In [7]:
```
import pandas as pd
import numpy as np
import numpy.linalg as la
import datetime
import statsmodels.api as sm
import datetime
from tqdm import tqdm

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import matplotlib.pyplot as plt
from scipy.stats import zscore
from statsmodels.regression.linear_model import OLS
from statsmodels.tools.tools import add_constant

import plotly.express as px
import plotly.graph_objects as go
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)
```

In [8]:
```
class Functions:
    """
    A class that provides various functionalities for data analysis and visualization
    in the context of financial time series and regression analysis. It includes metho
    for plotting, principal component analysis, and regression.

    Methods:
    - time_plot(data, name, line_thickness=1, digits=4): Plots a time series line plot
    - eigenweights(df_eigen, eigen_weight, date, title): Plots eigenweights for a spec
    - plot_hist(data, title, x_label=None, y_label=None, bins=None): Plots a histogram
    - pca(data): Performs principal component analysis on the given data.
    - regress_part1(df, ticker): Performs the first part of regression analysis.
    - regress_part2(df): Performs the second part of regression analysis.
    """

    def __init__(self):
        pass

    @staticmethod
    def time_plot(data, name, line_thickness=1, digits=4, return_fig = False):
        """
        Plots a time series line plot for the given data with summary statistics.

        :param data: DataFrame containing the data to plot.
        :param name: Title of the plot.
        :param line_thickness: Thickness of the lines (default is 1).
        :param digits: Number of digits to display for mean and standard deviation (de
        """
        fig = go.Figure()

        # Melt the data
        melted_data = pd.melt(data, id_vars='startTime', value_vars=data.columns[:])

        # Create traces for each variable
        for variable in melted_data['variable'].unique():
            subset = melted_data[melted_data['variable'] == variable]
            fig.add_trace(go.Scatter(
                x=subset['startTime'],
```

```
            y=subset['value'],
            mode='lines+markers',
            name=variable,
            line=dict(width=line_thickness)  # Set line thickness here
        ))

    # Calculate mean and standard deviation
    mean_value = np.mean(melted_data['value'])
    std_value = np.std(melted_data['value'])

    # Create annotations for mean and std
    fig.add_annotation(
        go.layout.Annotation(
            text=f"Mean: {mean_value:.{digits}f}<br>Std Dev: {std_value:.{digits}f
            x=0.7,  # Adjust x-coordinate
            y=0.9,  # Adjust y-coordinate
            xref="paper",
            yref="paper",
            showarrow=False,
            align="left"
        )
    )

    # Customize layout
    fig.update_layout(
        title=name,
        xaxis=dict(title='Time'),
        yaxis=dict(title='Value'),
        showlegend=True,
        plot_bgcolor='lightgray',
        paper_bgcolor='white',
    )

    # Show the plot
    fig.show()
    if return_fig:
        return fig



@staticmethod
def eigenweights(df_eigen, eigen_weight, date, title):
    """
    Plots eigenweights for a specified eigenvalue and date.

    :param df_eigen: DataFrame containing eigenvalues.
    :param eigen_weight: Eigenvalue for which weights are to be plotted.
    :param date: Specific date for plotting.
    :param title: Title of the plot.
    """
    df_eigenWeight = df_eigen[df_eigen['index'] == eigen_weight]
    df_eigenWeight = df_eigenWeight[df_eigenWeight['startTime'] == date]
    del df_eigenWeight['startTime'], df_eigenWeight['index']
    df_eigenWeight = df_eigenWeight.T
    df_eigenWeight.rename(columns={df_eigenWeight.columns[0]: 'EigenWeight'}, inpl
    df_eigenWeight = df_eigenWeight.sort_values(by=['EigenWeight'], ascending=Fals
    df_eigenWeight.dropna(inplace=True)
    fig = px.line(df_eigenWeight)
    fig.update_layout(title_text=title)
    fig.show()
```

```
    @staticmethod
    def plot_hist(data, title, x_label=None, y_label=None, bins=None, return_fig = Fal
        """
        Plots a histogram for the given data with customization options and summary st

        :param data: DataFrame or Series containing the data to plot.
        :param title: Title of the plot.
        :param x_label: Label for the x-axis (optional).
        :param y_label: Label for the y-axis (optional).
        :param bins: Number of bins for the histogram (optional).
        """
        fig = go.Figure()

        # Create a histogram trace
        hist_trace = go.Histogram(
            x=data,
            nbinsx=bins  # Adjust the number of bins as needed
        )
        fig.add_trace(hist_trace)

        # Calculate mean and standard deviation
        mean_value = np.mean(data)
        std_value = np.std(data)

        # Create annotations for mean and std
        fig.add_annotation(
            go.layout.Annotation(
                text=f"Mean: {mean_value:.4f}<br>Std Dev: {std_value:.4f}",
                x=0.7,  # Adjust x-coordinate
                y=0.9,  # Adjust y-coordinate
                xref="paper",
                yref="paper",
                showarrow=False,
                align="left"
            )
        )

        # Customize layout
        fig.update_layout(
            title=title,
            xaxis_title=x_label,
            yaxis_title=y_label,
            showlegend=False,  # Hide legend for a histogram
            plot_bgcolor='lightgray',
            paper_bgcolor='white',
        )

        # Show the plot
        fig.show()

        if return_fig:
            return fig

    @staticmethod
    def pca(data):
        """
        Performs principal component analysis on the given data.

        :param data: DataFrame containing the data for PCA.
```

```
        :return: DataFrame containing the first two principal components.
        """
        scaler = StandardScaler()
        data = scaler.fit_transform(data)
        R = np.cov(data.T)
        evals, evecs = la.eigh(R)
        idx = np.argsort(evals)[::-1]
        evecs = evecs[:, idx]
        evals = evals[idx]
        evecs = evecs[:, :2]
        evecs = pd.DataFrame(evecs)
        return evecs

    @staticmethod
    def regress_part1(df, ticker):
        """
        Performs the first part of regression analysis.

        :param df: DataFrame containing the data.
        :param ticker: The ticker for which regression is to be performed.
        :return: Residual of the regression model.
        """
        df['Intercept'] = 1
        X = df[['Factor1', 'Factor2']]
        y = df[ticker]
        model = sm.OLS(y, X).fit()
        residual = model.resid
        return residual

    @staticmethod
    def regress_part2(df):
        """
        Performs the second part of regression analysis.

        :param df: DataFrame containing the data.
        :return: DataFrame containing regression coefficients and residual variance.
        """
        df['Intercept'] = 1
        X = df[['cum_sum', 'Intercept']]
        y = df['cum_sum_shift']
        model = sm.OLS(y, X).fit()
        coef = model.params
        resid = (model.resid).var()
        df1 = pd.DataFrame(data=coef.values, index=coef.index).T
        df1['residual_var'] = resid
        df1 = df1.T
        return df1
    @staticmethod
    def scale_weights(df, weight_column, scaled_column):
        mms = MinMaxScaler()
        df[scaled_column] = mms.fit_transform(df[[weight_column]])
        return df
    @staticmethod
    def calculate_eigen_portfolio(df, return_column, weight_column, eigen_column):
        df[eigen_column] = df[return_column] * df[weight_column]
        return df

    def cumulative_returns(df, columns):
        for column in columns:
            total_column = f"total_{column}"
```

```
            cum_column = f"cum_{column}"
            df[total_column] = 1 + df[column]
            df[cum_column] = df[total_column].cumprod()
        return df
```

In [9]:
```
class CryptoDataProcessor:
    """
    A class for processing cryptocurrency data, including merging price and ticker dat
    calculating eigenvectors, weights, and coefficients related to cryptocurrency port
    """

    def __init__(self):
        """
        Initializes the CryptoDataProcessor class.
        """
        pass

    def read_and_process_data(self, price_file, ticker_file):
        """
        Reads cryptocurrency price and ticker data from CSV files, processes them by
        converting the 'startTime' to datetime format and ensuring inclusion of 'ETH'
        Merges the price and ticker dataframes on 'startTime'.

        :param price_file: Filepath for the CSV file containing price data.
        :param ticker_file: Filepath for the CSV file containing ticker data.
        :return: Merged DataFrame of price and ticker data.
        """
        df_price = pd.read_csv(price_file)
        df_price['startTime'] = pd.to_datetime(df_price['startTime'])

        df_ticker = pd.read_csv(ticker_file)
        df_ticker['startTime'] = pd.to_datetime(df_ticker['startTime'])

        # Convert ticker columns to a list and ensure BTC and ETH are included
        ticker_cols = [str(i) for i in range(40)]
        df_ticker['ticker_list'] = df_ticker[ticker_cols].values.tolist()
        df_ticker['ticker_list'] = df_ticker.apply(lambda x: set(x['ticker_list'] + ['

        df_merge = pd.merge(df_price, df_ticker, on='startTime', how='inner')
        return df_merge

    def eigen(self, data):
        """
        Computes the eigenvectors and weights from the given data.

        :param data: DataFrame containing the cryptocurrency data.
        :return: DataFrame of eigenvectors and weights, along with Factor1 and Factor2
        """

        std = data.std(skipna=True)
        evecs = Functions.pca(data)  # Placeholder for actual PCA computation
        evecs.index = std.index
        std = pd.DataFrame(std)
        evecs = evecs.reset_index()
        std = std.reset_index()

        evecs1 = evecs.merge(std, on='index')
        evecs1 = evecs1.set_axis(['Ticker', 'EigVec1', 'EigVec2', 'Std'], axis=1)
        evecs1['w_eig'] = evecs1['EigVec1'] / evecs1['Std']
```

```
        evecs1['w_eig2'] = evecs1['EigVec2'] / evecs1['Std']
        evecs1.replace([np.nan, np.inf, -np.inf], 0, inplace=True)

        Factor1 = data.dot(evecs1['w_eig'].to_numpy())
        Factor2 = data.dot(evecs1['w_eig2'].to_numpy())

        return evecs1, Factor1, Factor2

    def coef(self, df, ticker_list):
        """
        Calculates coefficients for each ticker in the ticker list.

        :param df: DataFrame containing the cryptocurrency data.
        :param ticker_list: List of tickers to calculate coefficients for.
        :return: DataFrame with calculated coefficients for each ticker.
        """
        coef_ab = pd.DataFrame()
        dict_residual_sum = {}
        for i in ticker_list:
            if str(i) == 'nan':
                continue
            residual_error = pd.DataFrame()
            residual_error[i] = Functions.regress_part1(df, i)
            residual_error['cum_sum'] = residual_error[i].cumsum(skipna=True)
            residual_error['cum_sum_shift'] = residual_error['cum_sum'].shift(-1)
            residual_error.replace([np.nan, np.inf, -np.inf], 0, inplace=True)
            dict_residual_sum[i] = residual_error[i].sum(skipna=True)
            coef_ab[i] = Functions.regress_part2(residual_error)

        coef_ab = coef_ab.T
        coef_ab['k'] = coef_ab.apply(lambda x: -1 * np.log(x['cum_sum']) * 8760, axis=
        coef_ab['m'] = coef_ab.apply(lambda x: x['Intercept'] / (1 - x['cum_sum']), ax
        coef_ab['sigma'] = coef_ab.apply(lambda x: np.sqrt((x['residual_var'] * 2 * x[
        coef_ab['sigma_eq'] = coef_ab.apply(lambda x: np.sqrt((x['residual_var']) / (1
        coef_ab.replace([np.nan, np.inf, -np.inf], 0, inplace=True)
        dict_residual_sum = pd.DataFrame(dict_residual_sum.items(), columns=['ticker',
        coef_ab = coef_ab.reset_index()
        coef_ab.rename(columns={'index': "ticker"}, inplace=True)
        coef_ab = coef_ab.merge(dict_residual_sum, on='ticker')
        coef_ab['S'] = coef_ab.apply(lambda x: (x['X'] - x['m']) / x['sigma_eq'] if x[

        return coef_ab
```

In [10]:

```
class MeanReversionStrategy:
    """
    A class representing a mean reversion trading strategy.
    It generates trading signals based on specified threshold levels.
    """

    def __init__(self):
        """
        Initializes the MeanReversionStrategy class.
        """
        # Threshold levels for generating signals
        self.sell_open_threshold = 1.25  # Threshold for opening sell positions
        self.buy_open_threshold = -1.25  # Threshold for opening buy positions
        self.sell_close_threshold = 0.75  # Threshold for closing sell positions
        self.buy_close_threshold = -0.5  # Threshold for closing buy positions
```

```
    def generate_signals(self, df_coeff, signal_dict):
        """
        Generates trading signals based on the mean reversion strategy.

        :param df_coeff: DataFrame containing coefficients for each ticker.
        :param signal_dict: Dictionary holding the current signal state for each ticke
        :return: Updated DataFrame with generated signals.
        """
        for index, row in df_coeff.iterrows():
            tick = row['ticker']
            current_signal = signal_dict.get(tick, 0)

            # Generate signals based on the strategy's thresholds
            if current_signal == 0:
                if row["S"] < self.buy_open_threshold:
                    df_coeff.at[index, 'signal'] = 1
                    signal_dict[tick] = 1
                elif row["S"] > self.sell_open_threshold:
                    df_coeff.at[index, 'signal'] = -1
                    signal_dict[tick] = -1
                else:
                    df_coeff.at[index, 'signal'] = current_signal
            elif current_signal == 1:
                if row["S"] > self.buy_close_threshold:
                    df_coeff.at[index, 'signal'] = 0
                    signal_dict[tick] = 0
                else:
                    df_coeff.at[index, 'signal'] = current_signal
            elif current_signal == -1:
                if row["S"] < self.sell_close_threshold:
                    df_coeff.at[index, 'signal'] = 0
                    signal_dict[tick] = 0
                else:
                    df_coeff.at[index, 'signal'] = current_signal

        return df_coeff
```

In [11]:
```
processor = CryptoDataProcessor()

price_data = pd.read_csv('coin_all_prices_full.csv')
price_data['startTime'] = pd.to_datetime(price_data['startTime'])
df_merge = processor.read_and_process_data('coin_all_prices_full.csv', 'coin_universe_
```

In [12]:
```
coefficients = Functions()

dict1, dict2, dict3 = {}, {}, {}
for index, row in tqdm(df_merge.iterrows()):
    # Extract relevant tickers based on ticker_list for the current row
    relevant_tickers = list(set(df_merge.columns).intersection(df_merge['ticker_list']
    current_data = df_merge[relevant_tickers].copy()

    # Add and calculate additional columns
    current_data['startTime'] = df_merge['startTime']
    current_data['trade_start'] = row['startTime']
    current_data['Diff'] = current_data['trade_start'] - current_data['startTime']

    # Filter data based on time difference
    current_data = current_data[(current_data['Diff'] > datetime.timedelta(hours=0)) &
                                (current_data['Diff'] <= datetime.timedelta(hours=241)
```

```
    # Calculate percent change and filter out first row
    current_data.iloc[:, :-3] = current_data.iloc[:, :-3].pct_change()
    current_data = current_data.iloc[1:, :]
    current_data.replace([np.nan, np.inf, -np.inf], 0, inplace=True)

    # Ensure we have enough data points
    if current_data.shape[0] < 240:
        continue

    # Calculate eigenvalues and factors
    factors = CryptoDataProcessor()
    evecs1, Factor1, Factor2 = factors.eigen(current_data.iloc[:, :-3])

    # Add Factor1 and Factor2 to the current data
    current_data['Factor1'] = pd.Series(Factor1)
    current_data['Factor2'] = pd.Series(Factor2)
    current_data.replace([np.nan, np.inf, -np.inf], 0, inplace=True)

    # Transpose and adjust evecs1 DataFrame
    evecs1 = evecs1.T
    evecs1.columns = evecs1.iloc[0]
    evecs1['startTime'] = row['startTime']
    evecs1 = evecs1.iloc[1:, :]

    # Store results in dictionaries
    dict1[row['startTime']] = current_data
    dict2["evecs" + str(row['startTime'])] = evecs1
    dict3["coef" + str(row['startTime'])] = factors.coef(current_data, df_merge['ticke
    dict3["coef" + str(row['startTime'])]['startTime'] = row['startTime']
```

```
14015it [1:12:52,  3.21it/s]
```

In [13]:

```
# Instead of appending DataFrames in a loop, use pd.concat for efficiency
df_eigen = pd.concat(tqdm(dict2.values())).reset_index()

# Reorder columns to bring 'startTime' to the front
cols = ['startTime'] + [col for col in df_eigen if col != 'startTime']
df_eigen = df_eigen[cols]

# Function to process weights into the desired long format
def process_weights(df, weight_indicator, value_name):
    weight_df = df[df['index'] == weight_indicator].copy()
    weight_df.drop(columns='index', inplace=True)
    return weight_df.melt(id_vars=['startTime'], var_name='ticker', value_name=value_r

# Process eigen weight 1 and eigen weight 2
df_eigen_w1 = process_weights(df_eigen, 'w_eig', 'Weight1')
df_eigen_w2 = process_weights(df_eigen, 'w_eig2', 'Weight2')
```

```
100%|██████████| 13774/13774 [00:00<00:00, 1743281.33it/s]
```

