About Delhivery

Delhivery is the largest and fastest-growing fully integrated player in India by revenue in Fiscal 2021. They aim to build the operating system for commerce, through a combination of world-class infrastructure, logistics operations of the highest quality, and cutting-edge engineering and technology capabilities.
The Data team builds intelligence and capabilities using this data that helps them to widen the gap between the quality, efficiency, and profitability of their business versus their competitors.

Column Profiling:

- **data - tells whether the data is testing or training data**
- **trip_creation_time – Timestamp of trip creation**
- **route_schedule_uuid – Unique Id for a particular route schedule**
- **route_type – Transportation type**
  - **FTL – Full Truck Load: FTL shipments get to the destination sooner, as the truck is making no other pickups or drop-offs along the way**
  - **Carting: Handling system consisting of small vehicles (carts)**
- **trip_uuid - Unique ID given to a particular trip (A trip may include different source and destination centers)**
- **source_center - Source ID of trip origin**
- **source_name - Source Name of trip origin**
- **destination_cente – Destination ID**
- **destination_name – Destination Name**
- **od_start_time – Trip start time**
- **od_end_time – Trip end time**
- **start_scan_to_end_scan – 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 Kms between source and ***
- **destination warehouse**
- **actual_time – Actual time taken to complete the delivery (Cumulative)**
- **osrm_time – An open-source routing engine time calculator which computes the shortest path between points in a given map (Includes usual traffic, distance through major and minor roads) and gives the time (Cumulative)**
- **osrm_distance – An open-source routing engine which computes the shortest path between points in a given map (Includes usual traffic, distance through major and minor roads) (Cumulative)**
- **factor – Unknown field**

- **segment_actual_time – This is a segment time. Time taken by the subset of the package delivery**
- **segment_osrm_time – This is the OSRM segment time. Time taken by the subset of the package delivery**
- **segment_osrm_distance – This is the OSRM distance. Distance covered by subset of the package delivery**
- **segment_factor – Unknown field**

Concept Used:

1. **Feature Creation**
2. **Relationship between Features**
3. **Column Normalization /Column Standardization**
4. **Handling categorical values**
5. **Missing values - Outlier treatment / Types of outliers**

1. Basic data cleaning and exploration

In [3]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import levene,shapiro,kruskal,zscore,probplot
from scipy.stats import ks_2samp
from statsmodels.graphics.gofplots import qqplot
from sklearn.preprocessing import StandardScaler,MinMaxScaler,OneHotEncoder
import warnings
warnings.filterwarnings('ignore')
```

In [4]:
```
df = pd.read_csv('delhivery_data.csv')
print("First 5 rows from delhivery dataset:\n")
df.head()
```

```
First 5 rows from delhivery dataset:
```

Out[4]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	so
0	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	INI
1	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	INI
2	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	INI
3	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	INI
4	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	INI

5 rows × 24 columns

In [5]:

```
print("\nColumn names and their data types:")
print(df.info())
```

```
Column names and their data types:
<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
None
```

In [6]:
```
print("Shape of dataset:",df.shape)
```

```
Shape of dataset: (144867, 24)
```

In [7]:
```
print("DATA TYPES:\n",df.dtypes)
```

```
DATA TYPES:
 data                             object
trip_creation_time               object
route_schedule_uuid              object
route_type                       object
trip_uuid                        object
source_center                    object
source_name                      object
destination_center               object
destination_name                 object
od_start_time                    object
od_end_time                      object
start_scan_to_end_scan          float64
is_cutoff                          bool
cutoff_factor                     int64
cutoff_timestamp                 object
actual_distance_to_destination  float64
actual_time                     float64
osrm_time                       float64
osrm_distance                   float64
factor                          float64
segment_actual_time             float64
segment_osrm_time               float64
segment_osrm_distance           float64
segment_factor                  float64
dtype: object
```

data types of columns

1. **int64: cutoff_factor**
2. **bool: is_cutoff**
3. **float64:**
   A. **start_scan_to_end_scan**
   B. **actual_distance_to_destination**
   C. **actual_time**
   D. **osrm_time**
   E. **osrm_distance**
   F. **segment_actual_time**
   G. **segment_osrm_time**
   H. **segment_osrm_distance**
   I. **segment_factor**
   J. **factor**
4. **object:**
   A. **data**
   B. **trip_creation_time**
   C. **route_schedule_uuid**
   D. **route_type**
   E. **trip_uuid**
   F. **source_center**
   G. **source_name**

H. **destination_center**
I. **destination_name**
J. **od_start_time**
K. **od_end_time**
L. **cutoff_timestamp**

Converting datatype

1. data,route_type to category
2. conataining time data to time

In [10]:
```
df['data'] = df['data'].astype('category')
df['route_type'] = df['route_type'].astype('category')
date_cols = ['trip_creation_time', 'od_start_time', 'od_end_time', 'cutoff_timestam
for col in date_cols:
    df[col] = pd.to_datetime(df[col], errors='coerce')
```

In [11]:
```
print("DESCRIPTION OF DATA:\n")
df.describe(include='all')
```

```
DESCRIPTION OF DATA:
```

Out[11]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uui
count	144867	144867	144867	144867	14486
unique	2	NaN	1504	2	1481
top	training	NaN	thanos::sroute:4029a8a2-6c74-4b7e-a6d8-f9e069f...	FTL	trip 15381121953589655
freq	104858	NaN	1812	99660	10
mean	NaN	2018-09-22 13:34:23.659819264	NaN	NaN	Na
min	NaN	2018-09-12 00:00:16.535741	NaN	NaN	Na
25%	NaN	2018-09-17 03:20:51.775845888	NaN	NaN	Na
50%	NaN	2018-09-22 04:24:27.932764928	NaN	NaN	Na
75%	NaN	2018-09-27 17:57:56.350054912	NaN	NaN	Na
max	NaN	2018-10-03 23:59:42.701692	NaN	NaN	Na
std	NaN	NaN	NaN	NaN	Na

11 rows × 24 columns

Dropping unknown fields

- Here the data contains some **Unknown fields** such as **"is_cutoff","cutoff_factor","cutoff_timestamp","factor","segment_factor"** ,since these fields are not required and no possible insights can be taken from these so drop them

In [13]:
```
unknown = ["is_cutoff","cutoff_factor","cutoff_timestamp","factor","segment_factor"
df.drop(columns = unknown,inplace = True)
```

In [14]:
```
df.describe()
```

Out[14]:

	trip_creation_time	od_start_time	od_end_time	start_scan_to_end_scan	ac
count	144867	144867	144867	144867.000000	
mean	2018-09-22 13:34:23.659819264	2018-09-22 18:02:45.855230720	2018-09-23 10:04:31.395393024	961.262986	
min	2018-09-12 00:00:16.535741	2018-09-12 00:00:16.535741	2018-09-12 00:50:10.814399	20.000000	
25%	2018-09-17 03:20:51.775845888	2018-09-17 08:05:40.886155008	2018-09-18 01:48:06.410121984	161.000000	
50%	2018-09-22 04:24:27.932764928	2018-09-22 08:53:00.116656128	2018-09-23 03:13:03.520212992	449.000000	
75%	2018-09-27 17:57:56.350054912	2018-09-27 22:41:50.285857024	2018-09-28 12:49:06.054018048	1634.000000	
max	2018-10-03 23:59:42.701692	2018-10-06 04:27:23.392375	2018-10-08 03:00:24.353479	7898.000000	
std	NaN	NaN	NaN	1037.012769	

Handling Missing values:

In [16]:

```
print("Missing values:\n", df.isna().sum())
```

```
Missing values:
 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
actual_distance_to_destination   0
actual_time                      0
osrm_time                        0
osrm_distance                    0
segment_actual_time              0
segment_osrm_time                0
segment_osrm_distance            0
dtype: int64
```

Inferences

- The **"Source_name"** has 293 missing values and 261 in **"destination_name"** when compared to total size(144867 rows) these are small in number so remove the rows with missing values or can also fill the missing values with the destination center and source center.

In [18]:
```python
c = 1
def create_center_name(df, center_col, name_col):
    global c
    mapping = {}
    centers = df[df[name_col].isna()][center_col].unique()

    for center in centers:
        names = df.loc[df[center_col] == center, name_col].dropna().unique()
        if len(names) > 0:
            mapping[center] = names[0]
        else:
            mapping[center] = f'location_{c}'
            c += 1
    return mapping

dest_map = create_center_name(df, 'destination_center', 'destination_name')
source_map = create_center_name(df, 'source_center', 'source_name')

df['destination_name'] = df.apply(
    lambda row: dest_map.get(row['destination_center'], row['destination_name'])
    if pd.isna(row['destination_name']) else row['destination_name'],
    axis=1
)

df['source_name'] = df.apply(
    lambda row: source_map.get(row['source_center'], row['source_name'])
    if pd.isna(row['source_name']) else row['source_name'],
    axis=1
)

print(df.isnull().sum())
```

```
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
actual_distance_to_destination    0
actual_time                       0
osrm_time                         0
osrm_distance                     0
segment_actual_time               0
segment_osrm_time                 0
segment_osrm_distance             0
dtype: int64
```

In [19]:
```
filtered_df = df[df['actual_distance_to_destination'] > 0]

avg_distance = filtered_df['actual_distance_to_destination'].mean()
min_distance = filtered_df['actual_distance_to_destination'].min()
max_distance = filtered_df['actual_distance_to_destination'].max()

# Print results
print(f"Average distance: {avg_distance:.2f} km")
print(f"Minimum distance: {min_distance:.2f} km")
print(f"Maximum distance: {max_distance:.2f} km")
print(f"date start time: {min(df['trip_creation_time'])} to end date: {max(df['trip
```

```
Average distance: 234.07 km
Minimum distance: 9.00 km
Maximum distance: 1927.45 km
date start time: 2018-09-12 00:00:16.535741 to end date: 2018-10-03 23:59:42.701692
```

observations

1. The data provided from **2018-09-12 00:00:16.535741 to end date: 2018-10-03 23:59:42.701692**
2. The **Average** distance between source and destination is **234.07 km** with **Least** distance being **9.00 km** and **maximum** with **1927.45 km**

Grouping and aggregating segments

1. Grouping by segment

a. 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**.

b. You can use inbuilt functions like groupby and aggregations like cumsum() to merge the rows in columns **segment_actual_time,segment_osrm_distance, segment_osrm_time** based on the **segment_key**.

c. This way you'll get new columns named **segment_actual_time_sum,segment_osrm_distance_sum, segment_osrm_time_sum**.

2. Aggregating at segment level

a. Create a dictionary named **create_segment_dict**, that defines how to aggregate and select values.

- You can keep the first and last values for some numeric/categorical fields if aggregating them won't make sense.

b. Further group the data by segment_key because you want to perform aggregation operations for different segments of each trip based on the segment_key value.

c. The aggregation functions specified in the create_segment_dict are applied to each group of rows with the same segment_key.

d. Sort the resulting DataFrame segment, by two criteria:

- First, it sorts by segment_key to ensure that segments are **ordered consistently**.
- Second, it sorts by od_end_time in ascending order, ensuring that segments within the same trip are ordered by their end times from earliest to latest.

In [22]:
```
df['segment_key'] = df['trip_uuid'] + df['source_center'] + df['destination_center'
cols = ['segment_actual_time', 'segment_osrm_time', 'segment_osrm_distance']
for col in cols:
    df[col + '_sum'] = df.groupby('segment_key')[col].cumsum()
print(f"{df[[col +'_sum' for col in cols]].head().to_markdown(index=False, numalign
```

```
| segment_actual_time_sum   | segment_osrm_time_sum   | segment_osrm_distance_sum
|
|:-------------------------|:-----------------------|:---------------------------
|
| 14                       | 11                     | 11.9653
|
| 24                       | 20                     | 21.7243
|
| 40                       | 27                     | 32.5395
|
| 61                       | 39                     | 45.5619
|
| 67                       | 44                     | 49.4772
|
```

In [23]:
```
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': 'last',
    'osrm_distance': 'last',

    'segment_actual_time_sum': 'last',
    'segment_osrm_distance_sum': 'last',
    'segment_osrm_time_sum': 'last'
}
```

```
trip = df.groupby('segment_key').agg(create_segment_dict).reset_index()
trip = trip.sort_values(by=['segment_key', 'od_end_time'], ascending=True).reset_in
trip.head()
```

Out[23]:

	segment_key	data	trip_creation_time	route_scl
0	trip-153671041653548748IND209304AAAIND000000ACB	training	2018-09-12 00:00:16.535741	thanos::srout a29k
1	trip-153671041653548748IND462022AAAIND209304AAA	training	2018-09-12 00:00:16.535741	thanos::srout a29k
2	trip-153671042288605164IND561203AABIND562101AAA	training	2018-09-12 00:00:22.886430	thanos::srout bb0k
3	trip-153671042288605164IND572101AAAIND561203AAB	training	2018-09-12 00:00:22.886430	thanos::srout bb0k
4	trip-153671043369099517IND000000ACBIND160002AAC	training	2018-09-12 00:00:33.691250	thanos::srout 7641

Observations:

1. It's important to conceptually validate that cumsum() is the correct aggregation for the segment-level time and distance.
2. For very large datasets, groupby().agg() is efficient, but for extremely complex aggregations or very high cardinality keys, performance should be monitored

2.Feature Engineering

1. Calculate the time between od_start_time and od_end_time**and keep it as a feature named od_time_diff_hour. Drop the original columns, if required**

In [27]:

```
print(f"\nNumber of unique trip_uuid: {df['trip_uuid'].nunique()}")
print(f"Number of unique segment_key: {df['segment_key'].nunique()}")
trip['od_time_diff'] = (trip['od_end_time'] - trip['od_start_time']).dt.total_secon
print(trip['od_time_diff'].head().to_markdown(index=False, numalign="left", stralig
```

```
Number of unique trip_uuid: 14817
Number of unique segment_key: 26368
| od_time_diff   |
|:---------------|
| 21.0101        |
| 16.6584        |
| 0.98054        |
| 2.04632        |
| 13.9106        |
```

2. Destination Name: Split and extract features out of destination. City-place-code (State)

3. Source Name: Split and extract features out of destination. City-place-code (State)

In [29]:
```python
def state(x):
    if pd.isna(x) or not isinstance(x, str): return 'Unknown_State'
    try: return x.split('(')[1].replace(')', '').strip()
    except IndexError: return 'Unknown_State'

def city(x):
    if pd.isna(x) or not isinstance(x, str): return 'Unknown_City'
    try: processed_x = x.split('(')[0].strip()
    except IndexError: return 'Unknown_City'
    city = processed_x.split('_')[0].strip() if '_' in processed_x else processed_x
    if city.lower() in ['mumbai antop hill', 'lowerparel', 'bom', 'mumbai hub']: re
    elif city.lower() == 'pnq vadgaon sheri dpc': return 'vadgaonsheri'
    elif city.lower() in ['pnq pashan dpc', 'pnq rahatani dpc', 'pune balaji nagar'
    elif city.lower() in ['bangalore', 'hbr layout pc', 'blr']: return 'bengaluru'
    elif city.lower() == 'bhopal mp nagar': return 'bhopal'
    elif city.lower() == 'amd': return 'ahmedabad'
    elif city.lower() == 'ccu': return 'kolkata'
    elif city.lower() == 'ggn': return 'gurgaon'
    elif city.lower() == 'gzb': return 'ghaziabad'
    return city

def place(x):
    if pd.isna(x) or not isinstance(x, str): return 'Unknown_Place_Code'
    try: processed_x = x.split('(')[0].strip()
    except IndexError: return 'Unknown_Place_Code'
    parts = processed_x.split('_')
    if len(parts) > 1: return '_'.join(parts[1:]).strip()
    else: return 'Unknown_Place_Code'

def code(x):
    if pd.isna(x) or not isinstance(x, str): return 'none'
    try:
        processed_x = x.split('(')[0].strip()
        parts = processed_x.split('_')
        if len(parts) >= 3: return parts[-1].strip()
        return 'none'
    except IndexError: return 'none'
```

In [30]:
```python
print("Source details:\n")
trip['source_state'] = trip['source_name'].apply(lambda x: state(x))
```

```
trip['source_city'] = trip['source_name'].apply(lambda x: city(x))
trip['source_place'] = trip['source_name'].apply(lambda x: place(x))
trip['source_code'] = trip['source_name'].apply(lambda x: code(x))
print(f"{trip[['source_name','source_state','source_city','source_place','source_co
print("Destination details:\n")
trip['destination_state'] = trip['destination_name'].apply(lambda x: state(x))
trip['destination_city'] = trip['destination_name'].apply(lambda x: city(x))
trip['destination_place'] = trip['destination_name'].apply(lambda x: place(x))
trip['destination_code'] = trip['destination_name'].apply(lambda x: code(x))
print(f"{trip[['destination_name','destination_state','destination_city','destinati
```

```
Source details:

| source_name                       | source_state   | source_city   | source_place
| source_code   |
|:----------------------------------|:---------------|:--------------|:------------
---|:--------------|
| Kanpur_Central_H_6 (Uttar Pradesh) | Uttar Pradesh  | Kanpur        | Central_H_6
| 6             |
| Bhopal_Trnsport_H (Madhya Pradesh) | Madhya Pradesh | Bhopal        | Trnsport_H
| H             |
| Doddablpur_ChikaDPP_D (Karnataka)  | Karnataka      | Doddablpur    | ChikaDPP_D
| D             |
| Tumkur_Veersagr_I (Karnataka)      | Karnataka      | Tumkur        | Veersagr_I
| I             |
| Gurgaon_Bilaspur_HB (Haryana)      | Haryana        | Gurgaon       | Bilaspur_HB
| HB            |

Destination details:

| destination_name                  | destination_state    | destination_city    | de
stination_place    | destination_code    |
|:----------------------------------|:---------------------|:--------------------|:--
------------------|:-------------------|
| Gurgaon_Bilaspur_HB (Haryana)      | Haryana              | Gurgaon             | Bi
laspur_HB          | HB                  |
| Kanpur_Central_H_6 (Uttar Pradesh) | Uttar Pradesh        | Kanpur              | Ce
ntral_H_6          | 6                   |
| Chikblapur_ShntiSgr_D (Karnataka)  | Karnataka            | Chikblapur          | Sh
ntiSgr_D           | D                   |
| Doddablpur_ChikaDPP_D (Karnataka)  | Karnataka            | Doddablpur          | Ch
ikaDPP_D           | D                   |
| Chandigarh_Mehmdpur_H (Punjab)     | Punjab               | Chandigarh          | Me
hmdpur_H           | H                   |
```

4. Trip_creation_time: Extract features like month, year, day, etc.

In [32]:
```
trip['trip_yr'] = trip['trip_creation_time'].dt.year
trip['trip_mnth'] = trip['trip_creation_time'].dt.month
trip['trip_day'] = trip['trip_creation_time'].dt.day
trip['trip_dayofweek'] = trip['trip_creation_time'].dt.dayofweek
trip['trip_creation_hr'] = trip['trip_creation_time'].dt.hour
print(f"{trip[['trip_creation_time','trip_yr','trip_mnth','trip_day','trip_dayofwee
```

```
| trip_creation_time         | trip_yr   | trip_mnth   | trip_day   | trip_dayofweek
| trip_creation_hr   |
|:--------------------------|:----------|:------------|:-----------|:--------------
---|:-------------------|
| 2018-09-12 00:00:16.535741 | 2018      | 9           | 12         | 2
| 0                  |
| 2018-09-12 00:00:16.535741 | 2018      | 9           | 12         | 2
| 0                  |
| 2018-09-12 00:00:22.886430 | 2018      | 9           | 12         | 2
| 0                  |
| 2018-09-12 00:00:22.886430 | 2018      | 9           | 12         | 2
| 0                  |
| 2018-09-12 00:00:33.691250 | 2018      | 9           | 12         | 2
| 0                  |
```

Insights:

1. **od_time_diff:** This new feature directly quantifies the duration of each origin-destination segment in hours and is highly relevant metric for understanding and predicting delivery times.
2. Using **source_name and destination_name**,extracted **city, place_code, and state information**, granular geographical features can be instrumental in identifying popular regions, problematic corridors, or specific centers that might require operational adjustments.
3. Extracting **year, month, day, day of week, hour** from **trip_creation_time** provides valuable temporal context and can help to identify seasonal trends, daily patterns or weekly.

Recommendations:

1. By exploring the interactions between these newly engineered features (e.g., od_time_diff vs. trip_creation_hr for different route_types) can uncover more complex patterns.
2. Consider mapping source_center and destination_center IDs to actual geographical coordinates to enable spatial analysis.

3. In-depth analysis:

1. Grouping and Aggregating at Trip-level

- Groups the segment data by the trip_uuid column to focus on aggregating data at the trip level.
- Apply suitable aggregation functions like first, last, and sum specified in the create_trip_dict dictionary to calculate summary statistics for each trip.

In [36]:
```python
create_trip_dict = {
    'data' : 'first',
    'trip_creation_time' : 'first',
```

```python
    'route_schedule_uuid' : 'first',
    'route_type' : 'first',
    'source_center' : 'first',
    'source_name' : 'first',
    'trip_uuid': 'first',
    'destination_center' : 'last',
    'destination_name' : 'last',

    'source_state' : 'first',
    'destination_state' : 'last',

    'source_city' : 'first',
    'source_place' : 'first',
    'source_code' : 'first',

    'destination_city' : 'last',
    'destination_place' : 'last',
    'destination_code' : 'last',

    'start_scan_to_end_scan' : 'sum',
    'od_time_diff' : 'sum',

    'actual_distance_to_destination' : 'sum',
    'actual_time' : 'sum',
    'osrm_time' : 'sum',
    'osrm_distance' : 'sum',
    'segment_actual_time_sum' : 'sum',
    'segment_osrm_distance_sum' : 'sum',
    'segment_osrm_time_sum' : 'sum',
}

trip_df = trip.groupby('trip_uuid').agg(create_trip_dict).reset_index(drop = True)
print("\nColumn Information of Trip-level DataFrame:\n")
trip_df.info()
print("\nDescriptive Statistics for Numerical Columns:")
trip_df.describe()
trip_df.shape
```

```
Column Information of Trip-level DataFrame:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14817 entries, 0 to 14816
Data columns (total 26 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   data                            14817 non-null  category
 1   trip_creation_time              14817 non-null  datetime64[ns]
 2   route_schedule_uuid             14817 non-null  object
 3   route_type                      14817 non-null  category
 4   source_center                   14817 non-null  object
 5   source_name                     14817 non-null  object
 6   trip_uuid                       14817 non-null  object
 7   destination_center              14817 non-null  object
 8   destination_name                14817 non-null  object
 9   source_state                    14817 non-null  object
 10  destination_state               14817 non-null  object
 11  source_city                     14817 non-null  object
 12  source_place                    14817 non-null  object
 13  source_code                     14817 non-null  object
 14  destination_city                14817 non-null  object
 15  destination_place               14817 non-null  object
 16  destination_code                14817 non-null  object
 17  start_scan_to_end_scan          14817 non-null  float64
 18  od_time_diff                    14817 non-null  float64
 19  actual_distance_to_destination  14817 non-null  float64
 20  actual_time                     14817 non-null  float64
 21  osrm_time                       14817 non-null  float64
 22  osrm_distance                   14817 non-null  float64
 23  segment_actual_time_sum         14817 non-null  float64
 24  segment_osrm_distance_sum       14817 non-null  float64
 25  segment_osrm_time_sum           14817 non-null  float64
dtypes: category(2), datetime64[ns](1), float64(9), object(14)
memory usage: 2.7+ MB

Descriptive Statistics for Numerical Columns:
```

Out[36]:

```
(14817, 26)
```

2. Outlier Detection & Treatment

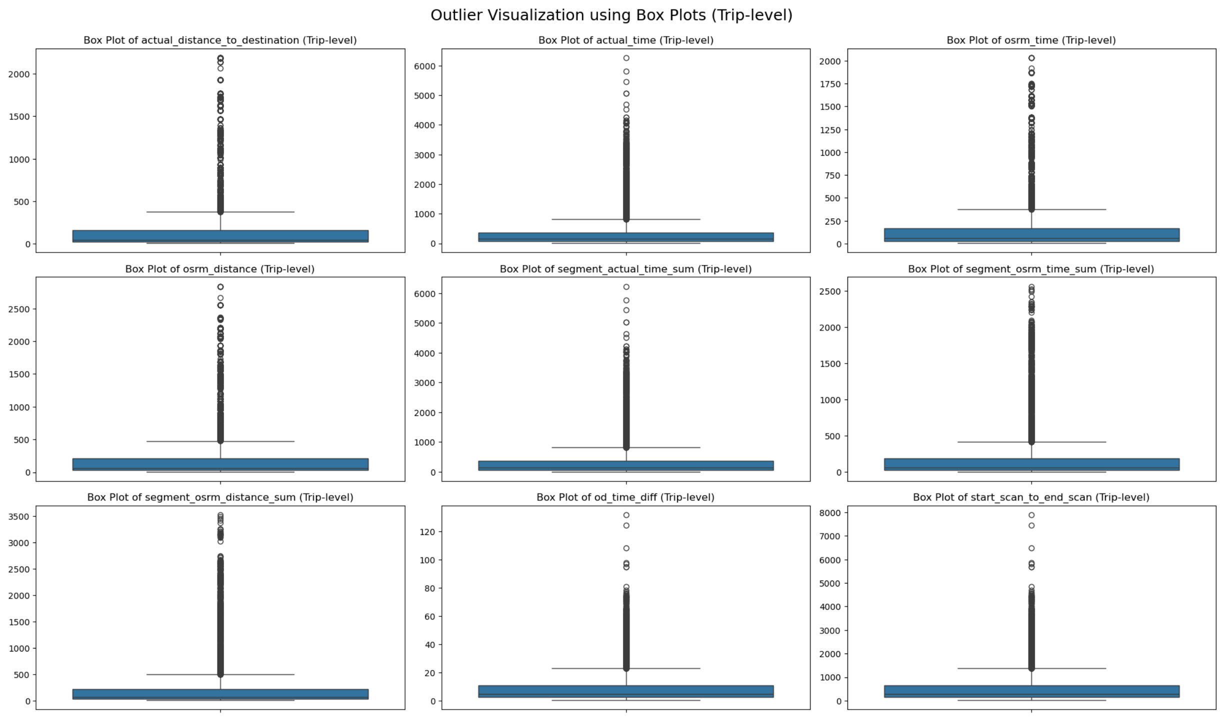
- **Find any existing outliers in numerical features.**
- **Visualize the outlier values using Boxplot.**
- **Handle the outliers using the IQR method.**

In [38]:

```python
numerical_cols_trip = [
    'actual_distance_to_destination', 'actual_time', 'osrm_time', 'osrm_distance','
    'start_scan_to_end_scan']
print(f"\nChecking for outliers in {len(numerical_cols_trip)} numerical columns in

plt.figure(figsize=(20, 15))
for i, col in enumerate(numerical_cols_trip):
    plt.subplot(4, 3, i + 1)
    sns.boxplot(y=trip_df[col])
```

```
    plt.title(f'Box Plot of {col} (Trip-level)')
    plt.ylabel('')
plt.tight_layout()
plt.suptitle('Outlier Visualization using Box Plots (Trip-level)', y=1.02, fontsize
plt.show()

print("\nSkewness of numerical columns in trip_df before treatment:")
print(trip_df[numerical_cols_trip].skew().to_markdown(numalign="left", stralign="le
```

```
Checking for outliers in 9 numerical columns in trip_df...
```

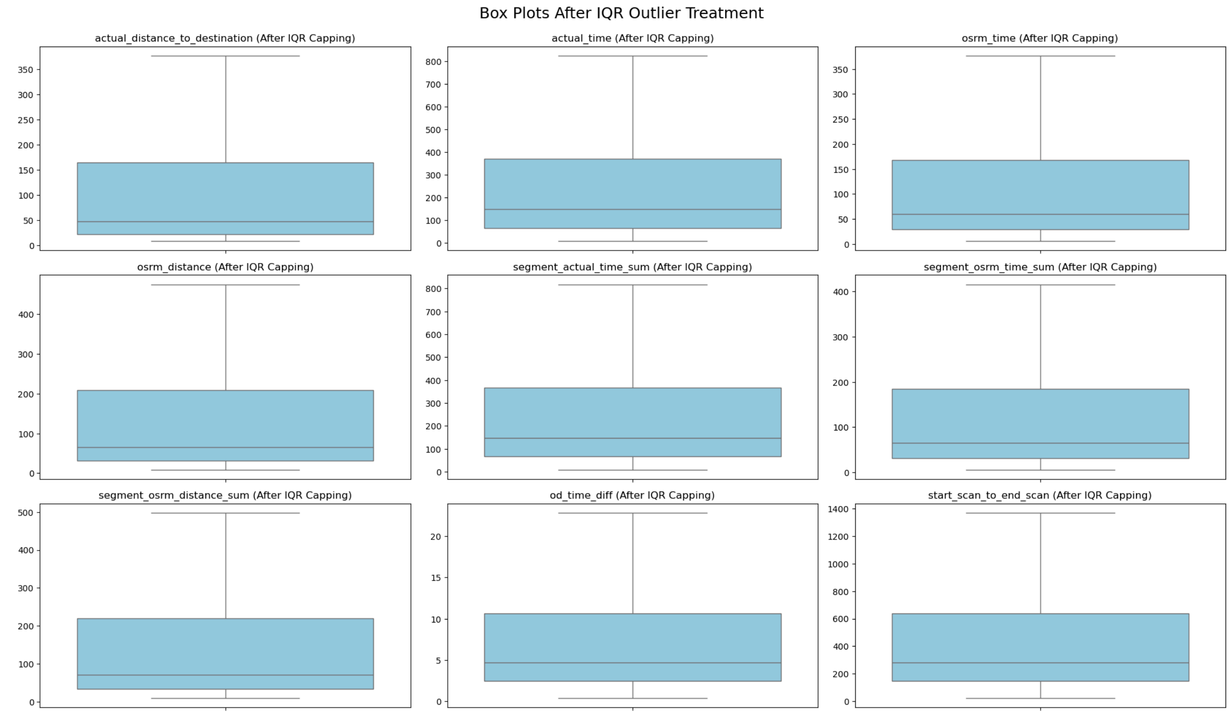



```
Skewness of numerical columns in trip_df before treatment:
|                                | 0        |
|:-------------------------------|:---------|
| actual_distance_to_destination | 3.55787  |
| actual_time                    | 3.36909  |
| osrm_time                      | 3.44952  |
| osrm_distance                  | 3.54917  |
| segment_actual_time_sum        | 3.36592  |
| segment_osrm_time_sum          | 3.59783  |
| segment_osrm_distance_sum      | 3.71012  |
| od_time_diff                   | 2.88425  |
| start_scan_to_end_scan         | 2.8863   |
```

In [39]:
```
def cap_outliers_iqr(series):
    Q1 = series.quantile(0.25)
    Q3 = series.quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    capped = np.where(series > upper_bound, upper_bound, series)
    capped = np.where(capped < lower_bound, lower_bound, capped)

    return pd.Series(capped, index=series.index)
```

```python
# Apply capping directly to the original DataFrame
for col in numerical_cols_trip:
    trip_df[col] = cap_outliers_iqr(trip_df[col])

# Display summary after capping
print("Descriptive statistics after IQR outlier treatment:")
print(trip_df[numerical_cols_trip].describe().to_markdown(numalign="left", stralign

# Box plots to visualize
plt.figure(figsize=(20, 15))
for i, col in enumerate(numerical_cols_trip):
    plt.subplot(4, 3, i + 1)
    sns.boxplot(y=trip_df[col], color="skyblue")
    plt.title(f'{col} (After IQR Capping)')
    plt.ylabel('')

plt.tight_layout()
plt.suptitle('Box Plots After IQR Outlier Treatment', y=1.02, fontsize=18)
plt.show()
```

