Smart City App

Contributors:

Manousos Linardakis , it22064
Christos Kazakos , it22033

Project Description:

Our project is a smart city app that provides users with information about cities worldwide. It offers rankings of the cities in areas like education, housing, and more. Furthermore, the app delivers weather and environmental forecasts for the upcoming 7 days, accompanied by images showcasing the cities and their precise locations on an interactive map. To gather this data, we utilize multiple APIs (see below), which efficiently store the information in a dedicated database. For a visual representation of the database structure, please refer to the ER diagram available here. To present our findings and insights, we rely on the power of **Node-RED** and its versatile dashboard components.

In addition, we seamlessly retrieve real-time weather data from four distinct weather APIs, which we also leveraged during the subflow challenge of the **2023 hackathon**. For more detailed information about the current weather APIs, please refer to the dedicated section in this document. We have also developed solutions for the remaining challenges of hackathon 2023, which are presented in the subsequent sections of this document.

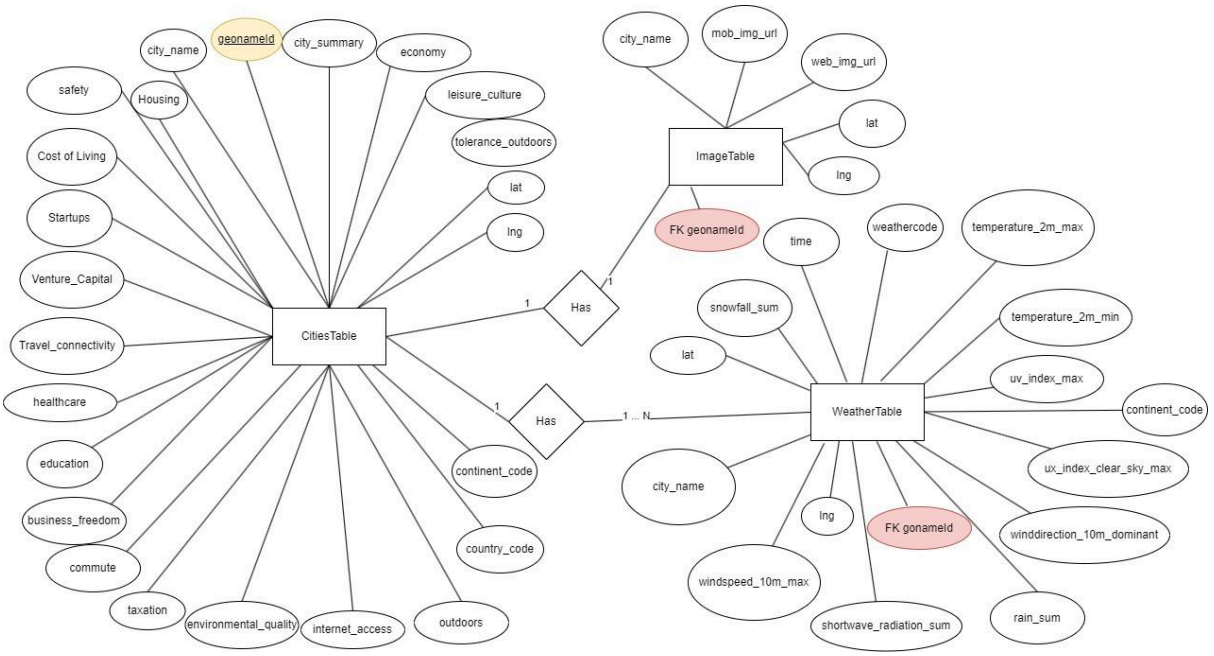
Contents:

APIs Used:

Description	API	Input	Output
Used to get the cities of a requested country.	https://www.geonames.org/	country code, geonames_username	cities (of the country code)
The API endpoint forecast accepts a geographical coordinate, a list of weather variables and responds with a JSON weather forecast for 7 days.	https://open-meteo.com/	geographical coordinate (lat, lon), time interval (hourly, daily, …), list of weather variables that they are interested in (see this link)	temperature, weather (in general like rain, wind, humidity, soil, etc)
Gets the quality of life of a requested city.	https://developers.teleport.org/api/getting_started/#life_quality_ua	city_name	Quality of life (education, environmental quality, etc – Life Quality Data for Cities)
Gets a photo of a requested city.	https://developers.teleport.org/api/getting_started/#photos_ua	city_name	Photos of cities (City Photos)
OpenweatherMap API	https://openweathermap.org/current	api key, latitude, longitude	Current weather data for a city

After fetching data from these APIs, we store the information of cities in our database. Please find below the Entity-Relationship (ER) diagram that illustrates the data schema for our database:

ER Diagram:



For a better view of the ER diagram, please refer to this link.

Messaging:

For messaging in Node-RED, we utilize the RabbitMQ node, which allows us to establish communication with a RabbitMQ message broker. This messaging system plays an important role in our application as we retrieve real-time weather data from the OpenWeatherMap API for the selected city and update it every 50 seconds. Specifically, Node-RED sends a request to the OpenWeatherMap API, fetches the latest weather data for the specified city, and publishes it as a message via the RabbitMQ node. This message is then received by the flows that have subscribed to the RabbitMQ exchange, allowing them to process and display the updated weather information in real time.

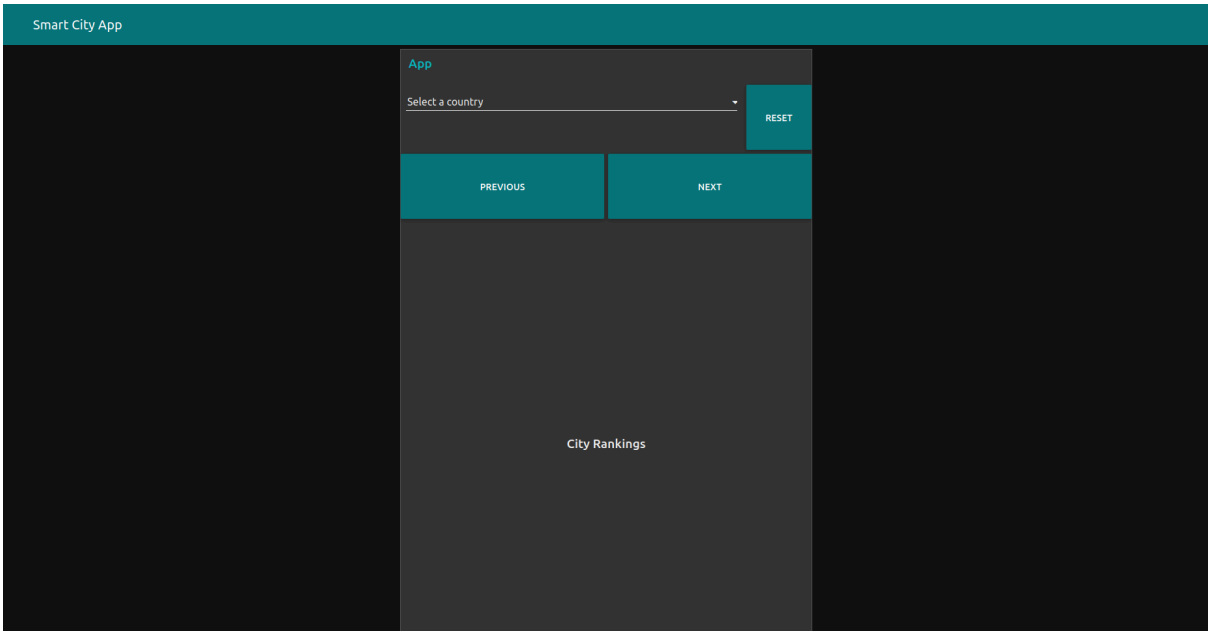
By utilizing RabbitMQ and the messaging capabilities of Node-RED, we establish a seamless and event-driven communication flow, ensuring that the weather data is consistently updated and made available to the relevant components of our application.

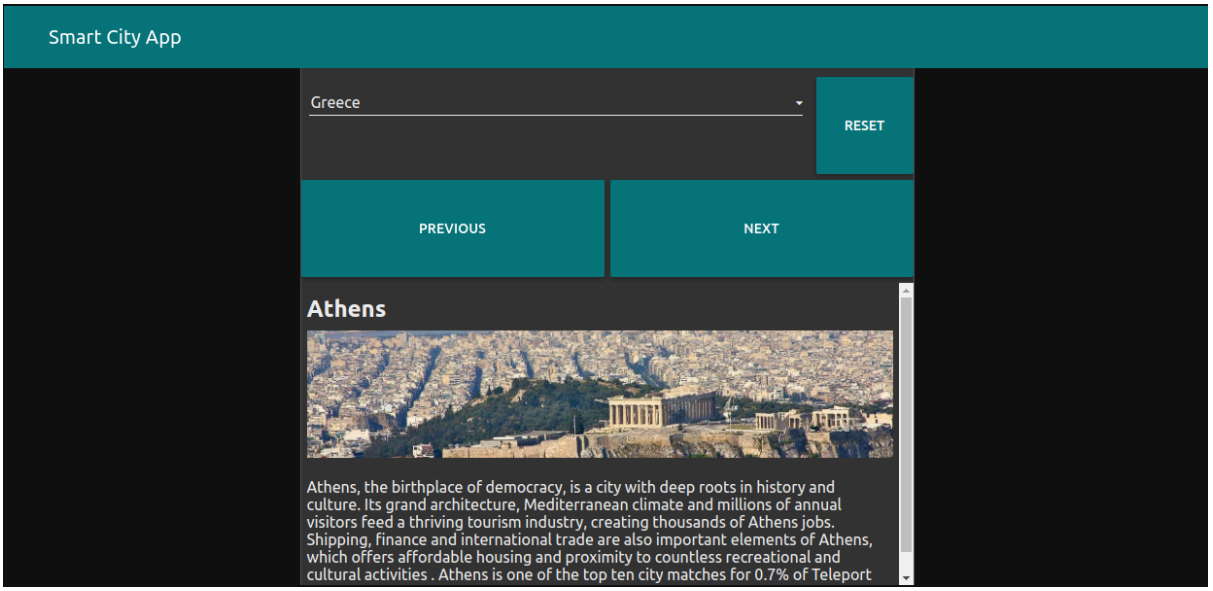
K-means Clustering:

We also implemented a k-means function, to group the centers of each city temperature.Then each city goes in a group of temperatures (the groups are high, middle, low).

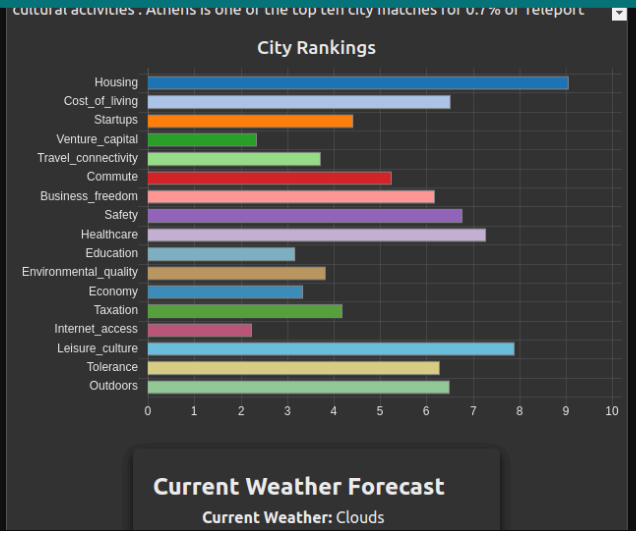
Example Output in GUI:

GUI Showcase:





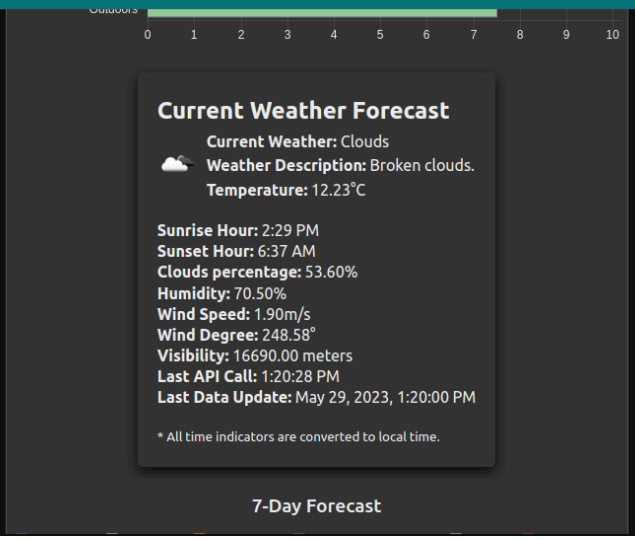
Smart City App



Current Weather Forecast

Current Weather: Clouds

Smart City App



Current Weather Forecast

Current Weather: Clouds
Weather Description: Broken clouds.
Temperature: 12.23°C

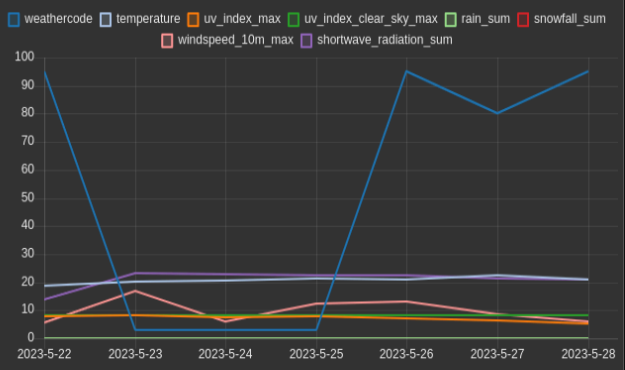
Sunrise Hour: 2:29 PM
Sunset Hour: 6:37 AM
Clouds percentage: 53.60%
Humidity: 70.50%
Wind Speed: 1.90m/s
Wind Degree: 248.58°
Visibility: 16690.00 meters
Last API Call: 1:20:28 PM
Last Data Update: May 29, 2023, 1:20:00 PM

* All time indicators are converted to local time.

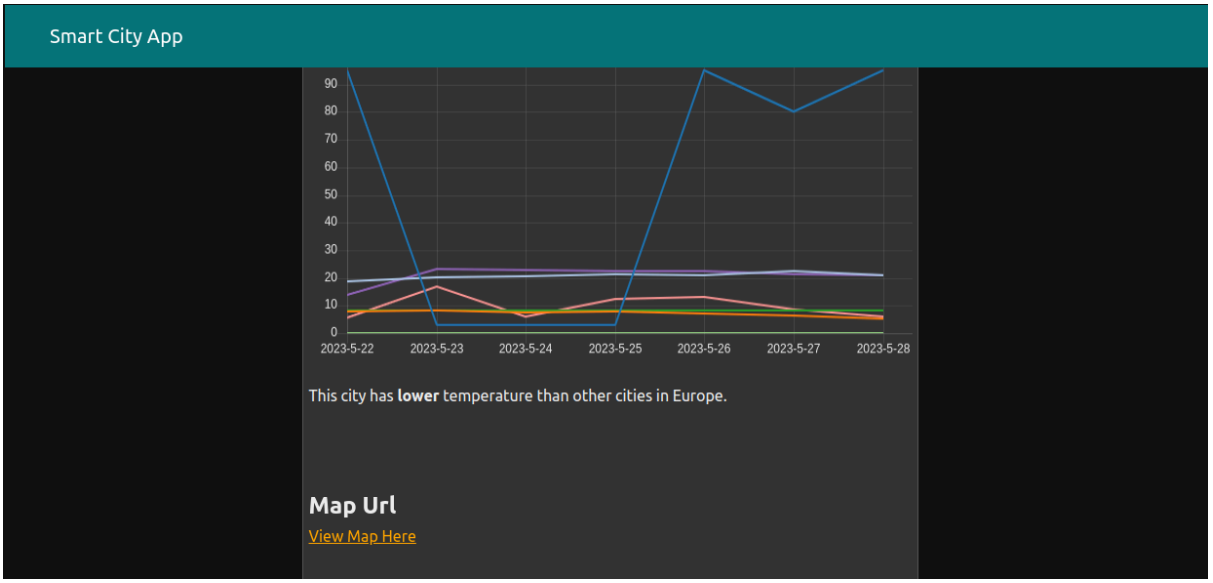
7-Day Forecast

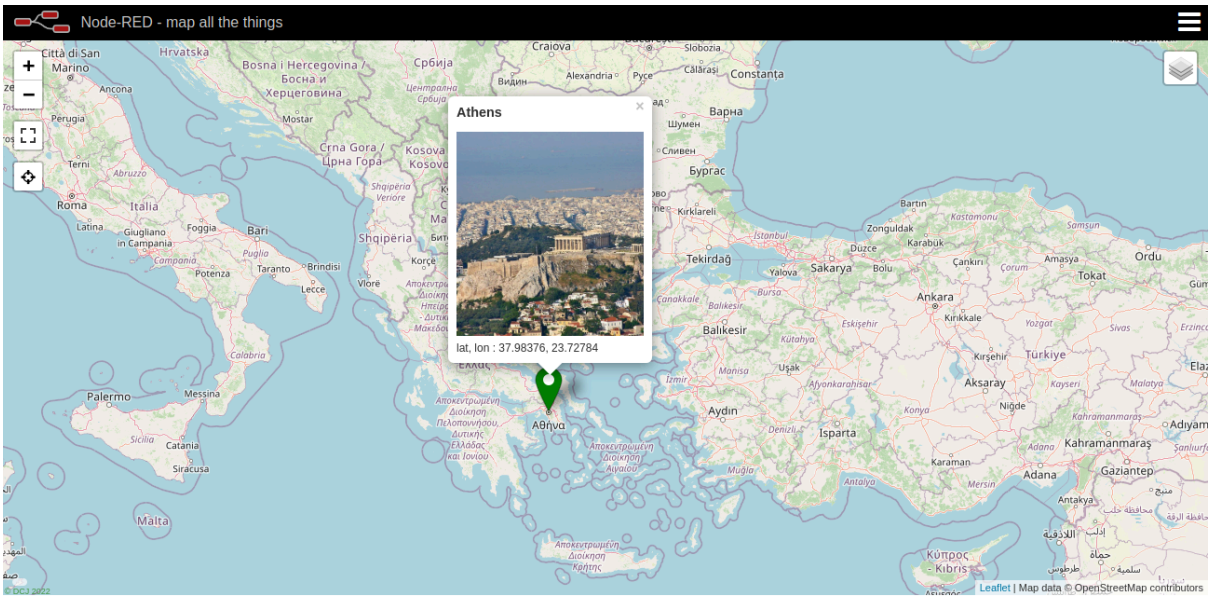
Smart City App

7-Day Forecast



This city has **lower** temperature than other cities in Europe.

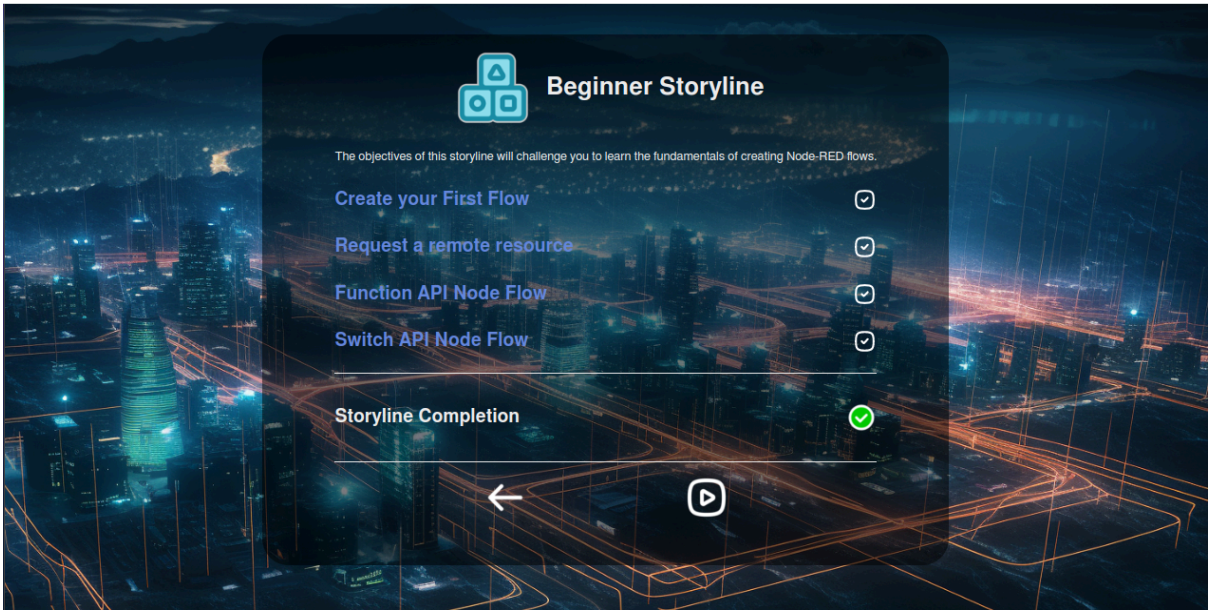


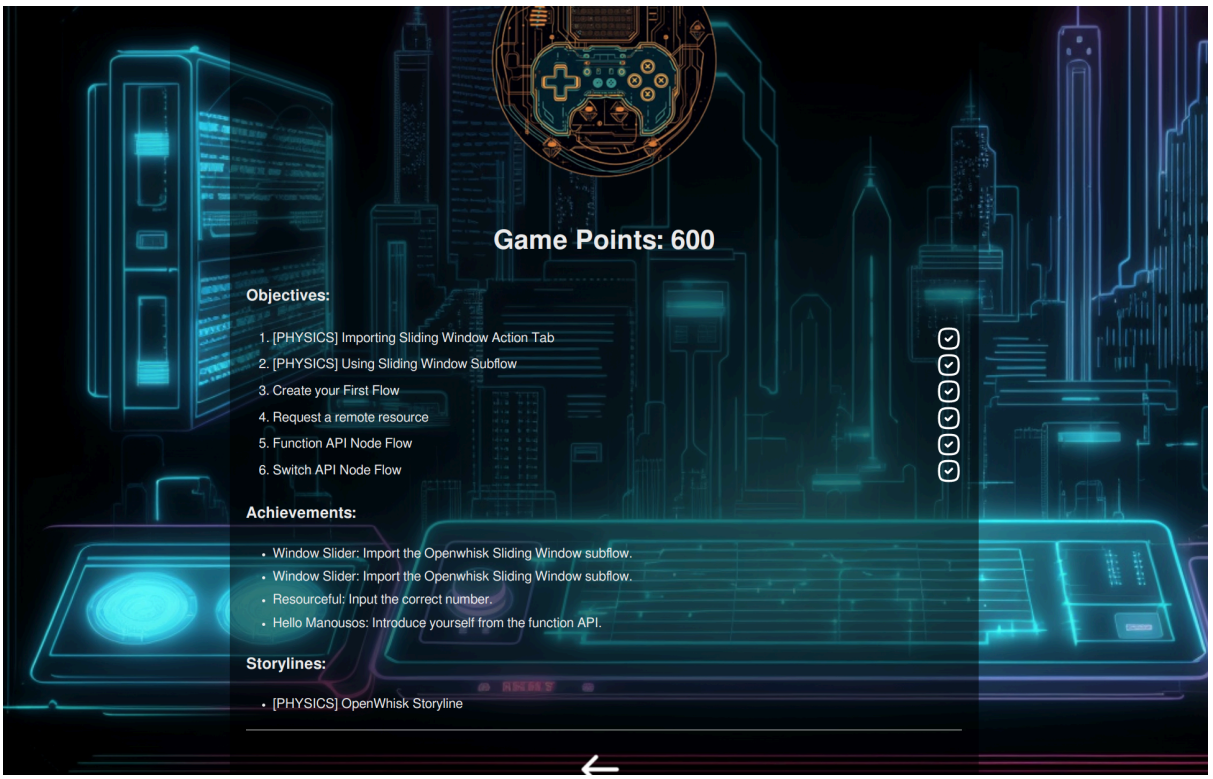


** The screenshots are taken after a few days from each other.

Hackathon

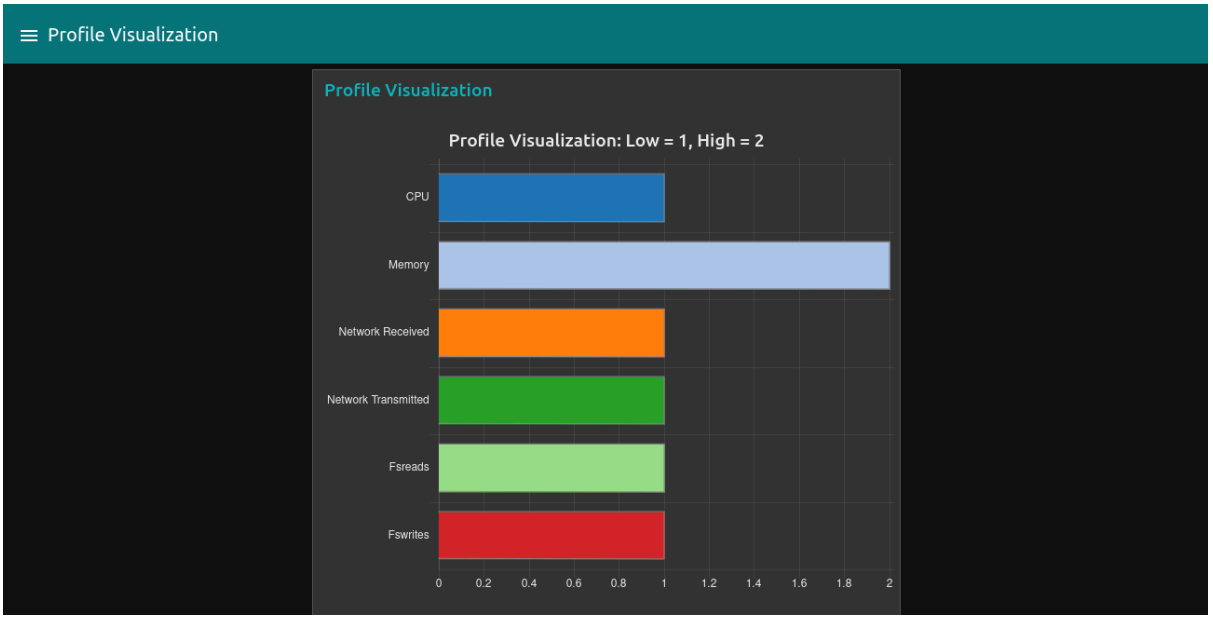
Gamification Server Challenges:





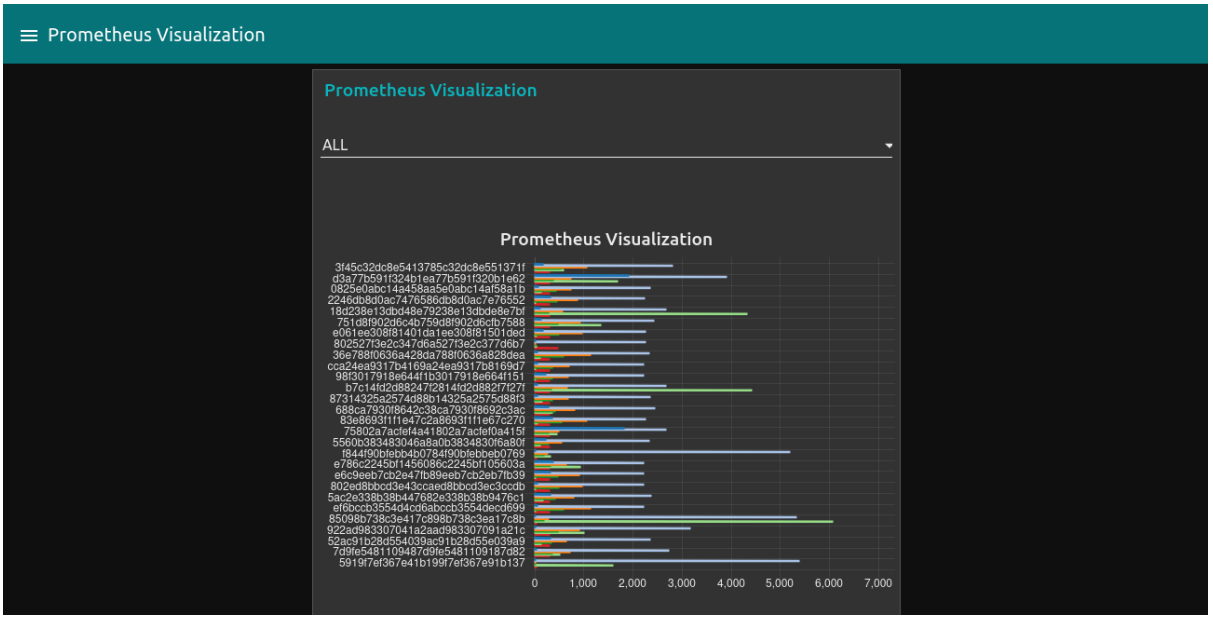
Visualization Challenge:

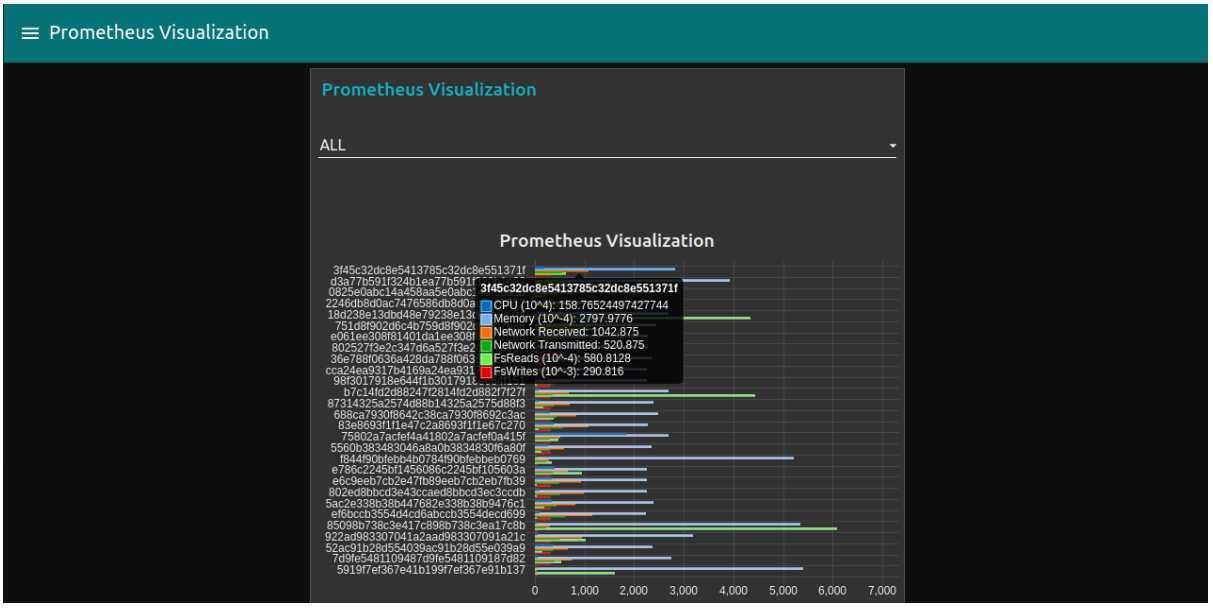
Profile-data Visualization:



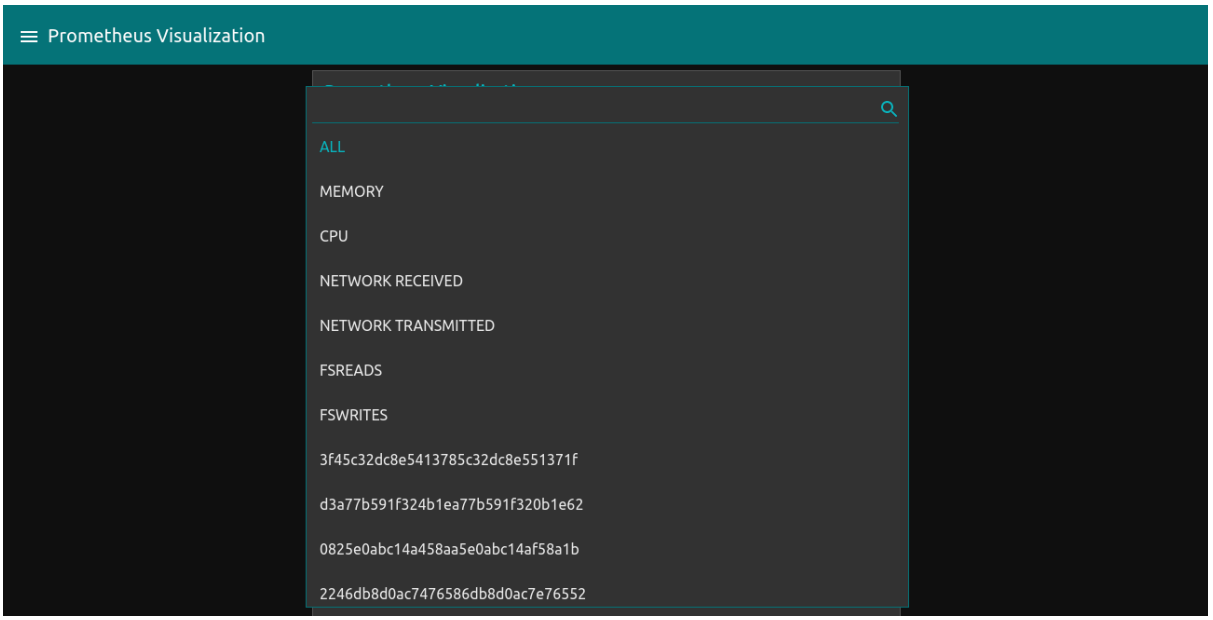
Prometheus-data Visualization:

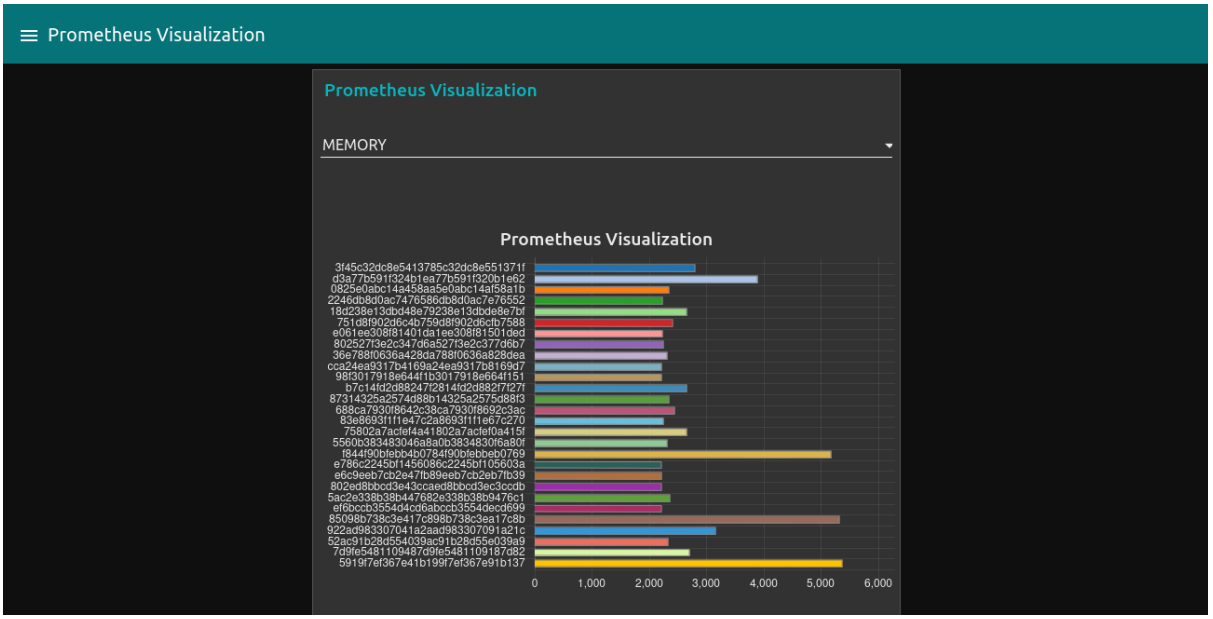
All activation id visualized (with all their categories):

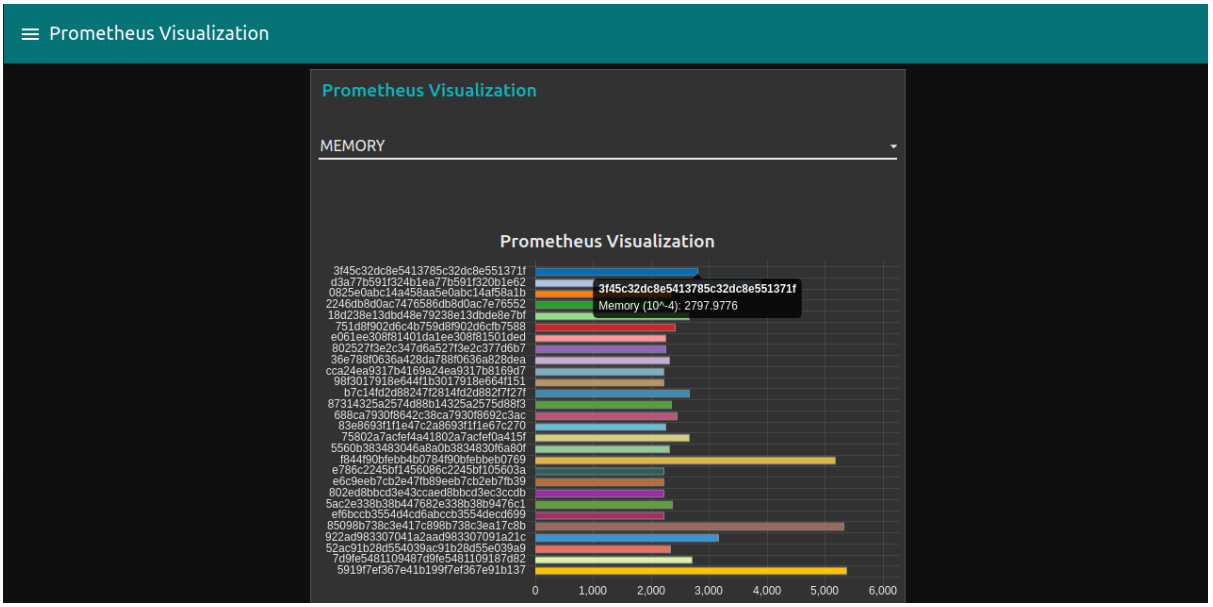


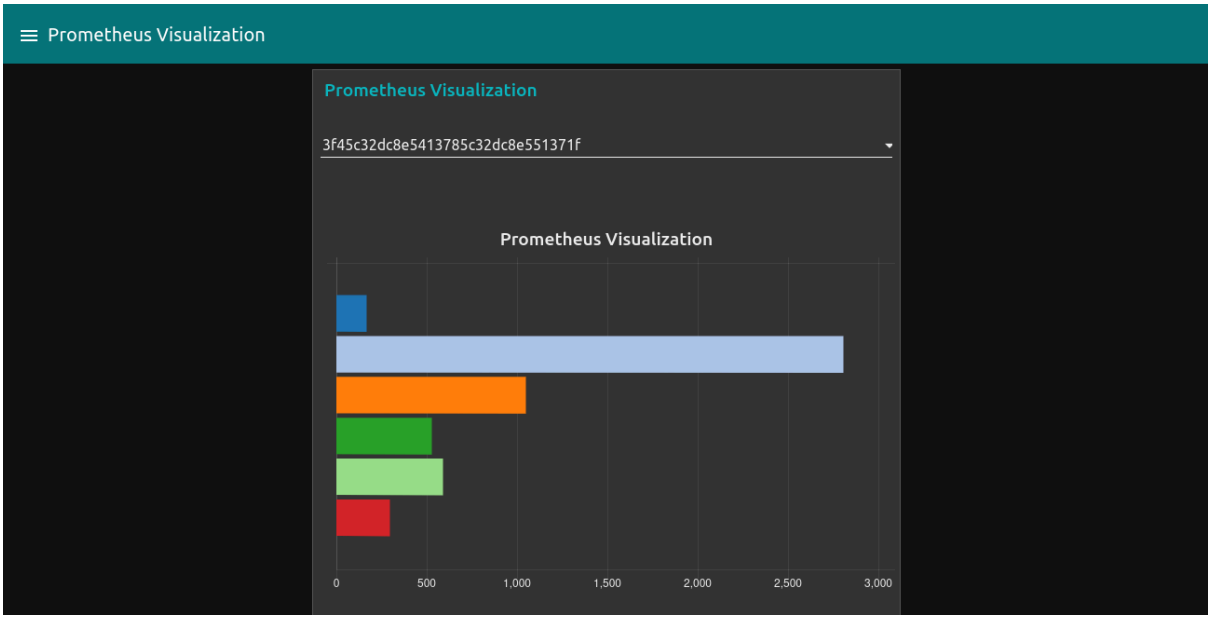


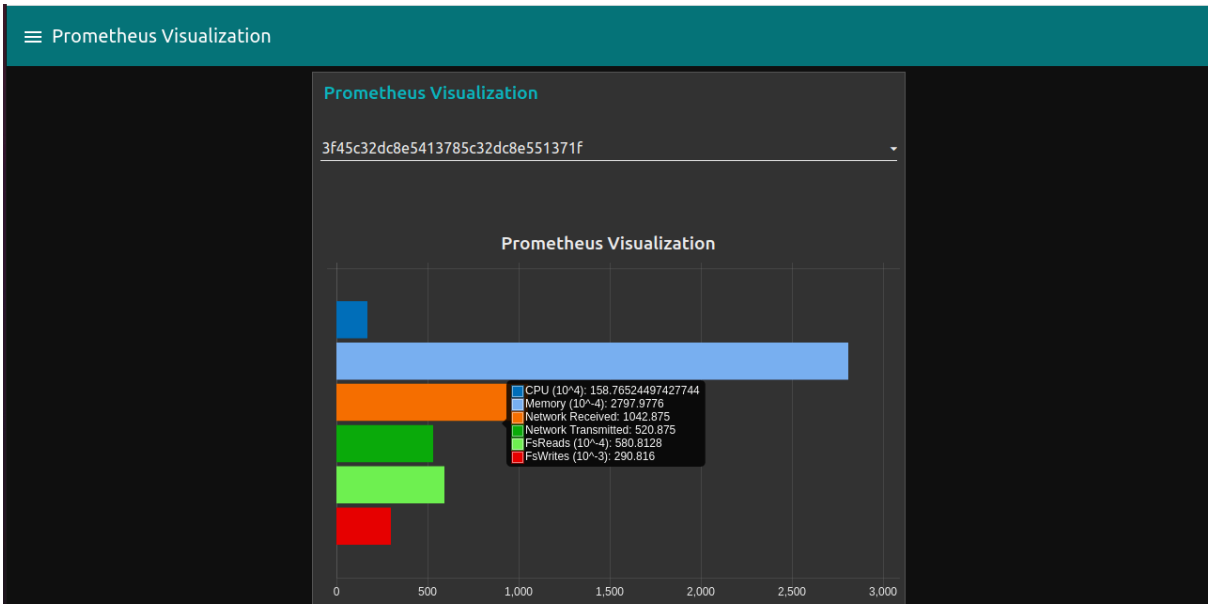
Users can also choose a category or one activation id:

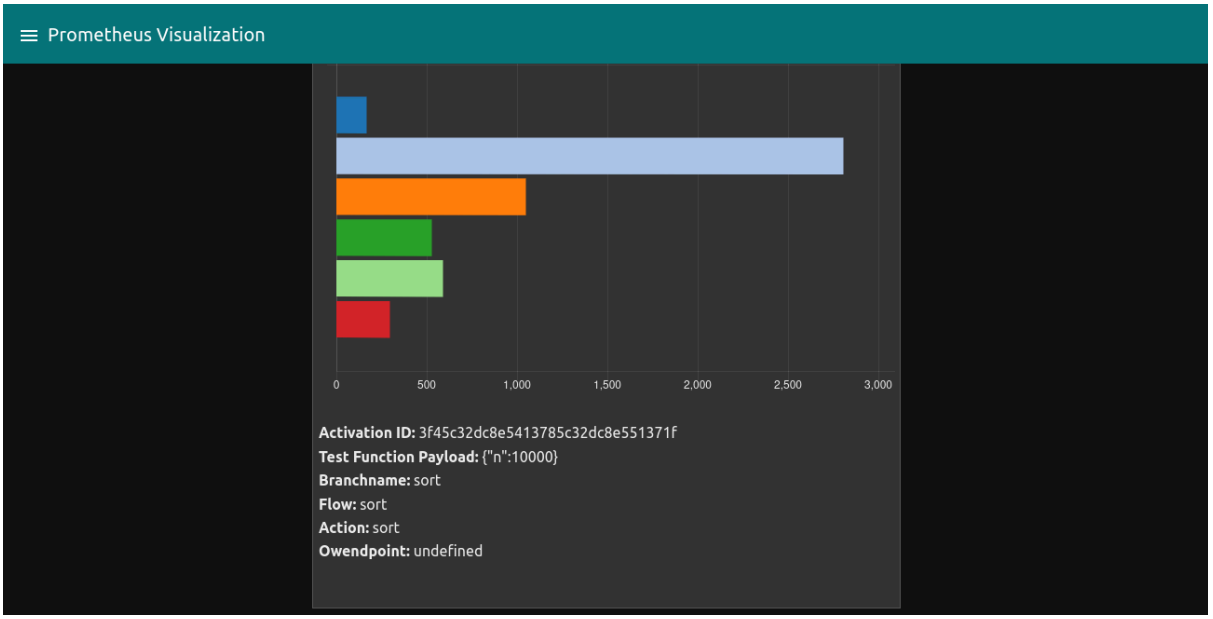



Prometheus Visualization
Prometheus Visualization
MEMORY
Prometheus Visualization
3f45c32dc8e5413785c32dc8e551371f
d3a77b591f324b1ea77b591f320b1e62
0825e0abc14a458aa5e0abc14af58a1b
2246db8d0ac7476586db8d0ac7e76552
18d238e13dbd48e79238e13dbde8e7bf
751d8f902d6c4b759d8f902d6cfb7588
e061ee308f81401da1ee308f81501ded
802527f3e2c347d6a527f3e2c377d6b7
36e788f0636a428da788f0636a828dea
cca24ea9317b4169a24ea9317b8169d7
98f3017918e644f1b3017918e664f151
b7c14fd2d88247f2814fd2d882f7f27f
87314325a2574d88b14325a2575d88f3
688ca7930f8642c38ca7930f8692c3ac
83e8693f1f1e472a8693f1f1e67c270
75802a7acfef4a41802a7acfef0a415f
5560b383483046a8a0b3834830f6a80f
f844f90bfebb4b0784f90bfebbeb0769
e786c2245bf1456086c2245bf105603a
e6c9eeb7cb2e47fb89eeb7cb2eb7fb39
802ed8bbcd3e43ccaed8bbcd3ec3ccdb
5ac2e338b38b447682e338b38b9476c1
ef6bccb3554d4cd6abccb3554decd699
85098b738c3e417c898b738c3ea17c8b
922ad983307041a2aad983307091a21c
52ac91b28d554039ac91b28d55e039a9
7d9fe5481109487d9fe5481109187d82
5919f7ef367e41b199f7ef367e91b137
0
1,000
2,000
3,000
4,000
5,000
6,000

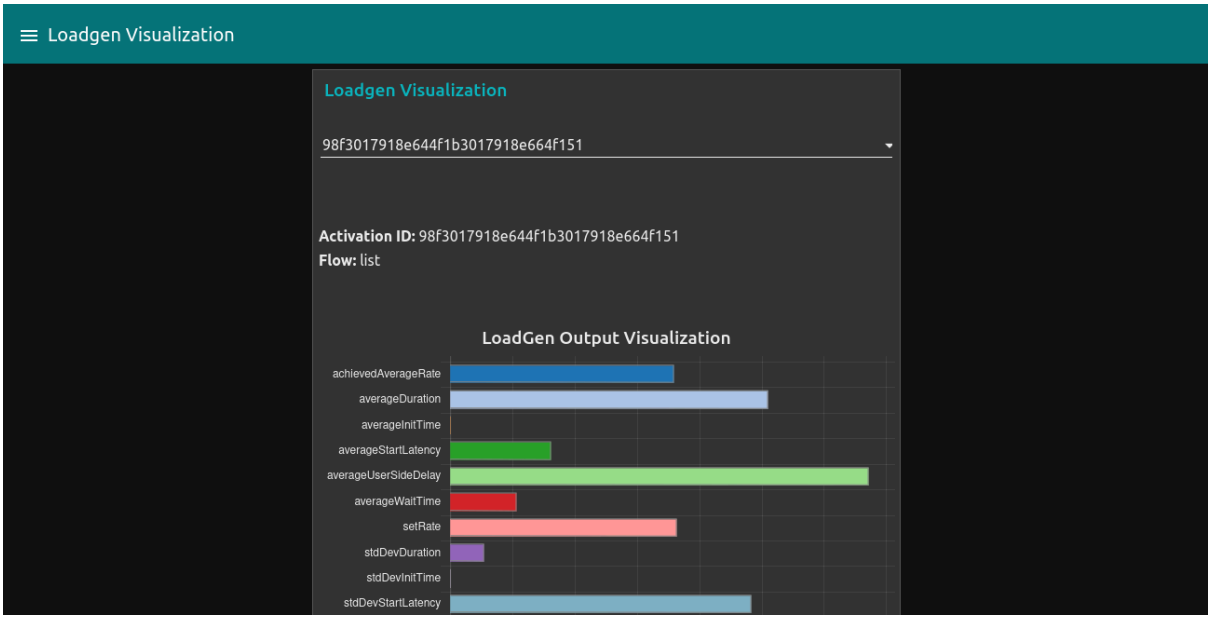

Prometheus Visualization
Prometheus Visualization
MEMORY
Prometheus Visualization
3f45c32dc8e5413785c32dc8e551371f
Memory (10^-4): 2797.9776
0
1,000
2,000
3,000
4,000
5,000
6,000


Prometheus Visualization
Prometheus Visualization
3f45c32dc8e5413785c32dc8e551371f
Prometheus Visualization
0
500
1,000
1,500
2,000
2,500
3,000

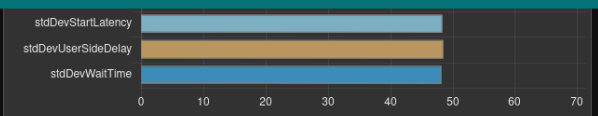

Prometheus Visualization
Prometheus Visualization
3f45c32dc8e5413785c32dc8e551371f
Prometheus Visualization
CPU (10^4): 158.76524497427744
Memory (10^-4): 2797.9776
Network Received: 1042.875
Network Transmitted: 520.875
FsReads (10^-4): 580.8128
FsWrites (10^-3): 290.816
0
500
1,000
1,500
2,000
2,500
3,000



Loadgen Visualization:



Loadgen Visualization



Action:
Method Payload:
Action: list
Namespace: guest
Version: 0.0.82
Status: Completed
Test Name: openwhisk_ow_long_11_1_10_1
Creds:
17dz39c2-62f7-3ed6-9b88-925ce4l3c104:459bO3xUERmNZ9v2BLU1dRYFpXlPkccOFqr

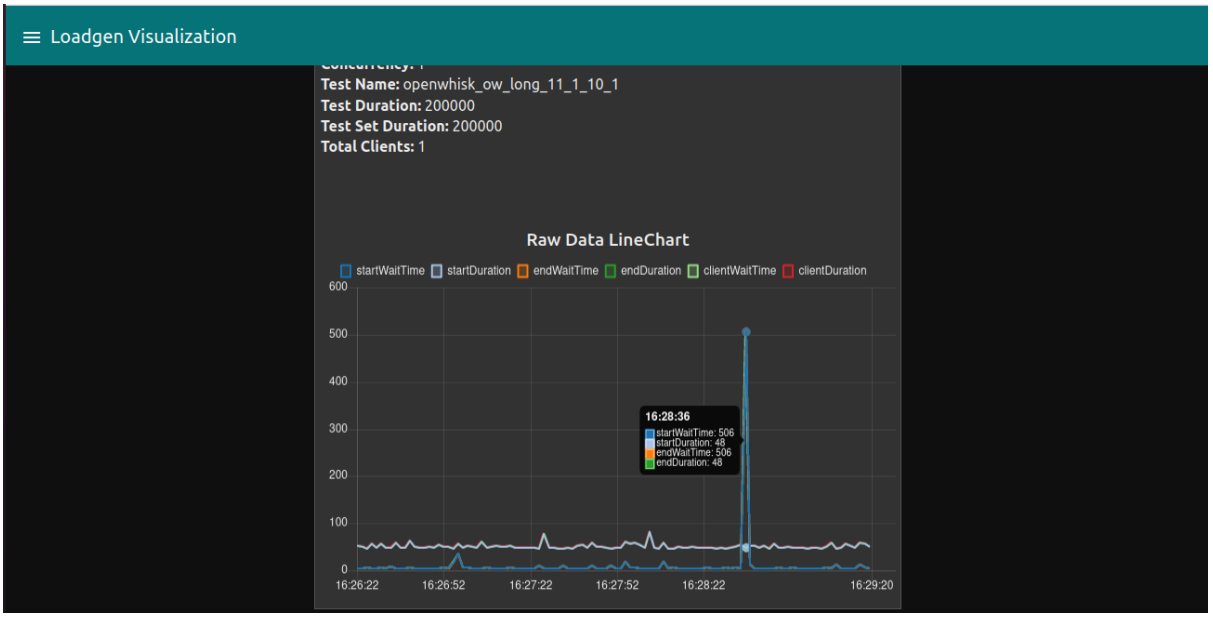
Start Time:
Actual Start Time: 5/3/2023, 4:26:20 PM
Global Start Time: 5/3/2023, 4:26:20 PM

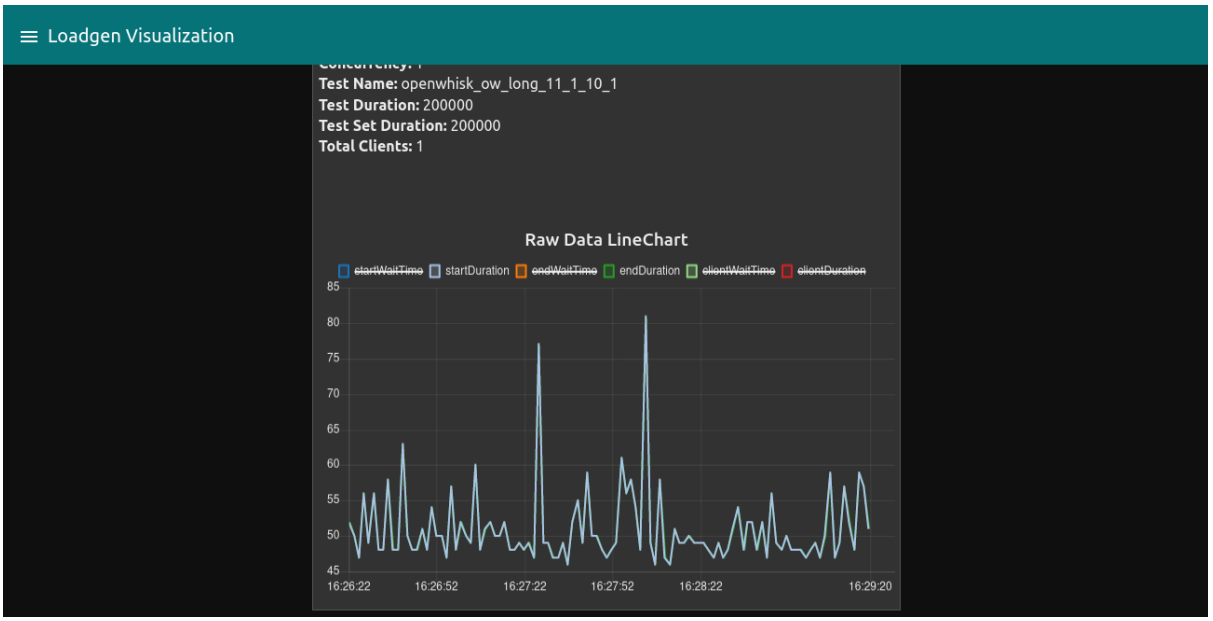
Client Information:
Node Type: physics_openwhisk:faas
Endpoint Method: POST
Load Gen Endpoint: https://172.30.197.240/api/v1/namespaces/guest/actions/loadgen
Delay: 1655.7

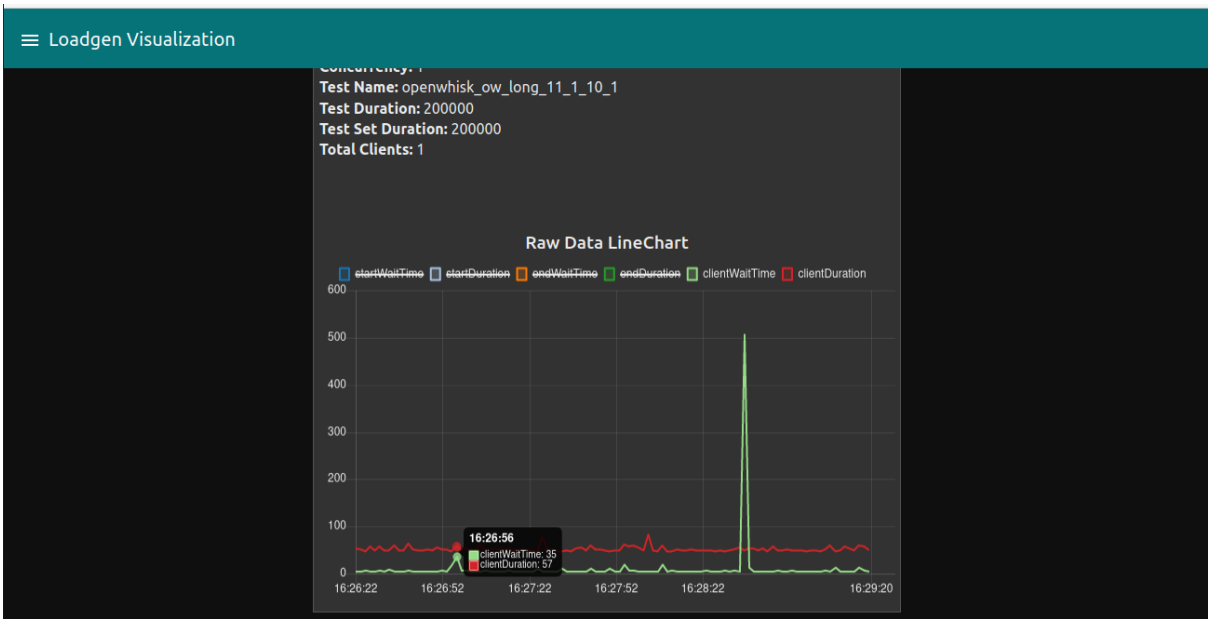
Loadgen Visualization

Client Information:
Node Type: physics_openwhisk:faas
Endpoint Method: POST
Load Gen Endpoint: https://172.30.197.240/api/v1/namespaces/guest/actions/loadgen
Delay: 1655.7
Client Number: 1
Cold Starts: 0
Launch Generator Delay: 160

Additional Information:
Memory: 256
Other Info: 1000
Parent Sample Time: 5/3/2023, 4:26:20 PM
Sample Number: 108
Status: Completed
Status Endpoint: https://172.30.197.240/api/v1/namespaces/guest/activations/
Target Endpoint: https://172.30.197.240/api/v1/namespaces/guest/actions/list
Success: true
Success Percentage: 100
Concurrency: 1
Test Name: openwhisk_ow_long_11_1_10_1
Test Duration: 200000
Test Set Duration: 200000
Total Clients: 1


Loadgen Visualization
Test Name: openwhisk_ow_long_11_1_10_1
Test Duration: 200000
Test Set Duration: 200000
Total Clients: 1
Raw Data LineChart
startWaitTime
startDuration
endWaitTime
endDuration
clientWaitTime
clientDuration
600
500
400
300
200
100
0
16:28:36
startWaitTime: 506
startDuration: 48
endWaitTime: 506
endDuration: 48
16:26:22
16:26:52
16:27:22
16:27:52
16:28:22
16:29:20


Loadgen Visualization
Test Name: openwhisk_ow_long_11_1_10_1
Test Duration: 200000
Test Set Duration: 200000
Total Clients: 1
Raw Data LineChart
startWaitTime
startDuration
endWaitTime
endDuration
clientWaitTime
clientDuration
85
80
75
70
65
60
55
50
45
16:26:22
16:26:52
16:27:22
16:27:52
16:28:22
16:29:20



Users can choose what will be displayed on the line chart.

Common Functions Challenge:

For the Common Functions Challenge, we have developed a versatile Docker image that includes three functions capable of handling a wide range of data analysis tasks, including weather data processing. While our primary focus was on integrating with four different weather APIs, these functions can be applied to any scenario where sorting, weighted averaging, or K-means clustering is required.
The first function is a customizable Quicksort algorithm designed to sort a given list of dates and values. This functionality is particularly useful for organizing weather data or any dataset that requires chronological ordering. By setting the 'reverse' parameter to true, users can obtain the more recent dates first.

The second function implements a flexible weighted average calculation. By providing a list of dates and values, users can calculate the average with a varying emphasis on recent dates. By default, the more recent dates have a higher impact on the average calculation. However, setting the 'reverse' parameter to true will reverse the weighting, giving more significance to earlier dates.

Lastly, our solution includes the K-means clustering function which can be applied to any dataset to identify clusters or groups based on the provided data points (for more info for the k-means clustering, see here).

You can access the Docker image containing these versatile functions here.

Openwhisk Run:

If you do not have openwhisk already installed, check the slides of this presentation, and follow the instructions.

First of all you have to be connected to the **Harokopio University Server with VPN**.

Secondly configure the whisk properly using this commands :

```
wsk property set --apihost https://10.100.59.208
wsk property set --auth
23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12Cd
AsMgRU4VrNZ9lyGVCGuMDGIwP
```

Once you have pulled the docker image, do the following to create the openwhisk action:

```
wsk action create -i <action_name> --docker kazakos13/common-functions
```

In the following examples, we set <action_name> to “functionstest”.

Here is a test run of the **weighted average** in openwhisk:

```
wsk -i action invoke functionstest --param data '{
  "temperature": [
    { "value": 20, "date": 1664515200000 },
    { "value": 23, "date": 1664774400000 },
    { "value": 18, "date": 1664428800000 },
    { "value": 25, "date": 1664601600000 },
    { "value": 22, "date": 1664688000000 }
  ],
  "humidity": [
    { "value": 20, "date": 1664515200000 },
    { "value": 23, "date": 1664774400000 },
    { "value": 18, "date": 1664428800000 },
    { "value": 25, "date": 1664601600000 },
    { "value": 22, "date": 1664688000000 }
  ]
}' --param reverse false --param function wavg
```

Then you take the id and run the following:

```
wsk activation get -i <activation_id>
```

You should take a result like so (it is in “value”[“data”]):


```
ok: got activation 1715ad361b08410495ad361b080104cf
{
    "namespace": "guest",
    "name": "functionstest",
    "version": "0.0.1",
    "subject": "guest",
    "activationId": "1715ad361b08410495ad361b080104cf",
    "start": 1685626783313,
    "end": 1685626783349,
    "duration": 36,
    "statusCode": 0,
    "response": {
        "status": "success",
        "statusCode": 0,
        "success": true,
        "result": {
            "action_name": "/guest/functionstest",
            "action_version": "0.0.1",
            "activation_id": "1715ad361b08410495ad361b080104cf",
            "deadline": "1685626843312",
            "namespace": "guest",
            "transaction_id": "yMRST8JTpAuVWehNNLe4Y2hP9wtIneHY",
            "value": {
                "data": {
                    "humidity": 22.4,
                    "temperature": 22.4
                },
                "function": "wavg",
                "reverse": false
            }
        }
    },
    "logs": [],
    "annotations": [
        {
            "key": "path",
            "value": "guest/functionstest"
        },
        {
            "key": "waitTime",
            "value": 435
        },
        {
            "key": "kind",
            "value": "blackbox"
```

```
        },
        {
            "key": "timeout",
            "value": false
        },
        {
            "key": "limits",
            "value": {
                "concurrency": 1,
                "logs": 10,
                "memory": 256,
                "timeout": 60000
            }
        }
    ],
    "publish": false
}
```

Here is also a input for running **quicksort** in openwhisk:

```
wsk -i action invoke functionstest --param data '{
  "temperature": [
    { "value": 20, "date": 1664515200000 },
    { "value": 23, "date": 1664774400000 },
    { "value": 18, "date": 1664428800000 },
    { "value": 25, "date": 1664601600000 },
    { "value": 22, "date": 1664688000000 }
  ],
  "humidity": [
    { "value": 20, "date": 1664515200000 },
    { "value": 23, "date": 1664774400000 },
    { "value": 18, "date": 1664428800000 },
    { "value": 25, "date": 1664601600000 },
    { "value": 22, "date": 1664688000000 }
  ]
}' --param reverse true --param function quicksort
```

Similarly, you run the following to get the result:

```
wsk activation get -i <activation_id>
```

To change between functions, put the desired function in the “function” param. Your options are:

- wavg
- quicksort
- kmeans

the input should be similar to the one specified in the docker image.

** make sure to have put the credentials key in the .node-red/settings.js before running wsk activation get -i:

```
module.exports = {
  // ...
  credentialSecret: 'your-credential-secret',
  // ...
};
```

Extra Weather APIs:

To leverage the aforementioned functions, we integrated several weather APIs to access up-to-date weather data. Here is a comprehensive list of the four APIs that were utilized for retrieving current weather data in our project:

API	Input	Output
https://www.weatherapi.com/docs/	Latitude, Longitude, API key	Current Weather data for given latitude, longitude.
https://docs.tomorrow.io/reference/realtime-weather	Latitude, Longitude, API key	Current Weather data for given latitude, longitude.
https://api.open-meteo.com/v1/forecast?latitude=52.52&longitude=13.41&current_weather=true	Latitude, Longitude, API key	Current Weather data for given latitude, longitude.
https://openweathermap.org/current	Latitude, Longitude, API key	Current Weather data for given latitude, longitude.

The APIs provide us with various data points, which we calculate and display as averages. The values obtained from these APIs include:

- Temperature
- Wind speed
- Pressure
- Visibility
- Humidity
- Cloud coverage

Lastly, we retrieve the following information from OpenWeatherMap display purposes, including icons and weather descriptions:

- Sunrise time
- Sunset time
- Weather icon
- Weather description

By incorporating these data sources, we ensure comprehensive and informative weather representations.

Subflows Challenge:

We have developed some subflows for the Subflows Challenge, which you can find in this collection:

1. **Quicksort**
2. **Weighted average**
3. Collecting and calculating the weighted **averages** of **four different weather APIs**. The more recent dates have a higher impact on the weighted average for each weather category (like temperature, humidity etc).
4. A subflow that runs with **Openwhisk** and contains two of the previously stated functions (quicksort, weighted average) and a k-means clustering from hackathon (which you can find here). Users can set a parameter “function” to choose which function will be executed. You can find the docker image with a detailed description on how to run it with Openwhisk and what input you should send here or previously in this document
5. **City Data** subflow, is a subflow that collects various city information (like city rankings, summary and forecast for the next seven days) from apis. The apis used in this subflow are teleport, openstreetmap, and openmeteo API.

We have also created two more advanced versions of the QuickSort and Weighted Average subflows, named **quicksort_request** and **weighted_avg_request**. These subflows allow you to request data from a URL, which can be set by the user in the subflow's UI. By default, the URL is set to the Docker image URL.

All subflows included in this challenge come with detailed documentation on how to effectively utilize their functionalities. Here's a brief description of each subflow:

Weighted Average Subflow:

The Weighted Average subflow calculates the weighted average of given values, considering the date of each value. The more recent the date, the higher the value will be weighted in the average calculation. The input structure is as follows (place this in msg.payload.value.data):

```
{
  "attr_name1": [
    {"value": x, "date": y},
    {"value": z, "date": e}
  ],
  "attr_name2": [
    {"value": q, "date": w},
    ...
  ]
}
```

In the input list, each attribute (attr_name1, attr_name2, etc.) is associated with a list of dictionaries. Each dictionary represents a value entry and contains two keys:

- "value": Specifies the value of the attribute for a specific date.
- "date": Specifies the date of the value entry in UNIX timestamp format.

The subflow returns an object with keys corresponding to the attribute names specified in the input list, along with their respective weighted averages. You can access the output in *msg.payload.value.data*. To give higher weight to older dates in the weighted average, set *msg.payload.value.reverse* to true.

QuickSort Subflow:

The QuickSort subflow implements the Quick Sort algorithm and allows sorting a list of values. It can perform a reverse sort if desired by setting *msg.payload.value.reverse* to true; otherwise, set it to false. The input structure is the same as for the Weighted Average subflow:

```
{
  "attr_name1": [
    {"value": x, "date": y},
    {"value": z, "date": e}
  ],
  "attr_name2": [
    {"value": q, "date": w},
    ...
  ]
}
```

In the input list, each attribute (attr_name1, attr_name2, etc.) is associated with a list of dictionaries. Each dictionary represents a value entry and contains two keys:

- "value": Specifies the value of the attribute for a specific date.
- "date": Specifies the date of the value entry in UNIX timestamp format.

The subflow returns the same structure as the input list, with the lists of each attribute key sorted either normally or in reverse order, based on the chosen option. The output can be found in *msg.payload.value.data*.

Many Weather APIs Average Subflow:

The Many Weather APIs Average subflow calculates the weighted averages of weather information from multiple weather APIs to provide a more precise estimation of the current weather conditions. It comprises several subflows, including QuickSort, Weighted Average, and Weighted Average with Docker integration, each with its own dedicated documentation.

To use this subflow, provide the latitude and longitude coordinates (as floats) in the *msg.city.lat* and *msg.city.lng* properties respectively. Additionally, you need to provide the API keys for the following weather APIs:

- OpenWeatherMap: Specify the API key in the subflow's configuration or use *msg.openweathermapapi_key* (with msg having priority).
- Tomorrow API: Specify the API key in the subflow's configuration or use *msg.tomorrowapi_key* (with msg having priority).
- WeatherAPI: Specify the API key in the subflow's configuration or use *msg.weatherapi_key* (with msg having priority).

Note that the OpenMeteo API is also utilized but does not require an API key. The output can be found in *msg.payload* and is an object similar to the response from the OpenWeatherMap API, but with the values replaced by the average weighted values calculated from all the previous APIs.

CommonFunctions Subflow:

The CommonFunctions subflow is an integral part of this docker image, providing three essential flows: k-means, quicksort, and weighted average. Users can easily select their preferred function by modifying the function parameter accordingly. For more detailed information on this subflow, please refer to the documentation available within the docker image.

City Data Subflow:

This subflow retrieves data for a specific city by accepting the city name as input. The city name can be set either through the user interface (UI) of this node or by using the msg.city_name attribute in the message (message takes priority). The subflow collects data from the teleport, openstreetmap, and openmeteo APIs.

The resulting data includes city rankings, latitude, longitude, country information, and a weather forecast for the next 7 days.

Example Result:


```
{
     "city": "perth",
     "lat": -31.9558933,
     "lng": 115.8605855,
     "rank": {
       "housing": 5.233,
```

```
      "cost_of_living": 4.795,
      "startups": 5.127000000000001,
      "venture_capital": 3.102,
      "travel_connectivity": 2.1104999999999996,
      "commute": 4.695,
      "business_freedom": 9.399666666666667,
      "safety": 7.1045,
      "healthcare": 9.312666666666665,
      "education": 5.143,
      "environmental_quality": 8.132500000000002,
      "economy": 6.0695000000000014,
      "taxation": 4.5885,
      "internet_access": 4.1785,
      "leisure_culture": 4.5115,
      "tolerance": 7.470500000000001,
      "outdoors": 5.6835
    },
    "forecast": [
      {
        "time": "2023-06-03",
        "weathercode": 3,
        "temperature": 13.15,
        "uv_index_max": 3.9,
        "uv_index_clear_sky_max": 3.9,
        "rain_sum": 0,
        "snowfall_sum": 0,
        "windspeed_10m_max": 8.7,
        "winddirection_10m_dominant": 78,
        "shortwave_radiation_sum": 12.21
      },
      ...
    ],
    "photos": {
      "mobile":
"https://d13k13wj6adfdf.cloudfront.net/urban_areas/perth-1e220f50f9.jpg"
,
      "web":
"https://d13k13wj6adfdf.cloudfront.net/urban_areas/perth_web-99a082a61e.
jpg"
    }
  }
```

Weighted Average Request Subflow:

This subflow facilitates communication with a specified URL, which can be customized in the user interface (UI) by modifying the FUNCTIONS_URL attribute. By default, it connects to a Docker container running on the user's PC and executes the Weighted Average Common Function. The Docker image used for this purpose is available on Docker Hub at kazakos13/common-functions. In case the Docker endpoint is inaccessible, the subflow automatically switches to running the local flow of the Weighted Average subflow.

To utilize the subflow, an input list must be provided, following a specific structure. This list should be placed within the msg.payload.value.data property. The structure of the input list is as follows:

```
{
  "attr_name1": [
    {"value": x, "date": y},
    {"value": z, "date": e}
  ],
  "attr_name2": [
    {"value": q, "date": w},
    ...
  ]
}
```

Each attribute, represented by attr_name1, attr_name2, etc., is associated with a list of dictionaries. Each dictionary represents a value entry and contains two fixed keys:

- "value": Indicates the value of the attribute for a specific date.
- "date": Specifies the date of the value entry in UNIX timestamp format.

The subflow returns an object with keys corresponding to the attribute names specified in the input list, along with their respective weighted averages. The resulting output is accessible through the msg.payload.value.data property.

If desired, it is possible to assign higher weights to older dates in the weighted average calculation. To enable this functionality, set msg.payload.value.reverse = True.

Quicksort Request Subflow:

This subflow facilitates communication with a specified URL, which can be customized in the user interface (UI) by modifying the FUNCTIONS_URL attribute. By default, it connects to a Docker container running on the user's PC and executes the Quicksort Algorithm. The Docker image used for this purpose is available on Docker Hub at kazakos13/common-functions. In case the Docker endpoint is inaccessible, the subflow automatically switches to running the local Quicksort subflow.

To utilize the subflow, an input list must be provided, following a specific structure. This list should be placed within the msg.payload.value.data property. The structure of the input list is as follows:

```
{
  "attr_name1": [
    {"value": x, "date": y},
    {"value": z, "date": e}
  ],
  "attr_name2": [
    {"value": q, "date": w},
    ...
  ]
}
```

Each attribute, represented by attr_name1, attr_name2, etc., is associated with a list of dictionaries. Each dictionary represents a value entry and contains two fixed keys:

- "value": Indicates the value of the attribute for a specific date.
- "date": Specifies the date of the value entry in UNIX timestamp format.

The subflow returns a structure identical to the input list, but with the lists of each attribute key sorted either in normal order or in reverse order, based on the chosen option. The resulting output is accessible through the msg.payload.value.data property.

In addition to the standard sorting, the user has the option to obtain the sorted lists in reverse order. To enable this functionality, set msg.payload.value.reverse = True.

For **further details** and usage instructions, please refer to the documentation provided for each subflow in node-red or in the node-red collection.

Source Code (& video link):

To view the source code of this project, please visit the following github repo. If you specifically want to view the code for the hackathon challenges 2023, click here. If you want to view a video demonstration of our hackathon solutions, click here. Here is also a short video showcasing the app.
Docker image link can be found here. Lastly, the collection of subflows created for hackathon can be found in this link.