1.Introduction

This project is a machine-learning-driven trading stratergy using a Random Forest Regressor trained on alpha signals. The project peforms the following:

- Walk Forward Validation to simulate real-model retraining.
- Monte Carlo Resampling with Antithetic Variates to evaluate robustness.
- Benchmark Comparison against buy-and-hold and naive strategies.

The goal is to estimate whether learned predicitive signals outpeform basic benchmarks on Sharpe ratio.

In [19]:
```
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
pd.set_option('display.max_rows', 10)
pd.set_option('display.max_columns', 10)
```

Load Historical Data

The function load_data() retrieves historical stock price data.

$P_t = ClosingPriceattimet$

In [6]:
```
def load_data(symbol="SPY", start="2010-01-01", end="2025-10-21"):
    """Downloads historical stock data."""
    df = yf.download(symbol, start=start, end=end, auto_adjust=False)
    return df
```

Featured Engineering - Alpha Signals

This function creates multiple alpha features derived from price momentum, trend, and volatility.

1. Simple Moving Average (SMA)

The **Simple Moving Average** over a period (n) smooths price data and helps identify trends:

$$SMA_t^{(n)} = \frac{1}{n}\sum_{i=0}^{n-1} P_{t-i}$$

Where:

- (P_t) = Close price at time (t)
- (n) = SMA window length

2. Alpha Momentum Signals (SMA Crossovers)

Short-term momentum relative to long-term trend:

$$\text{Alpha\_MOM}_{a,b} = SMA_t^{(a)} - SMA_t^{(b)}$$

A positive value indicates that the shorter-term SMA is above the longer-term SMA, suggesting bullish momentum.

3. Rate of Change (ROC)

Measures percentage change over (n) periods:

$$ROC_t^{(n)} = \frac{P_t - P_{t-n}}{P_{t-n}} = \frac{P_t}{P_{t-n}} - 1$$

ROC captures trend strength over different horizons.

4. SMA Ratio

Ratio of short-term to long-term SMA:

$$\text{Alpha\_Ratio}_{a,b} = \frac{SMA_t^{(a)}}{SMA_t^{(b)}}$$

If (\text{Alpha_Ratio}_{a,b} > 1), short-term momentum exceeds the long-term trend.

5. Average True Range (ATR)

Measures volatility using True Range (TR):

$$TR_t = \max\left(H_t - L_t, |H_t - C_{t-1}|, |L_t - C_{t-1}|\right)$$

$$ATR_t^{(n)} = \frac{1}{n}\sum_{i=0}^{n-1} TR_{t-i}$$

Where:

- H_t = High price at time (t)
- L_t = Low price at time (t)
- C_t = Close price at time (t)
- n = ATR window (commonly 14 days)

In [7]:
```
def gen_alpha_signals(df):
    """Calculates various alpha and intermediate technical features (SMAs, MOM, ROC, ATR)."""
    momentum_periods = [10,30,60,120,240]
    df_features = pd.DataFrame(index=df.index)

    # --- 1. Calculate Intermediate SMAs ---
    for period in momentum_periods:
        sma_col = f"SMA_{period}"
        df_features[sma_col] = df["Close"].rolling(window=period).mean()

    # --- 2. Alpha Momentum Signals (Crossover differences) ---
    df_features["Alpha_MOM_10_30"] = df_features["SMA_10"] - df_features["SMA_30"]
    df_features["Alpha_MOM_30_60"] = df_features["SMA_30"] - df_features["SMA_60"]
    df_features["Alpha_MOM_60_120"] = df_features["SMA_60"] - df_features["SMA_120"]
```

```
    df_features["Alpha_MOM_120_240"] = df_features["SMA_120"] - df_features["SMA_240"]

    # --- 3. Alpha Trend Signals (Rate of Change) ---
    df_features["Alpha_Trend_ROC_20"] = df["Close"].pct_change(20)
    df_features["Alpha_Trend_ROC_60"] = df["Close"].pct_change(60)
    df_features["Alpha_Trend_ROC_120"] = df["Close"].pct_change(120)
    df_features["Alpha_Ratio_10_60"] = df_features["SMA_10"] / df_features["SMA_60"]

    # --- 4. Alpha Volatility Signal (ATR) ---
    df_temp = df[['High', 'Low', 'Close']].copy()
    df_temp['High-Low'] = df_temp['High'] - df_temp['Low']
    df_temp['High-PrevClose'] = np.abs(df_temp['High'] - df_temp['Close'].shift(1))
    df_temp['Low-PrevClose'] = np.abs(df_temp['Low'] - df_temp['Close'].shift(1))
    df_temp['TR'] = df_temp[['High-Low', 'High-PrevClose', 'Low-PrevClose']].max(axis=1)
    df_features['Alpha_VOL_ATR_14'] = df_temp['TR'].rolling(window=14).mean()

    # --- 5. Cleanup ---
    sma_columns_to_drop = [f"SMA_{period}" for period in momentum_periods]
    df_features.drop(columns=sma_columns_to_drop, inplace=True, errors='ignore')
    df_features.dropna(inplace=True)

    alpha_cols = [col for col in df_features.columns if col.startswith('Alpha_')]

    return df_features[alpha_cols]
```

Future Return (Target Variable)

We define the target as the forward return over (n) periods (e.g., 10 days):

$$Y_t = \frac{P_{t+n} - P_t}{P_t} = \frac{P_{t+n}}{P_t} - 1$$

The model aims to predict (Y_t) from alpha signals.

Trading signal positions:

$$\text{position}_t = \begin{cases} +1 & \text{if } \hat{Y}_t > \text{threshold} \\ -1 & \text{if } \hat{Y}_t < -\text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

Strategy returns:

$$R_t^{\text{strategy}} = \text{position}_t \times Y_t$$

Trend filter (SMA):

$$\text{Take long only if } P_t > SMA_{100,t}$$

In [8]:
```
def create_target(df, period=10):
    df_target = df.copy()
    target_col = f"Future_Return_{period}D"
    df_target[target_col] = df_target["Close"].pct_change(period).shift(-period)
    df_target.dropna(inplace=True)
    return df_target[target_col]
```

Sharpe Ratio

Annualized Sharpe Ratio:

$$\text{Sharpe} = \frac{\sqrt{252} \cdot \mathbb{E}[R_t]}{\sigma(R_t)}$$

Where:

- (R_t) = daily strategy return
- (\sigma(R_t)) = standard deviation of returns
- (252) = trading days per year

In [9]:
```python
def calculate_sharpe_ratio(returns, days_per_year=252):
    """Calculates annualized Sharpe Ratio."""
    if returns.std() == 0:
        return 0
    return np.sqrt(days_per_year) * returns.mean() / returns.std()
```

Trading Simulation

$$\text{position}_t = \begin{cases} +1 & \text{if } \hat{Y}_t > \text{threshold} \\ -1 & \text{if } \hat{Y}_t < -\text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

In [10]:
```python
def simulate_trading(model, X_val, y_val, df_prices, sma_filter_df, threshold=0.0015):
    """
    Simulates trading positions based on model predictions and a trend filter.
    Returns the strategy returns and positions.
    """
    # Predict returns and ensure 1D shape (N,)
    predicted_returns = model.predict(X_val).ravel()

    # Get current prices and SMA filter ONLY for the X_val index
    # Use .loc and .squeeze() to guarantee 1D Series for comparison, fixing alignment errors.
    current_price_val = df_prices['Close'].loc[X_val.index].squeeze()
    sma_filter_val = sma_filter_df.loc[X_val.index].squeeze()

    # Calculate boolean trend: price > SMA
    trending_up = (current_price_val > sma_filter_val).values.ravel()

    # Calculate raw positions based on prediction and threshold
    positions_raw = np.where(predicted_returns > threshold, 1,
                             np.where(predicted_returns < -threshold, -1, 0))
    positions_raw = positions_raw.ravel() # Ensure positions_raw is 1D array

    # Apply the SMA trend filter: Force short positions to 0 if market is trending up
    positions = np.where(
        trending_up & (positions_raw == -1),
        0,
        positions_raw
    )

    positions = positions.ravel() # Ensure positions is 1D array
```

```
    # Create strategy positions Series, aligned to X_val index
    strategy_positions = pd.Series(positions, index=X_val.index)

    # Calculate strategy returns
    strategy_returns = strategy_positions * y_val.loc[X_val.index]

    return strategy_returns.dropna(), strategy_positions.dropna()
```

Walk-Forward Validation

Walk-forward validation simulates a **realistic, out-of-sample scenario** by training the model on past data and testing on future, unseen data.

Process:

1. Split data into training and validation sets:

   $$X_{\text{train}}, Y_{\text{train}} = \text{first } 70\% \text{ of data}$$

   $$X_{\text{val}}, Y_{\text{val}} = \text{remaining } 30\%$$

2. Train the Random Forest model on the training set:

   $$\hat{f} = \text{RandomForestRegressor}(X_{\text{train}}, Y_{\text{train}})$$

3. Predict and simulate trading on the validation set:

   $$\hat{Y}_{\text{val}} = \hat{f}(X_{\text{val}})$$

   $$R_t^{\text{strategy}} = \text{position}_t \times \hat{Y}_{\text{val},t}$$

4. Evaluate performance using **Sharpe ratio**, cumulative returns, and drawdowns.

Walk-forward validation ensures the model is tested in a **chronologically realistic environment**, preventing look-ahead bias.

In [11]:
```
def walk_forward_validation(X, Y, df_full_prices, train_ratio=0.7, n_estimators=500, min_samp
    """Performs walk-forward validation with a fixed training-validation split."""
    val_sharpe_ratios = []
    # Prepare full price DataFrame and SMA filter
    df_full_prices_copy = df_full_prices.copy()

    # Calculate 100-day SMA trend filter
    df_full_prices_copy['SMA_100_Filter'] = df_full_prices_copy['Close'].rolling(window=100).
    sma_filter_full = df_full_prices_copy['SMA_100_Filter']

    # Determine split point for train/validation sets
    split_point_idx = int(len(X) * train_ratio)

    X_train_wf = X.iloc[:split_point_idx]
    Y_train_wf = Y.iloc[:split_point_idx]
    X_val_wf = X.iloc[split_point_idx:]
    Y_val_wf = Y.iloc[split_point_idx:]

    print(f"\n--- Walk-Forward Validation (Single Split) ---")
    print(f"Train samples: {len(X_train_wf)}")
    print(f"Validation samples: {len(X_val_wf)}")
```

```
    # Train the Random Forest model
    model = RandomForestRegressor(n_estimators=n_estimators, min_samples_leaf=min_samples_lea
                                  max_depth=max_depth, random_state=42, n_jobs=-1)
    model.fit(X_train_wf, Y_train_wf)

    # Simulate trading on the validation set
    strategy_returns_val, _ = simulate_trading(model, X_val_wf, Y_val_wf, df_full_prices_copy
    # Calculate and store Sharpe ratio for validation
    sharpe = calculate_sharpe_ratio(strategy_returns_val)
    val_sharpe_ratios.append(sharpe)

    print(f"Validation Sharpe Ratio: {sharpe:.2f}")

    return val_sharpe_ratios, strategy_returns_val, X_val_wf, Y_val_wf
```

Monte Carlo Resampling

Monte Carlo resampling assesses the **robustness and stability** of the strategy under different random train-validation splits.

Key Steps:

1. Randomly choose training proportions p_1 in $[0.6, 0.85]$ and create an **antithetic pair**:

   $$p_2 = 0.6 + 0.85 - p_1$$

2. For each split:

   - Train a Random Forest on the first $p_i \cdot N$ samples
   - Validate on the remaining data
   - Compute strategy returns and Sharpe ratio

3. Repeat for multiple simulations (e.g., 500), then average the antithetic pair results:

   $$\text{Sharpe}_{\text{avg}} = \frac{\text{Sharpe}_{p_1} + \text{Sharpe}_{p_2}}{2}$$

Antithetic variates **reduce variance** in Monte Carlo estimates, giving a more stable evaluation of strategy performance across random splits.

In [12]:
```
def monte_carlo_resampling_antithetic(X, Y, df_full_prices, num_simulations=100, train_ratio_
    """Performs Monte Carlo resampling with antithetic variates for training-validation splits
    mc_sharpe_ratios = []
    # Prepare full price DataFrame and SMA filter
    df_full_prices_copy = df_full_prices.copy()
    df_full_prices_copy['SMA_100_Filter'] = df_full_prices_copy['Close'].rolling(window=100).
    sma_filter_full = df_full_prices_copy['SMA_100_Filter']

    print(f"\n--- Monte Carlo Resampling with Antithetic Variates ({num_simulations} pairs of
    # Define bounds for split proportions
    low_prop = train_ratio_min
    high_prop = train_ratio_max

    for i in range(num_simulations):

        # Generate two complementary split proportions (p1 and p2)
```

```
        p1 = np.random.uniform(low_prop, high_prop)
        p2 = low_prop + high_prop - p1

        split_idx_1 = int(len(X) * p1)
        split_idx_2 = int(len(X) * p2)

        split_indices = [split_idx_1, split_idx_2]

        valid_sharpes_for_pair = []

        for j, current_split_idx in enumerate(split_indices):

            # Skip invalid splits
            if current_split_idx < X.shape[0] * low_prop or current_split_idx > X.shape[0] *
                continue

            X_train_mc = X.iloc[:current_split_idx]
            Y_train_mc = Y.iloc[:current_split_idx]
            X_val_mc = X.iloc[current_split_idx:]
            Y_val_mc = Y.iloc[current_split_idx:]

            if len(X_train_mc) == 0 or len(X_val_mc) == 0:
                continue

            # Train and simulate for the current split
            model = RandomForestRegressor(n_estimators=n_estimators, min_samples_leaf=min_sam
                                          max_depth=max_depth, random_state=i*2 + j, n_jobs=-
            model.fit(X_train_mc, Y_train_mc)

            strategy_returns_mc, _ = simulate_trading(model, X_val_mc, Y_val_mc, df_full_pric

            if not strategy_returns_mc.empty:
                sharpe = calculate_sharpe_ratio(strategy_returns_mc)
                valid_sharpes_for_pair.append(sharpe)

        # Average the antithetic results
        if len(valid_sharpes_for_pair) == 2:
            mc_sharpe_ratios.append(np.mean(valid_sharpes_for_pair))
        elif len(valid_sharpes_for_pair) == 1:
            mc_sharpe_ratios.append(valid_sharpes_for_pair[0])

        # Progress update
        if (i + 1) % (num_simulations // 10 if num_simulations >= 10 else 1) == 0 or i == num
            if mc_sharpe_ratios:
                print(f"  Pair {i+1}/{num_simulations} complete. Current Mean Sharpe: {np.mea
            else:
                print(f"  Pair {i+1}/{num_simulations} complete. No valid Sharpe ratios yet."

    return mc_sharpe_ratios
```

Benchmark Comparisons

Buy & Hold (Daily):

$$R_t^{\mathrm{B\backslash\&H}} = \frac{P_t}{P_{t-1}} - 1$$

Total Buy & Hold Return:

$$R_{\text{total}}^{\mathrm{B\backslash\&H}} = \frac{P_T}{P_0} - 1$$

Matched Mechanics Benchmark:

$$R_t^{\text{bench}} = \frac{P_t}{P_{t-n}} - 1$$

Matches the same rolling return horizon as the strategy (e.g., 10-day returns).

In [13]:
```python
def calculate_all_benchmarks(y_val, df_prices, X_val_index, strategy_returns):

    benchmarks = {}

    # 1. Benchmark matching strategy mechanics (always long, 10-day returns)
    benchmarks['matched_mechanics'] = y_val.loc[X_val_index]

    # 2. Daily buy-and-hold
    benchmarks['buy_hold_daily'] = df_prices['Close'].pct_change(1).loc[X_val_index].dropna()

    # 3. Total buy-and-hold return (start to end)
    val_prices = df_prices['Close'].loc[X_val_index]
    start_price = val_prices.iloc[0]
    end_price = val_prices.iloc[-1]
    # Ensure scalar values
    if isinstance(start_price, pd.Series):
        start_price = start_price.item()
    if isinstance(end_price, pd.Series):
        end_price = end_price.item()
    benchmarks['buy_hold_total'] = (end_price / start_price) - 1

    # Align indices for fair comparison
    common_idx = strategy_returns.index.intersection(benchmarks['matched_mechanics'].index)

    return benchmarks, common_idx
```

Plotting + Drawdowns + Calmar Ratio

Drawdown at time (t):

$$DD_t = \frac{P_t - \max(P_0, P_1, \ldots, P_t)}{\max(P_0, P_1, \ldots, P_t)} \times 100\%$$

Maximum drawdown:

$$DD_{\max} = \min_t(DD_t)$$

Calmar ratio:

$$\text{Calmar} = \frac{\text{Total Return}}{|\text{Max Drawdown}|}$$

Measures risk-adjusted return relative to worst peak-to-trough decline.

In [21]:
```python
def plot_with_proper_benchmarks(strategy_returns, y_val, df_prices, X_val_index):
    """
    Creates comparison plots with proper benchmarks.
    """
    benchmarks, common_idx = calculate_all_benchmarks(y_val, df_prices, X_val_index, strategy_
```

```
# Align all series to common index
strategy_aligned = strategy_returns.loc[common_idx]
benchmark_mechanics = benchmarks['matched_mechanics'].loc[common_idx]

# Calculate cumulative returns
cum_strategy = (1 + strategy_aligned).cumprod()
cum_benchmark_mechanics = (1 + benchmark_mechanics).cumprod()

# For buy-and-hold daily, need to align separately
common_bh_idx = strategy_returns.index.intersection(benchmarks['buy_hold_daily'].index)
if len(common_bh_idx) > 0:
    strategy_bh_aligned = strategy_returns.loc[common_bh_idx]
    benchmark_bh = benchmarks['buy_hold_daily'].loc[common_bh_idx]
    cum_benchmark_bh = (1 + benchmark_bh).cumprod()
else:
    # Fallback if no common index
    benchmark_bh = benchmarks['buy_hold_daily']
    cum_benchmark_bh = (1 + benchmark_bh).cumprod()

# Calculate drawdowns for strategy
peak_strategy = cum_strategy.expanding(min_periods=1).max()
drawdown_strategy = ((cum_strategy / peak_strategy) - 1) * 100

# Calculate drawdowns for benchmark
peak_benchmark = cum_benchmark_mechanics.expanding(min_periods=1).max()
drawdown_benchmark = ((cum_benchmark_mechanics / peak_benchmark) - 1) * 100

# Create figure with 3 subplots
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(14, 12),
                                    gridspec_kw={'height_ratios': [3, 1, 1]})

# Plot 1: Cumulative returns
if len(cum_benchmark_bh) > 0:
    cum_benchmark_bh.plot(ax=ax1, label='Buy & Hold (Daily Returns)',
                          color='lightgray', linestyle=':', linewidth=2)
cum_benchmark_mechanics.plot(ax=ax1, label='Benchmark (Always Long, 10D Returns)',
                             color='orange', linestyle='--', linewidth=2)
cum_strategy.plot(ax=ax1, label='ML Strategy', color='blue', linewidth=2)

ax1.set_title('Strategy Performance: Proper Benchmark Comparison', fontsize=14, fontweigh
ax1.set_ylabel('Cumulative Return (Log Scale)', fontsize=12)
ax1.set_yscale('log')
ax1.legend(loc='upper left', fontsize=10)
ax1.grid(True, linestyle='--', alpha=0.7)

# Plot 2: Strategy drawdown
drawdown_strategy.plot(ax=ax2, color='red', linewidth=1.5, label='Strategy DD')
ax2.fill_between(drawdown_strategy.index, drawdown_strategy, 0, color='red', alpha=0.3)
ax2.set_ylabel('Strategy\nDrawdown (%)', fontsize=11)
ax2.grid(True, linestyle='--', alpha=0.7)
ax2.set_ylim(-100, 0)
ax2.yaxis.set_major_formatter(mtick.PercentFormatter(decimals=0))
ax2.legend(loc='lower left', fontsize=9)

# Plot 3: Benchmark drawdown
drawdown_benchmark.plot(ax=ax3, color='orange', linewidth=1.5, label='Benchmark DD')
ax3.fill_between(drawdown_benchmark.index, drawdown_benchmark, 0, color='orange', alpha=0
ax3.set_ylabel('Benchmark\nDrawdown (%)', fontsize=11)
ax3.set_xlabel('Date', fontsize=12)
ax3.grid(True, linestyle='--', alpha=0.7)
ax3.set_ylim(-100, 0)
ax3.yaxis.set_major_formatter(mtick.PercentFormatter(decimals=0))
ax3.legend(loc='lower left', fontsize=9)
```

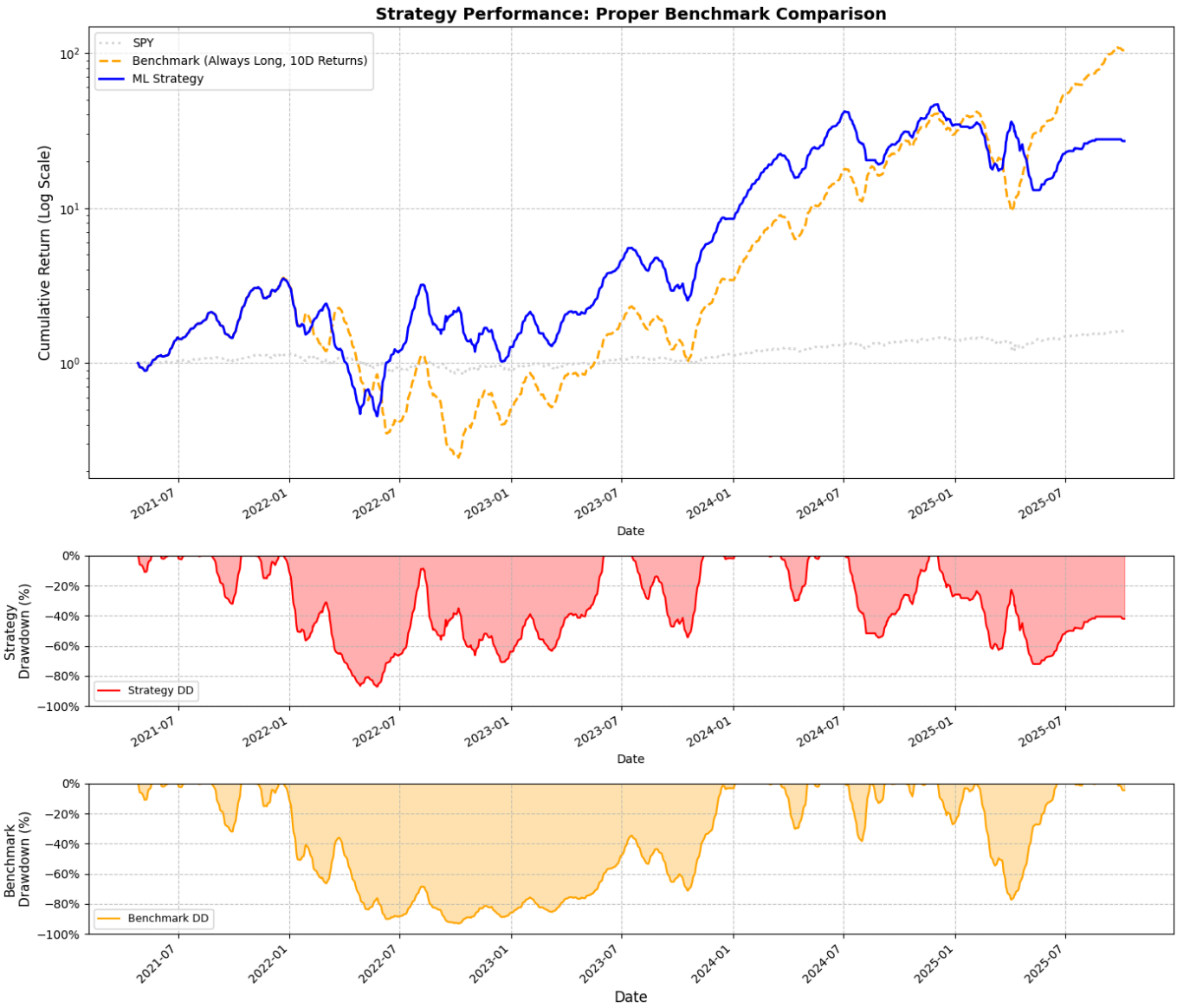
```
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Print statistics
print("\n=== BENCHMARK COMPARISON STATISTICS ===\n")

def calc_sharpe(returns, days_per_year=252):
    returns_std = returns.std()
    returns_mean = returns.mean()
    # Handle both scalar and Series returns
    if isinstance(returns_std, pd.Series):
        returns_std = returns_std.item() if len(returns_std) == 1 else returns_std.iloc[0
    if isinstance(returns_mean, pd.Series):
        returns_mean = returns_mean.item() if len(returns_mean) == 1 else returns_mean.il
    if returns_std == 0:
        return 0
    return np.sqrt(days_per_year) * returns_mean / returns_std

def calc_max_dd(returns):
    cum_ret = (1 + returns).cumprod()
    running_max = cum_ret.expanding(min_periods=1).max()
    dd = ((cum_ret / running_max) - 1) * 100
    dd_min = dd.min()
    # Handle both scalar and Series returns
    if isinstance(dd_min, pd.Series):
        dd_min = dd_min.item() if len(dd_min) == 1 else dd_min.iloc[0]
    return dd_min

def calc_calmar(total_return, max_dd):
    # Ensure total_return and max_dd are scalars
    if isinstance(total_return, pd.Series):
        total_return = total_return.item() if len(total_return) == 1 else total_return.il
    if isinstance(max_dd, pd.Series):
        max_dd = max_dd.item() if len(max_dd) == 1 else max_dd.iloc[0]
    if max_dd == 0:
        return 0
    return (total_return * 100) / abs(max_dd)

print("Strategy:")
print(f"  Total Return: {(cum_strategy.iloc[-1] - 1) * 100:.2f}%")
print(f"  Sharpe Ratio: {calc_sharpe(strategy_aligned):.2f}")
print(f"  Max Drawdown: {calc_max_dd(strategy_aligned):.2f}%")
print(f"  Calmar Ratio: {calc_calmar(cum_strategy.iloc[-1] - 1, calc_max_dd(strategy_alig

print("\nBenchmark (Always Long, 10D Returns):")
print(f"  Total Return: {(cum_benchmark_mechanics.iloc[-1] - 1) * 100:.2f}%")
print(f"  Sharpe Ratio: {calc_sharpe(benchmark_mechanics):.2f}")
print(f"  Max Drawdown: {calc_max_dd(benchmark_mechanics):.2f}%")
print(f"  Calmar Ratio: {calc_calmar(cum_benchmark_mechanics.iloc[-1] - 1, calc_max_dd(be

if len(cum_benchmark_bh) > 0:
    print("\nBuy & Hold (Daily Returns):")
    bh_total_return = cum_benchmark_bh.iloc[-1] if isinstance(cum_benchmark_bh.iloc[-1],
    print(f"  Total Return: {(bh_total_return - 1) * 100:.2f}%")
    print(f"  Sharpe Ratio: {calc_sharpe(benchmark_bh):.2f}")
    print(f"  Max Drawdown: {calc_max_dd(benchmark_bh):.2f}%")
    print(f"  Calmar Ratio: {calc_calmar(bh_total_return - 1, calc_max_dd(benchmark_bh)):

    print(f"\nSimple Buy & Hold (Start to End): {benchmarks['buy_hold_total'] * 100:.2f}%

    print("\n=== EXCESS RETURNS ===")
```

```
        print(f"Strategy vs Matched Benchmark: {((cum_strategy.iloc[-1] / cum_benchmark_mecha
        bh_cum = bh_total_return if isinstance(bh_total_return, (int, float)) else bh_total_r
        print(f"Strategy vs Buy & Hold: {((cum_strategy.iloc[-1] / bh_cum) - 1) * 100:.2f}%")
    else:
        print(f"\nSimple Buy & Hold (Start to End): {benchmarks['buy_hold_total'] * 100:.2f}%

        print("\n=== EXCESS RETURNS ===")
        print(f"Strategy vs Matched Benchmark: {((cum_strategy.iloc[-1] / cum_benchmark_mecha

    return benchmarks, cum_strategy, cum_benchmark_mechanics, cum_benchmark_bh
```

The overall workflow of the strategy:

In [24]:
```
# 1. Load Data
df = load_data()

# 2. Generate Features (X) and Target (Y)
x = gen_alpha_signals(df.copy())
y = create_target(df.copy())

# 3. Critical Data Alignment: Align X, Y, and the full price DataFrame (df)
# This prevents index mismatch errors during simulation.
common_index = x.index.intersection(y.index)
x = x.loc[common_index]
y = y.loc[common_index]
df_aligned = df.loc[common_index].copy()

print(f"Total Aligned Samples: {len(x)}")

# 4. Walk-Forward Validation
wf_sharpes, wf_strategy_returns, wf_X_val, wf_Y_val = walk_forward_validation(x, y, df_aligne

# 5. Plotting Walk-Forward Results with Proper Benchmarks
benchmarks, cum_strategy, cum_benchmark_mechanics, cum_benchmark_bh = plot_with_proper_benchm
```

```
[**********100%**********]  1 of 1 completed
Total Aligned Samples: 3725

--- Walk-Forward Validation (Single Split) ---
Train samples: 2607
Validation samples: 1118
Validation Sharpe Ratio: 1.78
```




```
=== BENCHMARK COMPARISON STATISTICS ===

Strategy:
  Total Return: 2607.26%
  Sharpe Ratio: 1.78
  Max Drawdown: -87.06%
  Calmar Ratio: 29.95

Benchmark (Always Long, 10D Returns):
  Total Return: 10316.77%
  Sharpe Ratio: 2.37
  Max Drawdown: -93.09%
  Calmar Ratio: 110.83

Buy & Hold (Daily Returns):
  Total Return: 61.16%
  Sharpe Ratio: 0.70
  Max Drawdown: -25.36%
  Calmar Ratio: 2.41

Simple Buy & Hold (Start to End): 60.82%

=== EXCESS RETURNS ===
Strategy vs Matched Benchmark: -74.01%
Strategy vs Buy & Hold: 1579.88%
```

Graph - Analysis

Cumulative Returns (Top Plot)

Blue line: ML strategy cumulative returns

Orange dashed line: Benchmark (always long, 10-day returns)

Grey dotted line: SPY (for context)

Observations:

The ML strategy has strong upward trends but is more volatile than daily buy-and-hold.

The benchmark shows higher total return (10316.77%) but suffers extreme drawdowns.

On a log scale, you can see how compounding affects both strategies over time.

Drawdowns (Middle and Bottom Plots)

Red area: Strategy drawdown

Orange area: Benchmark drawdown

Key points:

The ML strategy has a maximum drawdown of -87.06% — significant but less severe than the benchmark (-93.09%).

Drawdown visualization shows periods where capital temporarily declines before recovery.

Statistics - Analysis

Sharpe ratio: ML strategy is risk-adjusted and quite good (1.78), outperforming daily buy-and-hold (0.70).

Calmar ratio: Indicates return vs drawdown. ML strategy is less extreme than benchmark.

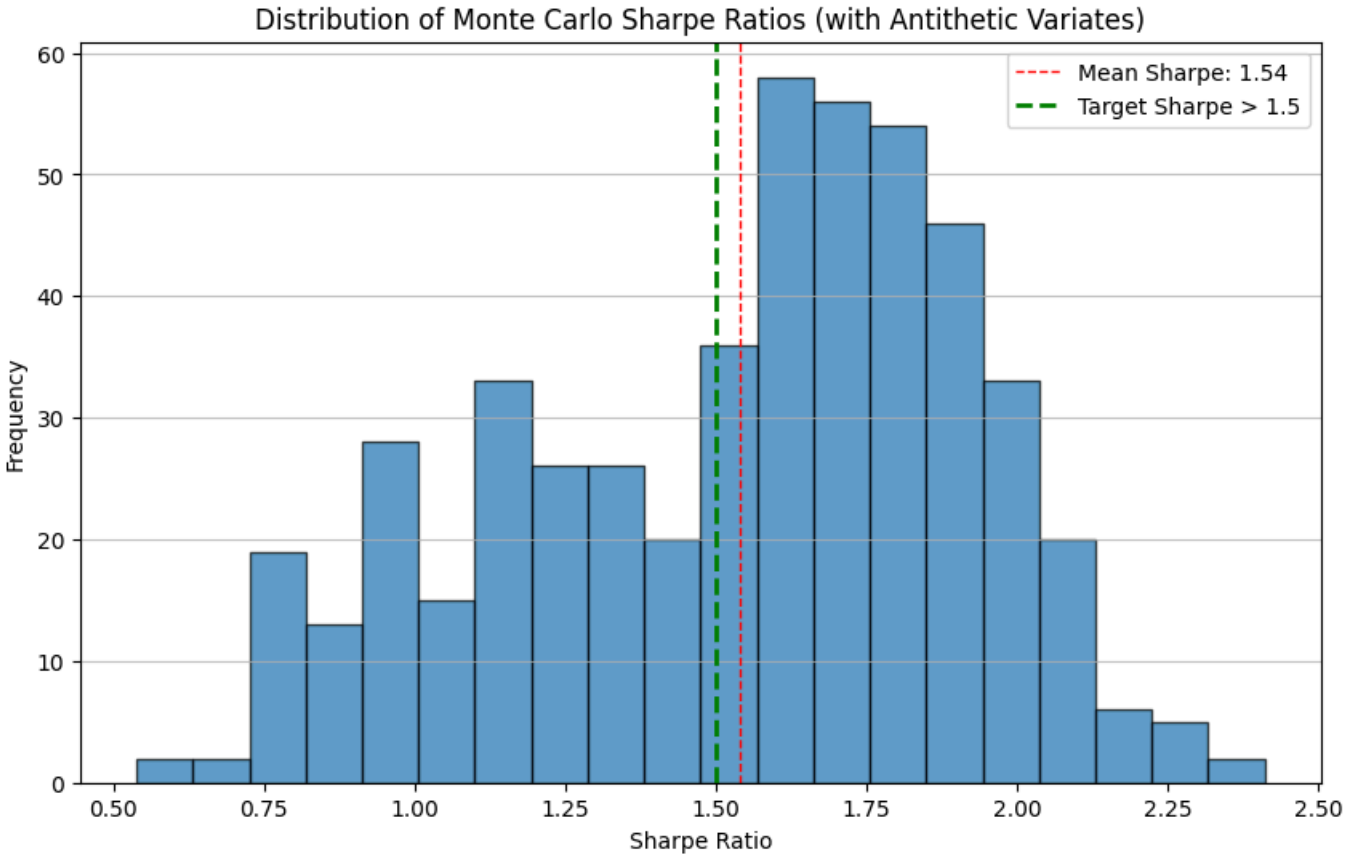
Excess Returns:

- vs matched benchmark: -74.01% (strategy underperformed in absolute returns)
- vs daily buy-and-hold: +1579.88% (outperformed simple long-term investment by a large margin)

In [16]:
```
mc_sharpes_antithetic = monte_carlo_resampling_antithetic(x, y, df_aligned, num_simulations=5

# 7. Reporting Monte Carlo Results
print(f"\n--- Monte Carlo Resampling Results (with Antithetic Variates) ---")
if mc_sharpes_antithetic:
    mc_sharpes_antithetic = np.array(mc_sharpes_antithetic)
    print(f"Number of successful averaged antithetic pairs: {len(mc_sharpes_antithetic)}")
    print(f"Mean Monte Carlo Sharpe Ratio: {np.mean(mc_sharpes_antithetic):.2f}")
    print(f"Median Monte Carlo Sharpe Ratio: {np.median(mc_sharpes_antithetic):.2f}")
    print(f"Standard Deviation of Monte Carlo Sharpe Ratios: {np.std(mc_sharpes_antithetic):.
    print(f"Sharpe Ratio > 1.5 in {np.sum(mc_sharpes_antithetic > 1.5) / len(mc_sharpes_antitl

    # Plot histogram of Monte Carlo Sharpe Ratios
    plt.figure(figsize=(10, 6))
```

```
    plt.hist(mc_sharpes_antithetic, bins=20, edgecolor='black', alpha=0.7)
    plt.axvline(np.mean(mc_sharpes_antithetic), color='red', linestyle='dashed', linewidth=1,
    plt.axvline(1.5, color='green', linestyle='--', linewidth=2, label='Target Sharpe > 1.5')
    plt.title('Distribution of Monte Carlo Sharpe Ratios (with Antithetic Variates)')
    plt.xlabel('Sharpe Ratio')
    plt.ylabel('Frequency')
    plt.legend()
    plt.grid(axis='y', alpha=0.75)
    plt.show()
else:
    print("No successful Monte Carlo simulations with antithetic variates to report.")
```

```
--- Monte Carlo Resampling with Antithetic Variates (500 pairs of simulations) ---
  Pair 50/500 complete. Current Mean Sharpe: 1.60
  Pair 100/500 complete. Current Mean Sharpe: 1.58
  Pair 150/500 complete. Current Mean Sharpe: 1.56
  Pair 200/500 complete. Current Mean Sharpe: 1.56
  Pair 250/500 complete. Current Mean Sharpe: 1.56
  Pair 300/500 complete. Current Mean Sharpe: 1.57
  Pair 350/500 complete. Current Mean Sharpe: 1.57
  Pair 400/500 complete. Current Mean Sharpe: 1.57
  Pair 450/500 complete. Current Mean Sharpe: 1.55
  Pair 500/500 complete. Current Mean Sharpe: 1.54

--- Monte Carlo Resampling Results (with Antithetic Variates) ---
Number of successful averaged antithetic pairs: 500
Mean Monte Carlo Sharpe Ratio: 1.54
Median Monte Carlo Sharpe Ratio: 1.62
Standard Deviation of Monte Carlo Sharpe Ratios: 0.38
Sharpe Ratio > 1.5 in 61.60% of simulations
```



Distribution of Monte Carlo Sharpe Ratios

The histogram above illustrates the frequency of the simulated Sharpe Ratios. Key Observations from the Histogram

1. Mean vs. Target: The calculated Mean Sharpe Ratio is 1.54 (indicated by the dashed red line), which is slightly above the Target Sharpe of 1.5 (indicated by the dashed green line). This suggests that, on average, the simulated portfolio meets the target performance criterion.
2. Distribution Shape: The distribution appears somewhat left-skewed (or negatively skewed), as the Median Sharpe (1.62) is higher than the Mean Sharpe (1.54). The bulk of the probability mass is concentrated to the right of the mean, specifically between 1.50 and 2.00.
3. Success Rate: Consistent with the summary table, 61.60% of the simulations resulted in a Sharpe Ratio greater than the target of 1.5. The bins to the right of the green line represent the majority of the outcomes.
4. Variability: The Standard Deviation of 0.38 indicates a moderate spread in the potential Sharpe Ratios, ranging from approximately 0.50 to 2.50. The presence of a significant number of outcomes well below the target (e.g., in the $0.75 - 1.25$ range) highlights the portfolio's downside volatility or potential for underperformance in certain scenarios.