Report: NYC Taxi Operations

Include your visualisations, analysis, results, insights, and outcomes. Explain your methodology and approach to the tasks. Add your conclusions to the sections.

1. Data Preparation

1.1 Loading the Dataset

- The warnings package was downloaded and essential libraries like NumPy, Pandas, Matplotlib, and Seaborn were imported to support the EDA process.
- The pyarrow package was installed to enable the loading of Parquet files, as the provided data was in Parquet format.
- A sample Parquet file was loaded initially to verify that the setup was working correctly. Basic checks such as head(), info(), and tail() were performed to validate the data.
- Upon successful validation, the process proceeded to load and sample data from all the available monthly files.

1.1.1 Sample the Data and Combine the Files

- As outlined in the process, sampling was performed on the *tpep_pickup_datetime* field, taking a 5% sample from each month's data.
- All Parquet files were looped through, and new hour and date columns were created from the *tpep_pickup_datetime* field to facilitate structured 5% sampling.
- The sampled data from each file was combined into a single DataFrame using the *pd.concat()* method.
- Basic verification of the consolidated DataFrame was carried out using methods such as info().
- The final sampled DataFrame was exported into a Parquet file using the df.to_parquet() method for future reference.

2. Data Cleaning

2.1 Fixing Columns

- The sampled Parquet file was imported to proceed with the EDA process.
- The df.info() method was executed to inspect the columns and their data types. Based on the initial observation, the columns appeared consistent with appropriate data types and no immediate issues were detected.

2.1.1 Fix the Index

- The index was reset using df.reset_index(drop=True, inplace=True), and the columns were inspected. During this inspection, it was found that there were two columns related to airport_fee.

df.reset_index(drop=True, inplace=True)
df.columns

Index(['VendorID', 'tpep_pickup_datetime', 'tpep_dropoff_datetime',
'passenger_count', 'trip_distance', 'RatecodeID', 'store_and_fwd_flag',
'PULocationID', 'DOLocationID', 'payment_type', 'fare_amount', 'extra',
'mta_tax', 'tip_amount', 'tolls_amount', 'improvement_surcharge',
'total_amount', 'congestion_surcharge', 'airport_fee', 'date', 'hour',
'Airport_fee'],
dtype='object'

- Dropped the columns date, hour, and store_and_fwd_flag.
- The date and hour columns were created during the sampling process and can be added again later if needed.
- The store_and_fwd_flag column was dropped as it was deemed unnecessary for the EDA process based on the data dictionary.

2.1.2 Combine the Two airport_fee Columns

- Since there were two *airport_fee* columns, they were merged into one.
- The duplicate *Airport_fee* column was dropped after merging to avoid redundancy.

df.airport_fee = df.airport_fee.fillna(df.Airport_fee)

Columns after dropping Airport_fee
Index(['VendorID', 'tpep_pickup_datetime', 'tpep_dropoff_datetime',
'passenger_count', 'trip_distance', 'RatecodeID', 'PULocationID',
'DOLocationID', 'payment_type', 'fare_amount', 'extra', 'mta_tax',
'tip_amount', 'tolls_amount', 'improvement_surcharge', 'total_amount',
'congestion_surcharge', 'airport_fee'],
dtype='object')

2.1.3 Fix Column with Negative Values

- The command countindex = df['fare_amount'].value_counts() counts the occurrences of each unique value in the fare_amount column and stores it in the variable countindex.
- The command df = df[df['RatecodeID'] != 99] filters the dataframe by removing all rows where the RatecodeID is equal to 99, retaining only valid rate code entries.
- The following data cleaning steps were performed to prepare the dataset for analysis:
  - Replaced missing or zero values in the extra, mta_tax, improvement_surcharge, total_amount, congestion_surcharge, and airport_fee columns using appropriate imputation techniques such as forward-filling or filling with mean/median values.
  - Dropped rows with missing values in critical fields like fare_amount, tip_amount, or passenger_count to ensure all trips had valid data.

- Removed invalid entries where the RatecodeID was 99, representing unknown or unassigned rate codes.
- Verified that no missing or zero values remain in key monetary columns (fare_amount, tip_amount, total_amount, etc.).
- Replaced remaining missing values with meaningful imputations or dropped the corresponding rows where necessary.

2.2 Handling Missing Values

2.2.1 Proportion of Missing Values

- The command df.isna().sum() was executed to verify if any missing values remained in the dataset after completing the data cleaning and imputation steps.

```
[192]: # Find the proportion of missing values in each column
       df.isna().sum()
```

```
[192]: VendorID                  0
       tpep_pickup_datetime      0
       tpep_dropoff_datetime     0
       passenger_count       64874
       trip_distance             0
       RatecodeID            64874
       PULocationID              0
       DOLocationID              0
       payment_type              0
       fare_amount               0
       extra                     0
       mta_tax                   0
       tip_amount                0
       tolls_amount              0
       improvement_surcharge     0
       total_amount              0
       congestion_surcharge  64874
       airport_fee           64874
       dtype: int64
```

2.2.2 Handling Missing Values in passenger_count

- The median value of the "passenger_count" column was calculated using df.passenger_count.median(), and the missing values in the "passenger_count" column were replaced with this median value using df.loc[df['passenger_count'].isna(), 'passenger_count'] = passenger_countMedian.

```
[196]: df.loc[df['passenger_count'].isna() , 'passenger_count'] = passenger_countMedian
```

```
[198]: df.isna().sum()
```

```
[198]: VendorID                  0
       tpep_pickup_datetime      0
       tpep_dropoff_datetime     0
       passenger_count           0
       trip_distance             0
       RatecodeID            64874
       PULocationID              0
       DOLocationID              0
       payment_type              0
       fare_amount               0
       extra                     0
       mta_tax                   0
       tip_amount                0
       tolls_amount              0
       improvement_surcharge     0
       total_amount              0
       congestion_surcharge  64874
       airport_fee           64874
       dtype: int64
```

- Zeros in the "***passenger_count***" column were identified and replaced with the median value to maintain data consistency and accuracy.

```
[200]: (df["passenger_count"] == 0).sum()
```

```
[200]: 29646
```

Did you find zeroes in passenger_count? Handle these.

```
[202]: df.loc[df['passenger_count'] == 0 , 'passenger_count'] = passenger_countMedian
```

2.2.3 Handle Missing Values in RatecodeID

- The mode of the "RatecodeID" column was calculated using df.RatecodeID.mode()[0], and the most frequent value was stored in a variable named RatecodeIDmedian to address missing or inconsistent entries.
- Missing values in the "RatecodeID" column were replaced with the most frequent value (mode) by using df.loc[df['RatecodeID'].isna(), 'RatecodeID'] = RatecodeIDmedian.

```
[208]: df.isna().sum()
```

```
[208]: VendorID                  0
       tpep_pickup_datetime      0
       tpep_dropoff_datetime     0
       passenger_count           0
       trip_distance             0
       RatecodeID                0
       PULocationID              0
       DOLocationID              0
       payment_type              0
       fare_amount               0
       extra                     0
       mta_tax                   0
       tip_amount                0
       tolls_amount              0
       improvement_surcharge     0
       total_amount              0
       congestion_surcharge    64874
       airport_fee             64874
       dtype: int64
```

2.2.4 Impute NaN in congestion_surcharge

- The missing values in the "congestion_surcharge" column were handled by calculating its median and replacing all NaN entries with this median value.
- The NaN values in the "airport_fee" column were identified and replaced with the median value of the "airport_fee" to ensure consistency.
- After reviewing all columns, NaN values were replaced with the appropriate median or mode values for each respective column to ensure data completeness and consistency.

```
df.isna().sum()
```

```
VendorID                0
tpep_pickup_datetime    0
tpep_dropoff_datetime   0
passenger_count         0
trip_distance           0
RatecodeID              0
PULocationID            0
DOLocationID            0
payment_type            0
fare_amount             0
extra                   0
mta_tax                 0
tip_amount              0
tolls_amount            0
improvement_surcharge   0
total_amount            0
congestion_surcharge    0
airport_fee             0
dtype: int64
```

2.3 Handling Outliers and Standardising Values

- After handling missing values, outlier treatment was carried out across the entire dataframe to further enhance data quality and reliability.
- A statistical summary of the entire dataframe was generated using df.describe().T to understand the central tendencies, dispersion, and identify potential outliers.

	count	mean	min	25%	50%	75%	max	std
VendorID	1885928.0	1.737094	1.0	1.0	2.0	2.0	6.0	0.445528
tpep_pickup_datetime	1885928	2023-07-02 18:13:51.878698	2022-12-31 23:51:30	2023-04-02 14:52:01	2023-06-27 13:51:23	2023-10-06 18:48:58	2023-12-31 23:57:51	NaN
tpep_dropoff_datetime	1885928	2023-07-02 18:31:10.158788	2022-12-31 23:56:06	2023-04-02 15:14:01.500000	2023-06-27 14:12:20.500000	2023-10-06 19:06:33.500000	2024-01-01 20:50:55	NaN
passenger_count	1885928.0	1.374293	1.0	1.0	1.0	1.0	9.0	0.866345
trip_distance	1885928.0	3.807185	0.0	1.04	1.79	3.39	126360.46	122.756054
RatecodeID	1885928.0	1.07222	1.0	1.0	1.0	1.0	6.0	0.388271
PULocationID	1885928.0	165.4853	1.0	132.0	162.0	234.0	265.0	63.879862
DOLocationID	1885928.0	164.241483	1.0	114.0	162.0	234.0	265.0	69.727039
payment_type	1885928.0	1.16472	0.0	1.0	1.0	1.0	4.0	0.509392
fare_amount	1885928.0	19.846322	0.0	9.3	13.5	21.9	143163.45	105.818594
extra	1885928.0	1.596819	0.0	0.0	1.0	2.5	20.8	1.830429
mta_tax	1885928.0	0.495305	0.0	0.5	0.5	0.5	4.0	0.048498
tip_amount	1885928.0	3.566655	0.0	1.0	2.86	4.45	223.08	4.057414
tolls_amount	1885928.0	0.590709	0.0	0.0	0.0	0.0	143.0	2.177288
improvement_surcharge	1885928.0	0.999077	0.0	1.0	1.0	1.0	1.0	0.027915
total_amount	1885928.0	28.942678	0.0	15.96	21.0	30.72	143167.45	106.701573
congestion_surcharge	1885928.0	2.327101	0.0	2.5	2.5	2.5	2.5	0.634313
airport_fee	1885928.0	0.138788	0.0	0.0	0.0	0.0	1.75	0.458715

2.3.1 Outliers in payment_type, trip_distance and tip_amount

- Removed trips where **passenger_count** was greater than 6, as such instances were extremely rare.

```
[113]: # remove passenger_count > 6
       # df = df[~(df['passenger_count'] > 6 )]

       df = df[~(df['passenger_count'] > 6 )]
```

```
[56]: df.shape
```

```
[56]: (1885907, 18)
```

- Dropped entries where **trip_distance** was 0 but **fare_amount** exceeded 300, indicating inconsistent data.

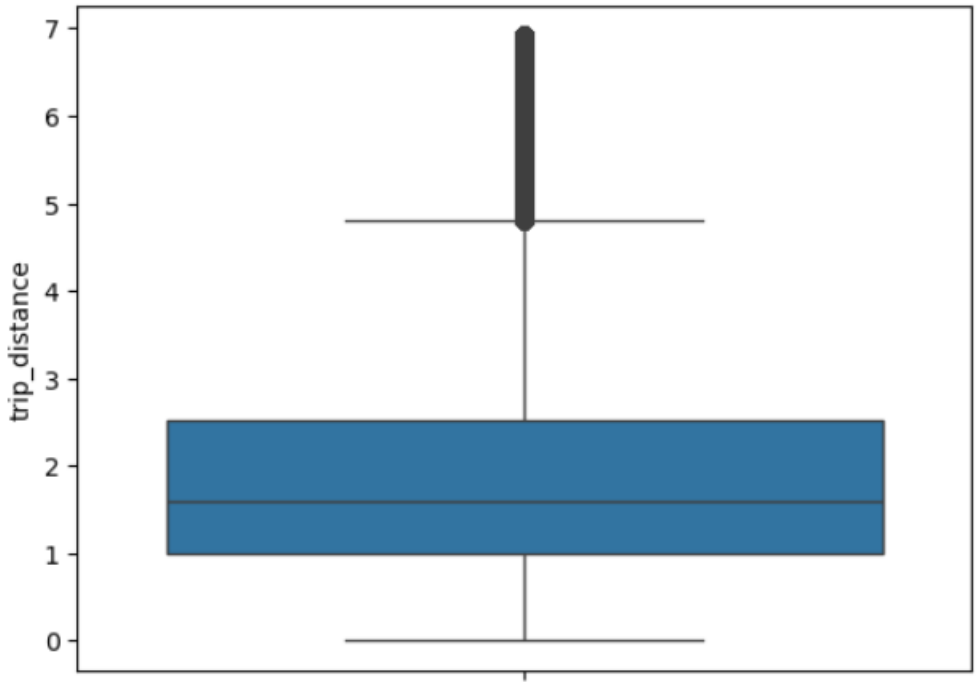
```
[117]: # Continue with outlier handling
       df = df[~((df['trip_distance'] == 0) & (df.fare_amount > 300))]

[119]: df.shape

[119]: (1885875, 18)
```

- Eliminated trips where both **trip_distance** and **fare_amount** were 0, but pickup and dropoff zones were different, as distance and fare should not both be zero in such cases.

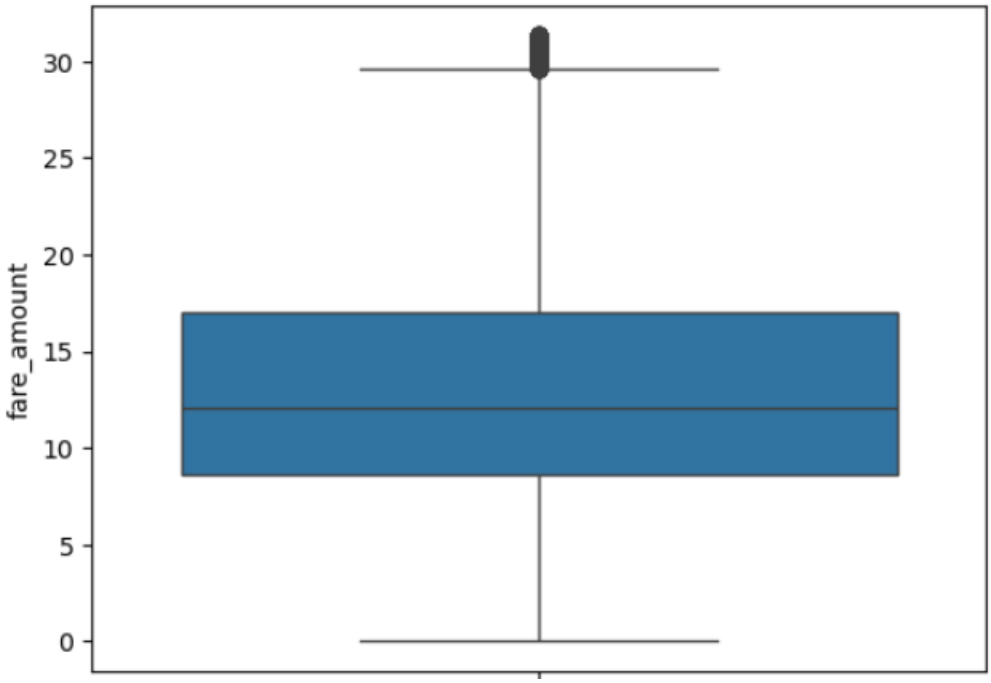
```
[121]: df = df[~((df['trip_distance'] == 0) & (df['fare_amount'] == 0)& (df['PULocationID'] != df['DOLocationID']))]
```

- Removed extreme outliers where **trip_distance** was less than 1.04 miles but **fare_amount** was unusually high (above 200).
  df = df[~((df['trip_distance'] < 1.04) & (df['fare_amount'] >= 200))]
  Boxplot after removing the outliers

```
sns.boxplot(df['trip_distance'])
plt.show()
```

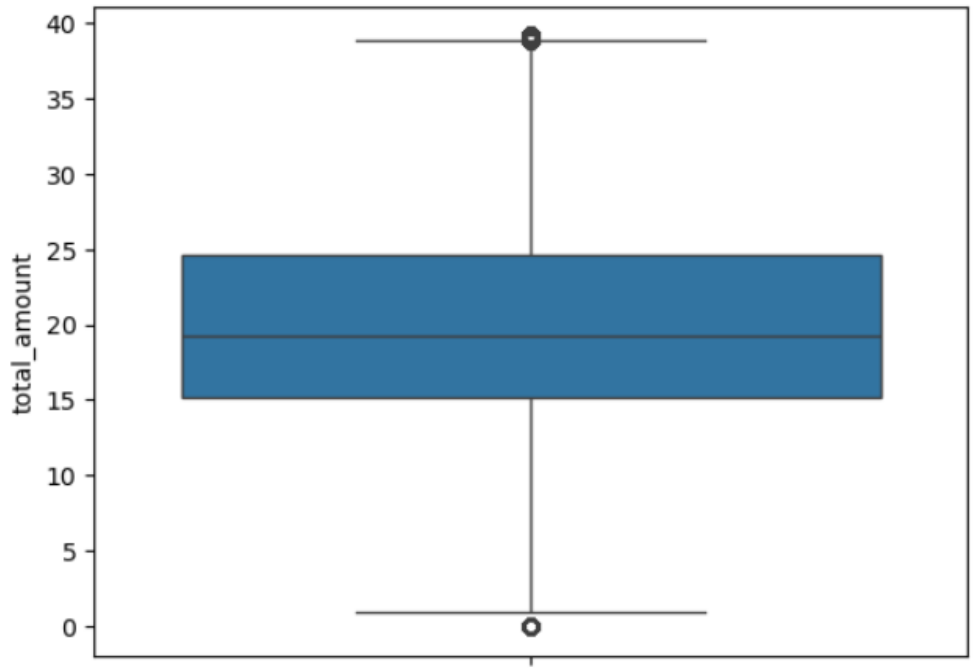


- Cleaned out entries where **fare_amount** was 0 while **trip_distance** was non-zero, indicating invalid records.
- Removed trips where **fare_amount** exceeded 300, based on outlier analysis using the upper bound (calculated as 37.35).
- Boxplot graph after removing the fare_amount



- Similarly, removed records where **total_amount** was greater than or equal to the upper limit to ensure consistency.
  Boxplot graph after removing the outliers

```
#graph after removing outliers
sns.boxplot(df['total_amount'])
plt.show()
```



- Excluded trips where **payment_type** was 0, as such entries indicate missing or invalid payment information.

```
[282]: #seems like payment_type is zero for some cases, need to remove those entries

       df = df[~(df['payment_type'] == 0) ] #removing 38k record
```

```
[284]: df.shape
```

```
[284]: (1504609, 18)
```

```
[286]: df[['payment_type']].describe().T
```

[286]:

	count	mean	std	min	25%	50%	75%	max
payment_type	1504609.0	1.207656	0.467683	1.0	1.0	1.0	1.0	4.0

3. Exploratory Data Analysis

3.1.General EDA: Finding Patterns and Trends

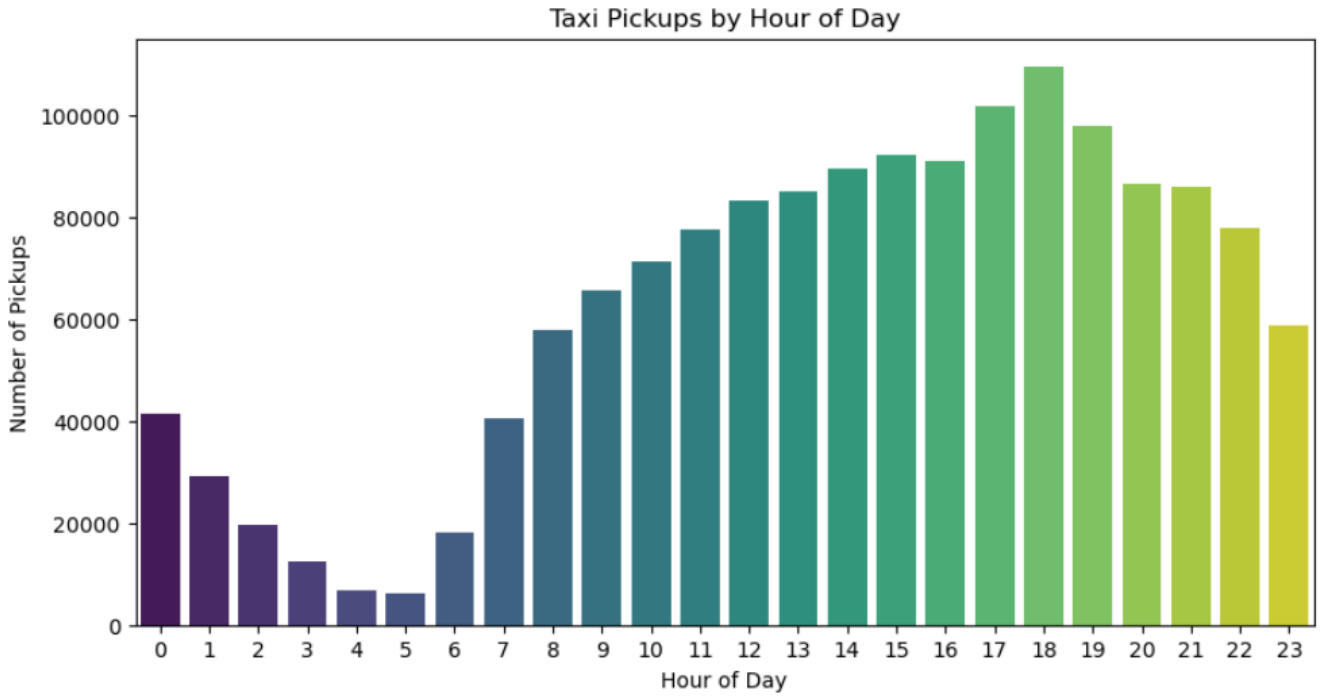
3.1.1. Classify variables into categorical and numerical

- Categorise the varaibles into Numerical or Categorical.
  - VendorID - Categorical
  - tpep_pickup_datetime - Date
  - tpep_dropoff_datetime - Date
  - passenger_count - Numerical
  - trip_distance - Numerical
  - RatecodeID - Categorical
  - PULocationID - Taxi Zone - Categorical
  - DOLocationID - Taxi Zone - Categorical
  - payment_type - Categorical
  - pickup_hour - Numerical
  - trip_duration - Numerical
- The following monetary parameters belong in the same category, is it categorical or numerical?
  - fare_amount - Numerical
  - extra - Numerical
  - mta_tax - Numerical
  - tip_amount - Numerical
  - tolls_amount - Numerical
  - improvement_surcharge - Numerical
  - total_amount - Numerical
  - congestion_surcharge - Numerical
  - airport_fee - Numerical

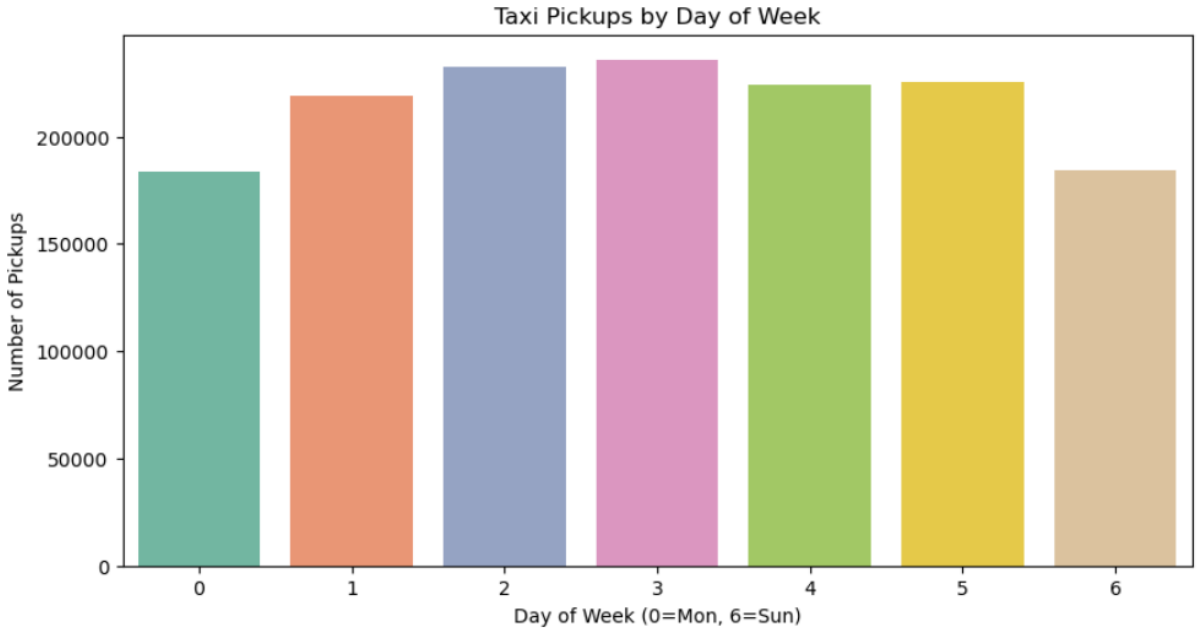
3.1.2. Analyse the distribution of taxi pickups by hours, days of the week, and months

Temporal Analysis

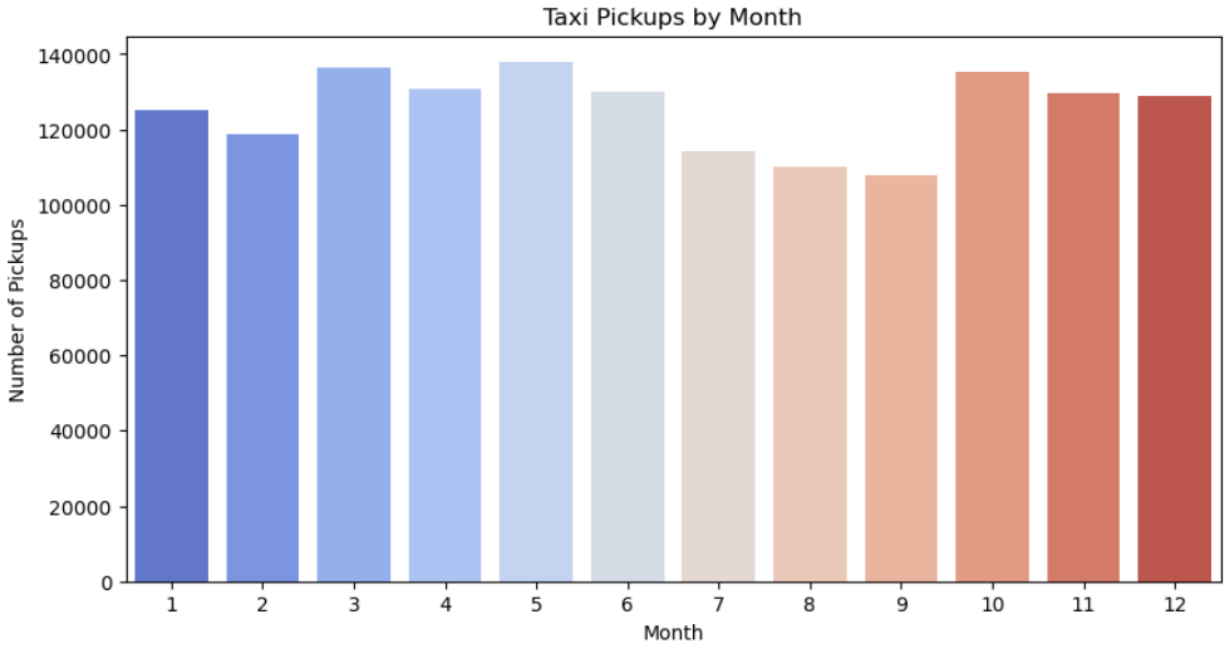
- To analyze the distribution of taxi pickups across different timeframes, new columns for hour, day of the week, and month were created from the pickup datetime field, and the distributions were visualized using count plots
- **Graph taxi pickups by hour of day – 15 – 19 hours have more pickups**



- **Graph counterplot by day of week**



- **Graph counterplot by Monthly**



3.1.3. Filter out the zero/negative values in fares, distance and tips

- To check the validity of financial parameters like fare_amount, tip_amount, total_amount, and trip_distance, the dataset was inspected for any zero or negative values using a filtering code snippet.

```
# Analyse the above parameters
cols_to_check = ['fare_amount', 'tip_amount', 'total_amount', 'trip_distance']
for col in cols_to_check:
    count_neg = df[df[col] < 0].shape[0]
    count_zero = df[df[col] == 0].shape[0]
    print(f"{col}:")
    print(f"  Negative values: {count_neg}")
    print(f"  Zero values:     {count_zero}\n")
```

- Output:
  - fare_amount:
    - Negative values: 0
    - Zero values: 250
  - tip_amount:
    - Negative values: 0
    - Zero values: 311227
  - total_amount:
    - Negative values: 0
    - Zero values: 176
  - trip_distance:
    - Negative values: 0
    - Zero values: 10241

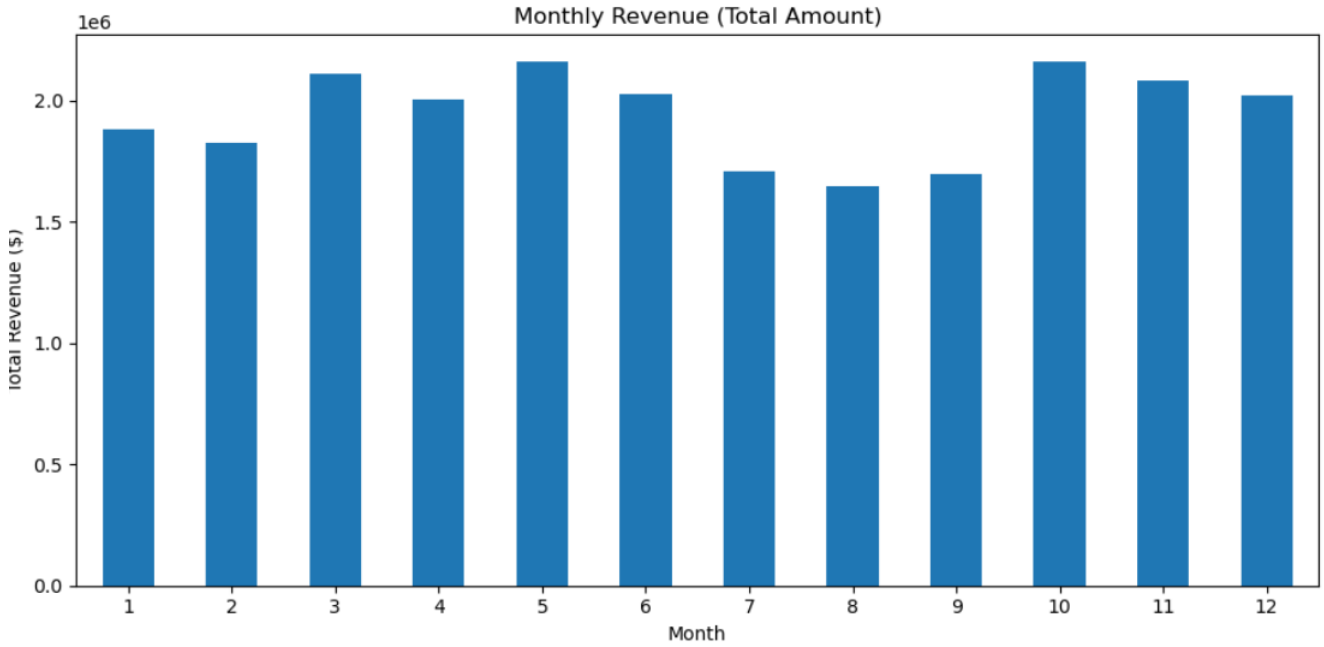
- All rows with zero values in key financial columns (fare_amount, tip_amount, total_amount, and trip_distance) were removed to ensure the dataset reflects only valid and meaningful taxi transactions.

```
# Create a df with non zero entries for the selected parameters.
# non_zero_df = df[(df[ ['fare_amount', 'tip_amount', 'total_amount', 'trip_distance'] ]!= 0)]
non_zero_df = df[(df[['fare_amount', 'tip_amount', 'total_amount', 'trip_distance']] != 0).all(axis=1)]
```

- After removing I found indexes are changed, now reset index is done using reset_index() method
- For future analysis, non_zero_df will be used.

3.1.4. Analyse the monthly revenue trends

- Next, the monthly revenue trend was analyzed by grouping the data by month and summing the total_amount. The resulting values were plotted as a bar graph to visualize the revenue distribution across months.
- **Plotting the bar graph for visual representation**

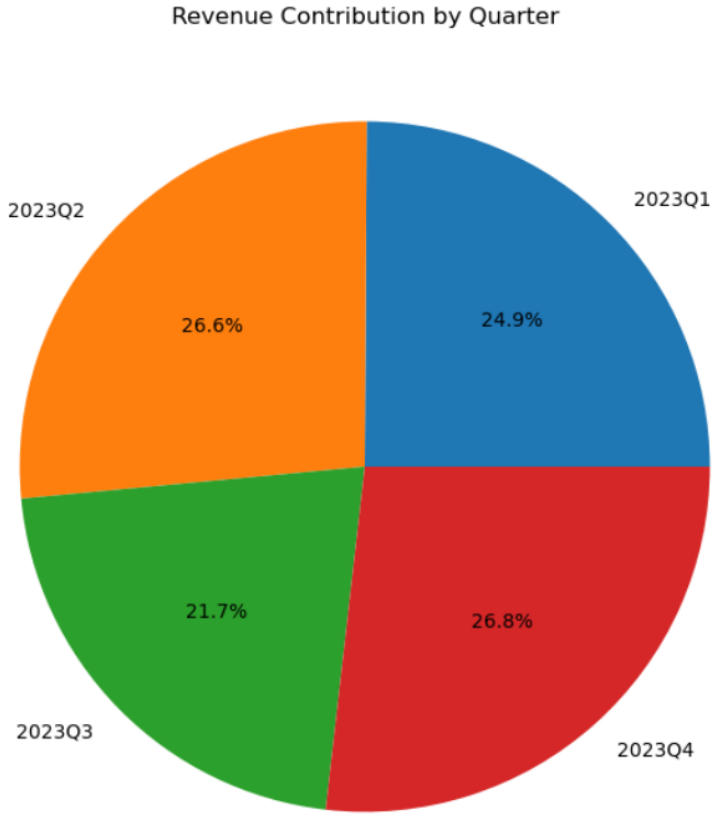


3.1.5. Find the proportion of each quarter's revenue in the yearly revenue

- The quarter column was created by extracting the quarter information from the tpep_pickup_datetime field. Then, the proportion of revenue contributed by each quarter of the year 2013 was calculated based on the total_amount.

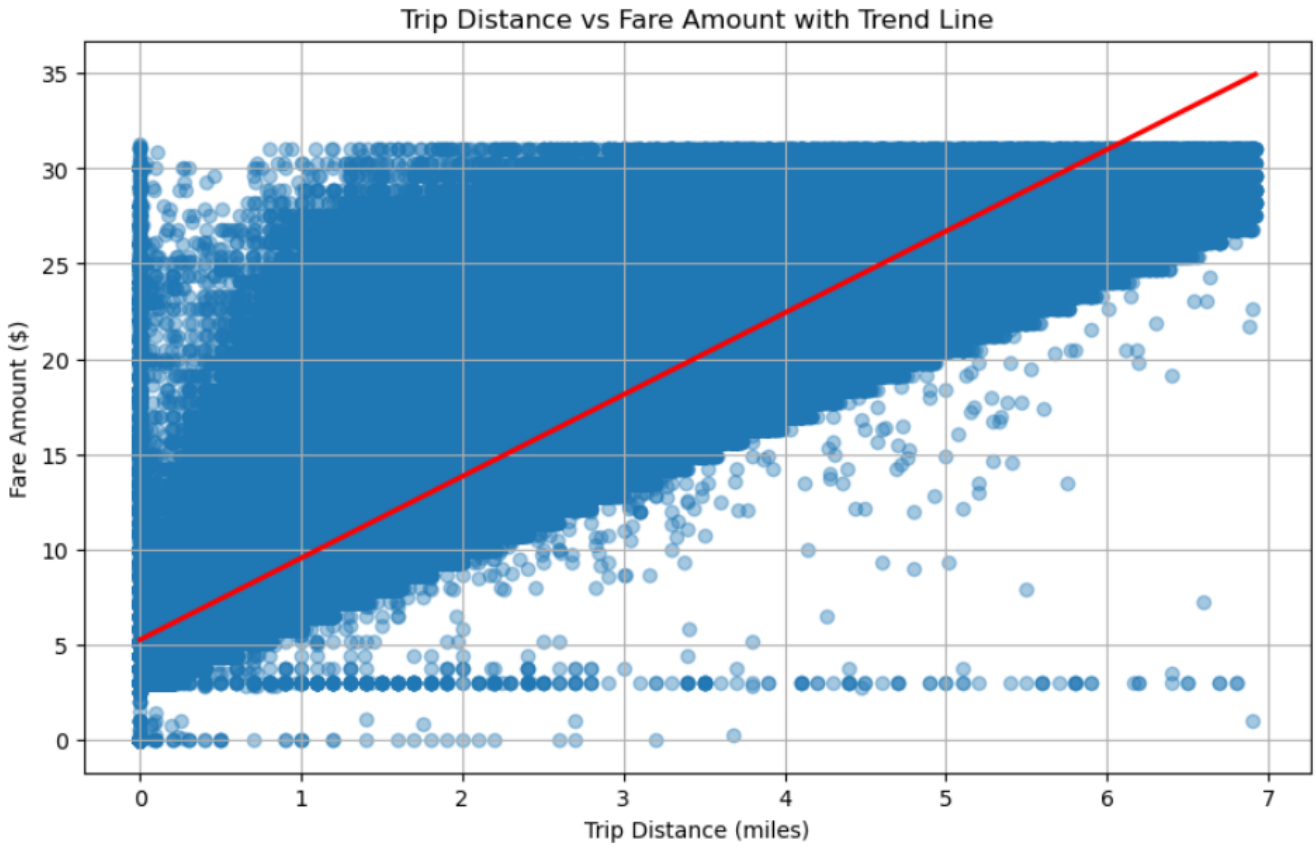
```
non_zero_df['quarter'] = non_zero_df['tpep_pickup_datetime'].dt.to_period('Q')
```

- The quarterly revenue was grouped and summed based on the quarter column, and the proportion of each quarter's contribution to the total annual revenue was calculated and printed for analysis.
- Now, plotting the quarterly revenue distribution to visualize the seasonal trends and identify the strongest revenue-generating periods.



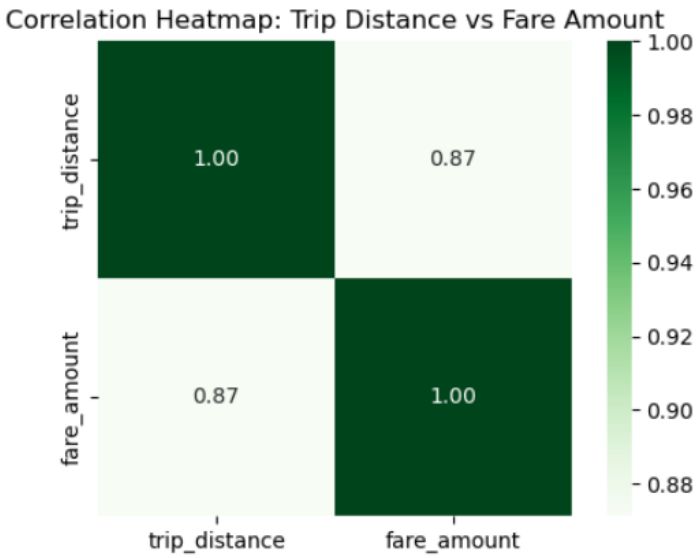
3.1.6. Analyse and visualise the relationship between and fare amount

- Next, visualized the relationship between trip_distance and fare_amount using a scatter plot to understand the distribution and trend between these variables.



- Additionally, plotted a heatmap to find the correlation coefficient, confirming the strength and direction of the relationship.

```
corr_features = df_filtered[['trip_distance', 'fare_amount']]
# Calculate the correlation matrix
corr_matrix = corr_features.corr()
```



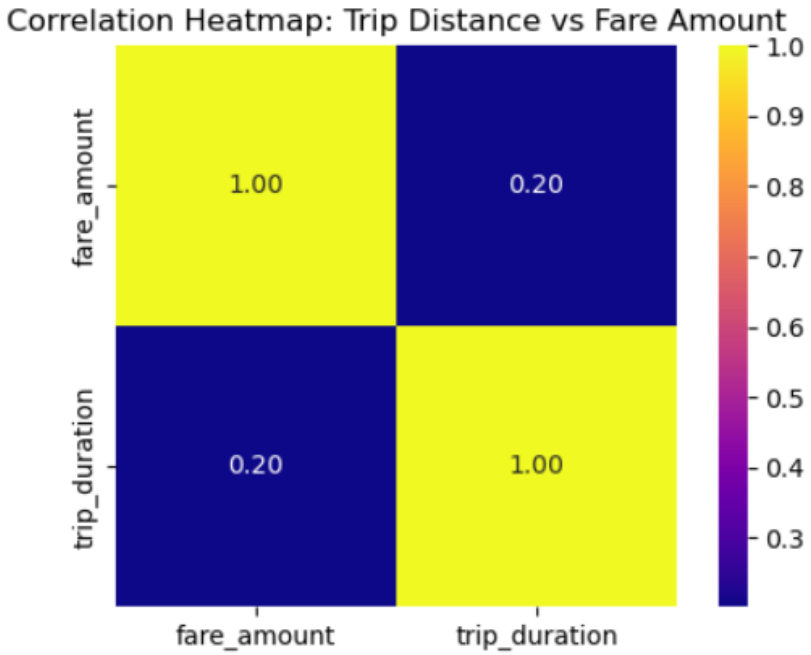
3.1.7. Analyse the relationship between fare/tips and trips/passengers

- Calculated and plotted the correlation between fare_amount and trip duration (difference between pickup and dropoff times).
    - Created a new column trip_duration by calculating the difference between tpep_dropoff_datetime and tpep_pickup_datetime, and converting the result into minutes using the below code

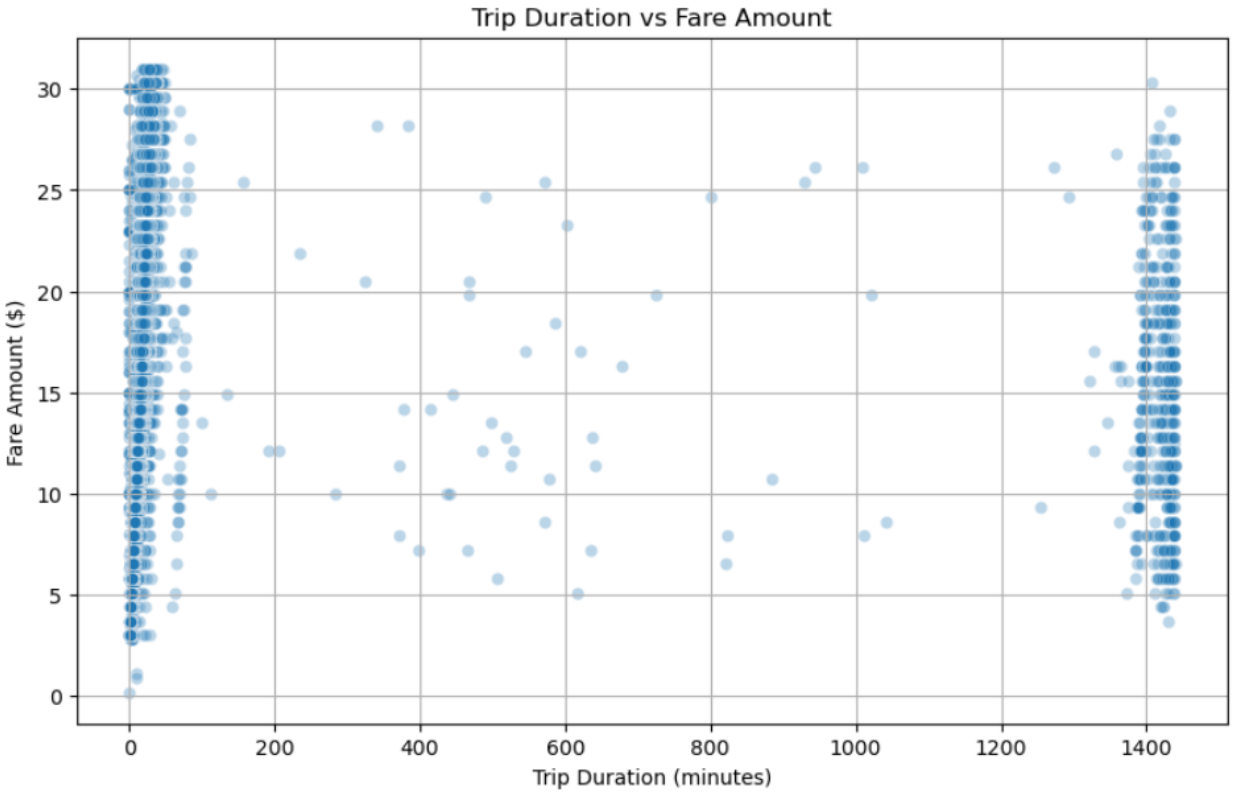
```
# Show relationship between fare and trip duration
##creating new column trip_duration by subtracting dropoff - pickup time
non_zero_df['trip_duration'] = (non_zero_df['tpep_dropoff_datetime'] - non_zero_df['tpep_pickup_datetime']).dt.total_seconds() / 60
```

    - This trip_duration column was then used to calculate and visualize the correlation between fare_amount and trip duration.

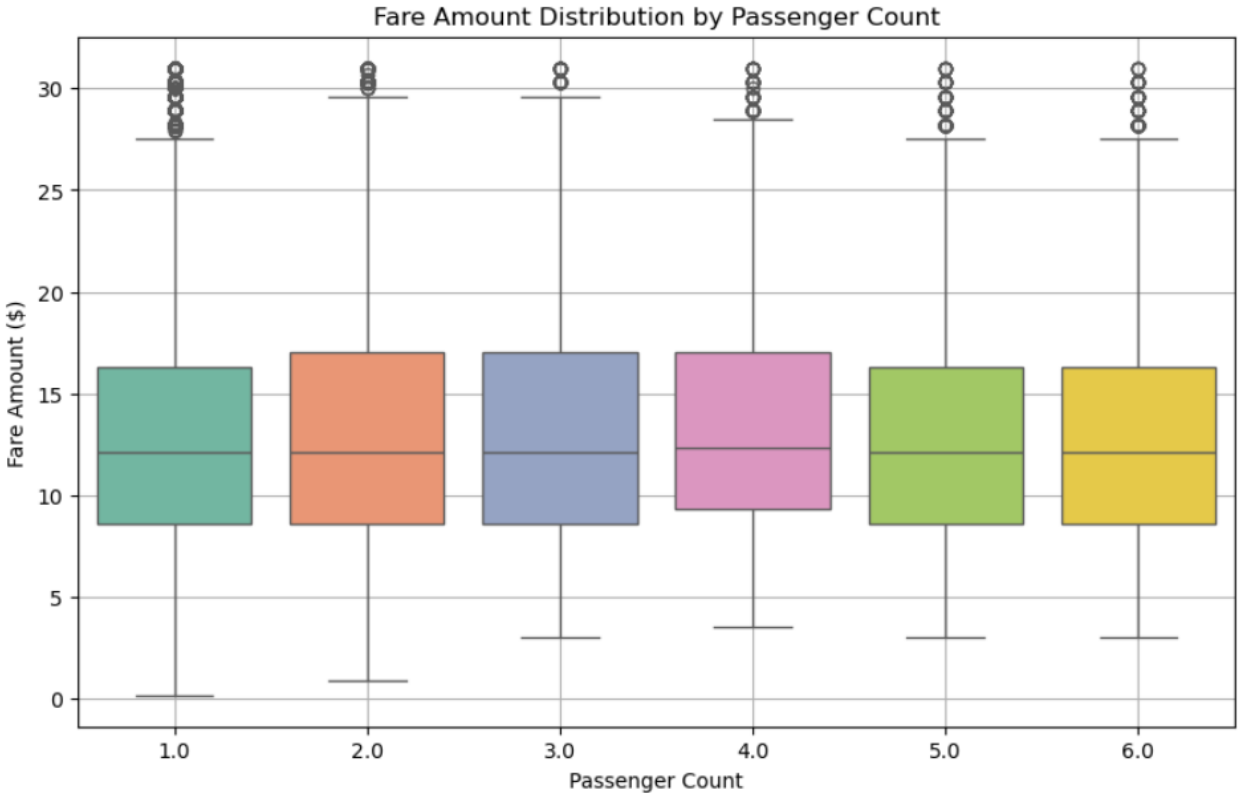
    - Correlation in heatmap



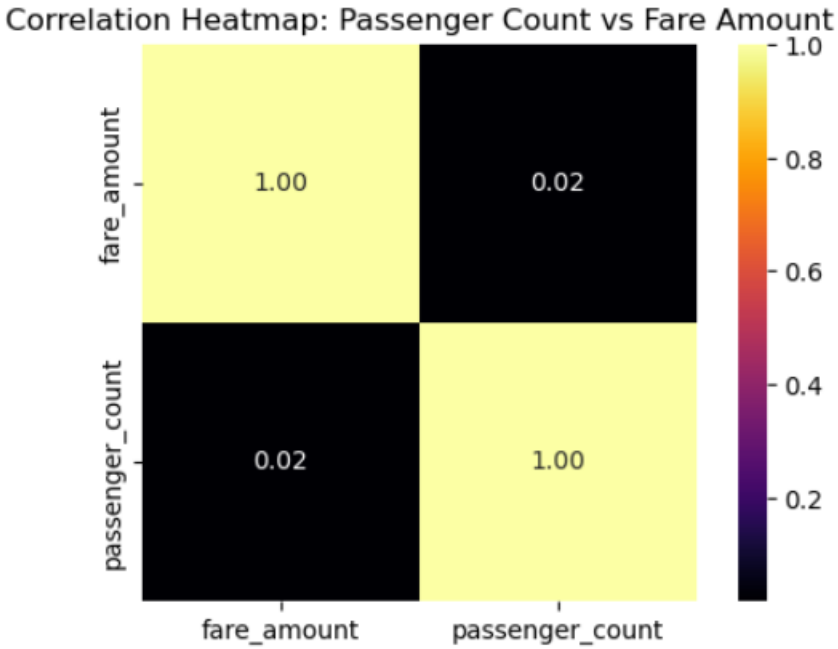
    - The relationship between fare_amount and trip_duration was visualized using a scatter plot to explore how fare changes with the duration of trips.



- The relationship between fare_amount and passenger_count was visualized using a box plot to understand the distribution of fare amounts for different numbers of passengers.

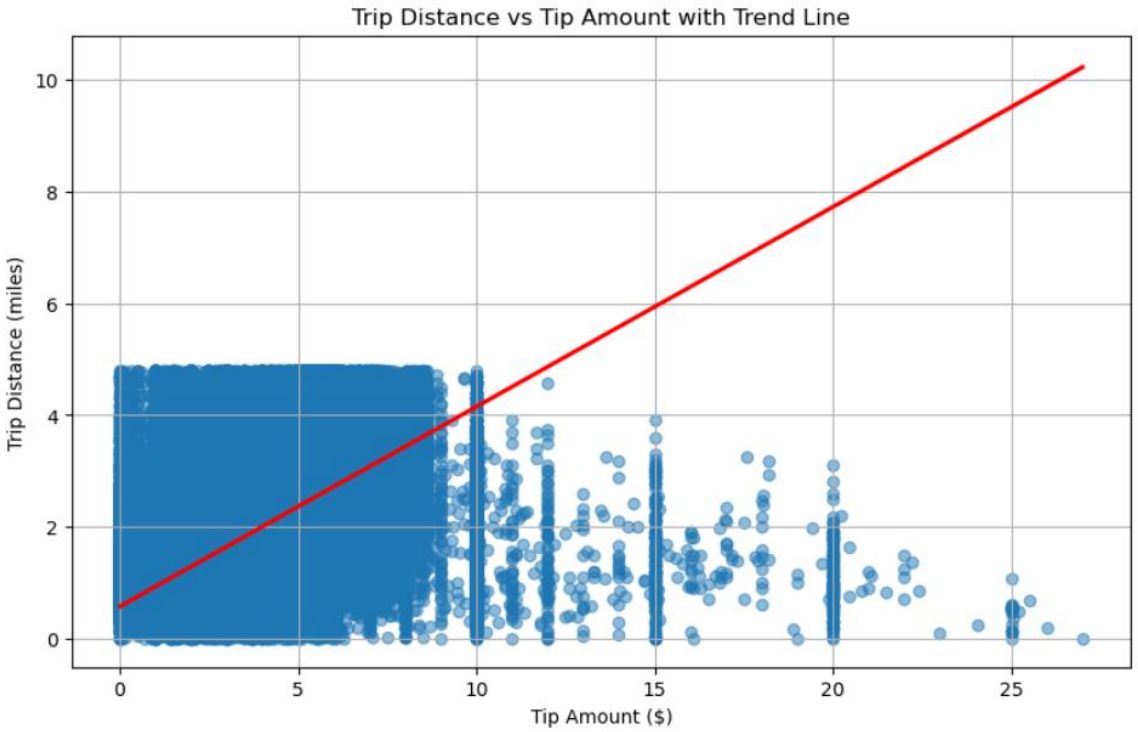


- Correclation between fare_amount and Passenger Count has been calculated, plotted on the heatmap

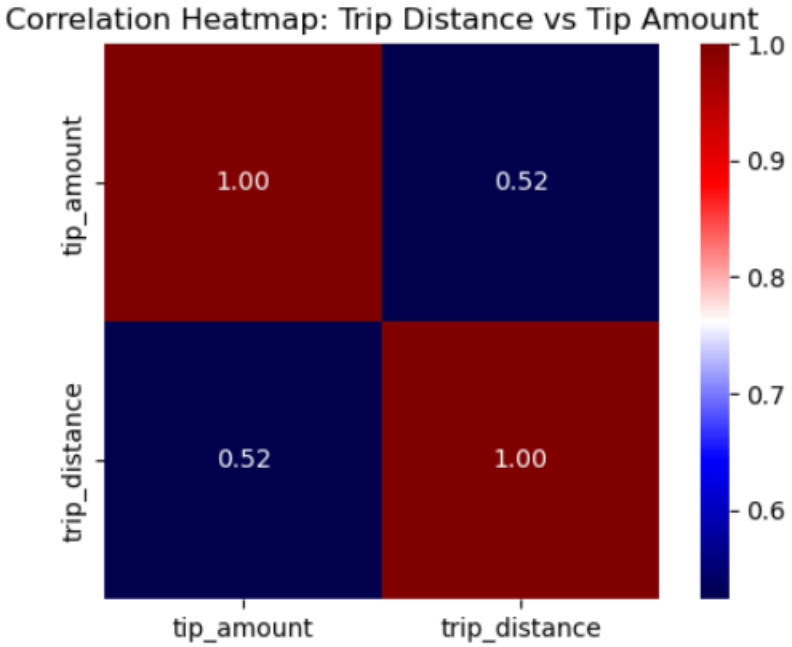


- The correlation between tip_amount and trip_distance will be analyzed by calculating the correlation coefficient between these two variables.

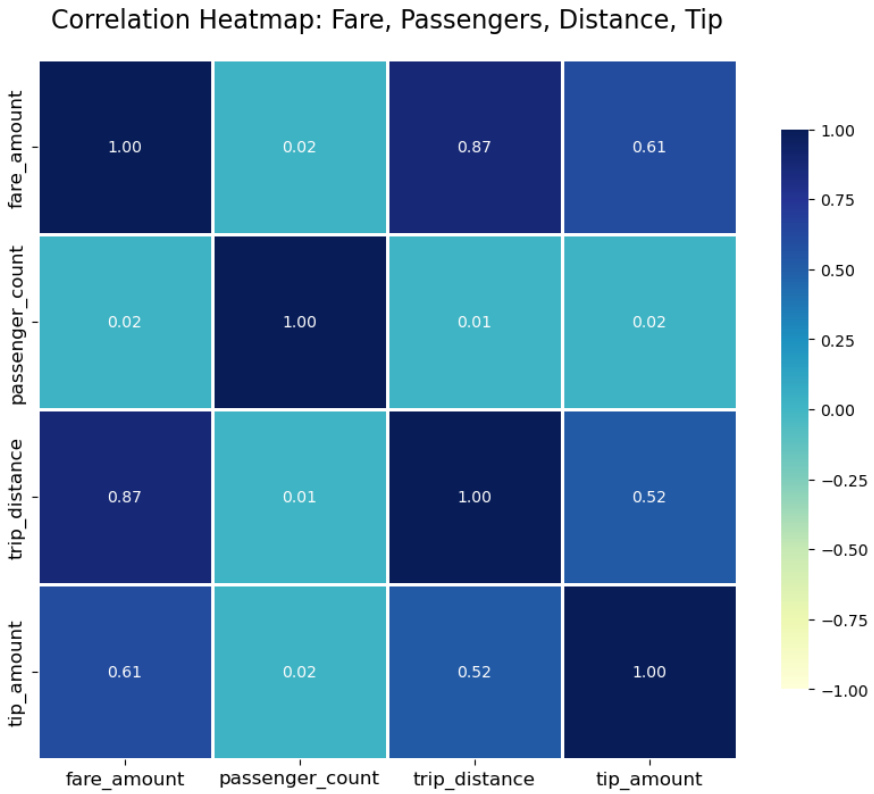
- A scatter plot will be created to visualize the relationship between tip amount and trip distance.



- Additionally, a heatmap will be plotted to show the strength and direction of their correlation, providing insights into how trip distance impacts the tip amount.



- Correlation for all the mentioned field (fare_amount,passenger_count, trip_distance, tip_amount)



3.1.8. Analyse the distribution of different payment types

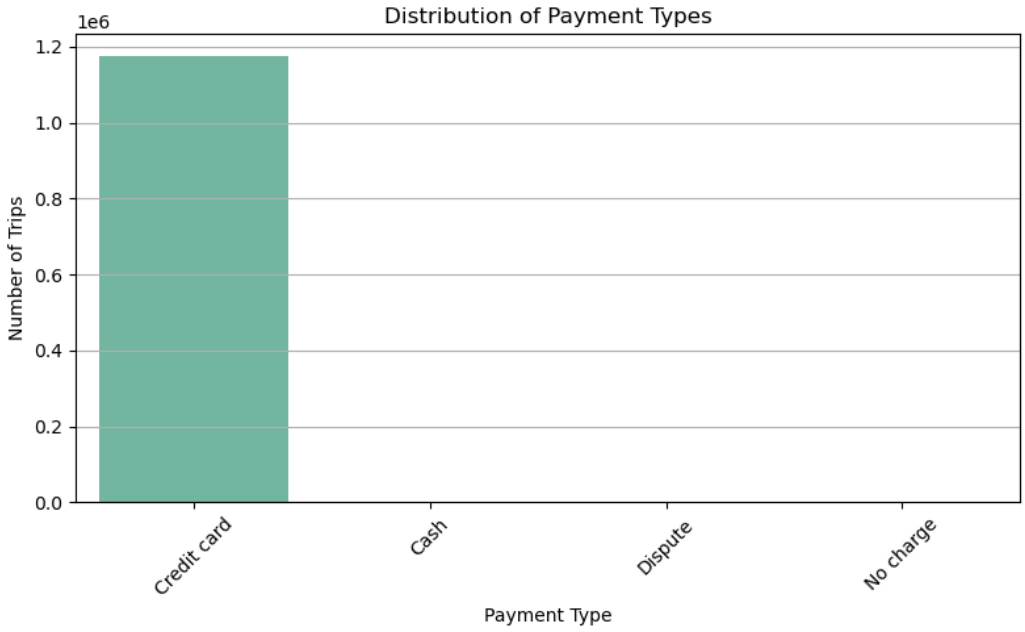
- Extract the payment_types, map with data dictonary.

```
[398]: payment_counts_label.head()
```

[398]:

	Payment Type	Value
0	Credit card	1173830
1	Cash	21
2	Dispute	11
3	No charge	5

- 99% of the people prefer Credit card as their primary mode of payment.
- That reflect in the graph



3.1.9. Load the taxi zones shapefile and display it

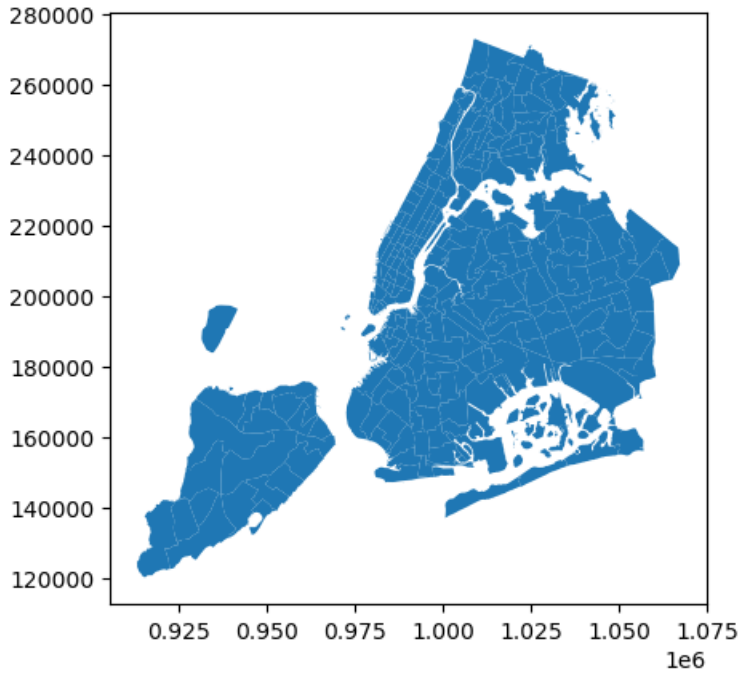
- Install the geopandas library to handle geospatial data.
- Read the shapefile (SHP) using geopandas to extract geographic information.
- Display the geographic data to visualize and analyze geographic patterns.
- Incorporate geo-coordinate details into the analysis for better insights.

[400]:

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID	borough	geometry
0	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...
1	2	0.433470	0.004866	Jamaica Bay	2	Queens	MULTIPOLYGON (((1033269.244 172126.008, 103343...
2	3	0.084341	0.000314	Allerton/Pelham Gardens	3	Bronx	POLYGON ((1026308.77 256767.698, 1026495.593 2...
3	4	0.043567	0.000112	Alphabet City	4	Manhattan	POLYGON ((992073.467 203714.076, 992068.667 20...
4	5	0.092146	0.000498	Arden Heights	5	Staten Island	POLYGON ((935843.31 144283.336, 936046.565 144...

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 263 entries, 0 to 262
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype   
---  ------      --------------  -----   
 0   OBJECTID    263 non-null    int32   
 1   Shape_Leng  263 non-null    float64 
 2   Shape_Area  263 non-null    float64 
 3   zone        263 non-null    object  
 4   LocationID  263 non-null    int32   
 5   borough     263 non-null    object  
 6   geometry    263 non-null    geometry
dtypes: float64(2), geometry(1), int32(2), object(2)
memory usage: 12.5+ KB
None
```

: <Axes: >



3.1.10. Merge the zone data with trips data

- Merge non_zero_df with the zones dataframe to combine the data.
- This merged dataframe will be used for future analysis, integrating both trip-related and geographic information.

```
[404]: #Renaming LocationID to PULocationID to before merge
       zones = zones.rename(columns={"LocationID": "PULocationID"})

[406]: zones.info()

[408]: # Merge zones and trip records using locationID and PULocationID
       df_merged = non_zero_df.merge(zones, on="PULocationID", how="left")

[410]: df_merged.info()
```

3.1.11. Find the number of trips for each zone/location ID

- Grouped the merged dataframe (df_merged) by PULocationID to find the total number of trips at each pickup location using the size() method.
- Reset the index to create a clean dataframe trip_count_df with columns PULocationID and total_trips.

```
trip_count = df_merged.groupby('PULocationID').size().reset_index(name='total_trips')
trip_count.head(20)
```

3.1.12. Add the number of trips for each zone to the zones dataframe

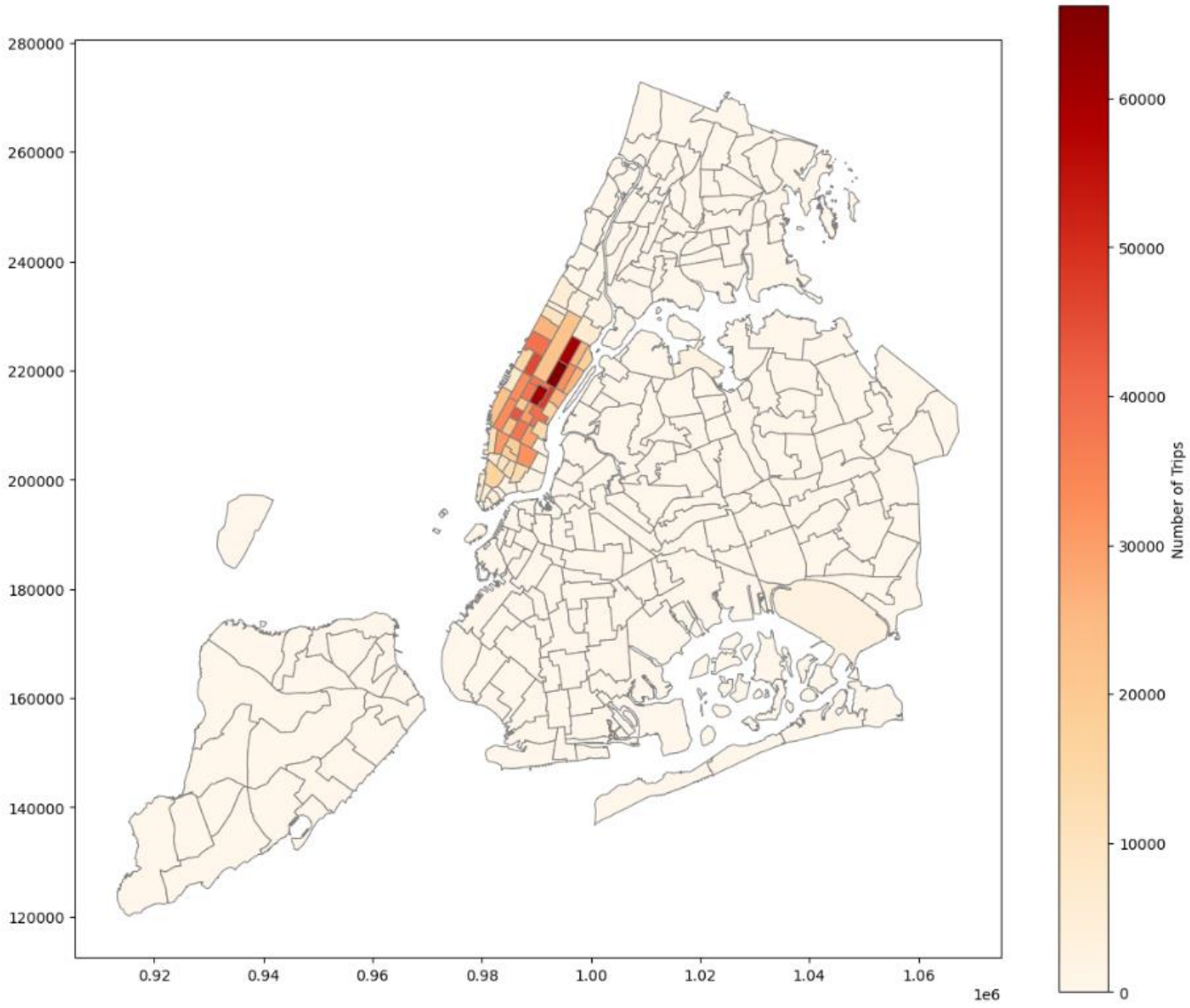
- Merged the trip_count_df with the zones GeoDataFrame using PULocationID as the key and left join method to retain all zone information.
- Created a new GeoDataFrame gdf_merged that contains geographic details along with the trip counts for each location.

```
# Merge trip counts back to the zones GeoDataFrame
gdf_merged = zones.merge(trip_count_df, on='PULocationID', how='left')
```

```
gdf_merged['total_trips'] = gdf_merged['total_trips'].fillna(0)
```

3.1.13. Plot a map of the zones showing number of trips

- Used the gdf_merged GeoDataFrame to plot the map, where the color of each zone represents the number of trips, using the "OrRd" color map.
- AS converted NaN as Zero



3.1.14. Conclude with results

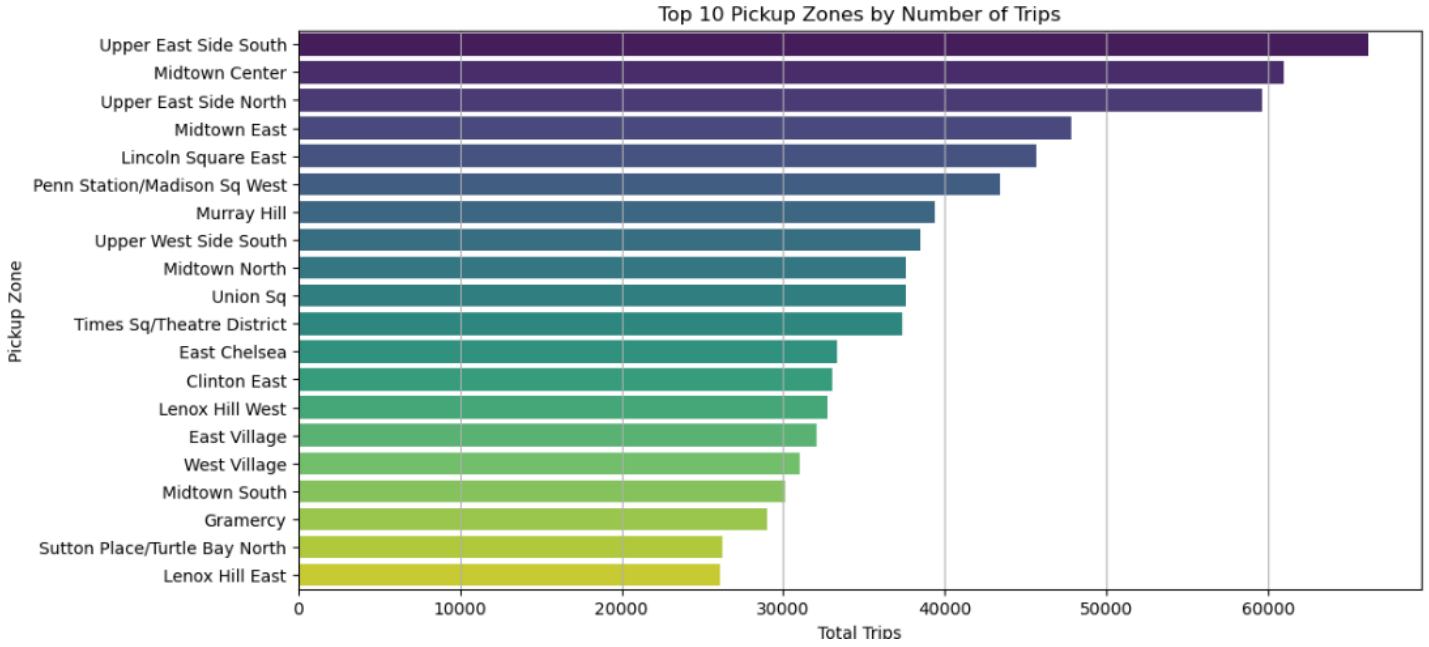
- Merged the GeoDataFrame containing location zone geometries with the DataFrame holding the number of trips per pickup location ID, ensuring that spatial information and trip counts were combined for further geospatial analysis and visualization.

```
# can you try displaying the zones DF sorted by the number of trips?
print(gdf_merged.info())

gdf_merged.sort_values(by='total_trips',ascending=False).head(20)
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 263 entries, 0 to 262
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   OBJECTID      263 non-null    int32
 1   Shape_Leng    263 non-null    float64
 2   Shape_Area    263 non-null    float64
 3   zone          263 non-null    object
 4   PULocationID  263 non-null    int32
 5   borough       263 non-null    object
 6   geometry      263 non-null    geometry
 7   index         186 non-null    float64
 8   total_trips   263 non-null    float64
dtypes: float64(4), geometry(1), int32(2), object(2)
memory usage: 16.6+ KB
None
```

	OBJECTID	Shape_Leng	Shape_Area	zone	PULocationID	borough	geometry	index	total_trips
236	237	0.042213	0.000096	Upper East Side South	237	Manhattan	POLYGON ((993633.442 216961.016, 993507.232 21...	166.0	66192.0
160	161	0.035804	0.000072	Midtown Center	161	Manhattan	POLYGON ((991081.026 214453.698, 990952.644 21...	108.0	60966.0
235	236	0.044252	0.000103	Upper East Side North	236	Manhattan	POLYGON ((995940.048 221122.92, 995812.322 220...	165.0	59643.0
161	162	0.035270	0.000048	Midtown East	162	Manhattan	POLYGON ((992224.354 214415.293, 992096.999 21...	109.0	47859.0
141	142	0.038176	0.000076	Lincoln Square East	142	Manhattan	POLYGON ((989380.305 218980.247, 989359.803 21...	93.0	45698.0
185	186	0.024696	0.000037	Penn Station/Madison Sq West	186	Manhattan	POLYGON ((986752.603 210853.699, 986627.863 21...	126.0	43432.0

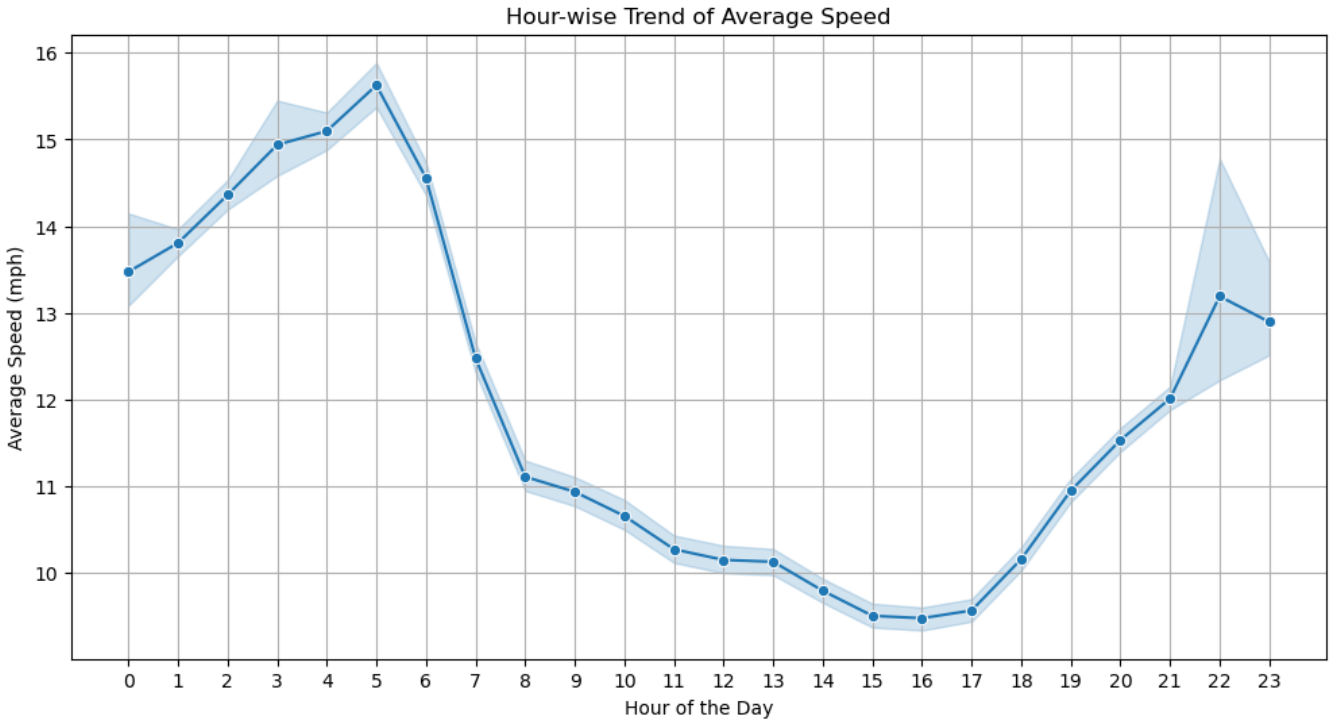
- A bar plot was created to display the top 20 pickup locations with the highest number of trips, helping to easily visualize which areas had the most taxi activity.



3.2.Detailed EDA: Insights and Strategies

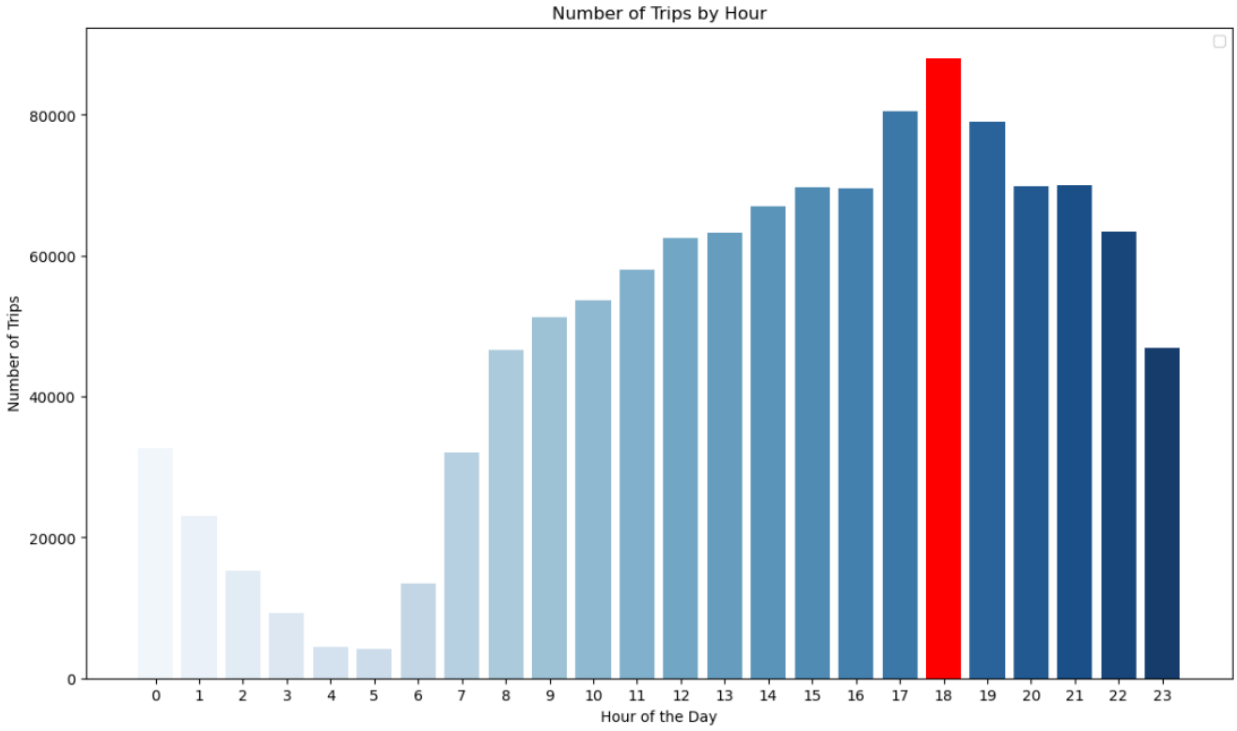
3.2.1. Identify slow routes by comparing average speeds on different routes

- Grouped the dataset by PULocationID, DOLocationID, and pickup_hour to calculate:
- Average trip distance (avg_distance)
- Average trip duration (avg_duration)
- Calculated the average speed (in miles per hour) for each route using the formula:
- avg_speed_mph = (avg_distance / avg_duration) * 60
- Computed the overall mean average speed across all routes to serve as a benchmark.
- Identified slow routes by filtering the routes where the average speed was less than the benchmark.
- Sorted the slow routes in ascending order based on their average speed to highlight the slowest ones
- Plot it in line graph



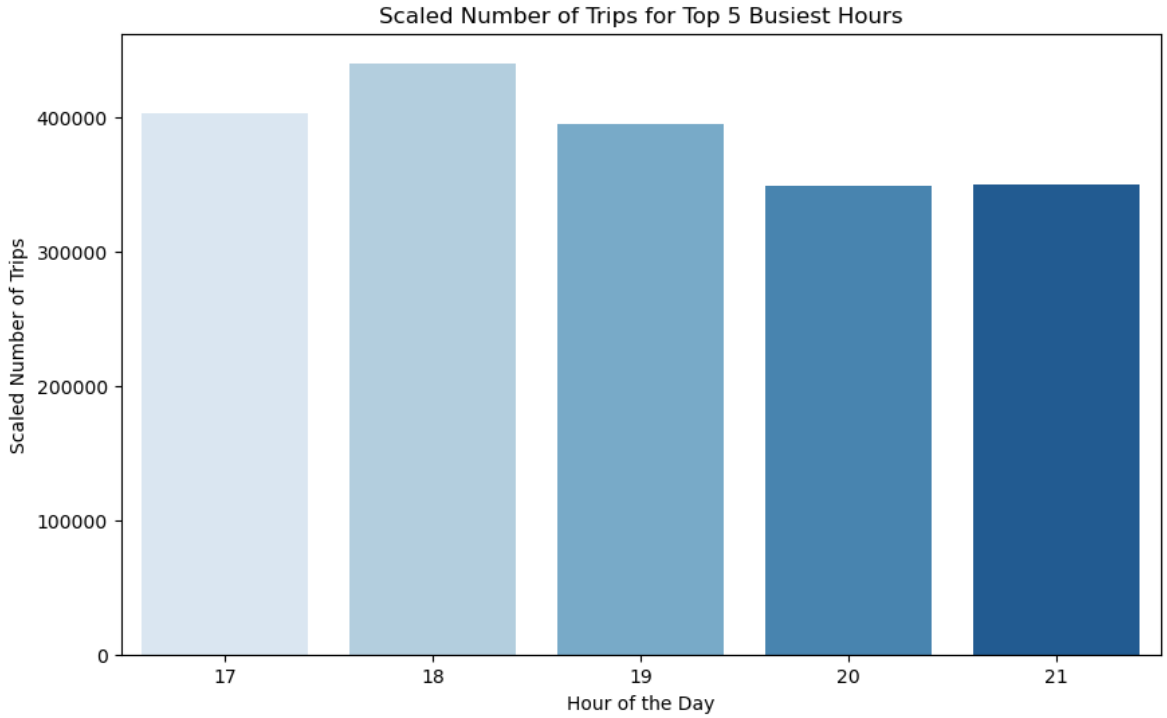
3.2.2. Calculate the hourly number of trips and identify the busy hours

- Calculated the number of trips for each hour of the day by grouping the data based on the pickup_hour column.
- Identified the busiest hour by finding the hour with the maximum number of trips.
- Plotted the distribution of trips per hour using a bar plot, with the busiest hour highlighted distinctly to show the peak activity



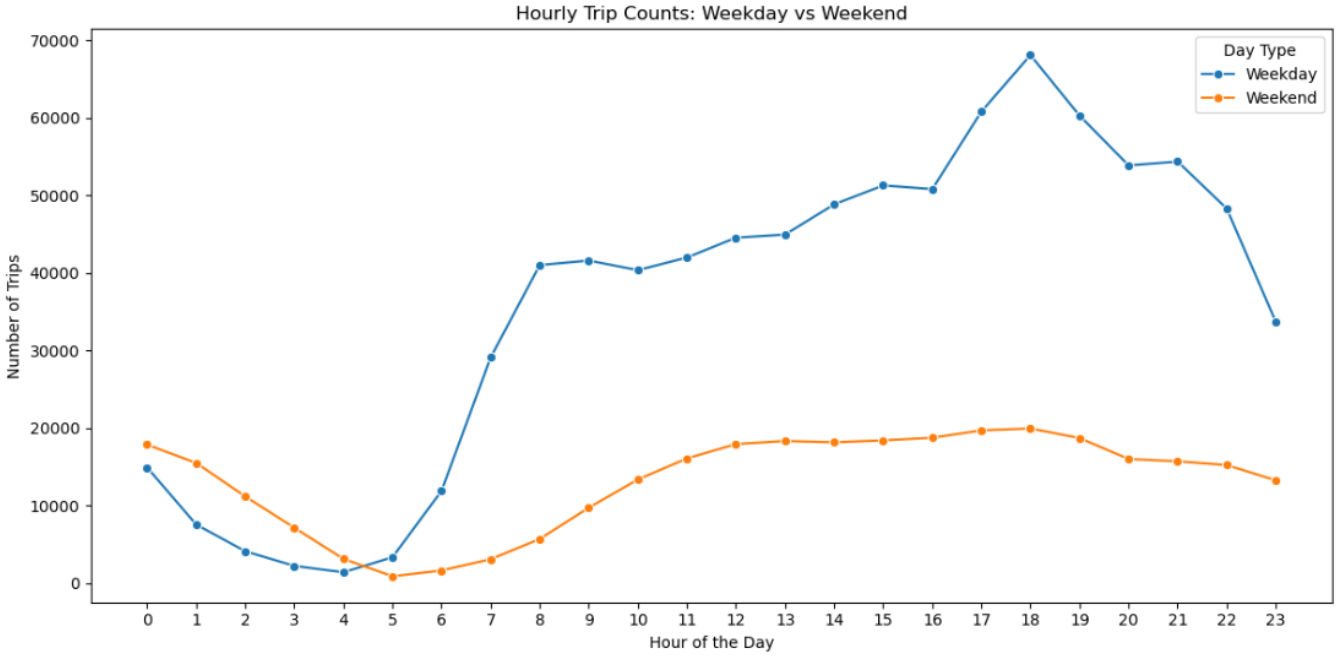
3.2.3. Scale up the number of trips from above to find the actual number of trips

- Identified the top 5 busiest hours by sorting the trip counts by hour in descending order.
- Scaled up the number of trips by dividing the counts by the sampling fraction (0.2), to estimate the total trips for each of those hours.
- Plotted the scaled trip counts for the top 5 busiest hours using a bar graph for clear visualization.



3.2.4. Compare hourly traffic on weekdays and weekends

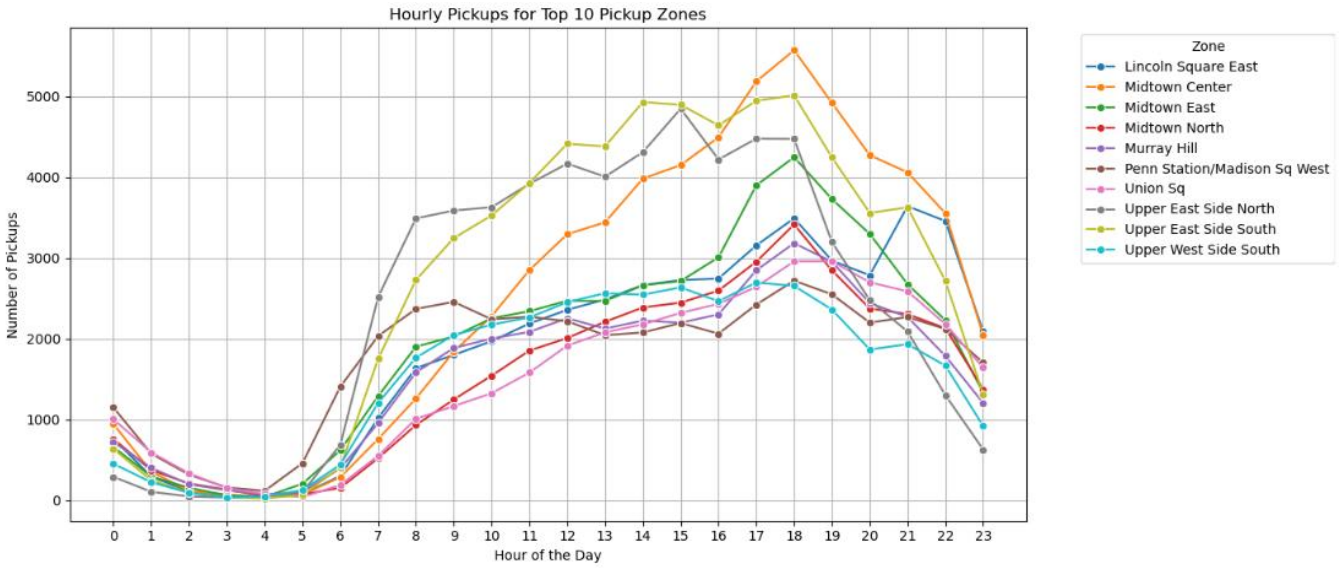
- Added a new column is_weekend in the dataframe based on the day_of_week, marking weekends (Saturday and Sunday) as 1 and weekdays as 0.
- Grouped the data by pickup_hour and is_weekend to count the number of trips for each hour separately for weekdays and weekends.
- Mapped the is_weekend values to a new day_type column with labels Weekday and Weekend for better readability.
- Plotted a **line graph** using seaborn.lineplot() to compare the **hourly trip counts** between weekdays and weekends:
    - **X-axis**: Pickup hour
    - **Y-axis**: Number of trips
    - **Hue**: Day type (Weekday or Weekend) with different colors



3.2.5. Identify the top 10 zones with high hourly pickups and drops

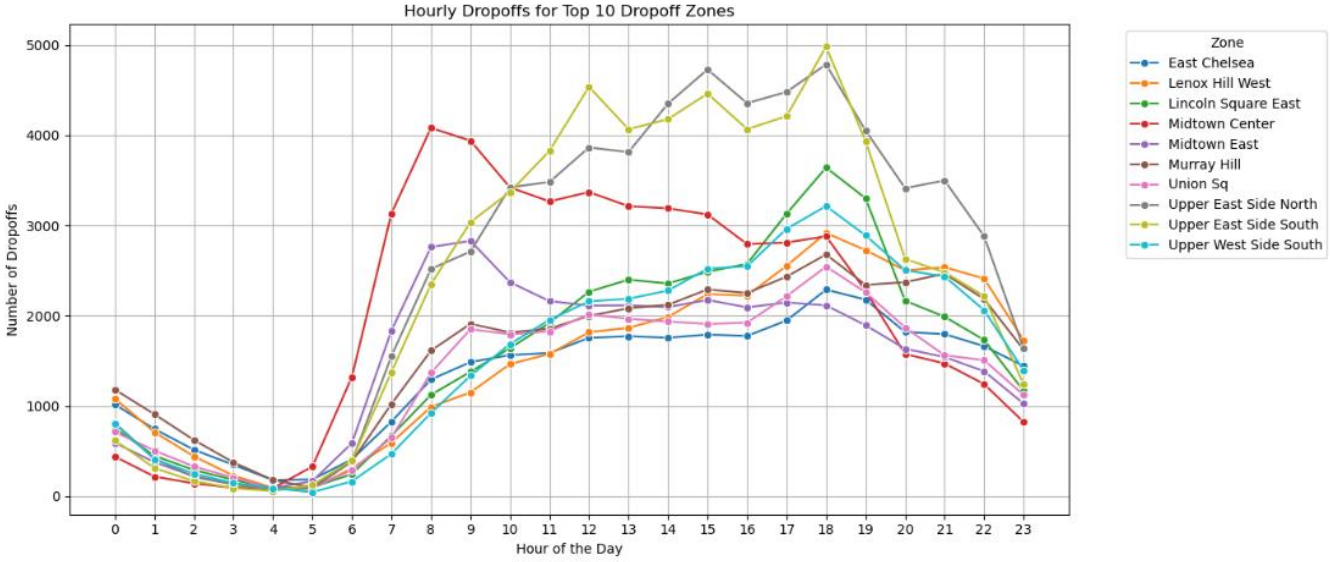
Top 10 Pick up analysis

- Grouped the dataset by PULocationID and pickup_hour to calculate the number of pickups at each location and hour.
- Aggregated the data to find the top 10 pickup zones based on the highest total number of pickups.
- Filtered the pickup data to include only these top 10 pickup zones.
- Merged the filtered pickup data with zone information to map the PULocationID to corresponding zone names



Top 10 Drop off Analysis

- Grouped the dataset by DOLocationID and dropoff_hour to calculate the number of drop-offs at each location and hour.
- Aggregated the data to find the top 10 drop-off zones based on the highest total number of drop-offs.
- Filtered the drop-off data to include only these top 10 drop-off zones.
- Merged the filtered drop-off data with zone information to map the DOLocationID to corresponding zone names.



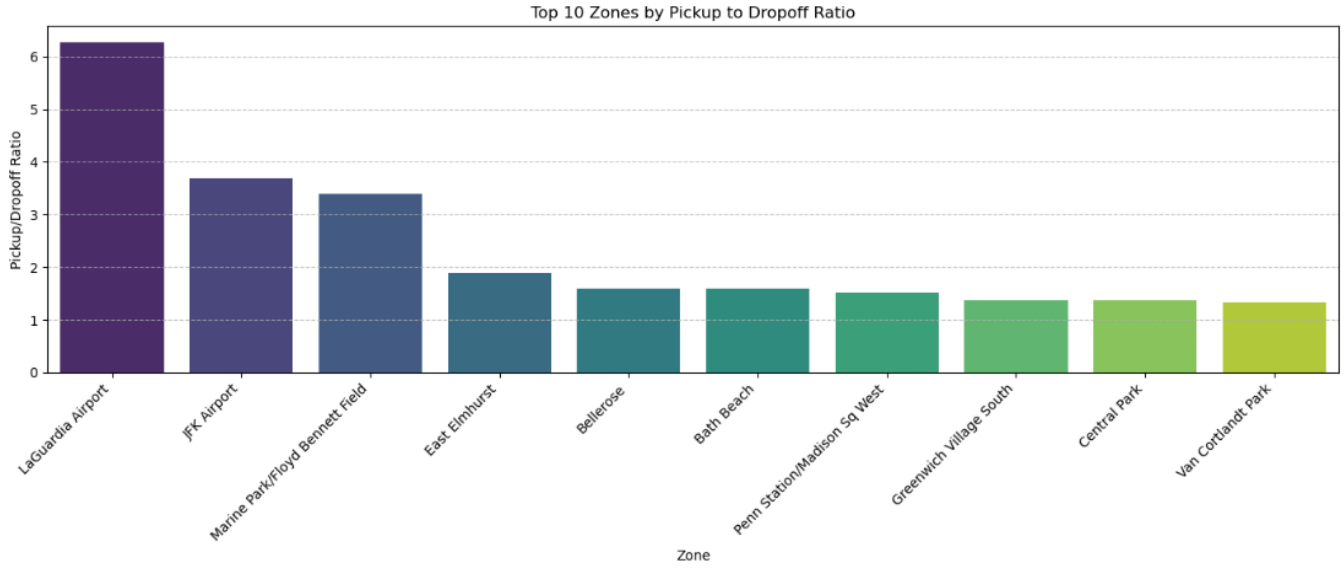
3.2.6. Find the ratio of pickups and dropoffs in each zone

Top 10 Zones with the Highest Pickup-to-Dropoff Ratio:

- The code top10_ratio = zone_flow.sort_values(by='pickup_to_dropoff_ratio', ascending=False).head(10) selects the top 10 zones where the pickup-to-dropoff ratio is the highest. This indicates which zones are

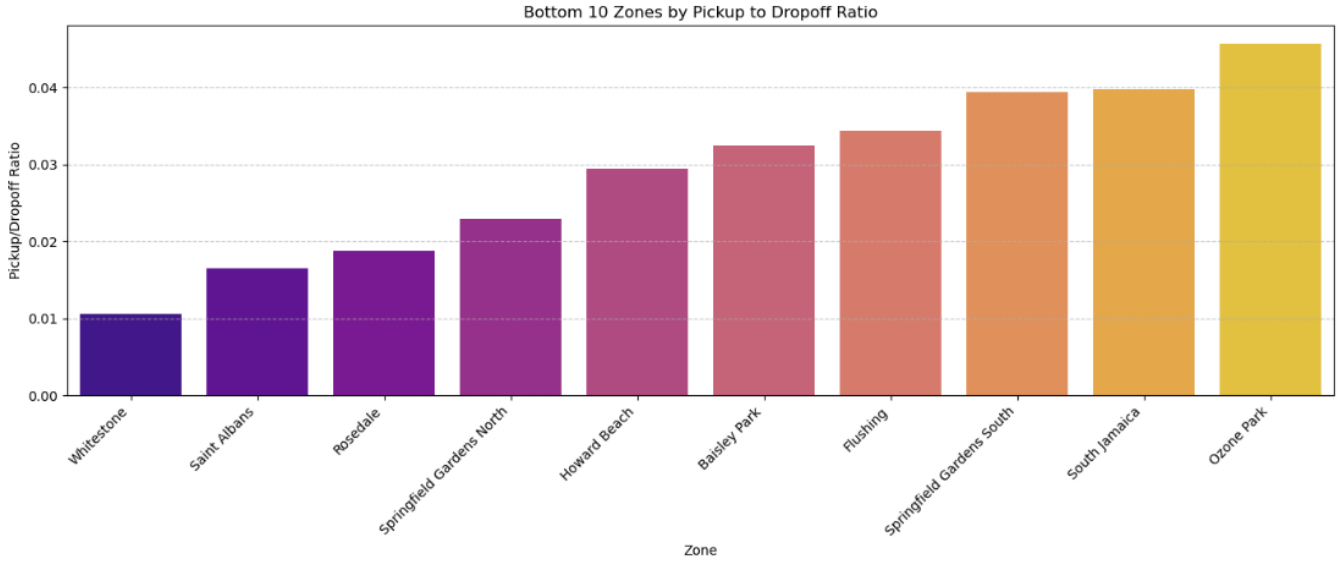
more focused on pickups compared to drop-offs. A higher ratio suggests that passengers are more frequently getting picked up in these zones relative to those being dropped off.

- We then displayed the relevant data for these top zones using print(top10_ratio[['zone', 'pickup_count', 'dropoff_count', 'pickup_to_dropoff_ratio']]).



Top 10 Zones with the Lowest Pickup-to-Dropoff Ratio:

- Similarly, the code bottom10_ratio = zone_flow.sort_values(by='pickup_to_dropoff_ratio', ascending=True).head(10) identifies the bottom 10 zones with the lowest pickup-to-dropoff ratio. This implies that these zones see more drop-offs compared to pickups.
- We printed the relevant information for these bottom zones using print(bottom10_ratio[['zone', 'pickup_count', 'dropoff_count', 'pickup_to_dropoff_ratio']]).



3.2.7. Identify the top zones with high traffic during night hours

- The dataset was filtered to capture only the rows where the pickup hour is between 11 PM and 5 AM.

```
# Filter for night hours (11PM to 5AM)
night_df = df_merged[(df_merged['pickup_hour'] >= 23) | (df_merged['pickup_hour'] <= 5)]
```

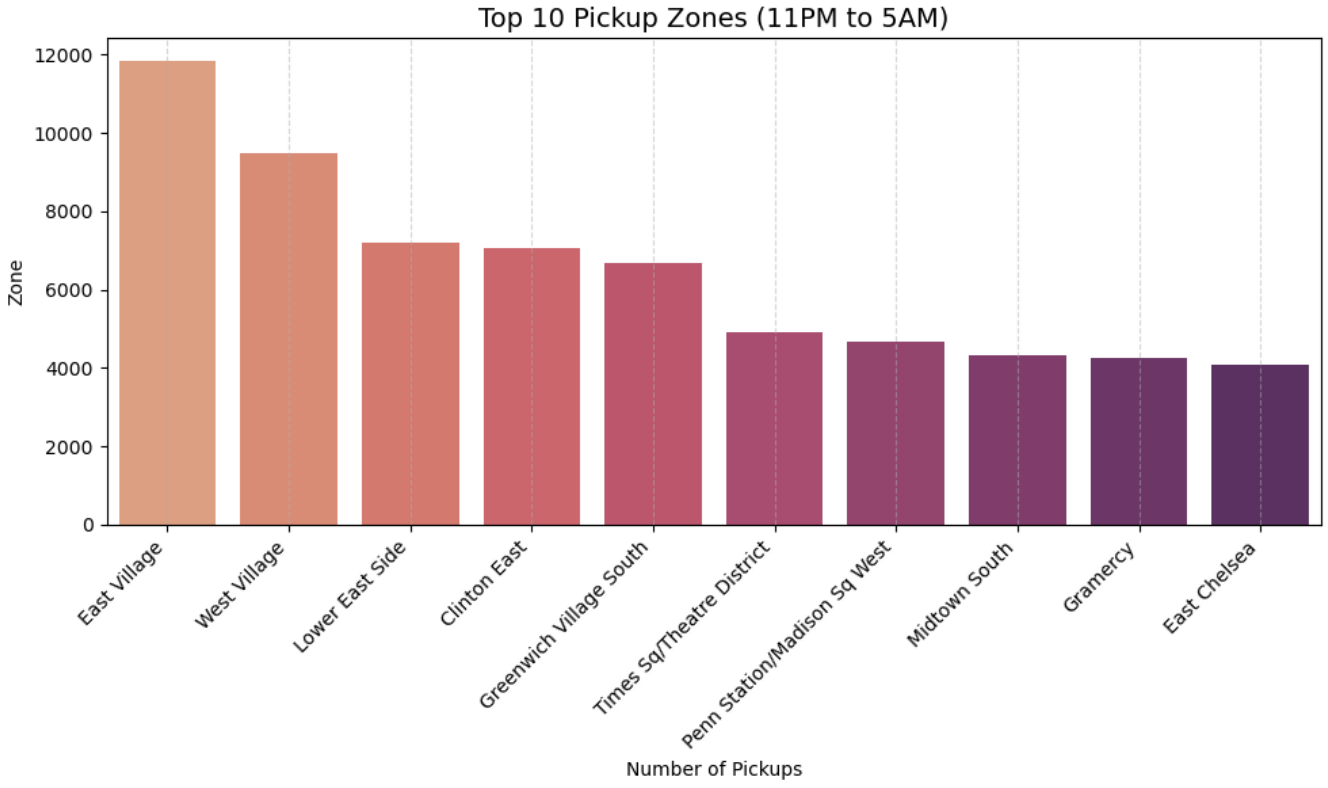
- The dataset was grouped by PULocationID (pickup location) to determine the number of pickups in each zone during the night. The code:

```
# Top 10 Pickup zones during night hours
night_pickups = (
    night_df.groupby('PULocationID')
    .size()
    .reset_index(name='pickup_count')
    .sort_values(by='pickup_count', ascending=False)
    .head(10)
)
```

- A similar approach was used to identify the top 10 dropoff zones by grouping the dataset by DOLocationID (dropoff location)

```
# Top 10 Dropoff zones during night hours
night_dropoffs = (
    night_df.groupby('DOLocationID')
    .size()
    .reset_index(name='dropoff_count')
    .sort_values(by='dropoff_count', ascending=False)
    .head(10)
)
```

- Plotted on the bar graph



3.2.8. Find the revenue share for nighttime and daytime hours

- The nighttime hours were defined as the time range between 11 PM and 5 AM. This was done by creating a list of hours for the night period

```
night_hours = list(range(23, 24)) + list(range(0, 5))
night_hours
```

- A new column, **is_night**, was added to the dataframe, where trips during the nighttime hours were marked as True and other trips as False

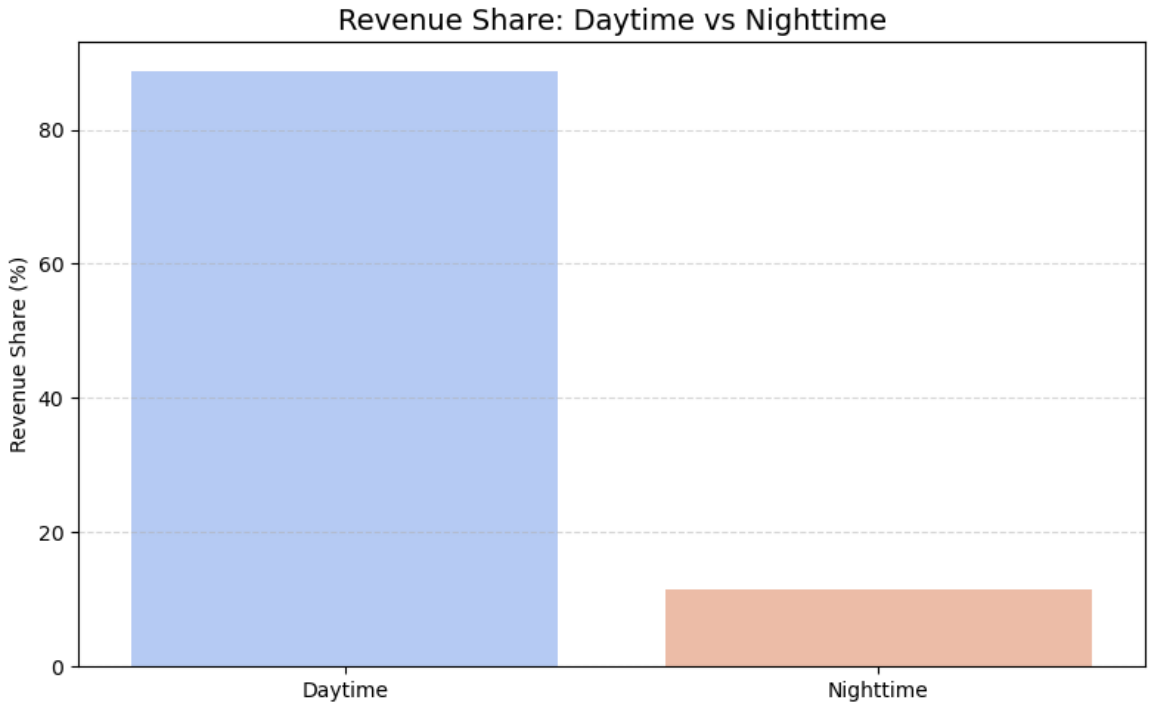
```
df_merged['is_night'] = df_merged['pickup_hour'].isin(night_hours)
# df_merged.head()
```

- After summing the revenues for nighttime and daytime hours, the revenue share percentage was calculated by dividing each group's revenue by the total revenue and multiplying by 100

```
revenue_split['revenue_share_pct'] = 100 * revenue_split['revenue'] / revenue_split['revenue'].sum()
revenue_split
```

	is_night	revenue	revenue_share_pct
0	False	21910211.15	88.585342
1	True	2823238.62	11.414658

- Plotting on the different on the barplot



3.2.9. For the different passenger counts, find the average fare per mile per passenger

- First, we need to calculate the fare per mile for each trip. This can be done by dividing the total fare (total_amount) by the total distance traveled (trip_distance):

```
df_merged['fare_per_mile'] = df_merged['total_amount'] / df_merged['trip_distance']
```

- Next, we calculate the fare per mile per passenger by dividing the fare per mile by the passenger count for each trip. This assumes that the passenger count is given in the dataset as passenger_count:

```
df_merged['fare_per_mile_per_passenger'] = df_merged['fare_per_mile'] / df_merged['passenger_count']
```

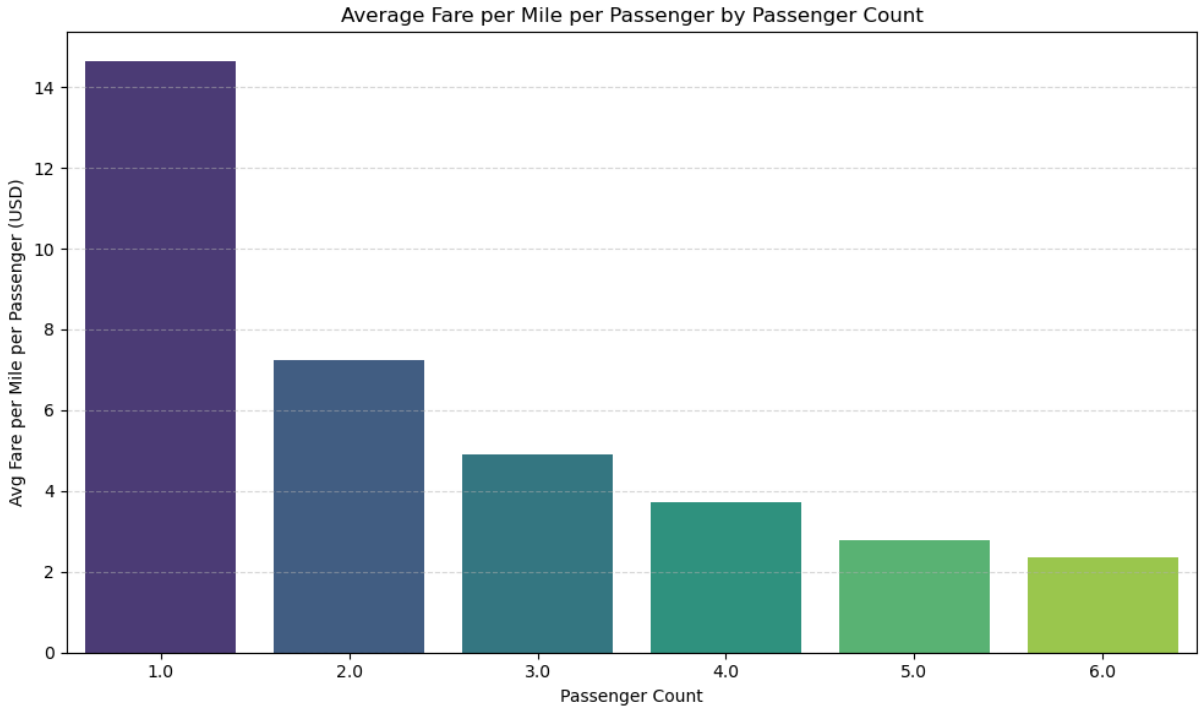
- After calculating the fare per mile per passenger, we group the data by passenger_count and calculate the average fare per mile per passenger for each passenger count group:

```
average_fare_per_passenger = df_merged.groupby('passenger_count')['fare_per_mile_per_passenger'].mean().reset_index()
average_fare_per_passenger
```

:

	passenger_count	fare_per_mile_per_passenger
0	1.0	14.627083
1	2.0	7.244167
2	3.0	4.904405
3	4.0	3.725887
4	5.0	2.789646
5	6.0	2.349978

- The final result, average_fare_per_passenger, will show the average fare per mile per passenger for each passenger count.

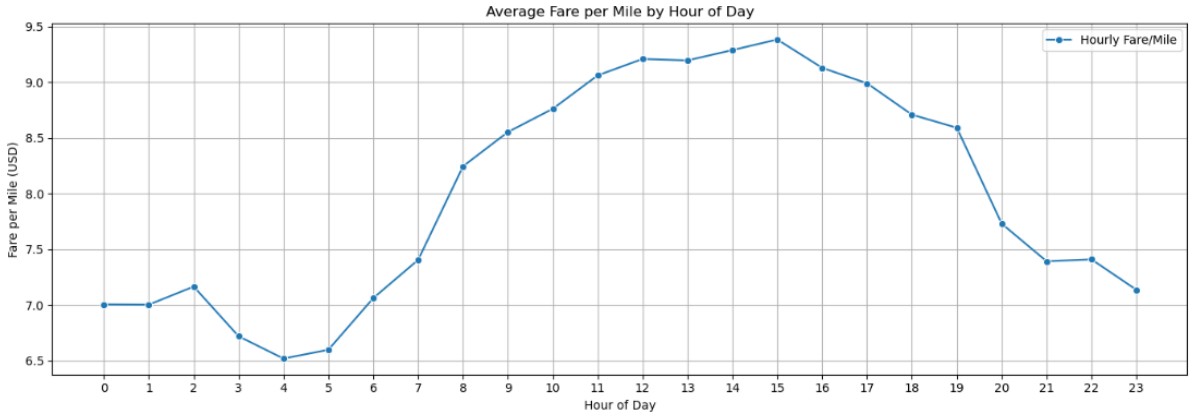


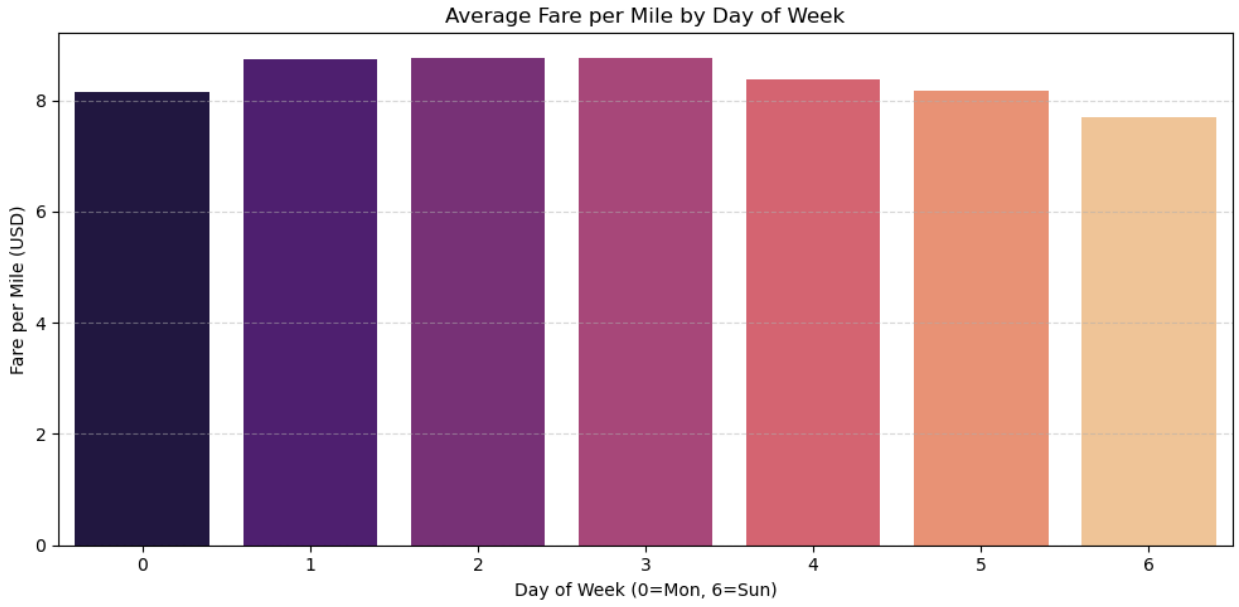
3.2.10. Find the average fare per mile by hours of the day and by days of the week

- The fare per mile is calculated by dividing the total fare by the trip distance for each trip.
- The average fare per mile is computed for each hour of the day by grouping the data by pickup hour and calculating the mean fare per mile for each hour.
- Similarly, the average fare per mile is calculated for each day of the week by grouping the data by the day of the week and calculating the mean fare per mile for each day.

```
# Compare the average fare per mile for different days and for different times of the day

df_merged['fare_per_mile'] = df_merged['fare_amount'] / df_merged['trip_distance']
# df_merged.info()

hourly_fare = df_merged.groupby('pickup_hour')['fare_per_mile'].mean().reset_index()
hourly_fare
daily_fare = df_merged.groupby('day_of_week')['fare_per_mile'].mean().reset_index()
# daily_fare.columns = []
# df_merged.head()
daily_fare
```

- The results can be visualized using line or bar plots to compare how the fare per mile varies across different hours of the day and days of the week.



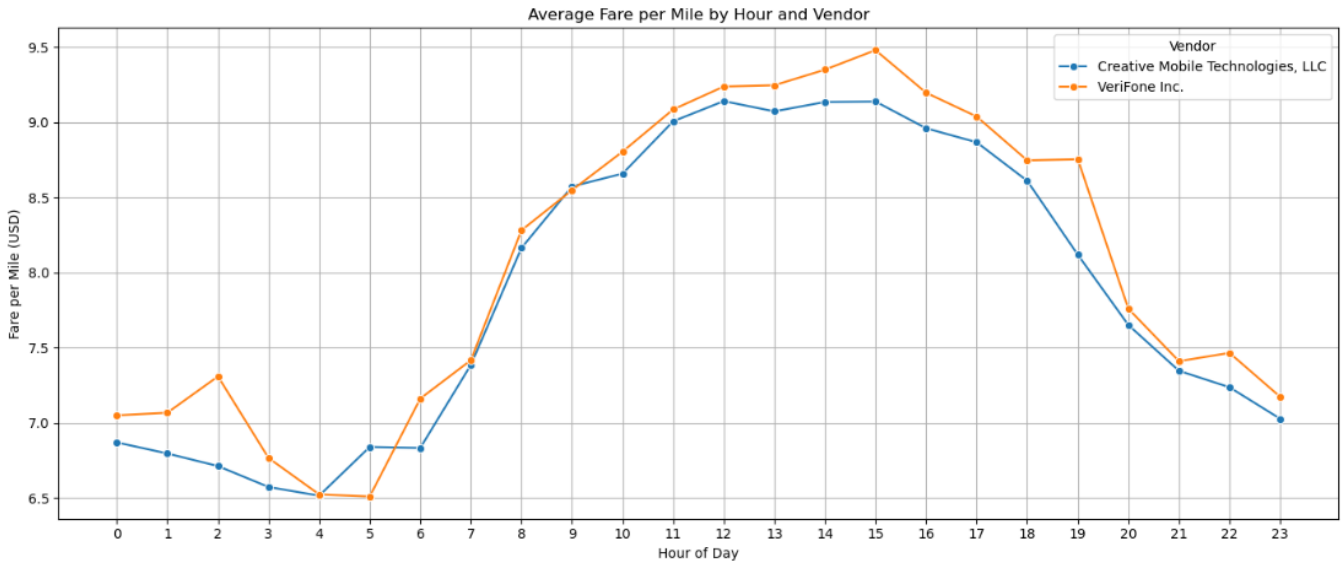


3.2.11. Analyse the average fare per mile for the different vendors

- The data is categorized by vendor and hour of the day, with each trip's fare per mile calculated.
- The vendor_name is mapped to the corresponding VendorID to identify the vendor for each trip.
- The average fare per mile is computed for each vendor across different pickup hours by grouping the data by vendor and hour, and calculating the mean fare per mile.

```
# Compare fare per mile for different vendors
vendor_map = {
    1: 'Creative Mobile Technologies, LLC',
    2: 'VeriFone Inc.'
}
df_merged['vendor_name'] = df_merged['VendorID'].map(vendor_map)
df_merged.head()

vendor_hour_fare = df_merged.groupby(['vendor_name', 'pickup_hour'])['fare_per_mile'].mean().reset_index()
vendor_hour_fare
```

- The results show how the fare per mile varies for each vendor at different times of the day, allowing a comparison of the pricing trends for each vendor.



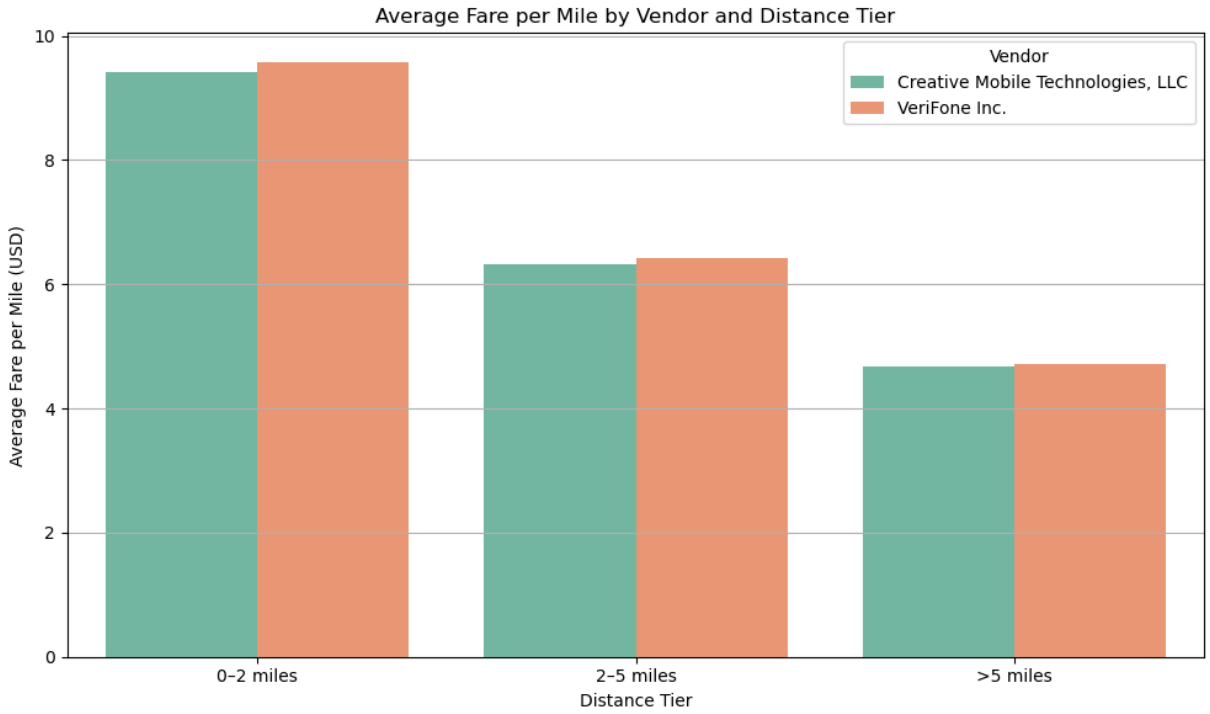
3.2.12. Compare the fare rates of different vendors in a distance-tiered fashion

- The analysis is based on distance tiers to compare fare rates of different vendors, categorizing trips into three groups: distances up to 2 miles, from 2 to 5 miles, and more than 5 miles.
- A function is used to label each trip according to its distance, categorizing them into the respective tiers.

- The average fare per mile is then calculated for each vendor within each distance tier, allowing a comparison of how fare rates differ based on the trip distance.

```
# Defining distance tiers
def label_trip(row):
    if row < 2:
        return '0-2 miles'
    elif row < 5:
        return '2-5 miles'
    else:
        return '>5 miles'

#using qcut to create distance-tier columns
df_merged['tierInfo'] = df_merged['trip_distance'].apply(label_trip)
# df_merged.head()
#
tier_fare = df_merged.groupby(['vendor_name', 'tierInfo'])['fare_per_mile'].mean().reset_index()
tier_fare
```

- The results provide insights into how fare per mile varies for different vendors across the three distance ranges, enabling a tiered comparison of fare rates.

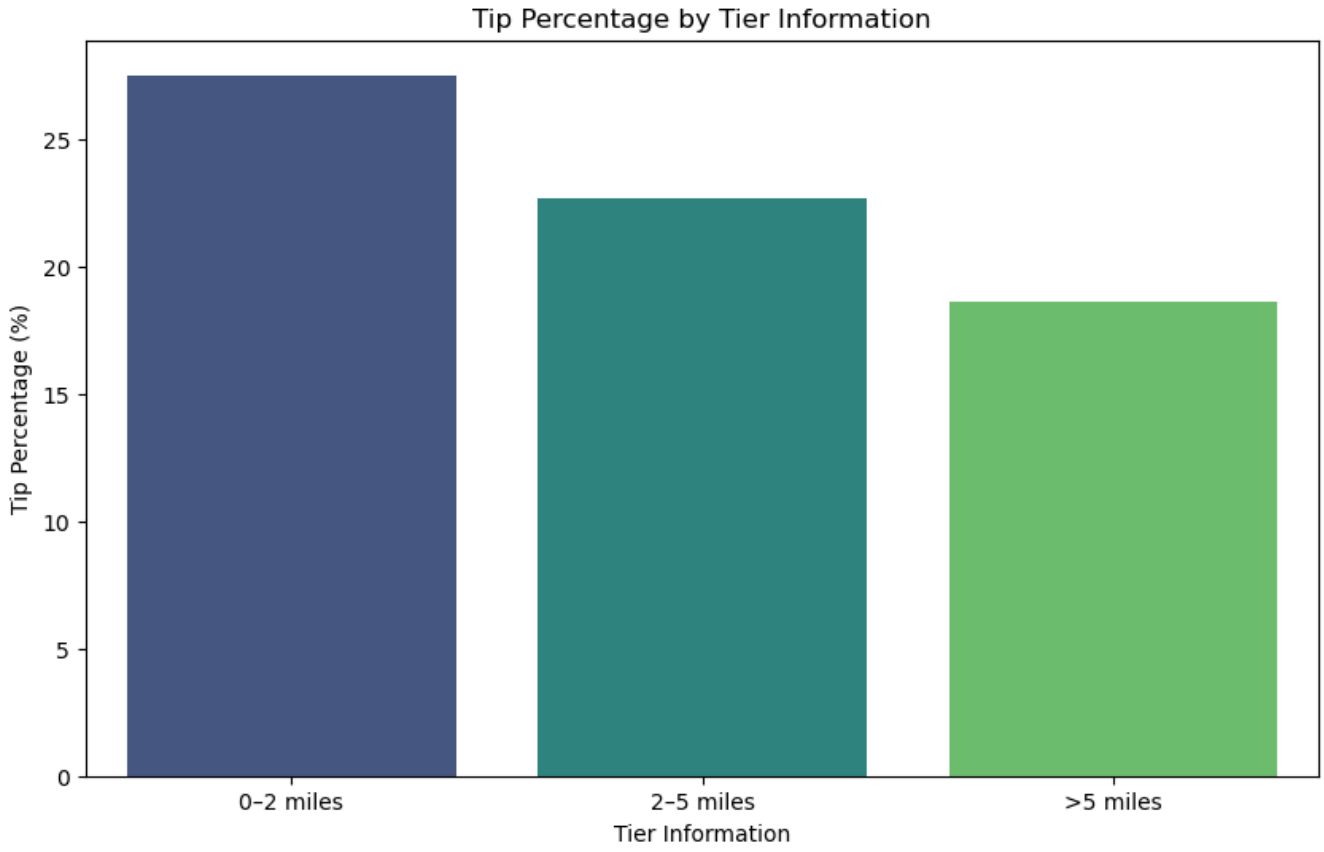


3.2.13. Analyse the tip percentages

- Tip percent vs TierInfo

[566]:

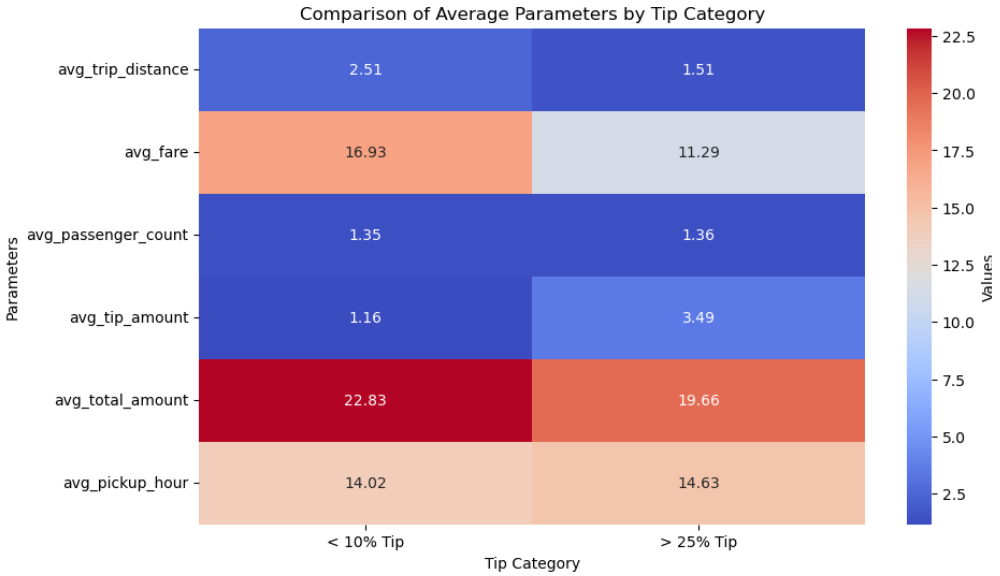
	tierInfo	tip_amount	fare_amount	tip_percentage
0	0–2 miles	2.774326	10.077034	27.531176
1	2–5 miles	4.129941	18.201538	22.690065
2	>5 miles	4.870698	26.185660	18.600631



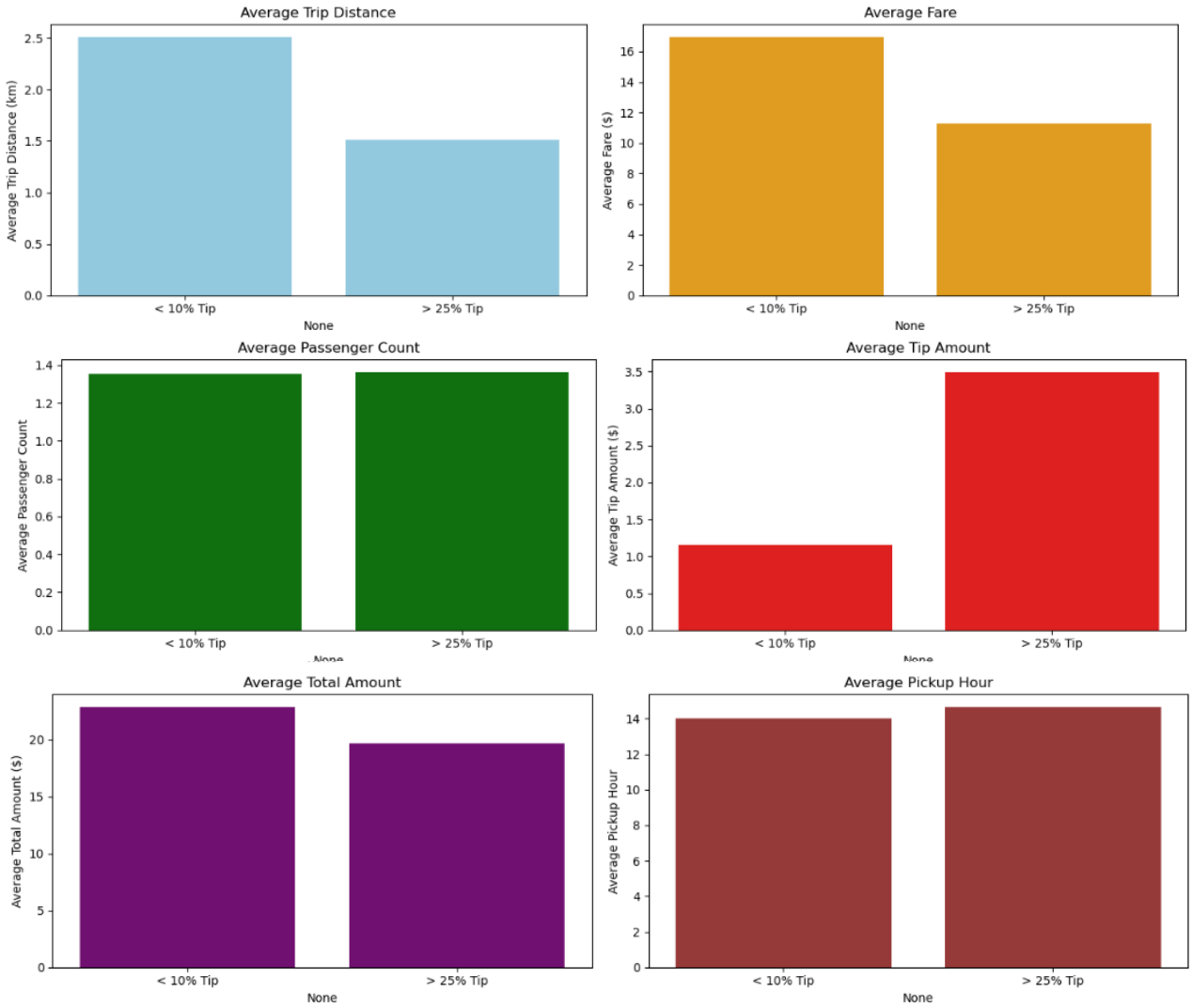
Additionally

- Low tip vs High tip scenario
  - Compare trips with tip percentage < 10% to trips with tip percentage > 25%

	< 10% Tip	> 25% Tip
avg_trip_distance	2.507561	1.514804
avg_fare	16.928838	11.291599
avg_passenger_count	1.353901	1.361684
avg_tip_amount	1.157469	3.487500
avg_total_amount	22.833545	19.655374
avg_pickup_hour	14.018800	14.629268

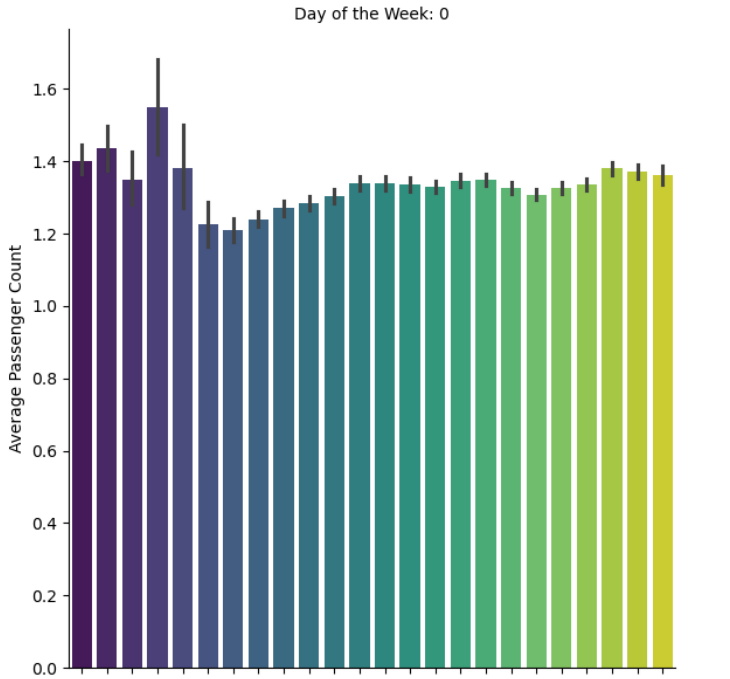


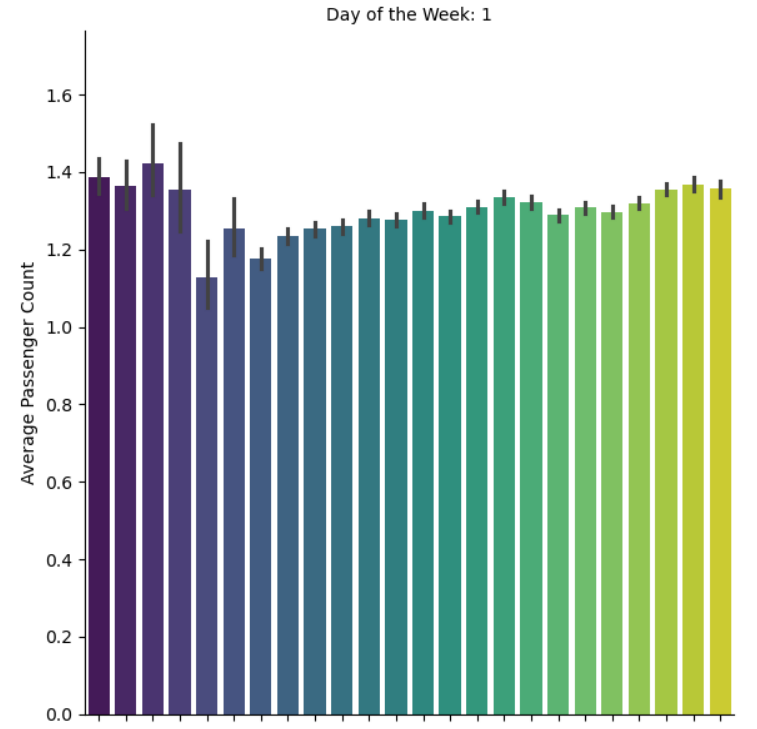
- Average other factors – high and low tip comparison

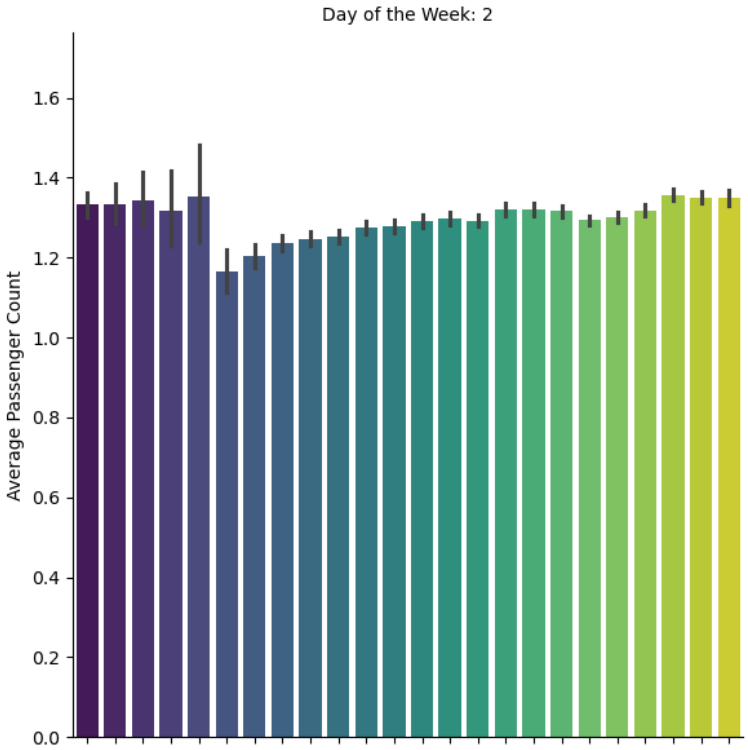


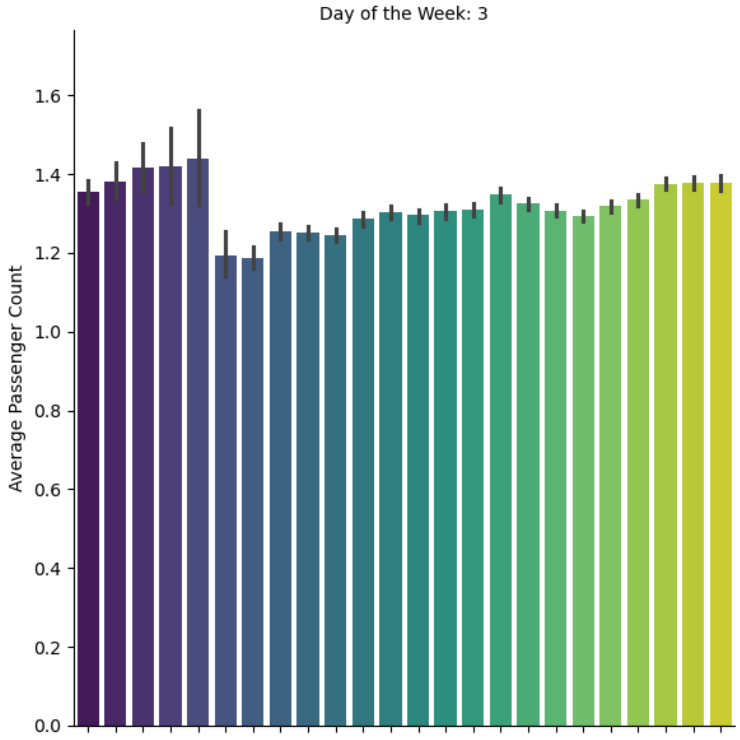
3.2.14. Analyse the trends in passenger count

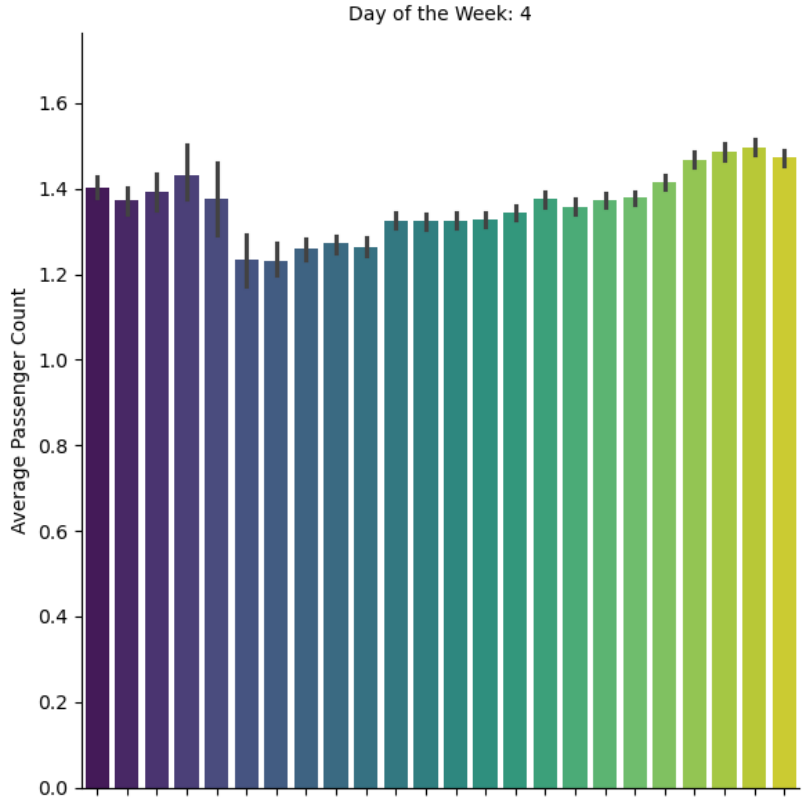
- Analyse the variation of passenger count across hours and days of the week.

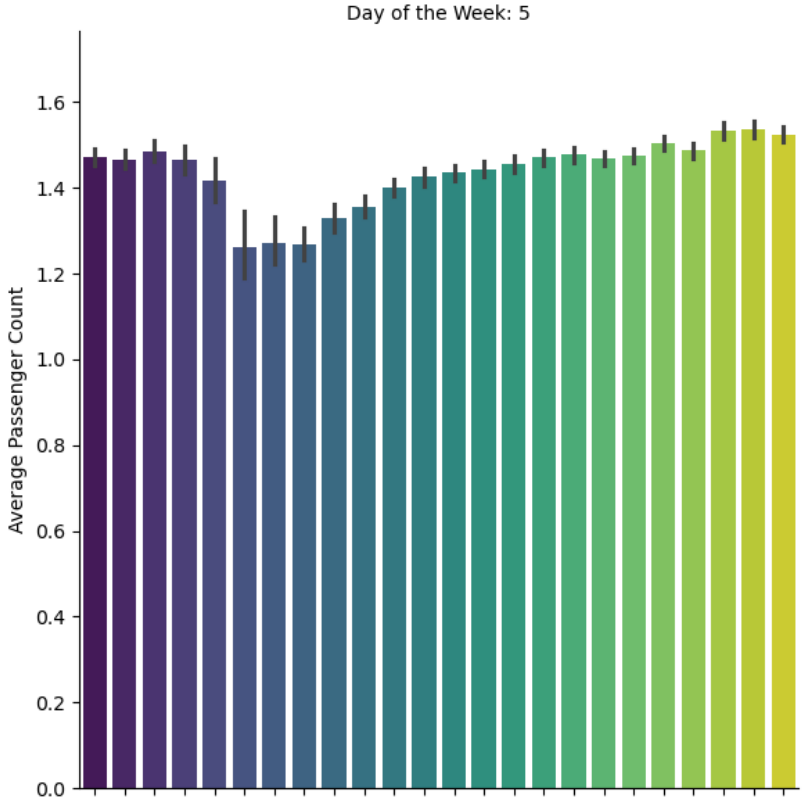


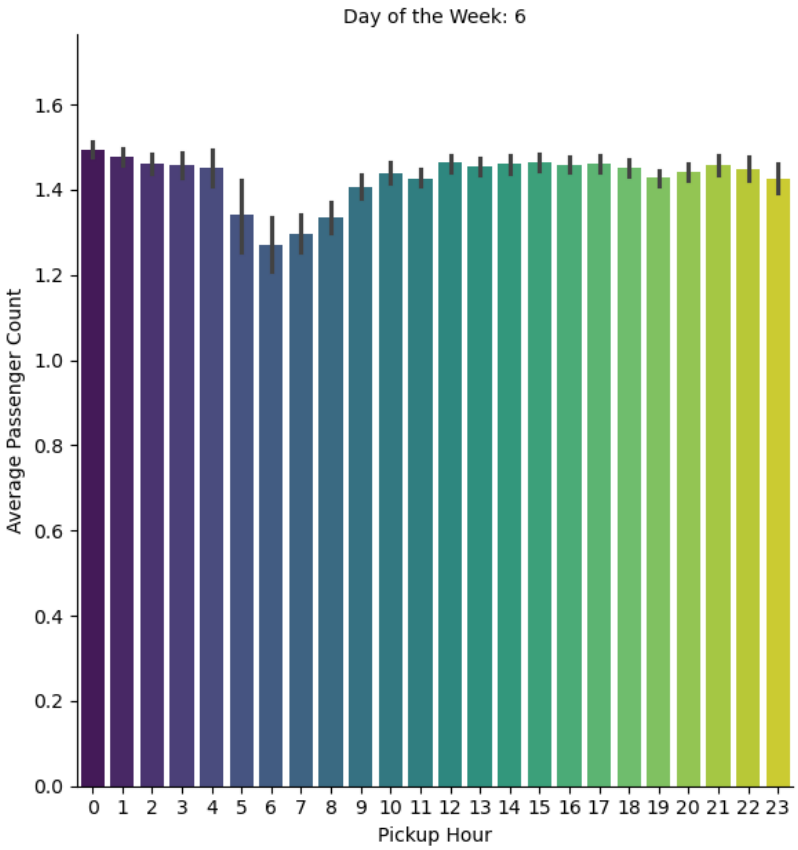



Day of the Week: 2
Average Passenger Count
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6


Day of the Week: 3
Average Passenger Count
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6


Day of the Week: 4
Average Passenger Count
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

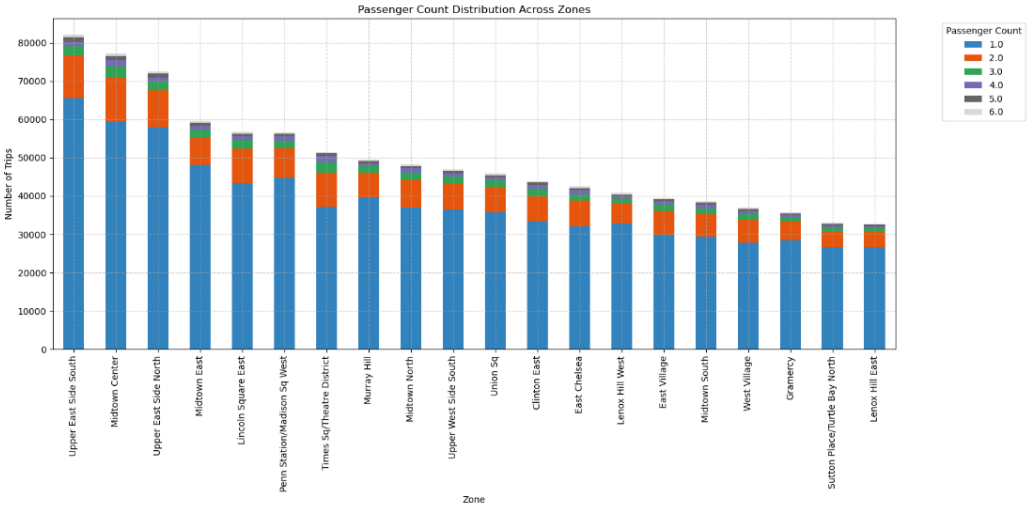

Day of the Week: 5
Average Passenger Count
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

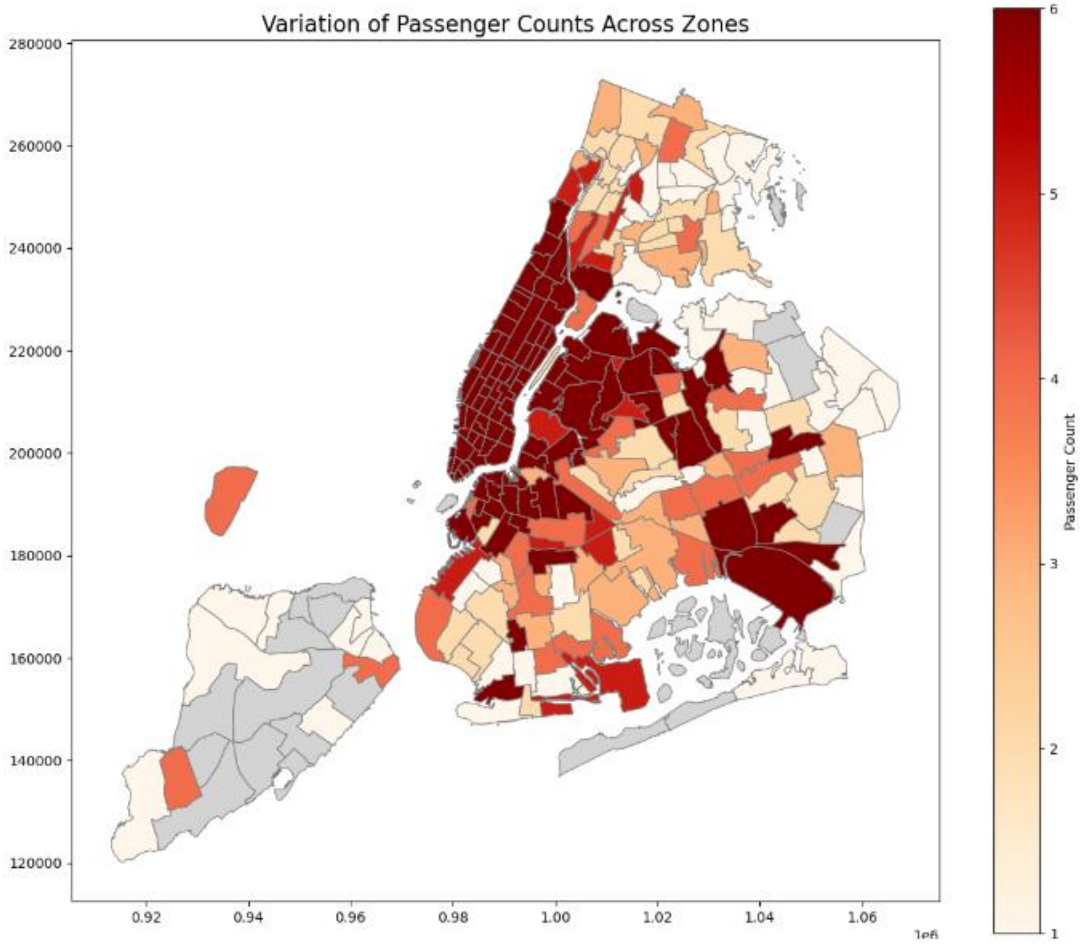


3.2.15. Analyse the variation of passenger counts across zones

- Analyse the variation of passenger counts across zones

  Top 10 zone wise plot





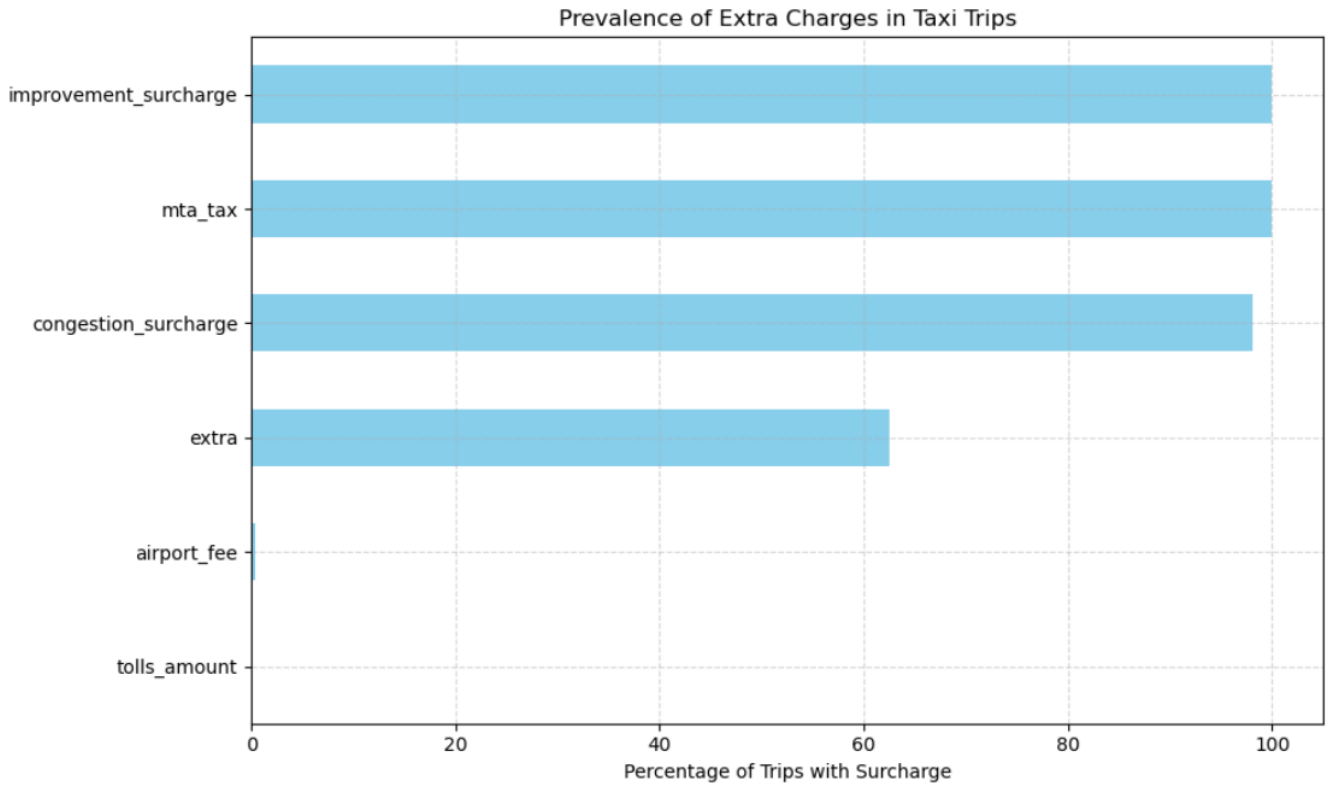
- High Passenger Count Zones: Darker zones likely indicate areas with high passenger counts (more trips or passengers). These might be zones with higher demand, such as central business districts, transportation hubs, or popular neighborhoods.
- Low Passenger Count Zones: Lighter-colored zones suggest areas with lower passenger counts, which could be less densely populated areas, residential zones, or regions with less traffic.
- Identifying Trends: By observing the geographic patterns, you can infer areas where passenger demand is consistently high or low, potentially indicating hotspots or underutilized zones.

3.2.16. Analyse the pickup/dropoff zones or times when extra charges are applied more frequently.

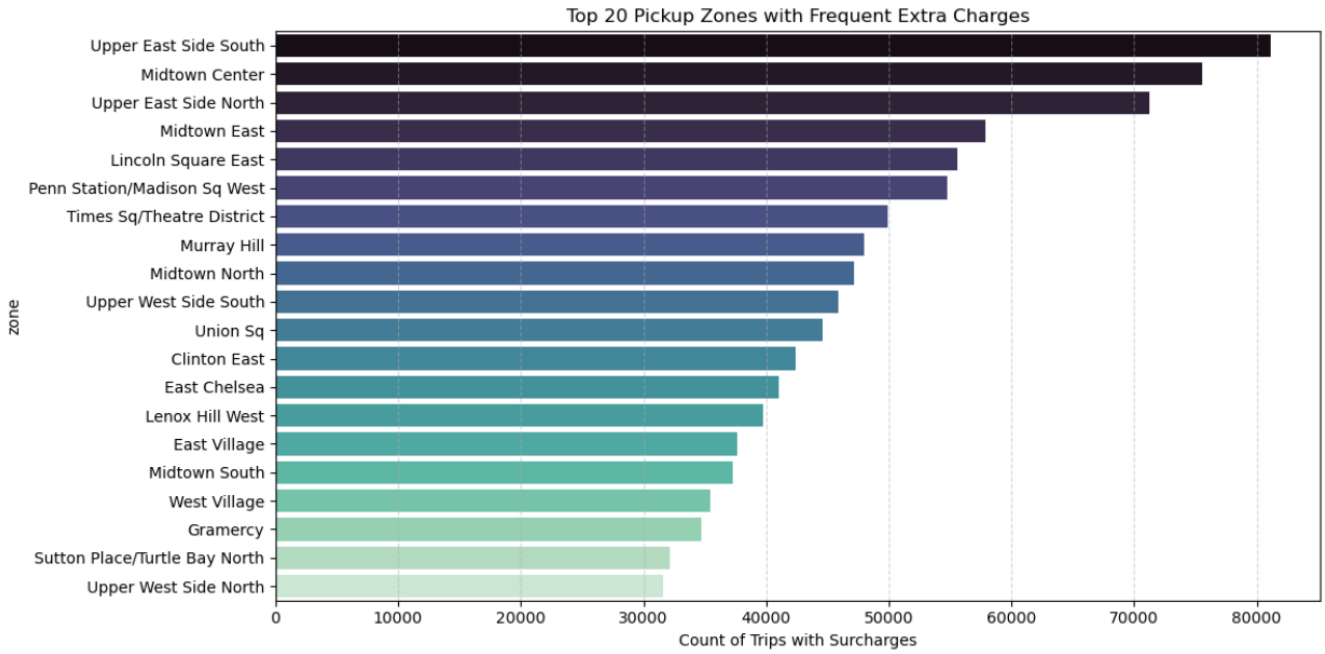
- Find out how often surcharges/extra charges are applied to understand their prevalence

74]:
```
surcharge_stats = {}

total_trips = len(df_merged)

for col in surcharge_cols:
    applied_count = (df_merged[col] > 0).sum()
    surcharge_stats[col] = {
        'Applied Count': applied_count,
        'Percentage (%)': round((applied_count / total_trips) * 100, 2)
    }

surcharge_df = pd.DataFrame(surcharge_stats).T
surcharge_df = surcharge_df.sort_values(by='Percentage (%)', ascending=False)
print(surcharge_df)
```

	Applied Count	Percentage (%)
improvement_surcharge	1135055.0	100.00
mta_tax	1134392.0	99.94
congestion_surcharge	1113784.0	98.13
extra	708951.0	62.46
airport_fee	3816.0	0.34
tolls_amount	1127.0	0.10



- Zones and its surcharges details – top(10)



4. Conclusions

4.1 Final Insights and Recommendations

4.1.1. Recommendations to optimize routing and dispatching based on demand patterns and operational inefficiencies.

- Target High-Demand Zones: Focus taxi availability in Midtown Center, Midtown, Upper East Side North, and Upper East Side South during peak demand hours.
- Optimize Evening Coverage: Demand significantly rises between 4 PM and 8 PM (16:00–20:00), making it essential to concentrate dispatch efforts in these zones during this window.
- Weekday Peak Hour: The 6 PM hour (18:00) stands out as the busiest time on weekdays, indicating a surge in commuter and post-work travel demand.
- Weekend Stability: Unlike weekdays, weekends show a more evenly distributed demand pattern with no significant peak around 6 PM, suggesting a steady but moderate flow of trips throughout the day
- Late-Night Hotspot: East Village consistently records the highest pickup and dropoff activity during the night hours (11 PM to 5 AM), highlighting it as a prime zone for late-night taxi availability.

4.1.2. Suggestions on strategically positioning cabs across different zones to make best use of insights uncovered by analysing trip trends across time, days and months.

- During the morning rush hours (7–10 AM), taxi demand surges as commuters travel from residential areas—like Upper Manhattan and Queens—towards business hubs such as Midtown and Downtown.
- In the evening rush period (4–8 PM), areas like Midtown, the Financial District, and the Upper East and West Sides experience high activity due to office-goers returning home and the early dinner crowd.
- Nighttime hours (9 PM–2 AM) see elevated demand around entertainment zones such as East Village, as well as at major airports, driven by late-night diners, nightlife seekers, and flight arrivals.
- For Midtown, it's essential to maintain a moderate fleet throughout the day, with increased deployment during business hours and special events to handle surges in demand.
- At airports, consistent taxi availability should be ensured during peak flight arrival windows to serve travelers efficiently.
- In residential areas such as Queens and the Bronx, taxis should be strategically dispatched during the morning and early evening hours to meet commuter traffic.
- Entertainment zones benefit from higher cab availability after 9 PM on weekends, catering to short but high-value trips related to nightlife activity.
- Lastly, slow zones—areas prone to congestion and reduced speeds—should generally be avoided during rush hours unless there's a forecasted spike in demand that justifies the inefficiency.

4.1.3. Propose data-driven adjustments to the pricing strategy to maximize revenue while maintaining competitive rates with other vendors.

- High-demand hours (4–8 PM, weekdays) and late-night slots (9 PM–2 AM, weekends) show peak activity, especially in Midtown and entertainment zones.
    - Increase base fare or per-mile rate slightly during these windows.
- Low-demand hours (10 AM–3 PM, weekdays) can be priced lower to attract more rides.
    - Offer fare discounts or promo pricing to improve utilization.
- Trips of 2–5 miles show high fare-per-mile potential.
    - Introduce a tiered pricing model with a slightly reduced per-mile rate for trips >5 miles to encourage longer bookings while balancing fuel/time costs.
- Extra charges like night surcharges and congestion fees are more common in high-traffic zones and night hours.
    - Keep surcharges transparent, but bundle them smartly like “night fare” includes tolls and extras
    - Use flat-rate surcharges instead of unpredictable ones to maintain customer trust.