
BEAM
SUMMIT

Integration of Batch and Streaming data processing with Apache Beam

BEAM SUMMIT NYC 2025

Yoichi Nagai

Data Engineer @ Mercari, Inc.

Agenda

- Mercari Pipeline
  - What is Mercari Pipeline ?
  - Configuration management
- Stateful Processing
  - Aggregation - Beam SQL
  - DoFn State API
- Stateful Processing with External Data sources
  - Processing with Cloud Bigtable

Data Processing in Mercari

Mercari has adopt Microservices architecture.
Various teams develop and operate data pipelines.

- Batch processing and integration of DB data
- Near-real-time campaigns and fraud detection

However, there are also issues.

- Each team needs to develop and operate pipelines.
  - Not all application engineers are familiar with pipeline development
- Many different databases, requires a lot of pipelines for data integration.

Tool that enable easy definition and deployment of data pipeline was developed

-> Mercari Pipeline

What is Mercari Pipeline?

Tool to define Apache Beam pipeline in YAML or JSON

```
system:
    system configuration
options:
    pipeline option configuration
sources:
    data source definition
transforms:
    data processing definition
sinks:
    data destination definition
failures:
    error data destination definition
```

Published as OSS (mercari/pipeline)

Almost the same configurations as Beam YAML

sources/sinks	transforms
• storage	• beamsql
• jdbc	• aggregation
• bigquery	• select
• pubsub	• bigtable
• etc…	• etc…

ref blog: Data operation with Cloud Spanner using Mercari Dataflow Template

Mercari Pipeline System Configuration

```
system:
    args:
        project: myproject-dev
        today: ${utils.datetime.currentDate()}
    context: daily
    imports:
        - files:
          - gs://xxx/a.yaml
          - gs://xxx/b.yaml
```

args:

Definition to rewrite variables specified in the config definition at runtime.
It is also possible to overwrite with specified variables at runtime.

context:

Specify the steps to be executed in the config definition.

imports:

Define config files separately and merge them at runtime.

```
system:
  context: train
sources:
  - name: source
    module: bigquery
    tags: [train]
    parameters:
      table: myproject.mydataset.mytable
  - name: source
    module: pubsub
    tags: [predict]
    parameters:
      format: avro
      subscription: projects/xxx/subscriptions/yyy
transforms:
  - name: features
    module: select
    tags: [train,predict]
    inputs: [source]
    parameters:
      select: …
sinks:
  - name: sink
    module: bigquery
    tags: [train]
    inputs: [features]
  - name: sink
    module: pubsub
    tags: [predict]
    inputs: [features]
```

Configuration for Batch and Streaming

Set tags for each scenario in each module.

By specifying system.context
(or the context parameter at runtime),
pipeline job can be executed only with modules that have tags matching the specified context.

Use Cases

- Training and prediction processes in ML pipelines
- Regular execution and backfilling at the first run or when recovering from a failure

