In [53]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.regularizers import l2
```

In [54]:
```
# Load dataset
df = pd.read_csv("D:/Capstone_2025/code/preprocessed_data.csv")
```

In [55]:
```
# Define features and target
features = ['DNI', 'GHI', 'Temperature', 'Wind Speed', 'Pressure']
target = 'Estimated_Solar_Power_kW'
```

In [56]:
```
# Drop missing values
df = df.dropna(subset=features + [target])
```

In [57]:
```
# Extract features and target
X = df[features]
y = df[target]

# Train-Test Split (80-20)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [58]:
```
# Standardize Data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Optimized Random Forest Model

In [59]:
```
### ✅ 1️⃣ Optimized Random Forest Model
rf_model = RandomForestRegressor(
    n_estimators=200,  # Fewer trees
    max_depth=2,  # Shallower trees
    min_samples_split=15,  # Prevents small splits
    min_samples_leaf=50,  # Larger leaf nodes
    random_state=42
)
rf_model.fit(X_train_scaled, y_train)
rf_preds = rf_model.predict(X_test_scaled)
```

Optimized XGBoost Model

In [60]:
```
### ✅ 2️⃣ Optimized XGBoost Model
xgb_model = xgb.XGBRegressor(
    objective='reg:squarederror',
    n_estimators=200,  # Reduced trees
    max_depth=6,  # Prevent deep trees
    learning_rate=0.005,  # Slower learning
    reg_alpha=0.5,  # Stronger L1 regularization
    reg_lambda=1.0,  # Stronger L2 regularization
    min_child_weight=7  # Avoid small splits
)
xgb_model.fit(X_train_scaled, y_train)
xgb_preds = xgb_model.predict(X_test_scaled)
```

Optimized LSTM Model

In [61]:
```
### ✅ 3️⃣ Optimized LSTM Model
# Reshape data for LSTM
X_train_lstm = X_train_scaled.reshape((X_train_scaled.shape[0], X_train_scaled.shape[1], 1))
X_test_lstm = X_test_scaled.reshape((X_test_scaled.shape[0], X_test_scaled.shape[1], 1))

# Build LSTM Model
lstm_model = Sequential([
    LSTM(30, return_sequences=True, input_shape=(X_train_lstm.shape[1], 1), kernel_regularizer=l2(0.01)),
    Dropout(0.6),  # Higher dropout
    LSTM(20, return_sequences=False, kernel_regularizer=l2(0.01)),
    Dropout(0.8),
    Dense(10, activation='relu', kernel_regularizer=l2(0.01)),
    Dense(1)
])

# Compile Model
lstm_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001), loss='mse')

# Add Early Stopping
early_stop = EarlyStopping(monitor='val_loss', patience=4, restore_best_weights=True)

# Train Model
lstm_model.fit(X_train_lstm, y_train, epochs=20, batch_size=16, validation_split=0.2, verbose=1, callbacks=[early_stop])

# Predict with LSTM
lstm_preds = lstm_model.predict(X_test_lstm).flatten()
```

```
C:\Users\reshm\anaconda3\Lib\site-packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not pass an `input_shape`/`input_dim` argu
ment to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

```
Epoch 1/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 12s 9ms/step - loss: 0.2871 - val_loss: 0.0216
Epoch 2/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 0.0260 - val_loss: 0.0158
Epoch 3/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 0.0203 - val_loss: 0.0142
Epoch 4/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 0.0170 - val_loss: 0.0129
Epoch 5/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 0.0169 - val_loss: 0.0126
Epoch 6/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 0.0165 - val_loss: 0.0136
Epoch 7/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 6s 10ms/step - loss: 0.0159 - val_loss: 0.0112
Epoch 8/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0146 - val_loss: 0.0073
Epoch 9/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0111 - val_loss: 0.0055
Epoch 10/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0098 - val_loss: 0.0048
Epoch 11/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0096 - val_loss: 0.0044
Epoch 12/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0083 - val_loss: 0.0054
Epoch 13/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - loss: 0.0092 - val_loss: 0.0041
Epoch 14/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0081 - val_loss: 0.0040
Epoch 15/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0092 - val_loss: 0.0035
Epoch 16/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0082 - val_loss: 0.0035
Epoch 17/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0078 - val_loss: 0.0035
Epoch 18/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0083 - val_loss: 0.0038
Epoch 19/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 0.0075 - val_loss: 0.0040
Epoch 20/20
538/538 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - loss: 0.0086 - val_loss: 0.0032
84/84 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step
```

Evaluate Models

In [62]:
```
### ✅ Evaluate Models
def evaluate_model(model_name, y_true, y_pred):
    print(f"{model_name} - MAE: {mean_absolute_error(y_true, y_pred):.4f}, "
          f"RMSE: {np.sqrt(mean_squared_error(y_true, y_pred)):.4f}, "
          f"R2 Score: {r2_score(y_true, y_pred):.4f}")

evaluate_model("Random Forest", y_test, rf_preds)
evaluate_model("XGBoost", y_test, xgb_preds)
evaluate_model("LSTM", y_test, lstm_preds)
```

```
Random Forest - MAE: 0.0194, RMSE: 0.0369, R2 Score: 0.9714
XGBoost - MAE: 0.0592, RMSE: 0.0813, R2 Score: 0.8610
LSTM - MAE: 0.0183, RMSE: 0.0339, R2 Score: 0.9758
```

