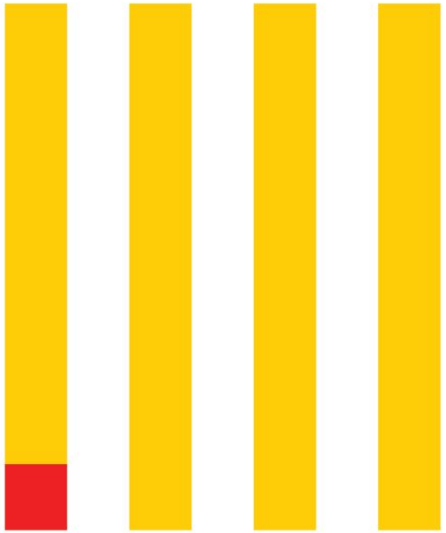
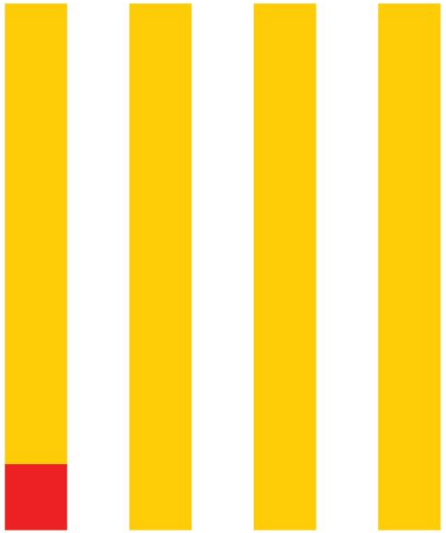


ClickHouse

Amsterdam Meetup With The ClickHouse Team – June 8th, 2022

05.25.2022

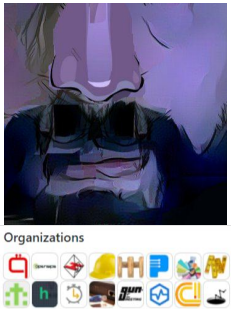


ClickHouse

Amsterdam Meetup With The ClickHouse Team – June 8th, 2022

05.25.2022

Good Evening & Welcome 👋



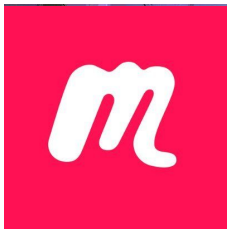
Lorenzo Mangani

- Co-Founder and CEO @ qXIP + uptrace
- Open-Source Fanatic, Proud Father, Workaholic and totally in love with my Job
- Leading an amazing team specialized in active and passive large scale monitoring and ingestion solutions for Traces, Logs, Packets and Metrics on top of **ClickHouse** quietly serving some of the *largest* and *most innovative enterprises* on the planet

- ⭐ Proud members of the *"wild"* ClickHouse OSS community since inception
- ⭐ Supporting hundreds of heavy duty ClickHouse server nodes worldwide
- ⭐ Contributors to our community with dozens of useful projects and tools
- ⭐ Early alpha testers of ClickHouse Cloud *(thanks to the wonderful ClickHouse team!)*

qXIP uptrace hepic. sentinl HOMER

Tonight's Menu



A tale of two ClickHouse projects

- MEET UPTRACE & CLOKI
  - What are they for and why do we care?
  - OpenTelemetry Everything vs. Unstructured Data
  - Merging towards Data MMA (mixed monitoring arts)
  - Competition & Advantages other other solutions
- UPTRACE & CLICKHOUSE
  - High Performance, Low Maintenance, Strong Leadership
  - Super powered by ClickHouse Inc & ClickHouse Cloud
- WHAT IS NEXT
  - Uptrace Cloud Promotion
  - Open-Source & Community

What is Uptrace?

Uptrace is an OpenTelemetry tracing tool that monitors performance, errors, and structured logs. It collects, aggregates and processes data to help users quickly pinpoint *service failures* and find *performance bottlenecks*.

Opentelemetry is an open source and vendor-neutral API for distributed traces *(including logs and errors)* and metrics. It specifies how to collect and export telemetry data. With OpenTelemetry, developers can instrument their application once and then add or change monitoring vendors without changing the instrumentation.

What is cLoki?

cLoki is fast and powerful Log & Metrics management tool built on top of ClickHouse. It provides ready-to-use APIs transparently compatible with *LogQL, PromQL, InfluxDB, Elastic and more* to ingest logs and metrics from industry standard Agents such as *Promtail, Grafana-Agent, Vector, Logstash, Telegraf and many others.*

cLoki implements a *custom, extensible LogQL/PromQL transpiler* to dynamically search, filter and extract data and metrics from structured and unstructured logs, events, shipping with plugin-less Grafana LogQL/Tempo integrations as well as a built in Explore UI and LogQL command line tools for querying and extracting data.

What is Uptrace?

Uptrace is an OpenTelemetry tracing tool rors, and structured logs. It collec
aggregates and processes data to help us *ilures* and find *performance bott*

Opentelemetry is an open source and ver traces *(including logs and er*
metrics. It specifies how to collect and ex Telemetry, you can instrument your
application once and then add or change instrume

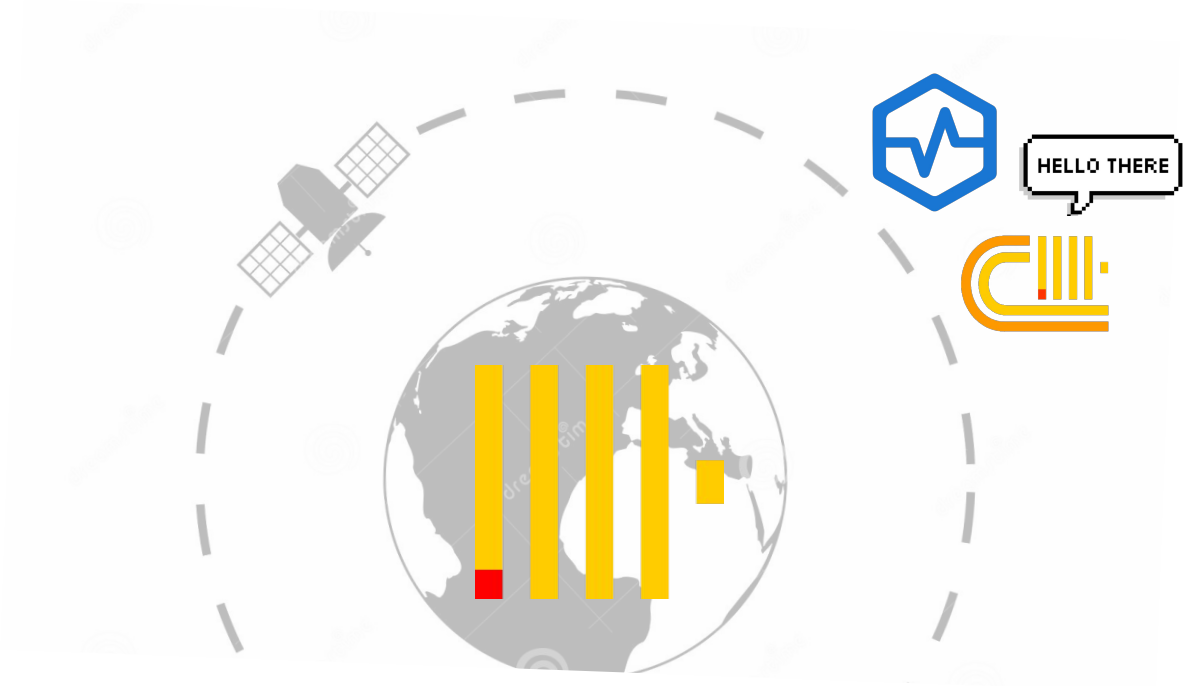
What is cLoki?

cLoki is fast and powerful Log & Metrics management tool built on top of ClickHouse. It provides ready-to-use
APIs transparently compatible with *LogQL, Prom Elastic and more* to ingest logs and metrics from
industry standard Agents such as *Promta tash, Telegraf and many*

cLoki implements a *custom, extensible* nically searc and
and metrics from structured and unstruc plugin-less Gra
integrations as well as a built in Explore UI and LogQL command line tools for querying and extrac

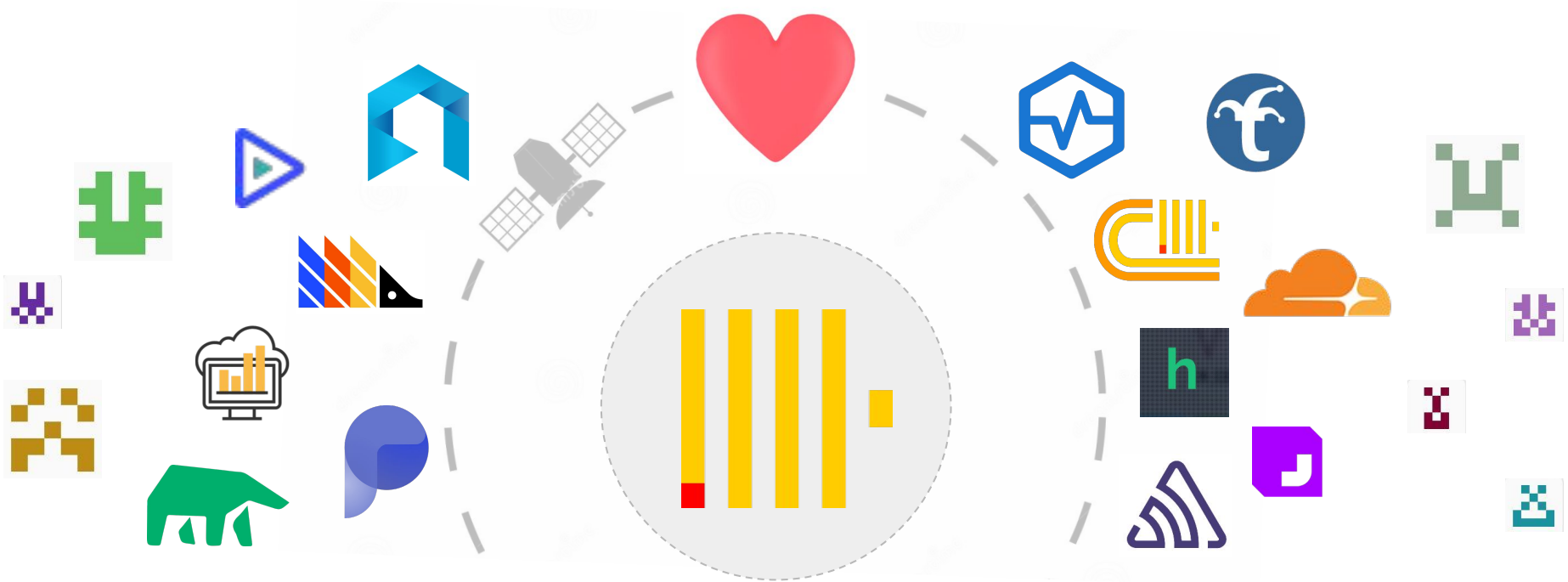
Community Powered

Like many, we've met in one of the many corners of the worldwide ClickHouse community.



Community Powered

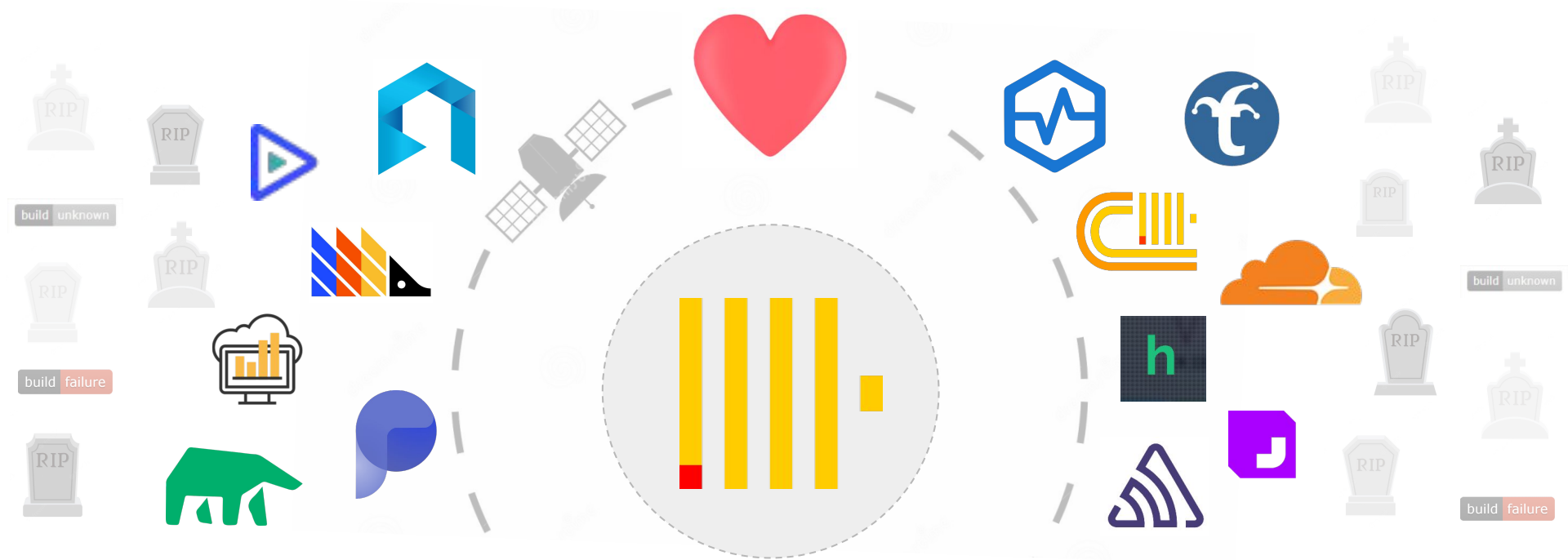
ClickHouse is like a powerful startup accelerator with many projects large and small floating around.



Community Powered…. *while staying alive*

ClickHouse is like a powerful startup accelerator with many projects large and small floating around.

Not without casualties. We lost count of semi-*abandonware* projects claiming to redefine logging and monitoring over ClickHouse over time



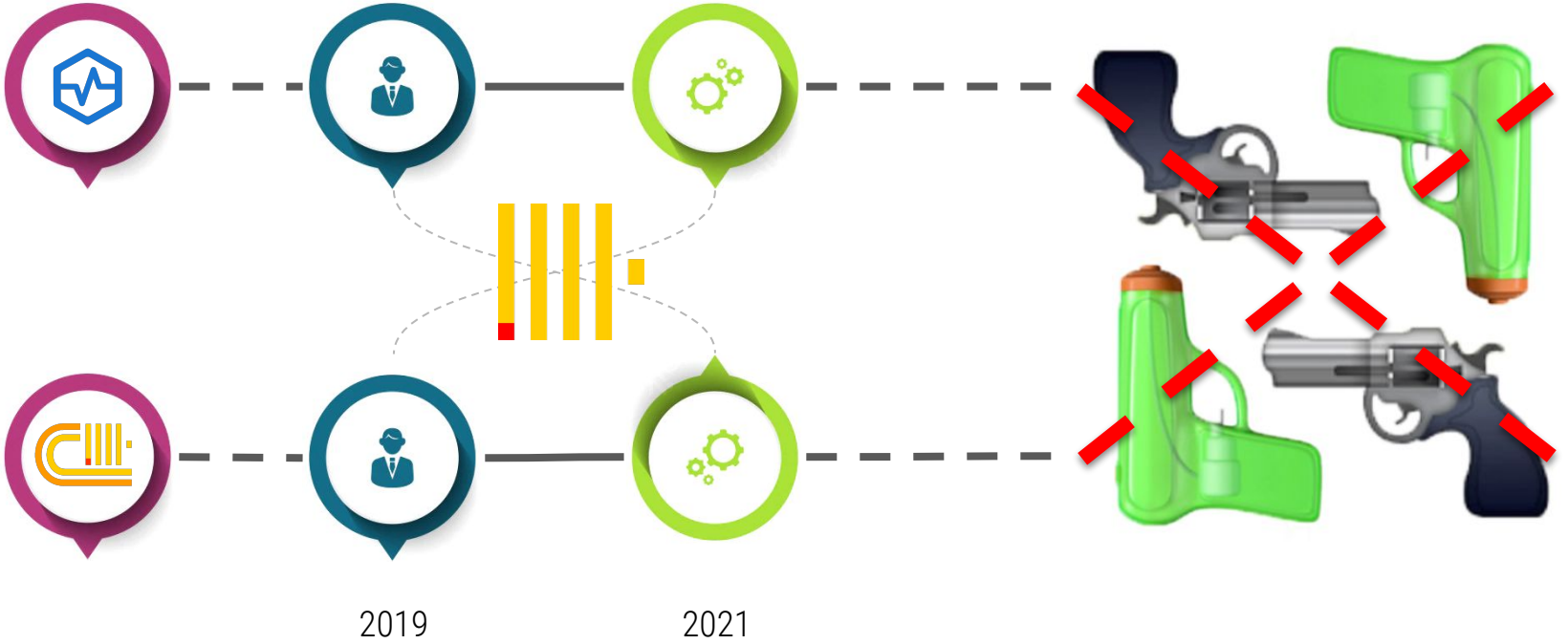
Community Powered…. *while staying alive*

Luckily both **Uptrace** and **cLoki** managed to remain healthy, got a following and growing user action!

Competing Powers?

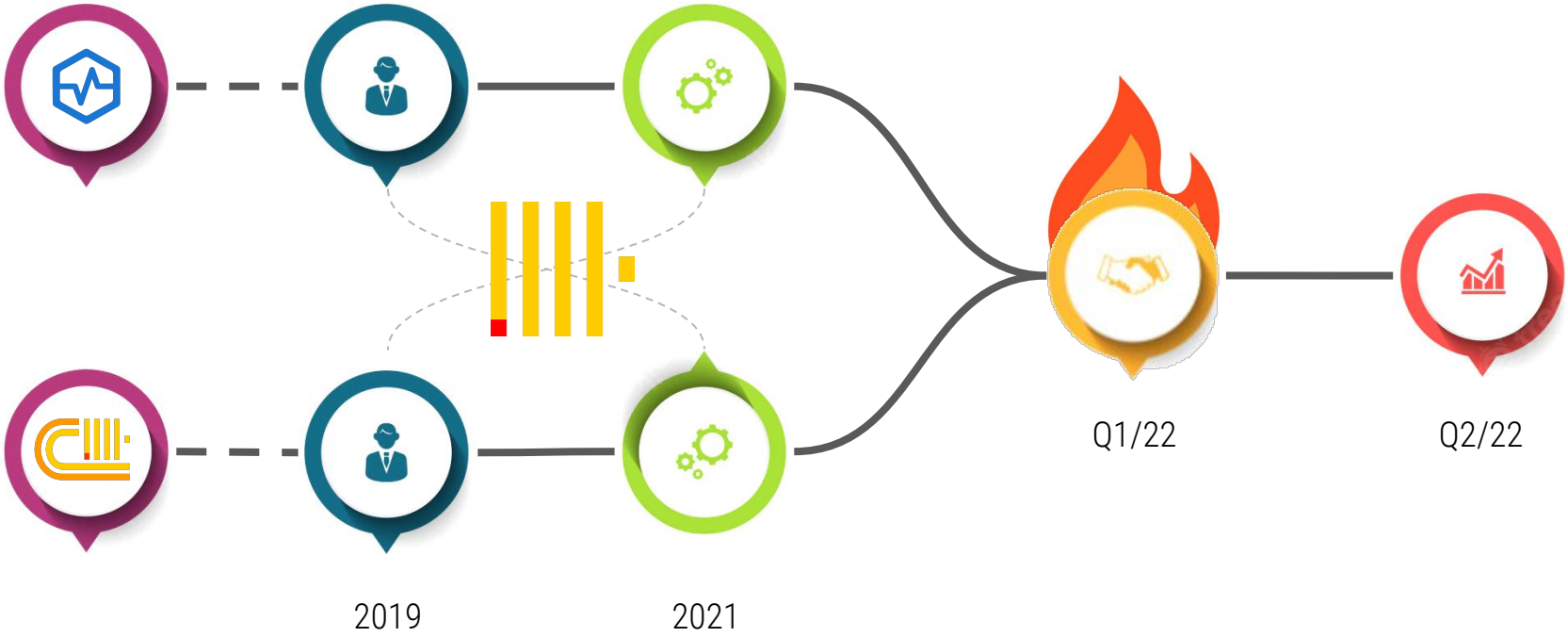
Our project coexisted fine but inevitably, users brought up the question - which does what best?

Since **Uptrace** could also do logging, and **cLoki** could also do tracing, *competition was inevitable* …



Combined Powers

Instead, we decided to **unite** and **combine** our skills & features into *one majestic single group/project*.
Under the **Uptrace** brand, we merged our *codebases, design & ideas* into *a fresh groundbreaking product*



Combined Dev Teams 🔥

Lorenzo Mangani

Vladimir Mihailenco

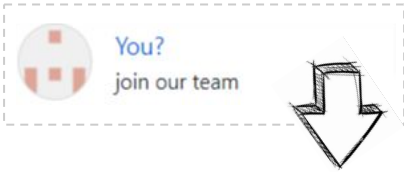
Alexandr Dubovikov

Anatolii Mihailenco

Volodymyr Akchurin

Joel Guerra

Jachen Duschletta



\+ *many OSS code contributors and supporters on gh*

Combined Projects 🔥

uptrace Public
OpenTelemetry distributed tracing tool that monitors performance, errors, and logs
Vue ☆ 702 38

uptrace-go Public
OpenTelemetry Go distribution for Uptrace
Go ☆ 53 9

uptrace-python Public
OpenTelemetry Python distribution for Uptrace
Python ☆ 4 2

uptrace-ruby Public
OpenTelemetry Ruby distribution for Uptrace
Ruby ☆ 3

uptrace-js Public
OpenTelemetry JavaScript distribution for Uptrace
TypeScript ☆ 6 3

uptrace-dotnet Public
OpenTelemetry .NET distribution for Uptrace
C# ☆ 2 1

cLoki Public
Forked from lmangani/cLoki
Clickhouse Loki: Grafana Loki API + ClickHouse Backend in NodeJS
JavaScript ☆ 4 1

fluXpipe Public
Experimental Flux API/Pipeline for ClickHouse and other embedded datasources
FLUX ☆ 10 2

promcasa Public
ClickHouse Custom Query / Metrics Exporter for Prometheus
Go ☆ 3

vLogQL Public
Forked from lmangani/vLogQL
LogQL Client in V
V ☆ 3

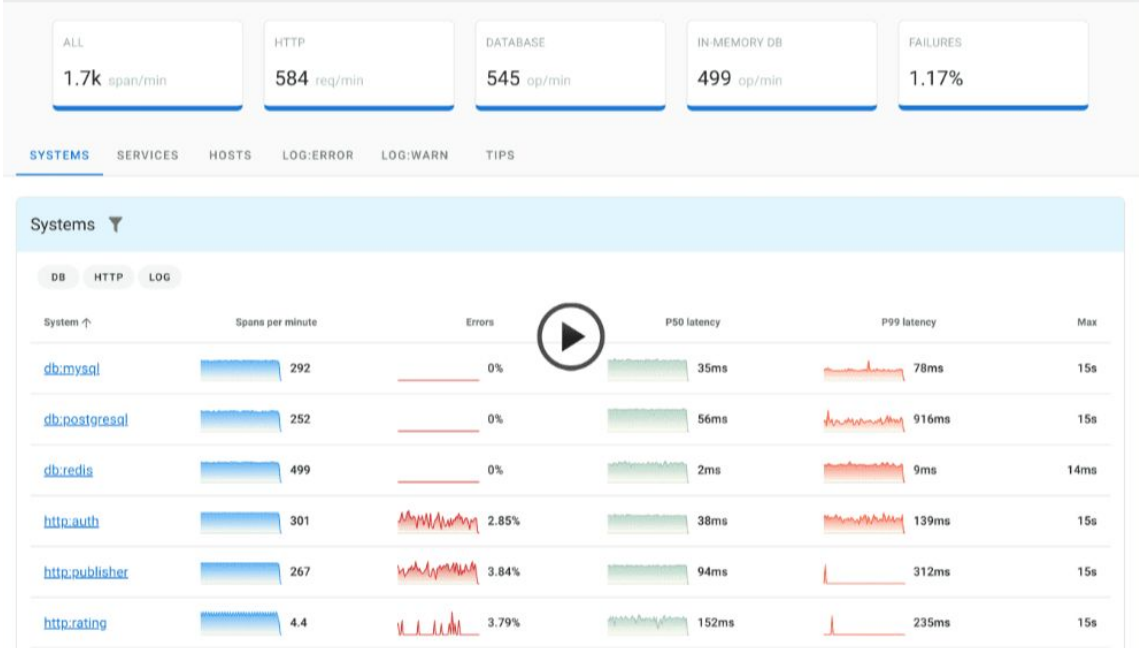
cloki-view Public
cLoki LogQL UI Client
JavaScript ☆ 12 2

clickhouse-mate Public
Advanced Clickhouse "Play" Client and Web User-Interface
TypeScript ☆ 7 1

uptrace

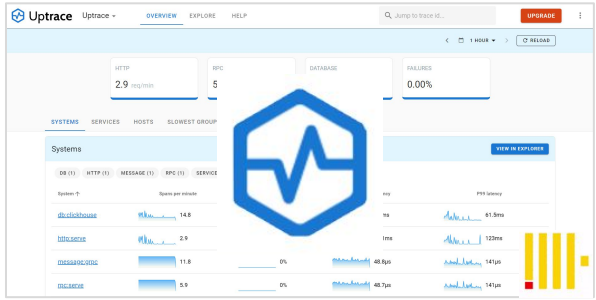
Uptrace

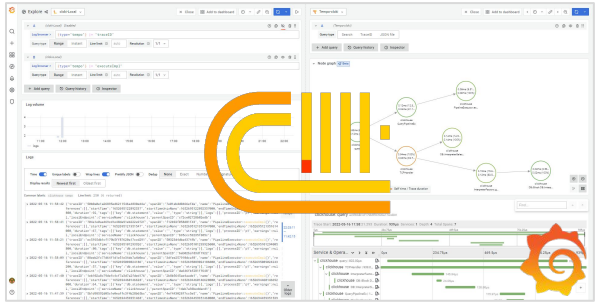
Uptrace is a cost-effective log and tracing solution that helps users monitor, understand, and optimize complex distributed systems faster than ever.



Uptrace

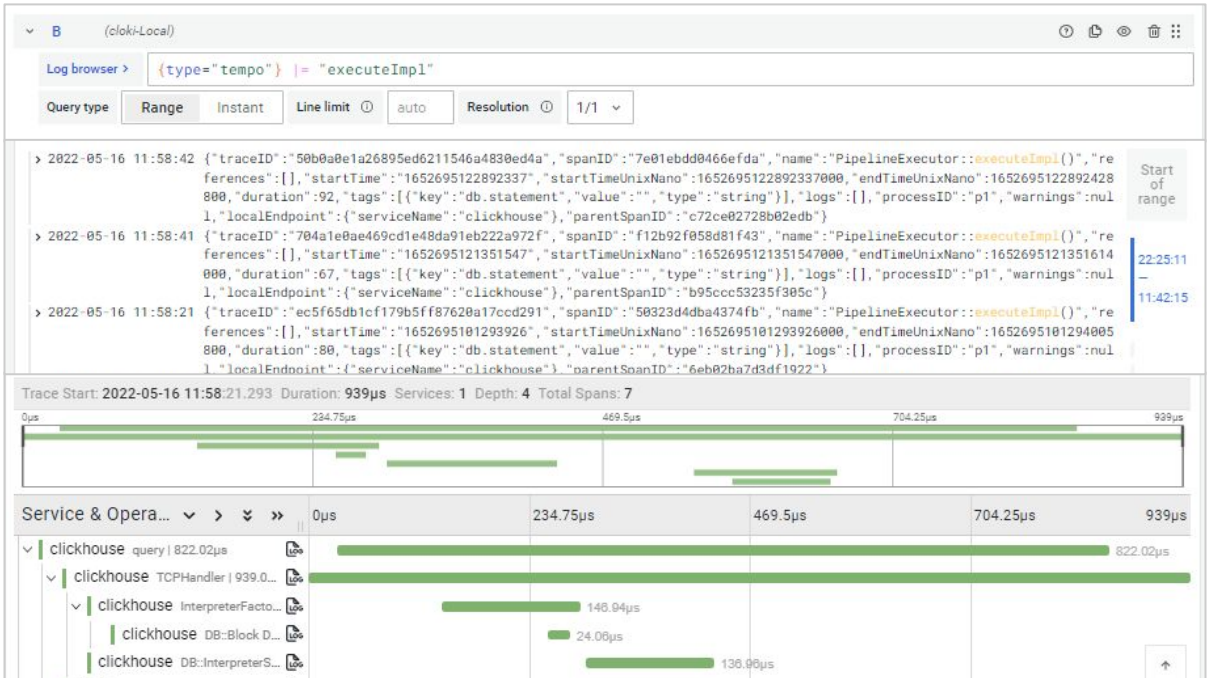
Uptrace allows analysis of logs, metrics and tracing events on a single plane, and natively supports data ingestion using *industry standard* tools and agents





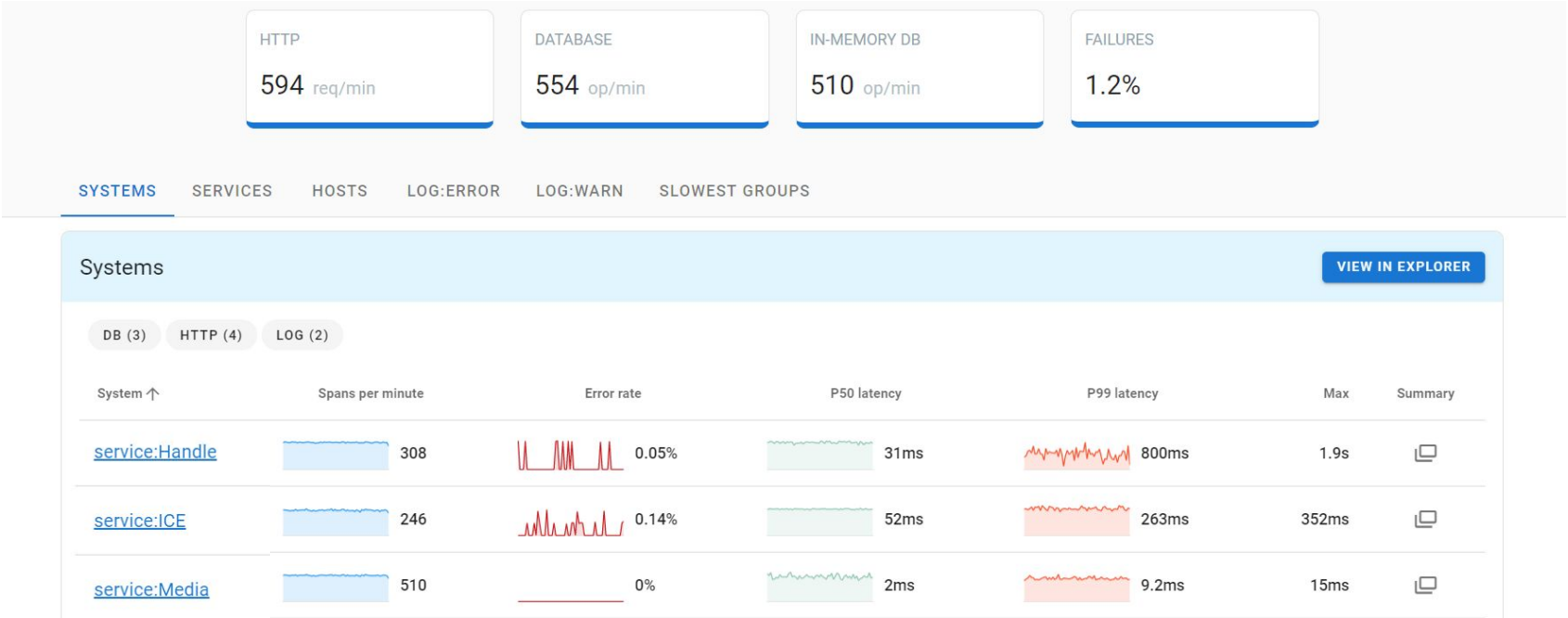
Uptrace

Uptrace natively integrates and supports **Standard APIs** *(logql, promql, tempo)*
providing unprecedented flexibility interfacing with its data from any platform



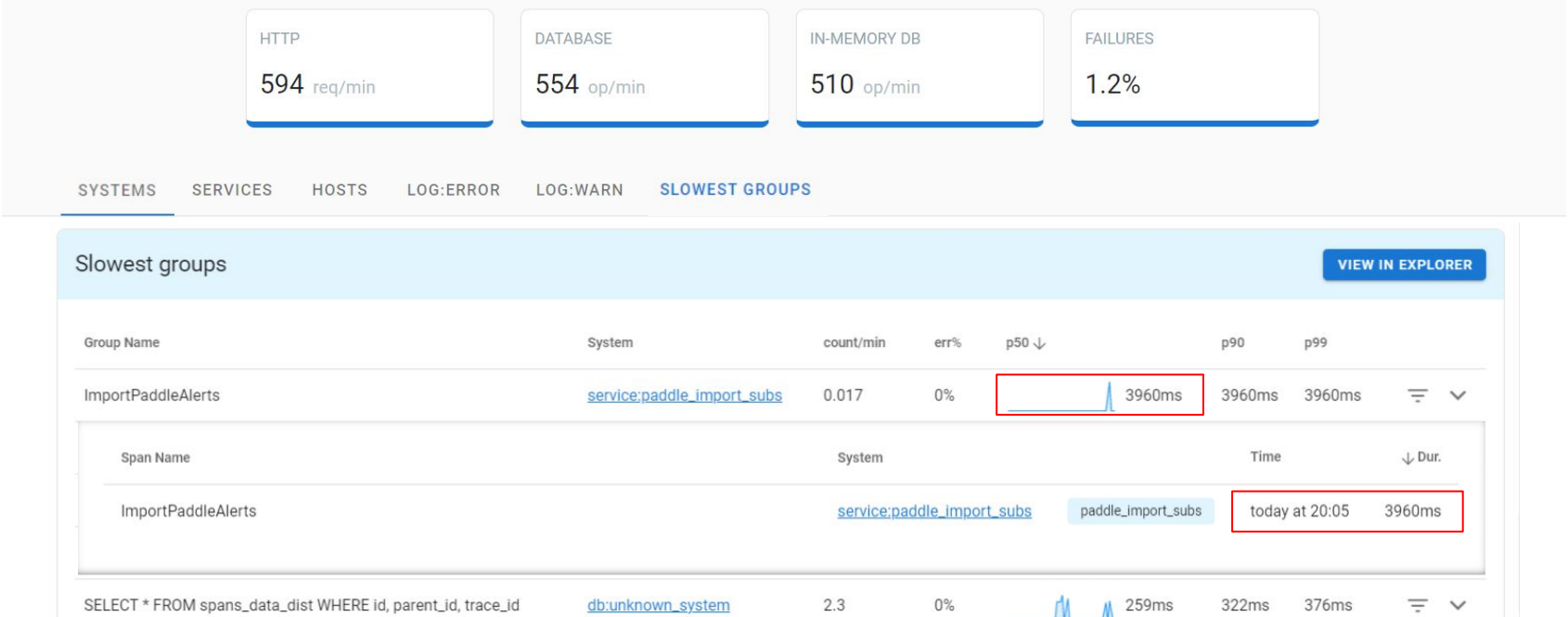
Uptrace

Uptrace can automatically correlate **spans**, **logs** and **events** with precision while retaining relevant event tags based on front or tail sampling techniques



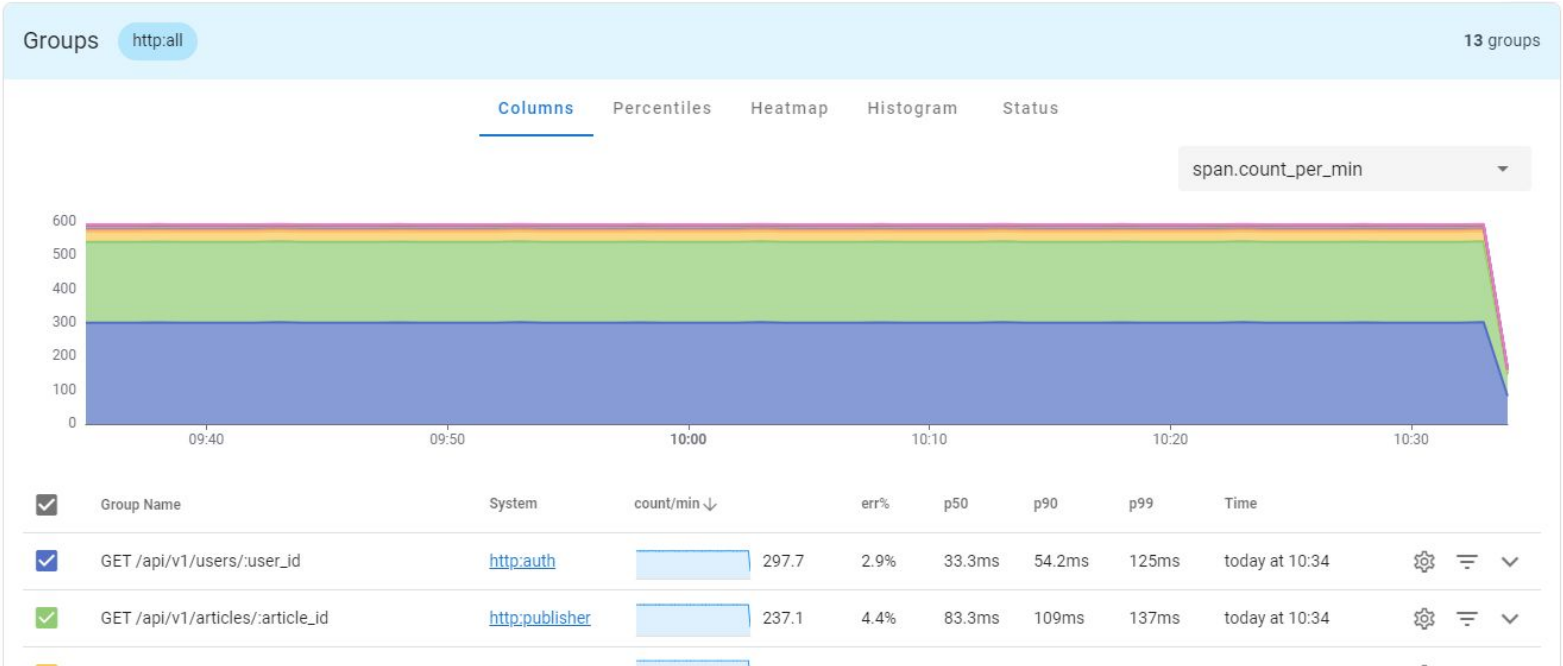
Uptrace

Uptrace does the job right and hunts for interesting and critical information.
"What are the slowest performing services right now?" - Here they are, *Mike*.



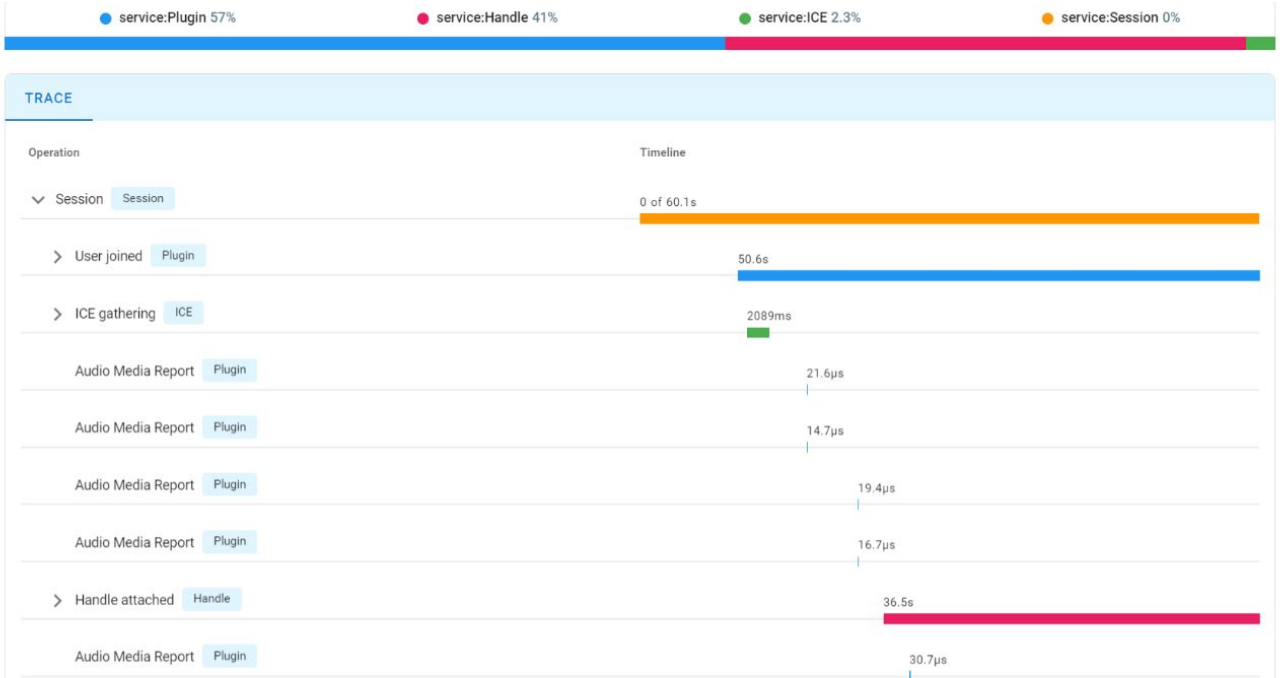
Uptrace

Uptrace can trace and analyze performance of monitored servers and services with great detail at any resolution, outlining performance metrics and issues in **real-time**



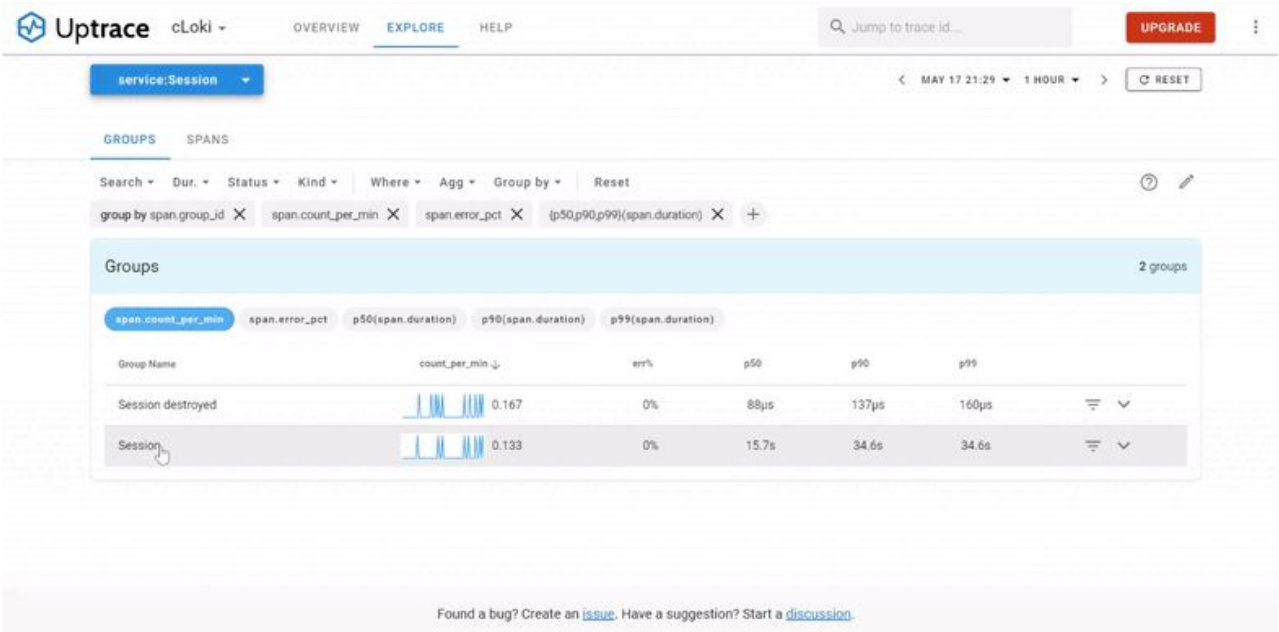
Uptrace

Telemetry spans are designed to reveal every detail about traced sessions with cross-referencing between processes (parent / child) and automated correlation



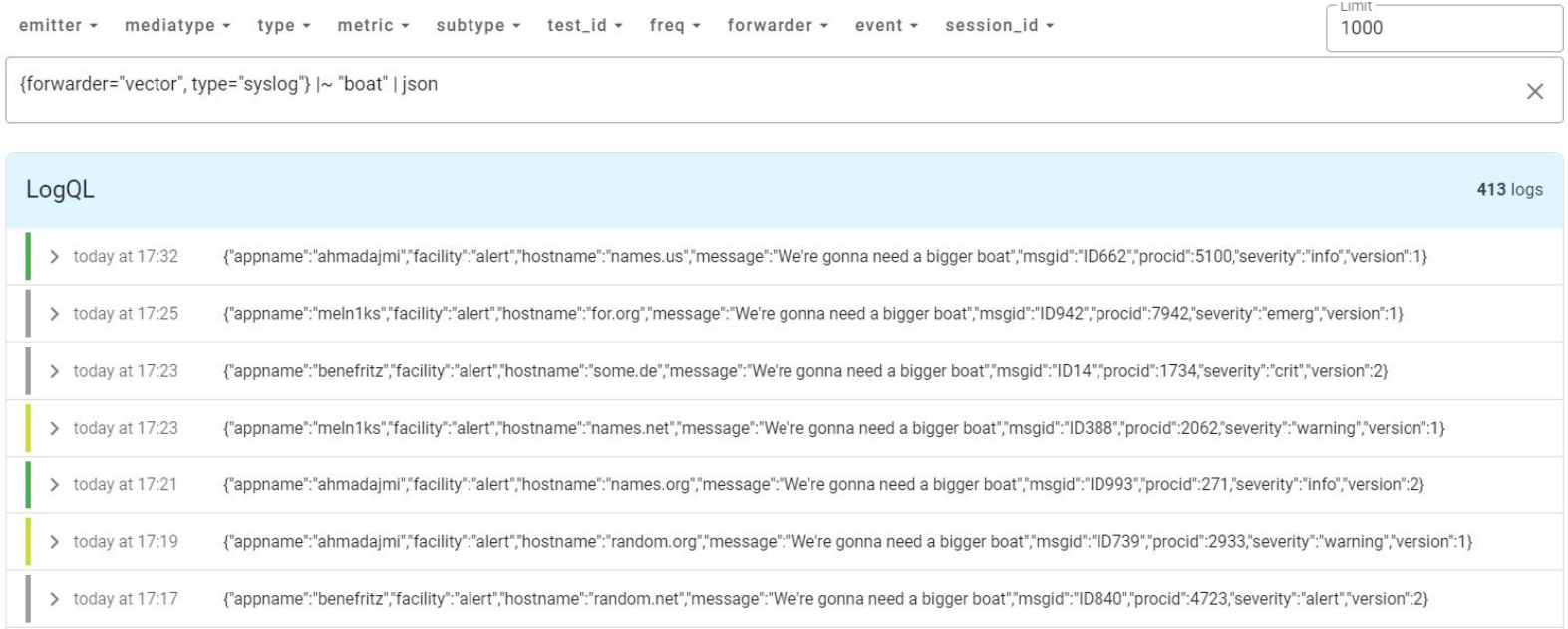
Uptrace

Uptrace workflows allows browsing data and spans at all levels of knowledge and expertise via the built-in user interface with zero training or learning curve



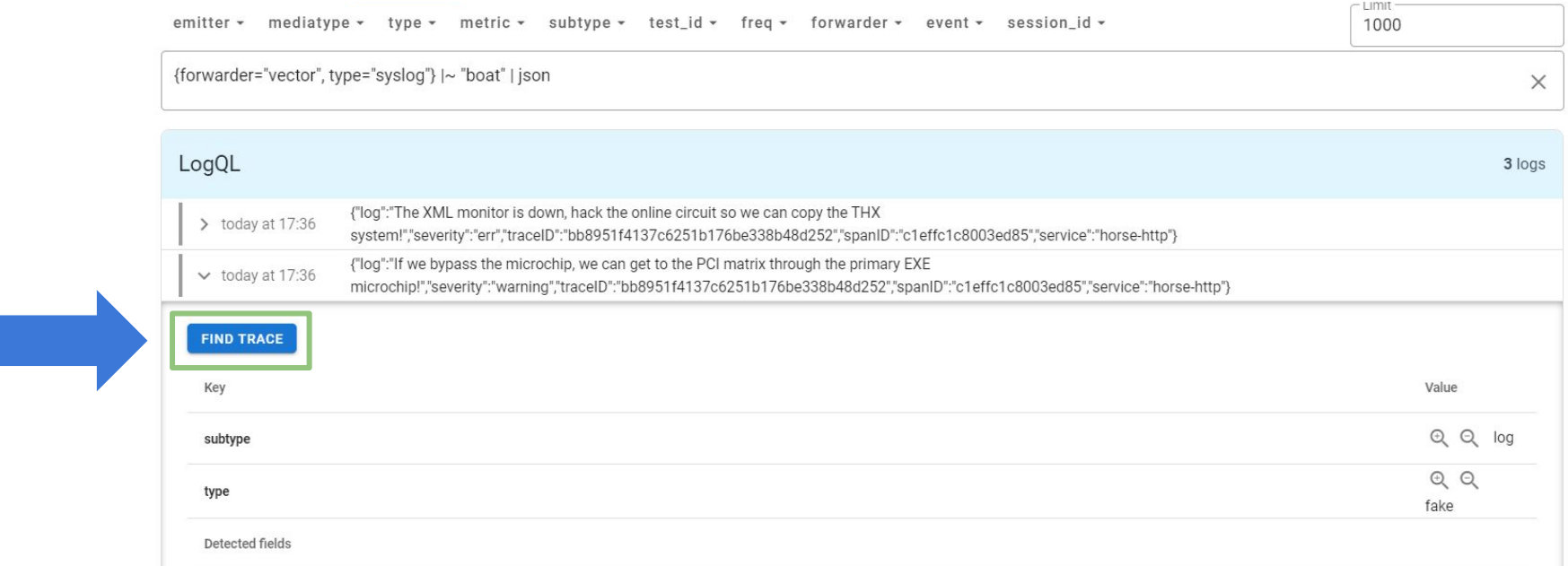
Uptrace

Uptrace now integrates 100% the **cLoki LogQL** functionality natively and can be used to to ingest massive amounts of inserts and *query, filter and extract* meta out of any **log** format.



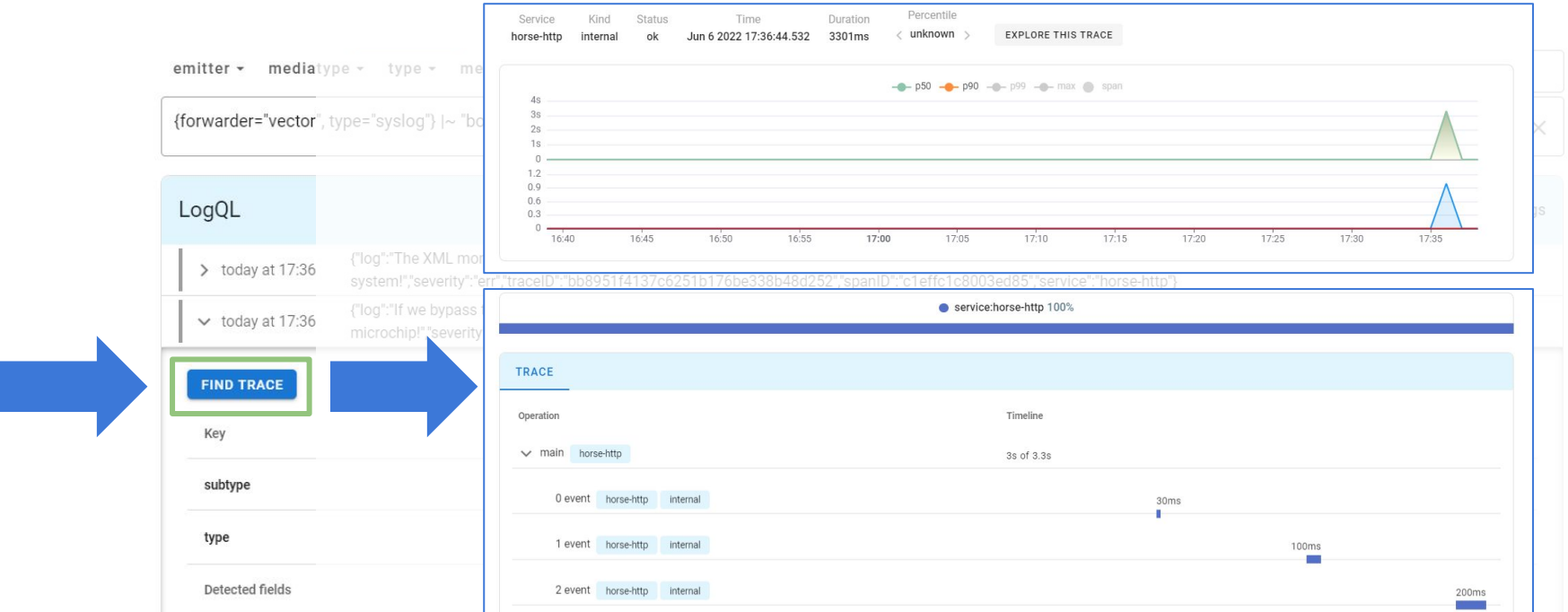
Uptrace

On top of the existing functionality **Uptrace** can now also automatically automatically correlate analyzed **Logs** to **Telemetry Spans** and/or scraped **Metrics** in a ***single click***



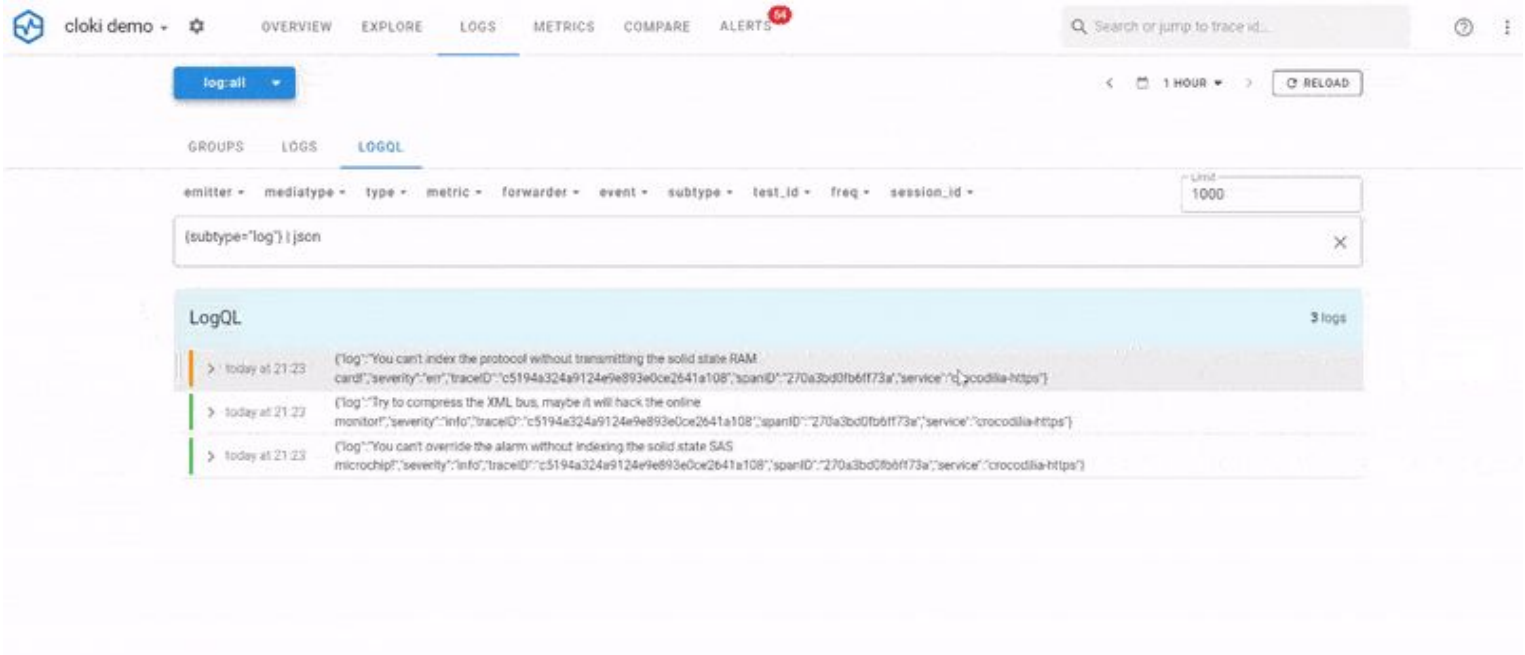
Uptrace

On top of the existing functionality **Uptrace** can now also automatically automatically correlate filtered Logs to Telemetry Spans and/or Collected Metrics in a ***single click***



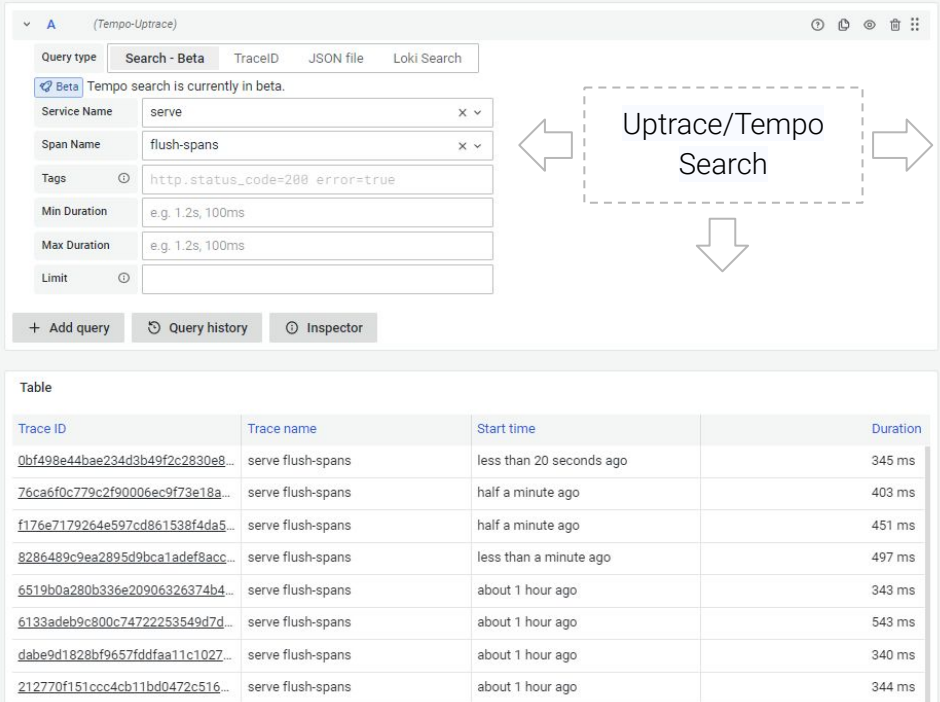
Uptrace

On top of the existing functionality **Uptrace** can now also automatically automatically correlate filtered Logs to Telemetry Spans and/or Collected Metrics in a ***single click***

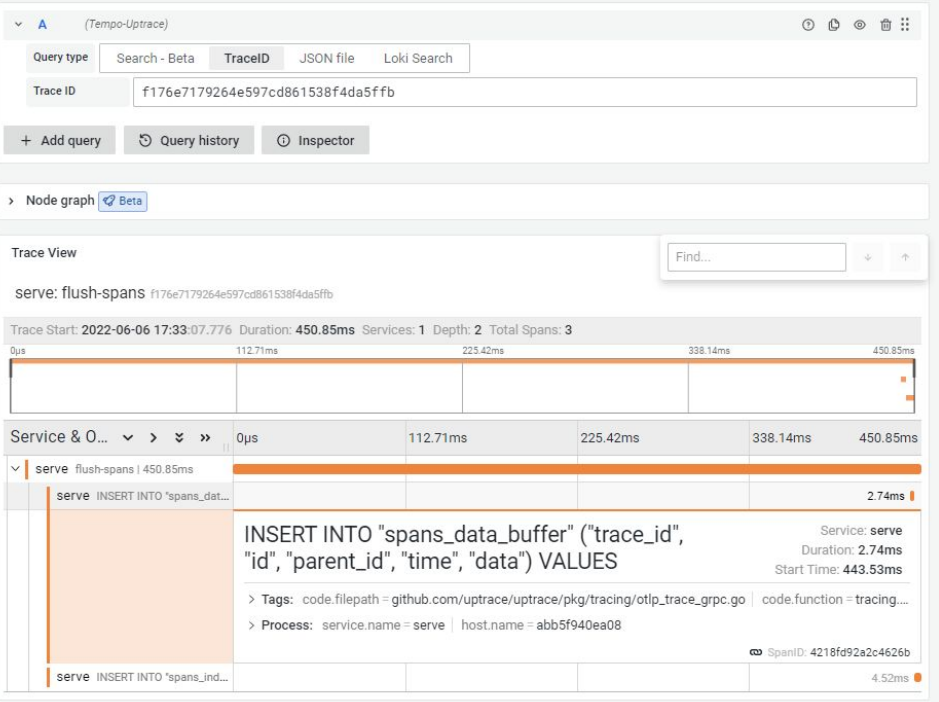


Uptrace

There's more! How about searching uptrace through **Grafana** using **LogQL + Tempo** APIs?

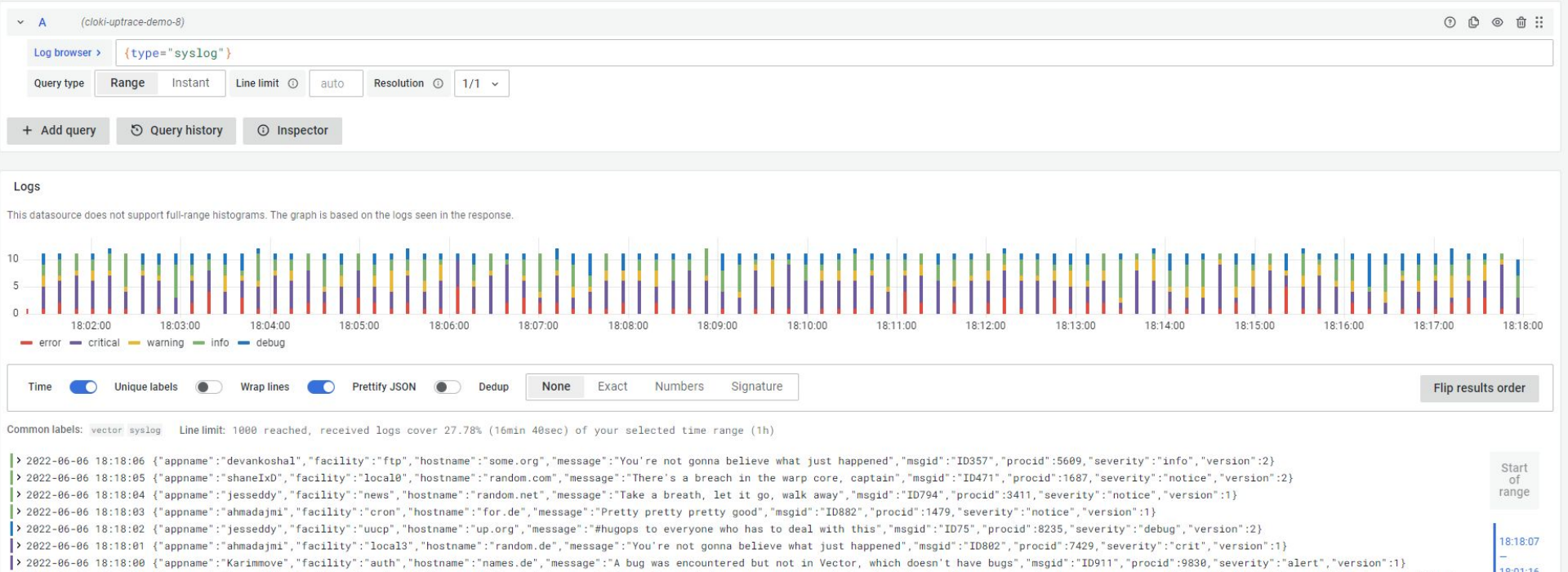


Uptrace/Tempo Search



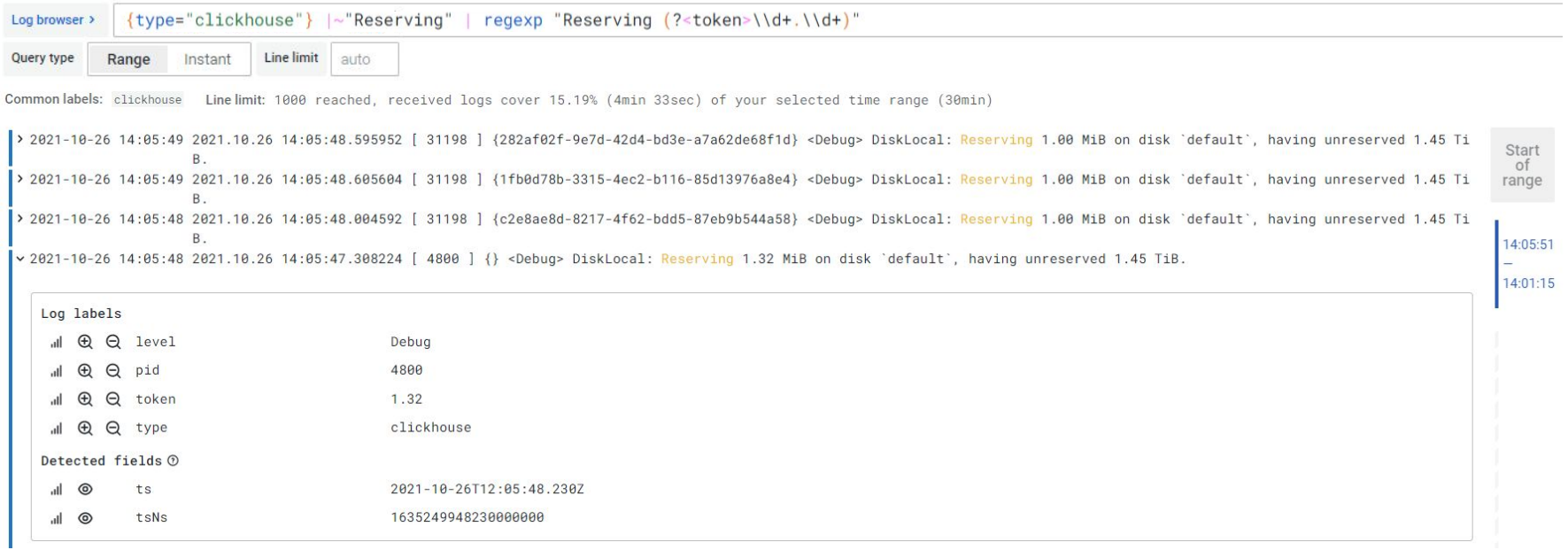
Uptrace

You can access your Telemetry, Logs and Metrics without having to install any **Grafana** plugin



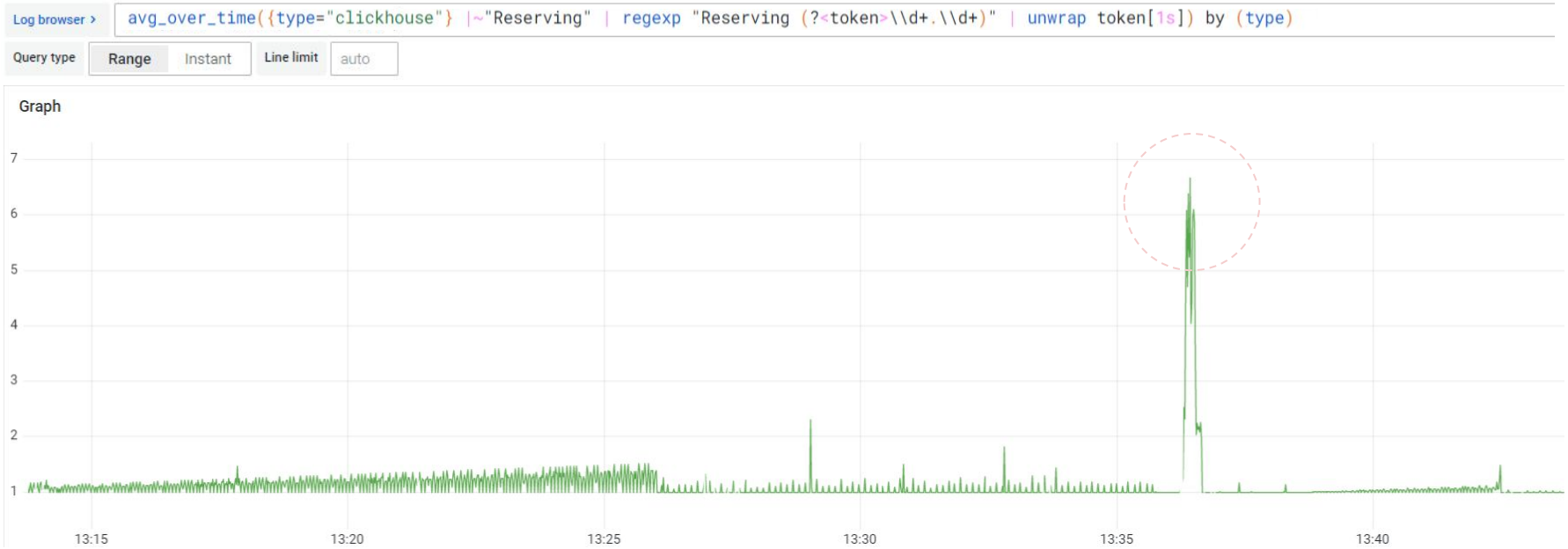
Uptrace

… including our favourite features allowing us to extract metrics from just about *anywhere*

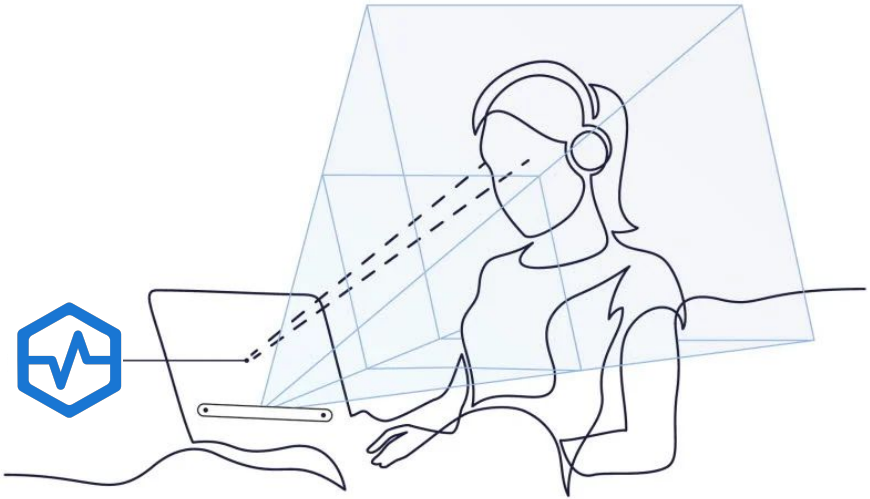


Uptrace

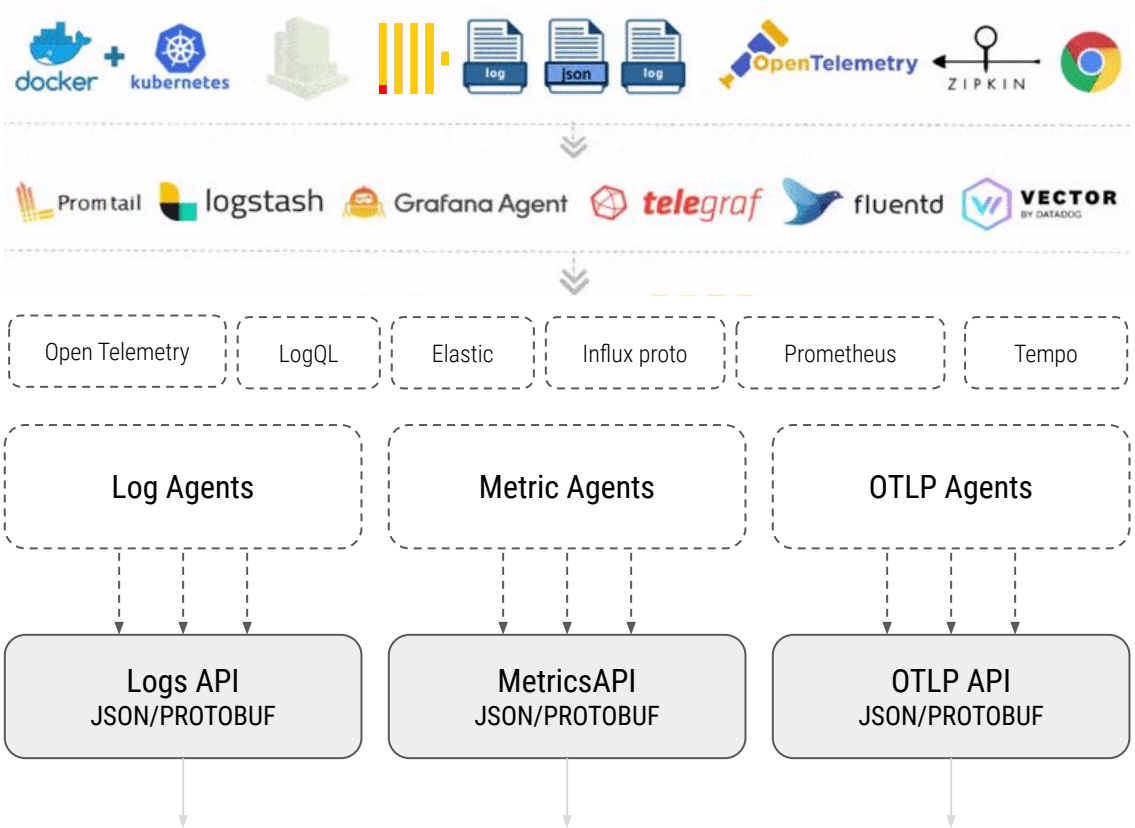
“How much disk memory was ClickHouse reserving between 13:13 and 13:40?” - **boom, done!**



High Level Design



Uptrace: Agents & APIs



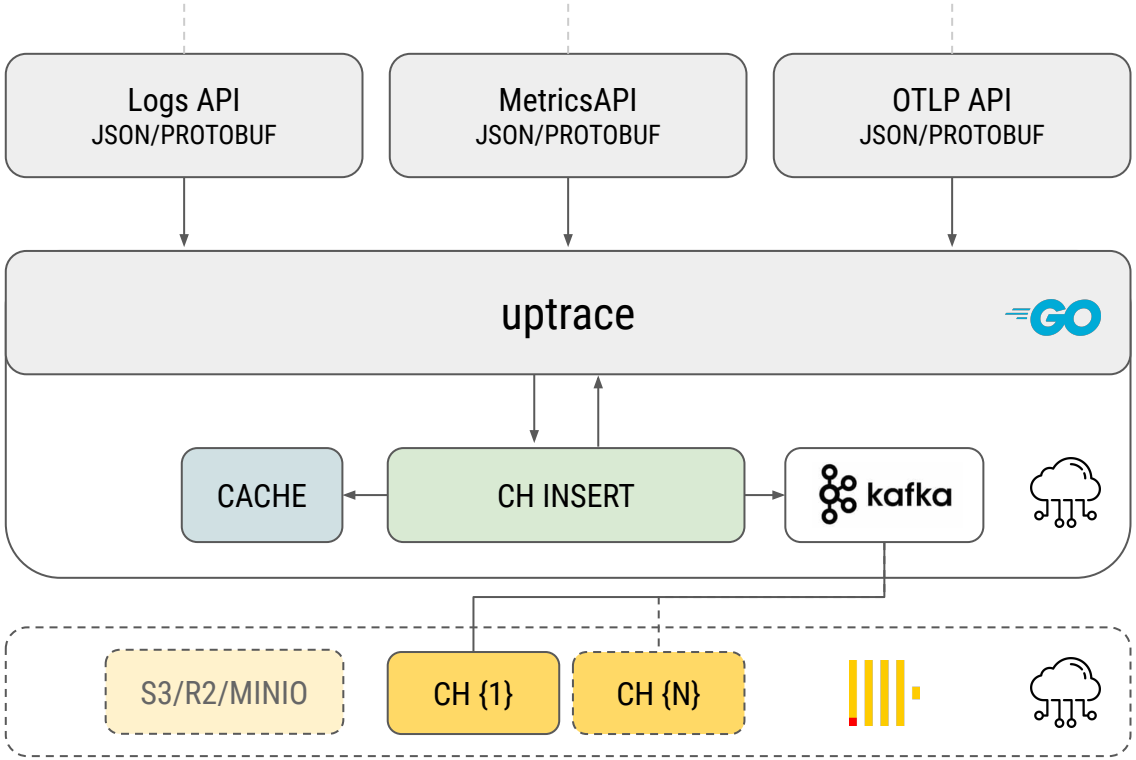
Industry Standard Agents

API endpoints

Transparent inputs

Uptrace Cloud APIs

Uptrace: Insert Pipeline



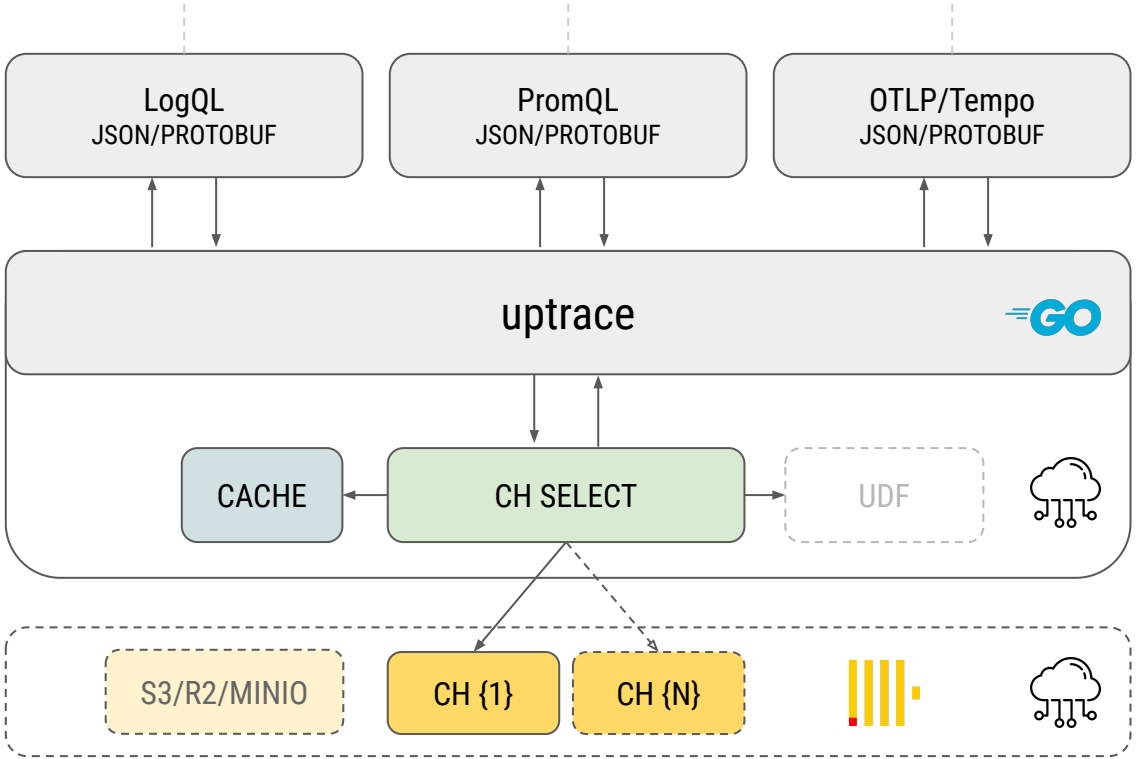
Uptrace Cloud APIs

Uptrace Cloud / On Premise

Distributed Insert Queues

Bring Your Own ClickHouse

Uptrace: Query Pipeline



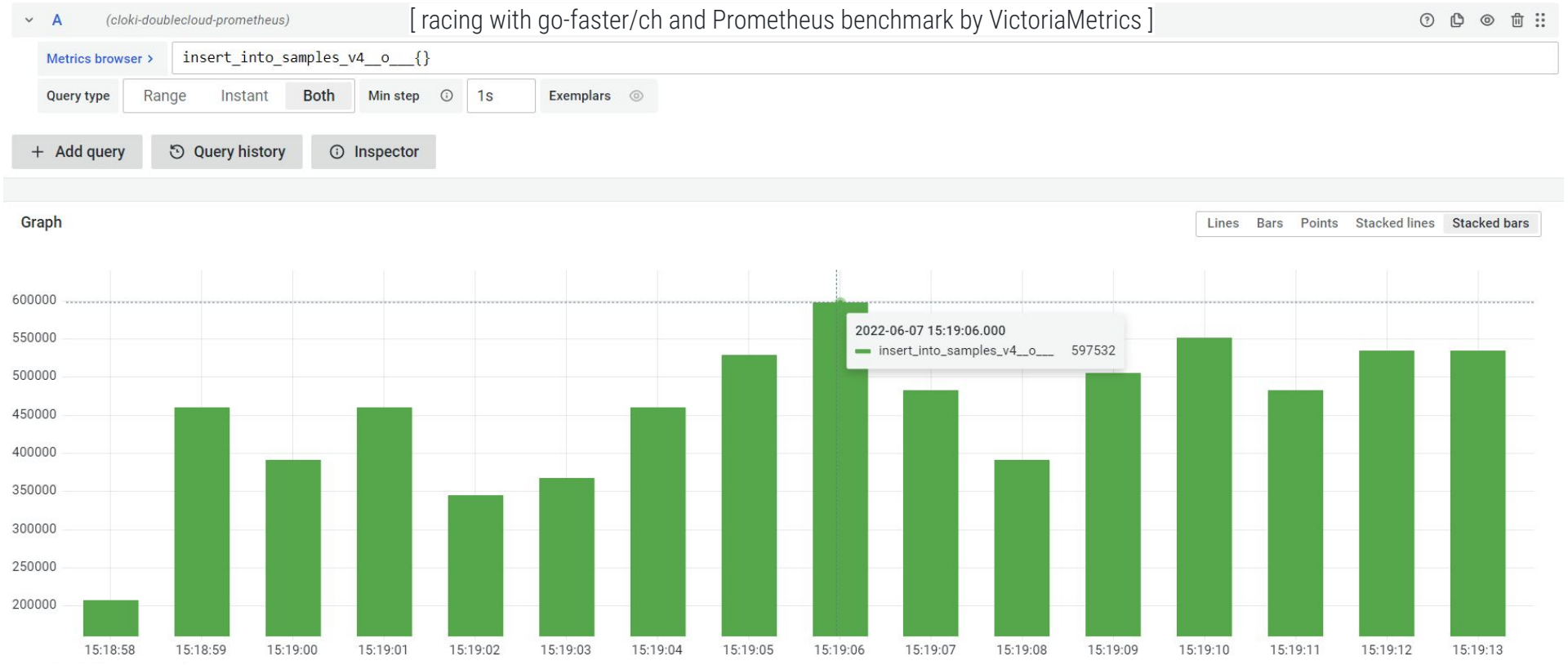
Uptrace Cloud APIs

Uptrace Cloud / On Premise

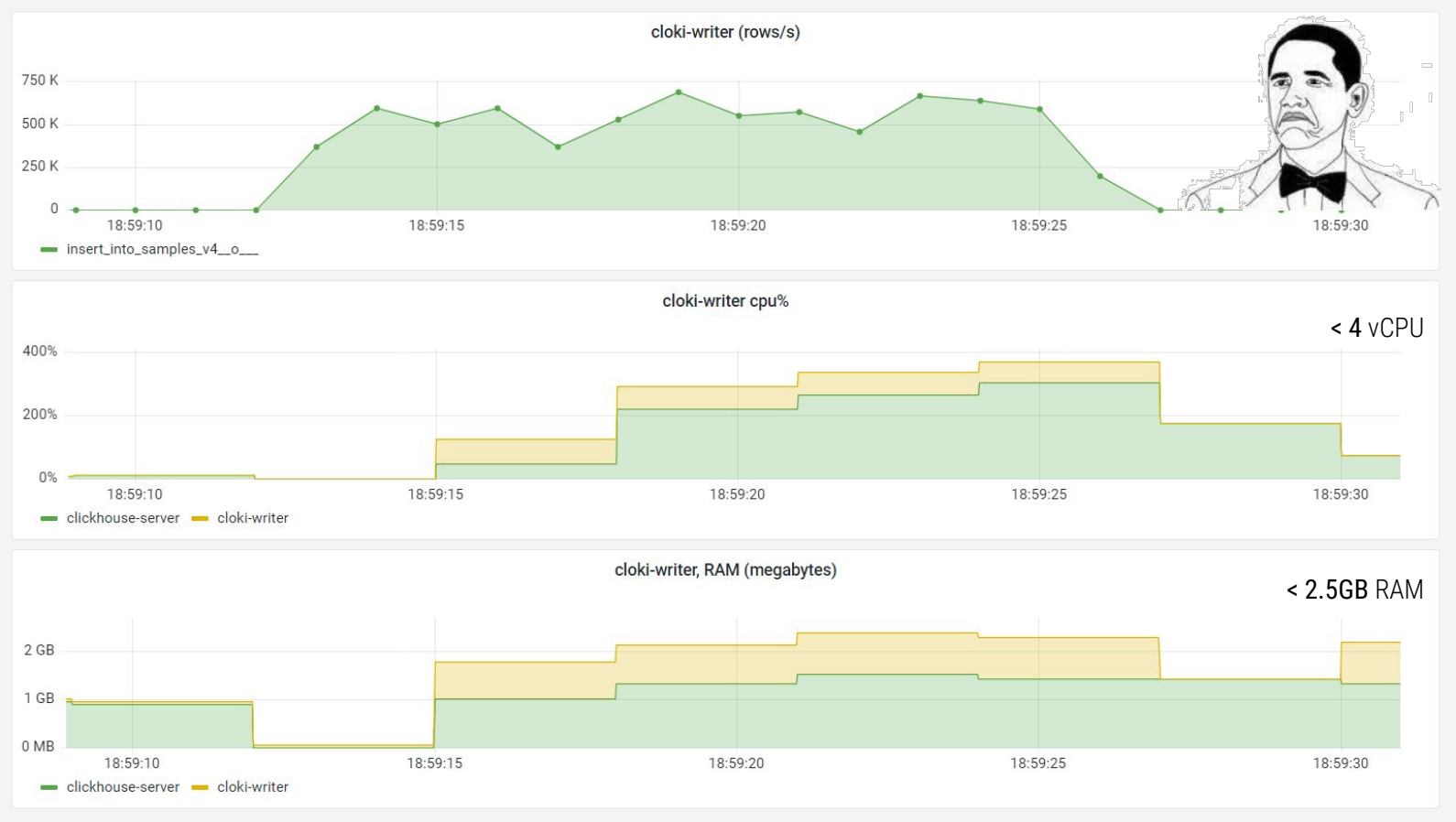
ClickHouse Query Transpiler

Bring Your Own ClickHouse

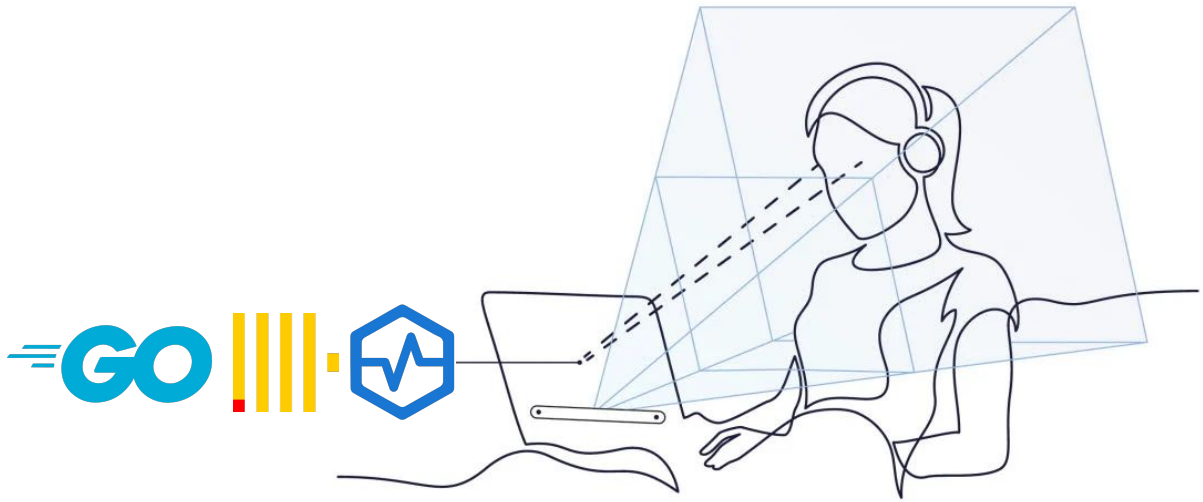
ClickHouse **INSERT** performance up to ~500k samples/thread/second

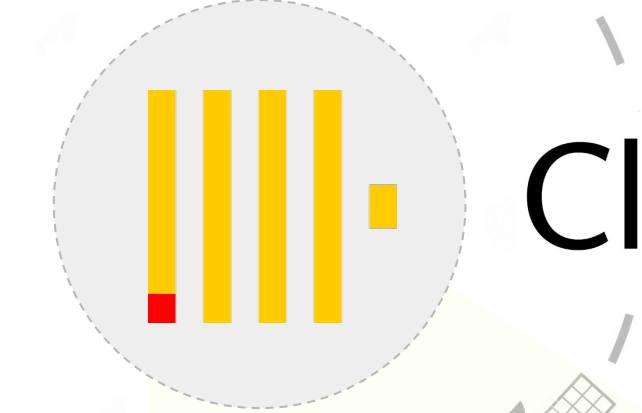


ClickHouse **INSERT** performance up to ~500k samples/thread/second



Under the Hood





ClickHouse

ClickHouse

Favourite ClickHouse Features @uptrace

Our team is very passionate about ClickHouse and we spend *lots* of time discussing techniques and performance, living and breathing all the material published its vibrant community and by awesome companies such as Altinity

Here's a brief list of the features we love to use everyday:

- Data Sampling
- Tiered storage and S3
- Codecs & Compression
- Distributed and Merge tables
- Distributed Tracing (MV + URL Engine)
- ClickHouse-Mate

… and last but not least, a brief look at our **ROADMAP** for 2022-23!

ClickHouse Sampling

Get approximated results **fast** by processing a fraction of data

```
CREATE TABLE sampling_test ( id UInt64, group_id UInt64, sum UInt64 )
ENGINE = MergeTree()
ORDER BY (group_id, intHash32(id))
SAMPLE BY intHash32(id)
```

Without Sampling

```
SELECT sum(sum) FROM sampling_test WHERE group_id != 0
┌─────sum(sum)─┐
│ 4950000000 │
└──────────────┘
```

Elapsed: 1.201 sec. Processed 1.09 billion rows, 17.42 GB

With Sampling

```
SELECT sum(sum * _sample_factor) FROM sampling_test SAMPLE 1/10 WHERE group_id != 0
┌─sum(multiply(sum, _sample_factor))─┐
│                         4951544350 │
└────────────────────────────────────┘
```

Elapsed: 0.697 sec. Processed 115.38 million rows, 2.77 GB

Sample key must

- Be a part of the primary key
- Be uniformly distributed (aka random)
- Come after low cardinality columns in the PK or performance will suffer

Syntax

- SAMPLE 1/10 - sample 10% of data
- SAMPLE 1000000 - sample not less than 1M rows
- SAMPLE 1/10 OFFSET 1/10 - sample the 2nd 10% of data

ClickHouse

Tiered Storage and S3

Configure **S3 Disk**

```
<storage_configuration>
  <disks>
    <s3>
      <type>s3</type>
      <endpoint>http://[BUCKET_NAME].s3.amazonaws.com/[NODE_NAME]/</endpoint>
      <access_key_id>[FIXME]</access_key_id>
      <secret_access_key>[FIXME]</secret_access_key>
    </s3>
  </disks>
</storage_configuration>
```

Enable **data cache** on SSD to optimize r/w performance

```
<storage_configuration>
  <disks>
    <s3>
      <type>s3</type>
      <endpoint>http://[BUCKET_NAME].s3.amazonaws.com/[NODE_NAME]/</endpoint>
      <access_key_id>[FIXME]</access_key_id>
      <secret_access_key>[FIXME]</secret_access_key>
      <data_cache_enabled>1</data_cache_enabled>
      <data_cache_max_size>8589934592</data_cache_max_size>
    </s3>
  </disks>
</storage_configuration>
```

Relocate data from **SSD** to **S3** when free space **< 5%**

```
<policies>
  <tiered>
    <move_factor>0.05</move_factor>
    <volumes>
      <hot>
        <disk>ssd</disk>
      </hot>
      <s3>
        <disk>s3</disk>
        <prefer_not_to_merge>true</prefer_not_to_merge>
      </s3>
    </volumes>
  </tiered>
</policies>
```

Manually relocate data from/to **S3** or **SSD** storage

```
-- upload to s3
ALTER TABLE table1 MOVE PARTITION '20220102' TO VOLUME 's3';

-- download from s3 if you changed your mind
ALTER TABLE table1 MOVE PARTITION '20220102' TO VOLUME 'ssd';
```

ClickHouse Codecs & Compression

ClickHouse **codecs** are a *must-have* to efficiently encode numeric values at scale:

- **DoubleDelta** for monotonic numbers
- **Delta** for mostly increasing/decreasing numbers
- **Gorilla** for floats
- **T64** for random-like numbers
- **None** don't use a codec if numbers are truly randomized

ClickHouse should use **LZ4 compression** for small partitions and **ZSTD** for larger ones:

```
<compression incl="clickhouse_compression">
  <!-- Use ZSTD(3) for parts >= 64MB -->
  <case>
    <method>zstd</method>
    <level>3</level>
    <min_part_size>67108864</min_part_size>
    <min_part_size_ratio>0.001</min_part_size_ratio>
  </case>

  <!-- Add more cases for ZSTD(5) etc -->
</compression>
```

- The average uncompressed span size is ~1KB
- ClickHouse can compress that data down to 40 bytes (25x)
- To get best results, use codecs and ZSTD compression levels

Use Default codec to specify the default compression method, for example: `Codec(Delta, Default).`

ClickHouse

ClickHouse Table Engines

Distributed Table Engine

You can have chunks of data on different servers and "join" them together using a distributed table engine:

```
CREATE TABLE table1_dist AS table1
ENGINE = Distributed(cluster1, currentDatabase(), table1)
```

Selecting from **table1_dist** will query all servers defined in the **cluster1**

Merge Table Engine

Merge tables allows reading from multiple local tables simultaneously.

```
CREATE TABLE table_merge AS table1
ENGINE = Merge(currentDatabase(), '^table1|table2|table3$');
```

Super useful for updating your schema without copying old data around

ClickHouse GIN Indexing

Fast filtering JSON data by key-value pairs during fingerprint selection

```
CREATE TABLE IF NOT EXISTS time_series_v2 (
        date Date,
        fingerprint UInt64, //the same ID
        labels String, //we know about JSONB, it's under our research for now :)
) ENGINE = ReplacingMergeTree(date) PARTITION BY (date) ORDER BY (fingerprint);

CREATE MATERIALIZED VIEW IF NOT EXISTS time_series_gin_v2
ENGINE = ReplacingMergeTree()
AS SELECT date, fingerprint, pairs.1 as key, pairs.2 as val,
FROM time_series_v2
ARRAY JOIN JSONExtractKeysAndValues(time_series_v2.labels, 'String') as pair
ORDER BY key, val, fingerprint
```

NO IDX (visitParamExtractString)

```
SELECT * FROM time_series_v2
WHERE (visitParamExtractString(labels, 'key_1_1') = 'val1_1')
  AND (visitParamExtractString(labels, 'key_2_1') = 'val2_1')
```

1 rows in set. Elapsed: 0.072 sec. Processed 3.00 million rows, 214.57 MB (41.91 million rows/s., 3.00 GB/s.)

With IDX

```
SELECT * FROM time_series_v2
WHERE fingerprint IN (
    SELECT fingerprint FROM time_series_gin_v2 ANY INNER JOIN (
        SELECT fingerprint FROM time_series_gin_v2 WHERE (key = 'key_2_1') AND (val = 'val2_1')
    ) AS idx_2
    ON time_series_gin_v2.fingerprint = idx_2.fingerprint
    PREWHERE (key = 'key_1_1') AND (val = 'val1_1')
)
```

1 rows in set. Elapsed: 0.024 sec. Processed 41.26 thousand rows, 1.18 MB (1.74 million rows/s., 49.56 MB/s.)

ClickHouse Distributed Tracing

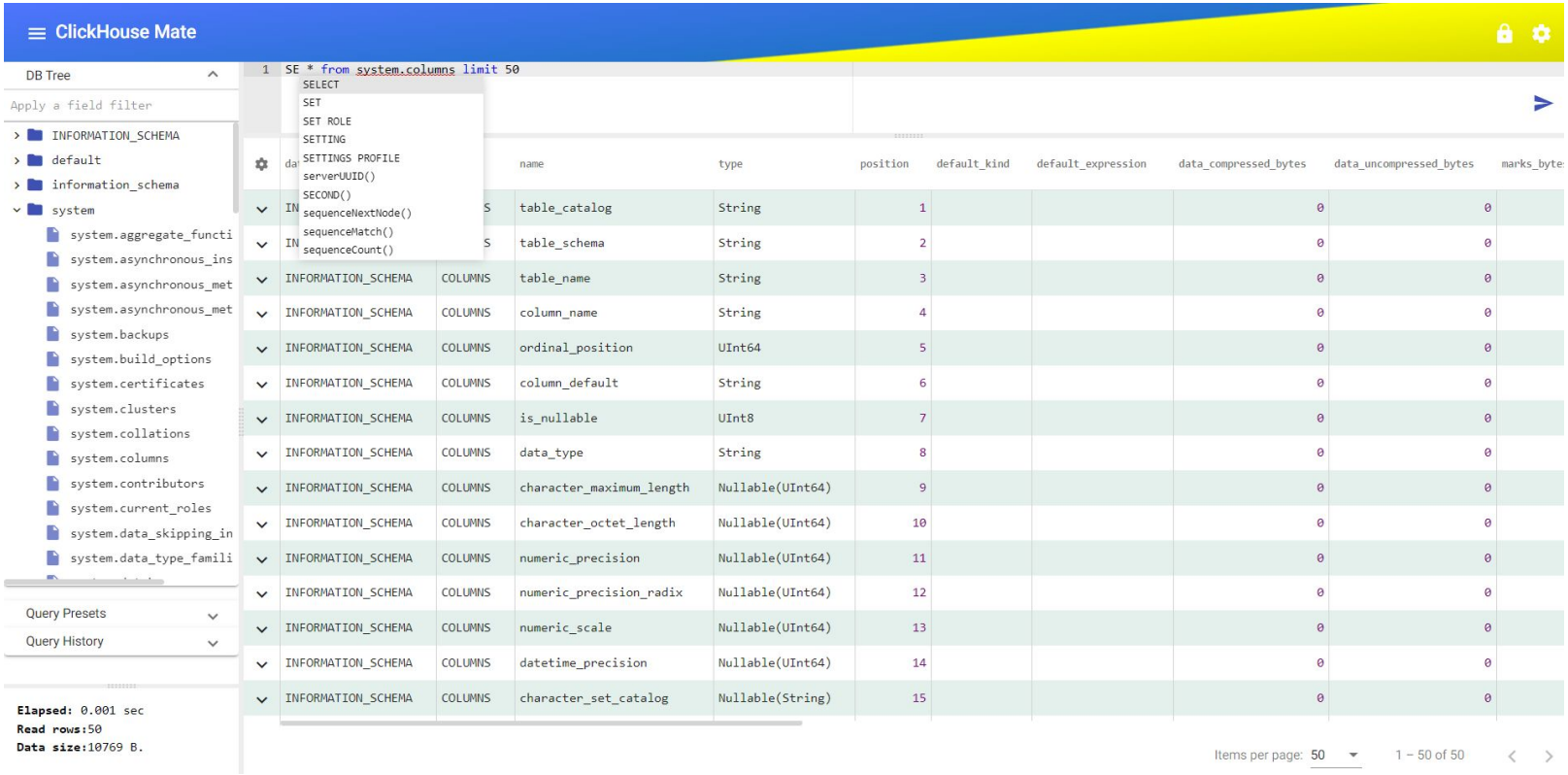
Send ClickHouse spans from the *opentelemetry_span_log* table to **Uptrace API** using **Materializev View** and **URL Engine**

```
CREATE MATERIALIZED VIEW default.zipkin_spans
ENGINE = URL('https://api.uptrace.dev/api/v2/spans', 'JSONEachRow')
SETTINGS output_format_json_named_tuples_as_objects = 1,
    output_format_json_array_of_rows = 1 AS
SELECT
    lower(hex(trace_id)) AS traceId,
    case when parent_span_id = 0 then '' else lower(hex(parent_span_id)) end AS parentId,
    lower(hex(span_id)) AS id,
    operation_name AS name,
    start_time_us AS timestamp,
    finish_time_us - start_time_us AS duration,
    cast(tuple('clickhouse'), 'Tuple(serviceName text)') AS localEndpoint,
    attribute AS tags
FROM system.opentelemetry_span_log
```

ClickHouse Mate

https://metrico.github.io/clickhouse-mate

Running queries all day? Try out *ClickHouse-Mate* the first *"play" webclient* written under bomb attacks by our brave **Ukranian** team



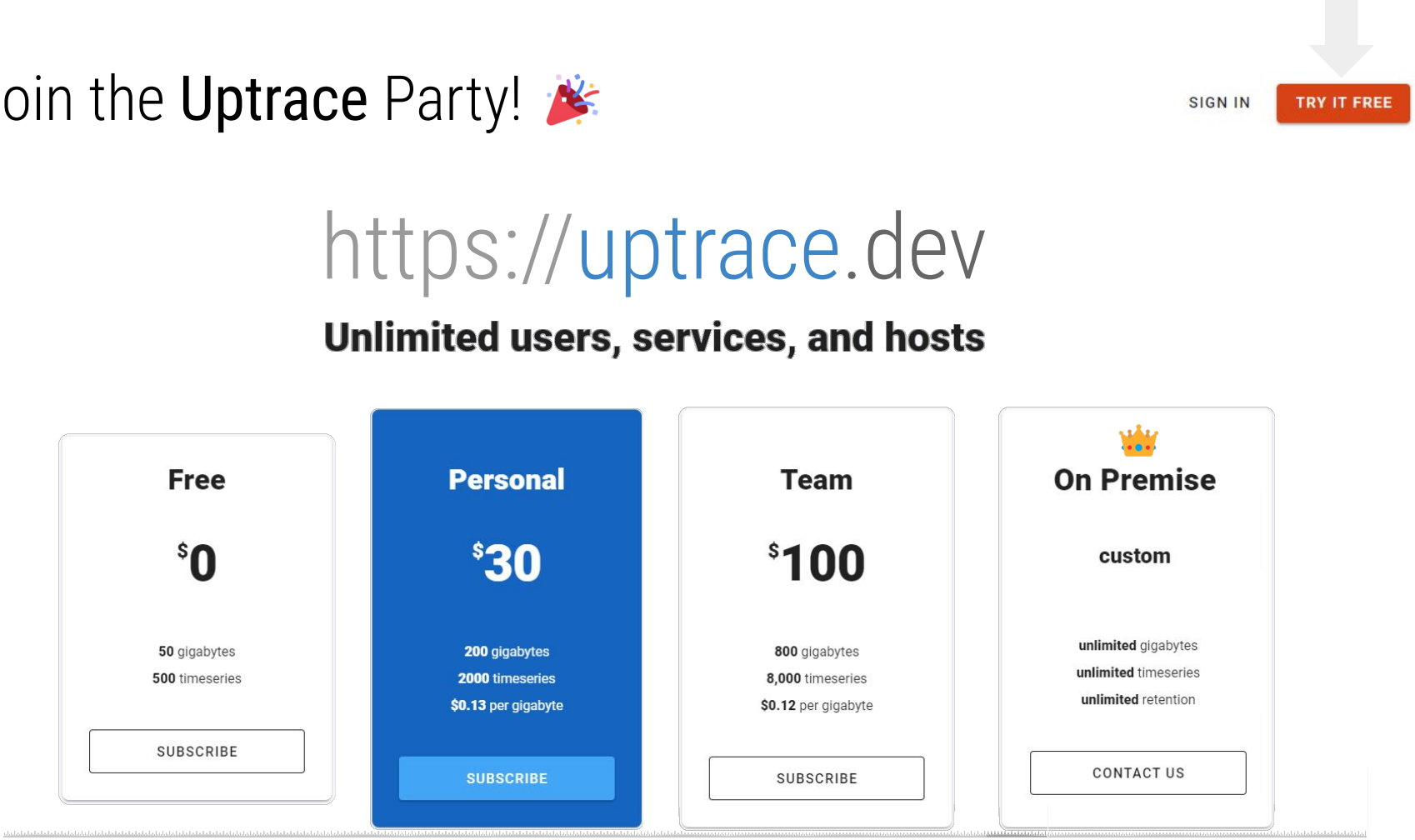
ClickHouse

Roadmap

- Multi-Tenancy (done)
- PromQL API support for OpenTelemetry Mertrics (in progress)
- JSON datatype option for semi structured data (in progress)
- Additional Input formats (what would you like to see?)
- Extend AlertManager API emulator
- External CH pull agent w/ custom Exporting and pseudo URL Engine
- Extended Provisioning APIs for Uptrace Cloud


IT'S TIME TO STOP

Join the **Uptrace** Party!



That's all Folks!

JOIN US

https://uptrace.dev ★ https://docs.uptrace.dev ★ https://uptrace.cloud/matrix

ClickHouse

uptrace

QKIP

Demo Links

Uptrace Demo

Performance Test 200k/s

LogQL Grafana Syslog (demo)

Tempo Grafana Uptrace (demo)

Zipkin Test Sender (demo)

ClickHouse Mate (demo)

Clickhouse VS GIN index

Task:

- Lots of plain JSON objects
- We want to filter by key:value pairs fast

In PostgreSQL it would have been done like that:

```
CREATE TABLE jsons (
  fingreprint int8, //like ID
  obj JSONB
);
CREATE INDEX jsons_gin_idx ON jsons USINGgin (obj);
SELECT * FROM jsons WHERE obj @> '{"key1":"val1", "key2":"val2"}';
```

Clickhouse VS GIN index (2)

In Clickhouse it transformed into the next:

```
CREATE TABLE IF NOT EXISTS time_series_v2 (
      date Date,
      fingerprint UInt64, //the same ID
      labels String, //we know about JSONB, it’s under our research for now :)
) ENGINE = ReplacingMergeTree(date) PARTITION BY (date) ORDER BY (fingerprint);

CREATE MATERIALIZED VIEW IF NOT EXISTS time_series_gin_v2
  ENGINE = ReplacingMergeTree()
  AS SELECT date, fingerprint, pairs.1 as key, pairs.2 as val,
  FROM time_series_v2
    ARRAY JOIN JSONExtractKeysAndValues(time_series_v2.labels, 'String') as pair
  ORDER BY key, val, fingerprint
```

After some research the fastest lookup request is:

```
SELECT * FROM time_series_v2 WHERE fingerprint IN (
  SELECT fingerprint FROM time_series_gin_v2 INNER ANY JOIN (
      SELECT fingerprint FROM time_series_gin_v2 WHERE key=’key2’ AND val=’val2’
  ) as idx_2 ON time_series_gin_v2.fingerprint = idx_2.fingerprint
  PREWHERE key=’key1’ AND val=’val1’
)
```

Clickhouse VS GIN index: comparison

Insert sample data:

```
INSERT INTO time_series_v2 (date, fingerprint, labels) SELECT
      toDate(NOW()) AS date,
      number AS fingerprint,
      format('{{"key_1_{0}":"val1_{0}", "key_2_{0}":"val2_{0}"}}', toString(number)) AS labels
FROM numbers(3000000)
```

NO IDX (visitParamExtractString):

```
SELECT * FROM time_series_v2
WHERE (visitParamExtractString(labels, 'key_1_1') = 'val1_1') AND (visitParamExtractString(labels, 'key_2_1')
= 'val2_1')
1 rows in set. Elapsed: 0.072 sec. Processed 3.00 million rows, 214.57 MB (41.91 million rows/s., 3.00 GB/s.)
```

With IDX:

```
SELECT * FROM time_series_v2
WHERE fingerprint IN (
    SELECT fingerprint FROM time_series_gin_v2 ANY INNER JOIN
      (SELECT fingerprint FROM time_series_gin_v2 WHERE (key = 'key_2_1') AND (val = 'val2_1') ) AS idx_2
    ON time_series_gin_v2.fingerprint = idx_2.fingerprint
    PREWHERE (key = 'key_1_1') AND (val = 'val1_1')
)
1 rows in set. Elapsed: 0.024 sec. Processed 41.26 thousand rows, 1.18 MB (1.74 million rows/s., 49.56 MB/s.)
```