

# Distributed Tracing with Tempo and TraceQL



**Suraj Sidh**  
Senior Software Engineer, Tempo



Explore Tempo ▼

A (Tempo)

Query type **Search - Beta** TraceID JSON file

Service Name	Select a service
Span Name	Select a span
Tags	http.status_code=200 error=true
Min Duration	e.g. 1.2s, 100ms, 500us
Max Duration	e.g. 1.2s, 100ms, 500us
Limit	

+ Add query Query history Inspector

Yardba + Inspector



Suraj Sidh

Senior Software Engineer  
@electron0zero

## Agenda

- **What is Distributed Tracing**
- **Distributed Tracing with Tempo**
- **Grafana Tempo & TraceQL**
- **Storage layouts & Parquet**
- **TraceQL Demo**
- **Q&A**



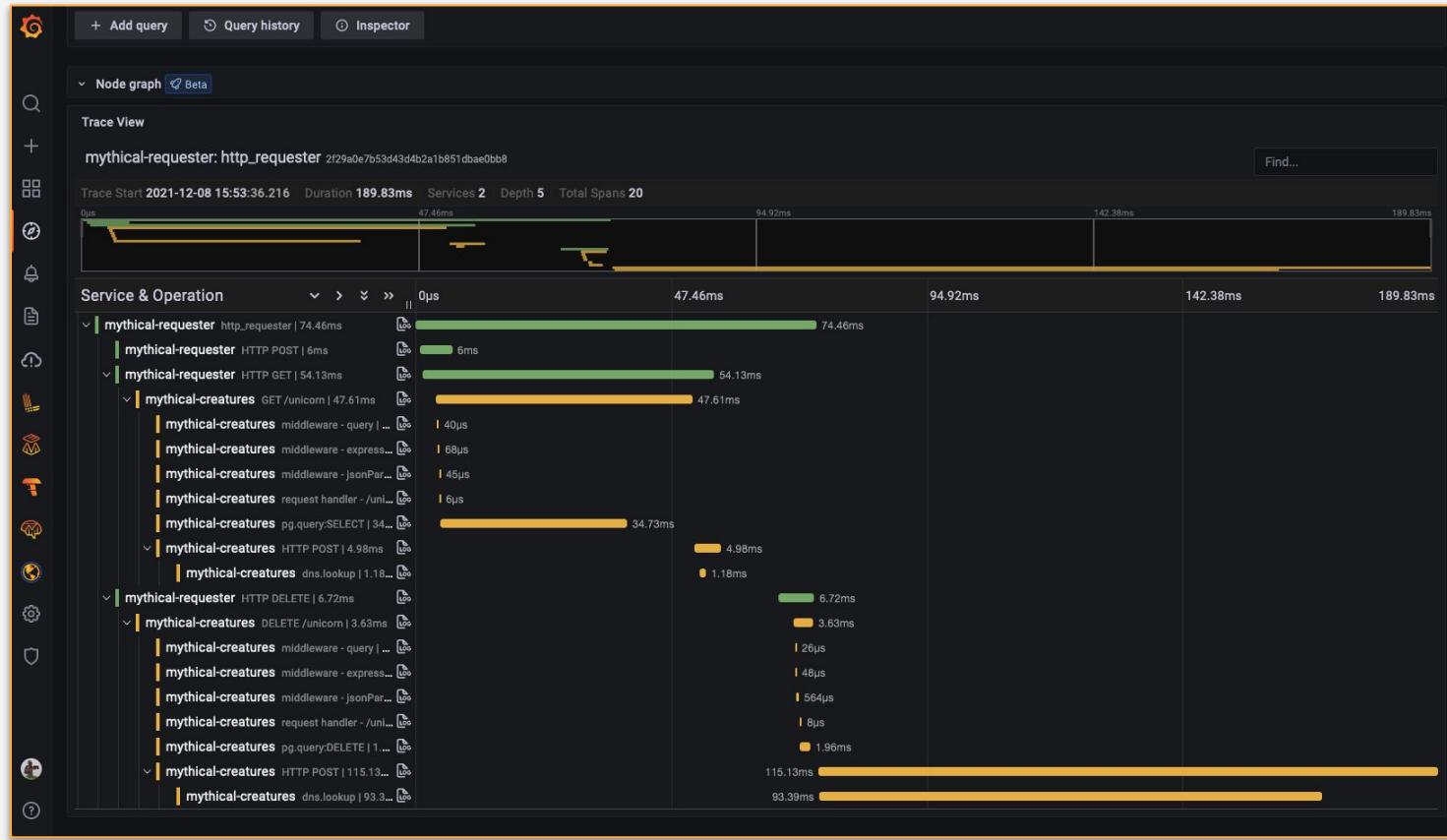
Poll: Are you familiar with distributed tracing?



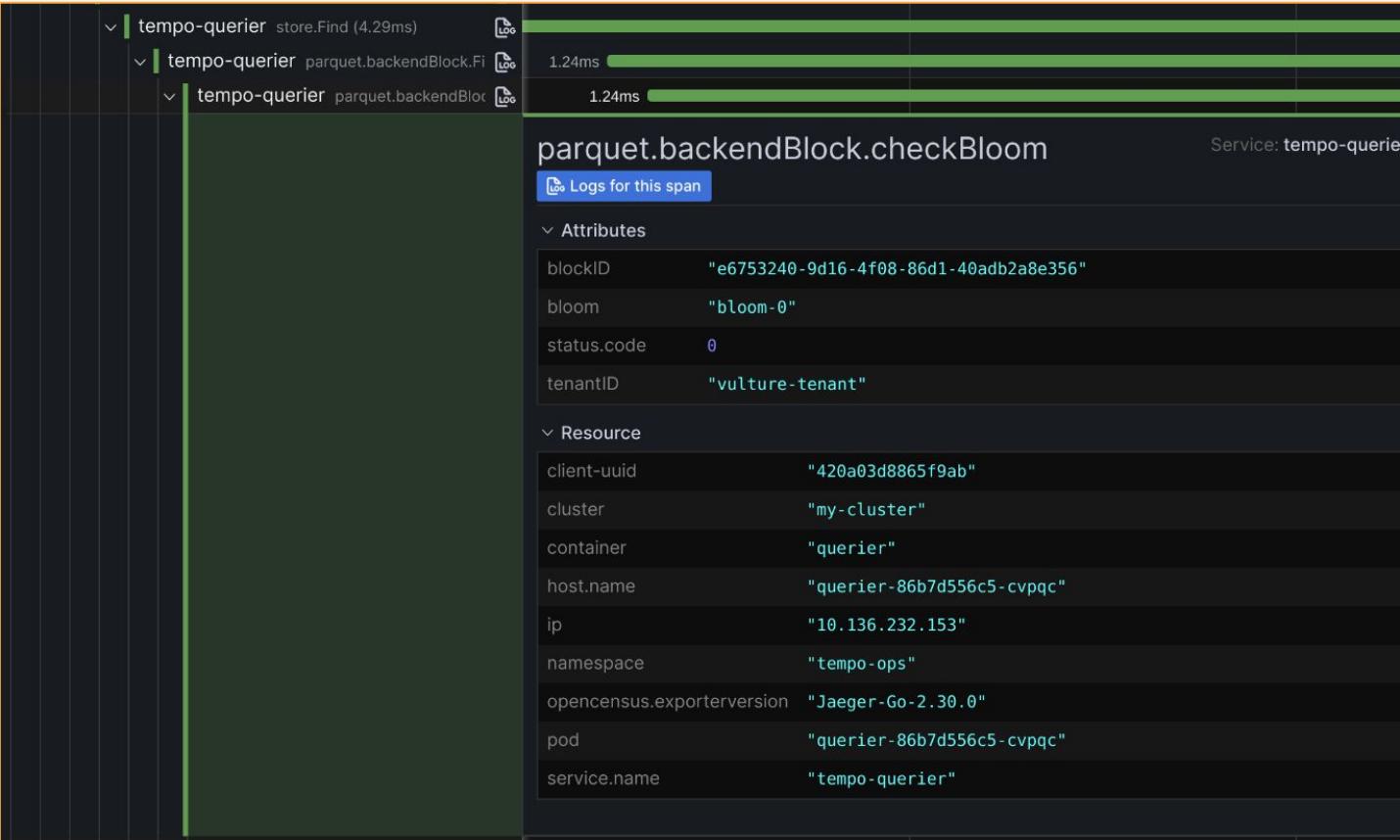
# What's distributed tracing?



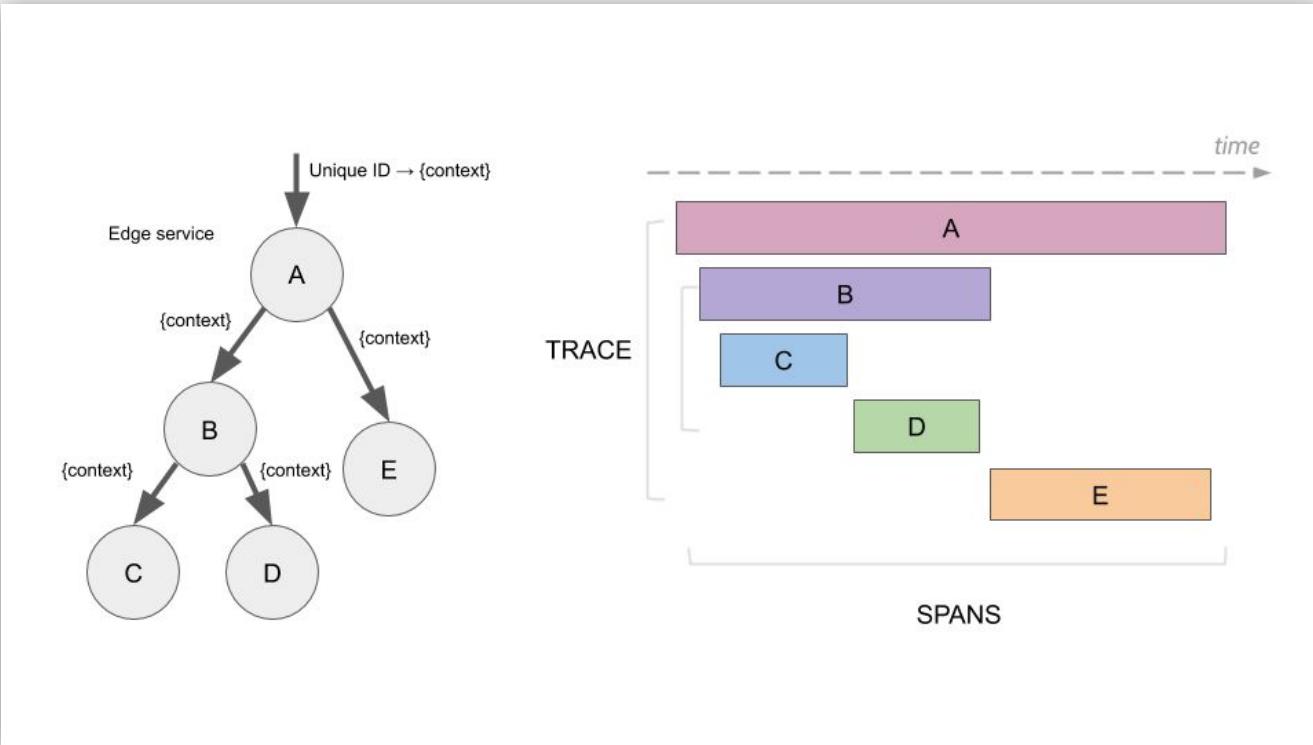
# Trace



# Span



# Context Propagation



# Why do we need tracing?



Modern services  
are complex

Logs and  
metrics are half  
of the story

More detailed  
view into data  
transactions



# So - how do I get started?



# Step 1 - Instrument your application

- Many ways to instrument
  - Open Telemetry
  - Jaeger
  - Zipkin
- Many Languages have auto Instrumentation in client SDK

## Instrumentation

OpenTelemetry code instrumentation is supported for the languages listed below. Depending on the language, topics covered will include some or all of the following:

- Automatic instrumentation
- Manual instrumentation
- Exporting data

### C++

A language-specific implementation of OpenTelemetry in C++.

### .NET

 A language-specific implementation of OpenTelemetry in .NET.

### Erlang/Elixir

 A language-specific implementation of OpenTelemetry in Erlang/Elixir.

### Go

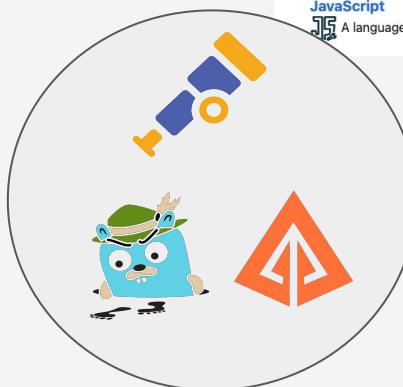
 A language-specific implementation of OpenTelemetry in Go.

### Java

 A language-specific implementation of OpenTelemetry in Java.

### JavaScript

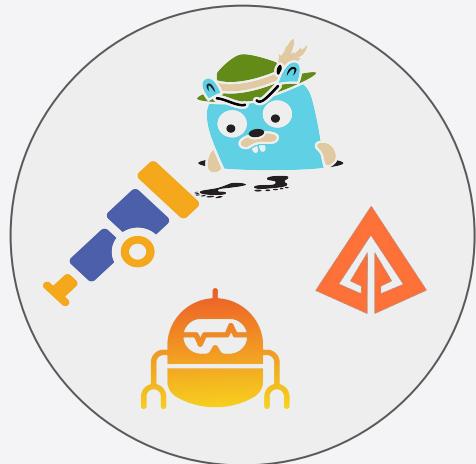
 A language-specific implementation of OpenTelemetry in JavaScript (for Node.JS & the browser).



# Step 2 - Set up your tracing pipeline

A pipeline (also collector) provides:

- Buffering of spans for efficient data transmission
- Ability to sample and control the flow of traces
- Enriching & processing of trace data



# Step 3 - Set up your tracing backend



Somewhere to store all the traces that you generate...

...and a way to query those traces back!



# Step 4 - Visualize

ROUTING: post /location/update/v4

Duration: 131.848ms Services: 10 Depth: 4 Total Spans: 13 Trace ID: a03e8fff1cd9b9a

ROUTING post /location/update/v4 [131.848ms]

Annotations

Tags

Service & Operation

The Jaeger UI interface displays a **DAG** (Directed Acyclic Graph) view of a trace. The graph consists of four teal circular nodes representing service instances:

- istio-ingressgateway** (top)
- followworld** (middle-left)
- istio-telemetry** (middle-right)
- istio-policy** (bottom-right)

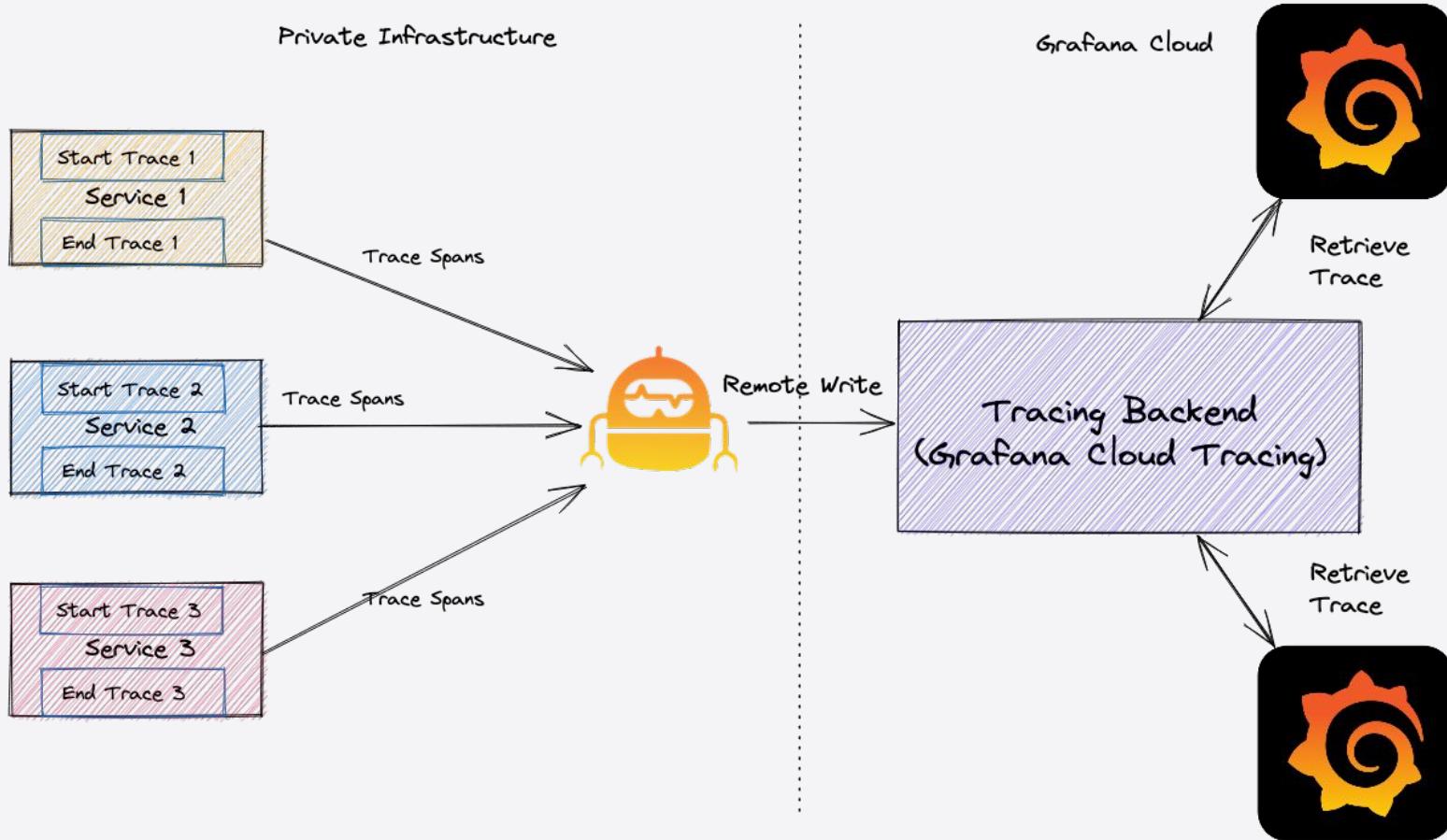
Arrows indicate the flow of data between these services, with numerical values representing latency in milliseconds:

- A bidirectional arrow between **istio-ingressgateway** and **followworld** has values 60 and 20.
- An arrow from **followworld** to **istio-telemetry** has a value of 20.
- An arrow from **istio-telemetry** to **istio-policy** has a value of 40.

A small blue cartoon character with a green hat is positioned on the right side of the interface.

Below the graph, a histogram provides a detailed view of the latency distribution for the trace. The x-axis represents time in milliseconds, with markers at 64.02ms, 96.04ms, and 128.05ms. The y-axis represents the number of samples. The histogram bars are colored red, blue, and orange, corresponding to the nodes in the graph. The distribution shows a peak around 96.04ms.

# How to do Distributed Tracing (with Tempo and Grafana Agent!)

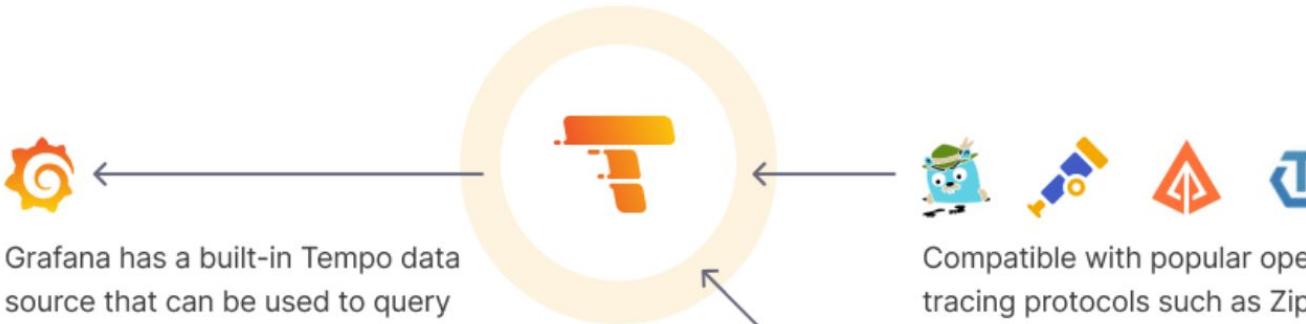




# Grafana Tempo



# How does Grafana Tempo work?



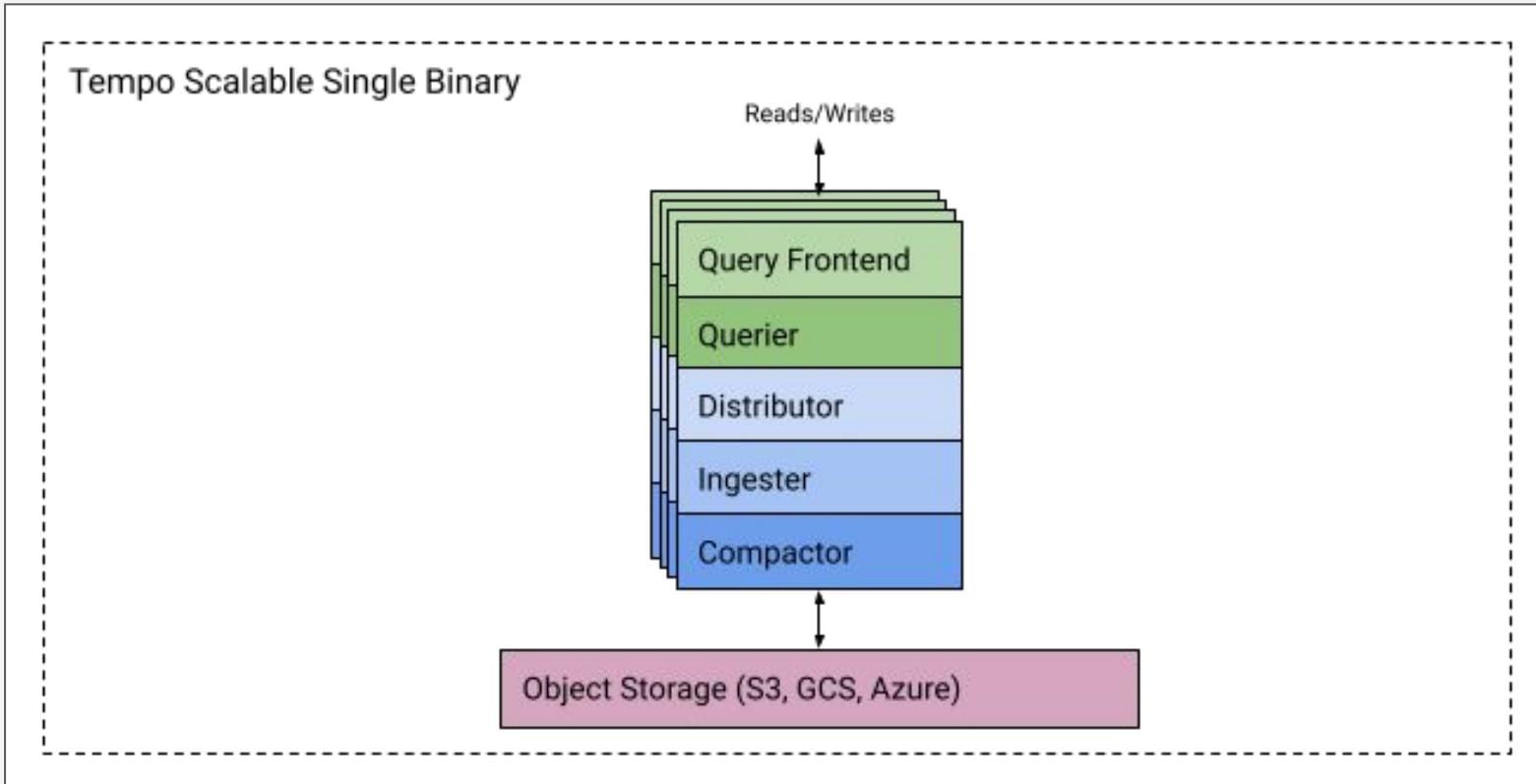
With Tempo, the only dependency is object storage (GCS, S3, Azure Blob Storage), which makes it easy to hit massive scale while keeping costs low.



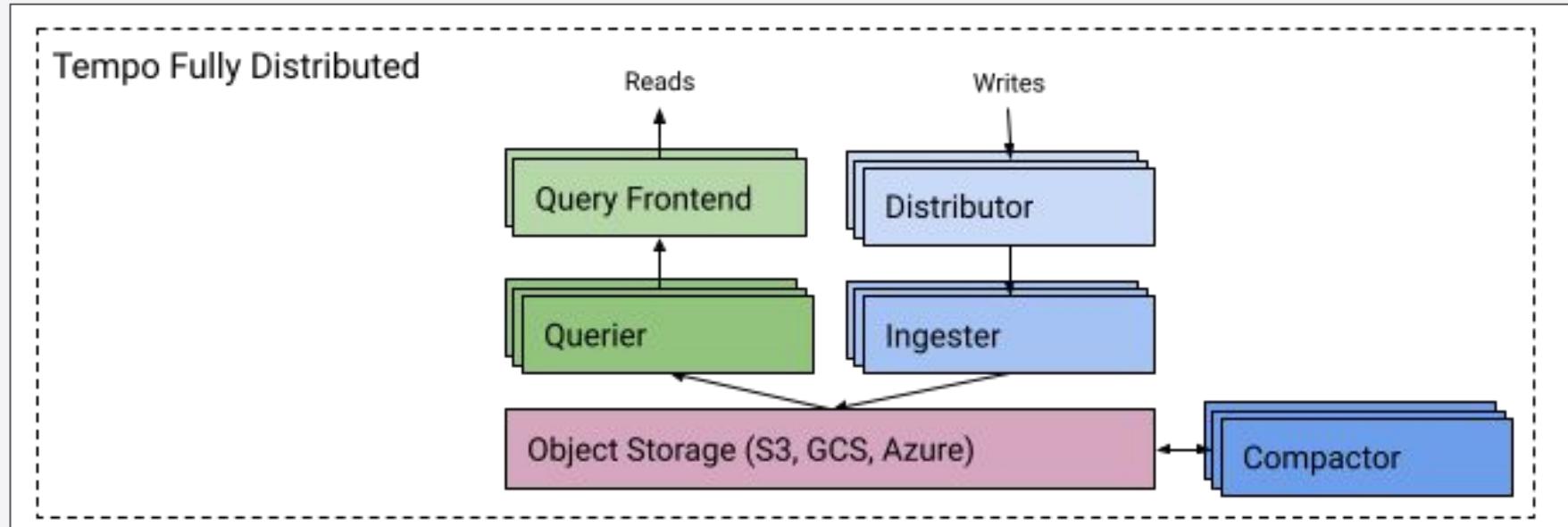
# Running Tempo



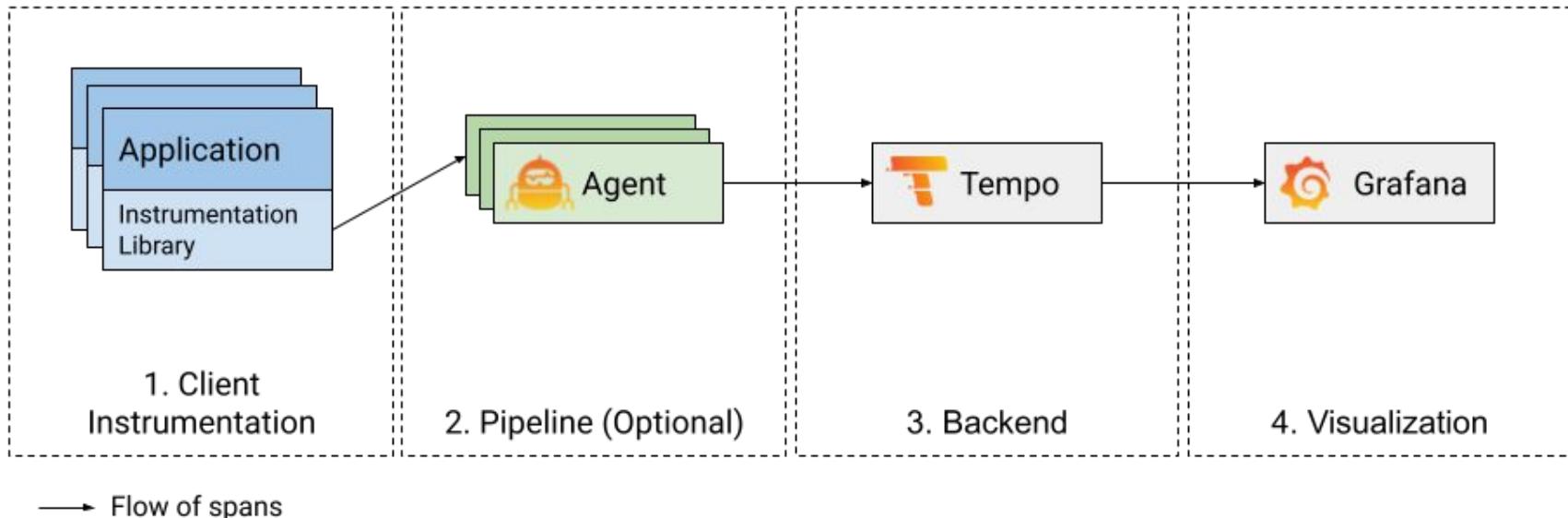
# Run it as a single process



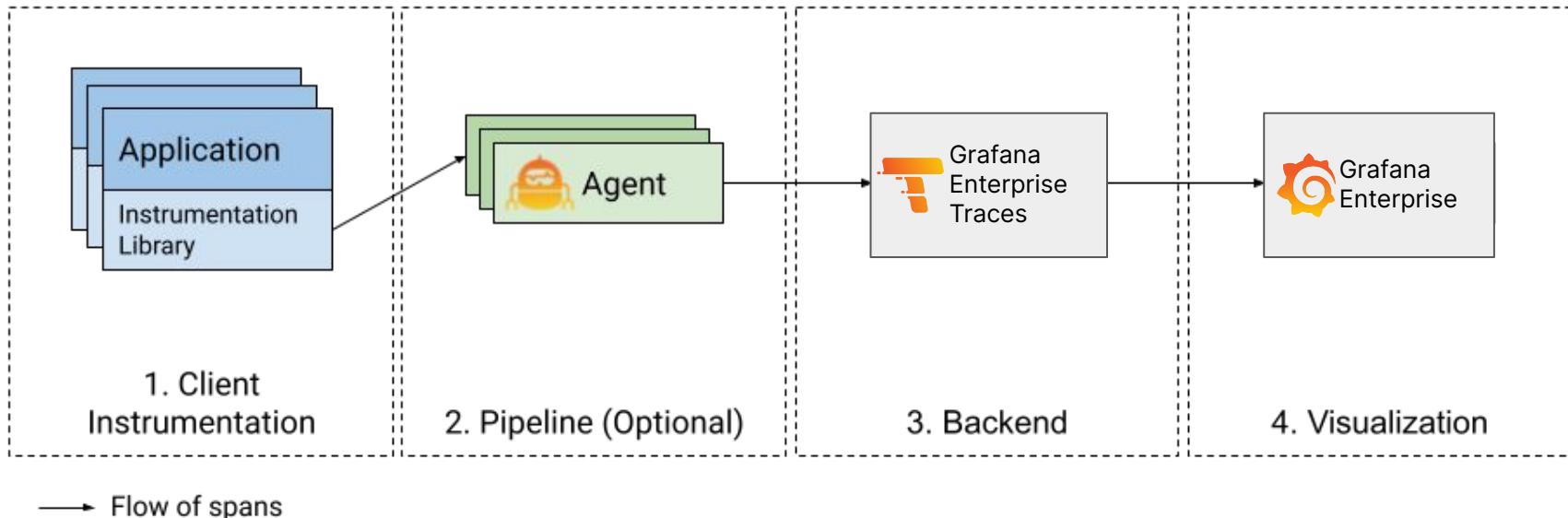
# Run it as microservices



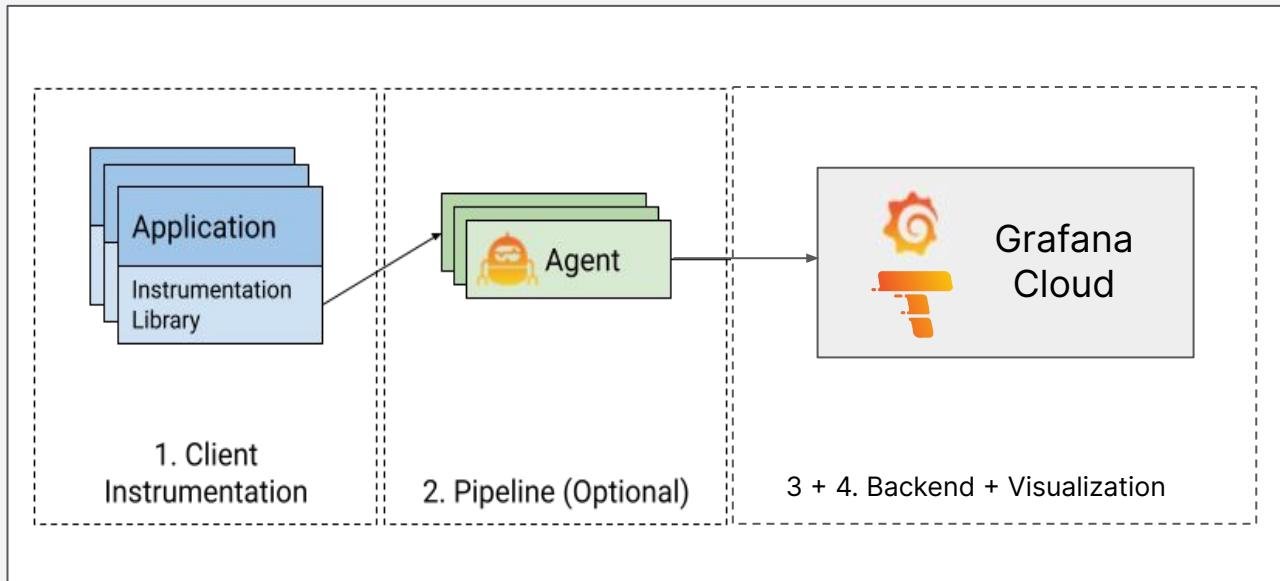
# Run it yourself with open source



# Run it yourself with Enterprise



# Grafana Cloud Traces, fully managed



You focus on instrumentation, we handle the rest.

Up to 50GB/month free, with 30 day retention



# TraceQL

```
{ .service.name = "foo" &&  
.http.status_code = 500 }
```



# I don't want to learn your garbage query language

2018-08-30

This is a bit of a rant but I really don't like software that invents its own query language. There's a trillion different ORMs out there. Another trillion databases with their own query language. Another trillion SaaS products where the only way to query is to learn some random query DSL they made up.

- Splunk has [SPL](#)
- Mixpanel has [JQL](#)
- Rollbar has [RQL](#)
- New Relic has [NRQL](#)
- Adwords has [AWQL](#)



**Hacker News** [new](#) | [past](#) | [comments](#) | [ask](#) | [show](#) | [jobs](#) | [submit](#)

▲ I don't want to learn your query language (2018) ([erikbern.com](#))

297 points by nud piedo on March 10, 2021 | hide | past | favorite [218 comments](#)



Source: [I don't want to learn your garbage query language](#)

# TraceQL - Why?

Traces are the flow of events throughout your components, in a tree structure.

We believe you should be able ask more questions from this data, instead of just looking at single trace



# TraceQL - Why?

## Example:

- *How many connections to the database are your slowest requests doing?*
- *Show me requests that took more than 10 seconds AND did more than 3 SQL queries?*



# Type awareness

```
{ .http.status_code >= 200 &&  
  .http.status_code < 300 }
```

```
{ duration > 1s && duration < 2s }
```



# Regex

```
{ .http.url =~ "/api/v1/.*" }
```

```
{ .environment =~ "prod-.*" }
```



# Aggregates

```
{ .db.name = "postgres" } | count() > 1
```

```
{ .db.statement =~ "INSERT.*" } |  
avg(duration) > 1s
```



```
{  
  .http.status_code >= 500 &&  
  .http.url =~ "/api/.*" &&  
  ( .http.method = "POST" || .http.method = "PUT" )  
} | avg(duration) > 1s
```

**Show me all the spans with 5xx or higher, where path contains /api/ and request is POST or PUT and avg duration of span is more then 1 second**



# TraceQL - Pipelining

The first version of TraceQL includes basic pipelining that can be used to filter span sets using the following **aggregations**:

- **Average:** `{name="service"} | avg(duration) > 1 ms`
  - Selects all the traces with spans with name “service” that have an average duration over 1 ms.
- **Count:** `{.service.name="db"} | count() > 10`
  - Selects all the traces that contain more than 10 spans with service.name “db”.



# Advanced TraceQL

## Aggregates

```
{ .cluster = "prod" } | by(.namespace)
```

```
{ .cluster = "dev" } | max(end) - min(start) > 10s
```

## Pipelining

```
{ .cluster = "prod" } | by(.namespace) | count() > 5
```

```
{ .cluster = "dev" } | avg(duration) > { .cluster = "prod" } | avg(duration)
```



# Advanced TraceQL

## Structural operators

```
{ .namespace = "test" } >> { .namespace = "ops" }
```

```
{ .namespace = "test" } ~ { .namespace = "ops" }
```

## TraceQL metrics (experimental)

```
{ .cluster = "prod" } | rate(5m)
```

```
{ .cluster = "prod" } | select(span.http.status_code, span.http.url)
```



# TraceQL - Grafana Support

Connects with the Trace view in order to expand any of the traces returned.

The screenshot illustrates the integration of TraceQL within the Grafana interface. On the left, the 'Query editor' pane displays a TraceQL query: `{ .http.status_code >= 500 || duration } duration`. A callout box highlights the 'Query editor with Intellisense to build TraceQL queries.' Below the query editor is a 'Results Table' showing trace information:

Trace ID	Name	Start time	Duration
59e0a4acbd34ab23...	mythical-requester r...	2 minutes ago	53 ms
2222e87b6f6dc9a8			201
c56bfefcbdd3d68c		2 minutes ago	204
557422d4a34b9b2...	mythical-requester r...	2 minutes ago	63 ms
17a0cc65809e9467...	mythical-requester r...	2 minutes ago	92 ms
1f7436f507e8145ef...	mythical-requester r...	3 minutes ago	51 ms
386cff5fd64ee7c3...	mythical-requester r...	3 minutes ago	68 ms
2c99a0ff6746e125...	mythical-server GET...	3 minutes ago	8 ms
6d55f16a6990c006...	mythical-requester r...	3 minutes ago	68 ms

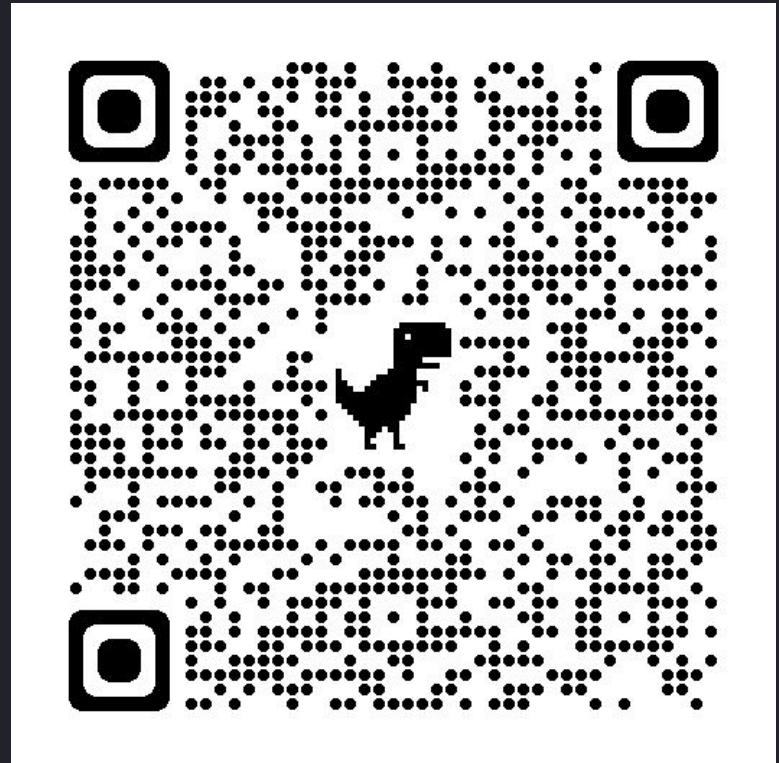
A callout box highlights the 'Results Table shows the traces with the spans matched by the query.'

On the right, the 'Trace' view displays a timeline of events for a specific trace. The timeline shows various spans and their durations. A detailed breakdown of one trace is shown on the right, with labels like 'mythical-requester' and 'mythical-server' indicating the service and component for each span. A callout box highlights the ability to 'Connects with the Trace view in order to expand any of the traces returned.'



Want to learn more TraceQL??

Scan QR code to watch an hour long session on TraceQL :)



# How is this possible with object storage?

i.e. Why and how parquet helps??

# Requirements for TraceQL

- Data with very high cardinality
- Large operations involving subsets of columns
- Operations involving rows only when retrieving traces
- Only object storage



**Apache Parquet + Object Storage**

# Physical storage layout models

Logical

	Col A	Col B	Col C
Row 0	A0	B0	C0
Row 1	A1	B1	C1
Row 2	A2	B2	C2
Row 3	A3	B3	C3
Row 4	A4	B4	C4
Row 5	A5	B5	C5

---

Physical



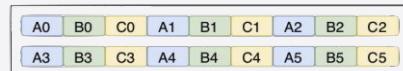
# Physical storage layout models

Logical

	Col A	Col B	Col C
Row 0	A0	B0	C0
Row 1	A1	B1	C1
Row 2	A2	B2	C2
Row 3	A3	B3	C3
Row 4	A4	B4	C4
Row 5	A5	B5	C5

---

Physical



Row-wise

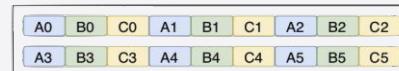


# Physical storage layout models

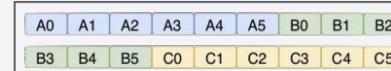
Logical

	Col A	Col B	Col C
Row 0	A0	B0	C0
Row 1	A1	B1	C1
Row 2	A2	B2	C2
Row 3	A3	B3	C3
Row 4	A4	B4	C4
Row 5	A5	B5	C5

Physical



Row-wise

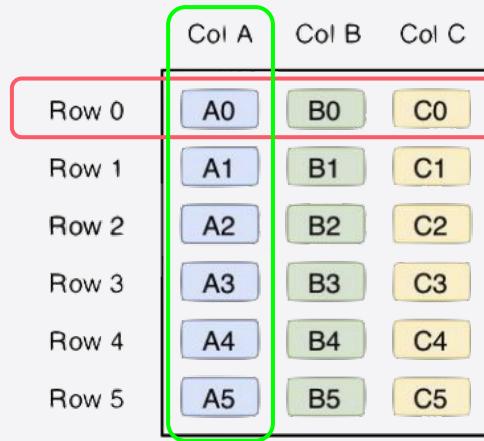


Columnar

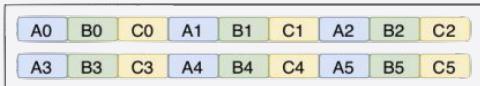


# Physical storage layout models

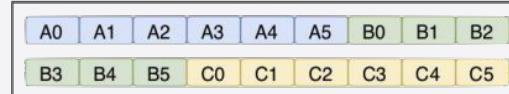
Logical



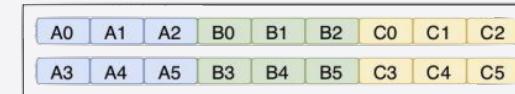
Physical



Row-wise



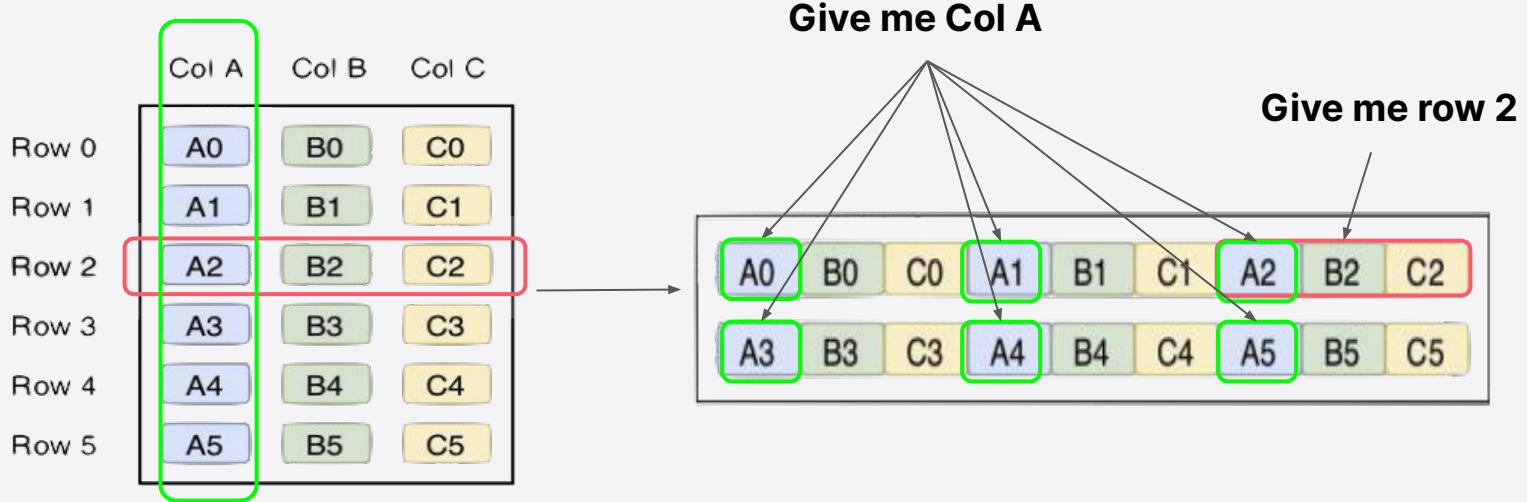
Columnar



Hybrid



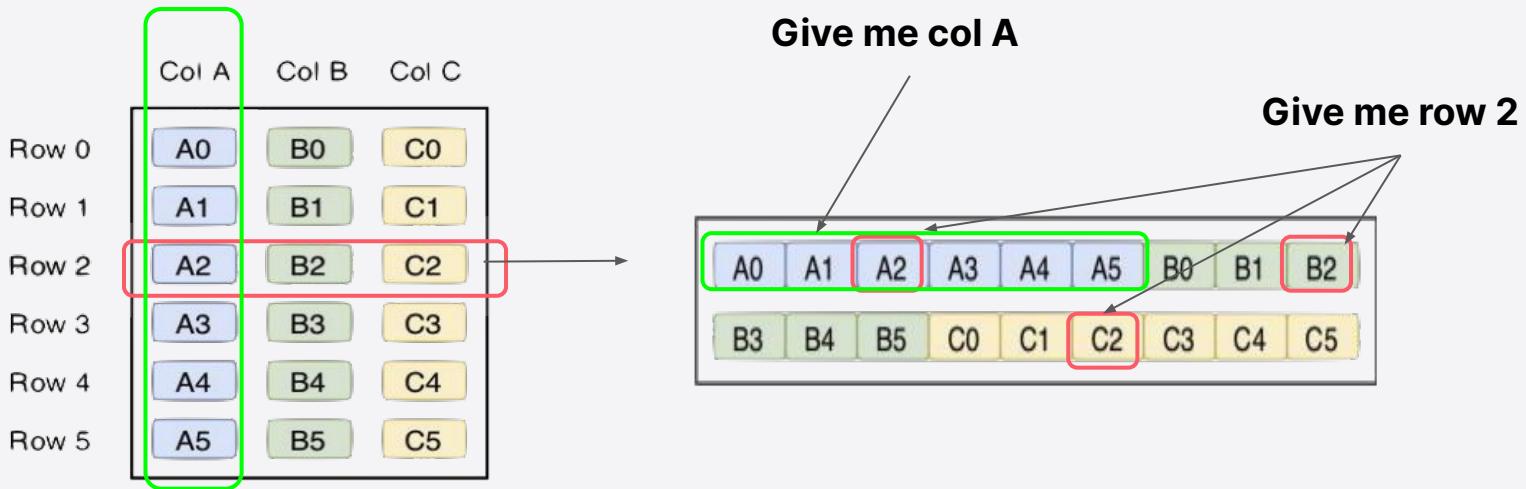
# Row-wise



- Horizontal partitioning
- Great for lots of small operations involving **whole rows**



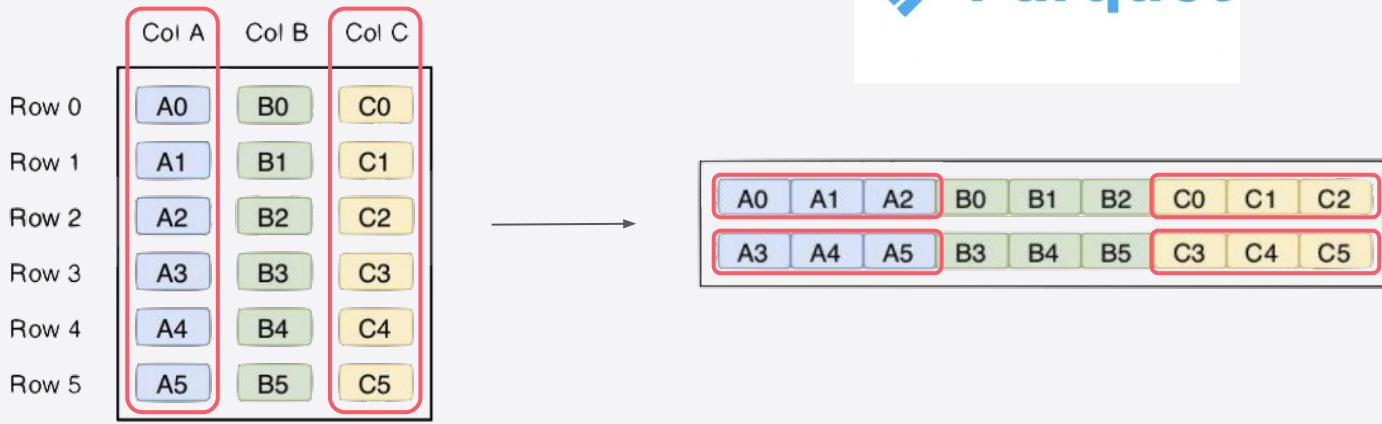
# Columnar



- Vertical partitioning
- Great for few large operations involving ***subset of all columns***



# Hybrid



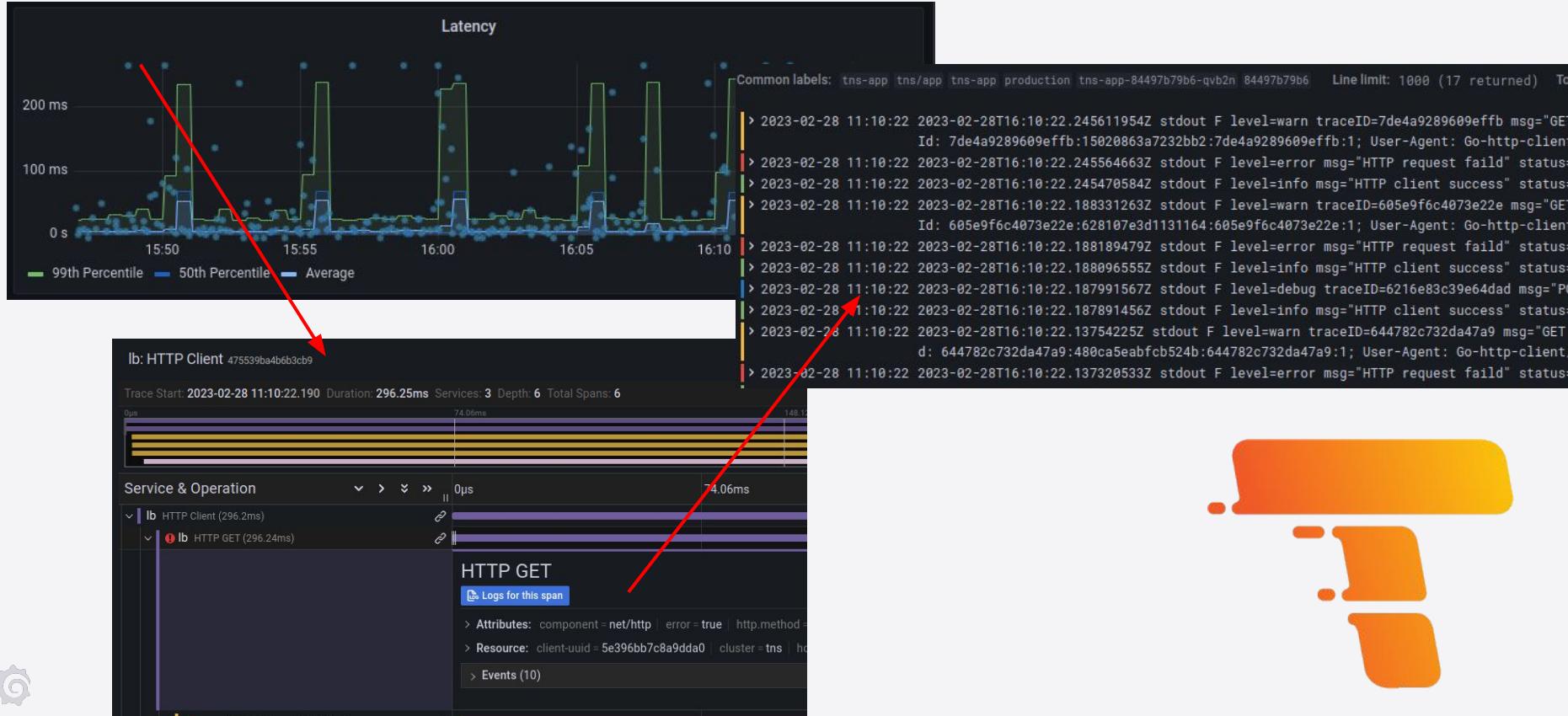
- Data with very high cardinality
- Large operations involving subsets of columns
- Operations involving rows only when retrieving traces



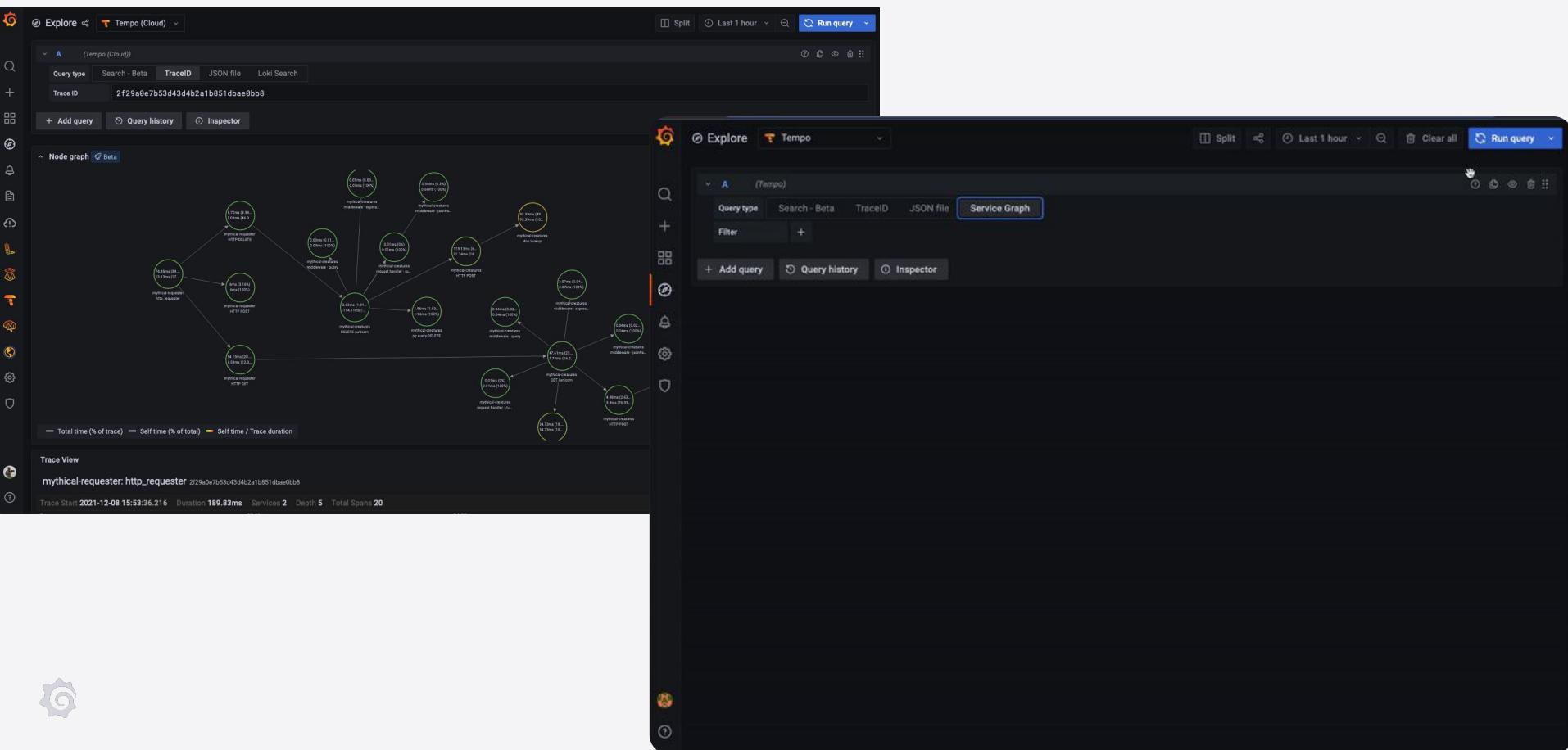
# How to get more from Tracing?



# Correlations - Triage

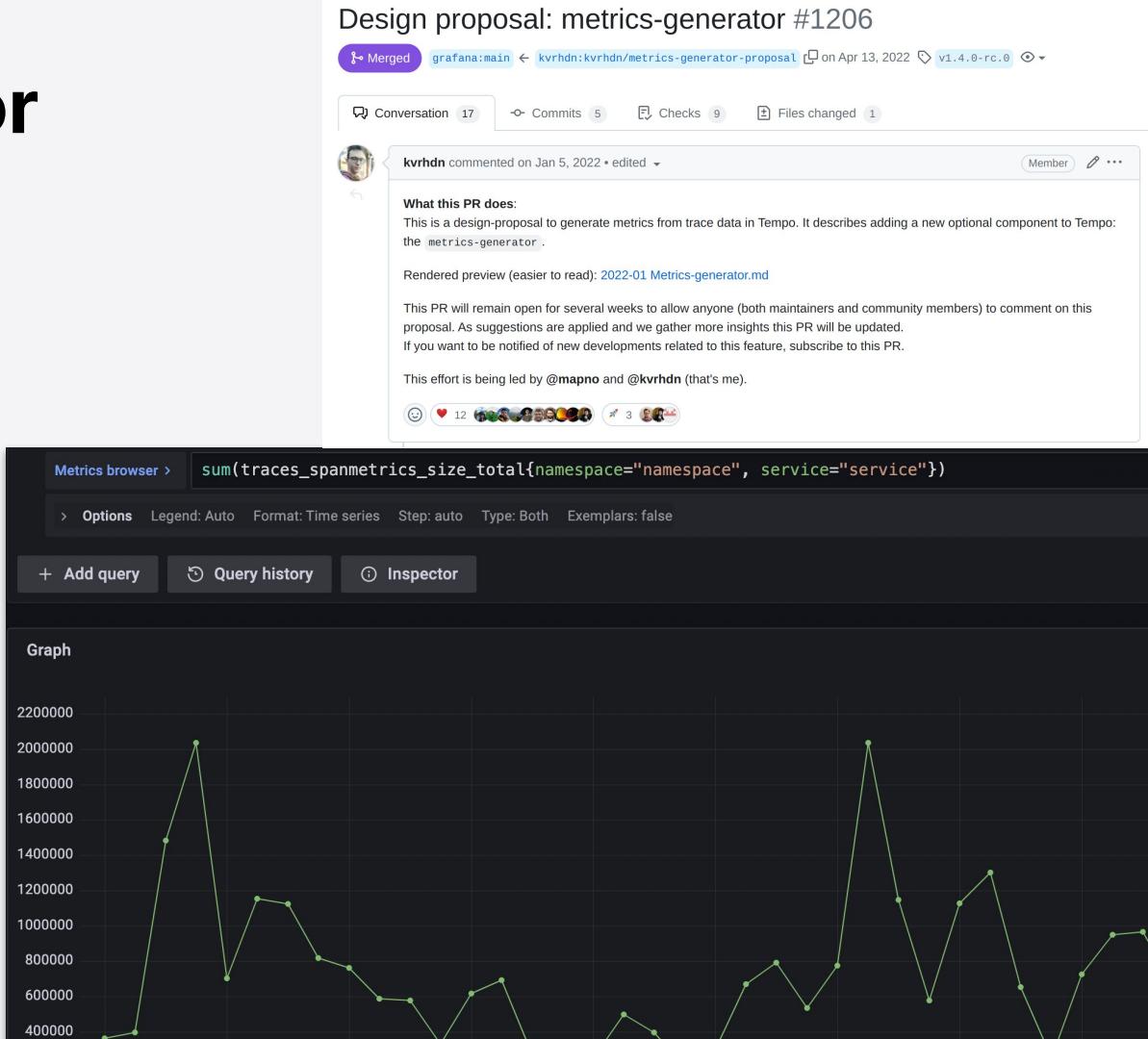


# node (or service) graphs

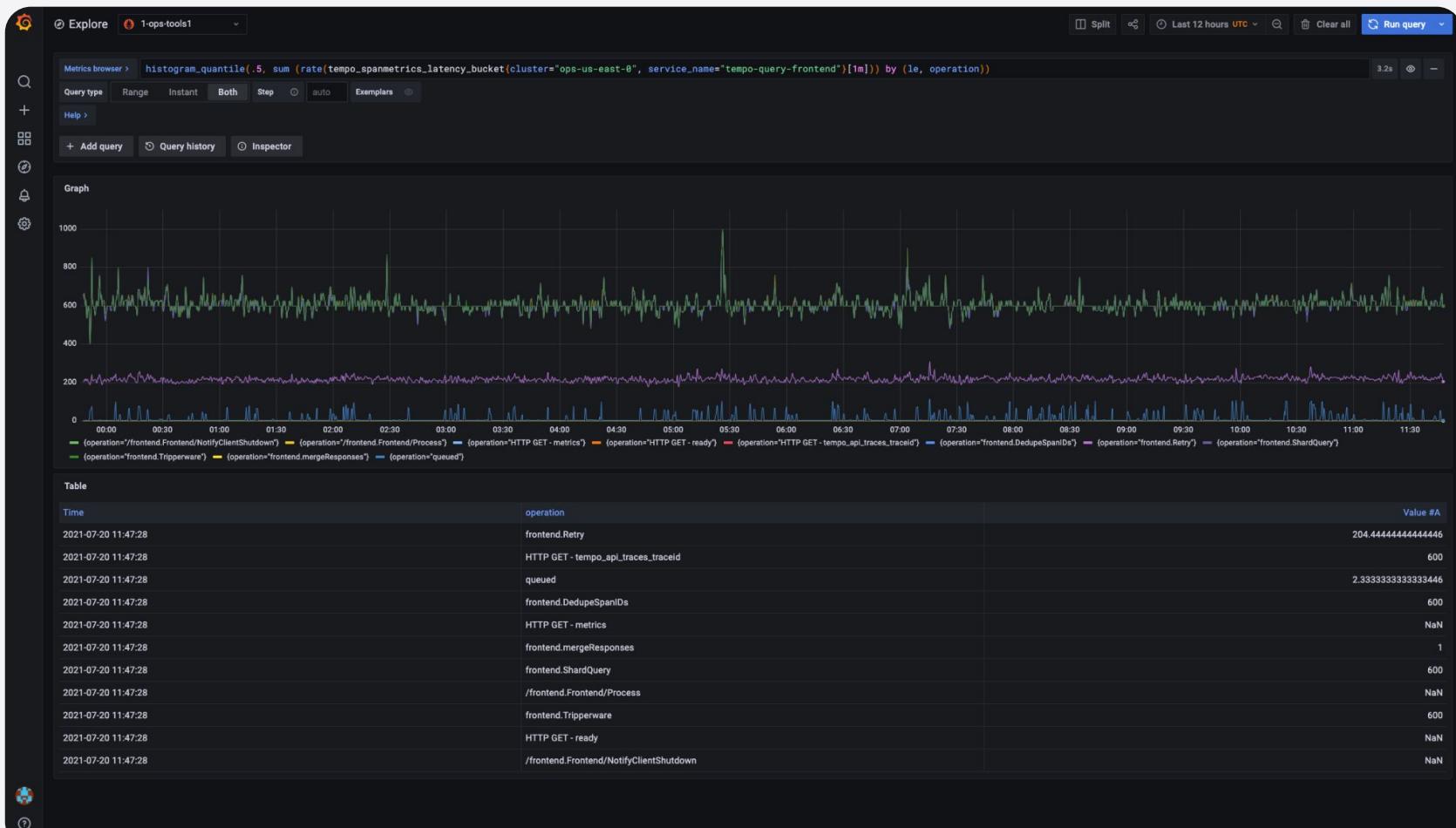


# Metrics generator

Tempo's metrics generator remote writes span metrics to a Prometheus compatible backend.



# span metrics





# Metrics Generation - Application Insights

Use data links to move easily from dashboard panels to trace exploration

The screenshot illustrates the integration between Grafana's metrics dashboard and its tracing capabilities.

**Metrics Dashboard:** On the left, a dashboard displays various metrics. A red arrow points from the error percentage for the endpoint `/account` (4.41) to the TraceQL query editor.

**TraceQL Editor:** In the center, the TraceQL editor shows a query:

```
{ .http.target = "/account" && status = error }
```

A red arrow points from the highlighted trace ID `a5fa9b1f604b4ac316fc6...` in the Trace View table below to the TraceID input field in the TraceQL editor.

**Trace View:** On the right, the Trace View details a specific trace. It shows the trace ID `1f2ed24945d1cdaa6c1a206d68bc0082`, the start time `2023-01-25 12:47:01.915`, and the duration `23.67ms`. The trace tree shows spans for `mythical-requester` and `mythical-server`.

Trace ID	Start time	Name
<code>a5fa9b1f604b4ac316fc6...</code>	<code>2023-02-28 11:25:14</code>	mythical-requester: requester
Span ID	Start time	Service & Operation
<code>51c315f7046cdf1f</code>	<code>2023-02-28 11:25:14</code>	mythical-requester: request

# Demo!

<https://github.com/grafana/intro-to-mlt>





# Q & A



# Thank you

Slides: [suraj.dev/talks](https://suraj.dev/talks)

Twitter: [@electron0zero](https://twitter.com/electron0zero)

