About Delhivery

Delhivery is the largest and fastest-growing fully integrated player in India by revenue in Fiscal 2021.
Their mission is to build the **operating system for commerce**, leveraging:

- World-class **infrastructure**
- **Logistics operations** of the highest quality
- Cutting-edge **engineering** and **technology** capabilities

The **Data Team** plays a pivotal role in achieving this by:

- Building intelligence and capabilities using data
- Enhancing the quality, efficiency, and profitability of their business compared to competitors

---

Problem Statement

Delhivery aims to understand and process the data flowing from their data engineering pipelines. Here's how you can contribute:

1. **Clean, sanitize, and manipulate data** to extract useful features from raw fields
2. Derive insights from raw data to support the Data Science team in building **forecasting models**

Column Profiling

Data Dictionary

- **data**: Indicates whether the data belongs to testing or training datasets.
- **trip_creation_time**: Timestamp of when the trip was created.
- **route_schedule_uuid**: Unique identifier for a particular route schedule.
- **route_type**: Describes the transportation type:
  - **FTL (Full Truck Load)**: FTL shipments reach the destination sooner, as the truck makes no other pickups or drop-offs along the way;
  - **Carting**: A handling system consisting of small vehicles (carts).
- **trip_uuid**: Unique ID assigned to a specific trip. A trip may include different source and destination centers.
- **source_center**: Unique identifier (ID) for the origin of the trip.
- **source_name**: Name of the trip's origin.
- **destination_center**: Unique identifier (ID) for the destination of the trip.
- **destination_name**: Name of the trip's destination.
- **od_start_time**: Start time of the trip.
- **od_end_time**: End time of the trip.
- **start_scan_to_end_scan**: Total time taken to deliver from source to destination.
- **is_cutoff**: Unknown field.
- **cutoff_factor**: Unknown field.
- **cutoff_timestamp**: Unknown field.
- **actual_distance_to_destination**: Distance in kilometers between the source and destination warehouses.
- **actual_time**: Actual cumulative time taken to complete the delivery.
- **osrm_time**: Cumulative time calculated by an open-source routing engine, which computes the shortest path between points on a map. It includes usual traffic conditions and distances through major and minor roads.
- **osrm_distance**: Cumulative distance calculated by an open-source routing engine, representing the shortest path between points on a map. It includes usual traffic conditions and distances through major and minor roads.
- **factor**: Unknown field.
- **segment_actual_time**: Actual time taken by a specific segment (subset) of the package delivery.
- **segment_osrm_time**: OSRM-calculated time for a specific segment (subset) of the package delivery.
- **segment_osrm_distance**: OSRM-calculated distance for a specific segment (subset) of the package delivery.
- **segment_factor**: Unknown field.

In [1]:
```
# importing libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats

import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
# Loading the dataset and making a copy of it so that original can be intact

original_data = pd.read_csv("delhivery_data.csv")
data = original_data.copy(deep= True)
data.head()
```

Out[2]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	destination_
0	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND38862
1	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND38862
2	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND38862
3	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND38862
4	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND38862

5 rows × 24 columns

In [3]:
```
# Knowing the Shape of the data
print(f"Number of Rows in given DataSet: {data.shape[0]}")
print(f"Number of Columns in given DataSet: {data.shape[1]}")
```

```
Number of Rows in given DataSet: 144867
Number of Columns in given DataSet: 24
```

In [4]:
```
# Columns and their dtypes
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144867 entries, 0 to 144866
Data columns (total 24 columns):
 #   Column                          Non-Null Count   Dtype  
---  ------                          --------------   -----  
 0   data                            144867 non-null  object 
 1   trip_creation_time              144867 non-null  object 
 2   route_schedule_uuid             144867 non-null  object 
 3   route_type                      144867 non-null  object 
 4   trip_uuid                       144867 non-null  object 
 5   source_center                   144867 non-null  object 
 6   source_name                     144574 non-null  object 
 7   destination_center              144867 non-null  object 
 8   destination_name                144606 non-null  object 
 9   od_start_time                   144867 non-null  object 
 10  od_end_time                     144867 non-null  object 
 11  start_scan_to_end_scan          144867 non-null  float64
 12  is_cutoff                       144867 non-null  bool   
 13  cutoff_factor                   144867 non-null  int64  
 14  cutoff_timestamp                144867 non-null  object 
 15  actual_distance_to_destination  144867 non-null  float64
 16  actual_time                     144867 non-null  float64
 17  osrm_time                       144867 non-null  float64
 18  osrm_distance                   144867 non-null  float64
 19  factor                          144867 non-null  float64
 20  segment_actual_time             144867 non-null  float64
 21  segment_osrm_time               144867 non-null  float64
 22  segment_osrm_distance           144867 non-null  float64
 23  segment_factor                  144867 non-null  float64
dtypes: bool(1), float64(10), int64(1), object(12)
memory usage: 25.6+ MB
```

In [5]:
```
# percentage of null values
data.isna().sum()
```

Out[5]:
```
data                              0
trip_creation_time                0
route_schedule_uuid               0
route_type                        0
trip_uuid                         0
source_center                     0
source_name                     293
destination_center                0
destination_name                261
od_start_time                     0
od_end_time                       0
start_scan_to_end_scan            0
is_cutoff                         0
cutoff_factor                     0
cutoff_timestamp                  0
actual_distance_to_destination    0
actual_time                       0
osrm_time                         0
osrm_distance                     0
factor                            0
segment_actual_time               0
segment_osrm_time                 0
segment_osrm_distance             0
segment_factor                    0
dtype: int64
```

In [6]:
```
# Duplicated values
print(f"Duplicated_values: {data.duplicated().sum()}")
```

```
Duplicated_values: 0
```

In [7]:
```
# Checking Unique Values
data.nunique()
```

Out[7]:
```
data                                   2
trip_creation_time                 14817
route_schedule_uuid                 1504
route_type                             2
trip_uuid                          14817
source_center                       1508
source_name                         1498
destination_center                  1481
destination_name                    1468
od_start_time                      26369
od_end_time                        26369
start_scan_to_end_scan              1915
is_cutoff                              2
cutoff_factor                        501
cutoff_timestamp                   93180
actual_distance_to_destination    144515
actual_time                         3182
osrm_time                           1531
osrm_distance                     138046
factor                             45641
segment_actual_time                  747
segment_osrm_time                    214
segment_osrm_distance             113799
segment_factor                      5675
dtype: int64
```

Summary of Current Data

- **Shape**: There are total 144867 rows and 24 columns in the dataset where most the columns are unknown
- **Null Values:** There are few null values in the dataset which we can deal with them later as they are having impact of less than 0.2%. We can either drop them or fill them with most frequent value.
- **Duplicate Rows:** There are no such duplicate values that have same values in all columns in the dataset.
- **Unique_values:** There are only 3 columns which have 2 unique values, we can categorize data using them whenever it is necessary
- **Mysterious Columns**: There are few unknown columns in the dataset which we don't know what to do with them. But we can deal with them later we can remove those columns if they interfere in EDA.
- **DateTime Columns:** We have few columns that can be convert into datetime format using pandas for ease in Data Analysis.

  Let's Deal with abnormal data one by one

In [8]:
```
data.iloc[:2, :15]
# using iloc you can also check hidden cols. for rows iloc:3 gets first 3 rows(012), for cols iloc=0 is index
```

Out[8]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	destination_
0	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND3886
1	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND3886

In [9]:
```
# there are few cols that have date & time in it but they are as object dtype.

datetime_cols = ['trip_creation_time','od_start_time',
                 'od_end_time','cutoff_timestamp']

# let's convert them into datetime format
for col in datetime_cols:
    data[col] = pd.to_datetime(data[col], format='mixed')

#verify
data[datetime_cols].info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144867 entries, 0 to 144866
Data columns (total 4 columns):
 #   Column              Non-Null Count   Dtype
---  ------              --------------   -----
 0   trip_creation_time  144867 non-null  datetime64[ns]
 1   od_start_time       144867 non-null  datetime64[ns]
 2   od_end_time         144867 non-null  datetime64[ns]
 3   cutoff_timestamp    144867 non-null  datetime64[ns]
dtypes: datetime64[ns](4)
memory usage: 4.4 MB
```

In [10]:
```
# Null data percentage is less than 0.2% we can remove those rows or we can replace them with more frequent v
# For now let's just drop it

data.dropna(inplace=True, axis= 0, how='any')

# axis=0 removes the rows, subset= check in only that col, thresh= no of rows to be deleted
# how= 'all'(if all values null) or 'any'(if any value is null)

data.isna().sum() # verify
```

Out[10]:
```
data                              0
trip_creation_time                0
route_schedule_uuid               0
route_type                        0
trip_uuid                         0
source_center                     0
source_name                       0
destination_center                0
destination_name                  0
od_start_time                     0
od_end_time                       0
start_scan_to_end_scan            0
is_cutoff                         0
cutoff_factor                     0
cutoff_timestamp                  0
actual_distance_to_destination    0
actual_time                       0
osrm_time                         0
osrm_distance                     0
factor                            0
segment_actual_time               0
segment_osrm_time                 0
segment_osrm_distance             0
segment_factor                    0
dtype: int64
```

Since delivery details of one package are divided into several rows (think of it as connecting flights to reach a particular destination). Now think about how we should treat their fields if we combine these rows? What aggregation would make sense if we merge. What would happen to the numeric fields if we merge the rows?

Create a unique identifier for different segments of a trip based on the combination of the trip_uuid, source_center, and destination_center and name it as segment_key.

In [11]:
```
# creating a new column as segmentkey by combining the below columns to group easily
data['segment_key'] = data["trip_uuid"]+"_"+data["source_center"]+"_"+data['destination_center']
```

In [12]:
```
aggregated_data = data.groupby(by=['segment_key']).agg(
    segment_actual_time_sum= ("segment_actual_time", "sum"),
    segment_osrm_time_sum = ("segment_osrm_time","sum"),
    segment_osrm_distance_sum = ("segment_osrm_distance","sum")).reset_index()

aggregated_data.head()
```

Out[12]:

	segment_key	segment_actual_time_sum	segment_osrm_time_sum	segment_osrm_distance_sum
0	trip-153671041653548748_IND209304AAA_IND000000ACB	728.0	534.0	670.6205
1	trip-153671041653548748_IND462022AAA_IND209304AAA	820.0	474.0	649.8528
2	trip-153671042288605164_IND561203AAB_IND562101AAA	46.0	26.0	28.1995
3	trip-153671042288605164_IND572101AAA_IND561203AAB	95.0	39.0	55.9899
4	trip-153671043369099517_IND000000ACB_IND160002AAC	608.0	231.0	317.7408

In [13]:
```
data['cumsum_segment_actual_time'] = data.groupby('segment_key')['segment_actual_time'].cumsum()
data['cumsum_segment_osrm_time'] = data.groupby('segment_key')['segment_osrm_time'].cumsum()
data['cumsum_segment_osrm_distance'] = data.groupby('segment_key')['segment_osrm_distance'].cumsum()

data.head(20)
```

Out[13]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	destin
0	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	INI
1	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	INI
2	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	INI
3	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	INI
4	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	INI
5	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	INI
6	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	INI
7	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	INI
8	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	INI
9	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	INI
10	training	2018-09-23 06:42:06.021680	thanos::sroute:ff52ef7a-4d0d-4063-9bfe-cc21172...	FTL	trip-153768492602129387	IND421302AAG	Bhiwandi_Mankoli_HB (Maharashtra)	INI
11	training	2018-09-23 06:42:06.021680	thanos::sroute:ff52ef7a-4d0d-4063-9bfe-cc21172...	FTL	trip-153768492602129387	IND421302AAG	Bhiwandi_Mankoli_HB (Maharashtra)	INI
12	training	2018-09-23 06:42:06.021680	thanos::sroute:ff52ef7a-4d0d-4063-9bfe-cc21172...	FTL	trip-153768492602129387	IND421302AAG	Bhiwandi_Mankoli_HB (Maharashtra)	INI
13	training	2018-09-23 06:42:06.021680	thanos::sroute:ff52ef7a-4d0d-4063-9bfe-cc21172...	FTL	trip-153768492602129387	IND421302AAG	Bhiwandi_Mankoli_HB (Maharashtra)	INI
14	training	2018-09-23 06:42:06.021680	thanos::sroute:ff52ef7a-4d0d-4063-9bfe-cc21172...	FTL	trip-153768492602129387	IND421302AAG	Bhiwandi_Mankoli_HB (Maharashtra)	INI
15	training	2018-09-14 15:42:46.437249	thanos::sroute:a16bfa03-3462-4bce-9c82-5784c7d...	Carting	trip-153693976643699843	IND400011AAA	LowerParel_CP (Maharashtra)	INI
16	training	2018-09-14 15:42:46.437249	thanos::sroute:a16bfa03-3462-4bce-9c82-5784c7d...	Carting	trip-153693976643699843	IND400011AAA	LowerParel_CP (Maharashtra)	INI
17	training	2018-09-13 20:44:19.424489	thanos::sroute:76951383-1608-44e4-a284-46d92e8...	FTL	trip-153687145942424248	IND562132AAA	Bangalore_Nelmngla_H (Karnataka)	INI
18	training	2018-09-13 20:44:19.424489	thanos::sroute:76951383-1608-44e4-a284-46d92e8...	FTL	trip-153687145942424248	IND562132AAA	Bangalore_Nelmngla_H (Karnataka)	INI
19	training	2018-09-13 20:44:19.424489	thanos::sroute:76951383-1608-44e4-a284-46d92e8...	FTL	trip-153687145942424248	IND560099AAB	Bengaluru_Bomsndra_HB (Karnataka)	INI

20 rows × 28 columns

2: Aggregating at Segment Level

- **a. Create a dictionary**: Define a dictionary named `create_segment_dict` to specify how to aggregate and select values:
  - Use the **first** and **last values** for some numeric/categorical fields where aggregation doesn't make sense.
- **b. Group the data by `segment_key`**: Group the data to perform aggregation operations for different segments of each trip based on the `segment_key` value.
- **c. Apply aggregation functions**: Use the aggregation functions defined in `create_segment_dict` for each group of rows sharing the same `segment_key`.
- **d. Sort the resulting DataFrame**:
  - **i. Sort by `segment_key`**: Ensures segments are ordered consistently.

- **ii. Sort by `od_end_time` (ascending)**: Ensures segments within the same trip are ordered by their end times from earliest

In [14]:
```
data.columns
```

Out[14]:
```
Index(['data', 'trip_creation_time', 'route_schedule_uuid', 'route_type',
       'trip_uuid', 'source_center', 'source_name', 'destination_center',
       'destination_name', 'od_start_time', 'od_end_time',
       'start_scan_to_end_scan', 'is_cutoff', 'cutoff_factor',
       'cutoff_timestamp', 'actual_distance_to_destination', 'actual_time',
       'osrm_time', 'osrm_distance', 'factor', 'segment_actual_time',
       'segment_osrm_time', 'segment_osrm_distance', 'segment_factor',
       'segment_key', 'cumsum_segment_actual_time', 'cumsum_segment_osrm_time',
       'cumsum_segment_osrm_distance'],
      dtype='object')
```

Why Define Aggregation Rules?

When rows represent parts of a larger entity (e.g., segments of a delivery trip), condensing them into one row per group (e.g., one row per segment) is essential. During this process:

- **Numeric Fields**: Represent totals, averages, or other summaries.
- **Categorical Fields**: Require a meaningful representation, such as the first or most frequent value.
- **Timestamps**: Must maintain chronological relevance, such as start and end times.

Without proper aggregation rules:

- The resulting data might lose important context.
- It may become inconsistent, reducing its usefulness for analysis or modeling.

