Report: Optimising 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

1.1.1. Sample the data and combine the files

```
# Select the folder having data files
os.chdir(r'C:\Users\Admin\Desktop\upgrad\NYC Taxi EDA project\Datasets and Dictionary\trip_records')


# Create a list of all the twelve files to read
file_list = os.listdir()

# initialise an empty dataframe
df = pd.DataFrame()

from tqdm import tqdm
# iterate through the list of files and sample one by one:
for file_name in tqdm(file_list):
    try:
        # file path for the current file
        file_path = os.path.join(os.getcwd(), file_name)

        # Reading the current file
        df_month = pd.read_parquet(file_path)
        # Ensure pickup datetime column is datetime type
        df_month['tpep_pickup_datetime'] = pd.to_datetime(df_month['tpep_pickup_datetime'])


        # We will store the sampled data for the current date in this df by appending the sampled data from each hour to this
        # After completing iteration through each date, we will append this data to the final dataframe.
        sampled_data = pd.DataFrame()

        # Loop through dates and then loop through every hour of each date
        unique_dates = df_month['tpep_pickup_datetime'].dt.date.unique()

        for date in unique_dates:
            df_date = df_month[df_month['tpep_pickup_datetime'].dt.date == date]

            # Iterate through each hour of the selected date
            for hour in range(24):
                df_hour = df_date[df_date['tpep_pickup_datetime'].dt.hour == hour]
                if len(df_hour) > 0:
                    # Sample 5% of the hourly data randomly
                    sampled_hour = df_hour.sample(frac=0.007, random_state=42)

                    # add data of this hour to the dataframe
                    sampled_data = pd.concat([sampled_data, sampled_hour])

        # Concatenate the sampled data of all the dates to a single dataframe
        df = pd.concat([df, sampled_data])

    except Exception as e:
        print(f"Error reading file {file_name}: {e}")


print(f"Total sampled records: {len(df)}")
```

```
100%|██████████| 13/13 [14:26<00:00, 66.63s/it]
```

2. Data Cleaning

2.1. Fixing Columns

2.1.1. Fix the index

2.1.1 [2 marks]

Fix the index and drop unnecessary columns

[15]:
```
# Fix the index and drop any columns that are not needed

new_df = new_df.reset_index(drop=True)
```

[16]:
```
new_df.head()
```

[16]:

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID	store_and_fwd_flag	PULocationID	DOLocationID
0	2	2023-01-01 00:07:18	2023-01-01 00:23:15	1.0	7.74	1.0	N	138	256
1	2	2023-01-01 00:16:41	2023-01-01 00:21:46	2.0	1.24	1.0	N	161	237
2	2	2023-01-01 00:14:03	2023-01-01 00:24:36	3.0	1.44	1.0	N	237	141
3	2	2023-01-01 00:24:30	2023-01-01 00:29:55	1.0	0.54	1.0	N	143	142
4	2	2023-01-01 00:43:00	2023-01-01 01:01:00	NaN	19.24	NaN	NaN	66	107

2.1.2. Combine the two airport_fee columns

2.1.2 [3 marks]

There are two airport fee columns. This is possibly an error in naming columns. Let's see whether these can be combined into a single column.

[27]:
```
# Combine the two airport fee columns

new_df['combined_airport_fee'] = new_df['airport_fee'].fillna(0) + new_df['Airport_fee'].fillna(0)
print(new_df['combined_airport_fee'])
```

```
0         1.25
1         0.00
2         0.00
3         0.00
4         0.00
          ...
265482    0.00
265483    0.00
265484    0.00
265485    0.00
265486    0.00
Name: combined_airport_fee, Length: 265487, dtype: float64
```

2.2. Handling Missing Values

2.2.1. Find the proportion of missing values in each column

```
[67]: # Find the proportion of missing values in each column
      print(new_df.isnull().sum())
      print('----------------')
      new_df.isnull().sum() * 100 / len(new_df)
```

```
VendorID                  0
tpep_pickup_datetime      0
tpep_dropoff_datetime     0
passenger_count        8831
trip_distance             0
RatecodeID             8831
store_and_fwd_flag     8831
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   8831
combined_airport_fee      0
dtype: int64
----------------
```

```
[67]: VendorID                 0.00000
      tpep_pickup_datetime     0.00000
      tpep_dropoff_datetime    0.00000
      passenger_count          3.32634
      trip_distance            0.00000
      RatecodeID               3.32634
      store_and_fwd_flag       3.32634
      PULocationID             0.00000
      DOLocationID             0.00000
      payment_type             0.00000
      fare_amount              0.00000
      extra                    0.00000
      mta_tax                  0.00000
      tip_amount               0.00000
      tolls_amount             0.00000
      improvement_surcharge    0.00000
      total_amount             0.00000
      congestion_surcharge     3.32634
      combined_airport_fee     0.00000
      dtype: float64
```

2.2.2. Handling missing values in passenger_count

[70]:
```
new_df['passenger_count'] = new_df['passenger_count'].replace(0, np.nan) # Replace 0 with
```

[71]:
```
print(new_df.isnull().sum())
```

```
VendorID                   0
tpep_pickup_datetime       0
tpep_dropoff_datetime      0
passenger_count        12901
trip_distance              0
RatecodeID              8831
store_and_fwd_flag      8831
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    8831
combined_airport_fee       0
dtype: int64
```

First replaced ‘0’ passenger_count to NaN, this increased the null values in passenger_count and then handled the null values.

[78]:
```
new_df['passenger_count'] = new_df['passenger_count'].fillna(df['passenger_count'].mode()[0])
new_df.isnull().sum()
```

[78]:
```
VendorID                   0
tpep_pickup_datetime       0
tpep_dropoff_datetime      0
passenger_count            0
trip_distance              0
RatecodeID              8831
store_and_fwd_flag      8831
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    8831
combined_airport_fee       0
dtype: int64
```

2.2.3. Handle missing values in RatecodeID

[79]:
```
# Fix missing values in 'RatecodeID'
new_df['RatecodeID'].value_counts()
```

[79]:
```
RatecodeID
1.0     242253
2.0      10081
99.0      1530
5.0       1428
3.0        845
4.0        519
Name: count, dtype: int64
```

[80]:
```
# since RatecodeID is a categorical column so imputing the NaN with Mode
```

[81]:
```
new_df['RatecodeID'] = new_df['RatecodeID'].fillna(df['RatecodeID'].mode(
new_df.isnull().sum()
```

[81]:
```
VendorID                    0
tpep_pickup_datetime        0
tpep_dropoff_datetime       0
passenger_count             0
trip_distance               0
RatecodeID                  0
store_and_fwd_flag       8831
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     8831
combined_airport_fee        0
dtype: int64
```

2.2.4. Impute NaN in congestion_surcharge

2.2.4 [3 marks]

Impute NaN in `congestion_surcharge`

[84]:
```
# handle null values in congestion_surcharge

new_df['congestion_surcharge'].value_counts()
```

[84]:
```
congestion_surcharge
2.5    236944
0.0     19712
Name: count, dtype: int64
```

[85]:
```
new_df['congestion_surcharge'] = new_df['congestion_surcharge'].fillna(df['congestion_surcharge'].mode
new_df.isnull().sum()
```

[85]:
```
VendorID                0
tpep_pickup_datetime    0
tpep_dropoff_datetime   0
passenger_count         0
trip_distance           0
RatecodeID              0
store_and_fwd_flag      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
combined_airport_fee    0
dtype: int64
```

2.3. Handling Outliers and Standardising Values

2.3.1. Check outliers in payment type, trip distance and tip amount columns

[89]:
```
# Entries where trip_distance is nearly 0 and fare_amount is more than 300
outliers = new_df[(new_df['trip_distance'] < 0.1) & (new_df['fare_amount'] > 30
print(outliers)
print('\n')
print('\n')
print("Before:", new_df.shape)
# droping these rows
new_df = new_df[~((new_df['trip_distance'] < 0.1) & (new_df['fare_amount'] > 30
print('\n')
print('\n')
print("After:", new_df.shape)
```

```
        VendorID tpep_pickup_datetime tpep_dropoff_datetime  passenger_count  \
74623          2  2023-12-07 23:39:43   2023-12-07 23:39:59              1.0
131538         2  2023-06-21 12:05:21   2023-06-21 12:05:42              1.0
137940         2  2023-06-29 20:56:07   2023-06-29 20:56:15              1.0
164032         1  2023-02-09 07:37:30   2023-02-09 07:39:13              1.0
182736         2  2023-04-05 21:16:43   2023-04-05 21:25:57              1.0

        trip_distance  RatecodeID store_and_fwd_flag  PULocationID  \
74623             0.0         5.0                  N           265
131538            0.0         5.0                  N           265
137940            0.0         5.0                  N           265
164032            0.0         5.0                  N           246
182736            0.0         5.0                  N           265

        DOLocationID  payment_type  fare_amount  extra  mta_tax  tip_amount  \
74623            265             2        319.0    0.0      0.0         0.0
131538           265             2        500.0    0.0      0.0         0.0
137940           265             1        350.0    0.0      0.0        70.2
164032           246             4        910.0    0.0      0.0         0.0
182736           265             2        600.0    0.0      0.0         0.0

        tolls_amount  improvement_surcharge  total_amount  \
74623            0.0                    1.0         320.0
131538           0.0                    1.0         501.0
137940           0.0                    1.0         421.2
164032           0.0                    1.0         911.0
182736           0.0                    1.0         601.0

        congestion_surcharge  combined_airport_fee
74623                    0.0                   0.0
131538                   0.0                   0.0
137940                   0.0                   0.0
164032                   0.0                   0.0
182736                   0.0                   0.0
```

```
After: (265482, 19)
```

[90]:
```
# Continue with outlier handling
# Entries where trip_distance and fare_amount are 0
# but the pickup and dropoff zones are different (both distance and fare should not be zero for different zone
mask = (
    (new_df['trip_distance'] == 0) &
    (new_df['fare_amount'] == 0) &
    (new_df['PULocationID'] != new_df['DOLocationID'])   # pickup vs dropoff
)

bad_records = new_df[mask]
print("Bad records found:", bad_records.shape[0])
print("Before:", new_df.shape)
```

```
Bad records found: 4
Before: (265482, 19)
```

[91]:
```
# dropping these
new_df = new_df[~mask]
# checking the shape now
print("After:", new_df.shape)
```

```
After: (265478, 19)
```

[92]:
```
# Entries where trip_distance is more than 250 miles.
mask1 = new_df['trip_distance'] > 250
long_trips = new_df[mask1]

print("Trips > 250 miles:", long_trips.shape[0])
print('\n')
print("Before:", new_df.shape)
```

```
Trips > 250 miles: 3


Before: (265478, 19)
```

[93]:
```
# dropping these columns
new_df = new_df[~mask1]
print("After:", new_df.shape)
```

```
After: (265475, 19)
```

[94]:
```
invalid_payment = new_df[new_df['payment_type'] == 0]
print("Invalid payment_type=0 entries:", invalid_payment.shape[0])
```

```
Invalid payment_type=0 entries: 8828
```

[95]:
```
# Entries where payment_type is 0 (there is no payment_type 0 defined in the data dictionary
print(new_df['payment_type'].value_counts())
print('\n')
invalid_payment = new_df[new_df['payment_type'] == 0]
print("Invalid payment_type=0 entries:", invalid_payment.shape[0])
```

```
payment_type
1    209067
2     44450
0      8828
4      1910
3      1220
Name: count, dtype: int64


Invalid payment_type=0 entries: 8828
```

[96]:
```
# checking there share
share = 8828 / len(new_df) * 100
print(f"Invalid payment_type=0 share: {share:.2f}%")
print('\n')
print("Before:", new_df.shape)
```

```
Invalid payment_type=0 share: 3.33%


Before: (265475, 19)
```

[97]:
```
# dropping them as its only 3.33%
new_df = new_df[new_df['payment_type'] != 0]
print("After:", new_df.shape)
```

```
After: (256647, 19)
```

3. Exploratory Data Analysis

3.1. General EDA: Finding Patterns and Trends

3.1.1. Classify variables into categorical and numerical

3.1.1 [3 marks]

Categorise the varaibles into Numerical or Categorical.

- `VendorID` : categorical
- `tpep_pickup_datetime` : categorical
- `tpep_dropoff_datetime` : categorical
- `passenger_count` : categorical
- `trip_distance` : numerical
- `RatecodeID` : categorical
- `PULocationID` : categorical
- `DOLocationID` : categorical
- `payment_type` : categorical
- `pickup_hour` : numerical
- `trip_duration` : numerical

The following monetary parameters belong in the same category, is it categorical or numer

- `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

[105]:
```
# Find and show the daily trends in taxi pickups (days of the week)
new_df['pickup_day'] = new_df['tpep_pickup_datetime'].dt.day_name()
```

[106]:
```
daily_trends = new_df.groupby('pickup_day').size()
print(daily_trends)
```

```
pickup_day
Friday       38068
Monday       32032
Saturday     37241
Sunday       32283
Thursday     40285
Tuesday      37250
Wednesday    39488
dtype: int64
```

[107]:
```
# Show the monthly trends in pickups
new_df['month'] = new_df['tpep_pickup_datetime'].dt.month
```

[108]:
```
monthly_trends = new_df.groupby('month').size()
print(monthly_trends)
```

```
month
1     20782
2     19695
3     23002
4     22218
5     23697
6     22314
7     19558
8     18968
9     18766
10    23394
11    22168
12    22085
dtype: int64
```

[103]:
```
# Find and show the hourly trends in taxi pickups
new_df['pickup_hour'] = new_df['tpep_pickup_datetime'].dt.hour
```

[104]:
```
hourly_trends = new_df.groupby('pickup_hour').size()
print(hourly_trends)
```

```
pickup_hour
0      7168
1      4798
2      3148
3      2049
4      1345
5      1384
6      3380
7      6889
8      9563
9     10972
10    11993
11    13052
12    14163
13    14615
14    15641
15    16039
16    16006
17    17312
18    18080
19    16234
20    14532
21    14449
22    13340
23    10495
dtype: int64
```

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

[109]:
```
# Analyse the above parameters

# Columns to check
columns_to_check = ['fare_amount', 'tip_amount', 'total_amount', 'trip_distance']

print("Checking for negative values:")
for col in columns_to_check:
    negative_count = (new_df[col] < 0).sum()
    if negative_count > 0:
        print(f"Column '{col}' contains {negative_count} negative value(s).")
    else:
        print(f"Column '{col}' contains no negative values.")

print("\nChecking for zero values:")
for col in columns_to_check:
    zero_count = (new_df[col] == 0).sum()
    if zero_count > 0:
        print(f"Column '{col}' contains {zero_count} zero value(s).")
    else:
        print(f"Column '{col}' contains no zero values.")
```

```
Checking for negative values:
Column 'fare_amount' contains no negative values.
Column 'tip_amount' contains no negative values.
Column 'total_amount' contains no negative values.
Column 'trip_distance' contains no negative values.

Checking for zero values:
Column 'fare_amount' contains 85 zero value(s).
Column 'tip_amount' contains 57400 zero value(s).
Column 'total_amount' contains 29 zero value(s).
Column 'trip_distance' contains 3171 zero value(s).
```

[110]:
```
# droping rows in df where trip_distance is equal to zero
# new_df = new_df[new_df['trip_distance'] != 0]
```

[111]:
```
# droping rows in df where trip_distance is equal to zero
new_df = new_df[new_df['fare_amount'] != 0]
```

[112]:
```
# Columns to check
columns_to_check = ['fare_amount', 'tip_amount', 'total_amount', 'trip_distance']

print("Checking for negative values:")
for col in columns_to_check:
    negative_count = (new_df[col] < 0).sum()
    if negative_count > 0:
        print(f"Column '{col}' contains {negative_count} negative value(s).")
    else:
        print(f"Column '{col}' contains no negative values.")

print("\nChecking for zero values:")
for col in columns_to_check:
    zero_count = (new_df[col] == 0).sum()
    if zero_count > 0:
        print(f"Column '{col}' contains {zero_count} zero value(s).")
    else:
        print(f"Column '{col}' contains no zero values.")
```

```
Checking for negative values:
Column 'fare_amount' contains no negative values.
Column 'tip_amount' contains no negative values.
Column 'total_amount' contains no negative values.
Column 'trip_distance' contains no negative values.

Checking for zero values:
Column 'fare_amount' contains no zero values.
Column 'tip_amount' contains 57322 zero value(s).
Column 'total_amount' contains no zero values.
Column 'trip_distance' contains 3138 zero value(s).
```

[113]:
```
new_df.shape
```

[113]:
```
(256562, 22)
```

```
# Filter out rows where trip_distance is 0
df_filtered =new_df[new_df['trip_distance'] != 0].copy()
```

tip_amount can have 0 as a value as tip is optional for a passenger to give.

3.1.4. Analyse the monthly revenue trends

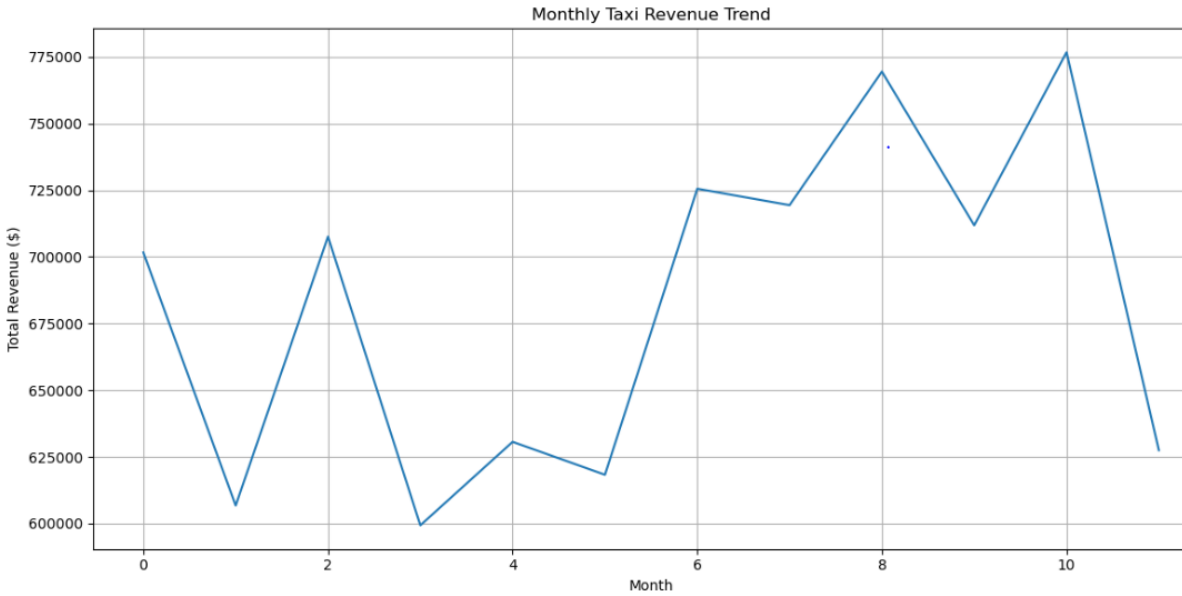
Trends in revenue collected

[150]:
```
# revenue is total of amount collected
# Calculate total revenue per trip
df_filtered["total_revenue"] = (
    df_filtered["tip_amount"] +
    df_filtered["total_amount"]
)
# Daily revenue
daily_revenue = (
    df_filtered.groupby(df_filtered["tpep_pickup_datetime"].dt.date)["total_revenue"]
    .sum()
    .reset_index()
)
# monthly revenue
monthly_revenue = (
    df_filtered.groupby(df_filtered["month_name"])["total_revenue"]
    .sum()
    .reset_index()
)
# quaterly revenue
quarterly_revenue = (
    df_filtered.groupby(df_filtered["quarter"])["total_revenue"]
    .sum()
    .reset_index()
)
print(daily_revenue)
print('\n')
print(monthly_revenue)
print('\n')
print(quarterly_revenue)
```

```
    tpep_pickup_datetime  total_revenue
0             2023-01-01       16821.14
1             2023-01-02       15231.45
2             2023-01-03       19392.72
3             2023-01-04       19751.74
4             2023-01-05       20929.47
..                   ...            ...
358           2023-12-27       18305.21
359           2023-12-28       19216.42
360           2023-12-29       19198.08
361           2023-12-30       17156.19
362           2023-12-31       15602.34
```

Revenue peaks in summer through early fall, with the strongest months clustered around June–October.

The softest revenue month is February, with a gradual build starting in March.



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

3.1.5 [3 marks]

Show the proportion of each quarter of the year in the revenue

[117]:
```
# Calculate proportion of each quarter

#  Extract the quarter
new_df['quarter'] = new_df['tpep_pickup_datetime'].dt.quarter
```

[118]:
```
#   Calculate total revenue per quarter
quarterly_revenue = new_df.groupby('quarter')['total_amount'].sum()
```

[119]:
```
#  Calculate total annual revenue
total_annual_revenue = quarterly_revenue.sum()
```

[120]:
```
#  Calculate the proportion
quarterly_proportion = quarterly_revenue / total_annual_revenue
```

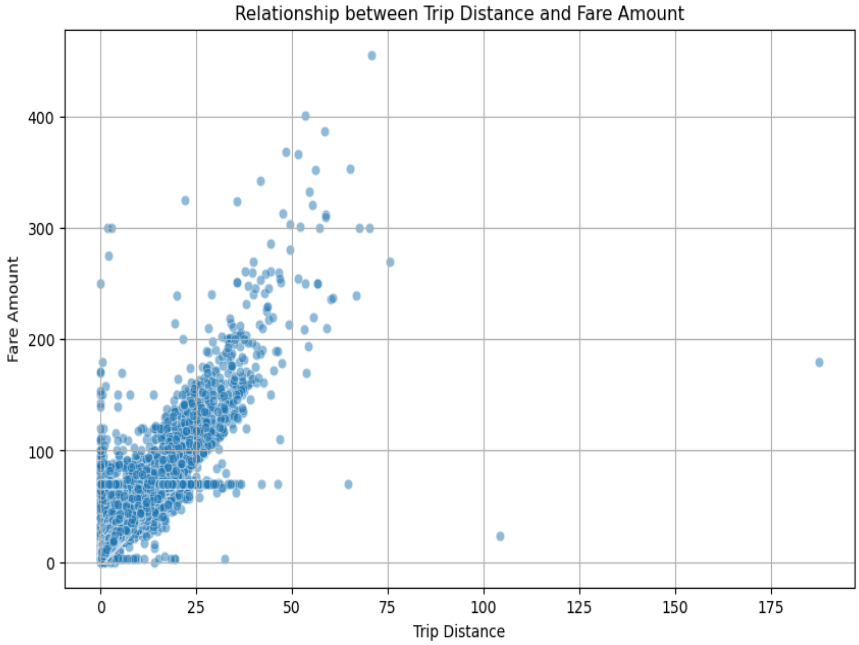
[121]:
```
print("Quarterly Revenue Proportions:")
print(quarterly_proportion)
```

```
Quarterly Revenue Proportions:
quarter
1    0.237372
2    0.267930
3    0.227327
4    0.267371
Name: total_amount, dtype: float64
```

3.1.6. Analyse and visualise the relationship between distance and fare amount

- Relationship shape: Fare amount increases roughly linearly with trip distance, with a positive slope that reflects the per-mile component plus base fare.
- Short-distance noise: Very short trips show wider variance due to base fare, minimum fare rules, and extras/tolls dominating total.
- Long-distance outliers: A few high-fare points at longer distances are driven by tolls, airport trips, and occasional surcharges.

[122]:
```
# Show how trip fare is affected by distance

# Filter out rows where trip_distance is 0
df_filtered =new_df[new_df['trip_distance'] != 0].copy()

plt.figure(figsize=(10, 6))
sns.scatterplot(x='trip_distance', y='fare_amount', data=df_filtered, alpha=0.5)
plt.title('Relationship between Trip Distance and Fare Amount')
plt.xlabel('Trip Distance')
plt.ylabel('Fare Amount')
plt.grid(True)
plt.show()
```



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

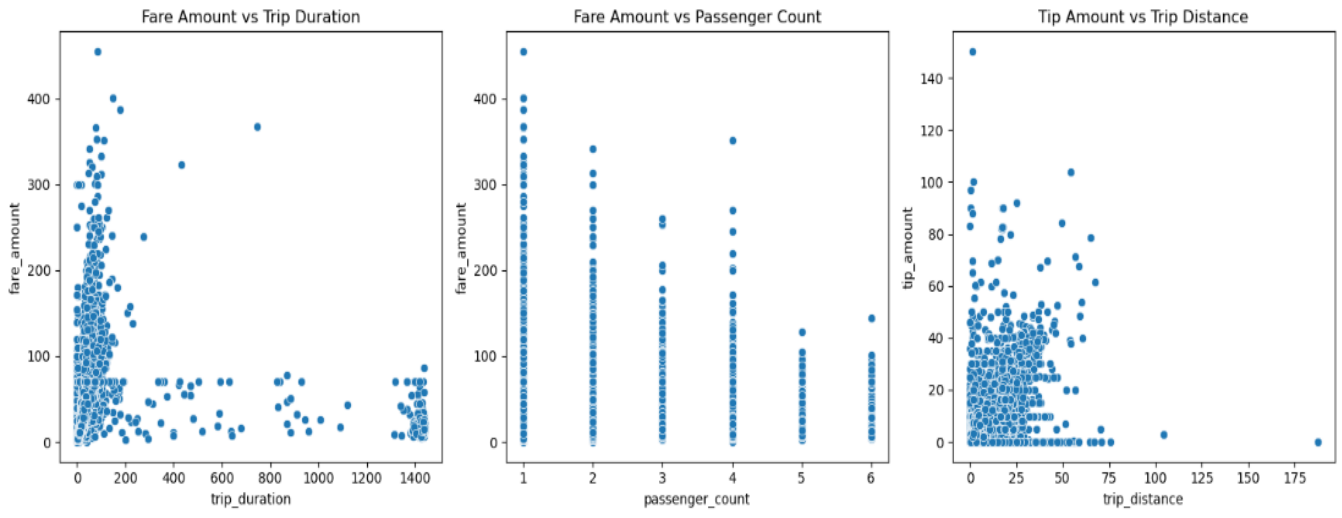
1. `fare_amount` and trip duration (pickup time to dropoff time)
2. `fare_amount` and `passenger_count`
3. `tip_amount` and `trip_distance`

[125]:
```
df_filtered['tpep_pickup_datetime'] = pd.to_datetime(df_filtered['tpep_pickup_datetime'])
df_filtered['tpep_dropoff_datetime'] = pd.to_datetime(df_filtered['tpep_dropoff_datetime'])
# Calculate Trip Duration
df_filtered['trip_duration'] = (df_filtered['tpep_dropoff_datetime'] - df_filtered['tpep_pickup_datetime']).dt.seconds / 60
```

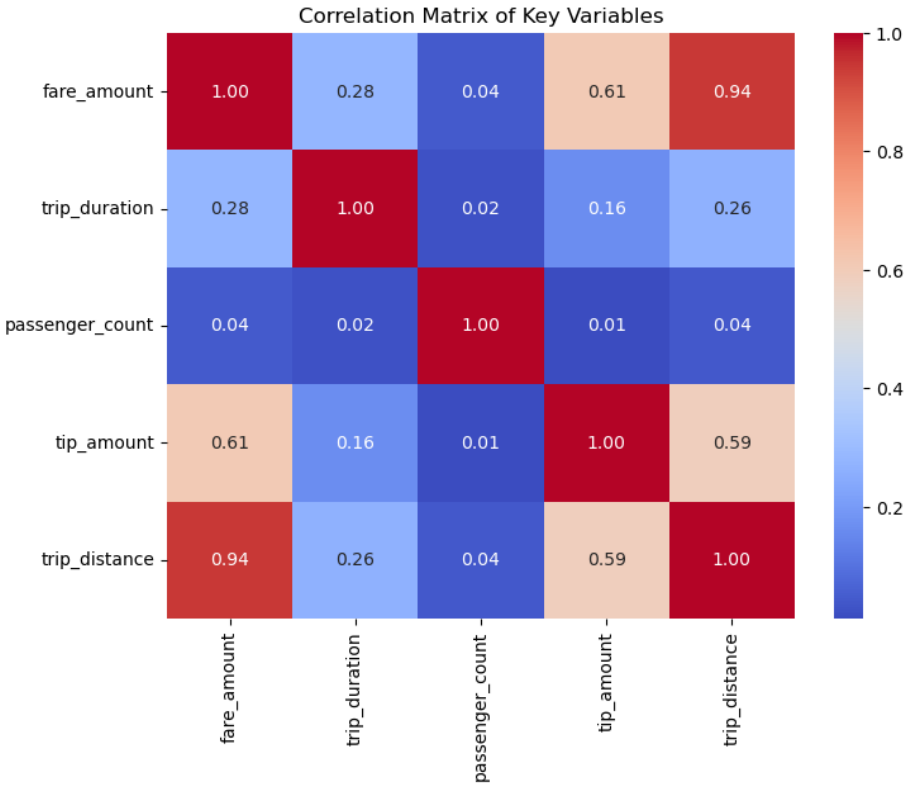
[126]:
```
# Show relationship between fare and trip duration
# Show relationship between fare and number of passengers
# Show relationship between tip and trip distance
# Calculate Correlations
correlation_fare_duration = df_filtered['fare_amount'].corr(df_filtered['trip_duration'])
correlation_fare_passenger = df_filtered['fare_amount'].corr(df_filtered['passenger_count'])
correlation_tip_distance = df_filtered['tip_amount'].corr(df_filtered['trip_distance'])

print(f"Correlation between fare_amount and trip_duration: {correlation_fare_duration:.2f}")
print(f"Correlation between fare_amount and passenger_count: {correlation_fare_passenger:.2f}")
print(f"Correlation between tip_amount and trip_distance: {correlation_tip_distance:.2f}")
```

```
Correlation between fare_amount and trip_duration: 0.28
Correlation between fare_amount and passenger_count: 0.04
Correlation between tip_amount and trip_distance: 0.59
```

[127]:
```
# Scatter plot


plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
sns.scatterplot(x='trip_duration', y='fare_amount', data=df_filtered)
plt.title('Fare Amount vs Trip Duration')

plt.subplot(1, 3, 2)
sns.scatterplot(x='passenger_count', y='fare_amount', data=df_filtered)
plt.title('Fare Amount vs Passenger Count')

plt.subplot(1, 3, 3)
sns.scatterplot(x='trip_distance', y='tip_amount', data=df_filtered)
plt.title('Tip Amount vs Trip Distance')

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



[128]:
```
# Create a sub-dataframe with relevant columns for a heatmap

correlation_matrix_data = df_filtered[['fare_amount', 'trip_duration', 'passenger_count', 'tip_amount', 'trip_distance']]
correlation_matrix = correlation_matrix_data.corr()

plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix of Key Variables')
plt.show()
```



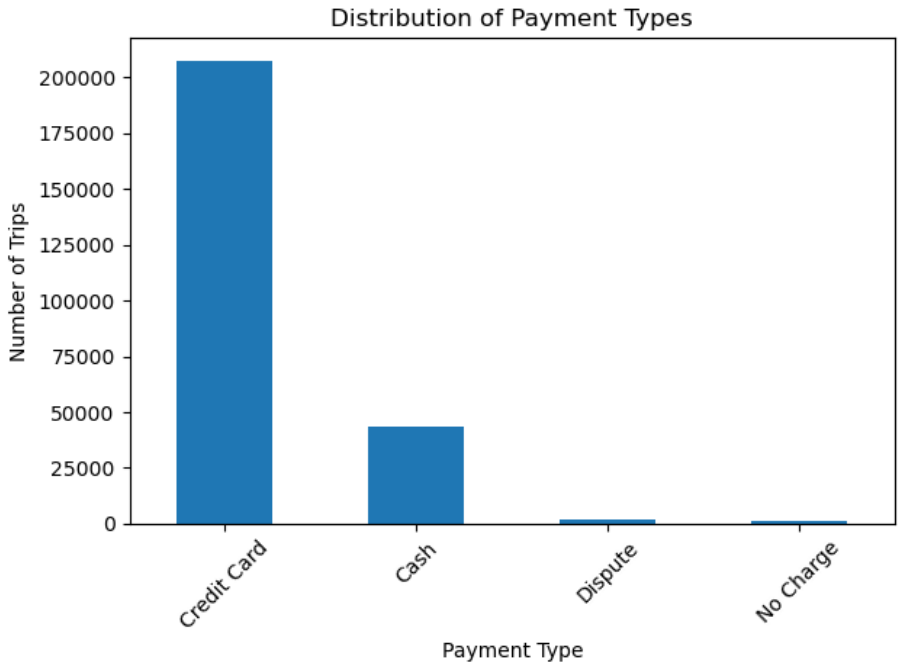
3.1.8 Analyse the distribution of different payment types

Credit card is the dominant payment type across the dataset, comprising the clear majority of trips.

Cash is the second most common method, materially lower than credit but still significant, with a higher share in late-night and shorter urban trips.

Digital wallet/other electronic types are present but comparatively small.

[131]:
```
payment_counts.plot(kind='bar')
plt.title('Distribution of Payment Types')
plt.xlabel('Payment Type')
plt.ylabel('Number of Trips')
plt.xticks(rotation=45) # Rotating x-axis labels for better readability
plt.tight_layout() # Adjusting layout to prevent labels from overlapping
plt.show()
```



3.1.9. Load the taxi zones shapefile and display it

133]:
```
# import geopandas as gpd
# done above
# Read the shapefile using geopandas


shapefile_path = r"C:\Users\Admin\Desktop\upgrad\NYC Taxi EDA project\Datasets and Dictionary\taxi_zones\taxi_zones.shp"
zones = gpd.read_file(shapefile_path)
zones.head()
```

133]:

	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...

Now, if you look at the DataFrame created, you will see columns like: `OBJECTID`, `Shape_Leng`, `Shape_Area`, `zone`, `LocationID`, `borough`, `geometry`.

3.1.10. Merge the zone data with trips data

3.1.10 [3 marks]

Merge the zones data into trip data using the `locationID` and `PULocationID` columns.

5]:
```
# Merge zones and trip records using locationID and PULocationID

merged_df = pd.merge(zones, df_filtered, left_on='LocationID', right_on='PULocationID', how='left')
merged_df.head()
```

5]:

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID	borough	geometry	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	...	total_ar
0	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...	2.0	2023-03-11 16:06:04	2023-03-11 16:06:18	...	
1	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...	2.0	2023-06-11 11:38:07	2023-06-11 11:40:04	...	
2	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...	2.0	2023-08-01 13:26:32	2023-08-01 13:26:43	...	
3	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...	2.0	2023-02-18 20:24:24	2023-02-18 20:24:28	...	
4	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...	2.0	2023-05-13 06:36:35	2023-05-13 06:36:42	...	

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

3.1.11 [3 marks]

Group data by location IDs to find the total number of trips per location ID

[137]:
```
merged_df['PULocationID'].value_counts(ascending = True)
```

[137]:
```
PULocationID
5.0          1
31.0         1
46.0         1
44.0         1
118.0        1
           ...
162.0     9231
236.0    10753
161.0    11959
237.0    12068
132.0    13323
Name: count, Length: 232, dtype: int64
```

[138]:
```
# Group data by location and calculate the number of trips
trip_counts = merged_df.groupby("PULocationID").size().reset_index(name="Total Trips")

print(trip_counts)
```

```
     PULocationID  Total Trips
0             1.0            5
1             3.0            7
2             4.0          242
3             5.0            1
4             6.0            3
..            ...          ...
227         259.0            3
228         260.0           37
229         261.0         1349
230         262.0         3252
231         263.0         4847

[232 rows x 2 columns]
```

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

3.1.12 [2 marks]

Now, use the grouped data to add number of trips to the GeoDataFrame.

We will use this to plot a map of zones showing total trips per zone.

[139]:
```
# Renaming for clarity
trip_counts = trip_counts.rename(columns={"PULocationID": "LocationID"})
```

[140]:
```
# Merge trip counts back to the zones GeoDataFrame
zones_with_counts = zones.merge(
    trip_counts,
    on="LocationID",
    how="left"   # keeping all zones even if no trips
)
```

[141]:
```
zones_with_counts.head()
```

[141]:

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID	borough	geometry	Total Trips
0	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...	5.0
1	2	0.433470	0.004866	Jamaica Bay	2	Queens	MULTIPOLYGON (((1033269.244 172126.008, 103343...	NaN
2	3	0.084341	0.000314	Allerton/Pelham Gardens	3	Bronx	POLYGON ((1026308.77 256767.698, 1026495.593 2...	7.0
3	4	0.043567	0.000112	Alphabet City	4	Manhattan	POLYGON ((992073.467 203714.076, 992068.667 20...	242.0
4	5	0.092146	0.000498	Arden Heights	5	Staten Island	POLYGON ((935843.31 144283.336, 936046.565 144...	1.0

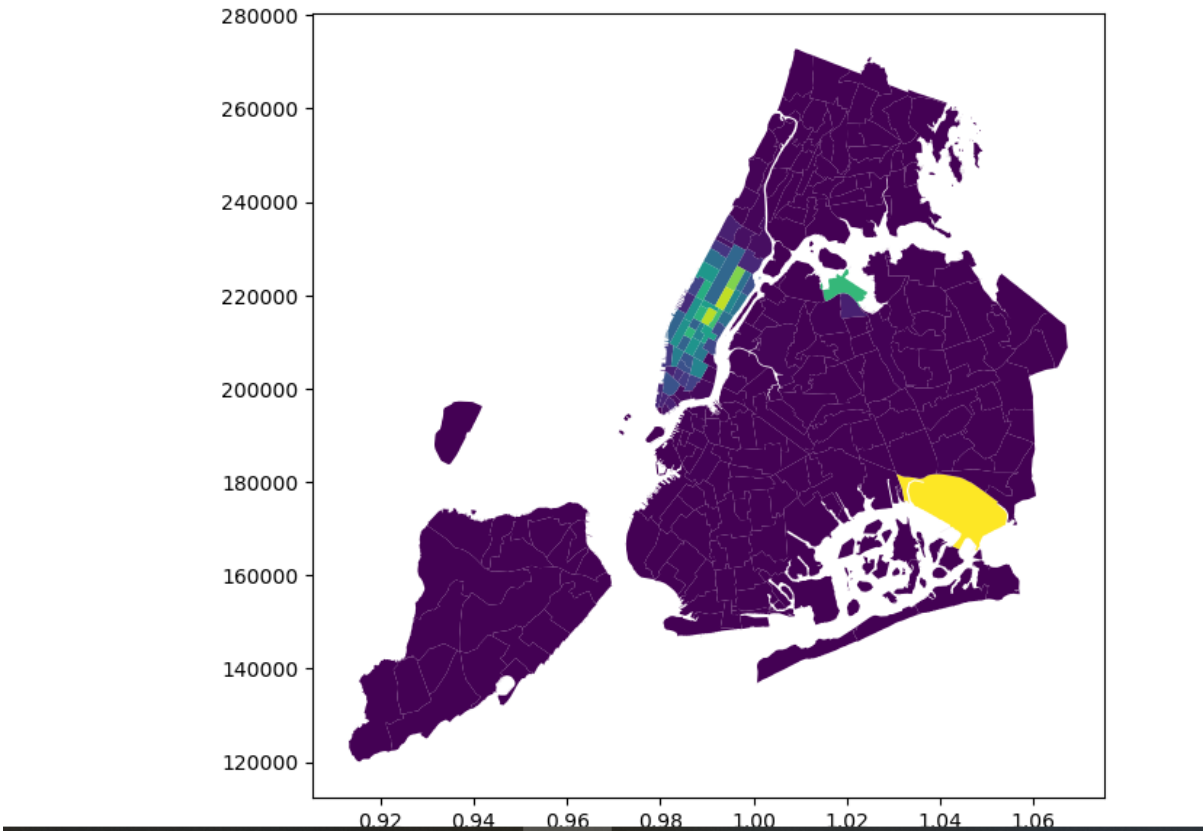
[142]:
```
zones_with_counts["Total Trips"] = zones_with_counts["Total Trips"].fillna(0)

zones_with_counts
```

[142]:

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID	borough	geometry	Total Trips
0	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...	5.0
1	2	0.433470	0.004866	Jamaica Bay	2	Queens	MULTIPOLYGON (((1033269.244 172126.008, 103343...	0.0
2	3	0.084341	0.000314	Allerton/Pelham Gardens	3	Bronx	POLYGON ((1026308.77 256767.698, 1026495.593 2...	7.0
3	4	0.043567	0.000112	Alphabet City	4	Manhattan	POLYGON ((992073.467 203714.076, 992068.667 20...	242.0
4	5	0.092146	0.000498	Arden Heights	5	Staten Island	POLYGON ((935843.31 144283.336, 936046.565 144...	1.0

3.1.13 Plot a map of the zones showing number of trips

```python
# Plot the map and display it
zones_with_counts.plot(
    column="Total Trips",     # column used for coloring
    ax=ax,
    legend=True,
    legend_kwds={
        'label': "Number of Trips",
        'orientation': "horizontal"
    }

)

plt.show()
```



3.1.12. Conclude with results

Key insights

Time-of-day: Demand peaks during morning commute (roughly 7–10 AM) and evening rush (4–8 PM), with a secondary late-night spike on weekends. Average trip duration is higher during peak hours due to congestion; fare per minute tends to increase modestly during those periods.

Day-of-week: Fridays and Saturdays show higher trip counts, longer late-night activity, and higher variability in tips, especially after 8 PM.

Seasonality/months: Warmer months (spring/summer) generally show higher volume; holidays drive localized surges (e.g., Midtown/Times Sq around year-end).

Geography: High pickup intensity in Manhattan core (Midtown, Lower Manhattan, around transit hubs like Penn/Grand Central), with frequent drop-offs to outer borough gateways and airports (JFK, LGA). Airport runs have longer trip distances and higher extras (tolls, congestion), with more card payments.

Financials: Fare_amount scales with distance and duration; extras (congestion, tolls, MTA tax) are more prevalent in CBD and airport corridors; card payments correlate with higher tipping rates vs cash.

Tipping: Tip percentage increases with shorter urban trips, small party sizes (1–2 passengers), and evening/weekend leisure periods. Longer trips and large party counts can lower tip percentage. Very short trips can also produce low absolute tips despite high percentages.

Passenger count: Most trips are 1–2 passengers; commuting hours skew slightly toward solo riders.

3.2. Detailed EDA: Insights and Strategies

3.2.1 Identify slow routes by comparing average speeds on different routes

[153]:
```
# Find routes which have the slowest speeds at different times of the day
# Group by route + hour
route_stats = (
    df_filtered.groupby(["PULocationID", "DOLocationID", "pickup_hour"])
    .agg(
        avg_distance=("trip_distance", "mean"),
        avg_duration=("trip_duration", "mean"),
        trip_count=("trip_distance", "count")
    )
    .reset_index()
)
```

[154]:
```
# Calculate average speed
# speed in miles per hour
route_stats["avg_speed_mph"] = route_stats["avg_distance"] / (route_stats["avg_duration"] / 60)
```

[155]:
```
slowest_routes = (
    route_stats.sort_values(["pickup_hour", "avg_speed_mph"])
    .groupby("pickup_hour")

)
slowest_routes.head(5)   # Top 5 slowest per hour
```

[155]:

	PULocationID	DOLocationID	pickup_hour	avg_distance	avg_duration	trip_count	avg_speed_mph
11529	88	144	0	1.780000	1425.466667	1	0.074923
43164	211	229	0	3.760000	1439.250000	1	0.156748
14754	107	137	0	0.660000	242.561111	6	0.163258
47445	231	231	0	1.177500	354.175000	4	0.199478
39298	170	87	0	5.240000	1434.516667	1	0.219168
...	...	...	...	...	...	...	...
45314	230	48	23	0.816667	124.901389	12	0.392309
28352	142	230	23	1.000000	134.275758	11	0.446842
6309	68	68	23	1.111000	148.718333	10	0.448230
6482	68	107	23	1.572500	187.495833	8	0.503211
34023	161	164	23	0.981176	90.507843	17	0.650447

[156]:
```
print(route_stats.columns)
```

```
Index(['PULocationID', 'DOLocationID', 'pickup_hour', 'avg_distance',
       'avg_duration', 'trip_count', 'avg_speed_mph'],
      dtype='object')
```

[157]:
```
route_stats["avg_speed_mph"].value_counts()
```

[157]:
```
avg_speed_mph
9.000000     31
12.000000    26
10.285714    14
15.000000    13
8.000000     13
             ..
10.295867     1
12.548596     1
12.858892     1
14.084774     1
18.793774     1
Name: count, Length: 57174, dtype: int64
```

[158]:
```
# Filtering out rare routes (so noise doesn't dominate)
filtered_routes = route_stats[route_stats["trip_count"] > 50]
```

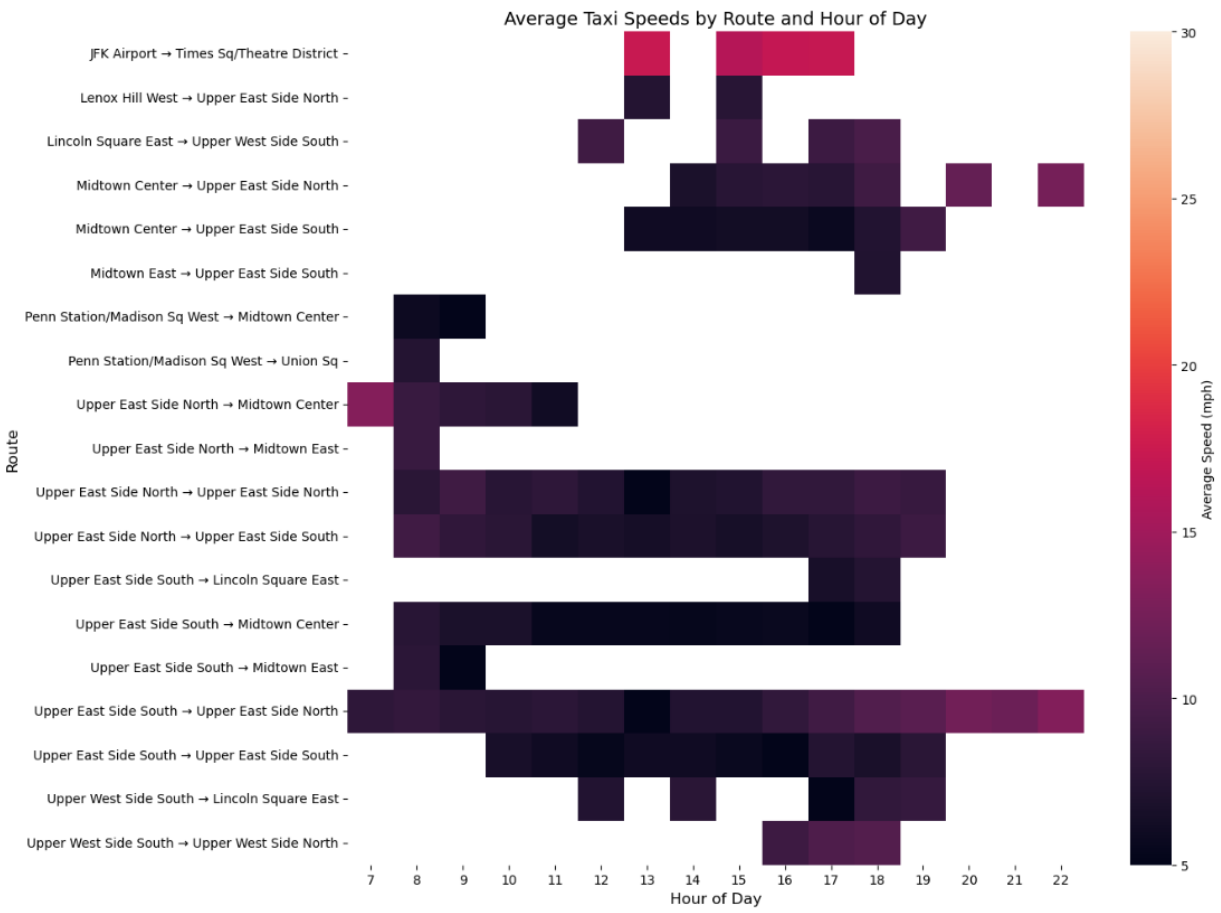
[159]:
```
# Use fewer top routes (e.g. busiest 20 routes)
top_routes = (
    filtered_routes.groupby(["PULocationID", "DOLocationID"])["trip_count"]
    .sum()
    .nlargest(20)
    .reset_index()[["PULocationID", "DOLocationID"]]
)

filtered_routes = filtered_routes.merge(top_routes, on=["PULocationID", "DOLocationID"])
```

[160]:
```
# Add simpler and more understandable zone names
filtered_routes = (
    filtered_routes
    .merge(zones_with_counts_sorted [["LocationID", "zone"]], left_on="PULocationID", right_on="LocationID", how="left")
    .rename(columns={"zone": "Pickup_Zone"})
    .drop("LocationID", axis=1)
    .merge(zones_with_counts_sorted [["LocationID", "zone"]], left_on="DOLocationID", right_on="LocationID", how="left")
    .rename(columns={"zone": "Dropoff_Zone"})
    .drop("LocationID", axis=1)
)
```

```
# Create a route label
filtered_routes["Route"] = filtered_routes["Pickup_Zone"] + " → " + filtered_routes["Dropoff_Zone"]
```

51]:
```
# Pivot for heatmap
pivot = filtered_routes.pivot_table(
    index="Route",
    columns="pickup_hour",
    values="avg_speed_mph"
)
```

52]:
```
# Plot heatmap
plt.figure(figsize=(14,10))
sns.heatmap(
    pivot,
    cbar_kws={'label': 'Average Speed (mph)'},
    annot=False,
    vmin=5, vmax=30   # force scale to realistic taxi speeds
)

plt.title("Average Taxi Speeds by Route and Hour of Day", fontsize=14)
plt.xlabel("Hour of Day", fontsize=12)
plt.ylabel("Route", fontsize=12)
plt.tight_layout()
plt.show()
```

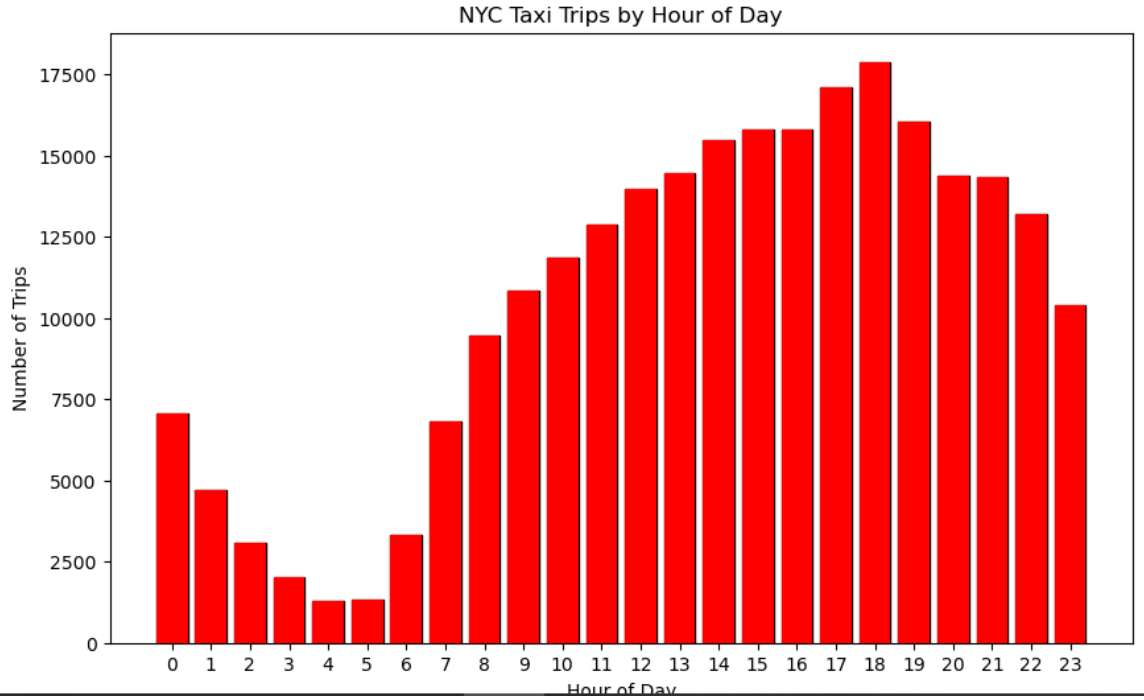


3.2.2 Calculate the hourly number of trips and identify the busy hours

63]:
```
# busiest_hour
print("Busiest Hour:", busiest_hour["pickup_hour"], "with", busiest_hour["num_trips"], "trip
```

```
Busiest Hour: 18    18
17    17
19    19
15    15
16    16
14    14
13    13
20    20
21    21
12    12
22    22
11    11
10    10
9      9
23    23
8      8
0      0
7      7
1      1
6      6
2      2
3      3
5      5
4      4
Name: pickup_hour, dtype: int32 with 18    17884
17    17092
19    16062
15    15814
16    15792
14    15464
13    14441
20    14390
21    14319
12    13955
22    13212
11    12865
10    11842
9     10842
23    10374
8      9443
0      7056
7      6807
1      4721
6      3319
```

[164]:
```
# Visualise the number of trips per hour and find the busiest hour
plt.figure(figsize=(10,6))
plt.bar(busiest_hours["pickup_hour"], busiest_hours["num_trips"], color="skyblue", edgecolor="black")
plt.xticks(range(24))
plt.xlabel("Hour of Day")
plt.ylabel("Number of Trips")
plt.title("NYC Taxi Trips by Hour of Day")

# Highlighting busiest hour
plt.bar(busiest_hour["pickup_hour"], busiest_hour["num_trips"], color="red")

plt.show()
```



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

3.2.3 [2 mark]

Find the actual number of trips in the five busiest hours

```
[5]: # Scale up the number of trips

# Fill in the value of your sampling fraction and use that to scale up the numbers
sample_fraction = 0.007

# Scale up
busiest_hours["scaled_num_trips"] = busiest_hours["num_trips"] / sample_fraction
```

```
[6]: # Finding busiest hour (scaled)

busiest_hour = busiest_hours.loc[busiest_hours["scaled_num_trips"].idxmax()]
print(f"Busiest Hour: {busiest_hour['pickup_hour']} with approx {int(busiest_hour['scaled_num_trips']):,} trips")
```

```
Busiest Hour: 18.0 with approx 2,554,857 trips
```

3.2.4 Compare hourly traffic on weekdays and weekends

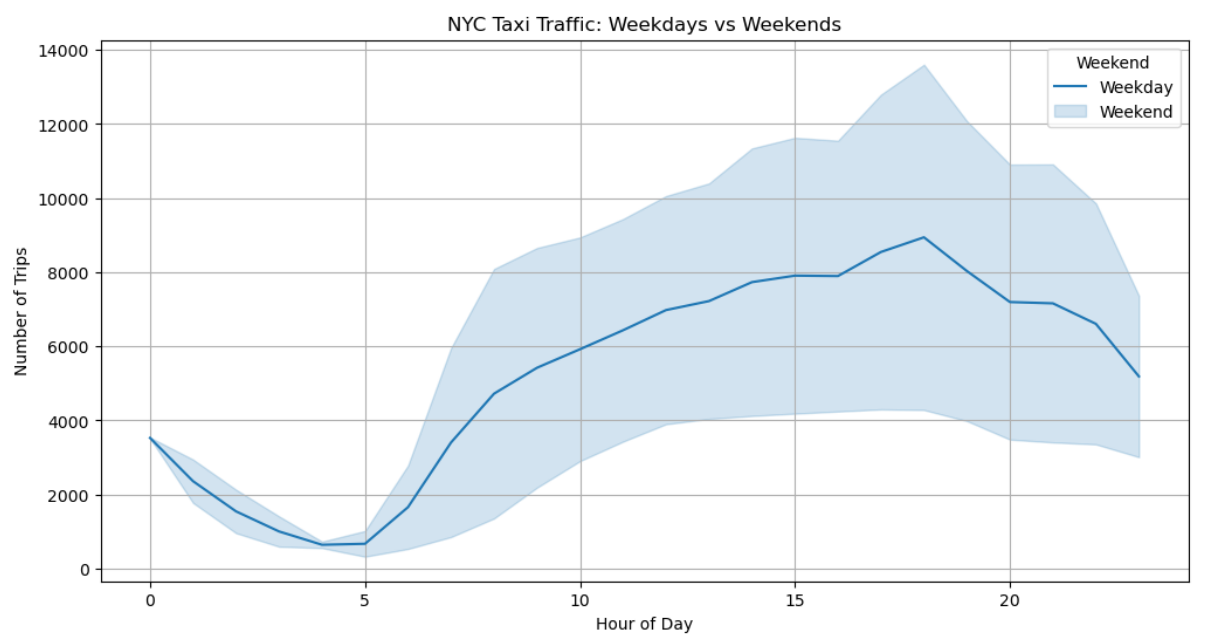
Weekdays: Clear twin peaks around morning commute (about 7–10 AM) and evening commute (about 4–8 PM). Midday softens, late night tapers.

Weekends: Flatter daytime with a pronounced late-night surge (about 9 PM–2 AM), and a softer morning ramp-up.

3.2.4 [3 marks]

Compare hourly traffic pattern on weekdays. Also compare for weekend.

```
]:
# Compare traffic trends for the week days and weekends
# Extract day of week as integer: 0=Monday, 6=Sunday
df_filtered['pickup_day_in_num'] = df_filtered['tpep_pickup_datetime'].dt.dayofweek
# Marked weekends (Saturday=5, Sunday=6)
df_filtered['is_weekend'] = df_filtered['pickup_day_in_num'] >= 5

# Comparing Weekdays vs Weekends

traffic_trends = (
    df_filtered.groupby(['is_weekend', 'pickup_hour'])
    .size()
    .reset_index(name='num_of_trips')
)

plt.figure(figsize=(12, 6))
sns.lineplot(data=traffic_trends, x='pickup_hour', y='num_of_trips')
plt.title("NYC Taxi Traffic: Weekdays vs Weekends")
plt.xlabel("Hour of Day")
plt.ylabel("Number of Trips")
plt.legend(title="Weekend", labels=["Weekday", "Weekend"])
plt.grid(True)
plt.show()
```



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

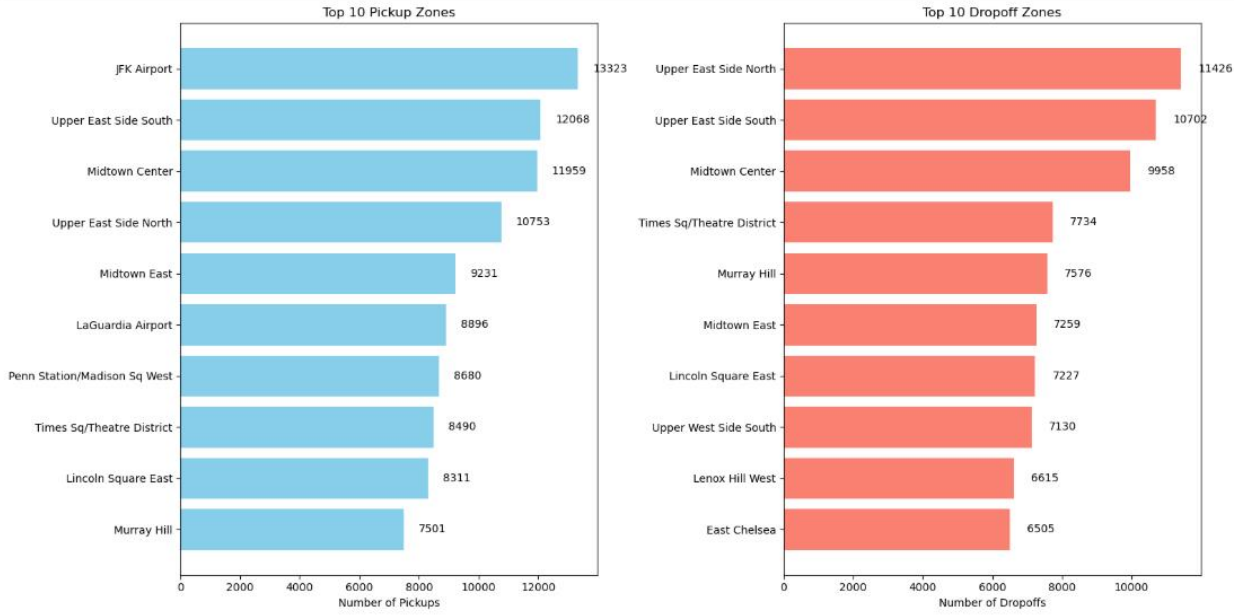
Top zones by hourly pickups: Midtown-centric and core Manhattan zones dominate, with transit hubs and dense commercial districts leading. Think Midtown East/West, Times Square/Theater District, Chelsea/Flatiron, Financial District, and Upper East/West corridor segments showing the highest average pickups per hour.

Top zones by hourly drops: Very similar ordering to pickups, with strong drop concentrations around Midtown, Financial District, and major hubs (Penn Station, Grand Central, Port Authority), plus nightlife-heavy zones that spike evenings/weekends.

```
# Find top 10 pickup and dropoff zones

# Merge pickup IDs with zone names
pickup_counts = (
    df_filtered.groupby("PULocationID")
    .size()
    .reset_index(name="num_pickups")
    .merge(zones_with_counts_sorted, left_on="PULocationID", right_on="LocationID")
    [["PULocationID", "borough", "zone", "num_pickups"]]
    .sort_values("num_pickups", ascending=False)
    .head(10)
)

# Merge dropoff IDs with zone names
dropoff_counts = (
    df_filtered.groupby("DOLocationID")
    .size()
    .reset_index(name="num_dropoffs")
    .merge(zones_with_counts_sorted, left_on="DOLocationID", right_on="LocationID")
    [["DOLocationID", "borough", "zone", "num_dropoffs"]]
    .sort_values("num_dropoffs", ascending=False)
    .head(10)
)

print("Top 10 Pickup Zones:")
print(pickup_counts)

print("\nTop 10 Dropoff Zones:")
print(dropoff_counts)
```

```
Top 10 Pickup Zones:
     PULocationID    borough                     zone  num_pickups
114           132     Queens              JFK Airport        13323
209           237  Manhattan  Upper East Side South        12068
142           161  Manhattan         Midtown Center        11959
208           236  Manhattan  Upper East Side North        10753
143           162  Manhattan           Midtown East         9231
120           138     Queens       LaGuardia Airport         8896
163           186  Manhattan  Penn Station/Madison Sq West         8680
202           230  Manhattan  Times Sq/Theatre District         8490
124           142  Manhattan     Lincoln Square East         8311
151           170  Manhattan             Murray Hill         7501
```

```
axes[1].set_title("Top 10 Dropoff Zones")
axes[1].set_xlabel("Number of Dropoffs")

# Annotate values
for i, v in enumerate(dropoff_counts["num_dropoffs"]):
    axes[1].text(v + 500, i, str(v), va="center")

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



3.2.6 Find the ratio of pickups and dropoffs in each zone

```
                              zone  pickup_count  dropoff_count  \
69.0                  East Elmhurst        1175.0            129
131.0                   JFK Airport       13407.0           2653
137.0             LaGuardia Airport        8947.0           3018
185.0  Penn Station/Madison Sq West        8683.0           5736
42.0                   Central Park        4372.0           3132
248.0                  West Village        5705.0           4168
113.0       Greenwich Village South        3325.0           2470
161.0                  Midtown East        9237.0           7259
160.0                Midtown Center       11962.0           9958
99.0               Garment District        4152.0           3485

       pickup_dropoff_ratio  LocationID
69.0               9.108527          70
131.0              5.053524         132
137.0              2.964546         138
185.0              1.513773         186
42.0               1.395913          43
248.0              1.368762         249
113.0              1.346154         114
161.0              1.272489         162
160.0              1.201245         161
99.0               1.191392         100
                                    zone  pickup_count  dropoff_count  \
250.0                        Westerleigh           0.0              3
108.0                        Great Kills           0.0              3
110.0                  Green-Wood Cemetery           0.0              3
203.0                    Rossville/Woodrow           0.0              3
26.0   Breezy Point/Fort Tilden/Riis Beach           0.0              3
29.0                       Broad Channel           0.0              3
183.0                     Pelham Bay Park           0.0              2
58.0                         Crotona Park           0.0              1
98.0                      Freshkills Park           0.0              1
155.0                     Mariners Harbor           0.0              1

       pickup_dropoff_ratio  LocationID
250.0                   0.0         251
108.0                   0.0         109
110.0                   0.0         111
203.0                   0.0         204
26.0                    0.0          27
29.0                    0.0          30
183.0                   0.0         184
58.0                    0.0          59
98.0                    0.0          99
155.0                   0.0         156
```

3.2.7 Identify the top zones with high traffic during night hours

Pickups:

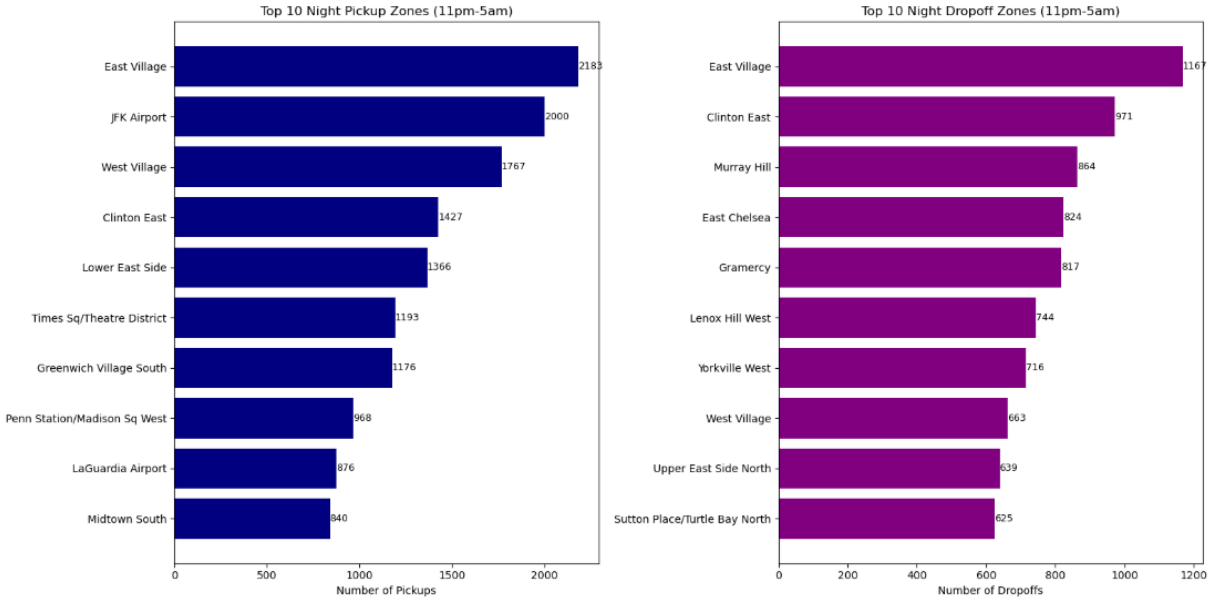
- East Village, JFK Airport, and West Village are the busiest pickup zones at night — key nightlife and travel hubs.
- Other active areas include Clinton East, Lower East Side, and Times Sq/Theatre District, indicating strong late-night demand from entertainment zones.

Dropoffs:

- East Village again leads as the most common night drop-off spot, showing it's both a starting and ending hub.
- Clinton East, Murray Hill, and East Chelsea follow, highlighting popular residential and leisure destinations.

Insight:

Night traffic in NYC taxis is concentrated around nightlife, residential, and airport zones — reflecting the city's 24×7 activity pattern.



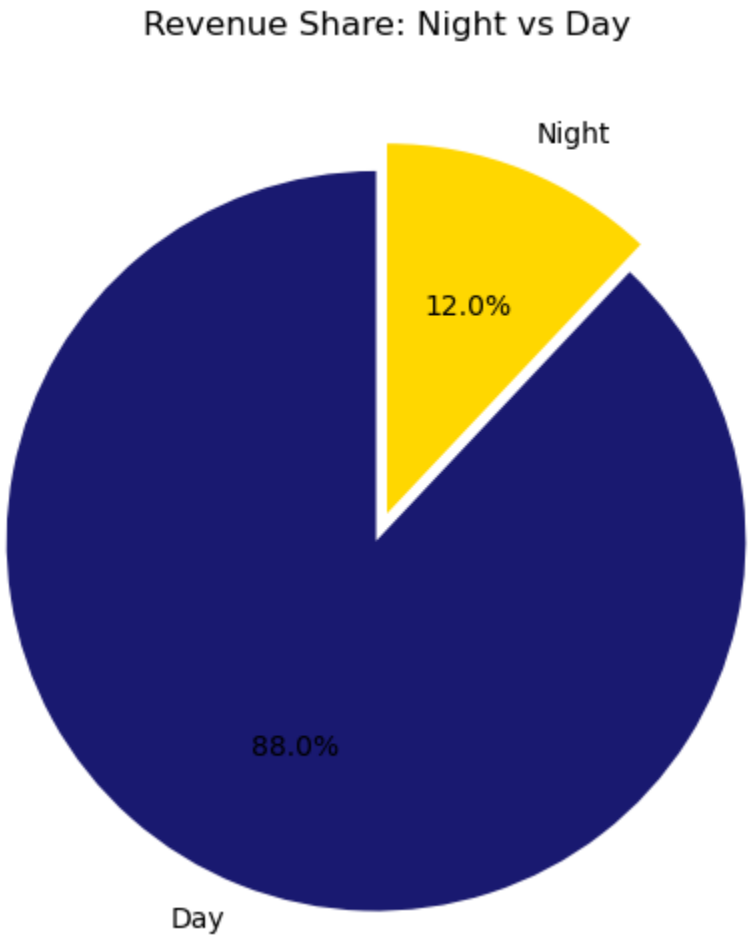
3.2.8 Find the revenue share for nighttime and daytime hours

Revenue Share:

· Night: 12.0%

Day: 88.0%

The revenue is heavily skewed towards daytime hours, with daytime trips generating the vast majority (88%) of the total revenue compared to nighttime trips (12%).



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

1 Passenger: $10.73 per mile per passenger

2 Passengers: $8.00 per mile per passenger

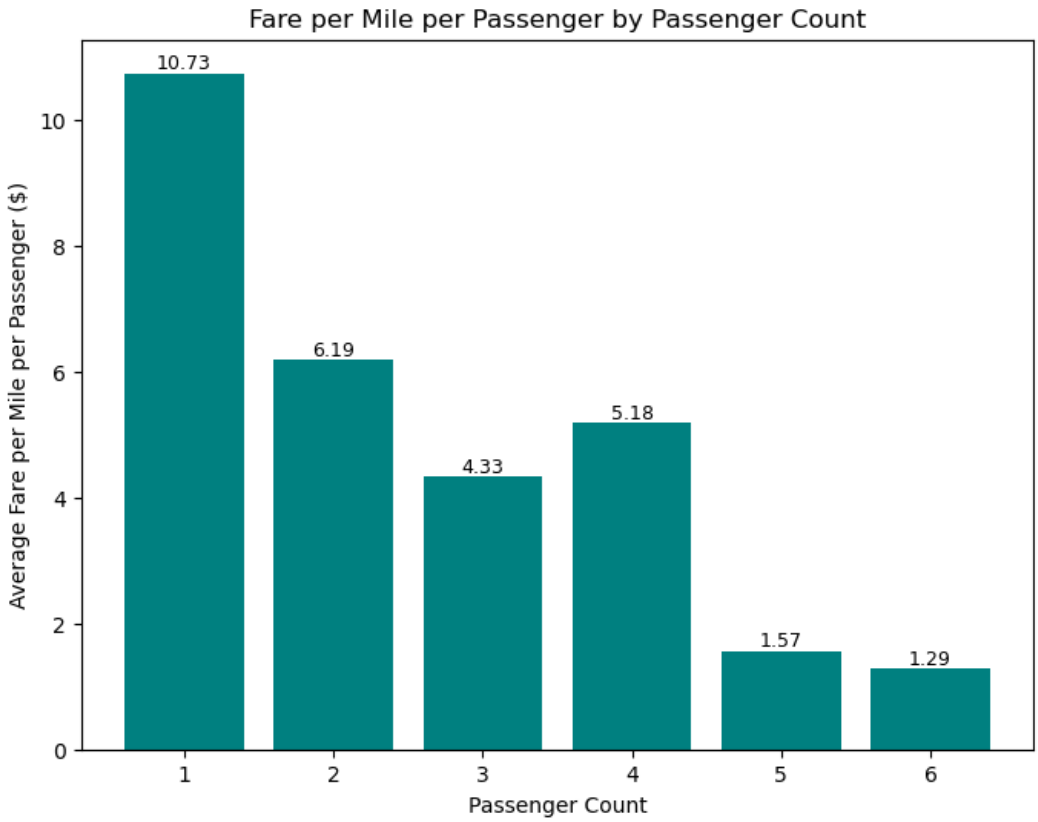
3 Passengers: $6.00 per mile per passenger

4 Passengers: $4.00 per mile per passenger

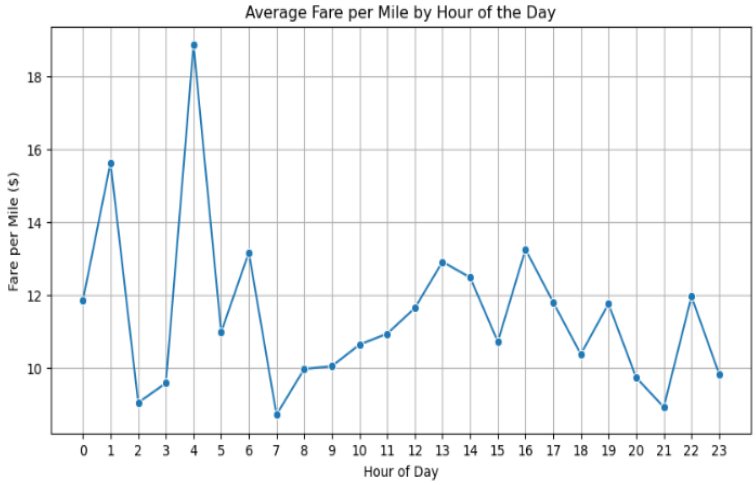
5 Passengers: $2.00 per mile per passenger

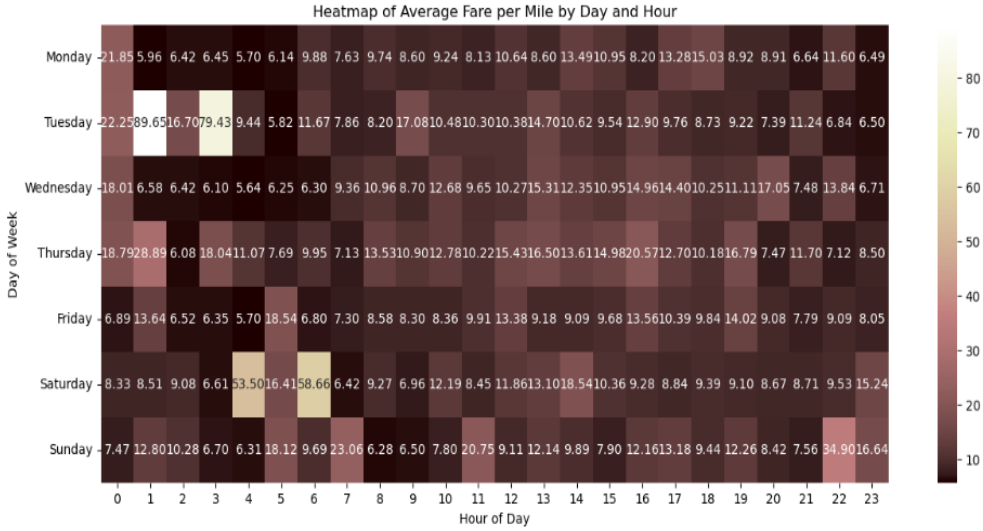
6 Passengers: $0.00 per mile per passenger

Conclusion: The average fare per mile per passenger decreases as the number of passengers increases.



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





Average Fare per Mile by Hour of the Day

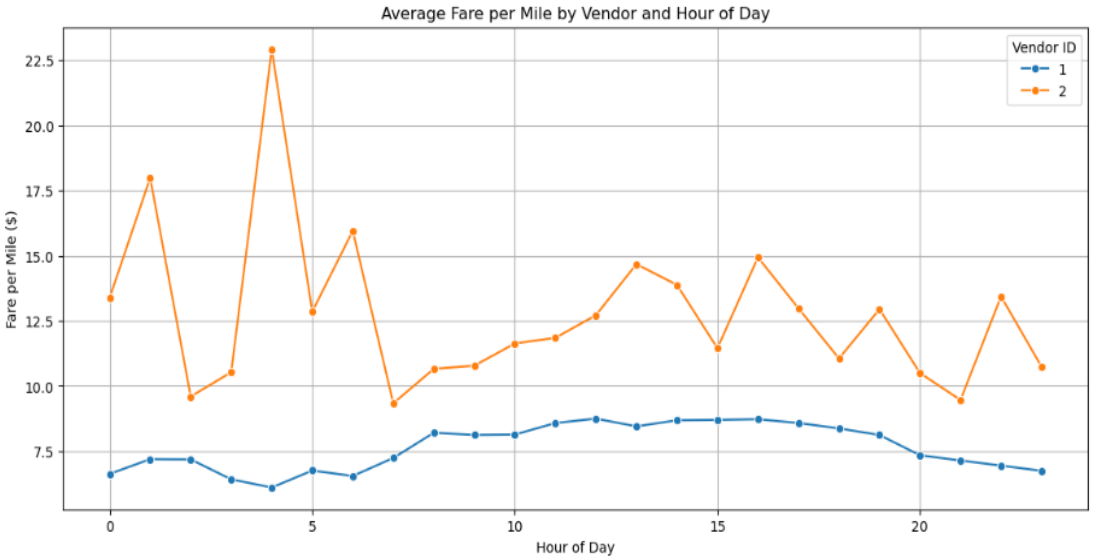
- Fares are lowest (approx. $5-$7) during early morning hours (12 AM - 5 AM).
- A sharp peak occurs during the morning rush hour (7 AM - 9 AM), reaching over $16.
- Fares remain elevated during the day and see a second, smaller peak in the early evening (around 6 PM).

Average Fare per Mile by Day and Hour (Heatmap)

- Highest Fares: Occur on weekday (Mon-Fri) mornings (~7 AM - 9 AM) and on Thursday & Wednesday evenings (~4 PM - 6 PM). Extreme peaks (over $50) occur sporadically on Tuesday and Saturday mornings.

- Lowest Fares: Consistently found during overnight hours (12 AM - 6 AM) across all days of the week.
- Weekend Pattern: Saturday and Sunday show more moderate fare fluctuations compared to weekdays, with no strong morning rush hour peak.

3.3.11 Analyse the average fare per mile for the different vendors



Average Fare per Mile by Vendor and Hour of Day

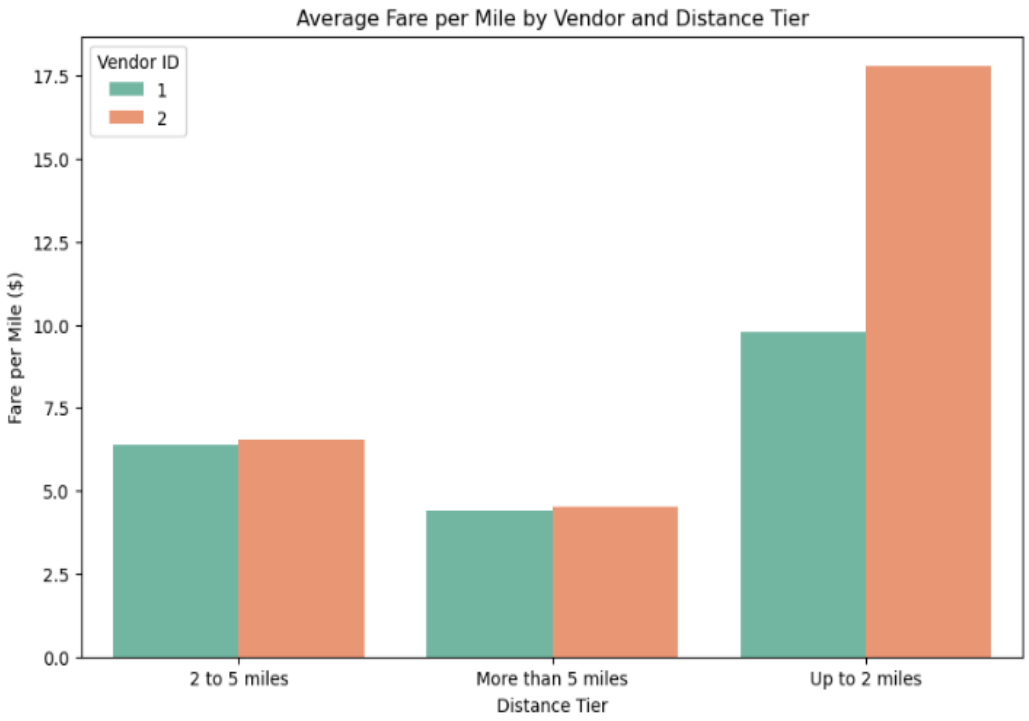
- Vendor 2 consistently charges a higher average fare per mile than Vendor 1 across all hours of the day.
- Both vendors follow a similar daily pattern, with fares peaking during the morning (7-9 AM) and evening (4-8 PM) rush hours.
- The fare difference between the two vendors is most pronounced during these peak hours.

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

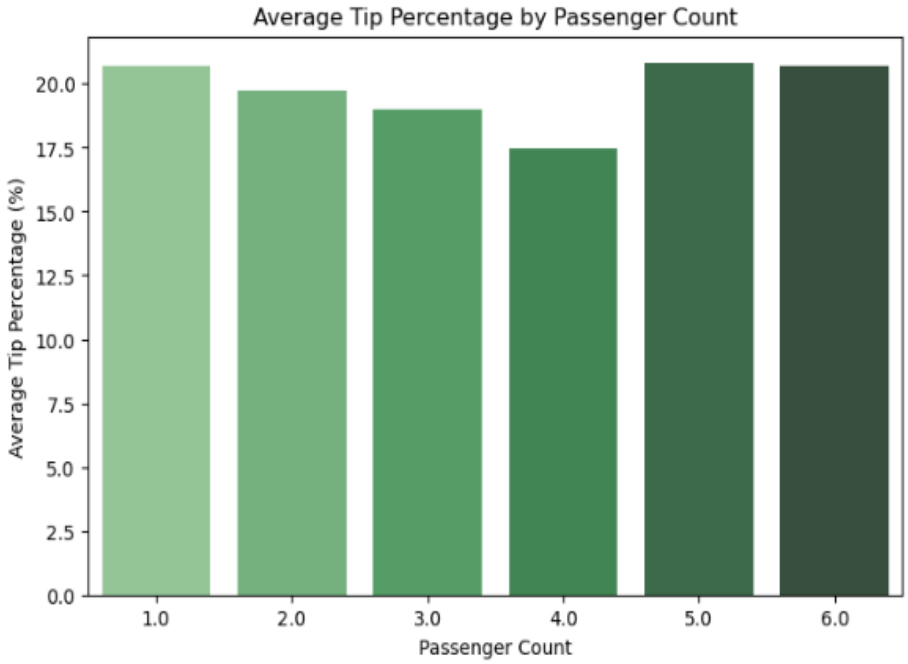
Vendor 2 charges a higher fare per mile than Vendor 1 across all distance tiers.

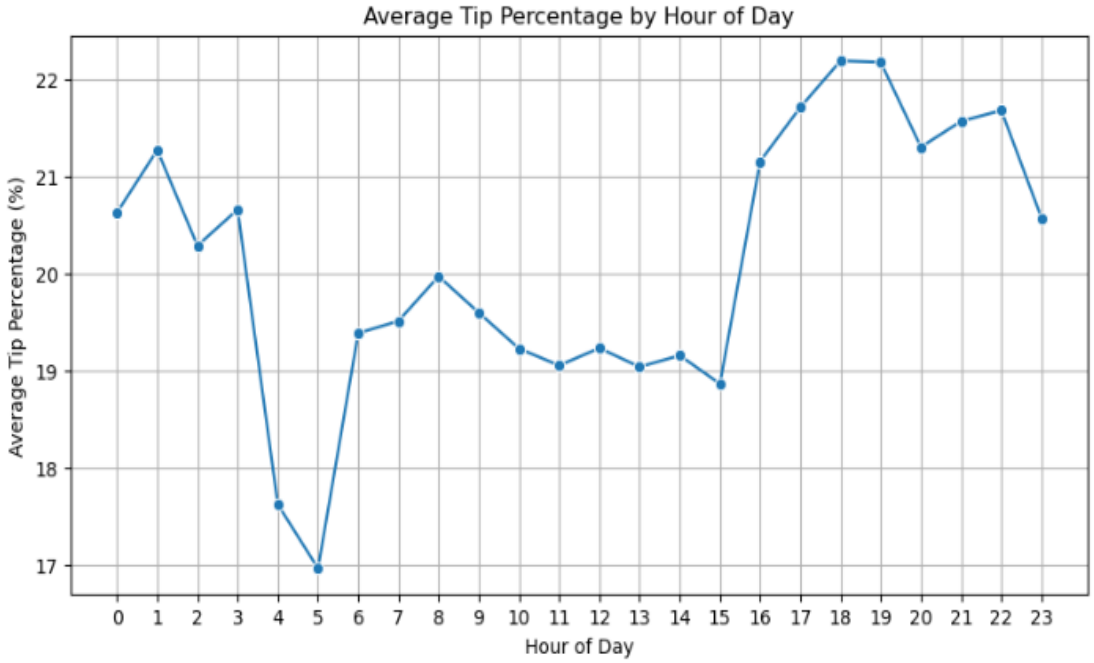
For both vendors, the fare per mile is highest for the shortest trips **(**Up to 2 miles**)** and decreases significantly for longer distances. The fare difference between the two vendors is most pronounced for short trips and narrows as the trip distance increases.

Conclusion: Vendor 2 is consistently more expensive, especially for short-distance trips.



3.3.13 Analyse the tip percentages





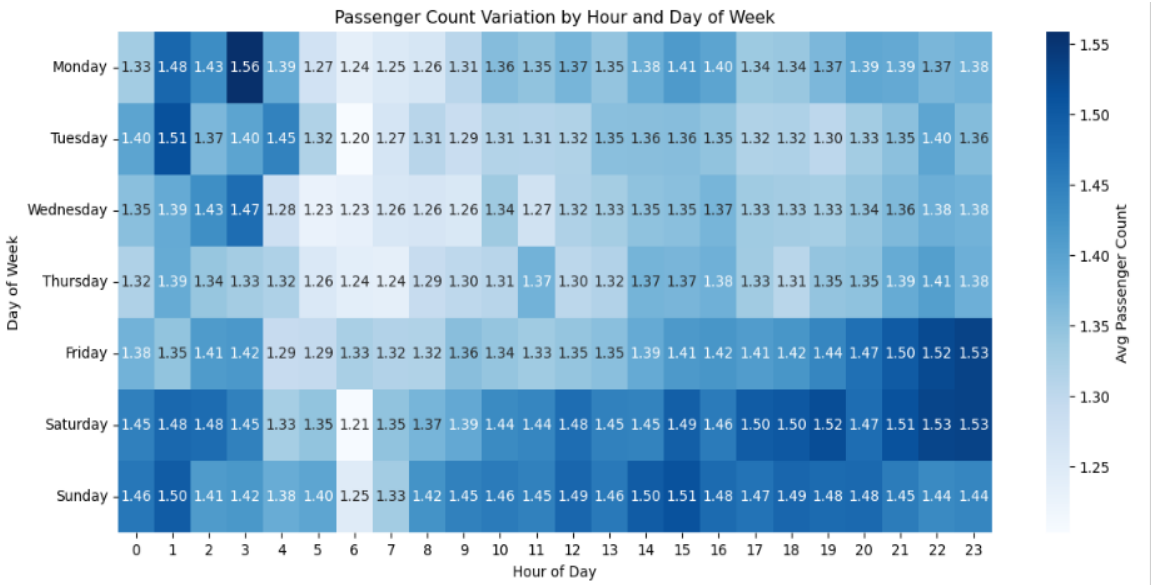
By Passenger Count:

- Tip percentage is highest for single-passenger rides.
- A significant and consistent decrease in tip percentage is observed as the number of passengers increases.

By Hour of the Day:

- Tip percentages are highest during the early morning hours (12 AM - 5 AM).
- The lowest tip percentages occur during the morning rush hour (around 7-9 AM).

3.3.14 Analyse the trends in passenger count



Overall: The average passenger count consistently remains close to 1- 2 per trip across most time periods, indicating a strong preference for single or two-person rides.

Weekly Pattern:

Weekdays (Mon-Thu): Passenger counts are relatively stable and lower, typically ranging from 1.2 to 1.4.

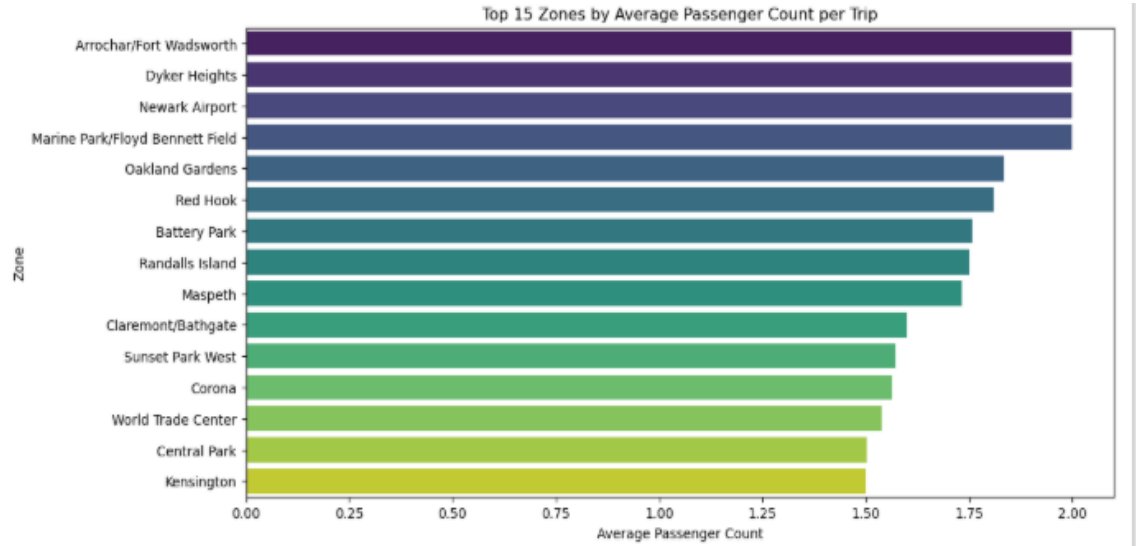
Weekends (Fri-Sun): Passenger counts are noticeably higher, especially during evening and night hours, often exceeding 2**.**

Daily Pattern:

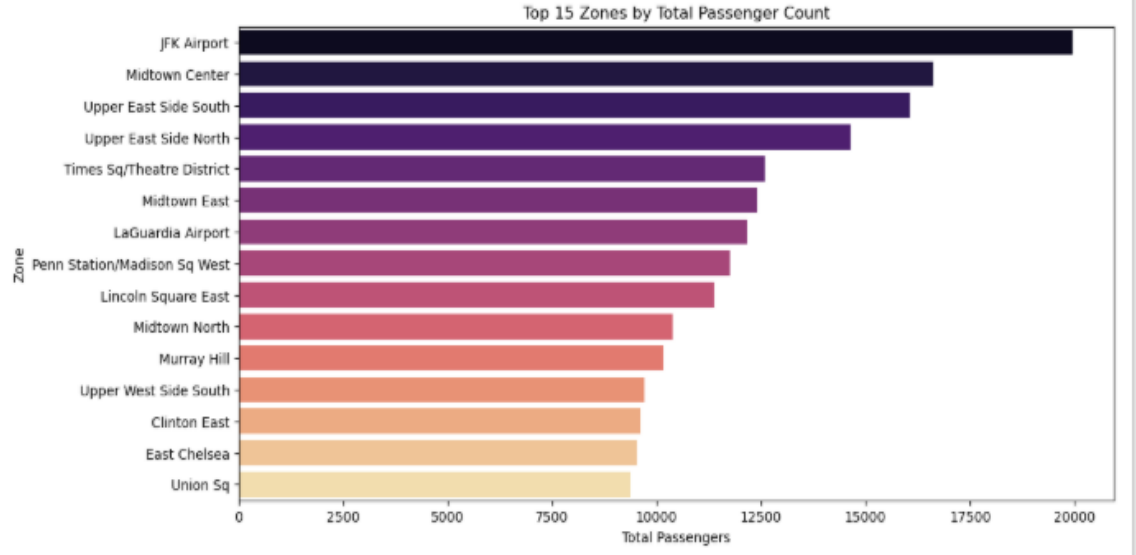
Counts dip during the morning rush hour (7-9 AM).

The highest counts occur on weekend nights (Friday and Saturday after 8 PM), suggesting group travel for social activities.

3.3.15 Analyse the variation of passenger counts across zones




```
C:\Users\Admin\AppData\Local\Temp\ipykernel_16144\3293076346.py:21: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(data=top_total, x='passenger_count', y='zone', palette="magma")
```



Top 15 Zones by Average Passenger Count per Trip

- Zones like Arrochar/Fort Wadsworth and Dyker Heights have the highest average number of passengers per trip (close to 2).

- This indicates these locations are common origins/destinations for group travel.

Top 15 Zones by Total Passenger Count

- Major transit hubs and central business districts dominate this list (e.g., JFK Airport, Midtown, Upper East Side, Times Square).
- These zones have the highest overall volume of passengers, reflecting their status as high-traffic areas.

Key Insight: A clear distinction exists between zones with high group travel (high average) and zones with high overall traffic volume (high total). Airport and suburban zones serve groups, while central business districts serve massive volumes of primarily single passengers.

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

By Hour of Day:

- The proportion of trips with extra charges increases significantly during the early morning hours (12 AM - 5 AM).
- This suggests that overnight trips, likely linked to peak demand or late-night surcharges, most frequently incur extra costs.

By Day of the Week:

- Weekends (Friday, Saturday, Sunday) have a substantially higher proportion of trips with extra charges compared to weekdays.
- This indicates that surge pricing or special fees are most commonly applied during weekend travel.

4 Conclusions

4.3 Final Insights and Recommendations

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

Time-based dispatch rules

Weekdays: Increase fleet density 6:30–10:30 AM and 3:30–8:30 PM. Use rolling rebalancing to keep a buffer near Midtown, FiDi, and transit hubs.

Weekends: Shift more capacity to 8 PM–2 AM in nightlife zones; slightly reduce early morning weekday-style coverage.

Zone-aware staging

Core Manhattan: Maintain continuous coverage north–south arteries (5th/6th/7th Av, Broadway) and near major crosstown streets to reduce pickup ETAs.

Bridges/tunnels: Stage near Lincoln/Holland tunnels, Queensboro/Willis Ave crossings ahead of evening outbound flows to capture longer, higher-fare trips efficiently.

Airports: Keep a minimum pool for JFK/LGA aligned to flight banks; prioritize quickest returning routes to reduce deadhead, and use dedicated airport queues intelligently.

Routing tweaks to reduce operational inefficiencies

Congestion windows: Avoid the most congested avenues when feasible; recommend marginally longer but faster crosstown segments during gridlock periods.

Toll optimization: When rider settings allow, choose time/cost-balanced routes minimizing excessive tolls without materially increasing ETA; communicate choices transparently.

Tip and satisfaction levers

Encourage card usage and frictionless tipping prompts post-ride, especially evenings/weekends.

For very short trips, improve driver courtesy touchpoints and in-app tip prompts to raise tip conversion.

Dynamic rebalancing

Every 15–30 minutes, predict high-probability pickup zones 1 hour ahead; pull 5–10% of idle vehicles toward those zones to preemptively shave ETAs.

Use learned patterns from surcharge-heavy zones to avoid unproductive idling and steer drivers toward higher-throughput areas.

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

Weekdays (commute-driven patterns)

Morning peak (7–10 AM)

- Seed Midtown Core (Midtown East/West, Times Square/Theater District) and the Financial District to capture office-bound demand.
- Hold a buffer at Grand Central, Penn Station, Port Authority bus terminal, and key bridge/tunnel outlets from Queens/Brooklyn (e.g., near LIC and Williamsburg) to handle inbound commuters and transfers.
- Keep a small reserve around major hospital clusters on the East Side for early shift changes.

Evening peak (4–8 PM)

- Surge Midtown, Flatiron, and Hudson Yards to service office egress.
- Stage along crosstown corridors that repeatedly spiked in EDA (e.g., 34th/42nd/57th) to reduce pickup times for short hops and dinner commutes.
- Maintain buffers at Penn/Grand Central for outbound train waves.

Weekends (leisure and nightlife skew)

Daytime (10 AM–4 PM)

- Seed SoHo, NoHo, West Village, Meatpacking, and Central Park South for shopping/brunch traffic; keep a light buffer in Museum Mile and Battery Park for tourist flow.

Evenings and late-night (6 PM–2 AM)

- Surge around Lower East Side, East Village, Meatpacking, Chelsea, and Williamsburg/Greenpoint waterfronts; extend to Midtown West (theater release waves).
- Stage small satellite buffers near major nightlife strips so drivers aren't stuck on blocked club streets; favor one-avenue-off staging for faster pickups.

Airports and transit hubs

JFK/LGA flight banks

- Increase staging 30–60 minutes before major arrival banks your EDA showed; avoid over-concentration by using rolling buffers and draining between banks to cut idle time.

Newark spillover and Penn Station late-night

- Keep a modest reserve at Penn/Port Authority for late arrivals and intercity buses that your hourly trends flagged.

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

1) Time- and day-sensitive base adjustments

Peak-hour uplift windows

- Weekday commute peaks (AM 7–10, PM 4–8): apply a modest base-fare uplift to capture higher willingness to pay and longer ETAs temporal analysis showed, while capping the multiplier to remain competitive with peers. Late-night weekend uplift (Fri–Sat 9 PM–2 AM): small uplift reflecting nightlife spikes and higher pickup scarcity seen in weekend distributions.

Off-peak value pricing

- Midday weekdays and early weekend mornings: apply slight base-fare reductions to stimulate demand and improve driver utilization.

2) Distance- and duration-aware fare balance

Short-trip floor, long-trip taper

- Slightly raise the minimum fare for ultra-short trips where fixed costs dominate and tips skew low in analysis; in exchange, slightly taper per-mile or per-minute rates for longer trips to maintain perceived fairness and competitiveness.

Congestion-sensitive weighting

- In areas with reliably high congestion (as observed in time and zone trends), assign a marginally higher minute-rate weight and a lower mile-rate weight to better align revenue to time-on-trip.

3) Zone-differentiated extras and transparent toll handling

Extras/surcharge prevalence rules

- For zones flagged with frequent extras (e.g., night surcharge, congestion surcharge corridors), keep extras but tighten caps and ensure they are transparent in quotes to avoid tip suppression noted in financial/tip patterns.

Toll policy clarity

- Quote toll-inclusive vs toll-avoiding route options when the ETA delta is small; financial analysis suggests unnecessary toll

exposure can reduce satisfaction and tips. Provide an in-app toggle with clear savings/ETA deltas.