Volatility Strategies Manual

This guide provides an overview of each trading strategy included in the package, with descriptions of their objectives, key components, parameters, logic, and notes on their unique features.

A Note on Backtesting: All strategies presented are for educational and illustrative purposes. The Python scripts provide a framework for backtesting these concepts. Past performance is not indicative of future results. Thorough testing, optimization, risk management, and consideration of transaction costs (slippage, commissions) are crucial before deploying any trading strategy with real capital. The code snippets included in the manual are illustrative and extracted from the provided standalone Python scripts.

Table of Contents

1. Empirical-Mode Decomposition (EMD) Channels Strategy

1.1. Overview and Objective

Overview:
This strategy utilizes Empirical-Mode Decomposition (EMD) to decompose a financial time series (closing prices) into its Intrinsic Mode Functions (IMFs). By summing the highest frequency IMFs, it creates a "noise envelope." A "denoised" signal is then derived by subtracting this noise envelope from the original price. Trading channels are constructed around this denoised signal.

Objective:
The primary goal is to trade breakouts from these EMD-derived channels. The strategy aims to capture significant price movements that emerge when prices push beyond the

dynamically estimated noise boundaries. An Average True Range (ATR) based trailing stop-loss is implemented for risk management.

1.2. Key Indicators and Components

The strategy relies on several calculated series derived from the price data:

- **Input Price Data (`df`):** OHLCV data downloaded using `yfinance` .

```
# --- 1. Download Data ---
df_raw = yf.download(
    [ticker],
    start=start_date,
    end=end_date,
    auto_adjust=False, # User preference
    progress=False
)
if isinstance(df_raw.columns, pd.MultiIndex): # User preference
    df = df_raw.droplevel(level=1, axis=1)
else:
    df = df_raw
df = df[['Open', 'High', 'Low', 'Close', 'Volume']].copy()
```

- **Empirical Mode Decomposition (EMD) & Intrinsic Mode Functions (IMFs):** The `PyEMD` library is used to decompose the closing price series.

```
# Step 2.1: EMD and IMF Processing
price_series_np = df['Close'].values
emd_analyzer = EMD.EMD()
try:
    imfs_all = emd_analyzer.emd(price_series_np)
except AttributeError: # Fallback for potentially older PyEMD syntax
    imfs_all = emd_analyzer(price_series_np)
```

- **Sum of Top K IMFs (Noise Envelope - `sum_top_k_imfs_col`):** The highest frequency IMFs are summed to represent noise.

```
# Ensure we don't try to sum more IMFs than available
actual_k_to_sum = min(emd_top_k_imfs, num_imfs_generated - 1 if
num_imfs_generated > 1 else 1)
# ... (warning if actual_k_to_sum differs from emd_top_k_imfs) ...

sum_top_k_imfs_array = np.sum(imfs_all[:actual_k_to_sum, :], axis=0)
# Assign to DataFrame
df[sum_top_k_imfs_col] = np.nan
len_to_assign = min(len(df), len(sum_top_k_imfs_array))
df.iloc[:len_to_assign, df.columns.get_loc(sum_top_k_imfs_col)] =
```

```
sum_top_k_imfs_array[:len_to_assign]
df[sum_top_k_imfs_col] =
df[sum_top_k_imfs_col].fillna(method='bfill').fillna(method='ffill')
```

- **Denoised Signal (`denoised_signal_col`):** The original signal with the noise envelope removed.

```
# Step 2.2: Create "Denoised" Signal and EMD Bands
df[denoised_signal_col] = df['Close'] - df[sum_top_k_imfs_col]
```

- **EMD Channels (`upper_band_col_emd`, `lower_band_col_emd`):** Bands created around the denoised signal.

```
df[upper_band_col_emd] = df[denoised_signal_col] + band_multiplier_emd *
np.abs(df[sum_top_k_imfs_col])
df[lower_band_col_emd] = df[denoised_signal_col] - band_multiplier_emd *
np.abs(df[sum_top_k_imfs_col])
```

- **Average True Range (ATR - `atr_col_name_sl`):** Used for the trailing stop-loss.

```
# Step 2.3: ATR for Stop Loss
df['H-L_sl'] = df['High'] - df['Low']
df['H-PC_sl'] = np.abs(df['High'] - df['Close'].shift(1))
df['L-PC_sl'] = np.abs(df['Low'] - df['Close'].shift(1))
df['TR_sl'] = df[['H-L_sl', 'H-PC_sl', 'L-PC_sl']].max(axis=1)
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

1.3. Strategy Parameters

The user-configurable parameters are defined at the beginning of the script:

```
# --- Parameters ---
ticker = "BTC-USD"          # Example: "AAPL", "EURUSD=X"
start_date = "2021-01-01"   # EMD benefits from longer series
end_date = "2024-12-31"

# EMD Parameters
emd_top_k_imfs = 2          # Number of top (highest frequency) IMFs to sum
for the noise envelope
                            # IMF0, IMF1, ... IMF(K-1) will be summed.
# EMD Channel Parameters
band_multiplier_emd = 1.0   # Multiplier for the absolute sum of top K IMFs
to set band width

# ATR Trailing Stop Parameters
```

```
atr_window_sl = 14
atr_multiplier_sl = 1.0

# Trading days per year
# TRADING_DAYS_PER_YEAR = 252
TRADING_DAYS_PER_YEAR = 365
```

1.4. Entry Logic

Entries are executed at the `Open` price of the current day (`today_open`) if conditions based on the *previous day's* data are met. The logic is iterated within the main backtesting loop.

Python

```
# --- 3. Strategy Logic & Backtesting Loop ---
# ... (initialization of active_position, entry_price, active_trailing_stop)
...
for i in range(len(df_analysis)):
    # ... (data loading for today_idx, prev_idx, prev_prev_idx) ...
    # ... (critical value checks and continue if NaN) ...

    # If not stopped out by ATR (action_taken_this_step is False)
    if not action_taken_this_step:
        target_signal_position = 0
        # Long Breakout Signal: prev_close broke above prev_upper_band
        if prev_close > prev_upper_band:
            is_fresh_breakout_up = True
            if prev_prev_idx: # Check bar before previous
                prev_prev_close = df_analysis.at[prev_prev_idx, 'Close']
                prev_prev_upper_band = df_analysis.at[prev_prev_idx,
upper_band_col_emd]
                if pd.notna(prev_prev_close) and
pd.notna(prev_prev_upper_band) and prev_prev_close > prev_prev_upper_band:
                    is_fresh_breakout_up = False # Already broken out
            if is_fresh_breakout_up or current_day_assumed_position == 0 :
target_signal_position = 1

        # Short Breakout Signal: prev_close broke below prev_lower_band
        elif prev_close < prev_lower_band:
            is_fresh_breakout_down = True
            if prev_prev_idx:
                prev_prev_close = df_analysis.at[prev_prev_idx, 'Close']
                prev_prev_lower_band = df_analysis.at[prev_prev_idx,
lower_band_col_emd]
```

```
                if pd.notna(prev_prev_close) and
pd.notna(prev_prev_lower_band) and prev_prev_close < prev_prev_lower_band:
                    is_fresh_breakout_down = False
            if is_fresh_breakout_down or current_day_assumed_position == 0:
target_signal_position = -1

        # Execute if signal changes or entering from flat
        if target_signal_position != 0 and target_signal_position !=
current_day_assumed_position :
            # ... (calculate pnl_exit_component if flipping position) ...
            current_day_assumed_position = target_signal_position

            if current_day_assumed_position == 1: # Entering New Long at
today_open
                current_day_assumed_entry_price = today_open
                # ... (calculate pnl_entry_component, set initial_ts and
current_day_assumed_trailing_stop) ...
            elif current_day_assumed_position == -1: # Entering New Short at
today_open
                current_day_assumed_entry_price = today_open
                # ... (calculate pnl_entry_component, set initial_ts and
current_day_assumed_trailing_stop) ...
            # ... (calculate pnl_for_day based on flip or new entry) ...
            action_taken_this_step = True
```

1.5. Exit Logic

Exits are primarily handled by an ATR-based trailing stop-loss. Positions are also exited if an opposing entry signal is generated (stop-and-reverse).

- **ATR Trailing Stop-Loss (checked at the start of each iteration for an active position):** Python

```
# 1. Check ATR Stop Loss
if current_day_assumed_position == 1 and
pd.notna(current_day_assumed_trailing_stop) and
pd.notna(current_day_assumed_entry_price):
    if today_low <= current_day_assumed_trailing_stop:
        exit_price_sl = min(today_open,
current_day_assumed_trailing_stop)
        pnl_for_day = (exit_price_sl / current_day_assumed_entry_price)
- 1
        current_day_assumed_position = 0
```

```
        action_taken_this_step = True
elif current_day_assumed_position == -1 and
pd.notna(current_day_assumed_trailing_stop) and
pd.notna(current_day_assumed_entry_price):
    if today_high >= current_day_assumed_trailing_stop:
        exit_price_sl = max(today_open,
current_day_assumed_trailing_stop)
        pnl_for_day = -((exit_price_sl /
current_day_assumed_entry_price) - 1)
        current_day_assumed_position = 0
        action_taken_this_step = True

if action_taken_this_step: # If stopped out
    current_day_assumed_entry_price = np.nan
    current_day_assumed_trailing_stop = np.nan
```

- **Trailing Stop Adjustment (while holding a position and not stopped out):** Python

```
# (within the 'else' block of 'if not action_taken_this_step', if 'elif
current_day_assumed_position != 0:')
elif current_day_assumed_position != 0: # Holding position
    if current_day_assumed_position == 1:
        # ... (calculate pnl_for_day) ...
        current_day_assumed_trailing_stop =
max(current_day_assumed_trailing_stop, today_close - atr_multiplier_sl *
today_atr_sl)
    elif current_day_assumed_position == -1:
        # ... (calculate pnl_for_day) ...
        current_day_assumed_trailing_stop =
min(current_day_assumed_trailing_stop, today_close + atr_multiplier_sl *
today_atr_sl)
```

- **Exit due to Signal Reversal:** If `target_signal_position` is opposite to `current_day_assumed_position`, the old position is exited (PnL calculated based on `today_open`) before the new position is entered. This is handled within the "Entry Logic" section (`if target_signal_position != 0 and target_signal_position != current_day_assumed_position :`).

1.6. Backtesting & Performance Evaluation

The script includes a loop for backtesting and functions to calculate performance.

- **Daily Return Calculation:** Strategy returns are calculated daily within the loop. Python

```
# Assign results for the day
df_analysis.at[today_idx, 'Position'] = active_position
df_analysis.at[today_idx, 'Strategy_Daily_Return'] = pnl_for_day
df_analysis.at[today_idx, 'Trailing_Stop'] = active_trailing_stop
```

- **Cumulative Returns and Alignment:** Python

```
# --- Post-Loop Calculations ---
df_analysis['Strategy_Daily_Return'].fillna(0, inplace=True)
df_analysis['Cumulative_Strategy_Return'] = (1 +
df_analysis['Strategy_Daily_Return']).cumprod()
df_analysis['Cumulative_Buy_Hold_Return'] = (1 +
df_analysis['Daily_Return'].fillna(0)).cumprod()
# Align Buy & Hold to start at the same point as strategy
# ... (logic for Cumulative_Buy_Hold_Return_Aligned) ...
```

- **Performance Metrics Function (`calc_performance_metrics`):** Python

```
# --- 4. Performance Metrics ---
def calc_performance_metrics(returns, name,
trading_days_per_year=TRADING_DAYS_PER_YEAR):
    if len(returns) < 2:
        # ... (handle insufficient data) ...
        return
    avg_daily_return = returns.mean()
    std_daily_return = returns.std()
    ann_ret = avg_daily_return * trading_days_per_year
    ann_vol = std_daily_return * np.sqrt(trading_days_per_year)
    sharpe_ratio = ann_ret / ann_vol if ann_vol > 1e-7 else np.nan
    cumulative_return_factor = (1 + returns).prod()
    # ... (print metrics) ...

# Call for strategy and buy & hold
calc_performance_metrics(strat_returns, strategy_name_detail)
calc_performance_metrics(bh_returns, f"{ticker} Buy & Hold")
```

1.7. Plotting Results

The script generates a multi-panel plot for visual analysis if sufficient data is available.

Python

```
# --- 5. Plotting ---
if not df_analysis.empty and len(df_analysis) > 5:
    fig, axs = plt.subplots(5, 1, figsize=(15, 25), sharex=True)

    # Panel 0: Price, Denoised Signal, EMD Bands, Trailing Stops
    axs[0].plot(df_analysis.index, df_analysis['Close'], ...)
    axs[0].plot(df_analysis.index, df_analysis[denoised_signal_col], ...)
    # ... (other plots for panel 0) ...

