University of Groningen

Scalable Computing

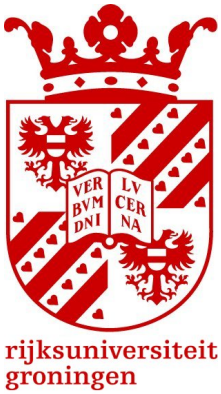
# Scalable Temperature Processing $\lambda$-Architecture

*Group:* 3

*Authors*

| | |
|---|---|
| Mark Soelman | s3224708 |
| Hindrik Stegenga | s3860205 |
| Filipe Capela | s4040112 |

March 26, 2020



# Contents

# 1 Introduction

This document provides an overview of the work carried out during the course of Scalable Computing 2019-2020. The aim of this document is *not* to walk through our code, but rather to highlight the roles and high-level functionality that each component provides within the system. We also try to highlight our design decisions.

First, a diagram of our overall architecture is given in Section 3. Hereafter, we elaborate on each component in our system in Section 4. We explain the components in chronological order, that is, from source to sink. In Section 5 an explanation of the deployment on Kubernetes is made.

# 2 Dataset

The dataset that we will be processing with our application is the KNMI temperature dataset. A link to the website from which these measurements were taken is https://projects.knmi.nl/klimatologie/uurgegevens/selectie.cgi. This comprises several metrics from spread out stations across the Netherlands (with the total number of stations being 50). As mentioned before, we will be focusing on the temperature measurements. The rate at which measurements are taken is hourly, with the dataset containing data between the 1st of January 1951, until the current date. Since we will be computing yearly averages, we decided to restrict it so that we obtain measurements until the 31st of December 2019.

The measurements come in the following shape: (280,20200313,14,89)

- 280 - an id which represents each station (unique per station)
- 20200313 - the day with the format yyyyMMdd
- 14 - the hour of said that for which the measurement is taken
- 89 - the desired temperature measurement which, in this example, corresponds to 8.9°C

Since this is a limited dataset, we created a separate application to generate data using the same format, which can run infinitely, creating data for multiple stations as well. This allows us to tackle the problem of running out of data.

Some of the measurements are missing, since some stations might not send the measurements for all hours or fail to measure the temperature, this can also be aided with the data simulator, since this creates temperatures for all hours of the day.

# 3 Overview of Architectural Components

sIn this section we present the applications pipeline, where the flow of data is demonstrated, as well as the components present in the system. Each of the said components is analyzed in-dept in Section 4.

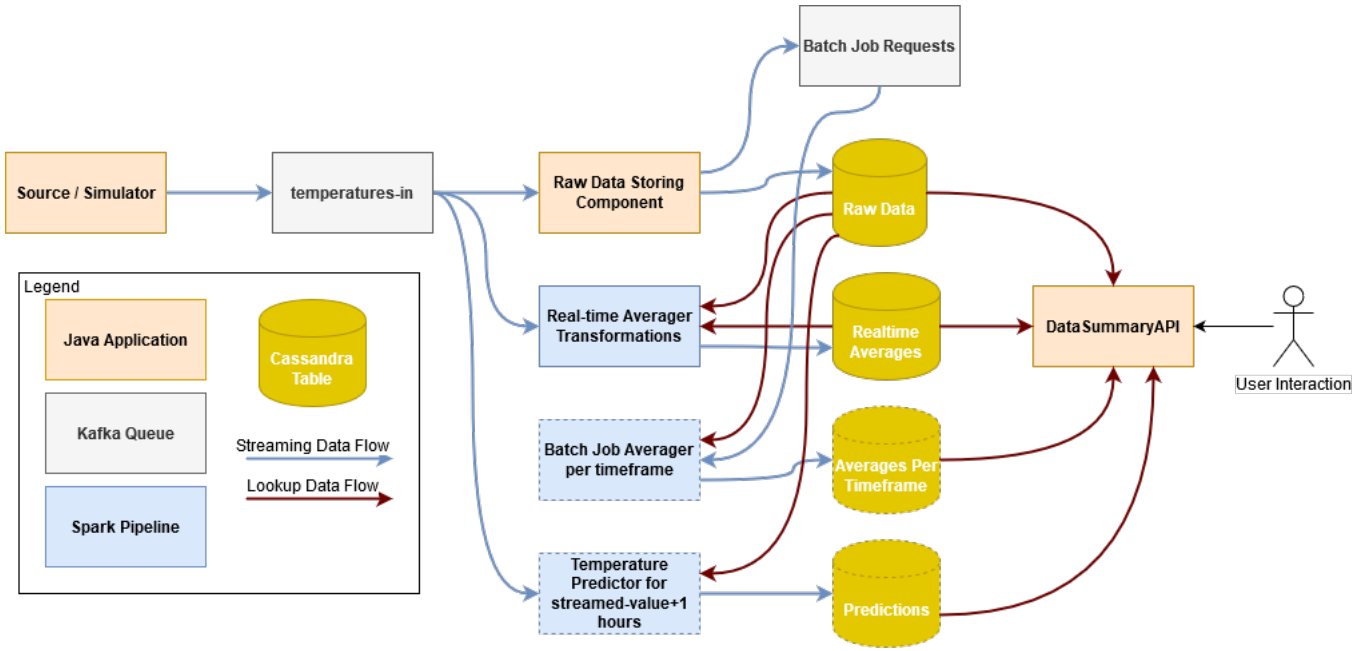



Figure 1: Our project's pipeline

# 4 Component Elaboration

In this Section, we highlight all components of our system, with the focus on high-level functionality, responsibilities and design decisions.

## 4.1 Simulator

Our pipeline starts with the simulator of weather station measurements. This simulator is written in Java 8. This component is responsible for outputting *real values* from our KNMI data set, as described in Section 2. Besides real values, this component is capable of outputting *simulated values*. Simulation logic is based on mostly 3 configuration options, discussed below.

Although threads can be used to simulate multiple stations, this component can also be deployed multiple times using a single thread. This is flexible, and depends on the preference of the one deploying the simulator.

- `THREADS`: This environment variable is the number of threads that are generating temperature measurements. Each thread has its own station ID, so the more threads we have, the more stations we simulate.
- `ONLY_FAKE_DATASETS`: If this environment variable is set to *false*, the first thread will output the real KNMI measurements, while other threads output simulated measurements. If set to *true*, all threads output only simulated measurements.
- `FIXED_DELAY_IN_MS`: The minimum time in milliseconds a station sleeps after generating a value. In reality, this would be an hour, but to obtain high volumes, we set this to between 1 to 10 seconds.

This component is not written solely in Java 8, but uses the *Spring Framework*, since it allows us to write the application much quicker because of the following reasons: Dependency Inversion, Configuration management, Using environment variables, Integrating with the Kafka Driver in an OOP-like fashion, serialization of OOP-models to JSON.

## 4.2 Kafka

The Kafka component is responsible for directing and controlling the incoming stream of data. It is a Message Queue component, which implements the Message Queue pattern for improving consistency and redundancy for the inserted data. All the incoming measurements are sent to the Kafka cluster, which makes sure the measurements are replicated and partitioned across the cluster.

Kafka itself also requires Zookeeper for service discovery, which is an additional added component required. Therefore, we added a zookeeper instance to keep track of all the available brokers.

## 4.3 Storage Middleware

To understand why we included this component, a little bit of background information is required. Our first attempt was to program all logic in Spark, i.e.: a single stream of temperature values enter the spark cluster, which performs all processing, and finally outputs the results using multiple streams.

Unfortunately, above approach did not work for us, since it appears Spark Structured Streaming only supports a single output stream. For instance, if we want to output multiple types like predictions, raw measurements, batch averages, and streaming averages, we cannot create different output streams per data type.

### 4.3.1 Responsibilities

In order to overcome this limitation, we completely changed our architecture: We decided to store raw measurements outside of spark using this component (which was a better choice, since spark does not operate on these raw measurements anyway), and we created a different spark application (or job) per pipeline. In the end, we obtained the following responsibilities of the storage middleware:

1. Reading newly created temperature measurements from Kafka
2. Store raw measurements in Cassandra asynchronously
3. Create a new job request to compute the daily average, if the raw measurement is the last measurement of the day.
4. Create a new job request to compute the weekly average, if the raw measurement is the last measurement of the week, i.e. on a Sunday.
5. Create a new job request to compute the monthly average, if the raw measurement is the last measurement of the month.
6. Create a new job request of variable length by submitting a POST-request. This is for performance evaluation of the algorithm for the demo.

## 4.4 Spark: Temperature Averager for Realtime Measurements

Our first Spark job computes realtime averages for each incoming measurement. Instead of computing the average for each calendar day, week, or month, we decided to compute the **Moving Average**. This decision was made, since we already wanted to compute the average per calendar day, week, or month using the batch processing.

### 4.4.1 Formulas

We challenged ourselves to create an algorithm in Spark that is as efficient as possible. Therefore, we did not use the naive approach of computing the average by summing up all temperatures, and dividing over the number of measurements, no. We compute the averages by taking the last-known average up until the previous measurement, multiply by the divisor, subtract the measurement that needs to roll out, and divide again by the new divisor. See the following formulas.

Formula for computing the **Cumulative Moving Average**. This formula is used whenever we want to add a new value to an existing average of N elements, without an older value sliding out.

$$\bar{x_t} = \frac{x_{t-1} * N + x_t}{N+1}$$

Formula for computing the **Sliding Moving Average**, provided the window of the previous moving average was filled. For instance, we can have a moving average over 24 hours. Then, to add a new measurement, the 24-th lagged value needs to slide out (be subtracted) of our average.

$$\bar{x_t} = \frac{N * x_{t-1} - x_{t-N} + x_t}{N}$$

### 4.4.2 Why not Sliding Window?

Another design decision we made, was to restrict ourselves to not using the built-in Sliding Window functionality of Spark. The reason for this, is that in the real world without simulated values, there will be on average 1 hour between every measurement. If we use a sliding window of a month, then the spark application needs to keep track of the values over the past month. Since our spark cluster is stateless, we do not want to keep track of information for such a long timespan. Therefore, we obtain the required measurements from the database.

### 4.4.3 Computation Step-By-Step

#### Importing DataSets

1. `KafkaSource`: `InputStream` for obtaining a `Dataset[Measurement]` from Kafka. These are the streaming measurements.
2. `CassandraMeasurementImport`: Allowing us to read a `Dataset[Measurement]` from Cassandra, containing all historic (raw) values.
3. `CassandraAveragesImport`: Allowing us to read a `Dataset[StationAverages]` from Cassandra, containing all moving averages of stations up until now.

#### Computing the new Moving Average

1. For every new temperature measurement of a station, we want to compute the new moving average, that includes this new value.
2. Select all historic measurements, with 2 additional columns: (1) a `lagIdx` which is a row number of all measurements per station. Most recent measurement has value 0, whereas the row before value 1, etc. (2) a `numRecords` column with the number of rows so far, per station.

3. Using the `lagIdx`, we filter the measurements to only obtain 3 rows: Measurement with `lagIdx` 23 to move out the daily moving average; Measurement with `lagIdx` 24*7-1 to move out the weekly moving average; and finally Measurement with `lagIdx` 24*31-1 to move out of the monthly moving average.

4. Join the streaming-temperature measurement (1 row) with the moving averages of the stations (1 row). (The only way to use data of multiple dataframes on workers, is to join them.)

5. The streaming temperature measurement + its moving average (1 row) is left-joined with the 3 rows to roll out from step (3). Again, we end up with 1 row.

6. Finally, we compute the new (cumulative/sliding) moving average and map this to a new `StationAverages`: a single model containing the station ID, average per day, average per week, average per month.

**Exporting the data** The updated `StationAverages` model is directly stored in a Cassandra table.

## 4.5 Spark: Temperature Averages for Batch Intervals

As mentioned earlier, we wanted to do different computations for the batch and streaming values, to keep it more challenging and interesting. For our batch intervals, we compute averages for certain timeframes:

1. *Daily*: Compute the average for a single day, for hour 1 to 24.
2. *Weekly*: Compute the average for every week, i.e. from Monday up to and including Sunday.
3. *Monthly*: Compute the average for every calendar month, i.e. from the 1st of a month, up to and including the last day of the month.

### 4.5.1 A single program for multiple timeframes

We made an important design decision for this component: we decided to use the same algorithm for all timeframes. The main reason for this, is that we wanted to experiment for different batch sizes.

While we have a single program, we still want to compute the averages for multiple timeframes, and store these in different tables in order to keep our data organized: 1 table for daily averages, 1 table for weekly averages, and 1 table for monthly averages. While submitting the program as a job, the timeframe can be specified as first argument.

Alternative 1: it would have been possible to compute the weekly/monthly averages based on daily averages (instead of raw hourly measurements). But this required an entirely different algorithm for computing the daily average as opposed to the weekly/monthl, which we did not want.

Alternative 2: it would have been possible to eliminate this batch component entirely. In our Spark streaming application for moving averages, we could simply store the moving average at the end of each day/week/month. This approach would eliminate the need for this batch component (or job). However, since we wanted to show we are capable of batch processing as well, we

still wanted to include this, albeit somewhat overlapping, batch processing component.

### 4.5.2 Computation Step-by-Step

#### Importing Datasets

1. `KafkaBatchJobSource`: `InputStream` for obtaining a Dataset[BatchJobRequest] from Kafka. These are streaming jobs flowing in our spark application. Each job contains a batch request to compute the average of a certain timeframe (day, week, month).
2. `CassandraMeasurementImport`: Allowing us to read a `Dataset[Measurement]` from Cassandra, containing all historic (raw) values.

#### Computing the new Moving Average

1. For every new `BatchJobRequest`, we want to compute the average within the given interval.
2. The Job Request is inner-joined with all historic measurements that are within the start- and end date.
3. The following 2 columns are added to each (measurement+job) row: (1) `totalSummed` with literal value 1, which is used to keep track of the total number of measurements we need to divide by, when computing the average. (2) `temperature` which is the measured temperature, but then cast from integer to a double. This is required for the reduce step later on, to ensure all data types used are consistent.
4. Group step: The joined data is grouped by the `station`, `start_date`, and `end_date`. The start- and end date are included to ensure that computations of different jobs are separated.
5. Reduce step: The new reduced measurement sums up the temperatures and `totalSummed` of both measurements.
6. Map step: The reduced result is mapped to a `BatchJobAverage` object.

**Exporting the data** The `BatchJobAverage` object is directly stored in a Cassandra table, depending on the argument of the job (daily, weekly, monthly).

## 4.6 Spark: Temperature Predictor for Streaming Measurements

This algorithm is in charge of computing the average for the hour after the hour that is contained in the new measurement which comes from the streaming pipeline. That is, if we receive a measurement corresponding to the date: 30-03-2020 08:00, we will be computing the prediction for the date: 30-03-2020 09:00.

For this prediction 3 different types of measurements were considered:

- The value for that data from the past year, for the above example would be 30-03-2019 09:00
- The new measurement which comes in from the streaming pipeline, for the above example is 30-03-2020 08:00
- The last 22 hours before the new measurement, so that we can have the 23 measurements before the hour we want to predict

The way the predicted value is computed depends on whether the measurements are present or not.

If there are values for the past year, as well as 22 measurements corresponding to the hours before the new measurement, the following formula is applied:

- A weight of 50% is assigned to the last year measurement
- A weight of 25% is assigned to the new measurement
- A weight of 25% is assigned to the last 22 records using the following criteria:
  - If the measurement corresponds to the most recent 2 hours, or the oldest 3 hours, that means that we can incorporate seasonality to our prediction.

- For the example of predicting for 09:00, we contain the 08:00 in the streamed measurement, 07:00 and 06:00 as the 2 most recent hours and 10:00, 11:00 and 12:00 (from the day before) in the oldest 3 hours of these 22 records.
- A weight of 60% of the 25% available is distributed by these 5 records, and the remaining 40% of the 25% is distributed among all the remaining hours, hence adding seasonality to the algorithm.

If a value is missing, then the average is computed according to an adjustment in the weights.

- If there are still less than 22 records, only the most recent 2 hours get weights assigned, while the remaining hours are disregarded.
- If there is no last year measurement, the weight of that measurement get assigned to the new measurement.

### 4.6.1 Formulas for the prediction

For most cases, when there are a considerable amount of data in the system, the prediction is computed according to the following formula:

$$\bar{prediction} = 0.5 * x1 + 0.25 * x2 + 0.25 * x3$$

Where x1 = `the temperature from the same date and hour as the predicted temperature, minus 12 months`, x2 = `new temperature from the stream` and `x3 = a weighted average of the 22 historic measurements before the streamed value's date`.

If the value of x1 is not present, the formula adjusts to:

$$\bar{prediction} = 0.3 * x2 + 0.7 * x3$$

Note that the above formulas are applied if the value of x3 is not present, since in both cases the value of x3 is is assigned to the same as x2 (since x2 will always exist).

### 4.6.2 Computation Step-by-Step

#### Importing Datasets

1. `KafkaSource`: `InputStream` for obtaining a Dataset[Measurement] from Kafka. These are the streamed measurements flowing in our spark application.

2. `CassandraMeasurementImport`: Allowing us to read a `Dataset[Measurement]` from Cassandra, containing all historic (raw) values.

#### Computing the prediction

1. The first step is to join the new measurement with the last year measurement, to do so a left-join is applied to the historic measurements, filtering it so only a measurement corresponding to the predicted date minus 12 months is retrieved.

2. After this step a dataframe composed of the station, the `lastYearDate`, the `lastYearTemperature` and the remaining columns from `newMeasurement` are joined as a Prediction object.

3. In order to retrieve the last 22 records before the date and hour of the new measurement, the historic measurements are indexed using the same logic from the streaming averager, but after indexing, only the records which have an index lower than 23 are retrieved.

4. To obtain the dataframe with all the data required to calculate the prediction, a join between the `Prediction` object created at step 2 and the 22 records obtained from the last step is performed, using the column `station`. This last object is a `PredictionReducable` and contains all columns from `Prediction` and `historic values`, plus a `predictedHour`, which is the modulo of the streamed hour by 24, increased by +1. There is a `predictedDate`, which is the same as the `newDate`, with the exception of the case where the `newHour` hour is 24, moving the `predictedDate` to the day after. There is also a

`predictedTemperature`, which is a placeholder for the final temperature, and a `totalWeightSum` which is a placeholder for the weights of each measurement, for the Auto-Regressive prediction.

5. Group step: The joined data is grouped by the `station`, `newDate`, and `newHour`. The `newDate` and `newHour` of the `newMeasurement` are included since this is a only unique per station.

6. Reduce step: This step is where the logic of the weights is introduced, here is where the `getWeight` function is called and assigns, according to the index, the corresponding weight to each of the 22 records from historic data, after being reduced. Each records index will be updated to -1 so that no more weights are executed, ending the reduce step when all the indexes are at -1.

7. Map step: The reduced result is mapped to a `Measurement` object, where the `predictedTemperature` is calculated using the `getFinalPrediction` method, where, according to the weights and availability of the measurements, the final prediction is computed.

**Exporting the data** The `Measurement` object is directly stored in a Cassandra table, in the Predictions table.

### 4.6.3 Results

In the below figures we can see the comparison between real measurements from the KNMI dataset, and the corresponding predicted values outputted from our application.

| station | date | hour | temperature |
|---|---|---|---|
| 210 | 1951-01-07 00:00:00 | 1 | 84 |
| 210 | 1951-01-07 00:00:00 | 2 | 80 |
| 210 | 1951-01-07 00:00:00 | 3 | 78 |
| 210 | 1951-01-07 00:00:00 | 4 | 72 |
| 210 | 1951-01-07 00:00:00 | 5 | 66 |
| 210 | 1951-01-07 00:00:00 | 6 | 58 |
| 210 | 1951-01-07 00:00:00 | 7 | 60 |
| 210 | 1951-01-07 00:00:00 | 8 | 60 |
| 210 | 1951-01-07 00:00:00 | 9 | 58 |

(a) KNMI original measurements

| station | date | hour | temperature |
|---|---|---|---|
| 210 | 1951-01-07 00:00:00 | 1 | 84 |
| 210 | 1951-01-07 00:00:00 | 2 | 84 |
| 210 | 1951-01-07 00:00:00 | 3 | 81 |
| 210 | 1951-01-07 00:00:00 | 4 | 78 |
| 210 | 1951-01-07 00:00:00 | 5 | 74 |
| 210 | 1951-01-07 00:00:00 | 6 | 68 |
| 210 | 1951-01-07 00:00:00 | 7 | 60 |
| 210 | 1951-01-07 00:00:00 | 8 | 59 |
| 210 | 1951-01-07 00:00:00 | 9 | 60 |
| 210 | 1951-01-07 00:00:00 | 10 | 58 |

(b) Predicted measurements

Figure 2: Measured vs Predicted

## 4.7 API for summarizing data

Finally, we have arrived at the end of our pipeline(s). Here, the API can be found for summarizing the data. This API was written in Java 8 and the Spring Framework. The API fetches all data on request from the Cassandra database in a reactive, non-blocking fashion, while outputting it using the JSON format. This API consists of two main endpoints, as discussed next.

### 4.7.1 Station Overview Endpoint

The main endpoint can be found at the website root `/`. This endpoint provides an overview of all stations in the system and their daily, weekly and monthly moving averages. All of these averages are near-realtime, i.e. constantly being updated in the database.

### 4.7.2 Station Summary Endpoint

The station summary endpoint can be located by navigating to the following endpoint: `/{stationID}`. This endpoint serves three different use cases.

**UC1: Obtaining Overall Summary** To obtain the overall summary, no arguments need to be passed to the endpoint. The endpoint returns the following information.

```
{
    stationAverage, // Realtime moving averages
    monthlyAverages: [], // All averages of calendar months
    weeklyAverages: [], // All averages of calendar weeks
    dailyAverages: [], // All averages of days
    measurements: [], // All raw measurements starting from
        the latest daily average. In other words: all measurements that
        have not yet been included in a daily average.
    predictions: [] // All predictions starting from the latest measurement.
```

```
        In other words: only the most recent predictions.
}
```

**UC2: Obtaining Overall Summary with offset** The following offset can be passed: `startDate=yyyy-MM-dd`. The endpoint returns the following information.

```
{
    stationAverage, // Realtime moving averages
    monthlyAverages: [], // All averages of calendar months since startDate
    weeklyAverages: [], // All averages of calendar weeks since startDate
    dailyAverages: [], // All averages of days since startDate
    measurements: [], // All measurements that have not yet been included
        in a daily average, and are after startDate
    predictions: [] // Only the most recent predictions, after the startDate
}
```

**UC3: Obtaining Overall Summary between two dates** The following arguments can be passed: `startDate=yyyy-MM-dd`, `endDate=yyyy-MM-dd`. The endpoint returns the following information.

```
{
    stationAverage, // Realtime moving averages
    monthlyAverages: [], // All averages of calendar months
        between startDate and endDate
    weeklyAverages: [], // All averages of calendar weeks
        between startDate and endDate
    dailyAverages: [], // All averages of days between startDate and endDate
    measurements: [], // All measurements between startDate and endDate
    predictions: [] // All predictions between startDate and endDate
}
```

## 4.8 Cassandra

For our storage database we decided to utilize Cassandra, since for the use-case of storing streamed data, Cassandra is the most suited for this operation due to its row-column approach, and from the fact that this data will not be changed (an operation which is hard to complete with Cassandra).

We utilize a Cassandra StatefulSet, which is comprised of 3 Cassandra nodes to ensure data correctness, availability and persistency.

Each table structure will be displayed below:

- Raw Data
  - station
  - date
  - hour
  - temperature
- Streamed averages
  - station
  - daily_average
  - weekly_average
  - monthly_average
- Predictions
  - station
  - date
  - hour
  - temperature (predicted)
- Batch Averages (one for daily, weekly and monthly)
  - station
  - date
  - temperature

# 5 Deployment on Kubernetes

## 5.1 Pods and Services

In order to make deployment easier and create a uniform and robust way to deploy an instance of our infrastructure, we decided to use Kubernetes. Our infrastructure consists of quite a few semi-dependent components:

- 3-N Simulator instances
- 3 Cassandra instances storing our data redundantly
- 3-N Kafka brokers
- 3-N Storage Middleware instance
- 1 Zookeeper instance
- 1 Spark master
- 6-N Spark workers
- 3-N Data Summary API

To make all these components work together we use Kubernetes pods and services to allow for a reproducible and easy to deploy system. Data communication between the pods is handled either by Kubernetes' services using DNS, or by Zookeeper in case of the Kafka brokers. By making use of these services, we can omit having to hardcode IP adresses into the containers, but simply resolve everything by hostnames. This allows for flexible and easy to setup communication, especially when using headless services.

## 5.2 Automating deployment

Manually deploying all the required Kubernetes deployments is a lot of work, especially since we need to enforce order. This ordering is required because we need to set-up the Kafka topics after all brokers have been deployed for example. To make sure all pods successfully deploy we have created a deployment script which automates all of this for us. It basically allows for a one-click deployment which sets up all the required topics and other configurations for all of our containers.

## 5.3 Cloud Deployment

Our deployment for Kubernetes works both on a local minikube-based kubernetes cluster and a cloud base GKE cluster. This cluster is based on 6 instances each having 4 cores and 16GB of RAM. It is configured in such a way that we can exploit the parallelism spark offers, and also run multiple spark jobs simultaneously. This is configured by setting the amount of available cores per worker in Kubernetes, and configuring how many executors and cores are available per job. This way, we prevent contention on the spark cluster, and keep all tasks running even if one would require way more resources.

## 5.4 Custom operator (BONUS)

The last part we implemented is a custom kubernetes operator, which makes use of Custom Resource Definitions to make our deployment even easier. This allows us to easily specify some basic properties for our specific deployment in a CRD yaml file, and the operator takes care of the rest of the deployment; setting all the required pods, services and configurations for them. The Operator itself is implemented using Rust and the operator crate. It runs as a HTTP server and responds to the appropriate endpoints required for managing the CRDs.