In [2]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor, BaggingRegressor, GradientBoost
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns
```

In [4]:
```
df = pd.read_csv("Housing.csv")
df.head()
```

Out[4]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotw
0	13300000	7420	4	2	3	yes	no	no	
1	12250000	8960	4	4	4	yes	no	no	
2	12250000	9960	3	2	2	yes	no	yes	
3	12215000	7500	4	2	2	yes	no	yes	
4	11410000	7420	4	1	2	yes	yes	yes	

Categorical Variables

In [6]:
```
df_encoded = df.copy()
label_encoders = {}

for column in df_encoded.select_dtypes(include='object').columns:
    le = LabelEncoder()
    df_encoded[column] = le.fit_transform(df_encoded[column])
    label_encoders[column] = le
```

Define Target

In [9]:
```
X = df_encoded.drop("price", axis=1)
y = df_encoded["price"]
```

Train Test

This function trains a model, predicts on test data, and calculates:

- **RMSE (Root Mean Squared Error):** How far predictions are from actual values.
- **R^2 Score:** Proportion of variance explained by the model (closer to 1 is better).

In [15]:
```
def evaluate_model(name, model, X_train, X_test, y_train, y_test):
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, preds))
    r2 = r2_score(y_test, preds)
    print(f"{name}:")
    print(f"  ➤ RMSE: {rmse:.2f}")
    print(f"  ➤ R² Score: {r2:.4f}")
    print("-" * 40)
    return (name, rmse, r2)
```

Linear Regression

Applies simple linear regression assuming a straight-line relationship between features and price. Serves as a baseline model for comparison.

In [17]:
```
lr = LinearRegression()
evaluate_model("Linear Regression", lr, X_train, X_test, y_train, y_test)
```

```
Linear Regression:
  ➤ RMSE: 1331071.42
  ➤ R² Score: 0.6495
----------------------------------------
```

Out[17]:
```
('Linear Regression', 1331071.4167895103, 0.6494754192267804)
```

Linear regression assumes a straight-line (linear) relationship between the target (house price) and input features (like area, number of bedrooms, etc.). It tries to fit a line that minimizes the sum of squared differences between actual and predicted values.Decent performance as a baseline. Explains about 65% of price variation. However, it cannot capture complex nonlinear patterns.

linear regression falls short.

In [20]:
```
poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)
poly_lr = LinearRegression()
evaluate_model("Polynomial Regression (deg=2)", poly_lr, X_train_poly, X_test_poly,
```

```
Polynomial Regression (deg=2):
  ➤ RMSE: 1330287.24
  ➤ R² Score: 0.6499
----------------------------------------
```

Out[20]:
```
('Polynomial Regression (deg=2)', 1330287.2411168437, 0.6498883074871176)
```

This extends linear regression by adding polynomial features (squared, interaction terms) to model non-linear relationships. For example, it considers area², area × bathrooms, etc.Slightly better than simple linear regression but not significantly. May help in datasets where relationships aren't purely linear, though can overfit if not tuned properly.

In [ ]:
```
```

Bagging Regressor

A bagging ensemble method that reduces variance by training multiple models on random subsets and averaging their predictions. It increases stability and reduces overfitting.

In [23]:
```
bagging = BaggingRegressor(random_state=42)
evaluate_model("Bagging Regressor", bagging, X_train, X_test, y_train, y_test)
```

```
Bagging Regressor:
  ➤ RMSE: 1369828.03
  ➤ R² Score: 0.6288
----------------------------------------
```

Out[23]:
```
('Bagging Regressor', 1369828.0260541113, 0.6287659030330632)
```

Bagging (Bootstrap Aggregating) trains multiple decision trees on random subsets of the data and averages their predictions. This reduces overfitting and increases stability.

Random Forest

An ensemble of decision trees trained on bootstrapped samples with feature randomness. It captures complex interactions but may need tuning to avoid overfitting or underfitting.

In [26]:
```
rf = RandomForestRegressor(random_state=42)
evaluate_model("Random Forest Regressor", rf, X_train, X_test, y_train, y_test)
```

```
Random Forest Regressor:
  ➤ RMSE: 1401263.08
  ➤ R² Score: 0.6115
----------------------------------------
```

Out[26]: ('Random Forest Regressor', 1401263.0789821919, 0.6115321143409216)

An advanced bagging technique using decision trees. Adds feature randomness when splitting nodes, which helps in decorrelating trees and improving predictions. Surprisingly underperformed compared to others. May be due to insufficient depth or not enough hyperparameter tuning. Could be improved with optimization.

In [ ]:
```
```

Gradient Boosting

An advanced ensemble that sequentially improves the model by correcting previous errors. Best suited for structured datasets with complex relationships.

In [29]:
```
gb = GradientBoostingRegressor(random_state=42)
evaluate_model("Gradient Boosting Regressor", gb, X_train, X_test, y_train, y_test)
```

```
Gradient Boosting Regressor:
  ➤ RMSE: 1301871.87
  ➤ R² Score: 0.6647
----------------------------------------
```

Out[29]: ('Gradient Boosting Regressor', 1301871.871671099, 0.6646855642239725)

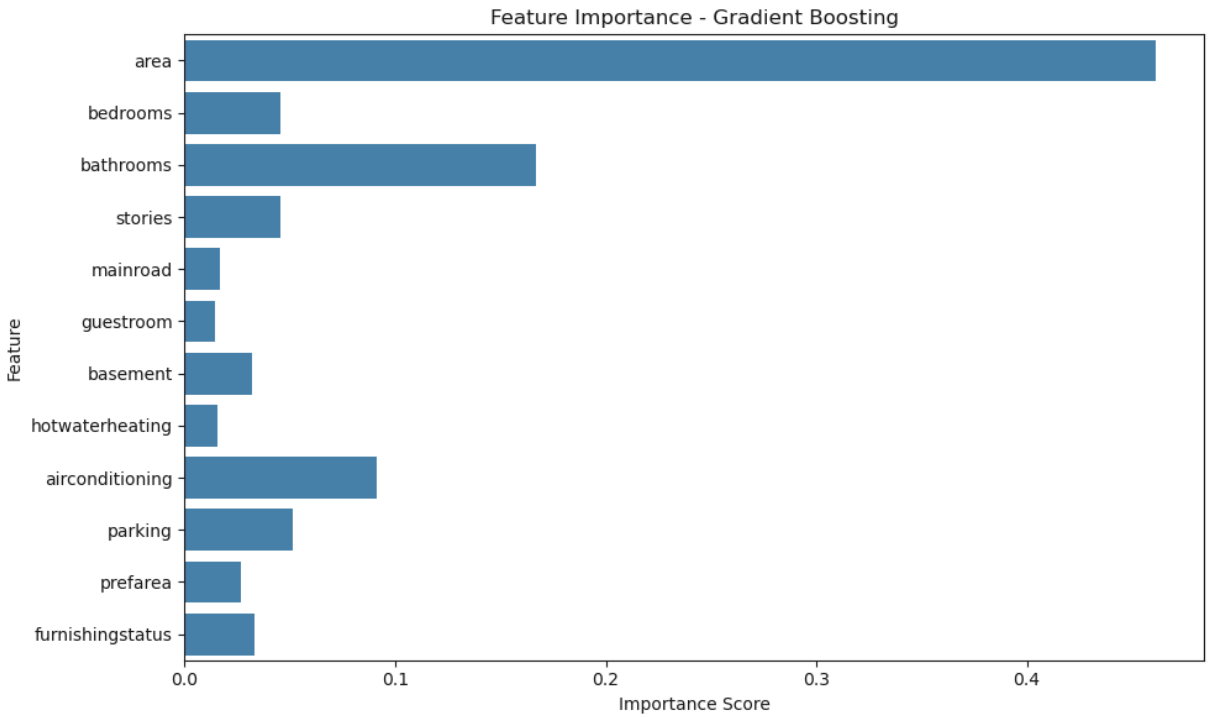
optimization.

In [ ]:

The bar chart displays the **feature importance scores** as determined by the **Gradient Boosting Regressor**, which was the best-performing model.

What it shows:

In [32]:
```
plt.figure(figsize=(10,6))
importances = gb.feature_importances_
feat_names = X.columns
sns.barplot(x=importances, y=feat_names)
plt.title("Feature Importance - Gradient Boosting")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.tight_layout()
plt.show()
```



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: