Real-time visualization of city public transport provider data and prediction of future trends

Jurić, Kristijan

UNIVERSITY OF ZAGREB

FACULTY OF TRANSPORT AND TRAFFIC SCIENCES

Kristijan Juric

REAL-TIME VISUALIZATION OF CITY PUBLIC TRANSPORT PROVIDER DATA AND PREDICTION OF FUTURE TRENDS

Zagreb, 2024.

UNIVERSITY OF ZAGREB
FACULTY OF TRANSPORT AND TRAFFIC SCIENCES
MASTER THESIS COMMITTEE

Zagreb, 22 April 2024

MASTER THESIS ASSIGNMENT No. 7674

Student:	**Kristijan Jurić (0128056654)**
Study:	Intelligent Transportation Systems and Logistics
Course:	Intelligent Transportation Systems

Title: **Real-time visualization of city public transport provider data and prediction of future trends**

Description:

The aim of the thesis is to create a web application for displaying real-time data obtained from the real-time data flow of public transport provider. First, it is necessary to study the format and standards of real-time data and connect the web application to it in order to display information in real-time. Within the web application, it is necessary to load a publicly available map and display the current positions of public city transport vehicles. Also, it is necessary to design a relational database to store the mentioned data. In parallel with the retrieval of real-time data for display in a web application, it is necessary to continuously store the data in the database. It is necessary to add interactive elements to the web application to display not only real-time data but also historical data saved in the database, such as the vehicle's route from the previous day. Also, it is necessary to analyze the saved data and develop a simple machine model that will predict future trends, such as the travel time of a particular transport.

Mentor:

Tomislav Erdelić

Tomislav Erdelić, PhD

Committee Chair:

Sveučilište u Zagrebu

Fakultet prometnih znanosti

Real-time visualization of city public transport provider data and prediction of future trends

Vizualizacija podataka pružatelja javnog gradskog prijevoza u stvarnom vremenu i predviđanje budućih trendova

Mentor: dr. sc. Tomislav Erdelić

Student: Kristijan Juric
JMBAG: 0128056654

Zagreb, September 2024.

Real-time visualization of city public transport provider data and prediction of future trends

Abstract:

In the highly changing area of Intelligent Transport Systems (ITS), real-time information is very important in making the public transportation system be more effective and reliable. This work discusses a real-time visualization application for Zagreb public transport system using General Transit Feed Specification (GTFS) data provided by Zagrebački Električni Tramvaj (ZET), the city's public transportation authority. The application is created to handle and show static GTFS data as well as live GTFS data allowing users to follow present locations of vehicles on stop arrival and schedules in real-time. There is also an inclusion of predictive modelling techniques to forecast future trends in public transport. The results highlight the possible integration of ITS with predictive analytics to enhance public transport systems and user experience in fast-expanding urban areas and to also contribute to more efficient urban mobility solutions.

Keywords: GTFS, real-time data, public transport, predictive modeling, ZET, urban planning, ITS

Vizualizacija podataka pružatelja javnog gradskog prijevoza u stvarnom vremenu i predviđanje budućih trendova

Sažetak:

U brzo mjenjajućem području inteligentnih transportnih sustava, informacije u stvarnom vremenu su izuzetno važne za poboljšanje same učinkovitosti i pouzdanosti javnog prijevoza. U ovom radu se raspravlja o aplikaciji za vizualizaciju javnog prijevoza grada Zagreba u stvarnom vremenu koristeći General Transit Feed Specification (GTFS) podatke koje osigurava ZET (Zagrebački Električni Tramvaj), gradska uprava za javni prijevoz. Aplikacija je stvorena za rukovanje i prikaz statičkih GTFS podataka kao i stvarno-vremenskih GTFS podataka koji korisnicima omogućuju praćenje trenutnih lokacija vozila pri dolasku na stanicu i rasporeda u stvarnom vremenu. Tu se također uključuju i tehnike prediktivnog modeliranja za predviđanje budućih trendova u javnom prijevozu. Rezultati naglašavaju moguću integraciju ITS-a sa prediktivnom analitikom kako bi se poboljšao javni prijevoz i korisničko iskustvo u brzo širećim urbanim područjima i isto kako bi se pridonjelo učinkovitijima rješenjima za urbanu mobilnost.

Ključne riječi: GTFS, informacije u stvarnom vremenu, javni prijevoz, prediktivno modeliranje, ZET, urbano planiranje, ITS

CONTENTS

1. Introduction

1.1. Background and Motivation

Today in densely populated cities like Zagreb, Croatia, public transport systems are the main aspect of its urban mobility. With the overall increase of transportation infrastructure, the demands grow as the city grows. There is a need for efficient, reliable and real-time data-driven solutions for managing public transportation and its operations. In this regard, it can be said that the General Transit Feed Specification (GTFS) data has become extremely important to provide a standard on how public transport schedules, routes and of course real-time locations of vehicles are managed and obtained.

The purpose of this thesis comes from the need to further develop the public transportation services in the city of Zagreb which can be obtained by integrating GTFS data which is provided by Zagrebački Električni Tramvaj (ZET) into an informational framework. Focus is put on developing a real-time visualization application which will be used for tracking public transport vehicles when on station arrival, analyzing historical data and predicting future trends using machine learning. By using technologies like Python for data handling and processing, MySQL for data storage and Dash framework for user interface development, the goal is to provide a not only useful but also comprehensive tool that could help the commuter find his public transport vehicle or anticipate a delay in a specific timeframe. Additionally, this tool can help city urban and transport planners make better decisions in planning and optimizing the current state of the public transport system.

Real-time data processing together with predictive modelling with standard and new machine learning techniques represents a big improvement in regards to advancing Intelligent Transportation Systems (ITS). Anticipating and mitigating potential transportation issues by using tools which predict future trends like potential delays can play a significant role in contributing to more efficient, sustainable and reliable urban transport. This work tends to connect data availability and overall informational insights in the domain of urban mobility.

1.2. Thesis Objectives

The goal of the master thesis called "Real-time visualization of city public transport provider data and prediction of future trends" is to design and develop a real-time visualization application integrating ZET's GTFS public transport data, where a connection between the static GTFS data and live updates is created. The application should monitor vehicles' current positions when on station arrival while providing access to a database of historical data for trend prediction or deeper analysis. Trend prediction is one of the aspects that could advance the service in providing better information and improving passenger satisfaction.

The specific objectives are like in the following:

- **To analyze and create an overview of the GTFS data**: this involves creating an overview of the history, implementation and challenges of the GTFS data.
- **To design a real-time data visualization application:** the application design should utilize GTFS data to deliver real-time updates regarding vehicle locations at station arrival, delays, and route visualization with additional functions for data evaluation and debugging.
- **To develop a relational database for GTFS data storage**: this database should store both real-time updates with static data referencing. With that, historical data should always be accessible for detailed analysis and trend prediction, which is essential for the application's functionality.
- **To implement predictive models**: this thesis will explore various machine learning models which are popular for capturing non-linear relationships between data and time-series data like Prophet etc.
- **To evaluate the effectiveness of prediction models**: testing will be conducted where based on the current stored frame of data an evaluation and comparison using a specific prediction model will be performed.

1.3. Thesis Structure

Following the introduction, Chapter 2 provides an overview of the literature in regards to GTFS which focuses on its data format, its key components and its application in public transportation systems. Particular focus is being forwarded towards on how ZET provided this data in Zagreb. Chapter 3 describes the overall architecture of the application and how it was developed. Proceeding, Chapter 4 describes the predictive models used which are applied to the historical transport data provided from the internal data frame. Moreover, this is then used to forecast future trends with an evaluation of different machine learning models. Chapter 5 presents and discusses the implementation outcomes, including real-time visualization, trend analysis and a comparison of results from chosen predictive models. Chapter 6 summarizes the key findings and suggests areas for future work on this topic.

2. Transit Feed Standards

In big cities, most people use private vehicles as the main means of transportation, which is not sustainable. In this case, public transportation plays an important role in providing mobility to citizens, especially those with disabilities and no means to private transport. In Zagreb, public transport varies from buses, trams, and cable cars, and it is managed predominantly by the city's public transportation company ZET. Over the years, the transit system has significantly improved the operational efficiency and passenger experience since it is directed mostly towards integrating technological advancements in public transportation.

Among these advancements are transit feed standards like GTFS, which provides static and real-time data for public transit systems, and European standards such as SIRI and NeTEx, which facilitate real-time information exchange and static data management for multimodal transport systems. This chapter explores the key transit feed standards, including GTFS, SIRI, and NeTEx, and their applications in public transportation systems, focusing on specifically integrating GTFS into Zagreb's public transportation system.

2.1. General Transit Feed Specification

2.1.1. Definition and History

GTFS is a data format standard that public transportation agencies use to publish their transit data. While operating, buses, trams, trains, and ferries provide this type of data to the transport agencies and agencies further to the public. Developed by Google, GTFS facilitates the presentation and collection of public transport data in applications like Google Maps and any other app that can utilize this type of data. Transit operators share their schedules, stations, details, departure and arrival times and also fare information trough GTFS standard data. With this, passengers can plan their journeys on public transportation and can access provided information regarding where and when services are available. GTFS data also encompasses travel information which includes terminals, stations, routes, schedules, etc. and is accessible to all users who rely on mobile apps or web services for trip planning and monitoring [1].

The origin and development of GTFS started in Portland USA in the summer of 2005 where Bibiana McHugh when traveling abroad was frustrated because she couldn't access a geographic mapping website like MapQuest. Because of this she could not plan her trip as easily as she could if she was in a car. After returning to USA, she started contacting companies like MapQuest, Yahoo, and Google making proposals that inclusion of transit data should be available in their mapping services. After sending proposals with TriMetMAX as her partner, Google was the only company that responded positively. Chris Harrelson, a software engineer in Google integrated TriMetMAX's transit data into Google Maps where the first instance of the Google transit trip planner spurred. TriMetMAX has prepared its data in a format compatible to Google Maps but also they took a proactive approach to managing its transit data, pushing Google to refine the implementation of transit data within Google Maps. With data refinement came the GTFS specification [2].

On December 7, 2005 the Google Transit Trip Planner was introduced which was primarily utilized by TriMetMAX. Making these services available and accessible on Google Maps triggered a response where cities like Eugene, Honolulu, Pittsburgh, Seattle and Tampa started adopting the specification. This expansion demonstrated the growing demand for real-time availability of transit information where till today Google Maps collaborates with more than 100 transit operators in the U.S and over 400 worldwide [2].

In the next section 2.1.2 the components that make the GTFS will be described. Each of the key data files will be examined and explained to understand how they connect to each other and contribute to making a comprehensive transit data set.

2.1.2. GTFS Components

GTFS data is split into two main components which serve different purposes but are interconnected with relationships from the systematic perspective which makes them complementary to each other.

Real-Time Data

One of the most essential components of GTFS is the real-time data since it provides a continuous flow of information about the status of transit services. Updates on vehicle positions, delays, service disruptions, arrival and departure times are parsed to the applications of transit users who are then informed about the current transit situation and availability [3].

When passengers have access to reliable and timely information they are more likely to choose public transportation over private vehicles in this regard. This reduces traffic congestion and lowers carbon emissions since there are fewer cars on the road. In this case, it will facilitate real-time data processing, making public transit a more attractive option for city commuters [1].

The primary entities in GTFS real-time data include trip updates, vehicle positions, service alerts and trip modifications which inform the commuter about the transit's overall status and will be described in the following:

- **Trip Updates**: are changes in the scheduled transit timetable. They do provide an arrival and departure time in addition to other information but in some cases can be also inaccurate or not provided. They are crucial for handling situations that require dynamicity in transit operations like delays, added trips, cancellations and such. Trip updates make sure that the passenger or potential commuter is informed on time of any changes or the nature of a specific trip [4].
- **Vehicle Positions**: provide real-time data in the shape of latitude and longitude of the current location of the vehicle, which helps in tracking its position on the transit network. This data can also include additional information such as the vehicle's speed and odometer readings [4].
- **Service Alerts**: provide information about disruptions that are affecting the transit network in that moment in time. These disruptions can be for example line suspensions, station closures or even network-wide issues which can be a response from a problem in the transit network. These alerts usually consist of some textual description in which the disruption is described and additionally could contain a URL for additional information [4].
- **Trip Modifications**: inform about changes of active or planned routes. This can create situations like providing new route shapes or even indicating temporary stops along a detour. Also in some regards trip modifications can occur when modified schedules and detours are applied because of upcoming expected or unforeseen circumstance like road construction or public events [4].

Static Data

GTFS static data are categorized as a group of text files with information which explain the operations of a transit system. Compared to real-time data which is updated in specific short time windows, static data provides the schedule information and details about the long-term operations of a transit system. There is usually a larger number of GTFS static files but a few of them are extremely important like the*"agency.txt"*, *"stops.txt"*, *"routes.txt"*, *"trips.txt"*, *"stop_times.txt"* and *"calendar.txt"*. All the files above have multiple fields within themselves, which have separate meanings in this context [2]. In the following, all of the most important data files are described:

- **"agency.txt"**: this file contains information relevant to the transit agency, including the agency's name, website, and contact information. It can also contain multiple agencies, in which case the agency data insertion is mandatory, specifying the data source.

The file contains the following data rows: *"agency_id"*, *"agency_name"*, *"agency_url"*, *"agency_timezone"*, *"agency_lang"*, *"agency_phone"*, and *"agency_fare_url"* [4, 2].

- **"stops.txt"**: this file describes all the stops in the transit system for the update cycle. It shows where passengers can board and alight from transit vehicles. For example, fields like *"stop_id"* and *"stop_name"* identify each specific stop and provide additional information to aid commuters in trip planning. Fields like *"stop_lat"* and *"stop_long"* provide the stop location in the form of geographic coordinates. The file contains the following data rows: *"stop_id"*, *"stop_code"*, *"stop_name"*, *"stop_desc"*, *"stop_lat"*, *"stop_long"*, *"stop_url"*, *"location_type"*, *"stop_timezone"*, and *"wheelchair_boarding"* [4, 2].
- **"routes.txt"**: this file represents all the routes currently used by the transit network. Each route has its unique identifier *"route_id"*, along with other data rows such as *"route_short_name"*, *"route_type"*, etc. For example, *"route_type"* represents the mode of transport, which, in the case of public transport, can be a bus, tram, or even a ferry. This data helps transportation planners understand which lines are available for that timeframe within the system. The file contains the following data rows: *"route_id"*, *"agency_id"*, *"route_short_name"*, *"route_long_name"*, *"route_desc"*, *"route_type"*, *"route_url"*, *"route_color"*, and *"route_text_color"* [4, 2].
- **"trips.txt"**: This file contains data where every individual trip is connected to its route. Each trip has its own unique *"trip_id"* which could also be connected to a *"service_id"* that could indicate the schedule, meaning if the trips are done over weekdays, weekends, etc. Also worth mentioning is *"shape_id"* which provides the path that the trip follows. Trips in most cases in the dataset have correlated connections to their sequences of stops that could help in operational planning [2]. The *"direction_id"* is also important as it indicates the direction in which the trip is being executed. Typically, *"direction_id"* values are binary, where ‘0‘ represents an inbound trip (heading towards the main destination or city centre), and ‘1‘ represents an outbound trip (heading away from the main destination or city centre) [4]. The file contains the following data rows: *"route_id"*, *"service_id"*, *"trip_id"*, *"trip_headsign"*, *"trip_short_name"*, *"direction_id"*, *"shape_id"*, *"wheelchair_accessible"*, and *"bikes_allowed"* [2].
- **"stop_times.txt"**: this file contains the schedule for every stop connected to its *"trip_id"*. This means that each specific trip has a timing for its stops in sequence. Other data rows included are *"arrival_time"*, *"departure_time"*, and *"stop_id"*, which links the trips to their corresponding stops and times when the transit vehicle is expected to arrive or depart. Without this file, the schedules in transit could not be created, nor could any specific trip details be sent to the passengers through applications. The file contains the following data rows: *"trip_id"*, *"arrival_time"*, *"departure_time"*, *"stop_id"*,

"stop_sequence", and *"timepoint"* [2].

- **"calendar.txt"**: this file contains the overall calendar of the transit services. With this calendar, a commuter can specify on which day a service is operating. To understand the schedule, this file is important since the start and end dates of a specific service are stated there. Key data rows in this file include *"service_id"* which is interconnected with *"trips.txt"*. There are also other fields like *"monday"*, *"tuesday"*, *"wednesday"*, etc., whose values can be 1 or 0. This shows whether the service is running on specific days. The file contains the following rows: *"service_id"*, *"monday"*, *"tuesday"*, *"wednesday"*, *"thursday"*, *"friday"*, *"saturday"*, *"sunday"*, *"start_date"*, *"end_date"*.
- **"fare_attributes.txt"**: this file describes the fare payment structure of a transit system, with data rows like *"fare_id"* connected to the value of *"price"* for the cost of fare, for example. All this data is interconnected since the price is linked to the *"currency_type"* and extends to *"payment_method"*, which describes how the fare could be paid. This data is always linked with other static data text files, such as *"agency_id"*, which links the fares to the specific agency. This helps commuters plan their trip from a financial standpoint. The file contains the following data: *"fare_id"*, *"price"*, *"currency_type"*, *"payment_method"*, *"transfers"*, *"agency_id"*, *"transfer_duration"* [4].

From the previous static and real-time data descriptions it can be seen that every part of the data is interconnected with each other by their corresponding id-s which makes it easy to find the information needed when integrating in specific applications. In the next chapter 2.2 will focus on how this interconnected data is differentiated from agency to agency but mostly focusing on ZET and how it would benefit the overall public transport in Zagreb.

2.2. GTFS Data Sources for Zagreb

2.2.1. Public Transport and GTFS in Zagreb

Zagreb's public transportation system consists of various forms of transportation, including trams, buses, railways, and cable cars. These different traffic systems work together to create extensive coverage over the city and connect the outer parts of Zagreb with Zagreb's centre. However, in this thesis, the focus will be mostly on the tram and bus networks, which are in the GTFS framework in regard to data availability.

One of the most fundamental aspects of Zagreb city's urban planning is its public transportation system which was outlined in the General Urban Plan (GUP). What GUP aims to organize and plan better is the point on how the city is using the city's space and in which way to expand the tram network and develop a light rail system to enhance connectivity throughout the city. The tram network in Zagreb operates on 148km of tracks across 15 daily and 4 night lines which has 256 tram stops that are intentionally placed so that the walking distances in areas where there are hospitals, schools and public institutions are adequately minimized. Tram lines in Zagreb are designed in the manner of diametral or tangential routes, which for Zagreb's situation do help connect different parts of the city as efficiently as possible [5].

If GTFS is fully utilized in this case, this could lead to a better GUP in regards to tram networks where urban planners could make better decision which is supported by real-time data gathered from GTFS databases.

With the trams, in the overall public transport system there is also an inclusion of the bus network which has 133 daily and 4 night lines, which in this regard play an important role in connecting the city's outline areas and the central parts of Zagreb as it can be seen from figure 2.1. There are 2103 bus stops on the transit network and from that, in Zagreb's area, there are 1614 of them. On weekdays during peak traffic times, there are around 303 buses; on Saturdays, 185 buses; and on Sundays an public holidays, 123 buses are actively in traffic.The bus routes in Zagreb are usually designed to be interconnected with the tram lines which minimizes the need for transfers and also ensures that if a passenger is doing a multimodal travel it's an easy-going experience [5].

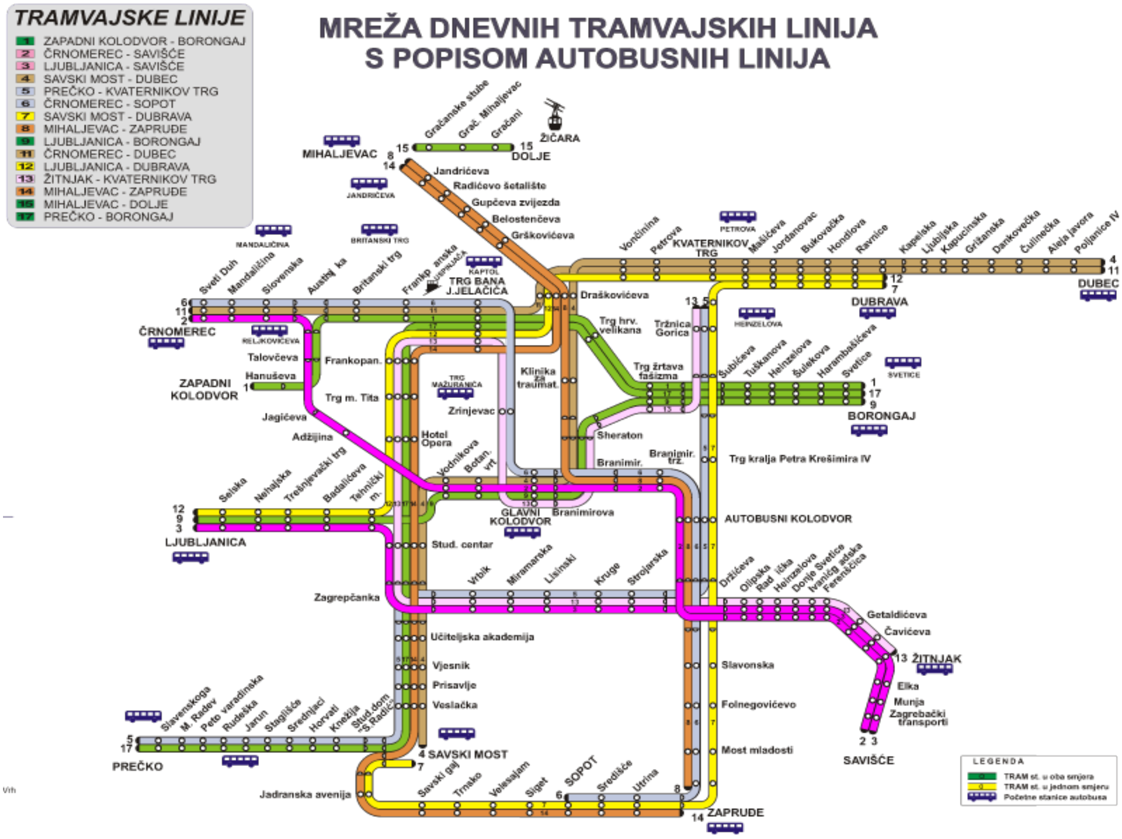


Figure 2.1: View of Zagreb's transit network [6]

According to a study conducted by the Faculty of Transport and Traffic Sciences, public transportation in Zagreb increased from 52% in 2009 to 61% in 2011 which highlights that people are putting more trust into the public transportation system overall [7]. Additionally, the bus network had some improvements over the years and between 2013 and 2017 there was a steady increase in the number of passengers which were transported by buses. The year 2017 was highest in marking in regards to ridership since the service coverage was enhanced and it made it more reliable [7].

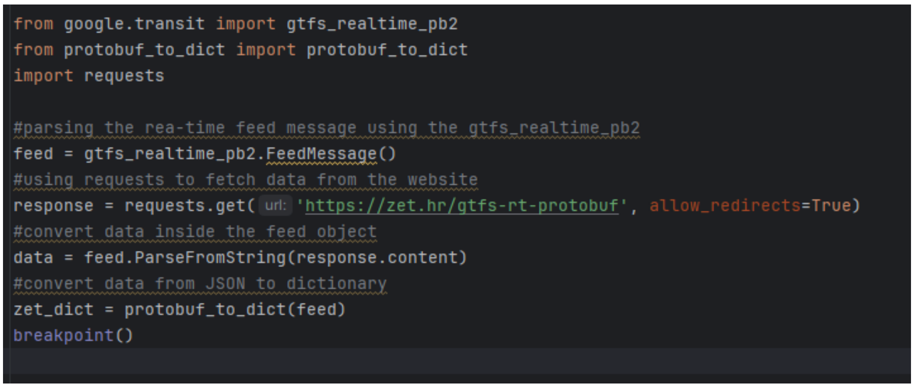
This kind of improvement shows the city's efforts to make a change and make public transportation more accessible to people. With that it can be said that GTFS could also enhance the passenger experience for bus commuters where it would make it easier for them to plan their multi-modal journeys, ensuring more accurate transfer times and making it more reliable which would make them choose public transport rather than private vehicles.

2.2.2. ZET GTFS Data Collection

For a collection of ZET GTFS data, the application in this thesis utilizes Python as the primary programming language with the addition of MySQL databases for data storage and other libraries which will be more explained in the next section 3. The real-time data feed can be accessed from the ZET server at the URL: https://zet.hr/gtfs-rt-protobuf.

In most occasions, when interacting with the data within the protocol buffer retrieved, the refresh rate of the data inside is perhaps 10-15 seconds. This is important since pinpointing the exact refresh rate is beneficial to acquiring better overall data quality in the end.

Data is retrieved by using libraries like *"gtfs_realtime_pb2"* for handling GTFS real-time feeds, *"protobuf_to_dict"* which converts protocol buffer messages to python dictionaries and *"requests"* a Python library that is used to make HTTP requests to get the trip updates. The script that is used in this case is visible on figure 2.2 which shows the aforementioned steps.

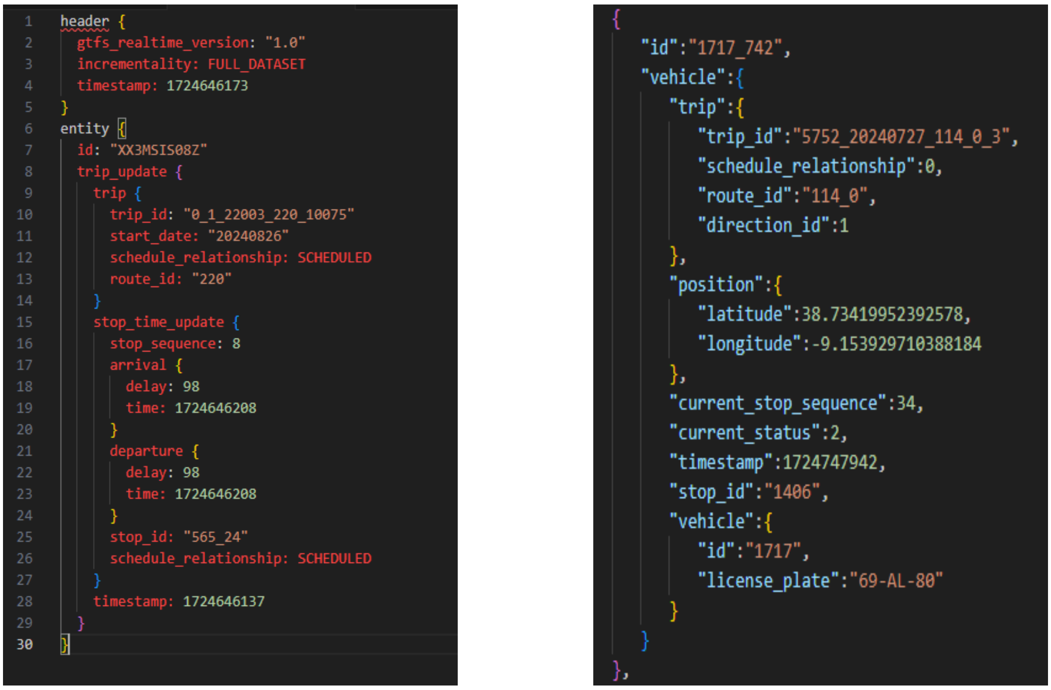


```
from google.transit import gtfs_realtime_pb2
from protobuf_to_dict import protobuf_to_dict
import requests

#parsing the rea-time feed message using the gtfs_realtime_pb2
feed = gtfs_realtime_pb2.FeedMessage()
#using requests to fetch data from the website
response = requests.get( url: 'https://zet.hr/gtfs-rt-protobuf', allow_redirects=True)
#convert data inside the feed object
data = feed.ParseFromString(response.content)
#convert data from JSON to dictionary
zet_dict = protobuf_to_dict(feed)
breakpoint()
```

Figure 2.2: Python script for ZET GTFS real-time trip updates retrieval

Afterwards, this type of python dictionary data can be fully converted to be integrated with data handling libraries like *"pandas"* for better data manipulation.

The data extracted from the ZET GTFS feed protocol buffer can be seen in the following figure 2.3(a) where it is visible ZET GTFS follows the standard specified by Google and includes most of the important data rows regarding its trip update real-time feed. Unfortunately, when comparing the batch from ZET GTFS feed to feeds from other countries, in the ZET GTFS feed batch there are no vehicle positions provided.



```
header {
  gtfs_realtime_version: "1.0"
  incrementality: FULL_DATASET
  timestamp: 1724646173
}
entity {
  id: "XX3MSIS08Z"
  trip_update {
    trip {
      trip_id: "0_1_22003_220_10075"
      start_date: "20240826"
      schedule_relationship: SCHEDULED
      route_id: "220"
    }
    stop_time_update {
      stop_sequence: 8
      arrival {
        delay: 98
        time: 1724646208
      }
      departure {
        delay: 98
        time: 1724646208
      }
      stop_id: "565_24"
      schedule_relationship: SCHEDULED
    }
    timestamp: 1724646137
  }
}
```

((**a**)) Zagreb’s ZET protocol buffer Trip Updates feed

```
{
    "id":"1717_742",
    "vehicle":{
        "trip":{
            "trip_id":"5752_20240727_114_0_3",
            "schedule_relationship":0,
            "route_id":"114_0",
            "direction_id":1
        },
        "position":{
            "latitude":38.73419952392578,
            "longitude":-9.153929710388184
        },
        "current_stop_sequence":34,
        "current_status":2,
        "timestamp":1724747942,
        "stop_id":"1406",
        "vehicle":{
            "id":"1717",
            "license_plate":"69-AL-80"
        }
    }
},
```

((**b**)) Portugal’s Carris Metropolitana GTFS real-time vehicle positions feed

Figure 2.3: Difference between trip update and vehicle position feed from different countries

The unavailability of vehicle positions occurred in 2010. when two students from FER (Faculty of Electrical Engineering and Computing) in Zagreb created an application "ZET Info" which utilized all of the ZET GTFS data and could show where every transit vehicle was at that moment in time. At that time City of Zagreb did not like the way the data was used then they quickly shut it down by blocking access to the data and mentioning the reason that this could potentially be used for terrorist purposes [8], [9].

After some years City of Zagreb did release the ZET GTFS data to the public but still the exclusion of vehicle position updates persist. It is notable to take a look at other public transit systems like Portugal’s Carris Metropolitana which offers vehicle positions for their buses. The Carris Metropolitana example of the real-time feed for vehicle positions is shown in figure 2.3(b) where it is clearly visible how the trip_id and positions of the vehicle with its longitude and latitude are categorized under the same vehicle entity with a unique id number.

Looking at the differences in transparency and availability of data, it is valid that security concerns should be upheld, but there should also be a balance between the openness of data and security measures. If open data policies are embraced fully, like those seen in Portugal, it could lead to significant benefits for Zagreb’s transit users.

By using the following techniques shown in Algorithm 1, an approach for retrieving, processing and merging real-time and static GTFS data from the ZET transit system is outlined. It uses the *"gtfs_realtime_pb2"* to parse real-time data, which is then converted to a dictionary format for easier handling. Once the real-time data is extracted, it is merged with the static information (such as stops, routes and trips) from the CSV files to create a unified dataset. Additionally, *"protobuf_to_dict"* is used to convert the feed from protobuf to a dictionary, and *"requests"* is used to fetch the real-time data from the ZET server providing it, which in the end creates the *"zet_df"* that is used for processing, storing and evaluation.

Algorithm 1 GTFS real-time data and static data retrieval and merging

Input: GTFS real-time feed URL, CSV file paths for stops, routes, trips

Initalize: GeoDataFrame with ZET GTFS transit data

```
feed ← gtfs_realtime_pb2.FeedMessage()                    # Initialize feed message
response ← GET request to GTFS URL                        # Fetch real-time data
feed.ParseFromString(response.content)                    # Parse response content into feed
zet_dict ← protobuf_to_dict(feed)                         # Convert feed to dictionary

stops ← Read CSV file stops.txt                           # Load static data for stops
routes ← Read CSV file routes.txt                         # Load static data for routes
trips ← Read CSV file trips.txt                           # Load static data for trips

zet_df ← Flatten zet_dict to DataFrame                    # Convert dictionary to DataFrame
timestamp ← Extract and format timestamp from zet_dict    # Process timestamp

Select and rename columns in zet_df for clarity           # Prepare DataFrame
Merge zet_df with stops on stop_id                        # Add stop data to DataFrame
Merge zet_df with routes on route_id                      # Add route data to DataFrame
Merge zet_df with trips using trip_id                     # Add trip data to DataFrame

Convert zet_df to GeoDataFrame using stop_lat and stop_lon    # Convert to GeoDataFrame
get zet_df dataframe
```

The acquired trip updates *"zet_df"* pandas data frame from Zagreb' ZET real-time feed can be categorized like in table 2.1 where the types of information it provides are highlighted.

Table 2.1 ZET GTFS trip updates data rows

Data Row	Description	Example
id	Unique identifier that distinguishes each vehicle in the system.	XWPGUI5OGN
trip_id	A unique identifier is assigned to each individual journey, which is utilized to monitor the trip from beginning to end.	0_1_12602_126_10055
route_id	Identifier that represents the specific route being used by the vehicle.	126
tp_timestamp	The Unix time format timestamp shows the most recent update time of the trip.	1724575783
stop_sequence	The order of stops that the vehicle follows in its journey are arranged sequentially.	1
arrival_time	The scheduled arrival time of the vehicle at the stop, presented in Unix time format.	1724757799
departure_time	The anticipated departure time of the vehicle from the stop, expressed in Unix time format.	1724757832
stop_id	Distinct code assigned to each stop, indicating the location where the vehicle will stop during its journey.	99_52
delay	The vehicle's delay time in seconds shows how behind schedule it is.	-237(early), +237(late)
stop_name	The stop's name is used for passengers and systems to identify the location.	Črnomerec
stop_lat	The stop's latitude coordinate is utilized for mapping and navigation purposes.	45.81580
stop_lon	The stop's longitude coordinate is utilized for mapping and navigation purposes.	15.93413
route_long_name	A detailed descriptor of the route, often indicating the start and end points of the same route.	Črnom.-G.K.-Kvraki
geometry	The location of the vehicle on stop arrival for tracking in real-time when vehicle arrives on a stop.	POINT (15.934131 45.8158)

2.3. SIRI, NeTEx and other transit standards in public transportation

2.3.1. SIRI (Service Interface for Real-time Information)

SIRI is a European standard which is widely used and it is designed to provide real-time public transportation information, including vehicle positions, trip updates, and service disruptions. From the GTFS side, which focuses primarily on North American transit systems, SIRI is particularly present in Europe, where it is used for real-time data for buses, trams and trains [10] [11].

If SIRI is integrated with the existing system while providing more detailed real-time information, it could offer real-time monitoring of vehicles, service interruptions, and even seat availability, enhancing the passenger experience by offering timely and accurate information. With the adoption of SIRI, for example, big cities like London and Paris have optimized their transit operations, ensuring that passengers can receive up-to-date alerts and plan their trips effectively [11].

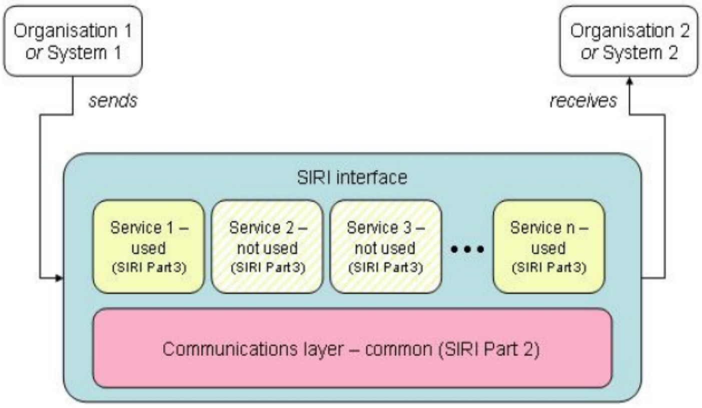

Organisation 1 *or* System 1
sends
Organisation 2 *or* System 2
receives
SIRI interface
Service 1 – used (SIRI Part3)
Service 2 – not used (SIRI Part3)
Service 3 – not used (SIRI Part3)
•••
Service n – used (SIRI Part3)
Communications layer – common (SIRI Part 2)

Figure 2.4: SIRI Communication Structure

On figure 2.4 it can be seen on how SIRI manages the overall communication of real-time public transportation data between two organizations in layers.

The SIRI interface is built around a communication layer that manages the processes involved in data requests and responses, ensuring uniformity in message referencing, error handling, and data flow management across all services[11]. SIRI can be used selectively based on specific system requirements, allowing organizations to utilize only the necessary services such as Service 1 and Service n, while omitting unnecessary ones. This adaptability enables efficient utilization of resources like bandwidth and processing power.

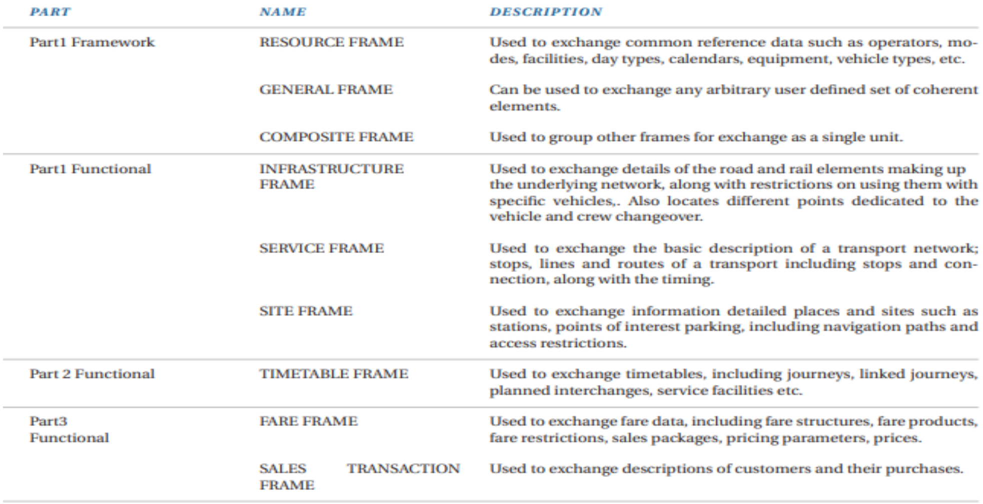
Two primary communication patterns supported by SIRI are Request/Response, in which one system requests data from another and receives a response, and Publish/Subscribe, a dynamic model that enables systems to subscribe to specific types of data and receive updates as they become available [11].

2.3.2. NeTEx (Network Timetable Exchange)

The NeTEx standard is a product of European development intended to simplify sharing static public transportation data. It encompasses schedules, fare details, and intricate network configurations, including routes, stops, and operator specifics [12].

NeTEx is especially well-suited for multimodal transportation systems, where buses, trains, and other transit modes operate cohesively in an interconnected network. Offering a more comprehensive dataset than GTFS, NeTEx is well-suited for managing intricate transit operations, such as adaptable scheduling or long-distance travel with transfers. NeTEx can potentially enhance urban planning and transportation system design in cities like Zagreb by supplying planners with data to optimize routes, timetables, and fare structures [12].

Figure 2.5 shows an overall NeTEx data structure divided into different parts and frames to manage various public transport data exchange aspects. They are divided into sections like framework and three types of functional data, which are separated based on the information provided to the user.



PART	NAME	DESCRIPTION
Part1 Framework	RESOURCE FRAME	Used to exchange common reference data such as operators, modes, facilities, day types, calendars, equipment, vehicle types, etc.
	GENERAL FRAME	Can be used to exchange any arbitrary user defined set of coherent elements.
	COMPOSITE FRAME	Used to group other frames for exchange as a single unit.
Part1 Functional	INFRASTRUCTURE FRAME	Used to exchange details of the road and rail elements making up the underlying network, along with restrictions on using them with specific vehicles,. Also locates different points dedicated to the vehicle and crew changeover.
	SERVICE FRAME	Used to exchange the basic description of a transport network; stops, lines and routes of a transport including stops and connection, along with the timing.
	SITE FRAME	Used to exchange information detailed places and sites such as stations, points of interest parking, including navigation paths and access restrictions.
Part 2 Functional	TIMETABLE FRAME	Used to exchange timetables, including journeys, linked journeys, planned interchanges, service facilities etc.
Part3 Functional	FARE FRAME	Used to exchange fare data, including fare structures, fare products, fare restrictions, sales packages, pricing parameters, prices.
	SALES TRANSACTION FRAME	Used to exchange descriptions of customers and their purchases.

Figure 2.5: NeTEx data structure

2.3.3. Additional Standards and Regional Usage

The UK uses TransXChange as the standard for sharing bus schedule information, while GTFS and GTFS-RT (General Feed Specification Real-Time) are more commonly used in the U.S. and other regions. European transit systems prefer NeTEx, SIRI, and IFOPT (Identification of Fixed Objects in Public Transport), focusing on infrastructure components like stops and stations. Moreover, IFOPT standardizes data regarding physical locations such as stops and interchange points, making it a valuable addition to NeTEx and SIRI for developing quality transportation systems.

In the following chapter 3 it will be discussed how the acquired and formatted data frame from the ZET GTFS feed could be utilized within the application framework and also how it is designed to do so from the front-end, back-end and storage side.

3. Application Design and Implementation

3.1. Application Architecture

Application in this thesis is designed by the front-end and back-end structure system with the connection in between handled by the used framework's internal connections. The primary framework used is Dash, a Python-based platform that can create a synergy between backend data processing and frontend visualization. In the next chapters 3.1.1 and 3.1.2, the used technologies for development will be explained and the overall system structure implemented.

3.1.1. Used Technologies

To develop the real-time visualization application various technologies and libraries have been used so that the application can achieve its functionality. In the following, all of the main technologies which are used together with Python are described:

- **Dash**: the center of the application is Dash which is built on top of Flask, React.js and Plotly.js. This framework can be used to create web applications with interactive visualizations using only Python code. With Dash, processing in the backend is possible since every function written in Python can be easily called upon with callback decorators. This enables the framework to send the processed data directly to the front-end where it can be rendered. There are two main components of Dash.
    - **Dash(app. layout)**: described as a component which defines how components like User Interface (UI) elements, dropdowns or graphs on specific tabs are handled and displayed. It can be said that this is the overall layout of the whole web application which ensures that all visual components are properly arranged by following basic horizontal ruling.
    - **Dash(app.callback)**: this component handles the connection from the front-end to the function getters of the backend as a Python decorator. The user inputs like tab switching, dropdown selection or even button clicks are connected to corresponding actions which return information that can be updated in the layout. This enables the application to have dynamic content like graph updates in

response to the user input.

- **Flask**: for Dash to work properly, the backend server should be initialized and that's where Flask comes into play. It enables functionalities like handling HTTP requests and managing sessions which in a way then enables a Dash app to be hosted.
- **Plotly**: this library is integrated into Dash and it provides tools for creating complex graphs that can also be interacted with. There is a wide range of chart types and because of this, it represents the main essential for visualizing the prediction graphs and overall GTFS data.
- **Pandas and NumPy**: Pandas and Numpy are used for data manipulation and computations. If there is structured data NumPy and Pandas have the functions to interact with it. They offer not only functions for data manipulation but also support for multi-dimensional arrays and matrices with already integrated mathematical functions.
- **Scikit-learn**: this is a machine learning library used with Python. It is used for model training and evaluation within the application. Various supervised and unsupervised learning algorithms are utilized to forecast delays in public transportation.
- **Prophet**: Prophet, created by Facebook, is a forecasting tool utilized for time series forecasting within the application. It is especially beneficial in forecasting patterns using past information, like delays in public transportation.
- **Dash Leaflet**: this library, an expansion of Dash, is employed for displaying maps and managing geospatial information. It works together with Leaflet.js, a JavaScript library for interactive maps, enabling in-depth geographic visualizations in the Dash framework.
- **MySQL Connector**: this library is used for SQL database interaction. It allows direct communication with the MysQL database for data storage and also retrieval.
- **GeoPandas and Shapely**: these libraries are expansions of pandas library which allows operations on geometric-type data. Geopandas makes it easier to perform spatial joins and calculate distances, tasks that are essential for analyzing public transportation routes and stops.
- **BeautifulSoup and Requests**: when in need to scrape the web these tools come in handy because they have the utilities to fetch and parse HTML and XML documents. This library is used to gather data from specific web sources.
- **Threading and concurrent.futures**: this are Python modules which enable concurrent execution. This means that the application can handle multiple tasks simultaneously, which in this application can be for example downloading GTFS static data, recording GTFS real-time data and visualizing the data with Plotly graphs at the same time.

All of the aforementioned technologies and libraries work together to create an application that can handle real-time data, provide visualizations with rendering and also train and predict at the same time.

3.1.2. Overall System Structure

The system structure of every application should focus on communication between the front-end and the back-end. It should be planned so that the application can parse large datasets in between and store the data in one specific session if it's server-based and accessed through a browser. To enable that, the system has been organized into specific components or classes, where each has specific roles and responsibilities. The following components are the most important ones which enable the application to function properly:

- **GtfsDataDownloader**: Responsible for acquiring and downloading static GTFS data, as well as initializing and downloading OpenStreetMap (OSM) data for a specified city which can be used for extracting polylines for trajectory drawings.
- **GtfsSqlDBParser**: Responsible for managing and storing GTFS data in a MySQL database, insertion of static and live data with multi-threaded data insertion and maintenance sub-tasks which are in the field of periodical backups and overall table management.
- **GTFS Public Transport Net Constructor**: This component is used for filtering bus and tram extracted OSM transport network data based on specific criteria such as route IDs, and stop IDs which enables detailed analysis of routes and stops.
- **DatePolygonDBProcessor**: This utility processes database records within specified date ranges and polygon coordinates, enabling spatial and temporal queries to extract relevant data for analysis.
- **GraphHistoricalConstructor**: This utility is used for creating constructor objects for specific types of graphs for historical delay data visualization. The function enables the user to create graphs like average delay by stop, delay distribution histograms, on-time performance pi-charts and similar. Additionally to that it also supports machine learning error metrics for model comparisons.
- **Backup and Scheduling System**: This component is used in situations when scheduled tasks are important for long-run data recordings. Automated backup functionality for MySQL database is implemented which ensures data availability in case some serious errors occurred when inserting or retrieving data.
- **LogParser**: Handles logging events within the application server. Log entries with various message levels (info, warning, error, debug) and store the last 100 entries in a

log buffer, which can be accessed on request. This is used to monitor and debug the application.

One thing that was thought over time is that the system should have modularity; while designing the app at the start, it was not planned to have functionalities divided into separate abstract layers, which did present a problem later in development when it was important, for example, to log messages additionally while the application is running on a server or have the functionality to seamlessly add any ml_model to the mix. The modular system simplifies overall development and eases maintenance since the developer knows exactly where specific things are located. Figure 3.1 shows a diagram of the overall system structure and the interactions between the components.

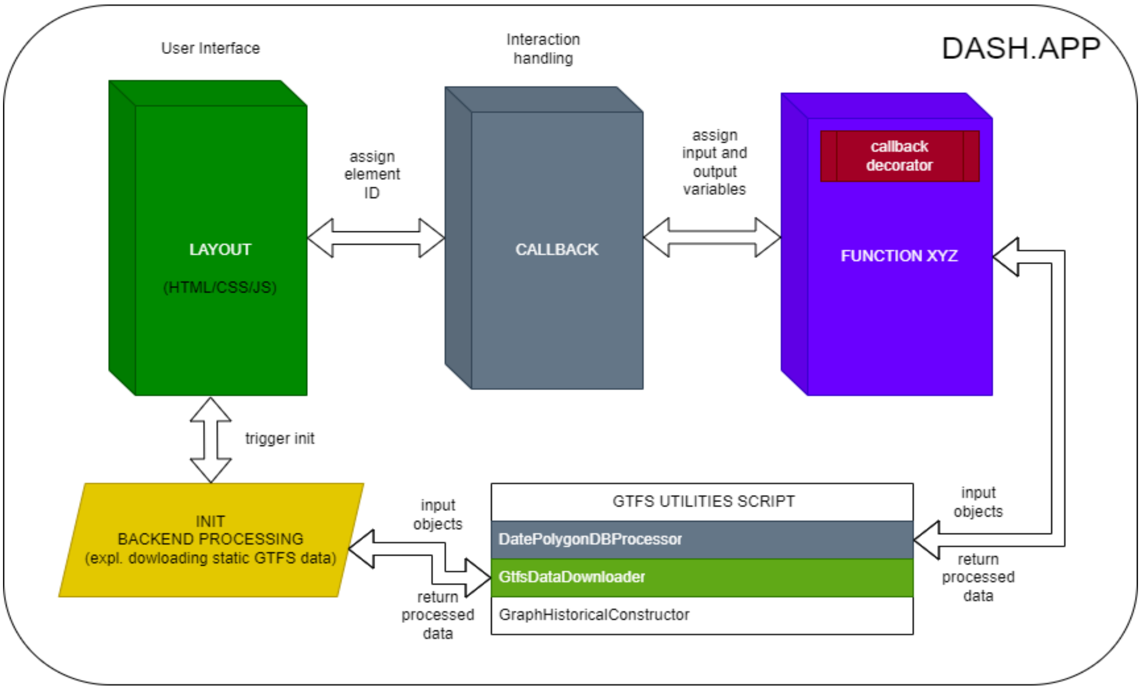


Figure 3.1: Simplified diagram of Dash app structure

It illustrates the data flow within the Dash application, which enables user interface components to be connected to the backend data processing and utility scripts. The component "layout" can be categorized as the front-end layer of the Dash application, which can be interacted through Python code or directly through HTML, CSS and JavaScript. This contains the structure of the UI elements, that can be dropdowns, buttons, graphs and etc. The element "callback" then interacts between the user interface and the backend functions by wrapping them as a decorator. The functions "XYZ" are then called via the decorator and executed where the data then is parsed back to the frontend components. There are additional "back end processing initialization" functions called once at the beginning. For example, if properly set up with the JSON configuration, the layout will trigger the downloading of static GTFS data only once at the beginning. This illustrates how the layout interacts with the processed data and then brings it back to the user to interact with it.

3.2. Front-End Integration with Back-End

3.2.1. Front-End Implementation

Front-end development is a part of the application creation that is mostly focused on how the user interacts with the app directly through the web browser. This involves creating layouts, designing them and making sure that the interactivity of the application is on a specific level, etc. Front-end technologies like HTML, CSS, and JavaScript are primarily used where it is important to understand how user input is handled and ensure that the interface does what it is designed to do [13].

The application's front-end layout is centred around the Single-PAge (SPA) design paradigm. Dash usually can't function with a multi-page approach and was not meant to do that. It was, in most cases, used as a dashboard for data analysis or machine learning showcases, which did not require multiple separate pages.

A single-page application is a web application or website that rewrites its active web page with new data from the web server dynamically while the user is interacting with it. It is not loading any new pages, only one page is open which constantly refreshes and gets new data on the same page. It can sometimes seem like the web page has multiple pages overall and loads extremely fast even though it is only one page, which loads its resources dynamically and adds them to the page when necessary, usually as a response to the user's actions [14].

The "app.layout" in Figure 3.2 shows the application's structure from the front-end side. It mentions the most relevant components, which interact directly with the user or the active session of the application instance in the browser. The "header" component, which contains the logos and application titles, is logically at the top, where every part of the big SPA is separated by a horizontal ruling. In the same layer, the session data storage is reserved for storing application states, such as which graphs were selected on the "Info" tab or on which coordinates and zoom level you left the "Map" tab.

The layer underneath manages the interactions that require some kind of confirmation. In this case, when you miss clicking some buttons, it will create a pop-up warning you not to do so, etc. On the other hand, interval counters are used for periodic updates to the interface. This functionality is very important since it is used to frequently update the vehicle positions on the map so that the user is informed of the vehicle location at stop arrival.

The Tab content is divided into three parts, which are in the following:

- **Map Tab**: this is used for visualizing real-time data directly on the map. That includes the current position of the vehicle for the selected route and other descriptions. Users can additionally click and crate polygons to include or exclude specific stops for later evaluation.

- **Options Tab**: this tab allows users to select specific routes or batches of routes then be shown on the "Map" tab through the "Tram or Bus Dropdowns" and, with that, also the data that should be loaded. Additionally, it also includes controls for starting and stopping of GTFS data recording and loading of static data.
- **Info Tab**: this tab is primarily used for displaying graphs and filtering through the extracted "store_dataframe" dataset so that specific data can be excluded or included in graph visualization. It also allows the user to select different machine-learning models to test out.

The layout is set up with the application's first initialization, and HTML containers with specific IDs are assigned to the callback, which then waits for the user input or interval trigger.

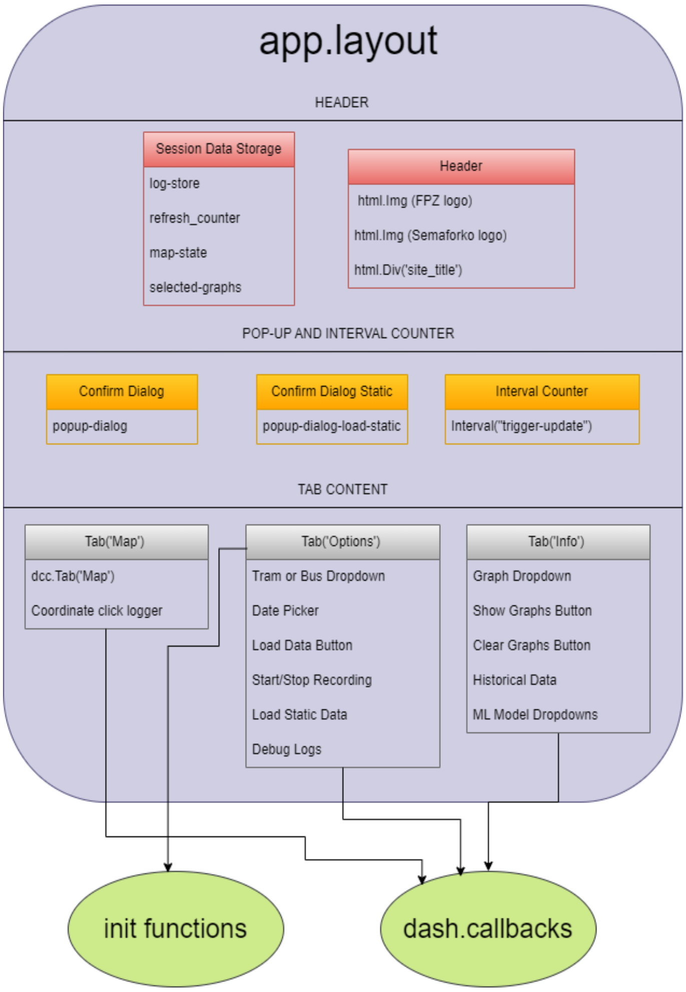


Figure 3.2: GTFS application front-end structure

3.2.2. Back-End Implementation

Back-end development is how the server side of the application is implemented to make the application do what it is supposed to. When a user interacts with the application, he triggers components located on the front-end side that communicate to the processing components on the server. The logic behind every component, the application's database, and the overall server state compose the back end. To maintain the server's state, developers use programming languages like Python, PHP, and Java, as well as frameworks that use the corresponding programming language, like Django, Node.js, or Flask, in the case of this thesis. For every user's interaction with trigger front-end components, he, in translation, makes a client request that the server, for example, executes some data transactions or assigned functions with specific server-side logic [15].

The GTFS application's back end is implemented with scripts and classes to handle data processing, machine learning, model training and concurrent database interactions. When the user triggers a button with a specific HTML container ID, which corresponds to a specific function with its Dash callback decorator, it starts a specific chain reaction where data starts being fetched, processed and then afterwards prepared for graph visualization in the front-end. The main components which make this work in the back-end were already explained in chapter 3.1.2, where the overall functionalities were described.

In the following figure 3.3, it can be seen that the front-end part with its corresponding tabs is connected to the back-end process chain. To better understand the corresponding figure, it first needs to start from the front-end side, which are the "init functions" and the "dash. callback" connectors.

In this regard when the functions are called directly from the "app.layout" the init functions will trigger the "DatePolygonDBProcessor", which, for example, processes "date_ranges" at every initialization of the app by doing a query to fetch the minimum and maximum timestamp from the database. This triggers the "DatePolygonDBProcessor" to call for the "GtfsSqlDBParser" which then connects to the database and inserts or fetches data on request. Onward, "GtfsSqlDBParser" in addition to that, then requests the "GtfsDataDownloader" to, if needed, fetch the GTFS data or process the OSM data that should be shown on the "Map Tab".

The same thing applies to the "dash.callback" where this can also trigger the "MiModelConstructor" with a drop-down selection from the "Info Tab" interface. The "MiModelConstructor" after training and predicting, and creating the data for visualization triggers the "GraphHistoricalConstructor" that takes this data and then generates or updates the graphs. In the end then, "GraphHistoricalConstructor" send the corresponding constructed figure object back to the callback to update the "app.layout".

All of this is designed to operate asynchronously, so fetching the data processing as described can be done separately on a different thread. This enables the GTFS application's most

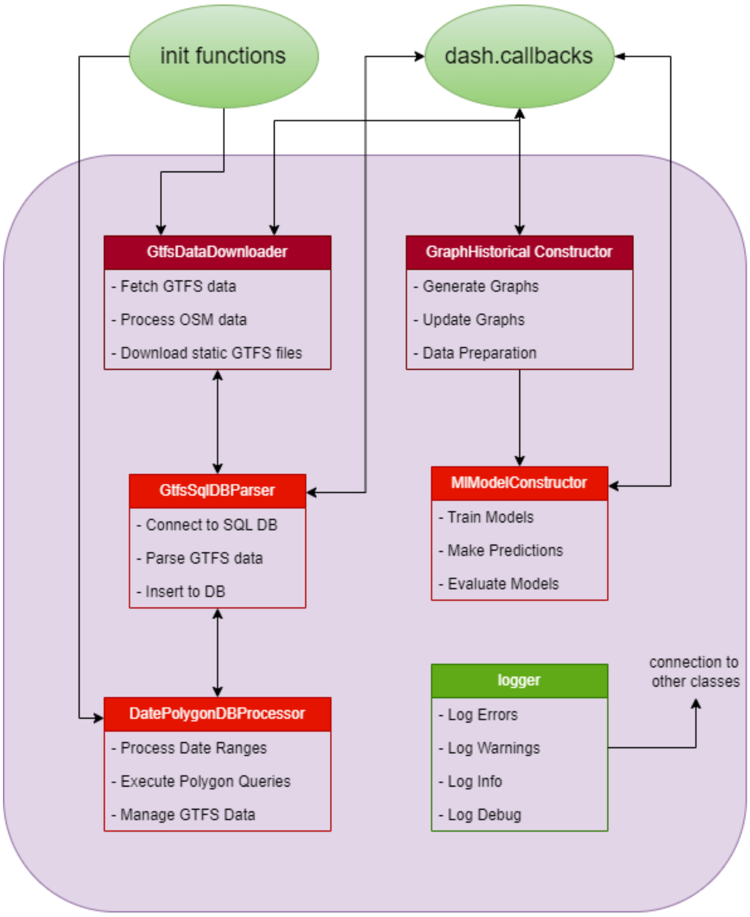


Figure 3.3: GTFS application Back-End process chain

important functionalities to operate even though fetching and processing are being done in the background to create evaluation data.

3.2.3. Fron-Back Integration

Integration, from a software development perspective, occurs when specific subsystems or parts of the software are integrated to create one whole system. Integrated systems like this demonstrate better performance than separate independent ones. If the application is separated into specific parts, it delivers greater functional significance [16].

There are different types of software integrations, such as star, horizontal, and vertical, and there are implementations of the common data format type integration. Since connecting the front and back end with Dash is only discussed, the focus will be on the integration type most accustomed to callbacks, the horizontal one [16].

Horizontal integration refers to creating subsystems specifically for communication purposes. The middle man between the server with processing functionalities and the application user is the "Enterprise Service Bus", which reduces the number of connections of each subsystem to one. This creates a layer that translates one interface into another where an example of the architecture can be seen on figure 3.4. For the GTFS application, it happens through callbacks, which Dash employs.

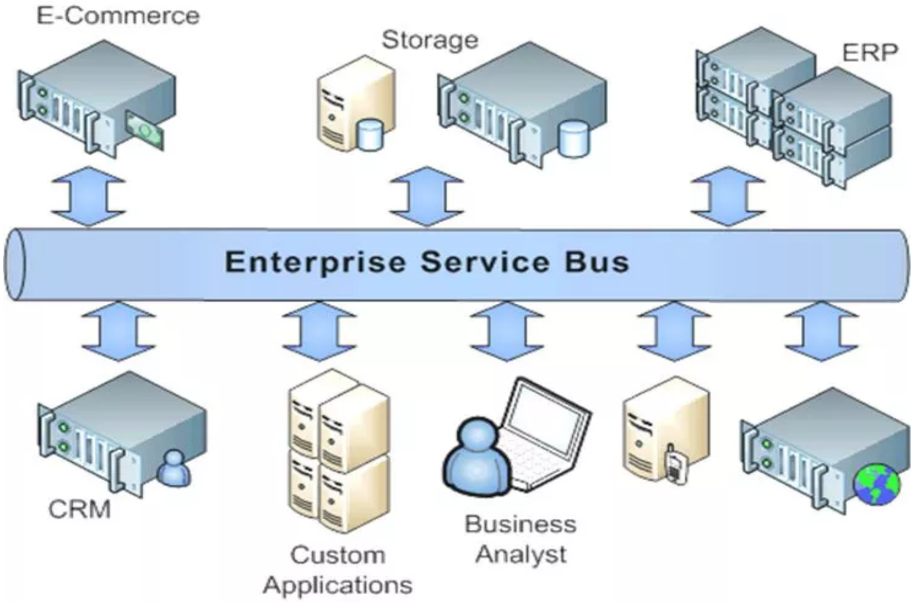


Figure 3.4: Horizontal Integration Example [16]

Horizontal integration of front-end and back-end only using the Python programming language, with the middle man being the callbacks, creates a cohesive system that can function independently. For example, when a user selects a date range or the route from the drop-down menu, the callback is triggered, initiating the back-end processing and operations.

In figure 3.5, it can be seen that the architecture is constructed with boundaries between each component. It also illustrated how horizontal integration is implemented in the Dash core with callbacks and setup_callbacks() function to emulate a central hub between the "layout" and the back-end functions. As seen from the central part of the figure, Dash App manages user inputs session data with server initialization, which is then connected to other backend components like "GtfsDataDownloader", "GtfsSqlDBParser", and etc. These components then interact with the callback interface, making the architecture easily changeable.

One of the important aspects was the logging part since debugging was always important in any big application, and the object logger is attached to every class with the "config" initialization import. This means that the "LogParser" is initialized once in a separate script where the object is created, and then afterwards, the object is imported to another script that needs to be used.

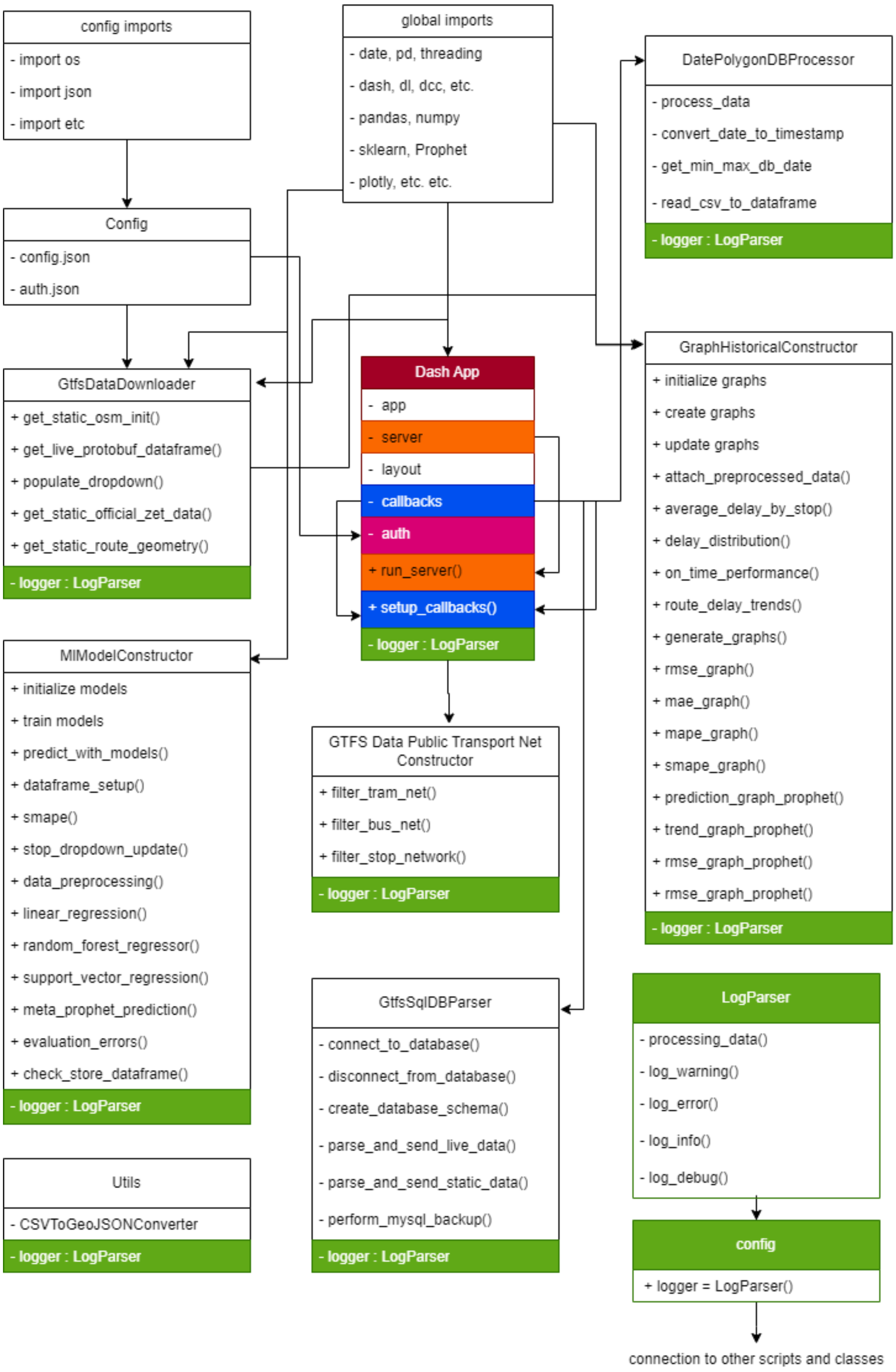


Figure 3.5: Detailed architecture of the whole GTFS Dash application

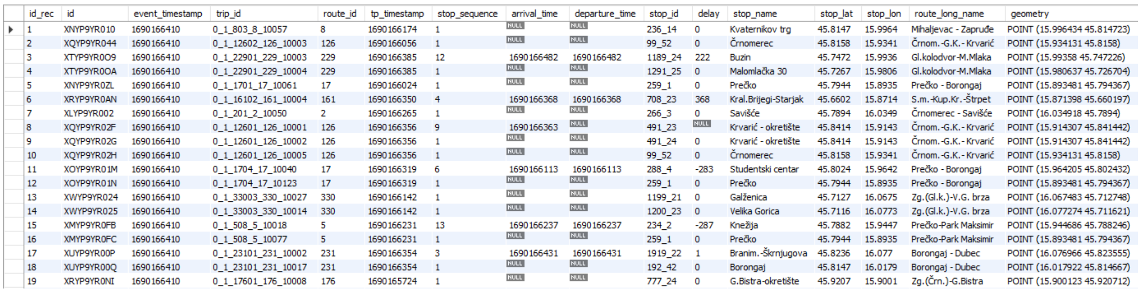
Ultimately, this creates a full application that can be flexible and scaled according to the user's needs. Regarding scaling, new functions and graphs can easily be integrated by adding new functions to classes for object creation, like "MlModelConstructor" and "GraphHistorical-Constructor."

In the following chapter 3.3, the way this data is stored with the "GtfsSqlDBParser" will be explained, focusing mostly on how to interact with the real-time and static GTFS data and the creation of the main "zet_feed_data" table.

3.3. Database Implementation

3.3.1. Overview of Databases

Databases are a collection of structured data. These data collections are connected to each other and can be served further to the end user. Databases are usually contained from specific elements, which are pieces of data called "data_items", for example, a "route_long_name" from the "zet_feed_data", and then there is a group of related data which are called "records" [17]. Records are viewed as separate entities in databases, such as "routes_id" connected to "zet_feed_data" as a column of the same entity. Since columns are attributes and records are rows, this together forms a relation, as shown in figure 3.6.



id_rec	id	event_timestamp	trip_id	route_id	tp_timestamp	stop_sequence	arrival_time	departure_time	stop_id	delay	stop_name	stop_lat	stop_lon	route_long_name	geometry
1	XNYP9YR010	1690166410	0_1_803_8_10057	8	1690166174	1	NULL	NULL	236_14	0	Kvaternikov trg	45.8147	15.9964	Mihaljevac - Zapruđe	POINT (15.996434 45.814723)
2	XQYP9YR044	1690166410	0_1_12602_126_10003	126	1690166056	1	NULL	NULL	99_52	0	Črnomerec	45.8158	15.9341	Črnom.-G.K.- Krvarić	POINT (15.934131 45.8158)
3	XTYP9YR0O9	1690166410	0_1_22901_229_10003	229	1690166385	12	1690166482	1690166482	1189_24	222	Buzin	45.7472	15.9936	Gl.kolodvor-M.Mlaka	POINT (15.99358 45.747226)
4	XTYP9YR0OA	1690166410	0_1_22901_229_10004	229	1690166385	1	NULL	NULL	1291_25	0	Malomlačka 30	45.7267	15.9806	Gl.kolodvor-M.Mlaka	POINT (15.980637 45.726704)
5	XNYP9YR0ZL	1690166410	0_1_1701_17_10061	17	1690166024	1	NULL	NULL	259_1	0	Prečko	45.7944	15.8935	Prečko - Borongaj	POINT (15.893481 45.794367)
6	XRYP9YR0AN	1690166410	0_1_16102_161_10004	161	1690166350	4	1690166368	1690166368	708_23	368	Kral.Brijegi-Starjak	45.6602	15.8714	S.m.-Kup.Kr.-Štrpet	POINT (15.871398 45.660197)
7	XLYP9YR002	1690166410	0_1_201_2_10050	2	1690166265	1	NULL	NULL	266_3	0	Savišće	45.7894	16.0349	Črnomerec - Savišće	POINT (16.034918 45.7894)
8	XQYP9YR02F	1690166410	0_1_12601_126_10001	126	1690166356	9	1690166363	NULL	491_23	NULL	Krvarić - okretište	45.8414	15.9143	Črnom.-G.K.- Krvarić	POINT (15.914307 45.841442)
9	XQYP9YR02G	1690166410	0_1_12601_126_10002	126	1690166356	1	NULL	NULL	491_24	0	Krvarić - okretište	45.8414	15.9143	Črnom.-G.K.- Krvarić	POINT (15.914307 45.841442)
10	XQYP9YR02H	1690166410	0_1_12601_126_10005	126	1690166356	1	NULL	NULL	99_52	0	Črnomerec	45.8158	15.9341	Črnom.-G.K.- Krvarić	POINT (15.934131 45.8158)
11	XOYP9YR01M	1690166410	0_1_1704_17_10040	17	1690166319	6	1690166113	1690166113	288_4	-283	Studentski centar	45.8024	15.9642	Prečko - Borongaj	POINT (15.964205 45.802432)
12	XOYP9YR01N	1690166410	0_1_1704_17_10123	17	1690166319	1	NULL	NULL	259_1	0	Prečko	45.7944	15.8935	Prečko - Borongaj	POINT (15.893481 45.794367)
13	XWYP9YR024	1690166410	0_1_33003_330_10027	330	1690166142	1	NULL	NULL	1199_21	0	Galženica	45.7127	16.0675	Zg.(Gl.k.)-V.G. brza	POINT (16.067483 45.712748)
14	XWYP9YR025	1690166410	0_1_33003_330_10014	330	1690166142	1	NULL	NULL	1200_23	0	Velika Gorica	45.7116	16.0773	Zg.(Gl.k.)-V.G. brza	POINT (16.077274 45.711621)
15	XMYP9YR0FB	1690166410	0_1_508_5_10018	5	1690166231	13	1690166237	1690166237	234_2	-287	Knežija	45.7882	15.9447	Prečko-Park Maksimir	POINT (15.944686 45.788246)
16	XMYP9YR0FC	1690166410	0_1_508_5_10077	5	1690166231	1	NULL	NULL	259_1	0	Prečko	45.7944	15.8935	Prečko-Park Maksimir	POINT (15.893481 45.794367)
17	XUYP9YR00P	1690166410	0_1_23101_231_10002	231	1690166354	3	1690166431	1690166431	1919_22	1	Branim.-Škrnjugova	45.8236	16.077	Borongaj - Dubec	POINT (16.076966 45.823555)
18	XUYP9YR00Q	1690166410	0_1_23101_231_10017	231	1690166354	1	NULL	NULL	192_42	0	Borongaj	45.8147	16.0179	Borongaj - Dubec	POINT (16.017922 45.814667)
19	XRYP9YR0NI	1690166410	0_1_17601_176_10008	176	1690165724	1	NULL	NULL	777_24	0	G.Bistra-okretište	45.9207	15.9001	Zg.(Črn.)-G.Bistra	POINT (15.900123 45.920712)

Figure 3.6: Zed_feed_data records

In addition to data, a relational database creates relationships between data groups. From what it can be concluded, the data must have meaning and mutual relationships for something to be considered a database. A group of data that is just random cannot be considered a database.

Since the database cannot be managed independently, DataBase Management Systems (DBMS) and software tools enable the creation, definition, handling and sharing of databases with the end users and applications [17]. When you join the DBMS and the database, you get a database system, as shown in figure 3.7.

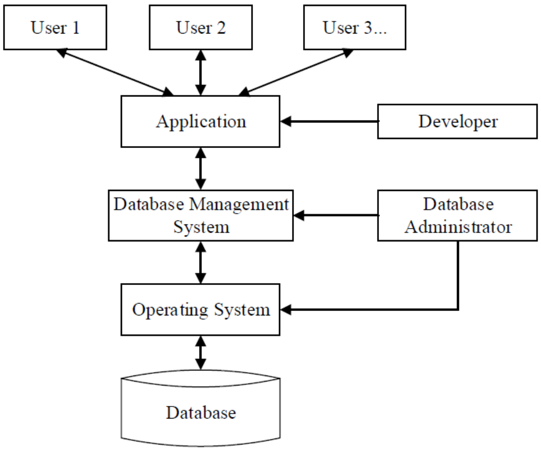


Figure 3.7: The composition of a database system [18]

Database systems were developed in conjunction with databases because long-term data storage and simultaneous access from multiple users have been needed. This data approach is evident through centralized data management, where independence between data and application increases integrity, supports different data views, and reduces data inconsistency [17].

Database systems are usually divided based on the number of logical levels and where software support is located. Two-tier, three-tier, or more-tier database architectures exist. Two-tier database architectures are client-server types of architecture in which the client has the Graphical User Interface (GUI), applications, and software support for business operations or processing, while the server side has software support for querying, servicing, and transaction processing. These functions are needed for data retrieval and manipulation. These types of servers are also called query servers or transaction servers. A two-tier database implementation can be seen in figure 3.8 [19].

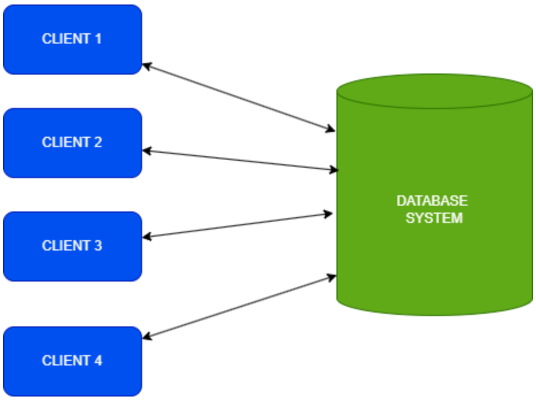


Figure 3.8: Tier-2 database architecture

If, in some regards, a tier between the server and the client is added, the client could achieve better performance since it provides great scalability of applications and can reuse the implemented software component. As can be seen from Figure 3.9, a three-tier architecture is constructed by adding a middle tier between the database system and the data presentation layer.

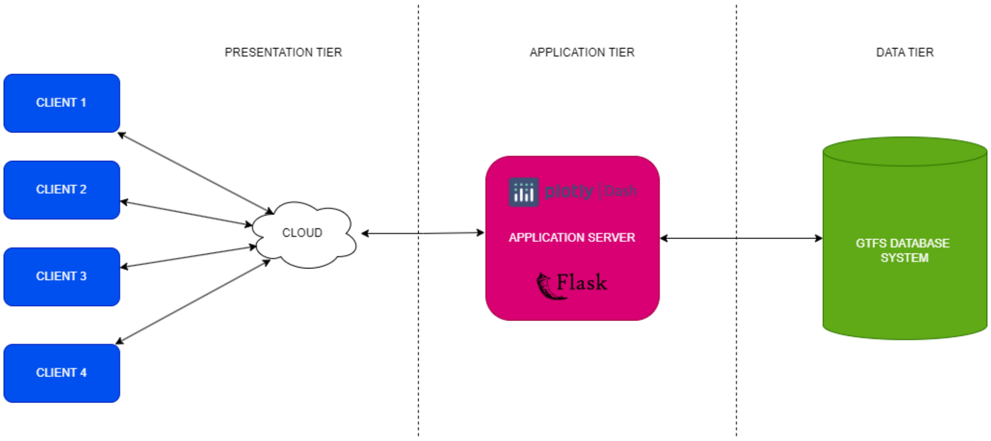


Figure 3.9: Tier-3 database architecture

This tier is an application or web server containing procedures and data retrieval and authentication methods. In this regard, the database server, logic, applications, and programs are separated completely from the client level, which is only for presenting data on a web or a graphical interface [17].

In the case of the GTFS application, tier-3 architecture is implemented, where clients could be every user who interacts with the Dash front-end presented on the web browser, which is hosted on a Flask server in the application tier. In this regard, every callback triggered by the user from the front-end could potentially also initialize the "GtfsSqlDBParser," which will either start retrieving or processing some data in the data tier.

3.3.2. GTFS Database Structure

In the previous sections, GTFS application development was discussed from the perspective of back-end processing and efficient data management, but now, with transitions, it is forwarded to the application's database design, which ensures that the application has efficient data storage with a function of retrieval and processing in the data tier. The GTFS database is structured as a relational database, which is important when dealing with data like public transport schedules and real-time vehicle tracking on stop arrival regarding data management.

The GTFS database organizes real-time feed and static data into tables with corresponding rows and columns. Each table stores data about a specific entity, such as routes, stops, trips, etc., in the context of GTFS public transportation system data. Every table has some relationship with another table where everything is defined using foreign and primary keys. This principle links records of one table to another, ensuring that data integrity exists. This also helps the user execute complex queries that are needed occasionally.

The relational GTFS database implemented is illustrated in the following schema on figure 3.10. This schema type can handle static and real-time data related to GTFS public transport data.

The main tables of the schema are the following:

- **zet_feed_data**: this table stores real-time data about vehicles' location on stop arrival. It also stores timestamps, route IDs, delays, geospatial data etc. This is the central piece regarding real-time data recording. It is referenced through foreign keys to tables "stops", "routes", and "trips".
- **stops**: the table contains static data about each stop in the ZET transport network, including the stop name, latitude, longitude and other metadata. It is referenced through primary keys to tables "zet_feed_data" and "stop_times".
- **trips**: this table specifies information about each trip taken. It has information like trip head signs and directions. It is the most referenced table in the schema which is con-

nected with its primary and foreign keys to tables "zet_feed_data", "time_record_static", "stop_times", "calendar", "routes", "calendar_dates".

- **routes**: this table stores data about each route, which includes the route ID, name and, most importantly, type (bus, tram). It is connected through its foreign key to "trips", and through its primary key to "agency".
- **stop_times**: this table stores the static data about scheduled arrival and departure times for each trip stop. This also includes the full stop sequence so that it is recognized which stop it is for which trip. It is connected through its foreign key to "trips" and its primary key to "stops. "
- **calendar**: this table stores data about service availability through the week and service changes. It is connected through its foreign key to "trips".
- **agency**: stores data about the agency's metadata. Not so important in this regard since we are only dealing with one agency, and that is ZET. It is connected through its foreign key to the "routes" table.

These tables are interconnected, allowing the data to be retrieved in different ways and conditions. For example, if the link is established between the "trip_id" in the "zet_feed_data" table to the "trips" table and then afterwards to the "routes" table using JOIN operators and adding columns in the final query result to get the data, we get the complete picture of a trip's nature in the transport network.

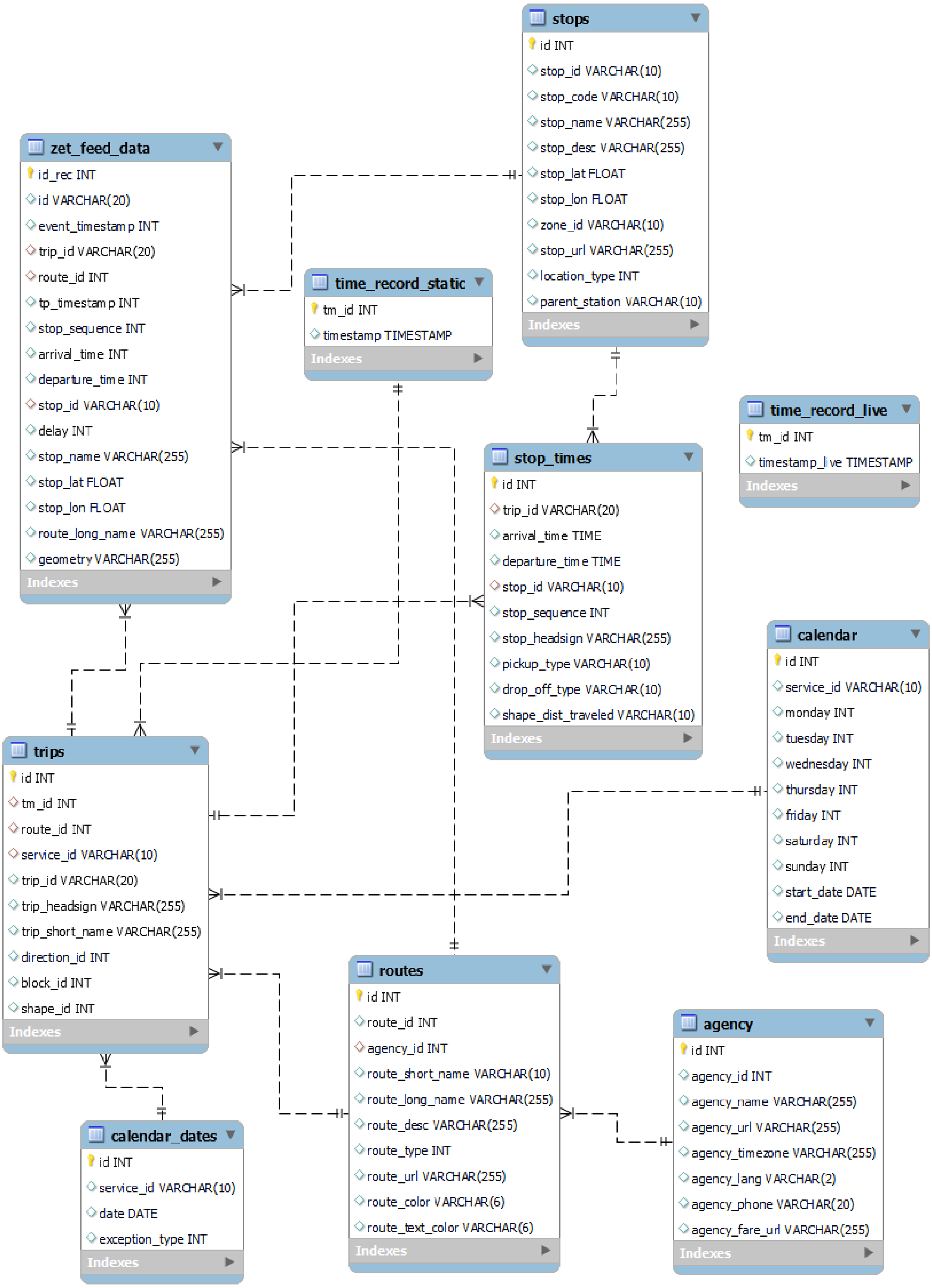


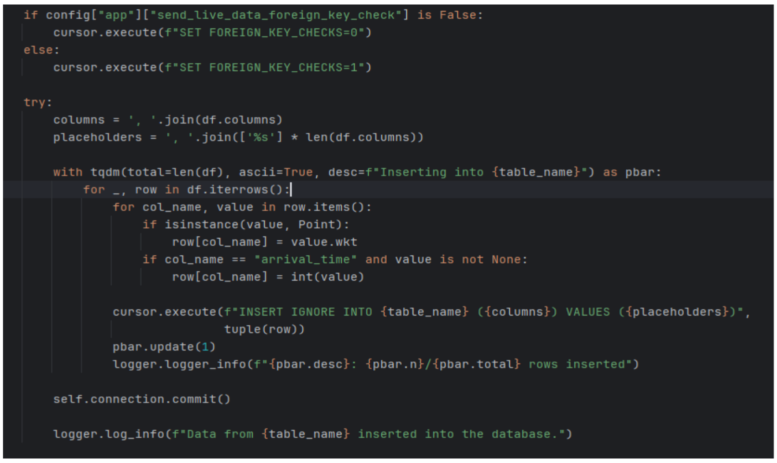
Figure 3.10: Relational Schema for ZET GTFS database

3.3.3. GTFS Database Data Handling

Data retrieval and insertion are parts of the basic ZET GTFS database data handling which are important in the scope of the GTFS application. After constructing the database schema, the tables, which are interconnected, allow for data insertion and retrieval when triggered by the user. These requests that the user can make could be to get a batch of information correlated to specific routes or a specific time frame for the chosen route. For this to be possible, three main functionalities were implemented:

Inserting real-time data

For real-time data insertion it is important to have a script that can handle dynamic data updates accordingly. On figure 3.11 it can be seen that the script first prepares the incoming real-time data so that it matches the database schema format. Columns and placeholders for the data positions are counted and the correctness of geospatial data types is checked. The script uses the "mysql.connector.cursor.execute" to execute the SQL command within the transaction block. Afterwards the the whole transaction can be committed accordingly.



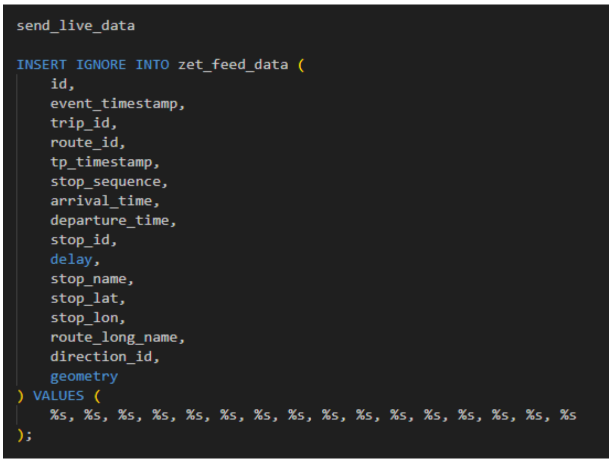
```python
if config["app"]["send_live_data_foreign_key_check"] is False:
    cursor.execute(f"SET FOREIGN_KEY_CHECKS=0")
else:
    cursor.execute(f"SET FOREIGN_KEY_CHECKS=1")

try:
    columns = ', '.join(df.columns)
    placeholders = ', '.join(['%s'] * len(df.columns))

    with tqdm(total=len(df), ascii=True, desc=f"Inserting into {table_name}") as pbar:
        for _, row in df.iterrows():
            for col_name, value in row.items():
                if isinstance(value, Point):
                    row[col_name] = value.wkt
                if col_name == "arrival_time" and value is not None:
                    row[col_name] = int(value)

            cursor.execute(f"INSERT IGNORE INTO {table_name} ({columns}) VALUES ({placeholders})",
                           tuple(row))
            pbar.update(1)
            logger.logger_info(f"{pbar.desc}: {pbar.n}/{pbar.total} rows inserted")

    self.connection.commit()

    logger.log_info(f"Data from {table_name} inserted into the database.")
```

Figure 3.11: Python Script for inserting real-time data into GTFS database

For each row of real-time data, the script constructs an SQL "INSERT IGNORE." This was set up intentionally so that when the GTFS data is corrupted due to problems in the data format, etc., it will not break but continue to record further. This approach creates redundancy, but with the time-frame extension of the data-interval update, it would be minimal even though some redundant entries are created. This did not significantly impact the data quality in the end.

Additionally, the script employs the "tqdm" library, which was not mentioned before, to provide an overview of the user's progress. This was really significant when inserting large volumes of data like "stop_times," for example, in the database, where it usually takes around 3-4 minutes to record the whole static data block completely.

The result for one row insert through "connection.execute" adding data placeholders to the transaction can be seen in figure 3.12 before commit. Also, in this regard, since this only enables the creation of the placeholder, all of the row record data is then inserted by adding the "tuple(row)" with its data to its corresponding placeholders.

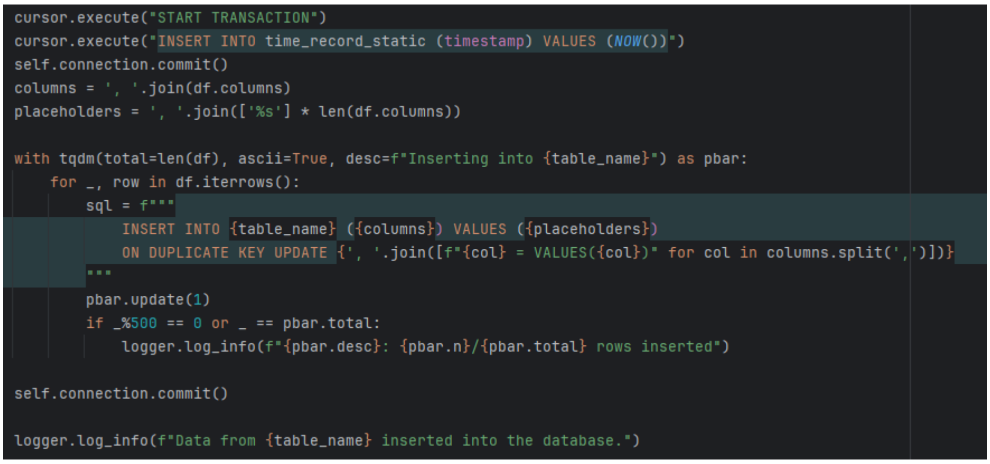


```
send_live_data

INSERT IGNORE INTO zet_feed_data (
    id,
    event_timestamp,
    trip_id,
    route_id,
    tp_timestamp,
    stop_sequence,
    arrival_time,
    departure_time,
    stop_id,
    delay,
    stop_name,
    stop_lat,
    stop_lon,
    route_long_name,
    direction_id,
    geometry
) VALUES (
    %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s
);
```

Figure 3.12: Result of one-row append to the transaction for real-time data

Inserting Static data

Static data insertion, which can include routes, stops, and agencies, for example, is implemented using a different approach since the updates are not as frequent as those in real-time insertion. The figure 3.13 shows the Python script which handles static data insertion into the GTFS database.

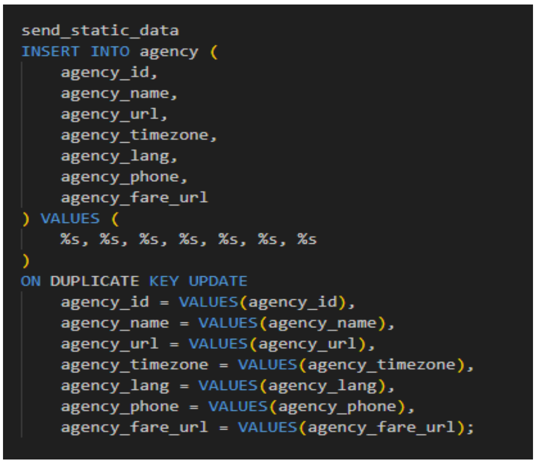


```
cursor.execute("START TRANSACTION")
cursor.execute("INSERT INTO time_record_static (timestamp) VALUES (NOW())")
self.connection.commit()
columns = ', '.join(df.columns)
placeholders = ', '.join(['%s'] * len(df.columns))

with tqdm(total=len(df), ascii=True, desc=f"Inserting into {table_name}") as pbar:
    for _, row in df.iterrows():
        sql = f"""
            INSERT INTO {table_name} ({columns}) VALUES ({placeholders})
            ON DUPLICATE KEY UPDATE {', '.join([f"{col} = VALUES({col})" for col in columns.split(',')])}
        """
        pbar.update(1)
        if _%500 == 0 or _ == pbar.total:
            logger.log_info(f"{pbar.desc}: {pbar.n}/{pbar.total} rows inserted")

self.connection.commit()

logger.log_info(f"Data from {table_name} inserted into the database.")
```

Figure 3.13: Python Script for inserting static data into GTFS database

In a similar manner, as it has been discussed in 3.3.3, the key columns of the placeholders are constructed to align with those of the "zet_feed_data" table so that the data can be placed accordingly. The script dynamically constructs a SQL "INSERT INTO...ON DUPLICATE KEY UPDATE" which inserts new records and updates existing ones if duplicates are detected based on primary keys. For static data, it is important to eliminate redundancy since appending static data would create problems later when fetching data for analysis since it doubles the querying time.

The outcome for one-row insertion is depicted in figure 3.14, which shows how the data is inserted with appropriate placeholders and values to its table.

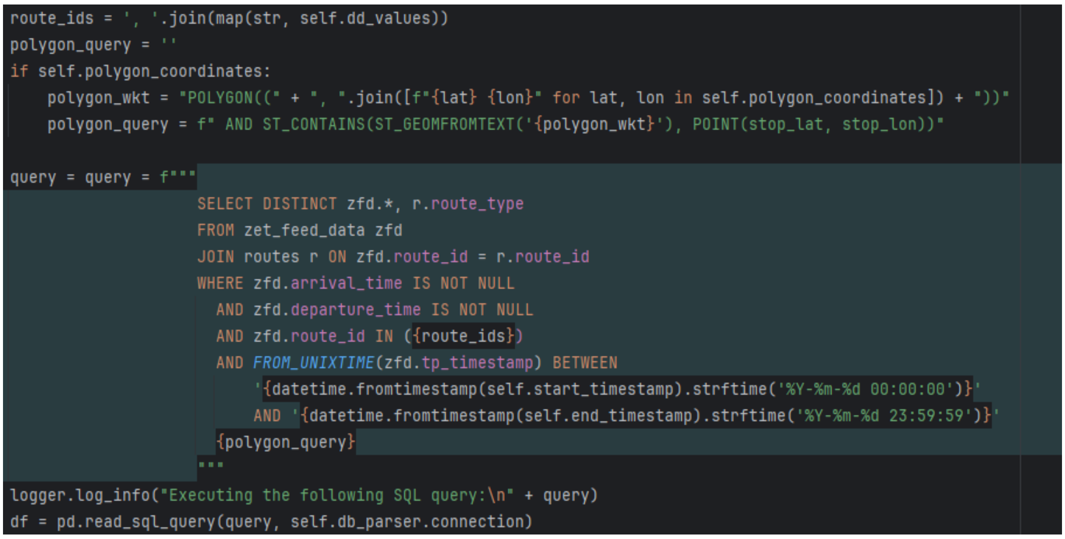


```
send_static_data
INSERT INTO agency (
    agency_id,
    agency_name,
    agency_url,
    agency_timezone,
    agency_lang,
    agency_phone,
    agency_fare_url
) VALUES (
    %s, %s, %s, %s, %s, %s, %s
)
ON DUPLICATE KEY UPDATE
    agency_id = VALUES(agency_id),
    agency_name = VALUES(agency_name),
    agency_url = VALUES(agency_url),
    agency_timezone = VALUES(agency_timezone),
    agency_lang = VALUES(agency_lang),
    agency_phone = VALUES(agency_phone),
    agency_fare_url = VALUES(agency_fare_url);
```

Figure 3.14: Result of one-row append to the transaction for static data

Fetching data for graphs and analysis

Fetching the data from the GTFS database for use in machine learning, average delays by stop, etc., required a slightly more complex SQL query. In this regard, it is also needed to define specific stops based on their location and how it is drawn through the polygon tool and select routes with other boundaries. The Python script shown in Figure 3.15 demonstrates how the data is fetched to be used later for analysis.

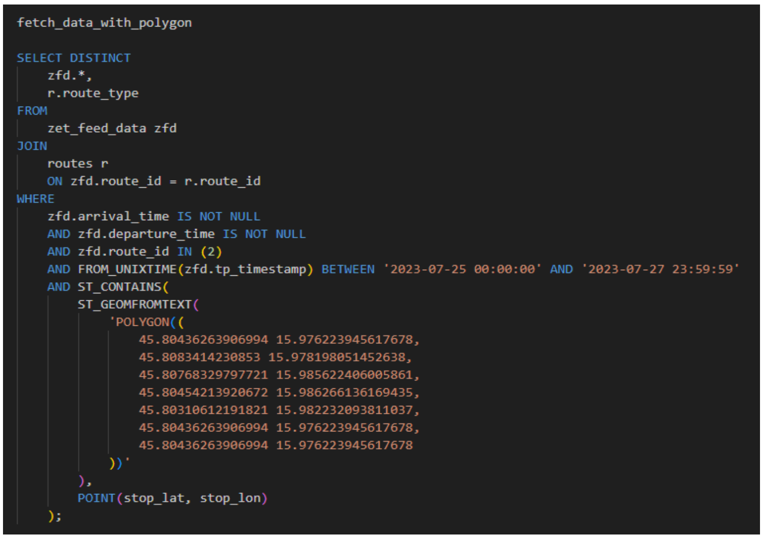


```
route_ids = ', '.join(map(str, self.dd_values))
polygon_query = ''
if self.polygon_coordinates:
    polygon_wkt = "POLYGON((" + ", ".join([f"{lat} {lon}" for lat, lon in self.polygon_coordinates]) + "))"
    polygon_query = f" AND ST_CONTAINS(ST_GEOMFROMTEXT('{polygon_wkt}'), POINT(stop_lat, stop_lon))"

query = query = f"""
                    SELECT DISTINCT zfd.*, r.route_type
                    FROM zet_feed_data zfd
                    JOIN routes r ON zfd.route_id = r.route_id
                    WHERE zfd.arrival_time IS NOT NULL
                      AND zfd.departure_time IS NOT NULL
                      AND zfd.route_id IN ({route_ids})
                      AND FROM_UNIXTIME(zfd.tp_timestamp) BETWEEN
                          '{datetime.fromtimestamp(self.start_timestamp).strftime('%Y-%m-%d 00:00:00')}'
                          AND '{datetime.fromtimestamp(self.end_timestamp).strftime('%Y-%m-%d 23:59:59')}'
                      {polygon_query}
                    """
logger.log_info("Executing the following SQL query:\n" + query)
df = pd.read_sql_query(query, self.db_parser.connection)
```

Figure 3.15: Python Script to fetch data for analysis and evaluation

The script dynamically builds the SQL query based on parameters that the user defines from the front-end side. Routes, date ranges, and geographical polygon coordinates are included in the query.There are two functions included which are "ST_CONTAINS" and "ST_GEOMFROMTEXT" which filter data based on geographical location using polygons and between specific date ranges. This type of querying is important when the user wants to analyse delays for specific stations in a chosen area and time period.

Figure 3.16 shows an example of a query constructed by the script with an example of a polygon area on the map, route, and time frame.



```
fetch_data_with_polygon

SELECT DISTINCT
    zfd.*,
    r.route_type
FROM
    zet_feed_data zfd
JOIN
    routes r
    ON zfd.route_id = r.route_id
WHERE
    zfd.arrival_time IS NOT NULL
    AND zfd.departure_time IS NOT NULL
    AND zfd.route_id IN (2)
    AND FROM_UNIXTIME(zfd.tp_timestamp) BETWEEN '2023-07-25 00:00:00' AND '2023-07-27 23:59:59'
    AND ST_CONTAINS(
        ST_GEOMFROMTEXT(
            'POLYGON((
                45.80436263906994 15.976223945617678,
                45.8083414230853 15.978198051452638,
                45.80768329797721 15.985622406005861,
                45.80454213920672 15.986266136169435,
                45.80310612191821 15.982232093811037,
                45.80436263906994 15.976223945617678,
                45.80436263906994 15.976223945617678
            ))'
        ),
        POINT(stop_lat, stop_lon)
    );
```

Figure 3.16: Result of a constructed SQL query to fetch data from the GTFS database

Collected GTFS data description

The collected data provides essential static and real-time information about the transit system, and the description of the static collected data can be seen from the following table 3.1, which gives an overview of the different static data tables in the GTFS dataset, their respective record counts, and file sizes.

Table 3.1 GTFS Data Overview

Table	Total Records	File Size
agency	1	256 b
calendar	8	392 b
calendar_dates	72	1.22 kb
feed_info	1	178 b
routes	157	8.65 kb
stop_times	1,213,893	80.2 MB
stops	3,854	211 kb
trips	72,664	3.64 MB

The smallest ones are *"agency"* and *"feed_info"* which contain only one record, with the sizes of 256 bytes and 178 bytes. Additionally, the *"calendar"* and *"calendar_dates"* contain schedule information with 8 and 72 records, ranging from 392 bytes to 1.22 kilobytes. The *"routes"* table, at 8.65 kilobytes, stores details of 157 transit routes. The largest, *"stop_times"*, has over 1.2 million records and occupies around 80.2 MB, detailing the schedule for each stop. The *"stops"* table, with 3,854 records, describes physical stops and is 211 kilobytes in size, while the *"trips"* table has 72,664 records with a size of 3.64 MB, providing information about trips.

In addition to the static data, real-time data is stored in the *"zet_feed_data"*, which gives a dynamic picture of the whole transit system's operation in real time. The *"zet_feed_data"* overall contains 35,958,139 records and occupies 5.735 GB of storage. This is a large dataset compared to the static GTFS data and reflects the quantity of real-time updates the system captures.

The earliest recorded event in this dataset has a timestamp of 1690164797, corresponding to July 24, 2023. The latest recorded event has a timestamp of 1692172828, corresponding to August 16, 2023.

The figure 3.17 displays real-time delays (in seconds) of transit vehicles recorded in the *"zet_feed _data"* from July 24, 2023, to August 17, 2023. On the x-axis, time in days is visible, and the y-axis represents the delay in seconds, with negative values indicating vehicles arriving earlier than scheduled.

Some data regarding trams consistently shows a shift towards negative values, with many delays concentrated around -200 seconds. This suggests that trams often arrive earlier than scheduled, likely due to the absence of traffic delays that affect buses. Unlike buses, which can be affected by traffic congestion, trams typically run on dedicated tracks and this enables them to have free passage when traffic jams are frequent.

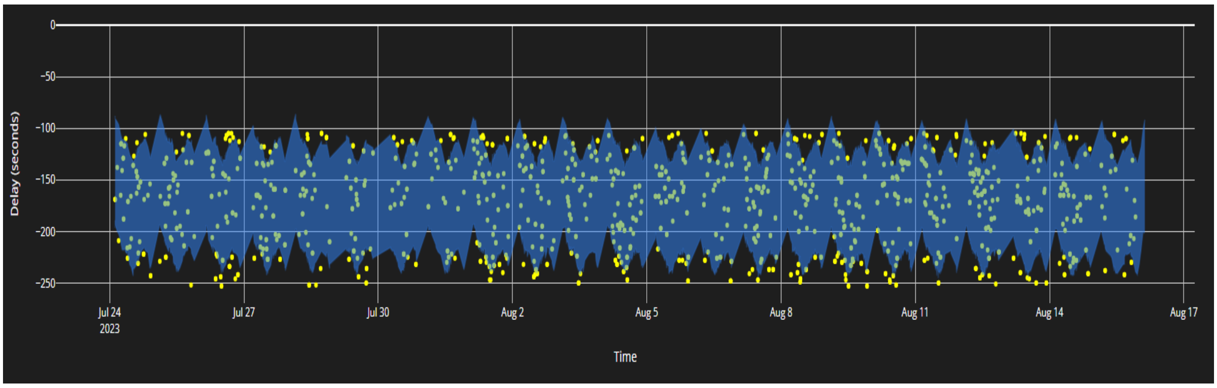


Figure 3.17: All delays in the collected GTFS data for the full period

A structured GTFS database utilizes the three main data handling approaches to ensure the data is readily accessible for analysis and adequately collected. This is important for the next chapter 4 where it will be seen how the data collected for the GTFS database can be used for machine learning to predict trends in delays.

4. Trend Prediction Modeling

Trend prediction is a vital component in public transport since it enables forecasting a series of future delays based on specific historical patterns. If machine learning techniques are applied to the GTFS Time-series data used in the ZET GTFS application, then it is possible to capture patterns and relationships in the data that can make the predictions more accurate.

4.1. Overview of Predictive Modeling

4.1.1. Machine learning for predictive modelling

Predictive modelling uses machine learning algorithms and statistical techniques to specify and create models more accustomed to predicting future outcomes based on historical data. When underlying patterns are detected, the user of the algorithms can predict future events to some extent based on the overall error metric. In public transportation, for example, this can be implemented to predict vehicle speeds or add the prediction to multivariate algorithms to predict service times and delays based on the weather or route conditions.

Machine Learning (ML) overall is a subset of collective artificial intelligence, which focuses on constructing and implementing algorithms that enable computers to learn from chosen data. Making predictions and decisions on the way without being programmed to do so explicitly. As described in predictive modelling, machine learning algorithms check and analyze historical data and develop models to forecast future trends and specific behaviours. Since ML is more flexible than implementing traditional statistical methods, it is also more accustomed to handling large datasets consisting of numerous variables. To fully understand the concept, the machine learning process is divided into multiple steps like in the following:

1) **Data Collection**: data collection is the first foundational step when starting with machine learning, where relevant information like zet_feed_data with its corresponding routes and stops are collected to train the model. For instance, data on stop_directions for specific stops is relevant to distinguish if the vehicle has currently stopped on an inbound or outbound type of stop. Of course, the quality and diversity of the data are important since the model directly learns from it. In translation, the more data you have, the more chance

is there that the model results in more accurate predictions. More diverse data enables the model to understand various patterns and reduces the risk of errors during its operation [20], [21].

2) **Data Preparation**: after data collection, the next step is to analyze it and see if something is wrong with the data. This involves data preprocessing, where inconsistencies and errors are removed, and then, afterwards, the data is split into training and testing sets. The data split usually occurs in the ratio of 80% data used for training and 20% for testing, but sometimes it is also ok to do a 70:30 ratio. This separation is done so that over-fitting does not occur. This happens if the model learns too well but fails to recognize the new data [20], [21].

3) **Choosing the Model**: one of the big aspects is choosing the model to fit the data. Since different models are suited to different data types, for example, the time-series type of data is different from the cluster type data where regression algorithms would work best. It is the same thing that decision trees would be more akin to classification tasks, while neural networks would be a better fit for image recognition tasks. The choice of a model should align with the specific objective a machine learning project brings. The other aspect is the nature of the data since more quality data would presume more accurate predictions [20].

4) **Training**: the training starts when the model has been chosen and the data is ready. With training, the model will identify patterns and relationships between data records, and during this stage, the model can also adjust its parameters iteratively to minimize errors and improve accuracy. This step is time-consuming since it requires multiple iterations, especially for complex models. If the training process is longer and refined, there is more chance that the model can generalize new data, making it very important for trend prediction [20].

5) **Evaluation**: after model training, the result or predictions should be evaluated using the testing dataset, which is just data that the model has not yet seen. This is an important step since, in this way, it can be seen how the model generalizes new unseen data so that it can perform correctly in real-world situations. The evaluation is also used so that overfitting of the model and underfitting could be eliminated. Before deploying a machine learning model, this step must always be passed [20].

6) **Hyperparameter Tuning**: if, from the evaluation, it can be seen that the model's performance is not optimal, the tuning of hyper-parameters is necessary. By tweaking the model's parameters, potentially, some of the accuracy and efficiency is improved, but that can be a long run if the data overall does not have good quality or if the data is just too

stochastic. This iterative process requires going in and out of training multiple times and testing different configurations so that the best combination is found. This minimizes errors in prediction results [20].

7) **Prediction**: after the models has been optimized and all the errors were dealt with as much as possible, the model could be ready for deployment. At this moment the models can then make predictions on new data. This is the resulting phase where the machine learning model can then be used to generate insight and impact on overall decision-making when planning specific things in the future [20].

Types of Machine Learning

There are three main types of machine learning, as visible from figure 4.1, and in most cases, they are used for different purposes, which can be described in the following:

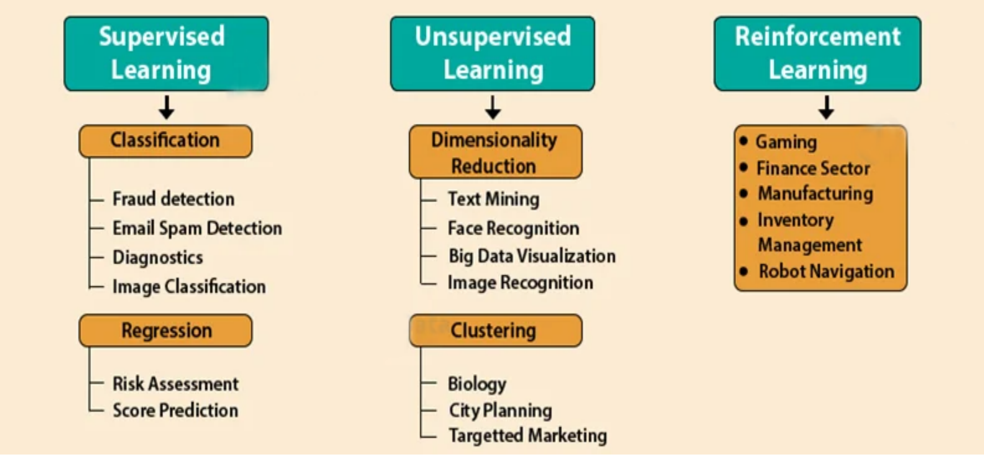


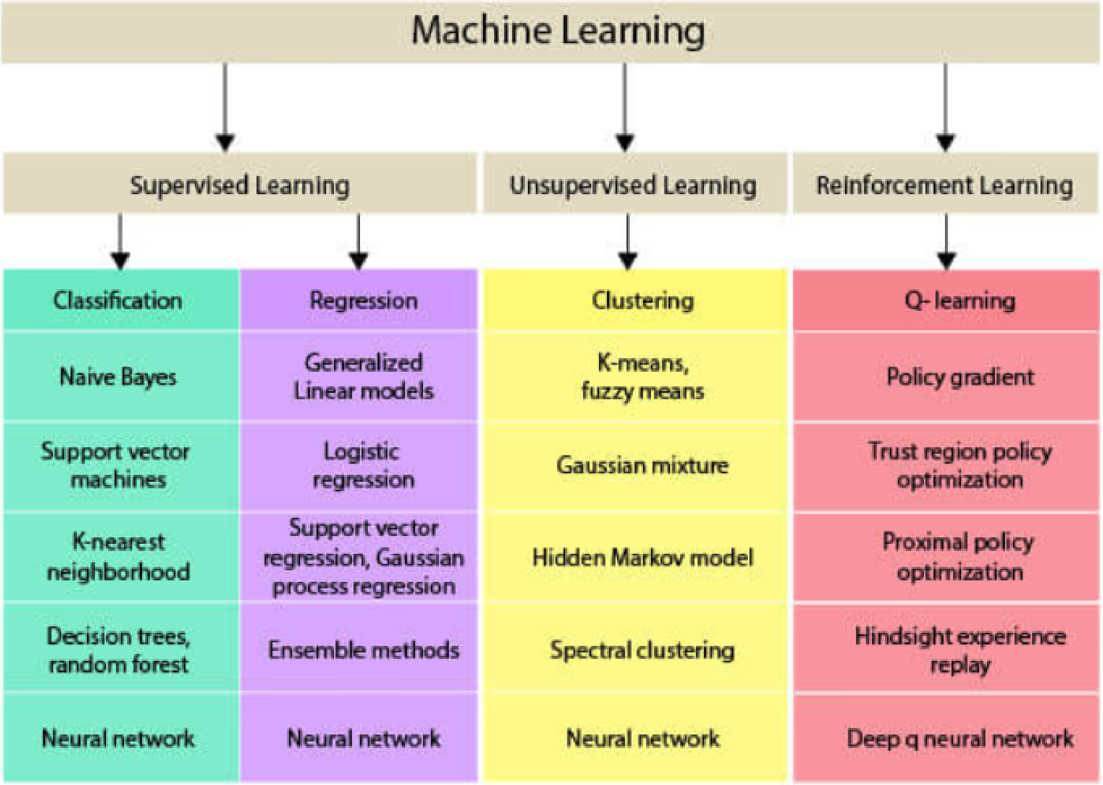
Figure 4.1: Types of Machine Learning [22]

1) **Supervised Learning**: Supervised learning is when training is conducted on labelled data. In this type of data, the correct output is already known for each input, which means that the model learns to map inputs to its outputs, and based on that, it tries to forecast new inputs accurately. This type of learning could be divided into classification (e.g. when trying to detect spam messages or binary patterns) and regression (e.g. commonly used for forecasting sales). This labelled data type helps the model to learn faster and more accurately than other types [21], [22].

2) **Unsupervised Learning**: Unsupervised learning is when some part of the data has no labelled responses. This type of learning tries to identify patterns and relationships within the data by clustering similar items. This is extremely useful when there is a need to discover hidden patterns data which can provide insights in situations when labelled data is unavailable [21], [22].

3) **Reinforcement Learning**: Reinforcement learning is a specific type of machine learning since it requires feedback from the environments and then iterates multiple times with the newfound data to learn better and make better decision. Decisions are based by maximizing rewards trough trial and error. This is commonly used in robotics, gaming and navigation systems where it is extremely important for the model to adapt to changing conditions and also learn over time from new data obtained from environments [21], [22].

Figure 4.2 shows the most common algorithms used for specific types of machine learning. Regarding the time-series data obtained for the ZET GTFS application, the scope adequately falls into the spectre of supervised machine learning models accustomed to time-series data like Facebook's prophet, which will be described more in the next chapter 4.1.2.



Machine Learning

Supervised Learning		Unsupervised Learning	Reinforcement Learning
Classification	Regression	Clustering	Q- learning
Naive Bayes	Generalized Linear models	K-means, fuzzy means	Policy gradient
Support vector machines	Logistic regression	Gaussian mixture	Trust region policy optimization
K-nearest neighborhood	Support vector regression, Gaussian process regression	Hidden Markov model	Proximal policy optimization
Decision trees, random forest	Ensemble methods	Spectral clustering	Hindsight experience replay
Neural network	Neural network	Neural network	Deep q neural network

Figure 4.2: Machine learning algorithms used for specific ML types[23]

4.1.2. Implemented Machine learning model

The ZET GTFS application implements one primary machine learning model best suited for time-series data to predict trends in public transport: Facebook's Prophet. Prophet is best accustomed to time-series forecasting with large quantities of data, which can predict quite well even if missing data exists.

The primary variable used for predictions with Prophet is the delay variable. This variable is important because, if predicted, it can increase the efficiency of the overall public transportation system evaluated. In cities where trams and buses usually operate, through predicting the delay variable, bottlenecks can be identified and used to adjust schedules for better operating service of the transport system. Predicting delays usually helps specify situations where trams or buses may arrive early or late, which can improve service reliability. The delay variable is crucial since it makes optimizing the public transport systems possible, especially when focusing on multi-modal transportation networks. It is also one of the reasons why it was chosen for predictions with the Prophet.

Since this model focuses on scenarios with strong seasonal effects and non-linear trends, like those present with delay prediction when, for example, there is no data between 11:00 PM and 4:20 AM, it is most appropriate to use it here. In figure 4.3, the missing data at night can be seen since the route is inactive at that time. This can present a daily seasonality, but it can only be useful if there is enough data if you want to add additional seasonality to the model. This will be more clearly described in chapter 4.2.2 since adding seasonality is one part of initializing a model.

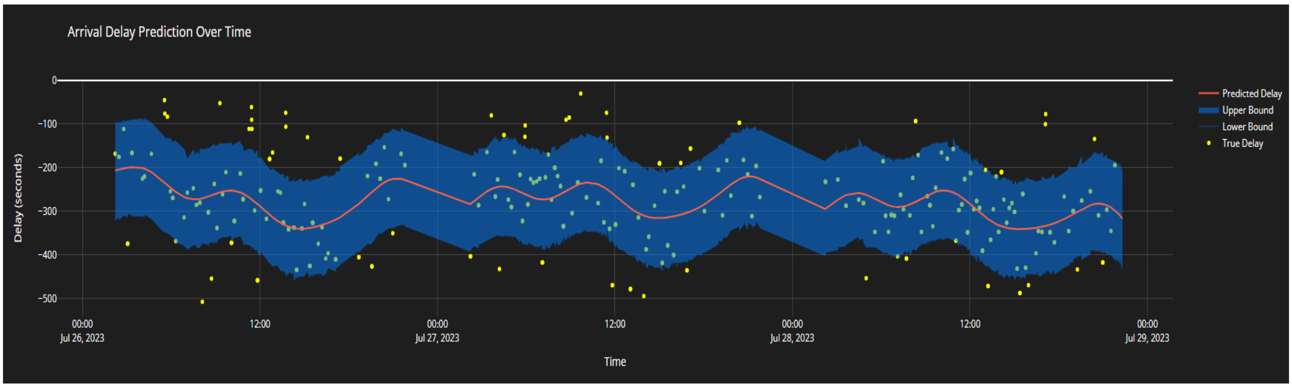


Figure 4.3: Example of Prophet night seasonality pattern for ZET GTFS data

Prophet is a type of additive model in which non-linear trends are fitted with yearly, weekly, and daily seasonality, including holiday effects. Since it has a logistic growth mode, it can easily capture trend changes. Other important factors are the quality and quantity of data on which the model depends.

The time-series data obtained gets segmented into multiple linear segments, where everyone has their growth rate. With their growth rate, these segments are separated but then connected with continuity constraints afterwards, so the connection happens smoothly. Sometimes, when there is a sudden shift in trend, for example, if there are active day-time routes at night, it could adapt very easily.

The following equation represents the general form of Prophet [24]:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t \tag{4.1}$$

Where:

- $g(t)$ is the trend component used to model non-periodic changes in the time series.
- $s(t)$ captures the regular fluctuations, like those occurring weekly or annually.
- $h(t)$ considers the influence of holidays or special events on the time series.
- ϵ_t represents an error term in the model is assumed to follow a normal distribution and represents any unique noise in the data.

Trend Component

A piece-wise linear or logistic growth function can extract the trend component $g(t)$. The piece-wise linear model allows the growth rate to change at specific points, making it suitable for data with sudden trends. In contrast, the logistic growth model is used for data exhibiting saturating growth, such as in population growth models for example [24].

In the following, the piece-wise linear trend model is shown:

$$g(t) = (k + a(t)^T \delta)t + (m + a(t)^T \gamma) \tag{4.2}$$

Where:

- k is the base growth rate.
- $a(t)$ is an indicator vector which indicates if time t has passed a changepoint.
- δ signifies the rate modifications made at every changepoint.
- m and γ are parameters to adjust the offset at changepoints.

Seasonality Component

Seasonality can be represented by using a Fourier series, which enables the model to capture periodic patterns over different periods (e.g. weekly, yearly, etc.) The seasonal component $s(t)$ can be represent by [24]:

$$s(t) = \sum_{n=1}^{N} \left(a_n \cos\left(\frac{2\pi nt}{P}\right) + b_n \sin\left(\frac{2\pi nt}{P}\right)\right) \tag{4.3}$$

Where:

- P is the length of the seasonal pattern (e.g., P = 365.25 for yearly patterns).
- N represents the count of Fourier components.
- a_n and b_n are the coefficients derived from the data.

Holidays and Events Component

Holidays and Special Events Component

Holidays and special events often have a significant impact on time series, causing deviations from typical patterns. Prophet allows users to explicitly include holidays as part of the forecasting model. The holiday effect $h(t)$ is modeled as:

$$h(t) = \sum_i \kappa_i \mathbf{1}_{t \in D_i} \tag{4.4}$$

Where κ_i represents the magnitude of the effect of holiday i on the time series. $\mathbf{1}_{t \in D_i}$ is an indicator function that is equal to 1 when t falls within the set of holiday dates D_i, and 0 otherwise.

Model Fitting

The Prophet models are typically fitted using maximum a posteriori estimation (MAP), a method that combines the likelihood of the data given in the model with prior distributions on the parameters. Fitting the model involves optimizing parameters such as k, m, δ, and the seasonality and holiday effects coefficients to effectively capture the observed data patterns [24].

4.2. Data Preparation and Training

Predictive modelling usually relies on the data type's quality, quantity, and structure to successfully train machine learning models. Proper data preparation is important for the model to learn effectively and produce reliable predictions.

4.2.1. Data Preprocessing

Data preprocessing is when data is transformed from a raw format to a suitable dataset that can be used for modelling. This is usually necessary since real-world data is often incomplete, noisy, and inconsistent. Preprocessing enables the machine learning algorithm to detect patterns much more easily.

In the case of ZET GTFS data, which is public transportation time-series data, it is necessary to be sure that the data is clean from any unnecessary outliers that could skew the model's accuracy. Outliers sometimes GO to extreme measures (e.g. sudden delays of -1498), which are not representative of typical conditions. This kind of outlier tends to distort the trend and seasonality patterns in Prophet, which leads to not-so-accurate predictions, and it is removed with the following approaches:

Outlier detection and removal: In this thesis, Z-score and Interquartile Range (IQR) are used to identify and remove outliers.

- *Z-score Method:* The Z-score method of removing outliers identify these outlier points by measuring how far each one is from the average. So, when every data point is taken and subtracted from the average, it is then divided by the standard deviation (standard spread of all points). So, if a point with a Z-score is too far from zero, it is considered an outlier and removed.

$$Z = \frac{X - \mu}{\sigma} \tag{4.5}$$

  where X represents a data point, μ is the average, and σ is the standard deviation (standard spread of all points).

- *Interquartile Range (IQR) Method:* This method finds outliers by looking at the spread of the middle half of the data. If the 25th percentile (Q1) and the 75th percentile (Q3) are taken, it will calculate the range between them. In some cases, regarding getting better prediction results, the collected data was too stochastic, creating problems for the model when finding patterns. Because of this reason, the interquartile range shrunk, going from 0.33 percentile to 0.58 percentile, which enabled the model to find patterns in data more easily. In this regard, it can be taken that if a data point is lower than $Q1 - 1.5 \times IQR$ or from the other side much higher than $Q3 + 1.5 \times IQR$, then it is considered an outlier. The IQR is :

$$\text{IQR} = Q3 - Q1 \tag{4.6}$$

  where $Q1$ and $Q3$ are the 25th and 75th percentiles of the data.

Handling Missing Values: Regarding the missing NaN values, they were dropped since it is important to maintain data quality, ensuring that the data is complete and reliable. The percentage of "null" values in relevant columns like *"arrival_time"* is 56.85% and *"departure_time"* is 51.96%. This shows that more than half of the collected data was unusable and had to be discarded to prevent inaccurate predictions. Keeping incomplete records could introduce bias or inconsistency in model training, negatively affecting the results, which is why they were dropped in this case. However, this also highlights a need for ZET, as a provider of GTFS data, to improve its data collection processes to ensure better service provision and higher data quality for future analyses.

With the preprocessing steps applied, the dataset becomes much more reliable in this regard, and that enhances the model's performance in detecting patterns in data, reduces noise and results in more accurate predictions. With the time-series data, there is still a limit to how much minimum data you can have, and that is the only aspect that could affect the model; nevertheless, the preprocessing/prefiltering techniques are applied. This should be considered when handling a small amount of time-series data.

4.2.2. Training for the Chosen Model

When training a machine learning model, it is important to fit the overall model to the filtered/preprocessed data to learn the underlying patterns. In the following steps in algorithm 2, the model is initialized with added seasonalities and fitted to preprocessed data to make better predictions.

The train/test ratio was set to 80/20, meaning that 80% of the available data was used for training the model, while the remaining 20% was reserved for testing and evaluating the model's performance. Splitting data with the Prophet is usually done manually, and Prophet cross-validation tools are used to evaluate the prediction.

Algorithm 2 Training Prophet Model for Time Series Forecasting

```
Input: Preprocessed time series data self.filtered_data, forecast period
    forecast_period
Initalize: Trained Prophet model and forecast results
Initialize Prophet model:
model ← Prophet(
    yearly_seasonality=False,
    weekly_seasonality=True,
    daily_seasonality=True,
    changepoint_prior_scale=0.05,
    seasonality_prior_scale=1.2)
                                        # Set seasonality modes and priors
Add custom seasonalities if needed (in case of datasets that have less than a month of
    data, there is no need):
model.add_seasonality(
    name='daily',
    period=1,
    fourier_order=10)

Fit model to data:
model.fit(self.filtered_data[['ds', 'y', 'cap', 'floor']])
                                        # Train model using preprocessed data
Create future data frame and make predictions:
future ← model.make_future_dataframe(
    periods=forecast_period,
    freq='H')
                                        # Generate future dates for prediction
future['cap'] ← 400
future['floor'] ← -400
forecast ← model.predict(future)
                                        # Predict future values using trained model
forecast
                                        # Return forecast results
```

The parameters show that the Prophet object is initialized with weekly and daily seasonality. Still, based on the amount of collected data, there is no need to use this parameter, meaning they could be ignored. The "changepoint_prior_scale" was set to 0.05 to better fit the smaller amount of collected data, and the fitting frequency was set to hours or "H" in this case. The future flooring and capping were necessary since sometimes there were situations when the prediction would overfit and lose stability when predicting the forecast, as shown in the figure 4.4.

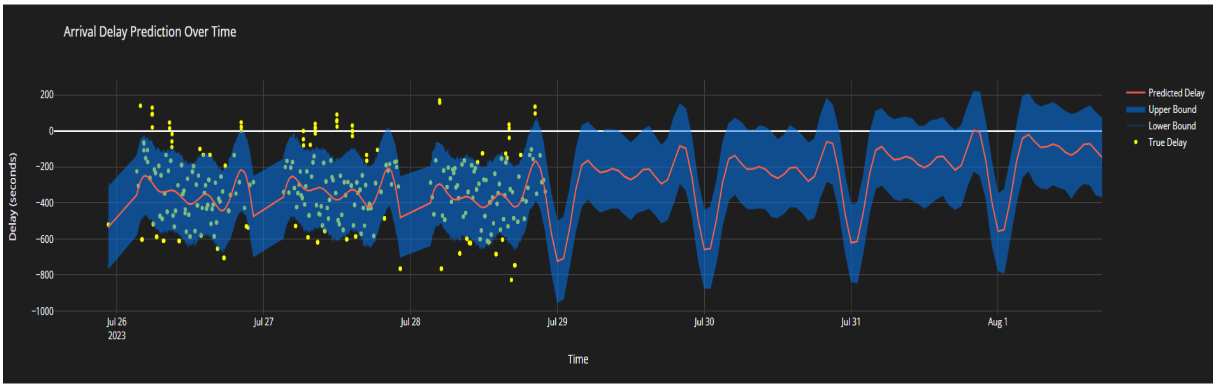


Figure 4.4: Example of Prophet instability with a added yearly seasonality for ZET GTFS data

If monthly and yearly seasonality with "model. add_seasonality" is added, the data amount should be at least 2 months of recordings. So, seasonality is only applicable when more data is in the dataset. Unfortunately, in this case, when only 3 weeks were collected, there is no need to use this. It can be commented out (only shown here as an explanatory reference).

4.2.3. Evaluation Metrics

When assessing any predictive model's performance, it is always important to use specific evaluation metrics. These metrics offer insight into how well the model recognizes patterns and generalizes new data. Choosing appropriate metrics is also important since they reflect variability in data, especially if the data contains noise or outliers.

- **Mean Squared Error (MSE):** The MSE calculates the average of the squared differences between the predicted and observed values, providing an overall evaluation of the model's performance, with particular sensitivity to outliers [25]:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.7}$$

  where y_i is the actual value and $\hat{y}_i$ is the predicted value.

- **Root Mean Squared Error (RMSE):** RMSE is the square root of the MSE, offering a measure of error in the same units as the target variable [25]:

$$\text{RMSE} = \sqrt{\text{MSE}} \tag{4.8}$$

- **Mean Absolute Error (MAE):** MAE calculates the average of the absolute differences between the predicted and observed values, providing an intuitive measure of average error [25]:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{4.9}$$

- **Mean Absolute Percentage Error (MAPE):** MAPE expresses the error as a percentage, showing the accuracy of a forecast in relative terms [25]:

$$\text{MAPE} = \frac{100\%}{n}\sum_{i=1}^{n} \left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{4.10}$$

- **Symmetric Mean Absolute Percentage Error (SMAPE):** SMAPE is a variation of MAPE that accounts for both over- and under-predictions, providing a more balanced error measurement [26]:

$$\text{SMAPE} = \frac{100\%}{n}\sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{(|y_i| + |\hat{y}_i|)/2} \tag{4.11}$$

Only four are shown in the ZET GTFS application: the RMSE, MAE, MAPE, and SMAPE. This decision occurred because of the data quality, and at some stops when trying to predict a forecast, an extreme MSE metric error was calculated, which, when compared to the other metrics, would skew the graph.

After implementing the database, getting the data, preprocessing and learning the model, it is now important to take a look in the next chapter 5 at the finished application with the implemented machine learning model to see the overall functionality of the ZET GTFS application and also evaluate the predictive model performance based on the acquired data.

5. Results and Analysis

5.1. Real-time Data Visualization

One of the main goals of this thesis is to visualize the ZET GTFS data in real time. It should provide users with a dynamic and interactive map-based interface to view the current status of public transportation regarding trams and buses in Zagreb, as well as record and evaluate collected data. The code for real-time data processing and visualization is available at the following link: GTFS Master Thesis Code.

5.1.1. User Interaction with the Application

The application's design was already described in the chapter 3.2.1. It was divided into three primary tabs: map, options, and info. Each tab communicates with the other when interacting with it since updating data on the Map and Info tab requires interaction from the Options tab, and every one of them complements each other. To better describe this, the specific tab interactions were divided by its tab in the following:

Map Tab Interaction

The main feature of the ZET GTFS application is that it visualizes real-time data of vehicles' current positions on stop arrival directly on the interactive map, which, when it first loads, shows all of the tram and bus collected public transport network visible in figure 5.1. The positions of public transport vehicles are filtered trough selected routes. Specific locations of trams and buses can be seen, which are represented by yellow for trams and blue for buses.

If clicked, the icons will also present details such as the route's name, stop name, and arrival times (see Figures 5.2 and 5.3). This tab also allows users to select specific stops for later processing and evaluation by circling with the polygon tool around the area of interest. This, in a way, allows the user to customize the view of the transit network by data filtering with spatial selection.

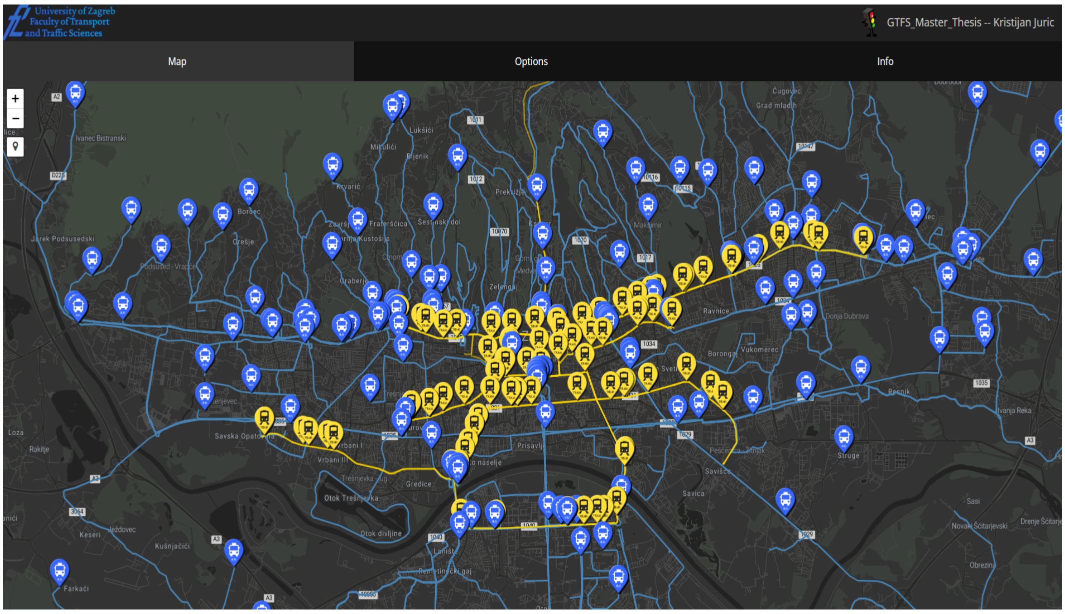


Figure 5.1: Map Tab on the first load showing all the tram and bus networks in Zagreb

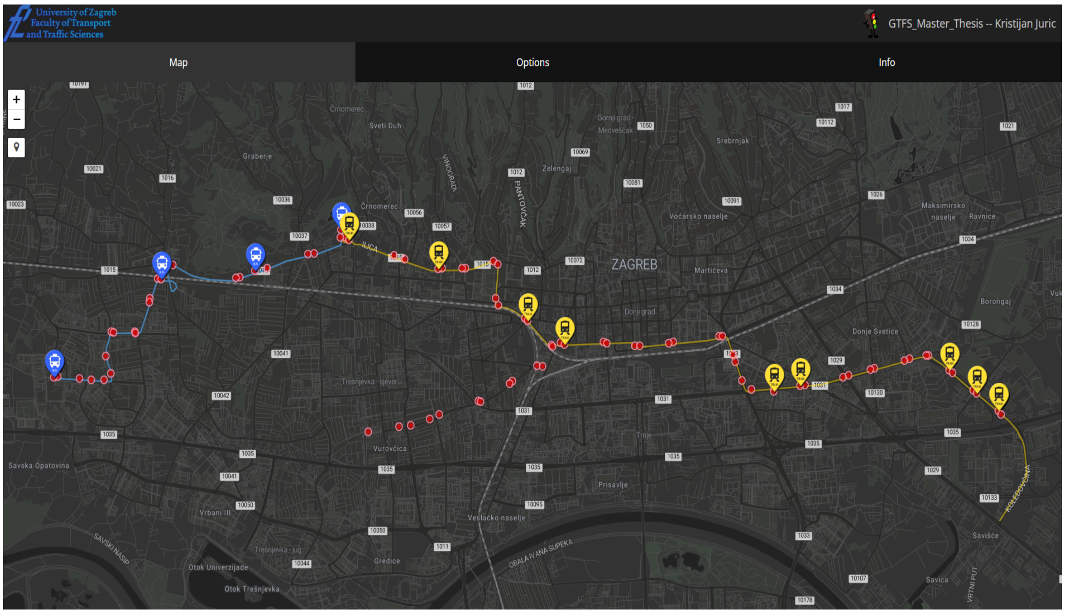
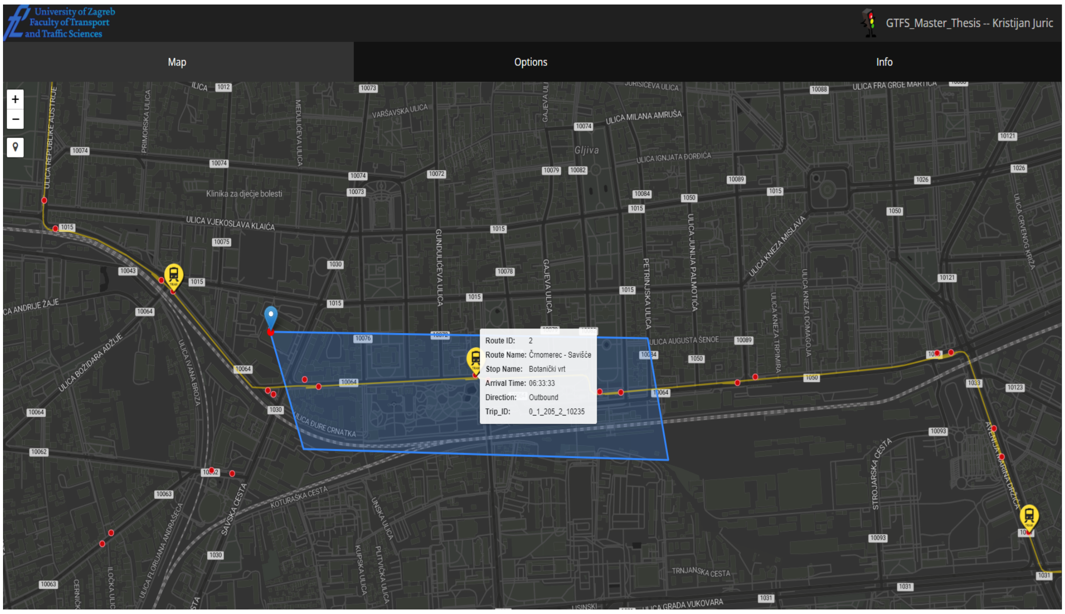


Figure 5.2: Map Tab with chosen routes

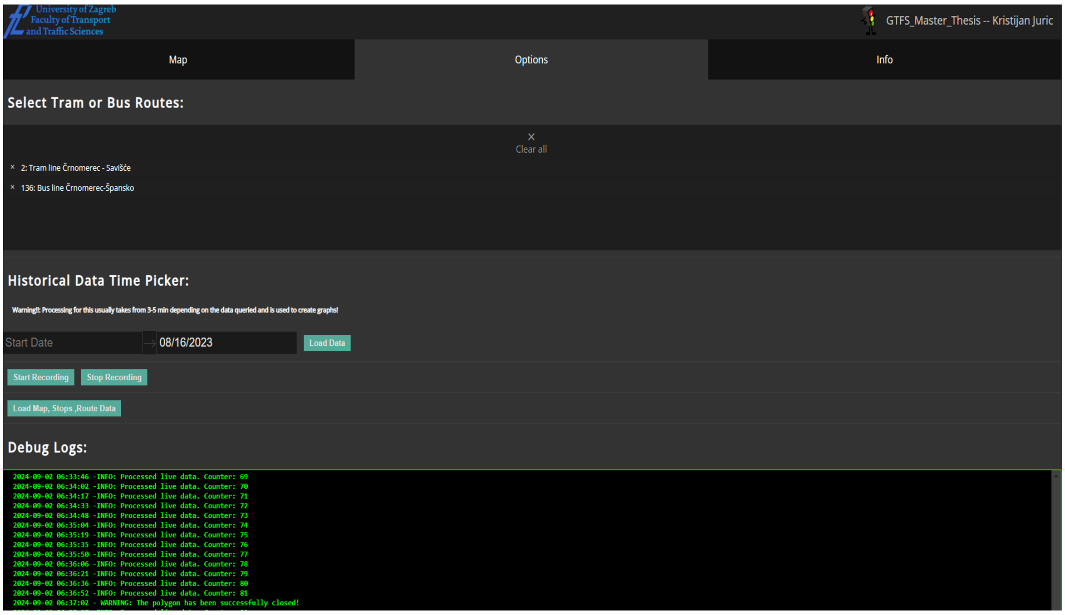


Route ID: 2
Route Name: Črnomerec - Savišće
Stop Name: Botanički vrt
Arrival Time: 06:33:33
Direction: Outbound
Trip_ID: 0_1_205_2_10235

Figure 5.3: Polygon selection for stops and a view of the additional details of the vehicle on stop arrival

Options Tab Interaction

The options tab usually communicates with other tabs either through filtering the coordinates of the routes specified by the drop-down visible on figure 5.4 to parse it to the Map tab for visualization or by executing query-s already described in the chapter 3.3.3. The queries are executed with recording and load buttons, which can stop and start GTFS data collection, load static data, and, through the "Load Data" button, create a database file called "store_dataframe", which will be used by the Info tab to create evaluations and trigger machine learning models to use it for forecasting after preprocessing.



Select Tram or Bus Routes:

× 2: Tram line Črnomerec - Savišće
× 136: Bus line Črnomerec-Špansko

Historical Data Time Picker:

Warning!!: Processing for this usually takes from 3-5 min depending on the data queried and is used to create graphs!

Start Date → 08/16/2023

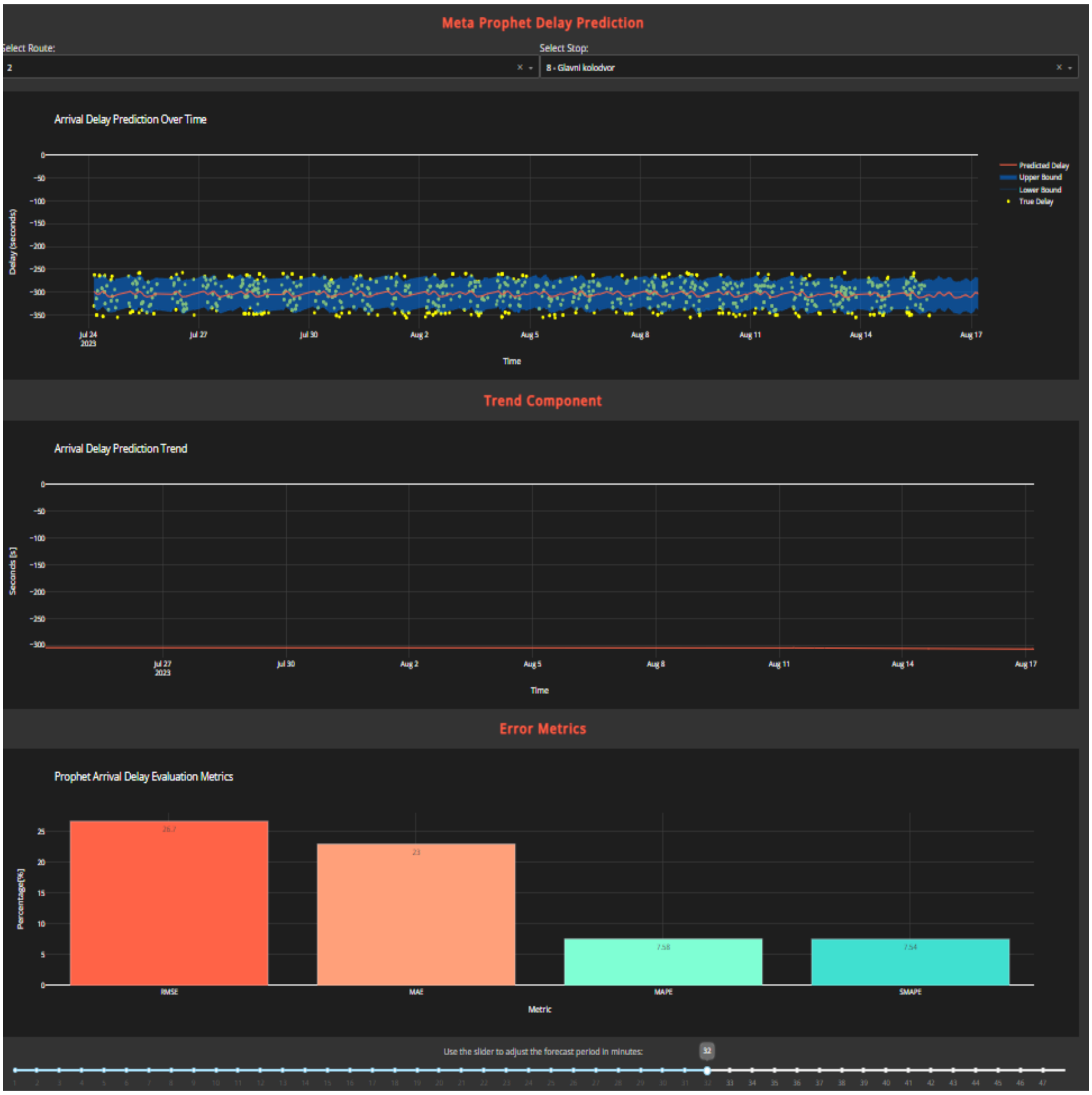
Debug Logs:

```
2024-09-02 06:33:46 -INFO: Processed live data. Counter: 69
2024-09-02 06:34:02 -INFO: Processed live data. Counter: 70
2024-09-02 06:34:17 -INFO: Processed live data. Counter: 71
2024-09-02 06:34:33 -INFO: Processed live data. Counter: 72
2024-09-02 06:34:48 -INFO: Processed live data. Counter: 73
2024-09-02 06:35:04 -INFO: Processed live data. Counter: 74
2024-09-02 06:35:19 -INFO: Processed live data. Counter: 75
2024-09-02 06:35:35 -INFO: Processed live data. Counter: 76
2024-09-02 06:35:50 -INFO: Processed live data. Counter: 77
2024-09-02 06:36:06 -INFO: Processed live data. Counter: 78
2024-09-02 06:36:21 -INFO: Processed live data. Counter: 79
2024-09-02 06:36:36 -INFO: Processed live data. Counter: 80
2024-09-02 06:36:52 -INFO: Processed live data. Counter: 81
2024-09-02 06:37:02 - WARNING: The polygon has been successfully closed!
```

Figure 5.4: Options Tab Overview

Info Tab Interaction

The Info tab, when interacted with, displays graphs based on the "store_dataframe" historical data located in the application's .csv database. It is primarily used for data analysis. Users can filter on request through the drop-down buttons, which select a specific stop to be evaluated. This gives the user insight into the performance of the implemented predictive model and its forecast accuracy through error metrics.



Meta Prophet Delay Prediction

Select Route: 2

Select Stop: 8 - Glavni kolodvor

Trend Component

Error Metrics

Use the slider to adjust the forecast period in minutes: 32

Figure 5.5: Info Tab Overview

Since the application was designed to be handled as a SPA, as explained in chapter 3.2.1, it makes things much faster when switching through the aforementioned tabs since the interaction occurs within the same window where the data gets updated dynamically.

The overall functionalities of the application do perform well and do provide a seamless interface for tracking public transport in real-time, but of course, in every application, there exist some problems, either it is because of bugs or just data limitations, which will be discussed in the chapter 5.1.2.

5.1.2. Application Troubleshooting

Although the GTFS application performs well in most cases, users could occasionally experience issues, especially when interacting more with the data shown on the Map tab. One of the main problems is the lagging and slow responsiveness when displaying a batch of routes selected from the drop-down menu on the Options tab.

This usually happens because of the high volume of data being processed and rendered simultaneously, especially when multiple routes are selected (more than 4 routes simultaneously). This can also happen if some complicated polygons are made to select outlier stops on the city's edges.

Users could select fewer routes to address these performance issues, reducing the data processing load and improving the application's responsiveness. Also, selecting far-fetched areas with stops is not advised when using the polygon tool. If more stops are included in the area of interest, it will not affect the processing much. Still, if the areas are far from each other (when the polygon is stretched out), the historical data retrieval will be slowed down.

Users can refer to the debug log feature at the bottom of the application's interface on the Options tab if some specific lagging occurs. This makes it easier to identify whether the ZET GTFS application is fetching new data or if there is an error in the processing pipeline.

5.2. Predictive Model Performance

The predictive performance of a model is measured based on the error metric described in the chapter 4.2.3, and it analyses how well a model predicts or is fitted correctly. Focusing on the data records per stop, a preprocessed ZET GTFS data collection was introduced to the machine learning model, which spans from the 24th of July to the 16th of August 2023. This data collection was already filtered out from the global data frame and only made it specified for one route specifically. To understand the model's effectiveness in forecasting delays, it was also important to fit the model correctly, so sometimes that means changing the upper and lower IRQ boundaries and similar so that the models fit the bets to get more accurate results.

5.2.1. Data Records per Stop

In the case of this evaluation, two tram routes (2, 7) and two bus routes (136, 109) were used. The number of data records/points can be seen from figure 5.7 to figure 5.13 for each stop along different routes.

Route 2 (Tram)

Route 2, as shown in figure 5.7, runs from *Črnomerec* to *Žitnjak*. Based on the data collected, the stops with the highest data density include *Glavni Kolodvor*, *Vodnikova*, *Držićeva*, and *Branimirova Tržnica*. The stop with the most recorded data points is *Glavni Kolodvor*, which appears across all recorded data. These stops have between 3000 to 4000 data records each, making them suitable for forecasting. However, the quality of data and seasonal variations must also be considered.

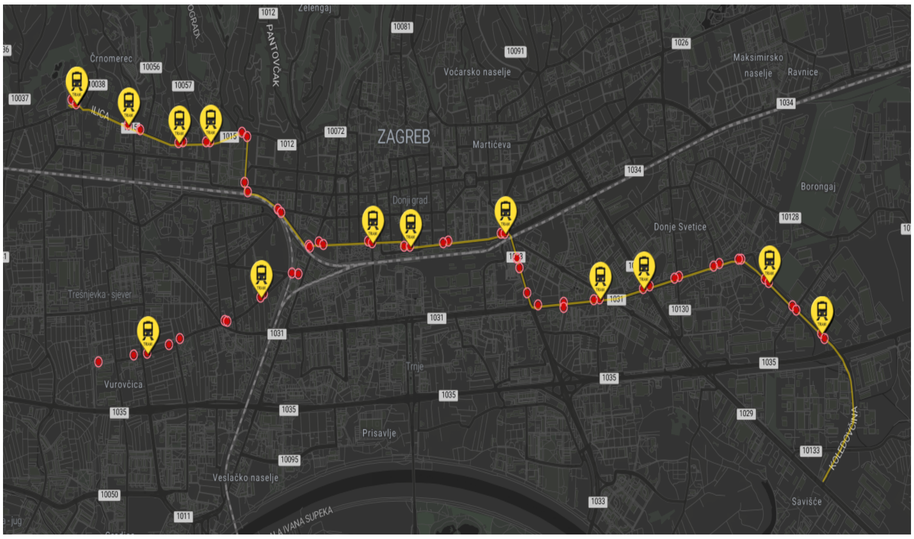


Figure 5.6: Geographical view of route 2

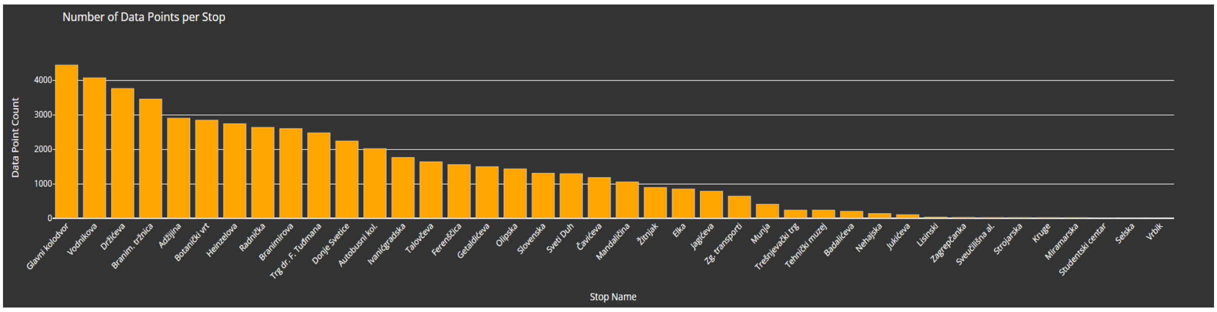


Figure 5.7: Data Records per Stop for Route 2

Route 7 (Tram)

Route 7, illustrated in figure 5.9, travels between *Savski Most* and *Dubec*. The busiest stops in terms of data collection include *Tržnica Kvatrić*, *Mašićeva*, and *Trg P. Krešimira*. These stops have a significant amount of data recorded, with 2000 to 3000 data points each. As with Route 2, there are several stops with low data collection, indicating that some parts of the route were not consistently recorded.

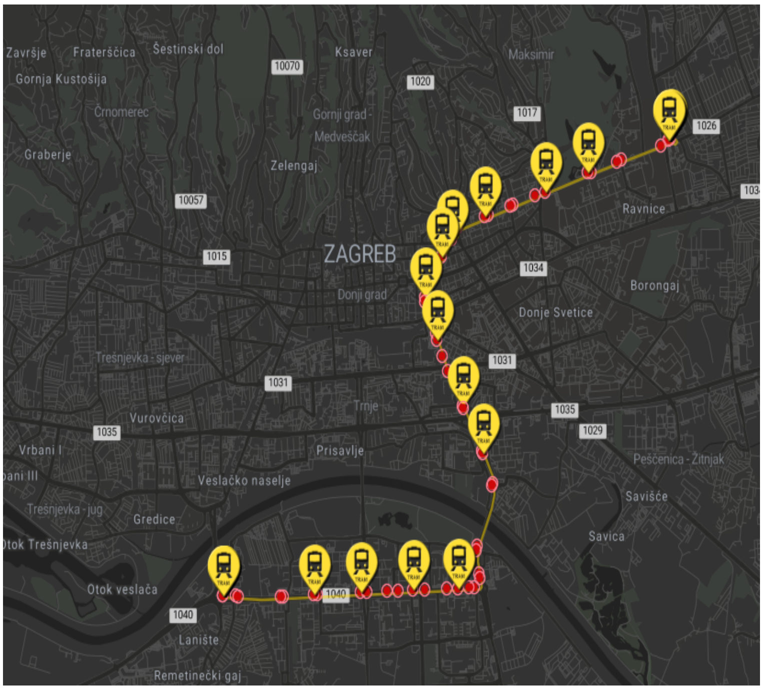


Figure 5.8: Geographical view of route 7

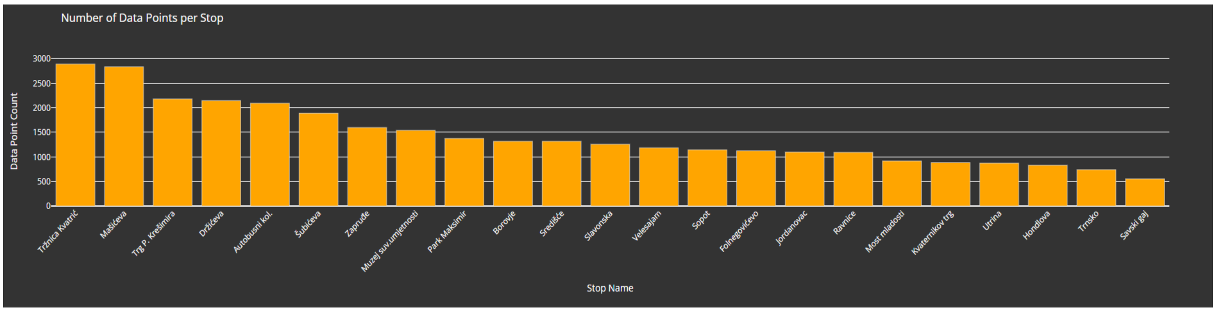


Figure 5.9: Data Records per Stop for Route 7

Route 136 (Bus)

Route 136 is a bus route that connects *Črnomerec* and *Špansko*, as seen in figure 5.11. The data points show that stops such as *Vrapčanska* and *I. Brlić Mažuranić* have the highest number of data points collected, with over 1000 records each. This indicates a high frequency of recordings at these stops compared to other bus routes.

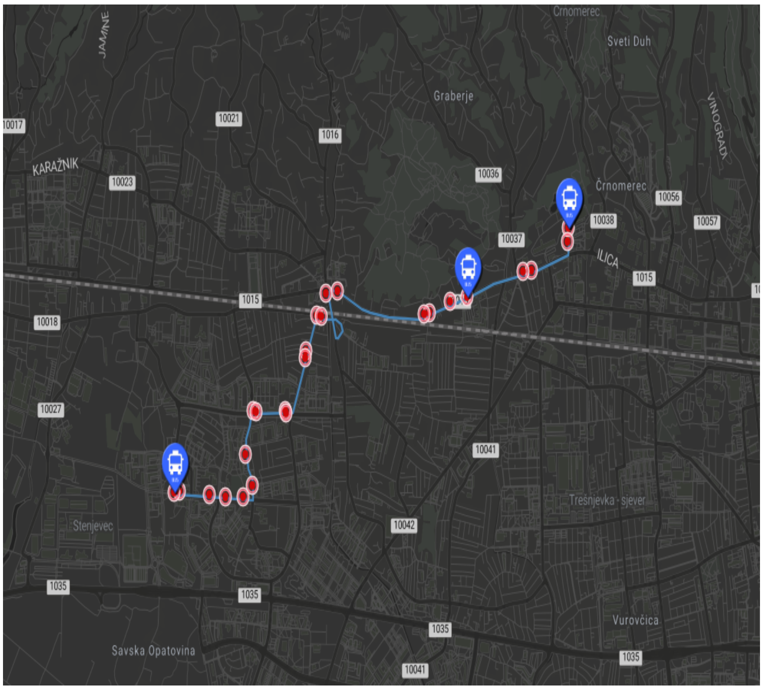


Figure 5.10: Geographical view of route 136

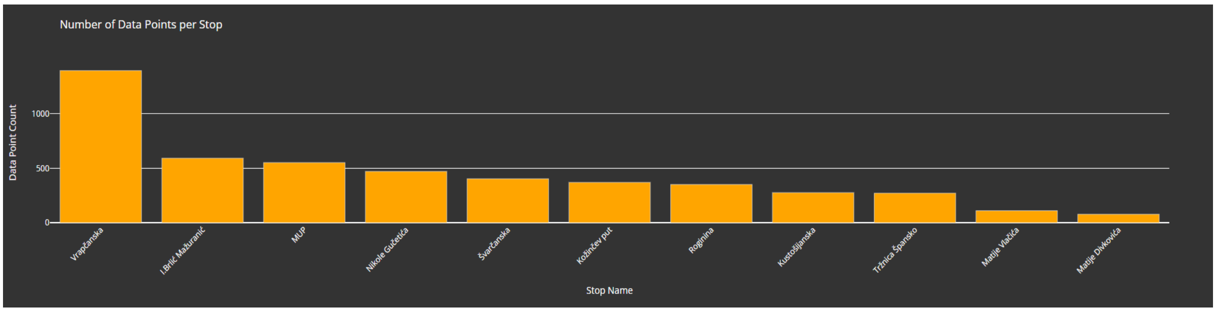


Figure 5.11: Data Records per Stop for Route 136

Route 109 (Bus)

Route 109, depicted in figure 5.13, connects *Črnomerec* and *Dugave*. The data shows that stops such as *Središće* and *Baštijanova* were recorded the most frequently, with around 1000 data points each. Several stops along this route, such as *Bolšićeva*, have very few data points, which could affect the reliability of predictions if used, and in these regards, are ignored.

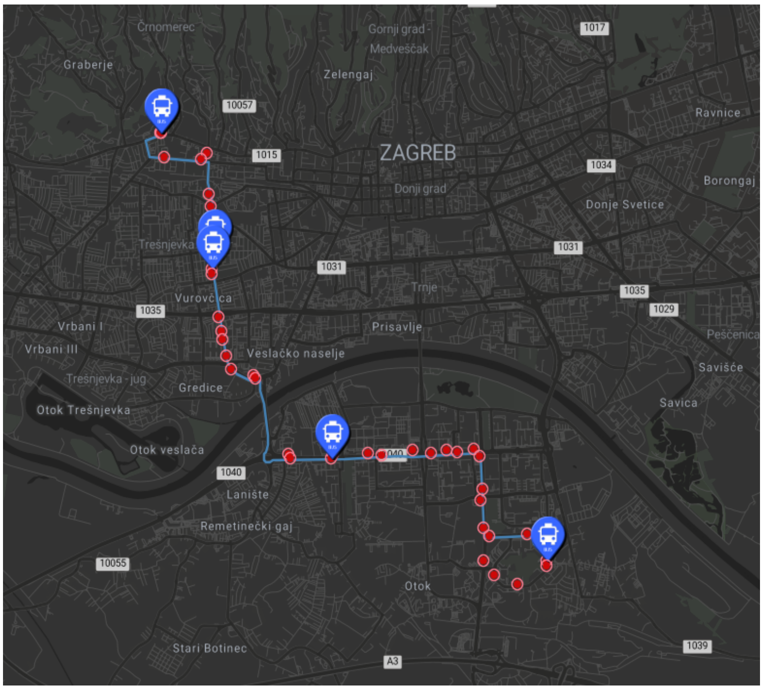


Figure 5.12: Geographical view of route 109

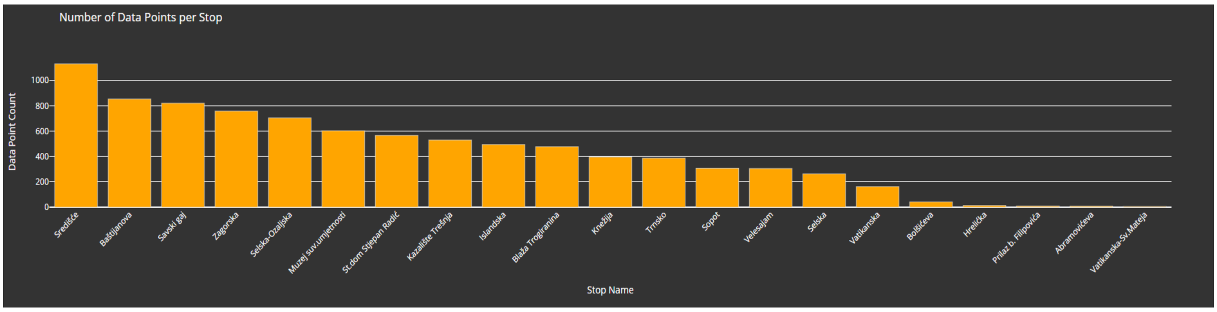


Figure 5.13: Data Records per Stop for Route 109

For all four routes, certain stops have a high density of recorded data points, making them more suitable for forecasting and analysis. However, some stops with low data collection were excluded from the analysis to maintain data integrity and avoid bias in the results. The charts clearly indicate that the density of data points varies significantly across different routes and stops.

5.2.2. Prediction Error Metrics

For each route, specific error metrics are included to evaluate prediction accuracy. In the case of this thesis, **RMSE**, **MAE**, **MAPE**, and **SMAPE** were used to evaluate the model's ability to accurately and correctly predict arrival delays. The following tables summarise prediction error metrics for each chosen route from *Table 5.1* to *Table 5.4*.

The analysis of *Route 2* data shows that the model delivers its best performance at *Glavni Kolodvor*, producing the lowest **RMSE** (26.7s), **MAE** (23s), and **MAPE** (7.58%). The large amount of available data at this stations probably contributes to this, allowing the model to understand the patterns of delays. Additionally, stops such as *Vodnikova* and *Branimirova Tržnica* display higher errors, with *Vodnikova* registering an **RMSE** of 51.94s and a **MAPE** of 11.96%. The model's accurate prediction may be challenged by external factors like traffic congestion or delays at crossings, for example.

Table 5.1 Prediction Performance Metrics for Route 2

Station	**RMSE (s)**	**MAE (s)**	**MAPE (%)**	**SMAPE (%)**
Glavni Kolodvor	26.7	23	7.58	7.54
Vodnikova	51.94	44.97	11.96	11.72
Držićeva	37.59	32.15	10.34	10.19
Branimirova Tržnica	37.06	31.42	10.66	10.47

The prediction performance for *Route 7* remains fairly consistent, with *Tržnica Kvatrić* and *Trg P. Krešimira* exhibiting **MAPE** values near 10%, suggesting that the model effectively predicts delays for these stops. The lower errors at these stops may be attributed to regularity in traffic flow and stop usage along these segments. Additionally, *Mašićeva* and *Držićeva* display higher errors, especially *Mašićeva* with a **MAPE** of 15.87%. These stops could be more vulnerable to unpredictable delays, possibly due to their proximity to high-traffic areas or frequent disruptions along the route, resulting in greater variability in arrival times.

Table 5.2 Prediction Performance Metrics for Route 7

Station	**RMSE (s)**	**MAE (s)**	**MAPE (%)**	**SMAPE (%)**
Tržnica Kvatrić	23.35	19.75	10.76	10.54
Mašićeva	30.84	26.65	15.87	15.48
Trg. P.Krešimira	23.58	20.07	10.87	10.73
Držićeva	35.58	30.38	15.32	14.93

The performance of the model on *Route 136* is good, particularly at stops like *Vrapčanska*, where the values for **RMSE** and **MAE** are relatively low (11.84s and 10.04s, respectively). The **MAPE** is also at a moderate level, indicating a reasonable level of prediction accuracy. However, stops such as *Nikole Gučetića* show a high **MAPE** (19.02%), suggesting that the model's predictions for these stops are less dependable. This might be due to unpredictable factors such as road conditions or variations in bus operations, which pose challenges for the model to accommodate.

Table 5.3 Prediction Performance Metrics for Route 136

Station	**RMSE (s)**	**MAE (s)**	**MAPE (%)**	**SMAPE (%)**
Vrapčanska	11.84	10.04	14.86	14.56
Ivane Brlić Mažuranić	15.57	13.52	14.66	14.88
MUP	7.21	6.21	5.36	5.33
Nikole Gučetića	8.39	7.4	19.02	18.02

The performance metrics for *Route 109* indicate a combination of outcomes. *Središće* demonstrates the most favorable error metrics, achieving a **MAPE** of 5.54%, highlighting its precise prediction capabilities. Furthermore, Zagorska and similar stops show higher error values, with a MAPE of 21.66%, which illustrates the challenges of predicting delays in the inconsistent parts of the route. These difficulties may arise from traffic congestion, which leads to irregular delays.

Table 5.4 Prediction Performance Metrics for Route 109

Station	**RMSE (s)**	**MAE (s)**	**MAPE (%)**	**SMAPE (%)**
Središće	15.58	12.74	5.54	5.52
Baštijanova	26.7	22.53	13.13	12.76
Savski Gaj	33.13	28.43	9.65	9.54
Zagorska	33.35	28.4	21.66	20.16

5.2.3. Prophet Arrival Prediction Graphs for Specified Routes

Regarding the forecast of arrival delay by the Prophet model for specific stops on every route, the predictive accuracy can be observed in the figures labeled 5.14 through 5.17. These visual representations portray the real delays (depicted by yellow points) in comparison to their predicted delays (illustrated by the red line), providing insights into the model's precision for each route based on the data points outlined in 5.2.1.

At *Črnomerec - Savišće*, *Figure 5.14* depicts *Route 2*. The Prophet model effectively captures the predominant trend of negative delays, indicating early arrivals. The trend closely matches the red predicted line, with a difference of about -200 seconds, corresponding to the early arrivals in the data. The model's confidence intervals, represented by the blue bounds, indicate its relative confidence in the predictions, as they tightly envelop the predicted values.

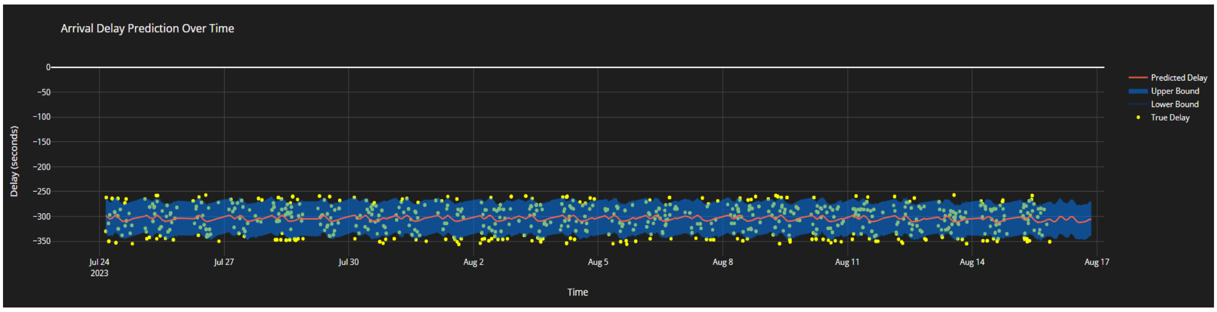


Figure 5.14: Prediction Graph for Route 2 – Črnomerec - Savišće

At *Savski Most - Dubrava*, the graph in *Figure 5.15* demonstrates that the model effectively tracks the overall delay trend, despite a few outliers. The red line representing predicted delays consistently aligns with the yellow data points reflecting recorded delays, indicating a reasonable level of accuracy. Additionally, the blue area representing the prediction interval remains consistent, suggesting that the Prophet model has access to quality data for generating confident predictions for this stop.

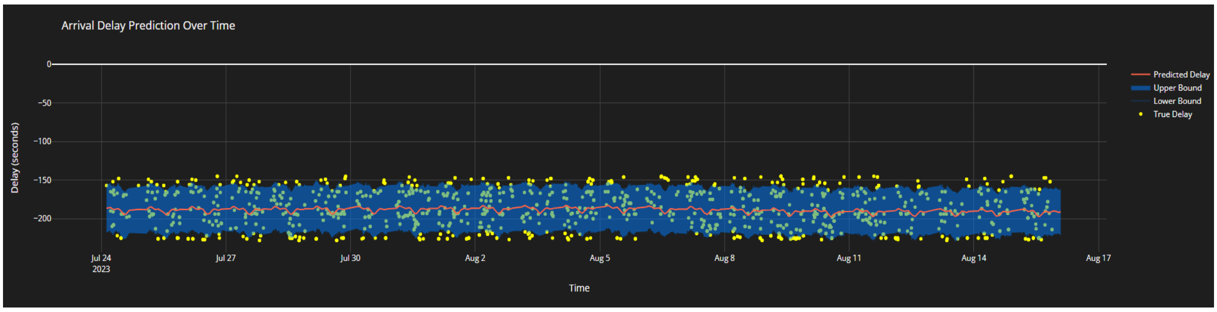


Figure 5.15: Prediction Graph for Route 7 – Savski Most - Dubrava

In *Figure 5.16*, the model displays a precise prediction for *Route 136* at *Črnomerec - Španisko*, with a reduced number of significant outliers. The narrower prediction bounds indicate greater confidence in this specific forecast. The recorded delays consistently show positive values, indicating that vehicles on this route typically arrive after the schedule, a pattern successfully captured by the model.

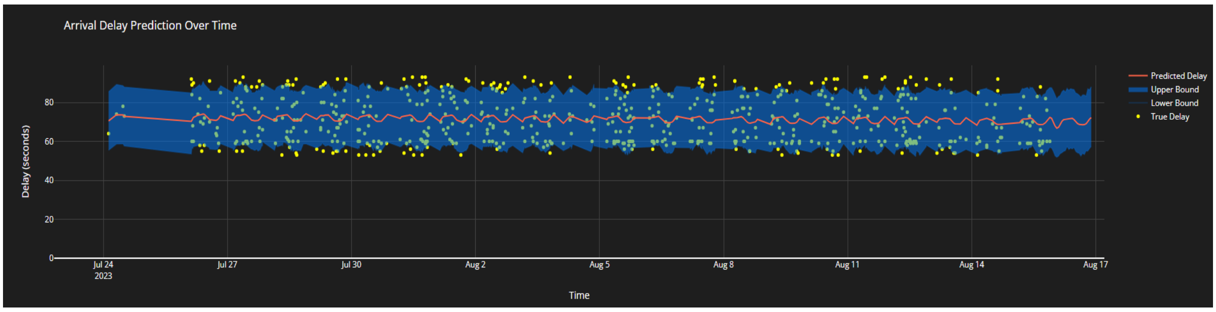


Figure 5.16: Prediction Graph for Route 136 – Črnomerec - Špansko

In conclusion, with regards to *Route 109* at *Črnomerec-Dugave* as depicted in *Figure 5.17*, the model effectively monitors the overall delay pattern, although the prediction range is slightly wider compared to other routes. This may indicate a higher degree of uncertainty in the data, potentially influenced by external factors impacting this specific stop. Nevertheless, the predicted line (in red) closely aligns with the general trend of recorded delays, indicating that the model continues to offer valuable predictions.

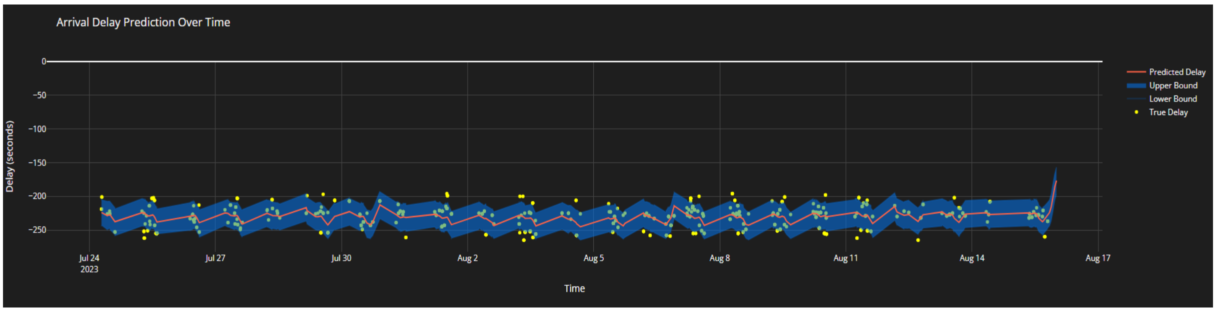


Figure 5.17: Prediction Graph for Route 109 – Črnomerec-Dugave

6. Conclusion

The thesis examined the creation of an application to visualize real-time transit data for Zagreb and use machine learning models to predict transit vehicle arrival times. It involved gathering and processing GTFS data for static and real-time feeds, focusing on the city's key tram and bus routes. The real-time data visualization enabled users to dynamically engage with the public transportation system, providing insights into vehicle positions and delays throughout the network. While the app effectively delivered real-time data and was responsive, it encountered performance challenges when handling large data volumes or complex spatial queries. These challenges underscored the importance of enhancing data optimization techniques and improving the handling of complex spatial queries for smoother user interaction.

When looking through the point of predictive modelling, the Prophet time-series forecasting model played an important role in anticipating arrival delays. Various error metrics, including **RMSE**, **MAE**, **MAPE**, and **SMAPE**, were used to evaluate the model's performance, revealing specific variations in results across different routes and stops. Stops with dense data, such as *Glavni Kolodvor* on *Route 2*, showed lower error rates and more accurate predictions. However, the model encountered challenges with routes that had sporadic or inconsistent data, such as *Route 109*, resulting in higher prediction errors. These findings emphasized the essential need for comprehensive and high-quality data to achieve accurate forecasting. This is a problem since the collected data overall has too much NaN values and the GTFS service provider which in this case is ZET should calibrate the sensors or enable adequate parsing of the data so that it can be continuous and not often broken.

In addition, an important lesson was learned during the app's development: the significance of designing a modular system. At first, the absence of modularity in the app's architecture led to difficulties in expanding functionalities, such as adding more machine learning models or logging features. As time passed, it became clear that implementing a modular system with multiple layers enabled overall development and simplified maintenance, enabling the integration of new features. This provided a clear framework for debugging and updates.

To conclude, the ZET GTFS app successfully demonstrated its ability to track public transportation in real-time, specifically public transportation vehicles like trams and buses, on stop arrivals and predict delays. However, specific areas can be improved, particularly in managing larger datasets, which means recording more data over a longer period of time to see if more seasonality can be added to the Prophet model and additionaly if more features can be added to the overall data.

BIBLIOGRAPHY

[1] Biondić P. Implementacija GTFS-a za pomorski javni prijevoz. University of Rijeka, Faculty of Maritime Studies; 2023. Downloaded from: https://urn.nsk.hr/urn:nbn:hr:187:354997.

[2] Galac D, Carić T. Izrada multimodalnog planera putovanja za područje grada Zadra. University of Zagreb; 2015. Downloaded from: https://repozitorij.fpz.unizg.hr/islandora/object/fpz:747.

[3] Šarić A. Obrada i vizualizacija podataka pružatelja javnog gradskog prijevoza u realnom vremenu. Fakultet prometnih znanosti; 2023. Downloaded from: https://repozitorij.fpz.unizg.hr/islandora/object/fpz:3036.

[4] Google Developers. General Transit Feed Specification (GTFS) Reference; 2024. https://developers.google.com/transit/gtfs/reference. Accessed: 2024-08-24.

[5] Uravić M, Brčić D. Analiza prometno-prostornog planiranja u Gradu Zagrebu; 2017. Downloaded from: https://urn.nsk.hr/urn:nbn:hr:119:277205.

[6] Majstorović I, Ahac M, Ahac S. The City of Zagreb Lower Town Urban mobility development program. Transportation Research Procedia. 2022;60:362–369. Downloaded from: https://doi.org/10.1016/j.trpro.2021.12.047.

[7] Klarić D. Analiza javnog gradskog prometa u Zagrebu; 2020. Downloaded from: https://repozitorij.fpz.unizg.hr/islandora/object/fpz%3A1962.

[8] Dragicevic M. Real-Time GTFS Data Dashboard in Python; 2022. Medium. Downloaded from: https://medium.com/@mladen.dragicevic/real-time-gtfs-data-dashboard-in-python-209801ba32f1.

[9] rep hr. ZET testira podatke o pozicijama tramvaja u stvarnom vremenu; 2024. https://rep.hr/vijesti/mobiteli/zet-testira-podatke-o-pozicijama-tramvaja-u-stvarnom-vremenu/8538/. Accessed: 2024-08-24.

[10] Transmodel. SIRI - Service Interface for Real-Time Information; 2024. https://transmodel-cen.eu/index.php/siri/. Accessed: 2024-08-24.

[11] Normes Donnees TC. SIRI - Service Interface for Real-Time Information (Part 1); 2015. http://www.normes-donnees-tc.org/wp-content/uploads/2015/04/SIRI-part1.pdf. Accessed: 2024-08-24.

[12] NeTEx. NeTEx Getting Started White Paper (Version 1.06); 2015. https://netex-cen.eu/wp-content/uploads/2015/12/02.NeTEx-GettingStarted-WhitePaper_1.06.pdf. Accessed: 2024-08-24.

[13] GeeksforGeeks. Front-End Development; 2024. https://www.geeksforgeeks.org/front-end-development/. Accessed: 2024-08-24.

[14] Wikipedia contributors. Single-page application; 2024. https://en.wikipedia.org/wiki/Single-page_application. Accessed: 2024-08-24.

[15] GeeksforGeeks. Backend Development - What is Backend Development?; 2024. https://www.geeksforgeeks.org/backend-development/#what-is-backend-development-. Accessed: 2024-08-24.

[16] Webcase Studio. What is Integration in Software Development?; 2024. https://webcase.studio/what-integration-software-development/. Accessed: 2024-08-24.

[17] Goldner H, Erdelić T. Razvoj usluge interaktivne karte za prikaz podataka o kretanju vozila prometnom mrežom. In: Sveučilište u Zagrebu, Fakultet prometnih znanosti. Zagreb, Croatia; 2023. Downloaded from: https://repozitorij.fpz.unizg.hr/islandora/object/fpz:2995.

[18] Zeng Z, Zhu X, Qi S, Shen X, Cai S. Database Programming under Labwindows/CVI Platform. IOP Conference Series: Materials Science and Engineering. 2018;394:032024. Downloaded from: https://doi.org/10.1088/1757-899X/394/3/032024.

[19] Ndemo P. 2 and 3 Tier Architecture; 2024. https://medium.com/@paulndemo/2-and-3-tier-architecture-4a473e5ced3d. Accessed: 2024-08-24.

[20] NetSuite. Predictive Modeling; 2024. https://www.netsuite.com/portal/resource/articles/financial-management/predictive-modeling.shtml. Accessed: 2024-08-24.

[21] Aleksić D. Mogućnosti primjene metoda strojnog učenja u području telekomunikacija. University of Zagreb, Faculty of Transport and Traffic Sciences; 2021. Downloaded from: https://urn.nsk.hr/urn:nbn:hr:119:003361.

[22] Data Flair. Types of Machine Learning Algorithms; 2024. https://data-flair.training/blogs/types-of-machine-learning-algorithms/. Accessed: 2024-08-24.

[23] NimbleBox. Supervised, Unsupervised, and Reinforcement Learning; 2024. https://blog.nimblebox.ai/supervised-unsupervised-reinforcement-learning. Accessed: 2024-08-24.

[24] Taylor SJ, Letham B. Forecasting at Scale. PeerJ Preprints. 2017 September;5:e3190v2. Downloaded from: https://doi.org/10.7287/peerj.preprints.3190v2. CC BY 4.0 Open Access.

[25] Jedox. Error Metrics: How to Evaluate Forecasts; 2024. https://www.jedox.com/en/blog/error-metrics-how-to-evaluate-forecasts/. Accessed: 2024-08-24.

[26] Kothari V. Time Series Evaluation Metrics: MAPE vs WMAPE vs SMAPE - Which One to Use, Why, and When (Part 1); 2024. https://medium.com/@vinitkothari.24/time-series-evaluation-metrics-mape-vs-wmape-vs-smape-which-one-to-use-why-and-when-part1-32d3852b4779. Accessed: 2024-08-24.

LIST OF FIGURES

LIST OF TABLES

University of Zagreb
Faculty of Transport and Traffic Sciences
10000 Zagreb
Vukelićeva 4

DECLARATION OF ACADEMIC INTEGRITY AND CONSENT

I declare and confirm by my signature that this graduate thesis
is an exclusive result of my own work based on my research and relies on published literature,
as can be seen by my notes and references.
I declare that no part of the thesis is written in an illegal manner,
nor is copied from unreferenced work, and does not infringe upon anyone's copyright.
I also declare that no part of the thesis was used for any other work in
any other higher education, scientific or educational institution.
I hereby confirm and give my consent for the publication of my graduate thesis
titled **Real-time visualization of city public transport provider data and prediction of future trends**
on the website and the repository of the Faculty of Transport and Traffic Sciences and
the Digital Academic Repository (DAR) at the National and University Library in Zagreb.

Student:

In Zagreb, 16.09.2024

Kristijan Jurić
(signature)