

Modern Observability with OpenTelemetry in C++

SREday 2025



Jose Gomez-Selles

- ★ Cloud Product Lead at VictoriaMetrics 
- ★ Associate Professor at Complutense Univ. Madrid
- ★ From Madrid, Spain 
- ★ Physicist, PhD in Materials Engineering
- ★ Cloud Native, Telco, Observability (Jaeger, OTel, Kepler)
- ★ Programming (C++), Metal & SimRacing



 github.com/jgomezselles

 [@jgomezselles.bsky.social](https://bsky.social/@jgomezselles)

 linkedin.com/in/joseluisgomezselles



"We need more mainstream talks about C++"
(...or something like that)

- Bjarne Stroustrup, C++ creator
Madrid, April 2025



Jose Gómez-Sellés • You
Product Lead - VictoriaMetrics Cloud
2d • 🌐

Two of my favorite topics: [OpenTelemetry](#) and C++ together! See you at [SREday](#) very soon!



SREday
4,336 followers
2d • 🌐

Join [Jose Gómez-Sellés](#) from [VictoriaMetrics](#), who presents how to turn simple logs into metrics and distributed traces using OpenTelemetry, making observability accessible for real-world systems.

📍 SREday Cologne Q4

📍 Ilt, Germany, October 28, 2025

🔗 RSVP: <https://lnkd.in/d4szw9MU>

🎫 Great sessions, tons of networking, and 10% off with code CLG10!

SRE DAY

Site Reliability, DevOps and Cloud
In-person conference

Jose Gomez-Selles
VictoriaMetrics

Modern Observability
with OpenTelemetry
in C++

October 28 2025
Ilt.com, Cologne, Germany
Tickets - sreday.com

10% OFF
CLG10





Jose Gómez-Sellés • You
Product Lead - VictoriaMetrics Cloud
2d • 🌐

Two of my favorite topics: [OpenTelemetry](#) and C++ together! See you at [SREday](#) very soon!



SREday
4,336 followers
2d • 🌐

Join [Jose Gómez-Sellés](#) from [VictoriaMetrics](#), who presents how to turn simple logs into metrics and distributed traces using OpenTelemetry, making observability accessible for real-world systems.

📍 SREday Cologne Q4

📍 Ilert, Germany, October 28, 2025

🔗 RSVP: <https://lnkd.in/d4szw9MUJ>



Zhu Jiekun • 1st
SWE @ VictoriaMetrics

2d ...

No way. C++ is the most expensive programming language. I bought a dozen books two years ago and they're still brand new. Learning nothing has already cost me a couple of dollars.

Funny • 🤔 1 | Reply • 1 reply

Modern Observability
with OpenTelemetry
in C++



October 28 2025
[Ilert.com](#), Cologne, Germany
Tickets - [sreday.com](#)

🌐 Aliaksandr Valialkin and 26 others

2 comments • 1 repost

Days 1 - 10
Teach yourself variables, constants, arrays, strings, expressions, statements, functions,....



Days 11 - 21
Teach yourself program flow, pointers, references, classes, objects, inheritance, polymorphism,



Days 22 - 697
Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.



Days 698 - 3648
Interact with other programmers. Work on programming projects together. Learn from them.



Days 3649 - 7781
Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.



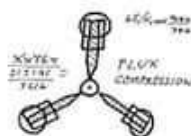
Days 7782 - 14611
Teach yourself biochemistry, molecular biology, genetics,...



Day 14611
Use knowledge of biology to make an age-reversing potion.



Day 14611
Use knowledge of physics to build flux capacitor and go back in time to day 21.



Day 21
Replace younger self.



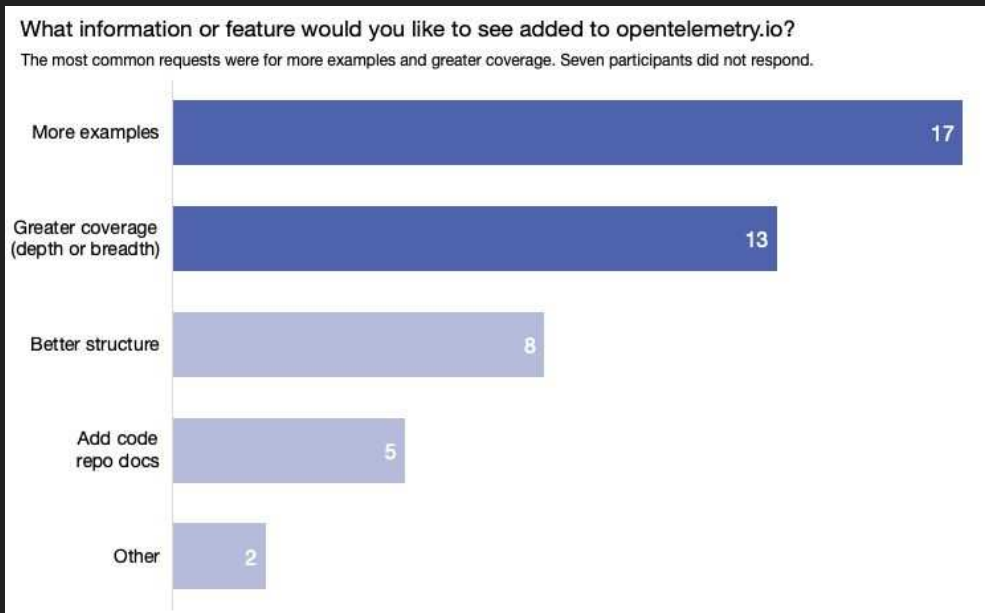
As far as I know, this
is the easiest way to
"Teach Yourself C++ in 21 Days".

OpenTelemetry



Insights from the OpenTelemetry Docs Usability Survey

By Tiffany Hrabusa (Grafana Labs) | Wednesday, December 18, 2024



<https://opentelemetry.io/blog/2024/otel-docs-survey/>

It can be tricky



r/OpenTelemetry · 5mo ago
Character_Internet_3

has anyone being successful using c++ metrics API?

Hello masters, I have been reading the otel documentation regarding to c++ api for metrics. What I now understand is that I have to create an exporter, then a metric provider and then create my instruments (gauges, counters). This have been extremely frustrating because it seems that there is not any implementation that works. The otel's web page example is not working, the github example is not implementing gauges and also is not working, and the readthedocs page shows examples with uncallable objects. I could compile a sample app with a provider and a metric exporter to Osstream, but there is no way to make an updowncounter or a gauge to work. Do you know if there are references/tutorials or even working documentation portals?

4 5

[Single comment thread](#) [See full discussion](#)

Character_Internet_3 OP · 3mo ago

Muchas gracias, tenias toda la razón. tu repo me sirvió para poder entender como Funciona otel en cpp y que importar. Pide por fin crear mis gauges y reportar datos. De nuevo, mil gracias.

2

jgomezselles · 3mo ago

Me alegre mucho! Happy OTel'ing! 😊

2

<https://www.reddit.com/r/OpenTelemetry/comments/1l4wvpk/comment/n20pjfu/>

OpenTelemetry



OpenTelemetry is a collection of **APIs, SDKs, and tools**. Use it to **instrument, generate, collect, and export** telemetry data (metrics, logs, and traces) to help you analyze your software's performance and behavior.

OpenTelemetry



generate, collect, and export telemetry data
analyze your software's performance

OpenTelemetry



generate, collect, and export telemetry data
analyze your software's performance

Instrumentation

(generating the signals)

OTel Instrumentation: Zero-code

- **Effortless**
- **Agent-driven**
- **Mechanisms**
 - **Monkey patching**
 - **Bytecode manipulation**
 - **eBPF**

<https://opentelemetry.io/docs/concepts/instrumentation/>



Monkey patching

```
>>> import math  
>>> math.pi  
3.141592653589793
```

https://en.wikipedia.org/wiki/Monkey_patch

Monkey patching

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.pi = 3.2    # monkey-patch the value of Pi in the math module
>>> math.pi
3.2
```



At runtime!

https://en.wikipedia.org/wiki/Monkey_patch

Monkey patching

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.pi = 3.2    # monkey-patch the value of Pi in the math module
>>> math.pi
3.2
===== RESTART =====
>>> import math
>>> math.pi
3.141592653589793
>>>
```

https://en.wikipedia.org/wiki/Monkey_patch

OTel Instrumentation: Zero-code

- **Effortless**
- **Agent-driven**
- **Mechanisms**
 - **Monkey patching**
 - **Bytecode manipulation**
 - **eBPF**
- **Config for libs and exporters**
- **Great for edges and libs**

<https://opentelemetry.io/docs/concepts/instrumentation/>



#NotAllLanguages

OTel Instrumentation: Zero-code

- Effortless
- Agent-driven
- Mechanisms
 - Monkey patching
 - Bytecode manipulation
 - eBPF
- Config for libs and exporters
- Great for edges and libs

But...

- Not all languages support it
- Hard to control data
 - Costs
 - Performance
- What about my code?

Manual instrumentation
*(generating the signals...
ourselves)*

IF YOU WANT
SOMETHING DONE
RIGHT
DO IT YOURSELF

OTel Spec: SDK & API

- **API**: enables developers to instrument their applications and libraries in order to make them ready to create and emit telemetry data
- **SDK**: exporting, sampling, and aggregating telemetry data
- This architecture enables developers to instrument applications and libraries with the OpenTelemetry API while being **completely agnostic of how telemetry data is exported and processed**.

Metrics

Instrumenting metrics in C++

New error!

```
void stats::add_error(const std::string& id, const int e)
{
    /*
    Some code here to process the error
    */
    ++responses_err;
}
```

Instrumenting metrics in C++

New error!

```
void stats::add_error(const std::string& id, const int e)
{
    /*
     Some code here to process the error
    */
    ++responses_err;
}
```

Goal



OTel Spec: SDK & API

API

- **Measurement:** represents a data point reported via the metrics API to the SDK
- **Instruments** are used to report Measurements. Each Instrument will have the following parameters: (name, kind - counter, histogram..., opt. Unit of measure, opt description)
- **Meter:** The meter is responsible for creating Instruments
- **Meter Provider:** the interface

<https://opentelemetry.io/docs/specs/otel/metrics/api>

OTel Spec: SDK & API

API

- **Measurement:** represents a data point reported via the metrics API to the SDK
- **Instruments** are used to report Measurements. Each Instrument will have the following parameters: (name, kind - counter, histogram..., opt. Unit of measure, opt description)
- **Meter:** The meter is responsible for creating Instruments
- **Meter Provider:** the interface

SDK

- **Reader:** periodically collects metrics from the Aggregation Store
- **Exporter:** sends data
- (...and more)

<https://opentelemetry.io/docs/specs/otel/metrics/api>

Instrumenting metrics in C++

Creating an instrument

```
auto provider = opentelemetry::metrics::Provider::GetMeterProvider();
```

Meter Provider: the interface



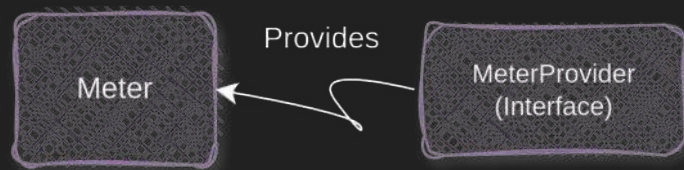
MeterProvider
(Interface)

Instrumenting metrics in C++

Creating an instrument

```
auto provider = opentelemetry::metrics::Provider::GetMeterProvider();  
auto meter = provider->GetMeter("io.opentelemetry.contrib.mongodb"); // lib, package, module or class name
```

Meter: The meter is responsible for creating Instruments

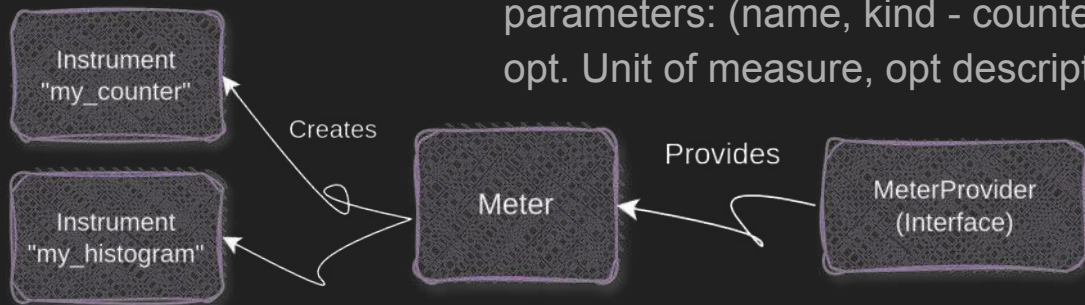


Instrumenting metrics in C++

Creating an instrument

```
auto provider = opentelemetry::metrics::Provider::GetMeterProvider();  
auto meter = provider->GetMeter("io.opentelemetry.contrib.mongodb"); // lib, package, module or class name  
auto resp_nok = meter->CreateUInt64Counter("hermes_responses_rcv_err", "Unsuccessful responses received");
```

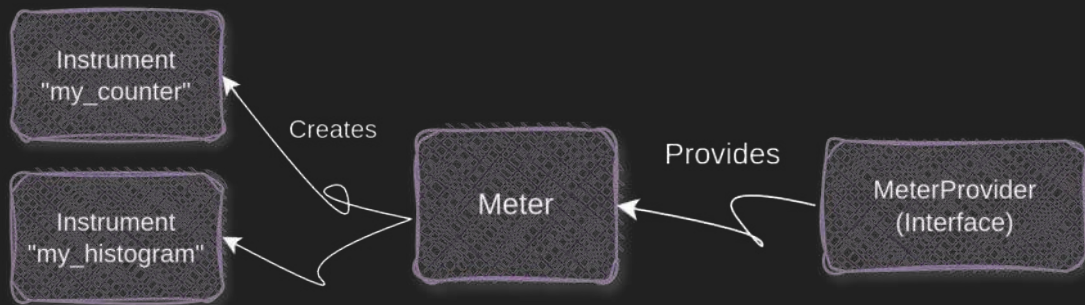
Instruments are used to report Measurements. Each Instrument will have the following parameters: (name, kind - counter, histogram..., opt. Unit of measure, opt description)



Instrumenting metrics in C++

Creating an instrument

```
auto provider = opentelemetry::metrics::Provider::GetMeterProvider();  
auto meter = provider->GetMeter("io.opentelemetry.contrib.mongodb"); // lib, package, module or class name  
auto resp_nok = meter->CreateUInt64Counter("hermes_responses_rcv_err", "Unsuccessful responses received");  
opentelemetry::v1::nstd::unique_ptr<opentelemetry::v1::metrics::Counter<uint64_t>> responses_err;  
responses_err = std::move(resp_nok);
```



Instrumenting metrics in C++

New error!

```
void stats::add_error(const std::string& id, const int e)
{
    /*
     Some code here to process the error
    */
    ++responses_err;
}
```

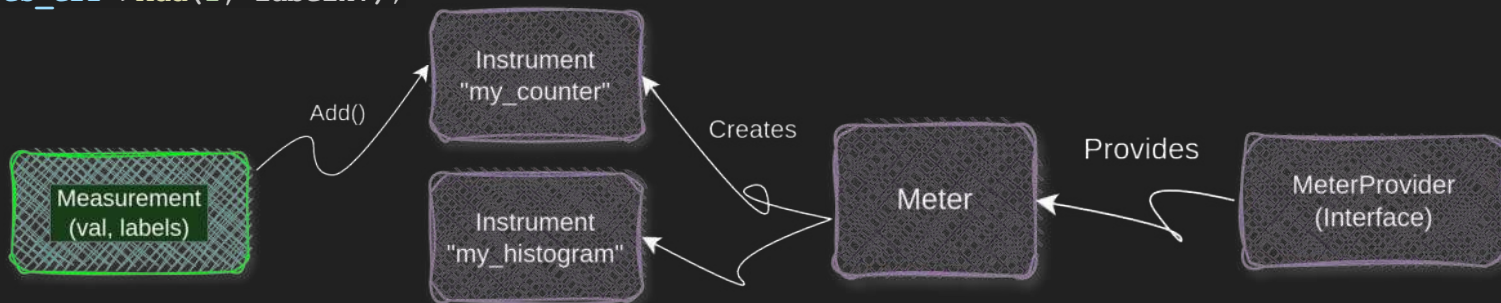
Goal



Instrumenting metrics in C++

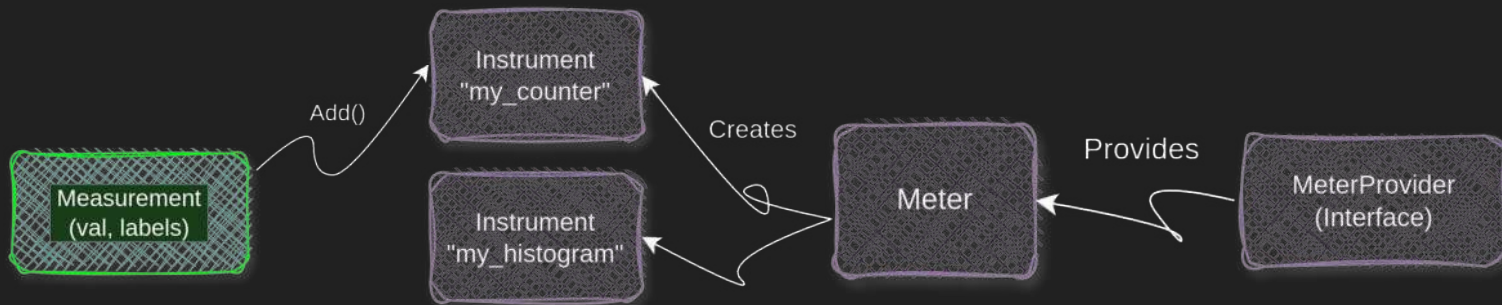
New error!

```
void stats::add_error(const std::string& id, const int e)
{
    /*
     * Some code here to process the error
     */
    std::map<std::string, std::string> labels{{"id", id}, {"response_code", std::to_string(e)}};
    auto labelkv = opentelemetry::common::KeyValueIterableView<decltype(labels)>{labels};
    responses_err->Add(1, labelkv);
}
```



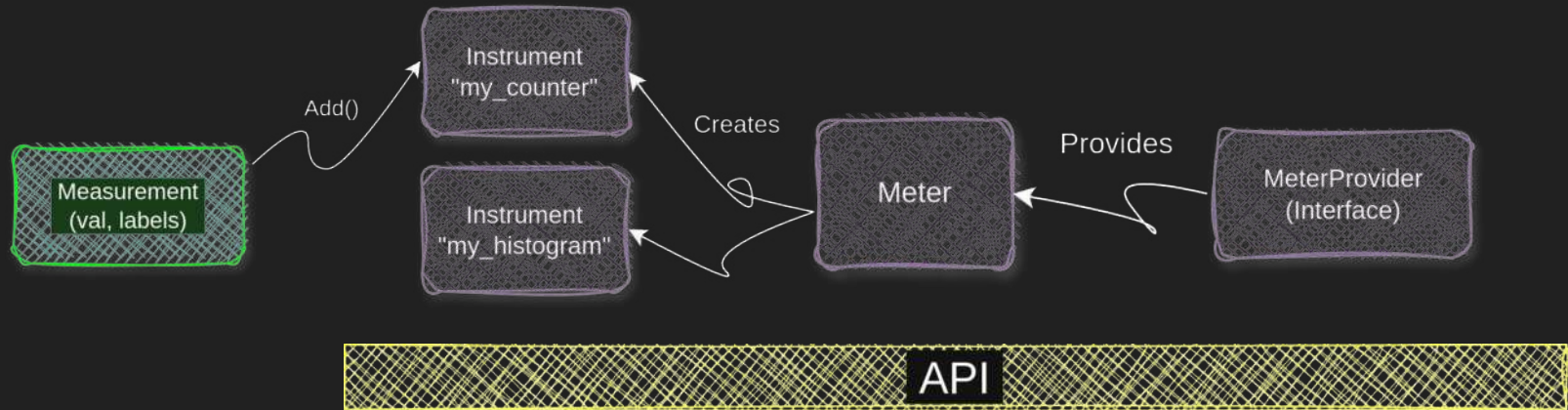
Instrumenting metrics in C++

Measurement: represents a data point reported via the metrics API to the SDK

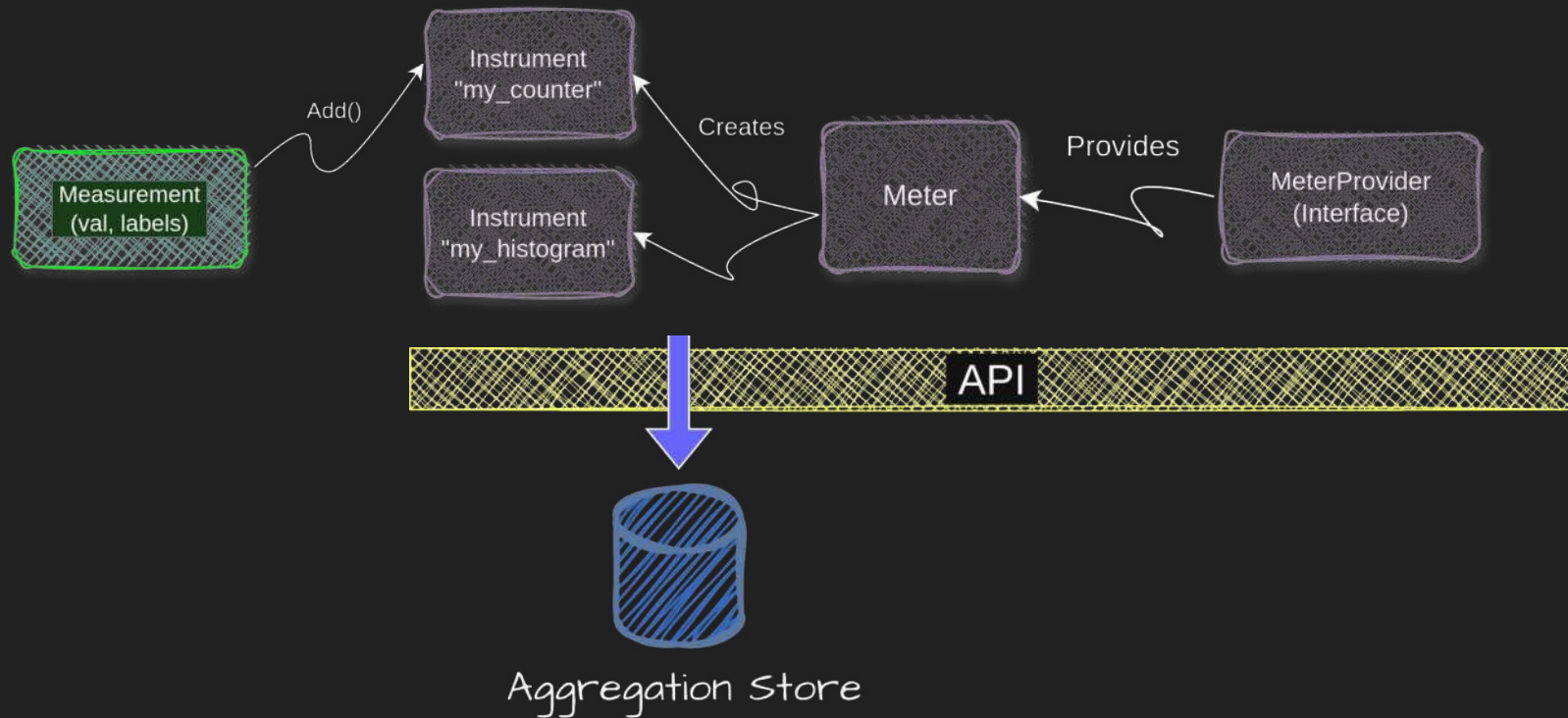


Instrumenting metrics in C++

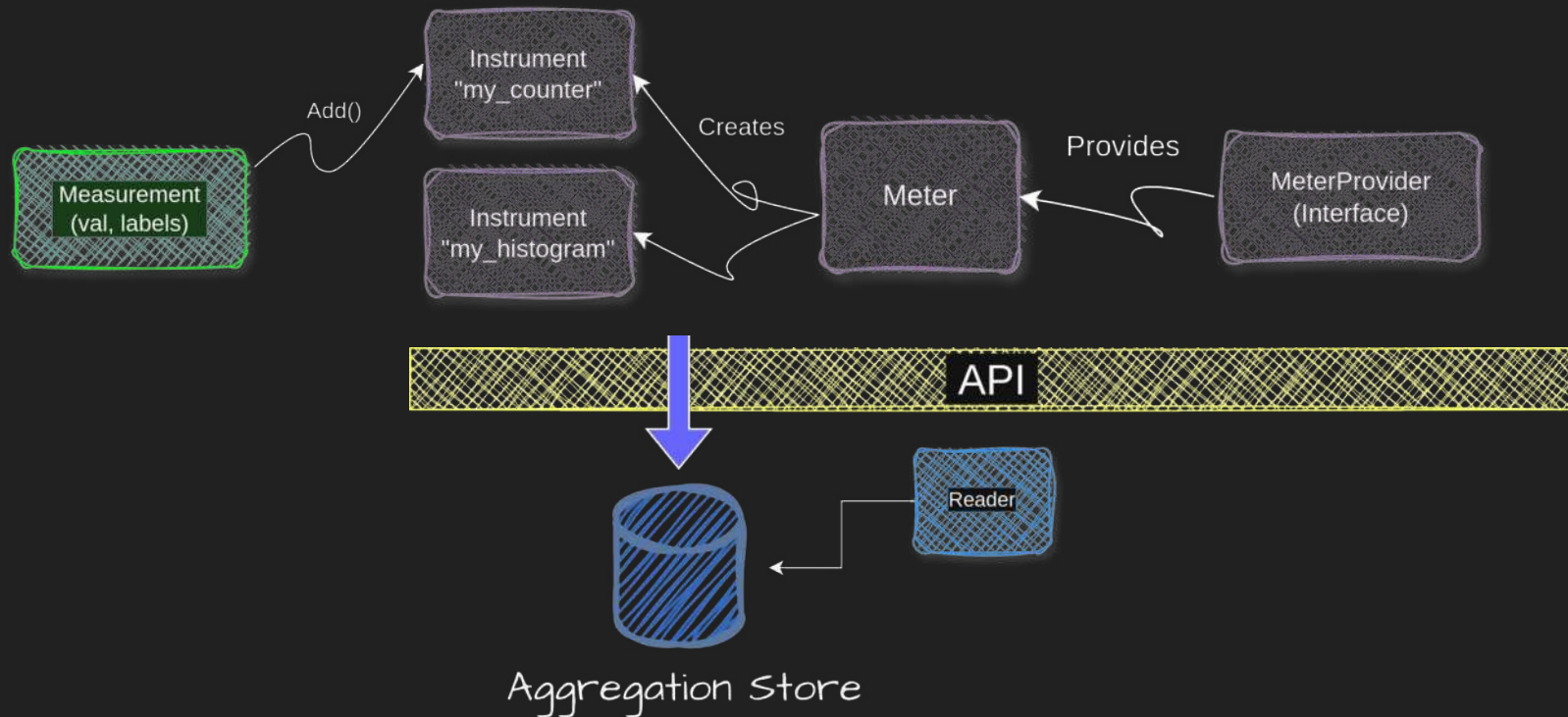
Measurement: represents a data point reported via the metrics **API** to the **SDK**



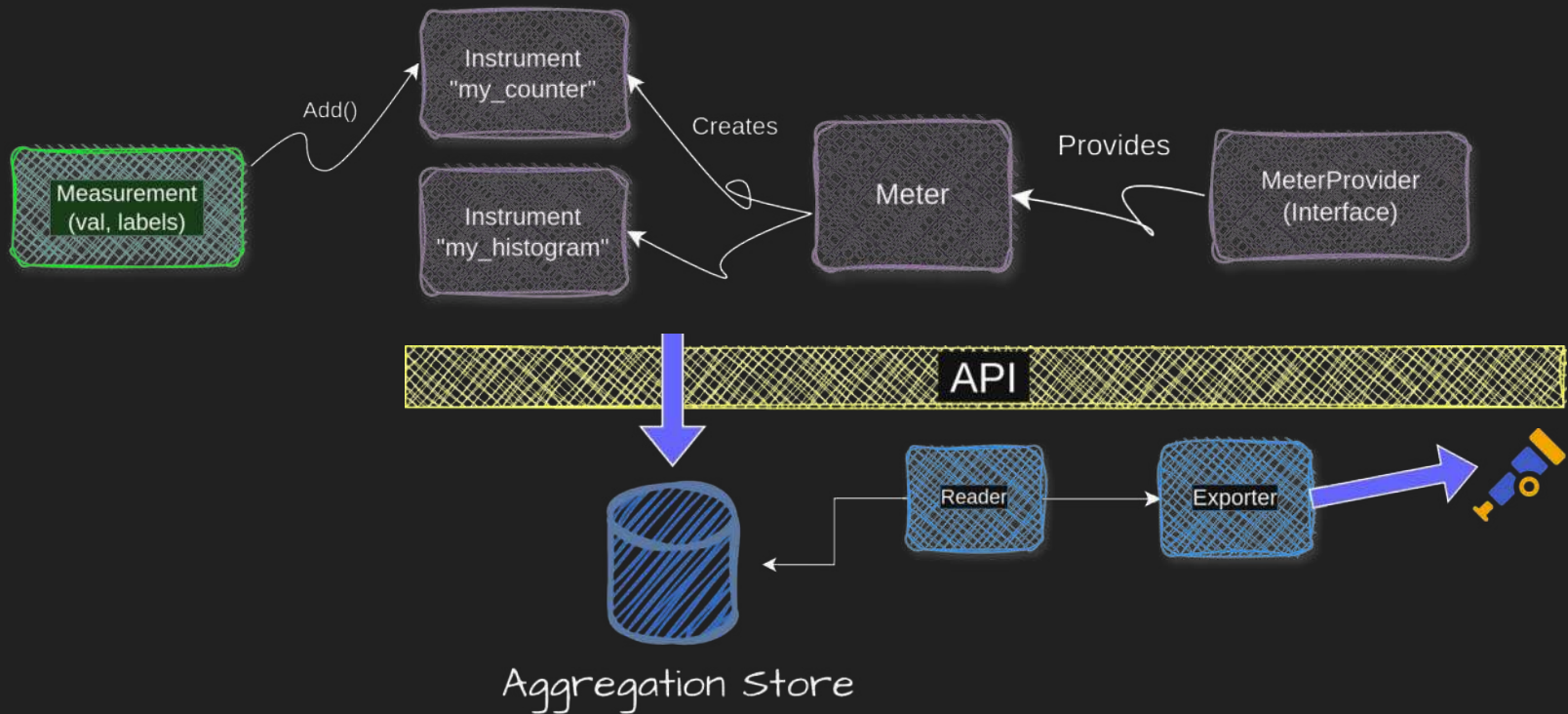
Instrumenting metrics in C++



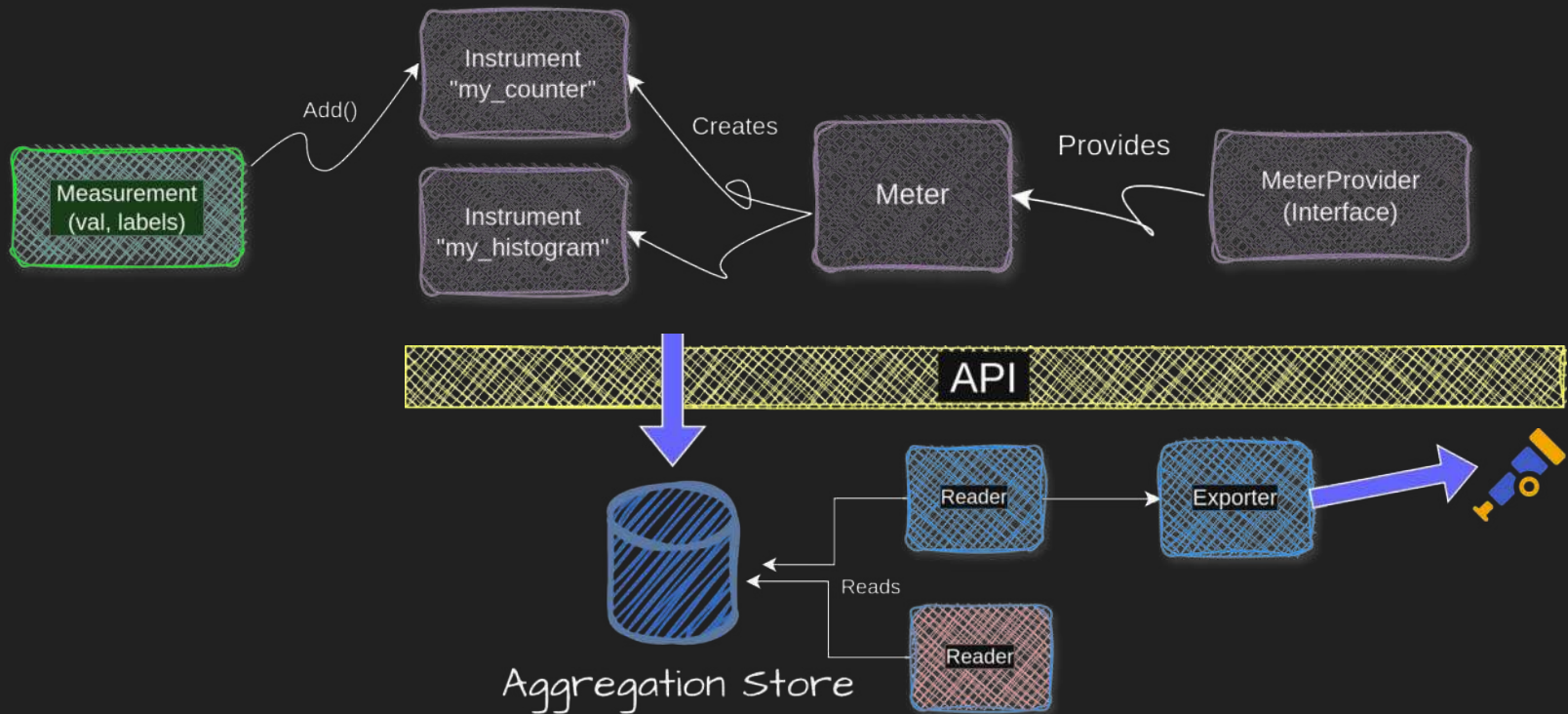
Instrumenting metrics in C++



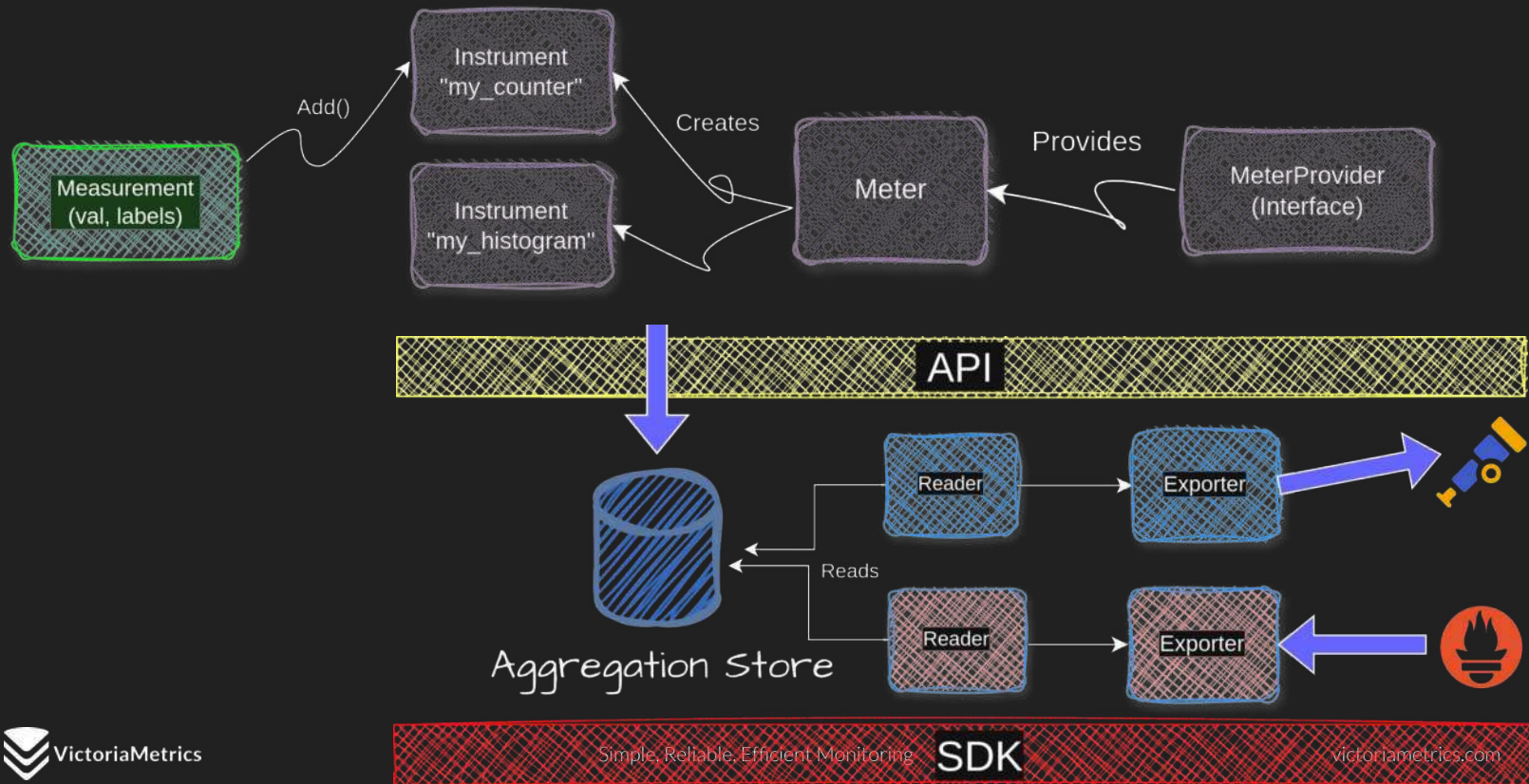
Instrumenting metrics in C++



Instrumenting metrics in C++



Instrumenting metrics in C++



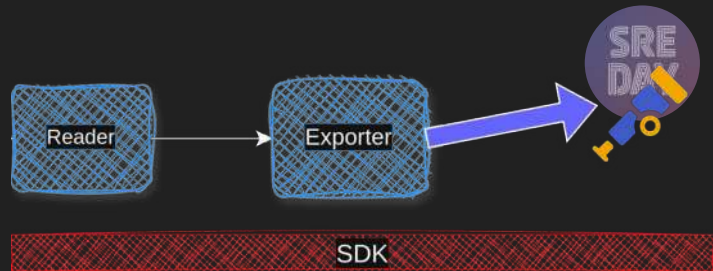
Instrumenting metrics in C++

```
void init_metrics_otlp_http(const std::string& url)
{
    sdk_metrics::PeriodicExportingMetricReaderOptions reader_options;
    reader_options.export_interval_millis = std::chrono::milliseconds(1000);
    reader_options.export_timeout_millis = std::chrono::milliseconds(500);

    opentelemetry::exporter::otlp::OtlpHttpMetricExporterOptions otlpOptions;
    otlpOptions.url = url;
    otlpOptions.content_type = opentelemetry::exporter::otlp::HttpRequestContentType::kBinary;
    otlpOptions.console_debug = true;
    auto exporter =
        opentelemetry::exporter::otlp::OtlpHttpMetricExporterFactory::Create(otlpOptions);
    auto reader = sdk_metrics::PeriodicExportingMetricReaderFactory::Create(std::move(exporter),
                                                                              reader_options);

    auto context = sdk_metrics::MeterContextFactory::Create();
    context->AddMetricReader(std::move(reader));

    auto u_provider = sdk_metrics::MeterProviderFactory::Create(std::move(context));
    ot_std::shared_ptr<ot_metrics::MeterProvider> provider(std::move(u_provider));
    ot_metrics::Provider::SetMeterProvider(provider);
}
```



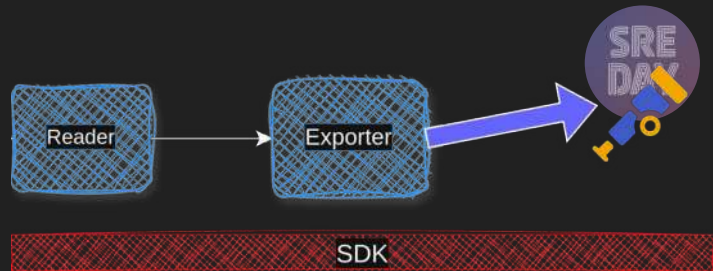
Instrumenting metrics in C++

```
void init_metrics_otlp_http(const std::string& url)
{
    sdk_metrics::PeriodicExportingMetricReaderOptions reader_options;
    reader_options.export_interval_millis = std::chrono::milliseconds(1000);
    reader_options.export_timeout_millis = std::chrono::milliseconds(500);

    opentelemetry::exporter::otlp::OtlpHttpMetricExporterOptions otlpOptions;
    otlpOptions.url = url;
    otlpOptions.content_type = opentelemetry::exporter::otlp::HttpRequestContentType::kBinary;
    otlpOptions.console_debug = true;
    auto exporter =
        opentelemetry::exporter::otlp::OtlpHttpMetricExporterFactory::Create(otlpOptions);
    auto reader = sdk_metrics::PeriodicExportingMetricReaderFactory::Create(std::move(exporter),
                                                                              reader_options);

    auto context = sdk_metrics::MeterContextFactory::Create();
    context->AddMetricReader(std::move(reader));

    auto u_provider = sdk_metrics::MeterProviderFactory::Create(std::move(context));
    ot_std::shared_ptr<ot_metrics::MeterProvider> provider(std::move(u_provider));
    ot_metrics::Provider::SetMeterProvider(provider);
}
```



C++ & Singleton stuff

```
void init_metrics_otlp_http(const std::string& url)
{
    sdk_metrics::PeriodicExportingMetricReaderOptions reader_options;
    reader_options.export_interval_millis = std::chrono::milliseconds(1000);
    reader_options.export_timeout_millis = std::chrono::milliseconds(500);

    opentelemetry::exporter::otlp::OtlpHttpMetricExporterOptions otlpOptions;
    otlpOptions.url = url;
    otlpOptions.content_type = opentelemetry::exporter::otlp::HttpRequestContentType;
    otlpOptions.console_debug = true;

    auto exporter =
        opentelemetry::exporter::otlp::OtlpHttpMetricExporterFactory::Create(otlpOptions);
    auto reader = sdk_metrics::PeriodicExportingMetricReaderFactory::Create(std::move(reader_options));

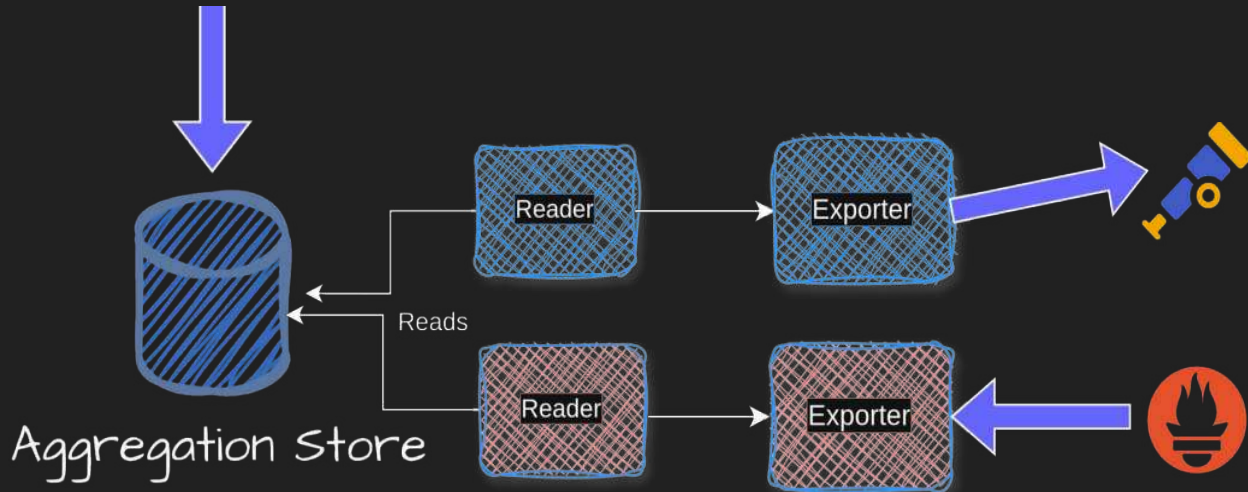
    auto context = sdk_metrics::MeterContextFactory::Create();
    context->AddMetricReader(std::move(reader));

    auto u_provider = sdk_metrics::MeterProviderFactory::Create(std::move(context));
    ot_std::shared_ptr<ot_metrics::MeterProvider> provider(std::move(u_provider));
    ot_metrics::Provider::SetMeterProvider(provider);
}
```

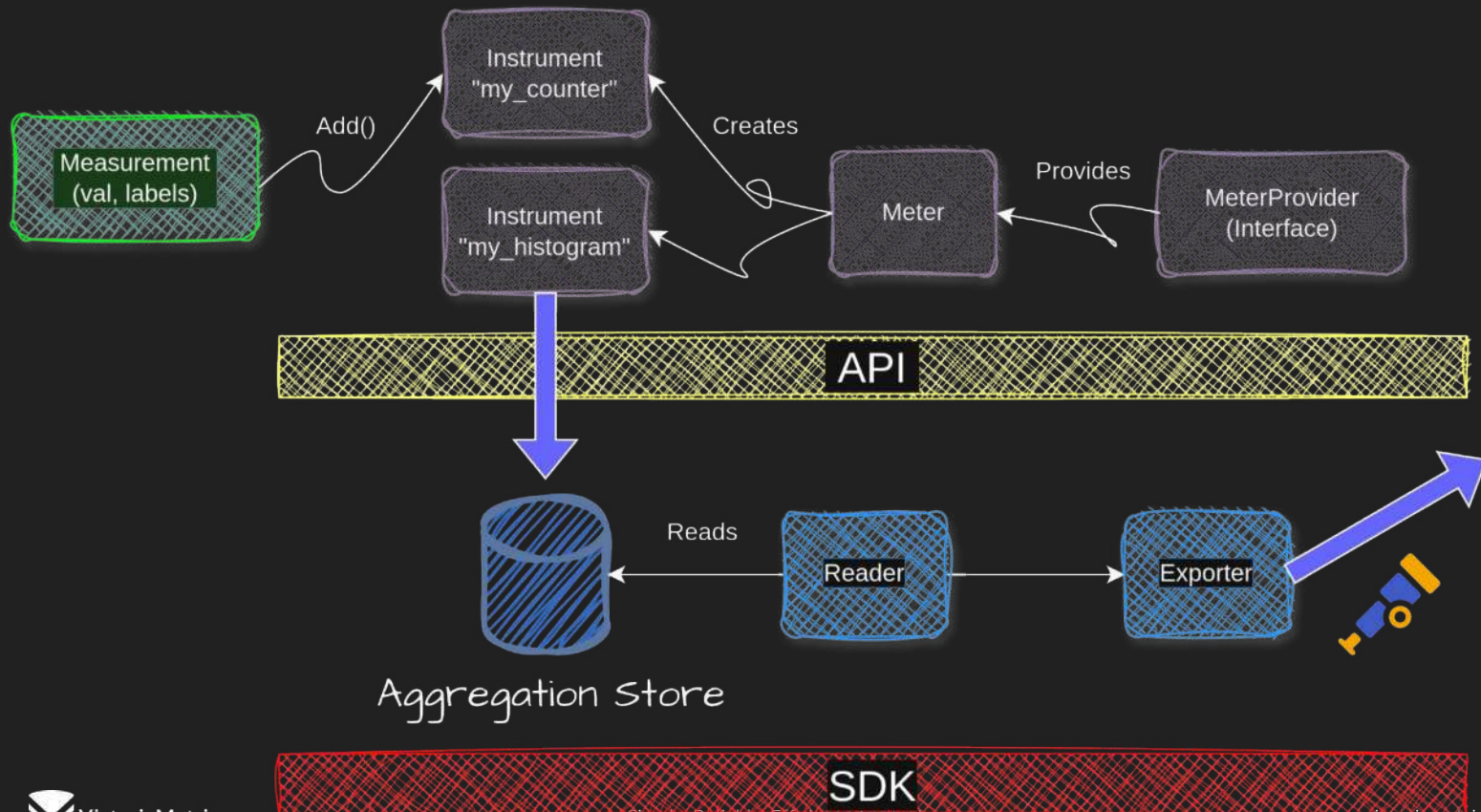


Instrumenting metrics in C++

```
void init_metrics_otlp_http(const std::string& url) { ... }  
void init_metrics_prometheus(const std::string& url) { ... }
```

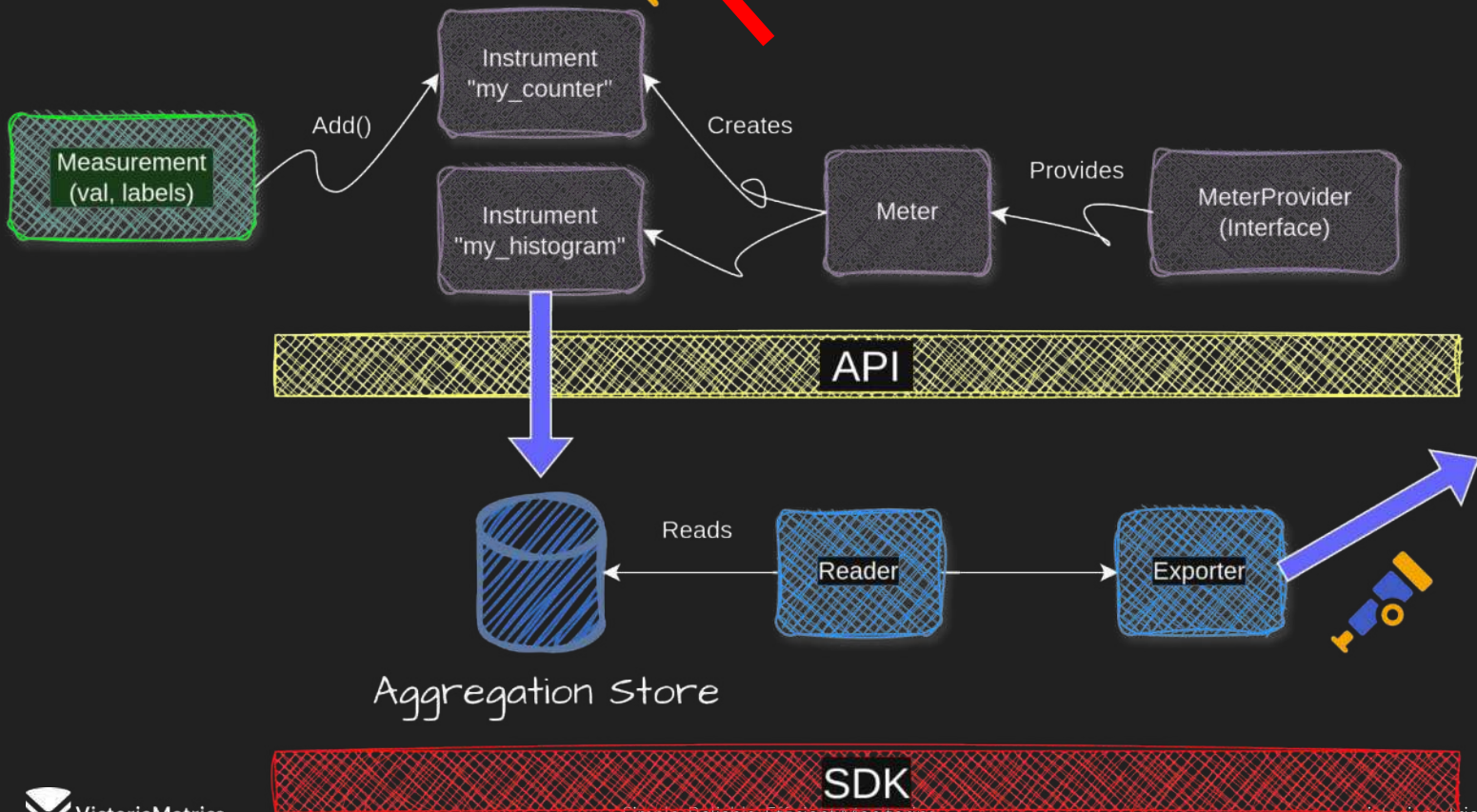


Instrumenting metrics in C++



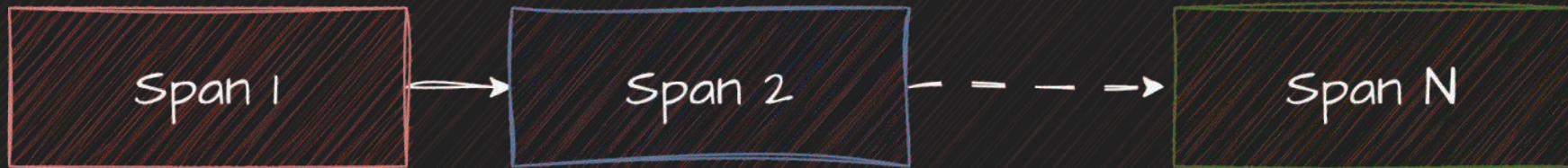
Instrumenting metrics in C++

Language agnostic!



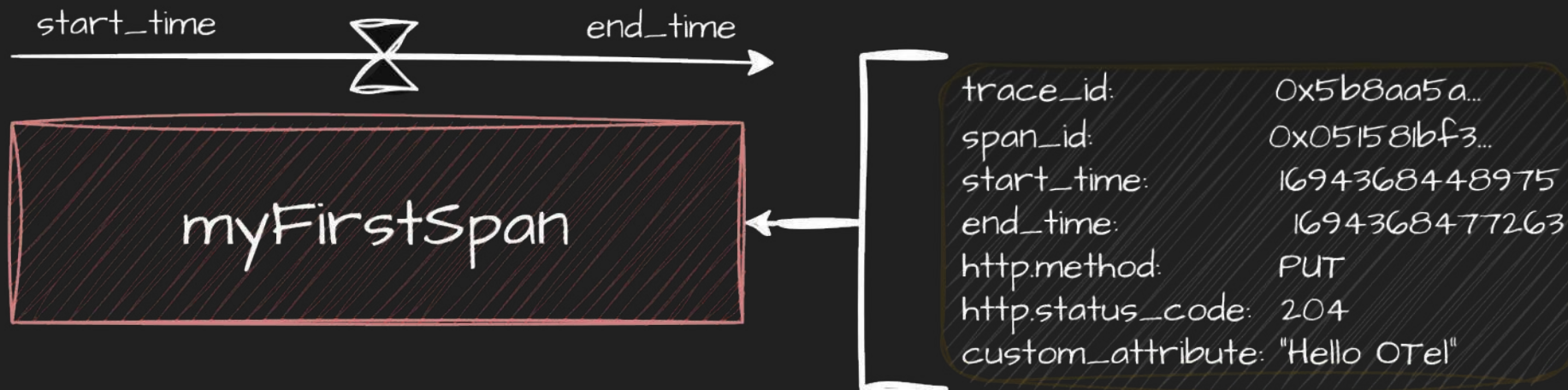
Traces (quick intro?)

Traces

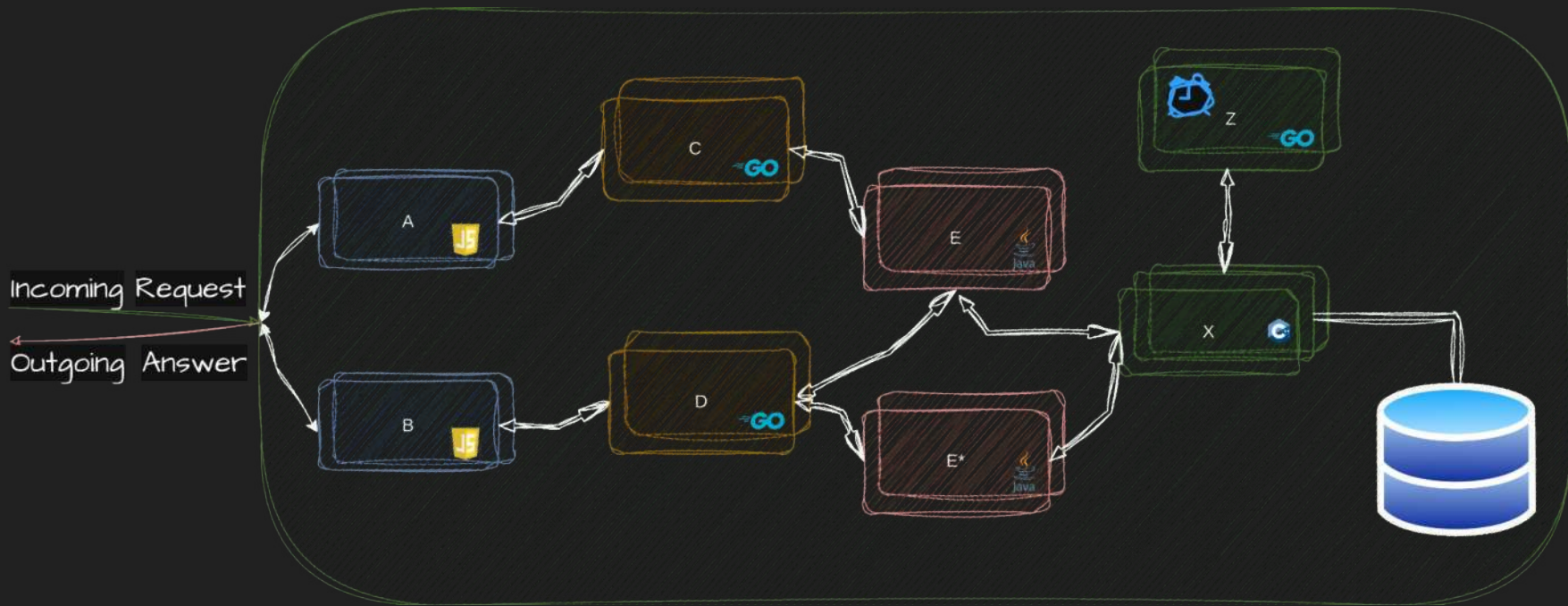


trace_id: 123e4567-e89b-12d3-a456-426655440000

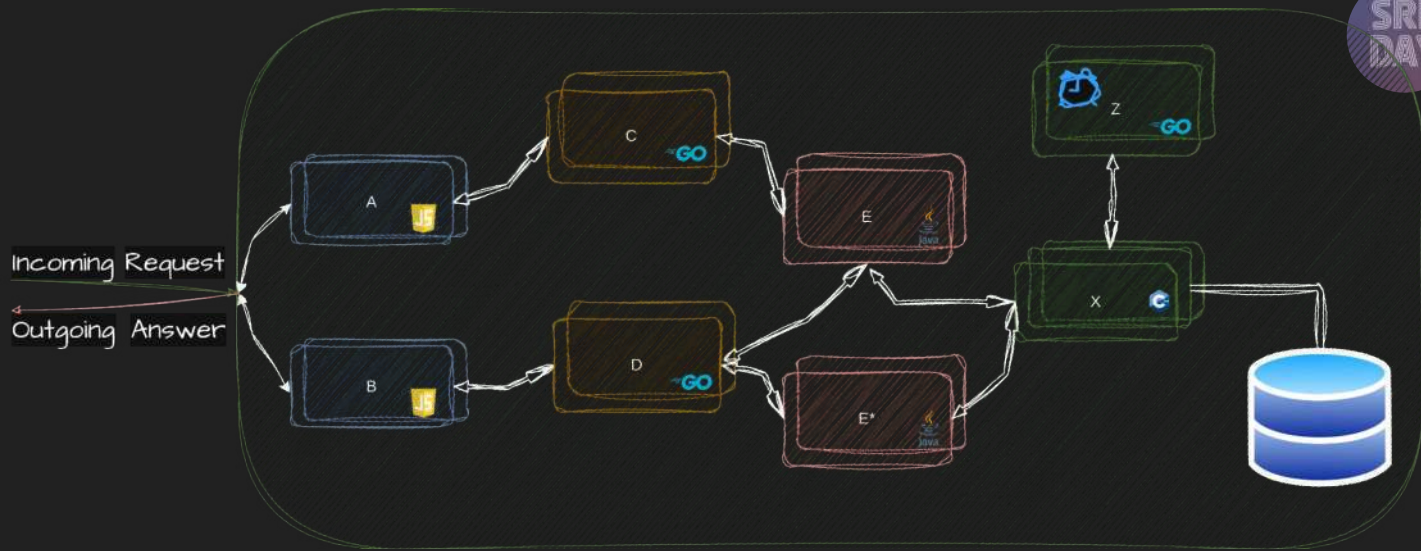
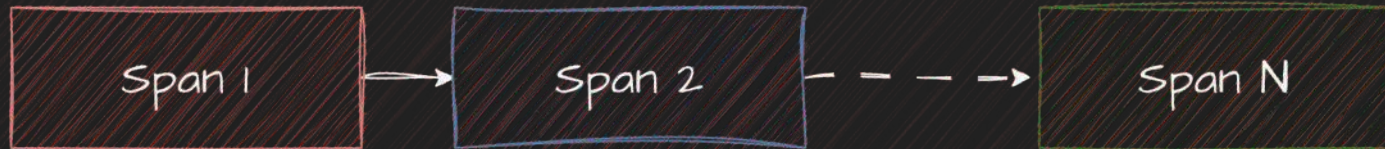
Traces



Traces



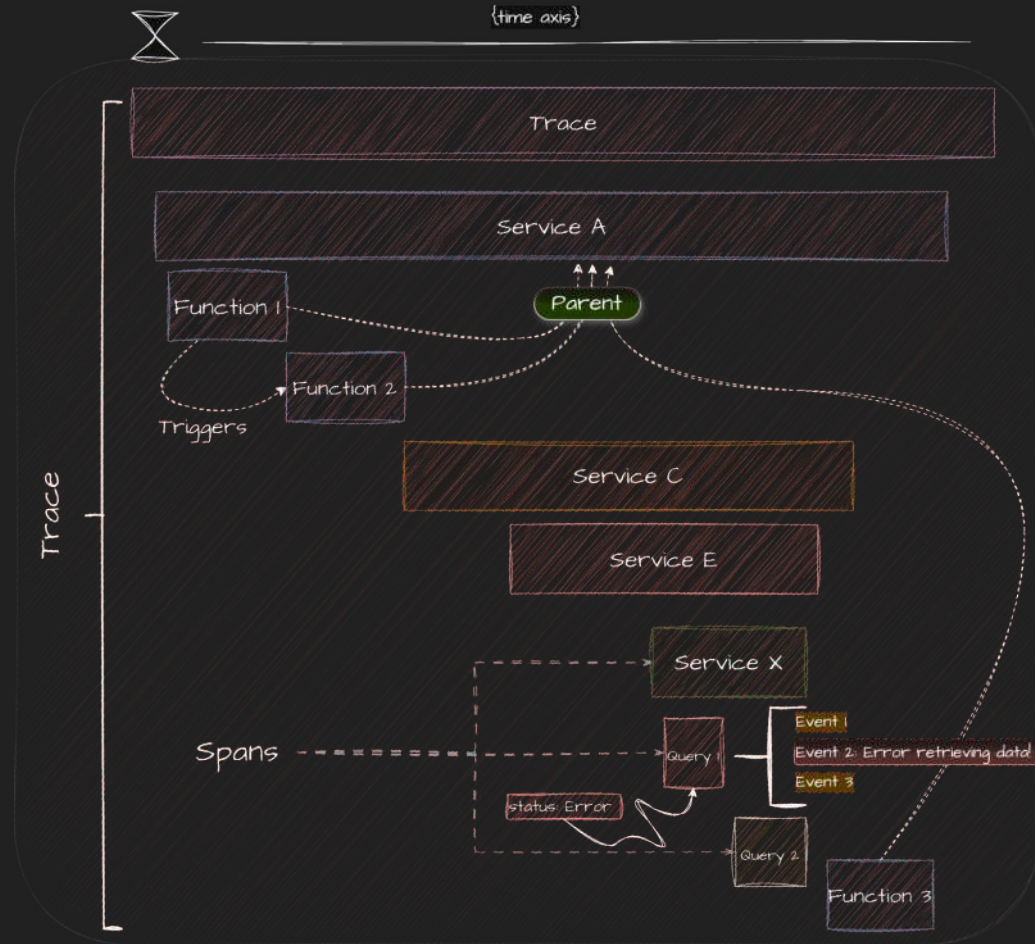
Traces



trace_id: 123e4567-e89b-12d3-a456-426655440000



Traces



Traces (instrumenting)

OTel Spec: SDK & API

API

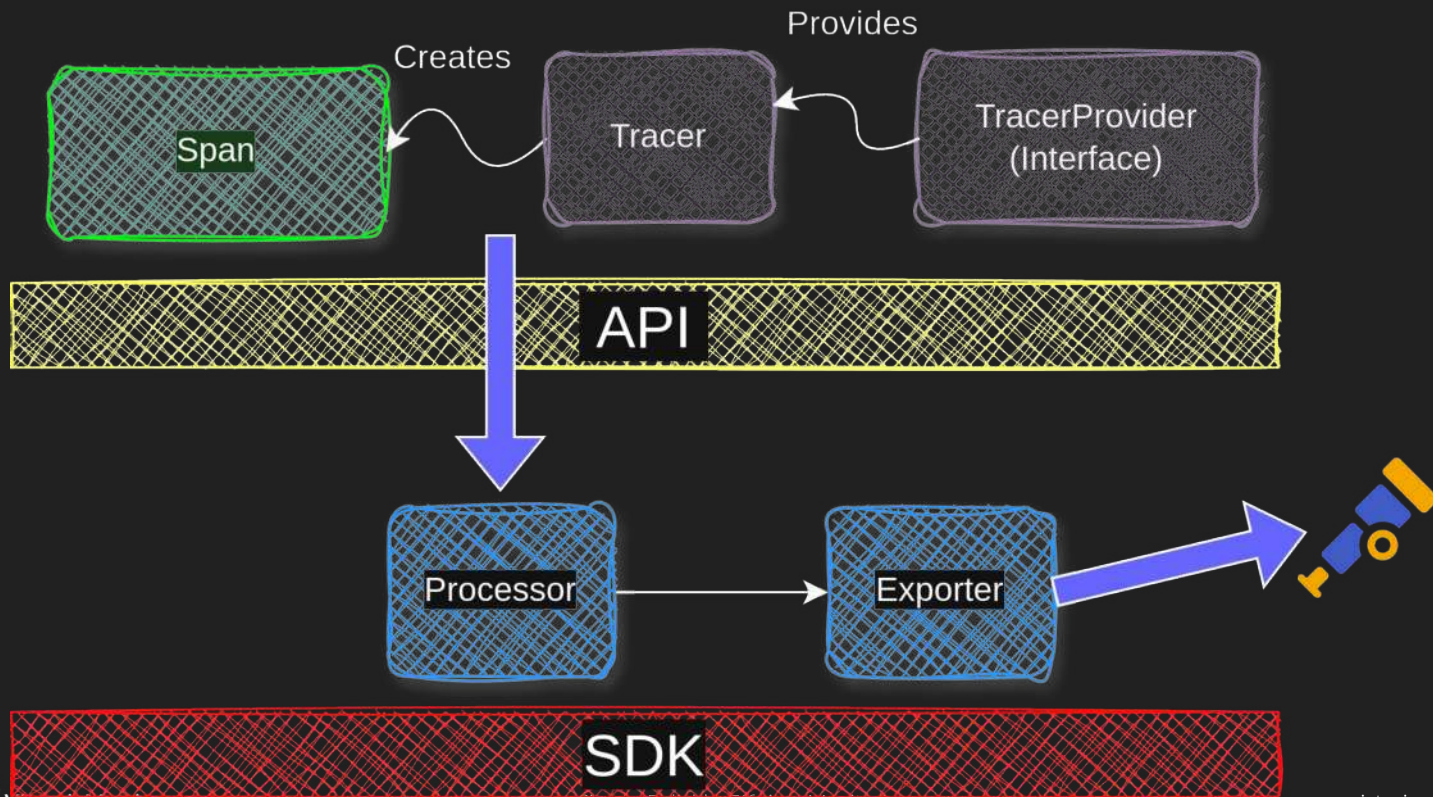
- **Tracer Provider:** the interface
- **Tracer:** The tracer is responsible for creating spans
- **Span:** represents a single operation within a trace

SDK

- **Span Processor:** for batching and conversion of spans to exportable representation and passing batches to exporters
- **Span Exporter:** sends data
- (...and more)

<https://opentelemetry.io/docs/specs/otel/trace/>

OTel Spec: SDK & API



Instrumenting traces in C++: init

```
void init_tracer(const std::string& url)
{
```

```
    auto attrs = opentelemetry::sdk::resource::ResourceAttributes{{"service.name", "my_service"}};
    auto resource = opentelemetry::sdk::resource::Resource::Create(attrs);
```

```
    opentelemetry::exporter::otlp::OtlpHttpExporterOptions http_opts;
```

```
    http_opts.url = url;
```

```
    auto exporter = opentelemetry::exporter::otlp::OtlpHttpExporterFactory::Create(http_opts);
```

```
    sdk_trace::BatchSpanProcessorOptions opts;
```

```
    opts.max_queue_size = 2048;
```

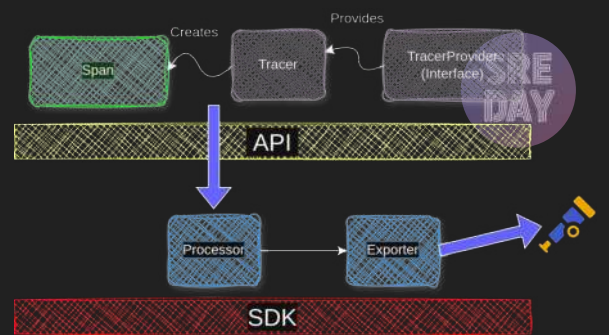
```
    opts.max_export_batch_size = 512;
```

```
    auto processor = sdk_trace::BatchSpanProcessorFactory::Create(std::move(exporter), opts);
```

```
    auto provider = sdk_trace::TracerProviderFactory::Create(std::move(processor), resource);
```

```
    ot_trace::Provider::SetTracerProvider(std::move(provider));
```

```
}
```



Instrumenting traces in C++: init

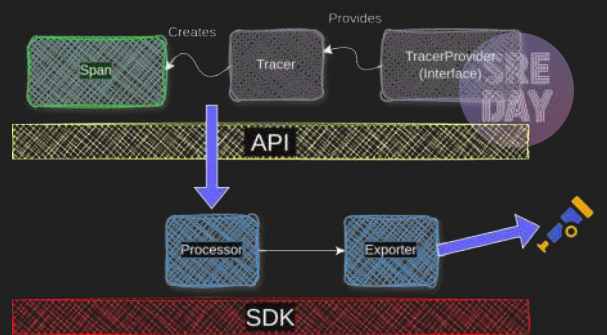
```
void init_tracer(const std::string& url)
{
    auto attrs = opentelemetry::sdk::resource::ResourceAttributes{{"service.name", "my_service"}};
    auto resource = opentelemetry::sdk::resource::Resource::Create(attrs);

    opentelemetry::exporter::otlp::OtlpHttpExporterOptions http_opts;
    http_opts.url = url;
    auto exporter = opentelemetry::exporter::otlp::OtlpHttpExporterFactory::Create(http_opts);

    sdk_trace::BatchSpanProcessorOptions opts;
    opts.max_queue_size = 2048;
    opts.max_export_batch_size = 512;
    auto processor = sdk_trace::BatchSpanProcessorFactory::Create(std::move(exporter), opts);

    auto provider = sdk_trace::TracerProviderFactory::Create(std::move(processor), resource);

    ot_trace::Provider::SetTracerProvider(std::move(provider));
}
```



Instrumenting traces in C++: init

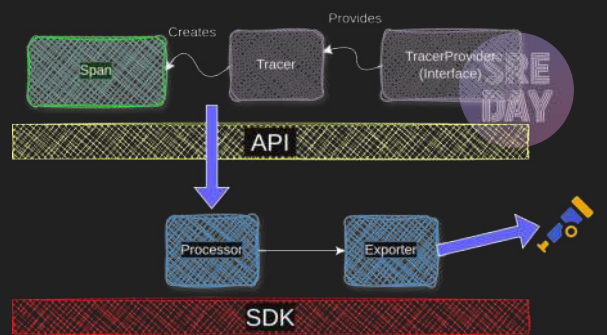
```
void init_tracer(const std::string& url)
{
    auto attrs = opentelemetry::sdk::resource::ResourceAttributes{{"service.name", "my_service"}};
    auto resource = opentelemetry::sdk::resource::Resource::Create(attrs);

    opentelemetry::exporter::otlp::OtlpHttpExporterOptions http_opts;
    http_opts.url = url;
    auto exporter = opentelemetry::exporter::otlp::OtlpHttpExporterFactory::Create(http_opts);

    sdk_trace::BatchSpanProcessorOptions opts;
    opts.max_queue_size = 2048;
    opts.max_export_batch_size = 512;
    auto processor = sdk_trace::BatchSpanProcessorFactory::Create(std::move(exporter), opts);

    auto provider = sdk_trace::TracerProviderFactory::Create(std::move(processor), resource);

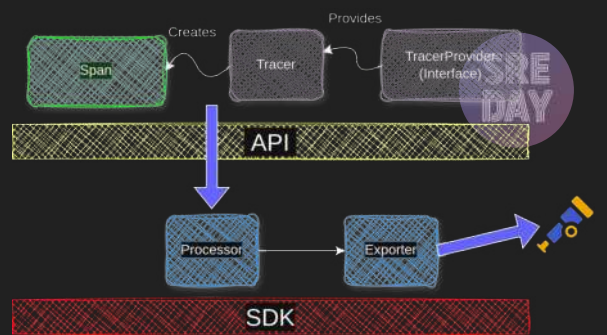
    ot_trace::Provider::SetTracerProvider(std::move(provider));
}
```



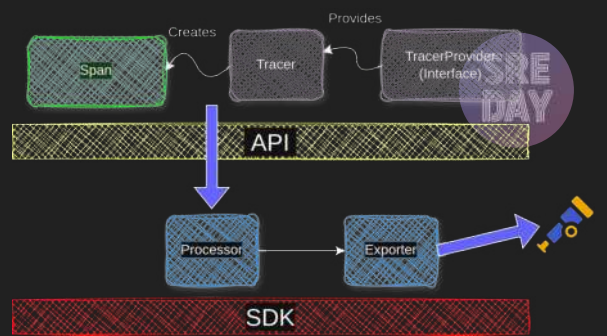
Instrumenting traces in C++: aux fn

```
ot_std::shared_ptr<ot_trace::Span> create_span(const std::string& name)
{
    ot_trace::StartSpanOptions opts;
    opts.kind = ot_trace::SpanKind::kClient;

    auto span = get_tracer("my_client")->StartSpan(name, opts);
    return span;
}
```



Instrumenting traces in C++: aux fn



```
ot_std::shared_ptr<Span> create_child_span(const std::string& name, const ot_std::shared_ptr<Span>& parent)
{
    ot_trace::StartSpanOptions opts;
    opts.kind = ot_trace::SpanKind::kClient;
    if (parent)
    {
        opts.parent = parent->GetContext();
    }

    auto span = get_tracer("my_client")->StartSpan(name, opts);
    return span;
}
```

Instrumenting traces in C++

```
void client_impl::send() {
```

```
    auto span = o11y::create_child_span(req.name, script.get_span());  
    span->SetAttribute(ot_trace::SemanticConventions::kUrlFull, req.url);  
    span->SetAttribute(ot_trace::SemanticConventions::kHttpRequestMethod, req.method);
```

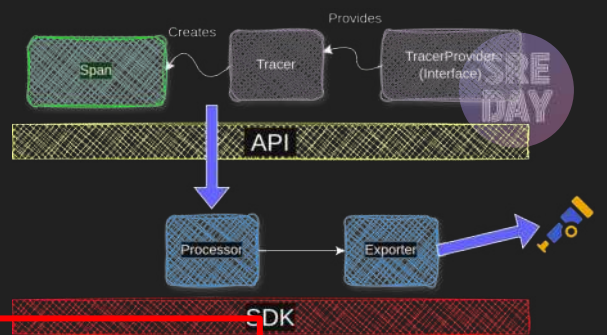
```
    auto nghttp_req = session.submit(ec, req.method, req.url, req.body, req.headers);
```

```
    requests_sent->Add(1, labelkv);
```

```
    span->AddEvent("Request sent");
```

```
    ...
```

```
}
```



Semantic conventions

<https://opentelemetry.io/docs/specs/semconv/>

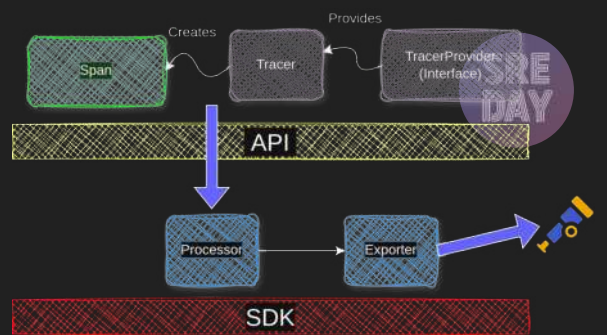
Semantic Conventions are defined for the following areas:

- **General**: General Semantic Conventions.
- CICD: Semantic Conventions for CICD systems.
- Code: Semantic Conventions for code.
- Cloud Providers: Semantic Conventions for cloud providers libraries.
- CloudEvents: Semantic Conventions for the CloudEvents specification.
- Database: Semantic Conventions for database operations.
- Exceptions: Semantic Conventions for exceptions.
- FaaS: Semantic Conventions for Function as a Service (FaaS) operations.
- Feature Flags: Semantic Conventions for feature flag evaluations.
- Generative AI: Semantic Conventions for generative AI (LLM, etc.) operations.
- GraphQL: Semantic Conventions for GraphQL implementations.
- HTTP: Semantic Conventions for HTTP client and server operations.
- Messaging: Semantic Conventions for messaging operations and systems.
- Object Stores: Semantic Conventions for object stores operations.
- RPC: Semantic Conventions for RPC client and server operations.
- System: System Semantic Conventions.

Attribute	Type	Description	Examples	Requirement Level	Stability
<code>http.request.method</code>	string	HTTP request method. [1]	GET ; POST ; HEAD	Required	stable
<code>server.address</code>	string	Host identifier of the "URI origin " [↗] HTTP request is sent to. [2]	example.com ; 10.1.2.80 ; /tmp/my.sock	Required	stable
<code>server.port</code>	int	Port identifier of the "URI origin " [↗] HTTP request is sent to. [3]	80 ; 8080 ; 443	Required	stable
<code>url.full</code>	string	Absolute URL describing a network resource according to RFC3986 [↗] [4]	https://www.foo.bar/search?q=OpenTelemetry#SemConv ; //localhost	Required	stable
<code>error.type</code>	string	Describes a class of error the operation ended with. [5]	timeout ; java.net.UnknownHostException ; server_certificate_invalid ; 500	Conditionally Required If request has ended with an error.	stable

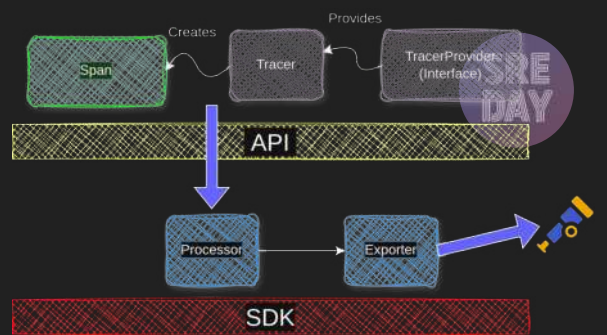
Instrumenting traces in C++

```
void client_impl::send() {  
  
    auto span = o11y::create_child_span(req.name, script.get_span());  
    span->SetAttribute(ot_trace::SemanticConventions::kUrlFull, req.url);  
    span->SetAttribute(ot_trace::SemanticConventions::kHttpRequestMethod, req.method);  
  
    auto nghttp_req = session.submit(ec, req.method, req.url, req.body, req.headers);  
  
    requests_sent->Add(1, labelkv);  
    span->AddEvent("Request sent");  
    ...  
}
```



Instrumenting traces in C++

```
void client_impl::send() {  
  
    auto span = o11y::create_child_span(req.name, script.get_span());  
    span->SetAttribute(ot_trace::SemanticConventions::kUrlFull, req.url);  
    span->SetAttribute(ot_trace::SemanticConventions::kHttpRequestMethod, req.method);  
  
    auto nghttp_req = session.submit(ec, req.method, req.url, req.body, req.headers);  
  
    requests_sent->Add(1, labelkv);  
    span->AddEvent("Request sent");  
    ...  
}
```

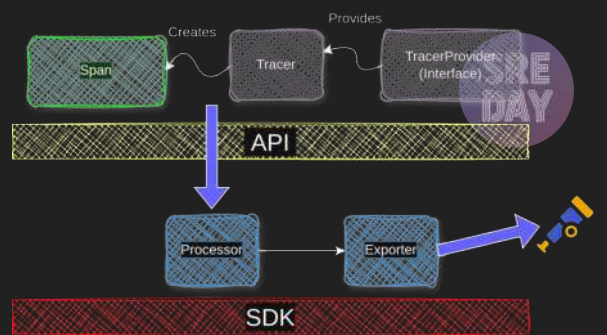


Instrumenting traces in C++

```
void client_impl::send() {  
    ...  
    nghttp_req->on_response( [this,..., span](const ng::client::response& res) {
```

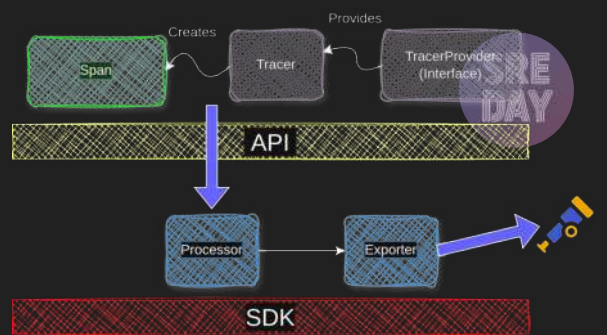
```
        span->AddEvent("Response received");  
        span->SetAttribute(ot_trace::SemanticConventions::kHttpResponseStatusCode,res.status_code());
```

```
    if (script.validate_answer(ans)) {  
        stats->add_measurement(req.name, elapsed_time, res.status_code());  
        span->SetStatus(ot_trace::StatusCode::kOk);  
    } else {  
        stats->add_error(req.name, res.status_code());  
        span->SetStatus(opentelemetry::trace::StatusCode::kError);  
        queue->cancel_script();  
    }  
    span->End();  
}  
}
```



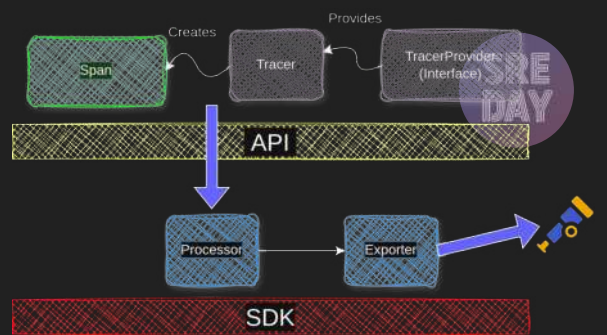
Instrumenting traces in C++

```
void client_impl::send() {  
    ...  
    nghttp_req->on_response( [this,..., span](const ng::client::response& res) {  
  
        span->AddEvent("Response received");  
        span->SetAttribute(ot_trace::SemanticConventions::kHttpResponseStatusCode,res.status_code());  
  
        if (script.validate_answer(ans)) {  
            stats->add_measurement(req.name, elapsed_time, res.status_code());  
            span->SetStatus(ot_trace::StatusCode::kOk);  
        } else {  
            stats->add_error(req.name, res.status_code());  
            span->SetStatus(opentelemetry::trace::StatusCode::kError);  
            queue->cancel_script();  
        }  
        span->End();  
    }  
}
```

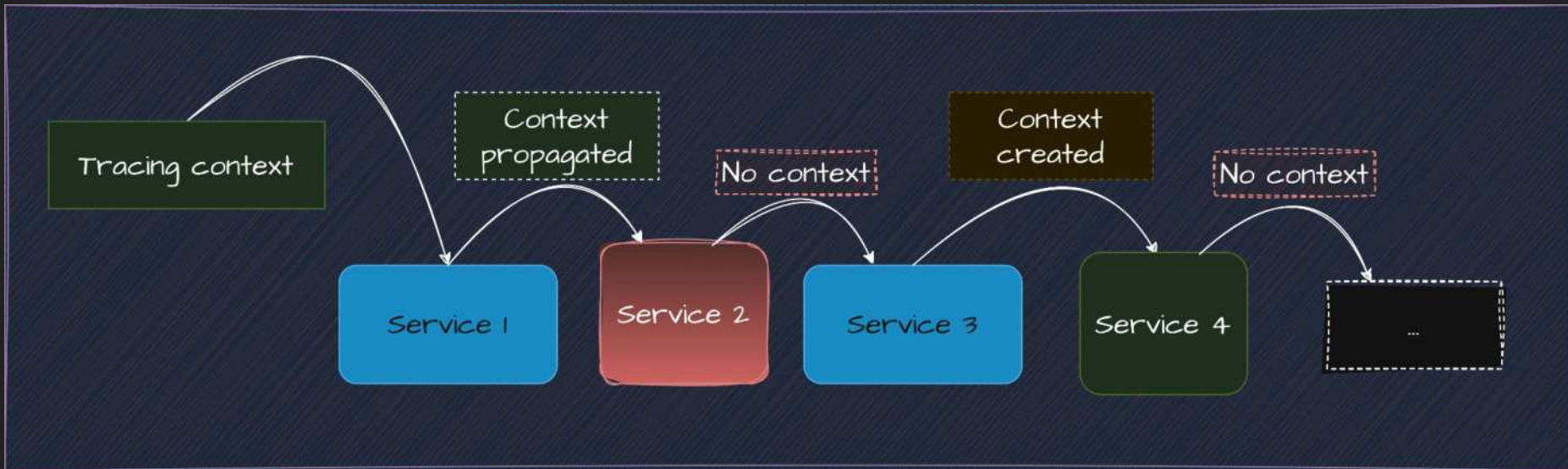


Instrumenting traces in C++

```
void client_impl::send() {  
    ...  
    nghttp_req->on_response( [this,..., span](const ng::client::response& res) {  
  
        span->AddEvent("Response received");  
        span->SetAttribute(ot_trace::SemanticConventions::kHttpResponseStatusCode,res.status_code());  
  
        if (script.validate_answer(ans)) {  
            stats->add_measurement(req.name, elapsed_time, res.status_code());  
            span->SetStatus(ot_trace::StatusCode::kOk);  
        } else {  
            stats->add_error(req.name, res.status_code());  
            span->SetStatus(opentelemetry::trace::StatusCode::kError);  
            queue->cancel_script();  
        }  
        span->End();  
    }  
}
```



Instrumenting traces: Context Propagation!



Instrumenting traces: Context Propagation

Tracing context

Trace Context

W3C Recommendation 23 November 2021

▼ More details about this document

This version:

<https://www.w3.org/TR/2021/REC-trace-context-1-20211123/>

Latest published version:

<https://www.w3.org/TR/trace-context-1/>

Latest editor's draft:

<https://w3c.github.io/trace-context/>

TABLE OF CONTENTS

Abstract

Status of This Document

Conformance

1.

Overview

2.

Problem Statement

2.1

Solution

2.2

Design Overview

2.3

Trace Context HTTP Headers Format

3.

VictoriaMetrics

Instrumenting traces: Context Propagation



§ 3.2 Traceparent Header

The `traceparent` HTTP header field identifies the incoming request in a tracing system. It has four fields:

- `version`
- `trace-id`
- `parent-id`
- `trace-flags`

1.

Overview

2.

Problem Statement

2.1

Solution

2.2

Design Overview

2.3

Trace Context HTTP Headers Format

3.

VictoriaMetrics

Instrumenting traces in C++: Context Propagation!

```
// set global propagator
opentelemetry::context::propagation::GlobalTextMapPropagator::SetGlobalPropagator(
    nostd::shared_ptr<opentelemetry::context::propagation::TextMapPropagator>(
        new opentelemetry::trace::propagation::HttpTraceContext()));

// get global propagator
opentelemetry::trace::propagation::HttpTextMapCarrier<opentelemetry::ext::http::client::Headers> carrier;
auto propagator = opentelemetry::context::propagation::GlobalTextMapPropagator::GetGlobalPropagator();
```

Instrumenting

```
// set global propagator
opentelemetry::context::propagator::set(
    nostd::shared_ptr<opentelemetry::propagator::Text>
        new opentelemetry::propagator::Text)
```

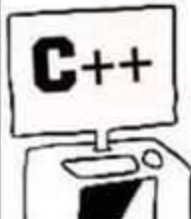
```
// get global propagator
opentelemetry::trace::propagator::get_global_propagator()
auto propagator = opentelemetry::trace::propagator::get_global_propagator()
```

ation!

