Amazon Stocks Market Predictions

In [ ]:
```
from google.colab import files
uploaded = files.upload()
```

Choose files No file chosen

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving Bulls and bears.PNG to Bulls and bears.PNG
```

In [ ]:
```
from IPython.display import Image

display(Image(filename='Bulls and bears.PNG'))
```

A. Business Overview

• Stock price prediction is a challenging but important task in the financial industry.
• Traditional stock price prediction methods have limitations.

Objective: To develop a robust stock price prediction model for Amazon stock market

Background

"In the world of finance, Warren Buffett once wisely stated, 'The stock market is a device for transferring money from the impatient to the patient.' This insightful quote reminds us of the fundamental truth that successful investing often requires a patient and long-term approach. With this perspective in mind, we embark on a journey to predict the future stock prices of one of the world's most iconic companies, Amazon. While the stock market can be a complex and unpredictable arena in the short term, we aim to harness the potential of machine learning to uncover patterns and trends that may offer valuable insights into Amazon's future performance

Business Problem

The primary challenge in predicting stock prices, especially in the short term, is the inherent volatility and unpredictability of the stock market. Investors and fund managers often face the problem of making optimal investment decisions. They need accurate forecasts to decide when to buy, sell, or hold Amazon's stock, but the accuracy of predictions can vary widely. Bulls and Bears seek to develop a reliable predictive model that helps investors make informed decisions about their Amazon stock holdings to maximize returns while managing risks. This problem encompasses the need for accurate predictions and the application of these predictions to real-world investment strategies.

B. Data Understanding and EDA

In [3]:
```
!pip install yfinance
import yfinance as yf
```

```
Requirement already satisfied: yfinance in /usr/local/lib/python3.10/dis
t-packages (0.2.31)
Requirement already satisfied: pandas>=1.3.0 in /usr/local/lib/python3.1
0/dist-packages (from yfinance) (1.5.3)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.1
0/dist-packages (from yfinance) (1.23.5)
Requirement already satisfied: requests>=2.31 in /usr/local/lib/python3.
10/dist-packages (from yfinance) (2.31.0)
Requirement already satisfied: multitasking>=0.0.7 in /usr/local/lib/pyt
hon3.10/dist-packages (from yfinance) (0.0.11)
Requirement already satisfied: lxml>=4.9.1 in /usr/local/lib/python3.10/
dist-packages (from yfinance) (4.9.3)
Requirement already satisfied: appdirs>=1.4.4 in /usr/local/lib/python3.
10/dist-packages (from yfinance) (1.4.4)
Requirement already satisfied: pytz>=2022.5 in /usr/local/lib/python3.1
0/dist-packages (from yfinance) (2023.3.post1)
Requirement already satisfied: frozendict>=2.3.4 in /usr/local/lib/pytho
n3.10/dist-packages (from yfinance) (2.3.8)
Requirement already satisfied: peewee>=3.16.2 in /usr/local/lib/python3.
```

In [4]:
```
# python.exe -m pip install --upgrade pip
```

In [5]:
```
pip install prophet
```

```
Requirement already satisfied: prophet in /usr/local/lib/python3.10/dist
-packages (1.1.5)
Requirement already satisfied: cmdstanpy>=1.0.4 in /usr/local/lib/python
3.10/dist-packages (from prophet) (1.2.0)
Requirement already satisfied: numpy>=1.15.4 in /usr/local/lib/python3.1
0/dist-packages (from prophet) (1.23.5)
Requirement already satisfied: matplotlib>=2.0.0 in /usr/local/lib/pytho
n3.10/dist-packages (from prophet) (3.7.1)
Requirement already satisfied: pandas>=1.0.4 in /usr/local/lib/python3.1
0/dist-packages (from prophet) (1.5.3)
Requirement already satisfied: holidays>=0.25 in /usr/local/lib/python3.
10/dist-packages (from prophet) (0.34)
Requirement already satisfied: tqdm>=4.36.1 in /usr/local/lib/python3.1
0/dist-packages (from prophet) (4.66.1)
Requirement already satisfied: importlib-resources in /usr/local/lib/pyt
hon3.10/dist-packages (from prophet) (6.1.0)
Requirement already satisfied: stanio~=0.3.0 in /usr/local/lib/python3.1
0/dist-packages (from cmdstanpy>=1.0.4->prophet) (0.3.0)
Requirement already satisfied: python-dateutil in /usr/local/lib/python
```

In [6]:

```
# Import libraries
import yfinance as yf

import numpy as np
import pandas as pd
import matplotlib
from matplotlib import pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
from matplotlib.pylab import rcParams
matplotlib.rcParams['timezone'] = 'UTC'
import seaborn as sns
import matplotlib.pyplot as plt

from scipy import stats
from random import gauss as gs
import math
import datetime
import random

from math import sqrt

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from keras.models import Sequential
from keras.layers import LSTM,Dense,Dropout,Flatten,Input
from keras.layers import LSTM,Dense,Dropout,MaxPooling1D,TimeDistributed,Cor
from keras.models import load_model
from keras.models import Model

import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

import statsmodels.api as sm
import statsmodels.tsa.stattools as st
from statsmodels.tsa.arima_model import ARMA
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller, acf, pacf
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
from pandas.plotting import autocorrelation_plot, lag_plot
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.statespace.sarimax import SARIMAX
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tools.sm_exceptions import ConvergenceWarning
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.api import ARIMA
from statsmodels.tsa.stattools import adfuller

import xgboost as xgb
```

```
from prophet import Prophet

import warnings
warnings.filterwarnings('ignore')

import itertools
```

In [7]:
```
pfz = yf.download("AMZN",start='2018-01-01')
pfz = pd.DataFrame(pfz)
pfz.head(5)
```

```
[**********100%%**********]  1 of 1 completed
```

Out[7]:

Date	Open	High	Low	Close	Adj Close	Volume
2018-01-02	58.599998	59.500000	58.525501	59.450500	59.450500	53890000
2018-01-03	59.415001	60.274502	59.415001	60.209999	60.209999	62176000
2018-01-04	60.250000	60.793499	60.233002	60.479500	60.479500	60442000
2018-01-05	60.875500	61.457001	60.500000	61.457001	61.457001	70894000
2018-01-08	61.799999	62.653999	61.601501	62.343498	62.343498	85590000

In [8]:
```
#loading the data and viewing the first five rows
data = yf.download("AMZN",start='2008-01-01')

df = pd.DataFrame(data)
df.head(5)
```

```
[**********100%%**********]  1 of 1 completed
```

Out[8]:

Date	Open	High	Low	Close	Adj Close	Volume
2008-01-02	4.7675	4.8715	4.7350	4.8125	4.8125	277174000
2008-01-03	4.8030	4.8625	4.7260	4.7605	4.7605	182450000
2008-01-04	4.6630	4.6700	4.4250	4.4395	4.4395	205400000
2008-01-07	4.4310	4.5285	4.2735	4.4410	4.4410	199632000
2008-01-08	4.3775	4.5915	4.3465	4.3940	4.3940	245666000

In [9]:
```
#checking columns in the data frame
df.columns
```

Out[9]:
```
Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume'], dtype='obje
ct')
```

In [10]:
```
df.tail()
```

Out[10]:

Date	Open	High	Low	Close	Adj Close	Volume
2023-10-11	129.740005	132.050003	129.610001	131.830002	131.830002	40741800
2023-10-12	132.169998	134.479996	131.229996	132.330002	132.330002	55528600
2023-10-13	132.979996	133.309998	128.949997	129.789993	129.789993	45786600
2023-10-16	130.690002	133.070007	130.429993	132.550003	132.550003	42790500
2023-10-17	130.389999	132.539993	128.710007	132.221405	132.221405	26096015

Data Features

The data has the following features:

- **Date**: date of the stock price observation.
- **Open price**: opening price of the stock on the given date.
- **High price**: highest price of the stock on the given date.
- **Low price**: lowest price of the stock on the given date.
- **Close price**: closing price of the stock on the given date.
- **Adjusted Close price**: closing price after adjustments for all applicable splits and dividend distributions
- **Volume**: number of shares of the stock traded on the given date.

Shape

- It has 3,960 rows and 5 columns.

Data Types

- All the data is numerical as expected.

Missing Values

- There are no missing values.

Checking for Duplicates

- There are no duplicates in the data.

Time Series Comformity

- The `Date` column is already in `DateTime` format and is the index.

In [11]:
```
df.shape
```

Out[11]:
```
(3976, 6)
```

In [12]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3976 entries, 2008-01-02 to 2023-10-17
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Open       3976 non-null   float64
 1   High       3976 non-null   float64
 2   Low        3976 non-null   float64
 3   Close      3976 non-null   float64
 4   Adj Close  3976 non-null   float64
 5   Volume     3976 non-null   int64  
dtypes: float64(5), int64(1)
memory usage: 217.4 KB
```

In [13]:
```
duplicates = df.duplicated().sum()
duplicates
```

Out[13]:
```
0
```

Summary Statistics

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

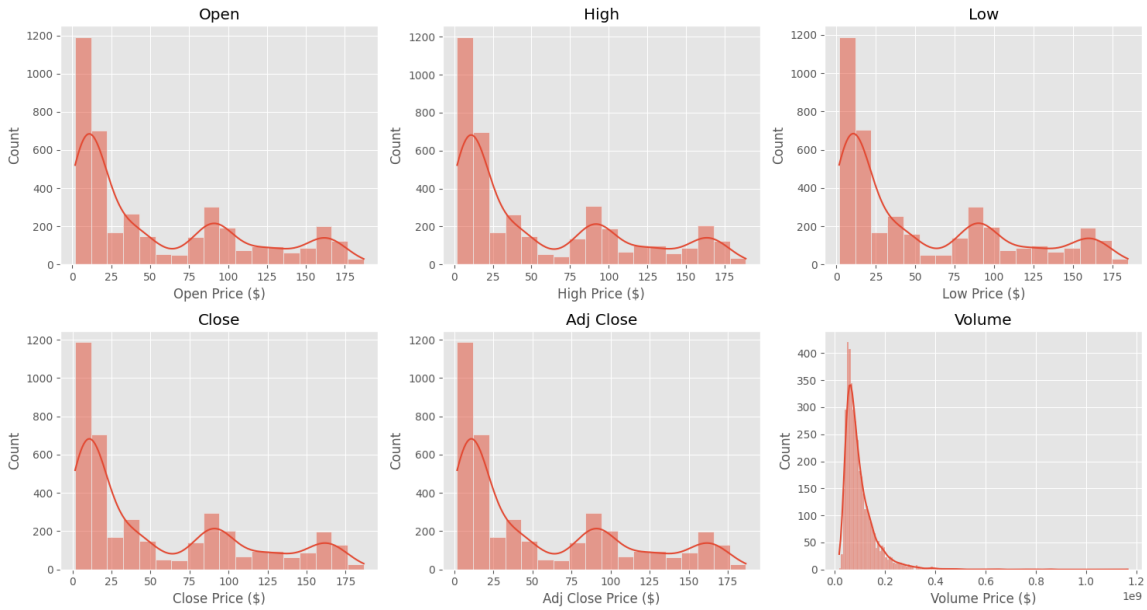
Out[14]:

	Open	High	Low	Close	Adj Close	Volume
count	3976.000000	3976.000000	3976.000000	3976.000000	3976.000000	3.976000e+03
mean	54.799059	55.443058	54.095301	54.781305	54.781305	9.918811e+07
std	54.356712	55.006205	53.638978	54.316355	54.316355	6.880273e+07
min	1.764500	1.950000	1.734000	1.751500	1.751500	1.762600e+07
25%	10.062250	10.258625	9.871750	10.104750	10.104750	5.761100e+07
50%	28.205750	28.557250	27.697750	28.034000	28.034000	7.962100e+07
75%	93.462124	94.518126	92.379002	93.474874	93.474874	1.181865e+08
max	187.199997	188.654007	184.839493	186.570496	186.570496	1.166116e+09

1. Univariate Analysis

a. Distributon of Data Within Columns using Histplots

In [15]:
```
columns_to_plot = ['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']

plt.figure(figsize=(15, 8))

for i, col in enumerate(columns_to_plot, 1):
    plt.subplot(2, 3, i)
    sns.histplot(df[col], kde=True)
    plt.title(f'{col}')
    plt.xlabel(f'{col} Price ($)')
    plt.grid(True)

plt.tight_layout()

plt.show()
```



Observations

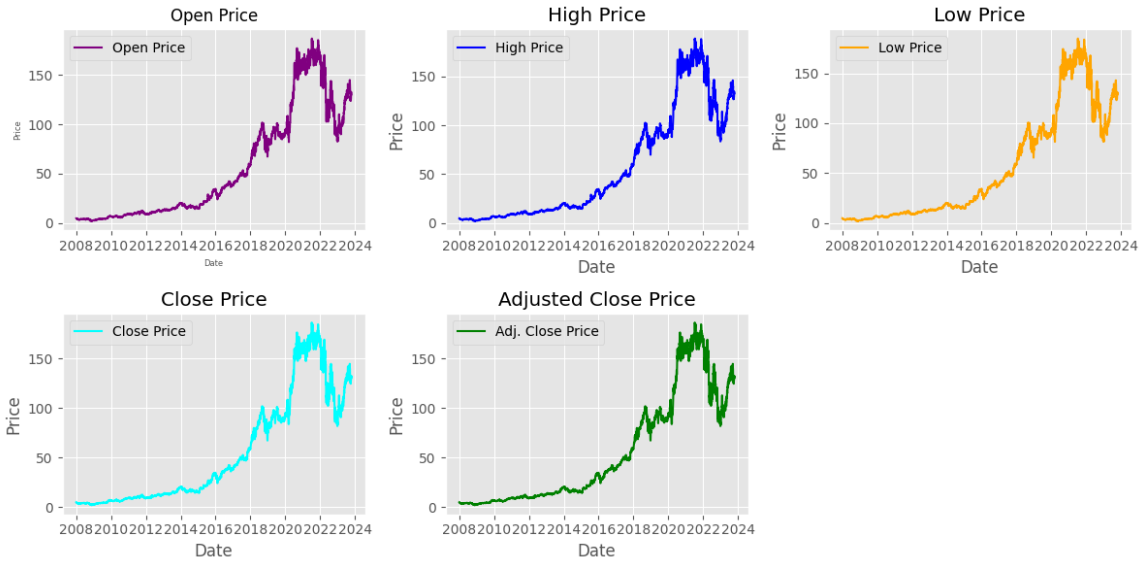
- The **`Open`**, **`High`**, **`Low`**, **`Close`**, **`Adj Close`** plots have similar distributions throughout the period under review (2008 to 2023).
  - They are trimodal (three peaks).
  - 0-25 dollars is the most frequent price.
- Volume seems to be heavily distributed around 100 million to 200 million.
- All are skewed to the right.
- No outliers are visible from these graphs.

b. Time Series Plots for `Open`, `High`, `Low`, `Close`, and `Adj Close` Columns

- The plots below visualize the historical price trends over the period under review.

In [16]:

```
plt.figure(figsize=(12, 6))

# 'Open' price
plt.subplot(2, 3, 1)
plt.plot(df['Open'], label='Open Price', color='purple')
plt.title('Open Price', fontsize=12)
plt.xlabel('Date', fontsize=6)
plt.ylabel('Price', fontsize=6)
plt.legend()

# 'High'price
plt.subplot(2, 3, 2)
plt.plot(df['High'], label='High Price', color='blue')
plt.title('High Price')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()

# 'Low' price
plt.subplot(2, 3, 3)
plt.plot(df['Low'], label='Low Price', color='orange')
plt.title('Low Price')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()

# 'Close' price
plt.subplot(2, 3, 4)
plt.plot(df['Close'], label='Close Price', color='cyan')
plt.title('Close Price')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()

# 'Adj Close' price
plt.subplot(2, 3, 5)
plt.plot(df['Adj Close'], label='Adj. Close Price', color='green')
plt.title('Adjusted Close Price')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()

# Adjust layout for better visualization
plt.tight_layout();
```

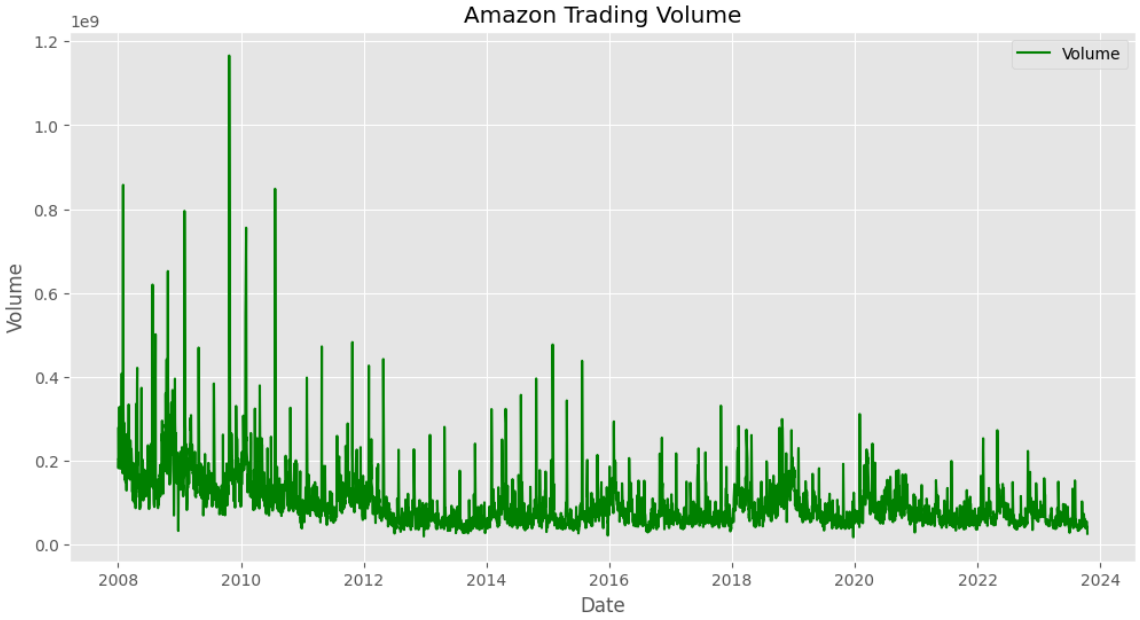


Observations

- The data seems to have similar seasonality and trend characteristics. This will be confirmed in later sections.

Time Series Plot for `Volume` Column

In [17]:
```
plt.figure(figsize=(12, 6))

plt.subplot()
plt.plot(df['Volume'], label='Volume', color='green')
plt.title('Amazon Trading Volume')
plt.xlabel('Date')
plt.ylabel('Volume')
plt.legend();
```



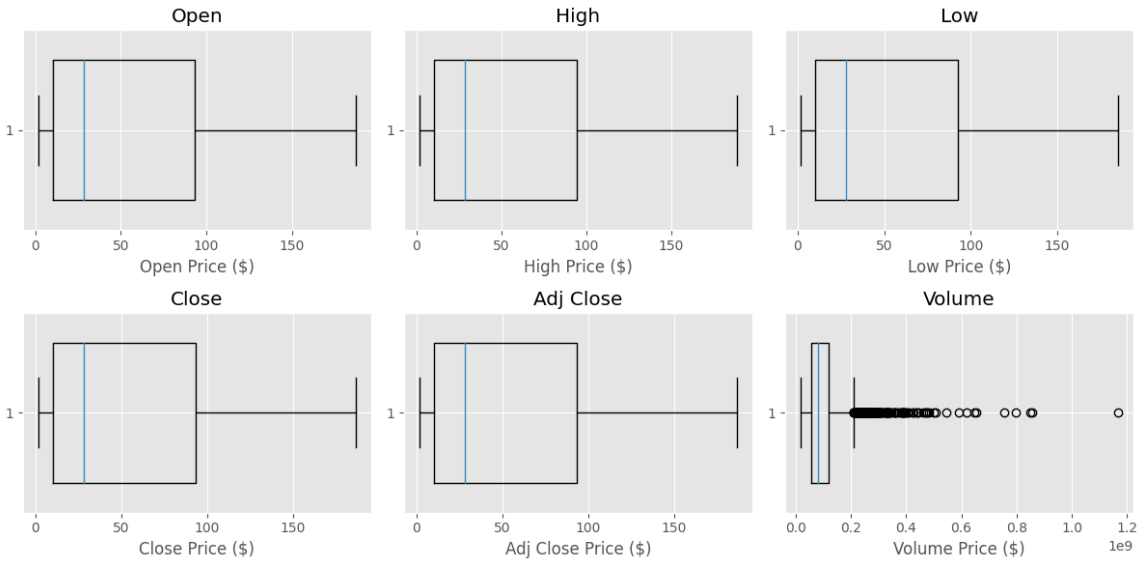
Observation

- The Volume column also looks seasonal and with trend.

- The volume decreases as time progresses with large share volumes bought in earlier years (2008 to 2010). This decreases sharply after 2010 and generally continues to

2. Checking for Outliers or Anomalies using Box Plots

In [18]:
```
columns_to_plot = ['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']

plt.figure(figsize=(12, 6))

for i, col in enumerate(columns_to_plot, 1):
    plt.subplot(2, 3, i)
    plt.boxplot(df[col], vert=False, widths=0.7)
    plt.title(f'{col}')
    plt.xlabel(f'{col} Price ($)')
    plt.grid(True)

plt.tight_layout()

plt.show()
```



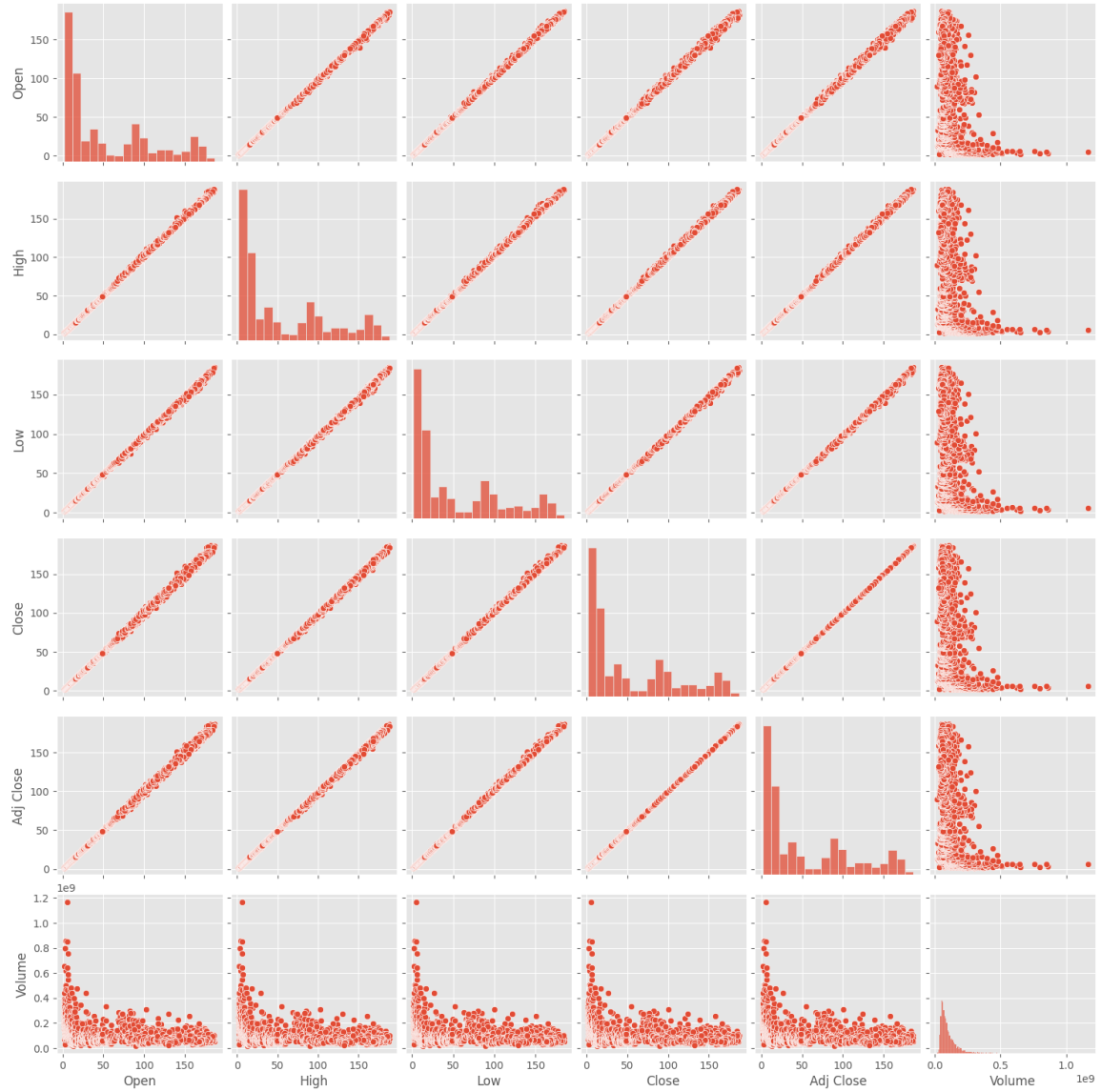
Observations

- The price is centered at approximately 10 and 90 dollars for all columns.
- All features do no have outliers except `Volume` , which has outliers as observed in the box plot.

3. Bivariate and Multivariate Analysis

a. Pairplots and Heatmap to Show Correlation

In [19]:
```
sns.pairplot(df[columns_to_plot])
plt.show()
```

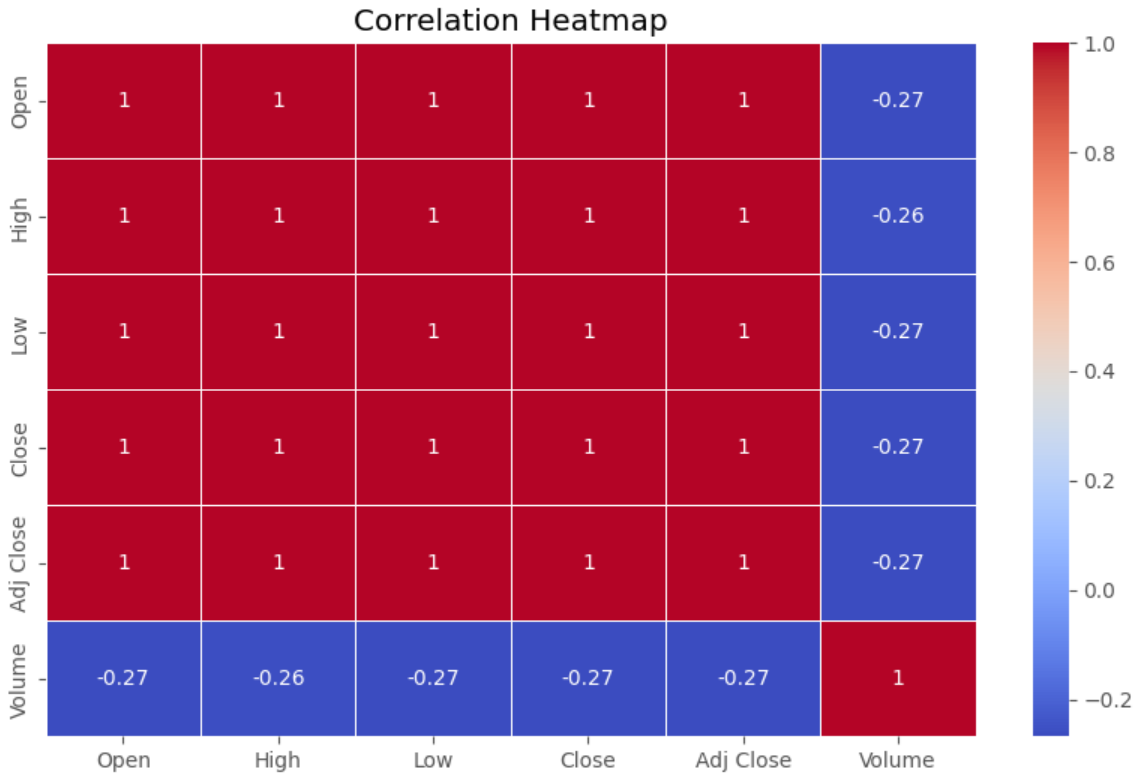


Observations

- The relationship among the variables is similar across the data.
- There is a linear relationship and very strong positive correlation among all features except with 'Volume'.
- There is a non-linear relationship and weak negative correlation with `Volume`.
- These observations are confirmed in the correlation heatmap below.

b. Correlation Heatmap

In [20]:
```
correlation_matrix = df[columns_to_plot].corr()
plt.figure(figsize=(10, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()
```



Observations

External factors that may have caused the positive correlations can be explained as follows:

1. Amazon stock price is driven by the overall performance of the company. When the company is performing well, its stock price tends to go up.
2. It is correlated with the overall performance of the stock market. When the stock market is doing well, Amazon's stock price tends to go up. This is because investors are more likely to buy risky assets, such as stocks, when the market is doing well.

For the weak negative correlation with `Volume` :

1. The volume of trading in Amazon stock is higher when the stock price is volatile. This is because investors are more likely to trade stocks when they are experiencing large price swings.
2. Volume of trading in Amazon stock is higher when there is a lot of news about the company. This is because investors may be more likely to buy or sell shares of the company based on news about its performance, products, or competitive landscape.

The state of the economy, interest rates, inflation and overall market sentiment are the general external factors that may have influenced the patterns visualized above.

Observations

- The time series is seasonal. The price gradually increases as the years go with a sharp rise noted between 2018 and 2022.
- However, there is a decrease towards the year 2023.

- Another increase begins as 2023 comes to an end.

C. Data Preparation

In [21]:
```
main_data = df.copy()
main_data.shape
```

Out[21]:
```
(3976, 6)
```

1. Feature Engineering

####a. Year

Creating a Column for Year from the Date Column.

In [22]:
```
# new columns day_of_week, month, year
main_data['day_of_week'] = main_data.index.dayofweek
main_data['month'] = main_data.index.month
main_data['year'] = main_data.index.year
```

In [23]:
```
# Slice the DataFrame to include only data from the year 2016
main_data = main_data[(main_data['year'] >= 2016) & (main_data.index <= '202
main_data
```

Out[23]:

Date	Open	High	Low	Close	Adj Close	Volume	day_of_week
2016-01-04	32.814499	32.886002	31.375500	31.849501	31.849501	186290000	0
2016-01-05	32.342999	32.345501	31.382500	31.689501	31.689501	116452000	1
2016-01-06	31.100000	31.989500	31.015499	31.632500	31.632500	106584000	2
2016-01-07	31.090000	31.500000	30.260500	30.396999	30.396999	141498000	3
2016-01-08	30.983000	31.207001	30.299999	30.352501	30.352501	110258000	4
...	...	...	...	...	...	...	...
2023-10-02	127.279999	130.470001	126.540001	129.460007	129.460007	48029700	0
2023-10-03	128.059998	128.520004	124.250000	124.720001	124.720001	51565000	1
2023-10-04	126.059998	127.360001	125.680000	127.000000	127.000000	44203900	2
2023-10-05	126.709999	126.730003	124.330002	125.959999	125.959999	39660600	3
2023-10-06	124.160004	128.449997	124.129997	127.959999	127.959999	46795900	4

1954 rows × 9 columns

In [24]:
```
main_data.shape
```

Out[24]:
```
(1954, 9)
```

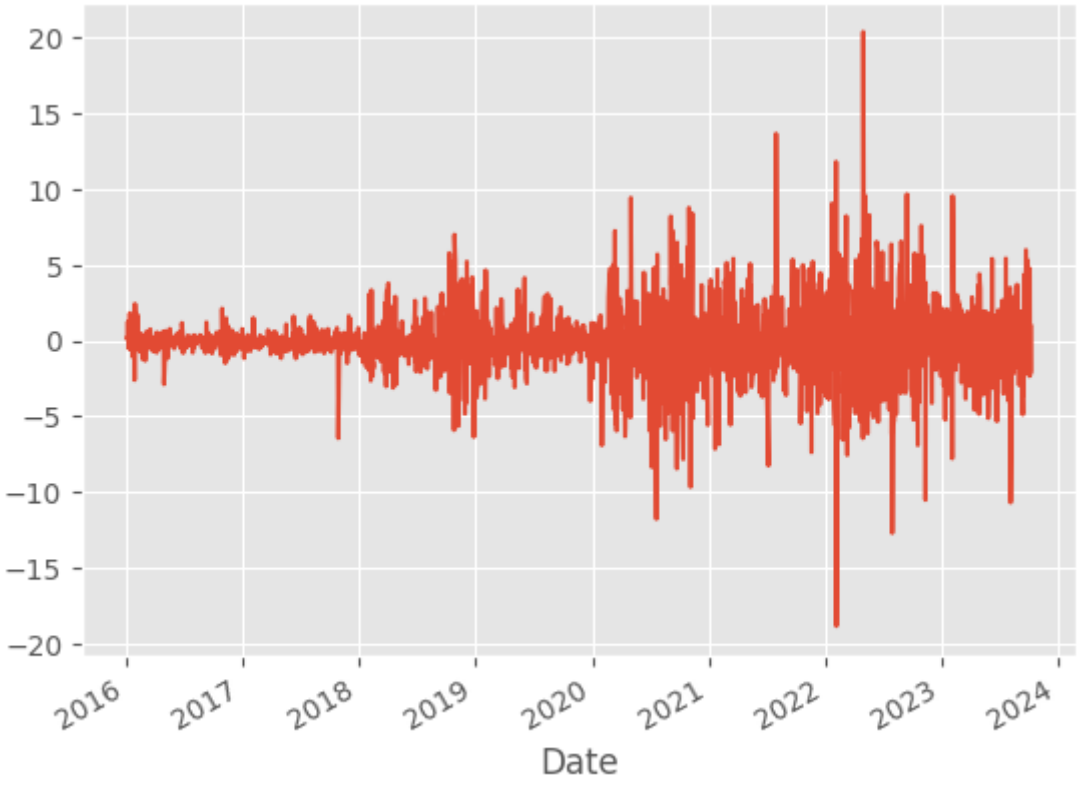
b. Returns

- A new column **Returns** is added to calculate the difference in price for two consecutive days.
- As seen in the plot, the series is seasonal as the period progresses.

In [25]:
```
# difference in two consecutive days
main_data['Returns'] = main_data['Adj Close'].shift(1) - main_data['Adj Clos
main_data['Returns']
```

Out[25]:
```
Date
2016-01-04         NaN
2016-01-05    0.160000
2016-01-06    0.057001
2016-01-07    1.235500
2016-01-08    0.044498
                ...
2023-10-02   -2.340004
2023-10-03    4.740005
2023-10-04   -2.279999
2023-10-05    1.040001
2023-10-06   -2.000000
Name: Returns, Length: 1954, dtype: float64
```

In [26]:
```
main_data['Returns'].plot();
```



c. Lag Features for `Adj Close` price

- Creating lag features will capture the historical behavior of the stock prices.
- Lag features for the adjusted closing price have been calculated in the code below.

In [27]:
```
# Calculate lag features
num_lags = 3  # Number of lag features
for i in range(1, num_lags + 1):
    main_data[f'Adj_Close_Lag_{i}'] = main_data['Adj Close'].shift(i)
```

In [28]:
```
# columns have been successfully created
print(main_data.columns)
```

```
Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', 'day_of_wee
k',
       'month', 'year', 'Returns', 'Adj_Close_Lag_1', 'Adj_Close_Lag_2',
       'Adj_Close_Lag_3'],
      dtype='object')
```

In [29]:
```
# viewing entire df
main_data
```

Out[29]:

Date	Open	High	Low	Close	Adj Close	Volume	day_of_week
2016-01-04	32.814499	32.886002	31.375500	31.849501	31.849501	186290000	0
2016-01-05	32.342999	32.345501	31.382500	31.689501	31.689501	116452000	1
2016-01-06	31.100000	31.989500	31.015499	31.632500	31.632500	106584000	2
2016-01-07	31.090000	31.500000	30.260500	30.396999	30.396999	141498000	3
2016-01-08	30.983000	31.207001	30.299999	30.352501	30.352501	110258000	4
...	...	...	...	...	...	...	...
2023-10-02	127.279999	130.470001	126.540001	129.460007	129.460007	48029700	0
2023-10-03	128.059998	128.520004	124.250000	124.720001	124.720001	51565000	1
2023-10-04	126.059998	127.360001	125.680000	127.000000	127.000000	44203900	2
2023-10-05	126.709999	126.730003	124.330002	125.959999	125.959999	39660600	3
2023-10-06	124.160004	128.449997	124.129997	127.959999	127.959999	46795900	4

1954 rows × 13 columns

d. Rolling Mean and Standard Deviation

- To help capture trends and volatility.

In [30]:
```
window_size = 7  # Rolling window size
main_data['Rolling_Mean'] = main_data['Adj Close'].rolling(window=window_siz
main_data['Rolling_Std'] = main_data['Adj Close'].rolling(window=window_size
```

In [31]:
```
main_data.isnull().sum()
```

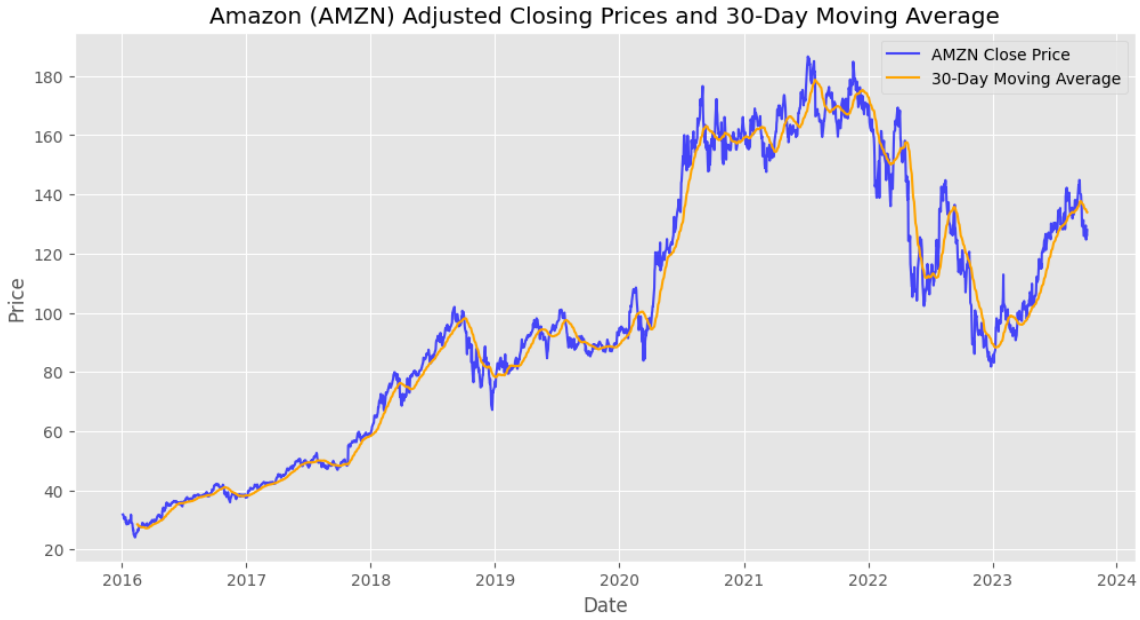
Out[31]:
```
Open               0
High               0
Low                0
Close              0
Adj Close          0
Volume             0
day_of_week        0
month              0
year               0
Returns            1
Adj_Close_Lag_1    1
Adj_Close_Lag_2    2
Adj_Close_Lag_3    3
Rolling_Mean       6
Rolling_Std        6
dtype: int64
```

In [32]:
```
main_data = main_data.fillna(method='bfill')
```

e. Moving Average

- The 30-day moving average tracks the underlying trend in Amazon's closing stock prices over time, helping to smooth out short-term fluctuations and provide insights into longer-term price movements.

In [33]:
```
# Calculate 30-day moving average
main_data['30-Day MA'] = main_data['Adj Close'].rolling(window=30).mean()

plt.figure(figsize=(12, 6))
plt.plot(main_data.index, main_data['Adj Close'], label='AMZN Close Price',
plt.plot(main_data.index, main_data['30-Day MA'], label='30-Day Moving Avera
plt.title('Amazon (AMZN) Adjusted Closing Prices and 30-Day Moving Average')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.show()
```



f. Exponential Moving Average (EMA) Smoothing Factor

- EMA can provide more weight to recent data points, which can be useful for capturing short-term trends.
- The small alpha value of 0.2 will result in a smoother EMA and higher sensitivity to recent price changes.

In [34]:
```
alpha = 0.2
main_data['EMA'] = main_data['Adj Close'].ewm(alpha=alpha).mean()
```

In [35]:
```
unique_days_of_week = main_data['day_of_week'].unique()
unique_days_of_week
```

Out[35]:
```
array([0, 1, 2, 3, 4])
```

g. Rate of Change (ROC) for `Adj Close`

- Measures the percentage change in price over a specified period.
- Positive ROC values indicate upward momentum, while negative values indicate downward momentum.
- It helps traders and analysts identify potential trends or reversals in the price of an asset.

In [36]:
```
period = 5  # ROC period
main_data['ROC'] = (main_data['Adj Close'] - main_data['Adj Close'].shift(pe
```

In [37]:
```
main_data.head()
```

Out[37]:

Date	Open	High	Low	Close	Adj Close	Volume	day_of_week	mont
2016-01-04	32.814499	32.886002	31.375500	31.849501	31.849501	186290000	0	
2016-01-05	32.342999	32.345501	31.382500	31.689501	31.689501	116452000	1	
2016-01-06	31.100000	31.989500	31.015499	31.632500	31.632500	106584000	2	
2016-01-07	31.090000	31.500000	30.260500	30.396999	30.396999	141498000	3	
2016-01-08	30.983000	31.207001	30.299999	30.352501	30.352501	110258000	4	

h. Relative Strength Index (RSI)

- It measures the speed and change of price movements.
- High RSI is normally placed as any value above 75% and a good indicator to sell.
- Low RSI is between 0 and 25%, a good indicator to buy.

In [38]:
```
def calculate_rsi(data, period=14):
    delta = data.diff()
    gain = (delta.where(delta > 0, 0)).fillna(0)
    loss = (-delta.where(delta < 0, 0)).fillna(0)
    avg_gain = gain.rolling(window=period).mean()
    avg_loss = loss.rolling(window=period).mean()
    rs = avg_gain / avg_loss
    rsi = 100 - (100 / (1 + rs))
    return rsi

main_data['RSI'] = calculate_rsi(main_data['Adj Close'])
main_data
```

Out[38]:

Date	Open	High	Low	Close	Adj Close	Volume	day_of_week
2016-01-04	32.814499	32.886002	31.375500	31.849501	31.849501	186290000	0
2016-01-05	32.342999	32.345501	31.382500	31.689501	31.689501	116452000	1
2016-01-06	31.100000	31.989500	31.015499	31.632500	31.632500	106584000	2
2016-01-07	31.090000	31.500000	30.260500	30.396999	30.396999	141498000	3
2016-01-08	30.983000	31.207001	30.299999	30.352501	30.352501	110258000	4
...	...	...	...	...	...	...	...
2023-10-02	127.279999	130.470001	126.540001	129.460007	129.460007	48029700	0
2023-10-03	128.059998	128.520004	124.250000	124.720001	124.720001	51565000	1
2023-10-04	126.059998	127.360001	125.680000	127.000000	127.000000	44203900	2
2023-10-05	126.709999	126.730003	124.330002	125.959999	125.959999	39660600	3
2023-10-06	124.160004	128.449997	124.129997	127.959999	127.959999	46795900	4

1954 rows × 19 columns

i. Average True Range (ATR)

- A volatility indicator.
- It measures the average range between the high and low prices over a specified period.
- It helps traders and analysts assess the level of volatility in the price movements of an asset.

In [39]:
```
# Calculate
def calculate_atr(data, period=14):
    high_low = data['High'] - data['Low']
    high_close = abs(data['High'] - data['Adj Close'].shift(1))
    low_close = abs(data['Low'] - data['Adj Close'].shift(1))
    true_range = pd.concat([high_low, high_close, low_close], axis=1).max(ax
    atr = true_range.rolling(window=period).mean()
    return atr

main_data['ATR'] = calculate_atr(main_data)
main_data
```

Out[39]:

Date	Open	High	Low	Close	Adj Close	Volume	day_of_week
2016-01-04	32.814499	32.886002	31.375500	31.849501	31.849501	186290000	0
2016-01-05	32.342999	32.345501	31.382500	31.689501	31.689501	116452000	1
2016-01-06	31.100000	31.989500	31.015499	31.632500	31.632500	106584000	2
2016-01-07	31.090000	31.500000	30.260500	30.396999	30.396999	141498000	3
2016-01-08	30.983000	31.207001	30.299999	30.352501	30.352501	110258000	4
...	...	...	...	...	...	...	...
2023-10-02	127.279999	130.470001	126.540001	129.460007	129.460007	48029700	0
2023-10-03	128.059998	128.520004	124.250000	124.720001	124.720001	51565000	1
2023-10-04	126.059998	127.360001	125.680000	127.000000	127.000000	44203900	2
2023-10-05	126.709999	126.730003	124.330002	125.959999	125.959999	39660600	3
2023-10-06	124.160004	128.449997	124.129997	127.959999	127.959999	46795900	4

1954 rows × 20 columns

In [40]:
```
main_data.isnull().sum()
```

Out[40]:
```
Open                0
High                0
Low                 0
Close               0
Adj Close           0
Volume              0
day_of_week         0
month               0
year                0
Returns             0
Adj_Close_Lag_1     0
Adj_Close_Lag_2     0
Adj_Close_Lag_3     0
Rolling_Mean        0
Rolling_Std         0
30-Day MA          29
EMA                 0
ROC                 5
RSI                13
ATR                13
dtype: int64
```

In [41]:
```
mean_30_day_MA = main_data['30-Day MA'].mean()
mean_roc = main_data['ROC'].mean()
mean_rsi = main_data['RSI'].mean()
mean_atr = main_data['ATR'].mean()

main_data['ROC'].fillna(mean_roc, inplace=True)
main_data['30-Day MA'].fillna(mean_30_day_MA, inplace=True)
main_data['ATR'].fillna(mean_atr, inplace=True)
main_data['RSI'].fillna(mean_rsi, inplace=True)

main_data.isnull().sum()
```

Out[41]:
```
Open               0
High               0
Low                0
Close              0
Adj Close          0
Volume             0
day_of_week        0
month              0
year               0
Returns            0
Adj_Close_Lag_1    0
Adj_Close_Lag_2    0
Adj_Close_Lag_3    0
Rolling_Mean       0
Rolling_Std        0
30-Day MA          0
EMA                0
ROC                0
RSI                0
ATR                0
dtype: int64
```

2. Seasonality, Stationarity and Trends

a. Checking for Seasonality

- First, we will check if there is seasonality and if it is statistically significant for each column in the time series.
- Remove seasonality using the **`seasonal_decompose`** function.
- Twelve (12) periods are specified to represent a year, the duration of a season in the data.
- After extracting the trend, seasonality and residuals, it plots each of these components in separate subplots.
- It is important to remove seasonality and trend because if they are part of the time series, there will be effects in the forecast value.

Let us look athe **`Adj Close`** plot again

In [42]:
```
plt.plot(main_data['Adj Close'])

# Set the title and axes labels
plt.title('Amazon Stock Price')
plt.xlabel('Date')
plt.ylabel('Stock Price');
```

In [43]:
```python
def check_seasonality(column_name, column_series):
    # Decompose the time series
    decomposition = seasonal_decompose(column_series, model='additive', peri

    # Check for seasonality by observing the seasonal component
    if np.std(decomposition.seasonal) > 0.5:
        print(f"Seasonality: {column_name} is Seasonal")
    else:
        print(f"Seasonality: {column_name} is Not Seasonal")
```

In [44]:
```python
# # Checking seasonality in each of the columns
for column in main_data.columns:
    check_seasonality(column, main_data[column])
```

```
Seasonality: Open is Seasonal
Seasonality: High is Seasonal
Seasonality: Low is Seasonal
Seasonality: Close is Seasonal
Seasonality: Adj Close is Seasonal
Seasonality: Volume is Seasonal
Seasonality: day_of_week is Seasonal
Seasonality: month is Seasonal
Seasonality: year is Not Seasonal
Seasonality: Returns is Seasonal
Seasonality: Adj_Close_Lag_1 is Seasonal
Seasonality: Adj_Close_Lag_2 is Seasonal
Seasonality: Adj_Close_Lag_3 is Seasonal
Seasonality: Rolling_Mean is Seasonal
Seasonality: Rolling_Std is Seasonal
Seasonality: 30-Day MA is Seasonal
Seasonality: EMA is Seasonal
Seasonality: ROC is Seasonal
Seasonality: RSI is Seasonal
Seasonality: ATR is Not Seasonal
```

- Several columns are seasonal.
- This needs to be removed. We shall use a function called **`remove_seasonality`**

In [45]:
```
# Define a function to remove seasonality from a column
def remove_seasonality(column_series):
    # Decompose the time series
    decomposition = seasonal_decompose(column_series, model='additive', peri

    # Remove the seasonal component
    seasonality_removed = column_series - decomposition.seasonal

    return seasonality_removed

# Create a new DataFrame to store the seasonality-removed data
seasonality_removed_data = main_data.copy()

# Iterate through columns with seasonality and remove seasonality
columns_with_seasonality = main_data.columns

for column in columns_with_seasonality:
    seasonality_removed_data[column] = remove_seasonality(main_data[column])
```

In [46]:
```
# Checking seasonality in each of the columns
for column in main_data.columns:
    check_seasonality(column, seasonality_removed_data[column])
```

```
Seasonality: Open is Not Seasonal
Seasonality: High is Not Seasonal
Seasonality: Low is Not Seasonal
Seasonality: Close is Not Seasonal
Seasonality: Adj Close is Not Seasonal
Seasonality: Volume is Not Seasonal
Seasonality: day_of_week is Not Seasonal
Seasonality: month is Not Seasonal
Seasonality: year is Not Seasonal
Seasonality: Returns is Not Seasonal
Seasonality: Adj_Close_Lag_1 is Not Seasonal
Seasonality: Adj_Close_Lag_2 is Not Seasonal
Seasonality: Adj_Close_Lag_3 is Not Seasonal
Seasonality: Rolling_Mean is Not Seasonal
Seasonality: Rolling_Std is Not Seasonal
Seasonality: 30-Day MA is Not Seasonal
Seasonality: EMA is Not Seasonal
Seasonality: ROC is Not Seasonal
Seasonality: RSI is Not Seasonal
Seasonality: ATR is Not Seasonal
```

Now, **`seasonality_removed_data`** contains the original data with seasonality removed for all columns.

b. Checking for Stationarity

- The **Augmented Dickey-Fuller (ADF) test** will be used to determine the stationarity of the time series.
- The ADF test helps us assess whether a time series is stationary by comparing it to the null hypothesis that it has a unit root (meaning it's non-stationary).
- The test involves estimating the model's **coefficients**, calculating a **test statistic**, and comparing it to **critical values** to determine whether the null hypothesis can be rejected.

- The **p-value** resulting from the test indicates the strength of evidence against the null hypothesis.
    - If the p-value is less than or equal to the significance level of 0.05, there's evidence to reject the null hypothesis, suggesting the series is stationary.
    - If the p-value is greater than the significance level of 0.05, there's weak evidence to

In [47]:

```python
# Iterate through columns
for column_name in seasonality_removed_data.select_dtypes(include='number'):
    ts = seasonality_removed_data[column_name]

    # Perform the Augmented Dickey-Fuller (ADF) test
    result = sm.tsa.adfuller(ts)

    # Extract and print the p-value from the test result
    p_value = result[1]
    print(f'{column_name} p-value : {p_value}')

    # Interpret the p-value
    if p_value <= 0.05:
        print(f'{column_name} is stationary.')
        print()
    else:
        print(f'{column_name} is non-stationary.')
        print()
```

```
Open p-value : 0.5284247826223666
Open is non-stationary.

High p-value : 0.5379446758094453
High is non-stationary.

Low p-value : 0.5266668260485647
Low is non-stationary.

Close p-value : 0.5392807119693683
Close is non-stationary.

Adj Close p-value : 0.5392807119693683
Adj Close is non-stationary.

Volume p-value : 9.790166778432027e-12
Volume is stationary.

day_of_week p-value : 4.092585899768034e-11
day_of_week is stationary.

month p-value : 0.00020816339999722532
month is stationary.

year p-value : 0.9314567490012978
year is non-stationary.

Returns p-value : 1.9510000297084976e-29
Returns is stationary.

Adj_Close_Lag_1 p-value : 0.5064656311149403
Adj_Close_Lag_1 is non-stationary.

Adj_Close_Lag_2 p-value : 0.5123049677074848
Adj_Close_Lag_2 is non-stationary.

Adj_Close_Lag_3 p-value : 0.5023467407101239
Adj_Close_Lag_3 is non-stationary.

Rolling_Mean p-value : 0.530762826051246
Rolling_Mean is non-stationary.

Rolling_Std p-value : 0.03729710676853022
Rolling_Std is stationary.

30-Day MA p-value : 0.7698005990066383
30-Day MA is non-stationary.

EMA p-value : 0.5570704367345408
EMA is non-stationary.

ROC p-value : 4.5446849408775765e-13
ROC is stationary.

RSI p-value : 2.0021998927731217e-10
RSI is stationary.

ATR p-value : 0.18111386371880922
ATR is non-stationary.
```

i. Remove Stationarity Through Differencing

- Some of the series is stationary:
  - Volume
  - Returns
  - day_of_week
  - ROC
  - RSI
- The rest are non-stationary.
- Next, differencing will be applied to remove stationarity and make the model-building process accurate and reliable.
- The data's statistical properties will remain consistent, allowing for meaningful insights and accurate predictions.

In [48]:
```
def remove_stationarity_through_differencing(column_series):
    # Apply differencing to make the data more stationary
    differenced_series = column_series.diff().dropna()

# iterate through the columns
for column in seasonality_removed_data.columns:
    remove_stationarity_through_differencing(seasonality_removed_data[columr
    # Check if the differenced training data is stationary
    if p_value < 0.05:
      print(f'The {column} differenced data is stationary (d=1).')
    else:
      print(f'The {column} differenced data is still non-stationary and may
```

```
The Open differenced data is still non-stationary and may require further
differencing.
The High differenced data is still non-stationary and may require further
differencing.
The Low differenced data is still non-stationary and may require further d
ifferencing.
The Close differenced data is still non-stationary and may require further
differencing.
The Adj Close differenced data is still non-stationary and may require fur
ther differencing.
The Volume differenced data is still non-stationary and may require furthe
r differencing.
The day_of_week differenced data is still non-stationary and may require f
urther differencing.
The month differenced data is still non-stationary and may require further
differencing.
The year differenced data is still non-stationary and may require further
differencing.
The Returns differenced data is still non-stationary and may require furth
er differencing.
The Adj_Close_Lag_1 differenced data is still non-stationary and may requi
re further differencing.
The Adj_Close_Lag_2 differenced data is still non-stationary and may requi
re further differencing.
The Adj_Close_Lag_3 differenced data is still non-stationary and may requi
re further differencing.
The Rolling_Mean differenced data is still non-stationary and may require
further differencing.
The Rolling_Std differenced data is still non-stationary and may require f
urther differencing.
The 30-Day MA differenced data is still non-stationary and may require fur
ther differencing.
The EMA differenced data is still non-stationary and may require further d
ifferencing.
The ROC differenced data is still non-stationary and may require further d
ifferencing.
The RSI differenced data is still non-stationary and may require further d
ifferencing.
The ATR differenced data is still non-stationary and may require further d
ifferencing.
```

ii. Further Differencing to Make the Data Stationary

- Further differencing and checks with Augmented Dickley-Fuller Test will be done to make the data stationary.

In [49]:
```
# Perform differencing along the rows (axis=0)
main_data_diff = np.diff(seasonality_removed_data, axis=0)

# Perform Augmented Dickey-Fuller test on the differenced data
result = sm.tsa.adfuller(main_data_diff.ravel())  # Flatten the differenced
p_value = result[1]

print("ADF p-value (main_data_diff):", p_value)

# Check if the differenced data is now stationary
if p_value < 0.05:
    print("The differenced data is stationary (d=1).")
else:
    print("The differenced data is still non-stationary and may require furt
```

```
ADF p-value (main_data_diff): 0.0
The differenced data is stationary (d=1).
```

Observations

- The differencing was successful in removing non-stationarity, making the data stationary.

3. Scaling

In [50]:
```
main_data.head()
```

Out[50]:

Date	Open	High	Low	Close	Adj Close	Volume	day_of_week	mont
2016-01-04	32.814499	32.886002	31.375500	31.849501	31.849501	186290000	0	
2016-01-05	32.342999	32.345501	31.382500	31.689501	31.689501	116452000	1	
2016-01-06	31.100000	31.989500	31.015499	31.632500	31.632500	106584000	2	
2016-01-07	31.090000	31.500000	30.260500	30.396999	30.396999	141498000	3	
2016-01-08	30.983000	31.207001	30.299999	30.352501	30.352501	110258000	4	

In [51]:
```
# Scale the data using MinMax Scaler()
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(main_data_diff)

# Save the scaled data
np.save('scaled_data.npy', scaled_data)
```

- We used Min-Max Scaler to scale the data within a range of (0,1), making it easier to compare and analyse features. This leads to more accurate and reliable predictions.

In [52]:
```
scaled_data.shape
```

Out[52]:
```
(1953, 20)
```

In [53]:
```
main_data_diff.dtype
```

Out[53]:
```
dtype('float64')
```

In [54]:
```
scaled_data.dtype
```

Out[54]:
```
dtype('float64')
```

In [55]:
```
# convert to df
main_data_diff_df = pd.DataFrame(scaled_data, columns=main_data.columns)
main_data_diff_df.head()
```

Out[55]:

	Open	High	Low	Close	Adj Close	Volume	day_of_week	month	ye
0	0.431953	0.424719	0.503978	0.543504	0.543504	0.176738	0.583111	0.761843	0.2381
1	0.427464	0.405198	0.531202	0.536907	0.536907	0.454484	0.722328	0.761843	0.2381
2	0.510061	0.487057	0.482942	0.503381	0.503381	0.655777	0.721948	0.761843	0.2381
3	0.468688	0.401680	0.498920	0.499775	0.499775	0.390733	0.583111	0.761843	0.2381
4	0.390251	0.386736	0.456337	0.514937	0.514937	0.425651	0.305439	0.740192	0.2381

In [56]:
```
# adding date column
main_data_diff_df['date'] = main_data.index[:-1]

# Print the DataFrame
main_data_diff_df
```

Out[56]:

	Open	High	Low	Close	Adj Close	Volume	day_of_week	month	
0	0.431953	0.424719	0.503978	0.543504	0.543504	0.176738	0.583111	0.761843	0.2
1	0.427464	0.405198	0.531202	0.536907	0.536907	0.454484	0.722328	0.761843	0.2
2	0.510061	0.487057	0.482942	0.503381	0.503381	0.655777	0.721948	0.761843	0.2
3	0.468688	0.401680	0.498920	0.499775	0.499775	0.390733	0.583111	0.761843	0.2
4	0.390251	0.386736	0.456337	0.514937	0.514937	0.425651	0.305439	0.740192	0.2
...	...	...	...	...	...	...	...	...	
1948	0.379775	0.378852	0.444517	0.487937	0.487937	0.405140	0.277596	0.848445	0.2
1949	0.387053	0.340353	0.360056	0.391562	0.391562	0.481618	0.583035	0.761843	0.2
1950	0.440157	0.443143	0.662743	0.678568	0.678568	0.476025	0.583035	0.761843	0.2
1951	0.630520	0.529223	0.583816	0.628536	0.628536	0.380420	0.722252	0.761843	0.2
1952	0.500798	0.650644	0.618957	0.678993	0.678993	0.419348	0.861088	0.740192	0.2

1953 rows × 21 columns

In [57]:
```
#Seting the date to be the index and dropping unneccessary columns
main_data_diff_df = main_data_diff_df.set_index('date')
main_data_diff_df = main_data_diff_df.drop(['year', 'day_of_week', 'month'],
main_data_diff_df.head()
```

Out[57]:

	Open	High	Low	Close	Adj Close	Volume	Returns	Adj_Close_Lag_1
date								
2016-01-04	0.431953	0.424719	0.503978	0.543504	0.543504	0.176738	0.500956	0.516450
2016-01-05	0.427464	0.405198	0.531202	0.536907	0.536907	0.454484	0.527626	0.543504
2016-01-06	0.510061	0.487057	0.482942	0.503381	0.503381	0.655777	0.545968	0.536907
2016-01-07	0.468688	0.401680	0.498920	0.499775	0.499775	0.390733	0.525588	0.503381
2016-01-08	0.390251	0.386736	0.456337	0.514937	0.514937	0.425651	0.512804	0.499775

In [58]:
```
#Printing the remaining columns
main_data_diff_df.columns
```

Out[58]:
```
Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', 'Returns',
       'Adj_Close_Lag_1', 'Adj_Close_Lag_2', 'Adj_Close_Lag_3', 'Rolling_M
ean',
       'Rolling_Std', '30-Day MA', 'EMA', 'ROC', 'RSI', 'ATR'],
      dtype='object')
```

In [59]:
```
#Checking the shape of the DataFrame
main_data_diff_df.shape
```

Out[59]:
```
(1953, 17)
```

In [60]:
```
# saving main_data_diff_df to .csv
main_data_diff_df.to_csv('main_data_diff_df.csv', index=False)
```

4. Performing Train-Test Split

In [61]:
```
# Split the DataFrame into x and y variables
X = main_data_diff_df.drop('Adj Close', axis=1)
y = main_data_diff_df['Adj Close']

# Split the x and y variables into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, ra
```

In [62]:
```
X_train.shape
```

Out[62]:
```
(1562, 16)
```

In [63]:
```
y_train.shape
```

Out[63]:
```
(1562,)
```

In [64]:
```
X_test.shape
```

Out[64]:
```
(391, 16)
```

D. Modeling

- For this project, we focus on the Adjusted Close prices which provides a more accurate representation of the stock values over time. The column of the Adjusted Close is therefore used in modeling.

1. Baseline Model - SARIMAX-1

- The project uses the SARIMAX model as the baseline model.
- Using the default values of the p,d,q for the baseline SARIMAX model. The valus of s is our seasonal period which is 5 days per week.

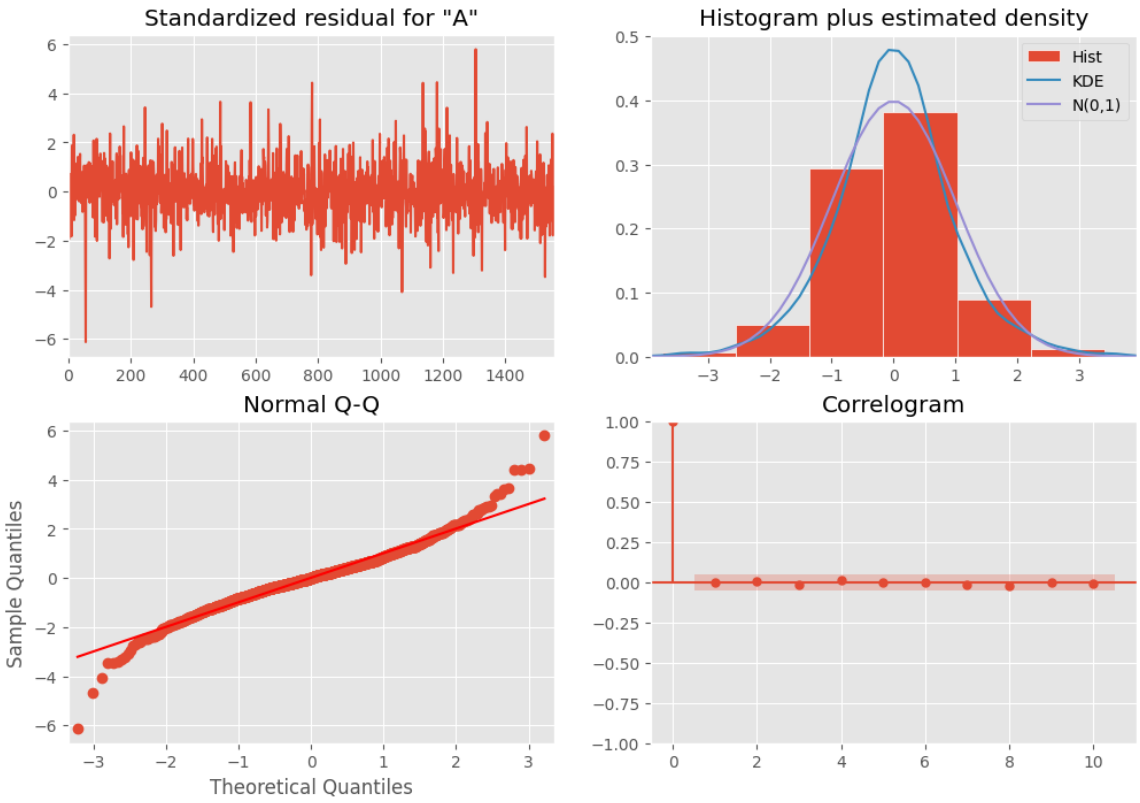
In [65]:

```
# Specify the dafault order and seasonal order for the SARIMA model
p = 1  # AutoRegressive (AR) order
d = 1  # Integrated (I) order (for differencing)
q = 1  # Moving Average (MA) order
seasonal_p = 1  # Seasonal AR order
seasonal_d = 1  # Seasonal I order (for seasonal differencing)
seasonal_q = 1  # Seasonal MA order
s = 5  # Seasonal period

# Select the y_train as the endog variable
endog_series = y_train

# Create the SARIMA model
sarima_1_model = sm.tsa.SARIMAX(endog_series, order=(p, d, q), seasonal_orde

# Fit the SARIMA model to the data
sarima_1_results = sarima_1_model.fit()

# Display the model summary
print(sarima_1_results.summary())
```

```
                                     SARIMAX Results
==========================================================================
==============
Dep. Variable:                          Adj Close   No. Observations:
1562
Model:             SARIMAX(1, 1, 1)x(1, 1, 1, 5)   Log Likelihood
1757.611
Date:                           Tue, 17 Oct 2023   AIC
-3505.222
Time:                                   16:54:18   BIC
-3478.473
Sample:                                        0   HQIC
-3495.275
                                          - 1562
Covariance Type:                            opg
==========================================================================
====
                 coef    std err          z      P>|z|      [0.025      0.
975]
--------------------------------------------------------------------------
----
ar.L1          0.0186      0.023      0.808      0.419      -0.027
0.064
ma.L1         -0.9994      0.040    -24.705      0.000      -1.079      -
0.920
ar.S.L5       -0.0309      0.024     -1.280      0.200      -0.078
0.016
ma.S.L5       -0.9998      0.160     -6.253      0.000      -1.313      -
0.686
sigma2         0.0059      0.001      6.265      0.000       0.004
0.008
==========================================================================
=========
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):
666.75
Prob(Q):                              1.00   Prob(JB):
0.00
Heteroskedasticity (H):               1.07   Skew:
0.07
Prob(H) (two-sided):                  0.43   Kurtosis:
6.20
==========================================================================
=========

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (com
plex-step).
```

- The SARIMAX result shows lower AIC of -3505.222 and BIC of -3478.473 indicating a good performance of the model.
- The higher Log Likelihood of 1757.611 shows that the model has a good fit on the data used.

In [66]:
```
# Display the model diagnostics
sarima_1_results.plot_diagnostics(figsize=(12, 8))
plt.show()
```

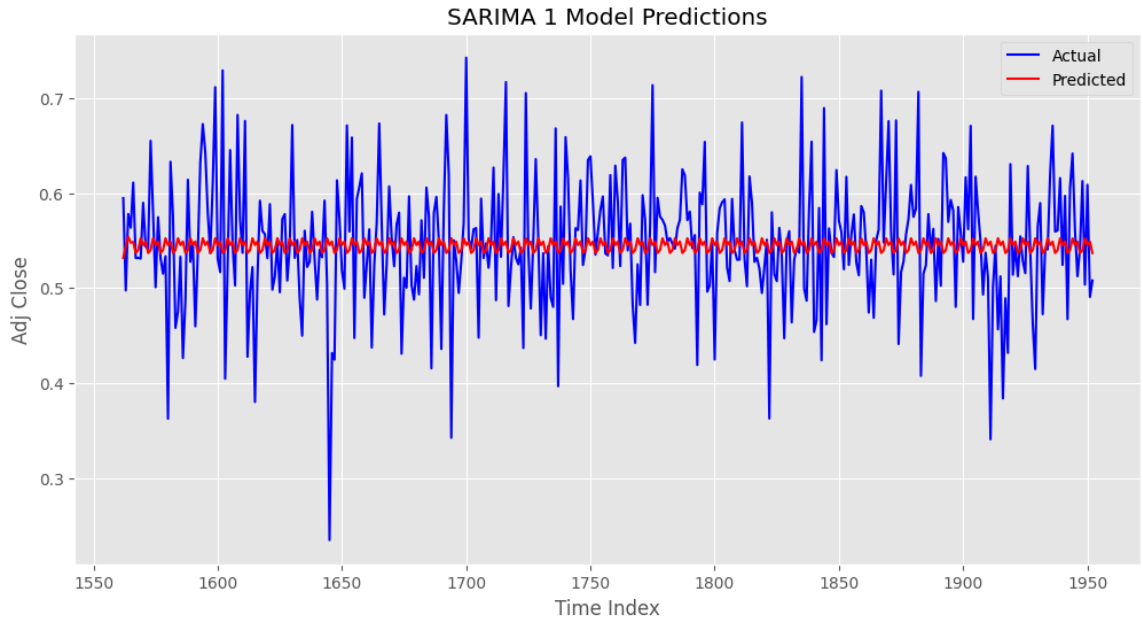


- In the Histogram, the blue KDE line follows closely with the N(0,1) line showing a standard notation for a normal distribution with mean of 0 and standard deviation of 1. This indicates that the residuals are normally distributed.
- The Normal Q-Q plotshows the ordered distribution of Residuals along the linear trend of the sample of a normal distribution with N(0,1) indicating that the residuals are normally distributed.
- The Correlogram shows that the time series residuals have low correlation with the lagged version of itself.
- It is concluded that the SARIMAX - 1 model provides a good fit that can help in forecasting future values.

####i. **Prediction on the Training and Test Data**

- Perform a prediction on both the training dataset and Test dataset to see the the performance of the model on both the Training dataset and Test dataset.

In [67]:
```
# Make predictions for the training and test sets
train_1_predictions = sarima_1_results.predict(start=0, end=len(y_train) - 1
test_1_predictions = sarima_1_results.predict(start=len(y_train), end=len(y_

# Plot the actual and predicted values for the test set
plt.figure(figsize=(12, 6))
plt.plot(np.arange(len(y_train), len(y_train) + len(X_test)), y_test, label=
plt.plot(np.arange(len(y_train), len(y_train) + len(X_test)), test_1_predict
plt.title('SARIMA 1 Model Predictions')
plt.xlabel('Time Index')
plt.ylabel('Adj Close')
plt.legend()
plt.grid(True)
plt.show()
```



####ii. SARIMA-1 Metrics

In [68]:
```
# 'y_test' contains the actual values for the test set
actual_values = y_test

# Calculate MSE, MAE, RMSE, MAPE, R2-Score
mae = mean_absolute_error(actual_values, test_1_predictions)
mse = mean_squared_error(actual_values, test_1_predictions)
rmse = np.sqrt(mse)
r2 = r2_score(actual_values, test_1_predictions)

y_test = np.array(actual_values)
y_pred = np.array(test_1_predictions)
z = pd.DataFrame(np.abs((y_test - y_pred)/y_test)).replace(np.inf, np.nan)
MAPE = z.mean() * 100

print('Mean Absolute Error (MAE) for Test Set Forecast:', mae)
print('Mean Squared Error (MSE) for Test Set Forecast:', mse)
print('Root Mean Square Error (RMSE) for Test Set Forecast:', rmse)
print('Mean Absolute Percentage Error:', MAPE)
print('R-squared (R2) Score for Test Set Forecast:', r2);
```

```
Mean Absolute Error (MAE) for Test Set Forecast: 0.05060312265618778
Mean Squared Error (MSE) for Test Set Forecast: 0.0046668676627897
Root Mean Square Error (RMSE) for Test Set Forecast: 0.06831447623154041
Mean Absolute Percentage Error: 0    9.68145
dtype: float64
R-squared (R2) Score for Test Set Forecast: 0.0016243973279298496
```

- The baseline model seems to have relatively low errors: MAE of 0.05060, MSE of 0.00467, and RMSE of 0.06831. This indicate a good performance of the model in making predictions
- The MAPE value of 9.68145% suggests that, on average, the model's predictions have around a 9.68145% relative error, which is good for the model's performance.
- The r2-score is 0.00162 indicating that the accuracy of the model is low.

In [69]:
```
from datetime import datetime

# Define the start and end dates for your desired date range
start_date = datetime(2023, 10, 7)
end_date = datetime(2025, 10, 6)

# Generate predictions for the specified date range
date_range = pd.date_range(start=start_date, end=end_date, freq='B')
predictions = sarima_1_results.predict(start=len(y_train), end=len(y_train)

# Create a DataFrame with the predictions and date range
predictions_df = pd.DataFrame({'Datetime': date_range, 'Predicted_Adj_Close'

# Save the predictions DataFrame to a CSV file
csv_filename = 'sarima_1_predictions.csv'
predictions_df.to_csv(csv_filename, index=False)
```

In [70]:
```
prediction_df = pd.read_csv('/content/sarima_1_predictions.csv')
prediction_df
```

Out[70]:

	Datetime	Predicted_Adj_Close
0	2023-10-09	0.531707
1	2023-10-10	0.544135
2	2023-10-11	0.553879
3	2023-10-12	0.547756
4	2023-10-13	0.548511
...	...	...
516	2025-09-30	0.539938
517	2025-10-01	0.552372
518	2025-10-02	0.545240
519	2025-10-03	0.548837
520	2025-10-06	0.536892

521 rows × 2 columns

Hyperparameters Tuning Using GridSearch CV

- Hyperparameter tuning using GridSearch CV is done to find the optimal values of hyperparameters (p,d,q) to be used SARIMA-2 modeling

In [71]:
```
# Identifying the column index for the target column
target_column_name = 'Adj Close'
column_names = ['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume',
       'Returns', 'Adj_Close_Lag_1', 'Adj_Close_Lag_2', 'Adj_Close_Lag_3',
       'Rolling_Mean', 'Rolling_Std', '30-Day MA', 'EMA', 'ROC', 'RSI', 'ATR
# Finding the index of the target column name
column_index = np.where(np.in1d(column_names, target_column_name))[0]

# Check if the column name was found
if len(column_index) > 0:
    # The index of 'Adj Close' is in column_index[0]
    print(f"The index of '{target_column_name}' is: {column_index[0]}")
else:
    print(f"'{target_column_name}' not found in the column names.")
```

```
The index of 'Adj Close' is: 4
```

In [72]:
```
#Define a function that calculates d based on the value of p value from the
def find_differencing_order(time_series):
    result = adfuller(time_series)
    p_value = result[1]

    if p_value > 0.05:
        d = 1
    else:
        d = 0

    return d
```

In [73]:
```
# # Define the hyperparameters to search
p_values = range(0, 3)
d_values = [find_differencing_order(y_train)]
q_values = range(0, 3)
s = 5

best_mse = float("inf")
best_order = (0, 0, 0)

for p in p_values:
    for d in d_values:
        for q in q_values:
            # Create and fit the SARIMAX model
            model = SARIMAX(y_train, order=(p, d, q), seasonal_order=(s, 0,
            try:
                model_fit = model.fit(disp=0)
                # Make predictions on the test set
                predictions = model_fit.forecast(steps=len(X_test))
                # Calculate Mean Squared Error (MSE)
                mse = mean_squared_error(y_test, predictions)
                if mse < best_mse:
                    best_mse = mse
                    best_order = (p, d, q)
                    best_seasonal_order = (p, d, q, s)
            except:
                pass

print("Best SARIMAX Order (p, d, q):", best_order)
print("Best SARIMAX Order (p, d, q, s):", best_seasonal_order)
print("Best Mean Squared Error:", best_mse)
```

```
Best SARIMAX Order (p, d, q): (2, 0, 2)
Best SARIMAX Order (p, d, q, s): (2, 0, 2, 5)
Best Mean Squared Error: 0.005872860199745769
```

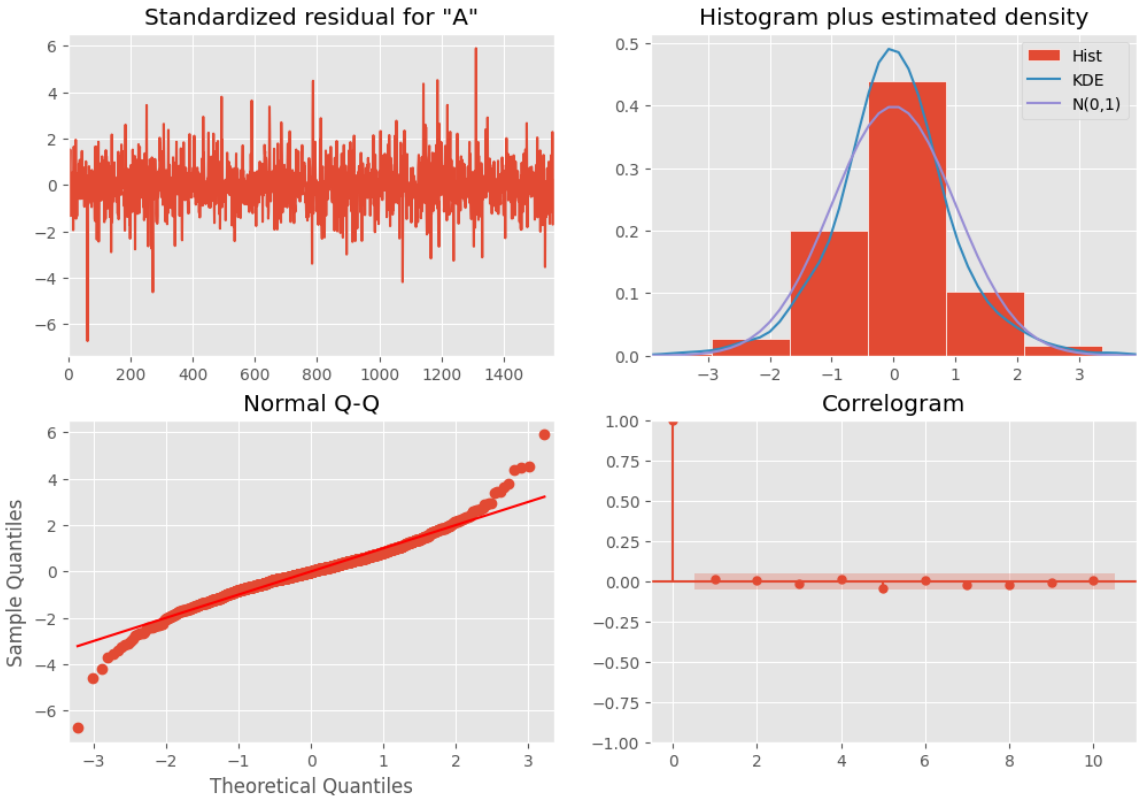
2. SARIMA-2

In [74]:
```
# Specify the order and seasonal order for the SARIMA model
p = 2  # AutoRegressive (AR) order
d = 0  # Integrated (I) order (for differencing)
q = 2  # Moving Average (MA) order
seasonal_p = 2  # Seasonal AR order
seasonal_d = 0  # Seasonal I order (for seasonal differencing)
seasonal_q = 2  # Seasonal MA order
s = 5  # Seasonal period

# Select the y_train as the endog variable
endog_series = y_train

# Create the SARIMA model
sarima_model = sm.tsa.SARIMAX(endog_series, order=(p, d, q), seasonal_order=

# Fit the SARIMA model to the data
sarima_results = sarima_model.fit()

# Display the model summary
print(sarima_results.summary())
```

```
                                     SARIMAX Results
==========================================================================
==============
Dep. Variable:                          Adj Close   No. Observations:
1562
Model:             SARIMAX(2, 0, 2)x(2, 0, 2, 5)   Log Likelihood
1770.502
Date:                           Tue, 17 Oct 2023   AIC
-3523.004
Time:                                   16:57:13   BIC
-3474.821
Sample:                                        0   HQIC
-3505.090
                                          - 1562
Covariance Type:                            opg
==========================================================================
====
                 coef    std err          z      P>|z|      [0.025      0.
975]
--------------------------------------------------------------------------
----
ar.L1          0.0265      0.041      0.643      0.520      -0.054
0.107
ar.L2          0.9734      0.041     23.640      0.000       0.893
1.054
ma.L1         -0.0218      0.065     -0.337      0.736      -0.149
0.105
ma.L2         -0.9776      0.090    -10.891      0.000      -1.154      -
0.802
ar.S.L5        0.1181      0.077      1.529      0.126      -0.033
0.270
ar.S.L10       0.8819      0.077     11.415      0.000       0.730
1.033
ma.S.L5       -0.1038      0.076     -1.362      0.173      -0.253
0.046
ma.S.L10      -0.8920      0.075    -11.853      0.000      -1.039      -
0.744
sigma2         0.0060      0.001     11.839      0.000       0.005
0.007
==========================================================================
=========
Ljung-Box (L1) (Q):                   0.29   Jarque-Bera (JB):
906.81
Prob(Q):                              0.59   Prob(JB):
0.00
Heteroskedasticity (H):               1.06   Skew:
0.02
Prob(H) (two-sided):                  0.54   Kurtosis:
6.73
==========================================================================
=========

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (com
plex-step).
```

- The SARIMA-2 result shows lower AIC of -3523.004 and BIC of -3474.821 indicating a better performance of the model.
- The higher Log Likelihood of 1770.502 shows that the model has a better fit on the data used as compared to the baseline model

In [75]:
```
# Display the model diagnostics
sarima_results.plot_diagnostics(figsize=(12, 8))
plt.show()
```



Observations

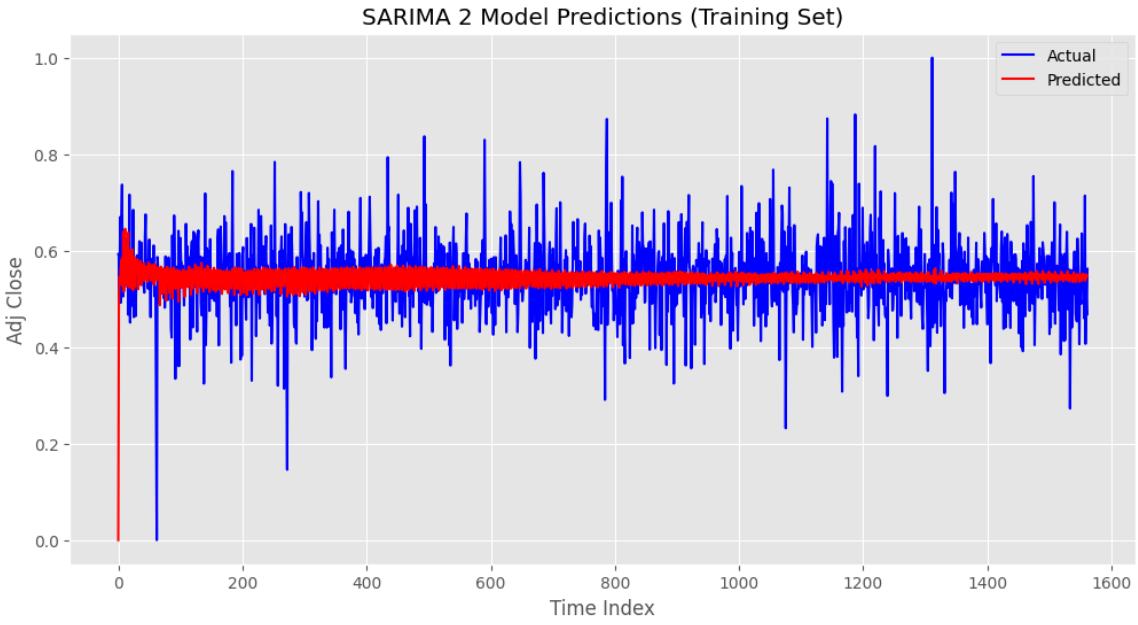
- In the Histogram, the blue KDE line follows closely with the N(0,1) line showing a standard notation for a normal distribution with mean of 0 and standard deviation of 1. This indicates that the residuals are normally distributed.
- The Normal Q-Q plotshows the ordered distribution of Residuals along the linear trend of the sample of a normal distribution with N(0,1) indicating that the residuals are normally distributed.
- The Correlogram shows that the time series residuals have low correlation with the lagged version of itself.
- It is concluded that the SARIMA-2 model provides a better fit that can help in forecasting future values.

i. Predicting on the Training and Test Data

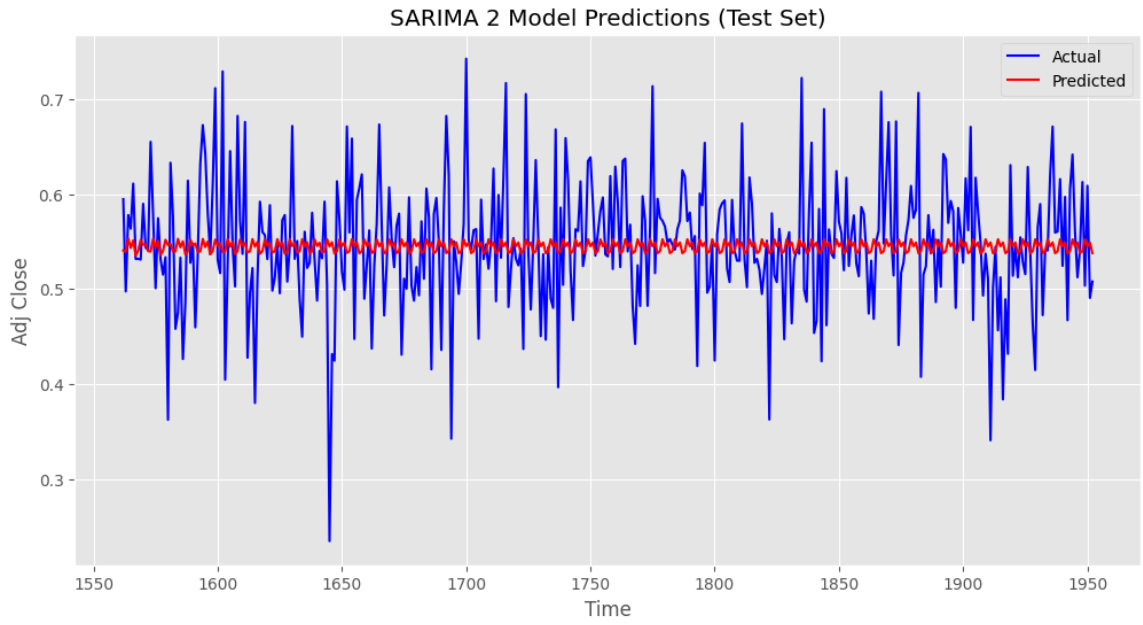
- Perform a prediction on both the training dataset and Test dataset to see the the performance of the model on both the Training dataset and Test dataset.

In [76]:
```
# Make predictions for the training and test sets
train_predictions = sarima_results.predict(start=0, end=len(y_train) - 1, dy
test_predictions = sarima_results.get_forecast(steps=len(X_test))
sarima_predictions = test_predictions.predicted_mean

# Create an array of x-values for test predictions
x_test_index = np.arange(len(y_train), len(y_train) + len(X_test))
```

In [77]:
```
# Plot the actual and predicted values for the training set
plt.figure(figsize=(12, 6))
plt.plot(np.arange(len(y_train)), y_train, label='Actual', color='blue')
plt.plot(np.arange(len(y_train)), train_predictions, label='Predicted', colo
plt.title('SARIMA 2 Model Predictions (Training Set)')
plt.xlabel('Time Index')
plt.ylabel('Adj Close')
plt.legend()
plt.grid(True)
plt.show()
```



In [78]:
```
# Plot the actual and predicted values for the test set
plt.figure(figsize=(12, 6))
plt.plot(x_test_index, y_test, label='Actual', color='blue')
plt.plot(x_test_index, test_predictions.predicted_mean, label='Predicted', c
plt.title('SARIMA 2 Model Predictions (Test Set)')
plt.xlabel('Time')
plt.ylabel('Adj Close')
plt.legend()
plt.grid(True)
plt.show()
```



- The predictions model has a better prediction on both the training and test datasets indicating a better performance of the SARIMA-2 model in making predictions.

In [79]:
```
# Define the start and end dates
start_date = datetime(2023, 10, 7)
end_date = datetime(2025, 10, 6)

# Generate predictions for the specified date range
date_range = pd.date_range(start=start_date, end=end_date, freq='B')
sarima_predictions = sarima_results.predict(start=len(y_test), end=len(y_tes

# Create a DataFrame with the predictions and date range
predictions_df = pd.DataFrame({'Datetime': date_range, 'Predicted_Adj_Close'

# Save the predictions DataFrame to a CSV file
predictions_df.to_csv('sarima_predictions.csv', index=False)
```

In [80]:
```
sarima_predictions_df = pd.read_csv('/content/sarima_predictions.csv')
sarima_predictions_df
```

Out[80]:

	Datetime	Predicted_Adj_Close
0	2023-10-09	0.538356
1	2023-10-10	0.517339
2	2023-10-11	0.545309
3	2023-10-12	0.563070
4	2023-10-13	0.548921
...	...	...
516	2025-09-30	0.533532
517	2025-10-01	0.538962
518	2025-10-02	0.555234
519	2025-10-03	0.542743
520	2025-10-06	0.552333

521 rows × 2 columns

In [81]:
```
forecast_period = 521  # 2 years
forecast = sarima_results.get_forecast(steps=forecast_period)

# Access the predicted values
sarima_predictions2 = forecast.predicted_mean

# Define the start and end dates for your desired date range
start_date = datetime(2023, 10, 7)
end_date = datetime(2025, 10, 6)
date_range = pd.date_range(start=start_date, end=end_date, freq='B')

# Create a DataFrame with the predictions and date range
predictions_df = pd.DataFrame({'Datetime': date_range, 'Predicted_Adj_Close'

# Save the predictions DataFrame to a CSV file
predictions_df.to_csv('sarima_predictions2.csv', index=False)
```

In [82]:
```
sarima_predictions_df2 = pd.read_csv('/content/sarima_predictions2.csv')
sarima_predictions_df2
```

Out[82]:

	Datetime	Predicted_Adj_Close
0	2023-10-09	0.540623
1	2023-10-10	0.539334
2	2023-10-11	0.553312
3	2023-10-12	0.543201
4	2023-10-13	0.551204
...	...	...
516	2025-09-30	0.539910
517	2025-10-01	0.552418
518	2025-10-02	0.545061
519	2025-10-03	0.548976
520	2025-10-06	0.537747

521 rows × 2 columns

ii. SARIMA - 2 Metrics

In [83]:
```
# 'y_test' contains the actual values for the test set
# actual_values = y_test

# Calculate MSE, MAE, RMSE, MAPE,and R2-Score
mae = mean_absolute_error(y_test, test_predictions.predicted_mean)
mse = mean_squared_error(y_test, test_predictions.predicted_mean)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, test_predictions.predicted_mean)

y_test_array = np.array(y_test)  # Convert y_test to an array
y_pred_array = np.array(test_predictions.predicted_mean)  # Convert predicte

mape = np.mean(np.abs((y_test_array - y_pred_array) / y_test_array)) * 100

print('Mean Absolute Error (MAE) for Test Set Forecast:', mae)
print('Mean Squared Error (MSE) for Test Set Forecast:', mse)
print('Root Mean Square Error (RMSE) for Test Set Forecast:', rmse)
print('Mean Absolute Percentage Error:', MAPE)
print('R-squared (R2) Score for Test Set Forecast:', r2)
```

```
Mean Absolute Error (MAE) for Test Set Forecast: 0.050555962813470426
Mean Squared Error (MSE) for Test Set Forecast: 0.004662935239053981
Root Mean Square Error (RMSE) for Test Set Forecast: 0.06828568839115544
Mean Absolute Percentage Error: 0    9.68145
dtype: float64
R-squared (R2) Score for Test Set Forecast: 0.002465654505289794
```

- The model seems to have relatively low errors: MAE of 0.05056, MSE of 0.00466, and RMSE of 0.06829. This indicate better performance of the model in making predictions
- The MAPE value of 9.67% suggests that, on average, the model's predictions have around a 9.67% relative error, which is better for the model's performance.

3. FB Prophet

- Using FBProphet to forecast the data to show trends and future predictions

In [84]:
```
# reviewing the df to use
main_data_diff_df.head()
```

Out[84]:

date	Open	High	Low	Close	Adj Close	Volume	Returns	Adj_Close_Lag_1
2016-01-04	0.431953	0.424719	0.503978	0.543504	0.543504	0.176738	0.500956	0.516450
2016-01-05	0.427464	0.405198	0.531202	0.536907	0.536907	0.454484	0.527626	0.543504
2016-01-06	0.510061	0.487057	0.482942	0.503381	0.503381	0.655777	0.545968	0.536907
2016-01-07	0.468688	0.401680	0.498920	0.499775	0.499775	0.390733	0.525588	0.503381
2016-01-08	0.390251	0.386736	0.456337	0.514937	0.514937	0.425651	0.512804	0.499775

In [85]:

```python
# Create a new DataFrame and rename the column
prophet_df = main_data_diff_df.copy()
prophet_df = pd.DataFrame(prophet_df).reset_index()
prophet_df = prophet_df.rename(columns={'date': 'ds'})

def calculate_rmse(actual, predicted):
    return mean_squared_error(actual, predicted, squared=False)

def calculate_metrics(actual, predicted):
    rmse = calculate_rmse(actual, predicted)
    metrics_df = pd.DataFrame({'RMSE': [rmse]}, index=[actual.name])
    return metrics_df

def prophet_model(column_name):
    new_subset = prophet_df.rename(columns={column_name: 'y'})

    subset_columns = ['ds', 'y']
    prophet_subset = new_subset[subset_columns]

    prophet_subset_reset = prophet_subset.reset_index(drop=True)

    model = Prophet(interval_width=0.95)
    fitted_model = model.fit(prophet_subset_reset)

    future_dates = model.make_future_dataframe(periods=12, freq='MS')
    forecast = model.predict(future_dates)

    # Print and plot the results
    print(f"Results for {column_name}:")
    print(forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail())

    # Plot the forecast and components
    model.plot(forecast, uncertainty=True)
    model.plot_components(forecast)
    plt.show()

    actual = prophet_subset['y']
    predicted = forecast['yhat'][-len(actual):]

    fb_metrics_df = calculate_metrics(actual, predicted)

    return actual, predicted, fb_metrics_df

fb_metrics_dfs = []
```

In [86]:
```
actual, predicted, metrics_df = prophet_model('Adj Close')

# metrics dataframe
metrics_df2 = calculate_metrics(actual, predicted)
fb_metrics_dfs.append(metrics_df2)
```

```
INFO:prophet:Disabling daily seasonality. Run prophet with daily_seasona
lity=True to override this.
DEBUG:cmdstanpy:input tempfile: /tmp/tmpobj5z5v7/orkxfvvz.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpobj5z5v7/ba1o2kaz.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-packages/
prophet/stan_model/prophet_model.bin', 'random', 'seed=71018', 'data',
'file=/tmp/tmpobj5z5v7/orkxfvvz.json', 'init=/tmp/tmpobj5z5v7/ba1o2kaz.j
son', 'output', 'file=/tmp/tmpobj5z5v7/prophet_model0mlkfvw2/prophet_mod
el-20231017165820.csv', 'method=optimize', 'algorithm=lbfgs', 'iter=1000
0']
16:58:20 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
16:58:20 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing

Results for Adj Close:
             ds      yhat  yhat_lower  yhat_upper
1960 2024-06-01  0.548017    0.401651    0.699814
```

Observation

- The model predicts into the future using the historical data provided. The trend shows good accuracy and fit on the data used.

i. FB Prophet Metrics

In [87]:
```
# Calculate MSE, MAE, RMSE, MAPE,and R2-Score
mae = mean_absolute_error(actual, predicted)
mse = mean_squared_error(actual, predicted)
rmse = np.sqrt(mse)
r2 = r2_score(actual, predicted)

z = pd.DataFrame(np.abs((actual - predicted)/actual)).replace(np.inf, np.nan
MAPE = z.mean() * 100

print('Mean Absolute Error (MAE): ', mae)
print('Mean Squared Error (MSE): ', mse)
print('Root Mean Square Error (RMSE): ', rmse)
print('Mean Absolute Percentage Error: ', MAPE)
print(f"R-squared (R2) Score for Test Set Forecast: {r2}")
```

```
Mean Absolute Error (MAE):  0.05512101759952172
Mean Squared Error (MSE):  0.005779614572542056
Root Mean Square Error (RMSE):  0.0760237763633329
Mean Absolute Percentage Error:  0    10.514219
dtype: float64
R-squared (R2) Score for Test Set Forecast: -0.013754485103370406
```

- The model's MAE and MAPE values suggest that, on average, the model's predictions have a moderate error of approximately 0.055 units or 10.514% relative error from the actual values.These lower values indicate good performance of the model.
- The MSE and RMSE values indicate that the model's predictions have relatively large squared errors, with an RMSE of approximately 0.076, which suggests that the model's errors can be quite significant in magnitude.

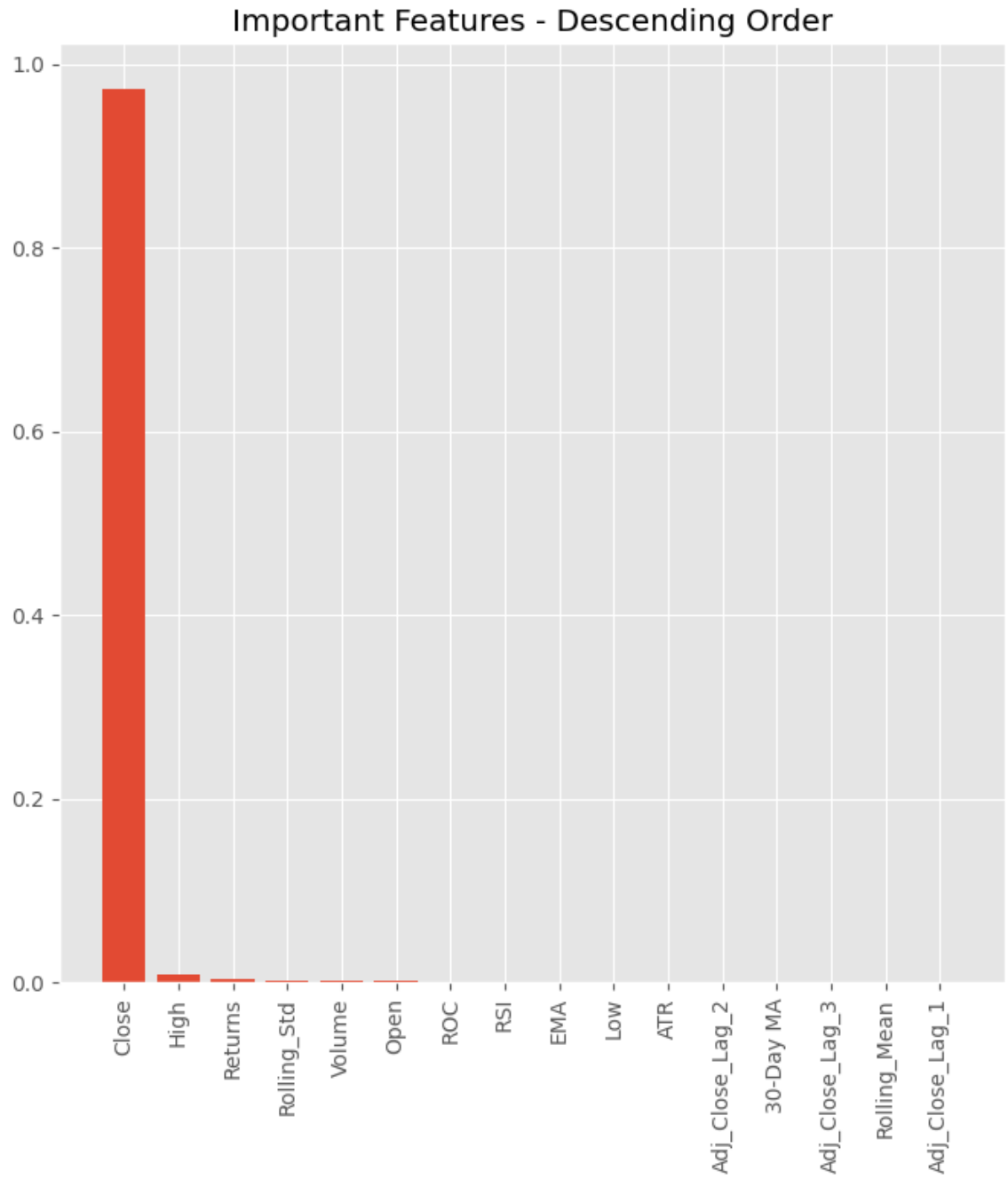
- Therefore, this may not be the best model for making prediction.

XGBOOST - Finding Important Features

XGBoost's feature importance is used here to provide a ranking of features based on their influence in the model's predictions, aiding in understanding and optimizing feature selection.

In [88]:
```
# Function to split the data into training and test sets based on feature im
def get_feature_importance_data(main_data_diff_df):
    y = main_data_diff_df['Adj Close']# target variable
    X = main_data_diff_df.drop('Adj Close', axis=1)  # Drop the target colum

    train_samples = int(X.shape[0] * 0.65) # Calculating the number of trair


    # Splitting the dataset into training and test sets
    X_train = X.iloc[:train_samples]
    X_test = X.iloc[train_samples:]

    y_train = y.iloc[:train_samples]
    y_test = y.iloc[train_samples:]

    return (X_train, y_train), (X_test, y_test)

# Using the function
(X_train, y_train), (X_test, y_test) = get_feature_importance_data(main_data

# Define and train the regressor
regressor = xgb.XGBRegressor(gamma=0.0, n_estimators=200, base_score=0.7, co
regressor.fit(X_train, y_train)

# Predict on the test set and evaluate
y_pred = regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on Test Set: {mse}")
print()

# Sort the feature importance in descending order
sorted_indices = np.argsort(regressor.feature_importances_)[::-1]

# Print the top 8 columns
print('Top 8 Columns:')
for i in range(8):
    print(X_test.columns[sorted_indices[i]])

# Plot the feature importance in descending order
fig = plt.figure(figsize=(8,8))
plt.xticks(rotation='vertical')
plt.bar([i for i in range(len(regressor.feature_importances_))], regressor.f
plt.title('Important Features - Descending Order')
plt.show()
```

```
Mean Squared Error on Test Set: 0.00028977215858672507

Top 8 Columns:
Close
High
Returns
Rolling_Std
Volume
Open
ROC
RSI
```



The model identified the top eight features as being crucial to its predictions. Among these, the primary stock metrics 'Close', 'High', 'Low', and 'Open' were especially influential. In addition, calculated attributes like past adjusted close values and the rolling mean were also deemed significant.

4. RNN

A Simple Recurrent Neural Network (RNN) is a foundational deep learning architecture designed for processing sequential data. By maintaining a memory of previous inputs, RNNs are particularly suited for tasks like time series analysis. In this section, we'll explore the basic structure and application of a Simple RNN

In [89]:
```
main_data_diff_df.head()
```

Out[89]:

date	Open	High	Low	Close	Adj Close	Volume	Returns	Adj_Close_Lag_1
2016-01-04	0.431953	0.424719	0.503978	0.543504	0.543504	0.176738	0.500956	0.516450
2016-01-05	0.427464	0.405198	0.531202	0.536907	0.536907	0.454484	0.527626	0.543504
2016-01-06	0.510061	0.487057	0.482942	0.503381	0.503381	0.655777	0.545968	0.536907
2016-01-07	0.468688	0.401680	0.498920	0.499775	0.499775	0.390733	0.525588	0.503381
2016-01-08	0.390251	0.386736	0.456337	0.514937	0.514937	0.425651	0.512804	0.499775

Model Architecture:

Simple RNN (50 neurons, ReLU activation): Processes stock price sequences, remembering previous inputs.

Dropout Layer: Reduces overfitting by randomly setting input units to 0 during training.

Simple RNN (50 neurons): Further processes the sequential data.

Dense Layer (1 neuron): Outputs predicted stock price.

Compilation:

Optimizer: Adam (Adjusts learning rate during training). Loss Function: Mean Squared Error (MSE).

Training Callbacks:

early_stopping: Stops training if no validation improvement after specified epochs.

reduce_lr: Decreases learning rate when validation loss plateaus.

In [90]:

```
# --- DATA PREPROCESSING ---)

# Scaling the data to fit between 0 and 1. This helps in improving the train
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(main_data_diff_df['Adj Close'].values.res

def transform_series(series, window_size):
    """Transforms the series data into sequences of length 'window_size' to
    X, y = [], []
    for i in range(len(series) - window_size):
        X.append(series[i:i+window_size])
        y.append(series[i+window_size])
    return np.array(X), np.array(y)

window_size = 5
X, y = transform_series(scaled_data, window_size)

# Splitting data into train and test sets. 80% of the data is used for train
train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# --- RNN MODEL DEFINITION AND TRAINING ---

# Defining a simple RNN model with dropout for regularization.
model_rnn = tf.keras.models.Sequential([
    tf.keras.layers.SimpleRNN(50, activation='relu', input_shape=(window_siz
    tf.keras.layers.Dropout(0.2),  # Dropout layer to prevent overfitting.
    tf.keras.layers.SimpleRNN(50, activation='relu'),
    tf.keras.layers.Dense(1)
])

model_rnn.compile(optimizer='adam', loss='mean_squared_error')

# Implementing early stopping. This stops training once the validation loss
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patier

# Implementing ReduceLROnPlateau to reduce learning rate when validation los
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=

model_rnn.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0


# --- PREDICTION AND RESCALING ---

# Predicting values for the test set.
y_pred = model_rnn.predict(X_test)

# Rescaling the predicted and actual values back to their original range.
y_pred_rescaled = scaler.inverse_transform(y_pred)
y_test_rescaled = scaler.inverse_transform(y_test)

# --- PREDICT FUTURE VALUES ---

# Predicting values for a future period using the trained model.
future_predictions = []
current_batch = scaled_data[-window_size:].reshape((1, window_size, 1))
number_of_future_predictions = 100

for i in range(number_of_future_predictions):
    current_prediction = model_rnn.predict(current_batch)[0]
```

```
    future_predictions.append(current_prediction)
    current_batch = np.append(current_batch[:,1:,:], [[current_prediction]],

future_predictions_rescaled = scaler.inverse_transform(future_predictions)

# Generate future dates for plotting.
last_date = main_data.index[-1]
future_dates = pd.date_range(last_date, periods=number_of_future_predictions

# --- PLOTTING RESULTS WITH DATES ON X-AXIS ---

# Plot the actual, predicted, and future values.
test_dates = main_data.index[train_size:train_size + len(X_test)]

plt.figure(figsize=(15,6))
plt.plot(test_dates, y_test_rescaled, label='True Test Values')
plt.plot(test_dates, y_pred_rescaled, label='Predicted Test Values')
plt.plot(future_dates, future_predictions_rescaled, label='Future Prediction
plt.legend()
plt.show()
```

```
Epoch 1/50
44/44 [==============================] - 3s 15ms/step - loss: 0.0263 - v
al_loss: 0.0106 - lr: 0.0010
Epoch 2/50
44/44 [==============================] - 0s 6ms/step - loss: 0.0070 - va
l_loss: 0.0108 - lr: 0.0010
Epoch 3/50
44/44 [==============================] - 0s 7ms/step - loss: 0.0064 - va
l_loss: 0.0101 - lr: 0.0010
Epoch 4/50
44/44 [==============================] - 0s 6ms/step - loss: 0.0062 - va
l_loss: 0.0100 - lr: 0.0010
Epoch 5/50
44/44 [==============================] - 0s 7ms/step - loss: 0.0057 - va
l_loss: 0.0098 - lr: 0.0010
Epoch 6/50
44/44 [==============================] - 1s 14ms/step - loss: 0.0056 - v
al_loss: 0.0101 - lr: 0.0010
Epoch 7/50
```

i. RNN Metrics

In [91]:
```
# Calculate Mean Squared Error (MSE)
mae = mean_absolute_error(y_test_rescaled, y_pred_rescaled)

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_test_rescaled, y_pred_rescaled)

# Calculate Root Mean Squared Error (RMSE)
rmse = np.sqrt(mse)

# Calculate r2- Score
r2 = r2_score(y_test_rescaled, y_pred_rescaled)

z = pd.DataFrame(np.abs((y_test_rescaled - y_pred_rescaled)/y_test_rescaled)
MAPE = z.mean() * 100

print('Mean Absolute Error', mae)
print('Mean Squared Error (MSE):', mse)
print('Root Mean Squared Error (RMSE):', rmse)
print('Mean Absolute Percentage Error:', MAPE)
print('R-squared (R2) Score for Test Set Forecast:', r2)
```

```
Mean Absolute Error 0.07780980613635775
Mean Squared Error (MSE): 0.010554588300537642
Root Mean Squared Error (RMSE): 0.10273552599046565
Mean Absolute Percentage Error: 0    15.045526
dtype: float64
R-squared (R2) Score for Test Set Forecast: -0.036014179158752935
```

Observation

- The model's MAE and MAPE values suggest that, on average, the model's predictions have a moderate error of approximately 0.078 units or 14.778% relative error from the actual values.These lower values indicate low performance of the model. -The MSE and RMSE values indicate that the model's predictions have relatively large squared errors, with an RMSE of approximately 0.103,and a negative value of R2-Score, which suggests that the model's errors can be quite significant in magnitude. -Therefore, this may not be the best model for making prediction.

5. LSTM

a. LSTM with Original Features

In [92]:
```
new_main_data = main_data_diff_df[['Open','High', 'Low', 'Close', 'Adj Close
new_main_data.head()
```

Out[92]:

date	Open	High	Low	Close	Adj Close	Volume
2016-01-04	0.431953	0.424719	0.503978	0.543504	0.543504	0.176738
2016-01-05	0.427464	0.405198	0.531202	0.536907	0.536907	0.454484
2016-01-06	0.510061	0.487057	0.482942	0.503381	0.503381	0.655777
2016-01-07	0.468688	0.401680	0.498920	0.499775	0.499775	0.390733
2016-01-08	0.390251	0.386736	0.456337	0.514937	0.514937	0.425651

In [93]:
```
look_back = 20
def processData(data, look_back):
    X,y = [],[]
    for i in range(look_back,len(data)):
        #X will contain 5 features from day i-N to day i-1
        #y will be closing price on day i
        X.append(data[i-look_back:i].values)
        y.append(data.iat[i,0])
        #Feature scaling
    X= np.array(X)
    y=np.array(y)
    scl = MinMaxScaler()

    y = scl.fit_transform(y.reshape(-1,1))
    for i in range(X.shape[2]):
        X[:,:,i] = scl.fit_transform(X[:,:,i])

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    return X_train,X_test,y_train,y_test
```

Now we create a function called processData() which preprocess time series data which we will use in our LSTM neural network. This function takes two arguments: the input data and the number of time steps to look back. After that the function returns four arrays: the training and testing input data (X_train and X_test) and training and testing output data (y_train and y_test).

In [94]:
```
def get_accuracy(y_pred,y_test):
    #Constructing predicted movement
    movement_pred = pd.DataFrame(y_pred) - pd.DataFrame(y_pred).shift()
    movement_pred.dropna(inplace = True)
    for i in range(len(movement_pred)):
        if movement_pred.iat[i,0] > 0:
            movement_pred.iat[i,0] = 1
        else:
            movement_pred.iat[i,0] = 0
    #Constructing groundtruth movement
    movement_gt = pd.DataFrame(y_test) - pd.DataFrame(y_test).shift()
    movement_gt.dropna(inplace = True)
    for i in range(len(movement_gt)):
        if movement_gt.iat[i,0] > 0:
            movement_gt.iat[i,0] = 1
        else:
            movement_gt.iat[i,0] = 0
    #Make a comparison then calculate accuracy
    movement = (movement_pred==movement_gt)
    accuracy = np.unique(movement, return_counts=True)[1][1] / len(movement)

    #Calculating RMSE and MAPE
    y_test = np.array(y_test)
    y_pred = np.array(y_pred)
    z = pd.DataFrame(np.abs((y_test - y_pred)/y_test)).replace(np.inf, np.na
    MAPE = z.mean() * 100

    MAE = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_pred, y_test)
    RMSE = math.sqrt(mean_squared_error(y_pred,y_test))
    r2 = r2_score(y_test, y_pred)
    return accuracy, MAPE, mae, mse, RMSE, r2
```

Now we create a function called get_accuracy(), this calculates the accuracy, MAPE, RSME of our LSTM neural network model. This function takes in two arguments: the predicted output data (y_pred) and the actual output data (y_test), this will return us three values: the accuracy, MAPE and RMSE.

In [95]:
```
weight_decay = 1e-3
from keras.regularizers import l2
from keras import regularizers, initializers, optimizers
```

Weight decay is a regularization technique which is used to prevent overfitting of machine learning models. The keras.regularizers module provide various regularisation techniques such as L1 and L2 regularisation, the keras.initializers module provides various initialisation techniques for weights and biases of the model, and the keras.

i. Constructing LSTM Network

In [96]:
```
input_layer = Input(shape=(look_back, 6))
x = LSTM(128,return_sequences=True,kernel_regularizer=l2(weight_decay),kerne
x = Dropout(0.5)(x)
x = LSTM(32,return_sequences = False,kernel_regularizer=l2(weight_decay),ker
x = Dropout(0.5)(x)
x = Dense(16,activation='relu')(x)
output_layer = Dense(1,activation='linear')(x)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(optimizer='rmsprop', loss='mean_squared_error')
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 20, 6)]           0

 lstm (LSTM)                 (None, 20, 128)           69120

 dropout_1 (Dropout)         (None, 20, 128)           0

 lstm_1 (LSTM)               (None, 32)                20608

 dropout_2 (Dropout)         (None, 32)                0

 dense_1 (Dense)             (None, 16)                528

 dense_2 (Dense)             (None, 1)                 17

=================================================================
Total params: 90273 (352.63 KB)
Trainable params: 90273 (352.63 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Here we define our neural network architecture of our model using the keras library. First we define the input layer of our model. We use the input() function of keras library. The shape of our input layers is set to (look_back, 5) which means that the model will take in a sequences of 5 features or columns for each time step.

The dense layer is used to add nonlinearity to the model and improve its performance. We compile the model using compile() function.

ii. Employing Reduce Learning Rate Strategy

We will employ a reduce learning rate strategy for training our LSTM neural network. First we need to import ReduceLROnPlateau() function from keras library. This will be used to reduce the learning rate of the optimizer when the validation loss of the model stops improving. In this code the monitor parameter is set to val_loss, this means that the validation loss of the model will be monitored. The factor parameter is set to 0.1, this means that the learning rate will be reduced by a factor of 0.1 when the validation loss stops improving.

In [97]:
```
#Employing reduce learning rate strategy
from keras.callbacks import ReduceLROnPlateau
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=3, ve
```

The min_lr parameter is set to 1e-5, which is the minimum learning rate that the optimizer can reach.

iii. Pre-processing the Time Series

We will preprocess time series data for use in our LSTM neural network.

In [98]:
```
X_train,X_test,y_train,y_test = processData(new_main_data,look_back)
```

In [102]:
```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(1546, 20, 6)
(1546, 1)
(387, 20, 6)
(387, 1)
```

iv. Train the LSTM Model

- We train our LSTM network using the fit() function from the keras library.
- This takes in the training input data which is `X_train` and output data which is `y_train` and sets the batch size, number of epochs, validation split, and callbacks for the model.
- The batch_size parameter is set to 16, epoch to 30, and validation_split to 0.3.

In [103]:
```
model.fit(X_train,y_train,batch_size = 16,epochs = 30,validation_split=0.3,c
```

```
Epoch 1/30
68/68 [==============================] - 2s 32ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 2/30
68/68 [==============================] - 2s 36ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 3/30
68/68 [==============================] - 3s 37ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 4/30
68/68 [==============================] - 2s 29ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 5/30
68/68 [==============================] - 2s 30ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 6/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 7/30
68/68 [==============================] - 2s 29ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 8/30
68/68 [==============================] - 2s 33ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 9/30
68/68 [==============================] - 3s 38ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 10/30
68/68 [==============================] - 2s 29ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 11/30
68/68 [==============================] - 2s 27ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 12/30
68/68 [==============================] - 2s 27ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 13/30
68/68 [==============================] - 2s 27ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 14/30
68/68 [==============================] - 2s 32ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 15/30
68/68 [==============================] - 3s 42ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 16/30
68/68 [==============================] - 2s 30ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 17/30
68/68 [==============================] - 3s 38ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 18/30
68/68 [==============================] - 3s 46ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 19/30
68/68 [==============================] - 2s 30ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 20/30
68/68 [==============================] - 3s 39ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 21/30
```

```
68/68 [==============================] - 2s 28ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 22/30
68/68 [==============================] - 2s 27ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 23/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 24/30
68/68 [==============================] - 2s 29ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 25/30
68/68 [==============================] - 2s 31ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 26/30
68/68 [==============================] - 3s 43ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 27/30
68/68 [==============================] - 2s 29ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 28/30
68/68 [==============================] - 2s 30ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 29/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
Epoch 30/30
68/68 [==============================] - 2s 30ms/step - loss: 0.0042 - val
_loss: 0.0120 - lr: 1.0000e-05
```

Out[103]:
```
<keras.src.callbacks.History at 0x7e4e4cbe52a0>
```

v. Predict on Test Data

- In our case the predicted output data is the adjusted closing price of the stock on the next day, which is what out neural network was trained to predict.

In [104]:
```
y_pred = model.predict(X_test)
```

```
13/13 [==============================] - 1s 21ms/step
```

vi. LSTM Metrics (Using Original Features)

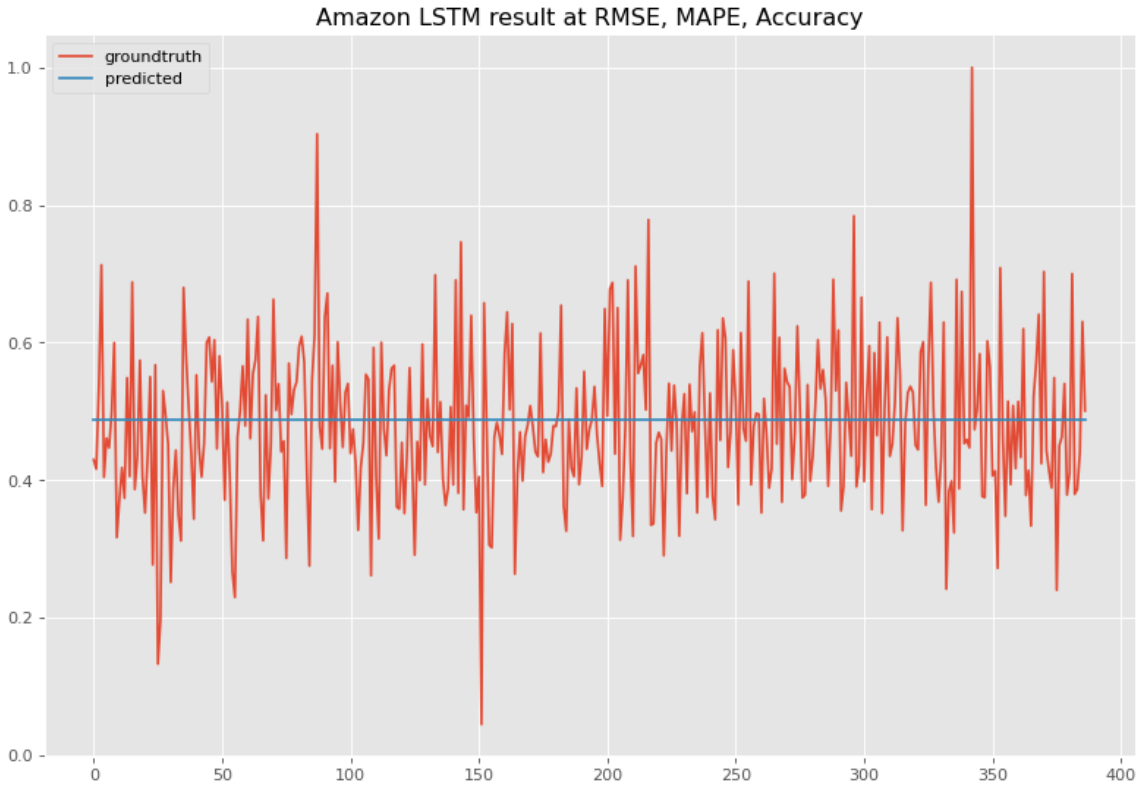
In [105]:
```
# Calling the function
accuracy, MAPE, mae, mse, RMSE, r2 = get_accuracy(y_pred,y_test)

# Printing the metrics
print('Accuracy:', accuracy)
print('MAE:', mae)
print('MSE:', mse)
print('RMSE:', RMSE)
print('MAPE:', MAPE)
print('r2:', r2)
```

```
Accuracy: 0.6062176165803109
MAE: 0.07780980613635775
MSE: 0.013296664402941806
RMSE: 0.11531116339254326
MAPE: 0    23.145292
dtype: float64
r2: -0.002871461338796344
```

Observations

- The model's MAE and MAPE values suggest that, on average, the model's predictions have a moderate error of approximately 0.078 units or 23.117% relative error from the actual values.This higher value of MAPE indicate low performance of the model. -The MSE and RMSE values indicate that the model's predictions have relatively large squared errors, with an RMSE of approximately 0.115,and a negative value of R2-Score, which suggests that the model's errors can be quite significant in magnitude. -Therefore, this may not be the best model for making prediction.

In [106]:
```
plt.figure(figsize=(12, 8), dpi=80)
plt.title('Amazon LSTM result at RMSE, MAPE, Accuracy')
plt.plot(y_test,label = 'groundtruth')
plt.plot(y_pred,label='predicted')
plt.legend();
```



b. LSTM with Important Features from XGBoost

The top 8 columns identified are:

- Close
- High
- Returns
- Rolling_Std
- Volume
- Open
- ROC
- RSI

In [107]:
```
# new df for the model
new_main_data_2 = main_data_diff_df[['Close', 'High', 'Returns', 'Rolling_St
new_main_data_2.head()
```

Out[107]:

date	Close	High	Returns	Rolling_Std	Volume	Open	ROC	RSI	
2016-01-04	0.543504	0.424719	0.500956	0.581436	0.176738	0.431953	0.411904	0.426493	0.54
2016-01-05	0.536907	0.405198	0.527626	0.512660	0.454484	0.427464	0.473835	0.449317	0.53
2016-01-06	0.503381	0.487057	0.545968	0.518774	0.655777	0.510061	0.551887	0.456892	0.50
2016-01-07	0.499775	0.401680	0.525588	0.510277	0.390733	0.468688	0.442362	0.379432	0.49
2016-01-08	0.514937	0.386736	0.512804	0.494565	0.425651	0.390251	0.342704	0.365815	0.51

i. Constructing LSTM Network

In [108]:
```
input_layer = Input(shape=(look_back, 9))
x = LSTM(128,return_sequences=True,kernel_regularizer=l2(weight_decay),kerne
x = Dropout(0.5)(x)
x = LSTM(32,return_sequences = False,kernel_regularizer=l2(weight_decay),ker
x = Dropout(0.5)(x)
x = Dense(16,activation='relu')(x)
output_layer = Dense(1,activation='linear')(x)
model2 = Model(inputs=input_layer, outputs=output_layer)
model2.compile(optimizer='rmsprop', loss='mean_squared_error')
model2.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 20, 9)]           0

 lstm_2 (LSTM)               (None, 20, 128)           70656

 dropout_3 (Dropout)         (None, 20, 128)           0

 lstm_3 (LSTM)               (None, 32)                20608

 dropout_4 (Dropout)         (None, 32)                0

 dense_3 (Dense)             (None, 16)                528

 dense_4 (Dense)             (None, 1)                 17

=================================================================
Total params: 91809 (358.63 KB)
Trainable params: 91809 (358.63 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

ii. Pre-processing the Time Series and Train LSTM Model on Important Features

In [109]:
```
X_train_2, X_test_2, y_train_2, y_test_2 = processData(new_main_data_2, look
```

In [110]:
```
print(X_train_2.shape)
print(y_train_2.shape)
print(X_test_2.shape)
print(y_test_2.shape)
```

```
(1546, 20, 9)
(1546, 1)
(387, 20, 9)
(387, 1)
```

In [111]:
```
model2.fit(X_train_2, y_train_2, batch_size = 16,epochs = 30, validation_spl
```

```
Epoch 1/30
68/68 [==============================] - 7s 43ms/step - loss: 1.2260 - val
_loss: 1.0690 - lr: 0.0010
Epoch 2/30
68/68 [==============================] - 2s 36ms/step - loss: 0.9434 - val
_loss: 0.8162 - lr: 0.0010
Epoch 3/30
68/68 [==============================] - 3s 37ms/step - loss: 0.7167 - val
_loss: 0.6199 - lr: 0.0010
Epoch 4/30
68/68 [==============================] - 2s 34ms/step - loss: 0.5358 - val
_loss: 0.4595 - lr: 0.0010
Epoch 5/30
68/68 [==============================] - 2s 30ms/step - loss: 0.3954 - val
_loss: 0.3404 - lr: 0.0010
Epoch 6/30
68/68 [==============================] - 2s 28ms/step - loss: 0.2868 - val
_loss: 0.2444 - lr: 0.0010
Epoch 7/30
68/68 [==============================] - 2s 28ms/step - loss: 0.2053 - val
_loss: 0.1752 - lr: 0.0010
Epoch 8/30
68/68 [==============================] - 3s 38ms/step - loss: 0.1441 - val
_loss: 0.1226 - lr: 0.0010
Epoch 9/30
68/68 [==============================] - 2s 32ms/step - loss: 0.0987 - val
_loss: 0.0849 - lr: 0.0010
Epoch 10/30
68/68 [==============================] - 2s 27ms/step - loss: 0.0660 - val
_loss: 0.0575 - lr: 0.0010
Epoch 11/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0430 - val
_loss: 0.0384 - lr: 0.0010
Epoch 12/30
68/68 [==============================] - 2s 29ms/step - loss: 0.0273 - val
_loss: 0.0259 - lr: 0.0010
Epoch 13/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0171 - val
_loss: 0.0179 - lr: 0.0010
Epoch 14/30
68/68 [==============================] - 2s 36ms/step - loss: 0.0108 - val
_loss: 0.0132 - lr: 0.0010
Epoch 15/30
68/68 [==============================] - 2s 36ms/step - loss: 0.0071 - val
_loss: 0.0107 - lr: 0.0010
Epoch 16/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0052 - val
_loss: 0.0091 - lr: 0.0010
Epoch 17/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0041 - val
_loss: 0.0084 - lr: 0.0010
Epoch 18/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0036 - val
_loss: 0.0082 - lr: 0.0010
Epoch 19/30
68/68 [==============================] - 2s 30ms/step - loss: 0.0034 - val
_loss: 0.0082 - lr: 0.0010
Epoch 20/30
68/68 [==============================] - 2s 35ms/step - loss: 0.0033 - val
_loss: 0.0079 - lr: 0.0010
Epoch 21/30
```

```
68/68 [==============================] - 3s 38ms/step - loss: 0.0033 - val
_loss: 0.0081 - lr: 0.0010
Epoch 22/30
68/68 [==============================] - 2s 30ms/step - loss: 0.0032 - val
_loss: 0.0079 - lr: 0.0010
Epoch 23/30
68/68 [==============================] - 2s 31ms/step - loss: 0.0033 - val
_loss: 0.0079 - lr: 0.0010
Epoch 24/30
68/68 [==============================] - 2s 30ms/step - loss: 0.0032 - val
_loss: 0.0079 - lr: 1.0000e-04
Epoch 25/30
68/68 [==============================] - 2s 29ms/step - loss: 0.0032 - val
_loss: 0.0079 - lr: 1.0000e-04
Epoch 26/30
68/68 [==============================] - 3s 41ms/step - loss: 0.0032 - val
_loss: 0.0079 - lr: 1.0000e-04
Epoch 27/30
68/68 [==============================] - 2s 31ms/step - loss: 0.0032 - val
_loss: 0.0079 - lr: 1.0000e-05
Epoch 28/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0032 - val
_loss: 0.0079 - lr: 1.0000e-05
Epoch 29/30
68/68 [==============================] - 2s 28ms/step - loss: 0.0032 - val
_loss: 0.0079 - lr: 1.0000e-05
Epoch 30/30
68/68 [==============================] - 2s 30ms/step - loss: 0.0032 - val
_loss: 0.0079 - lr: 1.0000e-05
```

Out[111]:
```
<keras.src.callbacks.History at 0x7e4e45e312a0>
```

iii. Predict on Test Data

In [112]:
```
y_pred_2 = model2.predict(X_test_2)
```

```
13/13 [==============================] - 1s 13ms/step
```

iv. LSTM Metrics (Using Important Features)

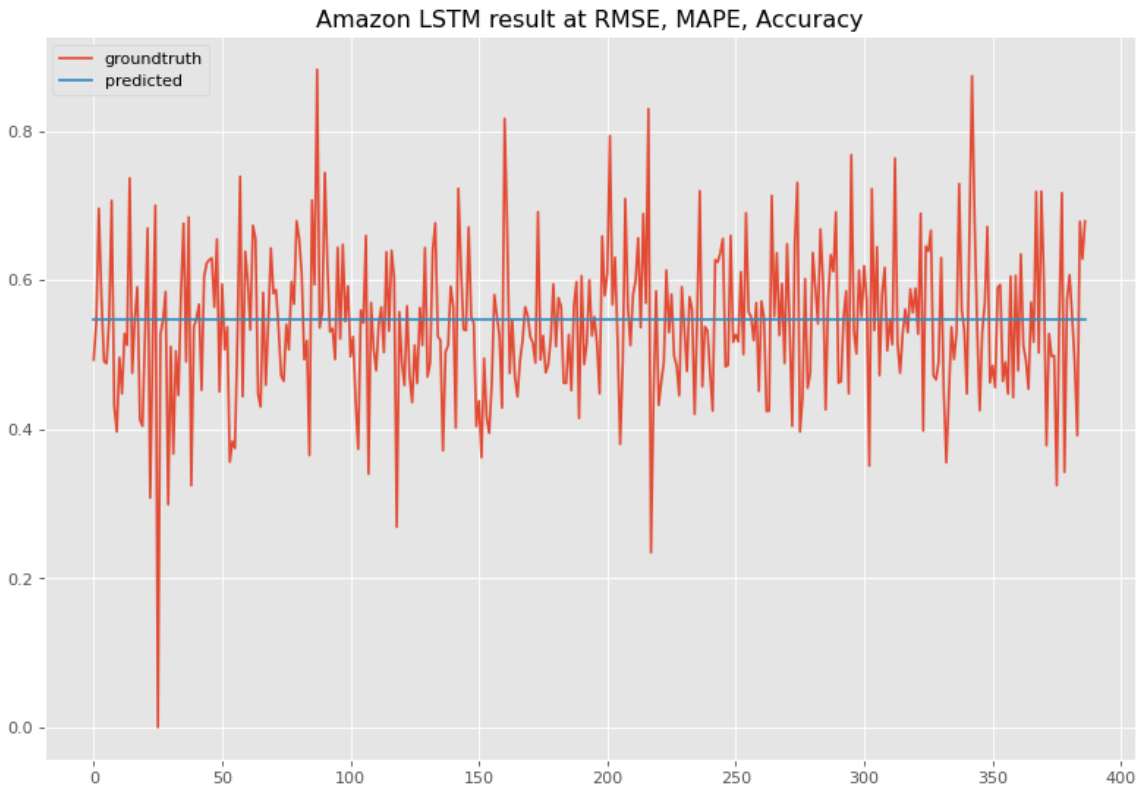
In [113]:
```
# Calling the function
accuracy, MAPE, mae, mse, RMSE, r2 = get_accuracy(y_pred_2,y_test_2)

# Printing the metrics
print('Accuracy:', accuracy)
print('MAE:', mae)
print('MSE:', mse)
print('RMSE:', RMSE)
print('MAPE:', MAPE)
print('r2:', r2)
```

```
Accuracy: 0.5051813471502591
MAE: 0.07780980613635775
MSE: 0.010263735894362254
RMSE: 0.10131009769199838
MAPE: 0    14.951426
dtype: float64
r2: -0.0035725205138810168
```

Observations

- The model's MAE and MAPE values suggest that, on average, the model's predictions have a moderate error of approximately 0.078 units or 14.954% relative error from the actual values.These values indicate low performance of the model. -The MSE and RMSE values indicate that the model's predictions have relatively large squared errors, with an RMSE of approximately 0.101,and a negative value of R2-Score, which suggests that the model's performance is lower as compared to SARIMA-2, RNN and FB Prophet models. -Therefore, this may not be the best model for making prediction.

In [114]:
```
plt.figure(figsize=(12, 8), dpi=80)
plt.title('Amazon LSTM result at RMSE, MAPE, Accuracy')
plt.plot(y_test_2,label = 'groundtruth')
plt.plot(y_pred_2,label='predicted')
plt.legend();
```



E. Evaluation of Models

In [115]:
```
# create the summary df and define columns
scores = pd.DataFrame(np.array([
        ['Baseline Model-SARIMA-1', 0.0506, 0.0047, 0.0683, 9.6815, 0.0016],
        ['SARIMA-2', 0.0506, 0.0047, 0.0683, 9.6815,  0.0025],
        ['FB Prophet', 0.0551, 0.0058, 0.0760, 10.5142, -0.0138],
        ['Simple RNN', 0.0779, 0.0107, 0.1033, 14.7780, -0.0471],
        ['LSTM-Original Features', 0.0779, 0.0133, 0.1153, 23.1175, -0.0025]
        ['LSTM-Important Features', 0.0779, 0.0103, 0.1013, 14.9543, -0.0036
]))
scores.columns = ['Model', 'MAE', 'MSE', 'RMSE', 'MAPE', 'R2-Score']

scores # scaled DataFrame (main_data_diff_df)
```

Out[115]:

	Model	MAE	MSE	RMSE	MAPE	R2-Score
0	Baseline Model-SARIMA-1	0.0506	0.0047	0.0683	9.6815	0.0016
1	SARIMA-2	0.0506	0.0047	0.0683	9.6815	0.0025
2	FB Prophet	0.0551	0.0058	0.076	10.5142	-0.0138
3	Simple RNN	0.0779	0.0107	0.1033	14.778	-0.0471
4	LSTM-Original Features	0.0779	0.0133	0.1153	23.1175	-0.0025
5	LSTM-Important Features	0.0779	0.0103	0.1013	14.9543	-0.0036

- **SARIMA-2** appears to be the best-performing model among the options listed. It has the highest R2-Score of 0.0025, and the lowest MAE, MSE, and RMSE, indicating that it provides the most accurate predictions with relatively low error rates. The MAPE is also below 10%, suggesting that, on average, its predictions are within 10% of the actual Amazon stock prices.
- **FB Prophet**, while having a low MAE, MSE, and RMSE but the values are higher as compared to SARIMA-2. It also has higher MAPE which is above 10%, and a negative value of R-2 Score. This indicates that it might not be as accurate as SARIMA-2 in predicting Amazon stock prices.
- The **Simple RNN** model performs lower than FB Prophet in terms of MAE, MSE, and RMSE, it is not as accurate as SARIMA-2 and FB Prophet, as it has higher error rates.
- The **LSTM with original features** model with original features has lowest performance as compared to SARIMA-2, FB Prophet and Simple RNN, with slightly higher MAE, MAPE and RMSE. It has the highest MAPE of 23.1175%.
- The **LSTM model with important features** has a better performance to the LSTM with original features and Simple RNN. However, it still does not perform as well as SARIMA-2 and FB Prophet in terms of accuracy.

In summary, the **SARIMA-2** model outperforms the other models in predicting Amazon stock prices, as it has the lowest MAE, MSE, and RMSE along with a reasonably low MAPE. It is also recorded the highest positive R-Squared Score value. The other models have higher errors and are less accurate in comparison.

Saving the best Model

In [121]:
```
import pickle
```

In [122]:
```
model_file = 'sarima_model.sav'
with open(model_file, 'wb') as model_file:
    pickle.dump(sarima_model, model_file)
```

In [119]:
```
# # sarima_orig_model

# model2_file = 'sarima_model2.sav'
# with open(model2_file, 'wb') as model2_file:
#     pickle.dump(sarima_orig_model, model2_file)
```

F. Post-Processing

i. De-scaling

In [123]:
```
post_data_df = pd.read_csv('sarima_predictions2.csv', index_col='Datetime')
post_data_df.head()
```

Out[123]:

Datetime	Predicted_Adj_Close
2023-10-09	0.540623
2023-10-10	0.539334
2023-10-11	0.553312
2023-10-12	0.543201
2023-10-13	0.551204

In [124]:
```
len(post_data_df)
```

Out[124]: 521

In [125]:
```
post_data_df.shape
```

Out[125]: (521, 1)

In [126]:
```
# Load the SARIMA predictions
sarima_predictions_df = pd.read_csv('sarima_predictions.csv', index_col='Dat

# Get the scaling factor
scaler = MinMaxScaler()
scaler.fit(sarima_predictions_df[['Predicted_Adj_Close']])
scaling_factor = scaler.scale_[0]

# Inverse the scaling transform
sarima_predictions_df['Predicted_Adj_Close'] = sarima_predictions_df['Predic

# Save the actual adjusted close values
sarima_predictions_df.to_csv('adjusted_close_predictions.csv', index=True)
```

In [127]:
```
# Create a MinMaxScaler for the numerical data
min_max_scaler = MinMaxScaler()

# Fit the scaler on the numerical data
numerical_data = post_data_df[['Predicted_Adj_Close']]
min_max_scaler.fit(numerical_data)

# Transform the numerical data
scaled_data = min_max_scaler.transform(numerical_data)
post_data_df['scaled_predictions'] = scaled_data

# Inverse the scaling transform
sarima_predictions_df['Predicted_Adj_Close'] = sarima_predictions_df['Predic

# Save the actual adjusted close values
sarima_predictions_df.to_csv('adjusted_close_predictions.csv', index=True)
```

In [128]:
```
descaled_predictions = pd.read_csv('adjusted_close_predictions.csv')
descaled_predictions
```

Out[128]:

	Datetime	Predicted_Adj_Close
0	2023-10-09	177.183017
1	2023-10-10	170.266142
2	2023-10-11	179.471366
3	2023-10-12	185.317009
4	2023-10-13	180.660095
...	...	...
516	2025-09-30	175.595504
517	2025-10-01	177.382662
518	2025-10-02	182.737904
519	2025-10-03	178.626984
520	2025-10-06	181.783151

521 rows × 2 columns

ii. Stationarizing

In [129]:
```
original_values = np.cumsum(sarima_predictions_df['Predicted_Adj_Close'], ax
sarima_predictions_df['Predicted_Adj_Close'] = original_values
sarima_predictions_df.head(10)
```

Out[129]:

Datetime	Predicted_Adj_Close
2023-10-09	177.183017
2023-10-10	347.449160
2023-10-11	526.920526
2023-10-12	712.237535
2023-10-13	892.897629
2023-10-16	1071.328928
2023-10-17	1242.760581
2023-10-18	1421.096216
2023-10-19	1606.887323
2023-10-20	1785.073345

In [130]:
```
original_values = np.cumsum(sarima_predictions_df['Predicted_Adj_Close'], ax
sarima_predictions_df['Predicted_Adj_Close'] = original_values
sarima_predictions_df.head(10)
```

Out[130]:

Datetime	Predicted_Adj_Close
2023-10-09	177.183017
2023-10-10	524.632177
2023-10-11	1051.552703
2023-10-12	1763.790238
2023-10-13	2656.687867
2023-10-16	3728.016794
2023-10-17	4970.777376
2023-10-18	6391.873592
2023-10-19	7998.760915
2023-10-20	9783.834260

iii. Restoring Seasonality

In [131]:
```
seasonal = seasonal_decompose(sarima_predictions_df['Predicted_Adj_Close'],
seasonal_component = seasonal.seasonal
sarima_predictions_df['Seasonalized_Adj_Close'] = sarima_predictions_df['Pre
```

In [132]:
```
sarima_predictions_df['Seasonalized_Adj_Close']
```

Out[132]:
```
Datetime
2023-10-09    1.329496e+02
2023-10-10    4.723721e+02
2023-10-11    9.920264e+02
2023-10-12    1.701481e+03
2023-10-13    2.594082e+03
                  ...
2025-09-30    2.398973e+07
2025-10-01    2.408255e+07
2025-10-02    2.417555e+07
2025-10-03    2.426874e+07
2025-10-06    2.436210e+07
Name: Seasonalized_Adj_Close, Length: 521, dtype: float64
```

G. Conclusion

From our time series analysis SARIMA-2 model performed the best with MAE score of 0.0506, MSE score of 0.0047 RMSE score of 0.0683 and MAPE score of 9.6815 compared to the other models we used which were:

1. FB Prophet
2. Simple RNN
3. LSTM-original features
4. LSTM-Important features

SARIMA-2 model performed well for short term predictions however long term predictions brought wide variations.

The top 8 features which highly influenced the price predictions in the amazon stock market are: Close price, Highest price, Returns, Rolling_Std, Volume, Open price, Rate Of Change, Relative Strength Index.

##H. Recommendations

1. Investors and financial institutions can use the model for short term prediction of Amazon stock prices to determine the general trend of the amazon prices. However other factor such as fundamental analysis need to be consider before making the final decision.
2. Our deployment model can also be improved and used for predicting other stock markets other than Amazon stocks only.
3. For better performance of the LSTM model, more data is required for analysis. More data will enhance the model's ability to recognize patterns and trends.
4. Carry out sentimental analysis alongside the model to factor in the impact of news and public sentiment on stock prices changes. This analysis will provide valuable contextual information.
5. Experiment with different train-test split ratios to evaluate how the model's performance is affected by the division of data. This will help determine the optimal balance between the training and testing data that gives better performance of the model.
6. Coming up with a model that can predict other stock markets, not just Amazon. This will provide valuable insights from diverse stock markets to the investors.

I. Limitations

1. Time series is an intensive machine learning models and hence it required more time for us to come up with optimum parameter and hyper-parameters for the model to perform much better which was a great constrain.
2. Stock prices are influenced by various external factors, including unforeseen events like wars and diseases/pandemics, which are difficult to predict and can significantly impact market values.