Discovering the Magic Behind OpenTelemetry Instrumentation

FOSDEM 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

UNIVERSIDAD COMPLUTENSE MADRID

github.com/jgomezselles

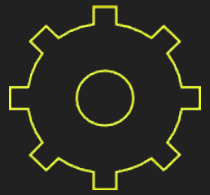
@jgomezselles.bsky.social

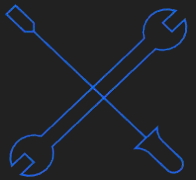
linkedin.com/in/joseluisgomezselles

Instrumentation

OpenTelemetry

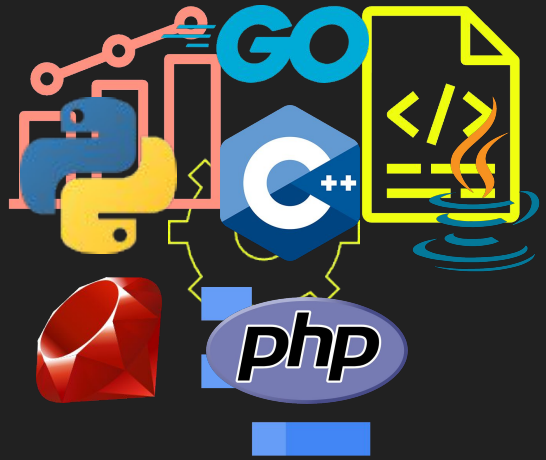
APIs, SDKs and Tools



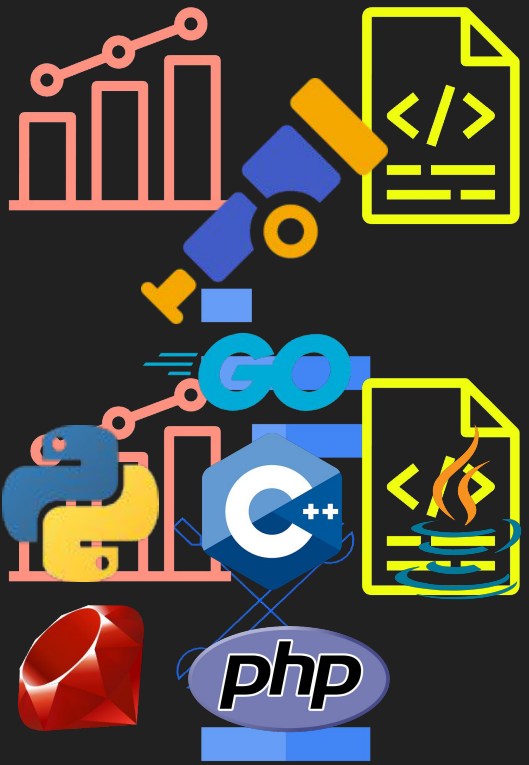


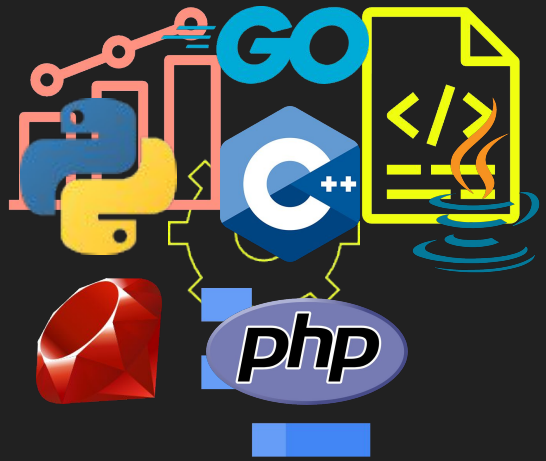
Metrics, Logs and Traces

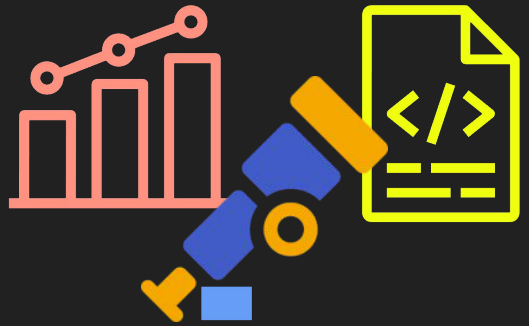
Metrics, Logs and Traces

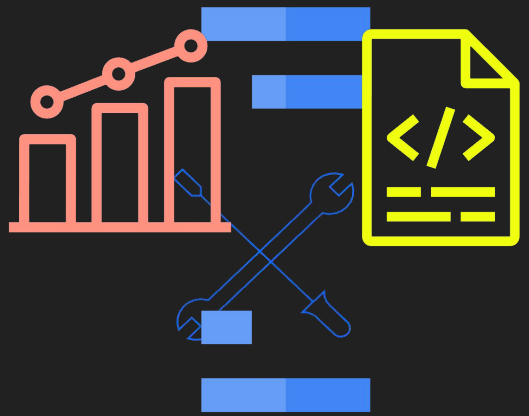


Languages!



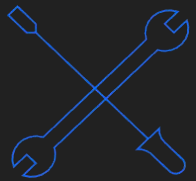




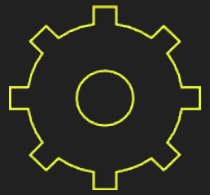


Protocol

Unify



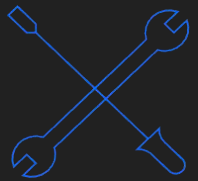
Protocol



Unify

Instrumentation

Manual or zero code!



Protocol

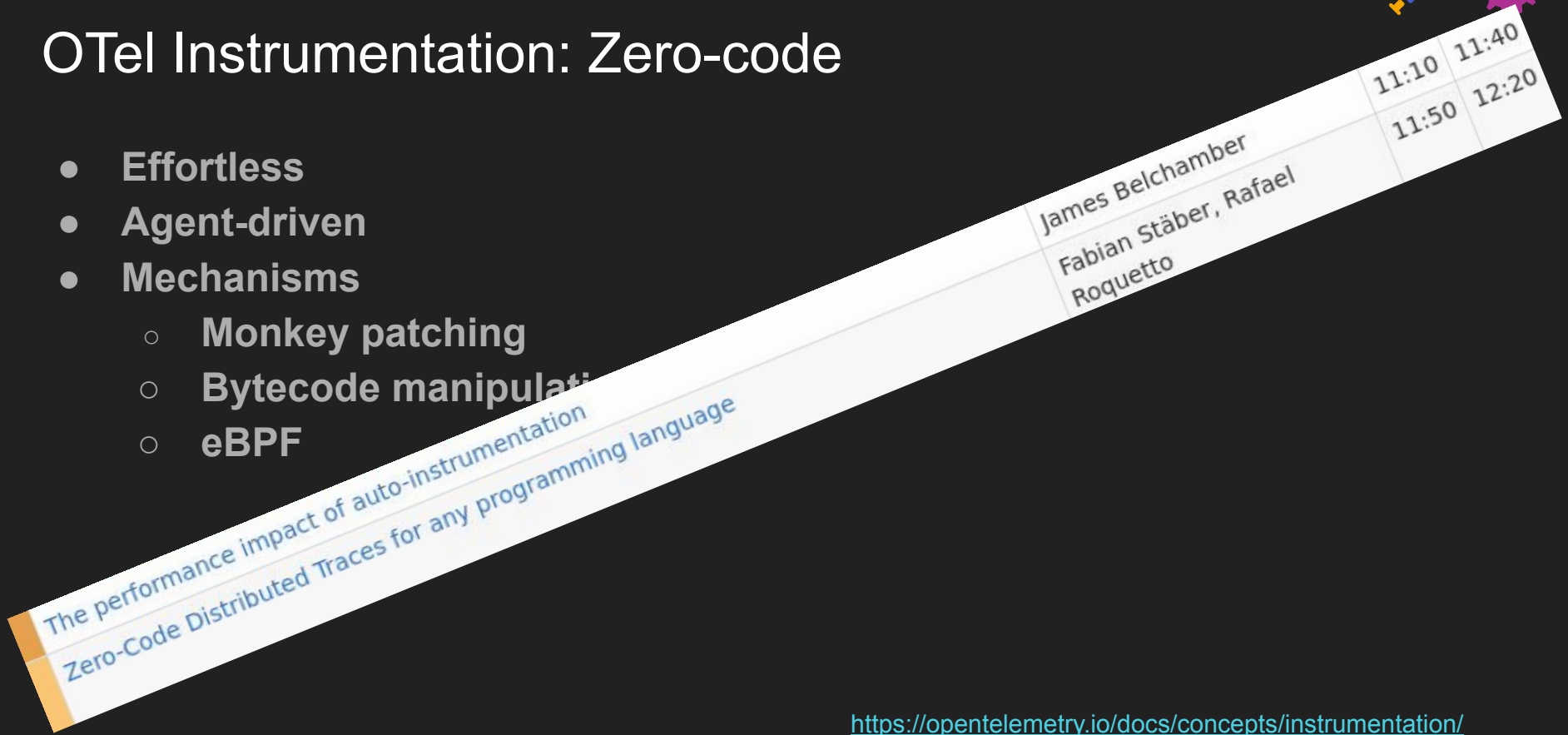
OTel Instrumentation: Zero-code

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

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

OTel Instrumentation: Zero-code

- Effortless
- Agent-driven
- Mechanisms
  - Monkey patching
  - Bytecode manipulati
  - 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 libs and exporters
- Great for edges and libs

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

OTel Instrumentation: Zero-code

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

But…

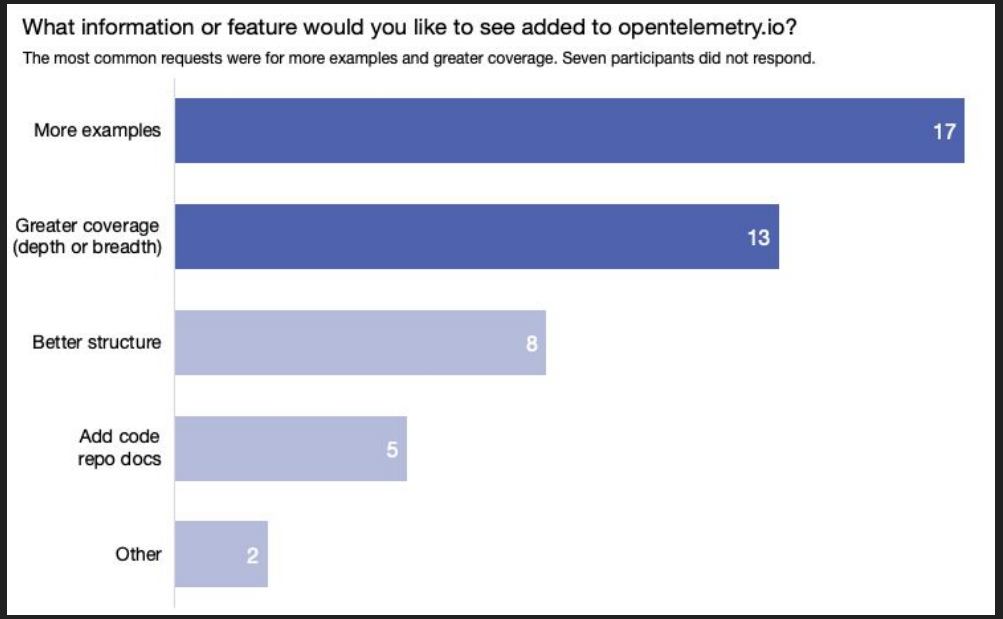
- Not all languages support it
- Hard to control data
  - Costs
  - Performance
- What about my code?

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

Manual instrumentation

Insights from the OpenTelemetry Docs Usability Survey

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



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

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

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
```

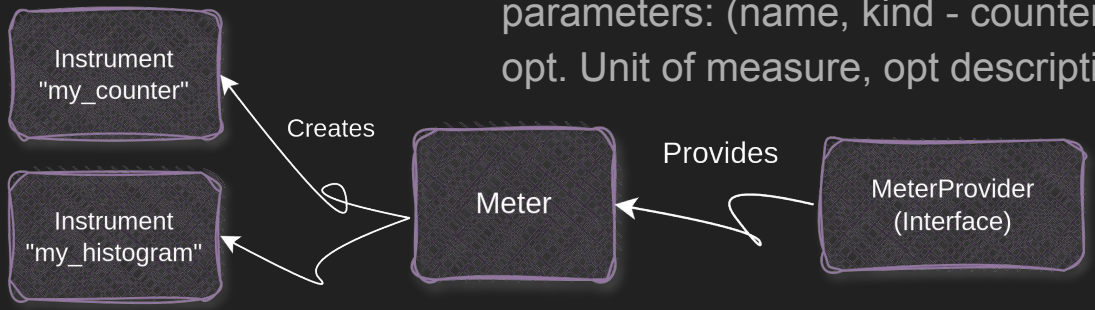
Meter: The meter is responsible for creating Instruments

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" );
```

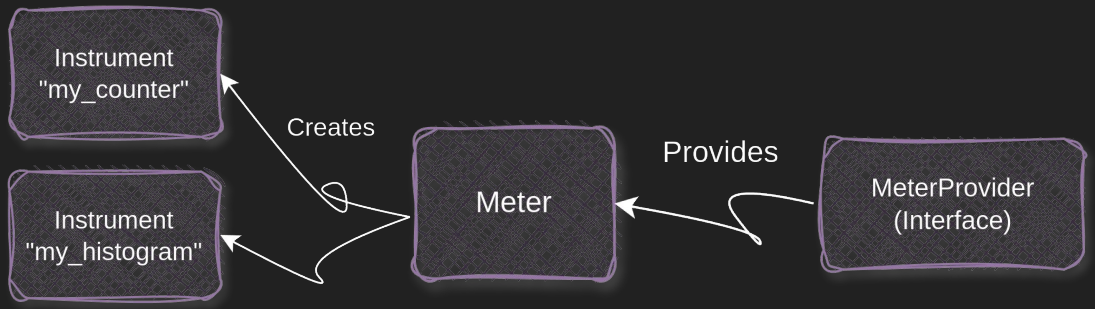
Instruments are used to report Measurements. Each Instrument will have the following parameters: (name, kind - counter, histogram.., opt. Unit of measure, opt description)



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::nostd::unique_ptr<opentelemetry::v1::metrics::Counter<uint64_t>> responses_err;
responses_err = std::move(resp_nok);
```



Instrumenting metrics in C++

New error!

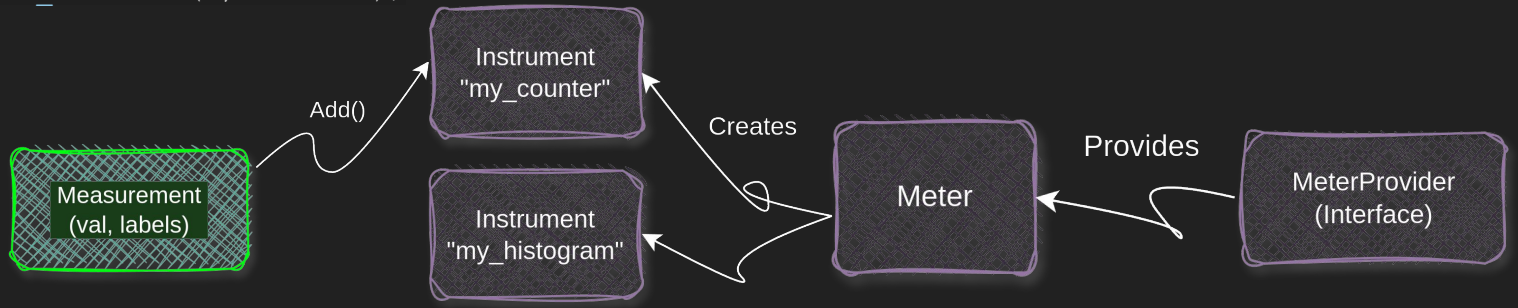
```
void stats::add_error(const std::string& id, const int e)
{
    /*
    Some code here to process the error
    */
    ++responses_err;
}
```

Goal

Instrumenting metrics in C++

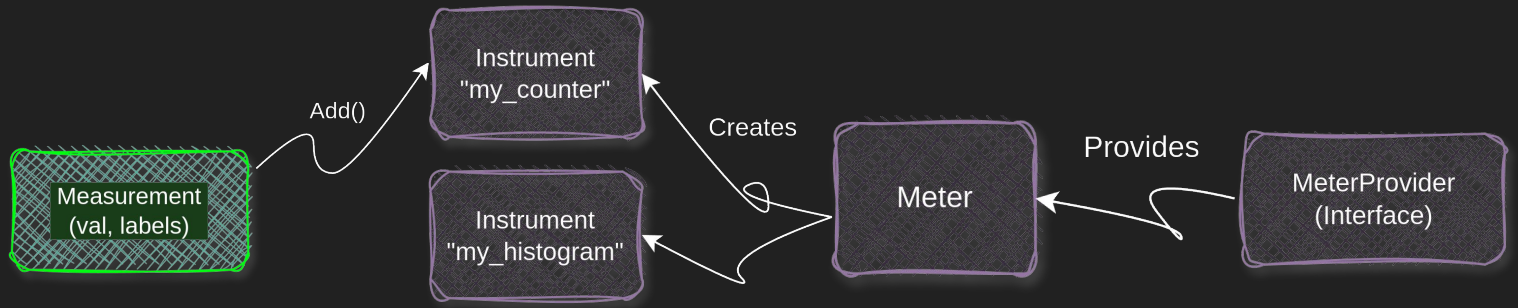
New error!

```
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);
}
```



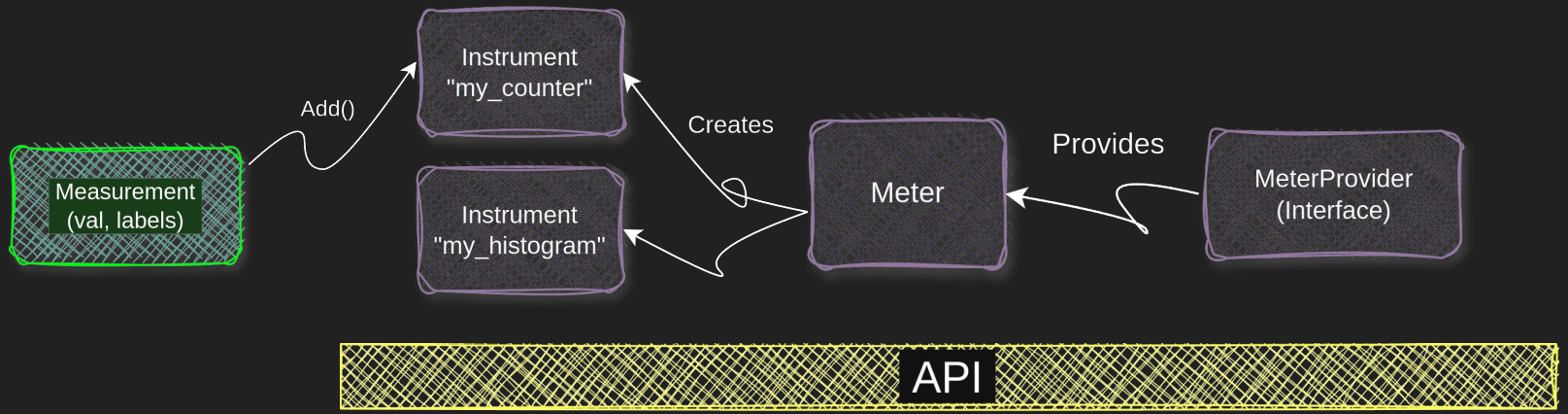
Instrumenting metrics in C++

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

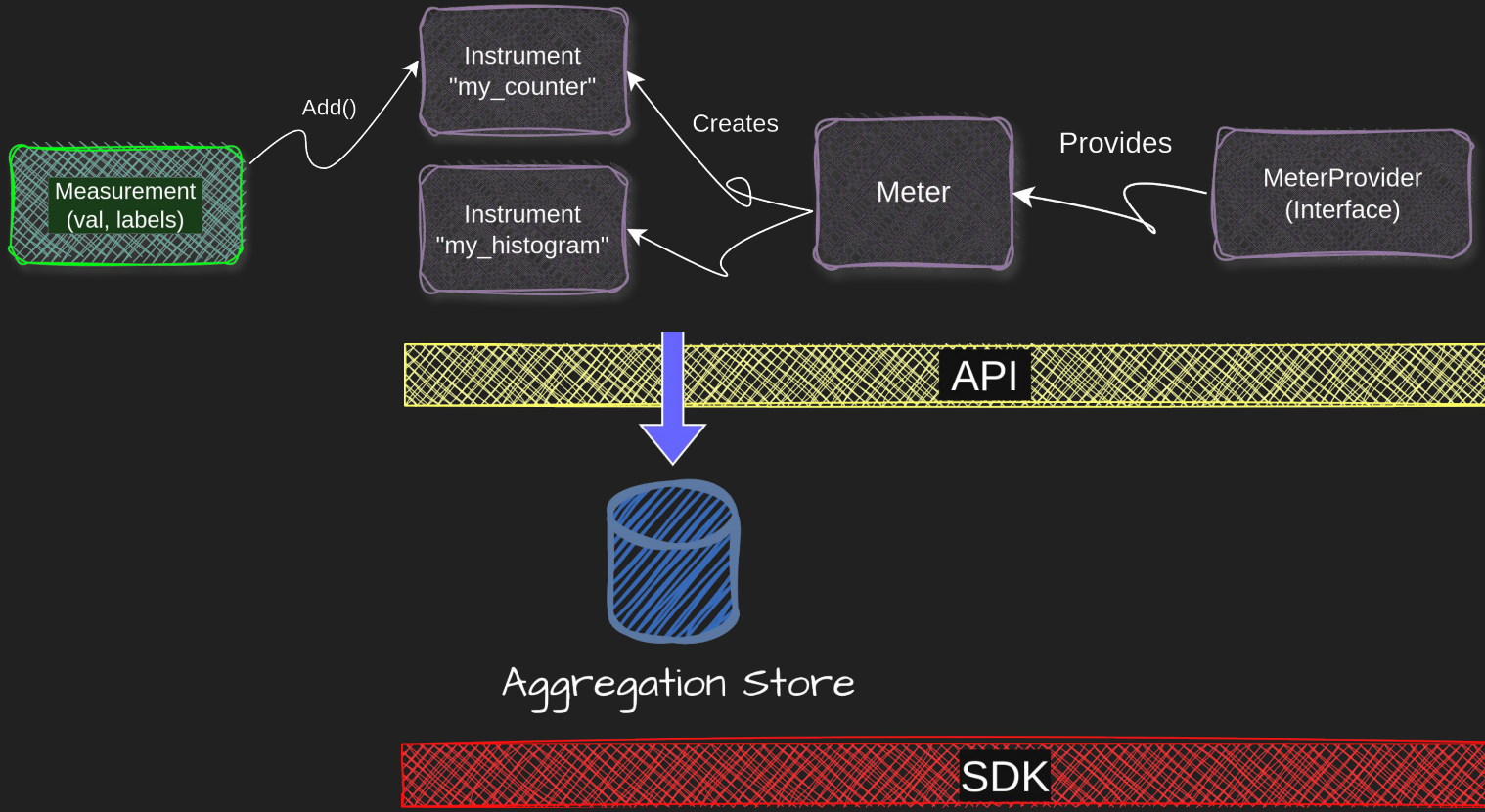


Instrumenting metrics in C++

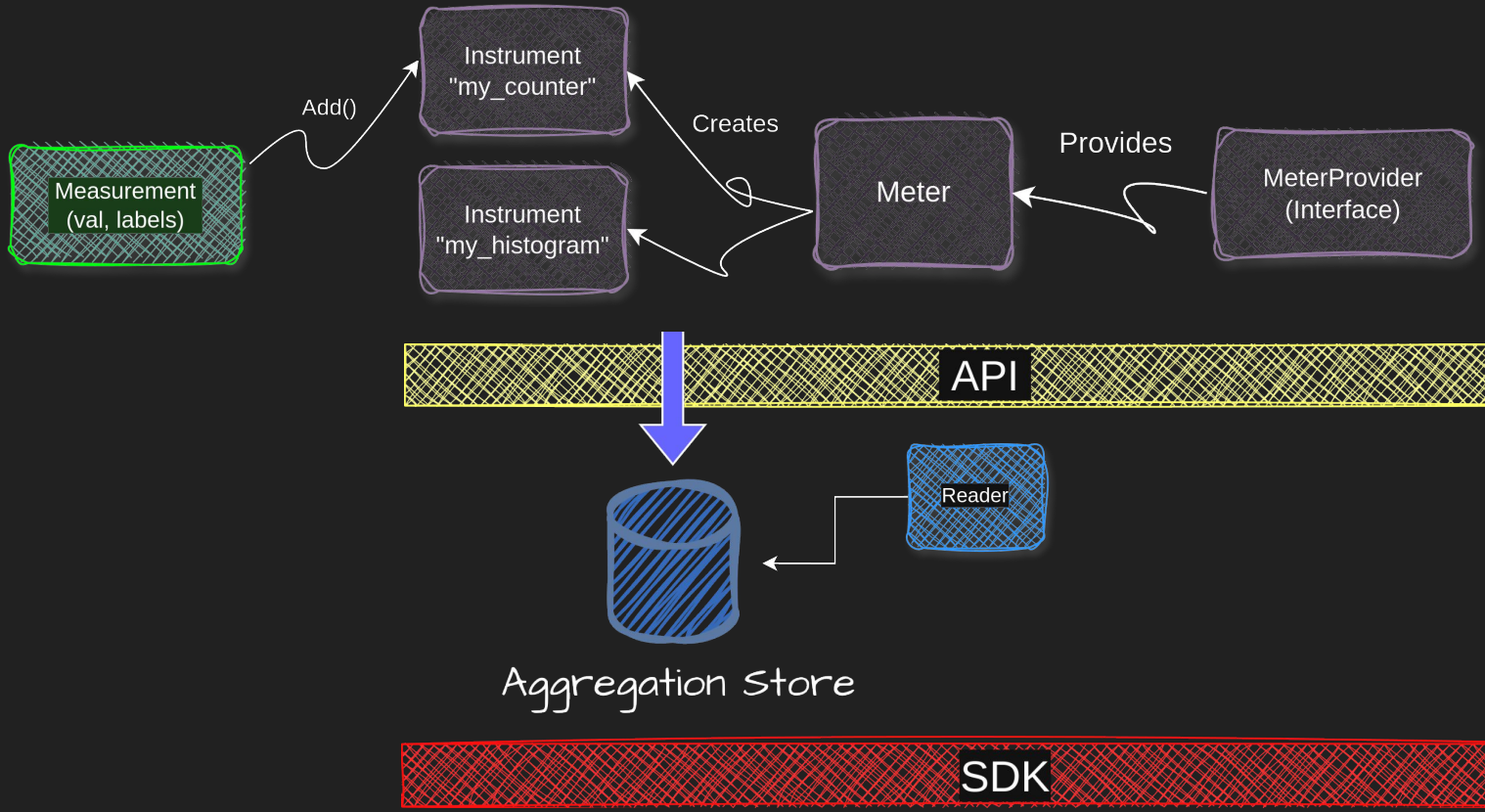
Measurement: represents a data point reported via the metrics API to the SDK



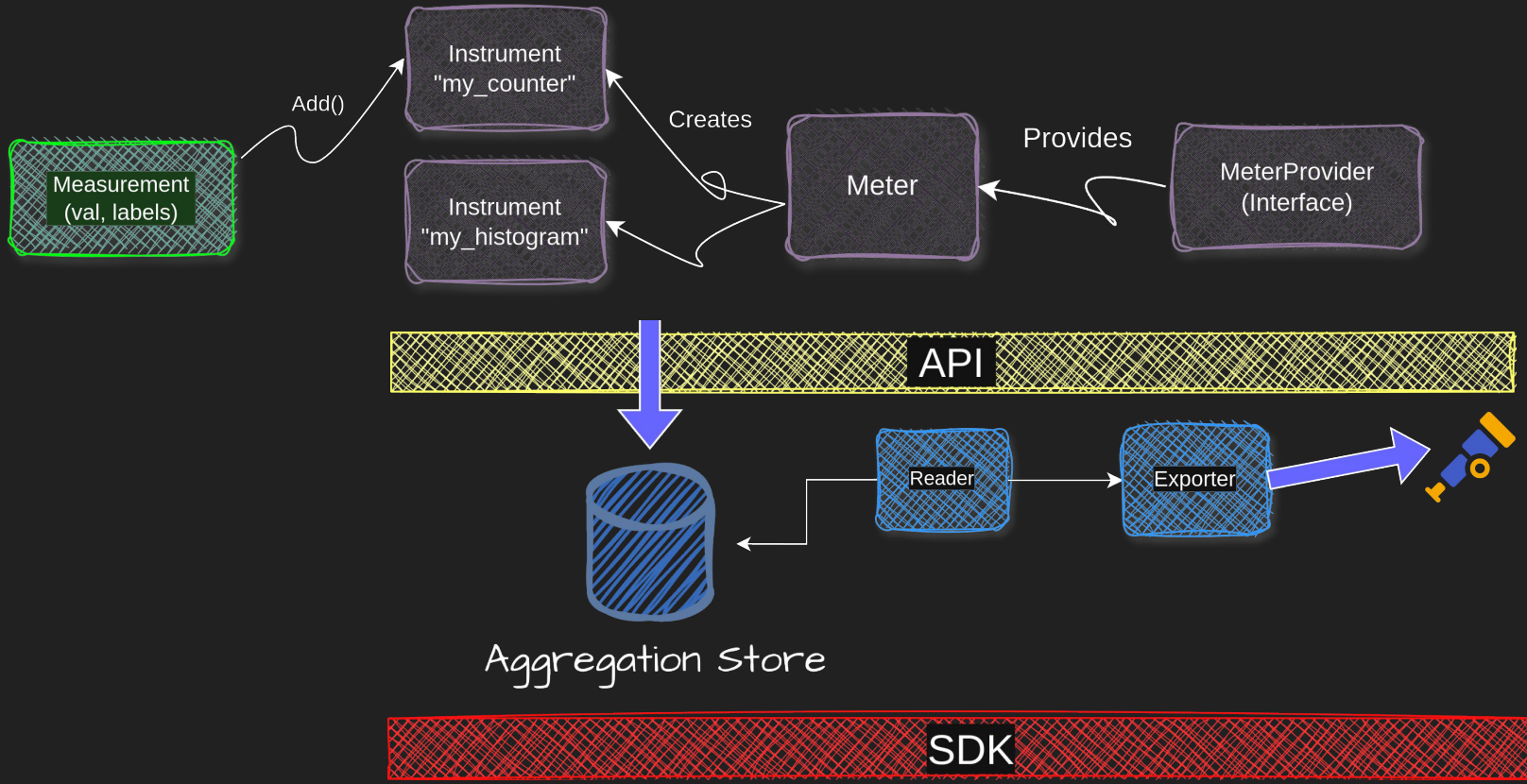
Instrumenting metrics in C++



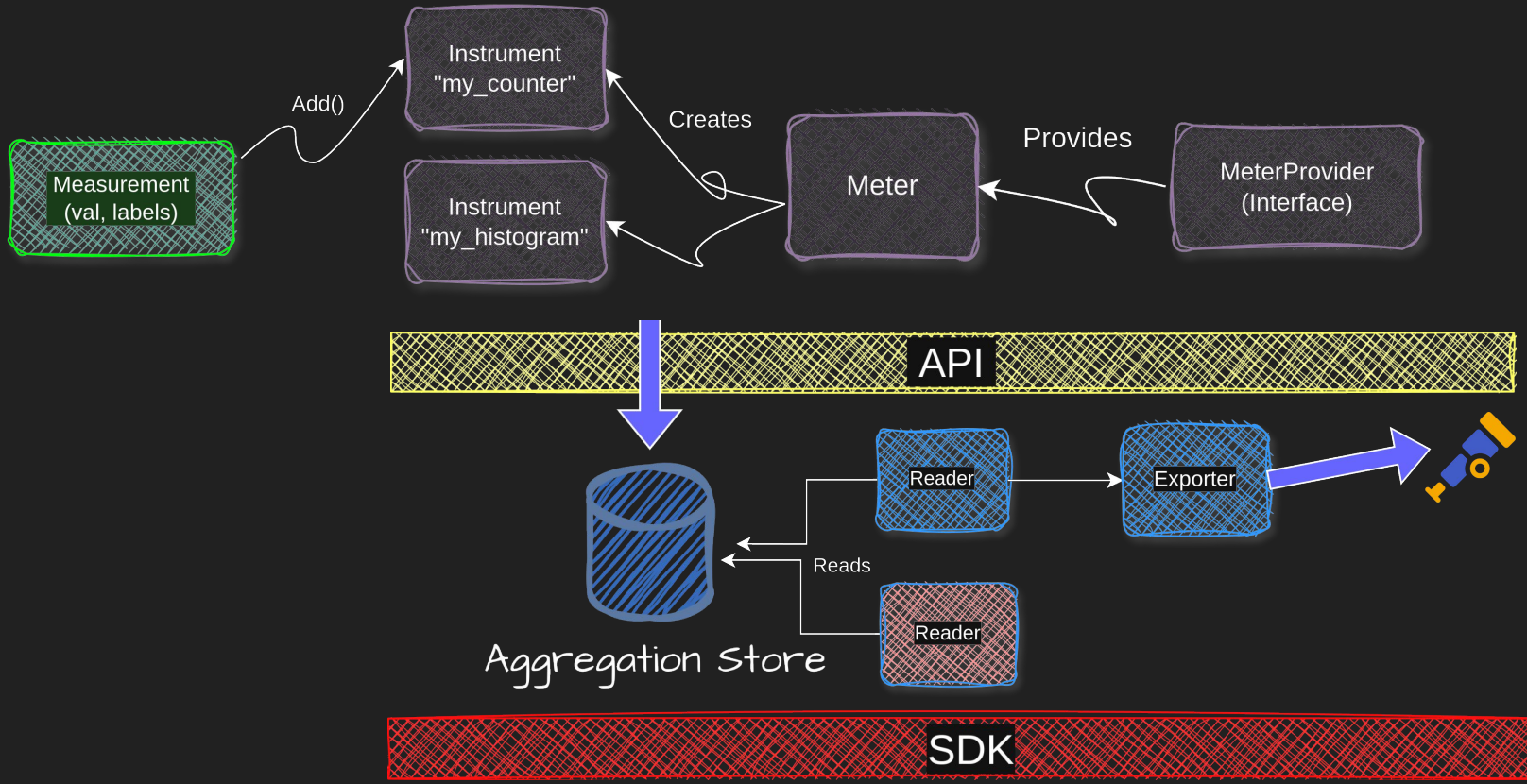
Instrumenting metrics in C++



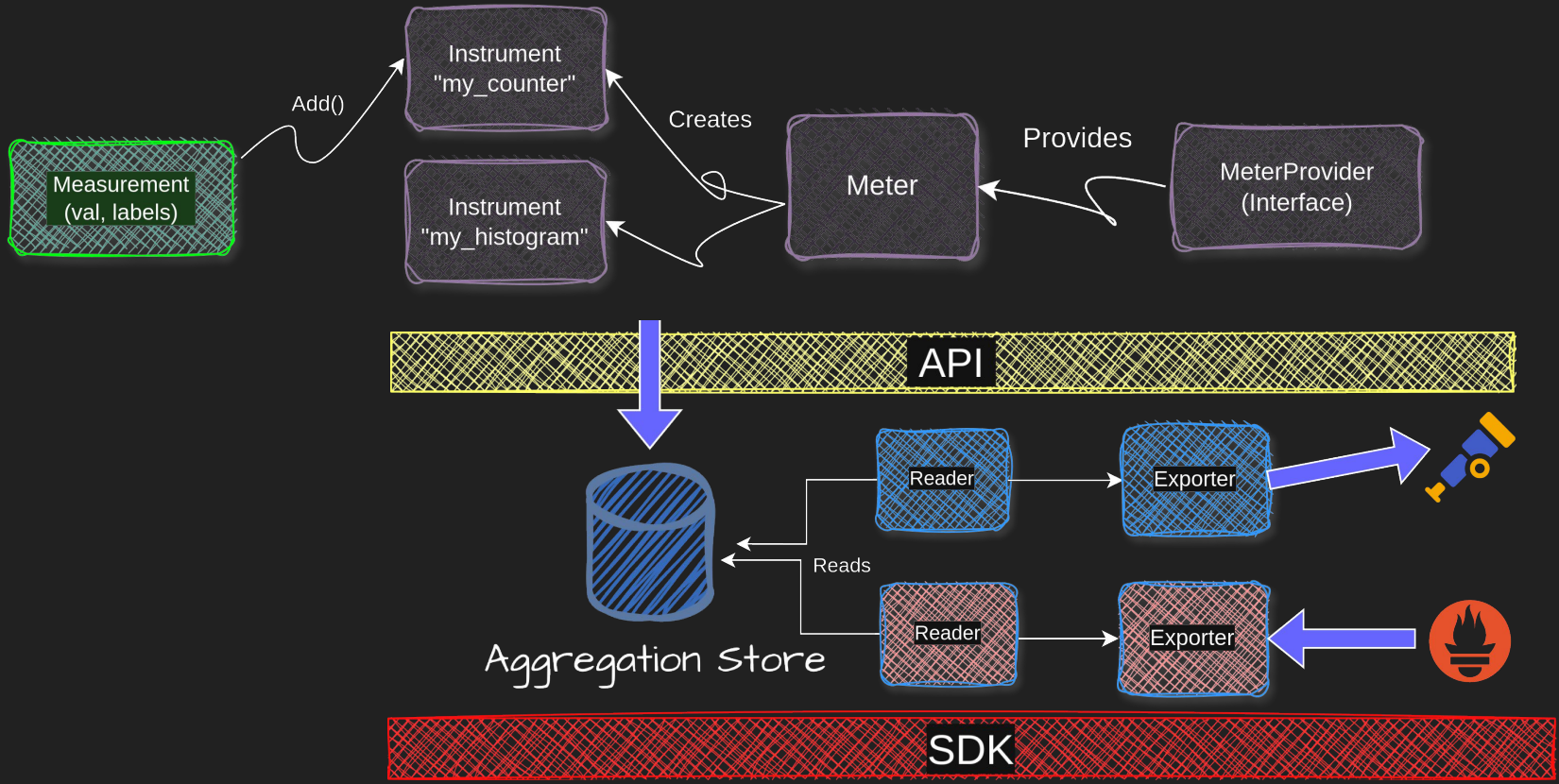
Instrumenting metrics in C++



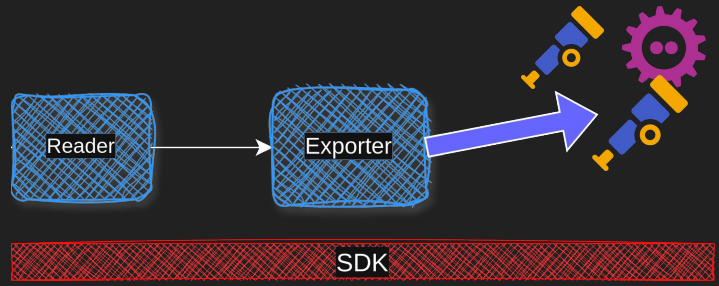
Instrumenting metrics in C++



Instrumenting metrics in C++

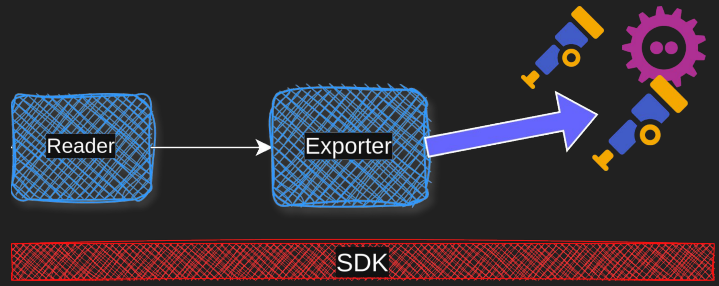


Instrumenting metrics in C++



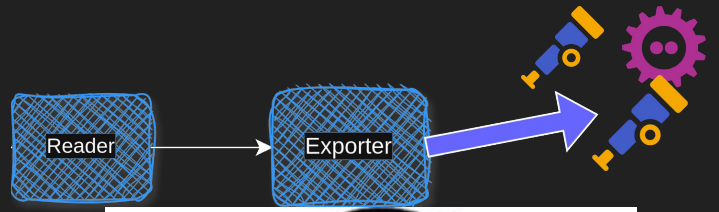

```
void init_metrics_otlp_http(const std::string& url)
{
  sdk_metrics::PeriodicExportingMetricReaderOptions reader_options;
  reader_options.export_interval_millis = std::chrono::milliseconds(1000);
  reader_options.export_timeout_millis = std::chrono::milliseconds(500);

  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);
}
```

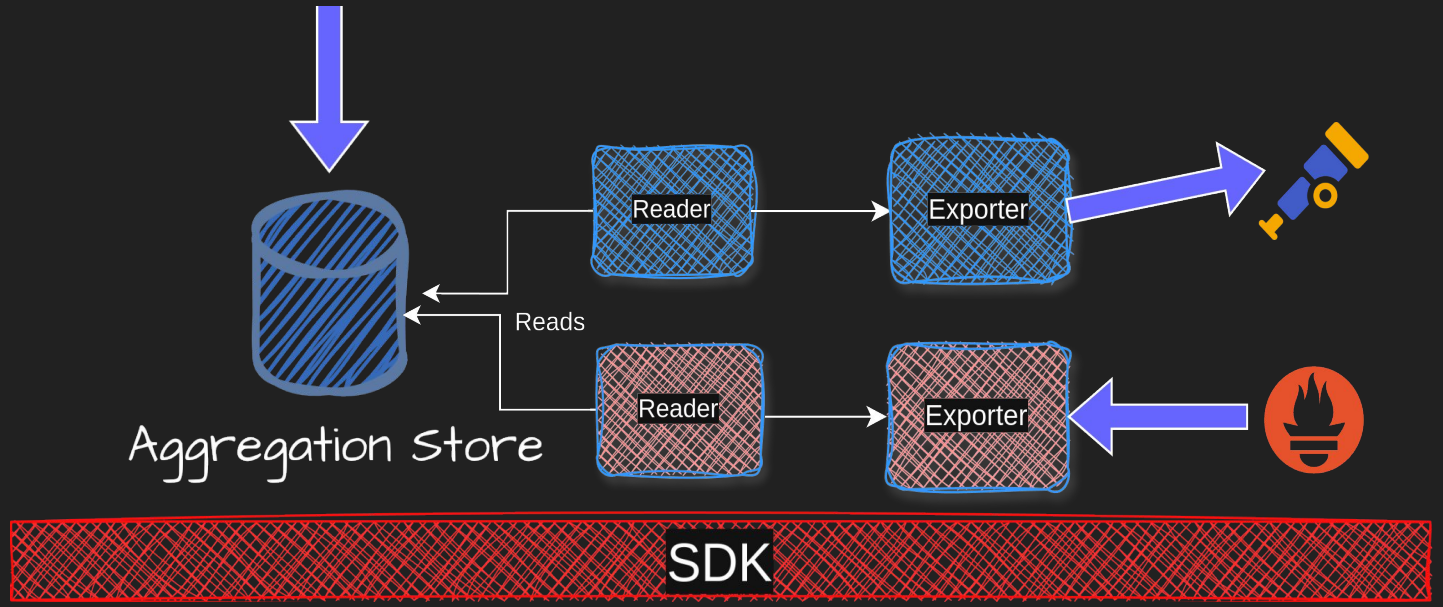
C++ & Singleton stuff



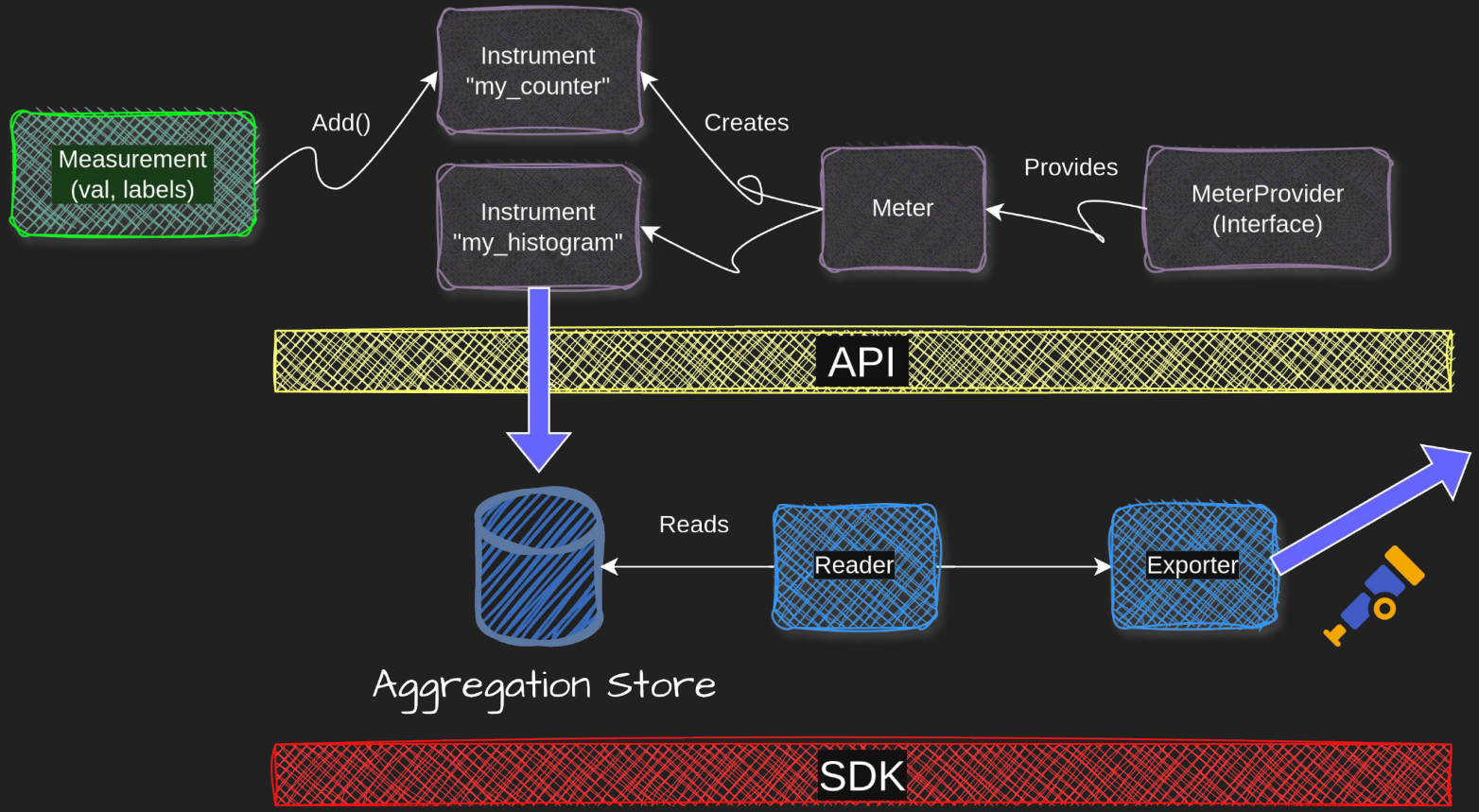

```
void init_metrics_otlp_http(const std::string& url)
{
    sdk_metrics::PeriodicExportingMetricReaderOptions reader_options;
    reader_options.export_interval_millis = std::chrono::milliseconds(1000);
    reader_options.export_timeout_millis = std::chrono::milliseconds(500);

    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(otlpOpti
    auto reader = sdk_metrics::PeriodicExportingMetricReaderFactory::Create(std::move
                                                                            reader_op
    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) { … }
```

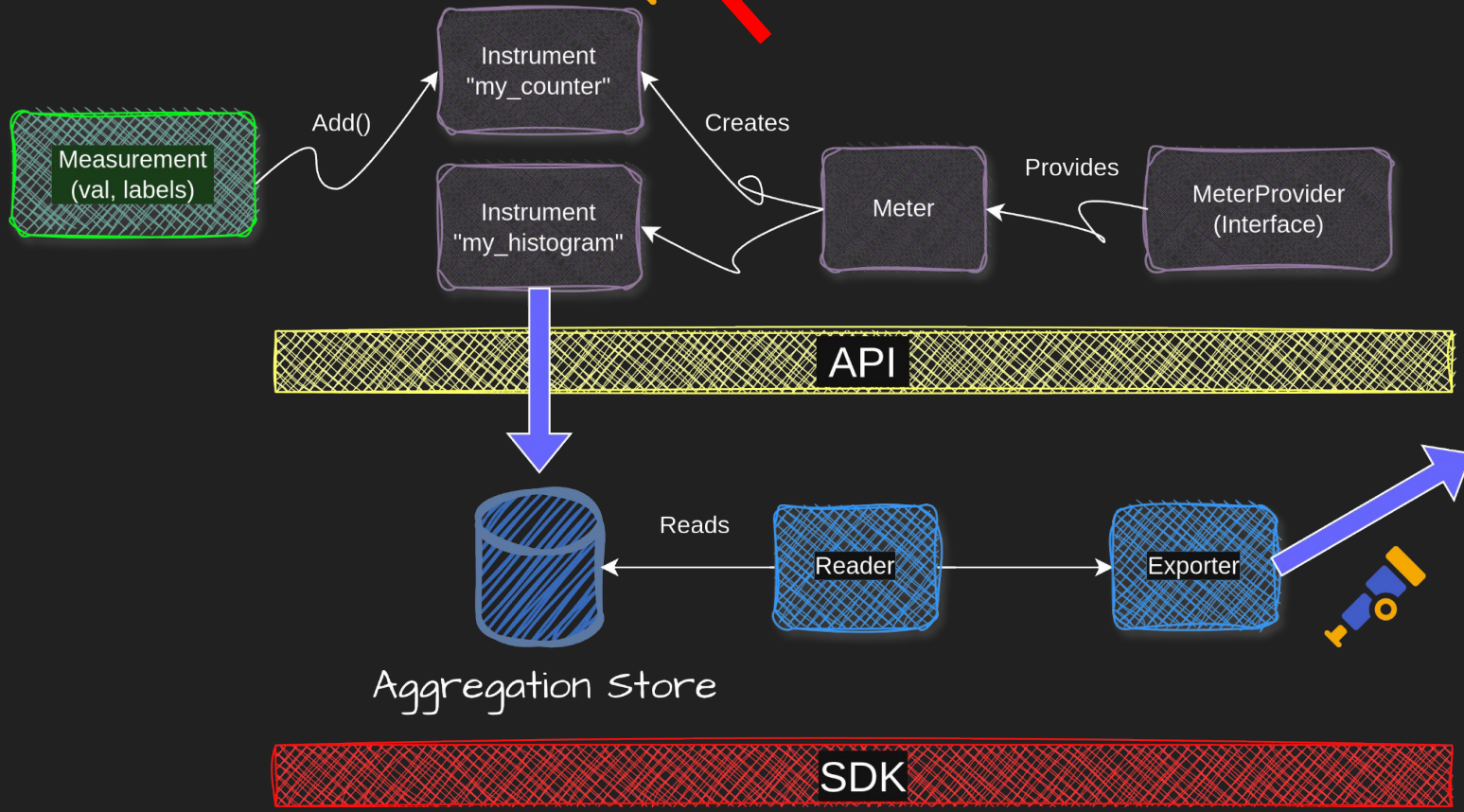


Instrumenting metrics in C++



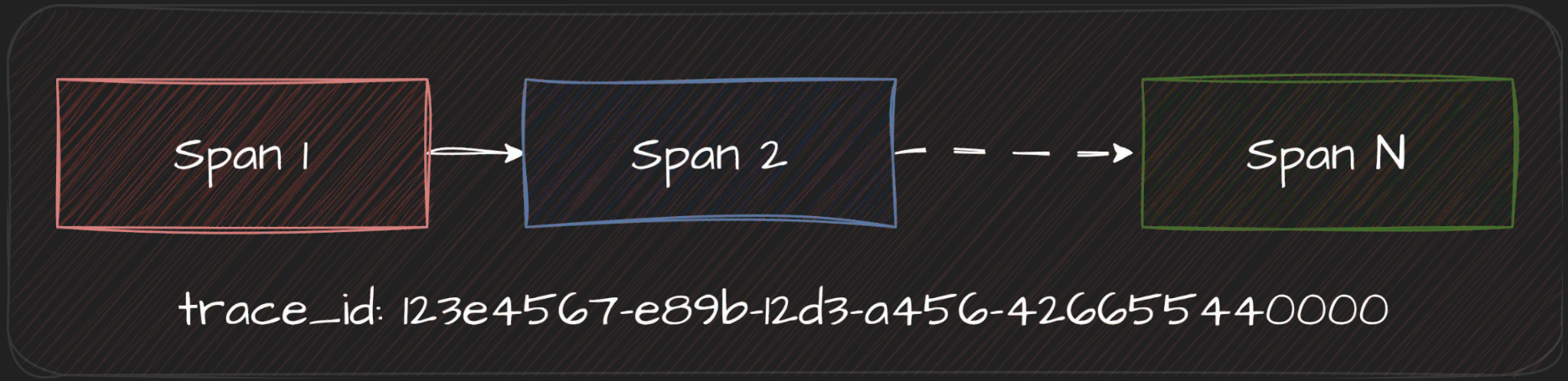
Instrumenting metrics in C++

Language agnostic!

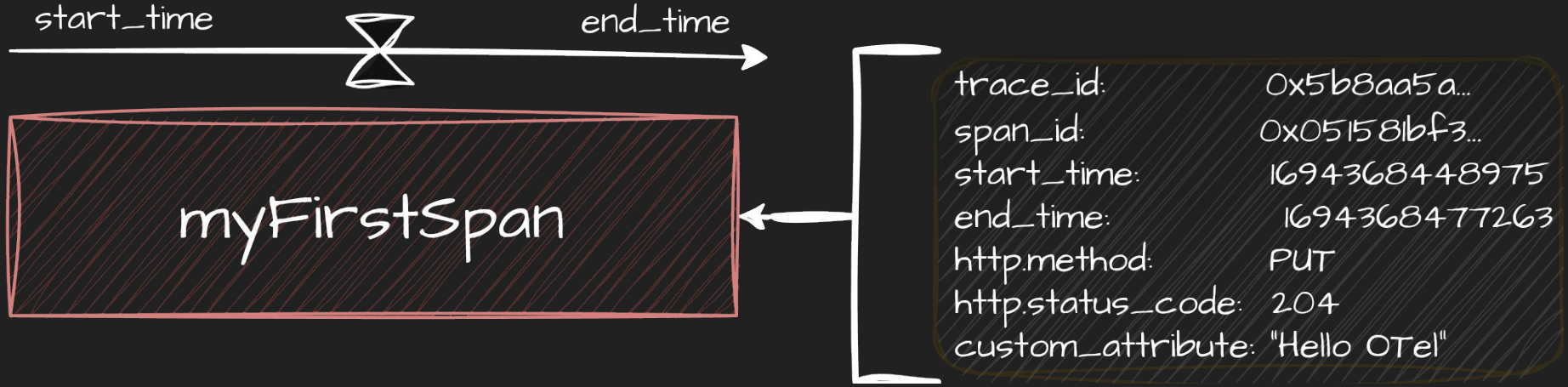


Traces
(quick intro?)

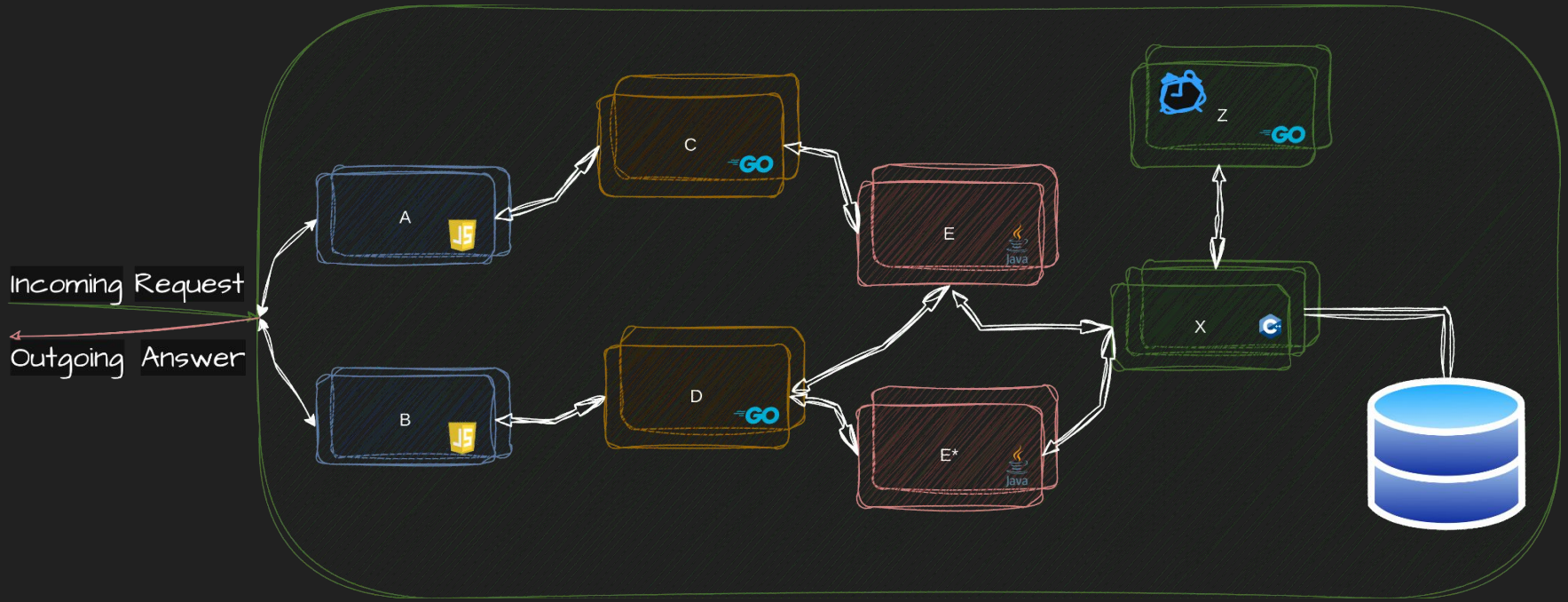
Traces


Span 1
Span 2
Span N
trace_id: 123e4567-e89b-12d3-a456-426655440000

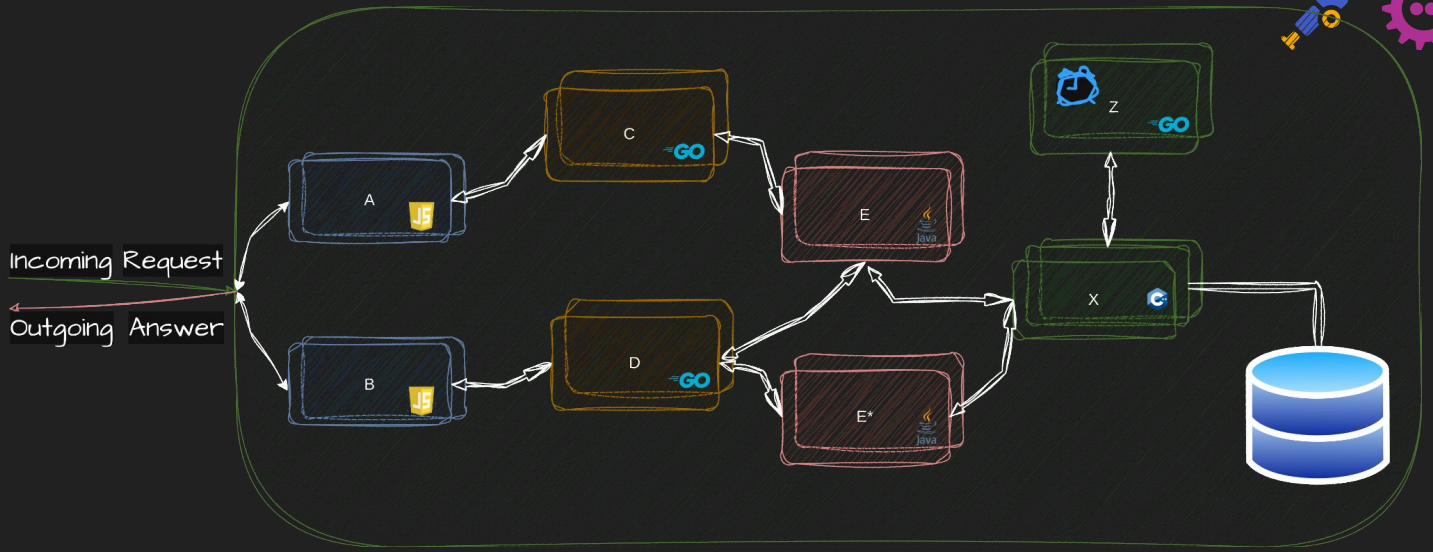
Traces

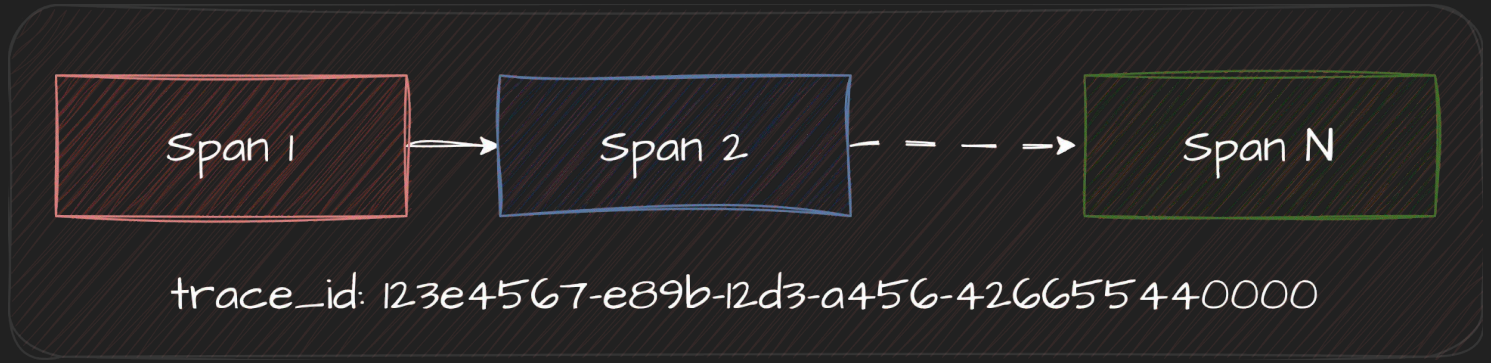

start_time
end_time
myFirstSpan
trace_id: 0x5b8aa5a...
span_id: 0x051581bf3...
start_time: 1694368448975
end_time: 1694368477263
http.method: PUT
http.status_code: 204
custom_attribute: "Hello OTel"

Traces

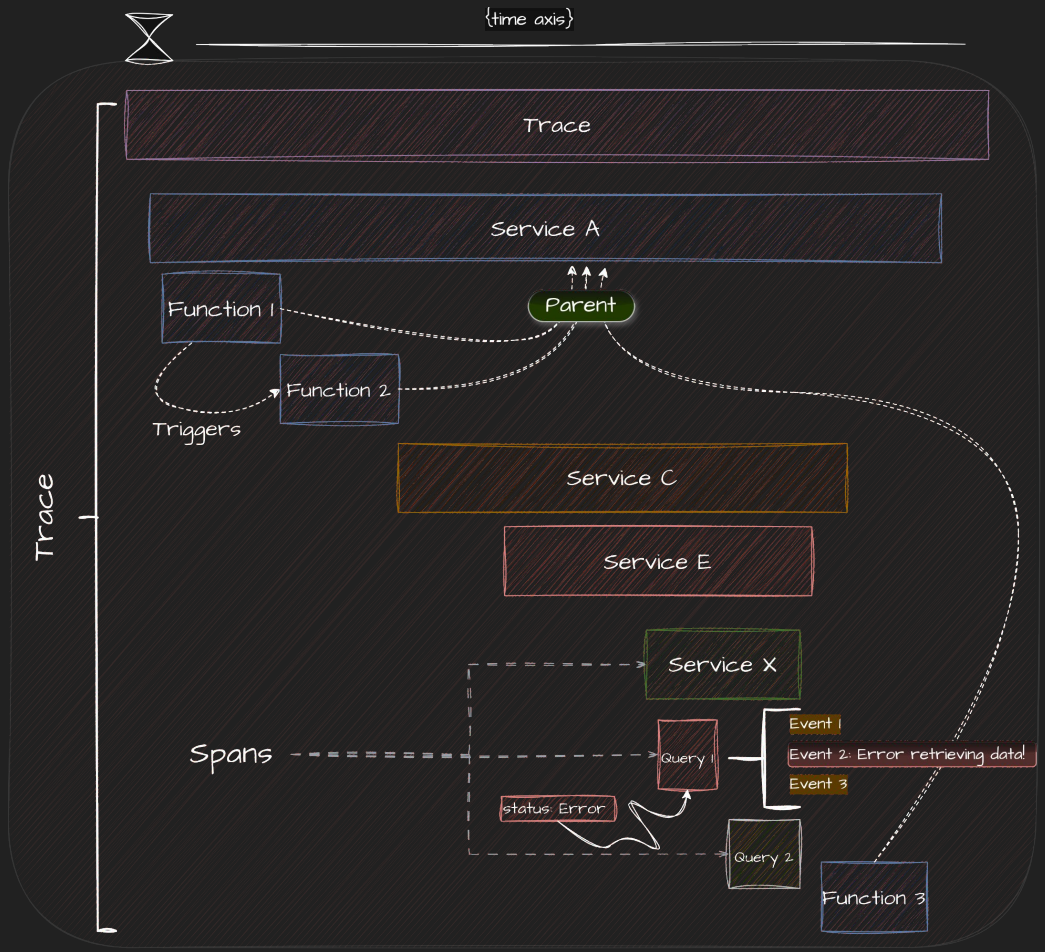


Traces


Incoming Request
Outgoing Answer
A
B
C
D
E
E*
X
Z
GO
Java


Span 1
Span 2
Span N
trace_id: 123e4567-e89b-12d3-a456-426655440000

Traces


{time axis}
Trace
Service A
Function 1
Parent
Function 2
Triggers
Trace
Service C
Service E
Service X
Spans
Query 1
Event 1
Event 2: Error retrieving data!
Event 3
status: Error
Query 2
Function 3

Traces
(instrumenting)

OTel Spec: SDK & API

API

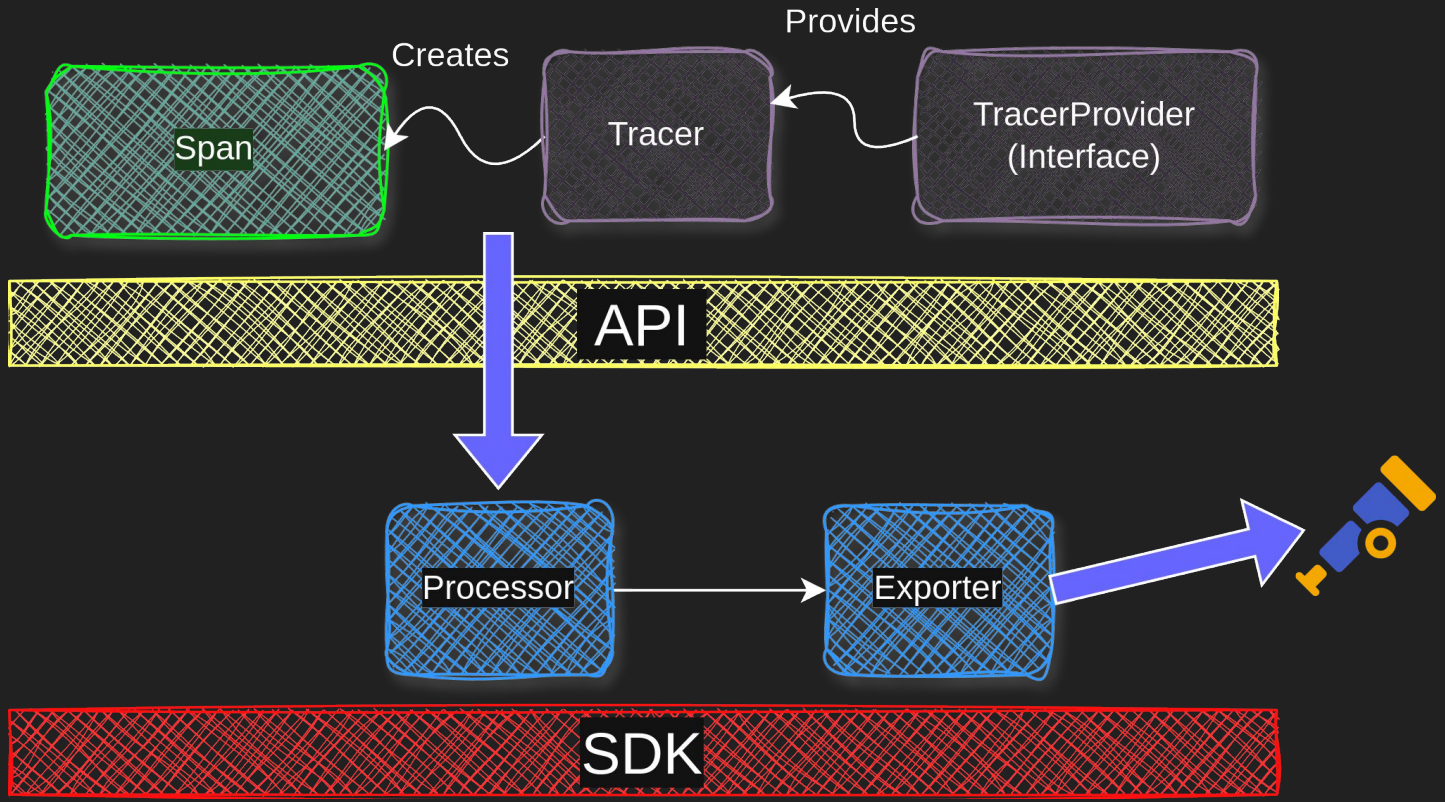
- **Tracer Provider**: the interface
- **Tracer**: The meter is responsible for creating spans
- **Span**: represents a single operation within a trace

SDK

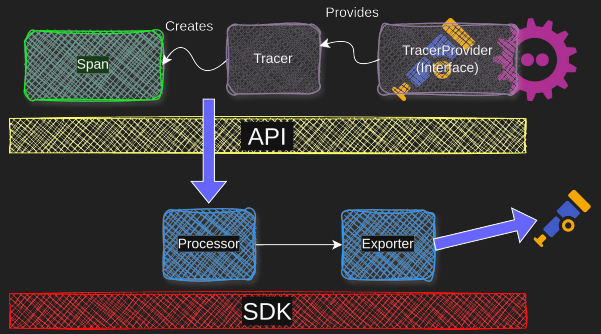
- **Span Processor:** for batching and conversion of spans to exportable representation and passing batches to exporters
- **Span Exporter**: sends data
- (...and more)

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

OTel Spec: SDK & API

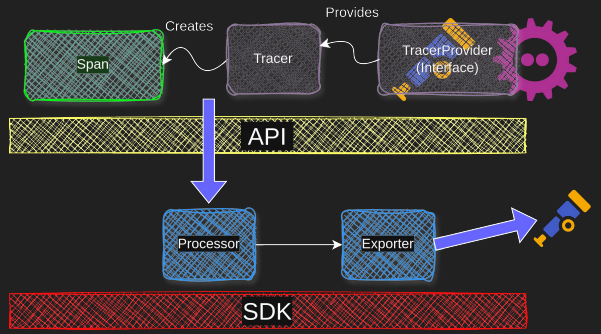


Instrumenting traces in C++: init



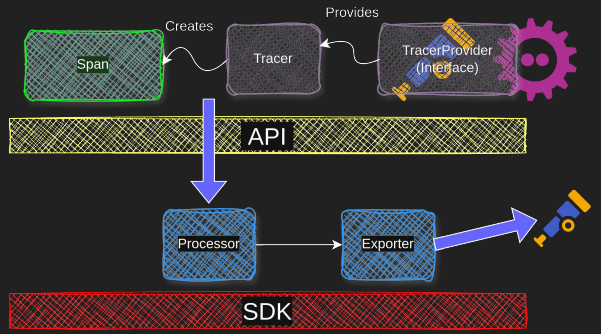

```cpp
void init_tracer(const std::string& url)
{
  auto attrs = opentelemetry::sdk::resource::ResourceAttributes{{"service.name", "my_service"}};
  auto resource = opentelemetry::sdk::resource::Resource::Create(attrs);

  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



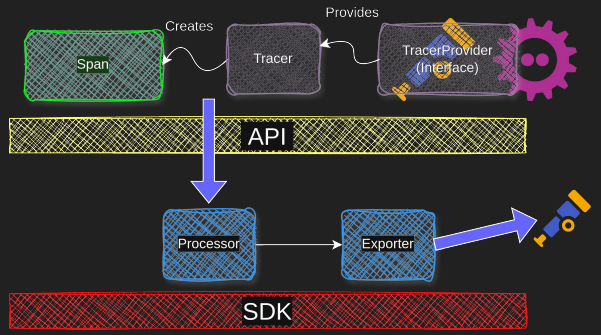

```cpp
void init_tracer(const std::string& url)
{
  auto attrs = opentelemetry::sdk::resource::ResourceAttributes{{"service.name", "my_service"}};
  auto resource = opentelemetry::sdk::resource::Resource::Create(attrs);

  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



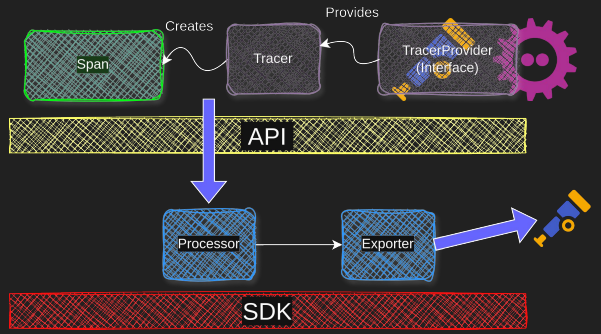

```cpp
void init_tracer(const std::string& url)
{
  auto attrs = opentelemetry::sdk::resource::ResourceAttributes{{"service.name", "my_service"}};
  auto resource = opentelemetry::sdk::resource::Resource::Create(attrs);

  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



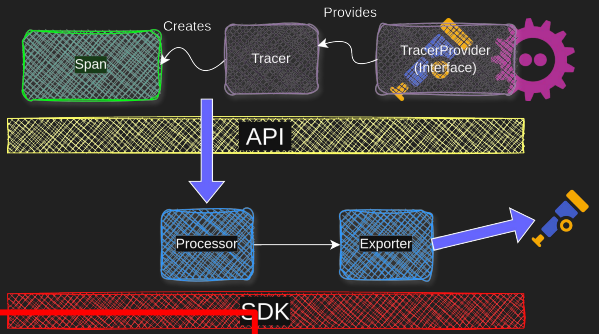

```cpp
ot_std::shared_ptr<ot_trace::Span> create_span(const std::string& name)
{
  ot_trace::StartSpanOptions opts;
  opts.kind = ot_trace::SpanKind::kClient;

  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

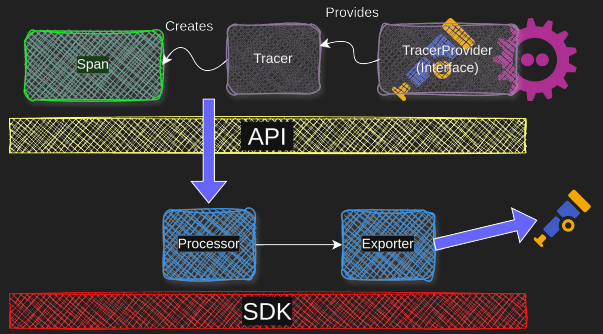
https://opentelemetry.io/docs/specs/semconv/

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.

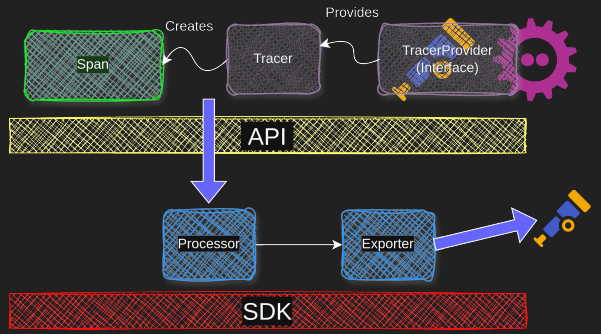
Attribute	Type	Description	Examples	Requirement Level	Stability
`http.request.method`	string	HTTP request method. [1]	`GET` ; `POST` ; `HEAD`	`Required`	stable
`server.address`	string	Host identifier of the "URI origin" HTTP request is sent to. [2]	`example.com` ; `10.1.2.80` ; `/tmp/my.sock`	`Required`	stable
`server.port`	int	Port identifier of the "URI origin" HTTP request is sent to. [3]	`80` ; `8080` ; `443`	`Required`	stable
`url.full`	string	Absolute URL describing a network resource according to RFC3986 [4]	`https://www.foo.bar/search?q=OpenTelemetry#SemConv` ; `//localhost`	`Required`	stable
`error.type`	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++



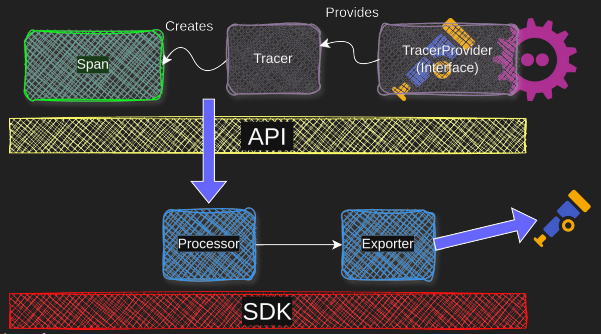

```
void client_impl::send() {

   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++



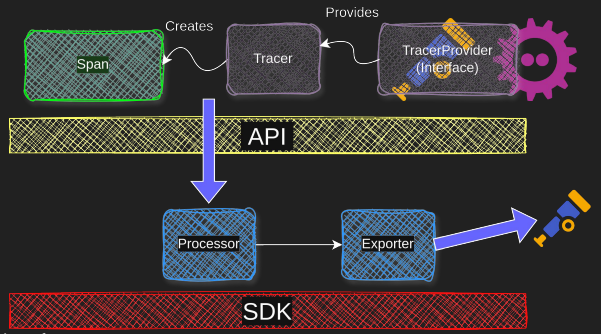

```
void client_impl::send() {

    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++



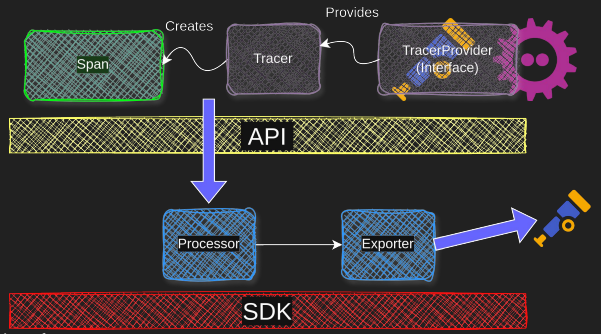

```
void client_impl::send() {
  ...
  nghttp_req->on_response( [this,..., span](const ng::client::response& res) {
    span->AddEvent("Response received");
    span->SetAttribute(ot_trace::SemanticConventions::kHttpResponseStatusCode,res.status_code());

    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++



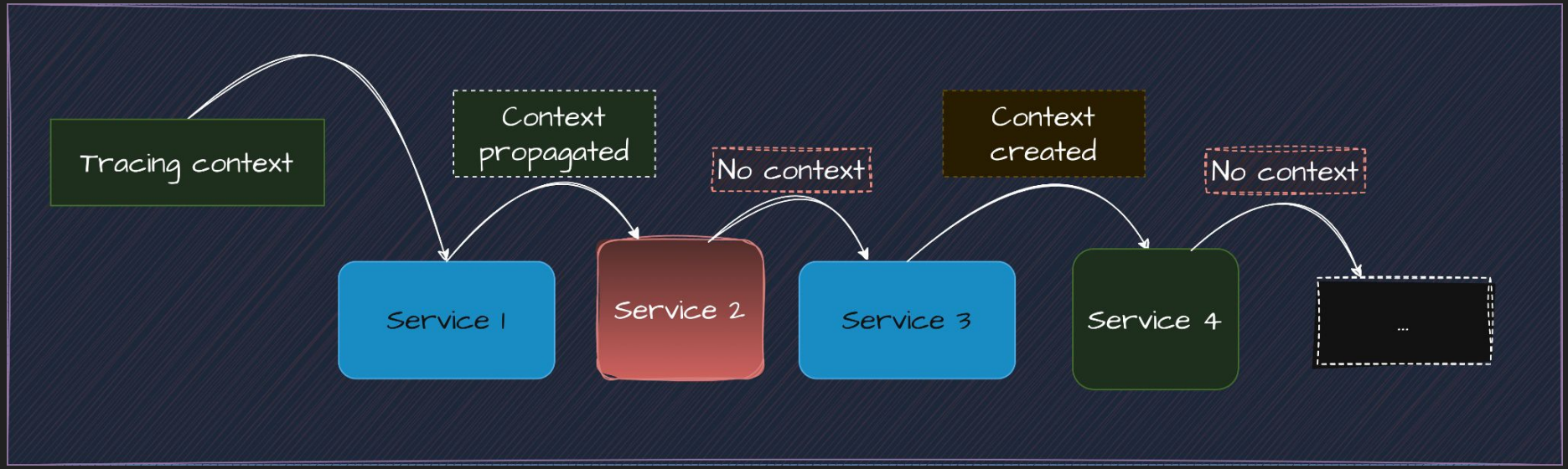

```
void client_impl::send() {
  ...
  nghttp_req->on_response( [this,..., span](const ng::client::response& res) {

    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++



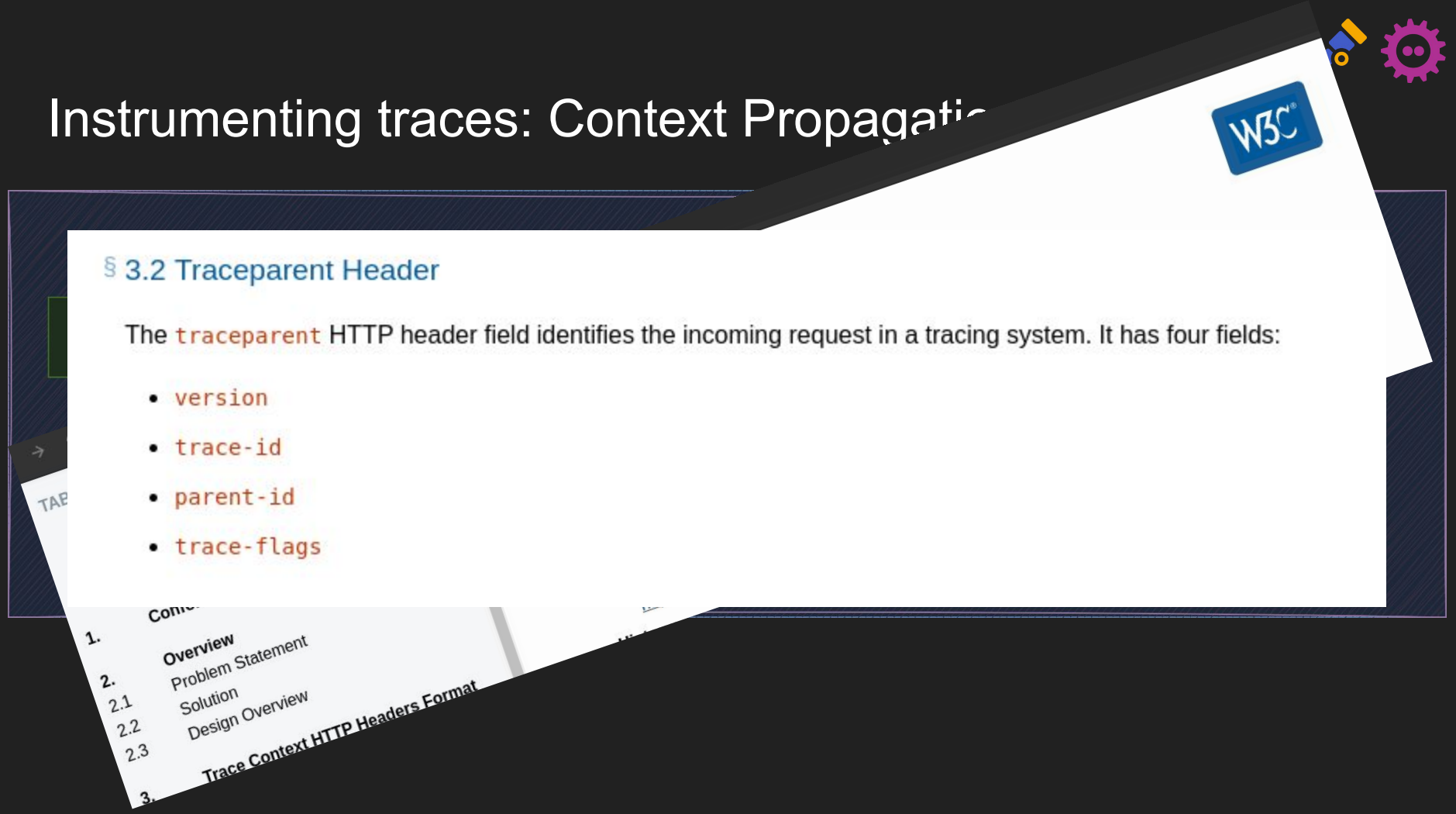

```
void client_impl::send() {
  ...
  nghttp_req->on_response( [this,..., span](const ng::client::response& res) {

    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 Propagati

Instrumenting traces: Context Propagati

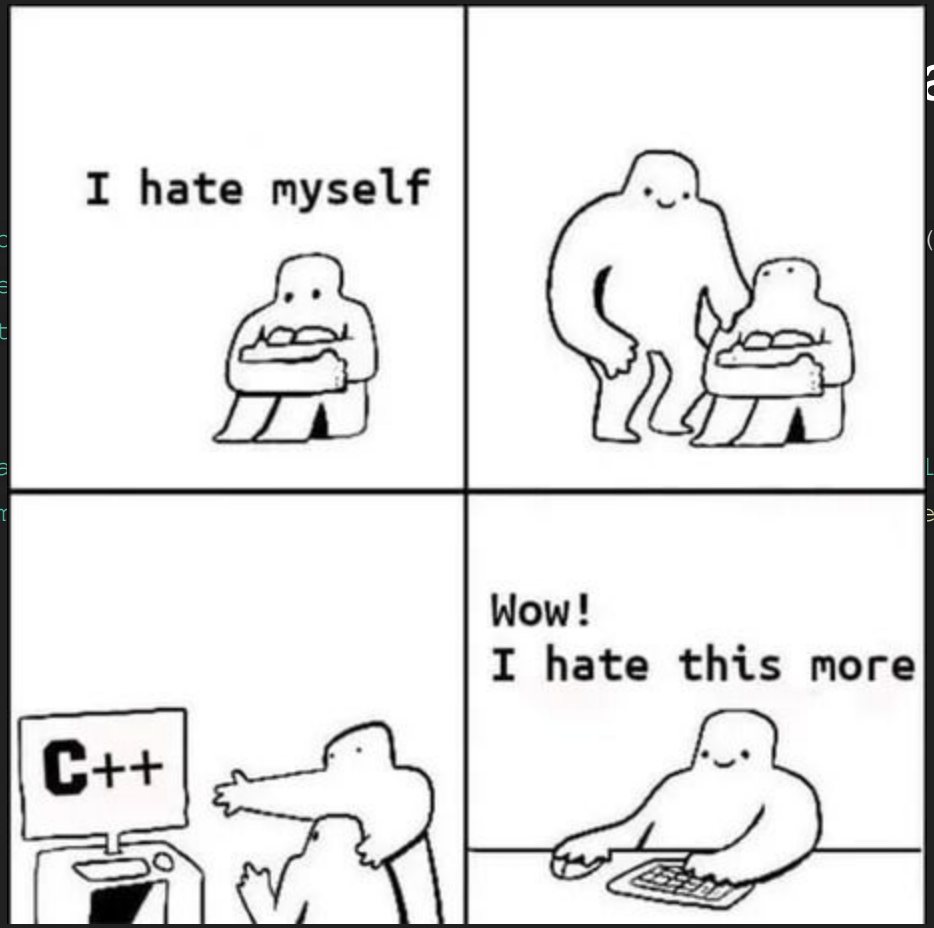


Instrumenting traces in C++: Context Propagation!

```
// 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 ation!

```
// set global propagator
opentelemetry::context::prc                                    (
    nostd::shared_ptr<opente
        new opentelemetry::t

// get global propagator
opentelemetry::trace::propa                                    lient::Headers> carrier;
auto propagator = opentelem                                    etGlobalPropagator();
```



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);
```

DEMO

JAEGER UI | Search | Compare | System Architecture | Monitor | Lookup by Trace ID... | About Jaeger

hermes: script 25c31f4

Find... | Trace Timeline | Archive Trace

Trace Start **February 1 2025, 08:11:03**.228 | Duration **33.62ms** | Services **1** | Depth **2** | Total Spans **5**

