Predicting tram delays based on weather

The Transaltlantic Scooters' Project Documentation

Laura Bakała
Wojciech Kretowicz
Karol Pysiak
Mateusz Szysz

16 January 2022

Contents

1 Overview

The project is focused on predicting Warsaw tram delays based on weather in a (almost) real-time. It may provide numerous advantages for both individual users and municipal administration. From the point of view of the former, it allows to predict the delays of trams which facilitates choosing the most appropriate mean of transport in a given moment. For example, when you have to move to another district of Warsaw, you can decide if it's worth to choose tram or if it's better to use your own car. Similarly, you can better plan an appropriate time of going out when you have no choice and need to use tram. On the other hand, our solution allows to store and analyze in details the historic data concerning delays. Careful analysis of the data will allow to find to some regularities in delays of some lines which may be a starting point of rescheduling some trams and improving the public transport system in Warsaw.
In the next sections we describe the data sources, data processing plan including technologies used and all the transformation that we make, architectural plans, the way of modelling the delays and initial results of our analysis concerning the explanatory data analysis.
The project was done in Google Cloud.

2 Data sources

2.1 Warsaw Open Data - online information about trams and buses

The primary data source for the project is Warsaw Open Data, an open database containing much information about Warsaw, *inter alia* public transport. It describes the position of trams and buses in real-time. This data can be found here: Otwarte dane po warszawsku.

The data can be accessed by a RESTFUL API with a dedicated endpoint that returns on-line data about trams and buses. This endpoint returns a JSON file, where each record consists of following fields:

1. Lines – a line number of a given tram or a bus
2. Lon – a longitude of the position of the vehicle
3. Lat – a latitude of the position of the vehicle
4. VehicleNumer – an identification number of the vehicle
5. Brigade – an identificator of a team that operates the vehicle at given time
6. Time – a timestamp created at the moment of reading GPS data

The data is updated with the declared frequency of 1 minute. However, in practice, we observed that the data flows with the frequency of 10 to 15 seconds. Single call to the API returns a variable number of records, up to about 300 during rush hours. Thus this data is treated by us as a stream data.

2.2 Warsaw Open Data - public transportation schedule

Moreover, we use Warsaw Open Data, available under the same link as in case of the online information about public transport (Otwarte dane po warszawsku), to obtain the schedule. This is once again a RESTFUL API. However, in this case the updates of the data are made in a much smaller frequency (schedule changes relatively rarely).

We utilize three different endpoints to collect all necessary data. All of them return a JSON file, where each record is described as a list of objects with `"key"` and `"value"` fields.

Each endpoint has a limited number of `"key"` values that can appear. These are as described below,

The endpoint with the list of bus/tram stops returns about 7000 records. The fields are as follows:

1. zespol – ID number of the complex of bus/tram stops (bus/tram stops are grouped together under one name when they are placed in the same location, usually on different sides of a road junction)
2. slupek – ID number of the exact bus/tram stop, unique for given stop complex
3. nazwa_zespolu – human-friendly name of a stop complex (e.g. "Morskie Oko")
4. id_ulicy – ID number of the street the stop is placed on
5. szer_geo – latitude
6. dlug_geo – longitude
7. kierunek – direction the stop is facing
8. obowiazuje_od – date when the data was last changed

The endpoint with the list of bus/tram lines for given bus/tram stop returns between 0 and 20 records. It requires `zespol` and `slupek` values from the previous call. The only field appearing is `linia`, giving the line number.

The endpoint with the timetable for given bus/tram stop and line number returns up to a hundred records. It requires `zespol`, `slupek`, and `linia` values from the previous calls. The fields are as follows:

1. czas – hour of departure (i.e. hour, minute and second, but the last value is always equal to 0)
2. trasa – ID code of the travel (understood as a sequence of stops)
3. kierunek – direction of the travel
4. brygada – ID number of the team operating the vehicle
5. symbol_1, symbol_2 – additional attributes, usually empty

If it wasn't for the fact that we filter out bus stops, the volume of the data (i.e. the number of timetable entries) would be as high as a few million records. However, with filtering included, the volume of the fully processed data is about 110 000 records.

2.3 Meteorological data

The third and last data source is open meteorological data source from which we can download prognoses for the specific amount of days as well as the historical data. This is a RESTFUL API available here: Weather API.

