



Testing / CI /CD

Lorenzo Caggioni

Strategic Cloud Engineer, Google Cloud PSO



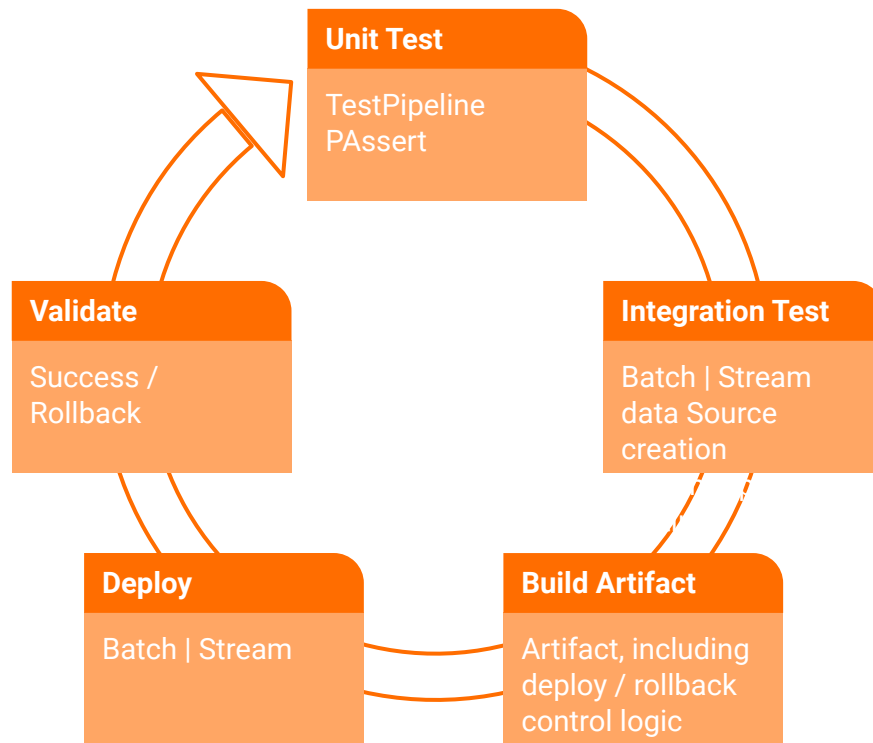
Testing / CI / CD



Lorenzo Caggioni

Strategic Cloud Engineering, Google Cloud PSO

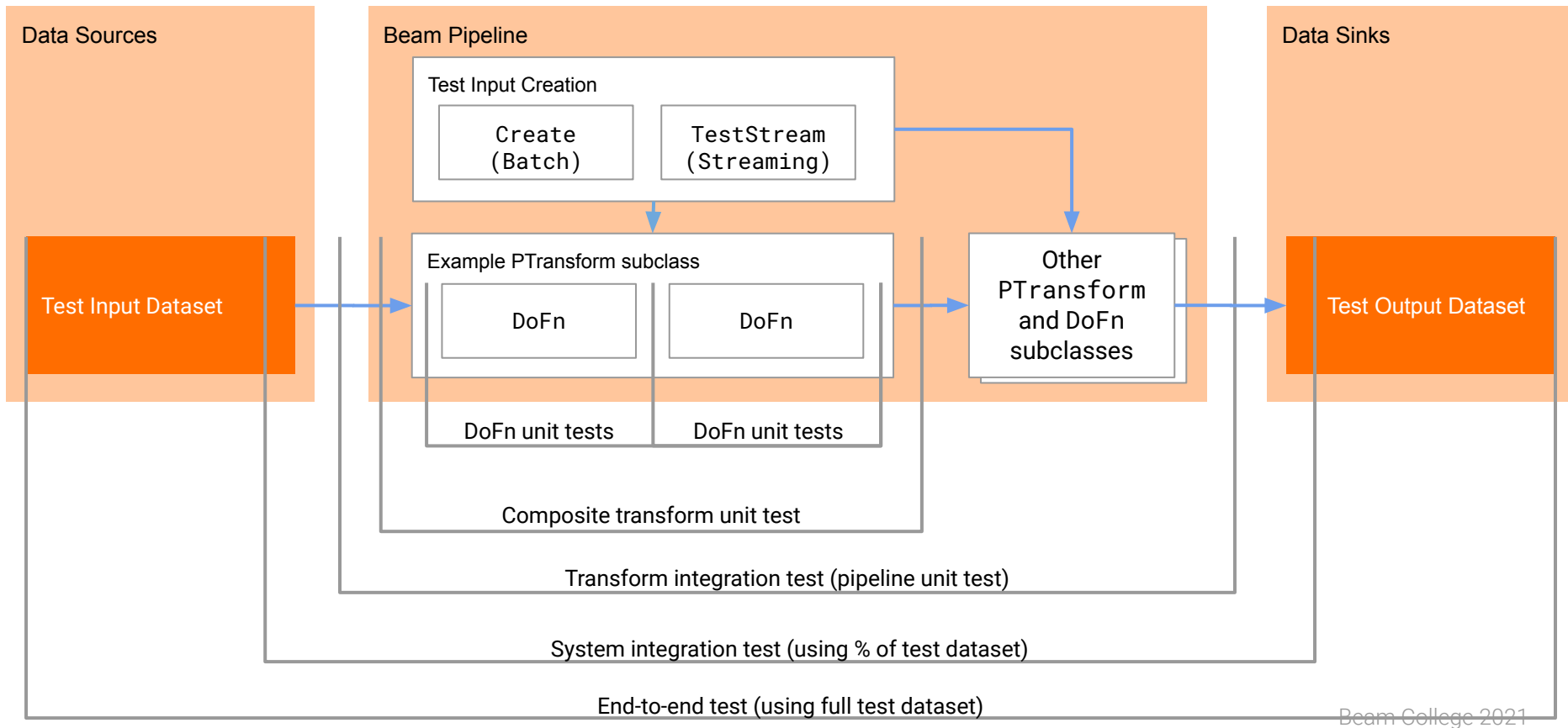
CI / CD with Apache Beam



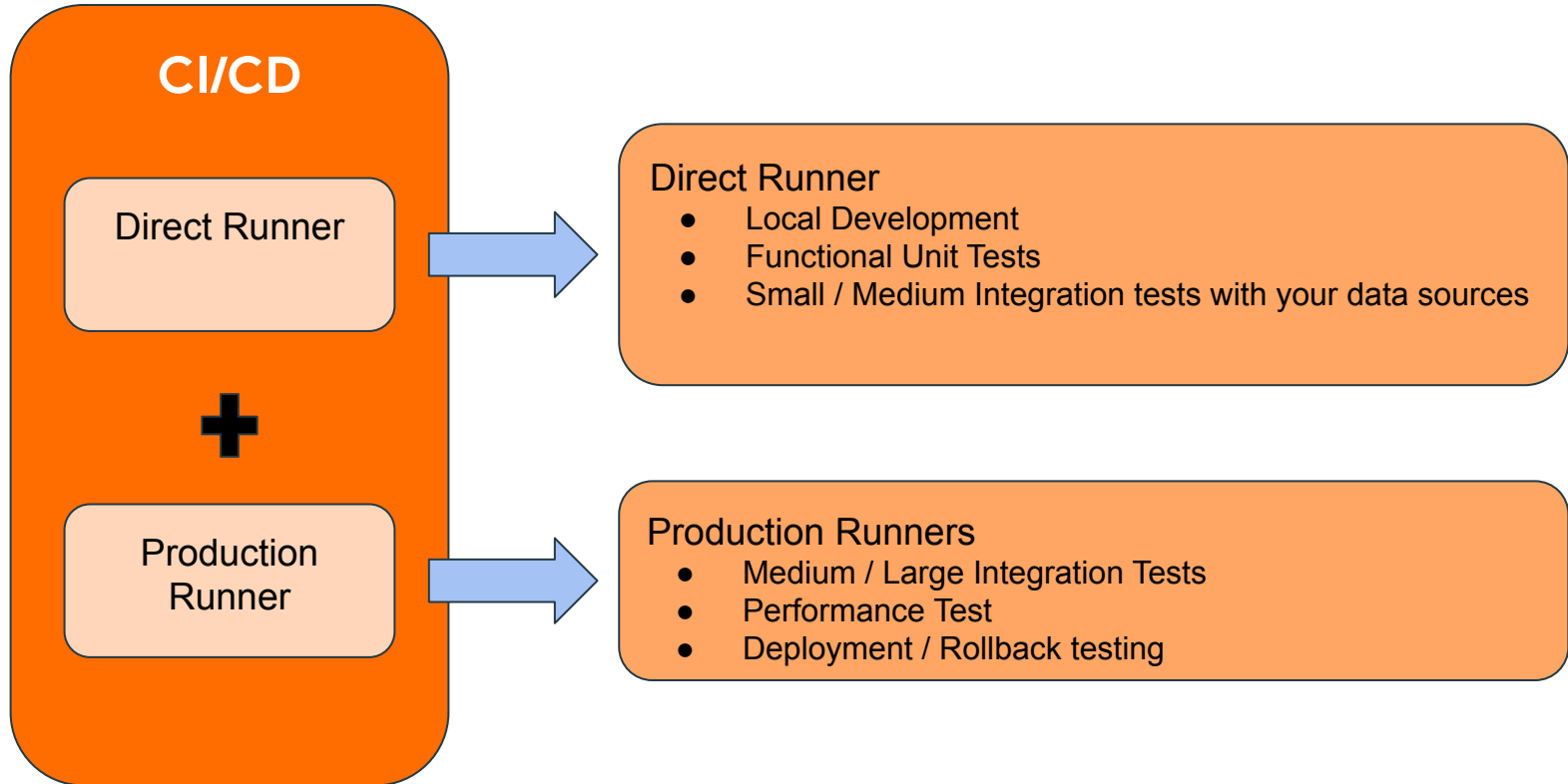
Testing in Beam Overview



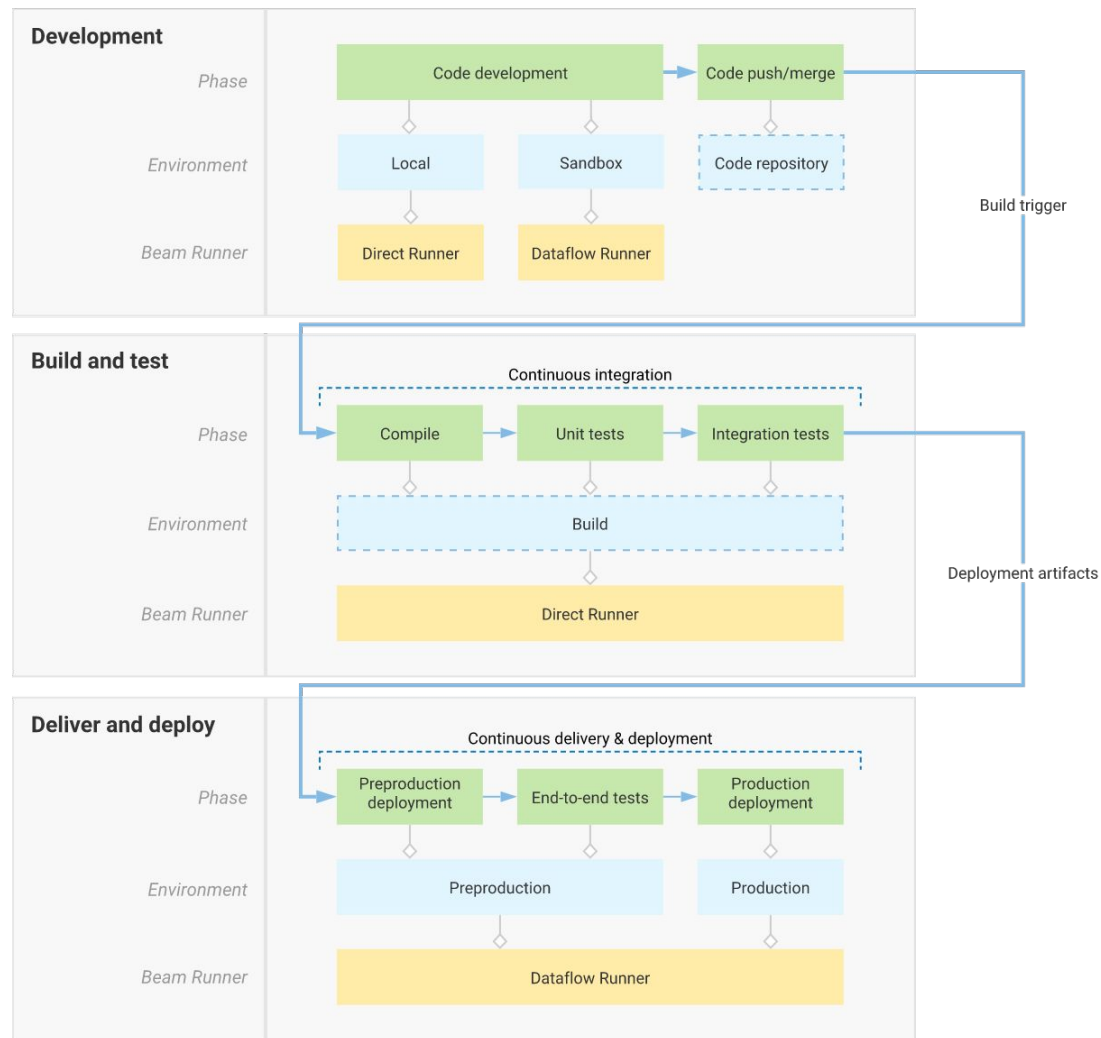
Beam Testing Overview



Testing Environment



CI/CD Lifecycle



Unit Testing



Unit Tests in Beam: PTransforms

Unit tests in Beam are mini-pipelines that **assert behavior** of one small testable piece of your production pipeline e.g. a single [PTransform](#) or [DoFn](#).

This should run quickly, locally, and without dependencies on external systems.

```
@Rule
public TestPipeline p = TestPipeline.create();

@Test
public void testASingleTranform() {
    // Setup your PCollection from an in-memory or local data source.
    ...
    // Apply your transform.
    PCollection<String> words = lines.apply(ParDo.of(new WordCount.ExtractWordsFn()));

    // Setup assertions on the pipeline.
    ...
    p.run();
}
```

Setup

```
<hamcrest.version>2.1</hamcrest.version>  
<junit.version>4.13-beta-3</junit.version>
```

...

```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>${junit.version}</version>  
</dependency>
```

JUnit 4 is required to use Beam testing utilities.

See issue [BEAM-3415](#) for JUnit 5 support.

<- pom.xml snippet taken from Word Count Beam archetype

Testing Classes

TestPipeline

TestPipeline is a class included in the Beam Java SDK specifically for testing transforms. For tests, use TestPipeline in place of Pipeline when you create the pipeline object.

Unlike Pipeline.create, TestPipeline.create handles setting PipelineOptions internally.

Java

```
@Rule
public final transient TestPipeline p =
    TestPipeline.create();
```

Python

```
with TestPipeline() as p:
```

Testing Classes

PAssert

An assertion on the contents of a PCollection incorporated into the pipeline. Such an assertion can be checked no matter what kind of PipelineRunner is used.

PAssert becomes part of your pipeline, and works on both local and production runners.

Java

```
@Test
@Category(NeedsRunner.class)
public void myPipelineTest() throws Exception {
    final PCollection<String> pcol = p.apply(...)
    PAssert.that(pcol).containsInAnyOrder(...);
    p.run();
}
```

Python

```
# Run our Transform (example: CountWords()).
output = input | CountWords();

# Assert PCollection output
assert_that(output, equal_to(EXPECTED_OUTPUT))
```

Beam Unit Testing Basics

```
@Rule
public final transient TestPipeline p =
    TestPipeline.create();

@Test
@Category(NeedsRunner.class)
public void myPipelineTest() throws Exception {
    final PCollection<String> pcol = p.apply(...)
    Assert.that(pcol).containsInAnyOrder(...);
    p.run();
}
```

Anti-Pattern: Anonymous Internal Classes

```
PipelineOptions options = PipelineOptionsFactory.create();
```

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

```
PCollection<Integer> output =  
    p.apply("Read from text", TextIO.Read.from(...))  
      .apply("Split words", ParDo.of(new DoFn() {  
          // Untestable anonymous transform 1  
      })))  
      .apply("Generate anagrams", ParDo.of(new DoFn() {  
          // Untestable anonymous transform 2  
      })))  
      .apply("Count words", Count.perElement());
```

Best Practice:

Use Named DoFn subclasses

```
PipelineOptions options = PipelineOptionsFactory.create();
```

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

```
PCollection<Integer> output =  
    p.apply("Read from text", TextIO.Read.from(...))  
      .apply("Split words", ParDo.of(new SplitIntoWordsFn()))  
      .apply("Generate anagrams", ParDo.of(new GenerateAnagramsFn()))  
      .apply("Count words", Count.perElement());
```

Tests for our named subclasses

```
@Rule
public final transient TestPipeline p = TestPipeline.create();

@Test
@Category(NeedsRunner.class)
public void testGenerateAnagramsFn() {
    // Create the test input
    PCollection<String> words = p.apply(Create.of("friend"));
    // Test a single DoFn using the test input
    PCollection<String> anagrams = words.apply("Generate anagrams", ParDo.of(new GenerateAnagramsFn()));
    // Assert correct output from
    PAssert.that(anagrams).containsInAnyOrder("finder", "friend", "redfin", "refind");

    p.run();
}
```


Tests for our named subclasses - Python

...

```
class CountTest(unittest.TestCase):

    def test_count(self):
        # Our input data, which will make up the initial PCollection.
        WORDS = ["hi", "there", "hi", "sue"]

        # Expected data that the final PCollection must match.
        EXPECTED_OUTPUT = [("hi", 2), ("there", 1), ("sue", 1)]

        # Example test that tests the pipeline's transforms.
        with TestPipeline() as p:

            # Create a PCollection from the WORDS static input data.
            input = p | beam.Create(WORDS);

            # Run our Transform (example: CountWords()).
            output = input | CountWords();

            # Assert that the output PCollection matches the EXPECTED_COUNTS data.
            assert_that(output, equal_to(EXPECTED_OUTPUT))
```

Testing Windowing Behavior

```
@Test
@Category(NeedsRunner.class)
public void testWindowedData() {
    PCollection<String> input =
        p.apply(Create.timestamped(
            TimestampedValue.of("a", new Instant(0L)),
            TimestampedValue.of("a", new Instant(0L)),
            TimestampedValue.of("b", new Instant(0L)),
            TimestampedValue.of("c", new Instant(0L)),
            TimestampedValue.of("c", new Instant(0L))
                .plus(WINDOW_DURATION)))
        .withCoder(StringUtf8Coder.of());
}
```

```
PCollection<KV<String, Long>> windowedCount =
    input
        .apply(Window.into(FixedWindows.of(WINDOW_DURATION)))
        .apply(Count.perElement());

PAssert.that(windowedCount)
    .containsInAnyOrder(
        // Output from first window
        KV.of("a", 2L),
        KV.of("b", 1L),
        KV.of("c", 1L),
        // Output from second window
        KV.of("c", 1L));

p.run();
}
```

Integration Testing



Beam Integration Test Basics

```
private class WeatherStatsPipeline extends
    PTransform<PCollection<Integer>, PCollection<WeatherSummary>> {
    @Override
    public PCollection<WeatherSummary> expand(PCollection<Integer> input)
    {
        // Pipeline transforms ...
    }
}

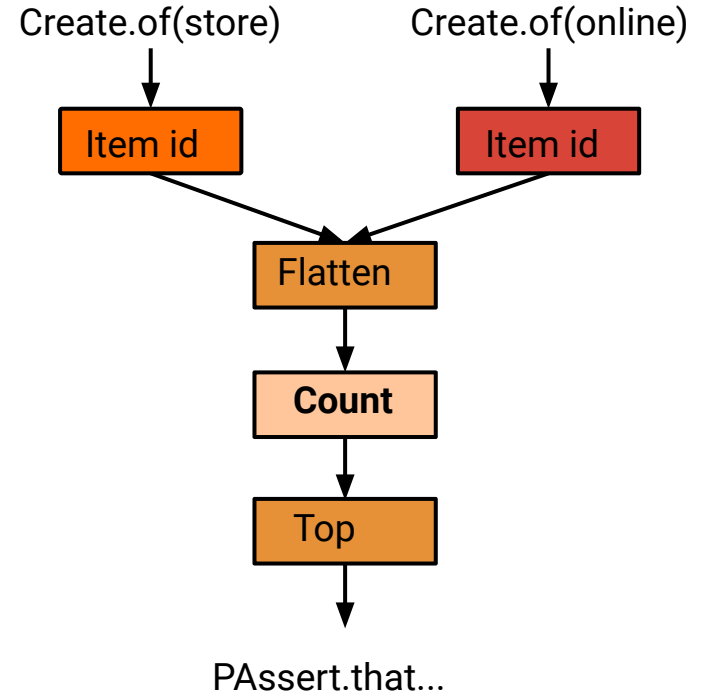
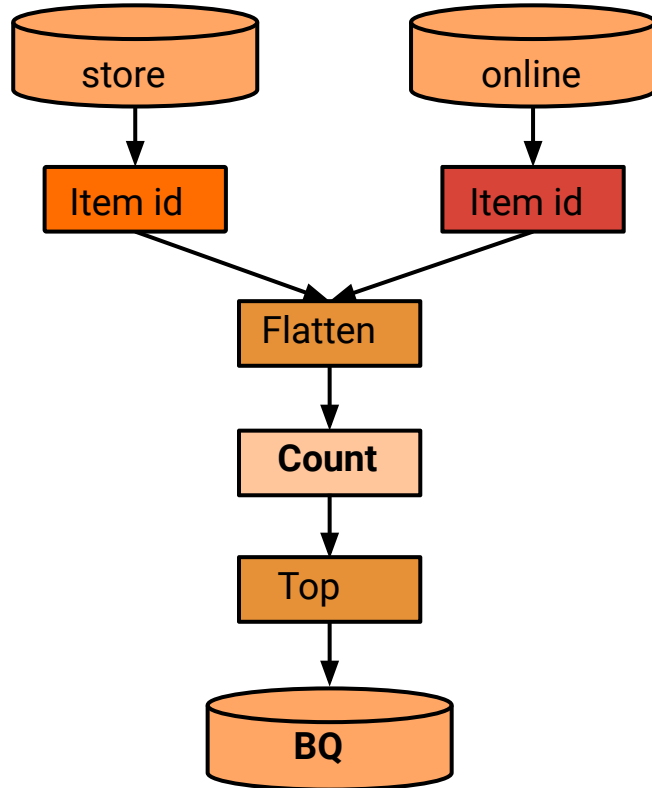
@Rule
public final transient TestPipeline p = TestPipeline.create();
```

```
@Test
@Category(NeedsRunner.class)
public void testWeatherPipeline() {
    // Create test input consisting of temperature readings
    PCollection<Integer> tempCelsius =
        p.apply(Create.of(24, 22, 20, 22, 21, 21, 20));

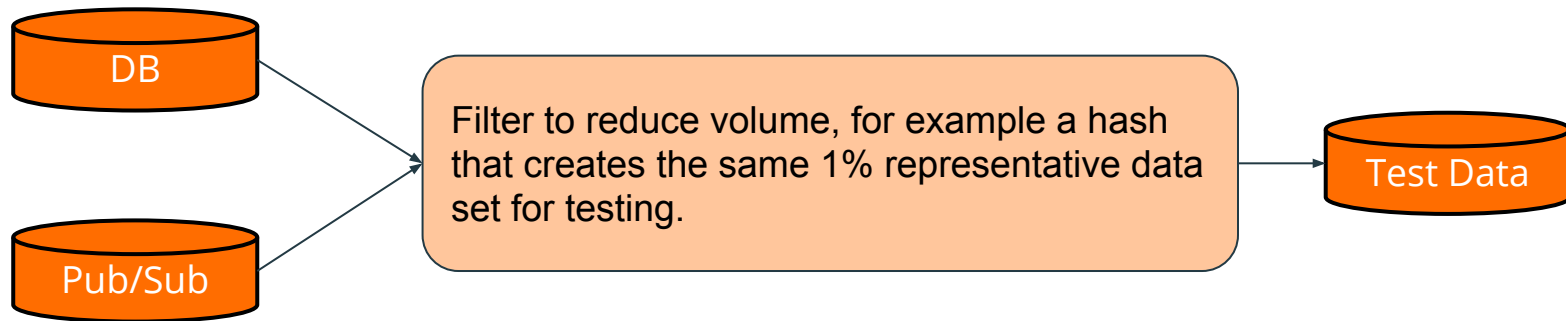
    // calculates the min, max, and average temperature
    PCollection<WeatherSummary> result =
        tempCelsius.apply("Calculate weather statistics",
            new WeatherStatsPipeline());

    // Assert correct output from CalculateWeatherStats
    PAssert.thatSingleton(result).isEqualTo(new WeatherSummary.Builder()
        .withAverageTemp(21)
        .withMaxTemp(24)
        .withMinTemp(20)
        .build());
    p.run();
}
```

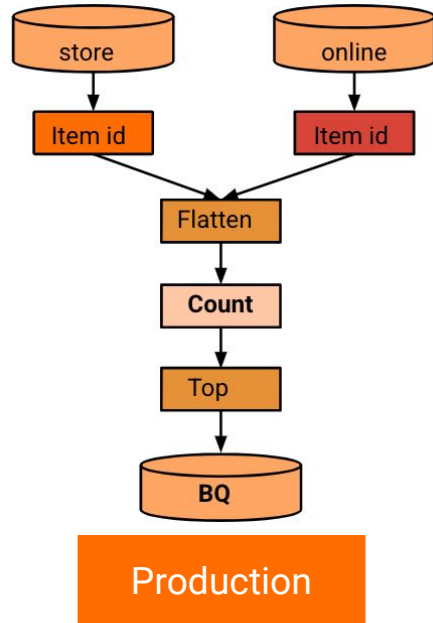
Testing Pipelines End-To-End (Small Integration)



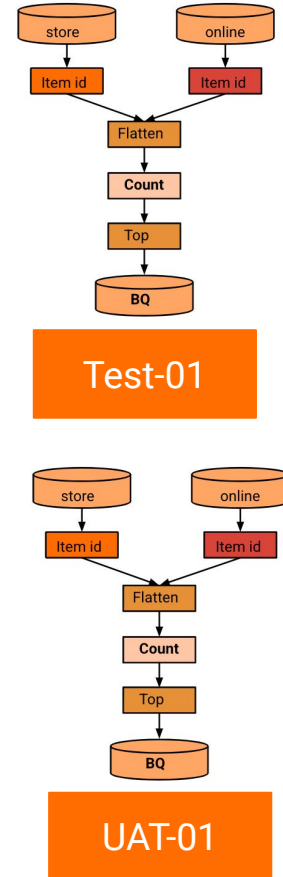
Testing Pipelines End-To-End (Medium Integration)



Testing Pipelines End-To-End (Large Integration)



Clone



Large Integration testing - Batch vs Stream

For large scale integration testing will diverge for batch vs stream pipelines.

Batch Pipelines

After medium integration tests, you can choose to move onto the validation stage. Deployment of batch pipelines is only necessary if you are using runner specific mechanisms like templates.

Stream Pipelines

The deployment process for streaming pipelines needs to be determined during integration testing and then validated.

Integration testing - Deployment exploration

Part of the integration testing role will be to establish the deployment path, which will become part of the final artifact.

Batch vs Stream

There are very different paths to deployment for batch vs streaming pipelines.

- With Batch we have choices around the use of templates vs direct running of the pipeline.
- With Streaming we need to explore the method to replace the existing pipeline, unless this is the first time the pipeline is deployed.

Testing streaming pipelines



Testing Classes

TestStream

A testing input that generates an unbounded PCollection of elements, advancing the watermark and processing time as elements are emitted. After all of the specified elements are emitted, ceases to produce output.

Each call to a TestStream.Builder method will only be reflected in the state of the Pipeline after each method before it has completed and no more progress can be made by the Pipeline. A PipelineRunner must ensure that no more progress can be made in the Pipeline before advancing the state of the TestStream.

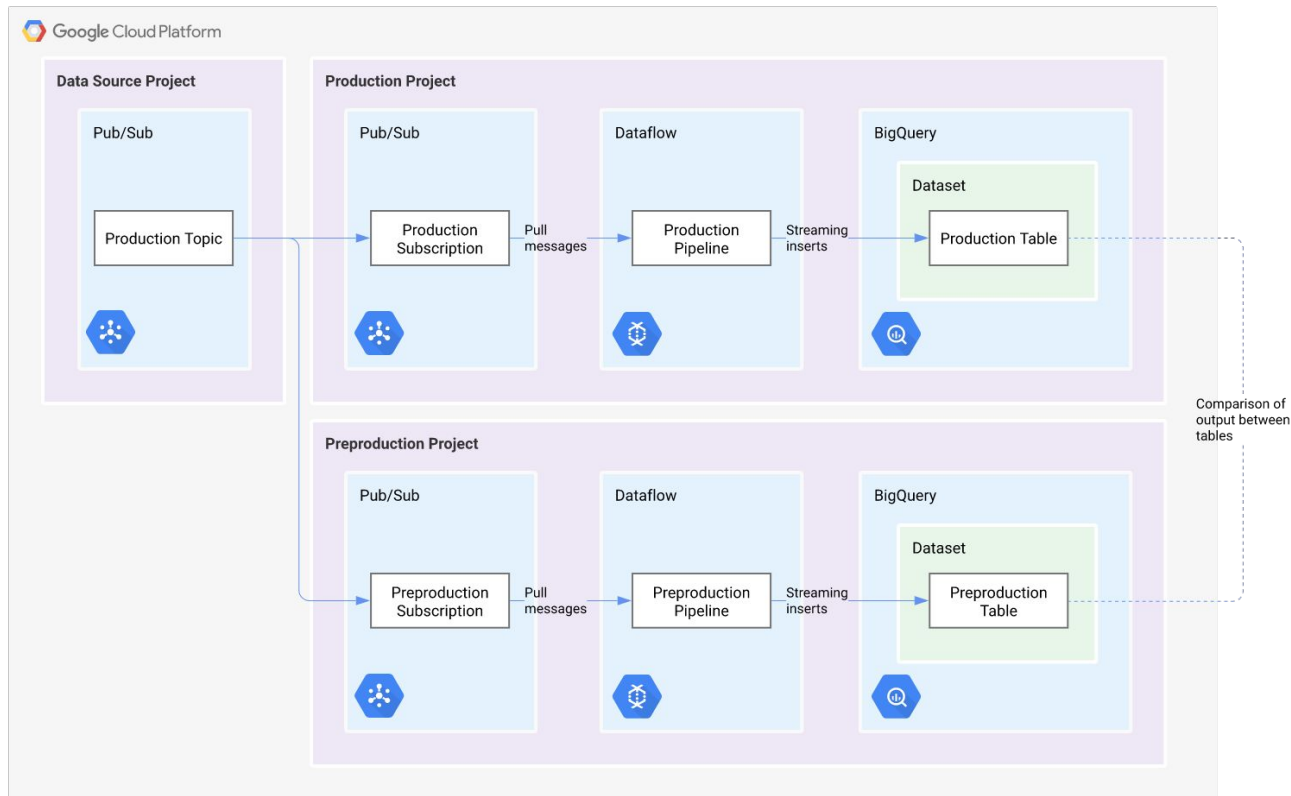
Streaming Pipeline Testing Basics

For streaming pipeline tests, use `TestStream` to create a pipeline object that enables you to model the effect of element timings.

Note, `TestStream` is not supported by the Dataflow Runner.

```
@Test
@Category(NeedsRunner.class)
public void testDroppedLateData() {
    TestStream<String> input = TestStream.create(StringUtf8Coder.of())
        // Add elements arriving before the watermark
        .addElements(
            TimestampedValue.of("a", new Instant(0L)),
            TimestampedValue.of("a", new Instant(0L)),
            TimestampedValue.of("b", new Instant(0L)),
            TimestampedValue.of("c", new Instant(0L).plus(Duration.standardMinutes(3))))
    // Advance the watermark past the end of the window
    .advanceWatermarkTo(new Instant(0L).plus(WINDOW_DURATION)
        .plus(Duration.standardMinutes(1)))
    // Add elements which will be dropped due to lateness
    .addElements(TimestampedValue.of("c", new Instant(0L)))
    // Advance the watermark to infinity which will close all windows
    .advanceWatermarkToInfinity();
    PCollection<KV<String, Long>> windowedCount = ...
    PAssert.that(windowedCount)
        .containsInAnyOrder(KV.of("a", 2L), KV.of("b", 1L), KV.of("c", 1L));
    p.run();
}
```

Testing Pipelines End-To-End (Large Integration)



Streaming Pipeline Deployment - Options

High level, detail on following slides;

Update

- When you update a job on the Cloud Dataflow service, you **replace** the existing job with a new job that runs your updated pipeline code. The Cloud Dataflow service retains the job name, but runs the replacement job with an updated job id.


Drain

- Using the Drain option to stop your job tells the Cloud Dataflow service to finish your job in its current state.

Cancel

- Using the Cancel option to stop your job tells the Cloud Dataflow service to cancel your job immediately.



Streaming Pipeline Deployment - Update

Name	Type	End time	Elapsed time	Start time	Status	SDK version
 <u>processorders</u>	Streaming	—	10 min 49 sec	May 2, 2019, 10:00:15 AM	Running	2.11.0





Job status	Updating...
SDK version	Apache Beam SDK for Java 2.11.0



Name	Type	End time	Elapsed time	Start time	Status	SDK version
 <u>processorders</u>	Streaming	—	56 sec	May 2, 2019, 10:15:22 AM	Not started	2.11.0
 <u>processorders</u>	Streaming	—	16 min 3 sec	May 2, 2019, 10:00:15 AM	Running	2.11.0



Name	Type	End time	Elapsed time	Start time	Status	SDK version
 <u>processorders</u>	Streaming	—	3 min 31 sec	May 2, 2019, 10:15:22 AM	Running	2.11.0
 <u>processorders</u>	Streaming	May 2, 2019, 10:18:40 AM	18 min 25 sec	May 2, 2019, 10:00:15 AM	Updated	2.11.0

Streaming Pipeline Deployment - Update

There are various reasons why you might want to update your existing Cloud Dataflow job:

- You want to enhance or otherwise improve your pipeline code.
- You want to fix bugs in your pipeline code.
- You want to update your pipeline to handle changes in data format, or to account for version or other changes in your data source.

Streaming Pipeline Deployment - Update

Steps

1. `--update` & `--jobName` used in `PipelineOptions` to trigger an update.
2. Compatibility check is run, if transform names have changed you must provide `--transformNameMapping`
3. Replacement job preserves any intermediate state data from the prior job. "In-flight" data will still be processed by the transforms in your new pipeline. Additional transforms that you add in your replacement pipeline code may or may not take effect, depending on where the records are buffered.

Changing Windows

Changing the windowing or trigger strategies will not affect data that is already buffered or otherwise in-flight.

Streaming Pipeline Deployment - Drain

Effects of a Drain

- Drain will stop pulling data from source.
- The watermark will be moved to ∞ and all windows will be closed.
- This will result in incomplete aggregations;
 - If you have non idempotent side effects your external systems will need correction after the drain operation.

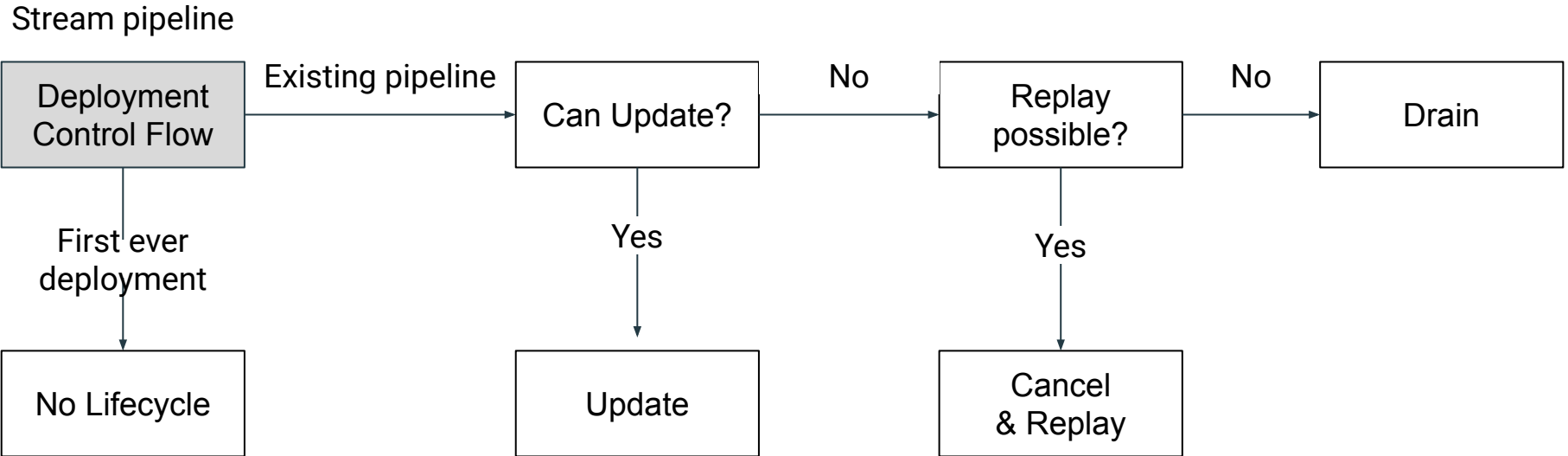
Note: PanelInfo is also provided, allowing you to detect the incomplete aggregations and write them elsewhere.

Streaming Pipeline Deployment - Cancel


Effects of a Cancel

- The service will halt all data ingestion and processing as soon as possible and immediately begin cleaning up the Google Cloud Platform (GCP) resources attached to your job. These resources may include shutting down Compute Engine worker instances and closing active connections to I/O sources or sinks.
- Because Cancel immediately halts processing, you may lose any "in-flight" data. "In-flight" data refers to data that has been read but is still being processed by your pipeline. Data written from your pipeline to an output sink before you issued the Cancel command may still be accessible on your output sink

Integration testing - Deployment exploration



Lifecycle testing


Name	Type	End time	Elapsed time	Start time	Status	SDK version
 processorders	Streaming	—	10 min 49 sec	May 2, 2019, 10:00:15 AM	Running	2.11.0



Job status

Updating...

SDK version

Apache Beam SDK for Java 2.11.0

Name	Type	End time	Elapsed time	Start time	Status	SDK version
 processorders	Streaming	—	56 sec	May 2, 2019, 10:15:22 AM	Not started	2.11.0
 processorders	Streaming	—	16 min 3 sec	May 2, 2019, 10:00:15 AM	Running	2.11.0

Name	Type	End time	Elapsed time	Start time	Status	SDK version
 processorders	Streaming	—	3 min 31 sec	May 2, 2019, 10:15:22 AM	Running	2.11.0
 processorders	Streaming	May 2, 2019, 10:18:40 AM	18 min 25 sec	May 2, 2019, 10:00:15 AM	Updated	2.11.0

Stop job

☐ Cancel

Dataflow will immediately stop this job and abort all data ingestion and processing. Any buffered data may be lost.

☐ Drain

Dataflow will cease all data ingestion, but will attempt to finish processing any remaining buffered data. Pipeline resources will be maintained until buffered data has finished processing and any pending output has finished writing.

[Read more about stopping Dataflow jobs](#)

DO NOTHING

STOP JOB

Artifact Building



Apache Beam SDK versions

Apache Beam uses semantic versioning.

Version numbers use the form major.minor.incremental and are incremented as follows:

- major version for incompatible API changes.
- minor version for new functionality added in a backward-compatible manner.
- incremental version for forward-compatible bug fixes.

<https://beam.apache.org/get-started/downloads/#releases>

@Experimental ... frankly is overused and due a clean up by the community.....

Dataflow Runner features can be dependent on the version of the SDK used.

Details can be found at:

<https://cloud.google.com/dataflow/docs/resources/release-notes-service>

Apache Beam SDK - Central Repos

Maven:

Note you will often need to pull in more than the core.

```
<groupId>org.apache.beam</groupId>  
<artifactId>beam-sdks-java-core</artifactId>
```

Provides the core Java sdk, but will not include runners, for Dataflow include:

```
<groupId>org.apache.beam</groupId>  
<artifactId>beam-runners-google-cloud-dataflow-java</artifactId>
```

For IO include:

```
<groupId>org.apache.beam</groupId>  
<artifactId>beam-sdks-java-io-google-cloud-platform</artifactId>
```


Deployment



Deployment - Batch

Templates

- If using templates as the triggering mechanism for your Batch pipelines, the final stage is to run the pipeline using: `--templateLocation=gs//`

Other schedulers / orchestration

- Provide the runnable jar which your system can run with PipelineOptions for variables that need to be supplied on the command line.

Initial deployment

- Provide unique name for your pipeline; This name will be used by the monitoring tools when you go to the Cloud Dataflow page (or anywhere else names are visible, like in the gsutil commands). It is also has to be unique within the project. This is a great safety feature, as it prevents two copies of the same pipeline from accidentally being started.

Deployment - Streaming

Initial deployment

- As per batch pipelines provide unique name for your pipeline;

Create backups of your pipelines

- Some sources, like Cloud Pub/Sub and Apache Kafka, let you replay a stream from a specific point in processing time. This feature (when available) lets you create a backup of the stream data for reuse if required.

Create multiple replica pipelines

- Pro tip from Google SRE; they will often create replica of the prod env and if anything goes wrong roll back to that instance.

Validation



Validating Pipeline Health

Monitor and perform rollback logic if necessary.

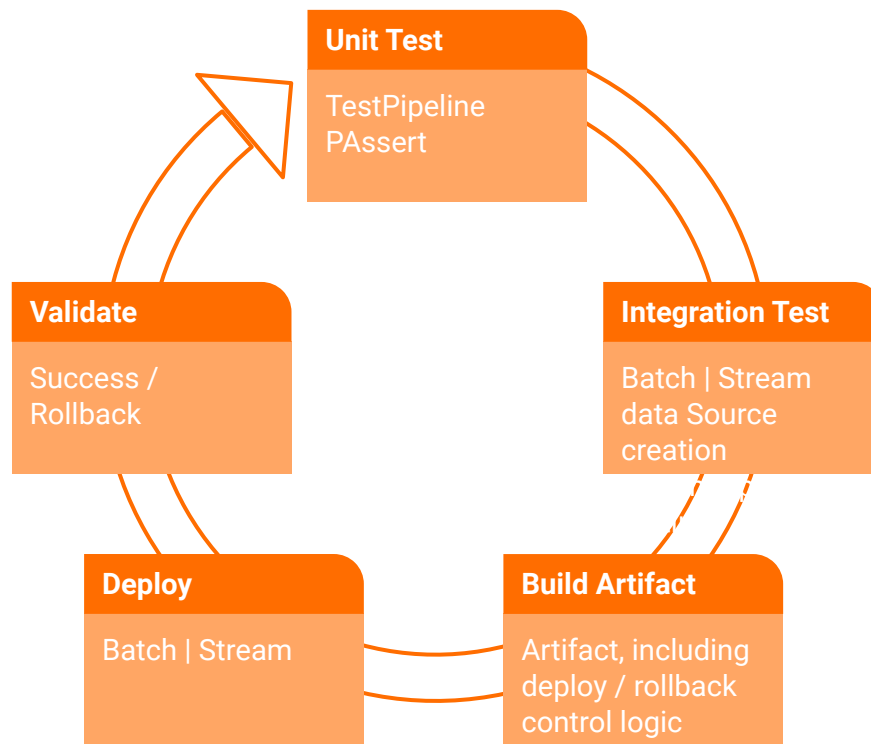
Batch

- Pipelines will fail automatically if there are too many processing failures.
- Still important to monitor message processing rates in case the pipeline is running abnormally slowly.

Streaming

- Monitoring message processing rates, watermark advancement, and system lag.

Summary



Thanks!

:wq!

