In [1]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEnco
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LinearRegression,Ridge,Lasso,RidgeCV
from sklearn.metrics import mean_squared_error, mean_squared_log_error
import seaborn as sns # Import seaborn for better visualizations
import math
from scipy import stats
from sklearn.ensemble import RandomForestRegressor,BaggingRegressor,Gradient
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

In [2]:
```
from utils import line_plot_viz, box_plot_viz, heat_map_viz, hist_plot_viz,
import warnings
warnings.filterwarnings('ignore')
```

Context This classic dataset contains the prices and other attributes of almost 54,000 diamonds. It's a great dataset for beginners learning to work with data analysis and visualization.

Content price price in US dollars ($326 - -$18,823)

carat weight of the diamond (0.2--5.01)

cut quality of the cut (Fair, Good, Very Good, Premium, Ideal)

color diamond colour, from J (worst) to D (best)

clarity a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))

x length in mm (0--10.74)

y width in mm (0--58.9)

z depth in mm (0--31.8)

depth total depth percentage = z / mean(x, y) = 2 * z / (x + y) (43--79)

table width of top of diamond relative to widest point (43--95)

In [3]:
```
og_data = pd.read_csv('./data/Diamonds/diamonds.csv')
data = og_data.copy()
```

In [4]:
```
data.describe()
```

Out[4]:

	index	carat	depth	table	price	
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53
mean	26970.500000	0.797940	61.749405	57.457184	3932.799722	
std	15571.281097	0.474011	1.432621	2.234491	3989.439738	
min	1.000000	0.200000	43.000000	43.000000	326.000000	
25%	13485.750000	0.400000	61.000000	56.000000	950.000000	
50%	26970.500000	0.700000	61.800000	57.000000	2401.000000	
75%	40455.250000	1.040000	62.500000	59.000000	5324.250000	
max	53940.000000	5.010000	79.000000	95.000000	18823.000000	