```
Descriptive statistics after IQR outlier treatment:
|       | actual_distance_to_destination   | actual_time    | osrm_time    | osrm_dist
ance    | segment_actual_time_sum    | segment_osrm_time_sum    | segment_osrm_distance
_sum    | od_time_diff    | start_scan_to_end_scan    |
|:------|:---------------------------------|:--------------|:------------|:---------
-------|:-------------------------|:------------------------|:---------------------
-------|:---------------|:------------------------|
| count | 14817                            | 14817          | 14817        | 14817
| 14817                        | 14817                     | 14817
| 14817           | 14817                     |
| mean  | 109.059                          | 263.85         | 114.875      | 138.518
| 261.574                      | 126.124                   | 146.94
| 7.57222         | 453.399                   |
| std   | 117.389                          | 260.944        | 116.309      | 147.725
| 259.134                      | 128.453                   | 155.187
| 6.83735         | 409.881                   |
| min   | 9.00246                          | 9              | 6            | 9.0729
| 9                            | 6                         | 9.0729
| 0.391024        | 23                        |
| 25%   | 22.8372                          | 67             | 29           | 30.8192
| 66                           | 31                        | 32.6545
| 2.49884         | 149                       |
| 50%   | 48.4741                          | 149            | 60           | 65.6188
| 147                          | 65                        | 70.1544
| 4.67943         | 280                       |
| 75%   | 164.583                          | 370            | 168          | 208.475
| 367                          | 185                       | 218.802
| 10.6367         | 637                       |
| max   | 377.202                          | 824.5          | 376.5        | 474.959
| 818.5                        | 416                       | 498.024
| 22.8434         | 1369                      |
```



Inferences:

1. The **create_trip_dict**,important in defining how various metrics should be consolidated at the trip level.
2. These extreme values could heavily influence statistical models if not addressed.
3. The skewness values confirm this, often being high and positive.
4. The descriptive statistics after treatment show that the maximum values for these columns are now within a more reasonable range, indicating that extreme values have been constrained.
5. The re-plotted box plots visually confirm that the distribution is now more compact, with fewer extreme points.

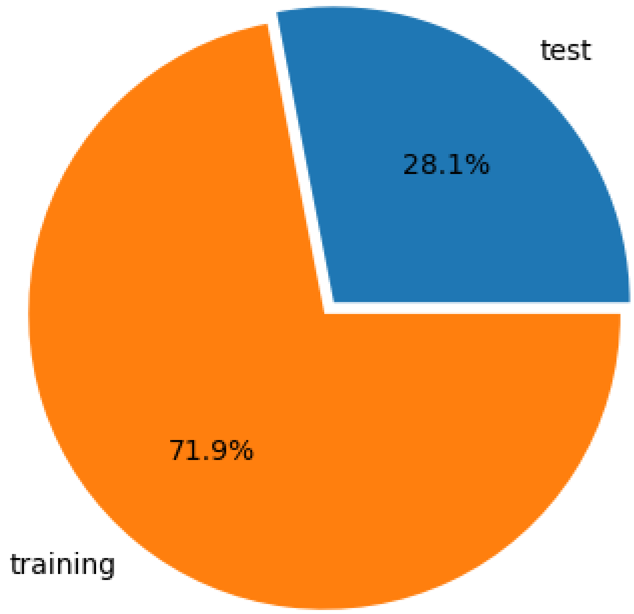
3. Perform one-hot encoding on categorical features.

- There are 2 categotical variables in data i.e.,data and route_type

In [42]:
```
trip_df['data'].value_counts()
```

Out[42]:
```
data
training    10654
test         4163
Name: count, dtype: int64
```

In [43]:
```
df_data = trip_df.groupby('data')['trip_uuid'].count().reset_index(name='trip_count
df_data['percentage'] = np.round(df_data['trip_count'] * 100 / df_data['trip_count'
plt.pie(x=df_data['trip_count'], labels=df_data['data'], explode=[0, 0.05], autopct
plt.show()
```

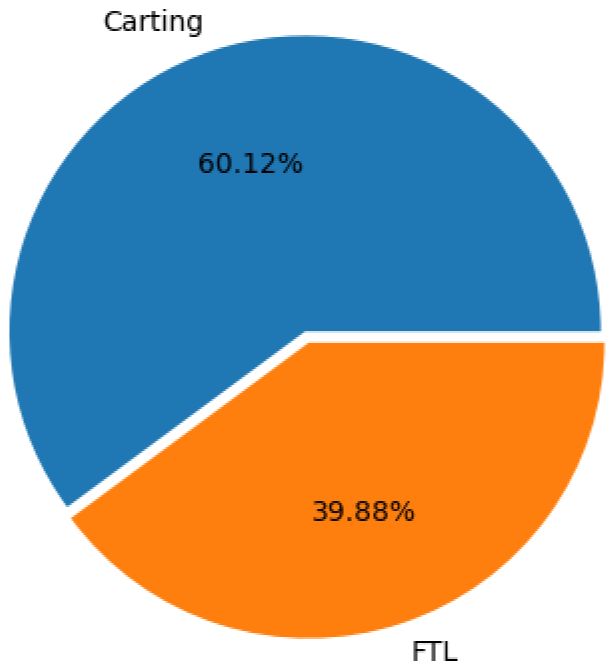


In [44]:
```
trip_df['route_type'].value_counts()
```

Out[44]:
```
route_type
Carting    8908
FTL        5909
Name: count, dtype: int64
```

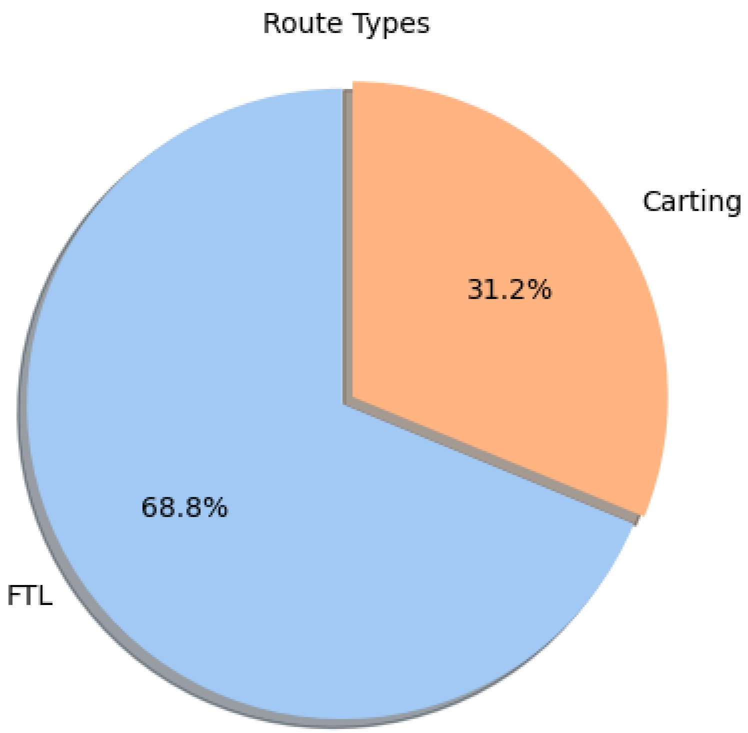
- The **route_type** categorical variable has two values Carting and FTL

In [46]:
```
df_route = trip_df.groupby('route_type')['trip_uuid'].count().to_frame().reset_inde
df_route['percent'] = np.round(df_route['trip_uuid'] * 100 / df_route['trip_uuid'].

plt.pie(x = df_route['trip_uuid'],labels =['Carting','FTL'],explode =[0,0.04],autop
plt.show()

print("\nDistribution of Trip Route Types:")
print(df_route.to_markdown(index=False, numalign="left", stralign="left"))
```




```
Distribution of Trip Route Types:
| route_type   | trip_uuid   | percent   |
|:-------------|:------------|:----------|
| Carting      | 8908        | 60.12     |
| FTL          | 5909        | 39.88     |
```

In [47]:
```
route_type_counts = df['route_type'].value_counts()
labels = route_type_counts.index.tolist()
sizes = route_type_counts.values.tolist()
colors = sns.color_palette('pastel')[0:len(labels)]
explode = [0, 0.04] if len(labels) == 2 else [0] * len(labels)  # Explode second sl
plt.figure(figsize=(4, 4))
plt.pie(sizes, explode=explode, labels=labels, colors=colors,autopct='%1.1f%%', sha
plt.title('Route Types', fontsize=10)
plt.axis('equal')
plt.tight_layout()
plt.show()
```



In [48]:
```
print("\n--- One-Hot Encoding ---")
categorical_cols = [
    col for col in trip_df.columns
    if trip_df[col].dtype == 'object' and col not in ['trip_uuid', 'route_schedule_
]
if not categorical_cols:
    print("No categorical columns found for one-hot encoding.")
else:
    print(f"Encoding the following categorical columns: {categorical_cols}")
    # Apply one-hot encoding
    trip_df_encoded = pd.get_dummies(trip_df, columns=categorical_cols, drop_first=
```

```
--- One-Hot Encoding ---
Encoding the following categorical columns: ['route_schedule_uuid', 'source_center',
'source_name', 'destination_center', 'destination_name', 'source_state', 'destinatio
n_state', 'source_city', 'source_place', 'source_code', 'destination_city', 'destina
tion_place', 'destination_code']
```

In [49]:
```
print("\n--- One-Hot Encoding ---")

categorical_cols = [
    col for col in trip_df.columns
    if trip_df[col].dtype == 'object' and col not in ['trip_uuid', 'route_schedule_
]
if not categorical_cols:
    print("No categorical columns found for one-hot encoding.")
else:
    print(f"Encoding the following categorical columns:\n{categorical_cols}\n")
    original_cols = set(trip_df.columns)
    trip_df_encoded = pd.get_dummies(trip_df, columns=categorical_cols, drop_first=
    new_cols = set(trip_df_encoded.columns) - original_cols
    print(f"Encoding complete. Added {len(new_cols)} new columns.")
```

```
    print(f"Sample new columns: {list(new_cols)[:10]}")
trip_df_encoded.shape
```

```
--- One-Hot Encoding ---
Encoding the following categorical columns:
['source_center', 'source_name', 'destination_center', 'destination_name', 'source_s
tate', 'destination_state', 'source_city', 'source_place', 'source_code', 'destinati
on_city', 'destination_place', 'destination_code']

Encoding complete. Added 7242 new columns.
Sample new columns: ['source_center_IND208017AAA', 'destination_city_Kalwakurthy',
'destination_name_Balaghat_Kosmi_D (Madhya Pradesh)', 'source_center_IND506002AAA',
'destination_name_JoguGadwal_ColctrOf_D (Telangana)', 'destination_place_Mwalibad_
D', 'destination_name_Lucknow_Pandriba_L (Uttar Pradesh)', 'destination_center_IND57
7116AAA', 'destination_name_Bettiah_BypassRd_D (Bihar)', 'source_center_IND140406AA
A']
```

Out[49]: (14817, 7256)

4. Normalize/ Standardize the numerical features using MinMaxScaler or StandardScaler.

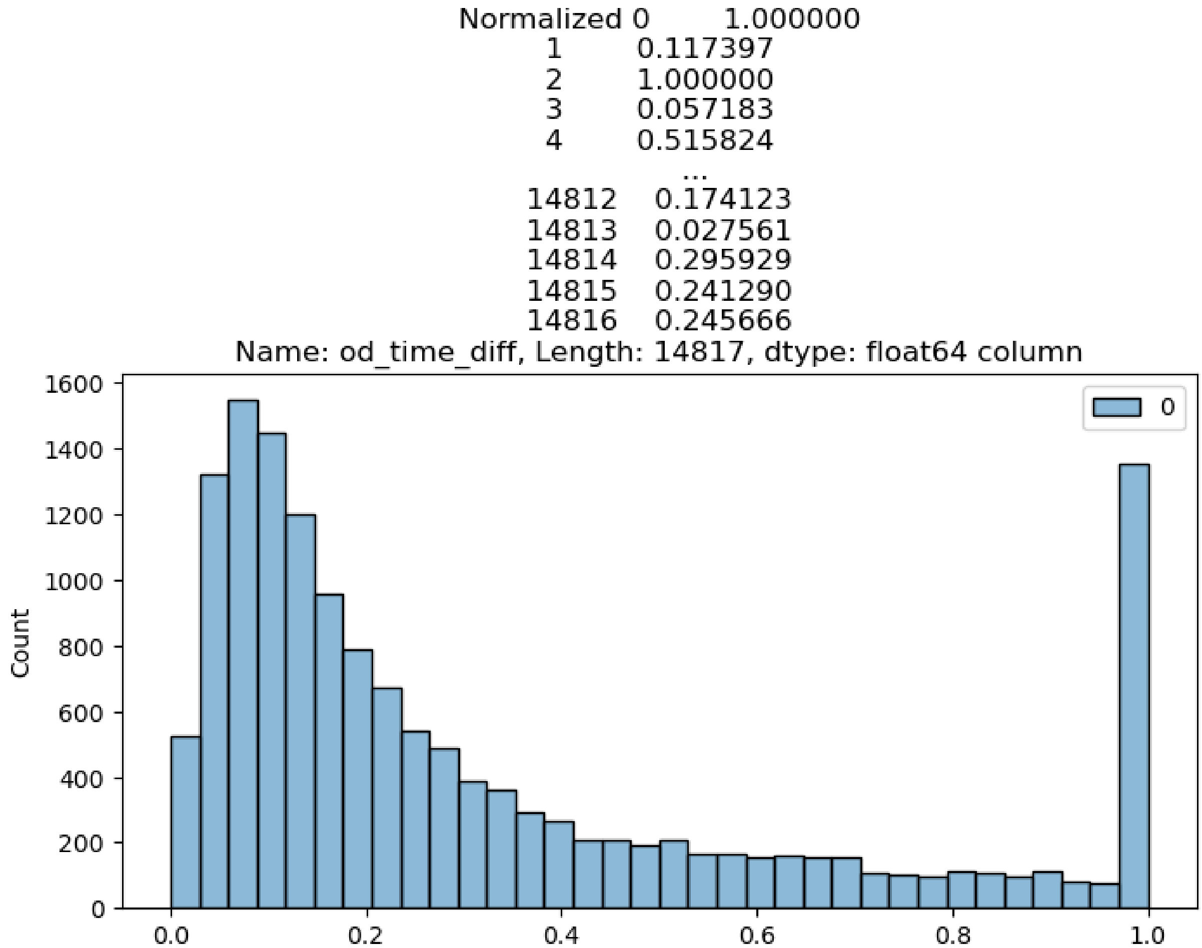
In [51]:
```
print("\n---  Normalization / Scaling using MinMaxScaler ---")

num_cols = trip_df.select_dtypes(include=np.number).columns.tolist()

scaler = MinMaxScaler()
scaler.fit(trip_df[num_cols])
trip_df[num_cols] = scaler.transform(trip_df[num_cols])
trip_df[num_cols].head()
```

```
---  Normalization / Scaling using MinMaxScaler ---
```

Out[51]:

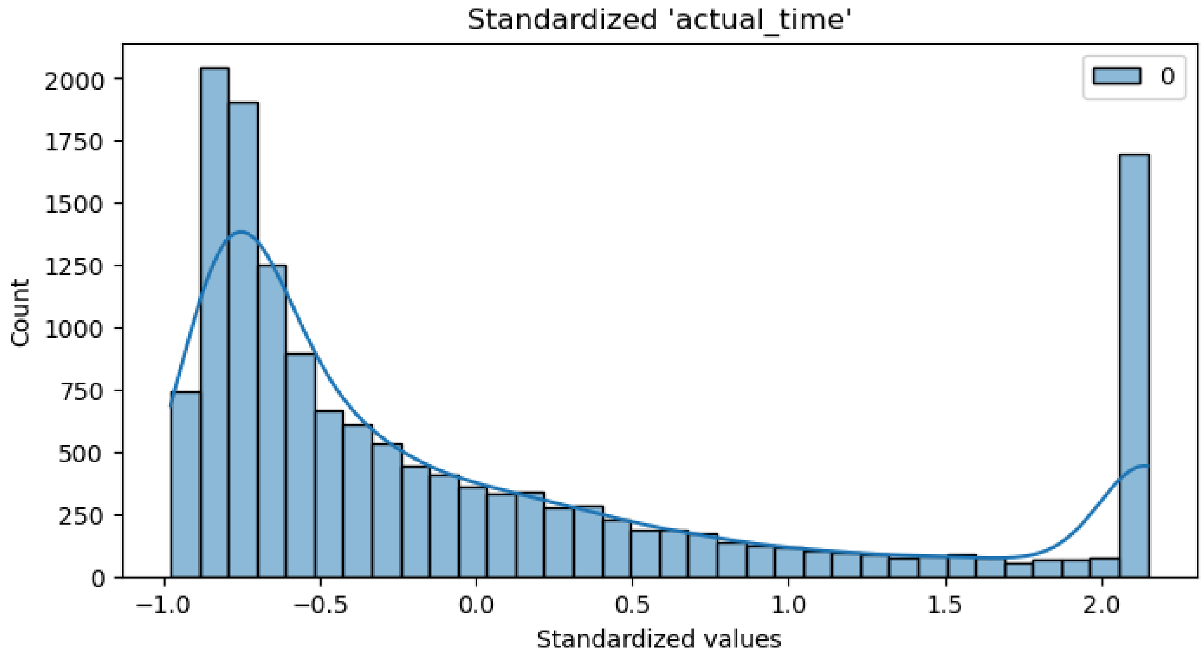
	start_scan_to_end_scan	od_time_diff	actual_distance_to_destination	actual_time	osrm_ti
0	1.000000	1.000000	1.000000	1.000000	1.0000
1	0.116642	0.117397	0.174320	0.164316	0.1673
2	1.000000	1.000000	1.000000	1.000000	1.0000
3	0.057207	0.057183	0.022197	0.061312	0.0242
4	0.515602	0.515824	0.321690	0.407112	0.2995

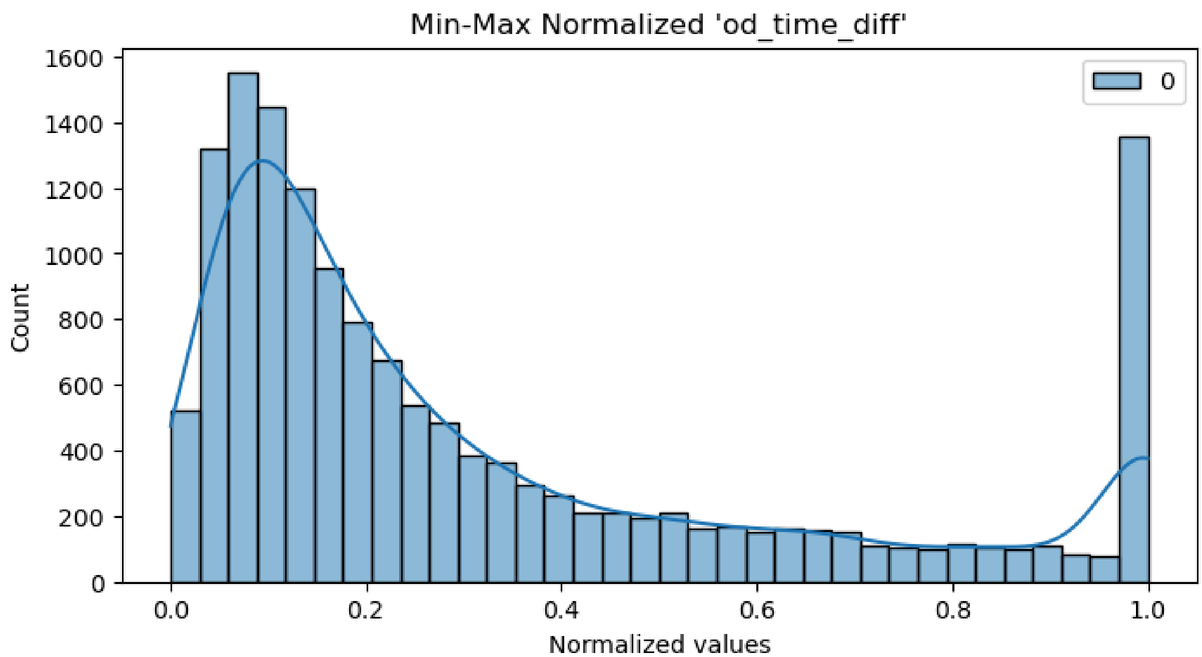
In [52]:
```
plt.figure(figsize=(8, 4))
scaler = MinMaxScaler()
scaled = scaler.fit_transform(trip_df['od_time_diff'].to_numpy().reshape(-1,1))
sns.histplot(scaled)
plt.title(f"Normalized {trip_df['od_time_diff']} column")
plt.show()
```

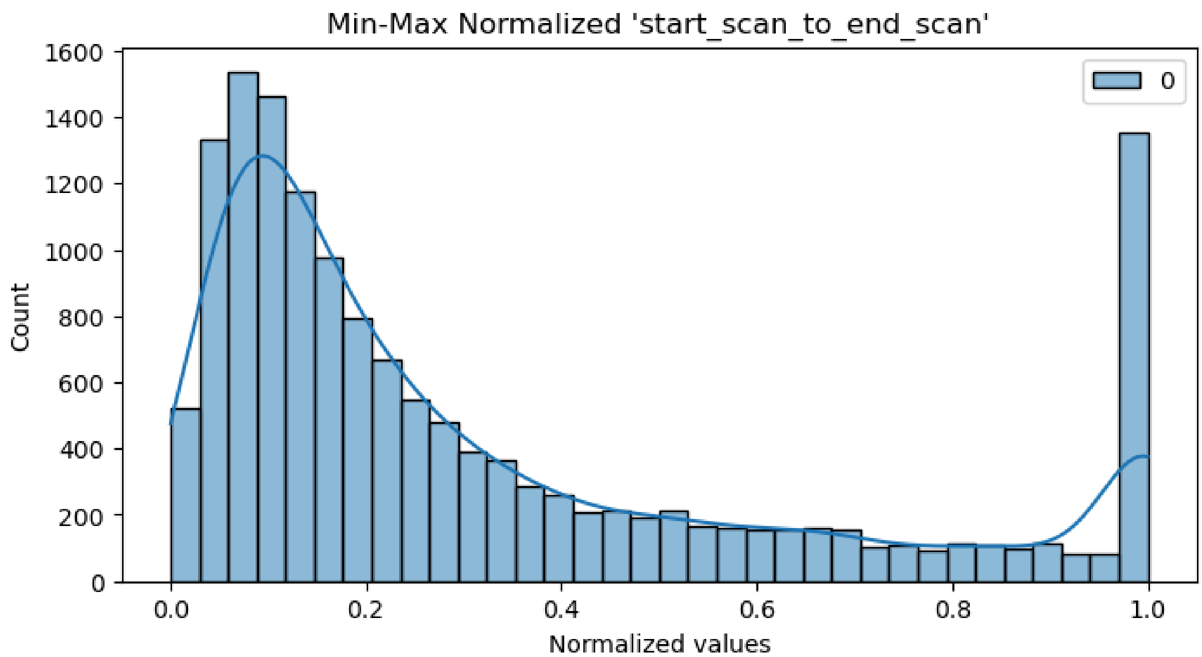


In [53]:

```
plt.figure(figsize=(8, 4))
standardized = StandardScaler().fit_transform(trip_df['actual_time'].to_numpy().res
sns.histplot(standardized, kde=True)
plt.title("Standardized 'actual_time'")
plt.xlabel("Standardized values")
plt.show()

# Normalize 'od_time_diff' from trip_df
plt.figure(figsize=(8, 4))
normalized_od = MinMaxScaler().fit_transform(trip_df['od_time_diff'].to_numpy().res
sns.histplot(normalized_od, kde=True)
plt.title("Min-Max Normalized 'od_time_diff'")
plt.xlabel("Normalized values")
plt.show()

# Normalize 'start_scan_to_end_scan' from trip_df
plt.figure(figsize=(8, 4))
normalized_scan = MinMaxScaler().fit_transform(trip_df['start_scan_to_end_scan'].to
sns.histplot(normalized_scan, kde=True)
plt.title("Min-Max Normalized 'start_scan_to_end_scan'")
plt.xlabel("Normalized values")
plt.show()
```


Standardized 'actual_time'
0
Count
Standardized values
2000
1750
1500
1250
1000
750
500
250
0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


Min-Max Normalized 'od_time_diff'
0
Count
Normalized values
1600
1400
1200
1000
800
600
400
200
0
0.0
0.2
0.4
0.6
0.8
1.0



Observations:

- **Normalization/Standardization:** Both StandardScaler and MinMaxScaler effectively transformed the numerical features. StandardScaler resulted in features with a mean close to 0 and a standard deviation close to 1, while MinMaxScaler scaled features into the range [0, 1].
- StandardScaler is typically preferred for algorithms that assume a Gaussian distribution
- The histogram you've provided shows the Min-Max Normalized distribution of the 'start_scan_to_end_scan' column and also suggests that many scan durations are short, but there are a few significantly longer durations.
- There's a sharp peak near 0, then a smooth decline toward 1, indicating most scan durations are close to the minimum value.
- The KDE curve shows a quick drop-off after the peak, with a slow tapering toward the right.

4. Hypothesis Testing:

Perform hypothesis testing / visual analysis between :

a. actual_time aggregated value and OSRM time aggregated value.

In [57]:
```
trip_df[['actual_time','osrm_time']].describe()
```

Out[57]:

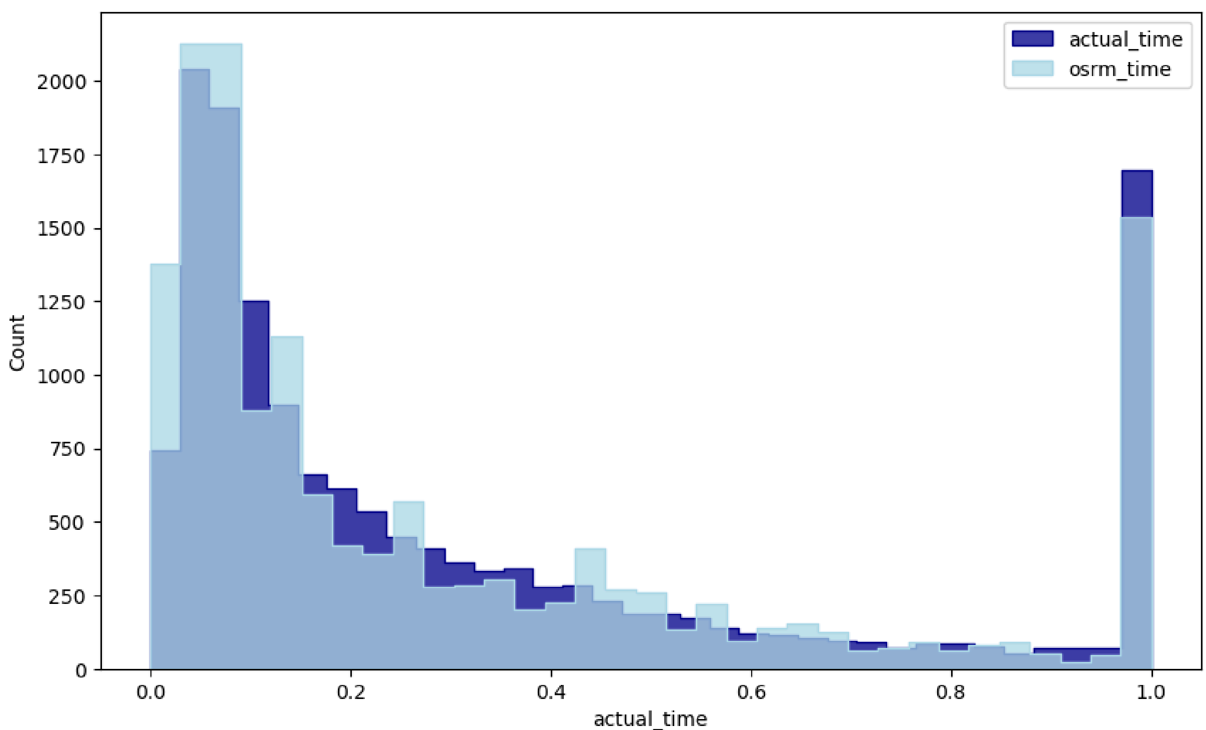
	actual_time	osrm_time
count	14817.000000	14817.000000
mean	0.312507	0.293861
std	0.319981	0.313924
min	0.000000	0.000000
25%	0.071122	0.062078
50%	0.171674	0.145749
75%	0.442673	0.437247
max	1.000000	1.000000

H0 : No difference between actual_time and osrm_time

Ha : Significant difference between actual_time and osrm_time

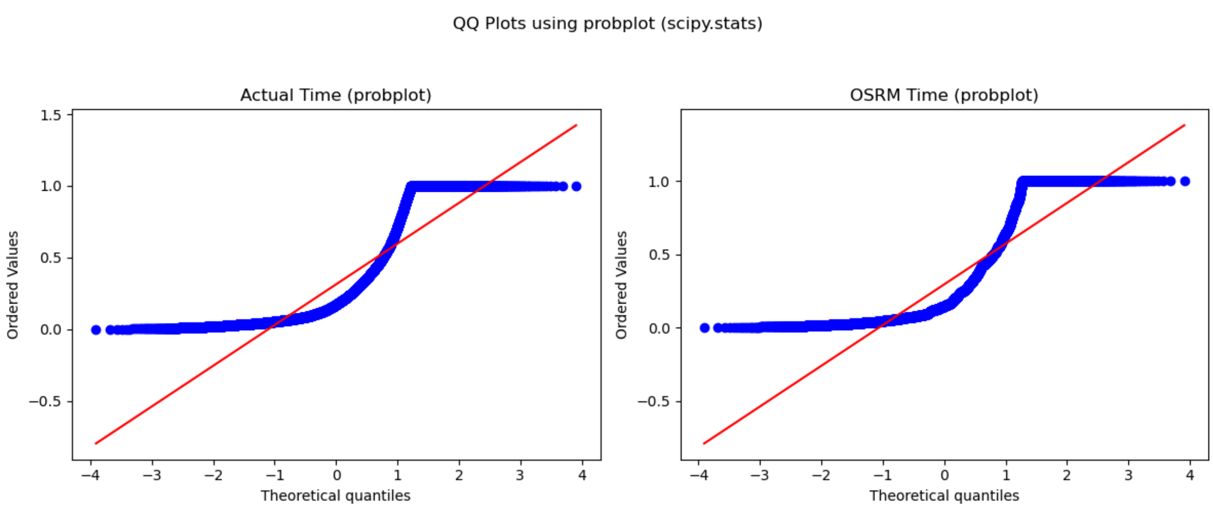
- check whether sample follows normal distribution or not

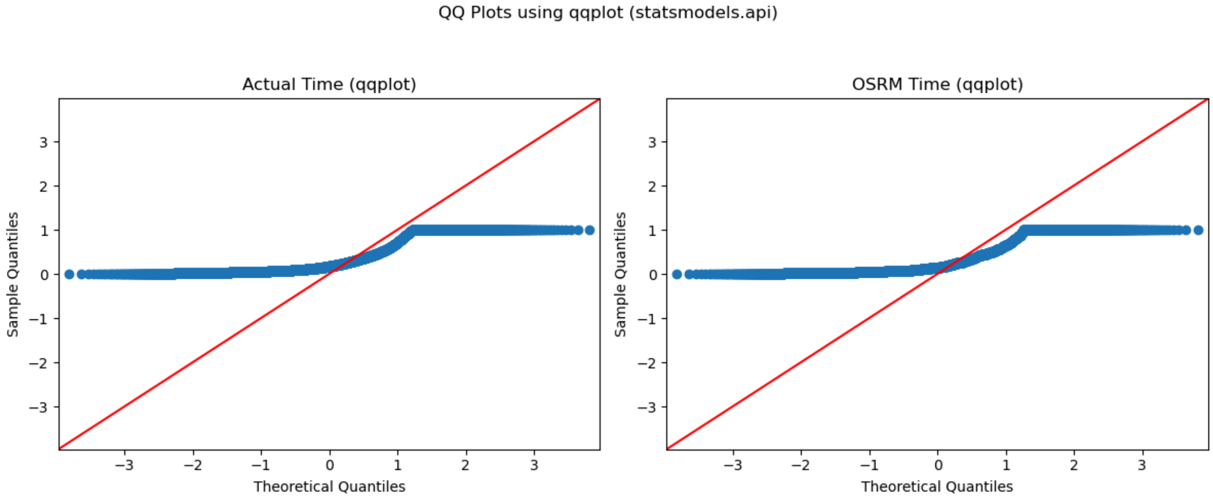
In [59]:
```
plt.figure(figsize =(10,6))
sns.histplot(trip_df['actual_time'],element = 'step',color ='darkblue')
sns.histplot(trip_df['osrm_time'],element = 'step',color ='lightblue')
plt.legend(['actual_time','osrm_time'])
plt.show()
```



QQ Plot

In [61]:
```python
plt.figure(figsize=(12, 5))
plt.suptitle('QQ Plots using probplot (scipy.stats)')

# Actual Time
ax1 = plt.subplot(1, 2, 1)
probplot(trip_df['actual_time'], dist='norm', plot=ax1)
ax1.set_title('Actual Time (probplot)')

# OSRM Time
ax2 = plt.subplot(1, 2, 2)
probplot(trip_df['osrm_time'], dist='norm', plot=ax2)
ax2.set_title('OSRM Time (probplot)')

plt.tight_layout(rect=[0, 0, 1, 0.93])
plt.show()


# ----------- QQPLOT from statsmodels -----------
plt.figure(figsize=(12, 5))
plt.suptitle('QQ Plots using qqplot (statsmodels.api)')

# Actual Time
ax3 = plt.subplot(1, 2, 1)
qqplot(trip_df['actual_time'], line='45', ax=ax3)
ax3.set_title('Actual Time (qqplot)')

# OSRM Time
ax4 = plt.subplot(1, 2, 2)
qqplot(trip_df['osrm_time'], line='45', ax=ax4)
ax4.set_title('OSRM Time (qqplot)')

plt.tight_layout(rect=[0, 0, 1, 0.93])
plt.show()
```





Both Actual Time and OSRM Time:

- Do not follow a normal distribution.
- Appear right-skewed (positive skew).
- May contain many identical or low values.
- Standard statistical tests that assume normality (like t-tests) may not be appropriate unless the data is transformed or non-parametric methods are used.

Shapiro-wilk test for normality:

- **H0:** Sample follows normal distribution
- **Ha:** Sample does not follow normal distribution
- Test statistics : Shapiro-wilk

Levene's test:

- **H0:** Variances are equal
- **Ha:** Unequal variances

In [63]:
```
stat_actual, p_actual = shapiro(trip_df['actual_time'])
stat_osrm, p_osrm = shapiro(trip_df['osrm_time'])
print("Shapiro-Wilk Test Results:")
print(f"Actual Time: Wstat = {stat_actual:.4f}, p-value = {p_actual:.4e}")
print(f"OSRM Time:  Wstat = {stat_osrm:.4f}, p-value = {p_osrm:.4e}")
if p_actual < 0.05:
    print("Actual Time is NOT normally distributed.")
else:
    print("Actual Time is likely normally distributed.")

if p_osrm < 0.05:
    print("OSRM Time is NOT normally distributed.")
else:
    print("OSRM Time is likely normally distributed.")
# Levene’s Test
print("Levene's test")
```

```python
stat, p = levene(trip_df['actual_time'], trip_df['osrm_time'])
print(f"Levene’s Test: stat = {stat:.4f}, p-value = {p:.4e}")
if p < 0.05:
    print("Unequal variances..")
else:
    print("Variances are equal...")
```

```
Shapiro-Wilk Test Results:
Actual Time: Wstat = 0.7888, p-value = 6.7488e-87
OSRM Time:  Wstat = 0.7824, p-value = 1.3679e-87
Actual Time is NOT normally distributed.
OSRM Time is NOT normally distributed.
Levene's test
Levene’s Test: stat = 6.3932, p-value = 1.1461e-02
Unequal variances..
```

Kolmogorov-Smirnov test (KS test)

In [65]:
```python
stat, p_value = ks_2samp(trip_df['actual_time'], trip_df['osrm_time'])
print("Two-sample Kolmogorov-Smirnov Test:")
print(f"KS statistic = {stat:.4f}")
print(f"p-value = {p_value:.4e}")

if p_value < 0.05:
    print("Reject H₀ → The two samples come from different distributions.")
else:
    print("Fail to reject H₀ → The two samples may come from the same distribution.
```

```
Two-sample Kolmogorov-Smirnov Test:
KS statistic = 0.0637
p-value = 1.3861e-26
Reject H₀ → The two samples come from different distributions.
```

Observations:

- As the samples do not exhibit a normal distribution, the application of the T-Test is not suitable in this context.
- Since p-value < alpha,it can be concluded that actual_time and osrm_time are not similar

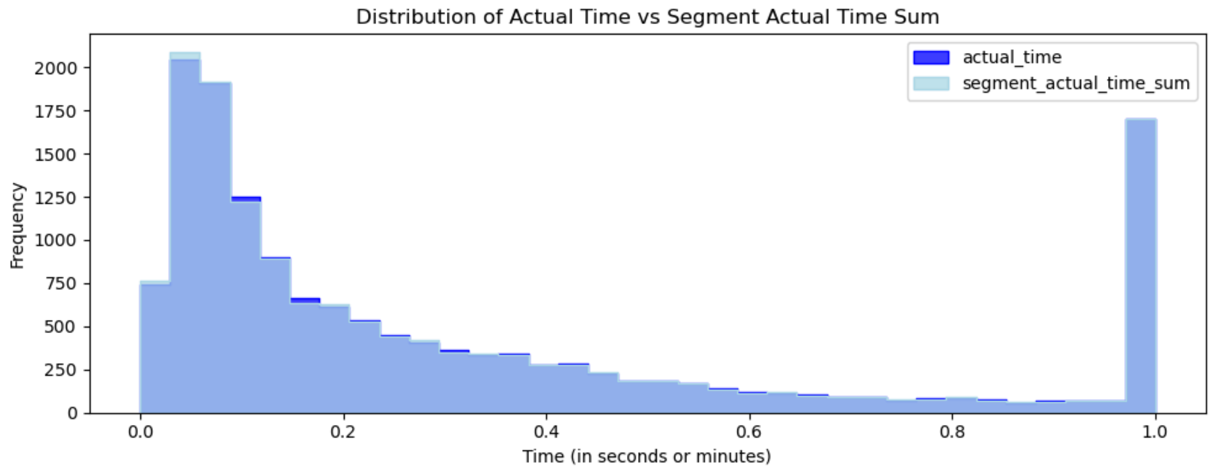
b. actual_time aggregated value and segment actual time aggregated value.

H0: No difference between actual_time and segment_actual_time.

Ha: significant difference between actual_time and segment_actual_time

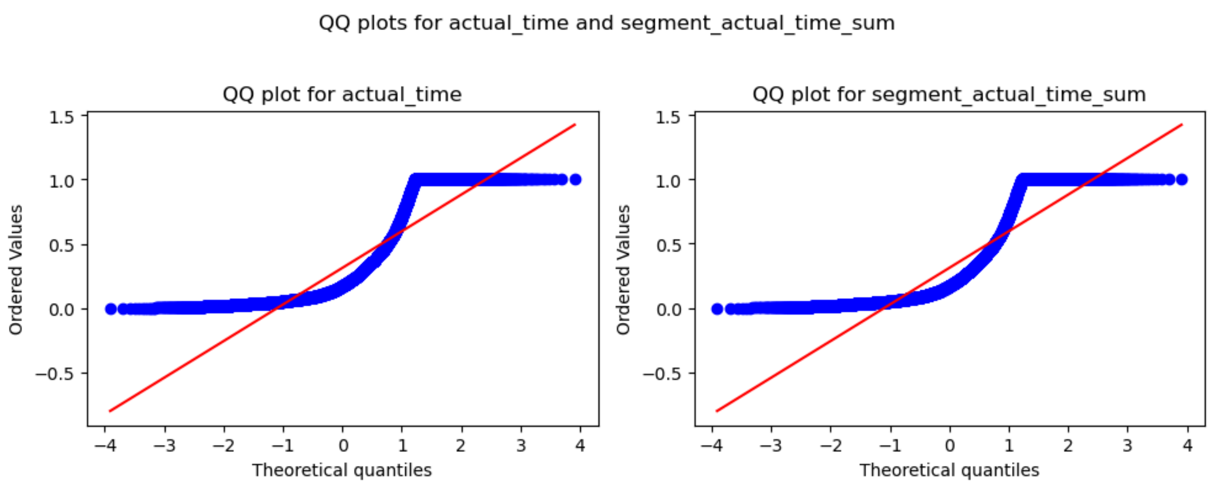
In [69]:
```python
plt.figure(figsize=(10, 4))
sns.histplot(trip_df['actual_time'], element='step', color='blue')
sns.histplot(trip_df['segment_actual_time_sum'], element='step', color='lightblue')
plt.legend(['actual_time', 'segment_actual_time_sum'])
plt.title("Distribution of Actual Time vs Segment Actual Time Sum")
plt.xlabel("Time (in seconds or minutes)")
plt.ylabel("Frequency")
```

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



QQ plot

In [71]:
```
plt.figure(figsize=(10, 4))
plt.suptitle('QQ plots for actual_time and segment_actual_time_sum')
ax1 = plt.subplot(1, 2, 1)
probplot(trip_df['actual_time'], plot=ax1, dist='norm')
ax1.set_title('QQ plot for actual_time')
ax2 = plt.subplot(1, 2, 2)
probplot(trip_df['segment_actual_time_sum'], plot=ax2, dist='norm')
ax2.set_title('QQ plot for segment_actual_time_sum')
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



Shapiro-wilk test for normality

- **H0:** Sample follows normal distribution
- **Ha:** Sample does not follow normal distribution
- Test statistics : Shapiro-wilk

Levene's test:

- **H0:** Variances are equal
- **Ha:** Unequal variances

In [73]:
```
for col in ['actual_time', 'segment_actual_time_sum']:
    sample = trip_df[col].dropna().sample(3000, random_state=1)
    stat, p_value = shapiro(sample)
    print(f"\nShapiro-Wilk Test for '{col}'")
    print(f"p-value = {p_value:.4f}")
    if p_value < 0.05:
        print("Reject H₀ : Not normally distributed")
    else:
        print("Fail to reject H₀ : Likely normally distributed")
print("\n")
print("LEVENE'S test:")
test_stat, p_value = levene(trip_df['actual_time'], trip_df['segment_actual_time_su
print('p-value:', p_value)
if p_value < 0.05:
    print('The samples do NOT have Homogeneous Variance')
else:
    print('The samples have Homogeneous Variance')
```

```
Shapiro-Wilk Test for 'actual_time'
p-value = 0.0000
Reject H₀ : Not normally distributed

Shapiro-Wilk Test for 'segment_actual_time_sum'
p-value = 0.0000
Reject H₀ : Not normally distributed


LEVENE'S test:
p-value: 0.9997804482446111
The samples have Homogeneous Variance
```

Kolmogorov-Smirnov test (KS test)

In [75]:
```
stat, p_value = ks_2samp(trip_df['actual_time'],trip_df['segment_actual_time_sum'])
print('p-value:', p_value)
if p_value < 0.05:
    print('The two samples come from different distributions')
else:
    print('The two samples come from the same distribution')
```

```
p-value: 8.641233525402584e-80
The two samples come from different distributions
```

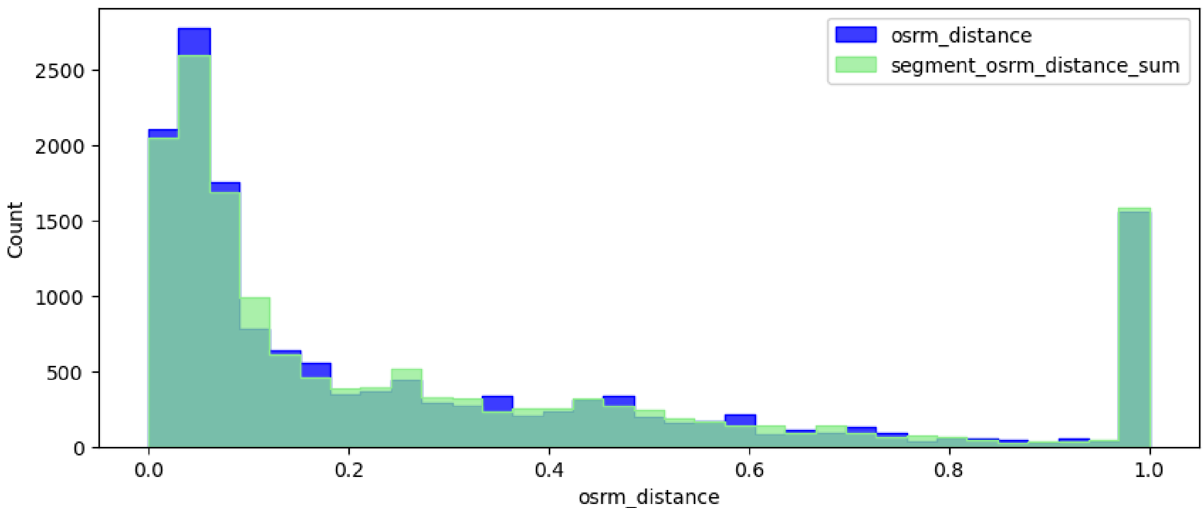
- Levene's test checks if two or more samples have equal variances.
- Here the samples don't follow normal distribution so T-test can't be applied here Kolmogorov-Smirnov Test (KS Test)
- Since p-value > alpha therfore it can be concluded that actual_time and segment_actual_time are similar

c. OSRM distance aggregated value and segment OSRM distance aggregated value.

H0: There is no difference between osrm distance and segment_osrm distance.

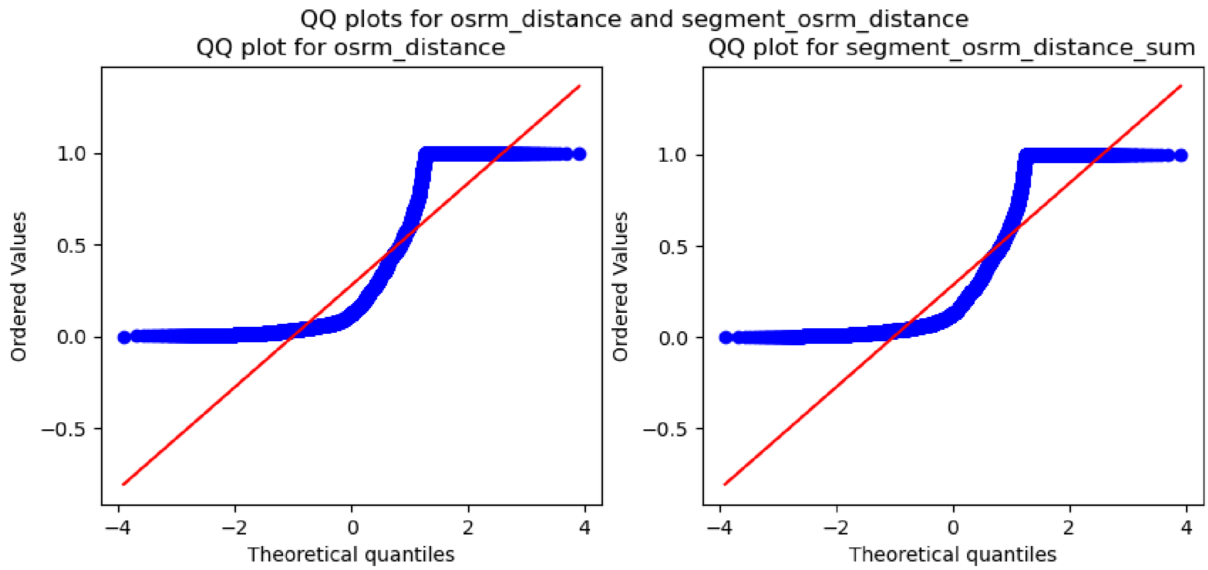
Ha: There is significant difference between osrm distance and segment_osrm distance.

In [79]:
```
plt.figure(figsize = (10, 4))
sns.histplot(trip_df['osrm_distance'], element = 'step', color = 'blue')
sns.histplot(trip_df['segment_osrm_distance_sum'], element = 'step', color = 'light
plt.legend(['osrm_distance', 'segment_osrm_distance_sum'])
plt.show()
```



QQ Plot

In [81]:
```
plt.figure(figsize = (10, 4))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for osrm_distance and segment_osrm_distance')
probplot(trip_df['osrm_distance'], plot = plt, dist = 'norm')
plt.title('QQ plot for osrm_distance')
plt.subplot(1, 2, 2)
probplot(trip_df['segment_osrm_distance_sum'], plot = plt, dist = 'norm')
plt.title('QQ plot for segment_osrm_distance_sum')
plt.show()
```



Shapiro-wilk test for normality

- **H0:** Sample follows normal distribution
- **Ha:** Sample does not follow normal distribution
- Test statistics : Shapiro-wilk

Levene's test:

- **H0:** Variances are equal
- **Ha:** Unequal variances

In [83]:
```
sample_osrm = trip_df['osrm_distance'].sample(3000, random_state=1)
stat, p_value = shapiro(sample_osrm)
print('osrm_distance p-value:', p_value)
if p_value < 0.05:
    print('The sample does NOT follow normal distribution')
else:
    print('The sample likely follows normal distribution')
print(" ")
sample_segment = trip_df['segment_osrm_distance_sum'].sample(3000, random_state=1)
stat, p_value = shapiro(sample_segment)
print('segment_osrm_distance_sum p-value:', p_value)
if p_value < 0.05:
    print('The sample does NOT follow normal distribution')
else:
    print('The sample likely follows normal distribution')
print(" ")
print("Levene's test")
test_stat, p_value = levene(trip_df['osrm_distance'], trip_df['segment_osrm_distanc
print('Levene’s test p-value:', p_value)

if p_value < 0.05:
    print('The samples do NOT have homogeneous variances')
else:
```

```
    print('The samples have homogeneous variances')
```

```
osrm_distance p-value: 3.70218184431216e-54
The sample does NOT follow normal distribution

segment_osrm_distance_sum p-value: 8.035321575539868e-54
The sample does NOT follow normal distribution

Levene's test
Levene’s test p-value: 0.6640699805647894
The samples have homogeneous variances
```

Kolmogorov–Smirnov (KS test):

In [85]:
```
stat, p_value = ks_2samp(trip_df['osrm_distance'], trip_df['segment_osrm_distance_s

print("KS 2-sample test statistic:", stat)
print("p-value:", p_value)

if p_value < 0.05:
    print("Reject H₀ :The two samples come from different distributions.")
else:
    print("Fail to reject H₀ : The two samples may come from the same distribution.
```

```
KS 2-sample test statistic: 0.03158534116217859
p-value: 7.436516334814034e-07
Reject H₀ :The two samples come from different distributions.
```

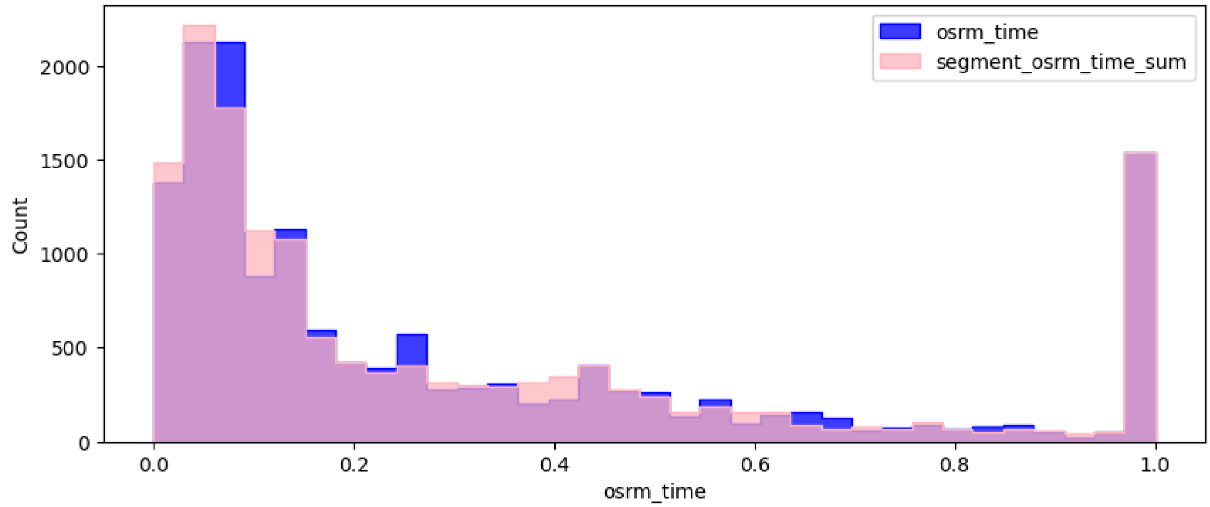
- Levene’s test checks if two or more samples have equal variances.
- Here the samples don't follow normal distribution so T-test can't be applied here Kolmogorov-Smirnov Test (KS Test)
- Since p-value > alpha therfore it can be concluded that actual_time and segment_actual_time are similar

d. OSRM time aggregated value and segment OSRM time aggregated value.

H0: There is no difference between osrm time and segment_osrm time.

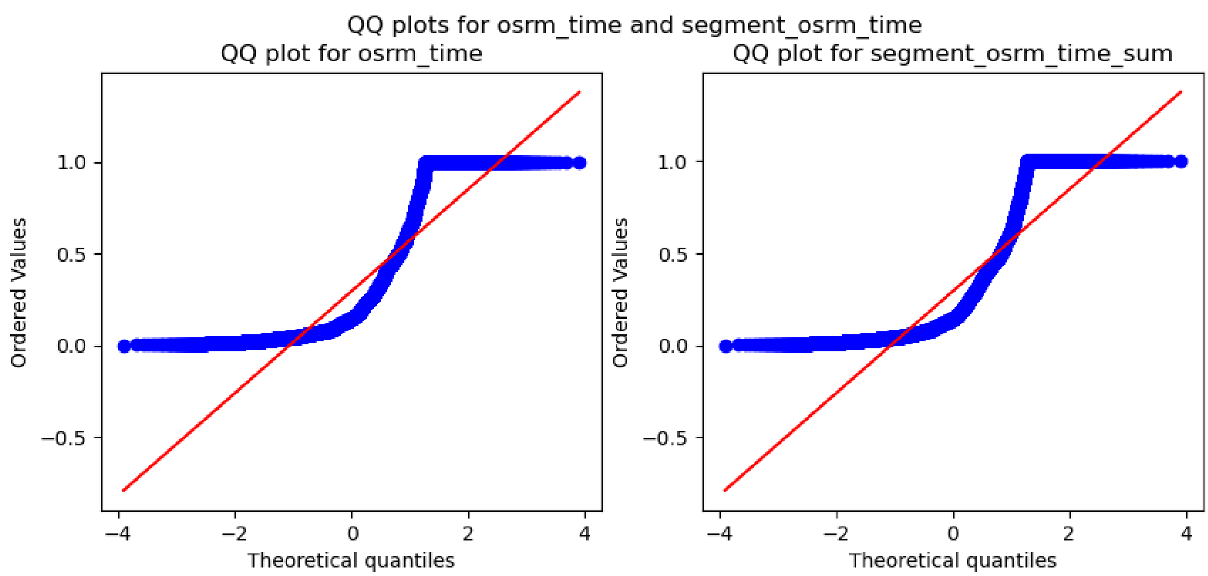
Ha: There is a significant difference between osrm time and segment_osrm time.

In [89]:
```
plt.figure(figsize=(10, 4))
sns.histplot(trip_df['osrm_time'], element='step', color='blue')
sns.histplot(trip_df['segment_osrm_time_sum'], element='step', color='lightpink')
plt.legend(['osrm_time', 'segment_osrm_time_sum'])
plt.show()
```



QQ Plot

In [91]:
```
plt.figure(figsize = (10, 4))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for osrm_time and segment_osrm_time')
probplot(trip_df['osrm_time'], plot = plt, dist = 'norm')
plt.title('QQ plot for osrm_time')
plt.subplot(1, 2, 2)
probplot(trip_df['segment_osrm_time_sum'], plot = plt, dist = 'norm')
plt.title('QQ plot for segment_osrm_time_sum')
plt.show()
```



Shapiro-wilk test for normality

- **H0:** Sample follows normal distribution
- **Ha:** Sample does not follow normal distribution
- Test statistics : Shapiro-wilk

Levene's test:

- **H0:** Variances are equal
- **Ha:** Unequal variances

In [93]:
```
test_stat, p_value = shapiro(trip_df['osrm_time'].sample(3000, random_state=1))
print('osrm_time p-value:', p_value)
if p_value < 0.05:
    print('The sample does NOT follow normal distribution')
else:
    print('The sample likely follows normal distribution')
print(" ")
test_stat, p_value = shapiro(trip_df['segment_osrm_time_sum'].sample(3000, random_s
print('segment_osrm_time p-value:', p_value)
if p_value < 0.05:
    print('The sample does NOT follow normal distribution')
else:
    print('The sample likely follows normal distribution')

print(" ")
print("Levene test")
test_stat, p_value = levene(trip_df['osrm_time'], trip_df['segment_osrm_time_sum'])
print('p-value:', p_value)
if p_value < 0.05:
    print('The samples do NOT have Homogeneous Variance')
else:
    print('The samples have Homogeneous Variance')
```

```
osrm_time p-value: 3.673886103390134e-53
The sample does NOT follow normal distribution

segment_osrm_time p-value: 6.359583763330919e-53
The sample does NOT follow normal distribution

Levene test
p-value: 0.8855462716900975
The samples have Homogeneous Variance
```

Kolmogorov-Smirnov Test (KS Test)

In [95]:
```
test_stat, p_value = ks_2samp(trip_df['osrm_time'], trip_df['segment_osrm_time_sum'
print("KS 2-sample test p-value:", p_value)
if p_value < 0.05:
    print("Reject H₀ : The two samples come from different distributions.")
else:
    print("Fail to reject H₀ : The two samples may come from the same distribution.
```

```
KS 2-sample test p-value: 5.09701578546931e-66
Reject H₀ : The two samples come from different distributions.
```

- Levene’s test checks if two or more samples have equal variances.
- Here the samples don't follow normal distribution so T-test can't be applied here Kolmogorov-Smirnov Test (KS Test)
- Since p-value > alpha therfore it can be concluded that actual_time and segment_actual_time are similar

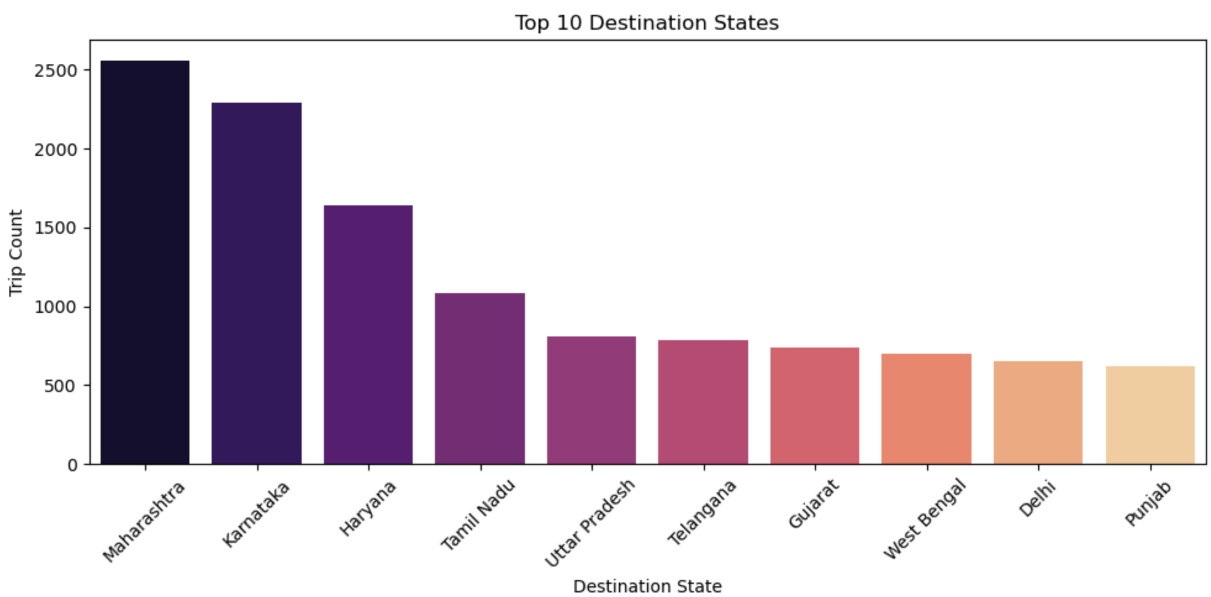
6. Business Insights & Recommendations

- Patterns observed in the data along with what you can infer from them.
- Check from where most orders are coming from (State, Corridor, etc.)
- Busiest corridor, avg distance between them, avg time taken, etc.
- Actionable items for the business.

6.1 Top 10 destination state

In [99]:
```
top_dest_state = trip_df['destination_state'].value_counts().head(10).reset_index()
top_dest_state.columns = ['Destination State', 'Trip Count']
print(top_dest_state.to_markdown(index=False))
plt.figure(figsize=(10, 5))
sns.barplot(x='Destination State', y='Trip Count', data=top_dest_state, palette='ma
plt.title('Top 10 Destination States')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
| Destination State   |   Trip Count |
|:--------------------|-------------:|
| Maharashtra         |         2561 |
| Karnataka           |         2294 |
| Haryana             |         1643 |
| Tamil Nadu          |         1084 |
| Uttar Pradesh       |          811 |
| Telangana           |          784 |
| Gujarat             |          734 |
| West Bengal         |          697 |
| Delhi               |          652 |
| Punjab              |          617 |
```

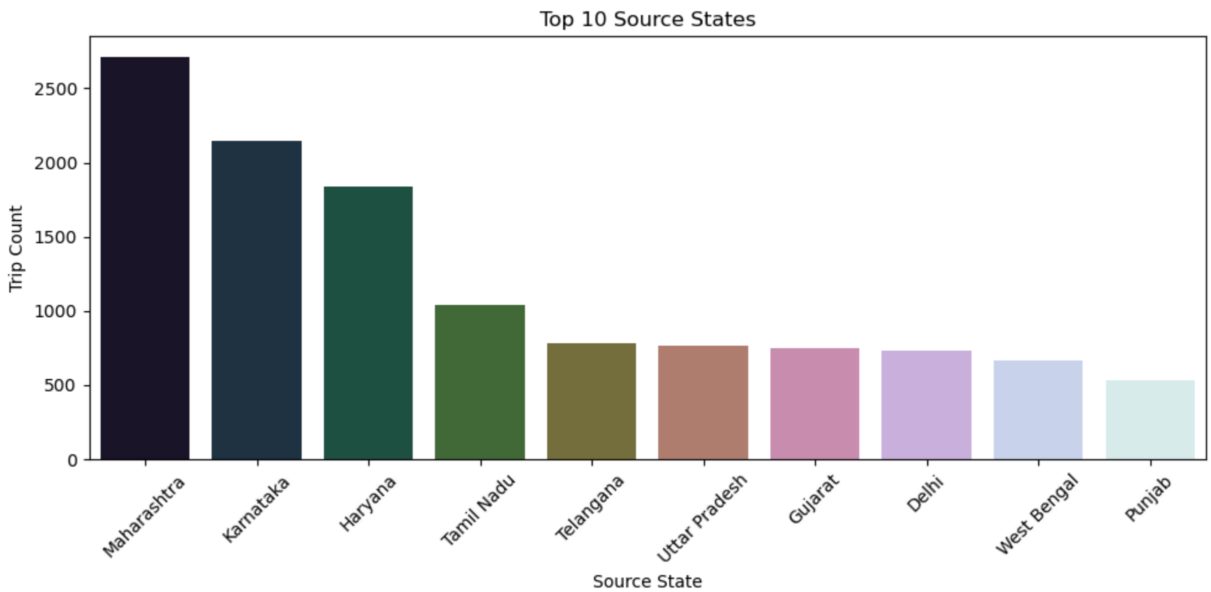


6.2 Top 10 source state

In [101…
```
top_src_state = trip_df['source_state'].value_counts().head(10).reset_index()
top_src_state.columns = ['Source State', 'Trip Count']
```

```
print(top_src_state.to_markdown(index=False))
plt.figure(figsize=(10, 5))
sns.barplot(x='Source State', y='Trip Count', data=top_src_state, palette='cubeheli
plt.title('Top 10 Source States')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
| Source State   |   Trip Count |
|:---------------|-------------:|
| Maharashtra    |         2714 |
| Karnataka      |         2143 |
| Haryana        |         1838 |
| Tamil Nadu     |         1039 |
| Telangana      |          781 |
| Uttar Pradesh  |          762 |
| Gujarat        |          750 |
| Delhi          |          728 |
| West Bengal    |          665 |
| Punjab         |          536 |
```

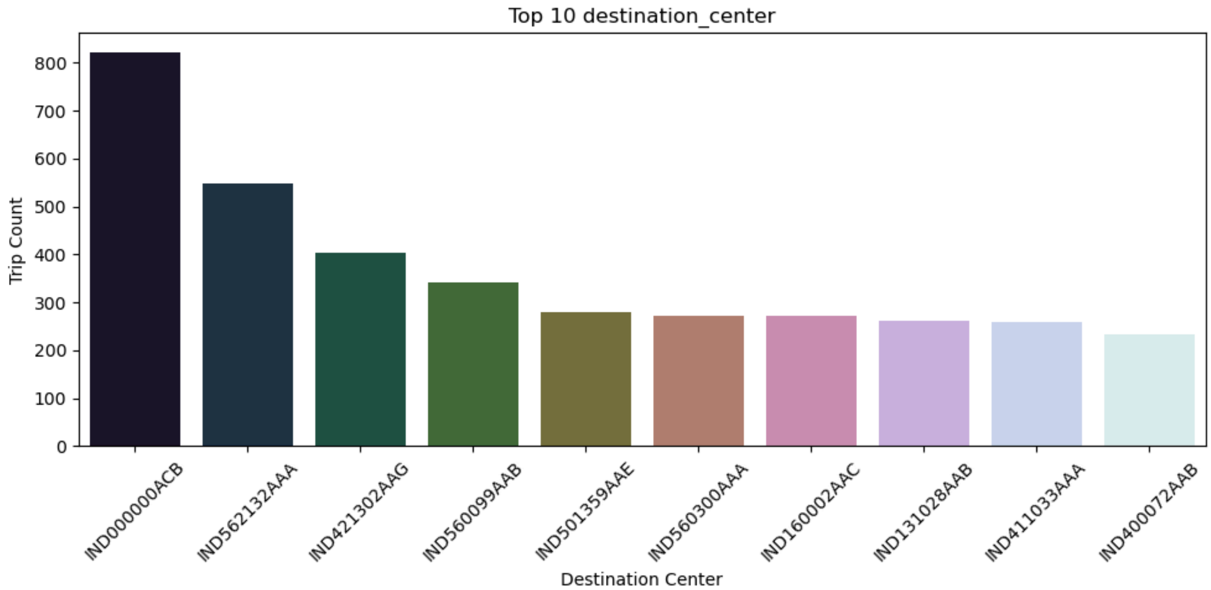


6.3 Top 10 destination_center

In [103…

```
# 5. destination_center
top_destination_center = trip_df['destination_center'].value_counts().head(10).rese
top_destination_center.columns = ['Destination Center', 'Trip Count']
print(top_destination_center.to_markdown(index=False))
plt.figure(figsize=(10, 5))
sns.barplot(x='Destination Center', y='Trip Count', data=top_destination_center, pa
plt.title('Top 10 destination_center')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
| Destination Center   |   Trip Count |
|:---------------------|-------------:|
| IND000000ACB         |          821 |
| IND562132AAA         |          548 |
| IND421302AAG         |          403 |
| IND560099AAB         |          342 |
| IND501359AAE         |          280 |
| IND560300AAA         |          272 |
| IND160002AAC         |          271 |
| IND131028AAB         |          262 |
| IND411033AAA         |          258 |
| IND400072AAB         |          234 |
```

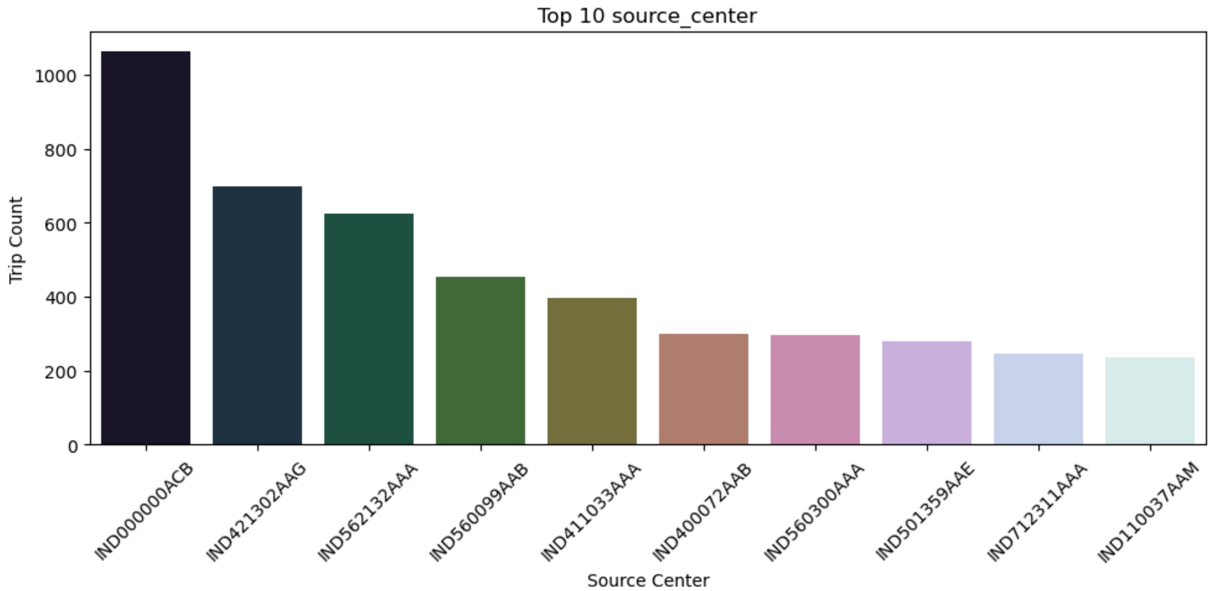


6.4 Top 10 Source centers

In [105…

```
top_source_center = trip_df['source_center'].value_counts().head(10).reset_index()
top_source_center.columns = ['Source Center', 'Trip Count']
print(top_source_center.to_markdown(index=False))

plt.figure(figsize=(10, 5))
sns.barplot(x='Source Center', y='Trip Count', data=top_source_center, palette='cub
plt.title('Top 10 source_center')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
| Source Center   |   Trip Count |
|:----------------|-------------:|
| IND000000ACB    |         1063 |
| IND421302AAG    |          697 |
| IND562132AAA    |          624 |
| IND560099AAB    |          455 |
| IND411033AAA    |          396 |
| IND400072AAB    |          300 |
| IND560300AAA    |          295 |
| IND501359AAE    |          278 |
| IND712311AAA    |          245 |
| IND110037AAM    |          237 |
```



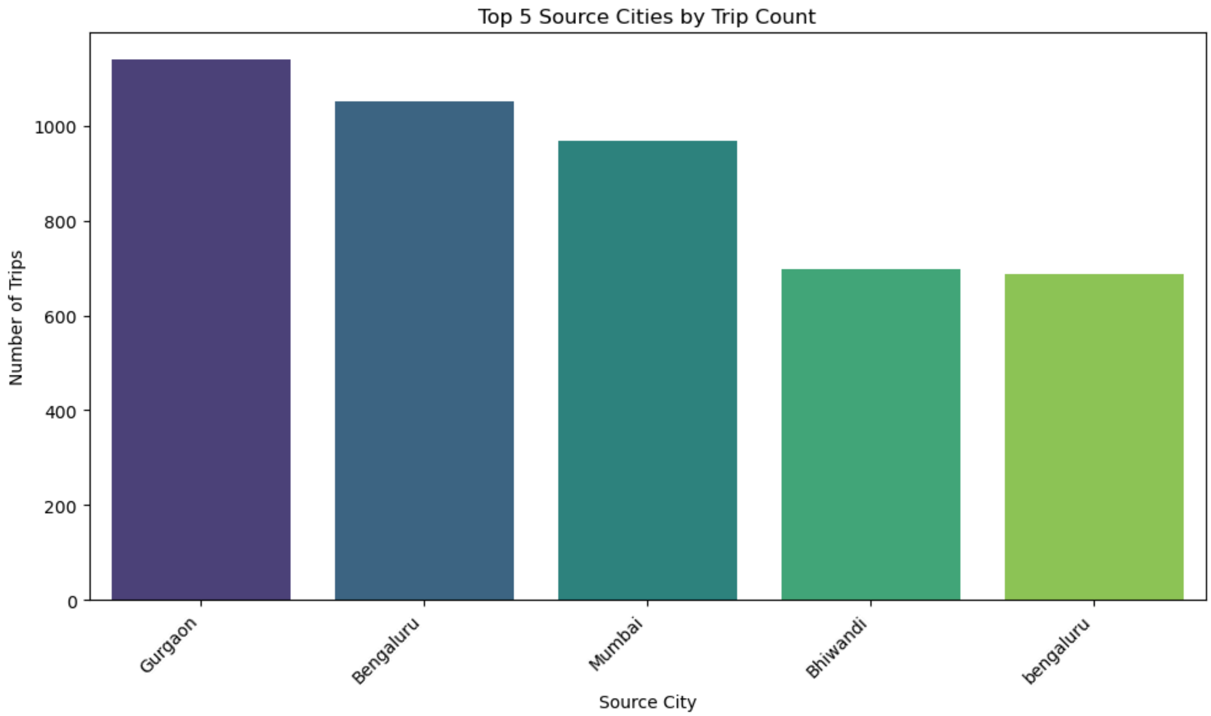
6.5 Top 5 source city and destination city

In [107…

```
print("\nTop 5 Source Cities by Trip Count:")
top5_source_cities = trip_df['source_city'].value_counts().head(5).reset_index()
top5_source_cities.columns = ['Source City', 'Trip Count']
print(top5_source_cities.to_markdown(index=False, numalign="left", stralign="left")

plt.figure(figsize=(10, 6))
sns.barplot(x='Source City', y='Trip Count', data=top5_source_cities, palette='viri
plt.title('Top 5 Source Cities by Trip Count')
plt.xlabel('Source City')
plt.ylabel('Number of Trips')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# 2. Top 5 Destination Cities
print("\nTop 5 Destination Cities by Trip Count:")
top5_destination_cities = trip_df['destination_city'].value_counts().head(5).reset_
top5_destination_cities.columns = ['Destination City', 'Trip Count']
print(top5_destination_cities.to_markdown(index=False, numalign="left", stralign="l

plt.figure(figsize=(10, 6))
sns.barplot(x='Destination City', y='Trip Count', data=top5_destination_cities, pal
plt.title('Top 5 Destination Cities by Trip Count')
plt.xlabel('Destination City')
plt.ylabel('Number of Trips')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

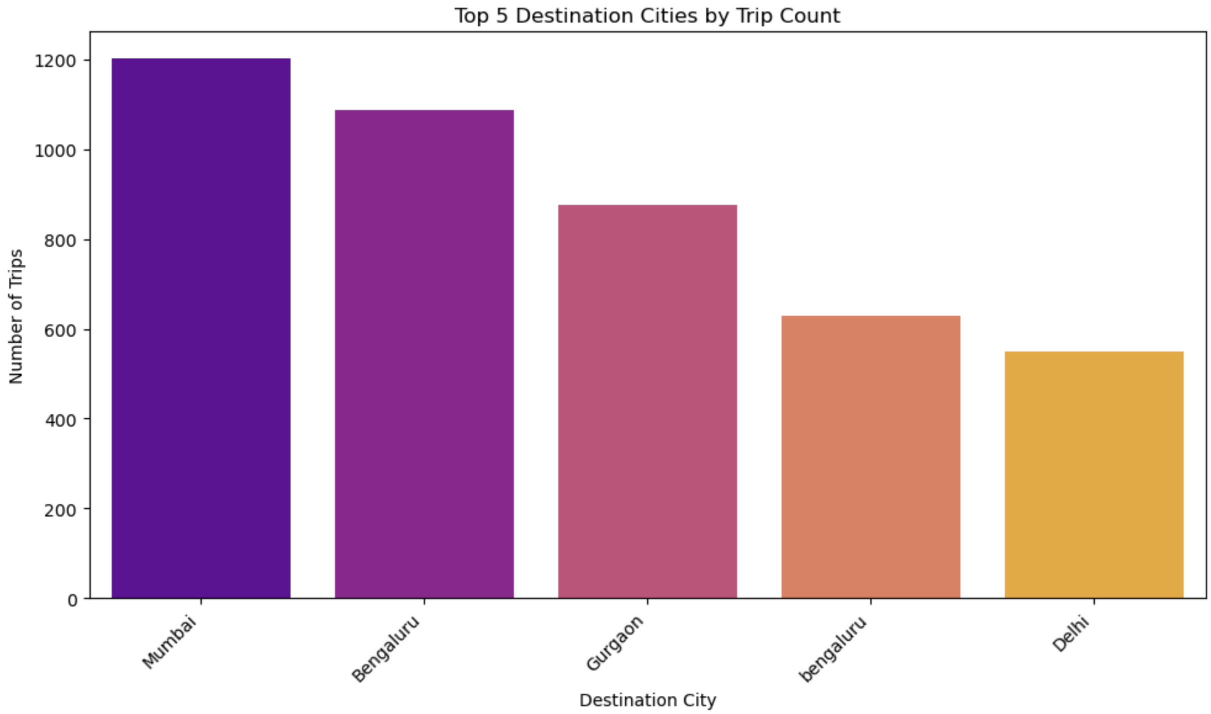
Top 5 Source Cities by Trip Count:

Source City	Trip Count
Gurgaon	1139
Bengaluru	1052
Mumbai	968
Bhiwandi	697
bengaluru	688



Top 5 Destination Cities by Trip Count:

Destination City	Trip Count
Mumbai	1202
Bengaluru	1088
Gurgaon	877
bengaluru	629
Delhi	549

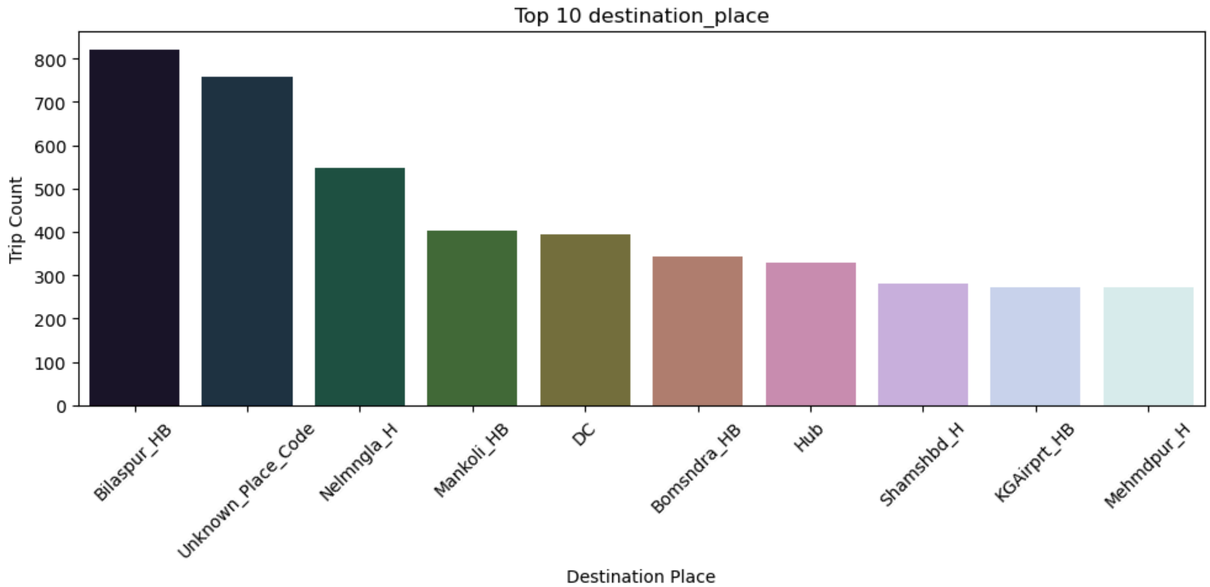


6.6 Top 10 destination place

In [109…

```
top_destination_place = trip_df['destination_place'].value_counts().head(10).reset_
top_destination_place.columns = ['Destination Place', 'Trip Count']
print(top_destination_place.to_markdown(index=False))

plt.figure(figsize=(10, 5))
sns.barplot(x='Destination Place', y='Trip Count', data=top_destination_place, pale
plt.title('Top 10 destination_place')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
| Destination Place   |   Trip Count |
|:--------------------|-------------:|
| Bilaspur_HB         |          821 |
| Unknown_Place_Code  |          757 |
| Nelmngla_H          |          548 |
| Mankoli_HB          |          403 |
| DC                  |          394 |
| Bomsndra_HB         |          342 |
| Hub                 |          330 |
| Shamshbd_H          |          280 |
| KGAirprt_HB         |          272 |
| Mehmdpur_H          |          271 |
```

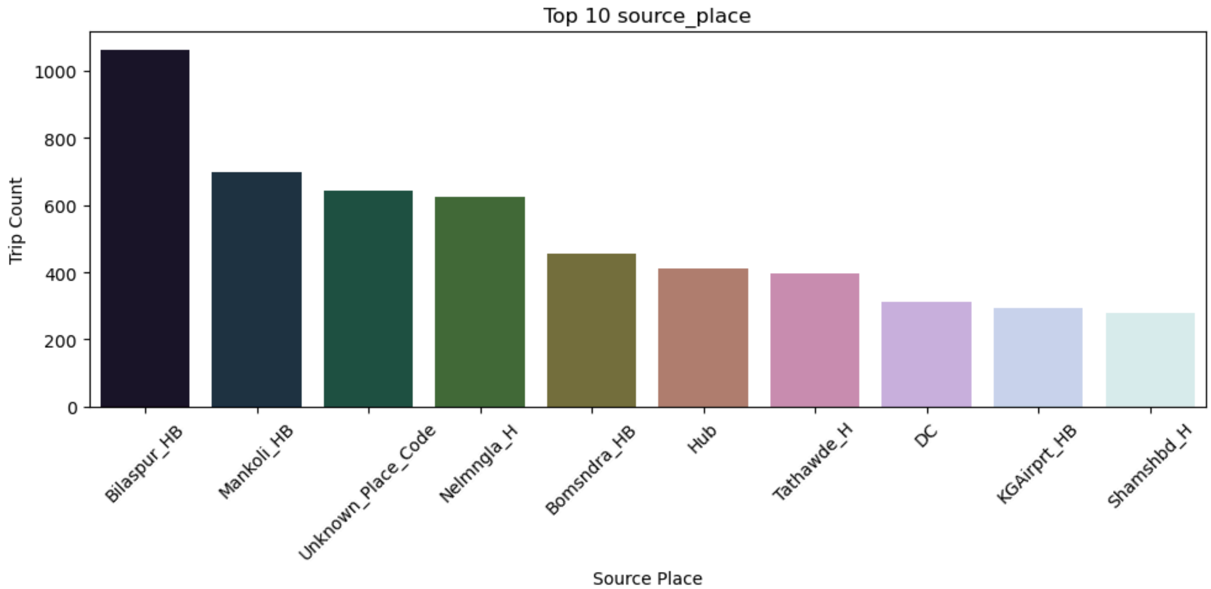


6.7 Top 10 Source place

In [111…

```
top_source_place = trip_df['source_place'].value_counts().head(10).reset_index()
top_source_place.columns = ['Source Place', 'Trip Count']
print(top_source_place.to_markdown(index=False))

plt.figure(figsize=(10, 5))
sns.barplot(x='Source Place', y='Trip Count', data=top_source_place, palette='cubeh
plt.title('Top 10 source_place')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

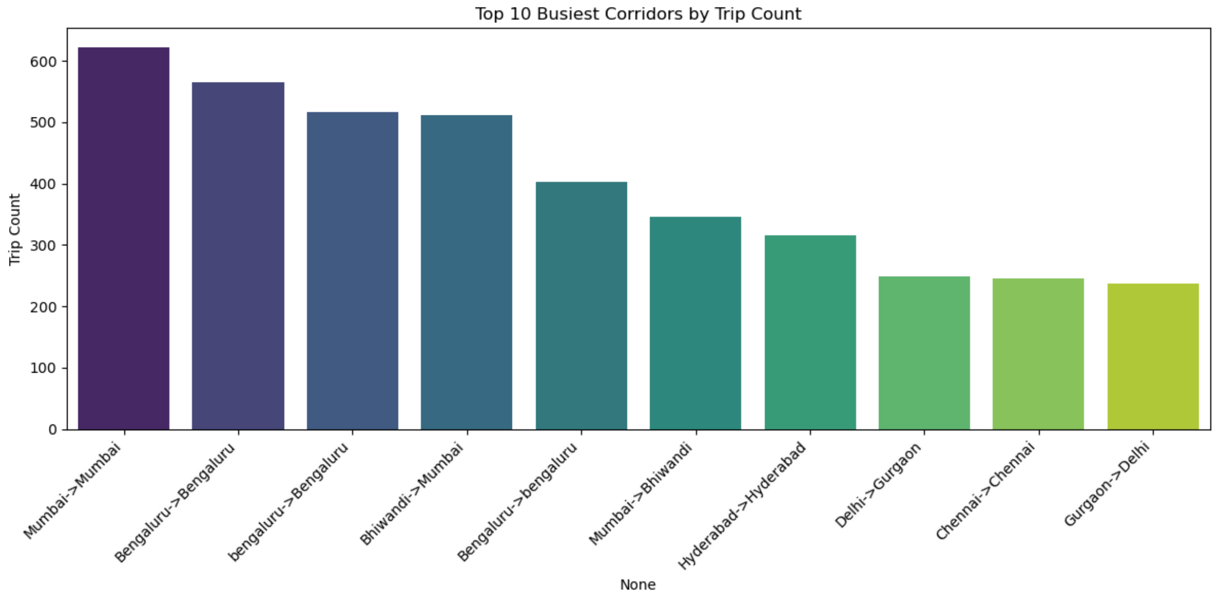
```
| Source Place       |   Trip Count |
|:-------------------|-------------:|
| Bilaspur_HB        |         1063 |
| Mankoli_HB         |          697 |
| Unknown_Place_Code |          642 |
| Nelmngla_H         |          624 |
| Bomsndra_HB        |          455 |
| Hub                |          412 |
| Tathawde_H         |          396 |
| DC                 |          311 |
| KGAirprt_HB        |          295 |
| Shamshbd_H         |          278 |
```



6.8 Top 10 Busiest corridors

In [113…

```
corridor_data = trip.groupby(['source_city', 'destination_city']).agg({
    'trip_uuid': 'count',
    'actual_distance_to_destination': 'mean',
    'actual_time': 'mean',
    'osrm_time': 'mean',
    'segment_osrm_time_sum': 'mean',
    'segment_actual_time_sum': 'mean'
}).reset_index().rename(columns={'trip_uuid': 'trip_count'})
busiest_corridors = corridor_data.sort_values(by='trip_count', ascending=False).hea

plt.figure(figsize=(12, 6))
sns.barplot( data=busiest_corridors, x=busiest_corridors.index, y='trip_count', pal
plt.xticks( busiest_corridors.index,labels=busiest_corridors.apply(lambda x: f"{x['
plt.ylabel('Trip Count')
plt.title('Top 10 Busiest Corridors by Trip Count')
plt.tight_layout()
plt.show()
```



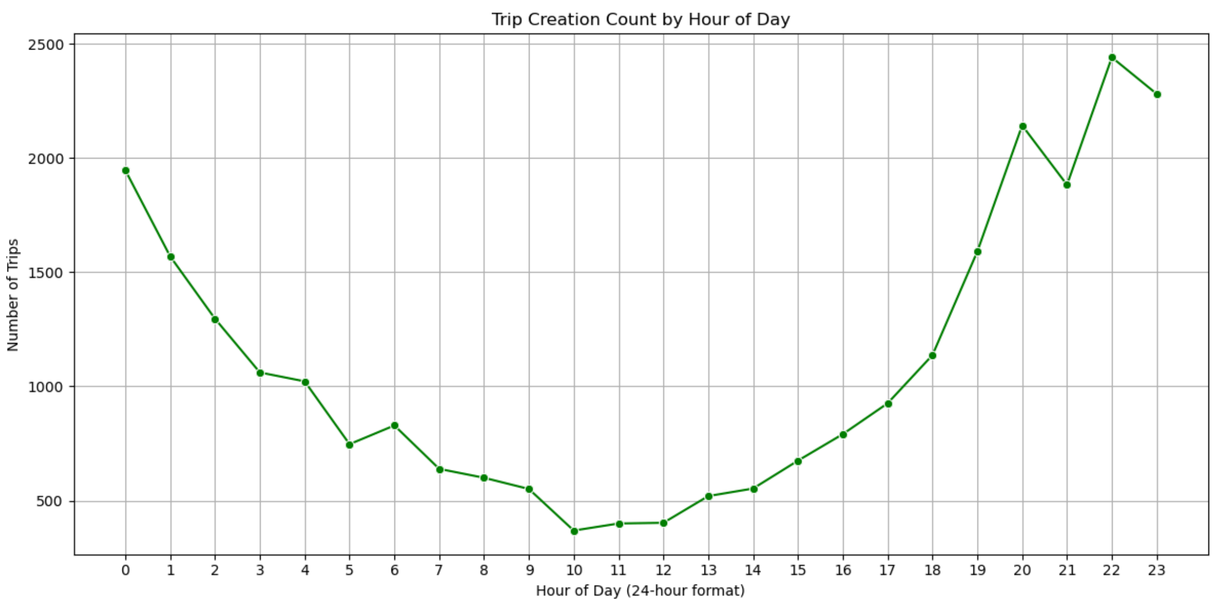
6.8 Trips in Hour (Trip Creation Hour)

In [115…

```
print("\nTrip Count by Hour of Day:")
trips_by_hour = trip['trip_creation_hr'].value_counts().sort_index().reset_index()
trips_by_hour.columns = ['Hour', 'Trip Count']
print(trips_by_hour.to_markdown(index=False, numalign="left", stralign="left"))

plt.figure(figsize=(12, 6))
sns.lineplot(x='Hour', y='Trip Count', data=trips_by_hour, marker='o', color='green
plt.title('Trip Creation Count by Hour of Day')
plt.xlabel('Hour of Day (24-hour format)')
plt.ylabel('Number of Trips')
plt.xticks(range(0, 24))
plt.grid(True)
plt.tight_layout()
plt.show()
```

Trip Count by Hour of Day:

Hour	Trip Count
0	1946
1	1569
2	1296
3	1061
4	1022
5	747
6	829
7	639
8	600
9	551
10	369
11	400
12	403
13	520
14	553
15	675
16	791
17	926
18	1137
19	1590
20	2142
21	1882
22	2440
23	2280



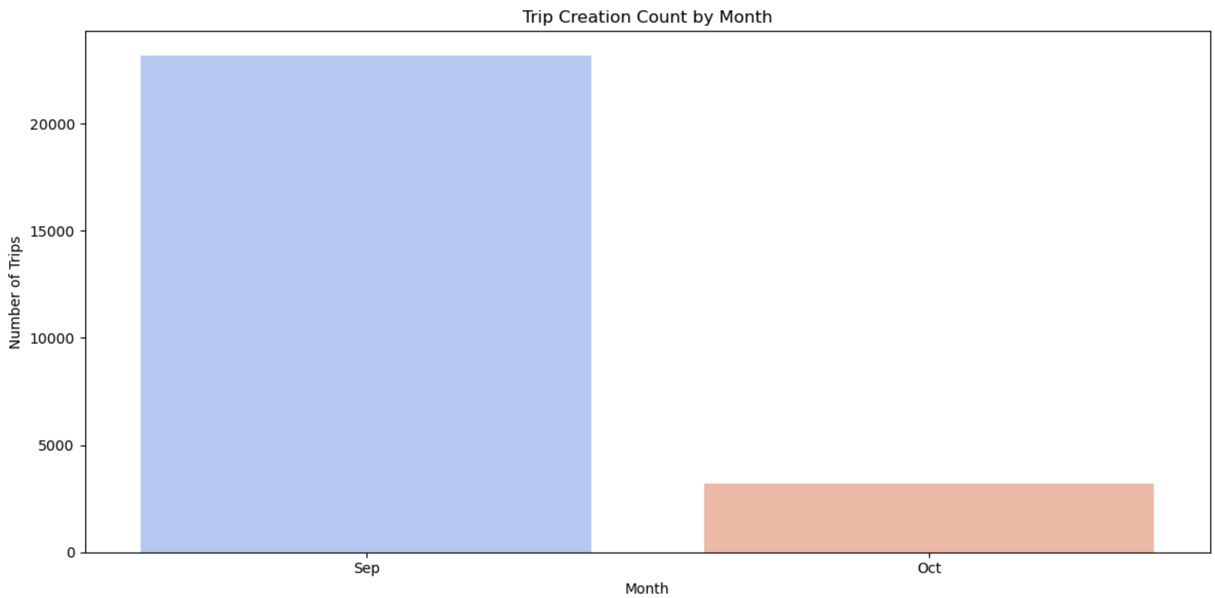
6.9 Top trips of Month (Trip Creation Month)

In [117…

```
print("\nTrip Count by Month:")
trips_by_month = trip['trip_mnth'].value_counts().sort_index().reset_index()
trips_by_month.columns = ['Month', 'Trip Count']
# Map month numbers to names for better readability
month_names = {
    1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr', 5: 'May', 6: 'Jun',
    7: 'Jul', 8: 'Aug', 9: 'Sep', 10: 'Oct', 11: 'Nov', 12: 'Dec'
}
```

```
trips_by_month['Month Name'] = trips_by_month['Month'].map(month_names)
print(trips_by_month[['Month Name', 'Trip Count']].to_markdown(index=False, numalig

plt.figure(figsize=(12, 6))
sns.barplot(x='Month Name', y='Trip Count', data=trips_by_month, palette='coolwarm'
plt.title('Trip Creation Count by Month')
plt.xlabel('Month')
plt.ylabel('Number of Trips')
plt.tight_layout()
plt.show()
```

```
Trip Count by Month:
| Month Name   | Trip Count   |
|:-------------|:-------------|
| Sep          | 23159        |
| Oct          | 3209         |
```



Insights:

- Our data shows a clear and consistent difference between real-world travel times and distances versus what the routing engine (OSRM) estimates.
- The actual trips are generally taking longer and covering slightly different distances than predicted. This means the current routing estimates are too optimistic and don't fully reflect real operational conditions, which can lead to delays and impact customer expectations.
- When we compare different route types, Carting routes stand out. They tend to be less efficient, with actual travel times much longer compared to the OSRM estimates, and lower average speeds.
- This makes sense since Carting often involves more stops and complex urban navigation, leading to greater delays than the Full Truck Load (FTL) routes. So, Carting is the area where improvements can really move the needle.
- The way we aggregate segment-level data into trip-level metrics is accurate.
- Summing up segment times and distances closely matches the overall trip numbers, so we can trust the underlying data and calculations.

- Though we didn’t dive deeply into it here, features like time of day, day of week, and origin/destination states and cities likely play a big role in trip performance.
- Peak hours or certain locations probably cause more delays due to traffic or operational hurdles.

Recommendations

- Fix Carting Route Inefficiencies First
- Integrate live traffic data, historical patterns, and weather info to improve routing estimates.
- Build a model that tweaks OSRM’s baseline with these factors, so ETAs are more accurate and reflective of what drivers actually experience.
- Analyze delays and inefficiencies by specific cities, states, and times to spot hotspots or recurring issues.
- Set up alerts or reports for trips with extreme travel times or distances. Find out if these are due to real incidents or data errors, and take steps to fix systemic issues or improve data collection
- Build a Real-Time Performance Dashboard

In [ ]: