Project: Amazon Stock Data Analysis

Introduction

Stock analysis is a critical process for understanding the dynamics of financial markets and making informed investment decisions. It involves evaluating historical price trends, trading volumes, and key financial indicators to assess a stock's performance and forecast future movements.

By leveraging data-driven insights, stock analysis empowers investors to uncover patterns, evaluate market conditions, and identify potential opportunities or risks.

Key aspects of the stock analysis include the following;

Historical Trend Analysis: *Observing stock price movements over time to identify long-term trends, seasonality, or anomalies.*

Volatility and Risk Assessment: *Measuring the stock's price fluctuations and understanding market behavior during periods of uncertainty.*

Fundamental Analysis: *Examining company-specific metrics such as earnings, revenue growth, and valuation ratios.*

Technical Analysis: *Using chart patterns and indicators like moving averages, RSI, and MACD to predict price movements.*

Sentiment Analysis: *Gauging public and investor sentiment, which often impacts stock prices.*

The Analysis

Anazon being one of the largest tech comapnies in globally, its stock performance often reflects broader market trends while also serving as a bellwether for the e-commerce, cloud computing, and technology sectors.

Analyzing Amazon's stock data offers insights into the following;

How investors react to news, earnings reports, or global economic shifts.

Growth Trajectory such as streaming, logistics, and AI are all in its stock performance.

Volatility Trends

In this project, we look at the Amazon stock data to uncover the following:

Historical Price Trends: Understanding how Amazon's stock has performed over time.

Volume Analysis: Studying trading volumes to assess periods of heightened activity.

Volatility and Moving Averages: Evaluating Amazon's price fluctuations and applying technical indicators to identify support and resistance levels.

Seasonality and Key Events: Analyzing seasonal trends and the impact of significant announcements on stock price movements.

Forecasting Stock Performance: Using machine learning models like ARIMA to predict Amazon's future stock prices.

Importing the necessary datasets

The 'yfinance'library provides the stock market data necessary for the analysis. With it, you can download Amazon's historical stock data and key metrics to explore trends and build forecasts.

```
import pandas as pd
import numpy as np
import yfinance as yf
```

In [1]:
```
import pandas as pd
import numpy as np
import yfinance as yf
```

The yf.download() function retrieves historical stock data for Amazon from Yahoo Finance.

The data includes key metrics like Open, High, Low, Close, Adj Close, and Volume for each trading day.

In [3]:
```
# Fetchning the Amazon data using the Yahoo Finance API

# Fetch AMZN stock data
ticker = "AMZN"
data = yf.download(ticker, start="2010-01-01", end="2024-12-31")
```

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

Displaying the some of the first rows of the data and saving it as a CSV file

In [5]:
```
# Display the first few rows of the dataset
print(data.head())

# Save the dataset for future use
data.to_csv("PG_stock_data.csv")
```

```
Price      Adj Close   Close    High     Low    Open     Volume
Ticker          AMZN    AMZN    AMZN    AMZN    AMZN       AMZN
Date
2010-01-04    6.6950  6.6950  6.8305  6.6570  6.8125  151998000
2010-01-05    6.7345  6.7345  6.7740  6.5905  6.6715  177038000
2010-01-06    6.6125  6.6125  6.7365  6.5825  6.7300  143576000
2010-01-07    6.5000  6.5000  6.6160  6.4400  6.6005  220604000
2010-01-08    6.6760  6.6760  6.6840  6.4515  6.5280  196610000
```

In [7]:
```
data
```

Out[7]:

Price	Adj Close	Close	High	Low	Open	Volume
Ticker	AMZN	AMZN	AMZN	AMZN	AMZN	AMZN
Date						
2010-01-04	6.695000	6.695000	6.830500	6.657000	6.812500	151998000
2010-01-05	6.734500	6.734500	6.774000	6.590500	6.671500	177038000
2010-01-06	6.612500	6.612500	6.736500	6.582500	6.730000	143576000
2010-01-07	6.500000	6.500000	6.616000	6.440000	6.600500	220604000
2010-01-08	6.676000	6.676000	6.684000	6.451500	6.528000	196610000
...	...	...	...	...	...	...
2024-12-09	226.089996	226.089996	230.080002	225.669998	227.210007	46819400
2024-12-10	225.039993	225.039993	229.059998	224.199997	226.089996	31199900
2024-12-11	230.259995	230.259995	231.199997	226.259995	226.410004	35385800
2024-12-12	228.970001	228.970001	231.089996	227.630005	229.830002	28204100
2024-12-13	227.460007	227.460007	230.199997	225.860001	228.399994	28749400

3763 rows × 6 columns

Resetting Index

Converting the Date index back to a column ensures that the Date field is explicitly accessible for analysis, plotting, or filtering operations.

Handling Missing Values

Dropping rows with missing values (data.dropna()) ensures data consistency and accuracy, which is crucial for reliable analysis and modeling.

Converting Date to Datetime

Ensuring the Date column is in datetime format facilitates efficient date-based filtering, aggregation (e.g., monthly or yearly), and plotting.

Preparing for Scaling

Using MinMaxScaler later requires clean and formatted numerical data, which this preprocessing step helps achieve.

In [9]:
```
# Data Preprocessing
import numpy as np
from sklearn.preprocessing import MinMaxScaler

# Reset index to make Date a column
data.reset_index(inplace=True)

# Handle missing values (fill or drop)
da a = data.dropna()
```


t

```
# Convert Date to datetime
data['Date'] = pd.to_datetime(data['Date'])
```

Feature Engineering

Adding Moving Averages

20-day Moving Average (MA20): Provides short-term trend insights, helping identify recent price movements and momentum.

50-day Moving Average (MA50): Captures medium-term trends, offering a smoother representation of price fluctuations over a longer period.

Trend Identification

These moving averages help detect potential buy or sell signals (e.g., when the MA20 crosses above or below the MA50) and assist in understanding the stock's overall direction.

Smoothing Volatility

Moving averages reduce daily price noise, making it easier to interpret patterns and trends in Procter & Gamble's stock performance.

In [11]:
```
# Feature engineering

# Add moving averages
data['MA20'] = data['Close'].rolling(window=20).mean()
data['MA50'] = data['Close'].rolling(window=50).mean()
```

Adding RSI (Relative Strength Index)

RSI is a momentum oscillator that helps identify overbought or oversold conditions in the stock. It ranges from 0 to 100 and indicates potential reversal points when above 70 (overbought) or below 30 (oversold). This is useful for identifying buying or selling opportunities based on market sentiment.

Normalizing Features

MinMaxScaler transforms the Close and Volume values to a range between 0 and 1, making them suitable for machine learning models. This ensures that no single feature dominates due to differences in scale, improving model performance.

Handling NaN Values

Dropping rows with NaN values ensures that only complete and reliable data is used for further analysis, preventing errors or skewed results in subsequent steps.

In [13]:
```
# Add RSI (Relative Strength Index)
delta = data['Close'].diff(1)
gain = delta.where(delta > 0, 0)
loss = -delta.where(delta < 0, 0)
avg_gain = gain.rolling(window=14).mean()
```

```
avg_loss = loss.rolling(window=14).mean()
rs = avg_gain / avg_loss
data['RSI'] = 100 - (100 / (1 + rs))

# Normalize features for machine learning
scaler = MinMaxScaler()
data[['Close_norm', 'Volume_norm']] = scaler.fit_transform(data[['Close', 'Volum

# Drop rows with NaN (resulting from rolling calculations)
data = data.dropna()

# Check the updated dataset
print(data.head())
```

```
Price        Date Adj Close   Close    High     Low    Open     Volume  \
Ticker                 AMZN    AMZN    AMZN    AMZN    AMZN       AMZN
49     2010-03-16    6.5895  6.5895  6.6145   6.525  6.5620   82650000
50     2010-03-17    6.5670  6.5670  6.6345   6.561  6.6205   87176000
51     2010-03-18    6.6380  6.6380  6.6425   6.522  6.5510  100426000
52     2010-03-19    6.5175  6.5175  6.6855   6.483  6.6855  178100000
53     2010-03-22    6.5235  6.5235  6.5480   6.432  6.5100  107668000

Price       MA20     MA50        RSI Close_norm Volume_norm
Ticker
49      6.236600  6.21460  76.715358   0.005155    0.078267
50      6.274175  6.21204  80.529750   0.005055    0.083715
51      6.310875  6.21011  81.574326   0.005371    0.099663
52      6.342950  6.20821  65.281422   0.004835    0.193157
53      6.374100  6.20868  63.616309   0.004861    0.108380
```

Exploratory Data Analysis

Visualizing Stock Price Trends

The line plot provides a clear visual representation of how Procter & Gamble's stock has evolved over time. By plotting the closing prices, you can easily identify long-term trends, periods of stability, and significant fluctuations.

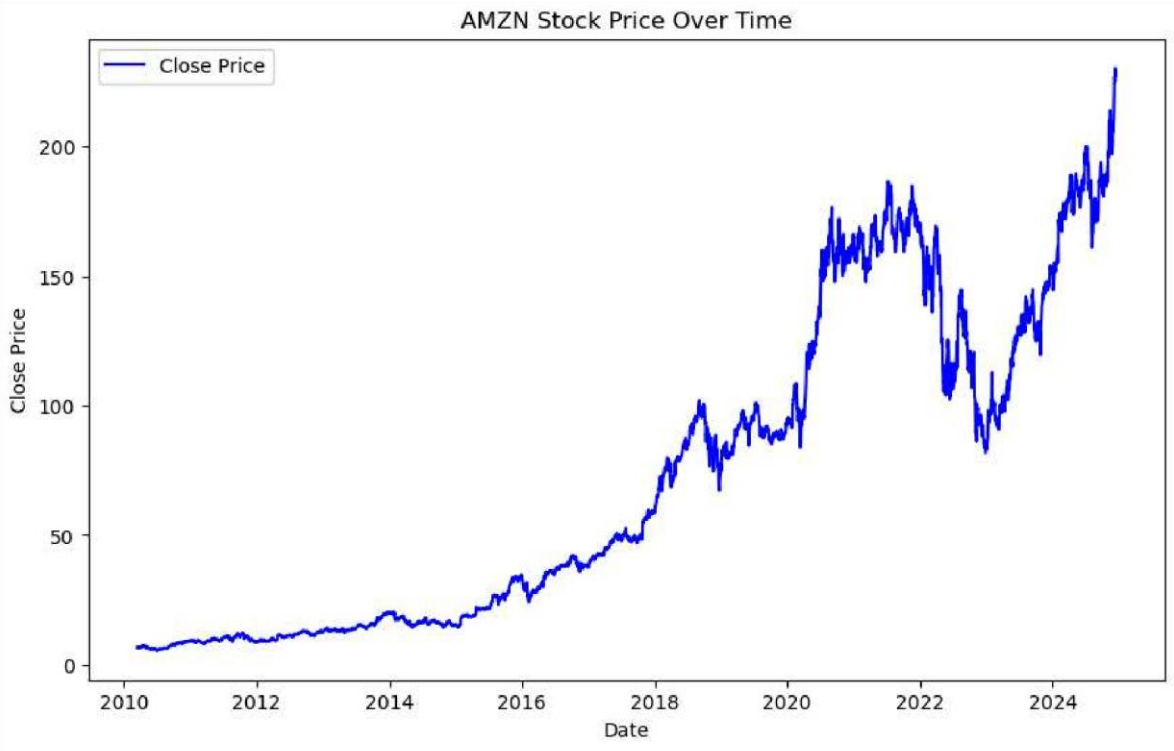
Identifying Patterns

The plot allows for the detection of patterns like upward or downward trends, seasonal fluctuations, and potential anomalies, which could signal important market events or company milestones.

Stock Trends for the AMAZON stock data

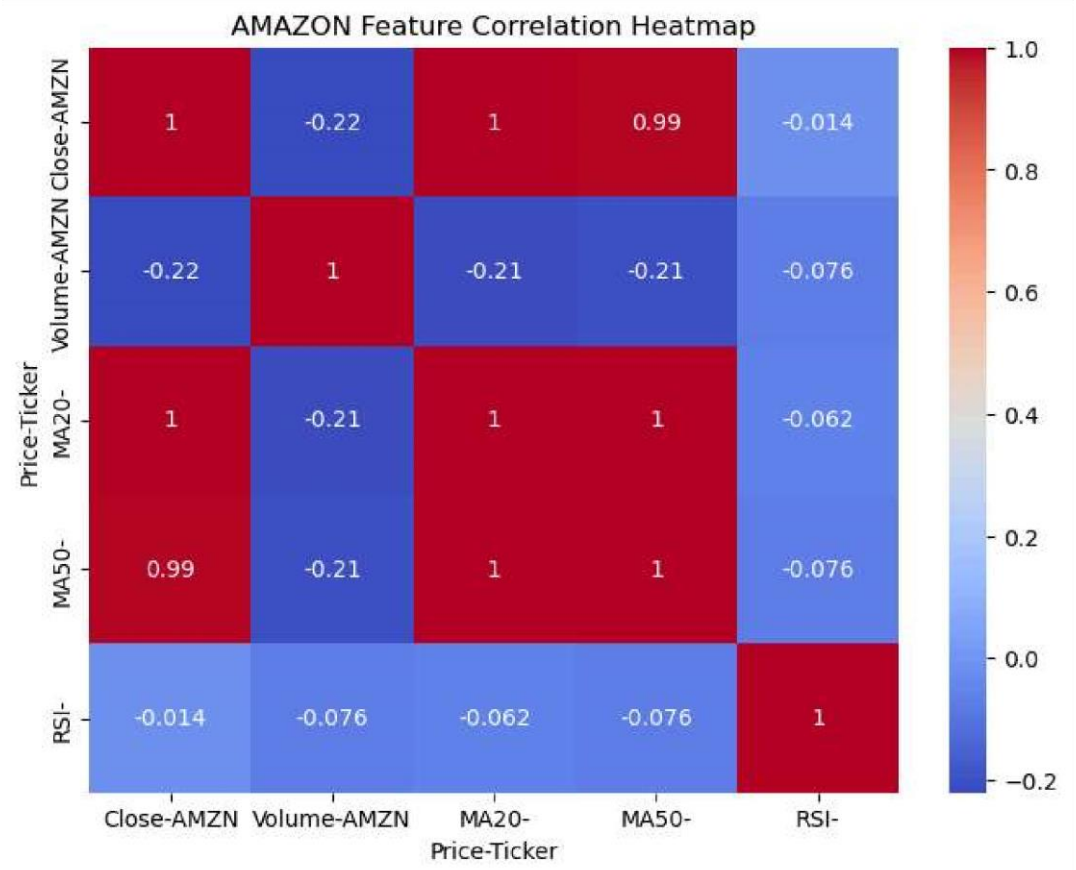
In [15]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Line plot for stock prices
plt.figure(figsize=(10, 6))
plt.plot(data['Date'], data['Close'], label='Close Price', color='blue')
plt.title("AMZN Stock Price Over Time")
plt.xlabel("Date")
plt.ylabel("Close Price")
plt.legend()
pl .show()
```


t



Correlation between Different Features

In [17]:
```
# Correlation heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(data[['Close', 'Volume', 'MA20', 'MA50', 'RSI']].corr(), annot=True,
plt.title("AMAZON Feature Correlation Heatmap")
plt.show()
```



Decomposition of the Time Series

Purpose of Decomposition

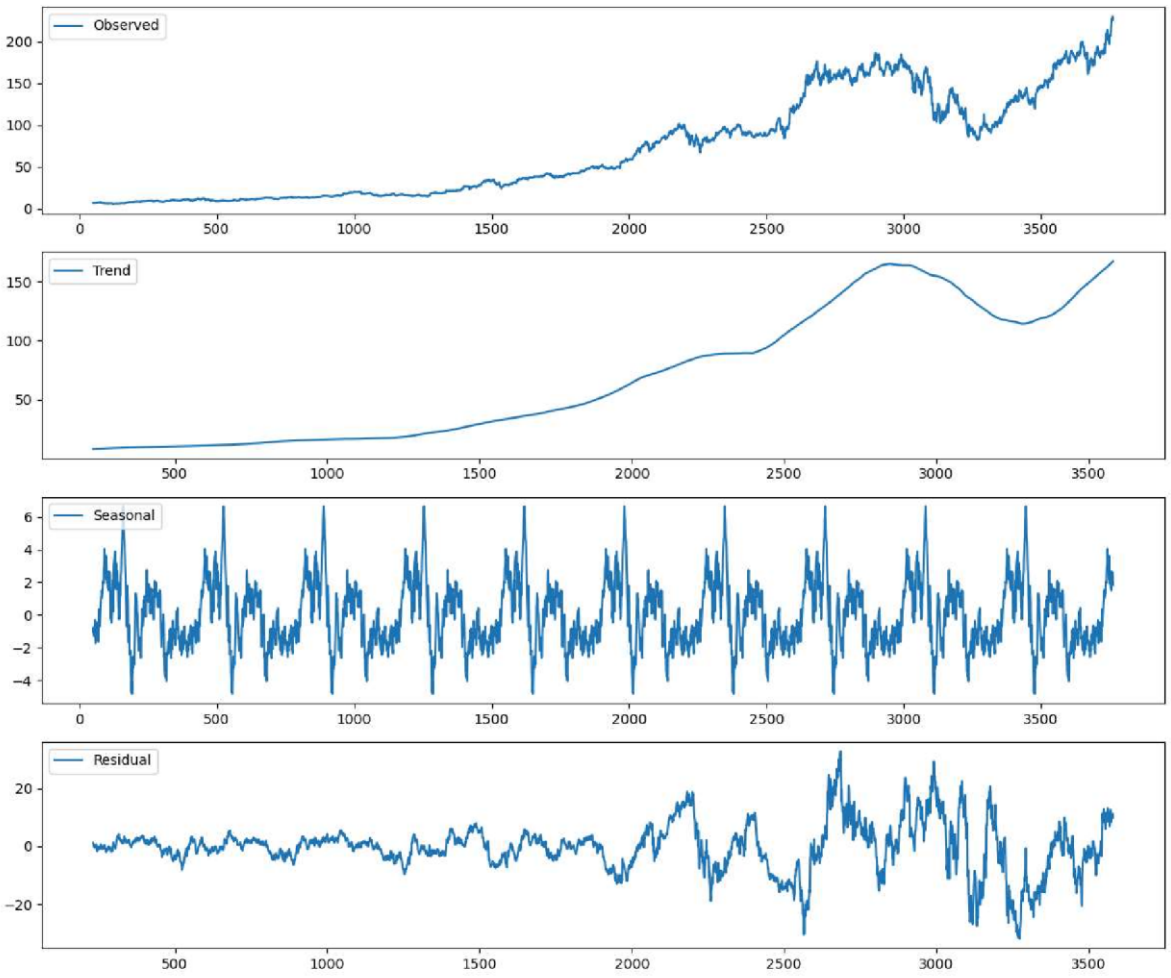
Breaks Down Complexity: *Time series decomposition separates the data into three primary components: trend, seasonality, and residual (or noise). This makes it easier to analyze and interpret patterns in the data.*

Improves Forecasting: *By isolating the trend and seasonal patterns, it allows for better prediction models since each component can be modeled separately.*

In [19]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
import numpy as np

decomposition = seasonal_decompose(data['Close'], model='additive', period=365)

# Plot decomposition components
plt.figure(figsize=(12, 10))
plt.subplot(411)
plt.plot(decomposition.observed, label='Observed')
plt.legend(loc='upper left')
plt.subplot(412)
plt.plot(decomposition.trend, label='Trend')
plt.legend(loc='upper left')
plt.subplot(413)
plt.plot(decomposition.seasonal, label='Seasonal')
pl .legend(loc='upper left')
```


t

```
plt.subplot(414)
plt.plot(decomposition.resid, label='Residual')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```



The ADF Statistic

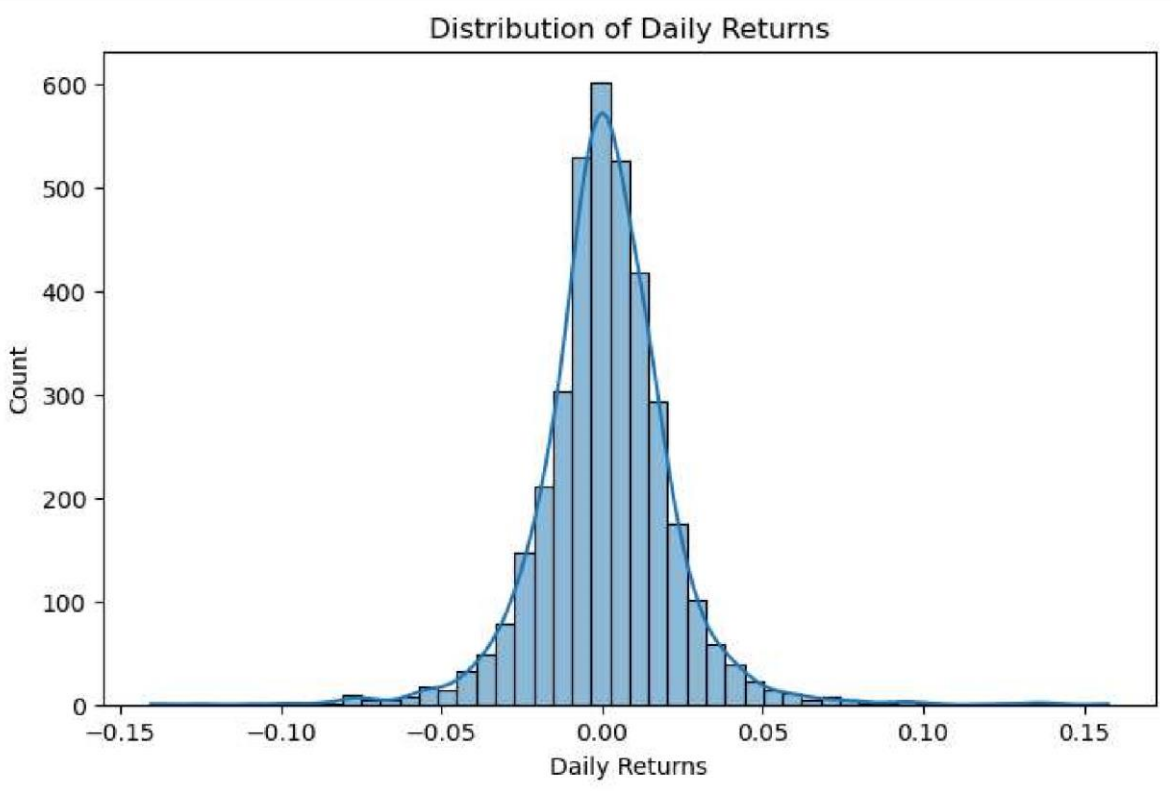
In [27]:
```
# Function to perform ADF test
def adf_test(series):
    result = adfuller(series.dropna())
    print("ADF Statistic:", result[0])
    print("p-value:", result[1])
    print("Critical Values:", result[4])
    if result[1] <= 0.05:
        print("The series is stationary.")
    else:
        print("The series is not stationary.")

# Perform ADF test
print("ADF Test Results for Original Data:")
adf_test(data['Close'])
```

```
ADF Test Results for Original Data:
ADF Statistic: 0.8374945670467888
p-value: 0.9922179153981535
Critical Values: {'1%': -3.4321267753335545, '5%': -2.862325080792323, '10%': -2.
5671879100443813}
The series is not stationary.
```

Daily returns distribution

In [ ]:

In [21]:
```
# Histogram
data['Daily_Returns'] = data['Close'].pct_change()
plt.figure(figsize=(8, 5))
sns.histplot(data['Daily_Returns'].dropna(), bins=50, kde=True)
plt.title("Distribution of Daily Returns")
plt.xlabel("Daily Returns")
plt.show()
```



Transforming the Data

From the above analysis, the data is not stationary with the P-value well above the 0.05 threshold. So, first-order differencing is often sufficient, but if the series is still not stationary.

In [29]:
```
data['Close_diff'] = data['Close'].diff()
```

Testing ADF statistic again

In [31]:
```
from statsmodels.tsa.stattools import adfuller
result = adfuller(data['Close_diff'].dropna())
print(f"ADF Statistic: {result[0]}")
print(f"p-value: {result[1]}")
```

```
ADF Statistic: -11.152764578780785
p-value: 2.9158235816099523e-20
```

Train - Test Split

Purpose of Train-Test Split:

Model Training and Evaluation

The training set is used to train the forecasting or machine learning model, helping it learn patterns (e.g., trends, seasonality, and relationships in the stock price data).

The test set is used to evaluate the model's performance on unseen data, simulating how the model would perform in the real world.

Ensures that the evaluation is unbiased by only testing the model on data it has not seen during training.

Why Use an 80:20 Split?

80% Training Data: A larger training set helps the model learn more robustly by exposing it to a wider range of patterns in the data.

20% Testing Data: This size is sufficient to provide meaningful insights into the model's performance without overly reducing the training data.

In [ ]:
```
# Train-test split (80:20)
train_size = int(len(data) * 0.8)
train = data[:train_size]
test = data[train_size:]
```

In [35]:
```
print(f"Train data size: {len(train)}")
print(f"Test data size: {len(test)}")
```

```
Train data size: 2971
Test data size: 743
```

In [33]:
```
train
```

Out[33]:

Price	Date	Adj Close	Close	High	Low	Open	Volume
Ticker		AMZN	AMZN	AMZN	AMZN	AMZN	AMZN
49	2010-03-16	6.589500	6.589500	6.614500	6.525000	6.562000	82650000
50	2010-03-17	6.567000	6.567000	6.634500	6.561000	6.620500	87176000
51	2010-03-18	6.638000	6.638000	6.642500	6.522000	6.551000	100426000
52	2010-03-19	6.517500	6.517500	6.685500	6.483000	6.685500	178100000
53	2010-03-22	6.523500	6.523500	6.548000	6.432000	6.510000	107668000
...	...	...	...	...	...	...	...
3015	2021-12-23	171.068497	171.068497	171.975006	170.149994	170.427994	36788000
3016	2021-12-27	169.669495	169.669495	172.942993	169.215500	171.037003	58688000
3017	2021-12-28	170.660995	170.660995	172.175995	169.135498	170.182495	54638000
3018	2021-12-29	169.201004	169.201004	171.212006	168.600494	170.839996	35754000
3019	2021-12-30	168.644501	168.644501	170.888000	168.524002	169.699997	37584000

2971 rows × 13 columns

Preparing the Training and Testing Series

This step prepares the data for time series forecasting using the ARIMA model from statsmodels and setting up metrics for performance evaluation using mean squared error (MSE) and mean absolute error (MAE).

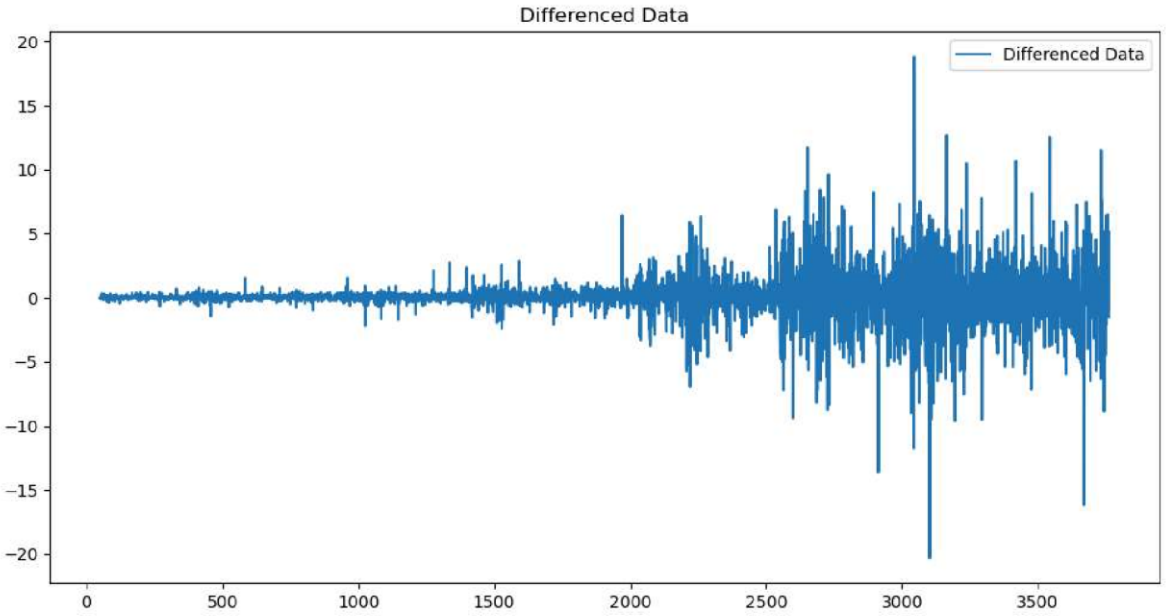
In [ ]:
```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error, mean_absolute_error

# Use 'Close' price for time series forecasting
train_series = train['Close']
test_series = test['Close']
```

Plotting the differenced data

In [39]:
```
data['Price_diff'] = data['Close'].diff()

print("ADF Test Results for Differenced Data:")
adf_test(data['Price_diff'])
```

```
# Plot the differenced data
plt.figure(figsize=(12, 6))
plt.plot(data['Price_diff'], label='Differenced Data')
plt.title("Differenced Data")
plt.legend()
plt.show()
```

```
ADF Test Results for Differenced Data:
ADF Statistic: -11.152764578780785
p-value: 2.9158235816099523e-20
Critical Values: {'1%': -3.432127258227728, '5%': -2.8623252940988464, '10%': -2.
567188023601466}
The series is stationary.
```



In [51]:
```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error

# Fit ARIMA model (order = (p, d, q); change based on tuning)
arima_model = ARIMA(train_series, order=(2, 4, 4))
arima_fit = arima_model.fit()

# Forecast on the test data
forecast = arima_fit.forecast(steps=len(test_series))

# Evaluate model performance
rmse = np.sqrt(mean_squared_error(test_series, forecast))
mae = mean_absolute_error(test_series, forecast)
print(f"ARIMA Performance Metrics:\n  RMSE: {rmse:.4f}\n  MAE: {mae:.4f}")
```

```
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473:
ValueWarning: An unsupported index was provided and will be ignored when e.g. for
ecasting.
  self._init_dates(dates, freq)
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473:
ValueWarning: An unsupported index was provided and will be ignored when e.g. for
ecasting.
  self._init_dates(dates, freq)
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473:
ValueWarning: An unsupported index was provided and will be ignored when e.g. for
ecasting.
  self._init_dates(dates, freq)
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\statespace\sarimax.py:
978: UserWarning: Non-invertible starting MA parameters found. Using zeros as sta
rting parameters.
  warn('Non-invertible starting MA parameters found.'
ARIMA Performance Metrics:
  RMSE: 40.8740
  MAE: 33.5369
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836:
ValueWarning: No supported index is available. Prediction results will be given w
ith an integer index beginning at `start`.
  return get_prediction_index(
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836:
FutureWarning: No supported index is available. In the next version, calling this
method in a model without a supported index will result in an exception.
  return get_prediction_index(
```

Forecasting the Model

In [53]:

```
# Forecast
arima_fit = arima_model.fit()
forecast = arima_fit.forecast(steps=len(test))

# Evaluate using RMSE
rmse = np.sqrt(mean_squared_error(test_series, forecast))
print(f"RMSE: {rmse}")


# Plot actual vs predicted values
plt.figure(figsize=(12, 6))
plt.plot(train['Date'], train['Close'], label='Training Data', color='blue')
plt.plot(test['Date'], test['Close'], label='Actual Test Data', color='green')
plt.plot(test['Date'], forecast, label='ARIMA Forecast', color='red')
plt.title("ARIMA Forecast vs Actual Prices")
plt.xlabel("Date")
plt.ylabel("Close Price")
plt.legend()
plt.show()
```

```
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\statespace\sarimax.py:
978: UserWarning: Non-invertible starting MA parameters found. Using zeros as sta
rting parameters.
  warn('Non-invertible starting MA parameters found.'
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836:
ValueWarning: No supported index is available. Prediction results will be given w
ith an integer index beginning at `start`.
  return get_prediction_index(
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836:
FutureWarning: No supported index is available. In the next version, calling this
method in a model without a supported index will result in an exception.
  return get_prediction_index(
RMSE: 40.874010564666044
```

Hyperparameter Tuning

Importance of Hyperparameter Tuning

Improves Model Performance

Prevents Overfitting/Underfitting

Enhances Generalization

Tailors the Model to the Dataset

In [49]:
```
from itertools import product

# Define parameter grid
p = d = q = range(0, 5)
pdq = list(product(p, d, q))

# Grid search to find the best parameters
best_score, best_params = float('inf'), None
for params in pdq:
    try:
        model = ARIMA(train_series, order=params)
        model_fit = model.fit()
        mse = mean_squared_error(test_series, model_fit.forecast(steps=len(test)
```

```
        if mse < best_score:
            best_score, best_params = mse, params
    except:
        continue

print(f"Best Parameters: {best_params} with MSE: {best_score}")
```

```
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836:
ValueWarning: No supported index is available. Prediction results will be given w
ith an integer index beginning at `start`.
  return get_prediction_index(
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836:
FutureWarning: No supported index is available. In the next version, calling this
method in a model without a supported index will result in an exception.
  return get_prediction_index(
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473:
ValueWarning: An unsupported index was provided and will be ignored when e.g. for
ecasting.
  self._init_dates(dates, freq)
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473:
ValueWarning: An unsupported index was provided and will be ignored when e.g. for
ecasting.
  self._init_dates(dates, freq)
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473:
ValueWarning: An unsupported index was provided and will be ignored when e.g. for
ecasting.
  self._init_dates(dates, freq)
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\statespace\sarimax.py:
978: UserWarning: Non-invertible starting MA parameters found. Using zeros as sta
rting parameters.
  warn('Non-invertible starting MA parameters found.'
Best Parameters: (2, 4, 4) with MSE: 1670.6847396404314
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836:
ValueWarning: No supported index is available. Prediction results will be given w
ith an integer index beginning at `start`.
  return get_prediction_index(
C:\Users\samue\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836:
FutureWarning: No supported index is available. In the next version, calling this
method in a model without a supported index will result in an exception.
  return get_prediction_index(
```

Regression with Random Forest

Random Forest is a robust and versatile machine learning algorithm that can be used for regression tasks. And below is how its implemetation is done;

In [55]:
```python
# Regression with Random Forests

# Add lagged features (e.g., previous day's close price)
data['Close_1'] = data['Close'].shift(1)
data['Close_2'] = data['Close'].shift(2)

# Drop NaN rows caused by shifting
data_rf = data.dropna()

# Define features (X) and target (y)
features = ['MA20', 'MA50', 'RSI', 'Volume', 'Close_1', 'Close_2']
X = data_rf[features]
y = data_rf['Close']

# Train-test split
train_size_rf = int(len(data_rf) * 0.8)
X_train, X_test = X[:train_size_rf], X[train_size_rf:]
y_train, y_test = y[:train_size_rf], y[train_size_rf:]
```

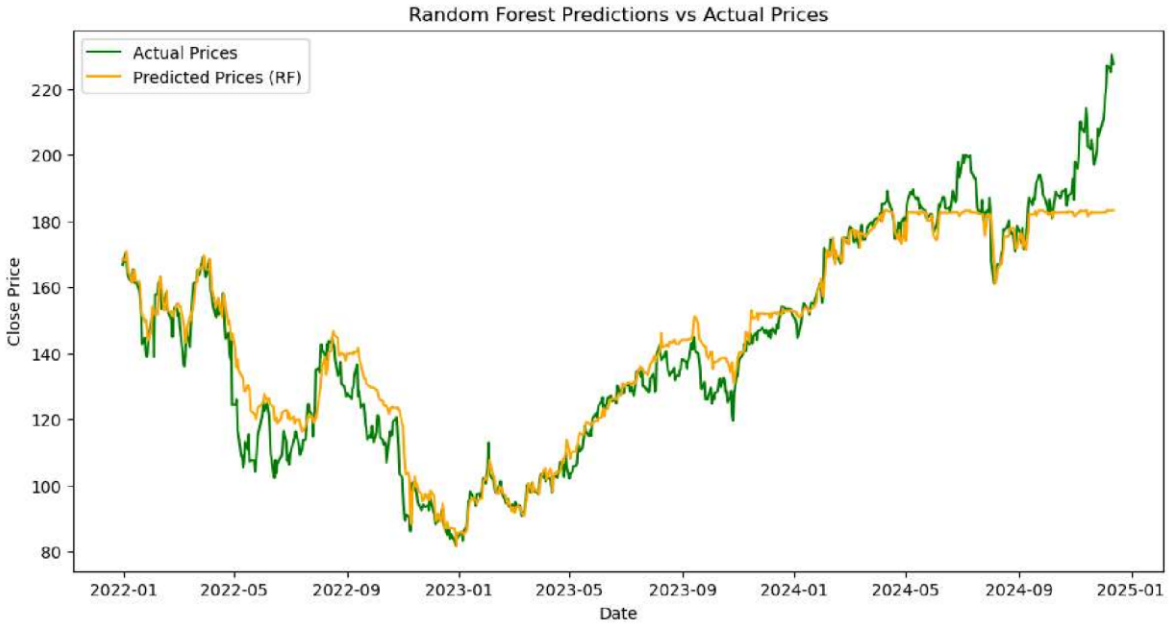
Training, making predictions and evaluating the Random forest regressor model

In [57]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score

# Train Random Forest Regressor
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Make predictions
rf_predictions = rf_model.predict(X_test)

# Evaluate model
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_predictions))
rf_mae = mean_absolute_error(y_test, rf_predictions)
rf_r2 = r2_score(y_test, rf_predictions)
print(f"Random Forest Performance Metrics:\n  RMSE: {rf_rmse:.4f}\n  MAE: {rf_ma
```

```
C:\Users\samue\anaconda3\Lib\site-packages\sklearn\base.py:1474: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change t
he shape of y to (n_samples,), for example using ravel().
  return fit_method(estimator, *args, **kwargs)
Random Forest Performance Metrics:
  RMSE: 9.0244
  MAE: 5.9881
  R²: 0.9296
```

Plotting the actual vs predicted trends

In [59]:
```
# Actual vs Predicted
plt.figure(figsize=(12, 6))
plt.plot(data_rf['Date'].iloc[train_size_rf:], y_test, label='Actual Prices', cc
plt.plot(data_rf['Date'].iloc[train_size_rf:], rf_predictions, label='Predicted
plt.title("Random Forest Predictions vs Actual Prices")
plt.xlabel("Date")
plt.ylabel("Close Price")
plt.legend()
plt.show()
```



Evaluating the Performance of my model in terms of percentage

In [63]:
```
from sklearn.metrics import r2_score

# Calculate R² score
r2 = r2_score(y_test, rf_predictions)

# Convert to percentage
r2_percentage = r2 * 100

print(f"R² Score: {r2:.2f}")
print(f"Model explains {r2_percentage:.2f}% of the variance in the target variab
```

```
R² Score: 0.93
Model explains 92.96% of the variance in the target variable.
```

The R^2 Score of 0.93 indicates that your Random Forest regression model explains 92.96% of the variance in the target variable. This is an excellent result and demonstrates that your model captures most of the patterns in the data.

Final Verdict

Your model's performance is excellent, with an R^2 score of 0.93 showing strong predictive accuracy. With minor refinements, it can serve as a reliable tool for forecasting and analyzing trends in Amazon stock prices.

Presenting Findings

Streamlit is a Python library designed to create web-based apps for machine learning and data analysis quickly and easily.

This step involves Streamlit and is crucial in terms of presenting the analysis and making insights actionable.

This step creates an interactive user interface (UI) for visualizing and sharing the stock price prediction results with others.

In [69]:
```
print(data.head())
print(data.columns)
```

```
Price        Date Adj Close   Close    High     Low    Open     Volume  \
Ticker                 AMZN    AMZN    AMZN    AMZN    AMZN       AMZN
49     2010-03-16    6.5895  6.5895  6.6145   6.525  6.5620   82650000
50     2010-03-17    6.5670  6.5670  6.6345   6.561  6.6205   87176000
51     2010-03-18    6.6380  6.6380  6.6425   6.522  6.5510  100426000
52     2010-03-19    6.5175  6.5175  6.6855   6.483  6.6855  178100000
53     2010-03-22    6.5235  6.5235  6.5480   6.432  6.5100  107668000
```

```
Price       MA20     MA50        RSI Close_norm Volume_norm Daily_Returns  \
Ticker
49      6.236600  6.21460  76.715358   0.005155    0.078267           NaN
50      6.274175  6.21204  80.529750   0.005055    0.083715     -0.003415
51      6.310875  6.21011  81.574326   0.005371    0.099663      0.010812
52      6.342950  6.20821  65.281422   0.004835    0.193157     -0.018153
53      6.374100  6.20868  63.616309   0.004861    0.108380      0.000921

Price  Close_diff Price_diff Close_1 Close_2
Ticker
49            NaN        NaN     NaN     NaN
50        -0.0225    -0.0225  6.5895     NaN
51         0.0710     0.0710  6.5670  6.5895
52        -0.1205    -0.1205  6.6380  6.5670
53         0.0060     0.0060  6.5175  6.6380
MultiIndex([(         'Date',     ''),
            (    'Adj Close', 'AMZN'),
            (        'Close', 'AMZN'),
            (         'High', 'AMZN'),
            (          'Low', 'AMZN'),
            (         'Open', 'AMZN'),
            (       'Volume', 'AMZN'),
            (         'MA20',     ''),
            (         'MA50',     ''),
            (          'RSI',     ''),
            (   'Close_norm',     ''),
            (  'Volume_norm',     ''),
            ('Daily_Returns',     ''),
            (   'Close_diff',     ''),
            (   'Price_diff',     ''),
            (      'Close_1',     ''),
            (      'Close_2',     '')],
           names=['Price', 'Ticker'])
```

Flattening the Multi-index colums

In [71]:
```
# Flatten MultiIndex column names
data.columns = ['_'.join(col).strip() if isinstance(col, tuple) else col for col

# Verify column names
print(data.columns)
```

```
Index(['Date_', 'Adj Close_AMZN', 'Close_AMZN', 'High_AMZN', 'Low_AMZN',
       'Open_AMZN', 'Volume_AMZN', 'MA20_', 'MA50_', 'RSI_', 'Close_norm_',
       'Volume_norm_', 'Daily_Returns_', 'Close_diff_', 'Price_diff_',
       'Close_1_', 'Close_2_'],
      dtype='object')
```

Adjusting the Streamlit Line Chart Code

In [ ]:
```python
import streamlit as st

st.title("AMZN Stock Price Prediction")

st.line_chart(data[['Close_AMZN', 'MA20_AMZN', 'MA50_AMZN']])

st.write("Predicted Close Prices:", forecast)
```

Handling Missing data

In [ ]:
```python
# Drop rows with NaN values for required columns
data = data.dropna(subset=['Close_AMZN', 'MA20_AMZN', 'MA50_AMZN'])
```

In [ ]:
```python

```