In [15]:
```
create_segment_dict = {

    'data': "first",
    'trip_creation_time': "first",
    'route_schedule_uuid': "first",
    'route_type': "first",
    'trip_uuid': "first",
    'source_center': "first",
    'source_name': "first",

    'destination_center': "last",
    'destination_name': "last",

    'od_start_time': "first",
    'od_end_time': "first",
    'start_scan_to_end_scan': "first",

    'actual_distance_to_destination': "last",
    'actual_time': "last",

    'osrm_time': "first",
    'osrm_distance': "first",

    'cumsum_segment_actual_time': "last",
    'cumsum_segment_osrm_time': "last",
    'cumsum_segment_osrm_distance': "last"
}
```

In [16]:
```
segmented_data = data.groupby(by= 'segment_key').agg(create_segment_dict).reset_index()
```

In [17]:
```python
# sorting the data by segment_key: Ensuring segments are ordered consistently.
# Sorting by od_end_time (ascending): Ensures segments within the same trip are ordered by their end times fr

segmented_data = segmented_data.sort_values(by=['segment_key', 'od_end_time'], ascending = [True, True])
segmented_data.head(20)
```

Out[17]:

	segment_key	data	trip_creation_time	route_schedule_uuid	route_type	trip_u
0	trip-153671041653548748_IND209304AAA_IND000000ACB	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	t 153671041653548
1	trip-153671041653548748_IND462022AAA_IND209304AAA	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	t 153671041653548
2	trip-153671042288605164_IND561203AAB_IND562101AAA	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	t 153671042288605
3	trip-153671042288605164_IND572101AAA_IND561203AAB	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	t 153671042288605
4	trip-153671043369099517_IND000000ACB_IND160002AAC	training	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	FTL	t 153671043369099
5	trip-153671043369099517_IND562132AAA_IND000000ACB	training	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	FTL	t 153671043369099
6	trip-153671046011330457_IND400072AAB_IND401104AAA	training	2018-09-12 00:01:00.113710	thanos::sroute:f0176492-a679-4597-8332-bbd1c7f...	Carting	t 153671046011330
7	trip-153671052974046625_IND583101AAA_IND583201AAA	training	2018-09-12 00:02:09.740725	thanos::sroute:d9f07b12-65e0-4f3b-bec8-df06134...	FTL	t 153671052974046
8	trip-153671052974046625_IND583119AAA_IND583101AAA	training	2018-09-12 00:02:09.740725	thanos::sroute:d9f07b12-65e0-4f3b-bec8-df06134...	FTL	t 153671052974046
9	trip-153671052974046625_IND583201AAA_IND583119AAA	training	2018-09-12 00:02:09.740725	thanos::sroute:d9f07b12-65e0-4f3b-bec8-df06134...	FTL	t 153671052974046
10	trip-153671055416136166_IND600056AAA_IND602105AAB	training	2018-09-12 00:02:34.161600	thanos::sroute:9bf03170-d0a2-4a3f-aa4d-9aaab3d...	Carting	t 153671055416136
11	trip-153671055416136166_IND600116AAB_IND600056AAA	training	2018-09-12 00:02:34.161600	thanos::sroute:9bf03170-d0a2-4a3f-aa4d-9aaab3d...	Carting	t 153671055416136
12	trip-153671066201138152_IND600044AAD_IND600048AAA	training	2018-09-12 00:04:22.011653	thanos::sroute:a97698cc-846e-41a7-916b-88b1741...	Carting	t 153671066201138
13	trip-153671066826362165_IND560043AAC_IND560064AAA	training	2018-09-12 00:04:28.263977	thanos::sroute:d5b71ae9-a11a-4f52-bcb7-274b65e...	Carting	t 153671066826362
14	trip-153671066826362165_IND560064AAD_IND560043AAC	training	2018-09-12 00:04:28.263977	thanos::sroute:d5b71ae9-a11a-4f52-bcb7-274b65e...	Carting	t 153671066826362
15	trip-153671074033284934_IND395009AAA_IND395023AAD	training	2018-09-12 00:05:40.333071	thanos::sroute:a0e60427-16ad-4b17-b3b0-6a06643...	Carting	t 153671074033284
16	trip-153671074033284934_IND395023AAD_IND395004AAB	training	2018-09-12 00:05:40.333071	thanos::sroute:a0e60427-16ad-4b17-b3b0-6a06643...	Carting	t 153671074033284
17	trip-153671079956500691_IND110024AAA_IND110014AAA	training	2018-09-12 00:06:39.565253	thanos::sroute:a10888ff-f794-41e1-9b7a-7f62ef6...	Carting	t 153671079956500
18	trip-153671090980523004_IND412105AAC_IND411017AAA	training	2018-09-12 00:08:29.805514	thanos::sroute:580c788b-ff17-4c1b-9bbd-c59e7b0...	Carting	t 153671090980523
19	trip-153671110078355292_IND121004AAB_IND121001AAA	training	2018-09-12 00:11:40.783923	thanos::sroute:c2ee580f-f4b2-4fa5-98ab-0c5b327...	Carting	t 153671110078355

In [18]:
```python
segmented_data.shape
```

Out[18]: (26222, 20)

In [19]:
```
segmented_data[segmented_data['trip_uuid'] == 'trip-153741093647649320']
```

Out[19]:

	segment_key	data	trip_creation_time	route_schedule_uuid	route_type	trip
10370	trip-153741093647649320_IND388121AAA_IND388620AAB	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	1537410936476
10371	trip-153741093647649320_IND388620AAB_IND388320AAA	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	1537410936476

In [20]:
```
segmented_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26222 entries, 0 to 26221
Data columns (total 20 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   segment_key                     26222 non-null  object
 1   data                            26222 non-null  object
 2   trip_creation_time              26222 non-null  datetime64[ns]
 3   route_schedule_uuid             26222 non-null  object
 4   route_type                      26222 non-null  object
 5   trip_uuid                       26222 non-null  object
 6   source_center                   26222 non-null  object
 7   source_name                     26222 non-null  object
 8   destination_center              26222 non-null  object
 9   destination_name                26222 non-null  object
 10  od_start_time                   26222 non-null  datetime64[ns]
 11  od_end_time                     26222 non-null  datetime64[ns]
 12  start_scan_to_end_scan          26222 non-null  float64
 13  actual_distance_to_destination  26222 non-null  float64
 14  actual_time                     26222 non-null  float64
 15  osrm_time                       26222 non-null  float64
 16  osrm_distance                   26222 non-null  float64
 17  cumsum_segment_actual_time      26222 non-null  float64
 18  cumsum_segment_osrm_time        26222 non-null  float64
 19  cumsum_segment_osrm_distance    26222 non-null  float64
dtypes: datetime64[ns](3), float64(8), object(9)
memory usage: 4.0+ MB
```

In [21]:
```
segmented_data.nunique()
```

Out[21]:
```
segment_key                       26222
data                                  2
trip_creation_time                14787
route_schedule_uuid                1497
route_type                            2
trip_uuid                         14787
source_center                      1496
source_name                        1496
destination_center                 1466
destination_name                   1466
od_start_time                     26222
od_end_time                       26222
start_scan_to_end_scan             1914
actual_distance_to_destination    26193
actual_time                        1657
osrm_time                           150
osrm_distance                     24511
cumsum_segment_actual_time         1676
cumsum_segment_osrm_time           1102
cumsum_segment_osrm_distance      25948
dtype: int64
```

3. Feature Engineering

Extract features from the following fields:

1. **Calculate time difference**:
    - Compute the time taken between `od_start_time` and `od_end_time`.
    - Store the result as a new feature named `od_time_diff_hour`.
    - Drop the original columns, if required.
2. **Destination Name**:
    - Split and extract features from `destination_name` in the format: **City-Place-Code (State)**.
3. **Source Name**:
    - Split and extract features from `source_name` in the format: **City-Place-Code (State)**.
4. **Trip_creation_time**:
    - Extract temporal features such as **month**, **year**, **day**, etc.

In [22]:
```
segmented_data["od_time_diff_hour"] = (segmented_data['od_end_time'] - segmented_data['od_start_time']).dt.to
```

In [23]:
```
# Let's drop the `od_start_time` and `od_end_time` columns

segmented_data.drop(labels= ['od_start_time', 'od_end_time'], axis= 1, inplace= True)
segmented_data.head()
```

Out[23]:

	segment_key	data	trip_creation_time	route_schedule_uuid	route_type	trip_uu
0	trip-153671041653548748_IND209304AAA_IND000000ACB	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	tri 15367104165354874
1	trip-153671041653548748_IND462022AAA_IND209304AAA	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	tri 15367104165354874
2	trip-153671042288605164_IND561203AAB_IND562101AAA	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	tri 15367104228860516
3	trip-153671042288605164_IND572101AAA_IND561203AAB	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	tri 15367104228860516
4	trip-153671043369099517_IND000000ACB_IND160002AAC	training	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	FTL	tri 1536710433690995

In [24]:
```
# Split and extract features from `destination_name` in the format: City-Place-Code (State)
import re

def extract_state(value): #to extract state from the name
    part1, state = value.split('(')
    state = state[:-1]
    return part1, state

def split_values(value):
    parts = re.split(r"[\s_]+",value, maxsplit= 2)
    if len(parts) == 3:
        return parts[0], parts[1], parts[2]  # city, place, code
    elif len(parts) == 2:
        return parts[0], parts[1], None  # city, place, None for missing code
    elif len(parts) == 1:
        return parts[0], None, None  # city, None, None for missing place and code
```

In [25]:
```
# let's apply the functions to Source column

segmented_data[['source_place_city_code', 'source_state']]= segmented_data['source_name'].apply(extract_state

segmented_data[['source_city','source_place','source_code']]= segmented_data['source_place_city_code'].apply(
```

In [26]:
```
# let's apply the functions to destination column

segmented_data[['destination_place_city_code', 'destination_state']]= segmented_data['destination_name'].appl

segmented_data[['destination_city','destination_place','destination_code']]= segmented_data['destination_plac
```

In [27]:
```
# drop the extra columns also the original source and desitnation columns as we have that data with us.

segmented_data.drop(labels=['source_place_city_code','destination_place_city_code' ], axis=1, inplace= True)
```

In [28]:
```
# Trip_creation_time: Extract features like month, year, day, etc.

segmented_data['trip_creation_day'] = segmented_data['trip_creation_time'].dt.day
segmented_data['trip_creation_month'] = segmented_data['trip_creation_time'].dt.month
segmented_data['trip_creation_year'] = segmented_data['trip_creation_time'].dt.year
```

In [29]:
```
segmented_data.head()
```

Out[29]:

	segment_key	data	trip_creation_time	route_schedule_uuid	route_type	trip_uu
0	trip-153671041653548748_IND209304AAA_IND000000ACB	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	tri 15367104165354874
1	trip-153671041653548748_IND462022AAA_IND209304AAA	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	tri 15367104165354874
2	trip-153671042288605164_IND561203AAB_IND562101AAA	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	tri 15367104228860516
3	trip-153671042288605164_IND572101AAA_IND561203AAB	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	tri 15367104228860516
4	trip-153671043369099517_IND000000ACB_IND160002AAC	training	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	FTL	tri 1536710433690995

5 rows × 30 columns

In [30]:
```
# there are few categorical columns in the dataframe let's identify them and convert them to categorical
cat_cols= ["data", "route_type"]

for col in segmented_data.columns:
    if 'state' in col or 'place' in col or 'code' in col or 'city' in col:
        segmented_data[col] = segmented_data[col].astype('category')

for col in cat_cols:
    segmented_data[col] = segmented_data[col].astype('category')

segmented_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26222 entries, 0 to 26221
Data columns (total 30 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   segment_key                     26222 non-null  object
 1   data                            26222 non-null  category
 2   trip_creation_time              26222 non-null  datetime64[ns]
 3   route_schedule_uuid             26222 non-null  object
 4   route_type                      26222 non-null  category
 5   trip_uuid                       26222 non-null  object
 6   source_center                   26222 non-null  object
 7   source_name                     26222 non-null  object
 8   destination_center              26222 non-null  object
 9   destination_name                26222 non-null  object
 10  start_scan_to_end_scan          26222 non-null  float64
 11  actual_distance_to_destination  26222 non-null  float64
 12  actual_time                     26222 non-null  float64
 13  osrm_time                       26222 non-null  float64
 14  osrm_distance                   26222 non-null  float64
 15  cumsum_segment_actual_time      26222 non-null  float64
 16  cumsum_segment_osrm_time        26222 non-null  float64
 17  cumsum_segment_osrm_distance    26222 non-null  float64
 18  od_time_diff_hour               26222 non-null  float64
 19  source_state                    26222 non-null  category
 20  source_city                     26222 non-null  category
 21  source_place                    26222 non-null  category
 22  source_code                     25880 non-null  category
 23  destination_state               26222 non-null  category
 24  destination_city                26222 non-null  category
 25  destination_place               26222 non-null  category
 26  destination_code                25776 non-null  category
 27  trip_creation_day               26222 non-null  int32
 28  trip_creation_month             26222 non-null  int32
 29  trip_creation_year              26222 non-null  int32
dtypes: category(10), datetime64[ns](1), float64(9), int32(3), object(7)
memory usage: 4.2+ MB
```

In [31]:
```
# after applying the functions we got few null values in code columns
segmented_data.isna().sum()
```

Out[31]:
```
segment_key                        0
data                               0
trip_creation_time                 0
route_schedule_uuid                0
route_type                         0
trip_uuid                          0
source_center                      0
source_name                        0
destination_center                 0
destination_name                   0
start_scan_to_end_scan             0
actual_distance_to_destination     0
actual_time                        0
osrm_time                          0
osrm_distance                      0
cumsum_segment_actual_time         0
cumsum_segment_osrm_time           0
cumsum_segment_osrm_distance       0
od_time_diff_hour                  0
source_state                       0
source_city                        0
source_place                       0
source_code                      342
destination_state                  0
destination_city                   0
destination_place                  0
destination_code                 446
trip_creation_day                  0
trip_creation_month                0
trip_creation_year                 0
dtype: int64
```

Source Code and desitnation code have few null values init. As it doesn't have much use as of now, we can either drop that column or we can change it accordance with place values. But for now let's just ignore it. It's doesn't have any impact on further analysis.

4. In-Depth Analysis

1. Grouping and Aggregating at Trip-Level

- **a. Group by `trip_uuid`** :
  Group the segment data by the `trip_uuid` column to focus on aggregating data at the trip level.
- **b. Apply aggregation functions**:
  Use aggregation functions like `first` , `last` , and `sum` as specified in the `create_trip_dict` dictionary to calculate summary statistics for each trip.

In [32]:
```
create_trip_dict = {

    'data': "first",
    'trip_creation_time': "first",
    'route_schedule_uuid': "first",
    'route_type': "first",
    'trip_uuid': "first",

    'source_center': "first",
    'source_name': "first",
    'source_city' : "first",
    'source_place': 'first',
    'source_state': 'first',

    'destination_center': "last",
    'destination_name': "last",
    'destination_city': 'last',
    'destination_place': 'last',
    'destination_state': 'last',

    'start_scan_to_end_scan': "sum",
    'od_time_diff_hour': 'sum',


    'actual_distance_to_destination': "sum",
    'actual_time': "sum",

    'osrm_time': "sum",
    'osrm_distance': "sum",

    'cumsum_segment_actual_time': "sum",
    'cumsum_segment_osrm_time': "sum",
    'cumsum_segment_osrm_distance': "sum"
}

trip_level_data = segmented_data.groupby('trip_uuid').agg(create_trip_dict).reset_index(drop= True)
```

In [33]:
```
trip_level_data.head()
```

Out[33]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	source_
0	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	trip-153671041653548748	IND209304AAA	Kanpur_Central_H_6 (Uttar Pradesh)	Ka
1	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	trip-153671042288605164	IND561203AAB	Doddablpur_ChikaDPP_D (Karnataka)	Doddab
2	training	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	FTL	trip-153671043369099517	IND000000ACB	Gurgaon_Bilaspur_HB (Haryana)	Gurg
3	training	2018-09-12 00:01:00.113710	thanos::sroute:f0176492-a679-4597-8332-bbd1c7f...	Carting	trip-153671046011330457	IND400072AAB	Mumbai Hub (Maharashtra)	Mur
4	training	2018-09-12 00:02:09.740725	thanos::sroute:d9f07b12-65e0-4f3b-bec8-df06134...	FTL	trip-153671052974046625	IND583101AAA	Bellary_Dc (Karnataka)	Be

5 rows × 24 columns

In [34]:
```
# there are few categorical columns in the dataframe let's identify them and convert them to categorical
cat_cols= ["data", "route_type"]

for col in trip_level_data.columns:
    if 'state' in col or 'place' in col or 'code' in col or 'city' in col:
        trip_level_data[col] = trip_level_data[col].astype('category')

for col in cat_cols:
    trip_level_data[col] = trip_level_data[col].astype('category')

trip_level_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14787 entries, 0 to 14786
Data columns (total 24 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   data                            14787 non-null  category
 1   trip_creation_time              14787 non-null  datetime64[ns]
 2   route_schedule_uuid             14787 non-null  object
 3   route_type                      14787 non-null  category
 4   trip_uuid                       14787 non-null  object
 5   source_center                   14787 non-null  object
 6   source_name                     14787 non-null  object
 7   source_city                     14787 non-null  category
 8   source_place                    14787 non-null  category
 9   source_state                    14787 non-null  category
 10  destination_center              14787 non-null  object
 11  destination_name                14787 non-null  object
 12  destination_city                14787 non-null  category
 13  destination_place               14787 non-null  category
 14  destination_state               14787 non-null  category
 15  start_scan_to_end_scan          14787 non-null  float64
 16  od_time_diff_hour               14787 non-null  float64
 17  actual_distance_to_destination  14787 non-null  float64
 18  actual_time                     14787 non-null  float64
 19  osrm_time                       14787 non-null  float64
 20  osrm_distance                   14787 non-null  float64
 21  cumsum_segment_actual_time      14787 non-null  float64
 22  cumsum_segment_osrm_time        14787 non-null  float64
 23  cumsum_segment_osrm_distance    14787 non-null  float64
dtypes: category(8), datetime64[ns](1), float64(9), object(6)
memory usage: 2.1+ MB
```

In [35]:
```
trip_level_data.describe().T
```

Out[35]:

	count	mean	min	25%	50%	75
trip_creation_time	14787	2018-09-22 12:26:28.269885696	2018-09-12 00:00:16.535741	2018-09-17 02:38:18.128431872	2018-09-22 03:39:19.609193984	2018-09-2 19:23:14.0743595
start_scan_to_end_scan	14787.0	529.429025	23.0	149.0	279.0	632
od_time_diff_hour	14787.0	8.838559	0.391024	2.494975	4.661846	10.5589
actual_distance_to_destination	14787.0	164.090196	9.002461	22.777099	48.287894	163.5912
actual_time	14787.0	356.306012	9.0	67.0	148.0	367
osrm_time	14787.0	36.193887	6.0	13.0	21.0	45
osrm_distance	14787.0	40.698444	9.051	12.94055	24.4979	53.293
cumsum_segment_actual_time	14787.0	353.059174	9.0	66.0	147.0	364
cumsum_segment_osrm_time	14787.0	180.511598	6.0	30.0	65.0	184
cumsum_segment_osrm_distance	14787.0	222.705466	9.0729	32.57885	69.7842	216.56

In [36]:
```
trip_level_data.describe(include=[ 'object', 'category']).T
```

Out[36]:

	count	unique	top	freq
data	14787	2	training	10645
route_schedule_uuid	14787	1497	thanos::sroute:a16bfa03-3462-4bce-9c82-5784c7d...	53
route_type	14787	2	Carting	8906
trip_uuid	14787	14787	trip-153671041653548748	1
source_center	14787	930	IND000000ACB	1052
source_name	14787	930	Gurgaon_Bilaspur_HB (Haryana)	1052
source_city	14787	714	Gurgaon	1128
source_place	14787	710	Bilaspur	1074
source_state	14787	29	Maharashtra	2714
destination_center	14787	1035	IND000000ACB	821
destination_name	14787	1035	Gurgaon_Bilaspur_HB (Haryana)	821
destination_city	14787	840	Mumbai	1202
destination_place	14787	803	Bilaspur	864
destination_state	14787	31	Maharashtra	2561

Insights and Observations:

1. **Training vs Test Data**: Most data is from the "training" set (10,645 records). Ensure both datasets are balanced for model validation.
2. **Route Data**: 1,497 unique routes, with some routes used more frequently (53 times). Focus on optimizing the most common routes.
3. **Route Type**: "Carting" is the most common type (8,906 occurrences). Analyze operational factors to improve efficiency in this category.
4. **Unique Trips**: Each trip is unique (14,787 different trip UUIDs). Track individual trips for performance and optimization.
5. **Source & Destination Locations**: Common source cities include "Gurgaon" and destination cities like "Mumbai". Focus on improving operations in these high-demand areas. Maharashtra is the most frequent source and destination state, suggesting a regional focus for improvements.

Actionable Items:

1. **Route Optimization**: Focus on high-frequency routes to optimize delivery times and reduce costs.
2. **Resource Allocation**: Ensure that high-usage centers (like "Gurgaon_Bilaspur_HB") are well-resourced for efficient operations.
3. **Focus on High-Demand Areas**: Prioritize improvements in cities/states with high delivery volumes (e.g., Mumbai, Maharashtra).
4. **Segment Data by Route Type**: Analyze "Carting" performance and compare it with other route types to identify improvement opportunities.
5. **Geospatial Analysis**: Visualize delivery data on a map to optimize fleet distribution and reduce delivery time.
6. **Performance at Source Locations**: Review and optimize operations at common source locations (e.g., "Gurgaon") to reduce delays.
7. **Handling Rare Locations**: Investigate uncommon source/destination centers for efficient handling.
8. **Data Consistency**: Check for mismatches in source and destination locations to ensure data accuracy.

Conclusion: By focusing on high-frequency routes, locations, and optimizing the "Carting" type, Delhivery can improve operational efficiency, reduce costs, and ensure better resource allocation

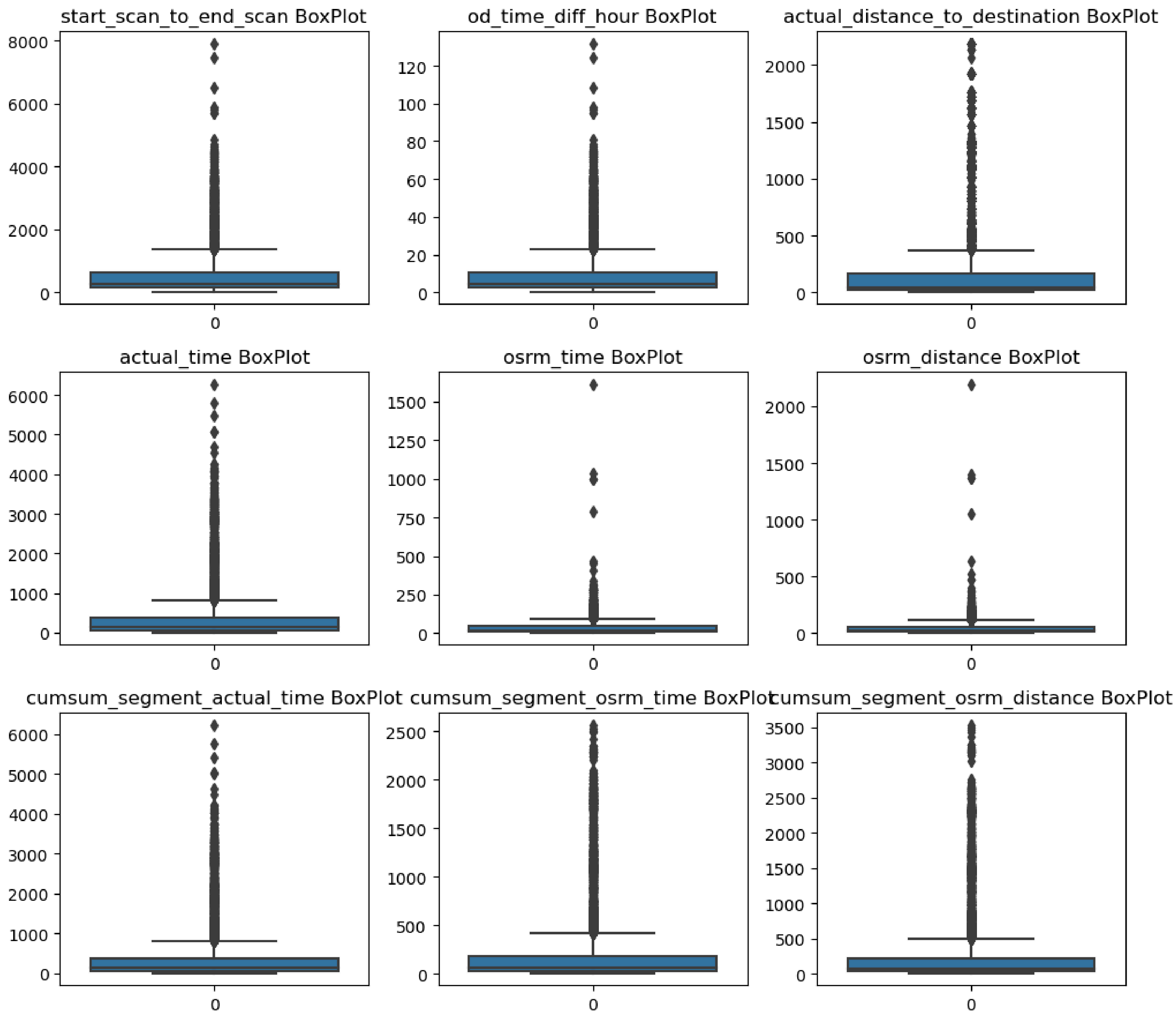
2. Outlier Detection & Treatment

- **a. Identify outliers**:
  Detect existing outliers in numerical features.
- **b. Visualize outliers**:
  Use **Boxplot** to visualize the outlier values.
- **c. Handle outliers**:
  Address outliers using the **IQR method** (Interquartile

In [37]:
```
# Let's get the numerical cols from the dataframe and check for outliers
numerical_features = []


for col in trip_level_data.columns:
    if trip_level_data[col].dtypes=='float64':
        numerical_features.append(col)

numerical_features
```

Out[37]:
```
['start_scan_to_end_scan',
 'od_time_diff_hour',
 'actual_distance_to_destination',
 'actual_time',
 'osrm_time',
 'osrm_distance',
 'cumsum_segment_actual_time',
 'cumsum_segment_osrm_time',
 'cumsum_segment_osrm_distance']
```

In [38]:
```
# Detecting outliers in numberical features using Boxplot

plt.figure(figsize= (10,9))

for i,col in enumerate(numerical_features,1):
    plt.subplot(3,3,i)
    sns.boxplot(trip_level_data[col])
    plt.title(f"{col} BoxPlot")


plt.tight_layout()
plt.show()
```

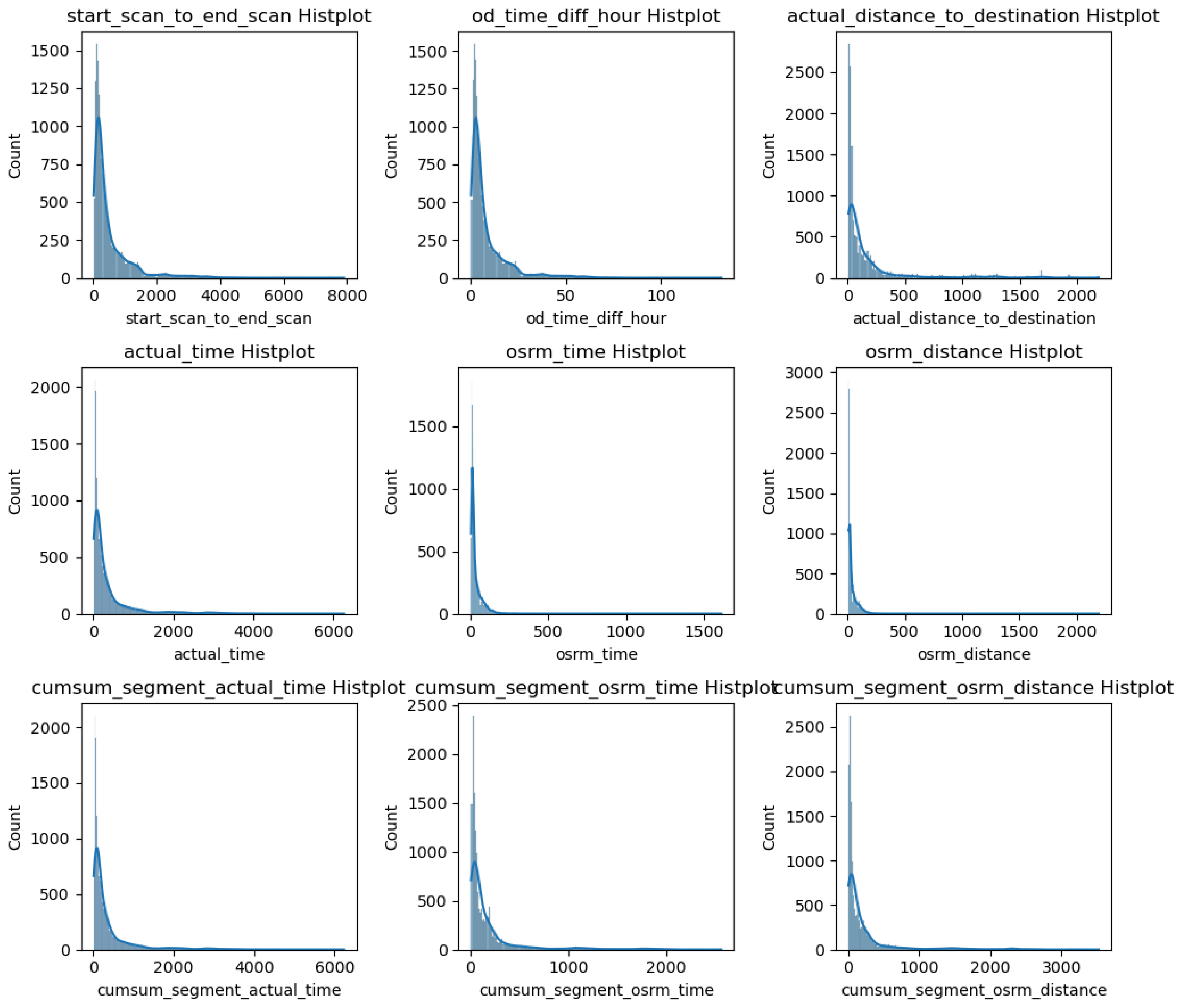


There are huge number of outliers in the Numerical Columns, Let's get the total count and percentage of those outliers

In [39]:
```
# let's know how much space are outliers taking in the data set.

for col in numerical_features:
    q1 = trip_level_data[col].quantile(0.25)
    q3 = trip_level_data[col].quantile(0.75)
    IQR = q3-q1

    lower_bound = q1 - 1.5*IQR
    upper_bound = q3 + 1.5*IQR
    no_of_outliers = len(trip_level_data[(trip_level_data[col] <= lower_bound )|( trip_level_data[col] >= upp

    print(f"The \033[1m{col}\033[0m has {no_of_outliers} number of outliers in it")
    percent = no_of_outliers/len(trip_level_data)
    print(f"The total pecentage of outliers is {np.round(percent*100,2)}%\n")
```

```
The start_scan_to_end_scan has 1282 number of outliers in it
The total pecentage of outliers is 8.67%

The od_time_diff_hour has 1275 number of outliers in it
The total pecentage of outliers is 8.62%

The actual_distance_to_destination has 1452 number of outliers in it
The total pecentage of outliers is 9.82%

The actual_time has 1648 number of outliers in it
The total pecentage of outliers is 11.14%

The osrm_time has 1229 number of outliers in it
The total pecentage of outliers is 8.31%

The osrm_distance has 995 number of outliers in it
The total pecentage of outliers is 6.73%

The cumsum_segment_actual_time has 1646 number of outliers in it
The total pecentage of outliers is 11.13%

The cumsum_segment_osrm_time has 1492 number of outliers in it
The total pecentage of outliers is 10.09%

The cumsum_segment_osrm_distance has 1550 number of outliers in it
The total pecentage of outliers is 10.48%
```

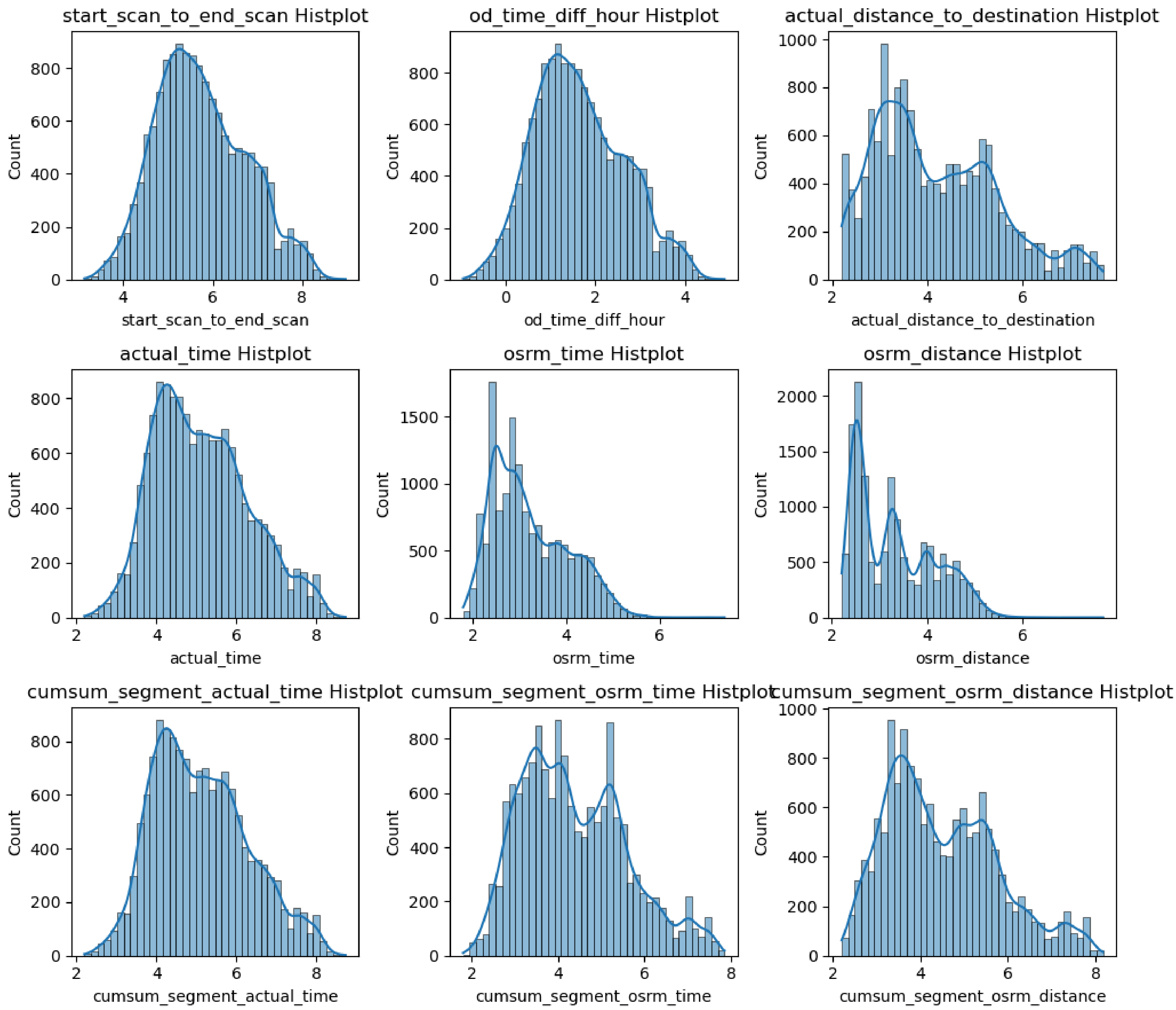
In [40]:
```
# let's check the distribution of the data

plt.figure(figsize= (10,9))

for i,col in enumerate(numerical_features,1):
    plt.subplot(3,3,i)
    sns.histplot(trip_level_data[col], kde= True)
    plt.title(f"{col} Histplot")


plt.tight_layout()
plt.show()
```



Since most of the data is positively skewed with long tails and extreme outliers, we plan to apply a log transformation to the numerical features.This transformation will help reduce skewness, bring the data closer to a normal distribution, and stabilize variances, making it more suitable for analysis and hypothesis testing.

In [41]:
```
transformed_data = trip_level_data.copy(deep= True)
```

In [42]:
```
for col in numerical_features:
    transformed_data[col] = np.log(transformed_data[col])

plt.figure(figsize= (10,9))

for i,col in enumerate(numerical_features,1):
    plt.subplot(3,3,i)
    sns.histplot(transformed_data[col], kde= True, bins = 40)
    plt.title(f"{col} Histplot")


plt.tight_layout()
plt.show()
```



In [43]:
```
# let's know how much space are outliers taking in the data set.

for col in numerical_features:
    q1 = transformed_data[col].quantile(0.25)
    q3 = transformed_data[col].quantile(0.75)
    IQR = q3-q1

    lower_bound = q1 - 1.5*IQR
    upper_bound = q3 + 1.5*IQR
    no_of_outliers = len(transformed_data[(transformed_data[col] <= lower_bound )|( transformed_data[col] >=

    print(f"The \033[1m{col}\033[0m has {no_of_outliers} number of outliers in it")
    percent = no_of_outliers/len(transformed_data)
    print(f"The total pecentage of outliers is {np.round(percent*100,2)}%\n")
```

```
The start_scan_to_end_scan has 7 number of outliers in it
The total pecentage of outliers is 0.05%

The od_time_diff_hour has 7 number of outliers in it
The total pecentage of outliers is 0.05%

The actual_distance_to_destination has 0 number of outliers in it
The total pecentage of outliers is 0.0%

The actual_time has 5 number of outliers in it
The total pecentage of outliers is 0.03%

The osrm_time has 14 number of outliers in it
The total pecentage of outliers is 0.09%

The osrm_distance has 9 number of outliers in it
The total pecentage of outliers is 0.06%

The cumsum_segment_actual_time has 5 number of outliers in it
The total pecentage of outliers is 0.03%

The cumsum_segment_osrm_time has 0 number of outliers in it
The total pecentage of outliers is 0.0%

The cumsum_segment_osrm_distance has 0 number of outliers in it
The total pecentage of outliers is 0.0%
```

In [44]:
```
# a function to detect outliers and remove them
def detect_outliers(value, col_name):

    value= pd.Series(value)

    q1 = value.quantile(0.25)
    q3 = value.quantile(0.75)
    IQR = q3-q1

    lower_bound = q1 - 1.5*IQR
    upper_bound = q3 + 1.5*IQR

    outliers = len(value[(value < lower_bound) | (value > upper_bound)])

    print(f"This \033[1m{col_name}\033[0m have {outliers} number of outliers in it which was {np.round((outli
```

One-Hot Encoding of Categorical Features:

- Convert categorical variables into a numerical format using **One-Hot Encoding**.
- This transformation creates binary columns for each category, making it easier for machine learning models to process categorical data.
- Example: For a `route_type` feature with categories like "Carting" and "FTL", create separate columns for each, where a `1` indicates the presence of that category, and a `0` indicates absence.

Normalization and Standardization of Numerical Features:

- **Normalization (MinMaxScaler)**:
  Scale the numerical features to a fixed range (usually [0, 1]). This method is particularly useful when features have different units or magnitudes, ensuring each feature contributes equally to the model.
  Example: Scaling delivery times from a range of 10 to 1000 to [0, 1].
- **Standardization (StandardScaler)**:
  Transform the numerical features so they have a mean of 0 and a standard deviation of 1. This is helpful when features follow a Gaussian distribution and when outliers are less of a concern.
  Example: Standardizing trip durations with a mean of 0 and a variance of 1, making features more comparable for machine learning models.
- Choose between **Normalization** or **Standardization** based on the distribution and scale of the data to prepare it for analysis and modeling.

In [45]:
```
trip_level_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14787 entries, 0 to 14786
Data columns (total 24 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   data                            14787 non-null  category
 1   trip_creation_time              14787 non-null  datetime64[ns]
 2   route_schedule_uuid             14787 non-null  object
 3   route_type                      14787 non-null  category
 4   trip_uuid                       14787 non-null  object
 5   source_center                   14787 non-null  object
 6   source_name                     14787 non-null  object
 7   source_city                     14787 non-null  category
 8   source_place                    14787 non-null  category
 9   source_state                    14787 non-null  category
 10  destination_center              14787 non-null  object
 11  destination_name                14787 non-null  object
 12  destination_city                14787 non-null  category
 13  destination_place               14787 non-null  category
 14  destination_state               14787 non-null  category
 15  start_scan_to_end_scan          14787 non-null  float64
 16  od_time_diff_hour               14787 non-null  float64
 17  actual_distance_to_destination  14787 non-null  float64
 18  actual_time                     14787 non-null  float64
 19  osrm_time                       14787 non-null  float64
 20  osrm_distance                   14787 non-null  float64
 21  cumsum_segment_actual_time      14787 non-null  float64
 22  cumsum_segment_osrm_time        14787 non-null  float64
 23  cumsum_segment_osrm_distance    14787 non-null  float64
dtypes: category(8), datetime64[ns](1), float64(9), object(6)
memory usage: 2.1+ MB
```

In [46]:
```
trip_level_data['data'].info()
```

```
<class 'pandas.core.series.Series'>
RangeIndex: 14787 entries, 0 to 14786
Series name: data
Non-Null Count  Dtype
--------------  -----
14787 non-null  category
dtypes: category(1)
memory usage: 14.7 KB
```

In [47]:
```
trip_level_data.describe(include= 'all').T
```

Out[47]:

	count	unique	top	freq	mean	min	25'
data	14787	2	training	10645	NaN	NaN	Na
trip_creation_time	14787	NaN	NaN	NaN	2018-09-22 12:26:28.269885696	2018-09-12 00:00:16.535741	2018-09-1 02:38:18.12843187
route_schedule_uuid	14787	1497	thanos::sroute:a16bfa03-3462-4bce-9c82-5784c7d...	53	NaN	NaN	Na
route_type	14787	2	Carting	8906	NaN	NaN	Na
trip_uuid	14787	14787	trip-153671041653548748	1	NaN	NaN	Na
source_center	14787	930	IND000000ACB	1052	NaN	NaN	Na
source_name	14787	930	Gurgaon_Bilaspur_HB (Haryana)	1052	NaN	NaN	Na
source_city	14787	714	Gurgaon	1128	NaN	NaN	Na
source_place	14787	710	Bilaspur	1074	NaN	NaN	Na
source_state	14787	29	Maharashtra	2714	NaN	NaN	Na
destination_center	14787	1035	IND000000ACB	821	NaN	NaN	Na
destination_name	14787	1035	Gurgaon_Bilaspur_HB (Haryana)	821	NaN	NaN	Na
destination_city	14787	840	Mumbai	1202	NaN	NaN	Na
destination_place	14787	803	Bilaspur	864	NaN	NaN	Na
destination_state	14787	31	Maharashtra	2561	NaN	NaN	Na
start_scan_to_end_scan	14787.0	NaN	NaN	NaN	529.429025	23.0	149.
od_time_diff_hour	14787.0	NaN	NaN	NaN	8.838559	0.391024	2.49497
actual_distance_to_destination	14787.0	NaN	NaN	NaN	164.090196	9.002461	22.77709
actual_time	14787.0	NaN	NaN	NaN	356.306012	9.0	67.
osrm_time	14787.0	NaN	NaN	NaN	36.193887	6.0	13.
osrm_distance	14787.0	NaN	NaN	NaN	40.698444	9.051	12.9405
cumsum_segment_actual_time	14787.0	NaN	NaN	NaN	353.059174	9.0	66.
cumsum_segment_osrm_time	14787.0	NaN	NaN	NaN	180.511598	6.0	30.
cumsum_segment_osrm_distance	14787.0	NaN	NaN	NaN	222.705466	9.0729	32.5788

Label Encoding

Label Encoding is a technique used to convert categorical values into numeric labels.

- Label encoding works well only when there are 2 groups in category columns like male-female, training-testing etc
- For other categories we can use TargetEncoding.

In [48]:
```
from sklearn.preprocessing import LabelEncoder

# get a deep copy to save the original
encoded_trip_level_data = trip_level_data.copy(deep= True)

# Select categorical columns
categorical_features = ['data', 'route_type']

# Initialize the encoder
label_encoder = LabelEncoder()

# Apply one-hot encoding
encoded_trip_level_data['data'] = label_encoder.fit_transform(encoded_trip_level_data['data'])

encoded_trip_level_data['route_type'] = label_encoder.fit_transform(encoded_trip_level_data['route_type'])

encoded_trip_level_data[['data', 'route_type']].value_counts()
```

Out[48]:
```
data  route_type
1     0             6243
      1             4402
0     0             2663
      1             1479
Name: count, dtype: int64
```

Standardization and Normalization

Standardization

Standardization is the process of transforming the features of your dataset so that they have a mean of 0 and a standard deviation of 1. This method is especially useful when working with machine learning models that assume the data follows a normal distribution.

In practice, standardization ensures that each feature contributes equally to the model. For example, when dealing with features like age and income, which may have different units and ranges, standardization helps bring them onto a comparable scale.

Use Case:
Standardization is ideal for models such as **linear regression**, **logistic regression**, and **SVMs** that rely on the assumption of normally distributed data.

Normalization

Normalization (or Min-Max Scaling) rescales the data to a fixed range, usually between 0 and 1. This method is useful when the dataset contains features with different scales, and you need all features to be on the same scale for models that are sensitive to the magnitude of the data.

Normalization is often applied when you want to preserve the relationships in the data, especially when working with algorithms like **k-NN** or **k-means clustering**, which use distance-based metrics.

Use Case:
Normalization is typically used when working with models like **neural networks**, where large differences in feature ranges can impact performance.

Normalization

Best for:

- **Numerical features with varying scales**: For example, features like **income**, **age**, **height**, and **weight** that have different units or ranges.
- **Features with non-normal distributions or bounded ranges**: Such as **percentages** or **ratings** (e.g., ratings from 1 to 5), where the values are constrained within a specific range.

Avoid for:

- **Categorical or ordinal features**: For example, features like **city names** or **product categories** where relationships are not numerical.
- **Features with outliers**: Outliers can distort the scaling, as normalization compresses all the values into a specific range, which

Standardization

Best for:

- **Numerical features with normal or nearly normal distributions**: For example, **test scores** or **sales amounts** where data is close to a normal distribution.
- **Features with varying units**: Such as **temperature in Celsius** or **weight in kilograms** that you want to scale to a common distribution, enabling better comparison between features.

Avoid for:

- **Categorical or binary features**: Features like **gender** or **country** that don't benefit from standardization as they are non-numeric.
- **Features with highly skewed distributions**: These may not perform well with standardization, as the process assumes a relatively symmetric distribution for optimal results.

Since Data distributions is not normal or Gaussian Distribution we can apply Normalisation for the numerical Columns

In [49]:
```
from sklearn.preprocessing import MinMaxScaler, StandardScaler

# Columns to normalize and standardize
numerical_columns = [
    'start_scan_to_end_scan', 'od_time_diff_hour', 'actual_distance_to_destination',
    'actual_time', 'osrm_time', 'osrm_distance',
    'cumsum_segment_actual_time', 'cumsum_segment_osrm_time', 'cumsum_segment_osrm_distance'
]

# Initialize scalers
min_max_scaler = MinMaxScaler()

# Apply normalization
normalized_data = min_max_scaler.fit_transform(encoded_trip_level_data[numerical_columns])

normalized_df = pd.DataFrame(normalized_data, columns=[col + '_normalized' for col in numerical_columns])

# Combine with the original data
encoded_trip_level_data = pd.concat([encoded_trip_level_data, normalized_df], axis=1)

# Display the first few rows of the transformed dataset
encoded_trip_level_data.head()
```

Out[49]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	source_ci
0	1	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	1	trip-153671041653548748	IND209304AAA	Kanpur_Central_H_6 (Uttar Pradesh)	Kanpı
1	1	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	0	trip-153671042288605164	IND561203AAB	Doddablpur_ChikaDPP_D (Karnataka)	Doddablpı
2	1	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	1	trip-153671043369099517	IND000000ACB	Gurgaon_Bilaspur_HB (Haryana)	Gurgac
3	1	2018-09-12 00:01:00.113710	thanos::sroute:f0176492-a679-4597-8332-bbd1c7f...	0	trip-153671046011330457	IND400072AAB	Mumbai Hub (Maharashtra)	Mumb
4	1	2018-09-12 00:02:09.740725	thanos::sroute:d9f07b12-65e0-4f3b-bec8-df06134...	1	trip-153671052974046625	IND583101AAA	Bellary_Dc (Karnataka)	Bellaı

5 rows × 33 columns

In [50]:
```
encoded_trip_level_data.shape
```

Out[50]: (14787, 33)

Doing Hypothesis testing/ visual analysis between actual_time aggregated value(cumsum_segment_actual_time) and osrm time aggregated value(cumsum_segment_osrm_time)

In [51]:
```
trip_level_data.head()
```

Out[51]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	source_
0	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	trip-153671041653548748	IND209304AAA	Kanpur_Central_H_6 (Uttar Pradesh)	Ka
1	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	trip-153671042288605164	IND561203AAB	Doddablpur_ChikaDPP_D (Karnataka)	Doddab
2	training	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	FTL	trip-153671043369099517	IND000000ACB	Gurgaon_Bilaspur_HB (Haryana)	Gurg
3	training	2018-09-12 00:01:00.113710	thanos::sroute:f0176492-a679-4597-8332-bbd1c7f...	Carting	trip-153671046011330457	IND400072AAB	Mumbai Hub (Maharashtra)	Mur
4	training	2018-09-12 00:02:09.740725	thanos::sroute:d9f07b12-65e0-4f3b-bec8-df06134...	FTL	trip-153671052974046625	IND583101AAA	Bellary_Dc (Karnataka)	Be

5 rows × 24 columns

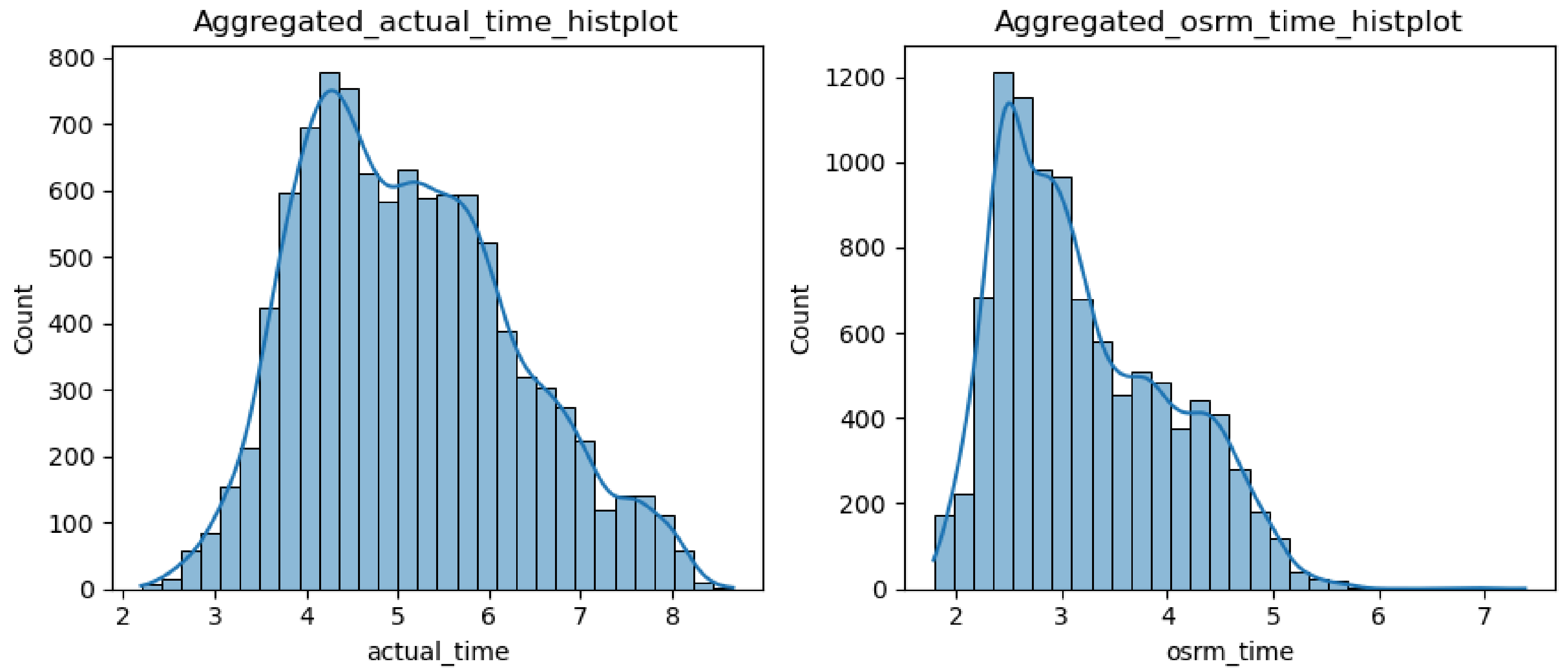
In [52]:
```
detect_outliers(trip_level_data['cumsum_segment_actual_time'],'cumsum_segment_actual_time')

detect_outliers(trip_level_data['cumsum_segment_osrm_time'],'cumsum_segment_osrm_time')
```

```
This cumsum_segment_actual_time have 1644 number of outliers in it which was 11.12% total data
This cumsum_segment_osrm_time have 1485 number of outliers in it which was 10.04% total data
```

In [53]:
```
trip_level_data.shape
```

Out[53]:
```
(14787, 24)
```

In [54]:
```
# There are quite a number of outliers so in order to do the tests we can apply log to the series.

aggregated_actual_time = np.log(trip_level_data['actual_time'].sample(10000))

aggregated_osrm_time = np.log(trip_level_data['osrm_time'].sample(10000))

plt.figure(figsize= (9,4))

plt.subplot(1,2,1)
sns.histplot(aggregated_actual_time, bins= 30, kde= True)
plt.title('Aggregated_actual_time_histplot')

plt.subplot(1,2,2)
sns.histplot(aggregated_osrm_time,bins= 30, kde= True)
plt.title('Aggregated_osrm_time_histplot')


plt.tight_layout()
plt.show()
```



In [55]:
```
detect_outliers(aggregated_actual_time, 'aggregated_actual_time')
detect_outliers(aggregated_osrm_time, 'aggregated_osrm_time')
```

```
This aggregated_actual_time have 2 number of outliers in it which was 0.02% total data
This aggregated_osrm_time have 10 number of outliers in it which was 0.1% total data
```

Let's perform Paired T-Test for the two columns

Paired T- Test: When you are comparing means of two related variableas and data is normally distributed.

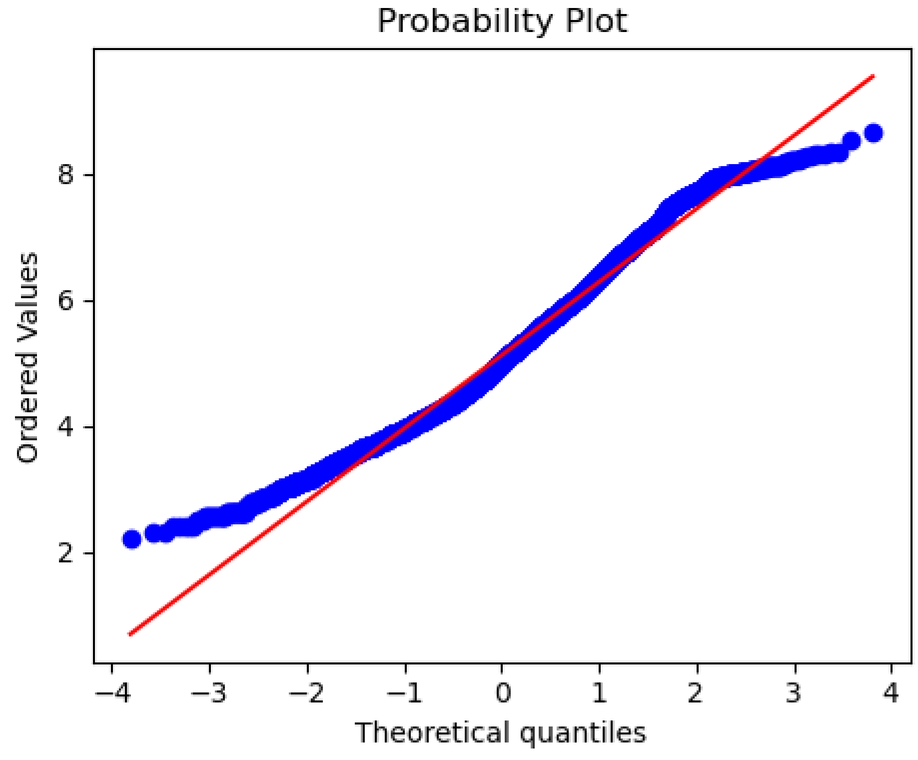
Why Paired T-Test? Let's Understand the assumptions:

- The data should normally distrubuted for both the series
- The samples should be paired
- The data should be continuous In which the two series satisfies all the conditions we use paired T-Test

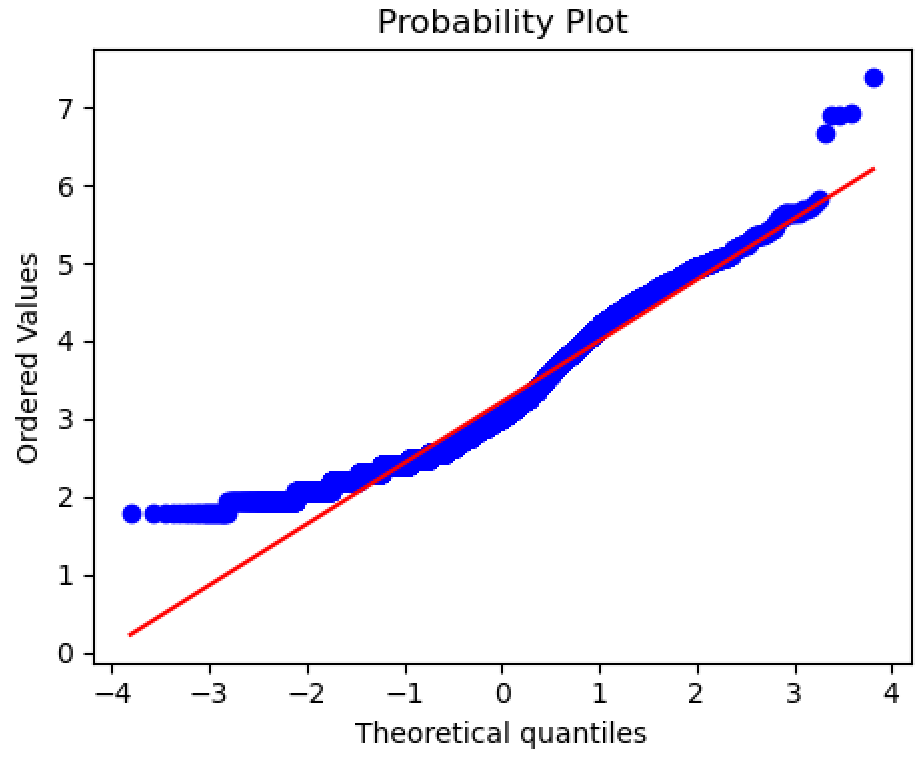
---

Let's Check whether the data satisfies or not using few tests like Shapiro-Wilk test for distribution and levene's test for variance

In [56]:
```
# Before check the distribution of data using QQ plot
plt.figure(figsize= (9,4))

plt.subplot(1,2,1)
stats.probplot(aggregated_actual_time, dist= 'norm', plot= plt)

plt.tight_layout()
plt.show()
```



In [57]:
```
# Before check the distribution of data using QQ plot
plt.figure(figsize= (9,4))

plt.subplot(1,2,1)
stats.probplot(aggregated_osrm_time, dist= 'norm', plot= plt)

plt.tight_layout()
plt.show()
```



Shapiro-Wilk Test:

Null Hypothesis(Ho): The data is not normally distributed

Alternate Hypothesis(Ha): The data is normally distributed

In [58]:
```
# perform shapiro's test to verify normal distribution

stat1, p_value1 = stats.shapiro(aggregated_actual_time)
stat2, p_value2 = stats.shapiro(aggregated_osrm_time)

# Print the results
print(f"Shapiro-Wilk test for aggregated_actual_time: Statistic = {stat1}, P-value = {p_value1}")
print(f"Shapiro-Wilk test for  aggregated_osrm_time: Statistic = {stat2}, P-value = {p_value2}")

# Interpret the results for each dataset
def interpret_shapiro(p_value):
    if p_value <= 0.05:
        return "Data is not normally distributed."
    else:
        return "Data is normally distributed."

print("\nInterpretation of results:")
print(f"aggregated_actual_time: {interpret_shapiro(p_value1)}")
print(f"aggregated_osrm_time: {interpret_shapiro(p_value2)}")
```

```
Shapiro-Wilk test for aggregated_actual_time: Statistic = 0.9800751805305481, P-value = 4.152019099617392e-3
5
Shapiro-Wilk test for  aggregated_osrm_time: Statistic = 0.9474515318870544, P-value = 0.0

Interpretation of results:
aggregated_actual_time: Data is not normally distributed.
aggregated_osrm_time: Data is not normally distributed.
```

Key Observation:

- Since the samples do not follow any of the assumptions T-Test cannot be applied here, we can perform its non parametric equivalent test i.e., wilcoxon OR Mann-Whitney U rank test for two independent samples.

In [59]:
```
trip_level_data.nunique()
```

Out[59]:
```
data                                   2
trip_creation_time                 14787
route_schedule_uuid                 1497
route_type                             2
trip_uuid                          14787
source_center                        930
source_name                          930
source_city                          714
source_place                         710
source_state                          29
destination_center                  1035
destination_name                    1035
destination_city                     840
destination_place                    803
destination_state                     31
start_scan_to_end_scan              2203
od_time_diff_hour                  14787
actual_distance_to_destination     14771
actual_time                         1850
osrm_time                            249
osrm_distance                      14413
cumsum_segment_actual_time          1885
cumsum_segment_osrm_time            1240
cumsum_segment_osrm_distance       14724
dtype: int64
```

Wilcoxon Signed-Rank Test:

- **The Wilcoxon test is a non-parametric statistical test used in hypothesis testing to compare two related samples or matched pairs when data is not normally distributed.**
- Used to test whether there is a median difference between two related or paired samples

---

Hypothesis Testing

- **Null Hypothesis (H_0)**:
  There is no significant difference between two variables.
- **Alternative Hypothesis (H_1)**:
  There is a significant difference between two variables.

In [60]:
```
# Perform the Wilcoxon signed-rank test
stat, p_value = stats.wilcoxon(aggregated_actual_time, aggregated_osrm_time)

print(f"Wilcoxon statistic: {stat}, p-value: {p_value}")

# Interpret the results
def interpret_wilcoxon(p_value):
    if p_value <= 0.05:
        return "There is a significant difference between the distributions of the two variables."
    else:
        return "There is no significant difference between the distributions of the two variables."

# Interpretation
print(interpret_wilcoxon(p_value))
```

```
Wilcoxon statistic: 1293381.0, p-value: 0.0
There is a significant difference between the distributions of the two variables.
```

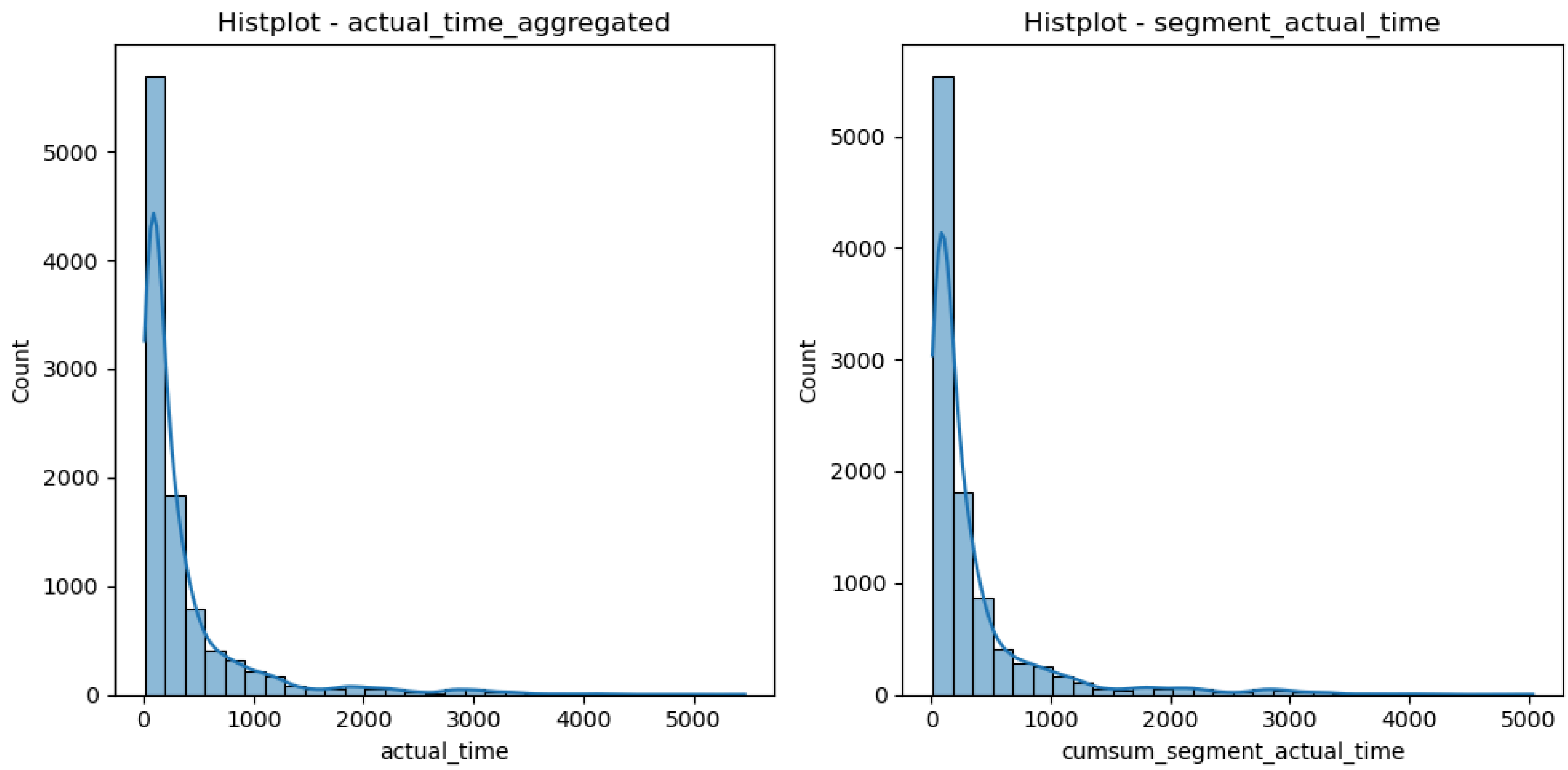
Hypothesis testing between actual_time aggregated value and segment actual time aggregated value to know thier relationship between them. As they are dependent variables we can perform Paired T-Test if all assumptions are satisfied.

In [61]:
```
trip_level_data.columns
```

Out[61]:
```
Index(['data', 'trip_creation_time', 'route_schedule_uuid', 'route_type',
       'trip_uuid', 'source_center', 'source_name', 'source_city',
       'source_place', 'source_state', 'destination_center',
       'destination_name', 'destination_city', 'destination_place',
       'destination_state', 'start_scan_to_end_scan', 'od_time_diff_hour',
       'actual_distance_to_destination', 'actual_time', 'osrm_time',
       'osrm_distance', 'cumsum_segment_actual_time',
       'cumsum_segment_osrm_time', 'cumsum_segment_osrm_distance'],
      dtype='object')
```

In [62]:
```
actual_time_aggregated = trip_level_data['actual_time'].sample(10000)
segment_actual_time = trip_level_data['cumsum_segment_actual_time'].sample(10000)
```

In [63]:
```
# Let's have some visual analysis to know how the distribution looks like

plt.figure(figsize= (10,5))

plt.subplot(1,2,1)
sns.histplot(actual_time_aggregated, kde= True,bins = 30 )
plt.title('Histplot - actual_time_aggregated')

plt.subplot(1,2,2)
sns.histplot(segment_actual_time, kde= True,bins = 30 )
plt.title('Histplot - segment_actual_time')

plt.tight_layout()
plt.show()
```



In [64]:
```
# detect outliers

detect_outliers(actual_time_aggregated,'actual_time_aggregated')
detect_outliers(segment_actual_time,'segment_actual_time')
```

```
This actual_time_aggregated have 1116 number of outliers in it which was 11.16% total data
This segment_actual_time have 1139 number of outliers in it which was 11.39% total data
```

The data is extremely left skewed and contains many outliers in it. So we can apply boxcox transformation or log and try to make the data normal.

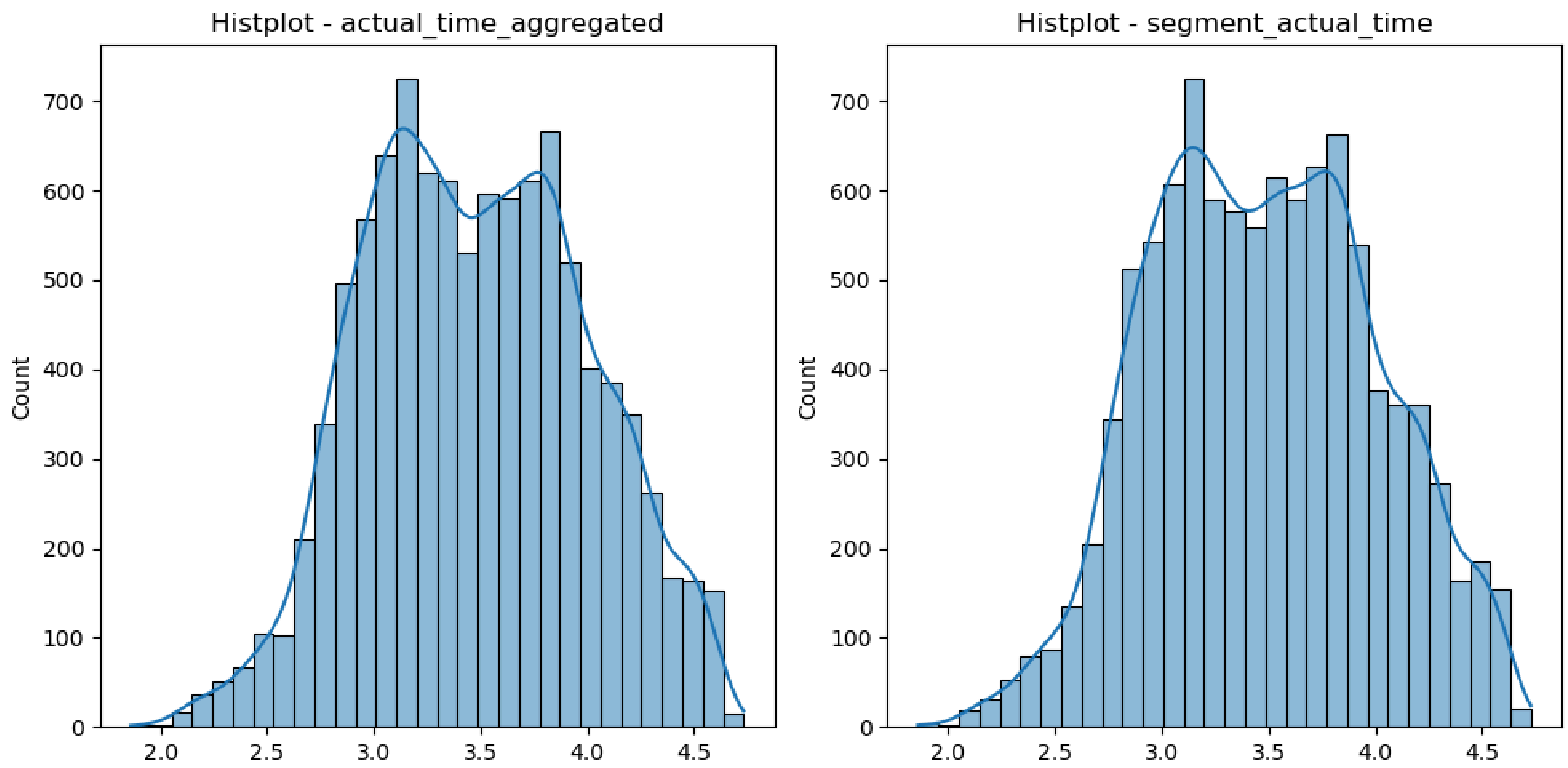
Box-Cox Transformation

The **Box-Cox transformation** is used to stabilize variance and make data more normal-like.

Assumptions:

1. The data must be **positive** (no zero or negative values).
2. It assumes the data can be transformed to approximate **normality**.

In [65]:
```
#apply box-cox transformation

transformed_actual_time, lambda_value1 = stats.boxcox(actual_time_aggregated)
transformed_segment_actual_time, lambda_value2 = stats.boxcox(segment_actual_time)

# plotting the data after applying boxcox transformation
plt.figure(figsize= (10,5))

plt.subplot(1,2,1)
sns.histplot(transformed_actual_time, kde= True,bins = 30 )
plt.title('Histplot - actual_time_aggregated')

plt.subplot(1,2,2)
sns.histplot(transformed_segment_actual_time, kde= True,bins = 30 )
plt.title('Histplot - segment_actual_time')

plt.tight_layout()
plt.show()
```

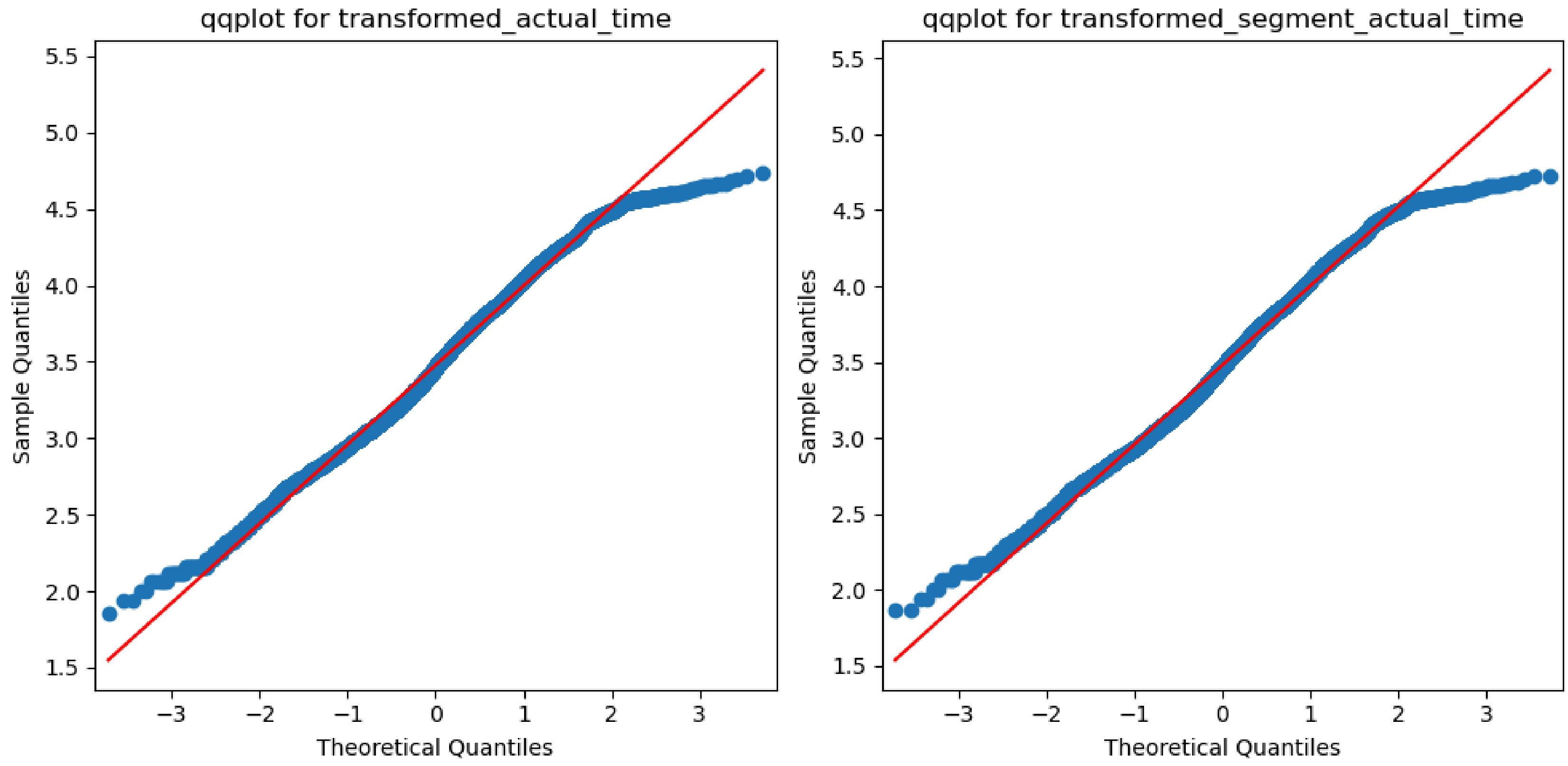


In [66]:
```
# data seems to be normal but let's perform few tests to check normality of data

detect_outliers(transformed_actual_time,'transformed_actual_time')
detect_outliers(transformed_segment_actual_time,'transformed_segment_actual_time')
```

This **transformed_actual_time** have 1 number of outliers in it which was 0.01% total data
This **transformed_segment_actual_time** have 2 number of outliers in it which was 0.02% total data

Let's check normality of data using qqplot an shapiro's test. Later for variance we can perform levene's test.

In [67]:
```
from statsmodels.graphics.gofplots import qqplot

plt.figure(figsize= (10,5))

plt.subplot(1,2,1)
qqplot(transformed_actual_time, line= 's',ax=plt.gca())
plt.title('qqplot for transformed_actual_time')

plt.subplot(1,2,2)
qqplot(transformed_segment_actual_time, line= 's',ax=plt.gca())
plt.title("qqplot for transformed_segment_actual_time")

plt.tight_layout()
plt.show()
```



From Q-Q plot the data seems to be approximate normal. To verify wheater data is normally distributed or not we can perform shapiro's normality test.

Shapiro-Wilk Test: Brief Overview

The **Shapiro-Wilk test** is a statistical test used to determine whether a dataset is normally distributed.
It is widely used because it is highly effective for small sample sizes.

Hypotheses in Shapiro-Wilk Test

- **Null Hypothesis (H_0)**:
  The data follows a normal distribution.
- **Alternative Hypothesis (H_1)**:
  The data does not follow a normal distribution.

In [68]:
```
# Perform Shapiro-Wilk test for aggregated_actual_time
stat, p_value = stats.shapiro(transformed_actual_time)

print(f"Shapiro-Wilk Statistic: {stat}, p-value: {p_value}")

# Interpret results
if p_value > 0.05:
    print("Data is likely normally distributed.")
else:
    print("Data is not normally distributed.")
```

```
Shapiro-Wilk Statistic: 0.9912139177322388, p-value: 2.369121467271263e-24
Data is not normally distributed.
```

In [69]:
```
# Perform Shapiro-Wilk test for transformed_segemt_actual_time
stat, p_value = stats.shapiro(transformed_segment_actual_time)

print(f"Shapiro-Wilk Statistic: {stat}, p-value: {p_value}")

# Interpret results
if p_value > 0.05:
    print("Data is likely normally distributed.")
else:
    print("Data is not normally distributed.")
```

```
Shapiro-Wilk Statistic: 0.9915910363197327, p-value: 7.807164685772385e-24
Data is not normally distributed.
```

In both the cases we can conclude that the data seems to be normally distributed but it is not. In this case we cannot perform paired T-test. although we can perform other non-parametric tests like wilcoxon, mann

But before that let's check variance by performing levene's test.

Levene's Test: Brief Overview

The **Levene's test** is a statistical test used to assess the equality of variances (**homoscedasticity**) across different groups.

Hypotheses in Levene's Test

- **Null Hypothesis (H_0)**:
  The variances across the groups are equal (**homoscedasticity**).
- **Alternative Hypothesis (H_1)**:
  At least one group's variance is different (**heteroscedasticity**).

In [70]:
```
# Perform levene's test

l_stat, p_value = stats.levene(transformed_actual_time,transformed_segment_actual_time)

print(f"Levene Statistic: {l_stat}, p-value: {p_value}")

# Interpret results
if p_value > 0.05:
    print("Variances are likely equal across groups.")
else:
    print("Variances are significantly different across groups.")
```

```
Levene Statistic: 0.2344187372223143, p-value: 0.6282718849174118
Variances are likely equal across groups.
```

Key Observation:

- Since the samples do not follow few assumptions T-Test cannot be applied here, we can perform its non parametric equivalent tests like Mann-Whitney U rank test for two independent samples OR Wilcoxon test for two related samples.

Wilcoxon Signed-Rank Test: Brief Overview

The **Wilcoxon Signed-Rank Test** is a non-parametric test used to compare two related samples, matched samples, or repeated measurements on the same individuals.
It is used when the assumptions of the paired t-test (such as normality) are violated.

Hypotheses in Wilcoxon Signed-Rank Test

- **Null Hypothesis (H_0)**:
  The median difference between the paired observations is zero (no significant difference).
- **Alternative Hypothesis (H_1)**:
  The median difference between the paired observations is not zero (a significant difference exists).

In [71]:
```
# Perform Wilcoxon Signed-Rank Test

k_stat, p_value = stats.wilcoxon(transformed_actual_time, transformed_segment_actual_time)

print(f"Wilcoxon Statistic: {k_stat}, p-value: {p_value}")

# Interpret the results
if p_value > 0.05:
    print("No significant difference between the paired groups.")
else:
    print("Significant difference between the paired groups.")
```

```
Wilcoxon Statistic: 24923522.0, p-value: 0.7844171664596464
No significant difference between the paired groups.
```

From above test we can conclude that there is no relation between actual time and segment actual time. As they are having significant difference between them.

Hypothesis testing between OSRM distance aggregated value and segment OSRM distance aggregated value.

In [72]:
```
trip_level_data.head()
```

Out[72]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	source_
0	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	trip-153671041653548748	IND209304AAA	Kanpur_Central_H_6 (Uttar Pradesh)	Ka
1	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	trip-153671042288605164	IND561203AAB	Doddablpur_ChikaDPP_D (Karnataka)	Doddat
2	training	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	FTL	trip-153671043369099517	IND000000ACB	Gurgaon_Bilaspur_HB (Haryana)	Gurç
3	training	2018-09-12 00:01:00.113710	thanos::sroute:f0176492-a679-4597-8332-bbd1c7f...	Carting	trip-153671046011330457	IND400072AAB	Mumbai Hub (Maharashtra)	Mur
4	training	2018-09-12 00:02:09.740725	thanos::sroute:d9f07b12-65e0-4f3b-bec8-df06134...	FTL	trip-153671052974046625	IND583101AAA	Bellary_Dc (Karnataka)	Be

5 rows × 24 columns

In [73]:
```
trip_level_data[['osrm_distance','cumsum_segment_osrm_distance']].describe().T
```

Out[73]:

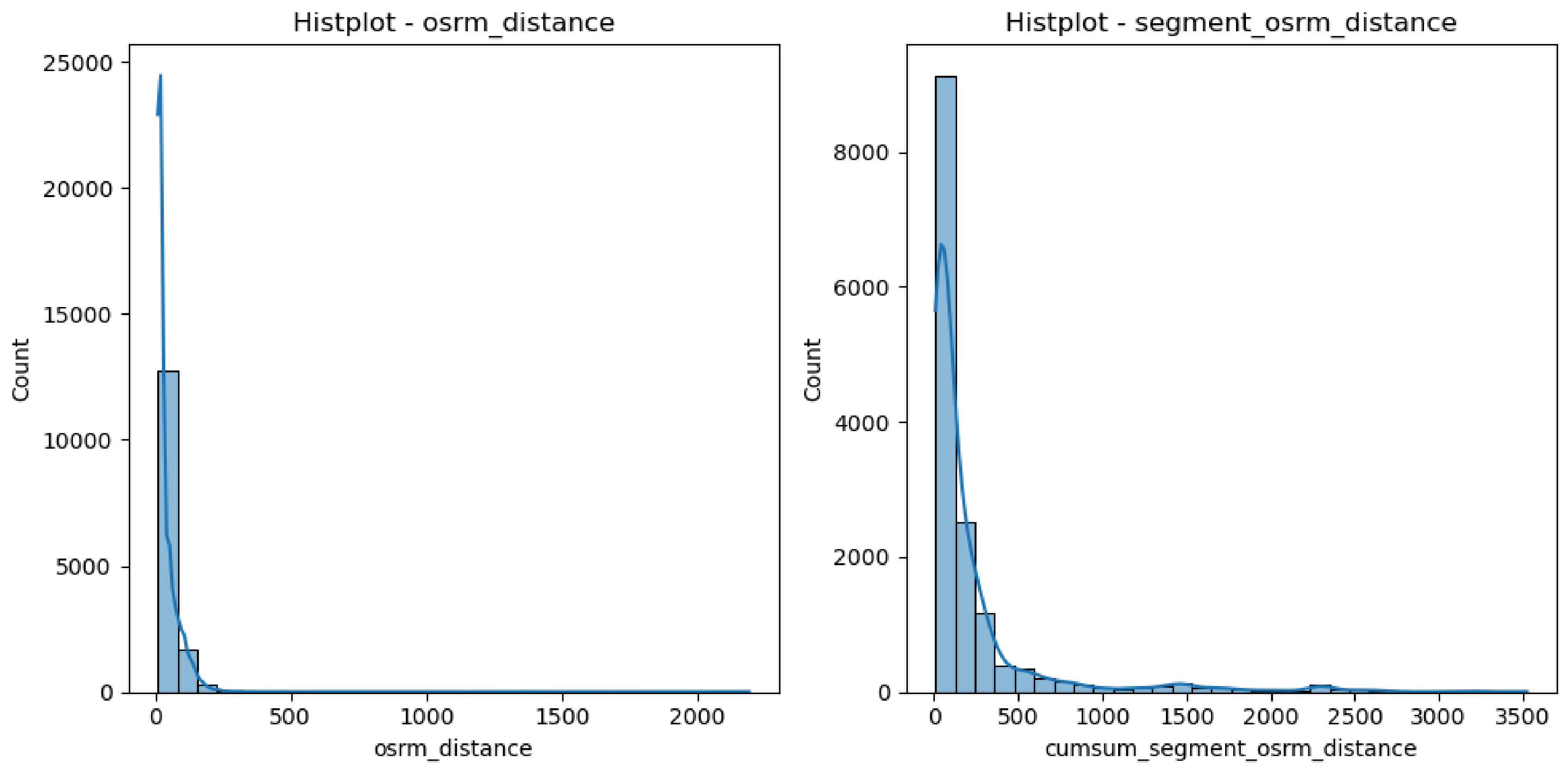
	count	mean	std	min	25%	50%	75%	max
osrm_distance	14787.0	40.698444	49.328946	9.0510	12.94055	24.4979	53.29315	2191.4037
cumsum_segment_osrm_distance	14787.0	222.705466	416.846279	9.0729	32.57885	69.7842	216.56060	3523.6324

In [74]:
```
osrm_distance = trip_level_data['osrm_distance']
segment_osrm_distance = trip_level_data['cumsum_segment_osrm_distance']
```

In [75]:
```
# detect outliers
detect_outliers(osrm_distance, 'osrm_distance')
detect_outliers(segment_osrm_distance, 'segment_osrm_distance')
```

```
This osrm_distance have 995 number of outliers in it which was 6.73% total data
This segment_osrm_distance have 1550 number of outliers in it which was 10.48% total data
```

In [76]:
```
# Let's have some visual analysis to know how the data distribution looks like

plt.figure(figsize= (10,5))

plt.subplot(1,2,1)
sns.histplot(osrm_distance, kde= True,bins = 30 )
plt.title('Histplot - osrm_distance')

plt.subplot(1,2,2)
sns.histplot(segment_osrm_distance, kde= True,bins = 30 )
plt.title('Histplot - segment_osrm_distance')

plt.tight_layout()
plt.show()
```



The data is extremely left skewed and contains many outliers in it. So we can apply boxcox transformation or log and try to make the data normal.

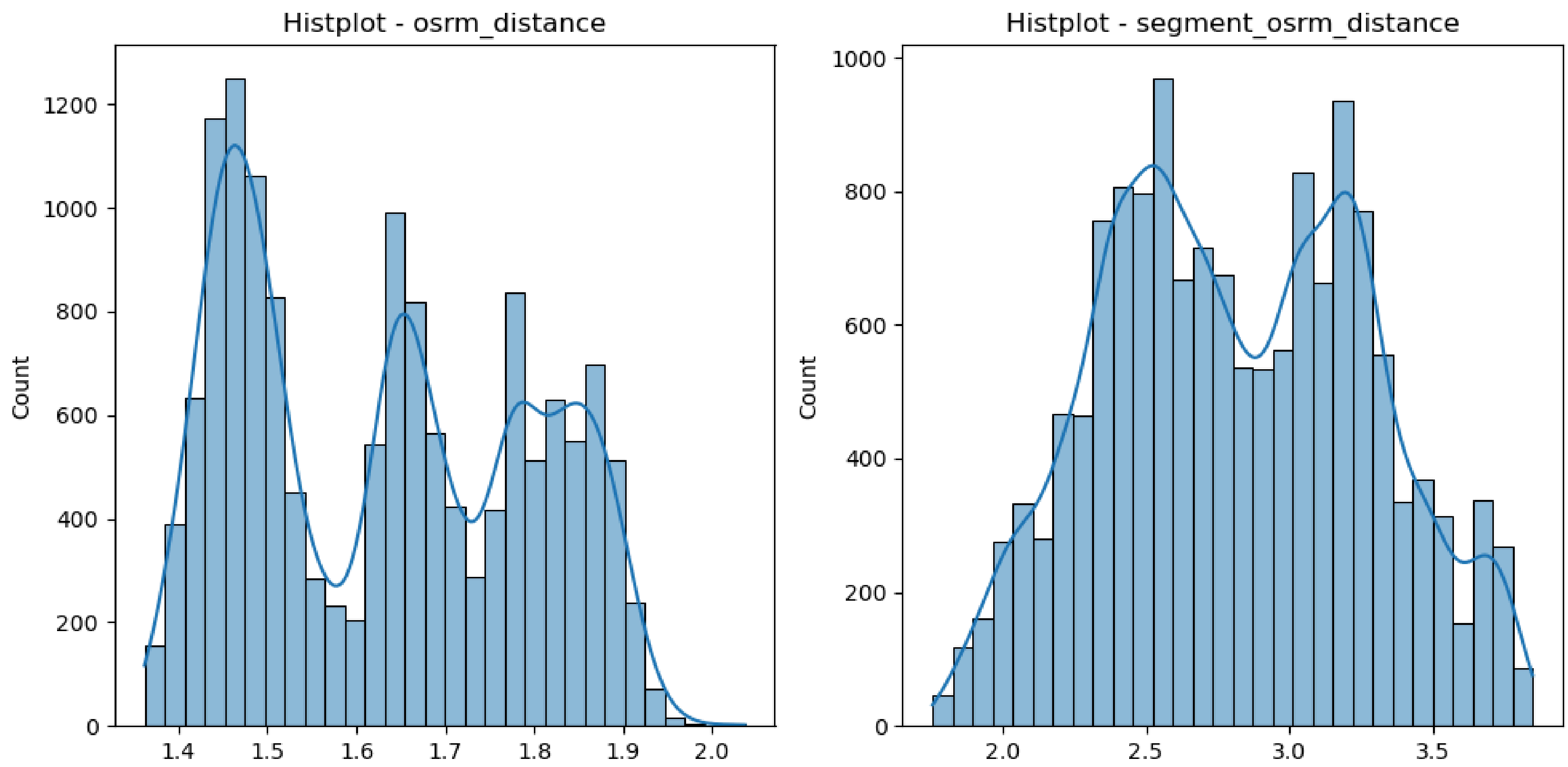
Box-Cox Transformation

The **Box-Cox transformation** is used to stabilize variance and make data more normal-like.

Assumptions:

1. The data must be **positive** (no zero or negative values).
2. It assumes the data can be transformed to approximate **normality**.

In [77]:
```
#apply box-cox transformation

transformed_osrm_distance, lambda_value1 = stats.boxcox(osrm_distance)
transformed_segment_osrm_distance, lambda_value2 = stats.boxcox(segment_osrm_distance)

# plotting the data after applying boxcox transformation
plt.figure(figsize= (10,5))

plt.subplot(1,2,1)
sns.histplot(transformed_osrm_distance, kde= True,bins = 30 )
plt.title('Histplot - osrm_distance')

plt.subplot(1,2,2)
sns.histplot(transformed_segment_osrm_distance, kde= True,bins = 30 )
plt.title('Histplot - segment_osrm_distance')

plt.tight_layout()
plt.show()
```

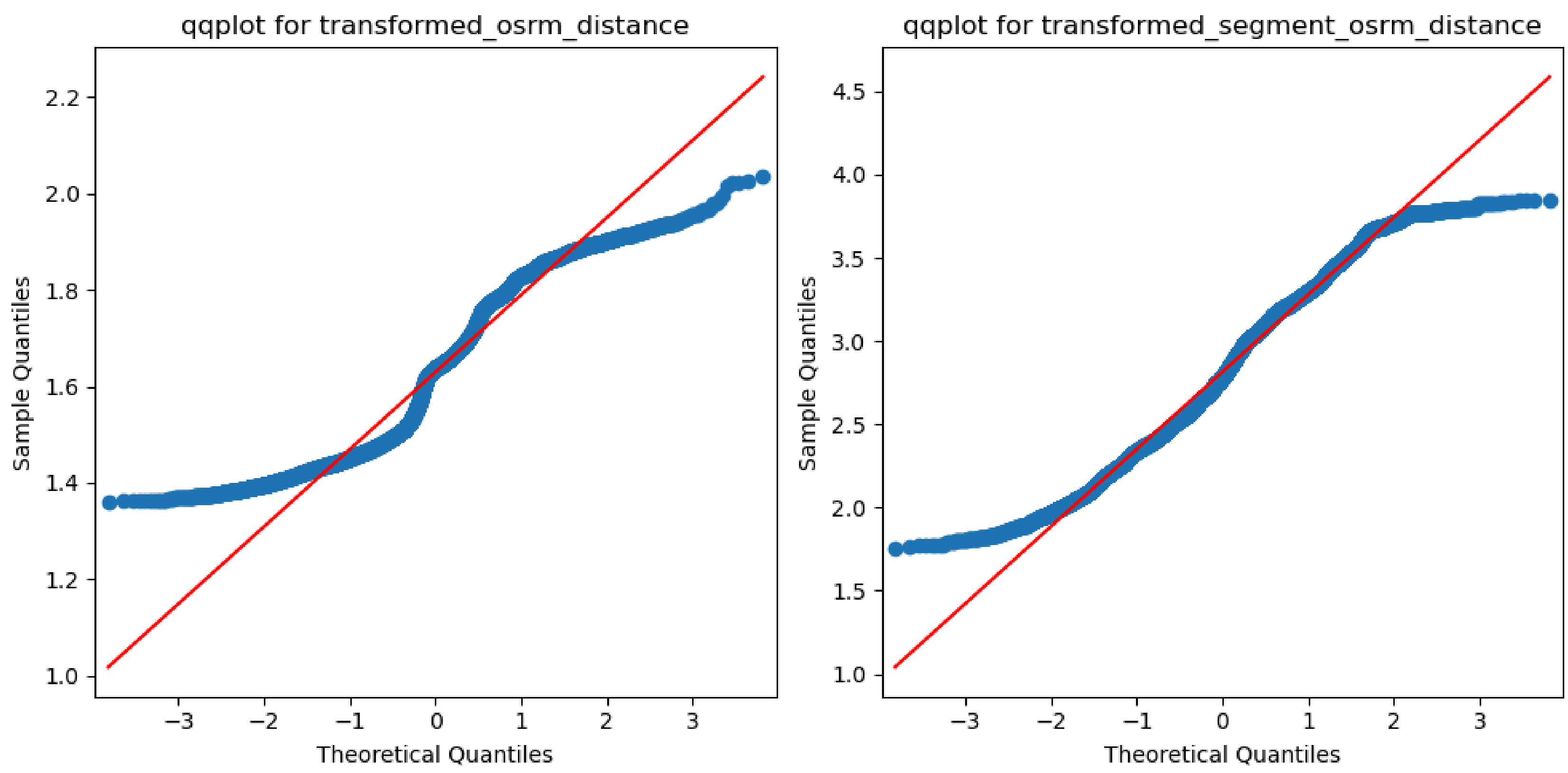


In [78]:
```
# detect outliers after transformation
detect_outliers(transformed_osrm_distance, 'osrm_distance')
detect_outliers(transformed_segment_osrm_distance, 'segment_osrm_distance')
```

This **osrm_distance** have 0 number of outliers in it which was 0.0% total data
This **segment_osrm_distance** have 0 number of outliers in it which was 0.0% total data

Let's check normality of data using qqplot an shapiro's test. Later for variance we can perform levene's test.

In [79]:
```
from statsmodels.graphics.gofplots import qqplot

plt.figure(figsize= (10,5))

plt.subplot(1,2,1)
qqplot(transformed_osrm_distance, line= 's',ax=plt.gca())
plt.title('qqplot for transformed_osrm_distance')

plt.subplot(1,2,2)
qqplot(transformed_segment_osrm_distance, line= 's',ax=plt.gca())
plt.title("qqplot for transformed_segment_osrm_distance")

plt.tight_layout()
plt.show()
```



To verify wheater data is normally distributed or not we can perform shapiro's normality test.

Shapiro-Wilk Test: Brief Overview

The **Shapiro-Wilk test** is a statistical test used to determine whether a dataset is normally distributed.
It is widely used because it is highly effective for small sample sizes.

Hypotheses in Shapiro-Wilk Test

- **Null Hypothesis (H_0)**:
  The data follows a normal distribution.
- **Alternative Hypothesis (H_1)**:
  The data does not follow a normal distribution.

In [80]:
```
# Perform Shapiro-Wilk test for transformed_osrm_distance
stat, p_value = stats.shapiro(transformed_osrm_distance)

print(f"Shapiro-Wilk Statistic: {stat}, p-value: {p_value}")

# Interpret results
if p_value > 0.05:
    print("Data is likely normally distributed.")
else:
    print("Data is not normally distributed.")
```

```
Shapiro-Wilk Statistic: 0.9323399662971497, p-value: 0.0
Data is not normally distributed.
```

In [81]:
```
# Perform Shapiro-Wilk test for transformed_segment_osrm_distance
stat, p_value = stats.shapiro(transformed_segment_osrm_distance)

print(f"Shapiro-Wilk Statistic: {stat}, p-value: {p_value}")

# Interpret results
if p_value > 0.05:
    print("Data is likely normally distributed.")
else:
    print("Data is not normally distributed.")
```

```
Shapiro-Wilk Statistic: 0.9833676815032959, p-value: 4.563222510909864e-38
Data is not normally distributed.
```

Key Observation:

- Since the samples do not follow any assumptions of T-Test, T-test cannot be applied here, we can perform its non parametric equivalent tests like Mann-Whitney U rank test for two independent samples OR Wilcoxon test for two related samples.

Wilcoxon Signed-Rank Test: Brief Overview

The **Wilcoxon Signed-Rank Test** is a non-parametric test used to compare two related samples, matched samples, or repeated measurements on the same individuals.
It is used when the assumptions of the paired t-test (such as normality) are violated.

Hypotheses in Wilcoxon Signed-Rank Test

- **Null Hypothesis (H_0)**:
  The median difference between the paired observations is zero (no significant difference).
- **Alternative Hypothesis (H_1)**:
  The median difference between the paired observations is not zero (a significant difference exists).

In [82]:
```
# Perform Wilcoxon Signed-Rank Test

k_stat, p_value = stats.wilcoxon(transformed_osrm_distance, transformed_segment_osrm_distance)

print(f"Wilcoxon Statistic: {k_stat}, p-value: {p_value}")

# Interpret the results
if p_value > 0.05:
    print("No significant difference between the paired groups.")
else:
    print("Significant difference between the paired groups.")
```

```
Wilcoxon Statistic: 0.0, p-value: 0.0
Significant difference between the paired groups.
```

Doing Hypothesis testing/ visual analysis between OSRM time aggregated value and segment OSRM time aggregated value.

In [83]:
```
trip_level_data.head()
```

Out[83]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	source_
0	training	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	trip-153671041653548748	IND209304AAA	Kanpur_Central_H_6 (Uttar Pradesh)	Ka
1	training	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	trip-153671042288605164	IND561203AAB	Doddablpur_ChikaDPP_D (Karnataka)	Doddak
2	training	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	FTL	trip-153671043369099517	IND000000ACB	Gurgaon_Bilaspur_HB (Haryana)	Gurg
3	training	2018-09-12 00:01:00.113710	thanos::sroute:f0176492-a679-4597-8332-bbd1c7f...	Carting	trip-153671046011330457	IND400072AAB	Mumbai Hub (Maharashtra)	Mur
4	training	2018-09-12 00:02:09.740725	thanos::sroute:d9f07b12-65e0-4f3b-bec8-df06134...	FTL	trip-153671052974046625	IND583101AAA	Bellary_Dc (Karnataka)	Be

5 rows × 24 columns

In [84]:
```
trip_level_data[['osrm_time','cumsum_segment_osrm_time']].describe().T
```

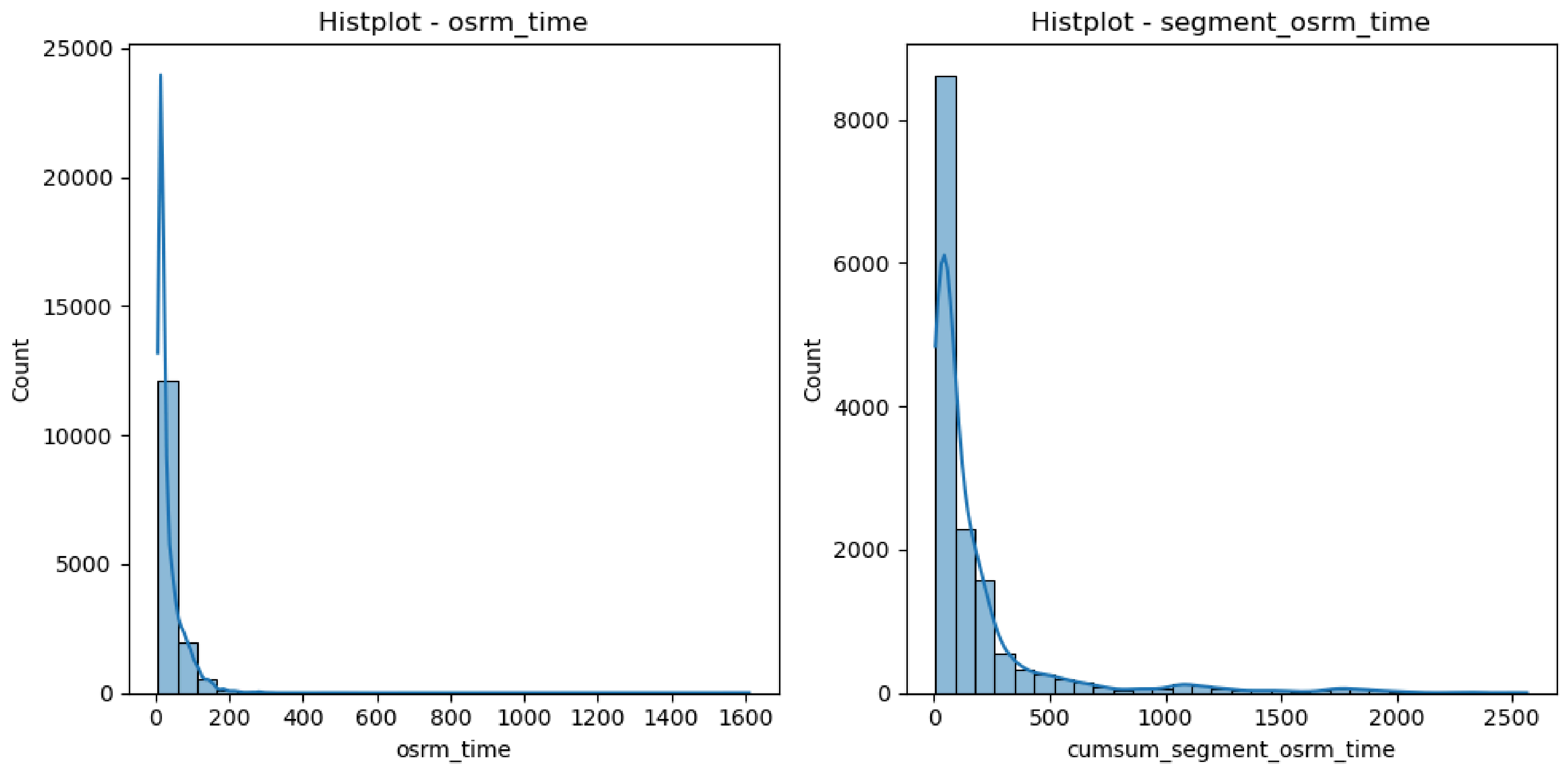
Out[84]:

	count	mean	std	min	25%	50%	75%	max
osrm_time	14787.0	36.193887	41.555735	6.0	13.0	21.0	45.0	1611.0
cumsum_segment_osrm_time	14787.0	180.511598	314.679279	6.0	30.0	65.0	184.0	2564.0

In [85]:
```
osrm_time = trip_level_data['osrm_time']
segment_osrm_time = trip_level_data['cumsum_segment_osrm_time']

# detect outliers
detect_outliers(osrm_distance, 'osrm_distance')
detect_outliers(segment_osrm_distance, 'segment_osrm_distance')
```

This **osrm_distance** have 995 number of outliers in it which was 6.73% total data
This **segment_osrm_distance** have 1550 number of outliers in it which was 10.48% total data

In [86]:
```
# Let's have some visual analysis to know how the data distribution looks like

plt.figure(figsize= (10,5))

plt.subplot(1,2,1)
sns.histplot(osrm_time, kde= True,bins = 30 )
plt.title('Histplot - osrm_time')

plt.subplot(1,2,2)
sns.histplot(segment_osrm_time, kde= True,bins = 30 )
plt.title('Histplot - segment_osrm_time')

plt.tight_layout()
plt.show()
```



The data is extremely left skewed and contains many outliers in it. So we can apply boxcox transformation or log and try to make the data normal.

Applying Log Transformation

1. Skewed Distributions:

- Log transformations are effective when the data is **right-skewed**, where a few high values distort the distribution (e.g., income, sales data).

2. Reducing Variability:

- When data varies significantly (e.g., wide range of values), applying a log transformation can help **stabilize the variance**, making patterns more visible.

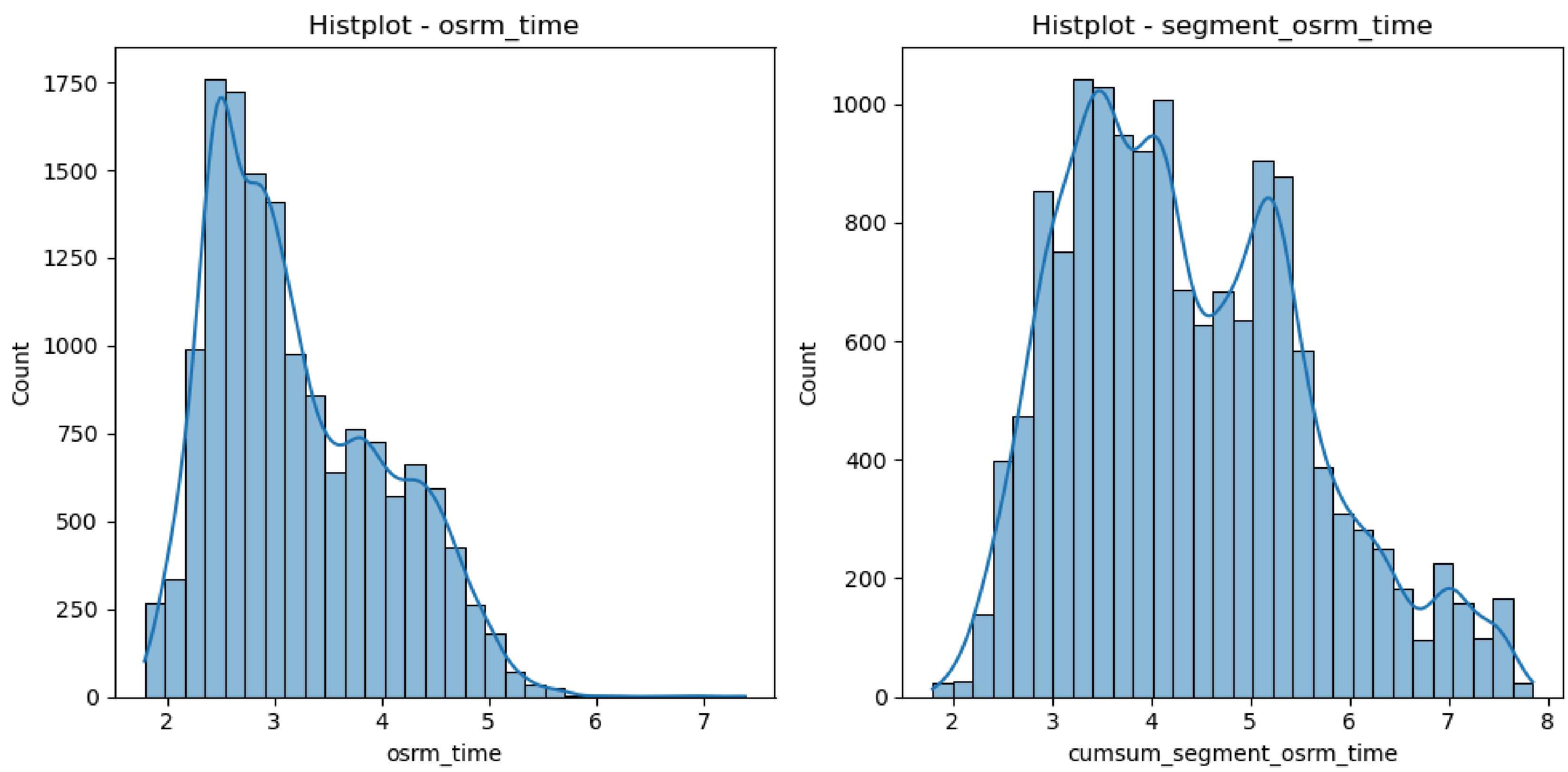
3. Dealing with Outliers:

- Large outliers can be **compressed** into a more manageable range after a log transformation.

How Log Transformation Works:

- The **log transformation** reduces large values and expands smaller ones, bringing the distribution closer to normality, which is often an assumption in statistical models.

In [87]:
```
osrm_time = np.log(osrm_time)
segment_osrm_time = np.log(segment_osrm_time)

# Let's have some visual analysis to know how the data distribution looks like

plt.figure(figsize= (10,5))

plt.subplot(1,2,1)
sns.histplot(osrm_time, kde= True,bins = 30 )
plt.title('Histplot - osrm_time')

plt.subplot(1,2,2)
sns.histplot(segment_osrm_time, kde= True,bins = 30 )
plt.title('Histplot - segment_osrm_time')

plt.tight_layout()
plt.show()
```



In [88]:
```
# detect outliers after transformation
detect_outliers(osrm_time, 'osrm_time')
detect_outliers(segment_osrm_time, 'segment_osrm_time')
```

```
This osrm_time have 14 number of outliers in it which was 0.09% total data
This segment_osrm_time have 0 number of outliers in it which was 0.0% total data
```

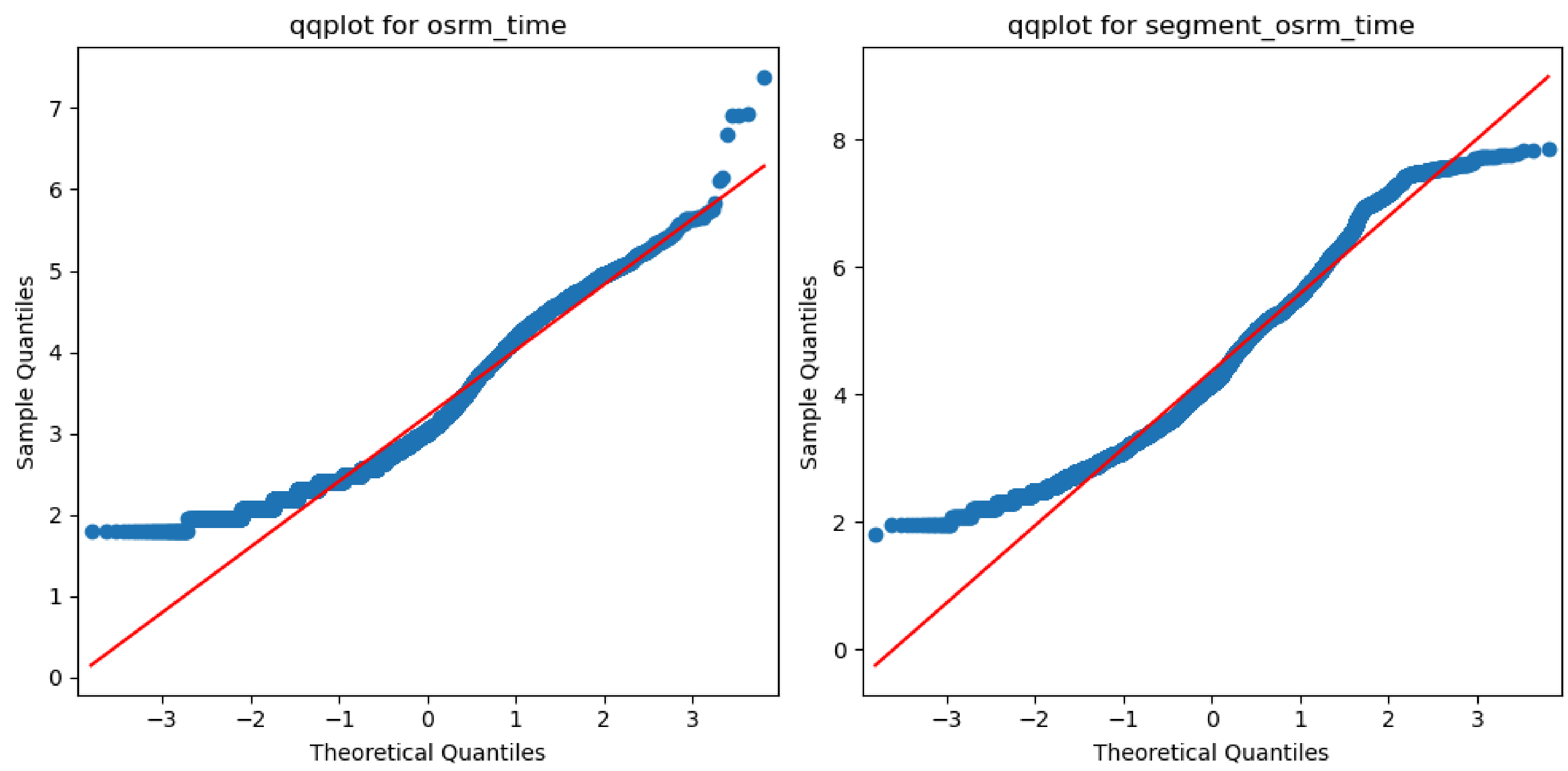
In [89]:
```
# remove outlier from the series

def remove_outliers(value, col_name= "column"):
    value= pd.Series(value)

    q1 = value.quantile(0.25)
    q3 = value.quantile(0.75)
    IQR = q3-q1

    lower_bound = q1 - 1.5*IQR
    upper_bound = q3 + 1.5*IQR
    rows = len(value)
    before_outliers = len(value[(value < lower_bound) | (value > upper_bound)])

#     print(f"This \033[1m{col_name}\033[0m have {outliers} number of outliers in it which was {np.round((out

    value = value[(value >= lower_bound) & (value <= upper_bound)]
    after_outliers = len(value[(value < lower_bound) | (value > upper_bound)])

#     print(f"This \033[1m{col_name}\033[0m have {before_outliers} number of outliers before transformation a


    return f"This {col_name} have {before_outliers} number of outliers before transformation and {after_outli

remove_outliers(osrm_time, 'osrm_time')
```

Out[89]:
```
'This osrm_time have 14 number of outliers before transformation and 0 after transformation and remaining ro
ws are 14773 of 14787 total number of rows'
```

In [90]:
```
osrm_time = osrm_time.sample(14000)
segment_osrm_time = segment_osrm_time.sample(14000)
```

Let's check normality of data using qqplot an shapiro's test. Later for variance we can perform levene's test.

In [91]:
```
from statsmodels.graphics.gofplots import qqplot

plt.figure(figsize= (10,5))

plt.subplot(1,2,1)
qqplot(osrm_time, line= 's',ax=plt.gca())
plt.title('qqplot for osrm_time')

plt.subplot(1,2,2)
qqplot(segment_osrm_time, line= 's',ax=plt.gca())
plt.title("qqplot for segment_osrm_time")

plt.tight_layout()
plt.show()
```



To verify wheater data is normally distributed or not we can perform shapiro's normality test.

Shapiro-Wilk Test: Brief Overview

The **Shapiro-Wilk test** is a statistical test used to determine whether a dataset is normally distributed.
It is widely used because it is highly effective for small sample sizes.

Hypotheses in Shapiro-Wilk Test

- **Null Hypothesis (H_0)**:
  The data follows a normal distribution.
- **Alternative Hypothesis (H_1)**:
  The data does not follow a normal distribution.

In [92]:
```
# Perform Shapiro-Wilk test for osrm_time
stat, p_value = stats.shapiro(osrm_time)

print(f"Shapiro-Wilk Statistic: {stat}, p-value: {p_value}")

# Interpret results
if p_value > 0.05:
    print("Data is likely normally distributed.")
else:
    print("Data is not normally distributed.")
```

```
Shapiro-Wilk Statistic: 0.9474644660949707, p-value: 0.0
Data is not normally distributed.
```

In [93]:
```
# Perform Shapiro-Wilk test for segment_osrm_time
stat, p_value = stats.shapiro(segment_osrm_time)

print(f"Shapiro-Wilk Statistic: {stat}, p-value: {p_value}")

# Interpret results
if p_value > 0.05:
    print("Data is likely normally distributed.")
else:
    print("Data is not normally distributed.")
```

```
Shapiro-Wilk Statistic: 0.9696999192237854, p-value: 0.0
Data is not normally distributed.
```

Key Observation:

- Since the samples do not follow any assumptions of T-Test, T-test cannot be applied here, we can perform its non parametric equivalent tests like Mann-Whitney U rank test for two independent samples OR Wilcoxon test for two related samples.

Wilcoxon Signed-Rank Test: Brief Overview

The **Wilcoxon Signed-Rank Test** is a non-parametric test used to compare two related samples, matched samples, or repeated measurements on the same individuals.
It is used when the assumptions of the paired t-test (such as normality) are violated.

Hypotheses in Wilcoxon Signed-Rank Test

- **Null Hypothesis (H_0)**:
  The median difference between the paired observations is zero (no significant difference).
- **Alternative Hypothesis (H_1)**:
  The median difference between the paired observations is not zero (a significant difference exists).

In [94]:
```
# Perform Wilcoxon Signed-Rank Test

k_stat, p_value = stats.wilcoxon(osrm_time, segment_osrm_time)

print(f"Wilcoxon Statistic: {k_stat}, p-value: {p_value}")

# Interpret the results
if p_value > 0.05:
    print("No significant difference between the paired groups.")
else:
    print("Significant difference between the paired groups.")
```

```
Wilcoxon Statistic: 12565842.0, p-value: 0.0
Significant difference between the paired groups.
```

Delhivery Business Case Study: Overall Summary

Insights:

1. **Operational Inefficiencies:**
   - Significant differences between predicted (OSRM) and actual delivery times highlight inefficiencies in route planning and execution.
   - High-demand routes, such as those involving **Gurgaon** and **Mumbai**, dominate operations, making them critical zones for optimization.
2. **Skewed Data Distribution:**
   - Delivery time and distance data are **left-skewed**, indicating potential outliers or inconsistencies that may impact decision-making accuracy.
3. **Frequent Route Usage:**
   - Some routes are used up to **53 times**, and **"Carting"** is the most common route type. Optimizing these high-frequency operations can significantly improve cost efficiency.
4. **Regional Concentration:**
   - **Maharashtra** emerges as a key region, serving as both a frequent source and destination, suggesting a need for focused resource allocation and operational improvements.
5. **Unique Trips:**
   - Each trip has a unique identifier, offering an opportunity to track performance granularly and identify bottlenecks or inefficiencies.
6. **Minimal Data Gaps:**
   - Null values and unknown columns have a negligible impact (<0.2%) and can be addressed with minimal effort.

Observations:

1. **Prediction vs. Reality Gap:**
    - OSRM models fail to capture real-world conditions accurately, leading to mismatches in predicted and actual delivery times.
2. **High-Demand Areas Drive Costs:**
    - High delivery volumes in cities like **Mumbai** and states like **Maharashtra** may strain resources without optimized processes.
3. **Data Transformation Success:**
    - Transformations like **log** and **Box-Cox** effectively addressed skewness, enabling better insights for decision-making.

Actionable Items for Profitability:

1. **Optimize Route Planning:**
    - Refine the **OSRM model** or adopt **machine learning-based solutions** to account for real-world delays, improving delivery time predictions.
    - Focus optimization efforts on the most frequently used routes and high-demand areas.
2. **Invest in High-Impact Regions:**
    - Strategically allocate resources to hubs like **Gurgaon** and **Mumbai**, ensuring infrastructure, manpower, and fleet capacities meet demand.
3. **Enhance Data Utilization:**
    - Leverage unique trip identifiers for granular performance analysis to pinpoint inefficiencies.
    - Implement **geospatial analytics** to visualize delivery routes and optimize fleet allocation.
4. **Improve Operational Efficiency:**
    - Prioritize **"Carting" route types** for process improvements, as they represent the majority of operations.
    - Address inefficiencies in low-frequency routes to reduce unnecessary costs.
5. **Ensure Data Consistency:**
    - Regularly audit and clean data to ensure accuracy, focusing on correcting mismatched source and destination locations.
6. **Regional Focus:**
    - Concentrate operational improvements in **Maharashtra**, given its high delivery volume. Establish dedicated teams or resources for this region.
7. **Strategic Investments:**
    - Invest in technology such as **route optimization tools** and **predictive analytics**, along with workforce training, to enhance reliability and customer satisfaction.

By addressing inefficiencies and leveraging data insights effectively, **Delhivery** can streamline operations, reduce costs, and achieve greater profitability in its logistics and delivery services.

In [ ]: