YFinance: Software Design Specification

YFinance will be used to acquire historical data and calculate daily, weekly, and monthly closing prices on target ticker. The data will then be used to calculate technical indicators such as:

- RSI
- Bollinger
- MA (Simple)
- MACD
- VWAP

Each ticker will be inheriting an entity/equity class. In this case [STOCKS] will be the entity that will need to be modeled out into a class.

Acceptance Criteria:

1. Class should have attributes for time series data for each technical indicator
2. Class should have functions for modeling technical indicators
3. Class should be modular to add additional technical indicators and supporting functions
4. Class should have finite range for time series data for each technical indicator for vectorized math operation performance considerations
5. Class should be inheritable and be able to be used in arbitrary programs and should be agnostic to other micro-services in the ecosystem.

YFinance_StockClass.py –

```
import yfinance as yf

class YFinance_Stock:
    def __init__(self, ticker, data_range=365):
        self.ticker = ticker
        self.data_range = data_range
        self.price = None
        self.RSI = None
        self.MACD = None
        self.BB_upper = None
        self.BB_lower = None
        self.VWAP = None

    def get_price(self, period="1y"):
        # get historical market data
        self.price = yf.Ticker(self.ticker).history(period=period)

    def get_close_prices(self):
        return self.price['Close'] if self.price is not None else None

    def calculate_daily_close(self):
        self.price['daily_close'] = self.price['Close'].resample('D').last()
```

```
    def calculate_weekly_close(self):
        self.price['weekly_close'] = self.price['Close'].resample('W').last()

    def calculate_monthly_close(self):
        self.price['monthly_close'] = self.price['Close'].resample('M').last()

    def calculate_RSI(self, period=14):
        delta = self.price['Close'].diff()
        up, down = delta.copy(), delta.copy()
        up[up < 0] = 0
        down[down > 0] = 0
        gain = up.rolling(window=period).mean()
        loss = abs(down.rolling(window=period).mean())
        RS = gain / loss
        self.RSI = (100 - (100 / (1 + RS)))[-self.data_range:]

    def calculate_MACD(self, short_period=12, long_period=26,
signal_period=9):
        short_EMA = self.price['Close'].ewm(span=short_period,
adjust=False).mean()
        long_EMA = self.price['Close'].ewm(span=long_period,
adjust=False).mean()
        MACD_line = short_EMA - long_EMA
        signal_line = MACD_line.ewm(span=signal_period, adjust=False).mean()
        self.MACD = (MACD_line - signal_line)[-self.data_range:]

    def calculate_BB(self, period=20):
        SMA = self.price['Close'].rolling(window=period).mean()
        std_dev = self.price['Close'].rolling(window=period).std()
        self.BB_upper = (SMA + (2 * std_dev))[-self.data_range:]
        self.BB_lower = (SMA - (2 * std_dev))[-self.data_range:]

    def calculate_VWAP(self):
        cum_volume = self.price['Volume'].cumsum()
        cum_vwap = (self.price['Close'] * self.price['Volume']).cumsum()
        self.VWAP = (cum_vwap / cum_volume)[-self.data_range:]

    def get_latest_values(self):
        return {
            'latest_close': self.get_close_prices()[-1] if self.price is not
None else None,
            'latest_RSI': self.RSI[-1] if self.RSI is not None else None,
            'latest_MACD': self.MACD[-1] if self.MACD is not None else None,
            'latest_BB_upper': self.BB_upper[-1] if self.BB_upper is not None
else None,
            'latest_BB_lower': self.BB_lower[-1] if self.BB_lower is not None
else None,
            'latest_VWAP': self.VWAP[-1] if self.VWAP is not None else None,
            # add lines for additional indicators here
        }
```

1. **Class should have attributes for time series data for each technical indicator:**

```
1          self.price = None
2.         self.RSI = None
3.         self.MACD = None
4.         self.BB_upper = None
5.         self.BB_lower = None
6.         self.VWAP = None
```

are all attributes with time series data
Note: calculating any indicators before getting price will result in error. Always do 'get._price()' before any calculations.

2. **Class should have functions for modeling technical indicators**

Get_price, calculate_RSI, calculate_MACD are all functions that model tech. indicators

3. **Class should be modular to add additional technical indicators and supporting functions**

To add a new technical indicator, define a new attribute and a new method for calculation.

4. **Class should have finite range for time series data for each technical indicator for vectorized math operation performance considerations**

The 'data_range' attribute ensures that each technical indicator only stores the most recent 'data_range' data points. This will improve performance by decrease # of data

5. **Class should be inheritable and be able to be used in arbitrary programs and should be agnostic to other micro-services in the ecosystem.**

Class doesn't have dependences on specific aspects of the programs and can be used in arbitrary programs.

Code Description:

```
import yfinance as yf

class YFinance_Stock:
    def __init__(self, ticker, data_range=365):
        self.ticker = ticker
        self.data_range = data_range
        self.price = None
        self.RSI = None
        self.MACD = None
        self.BB_upper = None
        self.BB_lower = None
        self.VWAP = None
```

This stock class uses __init__ method, takes parameters 'ticker' and 'data_range". 'ticker' parameter takes stock symbol "AAPL", "SPY". 'data_range" parameter is optional and defaults to 365 days. __init__ initializes attributes for price and technical indicators.

```
    def get_price(self, period="1y"):
        # get historical market data
        self.price = yf.Ticker(self.ticker).history(period=period)

    def get_close_prices(self):
        return self.price['Close'] if self.price is not None else None
```

'get_price' method takes the optional parameter 'period' (default 1 year), and uses the 'yf.Ticker' function to retrieve historical market data for a stock and assigns it to 'price' attribute.

'get_close_prices' method checks if historical data has been successfully retrieved, and provides a way to access close prices specifically.

```
    def calculate_daily_close(self):
        self.price['daily_close'] = self.price['Close'].resample('D').last()

    def calculate_weekly_close(self):
        self.price['weekly_close'] = self.price['Close'].resample('W').last()

    def calculate_monthly_close(self):
        self.price['monthly_close'] = self.price['Close'].resample('M').last()
```

calculate_daily_close, calculate_weekly_close, calculate_monthly_close adds a new column to the 'price' DataFrame by using the 'resample' function on the 'Close' column of the 'price' Dataframe, and specifying a frequency of D, W, and M. The 'resample' function is a way to convert the frequency of time-series data. In this case, it takes the last value of the 'close' prive for each D,W, M and assigns it to the 'daily_close','weekly_close'and 'monthly_close' column.

```
    def calculate_RSI(self, period=14):
```

```
        delta = self.price['Close'].diff()
        up, down = delta.copy(), delta.copy()
        up[up < 0] = 0
        down[down > 0] = 0
        gain = up.rolling(window=period).mean()
        loss = abs(down.rolling(window=period).mean())
        RS = gain / loss
        self.RSI = (100 - (100 / (1 + RS)))[-self.data_range:]

    def calculate_MACD(self, short_period=12, long_period=26,
signal_period=9):
        short_EMA = self.price['Close'].ewm(span=short_period,
adjust=False).mean()
        long_EMA = self.price['Close'].ewm(span=long_period,
adjust=False).mean()
        MACD_line = short_EMA - long_EMA
        signal_line = MACD_line.ewm(span=signal_period, adjust=False).mean()
        self.MACD = (MACD_line - signal_line)[-self.data_range:]

    def calculate_BB(self, period=20):
        SMA = self.price['Close'].rolling(window=period).mean()
        std_dev = self.price['Close'].rolling(window=period).std()
        self.BB_upper = (SMA + (2 * std_dev))[-self.data_range:]
        self.BB_lower = (SMA - (2 * std_dev))[-self.data_range:]

    def calculate_VWAP(self):
        cum_volume = self.price['Volume'].cumsum()
        cum_vwap = (self.price['Close'] * self.price['Volume']).cumsum()
        self.VWAP = (cum_vwap / cum_volume)[-self.data_range:]
```

The calculate_RSI method takes an optional parameter period which defaults to 14. The method first computes the price difference between consecutive days and stores it in the delta variable. It then creates two separate series called up and down, which are copies of delta. The up series retains only positive values from delta, while the down series retains only negative values (or zeros). Next, the method calculates the average gain (gain) and average loss (loss) over a rolling window of the specified period. The relative strength (RS) is then calculated by dividing the gain by the absolute value of the loss. Finally, the RSI is computed using the formula: 100 - (100 / (1 + RS)). The RSI values are assigned to the RSI attribute of the object, limited to the last data_range values.

The calculate_MACD method takes three optional parameters: short_period, long_period, and signal_period, which default to 12, 26, and 9 respectively. The method first calculates the short-term exponential moving average (EMA) and long-term EMA for the closing prices using the ewm function. The MACD line is obtained by subtracting the long-term EMA from the short-term EMA. The signal line is then calculated by applying the EMA to the MACD line. The resulting MACD values (MACD line minus signal line) are assigned to the MACD attribute of the object, limited to the last data_range values.

The calculate_BB method calculates takes an optional parameter period which defaults to 20. The method first computes the simple moving average (SMA) and standard deviation (std_dev) over a rolling window of the specified period for the closing prices. The upper Bollinger Band is calculated by adding twice the standard deviation to the SMA, while the lower Bollinger Band is calculated by

subtracting twice the standard deviation from the SMA. The upper and lower Bollinger Band values are assigned to the BB_upper and BB_lower attributes of the object, respectively, limited to the last data_range values.

The calculate_VWAP method calculates the cumulative sum of the volume and the cumulative sum of the product of the closing price and volume. Then, it divides the cumulative sum of the product by the cumulative sum of the volume to obtain the VWAP. The VWAP values are assigned to the VWAP attribute of the object, limited to the last data_range values.

```
    def get_latest_values(self):
        return {
            'latest_close': self.get_close_prices()[-1] if self.price is not
None else None,
            'latest_RSI': self.RSI[-1] if self.RSI is not None else None,
            'latest_MACD': self.MACD[-1] if self.MACD is not None else None,
            'latest_BB_upper': self.BB_upper[-1] if self.BB_upper is not None
else None,
            'latest_BB_lower': self.BB_lower[-1] if self.BB_lower is not None
else None,
            'latest_VWAP': self.VWAP[-1] if self.VWAP is not None else None,
            # add lines for additional indicators here
        }
```

‘get_latest_values’ returns a dictionary containing the lastest values of indicators and metrics for the stock

---