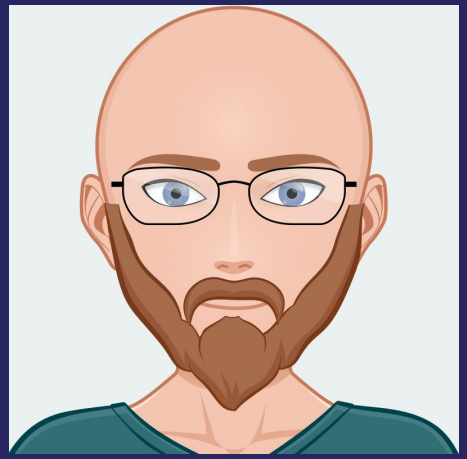
UNIVERSAL METRICS WITH BEAM

Etienne CHAUCHOT

Apache Beam Committer

@echauchot, echauchot@apache.org

Who am I?



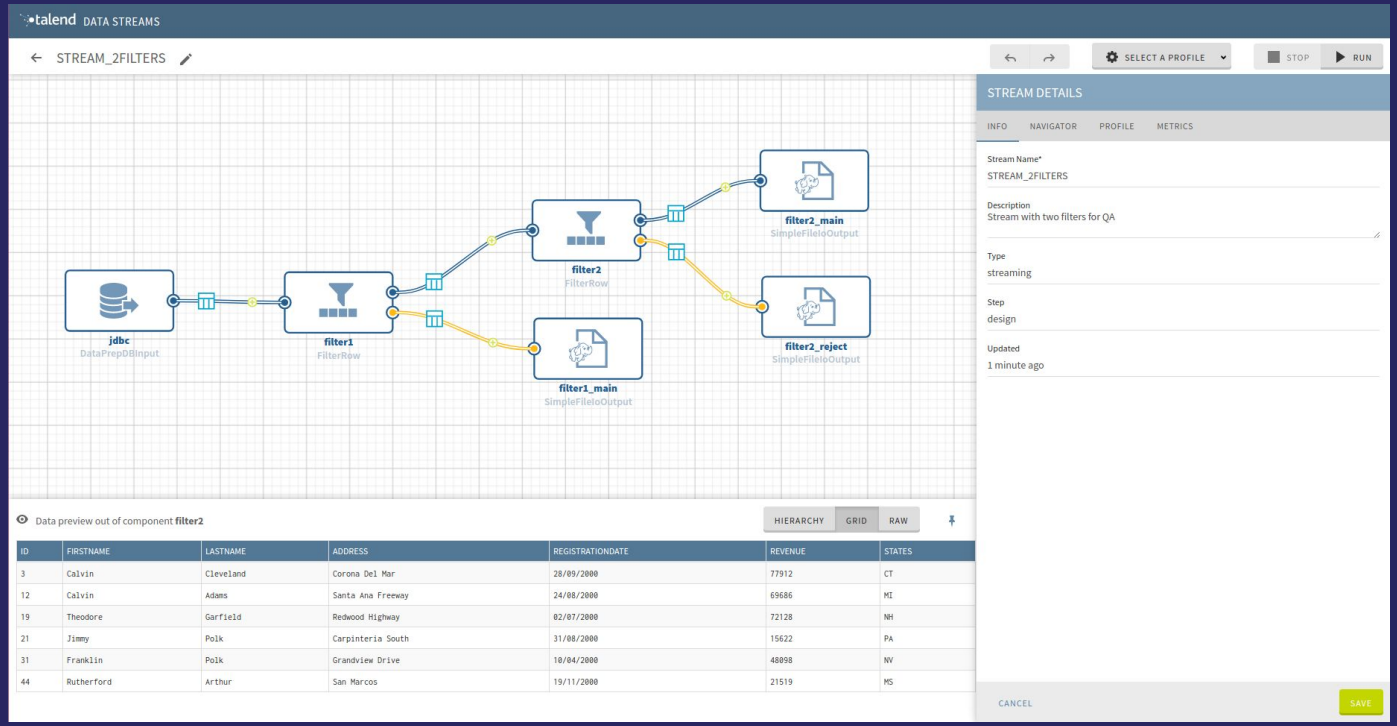
talend

@echauchot

Software Engineer
Apache Beam Committer

Integration Software
Big Data
Open Source

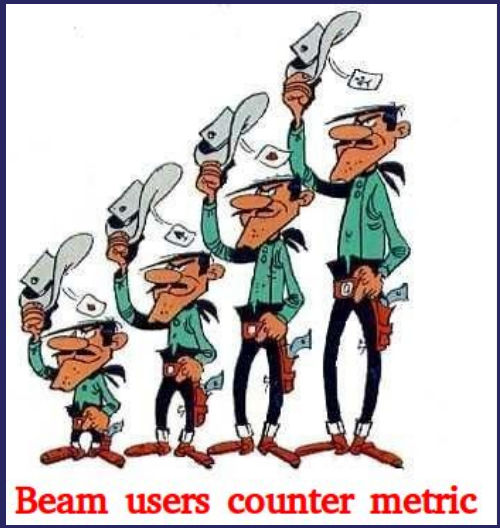
Talend: data integration software



Agenda

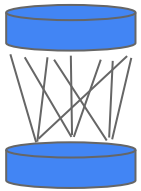
1. Introduction to Beam and portability
2. What are beam metrics ?
3. How does a pipeline author use them ?
4. Inside the runner: how does it work ?
5. Outside of the runner: How are they extracted ?
6. Between the runners: what are portable metrics ?
7. Example Use case
8. Current state and future work

1. Introduction to Beam and portability
2. What are beam metrics ?
3. How does a pipeline author use them (java) ?
4. Inside the runner: how does it work ?
5. Outside of the runner: How are they extracted ?
6. Between the runners : portability
7. Use case
8. Current state and future work



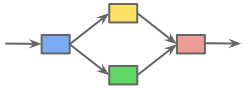
(2004) MapReduce

SELECT + GROUPBY



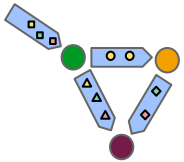
(2008) FlumeJava

High-level API



(2013) Millwheel

Deterministic streaming



(2014) Dataflow

Unified,
Multiple languages,
Multiple runners

(2016) Apache Beam

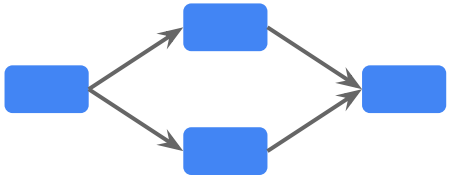
Open,
Community-driven,
Vendor-independent

Batch / streaming? Never heard of either.

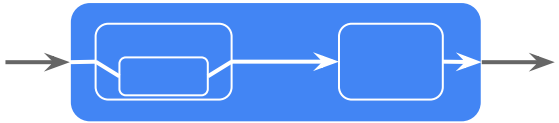
— Beam

(Batch is nearly always part of higher-level streaming)

User code



Libraries of PTransforms, IO

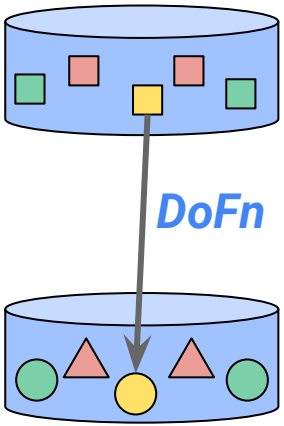


Language SDK

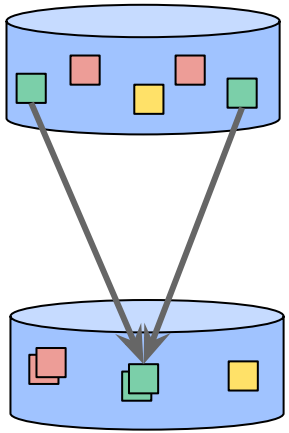
Beam Model representation

Runner

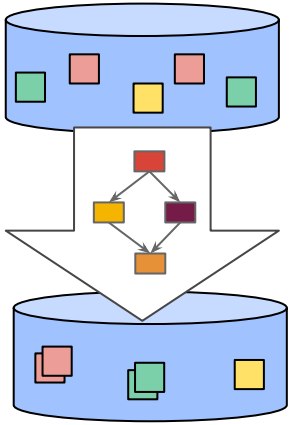
Beam PTransforms



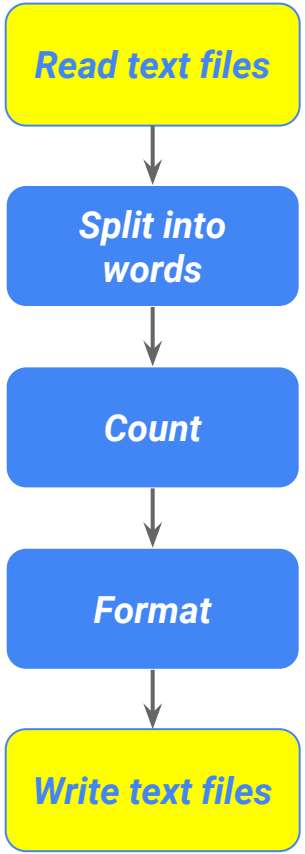
ParDo
(good old FlatMap)



GroupByKey

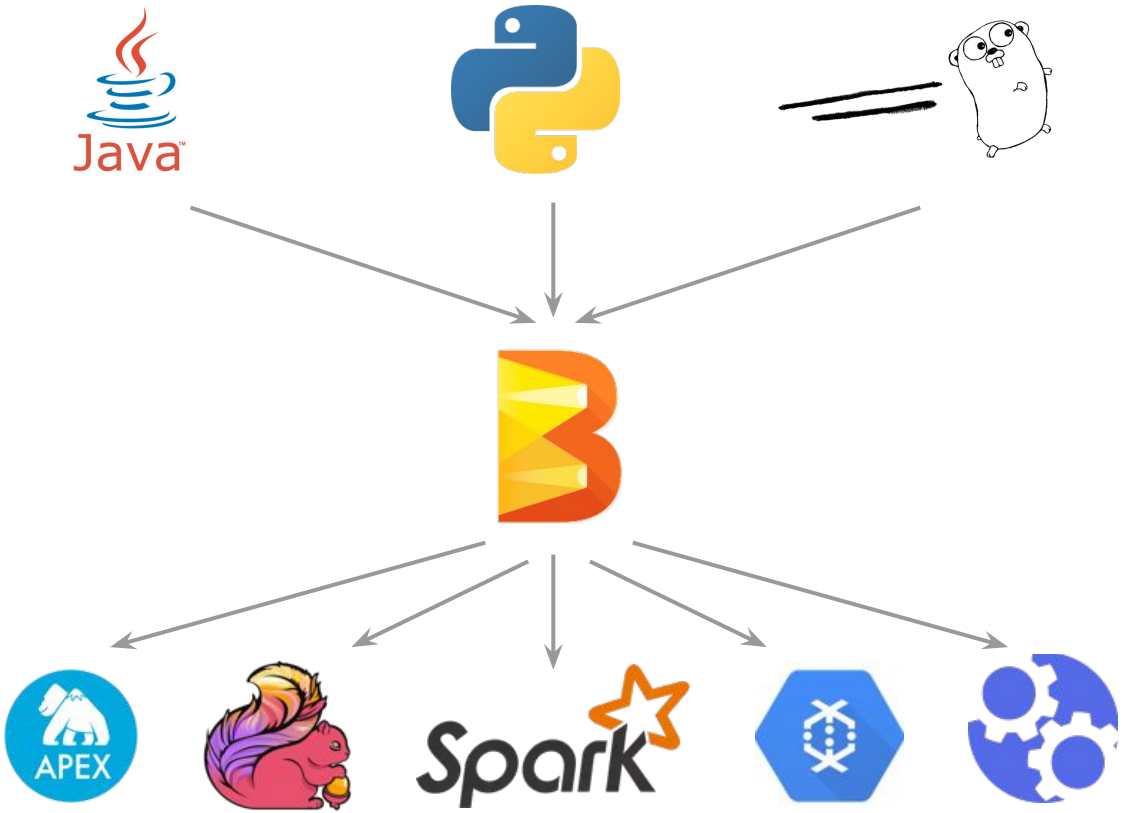


Composite

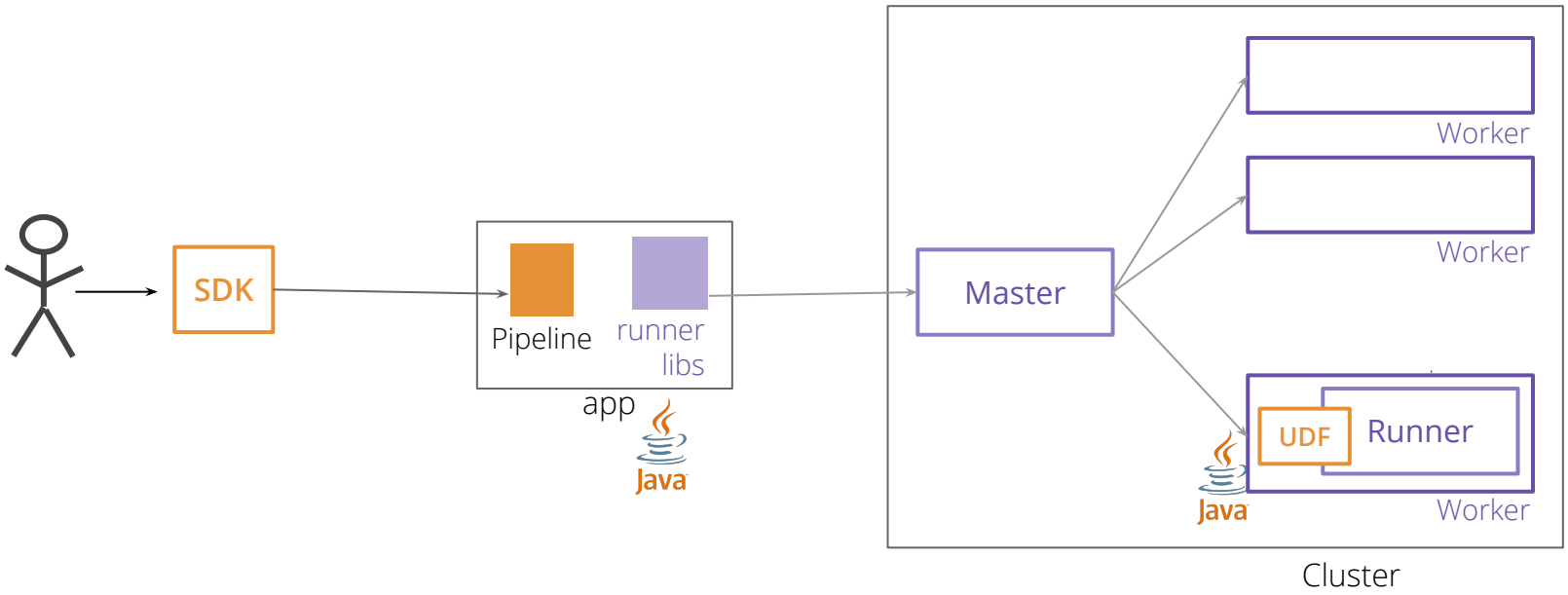



```
Pipeline p = Pipeline.create(options);

PCollection<String> lines = p.apply(

   TextIO.read().from("gs://.../*"));

PCollection<KV<String, Long>> wordCounts = lines

   .apply(FlatMapElements.via(word->word.split("\\W+")))

   .apply(Count.perElement());

wordCounts

   .apply(MapElements.via(count->count.getKey() + ": " + count.getValue())

      .apply(TextIO.write().to("gs://.../..."));

p.run();
```

Languages and runners are independent



How do Java-based runners work today?



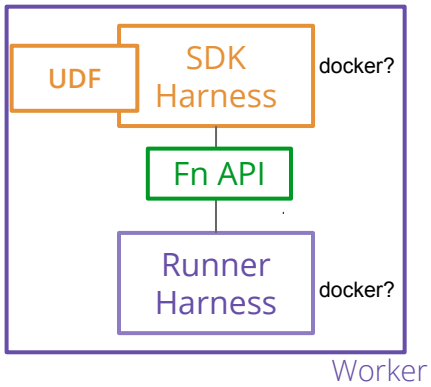
Goal of portability

Without portability framework

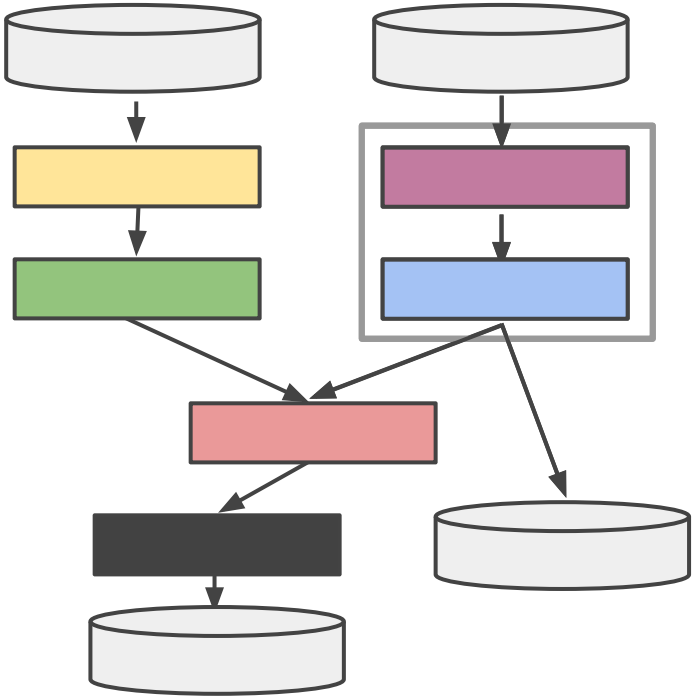
A pipeline written with a given language SDK must be run in a runner written with the same language (e.g. no Python to Java or Java to Python)

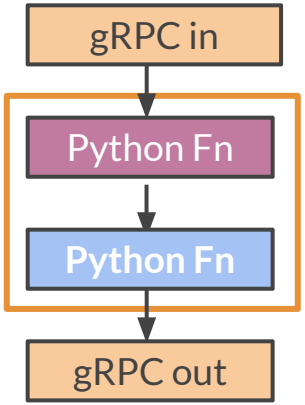
With portability framework

Execute user code from 'any' language in every runner.



Portability framework Design





1. Introduction to Beam and portability
2. What are beam metrics ?
3. How does a pipeline author use them (java) ?
4. Inside the runner: how does it work ?
5. Outside of the runner: How are they extracted ?
6. Between the runners : portability
7. Use case
8. Current state and future work

What are the Beam metrics ?

- The pipeline authors can specify metrics to be collected by the execution engine in any transform of their pipeline
- Declared at pipeline construction time
- 3 kind of user metrics (similar to dropwizard):
  - **Counter**: single long value that can be incremented or decremented
  - **Gauge**: reports the latest value out of reported values
  - **Distribution**: reports sum, count, min and max, mean values out of reported values (similar to histograms)

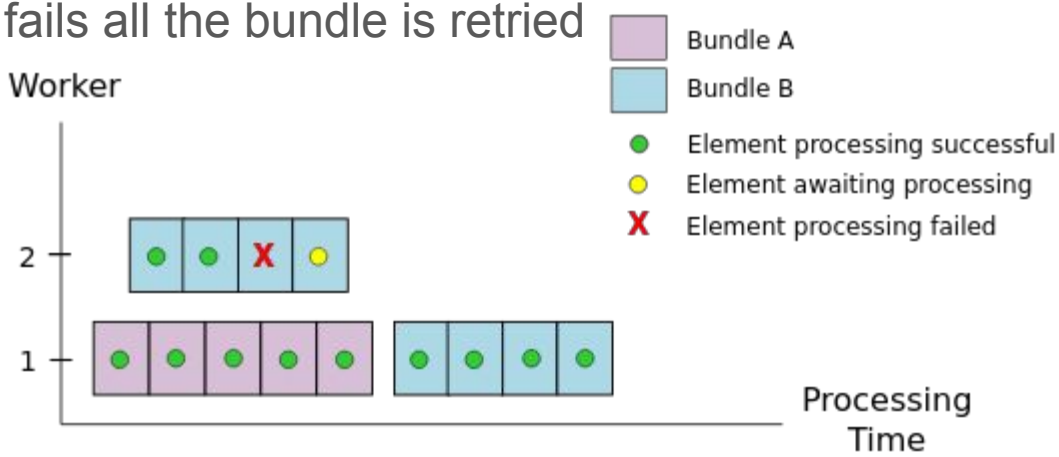
Beam metrics properties

- Accessible during execution of the pipeline or after it (depends on the runner)
- Metrics properties:
  - They are **named** : actual name + namespace (to differentiate and query by namespace)
  - They are **scoped** to the transform (pipeline step)
  - They are **dynamically created**: created during runtime (when your transform code runs), no pre-declaration in the runners/engines
  - They **degrade gracefully**: if a runner does not support it, it does not report it, no failure

Error handling: bundling

- The elements in a PCollection are processed in **bundles**
- bundles are the commit unit of the runner (persisting results, retry when failure) and allows parallelism between workers
- Division of the PCollection into bundles is an arbitrary choice by the runner (size might depend on runner streaming or batch orientation)
- If a transform fails all the bundle is retried

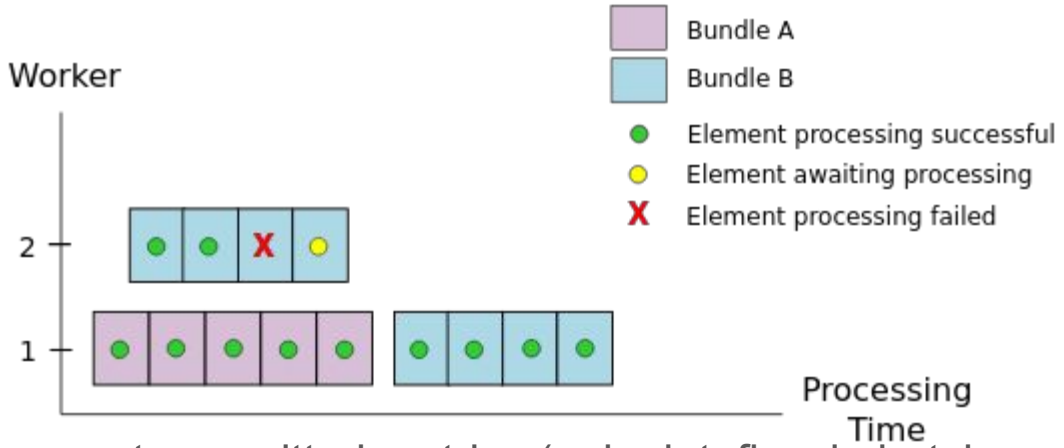
Pardo: the processing of an element within bundle B fails, and another worker retries the entire bundle



Error handling: committed/attempted metrics

- Attempted metrics = Metric includes values from all attempts
- Committed metrics = Metric includes values from successful attempts only
- If the parDo increments a counter with each element processed

committed counter = 9
attempted counter = 11



- Most of the runners do not support committed metrics (only dataflow in batch mode)

Querying metrics

- Through PipelineResult
- Supports filters:
  - names
  - step (transforms)

```
public interface PipelineResult {

 MetricResults metrics();
}
```

```
public abstract class MetricResults {

 public abstract MetricQueryResults queryMetrics(@Nullable MetricsFilter filter);
}
```

```
public interface MetricQueryResults {

 Iterable<MetricResult<Long>> getCounters();

 Iterable<MetricResult<DistributionResult>> getDistributions();

 Iterable<MetricResult<GaugeResult>> getGauges();
}
```

```
public interface MetricResult<T> {

 MetricName getName();

 String getStep();

 T getCommitted();

 T getAttempted();
}
```

1. Introduction to Beam and portability
2. What are beam metrics ?
3. How does a pipeline author use them (java) ?
4. Inside the runner: how does it work ?
5. Outside of the runner: How are they extracted ?
6. Between the runners : portability
7. Use case
8. Current state and future work

Usage: Pipeline and counter metric

```
pipeline
   .apply(...)
   .apply(ParDo.of(new CountingDoFn()));
pipelineResult = pipeline.run();
pipelineResult.waitUntilFinish(...);

MetricQueryResults metrics =
   pipelineResult
      .metrics()
      .queryMetrics(
         MetricsFilter.builder()
            .addNameFilter(MetricNameFilter.named(
                  "namespace", "counter1"))
            .build());
```

```
public class CountingDoFn extends DoFn<Integer, Integer> {

 private final Counter counter = Metrics.counter( "namespace",
"counter1");

 @ProcessElement
 public void processElement(ProcessContext context) {
    // count the elements
    counter.inc();
    context.output(context.element());
 }
}
```

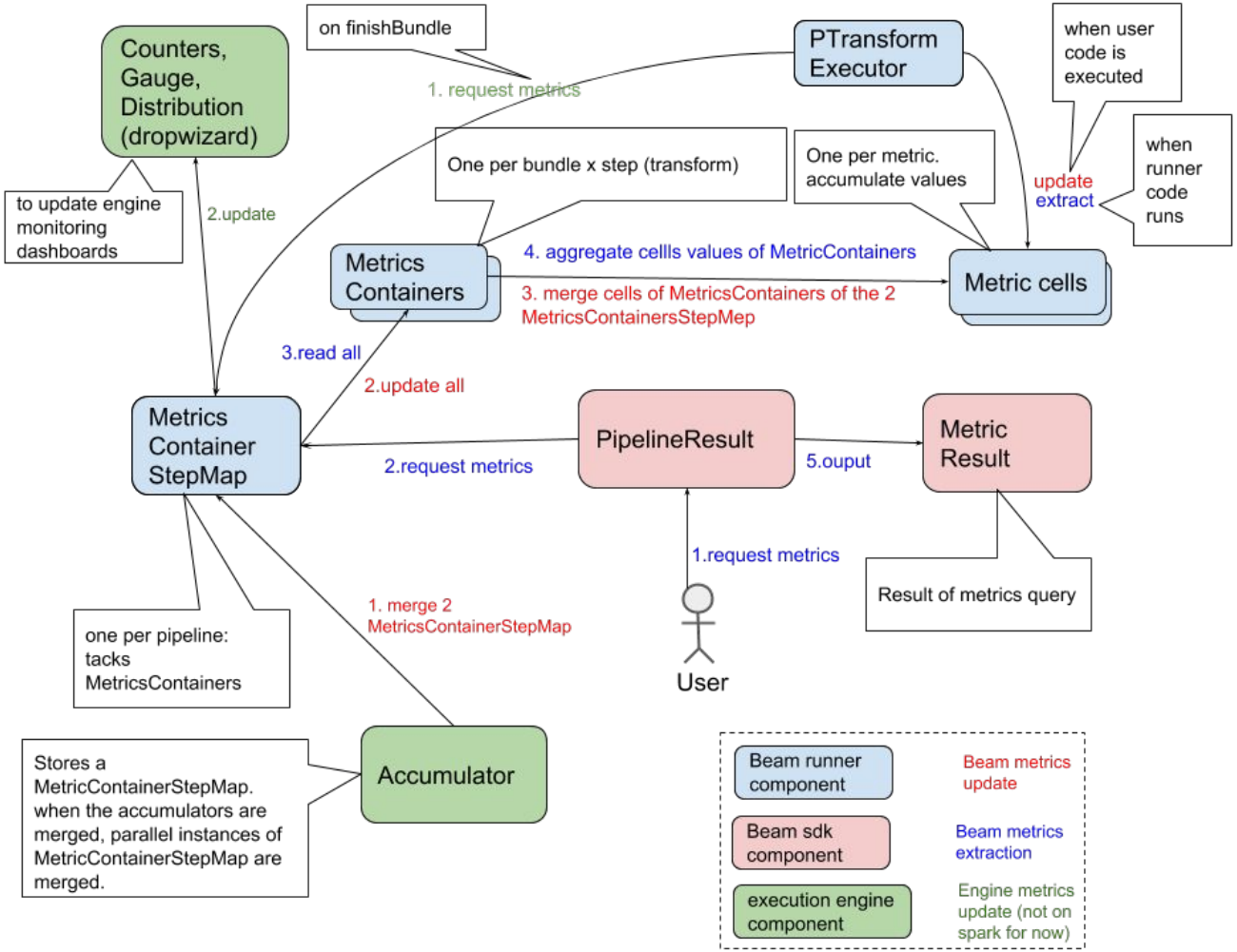
Usage: Distribution metric

```
public class CountingDoFn extends DoFn<Integer, Integer> {

 private final Distribution
 distribution = Metrics.distribution( "namespace", "distribution1");

 @ProcessElement
 public void processElement(ProcessContext context) {
    Integer element = context.element();
    // create a distribution (histogram) of the values
    distribution.update(element);
    context.output(context.element());
 }
}
```

Usage: Gauge metric

```
public class CountingDoFn extends DoFn<Integer, Integer> {

 private final Gauge
 gauge = Metrics.gauge( "namespace", "gauge1");

 @ProcessElement
 public void processElement(ProcessContext context) {
    Integer element = context.element();
    // create a gauge (latest value received) of the values
    gauge.set(element);
    context.output(context.element());
 }
}
```

1. Introduction to Beam and portability
2. What are beam metrics ?
3. How does a pipeline author use them (java)?
4. Inside the runner: how does it work ?
5. Outside of the runner: How are they extracted ?
6. Between the runners : portability
7. Use case
8. Current state and future work


Counters, Gauge, Distribution (dropwizard)
on finishBundle
1. request metrics
PTransform Executor
when user code is executed
when runner code runs
update extract
to update engine monitoring dashboards
2.update
One per bundle x step (transform)
One per metric. accumulate values
Metrics Containers
4. aggregate cellls values of MetricContainers
3. merge cells of MetricsContainers of the 2 MetricsContainersStepMep
Metric cells
3.read all
2.update all
Metrics Container StepMap
PipelineResult
2.request metrics
5.ouput
Metric Result
1.request metrics
Result of metrics query
1. merge 2 MetricsContainerStepMap
one per pipeline: tacks MetricsContainers
User
Accumulator
Stores a MetricContainerStepMap. when the accumulators are merged, parallel instances of MetricContainerStepMap are merged.
Beam runner component
Beam metrics update
Beam sdk component
Beam metrics extraction
execution engine component
Engine metrics update (not on spark for now)

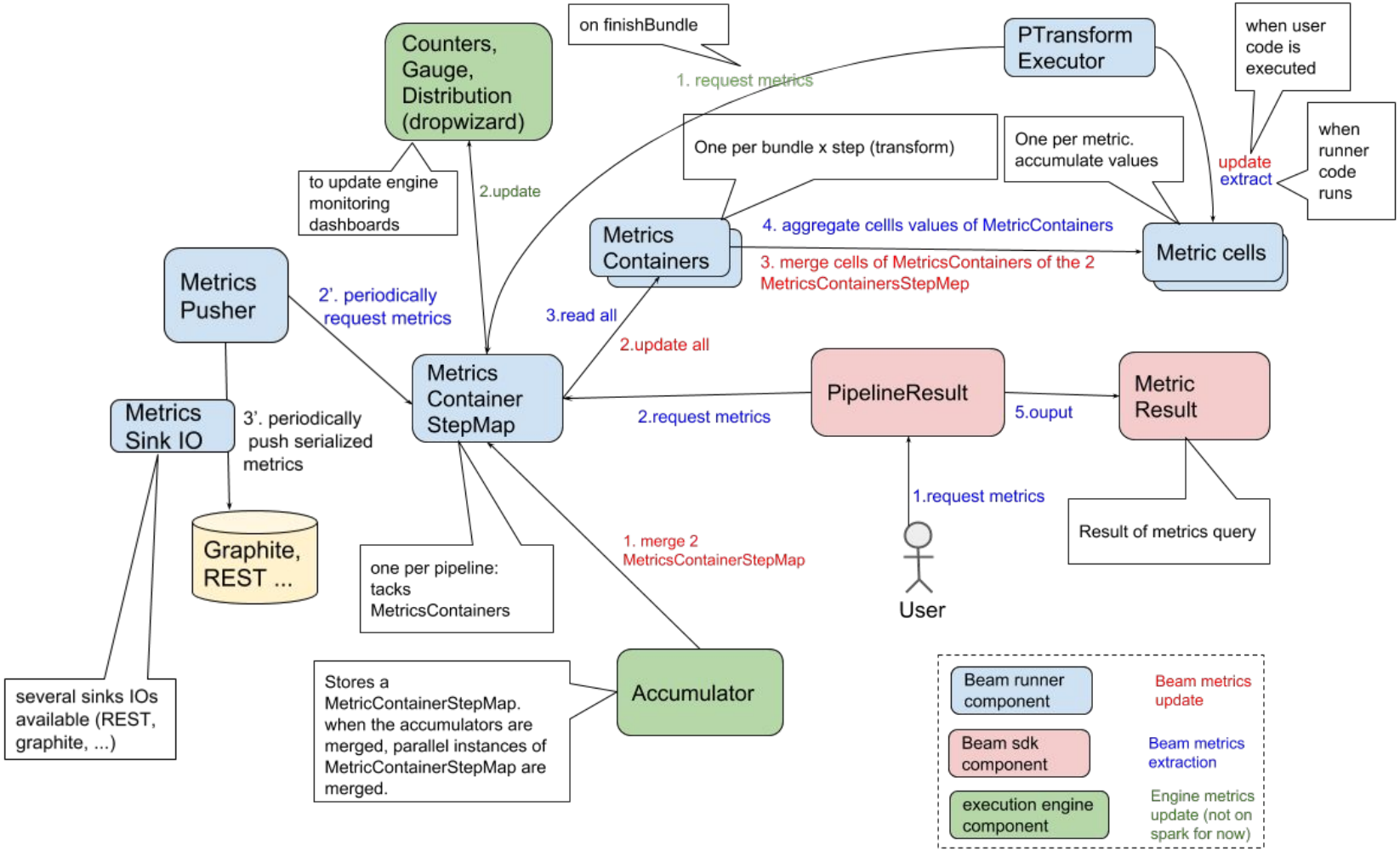
1. Introduction to Beam and portability
2. What are beam metrics ?
3. How does a pipeline author use them (java) ?
4. Inside the runner: how does it work ?
5. Outside of the runner: How are they extracted ?
6. Between the runners : portability
7. Use case
8. Current state and future work

Why metrics extraction should be independent from the chosen engine?

- Not all engines ship a way to push the metrics to external sinks (own monitoring)
- Consistency:
  - No common set of monitoring backend support among execution egines (Ganglia, graphite, ...)
  - Difference of availability moment when the pipeline runs
  - Beam needs to have a common metrics flow no matter the runners for pipelines to be portable
  - What if you need to have the metrics in your own application UI ? Request metrics backend (heterogeneous between the engines) ?

How metrics are extracted: design principles

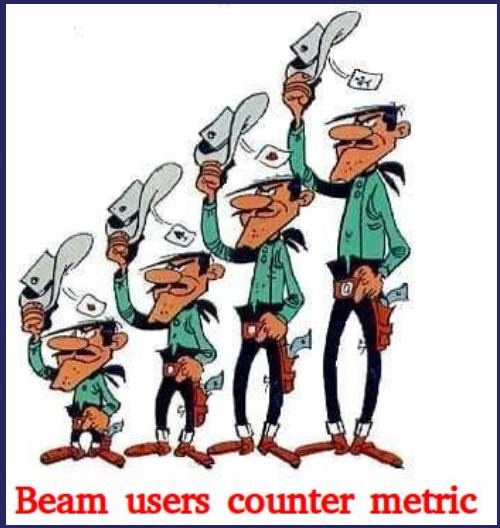
- no client polling (e.g. JMX)
  - infrastructure changes (cluster managers, ...) and must not need to be known of the users
  - would be non-aggregated metrics, that users would need to aggregate
  - timing issues (e.g. small batches)
- pushed from the runner and not pulled from the sdk
  - runners needs to decide when to push (e.g. committed metrics)
  - system runner metrics in the future
- periodic aggregation and not event based:
  - use aggregated metrics to avoid metrics backend need to merge
  - runners know how to merge metrics
- Beam managed sinks IOs
  - for coherence (avoid heterogeneous engines support)


Counters, Gauge, Distribution (dropwizard)
on finishBundle
1. request metrics
PTransform Executor
when user code is executed
when runner code runs
update extract
to update engine monitoring dashboards
2.update
One per bundle x step (transform)
One per metric. accumulate values
Metrics Containers
4. aggregate cellls values of MetricContainers
3. merge cells of MetricsContainers of the 2 MetricsContainersStepMep
Metric cells
Metrics Pusher
2'. periodically request metrics
3.read all
2.update all
Metrics Container StepMap
PipelineResult
5.ouput
Metric Result
Metrics Sink IO
3'. periodically push serialized metrics
2.request metrics
1.request metrics
Result of metrics query
Graphite, REST ...
one per pipeline: tacks MetricsContainers
1. merge 2 MetricsContainerStepMap
User
several sinks IOs available (REST, graphite, ...)
Stores a MetricContainerStepMap. when the accumulators are merged, parallel instances of MetricContainerStepMap are merged.
Accumulator
Beam runner component
Beam metrics update
Beam sdk component
Beam metrics extraction
execution engine component
Engine metrics update (not on spark for now)

How metrics are extracted: special case of cloud hosted engines

- runner = empty shell that translates the pipeline and submits the job and forwards user metric requests through REST calls. But it does not run it
- => the pusher needs to be instantiated in the cloud engine, close to the pipeline execution
- Solutions
  - When the job starts, the engine receives the serialized PipelineOptions and can trigger the creation of a MetricsPusher-like service
  - Pipeline pre-processing: DAG modification to include DoFn with native Metrics API calls
- Need to take care of possible network issues to push metrics to the outside world: open routes or compliant output sinks. (try to avoid pulling metrics from sdk cf before)

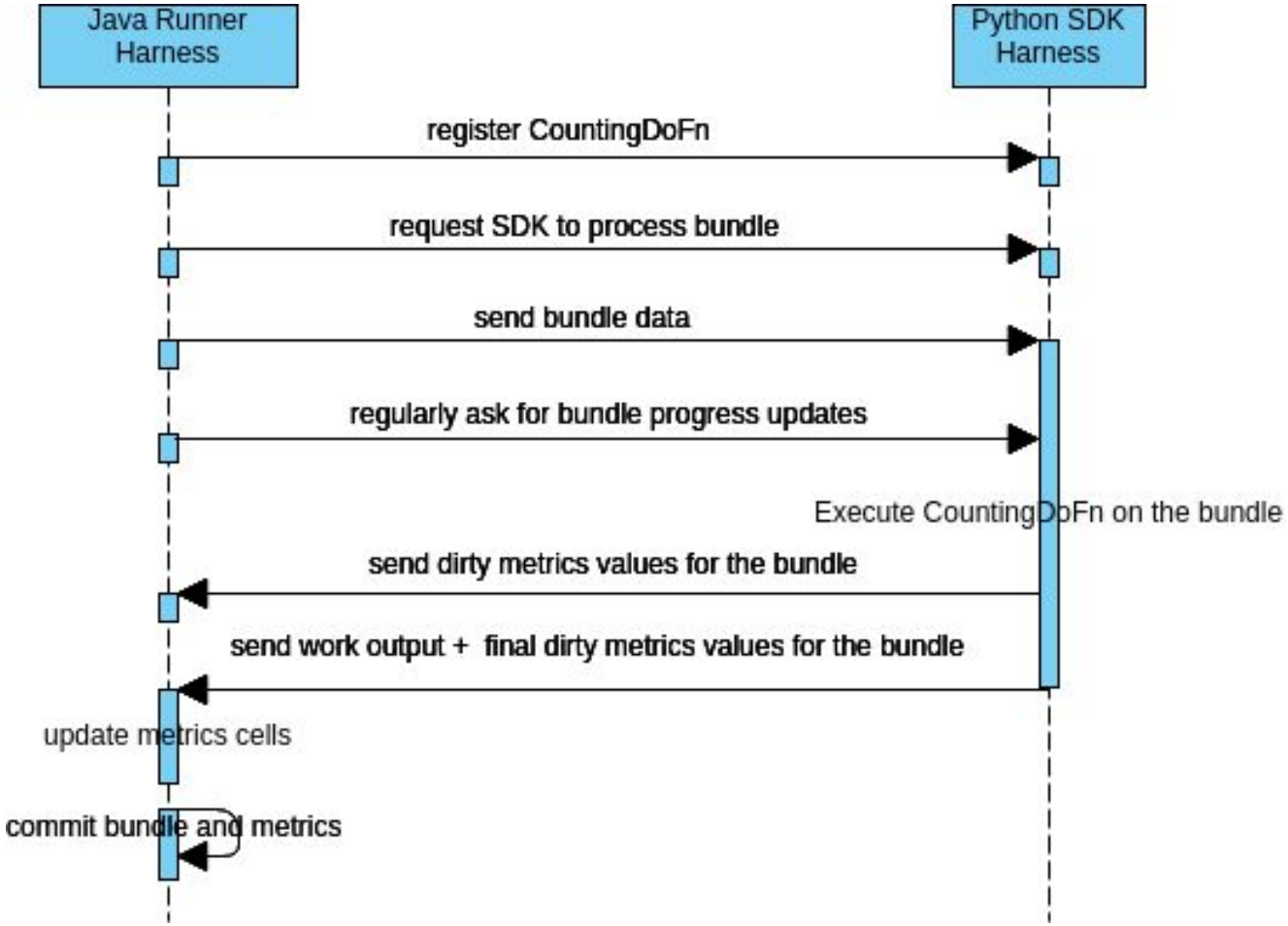
1. Introduction to Beam and portability
2. What are beam metrics ?
3. How does a pipeline author use them (java) ?
4. Inside the runner: how does it work ?
5. Outside of the runner: How are they extracted ?
6. Between the runners : portability
7. Use case
8. Current state and future work



Portable metrics (metrics over the Fn API)

- Metrics definition is part of the protobuf definition of the pipeline
- The SDK harness sends regular updates to the runner harness during the execution of user defined functions (if they contain metrics) through the FnAPI (GRPC calls).
- Main communication is on a bundle basis
- Runner still differentiate committed/attempted metrics
- Still very early stage (design of the communications and first step in python SDK)

Metrics communication over the FnAPI

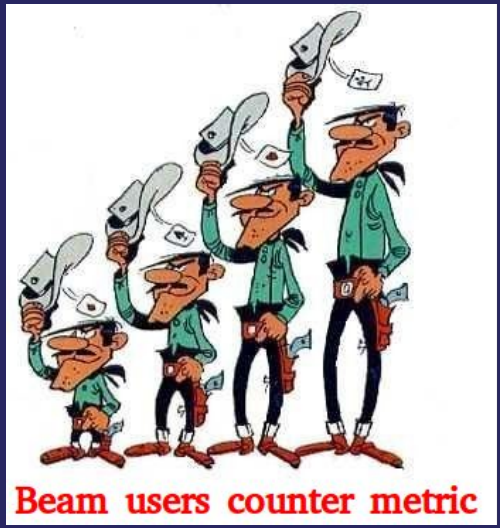


Portable metrics: new features

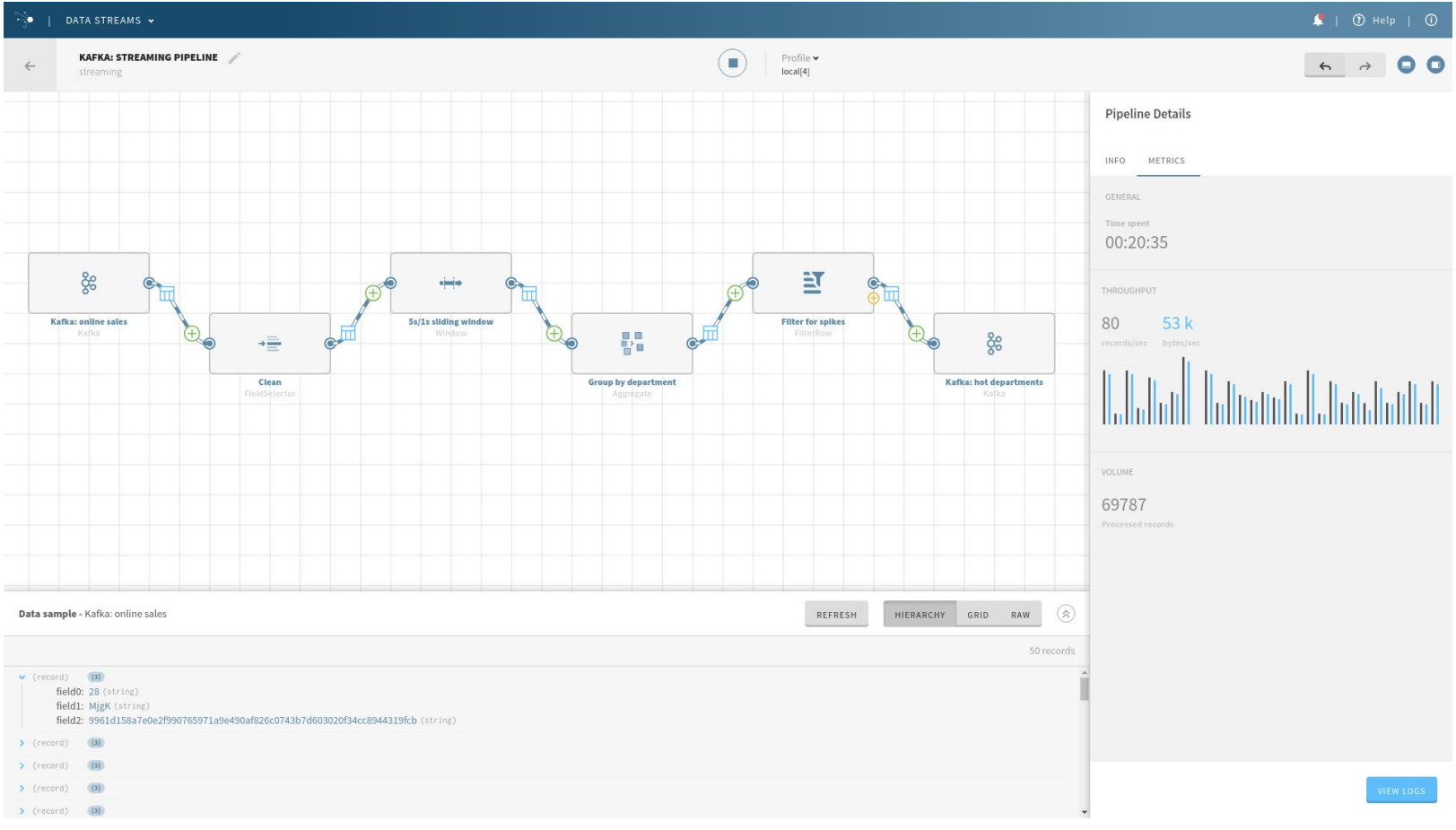
- Metrics are extended with custom ones (defined by the user)
- 2 scopes:
  - user metrics
  - system metrics
- 2 kinds of metrics:
  - Regular metrics (counter, distribution, gauge)
  - MonitoredState (customizable more complex data). Might not be supported by some metrics backends

file name	last modification	Current size
todo_list	09/26/18	1GB (too big)

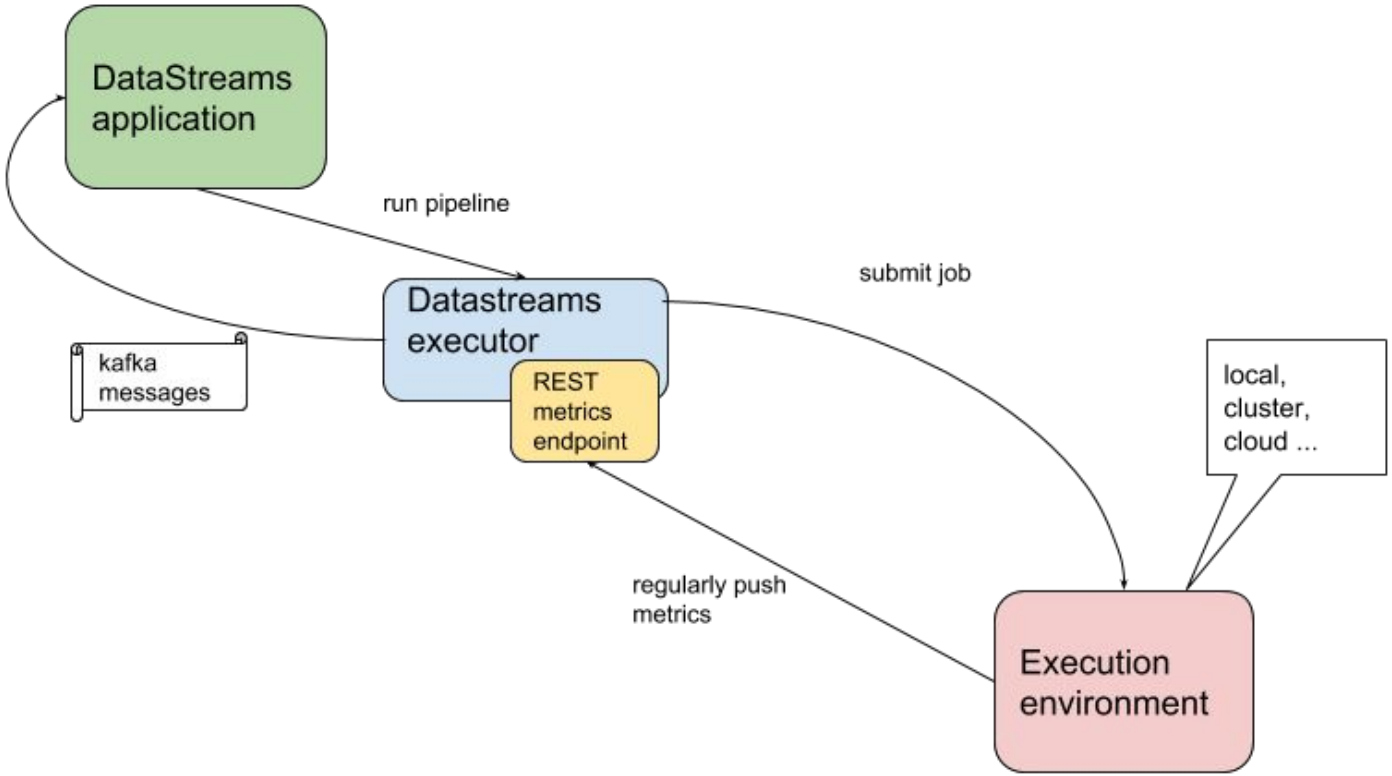
1. Introduction to Beam and portability
2. What are beam metrics ?
3. How does a pipeline author use them (java) ?
4. Inside the runner: how does it work ?
5. Outside of the runner: How are they extracted ?
6. Between the runners : portability
7. Example use case
8. Current state and future work



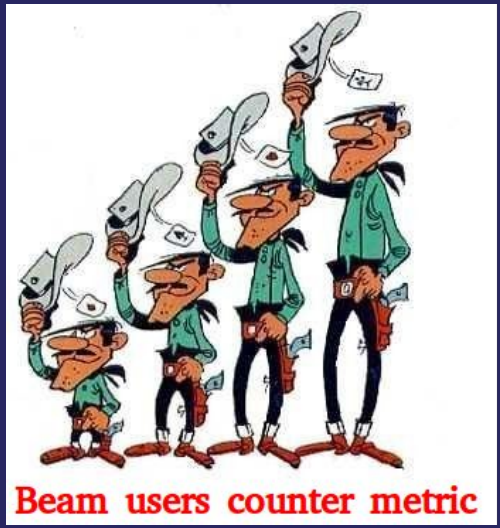
Use case: measure throughput using Beam metrics



Use case: Architecture



1. Introduction to Beam and portability
2. What are beam metrics ?
3. How does a pipeline author use them (java) ?
4. Inside the runner: how does it work ?
5. Outside of the runner: How are they extracted ?
6. Between the runners : portability
7. Example use case
8. Current state and future work



Current state and future work

- A few runners still need to implement the metrics API
  - Implement it for Apex and Gearpump runners.
- A few runners started to implement portability framework
  - Ongoing work for Dataflow and Flink
  - For the portable metrics
    - sdk/runner communication protocol defined
    - first step on the implementation in python sdk
- Most of the runners support only attempted metrics (except Dataflow)

Current state and future work

- System metrics: runner instrumentation (e.g failing elements on PCollections)
- Metrics Pusher is wired up only in Spark and Flink runners
    - Ongoing work for Dataflow
    - Metrics sinks:
        - Metrics Sink API is defined
        - HTTP REST sink available, Graphite sink ongoing
        - Implement sinks for popular backends (Prometheus, Ganglia, …)
        - Contribute new sinks ! (Metrics Sink API and HTTP Metrics Sink)

Take away

- Metrics can be defined in user code and collected in the native UI
- Metrics can be extracted to external backends independently of the chosen execution engine
- Metrics will be portable (whatever sdk to whatever runner)

References

Metrics

- **Metrics architecture**
- **User metrics**
- Portable metrics
- Metrics extraction

Apache Beam

https://beam.apache.org

Join the mailing lists!

user-subscribe@beam.apache.org
dev-subscribe@beam.apache.org

Follow @ApacheBeam on Twitter

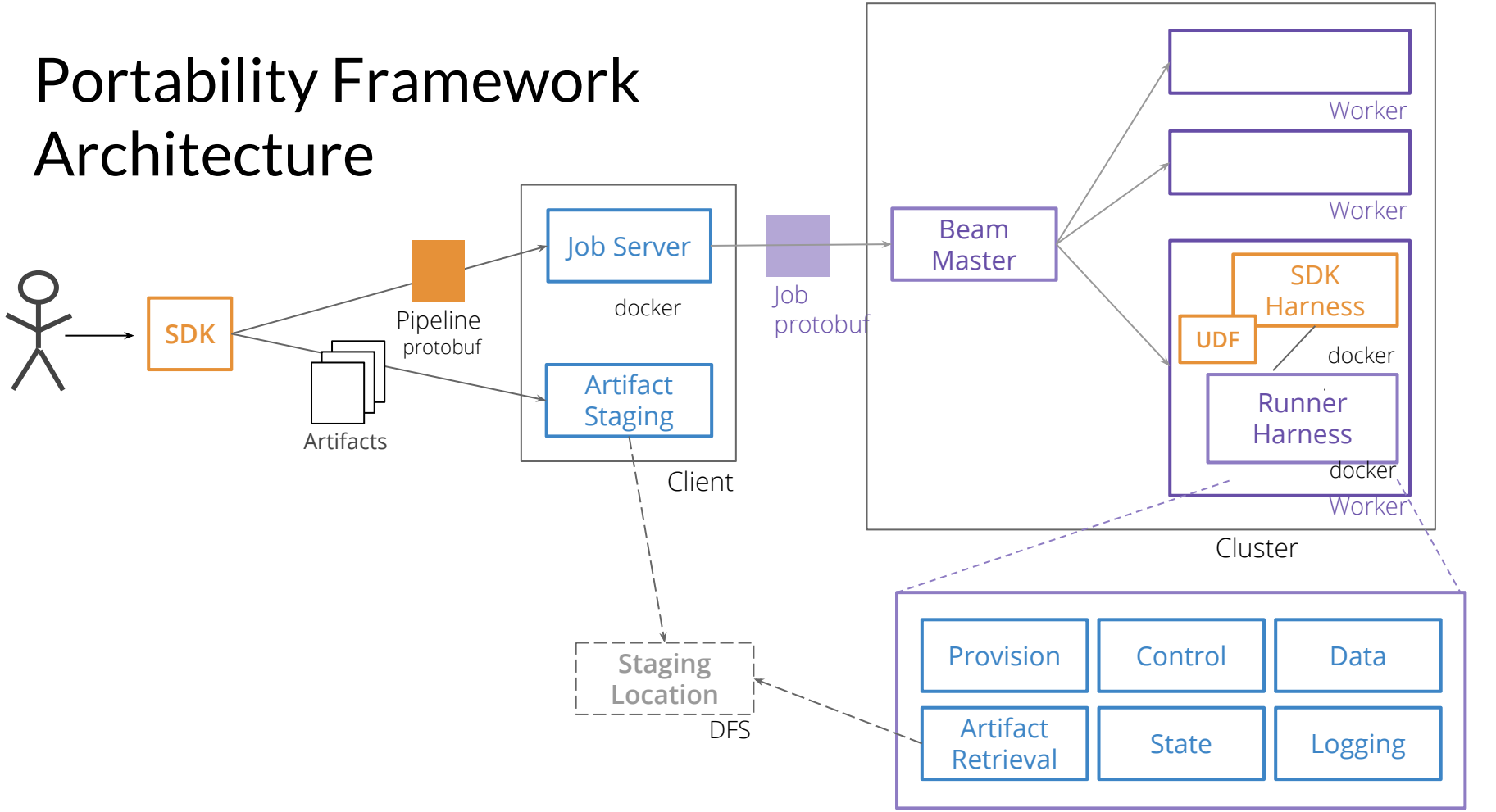
Thanks !

APACHECON North America

Sept. 24-27, 2018

Backup slides

Portability Framework Architecture



Abstract

Apache Beam provides a unified programming model to execute batch and streaming pipelines on all the popular big data engines. Lately it has gone beyond by also providing a unified way to supervise the pipeline execution: universal metrics. No matter what the chosen execution engine is, Apache Spark, Apache Flink or others, you specify the metrics to be collected with the same API and you extract them in a coherent way.

In this talk, you will discover the Beam metrics API, integration with the runners and finally an end to end example.