MetaStore

gRPC and Protobuf meta registry

Speaker - Alex Van Boxel

twitter: @alexvb

- Lead Data Architect - Veepee.com

  (what's a big data guy doing at a gRPC conference?!)
- Tech Geek
- Google Developer Expert

Key takeaways... (what already?!)

- We need a schema registry for Protobuf/gRPC
- **We need a standard registry API more!**
  - Look for every gRPC logo in the diagrams
- MetaStore wants to be a reference implementation, but it doesn't want to be the only one

History

Before MetaStore

Take 1 - Big Data, the old way

ETL - Extract > Transform > Load

Source schema changes resulted in

- Extraction failure (SQL queries)
  - Column changes or removal resulted in SQL failures
  - Addition were not picked up
- Transform
  - Deprecated column still used in transforms

Take 2 - Streaming Data

Opportunity for designing a new system.

Options on the table

- gRPC for microservice API
- Protobuf over PubSub
    - For async communication
    - For entry transfer to data warehouse

Take 2 - Streaming Data

Lost the fight on

- No gRPC, .NET usage not idiomatic
- No contract first on Protobuf (oh dear...)

But we **paid a price**

```
...  ...  @@ -11,5 +11,5 @@ message Reccon {
11   11       string id = 1;
12   12       .google.protobuf.Timestamp date = 2;
13   13       int32 unknowns = 3;
14        -    string inbound_transport_id = 4;
     14   +    int64 inbound_transport_id = 4;
15   15   }
```

Problem Domain

You learn for the past

Not going contract first

Developers are very resistant for contract first, mostly it doesn't fit in the agile workflow mindset, but not doing so...

- Flaky interoperability
- Bad design
- No client libraries

Big Data - Configuration

Configuration of the data pipelines where to far away from the contracts

- Leading to very complex configuration files
- Referencing fields in the contracts

Solution

Take 3

[1] Contract First

Learn for past mistakes

Contract first - API - gRPC

Ever tried writing swagger, by hand. I did... it's not fun...

Very well supported on major languages, even Microsoft is onboard in .NET Core 3.0

[1] Contract first - Bus - Protobuf

It’s data, why use another format over your async bus (be it Pub/Sub, Kafka, RabbitMQ, ...)

- Consistent way of working
- Single tool chain
- Same message can go over the API

  (we saw it happen in the past anyway)

[2] Safety first

Contract first is not enough

[2] Contract first - Safeguards

Tool that police schema correctness and evolution - MetaStore

- Linting
- Diffing

[2] Safety - Schema best practices (lint)

Examples of linting

- Contract leakage - It MUST always be an error when an other version of the same package is referenced.

[2] Safety - Schema best practices (lint)

Examples of linting

- Contract leakage - It MUST always be an error when an other version of the same package is referenced.

```
syntax = "proto3";

import "example/v1/event.proto";

package example.v2;

message UseOfWrongEvent {
    example.v1.Event event = 1;
}
```

[2] Safety - Schema best practices (lint)

Examples of linting

- Contract leakage - It MUST always be an error when an other version of the same package is referenced.
- gRPC request/response message - A Service method should have a Request / Response message.

[2] Safety - Schema best practices (lint)

Examples of linting

- Contract leakage - It MUST always be an error when an other version of the same package is referenced.
- gRPC request/response message - A Service method should have a Request / Response

```
service MethodService {
    rpc MethodOk (MethodRequest) returns (MethodResponse);
    rpc MethodEmpty (google.protobuf.Empty) returns (google.protobuf.Empty);
    rpc MethodEmptyI (google.protobuf.Empty) returns (MethodResponse);
    rpc MethodEmptyR (MethodRequest) returns (google.protobuf.Empty);
}

message MethodRequest {}

message MethodResponse {}
```

[2] Safety - Schema evolution (diff)

Protobuf is very resilient to schema change: Compatible, but data loss can occur when consumer has lower precision. (but should we allow this, maybe we need profiles..)

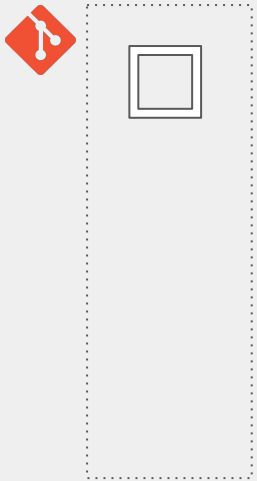
- Field type change int32, uint32, int64, uint64, and bool
- Field type change sint32 and sint64
- Field type change string and bytes - Compatible as long as bytes are UTF-8.
- Field type change bytes and message - Compatible with bytes if the bytes contain an encoded version of the message.

[3] Shadow Contracts

Enrich the contracts, without touching the originals

[3] Shadow Contracts

Contracts are owned by a certain team



gRPC and consumer / producer workload

master repo

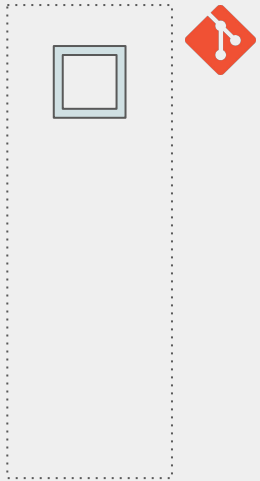
big data workload / other enriched views on the contracts

[3] Shadow Contracts

But they need enrichment for other workload (replaces configuration)

gRPC and consumer / producer workload

master repo



shadow repo

big data workload / other enriched views on the contracts

[3] Shadow Contracts

But they need enrichment for other workload (replaces configuration)

gRPC and consumer / producer workload

```
message Example {
    int64 member = 1 [(anemos.field_meta) = {
        bigQueryClusterIndex: 1
    }];
    string external_id = 2 [(anemos.field_meta) = {
        bigTableRowKey: true
        description: "Description ends up in BigQuery"
    }];
    string old_field = 3 [deprecated = true];
}
```

master repo

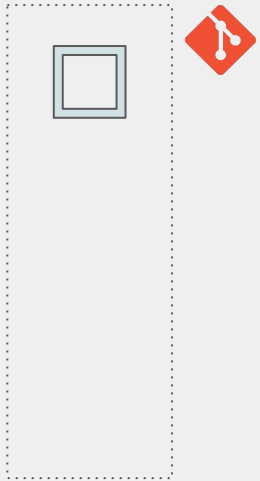
shadow repo

[3] Shadow Contracts

But they need enrichment for other workload (replaces configuration)

gRPC and consumer / producer workload

master repo



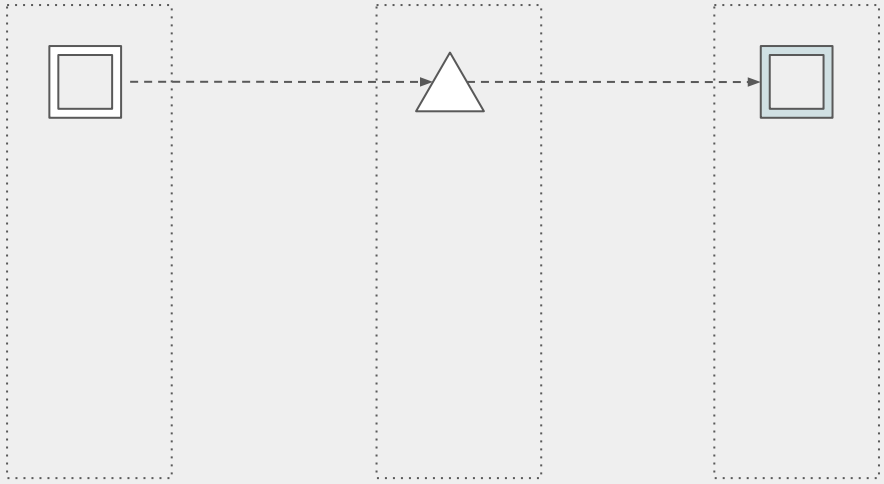
shadow repo

big data workload / other enriched views on the contracts

[3] Shadow Contracts

MetaStore tracks the delta

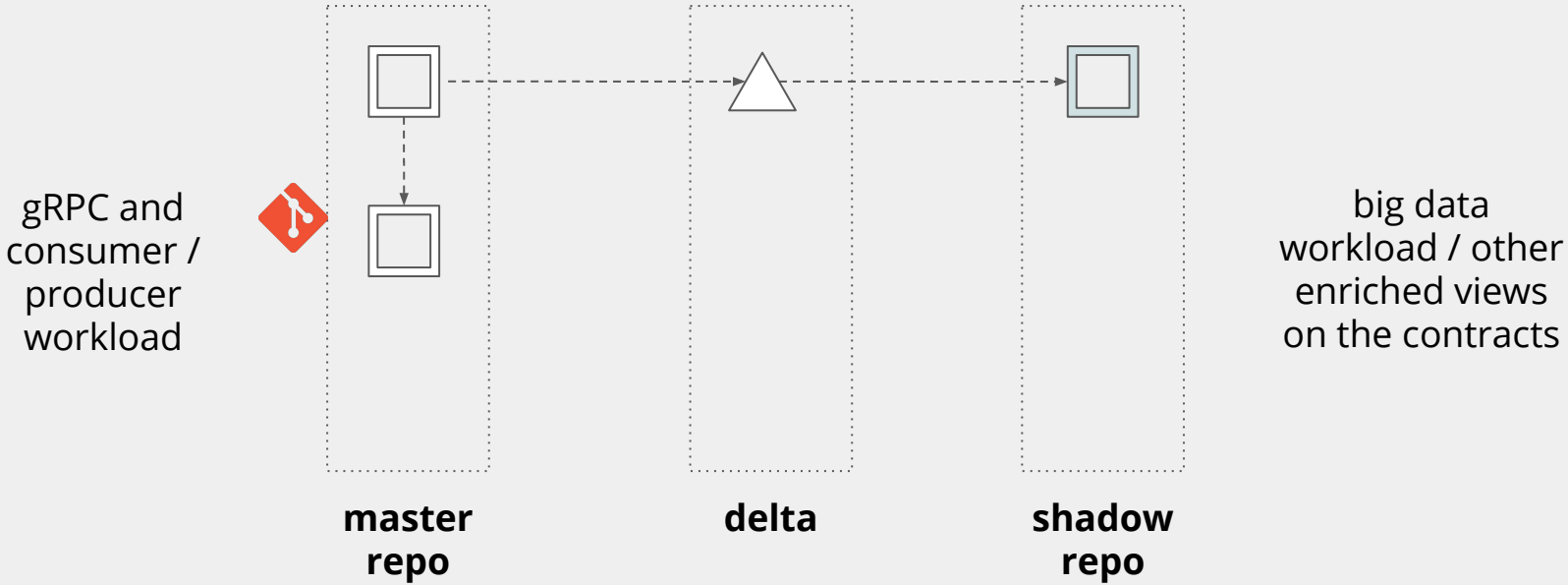
gRPC and consumer / producer workload



big data workload / other enriched views on the contracts

[3] Shadow Contracts

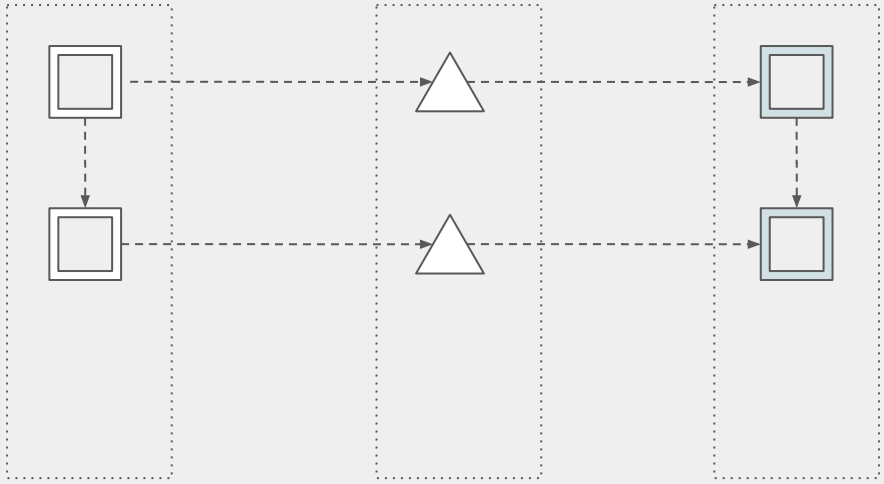
Each time the contract evolves...



[3] Shadow Contracts

... the delta is reapplied

gRPC and consumer / producer workload



master repo

delta

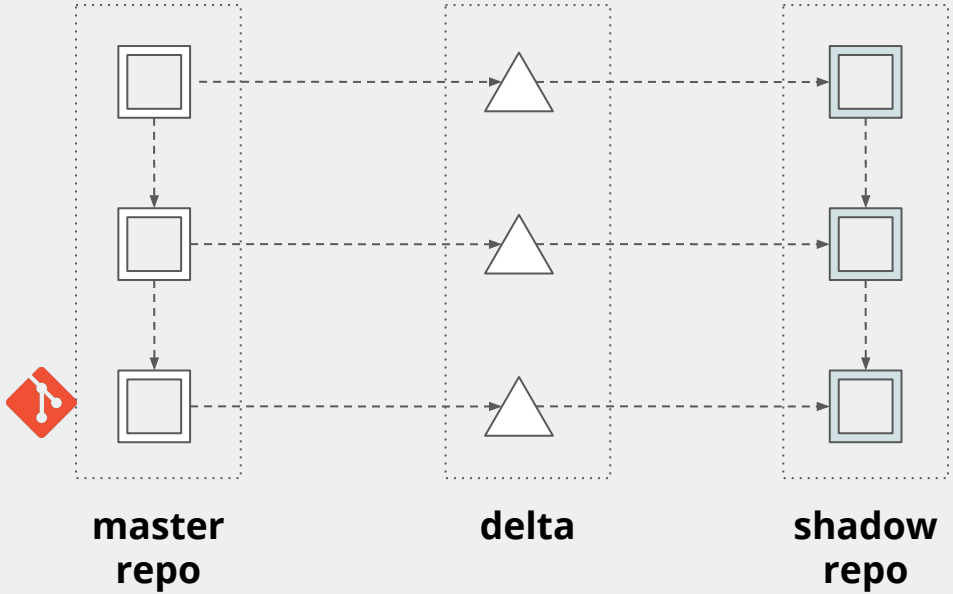
shadow repo

big data workload / other enriched views on the contracts

[3] Shadow Contracts

Every time

gRPC and consumer / producer workload

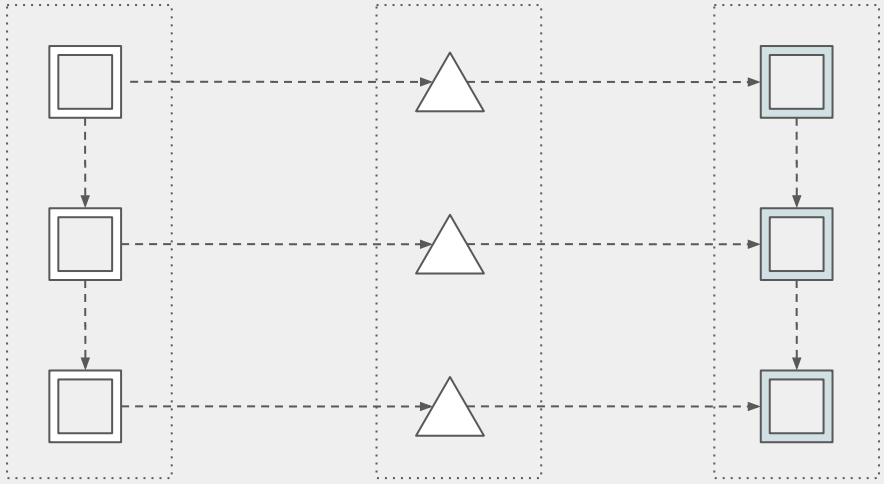


big data workload / other enriched views on the contracts

[3] Shadow Contracts

Every time

gRPC and consumer / producer workload



big data workload / other enriched views on the contracts

Workflow

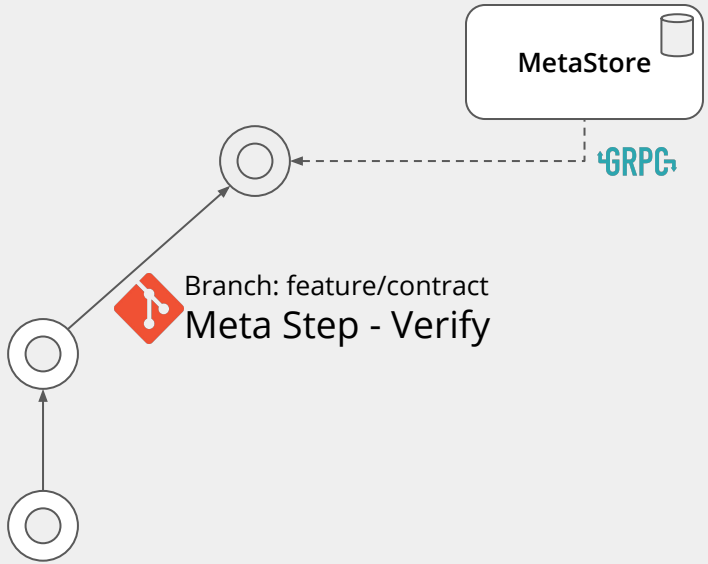
How most people will interact with the MetaStore

[1] MetaStep

Working with contracts on CI/CD pipelines

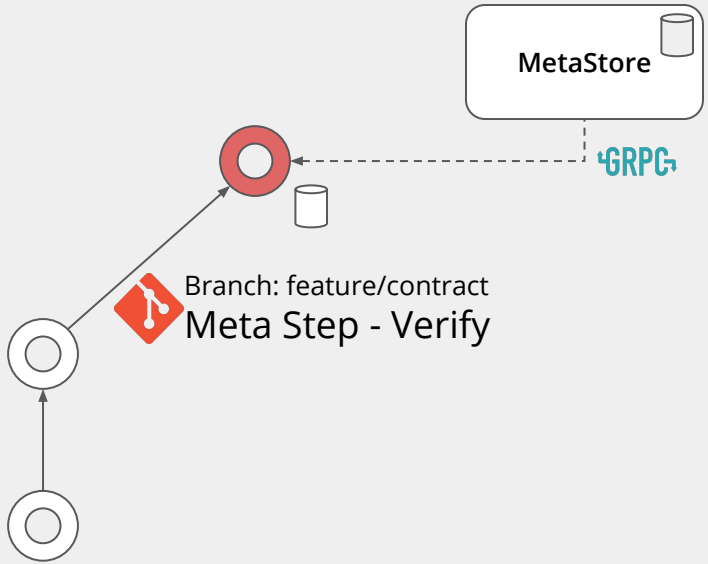
[1] Repo - Metastep

Edits on contracts are done in a branch, the branch is pushed



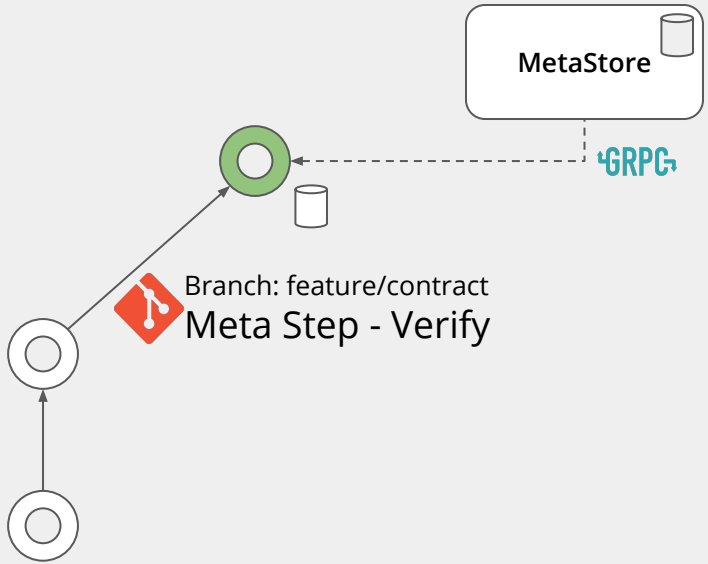
[1] Repo - Metastep

MetaStep - will check against the master contracts for breaking changes



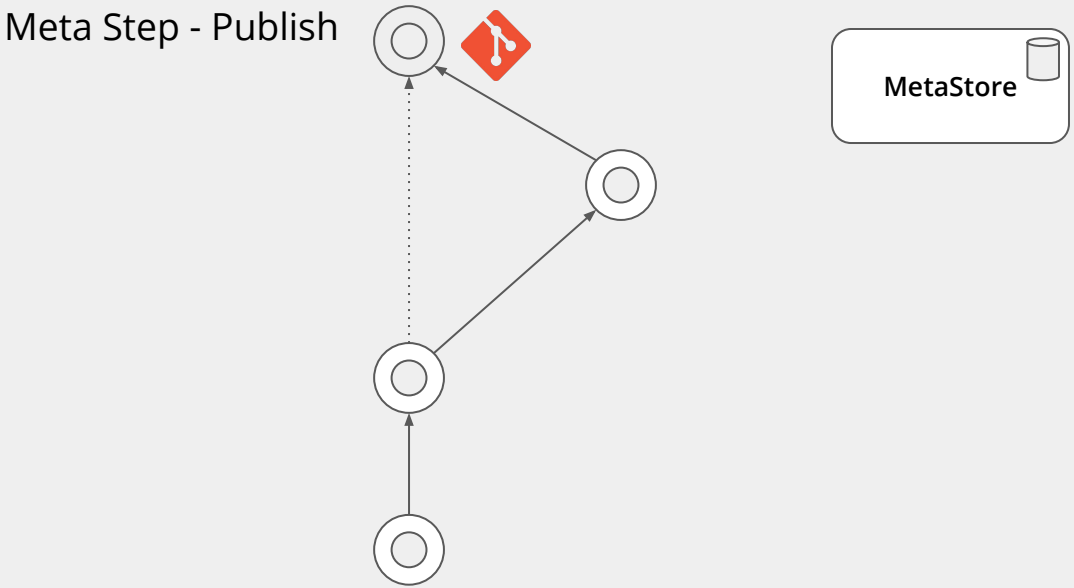
[1] Repo - Metastep

MetaStep - if it succeeds, allowed to merge to master



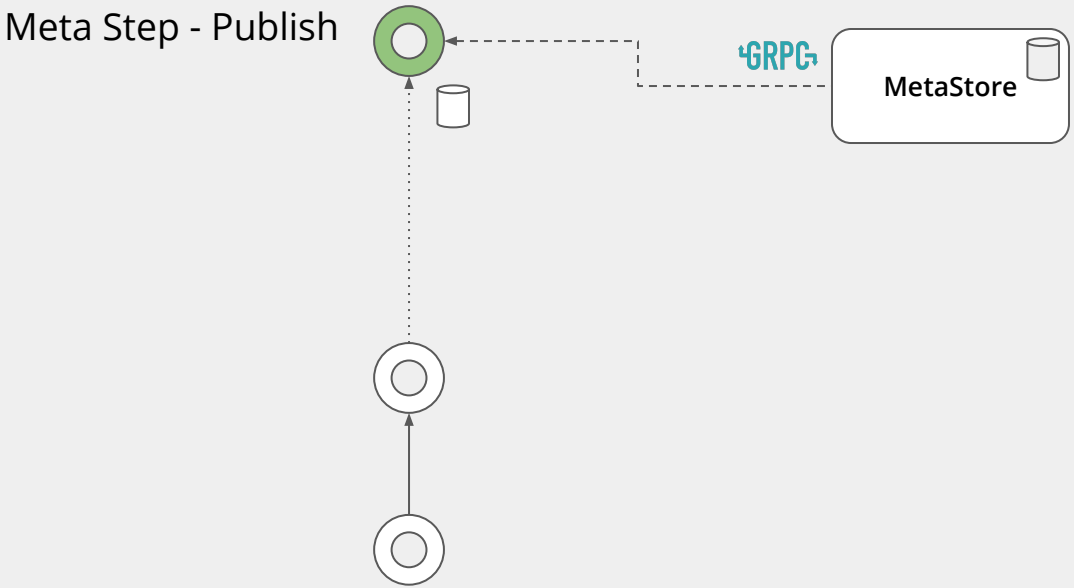
[1] Repo - Metastep

Manual merge in your CI tool of choice



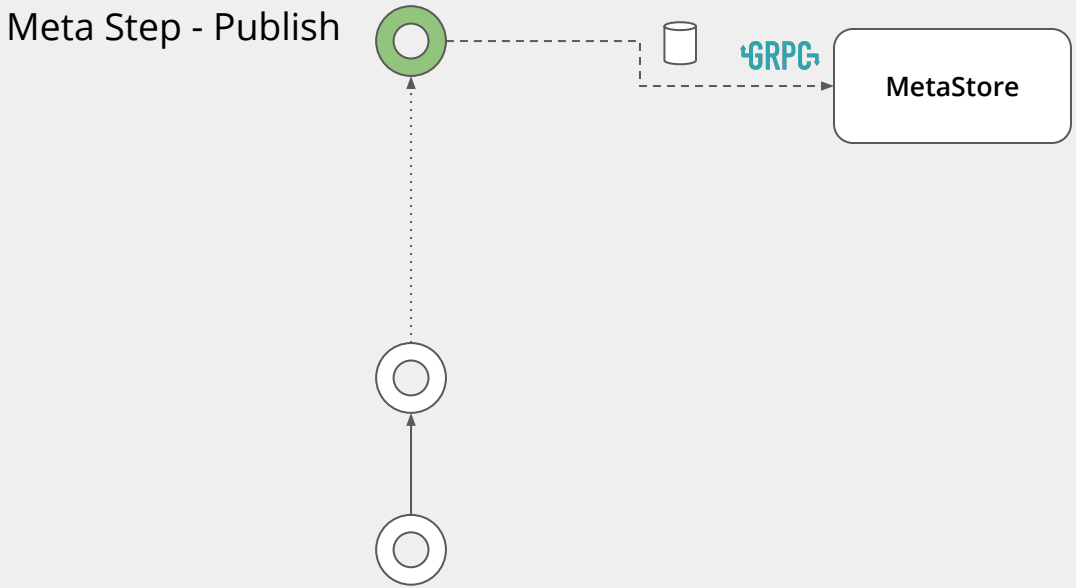
[1] Repo - Metastep

Build pipeline on master kicks in



[1] Repo - Metastep

If succeed the master is published



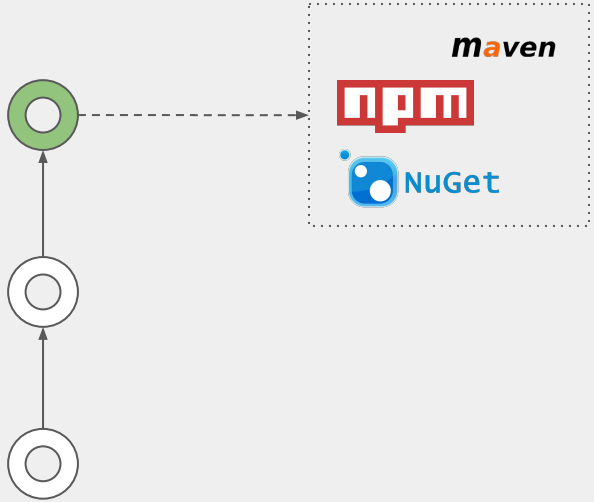
[1] Repo - Metastep

The system is synced

Synced with
MetaStore

[1] Repo - Metastep

Extra optional buildstep could be publishing the contracts to the repo’s

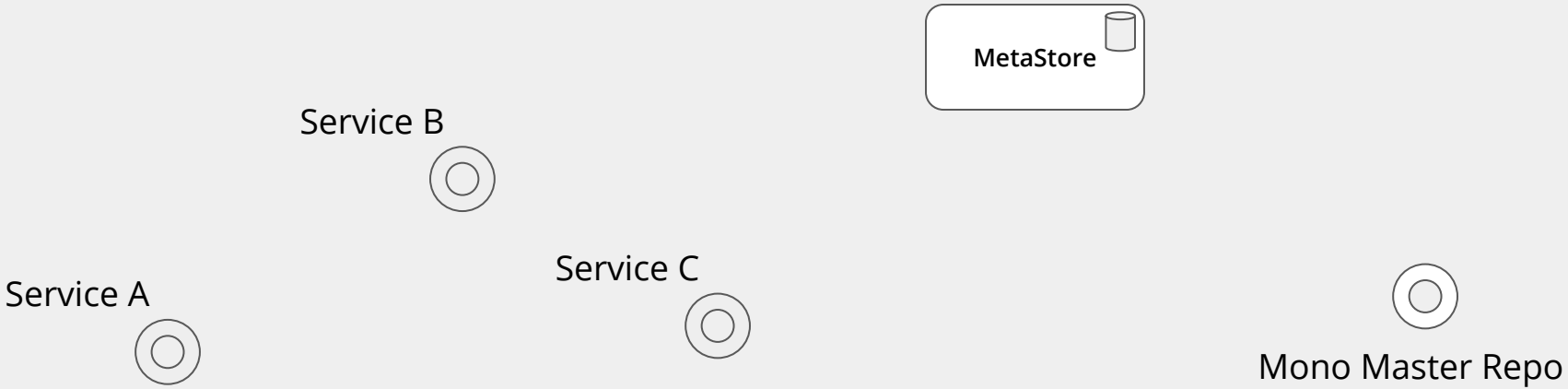


[2] Publishing Contracts

Microservices own the original contracts but publish the contracts

[2] Repo - Publish contracts

Microservices have their own contracts, Metastore has the world view, Git has a readable view



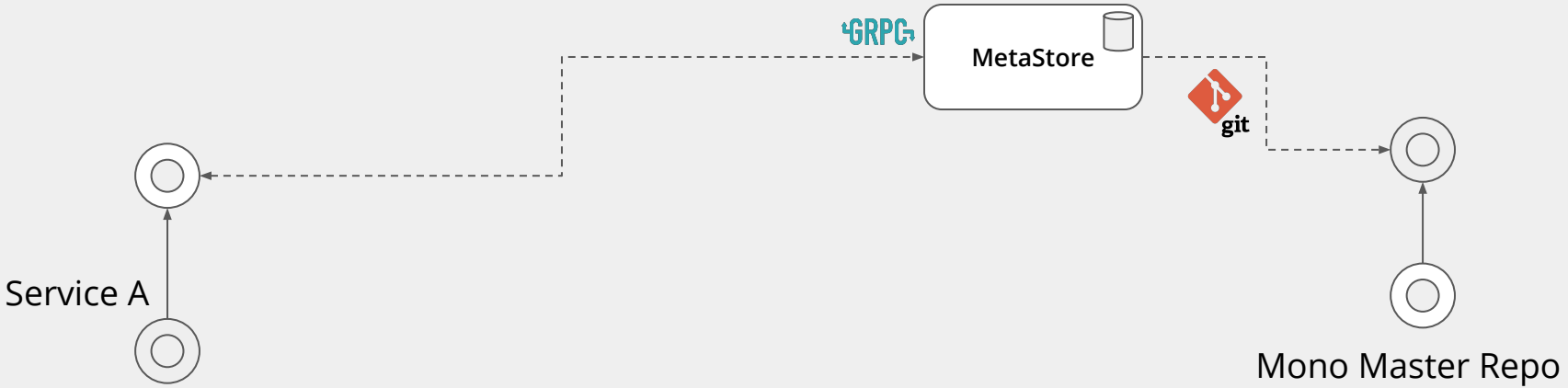
[2] Repo - Publish contracts

Contract owner verifies and publishes her contracts

Mono Master Repo

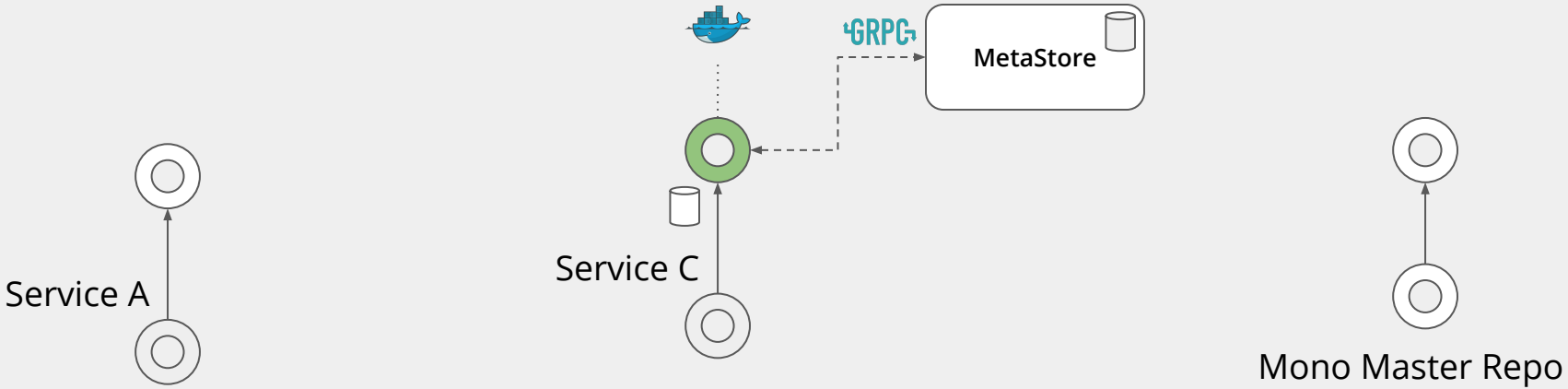
[2] Repo - Publish contracts

MetaStore has no UI, but after publish it will recreate the contracts in the master mono repo



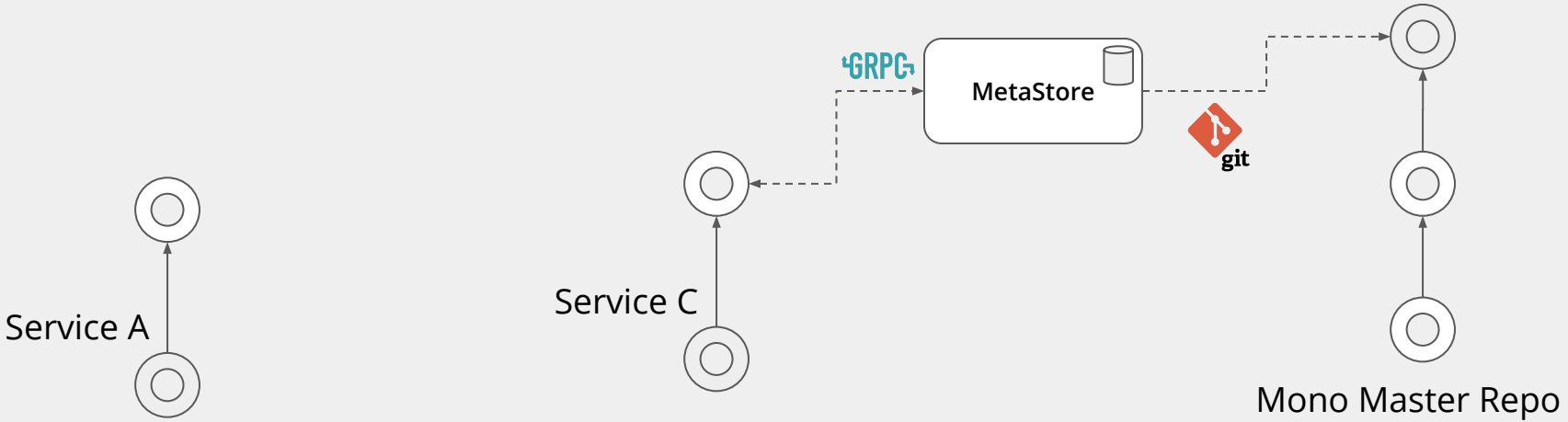
[2] Repo - Publish contracts

Each owner has his own **scoped** contracts, scope is important



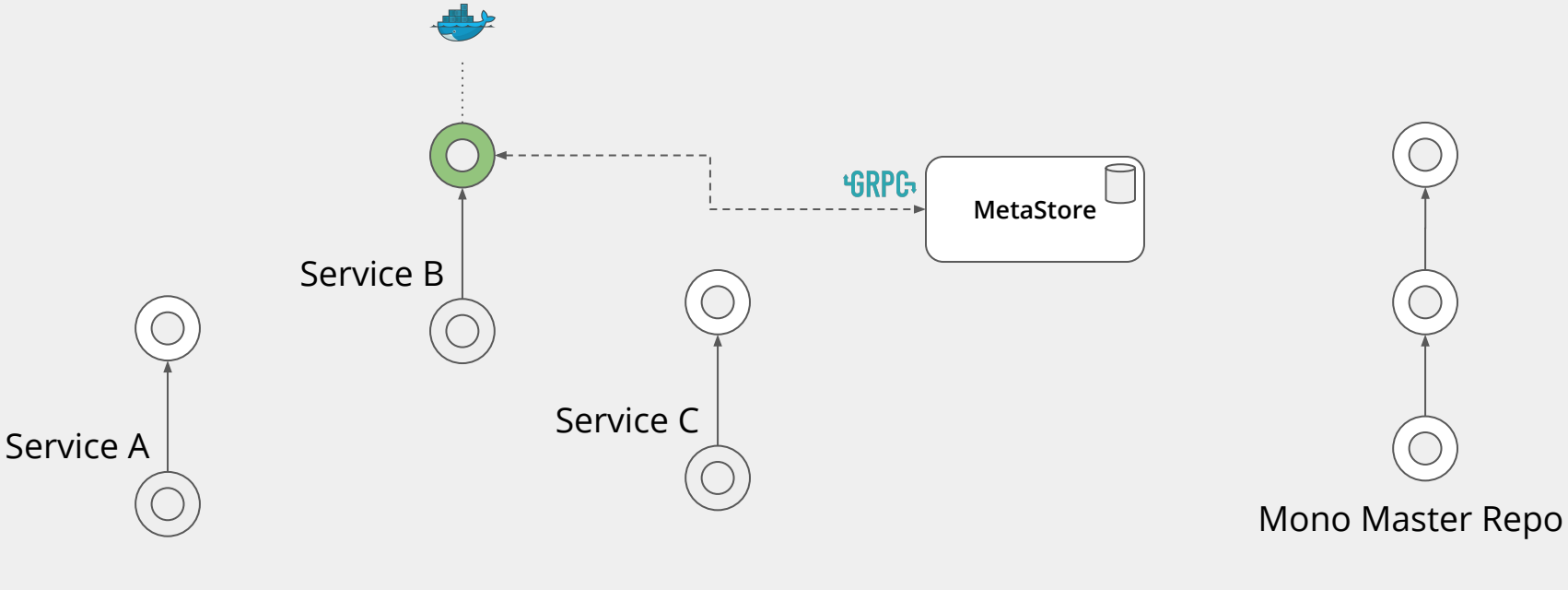
[2] Repo - Publish contracts

Again the contracts are written to mono repo



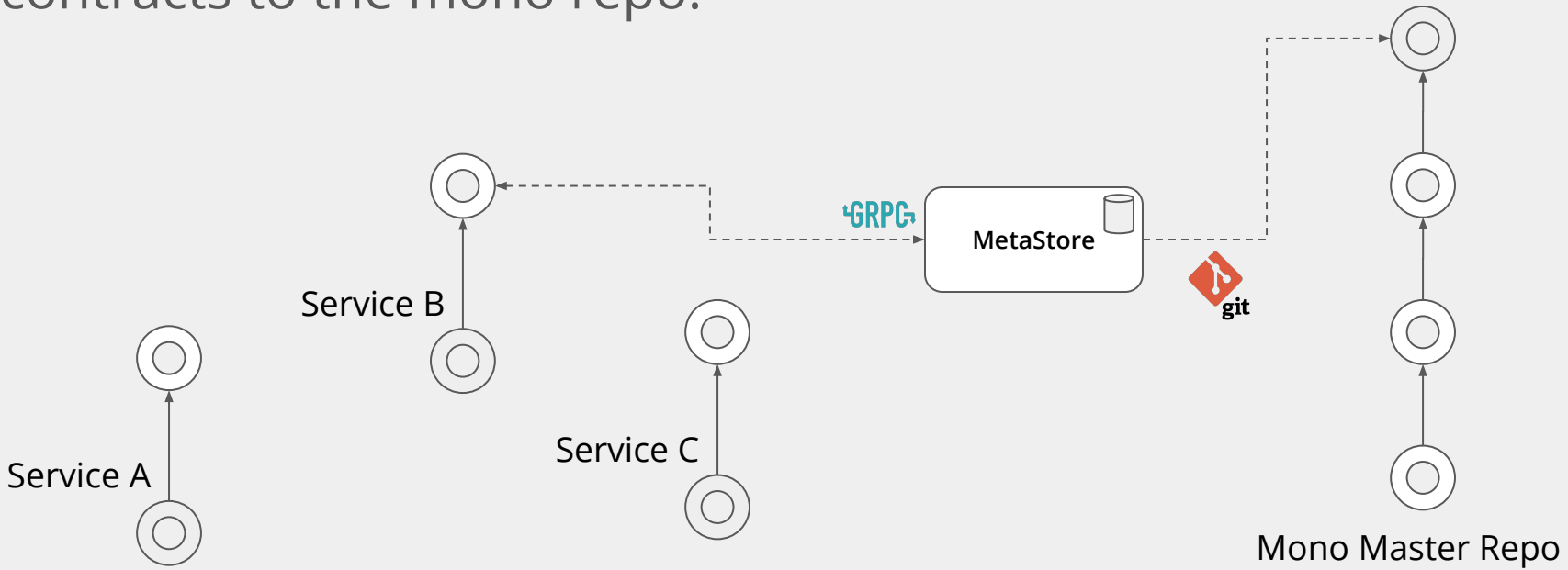
[2] Repo - Publish contracts

Each component has a master of contracts - Publish the contracts to the mono repo.



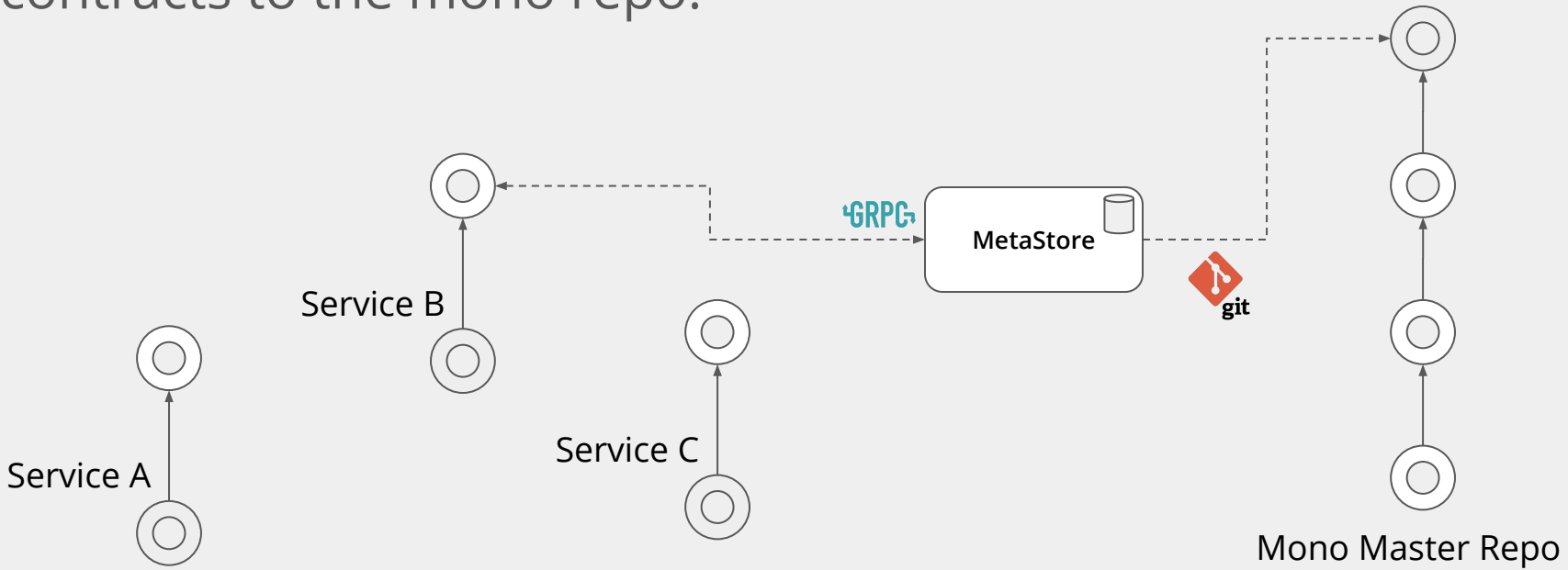
[2] Repo - Publish contracts

Each component has a master of contracts - Publish the contracts to the mono repo.



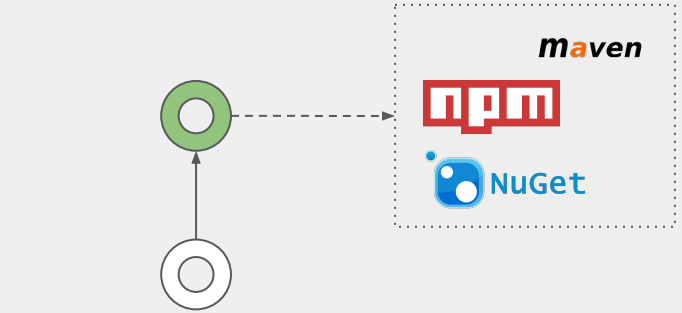
[2] Repo - Publish contracts

Each component has a master of contracts - Publish the contracts to the mono repo.



[2] Repo - Publish contracts

Each component has a master of contracts - Publish the contracts to the mono repo.



Architecture

How it works

Protos best kept secret: Descriptors

Descriptors are your proto contracts, parsed and stored in...

```
protoc \
 -Itestsets/test1 \
 -I/usr/local/include \
 -I$GOOGLEAPIS_DIR \
 --descriptor_set_out=tmp/test1.pb \
 testsets/test1/test/v1alpha1/simple.proto
```

https://developers.google.com/protocol-buffers/docs/techniques#self-description

Protos best kept secret: Descriptors

Descriptors are your proto contracts, parsed and stored in **proto** (say what?!)

```
protoc \
 -Itestsets/test1 \
 -I/usr/local/include \
 -I$GOOGLEAPIS_DIR \
 --descriptor_set_out=tmp/test1.pb \
 testsets/test1/test/v1alpha1/simple.proto
```

https://developers.google.com/protocol-buffers/docs/techniques#self-description

Protos best kept secret: Descriptors

As it’s part of the specification, all tools support it… **Gradle**

```
{ task ->
  task.generateDescriptorSet = true
  task.descriptorSetOptions.includeSourceInfo = true
  task.descriptorSetOptions.includeImports = true
}
```

https://github.com/google/protobuf-gradle-plugin

Protos best kept secret: Descriptors

Bazel

When compiled on the command-line, a `proto_library` creates a file named `foo-descriptor-set.proto.bin`, which is the descriptor set for the messages the rule srcs. The file is a serialized `FileDescriptorSet`, which is described in https://developers.google.com/protocol-buffers/docs/techniques#self-description.

https://docs.bazel.build/versions/master/be/protocol-buffer.html

Protos best kept secret: Descriptors

Available almost everywhere... Every class has static metadata embedded

```
package io.anemos.protobeam.examples;

public final class Basic {
  private Basic() {}
...
  static {
    java.lang.String[] descriptorData = {
     "\n%anemos/protobeam/examples/basic.proto\022" +
     "\031anemos.protobeam.examples\"\321\001\n\025ProtoBeam" +
     "BasicMessage\022\021\n\ttest_name\030\001 \001(\t\022\022\n\ntest_" +
     "index\030\002 \001(\005\022C\n\007message\030\003 \001(\01322.anemos.pr" +
     "otobeam.examples.ProtoBeamBasicPrimitive" +
     "\022L\n\020repeated_message\030\004 \003(\01322.anemos.prot" +
...
```

Protos best kept secret: Descriptors

GRPC Server Reflection Protocol - server reflection as an optional extension for servers to assist clients in runtime construction of requests without having stub information precompiled into the client

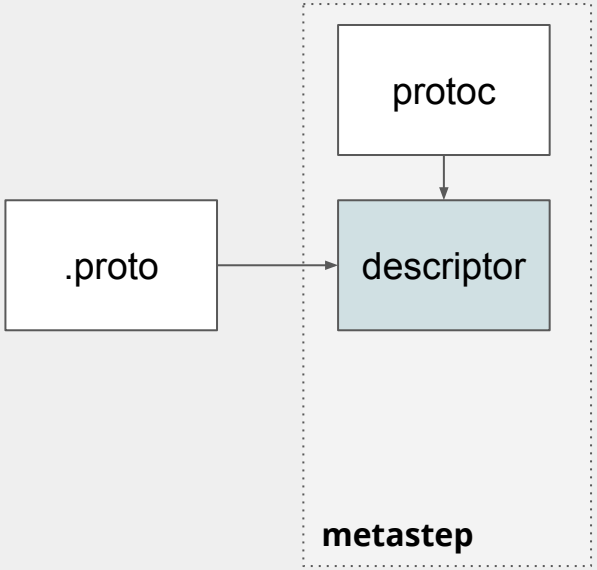
https://github.com/grpc/grpc/blob/master/doc/server-reflection.md

Descriptors

the backbone of the MetaStore architecture

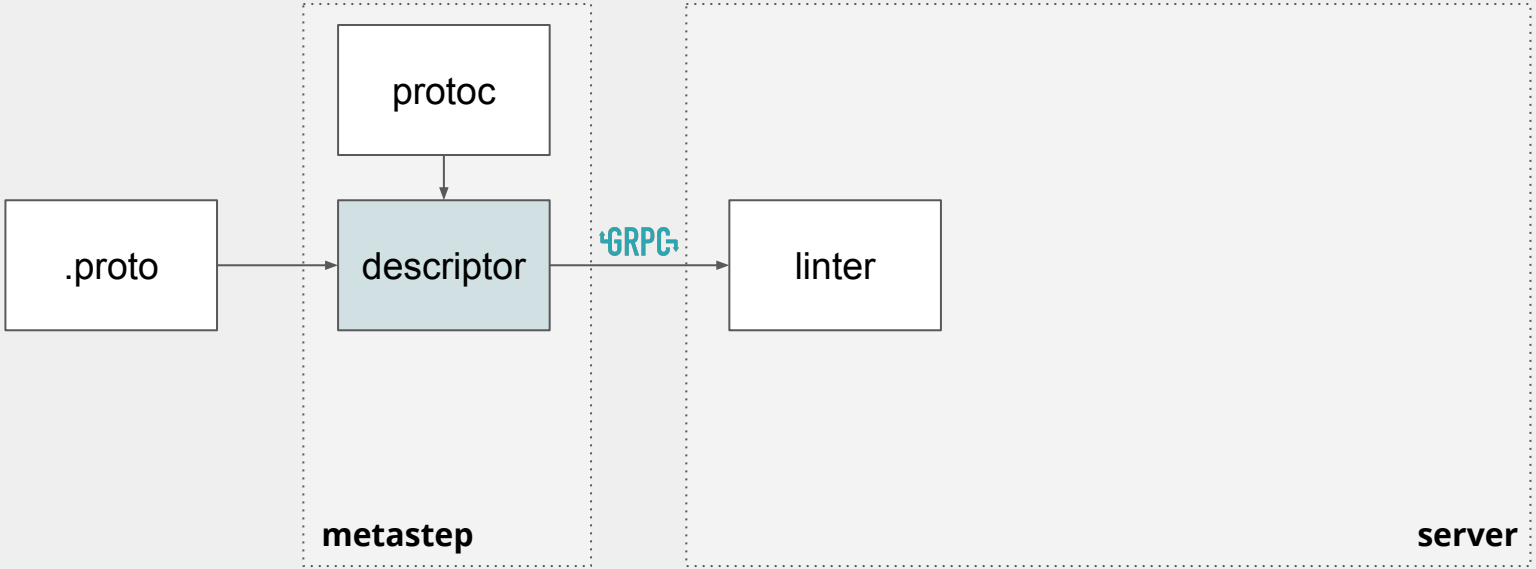
MetaStore - Architecture

High level architecture



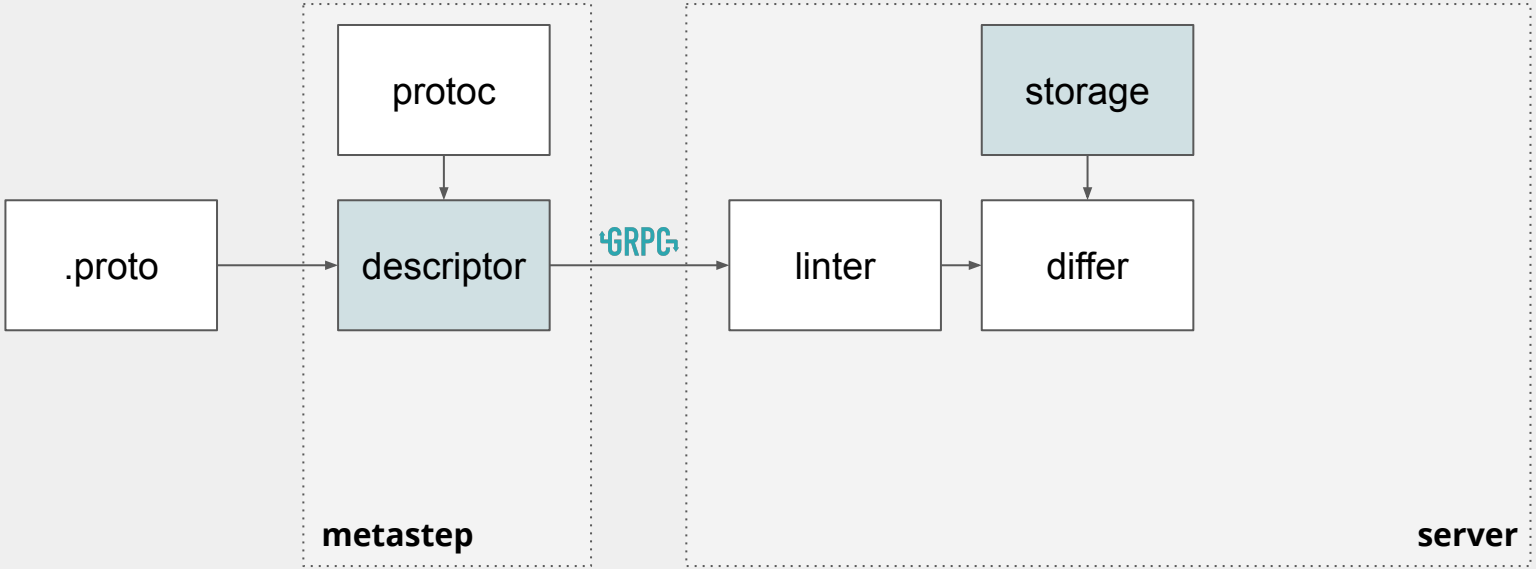
MetaStore - Architecture

High level architecture



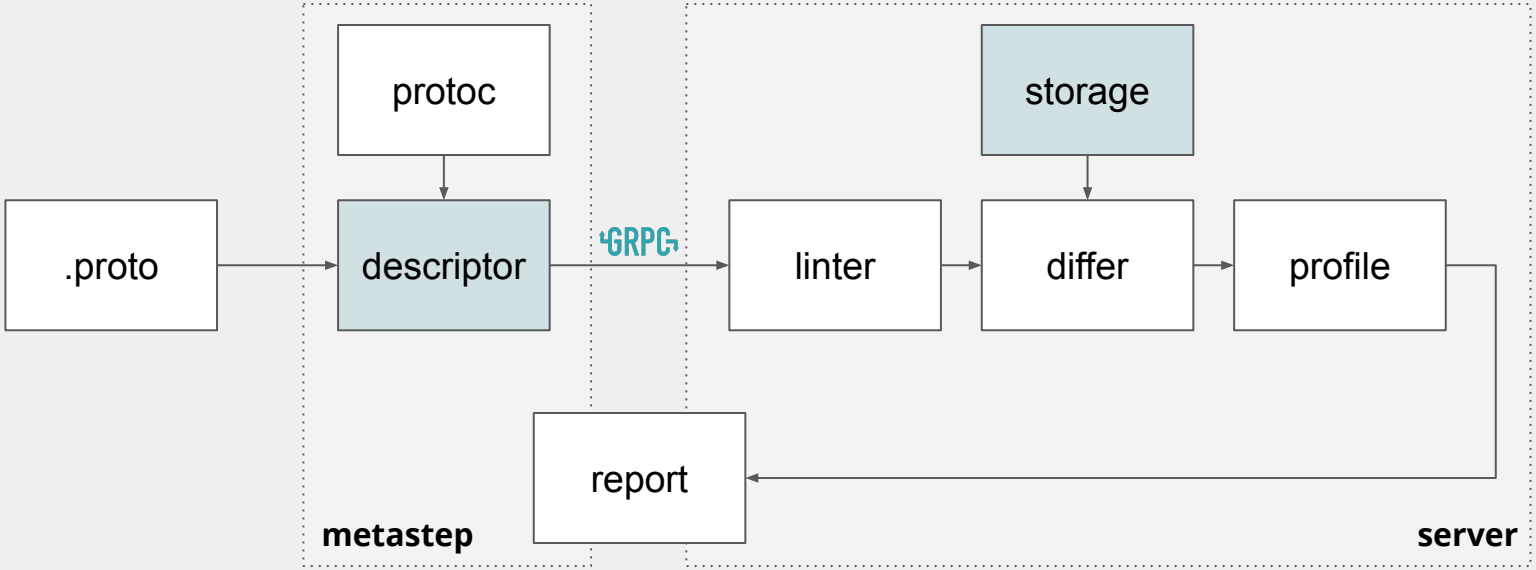
MetaStore - Architecture

High level architecture



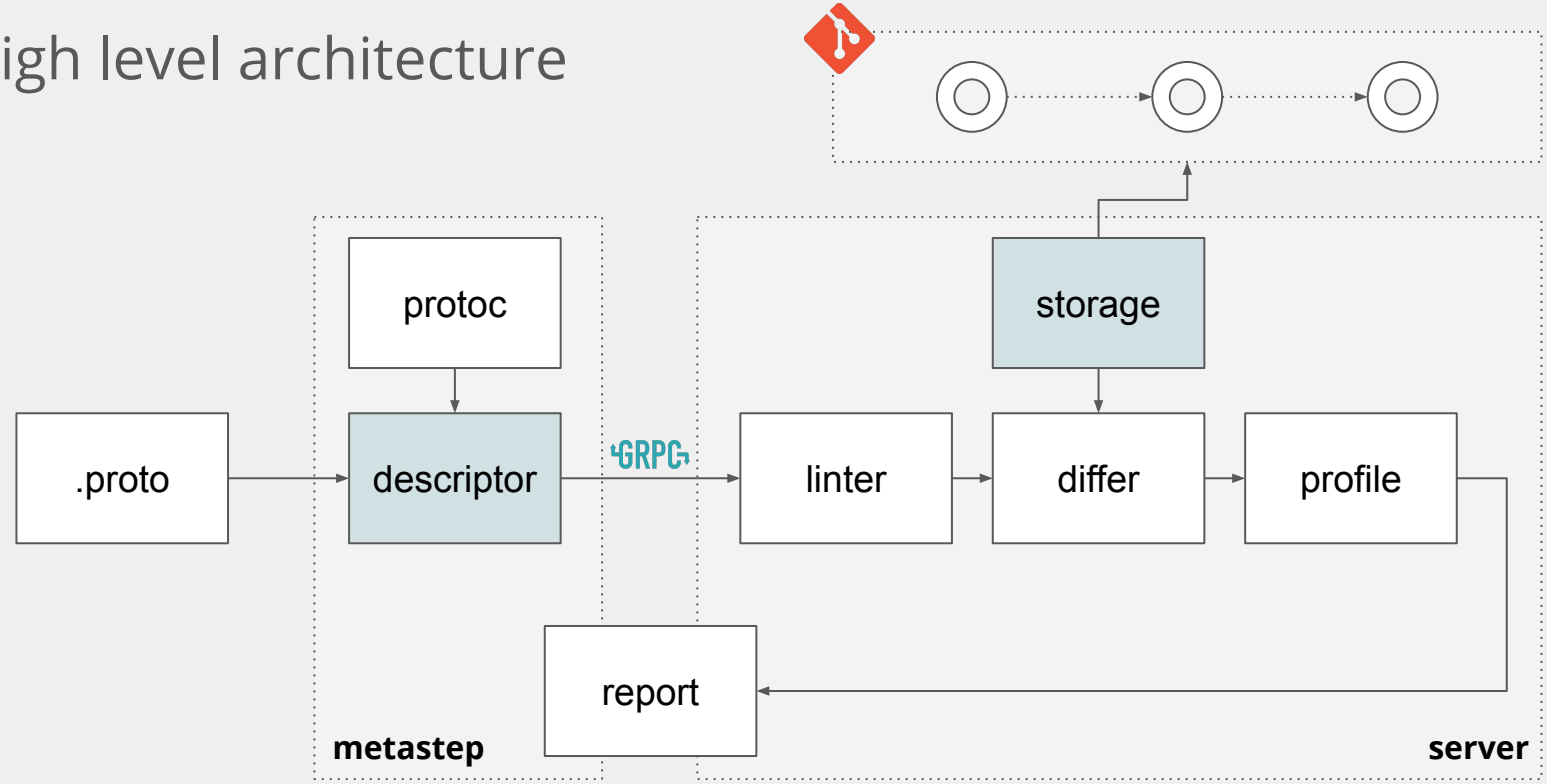
MetaStore - Architecture

High level architecture



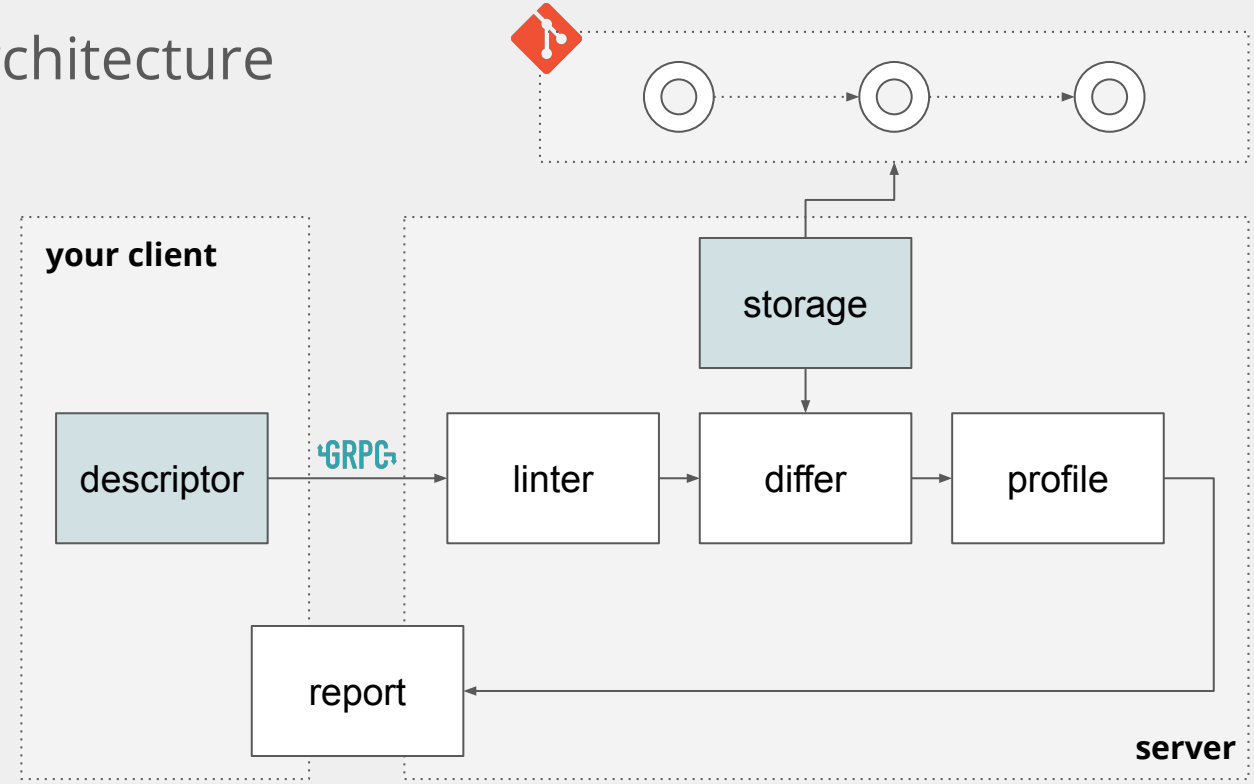
MetaStore - Architecture

High level architecture



MetaStore - Architecture

High level architecture



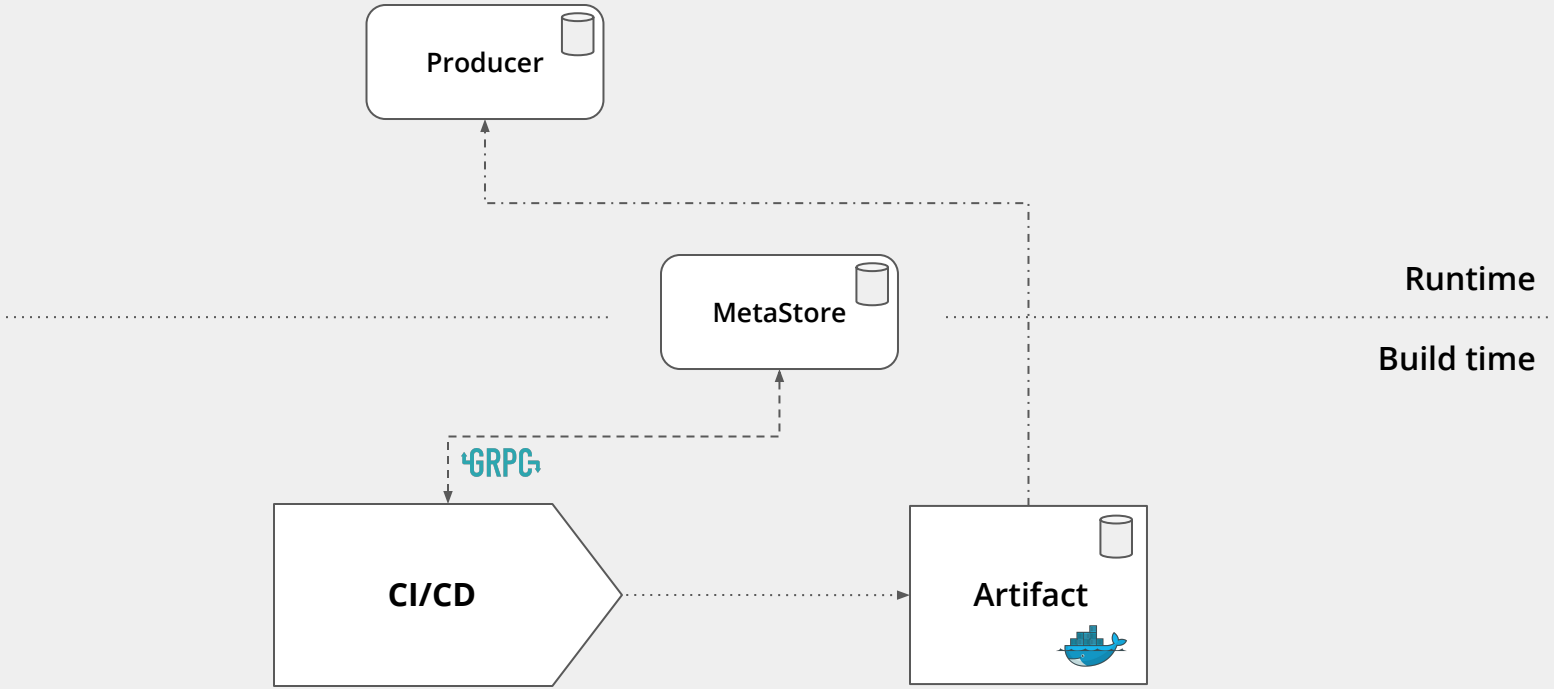
Use

What a registry enables

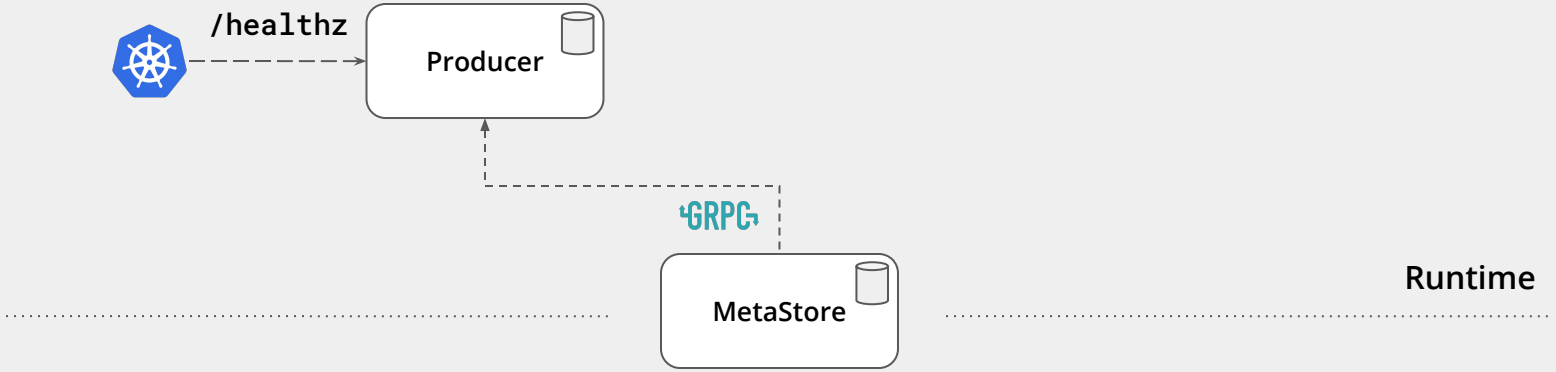
[1] Auditing

Auditing, Monitoring and Deprecation

Runtime auditing: Producer > Consumer

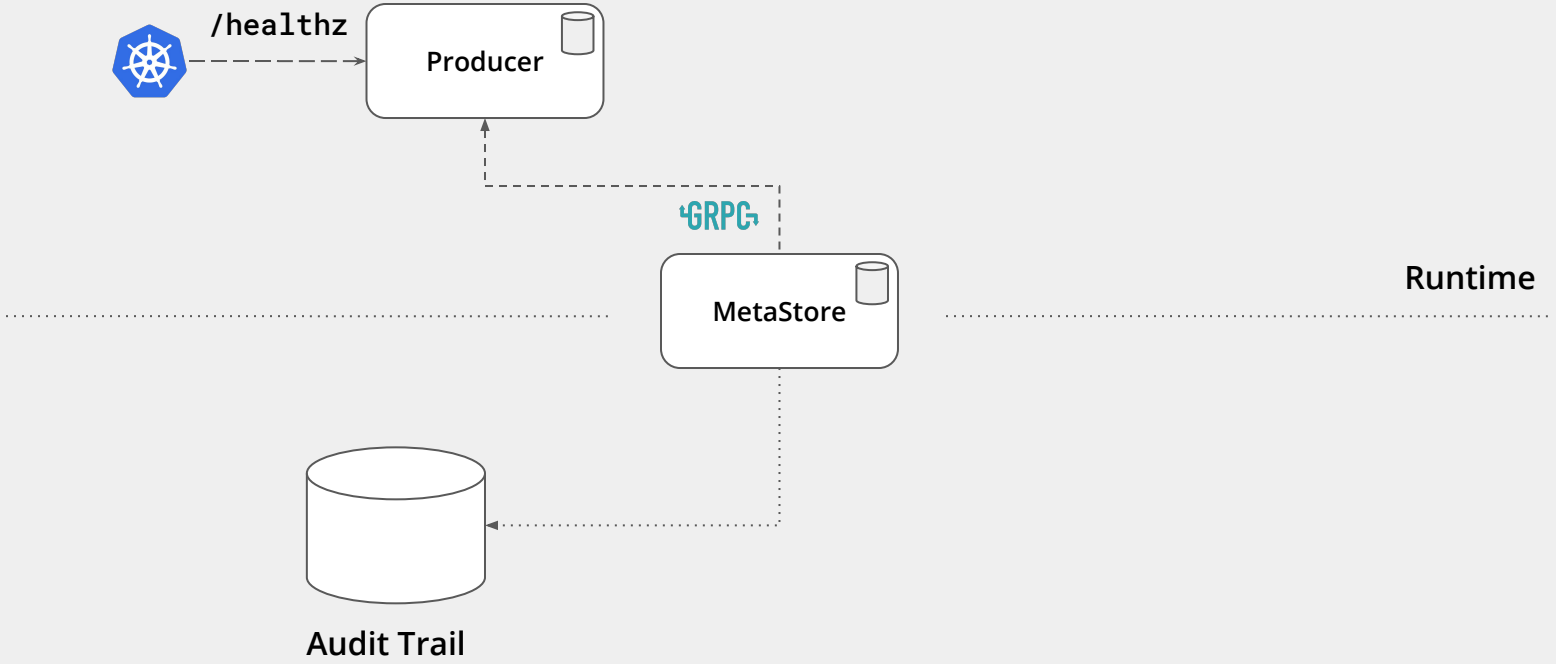


Runtime auditing: Producer > Consumer

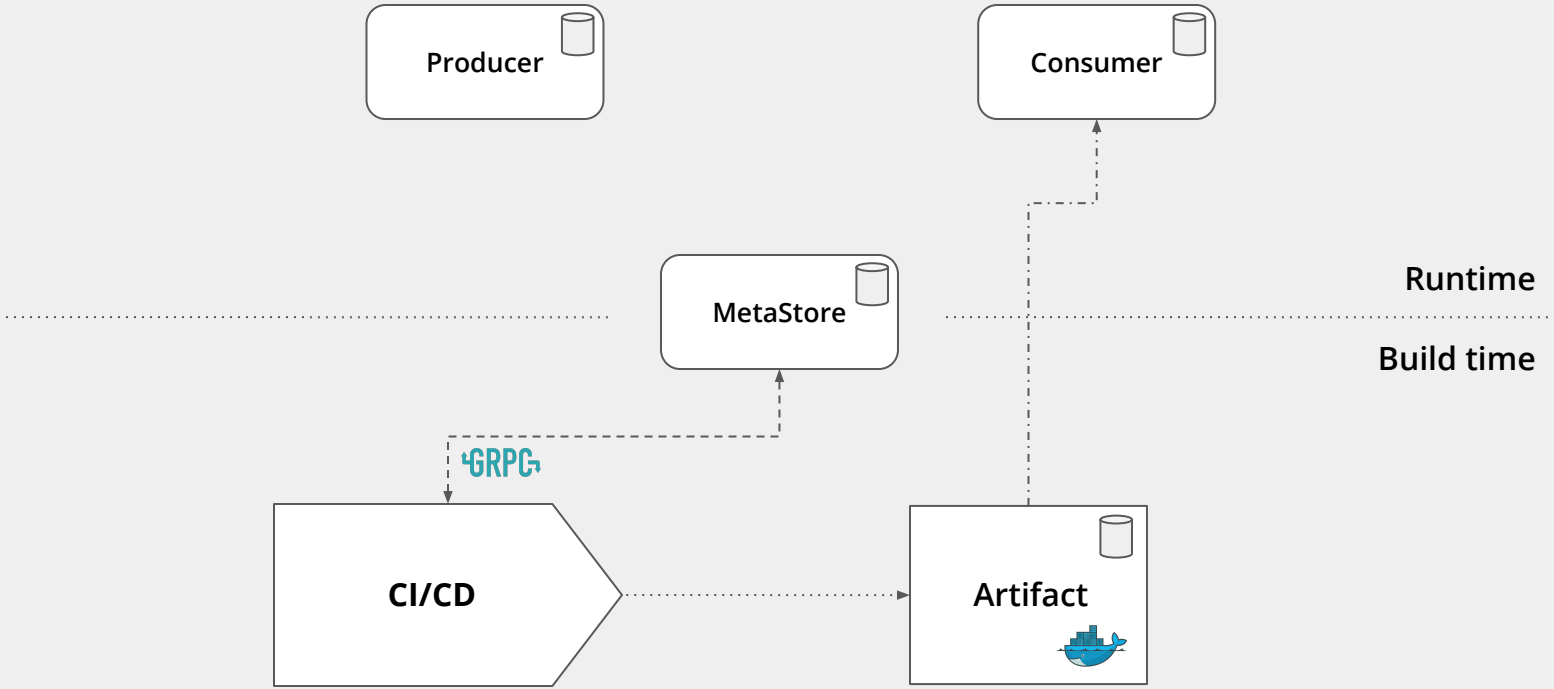


Standard libraries can **prevent** the component to ever start serving traffic. A good example is using the Kubernetes **liveness probe**, Wrong contract, you will get into a crash loop.

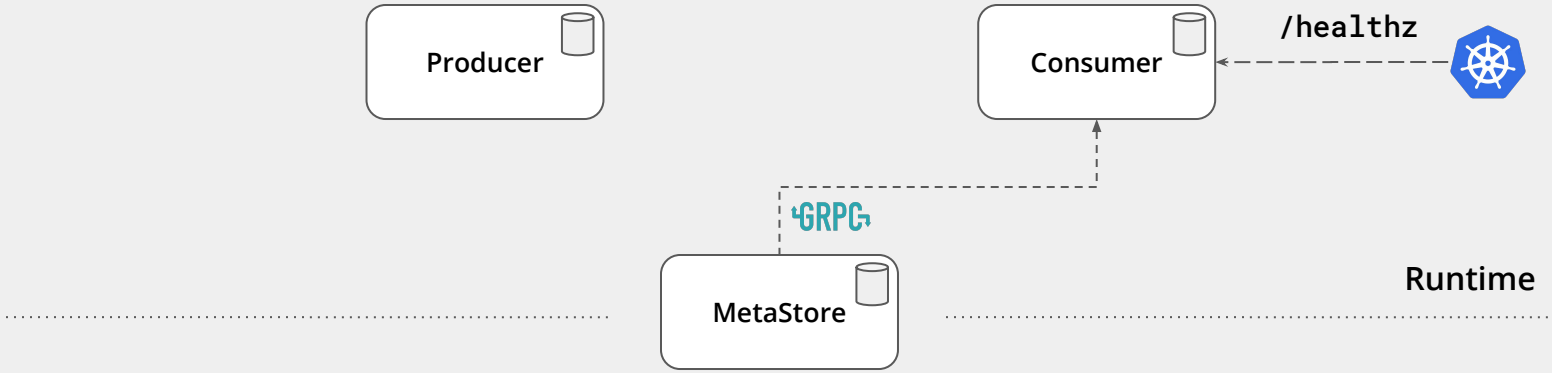
Runtime auditing: Producer > Consumer



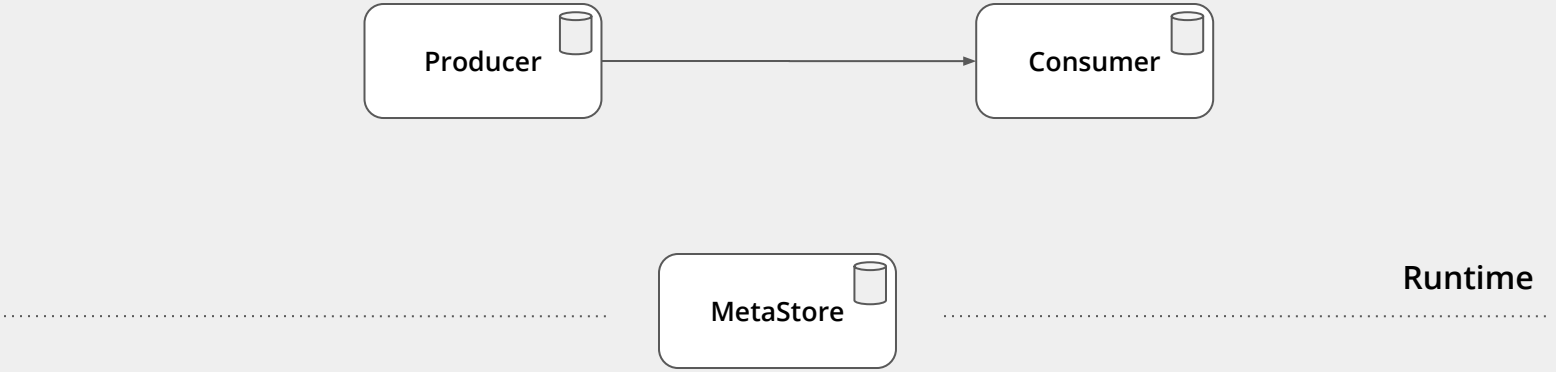
Runtime auditing: Producer > Consumer



Runtime auditing: Producer > Consumer

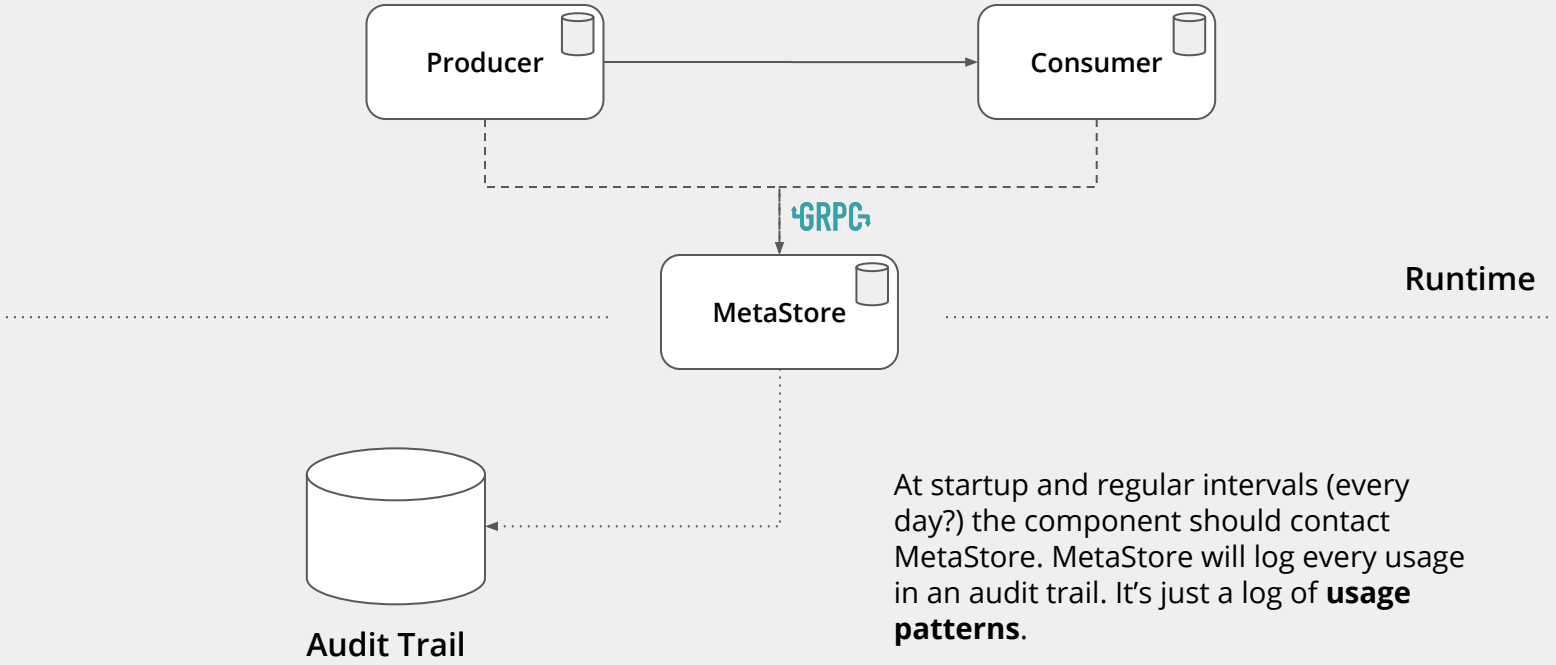


Runtime auditing: Producer > Consumer



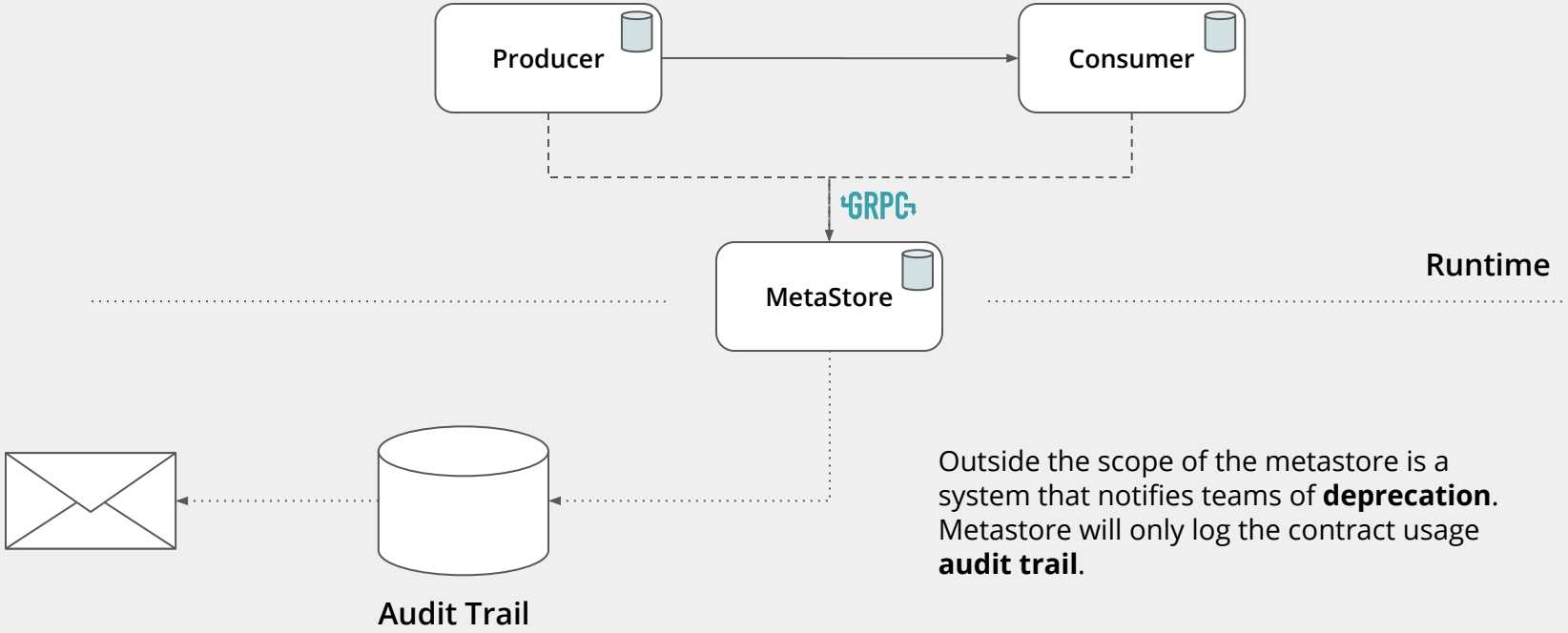
Once startet the component don’t need the store anymore

Runtime auditing: Producer > Consumer



At startup and regular intervals (every day?) the component should contact MetaStore. MetaStore will log every usage in an audit trail. It's just a log of **usage patterns**.

Runtime auditing: Producer > Consumer



Outside the scope of the metastore is a system that notifies teams of **deprecation**. Metastore will only log the contract usage **audit trail**.

Runtime auditing: Future

Action Required

Hello team clearance,

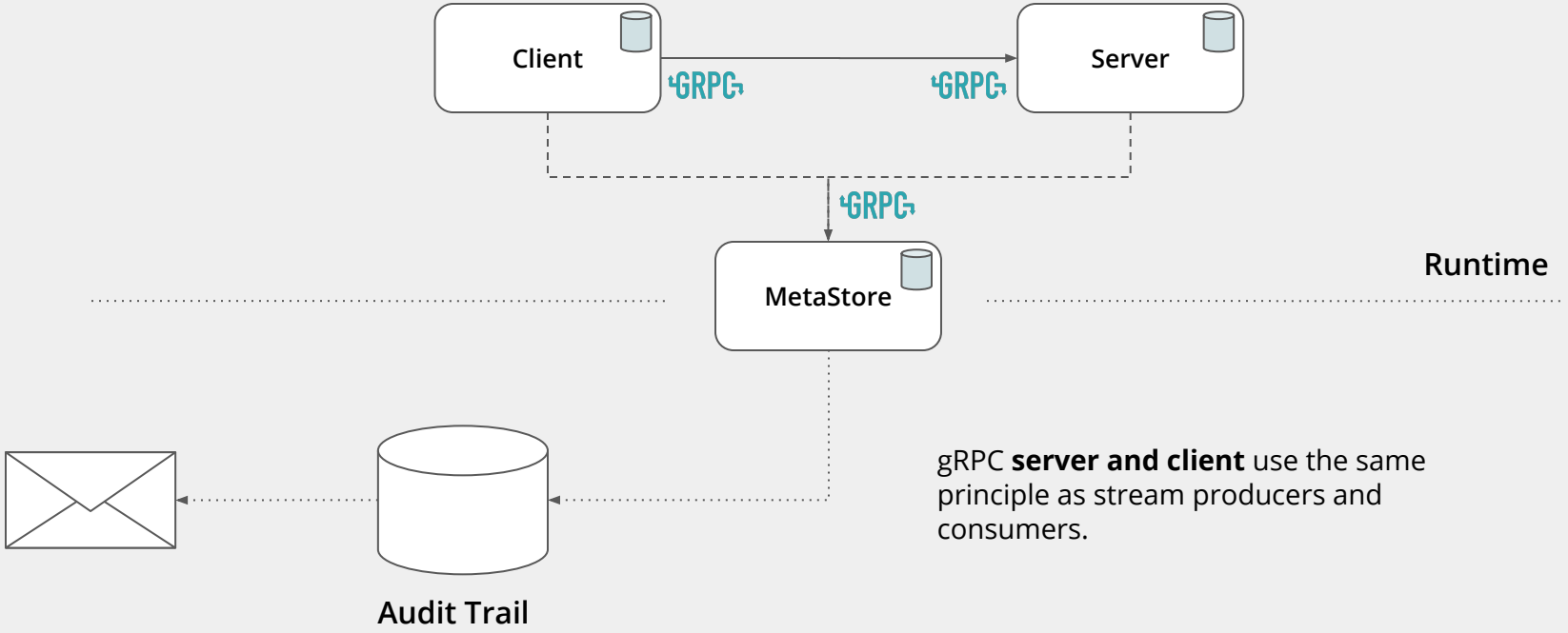
You are using **example.checkoutexp.cart.v1** api, the api is deprecated and is marked for removal from **2021-12-12**.

We have detected the the following modules are using this api:

- **datascience-recsys-api**
- **Data-backup-beam**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Runtime auditing: **gRPC** client > server

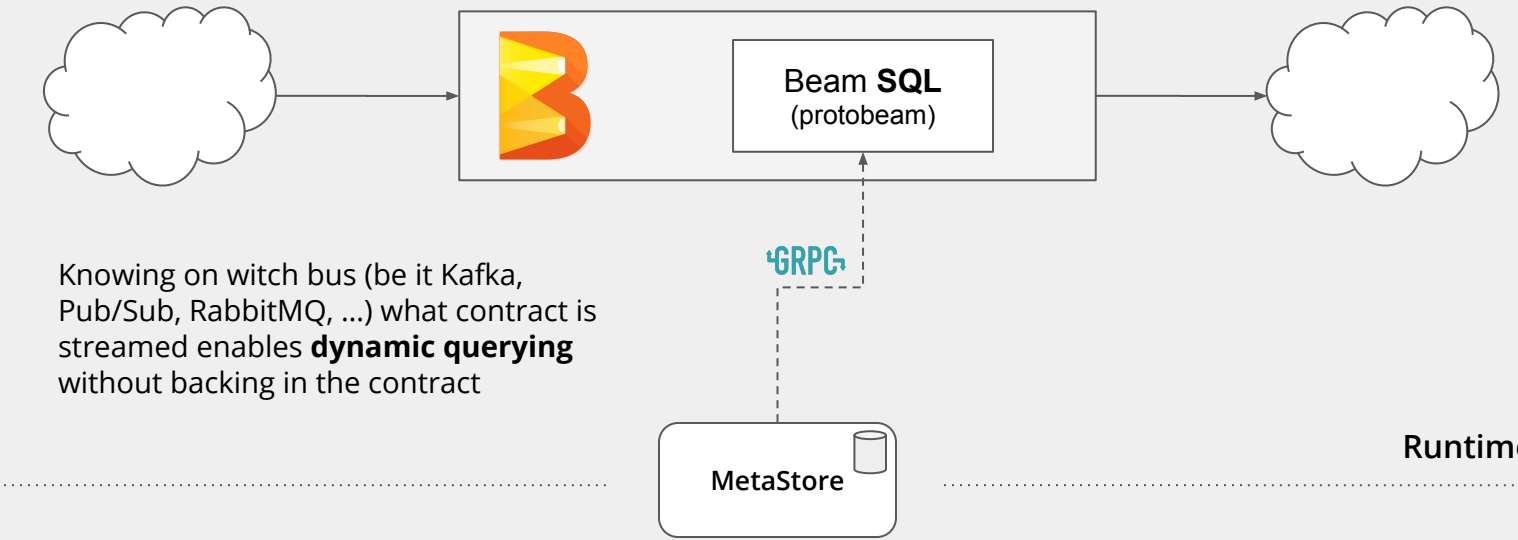


gRPC **server and client** use the same principle as stream producers and consumers.

[2a] Apache Beam

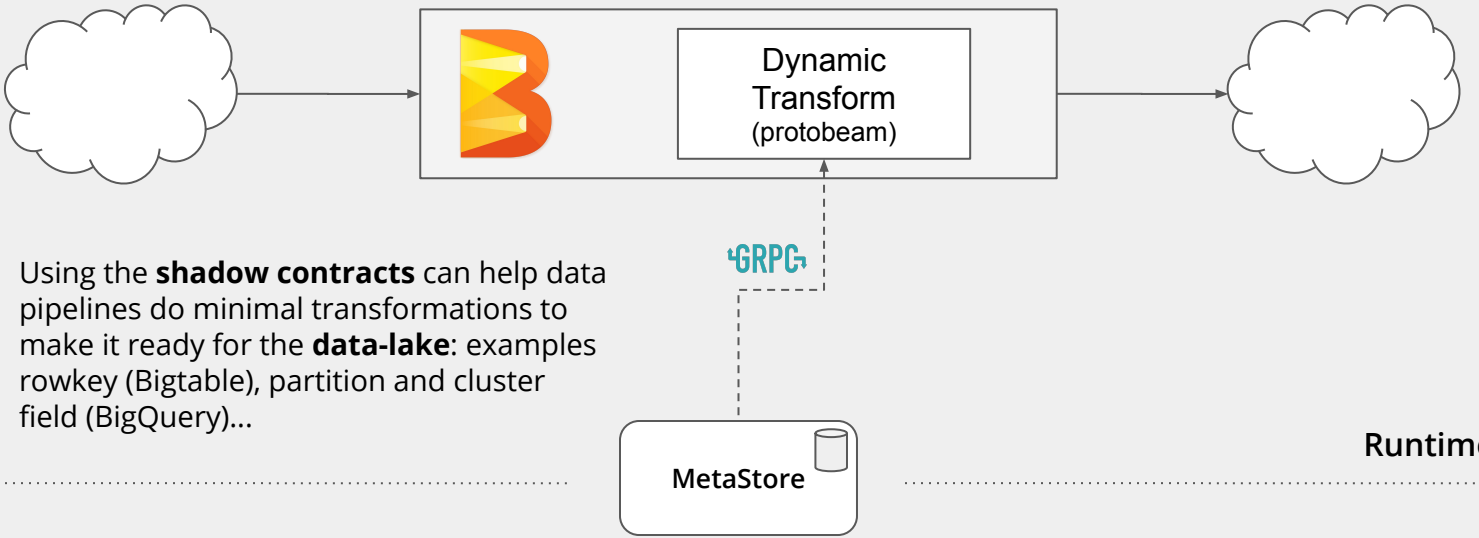
Dynamic Beam and Beam SQL

Apache Beam - Beam SQL



Knowing on witch bus (be it Kafka, Pub/Sub, RabbitMQ, ...) what contract is streamed enables **dynamic querying** without backing in the contract

Apache Beam - Beam SQL

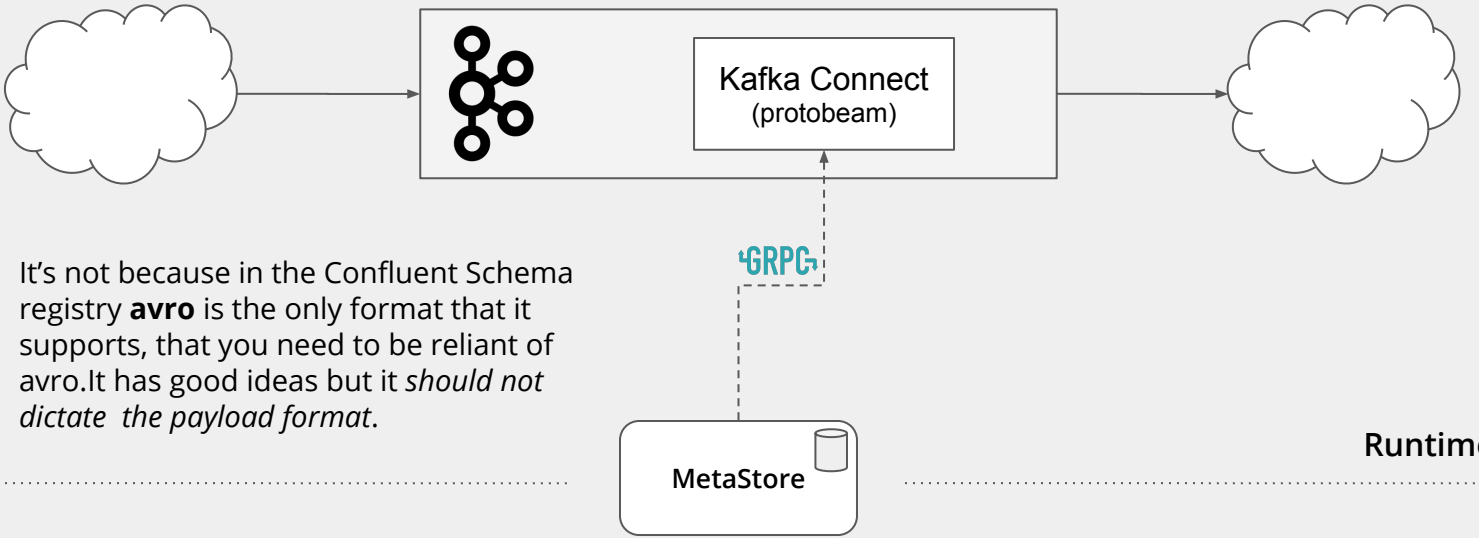


Using the **shadow contracts** can help data pipelines do minimal transformations to make it ready for the **data-lake**: examples rowkey (Bigtable), partition and cluster field (BigQuery)...

[2b] Apache Kafka

A messaging technology should not dictate its payload format

Apache Kafka doesn’t require avro



It’s not because in the Confluent Schema registry **avro** is the only format that it supports, that you need to be reliant of avro.It has good ideas but it *should not dictate the payload format*.

Take away

Conclusion anyone?

MetaStore can help you...

- Safeguard schema changes, evolution defined by profile
- Helps you align on best practices in an organisation
- Allows different way of working
  - Mono master repo and run as a build step
  - Publish from owning components to the master repo

But a standard API can...

Unlock a lot of useful dynamic use-cases

(did you find every gRPC logo?)

- Auditing
- Dynamic Pipelines

But we **need you** to help define the API.

Join in the conversation

- https://github.com/anemos-io/metastore the store and temporary home for the api (it will break daily)
- https://github.com/anemos-io/proto-beam will connect the Apache Beam to the metastore (probably will get a Kafka edition as well)
- Contact me alex@vanboxel.be to start a core API team, we'll take it from there
- Early API proposal doc, full of typo's