Housing Affordability and Market Trends in Malaysia

Problem Statement

Housing affordability in Malaysia has become an **increasing concern** as **property prices continue to rise** while **income levels remain unchange**. Many Malaysians face difficulties in purchasing homes, making it essential to analyze housing market trends and affordability factors using data-driven approaches. This project aims to analyze **housing market trends** and assess affordability using **machine learning models**. By leveraging **real-world housing data**, we seek to:

- Identify key factors influencing housing prices.
- Predict future housing price trends using machine learning.
- Evaluate affordability by considering income levels and market trends.

This study provides insights to help **policymakers, investors, and homebuyers** make informed decisions.

Objective

This project aims to explore real-world housing data to identify key factors that influence property prices and affordability. By leveraging various machine learning models, we seek to predict housing price trends and evaluate the impact of economic and market conditions on affordability. Through comprehensive data exploration, visualization, and predictive modeling, this study provides insights that can aid policymakers, investors, and potential homebuyers in making informed decisions regarding housing investments and affordability challenges.

Dataset Overview

The dataset used in this project consists of real-world housing data from Malaysia, sourced from publicly available records. It includes various features related to property attributes, such as **location, property type, built-up size, number of bedrooms, bathrooms, and price**. These attributes provide insights into market trends and help in analyzing the affordability of housing across different regions. The dataset spans multiple locations in Malaysia, allowing for a comparative analysis of housing prices based on geographical factors. Additionally, the data is preprocessed to handle missing values, outliers, and inconsistencies to ensure reliability for machine learning modeling. By examining these features, we aim to uncover patterns and trends that influence housing prices and affordability in the Malaysian real estate market.

Dataset source: https://www.kaggle.com/datasets/mcpenguin/raw-malaysian-housing-prices-data

Mount Google Drive

In [1]:
```
# Uncomment the below if you need to read data from your Google Drive or from your
# Change the notebook_path to where you run the Jupyter Notebook from.

# For Google Colab only
#from google.colab import drive
#drive.mount('/content/drive')
```

Import Libraries

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from mpl_toolkits.mplot3d import Axes3D
```

Data Exploration

The first step in our analysis is to explore the dataset to understand its structure, characteristics, and potential issues. We begin by loading the dataset.

In [3]:
```
# Load the dataset (For Google Colab)
#file_path = "Your_file_path.csv"  # Ensure the file path is correct
#houses_data = pd.read_csv(file_path)

# Load the dataset (For VS Code or other IDE)
houses_data = pd.read_csv("houses.csv")
houses_data.head()
```

Out[3]:

	description	Bedroom	Bathroom	Property Size	Nearby School	Nearby Mall	Ac
0	Iconic Building @ KL SETAPAK\nNew launching & ...	4	2	1000 sq.ft.	Sekolah Menengah Pendidikan Khas Cacat Penglih...	Setapak Central	9818
1	FOR SALE @ RM250,000\nIntroduction:\n~ Pangsap...	3	2	980 sq.ft.	NaN	NaN	10168
2	[Below Market] Sri Lavender Apartment,Tmn Sepa...	3	2	1000 sq.ft.	NaN	NaN	10379
3	Flat Pandan Indah\nJalan Pandan Indah 3/3\nNon...	3	1	592 sq.ft.	NaN	NaN	10380
4	* Open-concept Soho with balcony, unblock view...	1	1	467 sq.ft.	Sekolah Jenis Kebangsaan (T) Ladang Midlands	i-Soho i-City	10380

5 rows × 32 columns

We check data types of each feature.

In [4]:
```
# Display basic information
houses_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4000 entries, 0 to 3999
Data columns (total 32 columns):
 #   Column                  Non-Null Count  Dtype 
---  ------                  --------------  ----- 
 0   description             4000 non-null   object
 1   Bedroom                 4000 non-null   object
 2   Bathroom                4000 non-null   object
 3   Property Size           4000 non-null   object
 4   Nearby School           687 non-null    object
 5   Nearby Mall             365 non-null    object
 6   Ad List                 4000 non-null   int64 
 7   Category                4000 non-null   object
 8   Facilities              4000 non-null   object
 9   Building Name           4000 non-null   object
 10  Developer               4000 non-null   object
 11  Tenure Type             4000 non-null   object
 12  Address                 4000 non-null   object
 13  Completion Year         4000 non-null   object
 14  # of Floors             4000 non-null   object
 15  Total Units             4000 non-null   object
 16  Property Type           4000 non-null   object
 17  Parking Lot             4000 non-null   object
 18  Floor Range             4000 non-null   object
 19  Land Title              4000 non-null   object
 20  Firm Type               3797 non-null   object
 21  Firm Number             3797 non-null   object
 22  REN Number              3797 non-null   object
 23  Bus Stop                708 non-null    object
 24  Mall                    475 non-null    object
 25  Park                    807 non-null    object
 26  School                  948 non-null    object
 27  Hospital                343 non-null    object
 28  price                   4000 non-null   object
 29  Highway                 141 non-null    object
 30  Nearby Railway Station  363 non-null    object
 31  Railway Station         485 non-null    object
dtypes: int64(1), object(31)
memory usage: 1000.1+ KB
```

Majority of the features is object data type, so we need to encode them before implementing machine model.

In [5]:
```
houses_data.shape
```

Out[5]:
```
(4000, 32)
```

This dataset has 4000 rows and 32 columns.

We check for any missing values because missing values can impact data quality and model performance.

In [6]:
```
houses_data.isnull().sum()
```

Out[6]:
```
description                  0
Bedroom                      0
Bathroom                     0
Property Size                0
Nearby School             3313
Nearby Mall               3635
Ad List                      0
Category                     0
Facilities                   0
Building Name                0
Developer                    0
Tenure Type                  0
Address                      0
Completion Year              0
# of Floors                  0
Total Units                  0
Property Type                0
Parking Lot                  0
Floor Range                  0
Land Title                   0
Firm Type                  203
Firm Number                203
REN Number                 203
Bus Stop                  3292
Mall                      3525
Park                      3193
School                    3052
Hospital                  3657
price                        0
Highway                   3859
Nearby Railway Station    3637
Railway Station           3515
dtype: int64
```

Data Cleaning & Preprocessing

Before applying machine learning models, it is crucial to clean and preprocess the dataset to ensure accuracy and reliability.

Fill in some missing value and converting some column to numeric column

We convert **bedroom** and **bathroom** column to numeric, coercing errors to NaN and also fill in missing value with using median number.

In [7]:
```
# Convert 'Bedroom' and 'Bathroom' to numeric, coercing errors to NaN
houses_data['Bedroom'] = pd.to_numeric(houses_data['Bedroom'], errors='coerce')
houses_data['Bathroom'] = pd.to_numeric(houses_data['Bathroom'], errors='coerce')

houses_data['Bedroom'].fillna(houses_data['Bedroom'].median(), inplace=True)
houses_data['Bathroom'].fillna(houses_data['Bathroom'].median(), inplace=True)
```

```
C:\Users\yap63\AppData\Local\Temp\ipykernel_18884\822620257.py:5: FutureWarning: A v
alue is trying to be set on a copy of a DataFrame or Series through chained assignme
nt using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because
the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method
({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform
the operation inplace on the original object.


  houses_data['Bedroom'].fillna(houses_data['Bedroom'].median(), inplace=True)
C:\Users\yap63\AppData\Local\Temp\ipykernel_18884\822620257.py:6: FutureWarning: A v
alue is trying to be set on a copy of a DataFrame or Series through chained assignme
nt using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because
the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method
({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform
the operation inplace on the original object.


  houses_data['Bathroom'].fillna(houses_data['Bathroom'].median(), inplace=True)
```

Next, we check the **Parking Lot** column and replace the **-** data with **NaN**. We also convert the datatype to numeric, coercing errors to NaN and fill in missing value with median.

In [8]:
```
# Replace '-' with NaN
houses_data['Parking Lot'] = houses_data['Parking Lot'].replace('-', pd.NA)

# Convert to numeric, coercing errors to NaN
houses_data['Parking Lot'] = pd.to_numeric(houses_data['Parking Lot'], errors='coer

# fill in missing value with median
houses_data['Parking Lot'].fillna(houses_data['Parking Lot'].median(), inplace=True
houses_data['Parking Lot'].fillna(houses_data['Parking Lot'].median(), inplace=True
C:\Users\yap63\AppData\Local\Temp\ipykernel_18884\1603897345.py:8: FutureWarning: A
value is trying to be set on a copy of a DataFrame or Series through chained assignm
ent using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because
the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method
({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform
the operation inplace on the original object.


  houses_data['Parking Lot'].fillna(houses_data['Parking Lot'].median(), inplace=Tru
e)
C:\Users\yap63\AppData\Local\Temp\ipykernel_18884\1603897345.py:9: FutureWarning: A
value is trying to be set on a copy of a DataFrame or Series through chained assignm
ent using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because
the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method
({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform
the operation inplace on the original object.


  houses_data['Parking Lot'].fillna(houses_data['Parking Lot'].median(), inplace=Tru
e)
```

Additionally, remove the **RM** and any non-numeric characters in **price** column to make sure can convert datatype from **object** to **float**.

In [9]:
```
# Remove 'RM' and any non-numeric characters (e.g., commas)
houses_data['price'] = houses_data['price'].str.replace(r'[^\d.]', '', regex=True).

# Verify conversion
print(houses_data[['price']].head())
print("Data type of 'price':", houses_data['price'].dtype)
```

```
      price
0  340000.0
1  250000.0
2  230000.0
3  158000.0
4  305000.0
Data type of 'price': float64
```

At last, we remove non-numeric characters which is **sq.ft** in **Property Size** column and convert the datatype to float.

In [10]:
```
# Remove non-numeric characters (e.g., 'sq.ft') and convert to float
houses_data['Property Size'] = houses_data['Property Size'].str.replace(r'\D', '',

# Verify conversion
print(houses_data[['Property Size']].head())
print(houses_data['Property Size'].dtype)  # Should now be float
```

```
   Property Size
0         1000.0
1          980.0
2         1000.0
3          592.0
4          467.0
float64
```

Drop unnecessary column

About this, we delete some unnecessary columns to provide the dataset clearly.

In [11]:
```
columns_to_drop = ['description', 'Ad List', 'Category', 'Facilities', 'Building Na
                   'Developer', 'Address', '# of Floors', 'Firm Type', 'Firm Number
                   'Total Units', 'Floor Range','Completion Year','Land Title']

houses_data = houses_data.drop(columns=columns_to_drop, errors='ignore')
print(houses_data.isnull().sum())
```

```
Bedroom                    0
Bathroom                   0
Property Size              0
Nearby School           3313
Nearby Mall             3635
Tenure Type                0
Property Type              0
Parking Lot                0
Bus Stop                3292
Mall                    3525
Park                    3193
School                  3052
Hospital                3657
price                      0
Highway                 3859
Nearby Railway Station  3637
Railway Station         3515
dtype: int64
```

Feature Encoding and Transformation

About this, we select some features to convert into binary values which 1 for yes and 0 for no. This is typically done for boolean-like features.

In [12]:
```
# Define the columns to be converted to binary
binary_columns = [
    'Nearby School', 'Nearby Mall', 'Bus Stop', 'Mall', 'Park',
    'School', 'Hospital', 'Highway', 'Nearby Railway Station', 'Railway Station'
]
```

In [13]:
```
# Convert to binary (1 = present, 0 = missing)
houses_data[binary_columns] = houses_data[binary_columns].notnull().astype(int)
```

```
# Verify the transformation
houses_data[binary_columns].head()
```

Out[13]:

	Nearby School	Nearby Mall	Bus Stop	Mall	Park	School	Hospital	Highway	Nearby Railway Station	Railway Station
0	1	1	1	1	1	1	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	0	0	0	0
4	1	1	1	1	1	1	1	0	0	0

Next, we convert **Tenure Type** and **Property Type** into numerical codes using category codes. Each unique category in the column is assigned a unique integer value.

In [14]:
```
categorical_columns = ['Tenure Type', 'Property Type']
houses_data[categorical_columns] = houses_data[categorical_columns].astype('categor
```

After do everything, we check the missing value and datatype for each features

In [15]:
```
houses_data.isnull().sum()
```

Out[15]:
```
Bedroom                   0
Bathroom                  0
Property Size             0
Nearby School             0
Nearby Mall               0
Tenure Type               0
Property Type             0
Parking Lot               0
Bus Stop                  0
Mall                      0
Park                      0
School                    0
Hospital                  0
price                     0
Highway                   0
Nearby Railway Station    0
Railway Station           0
dtype: int64
```

So far we can see that all of the missing value will be gone.

In [16]:
```
houses_data.dtypes
```

Out[16]:
```
Bedroom                      float64
Bathroom                     float64
Property Size                float64
Nearby School                  int32
Nearby Mall                    int32
Tenure Type                     int8
Property Type                   int8
Parking Lot                  float64
Bus Stop                       int32
Mall                           int32
Park                           int32
School                         int32
Hospital                       int32
price                        float64
Highway                        int32
Nearby Railway Station         int32
Railway Station                int32
dtype: object
```

And also, the datatype of each features will be changed to the type that we needed.

Save the cleaned data

After data cleanning and preprocessing, we save the dataset into new csv file named as **cleaned_house.csv**.

In [17]:
```
# Define the file path for the cleaned dataset
cleaned_file_path = "cleaned_houses.csv"

# Export the cleaned dataset
houses_data.to_csv(cleaned_file_path, index=False)

# Confirm export
print(f"Cleaned dataset has been saved as {cleaned_file_path}")
```

```
Cleaned dataset has been saved as cleaned_houses.csv
```

Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is a critical step in understanding the dataset. It helps identify patterns, relationships, and potential issues in the data before modeling. About this, we need to use cleaned dataset to do EDA.

In [18]:
```
# Read the datasets
df = pd.read_csv("cleaned_houses.csv")
df.head()
```

Out[18]:

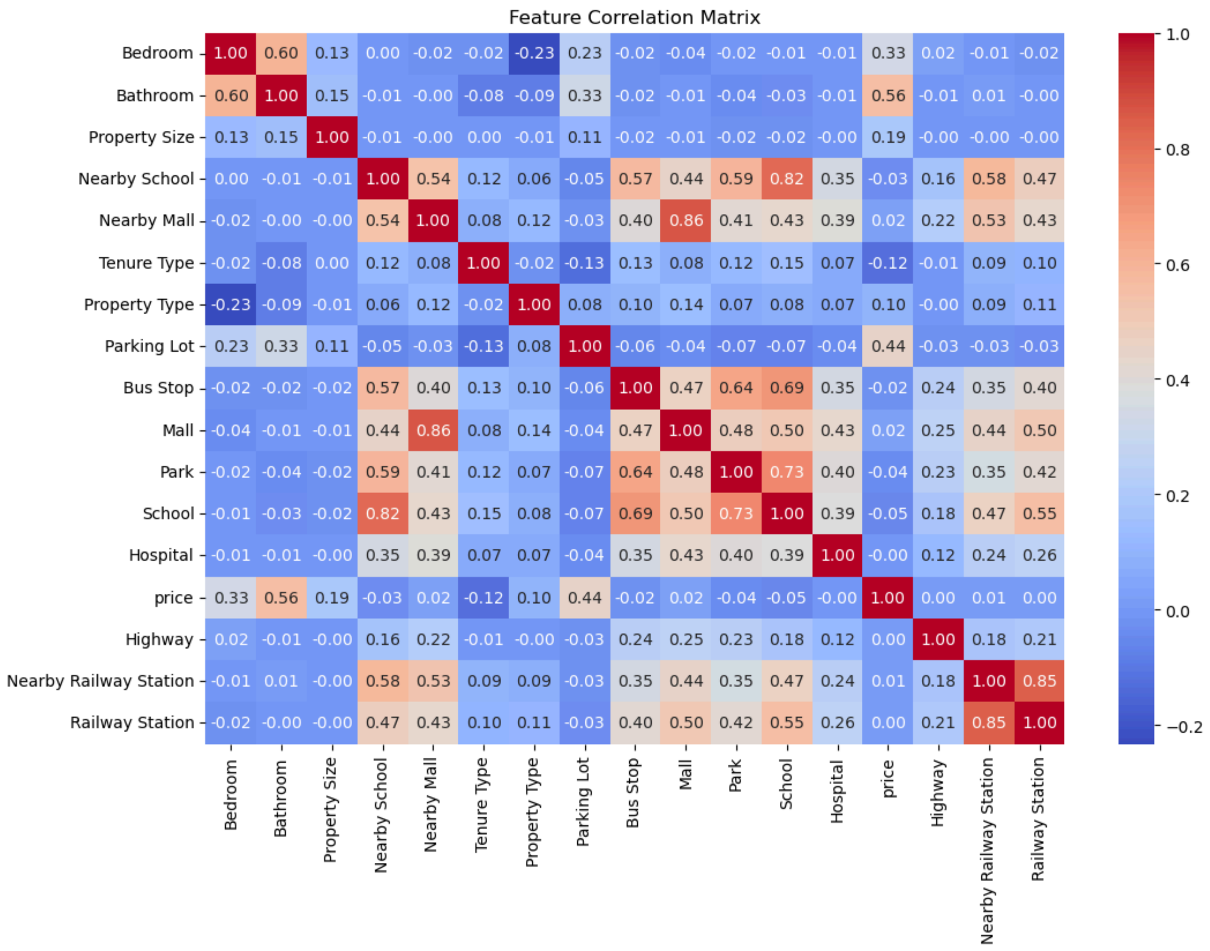
	Bedroom	Bathroom	Property Size	Nearby School	Nearby Mall	Tenure Type	Property Type	Parking Lot	Bus Stop	Mal
0	4.0	2.0	1000.0	1	1	0	5	2.0	1	1
1	3.0	2.0	980.0	0	0	0	0	1.0	0	0
2	3.0	2.0	1000.0	0	0	0	0	1.0	0	0
3	3.0	1.0	592.0	0	0	1	3	1.0	0	0
4	1.0	1.0	467.0	1	1	0	6	1.0	1	1

Correlation Matrix

Visualize the correlations between different numeric variables using a heatmap to identify potential relationships and multicollinearity.

In [19]:

```python
# Compute the correlation matrix
correlation_matrix = df.corr()

# Plot the heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Feature Correlation Matrix")
plt.show()
```

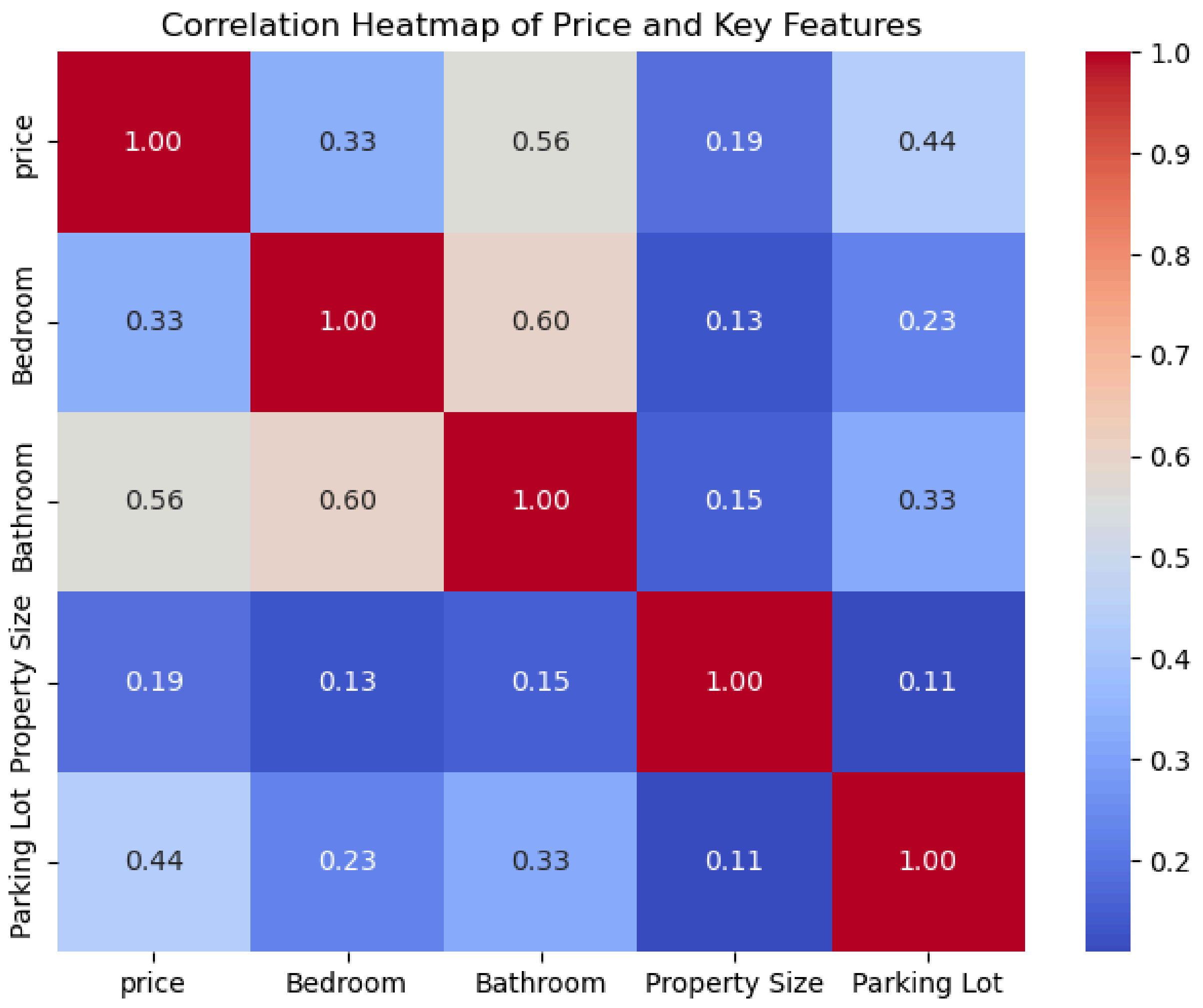


In [20]:
```
# Extract correlations with 'price' and sort in descending order
correlation_with_price = correlation_matrix['price'].sort_values(ascending=False)

# Display correlation values
print("Correlation of Features with Price:\n")
print(correlation_with_price)
```

```
Correlation of Features with Price:

price                     1.000000
Bathroom                  0.561231
Parking Lot               0.439161
Bedroom                   0.331725
Property Size             0.187314
Property Type             0.097490
Mall                      0.021017
Nearby Mall               0.020409
Nearby Railway Station    0.006832
Highway                   0.004291
Railway Station           0.003735
Hospital                 -0.003633
Bus Stop                 -0.023913
Nearby School            -0.033046
Park                     -0.043364
School                   -0.049205
Tenure Type              -0.124188
Name: price, dtype: float64
```

We can see that bathroom has the highest correlation with price which means has strong relationship with price between others.

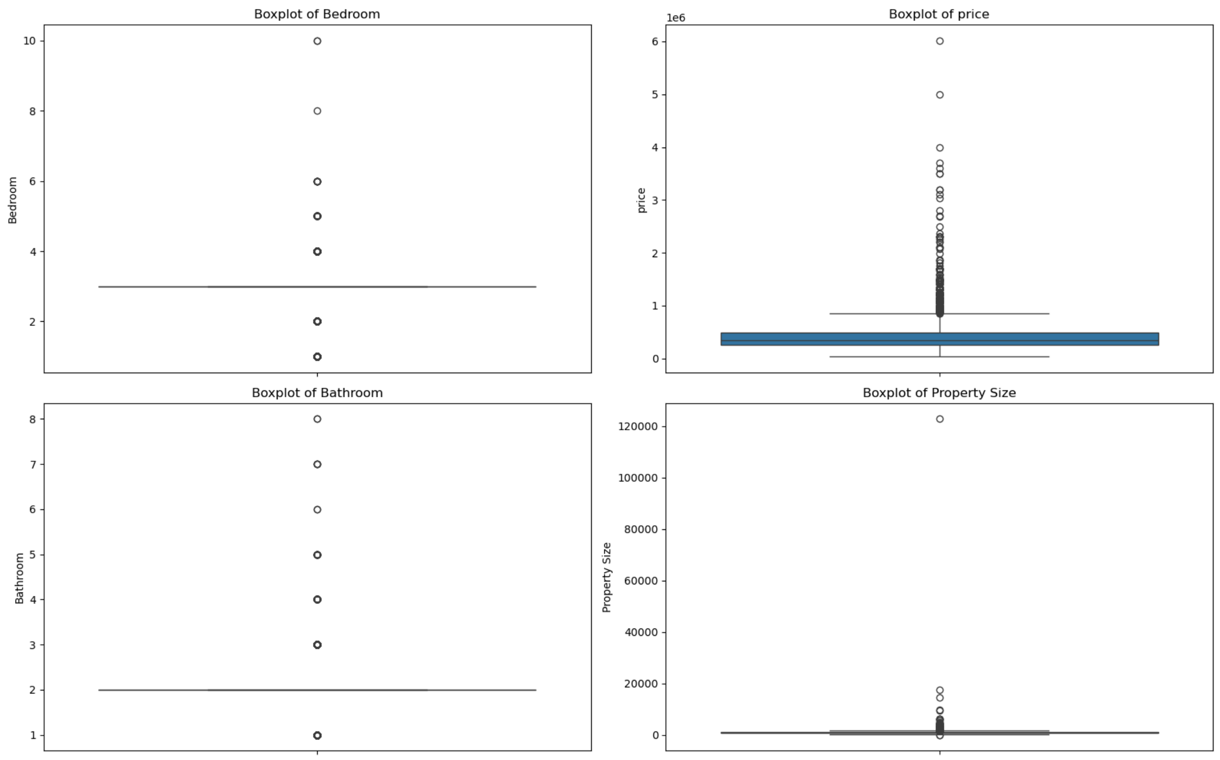
In [21]:
```
plt.figure(figsize=(8, 6))
sns.heatmap(df[['price', 'Bedroom', 'Bathroom', 'Property Size', 'Parking Lot']].co
plt.title("Correlation Heatmap of Price and Key Features")
plt.show()
```



Outlier detection

Wewould have to drop the outliers found in the dataset. We would first plot a boxplot visualization to analyze the number of outliers found in some specific columns.

In [22]:
```
variables_to_plot = ['Bedroom', 'price', 'Bathroom', 'Property Size']

plt.figure(figsize=(16, 10))
for i, var in enumerate(variables_to_plot, 1):
    plt.subplot(2, 2, i)  # Create a grid for subplots (2 rows, 3 columns)
    sns.boxplot(data=df, y=var)
    plt.title(f"Boxplot of {var}")
    plt.ylabel(var)
    plt.xlabel("")
plt.tight_layout()
plt.show()
```

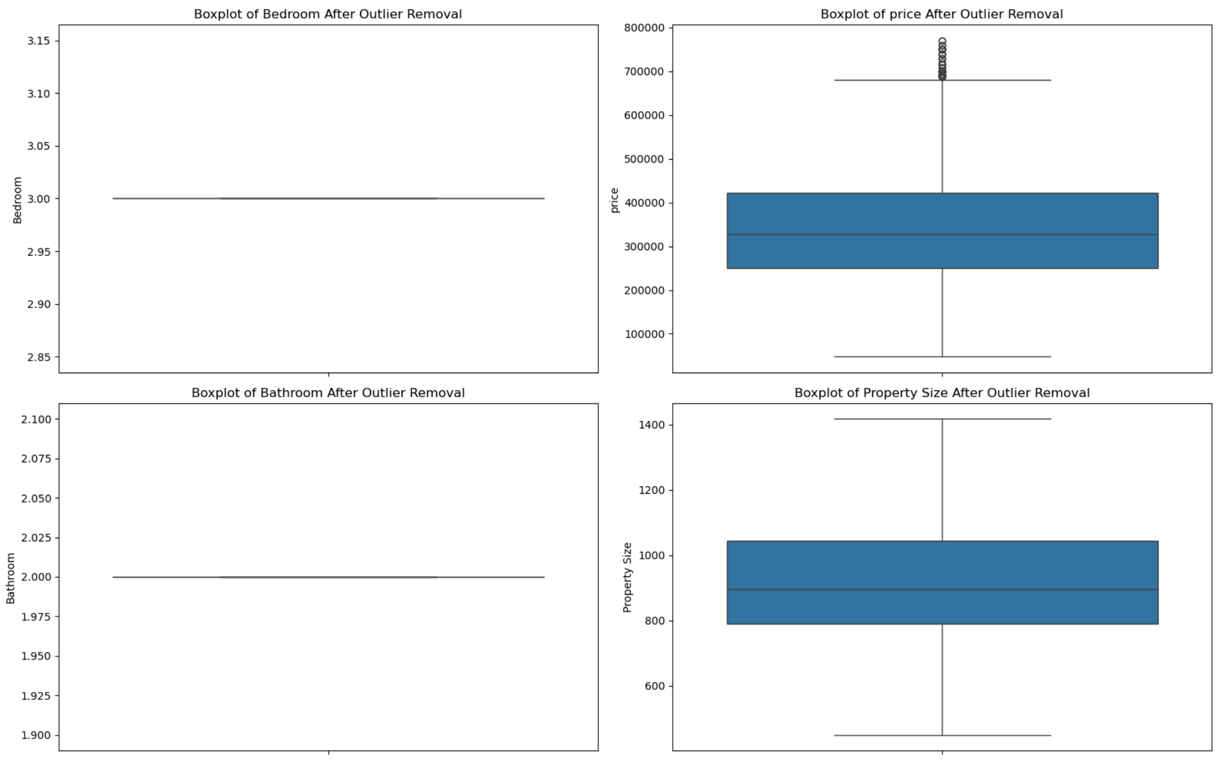


As we saw that, there are quite alot of outliers found in the dataset.Outliers are identified as data points outside the range defined by `Q1 - 1.5 * IQR` and `Q3 + 1.5 * IQR`, where Q1 and Q3 are the first and third quartiles, respectively. The function removes these outliers from each specific group of data.

In [23]:
```
def remove_outliers_by_iqr(df, variables_to_plot):
    Q1 = df[variables_to_plot].quantile(0.25)  # First quartile (25th percentile)
    Q3 = df[variables_to_plot].quantile(0.75)  # Third quartile (75th percentile)
    IQR = Q3 - Q1  # Interquartile Range
    lower_bound = Q1 - 1.5 * IQR  # Lower bound for outliers
    upper_bound = Q3 + 1.5 * IQR  # Upper bound for outliers
    return df[(df[variables_to_plot] >= lower_bound) & (df[variables_to_plot] <= up
```

After that, we visualize again and then we can see that mostly of the outlier will removed.

In [24]:
```
variables_to_clean = ['Bedroom', 'price', 'Bathroom', 'Property Size']

df_cleaned = df.copy()
for var in variables_to_clean:
    df_cleaned = remove_outliers_by_iqr(df_cleaned, var)

plt.figure(figsize=(16, 10))
for i, var in enumerate(variables_to_clean, 1):
    plt.subplot(2, 2, i)
    sns.boxplot(data=df_cleaned, y=var)
    plt.title(f"Boxplot of {var} After Outlier Removal")
    plt.ylabel(var)
    plt.xlabel("")
plt.tight_layout()
plt.show()
```



Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms a dataset with multiple correlated features into a smaller set of **uncorrelated** components called **principal components**. This helps simplify complex datasets while retaining as much variance (information) as possible. PCA helps by reducing dimensionality, which:

- Improves computational efficiency.
- Removes redundancy and noise from the data.
- Enhances visualization by reducing the dataset to 2D or 3D space.
- Helps in feature selection for predictive modeling.

The code below begins by standardizing specific features to ensure they have a mean of 0 and a standard deviation of 1, preventing any single feature from dominating the analysis. Then, PCA is applied without restricting the number of components, allowing all possible principal components to be retained. After transformation, the **explained variance ratio** is calculated to determine how much variance each principal component captures. The **cumulative variance** is also computed to assess the total variance retained when using multiple components.

In [25]:
```
# Add 'price' as part of the numeric data for PCA
numeric_features_with_price = ['Bedroom', 'Bathroom', 'Property Size', 'price']
X_numeric_with_price = df[numeric_features_with_price]
```

```
# Standardize the features
scaler = StandardScaler()
X_numeric_with_price_scaled = scaler.fit_transform(X_numeric_with_price)

# Apply PCA
pca_with_price = PCA(n_components=None)  # Retain all components
X_pca_with_price = pca_with_price.fit_transform(X_numeric_with_price_scaled)

# Calculate explained variance
explained_variance_with_price = pca_with_price.explained_variance_ratio_
cumulative_variance_with_price = explained_variance_with_price.cumsum()

# Determine the number of components for 90%-95% variance
optimal_components_with_price = sum(cumulative_variance_with_price < 0.95) + 1

# Display explained variance data
explained_variance_with_price_data = pd.DataFrame({
    'Component': [f'PC{i+1}' for i in range(len(explained_variance_with_price))],
    'Explained Variance Ratio': explained_variance_with_price,
    'Cumulative Variance': cumulative_variance_with_price
})

print(explained_variance_with_price_data)
print(f"Number of components to retain 90%-95% variance (with price): {optimal_comp
```

```
  Component  Explained Variance Ratio  Cumulative Variance
0       PC1                  0.518801             0.518801
1       PC2                  0.235102             0.753902
2       PC3                  0.165473             0.919376
3       PC4                  0.080624             1.000000
Number of components to retain 90%-95% variance (with price): 4
```

Based on the results of PCA analysis:

- Principal Component 1 (PC1) explains 51.88% of the variance, making it the most significant component in capturing the data's variability.
- Principal Component 2 (PC2) explains an additional 23.51% of the variance, bringing the cumulative variance to 75.39% when combined with PC1.
- Principal Component 3 (PC3) captures 16.55% of the variance, increasing the cumulative variance to 91.94%.
- Principal Component 4 (PC4) explains 8.06%, and its inclusion brings the total variance captured to 100%.

We do again PCA without **price** features.

```
In [26]: # Select numeric features without 'price'
numeric_features = ['Bedroom', 'Bathroom', 'Property Size']
X_numeric = df[numeric_features]
y = df['price']  # Target variable

# Standardize the features
scaler = StandardScaler()
X_numeric_scaled = scaler.fit_transform(X_numeric)
```

```
# Apply PCA
pca = PCA(n_components=None)  # Retain all components
X_pca = pca.fit_transform(X_numeric_scaled)

# Calculate explained variance
explained_variance = pca.explained_variance_ratio_
cumulative_variance = explained_variance.cumsum()

# Determine the number of components for 90%-95% variance
optimal_components = sum(cumulative_variance < 0.95) + 1

# Display explained variance data
explained_variance_data = pd.DataFrame({
    'Component': [f'PC{i+1}' for i in range(len(explained_variance))],
    'Explained Variance Ratio': explained_variance,
    'Cumulative Variance': cumulative_variance
})

print(explained_variance_data)
print(f"Number of components to retain 90%-95% variance (excluding price): {optimal
```

```
  Component  Explained Variance Ratio  Cumulative Variance
0       PC1                  0.555058             0.555058
1       PC2                  0.313464             0.868523
2       PC3                  0.131477             1.000000
Number of components to retain 90%-95% variance (excluding price): 3
```

Based on the results of PCA analysis excluding **'price'**:

- Principal Component 1 (PC1) explains 55.51% of the variance, capturing the majority of the variability in the dataset.
- Principal Component 2 (PC2) explains an additional 31.35% of the variance, bringing the cumulative variance to 86.85% when combined with PC1.
- Principal Component 3 (PC3) captures 13.15% of the variance, increasing the cumulative variance to 100%.

Key Insights:

- The reduction in the number of components (from 4 to 3) highlights that excluding price simplifies the dataset.
- 2nd approach is better suited because price will be used as the target variable.

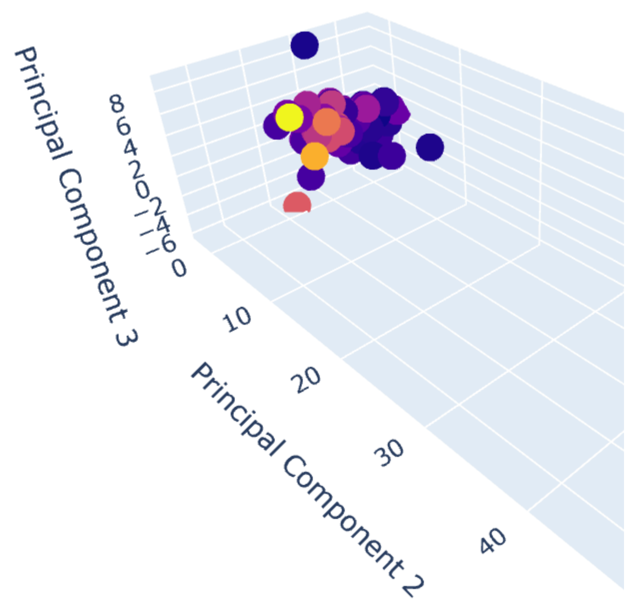
Below code is visualization of the PCA exluding **price** column

In [27]:
```
#pip install plotly
```

In [28]:
```
import plotly.express as px

X_pca_3d = pca.fit_transform(X_numeric_scaled)

# Create a DataFrame for the 3D PCA components
pca_3d_df = pd.DataFrame(X_pca_3d, columns=["PC1", "PC2", "PC3"])
```

```
pca_3d_df['Price'] = y  # Add price as the color scale

# Create an interactive 3D scatter plot
fig = px.scatter_3d(
    pca_3d_df,
    x="PC1",
    y="PC2",
    z="PC3",
    color="Price",
    title="Interactive 3D Visualization of PCA Components",
    labels={"PC1": "Principal Component 1", "PC2": "Principal Component 2", "PC3":
)

# Show the plot
fig.show()
```

Interactive 3D Visualization of PCA Components



Machine Learning Model

In this section, 4 machine learning models will be used to do the prediction which is

- **Linear Regression (LR)**

- **Random Forest (RF)**
- **Gradient Boosting**
- **Support Vector Regressor (SVR)**

Model Training

Each model was trained using **Principal Component Analysis (PCA)-transformed features** (`PC1` , `PC2` , `PC3`) to ensure dimensionality reduction.

Steps in Model Training

1. The dataset was split into **80% training data** and **20% test data** using `train_test_split()` .
2. The four models were initialized with default parameters for baseline comparison.
3. Each model was trained using the training dataset.
4. The trained models were used to make predictions on the test dataset.
5. The performance of each model was assessed using:
   - **Mean Squared Error (MSE)** – Measures average squared difference between actual and predicted prices.
   - **Mean Absolute Error (MAE)** – Measures the average absolute difference.
   - **R² Score** – Determines how well the model explains variance in price.

In [29]:
```
# Select features excluding 'price'
numeric_features = ['Bedroom', 'Bathroom', 'Property Size']
X_numeric = df[numeric_features]
y = df['price']  # Target variable

# Standardize the features
scaler = StandardScaler()
X_numeric_scaled = scaler.fit_transform(X_numeric)

# Apply PCA with 3 components
pca = PCA(n_components=3)
X_pca = pca.fit_transform(X_numeric_scaled)

# Create a DataFrame for PCA-transformed features
pca_df = pd.DataFrame(X_pca, columns=["PC1", "PC2", "PC3"])
pca_df['price'] = y  # Add the target variable back for reference

# Use PCA-transformed features (PC1, PC2, PC3) for training
X_pca_features = pca_df[["PC1", "PC2", "PC3"]]
y_target = pca_df["price"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_pca_features, y_target, test_

# Initialize models
models = [
    ('Linear Regression', LinearRegression()),
    ('Random Forest', RandomForestRegressor(random_state=42, n_estimators=100)),
    ('Gradient Boosting', GradientBoostingRegressor(random_state=42, n_estimators=1
    ('SVR', SVR(kernel='rbf', C=100, gamma=0.1))
]

# Train and evaluate models
results = []
# Train and evaluate each model
for name, model in models:
    # Train the model
    model.fit(X_train, y_train)
    # Predict on the test set
    y_pred = model.predict(X_test)
    # Evaluate the model
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    # Store results (including the model)
    results.append((name, model, mse, r2))
    # Print performance
    print(f"Model: {name}")
    print(f"  Mean Squared Error (MSE): {mse:.2f}")
    print(f"  Mean Absolute Error (MAE): {mae:.2f}")
    print(f"  R-squared (R²): {r2:.2f}")
    print()
```

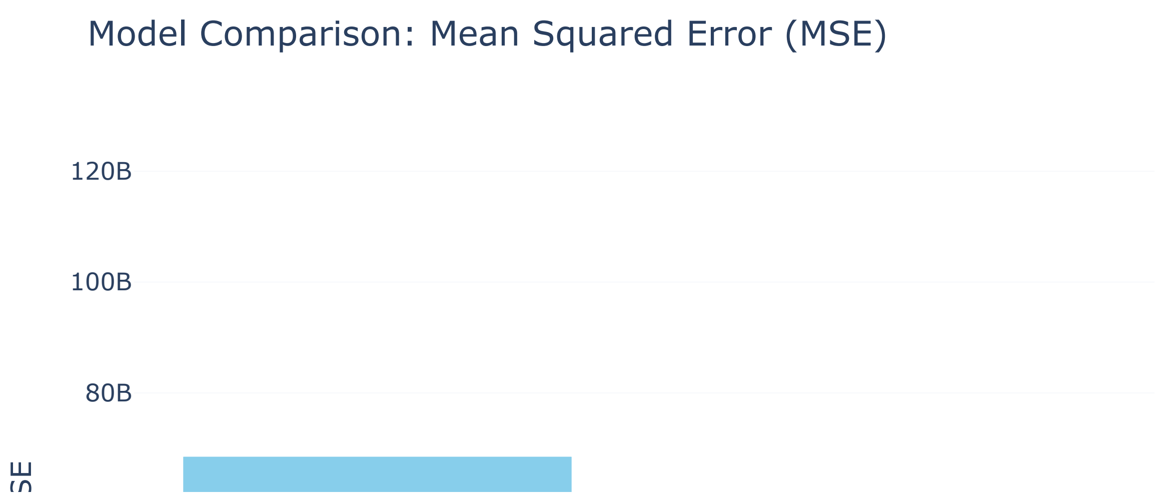
```
Model: Linear Regression
  Mean Squared Error (MSE): 68583773508.19
  Mean Absolute Error (MAE): 164187.71
  R-squared (R²): 0.40

Model: Random Forest
  Mean Squared Error (MSE): 41040550284.83
  Mean Absolute Error (MAE): 113050.20
  R-squared (R²): 0.64

Model: Gradient Boosting
  Mean Squared Error (MSE): 40253430014.55
  Mean Absolute Error (MAE): 117479.07
  R-squared (R²): 0.65

Model: SVR
  Mean Squared Error (MSE): 118386827552.48
  Mean Absolute Error (MAE): 184839.68
  R-squared (R²): -0.03
```

Below are visualization the results by graph.

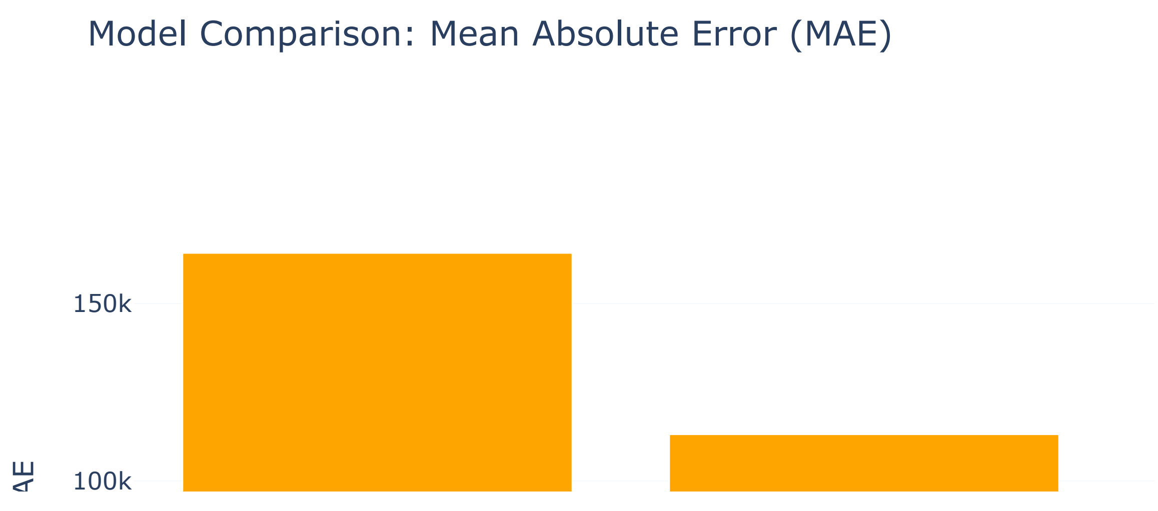
In [30]:
```
# Extract performance metrics from `results` list
model_names = [result[0] for result in results]
mse_values = [result[2] for result in results]
mae_values = [mean_absolute_error(y_test, result[1].predict(X_test)) for result in
r2_values = [result[3] for result in results]

# Create a DataFrame for MSE, MAE, and R² values
metrics_df = pd.DataFrame({
    "Model": model_names,
    "MSE": mse_values,
    "MAE": mae_values,
    "R² Score": r2_values
})
```

In [31]:

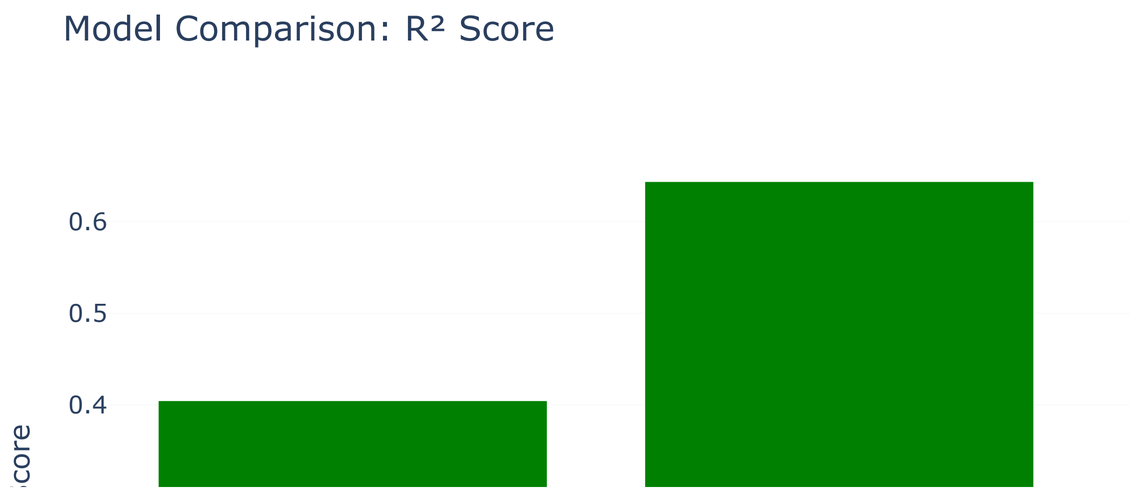
```
import plotly.graph_objects as go

# Create the MSE bar chart
fig_mse = go.Figure()

# Add MSE data
fig_mse.add_trace(go.Bar(
    x=metrics_df["Model"],
    y=metrics_df["MSE"],
    name="MSE",
    marker_color="skyblue"
))

# Update layout for MSE plot
fig_mse.update_layout(
    title="Model Comparison: Mean Squared Error (MSE)",
    xaxis_title="Model",
    yaxis_title="MSE",
    template="plotly_white"
)
fig_mse.show()
```



In [32]:
```
# Create the MAE bar chart
fig_mae = go.Figure()

# Add MAE data
fig_mae.add_trace(go.Bar(
    x=metrics_df["Model"],
    y=metrics_df["MAE"],
    name="MAE",
    marker_color="orange"
))

# Update layout for MAE plot
fig_mae.update_layout(
    title="Model Comparison: Mean Absolute Error (MAE)",
    xaxis_title="Model",
    yaxis_title="MAE",
    template="plotly_white"
)
fig_mae.show()
```



```
In [33]: # Create the R² Score bar chart
fig_r2 = go.Figure()

# Add R² Score data
fig_r2.add_trace(go.Bar(
    x=metrics_df["Model"],
    y=metrics_df["R² Score"],
    name="R² Score",
    marker_color="green"
))

# Update layout for R² Score plot
fig_r2.update_layout(
    title="Model Comparison: R² Score",
    xaxis_title="Model",
    yaxis_title="R² Score",
    template="plotly_white"
)
fig_r2.show()
```



After implement model training and visualizations of performance metrices, we found that:

1. **Gradient Boosting is the Best Model**
   - **R^2 Score = 0.65** (Highest among all models).
   - **MSE = 40,254,340,014.55, MAE = 117,479.07** (Lower error compared to other models).

Gradient Boosting shows the best balance between accuracy and error reduction, making it the most reliable model for predicting housing prices.

2. **Random Forest Also Performs Well**
   - **R^2 Score = 0.64**, slightly lower than Gradient Boosting.
   - **MSE = 41,040,550,284.83, MAE = 113,050.20** (similar performance to Gradient Boosting).

Random Forest performs well but slightly lower than Gradient Boosting.

4. **Linear Regression Struggles with Non-Linear Relationships**
   - **R^2 Score = 0.40**
   - High **MSE (68,583,773,508.19)** and **MAE (164,187.71)** indicate poor accuracy.

This suggests that the relationship between features and house prices is not purely linear, making it unsuitable for this dataset.

4. **Support Vector Regression (SVR) Performs the Worst**
   - **R² Score = -0.03**
   - **MSE (118,386,827,552.48)** and **MAE (184,839.68)** are extremely high.

Indicates that SVR does not generalize well to this dataset, possibly due to improper scaling or kernel selection.

Improve Model and Analyze Results

As we noticed that the results of **random forest** and **gradient boosting** are **close**, we can do some model improvement to finalize which one is the best model.

About this, we will do **Hyperparameter Tuning** and **Feature Importance** to improve this 2 models and do comparison.

Hyperparameter Tuning

To further improve model performance, Random Forest and Gradient Boosting were fine-tuned using **GridSearchCV** which helps to find the best combination of parameters that maximize model performance.

Random Forest Hyperparameter Tuning

In [34]:
```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid_rf = {
    "n_estimators": [100, 200, 300],
    "max_depth": [None, 10, 20],
    "min_samples_split": [2, 5, 10],
    "min_samples_leaf": [1, 2, 4]
}

# Initialize GridSearchCV
grid_search_rf = GridSearchCV(
    RandomForestRegressor(random_state=42),
    param_grid=param_grid_rf,
    cv=3,
    scoring="r2",
    n_jobs=-1
)

# Fit the model
grid_search_rf.fit(X_train, y_train)

# Print the best parameters
```

```
print("Best parameters for Random Forest:", grid_search_rf.best_params_)

# Re-train the model with the best parameters
best_rf = grid_search_rf.best_estimator_
```

```
Best parameters for Random Forest: {'max_depth': 10, 'min_samples_leaf': 4, 'min_sam
ples_split': 10, 'n_estimators': 100}
```

Impact on Performance:

- Reduces overfitting by controlling tree depth.
- Increases model generalization, improving test accuracy.

Gradient Boosting Hyperparameter Tuning

In [35]:
```
param_grid_gb = {
    "n_estimators": [100, 200, 300],
    "learning_rate": [0.01, 0.1, 0.2],
    "max_depth": [3, 5, 7]
}

grid_search_gb = GridSearchCV(
    GradientBoostingRegressor(random_state=42),
    param_grid=param_grid_gb,
    cv=3,
    scoring="r2",
    n_jobs=-1
)

grid_search_gb.fit(X_train, y_train)

print("Best parameters for Gradient Boosting:", grid_search_gb.best_params_)

best_gb = grid_search_gb.best_estimator_
```

```
Best parameters for Gradient Boosting: {'learning_rate': 0.1, 'max_depth': 3, 'n_est
imators': 100}
```

Impact on Performance:

- Improves predictive accuracy while avoiding extreme bias or variance.
- Fine-tunes learning speed so that the model does not overreact to small variations.

After tuning, we will get the new **MSE**, **MAE** and **R²** results.

In [36]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Evaluate the tuned Random Forest model
y_pred_rf = best_rf.predict(X_test)
mse_rf = mean_squared_error(y_test, y_pred_rf)
mae_rf = mean_absolute_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

# Evaluate the tuned Gradient Boosting model
y_pred_gb = best_gb.predict(X_test)
```

```
mse_gb = mean_squared_error(y_test, y_pred_gb)
mae_gb = mean_absolute_error(y_test, y_pred_gb)
r2_gb = r2_score(y_test, y_pred_gb)

# Store results for comparison
tuned_results = {
    "Model": ["Random Forest (Tuned)", "Gradient Boosting (Tuned)"],
    "MSE": [mse_rf, mse_gb],
    "MAE": [mae_rf, mae_gb],
    "R²": [r2_rf, r2_gb]
}

results_df = pd.DataFrame(tuned_results)
print(results_df)
```

```
                       Model           MSE            MAE        R²
0      Random Forest (Tuned)  4.270960e+10  116685.133663  0.629342
1  Gradient Boosting (Tuned)  4.025343e+10  117479.065399  0.650658
```

Compare Tuned vs. Original Model

After we done hyperparameter tuning, we will compare the results with original results.

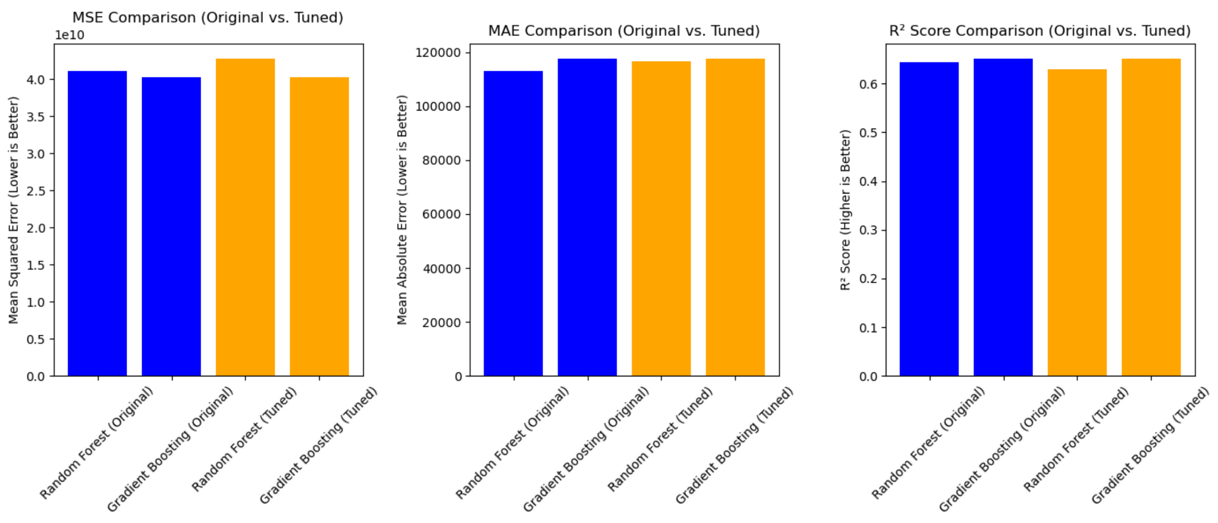
In [37]:
```
# Evaluate tuned models
y_pred_rf_tuned = best_rf.predict(X_test)
y_pred_gb_tuned = best_gb.predict(X_test)

# Store original model evaluation metrics
mse_rf_original = mean_squared_error(y_test, models[1][1].predict(X_test))  # Rando
mae_rf_original = mean_absolute_error(y_test, models[1][1].predict(X_test))
r2_rf_original = r2_score(y_test, models[1][1].predict(X_test))

mse_gb_original = mean_squared_error(y_test, models[2][1].predict(X_test))  # Gradi
mae_gb_original = mean_absolute_error(y_test, models[2][1].predict(X_test))
r2_gb_original = r2_score(y_test, models[2][1].predict(X_test))

# Compute evaluation metrics
mse_rf_tuned = mean_squared_error(y_test, y_pred_rf_tuned)
mae_rf_tuned = mean_absolute_error(y_test, y_pred_rf_tuned)
r2_rf_tuned = r2_score(y_test, y_pred_rf_tuned)

mse_gb_tuned = mean_squared_error(y_test, y_pred_gb_tuned)
mae_gb_tuned = mean_absolute_error(y_test, y_pred_gb_tuned)
r2_gb_tuned = r2_score(y_test, y_pred_gb_tuned)

# Store results
original_results = {
    "Model": ["Random Forest (Original)", "Gradient Boosting (Original)"],
    "MSE": [mse_rf_original, mse_gb_original],
    "MAE": [mae_rf_original, mae_gb_original],
    "R²": [r2_rf_original, r2_gb_original]
}
tuned_results = {
    "Model": ["Random Forest (Tuned)", "Gradient Boosting (Tuned)"],
    "MSE": [mse_rf_tuned, mse_gb_tuned],
    "MAE": [mae_rf_tuned, mae_gb_tuned],
    "R²": [r2_rf_tuned, r2_gb_tuned]
}

original_df = pd.DataFrame(original_results)
tuned_df = pd.DataFrame(tuned_results)

comparison_df = pd.concat([original_df, tuned_df], ignore_index=True)
comparison_df
```

Out[37]:

	Model	MSE	MAE	R²
0	Random Forest (Original)	4.104055e+10	113050.196239	0.643827
1	Gradient Boosting (Original)	4.025343e+10	117479.065399	0.650658
2	Random Forest (Tuned)	4.270960e+10	116685.133663	0.629342
3	Gradient Boosting (Tuned)	4.025343e+10	117479.065399	0.650658

We plot graph for clearly visualize the results.

In [38]:
```
plt.figure(figsize=(14, 6))

# MSE Comparison
plt.subplot(1, 3, 1)
plt.bar(comparison_df["Model"], comparison_df["MSE"], color=["blue", "blue", "orang
plt.title("MSE Comparison (Original vs. Tuned)")
plt.ylabel("Mean Squared Error (Lower is Better)")
plt.xticks(rotation=45)

# MAE Comparison
plt.subplot(1, 3, 2)
plt.bar(comparison_df["Model"], comparison_df["MAE"], color=["blue", "blue", "orang
plt.title("MAE Comparison (Original vs. Tuned)")
plt.ylabel("Mean Absolute Error (Lower is Better)")
plt.xticks(rotation=45)

# R² Score Comparison
plt.subplot(1, 3, 3)
plt.bar(comparison_df["Model"], comparison_df["R²"], color=["blue", "blue", "orange
plt.title("R² Score Comparison (Original vs. Tuned)")
plt.ylabel("R² Score (Higher is Better)")
plt.xticks(rotation=45)

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



After comparison, we noticed that:

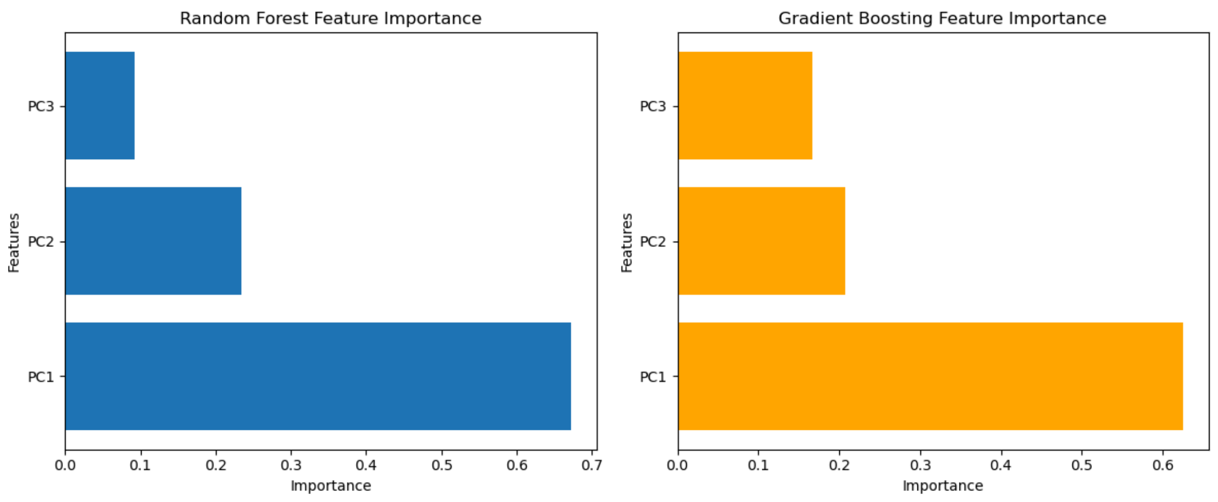
- Gradient Boosting results are remains the same.
- Random Forest reults are getting worse which the MSE and MAE results increased, R^2 results are slightly decreased.

Indicating that model before performing hyperparameter tuning is better than after performing hyperparameter tuning.

Feature Importance

Feature importance helps in understanding which PCA-transformed features (PC1, PC2, PC3) contribute the most to predicting housing prices.

In [39]:
```
# Get feature importance scores
rf_importance = best_rf.feature_importances_
gb_importance = best_gb.feature_importances_

# Visualize Feature Importance
features = ["PC1", "PC2", "PC3"]

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.barh(features, rf_importance)
plt.xlabel("Importance")
plt.ylabel("Features")
plt.title("Random Forest Feature Importance")

plt.subplot(1, 2, 2)
plt.barh(features, gb_importance, color="orange")
plt.xlabel("Importance")
plt.ylabel("Features")
plt.title("Gradient Boosting Feature Importance")
plt.tight_layout()
plt.show()
```



Results:

1. PC1 is the Most Important Feature
   - In both models, PC1 has the highest importance, around 0.6 to 0.7.

This suggests that PC1 captures the most variance in the dataset and is the strongest predictor of housing prices.

2. PC2 Contributes Moderately
   - PC2 has moderate importance, around 0.2 to 0.3 in both models.

While it plays a role in predicting housing prices, it is less significant than PC1.

3. PC3 Has the Lowest Importance
   - PC3 contributes the least, especially in Random Forest (importance < 0.1).

This suggests that PC3 captures very little useful variance for price prediction.

Gradient Boosting assigns slightly more importance to PC3 than Random Forest, indicating it may still hold some minor predictive value.

Final Comparison and Best Model Predictions

Last step of this project is to do final comparison between random forest and gradient boosting to analyze the best model predictions.

In [40]:
```
# Store final evaluation metrics for tuned models
final_results = {
    "Model": ["Random Forest (Tuned)", "Gradient Boosting (Tuned)"],
    "MSE": [mse_rf_tuned, mse_gb_tuned],
    "MAE": [mae_rf_tuned, mae_gb_tuned],
    "R²": [r2_rf_tuned, r2_gb_tuned]
}

final_df = pd.DataFrame(final_results)
final_df
```

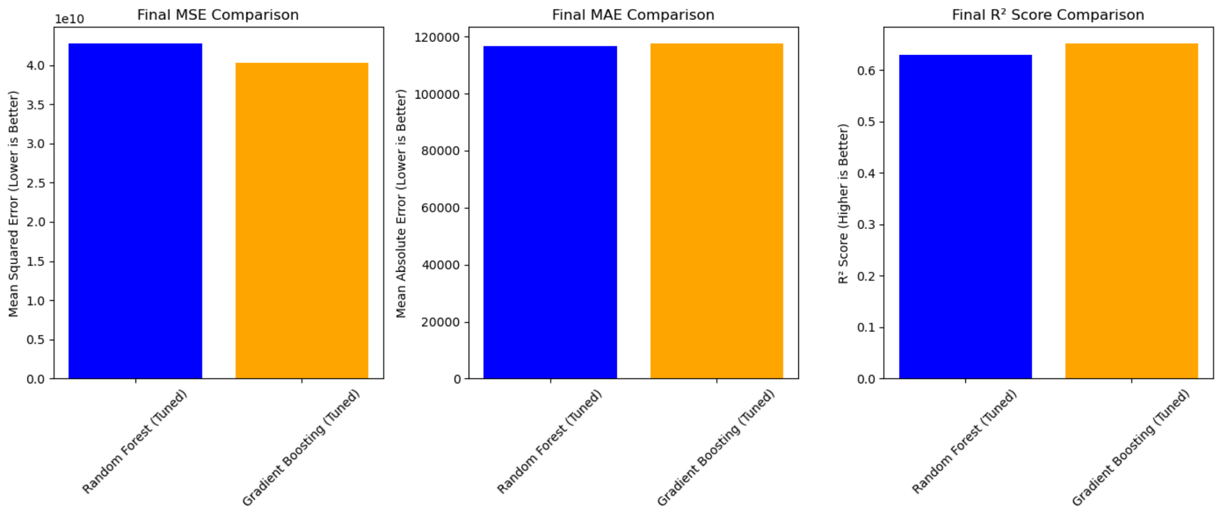
Out[40]:

	Model	MSE	MAE	R²
0	Random Forest (Tuned)	4.270960e+10	116685.133663	0.629342
1	Gradient Boosting (Tuned)	4.025343e+10	117479.065399	0.650658

Visualize the results.

In [41]:

```
# Plot Final Model Comparison
plt.figure(figsize=(14, 6))

# MSE Comparison
plt.subplot(1, 3, 1)
plt.bar(final_df["Model"], final_df["MSE"], color=["blue", "orange"])
plt.title("Final MSE Comparison")
plt.ylabel("Mean Squared Error (Lower is Better)")
plt.xticks(rotation=45)

# MAE Comparison
plt.subplot(1, 3, 2)
plt.bar(final_df["Model"], final_df["MAE"], color=["blue", "orange"])
plt.title("Final MAE Comparison")
plt.ylabel("Mean Absolute Error (Lower is Better)")
plt.xticks(rotation=45)

# R² Score Comparison
plt.subplot(1, 3, 3)
plt.bar(final_df["Model"], final_df["R²"], color=["blue", "orange"])
plt.title("Final R² Score Comparison")
plt.ylabel("R² Score (Higher is Better)")
plt.xticks(rotation=45)

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



Based on the final results, we can confirm that **Gradient Boosting** is the **best model** to use for house pricing prediction with getting resuls of **lowest MSE and MAE results** and **highest R^2 score**. Then we will save the model for doing price predictions.

In [42]:
```
import joblib

# Gradient Boosting was the best model
best_model = best_gb

# Save the model
joblib.dump(best_model, "best_model.pkl")
print("Best model saved successfully!")
```

```
Best model saved successfully!
```

Using the trained Gradient Boosting model, we made predictions on unseen test data.

In [43]:
```
# Use the test dataset to make predictions
y_pred_gb_final = best_gb.predict(X_test)

# Create a DataFrame to compare actual vs. predicted prices
predictions_df = pd.DataFrame({
    "Actual Price": y_test.values,
    "Predicted Price": y_pred_gb_final
})

# Display the first few predictions
predictions_df.head()
```

Out[43]:

	Actual Price	Predicted Price
0	265000.0	267650.661895
1	518000.0	369657.846869
2	360000.0	267650.661895
3	360000.0	423894.588556
4	338800.0	306410.681450

1st row is the house actual price and 2nd row is the predicted price. Result will be saved in a csv files named `predicted_prices.csv`.

In [44]:
```
predictions_df.to_csv("predicted_prices.csv", index=False)
```

Conclusion

In this study, we explored **housing affordability and market trends in Malaysia** using **machine learning techniques**. By analyzing real-world housing data, we gained valuable insights into factors influencing house prices and affordability across different regions. Our analysis revealed that key features such as **bedroom count, bathroom count and property size** significantly impact housing prices. Using **Principal Component Analysis (PCA)**, we successfully reduced dimensionality while preserving essential information, improving model efficiency and interpretability.Various machine learning models were employed for predictive analysis, with evaluation metrics such as RMSE and R^2 used to assess performance. The best-performing model demonstrated strong predictive capabilities, suggesting its potential for real-world applications in market forecasting and investment decision-making.

Based on our findings, we recommend gathering more data to ensure the model generalizes well across different property types and locations, enhance predictive accuracy. Exploring advanced models like XGBoost, CatBoost, or Neural Networks for potential performance gains. The insights from this study could support policymakers in formulating data-driven strategies for housing affordability and market regulation.

In conclusion, this project demonstrates the effectiveness of machine learning in analyzing and predicting housing market trends, offering valuable contributions to both research and practical applications.

References

1. Satish, G. N., Raghavendran, C. V., Rao, M. S., & Srinivasulu, C. (2019). House price prediction using machine learning. Journal of Innovative Technology and Exploring Engineering, 8(9), 717-722.
    - Link Here
2. Truong, Q., Nguyen, M., Dang, H., & Mei, B. (2020). Housing price prediction via improved machine learning techniques. Procedia Computer Science, 174, 433-442.
    - Link Here

END