    # Panel 1: Sum of Top K IMFs
    axs[1].plot(df_analysis.index, df_analysis[sum_top_k_imfs_col], ...)
    # ... (plot individual IMFs if actual_k_to_sum <=3) ...

    # Panel 2: Strategy Position
    axs[2].plot(df_analysis.index, df_analysis['Position'], ...)

    # Panel 3: ATR for Trailing Stop
    axs[3].plot(df_analysis.index, df_analysis[atr_col_name_sl], ...)

    # Panel 4: Cumulative Performance Comparison
    axs[4].plot(df_analysis.index,
df_analysis['Cumulative_Strategy_Return'], ...)
    axs[4].plot(df_analysis.index,
df_analysis['Cumulative_Buy_Hold_Return_Aligned'], ...)

    plt.tight_layout()
    plt.show()
```

1.8. Unique Features & Notes

- **Adaptive Channels:** The EMD channels dynamically adapt to changing market structure and volatility as reflected by the IMFs.
- **Noise Filtering Concept:** The strategy attempts to filter out market "noise" by identifying and using the highest frequency IMFs.
- **EMD Dependency:** Relies heavily on the `PyEMD` library and the quality of the EMD decomposition. The script includes a `try-except` block for importing `PyEMD` and provides installation instructions if not found.
- **Computational Cost:** EMD can be computationally intensive, especially for long data series.
- **Parameter Sensitivity:** Strategy performance can be sensitive to `emd_top_k_imfs`, `band_multiplier_emd`, and ATR parameters. Tuning may be required for different assets or timeframes.

- **"Fresh Breakout" Logic:** Includes a condition to prefer initial breakouts over sustained periods outside the bands.
- **Data Requirements:** Works best with longer time series for stable EMD results. A warning is issued for short series (less than 500 data points).
- **Backtest Assumptions:** Assumes trades at open, no slippage or commissions.
- **Column Naming:** Dynamically generated column names (e.g., `sum_top_k_imfs_col`, `upper_band_col_emd`) are used for clarity and to reflect key parameters in the column titles.
- **Warnings Suppression:** The script filters some common warnings from libraries to keep the output cleaner.

---

2. HAR-Model Volatility-Forecast Breakout Strategy

2.1. Overview and Objective

Overview:

This strategy employs a Heterogeneous Autoregressive (HAR) model to forecast daily realized volatility. The HAR model is a simple yet effective time series model that predicts future volatility using a linear combination of past realized volatilities observed over different time horizons (typically daily, weekly, and monthly averages). The strategy then uses these forecasts to identify potential volatility breakout events.

Objective:

The primary objective is to enter a long position when the actual realized volatility on a given day significantly exceeds the HAR model's forecasted volatility for that same day. The premise is that such a breakout might signal the beginning of a higher volatility period, potentially leading to larger price movements. The position is typically held for the next day.

2.2. Key Functions and Components

The script is structured with several helper functions to manage data acquisition, feature engineering, backtesting, and results presentation.

- Data Acquisition (get_data function): This function downloads historical stock data using yfinance, applying user-specified preferences for auto_adjust=False and droplevel for MultiIndex columns from yfinance. It also standardizes the 'Close' price to use 'Adj Close' if available. Python

```
# --- Data Acquisition (Adhering to User Preferences) ---
def get_data(ticker, start_date, end_date):
    """
```

```
    Downloads historical stock data using yfinance.
    Applies user-specified download preferences.
    """
    if isinstance(ticker, str):
        tickers_list = [ticker]
    else:
        tickers_list = ticker # Assuming it's already a list

    data = yf.download(
        tickers=tickers_list,
        start=start_date,
        end=end_date,
        auto_adjust=False,  # User preference
        progress=False
    )

    if data.empty:
        print(f"No data downloaded for {ticker}. Check ticker symbol or
date range.")
        return pd.DataFrame()

    if isinstance(data.columns, pd.MultiIndex):
        if len(tickers_list) == 1:
            data = data.droplevel(axis=1, level=1) # User preference

    if 'Adj Close' in data.columns:
        data['Close'] = data['Adj Close']
    elif 'Close' not in data.columns:
        raise ValueError("DataFrame must contain 'Close' or 'Adj Close'
column.")

    cols_to_use = ['Open', 'High', 'Low', 'Close', 'Volume']
    data = data[[col for col in cols_to_use if col in data.columns]]
    return data
```

- Realized Volatility Calculation (calculate_realized_volatility function): This function calculates daily log returns and uses their absolute values as a proxy for daily realized volatility (rv). Python

```
# --- Feature Engineering ---
def calculate_realized_volatility(df):
    """
    Calculates daily log returns and realized volatility (absolute log
```

```
returns).
    """
    df['log_return'] = np.log(df['Close'] / df['Close'].shift(1))
    # Using absolute log returns as a proxy for daily realized
volatility
    df['rv'] = df['log_return'].abs()
    return df.dropna() # Drop first NaN from log_return and rv
```

- HAR Feature Preparation (prepare_har_features function): This function creates the lagged realized volatility features required by the HAR model: daily lag (rv_lag_1), 5-day average lag (rv_lag_weekly), and 22-day average lag (rv_lag_monthly). These lags are based on information available up to day t-1 for predicting volatility on day t. Python

```
def prepare_har_features(df):
    """
    Prepares lagged features for the HAR model.
    The model predicts RV_t using information up to t-1.
    RV_k = beta_0 + beta_d*RV_{k-1} + beta_w*RV_avg_{k-1 to k-5} +
beta_m*RV_avg_{k-1 to k-22}
    """
    df['rv_lag_1'] = df['rv'].shift(1) # RV_{k-1}
    df['rv_lag_weekly'] = df['rv'].shift(1).rolling(window=5).mean() #
Avg RV over past 5 days, ending at k-1
    df['rv_lag_monthly'] = df['rv'].shift(1).rolling(window=22).mean() #
Avg RV over past 22 days, ending at k-1
    return df.dropna() # Drop NaNs from lags and rolling means
```

2.3. Main Execution Parameters

These parameters are set in the `if __name__ == '__main__':` block and control the overall execution of the backtest:

Python

```
if __name__ == '__main__':
    TICKER = 'BTC-USD'
    START_DATE = '2020-01-01'
    END_DATE = '2024-12-31'

    HAR_TRAIN_WINDOW = 30        # Days for rolling HAR model training
    BREAKOUT_THRESHOLD_PCT = 10  # Enter if RV_actual > RV_forecast by this
X%
```

- `TICKER` (String): The asset to backtest.
- `START_DATE` (String): Start date for data download.
- `END_DATE` (String): End date for data download.
- `HAR_TRAIN_WINDOW` (Integer): The length of the rolling window (in days) used to train the HAR model.
- `BREAKOUT_THRESHOLD_PCT` (Integer/Float): The percentage by which actual realized volatility must exceed forecasted realized volatility to trigger a breakout signal.

2.4. Trading Logic (within `run_har_breakout_backtest` function)

The core trading logic resides in the `run_har_breakout_backtest` function. It iterates through the data, re-training the HAR model and generating forecasts on a rolling basis.

2.4.1. HAR Model Training and Forecasting

On each day `t` in the backtest (after an initial warm-up period equal to `har_train_window`), the HAR model is trained using data up to `t-1`.

Python

```
# (Inside run_har_breakout_backtest loop)
for t_idx in range(har_train_window, len(df)):
    current_time = df.index[t_idx]
    train_df = df.iloc[t_idx - har_train_window : t_idx]

    Y_train = train_df['rv']
    X_train_base = train_df[feature_names] # feature_names = ['rv_lag_1',
'rv_lag_weekly', 'rv_lag_monthly']
    # ... (NaN checks for X_train_base and Y_train) ...
    X_train = sm.add_constant(X_train_base)

    model = sm.OLS(Y_train, X_train, missing='drop').fit()

    # Forecast RV for current day 't' using features from 't-1' (which are
in df.iloc[t_idx])
    current_X_base_for_pred = df.iloc[t_idx:t_idx+1][feature_names]
    # ... (NaN check for current_X_base_for_pred) ...
    current_X_for_pred_df = sm.add_constant(current_X_base_for_pred,
has_constant='add')

    rv_forecast_t_series = model.predict(current_X_for_pred_df)
```

```
    # ... (extraction of rv_forecast_t from series) ...
    forecasts.loc[current_time] = rv_forecast_t
```

The model (`sm.OLS`) predicts the realized volatility (`rv`) for the current day `t` (`rv_forecast_t`) using the lagged features available from day `t-1` .

2.4.2. Entry Condition

A long position is signaled if the *actual* realized volatility on day `t` (`rv_actual_t`) exceeds the *forecasted* realized volatility for day `t` (`rv_forecast_t`) by more than the `BREAKOUT_THRESHOLD_PCT` .

Python

```
# (Inside run_har_breakout_backtest loop, after forecast)
    rv_actual_t = df['rv'].iloc[t_idx]

    if pd.notna(rv_actual_t) and pd.notna(rv_forecast_t):
        if rv_actual_t > rv_forecast_t * breakout_multiplier: #
breakout_multiplier = 1 + (BREAKOUT_THRESHOLD_PCT / 100.0)
            if t_idx + 1 < len(df):
                positions.loc[df.index[t_idx + 1]] = 1.0
```

If a breakout is detected on day `t` , a long position (`1.0`) is recorded for the *next day* (`t+1`).

2.4.3. Position Sizing & Holding Period

- **Position Sizing:** The strategy takes a fixed position of `1.0` (long) when a signal occurs. It does not employ fractional sizing.
- **Holding Period:** The position is entered on day `t+1` based on the signal from day `t` . The calculation `strategy_returns = positions * df['log_return']` implies that the return for the position is based on the log return of day `t+1` . Since positions are re-evaluated daily, and a position is only set to `1.0` for the *next day* if a signal occurs, this effectively results in a **1-day holding period** for each trade. If no new signal occurs, the position for the subsequent day defaults to `0.0` (flat).

2.4.4. Exit Condition

There is no explicit stop-loss or take-profit mechanism within the `run_har_breakout_backtest` loop for an active trade. Exits are implicitly managed:

- The position is held for one day (day `t+1`).
- On day `t+1` , the HAR model is re-trained, a new forecast is made for day `t+1` 's volatility, and the actual RV for `t+1` is observed.

- A new decision to hold a position for day `t+2` is made based on whether `rv_actual_t+1` breaks out above `rv_forecast_t+1`. If not, the position for `t+2` becomes `0.0` (flat), effectively exiting any position held on `t+1`.

2.5. Performance Evaluation (`calculate_performance_metrics` function)

The script calculates various performance metrics for the strategy and a buy-and-hold benchmark. Log returns are primarily used.

Python

```
def calculate_performance_metrics(log_returns, label="Strategy"):
    # ... (NaN/empty checks) ...
    simple_returns = np.exp(log_returns) - 1
    total_return_cumulative = np.exp(log_returns.sum()) - 1

    num_days = len(log_returns)
    annualized_return = (np.exp(log_returns.sum()))**(252.0/num_days) - 1 if
num_days > 0 else 0
    annualized_volatility = log_returns.std() * np.sqrt(252)
    sharpe_ratio = (np.exp(log_returns.mean() * 252) -1) /
annualized_volatility if annualized_volatility != 0 else 0

    cumulative_returns_abs = np.exp(log_returns.cumsum())
    peak = cumulative_returns_abs.expanding(min_periods=1).max()
    drawdown = (cumulative_returns_abs - peak) / peak
    max_drawdown_pct = drawdown.min()

    # Approximate number of trades (assumes each block of non-zero positions
is one trade)
    num_trades = (log_returns != 0).astype(int).diff().fillna(0).abs().sum()
/ 2

    active_days = log_returns[log_returns != 0]
    win_rate = (active_days > 0).sum() / len(active_days) if
len(active_days) > 0 else 0

