Schema-First Telemetry

A ~~tired old~~ *new* approach to application telemetry metadata

Yuri Shkuro
META

Yuri Shkuro

Software Engineer
Meta

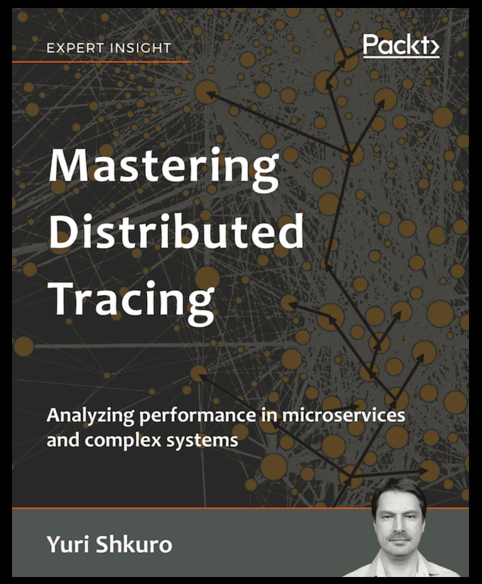
shkuro.com

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**.

TEMPLE - Six Pillars of Telemetry

Photo by Dario Crisafulli on Unsplash

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

Blog post: https://bit.do/telemetry-temple

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

request_latency{service="foo", endpoint="bar"}=0.0152

Dimensions: necessary, but not sufficient

latency{service=“team-baz/foo”, endpoint=“bar”} = 0.0152

request_latency{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 & correlation
- 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

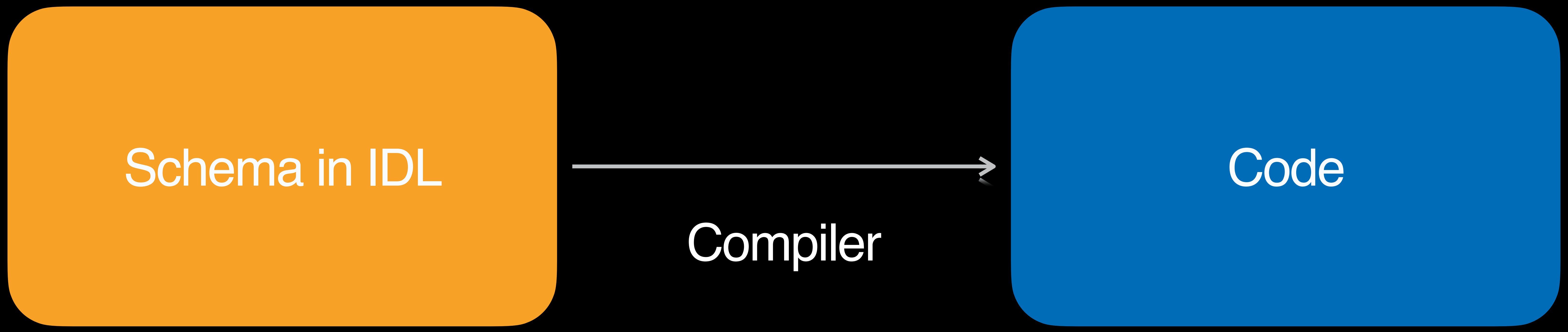
Metadata

⇓

Schemas

⇓

Schema-first Telemetry



Code-first telemetry

Producing a time series

Value (+1)

```
counter.Increment(
  service_id  = "foo",
  endpoint    = "bar",
  status_code = response.code,
)
```

Dimensions

Code-first telemetry

Adding new dimension

New dimension →

```
counter.Increment(
  service_id  = "foo",
  endpoint    = "bar",
  status_code = response.code,
  shard_id    = “baz",
)
```

Schema-first telemetry

Define schema

Schema in IDL

```
struct RequestCounter {
  1: string  service_id
  2: string  endpoint
  3: int     status_code
}
```

Schema-first telemetry

Emit telemetry

Schema in IDL

```
struct RequestCounter {
  1: string  service_id
  2: string  endpoint
  3: int     status_code
}
```

Code

```
counter.Increment(
  RequestCounter(
    service_id  = "foo",
    endpoint    = "bar",
    status_code = resp.code,
  )
)
```

Schema-first telemetry

Adding new dimension to schema

Schema in IDL

```
struct RequestCounter {
  1: string  service_id
  2: string  endpoint
  3: int     status_code
  4: string  shard_id
}
```

Code

```
counter.Increment(
  RequestCounter(
    service_id  = "foo",
    endpoint    = "bar",
    status_code = resp.code,
  )
)
```

Schema-first telemetry

Emitting new dimension

Schema in IDL

```
struct RequestCounter {
  1: string  service_id
  2: string  endpoint
  3: int     status_code
  4: string  shard_id
}
```

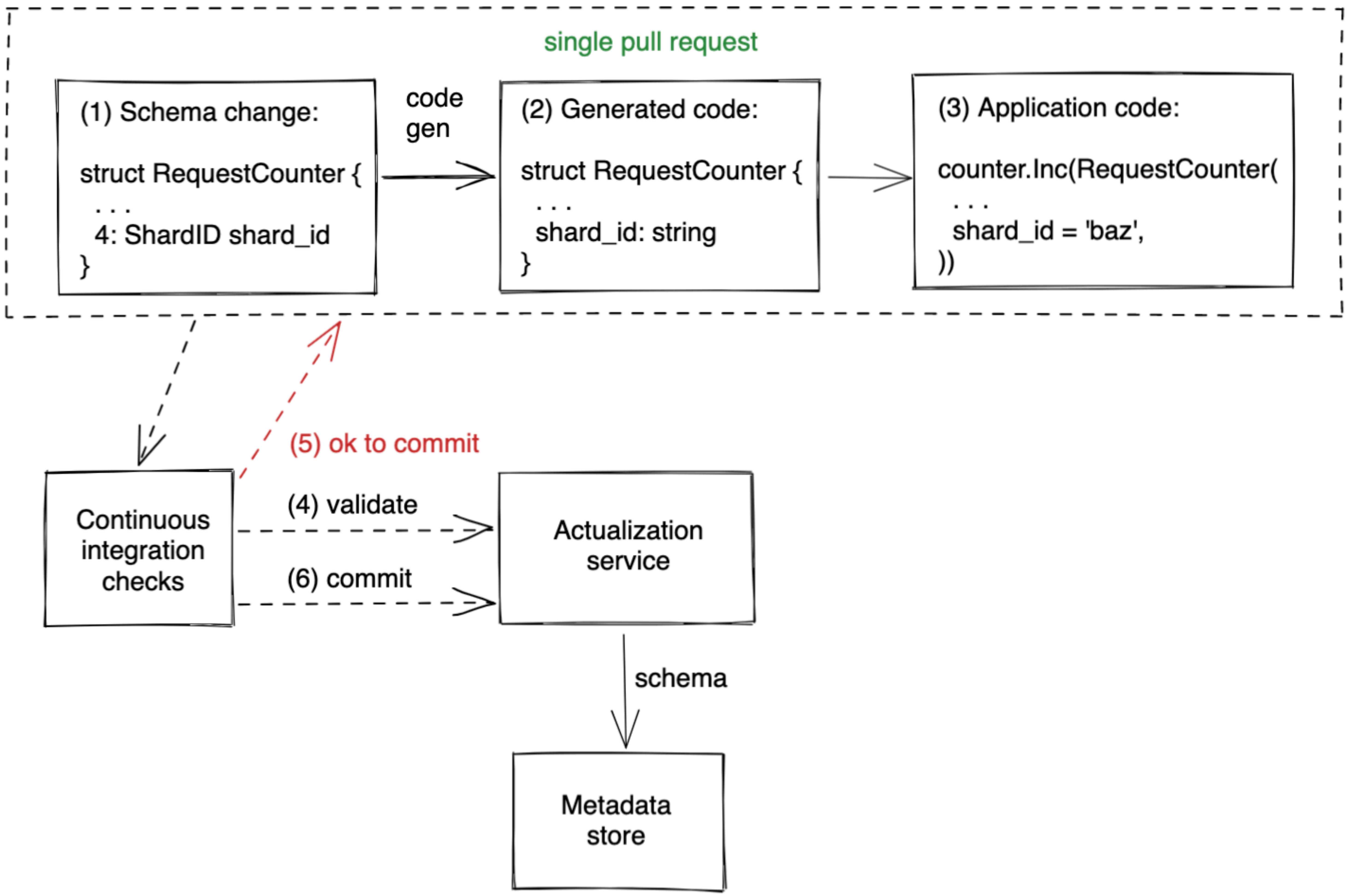
Code

```
counter.Increment(
  RequestCounter(
    service_id  = "foo",
    endpoint    = "bar",
    status_code = resp.code,
    shard_id    = “baz",
  )
)
```

Implementation

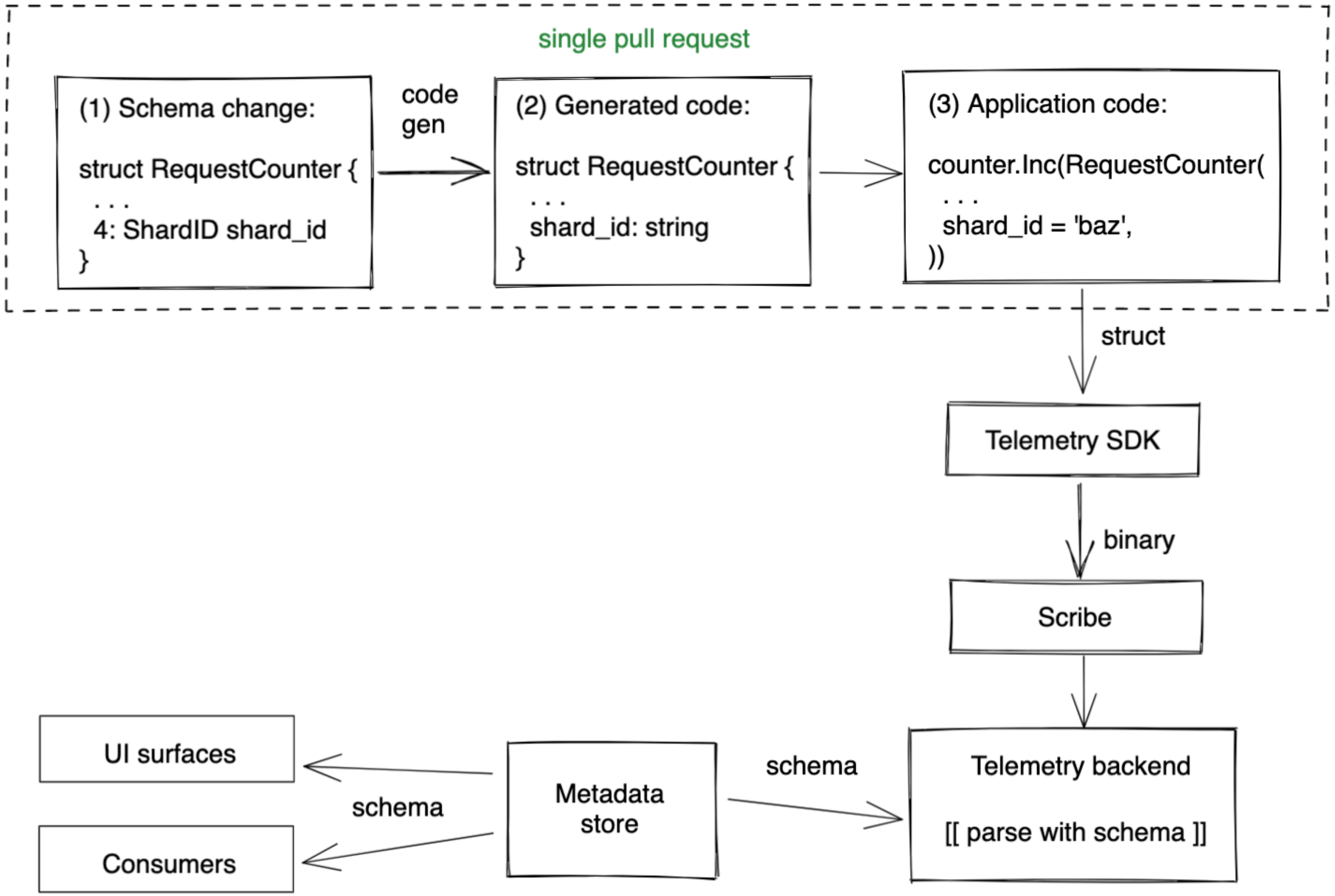
Schema-first telemetry

Authoring flow



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

Metadata in the schema

Redefining OpenTelemetry semantic convention for host resources

```
struct HostResource {

  1: string id


  2: string name


  3: string arch
}
```

Metadata in the schema

Redefining OpenTelemetry semantic convention for host resources

```
struct HostResource {
  @DisplayName{"Host ID"}
  @Description{"Unique host ID. For Cloud, this must be ..."}
  1: string id

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

  @DisplayName{"Architecture"}
  @Description{"The CPU architecture of the host system."}
  3: string arch
}
```

Metadata in the schema

Using rich types

Primitive types

```
struct RequestCounter {
  1: string  service_id
  2: string  endpoint
  3: int     status_code
  4: string  shard_id
}
```

Metadata in the schema

Using rich types

Primitive types → Type aliases

```
struct RequestCounter {
  1: string   service_id
  2: string   endpoint
  3: int      status_code
  4: string   shard_id
}
```

```
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
}
```

Metadata in the schema

Annotations on shared rich types

```
// Example: devvm123
@DisplayName{"HostName"}
typedef string HostName

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

Annotations in the schema

Defining two different representations of the same semantic type

```
// Example: devvm123
@DisplayName{"HostName"}
@SemanticType{InfraEnum.DataCenter_Host}
typedef string HostName

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

Annotations in the schema

Qualifying rich type fields with additional semantic meaning

```
struct RPC {

  @DisplayName{"Source service"}
  1: ServiceID source_service

  @DisplayName{"Target service"}
  2: ServiceID target_service
}
```

Annotations in the schema

Qualifying rich type fields with additional semantic meaning

```
enum OneWayMsgExchangeActorEnum {
  SOURCE = 1, TARGET = 2,
}

struct OneWayMsgExchangeActor {
  1: OneWayMsgExchangeActorEnum value
}
```

Annotations in the schema

Qualifying rich type fields with additional semantic meaning

```
enum OneWayMsgExchangeActorEnum {
  SOURCE = 1, TARGET = 2,
}

@SemanticQualifier
struct OneWayMsgExchangeActor {
  1: OneWayMsgExchangeActorEnum value
}
```

Annotations in the schema

Qualifying rich type fields with additional semantic meaning

```
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
}
```

Comparison

Authoring Experience

- Lines of code
- Deployment complexity
- Collaborative authoring
- Log site consistency

Change Management

- Schema evolution
- Change management safety
- Compile-time safety
- Automated code changes

Consumption

- Introspection
- Semantic x-filtering

Comparison: approaches to telemetry metadata

	Authoring experience				Change management				Consumption	
	Lines of code	Deployment	Distributed authoring	Schema consistency at log sites	Schema evolution	Change management safety	Compile time safety	Automated code changes	Introspection	Semantic x-filtering
Plain dimensional models	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
Semantic Conventions	✓	✓	✓	–	–	–	–	✗	–	–
OpenTelemetry Schemas	✓	✓	✓	–	✓	–	–	✗	–	–
Externally authored metadata	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓
Automatic data enrichment	✤	✤	✗	✓	✓	✓	✤	✤	✓	–
Schema-first approach	–	✓ *	✓	✓	✓	✓	✓	✓	✓	✓

* 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

Yuri Shkuro, Benjamin Renard, and Atul Singh. 2022. *Positional Paper: Schema-First Application Telemetry.* SIGOPS Oper. Syst. Rev. 56, 1 (June 2022), 8–17.

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 platforms 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 compile-time input validation, multi-signal correlations and cross-filtering, 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 incidents and mitigate them to bring the behavior back to normal, with mean 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 aggregations over the raw measurements [17], [10], [2]. Most of them treat dimensions as free-form collections of key-value pairs. 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 while treating all dimensions as strings. Little, if any, additional metadata is 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.

The complex nature of cloud native systems often requires investigations that involve more than a 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 located. Joining telemetry by this dimension as if 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 nature 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 *a-posteriori*, after the telemetry data has been produced and stored. In this paper we