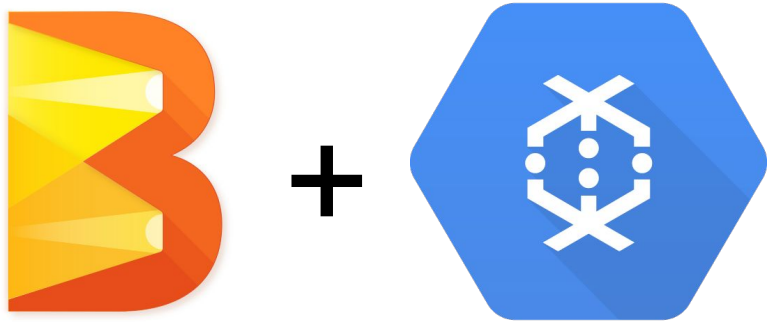
@meteatamel

# No One at Google Uses MapReduce Anymore

## Apache Beam (incubating) and Google Cloud Dataflow

Mete Atamel
Developer Advocate at Google



Google Cloud Platform

# Mete Atamel

Developer Advocate for Google Cloud Platform

@meteatamel

meteatamel.wordpress.com

atamel@google.com

# Agenda

## The Road to Dataflow

MapReduce's Batch Processing, FlumeJava's Clean APIs, MillWheel's Stream Processing

## Dataflow Model

One model unifying batch and streaming

## Dataflow to Apache Beam (incubating)

Evolution of Dataflow into Apache Beam

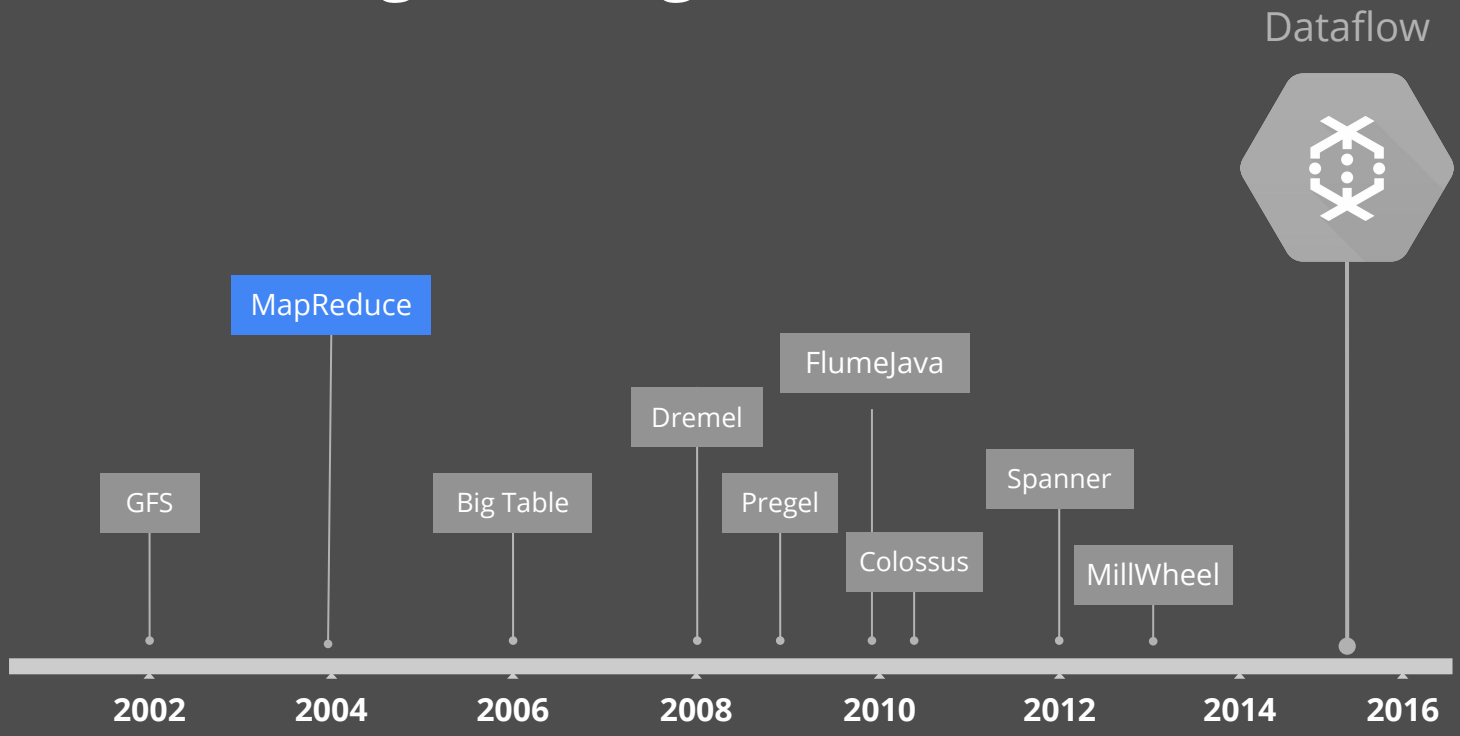
@meteatamel

# The Road to Dataflow

MapReduce's Batch Processing
FlumeJava's High Level APIs
MillWheel's Stream Processing

# Data Processing @ Google





# MapReduce: Batch Processing



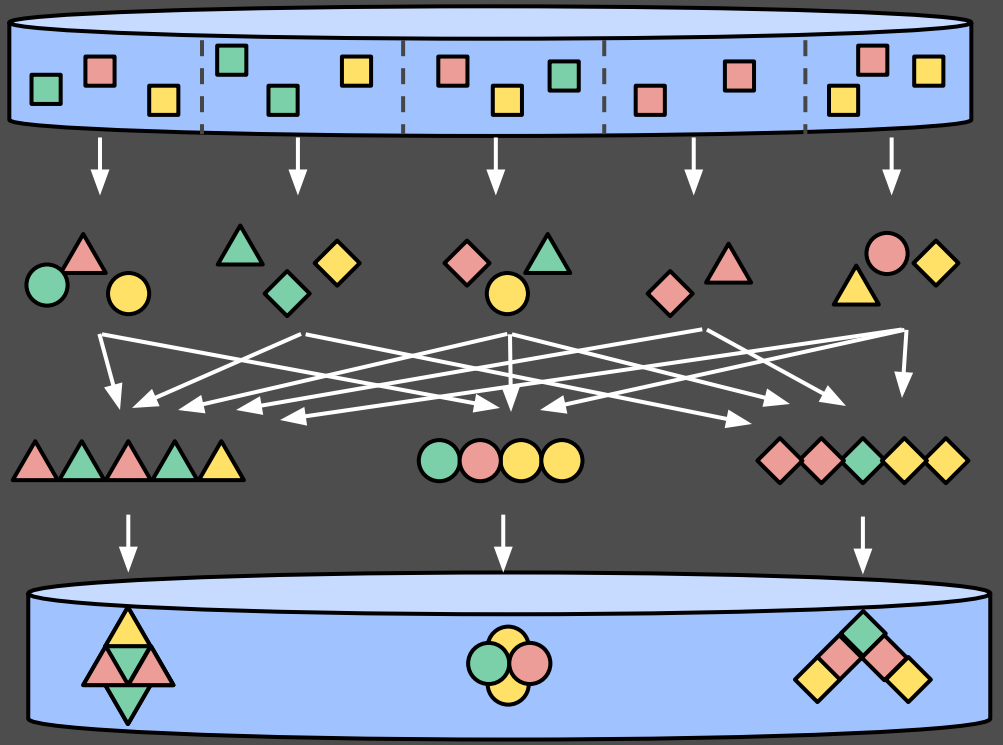
(Prepare)

**Map**

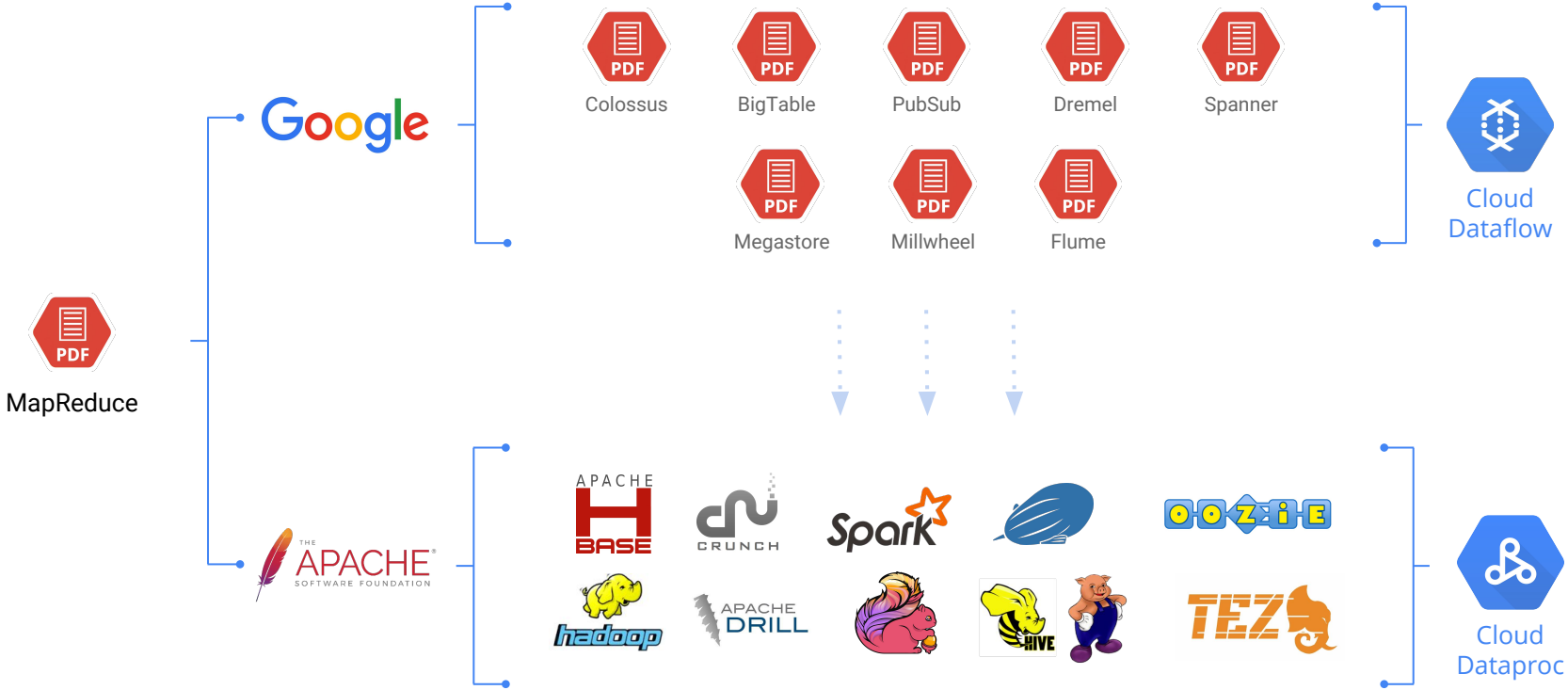
(Shuffle)

**Reduce**

(Produce)

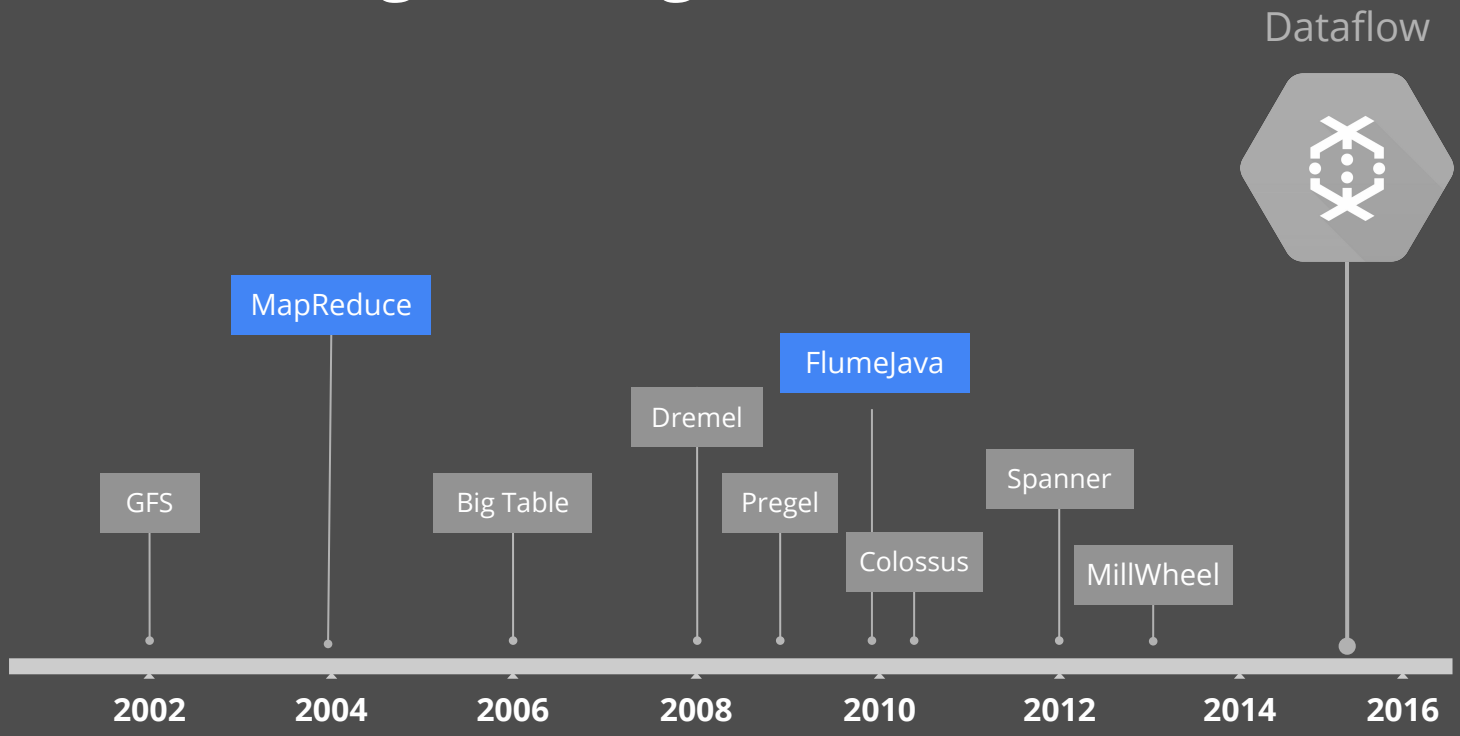


# After MapReduce, Innovation Diverges

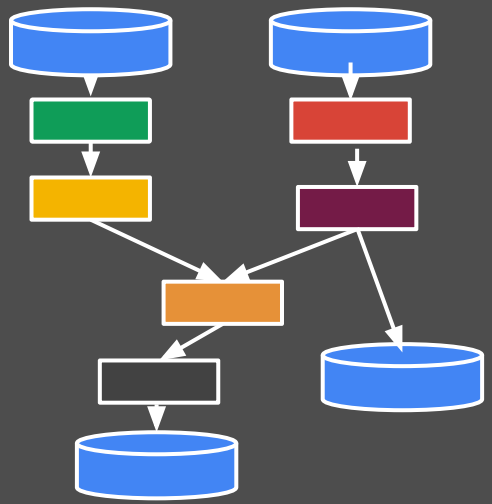


# Data Processing @ Google





# FlumeJava: Easy and Efficient MapReduce Pipelines



- Higher-level API with simple data processing abstractions.
  - Focus on what you want to do to your data, not what the underlying system supports.
- A graph of transformations is automatically transformed into an optimized series of MapReduces.

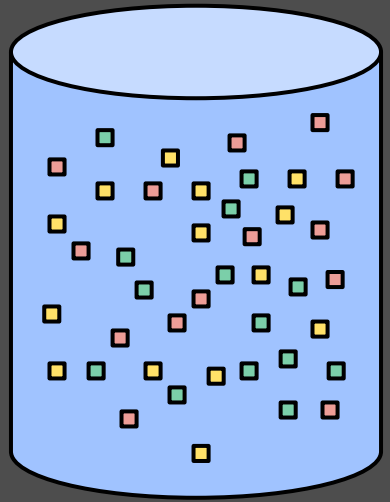
# Example: Computing mean temperature



```
// Collection of raw events
PCollection<SensorEvent> raw = ...;
// Element-wise extract location/temperature pairs
PCollection<KV<String, Double>> input =
    raw.apply(ParDo.of(new ParseFn()))
// Composite transformation containing an aggregation
PCollection<KV<String, Double>> output = input
  .apply(Mean.<Double>perKey());
// Write output
output.apply(BigtableIO.Write.to(...));
```

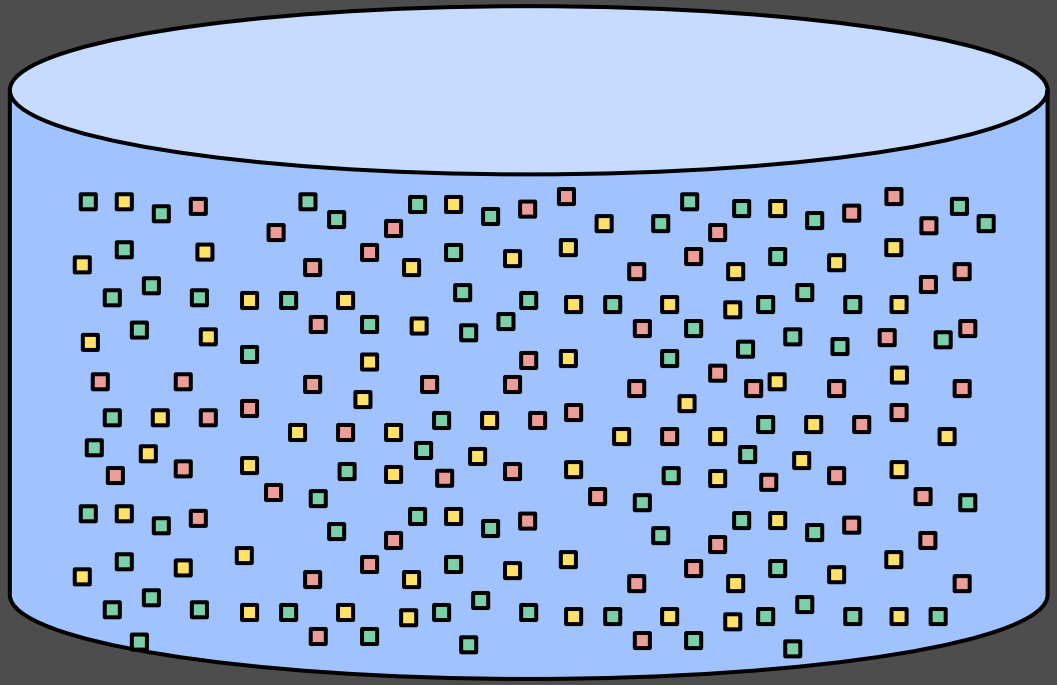
# So, people used FlumeJava to process data...





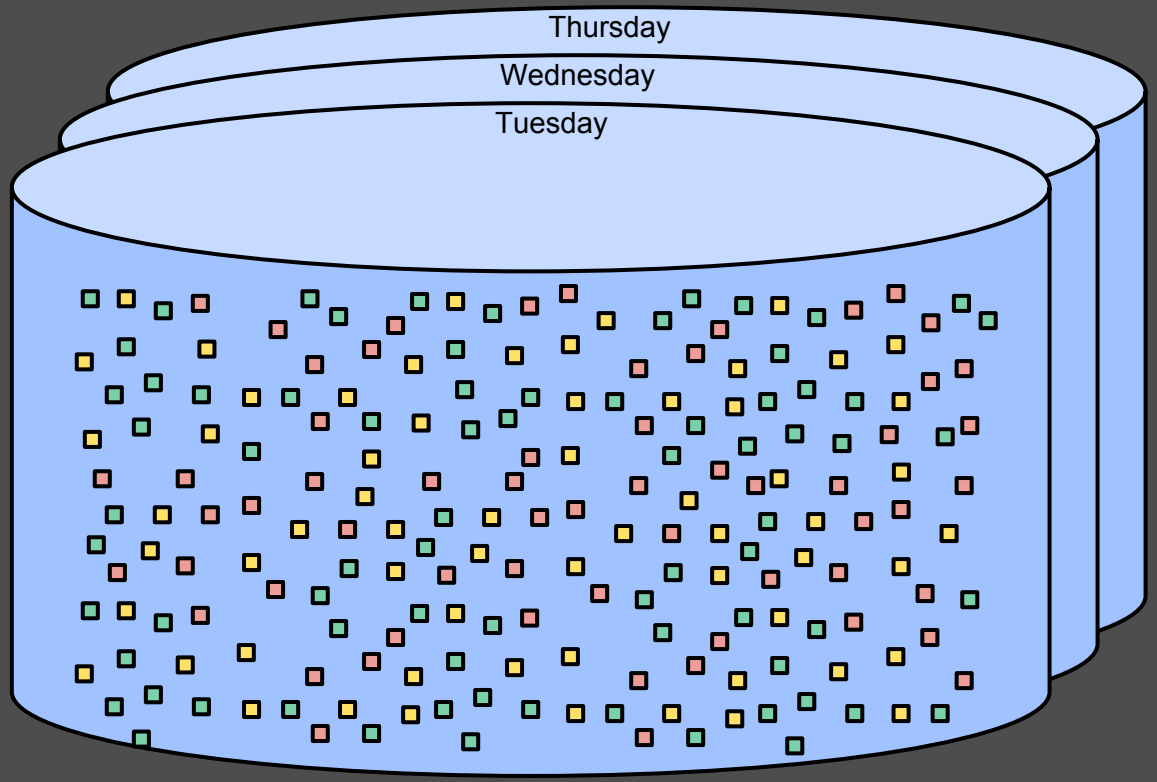
# ...big data...





# ...really, really big...





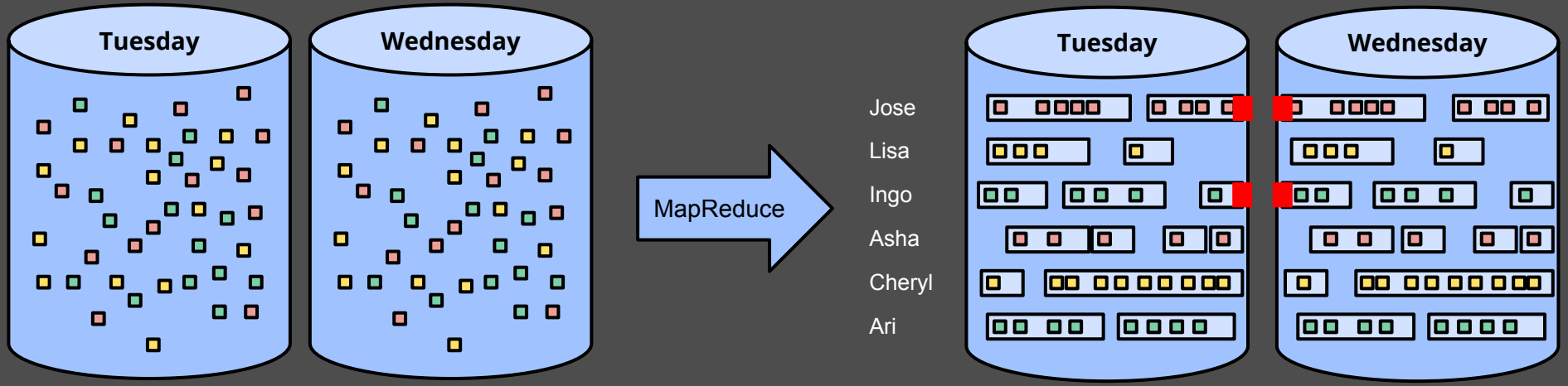
# Batch failure mode #1



**Latency**

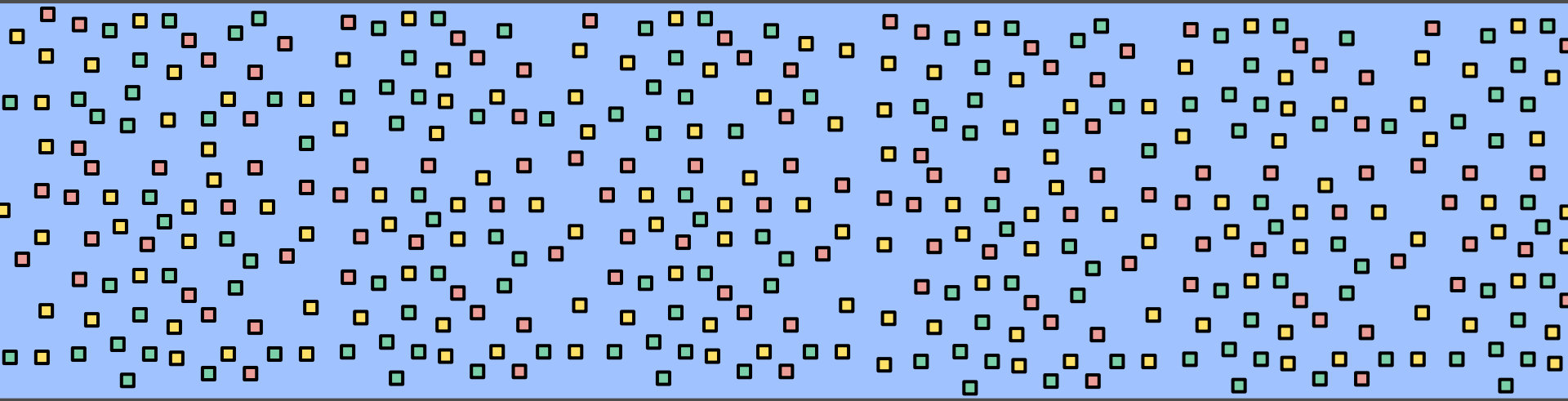
# Batch failure mode #2: Sessions





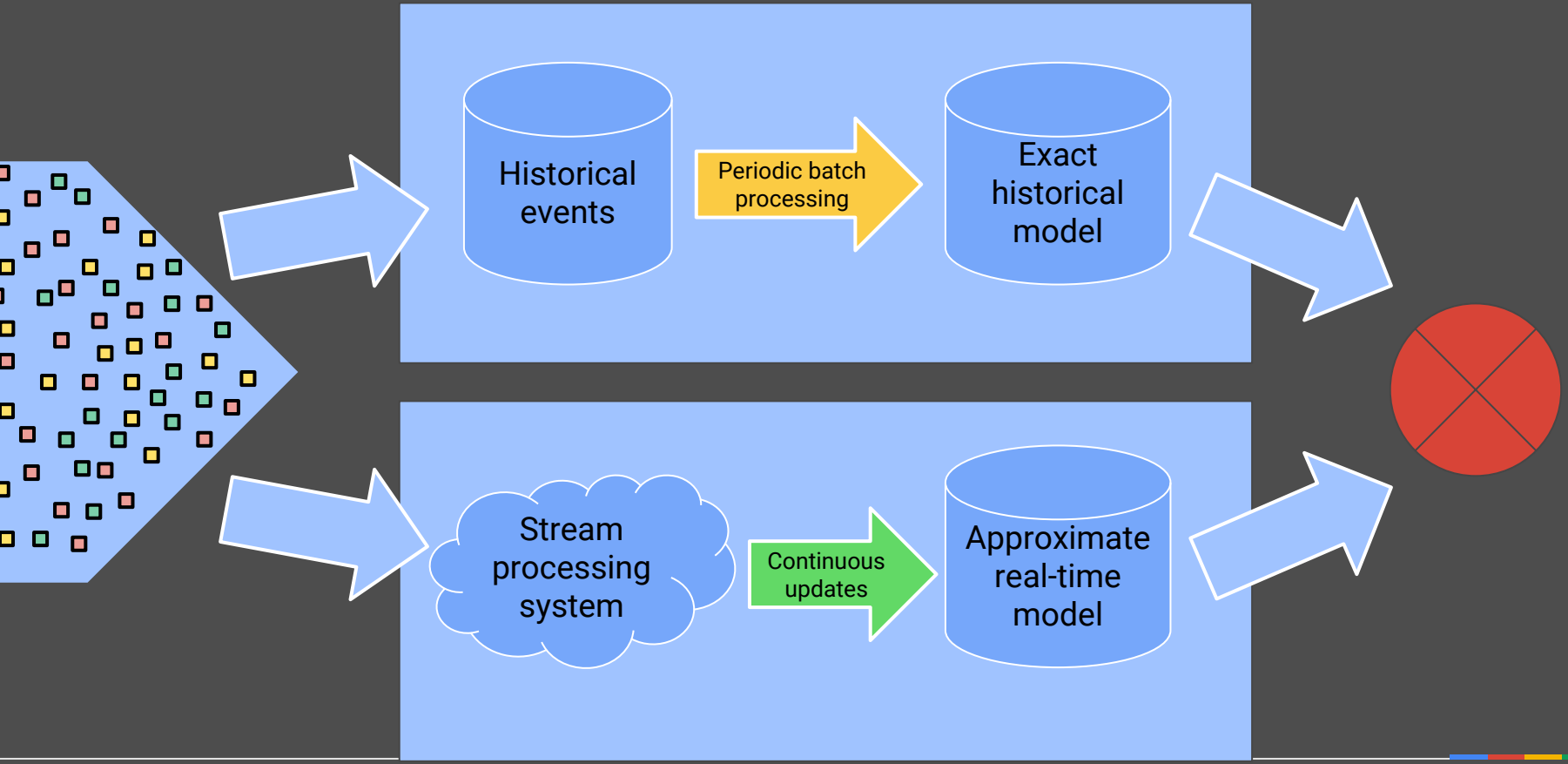
# Continuous & Unbounded

@meteatamel



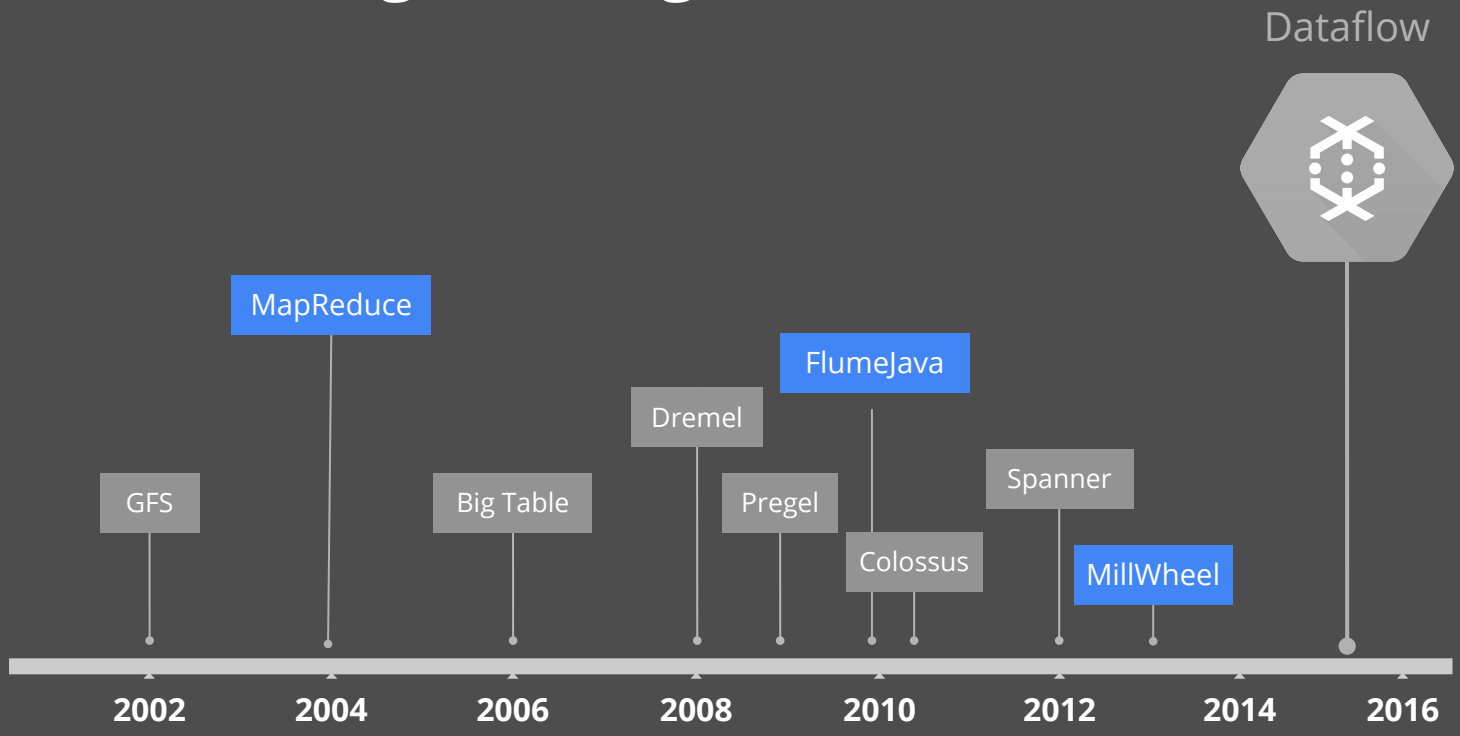
8:00 1:00 9:00 2:00 10:00 3:00 11:00 4:00 12:00 5:00 13:00 6:00 14:00 7:00

# State of the art until recently: Lambda Architecture



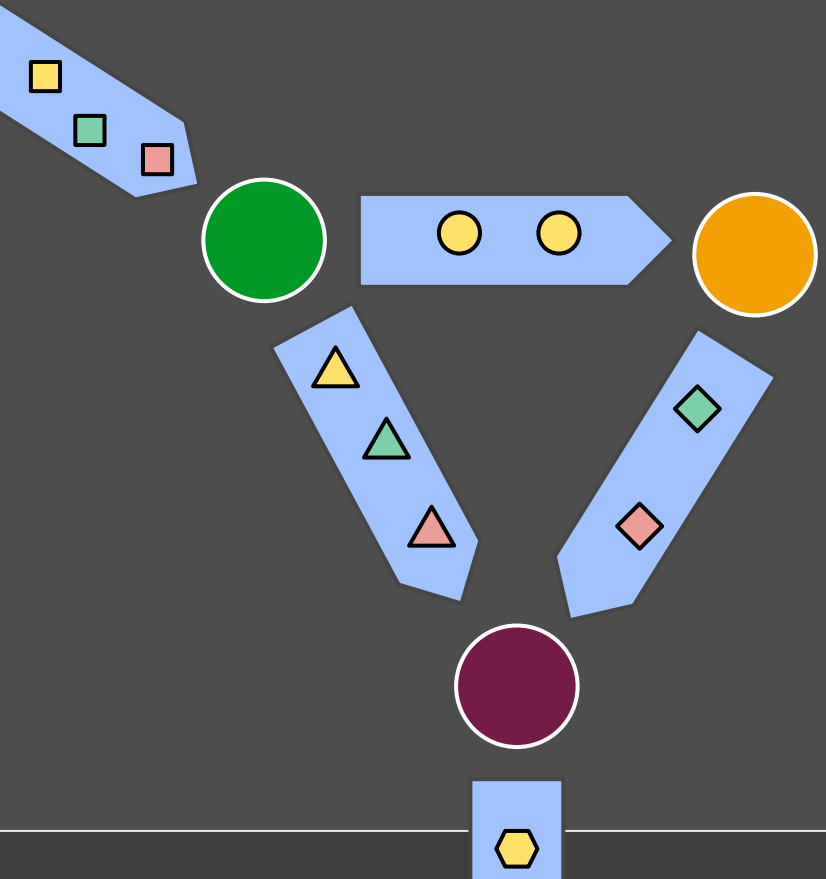
# Data Processing @ Google





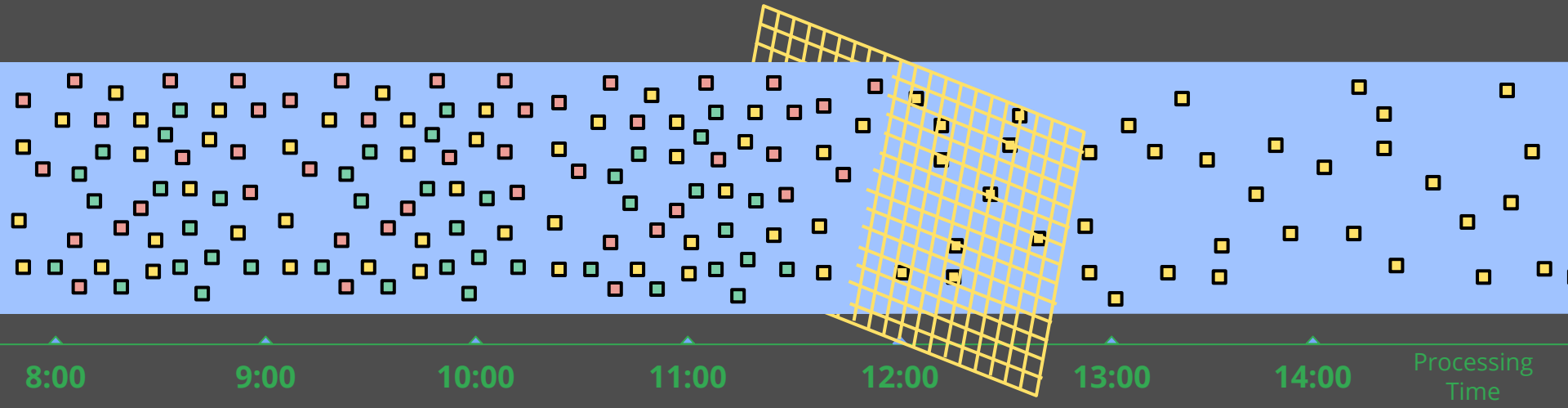
# MillWheel: Streaming Computations



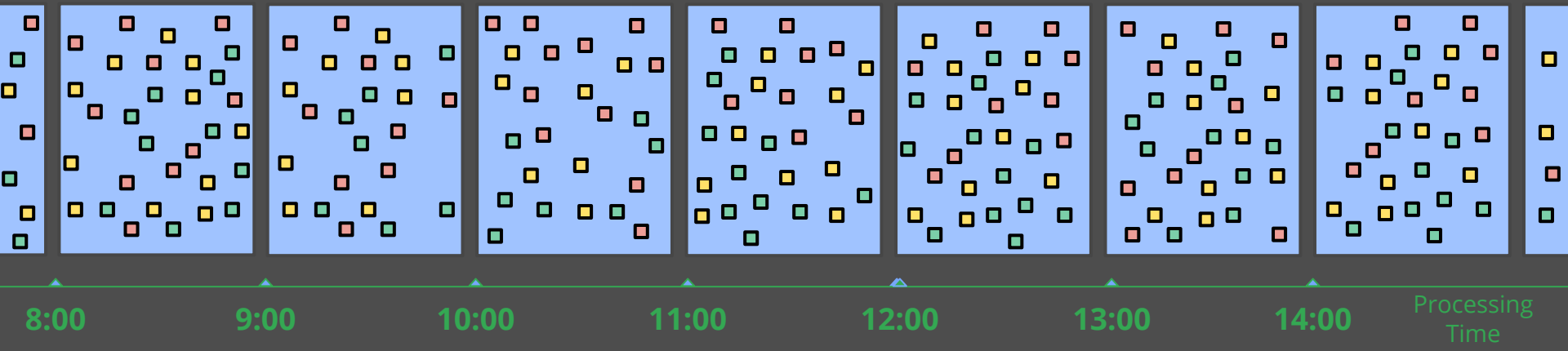


- Framework for building low-latency data-processing applications
- User provides a DAG of computations to be performed
- System manages state and persistent flow of elements

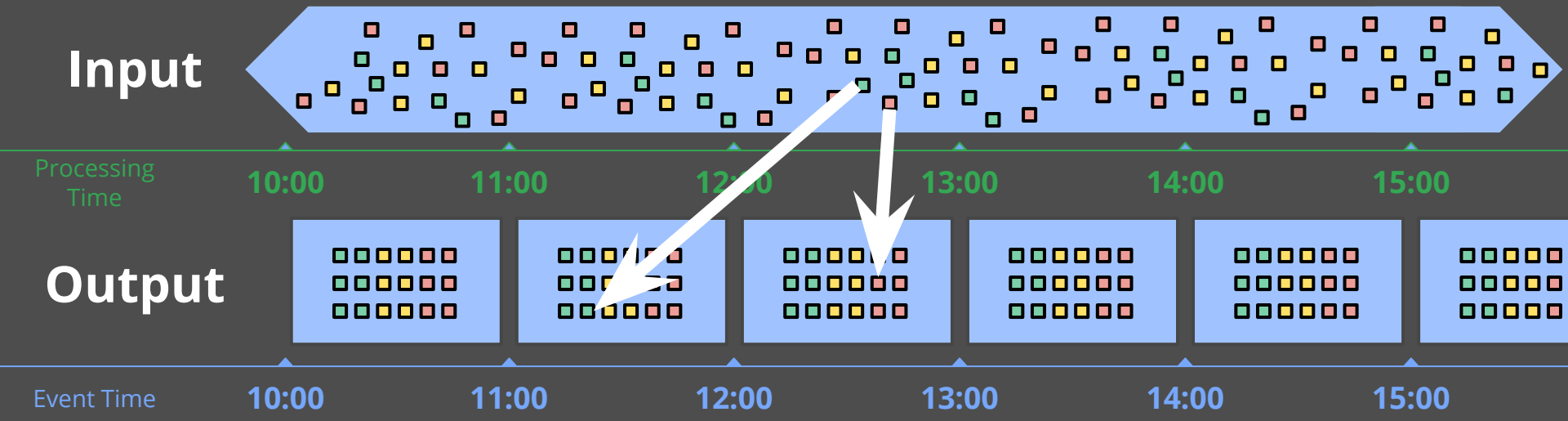
# Streaming Patterns: Element-wise transformations



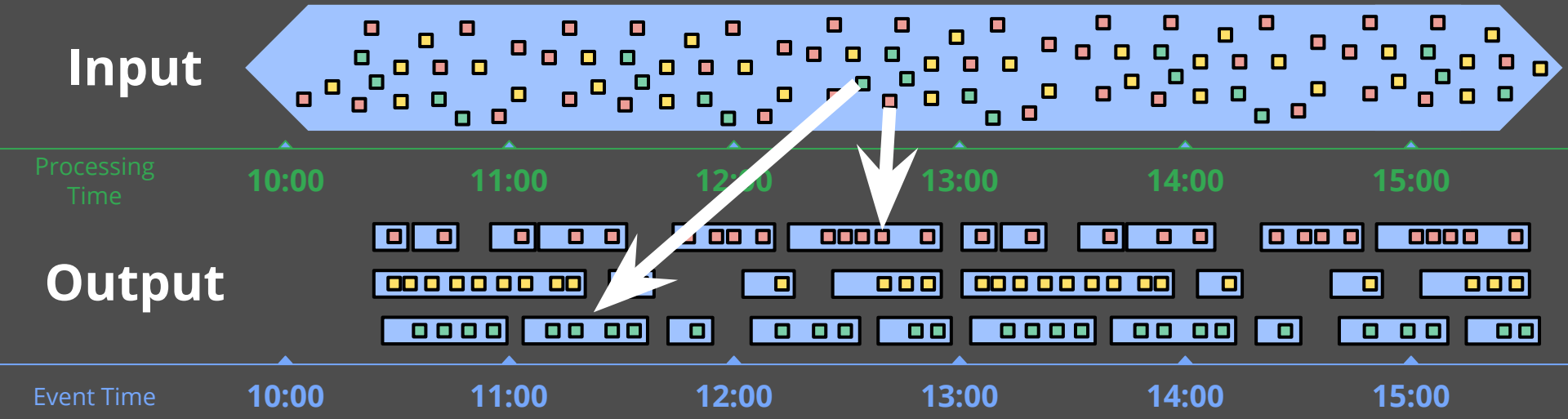
# Streaming Patterns: Aggregating Time Based Windows



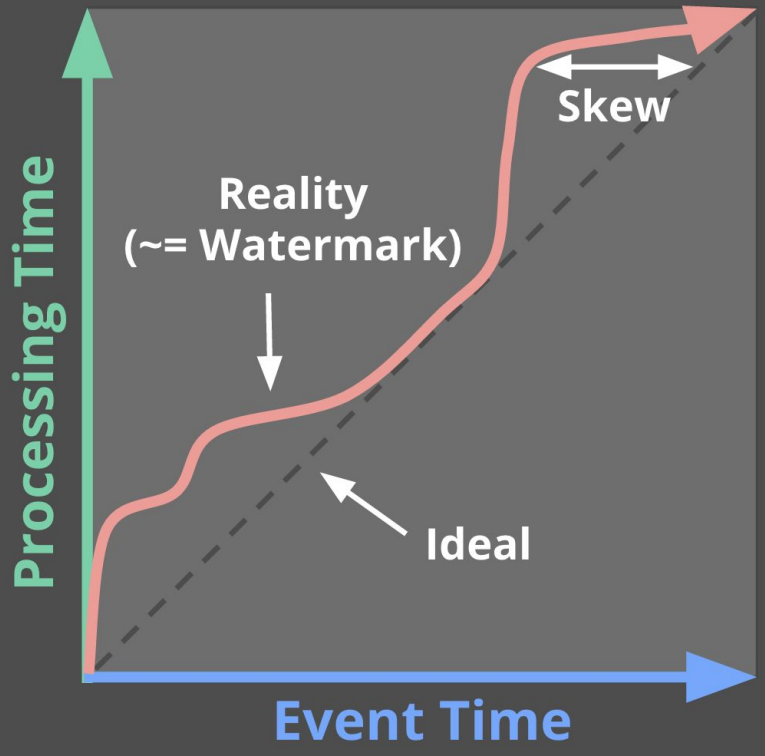
# Streaming Patterns: Event-Time Based Windows



# Streaming Patterns: Session Windows



# Formalizing Event-Time Skew




Watermarks describe event time progress.

*"No timestamp earlier than the watermark will be seen"*

Often heuristic-based.

Too Slow? Results are *delayed*.
Too Fast? Some data is *late*.

# Streaming or Batch?

1+1=2

**Completeness**

**Latency**

$$$

**Cost**

*Why not both?*

@meteatamel

# Dataflow Model

One model unifying batch and streaming

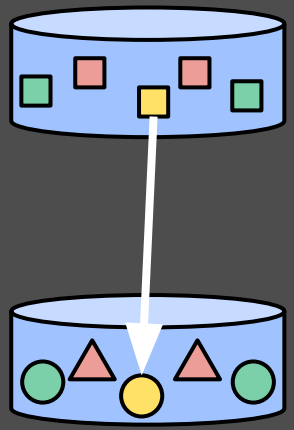
What are you computing?

Where in event time results are calculated?

When in processing time are results materialized?

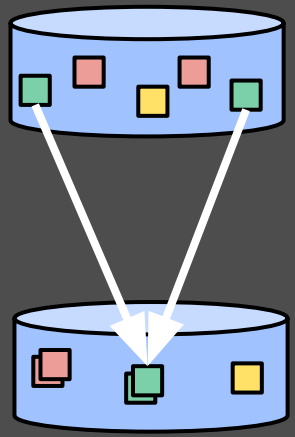
How do refinements relate?

# What are you computing?



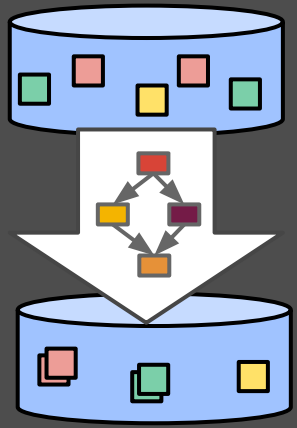
**Element-Wise**

**ParDo**



**Aggregating**

**GroupByKey, Combine**



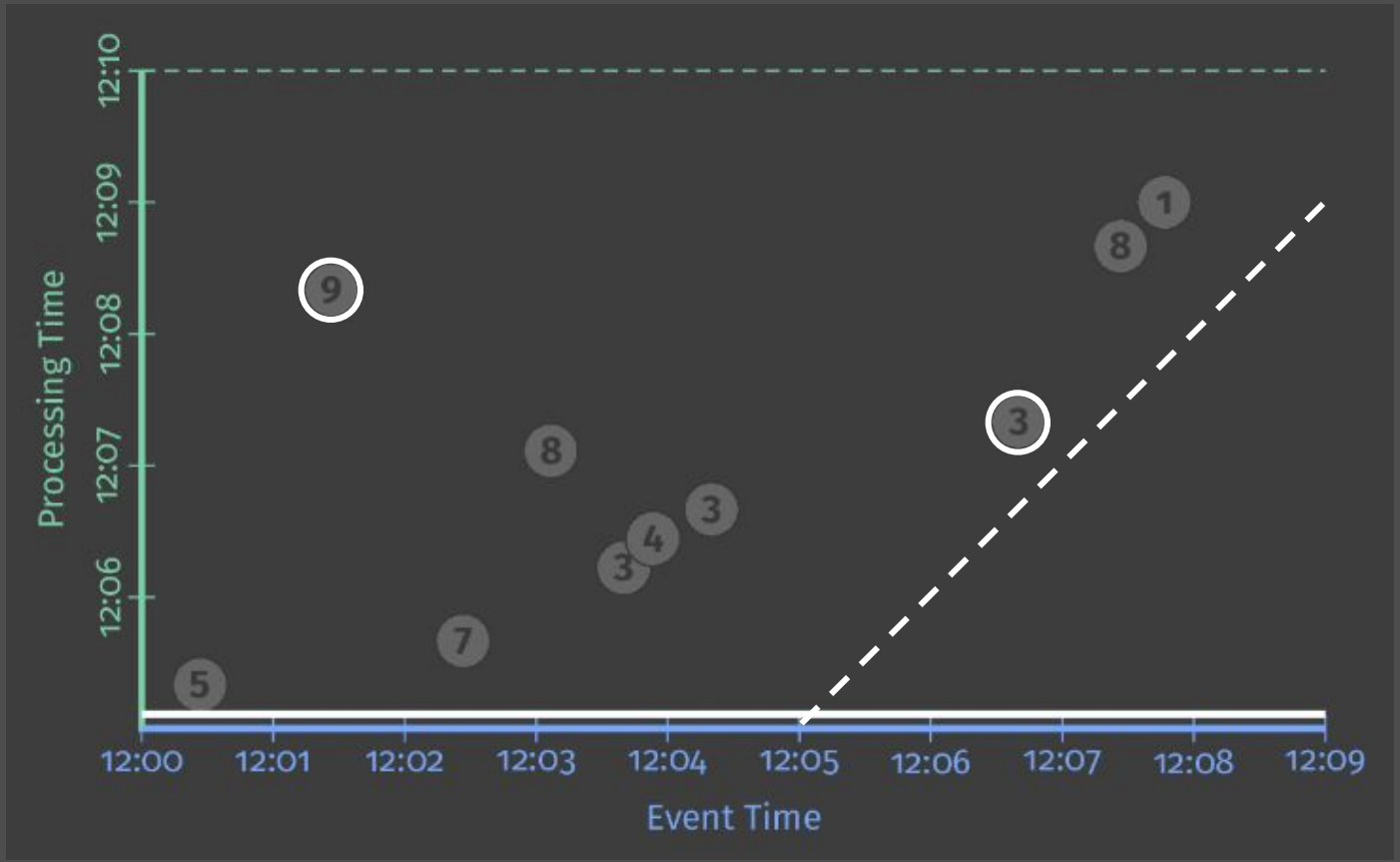
**Composite**

**ParDo + Count + ParDo**

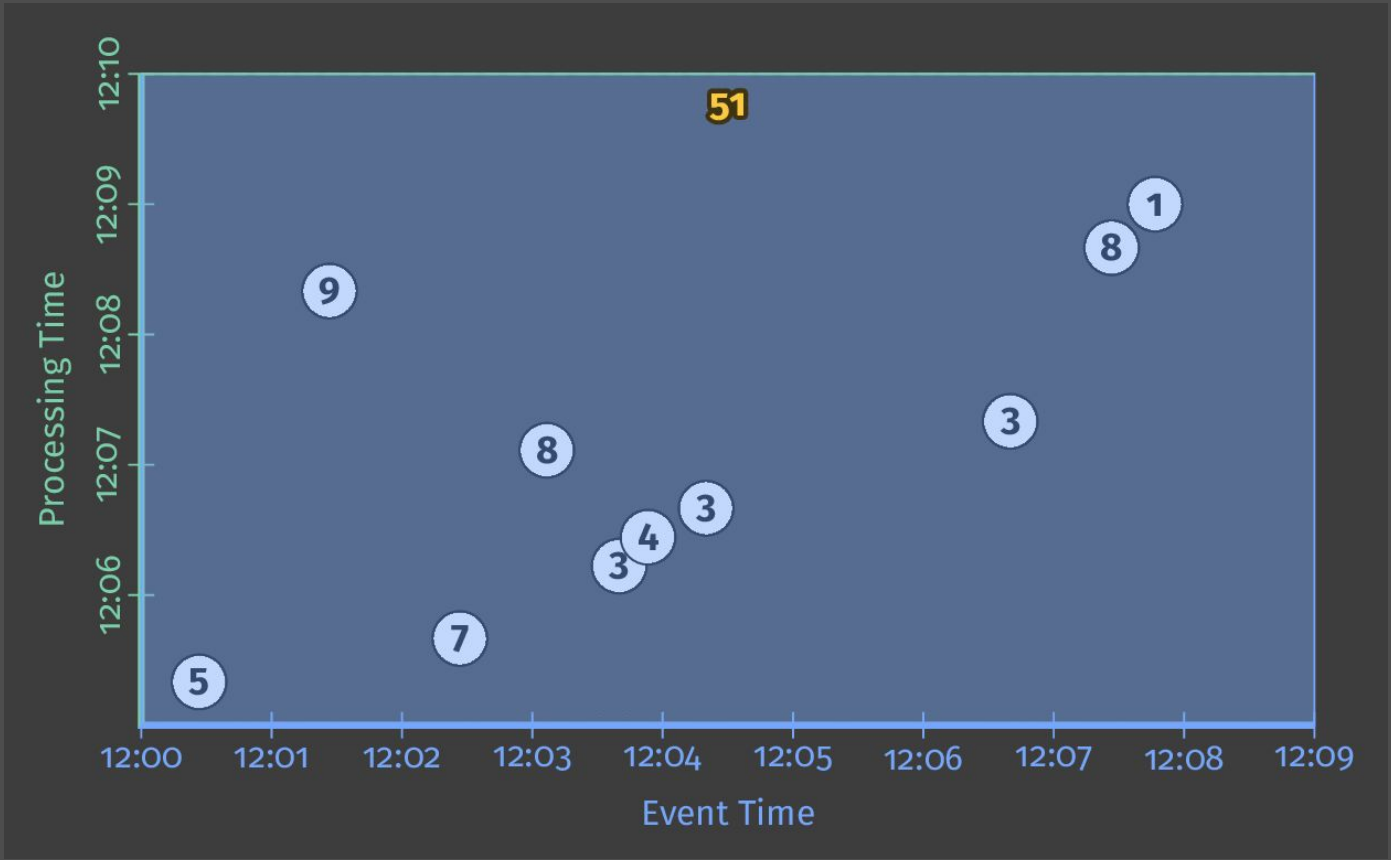
# What: Computing Integer Sums

```
// Collection of raw log lines
PCollection<String> raw = IO.read(...);

// Element-wise transformation into team/score pairs
PCollection<KV<String, Integer>> input =
     raw.apply(ParDo.of(new ParseFn()));

// Composite transformation containing an aggregation
PCollection<KV<String, Integer>> scores =
     input.apply(Sum.integersPerKey());
```

# What: Computing Integer Sums

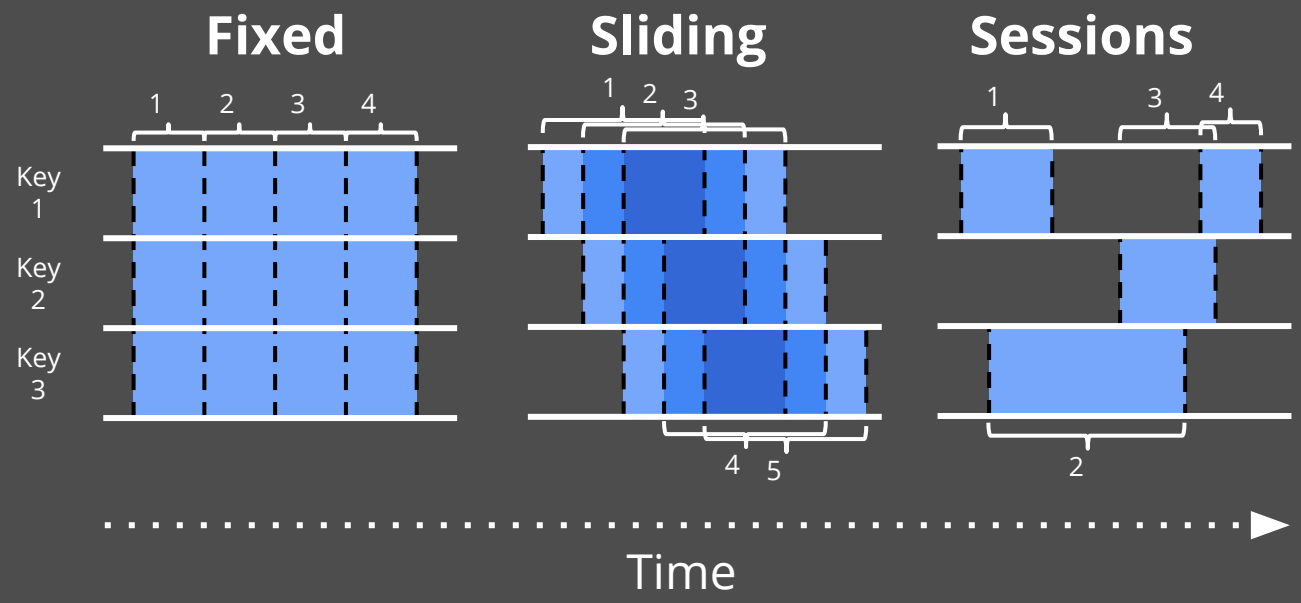


# What: Computing Integer Sums



# Where in event time?

Windowing divides data into event-time-based finite chunks.

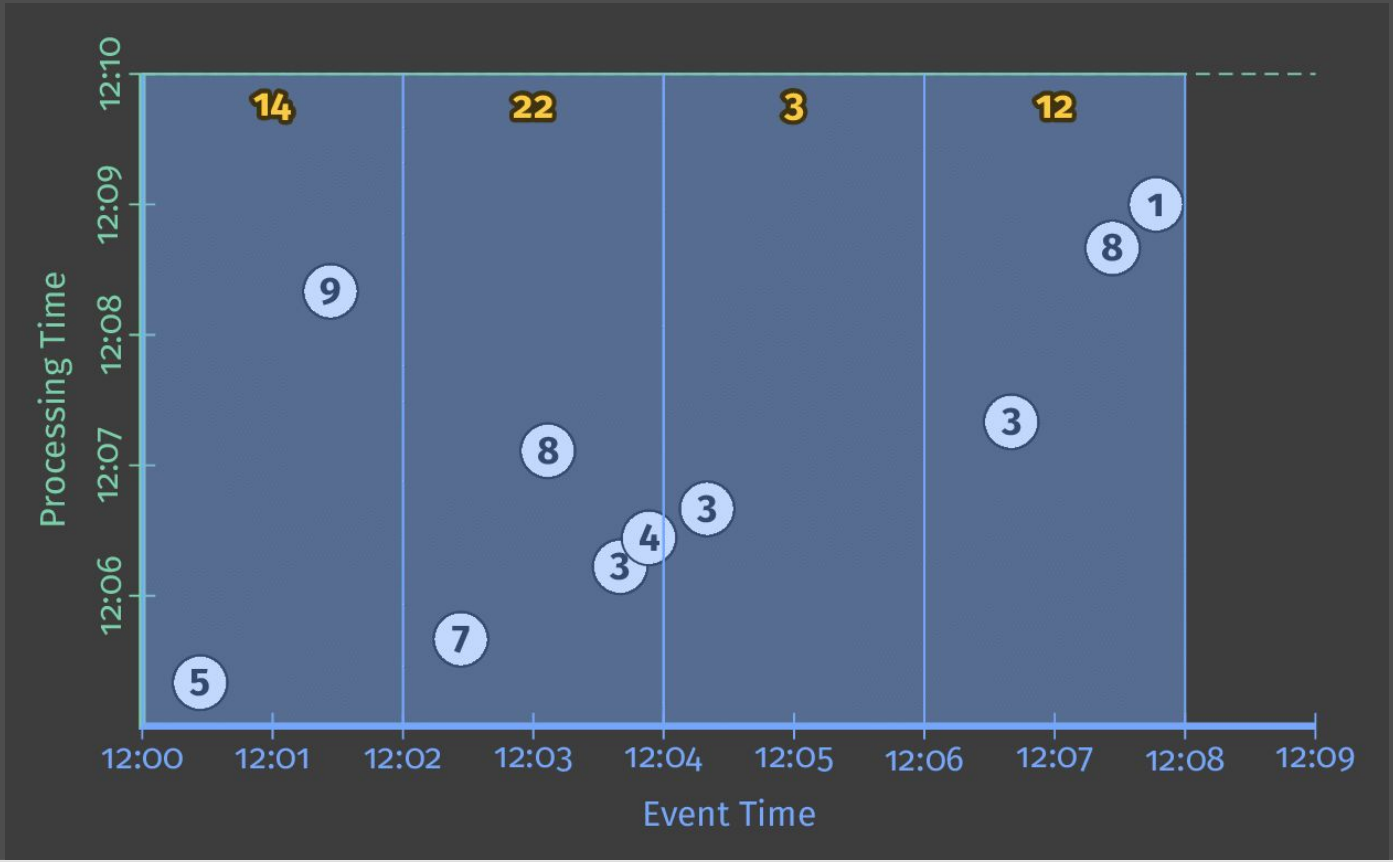



Often required when doing aggregations over unbounded data.

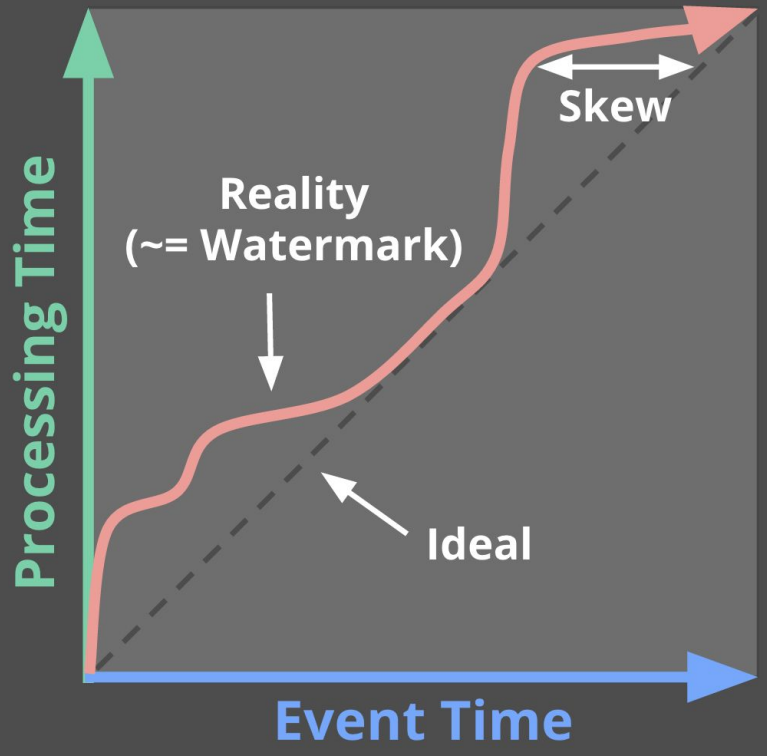
# Where: Fixed 2-minute Windows

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

# Where: Fixed 2-minute Windows



# When in processing time?

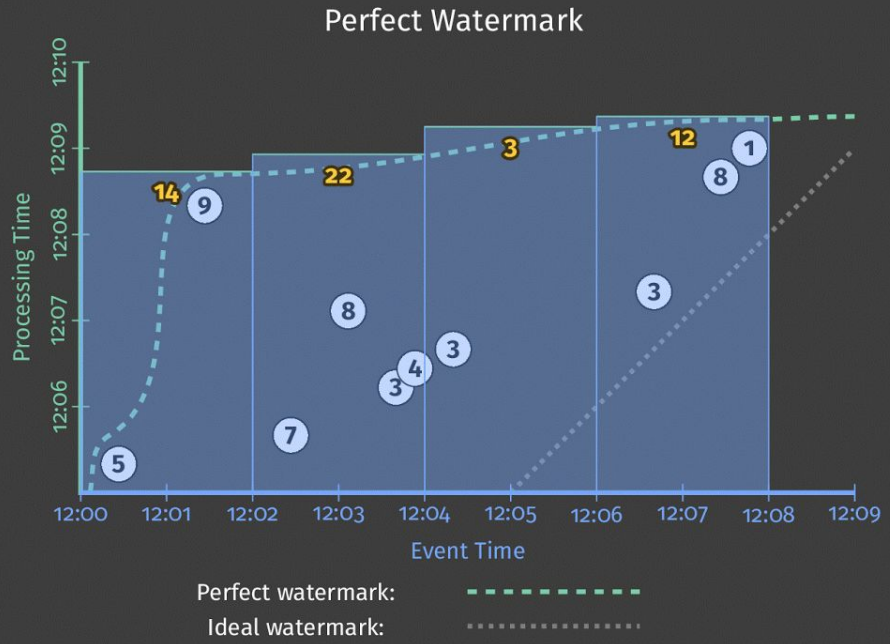


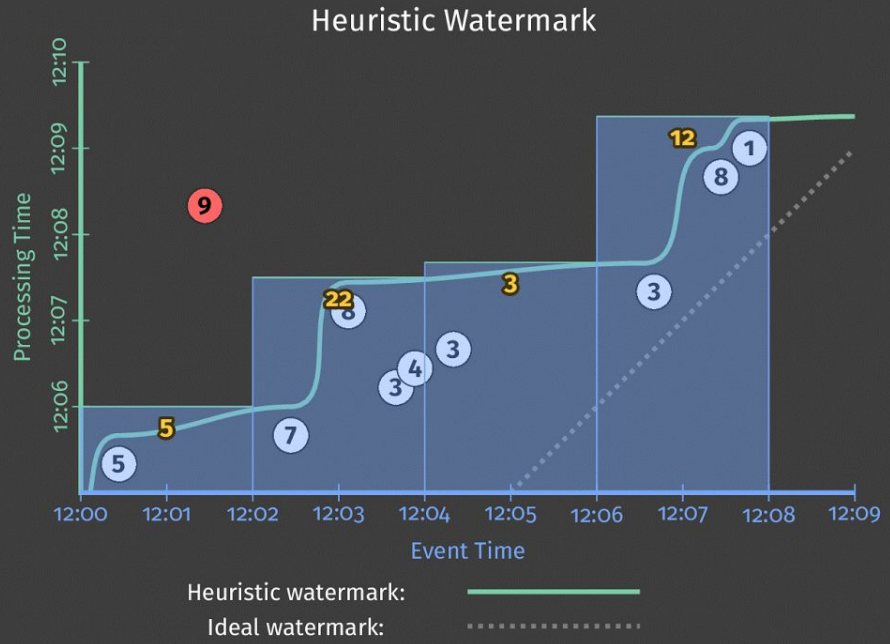

- Triggers control when results are emitted.
- Triggers are often relative to the watermark.

# When: Triggering at the Watermark

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

# When: Triggering at the Watermark

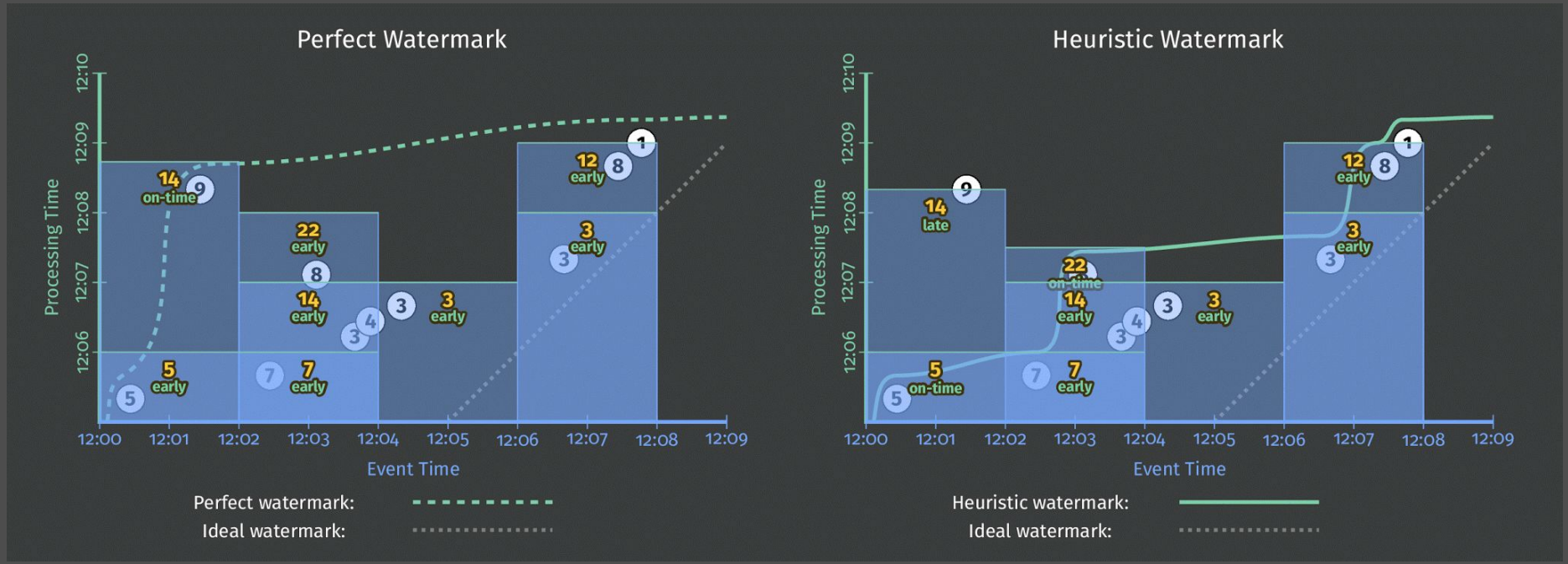


# When: Early and Late Firings

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

# When: Early and Late Firings



# How do refinements relate?

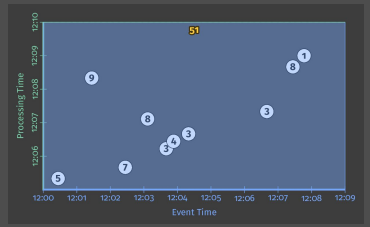
- How should multiple outputs per window accumulate?
- Should we emit the **running sum**, or **only the values that have come in since the last result**?

(Accumulating & Retracting not yet implemented in Apache Beam.)

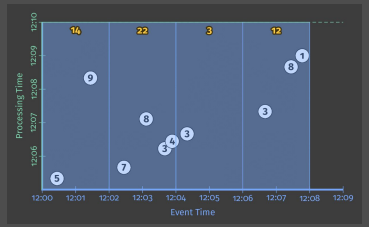
# How: Add Newest, Remove Previous

```
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());
```

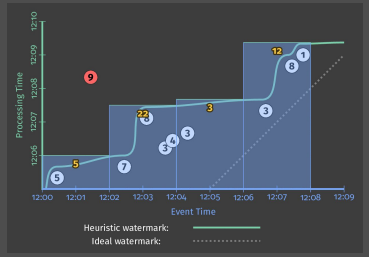
# Customizing What When Where How



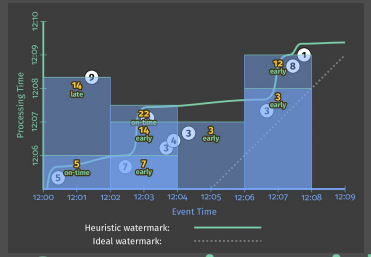

## 1.Classic Batch



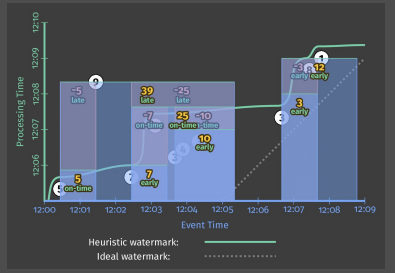

## 2. Batch with Fixed Windows




## 3. Streaming




## 4. Streaming with Speculative + Late Data




## 5. Streaming With Accumulations

@meteatamel

# Dataflow to Apache Beam (incubating)

Evolution of Dataflow into Apache Beam

# The Dataflow Model & Cloud Dataflow

**Dataflow Model & SDKs**

A unified model for
batch and stream processing

**Google Cloud Dataflow**

No-ops, fully managed service

# The *Beam* Model & Cloud Dataflow

**Apache Beam**

a unified model for
batch and stream processing
***supporting multiple runtimes***

**Google Cloud Dataflow**

***A great place to run Beam***

# What is Part of Apache Beam?

1. The Beam Model: **What** / **Where** / **When** / **How**

2. SDKs for writing Beam pipelines -- starting with Java

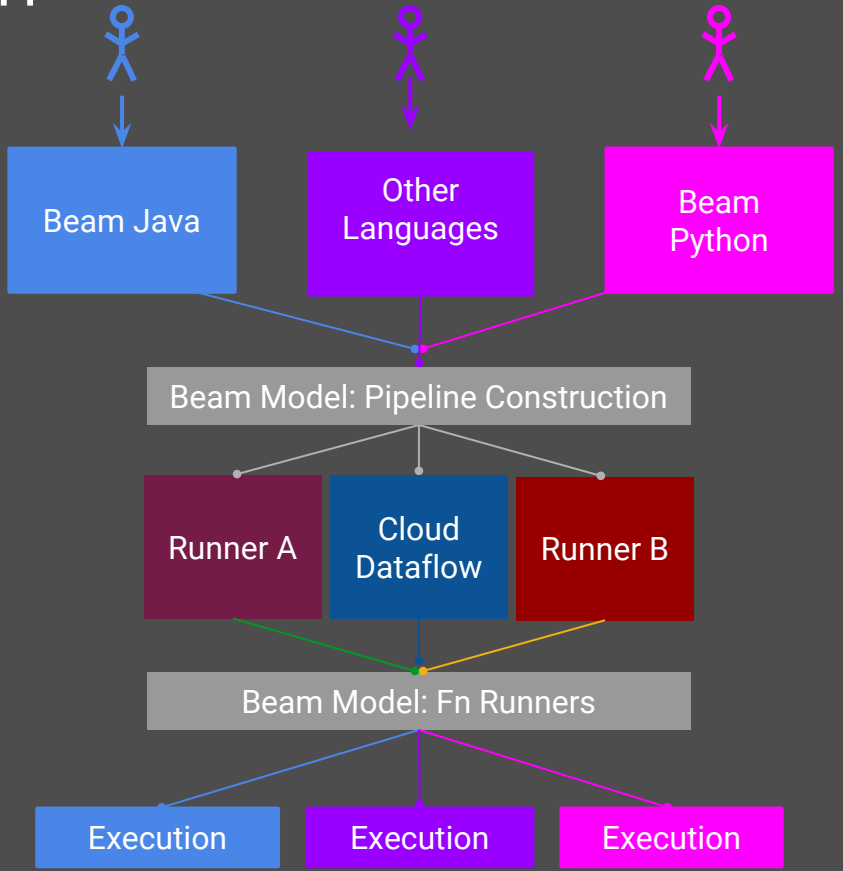
3. Runners for Existing Distributed Processing Backends
   - Apache Flink (thanks to data Artisans)
   - Apache Spark (thanks to Cloudera)
   - Google Cloud Dataflow (fully managed service)
   - Local (in-process) runner for testing

# Apache Beam Technical Vision

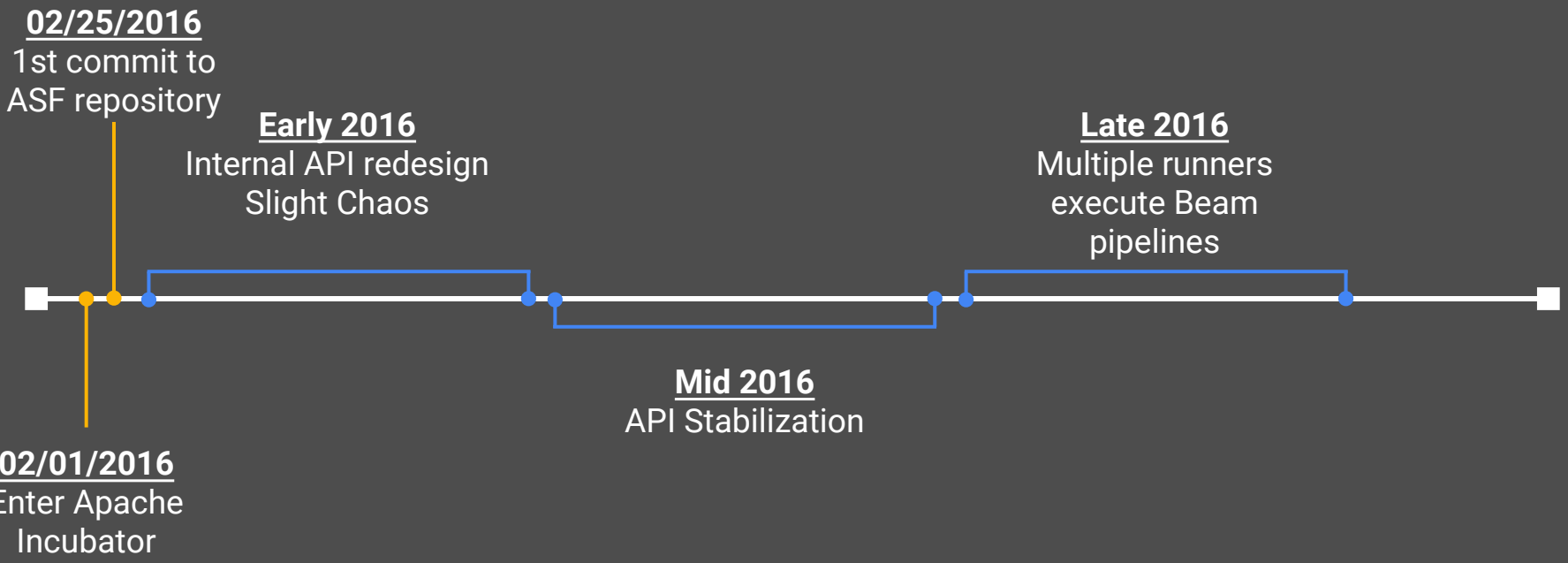
1. **End users:** who want to write pipelines in a language that's familiar.

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



# Apache Beam Roadmap

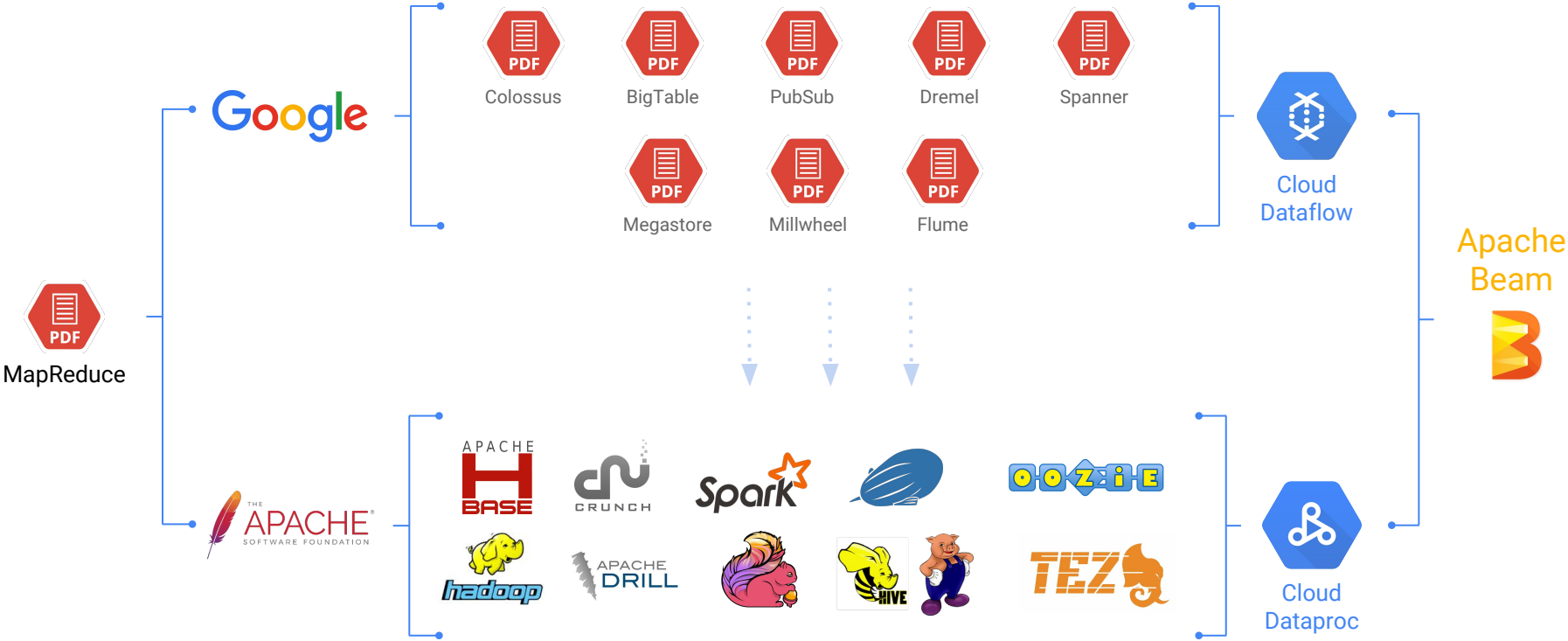


# Growing the Beam Community

***Collaborate*** - Beam is becoming a community-driven effort with participation from many organizations and contributors

***Grow*** - We want to grow the Beam ecosystem and community with active, open involvement so Beam is a part of the larger OSS ecosystem

# Data Processing with Apache Beam



# Thank You

cloud.google.com/dataflow
beam.incubator.apache.org
@ApacheBeam

Mete Atamel
@meteatamel
meteatamel.wordpress.com
atamel@google.com