Serverless data processing. **Dataflow**

Javier Briones
GFT

15 Enero 2022

EDEM
Escuela de Empresarios

GFT

About me

Data Engineer at GFT focus on BigData and IoT architectures on Google Cloud Platform.

Javier Briones
Data Engineer at GFT

Session objectives

Keep it simple

Do it, know it

Real-world cases

Agenda

01 Introduction to Serverless computing

Evolution of computation
Serverless. Basics
Serverless. GCP

02 Serverless data processing. Dataflow

Dataflow. Basics
Apache Beam programming model
How to use Dataflow?

03 Success stories. GFT

04 Hands-on. Demo

Setup Requirements

Google Cloud Platform - Free trial

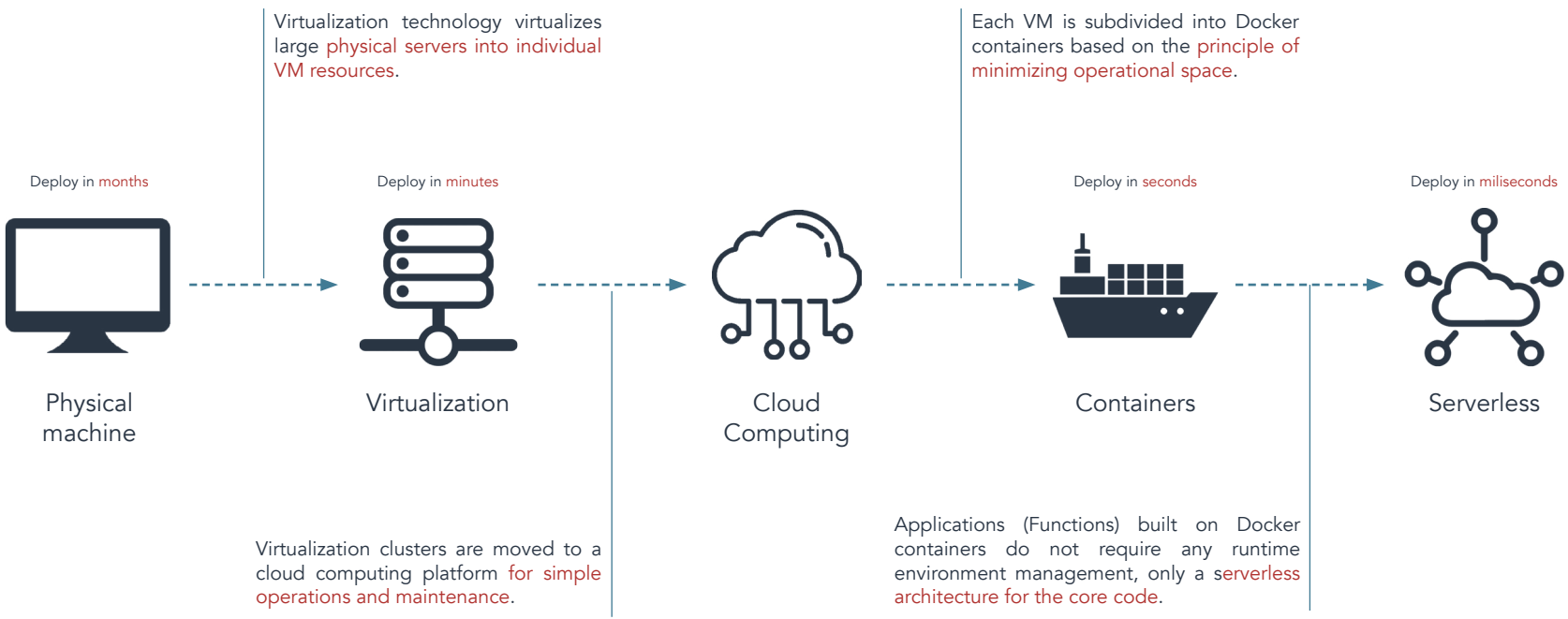
https://console.cloud.google.com/freetrial

GitHub repository

https://github.com/jabrio/Serverless_EDEM

Introduction to Serverless Computation 01

Evolution of computation



...What is Serverless?

...What is Serverless?

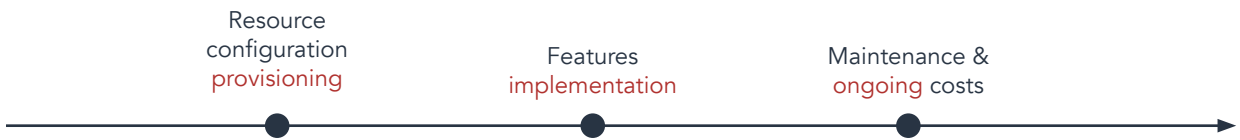
Definition

Serverless = Do not think about where your code runs.

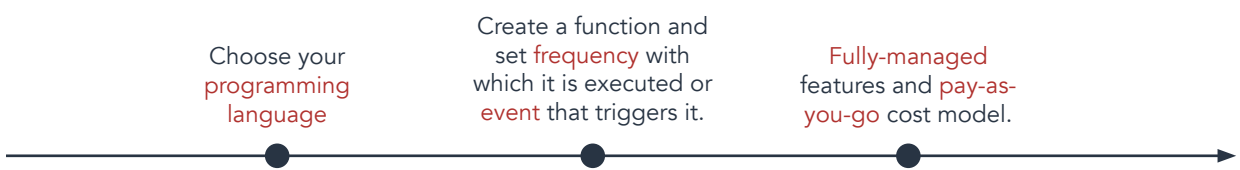
Serverless architecture runs code, manages data and integrates applications without managing servers. This kind of technology includes auto-scaling, high availability and a pay-as-you-go cost model, as well as eliminating infrastructure management tasks such as resource provisioning or security patches.

...Why Serverless?

Traditional vs Serverless architecture



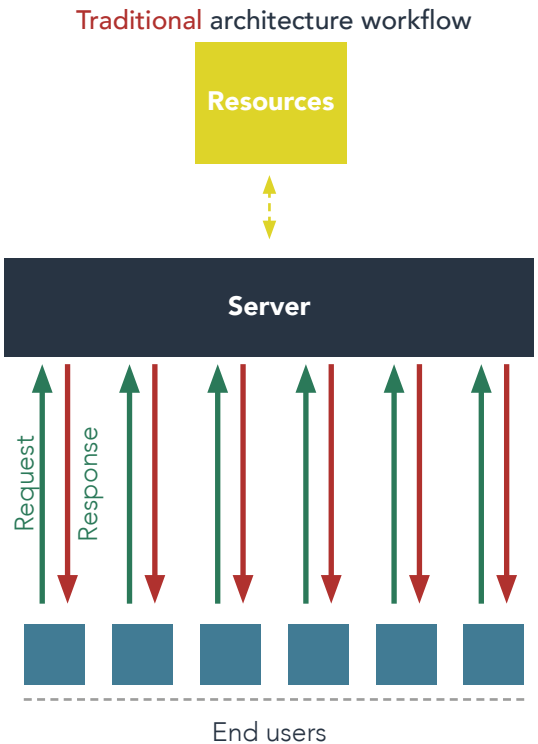
Traditional architecture workflow

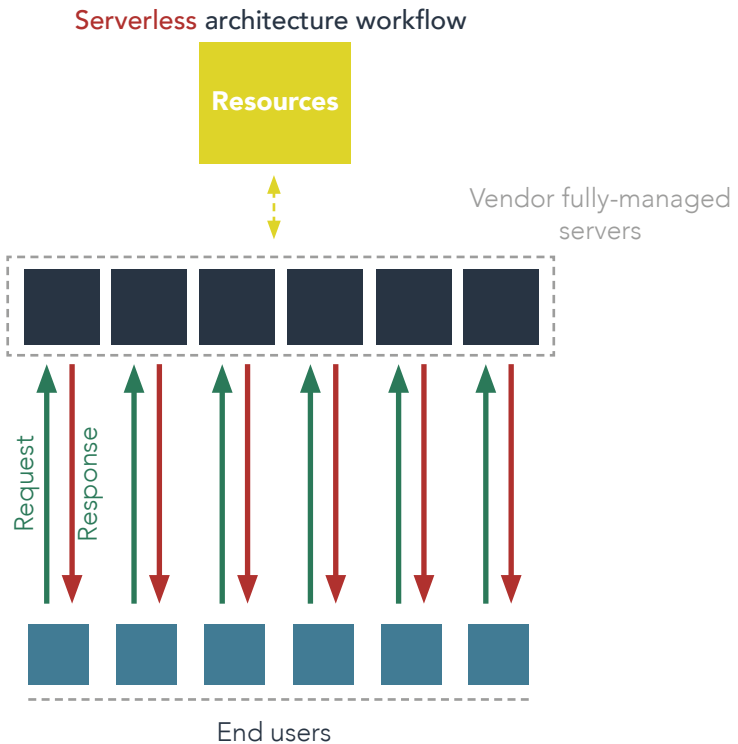


Serverless architecture workflow

...Why Serverless?

Traditional vs Serverless architecture

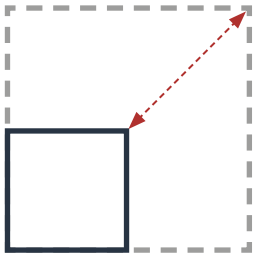




Serverless. Basics

Advantages

Fully-Managed
architecture



Auto-scaling &
elasticity

Pay as you go

Decrease
time-to-market

Serverless. Basics

Disadvantages

Loss of Control

Having a third party manage part of the infrastructure makes it tough to understand the whole system and adds debugging challenges.

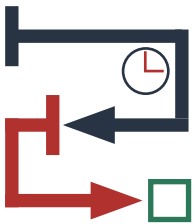
Test and error analysis

It can be very difficult to incorporate FaaS code into a local testing environment, making thorough testing of an application a more intensive task.

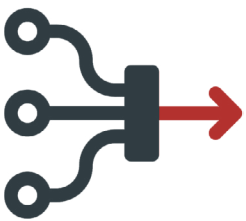
...When Serverless?

Serverless common use cases

CI/CD



Event-driven & CRON architecture

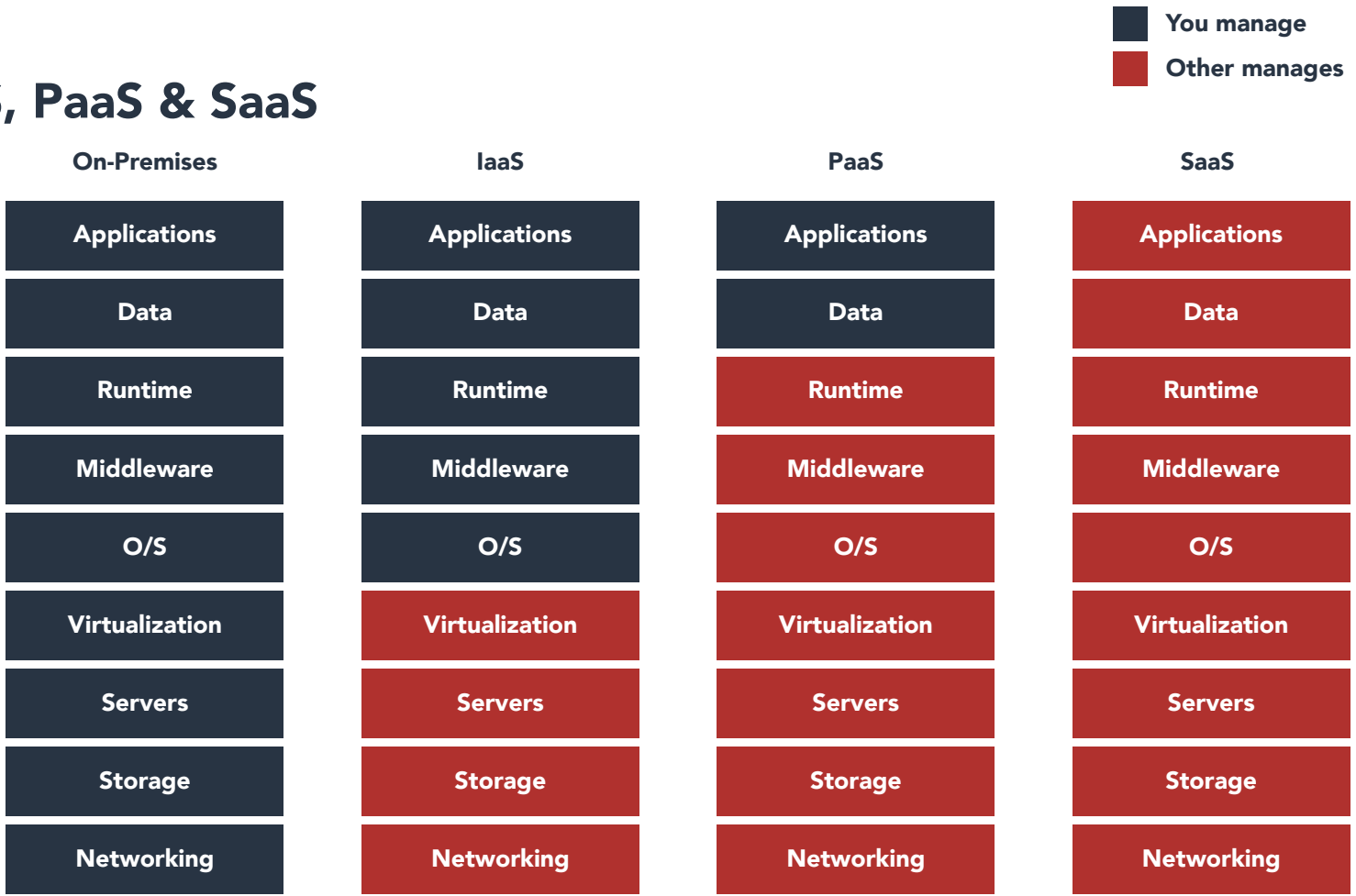


Data processing & IoT

Machine Learning

Building RESTful APIs

IaaS, PaaS & SaaS

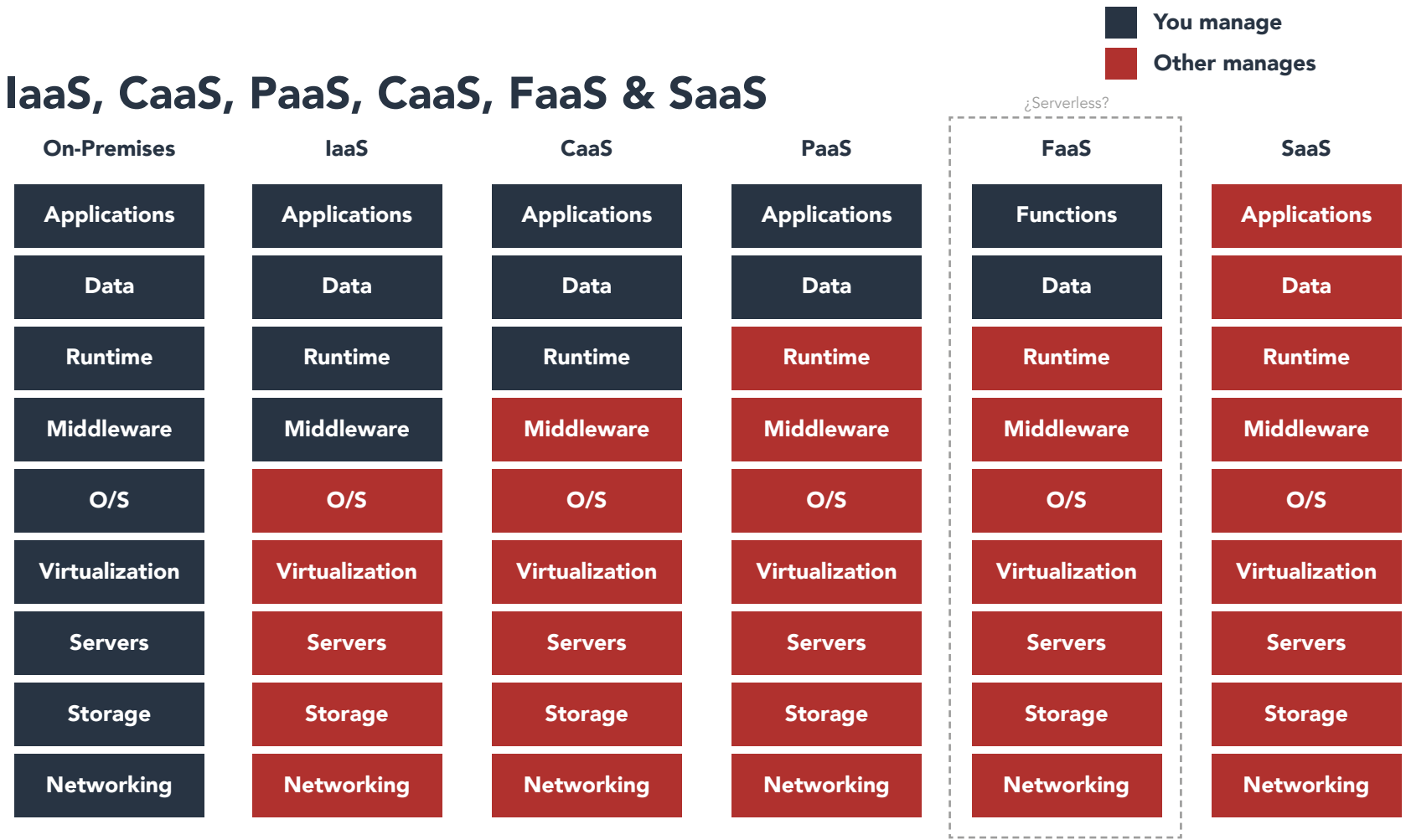


...And

Serverless

...?

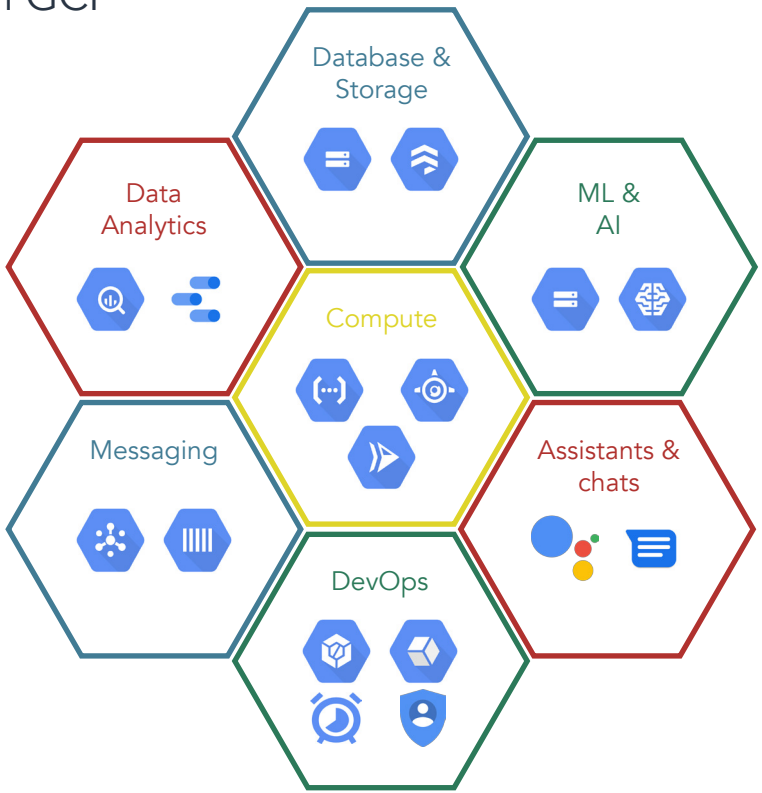
IaaS, CaaS, PaaS, CaaS, FaaS & SaaS



Let's dive
into GCP
Serverless

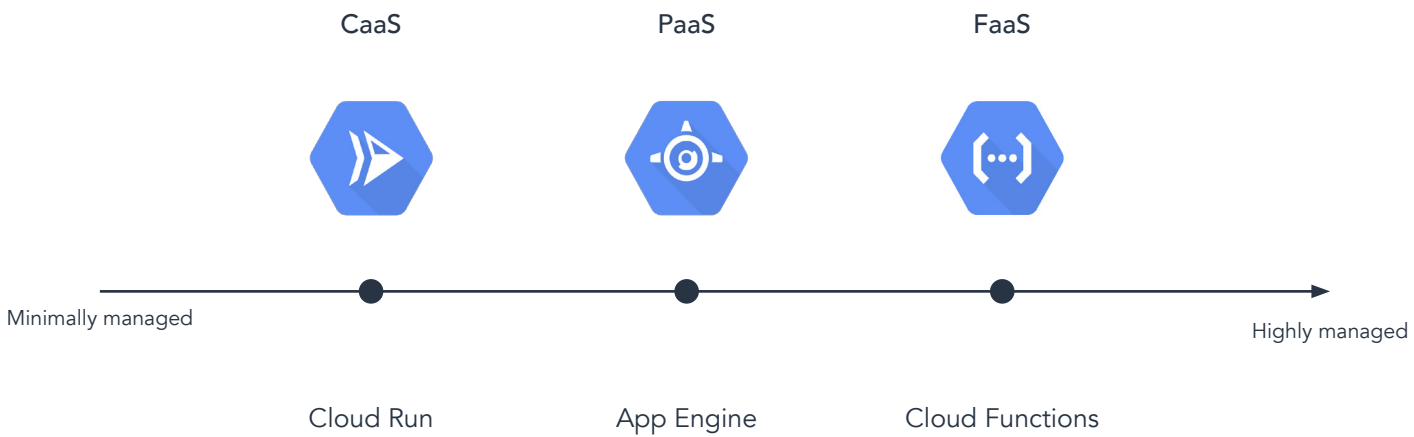
Serverless in GCP

Serverless architecture in GCP



Serverless in GCP

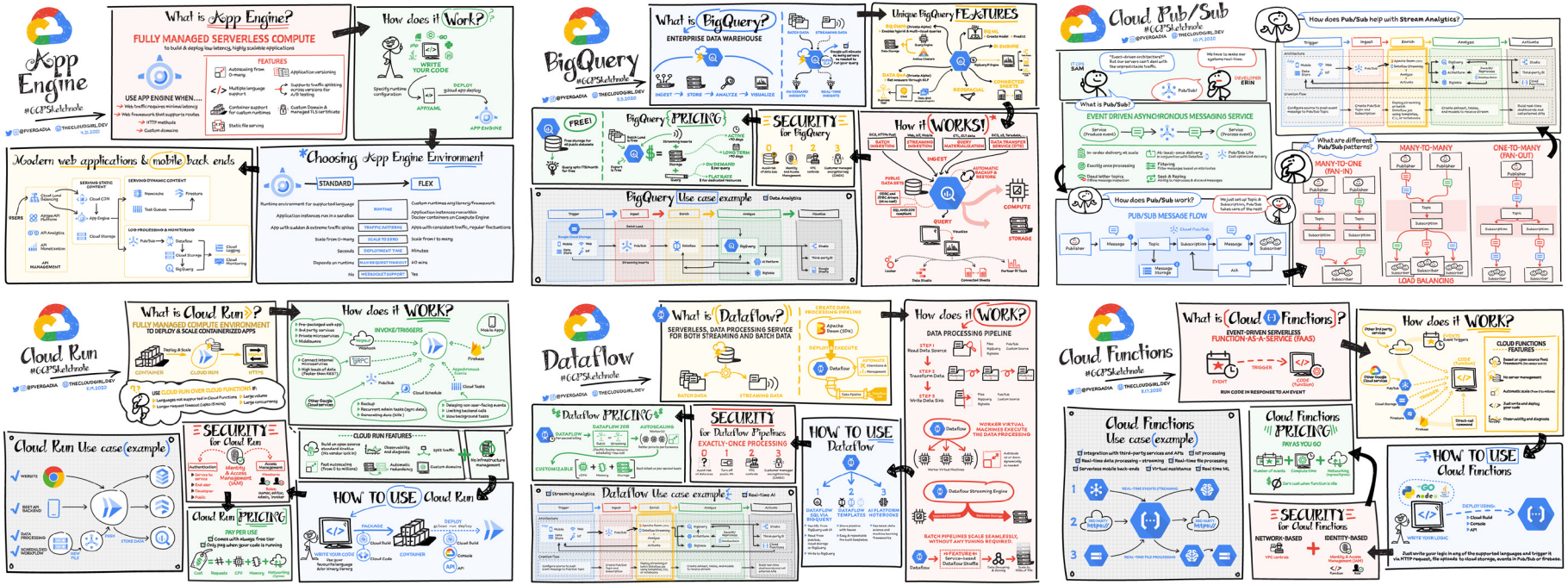
Serverless architecture in GCP



Serverless is much more than just FaaS

The Cloud Girl

https://thecloudgirl.dev/



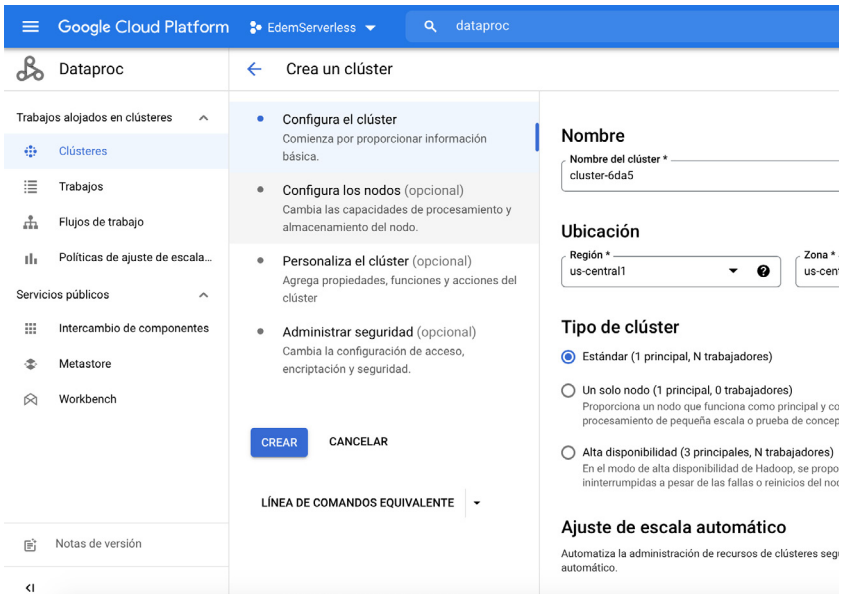
Serverless
...or not?

Practical tips...

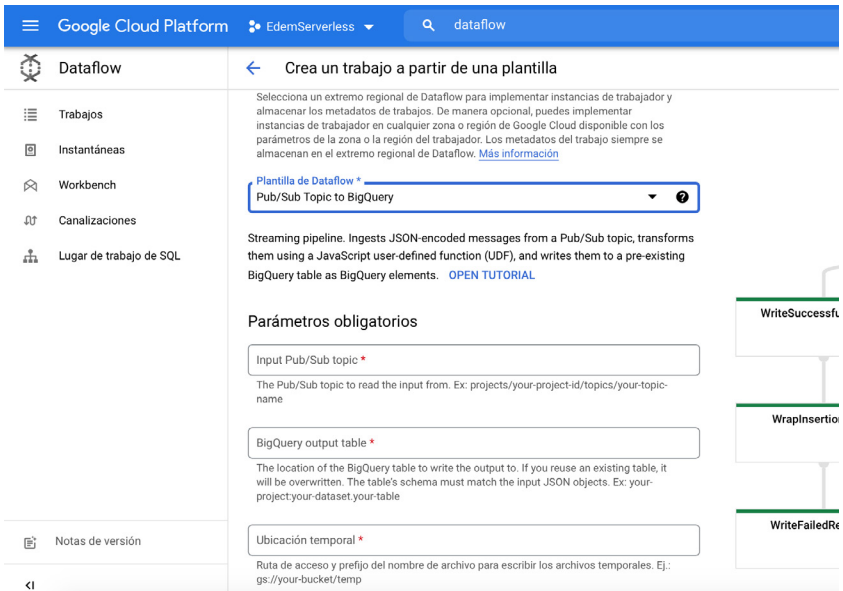
Serverless... or not?

Practical tips

Non-Serverless



Serverless



Dataflow

Serverless data processing 02

Definition & basic concepts

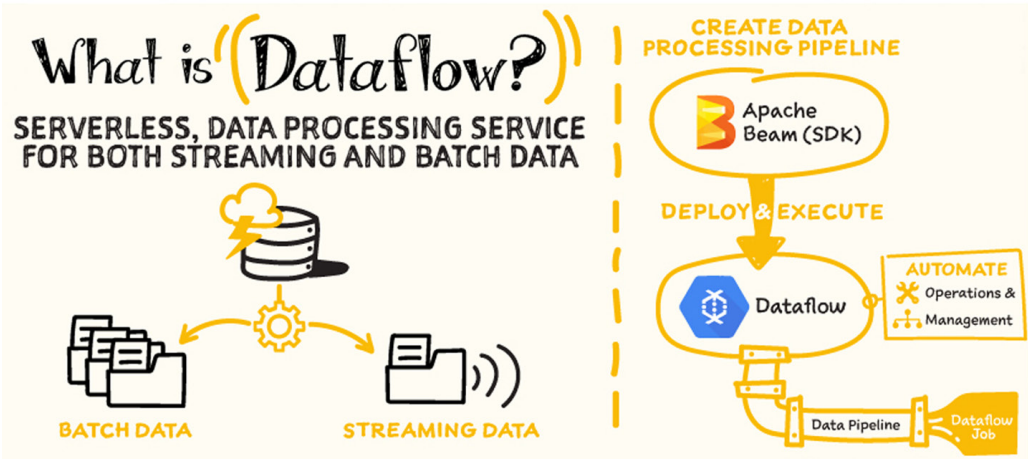
Apache Beam Programming model

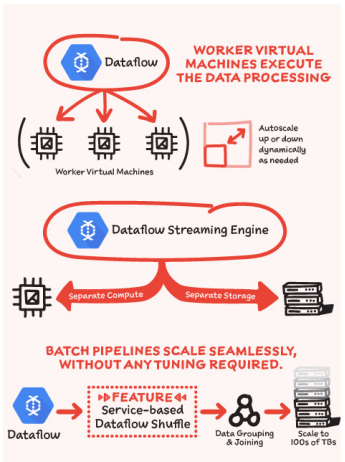
How to use Dataflow?

Dataflow

Definition

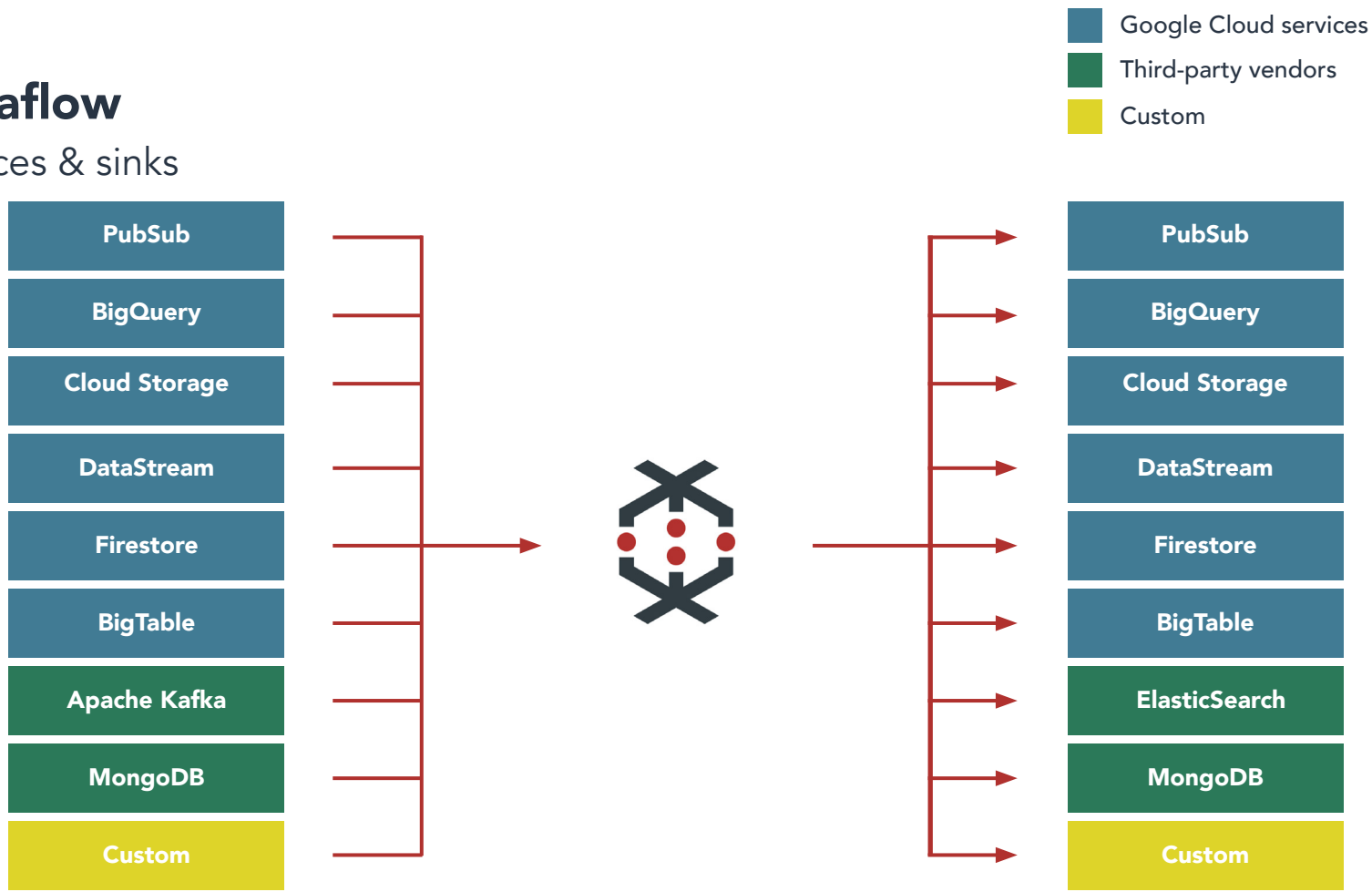
Cloud Datafusion is a Serverless data processing service for running batch and streaming Apache Beam pipelines.





Dataflow

Sources & sinks



Definition & basic concepts

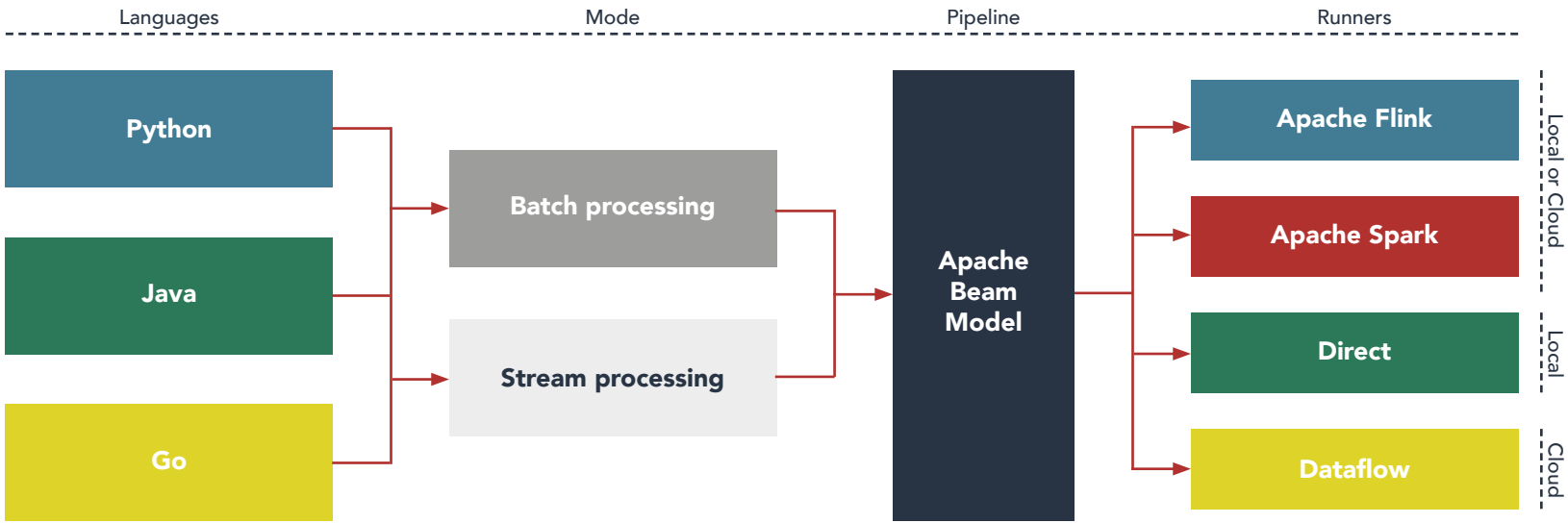
Apache Beam Programming model

How to use Dataflow?

Apache Beam

Basic concepts

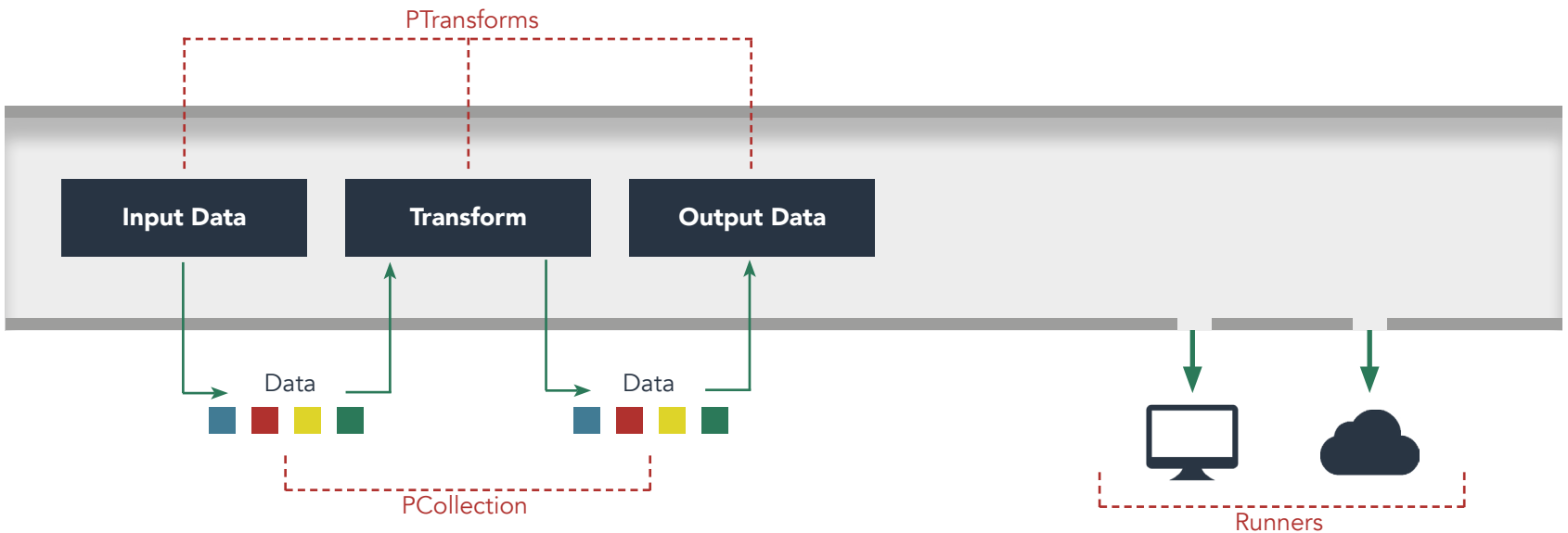
Apache Beam = Batch & streaming



Apache Beam

Basic concepts

Pipeline



Apache Beam

Basic concepts

Pipeline

- Pipeline concept represents the whole process to be performed: All data, calculations and required tasks to process that data.
- A Beam job starts by declaring a Pipeline object.

```
# Setting up the Beam pipeline options
options = PipelineOptions(save_main_session=True)
options.view_as(GoogleCloudOptions).project = opts.project
options.view_as(GoogleCloudOptions).region = opts.region
options.view_as(GoogleCloudOptions).staging_location = opts.staging_location
options.view_as(GoogleCloudOptions).temp_location = opts.temp_location
options.view_as(GoogleCloudOptions).job_name = '{0}{1}'.format('batch-minute-traffic-pipeline-',time.time_ns())
options.view_as(StandardOptions).runner = opts.runner
```

```
# Create the pipeline
p = beam.Pipeline(options=options)
```

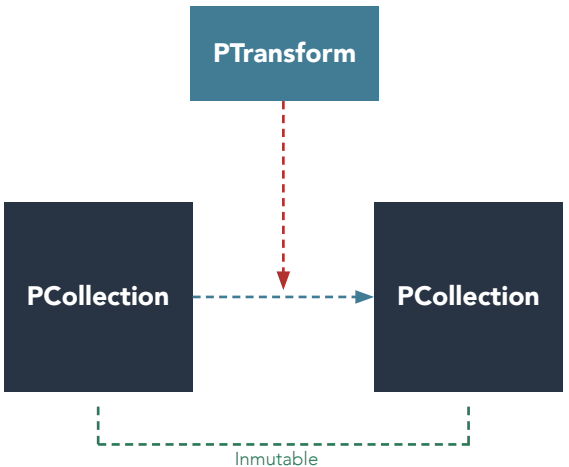
```
(p | 'ReadFromGCS' >> beam.io.ReadFromText(input_path)
   | 'ParseJson' >> beam.Map(parse_json).with_output_types(CommonLog)
   | 'AddEventTimestamp' >> beam.Map(add_timestamp)
   | "WindowByMinute" >> beam.WindowInto(beam.window.FixedWindows(60))
   | "CountPerMinute" >> beam.CombineGlobally(CountCombineFn()).without_defaults()
   | "AddWindowTimestamp" >> beam.ParDo(GetTimestampFn())
   | 'WriteToBQ' >> beam.io.WriteToBigQuery(
       table_name,
       schema=table_schema,
       create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
       write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE
       )
)

logging.getLogger().setLevel(logging.INFO)
logging.info("Building pipeline ...")

p.run()

if __name__ == '__main__':
  run()
```

Apache Beam

Basic concepts

PCollection = Parallel distribution

- Represents the data set for a pipeline. Is immutable. Each transformation results in a new PCollection.
- Random access. A PCollection does not support access to individual items, even though PTransforms are processed item by item.
- Size and boundedness: bounded(Batch) and unbounded(Streaming).
- Each element in a PCollection has an associated intrinsic timestamp assigned by the Source.
- The elements in a PCollection can be of any type, but all of the same type. However, to support distributed processing, Beam must be able to encode each individual element as a byte string.



Apache Beam

Basic concepts

PTransform

- Represents a processing operation on the dataset. It can be applied to one or more input PCollections, resulting in 0, 1, or more PCollections.

- User provides the processing logic (code) and this will be applied to each individual element of the PCollection, depending on the chosen backend and runner

- Typologies:
  - I/O Transforms (Read & write)
  - Conversion & transform (ParDo, GroupByKey, CoGroupByKey, Combine, Flatten, etc.)

```
# Setting up the Beam pipeline options
options = PipelineOptions(save_main_session=True)
options.view_as(GoogleCloudOptions).project = opts.project
options.view_as(GoogleCloudOptions).region = opts.region
options.view_as(GoogleCloudOptions).staging_location = opts.staging_location
options.view_as(GoogleCloudOptions).temp_location = opts.temp_location
options.view_as(GoogleCloudOptions).job_name = '{0}{1}'.format('batch-minute-traffic-pipeline-',time.time_ns())
options.view_as(StandardOptions).runner = opts.runner
```

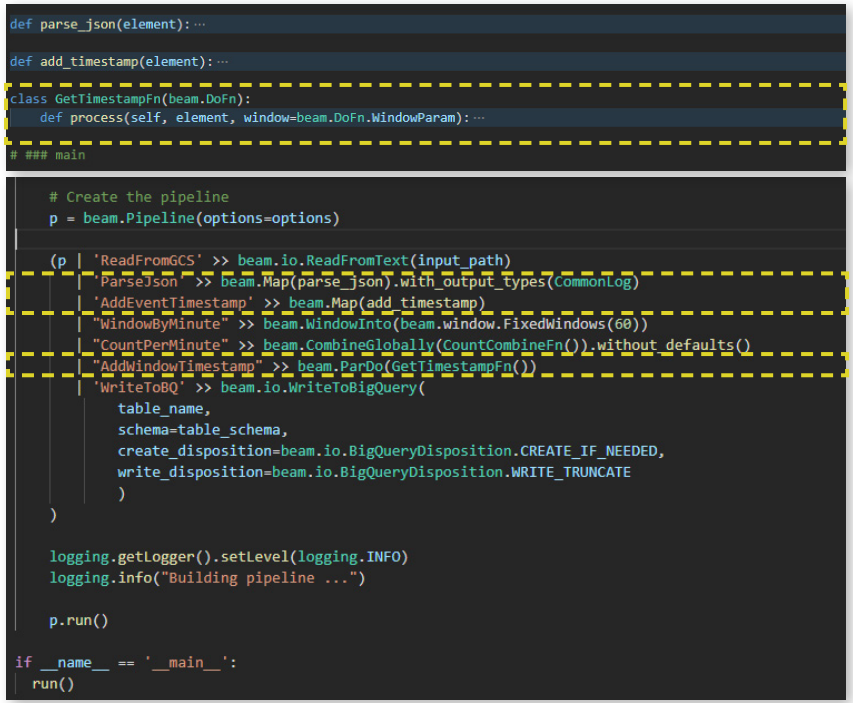
```
# Create the pipeline
p = beam.Pipeline(options=options)

(p | 'ReadFromGCS' >> beam.io.ReadFromText(input_path)
   | 'ParseJson' >> beam.Map(parse_json).with_output_types(CommonLog)
   | 'AddEventTimestamp' >> beam.Map(add_timestamp)
   | "WindowByMinute" >> beam.WindowInto(beam.window.FixedWindows(60))
   | "CountPerMinute" >> beam.CombineGlobally(CountCombineFn()).without_defaults()
   | "AddWindowTimestamp" >> beam.ParDo(GetTimestampFn())
   | 'WriteToBQ' >> beam.io.WriteToBigQuery(
      table_name,
      schema=table_schema,
      create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
      write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE
      )
)

logging.getLogger().setLevel(logging.INFO)
logging.info("Building pipeline ...")

p.run()

if __name__ == '__main__':
  run()
```

Apache Beam

Basic concepts

DoFn

- Beam class in which pipeline processing tasks are defined. here the logic to apply to each element of the PCollection is located, within a *Process* function.
- By having an input and an output PCollection, neither the element that is passed to the function, nor the *yield* or *return* that it returns, will be modified.
- If the DoFn function is simple, you can use a lambda function. Also, If there is a one-to-one mapping of input-output elements, Map function can be used.



```python
def parse_json(element): ...

def add_timestamp(element): ...

class GetTimestampFn(beam.DoFn):
    def process(self, element, window=beam.DoFn.WindowParam): ...

# ### main

    # Create the pipeline
    p = beam.Pipeline(options=options)

    (p | 'ReadFromGCS' >> beam.io.ReadFromText(input_path)
       | 'ParseJson' >> beam.Map(parse_json).with_output_types(CommonLog)
       | 'AddEventTimestamp' >> beam.Map(add_timestamp)
       | "WindowByMinute" >> beam.WindowInto(beam.window.FixedWindows(60))
       | "CountPerMinute" >> beam.CombineGlobally(CountCombineFn()).without_defaults()
       | "AddWindowTimestamp" >> beam.ParDo(GetTimestampFn())
       | 'WriteToBQ' >> beam.io.WriteToBigQuery(
           table_name,
           schema=table_schema,
           create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
           write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE
           )
    )

    logging.getLogger().setLevel(logging.INFO)
    logging.info("Building pipeline ...")

    p.run()

if __name__ == '__main__':
  run()
```

Apache Beam

Transformations

I/O Transforms

- Useful transformations to read or write data from some external source, such as files, databases or messaging queues. You can implement your own transformations if any source is not available, through a DoFn object.

```
    # Setting up the Beam pipeline options
    options = PipelineOptions(save_main_session=True)
    options.view_as(GoogleCloudOptions).project = opts.project
    options.view_as(GoogleCloudOptions).region = opts.region
    options.view_as(GoogleCloudOptions).staging_location = opts.staging_location
    options.view_as(GoogleCloudOptions).temp_location = opts.temp_location
    options.view_as(GoogleCloudOptions).job_name = '{0}{1}'.format('batch-minute-traffic-pipeline-',time.time_ns())
    options.view_as(StandardOptions).runner = opts.runner
```

```
    # Create the pipeline
    p = beam.Pipeline(options=options)

    (p | 'ReadFromGCS' >> beam.io.ReadFromText(input_path)
       | 'ParseJson' >> beam.Map(parse_json).with_output_types(CommonLog)
       | 'AddEventTimestamp' >> beam.Map(add_timestamp)
       | "WindowByMinute" >> beam.WindowInto(beam.window.FixedWindows(60))
       | "CountPerMinute" >> beam.CombineGlobally(CountCombineFn()).without_defaults()
       | "AddWindowTimestamp" >> beam.ParDo(GetTimestampFn())
       | 'WriteToBQ' >> beam.io.WriteToBigQuery(
           table_name,
           schema=table_schema,
           create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
           write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE
           )
    )

    logging.getLogger().setLevel(logging.INFO)
    logging.info("Building pipeline ...")

    p.run()

if __name__ == '__main__':
  run()
```

Apache Beam

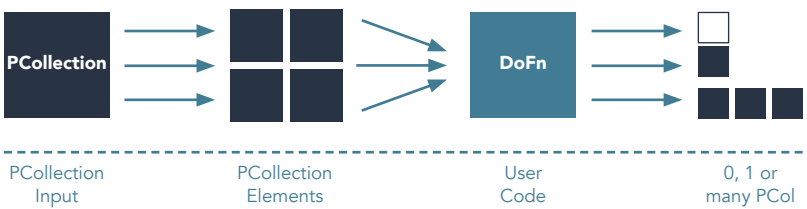
Transformations

ParDo

- Beam transform for generic parallel processing. Similar to Map in the MapReduce algorithm.

- Use cases

  Filtering a data set.
  Formatting each element in a data set.
  Extracting parts of each element in a data set.
  Performing computations of each element in a data set.




```python
def parse_json(element): ···

def add_timestamp(element): ···

class GetTimestampFn(beam.DoFn):
    def process(self, element, window=beam.DoFn.WindowParam): ···
# ### main
```

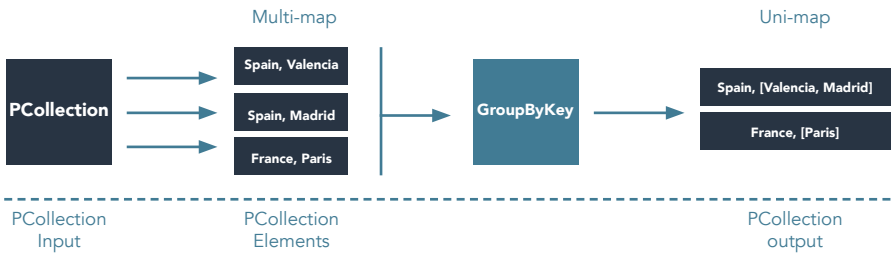
```python
    # Create the pipeline
    p = beam.Pipeline(options=options)

    (p | 'ReadFromGCS' >> beam.io.ReadFromText(input_path)
       | 'ParseJson' >> beam.Map(parse_json).with_output_types(CommonLog)
       | 'AddEventTimestamp' >> beam.Map(add_timestamp)
       | "WindowByMinute" >> beam.WindowInto(beam.window.FixedWindows(60))
       | "CountPerMinute" >> beam.CombineGlobally(CountCombineFn()).without_defaults()
       | "AddWindowTimestamp" >> beam.ParDo(GetTimestampFn())
       | 'WriteToBQ' >> beam.io.WriteToBigQuery(
           table_name,
           schema=table_schema,
           create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
           write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE
           )
    )

    logging.getLogger().setLevel(logging.INFO)
    logging.info("Building pipeline ...")

    p.run()

if __name__ == '__main__':
  run()
```

Apache Beam

Transformations

GroupByKey

- Beam transform to process key/value PCollections.
- Similar to Shuffle in MapReduce algorithm.
- Multimap Collection: same key for different values, grouping all the values by a single key.
- Global windows and triggers are required if you are dealing with unbounded data.

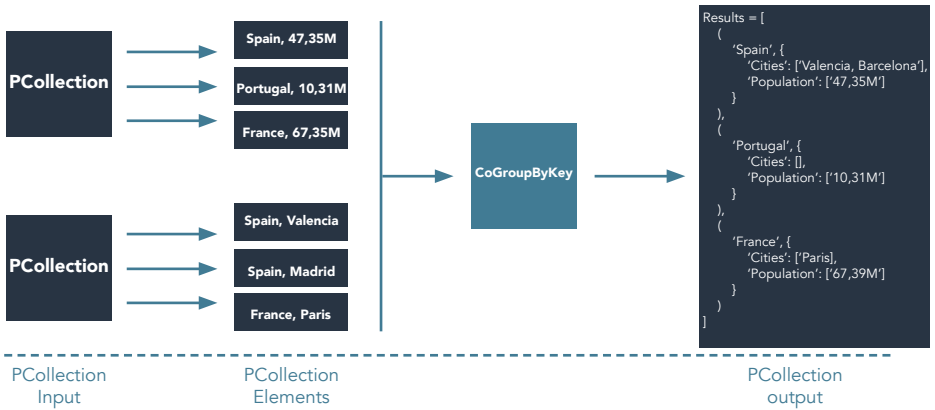


Apache Beam

Transformations

CoGroupByKey

- Beam transformation that performs a combination of multiple data sets that provide related information.
- As GroupByKey, if the dataset is unbounded, global windows and triggers should be used to limit the sample to work with.



Apache Beam

Transformations

Combine

```
# Create the pipeline
p = beam.Pipeline(options=options)

(p | 'ReadFromGCS' >> beam.io.ReadFromText(input_path)
   | 'ParseJson' >> beam.Map(parse_json).with_output_types(CommonLog)
   | 'AddEventTimestamp' >> beam.Map(add_timestamp)
   | "WindowByMinute" >> beam.WindowInto(beam.window.FixedWindows(60))
   | "CountPerMinute" >> beam.CombineGlobally(CountCombineFn()).without_defaults()
   | "AddWindowTimestamp" >> beam.ParDo(GetTimestampFn())
   | 'WriteToBQ' >> beam.io.WriteToBigQuery(
       table_name,
       schema=table_schema,
       create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
       write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE
       )
)
```

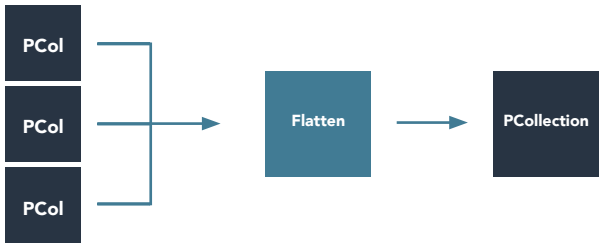
- Beam transform for combining collections of elements or values in your data. Combine has variants that work with full PCollections and some with values for each key in key / value collections.
- When you apply a Combine transform, you must provide the function that contains the logic for combining the elements or values. The combining function should be commutative and associative.
- For simple operations, a simple function can be used. In contrast, in complex operations, CombineFn will be use

Apache Beam

Transformations

Flatten

- Flatten is a Beam transform for PCollection objects that store the same data type. Flatten merges multiple PCollection objects into a single logical PCollection.
- When using Flatten to merge PCollection objects that have a windowing strategy applied, all of the PCollection objects you want to merge must use a compatible windowing strategy and window sizing.




```
def run():
    """Build and run the pipeline."""
    options = PipelineOptions(save_main_session=True, streaming=True)

    with beam.Pipeline(options=options) as p:

        p1 = p | 'read from vehicles' >> beam.io.ReadFromPubSub(subscription="projects/gm-ai-defect-inspection/subscriptions/vehicles-sub", with_attributes=True)
        p2 = p | 'read from dealers' >> beam.io.ReadFromPubSub(subscription="projects/gm-ai-defect-inspection/subscriptions/dealers-sub", with_attributes=True)
        p3 = p | 'read from transports' >> beam.io.ReadFromPubSub(subscription="projects/gm-ai-defect-inspection/subscriptions/transports-sub", with_attributes=True)
        p4 = p | 'read from events' >> beam.io.ReadFromPubSub(subscription="projects/gm-ai-defect-inspection/subscriptions/events-sub", with_attributes=True)
        p5 = p | 'read from defects' >> beam.io.ReadFromPubSub(subscription="projects/gm-ai-defect-inspection/subscriptions/defects-sub", with_attributes=True)
        p6 = p | 'read from telemetry' >> beam.io.ReadFromPubSub(subscription="projects/gm-ai-defect-inspection/subscriptions/telemetryEvents-sub", with_attributes=True)

        # Merge the PCollections
        merged = (p1, p2, p3, p4, p5, p6) | 'merge sources' >> beam.Flatten()
```

Apache Beam

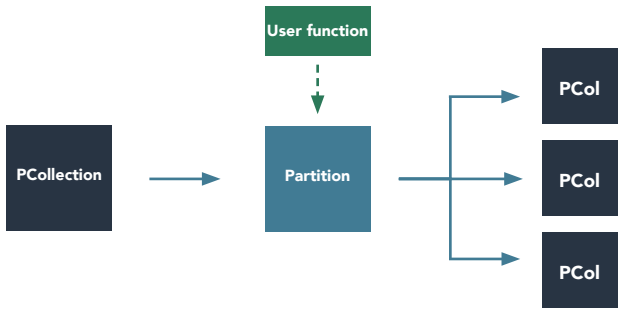
Transformations

Partition

- Beam transform for PCollection objects that store the same data type. Partition splits a single PCollection into a fixed number of smaller collections, according to a partitioning function that you provide.
- The number of partitions must be determined at graph construction time, but you cannot determine the number of partitions in mid-pipeline.

Code

```
def PartitionFn (element, num_partitions):
    return int()

p | "Partition" >> beam.Partition(PartitionFn, 3))
```

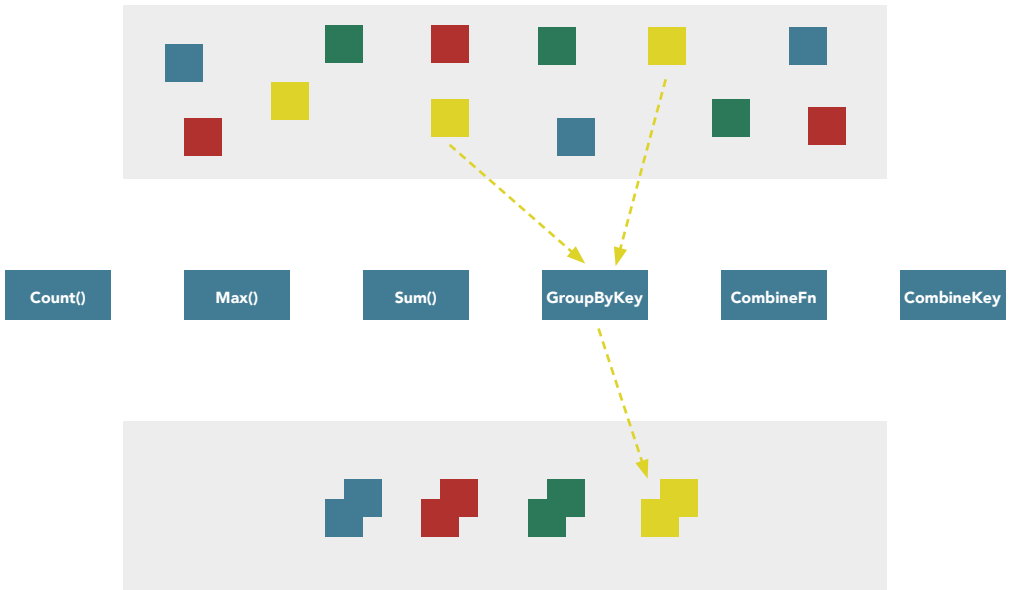


Apache Beam

Aggregations (Batch)

- Aggregation is computing a value from multiple input elements. Similar to Reduce in the Map/Reduce algorithm.

- PCollections must have the same k / v and be placed in the same window. The results are shown when all the data has been processed.

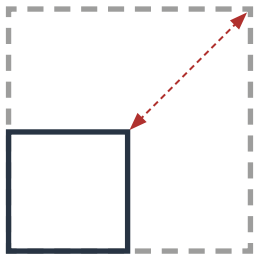


but...

Streaming?

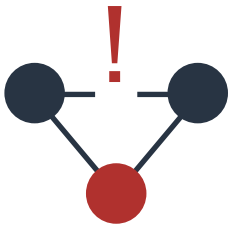
Apache Beam

Streaming challenges



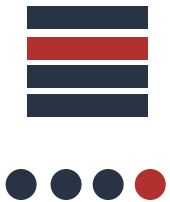
Scalability

Streaming data generally only grows larger and more frequent



Fault tolerance

Maintain fault tolerance despite increasing volumes of data



Data model

Is it streaming or repetead batch?

Timing

What if data arrives late?

Apache Beam

Advanced

Apache Beam

Advanced

Event Time

- Timestamp referring to when each individual event occurs.
- Typically embedded within the records at source system side. So it can be extracted from
- Deterministic. Event time gives correct results even on out of order events, late events, or
- Event time processing often incurs a certain latency , due to its nature of waiting a certain

Processing Time

- Processing time refers to the system time of the machine that is executing the respective operation.
- All time based operations will use the system clock of the machines that.
- Requires no coordination between stream and machines, providing the best performance and lowest latency.
- In distributed and asynchronous environments it is not deterministic, since it is susceptible according to the speed at which the data arrives from a message queue.

Apache Beam

Advanced

Windows

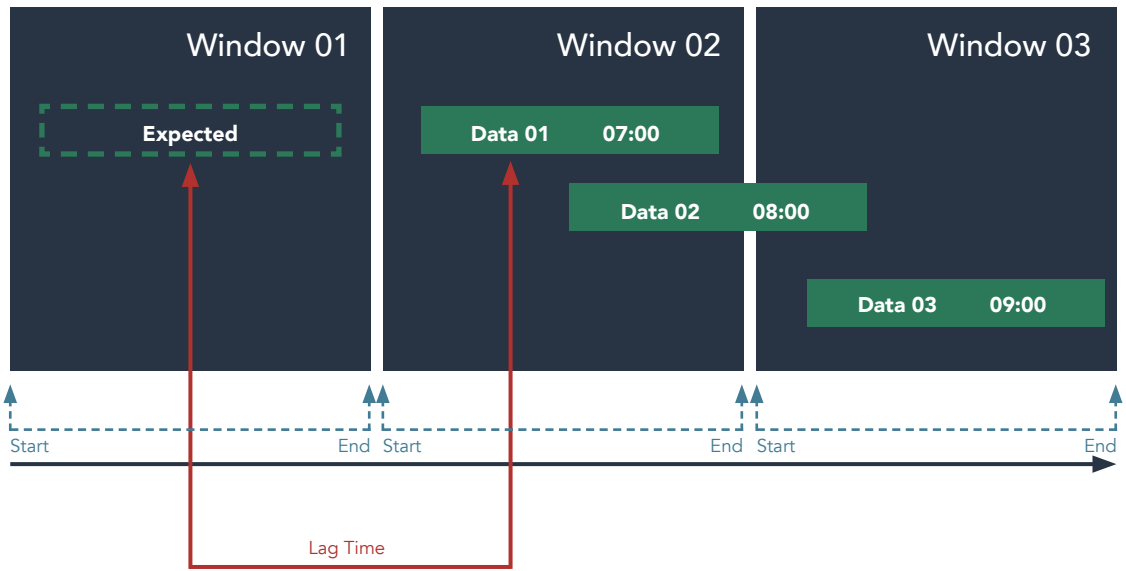
```
# Create the pipeline
p = beam.Pipeline(options=options)

(p | 'ReadFromGCS' >> beam.io.ReadFromText(input_path)
   | 'ParseJson' >> beam.Map(parse_json).with_output_types(CommonLog)
   | 'AddEventTimestamp' >> beam.Map(add_timestamp)
   | "WindowByMinute" >> beam.WindowInto(beam.window.FixedWindows(60))
   | "CountPerMinute" >> beam.CombineGlobally(CountCombineFn()).without_defaults()
   | "AddWindowTimestamp" >> beam.ParDo(GetTimestampFn())
   | 'WriteToBQ' >> beam.io.WriteToBigQuery(
       table_name,
       schema=table_schema,
```

- Subdivides a PCollection into windows according to the timestamps of its individual element and enable grouping operations over unbounded collections by dividing the collection into windows of finite collections.
- Although PCollection is unbounded, it is processed as a succession of multiple finite windows.
- Each item in a PCollection is added to a window, depending on the WindowsFn passed.
- By default, all PCollections are assigned to the single global window and late data is discarded. The single global window and the default trigger require the entire dataset to be available prior to processing, which is not possible with streaming data.

Apache Beam

Advanced

Windows



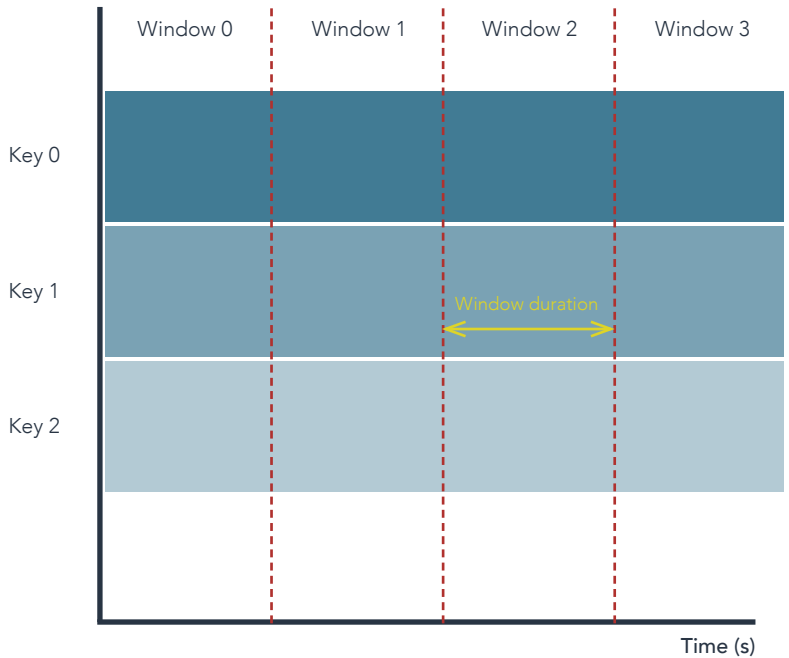
Apache Beam

Advanced

A. Fixed Windows

Code

```
p | "FixedWindow" >> beam.WindowInto(window.FixedWindows(10))
```



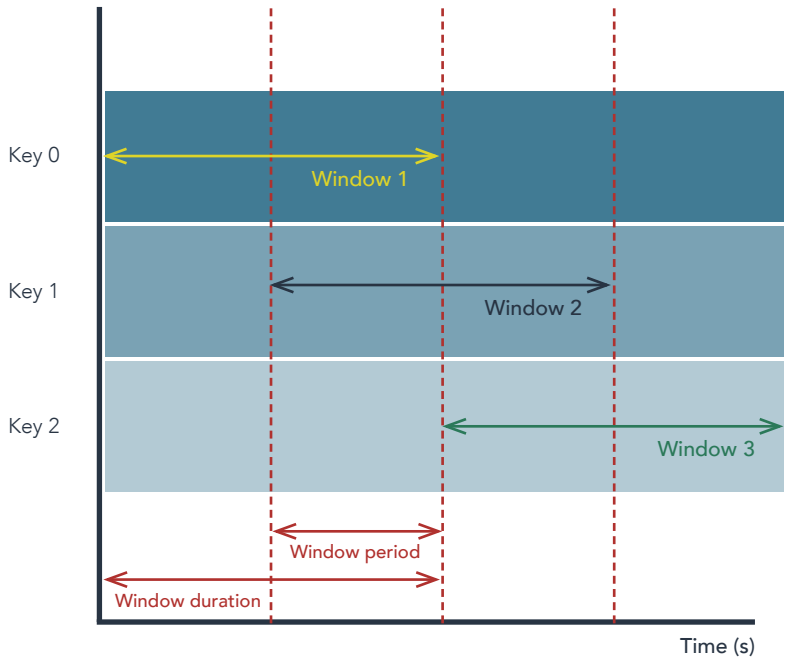
Apache Beam

Advanced

B. Sliding Windows

Code

```
p | "SlidingWindow" >> beam.WindowInto(window.SlidingWindows(10,5))
```



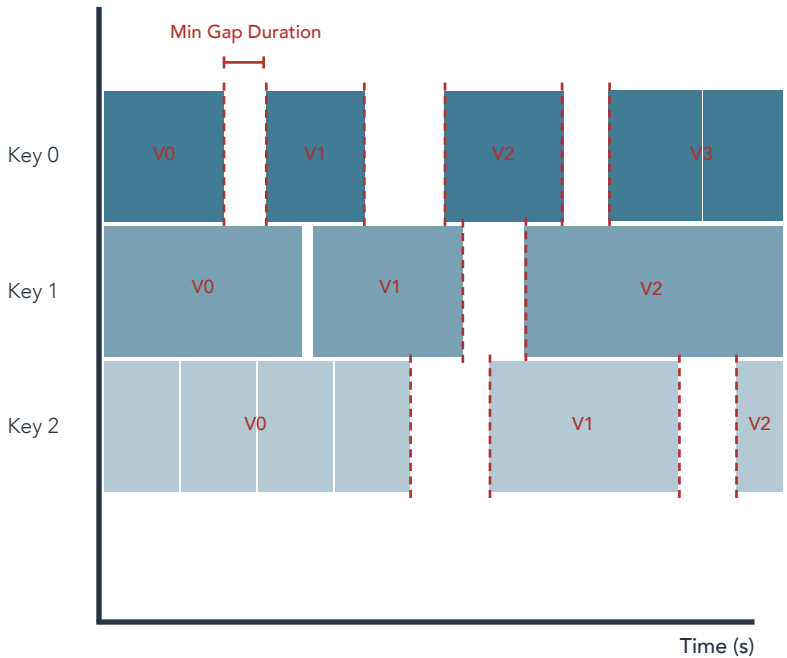
Apache Beam

Advanced

C. Session Windows

Code

```
p | "SessionWindow" >> beam.WindowInto(window.Sessions(10 * 5))
```

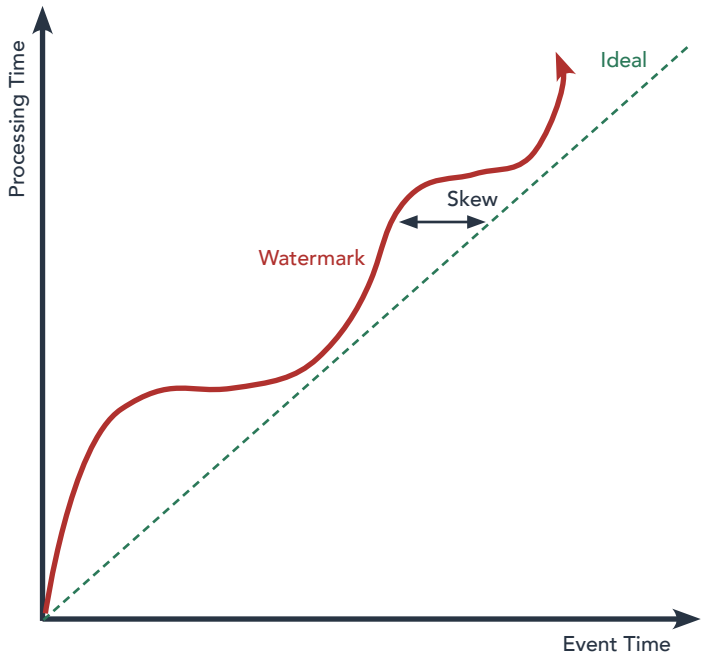


Apache Beam

Advanced

Watermarks

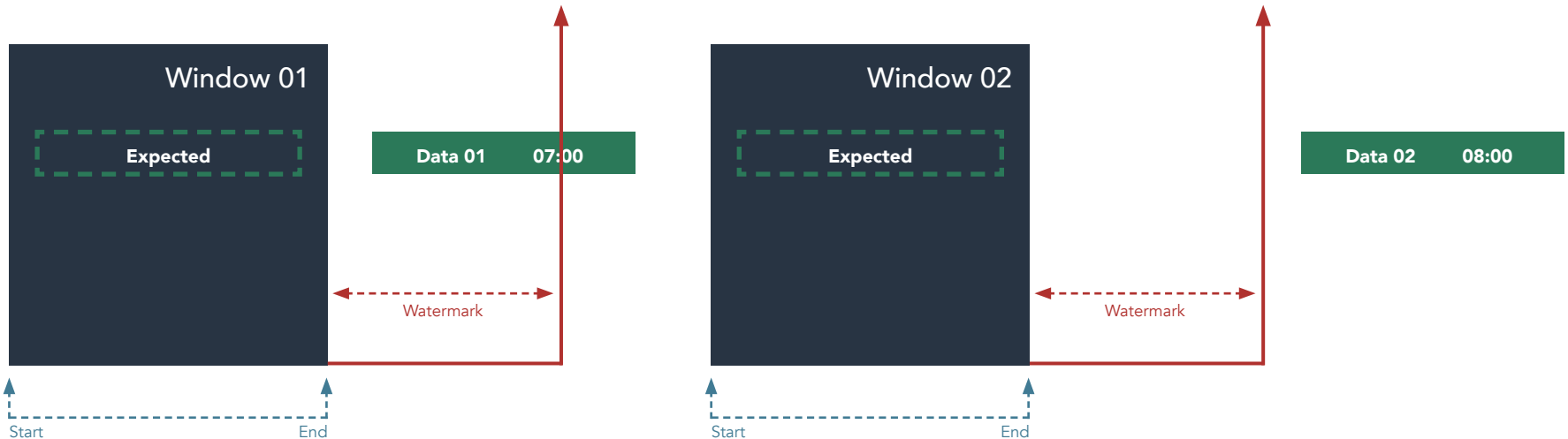
- How to deal with lag between Event Time vs Processing Time? How to deal with systems that do not preserve order?
- A watermark is an assumption of when all data from a certain window is expected to have reached the pipeline.
- A watermark is basically a timestamp based on event-time that the source is responsible for producing.
- Once the watermark passes the window, any additional items that arrive are considered late.
- Too slow? Results are delayed | Too fast? Some data is late.



Apache Beam

Advanced

Watermarks



The data still good

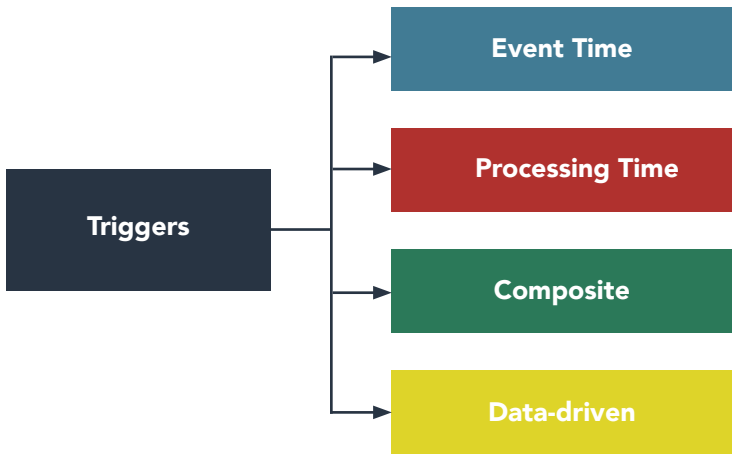
The data is late

Apache Beam

Advanced

Triggers

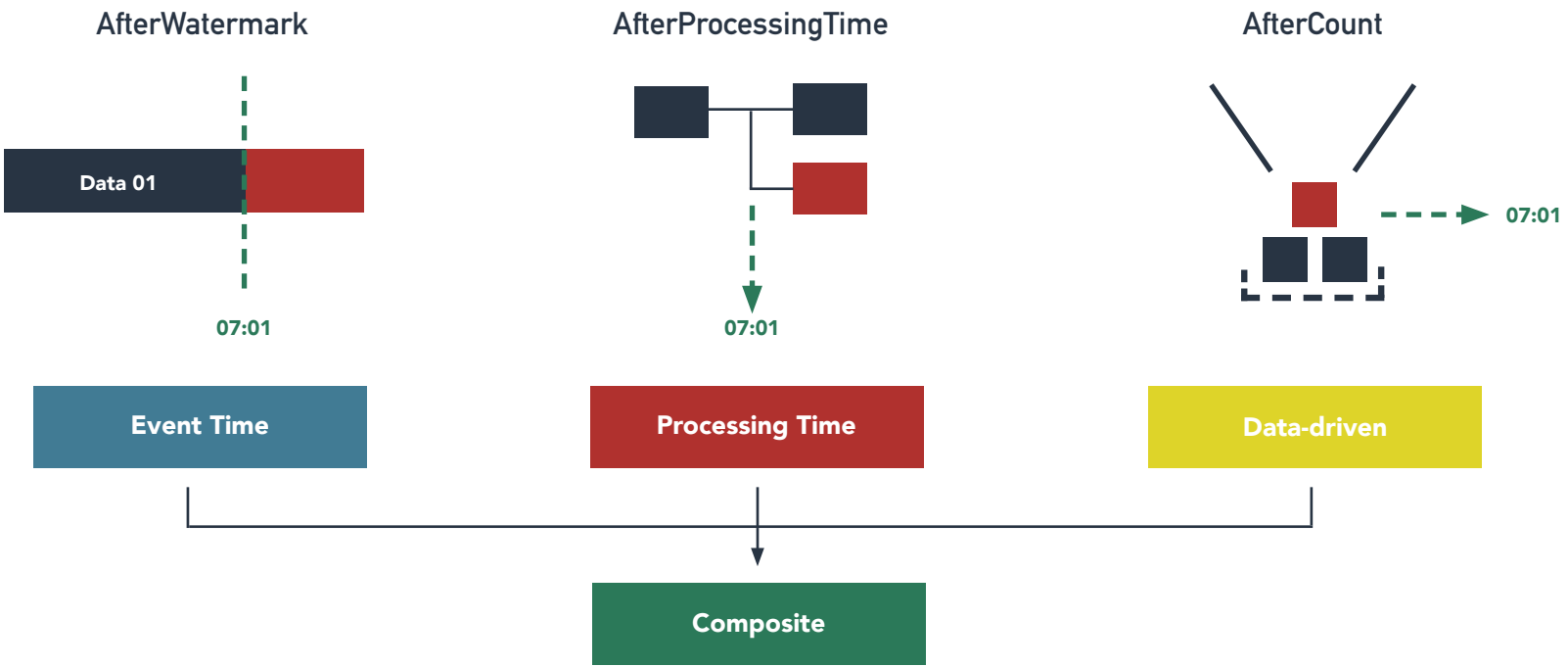
- Determine when to emit the aggregated results of each window
- Triggers allow Beam to emit early results, before all the data in a given window has arrived
- Triggers allow processing of late data by triggering after the event time watermark passes the end of the window.
- Important aspects to deal with:
  - Completeness
  - Latency
  - Cost



Apache Beam

Advanced

Triggers



Definition & basic concepts

Apache Beam Programming model

How to use Dataflow?

How to use Dataflow?

Dataflow Templates

Dataflow SQL

Dataflow Notebooks

...Why Templates?

Dataflow

Benefits of using Templates

Run recurring pipelines

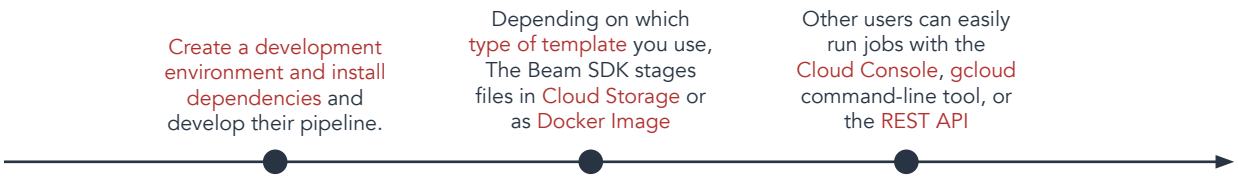
Share with your team

Custom pipeline with same Template

Dataflow

how does Templates works?

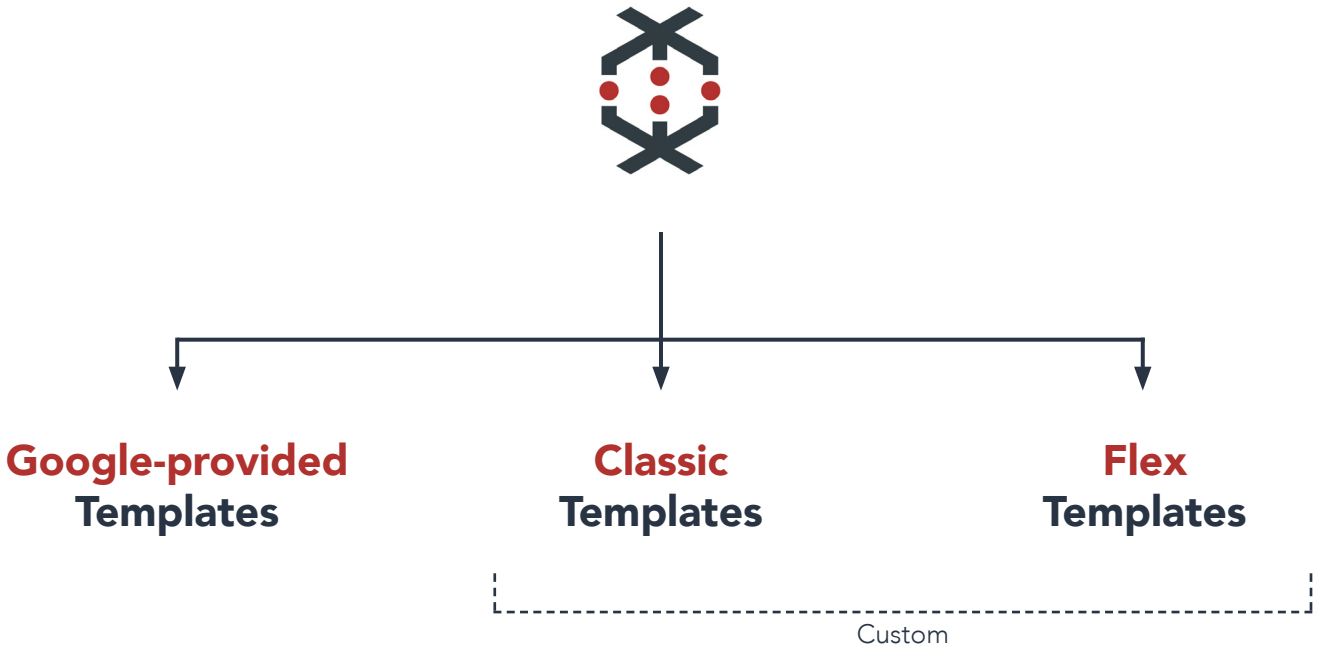
Non-templates workflow



Templates workflow

Dataflow

Templates



Dataflow

Templates

Google-provided Templates

Select Pre-built Template from Dataflow UI

Run Dataflow Job

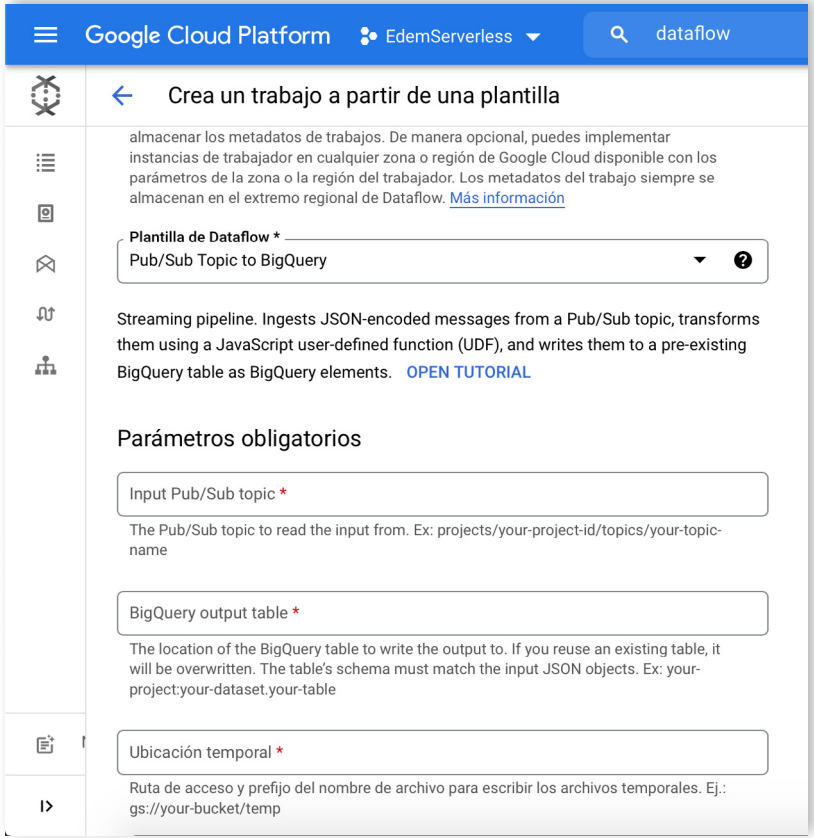
Advantages:

Rapid Deployment

Disadvantages:

A. Same input and output fields and formats.

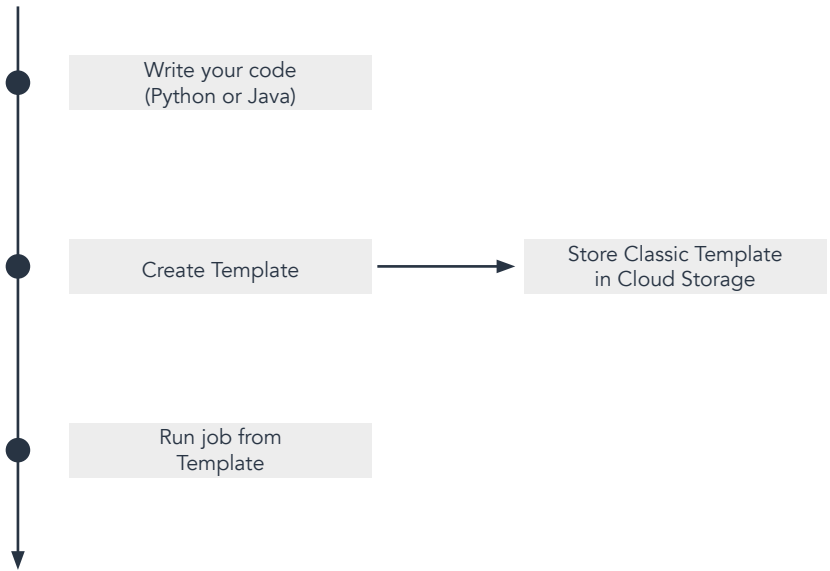
B. Resources must be created before running the job.



Dataflow

Templates

Classic Templates



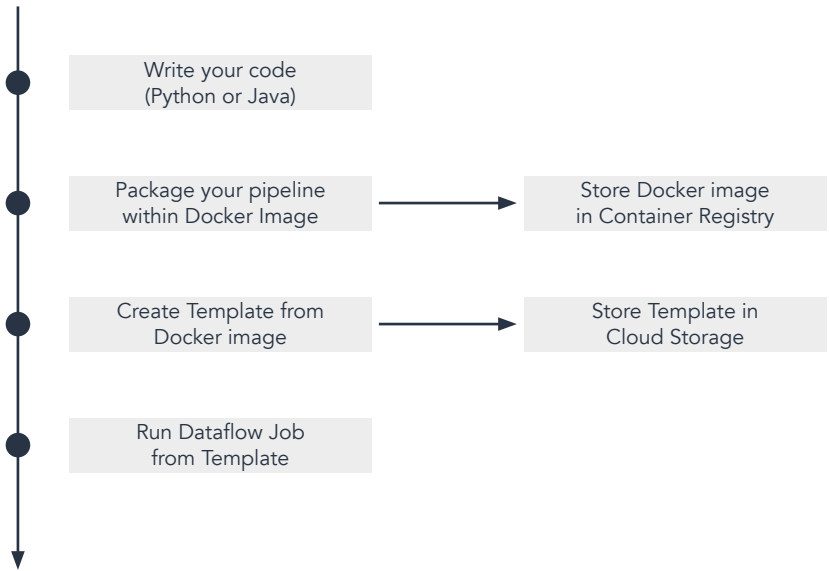
Create Template

```
Python -m <YOUR_PYTHON_MODULE> \
        --runner DataflowRunner \
        --project <YOUR_PROJECT_ID> \
        --staging_location gs://<BUCKET_NAME>/staging
        --temp_location gs://<BUCKET_NAME>/temp
        --template_location gs://<BUCKET_NAME>/<TEMPLATE_NAME>
        --region <REGION_ID>
```

Dataflow

Templates

Flex Templates



Build Image

```
gcloud builds submit —tag “gcr.io/<YOUR_PROJECT_ID>/<TEMPLATE_NAME>:latest” .
```

Create Template

```
gcloud dataflow flex-template build “gs://<BUCKET_NAME>/<TEMPLATE_NAME>.json \
        --image “DOCKER_IMAGE_CREATED_BEFORE” \
        --sdk-language “PYTHON”
```

Run Dataflow Job

```
gcloud dataflow flex-template run “<YOUR_DATAFLOW_JOB” \
        --template-file-gcs-location “<YOUR_TEMPLATE_CREATED_BEFORE>” \
        --parameters [OPTIONAL] \
        --region <REGION_ID>
```

Dataflow

Templates

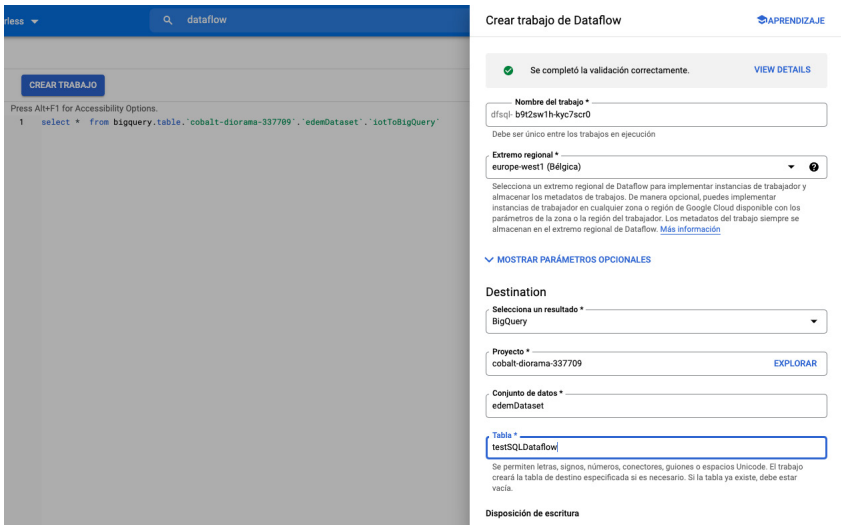
Classic Templates vs Flex Templates

Feature	Classic templates	Flex Templates
Separate staging and execution steps	Yes	Yes
Run the template using the Google Cloud console, `gcloud` tool, or REST API calls	Yes	Yes
Run pipeline without recompiling code	Yes	Yes
Run pipeline without development environment and associated dependencies	Yes	Yes
Customize pipeline execution with runtime parameters	Yes	Yes
Can update streaming jobs[1]	Yes	Yes
Supports FlexRS[1]	Yes	Yes
Run validations upon job graph construction to reduce runtime errors	No	Yes
Can change job execution graph after the template is created	No	Yes
Supports SQL parameters	No	Yes
Supports I/O interfaces beyond `ValueProvider`	No	Yes

Dataflow

SQL

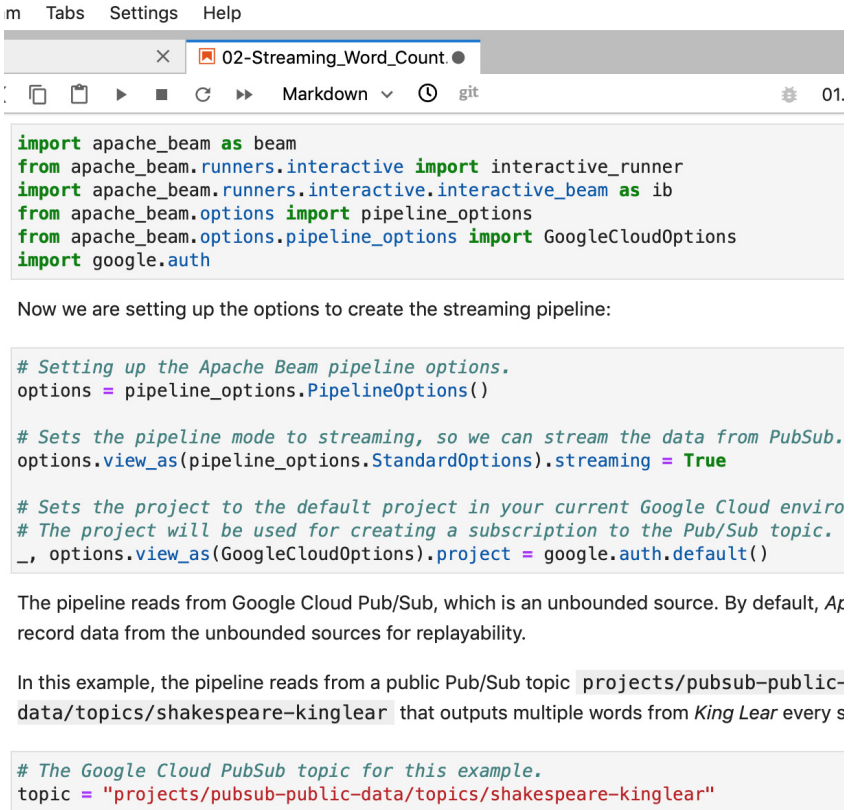
- Creates dataflow job with SQL syntax from Dataflow UI or BigQuery UI
- Uses ZetaSQL, the same dialect as BigQuery standard SQL
- It's a Beam SqlTransform in a Dataflow Flex template
- Simple way to create a streaming pipeline.
- Currently, Dataflow SQL can query from PubSub, BigQuery and Cloud Storage, but only write to PubSub topics and BigQuery tables.



Dataflow

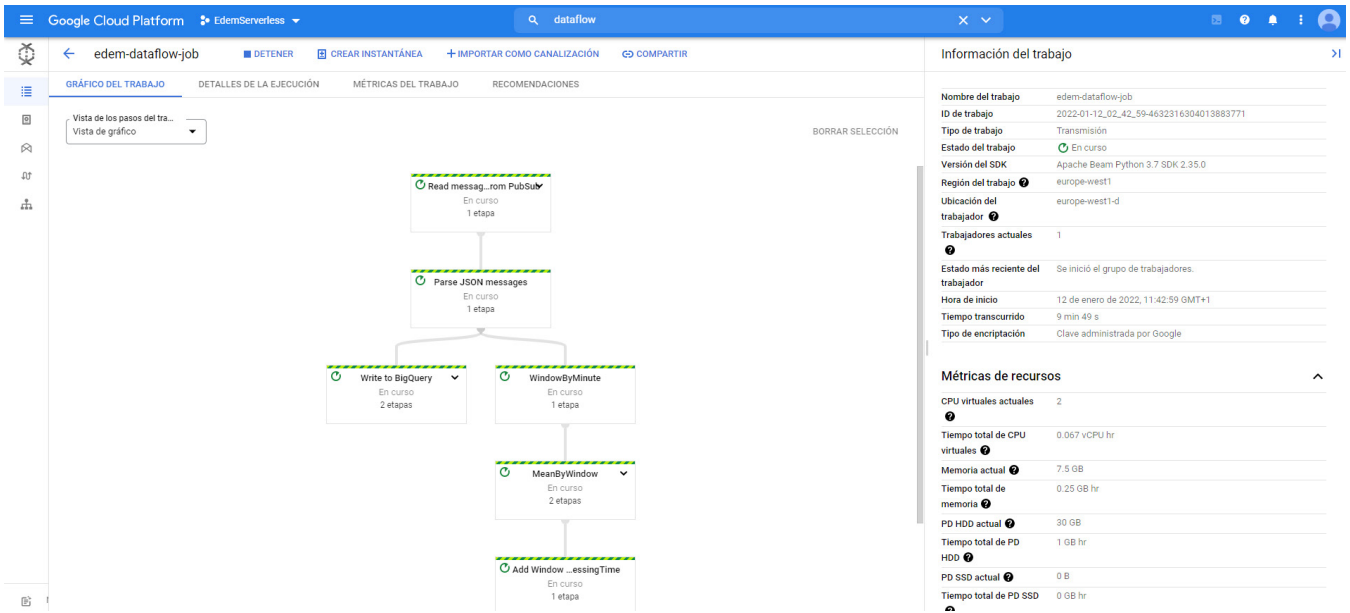
Notebooks

- Pythonic API compatible with Pandas Dataframes.
- Parallel processing with Beam model.
- Beam DataFrames as a DSL for Beam pipelines.
- Simple way to create a streaming pipeline.
- Provices access to familiar programming interfaces within Beam pipeline.



Dataflow

Monitoring. Execution details



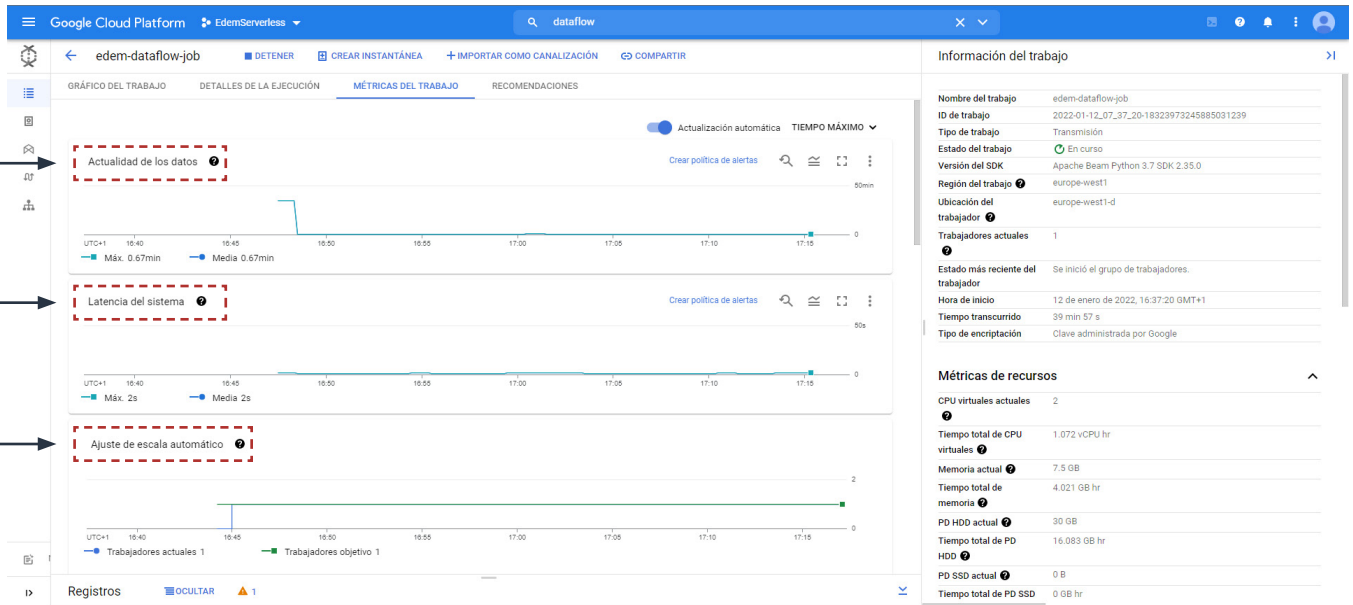
Dataflow

Monitoring. Execution details

Data freshness is the amount of time between real time and the output watermark. Each step of your pipeline has an output data watermark.

System Latency is the current maximum duration of time measured in seconds for which an item of data is processed or awaits processing.

Dataflow automatically chooses the number of worker instances required to run your **autoscaling** job. The number of worker instances can change over time according to the job requirements.



Success stories

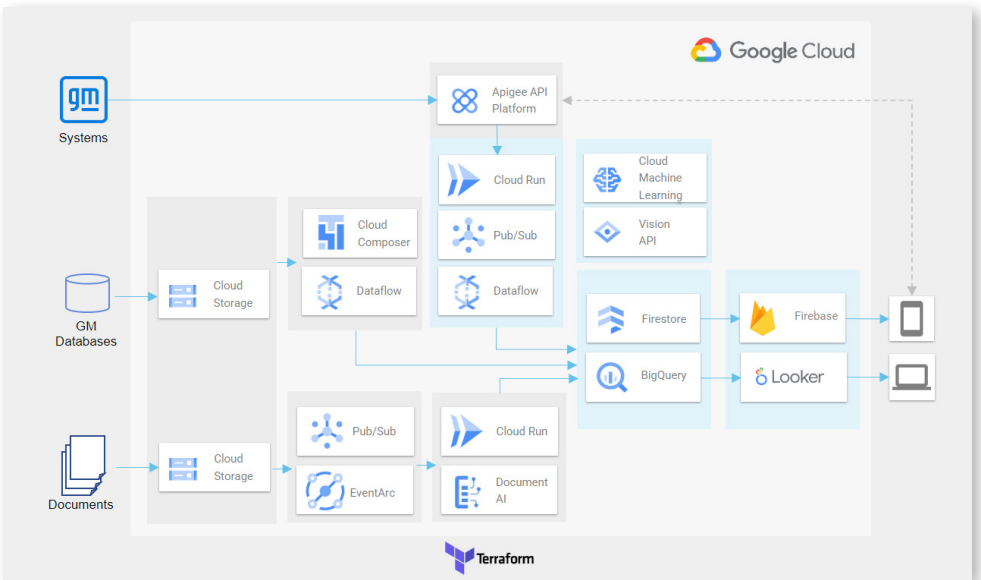
GFT

03

GFT

Success stories

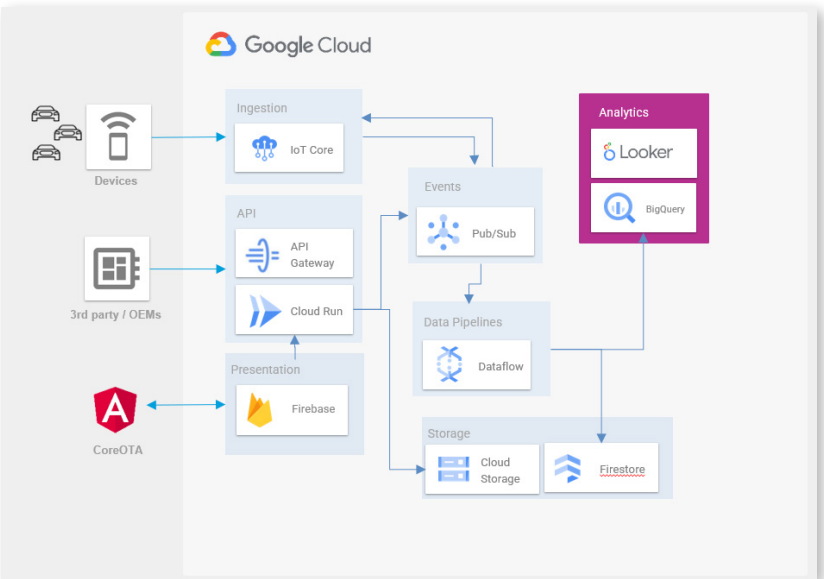
Data & AI for tier-1 automotive industry group



GFT

Success stories

Data & IoT architecture for an automotive industry manufacturer



Hands-on Demo

04

Demo

IoT real-time architecture

Case description

Wake is a startup focused on sustainable architecture. One of its many challenges is reducing electricity consumption in buildings.

To achieve this challenge, they have launched a RFP to monitor the temperature and humidity in order to regulate the optimal conditions of our homes.

Business challenges

- **Part 01 | Monitoring the registered temperature and humidity and display them on a dashboard to help stakeholders to make decisions.**

- Part 02 | Regulate climate control when an inappropriate temperature range is registered during a certain period.

...let's

Brainstorm!

Bibliography

Extended documentation

Apache Beam programming guide

https://beam.apache.org/documentation/programming-guide

Apache Beam basics

https://beam.apache.org/documentation/basics

Dataflow Documentation

https://cloud.google.com/dataflow/docs/concepts

Javier Briones
Data Engineer

jrbz@gft.com