Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science Engineering and Information Systems (SCORE)

Winter Semester 2024-25

| | |
|---|---|
| **Course Code:** | SWE – 2009 |
| **Course Name:** | Data Mining Techniques |
| **Faculty:** | Dr. DURAI RAJ VINCEMT P. M. |

# Digital Assignment

# Performing EDA on New York City's Taxi Fare Dataset

| | |
|---|---|
| **Student Name** | P. Mokshagna |
| **RegNo.** | 22MIS0377 |
| **Slot** | E2 + TE2 |

**Link to the Repository:** https://github.com/PAGADALA-MOKSHAGNA/EDA_Taxi_Trip_Data

# 1. Introduction

New York City's taxi system is **one of the largest and most complex transportation networks in the world.** With millions of trips recorded each year, analysing taxi fare data provides valuable insights into urban mobility, fare structures, and passenger demand patterns. **Exploratory Data Analysis (EDA) is a crucial step in understanding the dataset, identifying trends, and detecting anomalies** that could impact fare predictions, policymaking, or ride-hailing business strategies.

# 2. Statistical Insights

NYC's yellow taxis complete approximately **200,000–300,000 trips per day**.
The average trip fare is around **$12–$15**, with a median trip distance of **1–3 miles**.
Taxi fares are regulated and structured, consisting of **base fare, per-mile charges, surcharges, and tolls**.
The introduction of ride-hailing services like Uber and Lyft has led to **shifts in fare distribution**, making EDA essential for understanding new trends.

# 3. Description of the Dataset

Dataset is ideal for transport data analysis, predictive modeling, and fare optimization. Data scientists can use it to analyze traffic patterns, predict trip times, study passenger behavior, and evaluate taxi service efficiency. It's a rich source for exploring New York City's transportation dynamics and urban mobility trends.

- **VendorID**: A unique identifier for the taxi vendor or service provider.
- **tpep_pickup_datetime**: The date and time when the passenger was picked up.
- **tpep_dropoff_datetime**: The date and time when the passenger was dropped off.
- **passenger_count**: The number of passengers in the taxi.
- **trip_distance**: The total distance of the trip in miles or kilometers.
- **RatecodeID**: The rate code assigned to the trip, representing fare types.
- **store_and_fwd_flag**: Indicates whether the trip data was stored locally and then forwarded later (Y/N).
- **PULocationID**: The unique identifier for the pickup location (zone or area).
- **DOLocationID**: The unique identifier for the drop-off location (zone or area).
- **payment_type**: The method of payment used by the passenger (e.g., cash, card).
- **fare_amount**: The base fare for the trip.
- **extra**: Additional charges applied during the trip (e.g., night surcharge).
- **mta_tax**: The tax imposed by the Metropolitan Transportation Authority.
- **tip_amount**: The tip given to the driver, if applicable.
- **tolls_amount**: The total amount of tolls charged during the trip.
- **improvement_surcharge**: A surcharge imposed for the improvement of services.
- **total_amount**: The total fare amount, including all charges and surcharges.
- **congestion_surcharge**: An additional charge for trips taken during high traffic congestion times.

# New York City's Taxi Fare Dataset.

Performing Exploratory Data Analysis on NYC Taxi Fare Dataset.
https://www.kaggle.com/datasets/diishasiing/revenue-for-cab-drivers

## 1. Importing Necessary Libraries.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

## 2. Importing the Dataset.

In [2]:
```
data = pd.read_csv("TaxiFareDataset.csv")
data.head()
```

Out[2]:

| | VendorID | tpep_pickup_datetime | tpep_dropoff_datetime | passenger_count | trip_distar |
|---|---|---|---|---|---|
| **0** | 2.0 | 2020-01-17 18:18:36 | 2020-01-17 18:46:24 | 1.0 | 3 |
| **1** | 2.0 | 2020-01-25 10:49:58 | 2020-01-25 11:07:35 | 1.0 | 3 |
| **2** | 2.0 | 2020-01-15 07:30:08 | 2020-01-15 07:40:01 | 1.0 | 1 |
| **3** | 2.0 | 2020-01-09 06:29:09 | 2020-01-09 06:35:44 | 1.0 | 0 |
| **4** | 2.0 | 2020-01-26 12:24:04 | 2020-01-26 12:29:15 | 2.0 | 0 |

In [3]:
```
data.tail()
```

Out[3]:

| | VendorID | tpep_pickup_datetime | tpep_dropoff_datetime | passenger_count | trip_ |
|---|---|---|---|---|---|
| **799995** | 2.0 | 2020-01-21 20:44:43 | 2020-01-21 21:00:49 | 1.0 | |
| **799996** | 2.0 | 2020-01-26 00:10:36 | 2020-01-26 00:12:55 | 1.0 | |
| **799997** | 2.0 | 2020-01-05 11:30:02 | 2020-01-05 11:36:31 | 2.0 | |
| **799998** | 1.0 | 2020-01-31 11:26:00 | 2020-01-31 11:35:51 | 1.0 | |
| **799999** | 1.0 | 2020-01-15 12:45:27 | 2020-01-15 12:51:31 | 1.0 | |

Knowing the Dimensionality and Attribute Types.

In [4]:
```
print(f'The Dimensionality of Dataset: {data.shape}')
print('\033[33mThe Attributes of Dataset and types\033[0m')
print(data.dtypes)
```

```
The Dimensionality of Dataset: (800000, 18)
The Attributes of Dataset and types
VendorID                 float64
tpep_pickup_datetime      object
tpep_dropoff_datetime     object
passenger_count          float64
trip_distance            float64
RatecodeID               float64
store_and_fwd_flag        object
PULocationID               int64
DOLocationID               int64
payment_type             float64
fare_amount              float64
extra                    float64
mta_tax                  float64
tip_amount               float64
tolls_amount             float64
improvement_surcharge    float64
total_amount             float64
congestion_surcharge     float64
dtype: object
```

Characteristics of Numerical Dataset.

In [5]:
```
data.describe()
```

Out[5]:

| | VendorID | passenger_count | trip_distance | RatecodeID | PULocationID | D |
|---|---|---|---|---|---|---|
| **count** | 791815.000000 | 791815.000000 | 800000.000000 | 791815.000000 | 800000.000000 | 8 |
| **mean** | 1.668655 | 1.515306 | 2.898068 | 1.059749 | 164.649531 | |
| **std** | 0.470697 | 1.151033 | 3.823280 | 0.820551 | 65.609595 | |
| **min** | 1.000000 | 0.000000 | -29.100000 | 1.000000 | 1.000000 | |
| **25%** | 1.000000 | 1.000000 | 0.970000 | 1.000000 | 132.000000 | |
| **50%** | 2.000000 | 1.000000 | 1.600000 | 1.000000 | 162.000000 | |
| **75%** | 2.000000 | 2.000000 | 2.940000 | 1.000000 | 234.000000 | |
| **max** | 2.000000 | 9.000000 | 241.640000 | 99.000000 | 265.000000 | |

In [6]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800000 entries, 0 to 799999
Data columns (total 18 columns):
 #   Column                 Non-Null Count   Dtype  
---  ------                 --------------   -----  
 0   VendorID               791815 non-null  float64
 1   tpep_pickup_datetime   800000 non-null  object 
 2   tpep_dropoff_datetime  800000 non-null  object 
 3   passenger_count        791815 non-null  float64
 4   trip_distance          800000 non-null  float64
 5   RatecodeID             791815 non-null  float64
 6   store_and_fwd_flag     791815 non-null  object 
 7   PULocationID           800000 non-null  int64  
 8   DOLocationID           800000 non-null  int64  
 9   payment_type           791815 non-null  float64
 10  fare_amount            800000 non-null  float64
 11  extra                  800000 non-null  float64
 12  mta_tax                800000 non-null  float64
 13  tip_amount             800000 non-null  float64
 14  tolls_amount           800000 non-null  float64
 15  improvement_surcharge  800000 non-null  float64
 16  total_amount           800000 non-null  float64
 17  congestion_surcharge   800000 non-null  float64
dtypes: float64(13), int64(2), object(3)
memory usage: 109.9+ MB
```

In [7]:
```
# Dimensions to represent the Data.
data.ndim
```

Out[7]: 2

Unique Values and their count in the Dataset.

In [8]:
```
cat_attr = ['tpep_pickup_datetime', 'tpep_dropoff_datetime', 'store_and_fwd_flag
for col in cat_attr:
    print(f'\033[33mNo. of Unique Values for {col}\033[0m : {len(data[col].uniqu
    print(f'Unique Values : {data[col].unique()}')
```

```
No. of Unique Values for tpep_pickup_datetime : 665312
Unique Values : ['2020-01-17 18:18:36' '2020-01-25 10:49:58' '2020-01-15 07:30:0
8' ...
 '2020-01-05 11:30:02' '2020-01-31 11:26:00' '2020-01-15 12:45:27']
No. of Unique Values for tpep_dropoff_datetime : 665241
Unique Values : ['2020-01-17 18:46:24' '2020-01-25 11:07:35' '2020-01-15 07:40:0
1' ...
 '2020-01-26 00:12:55' '2020-01-05 11:36:31' '2020-01-31 11:35:51']
No. of Unique Values for store_and_fwd_flag : 3
Unique Values : ['N' 'Y' nan]
```

From the Demographics of the Dataset we can see that there is a need for Data Preprocessing as: -

1. There Exists **NULL Values** in the Dataset for certain records
2. There might be chance of Duplicate Records as well.
3. Certain Attributes like **trip_distance** has negative values where as distance is always a positive unit, **Amount**, **tip_amount**, etc., also contains negative value as minimum.

# 3. Handling Null Values.

## a. Identifying NULL Values.

In [9]:
```
data.isnull().sum()
```

Out[9]:
```
VendorID                 8185
tpep_pickup_datetime        0
tpep_dropoff_datetime       0
passenger_count          8185
trip_distance               0
RatecodeID               8185
store_and_fwd_flag       8185
PULocationID                0
DOLocationID                0
payment_type             8185
fare_amount                 0
extra                       0
mta_tax                     0
tip_amount                  0
tolls_amount                0
improvement_surcharge       0
total_amount                0
congestion_surcharge        0
dtype: int64
```

In [10]:
```
# Alternative way to find NULL Values.

missing_data = data.isnull()

for col in missing_data.columns.values.tolist():
    print(f'\033[33m{col}\033[0m')
    print(missing_data[col].value_counts())
    print('')
```

```
VendorID
VendorID
False    791815
True       8185
Name: count, dtype: int64

tpep_pickup_datetime
tpep_pickup_datetime
False    800000
Name: count, dtype: int64

tpep_dropoff_datetime
tpep_dropoff_datetime
False    800000
Name: count, dtype: int64

passenger_count
passenger_count
False    791815
True       8185
Name: count, dtype: int64

trip_distance
trip_distance
False    800000
Name: count, dtype: int64

RatecodeID
RatecodeID
False    791815
True       8185
Name: count, dtype: int64

store_and_fwd_flag
store_and_fwd_flag
False    791815
True       8185
Name: count, dtype: int64

PULocationID
PULocationID
False    800000
Name: count, dtype: int64

DOLocationID
DOLocationID
False    800000
Name: count, dtype: int64

payment_type
payment_type
False    791815
True       8185
Name: count, dtype: int64

fare_amount
fare_amount
False    800000
Name: count, dtype: int64
```

```
extra
extra
False    800000
Name: count, dtype: int64

mta_tax
mta_tax
False    800000
Name: count, dtype: int64

tip_amount
tip_amount
False    800000
Name: count, dtype: int64

tolls_amount
tolls_amount
False    800000
Name: count, dtype: int64

improvement_surcharge
improvement_surcharge
False    800000
Name: count, dtype: int64

total_amount
total_amount
False    800000
Name: count, dtype: int64

congestion_surcharge
congestion_surcharge
False    800000
Name: count, dtype: int64
```

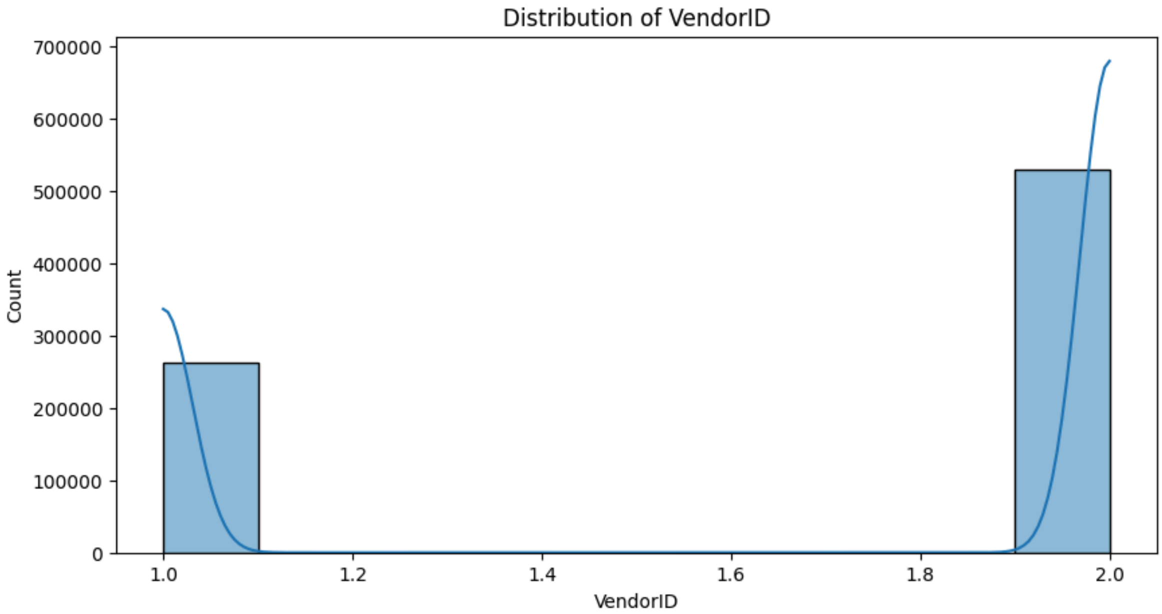
Hence, the NULL Values are: 1. VendorID, 2. passenger_count, 3. RatecodeID, 4. store_and_fwd_flag, 5. payment_type.

## b. Filling up or Removing NULL Value Records.

In [11]:
```
data['VendorID'].values
```

Out[11]:
```
array([2., 2., 2., ..., 2., 1., 1.])
```

In [12]:
```
# Analysing the Distribution of the VendorID Attribute.

plt.figure(figsize=(10, 5))
plt.title('Distribution of VendorID')
sns.histplot(data['VendorID'], kde=True, bins=10)
plt.show()
```




From the Data Distribution, and from the Dataset Description about **VendorID** Column, it describes the unique identifier for the taxi vendor or service provider. So, the records with null values in VendorID needs to be dropped as the future analysis comaprision between two types of providers is significant which gets changed if we replace the null values with value '2'.

In [13]:
```
data = data.dropna(subset=['VendorID'])
```

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

Out[14]:
```
(791815, 18)
```

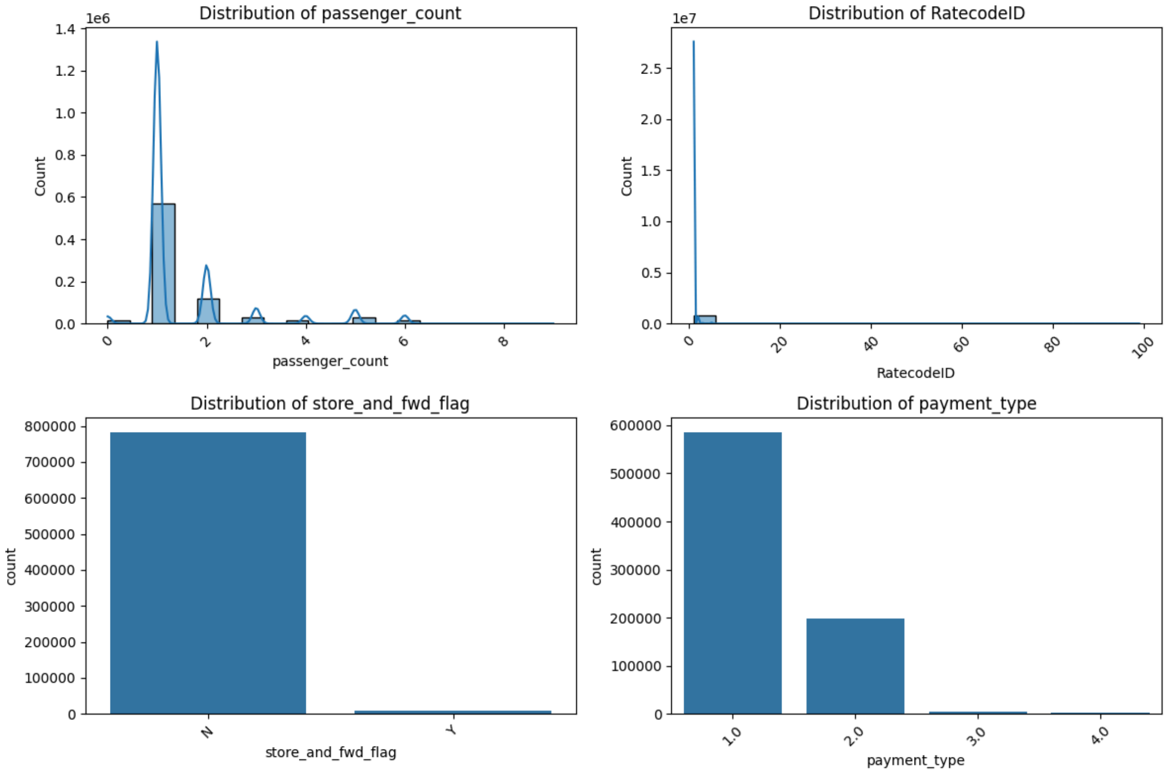
In [15]:
```
# Analysing the Distribution of the Attributes
cols = ['passenger_count', 'RatecodeID', 'store_and_fwd_flag', 'payment_type']

for col in cols:
    print(f'\033[33mThe Unique Values of the {col} are of\033[0m: {data[col].uni
```

```
The Unique Values of the passenger_count are of: [1. 2. 5. 4. 6. 0. 3. 9. 8.]
The Unique Values of the RatecodeID are of: [ 1.   2.   5.   3.   4. 99.   6.]
The Unique Values of the store_and_fwd_flag are of: ['N' 'Y']
The Unique Values of the payment_type are of: [1. 2. 3. 4.]
```

In [16]:
```
# Ensuring categorical columns to string type for plotting
data['store_and_fwd_flag'] = data['store_and_fwd_flag'].astype(str)
data['payment_type'] = data['payment_type'].astype(str)
```

In [17]:
```
# Plotting the Distributions of the Attributes

# Setting up subplots
plt.figure(figsize=(12, 8))

for i, col in enumerate(cols, 1):
    plt.subplot(2, 2, i)

    # Use histplot for numerical data, countplot for categorical
    if data[col].dtype in ['int64', 'float64']:
        sns.histplot(data[col], bins=20, kde=True)
```

```
    else:
        sns.countplot(x=data[col], order=data[col].value_counts().index)

    plt.title(f'Distribution of {col}')
    plt.xticks(rotation=45)

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




Since the attributes **passenger_count**, are highly skewed data with continuous values from 0 to 9. we can choose median to fill the missing values than the mean type.

In [18]:
```
data['passenger_count'].fillna(data['passenger_count'].median().astype(float), i
```

```
C:\Users\PMOKS\AppData\Local\Temp\ipykernel_20132\3151329574.py:1: FutureWarning:
A value is trying to be set on a copy of a DataFrame or Series through chained as
signment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work becau
se the intermediate object on which we are setting values always behaves as a cop
y.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.meth
od({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to pe
rform the operation inplace on the original object.


  data['passenger_count'].fillna(data['passenger_count'].median().astype(float),
inplace=True)
```

Remaining the attributes **RatecodeID, Store_and_fwd_flag, Payment_type** are all categorical data, and mode is optimal method to replace the null values.

In [19]:
```
cols = ['RatecodeID', 'store_and_fwd_flag', 'payment_type']
```

```python
for col in cols:
    data[col].fillna(data[col].mode()[0], inplace=True)
```

```
C:\Users\PMOKS\AppData\Local\Temp\ipykernel_20132\3323535535.py:4: FutureWarning:
A value is trying to be set on a copy of a DataFrame or Series through chained as
signment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work becau
se the intermediate object on which we are setting values always behaves as a cop
y.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.meth
od({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to pe
rform the operation inplace on the original object.


  data[col].fillna(data[col].mode()[0], inplace=True)
```

In [20]:
```python
data.isnull().sum()
```

Out[20]:
```
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
dtype: int64
```

### c. Duplicated Records

In [21]:
```python
data.duplicated().sum()
```

Out[21]:
```
0
```

## 4. Outlier Analysis.

Identifying and Handling the Outliers present in the Dataset.

### a. Identifying the Outliers

In [22]:
```python
data.reset_index(drop=True, inplace=True)
```

In [23]:
```python
data.info()
```

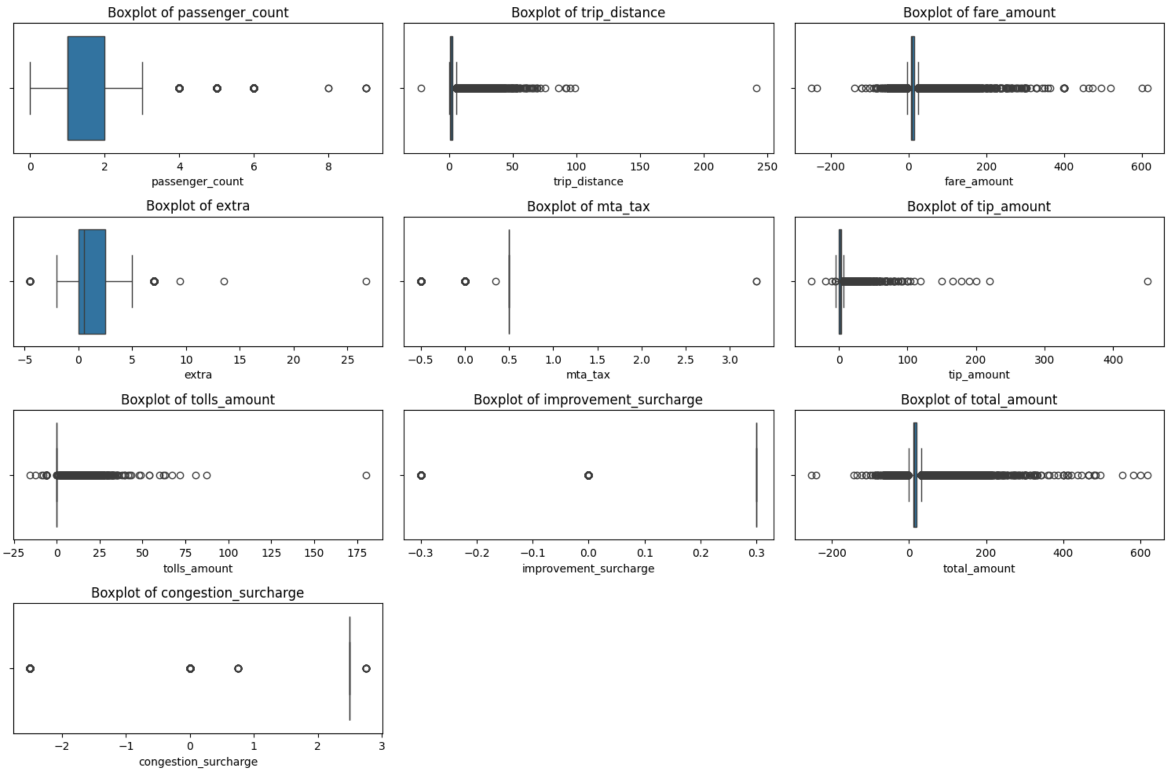
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 791815 entries, 0 to 791814
Data columns (total 18 columns):
 #   Column                 Non-Null Count   Dtype  
---  ------                 --------------   -----  
 0   VendorID               791815 non-null  float64
 1   tpep_pickup_datetime   791815 non-null  object 
 2   tpep_dropoff_datetime  791815 non-null  object 
 3   passenger_count        791815 non-null  float64
 4   trip_distance          791815 non-null  float64
 5   RatecodeID             791815 non-null  float64
 6   store_and_fwd_flag     791815 non-null  object 
 7   PULocationID           791815 non-null  int64  
 8   DOLocationID           791815 non-null  int64  
 9   payment_type           791815 non-null  object 
 10  fare_amount            791815 non-null  float64
 11  extra                  791815 non-null  float64
 12  mta_tax                791815 non-null  float64
 13  tip_amount             791815 non-null  float64
 14  tolls_amount           791815 non-null  float64
 15  improvement_surcharge  791815 non-null  float64
 16  total_amount           791815 non-null  float64
 17  congestion_surcharge   791815 non-null  float64
dtypes: float64(12), int64(2), object(4)
memory usage: 108.7+ MB
```

In [24]:

```python
# Selecting the Numerical Columns for Outlier Detection

num_cols = ['passenger_count', 'trip_distance', 'fare_amount', 'extra', 'mta_tax
            'tip_amount', 'tolls_amount', 'improvement_surcharge', 'total_amount
            'congestion_surcharge']
```

In [25]:

```python
# Creating box plots for each numerical column
plt.figure(figsize=(15, 10))
for i, col in enumerate(num_cols, 1):
    plt.subplot(4, 3, i)  # Adjust rows & columns as needed
    sns.boxplot(x=data[col])
    plt.title(f'Boxplot of {col}')

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




# b. Handling Outliers

The negative values present in the Attributes of **trip_distance, fare_amount, tip_amount, tolls_amount, total_amount** since these attributes cannot possess the neagtive values.

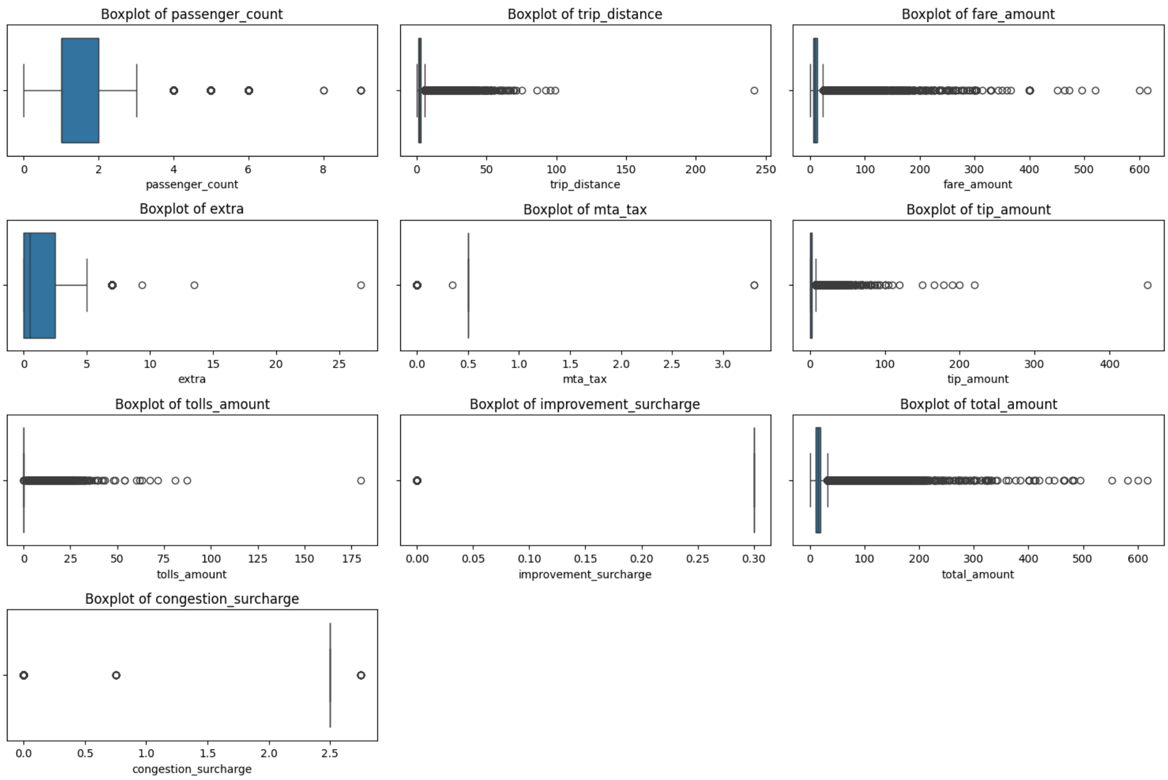
Hence removing them is optimal.

In [26]:
```
# Convert relevant columns to numeric (forcing errors='coerce' will replace non-
cols_to_check = ['fare_amount', 'tip_amount', 'total_amount', 'trip_distance', '
data[cols_to_check] = data[cols_to_check].apply(pd.to_numeric, errors='coerce')

# Identify rows with negative values
negative_values = data[
    (data['fare_amount'] < 0) |
    (data['tip_amount'] < 0) |
    (data['total_amount'] < 0) |
    (data['trip_distance'] < 0) |
    (data['tolls_amount'] < 0)
]

# Print the count of negative values
print("Negative value counts:")
print((data[cols_to_check] < 0).sum())
```

```
Negative value counts:
fare_amount      2421
tip_amount         19
total_amount     2421
trip_distance       1
tolls_amount       43
dtype: int64
```

In [27]:
```python
# Remove rows with negative values if they are errors
data = data[
    (data['fare_amount'] >= 0) &
    (data['tip_amount'] >= 0) &
    (data['total_amount'] >= 0) &
    (data['trip_distance'] >= 0) &
    (data['tolls_amount'] >= 0)
]
```

In [28]:
```python
# Creating box plots for each numerical column
plt.figure(figsize=(15, 10))
for i, col in enumerate(num_cols, 1):
    plt.subplot(4, 3, i)  # Adjust rows & columns as needed
    sns.boxplot(x=data[col])
    plt.title(f'Boxplot of {col}')

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




Applying the IQR(Inter Quartile Range) based filtering, Z-Score Filtering, and Log Transformations.

In [29]:
```python
# Function to remove outliers based on IQR
def remove_outliers_iqr(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return df[(df[column] >= lower_bound) & (df[column] <= upper_bound)]
```

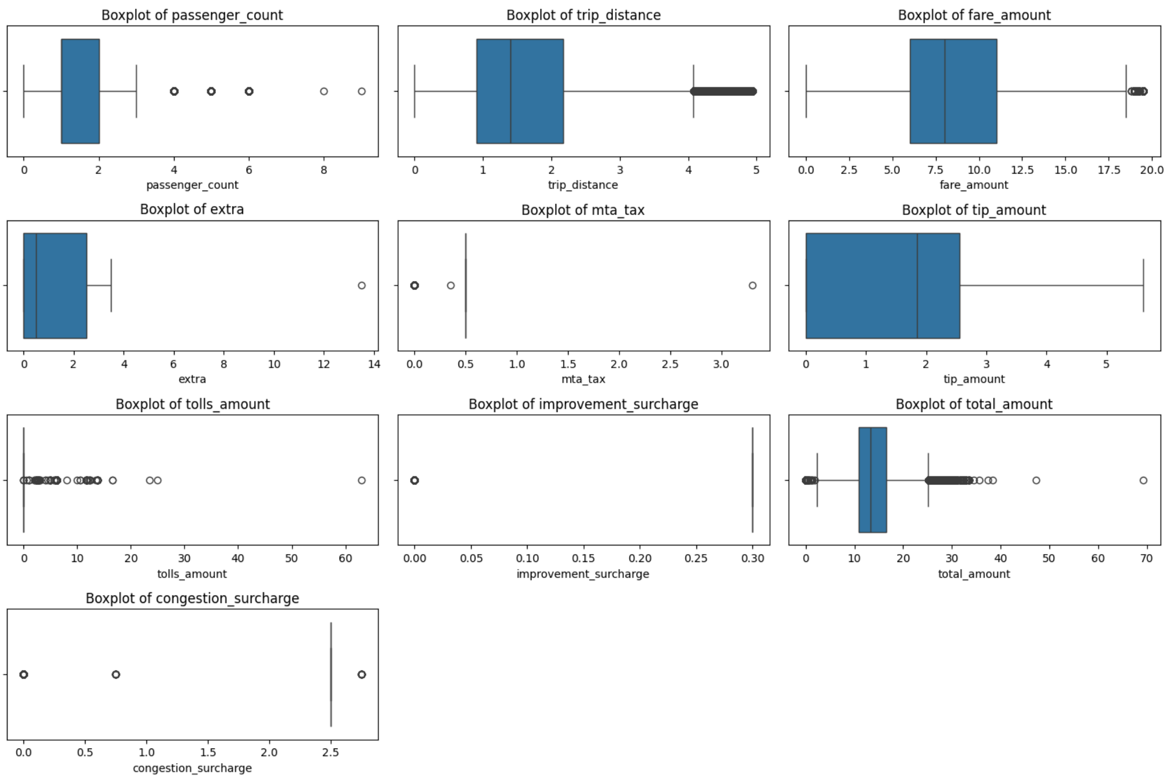
In [30]:
```python
from scipy import stats

# Function to remove outliers based on Z-score
```

```
def remove_outliers_zscore(df, column, threshold=3):
    z_scores = np.abs(stats.zscore(df[column], nan_policy='omit'))  # 'omit' to
    return df[z_scores < threshold]
```

In [31]:
```
# Apply IQR filtering to selected columns
columns_to_filter = ['trip_distance', 'fare_amount', 'tip_amount']
for col in columns_to_filter:
    data = remove_outliers_iqr(data, col)
```

In [32]:
```
# Apply Z-score filtering
for col in columns_to_filter:
    data = remove_outliers_zscore(data, col)
```

In [33]:
```
# Creating box plots for each numerical column
plt.figure(figsize=(15, 10))
for i, col in enumerate(num_cols, 1):
    plt.subplot(4, 3, i)  # Adjust rows & columns as needed
    sns.boxplot(x=data[col])
    plt.title(f'Boxplot of {col}')

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

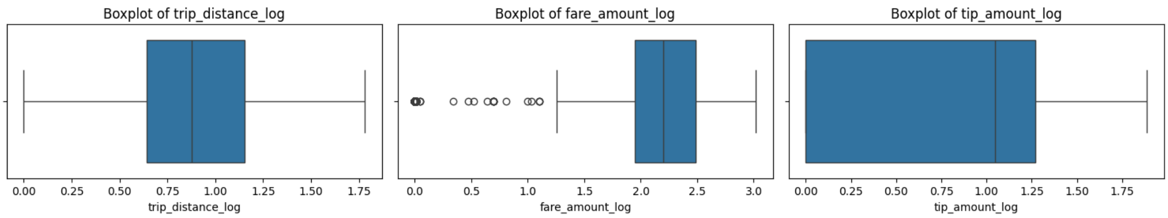



Outliers have been significantly reduced and can be even scaled down after performing the data normalization task.

# 5. Data Normalization

In [34]:
```
# Adding a small constant to avoid log(0) issues
data['trip_distance_log'] = np.log1p(data['trip_distance'])
data['fare_amount_log'] = np.log1p(data['fare_amount'])
data['tip_amount_log'] = np.log1p(data['tip_amount'])
```

In [35]:
```
temp = ['trip_distance_log', 'fare_amount_log', 'tip_amount_log']

# Creating box plots for each numerical column
plt.figure(figsize=(15, 10))
for i, col in enumerate(temp, 1):
    plt.subplot(4, 3, i)  # Adjust rows & columns as needed
    sns.boxplot(x=data[col])
    plt.title(f'Boxplot of {col}')

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




In [36]:
```
data['trip_distance'] = data['trip_distance_log']
data['fare_amount'] = data['fare_amount_log']
data['tip_amount'] = data['tip_amount_log']
```

In [37]:
```
data.describe()
```

Out[37]:

| | VendorID | passenger_count | trip_distance | RatecodeID | PULocationID | D |
|---|---|---|---|---|---|---|
| **count** | 668309.000000 | 668309.000000 | 668309.000000 | 668309.000000 | 668309.000000 | 6 |
| **mean** | 1.664781 | 1.514171 | 0.905440 | 1.009108 | 167.525471 | |
| **std** | 0.472067 | 1.152446 | 0.357931 | 0.670560 | 66.044688 | |
| **min** | 1.000000 | 0.000000 | 0.000000 | 1.000000 | 1.000000 | |
| **25%** | 1.000000 | 1.000000 | 0.641854 | 1.000000 | 125.000000 | |
| **50%** | 2.000000 | 1.000000 | 0.875469 | 1.000000 | 163.000000 | |
| **75%** | 2.000000 | 2.000000 | 1.153732 | 1.000000 | 234.000000 | |
| **max** | 2.000000 | 9.000000 | 1.781709 | 99.000000 | 265.000000 | |

# 6. Data Standardaization

In [38]:
```
data.head()
```

Out[38]:

| | VendorID | tpep_pickup_datetime | tpep_dropoff_datetime | passenger_count | trip_distar |
|---|---|---|---|---|---|
| **0** | 2.0 | 2020-01-17 18:18:36 | 2020-01-17 18:46:24 | 1.0 | 1.526( |
| **1** | 2.0 | 2020-01-25 10:49:58 | 2020-01-25 11:07:35 | 1.0 | 1.4539 |
| **2** | 2.0 | 2020-01-15 07:30:08 | 2020-01-15 07:40:01 | 1.0 | 1.0116 |
| **3** | 2.0 | 2020-01-09 06:29:09 | 2020-01-09 06:35:44 | 1.0 | 0.6259 |
| **4** | 2.0 | 2020-01-26 12:24:04 | 2020-01-26 12:29:15 | 2.0 | 0.683( |

5 rows × 21 columns

In [39]:

```
# Convert to datetime format (fixing the issue)
data['tpep_pickup_datetime'] = pd.to_datetime(data['tpep_pickup_datetime'], errc
data['tpep_dropoff_datetime'] = pd.to_datetime(data['tpep_dropoff_datetime'], er
```

In [40]:

```
# Splitting into separate columns for all rows
data['pickup_date'] = data['tpep_pickup_datetime'].dt.date
data['pickup_time'] = data['tpep_pickup_datetime'].dt.time

data['dropoff_date'] = data['tpep_dropoff_datetime'].dt.date
data['dropoff_time'] = data['tpep_dropoff_datetime'].dt.time
```

In [41]:

```
data.head()
```

Out[41]:

| | VendorID | tpep_pickup_datetime | tpep_dropoff_datetime | passenger_count | trip_distar |
|---|---|---|---|---|---|
| **0** | 2.0 | 2020-01-17 18:18:36 | 2020-01-17 18:46:24 | 1.0 | 1.526( |
| **1** | 2.0 | 2020-01-25 10:49:58 | 2020-01-25 11:07:35 | 1.0 | 1.4539 |
| **2** | 2.0 | 2020-01-15 07:30:08 | 2020-01-15 07:40:01 | 1.0 | 1.0116 |
| **3** | 2.0 | 2020-01-09 06:29:09 | 2020-01-09 06:35:44 | 1.0 | 0.6259 |
| **4** | 2.0 | 2020-01-26 12:24:04 | 2020-01-26 12:29:15 | 2.0 | 0.683( |

5 rows × 25 columns

In [42]:

```
data.drop({'tpep_pickup_datetime', 'tpep_dropoff_datetime'}, axis=1, inplace=Tru
```

In [43]:

```
data.drop({'trip_distance_log', 'fare_amount_log', 'tip_amount_log'}, axis=1, ir
```

In [44]:

```
data.head()
```

Out[44]:

| | VendorID | passenger_count | trip_distance | RatecodeID | store_and_fwd_flag | PULocatic |
|---|---|---|---|---|---|---|
| **0** | 2.0 | 1.0 | 1.526056 | 1.0 | N | |
| **1** | 2.0 | 1.0 | 1.453953 | 1.0 | N | |
| **2** | 2.0 | 1.0 | 1.011601 | 1.0 | N | |
| **3** | 2.0 | 1.0 | 0.625938 | 1.0 | N | |
| **4** | 2.0 | 2.0 | 0.683097 | 1.0 | N | |

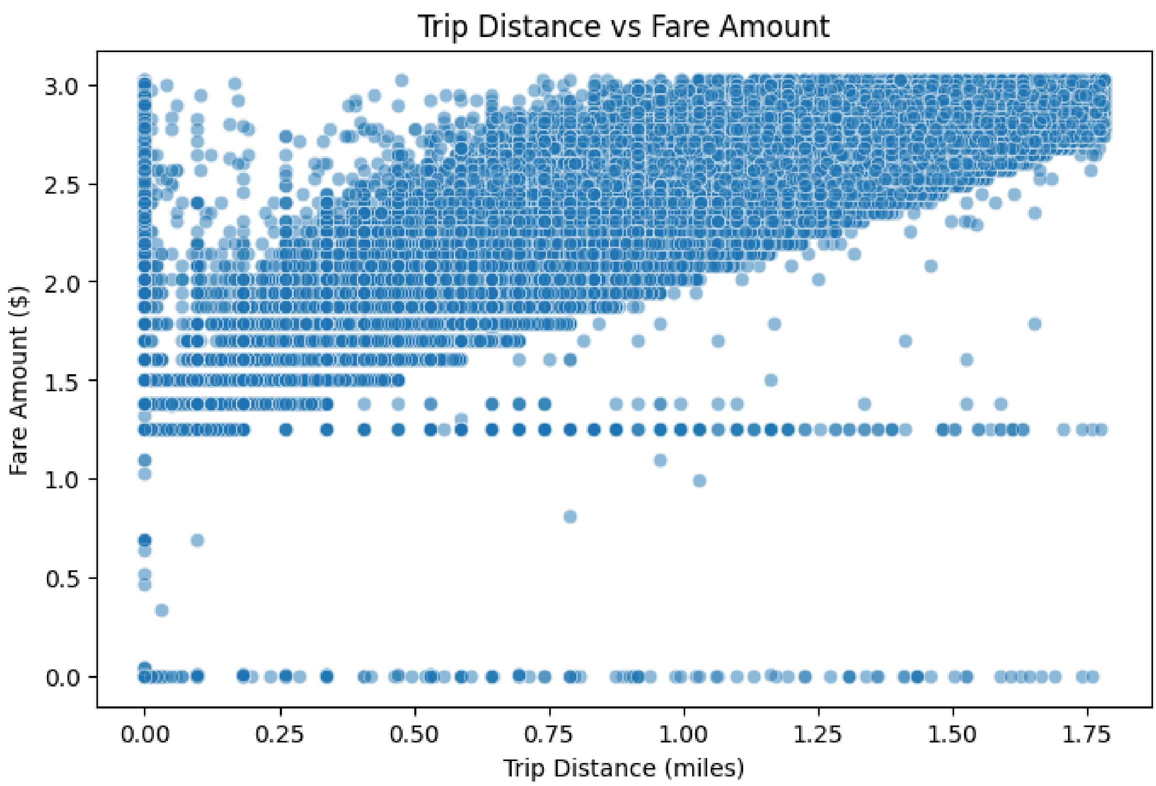
In [45]:
```
data.to_csv('TaxiFareCleaned.csv', index=False)
```

# Data Visualization.

## 1. Bivariate Analysis Graphs

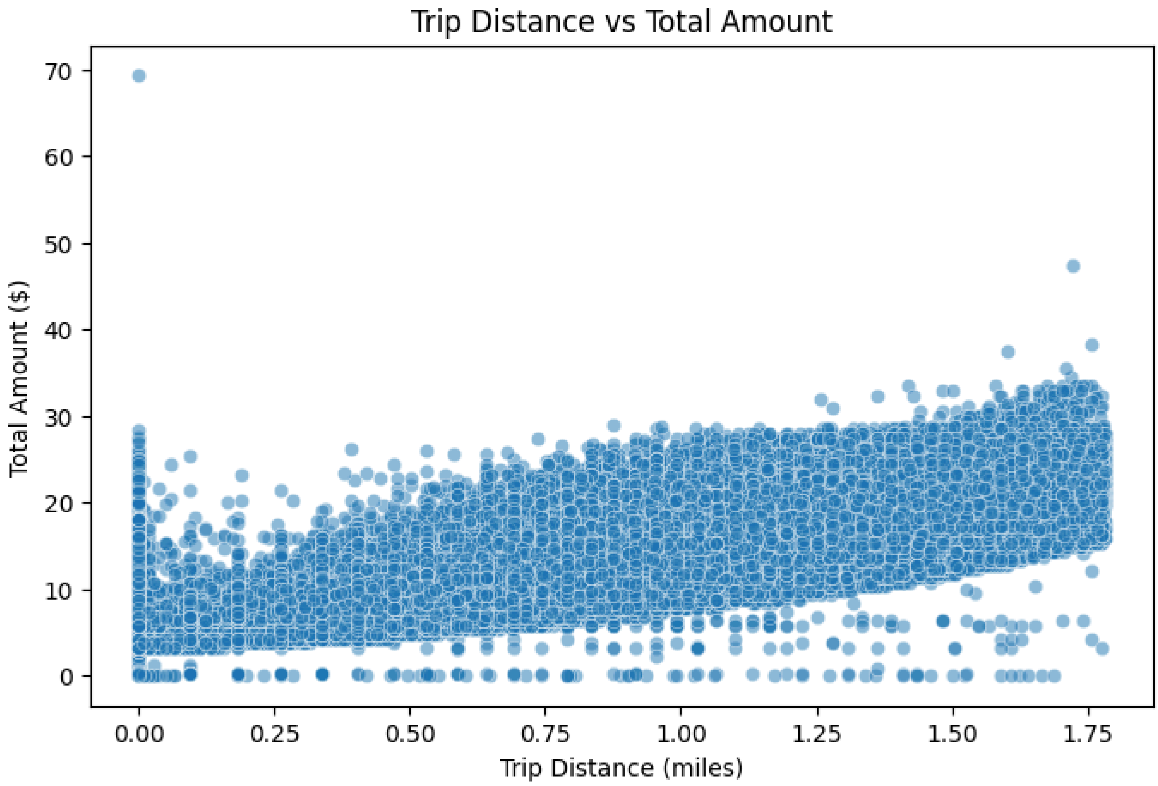
Relationship between Two Variables.

In [46]:
```
plt.figure(figsize=(8, 5))
sns.scatterplot(x=data['trip_distance'], y=data['fare_amount'], alpha=0.5)
plt.xlabel("Trip Distance (miles)")
plt.ylabel("Fare Amount ($)")
plt.title("Trip Distance vs Fare Amount")
plt.show()
```

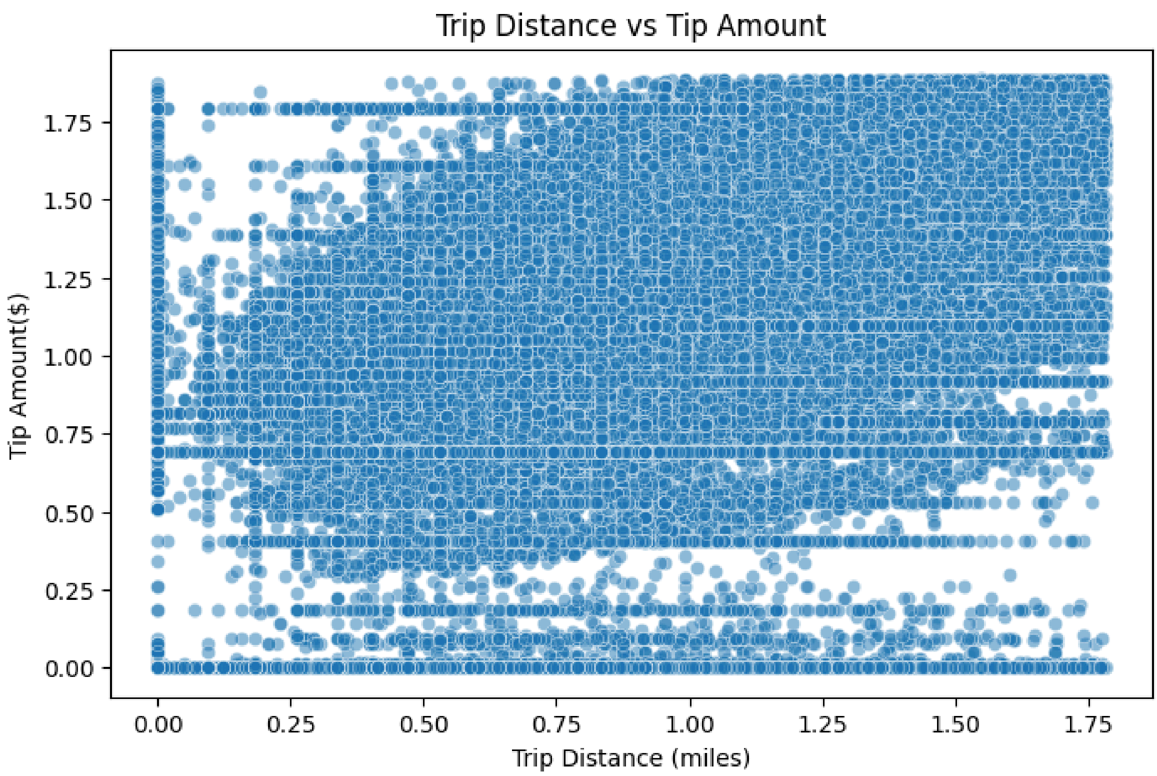



In [47]:
```
plt.figure(figsize=(8, 5))
sns.scatterplot(x=data['trip_distance'], y=data['total_amount'], alpha=0.5)
plt.xlabel("Trip Distance (miles)")
plt.ylabel("Total Amount ($)")
```

```
plt.title("Trip Distance vs Total Amount")
plt.show()
```



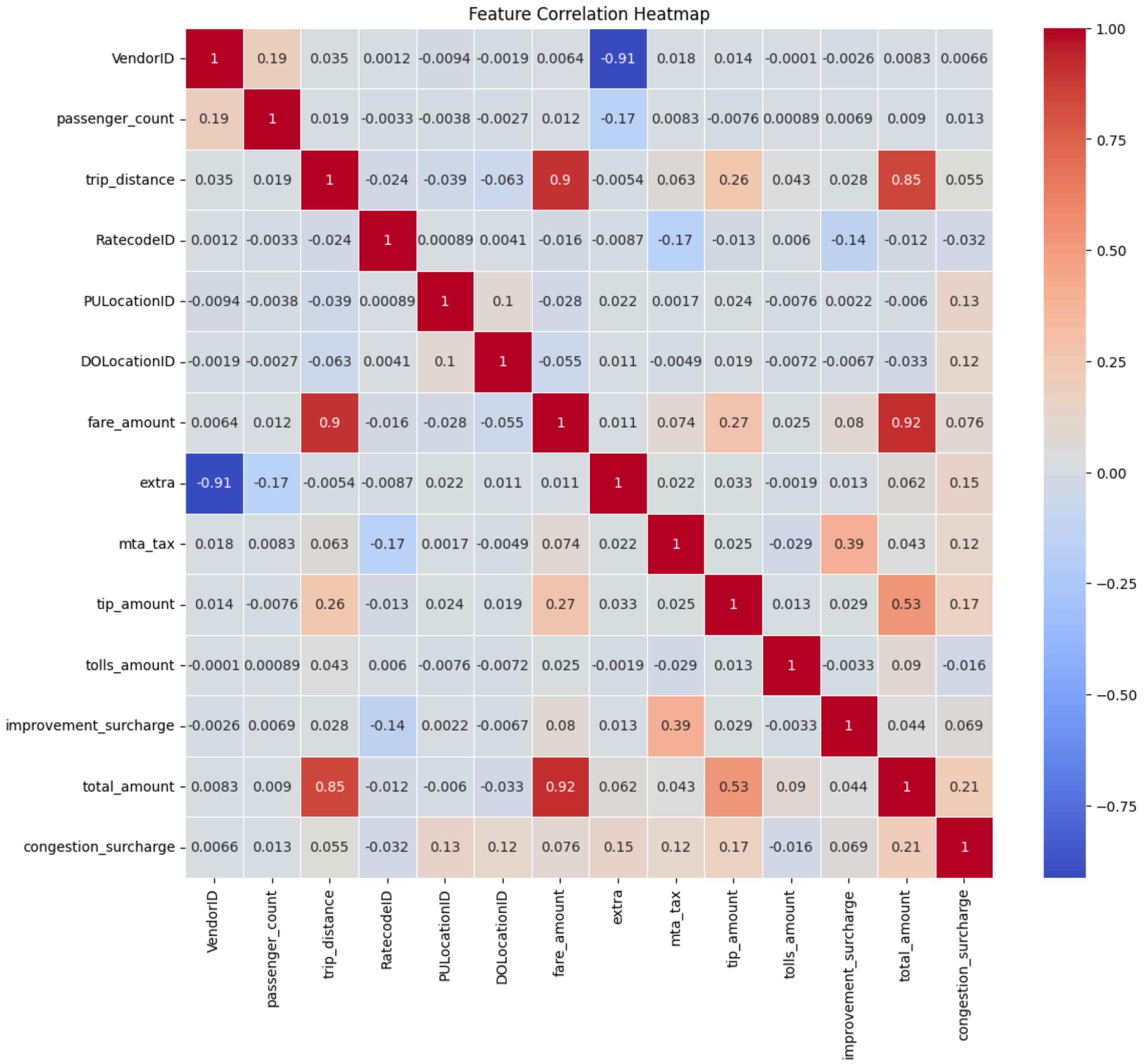

In [48]:
```
plt.figure(figsize=(8, 5))
sns.scatterplot(x=data['trip_distance'], y=data['tip_amount'], alpha=0.5)
plt.xlabel("Trip Distance (miles)")
plt.ylabel("Tip Amount($)")
plt.title("Trip Distance vs Tip Amount")
plt.show()
```




## 2. Correlation Analysis.

In [49]:

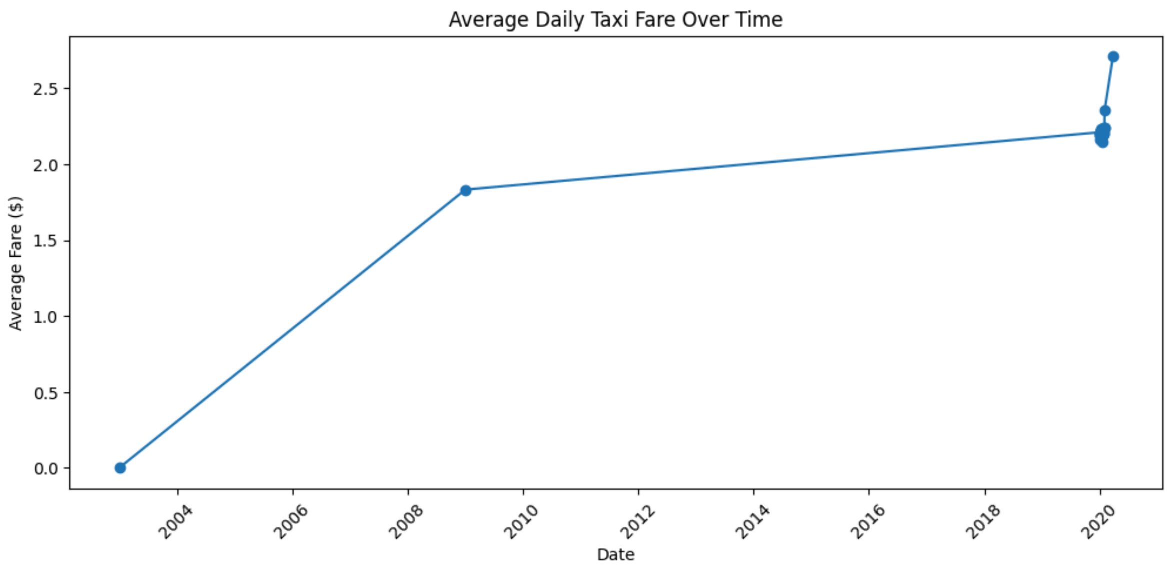
```python
plt.figure(figsize=(13, 11))
sns.heatmap(data.select_dtypes(include=["number"]).corr(), annot=True, cmap="coc
plt.title("Feature Correlation Heatmap")
plt.show()
```




## 3. Line Chart to understand the relationship

In [51]:

```python
daily_fare = data.groupby('pickup_date')['fare_amount'].mean()

plt.figure(figsize=(12, 5))
plt.plot(daily_fare.index, daily_fare.values, marker='o', linestyle='-')
plt.title("Average Daily Taxi Fare Over Time")
plt.xlabel("Date")
plt.ylabel("Average Fare ($)")
plt.xticks(rotation=45)
plt.show()
```


Average Daily Taxi Fare Over Time
Average Fare ($)
0.0
0.5
1.0
1.5
2.0
2.5
2004
2006
2008
2010
2012
2014
2016
2018
2020
Date