I hate myself



Wow!
I hate this more



Instrumenting traces in C++: Context Propagation!

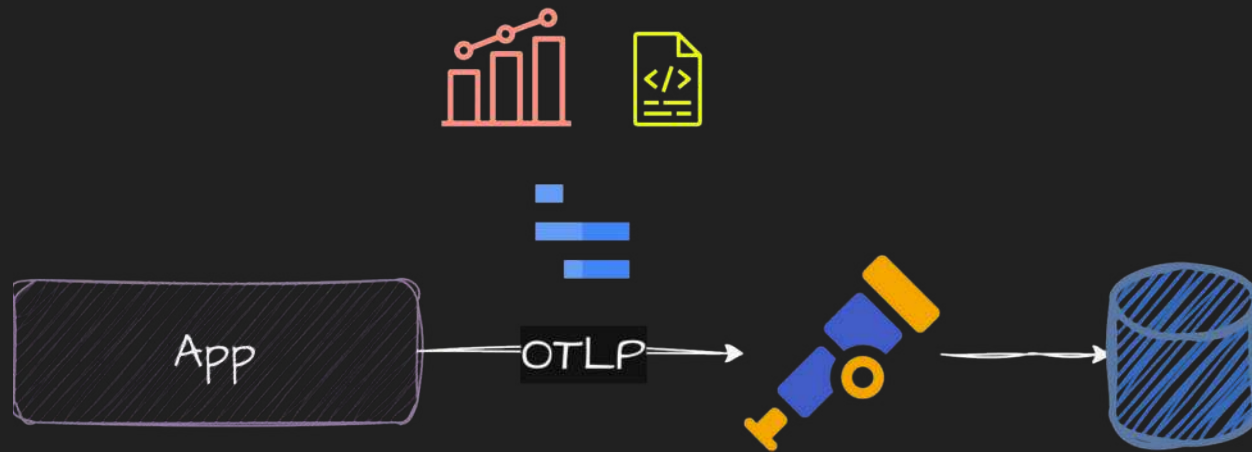
```
//inject context to headers  
auto current_ctx = opentelemetry::context::RuntimeContext::GetCurrent();  
propagator->Inject(carrier, current_ctx);
```

Instrumenting traces in C++: Context Propagation!

```
//inject context to headers
auto current_ctx = opentelemetry::context::RuntimeContext::GetCurrent();
propagator->Inject(carrier, current_ctx);

//Extract headers to context
auto current_ctx = opentelemetry::context::RuntimeContext::GetCurrent();
auto new_context = propagator->Extract(carrier, current_ctx);
auto remote_span = opentelemetry::trace::propagation::GetSpan(new_context);
```

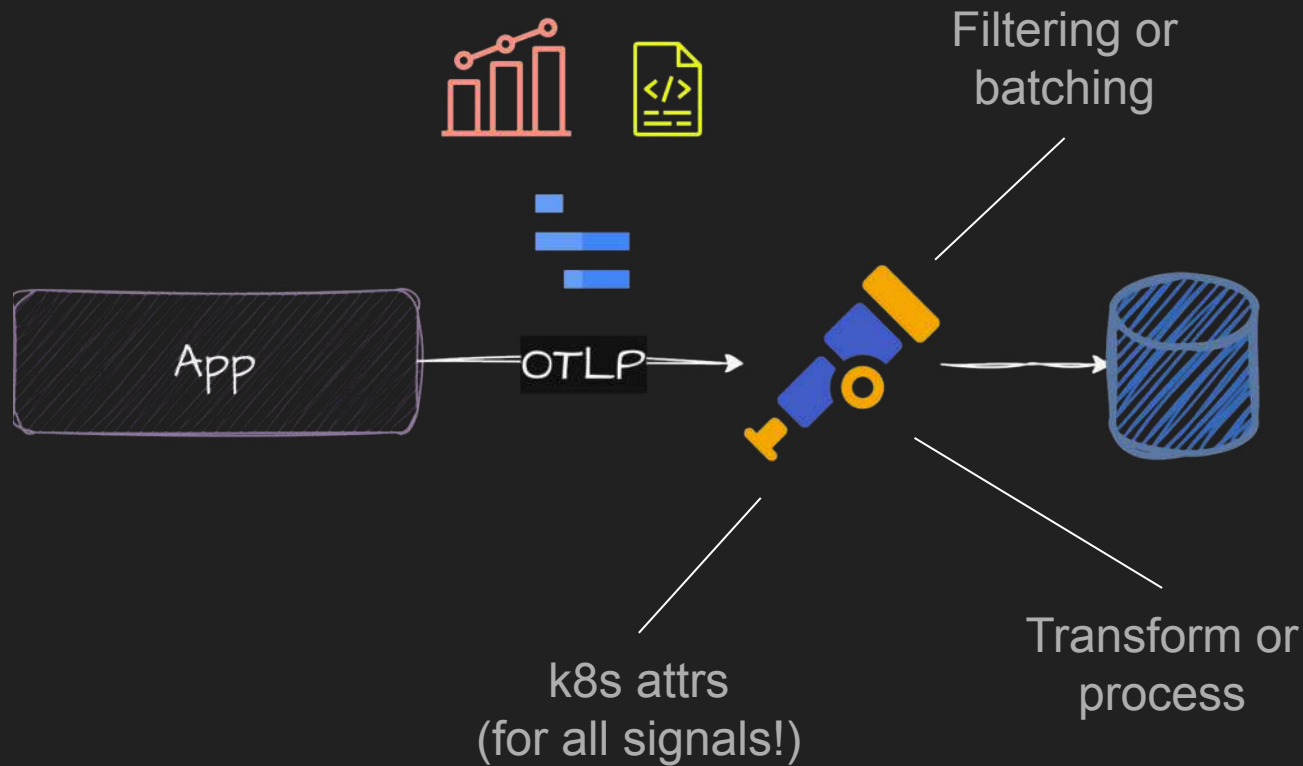
Logs

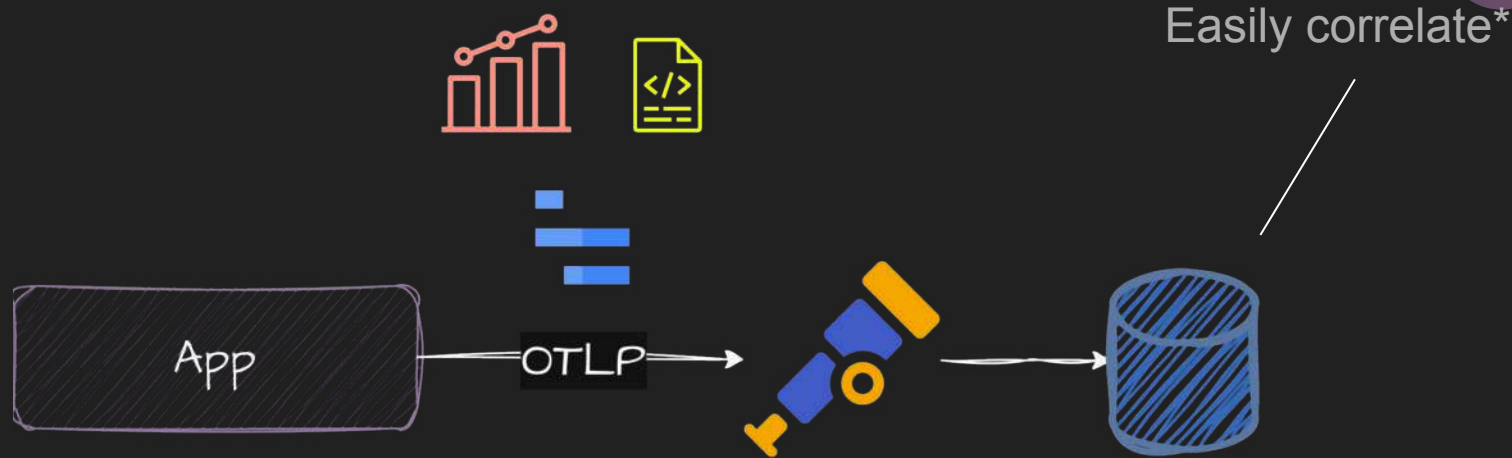




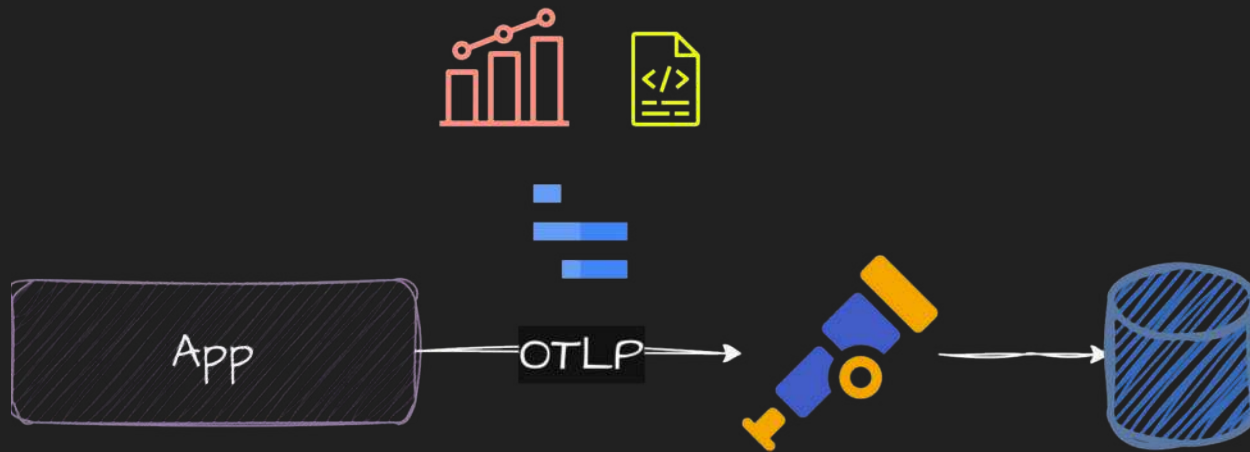
**AFTER ALL...
WHY SHOULDN'T I...**

`std::cout<<"Error here"<<std::endl;`





- By the **time** of execution
- By the execution **context** (TraceId or SpanId)
- By the **origin** of the telemetry (resource: who produced it?!)



- Logging levels
- Activation and deactivation from a single place
- Correlate with traces
- It's trendy



Reuse

Reinvent
the wheel

STD::COUT<<"ERROR HERE"<<STD::ENDL

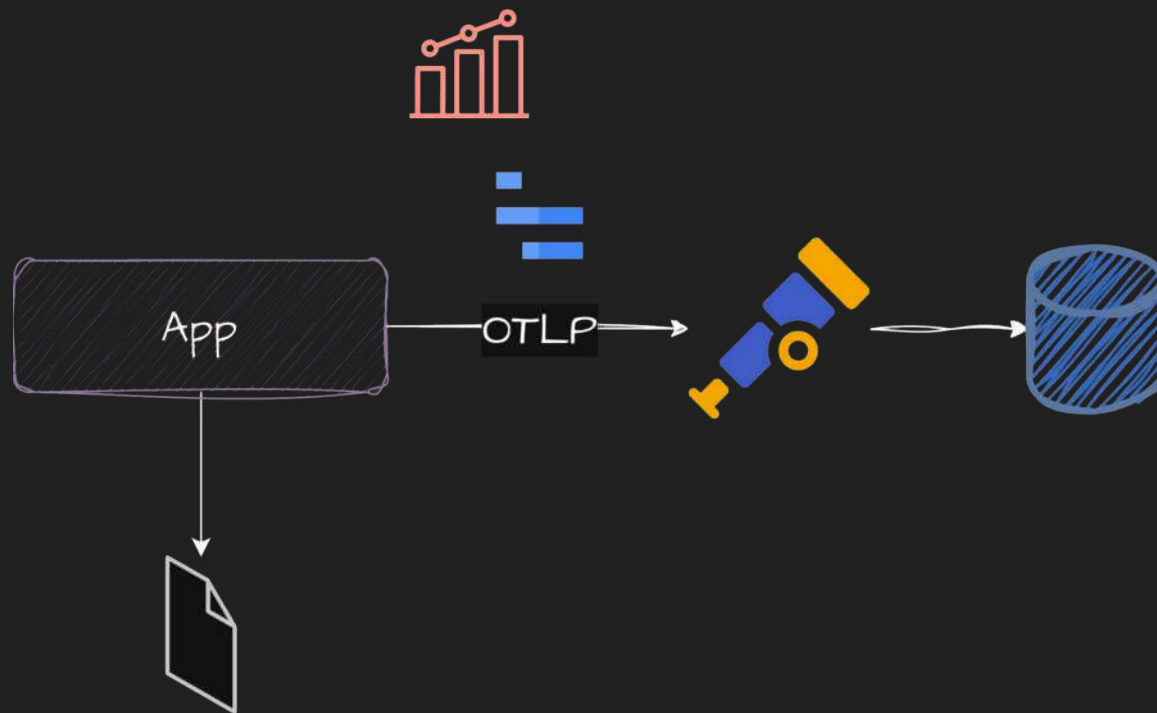
SRE
DAY

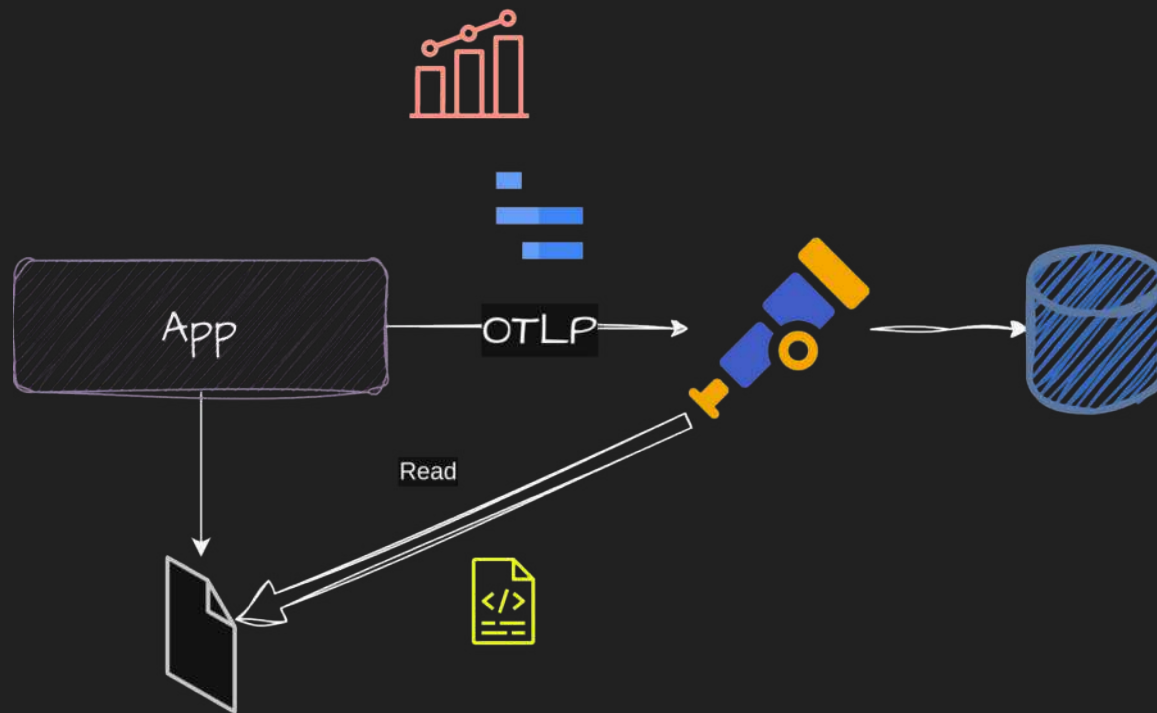


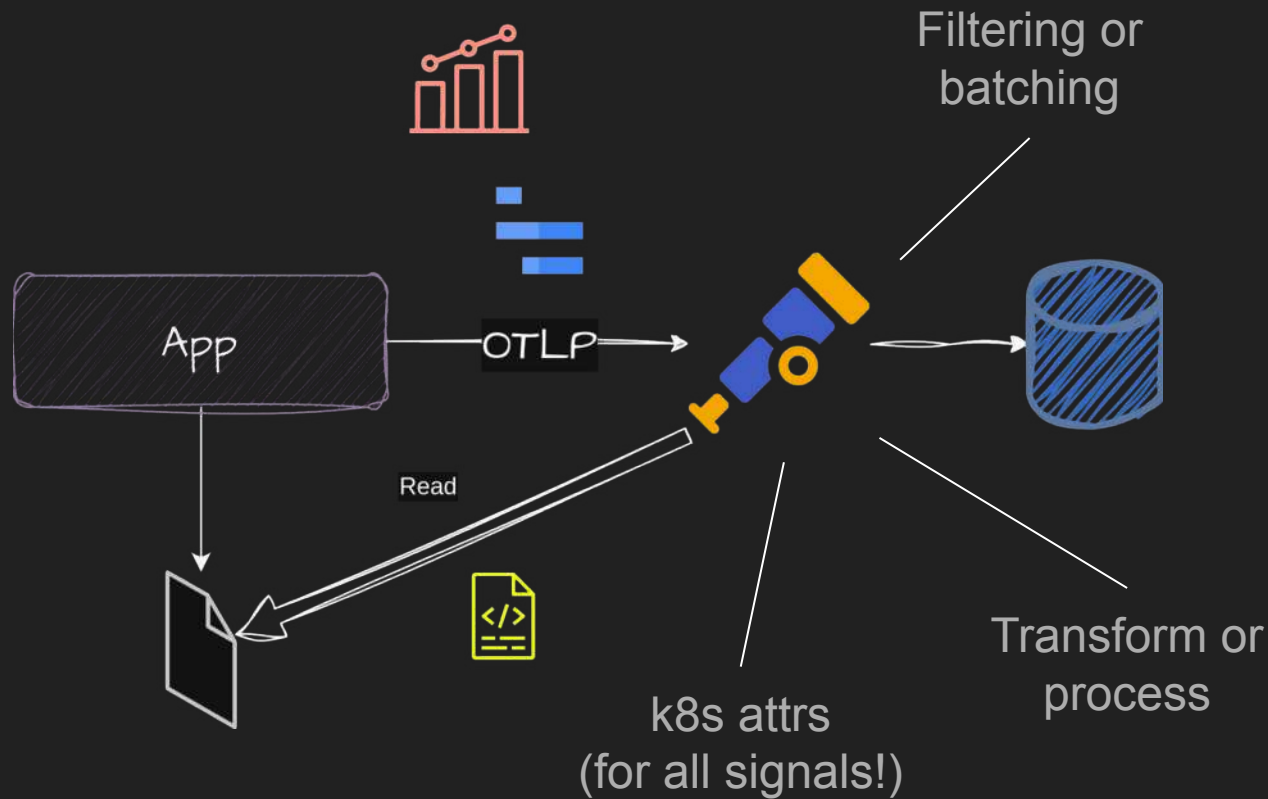
etrics.com



imgflip.com







Instrumenting logs in C++

```
void init_http_logger(const std::string& url, const std::string& token)
{
    opentelemetry::sdk::resource::ResourceAttributes resource_attributes = {{"service.name", name},
                                                                              {"service.version", version}};

    auto resource = opentelemetry::sdk::resource::Resource::Create(resource_attributes);

    opentelemetry::exporter::otlp::OtlpHttpLogRecordExporterOptions loggerOptions;
    loggerOptions.url = url;
    loggerOptions.http_headers.insert(std::make_pair<const std::string, std::string>("Authorization", token));
    loggerOptions.content_type = opentelemetry::exporter::otlp::HttpRequestContentType::kBinary;

    auto exporter = opentelemetry::exporter::otlp::OtlpHttpLogRecordExporterFactory::Create(loggerOptions);

    auto processor = logs_sdk::SimpleLogRecordProcessorFactory::Create(std::move(exporter));
    std::vector<std::unique_ptr<logs_sdk::LogRecordProcessor>> processors;
    processors.push_back(std::move(processor));

    ...
}
```

Instrumenting logs in C++

```
void init_http_logger(const std::string& url, const std::string& token)
{
    opentelemetry::sdk::resource::ResourceAttributes resource_attributes = {{"service.name", name},
                                                                              {"service.version", version}};
    auto resource = opentelemetry::sdk::resource::Resource::Create(resource_attributes);

    opentelemetry::exporter::otlp::OtlpHttpLogRecordExporterOptions loggerOptions;
    loggerOptions.url = url;
    loggerOptions.http_headers.insert(std::make_pair<const std::string, std::string>("Authorization", token));
    loggerOptions.content_type = opentelemetry::exporter::otlp::HttpRequestContentType::kBinary;

    auto exporter = opentelemetry::exporter::otlp::OtlpHttpLogRecordExporterFactory::Create(loggerOptions);

    auto processor = logs_sdk::SimpleLogRecordProcessorFactory::Create(std::move(exporter));
    std::vector<std::unique_ptr<logs_sdk::LogRecordProcessor>> processors;
    processors.push_back(std::move(processor));

    ...
}
```

Instrumenting logs in C++

```
void init_http_logger(const std::string& url, const std::string& token)
{
    opentelemetry::sdk::resource::ResourceAttributes resource_attributes = {{"service.name", name},
                                                                              {"service.version", version}};
    auto resource = opentelemetry::sdk::resource::Resource::Create(resource_attributes);

    opentelemetry::exporter::otlp::OtlpHttpLogRecordExporterOptions loggerOptions;
    loggerOptions.url = url;
    loggerOptions.http_headers.insert(std::make_pair<const std::string, std::string>("Authorization", token));
    loggerOptions.content_type = opentelemetry::exporter::otlp::HttpRequestContentType::kBinary;

    auto exporter = opentelemetry::exporter::otlp::OtlpHttpLogRecordExporterFactory::Create(loggerOptions);

    auto processor = logs_sdk::SimpleLogRecordProcessorFactory::Create(std::move(exporter));
    std::vector<std::unique_ptr<logs_sdk::LogRecordProcessor>> processors;
    processors.push_back(std::move(processor));

    ...
}
```

Instrumenting logs in C++

```
void init_http_logger(const std::string& url, const std::string& token)
{
    opentelemetry::sdk::resource::ResourceAttributes resource_attributes = {{"service.name", name},
                                                                              {"service.version", version}};

    auto resource = opentelemetry::sdk::resource::Resource::Create(resource_attributes);

    opentelemetry::exporter::otlp::OtlpHttpLogRecordExporterOptions loggerOptions;
    loggerOptions.url = url;
    loggerOptions.http_headers.insert(std::make_pair<const std::string, std::string>("Authorization", token));
    loggerOptions.content_type = opentelemetry::exporter::otlp::HttpRequestContentType::kBinary;

    auto exporter = opentelemetry::exporter::otlp::OtlpHttpLogRecordExporterFactory::Create(loggerOptions);

    auto processor = logs_sdk::SimpleLogRecordProcessorFactory::Create(std::move(exporter));
    std::vector<std::unique_ptr<logs_sdk::LogRecordProcessor>> processors;
    processors.push_back(std::move(processor));

    ...
}
```

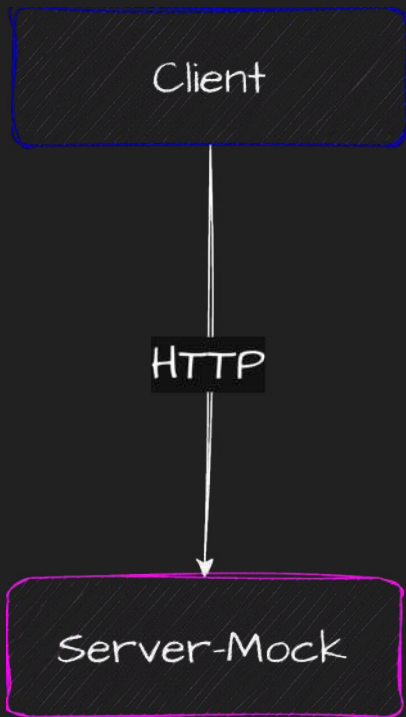
Instrumenting logs in C++

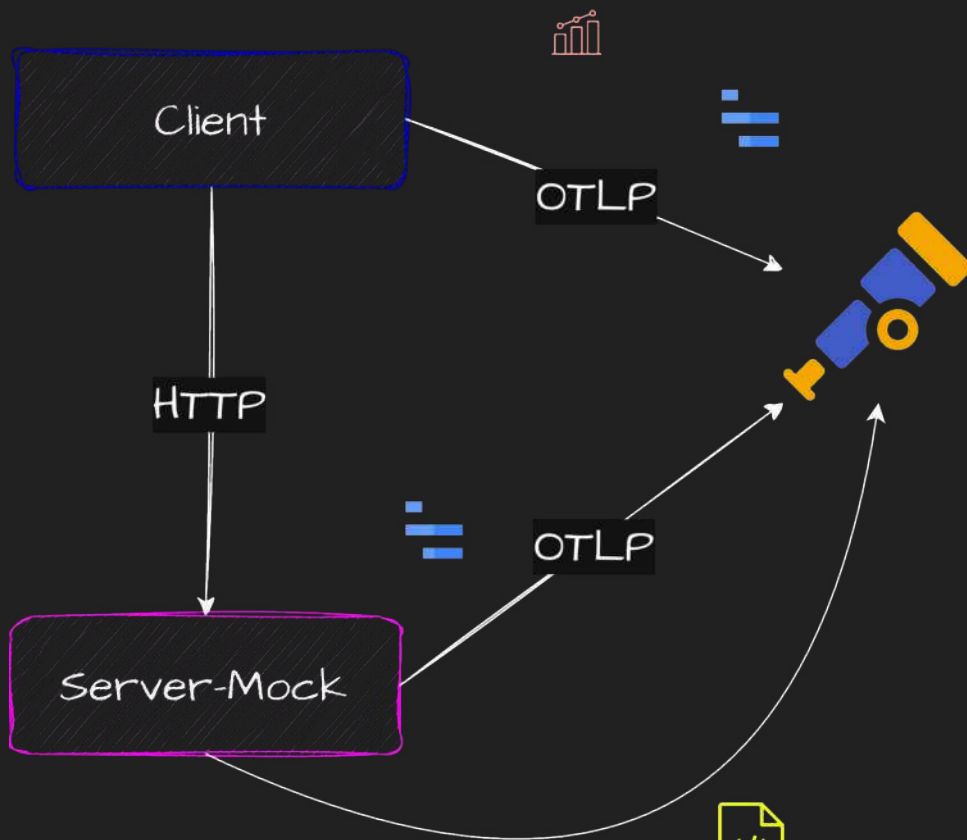
```
void my_function()  
{  
    auto logger = provider->GetLogger("my_client");  
    logger->Debug("Hello OTel!");  
}
```

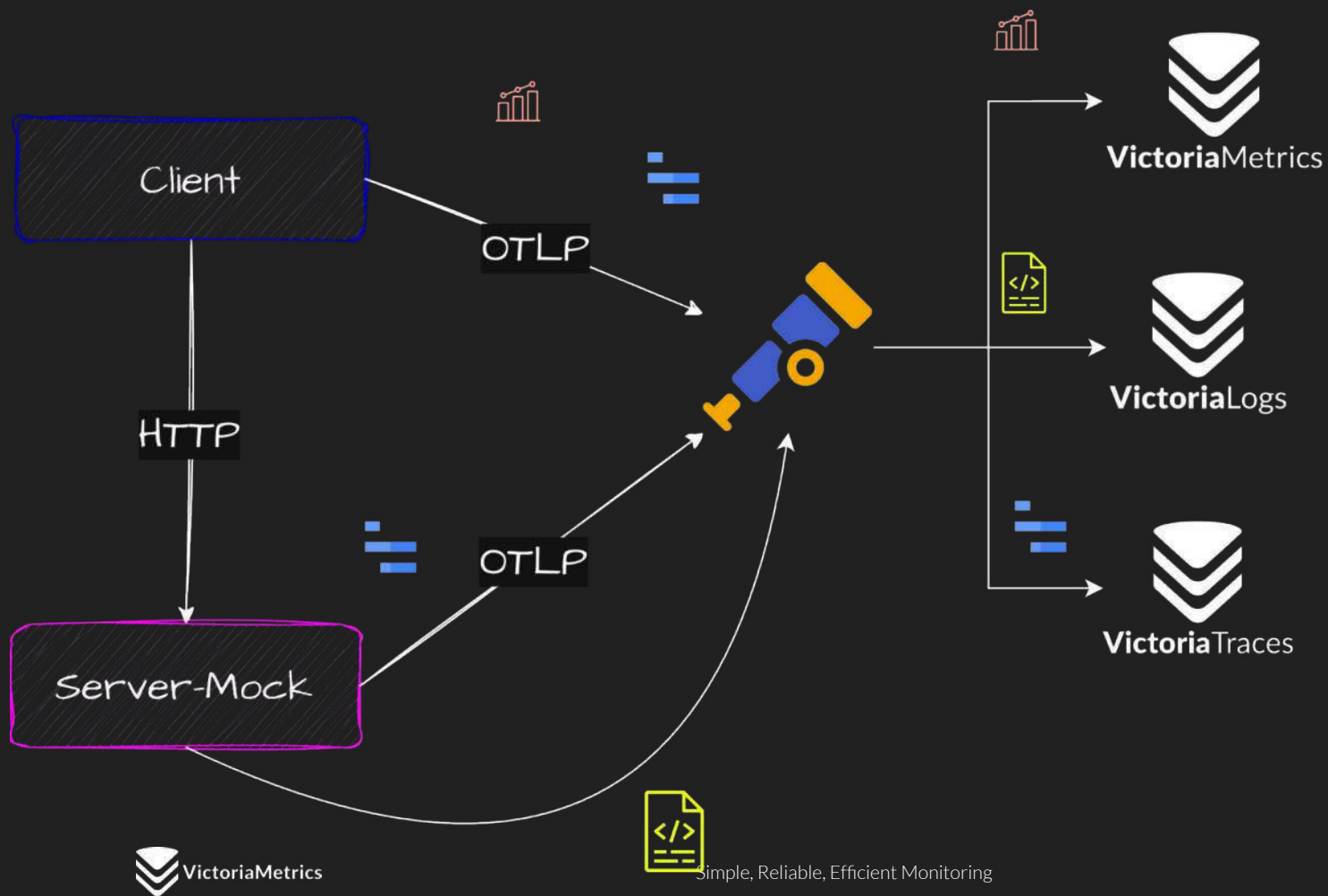
Instrumenting logs in C++

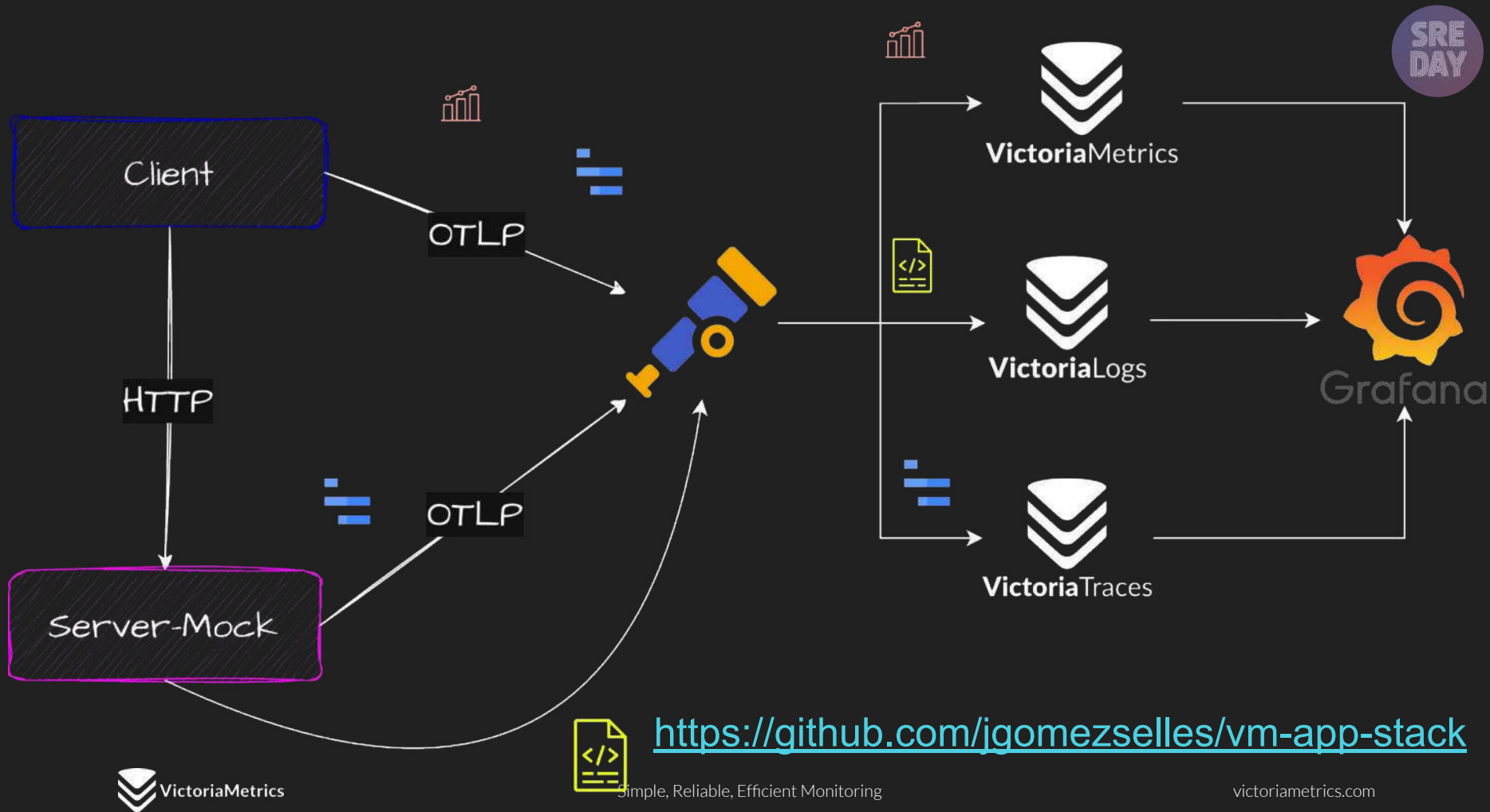
```
void my_function()  
{  
    auto logger = provider->GetLogger("my_client");  
    logger->Debug("Hello OTel!", ctx.trace_id(), ctx.span_id(), ctx.trace_flags());  
}
```

DEMO











Search or jump to...

ctrl+k



Home > Dashboards > Hermes dashboard ☆ 🔗

Add



Last 5 minutes



5s



request_id

All

Refresh dashboard

Success ratio (last) ⓘ



Requests sent/s (last)

46.7

Responses received/s (last)

46.7

RT (last) ⓘ

3.06 ms

Response times (ms) ⓘ





request_id All x x v

Last 5 minutes v

> SPM - Span Metrics (4 panels)

> GET debugging

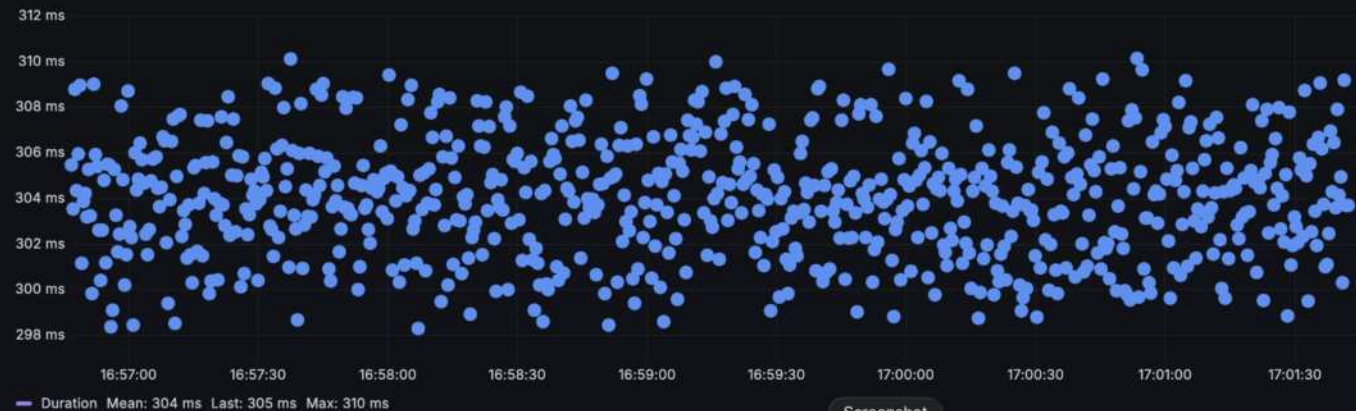
GET logs - server

```
> Request received. URI: /url/example/path?id=id-6483 Method: GET
> Request received. URI: /url/example/path?id=id-6482 Method: GET
> Request received. URI: /url/example/path?id=id-6481 Method: GET
> Request received. URI: /url/example/path?id=id-6480 Method: GET
> Request received. URI: /url/example/path?id=id-6479 Method: GET
> Request received. URI: /url/example/path?id=id-6478 Method: GET
> Request received. URI: /url/example/path?id=id-6477 Method: GET
> Request received. URI: /url/example/path?id=id-6476 Method: GET
> Request received. URI: /url/example/path?id=id-6475 Method: GET
> Request received. URI: /url/example/path?id=id-6474 Method: GET
> Request received. URI: /url/example/path?id=id-6473 Method: GET
> Request received. URI: /url/example/path?id=id-6472 Method: GET
```

GET > 10ms

Trace ID	Trace name	Start time	Duration
c6fe17ea5c7fd2...	hermes: script	2025-10-27 17:01:42	304 ms
a3243862432d...	hermes: script	2025-10-27 17:01:41	304 ms
787bcf89b5359...	hermes: script	2025-10-27 17:01:41	309 ms
046ab8298315...	hermes: script	2025-10-27 17:01:41	300 ms
c1000cd26bf58...	hermes: script	2025-10-27 17:01:40	305 ms
d85866089029...	hermes: script	2025-10-27 17:01:40	304 ms

GET > 10ms



Screenshot

om



OutlineVictoriaTraces

SplitAddLast 1 hourRun query

Query historyShare

QueriesTraces

hermes: script 305.45ms
2025-10-27 17:02:27.506 POST 200

Span Filters 9 spans Prev Next

hermes script (6µs)
post1 (4.64ms)

post1

Service: hermes Duration: 4.64ms
Start Time: 71µs (17:02:27.506) Child Count: 1

Span attributes

error	"false"	
http.request.method	"POST"	
http.response.status_code	"200"	
otel.scope.name	"hermes_client"	
span.kind	"client"	
url.full	"http://server-mock:8080/url/example/path/id-6590"	

> Resource attributes: k8s.deployment.name = hermes k8s.namespace.name = vm-stack k8s.node.name...

Events (3)

> 262µs: event = Request sent

> 4.61ms: event = Response received

> 4.63ms: event = Body received

Event timestamps are relative to the start time of the full trace.

SpanID: 4f882efce16c1c2

ServerMock http.server (3.64ms)
http.server
Screenshot

Service: ServerMock Duration: 3.64ms
Start Time: 647µs (17:02:27.507)



hermes: script 25c31f4 Find... Trace Timeline Archive Trace



Service & Operation

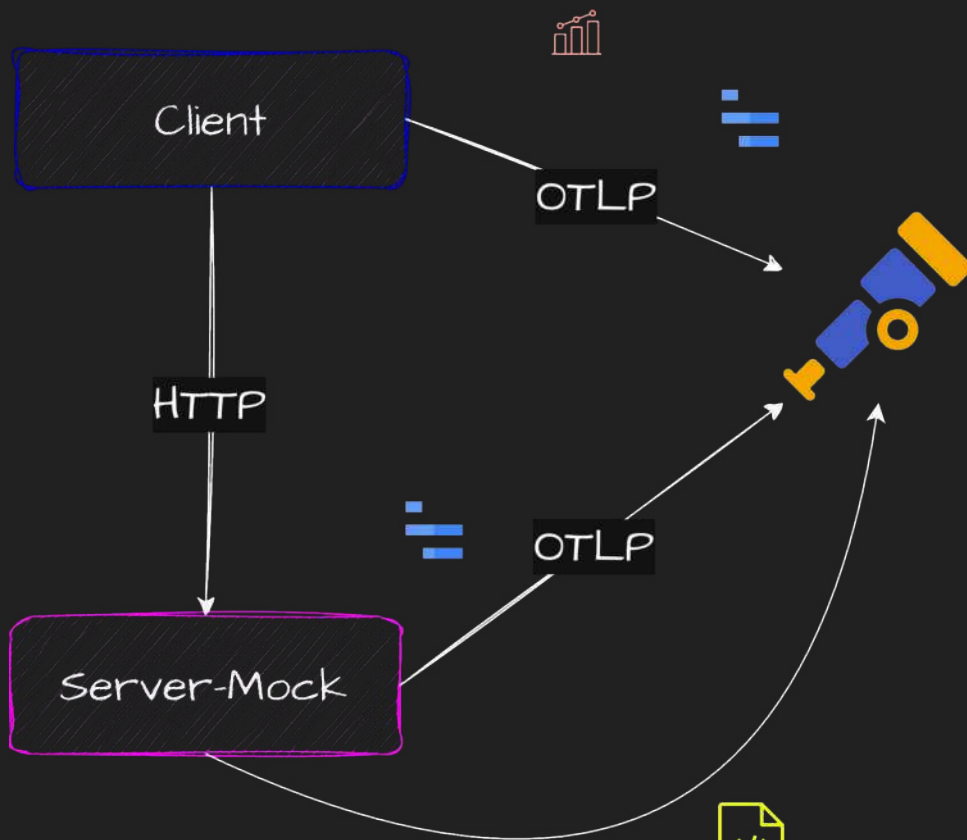
Service & Operation	0µs	8.41ms	16.81ms	25.22ms	33.62ms
otel.status_code	OK				
span.kind	client				
url.full	http://server-mock:8080/url/example/path/id-652				
Process					
otel.library.name	hermes_client				
telemetry.sdk.language	cpp				
telemetry.sdk.name	opentelemetry				
telemetry.sdk.version	1.12.0				
Logs (3)					
290µs					
event	Request sent				
3.51ms					
event	Response received				
3.54ms: event = Body received					

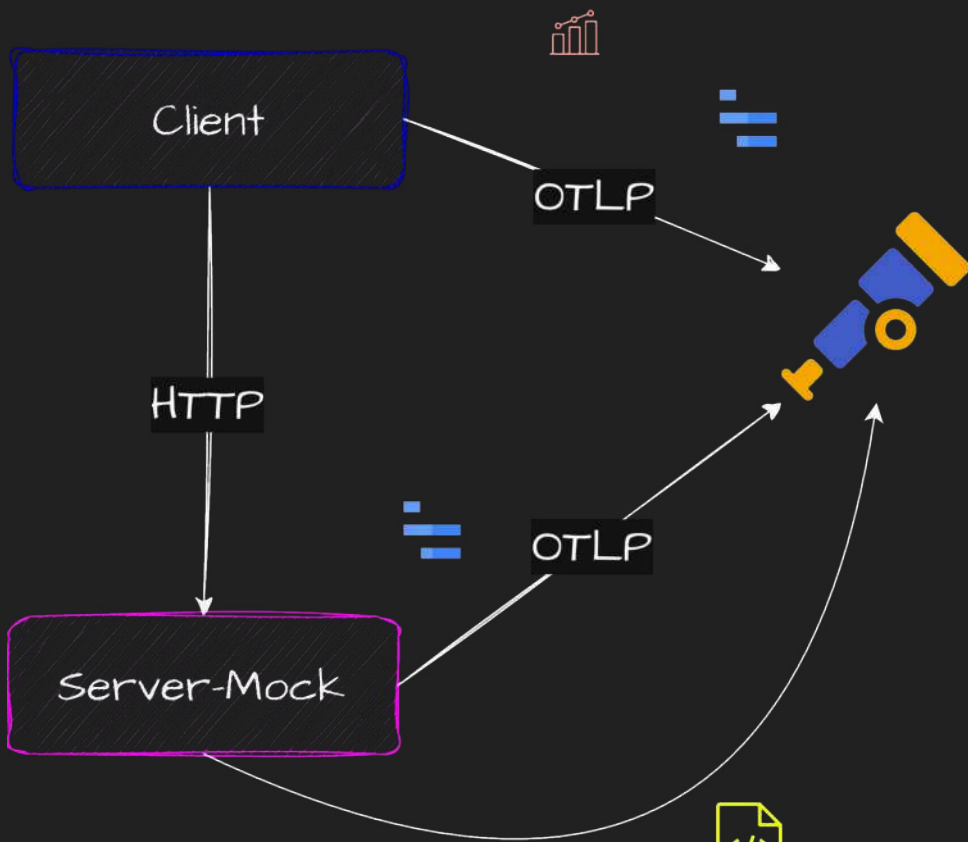
Log timestamps are relative to the start time of the full trace.

SpanID: 6c0e141853d448ea



SpanMetrics





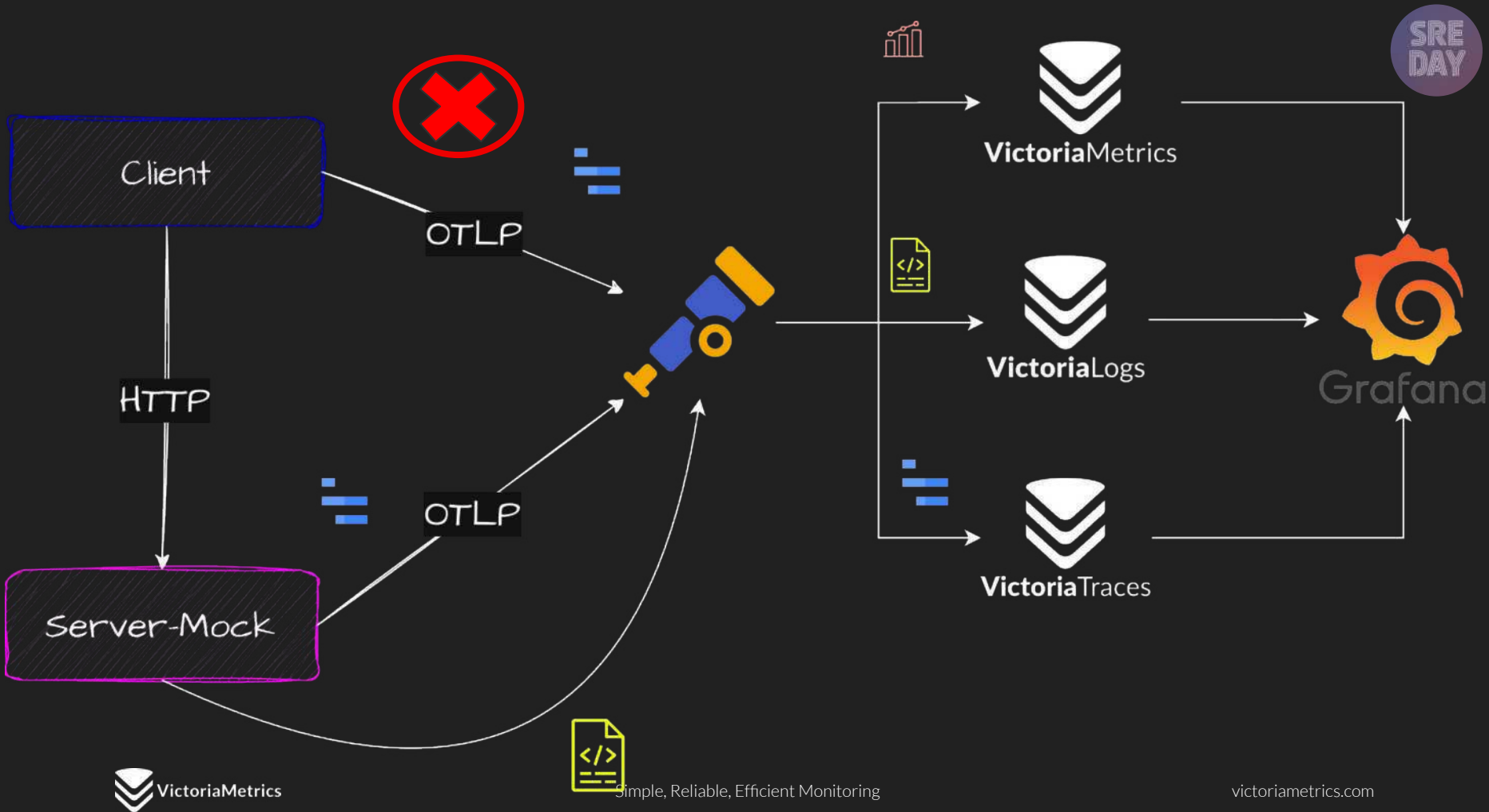
○○○

```
1 connectors:
2   spanmetrics: {}
3
4 service:
5   extensions: [health_check]
6   pipelines:
7     metrics:
8       receivers: [otlp]
9       processors: []
10      exporters: [otlphttp/victoriametrics]
11   traces:
12     receivers: [otlp]
13     processors: []
14     exporters: [spanmetrics, otlp/jaeger, otlphttp/victoriatraces]
15   metrics/spanmetrics:
16     receivers: [spanmetrics]
17     exporters: [otlphttp/victoriametrics]
18   logs:
19     receivers: [otlp]
20     processors: []
21     exporters: [otlphttp/victorialogs]
```



```
1 connectors:
2   spanmetrics: {}
3
4 service:
5   extensions: [health_check]
6   pipelines:
7     metrics:
8       receivers: [otlp]
9       processors: []
10      exporters: [otlphttp/victoriametrics]
11    traces:
12      receivers: [otlp]
13      processors: []
14      exporters: [spanmetrics, otlp/jaeger, otlphttp/victoriatraces]
15    metrics/spanmetrics:
16      receivers: [spanmetrics]
17      exporters: [otlphttp/victoriametrics]
18    logs:
19      receivers: [otlp]
20      processors: []
21      exporters: [otlphttp/victorialogs]
```







Edit

Export

Share

Last 5 minutes



Refresh

5s

request_id All x x v



Responses received/s (last)



RT (last)



SPM - Span Metrics

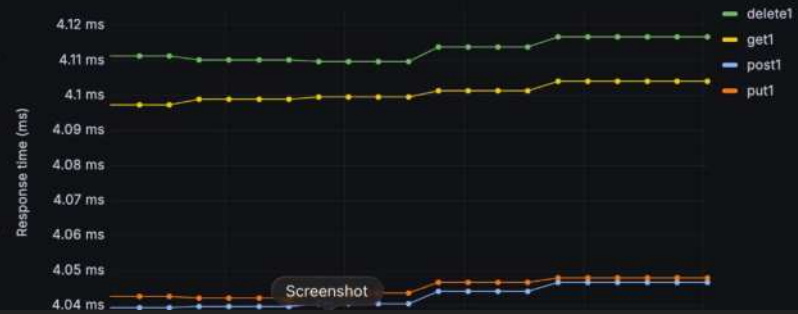
Success ratio (last)



Requests sent/s



Response times (ms)



Screenshot

Show me the log!

```
{
  "resourceLogs": [
    {
      "resource": {
        "attributes": [
          {
            "key": "service.name",
            "value": {
              "stringValue": "my.service"
            }
          }
        ]
      },
      ...
    },
    "scopeLogs": [
      {
        "scope": {
          "name": "my.library",
          "version": "1.0.0",
          "attributes": [
            {
              "key": "my.scope.attribute",
              "value": {
                "stringValue": "some scope attribute"
              }
            }
          ]
        },
        ...
      }
    ]
  }
}
```

```
    "logRecords": [
      {
        "timeUnixNano": "1544712660300000000",
        "observedTimeUnixNano":
          "1544712660300000000",
        "severityNumber": 10,
        "severityText": "Information",
        "traceId":
          "5B8EFFF798038103D269B633813FC60C",
        "spanId": "EEE19B7EC3C1B174",
        "body": {
          "stringValue": "Example log record"
        },
        "attributes": [
          {
            "key": "string.attribute",
            "value": {
              "stringValue": "some string"
            }
          },
          {
            "key": "boolean.attribute",
            "value": {
              "boolValue": true
            }
          }
        ]
      }
    ]
  }
}
```

Watch out!

(these things can get expensive)



Guard Log

\$2,500

Summary

- C++ code can be easily instrumented with OpenTelemetry
 - (and it can be fun and useful!)
- OpenTelemetry decouples SDK from API and tooling
 - Abstracts application code from exporter
 - Easily change between exporters without impacting the code
- Semantic conventions help
- OTel collector to the rescue!
 - Collecting logs
 - Format/Transform
 - SpanMetrics

Get Started Here

Signup



VictoriaMetrics
Cloud

console.victoriametrics.cloud/signup

Documentation



VictoriaMetrics Docs

docs.victoriametrics.com/victoriametrics-cloud/



VictoriaMetrics
Cloud

Simple, Reliable, Efficient Monitoring

Thanks!

