

Applied Project - Data Engineering Course

Final Project Report

Authors: Ana Ferreira, Diogo Martins, João Dias e Vânia Vieira

1. Introduction

This report presents the development of a real-time data pipeline to support a dashboard displaying the positions of Carris buses in Lisbon. The project integrates streaming and batch data processing to provide real-time updates and historical insights. The goal is to create a robust data architecture capable of handling transit data effectively.

2. Architecture Overview

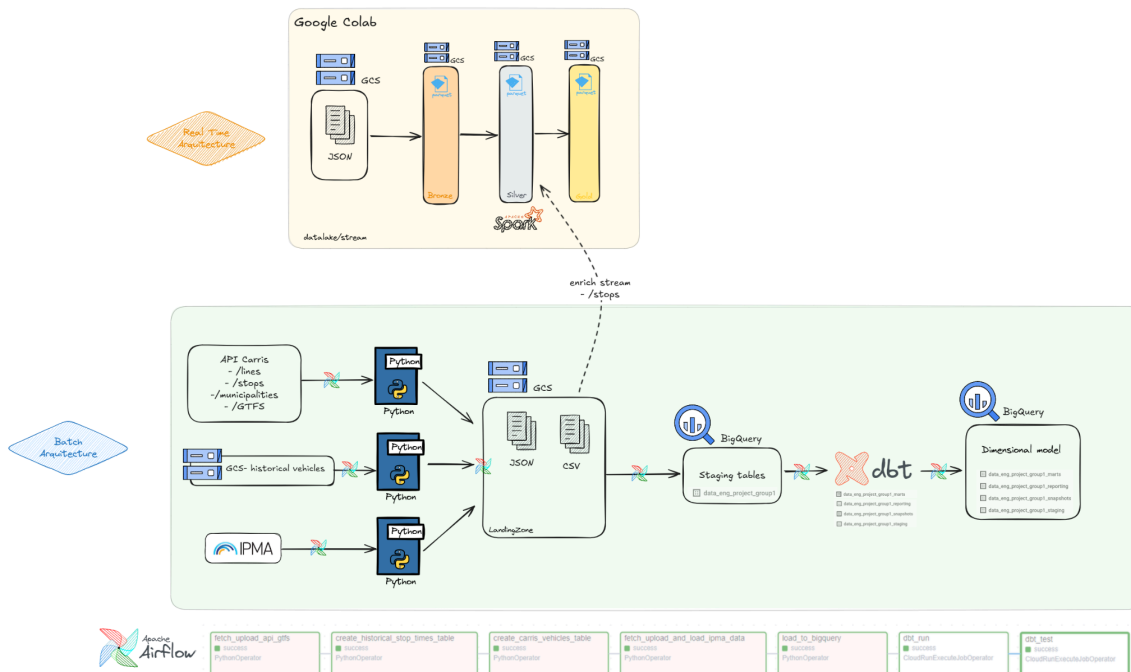
2.1 System Design

The system follows a hybrid architecture consisting of real-time and batch processing pipelines:

- **Streaming Pipeline:** This pipeline processes real-time vehicle position data from the Carris API where these files are previously stored in Google Cloud Storage (GCS). Using Google Colab to read these JSON data and store it in the different layers of GCS, the Spark Structured Streaming processes the data through a bronze-silver-gold architecture, where:
 - Bronze Layer: Stores raw JSON data.
 - Silver Layer: Cleans and enriches the data.
 - Gold Layer: Provides structured, ready-to-use data for analytics.
- **Batch Pipeline:** Runs periodic data ingestion and transformation jobs orchestrated by Apache Airflow. This pipeline collects:
 - Static data from the Carris API, such as stops, routes, and municipalities.
 - Historical vehicle movement data stored in GCS.
 - Weather data from IPMA.

These data sources are stored in a landing zone (in GCS) and then loaded into BigQuery as staging tables. dbt is used to transform the staging tables into a dimensional model for analytics and reporting.

The system architecture design looks like this:



2.2 Technology Stack

- **Cloud Platform: Google Cloud Platform (GCP)** – A cloud computing platform that provides a suite of infrastructure, storage, and analytics tools to support data processing, storage, and analysis.
- **Streaming Processing: Spark Structured Streaming** – A scalable and fault-tolerant stream processing engine built on Apache Spark. It enables real-time processing of data streams using a micro-batch approach while integrating seamlessly with cloud storage and databases.
- **Batch Orchestration: Apache Airflow** – An open-source workflow orchestration tool used to schedule, monitor, and manage batch data processing tasks. It ensures reliable execution of data pipelines and dependencies.
- **Data Storage: Google Cloud Storage (GCS) & BigQuery**
 - **GCS:** A scalable object storage service used for storing raw and intermediate data in the pipeline.
 - **BigQuery:** A serverless data warehouse optimized for fast SQL analytics on large datasets, used for structured storage and querying of processed data.
- **Transformation Layer: dbt (Data Build Tool)** – A transformation framework that enables analytics engineering by defining and managing SQL-based transformations in a modular, version-controlled manner. It allows for data modeling, testing, and documentation in a scalable way.

3. Data Sources

3.1 Primary Data Source

- **Carris API real-time Endpoints:** The Carris API offers multiple endpoints that serve different transit data needs. These include:
 - **/lines:** Returns information for lines. Each line can have several routes and patterns, and serves a set of municipalities and localities.
 - **/routes:** Provides route details. Each route can have at most two patterns, and serves a set of municipalities and localities.
 - **/stops:** Returns stop locations (static) and associated route information.
 - **/trips:** Contains data about planned and ongoing trips.
 - **/vehicles:** The core real-time endpoint for tracking active buses. Provides real-time vehicle positions.
 - **/gtfs:** Used for fetching historical transit data in GTFS format.

These endpoints collectively allow for a comprehensive understanding of the transit network, enabling enriched data processing and visualization.

3.2 Additional Data Sources

- **Weather Data from IPMA:** Integrated weather conditions from the Instituto Português do Mar e da Atmosfera (IPMA) to enhance transit analysis by correlating weather patterns with bus performance.

4. Data Pipelines

4.1 Streaming Pipeline

This chapter presents the implementation of a real-time data pipeline designed to process data from the Carris API, following an [ELT \(Extract, Load, Transform\) approach](#). Initially, we started using an ETL (Extract, Transform, Load) approach, but due to the use of Google Colab, we observed that since it is a free service, the transformation and writing processes were taking too long. Therefore, we decided to move forward with the ELT approach.

The pipeline is built using Google Cloud Platform (GCP) and consists of two main stages: streaming ingestion and batch transformation. This design ensures flexibility, scalability, and adaptability to evolving analytical needs.

The goal of this exercise is to display real-time metrics and attributes for a specific vehicle selected by the user. The required metrics include:

- **Velocity:** Average speed over the last two minutes.
- **Distance:** Distance traveled in the last two minutes.
- **Estimated Time to Next Stop:** Predicted time until the next stop.

A key challenge was to compute these metrics without directly using the speed information provided by the **/vehicles** endpoint. Instead, we derived them using a two-minute aggregation window applied to the vehicle's coordinates.

Additionally, the following attributes were displayed:

- Line
- Route
- Direction
- Next Stop

4.1.1 Architecture Overview

The data pipeline is structured into three layers, commonly referred to as the **bronze**, **silver**, and **gold** layers. Each layer serves a specific purpose in data processing and management:

- **Streaming Ingestion (Extract and Load - Bronze Layer)**
 - Data is ingested in real-time from the Carris API (This has been prepared prior by the professor).
 - The raw data is stored in an input bucket on GCP.
 - This raw data is then moved to a bronze layer bucket, preserving its original format (JSON) for future reference and processing.
 - Operations: Minimal transformation; only schema enforcement and deduplication.

bearing	block_id	current_status	id	lat	lon	line_id	route_id	schedule_relationship	shift_id	speed	stop_id	timestamp	trip_id						
137	20250121-64010205-112450234560	IN_TRANSIT	TO	44	12094	38.513027	4401	-8.8591585	4401_0_2	4401_0	SCHEDULED	112450234560	3.6111112	160479	2025-01-21 08:42:12	4401_0_2	3200	0830	AEMLZ
311	20250121-64010028-123290234560	IN_TRANSIT	TO	44	13597	38.701008	4705	-8.9522085	4705_0_2	4705_0	SCHEDULED	123290234560	4.7222223	100037	2025-01-21 08:42:12	4705_0_2	3200	0815	AEMLZ
134	20250121-64010022-123240234560	IN_TRANSIT	TO	44	12701	38.65296	4701	-9.00604	4701_0_2	4701_0	SCHEDULED	123240234560	0_0	090205	2025-01-21 08:42:12	4701_0_2	3200	0830	AEMLZ
10	20250121-64010232-121290234560	INCOMING	AT	44	12075	38.528786	4401	-8.870368	4401_0_1	4401_0	SCHEDULED	112190234560	9.166667	116022	2025-01-21 08:42:20	4401_0_1	3200	0800	AEMLZ
351	20250121-64010115-12110234560	IN_TRANSIT	TO	44	12515	38.5077	4612	-8.998533	4612_0_2	4612_0	SCHEDULED	12110234560	0_0	130066	2025-01-21 08:42:13	4612_0_2	3200	0800	AEMLZ
154	20250121-64010081-121640234560	IN_TRANSIT	TO	44	12097	38.706192	4504	-8.97577	4504_0_1	4504_0	SCHEDULED	121640234560	6.666665	100029	2025-01-21 08:42:12	4504_0_1	3200	0830	AEMLZ
22	20250121-64010341-112350234560	IN_TRANSIT	TO	44	12095	38.531322	4422	-8.885212	4422_0_2	4422_0	SCHEDULED	112350234560	7.222223	160201	2025-01-21 08:42:17	4422_0_2	3200	0820	AEMLZ
6	20250121-64010097-121490234560	INCOMING	AT	44	13608	38.658756	4600	-8.98978	4600_0_2	4600_0	SCHEDULED	121490234560	10.555555	090100	2025-01-21 08:42:12	4600_0_2	3200	0800	AEMLZ
280	20250121-64010118-121280234560	IN_TRANSIT	TO	44	12505	38.704437	4504	-8.973045	4504_0_2	4504_0	SCHEDULED	121280234560	7.222223	100027	2025-01-21 08:42:12	4504_0_2	3200	0800	AEMLZ
267	20250121-64010321-111030234560	IN_TRANSIT	TO	44	12516	38.482822	4451	-8.793863	4451_0_2	4451_0	SCHEDULED	111030234560	0_0	116080	2025-01-21 08:42:12	4451_0_2	3200	0805	AEMLZ
73	20250121-64010129-121180234560	IN_TRANSIT	TO	44	12068	38.65527	4602	-9.038903	4602_0_2	4602_0	SCHEDULED	121180234560	6.111111	090233	2025-01-21 08:42:19	4602_0_2	3200	0820	AEMLZ
33	20250121-64010266-111570234560	IN_TRANSIT	TO	44	12584	38.51783	4443	-8.837753	4443_0_1	4443_0	SCHEDULED	111570234560	6.666665	160714	2025-01-21 08:42:26	4443_0_1	3200	0840	AEMLZ
33	20250121-64010287-111360234560	INCOMING	AT	44	12550	38.64576	4312	-8.6903515	4312_0_2	4312_0	SCHEDULED	111360234560	13.055555	130283	2025-01-21 08:42:23	4312_0_2	3200	0815	AEMLZ
193	20250121-64010275-111480234560	IN_TRANSIT	TO	44	12552	38.57199	4562	-8.876545	4562_0_1	4562_0	SCHEDULED	111480234560	5.833335	130009	2025-01-21 08:42:11	4562_0_1	3200	0835	AEMLZ
99	20250121-64010593-111310234560	IN_TRANSIT	TO	44	12683	38.63144	4710	-8.915977	4710_0_2	4710_0	SCHEDULED	113110234560	0_0	130349	2025-01-21 08:42:12	4710_0_2	3200	0800	AEMLZ
317	20250121-64010136-121110234560	IN_TRANSIT	TO	44	12511	38.71933	4600	-8.998698	4600_0_2	4600_0	SCHEDULED	121110234560	5.2777777	010177	2025-01-21 08:42:15	4600_0_2	3200	0730	AEMLZ
169	20250121-64010143-121040234560	IN_TRANSIT	TO	44	12504	38.570717	4512	-8.888253	4512_0_1	4512_0	SCHEDULED	121040234560	0_0	130459	2025-01-21 08:42:21	4512_0_1	3200	0730	AEMLZ
295	20250121-64010134-121130234560	INCOMING	AT	44	12512	38.658627	4600	-9.041025	4600_0_1	4600_0	SCHEDULED	121130234560	4.7222223	090195	2025-01-21 08:42:21	4600_0_1	3200	0730	AEMLZ
58	20250121-64010276-111470234560	INCOMING	AT	44	12738	38.521328	4562	-9.00031	4562_0_2	4562_0	SCHEDULED	111470234560	13.055555	160778	2025-01-21 08:42:12	4562_0_2	3200	0835	AEMLZ
95	20250121-64010244-112080234560	IN_TRANSIT	TO	44	12098	38.53003	4441	-8.877542	4441_0_1	4441_0	SCHEDULED	112080234560	0_0	116026	2025-01-21 08:42:25	4441_0_1	3200	0830	AEMLZ

- **Batch Transformation (Transform - Silver Layer)**
 - Data from the bronze layer is processed in batches.
 - The transformation phase involves cleaning, standardizing, and structuring the data. In specific: Filter for getting only the columns required. Joins with reference data (Historical STOPS) for enrichment. Simple calculations and store it in new columns.
 - The transformed data is then stored in the silver layer bucket, making it suitable for analytical use.

stop_id	id	speed	timestamp	line_id	route_id	lat	lon	previous_lat	previous_lon	stop_lat	stop_lon	distance	distance_to_stop
120279	41	1100	0.2777778	2025-01-21 08:26:45	1606	1606_0	38.72538	-9.246014	38.72538	-9.246014	38.72538	0.0	0.0020833486
120281	41	1100	6.111111	2025-01-21 08:27:23	1606	1606_0	38.725426	-9.246371	38.72538	-9.246014	38.725094	0.03143868	0.23987857
120281	41	1100	6.111111	2025-01-21 08:27:23	1606	1606_0	38.725426	-9.246371	38.725426	-9.246371	38.725094	0.0	0.23987857
120431	41	1100	3.3333333	2025-01-21 08:28:07	1606	1606_0	38.725025	-9.249382	38.725426	-9.246371	38.723774	0.2649507	0.2056784
120431	41	1100	7.5	2025-01-21 08:28:51	1606	1606_0	38.72386	-9.248462	38.725025	-9.249382	38.723774	0.247636	0.15202379
120283	41	1100	9.444445	2025-01-21 08:29:16	1606	1606_0	38.723534	-9.247613	38.72386	-9.248462	38.72322	0.248406	0.08217244
120285	41	1100	8.888889	2025-01-21 08:29:28	1606	1606_0	38.72305	-9.248899	38.723534	-9.247613	38.720734	0.250854	0.12392749
120285	41	1100	7.222223	2025-01-21 08:30:05	1606	1606_0	38.72292	-9.250658	38.72305	-9.248899	38.720734	0.250854	0.1532401
120287	41	1100	10.0	2025-01-21 08:30:37	1606	1606_0	38.720783	-9.250787	38.72292	-9.250658	38.71751	0.252359	0.23780043
120289	41	1100	13.611111	2025-01-21 08:31:14	1606	1606_0	38.717346	-9.252539	38.720783	-9.250787	38.716	0.41129494	0.19833782
120289	41	1100	0.0	2025-01-21 08:31:45	1606	1606_0	38.71603	-9.254039	38.717346	-9.252539	38.716	0.19584359	0.0033944084
120291	41	1100	3.0555556	2025-01-21 08:32:21	1606	1606_0	38.714947	-9.255993	38.71603	-9.254039	38.71479	0.256263	0.20797867
120293	41	1100	8.055555	2025-01-21 08:32:42	1606	1606_0	38.714767	-9.256356	38.714947	-9.255993	38.715626	0.262791	0.03729978
120293	41	1100	6.388889	2025-01-21 08:32:58	1606	1606_0	38.714474	-9.257175	38.714767	-9.256356	38.715626	0.262791	0.07822138
120293	41	1100	1.6666666	2025-01-21 08:33:48	1606	1606_0	38.71565	-9.262754	38.714474	-9.257175	38.715626	0.262791	0.50136375
121219	41	1100	4.4444447	2025-01-21 08:34:10	1606	1606_0	38.715687	-9.263042	38.71565	-9.262754	38.715153	0.263576	0.025345437
121219	41	1100	4.4444447	2025-01-21 08:34:10	1606	1606_0	38.715687	-9.263042	38.715687	-9.263042	38.715153	0.263576	0.0
121219	41	1100	0.0	2025-01-21 08:35:22	1606	1606_0	38.71522	-9.263537	38.715687	-9.263042	38.715153	0.263576	0.06724656
121075	41	1100	10.277778	2025-01-21 08:35:51	1606	1606_0	38.71494	-9.263243	38.71522	-9.263537	38.709793	0.266945	0.04048403
121075	41	1100	10.277778	2025-01-21 08:35:51	1606	1606_0	38.71494	-9.263243	38.71494	-9.263243	38.709793	0.266945	0.0

- **Gold Layer (Optimized for Analytics)**

- Further aggregation and enrichment of data can be performed to optimize performance for specific business queries.
- Data in this layer is structured for easy consumption by the desired analytics dashboards.

	id	stop_id	window	distance_2_min	distance_to_stop	speed	time_to_stop
41	1102	120033	{2025-01-21 09:04...	0.28879860043525696	0.010783878	8.663958013057709	00:00:04
41	1102	120283	{2025-01-21 09:14...	0.2967355474829674	0.0025463942	8.902066424489021	00:00:01
41	1102	120261	{2025-01-22 07:12...	0.10032915696501732	0.03582338	3.0098747089505196	00:00:42
41	1102	120181	{2025-01-22 17:20...	0.3366996943950653	0.006795796	10.10099083185196	00:00:02
41	1102	120261	{2025-01-23 07:12...	0.06684991717338562	0.002651806	2.0054975152015686	00:00:04
41	1102	120061	{2025-01-23 11:00...	0.1589718833565712	0.0050084856	4.769156500697136	00:00:03
41	1103	120015	{2025-01-21 15:28...	0.2639096677303314	0.0013698483	7.917290031909943	00:00:00
41	1103	121067	{2025-01-21 18:58...	0.11611327528953552	0.06313308	3.4833982586860657	00:01:05
41	1103	030561	{2025-01-21 20:34...	0.2825373113155365	0.27729845	8.476119339466095	00:01:57
41	1103	120013	{2025-01-22 08:40...	0.187242291867733	0.0024670342	5.61726875603199	00:00:01
41	1103	120101	{2025-01-23 09:42...	0.3803628012537956	0.05295019	11.410884037613869	00:00:16
41	1105	120941	{2025-01-21 15:08...	0.11788272857666016	0.07494362	3.5364818572998047	00:01:16
41	1105	120982	{2025-01-21 15:58...	0.0685768574476242	0.0526165	2.057305723428726	00:01:32
41	1105	120994	{2025-01-21 17:46...	0.151200070977211	0.07514134	4.53600212931633	00:00:59
41	1105	121270	{2025-01-22 07:22...	0.24714062199927866	0.00850163	7.41421865997836	00:00:04
41	1105	170730	{2025-01-22 11:38...	0.8066409379243851	0.41696852	24.199228137731552	00:01:02
41	1105	121121	{2025-01-22 14:58...	0.09158371388912201	0.4682337	2.7475114166736603	00:10:13
41	1105	170775	{2025-01-22 15:30...	0.0924074724316597	0.508149	2.772224172949791	00:10:59
41	1105	120999	{2025-01-22 17:02...	0.18588903546333313	0.14382981	5.576671063899994	00:01:32
41	1105	121108	{2025-01-23 07:10...	0.39273974299430847	0.08853564	11.782192289829254	00:00:27

4.1.2 Implementation

To compute the required metrics, we followed these steps:

- **Data Ingestion:**

- The raw vehicle location data was extracted in real-time from the Carris GCS bucket and stored in the bronze layer.
- Example code snippet for streaming ingestion:

```
from pyspark.sql.types import *

# Create schema
vehicle_schema = StructType([
    StructField('bearing', IntegerType(), True),
    StructField('block_id', StringType(), True),
    StructField('current_status', StringType(), True),
    StructField('id', StringType(), True),
    StructField('lat', FloatType(), True),
    StructField('line_id', StringType(), True),
    StructField('lon', FloatType(), True),
    StructField('pattern_id', StringType(), True),
    StructField('route_id', StringType(), True),
    StructField('timestamp', TimestampType(), True)
])

# Read streaming data
stream =
spark.readStream.format("json").schema(vehicle_schema).load("gs://edit-de-project-streaming-data/carris-vehi
```

```

cles")

# Write streaming data to Bronze Layer
bronze_layer = "gs://edit-data-eng-project-group1/datalake/stream/ELT/bronze_layer"
query = (stream
    .writeStream
    .outputMode("append")
    .option("path", bronze_layer)
    .option('checkpointLocation',
'gs://edit-data-eng-project-group1/datalake/stream/ELT/bronze_layer/checkpoint')
    .start()
)
query.awaitTermination(60)

```

Spark Streaming is a framework in Apache Spark that enables real-time processing of streaming data. It allows you to process continuous data streams by ingesting data in mini-batches and applying transformations to extract insights. In this project, Spark Streaming is used to read vehicle location data in real time from the Carris GCS bucket and store it in the bronze layer.

The readStream is used to read data from the source (Google Cloud Storage in this case) in real-time. The writeStream method ensures the data is written continuously to storage while using checkpointLocation to maintain state. This tells Spark to save checkpoint data in the given Google Cloud Storage location. If the stream fails or restarts, Spark can use this checkpoint to resume processing from where it left off, preventing data deduplication in case of failure.

- **Processing and Aggregation:**

- A two-minute sliding window was applied to the coordinate data of each vehicle.
- Using geospatial calculations, we computed the distance traveled within this window.
- Example of distance calculation using the Haversine formula:

```

from pyspark.sql.functions import lag, col, coalesce, udf
from pyspark.sql.window import Window
from pyspark.sql.types import FloatType
import math

def haversine_distance(lat1, lon1, lat2, lon2):
    if any(x is None for x in [lat1, lon1, lat2, lon2]):
        return 0.0
    R = 6371 # Earth's radius in kilometers
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat/2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon/2)**2
    c = 2 * math.asin(math.sqrt(a))
    return R * c

```

```
distance_udf = udf(haversine_distance, FloatType())
```

```
# Define a window specification
windowSpec = Window.partitionBy("id").orderBy("timestamp")

#select columns
transform = parquet_df.select('id', 'speed', 'timestamp', 'line_id', 'route_id', 'stop_id', 'lat', 'lon')

# Create a new column 'previous_value' using lag
transform = transform.withColumn("previous_lat", coalesce(lag("lat", 1).over(windowSpec), col("lat")))
transform = transform.withColumn("previous_lon", coalesce(lag("lon", 1).over(windowSpec), col("lon")))

# Get the dataset from endpoint STOPS that we need to join to our main dataset
df_stops = spark.read.option("header",
"true").csv('gs://edit-data-eng-project-group1/LandingZone/GTFS/stops.txt')
df_stops = df_stops.select('stop_id', 'stop_lat', 'stop_lon')
df_stops = df_stops.withColumn("stop_lat", df_stops["stop_lat"].cast("float"))
df_stops = df_stops.withColumn("stop_lon", df_stops["stop_lon"].cast("float"))

# Join and add new calculated columns
transform = transform.join(df_stops, on='stop_id', how='left')

transform = transform.withColumn("distance",
distance_udf(transform["previous_lat"], transform["previous_lon"], transform["lat"], transform["lon"]))
transform = transform.withColumn("distance_to_stop",
distance_udf(transform["lat"], transform["lon"], transform["stop_lat"], transform["stop_lon"]))

transform.show()
```

The team opted by processing the vehicle location data using PySpark to compute distance metrics. To do that we defined a function - Haversine formula - for calculating the distance based on two geographic points (latitude and longitude).

Using a window function of 2 minutes and the lag function, we retrieved the previous latitude and longitude of each vehicle based on timestamps and ensured that the missing values were handled with coalesce().

To get the Stop information required by the exercise, we joined a dataset with Bus Stop Locations (stored on our GCS bucket in txt format). We then merged vehicle data with stop data to enable distance calculations.

With these efforts, we got new computed columns - the distance traveled by the vehicle between its last two known locations ([‘distance’]) and the distance from the vehicle's current position to the next stop ([‘distance_to_stop’]).

- **Transformation and Computation of Speed and Distance:**
 - Compute speed and estimated time to stop:

```
import pyspark.sql.functions as F
```



```

agg = transform.groupBy("id", "stop_id", F.window("timestamp", "2 minutes")).agg(
    F.sum("distance").alias("distance_2_min"),
    F.last("distance_to_stop").alias("distance_to_stop")
)

agg = agg.withColumn('speed', col('distance_2_min')/(2/60))

agg = agg.filter(agg.distance_to_stop.isNotNull() & (agg.distance_to_stop > 0) & (agg.speed.isNotNull()) &
    (agg.speed > 0)) \
    .withColumn("time_to_stop", (col('distance_to_stop')/col('speed') * 3600))

agg = agg.withColumn(
    'time_to_stop',
    F.from_unixtime(
        F.unix_timestamp(F.lit('00:00:00'), 'HH:mm:ss') + col('time_to_stop'),
        'HH:mm:ss'
    )
)
agg.show()

```

In this stage, the team grouped the data by id and stop_id and created time windows of 2 minutes. Then, the data has been aggregated to calculate the total of distance traveled during the 2 minutes as well as the last value of distance_to_stop for each group, which will give us the distance to the stop at the end of the 2-minute window. This has been calculated in order to obtain the speed (distance / time = distance_2_min / 2 min).

For the time_to_stop calculation, we have used the same logic but now distance / speed and multiplying by 3600 to convert from hours to seconds. We then converted the time_to_stop from seconds to a time format (HH:mm:ss) for a more readable format.

With that we have concluded the objective of the exercise calculating Velocity (speed), Distance (distance_2_min) and Estimated Time to Next Stop (time_to_stop).

	id	stop_id	window	distance_2_min	distance_to_stop	speed	time_to_stop
41	1102	120033	{2025-01-21 09:04...	0.28879860043525696	0.010783878	8.663958013057709	00:00:04
41	1102	120283	{2025-01-21 09:14...	0.2967355474829674	0.0025463942	8.902066424489021	00:00:01
41	1102	120261	{2025-01-22 07:12...	0.10032915696501732	0.03582338	3.0098747089505196	00:00:42
41	1102	120181	{2025-01-22 17:20...	0.3366996943950653	0.006795796	10.10099083185196	00:00:02
41	1102	120261	{2025-01-23 07:12...	0.06684991717338562	0.002651806	2.0054975152015686	00:00:04
41	1102	120061	{2025-01-23 11:00...	0.1589718833565712	0.0050084856	4.769156500697136	00:00:03
41	1103	120015	{2025-01-21 15:28...	0.2639096677303314	0.0013698483	7.917290031909943	00:00:00
41	1103	121067	{2025-01-21 18:58...	0.11611327528953552	0.06313308	3.4833982586860657	00:01:05
41	1103	030561	{2025-01-21 20:34...	0.2825373113155365	0.27729845	8.476119339466095	00:01:57
41	1103	120013	{2025-01-22 08:40...	0.187242291867733	0.0024670342	5.61726875603199	00:00:01
41	1103	120101	{2025-01-23 09:42...	0.3803628012537956	0.05295019	11.410884037613869	00:00:16
41	1105	120941	{2025-01-21 15:08...	0.11788272857666016	0.07494362	3.5364818572998047	00:01:16
41	1105	120982	{2025-01-21 15:58...	0.0685768574476242	0.0526165	2.057305723428726	00:01:32
41	1105	120994	{2025-01-21 17:46...	0.151200070977211	0.07514134	4.53600212931633	00:00:59
41	1105	121270	{2025-01-22 07:22...	0.24714062199927866	0.00850163	7.41421865997836	00:00:04
41	1105	170730	{2025-01-22 11:38...	0.8066409379243851	0.41696852	24.199228137731552	00:01:02
41	1105	121121	{2025-01-22 14:58...	0.09158371388912201	0.4682337	2.7475114166736603	00:10:13
41	1105	170775	{2025-01-22 15:30...	0.0924074724316597	0.508149	2.772224172949791	00:10:59
41	1105	120999	{2025-01-22 17:02...	0.18588903546333313	0.14382981	5.576671063899994	00:01:32
41	1105	121108	{2025-01-23 07:10...	0.39273974299430847	0.08853564	11.782192289829254	00:00:27

To ensure the accuracy and reliability of the resulting dataset, a validation process was conducted to assess whether the obtained values align with real-world expectations. Through exploratory data analysis, we confirmed that the computed results are consistent with expected behavior.

For instance, in the case of bus 41|1102 approaching stop 120261, we observe a logical progression in the data:

- The distance to the stop decreases from 0.03 km to 0.002 km, indicating that the bus is moving closer to its destination.
- The estimated time to stop correspondingly reduces from 42 seconds to 4 seconds, reflecting a realistic approximation of the time required to reach the stop.

These findings confirm that the dataset effectively captures the expected trends in vehicle movement, reinforcing confidence in the computed results.

4.1.3 Integration with a Near Real-Time Dashboard

With this approach, the dataset could be ingested into a dashboard that operates in near real-time, allowing users to monitor vehicle positions and estimated arrival times with minimal delay. However, for a fully real-time implementation, a more efficient processing approach would be required.

Initially, we attempted to implement the ETL process with on-the-fly transformations, but performance issues were observed. The delays experienced may be attributed to the usage of Google Colab, which has inherent limitations in terms of computing resources and network performance.

To achieve real-time data processing, the following alternative approaches could be considered:

- **Streaming Data Processing Frameworks:** Using technologies like Apache Kafka for event streaming combined with Spark Structured Streaming for real-time computations can significantly improve performance.
- **Optimized Cloud Infrastructure:** Deploying the pipeline on cloud-native services such as Google Cloud Dataflow would provide scalable and optimized real-time processing.

By implementing these more effective streaming architectures, we could achieve true real-time analytics, ensuring immediate updates in the dashboard with minimal latency.

4.2 Batch Pipeline

4.1.1 Architecture Overview

Our batch architecture follows a structured ETL pipeline that collects data from multiple sources (Carris API and IPMA), processes it, and stores it in a data warehouse for further analysis. The data ingestion phase includes fetching information from APIs (e.g., Carris, IPMA) and GCS-based storage. The extracted data is then staged in a landing zone before

being loaded into BigQuery. Further transformations are applied using **dbt** to create a dimensional model, enabling optimized querying for analytics and reporting.

4.1.2 Implementation

The batch processing is orchestrated using **Apache Airflow**, ensuring automated workflows and dependency management. The detailed implementation of this process is further elaborated in the **Orchestration** and **Data Model** chapters. The orchestration chapter focuses on task scheduling and monitoring, while the data model section describes how the data is structured and transformed to support analytical queries.

4.1.3 Integration with a Dashboard

The final processed data stored in **BigQuery** is designed to be easily integrated with visualization tools, enabling historical reporting insights. The structured data can be directly connected to BI tools. This setup allows stakeholders to interact with dashboards that display key metrics, including vehicle movement patterns, weather impact, and estimated arrival times, providing valuable insights for decision-making.

5. Orchestration

The tool chosen for the orchestration process was Airflow, which defined a single DAG ([group1_carris_reporting_batch](#)) for the batch process.



The DAG is described in the file [group1_carris_reporting_batch.py](#), which is stored in the GCP Bucket [edit-de-project-airflow-dags](#).

Name	Size	Type	Created	Storage class	Last modified	Public access	Version history
dag_extract_and_load_data_g3.py	87.2 KB	text/x-python-script	21 Jan 2025, 22:41:13	Standard	21 Jan 2025, 22:41:13	Not public	—
group1_carris_reporting_batch.py	12 KB	application/octet-stream	23 Jan 2025, 23:42:11	Standard	23 Jan 2025, 23:42:11	Not public	—
group2_dagAirflowBatch.py	16 KB	application/octet-stream	23 Jan 2025, 19:05:27	Standard	23 Jan 2025, 19:05:27	Not public	—
group3_dag_dbt_job.py	1 KB	text/x-python-script	18 Jan 2025, 11:30:03	Standard	18 Jan 2025, 11:30:03	Not public	—
historical_stop_times.py	1.3 KB	text/x-python	17 Jan 2025, 11:34:13	Standard	17 Jan 2025, 11:34:13	Not public	—
test_dbt_cloud_run.py	1.4 KB	text/x-python	17 Jan 2025, 11:05:24	Standard	17 Jan 2025, 11:05:24	Not public	—
test_historical.py	24.6 KB	application/octet-stream	15 Jan 2025, 22:52:30	Standard	15 Jan 2025, 22:52:30	Not public	—

It's scheduled to trigger the batch execution every day at 1am, using a cron format schedule, and with the following configurations:

```
# --- DAG DEFINITION ---
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'retries': 1,
}

dag = DAG(
    'group1_carris_reporting_batch',
    default_args=default_args,
    description='DAG for fetching data, uploading to GCS, and ingesting into BigQuery',
    schedule_interval='0 1 * * *',
    start_date=datetime(2025, 1, 18),
    catchup=False,
)
```

The main tasks defined for the orchestration are defined below:

fetch_api_GTFS_task: It's responsible for calling the function *fetch_upload_api_gtfs* that triggers the python functions responsible for the CARRIS API requests and storing the responses as JSON files in the GCP Bucket [“edit-data-eng-project-group1/LandingZone”](#). It also replaces all existing files in the bucket.

create_historical_stop_times_task: Triggers the python function *create_historical_stop_times_table* that will copy the *historical_stop_times* table from the *de_project_teachers* and create it with all the data in the *data_eng_project_group1* dataset, with the prefix *staging*.

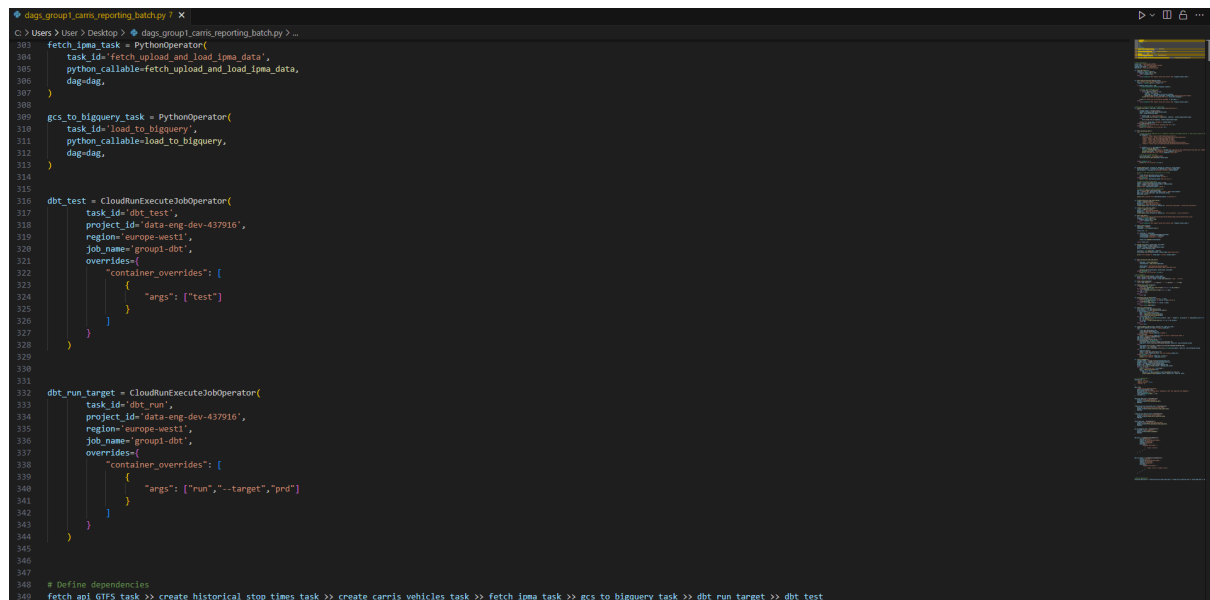
create_carris_vehicles_task: Triggers the python function *create_historical_stop_times_table* that will copy the *carris_vehicles* table from the *de_project_teachers* and create it with all the data in the *data_eng_project_group1* dataset, with the prefix *staging*.

fetch_ipma_task: It's responsible for calling the function *fetch_ipma_data* that triggers the python functions responsible for the IPMA API request and storing the response as a JSON file in the GCP Bucket [“edit-data-eng-project-group1/LandingZone”](#). It also triggers a python function responsible for filtering the response only for the stationid = "1200535" (Lisbon station).

gcs_to_bigquery_task: This function simply reads all the JSON files in the LandingZone bucket and creates or replaces the bigquery tables in the “*data_eng_project_group1*” dataset, with the prefix “*staging*”.

dbt_run_target: Triggers the job defined in the GCP Cloud Run “[group1-dbt](#)”, using a CloudRunExecuteJobOperator that will execute the **dbt-run** command, with the **prd** profile as an argument, for the containerized image of the dbt model available in the [docker hub repository](#).

dbt_test: Triggers the job defined in the GCP Cloud Run “[group1-dbt](#)”, using a CloudRunExecuteJobOperator that will execute the **dbt-test** command for the containerized image of the dbt model available in the [docker hub repository](#).



```
103 fetch_ipma_task = PythonOperator(
104     task_id='fetch_upload_and_load_ipma_data',
105     python_callable=fetch_upload_and_load_ipma_data,
106     dag=dag,
107 )
108
109 gcs_to_bigquery_task = PythonOperator(
110     task_id='load_to_bigquery',
111     python_callable=load_to_bigquery,
112     dag=dag,
113 )
114
115 dbt_test = CloudRunExecuteJobOperator(
116     task_id='dbt_test',
117     project_id='data-eng-dev-437916',
118     region='europe-west1',
119     job_name='group1-dbt',
120     overrides={
121         "container_overrides": [
122             {
123                 "args": ["test"]
124             }
125         ]
126     },
127 )
128
129 dbt_run_target = CloudRunExecuteJobOperator(
130     task_id='dbt_run',
131     project_id='data-eng-dev-437916',
132     region='europe-west1',
133     job_name='group1-dbt',
134     overrides={
135         "container_overrides": [
136             {
137                 "args": ["run", "--target", "prd"]
138             }
139         ]
140     },
141 )
142
143 # Define dependencies
144 fetch_api_GTFS_task >> create_historical_stop_times_task >> create_carris_vehicles_task >> fetch_ipma_task >> gcs_to_bigquery_task >> dbt_run_target >> dbt_test
```

All tasks use PythonOperators, with the exception of the dbt tasks that use CloudRunExecuteJobOperators. And have the following dependencies defined:

```
fetch_api_GTFS_task >> create_historical_stop_times_task >> create_carris_vehicles_task >>
fetch_ipma_task >> gcs_to_bigquery_task >> dbt_run_target >> dbt_test
```

6. Data Model

6.1 Data Warehousing with Big Query

Exploring Staging Tables

BigQuery was utilized to explore and transform raw data / staging tables into structured reporting tables before defining the final star schema. The following steps were taken:

1. **Data Ingestion**
 - Raw data was loaded into BigQuery staging tables from various sources.
2. **Data Profiling**
 - Identified missing values, duplicates, and inconsistencies.
3. **Joins and Relationship Analysis**
 - Verified relationships between staging tables to align with the final schema.

Example join query:

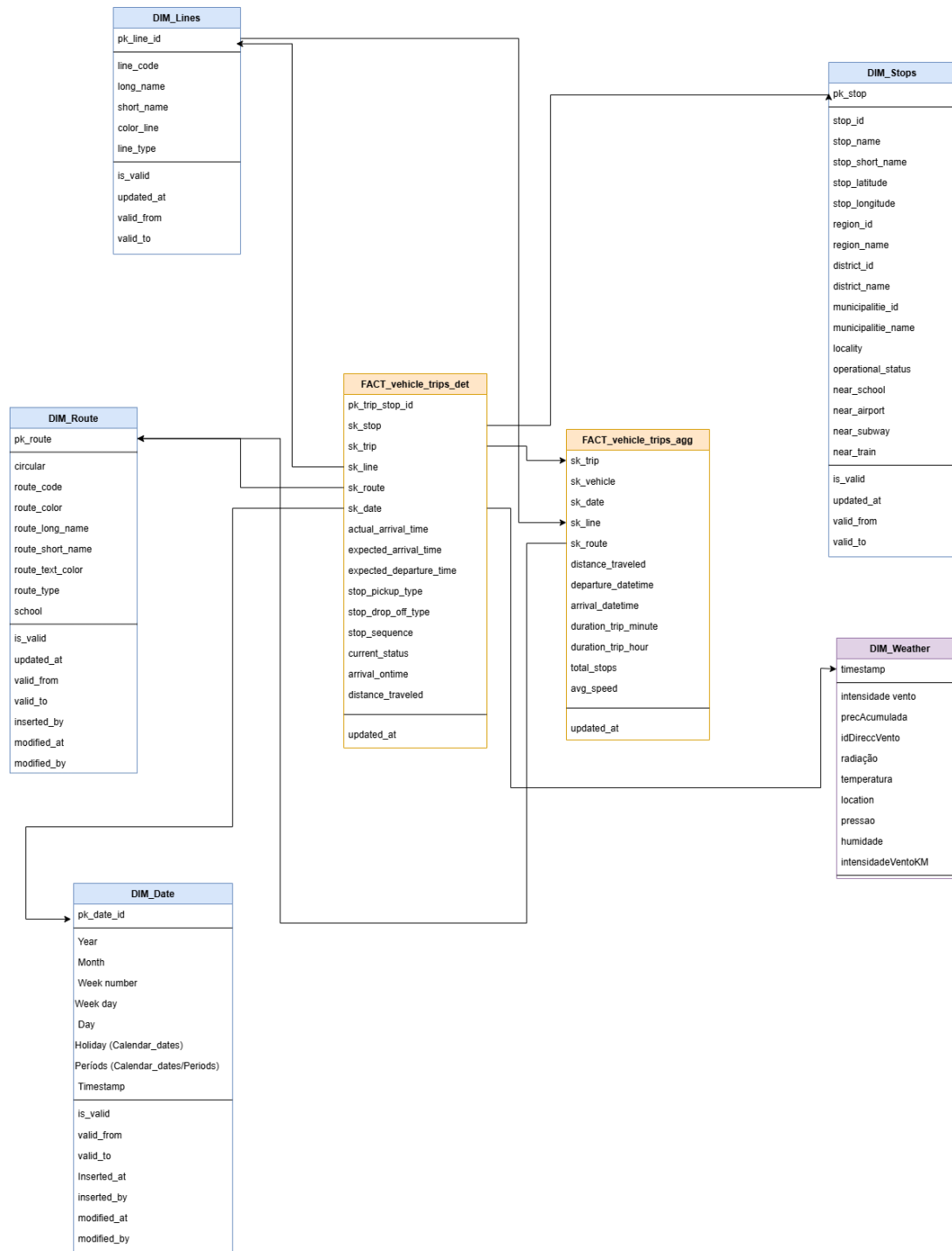
```
1  -- Detail fact trip stop
2
3  DECLARE now TIMESTAMP DEFAULT CURRENT_TIMESTAMP();
4  drop table if exists `data-eng-dev-437916.data_eng_project_group1.FACT_vehicles_trip_det`;
5  create table `data-eng-dev-437916.data_eng_project_group1.FACT_vehicles_trip_det` as
6  with trip_status as (
7    SELECT distinct
8      concat(b.stop_id, '_', b.trip_id) as pk_trip_stop_id,
9      b.stop_id as sk_stop,
10     b.trip_id as sk_trip,
11     b.vehicle_id as sk_vehicle,
12     b.line_id as sk_line,
13     t.route_id as sk_route,
14     DATE(b.timestamp) as sk_date,
15     b.timestamp as sk_hist_datetime,
16     a.arrival_time as expected_arrival_time,
17     a.departure_time as expected_departure_time,
18     a.pickup_type as stop_pickup_type,
19     a.drop_off_type as stop_drop_off_type,
20     a.stop_sequence as stop_sequence,
21     b.current_status as current_status,
22     a.shape_dist_traveled as distance_traveled,
23     now as inserted_at,
24     'pessoa' as inserted_by,
25     now as modified_at,
26     'pessoa' as modified_by,
27   FROM (select distinct * from `data-eng-dev-437916.data_eng_project_group1.staging_historical_stop_times`) b
28   left join `data-eng-dev-437916.data_eng_project_group1.staging_stop_times` a
29     on a.trip_id = b.trip_id and cast(a.stop_id as string) = cast(concat('0', b.stop_id) as string)
30   left join `data-eng-dev-437916.data_eng_project_group1.staging_trips` t
31     on b.trip_id = t.trip_id
32 )
33 select *
34 from trip_status
```

Defining the Final Star Schema

After data exploration, the final schema was structured into dimension and fact tables, ensuring optimized query performance and analytical efficiency.

6.2 Dimensional Model

The final dimensional model used for analyzing vehicle trips in relation to routes, stops, weather conditions, and other relevant factors. The model follows a star schema design, with fact tables capturing occurred trips with planned schedule and dimension tables providing contextual information.



- **Fact Tables :**

FACT_vehicle_trips_det - contains detailed records of vehicle trips granularity by stop to identify discrepancies between planned and actual stop times

FACT_vehicle_trips_agg - contains overall aggregated information of each trip for performance analysis.

- **Dimension Tables:**

DIM_Lines - Stores information about different transit lines.

DIM_Stops - Stores information about various routes taken by vehicles.

DIM_Route - Stores information about different stops along the routes.

DIM_Date - Stores date-related information for analysis over time.

DIM_Weather - Stores weather conditions at different points in time for correlation with trip performance.

7. Data Transformation in dbt

In this section, we provide a comprehensive breakdown of the data transformation processes executed using dbt (Data Build Tool) in our project. dbt was leveraged to convert raw data into well-structured, analytics-ready datasets while ensuring data quality and maintainability. This section covers:

- The project architecture and model organization.
- The transformation logic and key business rules applied.
- The use of dbt features, including macros, tests, and documentation.

7.1 Project Configuration

7.1.1 Core Configuration Files

The **dbt_project.yml** file defines the core structure, settings, and behavior of the dbt project, ensuring consistency in data transformations, schema management, and model materialization.

The **profiles.yml** file configures the connection settings for dbt, specifying how it interacts with the Google BigQuery data warehouse. The project includes three environments:

- **staging**
- **development** (dev)
- **production** (prd)

This setup enables structured data transformation workflows across different stages. Each environment connects to a specific BigQuery dataset, with OAuth-based authentication. The ability to switch environments using the `--target` argument provides flexibility in deployment and smooth transitions from development to production.

7.2 Project Structure and Model Organization

The models in the src/models directory are structured into three primary layers:

1. Staging Layer (staging/)

Responsible for extracting, standardizing, and structuring raw data before further processing. Staging models are materialized as **views** to ensure flexibility and minimize storage overhead. Common transformations include:

- Renaming columns
- Converting data types
- Applying basic cleaning rules

Key Staging Models:

- **stg_Lines.sql** – Ingests data related to transport lines from the staging_carris_lines_data source, applying UNNEST on the localities field.
- **stg_Routes.sql** – Ingests data from the staging_routes source, processing route-related information, renaming the route_type field, and transforming the circular field.
- **stg_IPMA_weather.sql** – Ingests weather data from the staging_ipma_lisbon_data source.
- **stg_Stops.sql** – Ingests stop-related data from the staging_stops source, renaming fields and converting boolean values to text.
- **stg_historical_stop_times.sql** – Ingests data such as trip ID, stop, line, route, and exact stop time from the staging_historical_stop_times source. Generates surrogate keys using dbt_utils.generate_surrogate_key.
- **stg_stop_times.sql** – Ingests data from the staging_stop_times source, including expected arrival and departure times, distance traveled, and stop sequence. Converts boolean values into descriptive text.
- **stg_trips.sql** – Ingests trip-related data from the staging_trips source, processing key relationships such as service_id and route_id.

2. Snapshot Layer (snapshots/)

This layer tracks changes in dimension tables over time using dbt snapshots with the **"check" strategy**, which detects updates based on specific columns. This ensures efficient processing of modified records while preserving historical tracking. Snapshots are stored outside the src/models directory.

Key Snapshot Models:

- **DIM_Lines_snapshot.sql** – Constructs the lines dimension table by joining stg_lines with stg_routes using a **left join**. Creates pk_line using dbt_utils.generate_surrogate_key and includes all relevant fields related to transport lines. Control columns are also added.

- **DIM_Routes_snapshot.sql** – Builds the routes dimension table using stg_routes. Generates pk_route with dbt_utils.generate_surrogate_key, incorporates key route-related fields, and includes control columns.
- **DIM_Stops_snapshot.sql** – Creates the stops dimension table based on stg_stops. Generates pk_stop using dbt_utils.generate_surrogate_key, includes relevant stop-related fields, and adds control columns.

Each snapshot model includes:

- A **target schema** for storing historical records.
- A **unique key** to identify records.
- A **change detection strategy** (check_cols) to track modifications.

3. Marts Layer (marts/)

This layer structures and optimizes data for analytics, materializing models as **tables** to enhance query performance.

Key Marts Models:

- **DIM_Dates.sql** – Generates a date dimension using dbt_date.get_date_dimension, covering a range from 2015 to 2050.
- **DIM_Lines.sql, DIM_Routes.sql, DIM_Stops.sql** – Created from snapshot tables with additional metadata columns (dbt_updated_at, dbt_valid_from, dbt_valid_to).
- **DIM_Weather.sql** – Builds a weather dimension from stg_IPMA_weather.
- **FACT_vehicle_trip_det.sql** – Constructs the primary fact table, integrating historical and planned trip data.

4. Reporting Layer (reporting/)

Designed for final analytical consumption, ensuring data is ready for dashboard reporting. Uses **table materialization** to optimize performance.

Key Reporting Models:

- **DIM_Line.sql, DIM_Date.sql, DIM_Routes.sql, DIM_Weather.sql** – Direct reference of marts models for reporting purposes.
- **FACT_trip_agg.sql** – Aggregates vehicle trip data, calculating:
 - Total distance traveled
 - Trip duration
 - Number of stops
 - Average speed

This model extracts insights from FACT_vehicle_trip_det, identifying the **first and last stops** of each trip and computing key performance metrics.

7.3 Schema Documentation and Data Quality Testing

7.3.1 Schema Documentation

- dbt generates documentation for all models, allowing stakeholders to explore data structures and understand transformations.
- Descriptions and metadata were added to models, improving transparency and collaboration.

7.3.2 Data Quality Testing

Custom tests were implemented to ensure data integrity, including:

- **Uniqueness constraints** (e.g., primary keys).
- **Non-null constraints** for essential fields.
- **Referential integrity checks** between related tables.

The use of dbt in this project streamlined data transformation, ensuring high-quality, structured, and well-documented datasets. The combination of well-defined environments, robust transformation logic, and automated testing improved efficiency and maintainability, making the data pipeline reliable and scalable for analytics.

9. Future Improvements

- Expand to include more real-time analytics features.
- Add more enrichment sources, that will result in more dimensions to provide better insights for the reporting users.
- Enhance dashboard visualization capabilities.
- Optimize batch processing efficiency.
- Implement a data cleansing step to handle inconsistent and null values, improving data quality and reliability.

10. Conclusion

This project successfully implemented a scalable data pipeline for real-time and historical transit analysis. The use of technologies such as Apache Spark, Airflow, and BigQuery enabled the development of an efficient system capable of ingesting, processing, and structuring data for analytical purposes.

The streaming pipeline ensured real-time updates on vehicle positions, while the batch pipeline allowed for the consolidation and transformation of historical data, providing deeper analytical insights. The applied dimensional modeling optimized data structuring, making information more accessible and relevant to users' needs.

Additionally, the integration of multiple data sources, including weather conditions, expanded the scope of analysis, enabling more comprehensive insights into the factors affecting public transport performance.

Despite the success achieved, we identified opportunities for improvement, such as optimizing the infrastructure to reduce latency in real-time data processing and expanding the data model with new sources to further enhance the analysis.

This project highlights the importance of data engineering in efficiently managing and analyzing large volumes of information, contributing to the improvement of urban transportation services and more informed decision-making.

11. References

- [Carris API Documentation](#)
- [IPMA API](#)

12. Deliverables

- [General Repo](#)
 - [DAGs Airflow](#)
 - [Projeto dbt](#)
 - [Jobs de Spark](#)