In [5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
Data columns (total 11 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   index    53940 non-null  int64  
 1   carat    53940 non-null  float64
 2   cut      53940 non-null  object 
 3   color    53940 non-null  object 
 4   clarity  53940 non-null  object 
 5   depth    53940 non-null  float64
 6   table    53940 non-null  float64
 7   price    53940 non-null  int64  
 8   x        53940 non-null  float64
 9   y        53940 non-null  float64
 10  z        53940 non-null  float64
dtypes: float64(6), int64(2), object(3)
memory usage: 4.5+ MB
```

In [6]:
```
data.columns
```

Out[6]:
```
Index(['index', 'carat', 'cut', 'color', 'clarity', 'depth', 'table', 'pric
e',
       'x', 'y', 'z'],
      dtype='object')
```

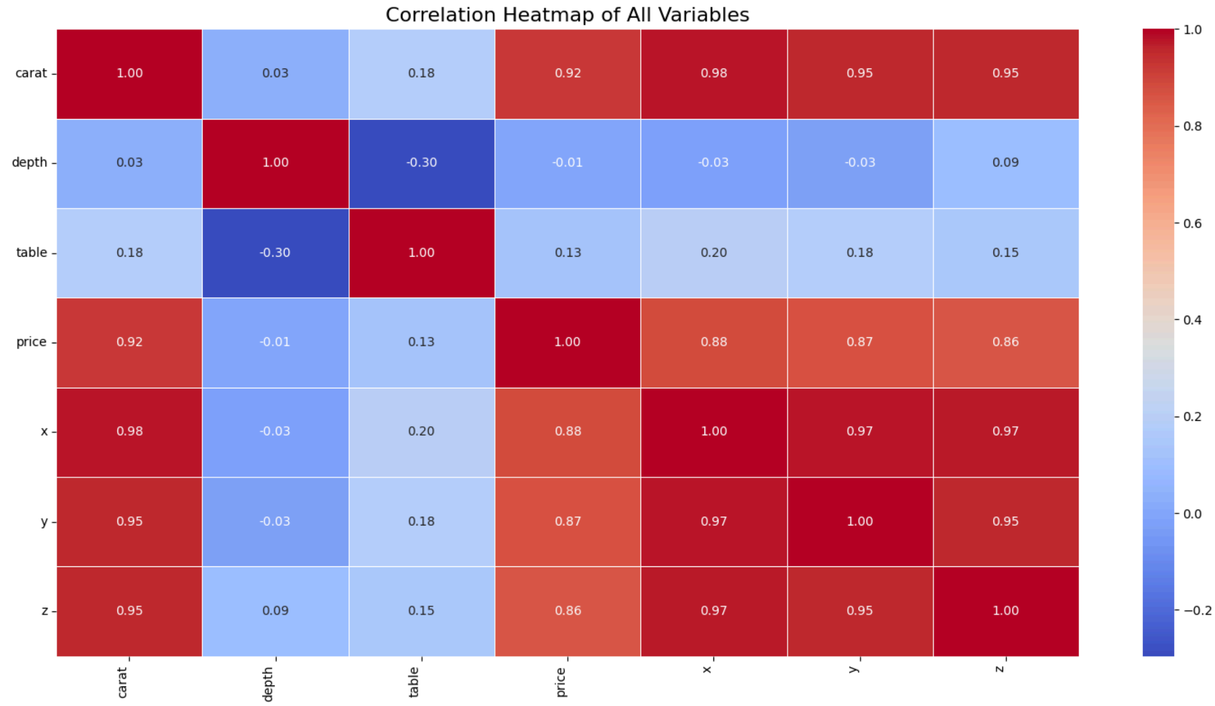
Remove columns

In [7]:
```
# Drop index column
data = data.drop('index', axis=1)
```

In [8]:
```
data.dtypes[0]
```

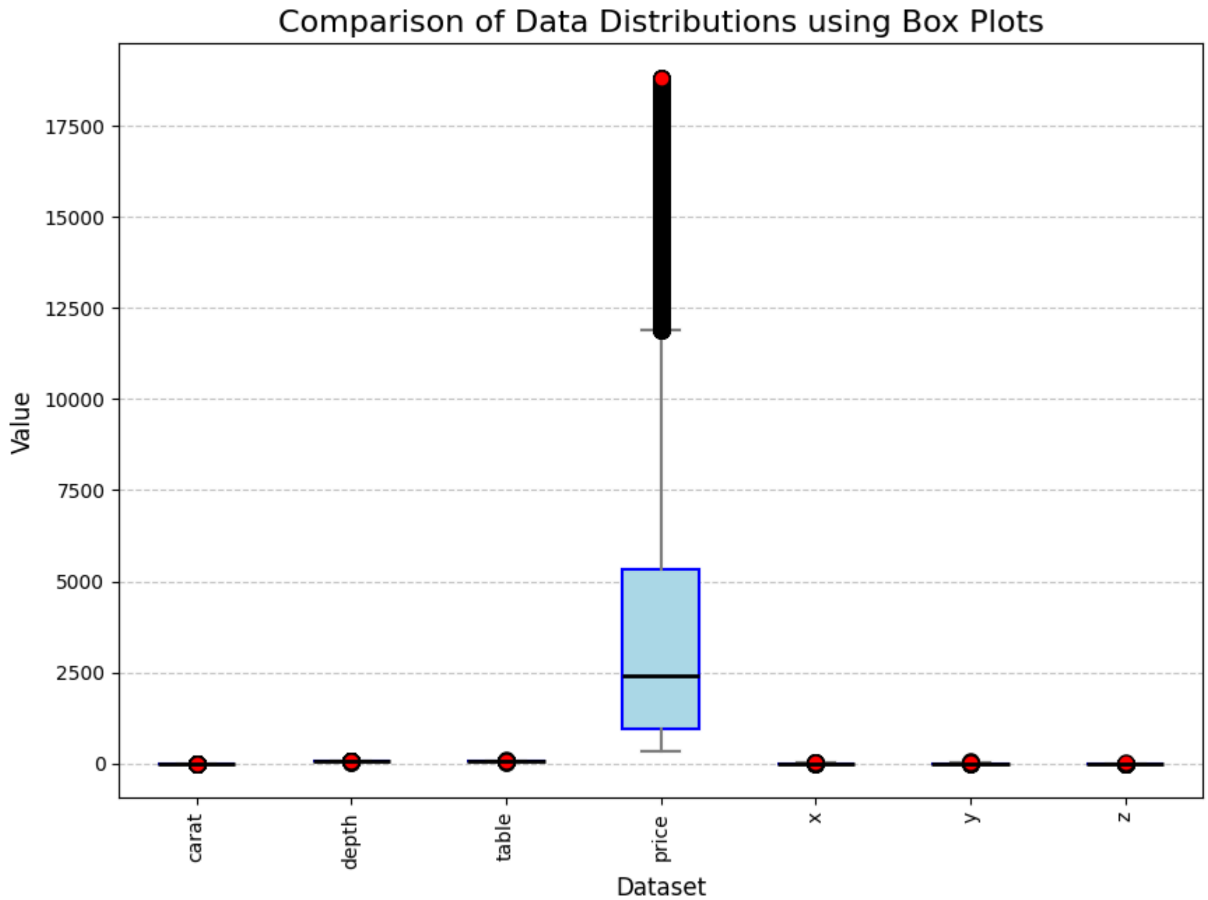
Out[8]:
```
dtype('float64')
```

In [9]:
```
cols= list(data.columns)
numeric_data = data.select_dtypes(include=['float64', 'int64'])
numeric_cols = list(numeric_data.columns)
heat_map_viz(numeric_data)
```

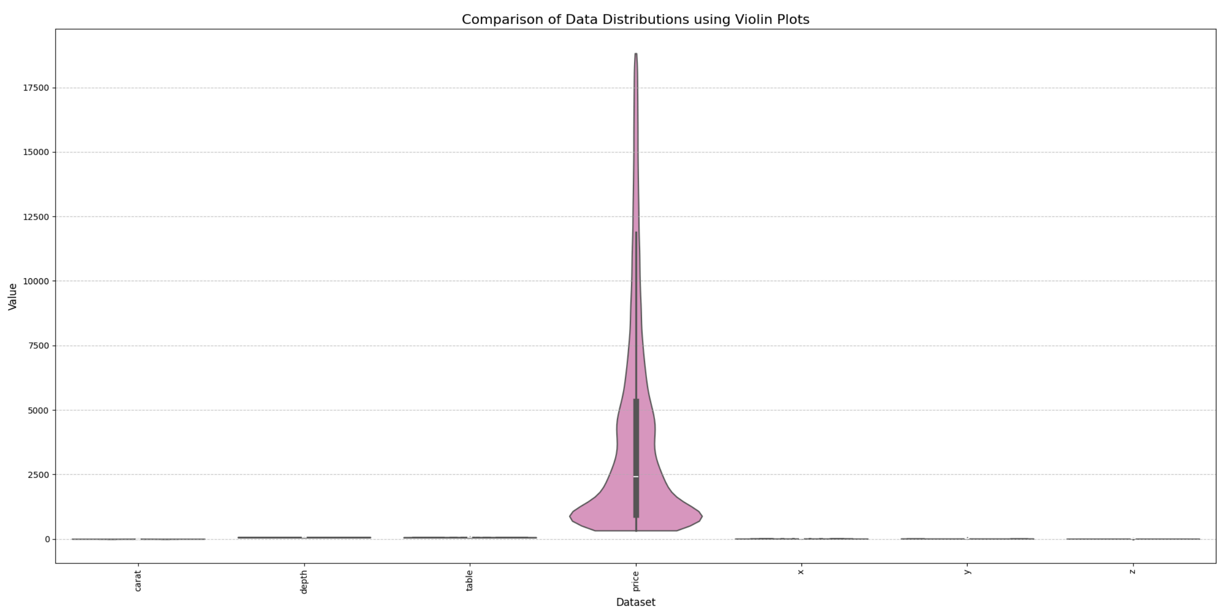


In [10]:
```
box_plot_viz(numeric_data)
```

<Figure size 2000x1000 with 0 Axes>



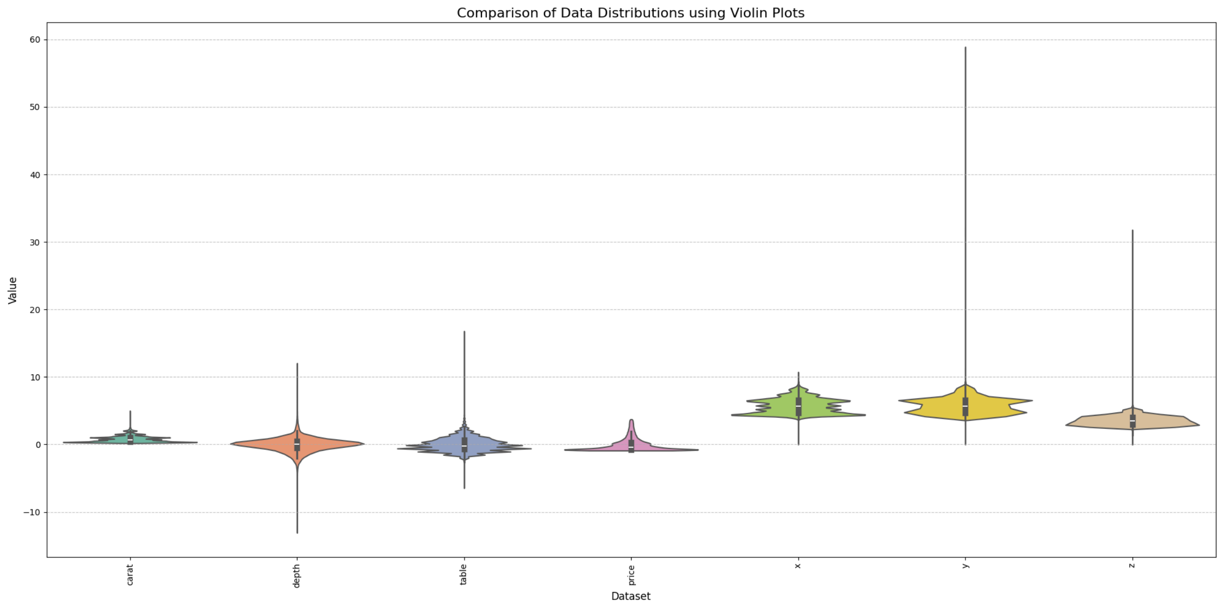
In [11]:
```
violin_plot_viz(numeric_data)
```



Scale data for the column price

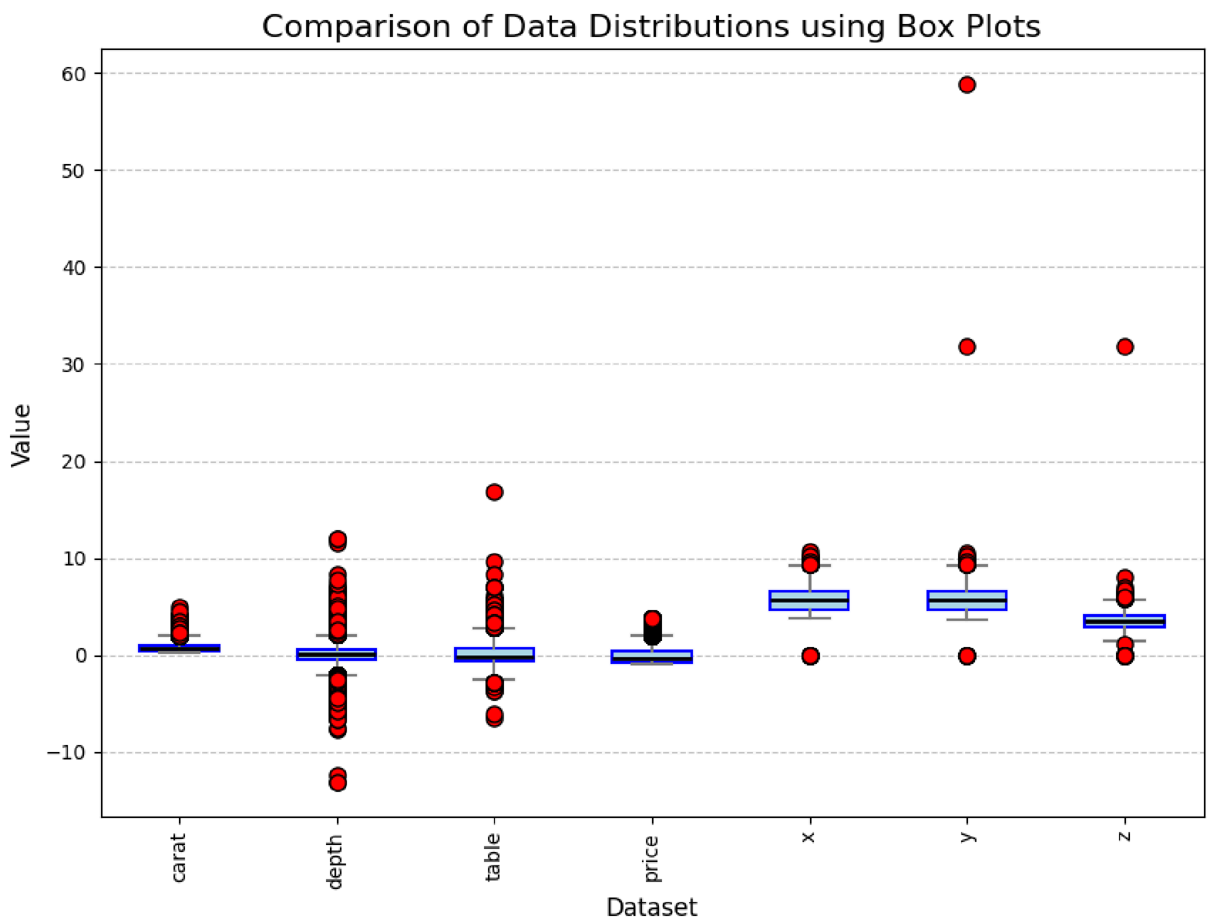
In [12]:
```
scaler = StandardScaler()
numeric_data['price'] = scaler.fit_transform(numeric_data[['price']])
numeric_data['depth'] = scaler.fit_transform(numeric_data[['depth']])
```

```
numeric_data['table'] = scaler.fit_transform(numeric_data[['table']])
violin_plot_viz(numeric_data)
```



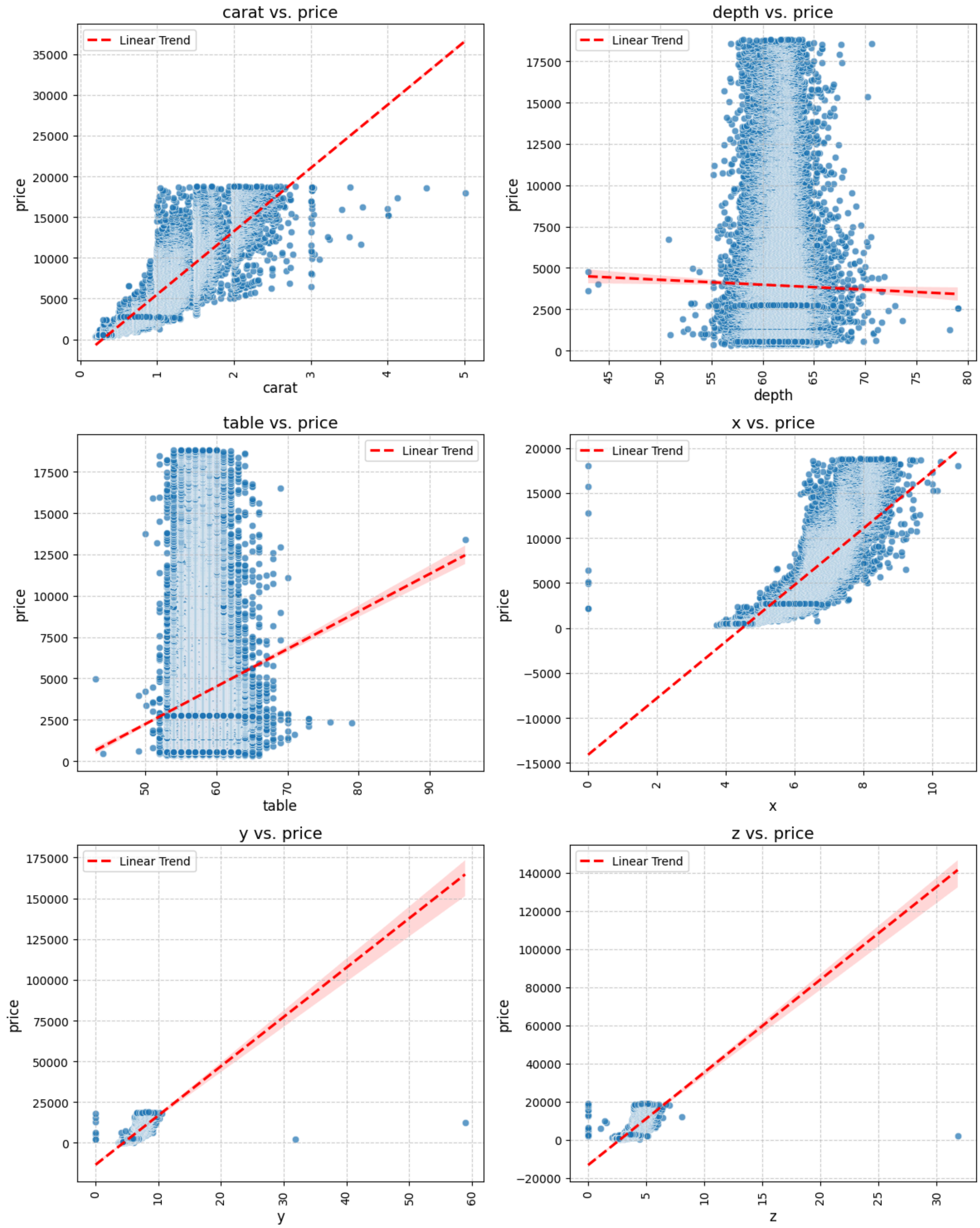
In [13]:
```
box_plot_viz(numeric_data)
```

```
<Figure size 2000x1000 with 0 Axes>
```

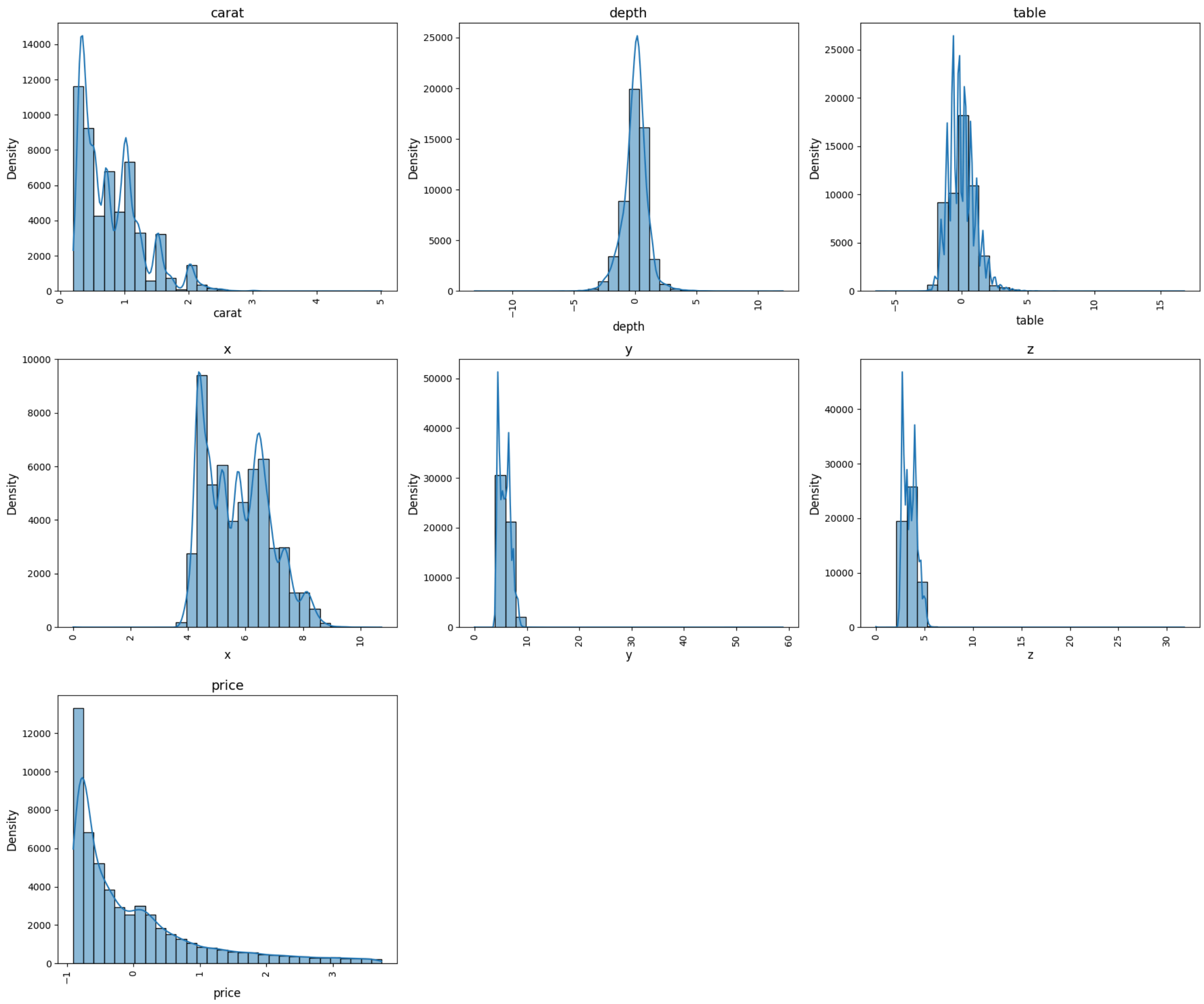


In [14]:
```
X_cols = ['carat', 'depth', 'table', 'x', 'y', 'z']
Y_cols = ['price']
```

```
line_plot_viz(X_columns=X_cols, Y_columns=Y_cols, data=data)
```



In [15]:
```
hist_plot_viz(columns=[*X_cols, *Y_cols], data=numeric_data)
```



VIZ Analysis to test for multi-colinearity

- From the heat map we can see that X, Y, Z and carat have high multicolinearity

In [16]:
```
# VIF calculation
X_scaled = StandardScaler().fit_transform(data[numeric_cols].drop(['price'],
vif_data = pd.DataFrame()
vif_data['feature'] = data[numeric_cols].drop('price', axis=1).columns
vif_data['VIF'] = [variance_inflation_factor(X_scaled, i) for i in range(X_s
vif_data.sort_values('VIF',ascending=False)
```

Out[16]:

	feature	VIF
3	x	56.187704
5	z	23.530049
0	carat	21.602712
4	y	20.454295
1	depth	1.496590
2	table	1.143225

In [17]:
```
# VIF calculation after removing highly multi-colinear features
X_scaled = StandardScaler().fit_transform(data[numeric_cols].drop(['price',
vif_data = pd.DataFrame()
vif_data['feature'] = data[numeric_cols].drop(['price', 'x', 'y', 'z'], axis
vif_data['VIF'] = [variance_inflation_factor(X_scaled, i) for i in range(X_s
vif_data.sort_values('VIF',ascending=False)
```

Out[17]:

	feature	VIF
2	table	1.141032
1	depth	1.104275
0	carat	1.042039

Feature Removal

- Remove x,y,z as they have high multicolinearity with carat

In [18]:
```
cols_to_remove = ['x', 'y', 'z']
numeric_data.drop(columns= cols_to_remove, inplace=True)
procssd_numeric_cols = [x for x in X_cols if x not in cols_to_remove]
procssd_numeric_cols
```

Out[18]:
```
['carat', 'depth', 'table']
```

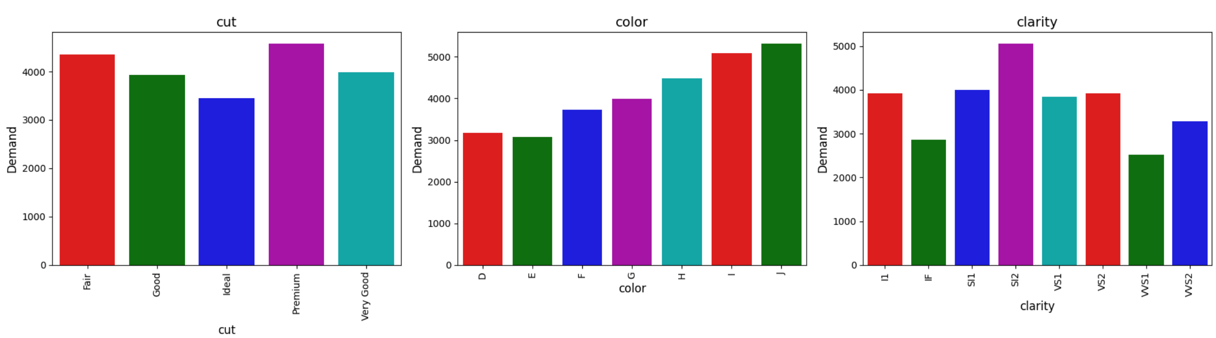
Identify the categorical variables

In [19]:
```
def convert_to_categorical_dtypes(cat_threshold, cols, proc_data):
    data = proc_data.copy()
    for feature in cols:
        uniq_vals = data[feature].unique()
        if(len(uniq_vals) < cat_threshold):
            data[feature] = data[feature].astype('category')
    print(data[cols].dtypes)
    return data
cat_threshold = 30
data = convert_to_categorical_dtypes(cat_threshold=cat_threshold, cols=data.
```

```
carat       float64
cut        category
color      category
clarity    category
depth       float64
table       float64
price         int64
x           float64
y           float64
z           float64
dtype: object
```

For the Categorical variables lets aggregate demand by each category and plot

In [20]:
```
cat_cols = data.select_dtypes(include='category').columns
num_features = len(cat_cols)
n_cols = 3 # You can adjust the number of columns in the grid
n_rows = (num_features + n_cols - 1) // n_cols # Calculate rows needed
colors = ['r', 'g', 'b', 'm', 'c']
fig, axes = plt.subplots(n_rows, n_cols, figsize=(n_cols * 6, n_rows * 5))
axes = axes.flatten() # Flatten the 2D array of axes for easy iteration
i = 0

for feature in cat_cols:
    uniq_vals = data[feature].unique()
    cat_average = data.groupby(feature)[*Y_cols].mean()

    # 3. Iterate through each feature and plot its relationship with the tar
    ax = axes[i] # Get the current subplot axis

    sns.barplot(data=cat_average, x= feature, y= 'price',hue=feature, ax=ax,

    ax.set_title(f'{feature}', fontsize=14)
    ax.set_xlabel(feature, fontsize=12)
    ax.set_ylabel('Demand', fontsize=12)
    ax.set_xticklabels(ax.get_xticklabels(), rotation=90)

    i = i + 1
for j in range(i, len(axes)):
        fig.delaxes(axes[j])

plt.tight_layout() # Adjust layout to prevent overlapping titles/labels
plt.show()
```



Remove columns

- No categorical columns to remove

Create the processed data by combining the processed numeric and categorical columns

In [21]:
```
processed_data = pd.concat([data[cat_cols], numeric_data], axis=1)
processed_data.head()
```

Out[21]:

	cut	color	clarity	carat	depth	table	price
0	Ideal	E	SI2	0.23	-0.174092	-1.099672	-0.904095
1	Premium	E	SI1	0.21	-1.360738	1.585529	-0.904095
2	Good	E	VS1	0.23	-3.385019	3.375663	-0.903844
3	Premium	I	VS2	0.29	0.454133	0.242928	-0.902090
4	Good	J	SI2	0.31	1.082358	0.242928	-0.901839

In [22]:
```
processed_data.dtypes
```

Out[22]:
```
cut         category
color       category
clarity     category
carat        float64
depth        float64
table        float64
price        float64
dtype: object
```

Encode the categorical variables using the below

cut quality of the cut (Fair, Good, Very Good, Premium, Ideal)

color diamond colour, from J (worst) to D (best)

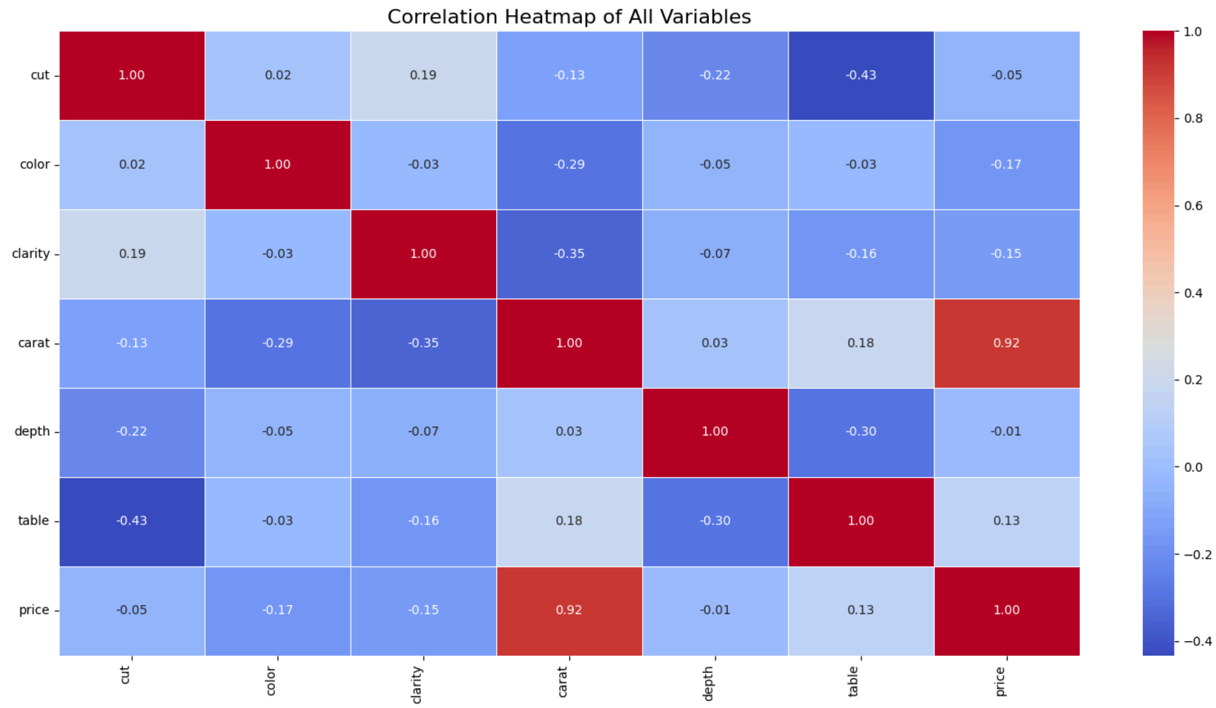
clarity a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))

In [23]:
```
encoder = OrdinalEncoder(categories=[['Fair', 'Good', 'Very Good', 'Premium'
processed_data['cut'] = encoder.fit_transform(processed_data[['cut']])

encoder = OrdinalEncoder(categories=[['J', 'I', 'H', 'G', 'F', 'E', 'D']])
processed_data['color'] = encoder.fit_transform(processed_data[['color']])

encoder = OrdinalEncoder(categories=[['I1', 'SI2', 'SI1', 'VS2', 'VS1', 'VVS
processed_data['clarity'] = encoder.fit_transform(processed_data[['clarity']

processed_data.head()
```

Out[23]:

	cut	color	clarity	carat	depth	table	price
0	4.0	5.0	1.0	0.23	-0.174092	-1.099672	-0.904095
1	3.0	5.0	2.0	0.21	-1.360738	1.585529	-0.904095
2	1.0	5.0	4.0	0.23	-3.385019	3.375663	-0.903844
3	3.0	1.0	3.0	0.29	0.454133	0.242928	-0.902090
4	1.0	0.0	1.0	0.31	1.082358	0.242928	-0.901839

In [24]:
```
heat_map_viz(processed_data)
```



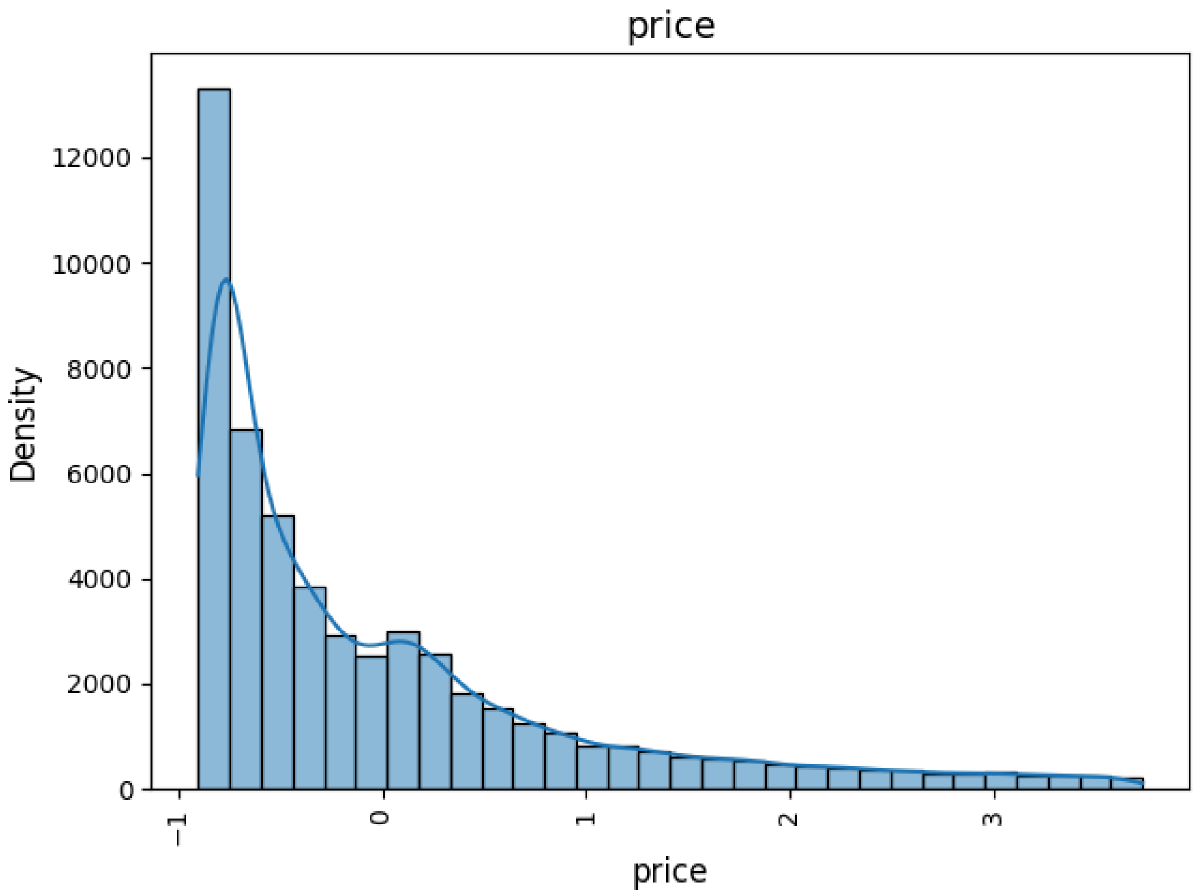
In [25]:
```
Y_cols = ['price']
Y = processed_data[Y_cols]
X = processed_data.drop(Y_cols, axis=1)
X_cols = list(X.columns)

print('Target column is: ', Y_cols)
print('Features are: ', X_cols)
```

```
Target column is:  ['price']
Features are:  ['cut', 'color', 'clarity', 'carat', 'depth', 'table']
```

A look at the distribution of the target variable Price

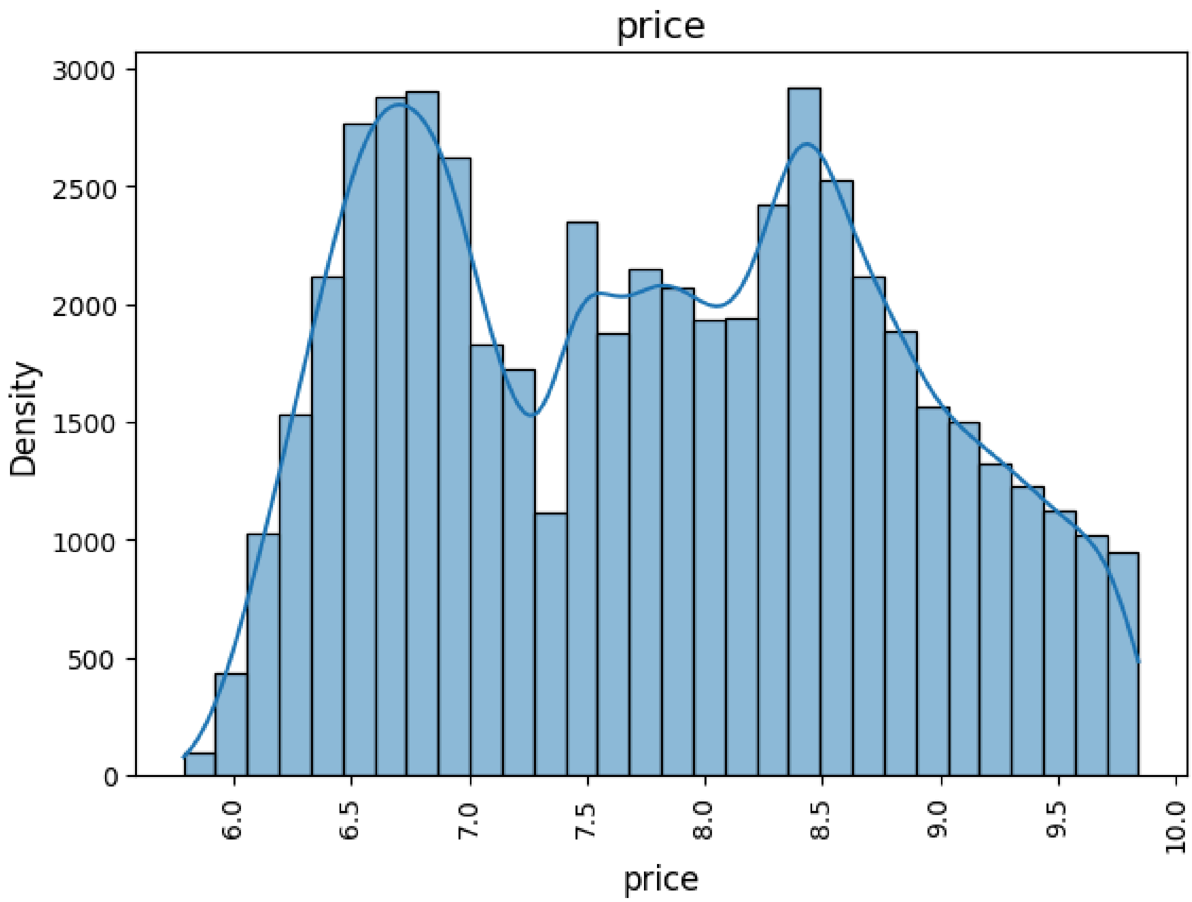
- According to the plot below the distribution is not normal so let us apply a log normal transform

In [26]:
```
hist_plot_viz(Y_cols, processed_data)
```



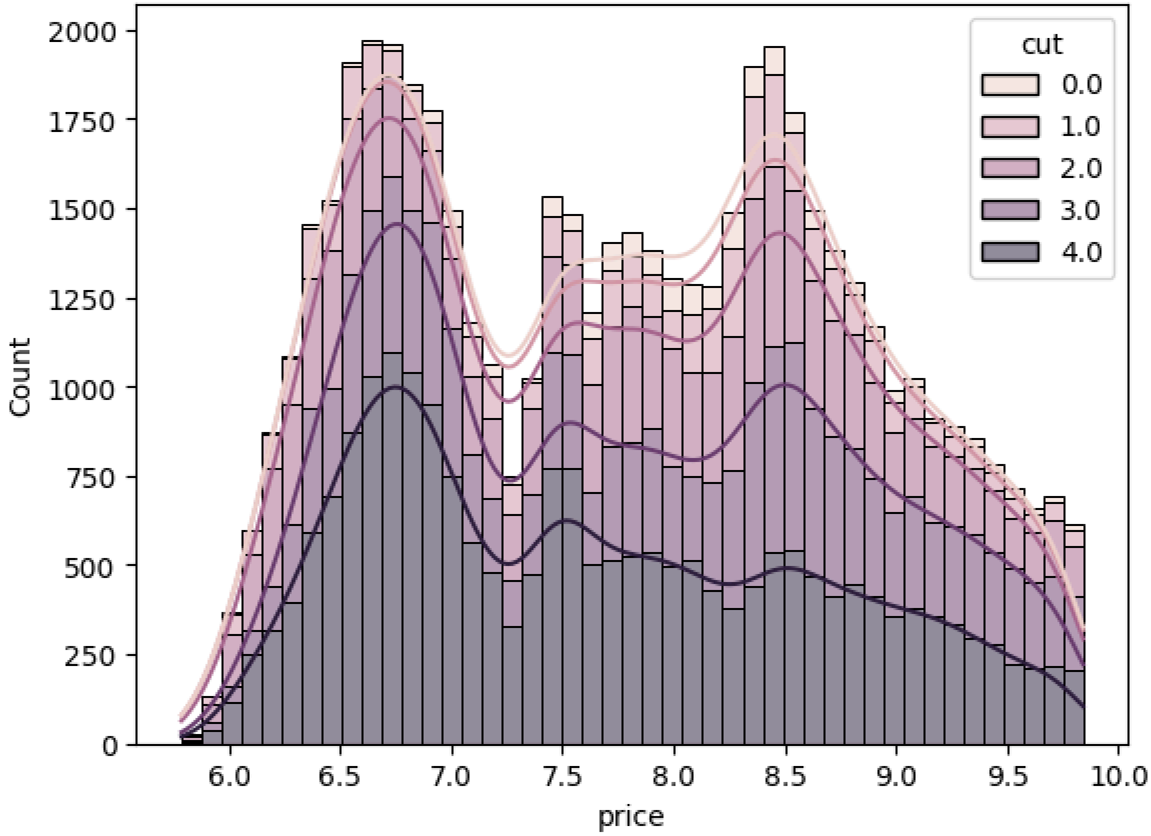
Below is the graph after the log normal transformation

In [27]:
```
processed_data[Y_cols] = np.log(data[Y_cols])
hist_plot_viz(Y_cols, processed_data)
```



In [28]:
```
sns.histplot(data=processed_data, x='price', hue='cut', kde=True, multiple='
```

Out[28]: <Axes: xlabel='price', ylabel='Count'>



In [29]:
```
df = processed_data.copy()
df['price_per_carat'] = df['price'] / df['carat']
df.groupby('cut')['price_per_carat'].mean().sort_values()
```

Out[29]:
```
cut
0.0     9.226632
1.0    11.575319
3.0    11.755446
2.0    12.499832
4.0    13.902481
Name: price_per_carat, dtype: float64
```

In [30]:
```
df.groupby('cut')['price'].mean().sort_values()
```

Out[30]:
```
cut
4.0    7.639467
2.0    7.798664
1.0    7.842809
3.0    7.950795
0.0    8.093441
Name: price, dtype: float64
```

In [31]:
```
df = processed_data.copy()
df['price_per_carat'] = df['price'] / df['carat']
df.groupby('color')['price_per_carat'].mean().sort_values()
```

Out[31]:
```
color
0.0     8.984839
1.0    10.415987
2.0    11.514335
3.0    12.933443
4.0    13.152478
6.0    14.149139
5.0    14.256642
Name: price_per_carat, dtype: float64
```

In [32]:
```
df = processed_data.copy()
df['price_per_carat'] = df['price'] / df['carat']
df.groupby('clarity')['price_per_carat'].mean().sort_values()
```

Out[32]:
```
clarity
0.0     7.419620
1.0     9.182076
2.0    11.441858
3.0    13.001680
4.0    13.477317
5.0    15.831236
6.0    17.426974
7.0    17.652448
Name: price_per_carat, dtype: float64
```

In [33]:
```
df['cut'].value_counts().sort_index()
```

Out[33]:
```
cut
0.0     1610
1.0     4906
2.0    12082
3.0    13791
4.0    21551
Name: count, dtype: int64
```

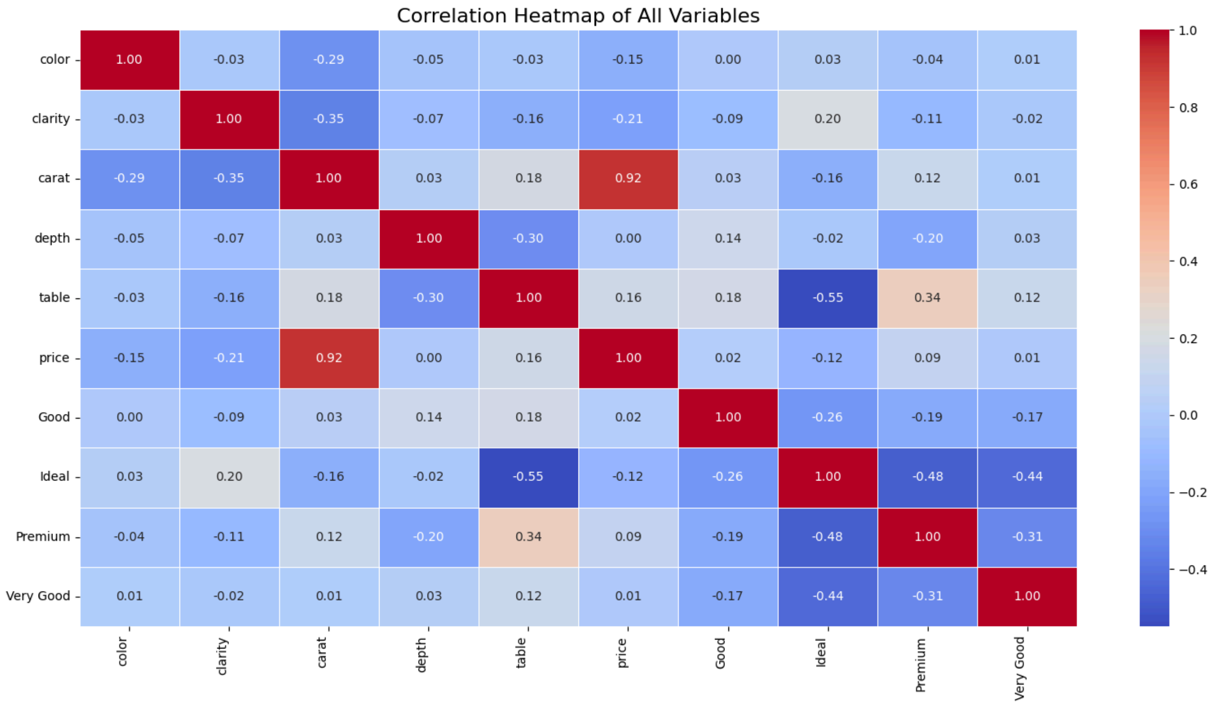
Some important observations from above results

- Though the data legend suggests that there is ordinality in the cut feature ie better cut should be priced higher but if you look at the effect of each type of cut on the price. It does not increase as the cuts get better. Infact I have a hypothesis that the mid quality cuts are overpriced and earn the most margin for the seller. There is enough data for the cut type 2 when compared to the better cuts so imbalance in the classes can also be ruled out.
- We need to remove the ordiality encoding and add one hot encoding.
- All the other categorical variables do show a trend in price due to ordinality so we will preserve that.

In [34]:
```
cuts_df = pd.get_dummies(data['cut'], drop_first=True, dtype='int')
processed_data = pd.concat([processed_data, cuts_df], axis=1)
```

In [35]:
```
processed_data.drop('cut', inplace=True, axis=1)
```

In [36]:
```
heat_map_viz(processed_data)
```



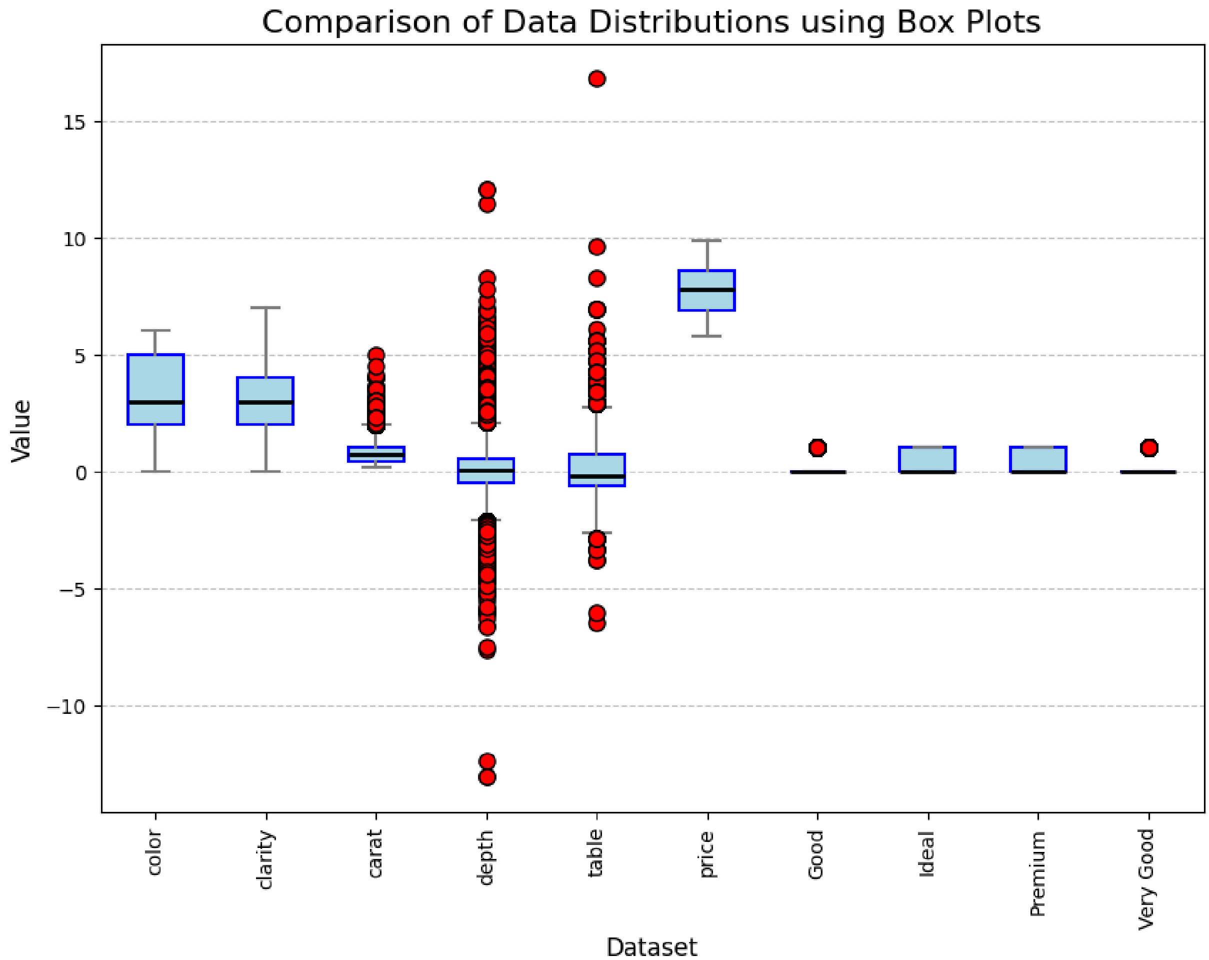
In [37]:
```
processed_data.dtypes
```

Out[37]:
```
color         float64
clarity       float64
carat         float64
depth         float64
table         float64
price         float64
Good            int64
Ideal           int64
Premium         int64
Very Good       int64
dtype: object
```

Detect outliers If any

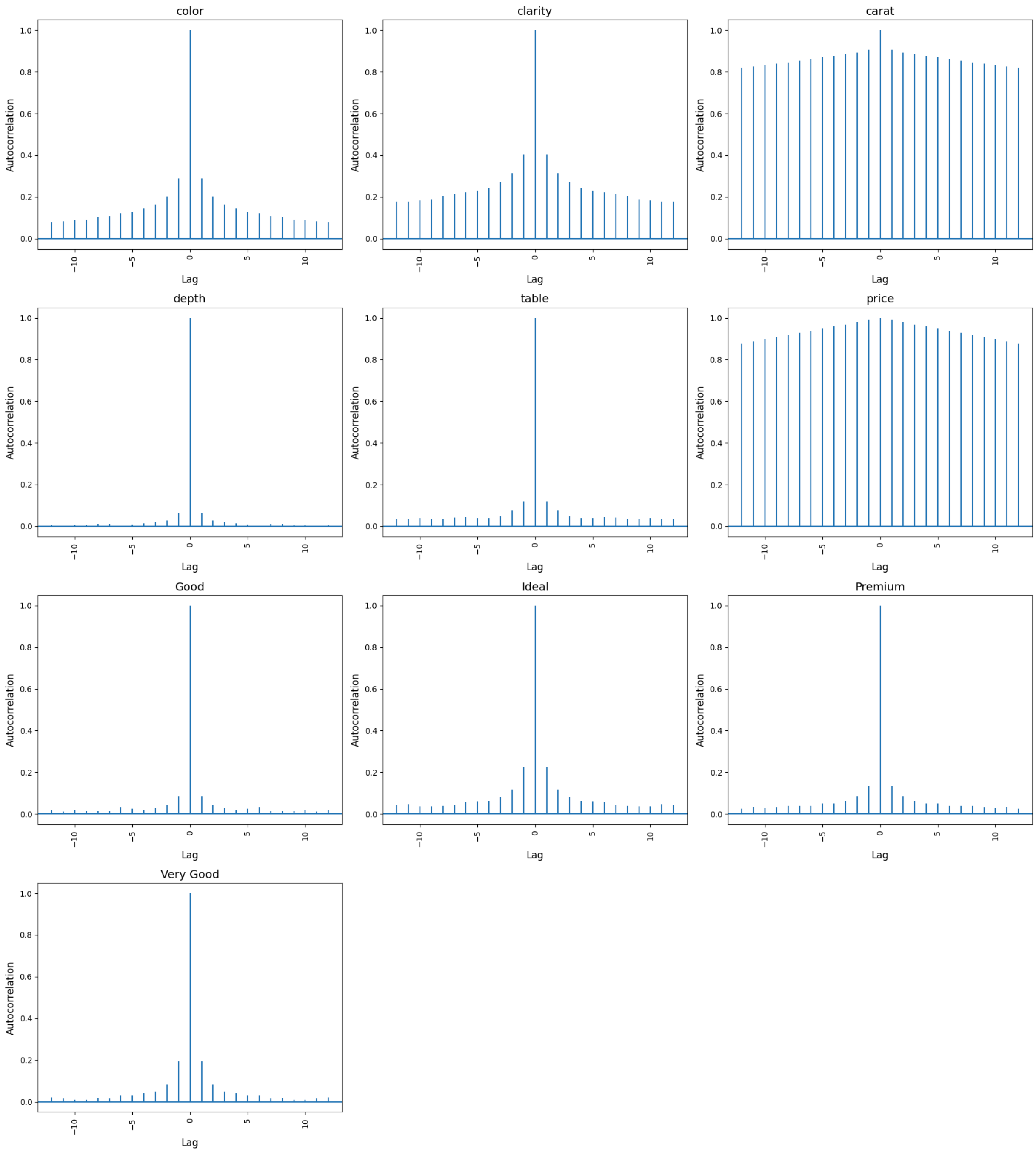
In [38]:
```
box_plot_viz(processed_data)
```

```
<Figure size 2000x1000 with 0 Axes>
```



Test for autocorrelation in the target column

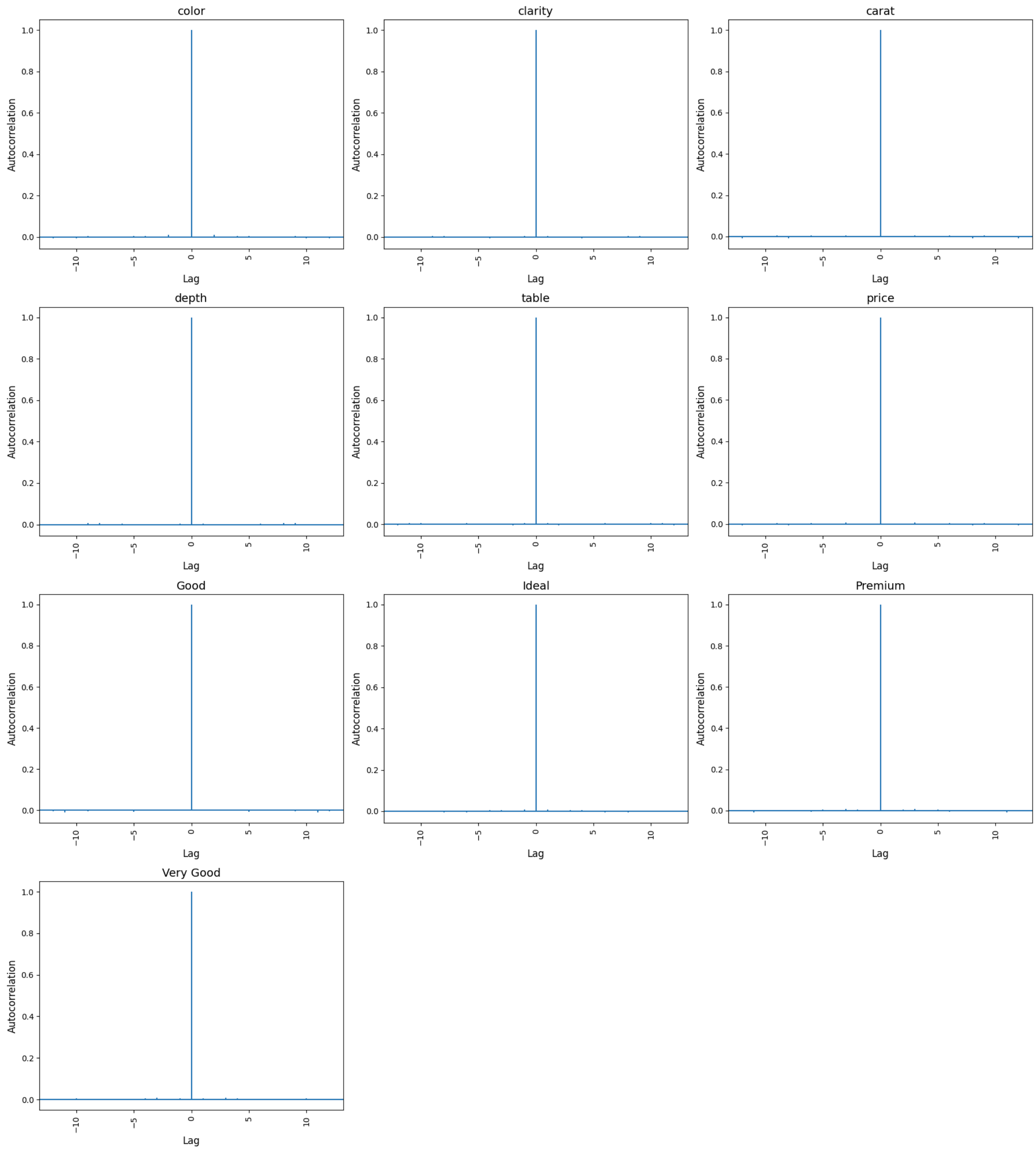
In [39]:
```
auto_corr_viz(processed_data=processed_data)
```



Looks like there is high auto corrleation in the features

- lets reshuffle the data and test this again

In [40]:
```
df_shuffled = processed_data.sample(frac=1, random_state=42).reset_index(drc
auto_corr_viz(df_shuffled)
```



No auto correlation found

Train Test Split

In [41]:
```
X = processed_data.drop(columns=['price'], axis=1)
y = processed_data['price']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Fit the linear regression model to get the baseline

In [42]:
```
model = LinearRegression()
model.fit(X = X_train, y = y_train)
```

Out[42]:

▾ LinearRegression

▶ Parameters

R Squared metrics of the model

In [43]:
```
r2_train = model.score(X = X_train, y = y_train)
print('r2 train : ', r2_train)
r2_test = model.score(X = X_test, y = y_test)
print('r2 test : ', r2_test)
```

```
r2 train :  0.8807282361735038
r2 test :  0.8786308335563693
```

In [44]:
```
# RMSE
y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f"RMSE: {rmse:.4f}")
```

```
RMSE: 0.3538
```

In [45]:
```
rmsle_lin = np.sqrt(mean_squared_log_error(y_test, y_pred))
print(f"RMSLE: {rmsle_lin:.4f}")
```

```
RMSLE: 0.0387
```

Try other models

In [46]:
```
models=[RandomForestRegressor(),AdaBoostRegressor(),BaggingRegressor(),SVR()
model_names=['RandomForestRegressor','AdaBoostRegressor','BaggingRegressor',
rmsle=[]
d={}
for model in range (len(models)):
    clf=models[model]
    clf.fit(X_train,y_train)
    test_pred=clf.predict(X_test)
    rmsle.append(np.sqrt(mean_squared_log_error(y_test, test_pred)))
d={'Modelling Algo':model_names,'RMSLE':rmsle}
```

In [47]:
```
model_metrics = pd.DataFrame(d)
model_metrics.loc[len(model_metrics)] = {'Modelling Algo':'Linear regressior
model_metrics.sort_values(by='RMSLE')
```

Out[47]:

	Modelling Algo	RMSLE
0	RandomForestRegressor	0.013271
2	BaggingRegressor	0.013634
3	SVR	0.015268
1	AdaBoostRegressor	0.023128
5	Linear regression	0.038740
4	KNeighborsRegressor	0.040098

Looks like Random forest is our best bet

- Lets do some cross validation to confirm this.

In [48]:
```
scores = cross_val_score(RandomForestRegressor(), X, y, cv=5, scoring='neg_m
rmsle_scores = np.sqrt(-scores)
print("CV RMSLE:", rmsle_scores.mean(), "+/-", rmsle_scores.std())
```

```
CV RMSLE: 0.027997162383701785 +/- 0.004863930254715491
```

Lets find the feature importance of the model features

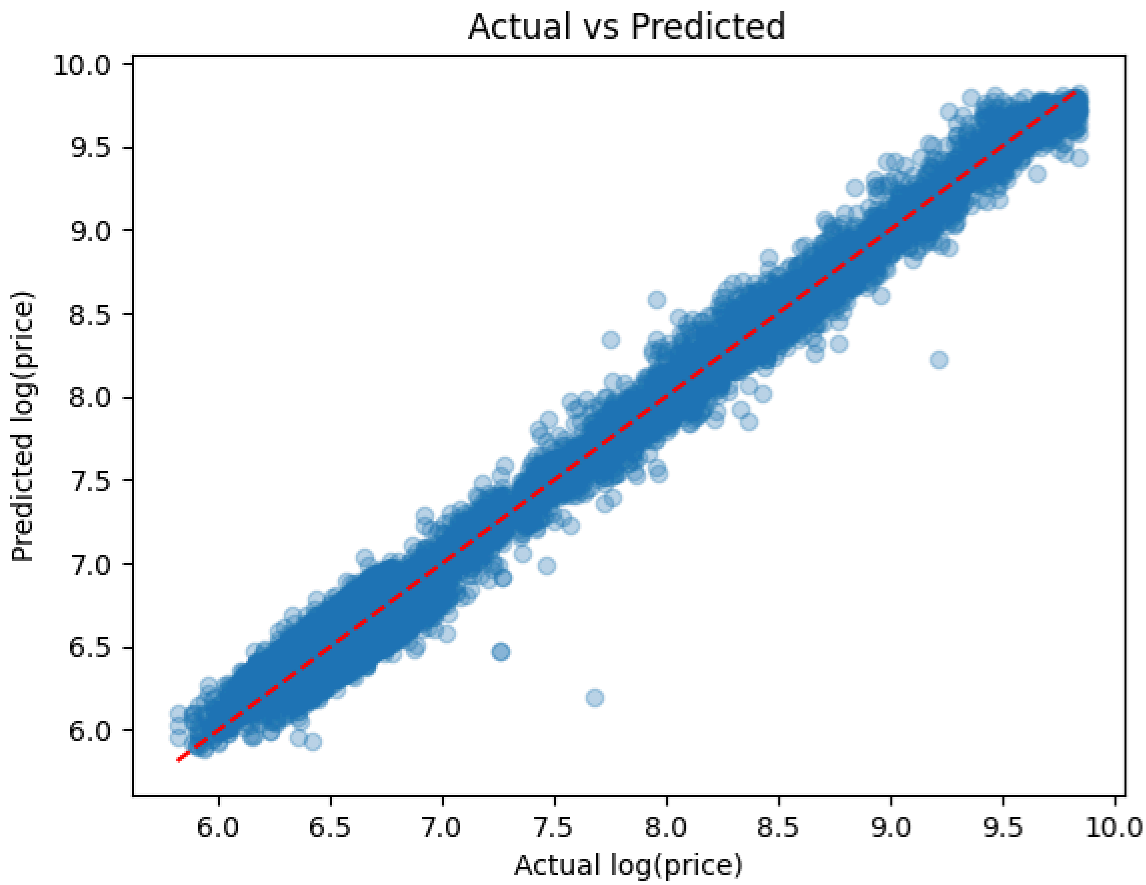
In [49]:
```
model = RandomForestRegressor().fit(X_train, y_train)
importances = model.feature_importances_
sorted(zip(importances, X.columns), reverse=True)
```

Out[49]:
```
[(np.float64(0.9430089772489625), 'carat'),
 (np.float64(0.033465851134556), 'clarity'),
 (np.float64(0.014820016813689572), 'color'),
 (np.float64(0.004488540757510453), 'depth'),
 (np.float64(0.00204680831862309), 'table'),
 (np.float64(0.0009175120906640404), 'Ideal'),
 (np.float64(0.0005276268881232428), 'Premium'),
 (np.float64(0.0004685753238514057), 'Very Good'),
 (np.float64(0.0002560914240197715), 'Good')]
```

Carat is the most important and dominant feature.

Now lets plot the Actuals vs predicted

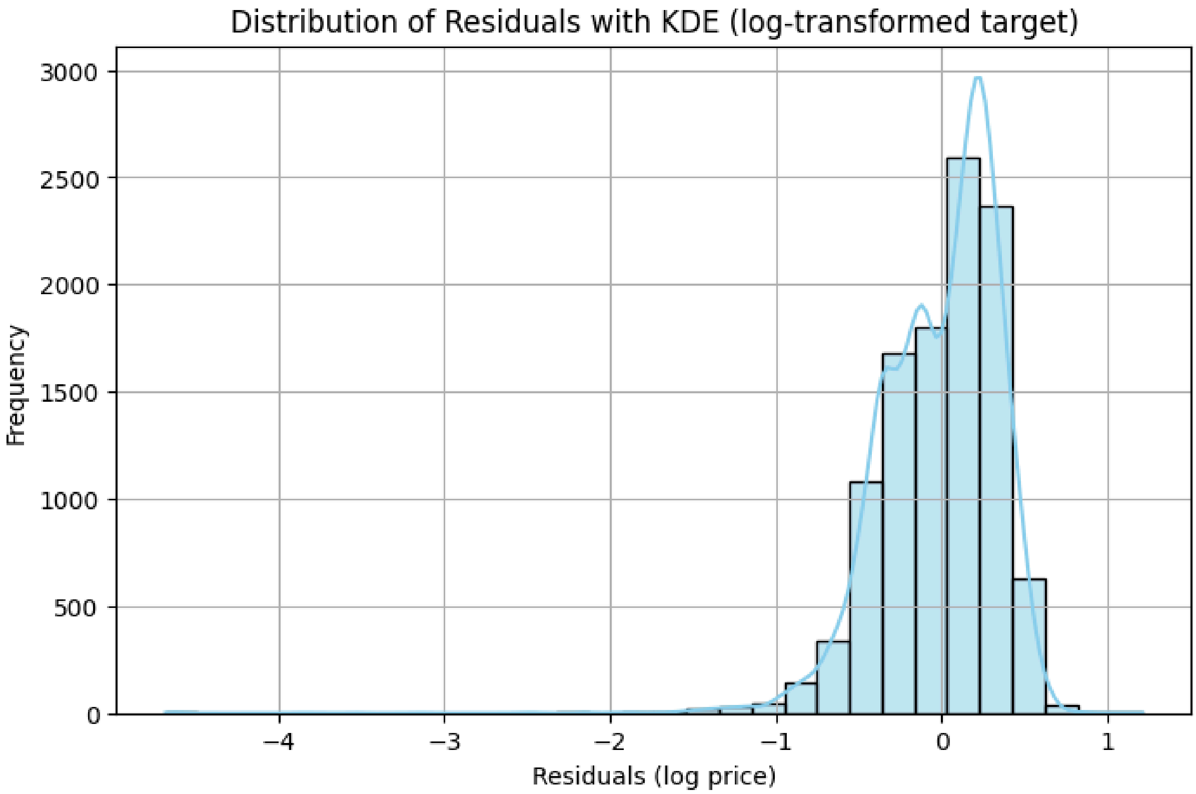
In [50]:
```
y_pred_rf = model.predict(X_test)
plt.scatter(y_test, model.predict(X_test), alpha=0.3)
plt.xlabel("Actual log(price)")
plt.ylabel("Predicted log(price)")
plt.title("Actual vs Predicted")
min_val = min(y_test.min(), y_pred_rf.min())
max_val = max(y_test.max(), y_pred_rf.max())
plt.plot([min_val, max_val], [min_val, max_val], 'r--', label='Perfect Predi
plt.show()
```



The Residual

- Looks roughly normal

In [51]:
```
residuals = y_test - y_pred

# Plot
plt.figure(figsize=(8, 5))
sns.histplot(residuals, kde=True, bins=30, color='skyblue', edgecolor='black
plt.title("Distribution of Residuals with KDE (log-transformed target)")
plt.xlabel("Residuals (log price)")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```



Important! learn about VIF

In [ ]: