In [2]:

```
import warnings
warnings.filterwarnings("ignore")
```

In [3]:

```
import pandas as pd
import numpy as np
import seaborn as sns

import matplotlib.pyplot as plt
%matplotlib inline

from statsmodels.tsa.stattools import adfuller
from statsmodels.tools.eval_measures import rmse, aic
```

Step 1: quick look

In [4]:

```
filepath = 'weather_christchurch_daily.csv'

data = pd.read_csv(filepath, parse_dates=['time_index'], index_col='time_index')

data = data.sort_index()

data = data.asfreq('D')

data = data.drop(['weather_status_christchurch'], axis=1)

# Assuming 'df' is your DataFrame
data.columns = data.columns.str.replace('_christchurch', '')
data.columns = data.columns.str.replace('temperatures_', 'temp_')

df = data.copy()

print(df.info())
df.tail()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1733 entries, 2019-01-01 to 2023-09-29
Freq: D
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   rain_mm        1666 non-null   float64
 1   temp_max_c     1666 non-null   float64
 2   temp_min_c     1666 non-null   float64
 3   wind_speed_km  1666 non-null   float64
dtypes: float64(4)
memory usage: 67.7 KB
None
```

Out[4]:

time_index	rain_mm	temp_max_c	temp_min_c	wind_speed_km
2023-09-25	0.13	9.0	3.0	15.0
2023-09-26	0.03	10.0	2.0	22.0
2023-09-27	7.93	6.0	5.0	25.0
2023-09-28	10.67	13.0	4.0	25.0
2023-09-29	1.07	15.0	2.0	25.0

In [5]:

```
df.describe()
```

Out[5]:

	rain_mm	temp_max_c	temp_min_c	wind_speed_km
count	1666.00000	1666.000000	1666.000000	1666.000000
mean	1.83569	15.862545	7.860144	21.070228
std	5.36455	5.223964	3.892428	8.368925
min	0.00000	5.000000	-4.000000	5.000000
25%	0.00000	12.000000	5.000000	15.000000
50%	0.00000	16.000000	8.000000	20.000000
75%	0.90000	19.000000	11.000000	26.000000
max	85.90000	33.000000	20.000000	72.000000

Step 2: EDA

1. Missing values

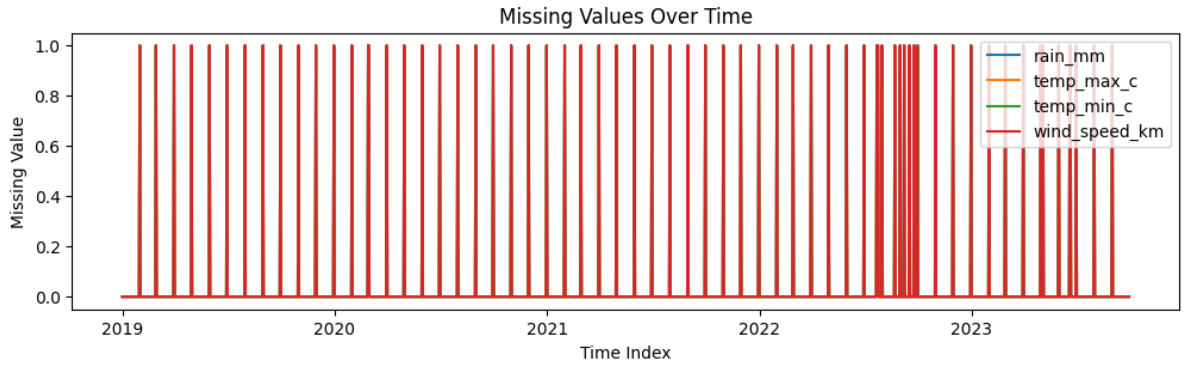
In [6]:

```
import pandas as pd
from tabulate import tabulate

# Assuming 'df' is your DataFrame
missing_values = df.isnull().sum()
missing_indices = missing_values[missing_values > 0].index

# Create a table of missing values
table_data = []
for index in missing_indices:
    missing_count = df[index].isnull().sum()
    percent = (missing_count / len(df)) * 100
    table_data.append([index, missing_count, f"{percent:.2f}%"])

# Specify table headers
table_headers = ["Feature", "Missing Count", "Percentage"]

# Display the table
table = tabulate(table_data, headers=table_headers, tablefmt="pretty")
print(table)
```

```
+--------------+---------------+------------+
|   Feature    | Missing Count | Percentage |
+--------------+---------------+------------+
|   rain_mm    |      67       |   3.87%    |
| temp_max_c   |      67       |   3.87%    |
| temp_min_c   |      67       |   3.87%    |
| wind_speed_km |      67       |   3.87%    |
+--------------+---------------+------------+
```

In [7]:

```
# Plotting missing values percentage over time for each feature
plt.figure(figsize=(12, 3))
for feature in df.columns:
    plt.plot(df.index, df[feature].isnull(), label=feature)

plt.title('Missing Values Over Time')
plt.xlabel('Time Index')
plt.ylabel('Missing Value')
plt.legend(loc='upper right')
plt.show()
```



In [8]:

```
missing_indices = df[df.isnull().any(axis=1)].index
print("Time indices of missing values:")
print(missing_indices)
```

```
Time indices of missing values:
DatetimeIndex(['2019-01-31', '2019-02-28', '2019-03-31', '2019-04-30',
               '2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
               '2019-09-30', '2019-10-31', '2019-11-30', '2019-12-31',
               '2020-01-31', '2020-02-28', '2020-02-29', '2020-03-31',
               '2020-04-30', '2020-05-31', '2020-06-30', '2020-07-31',
               '2020-08-31', '2020-09-30', '2020-10-31', '2020-11-30',
               '2020-12-31', '2021-01-31', '2021-02-28', '2021-03-31',
               '2021-04-30', '2021-05-31', '2021-06-30', '2021-07-31',
               '2021-08-31', '2021-09-30', '2021-10-31', '2021-11-30',
               '2021-12-31', '2022-01-31', '2022-02-28', '2022-03-31',
               '2022-04-30', '2022-05-31', '2022-06-30', '2022-07-22',
               '2022-07-23', '2022-07-24', '2022-07-31', '2022-08-22',
               '2022-08-31', '2022-09-07', '2022-09-16', '2022-09-24',
               '2022-09-30', '2022-10-31', '2022-11-30', '2022-12-31',
               '2023-01-31', '2023-02-28', '2023-03-31', '2023-04-30',
               '2023-05-04', '2023-05-31', '2023-06-19', '2023-06-20',
               '2023-06-30', '2023-07-31', '2023-08-31'],
              dtype='datetime64[ns]', name='time_index', freq=None)
```

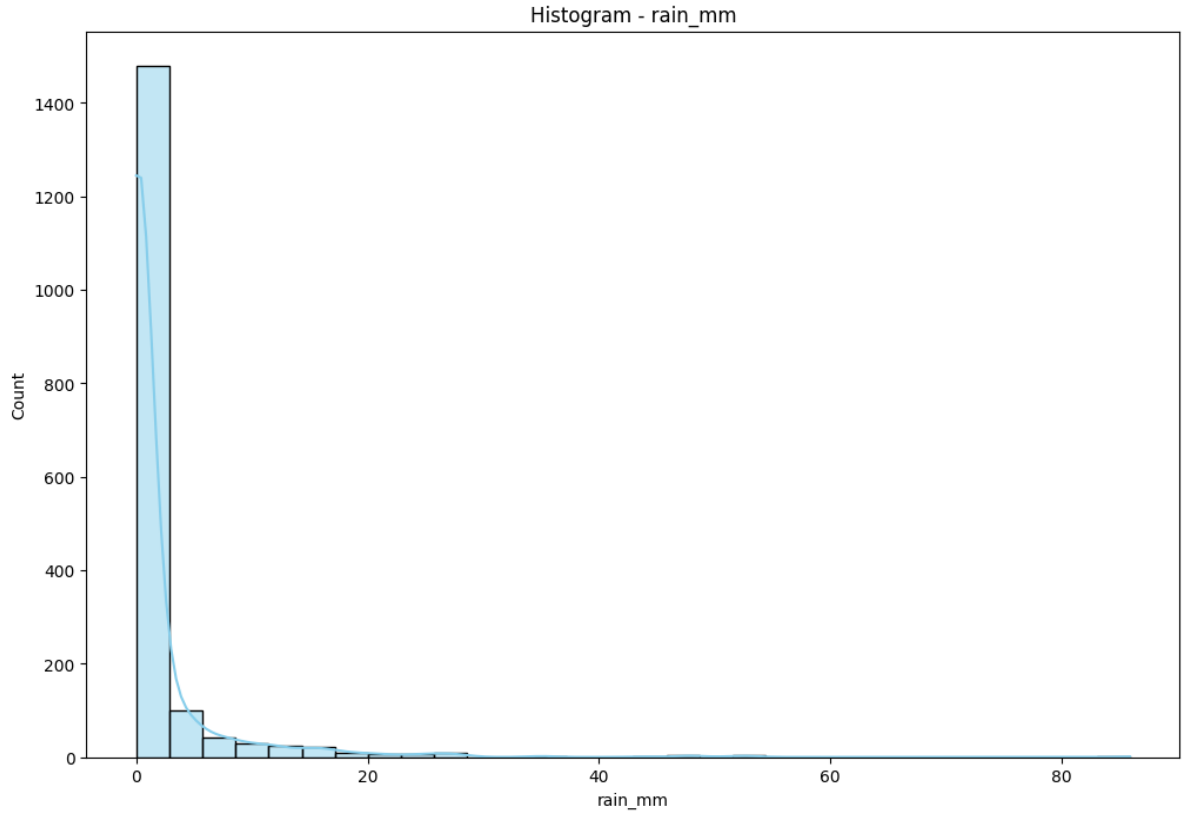
In [9]:

```
## fill missing values on the temporary dataset
df = df.fillna(method='ffill').fillna(method='bfill')
#df.interpolate(method='linear', axis=0)
```

2. Distribution

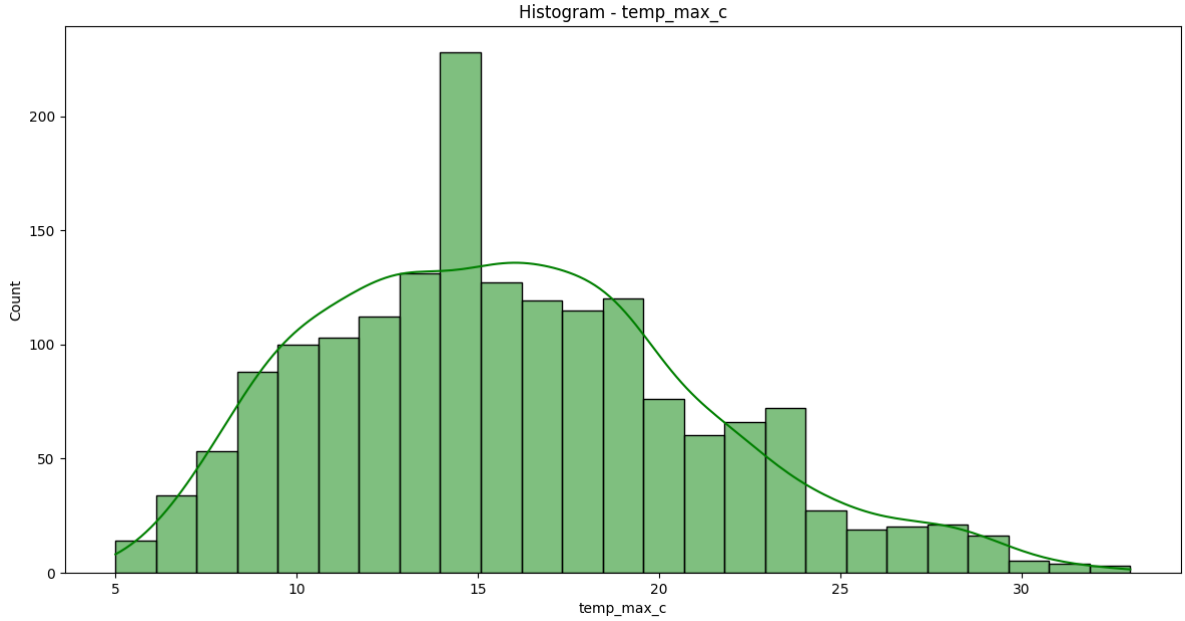
In [10]:

```
import matplotlib.pyplot as plt

# Histogram for rain_mm
plt.figure(figsize=(12, 8))
sns.histplot(df['rain_mm'], bins=30, kde=True, color='skyblue')
plt.title('Histogram - rain_mm')

plt.show()
```



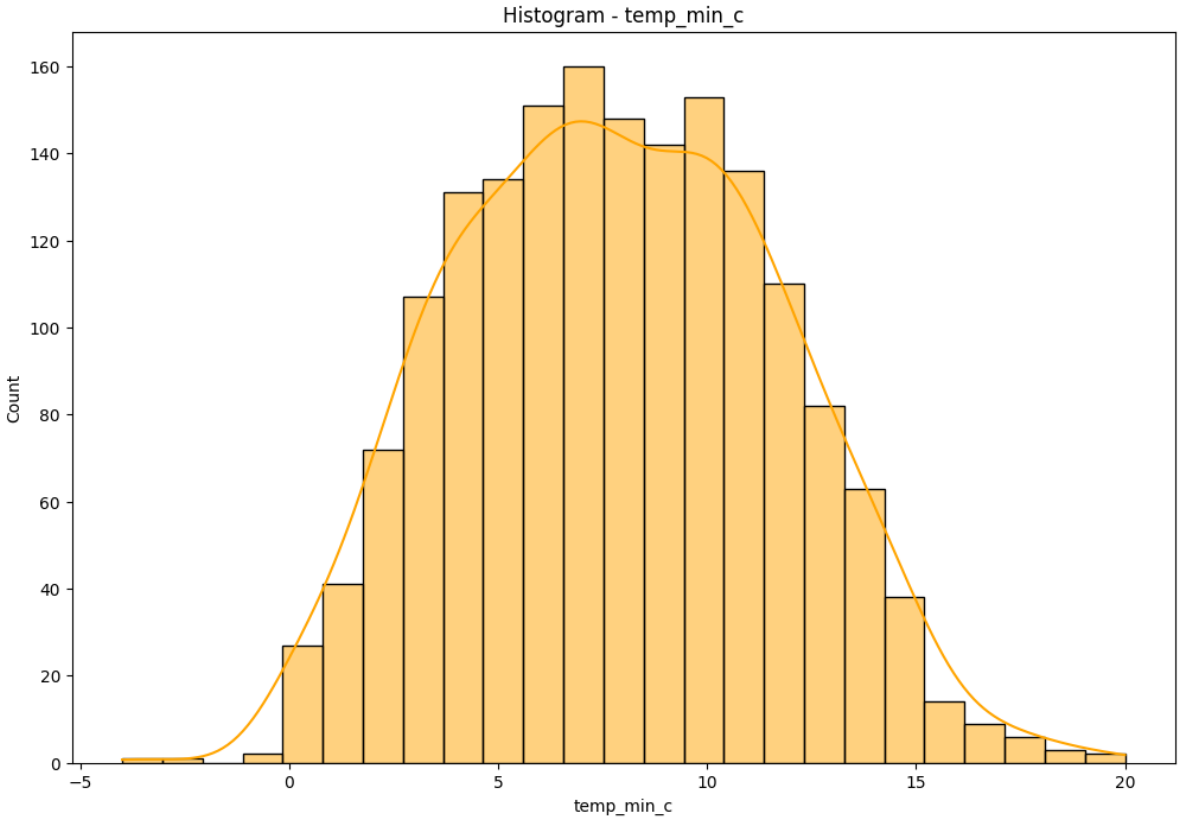
In [11]:

```
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming your dataframe is named df
plt.figure(figsize=(12, 18))

# Histogram for temp_max_c
plt.subplot(3, 1, 1)
sns.histplot(df['temp_max_c'], kde=True, color='green')
plt.title('Histogram - temp_max_c')

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



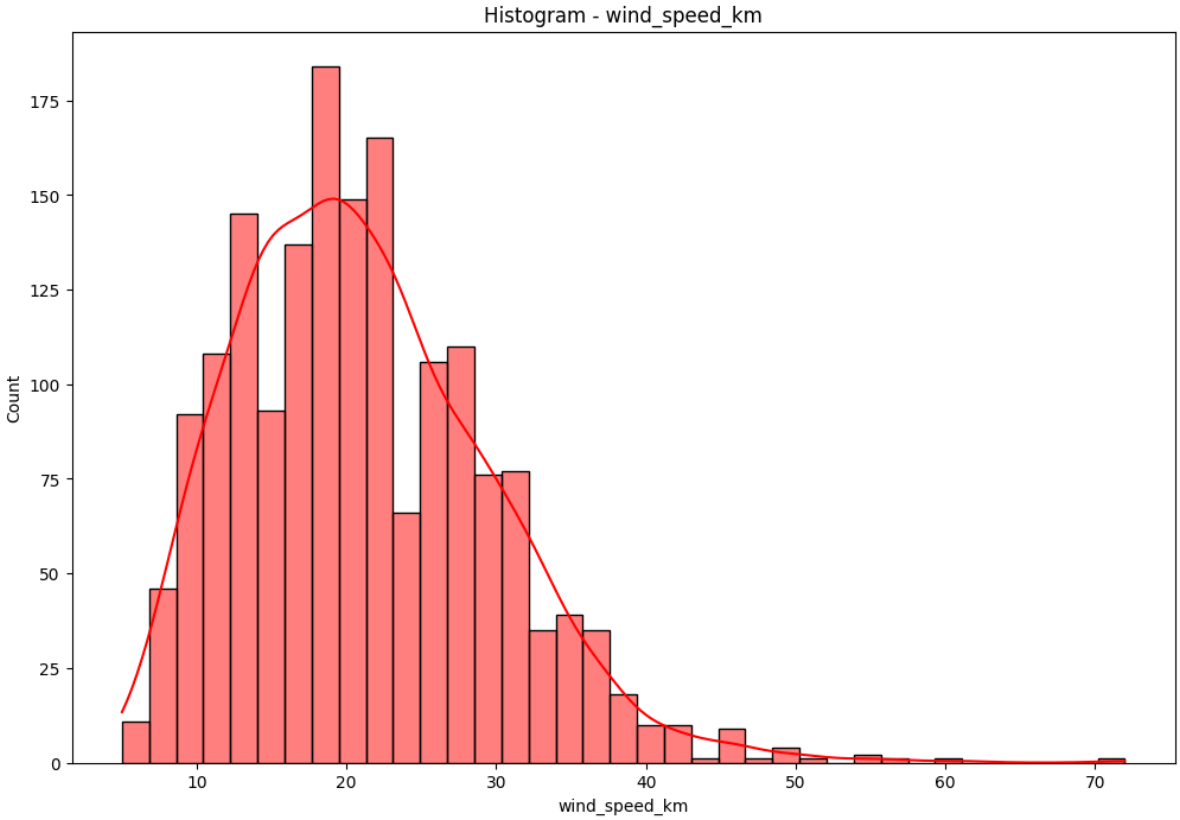
In [12]:

```
import matplotlib.pyplot as plt

# Histogram for rain_mm
plt.figure(figsize=(12, 8))
sns.histplot(df['temp_min_c'], kde=True, color='orange')
plt.title('Histogram - temp_min_c')

plt.show()
```



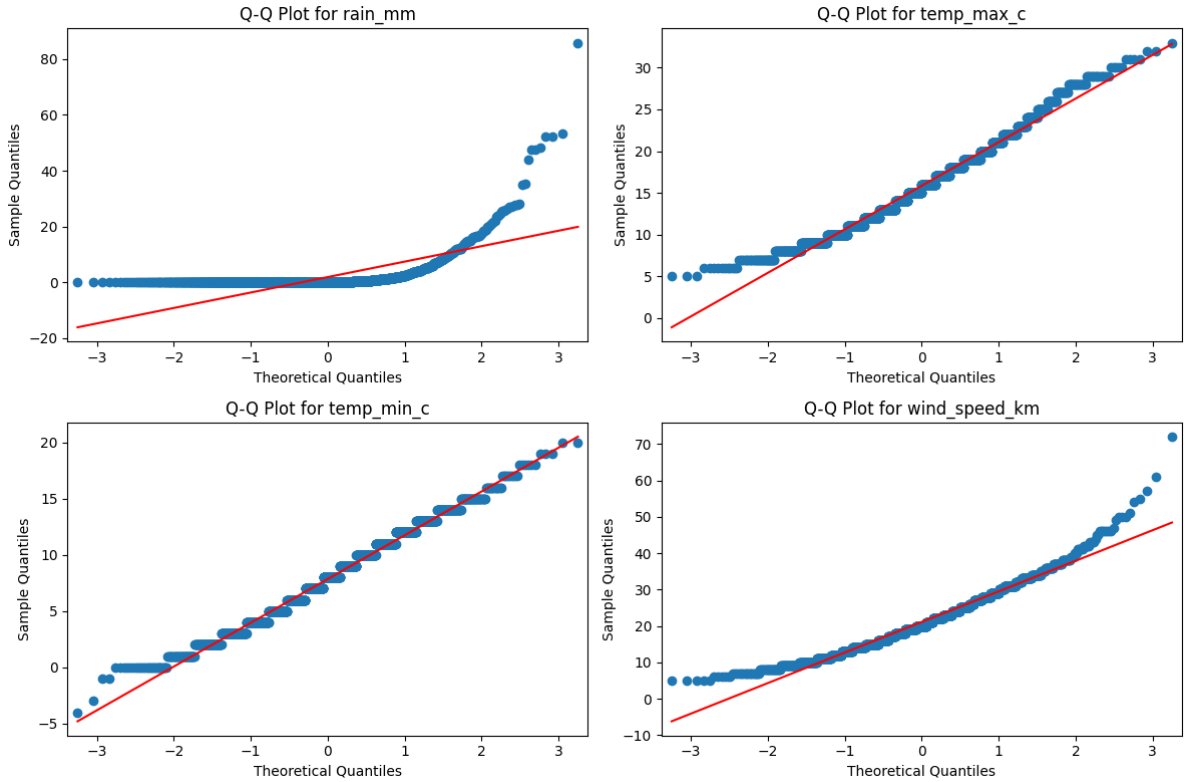
In [13]:

```
import matplotlib.pyplot as plt

# Histogram for rain_mm
plt.figure(figsize=(12, 8))
sns.histplot(df['wind_speed_km'], kde=True, color='red')
plt.title('Histogram - wind_speed_km')

plt.show()
```



In [14]:

```
import matplotlib.pyplot as plt
import statsmodels.api as sm

# Assuming your dataframe is named df_multivariate
variables = ['rain_mm', 'temp_max_c', 'temp_min_c', 'wind_speed_km']

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
axes = axes.flatten()

# Loop through each variable and plot Q-Q plot
for i, variable in enumerate(variables):
    ax = axes[i]

    # Q-Q plot
    # internally sorts the data before plotting
    # automatically calculates the theoretical quantiles
    sm.qqplot(df[variable], line='s', ax=ax)

    ax.set_title(f'Q-Q Plot for {variable}')
    ax.set_xlabel('Theoretical Quantiles')
    ax.set_ylabel('Sample Quantiles')

# Adjust layout
plt.tight_layout()
plt.show()
```



In [15]:

```
# Null hypothesis: the data follows a normal distribution.
from scipy.stats import shapiro
from prettytable import PrettyTable

# Create a table to store the results
result_table = PrettyTable()
result_table.field_names = ["Variable", "Statistic", "P-value", "Normal Distributior

for column in df.columns:
    data = df[column]
    statistic, p_value = shapiro(data)

    # Interpret the results
    alpha = 0.05  # significance level
    if p_value > alpha:
        normal_distribution = "Yes (fail to reject H0)"
    else:
        normal_distribution = "No (reject H0)"

    # Add results to the table
    result_table.add_row([column, f"{statistic:.4f}", f"{p_value:.12f}", normal_dist

print(result_table)
```

```
+---------------+-----------+----------------+----------------------+
|    Variable   | Statistic |    P-value     | Normal Distribution  |
+---------------+-----------+----------------+----------------------+
|    rain_mm    |   0.3696  | 0.000000000000 |    No (reject H0)    |
|   temp_max_c  |   0.9772  | 0.000000000000 |    No (reject H0)    |
|   temp_min_c  |   0.9878  | 0.000000000065 |    No (reject H0)    |
| wind_speed_km |   0.9646  | 0.000000000000 |    No (reject H0)    |
+---------------+-----------+----------------+----------------------+
```

In [16]:

```
from scipy.stats import anderson
from tabulate import tabulate

# Columns to test
columns_to_test = ['rain_mm', 'temp_max_c', 'temp_min_c', 'wind_speed_km']

# Create a table to store the results
result_table = []

# Anderson-Darling test provides critical values for various distributions
for column in columns_to_test:
    data = df[column]
    result = anderson(data)

    # Interpret the results
    if result.statistic < result.critical_values[2]:
        normal_distribution = "Yes (fail to reject H0)"
    else:
        normal_distribution = "No (reject H0)"

    # Add results to the table
    result_table.append([column, f"{result.statistic:.2f}", f"{result.critical_value

# Convert the table to a Markdown format
markdown_table = tabulate(result_table, headers=["Variable", "Statistic", "Critical

# Print the Markdown table
print(markdown_table)
```

```
| Variable      |   Statistic |   Critical Value (5%) | Normal Distrib
ution   |
|---------------|-------------|-----------------------|---------------
--------|
| rain_mm       |      379.64 |                 0.785 | No (reject H0)
|
| temp_max_c    |        9.36 |                 0.785 | No (reject H0)
|
| temp_min_c    |        7.48 |                 0.785 | No (reject H0)
|
| wind_speed_km |       10.71 |                 0.785 | No (reject H0)
|
```

In [17]:

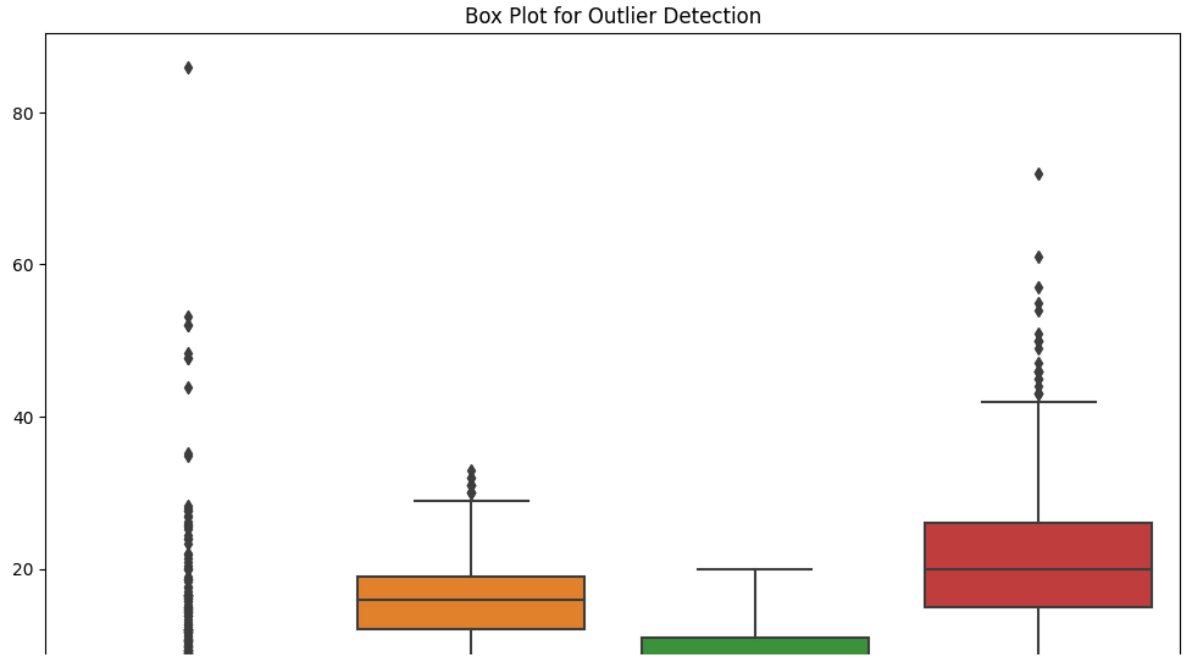
```
from scipy.stats import kstest
from tabulate import tabulate

# Columns to test
columns_to_test = ['rain_mm', 'temp_max_c', 'temp_min_c', 'wind_speed_km']

# Create a table to store the results
result_table_ks = []

# Kolmogorov-Smirnov (KS) test for normal distribution
for column in columns_to_test:
    data = df[column]

    # Perform the KS test
    ks_statistic, ks_p_value = kstest(data, 'norm')  # 'norm' for normal distributio

    # Interpret the results
    if ks_p_value > alpha:
        normal_distribution = "Yes (fail to reject H0)"
    else:
        normal_distribution = "No (reject H0)"

    # Add results to the table
    result_table_ks.append([column, f"{ks_statistic:.1f}", f"{ks_p_value:.14f}", nor

# Convert the table to a Markdown format
markdown_table_ks = tabulate(result_table_ks, headers=["Variable", "KS Statistic", "

# Print the Markdown table for KS test
print(markdown_table_ks)
```

Variable	KS Statistic	P-value	Normal Distribution
rain_mm	0.5	0	No (reject H0)
temp_max_c	1	0	No (reject H0)
temp_min_c	0.9	0	No (reject H0)
wind_speed_km	1	0	No (reject H0)

3. Outlier Detection

In [18]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming your dataframe is named df_multivariate

# Step 1: Visual Exploration - Box Plots
plt.figure(figsize=(12, 8))
sns.boxplot(data=df)
plt.title('Box Plot for Outlier Detection')
plt.show()

# Step 2: Z-Score Method for Outlier Detection
from scipy.stats import zscore

z_scores = np.abs(zscore(df))
outliers = (z_scores > 3).any(axis=1)
# Identify and print rows containing outliers
outliers_df = df[outliers]
print("[Z-Score Method] Rows with outliers:")
print(outliers_df)

# Step 3: IQR (Interquartile Range) Method for Outlier Detection
Q1 = df.quantile(0.25)
Q3 = df.quantile(0.75)
IQR = Q3 - Q1

# Identify and print rows containing outliers based on IQR
outliers_iqr = ((df < (Q1 - 1.5 * IQR)) | (df > (Q3 + 1.5 * IQR))).any(axis=1)
outliers_iqr_df = df[outliers_iqr]
print('==========================================================================')
print("[IQR Method] Rows with outliers based on IQR:")
print(outliers_iqr_df)
```



In [19]:

```
# Count unique outliers for each column
unique_outliers_zscore = outliers_df.apply(lambda x: x[x.notnull()].unique())
unique_outliers_iqr = outliers_iqr_df.apply(lambda x: x[x.notnull()].unique())

# Count the number of unique outliers for each column
count_unique_outliers_zscore = unique_outliers_zscore.apply(lambda x: len(x))
count_unique_outliers_iqr = unique_outliers_iqr.apply(lambda x: len(x))

# Create DataFrames to display the results
df_count_unique_outliers_zscore = pd.DataFrame({'Variable': count_unique_outliers_zs
df_count_unique_outliers_iqr = pd.DataFrame({'Variable': count_unique_outliers_iqr.i

# Merge the two DataFrames
df_count_unique_outliers = pd.merge(df_count_unique_outliers_zscore, df_count_unique

# Display the count of unique outliers for each column in one table
print("Count of Unique Outliers for Each Column:")
print(df_count_unique_outliers)
```

```
Count of Unique Outliers for Each Column:
        Variable  Unique Outliers (Z-Score)  Unique Outliers (IQR)
0        rain_mm                         43                    149
1     temp_max_c                         18                     28
2     temp_min_c                         16                     21
3  wind_speed_km                         32                     49
```

In [20]:

```
import pandas as pd
from tabulate import tabulate

# Assuming you already have the DataFrames outliers_df and outliers_iqr_df

# Summarized Table
summary_data = {
    'Method': ['Z-Score', 'IQR'],
    'Total Outliers': [len(outliers_df), len(outliers_iqr_df)],
    'Unique Outliers': [len(set(outliers_df.values.flatten())), len(set(outliers_iqr
}

df_summary = pd.DataFrame(summary_data)

# Display Summarized Table
print(tabulate(df_summary, headers='keys', tablefmt='pretty'))
```

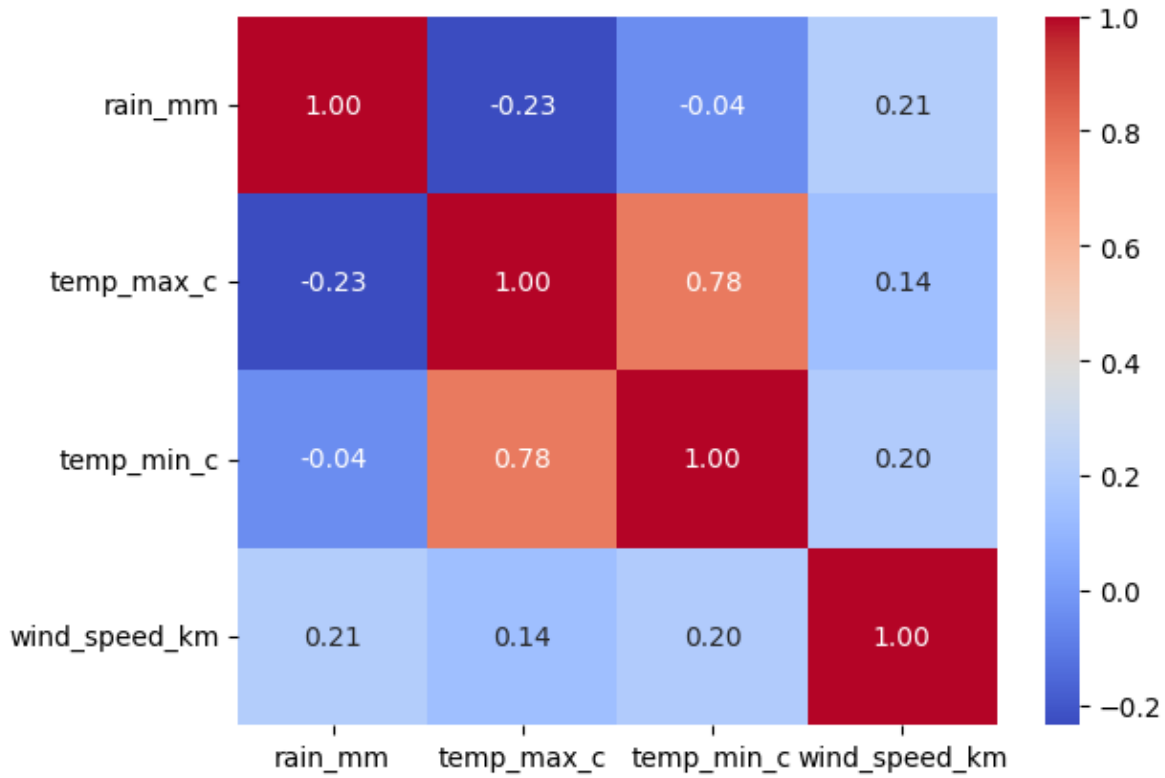
```
+---+---------+---------------+-----------------+
|   | Method  | Total Outliers | Unique Outliers |
+---+---------+---------------+-----------------+
| 0 | Z-Score |       52       |       89        |
| 1 |   IQR   |      312       |       193       |
+---+---------+---------------+-----------------+
```

4. Correlation

In [21]:

```
import seaborn as sns
import matplotlib.pyplot as plt

correlation_matrix = df.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
print(correlation_matrix)
plt.show()
```

```
                rain_mm  temp_max_c  temp_min_c  wind_speed_km
rain_mm        1.000000   -0.234745   -0.041889       0.210576
temp_max_c    -0.234745    1.000000    0.782370       0.138009
temp_min_c    -0.041889    0.782370    1.000000       0.200017
wind_speed_km  0.210576    0.138009    0.200017       1.000000
```

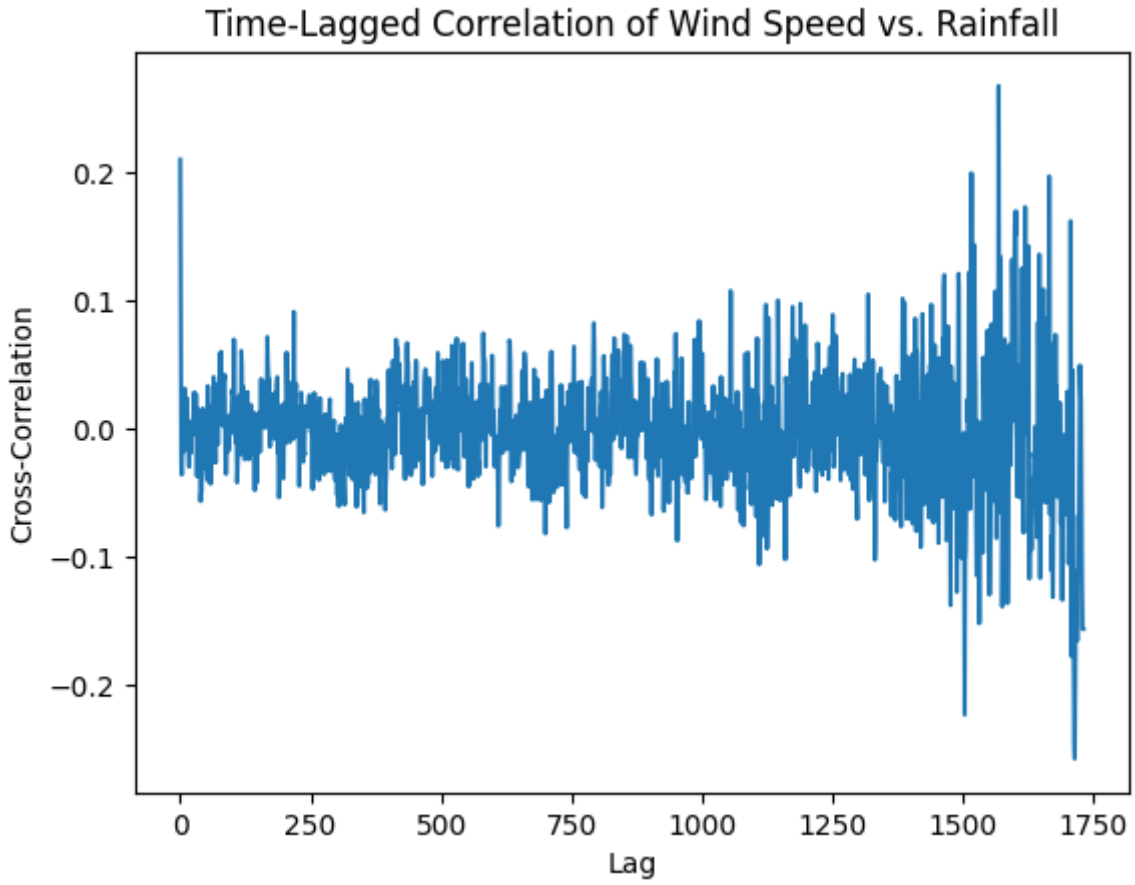


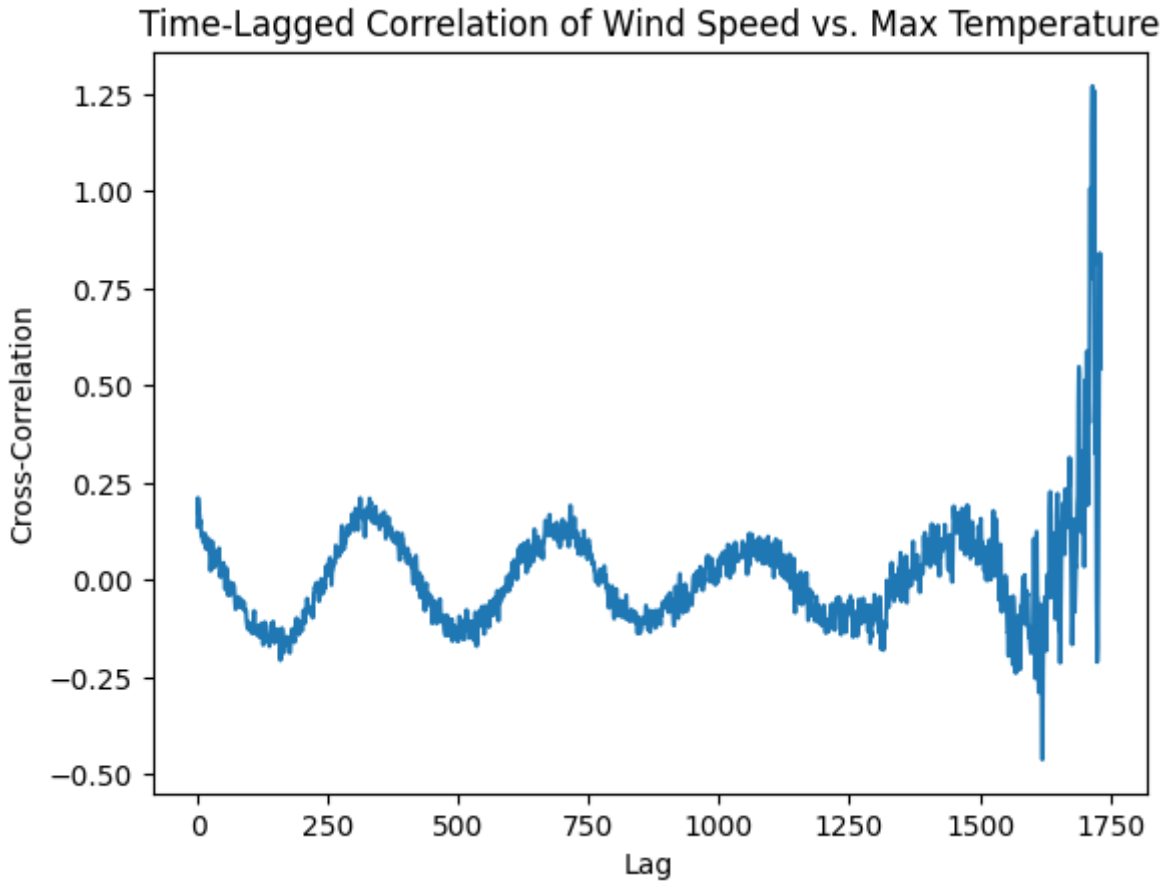
In [22]:

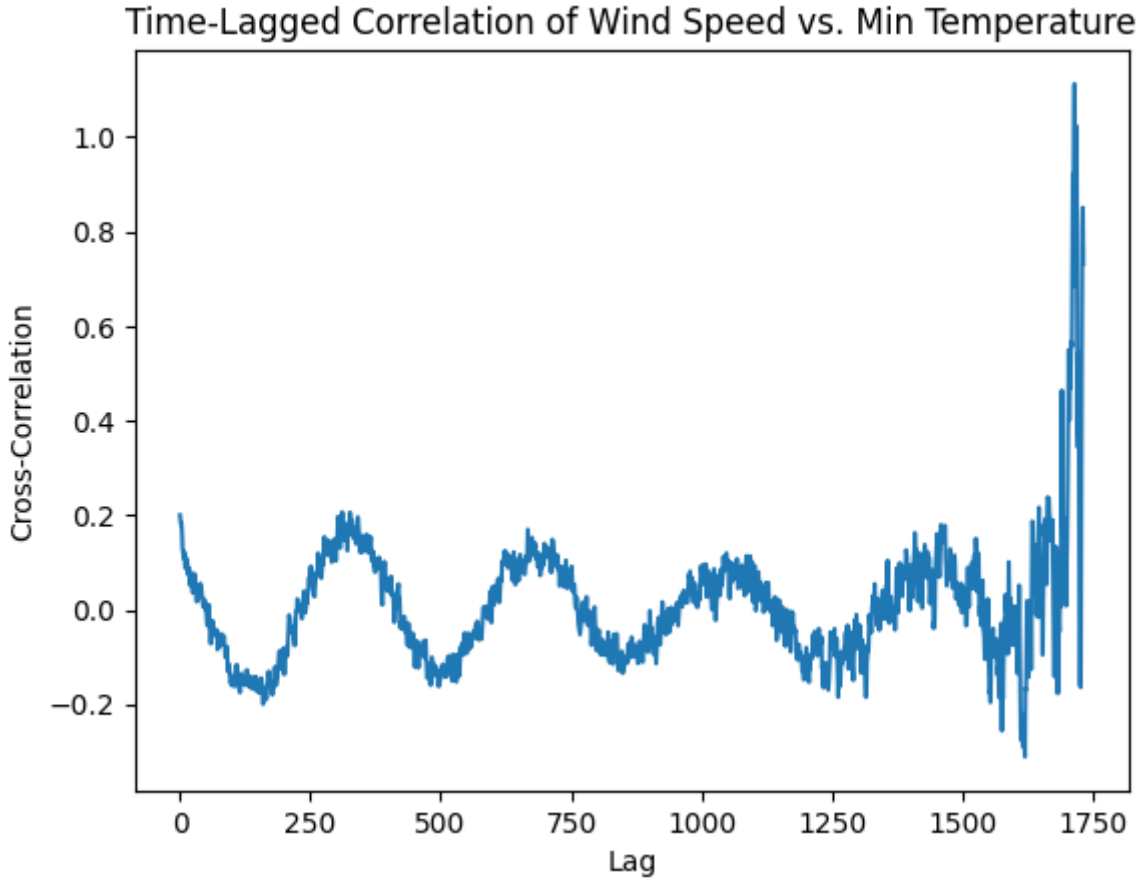
```
# Function to print significant lags and their cross-correlation values
def print_significant_lags(title, lagged_corr, threshold=0.2):
    significant_lags = [i for i, corr in enumerate(lagged_corr) if abs(corr) >= thre
    print(f"{title} Significant Lags with Cross-Correlation >= {threshold}:")
    for lag in significant_lags:
        print(f"Lag {lag}: {lagged_corr[lag]}")
```

In [23]:

```
# Cross-correlation at different time lags
from statsmodels.tsa.stattools import ccf

lagged_corr_rain = ccf(df['wind_speed_km'], df['rain_mm'], unbiased=True)
plt.plot(lagged_corr_rain)
plt.xlabel('Lag')
plt.ylabel('Cross-Correlation')
plt.title('Time-Lagged Correlation of Wind Speed vs. Rainfall')
plt.show()

lagged_corr_temp_max = ccf(df['wind_speed_km'], df['temp_max_c'], unbiased=True)
plt.plot(lagged_corr_temp_max)
plt.xlabel('Lag')
plt.ylabel('Cross-Correlation')
plt.title('Time-Lagged Correlation of Wind Speed vs. Max Temperature')
plt.show()

lagged_corr_temp_min = ccf(df['wind_speed_km'], df['temp_min_c'], unbiased=True)
plt.plot(lagged_corr_temp_min)
plt.xlabel('Lag')
plt.ylabel('Cross-Correlation')
plt.title('Time-Lagged Correlation of Wind Speed vs. Min Temperature')
plt.show()
```




Time-Lagged Correlation of Wind Speed vs. Max Temperature
Cross-Correlation
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
0
250
500
750
1000
1250
1500
1750
Lag


Time-Lagged Correlation of Wind Speed vs. Min Temperature
Cross-Correlation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
250
500
750
1000
1250
1500
1750
Lag

In [24]:

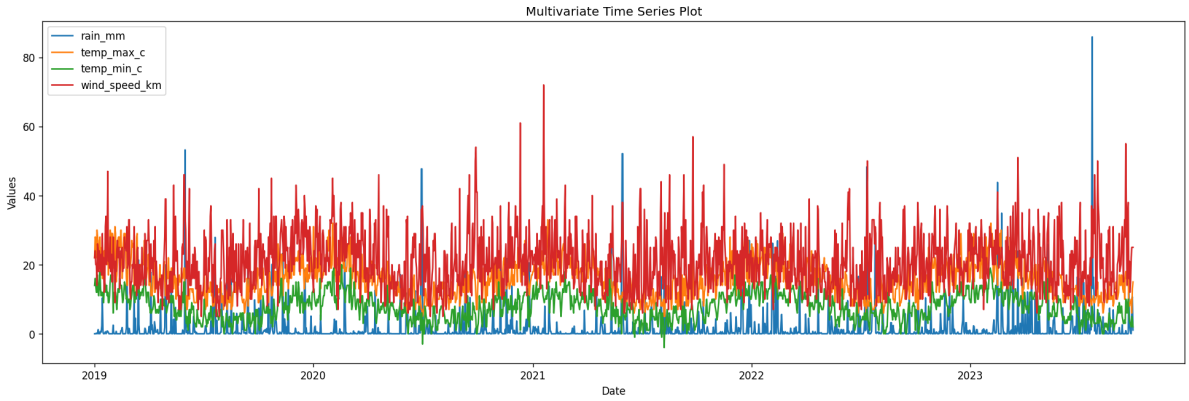
```
# Print significant lags for Rainfall vs. Wind Speed
print_significant_lags('Wind Speed vs. Rainfall',lagged_corr_rain)

# Print significant lags for Max Temperature vs. Wind Speed
print_significant_lags('Wind Speed vs. Max Temperature', lagged_corr_temp_max)

# Print significant lags for Min Temperature vs. Wind Speed
print_significant_lags('Wind Speed vs. Min Temperature', lagged_corr_temp_min)
```

```
Wind Speed vs. Rainfall Significant Lags with Cross-Correlation >= 0.
2:
Lag 0: 0.21057623629376487
Lag 1504: -0.2235678392529083
Lag 1569: 0.268114222439698
Lag 1714: -0.24578744547064052
Lag 1715: -0.2579198153085193
Wind Speed vs. Max Temperature Significant Lags with Cross-Correlation
>= 0.2:
Lag 1: 0.21137389684689656
Lag 159: -0.20619679307216088
Lag 312: 0.21076493187916523
Lag 330: 0.2097405111647191
Lag 332: 0.2007150962780972
Lag 1563: -0.21705739654481854
Lag 1568: -0.2042212818235656
Lag 1569: -0.2395269412670575
Lag 1577: -0.23022515011991376
Lag 1607: -0.25305070887481734
```

5. Time Series Overview

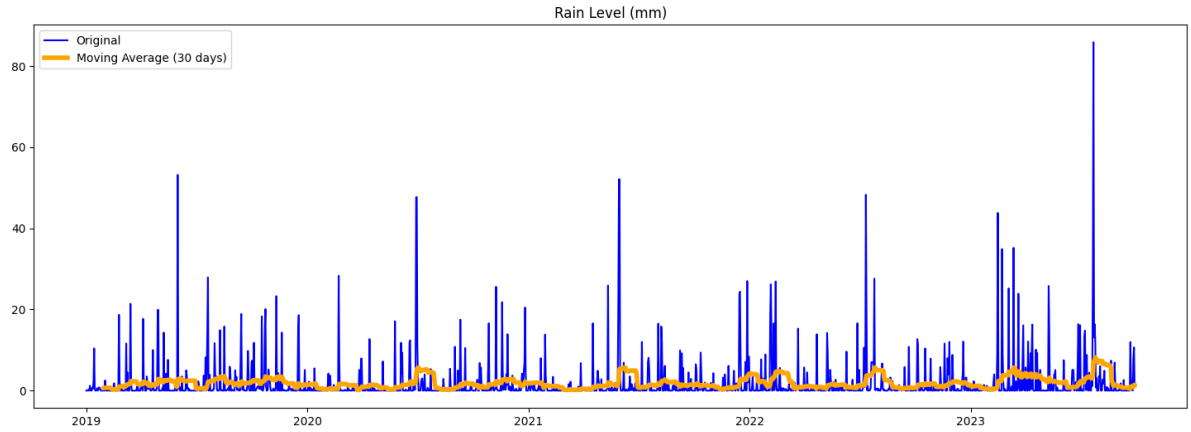
In [25]:

```
plt.figure(figsize=(20, 6),dpi=120)
for column in df.columns:
    plt.plot(df.index, df[column], label=column)

plt.title('Multivariate Time Series Plot')
plt.xlabel('Date')
plt.ylabel('Values')
plt.legend()
plt.show()
```



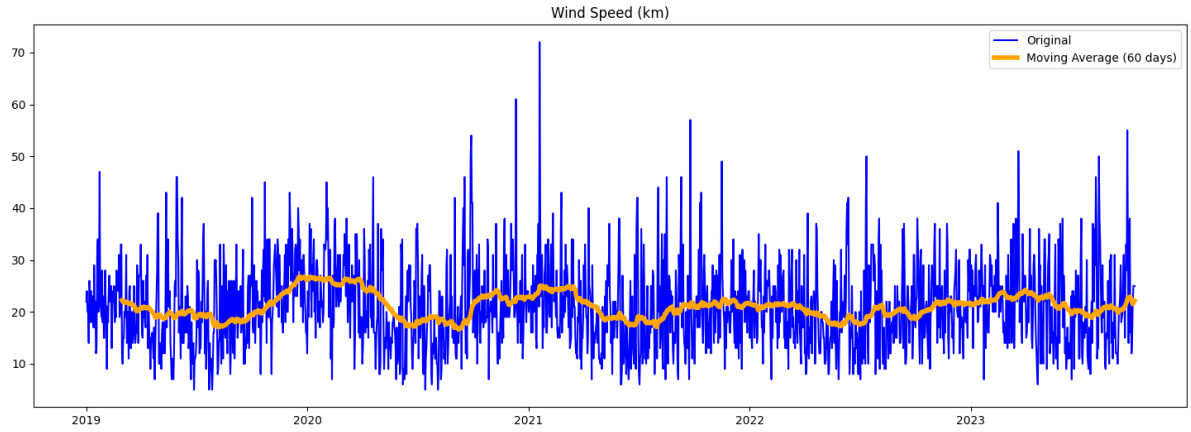
In [26]:

```
## visualization: time series line charts of rain_mm, temp_max_c, temp_min_c, wind_s
window_size = 30

data = df['rain_mm']
rolling_data = data.rolling(window=window_size).mean()

plt.figure(figsize=(18, 6))
plt.plot(data, label='Original', color='blue')
plt.plot(rolling_data, label=f'Moving Average ({window_size} days)', color='orange',
# Decorations
plt.title('Rain Level (mm)')
plt.legend()
plt.show()
```



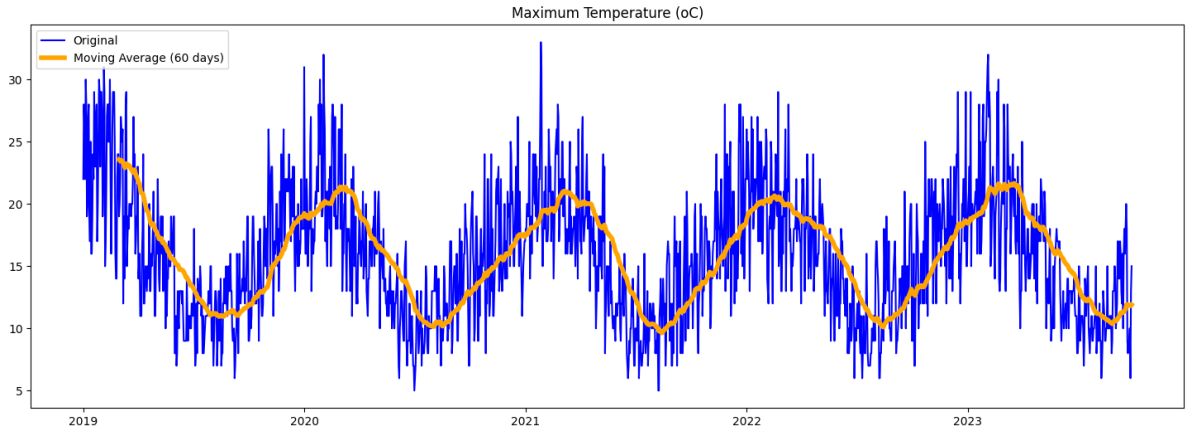
In [27]:

```
## visualization: time series line charts of rain_mm, temp_max_c, temp_min_c, wind_s
window_size = 60

data = df['wind_speed_km']
rolling_data = data.rolling(window=window_size).mean()

plt.figure(figsize=(18, 6))
plt.plot(data, label='Original', color='blue')
plt.plot(rolling_data, label=f'Moving Average ({window_size} days)', color='orange',
# Decorations
plt.title('Wind Speed (km)')
plt.legend()
plt.show()
```



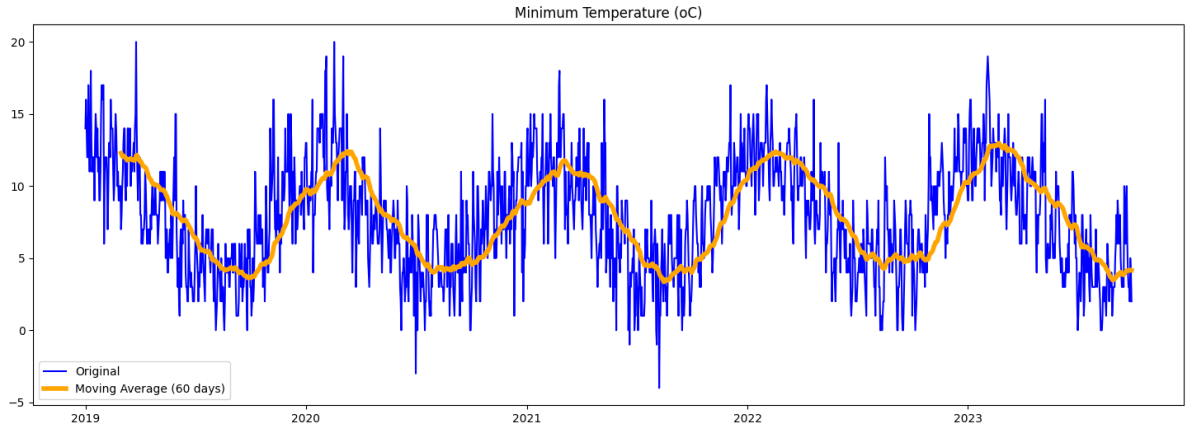
In [28]:

```
## visualization: time series line charts of rain_mm, temp_max_c, temp_min_c, wind_s
window_size = 60

data = df['temp_max_c']
rolling_data = data.rolling(window=window_size).mean()

plt.figure(figsize=(18, 6))
plt.plot(data, label='Original', color='blue')
plt.plot(rolling_data, label=f'Moving Average ({window_size} days)', color='orange',
# Decorations
plt.title('Maximum Temperature (oC)')
plt.legend()
plt.show()
```



In [29]:

```
## visualization: time series line charts of rain_mm, temp_max_c, temp_min_c, wind_s
window_size = 60

data = df['temp_min_c']
rolling_data = data.rolling(window=window_size).mean()

plt.figure(figsize=(18, 6))
plt.plot(data, label='Original', color='blue')
plt.plot(rolling_data, label=f'Moving Average ({window_size} days)', color='orange',
# Decorations
plt.title('Minimum Temperature (oC)')
plt.legend()
plt.show()
```

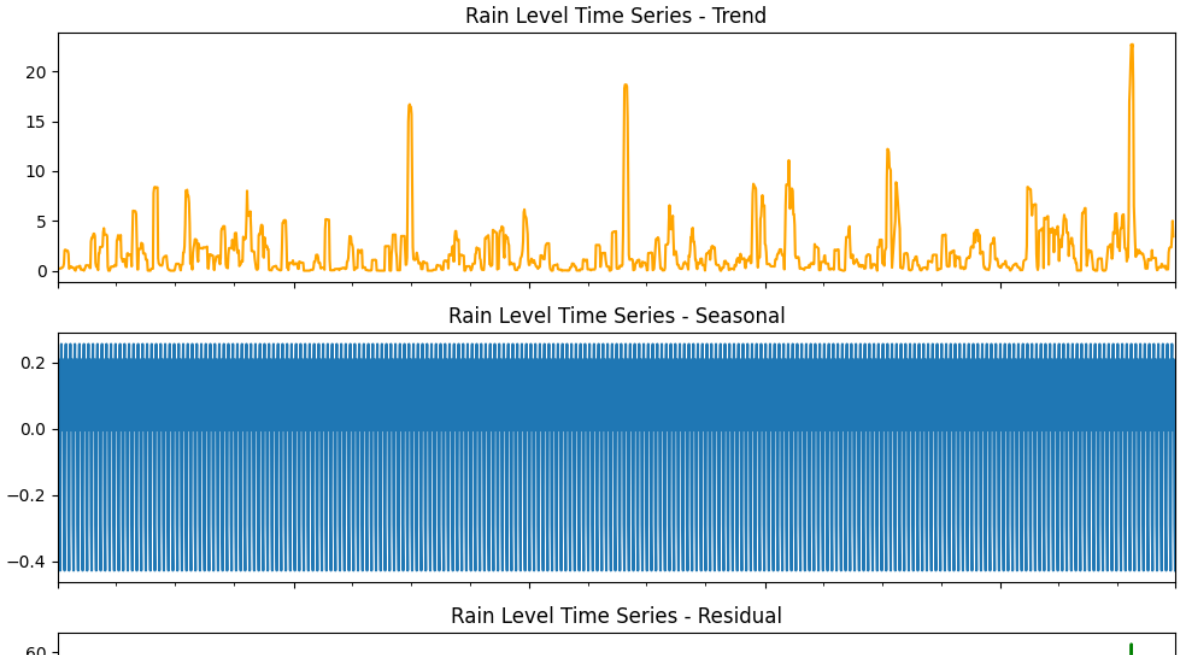


Step 3: Detailed Analysis Results

1. Seasonal Decomposition

In [30]:

```
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
from tabulate import tabulate

# Store summary statistics in a list
summary_data = []

titles = ['Rain Level Time Series', 'Max Temperature Time Series',
          'Min Temperature Time Series', 'Wind Speed Time Series']

for i, col in enumerate(df.columns):
    result = seasonal_decompose(df[col], model='additive') # period=365
    # Access the decomposed components
    trend = result.trend
    seasonal = result.seasonal
    residual = result.resid

    # Calculate summary statistics for each component
    trend_mean = trend.mean()
    seasonal_mean = seasonal.mean()
    residual_mean = residual.mean()

    trend_std = trend.std()
    seasonal_std = seasonal.std()
    residual_std = residual.std()

    # Append data to the summary list
    summary_data.append([col,
                         f"{trend_mean:.2f} ± {trend_std:.2f}",
                         f"{seasonal_mean:.2f} ± {seasonal_std:.2f}",
                         f"{residual_mean:.2f} ± {residual_std:.2f}"])

    # Specify the layout for subplots
    fig, axes = plt.subplots(3, 1, sharex=True, figsize=(10, 8))

    # Plot each component in a specific subplot
    #result.observed.plot(ax=axes[0], legend=False, title=titles[i], color='blue')
    trend.plot(ax=axes[0], legend=False, title=f'{titles[i]} - Trend', color='orange
    seasonal.plot(ax=axes[1], legend=False, title=f'{titles[i]} - Seasonal')
    residual.plot(ax=axes[2], legend=False, title=f'{titles[i]} - Residual', color='

    plt.tight_layout()
    plt.show()

# Create a table
table_headers = ["Feature", "Trend (Mean ± Std)", "Seasonal (Mean ± Std)", "Residual
table = tabulate(summary_data, headers=table_headers, tablefmt="pretty")

# Display the table
print(table)
```



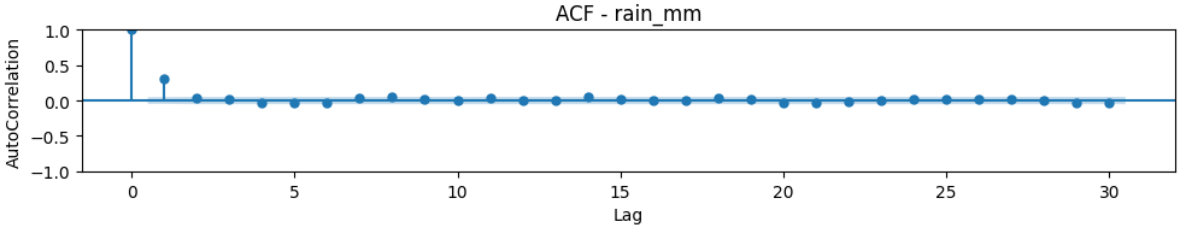
2. AR and MA terms

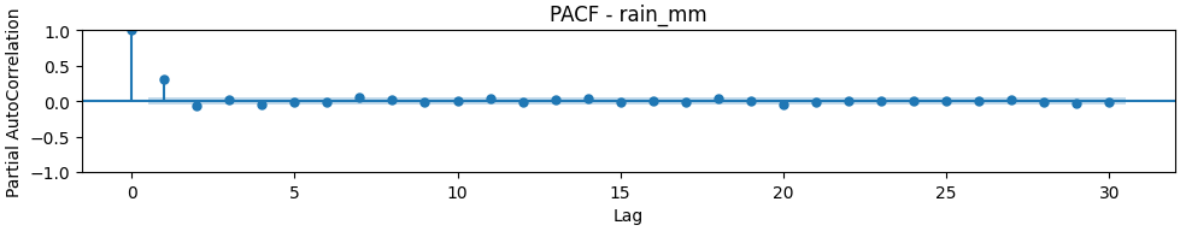
In [31]:

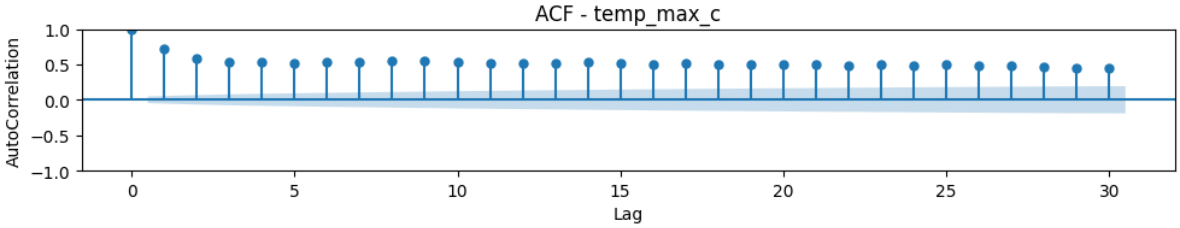
```
# Function to calculate ACF and PACF for a given time series
def calculate_acf_pacf(ts, lags=30):
    acf_vals = acf(ts, nlags=lags)
    pacf_vals = pacf(ts, nlags=lags)
    return acf_vals, pacf_vals

# Function to display numerical insights
def display_numerical_insights(acf_vals, pacf_vals, col):
    print(f"\nNumerical Insights for {col}:")
    print("Lag\tACF\t\tPACF")
    for lag in range(len(acf_vals)):
        print(f"{lag}\t{acf_vals[lag]:.4f}\t{pacf_vals[lag]:.4f}")
```

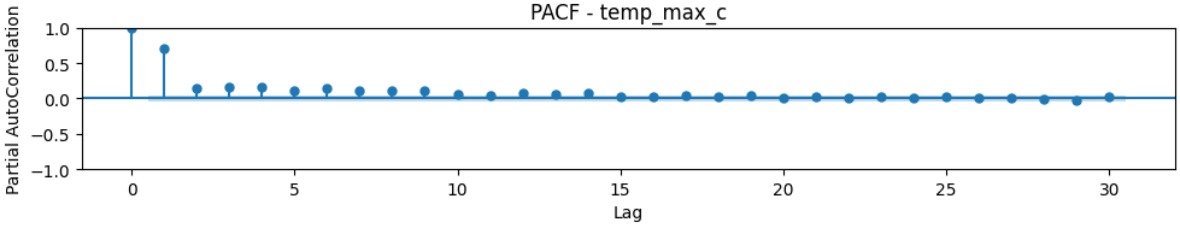
In [32]:

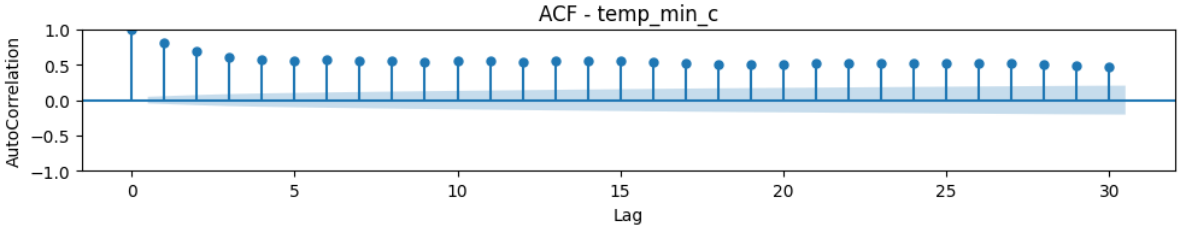
```
## AutoCorrelation Function (ACF) and Partial Autocorrelation (PACF)
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

import math

num_columns = len(df.columns)
num_rows = math.ceil(num_columns / 2)

fig, axes = plt.subplots(num_columns*2, 1, figsize=(10, 8 * num_rows))

# Flatten axes for iteration
axes = axes.flatten()

for i, col in enumerate(df.columns):
    plot_acf(df[col], lags=30, ax=axes[i*2], title=f'ACF - {col}')
    plot_pacf(df[col], lags=30, ax=axes[i*2 + 1], title=f'PACF - {col}')
    axes[i*2].set_xlabel('Lag')
    axes[i*2 + 1].set_xlabel('Lag')
    axes[i*2].set_ylabel('AutoCorrelation')
    axes[i*2 + 1].set_ylabel('Partial AutoCorrelation')

     # Calculate ACF and PACF values
    #acf_vals, pacf_vals = calculate_acf_pacf(df[col], lags=30)

    # Display numerical insights
    #display_numerical_insights(acf_vals, pacf_vals, col)

# Hide empty subplots
for i in range(num_columns, num_rows * 2):
    fig.delaxes(axes[i])

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

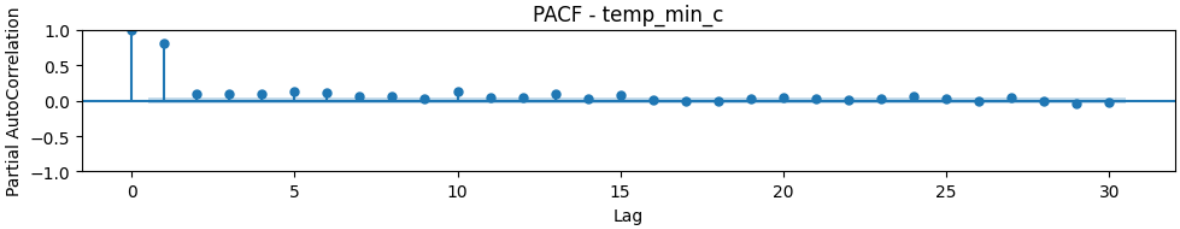


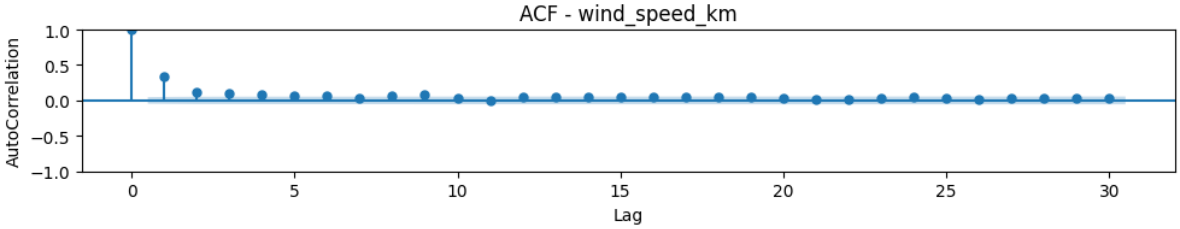


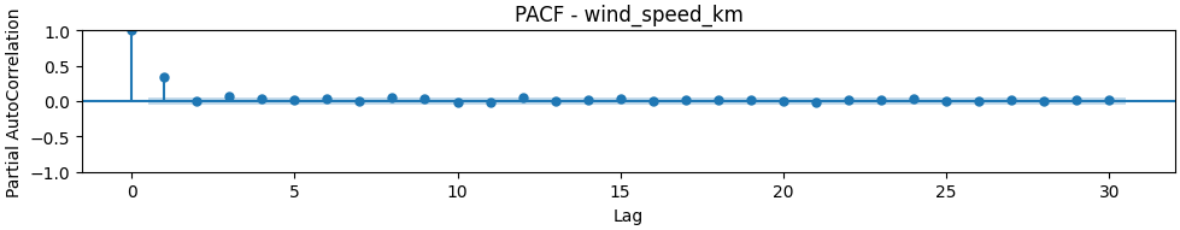












3. Seasonality

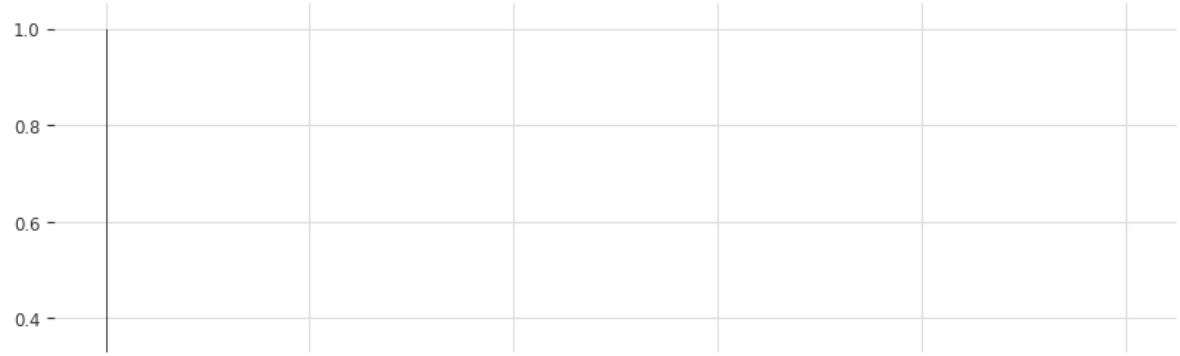
In [35]:

```
from darts.dataprocessing.transformers import (
    Scaler, MissingValuesFiller, Mapper, InvertibleMapper
)
from darts.dataprocessing import Pipeline
from darts.utils.statistics import check_seasonality, plot_residuals_analysis
from darts.metrics import coefficient_of_variation, mape, rmse
from darts.models import NaiveSeasonal, KalmanForecaster

from darts import TimeSeries
from darts.models import ExponentialSmoothing, ARIMA, AutoARIMA, Prophet

import warnings
warnings.filterwarnings("ignore")
import logging
logging.disable(logging.CRITICAL)
```

In [36]:

```
from darts.utils.statistics import check_seasonality, plot_acf
from tabulate import tabulate

# Initialize a list to store results
results = []

for name, column in df.items():
    series = TimeSeries.from_dataframe(df, value_cols=[name])

    for m in range(2, 365):
        is_seasonal, m = check_seasonality(series, m=m, alpha=0.05)
        if is_seasonal:
            break

    # Plot ACF with the relevant title
    plot_acf(series, m=m, alpha=0.05)

    result = {
        "Feature": name,
        "Is Seasonal": is_seasonal,
        "Seasonality Order": m
    }

    results.append(result)

# Create a table
table = tabulate(results, headers="keys", tablefmt="pretty")

# Display the table
print(table)
```

```
+---------------+-------------+--------------------+
|    Feature    | Is Seasonal | Seasonality Order  |
+---------------+-------------+--------------------+
|    rain_mm    |    True     |         8          |
|  temp_max_c   |    True     |         9          |
|  temp_min_c   |    True     |         6          |
| wind_speed_km |    True     |         9          |
+---------------+-------------+--------------------+
```



4. Integration (or Stationarity)

In [37]:

```
from statsmodels.tsa.stattools import adfuller
from prettytable import PrettyTable

def adfuller_test(series, signif=0.05, name=''):
    result = adfuller(series, autolag='AIC')
    stat = result[0]
    p_value = result[1]

    return [name, stat, p_value, 'Stationary' if p_value <= signif else 'Non-Statior

# Assuming df is your DataFrame with columns to be tested
results = []

for name, column in df.items():
    result = adfuller_test(column, name=column.name)
    results.append(result)

# Create a PrettyTable
table = PrettyTable(['Name', 'ADF Statistic', 'P-Value', 'Result'])
for result in results:
    table.add_row(result)

# Print the table
print(table)
```

```
+---------------+---------------------+-----------------------+-------
-----+
|      Name     |    ADF Statistic    |        P-Value        |   Resu
lt   |
+---------------+---------------------+-----------------------+-------
-----+
|    rain_mm    |  -25.98717818378997 |          0.0          | Statio
nary |
|   temp_max_c  | -3.3717124851837132 |  0.011961829953109075 | Statio
nary |
|   temp_min_c  | -3.5577025106126547 |  0.006623098004796129 | Statio
nary |
| wind_speed_km | -16.906623586500725 | 1.018104469707376e-29 | Statio
nary |
+---------------+---------------------+-----------------------+-------
-----+
```

Insights: time series of rain_mm, temp_max_c, temp_min_c, wind_speed_km are all stationary. They comply with the stationarity assumption in many time series models.

Note: non-stationarity can lead to biased estimates and unreliable forecasts. There are many approaches to make it stationary. They are differencing, de-trending, seasonal differencing, transformations, ARIMA modeling, or trend and seasonal decomposition techniques.

9. Co-integration Test

Assumption: all variables are integrated in the same order.

In [38]:

```
from statsmodels.tsa.vector_ar.vecm import coint_johansen
from tabulate import tabulate

# I(d=0): The time series is stationary (no differencing needed).
# I(d=1): The time series requires first-order differencing to become stationary.
# I(2): The time series requires second-order differencing, and so on.
# d (or lagged diff): no. of differencing, is the number of times this differencing

# Deterministic terms are constant terms or trends that are added to the model to ca
# det_order = -1: No deterministic terms are included. This assumes that there is no
# det_order = 0: A constant term is included in the model. This assumes that there i
# det_order = 1: Both a constant term and a linear trend are included in the model.

#If all variables are stationary, it suggests that there is no need to include deter

def cointegration_test(data, det_order, lagged_diff=0):
    """
    Perform Johansen's Cointegration Test and Report Summary
        -1: no deterministic terms
        0: constant term
        1: linear trend
    """
    result = coint_johansen(data, det_order, lagged_diff)

    traces_stat = result.lr1
    eigen_stat = result.lr2

    # alpha=0.05
    # {'0.90':0, '0.95':1, '0.99':2
    cvts = result.cvt[:, 1]

    # Summary
    headers = ['Name', 'Test Stat', 'C(95%)', 'Signif']
    table_data = []

    for col, trace, cvt in zip(df.columns, traces_stat, cvts):
        significance = trace > cvt
        table_data.append([col, f'{trace:.3f}', f'{cvt:.3f}', str(significance)])

    # Print tabulated summary
    print(tabulate(table_data, headers=headers, tablefmt='grid'))

# Assuming ADF has confirmed stationarity for all variables

print('Constant-term')
cointegration_test(df, 0, 0)
print('-----------------------------------------------------')

print('No constant/trend (deterministic)')
cointegration_test(df, -1, 0)
print('-----------------------------------------------------')

print('Linear trend')
cointegration_test(df, 1, 0)
```

```
Constant-term
+---------------+-------------+----------+----------+
| Name          |   Test Stat |   C(95%) | Signif   |
+===============+=============+==========+==========+
| rain_mm       |     2591.24 |   47.855 | True     |
+---------------+-------------+----------+----------+
| temp_max_c    |     1640.95 |   29.796 | True     |
+---------------+-------------+----------+----------+
| temp_min_c    |      852.153 |   15.494 | True     |
+---------------+-------------+----------+----------+
| wind_speed_km |      161.049 |    3.841 | True     |
+---------------+-------------+----------+----------+
------------------------------------------------------------
No constant/trend (deterministic)
+---------------+-------------+----------+----------+
| Name          |   Test Stat |   C(95%) | Signif   |
+===============+=============+==========+==========+
| rain_mm       |     2098.6  |   40.175 | True     |
+---------------+-------------+----------+----------+
| temp_max_c    |     1168.94 |   24.276 | True     |
+---------------+-------------+----------+----------+
| temp_min_c    |      386.451 |   12.321 | True     |
+---------------+-------------+----------+----------+
| wind_speed_km |       16.267 |    4.13  | True     |
+---------------+-------------+----------+----------+
------------------------------------------------------------
Linear trend
+---------------+-------------+----------+----------+
| Name          |   Test Stat |   C(95%) | Signif   |
+===============+=============+==========+==========+
| rain_mm       |     2598.45 |   55.246 | True     |
+---------------+-------------+----------+----------+
| temp_max_c    |     1645.41 |   35.012 | True     |
+---------------+-------------+----------+----------+
| temp_min_c    |      853.022 |   18.398 | True     |
+---------------+-------------+----------+----------+
| wind_speed_km |      161.782 |    3.841 | True     |
+---------------+-------------+----------+----------+
```

Step 4: Modeling and Forecasting Results

In [39]:

```
# forecast the next 30 days
nobs = 30
```

In [40]:

```
uni_df = df[['wind_speed_km']]
uni_df = uni_df.reset_index()
```

In [41]:

```
series = TimeSeries.from_dataframe(uni_df, time_col='time_index', value_cols='wind_s
```

Using MinMaxScaler as the distribution of every 4 features is not normal. The StandardScaler is not used here because it is only suitable for the data that has an approximately normal distribution.

In [42]:

```
from sklearn.preprocessing import MinMaxScaler

mm_scaler = MinMaxScaler()
scaler = Scaler(mm_scaler)
rescaled = scaler.fit_transform(series)

#back = scaler.inverse_transform(rescaled)
```

1. Univariate time series

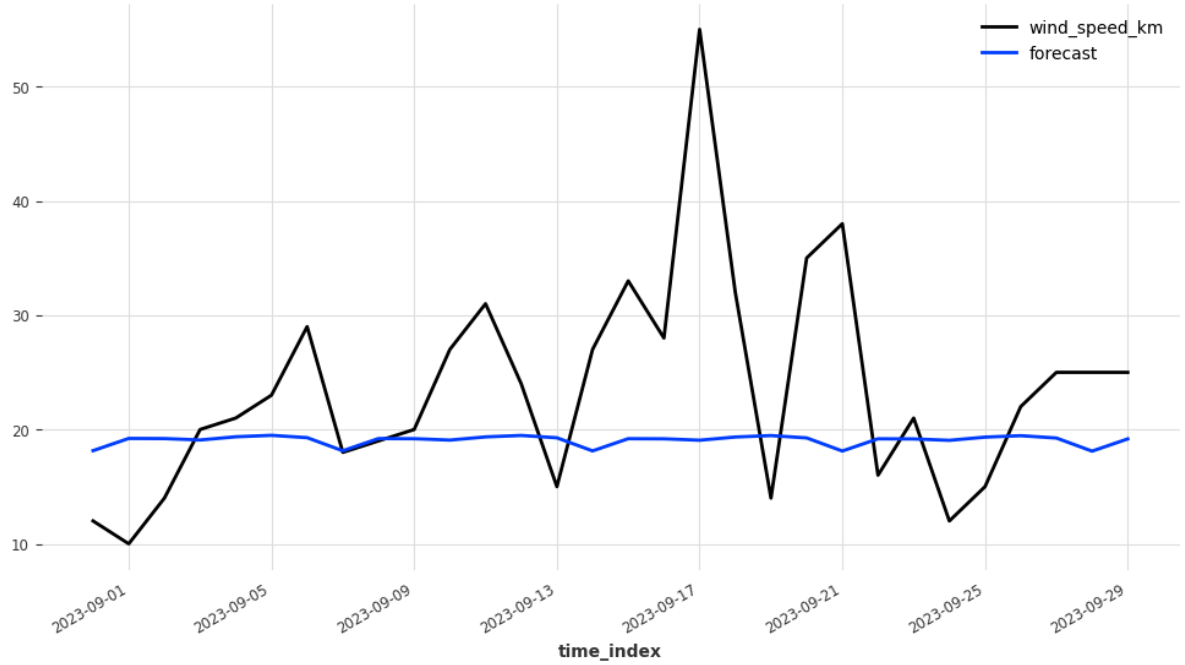
In [43]:

```
from darts.metrics import coefficient_of_variation, mae, rmse, mse, mape, smape

train, val = rescaled[:-nobs], rescaled[-nobs:]

# 1st try: Exponential Smoothing algorithm with applied Scaler
model = ExponentialSmoothing()
model.fit(train)

pred_scaled = model.predict(len(val))
pred_origin = scaler.inverse_transform(pred_scaled)
val_origin = scaler.inverse_transform(val)

from darts.metrics import coefficient_of_variation, mape, rmse
# Performance metrics
co_var = coefficient_of_variation(val_origin, pred_origin)
rmse_val = rmse(val_origin, pred_origin)
mae_val = mae(val_origin, pred_origin)
mse_val = mse(val_origin, pred_origin)
mape_val = mape(val_origin, pred_origin)
smape_val = smape(val_origin, pred_origin)

# Create a dictionary of metrics
metrics_dict = {
    'Coefficient of Variation': co_var,
    'RMSE': rmse_val,
    'MAE': mae_val,
    'MSE': mse_val,
    'MAPE': mape_val,
    'SMAPE': smape_val
}

# Convert the dictionary to a DataFrame
metrics_df = pd.DataFrame(list(metrics_dict.items()), columns=['Metric', 'Value'])

# Display the metrics table
print(tabulate(metrics_df, headers='keys', tablefmt='pretty', showindex=False))

# Plotting
plt.figure(figsize=(12, 6))
val_origin.plot()
pred_origin.plot(label='forecast', low_quantile=0.05, high_quantile=0.95)
plt.legend()
```

```
+--------------------------+--------------------+
|          Metric          |       Value        |
+--------------------------+--------------------+
| Coefficient of Variation | 43.69742803959475  |
|           RMSE           | 10.283461398651298 |
|           MAE            | 7.4815437881935285 |
|           MSE            | 105.7495783375513  |
|           MAPE           | 30.365351133099523 |
|          SMAPE           | 32.30269038967543  |
+--------------------------+--------------------+
```

Out[43]:

```
<matplotlib.legend.Legend at 0x7f657881b070>
```




```
# transformation
toDailyAverage = InvertibleMapper(
    fn = lambda timestamp, x: x / timestamp.days_in_month,
    inverse_fn = lambda timestamp, x: x * timestamp.days_in_month
)
dailyAverage = toDailyAverage.transform(series.copy())
```

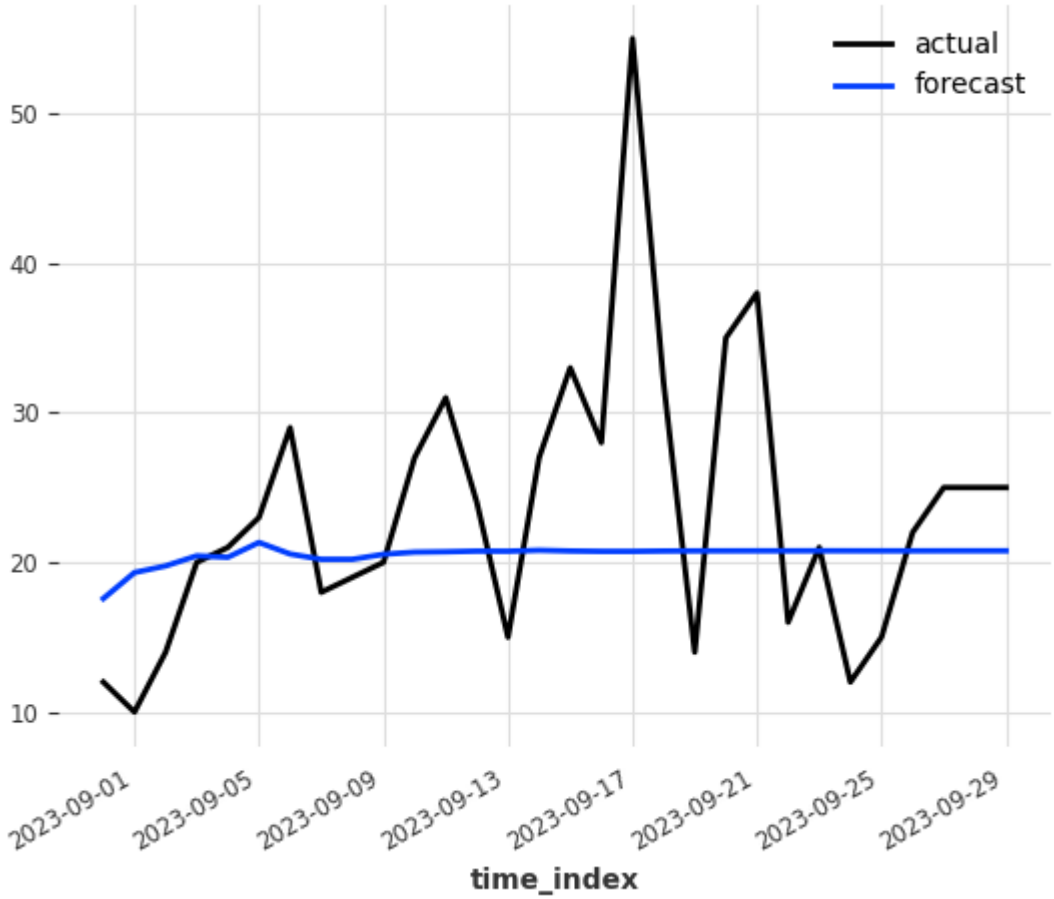
In [45]:

```
train_, val_ = series[:-nobs], series[-nobs:]

# 2nd try: ARIMA with Scaler
pipeline = Pipeline([toDailyAverage, scaler])
transformed_train = pipeline.fit_transform(train_)
transformed_test = pipeline.transform(val_)

model = ARIMA(p=1, d=0, q=0, seasonal_order=(1, 0, 0, 9))
model.fit(transformed_train)
dailyavg_forecast = model.predict(nobs)

# Inverse the transformation
# Here the forecast is stochastic; so we take the median value
pred_origin = pipeline.inverse_transform(dailyavg_forecast)
val_origin = pipeline.inverse_transform(transformed_test)

from darts.metrics import coefficient_of_variation, mape, rmse
# Performance metrics
co_var = coefficient_of_variation(val_origin, pred_origin)
rmse_val = rmse(val_origin, pred_origin)
mae_val = mae(val_origin, pred_origin)
mse_val = mse(val_origin, pred_origin)
mape_val = mape(val_origin, pred_origin)
smape_val = smape(val_origin, pred_origin)

# Create a dictionary of metrics
metrics_dict = {
    'Coefficient of Variation': co_var,
    'RMSE': rmse_val,
    'MAE': mae_val,
    'MSE': mse_val,
    'MAPE': mape_val,
    'SMAPE': smape_val
}

# Convert the dictionary to a DataFrame
metrics_df = pd.DataFrame(list(metrics_dict.items()), columns=['Metric', 'Value'])

# Display the metrics table
print(tabulate(metrics_df, headers='keys', tablefmt='pretty', showindex=False))

# Plotting
val_origin.plot(label="actual")
pred_origin.plot(label="forecast")
plt.legend()
```

```
+--------------------------+--------------------+
|          Metric          |       Value        |
+--------------------------+--------------------+
| Coefficient of Variation | 40.28623025161607  |
|           RMSE           | 9.480692852546982  |
|           MAE            | 6.809966343098013  |
|           MSE            | 89.88353696433542  |
|           MAPE           | 28.95903946110463  |
|          SMAPE           | 29.087637551620805 |
+--------------------------+--------------------+
```

Out[45]:

```
<matplotlib.legend.Legend at 0x7f657892b280>
```



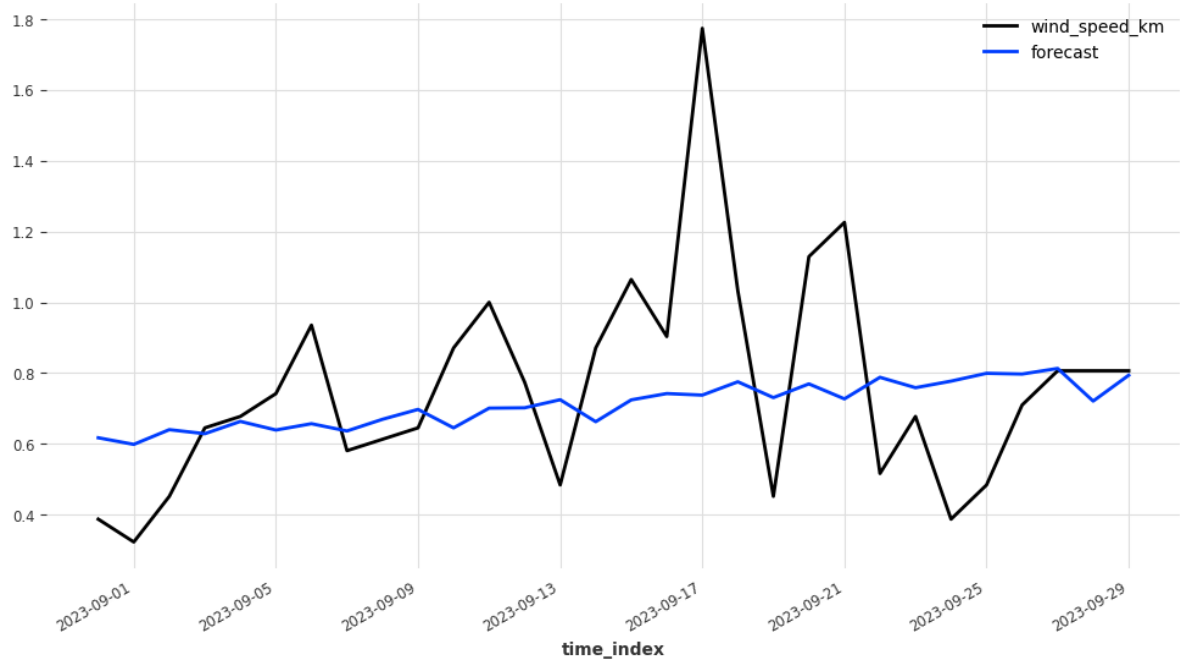
In [46]:

```
from tabulate import tabulate
from darts.metrics import coefficient_of_variation, mae, rmse, mse, mape, smape

train, val = rescaled[:-nobs], rescaled[-nobs:]

# 3rd try: Facebook Prophet with Scaler
model = Prophet(
    add_seasonalities={
        'name':"quarterly_seasonality",
        'seasonal_periods':9,
        'fourier_order':10
    }
)
model.fit(train)

pred_scaled = model.predict(len(val))
pred_origin = scaler.inverse_transform(pred_scaled)
val_origin = scaler.inverse_transform(val)

from darts.metrics import coefficient_of_variation, mape, rmse
# Performance metrics
co_var = coefficient_of_variation(val_origin, pred_origin)
rmse_val = rmse(val_origin, pred_origin)
mae_val = mae(val_origin, pred_origin)
mse_val = mse(val_origin, pred_origin)
mape_val = mape(val_origin, pred_origin)
smape_val = smape(val_origin, pred_origin)

# Create a dictionary of metrics
metrics_dict = {
    'Coefficient of Variation': co_var,
    'RMSE': rmse_val,
    'MAE': mae_val,
    'MSE': mse_val,
    'MAPE': mape_val,
    'SMAPE': smape_val
}

# Convert the dictionary to a DataFrame
metrics_df = pd.DataFrame(list(metrics_dict.items()), columns=['Metric', 'Value'])

# Display the metrics table
print(tabulate(metrics_df, headers='keys', tablefmt='pretty', showindex=False))

# Plotting
plt.figure(figsize=(12, 6))
val_origin.plot()
pred_origin.plot(label='forecast', low_quantile=0.05, high_quantile=0.95)
plt.legend()
```

```
+--------------------------+----------------------+
|          Metric          |        Value         |
+--------------------------+----------------------+
| Coefficient of Variation |  38.74093400952075   |
|           RMSE           | 0.29409784312603926  |
|           MAE            | 0.21668552409649275  |
|           MSE            | 0.08649354133138841  |
|           MAPE           | 30.804431416894566   |
|          SMAPE           |  28.71741812811165   |
+--------------------------+----------------------+
```

Out[46]:

```
<matplotlib.legend.Legend at 0x7f657810f610>
```



2. Multivariate time series

In [47]:

```
# Import Statsmodels
#from statsmodels.tsa.api import VAR
from statsmodels.tsa.vector_ar.var_model import VAR
```

1. find the lag order returning the best fit

In [48]:

```
model = VAR(df)

best_criteria_aic = np.inf
best_criteria_bic = np.inf
best_lag_aic = 0
best_lag_bic = 0

for lag_order in range(1, 100):
    result = model.fit(lag_order)

    aic = result.aic
    bic = result.bic
    fpe = result.fpe
    hqic = result.hqic

    if aic < best_criteria_aic:
        best_criteria_aic = aic
        best_lag_aic = lag_order

    if bic < best_criteria_bic:
        best_criteria_bic = bic
        best_lag_bic = lag_order

    #print('Lag Order =', lag_order, ' - AIC : ', round(aic, 3), ' BIC : ', round(bi
print('------------------------------------------------------')
print("Best Lag Order by AIC:", best_lag_aic)
print('------------------------------------------------------')
print("Best Lag Order by BIC:", best_lag_bic)
```

```
------------------------------------------------------
Best Lag Order by AIC: 10
------------------------------------------------------
Best Lag Order by BIC: 1
```

Conclusion: let's go with the lag order of 10

2. Model fitting and predicting

In [49]:

```
lag_order = best_lag_aic
#lag_order = best_lag_bic
model_fitted = model.fit(lag_order)
```

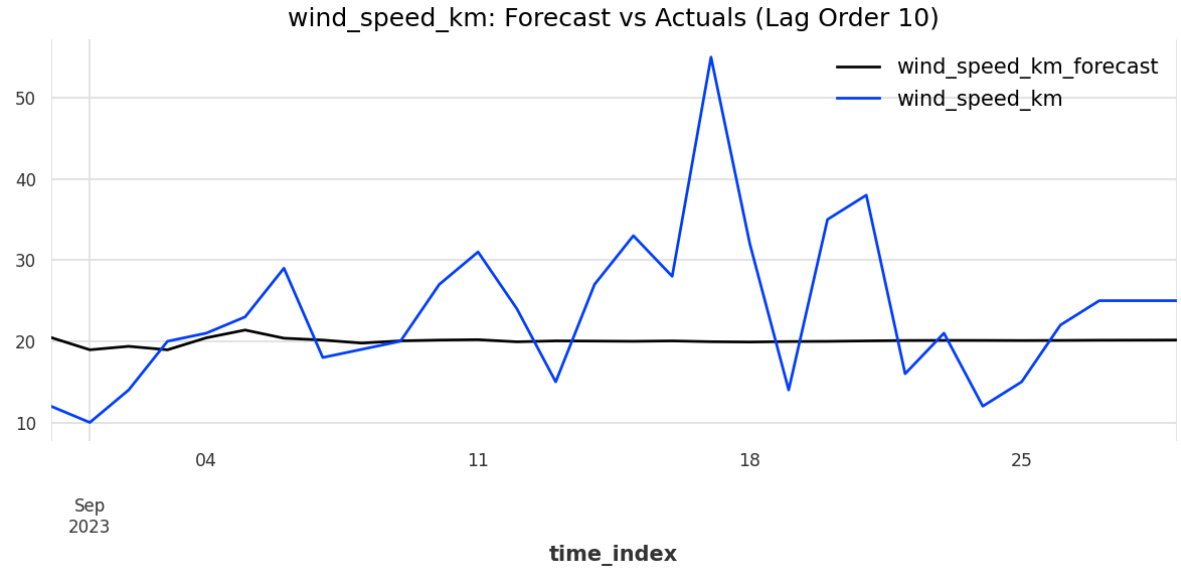
3. Forcasting

In [50]:

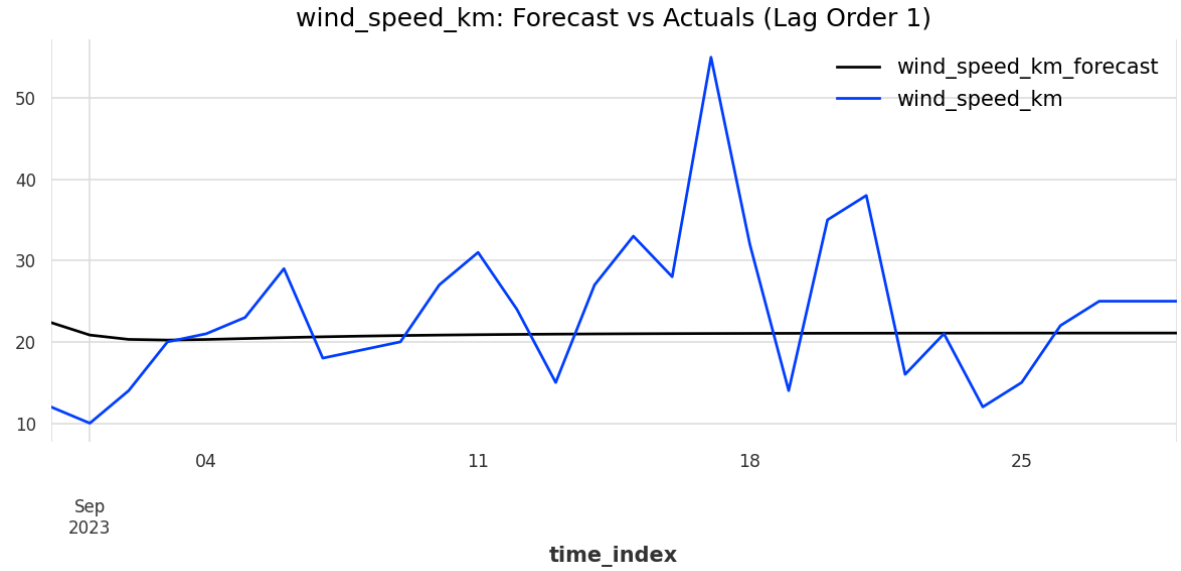
```
from prettytable import PrettyTable
from tabulate import tabulate
from statsmodels.stats.stattools import durbin_watson
from darts.metrics import coefficient_of_variation, mape, rmse, mae, mse, smape
import matplotlib.pyplot as plt

# Assuming best_lag_aic and best_lag_bic are calculated earlier
lag_order_aic = best_lag_aic
lag_order_bic = best_lag_bic

# Dictionary to store metrics for each lag order
metrics_data = {}

for lag_order in [lag_order_aic, lag_order_bic]:
    # Fit the model for the current lag order
    model_fitted = model.fit(lag_order)

    # Calculate Durbin-Watson statistic
    out = durbin_watson(model_fitted.resid)

    # Create a table
    table = PrettyTable()
    table.field_names = ['Column', 'Durbin-Watson']

    # Add data to the table
    for col, val in zip(df.columns, out):
        table.add_row([col, f'{val:.2f}'])

    # Print the table
    print(f'\nDurbin-Watson Statistic for Lag Order {lag_order}:')
    print(table)

    # Input data for forecasting
    forecast_input = df.values[-nobs:]

    # Forecast for the next nobs days
    fc = model_fitted.forecast(y=forecast_input, steps=nobs)
    df_forecast = pd.DataFrame(
        fc, index=df.index[-nobs:],
        columns=['rain_mm_forecast', 'temp_max_c_forecast', 'temp_min_c_forecast', '
    )

    # Plot forecast vs actual for wind_speed_km
    fig, ax = plt.subplots(dpi=150, figsize=(8, 4))
    df_forecast['wind_speed_km_forecast'].plot(legend=True, ax=ax).autoscale(axis='x
    df[['wind_speed_km']][-nobs:].plot(legend=True, ax=ax)
    ax.set_title(f"wind_speed_km: Forecast vs Actuals (Lag Order {lag_order})")
    ax.xaxis.set_ticks_position('none')
    ax.yaxis.set_ticks_position('none')
    ax.spines["top"].set_alpha(0)
    ax.tick_params(labelsize=8)
    plt.tight_layout()
    plt.show()

    # Performance metrics
    val_origin = TimeSeries.from_dataframe(
        df[['wind_speed_km']][-nobs:], value_cols='wind_speed_km'
    )
    pred_origin = TimeSeries.from_dataframe(
```

```
        df_forecast, value_cols='wind_speed_km_forecast'
    )

    # Calculate performance metrics
    co_var = coefficient_of_variation(val_origin, pred_origin)
    rmse_val = rmse(val_origin, pred_origin)
    mae_val = mae(val_origin, pred_origin)
    mse_val = mse(val_origin, pred_origin)
    mape_val = mape(val_origin, pred_origin)
    smape_val = smape(val_origin, pred_origin)

    # Store metrics in the dictionary
    metrics_data[f'Lag Order {lag_order}'] = {
        'Coefficient of Variation': co_var,
        'RMSE': rmse_val,
        'MAE': mae_val,
        'MSE': mse_val,
        'MAPE': mape_val,
        'SMAPE': smape_val
    }

# Convert the dictionary to a DataFrame
metrics_df = pd.DataFrame(metrics_data).T.reset_index()
metrics_df.rename(columns={'index': 'Lag Order'}, inplace=True)

# Display the metrics table
print('\nPerformance Metrics Comparison:')
print(tabulate(metrics_df, headers='keys', tablefmt='pretty', showindex=False))
```

```
Durbin-Watson Statistic for Lag Order 10:
+---------------+---------------+
|     Column    | Durbin-Watson |
+---------------+---------------+
|    rain_mm    |      2.00     |
|   temp_max_c  |      2.00     |
|   temp_min_c  |      2.01     |
| wind_speed_km |      2.00     |
+---------------+---------------+
```



```
Durbin-Watson Statistic for Lag Order 1:
+---------------+---------------+
|     Column    | Durbin-Watson |
+---------------+---------------+
|    rain_mm    |      1.95     |
|   temp_max_c  |      2.08     |
|   temp_min_c  |      2.12     |
| wind_speed_km |      1.99     |
+---------------+---------------+
```




```
Performance Metrics Comparison:
+--------------+--------------------------+--------------------+-------
------------+--------------------+--------------------+----------------
----+
|  Lag Order   | Coefficient of Variation |        RMSE        |
MAE         |        MSE         |        MAPE        |       SMAPE
|
+--------------+--------------------------+--------------------+-------
------------+--------------------+--------------------+----------------
----+
| Lag Order 10 |    41.72775841284079     | 9.819932479821867 | 7.1022
47025076247 | 96.43107390826044 | 29.881301062282787 | 30.479441419004
683 |
| Lag Order 1  |    40.800301839160696    | 9.601671032815817 | 7.0255
15432799921 | 92.19208662241435 | 31.133009107735475 | 30.109418749090
477 |
+--------------+--------------------------+--------------------+-------
------------+--------------------+--------------------+----------------
----+
```