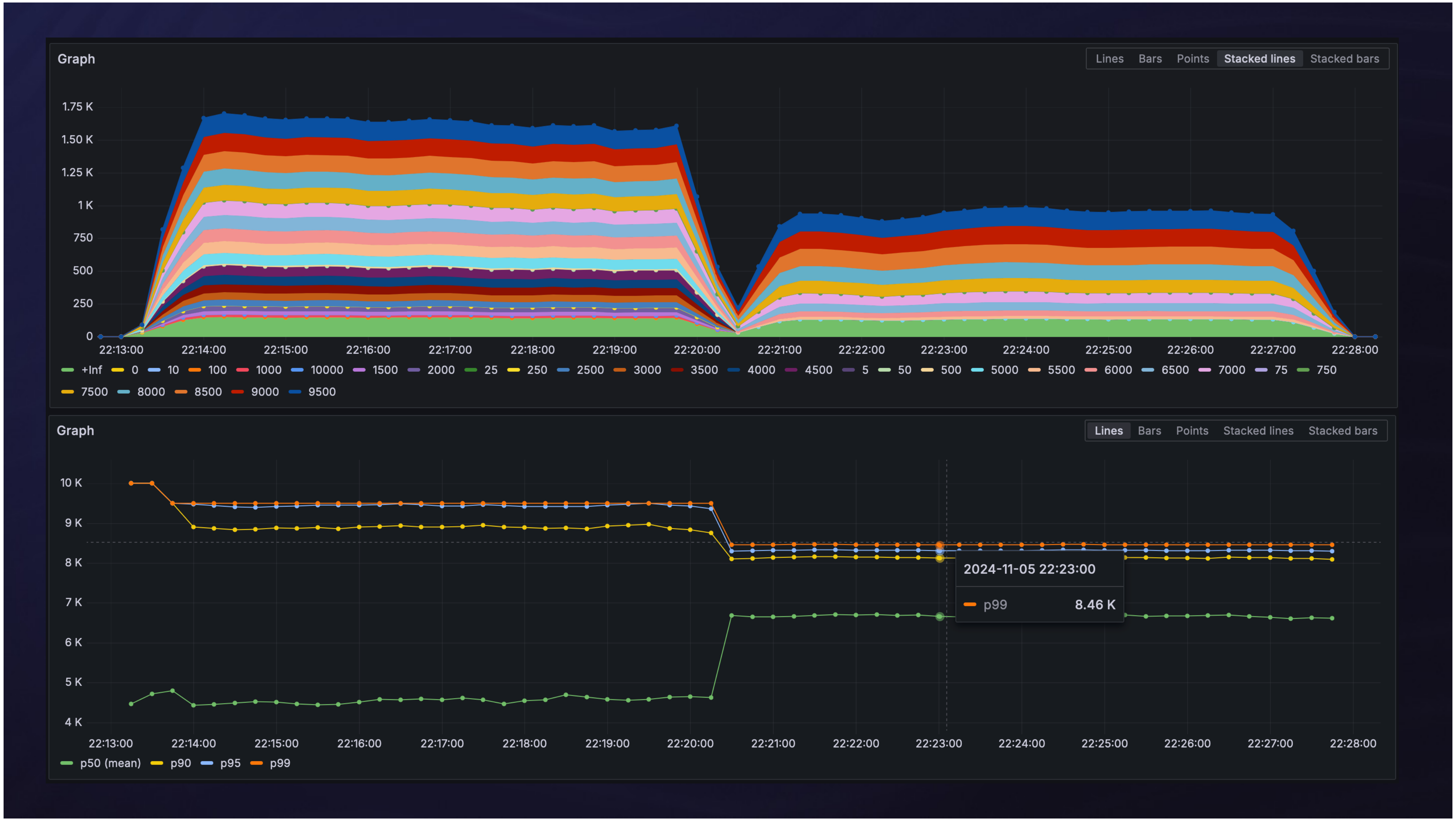
And now, what you’ve all been waiting for

Quantiles (a.k.a. percentiles)

1. A histogram is a set of *counters*
   - 0-100ms, 100-200ms, 200-300ms...
2. What is the 95th *percentile*?
   - Linear regression FTW... but maths!
3. PromQL to the rescue:

`histogram_quantile(0.99, http_server_duration_milliseconds_bucket)`




Graph
Lines
Bars
Points
Stacked lines
Stacked bars
1.75 K
1.50 K
1.25 K
1 K
750
500
250
0
22:13:00
22:14:00
22:15:00
22:16:00
22:17:00
22:18:00
22:19:00
22:20:00
22:21:00
22:22:00
22:23:00
22:24:00
22:25:00
22:26:00
22:27:00
22:28:00
+Inf
0
10
100
1000
10000
1500
2000
25
250
2500
3000
3500
4000
4500
5
50
500
5000
5500
6000
6500
7000
75
750
7500
8000
8500
9000
9500
Graph
Lines
Bars
Points
Stacked lines
Stacked bars
10 K
9 K
8 K
7 K
6 K
5 K
4 K
22:13:00
22:14:00
22:15:00
22:16:00
22:17:00
22:18:00
22:19:00
22:20:00
22:21:00
22:22:00
22:23:00
22:24:00
22:25:00
22:26:00
22:27:00
22:28:00
2024-11-05 22:23:00
p99 8.46 K
p50 (mean)
p90
p95
p99

Key Takeaways

1. Instrument early, instrument often

   There’s no such thing as “too much data”

2. Beware high *cardinality*

   Aggregate, don’t transact (that’s what logs/traces are for)

3. Learn ye *PromQL*

   That’s where the real leverage is

Lab time, again!

tomer@substrate.co.il

@tomerg

https://github.com/holograph/prometheus-workshop-service-python

Thank you for your attention

Questions?