0µs | 8.41ms | 16.81ms | 25.22ms | 33.62ms

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

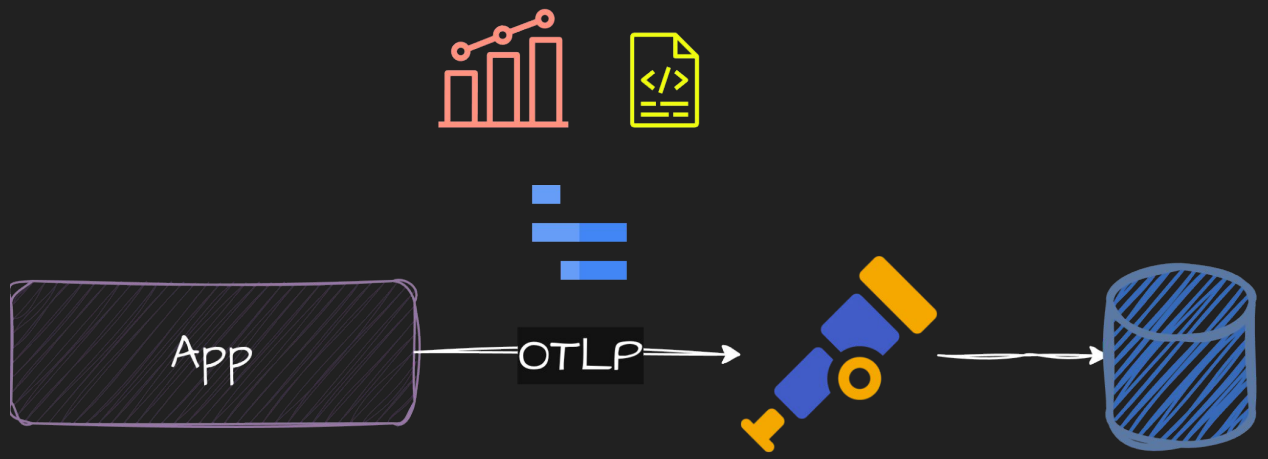
3.54ms: event = Body received

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

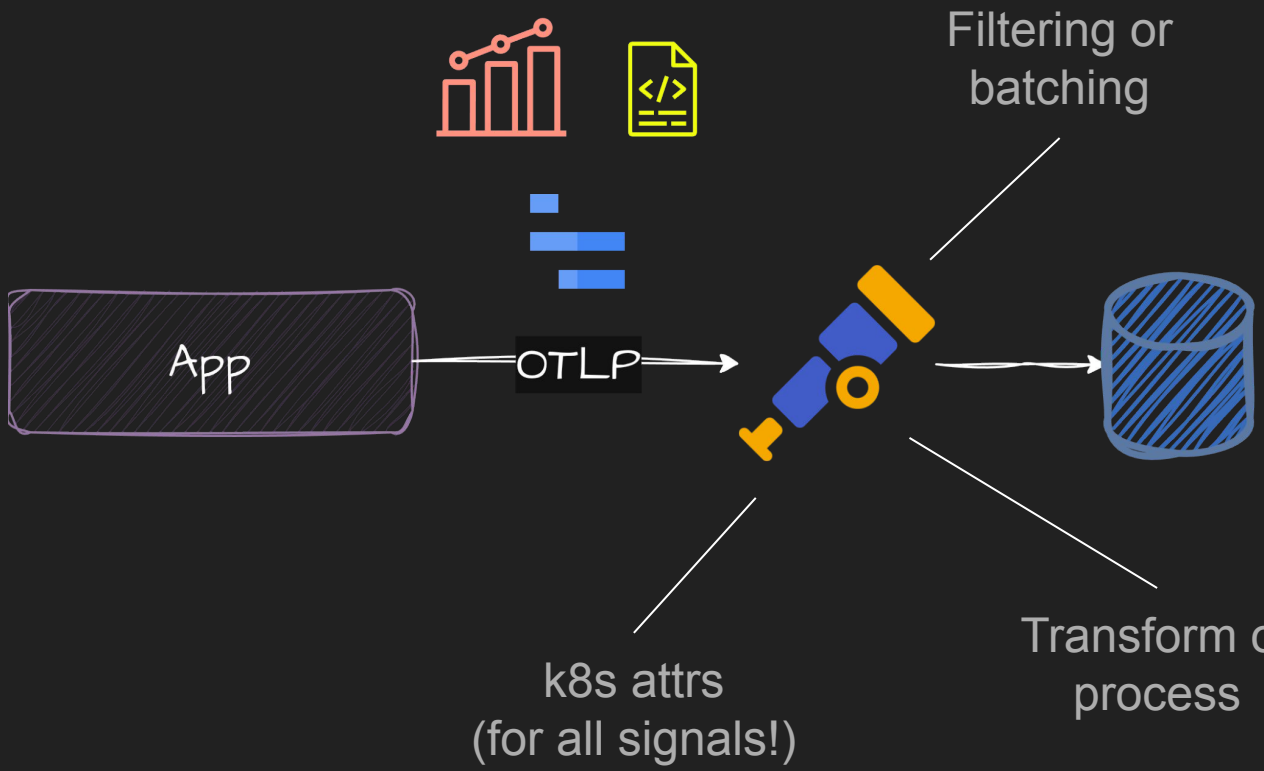
SpanID: 6c0e141853d48eea

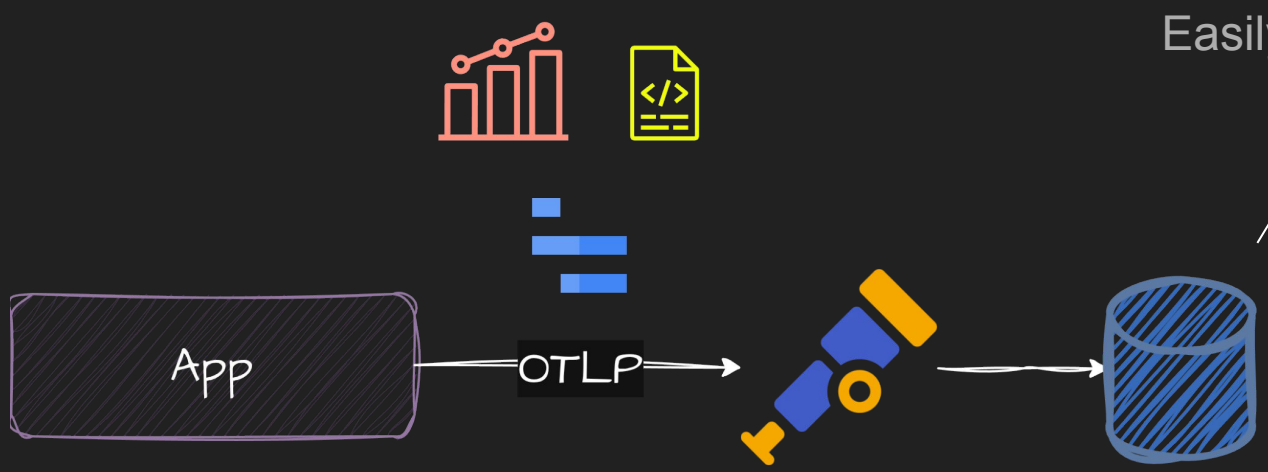

Search or jump to... ctrl+k
Home › Dashboards › Hermes dashboard
Add
Last 5 minutes
5s
Refresh dashboard
request_id
All
Success ratio (last)
100.00%
Requests sent/s (last)
46.7
Responses received/s (last)
46.7
RT (last)
3.06 ms
Response times (ms)
Response time (ms)
3.09 ms
3.08 ms
3.07 ms
3.06 ms
3.05 ms
3.04 ms
08:55:00
08:56:00
08:57:00
08:58:00
08:59:00
delete1
get1
post1
put1

Logs

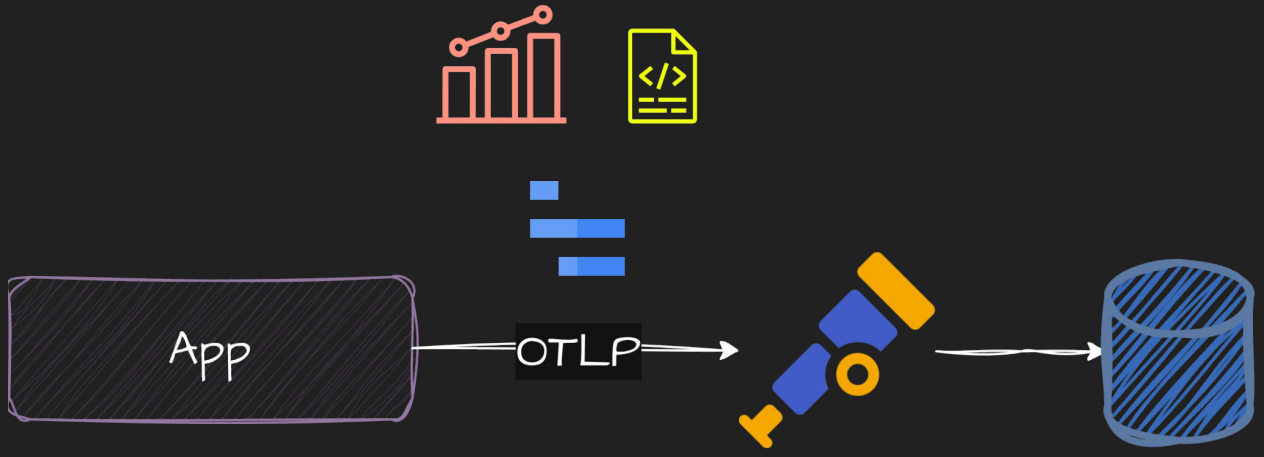

App
OTLP


AFTER ALL...
WHY SHOULDN'T I...
std::cout<<"Error here"<<std::endl;
imgflip.com


Filtering or
batching
App
OTLP
k8s attrs
(for all signals!)
Transform or
process

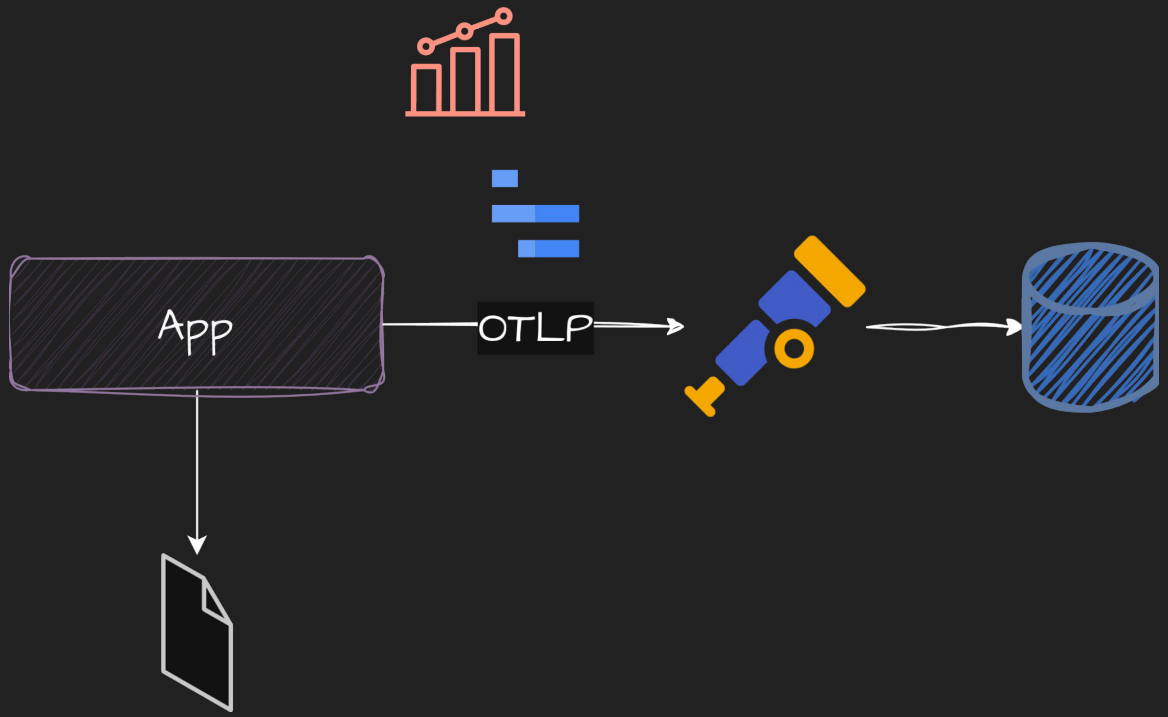


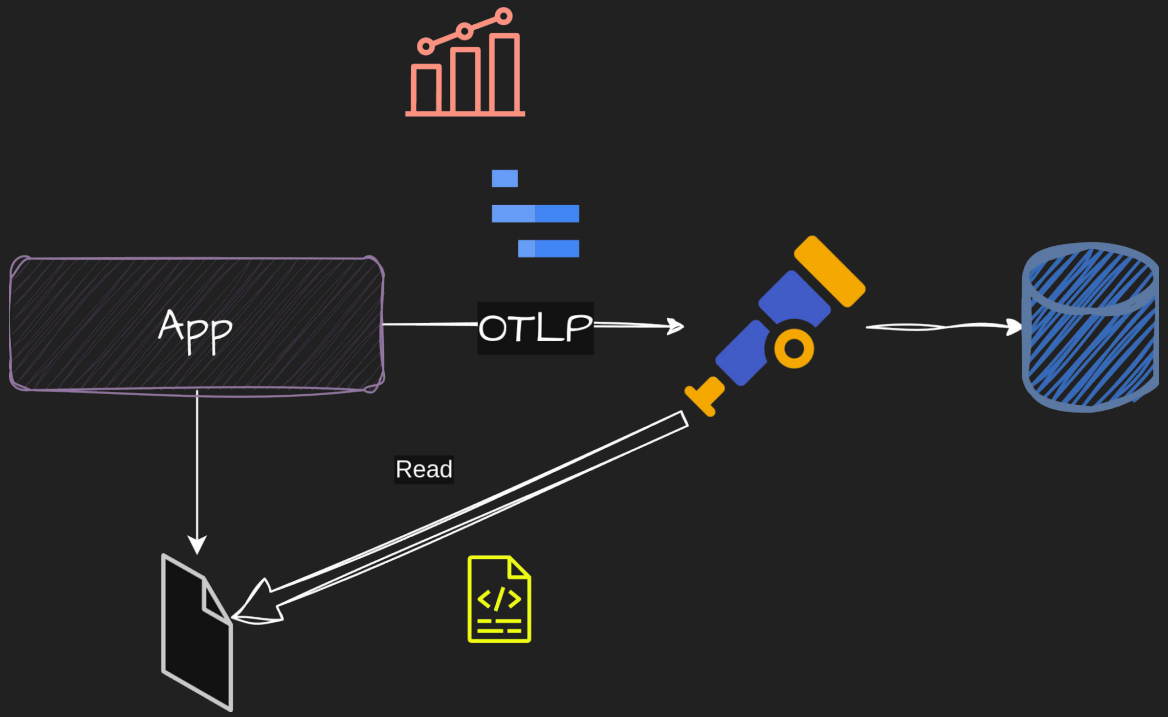
- By the **time** of execution
- By the execution **context** (TraceId or SpanId)
- By the **origin** of the telemetry (resource: who produced it?!)

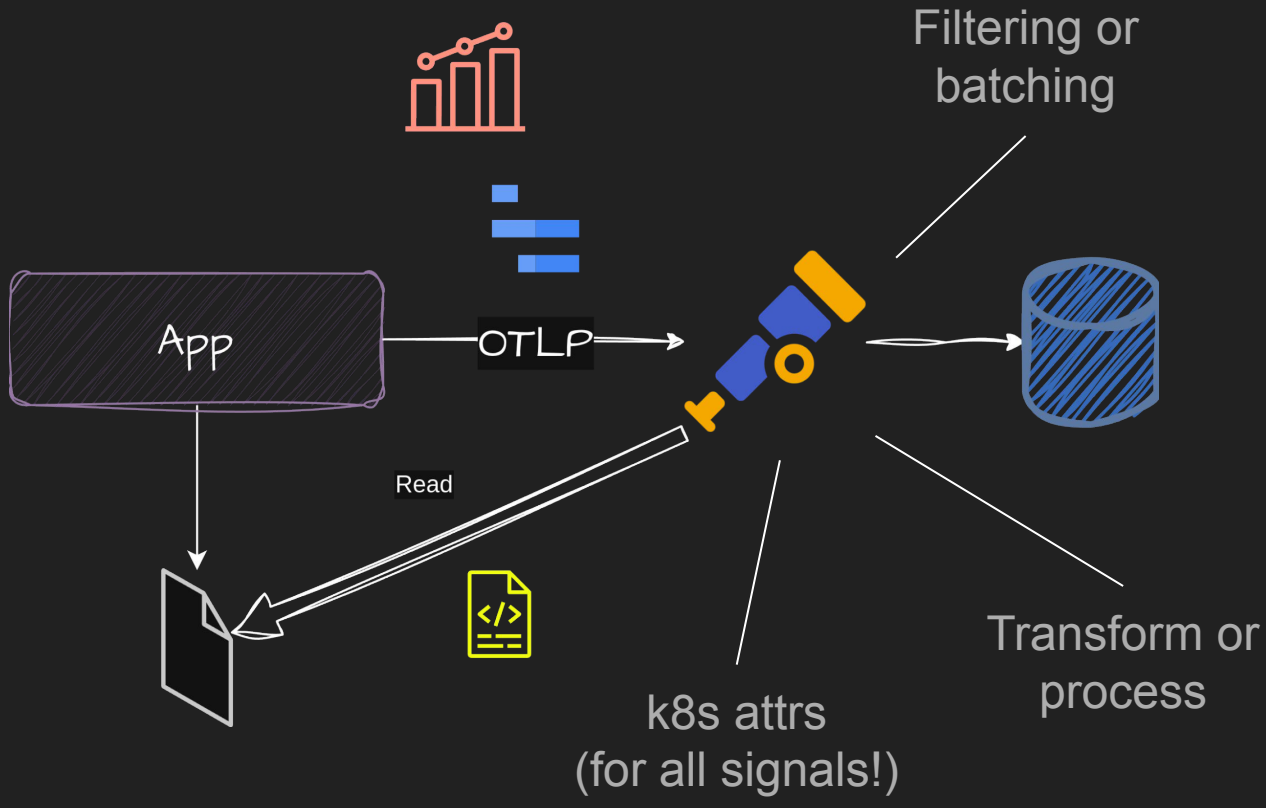


- Logging levels
- Activation and deactivation from a single place
- Correlate with traces
- It’s trendy


STD::COUT<<"ERROR HERE"<<STD::ENDL
38
imgflip.com


App
OTLP


App
OTLP
Read


Filtering or
batching
App
OTLP
Read
k8s attrs
(for all signals!)
Transform or
process

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());
}
```

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)

Summary

- Mixing automatic and manual instrumentation is key
- 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!

Thanks!