A tired old *new* approach to application telemetry metadata

Yuri Shkuro

META
Yuri Shkuro

Software Engineer
Meta

shkuro.com

CNCF Jaeger
Founder & Maintainer
jaegertracing.io

CNCF OpenTelemetry
Co-founder, GC & TC
opentelemetry.io

Mastering Distributed Tracing
Author
Agenda

- Telemetry Metadata
- Schema-First Approach
- Implementation
- Comparison
- Q & A
Observability: a measure of how well internal states of a system can be inferred from knowledge of its external outputs.

Observability begins with Telemetry
TEMPLE - Six Pillars of Telemetry

- T - Traces
- E - Events
- M - Metrics
- P - Profiles
- L - Logs
- E - Exceptions

Telemetry signals describe behaviors of observable entities.

- Host, pod
- Service, endpoint
- Database cluster, ...
- User activity
- Workflow
- Customer account, ...
Dimensions: attributes of telemetry signals that identify observable entities

Statsd: counter.foo.bar.errors

OpenMetrics: errors{service="foo", endpoint="bar"}
Dimensions: necessary, but not sufficient

\[
\text{latency}\{\text{service="team-baz/foo", endpoint="bar"}\} = 0.0152
\]

\[
\text{request\_latency}\{\text{service="foo", endpoint="Foo::bar"}\} = 15.2
\]
**Metadata**: additional info about telemetry that provides semantic meaning and identifies the nature and features of the data.

- Data types
- Units
- Descriptions
- Ownership
- Semantic identifiers
- Purpose policies, …
Metadata unlocks many capabilities

- Discoverability
- Exploration
- Cross-filtering
- Validation & enforcement
- Safe change management
- Privacy controls
Metadata approaches
Industry state of the art

- **Semantic Conventions**
  - OpenTelemetry
  - Elastic Common Schema

- **OpenTelemetry Schemas**
  - versioning of semantic conventions
  - transformations for names and values

- **Externally authored metadata**
  - a.k.a. *a-posteriori* metadata
  - centralized in a metadata store

- **Automatic data enrichment**
  - Agent-based instrumentation
  - limited to infra dimensions
Metadata

↓

Schemas

↓

Schema-first Telemetry

IDL

Compiler

Code
Code-first telemetry
Adding new dimension

```java
counter.Increment(
    service_id = "foo",
    endpoint = "bar",
    status_code = response.code,
    shard_id = "baz",
)
```
Schema-first telemetry
Step 1: Define schema change

typedef string ServiceID
typedef i32 StatusCode
typedef string ShardID

struct RequestCounter {
    1: ServiceID service_id
    2: string endpoint
    3: StatusCode status_code
    4: ShardID shard_id
}
Schema-first telemetry
Step 2: Emit telemetry via struct

counter.Increment(RequestCounter(
    service_id = "foo",
    endpoint = "bar",
    status_code = response.code,
    shard_id = "baz",
))
Schema-first telemetry

Authoring flow

(1) Schema change:
struct RequestCounter {
  ...
  4: ShardID shard_id
}

code gen

(2) Generated code:
struct RequestCounter {
  ...
  shard_id: string
}

(3) Application code:
counter Inc(RequestCounter(
  ...
  shard_id = 'baz',
))

(5) ok to commit

Continuous integration checks

(4) validate

Actualization service

(6) commit

schema

Metadata store
Schema-first telemetry
Production data flow
THRIFT for schema authoring
Why it makes sense for Meta

- De-facto standard at Meta
  - Defines interfaces between services, similar to Protobuf
  - Familiar to most engineers

- Powerful tool chain
  - Build & IDE support, code gen
  - x-language, x-repo syncing

- Language features
  - Type aliases
  - Annotations

- Namespaces & composition
  - Reuse of semantic data types
  - Collaborative authoring
struct HostResource {
  @DisplayName{"Host ID"}
  @Description{"Unique host ID. For Cloud, this must be
    the instance_id assigned by the cloud provider."}
  1: string id

  @DisplayName{"Short Hostname"}
  @Description{"Name of the host as returned
    by the ‘hostname’ command."}
  2: string name

  @DisplayName{"Architecture"}
  @Description{"The CPU architecture of the host system."}
  3: string arch
}
Annotations in the schema
Defining two different representations of the same semantic type

typedef string HostName

// Example: devvm123.zone1.facebook.com
@DisplayName{name="HostName (with FQDN)"}
@SemanticType{InfraEnum.DataCenter_Host}
typedef string HostNameWithFQDN

// Example: devvm123
@DisplayName{"HostName"}
@SemanticType{InfraEnum.DataCenter_Host}
enum OneWayMsgExchangeActorEnum {
    SOURCE = 1, TARGET = 2,
}

@SemanticQualifier
struct OneWayMsgExchangeActor {
    1: OneWayMsgExchangeActorEnum value
}

struct RPC {
    @OneWayMsgExchangeActor{SOURCE}
    @DisplayName{"Source service"}
    1: ServiceID source_service

    @OneWayMsgExchangeActor{TARGET}
    @DisplayName{"Target service"}
    2: ServiceID target_service
}
<table>
<thead>
<tr>
<th>Authoring Experience</th>
<th>Change Management</th>
<th>Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>Schema evolution</td>
<td>Introspection</td>
</tr>
<tr>
<td>Deployment complexity</td>
<td>Change management safety</td>
<td>Semantic x-filtering</td>
</tr>
<tr>
<td>Collaborative authoring</td>
<td>Compile-time safety</td>
<td></td>
</tr>
<tr>
<td>Log site consistency</td>
<td>Automated code changes</td>
<td></td>
</tr>
</tbody>
</table>
## Comparison: approaches to telemetry metadata

<table>
<thead>
<tr>
<th></th>
<th>Lines of code</th>
<th>Deployment</th>
<th>Distributed authoring</th>
<th>Schema consistency at log sites</th>
<th>Schema evolution</th>
<th>Change management safety</th>
<th>Compile time safety</th>
<th>Automated code changes</th>
<th>Introspection</th>
<th>Semantic x-filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain dimensional models</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
</tr>
<tr>
<td>OpenTelemetry Schemas</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
</tr>
</tbody>
</table>

* With automation
* Not applicable
Conclusion

Why schema-first telemetry makes sense for Meta:

- **Schema-first is a paved path**
  - Familiar to most engineers
  - Good tooling support

- **Incremental improvement / migration**
  - Existing a-posteriori metadata solutions
  - Can be applied one dataset at a time
Future work

- Versioning and A/B testing
  - How to “canary” a schema change

- Data governance
  - Defining common semantic types
  - Evolving annotations language
Can it work in OpenTelemetry?

Challenges to overcome

- IDL choice & capabilities
- Developer experience
- End-to-end schema coordination
- Culture change
Thank You
Find me @ https://shkuro.com

Q&A


http://bit.do/schema-first-telemetry

Positional Paper: Schema-First Application Telemetry

Yuri Shkuro, Meta Benjamin Renard, Meta Atul Singh, Meta

ABSTRACT
Application telemetry refers to measurements taken from software systems to assess their performance, availability, correctness, efficiency, and other aspects useful to operators, as well as to troubleshoot them when they behave abnormally. Many modern observability platforms support dimensional models of telemetry signals where the measurements are accompanied by additional dimensions used to identify either the resources described by the telemetry or the business-specific attributes of the activities (e.g., a customer identifier). However, most of these platforms lack any semantic understanding of the data, by not capturing any metadata about telemetry, from simple aspects such as units of measure or data types (treating all dimensions as strings) to more complex concepts such as purpose policies. This limits the ability of the platform to provide a rich user experience, especially when dealing with different telemetry assets, for example, linking an anomaly in a time series with the corresponding subset of logs or traces, which requires semantic understanding of the dimensions in the respective data sets.

In this paper, we describe a schema-first approach to application telemetry that is being implemented at Meta. It allows the observability platforms to capture metadata about telemetry from the start and enables a wide range of functionalities, including complex-time input validation, multi-sign features, and efficient logging and even privacy rules enforcement. We present a collection of design goals and demonstrate how schema-first approach provides better trade-offs than many of the existing solutions in the industry.

1. INTRODUCTION
Observability is a critical capability of today's cloud-native software systems that power products such as Facebook, Gmail, WhatsApp, Twitter, Uber, Rides, etc. Originally defined in control theory, observability provides operators with deeper insight into various aspects of the complex behavior of systems, including their performance, availability, correctness, and efficiency. When the systems behave abnormally, observability is used to troubleshoot the issues and mitigate them in a timely manner. To bring the behavior back to normal, with more time to mitigation being one of the critical success measures.

To provide observability, the systems are instrumented to produce various telemetry signals. The most common types of application telemetry used with today’s cloud-native systems are metrics, logs, events, and traces [12, 21]. A common characteristic of different telemetry types is that they usually combine one or more measurements with a set of identifying dimensions. For example, a metric is a numeric observation typically associated with a name, such as “request_count”, and some dimensions, such as “host” or “endpoint”. Similarly, in a semi-structured log message, the measurement part is played by the message text, accompanied by searchable dimensions such as log level, thread name, etc.

Modern telemetry platforms, in addition to ingesting vast amounts of telemetry data, usually perform extensive indexing of the dimensions to allow rich querying and aggregation over the raw measurements [17, 10, 12]. Most of them treat dimensions as fine-grained collections of key-value pairs, like platforms like OpenTelemetry [15] or Jaeger [20] allow associating basic types with dimension values, while systems like Prometheus [6] allow associating descriptions with the metrics’ names treating all dimensions as strings.

Lack of any additional metadata or captured or understood by these systems. This puts a burden on the user to understand how to interpret the dimensions and how to leverage them when querying data.

We propose a solution that takes advantage of schema-first modeling to provide a richer representation of the telemetry data. This includes enriching the data with semantic metadata, such as dimension descriptions, target datasets, labels, time stamps, and other aspects useful to operators, as well as to troubleshoot the issues when they behave abnormally.

The complex nature of cloud-native systems often requires investigations that involve more than one single source of telemetry. A spike in error rate in a single zone might warrant a look at the logs or traces from the same zone for better diagnosis of the issue. This is where many modern telemetry platforms fall short, as they lack semantic understanding of the data. Two telemetry signals might share a dimension “region”, but in one case referring to the region where the software runs and in the other case to the region where the user is hosted. Joining telemetry by this dimension as it is the same thing is probably meaningless. Metadata can be the missing link in solving these problems.

In this paper we define metadata as additional information that provides semantic meaning to telemetry data and helps in identifying the names and features of the data. Examples of observability metadata include data types, units, descriptions, ownership, purpose policies, semantic identifiers, etc.

There are different ways to associate metadata with telemetry, such as using naming conventions to imply semantic meaning or defining metadata as a postet, after the telemetry data has been produced and stored. In this paper we...