

BIG DATA ANALYTICS & DATA VISUALIZATION

By

Dhyey Bhandari

Problem Statement

Urban ride-hailing services like **Uber** and **Lyft** generate massive volumes of trip data every day across complex metropolitan areas. However, these companies often face challenges in effectively managing and analyzing this data to ensure operational efficiency and customer satisfaction. One of the primary concerns is the **inefficient allocation of drivers**, especially during **peak hours** or in **underserved zones**, which leads to longer **wait times, poor service quality, and loss of potential revenue**.

Another major issue is the **unpredictability of trip durations and fares**, which can be influenced by **traffic patterns, weather conditions, and route selection**. Without accurate predictive insights, both customers and drivers experience uncertainty, which negatively affects trust and platform reliability. In addition, companies encounter difficulty in **detecting fare anomalies**, such as unjustified surcharges or inconsistencies in pricing across similar trips.

From a business intelligence perspective, there is a growing need to **analyze trip volumes, revenue trends, and geographic patterns**. Identifying busiest hours, days, and months, as well as top pickup and dropoff locations, is crucial for strategic planning and resource deployment. However, manually deriving these insights from raw trip data is both time-consuming and error-prone.

In summary, the lack of integrated, **data-driven solutions** for these multifaceted problems hampers decision-making, efficiency, and long-term strategic planning in the ride-hailing industry. This project aims to address these issues by transforming raw data into actionable insights through analysis and predictive modeling.

Big Data Visualization

Overview of the “NYC For-Hire Vehicles Trip” dataset

Introduction:

The big dataset contains detailed **5.4 GB** of **trip-level records** for **For-Hire Vehicle (FHV) services** in New York City, comprising millions of entries collected over a defined period. It captures essential operational and financial information related to individual FHV rides, including identifiers for the dispatching company (such as Uber or Lyft), timestamps for pickup and dropoff, and pickup/dropoff location IDs.

Additional features include trip distance, trip duration, base passenger fare, tolls, surcharges (Black Car Fund, congestion, and airport fees), taxes, tips, and the final amount paid to the driver. Analyzing this data provides valuable insights into travel patterns, fare structures, and driver earnings, supporting optimization in urban mobility planning, policy-making, and FHV service management.

Source:

The big dataset is provided by <https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page>

Dataset Structure:

The NYC FHV trip dataset is stored in **Parquet format** and comprises a total of **239,448,588 records (rows)** and **15 columns (features)**. Each row represents a single for-hire vehicle trip recorded in New York City.

Data Headers:

Column Name	Data Type	Description
dispatching_base_num	string	Identifies the FHV base/company (e.g., Uber, Lyft) that dispatched the vehicle.
pickup_datetime	timestamp	The date and time when the trip began.
dropoff_datetime	timestamp	The date and time when the trip ended.
pulocationid	int	TLC Taxi Zone ID for the pickup location.
dolocationid	int	TLC Taxi Zone ID for the dropoff location.
trip_miles	float	Distance of the trip in miles.
trip_time	int	Duration of the trip in seconds.
base_passenger_fare	float	The base fare paid by the passenger, excluding taxes and surcharges.
tolls	float	Toll charges incurred during the trip.
bcf	float	Black Car Fund surcharge, a mandatory fee for FHV rides in NYC.
sales_tax	float	Sales tax applicable to the trip.
congestion_surcharge	float	Additional charge for trips in high-traffic zones during peak hours.
airport_fee	float	Fixed fee applied to trips to/from NYC airports.
tips	float	Tip amount paid by the customer.
driver_pay	float	Total amount paid to the driver for the trip, including base fare and tips.

AWS aspects that are explored:

- 1. AWS S3:** Creating & uploading 5.4 GB of big data into the S3 bucket created.
- 2. AWS Athena:** To run SQL queries and process data to gain valuable insights.
- 3. AWS Cloud9:** Creating an environment for Jupyter Notebook to visualize data on aggregated output given by Athena queries.

Steps Followed:

1. Creating S3 bucket:

Bucket Name: nyc-fhv-data-bucket

Region: us-east-1

S3 Search [Alt-S] United States (N. Virginia) us-east-1 Create bucket

Create bucket Info

Buckets are containers for data stored in S3.

General configuration

AWS Region: US East (N. Virginia) us-east-1

Bucket type: Info

General purpose Recommended for most use cases and access patterns. General purpose buckets are the original S3 bucket type. They allow a wide range of storage classes that automatically store objects across multiple Availability Zones.

Glacier Recommended for low-latency use cases. These buckets use only the S3 Express One Zone storage class, which provides faster processing of data within a single Availability Zone.

Bucket name: Info Bucket names must be 3 to 255 characters and unique within the global namespace. Bucket names must also begin and end with a letter or number. Valid characters are a-z, 0-9, periods (.), and hyphens. [Learn more](#)

Copy settings from existing bucket - optional Use the base settings in the following configuration to copy.

[Choose bucket](#) Format: `s3://nyc-fhv-data-bucket`

Object Ownership Info

Control ownership of objects written to this bucket from other AWS accounts and the use of access control lists (ACLs). Object ownership determines who can specify access to objects.

ACLs disabled (recommended) All objects in the bucket are owned by the owner, access to the bucketed objects is specified using object policies.

ACLs enabled Objects in this bucket can be owned by other AWS accounts. Access to the bucketed resources can be specified using ACLs.

Object Ownership

Bucket owner is informed

Block Public Access settings for this bucket

Public access is granted to buckets and objects through server control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to this bucket and its objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your application will work correctly without public access. If you require some level of public access to the bucket or objects within, you can customize the individual settings below to suit your specific storage use cases. [Learn more](#)

Block all public access Turning this setting on is the same as turning on all four settings below. Each of the following settings are independent of one another.

Block public access to buckets and objects granted through new access control lists (ACLS) AWS S3 block public access permission applied to newly added buckets or objects, but present the option of new public ACLs for existing buckets and objects. The setting does not change any existing permissions that allow public access to S3 resources using ACLs.

Block public access to buckets and objects granted through any access control lists (ACLS) AWS S3 ignores ACLs that grant public access to buckets and objects.

Block public access to buckets and objects granted through new public bucket or access point policies AWS S3 block new buckets and access point policies that grant public access to buckets and objects. This setting does not change any existing policies that allow public access to S3 resources.

Block public and cross-account access to buckets and objects through any public bucket or access point policies AWS S3 ignores public and cross-account access for buckets or access points with policies that grant public access to buckets and objects.

Bucket Versioning

Versioning is a means of keeping multiple versions of an object in the same bucket. You can use versioning to preserve, retrieve, and restore every version of every object stored in your Amazon S3 bucket. With versioning, you can easily recover from both unintended user actions and application failures. [Learn more](#)

Bucket Versioning

Disable

Enable

Tags = optional (0)

You can use bucket tags to track storage costs and organize buckets. [Learn more](#)

No tags associated with this bucket.

[Add tag](#)

Default encryption Info

Server-side encryption is automatically applied to new objects stored in this bucket.

Encryption type Info

Server-side encryption with Amazon S3 managed keys (SSE-S3)

Server-side encryption with AWS Key Management Service keys (SSE-KMS)

Dual-layer server-side encryption with AWS Key Management Service keys (DSS-EKMS)

Secure your objects with two separate layers of encryption. For details on pricing, see SSE-EKMS pricing on the Storage fees of the [Amazon S3 pricing](#) page.

Bucket Key Info Using an S3 Bucket Key for SSE-KMS reduces encryption costs by lowering costs to AWS KMS. S3 Bucket Keys aren't supported for DSS-EKMS. [Learn more](#)

Disable

Advanced settings

After creating the bucket, you can upload files and folders to the bucket, and configure additional bucket settings.

[Cancel](#) [Create bucket](#)

The screenshot shows the AWS S3 console with a green banner at the top stating "Successfully created bucket 'nyc-fhv-data-bucket'". Below the banner, there's an "Account snapshot" section with a link to "View Storage Lens dashboard". Under "General purpose buckets", there is one entry: "nyc-fhv-data-bucket" (1 item). The table includes columns for Name, AWS Region, IAM Access Analyzer, and Creation date. The bucket was created on May 23, 2025, at 17:57:24 (UTC+05:30) in US East (N. Virginia) (us-east-1).

Name	AWS Region	IAM Access Analyzer	Creation date
nyc-fhv-data-bucket	US East (N. Virginia) us-east-1	View analyzer for us-east-1	May 23, 2025, 17:57:24 (UTC+05:30)

2. Uploading Data: 12 parquet files, 5.4 GB

The screenshot shows the AWS S3 "Upload" interface. At the top, it says "Add the files and folders you want to upload to S3. To upload a file larger than 160GB, use the AWS CLI, AWS SDKs or Amazon S3 REST API. [Learn more](#)". Below this is a large blue dashed box for dragging and dropping files. A table titled "Files and folders (12 total, 5.4 GB)" lists the 12 parquet files. The table has columns for Name, Folder, Type, and Size. The files are all named "fhvhv_tripdata_2024-XX.parquet" and are located in the "fhv_big_dataset/" folder. The sizes range from 444.4 MB to 475.5 MB. At the bottom, there are sections for "Destination" (set to "s3://nyc-fhv-data-bucket"), "Permissions" (Grant public access), and "Properties" (Specify storage class, encryption settings, tags, and more). The "Upload" button is highlighted in orange.

Name	Folder	Type	Size
fhvhv_tripdata_2024-04.parquet	fhv_big_dataset/	-	454.1 MB
fhvhv_tripdata_2024-05.parquet	fhv_big_dataset/	-	475.5 MB
fhvhv_tripdata_2024-06.parquet	fhv_big_dataset/	-	462.0 MB
fhvhv_tripdata_2024-07.parquet	fhv_big_dataset/	-	446.7 MB
fhvhv_tripdata_2024-08.parquet	fhv_big_dataset/	-	444.4 MB
fhvhv_tripdata_2024-09.parquet	fhv_big_dataset/	-	444.9 MB
fhvhv_tripdata_2024-10.parquet	fhv_big_dataset/	-	462.6 MB
fhvhv_tripdata_2024-11.parquet	fhv_big_dataset/	-	458.0 MB
fhvhv_tripdata_2024-12.parquet	fhv_big_dataset/	-	484.0 MB
fhvhv_tripdata_2025-01.parquet	fhv_big_dataset/	-	468.3 MB

aws | Search [Alt+S] United States (N. Virginia) | vocabs/user3230148=dhyey.h.bhandari@nuv.ac.in @ 2501-8822-7884 ▾

Upload succeeded
For more information, see the [Files and folders](#) table.

Upload: status

After you navigate away from this page, the following information is no longer available.

Summary

Destination	Succeeded	Failed
s3://nyc-fhv-data-bucket	12 files, 5.4 GB (100.00%)	0 files, 0 B (0%)

Files and folders (12 total, 5.4 GB)

Name	Folder	Type	Size	Status	Error
fhvhv_tripdata_2024-04.parquet	fhv_big_dataset/	-	454.1 MB	✓ Succeeded	-
fhvhv_tripdata_2024-05.parquet	fhv_big_dataset/	-	475.5 MB	✓ Succeeded	-
fhvhv_tripdata_2024-06.parquet	fhv_big_dataset/	-	462.0 MB	✓ Succeeded	-
fhvhv_tripdata_2024-07.parquet	fhv_big_dataset/	-	446.7 MB	✓ Succeeded	-
fhvhv_tripdata_2024-08.parquet	fhv_big_dataset/	-	444.4 MB	✓ Succeeded	-
fhvhv_tripdata_2024-09.parquet	fhv_big_dataset/	-	444.9 MB	✓ Succeeded	-
fhvhv_tripdata_2024-10.parquet	fhv_big_dataset/	-	462.6 MB	✓ Succeeded	-
fhvhv_tripdata_2024-11.parquet	fhv_big_dataset/	-	458.0 MB	✓ Succeeded	-
fhvhv_tripdata_2024-12.parquet	fhv_big_dataset/	-	484.0 MB	✓ Succeeded	-
fhvhv_tripdata_2025-01.parquet	fhv_big_dataset/	-	468.3 MB	✓ Succeeded	-

CloudShell Feedback © 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

3. Database Creation:

Database Name: nyc_fhv

aws | Search [Alt+S] United States (N. Virginia) | vocabs/user3230148=dhyey.h.bhandari@nuv.ac.in @ 2501-8822-7884 ▾

Amazon Athena > Query editor

Editor Recent queries Saved queries Settings Workgroup primary

Data

Data source: AwsDataCatalog Catalog: None Database: default

Tables and views Create

Query 3 :

```
1 CREATE DATABASE nyc_fhv;
```

SQL Ln 1, Col 25

Run Explain Cancel Clear Create Reuse query results up to 60 minutes ago

Query results Query stats

Completed Time in queue: 82 ms Run time: 462 ms Data scanned: -

Query successful.

CloudShell Feedback © 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

4. Table Creation:

Table Name: trips

The screenshot shows the Amazon Athena Query Editor interface. On the left, the 'Data' sidebar is open, showing 'AwsDataCatalog' selected as the Data source, 'None' as the Catalog, and 'default' as the Database. Under 'Tables and views', there are 0 tables and 0 views. The main area displays a SQL query for creating an external table:

```
1 CREATE EXTERNAL TABLE nyc_fhv.trips (
2     hvfhs_license_num STRING,
3     dispatching_base_num STRING,
4     originating_base_num STRING,
5     request_datetime TIMESTAMP,
6     on_scene_datetime TIMESTAMP,
7     pickup_datetime TIMESTAMP,
8     dropoff_datetime TIMESTAMP,
9     PUlocationID INT,
10    DOlocationID INT,
11    trip_miles DOUBLE,
12    trip_time DOUBLE,
13    base_passenger_fare DOUBLE,
14    tolls DOUBLE,
15    bcf DOUBLE,
```

The status bar at the bottom indicates the query was completed successfully.

5. Running SQL Queries:

Hourly Trips:

```
SELECT
    hour(pickup_datetime) AS trip_hour,
    COUNT(*) AS total_trips
FROM nyc_fhv.trips
GROUP BY hour(pickup_datetime)
ORDER BY trip_hour;
```

Daily Trips:

```
SELECT
    date(pickup_datetime) AS trip_date,
    COUNT(*) AS total_trips
FROM nyc_fhv.trips
```

```
GROUP BY date(pickup_datetime)
ORDER BY trip_date;
```

Weekly Trips:

```
SELECT
    year(pickup_datetime) AS trip_year,
    week(pickup_datetime) AS trip_week,
    COUNT(*) AS total_trips
FROM nyc_fhv.trips
GROUP BY year(pickup_datetime), week(pickup_datetime)
ORDER BY trip_year, trip_week;
```

Monthly Trips:

```
SELECT
    year(pickup_datetime) AS trip_year,
    month(pickup_datetime) AS trip_month,
    COUNT(*) AS total_trips
FROM nyc_fhv.trips
GROUP BY year(pickup_datetime), month(pickup_datetime)
ORDER BY trip_year, trip_month;
```

Peak Hour of the Day:

```
SELECT
    hour(pickup_datetime) AS trip_hour,
    COUNT(*) AS trip_count
FROM nyc_fhv.trips
GROUP BY hour(pickup_datetime)
ORDER BY trip_count DESC
LIMIT 10;
```

Peak Day of the Week:

```
SELECT  
    day_of_week(pickup_datetime) AS day_of_week,  
    COUNT(*) AS trip_count  
FROM nyc_fhv.trips  
GROUP BY day_of_week(pickup_datetime)  
ORDER BY trip_count DESC;
```

Peak Week:

```
SELECT  
    year(pickup_datetime) AS trip_year,  
    week(pickup_datetime) AS trip_week,  
    COUNT(*) AS trip_count  
FROM nyc_fhv.trips  
GROUP BY year(pickup_datetime), week(pickup_datetime)  
ORDER BY trip_count DESC  
LIMIT 10;
```

Monthly Revenue Trend:

```
SELECT  
    year(pickup_datetime) AS trip_year,  
    month(pickup_datetime) AS trip_month,  
    SUM(base_passenger_fare + tolls + bcf + sales_tax + congestion_surcharge + airport_fee  
    + tips) AS total_revenue  
FROM nyc_fhv.trips  
GROUP BY year(pickup_datetime), month(pickup_datetime)  
ORDER BY trip_year, trip_month;
```

Daily Average Fare:

```
SELECT
```

```
date(pickup_datetime) AS trip_date,  
AVG(base_passenger_fare) AS avg_fare  
FROM nyc_fhv.trips  
GROUP BY date(pickup_datetime)  
ORDER BY trip_date;
```

Most Popular Routes:

```
SELECT  
PULocationID,  
DOlocationID,  
COUNT(*) AS trip_count  
FROM nyc_fhv.trips  
GROUP BY PULocationID, DOlocationID  
ORDER BY trip_count DESC  
LIMIT 10;
```

Top Pickup Location:

```
SELECT  
PULocationID,  
COUNT(*) AS pickup_count  
FROM nyc_fhv.trips  
GROUP BY PULocationID  
ORDER BY pickup_count DESC  
LIMIT 10;
```

Top Dropoff Location:

```
SELECT  
DOlocationID,  
COUNT(*) AS dropoff_count  
FROM nyc_fhv.trips
```

```
GROUP BY DOLocationID  
ORDER BY dropoff_count DESC  
LIMIT 10;
```

Trips by Company:

```
SELECT  
CASE  
    WHEN hvfhs_license_num LIKE 'HV0003%' THEN 'Uber'  
    WHEN hvfhs_license_num LIKE 'HV0005%' THEN 'Lyft'  
    ELSE 'Other'  
END AS company,  
COUNT(*) AS total_trips  
FROM nyc_fhv.trips  
GROUP BY  
CASE  
    WHEN hvfhs_license_num LIKE 'HV0003%' THEN 'Uber'  
    WHEN hvfhs_license_num LIKE 'HV0005%' THEN 'Lyft'  
    ELSE 'Other'  
END;
```

Revenue by Company:

```
SELECT  
CASE  
    WHEN hvfhs_license_num LIKE 'HV0003%' THEN 'Uber'  
    WHEN hvfhs_license_num LIKE 'HV0005%' THEN 'Lyft'  
    ELSE 'Other'  
END AS company,  
SUM(base_passenger_fare + tolls + bcf + sales_tax + congestion_surcharge + airport_fee  
+ tips) AS total_revenue  
FROM nyc_fhv.trips
```

```
GROUP BY  
CASE  
    WHEN hvfhs_license_num LIKE 'HV0003%' THEN 'Uber'  
    WHEN hvfhs_license_num LIKE 'HV0005%' THEN 'Lyft'  
    ELSE 'Other'  
END;
```

Shared vs Solo Rides:

```
SELECT  
    shared_request_flag,  
    COUNT(*) AS trip_count  
FROM nyc_fhv.trips  
GROUP BY shared_request_flag;
```

WAV (Wheelchair Accessible Vehicle) Requests:

```
SELECT  
    wav_request_flag,  
    COUNT(*) AS trip_count  
FROM nyc_fhv.trips  
GROUP BY wav_request_flag;
```

6. Creating Cloud9 environment:

Environment Name: fhv-big-data-viz

EC2 Instance Type: t3.small (2 GiB RAM + 2 vCPU)

Screenshot of the AWS Cloud9 'Create environment' configuration page.

Details

Name: fhv-big-data-viz

Description - optional: Big data visualization of NYC hired vehicles trip data.

Environment type (Info): New EC2 instance

Determines what the Cloud9 IDE will run on.

New EC2 instance: Cloud9 creates an EC2 instance in your account. The configuration of your EC2 instance cannot be changed by Cloud9 after creation.

Existing compute: You have an existing instance or server that you'd like to use.

New EC2 instance

Instance type (Info): t3.small (2 GiB RAM + 2 vCPU)

The memory and CPU of the EC2 instance that will be created for Cloud9 to run on.

t2.micro (1 GiB RAM + 1 vCPU): Free-tier eligible. Ideal for educational users and exploration.

t3.small (2 GiB RAM + 2 vCPU): Recommended for small web projects.

m5.large (8 GiB RAM + 2 vCPU): Recommended for production and most general-purpose development.

Platform (Info): Amazon Linux 2023

This will be installed on your EC2 instance. We recommend Amazon Linux 2023.

Timeout: How long Cloud9 can be inactive (no user input) before auto-hibernating. This helps prevent unnecessary charges.

4 hours

Network settings (Info)

Connection: AWS Systems Manager (SSM)

How your environment is accessed.

AWS Systems Manager (SSM): Accesses environment via SSM without opening inbound ports (no ingress).

Secure Shell (SSH): Accesses environment directly via SSH, opens inbound ports.

VPC settings (Info)

Tags - optional (Info): A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

The following IAM resources will be created in your account

- AWSserviceRoleForAWSCloud9 - AWS Cloud9 creates a service-linked role for you. This allows AWS Cloud9 to call other AWS services on your behalf. You can delete the role from the AWS IAM console once you no longer have any AWS Cloud9 environments. [Learn more](#)
- AWSCloud9SSMAccessRole and AWSCloud9SSMInstanceProfile - A service role and an instance profile are automatically created if Cloud9 accesses its EC2 instance through AWS Systems Manager. If your environments no longer require EC2 instances that block incoming traffic, you can delete these roles using the AWS IAM console. [Learn more](#)

[Cancel](#) [Create](#)

CloudShell Feedback © 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

7. Installing libraries in Cloud9 terminal:

Python Data Processing & Visualization Libraries: pandas, pyathena, sqlalchemy, matplotlib, seaborn, plotly, notebook

```

Go to Anything (Ctrl-P)
aws
voclabs:/environment $ python3 -m venv venv
voclabs:/environment $ python3 -m venv venv
voclabs:/environment $ source venv/bin/activate
(venv) voclabs:/environment $ pip install pandas pythana sqlalchemy matplotlib seaborn plotly notebook
Collecting pandas
  Downloading pandas-2.2.3-cp39-cp39-manylinux_2_17_x86_64_manylinux2014_x86_64.whl (13.1 MB)
    3.1 MB 22.1 MB/s
Collecting pythana
  Downloading pythana-3.14.0-py3-none-any.whl (76 kB)
    76 kB 11.0 MB/s
Collecting sqlalchemy
  Downloading sqlalchemy-2.0.41-cp39-cp39-manylinux_2_17_x86_64_manylinux2014_x86_64.whl (3.2 MB)
    3.2 MB 66.6 MB/s
Collecting matplotlib
  Downloading matplotlib-3.9.4-cp39-cp39-manylinux_2_17_x86_64_manylinux2014_x86_64.whl (8.3 MB)
    8.3 MB 3.3 MB/s
Collecting seaborn
  Downloading seaborn-0.13.2-py3-none-any.whl (294 kB)
    294 kB 89.9 MB/s
Collecting plotly
  Downloading plotly-6.1.1-py3-none-any.whl (16.1 MB)
    16.1 MB 37 kB/s
Collecting notebook
  Downloading notebook-7.4.2-py3-none-any.whl (14.3 MB)
    14.3 MB 33 kB/s
Collecting tzdata>=2022.7
  Downloading tzdata-2025.2-py2.py3-none-any.whl (347 kB)
    347 kB 41.6 MB/s
Collecting pytz>=2020.1
  Downloading pytz-2025.2-py2.py3-none-any.whl (509 kB)
    509 kB 88.7 MB/s
Collecting numpy>=1.22.4
  Downloading numpy-2.0.2-cp39-cp39-manylinux_2_17_x86_64_manylinux2014_x86_64.whl (19.5 MB)
    19.5 MB 10 kB/s
Collecting python-dateutil>=2.8.2
  Downloading python_dateutil-2.9.0.post0-py2.py3-none-any.whl (229 kB)
    229 kB 50.4 MB/s
Collecting boto3>=1.26.4
  Downloading boto3-1.38.22-py3-none-any.whl (139 kB)
    139 kB

```

AWS profile default

8. Initiating server for Jupyter Notebook:

```

Go to Anything (Ctrl-P)
aws
pythana - "ip-127-31-21-1 x" ⓘ
http://127.0.0.1:8881/tree?token=d574ebd52b24516176963b55e08eb13eed7353c81b72b4
Shut down this Jupyter server (y/[n])? y
[1] 2025-05-23 17:56:05.219 ServerApp[ ] Shutting down 5 extensions
(venv) voclabs:/environment $ ls of :1:8880
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
jupyter-n 2839 ec2-user 6u IPv4 40846 0t0 TCP *:webcache (LISTEN)
(venv) voclabs:/environment $ kill -9 2839
(venv) voclabs:/environment $ jupyter notebook --ip=0.0.0.0 --port=8888 --no-browser
[1] 2025-05-23 17:56:54.282 ServerApp[ ] jupyter_lsp | extension was successfully linked.
[1] 2025-05-23 17:56:54.283 ServerApp[ ] jupyter_server_terminals | extension was successfully linked.
[1] 2025-05-23 17:56:54.283 ServerApp[ ] jupyterlab | extension was successfully linked.
[1] 2025-05-23 17:56:54.284 ServerApp[ ] notebook | extension was successfully linked.
[1] 2025-05-23 17:56:54.505 ServerApp[ ] notebook_shim | extension was successfully linked.
[1] 2025-05-23 17:56:54.526 ServerApp[ ] notebook_shim | extension was successfully loaded.
[1] 2025-05-23 17:56:54.529 ServerApp[ ] jupyter_lsp | extension was successfully loaded.
[1] 2025-05-23 17:56:54.531 ServerApp[ ] jupyter_server_terminals | extension was successfully loaded.
[1] 2025-05-23 17:56:54.533 LabApp[ ] JupyterLab extension loaded from /home/ec2-user/environment/venv/lib64/python3.9/site-packages/jupyterlab
[1] 2025-05-23 17:56:54.538 LabApp[ ] JupyterLab application directory is /home/ec2-user/environment/venv/share/jupyter/lab
[1] 2025-05-23 17:56:54.539 LabApp[ ] Extension Manager is /tmp/
[1] 2025-05-23 17:56:54.569 ServerApp[ ] notebook | extension was successfully loaded.
[1] 2025-05-23 17:56:54.569 ServerApp[ ] notebook_shim | extension was successfully loaded.
[1] 2025-05-23 17:56:54.573 ServerApp[ ] Serving notebooks from local directory: /home/ec2-user/environment
[1] 2025-05-23 17:56:54.574 ServerApp[ ] Jupyter Server 2.16.0 is running at:
[1] 2025-05-23 17:56:54.574 ServerApp[ ]   file:///home/ec2-user/.local/share/jupyter/runtime/jpserver-4874-open.html
Or copy and paste one of these URLs:
[1] 2025-05-23 17:56:54.574 ServerApp[ ]   http://ip-127-31-21-131.ec2.internal:8888/tree?token=f8bc7d3cb9f9cd3a36178616518c67a3e6a084f75993fc6
[1] 2025-05-23 17:56:54.771 ServerApp[ ] Skipped non-installed server(s): bash-language-server, dockerfile-language-server-nodejs, javascript-typescript-langserver, jedi-language-server, julia-language-server, pyright, python-language-server, python-lsp-server, r-languageserver, sql-language-server, texlab, typescript-language-server, unified-language-server, vscode-css-languageserver-bin, vscode-html-languageserver-bin, vscode-json-languageserver-bin, yaml-language-server
[1] 2025-05-23 17:57:10.499 ServerApp[ ] 302 GET / (@127.0.0.1) 1.34ms
[1] 2025-05-23 17:57:10.718 JupyterNotebookApp[ ] 302 GET /tree (@127.0.0.1) 1.01ms
[1] 2025-05-23 17:56:54.578 ServerApp[ ]

```

AWS profile default

Name	Modified	File Size
venv	1 minute ago	
FHV_Big_Data_Viz.ipynb	1 second ago	337 B
README.md	4 days ago	569 B

9. Python Codes for Visualization:

Hourly Trips:

```
[3]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the CSV (adjust filename if needed)
df = pd.read_csv('fc1ddca3-f9e4-4264-bf28-161b8fa90541.csv') # or your actual file name

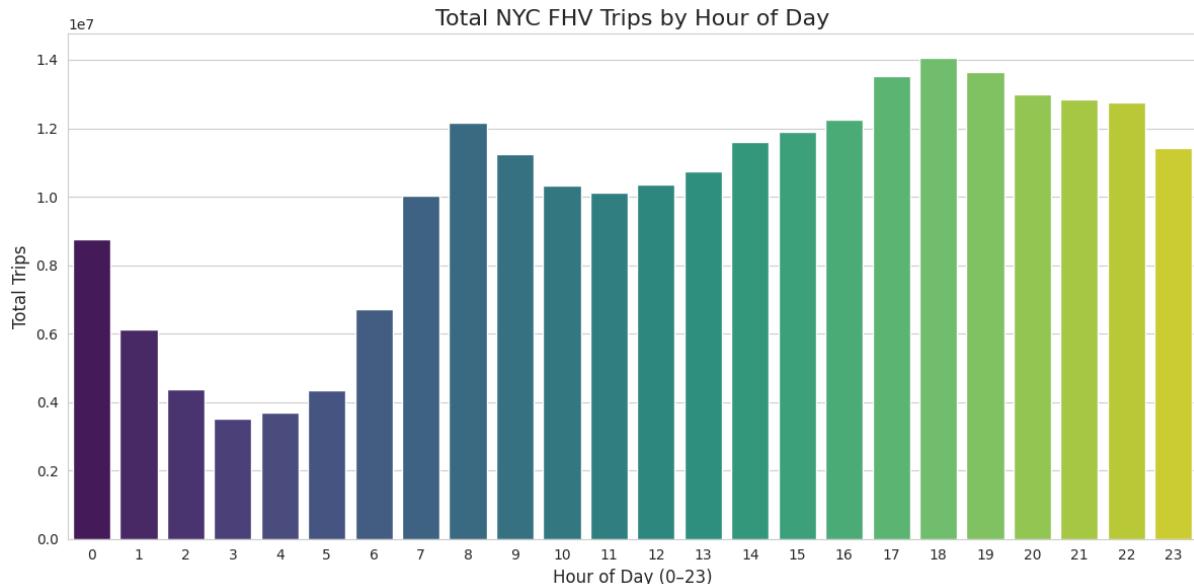
# Optional: ensure 'trip_hour' is treated as categorical (for correct ordering in plots)
df['trip_hour'] = df['trip_hour'].astype(int)

# Set style
sns.set_style("whitegrid")

# Create bar plot
plt.figure(figsize=(12,6))
sns.barplot(x='trip_hour', y='total_trips', data=df, palette='viridis')

# Add titles and labels
plt.title('Total NYC FHV Trips by Hour of Day', fontsize=16)
plt.xlabel('Hour of Day (0-23)', fontsize=12)
plt.ylabel('Total Trips', fontsize=12)
plt.xticks(rotation=0)
plt.tight_layout()

# Show the plot
plt.show()
```



Daily Trips:

```
[6]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the CSV
df = pd.read_csv('5ab0fcba-6432-4e95-a7b2-2567799eb7f0.csv', parse_dates=['trip_date']) # Adjust filename if needed

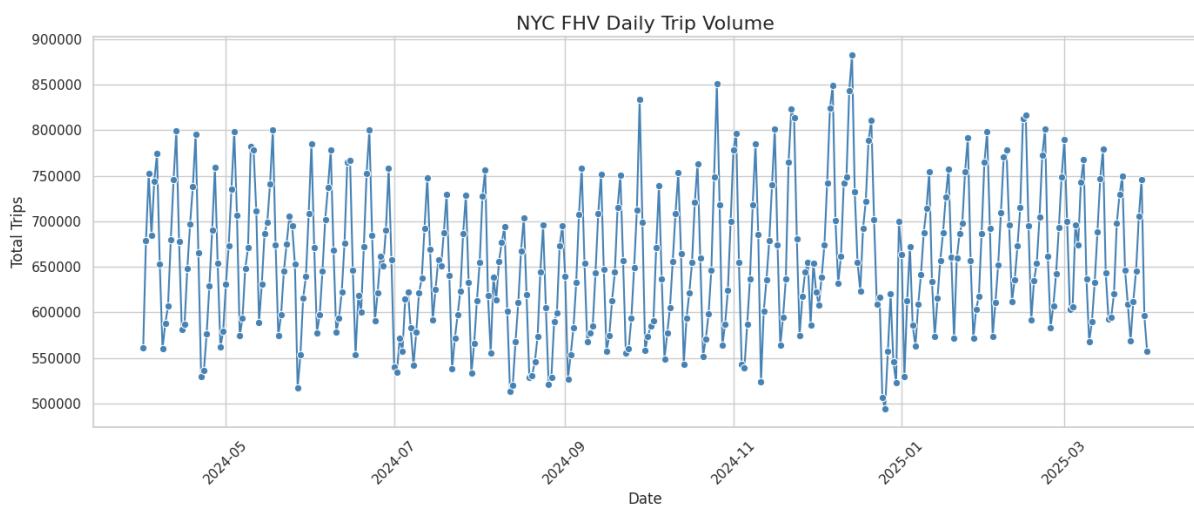
# Preview data
print(df.head())

# Set Seaborn style
sns.set(style='whitegrid')

# Create Line plot
plt.figure(figsize=(14,6))
sns.lineplot(x='trip_date', y='total_trips', data=df, marker='o', color='steelblue')

# Add titles and Labels
plt.title('NYC FHV Daily Trip Volume', fontsize=16)
plt.xlabel('Date', fontsize=12)
plt.ylabel('Total Trips', fontsize=12)
plt.xticks(rotation=45)
plt.tight_layout()

# Show the plot
plt.show()
```



Weekly Trips:

```
[8]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the CSV
df = pd.read_csv('1fef1b7a-812e-4013-b0c3-07cb09b63dfc.csv') # Adjust the filename

# Combine year and week into a single string for x-axis
df['year_week'] = df['trip_year'].astype(str) + '-W' + df['trip_week'].astype(str)

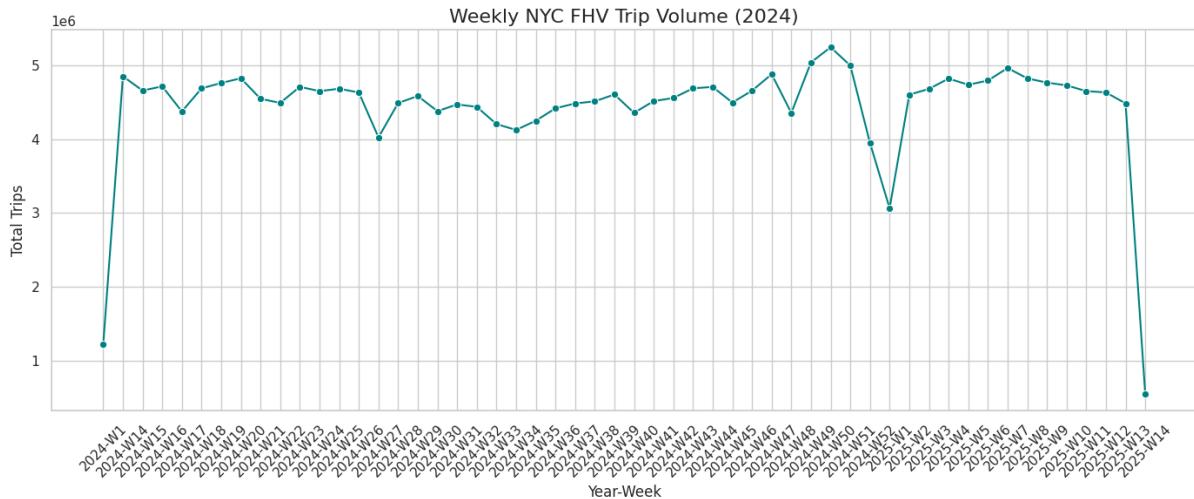
# Optional: convert to datetime (week start date), for time-based x-axis (advanced)
# df['week_start'] = pd.to_datetime(df['trip_year'].astype(str) + df['trip_week'].astype(str) + '1', format='%G%V%u')

# Set Seaborn style
sns.set(style='whitegrid')

# Plot
plt.figure(figsize=(14,6))
sns.lineplot(x='year_week', y='total_trips', data=df, marker='o', color='teal')

# Add labels
plt.title('Weekly NYC FHV Trip Volume (2024)', fontsize=16)
plt.xlabel('Year-Week', fontsize=12)
plt.ylabel('Total Trips', fontsize=12)
plt.xticks(rotation=45)
plt.tight_layout()

# Show plot
plt.show()
```



Monthly Trips:

```
[11]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the CSV
df = pd.read_csv('25cd1537-1e20-481f-b166-fcf1017bc014.csv') # Adjust filename if needed

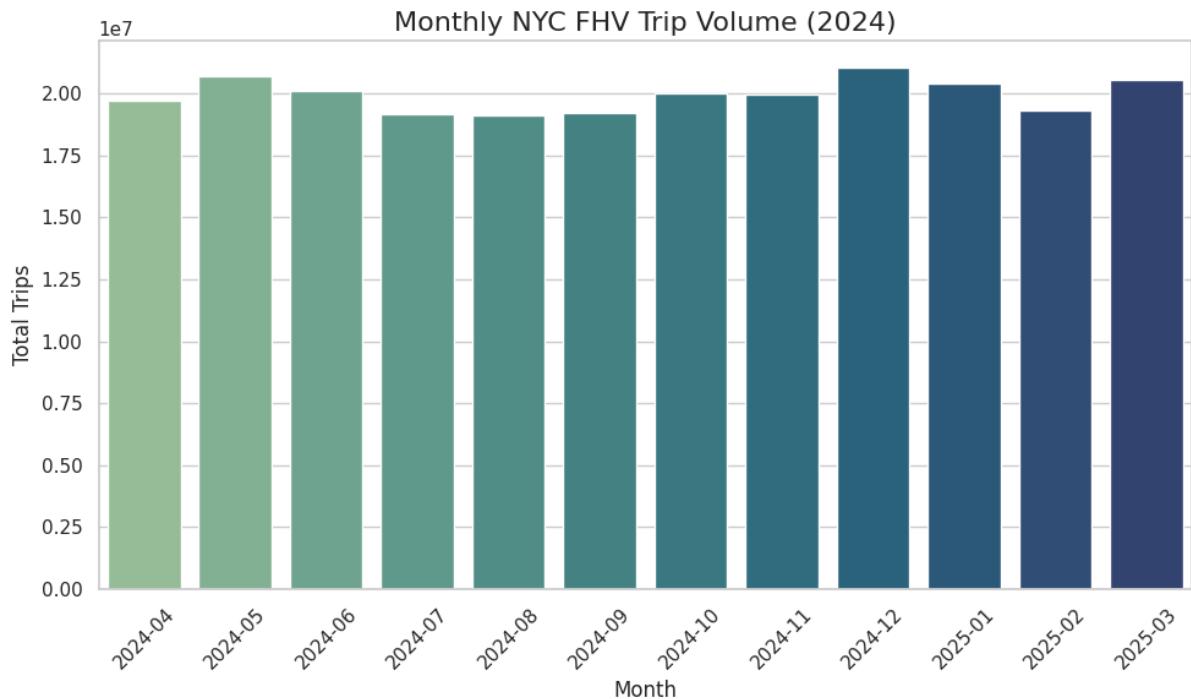
# Create a column for "YYYY-MM" labels
df['year_month'] = df['trip_year'].astype(str) + '-' + df['trip_month'].astype(str).str.zfill(2)

# Set Seaborn style
sns.set(style='whitegrid')

# Plot bar chart
plt.figure(figsize=(10,6))
sns.barplot(x='year_month', y='total_trips', data=df, palette='crest')

# Add titles and labels
plt.title('Monthly NYC FHV Trip Volume (2024)', fontsize=16)
plt.xlabel('Month', fontsize=12)
plt.ylabel('Total Trips', fontsize=12)
plt.xticks(rotation=45)
plt.tight_layout()

# Show plot
plt.show()
```



Peak Hour of the Day:

```
[31]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load data
df = pd.read_csv('fc1dcda3-f9e4-4264-bf28-161b8fa90541.csv') # Adjust filename if needed

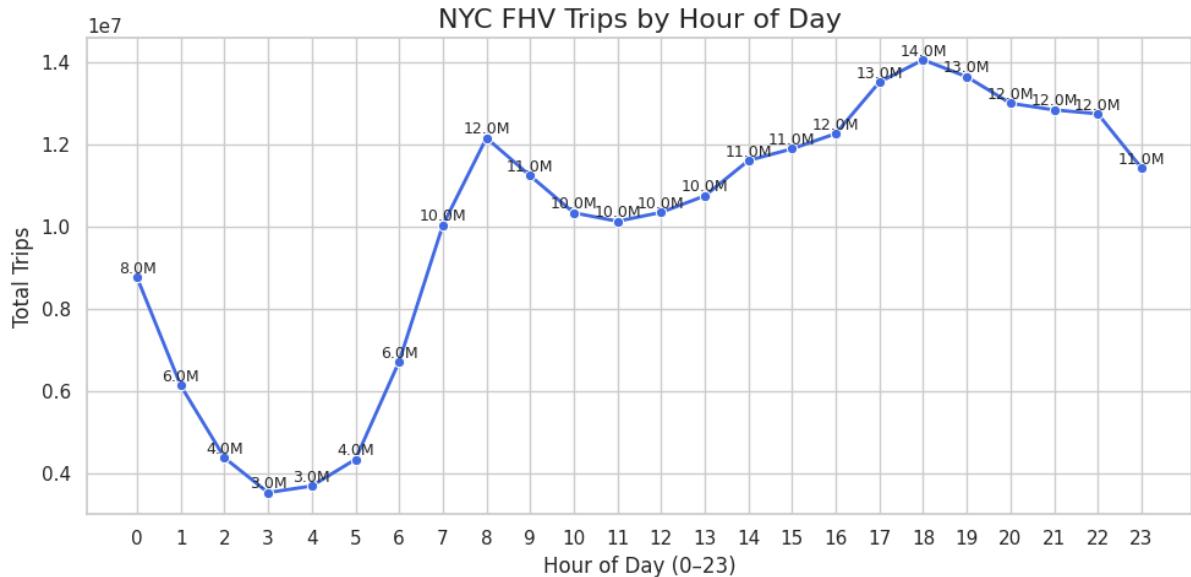
# Set style
sns.set(style='whitegrid')

# Plot Line chart
plt.figure(figsize=(10,5))
sns.lineplot(x='trip_hour', y='total_trips', data=df, marker='o', linewidth=2, color='royalblue')

# Annotate each point with total trips (optional)
for i, row in df.iterrows():
    plt.text(row['trip_hour'], row['total_trips'] + 100000, f'{row["total_trips"] // 1_000_000:.1f}M',
             ha='center', fontsize=9)

# Set axis labels and title
plt.title('NYC FHV Trips by Hour of Day', fontsize=16)
plt.xlabel('Hour of Day (0-23)', fontsize=12)
plt.ylabel('Total Trips', fontsize=12)
plt.xticks(range(0, 24))
plt.tight_layout()

# Show plot
plt.show()
```



Peak Day of the Week:

```
[33]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load CSV
df = pd.read_csv('5066ad66-6c9a-4a9f-ac51-c88b54d39a29.csv') # Adjust filename

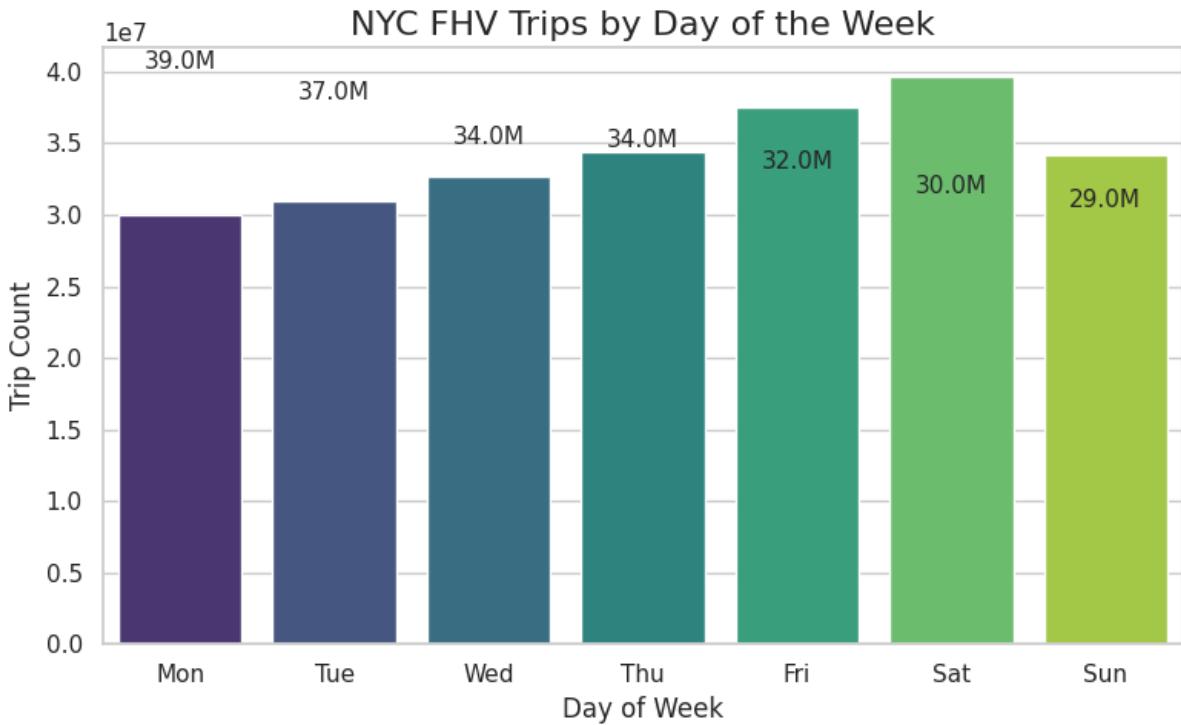
# Map day numbers to names
day_map = {
    1: 'Mon', 2: 'Tue', 3: 'Wed', 4: 'Thu',
    5: 'Fri', 6: 'Sat', 7: 'Sun'
}
df['day_name'] = df['day_of_week'].map(day_map)

# Sort days in correct week order
day_order = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
df['day_name'] = pd.Categorical(df['day_name'], categories=day_order, ordered=True)
df = df.sort_values('day_name')

# Plot bar chart
plt.figure(figsize=(8,5))
sns.barplot(x='day_name', y='trip_count', data=df, palette='viridis')

# Add value labels
for i, row in df.iterrows():
    plt.text(i, row['trip_count'] + 5e5, f'{row["trip_count"] // 1_000_000:.1f}M',
             ha='center', fontsize=11)

# Labels and title
plt.title('NYC FHV Trips by Day of the Week', fontsize=16)
plt.xlabel('Day of Week', fontsize=12)
plt.ylabel('Trip Count', fontsize=12)
plt.tight_layout()
plt.show()
```



Peak Week:

```
[35]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load CSV
df = pd.read_csv('b02b3d90-6d48-4106-899d-68f1a9195aa9.csv') # Adjust filename

# Create a datetime column from year and week number (ISO week)
# We'll use pandas to convert year-week to datetime (Monday of the week)
df['week_start_date'] = pd.to_datetime(df['trip_year'].astype(str) + df['trip_week'].astype(str) + '1', format='%G%V%u')

# Sort by date
df = df.sort_values('week_start_date')

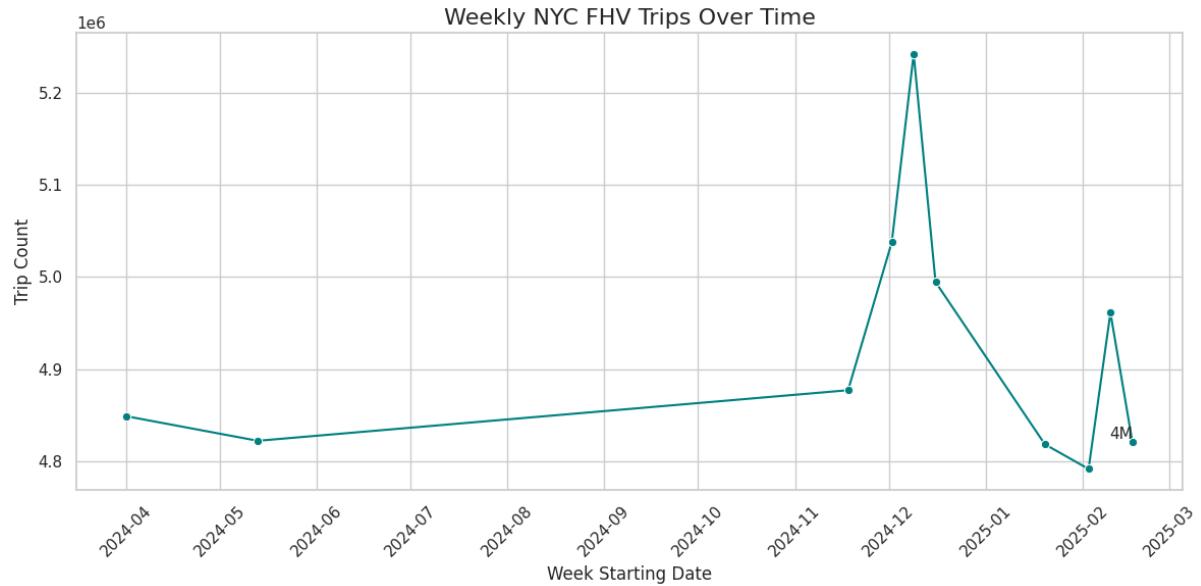
# Set plot style
sns.set(style='whitegrid')

# Plot Line chart
plt.figure(figsize=(12,6))
sns.lineplot(x='week_start_date', y='trip_count', data=df, marker='o', color='teal')

# Format x-axis for better date display
plt.xticks(rotation=45)
plt.xlabel('Week Starting Date', fontsize=12)
plt.ylabel('Trip Count', fontsize=12)
plt.title('Weekly NYC FHV Trips Over Time', fontsize=16)

# Optional: annotate last point
last_row = df.iloc[-1]
plt.text(last_row['week_start_date'], last_row['trip_count'], f"{last_row['trip_count']} // 1_000_000M",
        fontsize=11, ha='right', va='bottom')

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



Monthly Revenue Trends:

```
[13]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.ticker as ticker

# Load CSV
df = pd.read_csv('a6d8a6a1-cf83-4ce4-80e2-0ce227b65b63.csv') # Adjust filename

# Create a 'year_month' column like '2024-04'
df['year_month'] = df['trip_year'].astype(str) + '-' + df['trip_month'].astype(str).zfill(2)

# Set style
sns.set(style='whitegrid')

# Plot bar chart
plt.figure(figsize=(10,6))
ax = sns.barplot(x='year_month', y='total_revenue', data=df, palette='mako')

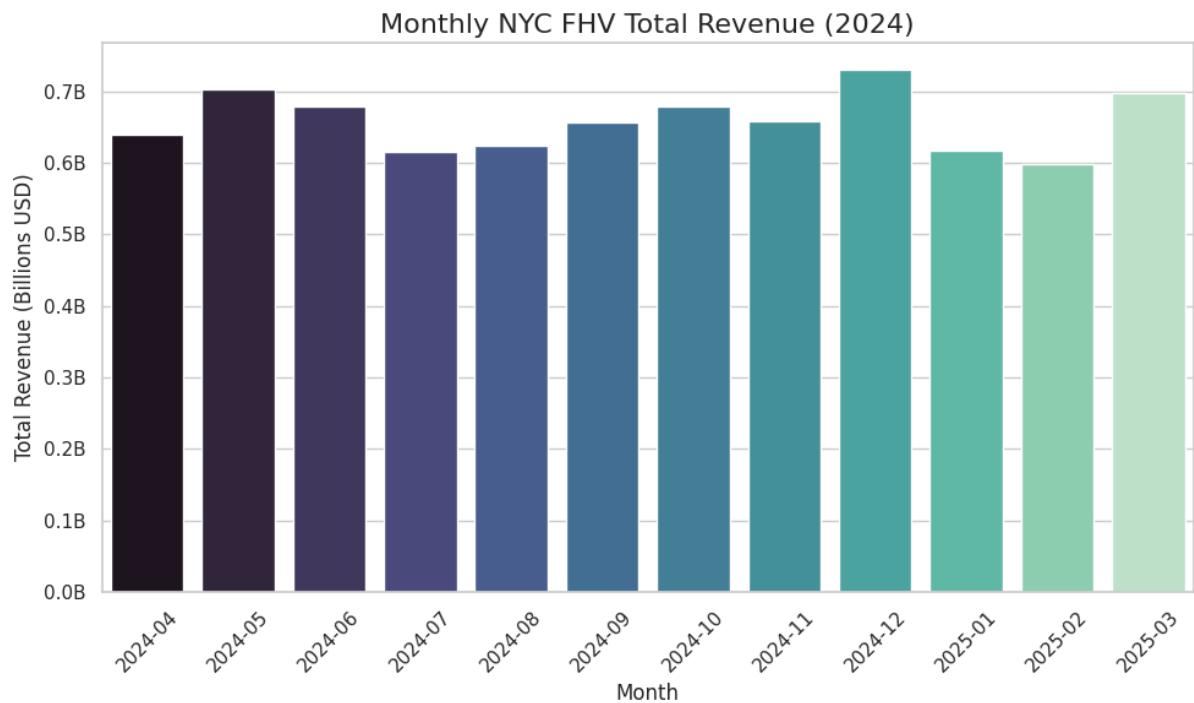
# Format y-axis labels as billions or millions
def billions(x, pos):
    return f'{x*1e-9:.1f}B' # Convert to billions

ax.yaxis.set_major_formatter(ticker.FuncFormatter(billions))

# Titles and labels
plt.title('Monthly NYC FHV Total Revenue (2024)', fontsize=16)
plt.xlabel('Month', fontsize=12)
plt.ylabel('Total Revenue (Billions USD)', fontsize=12)
plt.xticks(rotation=45)
plt.tight_layout()

plt.show()
```

2



Daily Average Fare:

```
[15]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

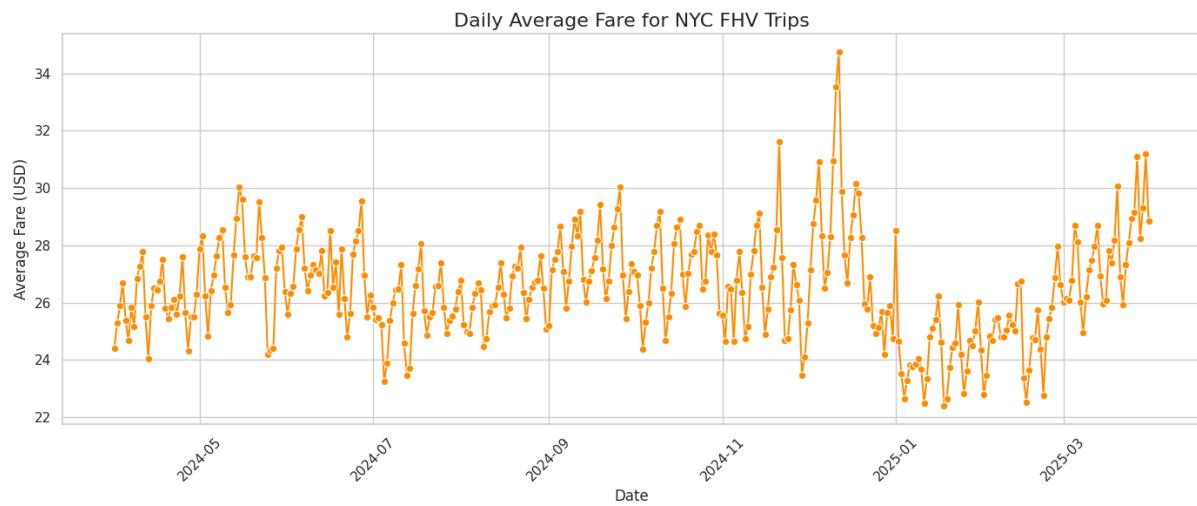
# Load CSV
df = pd.read_csv('f6f33939-b97e-4123-bef1-4965e7884b13.csv', parse_dates=['trip_date']) # Adjust filename

# Set style
sns.set(style='whitegrid')

# Plot Line chart
plt.figure(figsize=(14,6))
sns.lineplot(x='trip_date', y='avg_fare', data=df, marker='o', color='darkorange')

# Titles and Labels
plt.title('Daily Average Fare for NYC FHV Trips', fontsize=16)
plt.xlabel('Date', fontsize=12)
plt.ylabel('Average Fare (USD)', fontsize=12)
plt.xticks(rotation=45)
plt.tight_layout()

plt.show()
```



Top Pickup Locations:

```
[19]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

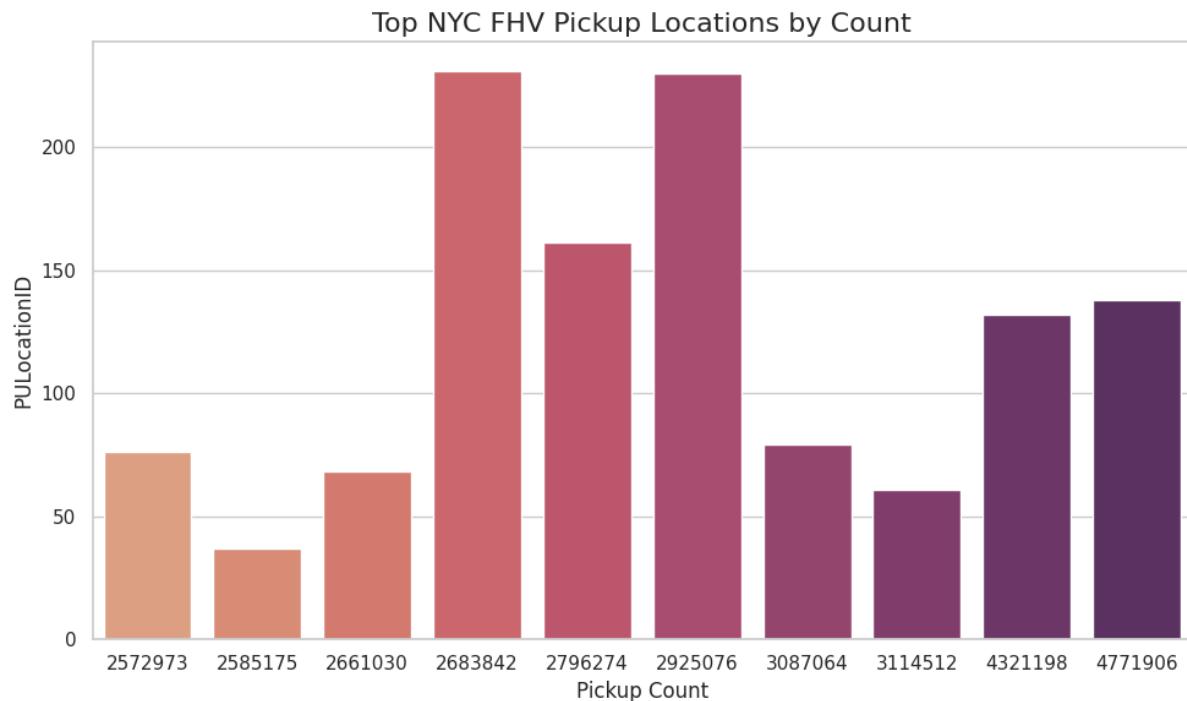
# Load the CSV
df = pd.read_csv('ab8d6f10-b864-4d56-b548-e5997203da9f.csv') # Adjust filename

# Optional: sort by pickup_count for cleaner bar chart
df = df.sort_values(by='pickup_count', ascending=True)

# Create Seaborn barplot (horizontal)
plt.figure(figsize=(10,6))
sns.barplot(x='pickup_count', y='PULocationID', data=df, palette='flare')

# Add Labels and title
plt.title('Top NYC FHV Pickup Locations by Count', fontsize=16)
plt.xlabel('Pickup Count', fontsize=12)
plt.ylabel('PULocationID', fontsize=12)
plt.tight_layout()

plt.show()
```



Top Dropoff Locations:

```
[21]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

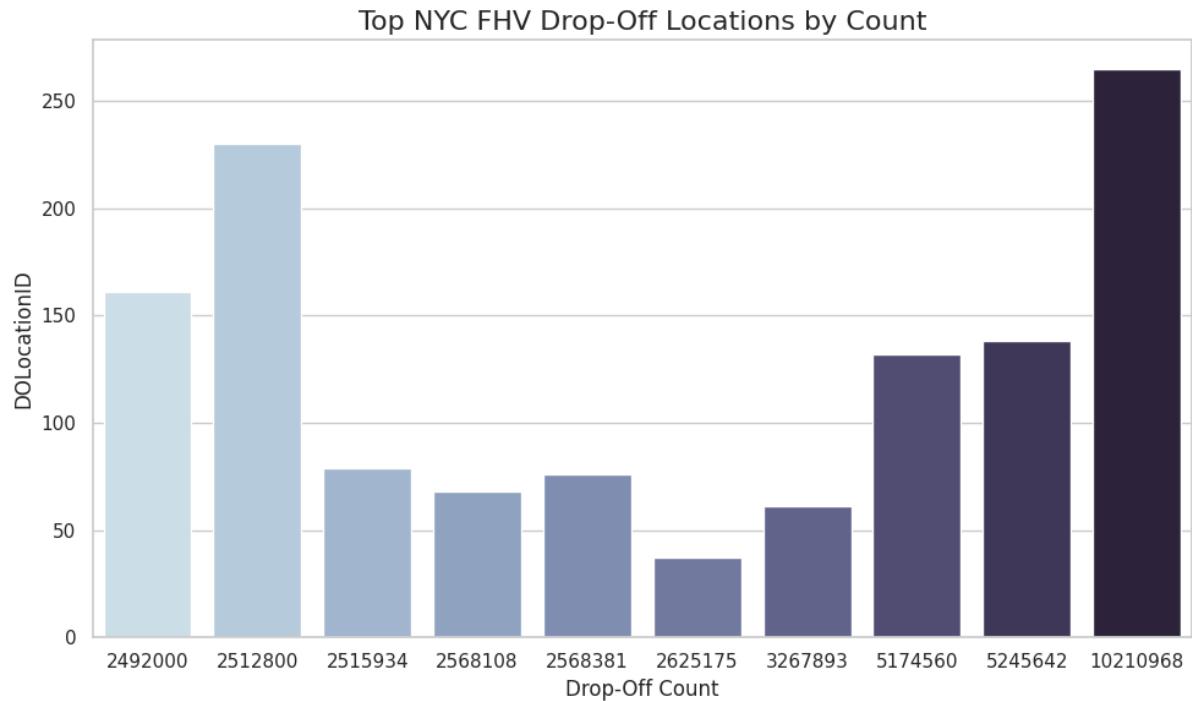
# Load the CSV
df = pd.read_csv('e85f542b-5304-4be8-a861-8ec06c7ed9ee.csv') # Adjust filename

# Sort by dropoff_count for clean visualization
df = df.sort_values(by='dropoff_count', ascending=True)

# Plot horizontal bar chart
plt.figure(figsize=(10,6))
sns.barplot(x='dropoff_count', y='DOLocationID', data=df, palette='ch:s=.25,rot=-.25')

# Add titles and axis labels
plt.title('Top NYC FHV Drop-Off Locations by Count', fontsize=16)
plt.xlabel('Drop-Off Count', fontsize=12)
plt.ylabel('DOLocationID', fontsize=12)
plt.tight_layout()

# Show the plot
plt.show()
```



Most Popular Routes:

```
[17]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load CSV
df = pd.read_csv('66e4a58d-88a3-4bb5-9dde-67fd124c5508.csv') # Adjust filename

# Create a 'route' label like "PULoc-DOLoc"
df['route'] = df['PULocationID'].astype(str) + ' → ' + df['DOLocationID'].astype(str)

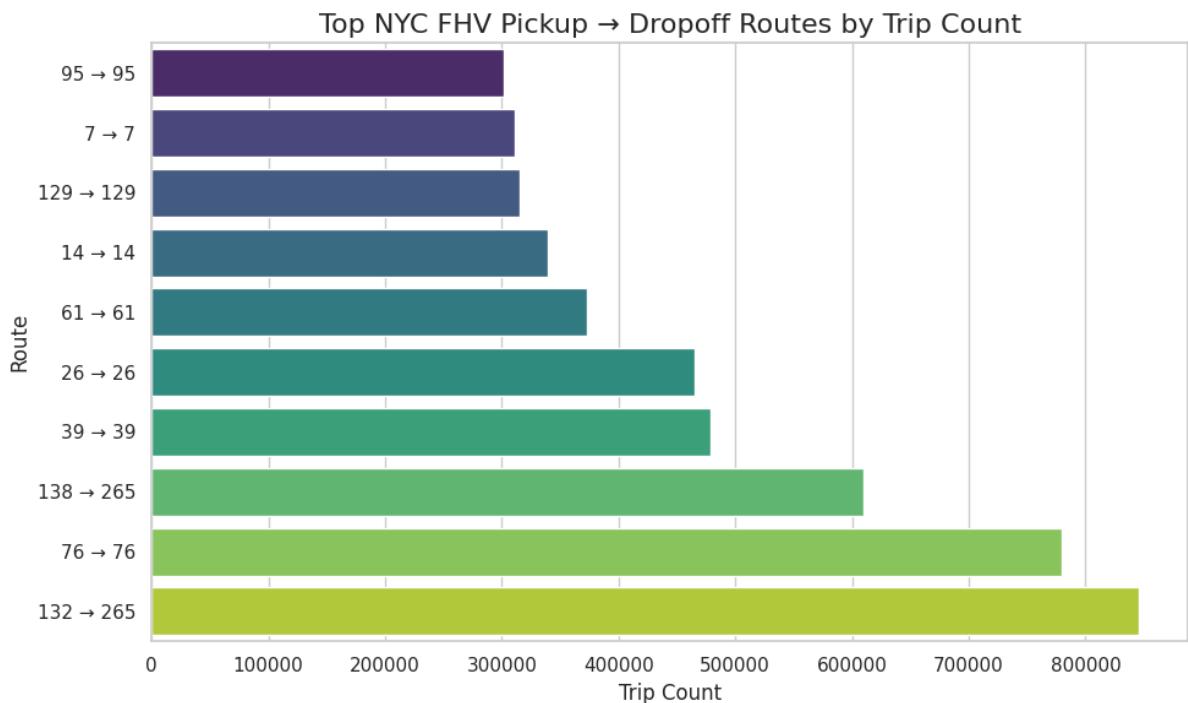
# Sort routes by trip_count (optional if not pre-sorted)
df = df.sort_values(by='trip_count', ascending=True)

# Set style
sns.set(style='whitegrid')

# Plot horizontal bar chart
plt.figure(figsize=(10,6))
sns.barplot(x='trip_count', y='route', data=df, palette='viridis')

# Add titles and labels
plt.title('Top NYC FHV Pickup → Dropoff Routes by Trip Count', fontsize=16)
plt.xlabel('Trip Count', fontsize=12)
plt.ylabel('Route', fontsize=12)
plt.tight_layout()

plt.show()
```



Trips by Company:

```
[23]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the CSV
df = pd.read_csv('9e705736-d158-454e-8f0e-ab746e517b79.csv') # Adjust filename

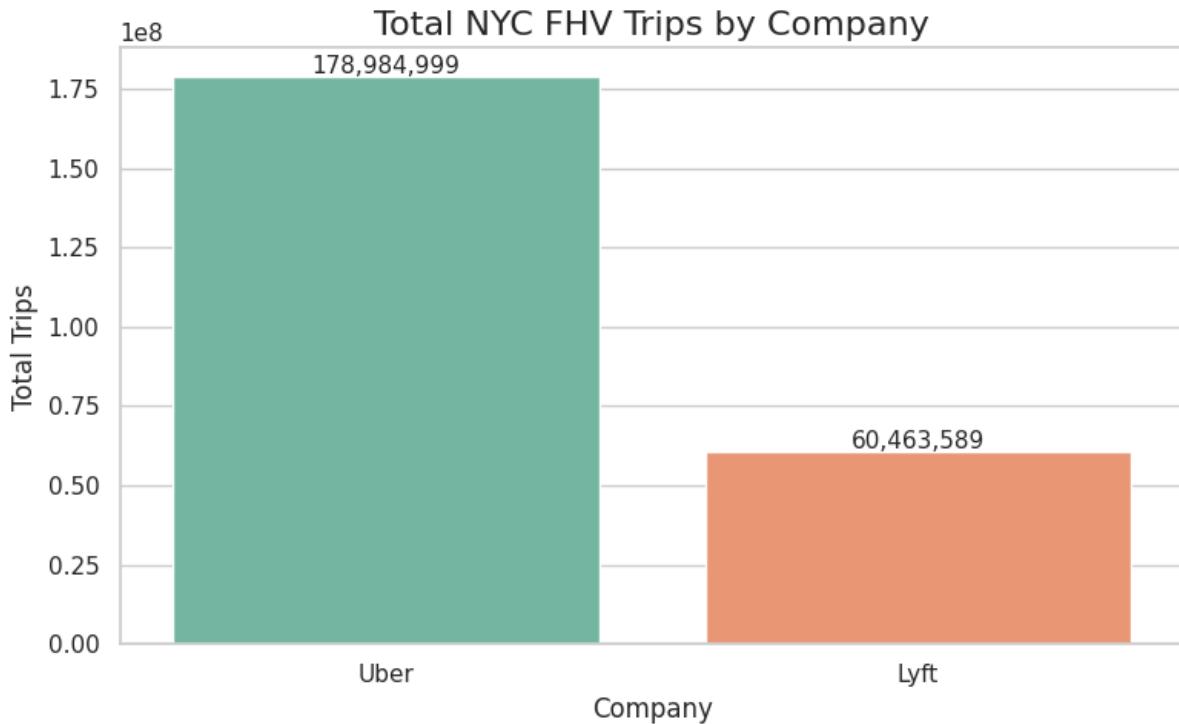
# Set style
sns.set(style='whitegrid')

# Plot bar chart
plt.figure(figsize=(8,5))
sns.barplot(x='company', y='total_trips', data=df, palette='Set2')

# Add Labels and title
plt.title('Total NYC FHV Trips by Company', fontsize=16)
plt.xlabel('Company', fontsize=12)
plt.ylabel('Total Trips', fontsize=12)

# Optional: Add value labels
for i, row in df.iterrows():
    plt.text(i, row['total_trips'] + 1e6, f'{row["total_trips"]:,}', ha='center', fontsize=11)

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



Revenue by Company:

```
[25]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.ticker as ticker

# Load the CSV
df = pd.read_csv('6c284c80-6404-4f3f-83fe-54a2707e8671.csv') # Adjust filename if needed

# Set plot style
sns.set(style='whitegrid')

# Create bar plot
plt.figure(figsize=(8,5))
ax = sns.barplot(x='company', y='total_revenue', data=df, palette='pastel')

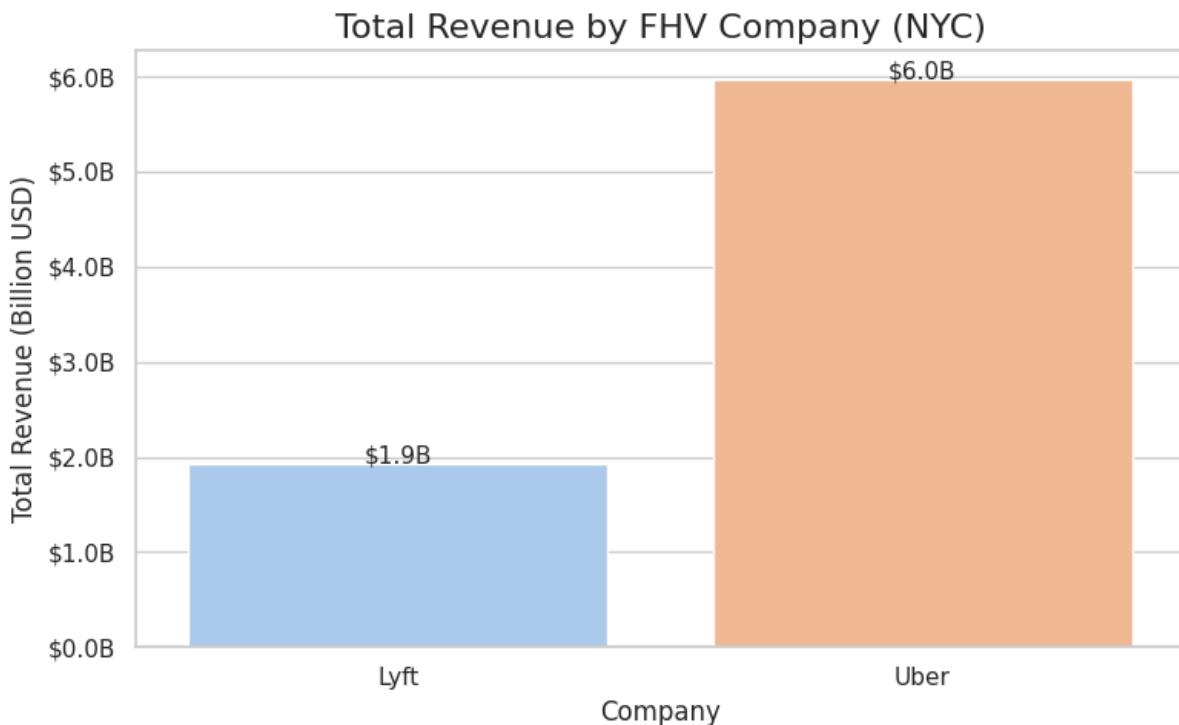
# Format y-axis to show billions
def format_billions(x, pos):
    return f'{x * 1e-9:.1f}B'

ax.yaxis.set_major_formatter(ticker.FuncFormatter(format_billions))

# Add value labels on bars
for i, row in df.iterrows():
    plt.text(i, row['total_revenue'] + 1e7, f"${row['total_revenue'] * 1e-9:.1f}B",
             ha='center', fontsize=11)

# Titles and labels
plt.title('Total Revenue by FHV Company (NYC)', fontsize=16)
plt.xlabel('Company', fontsize=12)
plt.ylabel('Total Revenue (Billion USD)', fontsize=12)
plt.tight_layout()

# Show the plot
plt.show()
```



Shared vs Solo Rides:

```
[27]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

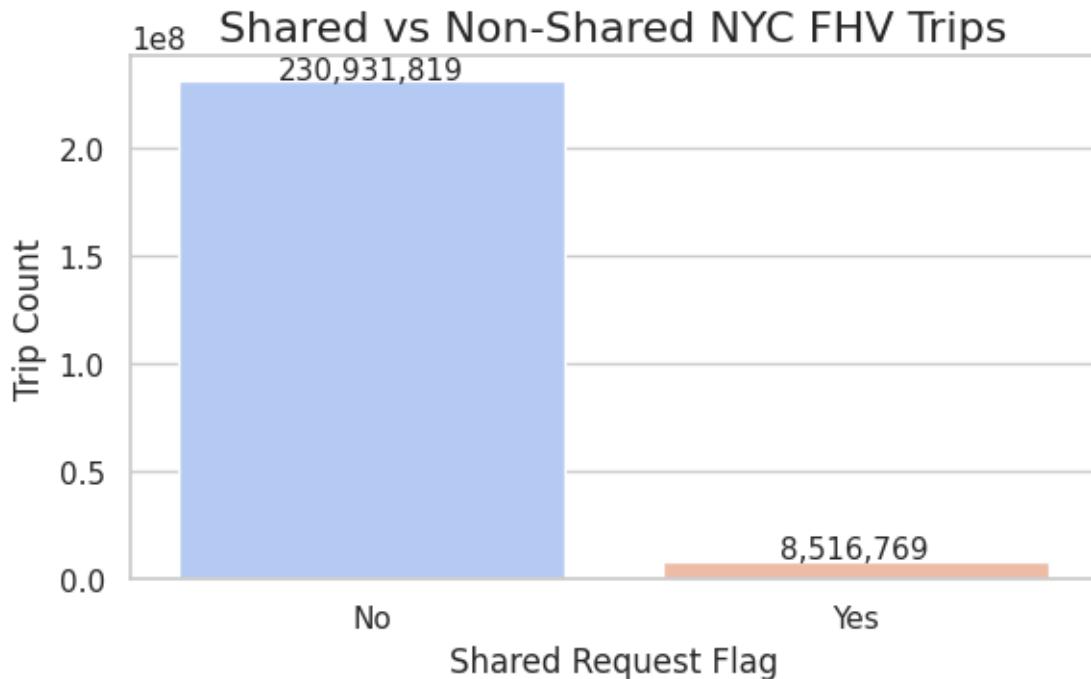
# Load CSV
df = pd.read_csv('59a56157-2b98-429a-b014-85b62268bade.csv') # Adjust filename

# Set style
sns.set(style='whitegrid')

# Plot bar chart
plt.figure(figsize=(6,4))
sns.barplot(x='shared_request_flag', y='trip_count', data=df, palette='coolwarm')

# Add value labels
for i, row in df.iterrows():
    plt.text(i, row['trip_count'] + 1e6, f'{row["trip_count"]:,}', ha='center', fontsize=11)

# Labels and title
plt.title('Shared vs Non-Shared NYC FHV Trips', fontsize=16)
plt.xlabel('Shared Request Flag', fontsize=12)
plt.ylabel('Trip Count', fontsize=12)
plt.xticks([0, 1], ['No', 'Yes'])
plt.tight_layout()
plt.show()
```



WAV (Wheelchair Accessible Vehicle) Requests:

```
[29]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

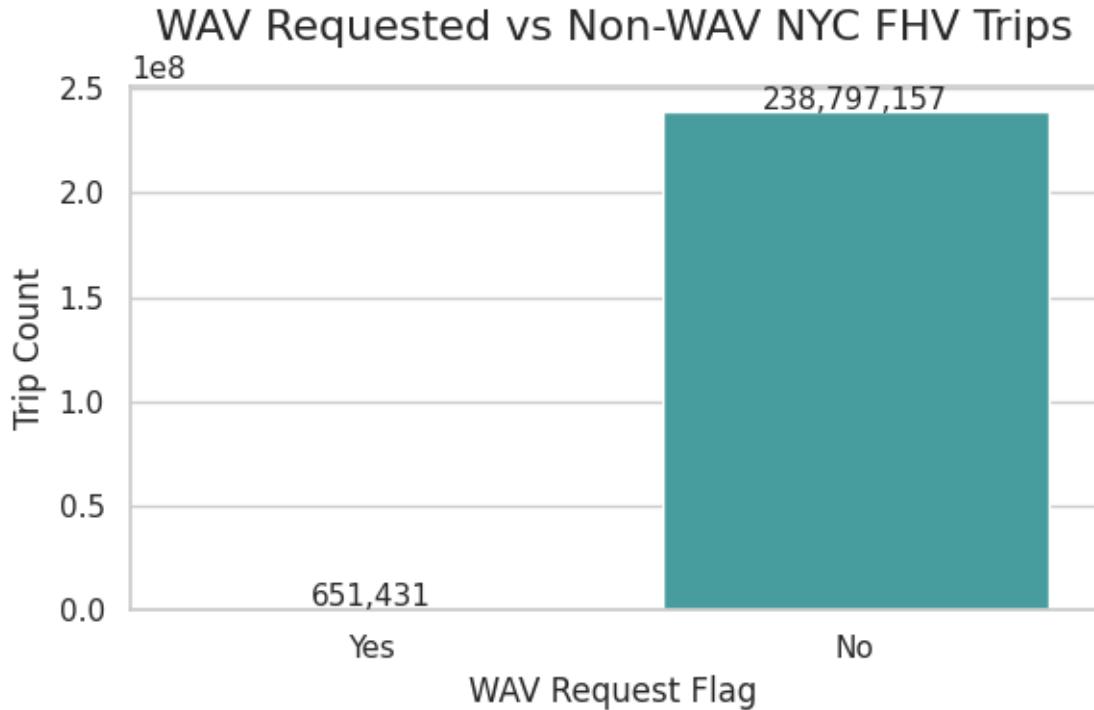
# Load the data
df = pd.read_csv('2365ddca-87f0-4b0b-a89a-b2636390a05c.csv') # Adjust filename if needed

# Set style
sns.set(style='whitegrid')

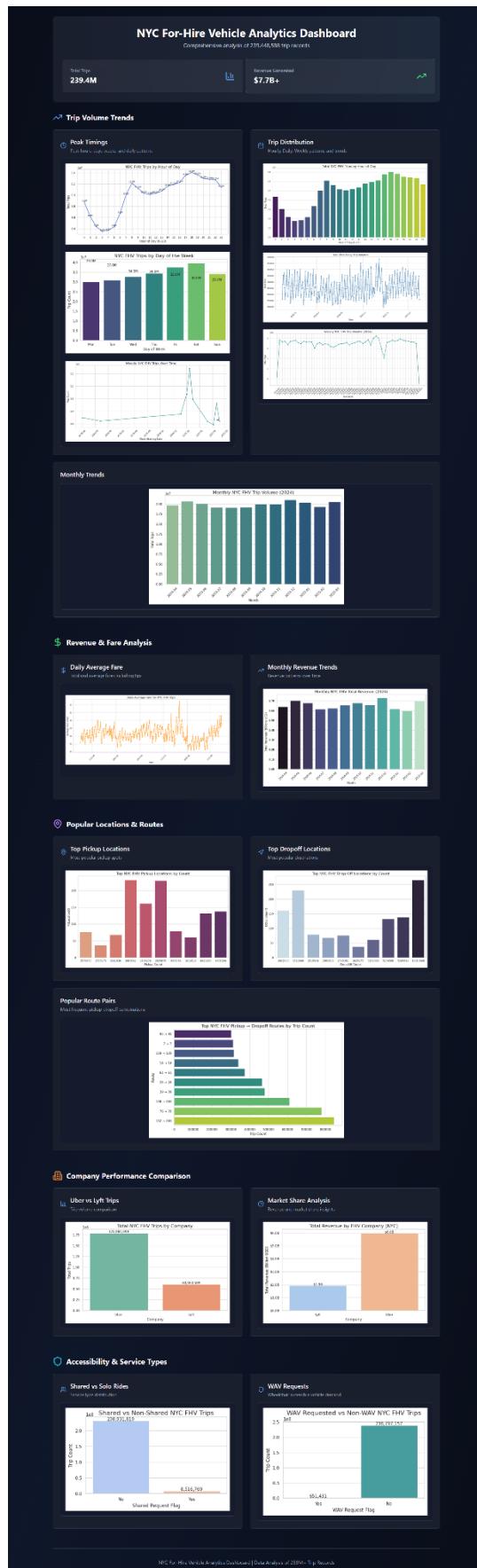
# Plot bar chart
plt.figure(figsize=(6,4))
sns.barplot(x='wav_request_flag', y='trip_count', data=df, palette='mako')

# Add Labels
for i, row in df.iterrows():
    plt.text(i, row['trip_count'] + 1e6, f'{row["trip_count"]:,}', ha='center', fontsize=11)

# Titles and axis labels
plt.title('WAV Requested vs Non-WAV NYC FHV Trips', fontsize=16)
plt.xlabel('WAV Request Flag', fontsize=12)
plt.ylabel('Trip Count', fontsize=12)
plt.xticks([0, 1], ['Yes', 'No']) # Assuming index order Y, N
plt.tight_layout()
plt.show()
```



Dashboard:



Key Insights Uncovered:

Trip Volume Trends

- **Hourly Trend:** Sharp increase in trips between **4–8 AM** and again from **12–7 PM**, peaking around **6 PM** — classic **commute hour spikes**.
- **Daily Trend:** **Fridays and Saturdays** show the **highest trip volume**, especially Friday evenings and late-night Saturday.
- **Weekly/Monthly Trend:** Trip counts are **highest in December**, with a visible dip in **January and February** — possibly due to winter weather.

Revenue & Fare Analysis

- **Total Fare Trends:** Monthly total revenue is **highest in December & summer months (May & June)**. Fare amounts dip during early months of the year.
- **Average Fare per Trip:** Fairly stable across months, but **slightly higher during weekends**, suggesting longer or higher-tipped trips.
- **Tips Contribution:** Tips increase proportionally during **weekend nights**, indicating more leisure-based or premium service usage.

Popular Locations & Routes

- **Top Pickup Locations:** Midtown Manhattan, JFK Airport, and Upper East Side dominate.
- **Top Dropoff Locations:** Similar hotspots — especially **JFK** and **downtown locations**.
- **Most Frequent Routes:** Repeated travel patterns between **Penn Station ↔ Times Square**, and **JFK ↔ Manhattan** corridors.

Company Performance Comparison

- **Trip Count:** Uber completes **more trips** than Lyft consistently across all time periods.
- **Revenue Share:** Uber also leads in **revenue generation**, although Lyft shows higher **average fare per trip**, suggesting longer or premium trips.
- **Market Share:** Uber holds a **clear majority share**, especially in central boroughs; Lyft shows more presence in **outer boroughs** and for WAVs.

Accessibility & Service Types

- **Solo vs Shared Rides:** Solo rides dominate (~80%+), though **shared rides spike during rush hours.**
- **WAV Requests:** Wheelchair-accessible vehicle demand is **low in volume but steady**, with minor increases during daytime and near hospitals.

Driver Compensation

- **Total Pay:** Closely follows total fare trend — higher in weekends.
- **Average Driver Pay per Trip:** Fairly stable, with **minor increases during peak hours and long-distance routes.**
- **Pay as % of Revenue:** On average, **drivers receive ~70–75% of trip revenue**, indicating consistent payout policies.

Temporal Seasonality

- **Weekly Patterns:** **Weekends (Fri-Sun)** show increased trip and revenue counts, especially at night.
- **Monthly Seasonality:** Peak during **summer**, low in **winter**. November and December show bumps — likely due to **holiday travel**.
- **Event Impact:** Unusual peaks around July 4th and New Year's Eve, pointing to **event-driven surges**.

Predictive Analysis

AWS aspects explored:

1. AWS SageMaker AI: Created a notebook instance & uploaded the dataset as parquet file to train the ML/DL models.

Steps Followed:

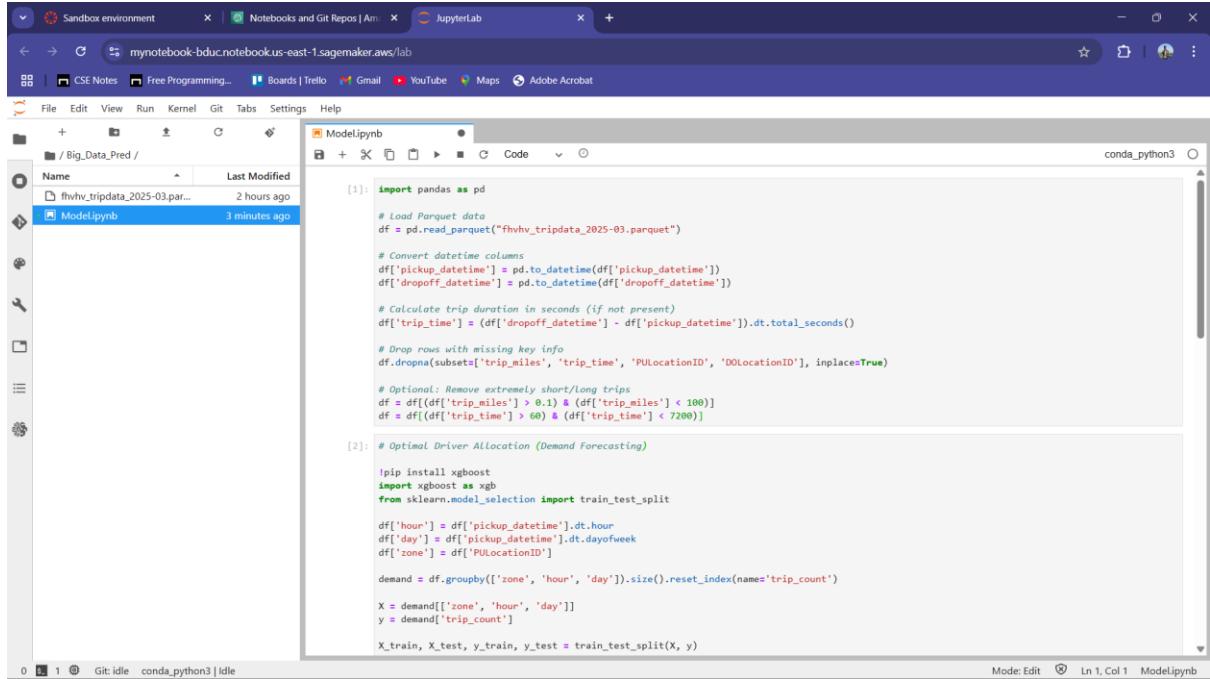
1. Notebook Instance Creation:

Notebook instance name: MyNotebook

Instance Type: ml.m4.large

The screenshot shows the AWS Lambda service dashboard. On the left, there's a sidebar with navigation links like 'Getting started', 'What's new', 'Applications and IDEs' (which includes 'Studio', 'Canvas', 'RStudio', 'TensorBoard', 'Profiler', and 'Notebooks'), and 'Admin configurations' (with 'Domains', 'Role manager', 'Images', and 'Lifecycle configurations'). The main content area has a heading 'Notebooks and Git repos'. Below it, there's a promotional banner for 'JupyterLab in SageMaker Studio' with a 'Get Started' button. Underneath the banner, there are tabs for 'Notebook instances' and 'Git repositories', with 'Notebook instances' being the active tab. A table lists the created notebook instance: Name: MyNotebook, Instance: ml.m4.large, Creation time: 5/24/2025, 12:11:20 PM, Status: InService, Actions: Open Jupyter | Open JupyterLab. At the bottom, there are links for 'CloudShell', 'Feedback', and copyright information: © 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences.

2. Uploading dataset & creating Python notebook for model training:



The screenshot shows the JupyterLab interface with a sidebar on the left containing a file tree with a folder 'Big_Data_Pred/' and a file 'Model.ipynb'. The main area displays the code for 'Model.ipynb' in a code editor. The code performs several steps: loading a parquet file, converting datetime columns to pandas datetime objects, calculating trip duration in seconds, dropping rows with missing key info, removing extremely short or long trips, and finally performing optimal driver allocation using XGBoost. The code editor has two cells: Cell [1] and Cell [2]. Cell [1] contains the initial data processing and filtering logic. Cell [2] contains the XGBoost setup and prediction code.

```
[1]: import pandas as pd

# Load Parquet data
df = pd.read_parquet("fhvhv_tripdata_2025-03.parquet")

# Convert datetime columns
df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'])
df['dropoff_datetime'] = pd.to_datetime(df['dropoff_datetime'])

# Calculate trip duration in seconds (if not present)
df['trip_time'] = (df['dropoff_datetime'] - df['pickup_datetime']).dt.total_seconds()

# Drop rows with missing key info
df.dropna(subset=['trip_miles', 'trip_time', 'PULocationID', 'DOLocationID'], inplace=True)

# Optional: Remove extremely short/long trips
df = df[(df['trip_miles'] > 0.1) & (df['trip_miles'] < 100)]
df = df[(df['trip_time'] > 60) & (df['trip_time'] < 7200)]

[2]: # Optimal Driver Allocation (Demand Forecasting)

!pip install xgboost
import xgboost as xgb
from sklearn.model_selection import train_test_split

df['hour'] = df['pickup_datetime'].dt.hour
df['day'] = df['pickup_datetime'].dt.dayofweek
df['zone'] = df['PULocationID']

demand = df.groupby(['zone', 'hour', 'day']).size().reset_index(name='trip_count')

X = demand[['zone', 'hour', 'day']]
y = demand['trip_count']

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

3. Python Codes for Predictions:

(i) Optimal Driver Allocation (Demand Forecasting)



The screenshot shows a Jupyter Notebook cell with the following Python code for optimal driver allocation using XGBoost. The code includes pip installation of xgboost, feature engineering (hour, day, zone), demand forecasting using groupby, and splitting the data into training and testing sets. It then creates an XGBRegressor, fits it to the training data, and prints the prediction for a specific example (zone 132, 5PM, Wednesday). A warning message at the bottom indicates compatibility issues with glibc versions.

```
[2]: # Optimal Driver Allocation (Demand Forecasting)

!pip install xgboost
import xgboost as xgb
from sklearn.model_selection import train_test_split

df['hour'] = df['pickup_datetime'].dt.hour
df['day'] = df['pickup_datetime'].dt.dayofweek
df['zone'] = df['PULocationID']

demand = df.groupby(['zone', 'hour', 'day']).size().reset_index(name='trip_count')

X = demand[['zone', 'hour', 'day']]
y = demand['trip_count']

X_train, X_test, y_train, y_test = train_test_split(X, y)

model = xgb.XGBRegressor()
model.fit(X_train, y_train)

print(model.predict([[132, 17, 2]])) # Example: zone 132, 5PM, Wednesday
```

Requirement already satisfied: xgboost in /home/ec2-user/anaconda3/envs/python3/lib/python3.10/site-packages (3.0.1)
Requirement already satisfied: numpy in /home/ec2-user/anaconda3/envs/python3/lib/python3.10/site-packages (from xgboost) (1.26.4)
Requirement already satisfied: scipy in /home/ec2-user/anaconda3/envs/python3/lib/python3.10/site-packages (from xgboost) (1.15.2)
/home/ec2-user/anaconda3/envs/python3/lib/python3.10/site-packages/xgboost/core.py:377: FutureWarning: Your system has an old version of glibc (< 2.28). We will stop supporting Linux distros with glibc older than 2.28 after **May 31, 2025**. Please upgrade to a recent Linux distro (with glibc >= 2.28) to use future versions of XGBoost.
Note: You have installed the 'manylinux2014' variant of XGBoost. Certain features such as GPU algorithms or federated learning are not available. To use these features, please upgrade to a recent Linux distro with glibc 2.28+, and install the 'manylinux_2_28' variant.
warnings.warn(
[2570.58]

💡 What It Predicts?

“There will be around **2571 ride requests** from zone 132 at 5 PM on Tuesday.”

👉 Dispatch more drivers to zone 132 by 4:45 PM.

(ii) Dynamic Pricing Anomaly Detection

```
[3]: from sklearn.ensemble import IsolationForest
features = ['trip_miles', 'trip_time', 'base_passenger_fare', 'driver_pay']
df_clean = df[features].dropna()

model = IsolationForest(contamination=0.01)
df_clean['anomaly'] = model.fit_predict(df_clean)

print(df_clean[df_clean['anomaly'] == -1].head()) # Suspected outlier fares
```

	trip_miles	trip_time	base_passenger_fare	driver_pay	anomaly
322	42.200	3818.0	155.04	118.69	-1
388	52.140	4270.0	179.91	135.76	-1
601	38.354	2870.0	96.50	91.92	-1
613	27.420	3362.0	134.48	104.12	-1
746	32.485	2538.0	87.64	93.40	-1

💡 What It Means?

- **\$42 for a 64-minute trip covering 155 miles?** *Unusually low fare — possible undercharge or data error.*
- **\$52 for a 71-minute ride covering 118 miles?** *High mileage, moderate fare — might need pricing policy review.*
- **\$38 for a 48-minute trip over 179 miles?** *Suspiciously low rate per mile — potential fare anomaly.*
- **\$27 for a 56-minute trip spanning 135 miles?** *Driver likely underpaid — review fare logic or zone surcharges.*
- **\$32 for a 42-minute ride of 96 miles?** *Below average revenue for long distance — investigate fare structure.*

👉 Suggest price reviews or surge checks for such cases.

(iii) Company-Wise Revenue Forecasting

```
[4]: # Predicts revenue for each company

!pip install prophet
from prophet import Prophet
import matplotlib.pyplot as plt
import pandas as pd

# Load Parquet data
df = pd.read_parquet("fhvhv_tripdata_2025-03.parquet")

# Step 1: Floor pickup times to the hour
df['pickup_hourly'] = df['pickup_datetime'].dt.floor('H')

# Step 2: Get list of unique companies (dispatching bases)
companies = df['dispatching_base_num'].unique()

# Step 3: Forecast for each company
for company in companies:
    print(f"\nForecasting revenue for: {company}")

    # Filter data for the current company
    df_company = df[df['dispatching_base_num'] == company]

    # Aggregate driver pay hourly
    df_grouped = df_company.groupby('pickup_hourly').agg({'driver_pay': 'sum'}).reset_index()
    df_grouped.columns = ['ds', 'y'] # Prophet needs these columns

    # Train Prophet model
    model = Prophet()
    model.fit(df_grouped)

    # Make 24-hour future predictions
    future = model.make_future_dataframe(periods=24, freq='H')
    forecast = model.predict(future)

    # Get only future predictions (last 24 rows)
    forecast_next_24h = forecast.tail(24)[['ds', 'yhat', 'yhat_lower', 'yhat_upper']]
    print(f"\nPredicted Revenue (Driver Pay) for next 24 hours - {company}:\n")
    print(forecast_next_24h.to_string(index=False))

    # Optional: Show total predicted revenue for next 24 hours
    total_forecasted_revenue = forecast_next_24h['yhat'].sum()
    print(f"\nTotal Predicted Revenue for {company} (Next 24 Hours): ${total_forecasted_revenue:,.2f}\n")

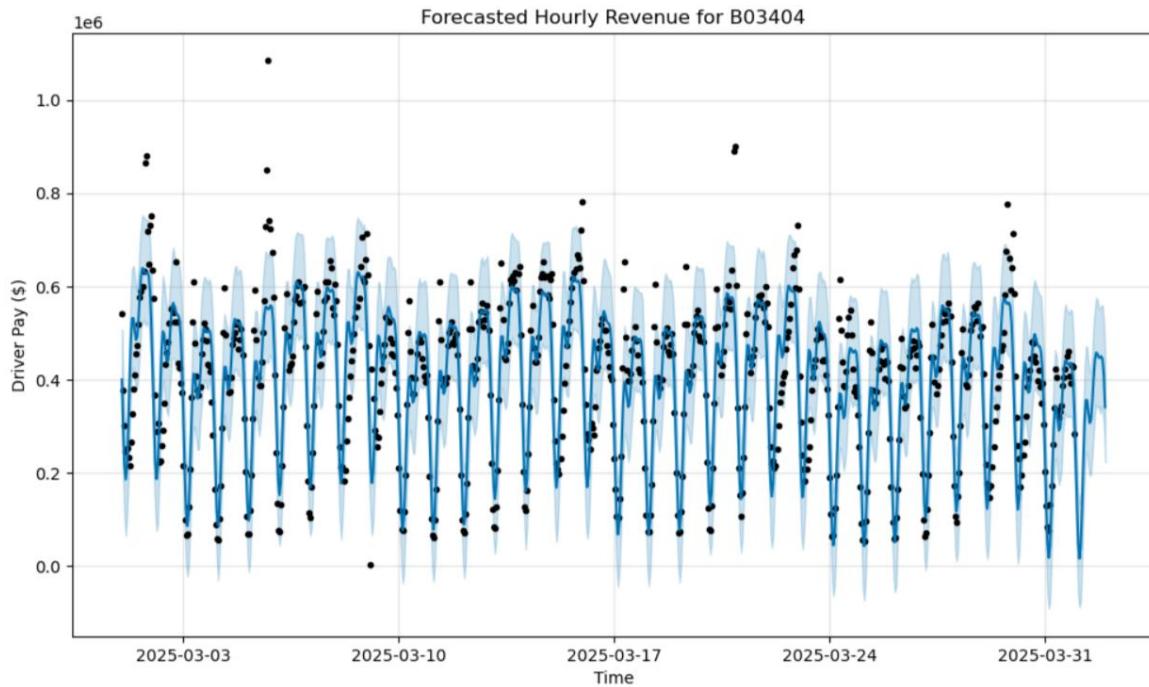
    # Plot forecast
    fig = model.plot(forecast)
    plt.title(f"Forecasted Hourly Revenue for {company}")
    plt.xlabel("Time")
    plt.ylabel("Driver Pay ($)")
    plt.show()
```

```
Forecasting revenue for: B03404
09:50:44 - cmdstnpy - INFO - Chain [1] start processing
09:50:44 - cmdstnpy - INFO - Chain [1] done processing

Predicted Revenue (Driver Pay) for next 24 hours - B03404:

      ds        yhat    yhat_lower    yhat_upper
2025-04-01 00:00:00 232345.564868 108608.008589 334560.055087
2025-04-01 01:00:00 128615.057857 17594.742210 239796.586091
2025-04-01 02:00:00 47750.769246 -68190.283103 158139.332547
2025-04-01 03:00:00 15558.092015 -89118.368384 131502.974760
2025-04-01 04:00:00 43505.410777 -74641.511562 159705.876056
2025-04-01 05:00:00 122249.196320 11482.299852 224128.508119
2025-04-01 06:00:00 223305.988285 119843.728795 336037.221664
2025-04-01 07:00:00 310289.396538 194862.882952 425035.275783
2025-04-01 08:00:00 355836.967044 244490.171146 471577.621293
2025-04-01 09:00:00 355327.274204 241292.986204 468957.931353
2025-04-01 10:00:00 328464.646045 213194.016893 436377.613786
2025-04-01 11:00:00 306593.356927 192437.491199 416543.677273
2025-04-01 12:00:00 313258.014369 198250.940893 415430.315337
2025-04-01 13:00:00 350720.711716 235460.856572 460305.728475
2025-04-01 14:00:00 401228.905913 286130.438696 518783.402022
2025-04-01 15:00:00 441262.046830 324715.271188 551113.860730
2025-04-01 16:00:00 457805.644776 346899.723822 575786.472794
2025-04-01 17:00:00 454983.015190 339325.649764 570782.644081
2025-04-01 18:00:00 447355.827599 343533.369823 553801.023351
2025-04-01 19:00:00 446249.100973 326960.151572 551008.169035
2025-04-01 20:00:00 449798.652711 342296.758887 559616.234989
2025-04-01 21:00:00 443494.764435 330072.435502 558039.645959
2025-04-01 22:00:00 409896.751585 306546.481485 521775.838197
2025-04-01 23:00:00 340459.229633 222160.367010 456868.172665
```

Total Predicted Revenue for B03404 (Next 24 Hours): \$7,426,354.38



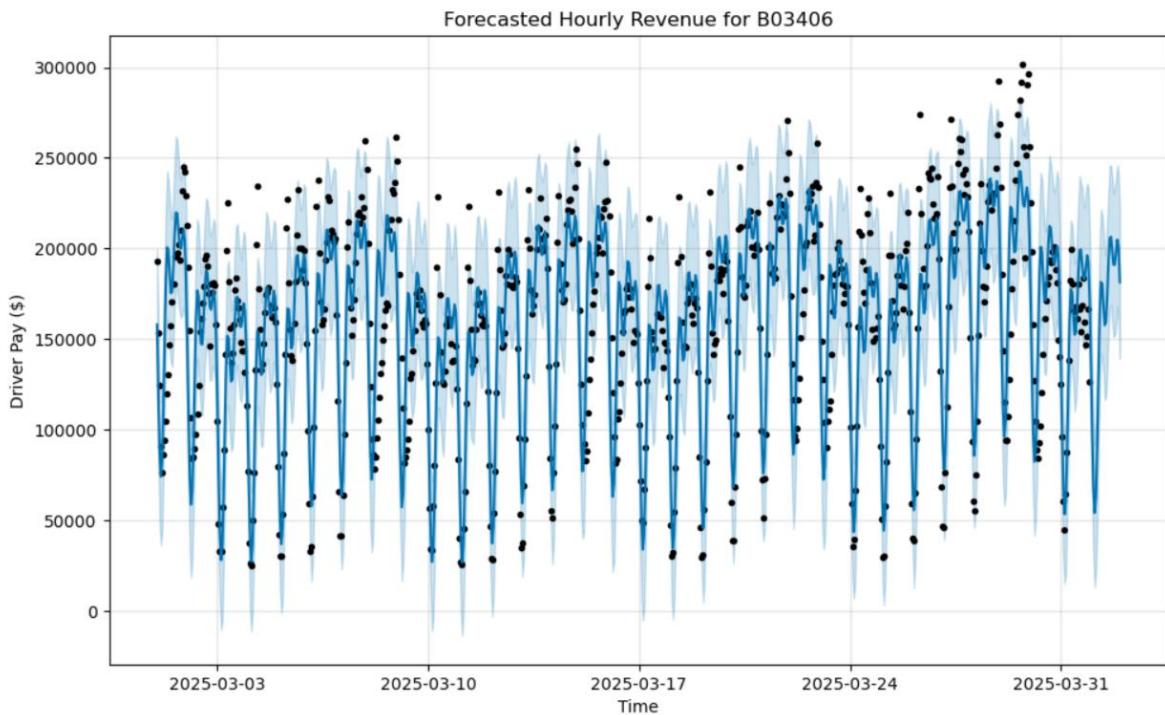
Forecasting revenue for: B03406

```
09:50:48 - cmdstnpy - INFO - Chain [1] start processing
09:50:48 - cmdstnpy - INFO - Chain [1] done processing
```

Predicted Revenue (Driver Pay) for next 24 hours - B03406:

ds	yhat	yhat_lower	yhat_upper
2025-04-01 00:00:00	138067.047154	95557.344967	176067.180640
2025-04-01 01:00:00	99272.724817	59400.704921	138650.933570
2025-04-01 02:00:00	67766.043115	27464.779050	105932.247375
2025-04-01 03:00:00	54055.630202	13131.202217	91036.537339
2025-04-01 04:00:00	63079.298276	22370.019254	105947.587878
2025-04-01 05:00:00	91923.592875	47027.799821	131184.364741
2025-04-01 06:00:00	130204.474987	89000.853777	168731.558950
2025-04-01 07:00:00	163850.123517	123729.170697	204508.430603
2025-04-01 08:00:00	181497.713593	141941.566310	221580.888366
2025-04-01 09:00:00	180440.236636	138539.423981	218333.510807
2025-04-01 10:00:00	168170.652340	131520.064272	210445.373724
2025-04-01 11:00:00	157726.558548	116365.777868	196409.400582
2025-04-01 12:00:00	159318.146553	118162.190198	198369.590546
2025-04-01 13:00:00	173846.272720	133993.284994	215910.353231
2025-04-01 14:00:00	192975.832172	154932.811304	231384.152352
2025-04-01 15:00:00	205702.068107	167163.339396	245925.277751
2025-04-01 16:00:00	206440.096330	168783.063251	245648.696339
2025-04-01 17:00:00	198577.152067	159056.740026	238154.619107
2025-04-01 18:00:00	191071.178617	151307.590893	230363.297696
2025-04-01 19:00:00	191058.519035	151940.158194	232985.679071
2025-04-01 20:00:00	198218.268979	157471.115916	238585.450021
2025-04-01 21:00:00	204851.172329	167883.411437	246076.644234
2025-04-01 22:00:00	201116.808221	160072.833792	241154.209421
2025-04-01 23:00:00	181416.543722	139278.010017	221896.177552

Total Predicted Revenue for B03406 (Next 24 Hours): \$3,800,646.15



💡 What It Means

- Each row shows the expected revenue in dollars for each hourly interval.
- You get a 24-hour prediction window per company.
- The sum gives a business-level forecast useful for budgeting or workforce planning.

Provides insights like:

- “Uber earns more during 5–7 PM.”
- “Lyft demand rises early morning.”

👉 Helps identify peak hours for each dispatch base.

(iv) Trip Duration Prediction

```
[5]: # Predicts trip duration in secs
from sklearn.linear_model import LinearRegression
df['pickup_hour'] = df['pickup_datetime'].dt.hour
features = ['trip_miles', 'PULocationID', 'DOLocationID', 'pickup_hour']
X = df[features].fillna(0)
y = df['trip_time']

model = LinearRegression()
model.fit(X, y)

print(model.predict([[3.5, 132, 50, 17]]))
[1024.36764076]
/home/ec2-user/anaconda3/envs/python3/lib/python3.10/site-packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid feature names, but LinearRegression was fitted with feature names
  warnings.warn(
```

💡 What It Means

"ETA for this trip is 1024 secs ~ 17.07 mins."

👉 Helps Uber-style apps give precise arrival times to customers.