Python Streaming Pipelines with Beam on Flink

Flink Forward Berlin, 2018

Apache Beam

Apache Flink

https://s.apache.org/streaming-python-beam-flink

Aljoscha Krettek, Thomas Weise

Agenda

1. What is Beam?
2. The Beam Portability APIs
3. Executing Pythonic Beam Jobs on Flink
4. The Future

Problem

- Many of the big data ecosystem projects are Java / JVM based
- Use cases with different language environments
    - Python is the primary option for Machine Learning
- Barrier to entry for teams that want to adopt streaming but have no Java experience
- Cost of too many API styles and runtime environments
- (Currently no good option for native Python + Streaming)

Multi-Language Support in Beam

- Effort to support multiple languages in Beam started late 2016
- Python SDK on Dataflow available for ~ 1 year
- Go SDK added recently
- At Flink Forward 2017…

2018: Portable Flink Runner MVP near completion (~ Beam release 2.8.0)

Talk Python to Me

Stream Processing in Your Favourite Language with Beam on Flink

Apache Beam

Apache Flink

Based on work and slides by Frances Perry, Tyler Akidau, Kenneth Knowles & Sourabh Bajaj

Slides by Aljoscha Krettek, September 2017, Flink Forward 2017

What is Beam?

What is Apache Beam?

Apache Beam is a unified programming model designed to provide efficient and portable data processing pipelines

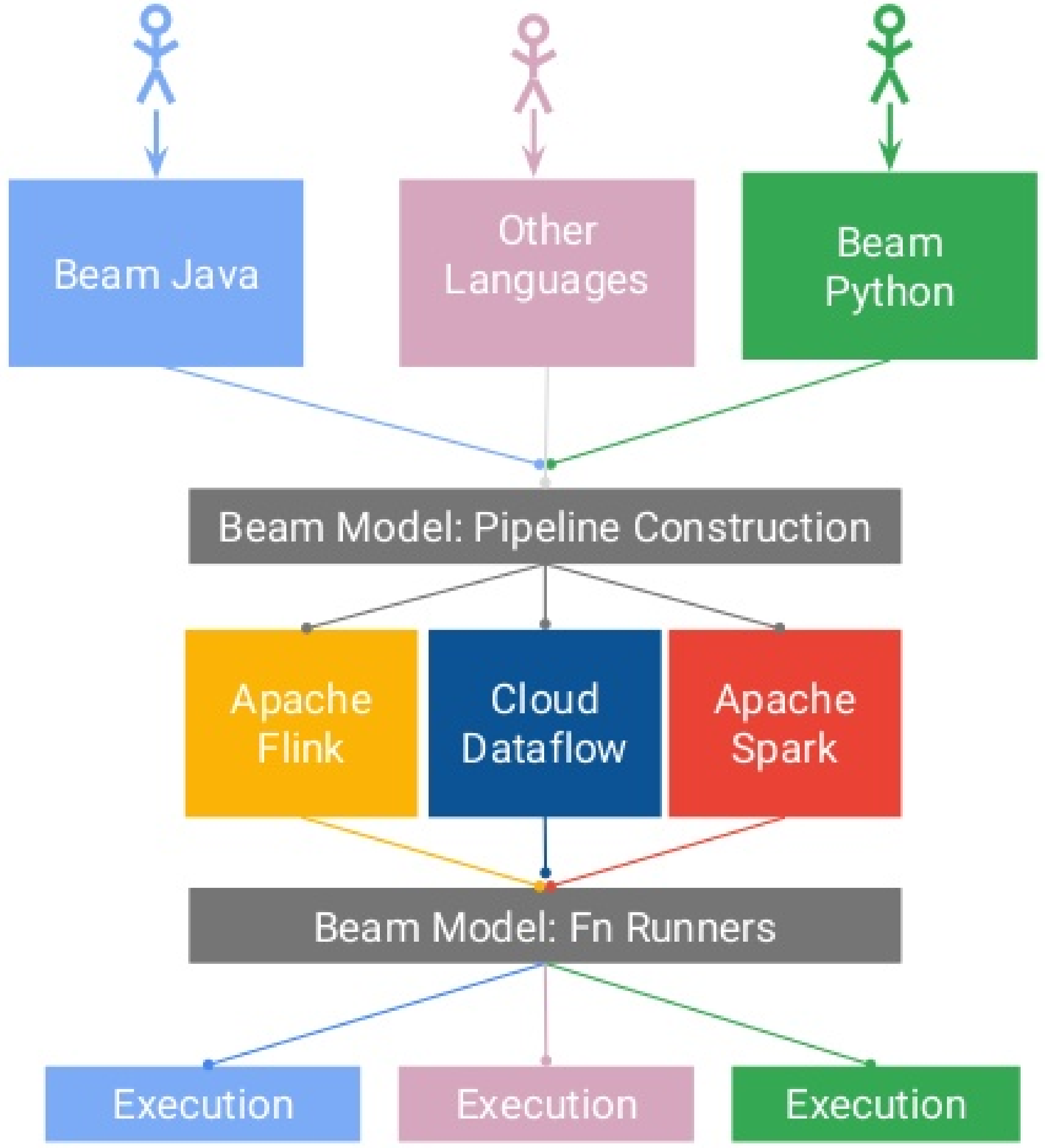
1. Unified model (**B**atch + str**EAM**)
   What / Where / When / How
2. **SDKs** (Java, Python, Go, ...) & **DSLs** (Scala, ...)
3. **Runners** for Existing Distributed Processing Backends (Google Dataflow, Spark, Flink, ...)
4. **IOs**: Data store Sources / Sinks

The Apache Beam Vision

1. **End users:** who want to write pipelines in a language that's familiar.

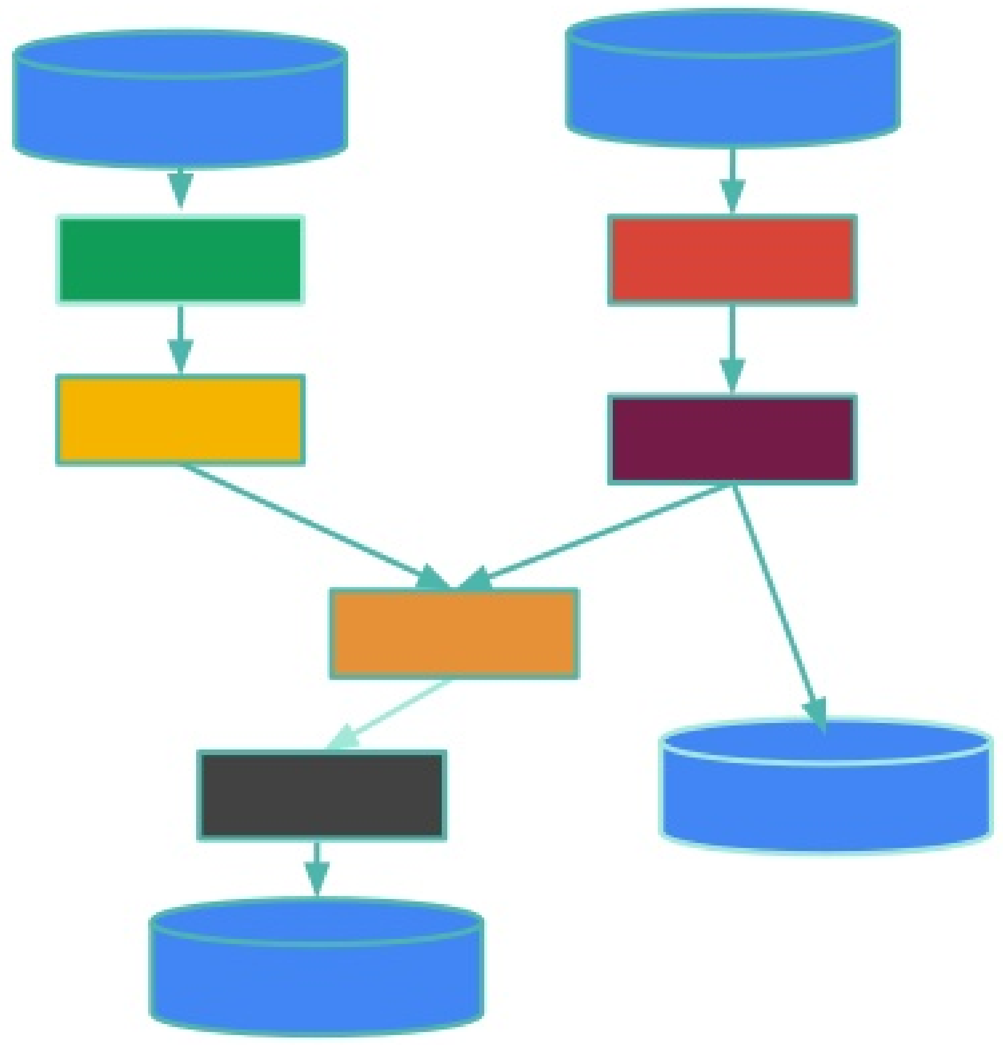
2. **SDK writers:** who want to make Beam concepts available in new languages.

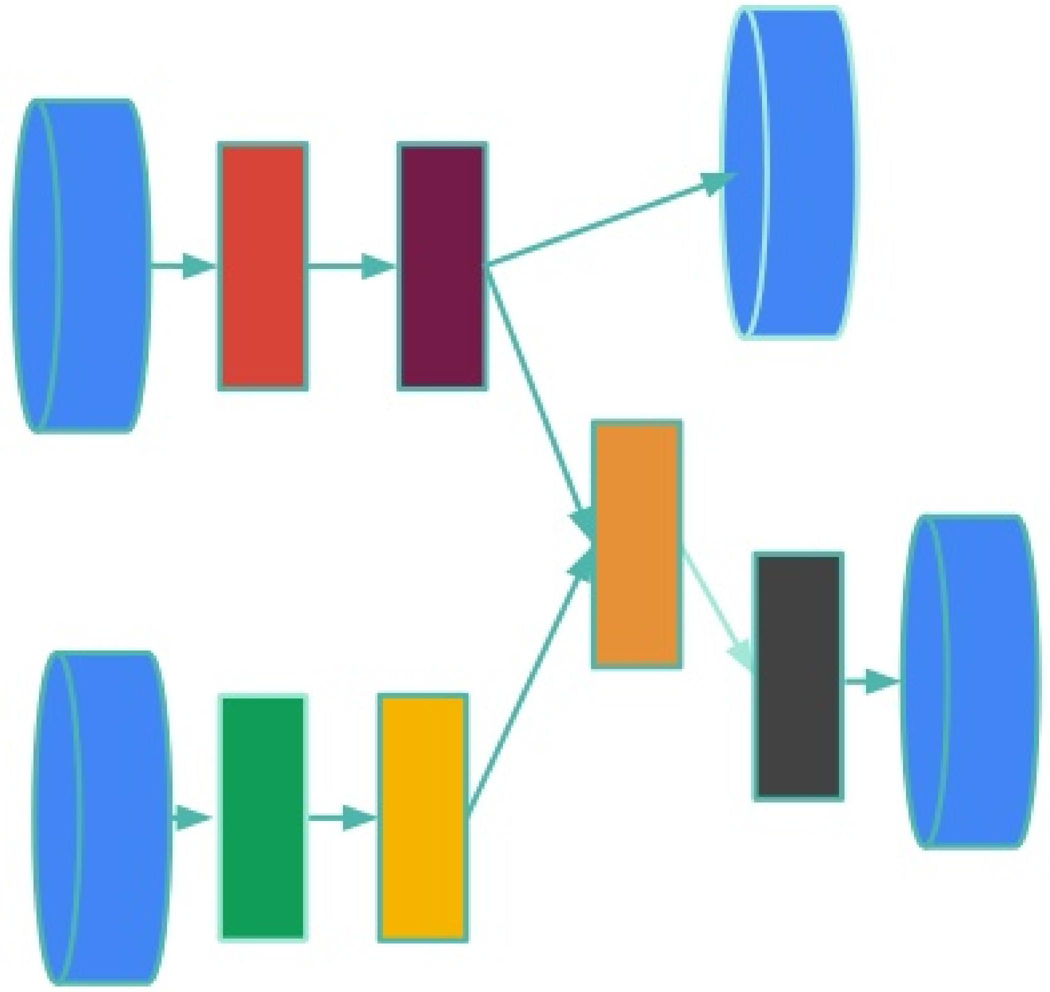
3. **Runner writers:** who have a distributed processing environment and want to support Beam pipelines



https://s.apache.org/apache-beam-project-overview

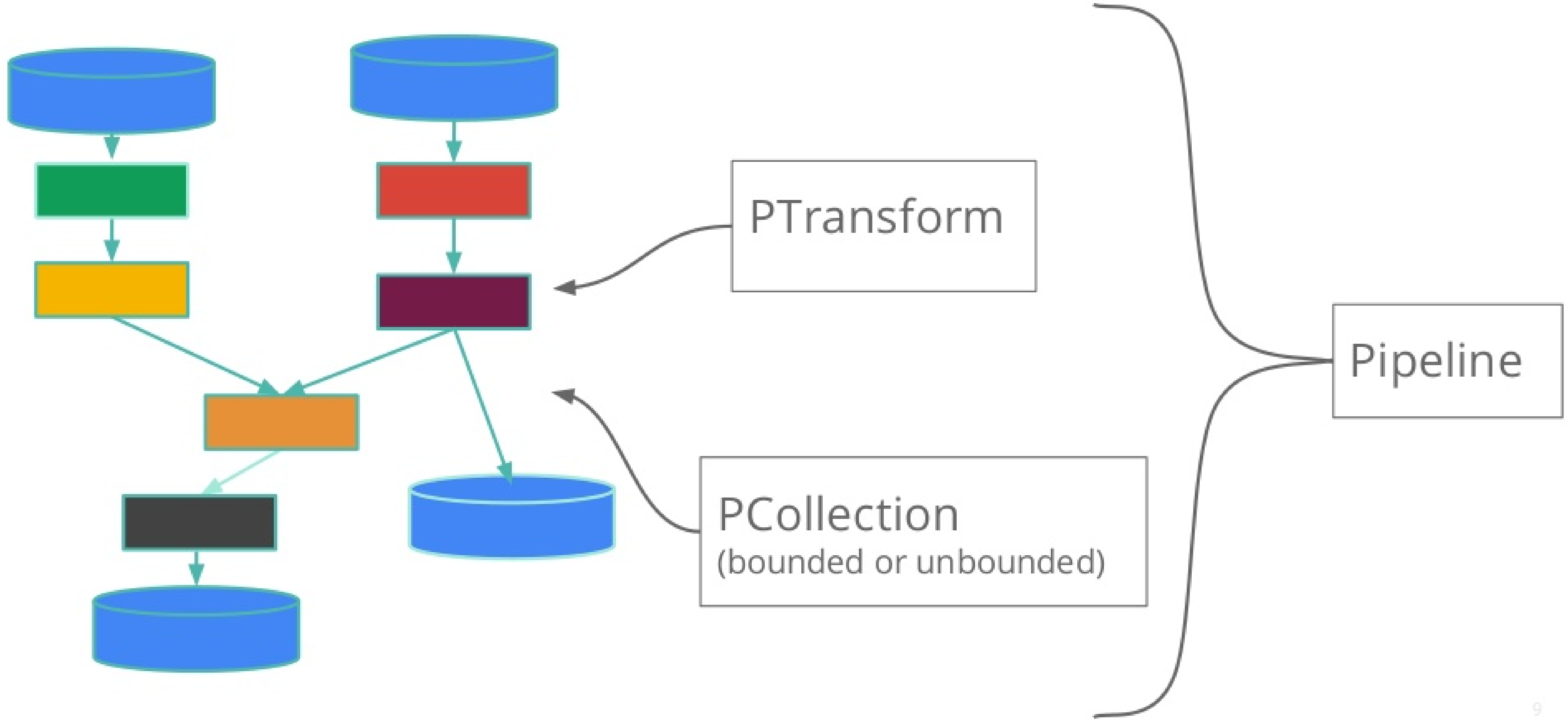
The Beam Model





(Flink draws it more like this)

The Beam Model



Beam Model: Asking the Right Questions

What results are calculated?

Where in event time are results calculated?

When in processing time are results materialized?

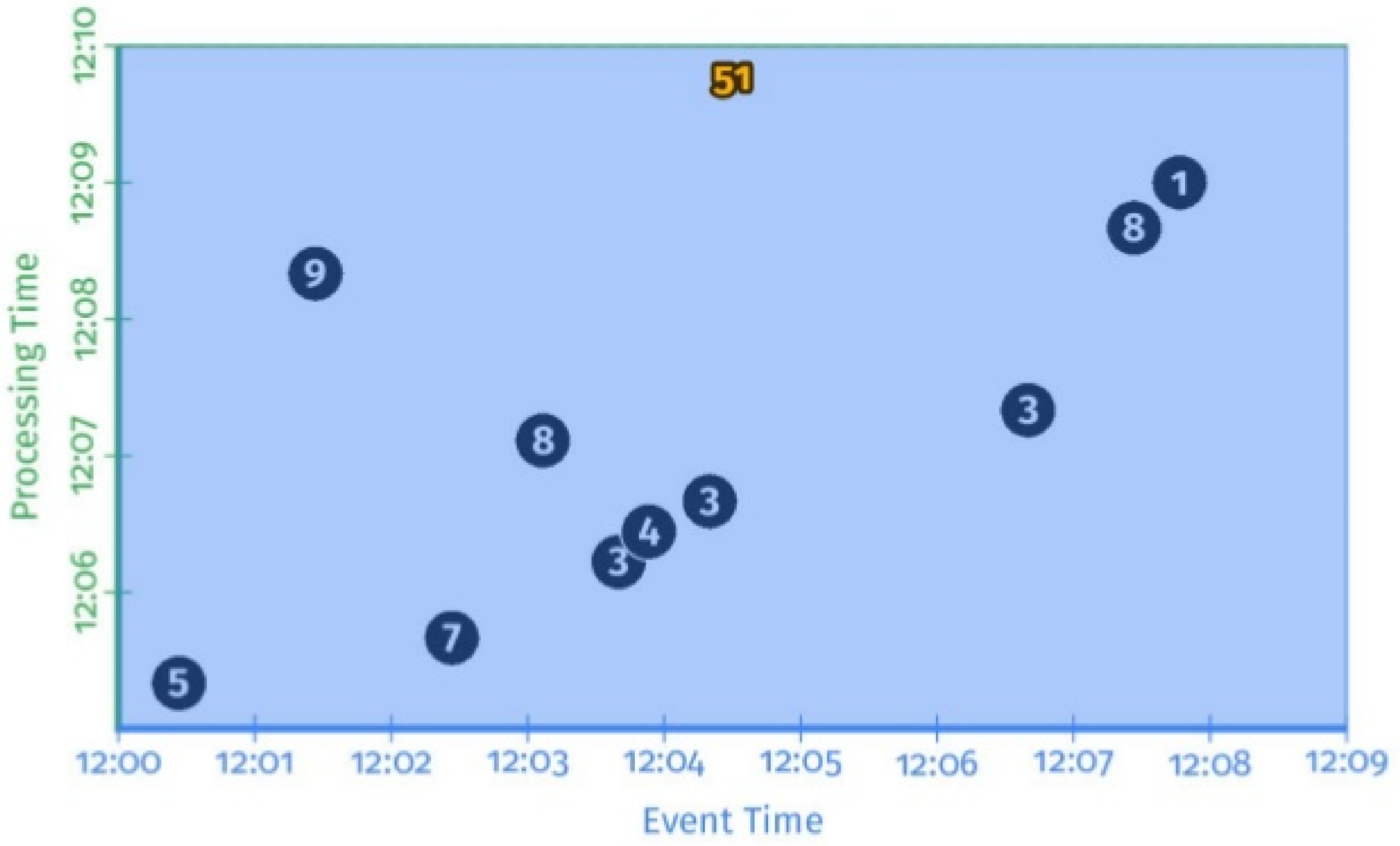
How do refinements of results relate?

The Beam Model: **What** is Being Computed?

```
PCollection<KV<String, Integer>> scores = input
     .apply(Sum.integersPerKey());
```

```
scores= (input
     | Sum.integersPerKey())
```

The Beam Model: **What** is Being Computed?

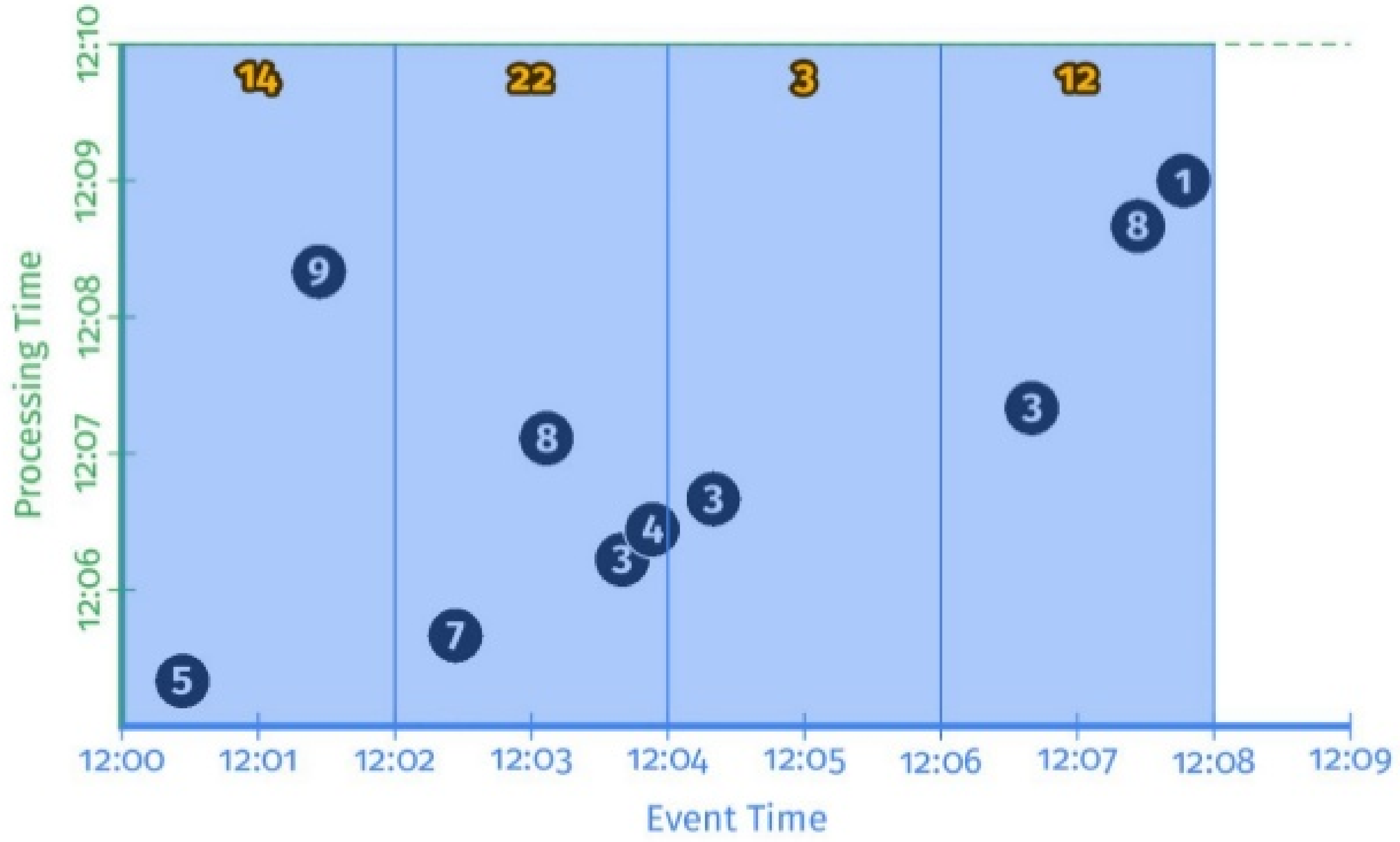


The Beam Model: **Where** in Event Time?

```
PCollection<KV<String, Integer>> scores = input
     .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
     .apply(Sum.integersPerKey());
```

```
scores= (input
     | beam.WindowInto(FixedWindows(2 * 60))
     | Sum.integersPerKey())
```

The Beam Model: **Where** in Event Time?

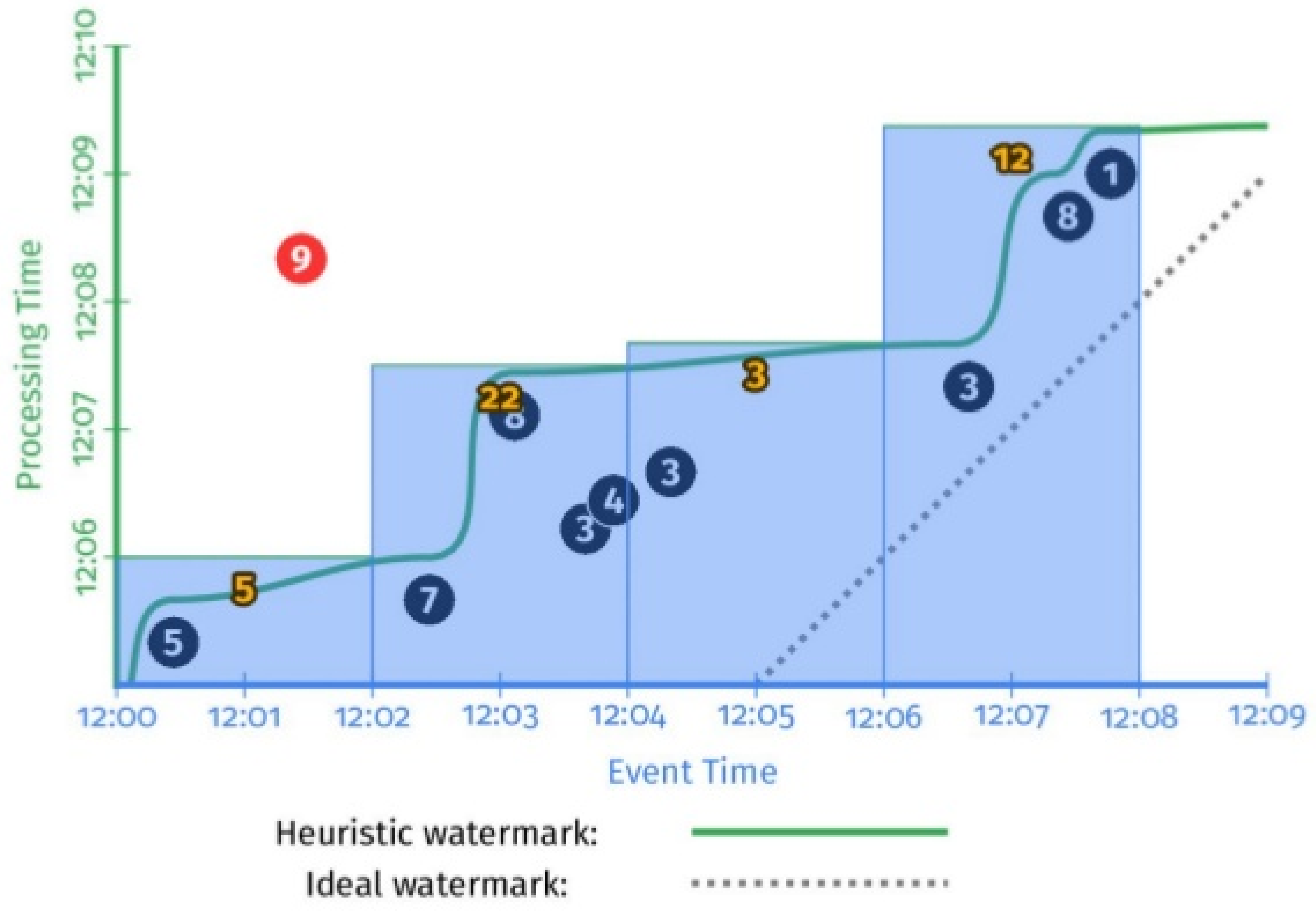


The Beam Model: **When** in Processing Time?

```
PCollection<KV<String, Integer>> scores = input
    .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))
          .triggering(AtWatermark()))
    .apply(Sum.integersPerKey());
```

```
scores = (input
    | beam.WindowInto(FixedWindows(2 * 60)
        .triggering(AtWatermark()))
    | Sum.integersPerKey())
```

The Beam Model: **When** in Processing Time?

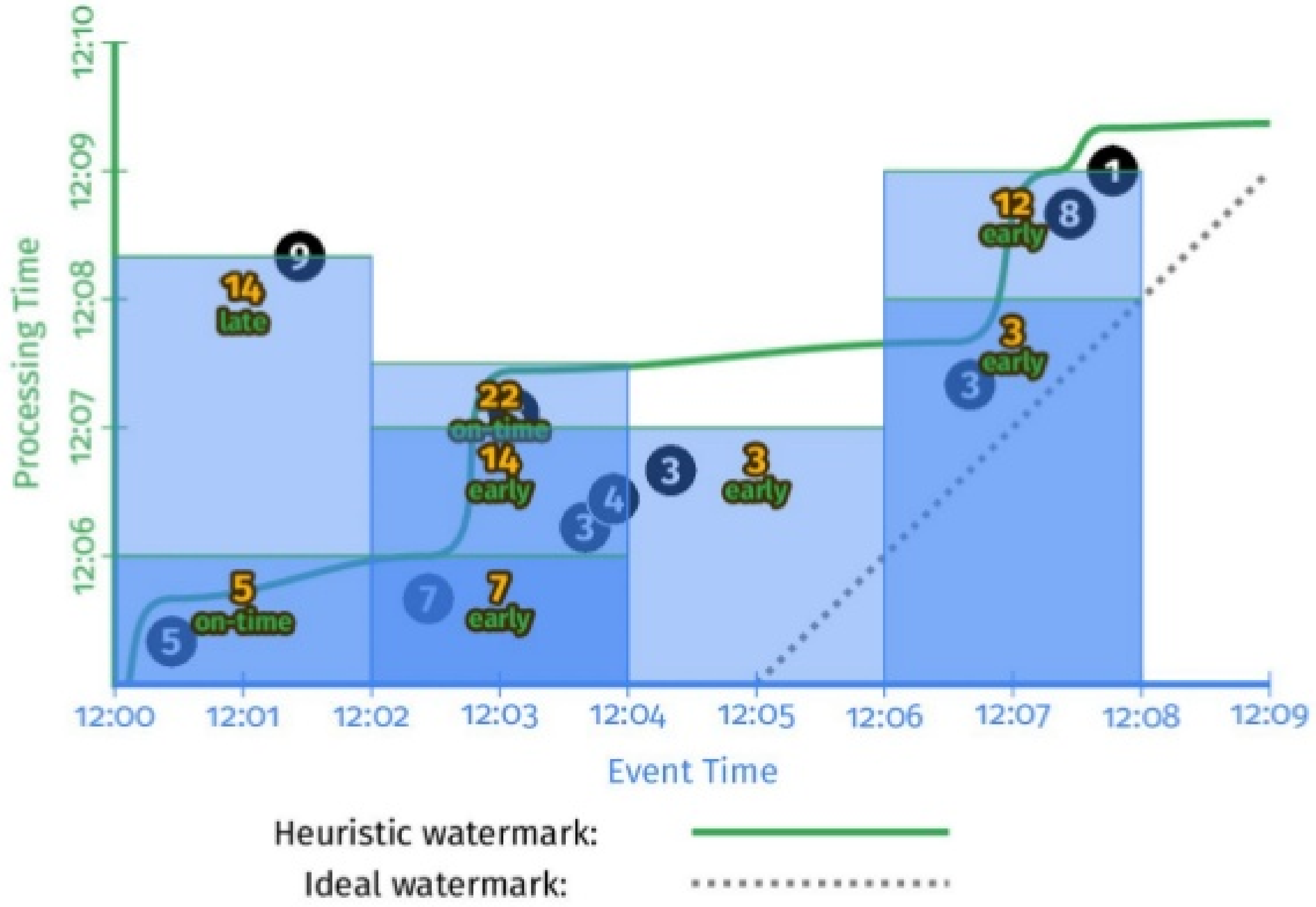


The Beam Model: **How** Do Refinements Relate?

```
PCollection<KV<String, Integer>> scores = input
    .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))
          .triggering(AtWatermark()
             .withEarlyFirings(AtPeriod(Duration.standardMinutes(1)))
             .withLateFirings(AtCount(1)))
          .accumulatingFiredPanes())
    .apply(Sum.integersPerKey());
```

```
scores = (input
    | beam.WindowInto(FixedWindows(2 * 60)
        .triggering(AtWatermark()
            .withEarlyFirings(AtPeriod(1 * 60))
            .withLateFirings(AtCount(1)))
        .accumulatingFiredPanes())
    | Sum.integersPerKey())
```

The Beam Model: **How** Do Refinements Relate?



Example of Pythonic Beam Code

```
import apache_beam as beam
with beam.Pipeline() as p:
    (p
        | beam.io.ReadStringsFromPubSub("twitter_topic")
        | beam.WindowInto(SlidingWindows(5*60, 1*60))
        | beam.Map(ParseHashTagDoFn())
        | beam.CombinePerKey(sum)
        | beam.Map(BigQueryOutputFormatDoFn())
        | beam.io.WriteToBigQuery("trends_table"))
```

Runners

Runners "**translate**" the code into the target runtime

Apache Beam Direct Runner

Apache Apex

Apache Spark

Apache Flink

Apache Gearpump

Apache Samza

Google Cloud Dataflow

IBM Streams

Apache Storm WIP

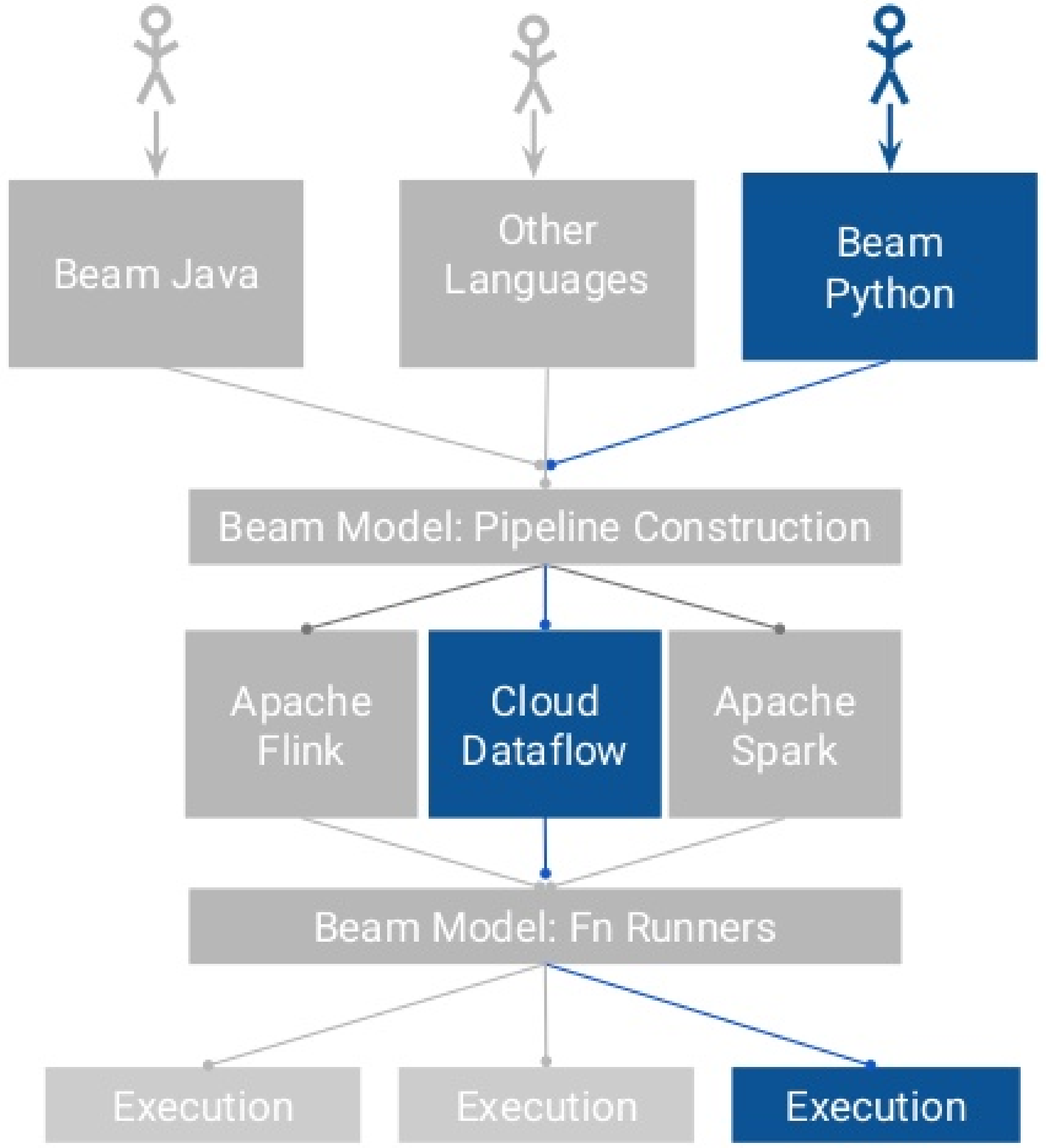
Ali Baba JStorm

Hadoop MapReduce

* Same code, different runners & runtimes

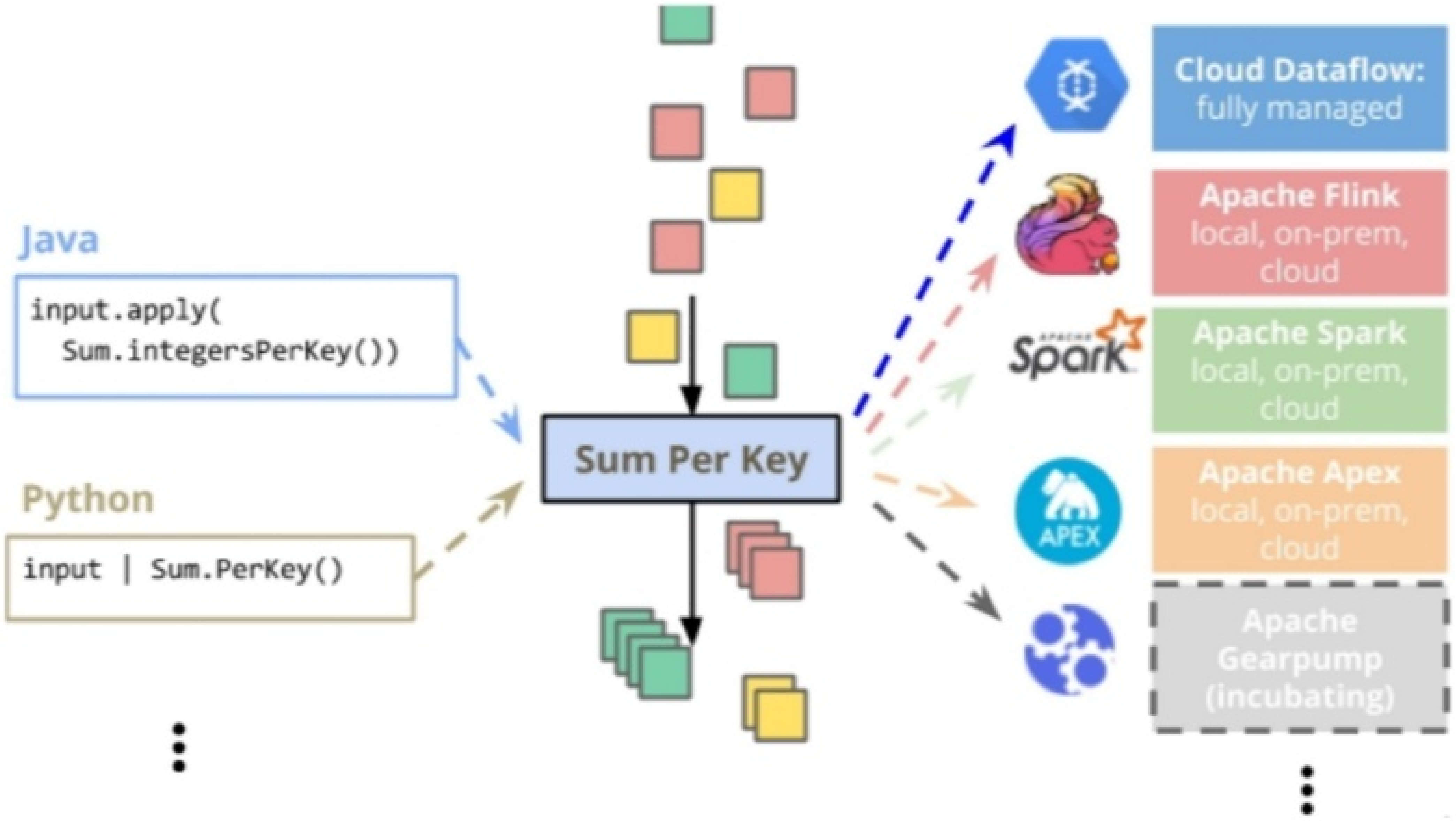
Awesome but...

- Can a Python pipeline run on any of the Java/JVM based runners?
- Can I use the Python Tensorflow transform from a Java pipeline?
- I want to read from Kafka in my Python pipeline, but there is no connector - can I use the Java implementation?



Beam Portability

What are we trying to solve?

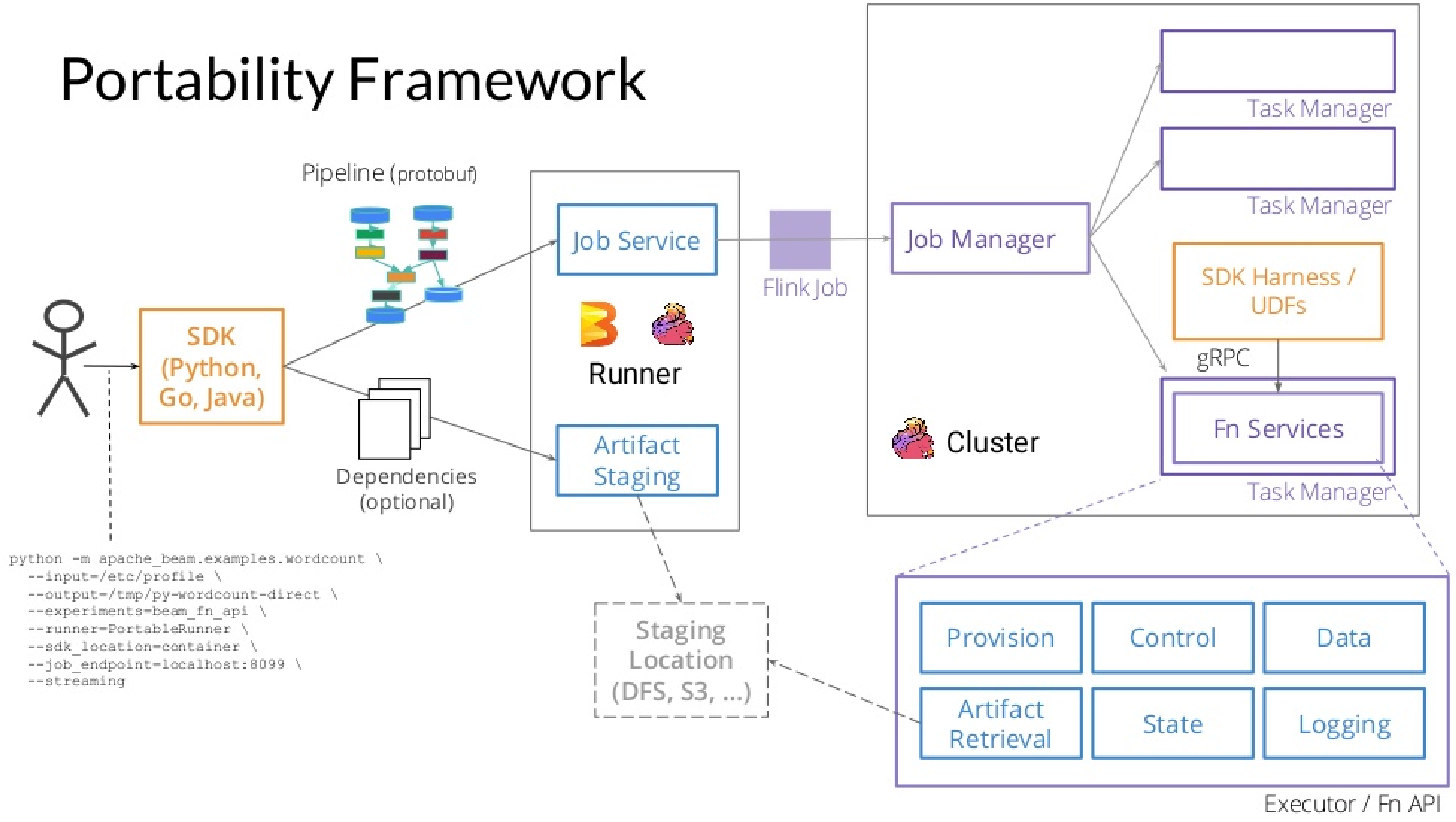


Java
Python
Sum Per Key
Cloud Dataflow: fully managed
Apache Flink local, on-prem, cloud
Apache Spark local, on-prem, cloud
Apache Apex local, on-prem, cloud
Apache Gearpump (incubating)
APEX
Spark

```
input.apply(
  Sum.integersPerKey())
```

```
input | Sum.PerKey()
```

Portability Framework



```
python -m apache_beam.examples.wordcount \
  --input=/etc/profile \
  --output=/tmp/py-wordcount-direct \
  --experiments=beam_fn_api \
  --runner=PortableRunner \
  --sdk_location=container \
  --job_endpoint=localhost:8099 \
  --streaming
```

APIs for Different Pipeline Lifecycle Stages

Pipeline API

- Used by the SDK to construct SDK-agnostic Pipeline representation
- Used by the Runner to translate a Pipeline to runner-specific operations

Job API

- Launching and interacting with a running Pipeline

Fn API

- Used by an SDK harness for communication with a Runner
- User by the Runner to push work into an SDK harness

Pipeline API (simplified)

- Definition of common primitive transformations (Impulse, ExecutableStage, Flatten, AssignWindow, GroupByKey, Reshuffle)
- Definition of serialized Pipeline (protobuf)

```
Pipeline = {PCollection*, PTransform*,
            WindowingStrategy*, Coder*}
PTransform = {Inputs*, Outputs*, FunctionSpec}
FunctionSpec = {URN, payload}
```

https://s.apache.org/beam-runner-api

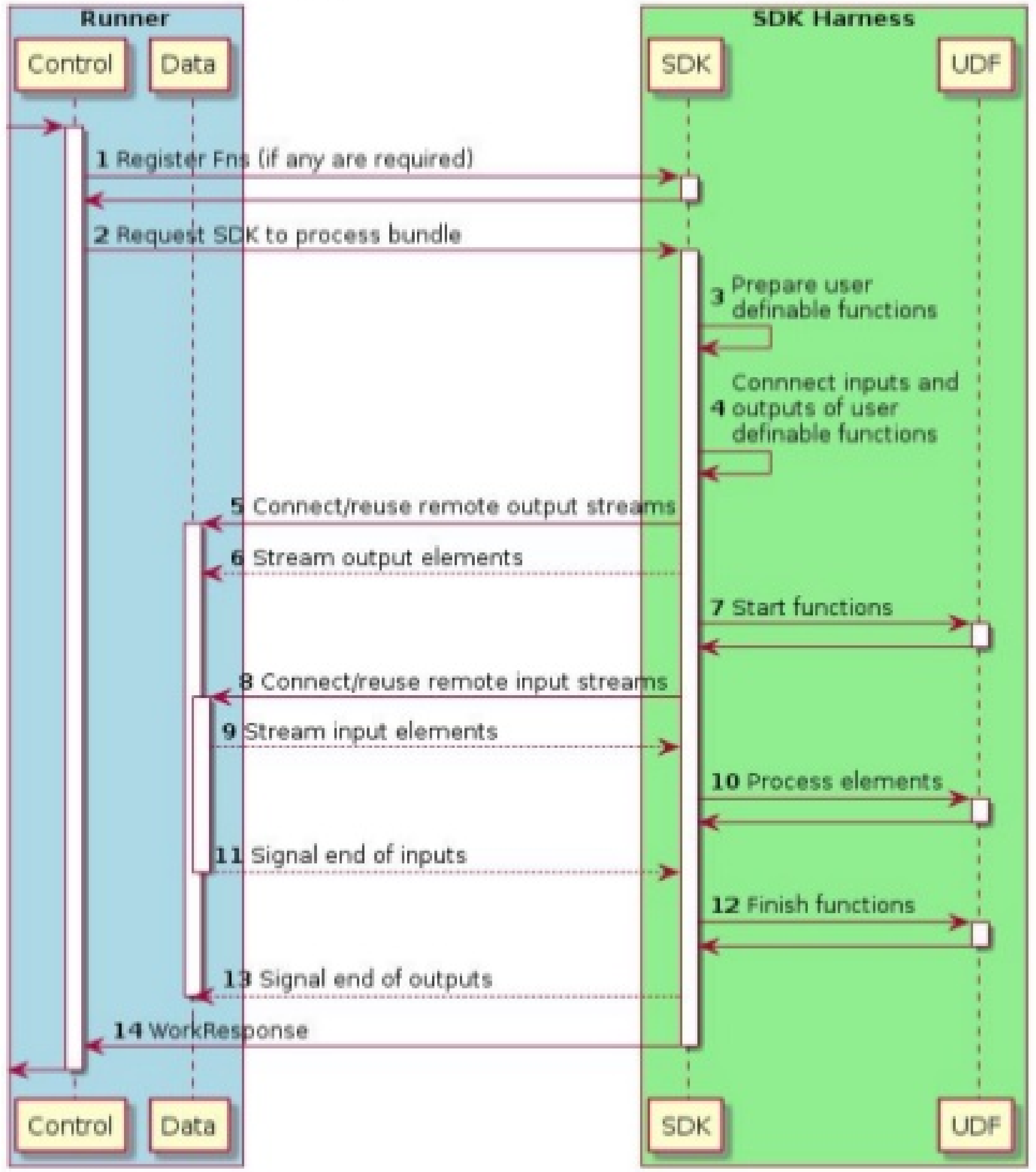
Fn API

gRPC interfaces for communication between SDK harness and Runner

- **Control**: Used to tell the SDK which UDFs to execute and when to execute them.
- **Data**: Used to move data between the language specific SDK harness and the runner.
- **State**: Used to support user state, side inputs, and group by key reiteration.
- **Logging**: Used to aggregate logging information from the language specific SDK harness.

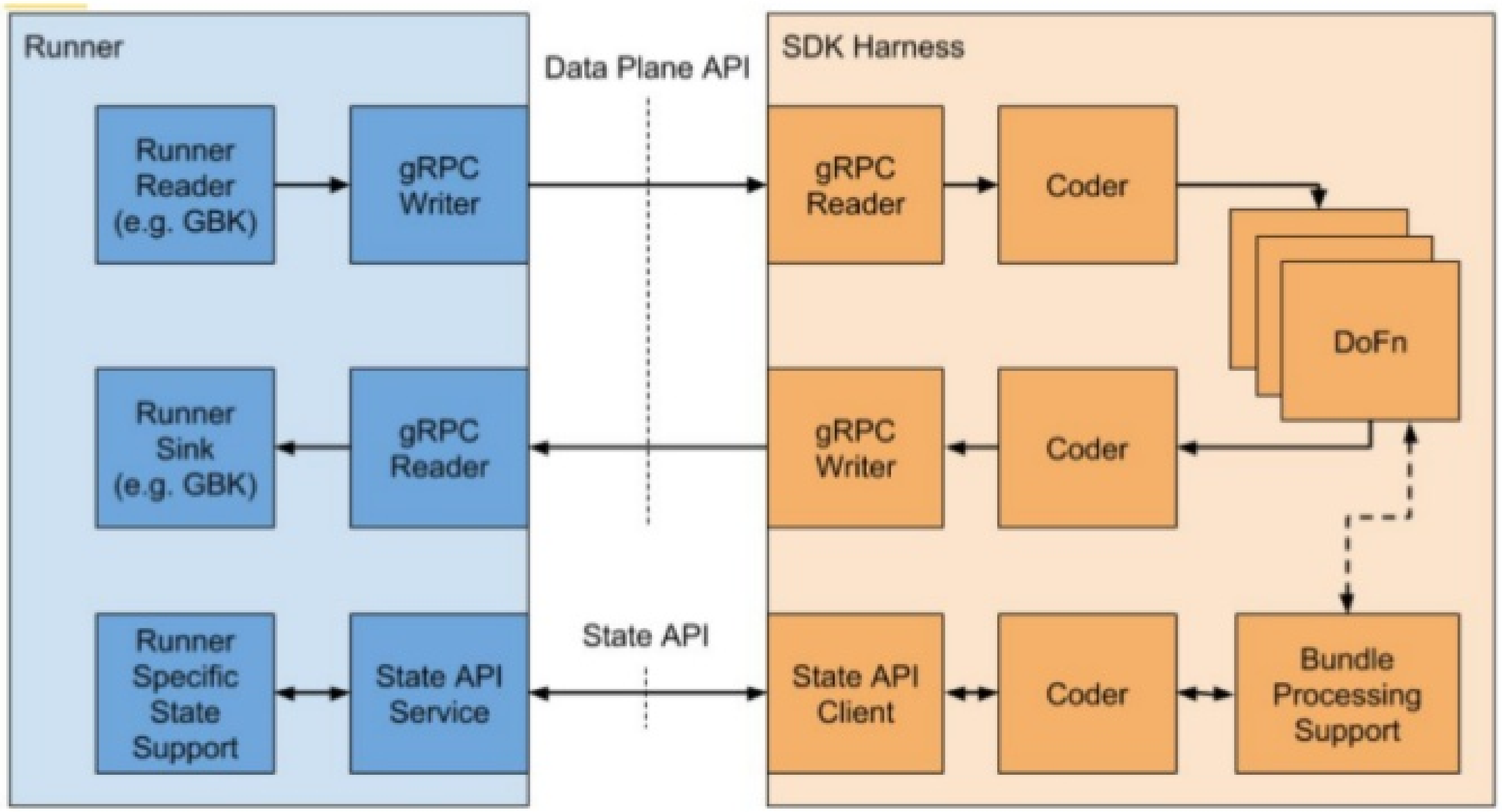
https://s.apache.org/beam-fn-api

Fn API - Bundle Processing


Runner
SDK Harness
Control
Data
SDK
UDF
1 Register Fns (if any are required)
2 Request SDK to process bundle
3 Prepare user definable functions
4 Connnect inputs and outputs of user definable functions
5 Connect/reuse remote output streams
6 Stream output elements
7 Start functions
8 Connect/reuse remote input streams
9 Stream input elements
10 Process elements
11 Signal end of inputs
12 Finish functions
13 Signal end of outputs
14 WorkResponse
Control
Data
SDK
UDF

https://s.apache.org/beam-fn-api-processing-a-bundle

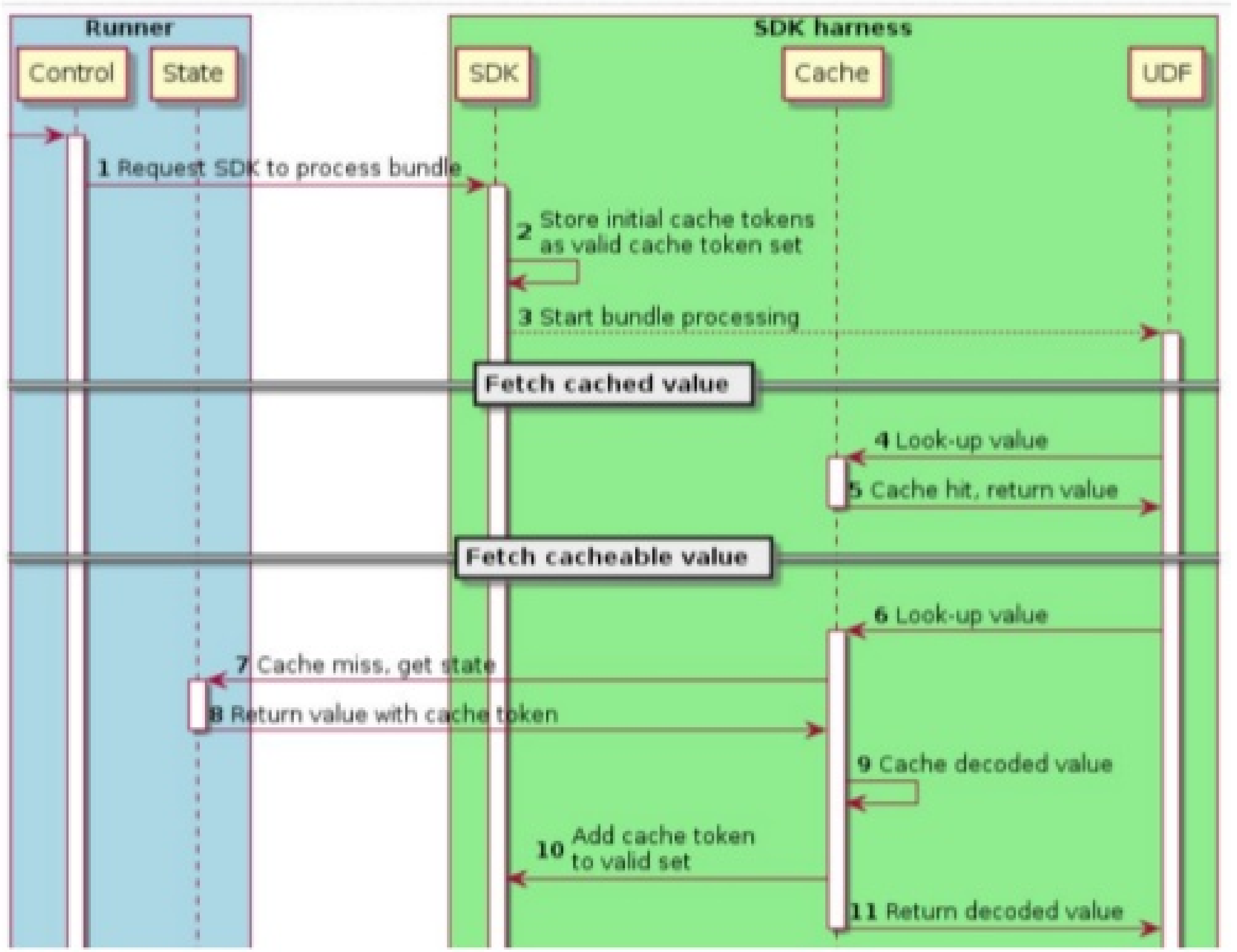
Fn API - Data



https://s.apache.org/beam-fn-api-send-and-receive-data

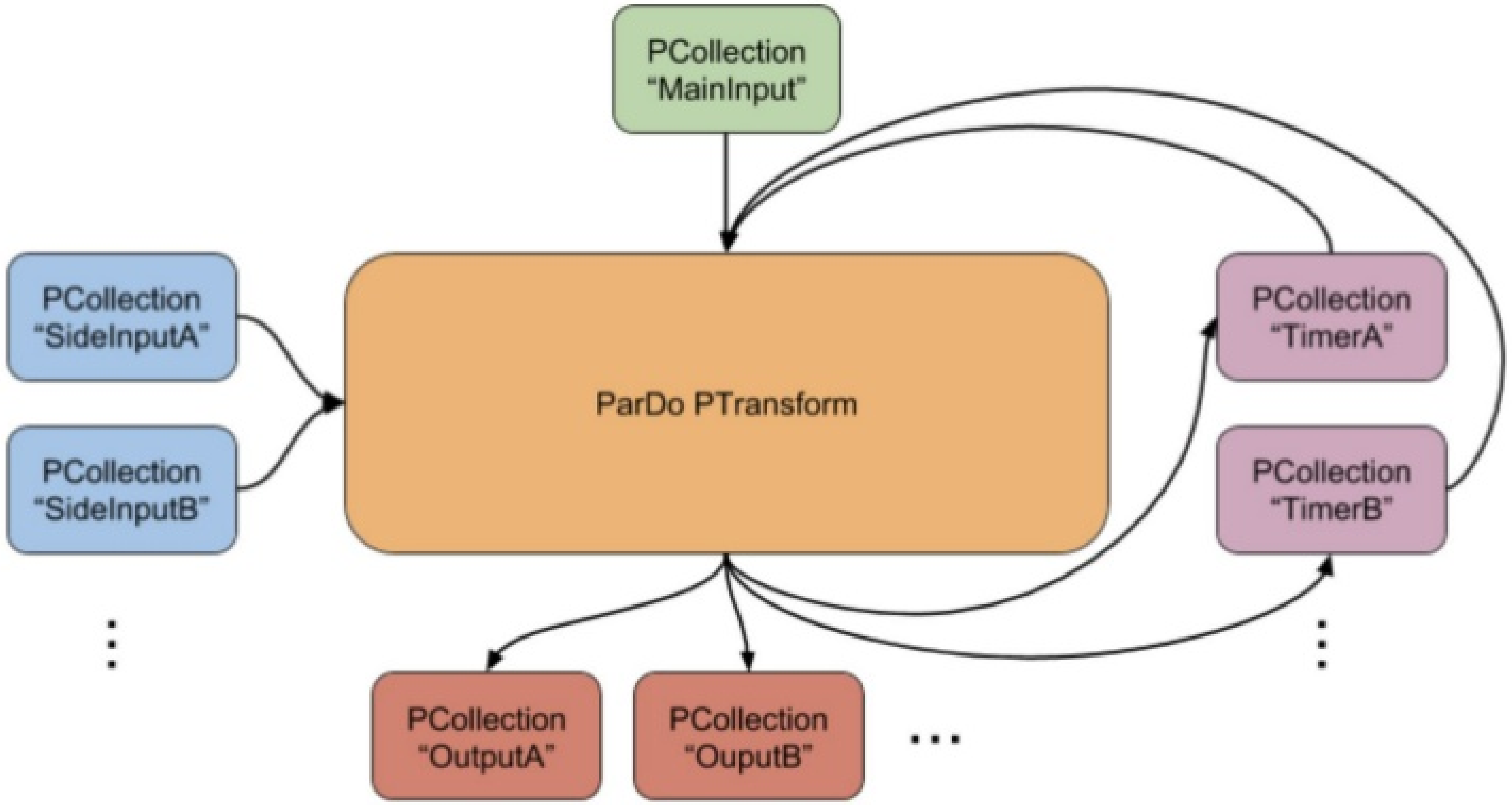
Fn API - State

User state, side inputs


Runner
SDK harness
Control
State
SDK
Cache
UDF
1 Request SDK to process bundle
2 Store initial cache tokens as valid cache token set
3 Start bundle processing
Fetch cached value
4 Look-up value
5 Cache hit, return value
Fetch cacheable value
6 Look-up value
7 Cache miss, get state
8 Return value with cache token
9 Cache decoded value
10 Add cache token to valid set
11 Return decoded value

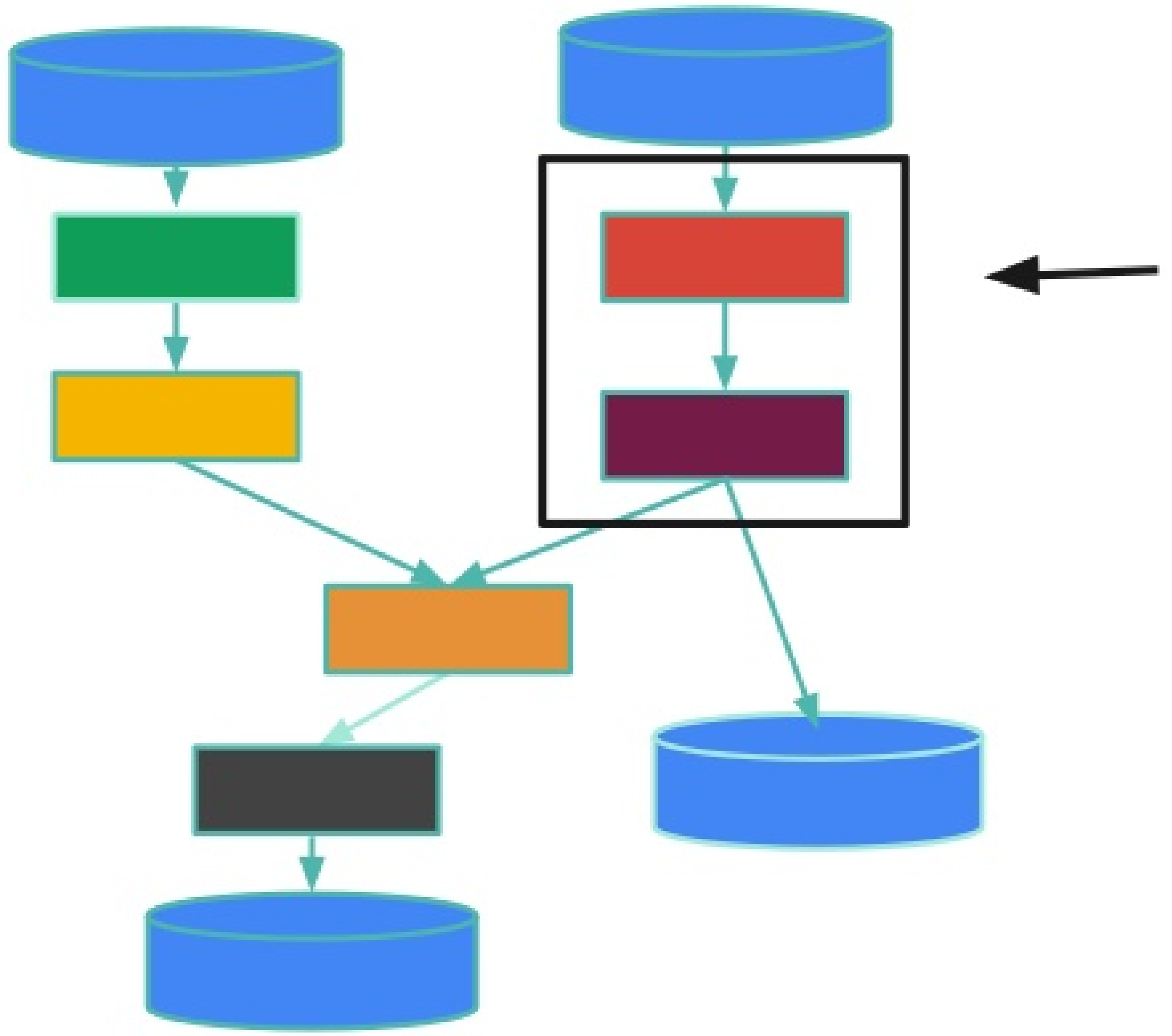
https://s.apache.org/beam-fn-state-api-and-bundle-processing

Fn API - Timers



https://s.apache.org/beam-portability-timers

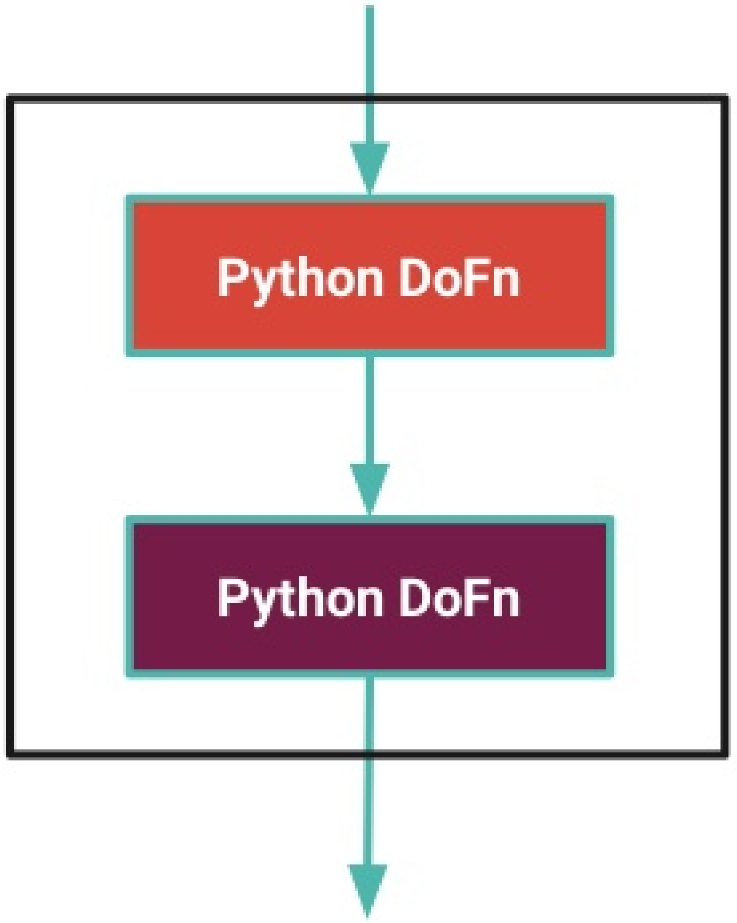
Fn API - Processing DoFns (Executable Stages)



Say we need to execute this part

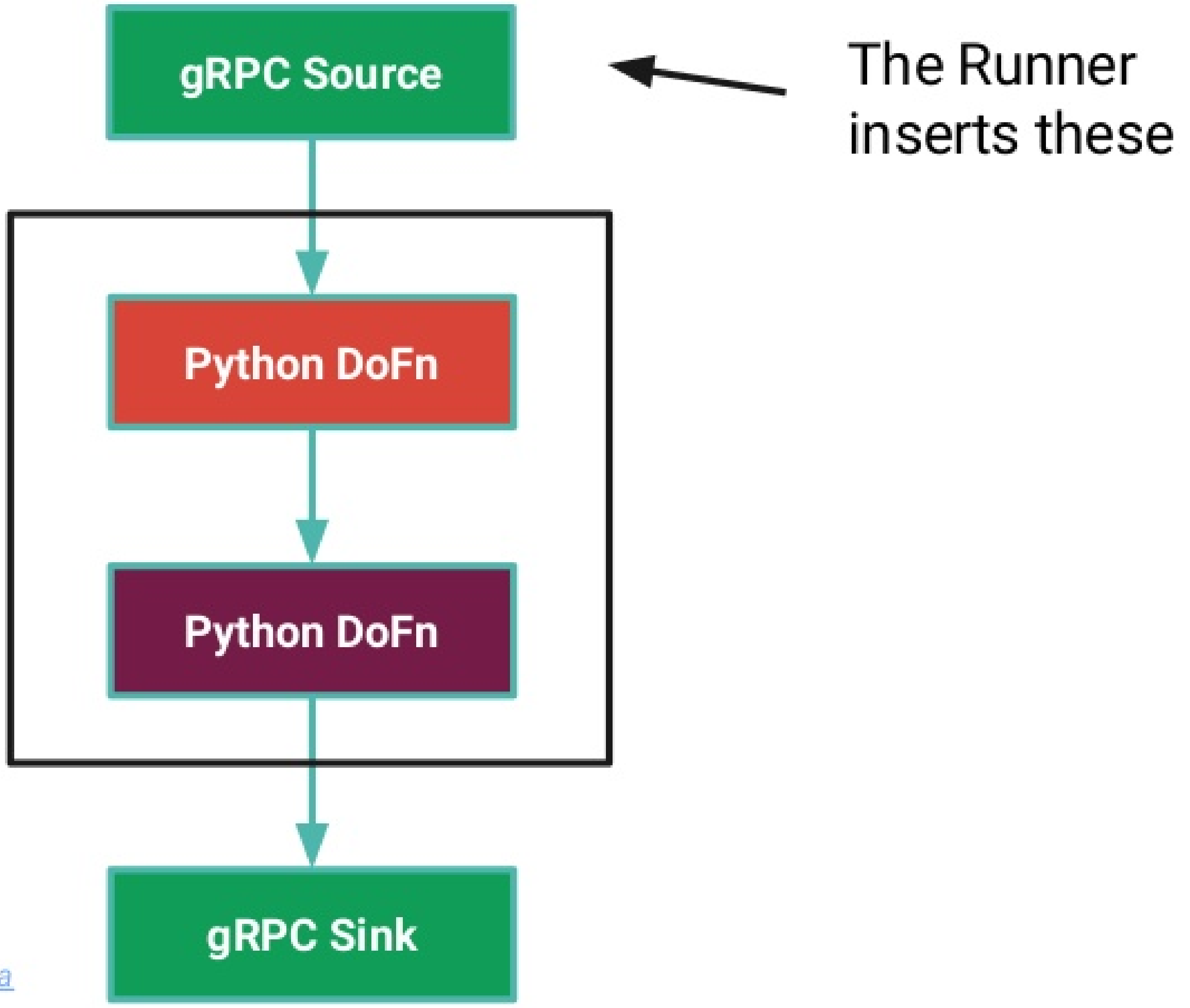
https://s.apache.org/beam-fn-api-send-and-receive-data

Fn API - Processing DoFns



https://s.apache.org/beam-fn-api-send-and-receive-data

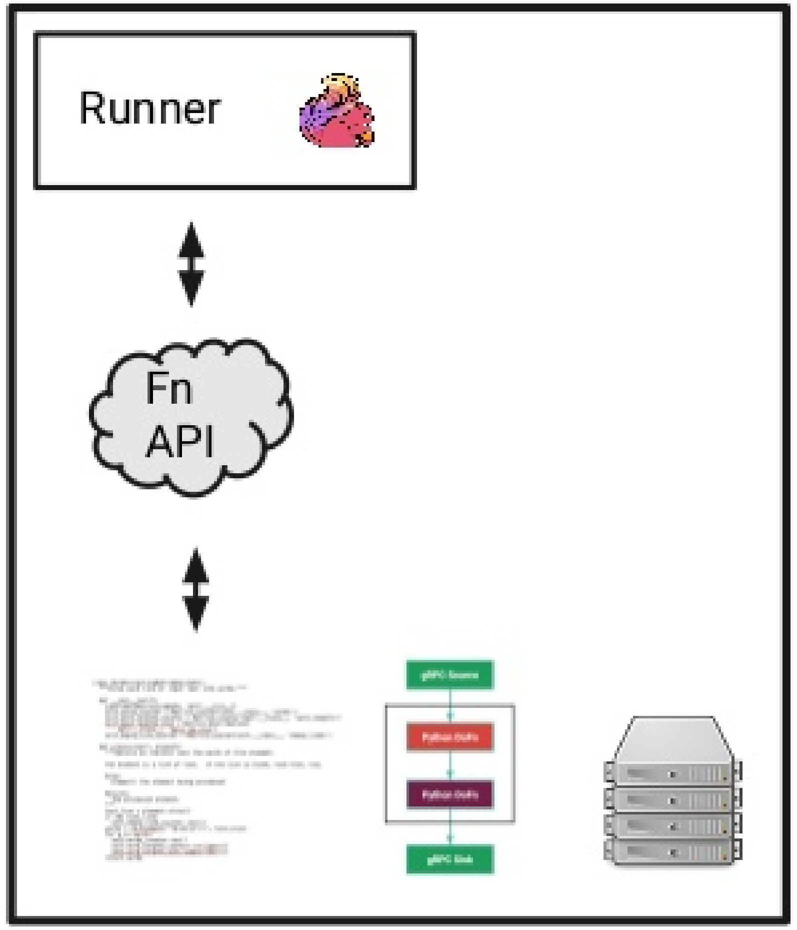
Fn API - Processing DoFns (Pipeline manipulation)



https://s.apache.org/beam-fn-api-send-and-receive-data

Fn API - Executing the user Fn in the SDK Harness

- Environments
  - Docker container
  - Separate process (BEAM-5187)
  - Embedded (SDK and runner same language) - TBD
- Repository of containers for different SDKs
- Container is user-configurable
- User code can be added to container at runtime (artifact retrieval service)



https://s.apache.org/beam-fn-api-container-contract

Executing Pythonic* Beam Jobs on Fink

*or other languages

What is the (Flink) Runner doing in all this?

- Provide Job Service endpoint (Job Management API)
- Translate portable pipeline representation to native API
- Provide gRPC endpoints for control/data/logging/state plane
- Manage SDK Harness processes that execute user code
- Execute bundles (with arbitrary user code) using the Fn API
- Manage state for side inputs, user state/timers

Reference runner provides common implementation baseline for JVM based runners (/runners/java-fn-execution) and we have a portable Validate Runner integration test suite in Python!

What’s specific to the Flink Runner?

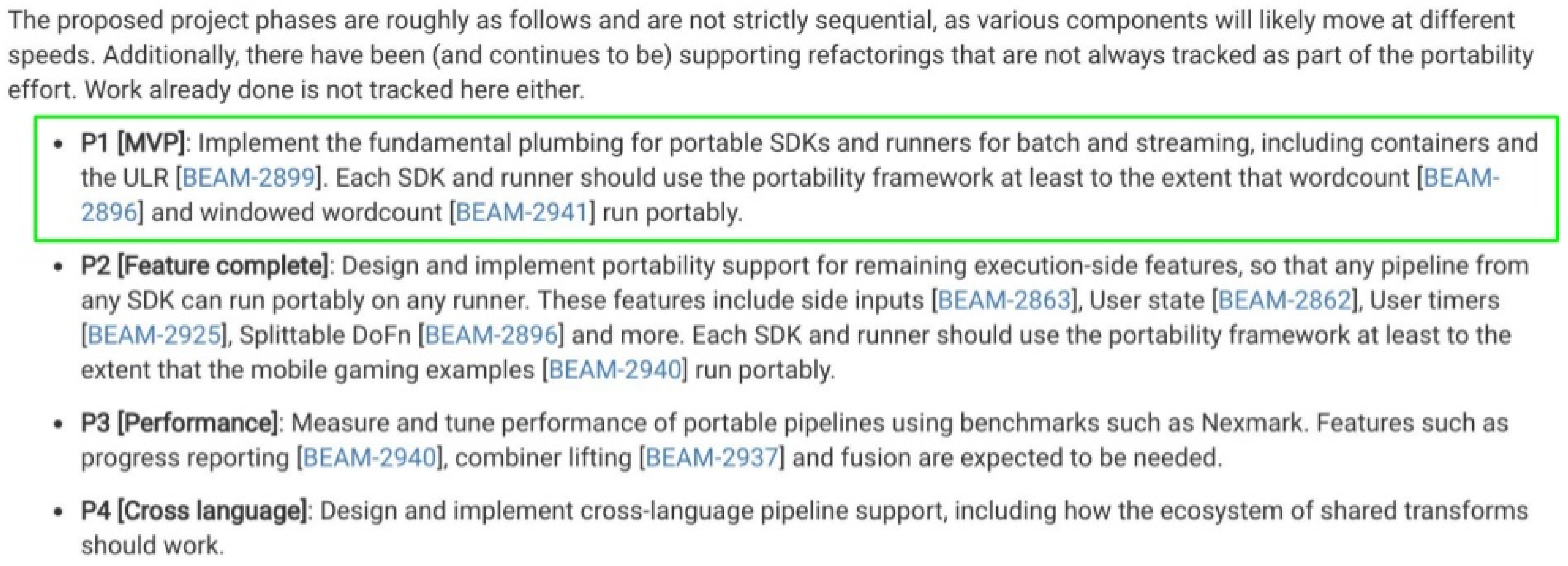
- Job Server packaging (fat jar and docker container)
- Pipeline translators for batch (DataSet) and streaming (DataStream)
    - Translation/operators for primitive URNs: Impulse, Flatten, GBK, Assign Windows, Executable Stage, Reshuffle
- Side input handlers based on Flink State
- User State and Timer integration (TBD)
- Flink Job Launch (same as old, non-portable runner)

Advantages/Disadvantages

- Complete isolation of user code
- Configurability of execution environment (Docker, ...)
- Support for code written in non-JVM languages
- Ability to mix code written in different languages in a single pipeline (future)

- Slower (RPC overhead)
- Using Docker requires docker 😉
  - Direct Process Executor WIP
- Early Adoption (complete runner overhaul)

Roadmap

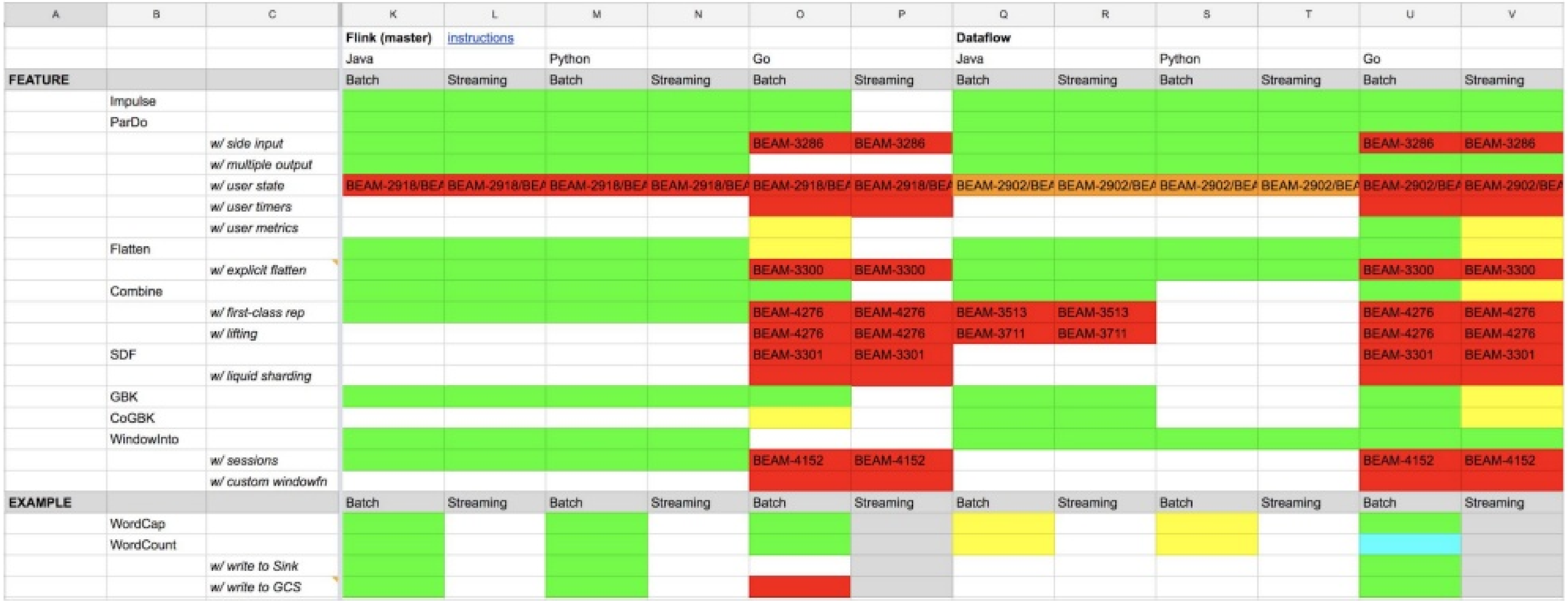


The proposed project phases are roughly as follows and are not strictly sequential, as various components will likely move at different speeds. Additionally, there have been (and continues to be) supporting refactorings that are not always tracked as part of the portability effort. Work already done is not tracked here either.

- **P1 [MVP]**: Implement the fundamental plumbing for portable SDKs and runners for batch and streaming, including containers and the ULR [BEAM-2899]. Each SDK and runner should use the portability framework at least to the extent that wordcount [BEAM-2896] and windowed wordcount [BEAM-2941] run portably.
- **P2 [Feature complete]**: Design and implement portability support for remaining execution-side features, so that any pipeline from any SDK can run portably on any runner. These features include side inputs [BEAM-2863], User state [BEAM-2862], User timers [BEAM-2925], Splittable DoFn [BEAM-2896] and more. Each SDK and runner should use the portability framework at least to the extent that the mobile gaming examples [BEAM-2940] run portably.
- **P3 [Performance]**: Measure and tune performance of portable pipelines using benchmarks such as Nexmark. Features such as progress reporting [BEAM-2940], combiner lifting [BEAM-2937] and fusion are expected to be needed.
- **P4 [Cross language]**: Design and implement cross-language pipeline support, including how the ecosystem of shared transforms should work.

https://beam.apache.org/contribute/portability/

Feature Support Matrix as of Beam 2.7.0



A	B	C	K	L	M	N	O	P	Q	R	S	T	U	V
			Flink (master)	instructions					**Dataflow**					
			Java		Python		Go		Java		Python		Go	
FEATURE			Batch	Streaming	Batch	Streaming	Batch	Streaming	Batch	Streaming	Batch	Streaming	Batch	Streaming
	Impulse		🟩	🟩	🟩	🟩	🟩		🟩	🟩	🟩	🟩	🟩	🟩
	ParDo		🟩	🟩	🟩	🟩	🟩		🟩	🟩	🟩	🟩	🟩	🟩
		w/ side input	🟩	🟩	🟩	🟩	🟥 BEAM-3286	🟥 BEAM-3286	🟩	🟩	🟩	🟩	🟥 BEAM-3286	🟥 BEAM-3286
		w/ multiple output	🟩	🟩	🟩	🟩			🟩	🟩	🟩	🟩	🟩	🟩
		w/ user state	🟥 BEAM-2918/BEA	🟥 BEAM-2918/BEA	🟥 BEAM-2918/BEA	🟥 BEAM-2918/BEA	🟥 BEAM-2918/BEA	🟥 BEAM-2918/BEA	🟧 BEAM-2902/BEA	🟧 BEAM-2902/BEA	🟧 BEAM-2902/BEA	🟧 BEAM-2902/BEA	🟥 BEAM-2902/BEA	🟥 BEAM-2902/BEA
		w/ user timers					🟥	🟥					🟥	🟥
		w/ user metrics					🟨						🟩	🟨
	Flatten		🟩	🟩	🟩	🟩	🟨		🟩	🟩	🟩	🟩	🟩	🟨
		w/ explicit flatten	🟩	🟩	🟩	🟩	🟥 BEAM-3300	🟥 BEAM-3300	🟩	🟩	🟩	🟩	🟥 BEAM-3300	🟥 BEAM-3300
	Combine		🟩	🟩	🟩	🟩	🟩		🟩	🟩			🟩	🟨
		w/ first-class rep	🟩	🟩	🟩	🟩	🟥 BEAM-4276	🟥 BEAM-4276	🟥 BEAM-3513	🟥 BEAM-3513			🟥 BEAM-4276	🟥 BEAM-4276
		w/ lifting					🟥 BEAM-4276	🟥 BEAM-4276	🟥 BEAM-3711	🟥 BEAM-3711			🟥 BEAM-4276	🟥 BEAM-4276
	SDF						🟥 BEAM-3301	🟥 BEAM-3301					🟥 BEAM-3301	🟥 BEAM-3301
		w/ liquid sharding					🟥	🟥					🟥	🟥
	GBK		🟩	🟩	🟩	🟩	🟩		🟩	🟩			🟩	🟨
	CoGBK						🟨		🟩	🟩			🟩	🟨
	WindowInto		🟩	🟩	🟩	🟩			🟩	🟩	🟩	🟩	🟩	🟩
		w/ sessions	🟩	🟩	🟩	🟩	🟥 BEAM-4152	🟥 BEAM-4152					🟥 BEAM-4152	🟥 BEAM-4152
		w/ custom windowfn					🟥	🟥					🟥	🟥
EXAMPLE			Batch	Streaming	Batch	Streaming	Batch	Streaming	Batch	Streaming	Batch	Streaming	Batch	Streaming
	WordCap		🟩		🟩		🟩	▨	🟨		🟨		🟩	▨
	WordCount		🟩		🟩		🟩	▨	🟨		🟨		🟦	▨
		w/ write to Sink	🟩		🟩			▨					🟩	▨
		w/ write to GCS	🟩		🟩		🟥	▨					🟩	▨

https://s.apache.org/apache-beam-portability-support-table

Demo

The Future

Future work

- Support for user state and timers
- Mixing and matching connectors written in different languages
- Wait for new SDKs in other languages, they will just work 😜
- Unified batch and streaming API in Flink?
  - currently 4 Flink translators (batch + streaming for each, portable and old, Java-only runner)
- Beam Flink Runner compatibility story
  - Flink upgrades
  - Pipeline upgrades / state migration

Streaming@Lyft

We are hiring! **lyft.com/careers**

https://goo.gl/RsyLkS

WE ARE HIRING

data-artisans.com/careers

dataArtisans

Learn More!

Beam Portability Framework

https://beam.apache.org/contribute/portability/
https://beam.apache.org/contribute/design-documents/#portability

Apache Beam

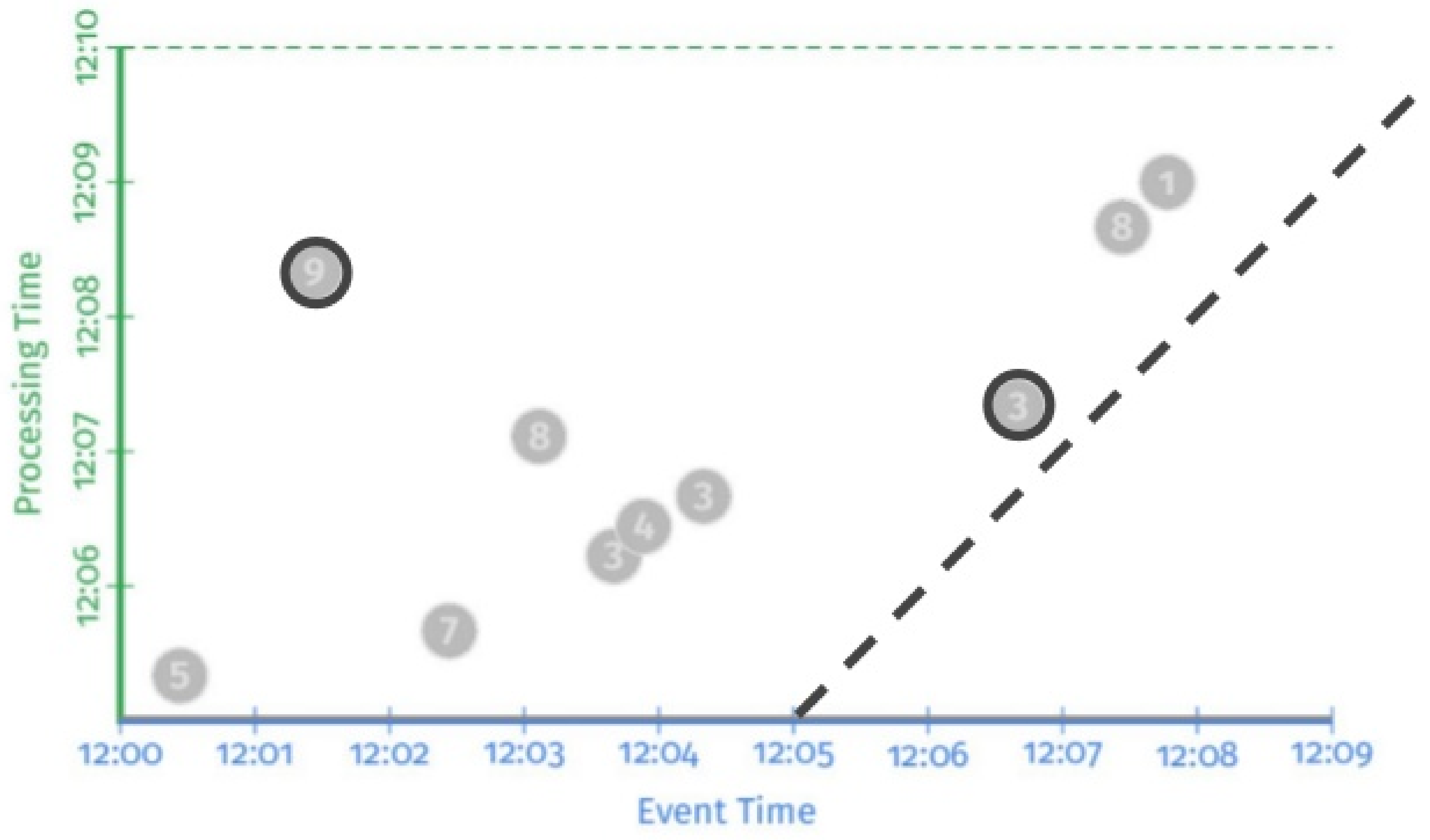
https://beam.apache.org
https://s.apache.org/beam-slack-channel **#beam #beam-portability**
https://beam.apache.org/community/contact-us/

Follow @ApacheBeam on Twitter

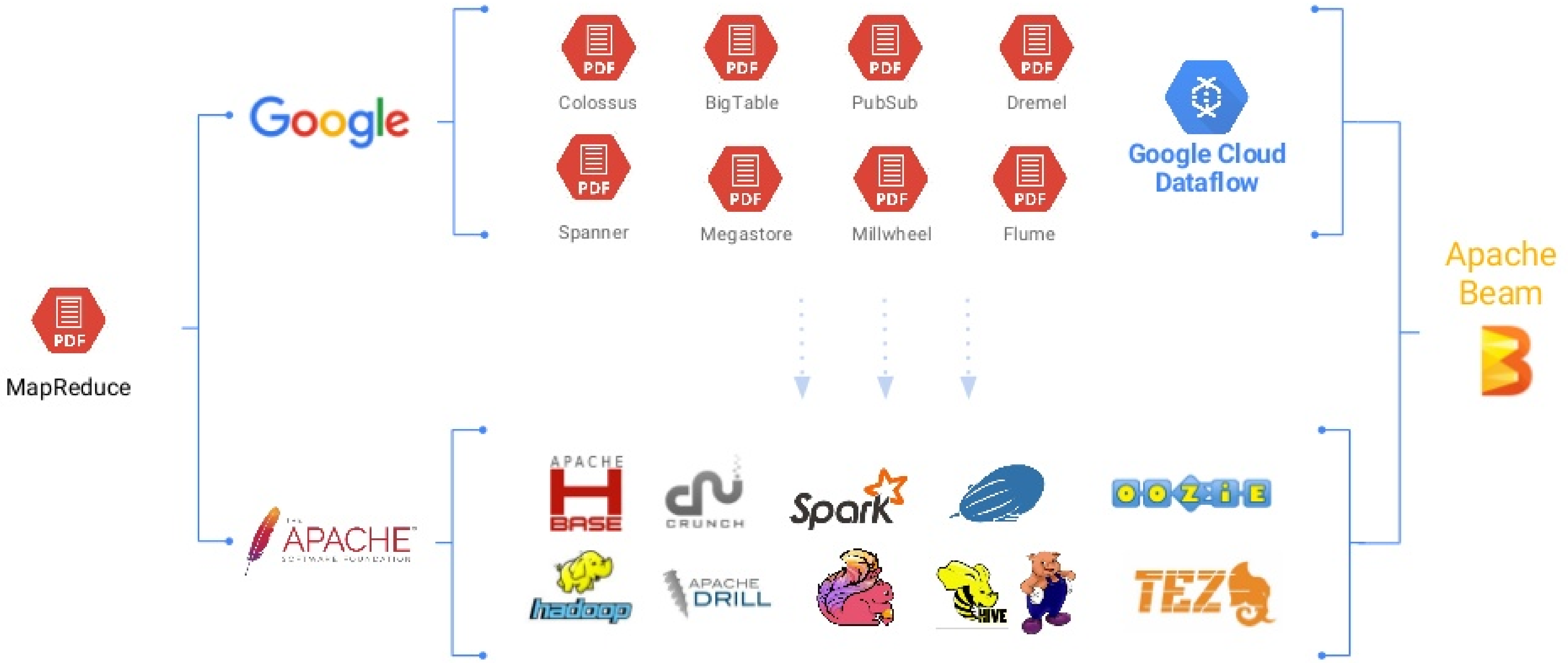
Thank you!

Backup Slides

Processing Time vs. Event Time



The Origin of Apache Beam

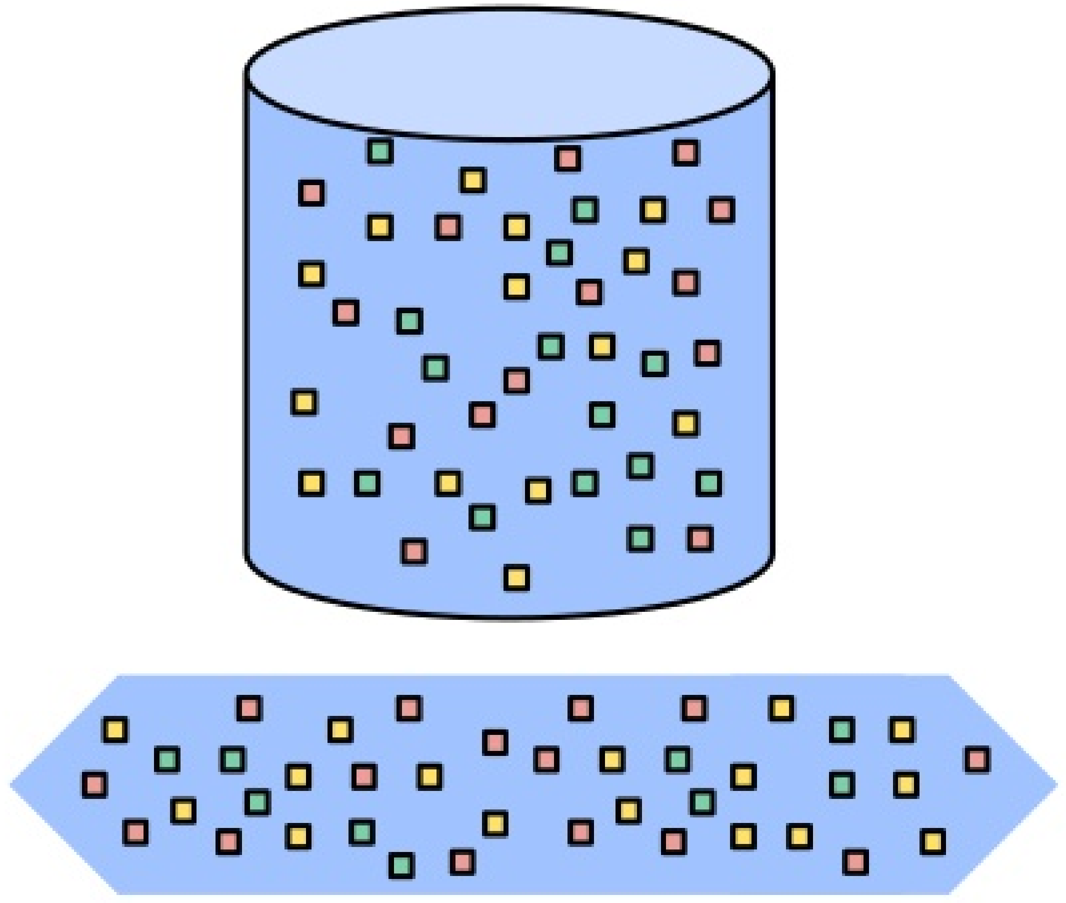


Beam Model: Generations Beyond MapReduce

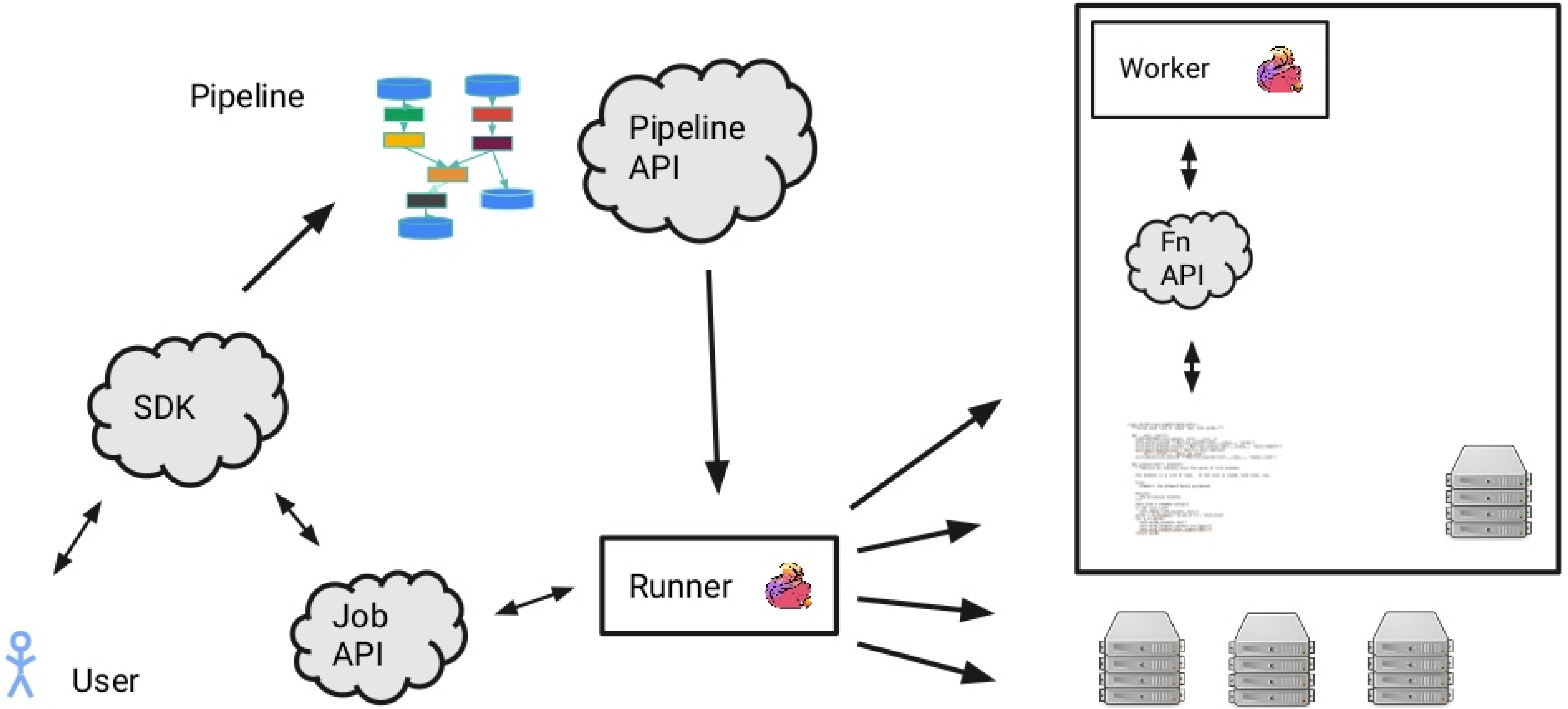
Improved abstractions let you focus on your application logic

Batch and stream processing are *both* first-class citizens -- no need to choose.

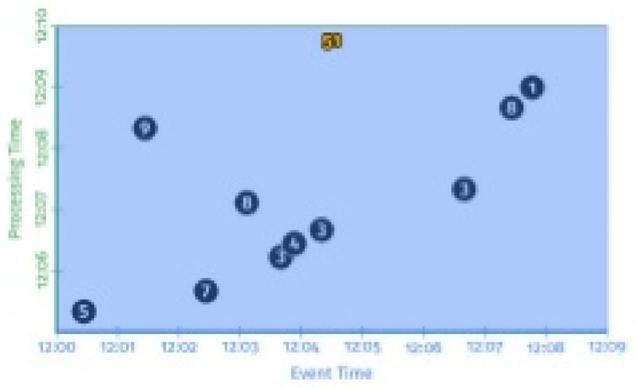
Clearly separates event time from processing time.



Executing a Beam Pipeline - The Big Picture

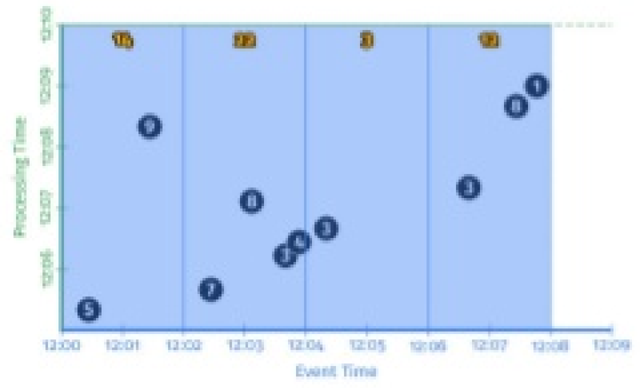


Customizing What Where When How



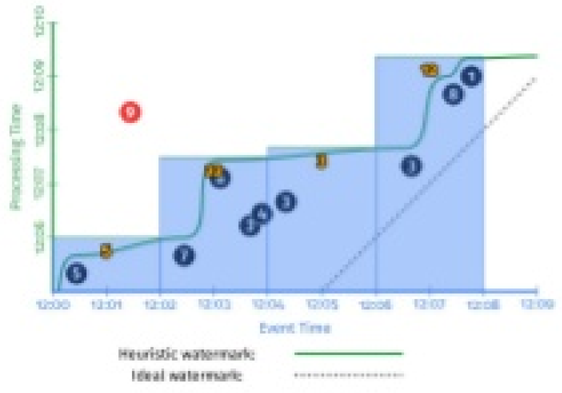
1

Classic Batch



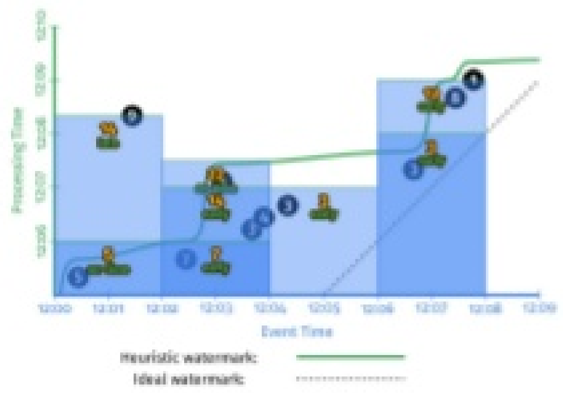
2

Windowed Batch



3

Streaming



4

Streaming + Accumulation

For more information see https://cloud.google.com/dataflow/examples/gaming-example

Terminology

Beam Model
Describes the API concepts and the possible operations on PCollections.

Pipeline
User-defined graph of transformations on PCollections. This is constructed using a Beam SDK. The transformations can contain UDFs.

Runner
Executes a Pipeline. For example: *FlinkRunner*.

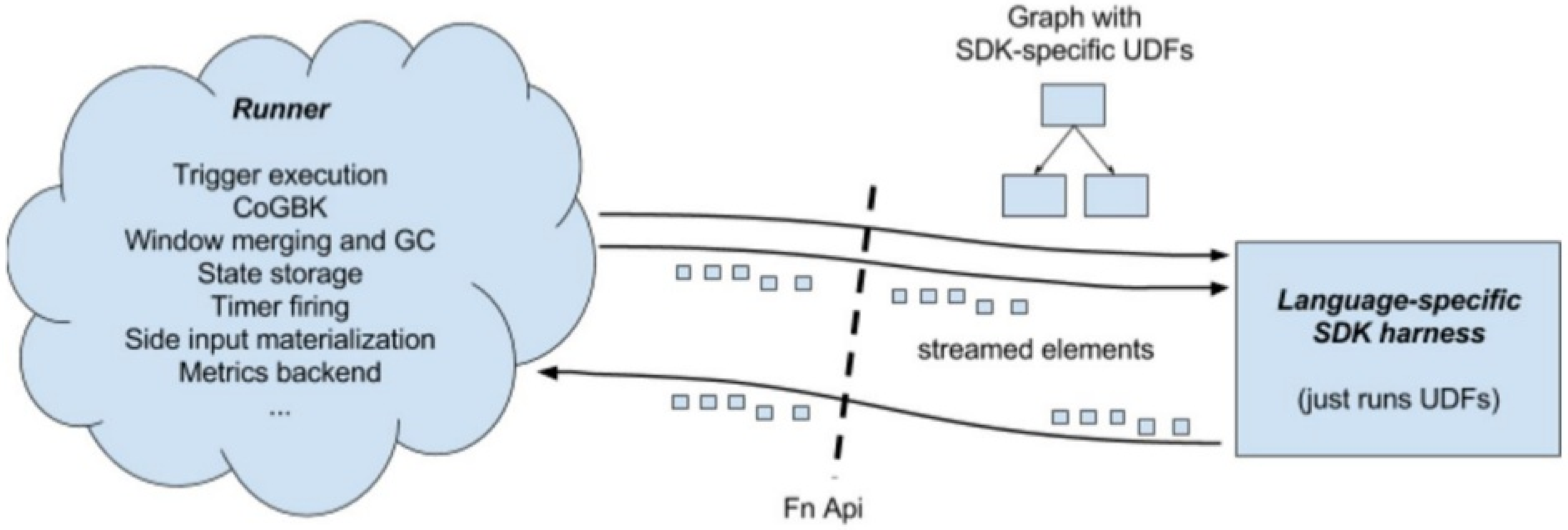
Beam SDK
Language specific library/framework for creating programs that use the Beam Model. Allows defining Pipelines and UDFs and provides APIs for executing them.

User-defined function (UDF)
Code in Java, Python, Go, … that specifies how data is transformed. For example *DoFn* or *CombineFn*.

Job API

```
public interface JobApi {
  State getState(); // RUNNING, DONE, CANCELED, FAILED ...
  State cancel() throws IOException;
  State waitUntilFinish(Duration duration);
  State waitUntilFinish();
  MetricResults metrics();
}
```

Fn API (continued)



https://s.apache.org/beam-fn-api