OCADO

[**Proposed Solution: 2**](#_i3rekpddht9)

[Loading Data in BigQuery Table using BigQuery Subscription 3](#_xqgx5lr5ljt)

[Aggregation using BigQuery Stored Procedure: 3](#_7sh0aax9y7)

[BigQuery Table function to build Hourly Partition 3](#_220o0mgwmvia)

[BigQuery Stored Procedure to build hourly aggregation: 5](#_qz196tme8lw6)

[Handling High WaterMark: 5](#_n7nhjokn3agk)

[Calculate Start time for processing Window 6](#_dddqfh2h3r50)

[Calculate End Time for processing Window: 6](#_ojyo7y6ksbk1)

[Building Temporary/Stage Table: Machines and Time Intervals: 7](#_ut8jp03i7w5m)

[Build aggregation for the data in temp/stage table: 9](#_cwdrxfblo07j)

[Merge data from stage/Temp table to BigQuery table: hourly\_sorting\_aggregate 12](#_jli8qp1481bt)

[Scheduling job : Bigquery Transfer service 14](#_yp876o3kgi0d)

[**Pro’s & Con’s Approach 14**](#_fdyn4z9f2vyv)

[Pro’s: 14](#_8qjbhdtciu40)

[**Con’s 15**](#_klv1jlqj0284)

[**Appendix 16**](#_91vk3orutwjn)

[1) Complete code for stored procedure building aggregate table. 16](#_h52slmfibpfv)

[2) Code for table Function calculating time interval 22](#_epbgidi675mz)

[**Questions Asked 23**](#_cw6unomr6zde)

# 

# Proposed Solution:

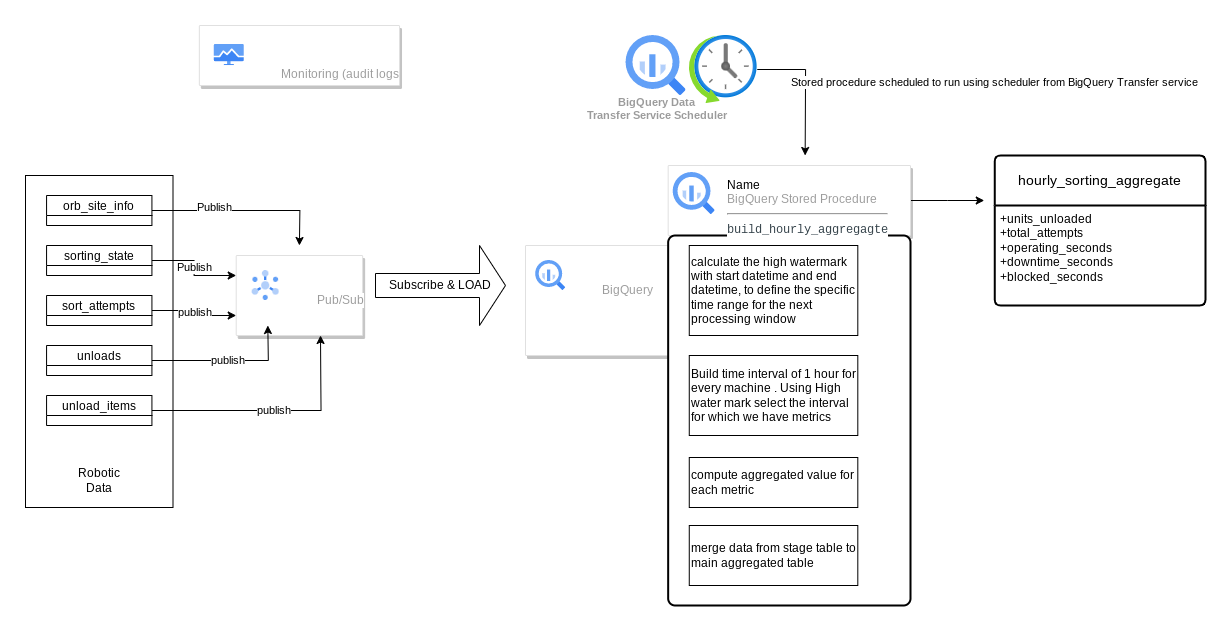
I recommend using the GCP (Google Cloud Platform) for hosting and processing Robotic Arm data. GCP offers excellent tools such as Pub/Sub, BigQuery, and Cloud Dataflow, which are well-suited for implementing solutions for test scenarios.

Implementation Steps:

1. Loading Data: Utilize BigQuery Subscriptions in Pub/Sub to write messages to a BigQuery table.
2. Define a BigQuery Stored Procedure for aggregating the data.
3. Schedule the BigQuery Stored Procedure using the Scheduler feature provided by the BigQuery Transfer Service.Alternatively scheduling could be done using event processing like Cloud Function Gen2 or additional pub/sub messaging.

By following these steps, we can efficiently load data from Pub/Sub into BigQu

ery and perform the necessary aggregations using a scheduled procedure. This approach leverages the capabilities of GCP to handle and process Robotic Arm data effectively.



## 

## Loading Data in BigQuery Table using [BigQuery Subscription](https://cloud.google.com/pubsub/docs/bigquery)

As an alternative to using simple data ingestion pipelines that often utilize Dataflow to write data to BigQuery, we can leverage the BigQuery subscription feature for data loading.

In this approach, the IoT devices publish data to a GCP Pub/Sub topic schema. By using the BigQuery subscription, Pub/Sub can automatically utilize the schema of the Pub/Sub topic to which the subscription is attached. Furthermore, Pub/Sub writes the fields in the messages directly to the corresponding columns in the BigQuery table.

By adopting this method, we can streamline the data loading process by leveraging the automatic schema inference and direct writing capabilities provided by the BigQuery subscription. This simplifies the data ingestion pipeline and ensures accurate mapping of fields to columns in the BigQuery table.

## Aggregation using BigQuery Stored Procedure:

The hourly aggregation process involves the following steps:

1. Building the hourly intervals for the aggregated table: A procedure is implemented to generate the hourly intervals based on the desired time range.
2. Obtaining the high watermark for the next processing window: The high watermark is calculated by determining the maximum timestamp of the data received for each table, which serves as the starting point for the next processing window.
3. Building the temporary/staging table: Using machine information and the hourly interval table, a temporary or staging table is created. The data in this table is restricted to the high watermark, ensuring that only relevant data is processed.
4. Calculating the hourly aggregation metrics: The received feed is processed, and the required hourly aggregation metrics are computed based on the data within the defined time intervals.
5. Merging the aggregated data: The aggregated data from the staging or temporary table is merged into the target BigQuery table. This ensures that the aggregated metrics are stored and available for further analysis.

### BigQuery Table function to build Hourly Partition

| CREATE OR REPLACE TABLE FUNCTION `aqueous-walker-379718.ocado\_sol.date\_interval`(date\_param TIMESTAMP) AS ( WITH intervals AS (  SELECT  GENERATE\_TIMESTAMP\_ARRAY(  TIMESTAMP\_TRUNC(TIMESTAMP(date\_param), HOUR),  TIMESTAMP\_ADD(TIMESTAMP\_TRUNC(TIMESTAMP(date\_param), HOUR), INTERVAL 23 HOUR),  INTERVAL 1 HOUR  ) AS ts\_array ), extended\_intervals AS (  SELECT  ARRAY\_CONCAT(ts\_array, [TIMESTAMP\_ADD(TIMESTAMP\_TRUNC(TIMESTAMP(date\_param), HOUR), INTERVAL 24 HOUR)]) AS extended\_ts\_array  FROM intervals ) SELECT  CAST(FORMAT\_TIMESTAMP('%F %T', start\_ts) AS TIMESTAMP) AS start\_time,  CAST(FORMAT\_TIMESTAMP('%F %T', end\_ts) AS TIMESTAMP) AS end\_time FROM extended\_intervals CROSS JOIN UNNEST(extended\_ts\_array) AS start\_ts JOIN UNNEST(extended\_ts\_array) AS end\_ts ON start\_ts = TIMESTAMP\_SUB(end\_ts, INTERVAL 1 HOUR) ORDER BY start\_time ); |
| --- |

To build the date interval for a given input date, we utilize the functionality of BigQuery's Table function. This feature allows us to define a reusable query that generates a table of results.

Our table function accepts a date parameter as input. Through a series of calculations, we generate a table of intervals. Initially, we create an array of timestamps using the GENERATE\_TIMESTAMP\_ARRAY function. This array consists of timestamps truncated to the hour for the provided date parameter. The duration of the array spans 24 hours, with intervals of 1 hour.

Expanding on this, we extend the array of timestamps by adding one additional timestamp at the end. This extra timestamp is precisely 24 hours after the initial timestamps. We achieve this using the array\_concat function.

Next, we select the start and end timestamps from the extended array of intervals. To accomplish this, we employ the unnest function, which expands the array into individual rows. We then join the start and end timestamps based on specific conditions. In this case, the start timestamp is equal to the end timestamp minus 1 hour.

The resulting table provides the start time and end time for each interval, with timestamps formatted accordingly. To ensure clarity and organization, the intervals are ordered by the start time.

**Query to Execute Table function for date Interval**

| SELECT \* FROM `aqueous-walker-379718.ocado\_sol.date\_interval`('2021-05-21 00:00:00') |
| --- |

| start\_time | end\_time |
| --- | --- |
| 2021-05-21 00:00:00.000000 UTC | 2021-05-21 01:00:00.000000 UTC |
| 2021-05-21 01:00:00.000000 UTC | 2021-05-21 02:00:00.000000 UTC |
| 2021-05-21 02:00:00.000000 UTC | 2021-05-21 03:00:00.000000 UTC |
| 2021-05-21 03:00:00.000000 UTC | 2021-05-21 04:00:00.000000 UTC |
| 2021-05-21 04:00:00.000000 UTC | 2021-05-21 05:00:00.000000 UTC |
| 2021-05-21 05:00:00.000000 UTC | 2021-05-21 06:00:00.000000 UTC |
| 2021-05-21 06:00:00.000000 UTC | 2021-05-21 07:00:00.000000 UTC |
| 2021-05-21 07:00:00.000000 UTC | 2021-05-21 08:00:00.000000 UTC |
| 2021-05-21 08:00:00.000000 UTC | 2021-05-21 09:00:00.000000 UTC |
| 2021-05-21 09:00:00.000000 UTC | 2021-05-21 10:00:00.000000 UTC |
| 2021-05-21 10:00:00.000000 UTC | 2021-05-21 11:00:00.000000 UTC |
| 2021-05-21 11:00:00.000000 UTC | 2021-05-21 12:00:00.000000 UTC |
| 2021-05-21 12:00:00.000000 UTC | 2021-05-21 13:00:00.000000 UTC |
| 2021-05-21 13:00:00.000000 UTC | 2021-05-21 14:00:00.000000 UTC |
| 2021-05-21 14:00:00.000000 UTC | 2021-05-21 15:00:00.000000 UTC |
| 2021-05-21 15:00:00.000000 UTC | 2021-05-21 16:00:00.000000 UTC |
| 2021-05-21 16:00:00.000000 UTC | 2021-05-21 17:00:00.000000 UTC |
| 2021-05-21 17:00:00.000000 UTC | 2021-05-21 18:00:00.000000 UTC |
| 2021-05-21 18:00:00.000000 UTC | 2021-05-21 19:00:00.000000 UTC |
| 2021-05-21 19:00:00.000000 UTC | 2021-05-21 20:00:00.000000 UTC |
| 2021-05-21 20:00:00.000000 UTC | 2021-05-21 21:00:00.000000 UTC |
| 2021-05-21 21:00:00.000000 UTC | 2021-05-21 22:00:00.000000 UTC |
| 2021-05-21 22:00:00.000000 UTC | 2021-05-21 23:00:00.000000 UTC |
| 2021-05-21 23:00:00.000000 UTC | 2021-05-22 00:00:00.000000 UTC |

### BigQuery Stored Procedure to build hourly aggregation:

The code for this stored procedure can be discussed under following top headings:

1. Handling of high watermarks
2. Buidling stage table with time interval
3. Build Aggregation
4. Merge aggregation to main table.

#### Handling High WaterMark:

The code provided is designed as a procedure that can be invoked by passing a start datetime and an end datetime as input parameters. If both input parameters are provided, no calculation for the high watermark is performed. Instead, the provided input values are directly used as the processing window. This option allows the team to rebuild historical time intervals that require manual intervention or additional processing.

The purpose of the code is to calculate the start\_datetime and end\_datetime based on the data available in the system. The code considers the existing data to determine the appropriate values for the processing window.

##### 

##### **Calculate Start time for processing Window**

| -- Check if the input parameter is null  IF input\_start\_process\_datetime IS NULL  AND table\_count > 0   THEN  -- Get start processing time from hourly\_sorting\_aggregate  SET start\_process\_datetime = (  SELECT TIMESTAMP\_SUB(MAX(start\_time), INTERVAL input\_slide\_window HOUR)  FROM `ocado\_sol.hourly\_sorting\_aggregate` hsa  );  ELSE  SET start\_process\_datetime = input\_start\_process\_datetime;  END IF; |
| --- |

In simple terms, the code finds the maximum value of the start\_time column in the ocado\_sol.hourly\_sorting\_aggregate table. This start\_time represents the start of a process.

Then, it subtracts a specific duration, referred to as input\_slide\_window, from the maximum start\_time. The input\_slide\_window is given in hours and determines how far back in time we want to go.

Once the calculation is done, the resulting timestamp is assigned to the variable start\_process\_datetime, which can be used further in the code for other purposes.

**For the given dataset the query returns**

**Input\_slide\_interval = 0**

| **2021-05-24 03:00:00 UTC** |
| --- |

**Input\_slide\_interval = 4**

| **2021-05-23 23:00:00 UTC** |
| --- |

To ensure that data for the previous 4 hours is reprocessed in the next run, a sliding window of 4 hours is used.

##### **Calculate End Time for processing Window:**

| **-- Identify end date for processing window using the max\_processed\_datetime  IF input\_end\_process\_datetime IS NULL THEN  SET last\_received\_datetime = (  SELECT MIN(meta\_received\_time) meta\_received\_time  FROM (  SELECT MAX(soat.meta\_received\_time) meta\_received\_time FROM `ocado\_sol.sort\_attempts` soat  UNION ALL  SELECT MAX(sost.meta\_received\_time) meta\_received\_time FROM `ocado\_sol.sorting\_states` sost  UNION ALL  SELECT MAX(unit.meta\_received\_time) meta\_received\_time FROM `ocado\_sol.unload\_items` unit  UNION ALL  SELECT MAX(unlo.meta\_received\_time) meta\_received\_time FROM `ocado\_sol.unloads` unlo  )  );  ELSE  SET last\_received\_datetime = input\_end\_process\_datetime;  END IF;** |
| --- |

The code identifies the end date for the processing window. It consists of an if-else condition that checks the value of input parameter input\_end\_process\_datetime.

If the input parameter input\_end\_process\_datetime is null then the code proceeds to find the last received datetime. It does this be selecting the minimum value of meta\_received\_time column from multiple tables viz (ocado\_sol.sort\_attempts,, ocado\_sol.sorting\_states,, ocado\_sol.unload\_items,, ocado\_sol.unloads). It uses subqueries to retrieve the maximum meta\_received\_time from each table and combines them using the UNION ALL clause. The overall minimum value of meta\_received\_time is assigned to the variable last\_received\_datetime.

On the other hand, if the input\_end\_process\_datetime is not NULL, which means a specific end date is provided, the code simply assigns the value of input\_end\_process\_datetime to the variable last\_received\_datetime.

In summary, the code is responsible for determining the end date of the processing window. If no specific end date is provided, it retrieves the minimum meta\_received\_time from various tables.

For the given dataset the query returns with null as input enddate

| 2021-05-24 03:31:07.948087 UTC |
| --- |

#### Building Temporary/Stage Table: Machines and Time Intervals:

| -- Build Stage Table  CREATE OR REPLACE TEMP TABLE `stg\_machine\_hourly\_aggregate` AS (  WITH  date\_to\_process AS (  SELECT \*  FROM ocado\_sol.date\_interval(start\_process\_datetime)  ),  machine\_list AS (  SELECT DISTINCT  sos.meta\_orb\_site,  osi.timezone,  sos.meta\_machine\_name,  sos.meta\_deploy\_mode  FROM  `ocado\_sol.sorting\_states` sos  JOIN  `ocado\_sol.orb\_site\_info` osi ON osi.orb\_site = sos.meta\_orb\_site  )  SELECT  mcl.meta\_orb\_site,  mcl.meta\_machine\_name,  mcl.meta\_deploy\_mode,  mcl.timezone,  dtp.start\_time,  dtp.end\_time,  DATETIME(dtp.start\_time, mcl.timezone) local\_start\_time,  DATETIME(dtp.end\_time, mcl.timezone) local\_end\_time  FROM  machine\_list mcl  CROSS JOIN  date\_to\_process dtp  WHERE  dtp.start\_time BETWEEN start\_process\_datetime AND last\_received\_datetime  ORDER BY  start\_time  ); |
| --- |

The code builds a temporary table named stg\_machine\_hourly\_aggregate. It uses CTEs to select the time intervals and gather information about distinct machines.

The table creation involves two CTEs. The first CTE date\_to\_process, select start and end time for processing interval using the BigQuery Table function date\_interval. The Second CTE ‘machine\_list’ gather sinformation about distinct machines along with orb\_site and timezone , machine\_name and deploy\_mode columns

The query performs a cross join between the machine\_list and date\_to\_process CTEs to combine the machines with the corresponding time intervals. *It applies a WHERE clause to filter the rows where the start\_time falls between the start\_process\_datetime and last\_received\_datetime.*

In Summary, the code builds a temporary table name stg\_machine\_hourly\_aggreggate. It uses CTEs to select time intervals and gather information about distinct machines. The final result is a table containing machine data along with their respective time intervals, which will be further utilised to build aggregate.

The output for this code for the data provided:

| meta\_orb\_site | meta\_machine\_name | meta\_deploy\_mode | timezone | start\_time | end\_time | local\_start\_time | local\_end\_time |
| --- | --- | --- | --- | --- | --- | --- | --- |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | America/New\_York | 2021-05-23 23:00:00.000000 UTC | 2021-05-24 00:00:00.000000 UTC | 2021-05-23 19:00:00 | 2021-05-23 20:00:00 |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | America/New\_York | 2021-05-24 00:00:00.000000 UTC | 2021-05-24 01:00:00.000000 UTC | 2021-05-23 20:00:00 | 2021-05-23 21:00:00 |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | America/New\_York | 2021-05-24 01:00:00.000000 UTC | 2021-05-24 02:00:00.000000 UTC | 2021-05-23 21:00:00 | 2021-05-23 22:00:00 |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | America/New\_York | 2021-05-24 02:00:00.000000 UTC | 2021-05-24 03:00:00.000000 UTC | 2021-05-23 22:00:00 | 2021-05-23 23:00:00 |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | America/New\_York | 2021-05-24 03:00:00.000000 UTC | 2021-05-24 04:00:00.000000 UTC | 2021-05-23 23:00:00 | 2021-05-24 00:00:00 |

By utilizing the sliding window approach, the code allows for the reprocessing of data from the previous date, specifically at the time 2021-05-23 23:00:00.00. This occurrence takes place because the sliding window includes this particular timestamp within its range. As a result, the data for that specific timestamp will be selected and processed again during subsequent runs

#### Build aggregation for the data in temp/stage table:

-- Build aggregate

| CREATE OR REPLACE TEMP TABLE stg\_hourly\_sorting\_aggregate AS (  WITH  sort\_attempt\_aggregate AS (  SELECT  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time,  COUNT(\*) total\_attempts,  SUM(CASE WHEN outcome = 'complete' THEN 1 ELSE 0 END) successful\_attempt,  SUM(CASE WHEN outcome = 'partial' THEN 1 ELSE 0 END) partial\_attempt  FROM  `stg\_machine\_hourly\_aggregate` sha  LEFT OUTER JOIN  `ocado\_sol.sort\_attempts` sat ON (  sat.meta\_orb\_site = sha.meta\_orb\_site  AND sat.meta\_machine\_name = sha.meta\_machine\_name  AND sat.meta\_deploy\_mode = sha.meta\_deploy\_mode  AND sat.meta\_event\_time BETWEEN sha.start\_time AND sha.end\_time  AND sat.meta\_received\_time BETWEEN start\_process\_datetime AND last\_received\_datetime  )  GROUP BY  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time  ),  sort\_state\_aggregate AS (  SELECT  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time,  SUM(CASE WHEN sos.state = 'sorting' THEN TIMESTAMP\_DIFF(sos.end\_time, sos.start\_time, SECOND) ELSE 0 END) sorting\_seconds,  SUM(CASE WHEN sos.state = 'blocked' THEN TIMESTAMP\_DIFF(sos.end\_time, sos.start\_time, SECOND) ELSE 0 END) locked\_seconds,  SUM(CASE WHEN sos.state = 'scheduled\_downtime' THEN TIMESTAMP\_DIFF(sos.end\_time, sos.start\_time, SECOND) ELSE 0 END) scheduled\_downtime\_seconds,  SUM(CASE WHEN sos.state = 'unscheduled\_downtime' THEN TIMESTAMP\_DIFF(sos.end\_time, sos.start\_time, SECOND) ELSE 0 END) unscheduled\_downtime\_seconds  FROM  `stg\_machine\_hourly\_aggregate` sha  LEFT OUTER JOIN  `ocado\_sol.sorting\_states` sos ON (  sha.meta\_orb\_site = sos.meta\_orb\_site  AND sha.meta\_machine\_name = sos.meta\_machine\_name  AND sha.meta\_deploy\_mode = sos.meta\_deploy\_mode  AND sos.meta\_event\_time BETWEEN sha.start\_time AND sha.end\_time  AND sos.meta\_received\_time BETWEEN start\_process\_datetime AND last\_received\_datetime  )  GROUP BY  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time  ),  units\_aggregate AS (  SELECT  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time,  COUNT(\*) number\_of\_units  FROM  `stg\_machine\_hourly\_aggregate` sha  LEFT OUTER JOIN  `ocado\_sol.unload\_items` uni ON (  uni.meta\_orb\_site = sha.meta\_orb\_site  AND uni.meta\_machine\_name = sha.meta\_machine\_name  AND uni.meta\_deploy\_mode = sha.meta\_deploy\_mode  AND uni.meta\_event\_time BETWEEN sha.start\_time AND sha.end\_time  AND uni.meta\_received\_time BETWEEN start\_process\_datetime AND last\_received\_datetime  )  GROUP BY  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time  )  SELECT  soat.meta\_orb\_site,  soat.meta\_machine\_name,  soat.meta\_deploy\_mode,  soat.start\_time,  soat.end\_time,  soat.total\_attempts,  soat.successful\_attempt,  soat.partial\_attempt,  sost.sorting\_seconds,  sost.locked\_seconds,  sost.scheduled\_downtime\_seconds,  sost.unscheduled\_downtime\_seconds,  unag.number\_of\_units  FROM  sort\_attempt\_aggregate soat  JOIN  sort\_state\_aggregate sost ON (  soat.meta\_orb\_site = sost.meta\_orb\_site  AND soat.meta\_machine\_name = sost.meta\_machine\_name  AND soat.meta\_deploy\_mode = sost.meta\_deploy\_mode  AND soat.start\_time = sost.start\_time  AND soat.end\_time = sost.end\_time  )  JOIN  units\_aggregate unag ON (  soat.meta\_orb\_site = unag.meta\_orb\_site  AND soat.meta\_machine\_name = unag.meta\_machine\_name  AND soat.meta\_deploy\_mode = unag.meta\_deploy\_mode  AND soat.start\_time = unag.start\_time  AND soat.end\_time = unag.end\_time  )  ); |
| --- |

The provided query is used to create a temporary table called stg\_hourly\_sorting\_aggregate. This table is constructed by aggregating data from different source tables using Common Table Expressions (CTEs). Here is a breakdown of the code:

1. The sort\_attempt\_aggregate CTE aggregates data related to sorting attempts. It calculates the total number of attempts, successful attempts, and partial attempts for each combination of site, machine, deploy mode, start time, and end time. It joins the stg\_machine\_hourly\_aggregate table with the sort\_attempts table based on specific join and filter conditions.
2. The sort\_state\_aggregate CTE aggregates data related to sorting states. It calculates the total time spent in different states (sorting, blocked, scheduled downtime, unscheduled downtime) for each combination of site, machine, deploy mode, start time, and end time. It joins the stg\_machine\_hourly\_aggregate table with the sorting\_states table based on specific join and filter conditions.
3. The units\_aggregate CTE aggregates data related to units. It calculates the total number of units for each combination of site, machine, deploy mode, start time, and end time. It joins the stg\_machine\_hourly\_aggregate table with the unload\_items table based on specific join and filter conditions.
4. The final SELECT statement combines the results from the three CTEs. It selects the relevant columns from each CTE, including the aggregated values, and joins the CTEs based on the site, machine, deploy mode, start time, and end time.

The resulting stg\_hourly\_sorting\_aggregate table contains the aggregated information about sorting attempts, sorting states, and units for each hourly interval within the specified processing window.

#### Merge data from stage/Temp table to BigQuery table: hourly\_sorting\_aggregate

| IF table\_count > 0 THEN  MERGE INTO `ocado\_sol.hourly\_sorting\_aggregate` hsa  USING (  SELECT \*  FROM stg\_hourly\_sorting\_aggregate  ) shsa ON (  hsa.meta\_orb\_site = shsa.meta\_orb\_site  AND hsa.meta\_deploy\_mode = shsa.meta\_deploy\_mode  AND hsa.meta\_machine\_name = shsa.meta\_machine\_name  AND hsa.start\_time = shsa.start\_time  AND hsa.end\_time = shsa.end\_time  )  WHEN MATCHED THEN  UPDATE SET  hsa.total\_attempts = shsa.total\_attempts,  -- hsa.successful\_attempt = shsa.successful\_attempt,  hsa.partial\_attempt = shsa.partial\_attempt,  hsa.sorting\_seconds = shsa.sorting\_seconds,  hsa.scheduled\_downtime\_seconds = shsa.scheduled\_downtime\_seconds,  hsa.unscheduled\_downtime\_seconds = shsa.unscheduled\_downtime\_seconds,  hsa.number\_of\_units = shsa.number\_of\_units  WHEN NOT MATCHED THEN  INSERT (  meta\_orb\_site,  meta\_machine\_name,  meta\_deploy\_mode,  start\_time,  end\_time,  total\_attempts,  sorting\_seconds,  scheduled\_downtime\_seconds,  number\_of\_units  )  VALUES (  meta\_orb\_site,  meta\_machine\_name,  meta\_deploy\_mode,  start\_time,  end\_time,  total\_attempts,  sorting\_seconds,  scheduled\_downtime\_seconds,  number\_of\_units  );  ELSE  CREATE OR REPLACE TABLE ocado\_sol.hourly\_sorting\_aggregate AS (  SELECT \*  FROM stg\_hourly\_sorting\_aggregate  );  END IF; |
| --- |

The code performs an upsert operation (merge) on the table ocado\_sol.hourly\_sorting\_aggregate using data from the stg\_hourly\_sorting\_aggregate temporary table.

The condition IF table\_count > 0 checks if the table ocado\_sol.hourly\_sorting\_aggregate already exists. If the table exists, the code executes the merge statement (MERGE INTO). If the table does not exist, it creates a new table using the CREATE OR REPLACE TABLE statement.

The merge statement compares the data between the target table and the source table using specific join conditions. When a match is found, the update operation is performed to modify certain columns in the target table with corresponding values from the source table. If no match is found, indicating a new row, the insert operation is performed to add a new row to the target table using the values from the source table.

In summary, the code checks if the target table exists. If it does, an upsert operation (update/insert) is performed on the target table based on the data from the source table. If the target table does not exist, a new table is created.

The final table has following output:

| meta\_orb\_site | meta\_machine\_name | meta\_deploy\_mode | start\_time | end\_time | total\_attempts | successful\_attempt | partial\_attempt | sorting\_seconds | locked\_seconds | scheduled\_downtime\_seconds | unscheduled\_downtime\_seconds | number\_of\_units |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | 2021-05-23 23:00:00.000000 UTC | 2021-05-24 00:00:00.000000 UTC | 1 |  |  | 0 |  | 0 |  | 1 |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | 2021-05-24 03:00:00.000000 UTC | 2021-05-24 04:00:00.000000 UTC | 1 | 0 | 0 | 0 | 943 | 0 | 0 | 8 |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | 2021-05-24 02:00:00.000000 UTC | 2021-05-24 03:00:00.000000 UTC | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 12 |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | 2021-05-24 01:00:00.000000 UTC | 2021-05-24 02:00:00.000000 UTC | 2 | 2 | 0 | 0 | 1232 | 0 | 0 | 12 |
| aehzt | [aehzt-orb129.ext.kindredai.net](http://aehzt-orb129.ext.kindredai.net) | production | 2021-05-24 00:00:00.000000 UTC | 2021-05-24 01:00:00.000000 UTC | 5 | 5 | 0 | 0 | 1729 | 0 | 0 | 13 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

#### Scheduling job : Bigquery Transfer service

Following code is to be scheduled:

| DECLARE input\_start\_process\_datetime TIMESTAMP DEFAULT NULL; DECLARE input\_end\_process\_datetime TIMESTAMP DEFAULT NULL; DECLARE input\_slide\_window INT64 DEFAULT NULL 4; CALL `aqueous-walker-379718.ocado\_sol.build\_hourly\_aggregagte`(input\_start\_process\_datetime, input\_end\_process\_datetime, input\_slide\_window); |
| --- |

# Pro’s & Con’s Approach

## Pro’s:

* Data loaded using BigQuery Subscription (Pub/Sub): By leveraging BigQuery Subscription and Pub/Sub, the approach eliminates the need for additional tools like DataFlow (Apache Beam) for data loading from IoT/Robotic sources to BigQuery tables. This simplifies the data loading process and reduces the dependency on external tools, resulting in a streamlined data pipeline.
* No additional cost of using DataFlow: Since the approach utilizes BigQuery Subscription and Pub/Sub for data loading, there is no need to incur the additional cost associated with using DataFlow templates or the infrastructure required for running DataFlow pipelines. This can lead to cost savings in the data ingestion process.
* BigQuery stored procedure for building aggregate: The use of BigQuery stored procedure allows for aggregating the data within the BigQuery processing engine itself. This eliminates the need to move the data out of BigQuery for processing using external tools like Python, Spark, or DataFlow or Dataproc. By leveraging the built-in processing capabilities of BigQuery, there is no additional I/O and no extra cost incurred for using third-party processing tools.
* In this scenario, where the transformations and aggregation can be effectively expressed as SQL-based operations, leveraging distributed processing with BigQuery stored procedures can offer advantages over cluster-based processing tools like Hadoop or in-memory processing tools like Spark. By utilizing the built-in distributed computing capabilities of BigQuery, the data processing tasks can be efficiently executed at scale without the need for managing a separate cluster infrastructure. This eliminates the overhead of setting up and maintaining a distributed computing environment, resulting in simplified operations and potentially better performance for the given SQL-based operations
* Scheduled using Scheduler from BigQuery Transfer: The code is scheduled using the Scheduler feature of BigQuery Transfer. This allows for automated and scheduled execution of the code, ensuring timely and efficient data processing. By leveraging the native scheduling capabilities of BigQuery, there is no need for external scheduling tools, resulting in simplified management of the data processing workflow.
* Dataform can be configured with git repo can be used for version control of the code.

Overall, this approach offers cost savings by eliminating the need for additional tools, utilizing the built-in capabilities of BigQuery for data processing, and leveraging the scheduling feature of BigQuery Transfer for automated execution. It streamlines the data pipeline and reduces the complexity and associated costs of using external processing tools and infrastructure.

## Con’s

* Unable to use the benefit of unified programming for batch and streaming using Apache beam in cloud dataflow.

# Appendix

#### Complete code for stored procedure building aggregate table.

| CREATE OR REPLACE PROCEDURE `aqueous-walker-379718.ocado\_sol.build\_hourly\_aggregagte`(input\_start\_process\_datetime TIMESTAMP, input\_end\_process\_datetime TIMESTAMP, input\_slide\_window INT64) BEGIN   DECLARE last\_received\_datetime TIMESTAMP;  DECLARE start\_process\_datetime TIMESTAMP;  DECLARE table\_count INT64;  -- Check Hourly aggregate table exists  SET table\_count = (  SELECT COUNT(1) AS cnt  FROM `ocado\_sol.\_\_TABLES\_SUMMARY\_\_`  WHERE table\_id = 'hourly\_sorting\_aggregate'  );  -- Check if the input parameter is null  IF input\_start\_process\_datetime IS NULL  AND table\_count > 0   THEN  -- Get start processing time from hourly\_sorting\_aggregate  SET start\_process\_datetime = (  SELECT TIMESTAMP\_SUB(MAX(start\_time), INTERVAL input\_slide\_window HOUR)  FROM `ocado\_sol.hourly\_sorting\_aggregate` hsa  );  ELSE  SET start\_process\_datetime = input\_start\_process\_datetime;  END IF;   -- Identify end date for processing window using the max\_processed\_datetime  IF input\_end\_process\_datetime IS NULL THEN  SET last\_received\_datetime = (  SELECT MIN(meta\_received\_time) meta\_received\_time  FROM (  SELECT MAX(soat.meta\_received\_time) meta\_received\_time FROM `ocado\_sol.sort\_attempts` soat  UNION ALL  SELECT MAX(sost.meta\_received\_time) meta\_received\_time FROM `ocado\_sol.sorting\_states` sost  UNION ALL  SELECT MAX(unit.meta\_received\_time) meta\_received\_time FROM `ocado\_sol.unload\_items` unit  UNION ALL  SELECT MAX(unlo.meta\_received\_time) meta\_received\_time FROM `ocado\_sol.unloads` unlo  )  );  ELSE  SET last\_received\_datetime = input\_end\_process\_datetime;  END IF;   -- Build Stage Table  CREATE OR REPLACE TEMP TABLE `stg\_machine\_hourly\_aggregate` AS (  WITH  date\_to\_process AS (  SELECT \*  FROM ocado\_sol.date\_interval(start\_process\_datetime)  ),  machine\_list AS (  SELECT DISTINCT  sos.meta\_orb\_site,  osi.timezone,  sos.meta\_machine\_name,  sos.meta\_deploy\_mode  FROM  `ocado\_sol.sorting\_states` sos  JOIN  `ocado\_sol.orb\_site\_info` osi ON osi.orb\_site = sos.meta\_orb\_site  )  SELECT  mcl.meta\_orb\_site,  mcl.meta\_machine\_name,  mcl.meta\_deploy\_mode,  mcl.timezone,  dtp.start\_time,  dtp.end\_time,  DATETIME(dtp.start\_time, mcl.timezone) local\_start\_time,  DATETIME(dtp.end\_time, mcl.timezone) local\_end\_time  FROM  machine\_list mcl  CROSS JOIN  date\_to\_process dtp  WHERE  dtp.start\_time BETWEEN start\_process\_datetime AND last\_received\_datetime  ORDER BY  start\_time  );   -- Build aggregate  CREATE OR REPLACE TEMP TABLE stg\_hourly\_sorting\_aggregate AS (  WITH  sort\_attempt\_aggregate AS (  SELECT  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time,  COUNT(\*) total\_attempts,  SUM(CASE WHEN outcome = 'complete' THEN 1 ELSE 0 END) successful\_attempt,  SUM(CASE WHEN outcome = 'partial' THEN 1 ELSE 0 END) partial\_attempt  FROM  `stg\_machine\_hourly\_aggregate` sha  LEFT OUTER JOIN  `ocado\_sol.sort\_attempts` sat ON (  sat.meta\_orb\_site = sha.meta\_orb\_site  AND sat.meta\_machine\_name = sha.meta\_machine\_name  AND sat.meta\_deploy\_mode = sha.meta\_deploy\_mode  AND sat.meta\_event\_time BETWEEN sha.start\_time AND sha.end\_time  AND sat.meta\_received\_time BETWEEN start\_process\_datetime AND last\_received\_datetime  )  GROUP BY  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time  ),  sort\_state\_aggregate AS (  SELECT  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time,  SUM(CASE WHEN sos.state = 'sorting' THEN TIMESTAMP\_DIFF(sos.end\_time, sos.start\_time, SECOND) ELSE 0 END) sorting\_seconds,  SUM(CASE WHEN sos.state = 'blocked' THEN TIMESTAMP\_DIFF(sos.end\_time, sos.start\_time, SECOND) ELSE 0 END) locked\_seconds,  SUM(CASE WHEN sos.state = 'scheduled\_downtime' THEN TIMESTAMP\_DIFF(sos.end\_time, sos.start\_time, SECOND) ELSE 0 END) scheduled\_downtime\_seconds,  SUM(CASE WHEN sos.state = 'unscheduled\_downtime' THEN TIMESTAMP\_DIFF(sos.end\_time, sos.start\_time, SECOND) ELSE 0 END) unscheduled\_downtime\_seconds  FROM  `stg\_machine\_hourly\_aggregate` sha  LEFT OUTER JOIN  `ocado\_sol.sorting\_states` sos ON (  sha.meta\_orb\_site = sos.meta\_orb\_site  AND sha.meta\_machine\_name = sos.meta\_machine\_name  AND sha.meta\_deploy\_mode = sos.meta\_deploy\_mode  AND sos.meta\_event\_time BETWEEN sha.start\_time AND sha.end\_time  AND sos.meta\_received\_time BETWEEN start\_process\_datetime AND last\_received\_datetime  )  GROUP BY  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time  ),  units\_aggregate AS (  SELECT  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time,  COUNT(\*) number\_of\_units  FROM  `stg\_machine\_hourly\_aggregate` sha  LEFT OUTER JOIN  `ocado\_sol.unload\_items` uni ON (  uni.meta\_orb\_site = sha.meta\_orb\_site  AND uni.meta\_machine\_name = sha.meta\_machine\_name  AND uni.meta\_deploy\_mode = sha.meta\_deploy\_mode  AND uni.meta\_event\_time BETWEEN sha.start\_time AND sha.end\_time  AND uni.meta\_received\_time BETWEEN start\_process\_datetime AND last\_received\_datetime  )  GROUP BY  sha.meta\_orb\_site,  sha.meta\_machine\_name,  sha.meta\_deploy\_mode,  sha.start\_time,  sha.end\_time  )  SELECT  soat.meta\_orb\_site,  soat.meta\_machine\_name,  soat.meta\_deploy\_mode,  soat.start\_time,  soat.end\_time,  soat.total\_attempts,  soat.successful\_attempt,  soat.partial\_attempt,  sost.sorting\_seconds,  sost.locked\_seconds,  sost.scheduled\_downtime\_seconds,  sost.unscheduled\_downtime\_seconds,  unag.number\_of\_units  FROM  sort\_attempt\_aggregate soat  JOIN  sort\_state\_aggregate sost ON (  soat.meta\_orb\_site = sost.meta\_orb\_site  AND soat.meta\_machine\_name = sost.meta\_machine\_name  AND soat.meta\_deploy\_mode = sost.meta\_deploy\_mode  AND soat.start\_time = sost.start\_time  AND soat.end\_time = sost.end\_time  )  JOIN  units\_aggregate unag ON (  soat.meta\_orb\_site = unag.meta\_orb\_site  AND soat.meta\_machine\_name = unag.meta\_machine\_name  AND soat.meta\_deploy\_mode = unag.meta\_deploy\_mode  AND soat.start\_time = unag.start\_time  AND soat.end\_time = unag.end\_time  )  );   IF table\_count > 0 THEN  MERGE INTO `ocado\_sol.hourly\_sorting\_aggregate` hsa  USING (  SELECT \*  FROM stg\_hourly\_sorting\_aggregate  ) shsa ON (  hsa.meta\_orb\_site = shsa.meta\_orb\_site  AND hsa.meta\_deploy\_mode = shsa.meta\_deploy\_mode  AND hsa.meta\_machine\_name = shsa.meta\_machine\_name  AND hsa.start\_time = shsa.start\_time  AND hsa.end\_time = shsa.end\_time  )  WHEN MATCHED THEN  UPDATE SET  hsa.total\_attempts = shsa.total\_attempts,  -- hsa.successful\_attempt = shsa.successful\_attempt,  hsa.partial\_attempt = shsa.partial\_attempt,  hsa.sorting\_seconds = shsa.sorting\_seconds,  hsa.scheduled\_downtime\_seconds = shsa.scheduled\_downtime\_seconds,  hsa.unscheduled\_downtime\_seconds = shsa.unscheduled\_downtime\_seconds,  hsa.number\_of\_units = shsa.number\_of\_units  WHEN NOT MATCHED THEN  INSERT (  meta\_orb\_site,  meta\_machine\_name,  meta\_deploy\_mode,  start\_time,  end\_time,  total\_attempts,  sorting\_seconds,  scheduled\_downtime\_seconds,  number\_of\_units  )  VALUES (  meta\_orb\_site,  meta\_machine\_name,  meta\_deploy\_mode,  start\_time,  end\_time,  total\_attempts,  sorting\_seconds,  scheduled\_downtime\_seconds,  number\_of\_units  );  ELSE  CREATE OR REPLACE TABLE ocado\_sol.hourly\_sorting\_aggregate AS (  SELECT \*  FROM stg\_hourly\_sorting\_aggregate  );  END IF;   END; |
| --- |

#### 

#### Code for table Function calculating time interval

| CREATE OR REPLACE TABLE FUNCTION `aqueous-walker-379718.ocado\_sol.date\_interval`(date\_param TIMESTAMP) AS ( WITH intervals AS (  SELECT  GENERATE\_TIMESTAMP\_ARRAY(  TIMESTAMP\_TRUNC(TIMESTAMP(date\_param), HOUR),  TIMESTAMP\_ADD(TIMESTAMP\_TRUNC(TIMESTAMP(date\_param), HOUR), INTERVAL 23 HOUR),  INTERVAL 1 HOUR  ) AS ts\_array ), extended\_intervals AS (  SELECT  ARRAY\_CONCAT(ts\_array, [TIMESTAMP\_ADD(TIMESTAMP\_TRUNC(TIMESTAMP(date\_param), HOUR), INTERVAL 24 HOUR)]) AS extended\_ts\_array  FROM intervals ) SELECT  CAST(FORMAT\_TIMESTAMP('%F %T', start\_ts) AS TIMESTAMP) AS start\_time,  CAST(FORMAT\_TIMESTAMP('%F %T', end\_ts) AS TIMESTAMP) AS end\_time FROM extended\_intervals CROSS JOIN UNNEST(extended\_ts\_array) AS start\_ts JOIN UNNEST(extended\_ts\_array) AS end\_ts ON start\_ts = TIMESTAMP\_SUB(end\_ts, INTERVAL 1 HOUR) ORDER BY start\_time ); |
| --- |

# 

# Questions Asked

1. How would you generate this table from the preceding ones? Assume the rows in the raw tables are immutable (once inserted they will never change), but you need to account for data showing up late (up to a limit, assume 1 to 2 days after being recorded). Write down the code and outline the orchestration process / infrastructure needed.

Answer:

* The solution utilizes BigQuery stored procedures and table functions for implementation. The code for the stored procedure can be found in the Appendix section of the document. The execution and explanation of the stored procedure are provided at the beginning of the document.
* To schedule the job, the BigQuery Transfer service's scheduled query functionality is used. This allows for automated execution of the procedure at specified intervals. Additionally scheduling could be done using Cloud function Gen 2 or pub/sub messages. The preferred approach is to have processing within BigQuery.
* The scheduled procedure accepts inputs for the start datetime and end datetime, enabling batch processing for a specific date range.
* Additionally, the job can handle late-arriving data by specifying a sliding window scheduled hour. This ensures that data received after the initial batch can still be processed correctly.
* The logic for high watermarks will ensure late coming data are handled.

1. How would you update this aggregation to handle late arriving data in an eﬃcient manner? Assuming this table has downstream dependencies, how would you update all tables in the dependency tree?

Answer:

* The solution handles late arriving data by using a high watermark approach, ensuring that the processing window only includes data with a complete set. This helps to maintain data integrity and consistency during the aggregation process.
* To initiate downstream processing, you can define a trigger point by sending a pub/sub message. Once the data has been aggregated, you can trigger the downstream process by publishing a message to the appropriate pub/sub topic. This allows you to seamlessly integrate the aggregated data with your downstream systems or processes, enabling further analysis, reporting, or any other required actions.
* By using pub/sub as a messaging mechanism, you can decouple the aggregation process from the downstream processing, providing flexibility, scalability, and reliability in handling and distributing the aggregated data.
* Regenerate response

1. If we need this derived data available in near real time, outline the infrastructure you would set up.

Answer:

* The proposed solution suggests using a streaming approach by utilizing Pub/Sub Topic subscription to load real-time data into a BigQuery table. This allows for immediate ingestion of data into BigQuery, enabling real-time processing.
* To trigger the processing of the data in near real-time, you can utilize Cloud Function Gen2 or Pub/Sub messaging. These mechanisms can be used to invoke stored procedures or initiate processing logic on the data.
* Another alternative is to employ Apache Beam, a unified programming model, to process the data within a sliding window. This approach enables the processing of incoming data as it arrives. Additionally, you can handle late-arriving data by incorporating it into the sliding window processing or by reprocessing the entire dataset.

Both approaches offer solutions for near real-time data processing in BigQuery, allowing you to handle real-time data streams and perform the necessary transformations and aggregations as data arrives.

1. Data between robot services uses protobuf messages created by GOlang services, how would you eﬃciently handle schemas across edge and cloud infrastructure

Answer

The proposed solution involves using Pub/Sub for handling messages from robotic/IoT devices, which are published and consumed by BigQuery Subscription. Pub/Sub supports the transmission of protobuf messages created by Golang services, providing the following capabilities:

1. Message Encoding: Pub/Sub allows to specify the encoding of the messages during publishing and subscribing. Protobuf messages can be encoded using the binary format, which is the default encoding for protobuf messages. Pub/Sub fully supports binary data transmission, ensuring compatibility with Golang-generated protobuf messages  
   .
2. Subscription and Deserialization: Subscribers of a Pub/Sub topic can receive messages that contain protobuf payloads. Whether the subscriber application is implemented in Golang or any other programming language, it can deserialize the payload into the corresponding protobuf message structure. Golang, being the language used to generate the protobuf messages, provides native support for deserialization.
3. Schema Evolution: Pub/Sub enables flexible schema evolution by decoupling the publishing and subscribing components. If there are changes made to the protobuf message structure over time, such as adding or removing fields, subscribers can handle different versions of the schema. Pub/Sub allows subscribers to adapt their deserialization logic accordingly, ensuring compatibility and seamless processing of messages with varying schemas.

In summary, Pub/Sub provides the necessary support for handling protobuf messages created by Golang services. It handles message encoding, facilitates subscription and deserialization of protobuf payloads, and allows for flexible schema evolution, ensuring efficient communication and data processing.