

# **Finalna dokumentacja projektu z Big Data**

Sebastian Trojan  
01171271@pw.edu.pl  
320664

Wiktor Woźniak  
01171277@pw.edu.pl  
320670

Mateusz Nizwantowski  
01161932@pw.edu.pl  
313839

11 stycznia 2025

# Spis treści

<b>1 Wstęp</b>	<b>4</b>
1.1 Co zrobiliśmy . . . . .	4
1.2 Cele . . . . .	5
1.3 Istniejące podobne rozwiązania . . . . .	5
<b>2 Źródła danych</b>	<b>6</b>
2.1 Dane pogodowe . . . . .	6
2.2 Dane dotyczące komunikacji publicznej . . . . .	6
<b>3 Szczegółowe informacje o danych</b>	<b>7</b>
3.1 Dane po warszawsku . . . . .	7
3.1.1 Lokalizacja on-line pojazdów komunikacji miejskiej m.st. Warszawy . . . . .	7
3.1.2 Informacje o przystankach . . . . .	8
3.1.3 Trasy pojazdów transportu miejskiego . . . . .	9
3.2 MeteoSource . . . . .	10
<b>4 Używane technologie</b>	<b>11</b>
<b>5 Architektura</b>	<b>11</b>
5.1 Porównanie z naszą pierwotną wizją architektury . . . . .	12
5.1.1 Co zostało takie same . . . . .	13
5.1.2 Co się zmieniło . . . . .	13
5.2 Opis finalnej architektury . . . . .	13
5.2.1 NiFi . . . . .	14
5.2.2 Kafka . . . . .	14
5.2.3 Spark - część analityczna . . . . .	14
5.2.4 Spark structured streaming - część przetwarzająca dane w czasie rzeczywistym . . . . .	15
5.2.5 HDFS . . . . .	15
5.2.6 Platforma do wyświetlania danych (aplikacja webowa) . .	15
5.3 Plik docker compose . . . . .	16
5.3.1 NiFi . . . . .	16
5.3.2 Kafka . . . . .	17
5.3.3 Spark . . . . .	17
5.3.4 HDFS . . . . .	18
5.3.5 Flask . . . . .	19
5.3.6 Używane zasoby komputerowe . . . . .	19
<b>6 Opis przetwarzania i składowania danych</b>	<b>20</b>
6.1 Pierwotny pogodowy schemat w NiFi . . . . .	20
6.2 Finalne przepływy danych . . . . .	21
<b>7 Opis Analizy danych w spark</b>	<b>23</b>

<b>8</b>	<b>Testy systemu</b>	<b>26</b>
<b>9</b>	<b>Planowane funkcjonalności</b>	<b>27</b>
9.1	Oglądarka lokalizacji w czasie rzeczywistym na mapce . . . . .	27
9.2	Możliwość wykonywania analiz . . . . .	27

# 1 Wstęp

## 1.1 Co zrobiliśmy

Stworzyliśmy *proof of concept (POC)* systemu do przetwarzania danych lokalizacji w czasie rzeczywistym warszawskiej komunikacji miejskiej. Dlaczego tylko POC a nie pełne rozwiązanie?

Jest kilka powodów. Po pierwsze taki projekt to ogromne przedsięwzięcie, i jego zakres znacząco by przewyższał czas przewidziany na wykonanie projektu w ramach przedmiotu "Składowanie danych w systemach Big Data". Nie chcemy tutaj teraz skupiać się na przewidywaniu ile by mógł taki pełno-prawny projekt zająć czasu. Ale wydaje nam się, że doprowadzenie do stanu podawania przewidywanych opóźnień na podstawie modelu uczenia maszynowego, UI webowym i aplikacja mobilna z pewnością mogła by zająć kilka miesięcy trzem programistom.

Co więcej, każdy z nas chce iść na studia magisterskie "Data Science". Jest tam przedmiot, który jest swoistą kontynuacją i rozszerzeniem "Składowania danych w systemach Big Data". W naszych rozmowach z profesorem prowadzącym wykład dowiedzieliśmy się, że zamysł jest taki aby i projekt, który wykonywaliśmy był kontynuowany na studiach magisterskich, i taka była idea która nam przyświecała przy tworzeniu tego rozwiązania. Po rozmowach z kolegami ze starszego rocznika, usłyszeliśmy, że 60% czasu poświęconego na tamten projekt polega na tym, żeby skonfigurować używane systemy Big Data i zintegrować podstawowe komponenty, dopiero wtedy coś zaczyna działać. My poszliśmy trochę pod prąd i część z tej pracy wykonaliśmy już teraz.

*Proof of concept* jest spowodowany również tym że nasz cel był taki, aby można było wszystkie wymagane serwisy odpalić na jednym urządzeniu, co gryzie się z definicją systemów Big Data. Co prawda w naszej konfiguracji *HDFS* postawiliśmy jeden *namenode* i dwa *datanode* jednak nadal były one odpalone na jednym urządzeniu. Każdy serwis jest wewnątrz *dockerowego* kontenera, komunikują się ze sobą po sieci stworzonej w dockerze a to wszystko połączyczyliśmy za pomocą funkcjonalności *docker-compose*. Dzięki temu łatwo odpalić nasze rozwiązania na dowolnej UNIX-like maszynie, przetestować je, wejść w interakcję z systemem, a nawet po drobnych modyfikacjach używać jako rozwiązanie produkcyjne.

Jesteśmy jednak świadomie, że to co stworzyliśmy nie jest skalowalne. Jednakże warto zaznaczyć, że w naszym rozwiążaniu każdy z serwisów może być na osobnym urządzeniu, i wymagało by to tylko drobnych modyfikacji. Ten fakt znacząco zwiększa możliwości i wolumen z jakim możemy pracować. Jest to zdecydowana przewaga naszego rozwiązania w porównaniu do używania zapewnionej maszyny wirtualnej. Łatwo również było by uzyskać "statyczną skalowalność", na przykład jeżeli serwery które obsługują *HDFS* zaczynają, nie dawać sobie rady z ilością zgromadzonych danych, w przeciągu kilku minut byliby-

śmy w stanie dodać nową instancję do istniejącego już klastra. Trzeba jednak wspomnieć, że byłby to proces manualny. Aby to zmienić należało by wymieścić *docker-compose* na *kubernetes* lub podobny odpowiednik dostarczany przez operatorów chmurowych. Pozwalało by to na komunikację pomiędzy maszynami oraz na automatyczne skalowanie komponentów najbardziej obciążonych. Według nas jest to jednak poza zakresem tego projektu.

## 1.2 Cele

Co do samego systemu przetwarzania danych, umożliwia on przetwarzanie w czasie rzeczywistym z małym opóźnieniem. Potencjalny użytkownik ma możliwość wybrania numeru linii i sprawdzenia na mapce gdzie znajdują się autobusy tej linii w przeglądarce internetowej. Oprócz tego rozwiązanie ma kластer Spark (sztuczny bo na jednej maszynie), umożliwiający przetwarzanie zgromadzonych danych historycznych.

Nie ukrywamy, że jako osoby używające codziennie komunikacji miejskiej, jesteśmy ciekawi jak działa ona od środka, i każdy "insight" jest cenny. Ten projekt z pewnością pozwoli nam na gromadzenie i eksplorację danych. W przeszłości będzie można ten system rozszerzyć o dane z korkami i przeprowadzić bardziej szczegółową analizę opóźnień. Przed przystąpieniem do projektu przeprowadziliśmy analizę rynku i nie znaleźliśmy nigdzie gotowych danych, z lokalizacją czy z opóźnieniami. Jest tylko API z lokalizacją w czasie rzeczywistym, które użyliśmy. Dla naszej trójki komunikacja jest bliska sercu, a że lubimy pogrzebać w danych, to wykorzystaliśmy ten projekt jako możliwość stworzenia odpowiedniej bazy, która nam to umożliwia.

Oczywistym celem tego projektu jest nauczenie się technologii Big Data, uważałyśmy, że go spełniliśmy. Nasze nietypowe podejście, w którym sami konfigurowaliśmy (do pewnego stopnia) technologie z których korzystaliśmy, spowodowało, że zrozumieliśmy je lepiej i głębiej niż gdybyśmy tylko traktowali je jako narzędzia. Co więcej udało nam się stworzyć rozwiązanie, które da się odtworzyć na innych maszynach, nie tylko na naszych komputerach, nie spędzając nad tym zadaniem ogromu czasu. Daje to także potencjał do uruchomienia aplikacji w środowisku chmurowym. Dzięki temu projektowi udoskonaliliśmy umiejętności pracy w grupie i zarządzanie projektami.

## 1.3 Istniejące podobne rozwiązania

Co ciekawe, nie znaleźliśmy rozwiązań, które spełniają nałożone przez nas wymagania. Jesteśmy przekonani, że Zarząd Transportu Miejskiego (ZTM) posiada infrastrukturę do przechowywania danych i ich analizy, jednak (co nas nie dziwi) udostępnia tylko surowe dane. Co więcej ZTM posiada dane dotyczące opóźnień, (w autobusie kierowca ma ekranik z informacją jaka jest różnica w czasie i za ile powinien być w miejscu w którym się znajduje).

Istnieją strony i aplikacje takie jak:

- [MobileMPK](#)
- [JakDojadę](#)
- [CzyNaCzas](#)
- [Jedzie.pl](#)
- [RealBus](#)

W czasie wykonywania projektu, (głównie aplikacji webowej) inspirowaliśmy i porównywaliśmy się do wyżej wymienionych stron, jednak podobnie jak API Miasta Stołecznego Warszawa, pozwalają tylko na przeglądanie danych w czasie rzeczywistym. Zatem z wysokim prawdopodobieństwem nie przechowują danych historycznych, tylko co jakiś czas wysyłają zapytanie do API i otrzymane dane przechowują w *Redisie* lub innym podobnym narzędziu. Czy na czas pokazuje kilka zagregowanych wyników, takich jak średnie opóźnienie czy 90 kwantyl, oraz kilka wykresów ale to tyle. Oczywiście, żadne z wyżej wymienionych narzędzi nie pozwala na analizę danych. Jeżeli chcę się takową wykonać trzeba dane uzyskać samemu.

## 2 Źródła danych

### 2.1 Dane pogodowe

Te dane są najbardziej powszechnne. Korzystamy z [MeteoSource](#). Strona pozwala na 400 darmowych zapytań dziennie. Posiada ona rozbudowane dane dla danych miejsc, więc bez problemu można uzyskać dane dla Warszawy. Posiada wiele informacji pogodowych, a także przewidywane prognozy pogody na kolejne dni. Więcej szczegółów dotyczących wybranych danych przedstawimy w dalszej części dokumentu.

### 2.2 Dane dotyczące komunikacji publicznej

Dane o lokalizacji, przystankach i trasach pobieramy z zasobów udostępnianych w ramach inicjatywy "[Dane po warszawsku](#)". Jest to platforma, która działa od 2015. Jej celem jest umożliwienie mieszkańcom tworzenia rozwiązań, które przyczynią się do poprawy ich codziennego życia oraz funkcjonowania samorządu.

“Zgodnie ze światowymi trendami udostępniamy dane w sposób otwarty, nie pobieramy za nie opłat i nie pytamy, kto je pobiera. Staramy się, żeby były możliwie pełne, aktualne i uporządkowane, aby korzystanie z nich było jak najwygodniejsze, ale przede wszystkim aby publikowane dane mogły być paliwem i inspiracją do tworzenia rzeczywistych i innowacyjnych rozwiązań i aplikacji ułatwiających życie mieszkańców naszego miasta” – wyjaśnia Tadeusz Osowski, dyr. Biura Cyfryzacji w stołecznym Ratuszu.

### 3 Szczegółowe informacje o danych

#### 3.1 Dane po warszawsku

Poszczególne endpointy dostarczane przez 'Dane po warszawsku' są najprawdopodobniej utrzymywane przez różne zespoły, ponieważ ich struktura jest różna, a te same dane są pod różnymi kluczami. Na przykład współrzędne geograficzne są raz opisane jako 'Lat' i 'Lon', a raz jako 'szer\_geo' i 'dlug\_geo', zatem nawet nie jest utrzymana spójność językowa.

##### 3.1.1 Lokalizacja on-line pojazdów komunikacji miejskiej m.st. Warszawy

Dane zawierają informację dotyczącą ostatniego znanego położenia poszczególnych pojazdów komunikacji miejskiej m.st. Warszawy. Dane są aktualizowane co 10 sekund. Są przekazywane za pomocą API i zwracają plik json z jednym kluczem 'result'. W środku znajduje się lista około 1900 słowników (wartość ta się zmienia w zależności od czasu). Jedna odpowiedź waży około 0.25MB. Co warto zaznaczyć dostajemy najbardziej aktualne dane dla każdego autobusu, jeżeli nie jest on w trasie, to dostajemy dane z ostatniego czasu kiedy czujnik był włączony. Na przykład najstarszy rekord ma ponad 200 dni. Parametry wywołania:

- apikey – do uzyskania po założeniu konta na api.um.warszawa.pl (parametr obligatoryjny)
- resource\_id - identyfikator zasobu podany przez autora API (parametr obligatoryjny)
- type - parametr zapytania filtrujący rezultaty zapytania (obligatoryjny):
  - 1 - dane dotyczące autobusów
  - 2 - dane dotyczące tramwai
- linie - parametr filtrujący rezultaty zapytania względem numeru linii (opcjonalny)
- brigade - parametr filtrujący rezultaty zapytania względem numeru linii (opcjonalny)

Dostępne informacje i ich struktura w pliku json:

- Lat - współrzędna szerokości geograficznej (float)
- Lon - współrzędna długości geograficznej (float)
- Time - czas wysłania sygnału GPS (string)
- Lines - numer linii autobusowej lub tramwajowej (string). Numer linii nie musi być liczbą np. N01.
- Brigade - numer brygady pojazdu (string)

### 3.1.2 Informacje o przystankach

Właściciel API nie dostarczył dokumentacji, zatem opis danych jest jedynie naszym przypuszczeniem. Dane zawierają szczegółowe informacje dotyczące przystanków autobusowych i tramwajowych w Warszawie. Dane są przekazywane za pomocą API i zwracają plik json z jednym kluczem 'result'. W środku znajduje się lista około 8200 słowników. Jedna odpowiedź waży 5.7MB. Zdecydowaliśmy się na pozyskiwanie danych raz na dzień.

W opisie przystanków jako zespół przystankowy rozumiemy zbiór przystanków mających tę samą nazwę, ale różne numery słupków, np. 'Politechnika 01', 'Politechnika 02'.

Parametry wywołania:

- apikey – do uzyskania po założeniu konta na api.um.warszawa.pl (parametr obligatoryjny)
- id - identyfikator zasobu podany przez autora API (parametr obligatoryjny)

Dostępne informacje i ich struktura w pliku json:

- 'values':
  - key - nazwa klucza (string). Dostępne klucze:
    - zespol - numer zespołu przystankowego
    - slupek - identyfikator słupka
    - nazwa\_zespolu - nazwa zespołu przystankowego
    - id\_ulicy - identyfikator ulicy
    - szer\_geo - współrzędna szerokości geograficznej
    - dlug\_geo - współrzędna długości geograficznej
    - kierunek - nazwa przystanku w kierunku, którego autobusy odjeżdżają z tego przystanku
    - obowiazuje\_od - data od której obowiązuje rozkład
  - value - wartość pod kluczem (string)

### 3.1.3 Trasy pojazdów transportu miejskiego

Dane zawierają informacje dotyczące tras autobusów i tramwai w Warszawie. Dane są przekazywane za pomocą API i zwracają plik json z jednym kluczem 'result'. Jako klucz jest lista 287 słowników. Jedna odpowiedź waży 4.9MB. Tutaj również zdecydowaliśmy się na pozyskiwanie danych raz na dzień.

Parametry wywołania:

- apikey – do uzyskania po założeniu konta na api.um.warszawa.pl (parametr obligatoryjny)

Dostępne informacje i ich struktura w pliku json:

- numer linii - dane odpowiadające kluczowi Lines w pierwszym pliku (string):
  - nazwa trasy - nazwa trasy (string). Nazwa trasy składa się z pewnego identyfikatora np. TP, TX, TZ oraz akronimu przystanku końcowego np. BRP. W dokumentacji nie jest wybrane jak interpretować pierwszą część nazwy trasy, podejrzewamy, że jeśli linia ma różne wersje trasy są one oznaczone innym przedrostkiem. Na przykład dla linii autobusowej 114 pełen kurs Młociny-Ukszw -> Bródno-Podgrodzie jest oznaczony TP-BRP, a kurs skrócony Młociny-Ukszw -> Metro Młociny dla tej samej linii jest oznaczony TX-MMLZ.:
    - numer porządkowy przystanku na trasie (string):
      - odleglosc - odległość od początku trasy w metrach (string)
      - ulica\_id - id ulicy na której znajduje się przystanek pojazdu (string)
    - nr\_zespolu - numer zespołu przystankowego (string). Dane odpowiadające kluczowi zespol w drugim pliku.
    - typ - typ przystanku (string)
    - nr\_przystanku - numer przystanku w zespole na którym zatrzymuje się pojazd (string). Dane odpowiadające kluczowi slupek w drugim pliku.

### 3.2 MeteoSource

Endpoint zawierający informacje dotyczące obecnej pogody i prognozy pozwala na stosowanie wielu parametrów wywołania i zwraca wiele informacji. Dlatego wypisane zostały jedynie parametry i informacje, których używaliśmy lub mogły by być przydatne w rozwoju naszego rozwiązania. Dane są aktualizowane co minutę. My pobieramy je co 10 minut, ze względu na ograniczenie do 400 zapytań na dzień. API zwraca plik json.

Parametry wywołania:

- place\_id - identyfikator miejsca, który można pozyskać za pomocą innego endpointa
- lat - szerokość geograficzna
- lon - długość geograficzna
- sections - sekcje, które mają się znaleźć w odpowiedzi, np. 'current' będzie wybierało obecną sytuację pogodową.
- units - jednostki, w których mają być przedstawione dane, np. metric daje system metryczny

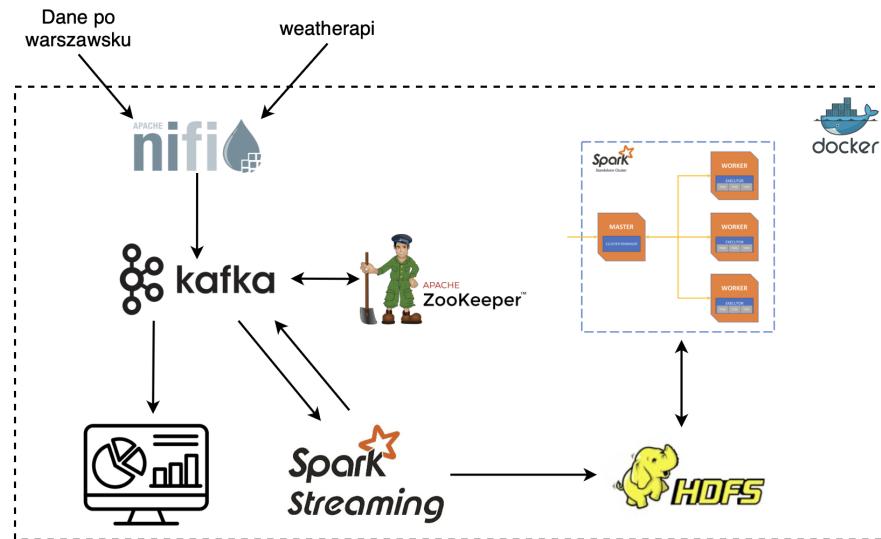
Dostępne informacje i ich struktura w pliku json:

- weather - identyfikator ikony ilustrującej pogodę np. sunny, partly\_sunny, light\_rain, ... (string)
- temperature - temperatura 2 metry nad ziemią (float)
- wind.speed - prędkość wiatru 10 metrów nad ziemią (float)
- wind.angle - kąt wiatru w stopniach (integer)
- wind.dir - kierunek wiatru np. N, NE, NNE, ... (string)
- cloud\_cover.total - procent zachmurzenia (integer)
- pressure - ciśnienie na poziomie morza
- precipitation.type - typ opadów np. none, rain, snow, rain\_snow, ice pellets, frozen rain (string)
- precipitation.convective - suma poziomu opadów z ostatniej godziny (float)

## 4 Używane technologie

- Git
- GitHub
- LaTeX
- Docker
- Docker Compose
- NiFi
- Kafka
- Spark
- Spark Structured Streaming
- ZooKeeper
- HDFS
- Flask

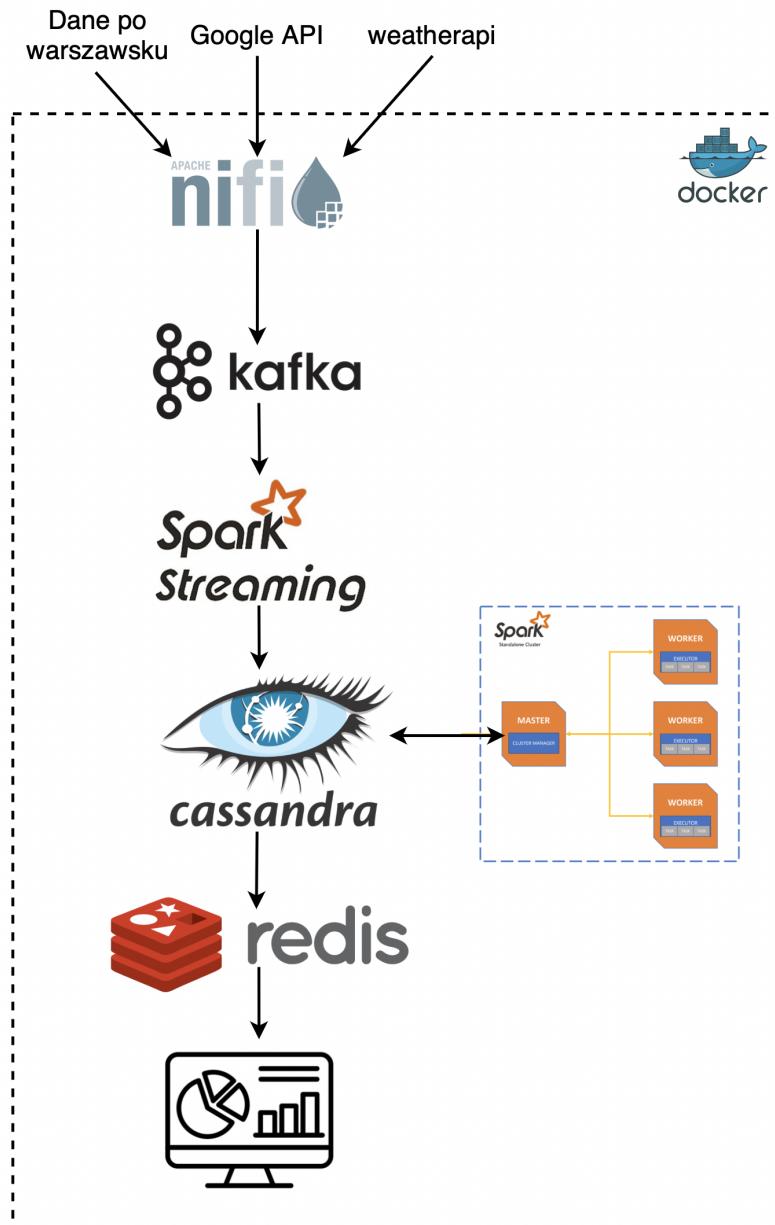
## 5 Architektura



Rysunek 1: Diagram finalnej architektury naszego rozwiązania

## 5.1 Porównanie z naszą pierwotną wizją architektury

Dla przypomnienia przedstawimy nasz pierwotny zamysł na rozwiązanie i jak widać dużo się nie zmienił.



Rysunek 2: Diagram pierwotnej architektury naszego rozwiązania

### **5.1.1 Co zostało takie same**

Ogólna koncepcja pozostała taka sama. Nifi jest tylko wrapeorem na źródła danych, następnie są one ładowane do Kafki, i strumieniowo przetwarzane przez Spark structured streaming. Spark umieszcza dane zarówno w HDFS jak i w Kafce skąd są pobierane przez aplikacje webową. Wizja dockera jako poziomu abstrakcji pomiędzy sprzętem a naszą aplikacją również pozostała bez zmian.

### **5.1.2 Co się zmieniło**

Z racji na ilość innych projektów i faktu, że musimy pisać prace inżynierską, odpuściliśmy sobie jedno źródło danych, jednak planujemy je zintegrować w przyszłości, na studiach magisterskich.

Zamieniliśmy Cassandrę na HDFS, gdyż nie oszukujmy się ale, nie planujemy na razie tego udostępniać i nie potrzebujemy "speed layer", chcemy w przyszłości wykorzystać dane do analizy i HDFS wydajniej przechowuje te same dane (w rozumieniu przestrzeni dyskowej). Zgromadzone dane będziemy procesować batchowo więc, w naszym przypadku użycia HDFS ma wiele sensu, jednak nie ukrywamy, że w prawdziwym projekcie Cassandra czy Hbase sprawowały się lepiej.

Nie wykorzystaliśmy w finalnym rozwiązaniu również Redisa, próbowaliśmy to zrobić i nam się udało, jednak postanowiliśmy uprościć nasze rozwiązanie, i przetworzone dane wrzucamy do Kafki i z Kafki czytamy używającą serwera aplikacji webowej napisanej we Flasku.

## **5.2 Opis finalnej architektury**

Od początku celowaliśmy w architekturę Kappa, jest ona prostsza i traktuje wszystkie dane napływające do systemu jako strumień. Pozwoliło to nam na takie samo postępowanie z danymi lokalizacyjnymi (strumieniowe), jak i z danymi dotyczącymi tras i przystanków (batchowe).

Każdy z komponentów jest wewnątrz kontenera dockerowego, kontenery po razumiewają się ze sobą za pomocą stworzonej sieci, poza tą sieć wychodzą jedynie strony z UI poszczególnych komponentów. Dodatkowo HDFS skonfigurowaliśmy z jednym namenodem i dwoma datanodami. Klaster sparkowy jest tylko koncepcyjny, nie chcieliśmy zbytnio obciążać naszych laptopów, jest to dużo serwisów działających na jednym urządzeniu. Dlatego aby przetestować czy wszystko działa, najpierw wykorzystaliśmy sparka do wsadzenia danych do HDFS a następnie przerwaliśmy procesowanie i włączylismy skrypt do zacztywania danych z HDFS.

Nasze rozwiązanie testowaliśmy na "świeżym" systemie Ubuntu. Dzięki temu jesteśmy pewni, że moglibyśmy nawet puścić nasze rozwiązanie w środowisku chmurowym, z minimalnymi modyfikacjami. Co więcej niewiele potrzeba

aby nasze rozwiązanie mogło działać na wielu urządzeniach. Co z pewnością poprawiło by wydajność i jest bardziej zgodne z przeznaczonym użyciem tych narzędzi, jednak nadal nie zgodne w pełni gdyż nie działające w środowisku multi-urządzeniowym.

### 5.2.1 NiFi

Do łączenia się z naszymi źródłami danych wykorzystaliśmy platformę Apache NiFi. Jest ona świetnym narzędziem do "ciągnięcia" danych z uwagi na zaimplementowane funkcjonalności. Nie wykonujemy na tym etapie przetwarzania danych. Dzięki dockerowi i gitowi jesteśmy w stanie w łatwy sposób wspólnie pracować nad przepływami danych w NiFi. Automatycznie przy tworzeniu i zamknięciu kontenera kopujemy folder w którym NiFi przechowuje te informacje do repozytorium. Pierwotnie planowaliśmy używać Nifi Registry, jednak była to kolejna usługa do konfiguracji, a z racji że nasze przepływy są proste to poszliśmy na łatwiznę (która działa). Jak zaznaczone na diagramie NiFi wpycha dane do Kafki.

### 5.2.2 Kafka

Żaden z nas nie pracował wcześniej z Kafką, więc mieliśmy z nią kilka problemów. Jeden z nich był bardzo uporczywy i rozwiązanie go zajęło nam trzy dni czytania wątków na forach dotyczących Kafki. Mianowicie używamy narzędzia nie w zgodzie z jego zastosowaniem (na jednej maszynie), z tego powodu Kafka na nas "krzyczała" gdy próbowaliśmy stworzyć topic, gdyż ona chce stworzyć trzy kopie i nie ma gdzie ich zrobić. W końcu udało nam się rozwiązać problem i ustalić odpowiednie zmienne konfiguracyjne. Na szczęście, Kafka dobrze współpracuje z NiFi i Sparkiem, i ustawienie aby razem rozmawiały nie było trudne. Stanowi ona buffer w procesowaniu i jest miejscem przechowywania danych strumieniowych w "speed layer". Każde źródło danych ma swój osobny *topic*, czyli razem mamy pięć topików (jeden testowy)

### 5.2.3 Spark - część analityczna

Pierwotnie, ta część miała być klastrem złożonym z 3 kontenerów: 1 master i 2 slave. Oczywiście, nie miało to dużo sensu w przypadku tego jak uruchamiamy całą infrastrukturę (na jednej maszynie), jednak finalnie fizycznie nie ma tego elementu, i mamy w naszym rozwiązaniu tylko jeden kontener ze Sparkiem na którym odpalamy różne skrypty (raz analizę batchową a raz procesowanie strumieniowe. Aby udowodnić, że potrafimy skonfigurować taki mini klaster zrobiliśmy to w ramach HDFS. Ta część infrastruktury bez dwóch zdań była sparkiem i nawet się nad tym nie zastanawialiśmy. Ten moduł ma możliwość analizowania danych z HDFS i wykorzystywania całego potencjału Sparda, w celu znalezienia ciekawych zależności w danych.

#### **5.2.4 Spark structured streaming - część przetwarzająca dane w czasie rzeczywistym**

Podobnie jak powyżej, ten krok jest uruchamiany w jednym sparkowym kontenerze i w zależności jaki skrypt wykonamy taką funkcjonalność otrzymujemy. Nie mieliśmy wcześniej styczności z programowaniem strumieniowym i dziwimy się, że przebiegło to tak bezproblemowo. Nie mieliśmy dużo czasu aby się z tym pobawić, gdyż konfiguracja innych komponentów zajęła tak dużo czasu, więc przeprowadziliśmy tam tylko prostą transformację z JSONa do Parquet, jednak jak to się mówi sky is the limit i moglibyśmy liczyć znacznie więcej rzeczy. Ponownie planujemy rozszerzyć ten moduł i funkcjonalności przez niego oferowane na studiach magisterskich. Logicznie rozdzieliśmy sparkowe komponenty gdyż nie chcemy, aby część analityczna i przetwarzająca dane wchodziły w inferencje ze sobą. W przypadku wykonywania obciążających obliczeń, część analityczna mogłaby znaczco spowolnić obliczenia wykonywane w czasie rzeczywistym i pogorszyć doświadczenie użytkowników. W tym module będziemy przetwarzać dane i zapisywać do bazy danych.

#### **5.2.5 HDFS**

Wiele debatowaliśmy na temat wyboru właściwej bazy danych. Pierwotnie wybór padł na Cassandrę, lecz w trakcie zmieniliśmy rozwiązanie na HDFS. Cassandra idealnie sprawiła by się jako serving layer w architekturze kappa gdybyśmy tworzyli prawdziwą platformę do wizualizacji danych komunikacji miejskiej. Jednak nie ukrywamy, że nie planujemy w najbliższym czasie tego upublicznić i naszym celem jest gromadzenie danych przez dłuższy okres i następnie analiza tego w ramach przedmioty na studiach magisterskich, dlatego teraz zdecydowaliśmy się na HDFS (dane zajmują mniej danych na dysku) i w przyszłości możliwe że zmienimy lub dodamy Cassandrę.

#### **5.2.6 Platforma do wyświetlania danych (aplikacja webowa)**

Podczas gdy, ten krok nie wnosi nic do procesowania danych o dużym wolumenie, to używaliśmy go jako test do weryfikowania opóźnień w naszym systemie i sprawdzania poprawności danych. Nie jest to nic wyrafinowanego, zwykła mapka w której można wybrać, numer linii autobusu, a następnie zobaczyć gdzie on jest, i jakie przystanki znajdują się na jego trasie. Poza tym wygląda fajnie i możemy się popisać procesowaniem strumieniowym.

## 5.3 Plik docker compose

W sercu naszego rozwiązania, jest plik docker compose (poświęcił on większość czasu poświęconego na ten projekt) jednak tworząc go zyskaliśmy głębsze zrozumienie narzędzi niż zwykłe korzystanie z nich. Podczas gdy jego finalna struktura może się wydawać prosta to przechodził on przez wiele iteracji i finalnie cieszymy się że nie jest tak skomplikowany jednakże otrzymanie go było bardzo skomplikowanym procesem. Narzędzia takie jak Kafka, HDFS, czy NiFi nie zostały zaprojektowane z myślą o działaniu na pojedynczej maszynie i sprawienie by zaczęły działać w takim trybie może się okazać wyzwaniem. Spójrzmy teraz na poszczególne fragmenty konfiguracji.

### 5.3.1 NiFi

```
nifi:
  image: 'apache/nifi:1.24.0'
  container_name: nifi_bd
  hostname: nifi_bd
  environment:
    - NIFI_WEB_HTTP_PORT=8080
    - SINGLE_USER_CREDENTIALS_USERNAME=admin
    - SINGLE_USER_CREDENTIALS_PASSWORD=S3curePa55word
    - NIFI_SENSITIVE_PROPS_KEY=pUaEVgyGKT61fMCANbjJPMwAcQDuDj4
    - WARSAW_API_KEY=${WARSAW_API_KEY}
    - WEATHER_API_KEY=${WEATHER_API_KEY}
  volumes:
    - './src/nifi/conf:/opt/nifi/nifi-current/conf'
  ports:
    - '8080:8080' # Nifi web UI
  networks:
    - big-data-network
```

Rysunek 3: Konfiguracja NiFi w docker compose

Nie chcemy się za bardzo rozwodzić i skupiać na każdej linijce, chociaż czasami wymyślenie jednej linijki potrafiło zająć kilka godzin. Ogólny schemat jest taki, że najpierw wybieramy odpowiedni image, czasami decydowaliśmy się na oficjalny dystrybuowany przez apache, czasami natomiast w celu prostszej konfiguracji na jednej maszynie decydowaliśmy się na image, który zrobił ktoś ze społeczności. Następnie ustawiamy nazwę kontenera i hosta, ta druga pozwala na odnoszenia się do innych kontenerów za pomocą zdefiniowanej nazwy zamiast numeru ip (192.168.0.18). Jest to o tyle wygodne, że numery mogłyby się zmieniać w czasie chyba, że ustalilibyśmy je na stałe co też da się zrobić ale wydaje nam się, że przykładowo w nifi łatwiej się odnosić do kafka\_bd niż do tajemniczego nic nie mówiącego adresu ip. Następnie są zmienne środowiskowe, między innymi przekazanie kluczów do api. Na koniec podpinamy wolumen z kontenera do repozytorium aby umożliwić wersjonowanie przepływów danych, ten krok był o tyle nietypowy gdyż kopiowanie pliku, który przechowuje te przepływy, nie działa, jednak gdy weźmiemy cały folder to wszystko zaczyna działać. Na koniec udostępniamy port z UI i podpinamy kontener do stworzonej sieci.

### 5.3.2 Kafka

```
kafka:
  image: wurstmeister/kafka:latest
  container_name: kafka_bd
  hostname: kafka_bd
  depends_on:
    - zookeeper
  environment:
    KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka_bd:9093,OUTSIDE://localhost:9092
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
    KAFKA_LISTENERS: INSIDE://0.0.0.0:9093,OUTSIDE://0.0.0.0:9092
    KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
    KAFKA_ZOOKEEPER_CONNECT: zookeeper_bd:2181
    KAFKA_CREATE_TOPICS: "my-topic:1:1, transport-location:1:1, weather:1:1, stops:1:1, routes:1:1"
    KAFKA_MESSAGE_MAX_BYTES: 10485760 # 10 MB
    KAFKA_REPLICA_FETCH_MAX_BYTES: 10485760
  ports:
    - '9092:9092'
  expose:
    - "9093"
  networks:
    - big-data-network
```

Rysunek 4: Konfiguracja Kafki w docker compose

Tutaj podobnie jak wcześniej, najpierw image, potem nazwa i host. Dodatkową funkcją jest tutaj ”depends on”, kontener odpala się dopiero gdy jego dependencje działają. Z poziomu docker compose jesteśmy w stanie zdefiniować topics i maksymalny rozmiar dopuszczalnej wiadomości, a także porty na których słuchamy i jakich protokołów używamy.

### 5.3.3 Spark

```
spark-structured-streaming:
  image: docker.io/bitnami/spark:3
  container_name: spark_structured_streaming_master_bd
  hostname: spark_structured_streaming_master_bd
  user: root
  environment:
    - SPARK_MODE=master
    - SPARK_RPC_AUTHENTICATION_ENABLED=no
    - SPARK_RPC_ENCRYPTION_ENABLED=no
    - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
    - SPARK_SSL_ENABLED=no
  ports:
    - '8081:8080' # Spark web UI
  volumes:
    - ./src/spark:/home
  networks:
    - big-data-network
```

Rysunek 5: Konfiguracja Sparka w docker compose

Ta część była relatywnie prosta, to samo co wcześniej, w zmiennych ustawiamy tryb naszego kontenera i wyłączamy szyfrowanie i uwierzytelnianie użytkowników.

#### 5.3.4 HDFS

```
hdfs-namenode:
  image: gradiant/hdfs-namenode
  container_name: hdfs-namenode
  hostname: hdfs-namenode
  ports:
    - '50070:50070' # NameNode web UI
    - '8020:8020'   # HDFS RPC port
  networks:
    - big-data-network

hdfs-datanode1:
  image: gradiant/hdfs-datanode
  container_name: hdfs-datanode1
  hostname: hdfs-datanode1
  links:
    - hdfs-namenode
  environment:
    - 'CORE_CONF_fs_defaultFS=hdfs://hdfs-namenode:8020'
  networks:
    - big-data-network

hdfs-datanode2:
  image: gradiant/hdfs-datanode
  container_name: hdfs-datanode2
  hostname: hdfs-datanode2
  links:
    - hdfs-namenode
  environment:
    - 'CORE_CONF_fs_defaultFS=hdfs://hdfs-namenode:8020'
  networks:
    - big-data-network
```

Rysunek 6: Konfiguracja HDFS w docker compose

Tutaj, musieliszy skonfigurować trzy kontenery, jednak to była najprostsza usługa. Jest to zasługa tego, że wybraliśmy, bardzo przyjemny do pracy image który był stworzony pod podobny przykład użycia jak nasz. Wystarczyło tylko zdefiniować namenode i następnie podpiąć pod niego datanode.

### 5.3.5 Flask

```
flask-app:
  build:
    context: ./src/flask_application
    dockerfile: Dockerfile
  container_name: flask-app
  hostname: flask-app
  ports:
    - "5000:5000"
  networks:
    - big-data-network
  depends_on:
    - kafka
```

Rysunek 7: Konfiguracja Flaska w docker compose

Tutaj, konfiguracja kryje się w Dockerfile, który przygotowaliśmy jednak jest on bardzo standardowy. Następnie na podstawie tego pliku budowany jest image który służy nam jako szablon do kontenera.

### 5.3.6 Używane zasoby komputerowe

Aby sprawdzić jak obciążający jest jakiś komponent, postanowiliśmy, że wykorzystamy kolejną funkcjonalność dockera. Testy były przeprowadzone na komputerze stacjonarnym z procesorem Ryzen 7600. Jest to relatywnie nowy procesor z 6 rdzeniami i 12 wątkami.

Terminal						
NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	
flask-app	0.12%	239.5MiB / 30.47GiB	0.77%	16.3MB / 365kB	0B / 291kB	
kafka_bd	1.00%	376.2MiB / 30.47GiB	1.21%	10.1MB / 92.2kB	0B / 10.9MB	
hdfs-datanode1	0.38%	215.6MiB / 30.47GiB	0.69%	13.9kB / 24.9kB	0B / 344kB	
hdfs-datanode2	0.19%	211.6MiB / 30.47GiB	0.68%	13.8kB / 24.9kB	0B / 311kB	
spark_structured_streaming_master_bd	0.04%	260.3MiB / 30.47GiB	0.83%	8.58kB / 0B	0B / 303kB	
nifi_bd	3.83%	1.388GiB / 30.47GiB	4.56%	10.5MB / 10.2MB	0B / 1.29GB	
zookeeper_bd	0.04%	59.84MiB / 30.47GiB	0.19%	77.6kB / 106kB	0B / 545kB	
hdfs-namenode	0.19%	242.9MiB / 30.47GiB	0.78%	58.1kB / 11.1kB	0B / 1.24MB	

Rysunek 8: Zasoby komputerowe zużywane przez nasze rozwiązanie

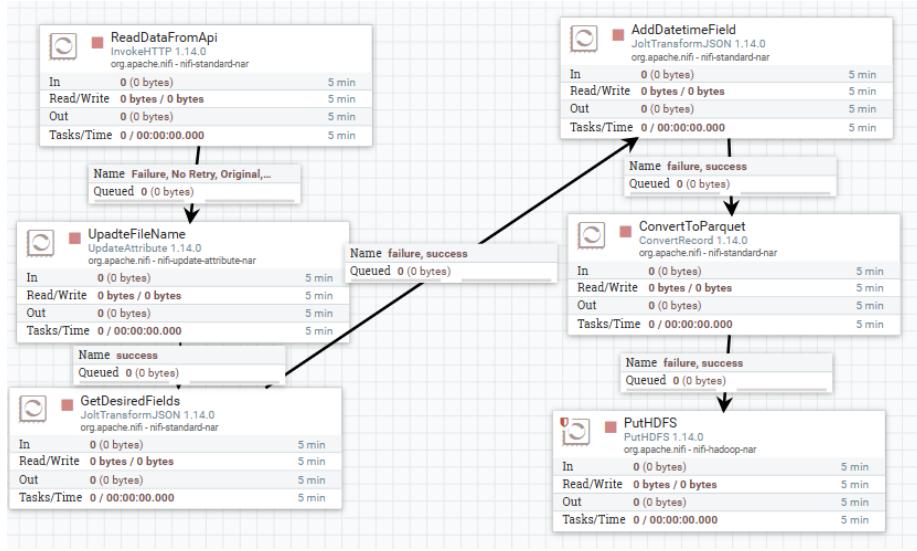
## 6 Opis przetwarzania i składowania danych

Przetwarzanie danych w naszym systemie najlepiej zobrazowanie jest w nagrany przez nas filmie, udostępnionym na platformie YouTube - [link](#). Pokazujemy tam funkcjonalności oraz komunikacje wszystkich komponentów, aby wszystko było widać najlepiej ustawić jakość na 4k albo 1444p.

### 6.1 Pierwotny pogodowy schemat w NiFi

Pierwotnie większą część transformacji wykonywaliśmy w NiFi, na rysunku (9) przedstawiamy schemat przetwarzania danych pogodowych za pomocą NiFi. Przestaliśmy go używać w celu zunifikowania naszych przepływów, jednak chcemy się pochwalić że umiemy to robić. W pierwszej kolejności, za pomocą InvokeHTTP, ładujemy surowe dane w postaci JSON z pogodowego API. Zapytania wykonujemy co 10 minut. Następnie zmieniamy nazwę pliku tak, aby wszystkie pliki zawierały wspólny prefiks, a następnie datę i czas z dokładnością do minut. Za pomocą dwóch procesorów JoltTransformJson wybieramy interesujące nas dane pogodowe, głównie odrzucając przewidywaną pogodę. Dodajemy także jedno dodatkowe pole, które wskazuje na datę oraz godzinę, wraz z minutami, przetwarzania. Jesteśmy świadomi, że obie te operacje można zrobić za pomocą jednego procesora, jednakże zdecydowaliśmy o rozbiciu ze względu na czytelność. Dodanie dwóch procesorów zamiast jednego niesie akceptowalny dla nas narzut czasowy, który jest minimalny. Ostatnim krokiem jest zmiana formatu na Parquet oraz zapisanie danych do HDFS. W wyniku otrzymujemy tabelę o następujących własnościach:

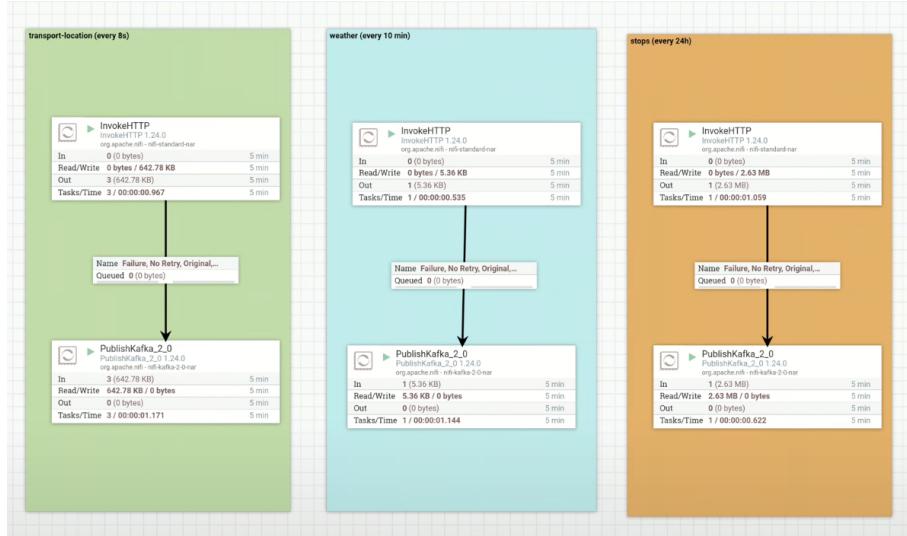
- **temperature** - temperatura z dokładnością do jednego miejsca po przecinku w stopniach Celsjusza. Mierzona jest ona na wysokości 2 metrów nad ziemią.
- **wind\_speed** - prędkość wiatru z dokładnością do jednego miejsca po przecinku w  $\frac{m}{s}$ . Mierzony jest on 10 metrów nad ziemią.
- **wind\_angle** - kąt wiania wiatru, gdzie  $180^\circ$  oznacza kierunek południowy.
- **wind\_dir** - kierunek wiatru oznaczony za pomocą odpowiednich kierunków geograficznych.
- **cloud\_cover** - procent nieba pokrytego chmurami.
- **processingTime** - moment w czasie otrzymania/przetworzenia danych.



Rysunek 9: Schemat NiFi dla pogody

## 6.2 Finalne przepływy danych

Trochę za radą prowadzącego wykład, finalnie potraktowaliśmy NiFi tylko jako opakowanie naszych źródeł danych tak aby w następnych warstwach naszego rozwiązania nie musieć się martwić, o to czy wszystko się dzieje poprawnie.



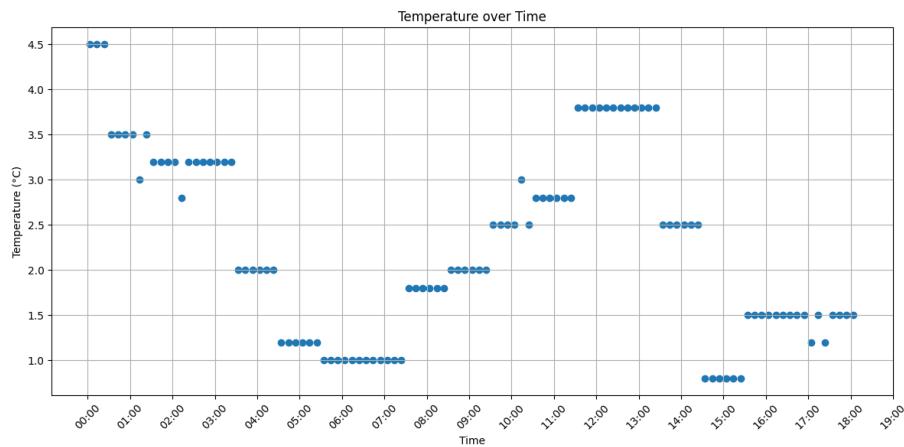
Rysunek 10: Konfiguracja Nifi

NiFi zawiera jeszcze dwa takie kafelki, jeden testowy i jeden do tras, jednak są one identyczne i chcieliśmy żeby było coś widać na rysunku. Periodycznie wykonywane jest zapytanie do api a następnie wpychane jest do Kafki. Kafka ma już wcześniej zdefiniowane topics. Spark structured streaming zaczyna z Kafki dane i je zamienia z JSONa który otrzymaliśmy na plik Parquet. Następnie zapisuje go do HDFS. Transformacje znajdują się w pliku *process\_location.py*

## 7 Opis Analizy danych w spark

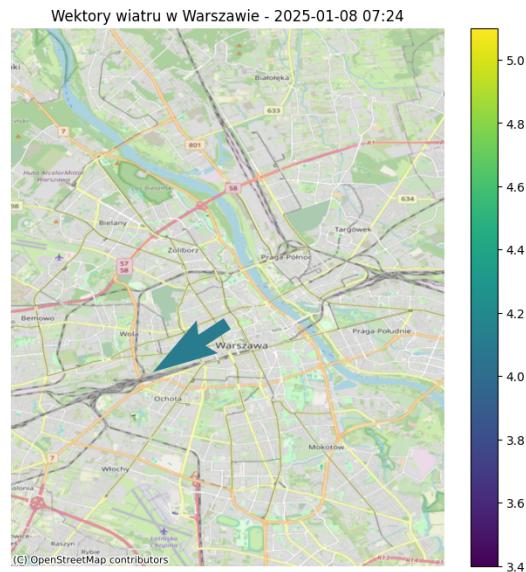
Na podstawie danych pogodowych stworzyliśmy dwie proste wizualizacje.

Pierwsza z wizualizacji widoczna na rysunku (11), przedstawia zmianę temperatury w czasie. Prezentujemy dane co 10 minut. Taka wizualizacja mogłaby zostać wykorzystywana do np. optymalizacji systemu grzania lub chłodzenia w komunikacji miejskiej.



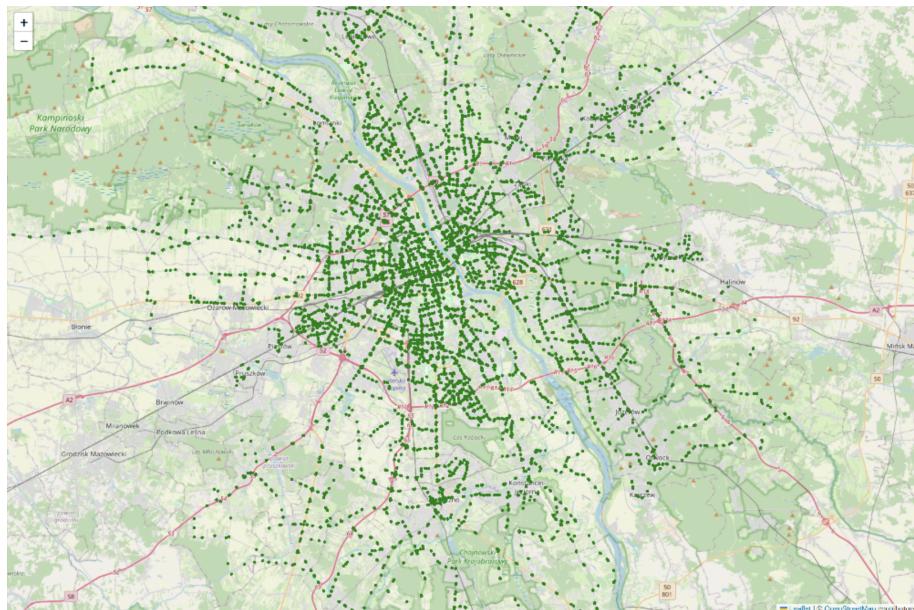
Rysunek 11: Wykres temperatury od czasu w dniu 08.01.2025

Drugą wizualizacją jest mapa (12), pokazująca moc oraz kierunek wiatru. Za pomocą suwaka można ustawać odpowiednią godzinę, z dokładnością do 10 minut. W raporcie przedstawiamy widok dla pojedynczego momentu w czasie. Strzałka wskazuje kierunek wiatru, a jej kolor wskazuje na jego prędkość. Taka wizualizacja, zrobiona w kontekście całego roku, mogłaby posłużyć do określenia struktury wiatrów przystankowych. Gdyby na przykład okazało się, że przez dużą część roku wiatr wieje z jednego kierunku, to można by było zabudować wiaty z przodu na odpowiednich przystankach, na których pasażerowie są najbardziej narażeni na wiatr.



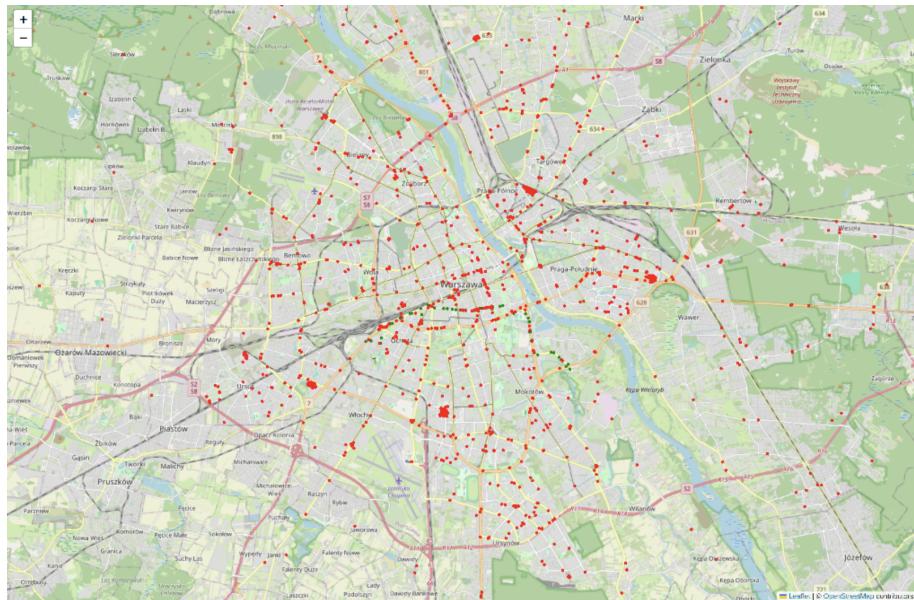
Rysunek 12: Mapa wiatru w dniu 08.01.2025

Innymi wizualizacjami, które wykonaliśmy jest wizualizacja wszystkich przystanków w Warszawie:



Rysunek 13: Wizualizacja wszystkich przystanków w Warszawie

Oraz wizualizacja ostatniej znanej lokalizacji autobusów:



Rysunek 14: Wizualizacja ostatniej znanej lokalizacji autobusów

## 8 Testy systemu

Ponownie odwołujemy się od naszego [filmu](#), testy opisane w tym rozdziale będą jedynie opisami tym co dzieje się w filmie. Tam jest to logiczną sekwencyjną całością i faktycznie widać, że to co zrobiliśmy działa, aby wszystko było widać najlepiej ustawić jakość na 4k albo 1444p.

## 9 Planowane funkcjonalności

### 9.1 Oglądanie lokalizacji w czasie rzeczywistym na mapce

Jest to funkcja, którą na ogół umożliwiają inne platformy zajmujące się tą tematyką. Aby móc to zrobić nie musimy przechowywać danych. Wystarczy tylko pobrać dane z API i przetworzyć je. Jednak, aby ujednolicić nasze rozwiązanie, najpierw będziemy je przetwarzać i zapisywać, a następnie odczytywać z bazy.

### 9.2 Możliwość wykonywania analiz

Szczególnie dla nas, studentów Inżynierii i Analizy Danych, ale pewnie i dla osób planujących trasy autobusów, ciekawa będzie funkcjonalność analizy opóźnień przy użyciu klastra, na którym postawiony jest Spark. Podczas gdy normalni użytkownicy będą mieli ograniczone możliwości interakcji z bazą danych, analitycy będą mieli dostęp do wszystkich gromadzonych danych. Pozwoli to na znajdywanie zależności pomiędzy zbieranymi parametrami.

Zamysł jest taki, aby po odkryciu interesujących zależności, na przykład pomiędzy pogodą, a opóźnieniami, stworzyć wizualizację i przygotować proces przetwarzania danych. Celem tego jest, aby zwykli użytkownicy mogli śledzić i obserwować dane zależności w czasie rzeczywistym (jeżeli to możliwe) a jeżeli nie to z akceptowalnym opóźnieniem, na przykład godzinowym albo dziennym. Przykładem takiego procesu może być analiza średniego opóźnienia komunikacji w ciągu dnia, które następnie po ustaleniu czy jest to wystarczająco ciekawe i odkrywcze, będzie udostępnione dla użytkowników i podliczane pod koniec dnia.