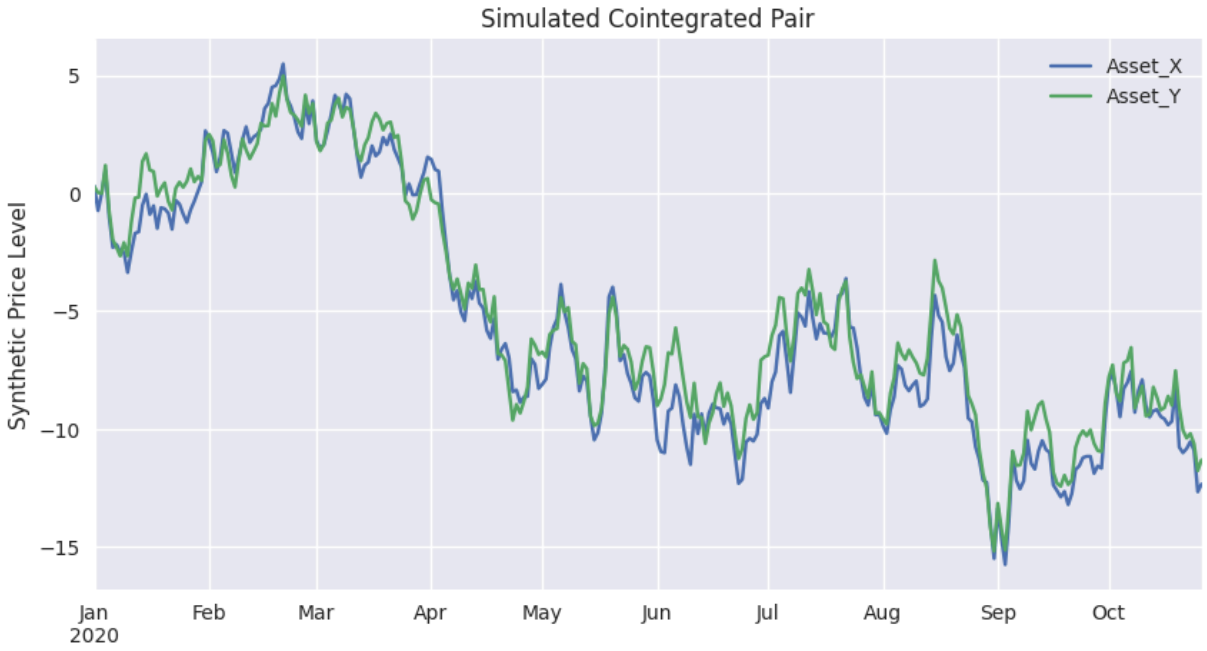
Pairs Trading Cointegration Sandbox

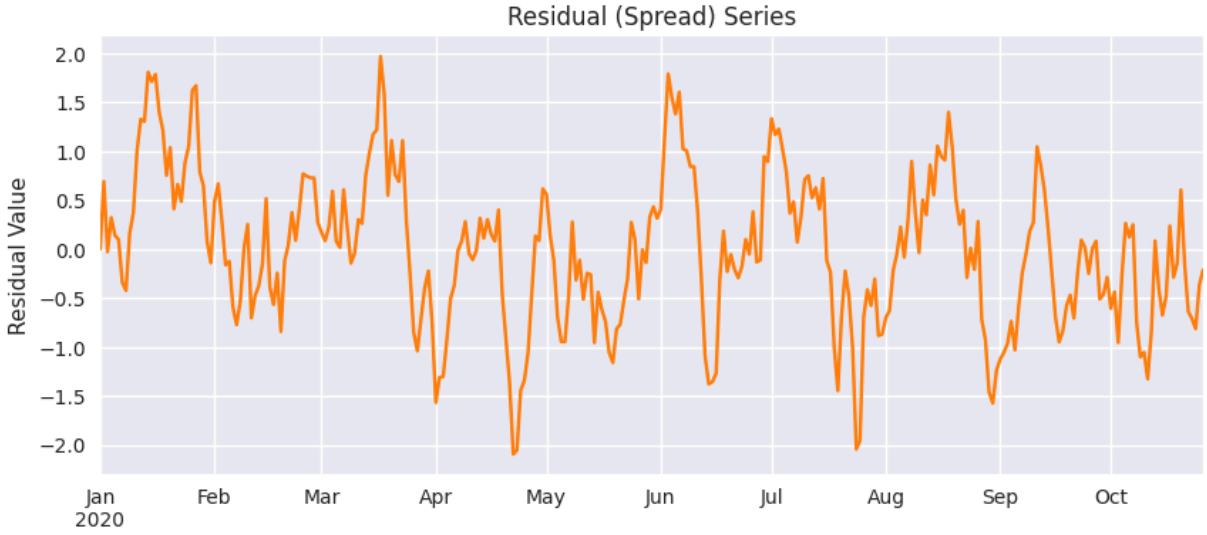
This notebook simulates a pair of cointegrated assets, visualizes their dynamics, and walks through key statistical diagnostics (Engle-Granger and Johansen). It closes with simple optimization heuristics—minimum profit checks and mean first passage times—to motivate parameter tuning for a pairs strategy.

Table of Contents

In [9]:
```
# Import numpy for numerical operations
import numpy as np
# Import pandas for labeled time series handling
import pandas as pd
# Import matplotlib for charting the simulated series
import matplotlib.pyplot as plt
# Import Plotly for interactive optimization visualizations
import plotly.graph_objects as go
# Apply a clean plotting style for readability
plt.style.use("seaborn-v0_8")

# Define a function that simulates a pair of cointegrated time series
def simulate_cointegrated_pair(n_points=300, beta=0.9, alpha=0.0, noise_scale=0.4,
    # Initialize a random number generator for reproducibility
    rng = np.random.default_rng(seed)
    # Draw random innovations that will build the primary price series
    x_innovations = rng.normal(0.0, 1.0, n_points)
    # Accumulate the innovations to obtain a random walk for the first asset
    x_series = np.cumsum(x_innovations)
    # Prepare an array to hold the stationary residual component
    residual = np.zeros(n_points)
    # Iterate through each time step to evolve the residual as a mean-reverting pro
```

```
    for t in range(1, n_points):
        # Combine persistence with fresh noise to keep the residual stationary
        residual[t] = 0.85 * residual[t - 1] + rng.normal(0.0, noise_scale)
    # Construct the second asset so that it stays close to a linear combination of
    y_series = alpha + beta * x_series + residual
    # Create a date index to label the simulated observations
    index = pd.date_range("2020-01-01", periods=n_points, freq="D")
    # Assemble the two series into a single DataFrame for convenience
    paired_df = pd.DataFrame({"Asset_X": x_series, "Asset_Y": y_series}, index=inde
    # Store the residual explicitly to simplify later diagnostics
    paired_df["Residual"] = residual
    # Return the labeled DataFrame so downstream code can analyze or plot it
    return paired_df

# Generate the simulated cointegrated pair with default parameters
simulated_pair = simulate_cointegrated_pair()
# Create a matplotlib figure and axis to host the price plot
fig, ax = plt.subplots(figsize=(10, 5))
# Plot both assets on the same axis to visualize the tight relationship
simulated_pair[["Asset_X", "Asset_Y"]].plot(ax=ax)
# Add a descriptive title to the chart
ax.set_title("Simulated Cointegrated Pair")
# Label the y-axis to clarify that the values represent synthetic price levels
ax.set_ylabel("Synthetic Price Level")
# Remove the default x-axis label for a cleaner look
ax.set_xlabel("")
# Display the price chart
plt.show()

# Build a second figure to inspect the residual spread
fig_resid, ax_resid = plt.subplots(figsize=(10, 4))
# Plot the residual to confirm it stays stationary around zero
simulated_pair["Residual"].plot(ax=ax_resid, color="tab:orange")
# Add a title for the residual chart
ax_resid.set_title("Residual (Spread) Series")
# Label the y-axis to emphasize these are spread values
ax_resid.set_ylabel("Residual Value")
# Remove the x-axis label for consistency with the first plot
ax_resid.set_xlabel("")
# Display the residual chart
plt.show()
```





In [10]:
```
# Import statsmodels' OLS implementation for the regression
import statsmodels.api as sm
# Import the Augmented Dickey-Fuller test to examine residual stationarity
from statsmodels.tsa.stattools import adfuller

# Create a function that walks through every Engle-Granger step explicitly
def engle_granger_stepwise(series_x, series_y, maxlag=1, significance=0.05):
    # Combine both series to ensure they are aligned on the same timestamps
    paired = pd.concat([series_x, series_y], axis=1).dropna()
    # Rename the columns for readability inside the function
    paired.columns = ["X", "Y"]
    # Extract numpy arrays for the regression step
    x_values = paired["X"].values
    # Extract the dependent variable values for the regression step
    y_values = paired["Y"].values
    # Add an intercept term so the regression can solve for alpha simultaneously
    regressors = sm.add_constant(x_values)
    # Fit the ordinary least squares regression Y = alpha + beta * X + eps
    ols_model = sm.OLS(y_values, regressors).fit()
```

```
    # Report the estimated coefficients to show the long-run relationship
    print(f"OLS Coefficients: alpha={ols_model.params[0]:.4f}, beta={ols_model.para
    # Capture the residuals, which represent the spread we expect to be stationary
    residuals = y_values - ols_model.fittedvalues
    # Run the Augmented Dickey-Fuller test on the residuals to test for a unit root
    adf_stat, p_value, used_lag, n_obs, crit_vals, ic_best = adfuller(residuals, ma
    # Print the ADF statistic so we can compare it against the critical values
    print(f"ADF Statistic: {adf_stat:.4f}")
    # Show the p-value that corresponds to the statistic
    print(f"ADF P-Value: {p_value:.4f}")
    # Display the number of lags automatically chosen in the regression
    print(f"Used Lags: {used_lag}")
    # Present the critical value table that determines rejection regions
    print(f"Critical Values: {crit_vals}")
    # Decide whether to reject the null of a unit root based on the requested signi
    if p_value < significance:
        print("Result: Residuals are stationary; the series are likely cointegrated
    else:
        print("Result: Residuals are not sufficiently stationary; cointegration not

# Execute the detailed Engle-Granger procedure on the simulated assets
engle_granger_stepwise(simulated_pair["Asset_X"], simulated_pair["Asset_Y"])
```

```
OLS Coefficients: alpha=0.1987, beta=0.9383
ADF Statistic: -5.3698
ADF P-Value: 0.0000
Used Lags: 1
Critical Values: {'1%': -3.4524859843440754, '5%': -2.871288184343229, '10%': -2.571
964047565425}
Result: Residuals are stationary; the series are likely cointegrated.
```

In [11]:

```
# Import numpy linear algebra helpers for eigen decomposition
from numpy.linalg import eig, inv

# Build a function that recreates the Johansen procedure step by step
def johansen_stepwise(dataframe):
    # Drop missing values and convert to a numpy matrix
    values = dataframe.dropna().values
    # Ensure there are enough observations to form differences
    if values.shape[0] < 2:
        raise ValueError("Need at least two observations for the Johansen test.")
    # Compute the first difference ΔX_t for each series
    delta_x = values[1:] - values[:-1]
    # Capture the lagged level X_{t-1} needed in the VECM representation
    lagged_x = values[:-1]
    # Center both matrices to remove deterministic means (deterministic order 0)
    delta_x -= delta_x.mean(axis=0)
    # Apply the same centering to the lagged levels
    lagged_x -= lagged_x.mean(axis=0)
    # Store the effective sample size after differencing
    T = delta_x.shape[0]
    # Form the covariance matrix of the differenced data
    S00 = (delta_x.T @ delta_x) / T
    # Form the cross-covariance between lagged levels and differences
    S01 = (lagged_x.T @ delta_x) / T
    # Use the transpose to obtain the reverse cross-covariance
```

```
    S10 = S01.T
    # Form the covariance matrix of the lagged levels
    S11 = (lagged_x.T @ lagged_x) / T
    # Compute the matrix that encodes the generalized eigenvalue problem
    eig_matrix = inv(S11) @ S10 @ inv(S00) @ S01
    # Solve for eigenvalues and eigenvectors, which reveal cointegration strength
    eigenvalues, eigenvectors = eig(eig_matrix)
    # Keep only the real parts since the system should yield real solutions
    eigenvalues = np.real(eigenvalues)
    # Sort eigenvalues (and vectors) from largest to smallest to follow convention
    order = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[order]
    # Reorder the eigenvectors to match the sorted eigenvalues
    eigenvectors = np.real(eigenvectors[:, order])
    # Clip eigenvalues to the open interval (0, 1) for numerical stability
    eigenvalues = np.clip(eigenvalues, 0.0, 0.999999)
    # Compute the trace statistic for each rank hypothesis r
    trace_stats = [-T * np.sum(np.log(1.0 - eigenvalues[r:])) for r in range(len(ei
    # Compute the eigen (maximum) statistic for each eigenvalue
    eigen_stats = [-T * np.log(1.0 - eigen) for eigen in eigenvalues]
    # Print the resulting diagnostics so the user can compare to published critical
    print("Eigenvalues:", eigenvalues)
    print("Trace Statistics:", trace_stats)
    print("Eigen Statistics:", eigen_stats)
    print("Cointegration Vectors:")

    print(eigenvectors)

# Run the handcrafted Johansen routine on the simulated pair
johansen_stepwise(simulated_pair[["Asset_X", "Asset_Y"]])
```

```
Eigenvalues: [0.08845105 0.00785753]
Trace Statistics: [30.049066962285956, 2.3586804320545602]
Eigen Statistics: [27.690386530231397, 2.3586804320545602]
Cointegration Vectors:
[[ 0.68754093 -0.83541943]
 [-0.72614563 -0.54961294]]
```

Minimum profit Optimization

Optimization on:

1. Where to intiate a trade? (Trade Location)
2. How often do we need to trade ? (Trade Frequency)

Tighter Thresholds, less trades, large trade profits, maybe longer trades on the other side ... looser Thresholds, more trades, smaller trade profits, maybe shorter trades

Trade location optimization method:

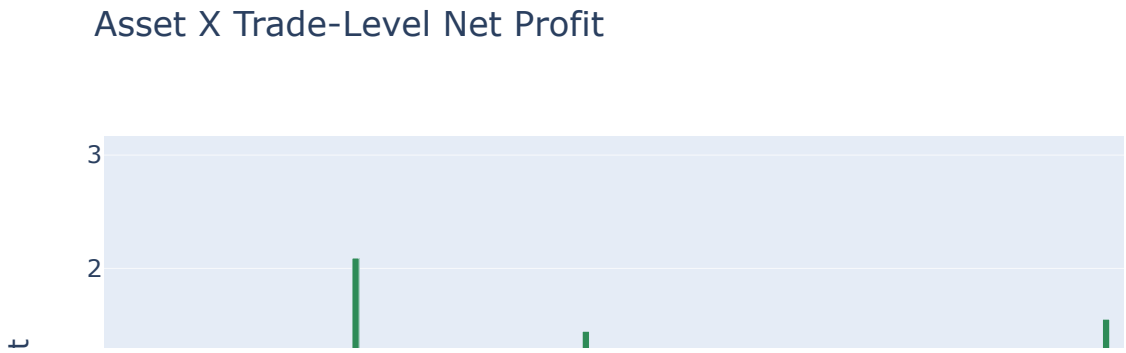
minimum profit per trade = NU

U = the upper boundary, U Must be > 0 and N = # of assets

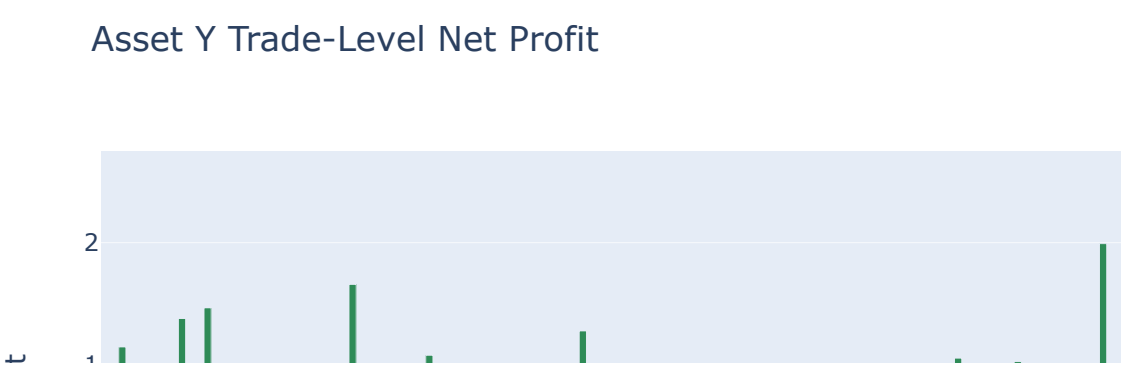
In [12]:
```
# Define a helper that inspects per-trade profits on a simple day-trading assumptio
def analyze_min_profit_per_trade(price_series, trading_cost=0.05, min_target=0.5):
    # Compute single-period price changes and drop the initial NaN
    raw_returns = price_series.diff().dropna()
    # Subtract a flat trading cost to mimic transaction fees per trade
    net_returns = raw_returns - trading_cost
    # Capture the trade index and profitability flags for plotting
    net_df = pd.DataFrame({
        "Trade": np.arange(1, len(net_returns) + 1),
        "NetReturn": net_returns.values,
    })
    # Mark trades that remained profitable after the cost adjustment
    net_df["Profitable"] = net_df["NetReturn"] > 0
    # Count how many trades stay profitable after costs
    profitable = net_df[net_df["Profitable"]]
    # Record the minimum profit achieved among all positive trades
    min_profit = profitable["NetReturn"].min() if not profitable.empty else np.nan
    # Determine what share of trades meets the requested minimum target
    hit_ratio = (profitable["NetReturn"] >= min_target).mean() if not profitable.em
    # Print a concise diagnostic summary for the series
    print(f"Total Trades: {len(net_returns)}")
    print(f"Profitable Trades After Costs: {len(profitable)}")
    print(f"Minimum Profit (Net): {min_profit:.4f}" if not np.isnan(min_profit) els
    print(f"Hit Ratio (≥ {min_target}): {hit_ratio:.2%}")
    # Return the trade-by-trade detail so we can build interactive charts
    return net_df

# Create an interactive Plotly bar chart that highlights profitable trades
def plot_trade_profit_chart(net_df, min_target, title):
    # Handle empty inputs gracefully to avoid rendering issues
    if net_df.empty:
        print("No trades available for visualization.")
        return
    # Choose colors that distinguish profitable and unprofitable trades
    colors = np.where(net_df["Profitable"], "seagreen", "crimson")
    # Build the layered chart with trade-level net returns
    fig = go.Figure()
    # Draw the bar series showing the net profit per trade
    fig.add_trace(
        go.Bar(
            x=net_df["Trade"],
            y=net_df["NetReturn"],
            marker_color=colors,
            name="Net Profit",
            hovertemplate="Trade %{x}<br>Net: %{y:.3f}<extra></extra>",
        )
    )
    # Overlay the minimum profit target as a guideline
    fig.add_trace(
        go.Scatter(
            x=net_df["Trade"],
            y=[min_target] * len(net_df),
            mode="lines",
```

```
            name="Min Target",
            line=dict(color="orange", dash="dash"),
        )
    )
    # Configure the chart aesthetics for clarity
    fig.update_layout(
        title=title,
        xaxis_title="Trade Number",
        yaxis_title="Net Profit After Cost",
        bargap=0.15,
    )
    # Display the interactive visualization
    fig.show()

# Evaluate the minimum profit landscape for each simulated asset
print("Asset X Minimum Profit Check")
net_results_x = analyze_min_profit_per_trade(simulated_pair["Asset_X"], trading_cos
plot_trade_profit_chart(net_results_x, min_target=0.5, title="Asset X Trade-Level N
print("\nAsset Y Minimum Profit Check")
net_results_y = analyze_min_profit_per_trade(simulated_pair["Asset_Y"], trading_cos
plot_trade_profit_chart(net_results_y, min_target=0.5, title="Asset Y Trade-Level N
```

```
Asset X Minimum Profit Check
Total Trades: 299
Profitable Trades After Costs: 138
Minimum Profit (Net): 0.0025
Hit Ratio (≥ 0.5): 49.28%
```




```
Asset Y Minimum Profit Check
Total Trades: 299
Profitable Trades After Costs: 135
Minimum Profit (Net): 0.0100
Hit Ratio (≥ 0.5): 54.07%
```



Trade Frequency

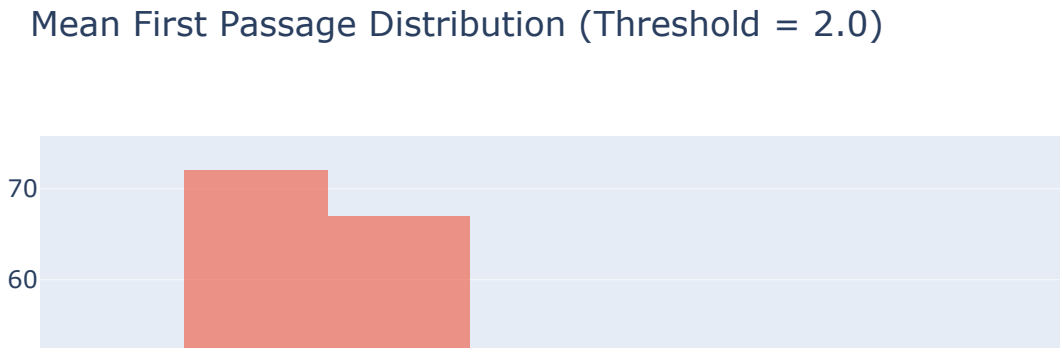
Mean First Passage Time

Assuming the spread of the cointegrated pair of assets is a stationary AR(1) process

In [13]:
```
# Create a function that estimates the mean first passage time for a price threshol
def estimate_mean_first_passage_time(price_series, threshold=2.0):
    # Convert the series to a numpy array while removing any missing data
    values = price_series.dropna().values
    # Prepare a list to collect the number of steps needed to hit the barrier each
    passage_times = []
    # Iterate over every possible starting point except the final observation
    for start_idx in range(len(values) - 1):
        # Capture the price at which we begin observing
        start_price = values[start_idx]
        # Walk forward in time until the absolute move exceeds the target threshold
        for look_ahead in range(start_idx + 1, len(values)):
            # Measure the absolute displacement from the starting price
            displacement = abs(values[look_ahead] - start_price)
            # Record the first time the displacement crosses the threshold
            if displacement >= threshold:
                passage_times.append(look_ahead - start_idx)
```

```
                break
    # Return both the mean and the individual passages for downstream visualization
    mean_passage = np.mean(passage_times) if passage_times else np.nan
    return mean_passage, passage_times

# Choose a practical threshold (in price units) for the simulated assets
threshold_value = 2.0
# Compute the mean first passage time along with the full distribution for each ass
mfpt_x, times_x = estimate_mean_first_passage_time(simulated_pair["Asset_X"], thres
mfpt_y, times_y = estimate_mean_first_passage_time(simulated_pair["Asset_Y"], thres
# Display the timing diagnostics for both series
print(f"Mean First Passage Time Asset X (|Δ| ≥ {threshold_value}): {mfpt_x:.2f} ste
print(f"Mean First Passage Time Asset Y (|Δ| ≥ {threshold_value}): {mfpt_y:.2f} ste

# Plot interactive histograms of the passage time distributions
fig = go.Figure()
# Add a histogram for Asset X if it crossed the threshold
if times_x:
    fig.add_trace(
        go.Histogram(
            x=times_x,
            name="Asset X",
            opacity=0.6,
            nbinsx=30,
            hovertemplate="Asset X<br>Steps: %{x}<extra></extra>",
        )
    )
# Add a histogram for Asset Y if it crossed the threshold
if times_y:
    fig.add_trace(
        go.Histogram(
            x=times_y,
            name="Asset Y",
            opacity=0.6,
            nbinsx=30,
            hovertemplate="Asset Y<br>Steps: %{x}<extra></extra>",
        )
    )
# Only show the figure when at least one asset generated passages
if fig.data:
    fig.update_layout(
        title=f"Mean First Passage Distribution (Threshold = {threshold_value})",
        xaxis_title="Steps Until Threshold Hit",
        yaxis_title="Count",
        barmode="overlay",
    )
    fig.show()
else:
    print("No threshold crossings occurred; histogram skipped.")
```

```
Mean First Passage Time Asset X (|Δ| ≥ 2.0): 7.79 steps
Mean First Passage Time Asset Y (|Δ| ≥ 2.0): 7.34 steps
```



Conclusion

Cointegration Takeaways

- **Upside:** Cointegration-based pairs unlock market-neutral trades by exploiting predictable mean-reversion, reducing exposure to broad market swings.
- **Downside:** Relationships can break when fundamentals diverge; regime shifts or structural breaks make historical cointegration unreliable, requiring constant monitoring.

Minimum Profit Optimization Insights

- **Upside:** Enforcing minimum net PnL per trade keeps capital focused on high-quality setups and aligns thresholds with fees/slippage realities.
- **Downside:** Strict filters reduce trade frequency, potentially missing profitable but marginal opportunities and leading to capital idleness during quiet regimes.

Real-World Applications & Next Steps

- **Execution Playbooks:** Deploy these diagnostics ahead of high-frequency pairs trading or ETF/ADR arbitrage engines to confirm spreads are still cointegrated before allocating capital.
- **Risk Systems:** Feed mean first passage estimates into intraday risk dashboards to size stop levels and expected holding periods for mean-reversion trades.
- **Project Idea:** Build a production-ready pairs screener that refreshes cointegration tests daily, ranks pairs by minimum net profit hit-rate, and publishes trade-ready signals via an internal API or dashboard.