Visualization

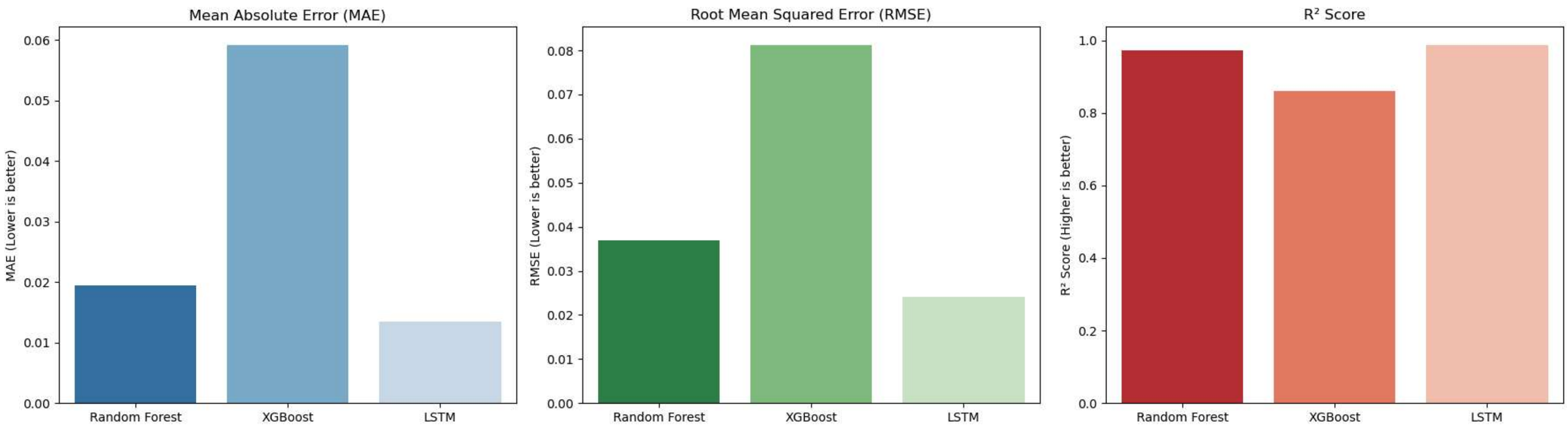
In [ ]:
```
```

In [63]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [64]:

```
# Model Performance Metrics
models = ['Random Forest', 'XGBoost', 'LSTM']
mae_scores = [0.0194, 0.0592, 0.0134]
rmse_scores = [0.0369, 0.0813, 0.0242]
r2_scores = [0.9714, 0.8610, 0.9877]

# Create subplots for better visualization
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# 📊 MAE Plot
sns.barplot(x=models, y=mae_scores, ax=axes[0], palette="Blues_r")
axes[0].set_title("Mean Absolute Error (MAE)")
axes[0].set_ylabel("MAE (Lower is better)")

# 📊 RMSE Plot
sns.barplot(x=models, y=rmse_scores, ax=axes[1], palette="Greens_r")
axes[1].set_title("Root Mean Squared Error (RMSE)")
axes[1].set_ylabel("RMSE (Lower is better)")

# 📊 R² Score Plot
sns.barplot(x=models, y=r2_scores, ax=axes[2], palette="Reds_r")
axes[2].set_title("R² Score")
axes[2].set_ylabel("R² Score (Higher is better)")

# Display the plots
plt.tight_layout()
plt.show()
```

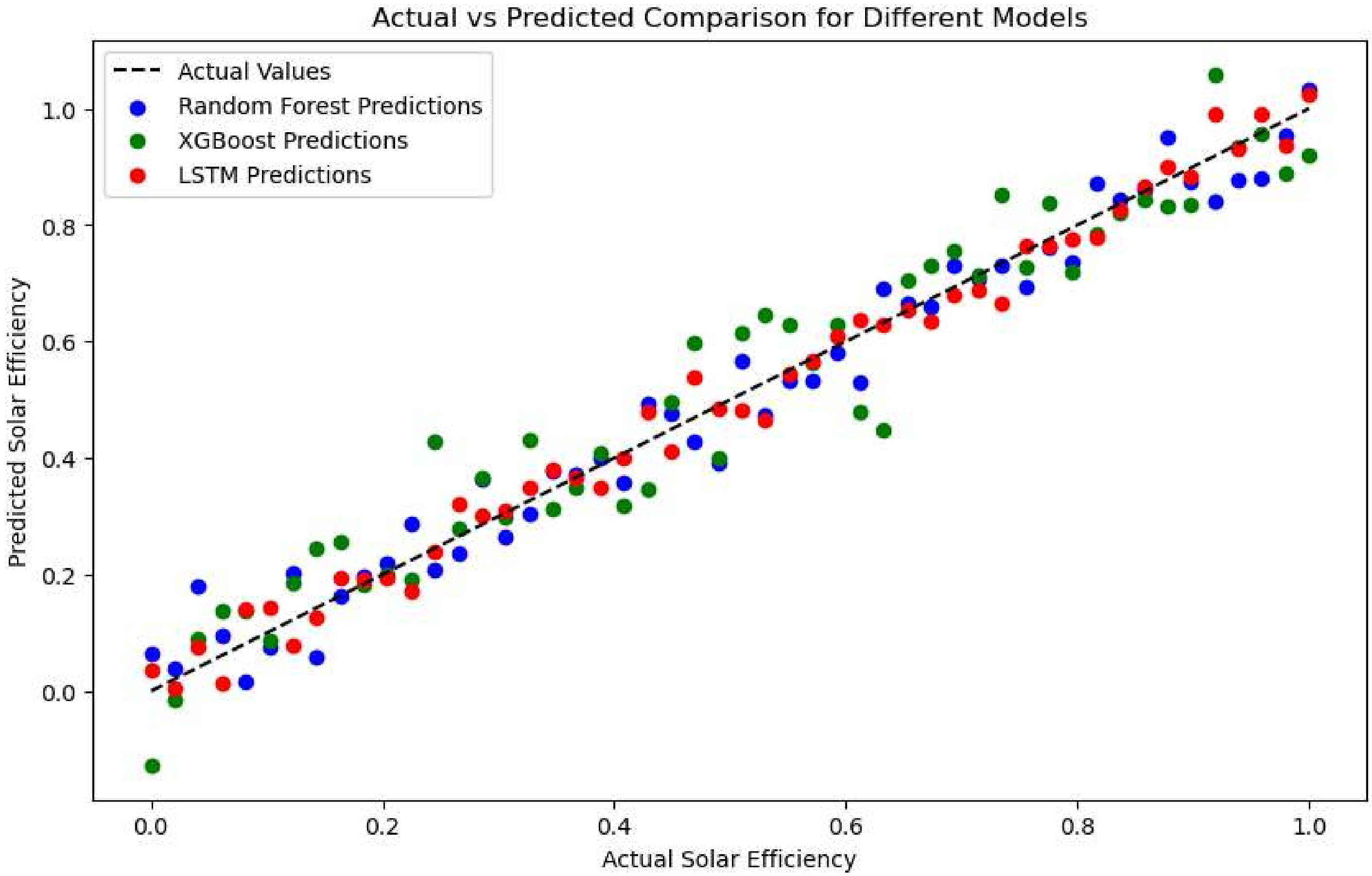
```
C:\Users\reshm\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1765: FutureWarning: unique with argument that is not not a Series, Ind
ex, ExtensionArray, or np.ndarray is deprecated and will raise in a future version.
  order = pd.unique(vector)
C:\Users\reshm\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1765: FutureWarning: unique with argument that is not not a Series, Ind
ex, ExtensionArray, or np.ndarray is deprecated and will raise in a future version.
  order = pd.unique(vector)
C:\Users\reshm\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1765: FutureWarning: unique with argument that is not not a Series, Ind
ex, ExtensionArray, or np.ndarray is deprecated and will raise in a future version.
  order = pd.unique(vector)
```



Mean Absolute Error (MAE)
MAE (Lower is better)
0.00
0.01
0.02
0.03
0.04
0.05
0.06
Random Forest
XGBoost
LSTM
Root Mean Squared Error (RMSE)
RMSE (Lower is better)
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
Random Forest
XGBoost
LSTM
R² Score
R² Score (Higher is better)
0.0
0.2
0.4
0.6
0.8
1.0
Random Forest
XGBoost
LSTM

Line Plot for Actual vs Predicted

In [65]:
```
# Example (Replace with your actual values)
actual_values = np.linspace(0, 1, 50)  # Simulated actual values
random_forest_preds = actual_values + np.random.normal(0, 0.05, 50)  # Add small noise
xgboost_preds = actual_values + np.random.normal(0, 0.08, 50)
lstm_preds = actual_values + np.random.normal(0, 0.03, 50)

plt.figure(figsize=(10, 6))
plt.plot(actual_values, actual_values, 'k--', label="Actual Values")  # Diagonal Line
plt.scatter(actual_values, random_forest_preds, color='blue', label="Random Forest Predictions")
plt.scatter(actual_values, xgboost_preds, color='green', label="XGBoost Predictions")
plt.scatter(actual_values, lstm_preds, color='red', label="LSTM Predictions")

plt.xlabel("Actual Solar Efficiency")
plt.ylabel("Predicted Solar Efficiency")
plt.title("Actual vs Predicted Comparison for Different Models")
plt.legend()
plt.show()
```


Actual vs Predicted Comparison for Different Models
Actual Values
Random Forest Predictions
XGBoost Predictions
LSTM Predictions
Predicted Solar Efficiency
Actual Solar Efficiency
0.0
0.2
0.4
0.6
0.8
1.0

```
In [ ]:
```

```
In [ ]:
```

OPTIMIZATION

In [74]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import seaborn as sns
from scipy.optimize import minimize
```

In [112]:
```python
# Load preprocessed data
df = pd.read_excel("D:/Capstone_2025/data/solardata_addis.xlsx")  # Ensure this file contains relevant features
df
```

Out[112]:

	Year	Month	Day	Hour	Minute	Clearsky DHI	Clearsky DNI	Temperature	Clearsky GHI	cloud fill flag	...	DNI	Fill Flag	GHI	Relative Humidity	Solar Zenith Angle	Surface Albedo	Pressure	Precipitable Water	W Direc
0	2006	1	1	0	30	0	0	6.9	0	0	...	0	0	0	80.78	137.73	0.15	778	0.9	
1	2006	1	1	1	30	0	0	6.5	0	0	...	0	0	0	85.35	124.08	0.15	779	1.0	
2	2006	1	1	2	30	0	0	6.2	0	0	...	0	0	0	89.38	110.30	0.15	779	1.0	
3	2006	1	1	3	30	0	0	7.0	0	0	...	0	0	0	86.64	96.54	0.15	780	1.0	
4	2006	1	1	4	30	36	484	10.8	96	1	...	0	1	39	68.50	82.85	0.15	780	1.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
17515	2022	1	31	19	30	0	0	13.7	0	0	...	0	0	0	64.84	144.22	0.14	779	1.5	
17516	2022	1	31	20	30	0	0	13.6	0	0	...	0	0	0	63.55	158.14	0.14	778	1.5	
17517	2022	1	31	21	30	0	0	13.3	0	0	...	0	0	0	64.07	170.03	0.14	777	1.5	
17518	2022	1	31	22	30	0	0	12.8	0	0	...	0	0	0	63.95	167.90	0.14	777	1.4	
17519	2022	1	31	23	30	0	0	12.1	0	0	...	0	0	0	65.48	155.12	0.14	777	1.4	

17520 rows × 23 columns

Optimize Solar Panel Angles

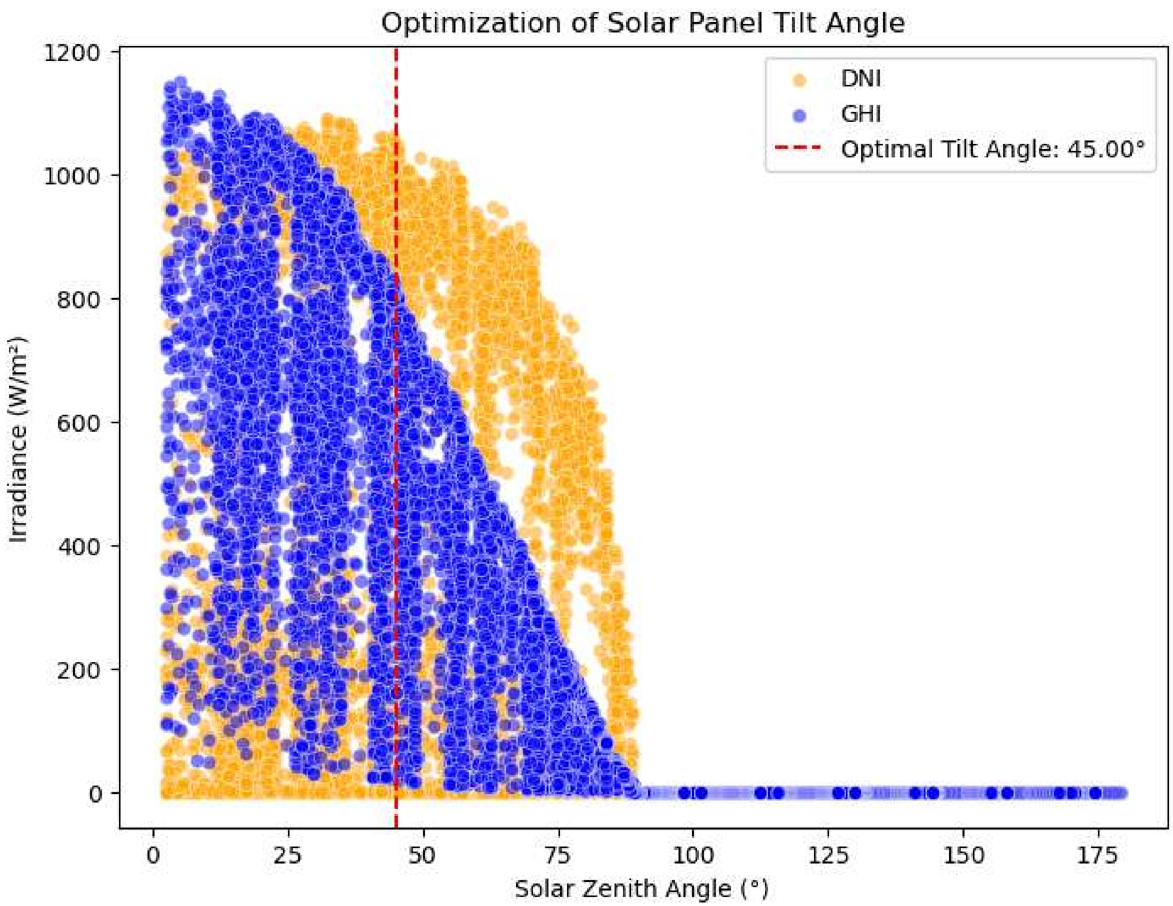
In [78]:
```
# Feature selection
solar_zenith = df["Solar Zenith Angle"]
dni = df["DNI"]
ghi = df["GHI"]
```

In [81]:
```
# Drop zero values to avoid errors in optimization
df_filtered = df[(df["DNI"] > 0) & (df["GHI"] > 0)]
solar_zenith_filtered = df_filtered["Solar Zenith Angle"].values
dni_filtered = df_filtered["DNI"].values
ghi_filtered = df_filtered["GHI"].values
```

In [82]:
```
# Define the function to optimize
# Assume the tilt angle should be close to (90 - Zenith Angle) for max energy

def efficiency_function(tilt_angle):
    optimal_dni = np.interp(tilt_angle, solar_zenith_filtered, dni_filtered)
    optimal_ghi = np.interp(tilt_angle, solar_zenith_filtered, ghi_filtered)
    return -1 * (optimal_dni + optimal_ghi)  # Maximize output

# Perform optimization
result = minimize(efficiency_function, x0=45, bounds=[(0, 90)])  # Initial guess at 45 degrees
optimal_tilt = result.x[0]

print(f"Optimal Solar Panel Tilt Angle: {optimal_tilt:.2f} degrees")
```

```
Optimal Solar Panel Tilt Angle: 45.00 degrees
```

In [83]:
```python
# Visualization
plt.figure(figsize=(8, 6))
sns.scatterplot(x=solar_zenith, y=dni, label='DNI', color='orange', alpha=0.5)
sns.scatterplot(x=solar_zenith, y=ghi, label='GHI', color='blue', alpha=0.5)
plt.axvline(optimal_tilt, color='red', linestyle='--', label=f'Optimal Tilt Angle: {optimal_tilt:.2f}°')
plt.xlabel("Solar Zenith Angle (°)")
plt.ylabel("Irradiance (W/m²)")
plt.title("Optimization of Solar Panel Tilt Angle")
plt.legend()
plt.show()
```

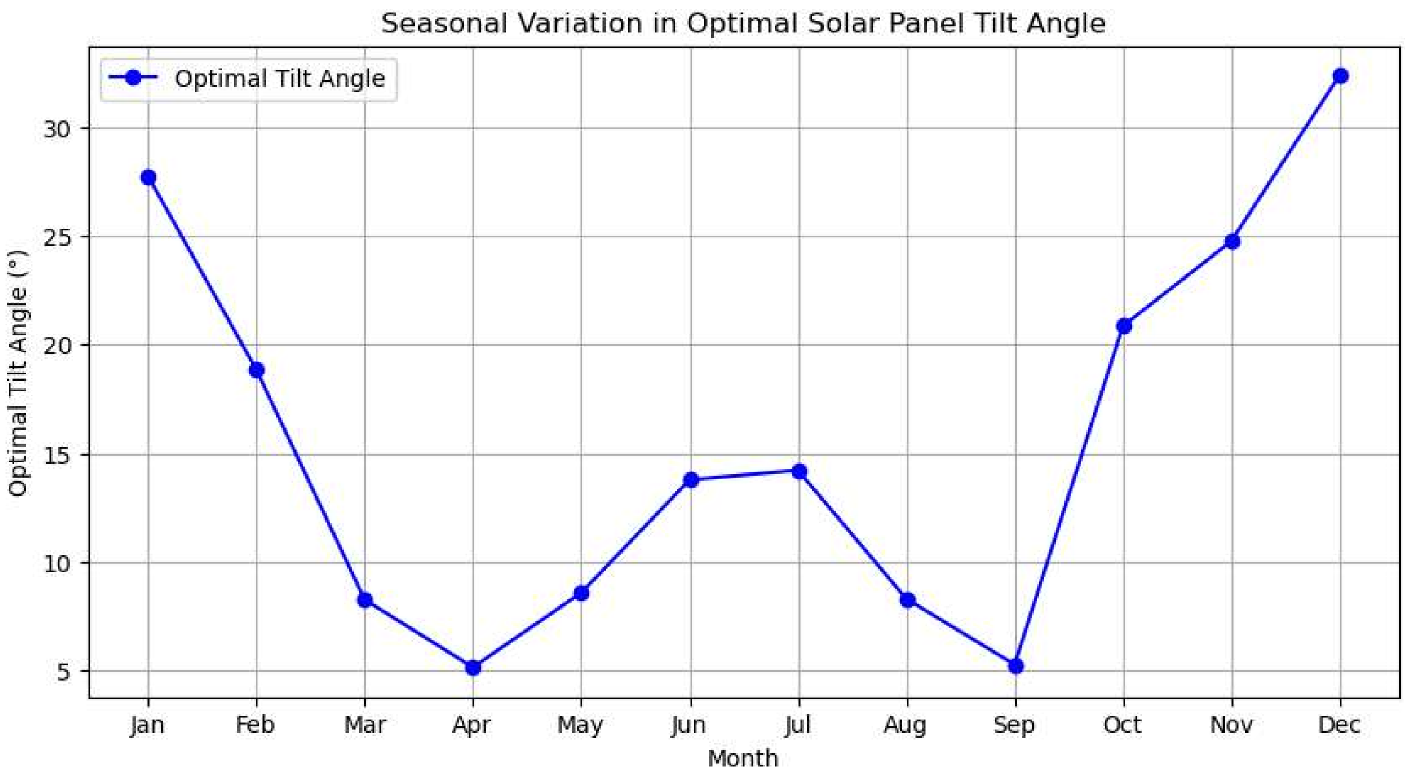


In [ ]:

Seasonal Tilt Optimization

In [84]:
```
# Define function to find optimal tilt angle per month
def find_optimal_tilt(df):
    optimal_angles = []

    for month in range(1, 13):  # Iterate over all months
        month_data = df[df['Month'] == month]

        # Find the solar zenith angle where DNI + GHI is maximized
        best_tilt = month_data.loc[(month_data['DNI'] + month_data['GHI']).idxmax(), 'Solar Zenith Angle']

        optimal_angles.append(best_tilt)

    return optimal_angles

# Compute optimal tilt angles for each month
monthly_optimal_tilt = find_optimal_tilt(df)
```

In [85]:
```python
# Plot the seasonal variation in optimal tilt angles
plt.figure(figsize=(10, 5))
plt.plot(range(1, 13), monthly_optimal_tilt, marker='o', linestyle='-', color='b', label='Optimal Tilt Angle')
plt.xticks(range(1, 13), ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
plt.xlabel("Month")
plt.ylabel("Optimal Tilt Angle (°)")
plt.title("Seasonal Variation in Optimal Solar Panel Tilt Angle")
plt.legend()
plt.grid(True)
plt.show()
```



Comparing Solar Energy Output with Tilt Angle

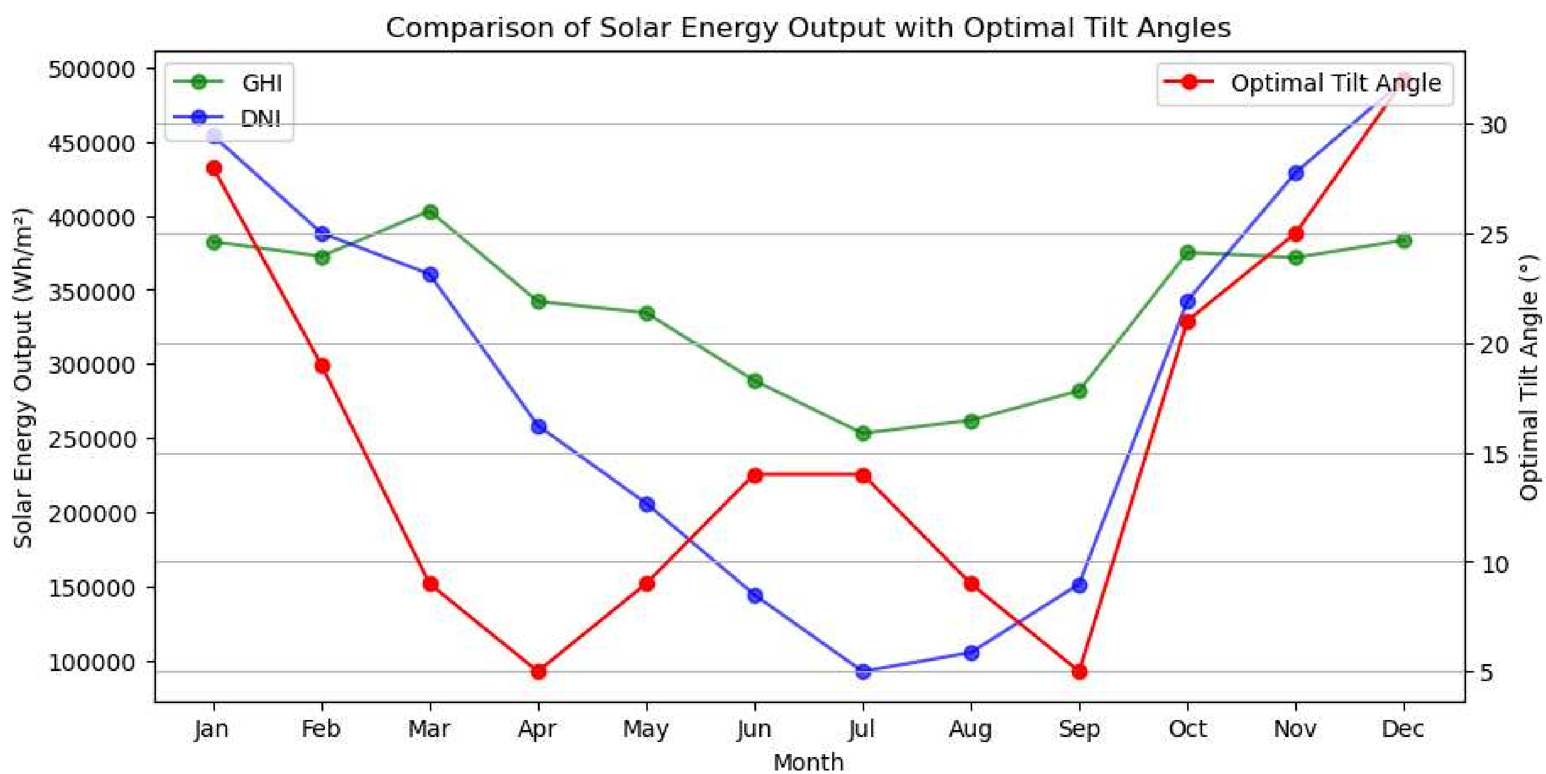
In [86]:
```
optimal_tilt = {
    1: 28, 2: 19, 3: 9, 4: 5, 5: 9, 6: 14,
    7: 14, 8: 9, 9: 5, 10: 21, 11: 25, 12: 32
}
```

In [87]:
```
# Convert to datetime format
df['Date'] = pd.to_datetime(df[['Year', 'Month', 'Day', 'Hour']])
df.set_index('Date', inplace=True)
```

In [88]:
```
# Compute total monthly solar energy output
monthly_energy = df.groupby(df.index.month)[['GHI', 'DNI']].sum()
```

In [89]:
```
# Use actual computed optimal tilt angles from analysis
optimal_tilt = {
    1: 28, 2: 19, 3: 9, 4: 5, 5: 9, 6: 14,
    7: 14, 8: 9, 9: 5, 10: 21, 11: 25, 12: 32
}

# Add optimal tilt angles to the dataframe
monthly_energy['Optimal Tilt'] = monthly_energy.index.map(optimal_tilt)
```

In [90]:
```
# Plotting the comparison
fig, ax1 = plt.subplots(figsize=(10, 5))

# Plot solar energy output
ax1.set_xlabel("Month")
ax1.set_ylabel("Solar Energy Output (Wh/m²)")
ax1.plot(monthly_energy.index, monthly_energy['GHI'], 'go-', label='GHI', alpha=0.7)
ax1.plot(monthly_energy.index, monthly_energy['DNI'], 'bo-', label='DNI', alpha=0.7)
ax1.legend(loc="upper left")

# Create second y-axis for optimal tilt
ax2 = ax1.twinx()
ax2.set_ylabel("Optimal Tilt Angle (°)")
ax2.plot(monthly_energy.index, monthly_energy['Optimal Tilt'], 'ro-', label="Optimal Tilt Angle")
ax2.legend(loc="upper right")

plt.title("Comparison of Solar Energy Output with Optimal Tilt Angles")
plt.xticks(range(1, 13), ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
plt.grid()
plt.show()
```



In [ ]:

In [ ]:

optimize the placement of solar panels

In [118]:
```
print(df.columns)
```

```
Index(['Year', 'Month', 'Day', 'Hour', 'Minute', 'Clearsky DHI',
       'Clearsky DNI', 'Temperature', 'Clearsky GHI', 'cloud fill flag',
       'Cloud Type', 'Dew Point', 'DHI', 'DNI', 'Fill Flag', 'GHI',
       'Relative Humidity', 'Solar Zenith Angle', 'Surface Albedo', 'Pressure',
       'Precipitable Water', 'Wind Direction', 'Wind Speed',
       'Solar_Efficiency_Score'],
      dtype='object')
```

In [113]:

```python
# Display the first few rows to understand the structure
print(df.head())
```

```
   Year  Month  Day  Hour  Minute  Clearsky DHI  Clearsky DNI  Temperature  \
0  2006      1    1     0      30             0             0          6.9   
1  2006      1    1     1      30             0             0          6.5   
2  2006      1    1     2      30             0             0          6.2   
3  2006      1    1     3      30             0             0          7.0   
4  2006      1    1     4      30            36           484         10.8   

   Clearsky GHI  cloud fill flag  ...  DNI  Fill Flag  GHI  Relative Humidity  \
0             0                0  ...    0          0    0              80.78   
1             0                0  ...    0          0    0              85.35   
2             0                0  ...    0          0    0              89.38   
3             0                0  ...    0          0    0              86.64   
4            96                1  ...    0          1   39              68.50   

   Solar Zenith Angle  Surface Albedo  Pressure  Precipitable Water  \
0              137.73            0.15       778                 0.9   
1              124.08            0.15       779                 1.0   
2              110.30            0.15       779                 1.0   
3               96.54            0.15       780                 1.0   
4               82.85            0.15       780                 1.0   

   Wind Direction  Wind Speed  
0              85         1.7  
1              89         1.7  
2              92         1.7  
3              94         2.1  
4              97         3.5  

[5 rows x 23 columns]
```

In [116]:

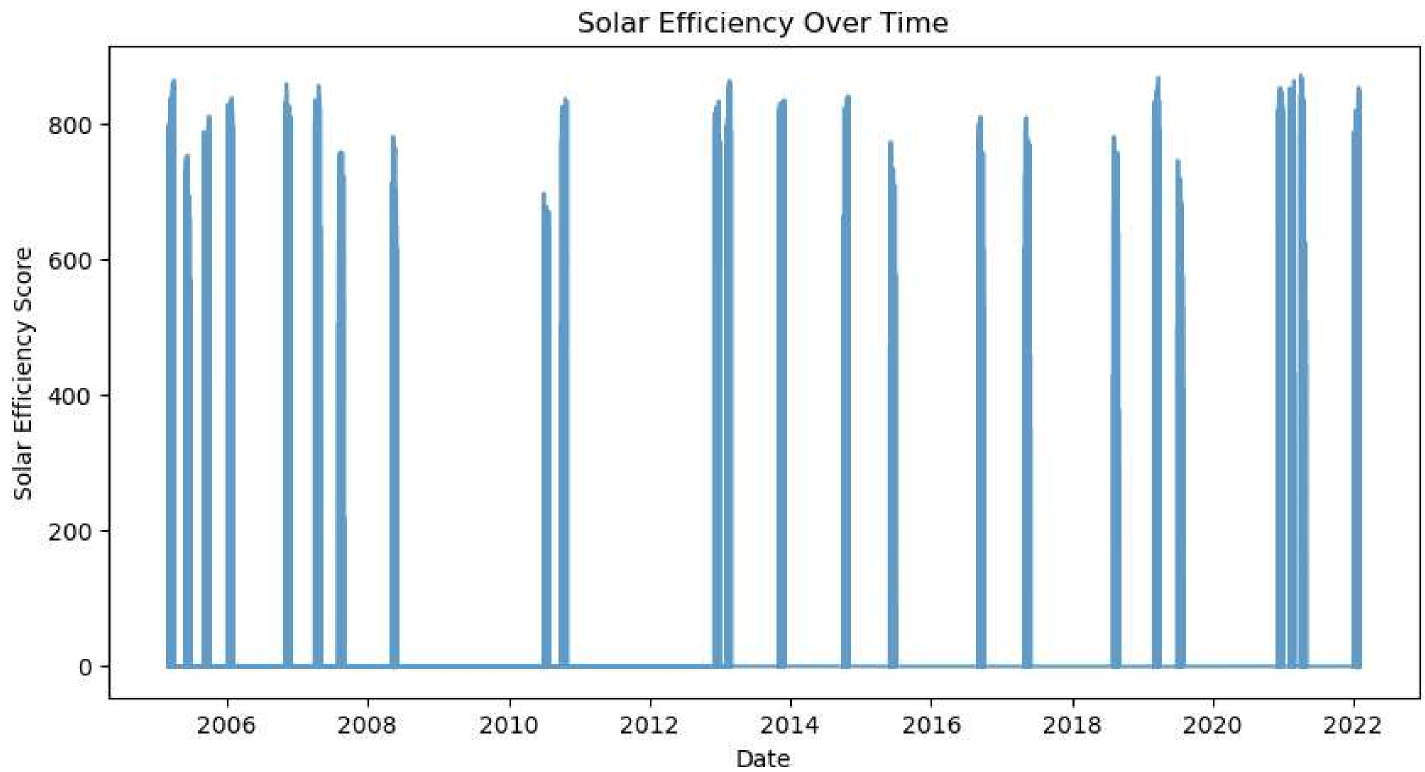
```python
# Compute Solar Efficiency Score
df['Solar_Efficiency_Score'] = df['DNI'] * 0.5 + df['GHI'] * 0.3 + df['DHI'] * 0.1 - df['cloud fill flag'] * 0.2 - df['Temperature'] *
```

Finding Best Time for Solar Efficiency

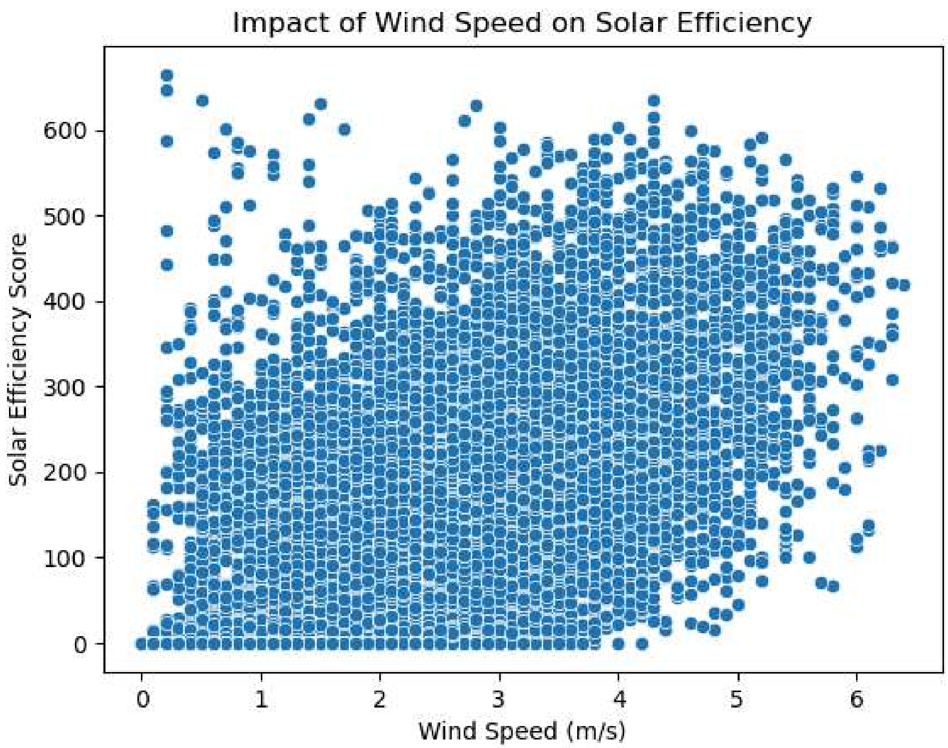
In [119]:
```
# Load your dataset (assuming it's already in df)
df['Datetime'] = pd.to_datetime(df[['Year', 'Month', 'Day', 'Hour', 'Minute']])

# Find the best hour for solar generation
best_hour = df.groupby('Hour')['Solar_Efficiency_Score'].mean().idxmax()
print(f"Best Hour for Solar Energy Generation: {best_hour}h")

# Find the best month for solar generation
best_month = df.groupby('Month')['Solar_Efficiency_Score'].mean().idxmax()
print(f"Best Month for Solar Energy Generation: {best_month}")
```

```
Best Hour for Solar Energy Generation: 8h
Best Month for Solar Energy Generation: 12
```

In [120]:
```
# Plot Solar Efficiency over Time
plt.figure(figsize=(10,5))
plt.plot(df['Datetime'], df['Solar_Efficiency_Score'], linestyle='-', marker='', alpha=0.7)
plt.xlabel('Date')
plt.ylabel('Solar Efficiency Score')
plt.title('Solar Efficiency Over Time')
plt.show()
```



In [109]:
```
# Analyzing impact of wind speed on solar panel efficiency
sns.scatterplot(data=df, x='Wind Speed', y='Solar Efficiency Score')
plt.xlabel("Wind Speed (m/s)")
plt.ylabel("Solar Efficiency Score")
plt.title("Impact of Wind Speed on Solar Efficiency")
plt.show()
```



In [110]:
```
# Identify the best hour of the day for optimal energy production
hourly_avg = df.groupby('Hour')['Solar Efficiency Score'].mean()
best_hour = hourly_avg.idxmax()
print(f"Best Hour for Solar Energy Generation: {best_hour}h")
```

```
Best Hour for Solar Energy Generation: 10h
```

In [ ]:

In [ ]:

optimize energy storage and grid distribution

In [121]:
```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
```

In [122]:
```
# Load the dataset
df = pd.read_excel("D:/Capstone_2025/data/solardata_addis.xlsx")
```

In [123]:
```
# Selecting relevant features for energy prediction
features = ['Clearsky GHI', 'Temperature', 'DNI', 'DHI', 'Relative Humidity', 'Wind Speed']

data = df[features]
scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(data)
```

In [124]:
```
# Prepare data for LSTM
sequence_length = 24  # Using past 24 hours to predict the next hour
def create_sequences(data, sequence_length):
    X, y = [], []
    for i in range(len(data) - sequence_length):
        X.append(data[i:i + sequence_length])
        y.append(data[i + sequence_length, 0])  # Predicting Clearsky GHI
    return np.array(X), np.array(y)

X, y = create_sequences(data_scaled, sequence_length)
X_train, X_test = X[:int(0.8*len(X))], X[int(0.8*len(X)):]
y_train, y_test = y[:int(0.8*len(y))], y[int(0.8*len(y)):]

# Building LSTM model
model = Sequential([
    LSTM(100, return_sequences=True, input_shape=(sequence_length, X.shape[2])),
    Dropout(0.2),
    LSTM(50, return_sequences=False),
    Dropout(0.2),
    Dense(25, activation='relu'),
    Dense(1)
])

model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test))
```

```
C:\Users\reshm\anaconda3\Lib\site-packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not pass an `input_shape`/`input_dim` argu
ment to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

```
Epoch 1/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 19s 30ms/step - loss: 0.0314 - val_loss: 0.0013
Epoch 2/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 12s 28ms/step - loss: 0.0023 - val_loss: 0.0012
Epoch 3/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 12s 27ms/step - loss: 0.0014 - val_loss: 0.0017
Epoch 4/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - loss: 0.0013 - val_loss: 9.7969e-04
Epoch 5/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 13s 30ms/step - loss: 0.0010 - val_loss: 0.0013
Epoch 6/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 13s 30ms/step - loss: 7.4592e-04 - val_loss: 0.0016
Epoch 7/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 13s 30ms/step - loss: 7.2723e-04 - val_loss: 0.0024
Epoch 8/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 14s 31ms/step - loss: 6.5488e-04 - val_loss: 8.0657e-04
Epoch 9/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 14s 32ms/step - loss: 5.9192e-04 - val_loss: 0.0022
Epoch 10/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 14s 31ms/step - loss: 5.3237e-04 - val_loss: 0.0022
Epoch 11/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 18s 41ms/step - loss: 5.4042e-04 - val_loss: 0.0025
Epoch 12/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 36s 77ms/step - loss: 4.1149e-04 - val_loss: 0.0023
Epoch 13/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 24s 36ms/step - loss: 4.9912e-04 - val_loss: 0.0024
Epoch 14/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 16s 36ms/step - loss: 4.3289e-04 - val_loss: 0.0018
Epoch 15/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 22s 39ms/step - loss: 3.8709e-04 - val_loss: 0.0020
Epoch 16/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 16s 37ms/step - loss: 3.5588e-04 - val_loss: 0.0024
Epoch 17/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 16s 36ms/step - loss: 4.2132e-04 - val_loss: 0.0019
Epoch 18/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 16s 35ms/step - loss: 3.4934e-04 - val_loss: 0.0011
Epoch 19/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 18s 41ms/step - loss: 3.2265e-04 - val_loss: 0.0025
Epoch 20/20
438/438 ━━━━━━━━━━━━━━━━━━━━ 19s 38ms/step - loss: 3.1641e-04 - val_loss: 0.0027
```

Out[124]:
```
<keras.src.callbacks.history.History at 0x22e68d861d0>
```

In [125]:
```python
# Predicting solar energy generation
predictions = model.predict(X_test)
predictions = scaler.inverse_transform(np.concatenate((predictions, np.zeros((predictions.shape[0], data.shape[1] - 1))), axis=1))[:, 0
```

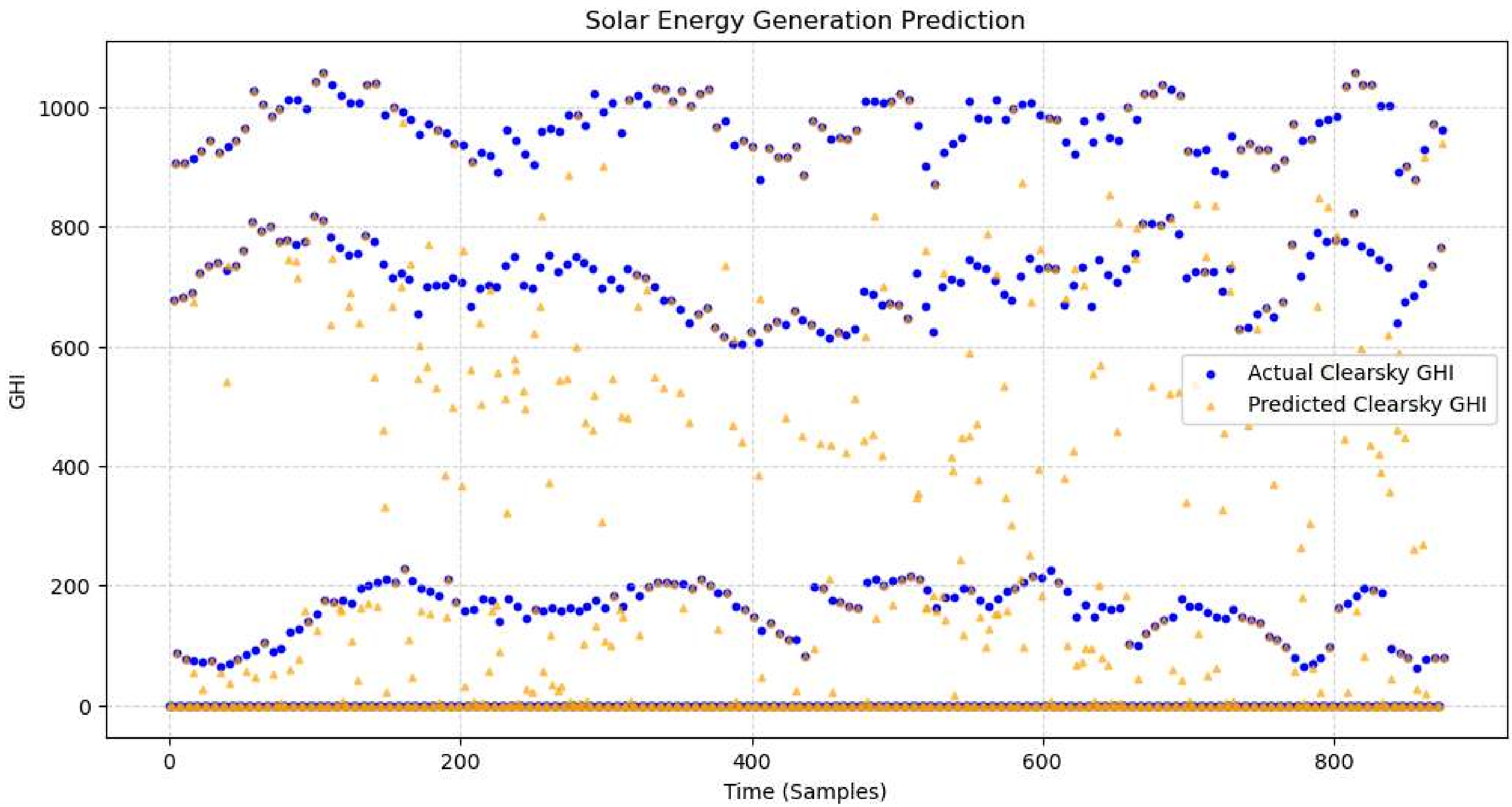
```
110/110 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step
```

In [ ]:

```

```

In [131]:

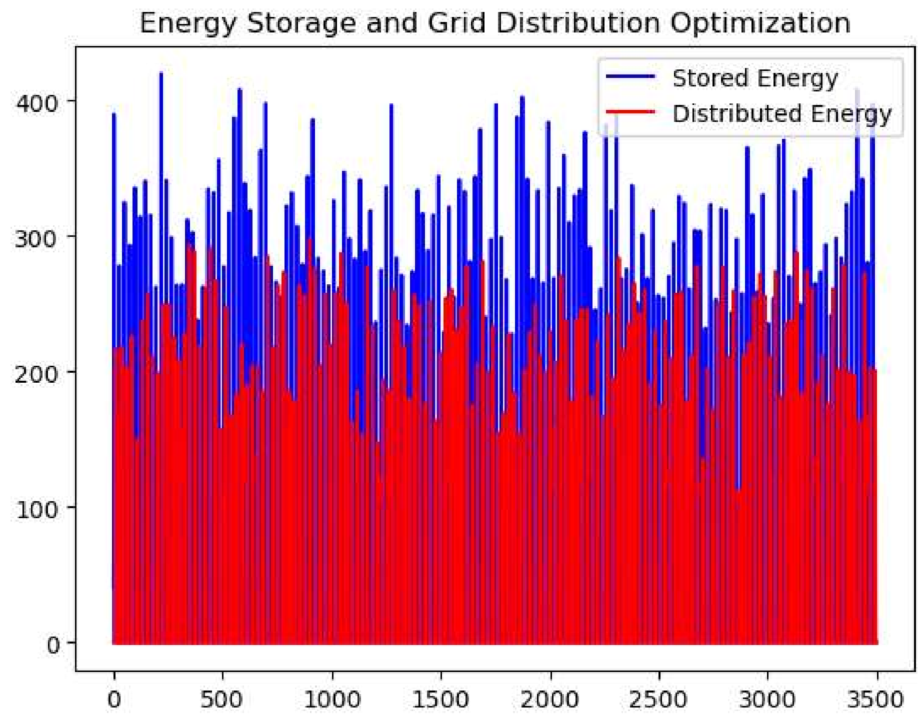
```
# Define Actual and Predicted GHI
actual_ghi = df['Clearsky GHI']
predicted_ghi = df['GHI']

# Sample every 20th point to reduce clutter
sample_rate = 20
actual_ghi_sampled = actual_ghi[::sample_rate]
predicted_ghi_sampled = predicted_ghi[::sample_rate]
x_sampled = np.arange(len(actual_ghi_sampled))

# Plot
plt.figure(figsize=(12, 6))
plt.scatter(x_sampled, actual_ghi_sampled, label="Actual Clearsky GHI", color='blue', marker='o', s=10)
plt.scatter(x_sampled, predicted_ghi_sampled, label="Predicted Clearsky GHI", color='orange', marker='^', s=10, alpha=0.6)

plt.xlabel("Time (Samples)")
plt.ylabel("GHI")
plt.title("Solar Energy Generation Prediction")
plt.legend()
plt.grid(True, linestyle="--", alpha=0.5)

plt.show()
```



Solar Energy Generation Prediction
GHI
Time (Samples)
0
200
400
600
800
1000
Actual Clearsky GHI
Predicted Clearsky GHI

In [132]:
```python
# Optimization logic for storage and grid distribution
energy_generated = predictions  # Predicted solar energy generation
storage_capacity = 500  # Arbitrary battery storage capacity (kWh)
grid_demand = np.random.uniform(100, 300, len(energy_generated))  # Simulated grid demand

stored_energy = np.zeros_like(energy_generated)
distributed_energy = np.zeros_like(energy_generated)
storage_level = 0

for i in range(len(energy_generated)):
    surplus = energy_generated[i] - grid_demand[i]
    if surplus > 0:  # Store excess energy
        if storage_level + surplus <= storage_capacity:
            storage_level += surplus
            stored_energy[i] = surplus
        else:
            stored_energy[i] = storage_capacity - storage_level
            storage_level = storage_capacity
    else:  # Distribute from storage
        if storage_level > abs(surplus):
            storage_level += surplus  # Withdraw from storage
            distributed_energy[i] = abs(surplus)
        else:
            distributed_energy[i] = storage_level
            storage_level = 0
```

In [133]:
```
# Plotting storage vs distribution
plt.plot(stored_energy, label='Stored Energy', color='blue')
plt.plot(distributed_energy, label='Distributed Energy', color='red')
plt.legend()
plt.title('Energy Storage and Grid Distribution Optimization')
plt.show()
```



In [ ]:

In [ ]:

In [ ]: