LSCI

August 23, 2025

Tools of this integrated analysis:

1) Compare country specific LSCI data with the year-year growth scatter plot for a dual selection of preferred 'Economy (ies)'. Consider that LSCI is a measure of world integration for Maritime trade (sea trade). Data is collected to measure the scale and efficiency of each countries transport sector of shipping.

2) Make assessments of geopolitical conflict with rolling z-scores. Select a time-frame 'window' that signifies the sensitivity that the user would like the filter the visualization. For simplistic purposes lets assume the 3 time-frames to consider are '6 month- 12 month- and 24 month'. A 6 month time-frame signifies the user is placing emphasis on short-term fluctuations that stem from an event as simple as a new policy announcement. A shorter time frame is more sensitive to false positive (indication of an economic shock stirred up by something as simple as monthly deviation/ trend). Alternatively, the 24 month schedule is super-solid, yet in its resilience is more prone to false negatives (may miss some of the acute shocks that are resolved within a short time frame).

My solution is to stick with the 12 month window for consistency unless logically you prefer to place emphasis on strictly acute or chronic shocks with the 6 month or 24 month time-frames respectively. The slider is continuous allowing from a 6-24 month time-frame.

3) Used a PCA (Principle Component Analysis) for data driven weighting of 4 LSCI statistical components. This predictive PCA weighting allowed me to create a more advanced volatility model conjoined with the ACLED 'Conflict Index.'

[1]:
```
# LSCI Data Analysis - Package Initialization



import pandas as pd
import numpy as np



import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots



import ipywidgets as widgets
```

```
from IPython.display import display, clear_output


from datetime import datetime, timedelta
import warnings
warnings.filterwarnings('ignore')


plt.style.use('default')
sns.set_palette("husl")

# Configure pandas display options
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 100)

# Plotly configuration for better interactivity
import plotly.io as pio
pio.renderers.default = "notebook_connected"

# Ensure that each line is fully output to properly use the models below
print("All packages loaded successfully!")
print("Ready for LSCI data analysis")
print("Dual-country comparison tools initialized")
print("Interactive visualization capabilities loaded")
```

```
All packages loaded successfully!
Ready for LSCI data analysis
Dual-country comparison tools initialized
Interactive visualization capabilities loaded
```

[2]:

```
# Main data-frame, includes time-series data from 2006-2025, measuring␣
↪individual countries efficiency, and scale of maritime trade/ sea transport.
df = pd.read_csv('LSCI_Main_SQLite.csv')

print("Main dataframe succesfully uploaded")
```

```
Main dataframe succesfully uploaded
```

[4]:

```
# Data Cleaning and Date Conversion's (Claude.AI)

# Create a copy to work with
df_clean = df.copy()

# Convert MonthLabel to 'datetime'
# Handle mixed formats: "Feb. 2006" and "May-24" styles of the MonthLabel␣
↪column
def parse_mixed_dates(date_str):
```

```python
    """Parse dates in mixed formats"""
    date_str = str(date_str).strip()

    try:
        # Handle "Feb. 2006" format
        if '.' in date_str and len(date_str.split()[-1]) == 4:
            return pd.to_datetime(date_str, format='%b. %Y')

        # Handle "May-24" format (assuming 20xx years)
        elif '-' in date_str and len(date_str.split('-')[-1]) == 2:
            month, year = date_str.split('-')
            # Convert 2-digit year to 4-digit (assumes 2000s)
            full_year = f"20{year}"
            return pd.to_datetime(f"{month} {full_year}", format='%b %Y')

        # Try standard pandas parsing as fallback
        else:
            return pd.to_datetime(date_str)

    except:
        return pd.NaT

# Apply the custom parsing function
print(" Converting mixed date formats...")
df_clean['Date'] = df_clean['MonthLabel'].apply(parse_mixed_dates)

# Check conversion results
successful_conversions = df_clean['Date'].notna().sum()
failed_conversions = df_clean['Date'].isna().sum()

print(f" Successfully converted: {successful_conversions} dates")
if failed_conversions > 0:
    print(f" Failed to convert: {failed_conversions} dates")
    print("Sample failed conversions:")
    failed_samples = df_clean[df_clean['Date'].isna()]['MonthLabel'].head()
    for sample in failed_samples:
        print(f"   - '{sample}'")

# Remove any rows with missing LSCI scores or economy labels (pre cleaned in␣
 ↪SQLite)
initial_rows = len(df_clean)
df_clean = df_clean.dropna(subset=['LSCI_Score', 'EconomyLabel'])
final_rows = len(df_clean)

print(f" Removed {initial_rows - final_rows} rows with missing data")
print(f" Final dataset: {final_rows} rows")
```

```python
# Sort by date for proper time series
df_clean = df_clean.sort_values(['EconomyLabel', 'Date'])

# Check date range
if 'Date' in df_clean.columns:
    print(f"\n Date Range: {df_clean['Date'].min} to {df_clean['Date'].max}")

    # Create year-month for easier filtering if needed
    df_clean['YearMonth'] = df_clean['Date'].dt.to_period('M')

# Display cleaned data info
print(f"\n Cleaned Dataset Preview:")
display(df_clean[['EconomyLabel', 'Date', 'LSCI_Score']].head())

print(f"\n Total Economies: {df_clean['EconomyLabel'].nunique()}")
print(f" Total Data Points:  {len(df_clean)}")
```

```
 Converting mixed date formats…
 Successfully converted: 17396 dates
 Removed 0 rows with missing data
 Final dataset: 17396 rows

 Date Range: <bound method Series.min of 521     2006-11-01
695     2007-02-01
868     2007-05-01
1043    2007-08-01
1218    2007-11-01
           …
16679   2025-02-01
16858   2025-03-01
17037   2025-04-01
17216   2025-05-01
17395   2025-06-01
Name: Date, Length: 17396, dtype: datetime64[ns]> to <bound method Series.max of
521     2006-11-01
695     2007-02-01
868     2007-05-01
1043    2007-08-01
1218    2007-11-01
           …
16679   2025-02-01
16858   2025-03-01
17037   2025-04-01
17216   2025-05-01
17395   2025-06-01
Name: Date, Length: 17396, dtype: datetime64[ns]>
```

```
Cleaned Dataset Preview:

     EconomyLabel       Date  LSCI_Score
521       Albania 2006-11-01     5.00790
695       Albania 2007-02-01     5.47757
868       Albania 2007-05-01     5.47757
1043      Albania 2007-08-01     5.03685
1218      Albania 2007-11-01     5.19215


Total Economies: 191
Total Data Points:  17396
```

[5]:

```python
# Cell 4 (BASIC version of dual plot): Interactive Widgets with Proper␣
↪Integration- an elementary visualization tool showcasing a dual-comparison␣
↪between two selected countries LSCI time-series
# For general LSCI comparisons- a base for integration and further analysis␣
↪(necessary to run the ADVANCED model)

from IPython.display import display
import ipywidgets as widgets

# Get sorted list of all available economies
available_economies = sorted(df_clean['EconomyLabel'].unique())

print(f"Found {len(available_economies)} economies in dataset")

# Dropdown for first country
country1_dropdown = widgets.Dropdown(
    options=available_economies,
    value=available_economies[0],
    description='Country 1:',
    style={'description_width': '100px'},
    layout=widgets.Layout(width='300px')
)

# Dropdown for second country
country2_dropdown = widgets.Dropdown(
    options=available_economies,
    value=available_economies[1] if len(available_economies) > 1 else␣
↪available_economies[0],
    description='Country 2:',
    style={'description_width': '100px'},
    layout=widgets.Layout(width='300px')
)

# Plot type radio buttons
plot_type = widgets.RadioButtons(
```

```
    options=['Line Plot', 'Scatter Plot', 'Both'],
    value='Line Plot',
    description='Plot Type:',
    style={'description_width': '100px'}
)

# Update button with purple styling and centered position
update_button = widgets.Button(
    description='Update Plot',
    layout=widgets.Layout(width='200px', height='40px'),
    style={'button_color': 'purple', 'font_weight': 'bold'}
)

# This centers the button
update_button_box = widgets.HBox([update_button])
update_button_box.layout.justify_content = 'center'

# Output area for plot
output_area = widgets.Output()

# Display layout
print("Interactive Controls:")
control_box = widgets.VBox([
    widgets.HBox([country1_dropdown, country2_dropdown]),
    plot_type,
    update_button_box  # button is now centered & below plot type
])

display(control_box)
display(output_area)

print("Continue to create the functioning backend...")
print("OR advance to the dual model with heightened features")
```

```
Found 191 economies in dataset
Interactive Controls:

VBox(children=(HBox(children=(Dropdown(description='Country 1:',␣
↪layout=Layout(width='300px'), options=('Alban…

Output()

Continue to create the functioning backend…
OR advance to the dual model with heightened features
```

[6]:
```
# Cell 5: Matplotlib Version

def create_dual_country_plot_matplotlib(country1, country2, plot_style='Line␣
↪Plot'):
```

```
    """
    Create dual-country comparison plot using matplotlib (more reliable)
    """
    try:
        print(f"Creating matplotlib plot for {country1} vs {country2}")

        # Filter and clean data
        data1 = df_clean[df_clean['EconomyLabel'] == country1].copy()
        data2 = df_clean[df_clean['EconomyLabel'] == country2].copy()

        data1 = data1.dropna(subset=['Date', 'LSCI_Score']).sort_values('Date')
        data2 = data2.dropna(subset=['Date', 'LSCI_Score']).sort_values('Date')

        if len(data1) == 0 or len(data2) == 0:
            print(f"  No data available for selected countries")
            return

        # Create matplotlib figure
        fig, ax = plt.subplots(figsize=(12, 8))

        # Plot based on style
        if plot_style in ['Line Plot', 'Both']:
            ax.plot(data1['Date'], data1['LSCI_Score'],
                   marker='o', linewidth=2, markersize=6, label=country1, alpha=0.8)
            ax.plot(data2['Date'], data2['LSCI_Score'],
                   marker='s', linewidth=2, markersize=6, label=country2, alpha=0.8)

        if plot_style in ['Scatter Plot', 'Both']:
            ax.scatter(data1['Date'], data1['LSCI_Score'],
                      s=50, alpha=0.7, label=f'{country1} (Scatter)' if plot_style == 'Both' else country1)
            ax.scatter(data2['Date'], data2['LSCI_Score'],
                      s=50, alpha=0.7, marker='^', label=f'{country2} (Scatter)' if plot_style == 'Both' else country2)

        # Customize plot
        ax.set_title(f'LSCI Score Comparison: {country1} vs {country2}', fontsize=16, fontweight='bold')
        ax.set_xlabel('Date', fontsize=12)
        ax.set_ylabel('LSCI Score', fontsize=12)
        ax.legend(fontsize=11)
        ax.grid(True, alpha=0.3)

        # Format dates on x-axis
        fig.autofmt_xdate()
```

```
        # Add some styling
        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)

        plt.tight_layout()
        plt.show()

        return True

    except Exception as e:
        print(f" Error creating matplotlib plot: {e}")
        return False

# Updated button click handler for matplotlib
def on_button_click_matplotlib(b):
    """Handle button click with matplotlib plotting"""
    with output_area:
        clear_output(wait=True)

        try:
            c1 = country1_dropdown.value
            c2 = country2_dropdown.value
            plot_style = plot_type.value

            print(f"Creating {plot_style} for {c1} vs {c2}")

            # Create matplotlib plot
            success = create_dual_country_plot_matplotlib(c1, c2, plot_style)

            if success:
                # Display summary statistics
                data1 = df_clean[df_clean['EconomyLabel'] == c1]['LSCI_Score']
                data2 = df_clean[df_clean['EconomyLabel'] == c2]['LSCI_Score']

                print(f"\n Summary Statistics:")
                print(f"{c1}: Mean={data1.mean():.2f}, Std={data1.std():.2f}, Range=[{data1.min():.2f}, {data1.max():.2f}]")
                print(f"{c2}: Mean={data2.mean():.2f}, Std={data2.std():.2f}, Range=[{data2.min():.2f}, {data2.max():.2f}]")
                print("  Plot created successfully!")

        except Exception as e:
            print(f" Error: {e}")

# Disconnect old button and connect new one
try:
```

```
    # Remove old callback
    update_button._click_handlers.callbacks.clear()
except:
    pass

# Connect to new matplotlib function
update_button.on_click(on_button_click_matplotlib)

print(" Matplotlib backup plotting system ready!")
print(" Click 'Update Plot' to generate matplotlib visualization")
print(" This version uses matplotlib instead of Plotly for better␣
 ↪compatibility")
```

```
 Matplotlib backup plotting system ready!
 Click 'Update Plot' to generate matplotlib visualization
 This version uses matplotlib instead of Plotly for better compatibility
```

[7]:
```
# Cell 6: Display Interactive Interface and Test (conclusion of backend)
#

print("Basic Interactive LSCI Analysis Dashboard")
print("=" * 50)

# Show current widget status
print(f" Available Countries: {len(available_economies)}")
print(f" Current Selection: {country1_dropdown.value} vs {country2_dropdown.
 ↪value}")
print(f" Plot Type: {plot_type.value}")

print("\n  Instructions:")
print("   1. Use the dropdowns after Cell 4 to select two countries")
print("   2. Choose your preferred plot type (Line, Scatter, or Both)")
print("   3. Click the 'Update Plot' button to generate/refresh the␣
 ↪visualization")
print("   4. The plot will appear in the output area below, basic summary␣
 ↪statistics displayed below the chart")

print("\n Click 'Update Plot' to begin.")
```

```
Basic Interactive LSCI Analysis Dashboard
==================================================
 Available Countries: 191
 Current Selection: China vs Angola
 Plot Type: Line Plot

  Instructions:
    1. Use the dropdowns after Cell 4 to select two countries
```

2. Choose your preferred plot type (Line, Scatter, or Both)
3. Click the 'Update Plot' button to generate/refresh the visualization
4. The plot will appear in the output area below, basic summary statistics displayed below the chart

Click 'Update Plot' to begin.

[9]:
```
# Cell 7: Enhanced Visualization with Regression Lines (ADVANCED dual plot)
# Covers both the front and back ends in this cell, more advanced summary␣
↪statistics, and a visual regression line for a quick sense of trends

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import r2_score

#dual import
import numpy as np

def create_enhanced_dual_country_plot(country1, country2, plot_style='Scatter␣
↪Plot', regression_type='Linear'):
    """
    Enhanced dual-country comparison with regression analysis
    """
    try:
        print(f" Creating enhanced plot: {country1} vs {country2}␣
↪({plot_style})")

        # Filter and clean data
        data1 = df_clean[df_clean['EconomyLabel'] == country1].copy()
        data2 = df_clean[df_clean['EconomyLabel'] == country2].copy()

        data1 = data1.dropna(subset=['Date', 'LSCI_Score']).sort_values('Date')
        data2 = data2.dropna(subset=['Date', 'LSCI_Score']).sort_values('Date')

        if len(data1) < 3 or len(data2) < 3:
            print(f"  Need at least 3 data points for regression analysis")
            return False

        # Convert dates to numeric for regression
        data1['date_numeric'] = pd.to_numeric(data1['Date'])
        data2['date_numeric'] = pd.to_numeric(data2['Date'])

        # Create figure with larger size for enhanced visualization
        fig, ax = plt.subplots(figsize=(14, 10))

        # Define colors for consistency
        color1 = '#1f77b4'  # Blue
```

```
        color2 = '#ff7f0e'  # Orange

        # Plot scatter points
        if plot_style in ['Scatter Plot', 'Both']:
            scatter1 = ax.scatter(data1['Date'], data1['LSCI_Score'],
                                s=80, alpha=0.7, color=color1, label=country1,
                                edgecolors='white', linewidth=1, zorder=5)
            scatter2 = ax.scatter(data2['Date'], data2['LSCI_Score'],
                                s=80, alpha=0.7, color=color2, marker='^', label=country2,
                                edgecolors='white', linewidth=1, zorder=5)

        # Plot line plots
        if plot_style in ['Line Plot', 'Both']:
            ax.plot(data1['Date'], data1['LSCI_Score'],
                   color=color1, linewidth=2.5, alpha=0.8, label=f'{country1} (Line)')
            ax.plot(data2['Date'], data2['LSCI_Score'],
                   color=color2, linewidth=2.5, alpha=0.8, label=f'{country2} (Line)')

        # Add regression lines for scatter plots
        if plot_style in ['Scatter Plot', 'Both']:
            # Country 1 regression
            X1 = data1['date_numeric'].values.reshape(-1, 1)
            y1 = data1['LSCI_Score'].values

            if regression_type == 'Linear':
                reg1 = LinearRegression().fit(X1, y1)
                y1_pred = reg1.predict(X1)
                r2_1 = r2_score(y1, y1_pred)
            else:  # Polynomial
                poly_features = PolynomialFeatures(degree=2)
                X1_poly = poly_features.fit_transform(X1)
                reg1 = LinearRegression().fit(X1_poly, y1)
                y1_pred = reg1.predict(X1_poly)
                r2_1 = r2_score(y1, y1_pred)

            ax.plot(data1['Date'], y1_pred,
                   color=color1, linestyle='--', linewidth=3, alpha=0.9,
                   label=f'{country1} Trend (R²={r2_1:.3f})', zorder=4)

            # Country 2 regression
            X2 = data2['date_numeric'].values.reshape(-1, 1)
            y2 = data2['LSCI_Score'].values

            if regression_type == 'Linear':
```

```
                reg2 = LinearRegression().fit(X2, y2)
                y2_pred = reg2.predict(X2)
                r2_2 = r2_score(y2, y2_pred)
            else:  # Polynomial
                poly_features = PolynomialFeatures(degree=2)
                X2_poly = poly_features.fit_transform(X2)
                reg2 = LinearRegression().fit(X2_poly, y2)
                y2_pred = reg2.predict(X2_poly)
                r2_2 = r2_score(y2, y2_pred)

            ax.plot(data2['Date'], y2_pred,
                   color=color2, linestyle='--', linewidth=3, alpha=0.9,
                   label=f'{country2} Trend (R²={r2_2:.3f})', zorder=4)

            # Calculate and display trend statistics with normalization
            slope1_raw = reg1.coef_[0] if regression_type == 'Linear' else None
            slope2_raw = reg2.coef_[0] if regression_type == 'Linear' else None

            print(f"\n Regression Analysis:")
            if regression_type == 'Linear':
                # Calculate time span for normalization
                time_span_days1 = (data1['Date'].max() - data1['Date'].min()).
↪days
                time_span_days2 = (data2['Date'].max() - data2['Date'].min()).
↪days
                time_span_years1 = time_span_days1 / 365.25
                time_span_years2 = time_span_days2 / 365.25

                # Normalize slopes (change per year)
                slope1_per_year = slope1_raw * 365.25 * 24 * 60 * 60 * 1e9  #␣
↪Convert from nanoseconds to years
                slope2_per_year = slope2_raw * 365.25 * 24 * 60 * 60 * 1e9

                # Calculate total change over period
                total_change1 = slope1_per_year * time_span_years1
                total_change2 = slope2_per_year * time_span_years2

                # Display multiple slope representations
                print(f"{country1}:")
                print(f"  • Raw slope: {slope1_raw:.2e}")
                print(f"  • Change per year: {slope1_per_year:.4f}")
                print(f"  • Total change over {time_span_years1:.1f} years:␣
↪{total_change1:.3f}")
                print(f"  • R² fit: {r2_1:.3f}")

                print(f"{country2}:")
                print(f"  • Raw slope: {slope2_raw:.2e}")
```

```
                print(f"  • Change per year: {slope2_per_year:.4f}")
                print(f"  • Total change over {time_span_years2:.1f} years: {total_change2:.3f}")
                print(f"  • R² fit: {r2_2:.3f}")

                # Enhanced trend interpretation
                threshold = 0.001  # Threshold for meaningful change

                if abs(slope1_per_year) < threshold:
                    trend1 = "→ Stable (minimal change)"
                elif slope1_per_year > 0:
                    trend1 = f"  Improving (+{slope1_per_year:.4f}/year)"
                else:
                    trend1 = f"  Declining ({slope1_per_year:.4f}/year)"

                if abs(slope2_per_year) < threshold:
                    trend2 = "→ Stable (minimal change)"
                elif slope2_per_year > 0:
                    trend2 = f"  Improving (+{slope2_per_year:.4f}/year)"
                else:
                    trend2 = f"  Declining ({slope2_per_year:.4f}/year)"

                print(f"\n  Trend Interpretation:")
                print(f"{country1}: {trend1}")
                print(f"{country2}: {trend2}")

                # Comparative slope analysis
                if abs(slope1_per_year) > abs(slope2_per_year):
                    faster_change = country1
                    change_diff = abs(abs(slope1_per_year) - abs(slope2_per_year))
                else:
                    faster_change = country2
                    change_diff = abs(abs(slope1_per_year) - abs(slope2_per_year))

                print(f"\n  Rate Comparison:")
                print(f"{faster_change} has faster rate of change by {change_diff:.4f} points/year")

            else:
                print(f"{country1}: R²={r2_1:.3f} (Polynomial fit)")
                print(f"{country2}: R²={r2_2:.3f} (Polynomial fit)")

        # Enhanced styling
        ax.set_title(f'Enhanced LSCI Analysis: {country1} vs {country2}',
                    fontsize=18, fontweight='bold', pad=20)
```

```
        ax.set_xlabel('Date', fontsize=14, fontweight='bold')
        ax.set_ylabel('LSCI Score', fontsize=14, fontweight='bold')

        # Improved legend
        legend = ax.legend(fontsize=11, frameon=True, fancybox=True,␣
↪shadow=True,
                           bbox_to_anchor=(1.05, 1), loc='upper left')
        legend.get_frame().set_facecolor('white')
        legend.get_frame().set_alpha(0.9)

        # Enhanced grid
        ax.grid(True, alpha=0.3, linestyle='-', linewidth=0.5)
        ax.set_facecolor('#f8f9fa')

        # Remove top and right spines
        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)
        ax.spines['left'].set_linewidth(1.5)
        ax.spines['bottom'].set_linewidth(1.5)

        # Format dates
        fig.autofmt_xdate()

        plt.tight_layout()
        plt.show()

        return True

    except Exception as e:
        print(f" Error creating enhanced plot: {e}")
        import traceback
        traceback.print_exc()
        return False

# Create enhanced widgets
print(" Enhanced Controls:")

# Regression type selector
regression_type_widget = widgets.RadioButtons(
    options=['Linear', 'Polynomial'],
    value='Linear',
    description='Regression:',
    style={'description_width': '100px'}
)

# Enhanced plot type with regression info
enhanced_plot_type = widgets.RadioButtons(
```

```
    options=['Scatter Plot', 'Line Plot', 'Both'],
    value='Scatter Plot',
    description='Plot Type:',
    style={'description_width': '100px'}
)

# Enhanced update button
enhanced_update_button = widgets.Button(
    description=' Enhanced Plot',
    button_style='success',
    layout=widgets.Layout(width='200px', height='40px')
)

# Display enhanced controls
enhanced_controls = widgets.VBox([
    widgets.HBox([country1_dropdown, country2_dropdown]),
    widgets.HBox([enhanced_plot_type, regression_type_widget]),
    enhanced_update_button
])

display(enhanced_controls)

# Enhanced button click handler
def on_enhanced_button_click(b):
    """Handle enhanced plot generation"""
    with output_area:
        clear_output(wait=True)

        try:
            c1 = country1_dropdown.value
            c2 = country2_dropdown.value
            plot_style = enhanced_plot_type.value
            reg_type = regression_type_widget.value

            print(f" Generating enhanced visualization...")
            print(f"Countries: {c1} vs {c2}")
            print(f"Style: {plot_style} with {reg_type} regression")

            success = create_enhanced_dual_country_plot(c1, c2, plot_style,␣
↪reg_type)

            if success:
                # Enhanced statistics
                data1 = df_clean[df_clean['EconomyLabel'] == c1]['LSCI_Score']
                data2 = df_clean[df_clean['EconomyLabel'] == c2]['LSCI_Score']

                print(f"\n Enhanced Statistics:")
```

```
            print(f"{c1}:  ={data1.mean():.3f},  ={data1.std():.3f},␣
↪Range=[{data1.min():.2f}, {data1.max():.2f}]")
            print(f"{c2}:  ={data2.mean():.3f},  ={data2.std():.3f},␣
↪Range=[{data2.min():.2f}, {data2.max():.2f}]")

            # Comparative analysis
            mean_diff = abs(data1.mean() - data2.mean())
            print(f"\n Comparative Analysis:")
            print(f"Mean difference: {mean_diff:.3f}")
            print(f"Better performer: {c1 if data1.mean() > data2.mean()␣
↪else c2}")

        except Exception as e:
            print(f"  Enhanced plotting error: {e}")

# Connect enhanced button
enhanced_update_button.on_click(on_enhanced_button_click)

print("  Enhanced visualization system ready!")
print("Features Include: Regression lines, trend analysis, enhanced styling")
print("Click ' Enhanced Plot' to generate advanced visualization")

print("--R-squared values are absolute--")
```

```
 Enhanced Controls:

VBox(children=(HBox(children=(Dropdown(description='Country 1:', index=28,␣
↪layout=Layout(width='300px'), optio…

 Enhanced visualization system ready!
Features Include: Regression lines, trend analysis, enhanced styling
Click ' Enhanced Plot' to generate advanced visualization
R-squared values are absolute
```

[10]:
```
# Cell 8: Rolling Z-Score Economic Shock Analysis - FIXED PERCENTAGES

from scipy import stats
import warnings
warnings.filterwarnings('ignore')

def calculate_global_shock_threshold():
    """
    Calculate shock threshold based on all countries' data
    """

    # Calculate rolling z-scores for all countries
    all_z_scores = []
    window_size = 12  # 12-month rolling window
```

```python
    for country in df_clean['EconomyLabel'].unique():
        country_data = df_clean[df_clean['EconomyLabel'] == country].copy()
        country_data = country_data.dropna(subset=['Date', 'LSCI_Score']).sort_values('Date')

        if len(country_data) >= window_size * 2:  # Need enough data for rolling calc
            # Calculate rolling mean and std
            country_data['rolling_mean'] = country_data['LSCI_Score'].rolling(window=window_size, center=True).mean()
            country_data['rolling_std'] = country_data['LSCI_Score'].rolling(window=window_size, center=True).std()

            # Calculate z-scores
            country_data['z_score'] = (country_data['LSCI_Score'] - country_data['rolling_mean']) / country_data['rolling_std']

            # Collect valid z-scores
            valid_z_scores = country_data['z_score'].dropna()
            all_z_scores.extend(valid_z_scores.tolist())

    all_z_scores = np.array(all_z_scores)

    # Calculate threshold statistics
    z_mean = np.mean(all_z_scores)
    z_std = np.std(all_z_scores)
    z_95 = np.percentile(all_z_scores, 95)
    z_5 = np.percentile(all_z_scores, 5)

    # Set shock thresholds (more conservative than pure statistical)
    moderate_shock_threshold = 1.5  # 1.5 standard deviations
    severe_shock_threshold = 2.0    # 2.0 standard deviations
    extreme_shock_threshold = 2.5   # 2.5 standard deviations

    print(f"Global Z-Score Distribution:")
    print(f"   Mean: {z_mean:.3f}")
    print(f"   Std Dev: {z_std:.3f}")
    print(f"   5th percentile: {z_5:.3f}")
    print(f"   95th percentile: {z_95:.3f}")

    print(f"\n Economic Shock Thresholds:")
    print(f"   Moderate shock: |z| > {moderate_shock_threshold}")
    print(f"   Severe shock: |z| > {severe_shock_threshold}")
    print(f"   Extreme shock: |z| > {extreme_shock_threshold}")

    return {
```

```python
        'moderate': moderate_shock_threshold,
        'severe': severe_shock_threshold,
        'extreme': extreme_shock_threshold,
        'global_stats': {
            'mean': z_mean,
            'std': z_std,
            'p5': z_5,
            'p95': z_95
        }
    }

def analyze_country_shocks(country_name, window_size=12, shock_thresholds=None):
    """
    Analyze economic shocks for a specific country using rolling z-scores
    """
    try:
        print(f" Analyzing economic shocks for: {country_name}")

        # Filter country data
        country_data = df_clean[df_clean['EconomyLabel'] == country_name].copy()
        country_data = country_data.dropna(subset=['Date', 'LSCI_Score']).
 ↪sort_values('Date')

        if len(country_data) < window_size * 2:
            print(f"  Insufficient data for {country_name} (need at least␣
 ↪{window_size * 2} points)")
            return None

        print(f" Data period: {country_data['Date'].min()} to␣
 ↪{country_data['Date'].max()}")
        print(f" Total data points: {len(country_data)}")

        # Calculate rolling statistics
        country_data['rolling_mean'] = country_data['LSCI_Score'].
 ↪rolling(window=window_size, center=True).mean()
        country_data['rolling_std'] = country_data['LSCI_Score'].
 ↪rolling(window=window_size, center=True).std()

        # Calculate z-scores
        country_data['z_score'] = (country_data['LSCI_Score'] -␣
 ↪country_data['rolling_mean']) / country_data['rolling_std']

        # Identify shock periods
        if shock_thresholds is None:
            shock_thresholds = calculate_global_shock_threshold()
```

```
        country_data['shock_level'] = 'Normal'
        country_data.loc[abs(country_data['z_score']) >␣
↪shock_thresholds['moderate'], 'shock_level'] = 'Moderate'
        country_data.loc[abs(country_data['z_score']) >␣
↪shock_thresholds['severe'], 'shock_level'] = 'Severe'
        country_data.loc[abs(country_data['z_score']) >␣
↪shock_thresholds['extreme'], 'shock_level'] = 'Extreme'

        # Create visualization
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(15, 12))

        # Top plot: LSCI Score with shock highlighting
        ax1.plot(country_data['Date'], country_data['LSCI_Score'],
                color='black', linewidth=1.5, label='LSCI Score', zorder=3)

        # Add rolling mean
        ax1.plot(country_data['Date'], country_data['rolling_mean'],
                color='blue', linewidth=2, alpha=0.7,␣
↪label=f'{window_size}-month Rolling Mean', zorder=2)

        # Highlight shock periods with different colors
        shock_colors = {
            'Moderate': '#FFA500',  # Orange
            'Severe': '#FF6B35',    # Red-Orange
            'Extreme': '#DC143C'    # Dark Red
        }

        for shock_type, color in shock_colors.items():
            shock_mask = country_data['shock_level'] == shock_type
            if shock_mask.any():
                ax1.scatter(country_data.loc[shock_mask, 'Date'],
                           country_data.loc[shock_mask, 'LSCI_Score'],
                           color=color, s=60, alpha=0.8, label=f'{shock_type}␣
↪Shock',
                           zorder=4, edgecolors='white', linewidth=1)

        ax1.set_title(f'Economic Shock Analysis: {country_name}', fontsize=16,␣
↪fontweight='bold')
        ax1.set_ylabel('LSCI Score', fontsize=12, fontweight='bold')
        ax1.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
        ax1.grid(True, alpha=0.3)

        # Bottom plot: Z-Score with shock thresholds
        ax2.plot(country_data['Date'], country_data['z_score'],
                color='black', linewidth=1.5, label='Rolling Z-Score')
```

```python
    # Add threshold lines
    ax2.axhline(y=shock_thresholds['moderate'], color='orange', linestyle='--', alpha=0.7, label='Moderate Threshold')
    ax2.axhline(y=-shock_thresholds['moderate'], color='orange', linestyle='--', alpha=0.7)
    ax2.axhline(y=shock_thresholds['severe'], color='red', linestyle='--', alpha=0.7, label='Severe Threshold')
    ax2.axhline(y=-shock_thresholds['severe'], color='red', linestyle='--', alpha=0.7)
    ax2.axhline(y=0, color='gray', linestyle='-', alpha=0.5, label='Mean')

    # Fill shock regions
    ax2.fill_between(country_data['Date'], -shock_thresholds['moderate'], shock_thresholds['moderate'],
                    alpha=0.1, color='green', label='Normal Range')
    ax2.fill_between(country_data['Date'], shock_thresholds['moderate'], shock_thresholds['severe'],
                    alpha=0.1, color='orange')
    ax2.fill_between(country_data['Date'], -shock_thresholds['severe'], -shock_thresholds['moderate'],
                    alpha=0.1, color='orange')
    ax2.fill_between(country_data['Date'], shock_thresholds['severe'], 10,
                    alpha=0.1, color='red')
    ax2.fill_between(country_data['Date'], -10, -shock_thresholds['severe'],
                    alpha=0.1, color='red')

    ax2.set_xlabel('Date', fontsize=12, fontweight='bold')
    ax2.set_ylabel('Z-Score', fontsize=12, fontweight='bold')
    ax2.set_ylim(-4, 4)
    ax2.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    # FIXED PERCENTAGE CALCULATION
    # Only count rows where z_score is not NaN (actual analysis points)
    valid_analysis_data = country_data.dropna(subset=['z_score'])
    shock_summary = valid_analysis_data['shock_level'].value_counts()
    total_valid_points = len(valid_analysis_data)

    print(f"\n Shock Period Summary for {country_name}:")
    print(f"   Analysis based on {total_valid_points} valid data points")
    print(f"   Normal periods: {shock_summary.get('Normal', 0)} ({shock_summary.get('Normal', 0)/total_valid_points*100:.1f}%)")
```

```
        print(f"   Moderate shocks: {shock_summary.get('Moderate', 0)}␣
↪({shock_summary.get('Moderate', 0)/total_valid_points*100:.1f}%)")
        print(f"   Severe shocks: {shock_summary.get('Severe', 0)}␣
↪({shock_summary.get('Severe', 0)/total_valid_points*100:.1f}%)")
        print(f"   Extreme shocks: {shock_summary.get('Extreme', 0)}␣
↪({shock_summary.get('Extreme', 0)/total_valid_points*100:.1f}%)")

        # Verification: Check that percentages sum to 100%
        total_percentage = sum([
            shock_summary.get('Normal', 0)/total_valid_points*100,
            shock_summary.get('Moderate', 0)/total_valid_points*100,
            shock_summary.get('Severe', 0)/total_valid_points*100,
            shock_summary.get('Extreme', 0)/total_valid_points*100
        ])
        print(f"    Total: {total_percentage:.1f}% (should equal 100.0%)")

        # Identify specific shock periods
        shock_periods = valid_analysis_data[valid_analysis_data['shock_level'] !
↪= 'Normal'].copy()
        if len(shock_periods) > 0:
            print(f"\n Identified Shock Periods:")
            for _, period in shock_periods.iterrows():
                direction = "positive" if period['z_score'] > 0 else "negative"
                print(f"   {period['Date'].strftime('%Y-%m')}:␣
↪{period['shock_level']} {direction} shock (z={period['z_score']:.2f})")
        else:
            print(f"\n  No significant shock periods identified for␣
↪{country_name}")

        return country_data

    except Exception as e:
        print(f"  Error analyzing shocks for {country_name}: {e}")
        import traceback
        traceback.print_exc()
        return None

# Create single country selector widget
print(" Economic Shock Analysis Controls:")

single_country_dropdown = widgets.Dropdown(
    options=sorted(df_clean['EconomyLabel'].unique()),
    value=sorted(df_clean['EconomyLabel'].unique())[0],
    description='Select Country:',
    style={'description_width': '120px'},
    layout=widgets.Layout(width='400px')
```

```
)

window_size_slider = widgets.IntSlider(
    value=12,
    min=6,
    max=24,
    step=1,
    description='Window Size (months):',
    style={'description_width': '140px'},
    layout=widgets.Layout(width='400px')
)

shock_analysis_button = widgets.Button(
    description='  Analyze Economic Shocks',
    button_style='warning',
    layout=widgets.Layout(width='250px', height='40px')
)

# Display controls
shock_controls = widgets.VBox([
    single_country_dropdown,
    window_size_slider,
    shock_analysis_button
])

display(shock_controls)

# Create output area for shock analysis
shock_output_area = widgets.Output()
display(shock_output_area)

# Button click handler
def on_shock_analysis_click(b):
    """Handle shock analysis button click"""
    with shock_output_area:
        clear_output(wait=True)

        try:
            country = single_country_dropdown.value
            window = window_size_slider.value

            print(f" Initiating shock analysis...")

            # Calculate global thresholds first
            thresholds = calculate_global_shock_threshold()

            # Analyze specific country
```

```
            result = analyze_country_shocks(country, window, thresholds)

            if result is not None:
                print(f"\n  Economic shock analysis completed for {country}")

        except Exception as e:
            print(f"  Shock analysis error: {e}")

# Connect button
shock_analysis_button.on_click(on_shock_analysis_click)
```

```
 Economic Shock Analysis Controls:
```

```
VBox(children=(Dropdown(description='Select Country:',␣
↪layout=Layout(width='400px'), options=('Albania', 'Alge…
```

```
Output()
```

VOLATILITY MODELS (Beta versions, testing and finally the polished volatility mapping including rolling st dev, and GARCH)

1. PCA for LSCI_volatility- with fuzzy matching
2. GARCH model

[11]:
```
#Volatility model including rolling z score logic

import pandas as pd
import numpy as np
from datetime import datetime
import re
from sklearn.preprocessing import MinMaxScaler
import warnings
warnings.filterwarnings('ignore')

# Load the datasets
lsci_df = pd.read_csv('LSCI_Main_SQLite.csv')
acled_df = pd.read_csv('ACLED_Conflict_Index.csv')

print("Dataset shapes:")
print(f"LSCI Data: {lsci_df.shape}")
print(f"ACLED Data: {acled_df.shape}")

# Function to parse and standardize month labels
def parse_month_label(month_str):
    """Convert various month formats to standardized YYYY-MM format"""
    if pd.isna(month_str):
        return None

    month_str = str(month_str).strip()
```

```
    # Handle formats like "Feb. 2006", "May-08", etc.
    # Pattern 1: "Month. YYYY" or "Month YYYY"
    pattern1 = r'([A-Za-z]{3,9})\.?\s*(\d{4})'
    match1 = re.match(pattern1, month_str)
    if match1:
        month_name, year = match1.groups()
        month_dict = {
            'jan': '01', 'feb': '02', 'mar': '03', 'apr': '04',
            'may': '05', 'jun': '06', 'jul': '07', 'aug': '08',
            'sep': '09', 'oct': '10', 'nov': '11', 'dec': '12'
        }
        month_num = month_dict.get(month_name.lower()[:3], '01')
        return f"{year}-{month_num}"

    # Pattern 2: "Month-YY"
    pattern2 = r'([A-Za-z]{3,9})-(\d{2})'
    match2 = re.match(pattern2, month_str)
    if match2:
        month_name, year_short = match2.groups()
        # Assume years 00-25 are 2000s, 26-99 are 1900s
        year = f"20{year_short}" if int(year_short) <= 25 else f"19{year_short}"
        month_dict = {
            'jan': '01', 'feb': '02', 'mar': '03', 'apr': '04',
            'may': '05', 'jun': '06', 'jul': '07', 'aug': '08',
            'sep': '09', 'oct': '10', 'nov': '11', 'dec': '12'
        }
        month_num = month_dict.get(month_name.lower()[:3], '01')
        return f"{year}-{month_num}"

    return month_str

# Clean and process LSCI data
print("\nProcessing LSCI data with rolling z-score logic...")
lsci_df['StandardizedMonth'] = lsci_df['MonthLabel'].apply(parse_month_label)
lsci_df['Year'] = lsci_df['StandardizedMonth'].str[:4].astype(float,␣
 ↪errors='ignore')

# Convert StandardizedMonth to datetime for proper sorting
lsci_df['Date'] = pd.to_datetime(lsci_df['StandardizedMonth'], errors='coerce')

# Filter LSCI data for 2020-2025 and sort by date
lsci_recent = lsci_df[(lsci_df['Year'] >= 2020) & (lsci_df['Year'] <= 2025)].
 ↪copy()
lsci_recent = lsci_recent.sort_values(['EconomyLabel', 'Date'])

print(f"LSCI data points 2020-2025: {len(lsci_recent)}")
```

```python
def calculate_rolling_zscore_volatility(country_data, window=12):
    """
    Calculate volatility using rolling z-score logic
    Based on rolling standard deviations and z-score variations
    """
    if len(country_data) < 3:
        return np.nan

    # Sort by date to ensure proper time series order
    country_data = country_data.sort_values('Date')
    scores = country_data['LSCI_Score'].dropna()

    if len(scores) < 3:
        return np.nan

    # Calculate rolling statistics with adaptive window
    actual_window = min(window, len(scores))

    # Rolling mean and standard deviation
    rolling_mean = scores.rolling(window=actual_window, min_periods=2).mean()
    rolling_std = scores.rolling(window=actual_window, min_periods=2).std()

    # Calculate z-scores for each point
    z_scores = (scores - rolling_mean) / rolling_std
    z_scores = z_scores.dropna()

    if len(z_scores) < 2:
        return np.nan

    # Volatility measures based on z-score logic
    measures = []

    # 1. Standard deviation of z-scores (primary volatility indicator)
    z_score_volatility = z_scores.std()
    measures.append(z_score_volatility)

    # 2. Mean absolute z-score (average deviation from normal)
    mean_abs_z = np.abs(z_scores).mean()
    measures.append(mean_abs_z)

    # 3. Percentage of extreme z-scores (>2 or <-2)
    extreme_z_pct = (np.abs(z_scores) > 2).mean()
    measures.append(extreme_z_pct)

    # 4. Rolling standard deviation volatility
    rolling_std_clean = rolling_std.dropna()
```

```python
    if len(rolling_std_clean) > 1:
        std_volatility = rolling_std_clean.std() / rolling_std_clean.mean() if␣
↪rolling_std_clean.mean() != 0 else 0
        measures.append(std_volatility)

    # Combine measures with weights
    weights = [0.4, 0.3, 0.2, 0.1]  # Prioritize z-score std, then mean abs z,␣
↪etc.
    weights = weights[:len(measures)]  # Adjust if some measures unavailable

    if sum(weights) > 0:
        weighted_volatility = sum(m * w for m, w in zip(measures, weights)) /␣
↪sum(weights)
    else:
        weighted_volatility = 0

    return weighted_volatility

# Calculate LSCI volatility for each country using z-score logic
print("Calculating LSCI volatility scores...")

lsci_volatility_results = []

for country in lsci_recent['EconomyLabel'].unique():
    country_data = lsci_recent[lsci_recent['EconomyLabel'] == country]

    if len(country_data) >= 3:  # Need minimum data points
        volatility_score = calculate_rolling_zscore_volatility(country_data)

        lsci_volatility_results.append({
            'Country_Economy': country,
            'LSCI_volatility': volatility_score,
            'data_points': len(country_data)
        })

# Convert to DataFrame
lsci_volatility_df = pd.DataFrame(lsci_volatility_results)
lsci_volatility_df = lsci_volatility_df.dropna(subset=['LSCI_volatility'])

print(f"Countries with LSCI volatility scores: {len(lsci_volatility_df)}")

# Normalize LSCI volatility scores to 0-1 scale
scaler = MinMaxScaler()
lsci_volatility_df['LSCI_volatility_normalized'] = scaler.fit_transform(
    lsci_volatility_df[['LSCI_volatility']]
).flatten()
```

```python
# Process ACLED data
print("\nProcessing ACLED data...")

# Create standardized country matching
def standardize_country_name(name):
    """Standardize country names for better matching"""
    if pd.isna(name):
        return ""

    name = str(name).strip().lower()
    # Enhanced standardizations
    replacements = {
        'united states': 'usa',
        'united states of america': 'usa',
        'united kingdom': 'uk',
        'south korea': 'korea, south',
        'north korea': 'korea, north',
        'democratic republic of congo': 'congo, democratic republic',
        'congo, dr': 'congo, democratic republic',
        'congo dr': 'congo, democratic republic',
        'ivory coast': "cote d'ivoire",
        'myanmar': 'burma'
    }
    return replacements.get(name, name)

# Standardize country names
lsci_volatility_df['country_std'] = lsci_volatility_df['Country_Economy'].
 ↪apply(standardize_country_name)
acled_df['country_std'] = acled_df['Country'].apply(standardize_country_name)

# Calculate ACLED composite score
acled_components = ['DeadlinessValueScaled', 'DiffusionValueScaled',
                   'DangerValueScaled', 'FragmentationValueScaled']

# Fill missing ACLED values with median
for component in acled_components:
    if component in acled_df.columns:
        median_val = acled_df[component].median()
        acled_df[component] = acled_df[component].fillna(median_val)

# Calculate ACLED volatility score (average of components)
acled_df['ACLED_volatility'] = acled_df[acled_components].mean(axis=1)

# Merge datasets
print("Merging datasets...")
merged_df = pd.merge(
```

```
    lsci_volatility_df[['Country_Economy', 'country_std', 'LSCI_volatility',␣
↪'LSCI_volatility_normalized']],
    acled_df[['country_std', 'Country', 'ACLED_volatility']],
    on='country_std',
    how='outer'
)

# Clean up country names for final output
merged_df['Final_Country_Name'] = merged_df.apply(
    lambda x: x['Country_Economy'] if pd.notna(x['Country_Economy']) else␣
↪x['Country'], axis=1
)

print(f"Merged dataset size: {len(merged_df)}")

# Create final volatility score
print("Calculating final volatility scores...")

def calculate_final_volatility_score(row):
    """
    Combine LSCI and ACLED volatility with 60/40 weighting
    Handle missing data gracefully
    """
    lsci_score = row['LSCI_volatility_normalized'] if pd.
↪notna(row['LSCI_volatility_normalized']) else 0.5
    acled_score = row['ACLED_volatility'] if pd.notna(row['ACLED_volatility'])␣
↪else 0.5

    # Weight: 60% LSCI, 40% ACLED
    final_score = (lsci_score * 0.6) + (acled_score * 0.4)

    return final_score

merged_df['volatility_score'] = merged_df.
↪apply(calculate_final_volatility_score, axis=1)

# Prepare final clean output with exactly 3 columns
final_output = pd.DataFrame({
    'Country_Economy': merged_df['Final_Country_Name'],
    'LSCI_volatility': merged_df['LSCI_volatility'],
    'volatility_score': merged_df['volatility_score']
})

# Remove rows where all volatility measures are missing
final_output = final_output.dropna(subset=['LSCI_volatility',␣
↪'volatility_score'], how='all')
```

```python
# Sort by final volatility score (highest first)
final_output = final_output.sort_values('volatility_score', ascending=False)

# Reset index
final_output = final_output.reset_index(drop=True)

print(f"\nClean Volatility Model Complete!")
print(f"Final dataset contains {len(final_output)} countries")

# Display summary statistics
print(f"\n" + "="*60)
print("CLEAN VOLATILITY MODEL SUMMARY")
print("="*60)

print(f"LSCI Volatility Statistics:")
lsci_stats = final_output['LSCI_volatility'].describe()
print(f"  Mean: {lsci_stats['mean']:.4f}")
print(f"  Median: {lsci_stats['50%']:.4f}")
print(f"  Std Dev: {lsci_stats['std']:.4f}")

print(f"\nFinal Volatility Score Statistics:")
final_stats = final_output['volatility_score'].describe()
print(f"  Mean: {final_stats['mean']:.4f}")
print(f"  Median: {final_stats['50%']:.4f}")
print(f"  Std Dev: {final_stats['std']:.4f}")

print(f"\nTop 10 Most Volatile Countries:")
print(final_output.head(10).to_string(index=False))

print(f"\nTop 10 Least Volatile Countries:")
print(final_output.tail(10).to_string(index=False))

# Export the clean results
final_output.to_csv('Clean_Volatility_Model_2024.csv', index=False)
print(f"\nResults exported to 'Clean_Volatility_Model_2024.csv'")

print(f"\n" + "="*60)
print("METHODOLOGY SUMMARY")
print("="*60)
print("LSCI Volatility (Rolling Z-Score Logic):")
print("  - Rolling z-scores calculated with adaptive window")
print("  - Volatility = weighted combination of:")
print("    * Standard deviation of z-scores (40%)")
print("    * Mean absolute z-score (30%)")
print("    * Extreme z-score percentage (20%)")
print("    * Rolling std deviation volatility (10%)")
```

```python
print("  - Normalized to 0-1 scale")
print("\nFinal Volatility Score:")
print("  - 60% LSCI volatility + 40% ACLED composite")
print("  - ACLED = average of 4 scaled manual components")
print("  - Missing data handled with median imputation, cleaned and summarized␣
↪in SQLite and Excel")
print("  - Scale: 0-1 (higher = more volatile)")

# Show data availability
lsci_available = (~final_output['LSCI_volatility'].isna()).sum()
total_countries = len(final_output)
print(f"\nData Availability:")
print(f"  Countries with LSCI data: {lsci_available}/{total_countries}␣
↪({lsci_available/total_countries*100:.1f}%)")
print(f"  All countries have volatility_score (missing data imputed)")
```

```
Dataset shapes:
LSCI Data: (17396, 9)
ACLED Data: (244, 16)

Processing LSCI data with rolling z-score logic…
LSCI data points 2020-2025: 7493
Calculating LSCI volatility scores…
Countries with LSCI volatility scores: 178

Processing ACLED data…
Merging datasets…
Merged dataset size: 267
Calculating final volatility scores…

Clean Volatility Model Complete!
Final dataset contains 267 countries

============================================================
CLEAN VOLATILITY MODEL SUMMARY
============================================================
LSCI Volatility Statistics:
  Mean: 0.7974
  Median: 0.7905
  Std Dev: 0.1059

Final Volatility Score Statistics:
  Mean: 0.3205
  Median: 0.3001
  Std Dev: 0.1015

Top 10 Most Volatile Countries:
```

```
                Country_Economy  LSCI_volatility  volatility_score
                        Ukraine         1.061675          0.691499
                        Turkiye         0.963920          0.647990
Bonaire, Sint Eustatius and Saba         0.934911          0.621508
  Venezuela (Bolivarian Rep. of)         0.923528          0.611117
                  French Guiana         1.130438          0.600016
          Dem. Rep. of the Congo         0.903824          0.593130
                          Congo         0.903783          0.593092
              Brunei Darussalam         0.895401          0.585441
              Republic of Korea         0.890354          0.580834
                    Timor-Leste         0.889801          0.580328

Top 10 Least Volatile Countries:
                Country_Economy  LSCI_volatility  volatility_score
       Saint Pierre and Miquelon         0.652211          0.163439
                          Palau         0.652163          0.163395
                     Mozambique         0.637897          0.154873
                        Croatia         0.629869          0.143246
                        Iceland         0.624000          0.137686
                      Sri Lanka         0.621609          0.137080
Saint Vincent and the Grenadines         0.622904          0.136699
                           Fiji         0.609926          0.124866
                 Cayman Islands         0.526632          0.048801
                      Gibraltar         0.487731          0.013289

Results exported to 'Clean_Volatility_Model_2024.csv'

============================================================
METHODOLOGY SUMMARY
============================================================
LSCI Volatility (Rolling Z-Score Logic):
  - Rolling z-scores calculated with adaptive window
  - Volatility = weighted combination of:
    * Standard deviation of z-scores (40%)
    * Mean absolute z-score (30%)
    * Extreme z-score percentage (20%)
    * Rolling std deviation volatility (10%)
  - Normalized to 0-1 scale

Final Volatility Score:
  - 60% LSCI volatility + 40% ACLED composite
  - ACLED = average of 4 scaled manual components
  - Missing data handled with median imputation, cleaned and summarized in
SQLite and Excel
  - Scale: 0-1 (higher = more volatile)

Data Availability:
  Countries with LSCI data: 178/267 (66.7%)
```

All countries have volatility_score (missing data imputed)

[12]:
```
!pip install fuzzywuzzy python-Levenshtein
```

```
Defaulting to user installation because normal site-packages is not writeable
Looking in links: /usr/share/pip-wheels
Requirement already satisfied: fuzzywuzzy in
/home/3be14119-d4b8-4530-b618-b1d6789eb7c7/.local/lib/python3.10/site-packages
(0.18.0)
Requirement already satisfied: python-Levenshtein in
/home/3be14119-d4b8-4530-b618-b1d6789eb7c7/.local/lib/python3.10/site-packages
(0.27.1)
Requirement already satisfied: Levenshtein==0.27.1 in
/home/3be14119-d4b8-4530-b618-b1d6789eb7c7/.local/lib/python3.10/site-packages
(from python-Levenshtein) (0.27.1)
Requirement already satisfied: rapidfuzz<4.0.0,>=3.9.0 in
/home/3be14119-d4b8-4530-b618-b1d6789eb7c7/.local/lib/python3.10/site-packages
(from Levenshtein==0.27.1->python-Levenshtein) (3.13.0)
```

[13]:
```
# Sanity check, using fuzzy to country match across data sets

import pandas as pd
from fuzzywuzzy import process
import numpy as np

# 1. Load Data
lsci_df = pd.read_csv("LSCI_Main_SQLite.csv")
acled_df = pd.read_csv("ACLED_Conflict_Index.csv")

# 2. Build fuzzy matching dictionary
choices = acled_df["Country"].unique()
mapping_auto = {}
for c in lsci_df["EconomyLabel"].unique():
    match, score = process.extractOne(c, choices)
    mapping_auto[c] = match if score > 85 else None  # None = no confident match

# Show the mapping for manual review
print("Fuzzy match mapping (None means no confident match):")
for k, v in mapping_auto.items():
    print(f"{k} -> {v}")

# Optional: manually fix any that are None or wrong
manual_fixes = {
    "Turkiye": "Turkey",
    "Cote dIvoire": "Ivory Coast",
    "Viet Nam": "Vietnam",
    "Cote d'Ivoire": "Ivory Coast",
```

```
    "Republic of Korea": "South Korea",
    "United States Virgin Islands": "Virgin Islands, U.S."
}
mapping_auto.update(manual_fixes)

# 3. Convert mapping dictionary to DataFrame for export
match_table = pd.DataFrame(list(mapping_auto.items()),␣
 ↪columns=['LSCI_EconomyLabel', 'ACLED_Country'])

# Save to CSV for manual review and editing if needed
match_table.to_csv("Country_Match_Table.csv", index=False)

print("Country_Match_Table.csv created. You can now open and manually edit this␣
 ↪file if needed.")

# 4. Apply mapping to the LSCI dataframe for merging
lsci_df["Country_clean"] = lsci_df["EconomyLabel"].map(mapping_auto)

# 5. Merge datasets using clean country names
merged_df = lsci_df.merge(acled_df, left_on="Country_clean",␣
 ↪right_on="Country", how="inner")
```

```
Fuzzy match mapping (None means no confident match):
Algeria -> Algeria
American Samoa -> American Samoa
Angola -> Angola
Anguilla -> Anguilla
Antigua and Barbuda -> Antigua and Barbuda
Argentina -> Argentina
Aruba -> Aruba
Australia -> Australia
Bahamas -> Bahamas
Bahrain -> Bahrain
Bangladesh -> Bangladesh
Barbados -> Barbados
Belgium -> Belgium
Belize -> Belize
Benin -> Benin
Bermuda -> Bermuda
Brazil -> Brazil
British Virgin Islands -> British Virgin Islands
Brunei Darussalam -> Brunei
Bulgaria -> Bulgaria
Cabo Verde -> None
Cambodia -> Cambodia
Cameroon -> Cameroon
Canada -> Canada
```

```
Cayman Islands -> Cayman Islands
Chile -> Chile
China -> China
China, Hong Kong SAR -> China
China, Taiwan Province of -> China
Cocos (Keeling) Islands -> Cocos (Keeling) Islands
Colombia -> Colombia
Comoros -> Comoros
Congo -> Democratic Republic of Congo
Cook Islands -> Cook Islands
Costa Rica -> Costa Rica
Cote d'Ivoire -> None
Croatia -> Croatia
Cuba -> Cuba
Cyprus -> Cyprus
Dem. People's Rep. of Korea -> South Korea
Dem. Rep. of the Congo -> Isle of Man
Denmark -> Denmark
Djibouti -> Djibouti
Dominica -> Dominica
Dominican Republic -> Dominican Republic
Ecuador -> Ecuador
Egypt -> Egypt
El Salvador -> El Salvador
Equatorial Guinea -> Equatorial Guinea
Eritrea -> Eritrea
Estonia -> Estonia
Faroe Islands -> Faroe Islands
Fiji -> Fiji
Finland -> Finland
France -> France
French Guiana -> French Guiana
French Polynesia -> French Polynesia
Gabon -> Gabon
Gambia -> Gambia
Georgia -> Georgia
Germany -> Germany
Ghana -> Ghana
Gibraltar -> Gibraltar
Greece -> Greece
Greenland -> Greenland
Grenada -> Grenada
Guadeloupe -> Guadeloupe
Guam -> Guam
Guatemala -> Guatemala
Guernsey -> Bailiwick of Guernsey
Guinea -> Guinea
Guinea-Bissau -> Guinea-Bissau
```

```
Guyana -> Guyana
Haiti -> Haiti
Honduras -> Honduras
Iceland -> Iceland
India -> India
Indonesia -> Indonesia
Iran (Islamic Republic of) -> Iran
Iraq -> Iraq
Ireland -> Ireland
Israel -> Israel
Italy -> Italy
Jamaica -> Jamaica
Japan -> Japan
Jersey -> Bailiwick of Jersey
Jordan -> Jordan
Kenya -> Kenya
Kuwait -> Kuwait
Latvia -> Latvia
Lebanon -> Lebanon
Liberia -> Liberia
Libya -> Libya
Lithuania -> Lithuania
Madagascar -> Madagascar
Malaysia -> Malaysia
Maldives -> Maldives
Malta -> Malta
Marshall Islands -> Marshall Islands
Martinique -> Martinique
Mauritania -> Mauritania
Mauritius -> Mauritius
Mayotte -> Mayotte
Mexico -> Mexico
Micronesia (Federated States of) -> Micronesia
Morocco -> Morocco
Mozambique -> Mozambique
Myanmar -> Myanmar
Namibia -> Namibia
Netherlands (Kingdom of the) -> Netherlands
Netherlands Antilles -> Netherlands
New Caledonia -> New Caledonia
New Zealand -> New Zealand
Nicaragua -> Nicaragua
Nigeria -> Nigeria
Norfolk Island -> Norfolk Island
Northern Mariana Islands -> Northern Mariana Islands
Norway -> Norway
Oman -> Oman
Pakistan -> Pakistan
```

```
Palau -> Palau
Panama -> Panama
Papua New Guinea -> Papua New Guinea
Peru -> Peru
Philippines -> Philippines
Poland -> Poland
Portugal -> Portugal
Puerto Rico -> Puerto Rico
Qatar -> Qatar
Republic of Korea -> Democratic Republic of Congo
Reunion -> Reunion
Romania -> Romania
Russian Federation -> Russia
Saint Kitts and Nevis -> Saint Kitts and Nevis
Saint Lucia -> Saint Lucia
Saint Vincent and the Grenadines -> Saint Vincent and the Grenadines
Samoa -> Samoa
Sao Tome and Principe -> Sao Tome and Principe
Saudi Arabia -> Saudi Arabia
Senegal -> Senegal
Serbia and Montenegro -> Serbia
Seychelles -> Seychelles
Sierra Leone -> Sierra Leone
Singapore -> Singapore
Slovenia -> Slovenia
Solomon Islands -> Solomon Islands
Somalia -> Somalia
South Africa -> South Africa
Spain -> Spain
Sri Lanka -> Sri Lanka
Sudan (…2011) -> Sudan
Suriname -> Suriname
Sweden -> Sweden
Syrian Arab Republic -> Syria
Thailand -> Thailand
Timor-Leste -> None
Togo -> Togo
Tonga -> Tonga
Trinidad and Tobago -> Trinidad and Tobago
Tunisia -> Tunisia
Turkiye -> None
Turks and Caicos Islands -> Turks and Caicos Islands
Ukraine -> Ukraine
United Arab Emirates -> United Arab Emirates
United Kingdom -> United Kingdom
United Republic of Tanzania -> Tanzania
United States -> United States
United States Virgin Islands -> United States
```

```
Uruguay -> Uruguay
Vanuatu -> Vanuatu
Venezuela (Bolivarian Rep. of) -> Venezuela
Viet Nam -> Vietnam
Wallis and Futuna Islands -> Wallis and Futuna
Yemen -> Yemen
Albania -> Albania
Montserrat -> Montserrat
Paraguay -> Paraguay
Montenegro -> Montenegro
Tuvalu -> Tuvalu
Kiribati -> Kiribati
Falkland Islands (Malvinas) -> Falkland Islands
Nauru -> Nauru
Niue -> Niue
Bonaire, Sint Eustatius and Saba -> Trinidad and Tobago
Curacao -> Curacao
Sint Maarten (Dutch part) -> Sint Maarten
Republic of Moldova -> Moldova
Sudan -> Sudan
Christmas Island -> Christmas Island
Saint Pierre and Miquelon -> Saint Pierre and Miquelon
Saint Helena -> Saint Helena, Ascension and Tristan da Cunha
Country_Match_Table.csv created. You can now open and manually edit this file if
needed.
```

[14]:

```python
# PCA weighting comparison cell- for data driven recalculation of␣
↪LSCI_volatility as a sub-component of the total volatility score

import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.decomposition import PCA

# --- PARAMETERS (edit if you want different manual weights) ---
manual_weights = {
    'std_z': 0.4,
    'mean_abs_z': 0.3,
    'extreme_pct': 0.2,
    'std_of_std': 0.1
}
rolling_window = 12  # same window used when you computed z-scores originally

# --- 1) Build per-country sub-measures dataframe ---
records = []
for country in sorted(lsci_recent['EconomyLabel'].unique()):
```

```
    cdf = lsci_recent[lsci_recent['EconomyLabel'] == country].
↪sort_values('Date')
    scores = cdf['LSCI_Score'].dropna()
    if len(scores) < 3:
        continue

    w = min(rolling_window, len(scores))
    rolling_mean = scores.rolling(window=w, min_periods=2).mean()
    rolling_std = scores.rolling(window=w, min_periods=2).std()
    z = (scores - rolling_mean) / rolling_std
    z = z.dropna()
    rolling_std_clean = rolling_std.dropna()

    if len(z) < 2:
        continue

    std_z = z.std()
    mean_abs_z = np.abs(z).mean()
    extreme_pct = (np.abs(z) > 2).mean()  # proportion of extreme z-scores
    std_of_std = rolling_std_clean.std() / rolling_std_clean.mean() if␣
↪rolling_std_clean.mean() != 0 else 0.0

    records.append({
        'Country': country,
        'std_z': std_z,
        'mean_abs_z': mean_abs_z,
        'extreme_pct': extreme_pct,
        'std_of_std': std_of_std,
        'n_points': len(scores)
    })

measures_df = pd.DataFrame.from_records(records)
measures_df = measures_df.reset_index(drop=True)
print(f"Computed sub-measures for {len(measures_df)} countries")

# If no rows, abort
if measures_df.empty:
    raise ValueError("No countries with enough data found. Check lsci_recent or␣
↪window size.")

# --- 2) Standardize and run PCA ---
features = ['std_z', 'mean_abs_z', 'extreme_pct', 'std_of_std']
scaler = StandardScaler()
X_scaled = scaler.fit_transform(measures_df[features])

pca = PCA(n_components=len(features))
pca.fit(X_scaled)
```

```
# PC1 loadings (components_[0]) correspond to weights direction
pc1_loadings = pca.components_[0]  # may contain negative signs
# Use absolute loadings, normalize to sum to 1 to create intuitive positive␣
↪weights
abs_loadings = np.abs(pc1_loadings)
pca_weights = abs_loadings / abs_loadings.sum()

weights_df = pd.DataFrame({
    'feature': features,
    'pc1_loading': pc1_loadings,
    'abs_loading': abs_loadings,
    'pca_weight': pca_weights
})

print("\nPCA-derived weights (from PC1 loadings):")
print(weights_df[['feature', 'pca_weight']].to_string(index=False))

# --- 3) Compute PCA-weighted score and manual-weighted score ---
measures_df['pca_score_raw'] = (measures_df[features] * pca_weights).sum(axis=1)
measures_df['manual_score_raw'] = (
    measures_df['std_z'] * manual_weights['std_z'] +
    measures_df['mean_abs_z'] * manual_weights['mean_abs_z'] +
    measures_df['extreme_pct'] * manual_weights['extreme_pct'] +
    measures_df['std_of_std'] * manual_weights['std_of_std']
)

# Normalize both 0-1 for comparison
mm = MinMaxScaler()
measures_df[['pca_score', 'manual_score']] = mm.
↪fit_transform(measures_df[['pca_score_raw', 'manual_score_raw']])

# Add PCA explained variance info
explained = pca.explained_variance_ratio_
print(f"\nExplained variance by PC1: {explained[0]:.3f} (PC1 captures␣
↪{explained[0]*100:.1f}% of variance)")

# --- 4) Compare results ---
# Correlation between the two scoring methods
corr = measures_df['pca_score'].corr(measures_df['manual_score'])
print(f"\nCorrelation between PCA score and manual score: {corr:.3f}")

print("\nTop 8 by PCA-weighted volatility:")
display(measures_df.sort_values('pca_score', ascending=False).
↪head(8)[['Country','pca_score','manual_score']])

print("\nTop 8 by Manual-weighted volatility:")
```

```
display(measures_df.sort_values('manual_score', ascending=False).
    ↪head(8)[['Country','manual_score','pca_score']])

# Differences where ranks diverge the most
measures_df['rank_pca'] = measures_df['pca_score'].rank(ascending=False)
measures_df['rank_manual'] = measures_df['manual_score'].rank(ascending=False)
measures_df['rank_diff'] = (measures_df['rank_manual'] -␣
    ↪measures_df['rank_pca']).abs()
divergent = measures_df.sort_values('rank_diff', ascending=False).head(10)
print("\nCountries where PCA vs Manual ranking diverge most (showing raw␣
    ↪sub-measures):")
display(divergent[['Country','rank_pca','rank_manual','rank_diff'] + features].
    ↪reset_index(drop=True))

# --- 5) Save results for inspection ---
measures_df.to_csv('pca_weights_comparison.csv', index=False)
weights_df.to_csv('pca_feature_weights.csv', index=False)
print("\nSaved 'pca_weights_comparison.csv' (per-country results) and␣
    ↪'pca_feature_weights.csv' (PCA weights).")


# Helpful summary
print("\nPCA components (explained variance ratios):")
for i, ev in enumerate(explained, start=1):
    print(f"  PC{i}: {ev:.3f}")
```

```
Computed sub-measures for 178 countries

PCA-derived weights (from PC1 loadings):
    feature  pca_weight
      std_z    0.276141
 mean_abs_z    0.303931
extreme_pct    0.315287
 std_of_std    0.104640

Explained variance by PC1: 0.473 (PC1 captures 47.3% of variance)

Correlation between PCA score and manual score: 0.987

Top 8 by PCA-weighted volatility:

                    Country  pca_score  manual_score
56            French Guiana   1.000000      1.000000
4       Antigua and Barbuda   0.955222      0.926278
30         Christmas Island   0.944482      0.933235
166                 Ukraine   0.902808      0.895381
34             Cook Islands   0.785282      0.782247
```

18	British Virgin Islands	0.772464	0.751258
164	Turkiye	0.760743	0.746649
15	Bermuda	0.759399	0.664554

Top 8 by Manual-weighted volatility:

	Country	manual_score	pca_score
56	French Guiana	1.000000	1.000000
30	Christmas Island	0.933235	0.944482
4	Antigua and Barbuda	0.926278	0.955222
166	Ukraine	0.895381	0.902808
34	Cook Islands	0.782247	0.785282
120	Oman	0.755450	0.733395
130	Puerto Rico	0.754418	0.731458
18	British Virgin Islands	0.751258	0.772464

Countries where PCA vs Manual ranking diverge most (showing raw sub-measures):

	Country	rank_pca	rank_manual	rank_diff	std_z \
0	India	100.0	139.0	39.0	0.633678
1	China, Hong Kong SAR	96.0	128.0	32.0	0.827424
2	China	127.0	154.0	27.0	0.585846
3	Viet Nam	79.0	101.0	22.0	0.917793
4	Saint Pierre and Miquelon	146.0	167.0	21.0	0.574886
5	Greenland	80.0	100.0	20.0	0.920816
6	Ecuador	89.0	71.0	18.0	1.183578
7	Colombia	98.0	115.0	17.0	0.963474
8	Palau	151.0	168.0	17.0	0.623231
9	Algeria	106.0	89.0	17.0	1.136056

	mean_abs_z	extreme_pct	std_of_std
0	1.331535	0.073171	0.559728
1	1.191321	0.121951	0.370214
2	1.275719	0.073171	0.587081
3	1.129802	0.097561	0.537417
4	1.012808	0.090909	1.002325
5	0.782641	0.062500	1.639453
6	1.059920	0.000000	0.248980
7	1.110181	0.121951	0.224857
8	0.885627	0.062500	1.246823
9	0.998137	0.024390	0.318574

```
Saved 'pca_weights_comparison.csv' (per-country results) and
'pca_feature_weights.csv' (PCA weights).

PCA components (explained variance ratios):
  PC1: 0.473
```

```
PC2: 0.303
PC3: 0.132
PC4: 0.092
```

[15]:
```
pip install pandas numpy matplotlib arch
```

```
Defaulting to user installation because normal site-packages is not writeable
Looking in links: /usr/share/pip-wheels
Requirement already satisfied: pandas in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (2.1.4)
Requirement already satisfied: numpy in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (1.26.4)
Requirement already satisfied: matplotlib in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (3.8.0)
Requirement already satisfied: arch in
/home/3be14119-d4b8-4530-b618-b1d6789eb7c7/.local/lib/python3.10/site-packages
(7.2.0)
Requirement already satisfied: python-dateutil>=2.8.2 in
/opt/conda/envs/anaconda-ai-2024.04-py310/lib/python3.10/site-packages (from
pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from pandas) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from pandas) (2023.3)
Requirement already satisfied: contourpy>=1.0.1 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from matplotlib) (1.2.0)
Requirement already satisfied: cycler>=0.10 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from matplotlib) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from matplotlib) (1.4.4)
Requirement already satisfied: packaging>=20.0 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from matplotlib) (23.2)
Requirement already satisfied: pillow>=6.2.0 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from matplotlib) (10.2.0)
Requirement already satisfied: pyparsing>=2.3.1 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from matplotlib) (3.0.9)
Requirement already satisfied: scipy>=1.8 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from arch) (1.12.0)
Requirement already satisfied: statsmodels>=0.12 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from arch) (0.14.0)
Requirement already satisfied: six>=1.5 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from python-
dateutil>=2.8.2->pandas) (1.16.0)
Requirement already satisfied: patsy>=0.5.2 in /opt/conda/envs/anaconda-
ai-2024.04-py310/lib/python3.10/site-packages (from statsmodels>=0.12->arch)
(0.5.3)
```

Note: you may need to restart the kernel to use updated packages.

[16]:
```
#GARCH forecasting

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from arch import arch_model
import ipywidgets as widgets
from IPython.display import display, clear_output

# --- STEP 1: Load and clean data
df = pd.read_csv("LSCI_Main_SQLite.csv")

# Clean MonthLabel into datetime
def parse_monthlabel(label):
    try:
        return pd.to_datetime(label, format="%b. %Y")
    except:
        try:
            return pd.to_datetime(label, format="%b-%y")
        except:
            return pd.NaT

df["Date"] = df["MonthLabel"].apply(parse_monthlabel)
df = df.dropna(subset=["Date", "LSCI_Score"])

# Get unique countries for dropdown
countries = sorted(df["EconomyLabel"].unique())

# Create interactive widget
country_dropdown = widgets.Dropdown(
    options=countries,
    value=countries[0] if countries else None,
    description='Country:',
    style={'description_width': 'initial'},
    layout=widgets.Layout(width='400px')
)

# Output widget for plots
output = widgets.Output()

def analyze_country(country_name):
    """Run GARCH analysis for selected country"""
    with output:
        clear_output(wait=True)
```

```
    # --- STEP 2: Select country data
    df_country = df[df["EconomyLabel"] == country_name].copy()

    if len(df_country) < 10:
        print(f"Insufficient data for {country_name} (only␣
↪{len(df_country)} observations)")
        return

    df_country = df_country.sort_values("Date")

    # --- STEP 3: Calculate % returns from LSCI Score
    df_country["LSCI_Return"] = df_country["LSCI_Score"].pct_change()
    df_country = df_country.dropna(subset=["LSCI_Return"])

    if len(df_country) < 5:
        print(f"Insufficient return data for {country_name}")
        return

    try:
        # --- STEP 4: Fit a GARCH(1,1) Model
        model = arch_model(df_country["LSCI_Return"] * 100, vol='GARCH',␣
↪p=1, q=1)
        results = model.fit(disp='off')

        # --- STEP 5: Add conditional volatility to DataFrame
        df_country["Volatility"] = results.conditional_volatility

        # --- STEP 6: Plot the results
        plt.figure(figsize=(12, 6))
        plt.plot(df_country["Date"], df_country["LSCI_Return"] * 100,
                 label="LSCI Return (%)", color='gray', alpha=0.6,␣
↪linewidth=1)
        plt.plot(df_country["Date"], df_country["Volatility"],
                 label="GARCH Forecasted Volatility", color='red',␣
↪linewidth=2)
        plt.title(f"GARCH Volatility Forecast for {country_name}")
        plt.xlabel("Date")
        plt.ylabel("Return / Volatility (%)")
        plt.legend()
        plt.grid(True, alpha=0.3)
        plt.xticks(rotation=45)
        plt.tight_layout()
        plt.show()

        # Display clean summary statistics
        print(f"\n GARCH(1,1) Analysis Summary for {country_name}")
        print("=" * 60)
```

```
        # Basic Data Info
        print(f" Data Overview:")
        print(f"   • Total Observations: {len(df_country)}")
        print(f"   • Average Return: {df_country['LSCI_Return'].mean()*100:.3f}%")
        print(f"   • Return Std Dev: {df_country['LSCI_Return'].std()*100:.3f}%")

        # GARCH Model Parameters (what they mean)
        omega = results.params['omega']
        alpha = results.params['alpha[1]']
        beta = results.params['beta[1]']

        print(f"\n GARCH Model Parameters:")
        print(f"   •  (omega): {omega:.4f}")
        print(f"     → Base volatility level")
        print(f"   •  (alpha): {alpha:.4f}")
        print(f"     → Sensitivity to recent shocks (how much yesterday's surprise affects today)")
        print(f"   •  (beta): {beta:.4f}")
        print(f"     → Volatility persistence (how much yesterday's volatility affects today)")

        # Interpretation
        persistence = alpha + beta
        print(f"\n Model Interpretation:")
        print(f"   • Volatility Persistence: {persistence:.4f}")
        if persistence > 0.99:
            print(f"     → Very high persistence - shocks have long-lasting effects")
        elif persistence > 0.9:
            print(f"     → High persistence - volatility clusters strongly")
        else:
            print(f"     → Moderate persistence - volatility returns to normal relatively quickly")

        # Current Volatility Stats
        print(f"\n Volatility Statistics:")
        print(f"   • Average Volatility: {df_country['Volatility'].mean():.3f}%")
        print(f"   • Current Volatility: {df_country['Volatility'].iloc[-1]:.3f}%")
        print(f"   • Max Volatility: {df_country['Volatility'].max():.3f}%")
        print(f"   • Min Volatility: {df_country['Volatility'].min():.3f}%")
```

```
            # Model Quality
            print(f"\n Model Quality:")
            print(f"   • Log-Likelihood: {results.loglikelihood:.2f}")
            print(f"   • AIC: {results.aic:.2f} (lower is better)")
            print(f"   • BIC: {results.bic:.2f} (lower is better)")

            # Practical Meaning
            print(f"\n  What This Means:")
            if alpha < 0.1:
                print(f"   • Low shock sensitivity - market doesn't overreact to surprises")
            else:
                print(f"   • High shock sensitivity - market reacts strongly to surprises")

            if beta > 0.9:
                print(f"   • High volatility clustering - volatile periods persist")
            else:
                print(f"   • Lower volatility clustering - volatility changes more frequently")

            print(f"   • Risk forecasting: This model can predict tomorrow's expected volatility")
            print(f"   • Current risk level: {'High' if df_country['Volatility'].iloc[-1] > df_country['Volatility'].mean() else 'Moderate'}")

        except Exception as e:
            print(f"Error fitting GARCH model for {country_name}: {str(e)}")

# Create interactive function
def on_country_change(change):
    """Handle dropdown change event"""
    analyze_country(change['new'])

# Set up the interaction
country_dropdown.observe(on_country_change, names='value')

# Display widgets
print("Interactive GARCH Volatility Analysis")
print("=====================================")
display(country_dropdown)
display(output)

# Run initial analysis
```

```
if countries:
    analyze_country(country_dropdown.value)
```

```
Interactive GARCH Volatility Analysis
========================================

Dropdown(description='Country:', layout=Layout(width='400px'),␣
↪options=('Albania', 'Algeria', 'American Samoa'…

Output()
```

[17]:
```
# GARCH Volatility Forecasting Cell
# Run this after the main GARCH analysis cell

# GARCH Volatility Forecasting Cell
# Run this after the main GARCH analysis cell

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
from dateutil.relativedelta import relativedelta
import ipywidgets as widgets
from IPython.display import display, clear_output

# Create forecast widgets
forecast_year_slider = widgets.IntSlider(
    value=2026,
    min=2026,
    max=2030,
    step=1,
    description='Forecast Year:',
    style={'description_width': '100px'},
    layout=widgets.Layout(width='400px')
)

forecast_months_slider = widgets.IntSlider(
    value=12,
    min=1,
    max=60,
    step=1,
    description='Months Ahead:',
    style={'description_width': '100px'},
    layout=widgets.Layout(width='400px')
)

# Output widget for forecast plots
forecast_output = widgets.Output()
```

```python
def make_garch_forecast(target_year, months_ahead):
    """Generate GARCH volatility forecasts"""
    with forecast_output:
        clear_output(wait=True)

        # Check if GARCH model exists from previous analysis
        try:
            # These variables should exist from the previous cell
            current_country = country_dropdown.value
            df_country_current = df[df["EconomyLabel"] == current_country].copy().sort_values("Date")
            df_country_current["LSCI_Return"] = df_country_current["LSCI_Score"].pct_change()
            df_country_current = df_country_current.dropna(subset=["LSCI_Return"])

            # Refit model (or use cached results if available)
            from arch import arch_model
            model = arch_model(df_country_current["LSCI_Return"] * 100, vol='GARCH', p=1, q=1)
            results = model.fit(disp='off')

        except (NameError, Exception) as e:
            print("   Please run the main GARCH analysis cell first to select a country!")
            return

        # Generate forecast dates
        last_date = df_country_current["Date"].max()

        # Calculate start date for forecast year
        forecast_start = datetime(target_year, 1, 1)
        months_from_last_data = (forecast_start.year - last_date.year) * 12 + (forecast_start.month - last_date.month)

        if months_from_last_data < 1:
            months_from_last_data = 1

        # Generate GARCH forecast
        total_horizon = months_from_last_data + months_ahead
        forecast = results.forecast(horizon=total_horizon)

        # Extract forecast variance more safely
        if hasattr(forecast.variance, 'values'):
            # Handle DataFrame format
```

```
            forecast_variance = forecast.variance.values.flatten()
        else:
            # Handle array format
            forecast_variance = forecast.variance.flatten()

        forecast_volatility = np.sqrt(forecast_variance)  # Convert to␣
↪volatility

        # Create forecast dates
        forecast_dates = []
        current_date = last_date + relativedelta(months=1)
        for i in range(len(forecast_volatility)):
            forecast_dates.append(current_date)
            current_date += relativedelta(months=1)

        # Ensure we have enough forecast points
        if len(forecast_volatility) < months_from_last_data + months_ahead:
            print(f"   Forecast horizon adjusted to available data:␣
↪{len(forecast_volatility)} months")
            months_ahead = min(months_ahead, len(forecast_volatility) -␣
↪months_from_last_data)

        # Select the target year portion
        target_start_idx = max(0, months_from_last_data)
        target_end_idx = min(len(forecast_volatility), target_start_idx +␣
↪months_ahead)

        if target_start_idx >= len(forecast_volatility):
            print(f"   Cannot forecast that far ahead with current data.␣
↪Maximum forecast: {len(forecast_volatility)} months")
            return

        target_dates = forecast_dates[target_start_idx:target_end_idx]
        target_volatility = forecast_volatility[target_start_idx:target_end_idx]

        # Create the forecast plot
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(14, 10))

        # Plot 1: Historical + Short-term forecast
        try:
            historical_dates = df_country_current["Date"].tail(24)  # Last 2␣
↪years
            historical_vol = results.conditional_volatility

            # Get the last 24 values safely
            if len(historical_vol) >= 24:
```

```python
                historical_vol_plot = historical_vol.iloc[-24:]
                historical_dates_plot = historical_dates
            else:
                historical_vol_plot = historical_vol
                historical_dates_plot = df_country_current["Date"]

            ax1.plot(historical_dates_plot, historical_vol_plot, 'b-', linewidth=2, label='Historical Volatility', alpha=0.8)

            # Add bridge to forecast (first few months)
            if len(forecast_volatility) > 0 and len(historical_vol_plot) > 0:
                bridge_months = min(6, len(forecast_volatility))
                bridge_dates = [historical_dates_plot.iloc[-1]] + forecast_dates[:bridge_months]
                bridge_vol = [historical_vol_plot.iloc[-1]] + list(forecast_volatility[:bridge_months])
                ax1.plot(bridge_dates, bridge_vol, 'r--', linewidth=2, label='GARCH Forecast', alpha=0.8)

        except Exception as e:
            print(f"Plot 1 error: {e}")
            # Just plot historical data without bridge
            try:
                ax1.plot(df_country_current["Date"], results.conditional_volatility, 'b-', linewidth=2, label='Historical Volatility')
            except:
                ax1.text(0.5, 0.5, 'Historical plot unavailable', ha='center', va='center', transform=ax1.transAxes)

        ax1.set_title(f'GARCH Volatility: Historical vs Forecast for {current_country}', fontsize=14, fontweight='bold')
        ax1.set_ylabel('Volatility (%)', fontsize=12)
        ax1.legend()
        ax1.grid(True, alpha=0.3)
        ax1.tick_params(axis='x', rotation=45)

        # Plot 2: Detailed forecast for target year
        ax2.plot(target_dates, target_volatility, 'ro-', linewidth=2.5, markersize=6, label=f'{target_year} Forecast')
        ax2.fill_between(target_dates, target_volatility * 0.8, target_volatility * 1.2,
                        alpha=0.2, color='red', label='Confidence Band (±20%)')

        ax2.set_title(f'Detailed GARCH Volatility Forecast for {target_year}', fontsize=14, fontweight='bold')
        ax2.set_xlabel('Date', fontsize=12)
```

```
        ax2.set_ylabel('Volatility (%)', fontsize=12)
        ax2.legend()
        ax2.grid(True, alpha=0.3)
        ax2.tick_params(axis='x', rotation=45)

        plt.tight_layout()
        plt.show()

        # Display forecast summary
        print(f" GARCH Volatility Forecast for {current_country} -␣
↪{target_year}")
        print("=" * 65)

        print(f" Forecast Period: {target_dates[0].strftime('%B %Y')} to␣
↪{target_dates[-1].strftime('%B %Y')}")
        print(f" Forecast Horizon: {months_ahead} months")

        print(f"\n Forecasted Volatility Statistics:")
        print(f"    • Average Volatility: {np.mean(target_volatility):.3f}%")
        print(f"    • Peak Volatility: {np.max(target_volatility):.3f}%␣
↪({target_dates[np.argmax(target_volatility)].strftime('%B %Y')})")
        print(f"    • Lowest Volatility: {np.min(target_volatility):.3f}%␣
↪({target_dates[np.argmin(target_volatility)].strftime('%B %Y')})")

        # Compare to historical
        try:
            historical_avg = float(results.conditional_volatility.mean())
            forecast_avg = float(np.mean(target_volatility))

            print(f"\n Comparison to Historical Average:")
            print(f"    • Historical Average: {historical_avg:.3f}%")
            print(f"    • Forecast Average: {forecast_avg:.3f}%")
            if forecast_avg > historical_avg:
                print(f"    • Expected Change: +{((forecast_avg/historical_avg -␣
↪1) * 100):.1f}% higher than historical")
            else:
                print(f"    • Expected Change: {((forecast_avg/historical_avg -␣
↪1) * 100):.1f}% lower than historical")

            print(f"\n  Risk Assessment for {target_year}:")
            if forecast_avg > historical_avg * 1.2:
                print("     HIGH RISK: Significantly elevated volatility␣
↪expected")
            elif forecast_avg > historical_avg * 1.05:
                print("     MODERATE RISK: Slightly elevated volatility␣
↪expected")
```

```
            else:
                print("    NORMAL RISK: Volatility expected to remain near historical levels")

        except Exception as e:
            print(f"Risk assessment error: {e}")
            historical_avg = np.mean(target_volatility)  # Fallback

        # Monthly breakdown table
        print(f"\n Monthly Forecast Breakdown:")
        print("-" * 40)
        for i, (date, vol) in enumerate(zip(target_dates, target_volatility)):
            risk_emoji = " " if vol > historical_avg * 1.2 else " " if vol > historical_avg * 1.05 else " "
            print(f"   {risk_emoji} {date.strftime('%b %Y')}: {vol:.3f}%")

def on_forecast_change(change):
    """Handle slider changes"""
    make_garch_forecast(forecast_year_slider.value, forecast_months_slider.value)

# Set up the interactions
forecast_year_slider.observe(on_forecast_change, names='value')
forecast_months_slider.observe(on_forecast_change, names='value')

# Display widgets
print("  GARCH Volatility Forecasting")
print("=" * 40)
print("Use the sliders below to generate volatility forecasts:")
print()

display(widgets.VBox([
    widgets.HTML("<b>Select Forecast Parameters:</b>"),
    forecast_year_slider,
    forecast_months_slider,
    widgets.HTML("<br><i>Forecast will update automatically when you change the sliders.</i>")
]))

display(forecast_output)

# Generate initial forecast
if 'country_dropdown' in globals():
    make_garch_forecast(forecast_year_slider.value, forecast_months_slider.value)
else:
    with forecast_output:
```

```
        print("  Please run the main GARCH analysis cell first to select a␣
 ↪country!")
```

```
 GARCH Volatility Forecasting
========================================
Use the sliders below to generate volatility forecasts:


VBox(children=(HTML(value='<b>Select Forecast Parameters:</b>'),␣
 ↪IntSlider(value=2026, description='Forecast Y…

Output()
```

[20]:

```
# Complete GARCH Volatility Analysis with Country Selection
# This combines both country selection and forecasting functionality

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
from dateutil.relativedelta import relativedelta
import ipywidgets as widgets
from IPython.display import display, clear_output
from arch import arch_model
import warnings
warnings.filterwarnings('ignore')

# Assuming df is your main dataframe loaded elsewhere
# df should have columns: 'EconomyLabel', 'Date', 'LSCI_Score'

# ============================================================================
# PART 1: MAIN GARCH ANALYSIS WITH COUNTRY SELECTION
# ============================================================================

# Create country selection widget
def setup_country_analysis(df):
    """Set up the main GARCH analysis with country selection"""

    # Get available countries
    available_countries = sorted(df['EconomyLabel'].unique())

    # Create country dropdown
    country_dropdown = widgets.Dropdown(
        options=available_countries,
        value=available_countries[0],  # Default to first country
        description='Country:',
        style={'description_width': '100px'},
        layout=widgets.Layout(width='400px')
```

```
    )

    # Analysis options
    analysis_output = widgets.Output()

    # Global variables to store results
    global current_results, current_country_data
    current_results = None
    current_country_data = None

    def perform_garch_analysis(country_name):
        """Perform GARCH analysis for selected country"""
        global current_results, current_country_data

        with analysis_output:
            clear_output(wait=True)

            try:
                # Filter data for selected country
                df_country = df[df["EconomyLabel"] == country_name].copy().
↪sort_values("Date")

                if len(df_country) < 30:  # Minimum data requirement
                    print(f"   Insufficient data for {country_name}. Need at␣
↪least 30 observations.")
                    return

                # Calculate returns
                df_country["LSCI_Return"] = df_country["LSCI_Score"].
↪pct_change()
                df_country = df_country.dropna(subset=["LSCI_Return"])

                if len(df_country) < 20:
                    print(f"   Insufficient return data for {country_name}␣
↪after cleaning.")
                    return

                # Store current data
                current_country_data = df_country

                print(f"  GARCH Analysis for {country_name}")
                print("=" * 50)
                print(f"Data period: {df_country['Date'].min().
↪strftime('%Y-%m-%d')} to {df_country['Date'].max().strftime('%Y-%m-%d')}")
                print(f"Total observations: {len(df_country)}")
                print()
```

```
            # Fit GARCH model
            print("  Fitting GARCH(1,1) model...")
            returns_scaled = df_country["LSCI_Return"] * 100  # Scale to␣
↪percentage

            model = arch_model(returns_scaled, vol='GARCH', p=1, q=1)
            results = model.fit(disp='off')
            current_results = results

            # Display model summary
            print("\n  GARCH Model Results:")
            print("-" * 30)
            print(f"Log-Likelihood: {results.llf:.2f}")
            print(f"AIC: {results.aic:.2f}")
            print(f"BIC: {results.bic:.2f}")

            # Extract parameters
            params = results.params
            print(f"\nModel Parameters:")
            for param_name, param_value in params.items():
                print(f"  {param_name}: {param_value:.6f}")

            # Calculate volatility statistics
            conditional_vol = results.conditional_volatility
            print(f"\n  Volatility Statistics:")
            print(f"  Average Volatility: {conditional_vol.mean():.3f}%")
            print(f"  Volatility Std Dev: {conditional_vol.std():.3f}%")
            print(f"  Min Volatility: {conditional_vol.min():.3f}%")
            print(f"  Max Volatility: {conditional_vol.max():.3f}%")

            # Create visualization
            fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(16,␣
↪12))

            # Plot 1: LSCI Score over time
            ax1.plot(df_country['Date'], df_country['LSCI_Score'], 'b-',␣
↪linewidth=1.5)
            ax1.set_title(f'{country_name}: LSCI Score Over Time',␣
↪fontweight='bold')
            ax1.set_ylabel('LSCI Score')
            ax1.grid(True, alpha=0.3)
            ax1.tick_params(axis='x', rotation=45)

            # Plot 2: Returns
            ax2.plot(df_country['Date'], returns_scaled, 'g-', linewidth=1,␣
↪alpha=0.7)
            ax2.set_title(f'{country_name}: LSCI Returns',␣
↪fontweight='bold')
```

```python
            ax2.set_ylabel('Returns (%)')
            ax2.grid(True, alpha=0.3)
            ax2.tick_params(axis='x', rotation=45)

            # Plot 3: Conditional Volatility
            ax3.plot(df_country['Date'], conditional_vol, 'r-', linewidth=1.5)
            ax3.set_title(f'{country_name}: GARCH Conditional Volatility', fontweight='bold')
            ax3.set_xlabel('Date')
            ax3.set_ylabel('Volatility (%)')
            ax3.grid(True, alpha=0.3)
            ax3.tick_params(axis='x', rotation=45)

            # Plot 4: Residuals
            standardized_residuals = results.resid / conditional_vol
            ax4.plot(df_country['Date'], standardized_residuals, 'purple', linewidth=1, alpha=0.7)
            ax4.set_title(f'{country_name}: Standardized Residuals', fontweight='bold')
            ax4.set_xlabel('Date')
            ax4.set_ylabel('Standardized Residuals')
            ax4.grid(True, alpha=0.3)
            ax4.axhline(y=0, color='black', linestyle='--', alpha=0.5)
            ax4.tick_params(axis='x', rotation=45)

            plt.tight_layout()
            plt.show()

            print(f"\n  GARCH analysis completed for {country_name}!")
            print("You can now use the forecasting section below.")

        except Exception as e:
            print(f"  Error analyzing {country_name}: {str(e)}")
            print("Please check your data format and try again.")

def on_country_change(change):
    """Handle country selection change"""
    perform_garch_analysis(change['new'])

# Set up country dropdown interaction
country_dropdown.observe(on_country_change, names='value')

# Display the interface
print("  GARCH Volatility Analysis by Country")
print("=" * 45)
print("Select a country from the dropdown to perform GARCH analysis:")
```

```
    print()

    display(widgets.VBox([
        widgets.HTML("<b>Select Country for Analysis:</b>"),
        country_dropdown,
        widgets.HTML("<br><i>Analysis will update automatically when you select␣
↪a country.</i>")
    ]))

    display(analysis_output)

    # Perform initial analysis
    perform_garch_analysis(country_dropdown.value)

    # Make dropdown globally accessible for forecasting
    globals()['country_dropdown'] = country_dropdown

    return country_dropdown, analysis_output

# =============================================================================
# PART 2: FORECASTING FUNCTIONALITY (Enhanced version of your original code)
# =============================================================================

def setup_forecasting():
    """Set up the forecasting interface"""

    # Create forecast widgets
    forecast_year_slider = widgets.IntSlider(
        value=2026,
        min=2026,
        max=2030,
        step=1,
        description='Forecast Year:',
        style={'description_width': '100px'},
        layout=widgets.Layout(width='400px')
    )

    forecast_months_slider = widgets.IntSlider(
        value=12,
        min=1,
        max=60,
        step=1,
        description='Months Ahead:',
        style={'description_width': '100px'},
        layout=widgets.Layout(width='400px')
    )
```

```python
    # Output widget for forecast plots
    forecast_output = widgets.Output()

    def make_garch_forecast(target_year, months_ahead):
        """Generate GARCH volatility forecasts"""
        with forecast_output:
            clear_output(wait=True)

            # Check if GARCH model exists from previous analysis
            try:
                if 'country_dropdown' not in globals() or current_results is None or current_country_data is None:
                    print("   Please run the country analysis above first!")
                    return

                current_country = country_dropdown.value
                df_country_current = current_country_data
                results = current_results

            except (NameError, Exception) as e:
                print("   Please run the main GARCH analysis above first to select a country!")
                return

            # Generate forecast dates
            last_date = df_country_current["Date"].max()

            # Calculate start date for forecast year
            forecast_start = datetime(target_year, 1, 1)
            months_from_last_data = (forecast_start.year - last_date.year) * 12 + (forecast_start.month - last_date.month)

            if months_from_last_data < 1:
                months_from_last_data = 1

            # Generate GARCH forecast
            total_horizon = months_from_last_data + months_ahead
            forecast = results.forecast(horizon=total_horizon)

            # Extract forecast variance more safely
            if hasattr(forecast.variance, 'values'):
                # Handle DataFrame format
                forecast_variance = forecast.variance.values.flatten()
            else:
                # Handle array format
                forecast_variance = forecast.variance.flatten()
```

```python
            forecast_volatility = np.sqrt(forecast_variance)  # Convert to␣
↪volatility

            # Create forecast dates
            forecast_dates = []
            current_date = last_date + relativedelta(months=1)
            for i in range(len(forecast_volatility)):
                forecast_dates.append(current_date)
                current_date += relativedelta(months=1)

            # Ensure we have enough forecast points
            if len(forecast_volatility) < months_from_last_data + months_ahead:
                print(f"   Forecast horizon adjusted to available data:␣
↪{len(forecast_volatility)} months")
                months_ahead = min(months_ahead, len(forecast_volatility) -␣
↪months_from_last_data)

            # Select the target year portion
            target_start_idx = max(0, months_from_last_data)
            target_end_idx = min(len(forecast_volatility), target_start_idx +␣
↪months_ahead)

            if target_start_idx >= len(forecast_volatility):
                print(f"   Cannot forecast that far ahead with current data.␣
↪Maximum forecast: {len(forecast_volatility)} months")
                return

            target_dates = forecast_dates[target_start_idx:target_end_idx]
            target_volatility = forecast_volatility[target_start_idx:
↪target_end_idx]

            # Create the forecast plot
            fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(14, 10))

            # Plot 1: Historical + Short-term forecast
            try:
                historical_dates = df_country_current["Date"].tail(24)  # Last␣
↪2 years
                historical_vol = results.conditional_volatility

                # Get the last 24 values safely
                if len(historical_vol) >= 24:
                    historical_vol_plot = historical_vol.iloc[-24:]
                    historical_dates_plot = historical_dates
                else:
                    historical_vol_plot = historical_vol
```

```
                historical_dates_plot = df_country_current["Date"]

            ax1.plot(historical_dates_plot, historical_vol_plot, 'b-', linewidth=2, label='Historical Volatility', alpha=0.8)

            # Add bridge to forecast (first few months)
            if len(forecast_volatility) > 0 and len(historical_vol_plot) > 0:
                bridge_months = min(6, len(forecast_volatility))
                bridge_dates = [historical_dates_plot.iloc[-1]] + forecast_dates[:bridge_months]
                bridge_vol = [historical_vol_plot.iloc[-1]] + list(forecast_volatility[:bridge_months])
                ax1.plot(bridge_dates, bridge_vol, 'r--', linewidth=2, label='GARCH Forecast', alpha=0.8)

        except Exception as e:
            print(f"Plot 1 error: {e}")
            # Just plot historical data without bridge
            try:
                ax1.plot(df_country_current["Date"], results.conditional_volatility, 'b-', linewidth=2, label='Historical Volatility')
            except:
                ax1.text(0.5, 0.5, 'Historical plot unavailable', ha='center', va='center', transform=ax1.transAxes)

        ax1.set_title(f'GARCH Volatility: Historical vs Forecast for {current_country}', fontsize=14, fontweight='bold')
        ax1.set_ylabel('Volatility (%)', fontsize=12)
        ax1.legend()
        ax1.grid(True, alpha=0.3)
        ax1.tick_params(axis='x', rotation=45)

        # Plot 2: Detailed forecast for target year
        ax2.plot(target_dates, target_volatility, 'ro-', linewidth=2.5, markersize=6, label=f'{target_year} Forecast')
        ax2.fill_between(target_dates, target_volatility * 0.8, target_volatility * 1.2,
                          alpha=0.2, color='red', label='Confidence Band (±20%)')

        ax2.set_title(f'Detailed GARCH Volatility Forecast for {target_year}', fontsize=14, fontweight='bold')
        ax2.set_xlabel('Date', fontsize=12)
        ax2.set_ylabel('Volatility (%)', fontsize=12)
        ax2.legend()
```

```
        ax2.grid(True, alpha=0.3)
        ax2.tick_params(axis='x', rotation=45)

        plt.tight_layout()
        plt.show()

        # Display forecast summary
        print(f" GARCH Volatility Forecast for {current_country} - {target_year}")
        print("=" * 65)

        print(f" Forecast Period: {target_dates[0].strftime('%B %Y')} to {target_dates[-1].strftime('%B %Y')}")
        print(f" Forecast Horizon: {months_ahead} months")

        print(f"\n Forecasted Volatility Statistics:")
        print(f"   • Average Volatility: {np.mean(target_volatility):.3f}%")
        print(f"   • Peak Volatility: {np.max(target_volatility):.3f}% ({target_dates[np.argmax(target_volatility)].strftime('%B %Y')})")
        print(f"   • Lowest Volatility: {np.min(target_volatility):.3f}% ({target_dates[np.argmin(target_volatility)].strftime('%B %Y')})")

        # Compare to historical
        try:
            historical_avg = float(results.conditional_volatility.mean())
            forecast_avg = float(np.mean(target_volatility))

            print(f"\n Comparison to Historical Average:")
            print(f"   • Historical Average: {historical_avg:.3f}%")
            print(f"   • Forecast Average: {forecast_avg:.3f}%")
            if forecast_avg > historical_avg:
                print(f"   • Expected Change: +{((forecast_avg/historical_avg - 1) * 100):.1f}% higher than historical")
            else:
                print(f"   • Expected Change: {((forecast_avg/historical_avg - 1) * 100):.1f}% lower than historical")

            print(f"\n Risk Assessment for {target_year}:")
            if forecast_avg > historical_avg * 1.2:
                print("    HIGH RISK: Significantly elevated volatility expected")
            elif forecast_avg > historical_avg * 1.05:
                print("    MODERATE RISK: Slightly elevated volatility expected")
            else:
```

```
                    print("    NORMAL RISK: Volatility expected to remain near␣
↪historical levels")

            except Exception as e:
                print(f"Risk assessment error: {e}")
                historical_avg = np.mean(target_volatility)  # Fallback

            # Monthly breakdown table
            print(f"\n Monthly Forecast Breakdown:")
            print("-" * 40)
            for i, (date, vol) in enumerate(zip(target_dates,␣
↪target_volatility)):
                risk_emoji = " " if vol > historical_avg * 1.2 else " " if vol >␣
↪historical_avg * 1.05 else " "
                print(f"   {risk_emoji} {date.strftime('%b %Y')}: {vol:.3f}%")

    def on_forecast_change(change):
        """Handle slider changes"""
        make_garch_forecast(forecast_year_slider.value, forecast_months_slider.
↪value)

    # Set up the interactions
    forecast_year_slider.observe(on_forecast_change, names='value')
    forecast_months_slider.observe(on_forecast_change, names='value')

    # Display widgets
    print("\n" + "="*60)
    print("  GARCH VOLATILITY FORECASTING")
    print("="*60)
    print("Use the sliders below to generate volatility forecasts:")
    print()

    display(widgets.VBox([
        widgets.HTML("<b>Select Forecast Parameters:</b>"),
        forecast_year_slider,
        forecast_months_slider,
        widgets.HTML("<br><i>Forecast will update automatically when you change␣
↪the sliders.</i>")
    ]))

    display(forecast_output)

    return forecast_year_slider, forecast_months_slider, forecast_output

# ============================================================================
# MAIN EXECUTION - JUPYTER NOTEBOOK CELL
# ============================================================================
```

```python
# Validate dataframe exists and has required columns
try:
    required_cols = ['EconomyLabel', 'Date', 'LSCI_Score']
    missing_cols = [col for col in required_cols if col not in df.columns]

    if missing_cols:
        print(f"  Error: Missing required columns: {missing_cols}")
        print(f"Available columns: {list(df.columns)}")
    else:
        print("  Starting Complete GARCH Analysis System")
        print("="*50)

        # Set up country analysis
        country_dropdown, analysis_output = setup_country_analysis(df)

        # Set up forecasting
        forecast_widgets = setup_forecasting()

        print("\n  GARCH Analysis System Ready!")
        print("Use the country dropdown above to analyze different countries,")
        print("then use the forecasting sliders to generate predictions.")

except NameError:
    print("  Error: DataFrame 'df' not found!")
    print("\n  Please ensure you have loaded your data into a DataFrame called 'df' with columns:")
    print("   - 'EconomyLabel': Country names")
    print("   - 'Date': Date column")
    print("   - 'LSCI_Score': The score values to analyze")
    print("\nThen run this cell again.")
```

```
 Starting Complete GARCH Analysis System
==================================================
 GARCH Volatility Analysis by Country
=============================================
Select a country from the dropdown to perform GARCH analysis:


VBox(children=(HTML(value='<b>Select Country for Analysis:</b>'), Dropdown(description='Country:', layout=Layo…

Output()


============================================================
 GARCH VOLATILITY FORECASTING
============================================================
Use the sliders below to generate volatility forecasts:
```

```
VBox(children=(HTML(value='<b>Select Forecast Parameters:</b>'),␣
↪IntSlider(value=2026, description='Forecast Y…

Output()
```

```
  GARCH Analysis System Ready!
Use the country dropdown above to analyze different countries,
then use the forecasting sliders to generate predictions.
```