Mercari Pipeline Resources

Dataflow

… or other runners

Deploy

Execute in Production

Web UI

as REST API Server

Develop & Debug
by Browser

Server

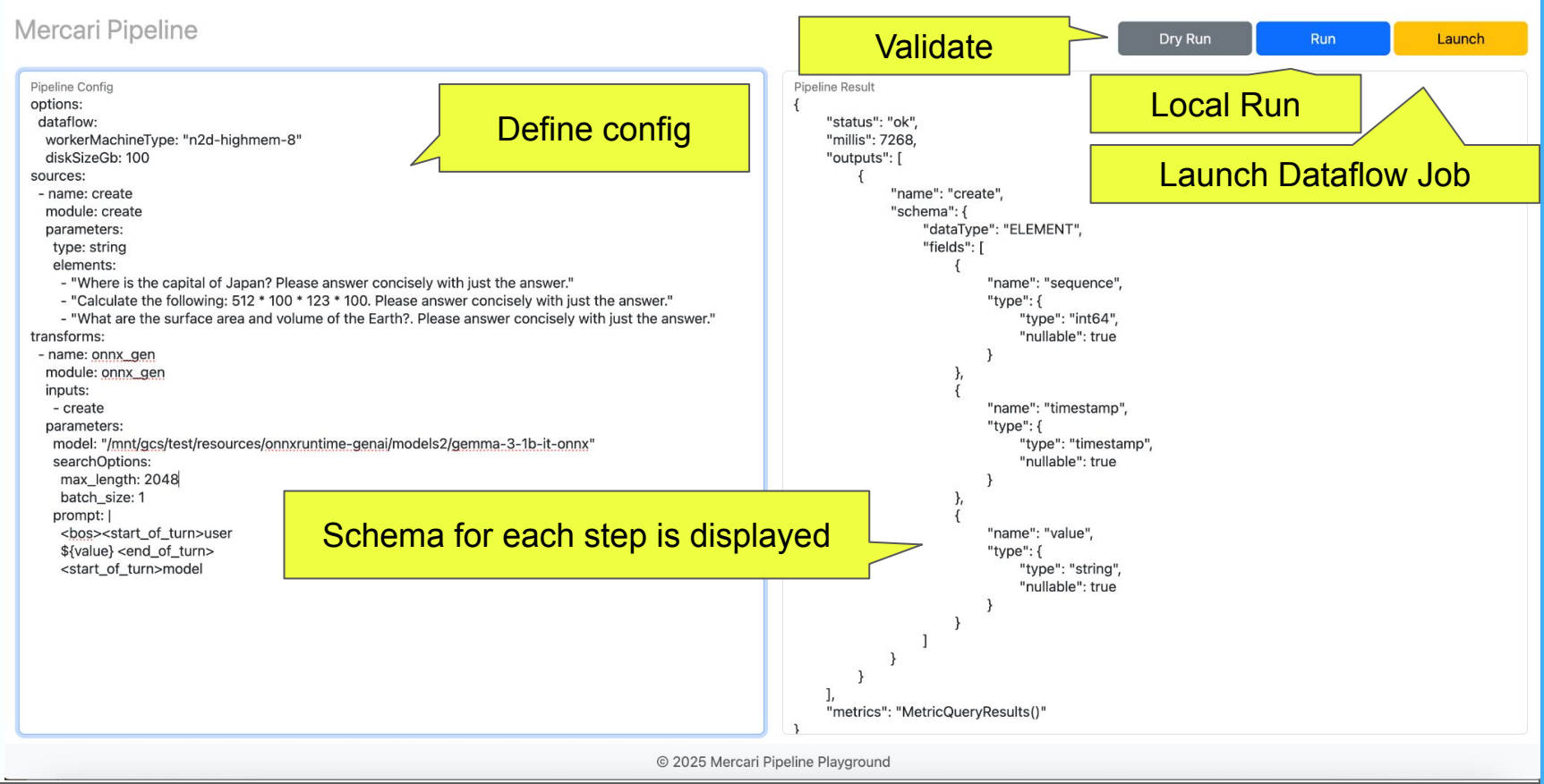
Develop & Debug
by Local Execution

as MCP Server

Direct

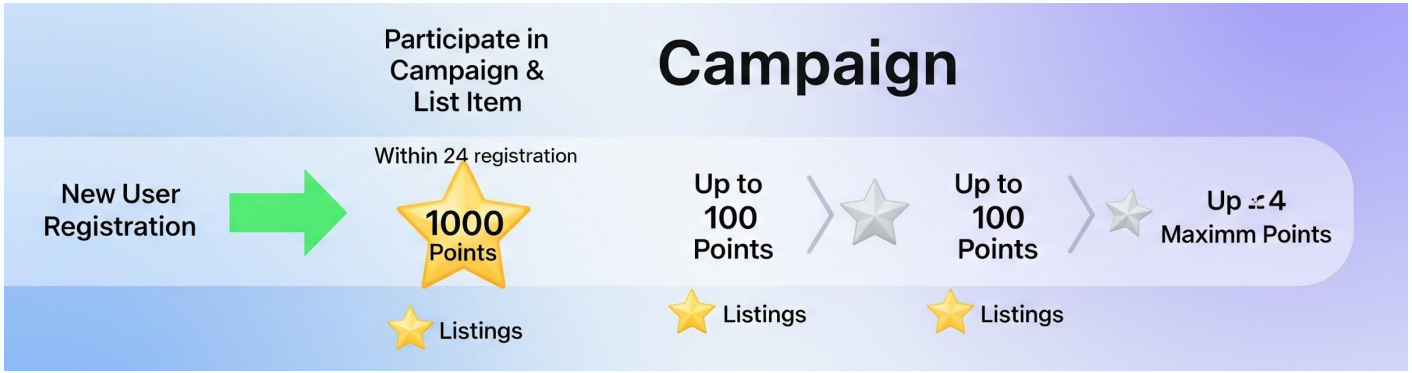
MCP Host

Mercari Pipeline - Server UI



Stateful Processing Use-case in Mercari

Some campaigns require stateful streaming processing with multiple types of events



The following user events need to be integrated and evaluated

- User registration
- Campaign registration
- Listing

Stateful Processing - Beam SQL

```
WITH withAdditionalFields AS (
  SELECT
    *,
    IF(event_type="event_registration", timestamp, NULL) AS registration_time,
    IF(event_type="event_entry_campaign" AND campaign_name="CampaignA", timestamp, NULL) AS entry_time,
    IF(event_type="event_entry_campaign", campaign_name, NULL) AS campaign_name,
    IF(event_type="event_listing_item", timestamp, NULL) AS listing_time,
    IF(event_type="event_listing_item", item_id, NULL) AS listing_item_id
  FROM Inputs
),
grantedIncentive AS (
  SELECT
    user_id,
    MAX(registration_time) AS registration_time,
    MAX(entry_time) AS entry_time,
    MAX(listing_time) AS latest_listing_time,
    MAX(item_id) AS latest_listing_item_id,
  FROM withAdditionalFields
  WHERE
    event_type IN UNNEST(["event_registration", "event_entry_campaign", "event_listing_item"])
  GROUP BY user_id
  HAVING
    entry_time > registration_time
    AND latest_listing_time > entry_time
    AND TIMESTAMP_DIFF(entry_time, registration_time, HOUR) < 24
    AND TIMESTAMP_DIFF(latest_listing_time, registration_time, HOUR) BETWEEN 0 AND 23
)
SELECT
  user_id,
  COUNT(latest_listing_item_id) AS listing_count,
  CASE COUNT(latest_listing_item_id)
    WHEN 1 THEN 1000
    WHEN 2 THEN 100
    WHEN 3 THEN 100
    WHEN 4 THEN 100
    WHEN 5 THEN 100
    ELSE 0 END AS incentive
  CURRENT_TIMESTAMP() AS granted_timestamp
FROM grantedIncentive
GROUP BY user_id
HAVING
  listing_count <= 5
```

Retaining state as a result of aggregation functions

Specify the windowing strategy based on the following requirements.

- State must be retained throughout the campaign period
- Conditions must be evaluated each time an event is added

```
.apply( name: "WindowingStrategy", Window
        .<MElement>into(new GlobalWindows())
        .accumulatingFiredPanes()
        .triggering(Repeatedly
                .forever(AfterProcessingTime
                        .pastFirstElementInPane())));
```

Stateful Processing - Beam SQL

Problems with retaining state as a result of simple aggregate function

- The event order is not kept
  - Difficult to evaluate order-dependent conditions

```
# Compare the amount purchased immediately before
PayAmount[0] > PayAmount[1]

# Compare the total amount of the last three purchases
sum(PayAmount[0:3]) > 3000
```

  - Batch processing does not produce the same results as streaming
    - The use of past data to confirm campaign target audiences
    - Switching the trigger settings will produce the same final result. However, the evaluation process will become unknown.

Stateful Processing - DoFn State API

DoFn **State API** enables DoFn to perform stateful processing

- **OrderedListState**
  - State enables simple and efficient retrieval of events based on eventtime
- **RequiresTimeSortedInput**
  - Annotation for ensuring event time order, can be used
  - Ensure that batch processing is equivalent to streaming processing

※Be aware of performance,
- Paralleling is as difficult as CombineFn.
- Performance may decrease when using the State API with runner v2.

Stateful Processing - DoFn State API

```java
@ProcessElement  no usages
@RequiresTimeSortedInput // Need for Batch
public void processElement(
        final ProcessContext c,
        final @StateId(STATE_ID_BUFFER) OrderedListState<MElement> bufferState,
        final @StateId(STATE_ID_MAX_COUNT_TIME) ValueState<Instant> maxCountState) {

    final Instant eventTime = c.timestamp();
    // read state
    final Instant maxMinTimestamp = calcMinTimestamp(maxCountState, eventTime);
    final List<TimestampedValue<MElement>> buffer = Lists
            .newArrayList(bufferState.readRange(maxMinTimestamp, eventTime));

    // stateful processing
    final Map<String, Object> output = select.select(c.element().getValue(), buffer, eventTime);

    // update state
    if(buffer.size() >= maxRange.maxCount) {
        maxCountState.write(buffer.get(buffer.size() - maxRange.maxCount).getTimestamp());
    }
    bufferState.clearRange(Instant.ofEpochMilli( i: 0L), maxMinTimestamp);
    bufferState.add(TimestampedValue.of(MElement.of(output, eventTime), eventTime));

    c.output(MElement.of(output, eventTime));
}
```

Process in order of event time for each key using **RequiresTimeSorted** Input annotation.

Keep the oldest event time as the state.
(Supports both count and time based types)

Retrieve data between the current event time and the oldest event time from **OrderedListState**.

Execute stateful processing according to the config definition.

Delete old states that are no longer referenced.
Add new input with event time to the OrderedListState

Stateful Processing - DoFn State API

```
select:
  - name: output field name
    func: function name
    range: scope of processed data
    additional_parameters…:
      parameters for each func
```

```
select:
  - name: sum_30_longFields
    func: sum
    range:
      count: 30
    expression: "longField1 * longField2"
```

```
select:
  - name: lag_expression
    func: lag
    expression: "(longField[2] - longField[0])/(1 + longField[0])"
```

Refer to past data values with the current data index as 0.

Easily define and use stateful processing

- Single row processing
  - Exp: expression, cast, replace
- Window function
  - Exp: avg, sum, max, min, lag
  - Aggregate within specified range
  - Navigation func(lag) is also supported

Stateful Processing - DoFn State API example

```
transforms:
 - name: stateful_processing
   module: select
   inputs:
     - pubsub_source
   parameters:
     groupFields:
       - user_id
     select:
       - name: array_agg
         func: array_agg
         field: value
         range:
           count: 64
 - name: onnx_prediction
   module: onnx
   inputs:
     - stateful_processing
   parameters:
     model:
       path: gs://xxx/chronos-bolt-mini.onnx
     inferences:
       - input: stateful_processing
         mappings:
           inputs:
             context: array_agg
           outputs:
             forecasts: predictions
         select:
           - name: user_id
           - name: forecasts_4_1
             func: reshape
             field: predictions
             shape: [64,9]
             indices: [0,4]
```

Holds the values of the most recent 64 value fields as an array for each user.

Definition of mapping between input/output field names in the Chronos-bolt model and input data field names.

In this Chronos-bolt model, predictions are made for 64 steps, and nine candidates are output as fluctuations, so post-processing is performed to extract the central prediction for the next step.

By working with the onnx module, stateful features generated in real-time can be easily used with onnx prediction models.

- Example of using time-series model Chronos-Bolt

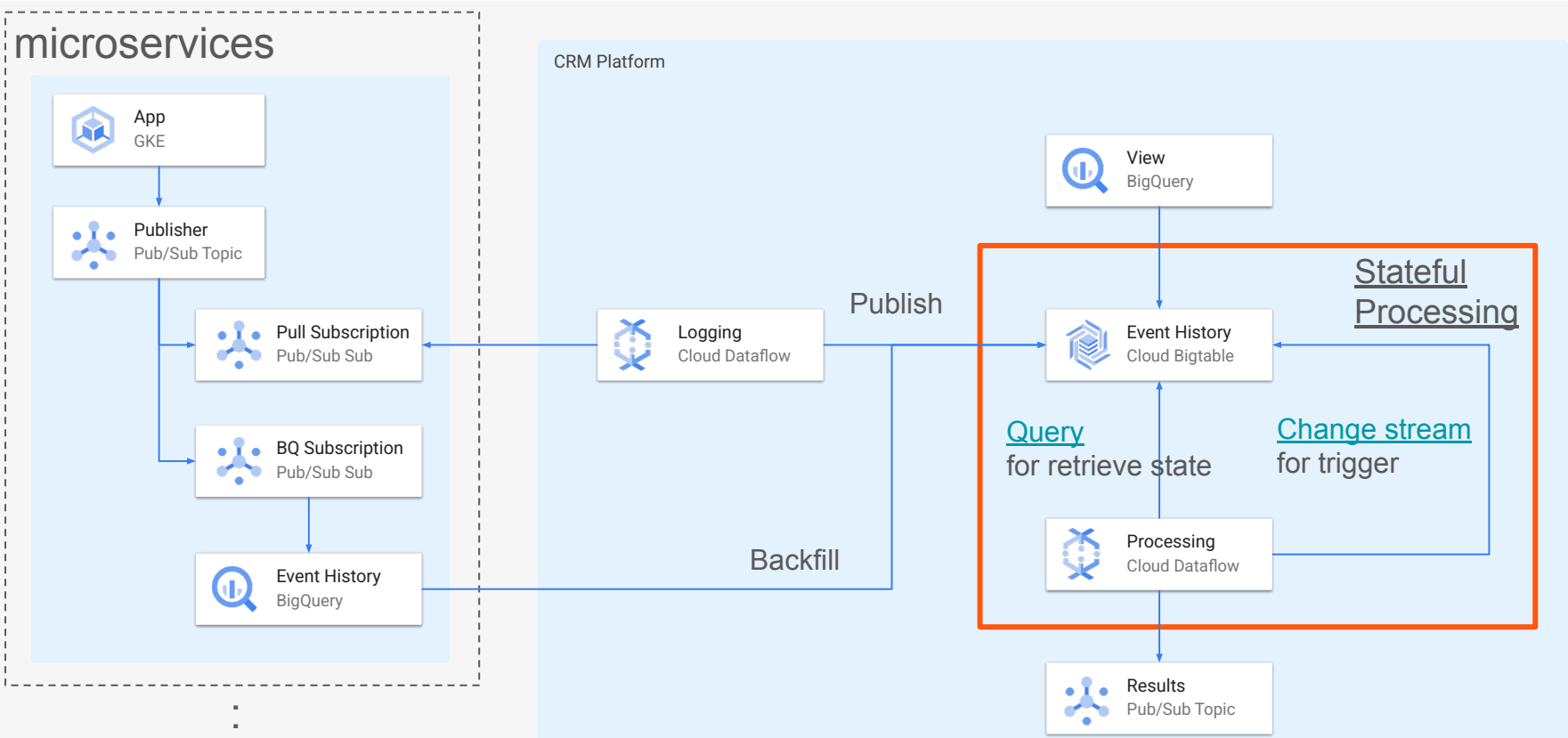
Stateful Processing - External Data sources

Long-term state retention is problematic

- Some requirements need to retain state for several months
  - Some campaigns last for few months.
- Need to restore state if streaming job fails
  - Costly to keep state periodically per key
- All data for the period required to construct the state must be fed in
  - Exp: Adding new fields to change conditions during the campaign period

Instead of keeping the state in the pipeline worker process memory, keep it in an external data store and retrieve it as needed.

Stateful Processing - External Data sources



Stateful Processing - External Data sources

① Insert events into data store

row_key	buy	sell
1234567#1753430662014	id=xxx,amount=30	-
1234567#1753430768338	-	id=yyy,amount=150

② Capture Change Data Record

```
SELECT
  buy, sell”
FROM
  event_history
WHERE
  __key BETWEEN
  ”${user_id}#${prev_timestamp}”
  AND ”${user_id}#${timestamp}”
```

③ Generate Query to retrieve event history from Change Data

④ Retrieve event history

※Need to use cache mechanism in-memory

⑤ Stateful Processing

Stateful Processor

Stateful Processing - External Data sources

Instead of applying processing to the input, build a query from the input data and applying stateful (or single-row) processing to the retrieved data.

```
sources:
  - name: bigtable_cdc
    module: bigtable
    mode: changeDataCapture
    parameters:
      projectId: xxx
      instanceId: xxx
      tableId: event_history
      changeStream:
        changeStreamName: event_history_cdc
transforms:
  - name: stateful_processing
    module: bigtable
    inputs:
      - bigtable_cdc
    parameters:
      projectId: xxx
      instanceId: xxx
      query: |
        SELECT
          ...
        FROM
          event_history
        WHERE
          _key BETWEEN  '${user_id}#${prev_timestamp}'
          AND '${user_id}#${timestamp}'
      select:
        - name: avg_32_field_A
          func: avg
          field: field_A
          range:
```

Generate query to retrieve targets from fields contained in CDC data.

Define post-processing for query results Stateful processing can also be used

※ Not yet published in the OSS ver

Summary

- Mercari Pipeline
  - What is Mercari Pipeline ?
  - Configuration management
- Stateful Processing
  - Aggregation - Beam SQL
  - DoFn State API
- Stateful Processing with External Data sources
  - Processing with Cloud Bigtable

Yoichi Nagai

QUESTIONS?

orfeon@ Twitter Linkedin Github

BEAM SUMMIT NYC 2025