Big Data - Milestone 2

Adrianna Klimczak, Adam Kowalczyk, Wojciech Replin and Marcel Wenka

November 29, 2021

1 Introduction

The aim of this milestone is to present:

- The description of data resources acquired from individual data sources, and for each of these resources:
  - The category of the data (e.g. location of public transport buses), the number of records, the features describing every record, the time period for which data were collected, data quality
  - The format of source data e.g. JSON, CSV, XML
  - In the case of data sets and/or streams developed to be used as an input for classification and regression: the list and meaning of independent and dependent features and the way raw data has been converted to develop data for machine learning purposes
- The way of acquiring, storing and analysing the data including the architecture of the system, tools and programming languages used to develop it
- The type of ML tasks (regression) and other analytical tasks
- The use of batch and stream processing for individual tasks
- Report from functionality tests

2 Project goal

The aim of our project is to predict air quality based on the weather conditions in selected cities around the world.

3 Data sources

We are using 2 data sources:

1. Air Quality `www.aqicn.org`
2. Weather `www.openweathermap.org/api`

In the following sections we'll describe how we're using each of them for the purposes of the project

3.1 Air Quality API

Air quality data is provided by `www.aqicn.org` API. It provides virtually unquoted (1000 requests/second), real-time data from all around the world. For the purposes of this project, we are querying this API by geographical location using `Geolocalized Feed` endpoint. An example response from this endpoint for New Delhi, India can be found in listing 1. We are using following fields from AQ API:

- `data.aqi`: air quality index (`scale`)

- `data.time.v`: Unix timestamp of the measurement
- `data.city.geo`: geographic coordinates of the requested location
- `data.forecast.daily`: daily forecast of pollution for the requested location


```
{
  "status": "ok",
  "data": {
    "aqi": 57,
    "idx": 7024,
    "city": {
      "geo": [28.63576, 77.22445],
      "name": "New Delhi US Embassy, India",
      "url": "https://aqicn.org/city/india/new-delhi/us-embassy"
    },
    "dominentpol": "pm25",
    "iaqi": { "pm25": { "v": 57 } },
    "time": {
        "s": "2021-10-17 19:00:00",
        "tz": "+05:30",
        "v": 1634497200,
        "iso": "2021-10-17T19:00:00+05:30"
    },
    "forecast": {
      "daily": {
        "pm25": [
          { "avg": 164, "day": "2021-10-15", "max": 170, "min": 159 },
          { "avg": 173, "day": "2021-10-16", "max": 174, "min": 159 },
          ...
        ]
      }
    }
  }
}
```


Listing 1: Response from `https://api.waqi.info/feed/delhi/?token=TOKEN`

Full documentation of this API can be found here https://aqicn.org/json-api/doc/

3.2 OpenWeatherMap API

Weather data is provided by https://openweathermap.org/api. It provides free real-time data subject to 60 requests/minute and 10^6 requests/month quotas. For the purposes of this project, we are querying the API by the city name. Example request for the weather in Delhi, India can be found in listing 2. We are using the following fields from the OpenWeatherMap API:

- `coord.lon`, `coord.lat`: coordinates of the requested location
- `main`: weather parameters
  - `temp`: air temperature
  - `pressure`: air pressure
  - `humidity`: air humidity

- `visibility`: visibility in meters
- `wind`: wind parameters
  - `speed`: wind speed
  - `deg`: wind angle in degrees
- `dt`: Unix timestamp of the measurement


```
{
  "coord": { "lon": 77.2167, "lat": 28.6667 },
  "weather": [
    {
      "id": 804,
      "main": "Clouds",
      "description": "overcast clouds",
      "icon": "04n"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 296.59,
    "feels_like": 296.92,
    "temp_min": 296.59,
    "temp_max": 296.59,
    "pressure": 1009,
    "humidity": 74,
    "sea_level": 1009,
    "grnd_level": 983
  },
  "visibility": 10000,
  "wind": { "speed": 4.24, "deg": 40, "gust": 9.69 },
  "clouds": { "all": 100 },
  "dt": 1634487119,
  "sys": { "country": "IN", "sunrise": 1634431975, "sunset": 1634473186 },
  "timezone": 19800,
  "id": 1273294,
  "name": "Delhi",
  "cod": 200
}
```


Listing 2: Response from `https://api.openweathermap.org/data/2.5/weather?q=Delhi,in&appid=TOKEN`

Full documentation of this API can be found here https://openweathermap.org/current

4 Architecture

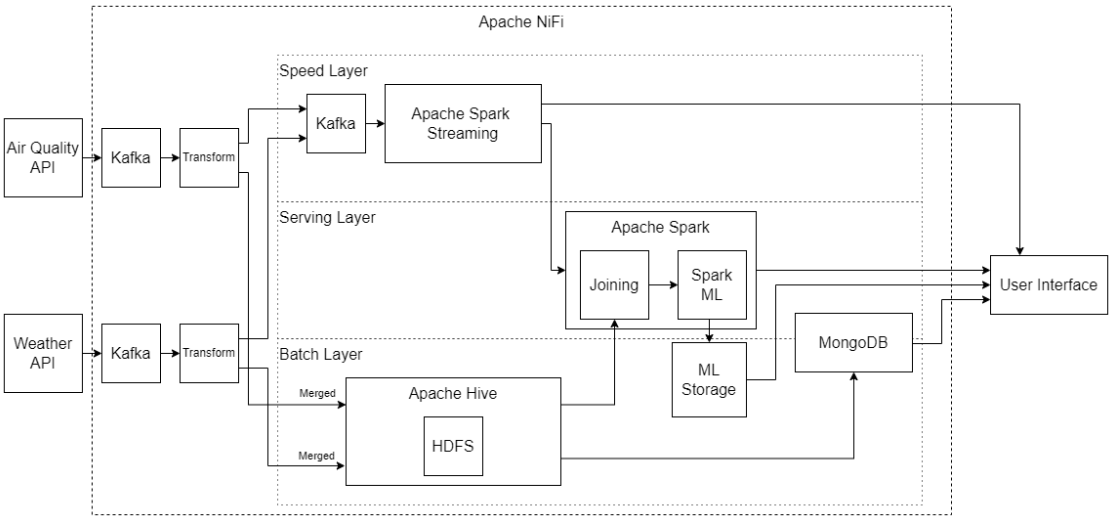

Figure 1: System architecture

4.1 Technologies

The following technologies are used in the current state of the project:

1. **Apache Nifi** is used to orchestrate the system and manage data flow. It also transforms the data and integrates other components.
2. **Apache Kafka** is used to receive real-time data feeds from the APIs and distribute it across the system.
3. **Apache Hive** is used to store batch data and provides SQL-like interface to query under-laying databases.

We also plan to incorporate other technologies in the later stages of development:

1. **Apache Spark Streaming** will be used to process streaming data and integrate it with other Spark components.
2. **Apache Spark + MLlib** will be used to feed the data to a machine learning model from MLlib library (contained in Spark Project) and handle the training process.
3. **MongoDB** will be used to store the data in non-relational database for the purpose of displaying it in the User Interface.
4. **UI** - React

4.2 Data flow

The data is acquired for 4 cities around the world with varying pollution patterns and levels: Delhi, Warsaw, Berlin and Moscow.

Firstly, the data is fetched from the APIs (3) by the Kafka running adequate Python scripts for each API. All endpoints are queried every 20 seconds and fed into Kafka with appropriate topic (corresponding to the city and pollution/weather).

NiFi collects the data by specifying the topic and transforms it so it only contains needed information. The data is filtered to remove meaningless features (e.g. ids). For this purpose JOLT JSON transformation

language is used. The FlowFiles, generated by these operations, are redirected back to Kafka (using other topic) in order to later integrate it with Spark Streaming, and are in parallel merged into batches of specified size (20 by default).

The merged data is then directly saved to Hives' HDFS, where the files are kept in separate directories for each city. In Hive we have created external tables build on top of these directories that can be queried by SQL-like syntax.

In the next stages of the project we plan to direct the data from Kafka to Spark components, where the pollution data will be joined with the weather data and fed to ML Model. We plan to save the trained parameters of the model to ML Storage. The data from the Hive will be transferred to MongoDB database so it can be used to display the past measurements and weather on the UI.

4.3 Lambda architecture standardisation

Our system uses Lambda architecture, meaning it takes advantage of both batch and stream-processing data. On the diagram (1) the components where divided into three layers:

- Speed Layer - Kafka + Spark Streaming - real-time processing of the most recent data.
- Batch Layer - Hive + (partly) MongoDB - storing and processing all available data.
- Serving Layer - Spark + ML + (partly) MongoDB - combining output from the Speed and Batch Layer.

5 ML Model

The aim of the project is to forecast pollution based on weather conditions. For that we are going to use Apache Spark and MLlib. The data from the speed layer will be fed into Spark and a machine learning model will be trained on the joined weather and pollution data. We are planning to use a retrainable AI model to avoid having to query Hive for historical data every time the model is trained on new data. MLlib offers streaming ML models (like Random Forest) which should fit our use case.

6 Data presentation

The mock up for our data visualisation can be seen in the figure 2.

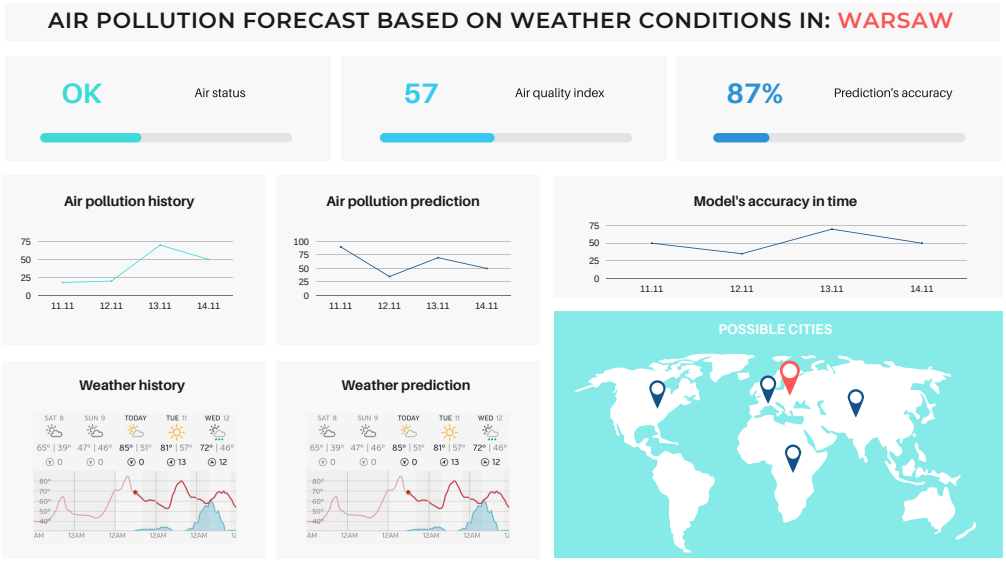

Figure 2: Example data visualisation

We are planning to show the following values and plots:

- current air status
- current air quality index
- overall model's accuracy
- plot of air pollution history
- plot of weather history
- plot of weather predictions (from the source)
- plot of air pollution prediction (ours)
- plot of model's history in time

The user will be choosing from the map of all possible cities, the one he wants to check the prediction for air pollution. The plots will change accordingly to the user's choice.

We are planning to aggregate by hour and store the following values in order to avoid counting them every time the UI refreshes:

- air temperature
- air pressure
- air humidity
- speed
- wind angle in degrees

7 Functional tests

7.1 Full system test

Test objective: having set up and enabled the whole system wait for the data to be processed and query the hive server to verify the data saved on HDFS.

Test steps:

1. Build and start the docker containers by running `docker-compose up -d` in the project root directory.
2. In a web browser go to `localhost:8443/nifi/` and log in using credentials present in `docker-compose.yml` file.
3. Upload and add `nifi-template.xml` template.
4. Enable all components.
5. Wait approximately 3 to 5 minutes for the data to be fetched, filtered, transformed and put to HDFS.
6. Open Hive Server command line interface by opening Docker Desktop and pressing the button shown in Figure 3.
7. Connect to Hive using the following command `beeline -u jdbc:hive2://`.
8. Verify the data by running HiveQL queries to all tables (e.g. `select * from pollutionwarsaw;`). Table names are in format `pollution<cityname>` and `weather<cityname>` (Figure 4).

Expected result: the data is present in the database
Actual result (pass) can be seen on figure 4

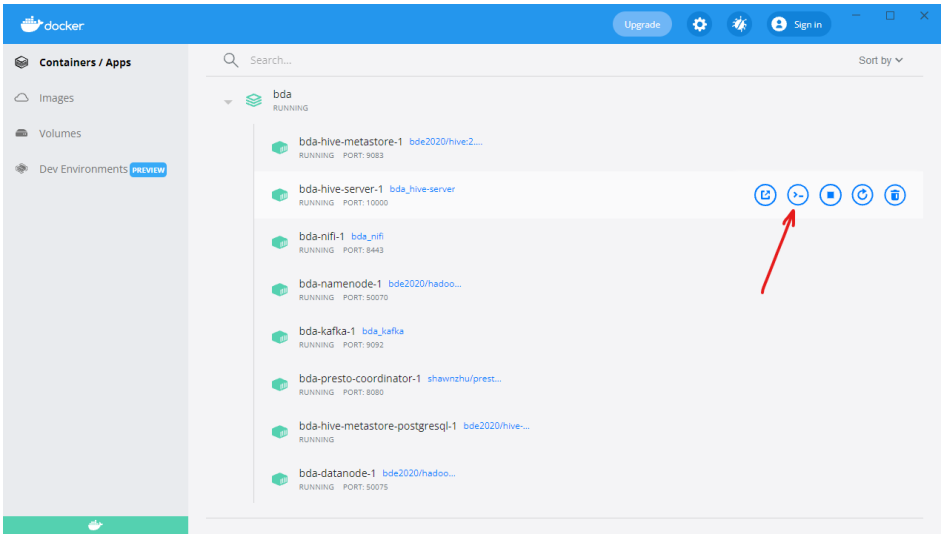

Figure 3: Means of accessing the Hive Server CLI

```
0: jdbc:hive2://> select * from pollutionwarsaw;
+--------------------------------------+-----------------------+-------------------------+-------------------------+-------------------------+-----------------------------+
|          pollutionwarsaw.id          |  pollutionwarsaw.ts   | pollutionwarsaw.aqi | pollutionwarsaw.lon | pollutionwarsaw.lat | pollutionwarsaw.measured |
+--------------------------------------+-----------------------+-------------------------+-------------------------+-------------------------+-----------------------------+
| a997935c-36c1-44e6-893c-2680d8f5eeee | 2021-11-26 11:51:49.837 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 6c0e5ac5-1dde-4585-a5df-b9223db6dbf7 | 2021-11-26 11:51:27.59  | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 54f257e7-0110-446a-bba9-9e10ae508cb1 | 2021-11-27 19:47:45.616 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 8ff24762-6848-408d-ab47-51aef6515ed5 | 2021-11-27 19:47:45.623 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 2f78de6b-e038-4583-a546-f2b3c8b30761 | 2021-11-27 19:47:45.73  | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 5f1244e2-c182-45ec-ab83-b9dafd6c87df | 2021-11-27 19:47:45.795 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| b87dd8fa-4efe-4a6e-a4f1-12c7f3bbd124 | 2021-11-27 19:47:45.835 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 68bef5fc-e3b8-4277-bf13-e61d08b0af36 | 2021-11-27 19:47:45.671 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 9757c838-de92-4e3e-a17d-8ffaee0bd963 | 2021-11-27 19:47:45.79  | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 35e04446-171c-44d8-b060-0ef35b40a916 | 2021-11-27 19:47:45.793 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 58d8ad5b-97da-4250-bcdf-0e0dcdf620bd | 2021-11-27 19:47:45.817 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 8f4621c8-0a44-4e05-9f37-de079b8367da | 2021-11-27 19:47:45.841 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| b4edb402-3fd5-4eaf-bc9b-cb4cb8c2ad4e | 2021-11-27 19:47:45.853 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 4751cc45-0c1e-428d-8cc4-fc1ddce5953c | 2021-11-27 19:47:45.882 | 74                  | 52.22517            | 21.014784           | 1638039600                  |
| d481250e-1798-4bbe-957a-72a8ffeaef7e | 2021-11-27 19:47:45.53  | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 2d169e00-f153-4c73-bd4b-2dae70caac88 | 2021-11-27 19:47:45.593 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| d234e47b-2dde-4ec0-8b88-fa50f453654d | 2021-11-27 19:47:45.604 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 56c70a20-63ed-40df-8dc3-9d23a0a8bd82 | 2021-11-27 19:47:45.607 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 3ab583c7-87d5-4f0d-ac5f-8300bf63a6c8 | 2021-11-27 19:47:45.72  | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 51036054-49e2-4478-a31e-f37c520ab1c1 | 2021-11-27 19:47:45.767 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| d317647d-0279-48ce-89a9-f3b1c5c87f5c | 2021-11-27 19:47:45.8   | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| bc5e9497-0493-4f8d-ac6b-d775570366e2 | 2021-11-27 19:47:45.82  | 87                  | 52.22517            | 21.014784           | 1637924400                  |
| 1ef8cc9e-f0c9-4c77-b541-13ea0f3009a2 | 2021-11-27 19:47:45.824 | 87                  | 52.22517            | 21.014784           | 1637924400                  |
+--------------------------------------+-----------------------+-------------------------+-------------------------+-------------------------+-----------------------------+
```

Figure 4: Example test output for pollutionwarsaw table

7.2 System failure email test

Test objective: simulate system failure and check whether an email is sent to notify the user about a failure.
Test steps:

1. Follow steps 1-4 from the test 7.1 (setting up the environment and running all NIFI components).
2. Simulate a system failure. For instance, open configuration window for any JoltTransformJSON processor (e.g. Pollution Data Aquisition / Warsaw / Transform Pollution JSON). Stop the processor and invalidate the Jolt Specification (Figure 5).
3. Log into the email account that gets notified about failures (weather.pollution.alert@gmail.com) and verify whether and email was received (Figure 6).

Expected result: the email is sent
Actual result (pass) can be seen on figure 4

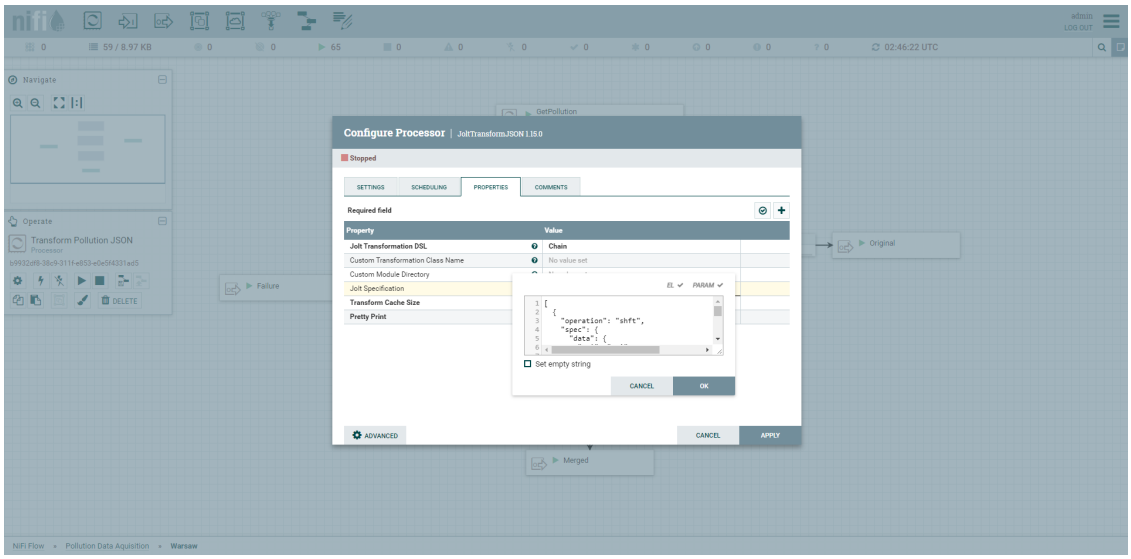

Figure 5: Invalid jolt specification

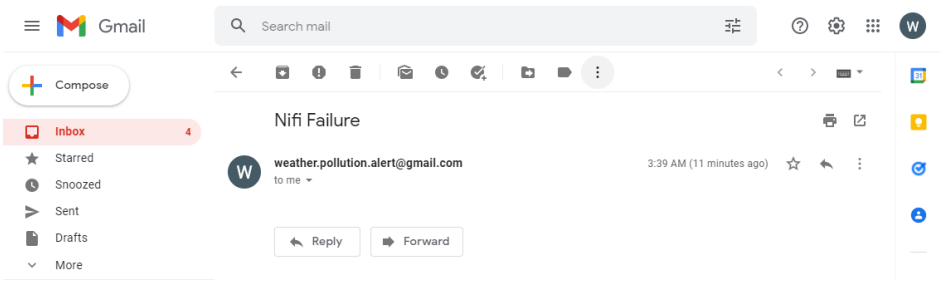

Figure 6: System failure email

7.3 Kafka queues test

Test objective: verify that Kafka producers are gathering data properly and without errors.
Test steps:

1. Follow steps 1-2 from the test 7.1 (setting up the environment).
2. Create `ConsumeKafka_2_6` processor
3. Create `LogAttribute` processor
4. Disable `LogAttribute` processor
5. Connect `ConsumeKafka_2_6` to `LogAttribute`
6. Configure `ConsumeKafka_2_6` processor as follows:
   (a) `Kafka Brokers`: `kafka:9092`
   (b) `Topic name(s)`: any of the available topic e.g. `pollutionwarsaw`
   (c) `Group ID`: `grid1`
   (d) `Offset Reset`: `earliest`
7. Run `ConsumeKafka_2_6` processor and wait 20 seconds
8. Verify that messages are queueing up to `LogAttribute` processor
9. Copy and paste `ConsumeKafka_2_6` processor
10. Configure new `ConsumeKafka_2_6` processor as follows:
    (a) `Topic name(s)`: error
11. Connect new `ConsumeKafka_2_6` to `LogAttribute`
12. Run new `ConsumeKafka_2_6` processor and wait 20 seconds
13. Verify that messages are not queueing up to `LogAttribute` processor from new `ConsumeKafka_2_6` (Figure 7)

Expected result: messages are queued up without errors
Actual result (pass) can be seen on figure 7

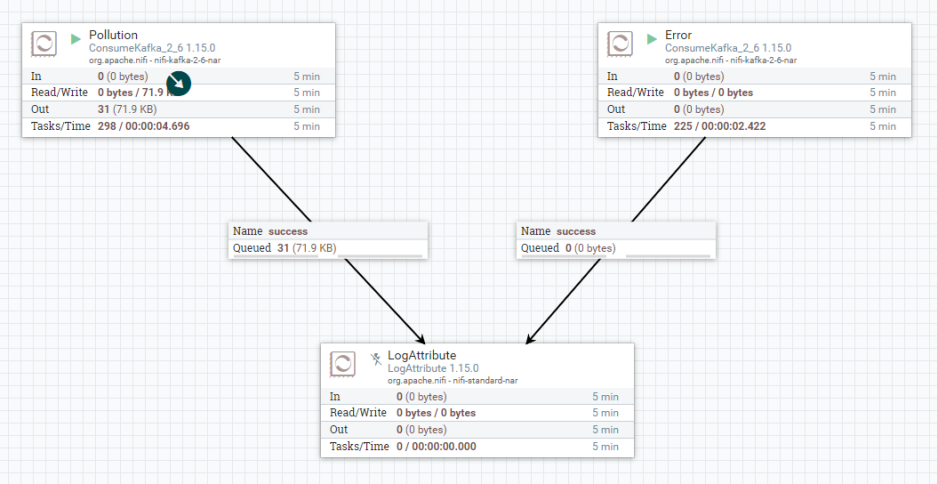

Figure 7: Kafka queues test test result

7.4 Data persistency test

Test objective: verify that the data is stored in a non-volatile memory.
Test steps:

1. Follow all steps from 7.1
2. Stop the environment by running the command `docker-compose stop`
3. Follow steps 1,5-8 from 7.1

Expected result: messages are queued up without errors
Actual result (pass) is identical to 4