In [14]:

```
tradingsignal = MeanReversionStrategy()


# Combine all DataFrames from dict3 into a single DataFrame
df_coeff = pd.concat([x for x in tqdm(dict3.values())], ignore_index=True)

# Initialize signal_dict with 0 for each unique ticker
signal_dict = {ticker: 0 for ticker in df_coeff['ticker'].drop_duplicates()}
```

```
# Add a 'signal' column with NaN values
df_coeff["signal"] = np.nan

# Example: tradingsignal = MeanReversionStrategy() or import tradingsignal
df_coeff = tradingsignal.generate_signals(df_coeff, signal_dict)

# Pivot the DataFrame for startTime and ticker, with signal values
df_coeff_signal = df_coeff.pivot(index='startTime', columns='ticker', values='signal')

# Exporting Trading signal file if necessary
#df_coeff_signal.to_csv('trading_signals.csv')
```

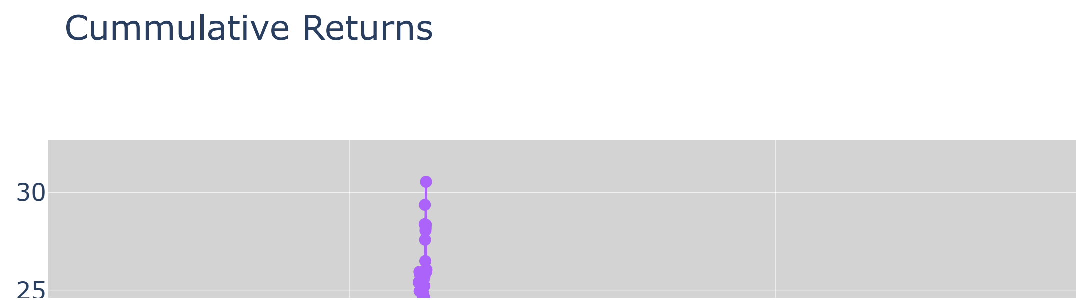
```
100%|██████████| 13774/13774 [00:00<00:00, 3058194.02it/s]
```

In [15]:

```
df_price_transformed = price_data.melt(id_vars=["startTime", "time"], var_name="ticker
df_price_transformed.sort_values(by=['ticker', 'startTime'], inplace=True)
df_price_transformed['ret'] = df_price_transformed.groupby('ticker')['price'].pct_chan
df_price_transformed['ret'] = df_price_transformed.groupby('ticker')['ret'].shift(-1)

# Merge the transformed price data with df_coeff
df_coeff_final = pd.merge(df_coeff, df_price_transformed, on=['startTime', 'ticker'])

# Replace NaN, inf, and -inf in 'ret' with 0
df_coeff_final['ret'].replace([np.nan, np.inf, -np.inf], 0, inplace=True)

# Calculate traded returns based on the signal
df_coeff_final['traded_ret'] = df_coeff_final['ret'] * df_coeff_final['signal']
```

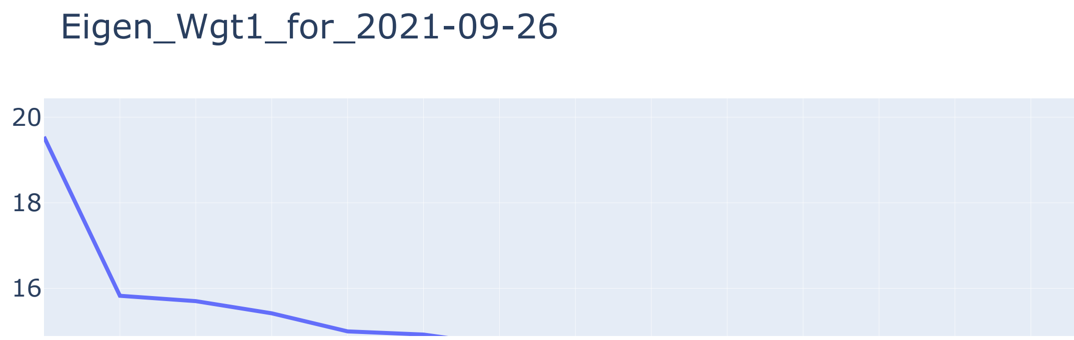
In [16]:

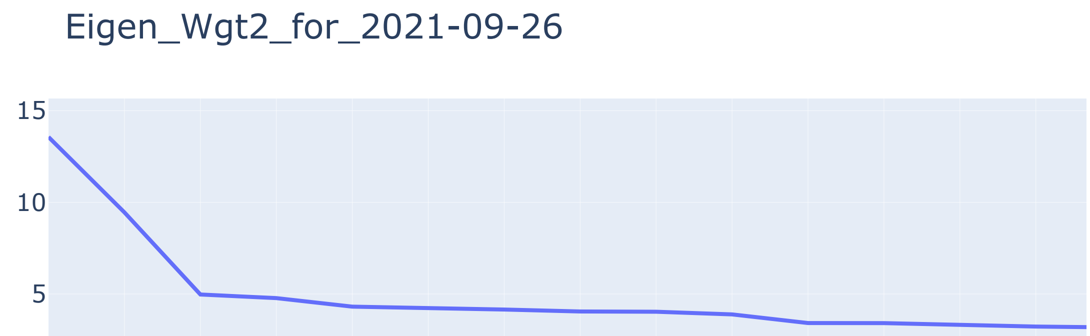
```
# Merging dataframes
data_frames = [df_coeff_final, df_eigen_w1, df_eigen_w2]
merged_df = data_frames[0]
for df in data_frames[1:]:
    merged_df = merged_df.merge(df, on=['startTime', 'ticker'], how='inner')

# Scaling weights
merged_df = Functions.scale_weights(merged_df, 'Weight1', 'Scaled_Weight1')
merged_df = Functions.scale_weights(merged_df, 'Weight2', 'Scaled_Weight2')

# Calculating Eigen portfolios
merged_df = Functions.calculate_eigen_portfolio(merged_df, 'ret', 'Scaled_Weight1', 'E
merged_df = Functions.calculate_eigen_portfolio(merged_df, 'ret', 'Scaled_Weight2', 'E

# Group by 'startTime' and calculate cumulative returns
grouped_df = merged_df.groupby('startTime')[['Eigen_pf1', 'Eigen_pf2']].sum()

grouped_df = Functions.cumulative_returns(grouped_df, ['Eigen_pf1', 'Eigen_pf2'])

# Processing df_price
df_price_secs = price_data[['startTime', 'ETH', 'BTC']].copy()
df_price_secs['ETH'] = df_price_secs['ETH'].bfill(axis='rows')
df_price_secs.iloc[:, 1:] = df_price_secs.iloc[:, 1:].pct_change()
df_price_secs = df_price_secs.iloc[1:, :]
df_price_secs.iloc[:, 1:] = (1 + df_price_secs.iloc[:, 1:]).cumprod()

# Merging and plotting cumulative returns
cumulative_returns_df = df_price_secs.merge(grouped_df[['cum_Eigen_pf1', 'cum_Eigen_pf
Functions.time_plot(cumulative_returns_df, 'Cummulative Returns')
```

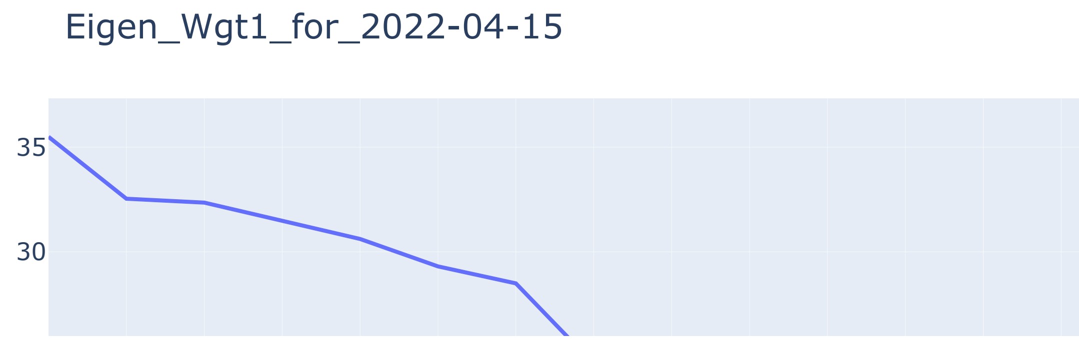


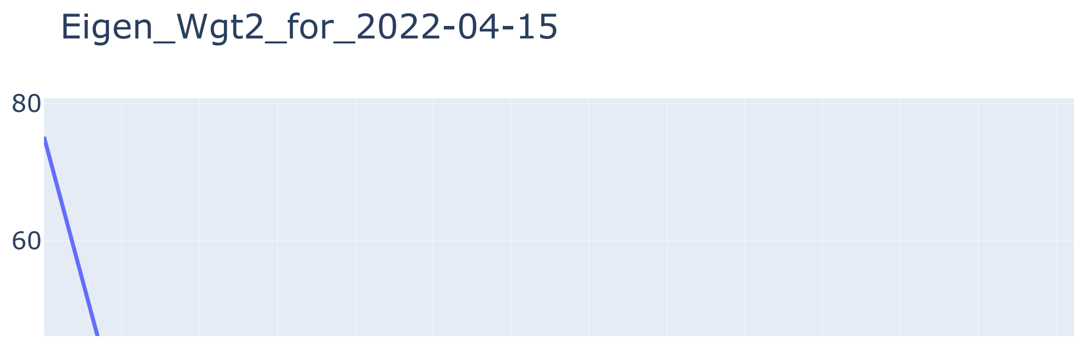
In [17]:
```
df_eigenvec1 = df_eigen[df_eigen['index'] == 'EigVec1']
df_eigenvec2 = df_eigen[df_eigen['index'] == 'EigVec2']
del df_eigenvec1['index'], df_eigenvec2['index']


#export as required
df_eigenvec1.to_csv('task1a_1.csv')
df_eigenvec2.to_csv('task1a_2.csv')
```

In [18]:
```
Functions.eigenweights(df_eigen,'w_eig','2021-09-26 12:00:00+00:00','Eigen_Wgt1_for_20
Functions.eigenweights(df_eigen,'w_eig2','2021-09-26 12:00:00+00:00','Eigen_Wgt2_for_2
Functions.eigenweights(df_eigen,'w_eig','2022-04-15 20:00:00+00:00','Eigen_Wgt1_for_20
Functions.eigenweights(df_eigen,'w_eig2','2022-04-15 20:00:00+00:00','Eigen_Wgt2_for_2

BTC_df = df_coeff[df_coeff['ticker'] == 'BTC']
BTC_df = BTC_df[(BTC_df['startTime'] >= '2021-09-26 06:00:00+00:00') & (BTC_df['startT
ETH_df = df_coeff[df_coeff['ticker'] == 'ETH']
ETH_df = ETH_df[(ETH_df['startTime'] >= '2021-09-26 06:00:00+00:00') & (ETH_df['startT
Functions.time_plot(BTC_df[['startTime','S']],'BTC Signal score trend')
Functions.time_plot(ETH_df[['startTime','S']],'ETH Signal score trend')
```

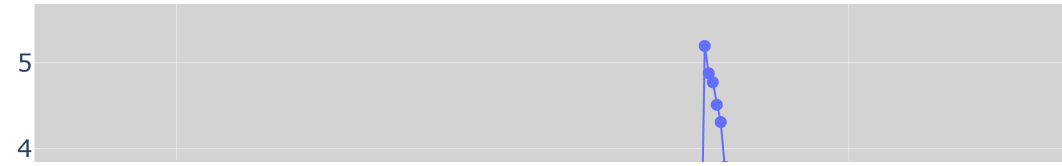

Eigen_Wgt1_for_2021-09-26
20
18
16

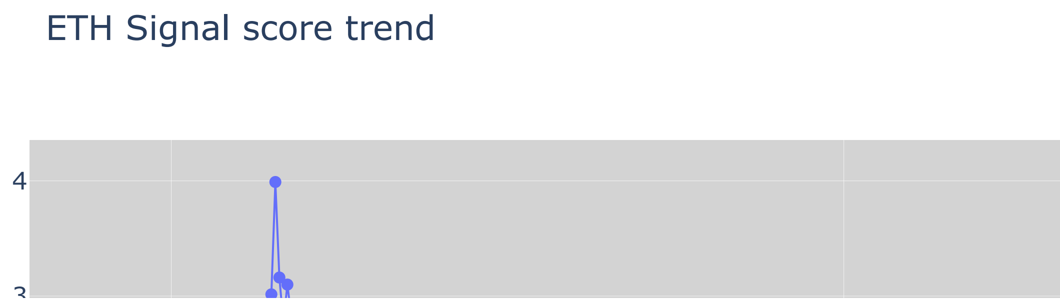

Eigen_Wgt2_for_2021-09-26
15
10
5


Eigen_Wgt1_for_2022-04-15
35
30


Eigen_Wgt2_for_2022-04-15
80
60

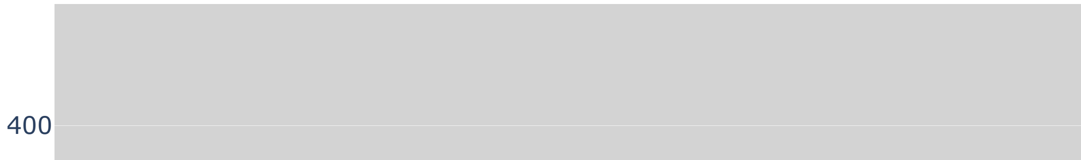
BTC Signal score trend

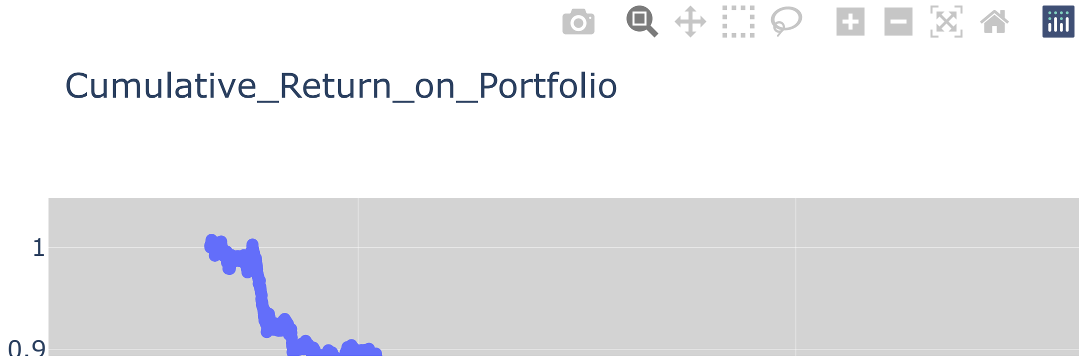


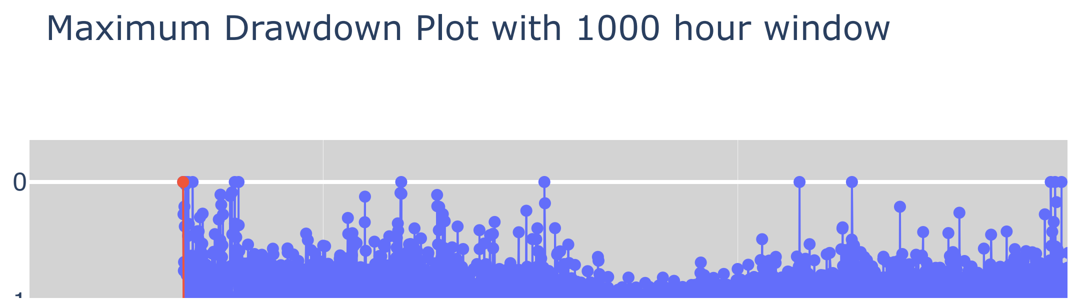


In [19]:
```
df_final = df_coeff_final.groupby('startTime')['traded_ret'].mean()
df_final = pd.DataFrame(df_final)
df_final['ret'] = 1 + df_final['traded_ret']
df_final['cum_ret'] = df_final['ret'].cumprod()
df_final = df_final.reset_index()


histogram = Functions.plot_hist(df_final['traded_ret'],'Traded Returns Histogram', ret
cumulativereturn = Functions.time_plot(df_final[['startTime','cum_ret']], 'Cumulative_


sharpe_ratio = df_final['traded_ret'].mean() / df_final['traded_ret'].std()


window = 1000
df_final['Roll_Max'] = df_final['traded_ret'].rolling(window, min_periods=1).max()
df_final['Daily_Drawdown'] = df_final['traded_ret']/df_final['Roll_Max'] - 1.0
df_final['Max_Daily_Drawdown'] = df_final['Daily_Drawdown'].rolling(window, min_perioc

Functions.time_plot(df_final[['startTime','Daily_Drawdown','Max_Daily_Drawdown']],'Max
```

Traded Returns Histogram




Cumulative_Return_on_Portfolio
1
0.9



In [20]:
```
import plotly.io as pio

# Calculate daily drawdown
daily_drawdown = (df_final['traded_ret'] - df_final['traded_ret'].cummax()) / df_final

# Calculate maximum daily drawdown
max_daily_drawdown = daily_drawdown.min()  # This will be negative

# Print maximum daily drawdown in green
print("\033[32mMaximum Daily Drawdown: {:.2f}%\033[0m".format(max_daily_drawdown * 100

# Print Sharpe Ratio
print("\033[32mSharpe_Ratio is {:.5f}\033[0m".format(round(sharpe_ratio, 5)))
print()

# Write images to files
pio.write_image(cumulativereturn, 'cumulative_return.jpeg', format='jpeg')
pio.write_image(histogram, 'hist_return.jpeg', format='jpeg')

print("\033[32mImages saved to folder successfully.\033[0m")
```

```
Maximum Daily Drawdown: -411.77%
Sharpe_Ratio is -0.03143

Images saved to folder successfully.
```

In [ ]:

```

```