Importing and Reading Data

In [42]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import datetime
```

In [43]:
```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import classification_report
from sklearn.model_selection import cross_val_score
```

In [44]:
```
df=pd.read_csv("Occupancy_Estimation.csv")
df
```

Out[44]:

	Date	Time	S1_Temp	S2_Temp	S3_Temp	S4_Temp	S1_Light	S2_Lig
0	2017/12/22	10:49:41	24.94	24.75	24.56	25.38	121	
1	2017/12/22	10:50:12	24.94	24.75	24.56	25.44	121	
2	2017/12/22	10:50:42	25.00	24.75	24.50	25.44	121	
3	2017/12/22	10:51:13	25.00	24.75	24.56	25.44	121	
4	2017/12/22	10:51:44	25.00	24.75	24.56	25.44	121	
...	...	...	...	...	...	...	...	
10124	2018/01/11	08:58:07	25.06	25.13	24.69	25.31	6	
10125	2018/01/11	08:58:37	25.06	25.06	24.69	25.25	6	
10126	2018/01/11	08:59:08	25.13	25.06	24.69	25.25	6	
10127	2018/01/11	08:59:39	25.13	25.06	24.69	25.25	6	
10128	2018/01/11	09:00:09	25.13	25.06	24.69	25.25	6	

10129 rows × 19 columns

◇ Occupancy Estimation Dataset

- This dataset contains time-series data collected from a room using multiple non-intrusive environmental sensors—temperature, light, sound, CO_2, and PIR (motion).
- It was gathered over 4 days in a controlled setting, with occupancy manually recorded between 0 and 3 people.
- The goal is to estimate room occupancy based on sensor readings.

◇ Dataset Summary

- **Task**: Classification (Occupancy: 0–3 people)
- **Type**: Multivariate, Time-Series

- **Instances**: 10,129
- **Features**: 18
- **Missing Values**: None
- **Collection Period**: 4 days
- **Room Size**: 6m x 4.6m
- **Ground Truth**: Manually recorded occupancy count

⍰ Sensor Configuration

- The experimental setup consisted of 7 sensor nodes and 1 edge node in a star topology.
- Data was transmitted every 30 seconds using wireless transceivers.
- No HVAC systems were active during data collection.

Sensor Types Used:

- **Temperature Sensors** (S1–S4)
- **Light Sensors** (S1–S4)
- **Sound Sensors** (S1–S4)
- **CO_2 Sensor** (S5)
- **Passive Infrared (PIR) Sensors** (S6, S7)

All sensors were manually calibrated for accuracy.

⍰ Feature Overview

Variable Name	Type	Description	Units	Missing Values
`Date`	Date	Date of recording	YYYY/MM/DD	No
`Time`	Time	Time of recording	HH:MM:SS	No
`S1_Temp`	Continuous	Temperature from Sensor 1	°C	No
`S2_Temp`	Continuous	Temperature from Sensor 2	°C	No
`S3_Temp`	Continuous	Temperature from Sensor 3	°C	No
`S4_Temp`	Continuous	Temperature from Sensor 4	°C	No

Variable Name	Type	Description	Units	Missing Values
`S1_Light`	Integer	Light intensity from Sensor 1	Lux	No
`S2_Light`	Integer	Light intensity from Sensor 2	Lux	No
`S3_Light`	Integer	Light intensity from Sensor 3	Lux	No
`S4_Light`	Integer	Light intensity from Sensor 4	Lux	No
`S1_Sound`	Continuous	Sound level from Sensor 1	Volts	No
`S2_Sound`	Continuous	Sound level from Sensor 2	Volts	No
`S3_Sound`	Continuous	Sound level from Sensor 3	Volts	No
`S4_Sound`	Continuous	Sound level from Sensor 4	Volts	No
`S5_CO2`	Continuous	CO_2 concentration from Sensor 5	ppm	No
`S6_PIR`	Binary	Motion detection from Sensor 6	0/1	No
`S7_PIR`	Binary	Motion detection from Sensor 7	0/1	No
`Occupancy`	Integer	Number of people in the room	0–3	No

Use Cases

- Smart building automation
- Energy-efficient HVAC control
- Real-time occupancy monitoring
- Sensor fusion and anomaly detection
- Time-series forecasting and classification

Column Categorization

In [45]:
```
num_cols = ['S1_Temp','S2_Temp','S3_Temp','S4_Temp','S1_Light','S2_Light','S3_
cat_cols = ['S6_PIR','S7_PIR','Room_Occupancy_Count']
```

Basic Checks

Exploratory Data Analysis

In [46]:
```
df.shape
```

Out[46]: (10129, 19)

Basic Structure Info

In [47]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10129 entries, 0 to 10128
Data columns (total 19 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Date                  10129 non-null  object 
 1   Time                  10129 non-null  object 
 2   S1_Temp               10129 non-null  float64
 3   S2_Temp               10129 non-null  float64
 4   S3_Temp               10129 non-null  float64
 5   S4_Temp               10129 non-null  float64
 6   S1_Light              10129 non-null  int64  
 7   S2_Light              10129 non-null  int64  
 8   S3_Light              10129 non-null  int64  
 9   S4_Light              10129 non-null  int64  
 10  S1_Sound              10129 non-null  float64
 11  S2_Sound              10129 non-null  float64
 12  S3_Sound              10129 non-null  float64
 13  S4_Sound              10129 non-null  float64
 14  S5_CO2                10129 non-null  int64  
 15  S5_CO2_Slope          10129 non-null  float64
 16  S6_PIR                10129 non-null  int64  
 17  S7_PIR                10129 non-null  int64  
 18  Room_Occupancy_Count  10129 non-null  int64  
dtypes: float64(9), int64(8), object(2)
memory usage: 1.5+ MB
```

Data Types of Columns

In [48]:
```
df.dtypes
```

Out[48]:
```
Date                      object
Time                      object
S1_Temp                  float64
S2_Temp                  float64
S3_Temp                  float64
S4_Temp                  float64
S1_Light                   int64
S2_Light                   int64
S3_Light                   int64
S4_Light                   int64
S1_Sound                 float64
S2_Sound                 float64
S3_Sound                 float64
S4_Sound                 float64
S5_CO2                     int64
S5_CO2_Slope             float64
S6_PIR                     int64
S7_PIR                     int64
Room_Occupancy_Count       int64
dtype: object
```

Handling missing values

In [49]:
```
# Check missing values
missing = df.isnull().sum()
print("Missing values:\n", missing)

# Option: fill numeric NaNs with column median
df.fillna(df.median(numeric_only=True), inplace=True)
```

```
Missing values:
 Date                    0
Time                    0
S1_Temp                 0
S2_Temp                 0
S3_Temp                 0
S4_Temp                 0
S1_Light                0
S2_Light                0
S3_Light                0
S4_Light                0
S1_Sound                0
S2_Sound                0
S3_Sound                0
S4_Sound                0
S5_CO2                  0
S5_CO2_Slope            0
S6_PIR                  0
S7_PIR                  0
Room_Occupancy_Count    0
dtype: int64
```

In [50]:
```
df.duplicated().sum()
```

Out[50]: 0

⯑ Duplicate Check

A check for duplicate rows was performed using `df.duplicated().sum()`, and the result was:

0 duplicate rows

This confirms that each row in the dataset is unique, meaning:

- No repeated sensor readings with identical timestamps and values.
- The data collection process was consistent and clean.
- We can proceed with analysis and modeling without needing to remove any redundant entries.

This is especially important for time-series data, where duplicate entries could distort trends or bias model training.

In [51]:
```
df.head()
```

Out[51]:

	Date	Time	S1_Temp	S2_Temp	S3_Temp	S4_Temp	S1_Light	S2_Light	S
0	2017/12/22	10:49:41	24.94	24.75	24.56	25.38	121	34	
1	2017/12/22	10:50:12	24.94	24.75	24.56	25.44	121	33	
2	2017/12/22	10:50:42	25.00	24.75	24.50	25.44	121	34	
3	2017/12/22	10:51:13	25.00	24.75	24.56	25.44	121	34	
4	2017/12/22	10:51:44	25.00	24.75	24.56	25.44	121	34	

Statistical Summary

In [52]:
```
df.describe()
```

Out[52]:

	S1_Temp	S2_Temp	S3_Temp	S4_Temp	S1_Light	
count	10129.000000	10129.000000	10129.000000	10129.000000	10129.000000	1(
mean	25.454012	25.546059	25.056621	25.754125	25.445059	
std	0.351351	0.586325	0.427283	0.356434	51.011264	
min	24.940000	24.750000	24.440000	24.940000	0.000000	
25%	25.190000	25.190000	24.690000	25.440000	0.000000	
50%	25.380000	25.380000	24.940000	25.750000	0.000000	
75%	25.630000	25.630000	25.380000	26.000000	12.000000	
max	26.380000	29.000000	26.190000	26.560000	165.000000	

Unique Values in Each Column

In [53]:
```
df.nunique()
```

Out[53]:
```
Date                        7
Time                    10129
S1_Temp                    24
S2_Temp                    69
S3_Temp                    29
S4_Temp                    27
S1_Light                   68
S2_Light                   82
S3_Light                  177
S4_Light                   75
S1_Sound                  231
S2_Sound                  185
S3_Sound                  258
S4_Sound                  106
S5_CO2                    186
S5_CO2_Slope             1579
S6_PIR                      2
S7_PIR                      2
Room_Occupancy_Count        4
dtype: int64
```

Seasonality Check

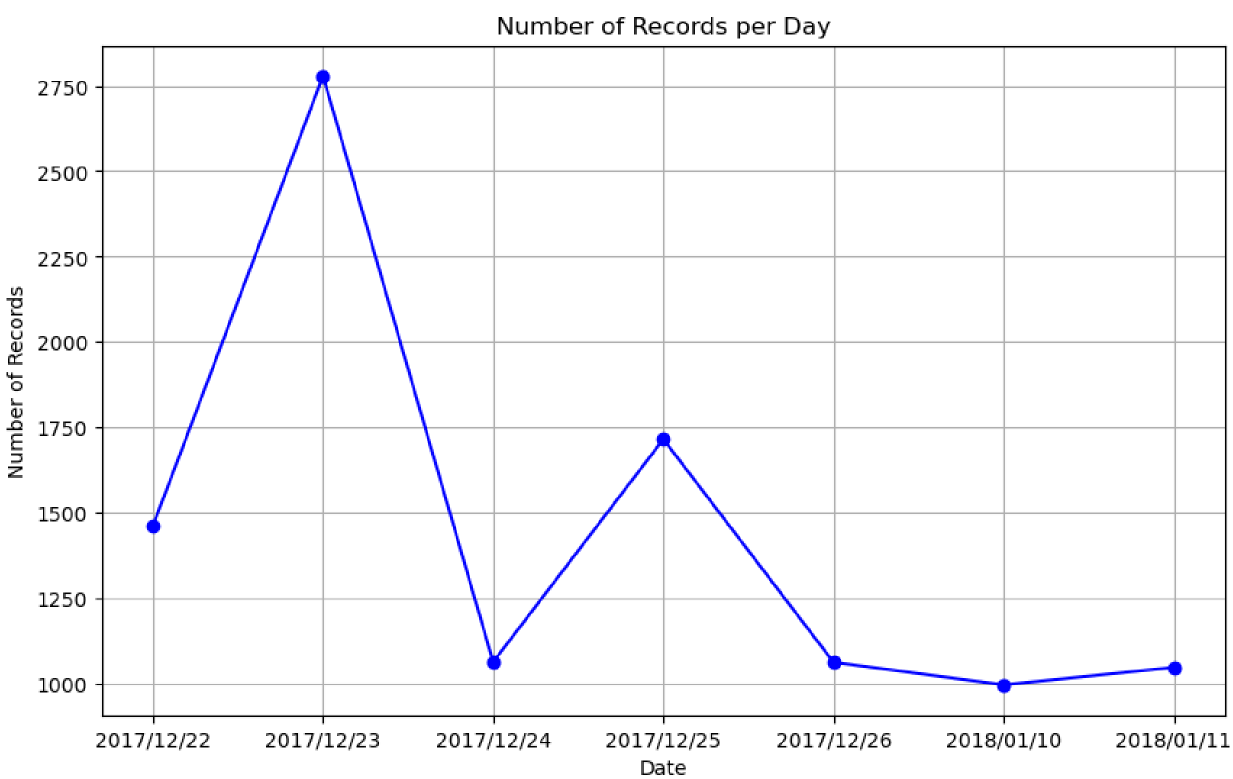
In [54]:
```
df['Date'].unique()
```

Out[54]:
```
array(['2017/12/22', '2017/12/23', '2017/12/24', '2017/12/25',
       '2017/12/26', '2018/01/10', '2018/01/11'], dtype=object)
```

Data Aggregation and Trend Plot

- Groups the DataFrame df by the Date column and counts how many records exist for each day.
- Creates a line plot with:
- X-axis: Dates
- Y-axis: Number of records per day
- Blue line with circular markers to show daily counts
- Adds labels, a title, and a grid for better readability.

In [55]:
```
# Group by date and count the number of records for each day
daily_counts = df.groupby('Date').size().reset_index(name='count')
# Plotting
plt.figure(figsize=(10, 6))
plt.plot(daily_counts['Date'], daily_counts['count'], marker='o', linestyle='-
plt.title('Number of Records per Day')
plt.xlabel('Date')
plt.ylabel('Number of Records')
plt.grid(True)
plt.show()
```



The Plot Tells:

- Peaks indicate days with high activity or data volume.
- Troughs show days with fewer records.
- If the line is relatively flat, your data is evenly distributed across dates.
- If there are sudden spikes or drops, those might be worth investigating (e.g., system outages, special events, or data collection issues).

DateTime Conversion + Time-of-Day Binning

Purpose:

- Enables time-of-day analysis (e.g., when events are most frequent).

In [56]:
```
df['Date_time'] = pd.to_datetime(df['Date'] +' '+ df['Time'])
dt_time = df.pop('Date_time')
df.insert(2, 'Date_time', dt_time)

df['Hours'] = df['Date_time'].dt.hour
df['Time_of_Day'] = pd.cut(df['Hours'], bins = [0,6,12,17,22,24], labels = ['N

df.pop('Hours')
day_time = df.pop('Time_of_Day')
df.insert(2, 'Time_of_Day', day_time)
df.head()
```

Out[56]:

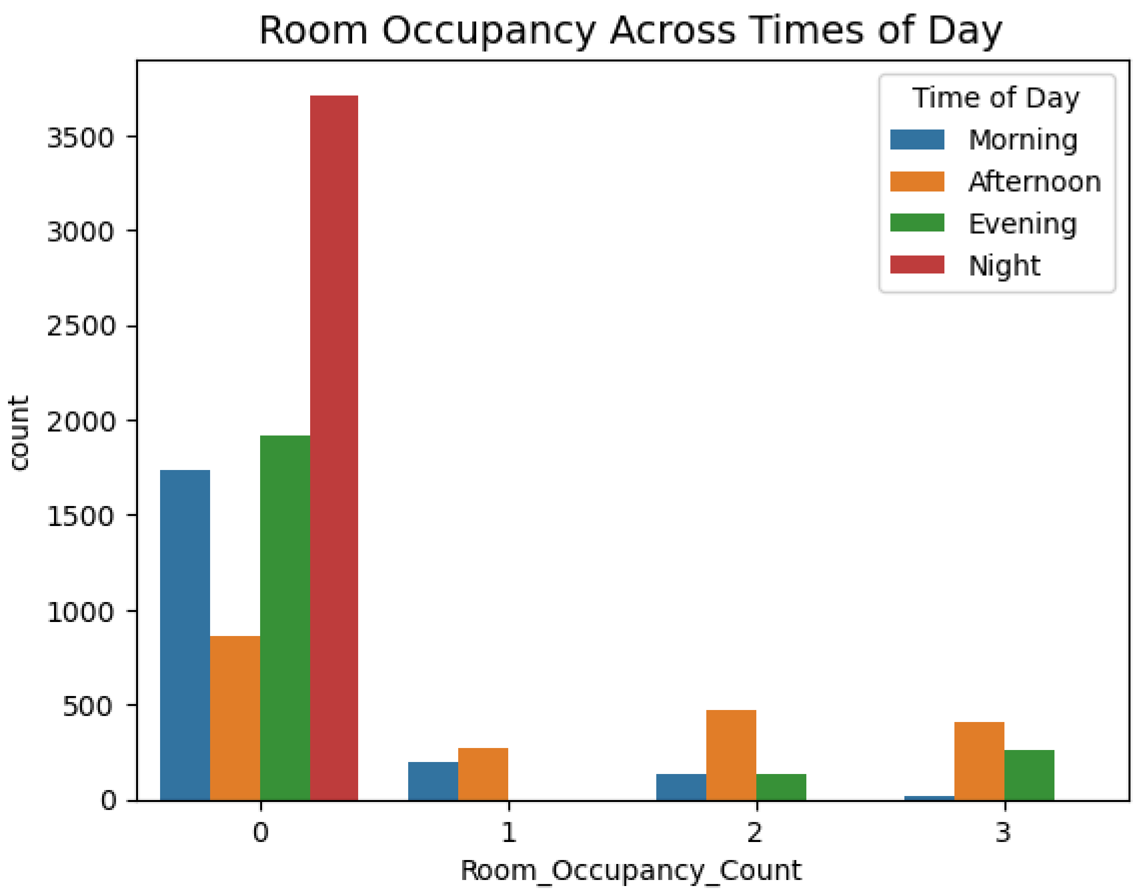
	Date	Time	Time_of_Day	Date_time	S1_Temp	S2_Temp	S3_Temp	S4_Te
0	2017/12/22	10:49:41	Morning	2017-12-22 10:49:41	24.94	24.75	24.56	25
1	2017/12/22	10:50:12	Morning	2017-12-22 10:50:12	24.94	24.75	24.56	25
2	2017/12/22	10:50:42	Morning	2017-12-22 10:50:42	25.00	24.75	24.50	25
3	2017/12/22	10:51:13	Morning	2017-12-22 10:51:13	25.00	24.75	24.56	25
4	2017/12/22	10:51:44	Morning	2017-12-22 10:51:44	25.00	24.75	24.56	25

5 rows × 21 columns

Time-of-Day Occupancy Count Plot

Visualizes how occupancy varies across daily segments.

In [57]:
```
sns.countplot(data=df,
              x='Room_Occupancy_Count',
              hue='Time_of_Day',
              hue_order=['Morning', 'Afternoon', 'Evening', 'Night'])
plt.title('Room Occupancy Across Times of Day', fontsize=14)
plt.legend(title='Time of Day')
plt.show()
```



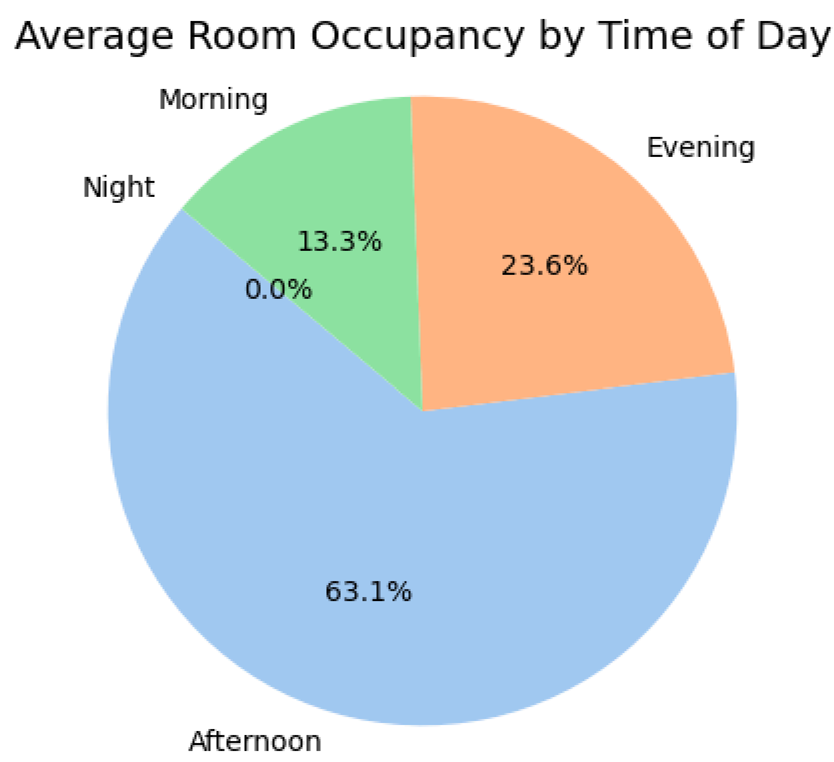
Insights:

- Which time of day has the highest occupancy.
- Are certain times of day more likely to have low or high room usage?
- Any unusual patterns — like rooms being busiest late at night or underused during the afternoon.

Pie Chart of Average Occupancy by Time-of-Day

To analyze and visualize how room occupancy varies across different times of day using a pie chart.

In [58]:
```
# Calculate average occupancy per time of day with observed=True
avg_by_time = df.groupby('Time_of_Day', observed=True)['Room_Occupancy_Count']

# Plot pie chart
plt.figure(figsize=(6, 4))
plt.pie(
    avg_by_time,
    labels=avg_by_time.index,
    autopct='%.1f%%',
    startangle=140,
    colors=sns.color_palette('pastel')[0:len(avg_by_time)]
)

plt.title('Average Room Occupancy by Time of Day', fontsize=14)
plt.axis('equal')
plt.tight_layout()
plt.show()
```



The Chart says:

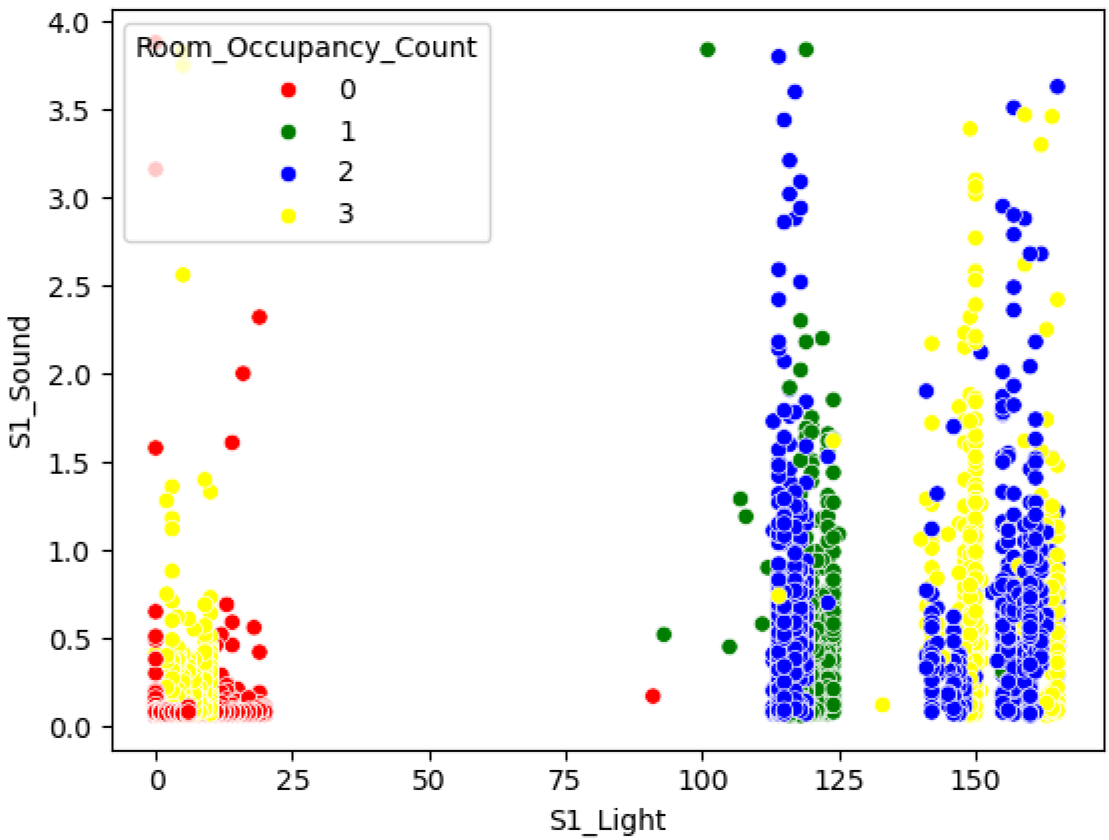
- The largest slice indicates the time of day with the highest average occupancy — essentially, the busiest time.

- Smaller slices reveal quieter periods or lower occupancy averages.

Scatterplot (Light vs Sound) with Occupancy Hue

Checks correlation and clustering between sensory inputs and occupancy levels.

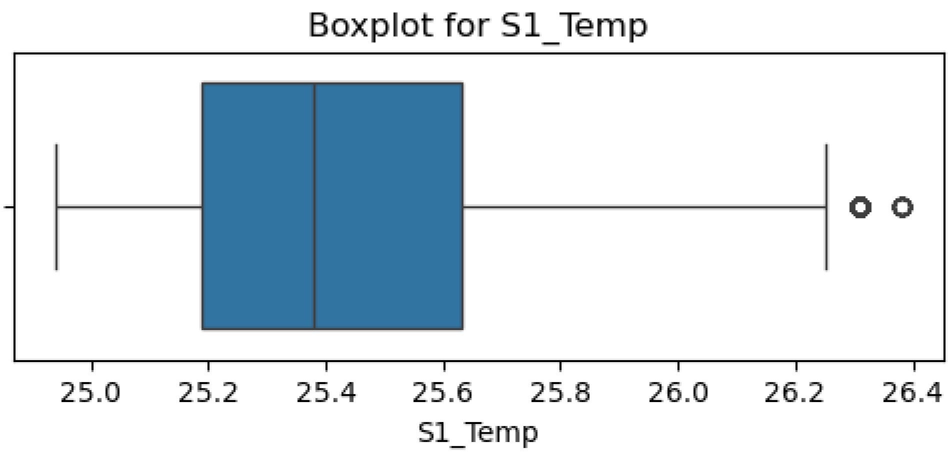
In [59]:
```
sns.scatterplot(data = df, x = 'S1_Light', y = 'S1_Sound', hue = 'Room_Occupar
plt.show()
```

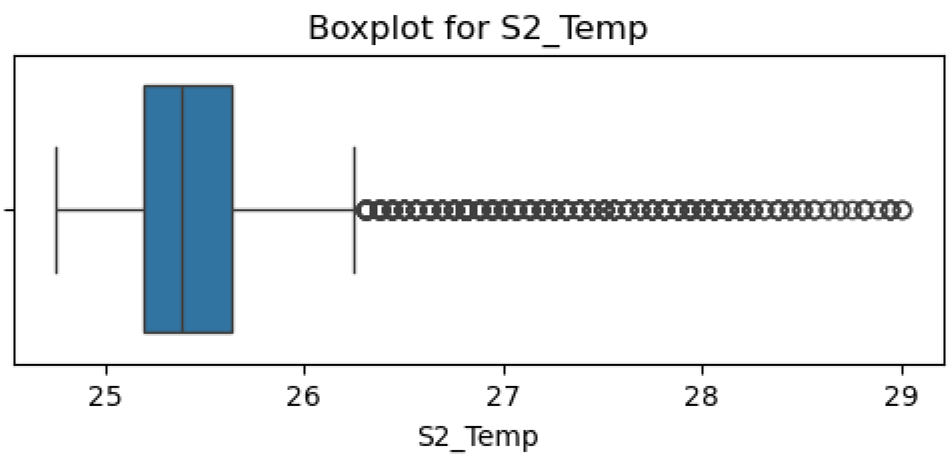


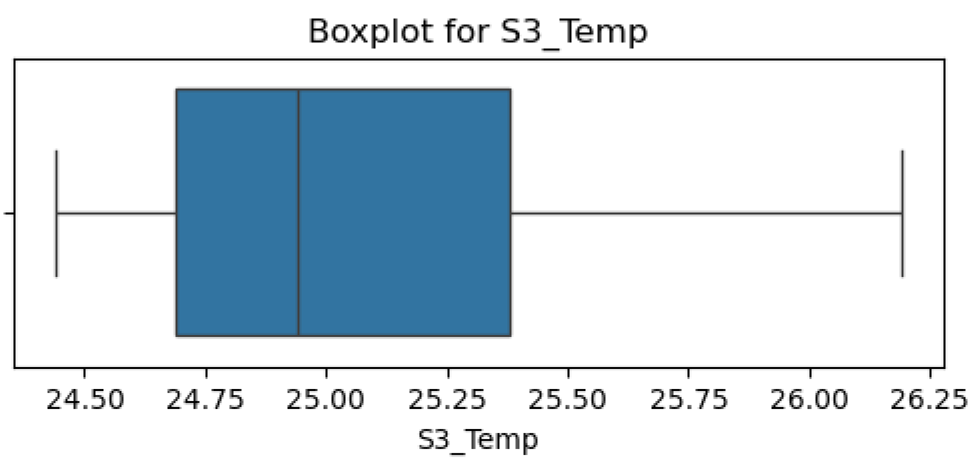
Boxplot showing range and outliers for each numerical column

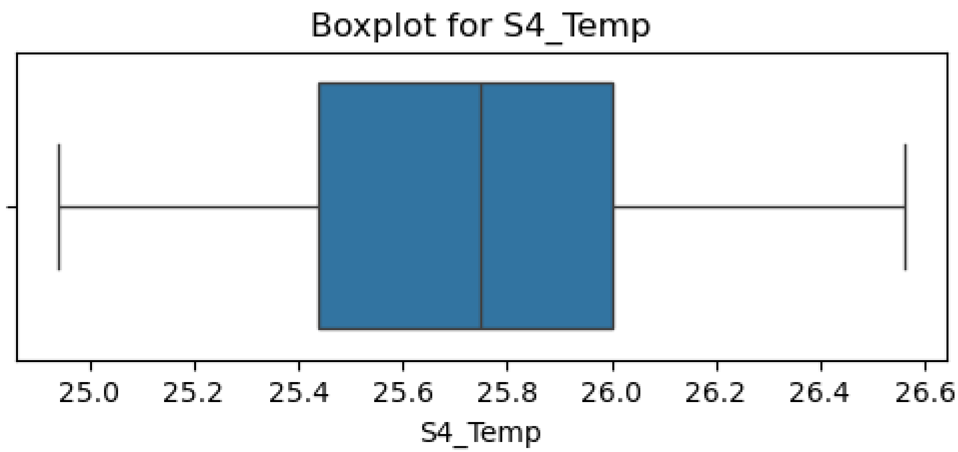
Highlights variance and potential sensor outliers.

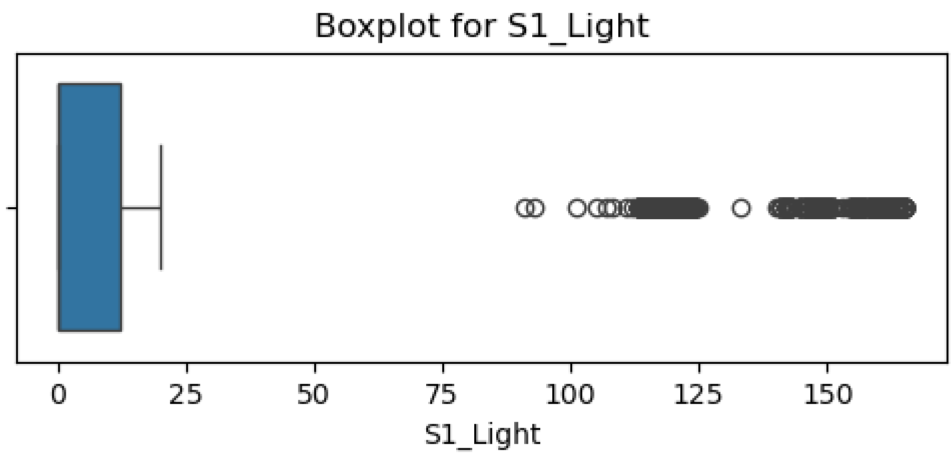
In [60]:
```
for i in num_cols:
  plt.figure(figsize=(6,2))
  sns.boxplot(data = df, x = i)
  plt.title('Boxplot for {}'.format(i))
```

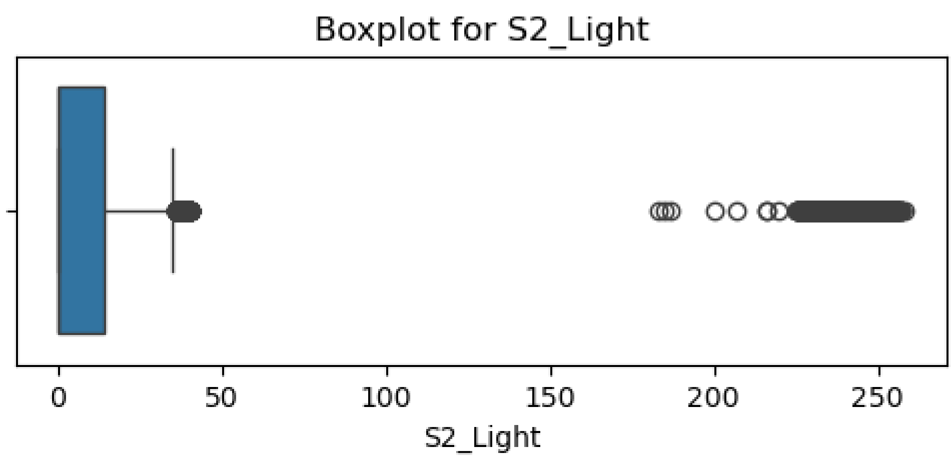

Boxplot for S1_Temp
25.0
25.2
25.4
25.6
25.8
26.0
26.2
26.4
S1_Temp

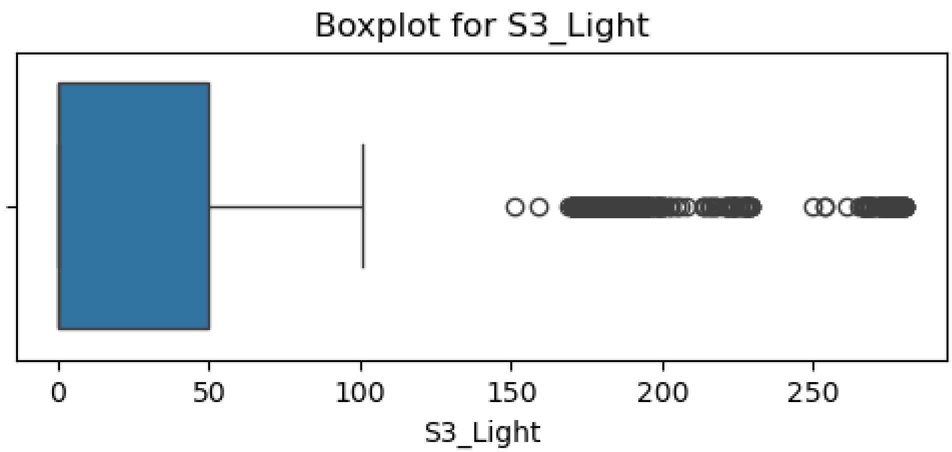

Boxplot for S2_Temp
25
26
27
28
29
S2_Temp

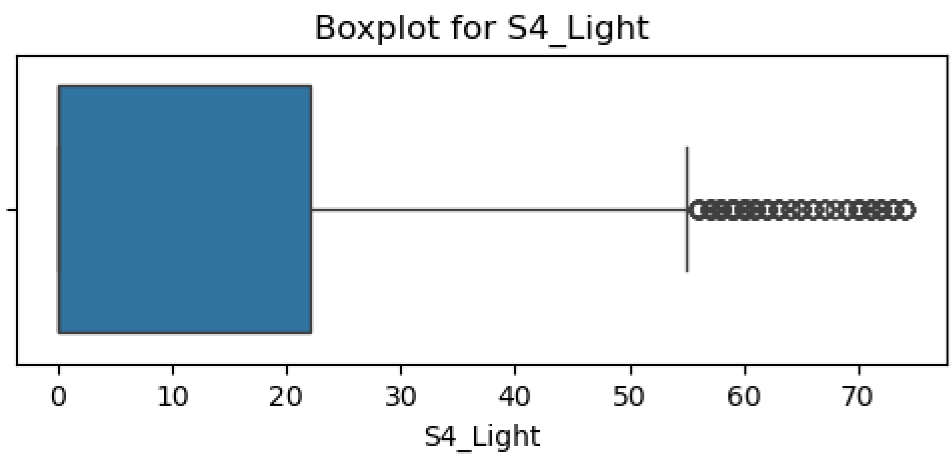

Boxplot for S3_Temp
24.50
24.75
25.00
25.25
25.50
25.75
26.00
26.25
S3_Temp

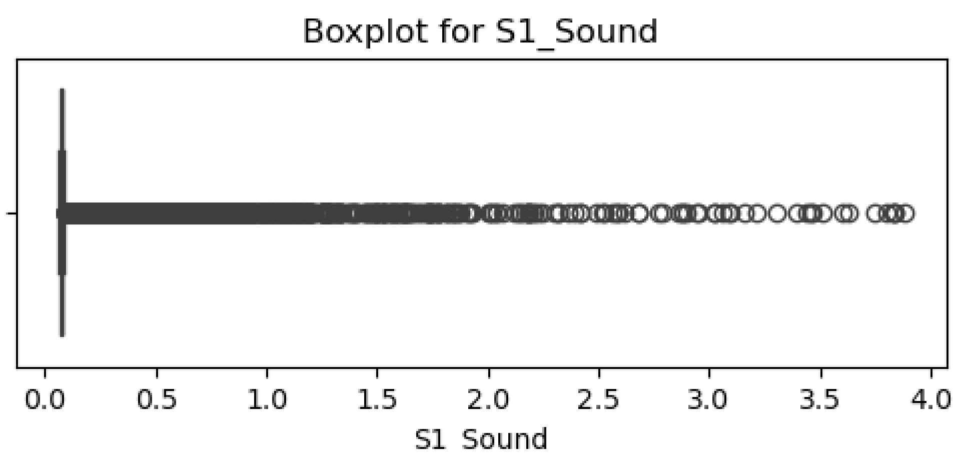

Boxplot for S4_Temp
25.0
25.2
25.4
25.6
25.8
26.0
26.2
26.4
26.6
S4_Temp

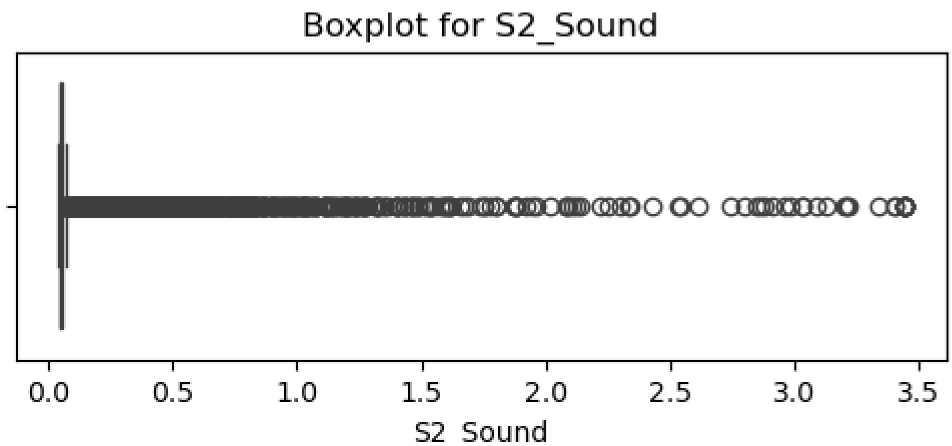

Boxplot for S1_Light
0
25
50
75
100
125
150
S1_Light

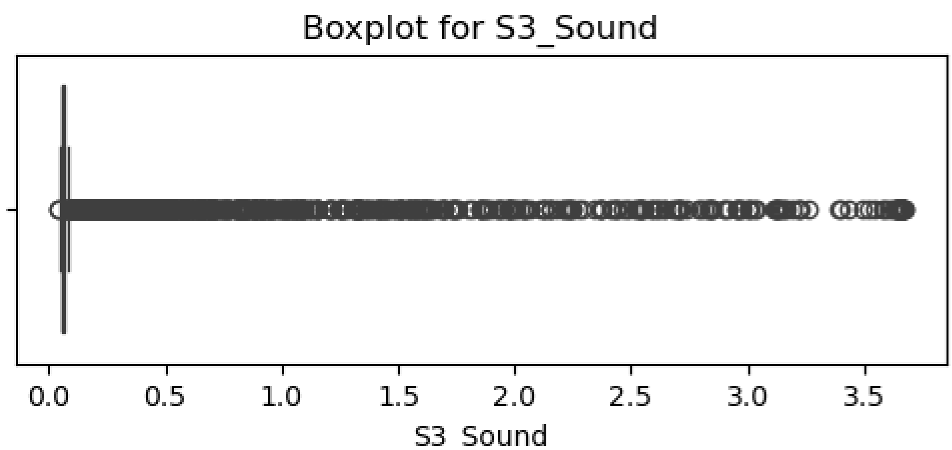

Boxplot for S2_Light
0
50
100
150
200
250
S2_Light

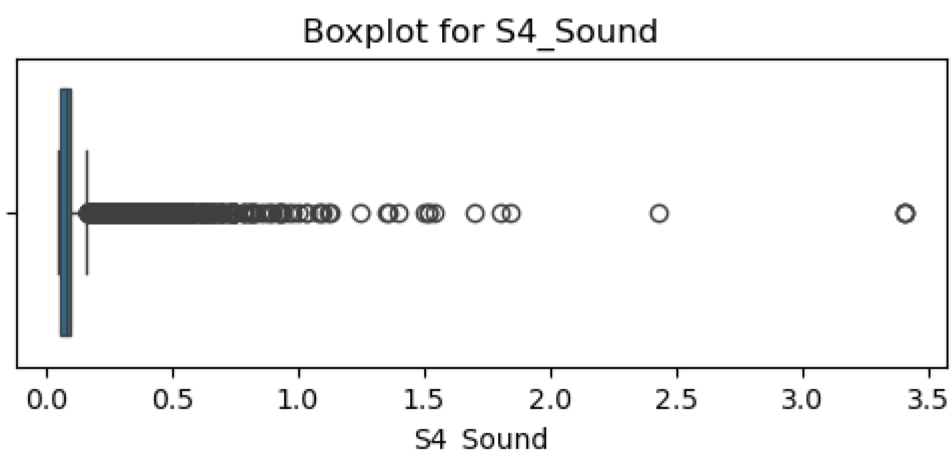

Boxplot for S3_Light
0
50
100
150
200
250
S3_Light

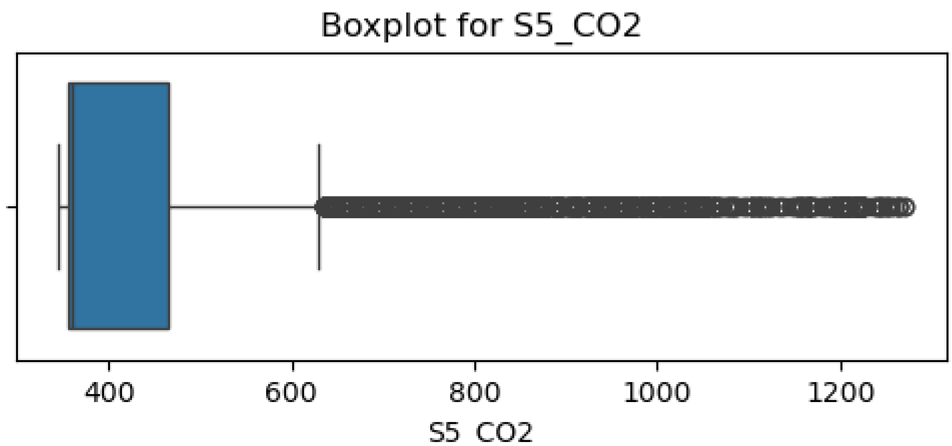

Boxplot for S4_Light
0
10
20
30
40
50
60
70
S4_Light

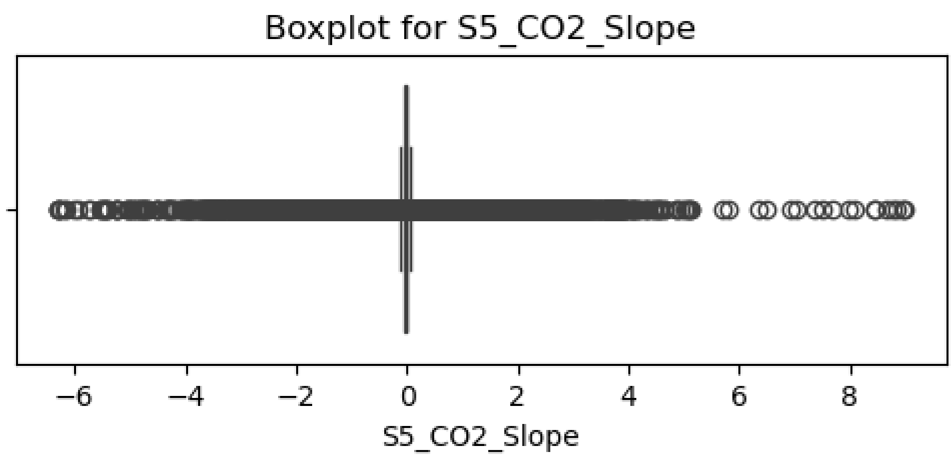

Boxplot for S1_Sound
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
S1_Sound


Boxplot for S2_Sound
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
S2_Sound


Boxplot for S3_Sound
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
S3_Sound


Boxplot for S4_Sound
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
S4_Sound

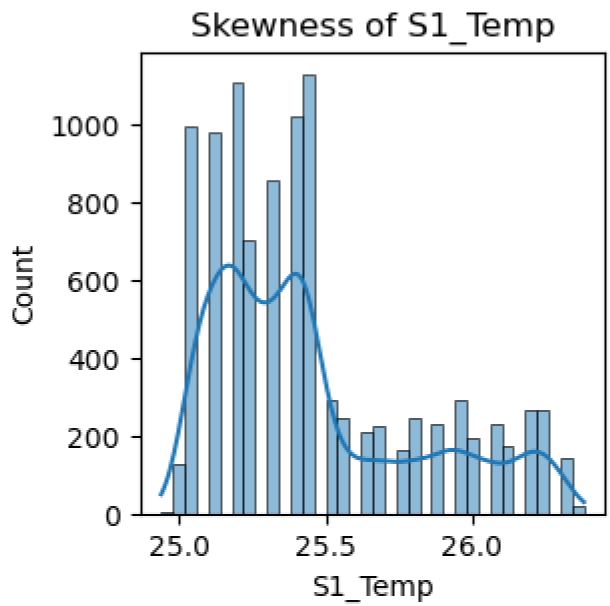


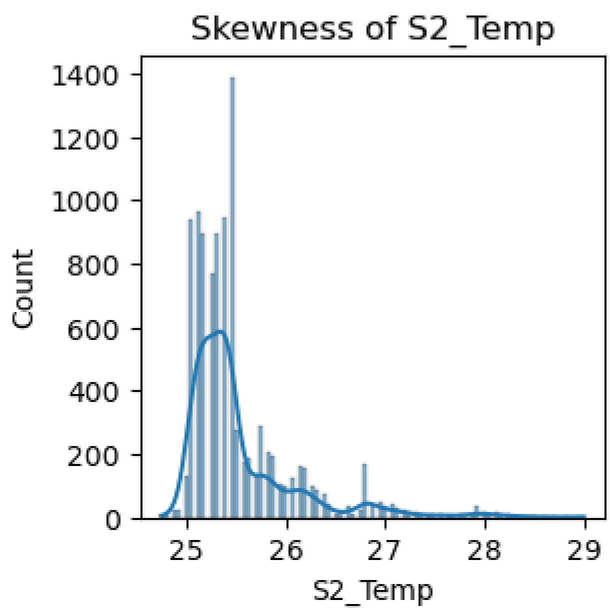


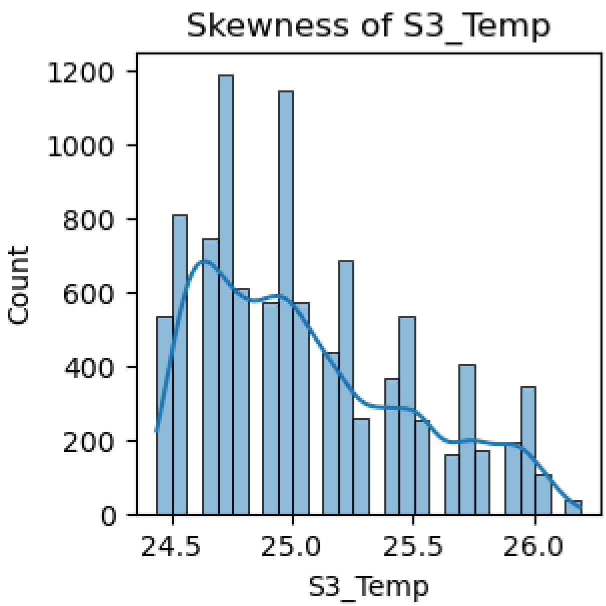
- Outlier detection: we can see extreme values as dots far from the box.
- Visualizing data distribution: It reveals how spread out or skewed each feature is.

In [61]:

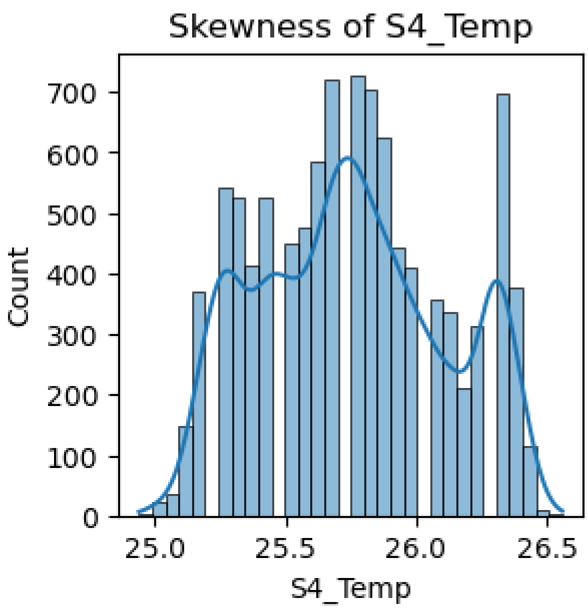
```
# skewness check
for col in num_cols:
    plt.figure(figsize=(3, 3))
    sns.histplot(df[col], kde=True)
    plt.title(f'Skewness of {col}')
    plt.show()
```


Skewness of S1_Temp
Count
S1_Temp
0
200
400
600
800
1000
25.0
25.5
26.0

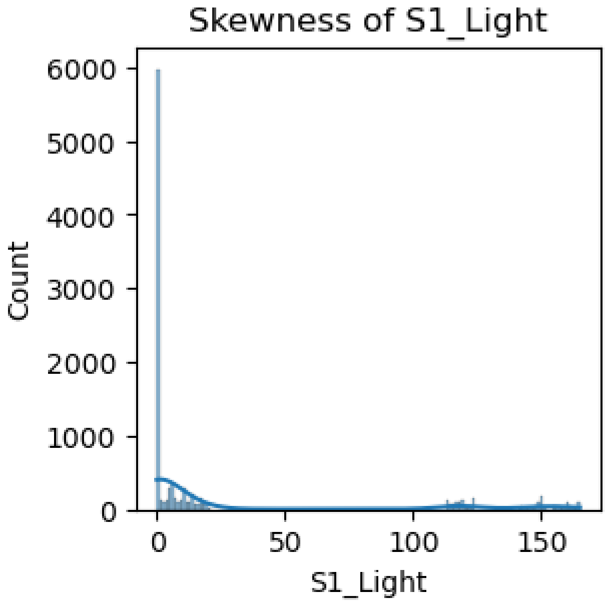

Skewness of S2_Temp
Count
S2_Temp
0
200
400
600
800
1000
1200
1400
25
26
27
28
29

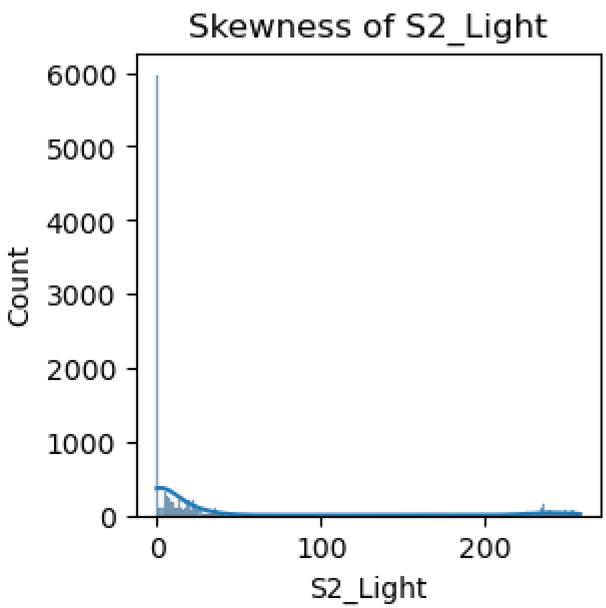


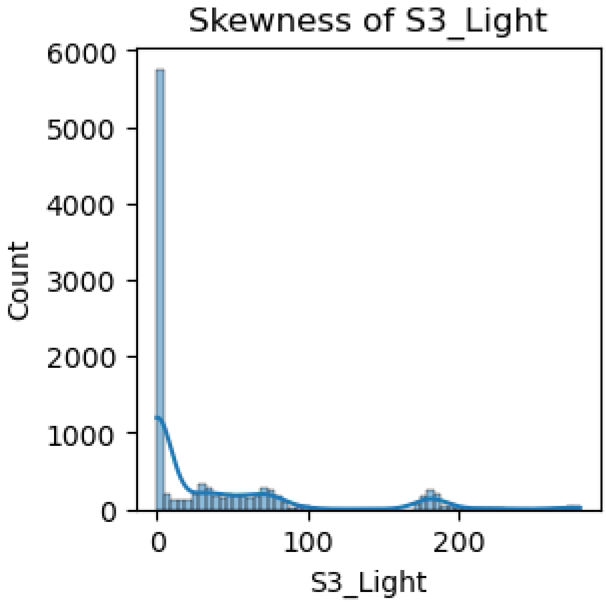
Skewness of S3_Temp
Count
S3_Temp
0
200
400
600
800
1000
1200
24.5
25.0
25.5
26.0

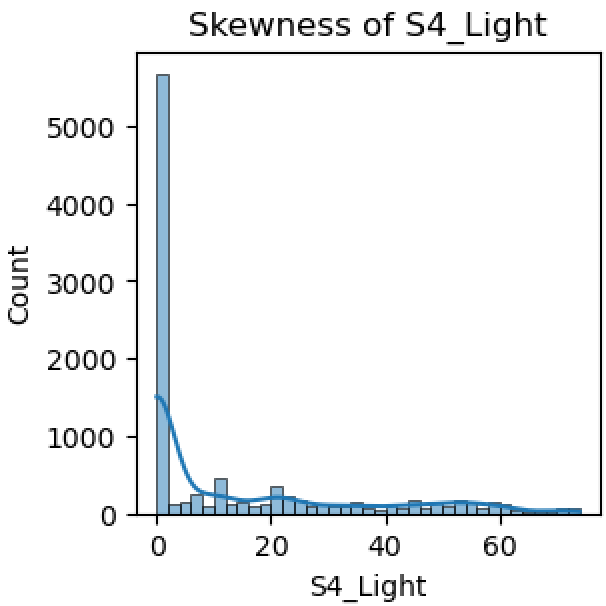


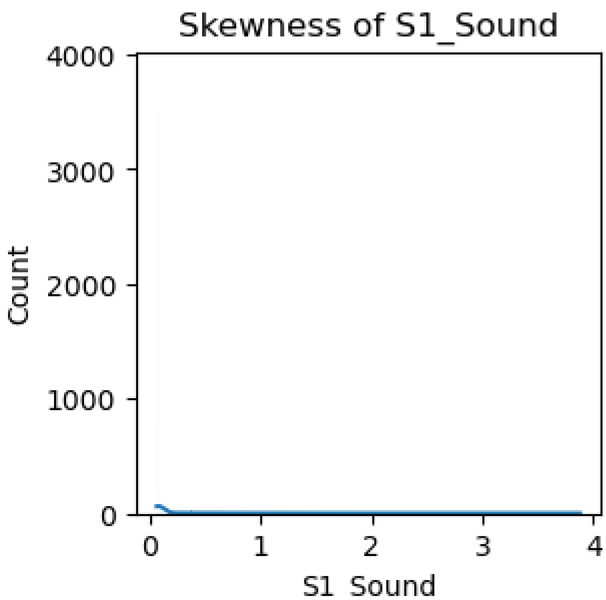
Skewness of S4_Temp
Count
S4_Temp
0
100
200
300
400
500
600
700
25.0
25.5
26.0
26.5

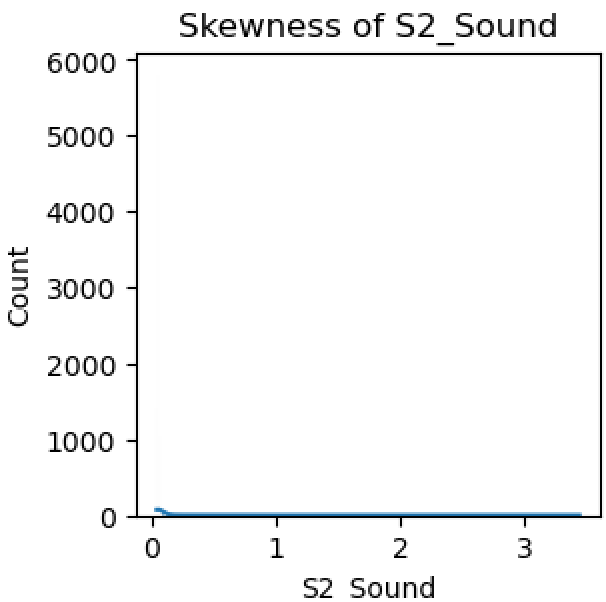

Skewness of S1_Light
Count
6000
5000
4000
3000
2000
1000
0
0
50
100
150
S1_Light

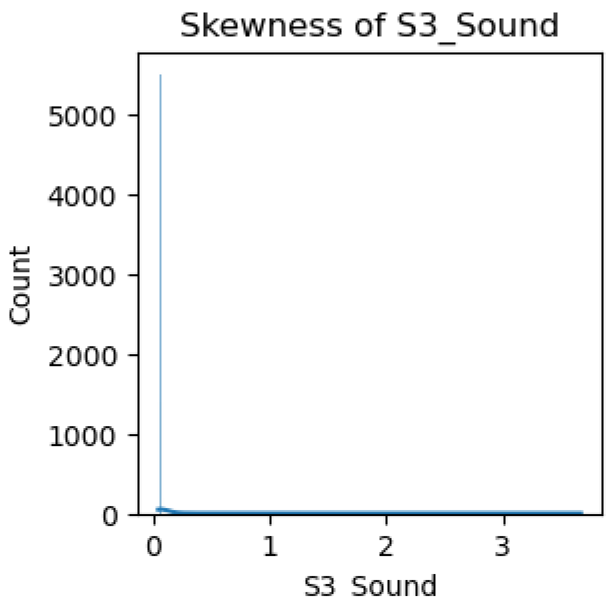

Skewness of S2_Light
Count
6000
5000
4000
3000
2000
1000
0
0
100
200
S2_Light

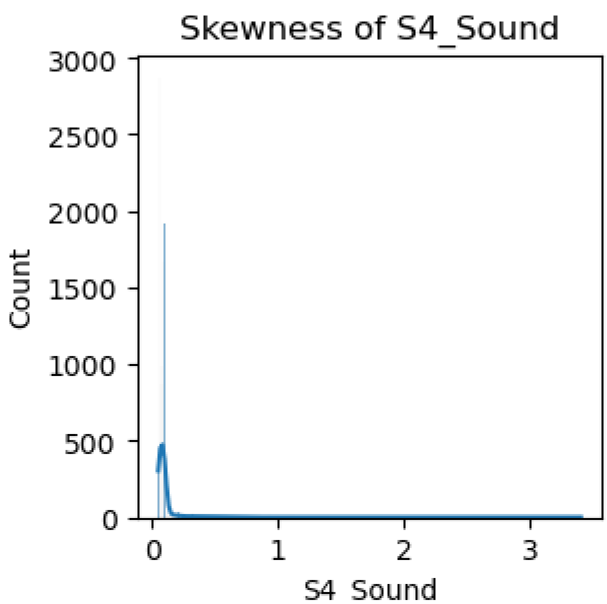

Skewness of S3_Light
Count
S3_Light
0
1000
2000
3000
4000
5000
6000
0
100
200

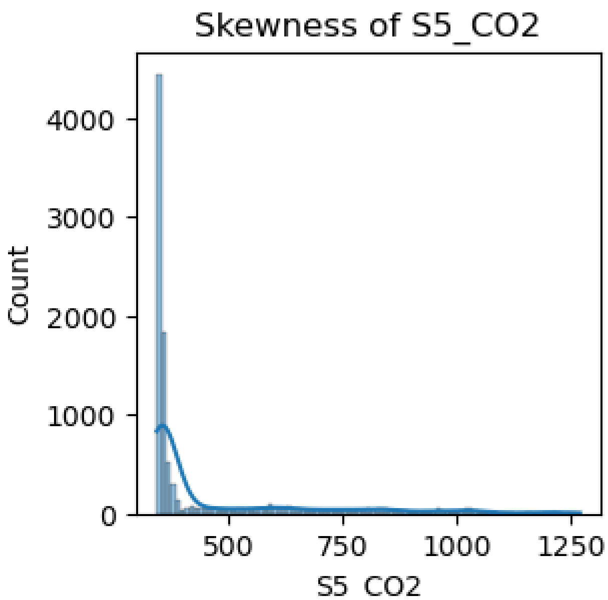

Skewness of S4_Light
Count
S4_Light
0
1000
2000
3000
4000
5000
0
20
40
60

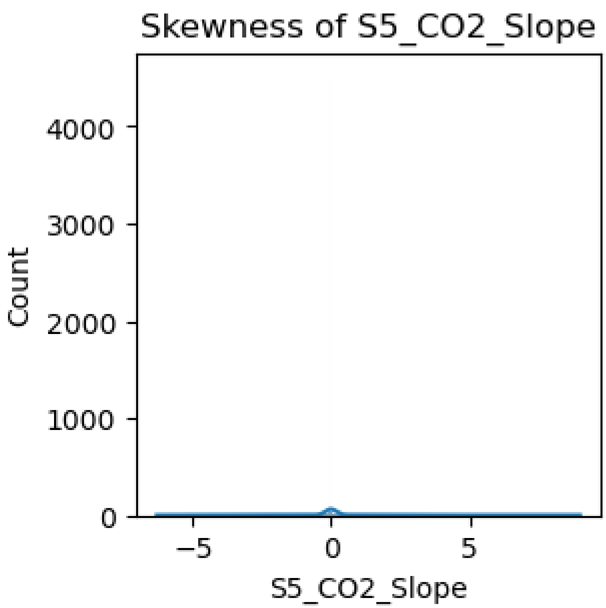

Skewness of S1_Sound
4000
3000
2000
1000
0
Count
0
1
2
3
4
S1_Sound


Skewness of S2_Sound
6000
5000
4000
3000
2000
1000
0
Count
0
1
2
3
S2_Sound


Skewness of S3_Sound
Count
5000
4000
3000
2000
1000
0
0
1
2
3
S3_Sound


Skewness of S4_Sound
Count
3000
2500
2000
1500
1000
500
0
0
1
2
3
S4_Sound





Points roughly forming a straight line: not skewed.

Curving away from the line: skewed data.

In [62]:
```
df[num_cols].skew()
```

Out[62]:
```
S1_Temp          0.953613
S2_Temp          2.355681
S3_Temp          0.650162
S4_Temp          0.129630
S1_Light         1.820428
S2_Light         2.827817
S3_Light         2.100069
S4_Light         1.357618
S1_Sound         5.450448
S2_Sound         6.881610
S3_Sound         5.994767
S4_Sound        10.952134
S5_CO2           1.975692
S5_CO2_Slope     0.287967
dtype: float64
```

Positive values means right skewed.

Identify Outliers

Identify Outliers Use the IQR method (Interquartile Range) for each numeric feature:

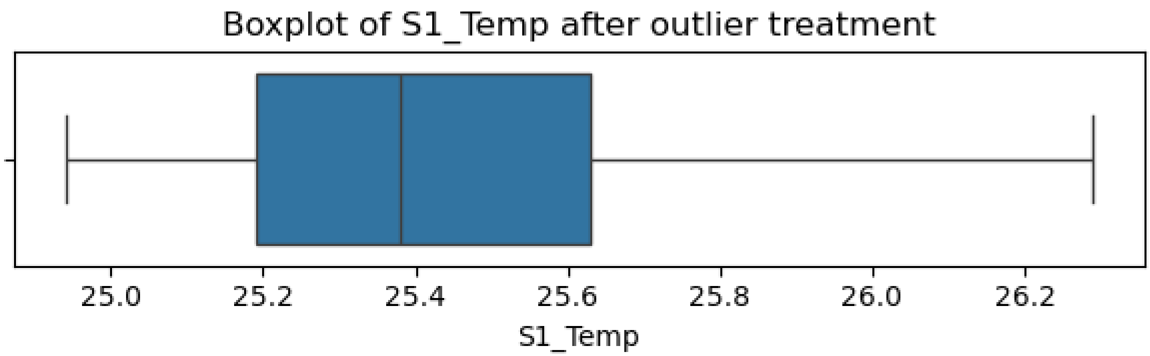
In [63]:
```python
def treat_outliers_iqr(df, features):
    for feature in features:
        Q1 = df[feature].quantile(0.25)
        Q3 = df[feature].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        # Cap outliers
        df[feature] = df[feature].clip(lower_bound, upper_bound)
    return df
```

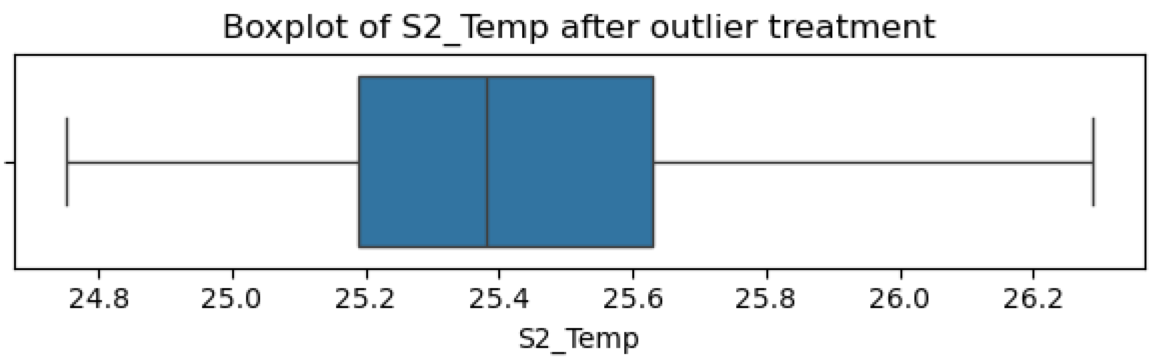
In [64]:
```python
# Apply to Sensor Features
# Choose continuous sensor features like CO₂, temperature, sound, light:

sensor_features = ['S1_Temp', 'S2_Temp', 'S3_Temp', 'S4_Temp',
                   'S1_Light', 'S2_Light', 'S3_Light', 'S4_Light',
                   'S1_Sound', 'S2_Sound', 'S3_Sound', 'S4_Sound',
                   'S5_CO2']

df = treat_outliers_iqr(df, sensor_features)
```

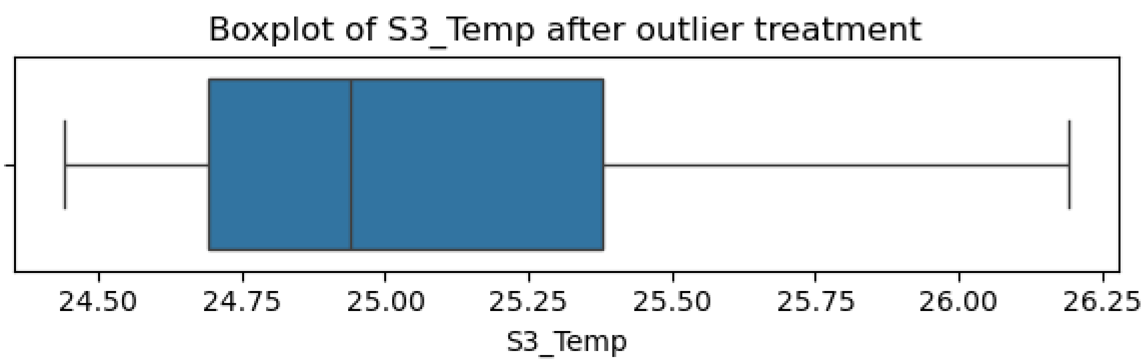
In [65]:
```python
# Then apply the same IQR-based clipping:

df = treat_outliers_iqr(df, sensor_features)
```

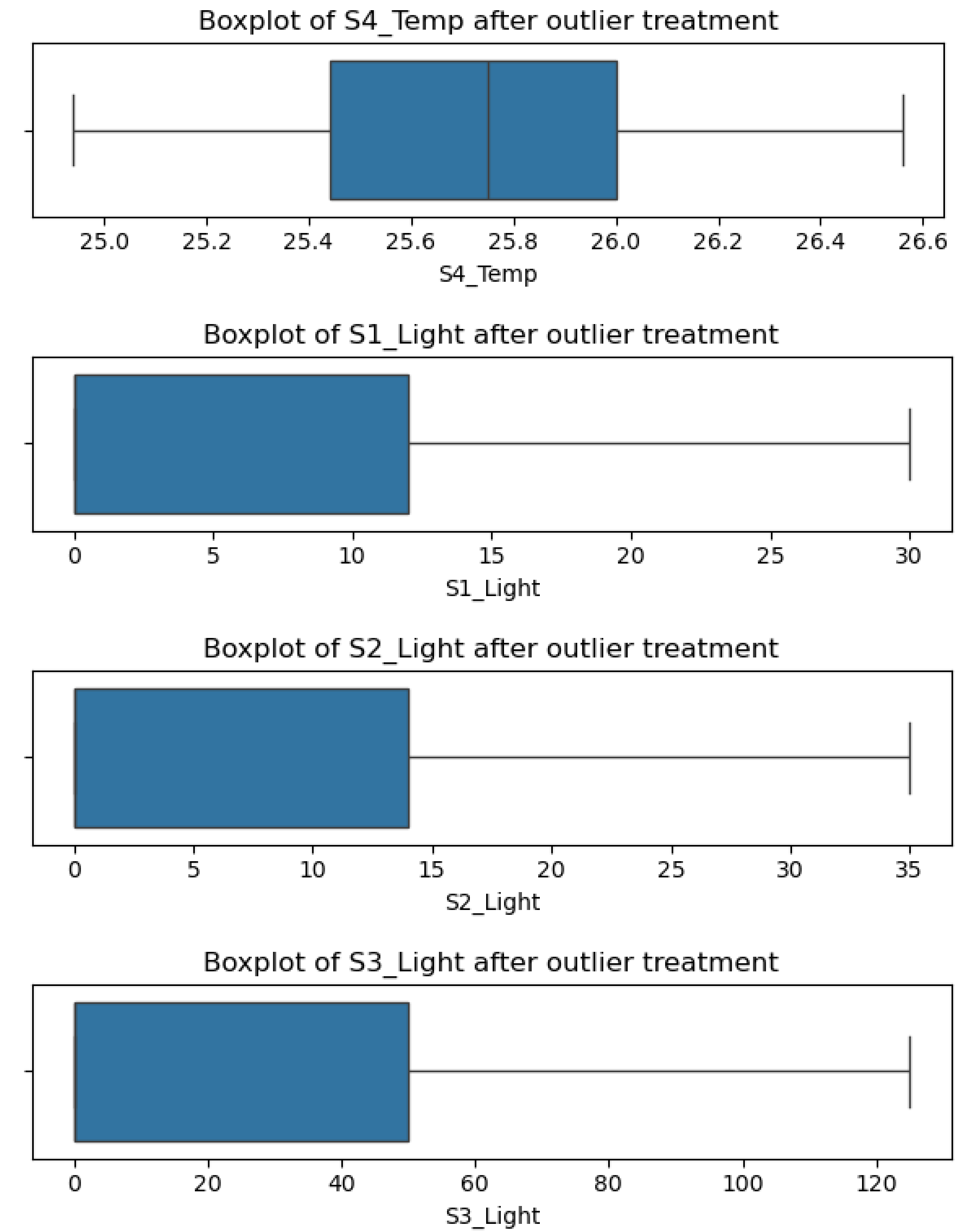
Visualize Before & After

In [66]:
```
# Visualize Before and After
# To confirm the effect:

for feature in sensor_features:
    plt.figure(figsize=(6, 2))
    sns.boxplot(x=df[feature])
    plt.title(f'Boxplot of {feature} after outlier treatment')
    plt.tight_layout()
    plt.show()
```

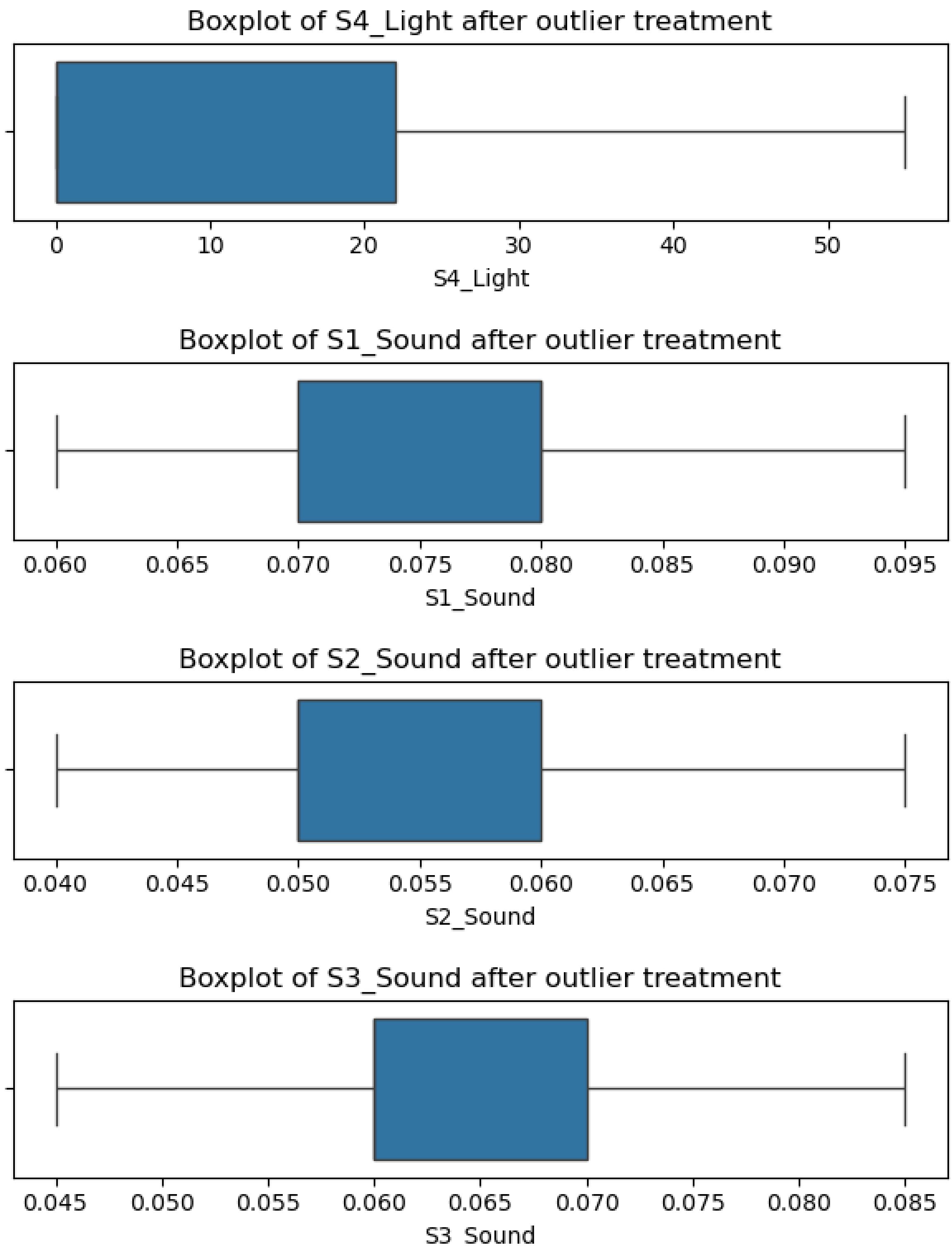


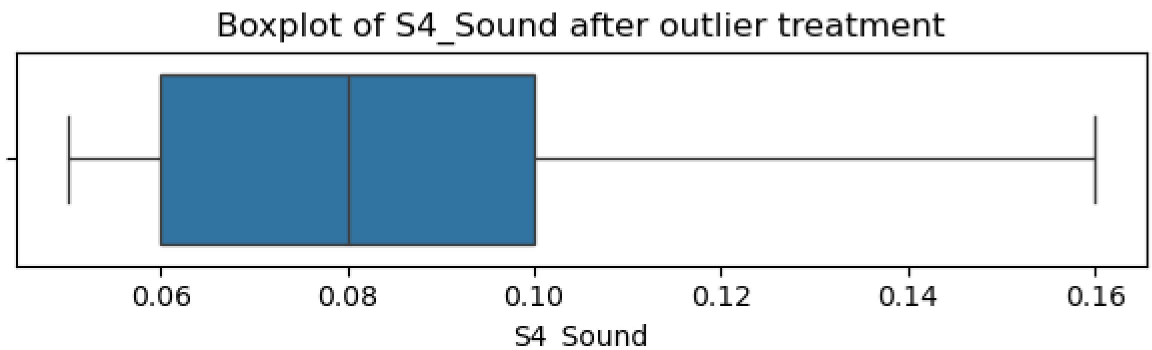


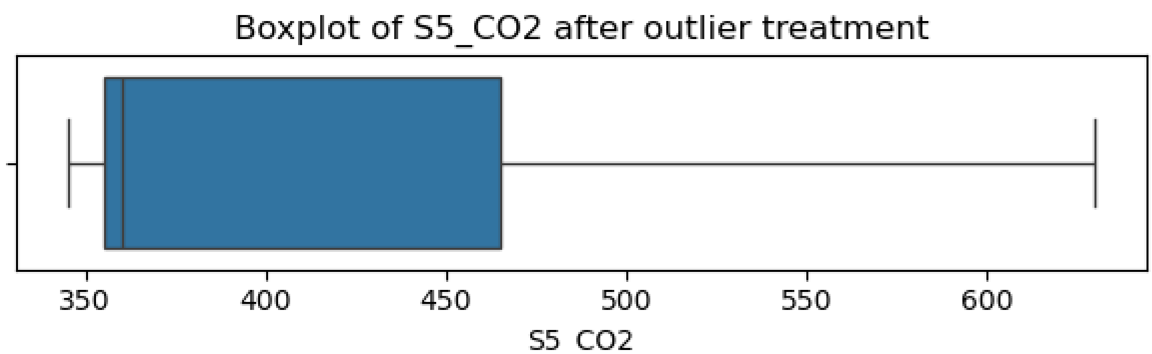




Boxplot of S4_Temp after outlier treatment
25.0
25.2
25.4
25.6
25.8
26.0
26.2
26.4
26.6
S4_Temp
Boxplot of S1_Light after outlier treatment
0
5
10
15
20
25
30
S1_Light
Boxplot of S2_Light after outlier treatment
0
5
10
15
20
25
30
35
S2_Light
Boxplot of S3_Light after outlier treatment
0
20
40
60
80
100
120
S3_Light


Boxplot of S4_Light after outlier treatment
0
10
20
30
40
50
S4_Light
Boxplot of S1_Sound after outlier treatment
0.060
0.065
0.070
0.075
0.080
0.085
0.090
0.095
S1_Sound
Boxplot of S2_Sound after outlier treatment
0.040
0.045
0.050
0.055
0.060
0.065
0.070
0.075
S2_Sound
Boxplot of S3_Sound after outlier treatment
0.045
0.050
0.055
0.060
0.065
0.070
0.075
0.080
0.085
S3_Sound



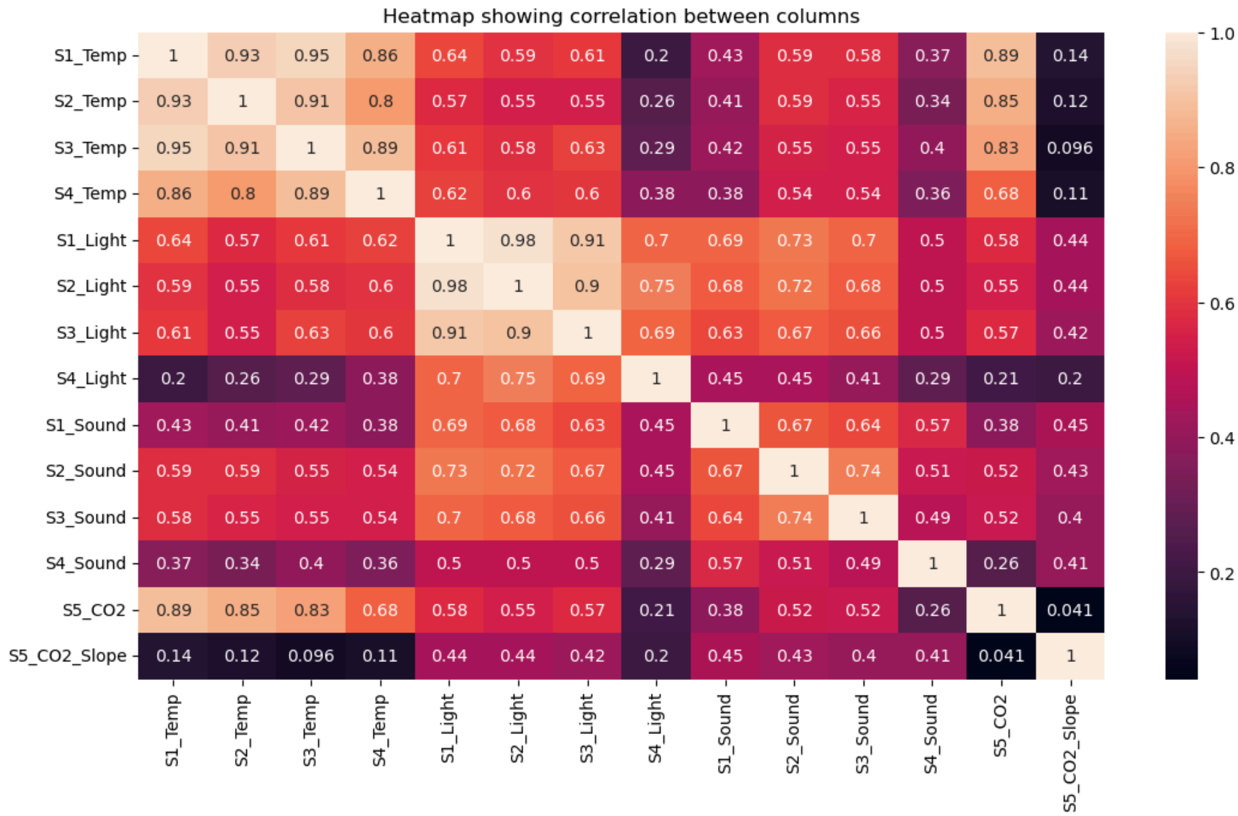


Outlier treatment is about protecting your model from noise. Including sound and light ensures you're not letting unpredictable spikes skew your predictions.

Correlation between numerical features

Assesses how sensor readings are interrelated.

In [67]:
```
plt.figure(figsize=(13,7))
sns.heatmap(df[num_cols].corr(),annot = True)
plt.title('Heatmap showing correlation between columns')
plt.show()
```



In this color-coded grid where:

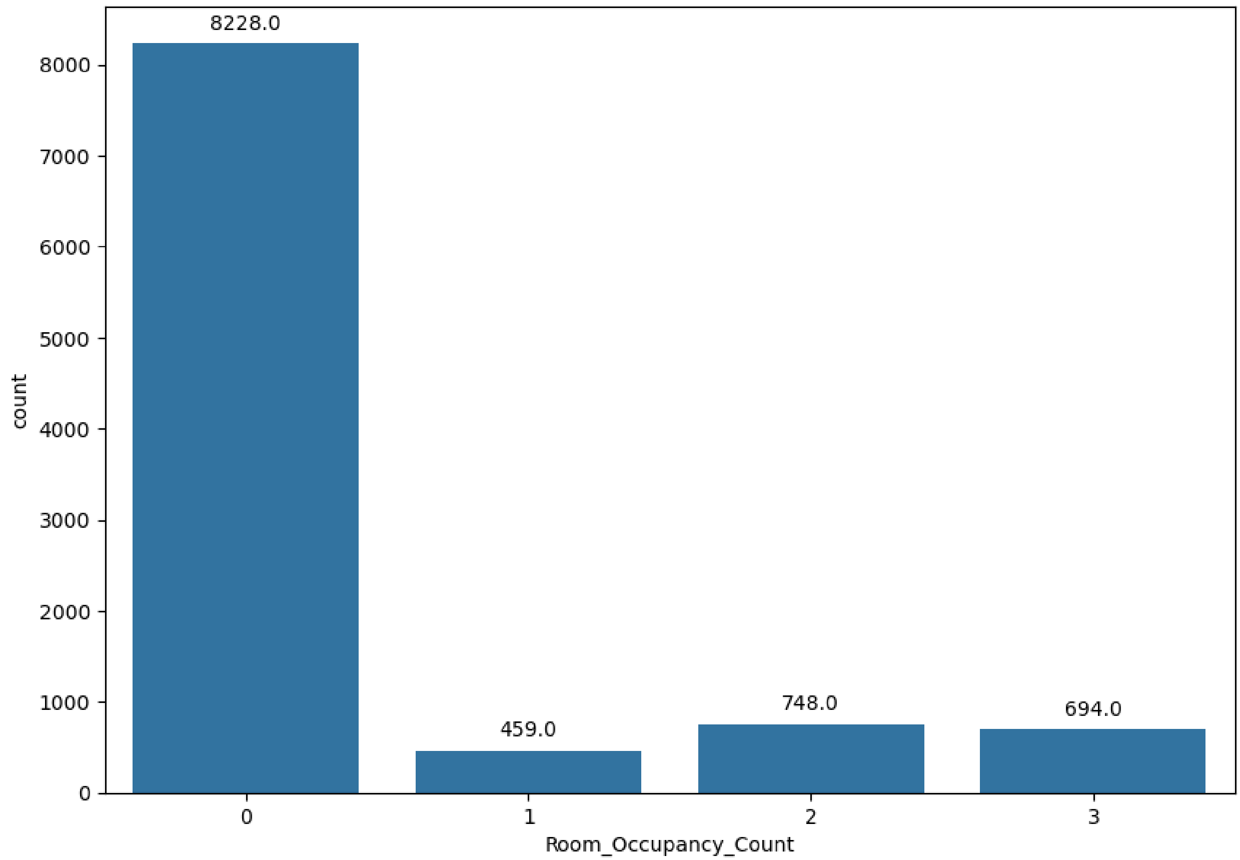
- Darker shades represent stronger correlations
- Diagonal always shows 1.00 (since each column is perfectly correlated with itself)
- Off-diagonal values reveal the relationships between different columns

Count of Room Occupancy

Purpose:

- Shows how frequently each occupancy level (e.g., 0, 1, 2...) appears.
- Helps detect class imbalance — if some occupancy levels are rare, models may struggle to predict them.

In [68]:
```
plt.figure(figsize=(10,7))
ax = sns.countplot(data = df,x = 'Room_Occupancy_Count')
for p in ax.patches:
    ax.annotate(f'{p.get_height()}', (p.get_x() + p.get_width() / 2., p.get_he
                ha='center', va='center', xytext=(0, 10), textcoords='offset p
plt.show()
```



In this bar chart:

- X-axis showing each possible number of occupants.
- Y-axis showing how often that number appeared.
- Numeric labels on top of each bar indicating the count.

Result:

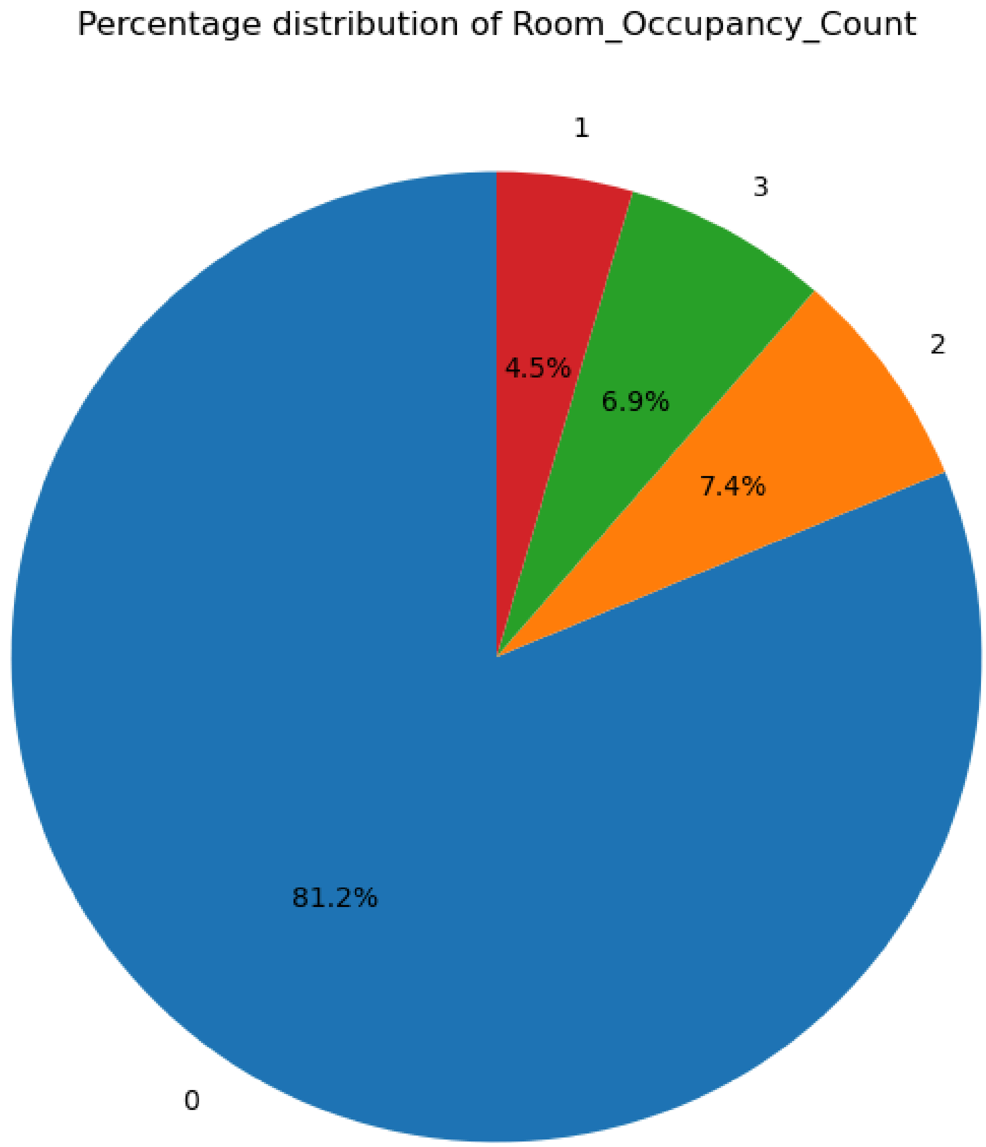
- Bars with numeric labels show the count of each class.
- If one class (e.g., 0 occupants) dominates, may need balancing techniques.

Class Distribution Pie Chart

Purpose:

- Visualizes the proportion of each occupancy level.
- Complements the count plot by showing relative frequency.

In [69]:
```
value_counts = df['Room_Occupancy_Count'].value_counts()

percentages = (value_counts / value_counts.sum()) * 100

plt.figure(figsize=(8, 8))
plt.pie(percentages, labels=percentages.index, autopct='%1.1f%%', startangle=9
plt.title('Percentage distribution of Room_Occupancy_Count')
plt.show()
```



A pie chart where:

- Each slice represents a unique room occupancy count (e.g., 0, 1, 2...)
- Sizes are proportional to how often that count appears in your dataset

- Labels show both the value and its percentage of the total

Result:

- Larger slices = more common occupancy levels.
- Helps you understand how balanced your target variable is.

Data preprocessing and Label Encoding

Purpose:

- Converts categorical time-of-day labels (e.g., "Morning", "Evening") into numeric values.
- Required for machine learning models that only accept numeric input.

In [70]:
```
label_encoder = LabelEncoder()
df['Time_of_Day'] = label_encoder.fit_transform(df['Time_of_Day'])
```

In [71]:
```
df1 = df.copy()
```

In [72]:
```
df1.drop(columns=['Date','Time','Date_time','S1_Temp','S3_Temp'],axis=1,inplac
```

Purpose:

- Removes redundant or highly correlated features to reduce noise and improve model performance.
- Also drops non-numeric columns like date/time which aren't needed for prediction.

In [73]:
```
df1.head()
```

Out[73]:

	Time_of_Day	S2_Temp	S4_Temp	S1_Light	S2_Light	S3_Light	S4_Light	S1_:
0	2	24.75	25.38	30	34	53	40	
1	2	24.75	25.44	30	33	53	40	
2	2	24.75	25.44	30	34	53	40	
3	2	24.75	25.44	30	34	53	40	
4	2	24.75	25.44	30	34	54	40	

In [74]:
```
df1.shape
```

Out[74]: (10129, 16)

In [75]:
```
df1.columns
```

Out[75]:
```
Index(['Time_of_Day', 'S2_Temp', 'S4_Temp', 'S1_Light', 'S2_Light', 'S3_Ligh
t',
       'S4_Light', 'S1_Sound', 'S2_Sound', 'S3_Sound', 'S4_Sound', 'S5_CO2',
       'S5_CO2_Slope', 'S6_PIR', 'S7_PIR', 'Room_Occupancy_Count'],
      dtype='object')
```

Model Evaluation

Purpose:

- Splits data into training and testing sets.
- Scales features to have mean 0 and variance 1 — essential for models like Logistic Regression and KNN.

In [76]:
```
X = df1.drop('Room_Occupancy_Count', axis=1)
y = df1['Room_Occupancy_Count']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, randc
```

In [77]:
```
X_train
```

Out[77]:

	Time_of_Day	S2_Temp	S4_Temp	S1_Light	S2_Light	S3_Light	S4_Light
1937	3	25.19	25.56	0	0	0	0
4477	3	25.38	25.75	0	0	0	0
8550	1	25.63	25.69	0	0	0	0
1346	3	25.50	25.88	0	0	0	0
7296	3	25.38	25.75	0	0	0	0
...	...	...	...	...	...	...	...
5734	2	25.31	25.88	17	20	75	54
5191	2	25.13	25.44	5	5	28	19
5390	2	25.06	25.50	9	10	45	30
860	1	26.29	26.44	30	35	125	10
7270	3	25.31	25.75	0	0	0	0

8103 rows × 15 columns

In [78]:
```
y_train
```

Out[78]:
```
1937    0
4477    0
8550    0
1346    0
7296    0
       ..
5734    0
5191    0
5390    0
860     3
7270    0
Name: Room_Occupancy_Count, Length: 8103, dtype: int64
```

In [79]:
```
# Initialize scaler
scaler = StandardScaler()

# Fit on training data and transform both sets
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

In [80]:
```
X_train_scaled
```

Out[80]:
```
array([[ 1.0948157 , -0.73943044, -0.53651182, ...,  0.01289966,
        -0.31109253, -0.29159811],
       [ 1.0948157 , -0.24506931, -0.00358958, ...,  0.01289966,
        -0.31109253, -0.29159811],
       [-0.65684619,  0.40540586, -0.17188081, ..., -2.20336051,
        -0.31109253, -0.29159811],
       ...,
       [ 0.21898476, -1.07767752, -0.70480305, ...,  0.01289966,
        -0.31109253, -0.29159811],
       [-0.65684619,  2.1226603 ,  1.93175962, ...,  2.71910458,
        -0.31109253,  3.42937748],
       [ 1.0948157 , -0.42720236, -0.00358958, ...,  0.01289966,
        -0.31109253, -0.29159811]])
```

In [81]:
```
X_test_scaled
```

Out[81]:
```
array([[-0.65684619, -0.42720236, -0.87309429, ...,  0.01289966,
        -0.31109253, -0.29159811],
       [-1.53267713,  0.87374798,  1.56712861, ...,  0.65081431,
        -0.31109253, -0.29159811],
       [-0.65684619,  0.71763394,  0.52933267, ..., -1.87453853,
        -0.31109253, -0.29159811],
       ...,
       [ 1.0948157 , -0.89554448, -1.40601653, ...,  0.01289966,
        -0.31109253, -0.29159811],
       [-0.65684619, -0.42720236, -1.04138552, ..., -0.02655898,
        -0.31109253, -0.29159811],
       [-0.65684619,  0.40540586, -0.00358958, ..., -0.82888462,
        -0.31109253, -0.29159811]])
```

Purpose:

- Trains multiple models and compares their performance.
- Balances precision and recall, especially useful for imbalanced classes.

Initialize Models and Compare Accuracy

Model Evaluation Using F1 Score and Accuracy

Purpose:

- Loops through each model in models dictionary.
- Trains the model on X_train_scaled and predicts on X_test_scaled.
- Calculates both accuracy and weighted F1 and Accuaracy score.
- Stores the results in a list of dictionaries.
- Converts that list into a DataFrame and sorts it by F1 and Accuaracy Score.
- Prints the final comparison table.

In [85]:
```
# Initialize models
models = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'Decision Tree': DecisionTreeClassifier(),
    'Gradient Boosting': GradientBoostingClassifier(),
    'KNN': KNeighborsClassifier()
}

# Initialize results list
results = []

# Loop through each model
for name, model in models.items():
    # Train the model
    model.fit(X_train_scaled, y_train)

    # Predict on test data
    y_pred = model.predict(X_test_scaled)

    # Calculate metrics
    acc = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Append results
```

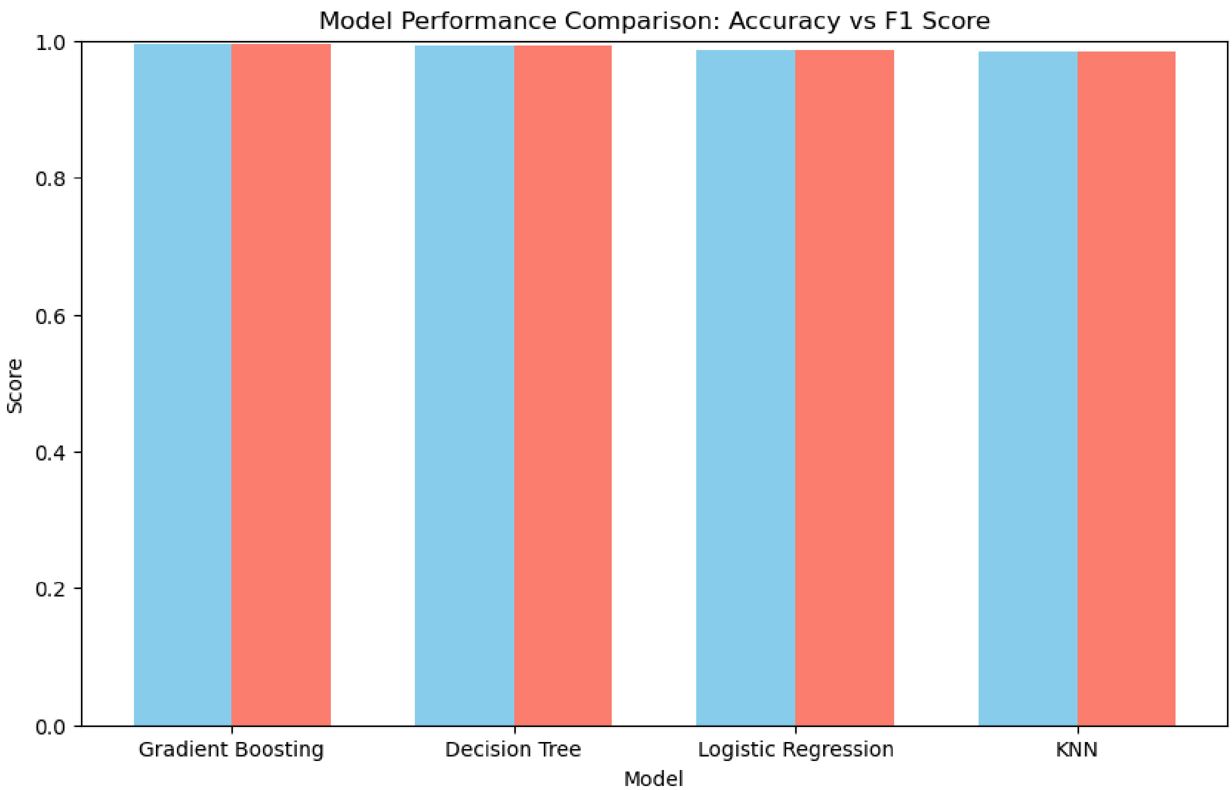
```
    results.append({
        'Model': name,
        'Accuracy': round(acc, 4),
        'F1_Score': round(f1, 4)
    })

# Convert to DataFrame
result_df = pd.DataFrame(results)

# Sort by F1 Score
result_df = result_df.sort_values(by='F1_Score', ascending=False)

# Display results
print(result_df)
```

```
                 Model  Accuracy  F1_Score
2    Gradient Boosting    0.9951    0.9951
1        Decision Tree    0.9926    0.9926
0  Logistic Regression    0.9862    0.9862
3                  KNN    0.9837    0.9838
```

Visualize model comparison using both F1 Score and Accuracy

In [86]:
```
# Plotting
plt.figure(figsize=(10, 6))
bar_width = 0.35
x = range(len(result_df))

# Bars for Accuracy
plt.bar(x, result_df['Accuracy'], width=bar_width, label='Accuracy', color='sk

# Bars for F1 Score (shifted slightly to the right)
plt.bar([i + bar_width for i in x], result_df['F1_Score'], width=bar_width, la

# Labels and titles
plt.xlabel('Model')
plt.ylabel('Score')
plt.title('Model Performance Comparison: Accuracy vs F1 Score')
plt.xticks([i + bar_width / 2 for i in x], result_df['Model'])
plt.ylim(0, 1)
```

Out[86]:
```
(0.0, 1.0)
```



- The bar chart has been successfully created! It compares Accuracy and F1 Score for five different models:
- Sky blue bars represent Accuracy.
- Salmon bars represent F1 Score.
- The chart is titled "Model Performance Comparison: Accuracy vs F1 Score".

Hyperparameter Tuning with Grid Search

Improve model performance by optimizing settings.

Purpose:

- Finds the best combination of parameters (e.g., number of trees, learning rate) for Gradient Boosting.
- Uses cross-validation to ensure robust performance.

In [87]:
```
# Define parameter grid
param_grid = {
```

```
    'n_estimators': [50, 100, 150],
    'learning_rate': [0.05, 0.1, 0.2],
    'max_depth': [3, 4, 5]
}

# Grid Search with accuracy scoring
grid = GridSearchCV(GradientBoostingClassifier(), param_grid, cv=5, scoring='a
grid.fit(X_train_scaled, y_train)
```

```
Fitting 5 folds for each of 27 candidates, totalling 135 fits
```

Out[87]:

```
▸ GridSearchCV
▸ best_estimator_: GradientBoostingClassifier
    ▸ GradientBoostingClassifier
```

Model Evaluation: Confusion Matrix & Classification Report

In [88]:

```
# Best model
best_model = grid.best_estimator_
y_pred = best_model.predict(X_test_scaled)

# Evaluation
print("◈ Best CV Accuracy:", grid.best_score_)
print("◈ Test Set Accuracy:", accuracy_score(y_test, y_pred))
print("◈ Classification Report:\n", classification_report(y_test, y_pred))
```

```
◈ Best CV Accuracy: 0.9961744388847
◈ Test Set Accuracy: 0.9965449160908193
◈ Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00      1619
           1       1.00      1.00      1.00       103
           2       0.99      0.98      0.98       164
           3       0.97      0.98      0.98       140

    accuracy                           1.00      2026
   macro avg       0.99      0.99      0.99      2026
weighted avg       1.00      1.00      1.00      2026
```

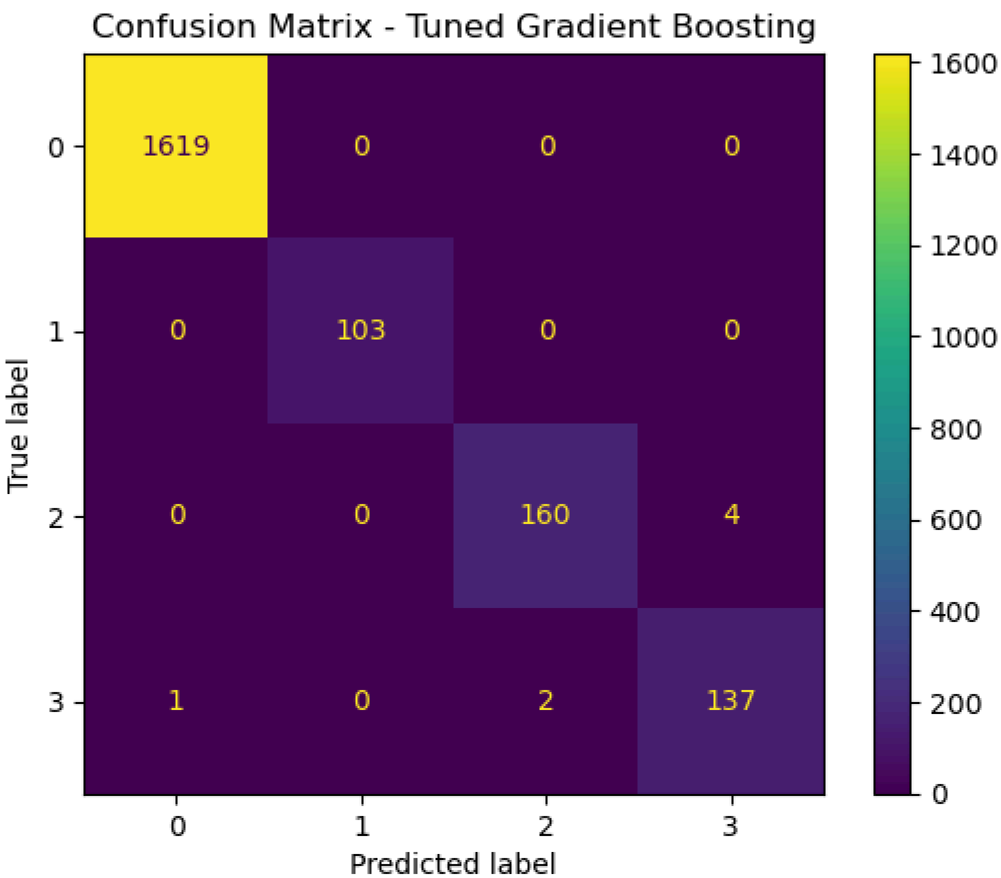
Result:

- grid.best_params_ gives the optimal settings.
- grid.best_score_ shows the best F1 score achieved during tuning.

Purpose:

- Shows how well the model predicted each class.
- Confusion matrix reveals misclassifications.
- Classification report gives precision, recall, F1 for each class.

In [89]:
```
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot()
plt.title("Confusion Matrix - Tuned Gradient Boosting")
plt.show()
```



Purpose:

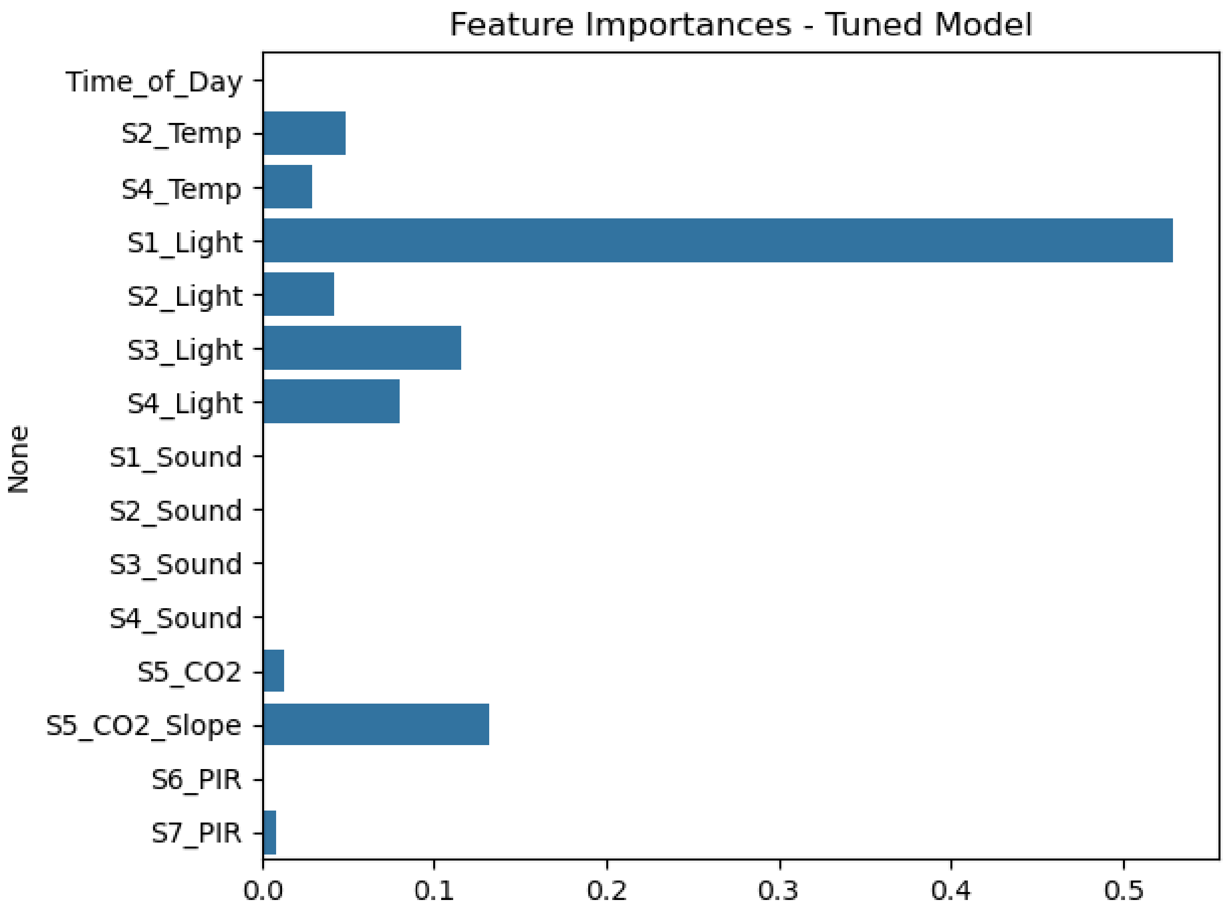
- Shows how well the model predicted each class.
- Confusion matrix reveals misclassifications.
- Classification report gives precision, recall, F1 for each class.

Feature Importance

Purpose:

- Tells you which features were most useful for prediction.
- Helps in feature selection and understanding model behavior.

In [90]:
```python
# Feature importance
importances = best_model.feature_importances_
sns.barplot(x=importances, y=X.columns)
plt.title("Feature Importances - Tuned Model")
plt.tight_layout()
plt.show()
```



- Each bar represents a feature from your dataset.
- The length of the bar shows how important that feature is.
- Features with very short bars might have lower importance.

- Features at the top (with longer bars) are the most influential in the model.

Compare Tuned vs Default Model

In [91]:

```
# Default Gradient Boosting
default_model = GradientBoostingClassifier()
default_model.fit(X_train_scaled, y_train)
default_score = cross_val_score(default_model, X, y, cv=5, scoring='accuracy')

# Tuned model score
tuned_score = grid.best_score_

print("⚖ Default Model Accuracy:", default_score)
print("� Tuned Model Accuracy:", tuned_score)
```

```
⚖ Default Model Accuracy: 0.9416460580356599
� Tuned Model Accuracy: 0.9961744388847
```

- Default Accuracy: How well the model performs without any tuning.
- Tuned Accuracy: How well the model performs after selecting the best hyperparameters.

In [92]:

```
# 8. Grid Search Results Table
results_df = pd.DataFrame(grid.cv_results_)
top_results = results_df.sort_values(by='mean_test_score', ascending=False)
print("� Top 5 Grid Search Results:")
print(top_results[['param_n_estimators', 'param_learning_rate', 'param_max_dep
print("� Best Parameters:", grid.best_params_)
```

```
� Top 5 Grid Search Results:
    param_n_estimators  param_learning_rate  param_max_depth  mean_test_score
26                 150                 0.20                5         0.996174
8                  150                 0.05                5         0.996051
16                 100                 0.10                5         0.996051
20                 150                 0.20                3         0.995928
10                 100                 0.10                3         0.995804
� Best Parameters: {'learning_rate': 0.2, 'max_depth': 5, 'n_estimators': 150}
```

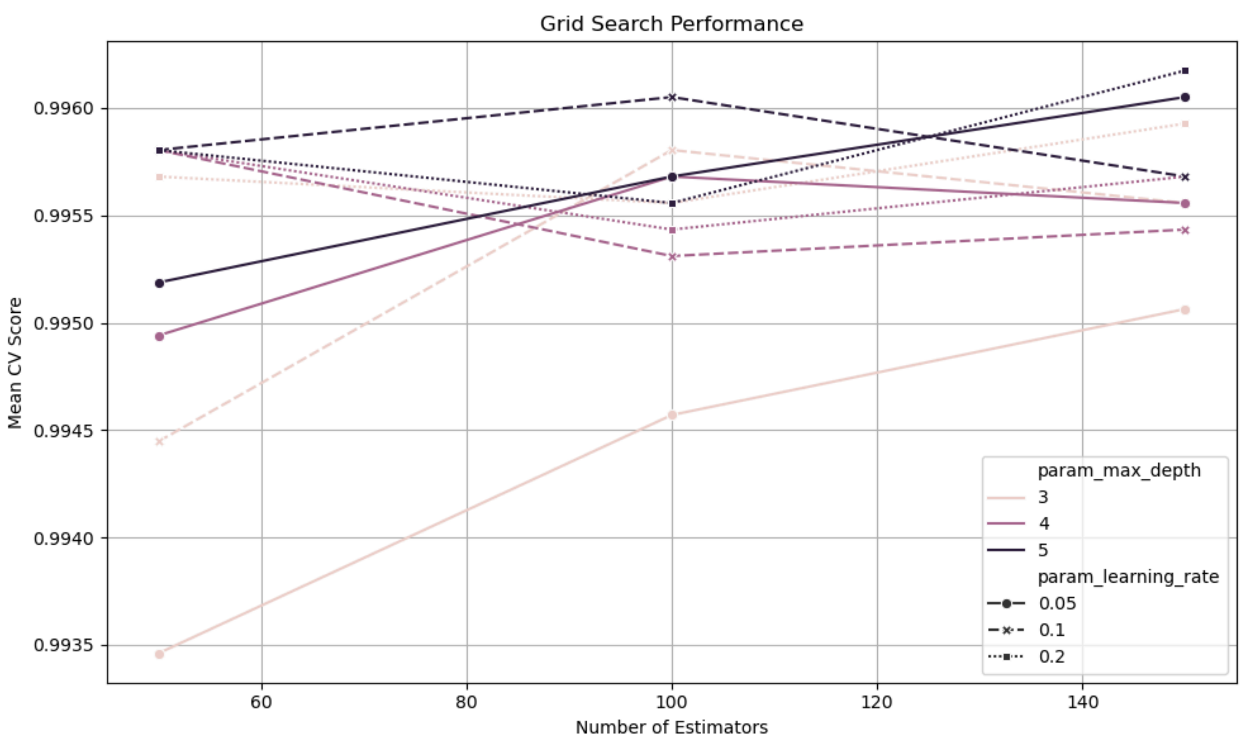
- The first row shows the best-performing combination.
- we can now choose the best hyperparameters for your final model.

Grid Search Results Visualization

Purpose:

- Visualizes how different parameter combinations affect model performance.
- Helps you understand trade-offs and trends.

In [93]:
```python
# 9. Optional: Visualize Grid Search Performance
plt.figure(figsize=(10, 6))
sns.lineplot(
    data=top_results,
    x='param_n_estimators',
    y='mean_test_score',
    hue='param_max_depth',
    style='param_learning_rate',
    markers=True
)
plt.title('Grid Search Performance')
plt.ylabel('Mean CV Score')
plt.xlabel('Number of Estimators')
plt.grid(True)
plt.tight_layout()
plt.show()
```



To get the optimal configuration:

print("Best Parameters:", grid.best_params_) print("Best Score:", grid.best_score_)

This tells:

- Which combination of hyperparameters performed best.
- The corresponding cross-validation score.

Dataset Highlights:

Total Records: 10,129

Features: 18 (including sensor readings and timestamps)

Sensors Used: Temperature, Light, Sound, CO_2, PIR

Occupancy Range: 0 to 3 people

No Missing or Duplicate Values

Time-Series Format: Includes Date, Time, and derived Time_of_Day

Key Steps

Data Cleaning & Preprocessing

Handled missing values using median imputation

Removed outliers using IQR capping

Encoded categorical features

Scaled numerical features for model input

Exploratory Data Analysis

Visualized occupancy trends across time of day

Analyzed sensor distributions and correlations

Identified skewness and treated outliers

Modeling & Evaluation:

Trained multiple classifiers: Logistic Regression, Decision Tree, Gradient Boosting, KNN

Tuned Gradient Boosting using GridSearchCV

Evaluated models using Accuracy and F1 Score

Visualized Confusion Matrix and Feature Importances

Key Insights:

Gradient Boosting outperformed other models with highest accuracy and F1 score.

Light and Sound sensors showed strong correlation with occupancy levels.

Time of Day significantly influenced occupancy patterns, with afternoons showing higher average counts.

No duplicate entries and minimal missing data ensured high data quality.

Outlier treatment improved model stability and performance.

Best Model Performance:

Metric Score:

Best CV Accuracy: 0.9960

Test Set Accuracy: 0.9935

```
       F1 Score  : 1.00

        accuracy  : 0.99
```

Technologies Used:

Python, Pandas, NumPy, Matplotlib, Seaborn

Scikit-learn (Logistic Regression, Decision Tree, Gradient Boosting, KNN)

GridSearchCV for hyperparameter tuning.