In [5]:
```
import ccxt
import pandas as pd
import numpy as np
import talib as ta
import matplotlib.pyplot as plt
from datetime import datetime
from sqlalchemy import create_engine
import urllib
import os
```

In [6]:
```
# Database connection configuration
DATABASE_TYPE = 'mssql'
DBAPI = 'pyodbc'
SERVER = 'MARTIN'
DATABASE = 'crypto_data'
DRIVER = 'ODBC Driver 17 for SQL Server'

# Create a connection URI for SQLAlchemy
params = urllib.parse.quote_plus(f"DRIVER={DRIVER};SERVER={SERVER};DATABASE={DATABA
DATABASE_URI = f"{DATABASE_TYPE}+{DBAPI}:///?odbc_connect={params}"

# Create SQLAlchemy engine
engine = create_engine(DATABASE_URI, echo=False)
```

In [7]:
```
# Initialize Kraken exchange via ccxt
kraken = ccxt.kraken()
```

In [8]:
```
# Download historical data from Kraken
def get_crypto_data(symbol, timeframe='1m', since=None):
    ohlcv = kraken.fetch_ohlcv(symbol, timeframe=timeframe, since=since)
    df = pd.DataFrame(ohlcv, columns=['timestamp', 'Price', 'Open', 'High', 'Low',
    df['timestamp'] = pd.to_datetime(df['timestamp'], unit='ms')
    df.set_index('timestamp', inplace=True)
    return df
```

In [9]:
```
# Create Calculations of Volatility
def calculate_volatility(df, window):
    df['returns'] = df['Close'].pct_change()
    df['volatility'] = df['returns'].rolling(window=window).std() * np.sqrt(window)
    return df
```

In [10]:
```
# Find the support and Resistancce High and Low Factors
def find_support_resistance(df):
    df['support'] = df['Low'].rolling(window=60).min()
    df['resistance'] = df['High'].rolling(window=60).max()
    return df
```

In [11]:
```
# Calculate the moving averages
def calculate_moving_averages(df, short_window=14, long_window=50):
    df['SMA_14'] = ta.SMA(df['Close'], timeperiod=short_window)
    df['EMA_50'] = ta.EMA(df['Close'], timeperiod=long_window)
    return df
```

In [12]:
```python
# Calcualte the bollinger bands
def calculate_bollinger_bands(df, window=20, num_std=2):
    df['BB_upper'], df['BB_middle'], df['BB_lower'] = ta.BBANDS(df['Close'], timepe
    return df
```

In [13]:
```python
# Calculate the RSI
def calculate_rsi(df, period=14):
    df['RSI'] = ta.RSI(df['Close'], timeperiod=period)
    return df
```

In [14]:
```python
# Calculate the VWAP
def calculate_vwap(df):
    df['vwap'] = (df['Volume'] * (df['High'] + df['Low'] + df['Close']) / 3).cumsum
    return df
```

In [15]:
```python
# Calcualte the Fibonacci Levels
def calculate_fibonacci_levels(df):
    max_price = df['Close'].max()
    min_price = df['Close'].min()
    diff = max_price - min_price
    df['fib_0.236'] = max_price - 0.236 * diff
    df['fib_0.382'] = max_price - 0.382 * diff
    df['fib_0.5'] = max_price - 0.5 * diff
    df['fib_0.618'] = max_price - 0.618 * diff
    df['fib_1'] = min_price
    return df
```

In [16]:
```python
# Calculate the MACD
def calculate_macd(df):
    df['macd'], df['macdsignal'], df['macdhist'] = ta.MACD(df['Close'], fastperiod=
    return df
```

In [17]:
```python
# Calcualte the ATR
def calculate_atr(df, window=14):
    df['ATR'] = ta.ATR(df['High'], df['Low'], df['Close'], timeperiod=window)
    return df
```

In [18]:
```python
# Stochastic Oscillator
def calculate_stochastic(df, k_window=14, d_window=3):
    df['slowk'], df['slowd'] = ta.STOCH(df['High'], df['Low'], df['Close'], fastk_p
    return df
```

In [19]:
```python
# Ichimoku Cloud
def calculate_ichimoku(df):
    df['ichimoku_a'], df['ichimoku_b'], df['ichimoku_c'], df['ichimoku_d'], df['ich
    return df
```

In [20]:
```python
# Parabolic SAR (Stop and Reverse)
def calculate_parabolic_sar(df):
    df['SAR'] = ta.SAR(df['High'], df['Low'], acceleration=0.02, maximum=0.2)
    return df
```

In [21]:
```python
# ADX (Average Directional Index)
def calculate_adx(df, period=14):
    df['ADX'] = ta.ADX(df['High'], df['Low'], df['Close'], timeperiod=period)
    return df
```

In [22]:
```python
# Chaikin Money Flow (CMF)
def calculate_cmf(df, window=20):
    df['CMF'] = ta.ADOSC(df['High'], df['Low'], df['Close'], df['Volume'],  df['4H
    return df
```

In [23]:
```python
# On-Balance Volume (OBV)
def calculate_obv(df):
    df['OBV'] = ta.OBV(df['Close'], df['Volume'])
    return df
```

In [24]:
```python
# Sweep and Clean the data
def clean_data(df):
    df.dropna(how='all', inplace=True)
    df.ffill(inplace=True)  # Forward fill missing data
    df.bfill(inplace=True)  # Backward fill missing data
    df.replace([np.inf, -np.inf], np.nan, inplace=True)
    df.dropna(inplace=True)
    return df
```

In [25]:
```python
# Create the table from the return data in SQL and save
def save_to_sql(df, table_name):
    try:
        if df.empty:
            print("Data is empty after cleaning. Nothing to save.")
            return
        df.to_sql(table_name, con=engine, if_exists='replace', index_label='timesta
        print(f"Data successfully saved to {table_name} in SQL Server.")
    except Exception as e:
        print(f"Error saving to SQL Server: {e}")
    finally:
        engine.dispose()
        print("SQL connection closed.")
```

In [26]:
```python
# Save data to CSV
def save_to_csv(df, file_name):
    try:
        if df.empty:
            print("Data is empty after cleaning. Nothing to save.")
            return  # Ensure proper indentation
        df.to_csv(file_name, index=False)  # Specify whether to include the index
        print(f"Data successfully saved to {file_name}.")
    except Exception as e:
        print(f"Error saving to CSV: {e}")
```

In [27]:
```python
# Calculate buy/sell signal based on percentage change
def calculate_buy_sell_signal(df, threshold=0.15):
    # Calculate the percentage change from the previous close
    df['percent_change'] = df['Close'].pct_change() * 100
```

```
    # Generate "BUY" or "SELL" based on the threshold
    df['Signal'] = df['percent_change'].apply(lambda x: "SELL" if abs(x) >= thresho
    return df
```

In [28]:
```
# Plot various data points
def plot_data(df, symbol):
    plt.figure(figsize=(14, 8))

    # Plot Close Price, Moving Averages, and Bollinger Bands
    plt.subplot(2, 1, 1)
    plt.plot(df['Close'], label='Close Price')
    plt.plot(df['SMA_14'], label='SMA 14', linestyle='--')
    plt.plot(df['EMA_50'], label='EMA 50', linestyle='--')
    plt.plot(df['BB_upper'], label='Upper BB', linestyle='--')
    plt.plot(df['BB_lower'], label='Lower BB', linestyle='--')
    plt.title(f'{symbol} Close Price with Moving Averages and Bollinger Bands')
    plt.legend()

    # Plot RSI
    plt.subplot(2, 1, 2)
    plt.plot(df['RSI'], label='RSI', color='green')
    plt.axhline(70, color='red', linestyle='--', label='Overbought (70)')
    plt.axhline(30, color='blue', linestyle='--', label='Oversold (30)')
    plt.title(f'{symbol} RSI')
    plt.legend()

    plt.tight_layout()
    plt.show()

    # Plot Returns and Volatility
    plt.figure(figsize=(14, 8))
    plt.subplot(2, 1, 1)
    plt.plot(df.index, df['returns'], label='Returns')
    plt.title(f'{symbol} Returns')
    plt.legend()

    plt.subplot(2, 1, 2)
    plt.plot(df.index, df['volatility'], label='Volatility', color='orange')
    plt.title(f'{symbol} Volatility')
    plt.legend()
    plt.tight_layout()
    plt.show()
```

In [29]:
```
def plot_data(df, symbol):
    """Plot the closing price and indicators for a given symbol."""
    plt.figure(figsize=(14, 10))
    plt.subplot(2, 1, 1)
    plt.plot(df['Close'], label='Close Price')

    # Plot SMA_14 if available
    if 'SMA_14' in df.columns:
        plt.plot(df['SMA_14'], label='SMA 14', linestyle='--')
    else:
        print(f"'SMA_14' not found for {symbol}. Skipping SMA plot.")
```

```
    # Plot EMA_50 if available
    if 'EMA_50' in df.columns:
        plt.plot(df['EMA_50'], label='EMA 50', linestyle='--')
    else:
        print(f"'EMA_50' not found for {symbol}. Skipping EMA plot.")

    # Plot Bollinger Bands if available
    if 'BB_upper' in df.columns and 'BB_lower' in df.columns:
        plt.plot(df['BB_upper'], label='Upper BB', linestyle='--')
        plt.plot(df['BB_lower'], label='Lower BB', linestyle='--')
    else:
        print(f"Bollinger Bands not found for {symbol}. Skipping BB plot.")

    plt.title(f"{symbol} Price with Indicators")
    plt.legend()
    plt.show()
```

In [30]:
```
def process_symbol_data(symbol, timeframe, since):
    """Fetch and process data for a specific symbol."""
    print(f"\nFetching data for {symbol}...")
    df = get_crypto_data(symbol, timeframe, since)
    if df is None or df.empty:
        print(f"No data returned for {symbol}. Skipping...")
        return None  # Skip processing if no data is available

    # Sequential data processing with error handling
    try:
        df = clean_data(df)
        df = calculate_moving_averages(df)
        df = calculate_bollinger_bands(df)
        df = calculate_rsi(df)
        df = calculate_volatility(df, window=14)
        df = find_support_resistance(df)
        df = calculate_vwap(df)
        df = calculate_fibonacci_levels(df)
        df = calculate_macd(df)
        df = calculate_atr(df, window=14)
        df = calculate_buy_sell_signal(df)
    except Exception as e:
        print(f"Error processing data for {symbol}: {e}")
        return None

    return df
```

In [31]:
```
def process_symbol_data(symbol, timeframe, since):
    try:
        df = get_crypto_data(symbol, timeframe, since)
        print(f"Data for {symbol}:\n{df.head()}")  # Print the DataFrame for inspec
        # Ensure 'Close' column exists
        if 'close' not in df.columns:
            print(f"Warning: 'close' column missing for {symbol}")
            return None  # Return None if the 'close' column is missing
        df.rename(columns={'close': 'Close'}, inplace=True)  # Rename to 'Close' if
        return df
    except Exception as e:
```

```
        print(f"Error processing data for {symbol}: {e}")
        return None
```

In [32]:

```
import pandas as pd
import sqlite3
import os

# Define the function to save data to SQL
def save_to_sql(df, table_name, db_name='crypto_data.db'):
    with sqlite3.connect(db_name) as conn:
        df.to_sql(table_name, conn, if_exists='replace', index=True)
    print(f"Data saved to SQL table: {table_name}")

# Define the function to save data to CSV
def save_to_csv(df, file_name):
    file_path = os.path.join(os.getcwd(), file_name)
    df.to_csv(file_path, index=True)
    print(f"Data saved to CSV file: {file_name}")

# Define the function to fetch and process data for a specific symbol
def get_crypto_data(symbol, timeframe='1m', since=None):
    ohlcv = kraken.fetch_ohlcv(symbol, timeframe=timeframe, since=since)
    df = pd.DataFrame(ohlcv, columns=['timestamp', 'open', 'high', 'low', 'close',
    df['timestamp'] = pd.to_datetime(df['timestamp'], unit='ms')
    df.set_index('timestamp', inplace=True)
    return df

# Define the function to process symbol data
def process_symbol_data(symbol, timeframe, since):
    try:
        df = get_crypto_data(symbol, timeframe, since)

        if 'close' not in df.columns:
            print(f"Warning: 'close' column missing for {symbol}")
            return None

        # Rename to 'Close' for consistency
        df.rename(columns={'close': 'Close'}, inplace=True)

        # Initialize 'Signal' column using `iloc` for proper positional indexing
        df['Signal'] = None
        df.loc[df.index[1:], 'Signal'] = [
            'BUY' if df['Close'].iloc[i] > df['Close'].iloc[i-1] else 'SELL'
            for i in range(1, len(df))
        ]

        # Calculate 'Volatility'
        df['Volatility'] = df['high'] - df['low']

        return df

    except Exception as e:
        print(f"Error processing data for {symbol}: {e}")
        return None

# Define the main function with the Crypto Symbols
```

```python
def main():
    symbols = list(set([
        'ADA/USD', 'APE/USD', 'AUCTION/USD', 'BODEN/USD', 'BTC/USD', 'CPOOL/USD', '
        'EUL/USD', 'GMT/USD', 'LINK/USD', 'USDT/USD', 'MEME/USD', 'MNT/USD', 'MOG/U
        'NTRN/USD', 'PYTH/USD', 'RENDER/USD', 'SAFE/USD', 'SUPER/USD', 'TNSR/USD',
        'XMR/USD', 'ZRX/USD', 'LTC/USD', 'DOGE/USD'
    ]))

    timeframe = '1m'
    since = kraken.parse8601('2024-01-01T00:00:00Z')
    risk_threshold = 0.05
    all_crypto_data = {}
    sell_counts = {}
    buy_counts = {}
    risk_labels = {}

    for symbol in symbols:
        df = process_symbol_data(symbol, timeframe, since)
        if df is None or 'Close' not in df.columns or 'Signal' not in df.columns:
            print(f"Skipping {symbol}: Data not available or required columns missi
            continue

        # Process sell and buy counts
        sell_counts[symbol] = df['Signal'].value_counts().get('SELL', 0)
        buy_counts[symbol] = df['Signal'].value_counts().get('BUY', 0)

        avg_volatility = df['Volatility'].mean() if 'Volatility' in df.columns else
        risk_label = "High Risk" if avg_volatility and avg_volatility > risk_thresh
        risk_labels[symbol] = risk_label
        df['Risk'] = risk_label
        all_crypto_data[symbol] = df[['Close', 'Signal', 'Risk']]

    # Display summary results
    top_sell_symbols = sorted(sell_counts.items(), key=lambda x: x[1], reverse=True
    print("\nTop 5 Symbols with Most 'SELL' Signals:")
    for symbol, count in top_sell_symbols:
        print(f"{symbol}: {count} 'SELL' signals")

    top_buy_symbols = sorted(buy_counts.items(), key=lambda x: x[1], reverse=True)[
    print("\nTop 5 Symbols with Most 'BUY' Signals:")
    for symbol, count in top_buy_symbols:
        print(f"{symbol}: {count} 'BUY' signals")

    print("\nRisk Classification:")
    for symbol, risk in risk_labels.items():
        print(f"{symbol}: {risk}")
    # Create the table from the return data in SQL and save
#def save_to_sql(df, table_name):
#    try:
#        if df.empty:
#            print("Data is empty after cleaning. Nothing to save.")
#            return
#        df.to_sql(table_name, con=engine, if_exists='replace', index_label='timest
#        print(f"Data successfully saved to {table_name} in SQL Server.")
#    except Exception as e:
#        print(f"Error saving to SQL Server: {e}")
```

```
#    finally:
#        engine.dispose()
#        print("SQL connection closed.")

    # Save data to SQL and CSV
#    for symbol, df in all_crypto_data.items():
#        table_name = symbol.replace('/', '_').lower()
#        save_to_sql(df, table_name)
#        csv_file_name = f"{symbol.replace('/', '_').lower()}.csv"
#        save_to_csv(df, csv_file_name)

    return all_crypto_data

if __name__ == "__main__":
    all_crypto_data = main()
```

```
Top 5 Symbols with Most 'SELL' Signals:
AUCTION/USD: 717 'SELL' signals
MNT/USD: 713 'SELL' signals
GMT/USD: 712 'SELL' signals
NTRN/USD: 711 'SELL' signals
EUL/USD: 700 'SELL' signals

Top 5 Symbols with Most 'BUY' Signals:
DOGE/USD: 372 'BUY' signals
ETH/USD: 348 'BUY' signals
BTC/USD: 338 'BUY' signals
LTC/USD: 241 'BUY' signals
USDT/USD: 239 'BUY' signals

Risk Classification:
BODEN/USD: Low Risk
TNSR/USD: Low Risk
SAFE/USD: Low Risk
APE/USD: Low Risk
BTC/USD: High Risk
DOGE/USD: Low Risk
ADA/USD: Low Risk
PYTH/USD: Low Risk
LTC/USD: Low Risk
USDT/USD: Low Risk
MNT/USD: Unknown Risk
XMR/USD: Low Risk
LINK/USD: Low Risk
AUCTION/USD: Unknown Risk
NOS/USD: Low Risk
ZRX/USD: Low Risk
CPOOL/USD: Low Risk
EUL/USD: Low Risk
GMT/USD: Unknown Risk
SUPER/USD: Low Risk
NTRN/USD: Low Risk
MEME/USD: Low Risk
RENDER/USD: Low Risk
ETH/USD: High Risk
MOG/USD: Low Risk
TREMP/USD: Low Risk
```

In [33]:

```python
import dash
from dash import dcc, html
import plotly.express as px
import plotly.graph_objects as go
import pandas as pd

# Load the crypto data and handle the possibility of None
all_crypto_data = main()  # Load data from the main script
if not isinstance(all_crypto_data, dict):
    all_crypto_data = {}  # Ensure all_crypto_data is a dictionary if main() fails

# function to calculate volatility (standard deviation of returns)
def calculate_volatility(df):
    df['Returns'] = df['Close'].pct_change()
```

```
    return df['Returns'].std()

# Preprocess data to add volatility, if data is available
for crypto_name, df in all_crypto_data.items():
    if 'Close' in df.columns:  # Ensure 'Close' column exists before calculation
        df['Volatility'] = calculate_volatility(df)

app = dash.Dash(__name__)

# Check if all_crypto_data is empty and set default value
crypto_names = list(all_crypto_data.keys())
default_crypto = crypto_names[0] if crypto_names else None  # Fallback if no data

# Function to create line chart for selected cryptocurrency's price data
def create_crypto_line_chart(df, crypto_name):
    if df.empty:
        return go.Figure()  # Return an empty figure if there's no data
    return px.line(df, x=df.index, y='Close', title=f'{crypto_name.capitalize()} Pr

# Function to create bar chart for buy/sell counts
def create_buy_sell_chart(df, crypto_name):
    if df.empty:
        return go.Figure()  # Return an empty figure if there's no data
    buy_count = (df['Signal'] == 'BUY').sum()
    sell_count = (df['Signal'] == 'SELL').sum()
    fig = go.Figure(data=[
        go.Bar(name='BUY', x=[crypto_name], y=[buy_count], marker_color='green'),
        go.Bar(name='SELL', x=[crypto_name], y=[sell_count], marker_color='red')
    ])
    fig.update_layout(barmode='group', title=f'{crypto_name.capitalize()} Buy/Sell
    return fig

# Dashboard layout
app.layout = html.Div([
    html.H1("Cryptocurrency Volatility Dashboard"),

    # Dropdown for selecting cryptocurrency
    dcc.Dropdown(id='crypto-selector',
                 options=[{'label': name, 'value': name} for name in crypto_names],
                 value=default_crypto,  # Set default to first symbol or None
                 style={'width': '50%'}),

    # Show a message if no cryptocurrencies are available
    html.Div(id='no-data-message', style={'color': 'red', 'display': 'none'}),

    # Dropdown for selecting volatility level
    dcc.Dropdown(id='volatility-selector',
                 options=[{'label': 'All', 'value': 'All'},
                          {'label': 'High Volatility', 'value': 'High'},
                          {'label': 'Low Volatility', 'value': 'Low'}],
                 value='All',  # Default value
                 style={'width': '50%', 'margin-top': '10px'}),

    # Graph to display selected cryptocurrency's price data
    dcc.Graph(id='price-chart'),
```

```
    # Graph to display Buy/Sell signal count
    dcc.Graph(id='buy-sell-chart')
])

# Callback to update charts based on selected cryptocurrency and volatility level
@app.callback(
    [dash.dependencies.Output('price-chart', 'figure'),
     dash.dependencies.Output('buy-sell-chart', 'figure'),
     dash.dependencies.Output('no-data-message', 'style')],
    [dash.dependencies.Input('crypto-selector', 'value'),
     dash.dependencies.Input('volatility-selector', 'value')]
)
def update_charts(crypto_name, volatility_level):
    if crypto_name is None:
        return go.Figure(), go.Figure(), {'display': 'block'}  # Show no data messa

    # Filter the DataFrame for the selected cryptocurrency
    df = all_crypto_data.get(crypto_name, pd.DataFrame()).copy()  # Ensure df is a

    # Filter by volatility level if selected
    if volatility_level == 'High':
        df = df[df['Volatility'] > df['Volatility'].median()]
    elif volatility_level == 'Low':
        df = df[df['Volatility'] <= df['Volatility'].median()]

    # Generate charts
    price_chart = create_crypto_line_chart(df, crypto_name)
    buy_sell_chart = create_buy_sell_chart(df, crypto_name)

    return price_chart, buy_sell_chart, {'display': 'none'}  # Hide no data message

if __name__ == '__main__':
    app.run_server(debug=True, port=8055)
```

Top 5 Symbols with Most 'SELL' Signals:
AUCTION/USD: 717 'SELL' signals
MNT/USD: 713 'SELL' signals
GMT/USD: 712 'SELL' signals
NTRN/USD: 711 'SELL' signals
EUL/USD: 700 'SELL' signals

Top 5 Symbols with Most 'BUY' Signals:
DOGE/USD: 371 'BUY' signals
ETH/USD: 348 'BUY' signals
BTC/USD: 338 'BUY' signals
LTC/USD: 241 'BUY' signals
USDT/USD: 239 'BUY' signals

Risk Classification:
BODEN/USD: Low Risk
TNSR/USD: Low Risk
SAFE/USD: Low Risk
APE/USD: Low Risk
BTC/USD: High Risk
DOGE/USD: Low Risk
ADA/USD: Low Risk
PYTH/USD: Low Risk
LTC/USD: Low Risk
USDT/USD: Low Risk
MNT/USD: Unknown Risk
XMR/USD: Low Risk
LINK/USD: Low Risk
AUCTION/USD: Unknown Risk
NOS/USD: Low Risk
ZRX/USD: Low Risk
CPOOL/USD: Low Risk
EUL/USD: Low Risk
GMT/USD: Unknown Risk
SUPER/USD: Low Risk
NTRN/USD: Low Risk
MEME/USD: Low Risk
RENDER/USD: Low Risk
ETH/USD: High Risk
MOG/USD: Low Risk
TREMP/USD: Low Risk

Cryptocurrency Volatility Dashboard

BODEN/USD

All