Single call to this API returns a JSON file with following fields:

1. cod - Internal parameter
2. message - Internal parameter
3. cnt - A number of timestamps returned in the API response
4. list - List of weather predictions
    - dt - Time of data forecasted, unix, UTC
    - main
        - temp - Temperature. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
        - feels_like - This temperature parameter accounts for the human perception of weather. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
        - temp_min - Minimum temperature at the moment of calculation. This is minimal forecasted temperature (within large megalopolises and urban areas), use this parameter optionally. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
        - temp_max - Maximum temperature at the moment of calculation. This is maximal forecasted temperature (within large megalopolises and urban areas), use this parameter optionally. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
        - pressure - Atmospheric pressure on the sea level by default, hPa
        - sea_level - Atmospheric pressure on the sea level, hPa
        - grnd_level - Atmospheric pressure on the ground level, hPa
        - humidity - Humidity, %
        - temp_kf - Internal parameter
    - weather
        - id - Weather condition id
        - main - Group of weather parameters (Rain, Snow, Extreme etc.)
        - description - Weather condition within the group. You can get the output in your language.
        - icon - Weather icon id
    - clouds
        - all - Cloudiness, %
    - wind

        - speed - Wind speed. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour.
        - deg - Wind direction, degrees (meteorological)
        - gust - Wind gust. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour
    - visibility - Average visibility, metres
    - pop - Probability of precipitation
    - rain
        - 3h - Rain volume for last 3 hours, mm
    - snow
        - 3h - Snow volume for last 3 hours
    - sys
        - pod - Part of the day (n - night, d - day)
    - dt_txt - Time of data forecasted, ISO, UTC
5. city
    - id - City ID
    - name - City name
    - coord
        - lat - City geo location, latitude
        - lon - City geo location, longitude
    - country - Country code (GB, JP etc.)
    - timezone - Shift in seconds from UTC

This data is updated with the frequency of 3 hours. Single call returns 40 weather predictions. Thus it is treated by us as a batch data.

We are using the free pricing plan provided by the website. This gives us an information about the current weather conditions, a minute forecast for 1 hour, hourly forecast for 2 days, daily forecast for 7 days and historical weather for 5 days. This API is up 95% of time. We are allowed to call this API 1000 times a day or 60 times a minute.

3 Architecture

The general schema of our architecture is shown in fig. 1. The data is ingested from three sources described in the previous section. Timetable and weather data should be considered as batch sources, while trams positions as a stream source. After ingestion the data is processed with NiFi. This process is described in details in the next section.

After being processed, the data may be sent to three destinations. The first one is HBase on Hadoop Cluster where the whole data is stored without any updates. This

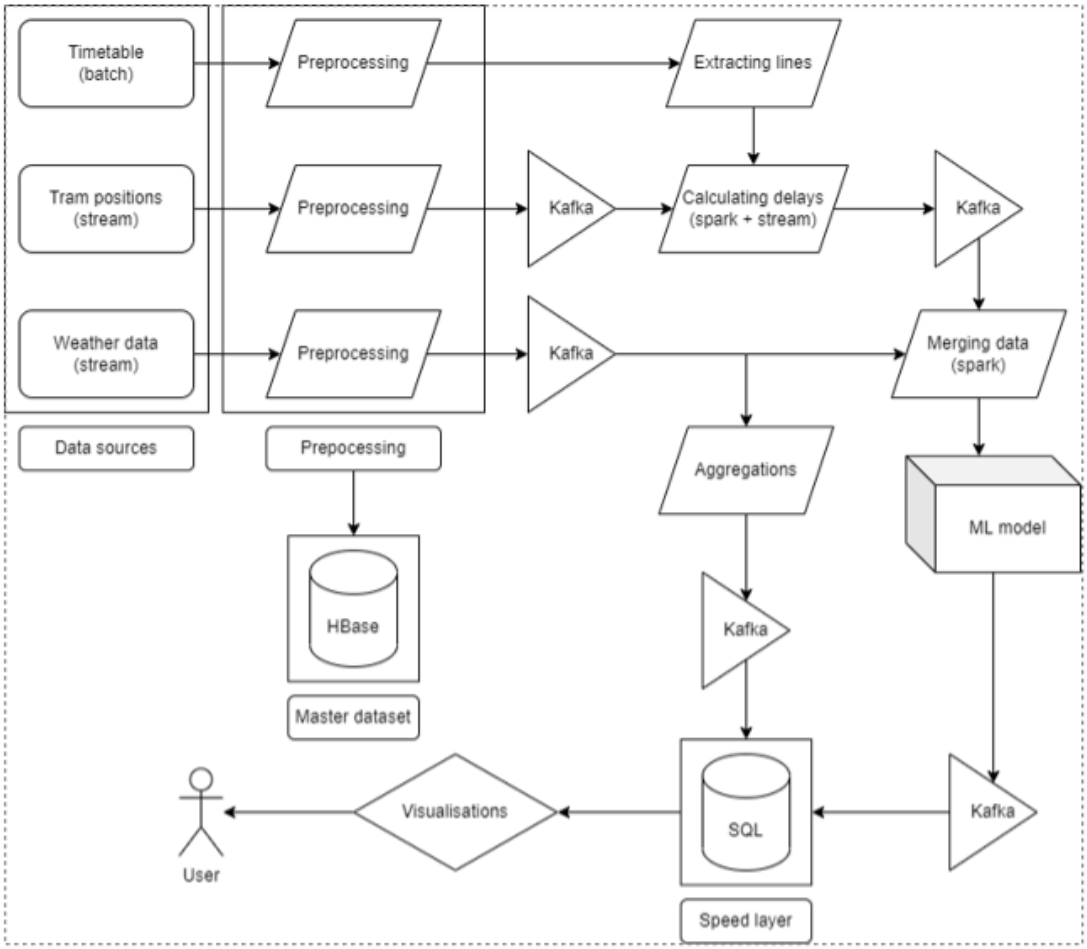

Figure 1: Architecture schema

is useful in case of failures in the further parts of the system. The second destination is machine learning layer where the models are estimated. Between processing and ML parts we extract lines and calculate delays. The last destination is speed layer in which we can query weather data and where the results of the model are stored. More detailed descriptions of ML part and serving layer are included in the next sections. The data in the speed layer is stored Google Cloud's MySQL. It's easy to connect to this service from different BI services available in Google Cloud and other external tools in which you can easily visualize the data. Examples of visualizations are included in the speed layer section.

4 Data processing plan

The whole data processing pipeline is divided into three parts described separately. Please, note, that a failure at any step is reported by an email. Also, after presenting the previous milestone, we added Kafkas that are used in a real-time streaming pipelines.

4.1 Warsaw Open Data - online information about trams and buses

Warszawskie Dane offers access to the data in the streaming fashion. Thus, a whole pipeline of data transformation is processed as a stream.

Processors used in the data transformation are listed below:

1. InvokeHTTP – calls an API every 10 seconds in order to obtain records about trams and buses
   - P ublish_Kafka_2_6 – creates a topic for further streaming
2. FilterAndSplitJSON – filters incorrect records and splits them into separate JSON files, position that is outside of Warsaw is considered as incorrect record
3. EvaluateJSONPath – takes out required fields from each record
4. Publish_Kafka_2_6 – creates a topic for further streaming
   - P utHBaseJSON – puts each record into HBase

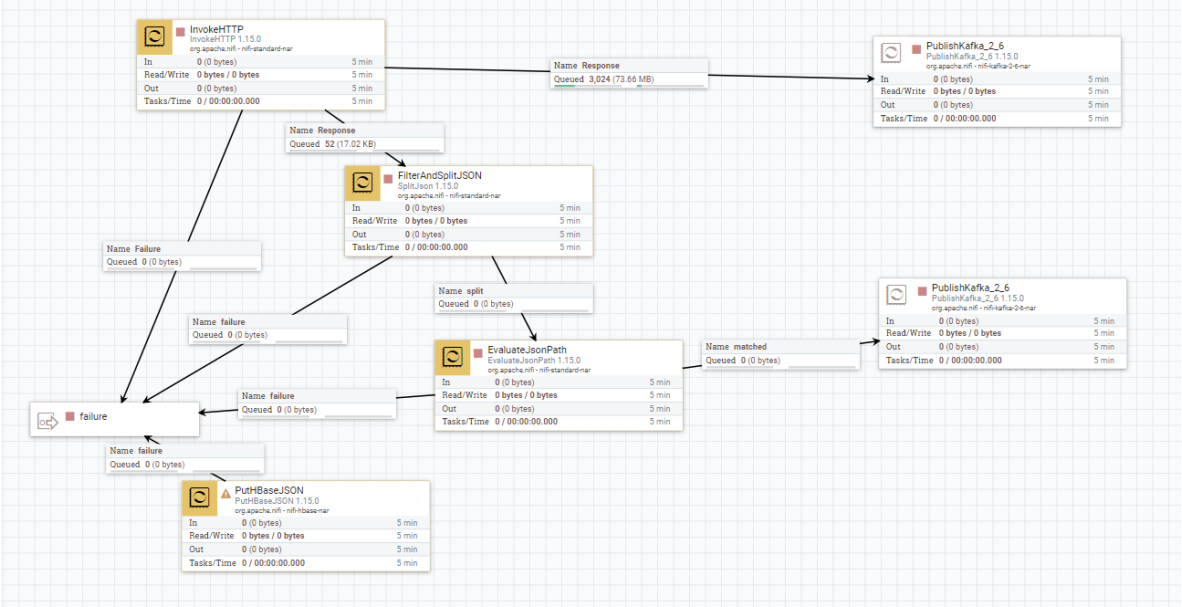

Figure 2: Tram positions NiFi schema.

This is visualized in fig. 2.

4.2 Public transport schedule

Processors used in this part of data processing have following tasks:

1. GenerateFlowFile – saves an API key to an attribute for later API calls
2. InvokeHTTP – calls an API once a day in order to obtain a list of bus and tram stops

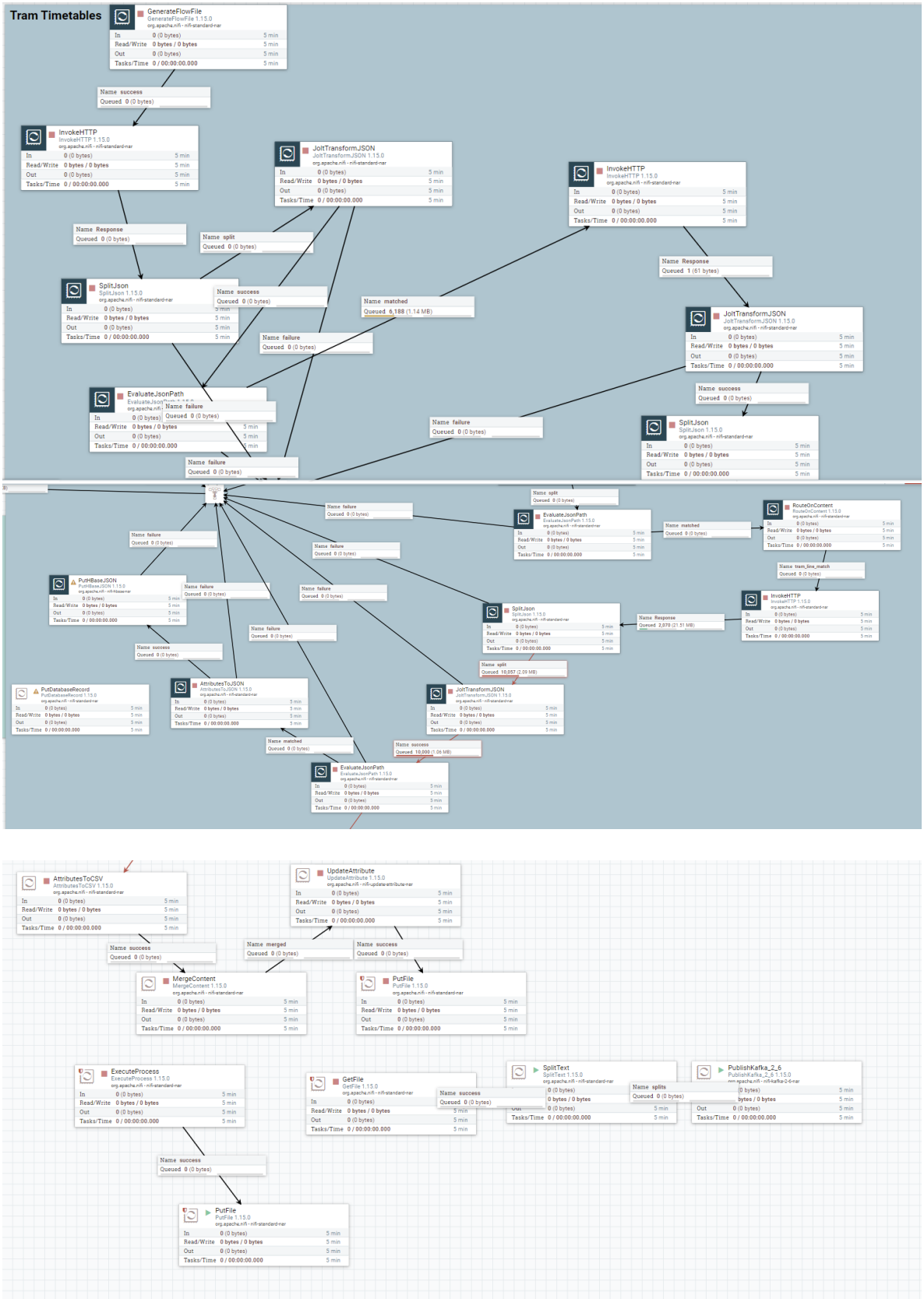

Figure 3: Tram timetables NiFi schema.

3. SplitJson – splits the list into separate stops
4. JoltTransformJSON – rearranges stop data into `"key": "value"` form
5. EvaluateJsonPath – saves a majority of information into FlowFile attributes: stop id, stop number, stop name, latitude, longitude, and direction
6. InvokeHTTP – calls an API for each bus/tram stop to obtain a list of bus/tram lines that stop there
7. JoltTransformJSON – rearranges data into a list of {`"linia": "XXX"`} objects
8. SplitJson – splits the list into separate FlowFiles for each bus/tram line
9. EvaluateJsonPath – extracts line as a FlowFile attribute
10. RouteOnContent – filters tram lines, understood as one- or two-digit numbers; bus lines are usually three-digit numbers or contain letters like "N" for night lines
11. InvokeHTTP – calls an API for each tram stop and line to retrieve a timetable (a list of departure times)
12. SplitJson – splits the list into separate timetable entries
13. JoltTransformJSON – rearranges stop data into `"key": "value"` form
14. EvaluateJsonPath – extracts time as a FlowFile attribute

    - A ttributesToJSON – saves data extracted as FlowFile attributes into FlowFile content in JSON format

    - P utHBaseJSON – puts each record into HBase

Next processors are AttributesToCSV, MergeContent, UpdateAttributte and PutFile. The role of this part is to save the timetable in a CSV file. The whole pipeline is run once a day at night. After that, the Execute Process is invoked. This is a Python script which creates all the lines from the timetable which are saved in a CSV file. SplitText divides this files into separate rows which are further conveyed through Kafka to PySpark in which we calculate the delays.

This is visualized in fig. 3.

4.3 Weather data

Processors used in this pipeline have a following tasks to do:

1. GetHTTP – calls an API every 3 hours in order to download a JSON about the weather with multiple records, each call is sent just one minute after the scheduled update in the API

    - P ublishKafka_2_6

2. SplitJSON – splits a received JSON into multiple records

3. EvaluateJSONPath – takes out from each record needed fields
4. FlattenJSON – simplifies a structure of received JSON to make it more adequate for HBase
   - P utHBaseJSON – puts a single record into HBase
5. PublishKafka_2_6 –

This is visualized on a figure 4.

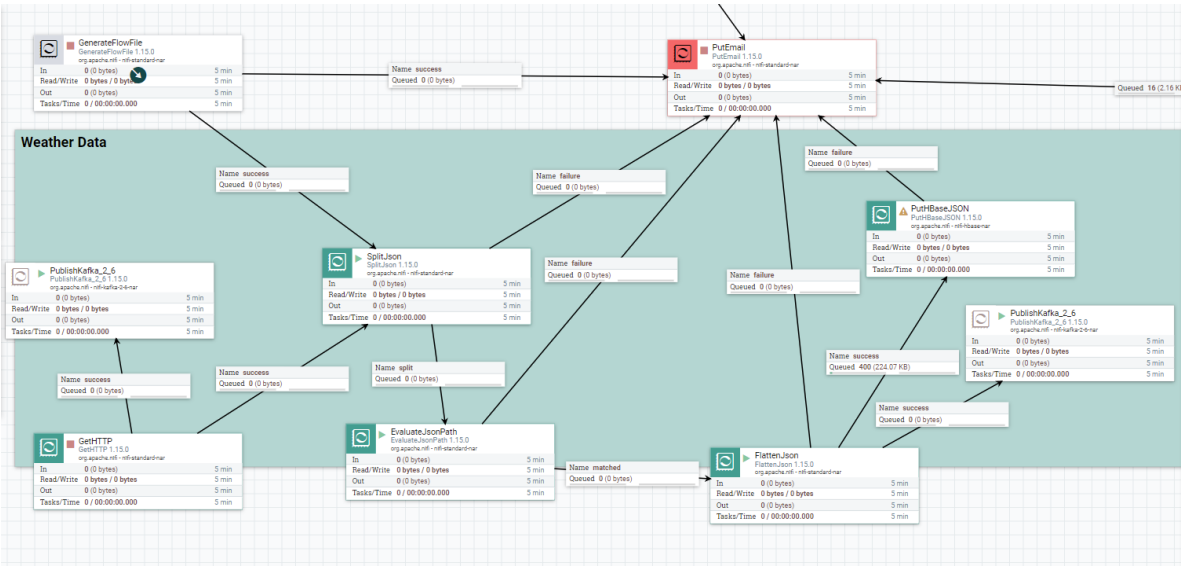

Figure 4: Weather processing NiFi schema.

4.4 Calculating discrepancies between scheduled and actual arrival

One of the most challenging part of the project is calculating delays. To do so, firstly we had to extract lines from timetable which was done as follows:

1. sorting all entries by "time"
2. grouping all entries by "line", "busstopId", and "busstopNr"
3. numbering consecutive entries for each group

Then entries across different groups with the same number constitutes a single line in a single direction preserving their order. After that we can calculate delays which is done as follows:

1. for each message about tram we find 4 closest stops (they are grouped in two bus stop groups each with two actual stops)
2. for these 4 stops we calculate time discrepancies and we find iteration that achieves minimum absolute time discrepancy
3. for these 4 stops within best iteration we select 2 stops with minimum absolute time discrepancy and we calculate an average of delays

4.5 Architecture solution for delays approximation

- Delays has to be calculated in a continuous manner, as part of a stream
- Thus, we use Spark to enhance the tram position stream
- Spark pulls timetable data from kafka topic once a day in order to have the most timely data
- Spark pulls stream of tram positions from kafka topic
- Both objects stream and static dataframe are then processed together and a new stream with delays is being pushed to a new kafka topic
- Data transformation currently implemented in Python and batch processing, stream processing is still being developed

4.6 Architecture solution for delays model

- Model will be updated once a day
- Predictions will be made in the streaming fashion, when weather data arrives
- Every delay prediction will be streamed to the speed layer
- Possibly, we will base the model not only on weather data, but on trams data either. Still, predictions will be in the streaming fashion

4.7 Architectural solution for additional weather data

- When we receive the weather forecasts we push the data to the speed layer for users to look up
- We filter mainly unimportant data
- Everything is done in the streaming fashion
- Transformed data is stored in the sql database ready to analyse

5 Explanatory analysis

In this section, we present the results of the explanatory data analysis based on the data in SQL. Currently, we haven't finished the part with calculating delays and estimating ML model. Therefore we don't have any data concerning trams and their delays. However we can show results of explanatory analysis of the weather data. As an example we show weather prognoses from January 10. In figures 5, 6, 7 we showed different elements of the prognoses. The x axis shows days, y axis values of given measures and number close to the circles are hours for which prognoses are given.

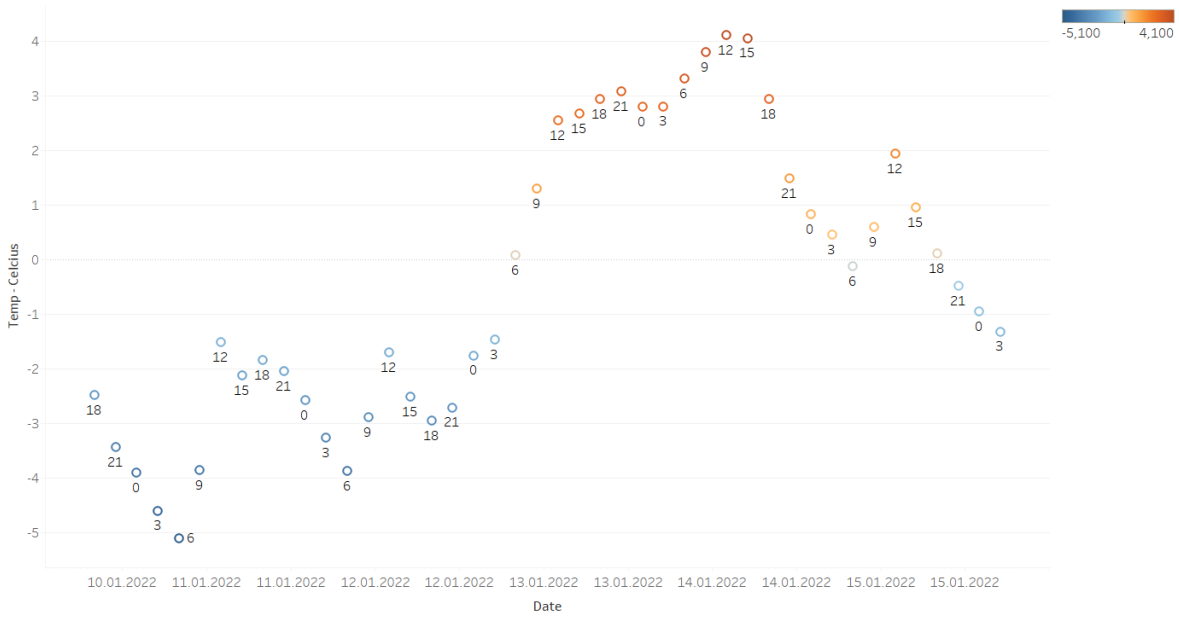

Figure 5: Temperature prognoses

6 Tests

All the tests are described in a separate file.

7 Future work

For the final milestone, we will finish making predictive models. Then we will be able to calculate different metrics of our predictions and evaluate its quality. Also, we will save all the results in SQL which will enable us to analyze historical delays and show current predictions on real-time dashboards.

8 Task allocation

Note: All the team members worked on a general idea of how the project should look like, which technologies should be used and what further steps should be taken at each step of the project development.

- **Karol**: weather data gathering, creation of predictive model, spark configuration
- **Laura**: tram data gathering (timetable and tram positions)
- **Mateusz**: report development, testing, visualizations, dashboards
- **Wojciech**: extracting lines from tram data, calculating delays, managing kafka topics, nifi configuration

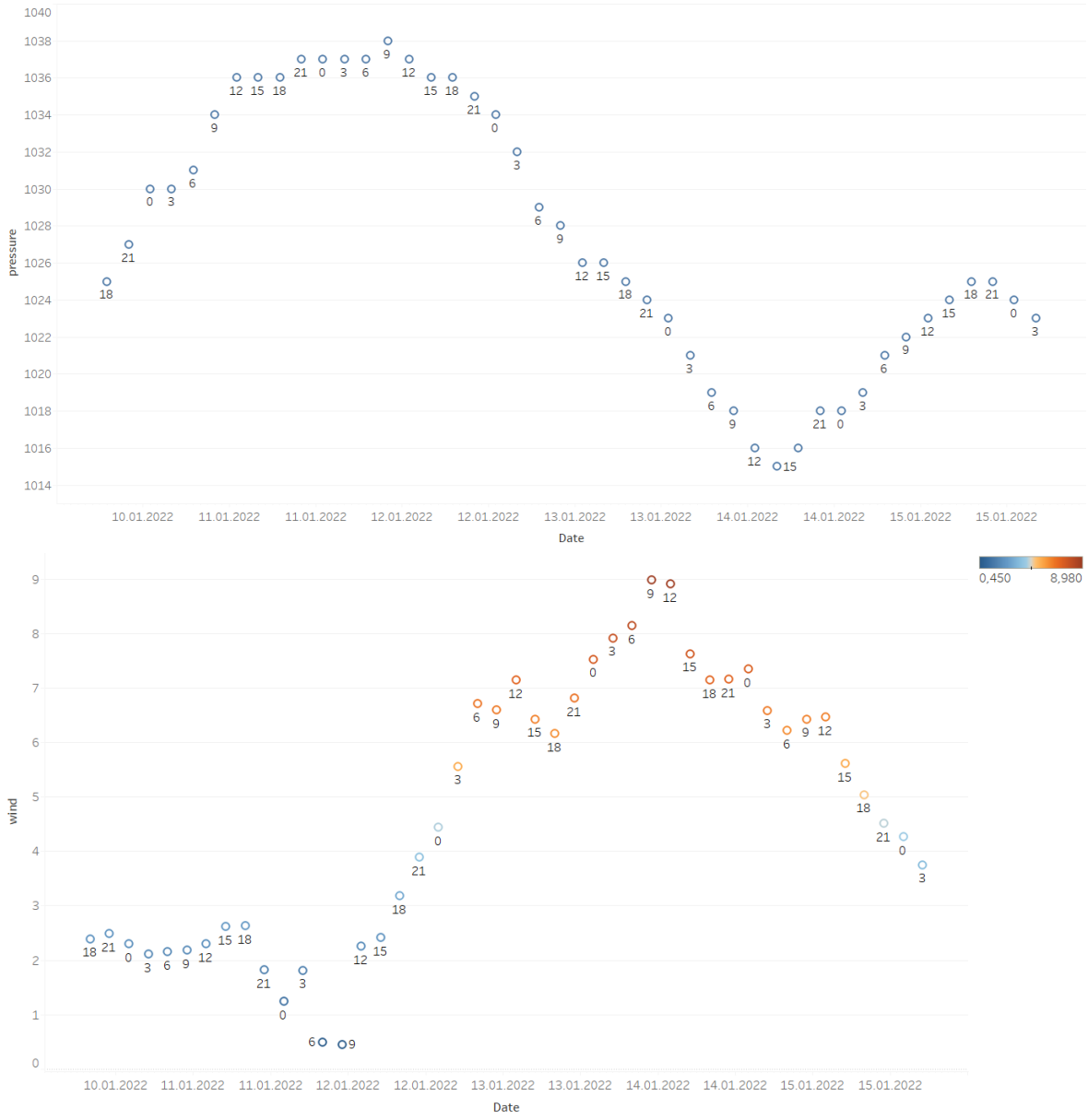

Figure 6: Air pressure and wind prognoses.

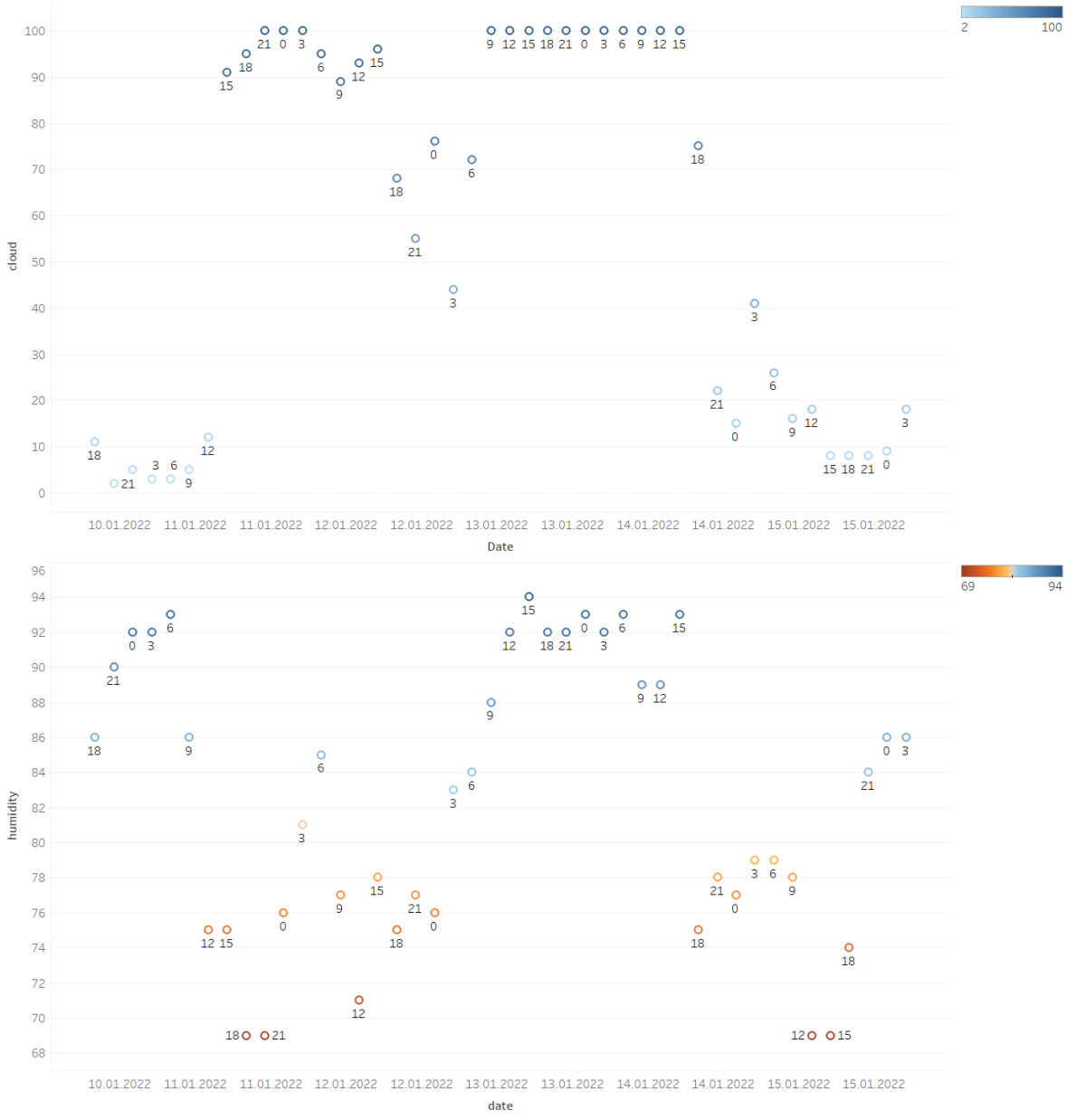

Figure 7: Clouds and humidity prognoses.