1 Import Libraries and Data

In [1]:

```python
import pandas as pd
import numpy as np
import math
from sklearn.metrics import mean_squared_error

import matplotlib.pyplot as plt
plt.style.use('dark_background')

import plotly.express as px
import plotly.io as pio
pio.templates.default = 'plotly_dark'

import statsmodels.api as sm

import warnings
warnings.filterwarnings("ignore")

df = pd.read_csv('data/eth.csv')
print(df.shape)
df
```

executed in 4.92s, finished 17:08:18 2021-06-16

```
(2134, 7)
```

	Date	Open_	High	Low	Close__	Volume	MarketCap
0	Jun 08, 2021	$2,594.60	$2,620.85	$2,315.55	$2,517.44	$41,909,736,778	$292,557,075,207
1	Jun 07, 2021	$2,713.05	$2,845.19	$2,584.00	$2,590.26	$30,600,111,277	$300,985,400,826
2	Jun 06, 2021	$2,629.75	$2,743.44	$2,616.16	$2,715.09	$25,311,639,414	$315,453,931,558
3	Jun 05, 2021	$2,691.62	$2,817.48	$2,558.23	$2,630.58	$30,496,672,724	$305,598,725,249
4	Jun 04, 2021	$2,857.17	$2,857.17	$2,562.64	$2,688.19	$34,173,841,611	$312,256,566,095

	Date	Open_	High	Low	Close__	Volume	MarketCap
...	...	...	...	...	...	...	...
2129	Aug 10, 2015	$0.71	$0.73	$0.64	$0.71	$405,283	$42,818,364
2130	Aug 09, 2015	$0.71	$0.88	$0.63	$0.7	$532,170	$42,399,574
2131	Aug 08, 2015	$2.79	$2.80	$0.71	$0.75	$674,188	$45,486,894
2132	Aug 07, 2015	$2.83	$3.54	$2.52	$2.77	$164,329	$166,610,555
2133	Jun 09, 2021	$2,510.20	$2,625.07	$2,412.20	$2,608.27	$36,075,832,186	$303,147,462,062

2134 rows × 7 columns

2 Goal and Data Description

2.1 Goal

The goal of this project is to create a model that predicts prices that allow for successful day-trading. I want to make sure that the model predicts one day ahead, and that the culmination of all of these predictions follows the general trend of the actual prices, in order to allow day traders to make proper predictions to maximize profit or minimize loss.

2.2 Data Source

This data was scraped from CoinMarketCap.com using the webscraper Octoparse. The webpages used ajax syntax for the "load page" button, and therfore ajax timeout time needed to be applied in order to properly extract the data. This data is only concerned with Ethereum, and no other coin or blockchain.

2.3 Features

The data includes the following features:

1. Open
2. High
3. Low
4. Close

5. Volume
6. Market Cap

This dataset provides a timeline of Ethereum prices and related data from August 7th, 2015 to June 8th, 2021.

3 Data Preprocessing

In [2]:

```python
# Convert the 'Date' column to a datetime datatype and set it as the index, then sort the index
df['Date'] = pd.to_datetime(df.Date)
df.set_index(df.Date, inplace=True)
df.drop(df.tail(1).index, inplace=True)
df = df.sort_index()

# Drop the Date column
df = df.drop(columns=['Date'], axis=1)

# Specify columns
cols = list(df.columns)

# Replace the dollar signs and commas with empty character
df[cols] = df[cols].replace({'\$': '', ',': ''}, regex=True)

## Convert all entries to numerical data type
for col in cols:
    df[col] = pd.to_numeric(df[col], errors='coerce')

# Rename the columns with unconventinal text in the string
df.rename(columns={'Open_':'Open', 'Close__':'Close'}, inplace=True)

# Find missing values
print(df.isna().sum())

# There are very few missing values, so we will drop all of them
df = df.dropna()

# Check for duplicates in index
print(df.index.duplicated().sum())
```

```python
# Check for duplicates in columns
print(df.duplicated().sum())

# Check how much of the data are duplicates overall
print(df[df.duplicated()==True].shape[0] / df.shape[0])

# There are no duplicates but let's use the drop_duplciates method just as good practice
df = df.drop_duplicates()
print(df.shape)
df.info()
```

executed in 338ms, finished 17:08:26 2021-06-16

```
Open         0
High         0
Low          0
Close        0
Volume       0
MarketCap    0
dtype: int64
0
0
0.0
(2133, 6)
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2133 entries, 2015-08-07 to 2021-06-08
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Open       2133 non-null   float64
 1   High       2133 non-null   float64
 2   Low        2133 non-null   float64
 3   Close      2133 non-null   float64
 4   Volume     2133 non-null   int64  
 5   MarketCap  2133 non-null   int64  
dtypes: float64(4), int64(2)
memory usage: 116.6 KB
```

4 EDA

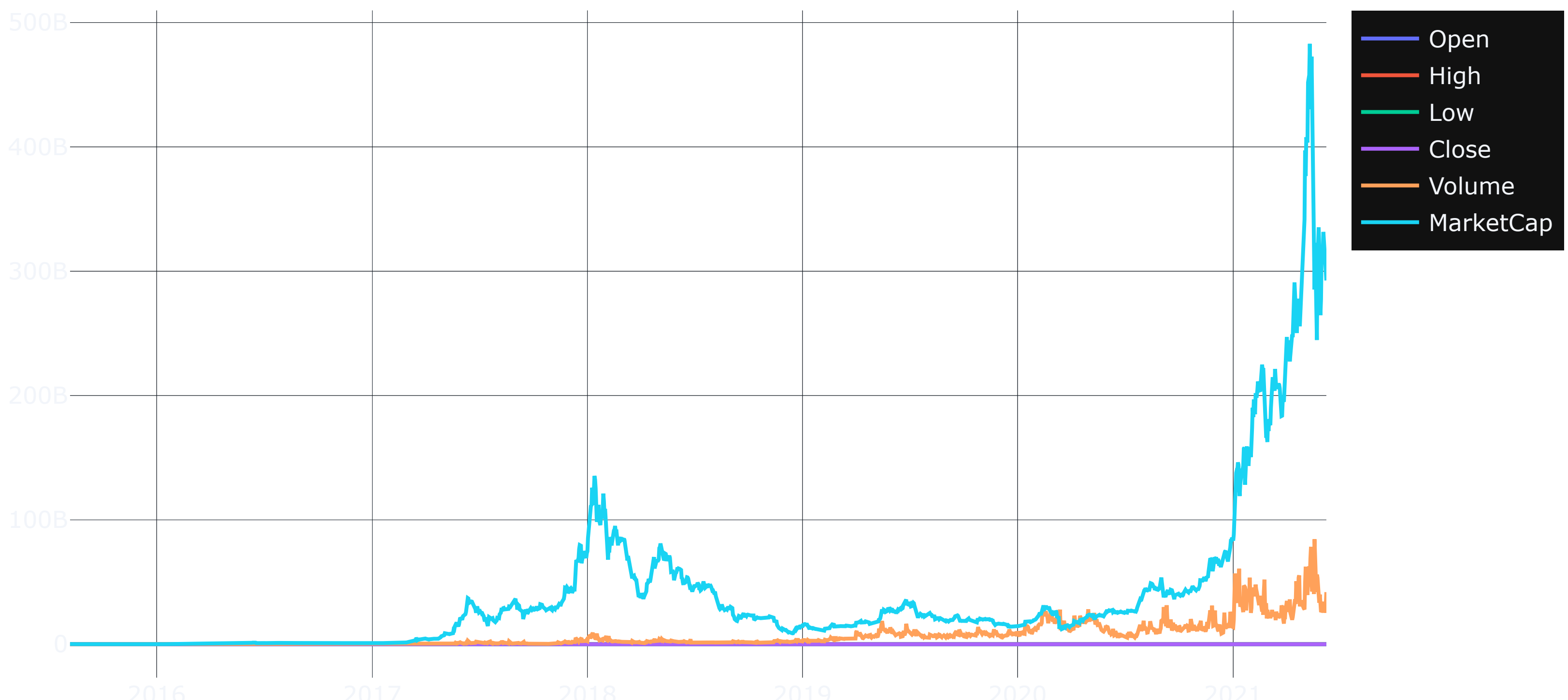
4.0.1 Original Time Series Visualizations

Let's take a look at the time series.

In [3]:

```
# Import graph objects
import plotly.graph_objects as go
fig = go.Figure()

# Add traces
for c in list(df.columns):
    fig.add_trace(go.Scatter(x=df.index, y=df[c], mode='lines', name=f'{c}'))
fig
```

executed in 779ms, finished 19:06:27 2021-06-15

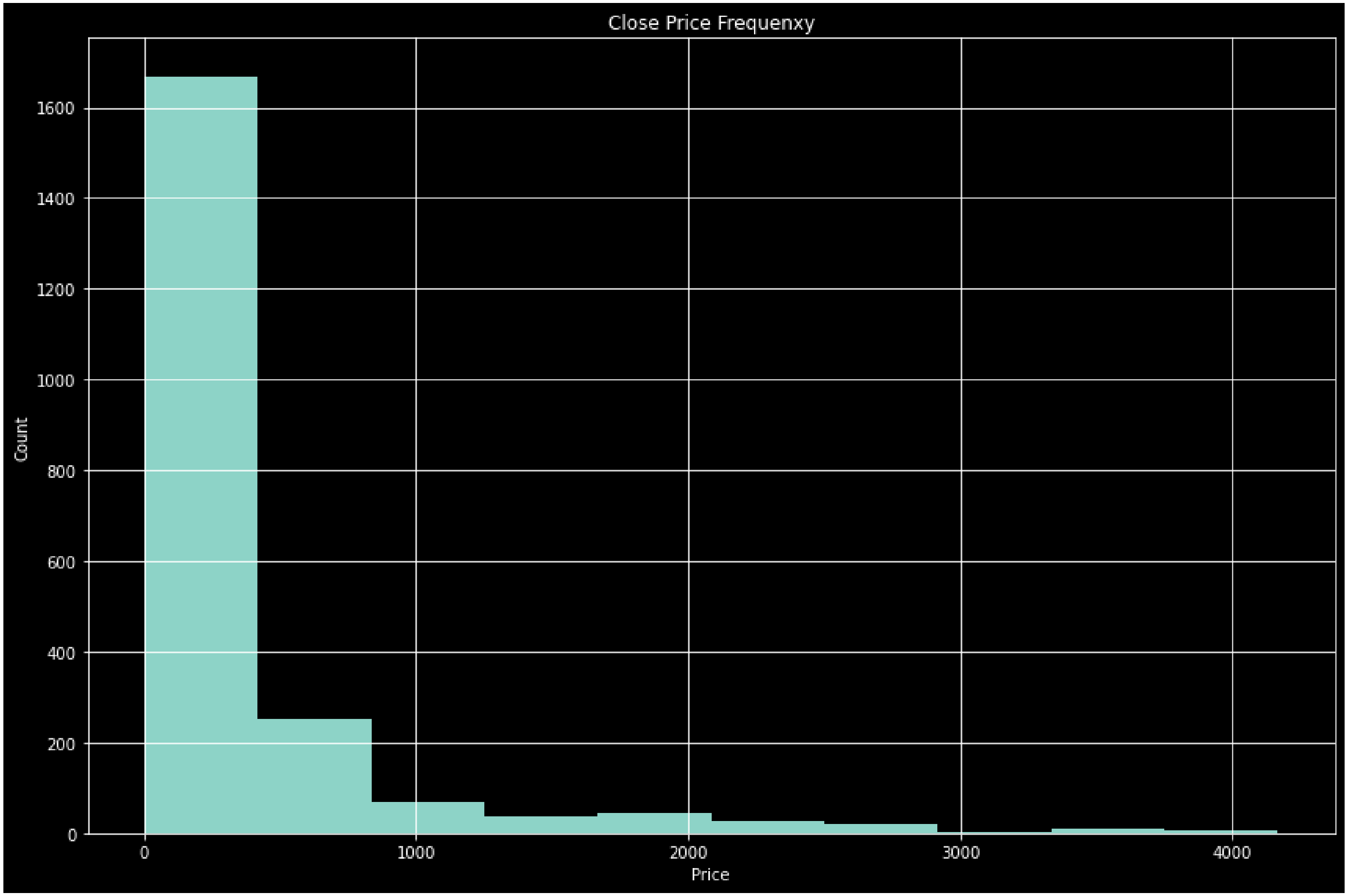


4.0.2 Histograms

In [18]:

```
fig, ax = plt.subplots(figsize=(12,8))
df.hist(column=['Close'], ax=ax)
ax.set_title('Close Price Frequenxy')
ax.set_xlabel('Price')
ax.set_ylabel('Count')
plt.tight_layout()
# plt.savefig('Histograms')
```

executed in 193ms, finished 14:12:26 2021-06-16



Above is a histogram of the frequency of occurences of price value. Their distribution exemplifies the volatility of the asset. The large majority of prices fall between 0 and 1000, however there are low-frequency instances of prices that are 2, 3, and 4 times the max value of that range. This shows that the price spiked and fell, never maintaining a high value for very long at all.

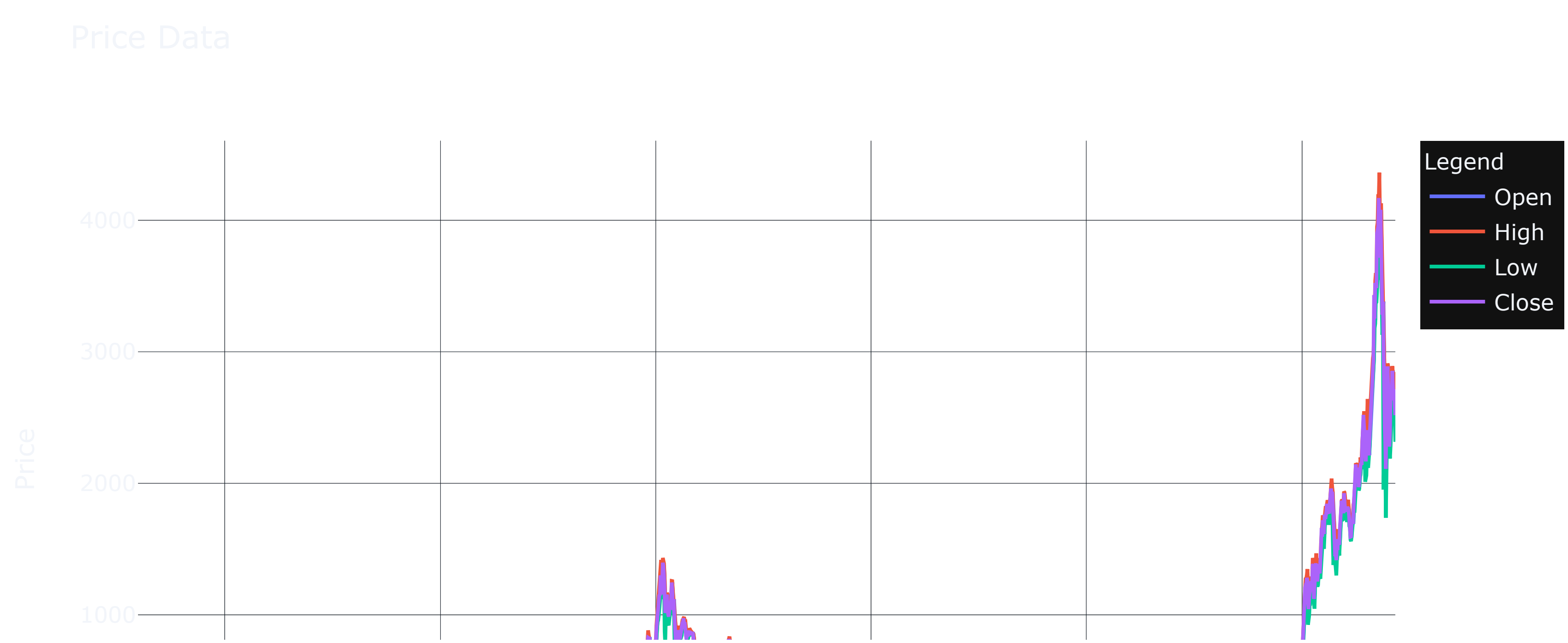
4.0.3 Clean up the Graphs

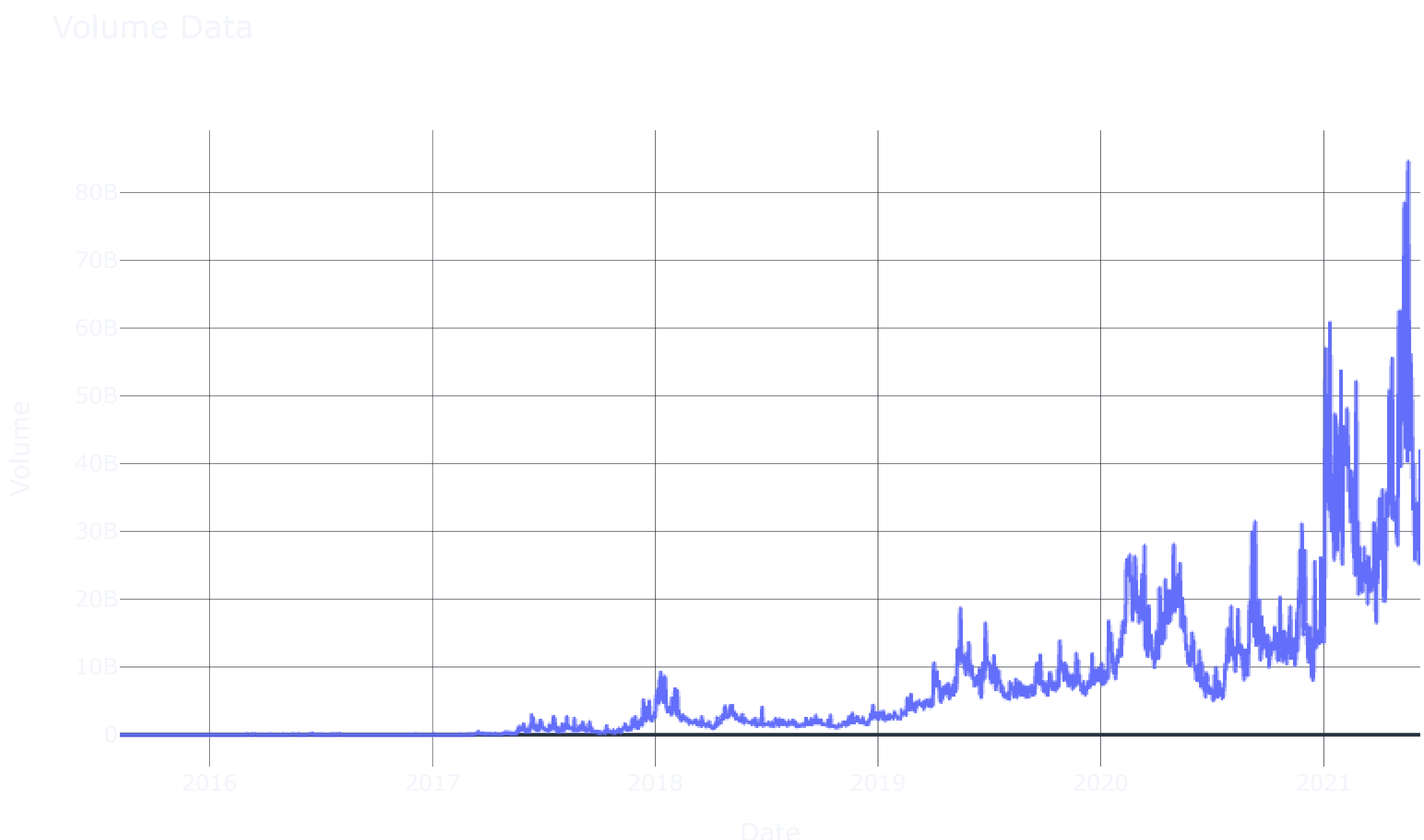
The original time series was very hard to interpret because the volume column has very large numbers that messed with the scale of the graph. In order to remedy this, we will plot the price data and the volume data seperately, and we will resample the Volume data in order

In [5]:

```
# Plot the time series
fig = go.Figure()
col = ['Open', 'High', 'Low', 'Close']

# Add traces
for c in col:
    fig.add_trace(go.Scatter(x=df.index, y=df[c], mode='lines', name=f'{c}'))
fig.update_layout(
title='Price Data',
xaxis_title='Date',
yaxis_title='Price',
legend_title='Legend')
fig.show()
display(px.line(data_frame=df, x=df.index, y=df['Volume'], title='Volume Data'))
```

executed in 570ms, finished 19:06:29 2021-06-15





Target Variable

The trends of each series for each price related column (our target) are pretty much identical, so we can choose one of the features as a target variable and stick with that.

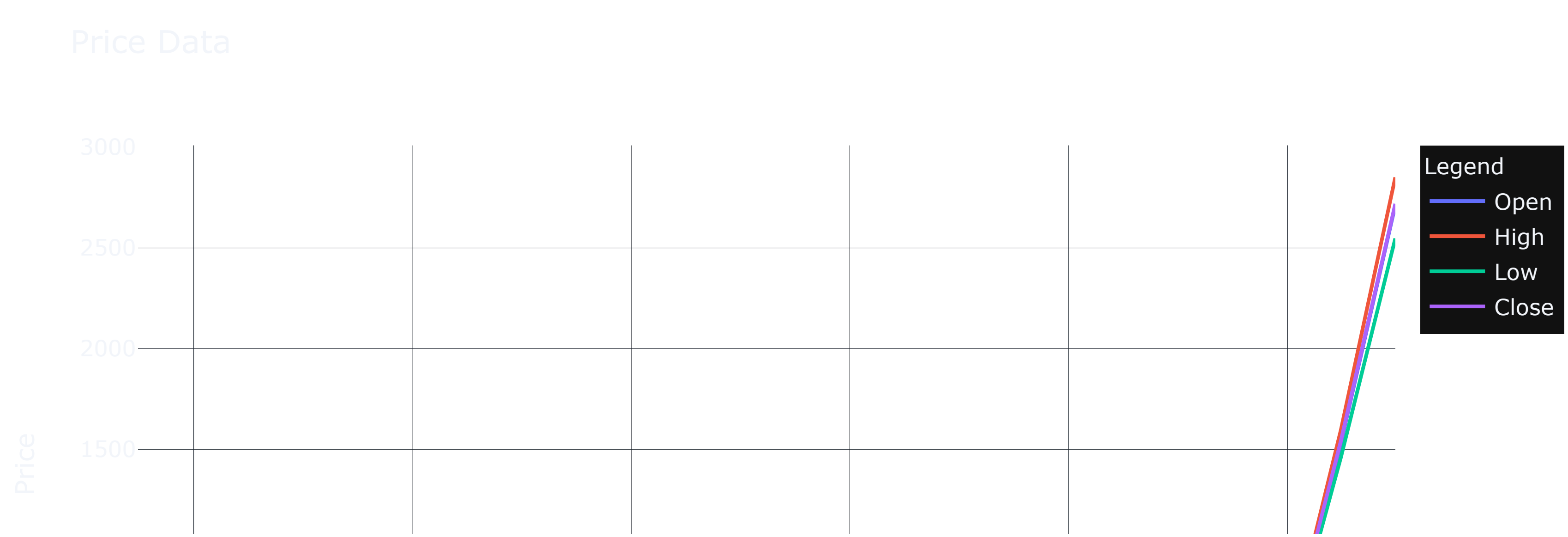
I will be using the "Close" price for Ethereum, which is the price of the asset at the close of normal trading hours at 4pm.

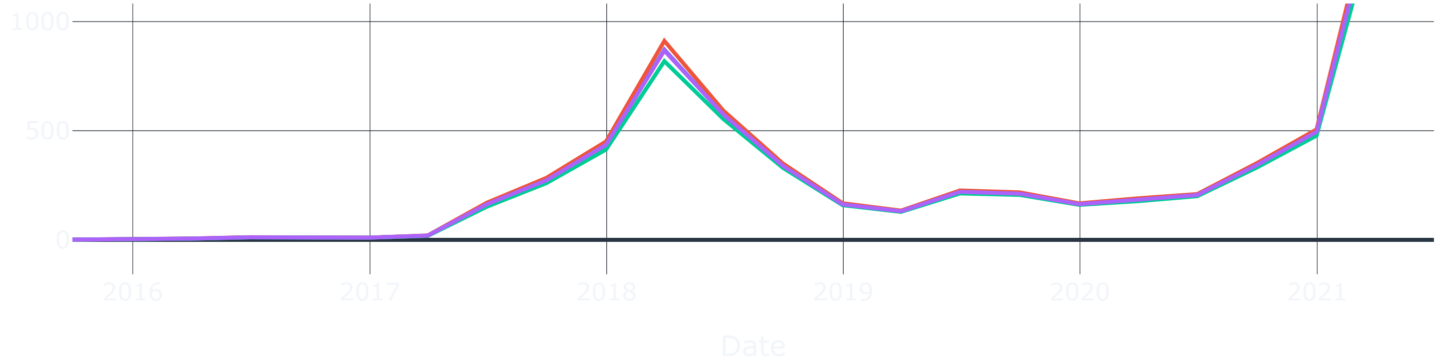
4.0.4 Resample Data (Week, Month, Year)

In [6]:

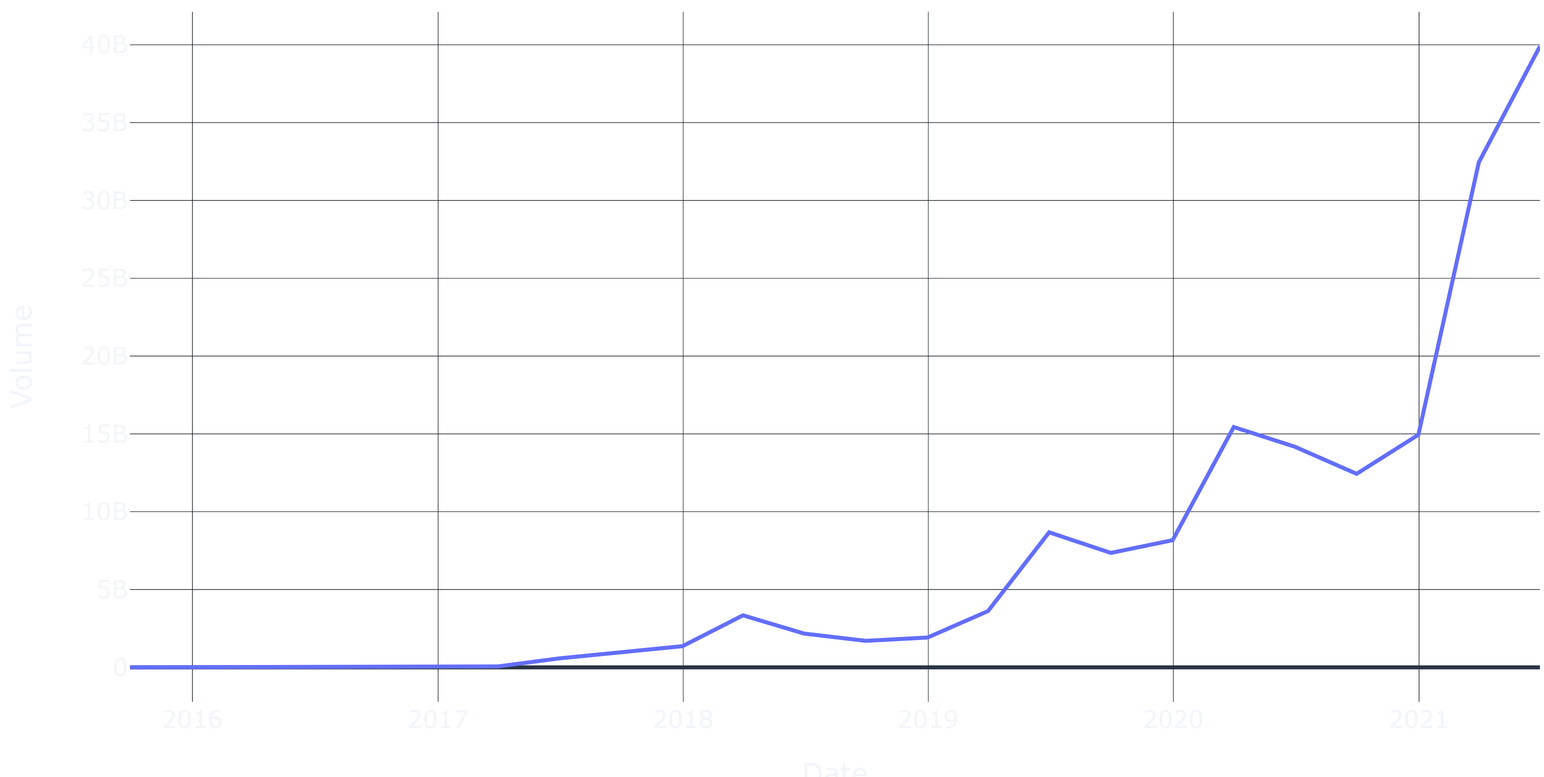
```
# Create resampled DataFrame for more smooth visualization
quarterly_df = pd.DataFrame(df.resample('Q').mean())

# Plot the time series
fig = go.Figure()
col = ['Open', 'High', 'Low', 'Close']
# Add traces
for c in col:
    fig.add_trace(go.Scatter(x=quarterly_df.index, y=quarterly_df[c], mode='lines', name=f'{c}'))
fig.update_layout(
title='Price Data',
xaxis_title='Date',
yaxis_title='Price',
legend_title='Legend')
fig.show()
display(px.line(data_frame=quarterly_df, x=quarterly_df.index, y=quarterly_df['Volume'], title='Volume Data
```

executed in 92ms, finished 19:06:29 2021-06-15




1000
500
0
2016
2017
2018
2019
2020
2021
Date

Volume Data

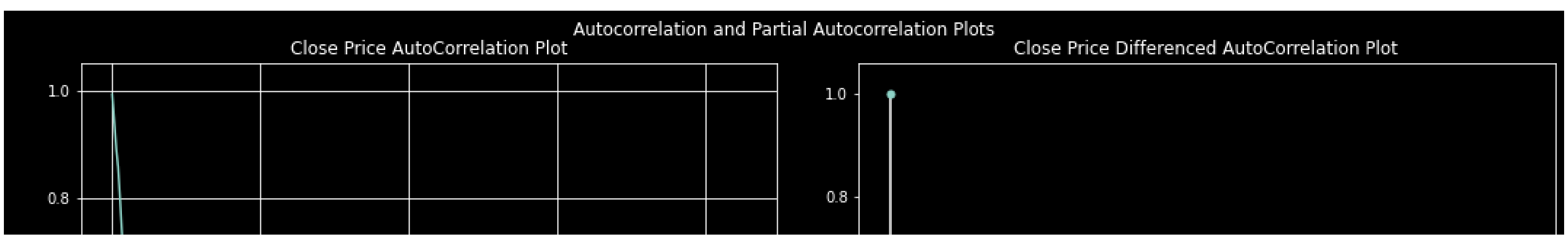

40B
35B
30B
25B
20B
15B
10B
5B
0
Volume
2016
2017
2018
2019
2020
2021
Date

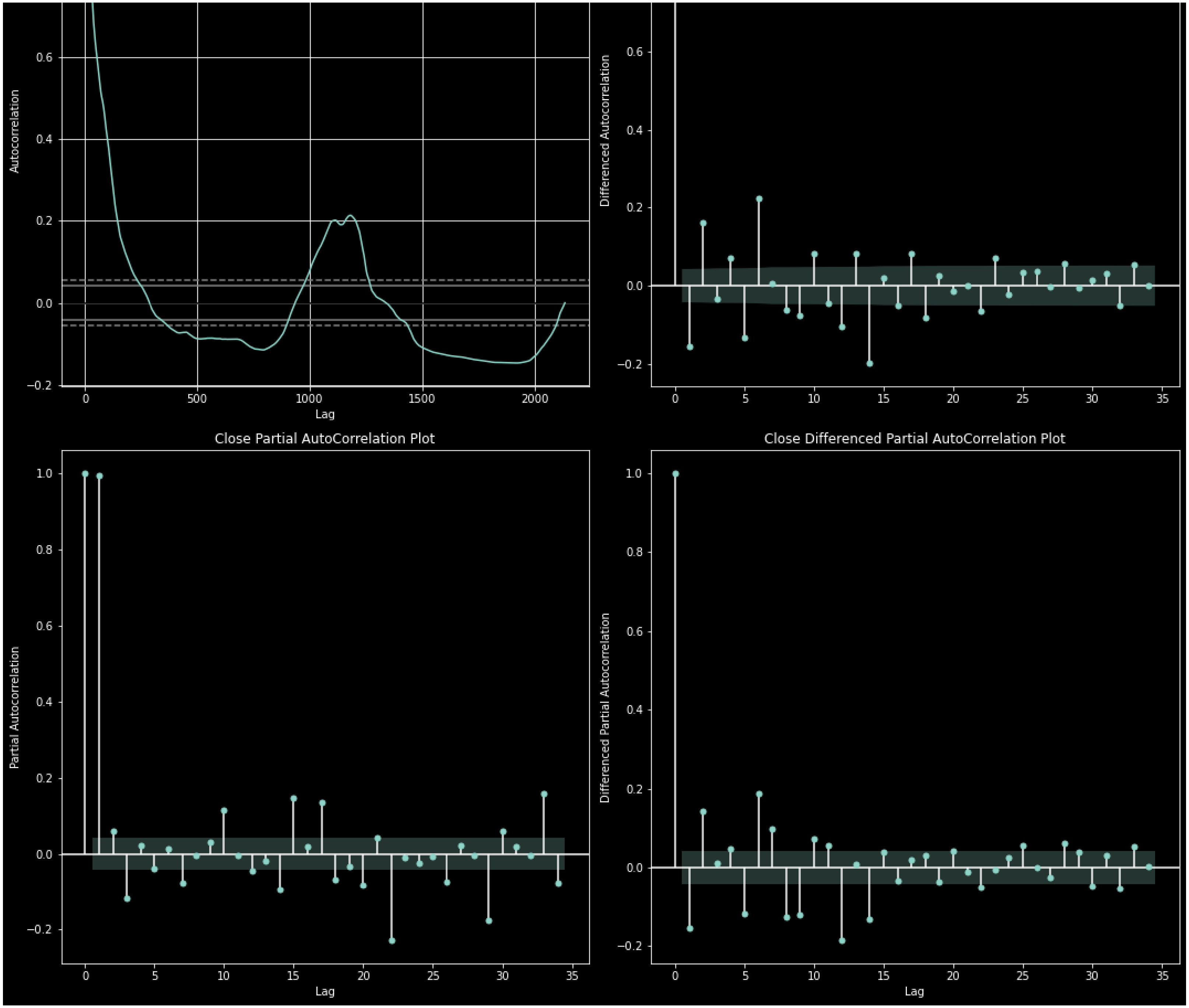
4.0.5 Autocorrelation Plots

In [7]:

```
from pandas.plotting import autocorrelation_plot
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf
fig, ax = plt.subplots(2, 2, figsize=(15, 15))

autocorrelation_plot(df['Close'].dropna(), ax=ax[0][0])
ax[0][0].set_title('Close Price AutoCorrelation Plot')

plot_acf(df['Close'].diff().dropna(), ax=ax[0][1])
ax[0][1].set_title('Close Price Differenced AutoCorrelation Plot')
ax[0][1].set_ylabel('Differenced Autocorrelation')

plot_pacf(df['Close'].dropna(), ax=ax[1][0])
ax[1][0].set_title('Close Partial AutoCorrelation Plot')
ax[1][0].set_xlabel('Lag')
ax[1][0].set_ylabel('Partial Autocorrelation')

plot_pacf(df['Close'].diff().dropna(), ax=ax[1][1])
ax[1][1].set_title('Close Differenced Partial AutoCorrelation Plot')
ax[1][1].set_xlabel('Lag')
ax[1][1].set_ylabel('Differenced Partial Autocorrelation')

plt.suptitle('Autocorrelation and Partial Autocorrelation Plots')
plt.tight_layout()
plt.savefig('acf_plots')
```

executed in 1.14s, finished 19:06:30 2021-06-15





4.0.6 Test Stationarity

In [8]:

```python
from statsmodels.tsa.stattools import adfuller

# ADF Test for Non-differenced target variable
result = adfuller(df['Close'], autolag='AIC')
print('NON-DIFFERENCED TARGET VARIABLE')
print(f'ADF Statistic: {result[0]}')
print(f'p-value: {result[1]}')

print(' ')
print(' ')

# ADF Test for Differenced target variable
result = adfuller(df['Close'].diff().dropna(), autolag='AIC')
print('DIFFERENCED TARGET VARIABLE')
print(f'ADF Statistic: {result[0]}')
print(f'p-value: {result[1]}')
```

executed in 252ms, finished 19:06:30 2021-06-15

```
NON-DIFFERENCED TARGET VARIABLE
ADF Statistic: 1.0029061147236595
p-value: 0.9942965169904011


DIFFERENCED TARGET VARIABLE
ADF Statistic: -9.300900887869764
p-value: 1.1132363356594116e-15
```

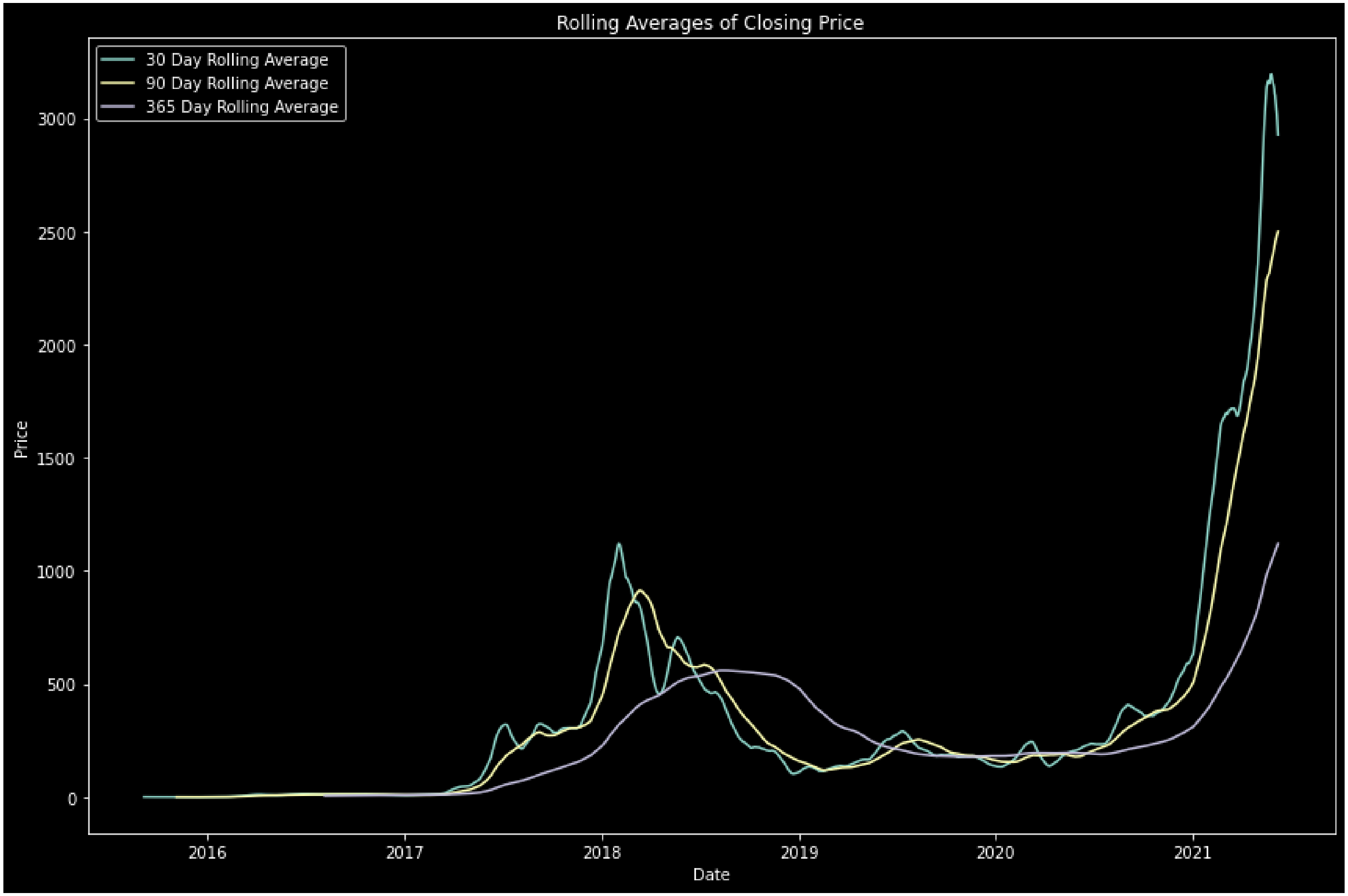
- A first-order difference is enough to stationarize the data

4.0.7 Rolling Averages

In [112]:

```python
fig, ax = plt.subplots(figsize=(12,8))
df_30d_rol = df['Close'].rolling(window = 30).mean()
df_90d_rol = df['Close'].rolling(window = 90).mean()
df_365d_rol = df['Close'].rolling(window = 365).mean()
ax.plot(df_30d_rol, label='30 Day Rolling Average')
ax.plot(df_90d_rol, label='90 Day Rolling Average')
ax.plot(df_365d_rol, label='365 Day Rolling Average')
ax.set_xlabel('Date')
ax.set_ylabel('Price')
ax.set_title('Rolling Averages of Closing Price')
plt.legend()
plt.tight_layout()
plt.savefig('rolling_averages')
```

executed in 497ms, finished 01:38:34 2021-06-16



Rolling Averages of Closing Price
30 Day Rolling Average
90 Day Rolling Average
365 Day Rolling Average
Price
3000
2500
2000
1500
1000
500
0
2016
2017
2018
2019
2020
2021
Date

In [111]:

```python
# # Full compiled graph of 30-day, 90-day, and 365-day rolling averages

# fig = go.Figure()



# df_30d_rol = df['Close'].rolling(window = 30).mean()
# df_90d_rol = df['Close'].rolling(window = 90).mean()
# df_365d_rol = df['Close'].rolling(window = 365).mean()
# fig.add_trace(go.Scatter(x=df.index, y=df_30d_rol, mode='lines', name=f'30d Close'))
# fig.add_trace(go.Scatter(x=df.index, y=df_90d_rol, mode='lines', name=f'90d Close'))
# fig.add_trace(go.Scatter(x=df.index, y=df_365d_rol, mode='lines', name=f'365d Close'))

# fig.update_layout(
# title='Price Data',
# xaxis_title='Date',
# yaxis_title='Price',
# legend_title='Legend')
# fig.show()
```

executed in 8ms, finished 01:38:24 2021-06-16

The rolling averages calculated from three different windows (30, 90, 365) provide some more insight to the data. As the window increases in size, the rolling averages' values have very different values during the highly volatile periods of the price of Ethereum. This volatility resulted in each of these periods having wildly different minimum and maximum values, which results in rolling averages that also different by quite a lot. Unsurprisingly, the 30-day and 90-day rolling averages were the most closely related, especially during the first period of steep upwards trend. The prices did not reach magnitude differences during these windows that warranted such a drastic rolling average difference. However, at the end of our time period, the rolling averages end up differing in value by almost $500, which goes to show the extreme volatility that Ethereum experienced during this time period (the most recent months when Ethereum had a meteoric rise). In short summary, the 365-day moving average had the lowest average value because it generalized the most volatility, however its final value was very below the true price. The 30-day moving average had the highest value because it strongly accounted for the high volatility, and its final value was a little higher than the true price (the extreme upper values pulled the average upwards). The 90-day moving average was the closest to the true price, showing that it both accounted for and generalized the volatility the best of the three windows!

4.0.8 Seasonality

In [98]:

```python
# Investigate Monthly Seasonality per Year
monthly_df = pd.DataFrame(df.resample('MS').mean())
fig, ax = plt.subplots(3,2, figsize=(12, 8))

ax[0][0].plot(monthly_df['Close']['2015'])
ax[0][0].set_title('2015')
ax[0][0].set_xlabel('Month')
ax[0][0].set_ylabel('Price')
ax[0][0].set_xticklabels(labels=monthly_df['Close']['2015'].index.month,rotation=45)

ax[1][0].plot(monthly_df['Close']['2016'])
ax[1][0].set_title('2016')
ax[1][0].set_xlabel('Month')
ax[1][0].set_ylabel('Price')
ax[1][0].set_xticklabels(labels=monthly_df['Close']['2016'].index.month,rotation=45)

ax[2][0].plot(monthly_df['Close']['2017'])
ax[2][0].set_title('2017')
ax[2][0].set_xlabel('Month')
ax[2][0].set_ylabel('Price')
ax[2][0].set_xticklabels(labels=monthly_df['Close']['2017'].index.month,rotation=45)


ax[0][1].plot(monthly_df['Close']['2018'])
ax[0][1].set_title('2018')
ax[0][1].set_xlabel('Month')
ax[0][1].set_ylabel('Price')
ax[0][1].set_xticklabels(labels=monthly_df['Close']['2018'].index.month,rotation=45)


ax[1][1].plot(monthly_df['Close']['2019'])
```

```
ax[1][1].set_title('2019')
ax[1][1].set_xlabel('Month')
ax[1][1].set_ylabel('Price')
ax[1][1].set_xticklabels(labels=monthly_df['Close']['2019'].index.month,rotation=45)


ax[2][1].plot(monthly_df['Close']['2020'])
ax[2][1].set_title('2020')
ax[2][1].set_xlabel('Month')
ax[2][1].set_ylabel('Price')
ax[2][1].set_xticklabels(labels=monthly_df['Close']['2020'].index.month,rotation=45)

plt.suptitle('Monthly Close Price Trends by Year')
plt.tight_layout()
plt.savefig('monthly_price_trends')
```

executed in 2.49s, finished 01:28:31 2021-06-16

There are no seasonal trends shown. Each year shows varying periods of time where the price and volume experienced both upwards and downwards trends. Since

4.0.9 Findings

Ethereum prices follow what is called a "cyclical trend", which means that it has trends however these trends so no specific pattern of repetition. To illustrate this, we can look at two of the graphs, which are both displayed above.

From the year 2015 to the first quarter of 2017, the price of Ethereum remained quite stationary, with a very strong rise starting between March and April, which led to a strong upwards trend that lasted throughout the rest of the year of 2017, bring the price to a maximum value of 826.82 by the end of the year. This constituted a 10,106 percent price increase from the minimum price of 8.17 in the year of 2017, which is by all standards a very strong upwards trend. The volume of trades also followed this trend quite closely, matching the sentiment idea that as an asset shoots up in price, more people attempt to join in on the ride, and hence more trades are made. After the year 2017, the price of Ethereum immediately started a strong downwards trend beginning in January of 2018, and by the end of 2018 the price had settled to a minimum value of 84.30, roughly a 94% drop from its all time high at the very beginning of 2018. Volume for the rest of 2018 remained on average higher than the two years afterwards and the year before because at first people were participating in frequent trades due to the meteoric rise in price, and then people continued to sell their coins over the year as the price tanked. From 2019 to mid-2020, the price once again mostly resumed the stationary trend that it had exemplified from 2015 to about a quarter of the way through 2017, indicating that perhaps people lost interest in the Ethereum block-chain, doubted its potential, or simply moved on to different investments. There was a sharp rise in prise to a little over 250 during 2019, but it just as quickly fell back to close to the minimum value of that year, failing to breakout of its strong downwards trend. The volume from 2019 to mid-2020 would never drop to the levels seen before the coin's meteoric rise, most likely because such a note-worthy event put Ethereum on the map permanently. During 2019, there was a sharp rise and fall in volume that mirrored the trend of the quick rise and fall of price during that year. 2019-2021 would be the period of time when Ethereum would consistently reflect a yearly upwards trend. Volume was higher than its ever been, and the price rose to an unprecedented level of roughly 4000. During this upwards trend, there were several downwards trends that occured during certain months of the years. They seemed to be relatively random, with no predictability in their occurences, highlighting the unstationarity of the price of Ethereum, and also the idea that the price follows a "cyclical trend". There are very clear bull and bear markets, however the trickly part is timing these.

5 Modeling

5.1 Scale the Data

We are going to want to scale the data because of the massive magnitude differences between values. This will most likely improve the accuracy of our forecast

In [30]:

```
from sklearn.preprocessing import MinMaxScaler
ss = MinMaxScaler()
scaled_data = pd.DataFrame(ss.fit_transform(df), columns=df.columns, index=df.index)
```

executed in 18ms, finished 17:26:39 2021-06-16

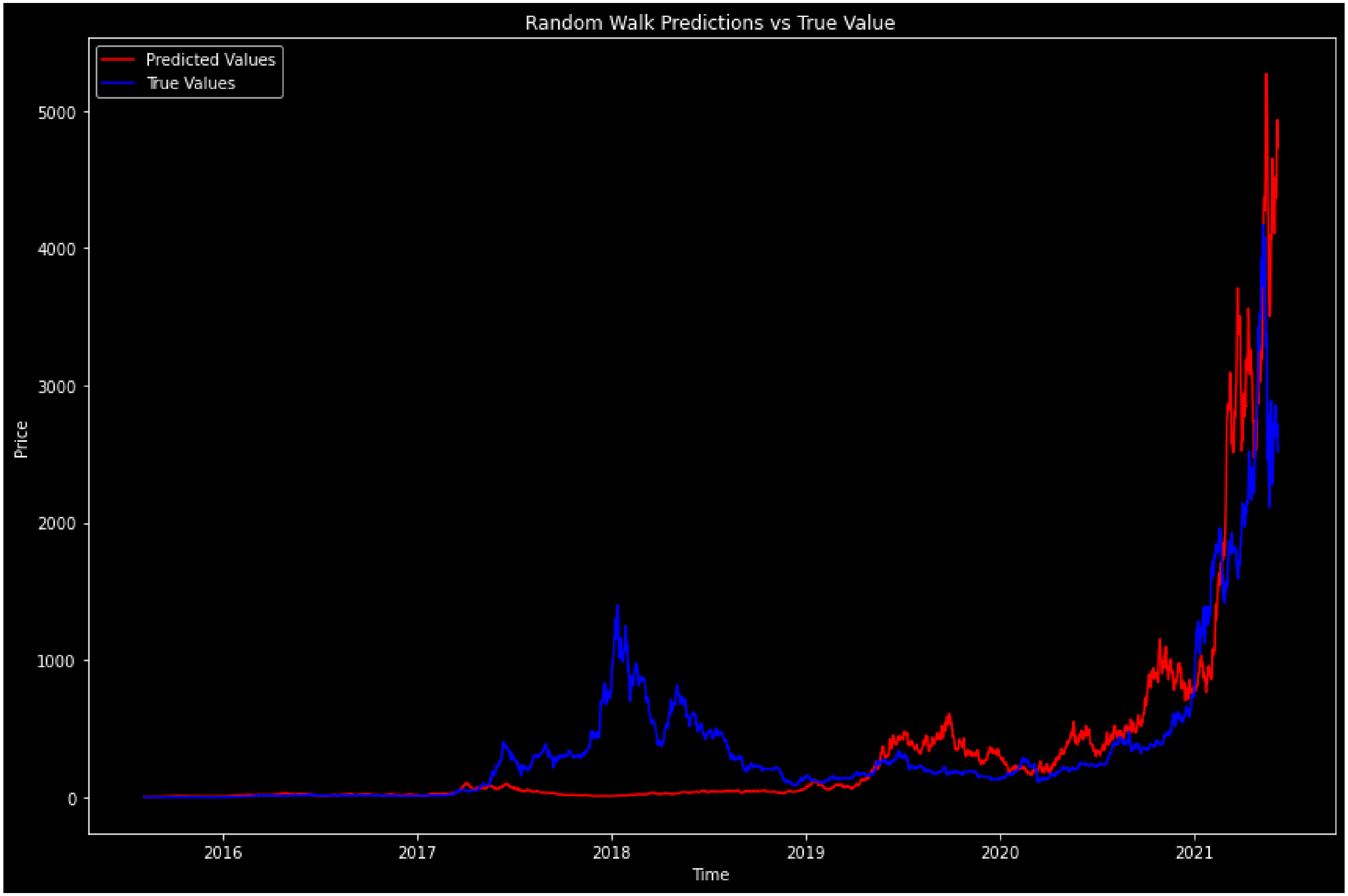
5.2 Random-Walk

In [28]:

```
## Walk
rwdata = pd.DataFrame(df['Close'], columns=['Close'])
rwdata['change'] = df['Close'].pct_change()
mean = rwdata['change'][1:].mean()
sd = rwdata['change'][1:].std()

## Predict
model = {}
model['Prediction'] = [rwdata['Close'][0]]
for time in range(1, len(rwdata)):
    old = model['Prediction'][time -1]
    new_price = old*(1+ mean) + old*sd*np.random.normal(0,1)
    model['Prediction'].append(new_price)

## Plot
rwdf = pd.DataFrame(model, index=rwdata.index)

fig, ax = plt.subplots(figsize=(12,8))
ax.plot(rwdf, label='Predicted Values', color='Red')
ax.plot(rwdata['Close'], label='True Values', color='Blue')

plt.xlabel('Time')
plt.ylabel('Price')
plt.title('Random Walk Predictions vs True Value')
plt.legend()
plt.tight_layout()

rmse = math.sqrt(mean_squared_error(rwdf, rwdata['Close']))
print(f'RMSE = {rmse}')
```

executed in 233ms, finished 17:14:30 2021-06-16

```
RMSE = 383.27178854376854
```



The model was run multiple times, in an attempt to aquire the best possible model for the problem. The best random-walk achieved had an RMSE of 323.097. Since the business strategy we are focusing on is day-trading, it is preferable to have tighter margins of error, because we are not holding for long periods of time and therefore a wrong guess affects our success more strongly.

5.3 ARIMA Model

In [31]:

```
### Train-Test-Split the Non-Scaled Data
y_train = df['Close'][:'2019-06-13']
y_test = df['Close']['2019-06-14':]
x_train = df.index[:1407]
x_test = df.index[1407:]

###
y_train_scaled = scaled_data['Close'][:'2019-06-13']
y_test_scaled = scaled_data['Close']['2019-06-14':]
```

executed in 29ms, finished 17:26:46 2021-06-16

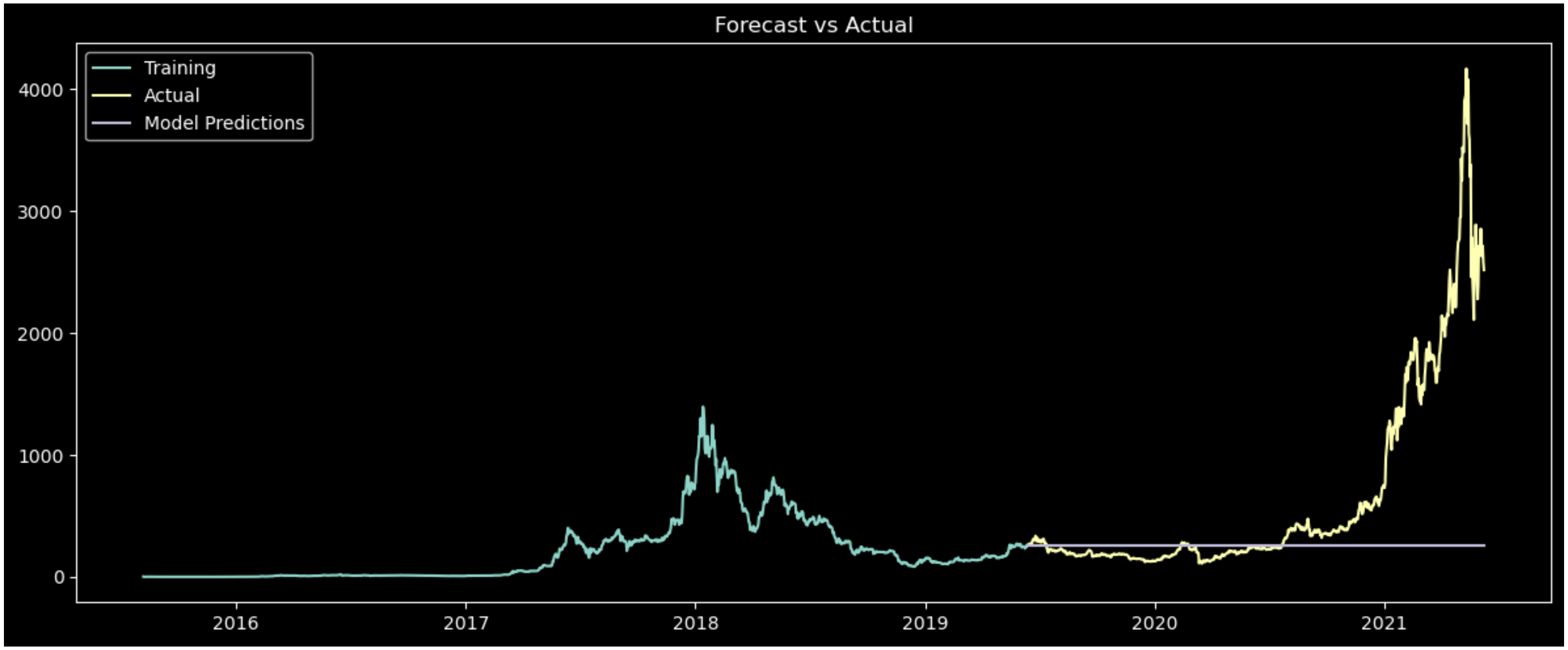
In [114]:

```python
from statsmodels.tsa.stattools import acf
from statsmodels.tsa.arima.model import ARIMA

# Build a 1,1,1 ARIMA model
p, d, q = 1, 1, 1
model = ARIMA(y_train, order=(p, d, q))
model_fit = model.fit()


### Model Summary
print(model_fit.summary())


### Forecast
forecast, se, conf = model_fit.forecast(3, alpha=0.05)


### Convert to series so we can plot the data
forecast_series = pd.Series(forecast, index=y_test.index)


### Plot
plt.figure(figsize=(12,5), dpi=100)
plt.plot(y_train, label='Training')
plt.plot(y_test, label='Actual')
plt.plot(forecast_series, label='Model Predictions')
plt.title('Forecast vs Actual')
plt.legend(loc='upper left')
plt.tight_layout()
plt.savefig('arima')
```

```python
rmse = math.sqrt(mean_squared_error(forecast_series, y_test))
print(f' RMSE = {rmse}')
```

executed in 894ms, finished 01:39:24 2021-06-16

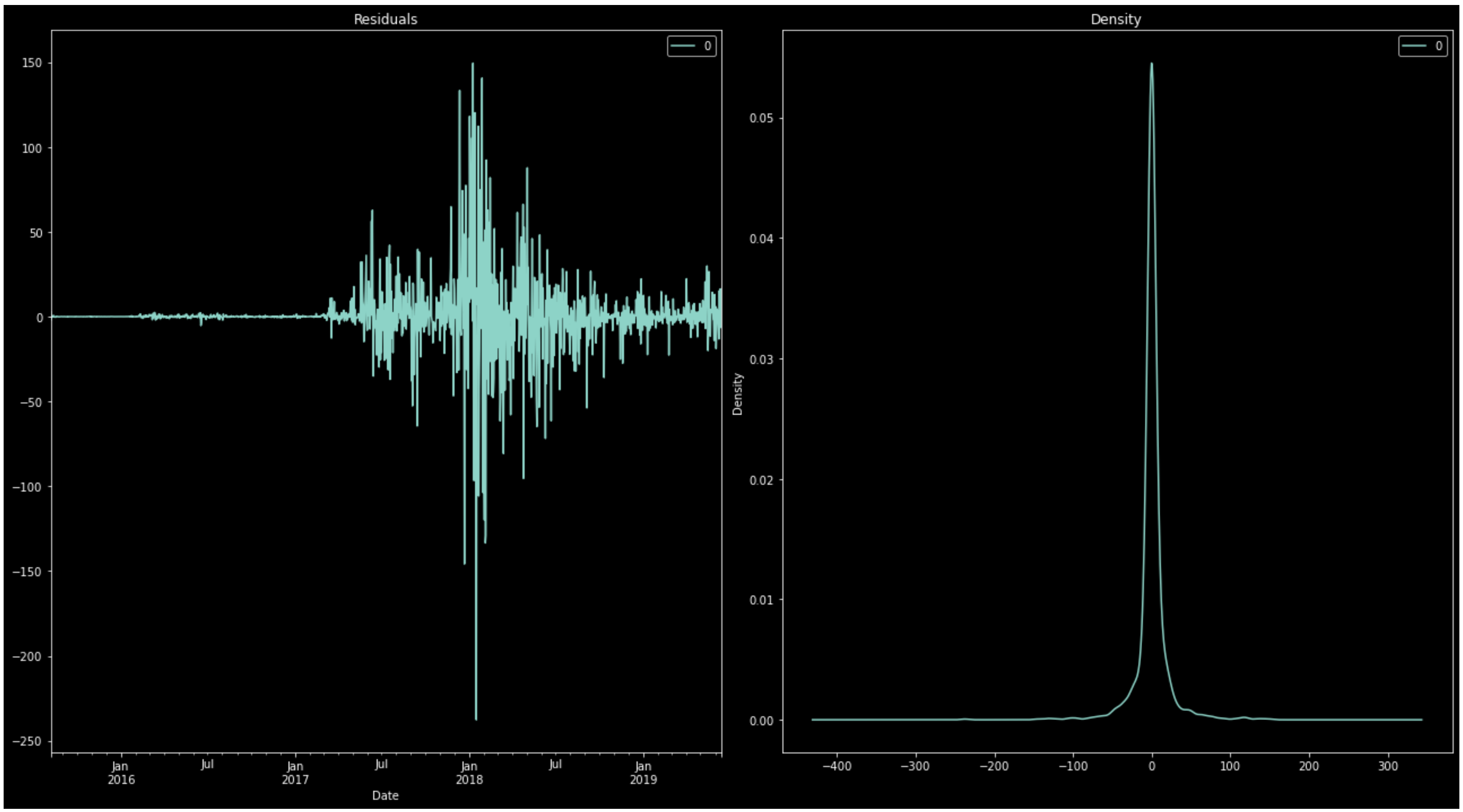
```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  Close   No. Observations:                 1407
Model:                 ARIMA(1, 1, 1)   Log Likelihood               -6277.836
Date:                Wed, 16 Jun 2021   AIC                          12561.672
Time:                        01:39:23   BIC                          12577.417
Sample:                    08-07-2015   HQIC                         12567.557
                         - 06-13-2019
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.8868      0.031    -28.690      0.000      -0.947      -0.826
ma.L1          0.9183      0.027     33.923      0.000       0.865       0.971
sigma2       442.3797      4.649     95.156      0.000     433.268     451.492
===================================================================================
Ljung-Box (L1) (Q):                   0.44   Jarque-Bera (JB):             38580.87
Prob(Q):                              0.51   Prob(JB):                         0.00
Heteroskedasticity (H):             877.14   Skew:                            -0.68
Prob(H) (two-sided):                  0.00   Kurtosis:                        28.63
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
 RMSE = 915.887738657556
```


Forecast vs Actual
Training
Actual
Model Predictions
4000
3000
2000
1000
0
2016
2017
2018
2019
2020
2021

In [101]:

```python
# Calculate Residuals
residuals = pd.DataFrame(model_fit.resid)

# Plot residuals
fig, ax = plt.subplots(1,2, figsize=(18,10))
residuals.plot(title="Residuals", ax=ax[0])
residuals.plot(kind='kde', title='Density', ax=ax[1])

plt.tight_layout()
```

executed in 732ms, finished 01:33:52 2021-06-16



The ARIMA model performed poorly for the data provided. This can almost certainly be attributed to the exaggerated volatility of Ethereum prices. The period of time that ARIMA was trained on showed an interesting trend. The price remained low, then spiked to a value that was much higher than before, and just as quickly fell down to a very low value again and remained there for quite some time. In other words, it was relatively stationary, then had a steep upwards trend, a steep downwards trend, and then remained relatively stationary again. The two main determinants of ARIMA predicitons, past values and moving average, are very hard to predict upon because thei values vary by so much. In order to try and improve my model, I will be using the "pmdarima" package to try and optimize the hyperparamters of the ARIMA model.

The RMSE of the model was 915.88, a very poor metric, and significantly worse than our random-walk metric measurement.

5.4 Auto-ARIMA

In [115]:

```
import pmdarima as pm
model = pm.auto_arima(y_train, start_P=0, d=1, start_q=0, max_p=5, max_d=5, max_q=5,
                      D=1, start_Q=0, max_D=5, max_Q=5, m=90, seasonal=False, error_action='warn',
                      trace=True, supress_warnings=True, stepwise=True, random_state=20, n_fits=30)
model.summary()
```

executed in 8.49s, finished 01:39:56 2021-06-16

```
Performing stepwise search to minimize aic
 ARIMA(2,1,0)(0,0,0)[0] intercept   : AIC=12567.097, Time=0.19 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=12566.076, Time=0.04 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=12565.161, Time=0.08 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=12565.123, Time=0.20 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=12564.178, Time=0.04 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=12563.573, Time=0.77 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=12564.978, Time=1.13 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=12564.998, Time=1.39 sec
 ARIMA(0,1,2)(0,0,0)[0] intercept   : AIC=12567.055, Time=0.30 sec
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=12566.273, Time=1.48 sec
 ARIMA(1,1,1)(0,0,0)[0]             : AIC=12561.672, Time=0.41 sec
 ARIMA(0,1,1)(0,0,0)[0]             : AIC=12563.216, Time=0.08 sec
 ARIMA(1,1,0)(0,0,0)[0]             : AIC=12563.255, Time=0.03 sec
 ARIMA(2,1,1)(0,0,0)[0]             : AIC=12563.072, Time=0.52 sec
 ARIMA(1,1,2)(0,0,0)[0]             : AIC=12563.093, Time=0.63 sec
 ARIMA(0,1,2)(0,0,0)[0]             : AIC=12565.150, Time=0.16 sec
 ARIMA(2,1,0)(0,0,0)[0]             : AIC=12565.191, Time=0.13 sec
 ARIMA(2,1,2)(0,0,0)[0]             : AIC=12564.348, Time=0.86 sec

Best model:  ARIMA(1,1,1)(0,0,0)[0]
Total fit time: 8.455 seconds
```

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	1407
Model:	SARIMAX(1, 1, 1)	**Log Likelihood**	-6277.836
Date:	Wed, 16 Jun 2021	**AIC**	12561.672
Time:	01:39:56	**BIC**	12577.417

Sample:	0	**HQIC**	12567.557
	- 1407		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	-0.8868	0.031	-28.690	0.000	-0.947	-0.826
ma.L1	0.9183	0.027	33.923	0.000	0.865	0.971
sigma2	442.3797	4.649	95.156	0.000	433.268	451.492

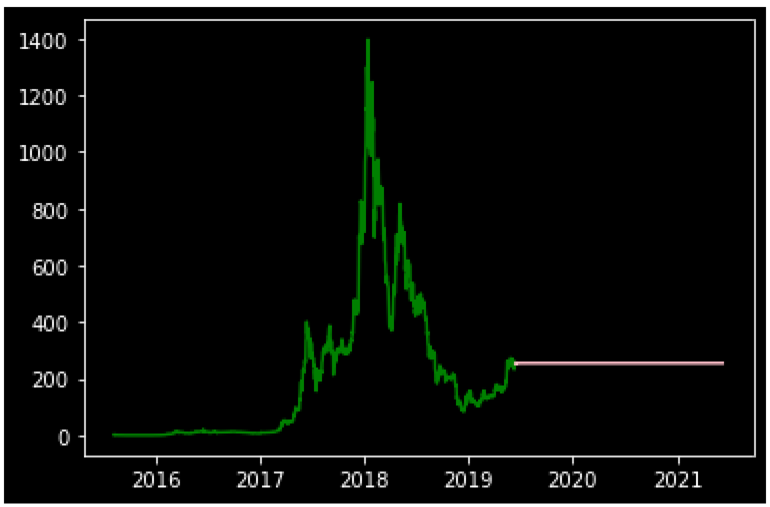
Ljung-Box (L1) (Q):	0.44	**Jarque-Bera (JB):**	38580.87
Prob(Q):	0.51	**Prob(JB):**	0.00
Heteroskedasticity (H):	877.14	**Skew:**	-0.68
Prob(H) (two-sided):	0.00	**Kurtosis:**	28.63

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

In [103]:

```python
# make your forecasts
prediction = pd.DataFrame(model.predict(n_periods=726), index=y_test.index)
# Visualize the forecasts (blue=train, green=forecasts)
x = np.arange(y_test.shape[0])
plt.plot(y_train, c='green')
plt.plot(prediction, c='pink')
plt.show()

rmse = math.sqrt(mean_squared_error(prediction, y_test))
print(f'RMSE = {rmse}')
```

executed in 208ms, finished 01:34:40 2021-06-16




```
RMSE = 915.5505585804128
```

The AUTO-ARIMA model was ran and forecasted. It performed marginally better than the ARIMA model, with an RMSE of 915.55 rather than an RMSE of 915.88. This value is still significant worse than the metric calculated from our Random-Walk model. Since these metrics are so poor, we are going to move on to different model.

5.5 Sarima and One-Step-Ahead Model

In [32]:

```python
### Train-Test-Split the Non-Scaled Data
y_train = df['Close'][:'2019-06-13']
y_test = df['Close']['2019-06-14':]
x_train = df.index[:1407]
x_test = df.index[1407:]

###
y_train_scaled = scaled_data['Close'][:'2019-06-13']
y_test_scaled = scaled_data['Close']['2019-06-14':]
```

executed in 9ms, finished 17:26:56 2021-06-16

5.5.1 Grid Search

In [33]:

```
import itertools
y=df['Close'].diff().dropna()
def sarima_grid_search(y,seasonal_period):
    p = d = q = range(0, 5)
    pdq = list(itertools.product(p, d, q))
    seasonal_pdq = [(x[0], x[1], x[2],seasonal_period) for x in list(itertools.product(p, d, q))]

    best_aic = float('+inf')



    for param in pdq:
        for s_param in seasonal_pdq:
            try:
                mod = sm.tsa.statespace.SARIMAX(y, order=param, seasonal_order=s_param)

                results = mod.fit()

                if results.aic < best_aic:
                    best_aic = results.aic
                    optimal_param = param
                    s_optimal_param = s_param

                    print(f'SARIMA{param}x{s_param} - AIC:{results.aic}')
            except:
                continue
    print(f'Optimal Parameters for SARIMA Model: SARIMA{optimal_param}x{s_optimal_param} - AIC:{best_aic}')
```

executed in 19ms, finished 17:26:58 2021-06-16

In [*]:

```
### Grid Search
sarima_grid_search(y,12)
```

execution queued 17:27:10 2021-06-16

```
SARIMA(0, 0, 0)x(0, 0, 0, 12) - AIC:22443.34146633209
SARIMA(0, 0, 0)x(0, 0, 1, 12) - AIC:22419.516519416262
SARIMA(0, 0, 0)x(0, 1, 1, 12) - AIC:22380.867514605157
SARIMA(0, 0, 0)x(0, 1, 2, 12) - AIC:22352.020220710816
SARIMA(0, 0, 0)x(1, 1, 2, 12) - AIC:22351.1112859146
SARIMA(0, 0, 0)x(2, 1, 4, 12) - AIC:22350.21073754495
SARIMA(0, 0, 0)x(2, 4, 1, 12) - AIC:412.05080220139274
```

5.5.2 Model Predictions

In [29]:

```python
def sarima_and_osa(series, order, order_season, prediction_date):

    ### Train model
    model = sm.tsa.statespace.SARIMAX(series, order=order, order_season=order_season, trend='c')
    results = model.fit()
    print(results.summary().tables[1])

    ### RMSE for One-Step_Ahead Forecast
    forecast = results.get_prediction(start=(pd.to_datetime(prediction_date)), dynamic=False)

    ### RMSE and Plot

    mean_forecast = forecast.predicted_mean
    confidence_intervals = forecast.conf_int()

    mse = ((mean_forecast - y_test) ** 2).mean()
    print(f'The Sarima RMSE for the One-Step-Ahead Forecast is {round(np.sqrt(mse), 2)}')

    ax = series.plot(label='Observed')
    mean_forecast.plot(ax=ax, label='One-step Ahead Model Predictions of Data', alpha=.7, figsize=(12, 8))

    ax.set_xlabel('Date')
    ax.set_ylabel('Close Price')
    plt.xlim('2021-01-01', x_test[-1])
    plt.legend()
    plt.show()
#     plt.savefig('one_step_ahead')

    ### Root-Mean-Squared-Error of Dynamic Forecast
    pred_dynamic = results.get_prediction(start=pd.to_datetime(prediction_date), dynamic=True, full_results=
    pred_dynamic_ci = pred_dynamic.conf_int()
```

```python
    forecast_dynamic = pred_dynamic.predicted_mean
    mse_dynamic = ((forecast_dynamic - y_test) ** 2).mean()
    print(f'The Sarima RMSE for the Dynamic Model Predictions is {round(np.sqrt(mse_dynamic), 2)}')

    ### Plot Dynamic Forecast
    ax = y_train.plot(label='Observed')
    forecast_dynamic.plot(label='Dynamic Model Predictions of Data', ax=ax, figsize=(12, 8))
    y_test.plot(label='True Values', ax=ax, figsize=(12,8))
    ax.set_xlabel('Date')
    ax.set_ylabel('Close Price')

    plt.legend()
    plt.show()
#     plt.savefig('sarimax')
    return (results)
```

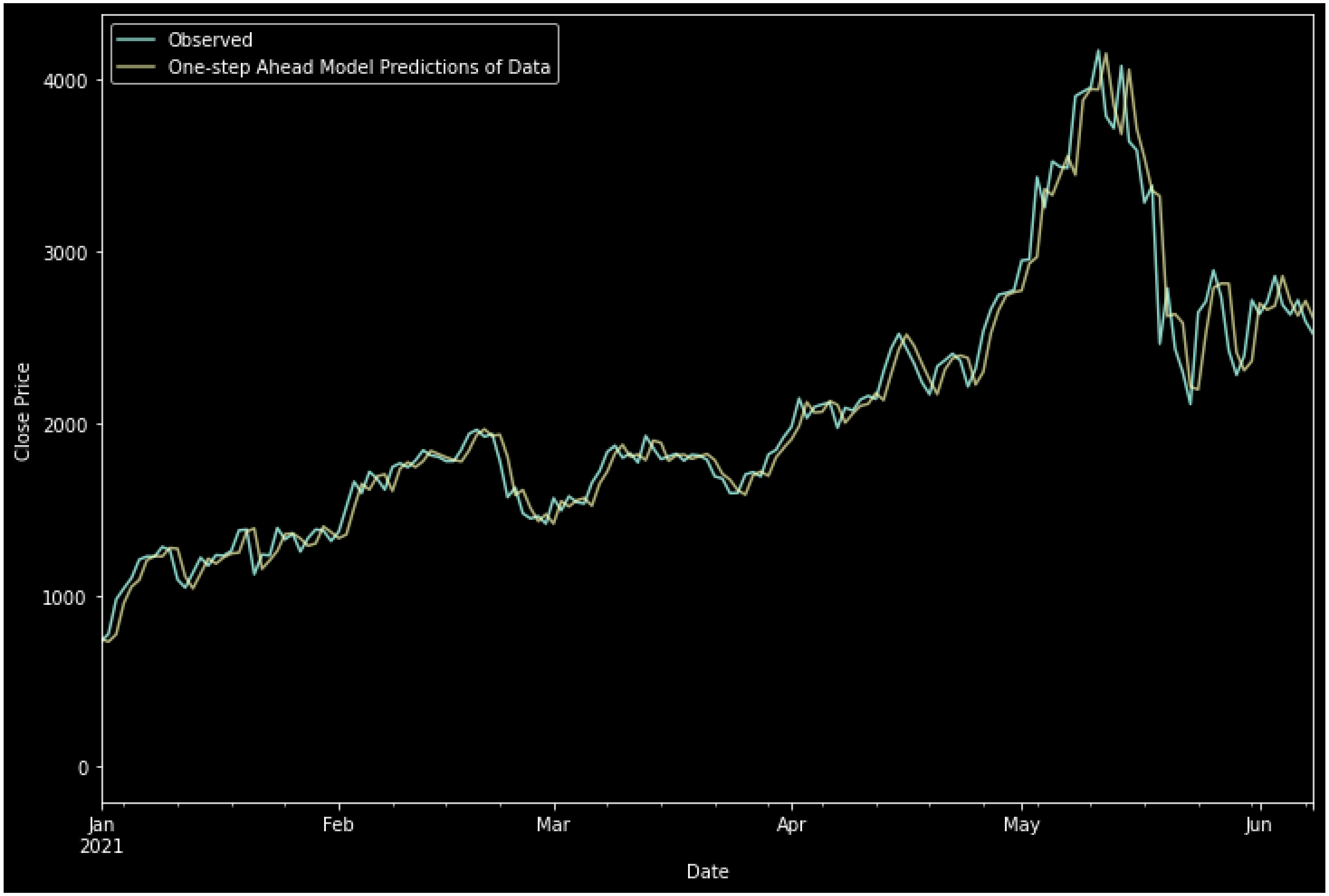
executed in 14ms, finished 10:40:04 2021-06-16

In [30]:

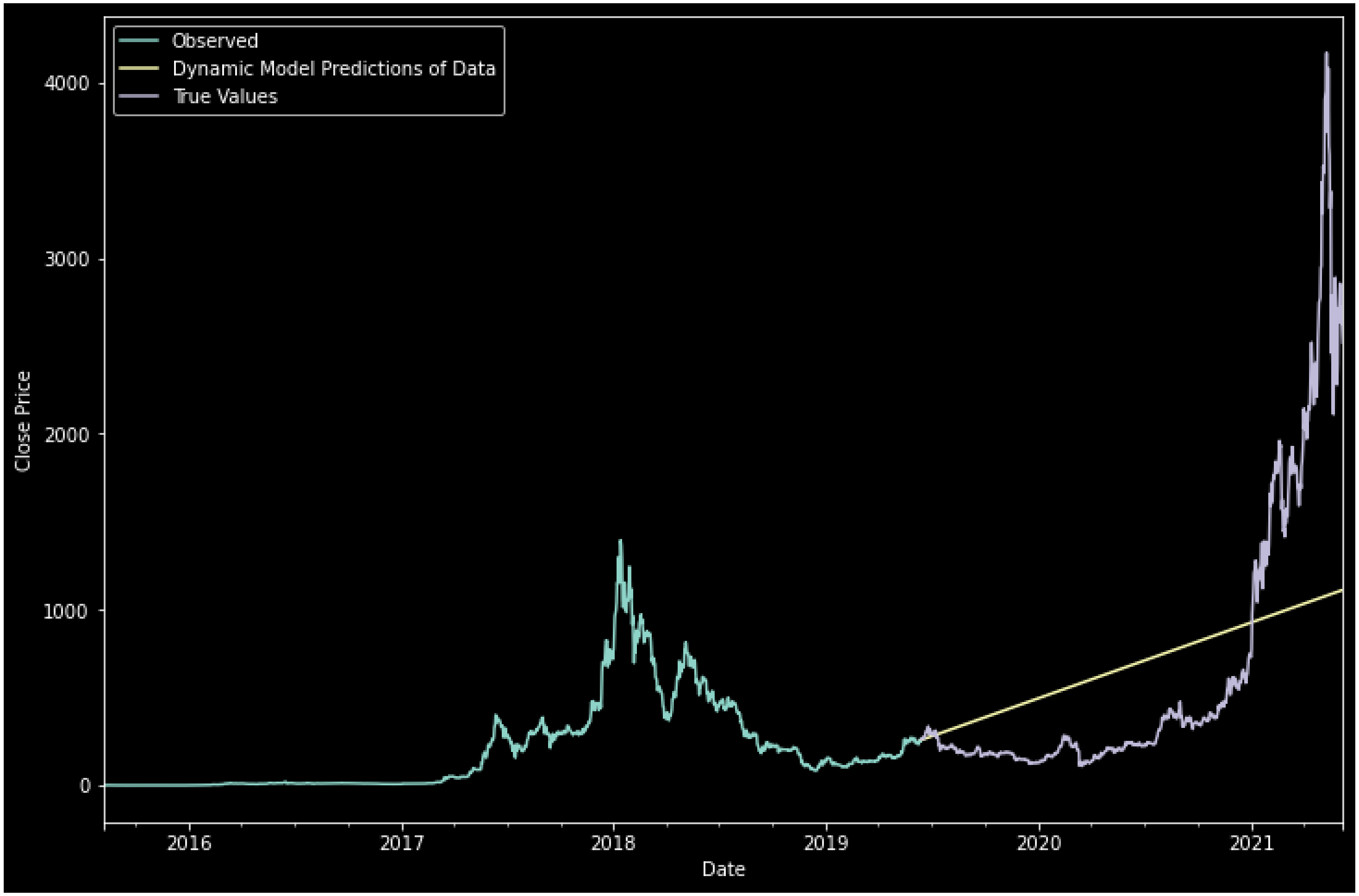
```python
series = df['Close']
model = sarima_and_osa(series, (1, 1, 1), (1, 1, 1, 90), x_test[0])
plt.savefig('bla')
```

executed in 1.53s, finished 10:40:06 2021-06-16

```
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept      2.1431      1.733      1.236      0.216      -1.254       5.541
ar.L1         -0.8185      0.010    -80.502      0.000      -0.838      -0.799
ma.L1          0.6901      0.015     47.243      0.000       0.661       0.719
sigma2      2076.8164     10.583    196.246      0.000    2056.075    2097.558
==============================================================================
The Sarima RMSE for the One-Step-Ahead Forecast is 72.08
```




```
The Sarima RMSE for the Dynamic Model Predictions is 658.49
```



<Figure size 432x288 with 0 Axes>

The one-step ahead model performed very well, with an RMSE of 72.12, the lowest of all of the models so far. This model's predictions were very close to that of the actual values, and utilization of this model for day trading would be highly effective. However, that is where this model's utilization ability stops. If there is an intention to use the model to predict farther in the future (30 days, 90 days, etc), a different model might be more effective.

The original SARIMAX model performed very poorly, with an RMSE value very similar to that of the ARIMA models. A gridsearch was done on the model to try and optimize the hyperparameters, and a much better model was constructed as a result.

The SARIMAX model performed much better than the ARIMA model, with an RMSE of 658.58.

5.6 Deep Learning

5.6.1 LSTM

5.6.1.1 LSTM With Manual Timeseries Sampling

[...]

5.6.1.2 LSTM with TimeseriesGenerator (Best results of the two)

In [3]:

```
data = np.asarray(df['Close']).reshape(-1,1)
```

executed in 7ms, finished 17:08:52 2021-06-16

In [4]:

```
# Scale the data
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
data = scaler.fit_transform(data)

# split into train and test sets
train_size = int(len(data) * 0.6)
test_size = len(data) - train_size

train = data[0:train_size,:]
test = data[train_size:len(data),:]
```

executed in 19ms, finished 17:08:52 2021-06-16

In [5]:

```
# Use TimeseriesGenerator to create the samples
from keras.preprocessing.sequence import TimeseriesGenerator
n_input = 90

train_data = TimeseriesGenerator(train, train,
    length=n_input,
    batch_size=3)

test_data = TimeseriesGenerator(test, test,
    length=n_input,
    batch_size=1)
```

executed in 4.71s, finished 17:08:58 2021-06-16

In [6]:

```python
# Create the model!
from keras.models import Sequential
from keras.layers import Dense, LSTM
model = Sequential()
model.add(LSTM(units=32, return_sequences=True,
                    input_shape=(90,1), dropout=0.2))
model.add(LSTM(units=32, return_sequences=True,
                    dropout=0.2))
model.add(LSTM(units=32, dropout=0.2))
model.add(Dense(units=1))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['MeanSquaredError'])

# Summarize the model
model.summary()
```

executed in 2.42s, finished 17:09:00 2021-06-16

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 90, 32)            4352
_________________________________________________________________
lstm_1 (LSTM)                (None, 90, 32)            8320
_________________________________________________________________
lstm_2 (LSTM)                (None, 32)                8320
_________________________________________________________________
dense (Dense)                (None, 1)                 33
=================================================================
Total params: 21,025
Trainable params: 21,025
Non-trainable params: 0
_________________________________________________________________
```

In [7]:

```
# Run the model
history = model.fit_generator(train_data, epochs=12)
```

executed in 3m 24s, finished 17:12:24 2021-06-16

```
WARNING:tensorflow:From <ipython-input-7-14853c05db4a>:2: Model.fit_generator (from tensorflow.python.keras.engine.training) is deprecated
and will be removed in a future version.
Instructions for updating:
Please use Model.fit, which supports generators.
Epoch 1/12
397/397 [==============================] - 18s 44ms/step - loss: 0.0011 - mean_squared_error: 0.0011
Epoch 2/12
397/397 [==============================] - 17s 43ms/step - loss: 6.8501e-04 - mean_squared_error: 6.8501e-04
Epoch 3/12
397/397 [==============================] - 17s 42ms/step - loss: 4.1154e-04 - mean_squared_error: 4.1154e-04
Epoch 4/12
397/397 [==============================] - 17s 42ms/step - loss: 3.9977e-04 - mean_squared_error: 3.9977e-04
Epoch 5/12
397/397 [==============================] - 16s 41ms/step - loss: 4.1705e-04 - mean_squared_error: 4.1705e-04
Epoch 6/12
397/397 [==============================] - 16s 41ms/step - loss: 3.8272e-04 - mean_squared_error: 3.8272e-04
Epoch 7/12
397/397 [==============================] - 16s 40ms/step - loss: 3.8784e-04 - mean_squared_error: 3.8784e-04
Epoch 8/12
397/397 [==============================] - 16s 41ms/step - loss: 3.6302e-04 - mean_squared_error: 3.6302e-04
Epoch 9/12
397/397 [==============================] - 16s 41ms/step - loss: 4.0844e-04 - mean_squared_error: 4.0844e-04
Epoch 10/12
397/397 [==============================] - 16s 41ms/step - loss: 4.8079e-04 - mean_squared_error: 4.8079e-04
Epoch 11/12
397/397 [==============================] - 16s 41ms/step - loss: 4.1651e-04 - mean_squared_error: 4.1651e-04
Epoch 12/12
397/397 [==============================] - 17s 42ms/step - loss: 3.8736e-04 - mean_squared_error: 3.8736e-04
```

In [8]:

```
# Predict the data using the model!
train_pred = model.predict_generator(train_data)
test_pred = model.predict_generator(test_data)

# Inverse the transformation we did earlier so we have the true values of the predictions
train_pred = scaler.inverse_transform(train_pred)
test_pred = scaler.inverse_transform(test_pred)
```

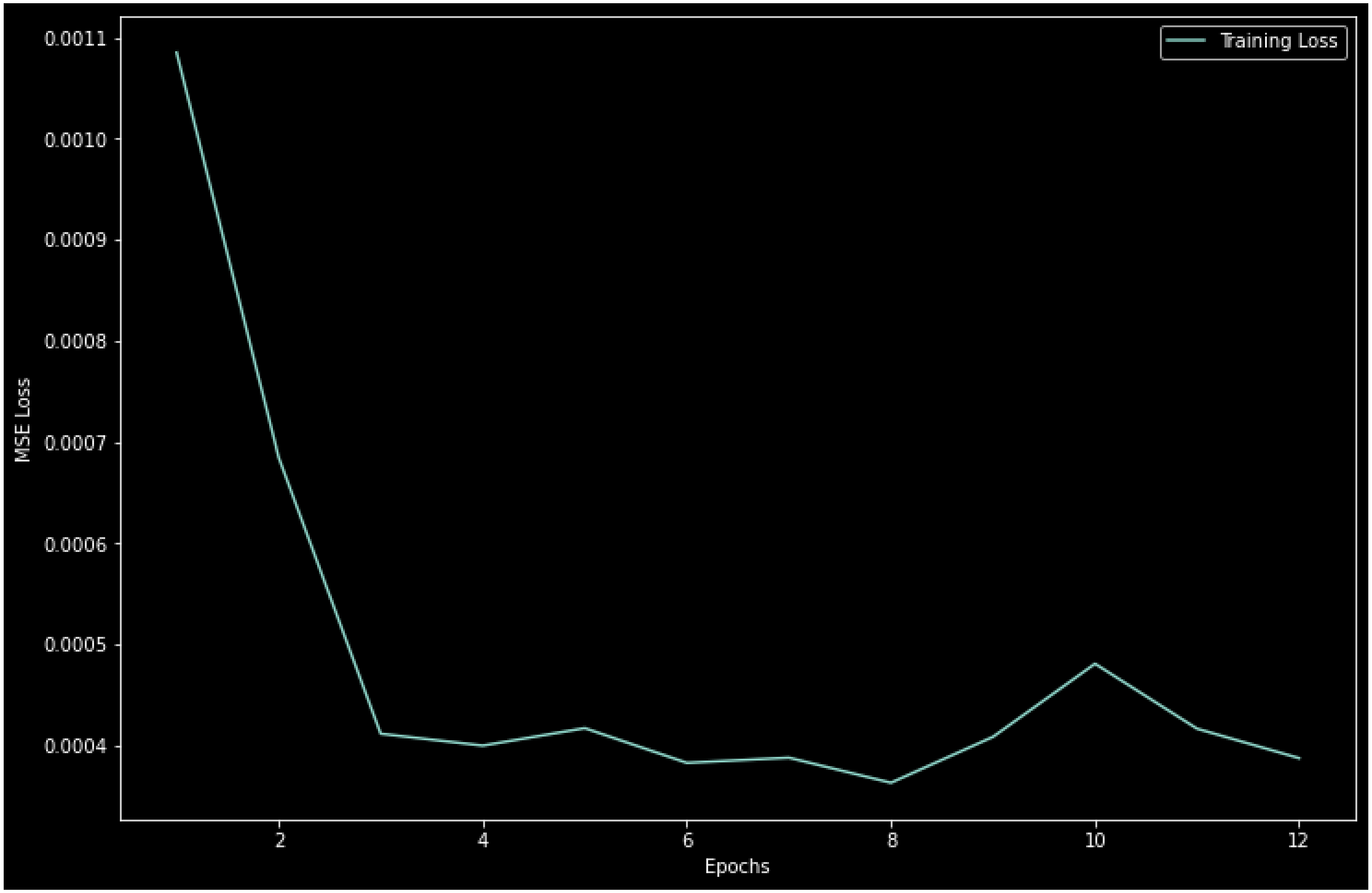
executed in 8.89s, finished 17:12:33 2021-06-16

```
WARNING:tensorflow:From <ipython-input-8-7144852fdcb9>:2: Model.predict_generator (from tensorflow.python.keras.engine.training) is deprec
ated and will be removed in a future version.
Instructions for updating:
Please use Model.predict, which supports generators.
```

In [9]:

```
loss = history.history['loss']
epochs = range(1, 13)
plt.figure(figsize=(12,8))
plt.plot(epochs, loss)
plt.legend(['Training Loss'])
plt.xlabel('Epochs')
plt.ylabel('MSE Loss')
plt.show();
```

executed in 204ms, finished 17:12:34 2021-06-16



In [10]:

```
def get_y_from_generator(gen):
    '''
    Get all targets y from a TimeseriesGenerator instance.
    '''
    y = None
    for i in range(len(gen)):
        batch_y = gen[i][1]
        if y is None:
            y = batch_y
        else:
            y = np.append(y, batch_y)
    y = y.reshape((-1,1))
    print(y.shape)
    return y
```

executed in 14ms, finished 17:12:34 2021-06-16

In [11]:

```
# Get the y values
train_output = get_y_from_generator(train_data)
test_output = get_y_from_generator(test_data)

# Reverse transform those
train_output = scaler.inverse_transform(train_output)
test_output = scaler.inverse_transform(test_output)
```

executed in 29ms, finished 17:12:34 2021-06-16

```
(1189, 1)
(764, 1)
```

In [14]:

```
fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(test_pred, label='Predictions', color='red')
ax.plot(test_output, label="True Values")
ax.set_xlabel('Date')
ax.set_ylabel('Price')
plt.legend()
ax.set_title('LSTM Model Prediction vs True Values')
plt.tight_layout()
plt.savefig('lstm')
```

executed in 341ms, finished 17:13:21 2021-06-16


LSTM Model Prediction vs True Values
Predictions
True Values
Price
Date
0
500
1000
1500
2000
2500
3000
3500
4000
0
100
200
300
400
500
600
700
800

In [13]:

```
rmse_df = pd.DataFrame(df['Close'], index=df.index[1369:])
rmse_df['Pred'] = test_pred
display(rmse_df)
rmse = math.sqrt(mean_squared_error(rmse_df['Close'], rmse_df['Pred']))
print(f'RMSE = {rmse}')
```

executed in 30ms, finished 17:12:35 2021-06-16

Date	Close	Pred
2019-05-07	169.80	173.876938
2019-05-08	170.95	174.680222
2019-05-09	170.29	175.524139
2019-05-10	173.14	176.281662
2019-05-11	194.30	177.128052
...	...	...
2021-06-04	2688.19	1725.112915
2021-06-05	2630.58	1731.218384
2021-06-06	2715.09	1729.373291
2021-06-07	2590.26	1729.859619
2021-06-08	2517.44	1724.409058

764 rows × 2 columns

```
RMSE = 360.43293570163524
```

The LSTM model went through two different iterations. At first, I used a singular LSTM layer, however I was

	dissapointed with my results. I then moved on too three different layers, which greatly improved my RMSE value and the forecasted trend looked much better when compared to the actual values. The RMSE of the LSTM model is 302.61, which is significantly better than that of the ARIMA and SARIMAX. However, it did not beat out the one-step-ahead model.