Breaking the Language Barrier: Easy Cross-Language with Generated Python Wrappers

Ahmed Abualsaud

BEAM SUMMIT

September 4-5, 2024

Sunnyvale, CA. USA

About me

Google BigQuery

ICEBERG

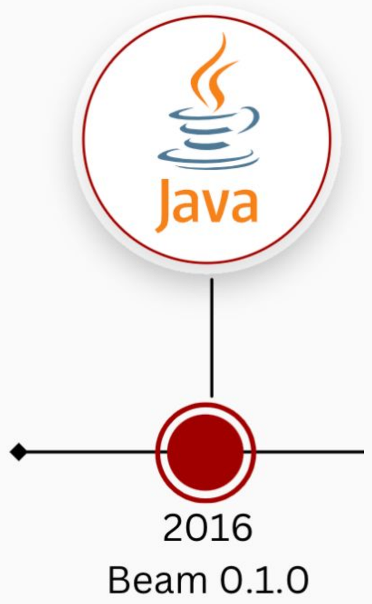
BEAM SUMMIT

Outline

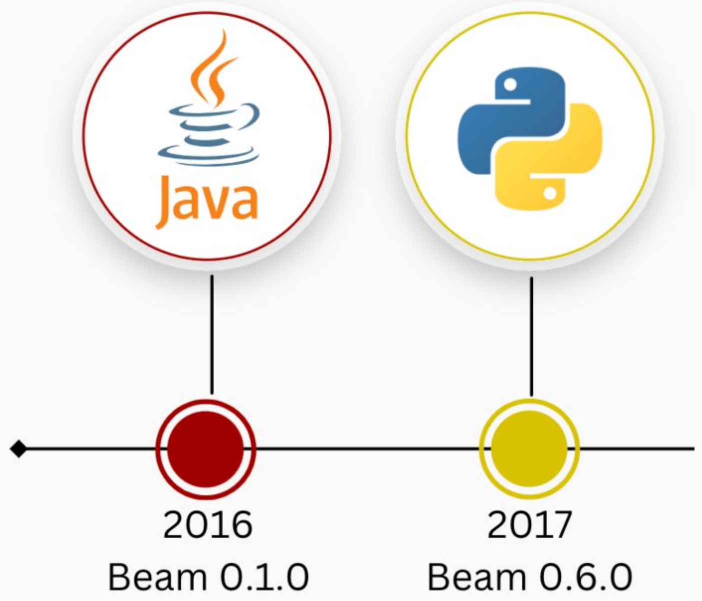
- Motivation for multi-language pipelines
- Definitions and refresher on the Beam model
- **Creating a portable transform** using the SchemaTransform framework
- **Creating an expansion service** that holds our portable transform
- **Using the portable transform** in a foreign SDK
- Future steps

Motivation for multi-language pipelines

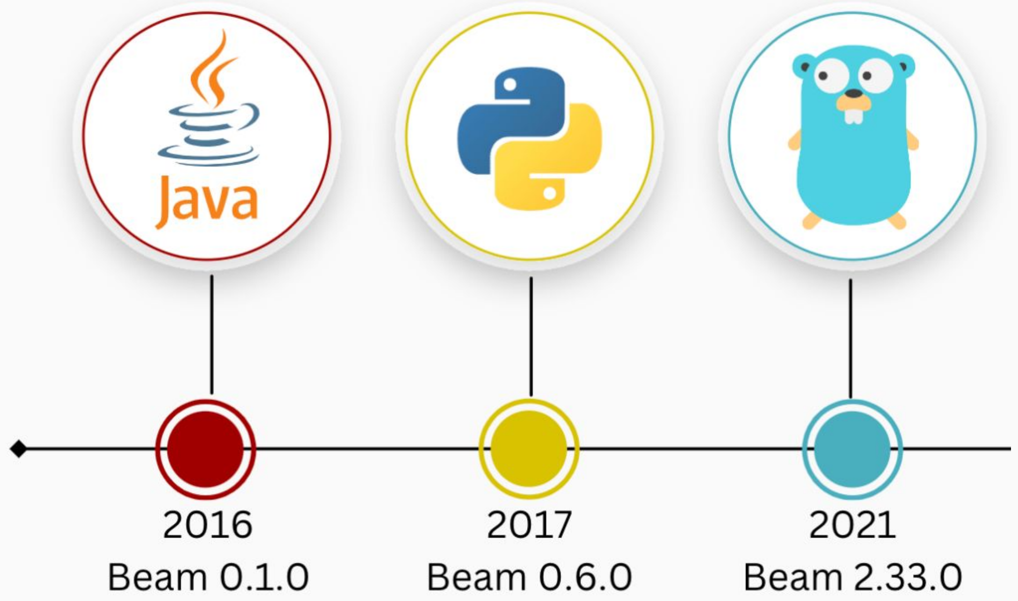
Motivation for multi-language pipelines



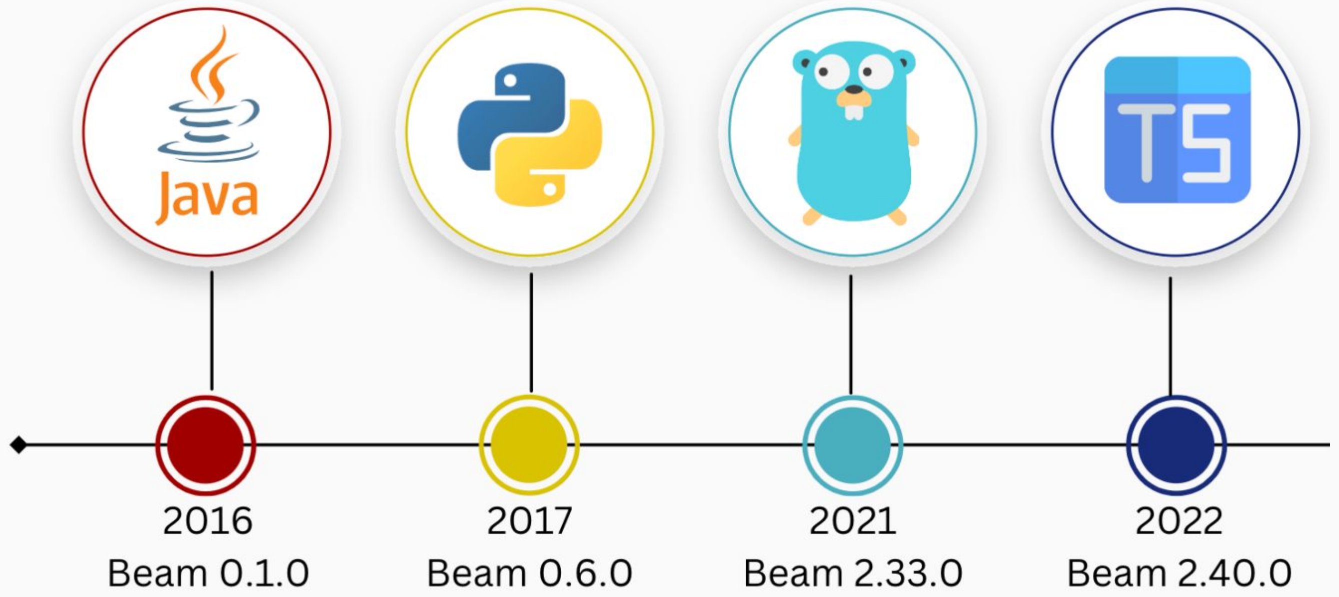
Motivation for multi-language pipelines



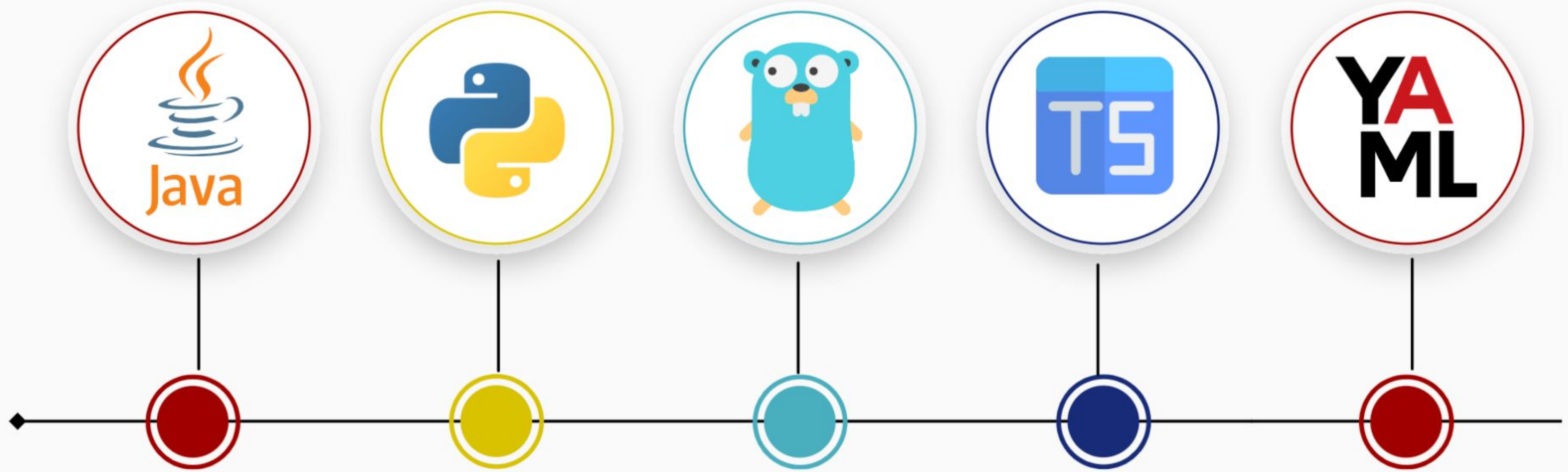
Motivation for multi-language pipelines



Motivation for multi-language pipelines



Motivation for multi-language pipelines



2016
Beam 0.1.0

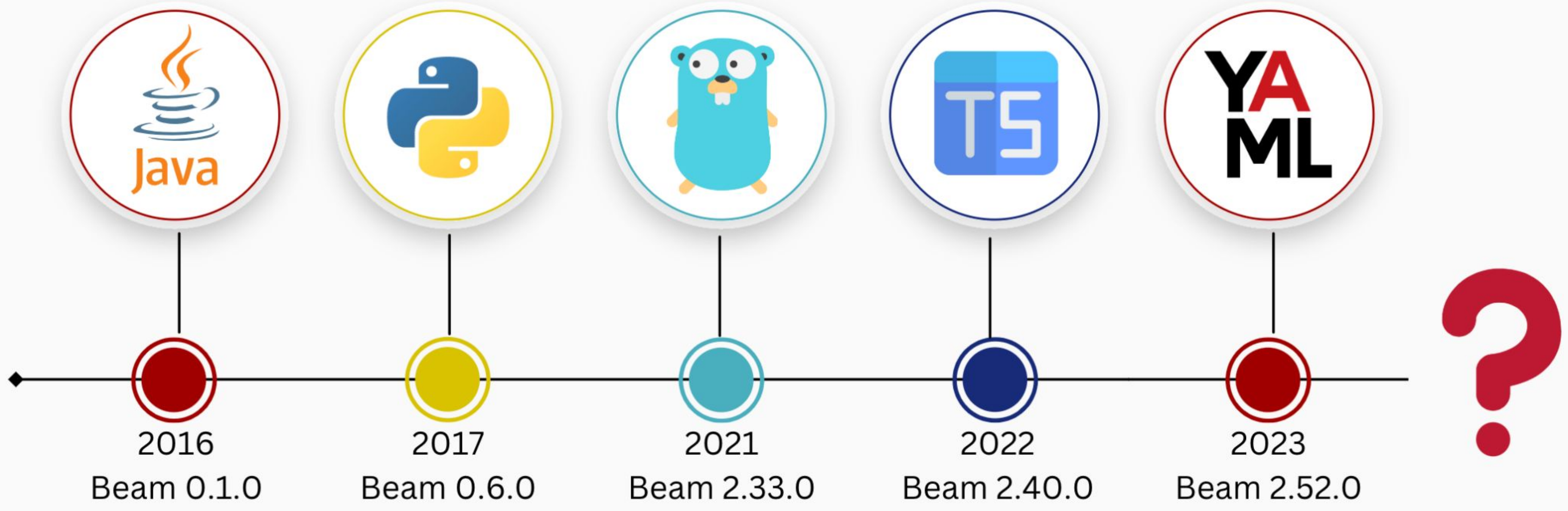
2017
Beam 0.6.0

2021
Beam 2.33.0

2022
Beam 2.40.0

2023
Beam 2.52.0

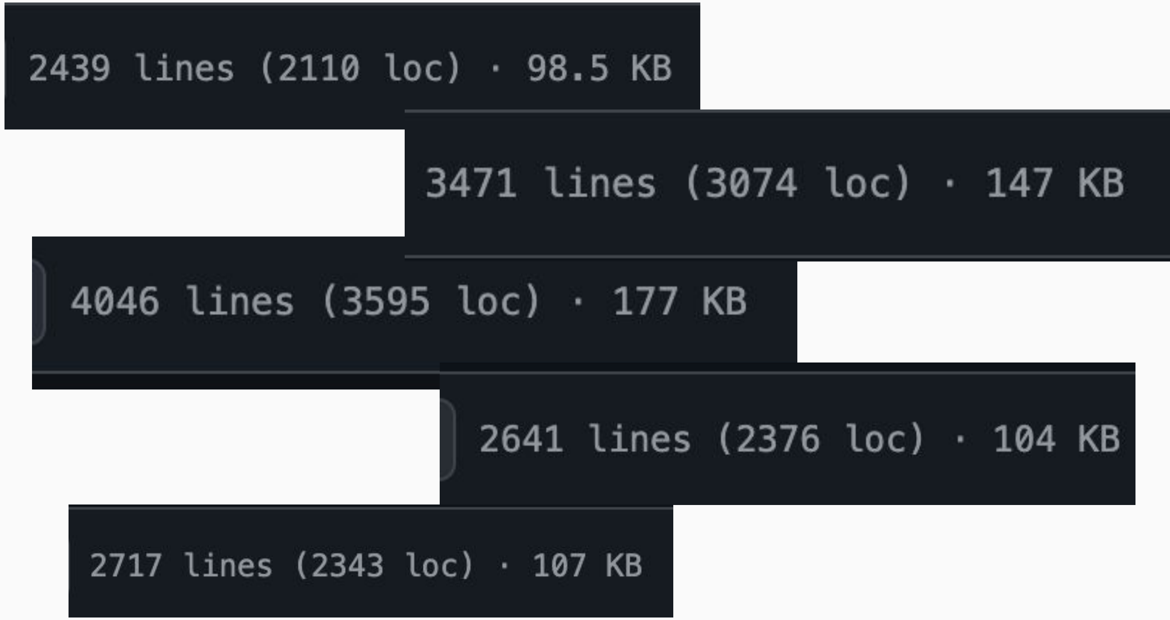
Motivation for multi-language pipelines



Motivation for multi-language pipelines

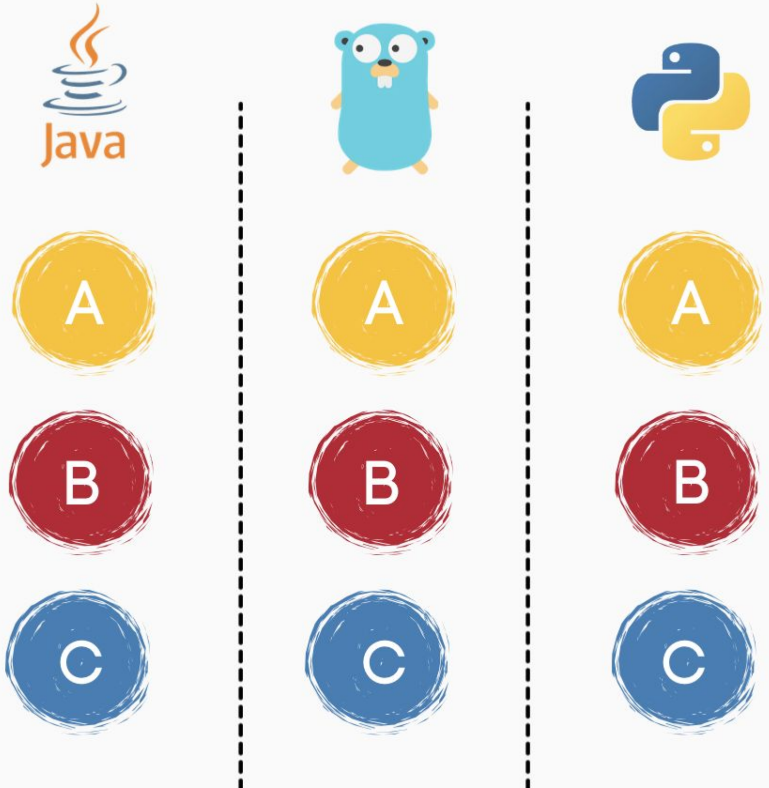
Each transform needs

- Robust functionality
- Resilient retry logic
- Edge case handling
- Clear documentation
- IO client integration
- …

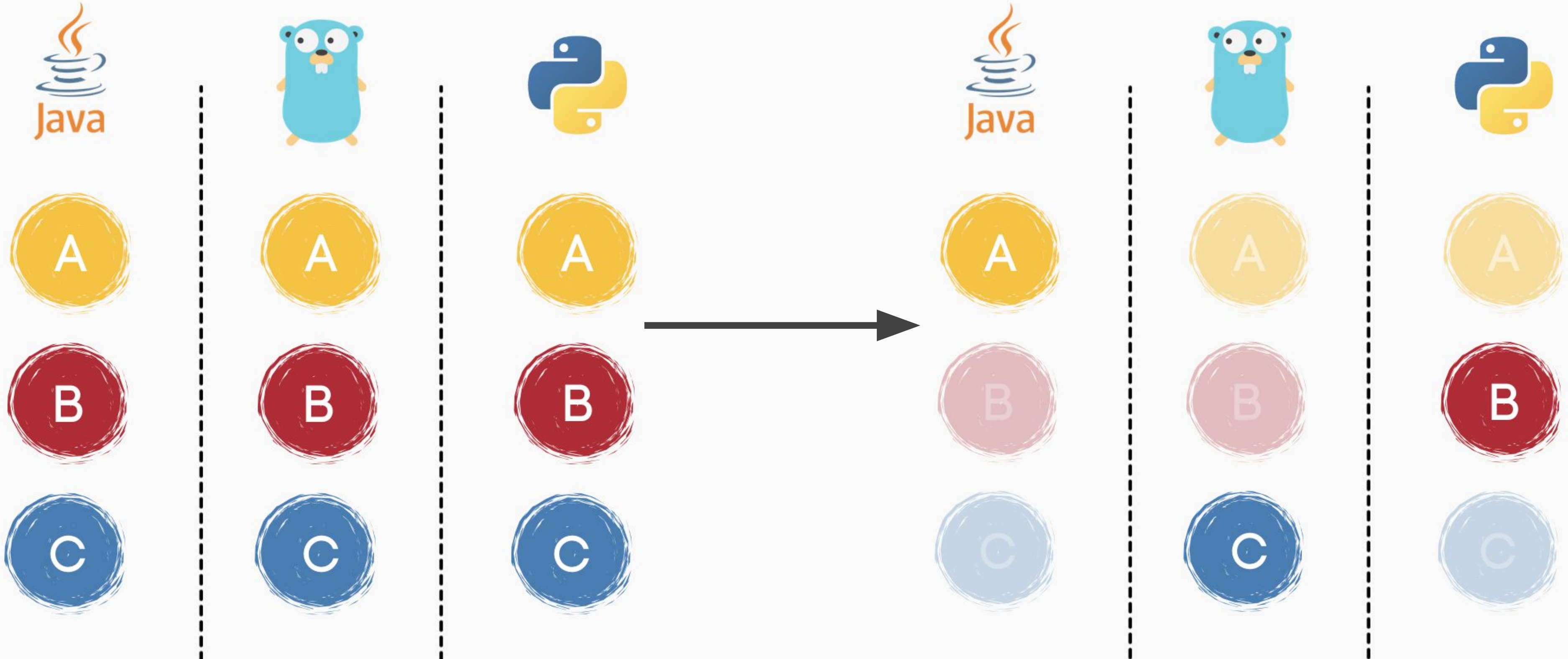


```
2439 lines (2110 loc) · 98.5 KB
3471 lines (3074 loc) · 147 KB
4046 lines (3595 loc) · 177 KB
2641 lines (2376 loc) · 104 KB
2717 lines (2343 loc) · 107 KB
```

Motivation for multi-language pipelines



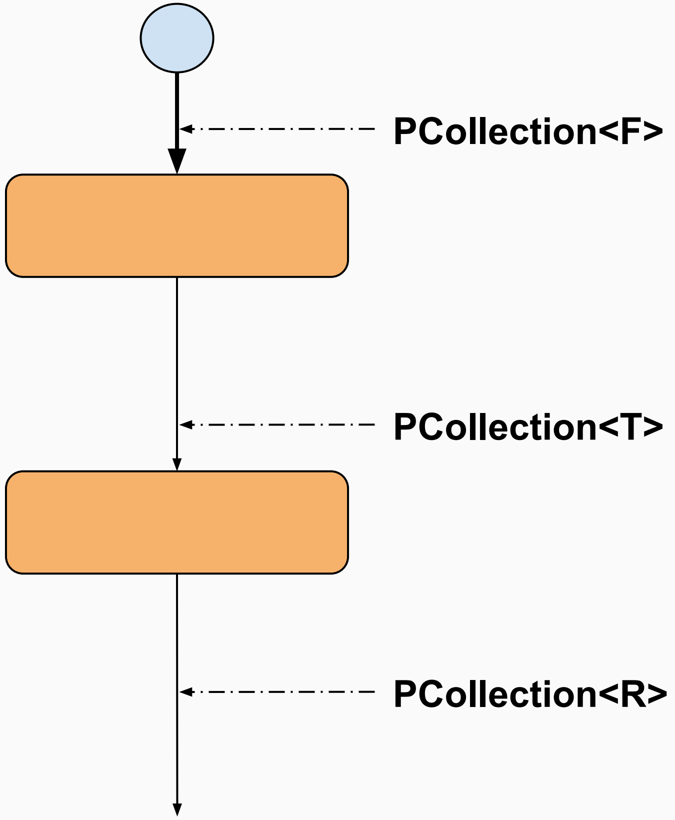
Motivation for multi-language pipelines



Quick Refresher on the Beam model

PCollections

- “Parallel Collection”
- Distributed collection of data
- Modes
  - Bounded (batch)
  - Unbounded (streaming)
- Is the input and output for each step in your pipeline
- PCollections contain elements of a particular type

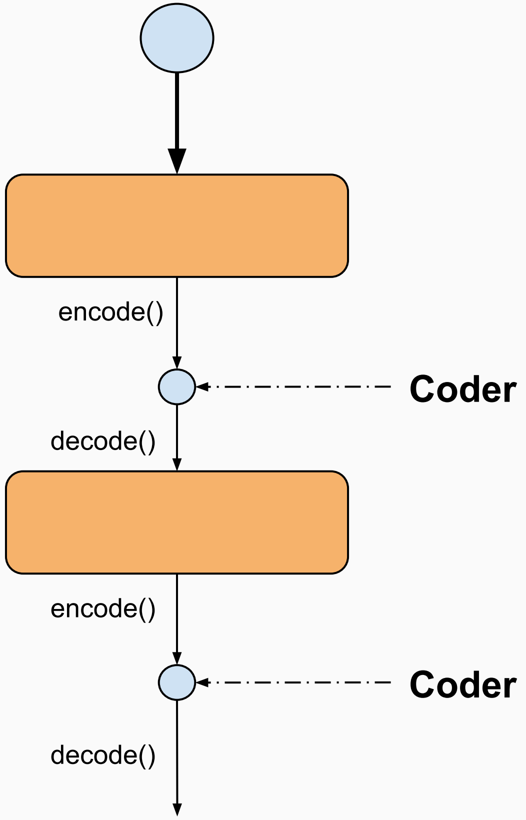


Coders

```
Coder<T> {
  byte[] encode(T obj);

  T decode(byte[] payload);
}
```

As a distributed data processing framework, Beam needs to **serialize objects to pass bytes over the wire**



Schemas

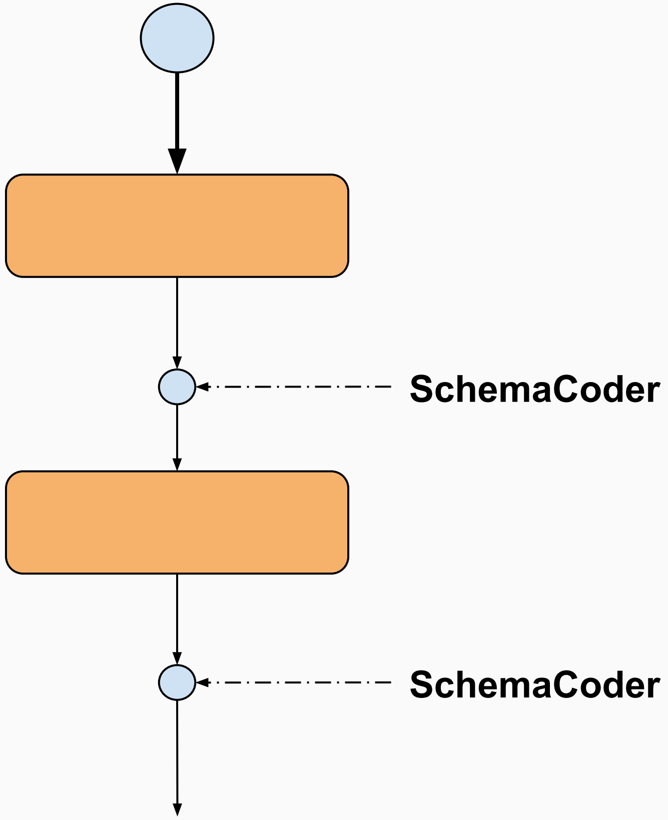
- Beam's native and **language-agnostic type system**
- PCollections with structured data can define a Schema
- Extends Beam with knowledge of the data's structure

- Schemas are useful for many things: https://www.youtube.com/watch?v=aRIZXtQiCHw

BYTE
INT16
INT32
INT64
FLOAT
DOUBLE
STRING
BOOLEAN
BYTES

ARRAY<T>
MAP<K, V>
STRUCT<...>

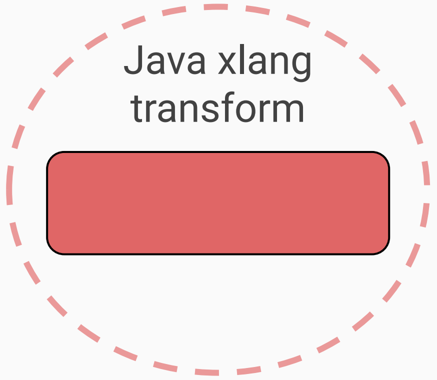
NULLABLE



Cross-language transform

- Is a portable transform
- Must be constructible using language-agnostic parameters
- Input/output PCollection element types must be language-agnostic

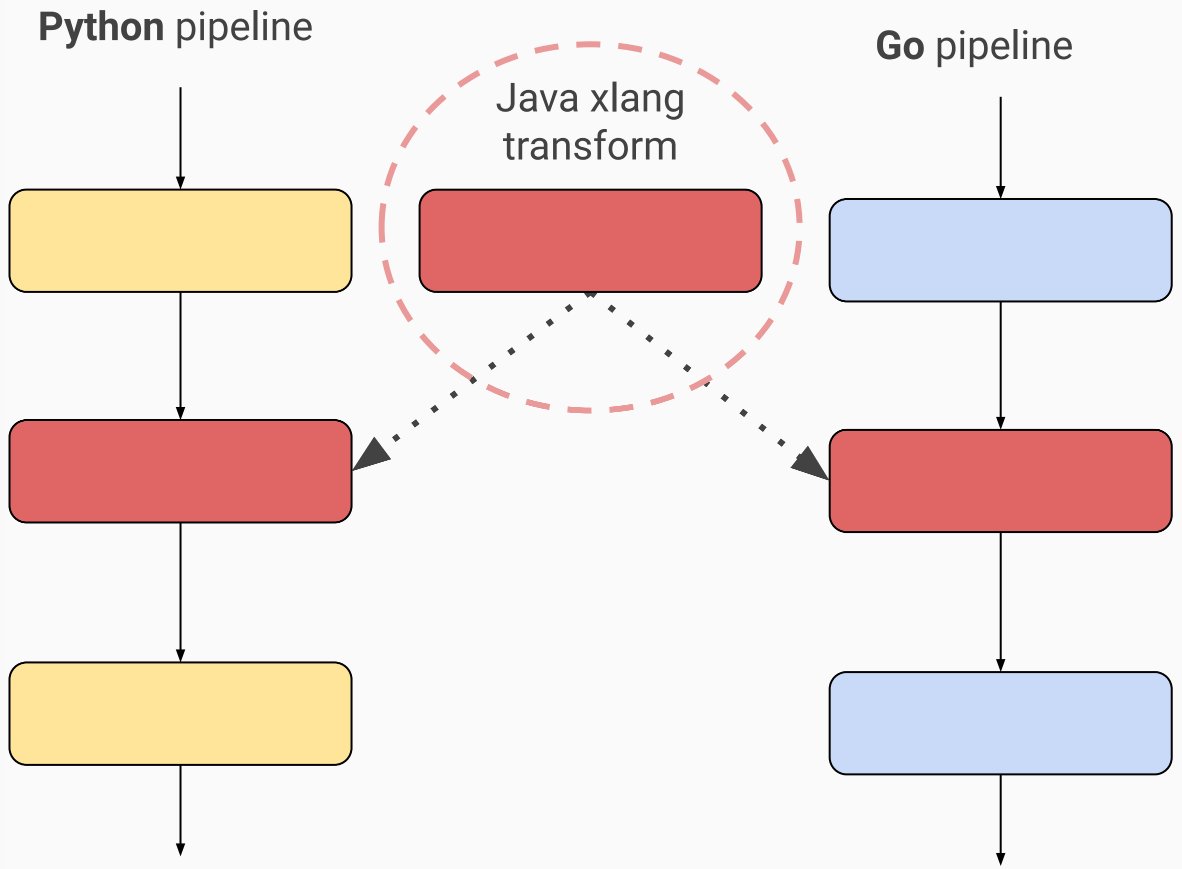
- Can be used by “foreign” SDKs via an **expansion service**:
  - provides and expands transforms



Cross-language transform

- Is a portable transform
- Must be constructible using language-agnostic parameters
- Input/output PCollection element types must be language-agnostic

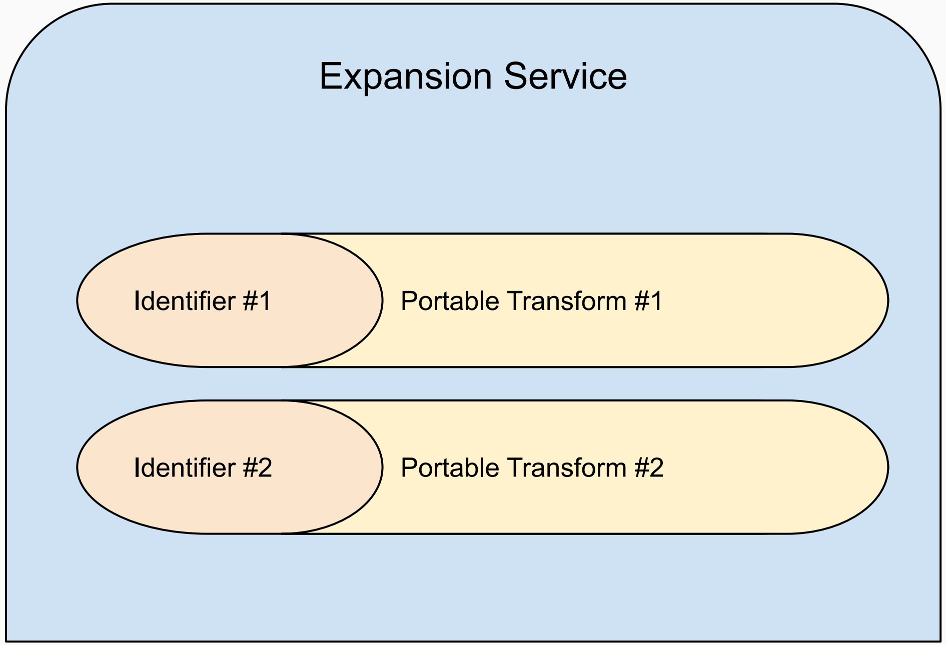
- Can be used by “foreign” SDKs via an **expansion service**



BEAM SUMMIT

Expansion Service

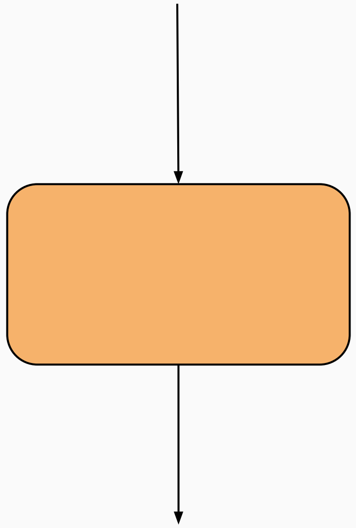
- A gRPC service
- Container that holds a list of portable transforms
- We can request a transform by its unique identifier
- Expands and provides the requested transform, ready to be applied to your pipeline



Creating a Portable Transform (Java)

The SchemaTransform framework

- Transforms are constructed using a **Beam Row**
  - language-agnostic configuration object
- Takes and produces Schema'd PCollections of **Beam Rows**
  - language-agnostic data types



Step 1) Design a configuration

```
Schema:
  STRING foo
  INT32 bar
```

Step 1) Design a configuration

```
Schema.builder()
    .addStringField("foo")
    .addInt32Field("bar")
    .build();
```

```
Schema:
  STRING foo
  INT32 bar
```

Step 1) Design a configuration

```
@DefaultSchema(AutoValueSchema.class)
@AutoValue
abstract class MyConfiguration {
  static Builder builder() {
    return new AutoValue_MyConfiguration.Builder();
  }
  abstract String getFoo();

  abstract Integer getBar();

  @AutoValue.Builder
  abstract static class Builder {
    abstract Builder setFoo(String foo);

    abstract Builder setBar(Integer bar);

    abstract MyConfiguration build();
  }
}
```

```
Schema:
  STRING foo
  INT32 bar
```

Step 1) Design a configuration

```
# Python's POV
with beam.Pipeline() as p:
  (p
   | Create([...])
   | MySchemaTransform(foo="abc", bar=123))
```

```
Schema:
  STRING foo
  INT32 bar
```

Step 1) Design a configuration

```
# YAML's POV
pipeline:
  transforms:
    - type: Create
      ...
    - type: MySchemaTransform
      config:
        foo: "abc"
        bar: 123
```

```
Schema:
  STRING foo
  INT32 bar
```

Step 2) Implement a SchemaTransformProvider

```
SchemaTransformProvider {
  String identifier();

  SchemaTransform from(Row configuration);

  Schema configurationSchema();
}
```

Step 2) Implement a SchemaTransformProvider

```
SchemaTransformProvider {
  String identifier();

  SchemaTransform from(Row configuration);

  Schema configurationSchema();
}
```

```
TypedSchemaTransformProvider<T> {
  String identifier();

  SchemaTransform from(T configuration);
}
```

2) Implement a SchemaTransformProvider

Example

```java
@AutoService(SchemaTransformProvider.class)
public class MyProvider
      extends TypedSchemaTransformProvider<MyConfiguration> {
  @Override
  public String identifier() {
    return "beam:schematransform:org.apache.beam:my_transform:v1";
  }

  @Override
  protected SchemaTransform from(MyConfiguration configuration) {
    return new MySchemaTransform(configuration);
  }

  static class MySchemaTransform extends SchemaTransform {
    MySchemaTransform(MyConfiguration configuration) {...}

    @Override
    public PCollectionRowTuple expand(PCollectionRowTuple input) {
        PCollection<Row> inputRows = input.get("input");
        PCollection<Row> outputRows = inputRows.apply(
                new SomeTransformIO(config.getFoo(), config.getBar()));

        return PCollectionRowTuple.of("output", outputRows);
    }
  }
}
```

Creating an expansion service that holds our portable transform

Shaded jar with ExpansionService and your portable transform

```
plugins {
    id 'com.github.johnrengelman.shadow' version '8.1.1'
    id 'application'
}

mainClassName = "org.apache.beam.sdk.expansion.service.ExpansionService"

dependencies {
    ...
    runtimeOnly 'org.apache.beam:beam-sdks-java-expansion-service:2.59.0'

    compileOnly "com.google.auto.service:auto-service-annotations:1.0.1"
    annotationProcessor "com.google.auto.service:auto-service:1.0.1"
    annotationProcessor "com.google.auto.value:auto-value:1.9"
}
```

Execute the shaded jar with a port

```
$ java -jar path/to/my-expansion-service.jar 12345

Starting expansion service at localhost:12345

Registered SchemaTransformProviders:
        beam:schematransform:org.apache.beam:my_transform:v1
```

Using the portable transform in a foreign SDK (Python)

Connect to an expansion service

```
from apache_beam.transforms.external_transform_provider import ExternalTransformProvider

# connect to an already running service
provider = ExternalTransformProvider("localhost:12345")
```

Connect to an expansion service

```python
from apache_beam.transforms.external import JavaJarExpansionService
from apache_beam.transforms.external_transform_provider import ExternalTransformProvider

# connect to an already running service
provider = ExternalTransformProvider("localhost:12345")
# start a service based on a Java jar
provider = ExternalTransformProvider(JavaJarExpansionService("path/to/my-expansion-service.jar"))
```

Connect to an expansion service

```
from apache_beam.transforms.external import JavaJarExpansionService
from apache_beam.transforms.external_transform_provider import ExternalTransformProvider

# connect to an already running service
provider = ExternalTransformProvider("localhost:12345")
# start a service based on a Java jar
provider = ExternalTransformProvider(JavaJarExpansionService("path/to/my-expansion-service.jar"))

provider = ExternalTransformProvider([
    "localhost:12345",
    JavaJarExpansionService("path/to/my-expansion-service.jar"),
    JavaJarExpansionService("path/to/another-expansion-service.jar")])
```

Retrieve and use the transform

```
transform_urn = "beam:schematransform:org.apache.beam:my_transform:v1"
MyTransform = provider.get_urn(transform_urn)

with beam.Pipeline() as p:
  (p
   | beam.Create(...)
   | MyTransform(foo="abc", bar=123)
   | beam.ParDo(...))
```

Generated metadata (> 2.60.0)

```
transform_urn = "beam:schematransform:org.apache.beam:my_transform:v1"
MyTransform = provider.get_urn(transform_urn)

import inspect

inspect.getdoc(MyTransform)
# Output: "MyTransform does this and that..."

inspect.signature(MyTransform)
# Output: (foo: ‘str: use foo like this...',
           bar: ‘int: use bar like that...')
```

Resources

Example:
https://github.com/apache/beam/tree/master/examples/multi-language#using-java-transforms-from-python

Quickstart guide with more details coming out soon…

Future steps…

- Improve experience going the other way around (Python transform in Java, e.g. RunInference)
- Enable and improve multi-lang support for the Go SDK

BEAM SUMMIT

Thank you!

Questions?

Ahmed Abualsaud

linkedin.com/in/ahmedabu98/

github.com/ahmedabu98/

ahmedabualsaud@apache.org