    # ... (print statements for metrics) ...
```

Metrics include: Total Cumulative Return, Annualized Return (Geometric), Annualized Volatility (Log Returns), Sharpe Ratio, Max Drawdown, approximate Number of Trades, and Win Rate (of active trade days).

2.6. Plotting Results (`plot_results` function)

The `plot_results` function generates several charts to visualize the strategy's performance and behavior:

1. **Cumulative Returns:** Compares the strategy's cumulative returns (exponentiated cumulative log returns) against a buy-and-hold benchmark.

   Python

   ```
   # Cumulative Log Returns
   plt.figure(figsize=(12, 6))
   strategy_log_returns.cumsum().apply(np.exp).plot(label=f'{ticker_name}
   Strategy Cumulative Returns', lw=2)
   # ... (benchmark plotting) ...
   plt.title(f'Cumulative Returns: {ticker_name} HAR Volatility Breakout
   vs. Benchmark')
   # ... (labels, legend, show) ...
   ```

2. **Actual vs. Forecasted Volatility:** Shows the daily actual realized volatility, the HAR model's forecasted volatility, and scatter points indicating when volatility breakout signals occurred.

   Python

   ```
   # Actual vs. Forecasted RV & Breakout Signals
   plt.figure(figsize=(14, 7))
   analysis_df['Actual_RV'].plot(label='Actual Realized Volatility
   (Daily)', alpha=0.7)
   analysis_df['Forecast_RV'].plot(label='HAR Forecasted Volatility',
   linestyle='--', alpha=0.9)
   breakout_signals = analysis_df[analysis_df['Position'].shift(-1) == 1.0]
   if not breakout_signals.empty:
       plt.scatter(breakout_signals.index, breakout_signals['Actual_RV'],
   ...)
   # ... (title, labels, legend, show) ...
   ```

3. **Strategy Positions Over Time:** A step plot showing when the strategy held a long position (1.0) or was flat (0.0).

   Python

   ```
   # Strategy Positions
   plt.figure(figsize=(12, 4))
   ```

```
analysis_df['Position'].plot(label='Strategy Position',
drawstyle='steps-post')
# ... (title, labels, legend, show) ...
```

2.7. Unique Features & Notes

- **Volatility Forecasting Model:** Utilizes a formal statistical model (HAR via OLS) to forecast volatility, rather than relying solely on fixed indicator rules.
- **Rolling Window Training:** The HAR model is re-trained on a rolling window, allowing it to adapt to changing market conditions over time.
- **Breakout Confirmation:** The strategy waits for actual realized volatility to confirm a breakout above the forecast, rather than trading on the forecast alone.
- **Long-Only:** This implementation is a long-only strategy, entering on positive volatility shocks.
- **1-Day Holding Period:** Trades are implicitly held for one day.
- **Proxy for Realized Volatility:** Uses absolute daily log returns as a proxy for realized volatility. More sophisticated measures (e.g., using intraday data) could be used if available.
- **Parameter Sensitivity:** The `HAR_TRAIN_WINDOW` and `BREAKOUT_THRESHOLD_PCT` are key parameters that can significantly affect performance and should be tuned.
- **Data Requirements:** Requires sufficient historical data to cover the initial largest HAR lag (22 days) plus the `HAR_TRAIN_WINDOW` before backtesting can effectively begin. The script includes a check for this.
- **Statsmodels Dependency:** Relies on `statsmodels` for OLS regression.

---

3. Intraday Volatility Breakout Strategy

3.1. Overview and Objective

Overview:

This strategy operates on intraday (typically 5-minute) price bars. It identifies periods of low volatility, where the trading range has compressed, by comparing the current Average True Range (ATR) to its moving average. When such a compression is detected, the strategy anticipates a potential breakout.

Objective:

The primary objective is to enter a trade (long or short) when the price breaks out of a recently established rolling high/low channel, but only if the breakout occurs during a period of identified range compression. The aim is to capture sharp price movements that often

follow periods of very low volatility. An ATR-based trailing stop-loss is used for risk management on active trades.

3.2. Key Indicators and Components

The strategy uses several indicators calculated on the intraday bars:

- **Rolling Price Channel (`upper_channel_col`, `lower_channel_col`):** Defines the recent trading range. Python

```
# Rolling Channel
df[upper_channel_col] = df['High'].rolling(window=channel_window).max()
df[lower_channel_col] = df['Low'].rolling(window=channel_window).min()
```

- **ATR for Range Compression Analysis:**
  - `atr_calc_col`: The primary ATR calculation.
  - `atr_ma_comp_col`: A moving average of the `atr_calc_col`.
  - `is_compressed_col`: A boolean flag indicating if the range is compressed (`atr_calc_col` is significantly below its MA).

  Python

```
# ATR for Range Compression Calculation
df['H-L_calc'] = df['High'] - df['Low']
df['H-PC_calc'] = np.abs(df['High'] - df['Close'].shift(1))
df['L-PC_calc'] = np.abs(df['Low'] - df['Close'].shift(1))
df['TR_calc'] = df[['H-L_calc', 'H-PC_calc', 'L-PC_calc']].max(axis=1)
df[atr_calc_col] = df['TR_calc'].rolling(window=atr_calc_window).mean()

df[atr_ma_comp_col] =
df[atr_calc_col].rolling(window=atr_ma_window_for_compression).mean()
df[is_compressed_col] = df[atr_calc_col] < (df[atr_ma_comp_col] *
atr_compression_factor)
```

- **ATR for Trailing Stop-Loss (`atr_tsl_col`):** Used to manage risk on open positions. Python

```
# ATR for Trailing Stop Loss
df['H-L_tsl'] = df['High'] - df['Low']
df['H-PC_tsl'] = np.abs(df['High'] - df['Close'].shift(1))
df['L-PC_tsl'] = np.abs(df['Low'] - df['Close'].shift(1))
```

```
df['TR_tsl'] = df[['H-L_tsl', 'H-PC_tsl', 'L-PC_tsl']].max(axis=1)
df[atr_tsl_col] = df['TR_tsl'].rolling(window=atr_tsl_window).mean()
```

3.3. Script Parameters

User-configurable parameters are defined at the beginning of the script:

Python

```
# --- Parameters ---
ticker = "SOL-USD"          # Example: "EURUSD=X", "SPY", "MSFT"
intraday_interval = "5m"

data_download_period = "30d" # Fetches last N days of data for the specified
interval

# Rolling Channel Parameters
channel_window = 3 * 12     # Number of 5-min bars for High/Low channel
(e.g., 3 hours for 5m bars)

# ATR for Range Compression
atr_calc_window = 14        # Window for calculating the 5-min ATR
atr_ma_window_for_compression = 12 # Window for the Moving Average of ATR
atr_compression_factor = 0.8 # ATR must be < (ATR_MA * factor) for
compression

# ATR Trailing Stop Parameters
atr_tsl_window = 14         # Window for ATR used in Trailing Stop Loss
atr_tsl_multiplier = 1.0    # Multiplier for ATR TSL

# Trading days per year (for annualizing metrics from daily aggregated
returns)
TRADING_DAYS_PER_YEAR = 252
if "USD" in ticker.upper() and any(crypto_sym in ticker.upper() for
crypto_sym in ['BTC', 'ETH', 'SOL', 'ADA']):
    TRADING_DAYS_PER_YEAR = 365 # Crypto trades 24/7
```

3.4. Data Handling

- **Intraday Data Download:** The script downloads intraday data using `yfinance`. It uses the `period` argument (e.g., "30d") as `yfinance` has limitations on date ranges for very fine intervals like "5m" (often max 60 days). Python

```
# --- 1. Download Data ---
df_raw = yf.download(
    tickers=ticker,
    period=data_download_period,
    interval=intraday_interval,
    auto_adjust=False, # Per user preference
    progress=False
).droplevel(1, 1) # Per user preference for single ticker
```

- **Timezone Localization:** The script attempts to localize the DataFrame's index to UTC if it's naive, or convert it if it's localized to a different timezone than the system's current timezone. This is important for consistent intraday analysis. Python

```
if df.index.tz is None:
    try:
        df.index = df.index.tz_localize('UTC')
        print("Localized DataFrame index to UTC.")
    except Exception as e:
        print(f"Could not localize index (may already be localized or
it's naive): {e}")
elif df.index.tzinfo != pd.Timestamp.now().tzinfo: # If localized but to
different zone
    print(f"Converting index from {df.index.tzinfo} to
{pd.Timestamp.now().tzinfo}")
    df.index = df.index.tz_convert(pd.Timestamp.now().tzinfo)
```

3.5. Trading Logic (within the main backtesting loop)

The strategy iterates bar-by-bar through the intraday data (`df_analysis`).

3.5.1. Range Compression Check

For a new trade to be considered, the range on the *previous bar* must have been identified as compressed.

Python

```
# (Inside the backtesting loop)
    # Data from previous bar for signals
    prev_is_range_compressed = df_analysis.at[prev_idx, is_compressed_col]

    # Check for New Breakout Entries (if flat and range is compressed)
```

```
 if current_bar_assumed_position == 0 and prev_is_range_compressed:
     # ... entry logic follows ...
```

3.5.2. Entry Conditions

If flat and the previous bar's range was compressed, the strategy looks for breakouts on the current bar:

- **Long Breakout:** Python

```
# Long Breakout
if today_high > prev_upper_channel and today_open <= prev_upper_channel
: # Price broke upwards
    entry_exec_price = prev_upper_channel # Assume entry at breakout
level
    if today_low <= entry_exec_price : # Ensure price actually traded
at/above breakout
        current_bar_assumed_position = 1
        current_bar_assumed_entry_price = entry_exec_price
        pnl_for_this_bar = (today_close /
current_bar_assumed_entry_price) - 1
        initial_ts = current_bar_assumed_entry_price -
atr_tsl_multiplier * prev_atr_tsl
        current_bar_assumed_trailing_stop = max(initial_ts, today_close
- atr_tsl_multiplier * today_atr_tsl)
        action_occurred_this_bar = True
        entered_long_this_bar = True
elif today_open > prev_upper_channel: # Gapped up over channel
    entry_exec_price = today_open
    # ... (similar logic for position, P&L, stop) ...
    action_occurred_this_bar = True
    entered_long_this_bar = True
```

  Entry is assumed at the `prev_upper_channel` level if price crosses it intra-bar, or at `today_open` if price gapped above.
- **Short Breakout:** (Only if not already entered long on the same bar) Python

```
# Short Breakout (only if not already entered long)
if not entered_long_this_bar:
    if today_low < prev_lower_channel and today_open >=
prev_lower_channel: # Price broke downwards
        entry_exec_price = prev_lower_channel # Assume entry at breakout
level
```

```
        if today_high >= entry_exec_price: # Ensure price actually
traded at/below breakout
            current_bar_assumed_position = -1
            current_bar_assumed_entry_price = entry_exec_price
            pnl_for_this_bar = -((today_close /
current_bar_assumed_entry_price) - 1)
            initial_ts = current_bar_assumed_entry_price +
atr_tsl_multiplier * prev_atr_tsl
            current_bar_assumed_trailing_stop = min(initial_ts,
today_close + atr_tsl_multiplier * today_atr_tsl)
            action_occurred_this_bar = True
            entered_short_this_bar = True
    elif today_open < prev_lower_channel: # Gapped down below channel
        entry_exec_price = today_open
        # ... (similar logic for position, P&L, stop) ...
        action_occurred_this_bar = True
        entered_short_this_bar = True
```

Entry is assumed at the `prev_lower_channel` level if price crosses it intra-bar, or at `today_open` if price gapped below.

3.5.3. Exit Conditions

Exits are primarily managed by an ATR-based trailing stop-loss.

- **ATR Trailing Stop-Loss Check:** Python

```
# 1. Check ATR Trailing Stop Loss (if in a position)
if current_bar_assumed_position == 1 and
pd.notna(current_bar_assumed_trailing_stop) and
pd.notna(current_bar_assumed_entry_price):
    if today_low <= current_bar_assumed_trailing_stop:
        exit_price_sl = min(today_open,
current_bar_assumed_trailing_stop) # Realistic exit
        pnl_for_this_bar = (exit_price_sl /
current_bar_assumed_entry_price) - 1
        current_bar_assumed_position = 0
        action_occurred_this_bar = True
elif current_bar_assumed_position == -1 and
pd.notna(current_bar_assumed_trailing_stop) and
pd.notna(current_bar_assumed_entry_price):
    if today_high >= current_bar_assumed_trailing_stop:
        exit_price_sl = max(today_open,
current_bar_assumed_trailing_stop) # Realistic exit
```

```
        pnl_for_this_bar = -((exit_price_sl /
current_bar_assumed_entry_price) - 1)
        current_bar_assumed_position = 0
        action_occurred_this_bar = True

if action_occurred_this_bar and current_bar_assumed_position == 0: # If
stop loss was hit
    current_bar_assumed_entry_price = np.nan
    current_bar_assumed_trailing_stop = np.nan
```

- **Trailing Stop Adjustment (while holding a position):** The trailing stop is updated at the end of each bar if the position is maintained. Python

```
# (If holding a position and no stop/new entry on current bar)
if not action_occurred_this_bar and current_bar_assumed_position != 0:
    if current_bar_assumed_position == 1:
        # ... (calculate pnl_for_this_bar based on Close vs prev_close)
...
        current_bar_assumed_trailing_stop =
max(current_bar_assumed_trailing_stop, today_close - atr_tsl_multiplier
* today_atr_tsl)
    elif current_bar_assumed_position == -1:
        # ... (calculate pnl_for_this_bar) ...
        current_bar_assumed_trailing_stop =
min(current_bar_assumed_trailing_stop, today_close + atr_tsl_multiplier
* today_atr_tsl)
```

3.5.4. Position Sizing & P&L Calculation

- **Position Sizing:** Positions are binary (1 for long, -1 for short, 0 for flat).
- **P&L Calculation (`Strategy_Bar_Return`):** Profit and loss are calculated on a bar-by-bar basis.
  - On entry: P&L is from entry price to `today_close` .
  - On stop-loss: P&L is from entry price to `exit_price_sl` .
  - While holding: P&L is `(today_close / prev_close) - 1` (or its inverse for shorts).

3.6. Performance Evaluation

Since the strategy operates on intraday bars, the bar-level returns (`Strategy_Bar_Return`) are first aggregated to daily returns. Standard performance metrics are then calculated on these daily returns.

- **Aggregation to Daily Returns:** Python

```
# --- Post-Loop Calculations ---
if not df_analysis.empty:
    df_analysis['Date'] = df_analysis.index.date
    daily_strat_returns = df_analysis.groupby('Date')
['Strategy_Bar_Return'].apply(lambda x: (1 + x).prod() - 1).fillna(0)
    daily_bh_returns =
df_analysis['Close'].resample('D').last().pct_change().loc[daily_strat_r
eturns.index].fillna(0)
```

- **Performance Metrics Function (`calc_performance_metrics`):** This function calculates cumulative return, annualized return, annualized volatility, and Sharpe ratio based on the daily aggregated returns. Python

```
# --- 4. Performance Metrics (based on Daily Aggregated Returns) ---
def calc_performance_metrics(returns_daily, name,
trading_days_per_year=TRADING_DAYS_PER_YEAR):
    # ... (calculation of avg_daily_return, std_daily_return, ann_ret,
ann_vol, sharpe_ratio, cumulative_return_factor) ...
    print(f"\n--- {name} ---")
    print(f"Cumulative Return: {cumulative_return_factor:.2f}x")
    # ... (other print statements) ...
```

3.7. Plotting Results

The script generates a 4-panel plot:

1. **Price, Channel, Trailing Stops (Intraday):** Shows a subset of intraday bars (e.g., the last day or last 300 bars) with the closing price, trading channel, and active trailing stops.
2. **ATR and Compression State (Intraday):** Displays the ATR, its moving average, and shaded regions indicating when the range was compressed, for the same intraday subset.
3. **Strategy Position (Intraday):** A step plot of the strategy's position (Long/Short/Flat) for the intraday subset.
4. **Cumulative Performance Comparison (Daily):** Shows the cumulative returns of the strategy (based on aggregated daily returns) versus a daily buy-and-hold benchmark, on a log scale.

Python

```
# --- 5. Plotting ---
if not df_analysis.empty and len(df_analysis) > 5 and not
daily_strat_returns.empty:
    fig, axs = plt.subplots(4, 1, figsize=(15, 22), sharex=False)
    # ... (logic for plotting subset_df for intraday plots) ...

    # axs[0]: Price, Channel, Stops
    # axs[1]: ATR, Compression State
    # axs[2]: Strategy Position (Intraday)
    # axs[3]: Cumulative Performance (Daily)

    plt.tight_layout()
    plt.show()
```

3.8. Unique Features & Notes

- **Intraday Operation:** Specifically designed for high-frequency intraday data (e.g., 5-minute bars).
- **Range Compression Trigger:** Trades are only initiated if the market is assessed to be in a state of low volatility (range compression) prior to the breakout.
- **Bidirectional Breakouts:** Can trade both long and short breakouts from the channel.
- **Dynamic Trailing Stop:** Uses an ATR-based trailing stop-loss that adjusts with market volatility.
- `yfinance` **Intraday Limitations:** Users should be aware of `yfinance`'s limitations regarding the historical depth of intraday data (typically 60 days using start/end dates, hence the `period` parameter is used for fetching recent data).
- **Timezone Handling:** Includes logic to manage timezone localization for intraday timestamps, crucial for consistency.
- **Execution Assumptions:** Entry prices are estimated based on channel breakout levels or the open of the breakout bar, which simplifies real-world slippage.
- **Performance Aggregation:** Intraday bar returns are compounded daily for performance reporting, providing a more standard view of strategy returns.
- **Plotting Intraday Detail:** The plots attempt to show detailed intraday action for a subset of the data due to the high number of bars.

---

4. Volatility Momentum Strategy

4.1. Overview and Objective

Overview:

This strategy operates on the principle that accelerating volatility, when combined with a clear price trend, can signal trading opportunities. It calculates historical volatility and then the momentum of this volatility (i.e., how quickly volatility itself is changing). Trades are initiated in the direction of the prevailing price trend, but only when volatility is observed to be increasing.

Objective:

The objective is to capitalize on periods where an increase in volatility supports an existing price trend. The strategy aims to go long if the price is trending up and volatility is accelerating, and go short if the price is trending down and volatility is accelerating. Positions are managed with an ATR-based trailing stop-loss and are also exited if the volatility ceases to accelerate.

4.2. Key Indicators and Components

The strategy utilizes the following indicators calculated from daily price data:

- **Daily Returns (`daily_return_col`):** Standard percentage change in closing prices. Python

```
df[daily_return_col] = df['Close'].pct_change()
```

- **Historical Volatility (`volatility_col`):** Calculated as the rolling standard deviation of daily returns. Python

```
# Historical Volatility (std dev of daily returns)
df[volatility_col] =
df[daily_return_col].rolling(window=vol_window).std()
```

- **Volatility Momentum (`vol_momentum_col`):** The difference between current volatility and volatility N periods ago (σt–σt–N). Python

```
# Volatility Momentum (σt − σt−N)
df[vol_momentum_col] = df[volatility_col] −
df[volatility_col].shift(vol_momentum_window)
```

- **Price Trend SMA (`price_trend_sma_col`):** A Simple Moving Average of closing prices to determine the primary price trend. Python

```
# Price Trend SMA
df[price_trend_sma_col] =
df['Close'].rolling(window=price_trend_sma_window).mean()
```

- **Average True Range (ATR - `atr_col_name_sl`):** Used for calculating the trailing stop-loss. Python

```
# ATR for Stop Loss
df['H-L_sl'] = df['High'] - df['Low']
df['H-PC_sl'] = np.abs(df['High'] - df['Close'].shift(1))
df['L-PC_sl'] = np.abs(df['Low'] - df['Close'].shift(1))
df['TR_sl'] = df[['H-L_sl', 'H-PC_sl', 'L-PC_sl']].max(axis=1)
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

4.3. Script Parameters

Key parameters for the strategy are defined at the beginning of the script:

Python

```
# --- Parameters ---
ticker = "SOL-USD"          # Example ticker
start_date = "2021-01-01"
end_date = "2024-12-31"

# Volatility Calculation Parameters
vol_window = 30             # Rolling window for historical volatility

# Volatility Momentum Parameters
vol_momentum_window = 7     # Lookback period for volatility momentum (σt -
σt-N)

# Price Trend Parameters
price_trend_sma_window = 30 # SMA window for determining price trend

# ATR Trailing Stop Parameters
atr_window_sl = 14
atr_multiplier_sl = 1.0

# Trading days per year
TRADING_DAYS_PER_YEAR = 365 # Adjusted for crypto; 252 for stocks
```

4.4. Data Handling

- **Data Download:** Daily OHLCV data is downloaded using `yfinance`. User preferences for `auto_adjust=False` and `droplevel` for MultiIndex columns are applied. Python

```
# --- 1. Download Data ---
df_raw = yf.download(
    [ticker], # Pass as list
    start=start_date,
    end=end_date,
    auto_adjust=False, # Per user preference
    progress=False
)
if isinstance(df_raw.columns, pd.MultiIndex):
    df = df_raw.droplevel(level=1, axis=1) # Per user preference
else:
    df = df_raw
df = df[['Open', 'High', 'Low', 'Close', 'Volume']].copy()
```

4.5. Trading Logic (within the main backtesting loop)

The strategy iterates through each day in the `df_analysis` DataFrame.

4.5.1. Signal Generation Conditions

Signals are based on the previous day's data (`prev_idx`):

- **Volatility Accelerating:** True if the `prev_vol_momentum` is greater than 0. Python

```
is_vol_accelerating = pd.notna(prev_vol_momentum) and prev_vol_momentum
> 0
```

- **Price Trend:** Determined by comparing the `prev_close` to the `prev_price_trend_sma`. Python

```
is_price_trending_up = pd.notna(prev_price_trend_sma) and prev_close >
prev_price_trend_sma
is_price_trending_down = pd.notna(prev_price_trend_sma) and prev_close <
prev_price_trend_sma
```

4.5.2. Entry Conditions

A new position (long or short) is considered if volatility is accelerating, and the price trend aligns. Entries are made at `today_open`.

Python

```
# (Inside backtesting loop, after stop-loss and strategy exit checks)
            # 3. New Entry Signals / Holding Logic
            target_signal_position = 0 # Default to flat
            # ... (is_vol_accelerating, is_price_trending_up/down defined
above) ...

            if is_vol_accelerating:
                if is_price_trending_up:
                    target_signal_position = 1 # Signal to go Long
                elif is_price_trending_down:
                    target_signal_position = -1 # Signal to go Short

            # 3a. Change in position based on signal
            if target_signal_position != current_day_assumed_position:
                # ... (P&L for exiting old position if any) ...
                current_day_assumed_position = target_signal_position

                if current_day_assumed_position == 1: # Entering New Long
                    current_day_assumed_entry_price = today_open
                    # ... (set initial_ts and
current_day_assumed_trailing_stop) ...
                elif current_day_assumed_position == -1: # Entering New
Short
                    current_day_assumed_entry_price = today_open
                    # ... (set initial_ts and
current_day_assumed_trailing_stop) ...
                else: # Going/Staying Flat
                    current_day_assumed_entry_price = np.nan
                    current_day_assumed_trailing_stop = np.nan
                # ... (P&L calculation for entry/flip) ...
                action_taken_today = True
```

4.5.3. Exit Conditions

Positions can be exited under two main conditions:

1. **ATR Trailing Stop-Loss:** This is checked first. If the stop is hit, the position is closed.

   Python

```
# 1. Check ATR Stop Loss (highest priority if in position)
if current_day_assumed_position == 1 and
pd.notna(current_day_assumed_trailing_stop) # ...
    if today_low <= current_day_assumed_trailing_stop:
        exit_price_sl = min(today_open,
current_day_assumed_trailing_stop)
        # ... (P&L calculation, set position to 0) ...
        action_taken_today = True
elif current_day_assumed_position == -1 and
pd.notna(current_day_assumed_trailing_stop) # ...
    if today_high >= current_day_assumed_trailing_stop:
        exit_price_sl = max(today_open,
current_day_assumed_trailing_stop)
        # ... (P&L calculation, set position to 0) ...
        action_taken_today = True
```

The trailing stop is updated daily if the position is held:

Python

```
# (If holding position, in section 3b)
if current_day_assumed_position == 1:
    current_day_assumed_trailing_stop =
max(current_day_assumed_trailing_stop, today_close - atr_multiplier_sl *
today_atr_sl)
elif current_day_assumed_position == -1:
    current_day_assumed_trailing_stop =
min(current_day_assumed_trailing_stop, today_close + atr_multiplier_sl *
today_atr_sl)
```

2. **Volatility No Longer Accelerating (Strategy Exit Rule):** If an existing position is open and the previous day's volatility momentum (`prev_vol_momentum`) is no longer positive (i.e., volatility is not accelerating), the position is exited at `today_open` .

   Python

   ```
   # (After ATR stop check, if not stopped out)
   else: # Not stopped out by ATR
       # 2. Check Strategy Exit (Volatility no longer accelerating)
       if current_day_assumed_position != 0 and pd.notna(prev_vol_momentum)
   and prev_vol_momentum <= 0:
           # ... (P&L calculation for exit at today_open, set position to
   ```

```
0) ...
        action_taken_today = True
```

4.5.4. Position Sizing & P&L Calculation

- **Position Sizing:** Positions are binary (1 for long, -1 for short, 0 for flat).
- **P&L Calculation (`Strategy_Daily_Return`):** Profit and loss are calculated daily.
  - If stopped out: P&L from entry to stop price.
  - If strategy exit rule triggered: P&L from entry to `today_open` .
  - If position flipped (e.g., long to short): Combines P&L from exiting old position at `today_open` and P&L of new position from `today_open` to `today_close` .
  - If new position entered from flat: P&L from `today_open` to `today_close` .
  - If holding an existing position (and not exited): P&L from `prev_close` to `today_close` .

4.6. Performance Evaluation

Daily strategy returns are used to calculate standard performance metrics.

- **Cumulative Returns:** Python

```
# --- Post-Loop Calculations ---
df_analysis['Strategy_Daily_Return'].fillna(0, inplace=True)
df_analysis['Cumulative_Strategy_Return'] = (1 +
df_analysis['Strategy_Daily_Return']).cumprod()
df_analysis['Cumulative_Buy_Hold_Return'] = (1 +
df_analysis[daily_return_col].fillna(0)).cumprod()
# ... (Alignment for Buy & Hold cumulative return) ...
```

- **Performance Metrics Function (`calc_performance_metrics`):** This function calculates and prints cumulative return, annualized return, annualized volatility, and Sharpe ratio for both the strategy and a buy-and-hold benchmark. Python

```
# --- 4. Performance Metrics ---
def calc_performance_metrics(returns, name,
trading_days_per_year=TRADING_DAYS_PER_YEAR):
    # ... (Check for sufficient data) ...
    avg_daily_return = returns.mean()
    std_daily_return = returns.std()
    ann_ret = avg_daily_return * trading_days_per_year
    ann_vol = std_daily_return * np.sqrt(trading_days_per_year)
    sharpe_ratio = ann_ret / ann_vol if ann_vol > 0.00001 else np.nan
```

```
    cumulative_return_factor = (1 + returns).prod()
    # ... (Print metrics) ...
```

4.7. Plotting Results

The script generates a 5-panel plot for visual analysis of the strategy:

1. **Close Price, Trend SMA, and Trailing Stops:** Displays the asset's closing price, the price trend SMA, and active trailing stop levels.
2. **Historical Volatility:** Shows the calculated historical volatility over time.
3. **Volatility Momentum:** Plots the volatility momentum indicator and a zero line to easily identify accelerating/decelerating volatility.
4. **Strategy Position:** A step plot indicating Long (1), Short (-1), or Flat (0) positions.
5. **Cumulative Performance Comparison:** Log-scaled chart comparing the strategy's cumulative returns against a buy-and-hold benchmark.

Python

```
# --- 5. Plotting ---
if not df_analysis.empty and len(df_analysis) > 1 :
    fig, axs = plt.subplots(5, 1, figsize=(15, 25), sharex=True)
    # axs[0]: Close Price, Trend SMA, Trailing Stops
    # axs[1]: Historical Volatility
    # axs[2]: Volatility Momentum
    # axs[3]: Strategy Position
    # axs[4]: Cumulative Performance
    plt.tight_layout()
    plt.show()
```

4.8. Unique Features & Notes

- **Dual Condition Entry:** Requires both volatility acceleration and price trend alignment for initiating trades.
- **Specific Exit Rule:** In addition to ATR trailing stops, positions are exited if the primary condition of accelerating volatility is no longer met.
- **Daily Timeframe:** The strategy operates on daily data.
- **Parameterization:** The windows for volatility calculation, momentum, price trend, and ATR stops are all configurable.
- **Trailing Stop Mechanism:** Employs an ATR-based trailing stop for risk management once a position is opened.
- **NaN Handling:** The backtesting loop includes checks for NaN values in critical indicators to prevent errors and maintain position during data gaps.

5. Volatility Ratio Reversion with Trend Filter & ATR Trailing Stop Strategy

5.1. Overview and Objective

Overview:

This strategy is based on the concept of volatility mean reversion, specifically looking at the ratio of short-term historical volatility to long-term historical volatility. It hypothesizes that when this ratio reaches extreme levels, it's likely to revert. The strategy combines these volatility ratio signals with a trend filter to improve entry decisions.

Objective:

The primary objective is to:

- Enter **long** positions when short-term volatility is significantly lower than long-term volatility (ratio < `lower_threshold`), suggesting volatility might expand, but only if the price is in an uptrend.
- Enter **short** positions when short-term volatility is significantly higher than long-term volatility (ratio > `upper_threshold`), suggesting volatility might contract, but only if the price is in a downtrend. An ATR-based trailing stop-loss is used for risk management.

5.2. Key Indicators and Components

The strategy relies on several indicators calculated from daily data:

- **Daily Returns (`Daily_Return`):** The percentage change in closing prices. Python

```
df['Daily_Return'] = df['Close'].pct_change()
```

- **Short-Term Historical Volatility (`short_vol_col_name`):** Standard deviation of daily returns over a short window. Python

```
df[short_vol_col_name] =
df['Daily_Return'].rolling(window=short_vol_window,
min_periods=short_vol_window).std()
```

- **Long-Term Historical Volatility (`long_vol_col_name`):** Standard deviation of daily returns over a longer window. Python

```
df[long_vol_col_name] =
df['Daily_Return'].rolling(window=long_vol_window,
```

```
min_periods=long_vol_window).std()
```

- **Volatility Ratio (`ratio_col_name`):** The ratio of short-term volatility to long-term volatility. Python

```
df[ratio_col_name] = df[short_vol_col_name] / df[long_vol_col_name]
```

- **Trend Filter SMA (`trend_sma_col_name`):** A Simple Moving Average of closing prices used to determine the prevailing trend. Python

```
df[trend_sma_col_name] =
df['Close'].rolling(window=trend_sma_window).mean()
```

- **Average True Range (ATR - `atr_col_name_sl`):** Used for the trailing stop-loss mechanism. Python

```
df['H-L_sl'] = df['High'] - df['Low']
df['H-PC_sl'] = np.abs(df['High'] - df['Close'].shift(1))
df['L-PC_sl'] = np.abs(df['Low'] - df['Close'].shift(1))
df['TR_sl'] = df[['H-L_sl', 'H-PC_sl', 'L-PC_sl']].max(axis=1)
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

5.3. Script Parameters

The following parameters are configurable at the beginning of the script:

Python

```
# --- Parameters ---
ticker = "BTC-USD"          # Example ticker
start_date = "2021-01-01"
end_date = "2024-12-31"

# Volatility Ratio Parameters
short_vol_window = 7
long_vol_window = 30
upper_threshold = 1.2       # Short if R > upper_threshold (and trend
allows)
lower_threshold = 0.8       # Long if R < lower_threshold (and trend allows)

# Trend Filter Parameters
```

```
trend_sma_window = 50       # SMA window for trend determination

# ATR Trailing Stop Parameters
atr_window_sl = 14
atr_multiplier_sl = 1.0
```

(Note: `TRADING_DAYS_PER_YEAR` is also defined in the script but not listed here as it's used in the performance calculation function, which is standard across scripts).

5.4. Data Handling

- **Data Download:** Daily OHLCV data is fetched using `yfinance`. User preferences for `auto_adjust=False` and `droplevel` (for MultiIndex columns when a single ticker is provided as a list) are respected. Python

```
# --- 1. Download Data ---
df_raw = yf.download(
    [ticker],
    start=start_date,
    end=end_date,
    auto_adjust=False,
    progress=False
)
if isinstance(df_raw.columns, pd.MultiIndex):
    df = df_raw.droplevel(level=1, axis=1)
else:
    df = df_raw
```

5.5. Trading Logic (within the main backtesting loop)

The strategy logic iterates daily through the `df_analysis` DataFrame. Trades are based on the previous day's signals and executed at the current day's `Open`.

5.5.1. Signal Generation Conditions

Signals are determined based on the previous day's (`prev_idx`) Volatility Ratio (`R_prev`) and its relation to thresholds, combined with a trend filter.

- **Volatility Signal (`vol_signal`):** Python

```
# B. Determine Volatility Signal (if not stopped out)
vol_signal = 0
if pd.notna(R_prev):
```

```
    if R_prev < lower_threshold:
        vol_signal = 1  # Potential Long
    elif R_prev > upper_threshold:
        vol_signal = -1 # Potential Short
```

- **Trend Filter Application & Target Position (`current_target_position`):** Python

```
# C. Apply Trend Filter & Determine Target Position
current_target_position = 0
if pd.notna(prev_trend_sma): # Ensure trend SMA is available
    if vol_signal == 1 and prev_close > prev_trend_sma: # Long signal +
Uptrend
        current_target_position = 1
    elif vol_signal == -1 and prev_close < prev_trend_sma: # Short
signal + Downtrend
        current_target_position = -1
```

5.5.2. Entry Conditions

If the `current_target_position` is different from the `active_position` (and not due to a stop-loss on the current bar), a new position is considered. Entry occurs at `today_open`.

Python

```
# D. Handle Position Changes or Holding (if not stopped out)
if current_target_position != active_position:
    # ... (P&L for exiting old position if any) ...
    active_position = current_target_position # Assume new target position

    if active_position == 1: # Entering New Long
        current_entry_price = today_open
        # ... (pnl_entry_hold calculation) ...
        initial_ts = current_entry_price - atr_multiplier_sl * prev_atr_sl
        active_trailing_stop = max(initial_ts, today_close -
atr_multiplier_sl * today_atr_sl)
        entry_price = current_entry_price # Set entry price for this new
trade
    elif active_position == -1: # Entering New Short
        current_entry_price = today_open
        # ... (pnl_entry_hold calculation) ...
        initial_ts = current_entry_price + atr_multiplier_sl * prev_atr_sl
        active_trailing_stop = min(initial_ts, today_close +
```

```
atr_multiplier_sl * today_atr_sl)
        entry_price = current_entry_price # Set entry price for this new
trade
    else: # Going flat
        active_trailing_stop = np.nan
        entry_price = np.nan
    # ... (Combine P&L if flip occurred) ...
```

5.5.3. Exit Conditions

The primary exit mechanism is the ATR Trailing Stop-Loss.

- **ATR Trailing Stop-Loss Check (highest priority):** Python

```
# A. Check Stop Loss First (if in a position)
if active_position == 1 and pd.notna(active_trailing_stop):
    if today_low <= active_trailing_stop:
        exit_price = min(today_open, active_trailing_stop)
        pnl = (exit_price / entry_price) - 1
        active_position = 0
        entry_price = np.nan
        stop_triggered_today = True
elif active_position == -1 and pd.notna(active_trailing_stop):
    if today_high >= active_trailing_stop:
        exit_price = max(today_open, active_trailing_stop)
        pnl = -((exit_price / entry_price) - 1)
        active_position = 0
        entry_price = np.nan
        stop_triggered_today = True

if stop_triggered_today:
    active_trailing_stop = np.nan
```

- **Trailing Stop Adjustment (if holding a position and not stopped):** Python

```
# (If holding existing position and not stopped, within section D)
elif active_position != 0: # Holding existing position
    if active_position == 1:
        # ... (pnl calculation for holding) ...
        active_trailing_stop = max(active_trailing_stop, today_close -
atr_multiplier_sl * today_atr_sl)
    elif active_position == -1:
        # ... (pnl calculation for holding) ...
```

```
        active_trailing_stop = min(active_trailing_stop, today_close +
atr_multiplier_sl * today_atr_sl)
```

- **Exit due to Signal Change:** If `current_target_position` becomes 0 (e.g., trend filter no longer aligns or vol_signal becomes neutral) or flips to the opposite direction, the existing position is closed at `today_open`.

5.5.4. Position Sizing & P&L Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **P&L Calculation (`Strategy_Daily_Return`):** Calculated daily based on the outcome of the bar.
  - If stopped out: P&L from original `entry_price` to `exit_price` (stop).
  - If position changed due to new signal (entry/flip/exit to flat):
    - `pnl_exit`: Calculated from `entry_price` (or `prev_close` if `entry_price` was NaN for some reason) to `today_open` for the old position.
    - `pnl_entry_hold`: Calculated from `today_open` to `today_close` for the new position (if any).
    - Total `pnl` for the day combines these appropriately for flips.
  - If holding: P&L from `prev_close` to `today_close`.

5.6. Performance Evaluation

Standard performance metrics are computed using daily strategy returns.

- **Cumulative Returns:** Python

```
df_analysis['Strategy_Daily_Return'].fillna(0, inplace=True)
df_analysis['Cumulative_Strategy_Return'] = (1 +
df_analysis['Strategy_Daily_Return']).cumprod()
df_analysis['Cumulative_Buy_Hold_Return'] = (1 +
df_analysis['Daily_Return'].fillna(0)).cumprod()
# ... (Alignment for Buy & Hold cumulative return) ...
```

- **Performance Metrics Function (`calc_performance_metrics`):** Calculates and prints cumulative return, annualized return, annualized volatility, and Sharpe ratio. Python

```
# --- 4. Performance Metrics (Simple) ---
def calc_performance_metrics(returns, name, trading_days_per_year=365):
# Defaulting to 365
    # ... (calculations as in previous scripts) ...
```

```
    print(f"Cumulative Return: {cumulative_return_factor:.2f}x")
    # ... (other print statements) ...
```

5.7. Plotting Results

The script generates four plots to visualize the strategy:

1. **Close Price, Trend SMA, and Trailing Stops:** Shows price, the trend-defining SMA, and active trailing stop levels.
2. **Volatility Ratio with Thresholds:** Plots the calculated volatility ratio along with the defined upper and lower threshold lines.
3. **Strategy Position:** A step plot indicating Long (1), Short (-1), or Flat (0) positions over time.
4. **Cumulative Performance Comparison:** A log-scaled chart comparing the strategy's cumulative returns to a buy-and-hold benchmark.

Python

```
# --- 5. Plotting ---
if not df_analysis.empty and len(df_analysis) > 1:
    # Plot 1: Close Price, Trend SMA, and Trailing Stops
    plt.figure(figsize=(15, 6)) # ... (plotting code) ...
    plt.show()

    # Plot 2: Volatility Ratio with Thresholds
    plt.figure(figsize=(15, 6)) # ... (plotting code) ...
    plt.show()

    # Plot 3: Strategy Position
    plt.figure(figsize=(15, 4)) # ... (plotting code) ...
    plt.show()

    # Plot 4: Cumulative Performance Comparison
    plt.figure(figsize=(15, 6)) # ... (plotting code) ...
    plt.show()
```

5.8. Unique Features & Notes

- **Volatility Mean Reversion Focus:** The core signal is derived from the ratio of short-term to long-term volatility, expecting extremes in this ratio to revert.
- **Trend Confirmation:** Trades are only taken if the prevailing price trend (defined by an SMA) aligns with the direction implied by the volatility ratio signal.

- **Dual Thresholds:** Uses distinct upper and lower thresholds for the volatility ratio to signal potential short and long entries, respectively.
- **Daily Operation:** The strategy analyzes data and makes decisions on a daily basis.
- **ATR Trailing Stop:** Employs a standard ATR-based trailing stop for risk management of open positions.
- **NaN Handling:** The backtest loop includes checks for NaN values in critical indicators to ensure robustness.

---

6. Volatility-Clustering Reversion Strategy

6.1. Overview and Objective

Overview:

This strategy is built on the observation of volatility clustering, where periods of high volatility tend to be followed by more high volatility, and periods of low volatility by low volatility. However, after a sustained cluster of high-volatility days, this strategy anticipates a potential (temporary) return to calmer conditions. It identifies such clusters and then makes a contrarian bet on price mean-reversion, filtered by the overall market trend.

Objective:

The primary objective is to:

1. Detect a predefined number of consecutive "high-volatility" days.
2. After such a cluster, if the price moved down during the cluster, take a **long** position (betting on a reversion upwards), provided the broader market trend is up.
3. Conversely, if the price moved up during the cluster, take a **short** position (betting on a reversion downwards), provided the broader market trend is down. The strategy uses an ATR-based trailing stop-loss for risk management.

6.2. Key Indicators and Components

The strategy utilizes several indicators calculated from daily data:

- **Daily Returns (`daily_return_col`):** Standard percentage change in closing prices. Python

```
df[daily_return_col] = df['Close'].pct_change()
```

- **Annualized Historical Volatility (`hist_vol_col`):** Rolling standard deviation of daily returns, annualized. Python

```
df[hist_vol_col] = df[daily_return_col].rolling(window=vol_window).std()
* np.sqrt(TRADING_DAYS_PER_YEAR)
```

- **High-Volatility Day Detection (`is_high_vol_day_col`):** A day is flagged as high volatility if its `hist_vol_col` exceeds a dynamic threshold (mean of historical volatility plus a factor of its standard deviation over a `vol_stats_window`). Python

```
df[mean_hist_vol_col] =
df[hist_vol_col].rolling(window=vol_stats_window).mean()
df[std_hist_vol_col] =
df[hist_vol_col].rolling(window=vol_stats_window).std()
df[is_high_vol_day_col] = df[hist_vol_col] > (df[mean_hist_vol_col] +
vol_cluster_threshold_factor * df[std_hist_vol_col])
```

- **Consecutive High-Volatility Days (`consecutive_high_vol_col`):** Counts the number of consecutive days flagged as `Is_High_Vol_Day` . Python

```
# Create groups for consecutive True/False in Is_High_Vol_Day
df['High_Vol_Group_ID'] = (df[is_high_vol_day_col] !=
df[is_high_vol_day_col].shift(1)).cumsum()
# Calculate cumulative count within each group
df[consecutive_high_vol_col] =
df.groupby('High_Vol_Group_ID').cumcount() + 1
# Reset count to 0 for days that are not high-vol
df.loc[~df[is_high_vol_day_col], consecutive_high_vol_col] = 0
```

- **Trend Filter SMA (`trend_filter_sma_col`):** A Simple Moving Average of closing prices to determine the overall market trend. Python

```
df[trend_filter_sma_col] =
df['Close'].rolling(window=trend_filter_sma_window).mean()
```

- **Average True Range (ATR - `atr_col_name_sl`):** Used for the trailing stop-loss. Python

```
df['H-L_sl'] = df['High'] - df['Low']
df['H-PC_sl'] = np.abs(df['High'] - df['Close'].shift(1))
df['L-PC_sl'] = np.abs(df['Low'] - df['Close'].shift(1))
df['TR_sl'] = df[['H-L_sl', 'H-PC_sl', 'L-PC_sl']].max(axis=1)
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

6.3. Script Parameters

Configurable parameters are set at the beginning of the script:

Python

```
# --- Parameters ---
ticker = "EURUSD=X"
start_date = "2021-01-01"
end_date = "2024-12-31"

# Volatility Calculation
vol_window = 7              # Rolling window for historical volatility

# High-Volatility Day Detection
vol_stats_window = 30       # Window for rolling mean and std dev of
historical volatility
vol_cluster_threshold_factor = 1.0 # Hist_Vol > (Mean_Hist_Vol + N *
Std_Hist_Vol)

# Volatility Cluster
vol_cluster_days_trigger = 3 # Number of consecutive high-vol days to
trigger a signal

# Trend Filter Parameters
trend_filter_sma_window = 30 # SMA window for the overall trend filter

# ATR Trailing Stop Parameters
atr_window_sl = 14
atr_multiplier_sl = 2.0

TRADING_DAYS_PER_YEAR = 252 # Adjusted for crypto if detected
# ... (crypto check for TRADING_DAYS_PER_YEAR) ...
```

6.4. Data Handling

- **Data Download:** Daily OHLCV data is downloaded using `yfinance`. User preferences for `auto_adjust=False` and `droplevel` for MultiIndex columns are applied. Python

  ```
  # --- 1. Download Data ---
  df_raw = yf.download(
      [ticker], start=start_date, end=end_date,
      auto_adjust=False, progress=False
  ```

```
)
if df_raw.empty: raise SystemExit(f"No data for {ticker}.")
df = df_raw.droplevel(level=1, axis=1) if isinstance(df_raw.columns,
pd.MultiIndex) else df_raw
df = df[['Open', 'High', 'Low', 'Close', 'Volume']].copy()
```

6.5. Trading Logic (within the main backtesting loop)

The strategy logic is processed daily. Trades are based on the previous day's conditions and entered at the current day's `Open`.

6.5.1. Signal Generation Conditions

A trade signal is generated if the strategy is currently flat and specific conditions regarding volatility clusters and price movement are met.

- **Volatility Cluster Trigger:** The primary trigger is observing `vol_cluster_days_trigger` (e.g., 3) consecutive high-volatility days on the *previous day*. Python

```
# (Inside backtesting loop, if flat and not stopped out)
if current_day_assumed_position == 0: # Only consider new entries if
flat
    if prev_consecutive_high_vol == vol_cluster_days_trigger:
        # ... proceed to determine trade direction ...
```

- **Contrarian Price Movement Direction (`potential_trade_direction`):** The strategy then looks at the price movement *during* the detected N-day high-volatility cluster. The price at the end of the cluster (`price_at_cluster_end` , which is `prev_idx` 's close) is compared to the price on the day *before* the cluster started. Python

```
        idx_day_before_cluster_start_relative_to_i = i - 1 -
vol_cluster_days_trigger
        if idx_day_before_cluster_start_relative_to_i >= 0:
            day_before_cluster_starts_idx =
df_analysis.index[idx_day_before_cluster_start_relative_to_i]
            price_at_cluster_end = df_analysis.at[prev_idx, 'Close']
            price_before_cluster =
df_analysis.at[day_before_cluster_starts_idx, 'Close']

            potential_trade_direction = 0
            if pd.notna(price_before_cluster) and
pd.notna(price_at_cluster_end):
                if price_at_cluster_end < price_before_cluster:
```

```
potential_trade_direction = 1  # Bet Long
                elif price_at_cluster_end > price_before_cluster:
potential_trade_direction = -1 # Bet Short
```

- **Trend Filter Application (`trade_allowed`):** The `potential_trade_direction` is then filtered by the overall market trend (based on `prev_close` vs `prev_trend_filter_sma`). Python

```
            trade_allowed = False
            if potential_trade_direction == 1 and prev_close >
prev_trend_filter_sma: # Bet Long, Trend Up
                trade_allowed = True
            elif potential_trade_direction == -1 and prev_close <
prev_trend_filter_sma: # Bet Short, Trend Down
                trade_allowed = True
```

6.5.2. Entry Conditions

If `trade_allowed` is true and a `potential_trade_direction` is determined, a position is entered at `today_open` .

Python

```
                if trade_allowed and potential_trade_direction != 0:
                    current_day_assumed_position = potential_trade_direction
                    current_day_assumed_entry_price = today_open # Enter at
today's open

                    if current_day_assumed_position == 1: # Long
                        pnl_for_day = (today_close /
current_day_assumed_entry_price) - 1
                        initial_ts = current_day_assumed_entry_price -
atr_multiplier_sl * prev_atr_sl
                        current_day_assumed_trailing_stop = max(initial_ts,
today_close - atr_multiplier_sl * today_atr_sl)
                    else: # Short
                        pnl_for_day = -((today_close /
current_day_assumed_entry_price) - 1)
                        initial_ts = current_day_assumed_entry_price +
atr_multiplier_sl * prev_atr_sl
                        current_day_assumed_trailing_stop = min(initial_ts,
today_close + atr_multiplier_sl * today_atr_sl)
```

```
                    action_taken_this_step = True # An entry action was
taken
```

6.5.3. Exit Conditions

The primary exit mechanism is the ATR Trailing Stop-Loss. There is no explicit rule to exit if volatility calms down after entry; the trade is managed by the TSL.

- **ATR Trailing Stop-Loss Check (highest priority if in a position):** Python

```
# 1. Check ATR Stop Loss
if current_day_assumed_position == 1 and
pd.notna(current_day_assumed_trailing_stop) # ...
    if today_low <= current_day_assumed_trailing_stop:
        exit_price_sl = min(today_open,
current_day_assumed_trailing_stop)
        # ... (P&L calculation, set position to 0) ...
        action_taken_this_step = True
elif current_day_assumed_position == -1 and
pd.notna(current_day_assumed_trailing_stop) # ...
    if today_high >= current_day_assumed_trailing_stop:
        exit_price_sl = max(today_open,
current_day_assumed_trailing_stop)
        # ... (P&L calculation, set position to 0) ...
        action_taken_this_step = True

if action_taken_this_step: # If stopped out
    current_day_assumed_entry_price = np.nan
    current_day_assumed_trailing_stop = np.nan
```

- **Trailing Stop Adjustment (if holding a position and not stopped out):** Python

```
# (If holding a position and no stop/new entry on current bar)
if not action_taken_this_step and current_day_assumed_position != 0:
    if current_day_assumed_position == 1:
        # ... (P&L for holding) ...
        current_day_assumed_trailing_stop =
max(current_day_assumed_trailing_stop, today_close - atr_multiplier_sl *
today_atr_sl)
    elif current_day_assumed_position == -1:
        # ... (P&L for holding) ...
        current_day_assumed_trailing_stop =
```

```
min(current_day_assumed_trailing_stop, today_close + atr_multiplier_sl *
today_atr_sl)
```

6.5.4. Position Sizing & P&L Calculation

- **Position Sizing:** Positions are binary (1 for long, -1 for short, 0 for flat).
- **P&L Calculation (`Strategy_Daily_Return`):** Calculated daily.
  - On entry: P&L from `today_open` to `today_close`.
  - If stopped out: P&L from entry price to stop-loss exit price.
  - If holding: P&L from `prev_close` to `today_close`.

6.6. Performance Evaluation

Daily strategy returns are used for performance metric calculations.

- **Cumulative Returns & Alignment:** Python

```
# --- Post-Loop Calculations ---
df_analysis['Strategy_Daily_Return'].fillna(0, inplace=True)
df_analysis['Cumulative_Strategy_Return'] = (1 +
df_analysis['Strategy_Daily_Return']).cumprod()
# ... (Buy & Hold calculation and alignment) ...
```

- **Performance Metrics Function (`calc_performance_metrics`):** Calculates cumulative return, annualized return, annualized volatility, and Sharpe ratio. Python

```
# --- 4. Performance Metrics ---
def calc_performance_metrics(returns, name,
trading_days_per_year=TRADING_DAYS_PER_YEAR):
    # ... (standard calculations as in previous scripts) ...
    print(f"Cumulative Return: {cumulative_return_factor:.2f}x")
    # ... (other print statements) ...
```

6.7. Plotting Results

The script generates a 5-panel plot:

1. **Close Price, Trend Filter SMA, and Trailing Stops.**
2. **Historical Volatility and the dynamic High-Volatility Threshold.**
3. **Consecutive High-Volatility Day Count** with the trigger level highlighted.
4. **Strategy Position** (Long/Short/Flat).

5. **Cumulative Performance Comparison** (Strategy vs. Buy & Hold, log scale).

Python

```
# --- 5. Plotting ---
if not df_analysis.empty and len(df_analysis) > 5:
    fig, axs = plt.subplots(5, 1, figsize=(15, 25), sharex=True)
    # axs[0]: Close Price, Trend Filter, Stops
    # axs[1]: Historical Volatility & Threshold
    # axs[2]: Consecutive High-Vol Days & Trigger
    # axs[3]: Strategy Position
    # axs[4]: Cumulative Performance
    plt.tight_layout()
    plt.show()
```

6.8. Unique Features & Notes

- **Volatility Cluster Detection:** Identifies periods of sustained high volatility using a dynamic threshold.
- **Contrarian Reversion Post-Cluster:** Bets on a price reversion after the volatility cluster, but this "contrarian" element is with respect to the price movement *during* the cluster.
- **Trend Confirmation:** The contrarian reversion signal is filtered by the broader market trend, adding a layer of confirmation.
- **Dynamic Threshold:** The definition of a "high-volatility day" is adaptive, based on the rolling mean and standard deviation of historical volatility.
- **Daily Timeframe:** The strategy analyzes and trades on daily data.
- **ATR Trailing Stop:** Standard risk management for open positions.

---

7. Volatility-Oscillator Divergence Strategy

7.1. Overview and Objective

Overview:

This strategy identifies potential trading opportunities by looking for divergences between the price movement of an asset and the movement of a volatility oscillator. The volatility oscillator is constructed as the Rate of Change (ROC) of historical volatility. The core idea is that if price is trending in one direction, but the volatility oscillator is moving in the opposite direction (diverging), it can signal a confirmation or strengthening of the price trend.

Objective:

The objective is to:

- Enter a **long** position when the price is making higher highs (or generally trending up), the volatility oscillator is making lower highs (or generally trending down – a bullish divergence), and the overall long-term market trend is also upwards.
- Enter a **short** position when the price is making lower lows (or generally trending down), the volatility oscillator is making higher lows (or generally trending up – a bearish divergence), and the overall long-term market trend is also downwards. Trades are managed with an ATR-based trailing stop-loss.

7.2. Key Indicators and Components

The strategy relies on several indicators calculated from daily data:

- **Daily Returns (`daily_return_col`):** Standard percentage change in closing prices. Python

```
df[daily_return_col] = df['Close'].pct_change()
```

- **Historical Volatility (`volatility_col`):** Rolling standard deviation of daily returns. Python

```
df[volatility_col] =
df[daily_return_col].rolling(window=vol_window).std()
```

- **Volatility Oscillator (`vol_osc_col`):** The Rate of Change (ROC) of the historical volatility. Python

```
df[vol_osc_col] =
df[volatility_col].pct_change(periods=vol_osc_roc_period) * 100
```

- **Price Change (`price_change_col`):** The difference in closing price over a lookback period, used for divergence detection. Python

```
df[price_change_col] =
df['Close'].diff(periods=divergence_price_lookback)
```

- **Volatility Oscillator Change (`vol_osc_change_col`):** The difference in the volatility oscillator's value over a lookback period, used for divergence detection. Python

```
df[vol_osc_change_col] =
df[vol_osc_col].diff(periods=divergence_vol_osc_lookback)
```

- **Long-Term Trend SMA (`long_term_sma_col`):** A Simple Moving Average of closing prices to define the overall market trend. Python

```
df[long_term_sma_col] =
df['Close'].rolling(window=long_term_sma_window).mean()
```

- **Average True Range (ATR - `atr_col_name_sl`):** Used for calculating the trailing stop-loss. Python

```
df['H-L_sl'] = df['High'] - df['Low']
df['H-PC_sl'] = np.abs(df['High'] - df['Close'].shift(1))
df['L-PC_sl'] = np.abs(df['Low'] - df['Close'].shift(1))
df['TR_sl'] = df[['H-L_sl', 'H-PC_sl', 'L-PC_sl']].max(axis=1)
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

7.3. Script Parameters

Key parameters for the strategy are defined at the beginning of the script:

Python

```
# --- Parameters ---
ticker = "ETH-USD"          # Example: "BTC-USD", "AAPL", "GOOGL"
start_date = "2021-01-01"
end_date = "2024-12-31"

# Volatility Calculation
vol_window = 30             # Rolling window for historical volatility

# Volatility Oscillator (ROC of Volatility)
vol_osc_roc_period = 7      # Period for ROC calculation on volatility

# Divergence Detection Lookbacks
divergence_price_lookback = 7       # For defining "price up/down" (Close_t
- Close_t-N)
divergence_vol_osc_lookback = 7   # For defining "vol-osc up/down" (VolOsc_t
- VolOsc_t-N)
```

```
# Overall Trend Definition
long_term_sma_window = 30   # SMA window for determining overall long-term
trend

# ATR Trailing Stop Parameters
atr_window_sl = 14
atr_multiplier_sl = 1.0

# Trading days per year
TRADING_DAYS_PER_YEAR = 365 # Adjusted for crypto; 252 for stocks
```

7.4. Data Handling

- **Data Download:** Daily OHLCV data is downloaded using `yfinance`. User preferences for `auto_adjust=False` and `droplevel` for MultiIndex columns are applied. Python

```
# --- 1. Download Data ---
df_raw = yf.download(
    [ticker],
    start=start_date,
    end=end_date,
    auto_adjust=False,
    progress=False
)
if isinstance(df_raw.columns, pd.MultiIndex):
    df = df_raw.droplevel(level=1, axis=1)
else:
    df = df_raw
df = df[['Open', 'High', 'Low', 'Close', 'Volume']].copy()
```

7.5. Trading Logic (within the main backtesting loop)

The strategy iterates through each day in the `df_analysis` DataFrame. Signals are based on the previous day's data, and trades are executed at the current day's `Open`.

7.5.1. Signal Generation Conditions

Divergence conditions and the overall trend filter are checked based on the *previous day's* data (`prev_idx`):

- **Divergence and Trend Conditions:** Python

```
# Define conditions based on previous day's data
is_price_up          = prev_price_change > 0
is_vol_osc_down      = prev_vol_osc_change < 0
is_overall_uptrend   = prev_close > prev_long_term_sma

is_price_down        = prev_price_change < 0
is_vol_osc_up        = prev_vol_osc_change > 0
is_overall_downtrend = prev_close < prev_long_term_sma
```

- **Target Position Determination (`target_signal_position`):** Python

```
# Determine target position from signals
if is_price_up and is_vol_osc_down and is_overall_uptrend:
    target_signal_position = 1 # Long signal (Bullish divergence in
uptrend)
elif is_price_down and is_vol_osc_up and is_overall_downtrend:
    target_signal_position = -1 # Short signal (Bearish divergence in
downtrend)
```

7.5.2. Entry Conditions

If the `target_signal_position` (determined from the previous day's data) is different from the `current_day_assumed_position` (the position held at the start of the current day, before any stop-loss action), a new trade is considered. Entry occurs at `today_open`.

Python

```
# (Inside backtesting loop, after stop-loss check)
    # 2a. Change in position based on signal
    if target_signal_position != current_day_assumed_position:
        # ... (P&L for exiting old position if any) ...
        current_day_assumed_position = target_signal_position # Assume new
position

        if current_day_assumed_position == 1: # Entering New Long
            current_day_assumed_entry_price = today_open
            # ... (set initial_ts and current_day_assumed_trailing_stop) ...
        elif current_day_assumed_position == -1: # Entering New Short
            current_day_assumed_entry_price = today_open
            # ... (set initial_ts and current_day_assumed_trailing_stop) ...
        else: # Going/Staying Flat
            current_day_assumed_entry_price = np.nan
```

```
            current_day_assumed_trailing_stop = np.nan
        # ... (P&L calculation for entry/flip) ...
```

7.5.3. Exit Conditions

The primary mechanism for exiting trades is the ATR Trailing Stop-Loss.

- **ATR Trailing Stop-Loss Check (highest priority):** This is checked at the beginning of each day's logic for any active position. Python

```
# 1. Check ATR Stop Loss
if current_day_assumed_position == 1 and
pd.notna(current_day_assumed_trailing_stop) # ...
    if today_low <= current_day_assumed_trailing_stop:
        exit_price_sl = min(today_open,
current_day_assumed_trailing_stop)
        # ... (P&L calculation, set position to 0) ...
        action_taken_this_step = True
elif current_day_assumed_position == -1 and
pd.notna(current_day_assumed_trailing_stop) # ...
    if today_high >= current_day_assumed_trailing_stop:
        exit_price_sl = max(today_open,
current_day_assumed_trailing_stop)
        # ... (P&L calculation, set position to 0) ...
        action_taken_this_step = True

if action_taken_this_step: # If stopped out
    current_day_assumed_entry_price = np.nan
    current_day_assumed_trailing_stop = np.nan
```

- **Trailing Stop Adjustment (if holding a position and not stopped):** The stop is adjusted at the end of the day based on `today_close` and `today_atr_sl`. Python

```
# (If holding position and no signal change, in section 2b)
elif current_day_assumed_position != 0:
    if current_day_assumed_position == 1:
        current_day_assumed_trailing_stop =
max(current_day_assumed_trailing_stop, today_close - atr_multiplier_sl *
today_atr_sl)
    elif current_day_assumed_position == -1:
        current_day_assumed_trailing_stop =
```

```
min(current_day_assumed_trailing_stop, today_close + atr_multiplier_sl *
today_atr_sl)
```

- **Exit due to Signal Change:** If `target_signal_position` becomes 0 or flips to the opposite direction, the existing position is closed at `today_open`.

7.5.4. Position Sizing & P&L Calculation

- **Position Sizing:** Positions are binary (1 for long, -1 for short, 0 for flat).
- **P&L Calculation (`Strategy_Daily_Return`):** Profit and loss are calculated on a daily basis.
  - If stopped out: P&L from entry to stop price.
  - If position flipped or exited due to new signal: P&L from exiting old position at `today_open`, plus P&L of new position (if any) from `today_open` to `today_close`.
  - If new position entered from flat: P&L from `today_open` to `today_close`.
  - If holding an existing position (and not exited): P&L from `prev_close` to `today_close`.

7.6. Performance Evaluation

Daily strategy returns are used to calculate standard performance metrics.

- **Cumulative Returns & Alignment:** Python

```
# --- Post-Loop Calculations ---
df_analysis['Strategy_Daily_Return'].fillna(0, inplace=True)
df_analysis['Cumulative_Strategy_Return'] = (1 +
df_analysis['Strategy_Daily_Return']).cumprod()
# ... (Buy & Hold calculation and alignment) ...
```

- **Performance Metrics Function (`calc_performance_metrics`):** This function calculates and prints cumulative return, annualized return, annualized volatility, and Sharpe ratio. Python

```
# --- 4. Performance Metrics ---
def calc_performance_metrics(returns, name,
trading_days_per_year=TRADING_DAYS_PER_YEAR):
    # ... (standard calculations as in previous scripts) ...
    print(f"Cumulative Return: {cumulative_return_factor:.2f}x")
    # ... (other print statements) ...
```

7.7. Plotting Results

The script generates a 5-panel plot for visual analysis:

1. **Close Price, Long-Term SMA, and Trailing Stops.**
2. **Historical Volatility.**
3. **Volatility Oscillator (ROC of Volatility)** with a zero line.
4. **Strategy Position** (Long/Short/Flat).
5. **Cumulative Performance Comparison** (Strategy vs. Buy & Hold, log scale).

Python

```
# --- 5. Plotting ---
if not df_analysis.empty and len(df_analysis) > 1:
    fig, axs = plt.subplots(5, 1, figsize=(15, 25), sharex=True)
    # axs[0]: Price, Long-Term SMA, Trailing Stops
    # axs[1]: Historical Volatility
    # axs[2]: Volatility Oscillator
    # axs[3]: Strategy Position
    # axs[4]: Cumulative Performance
    plt.tight_layout()
    plt.show()
```

7.8. Unique Features & Notes

- **Divergence Trading:** Leverages divergences between price momentum and volatility oscillator momentum as the primary signal.
- **Trend Confirmation:** Divergence signals are only traded if they align with the broader market trend defined by a long-term SMA. This aims to filter out divergences that might be false signals against the dominant trend.
- **Volatility ROC:** Uses the Rate of Change of historical volatility as the oscillator, providing a measure of the speed at which volatility is changing.
- **Daily Timeframe:** The strategy is designed for daily data.
- **ATR Trailing Stop:** Standard risk management for open positions.
- **Parameter Sensitivity:** The lookback periods for volatility, the oscillator, divergence detection, and the trend SMA are key parameters that can influence performance.

---

8. Wavelet-Decomposed Volatility Bands Strategy

8.1. Overview and Objective

Overview:

This strategy employs wavelet decomposition, a signal processing technique, to analyze the price series. It decomposes the price into different frequency components. By reconstructing a signal using only selected high-frequency detail coefficients, the strategy aims to isolate the more rapid oscillations or "noise" component of the price action. Volatility bands are then constructed around a moving average of the price, with their width determined by this reconstructed high-frequency signal.

Objective:

The primary objective is to trade breakouts of these wavelet-derived volatility bands. The rationale is that the reconstructed high-frequency signal provides a dynamic measure of recent volatility, and breaks beyond bands scaled by this measure can indicate significant price moves. The strategy enters long on an upward breakout and short on a downward breakout, with positions managed by an ATR-based trailing stop-loss.

8.2. Key Concepts: Wavelet Decomposition

Wavelet decomposition breaks down a time series (in this case, closing prices) into different frequency components. This is done by applying a chosen wavelet function (e.g., 'db4' - Daubechies 4) at multiple levels.

- **Approximation Coefficients (cA):** Represent the low-frequency, smoother part of the signal at each decomposition level.
- **Detail Coefficients (cD):** Represent the high-frequency parts of the signal. `cD1` captures the highest frequencies, `cD2` the next highest, and so on.

This strategy focuses on reconstructing a signal using only some of the highest frequency detail coefficients (e.g., `cD1` and/or `cD2`) to capture the "volatility" or "noise" aspect of the price.

8.3. Key Indicators and Components

The strategy relies on the following calculated series from daily data:

- Wavelet Decomposition & High-Frequency Reconstruction (hf_recon_col_name): The closing price series is decomposed using pywt.wavedec. Then, a new signal is reconstructed using only the selected high-frequency detail coefficients specified in wavelet_recon_hf_levels. Python

```
# Step 2.1: Wavelet Decomposition and High-Frequency Reconstruction
price_series = df['Close'].dropna()
# ... (data length check) ...
coeffs = pywt.wavedec(price_series, wavelet_type,
level=wavelet_decomp_level)
```

```
# Create a zeroed-out coefficient list structure for reconstruction
coeffs_hf_only_template = []
for c_arr in coeffs:
    coeffs_hf_only_template.append(np.zeros_like(c_arr))

# Populate with selected high-frequency detail coefficients
# coeffs list is [cA_n, cD_n, cD_n-1, ..., cD_1]
# cD_k corresponds to coeffs[-(k)]
num_coeffs_arrays = len(coeffs)
for hf_level_idx in wavelet_recon_hf_levels: # 1-based for cD1, cD2
    coeff_array_index = -hf_level_idx
    if 0 < hf_level_idx < num_coeffs_arrays :
        coeffs_hf_only_template[coeff_array_index] =
coeffs[coeff_array_index]
    # ... (warning if index out of bounds) ...

hf_reconstructed_signal_raw = pywt.waverec(coeffs_hf_only_template,
wavelet_type)
# ... (Alignment logic for hf_reconstructed_aligned with
price_series.index) ...
df.loc[price_series.index, hf_recon_col_name] = hf_reconstructed_aligned
df[hf_recon_col_name] = df[hf_recon_col_name].fillna(0)
```

- **Band Center Line (`center_line_col_name`):** A simple moving average of the closing prices. Python

```
# Step 2.2: Calculate Bands
df[center_line_col_name] =
df['Close'].rolling(window=band_center_ma_window).mean()
```

- Wavelet Volatility Bands (upper_band_col_name, lower_band_col_name): The bands are created by adding/subtracting a multiple of the absolute value of the reconstructed high-frequency signal from the center line. Python

```
df[upper_band_col_name] = df[center_line_col_name] + band_hf_multiplier
* np.abs(df[hf_recon_col_name])
df[lower_band_col_name] = df[center_line_col_name] - band_hf_multiplier
* np.abs(df[hf_recon_col_name])
```

- **Average True Range (ATR - `atr_col_name_sl`):** Used for the trailing stop-loss. Python

```
# Step 2.3: ATR for Stop Loss
df['H-L_sl'] = df['High'] - df['Low']
# ... (TR calculation) ...
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

8.4. Script Parameters

Key parameters are defined at the beginning of the script:

Python

```
# --- Parameters ---
ticker = "BTC-USD"
start_date = "2021-01-01"
end_date = "2024-12-31"

# Wavelet Parameters
wavelet_type = 'db4'        # Example: 'db1'-'db20', 'sym2'-'sym20', etc.
wavelet_decomp_level = 3    # Level of decomposition
wavelet_recon_hf_levels = [1] # Detail coefficient levels to reconstruct (1-
indexed for cD1)

# Volatility Band Parameters
band_center_ma_window = 7   # Moving average window for the band's center
band_hf_multiplier = 1.5    # Multiplier for the abs reconstructed HF signal
for band width

# ATR Trailing Stop Parameters
atr_window_sl = 14
atr_multiplier_sl = 1.0
```

8.5. Data Handling

- **Data Download:** Daily OHLCV data is downloaded via `yfinance`, with user preferences for `auto_adjust=False` and `droplevel` applied. Python

```
# --- 1. Download Data ---
df_raw = yf.download(
    [ticker],
    start=start_date,
    end=end_date,
    auto_adjust=False,
```

```
    progress=False
)
# ... (droplevel and column selection) ...
```

- **Data Length Check for Wavelets:** A basic check is performed to ensure there's enough data for the specified wavelet type and decomposition level. Python

```
if len(price_series) < pywt.Wavelet(wavelet_type).dec_len *
(2**wavelet_decomp_level):
    raise SystemExit(...)
```

8.6. Trading Logic (within the main backtesting loop)

The strategy processes data daily. Signals are derived from the previous day's band levels, and trades are executed at the current day's `Open` or the breakout level.

8.6.1. Signal Generation Conditions (Breakouts)

Breakouts are detected if the current day's `Open` price crosses beyond the *previous day's* upper or lower wavelet band.

- **Upward Breakout (Potential Long):** `today_open > prev_upper_band` when `prev_close <= prev_upper_band`.
- **Downward Breakout (Potential Short):** `today_open < prev_lower_band` when `prev_close >= prev_lower_band`.

8.6.2. Entry Conditions

If a breakout occurs and the strategy is either flat or in an opposing position, a new position is entered.

Python

```
# (Inside backtesting loop, after stop-loss check)
    # B. Check for New Entry Signals (Breakouts)
    target_position_based_on_signal = active_position # Assume no change

    if prev_close <= prev_upper_band and today_open > prev_upper_band: #
Upward breakout
        if active_position != 1 : # If flat or short, consider long
            target_position_based_on_signal = 1
    elif prev_close >= prev_lower_band and today_open < prev_lower_band: #
Downward breakout
```

```
        if active_position != -1: # If flat or long, consider short
            target_position_based_on_signal = -1

    # Handle Position Changes
    if target_position_based_on_signal != active_position:
        # ... (P&L for exiting old position if any) ...
        active_position = target_position_based_on_signal

        if active_position == 1: # Entering new Long
            current_entry_price = max(today_open, prev_upper_band) # Enter
at open or breakout point
            # ... (P&L calculation, set initial_ts and active_trailing_stop)
...
            entry_price = current_entry_price
        elif active_position == -1: # Entering new Short
            current_entry_price = min(today_open, prev_lower_band) # Enter
at open or breakout point
            # ... (P&L calculation, set initial_ts and active_trailing_stop)
...
            entry_price = current_entry_price
        # ... (Combine P&L if flip occurred) ...
```

The entry price (`current_entry_price`) is taken as the breakout level (`prev_upper_band` or `prev_lower_band`) or `today_open` if the open gapped beyond the band.

8.6.3. Exit Conditions

The primary exit is via an ATR Trailing Stop-Loss.

- **ATR Trailing Stop-Loss Check (highest priority):** Python

```
# A. Check Stop Loss
if active_position == 1 and pd.notna(active_trailing_stop) and
pd.notna(entry_price):
    if today_low <= active_trailing_stop:
        exit_price = min(today_open, active_trailing_stop)
        # ... (P&L calculation, set position to 0) ...
        stop_triggered_today = True
elif active_position == -1 and pd.notna(active_trailing_stop) and
pd.notna(entry_price):
    if today_high >= active_trailing_stop:
        exit_price = max(today_open, active_trailing_stop)
        # ... (P&L calculation, set position to 0) ...
        stop_triggered_today = True
```

```
if stop_triggered_today:
    active_trailing_stop = np.nan
    entry_price = np.nan
```

- **Trailing Stop Adjustment (if holding and not stopped):** The stop is updated daily based on `today_close` and `today_atr_sl`. Python

```
# (If holding existing position and not stopped)
elif active_position != 0 and pd.notna(active_trailing_stop):
    if active_position == 1:
        active_trailing_stop = max(active_trailing_stop, today_close -
atr_multiplier_sl * today_atr_sl)
    elif active_position == -1:
        active_trailing_stop = min(active_trailing_stop, today_close +
atr_multiplier_sl * today_atr_sl)
```

- **Exit due to No New Signal:** If `target_position_based_on_signal` becomes 0 (e.g., price re-enters bands without a new breakout signal in the opposite direction), any existing position is closed at `today_open`.

8.6.4. Position Sizing & P&L Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **P&L Calculation (`Strategy_Daily_Return`):** Calculated daily based on the day's outcome.
  - If stopped out: P&L from entry to stop price.
  - If position changed (entry/flip/exit to flat): Combines P&L from exiting the old position at `today_open` and P&L of the new position (if any) from entry to `today_close`.
  - If holding an existing position: P&L from `prev_close` to `today_close`.

8.7. Performance Evaluation

Daily strategy returns are used to calculate standard performance metrics.

- **Cumulative Returns & Alignment:** Python

```
# --- Post-Loop Calculations ---
df_analysis['Strategy_Daily_Return'].fillna(0, inplace=True)
df_analysis['Cumulative_Strategy_Return'] = (1 +
```

```
df_analysis['Strategy_Daily_Return']).cumprod()
# ... (Buy & Hold calculation and alignment) ...
```

- **Performance Metrics Function (`calc_performance_metrics`):** Calculates and prints cumulative return, annualized return, annualized volatility, and Sharpe ratio. Python

```
# --- 4. Performance Metrics (Simple) ---
def calc_performance_metrics(returns, name, trading_days_per_year=365):
# Defaulting to 365
    # ... (standard calculations as in previous scripts) ...
    print(f"Cumulative Return: {cumulative_return_factor:.2f}x")
    # ... (other print statements) ...
```

8.8. Plotting Results

The script generates a 4-panel plot:

1. **Price, Wavelet Bands, Center Line, and Trailing Stops.**
2. **Reconstructed High-Frequency Signal:** The wavelet-derived HF component used for band width.
3. **Strategy Position** (Long/Short/Flat).
4. **Cumulative Performance Comparison** (Strategy vs. Buy & Hold, log scale).

Python

```
# --- 5. Plotting ---
if not df_analysis.empty and len(df_analysis) > 1:
    fig, axs = plt.subplots(4, 1, figsize=(15, 22), sharex=True)
    # axs[0]: Price, Wavelet Bands, Center Line, Stops
    # axs[1]: Reconstructed HF Signal
    # axs[2]: Strategy Position
    # axs[3]: Cumulative Performance
    plt.tight_layout()
    plt.show()
```

8.9. Unique Features & Notes

- **Wavelet-Based Volatility:** Uses wavelet decomposition to isolate high-frequency components as a basis for volatility band construction, offering a different perspective than traditional ATR or standard deviation bands.
- **Adaptive Band Width:** The width of the bands dynamically changes based on the magnitude of the reconstructed high-frequency signal.

- **Breakout Strategy:** Trades breakouts of these dynamically adjusting bands.
- **Parameter Choices:** The choice of `wavelet_type`, `wavelet_decomp_level`, and `wavelet_recon_hf_levels` significantly impacts the nature of the reconstructed HF signal and thus the bands. These require careful consideration and possibly experimentation.
- `PyWavelets` **Dependency:** Requires the `PyWavelets` library.
- **Data Length:** Wavelet decomposition, especially at higher levels, requires a sufficient amount of data.
- **Daily Timeframe:** Operates on daily data.