In [1]:
```
import warnings
import numpy as np
import pandas as pd
from pathlib import Path
import os
import vectorbt as vbt
import io
import sys
from contextlib import redirect_stdout
from datetime import datetime
import math
import plotly.graph_objects as go
from plotly.subplots import make_subplots

import warnings
warnings.filterwarnings('ignore')

# Add project root to path
project_root = Path().absolute().parent
sys.path.append(str(project_root))

# Import utils with different aliases
from src.utils import csv_exporter as csv_utils
from src.utils import validation as val_utils
from src.utils import transformations as trans_utils
from src.utils import data_merger as merge_utils
from src.utils import config_validator as config_utils
from src.utils import metrics as metric_utils
from src.core.bloomberg_fetcher import fetch_bloomberg_data
from src.utils.transformations import get_ohlc
```

In [2]:
```
# Getting all the data
mapping = {
    ('I05510CA Index', 'INDEX_OAS_TSY_BP'): 'cad_oas',
    ('LF98TRUU Index', 'INDEX_OAS_TSY_BP'): 'us_hy_oas',
    ('LUACTRUU Index', 'INDEX_OAS_TSY_BP'): 'us_ig_oas',
    ('SPTSX Index', 'PX_LAST'): 'tsx',
    ('VIX Index', 'PX_LAST'): 'vix',
    ('USYC3M30 Index', 'PX_LAST'): 'us_3m_10y',
    ('BCMPUSGR Index', 'PX_LAST'): 'us_growth_surprises',
    ('BCMPUSIF Index', 'PX_LAST'): 'us_inflation_surprises',
    ('LEI YOY  Index', 'PX_LAST'): 'us_lei_yoy',
    ('.HARDATA G Index', 'PX_LAST'): 'us_hard_data_surprises',
    ('CGERGLOB Index', 'PX_LAST'): 'us_equity_revisions',
    ('.ECONREGI G Index', 'PX_LAST'): 'us_economic_regime',

}

# Calculate dates
end_date = datetime.now().strftime('%Y-%m-%d')
start_date ='2002-01-01'

# Fetch the data
df = fetch_bloomberg_data(
```

```
    mapping=mapping,
    start_date=start_date,
    end_date=end_date,
    periodicity='D',
    align_start=True
).dropna()

# Getting all the er_ytd data
mapping1 = {
    ('I05510CA Index', 'INDEX_EXCESS_RETURN_YTD'): 'cad_ig_er',
    ('LF98TRUU Index', 'INDEX_EXCESS_RETURN_YTD'): 'us_hy_er',
    ('LUACTRUU Index', 'INDEX_EXCESS_RETURN_YTD'): 'us_ig_er',
}

# Fetch the er_ytd_data
df1 = fetch_bloomberg_data(
    mapping=mapping1,
    start_date=start_date,
    end_date=end_date,
    periodicity='D',
    align_start=True
).dropna()

# Conver er_ytd data to an index
df2= trans_utils.convert_er_ytd_to_index(df1[['cad_ig_er','us_hy_er','us_ig_er']])
final_df=merge_utils.merge_dfs(df, df2, fill='ffill', start_date_align='yes')

# Handle bad data point for cad_oas on Nov 15 2005
bad_date = '2005-11-15'
if bad_date in final_df.index:
    final_df.loc[bad_date, 'cad_oas'] = final_df.loc[final_df.index < bad_date, 'ca


final_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5861 entries, 2002-10-31 to 2025-01-01
Data columns (total 15 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   cad_oas                 5861 non-null   float64
 1   us_hy_oas               5861 non-null   float64
 2   us_ig_oas               5861 non-null   float64
 3   tsx                     5861 non-null   float64
 4   vix                     5861 non-null   float64
 5   us_3m_10y               5861 non-null   float64
 6   us_growth_surprises     5861 non-null   float64
 7   us_inflation_surprises  5861 non-null   float64
 8   us_lei_yoy              5861 non-null   float64
 9   us_hard_data_surprises  5861 non-null   float64
 10  us_equity_revisions     5861 non-null   float64
 11  us_economic_regime      5861 non-null   float64
 12  cad_ig_er_index         5861 non-null   float64
 13  us_hy_er_index          5861 non-null   float64
 14  us_ig_er_index          5861 non-null   float64
dtypes: float64(15)
memory usage: 861.7 KB
```

In [3]:

```python
# Viz to make sure all the data looks ok
def create_spread_plots(df):
    # Calculate number of rows and columns needed based on number of series
    n_series = len(df.columns)
    n_rows = math.ceil(n_series / 3)  # Calculate required rows
    n_cols = min(3, n_series)  # Use 3 columns or less if fewer series

    # Adjust vertical spacing based on number of rows
    vertical_spacing = min(0.08, 1.0 / (n_rows + 1))  # Dynamic spacing

    # Create subplot grid
    fig = make_subplots(
        rows=n_rows,
        cols=n_cols,
        subplot_titles=df.columns,
        vertical_spacing=vertical_spacing,
        horizontal_spacing=0.05
    )

    # Add each series to a subplot
    for idx, column in enumerate(df.columns):
        row = (idx // n_cols) + 1
        col = (idx % n_cols) + 1

        fig.add_trace(
            go.Scatter(
                x=df.index,
                y=df[column],
                name=column,
                line=dict(width=1),
                showlegend=False,
                hovertemplate=
                "<b>%{x}</b><br>" +
```

```
                "Value: %{y:.2f}<br>" +
                "<extra></extra>"
            ),
            row=row,
            col=col
        )

        # Update axes labels
        fig.update_xaxes(
            title_text="Date",
            row=row,
            col=col,
            showgrid=True,
            gridcolor='rgba(128, 128, 128, 0.2)',
            tickangle=45,
            tickformat='%Y-%m-%d'
        )
        fig.update_yaxes(
            title_text="Spread",
            row=row,
            col=col,
            showgrid=True,
            gridcolor='rgba(128, 128, 128, 0.2)'
        )

    # Update layout for dark theme and responsiveness
    fig.update_layout(
        template='plotly_dark',
        showlegend=False,
        height=250 * n_rows,  # Adjusted height per row
        title={
            'text': 'Spread Series Over Time',
            'y':0.98,
            'x':0.5,
            'xanchor': 'center',
            'yanchor': 'top'
        },
        paper_bgcolor='rgb(30, 30, 30)',
        plot_bgcolor='rgb(30, 30, 30)',
        margin=dict(t=80, l=50, r=50, b=50),
        font=dict(
            family="Arial",
            size=10,
            color="white"
        )
    )

    # Make it responsive
    fig.update_layout(
        autosize=True,
    )

    # Show the plot
    fig.show(config={
        'responsive': True,
        'displayModeBar': True,
```

```
        'scrollZoom': True,
        'modeBarButtonsToAdd': ['drawline', 'drawopenpath', 'eraseshape']  # Add dr
    })

# Create the plots
create_spread_plots(final_df)
```

In [7]:

```
import sys
import os
import io
from contextlib import redirect_stdout
import pandas as pd
import numpy as np
from datetime import datetime
from pathlib import Path
import vectorbt as vbt
from scipy import signal, stats
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from abc import ABC, abstractmethod
from typing import Dict, Tuple, List

import warnings
warnings.filterwarnings('ignore')

# Base Strategy Class
class Strategy(ABC):
    """Base class for all trading strategies"""

    def __init__(self, df: pd.DataFrame, target_col: str = 'cad_ig_er_ytd_index'):
        self.df = df.copy()
        self.target_col = target_col
        self.name = self.__class__.__name__

    @abstractmethod
    def generate_signals(self) -> pd.Series:
        """Generate trading signals"""
        pass

# Strategy Implementations
class BuyAndHoldStrategy(Strategy):
    """Buy and Hold Strategy - Always invested"""

    def generate_signals(self) -> pd.Series:
        """Generate constant True signals for buy and hold"""
        return pd.Series(True, index=self.df.index)

class VolatilityRegimeStrategy(Strategy):
    """Volatility Regime Strategy"""

    def __init__(self, df: pd.DataFrame,
                 vol_window: int = 30,
                 correlation_window: int = 90,
                 regime_window: int = 252,
                 vol_threshold: float = 1.2):
```

```
        super().__init__(df)
        self.vol_window = vol_window
        self.correlation_window = correlation_window
        self.regime_window = regime_window
        self.vol_threshold = vol_threshold

    def _calculate_vol_surface_score(self) -> pd.Series:
        implied_vol = self.df['vix']
        realized_vols = pd.DataFrame(index=self.df.index)
        realized_vols[f'vol_{self.vol_window}'] = self.df[self.target_col].pct_chan
        vol_premium = implied_vol - realized_vols.mean(axis=1)
        vol_premium_zscore = (vol_premium - vol_premium.rolling(252).mean()) / vol_
        return -vol_premium_zscore

    def _calculate_correlation_score(self) -> pd.Series:
        target_returns = self.df[self.target_col].pct_change()
        assets = ['cad_oas', 'us_hy_oas', 'us_ig_oas']
        asset_returns = self.df[assets].pct_change()
        correlations = pd.DataFrame(index=self.df.index)
        for asset in assets:
            correlations[asset] = target_returns.rolling(self.correlation_window).c
        avg_correlation = correlations.mean(axis=1)
        correlation_zscore = (avg_correlation - avg_correlation.rolling(252).mean()
        return -correlation_zscore

    def _calculate_vol_regime(self) -> pd.Series:
        assets = ['cad_oas', 'us_hy_oas', 'us_ig_oas']
        vol_indicators = pd.DataFrame(index=self.df.index)
        for asset in assets:
            vol = self.df[asset].pct_change().rolling(20).std() * np.sqrt(252)
            vol_indicators[f'{asset}_vol'] = (vol < vol.rolling(252).mean())
        vol_indicators['vix_regime'] = self.df['vix'] < self.df['vix'].rolling(252)
        low_vol_regime = vol_indicators.mean(axis=1) > 0.5
        return low_vol_regime

    def generate_signals(self) -> pd.Series:
        vol_surface_score = self._calculate_vol_surface_score()
        correlation_score = self._calculate_correlation_score()
        vol_regime = self._calculate_vol_regime()
        returns = self.df[self.target_col].pct_change()
        trend = returns.rolling(60).mean() / returns.rolling(60).std()
        trend_strength = trend.abs()

        print("\nVolatility Regime Strategy Analysis:")
        print("==================================")
        low_vol_days = vol_regime.sum()
        print(f"Low Volatility Regime: {low_vol_days} days ({low_vol_days/len(vol_r
        print(f"Average Trend Strength: {trend_strength.mean():.2f}")
        print(f"Average Correlation Score: {correlation_score.mean():.2f}")

        signals = (
            vol_regime &
            (vol_surface_score > 0) &
            (correlation_score > -0.3) &
            (trend_strength > 0.1)
        )
```

```
        signals = signals.rolling(5).mean() > 0.6
        return signals

class AdaptiveTrendStrategy(Strategy):
    """Adaptive Trend Strategy"""

    def __init__(self, df: pd.DataFrame,
                 cycle_lookbacks: list = [10, 20, 40],
                 efficiency_window: int = 10,
                 min_trend_strength: float = 0.4):
        super().__init__(df)
        self.cycle_lookbacks = cycle_lookbacks
        self.efficiency_window = efficiency_window
        self.min_trend_strength = min_trend_strength

    def _decompose_series(self, series: pd.Series, window: int) -> tuple:
        normalized = (series - series.mean()) / series.std()
        nyq = 0.5 * 1
        cutoff = 1 / window
        order = 2
        b, a = signal.butter(order, cutoff/nyq, btype='low')
        trend = pd.Series(signal.filtfilt(b, a, normalized), index=series.index)
        cycle = normalized - trend
        return trend, cycle

    def _calculate_trend_strength(self, series: pd.Series, window: int) -> pd.Serie
        trend_strength = pd.Series(index=series.index)
        for i in range(window, len(series)):
            y = series.iloc[i-window:i]
            X = np.arange(window).reshape(-1, 1)
            reg = LinearRegression().fit(X, y)
            trend_strength.iloc[i] = reg.score(X, y)
        return trend_strength.fillna(0)

    def _calculate_cycle_score(self, cycle: pd.Series) -> pd.Series:
        cycle_zscore = (cycle - cycle.rolling(252).mean()) / cycle.rolling(252).std
        cycle_score = -cycle_zscore
        return cycle_score

    def _calculate_adaptive_lookback(self) -> pd.Series:
        vol = self.df[self.target_col].pct_change().rolling(20).std() * np.sqrt(252
        vol_ratio = vol / vol.rolling(252).mean()
        base_lookback = np.mean(self.cycle_lookbacks)
        lookbacks = pd.Series(base_lookback, index=self.df.index)
        adjusted_lookbacks = lookbacks * vol_ratio.fillna(1)
        return adjusted_lookbacks.clip(min(self.cycle_lookbacks), max(self.cycle_lo

    def _calculate_market_efficiency_ratio(self) -> pd.Series:
        price = self.df[self.target_col]
        dir_move = abs(price - price.shift(self.efficiency_window))
        total_move = pd.Series(0, index=price.index)
        for i in range(1, self.efficiency_window + 1):
            total_move += abs(price - price.shift(i))
        efficiency_ratio = dir_move / total_move
        return efficiency_ratio
```

```
    def generate_signals(self) -> pd.Series:
        signals = pd.Series(False, index=self.df.index)
        lookbacks = self._calculate_adaptive_lookback()
        avg_lookback = int(lookbacks.mean())
        trend, cycle = self._decompose_series(self.df[self.target_col], avg_lookbac
        trend_strength = self._calculate_trend_strength(self.df[self.target_col], a
        cycle_score = self._calculate_cycle_score(cycle)
        efficiency_ratio = self._calculate_market_efficiency_ratio()

        print("\nAdaptive Trend Strategy Analysis:")
        print("================================")
        print(f"Average Trend Strength: {trend_strength.mean():.2f}")
        print(f"Average Efficiency Ratio: {efficiency_ratio.mean():.2f}")
        print(f"Average Lookback Period: {avg_lookback} days")

        trending_market = trend_strength > self.min_trend_strength
        efficient_market = efficiency_ratio > 0.3
        trend_signals = trending_market & (trend.diff() > 0)
        reversion_signals = (~trending_market) & (cycle_score > 0.5)
        signals = trend_signals | reversion_signals
        return signals.fillna(False)

# Backtest Configuration
class BacktestConfig:
    def __init__(self,
                 start_date=None,
                 end_date=None,
                 rebalance_freq='1D',  # '1D' for daily, 'M' for monthly
                 initial_capital=100,
                 size=1.0,
                 size_type='percent'):
        self.start_date = pd.to_datetime(start_date) if start_date else None
        self.end_date = pd.to_datetime(end_date) if end_date else None
        self.rebalance_freq = rebalance_freq  # Can use 'M' directly for resampling
        self.initial_capital = initial_capital
        self.size = size
        self.size_type = size_type

    @classmethod
    def DAILY(cls):
        return cls(rebalance_freq='1D')

    @classmethod
    def MONTHLY(cls):
        return cls(rebalance_freq='M')

def load_data(config: BacktestConfig) -> pd.DataFrame:
    data_path = os.path.join(os.getcwd(), '..', 'raw_data', 'df.csv')
    df = pd.read_csv(data_path)
    df['Date'] = pd.to_datetime(df['Date'])
    df.set_index('Date', inplace=True)

    # Get actual data range
    data_start = df.index.min()
    data_end = df.index.max()
```

```
    # Adjust config dates to available data range
    if config.start_date:
        config.start_date = max(config.start_date, data_start)
    else:
        config.start_date = data_start

    if config.end_date:
        config.end_date = min(config.end_date, data_end)
    else:
        config.end_date = data_end

    # Filter data using adjusted dates
    df = df[(df.index >= config.start_date) & (df.index <= config.end_date)]

    return df

def create_portfolio(strategy, price, signals, config: BacktestConfig):
    # Filter price and signals to config date range if dates are specified
    if config.start_date is not None:
        price = price[price.index >= config.start_date]
        signals = signals[signals.index >= config.start_date]
    if config.end_date is not None:
        price = price[price.index <= config.end_date]
        signals = signals[signals.index <= config.end_date]

    # Convert signals to boolean if they're not already
    signals = signals.astype(bool)

    # Resample signals based on rebalance frequency
    if config.rebalance_freq != '1D':
        monthly_signals = signals.resample('M').last()
        signals = monthly_signals.reindex(price.index, method='ffill')
        signals = signals.astype(bool)

    # Generate entries and exits
    entries = signals & ~signals.shift(1).fillna(False)
    exits = ~signals & signals.shift(1).fillna(False)

    return vbt.Portfolio.from_signals(
        price,
        entries,
        exits,
        freq='1D',
        init_cash=config.initial_capital,
        size=config.size,
        size_type=config.size_type,
        accumulate=False
    )

def format_results(stats_dict):
    df_stats = pd.DataFrame.from_dict(stats_dict, orient='index').T
    df_stats = df_stats.sort_values(by='Total Return [%]', axis=1, ascending=False)

    ordered_rows = df_stats.index.tolist()
    total_return_idx = ordered_rows.index('Total Return [%]')
    ordered_rows.remove('Annualized Return [%]')
```

```
    ordered_rows.remove('Annualized Volatility [%]')
    ordered_rows.insert(total_return_idx + 1, 'Annualized Return [%]')
    ordered_rows.insert(total_return_idx + 2, 'Annualized Volatility [%]')
    df_stats = df_stats.reindex(ordered_rows)

    formatted_df = df_stats.copy()

    formatted_df.loc['Start'] = formatted_df.loc['Start'].apply(lambda x: pd.to_dat
    formatted_df.loc['End'] = formatted_df.loc['End'].apply(lambda x: pd.to_datetim

    percentage_rows = ['Total Return [%]', 'Annualized Return [%]', 'Annualized Vol
    for row in percentage_rows:
        formatted_df.loc[row] = formatted_df.loc[row].apply(
            lambda x: f"{x:.2f}%" if pd.notnull(x) else x
        )

    for row in ['Start Value', 'End Value']:
        formatted_df.loc[row] = formatted_df.loc[row].apply(lambda x: f"{x:.2f}" if

    duration_rows = ['Avg Winning Trade Duration', 'Avg Losing Trade Duration', 'Ma
    for row in duration_rows:
        if row in formatted_df.index:
            formatted_df.loc[row] = formatted_df.loc[row].apply(
                lambda x: f"{pd.Timedelta(x).days} days" if pd.notnull(x) else x
            )

    numeric_rows = [idx for idx in formatted_df.index
                    if idx not in ['Start', 'End'] + percentage_rows + duration_rows
    for row in numeric_rows:
        formatted_df.loc[row] = formatted_df.loc[row].apply(
            lambda x: f"{float(x):.2f}" if pd.notnull(x) and not isinstance(x, pd.T
        )

    styled_df = formatted_df.style.set_properties(**{
        'text-align': 'center'
    }).set_table_styles([
        {'selector': 'th', 'props': [('text-align', 'center')]}
    ])

    return styled_df

def run_backtest(strategies, config: BacktestConfig):
    all_stats = {}

    for strategy in strategies:
        with redirect_stdout(io.StringIO()):
            signals = strategy.generate_signals()

        price = strategy.df[strategy.target_col]
        signals = signals.reindex(price.index)

        if not isinstance(signals.dtype, pd.BooleanDtype):
            signals = signals.astype(bool)

        pf = create_portfolio(strategy, price, signals, config)
        stats_series = pf.stats()
```

```
        returns = pf.returns()
        returns_stats = returns.vbt.returns(freq='1D', year_freq='365D')

        stats_series['Annualized Return [%]'] = returns_stats.annualized() * 100
        stats_series['Annualized Volatility [%]'] = returns_stats.annualized_volati

        all_stats[strategy.__class__.__name__] = stats_series

    return format_results(all_stats)

# Initialize and run backtests
df = load_data(BacktestConfig())
strategies = [
    BuyAndHoldStrategy(df),
    VolatilityRegimeStrategy(df),
    AdaptiveTrendStrategy(df),
]

config_default = BacktestConfig(
    rebalance_freq='1D',
    initial_capital=100.0,
    size=1.0,
    size_type='percent'
)
results_default = run_backtest(strategies, config_default)

config_2020 = BacktestConfig(
    start_date='2009-01-01',
    end_date='2025-12-31',
    rebalance_freq='M',  # Use 'M' explicitly for monthly
    initial_capital=100,
    size=1.0,
    size_type='percent'
)
results_2020 = run_backtest(strategies, config_2020)

display(results_default)
display(results_2020)
```

	AdaptiveTrendStrategy	BuyAndHoldStrategy	VolatilityRegimeStrategy
Start	10/31/2002	10/31/2002	10/31/2002
End	12/27/2024	12/27/2024	12/27/2024
Period	5562 days	5562 days	5562 days
Start Value	100.00	100.00	100.00
End Value	225.32	134.69	109.89
Total Return [%]	125.32%	34.69%	9.89%
Annualized Return [%]	5.48%	1.97%	0.62%
Annualized Volatility [%]	1.07%	1.82%	0.41%
Benchmark Return [%]	34.69	34.69	34.69
Max Gross Exposure [%]	100.00	100.00	100.00
Total Fees Paid	0.00	0.00	0.00
Max Drawdown [%]	0.73	15.48	1.37
Max Drawdown Duration	72 days	981 days	950 days
Total Trades	302.00	1.00	51.00
Total Closed Trades	302.00	0.00	51.00
Total Open Trades	0.00	1.00	0.00
Open Trade PnL	0.00	34.69	0.00
Win Rate [%]	84.44	nan	70.59
Best Trade [%]	7.34	nan	1.72
Worst Trade [%]	-0.55	nan	-1.33
Avg Winning Trade [%]	0.33	nan	0.34
Avg Losing Trade [%]	-0.07	nan	-0.19
Avg Winning Trade Duration	11 days	NaT	19 days
Avg Losing Trade Duration	3 days	NaT	7 days

	AdaptiveTrendStrategy	BuyAndHoldStrategy	VolatilityRegimeStrategy
Profit Factor	25.71	nan	4.39
Expectancy	0.41	nan	0.19
Sharpe Ratio	4.97	1.08	1.50
Calmar Ratio	7.46	0.13	0.45
Omega Ratio	3.60	1.23	1.90
Sortino Ratio	11.04	1.43	2.42

	BuyAndHoldStrategy	AdaptiveTrendStrategy	VolatilityRegimeStrategy
Start	01/02/2009	01/02/2009	01/02/2009
End	12/27/2024	12/27/2024	12/27/2024
Period	4019 days	4019 days	4019 days
Start Value	100.00	100.00	100.00
End Value	150.27	144.61	103.90
Total Return [%]	50.27%	44.61%	3.90%
Annualized Return [%]	3.77%	3.41%	0.35%
Annualized Volatility [%]	1.87%	1.56%	0.50%
Benchmark Return [%]	50.27	50.27	50.27
Max Gross Exposure [%]	100.00	100.00	100.00
Total Fees Paid	0.00	0.00	0.00
Max Drawdown [%]	9.77	9.77	1.79
Max Drawdown Duration	308 days	313 days	1424 days
Total Trades	1.00	41.00	20.00
Total Closed Trades	0.00	40.00	20.00
Total Open Trades	1.00	1.00	0.00
Open Trade PnL	50.27	2.93	0.00
Win Rate [%]	nan	87.50	55.00
Best Trade [%]	nan	7.44	1.51
Worst Trade [%]	nan	-0.48	-0.77
Avg Winning Trade [%]	nan	1.06	0.67
Avg Losing Trade [%]	nan	-0.34	-0.39
Avg Winning Trade Duration	NaT	63 days	38 days
Avg Losing Trade Duration	NaT	41 days	25 days

	BuyAndHoldStrategy	AdaptiveTrendStrategy	VolatilityRegimeStrategy
Profit Factor	nan	21.07	2.06
Expectancy	nan	1.04	0.20
Sharpe Ratio	1.99	2.16	0.69
Calmar Ratio	0.39	0.35	0.19
Omega Ratio	1.48	1.79	1.35
Sortino Ratio	2.70	2.91	1.14

In [ ]: