In [1]:
```
#!pip install yellowbrick
```

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from yellowbrick.cluster import KElbowVisualizer
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.cluster import DBSCAN
from sklearn.mixture import GaussianMixture
```

In [3]:
```
df=pd.read_csv('data.csv')
df
```

0	0	-0.389420	-0.912791	0.648951	0.589045	-0.830817	0.733624	2.258560	2	13
1	1	-0.689249	-0.453954	0.654175	0.995248	-1.653020	0.863810	-0.090651	2	3
2	2	0.809079	0.324568	-1.170602	-0.624491	0.105448	0.783948	1.988301	5	11
3	3	-0.500923	0.229049	0.264109	0.231520	0.415012	-1.221269	0.138850	6	2
4	4	-0.671268	-1.039533	-0.270155	-1.830264	-0.290108	-1.852809	0.781898	8	7
...	...	...	...	...	...	...	...	...	...	...
97995	97995	0.237591	1.657034	-0.689282	0.313710	-0.299039	0.329139	1.607378	5	7
97996	97996	0.322696	0.710411	0.562625	-1.321713	-0.357708	0.182024	0.178558	3	9
97997	97997	-0.249364	-0.459545	1.886122	-1.340310	0.195029	-0.559520	-0.379767	8	9
97998	97998	0.311408	2.185237	0.761367	0.436723	0.464967	0.062321	-0.334025	1	8
97999	97999	0.755170	0.567483	1.456767	-0.579071	-0.048474	-1.206240	0.784305	0	11

98000 rows × 30 columns

EDA

For EDA, we will proceed with the steps below:

Describe the Dataset Components: We will look into the distributions of the features using statistical summaries

and visualizations.

Check for Correlations: Identify if any features are strongly correlated with others.

Data Cleaning: Look for any anomalies, such as missing values, outliers, or skewed information and treat the data.

Feature Engineering: As there is 29 features, we might want to tackle the dimensionality issue,and see if we can conduct

feature selection, extraction or reduction (although without contextual knowledge) to aid in subsequent modeling.

In [4]:
```
df.info()
df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 98000 entries, 0 to 97999
Data columns (total 30 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   id      98000 non-null  int64  
 1   f_00    98000 non-null  float64
 2   f_01    98000 non-null  float64
 3   f_02    98000 non-null  float64
 4   f_03    98000 non-null  float64
 5   f_04    98000 non-null  float64
 6   f_05    98000 non-null  float64
 7   f_06    98000 non-null  float64
 8   f_07    98000 non-null  int64  
 9   f_08    98000 non-null  int64  
 10  f_09    98000 non-null  int64  
 11  f_10    98000 non-null  int64  
 12  f_11    98000 non-null  int64  
 13  f_12    98000 non-null  int64  
 14  f_13    98000 non-null  int64  
 15  f_14    98000 non-null  float64
 16  f_15    98000 non-null  float64
 17  f_16    98000 non-null  float64
 18  f_17    98000 non-null  float64
 19  f_18    98000 non-null  float64
 20  f_19    98000 non-null  float64
 21  f_20    98000 non-null  float64
 22  f_21    98000 non-null  float64
 23  f_22    98000 non-null  float64
 24  f_23    98000 non-null  float64
 25  f_24    98000 non-null  float64
 26  f_25    98000 non-null  float64
 27  f_26    98000 non-null  float64
 28  f_27    98000 non-null  float64
 29  f_28    98000 non-null  float64
dtypes: float64(22), int64(8)
memory usage: 22.4 MB
```

Out[4]:

	id	f_00	f_01	f_02	f_03	f_04	f_0
count	98000.000000	98000.000000	98000.000000	98000.000000	98000.000000	98000.000000	98000.00000
mean	48999.500000	0.001220	0.005580	-0.001042	-0.000700	-0.003522	-0.00161
std	28290.307527	1.002801	1.000742	1.001373	1.000422	1.003061	1.00053
min	0.000000	-4.732235	-4.202795	-4.377021	-4.010826	-4.535903	-4.30076
25%	24499.750000	-0.675226	-0.670985	-0.672779	-0.672540	-0.682510	-0.67506
50%	48999.500000	0.002022	0.006650	-0.000324	-0.003185	-0.003307	0.00102
75%	73499.250000	0.677271	0.677746	0.677086	0.672097	0.677589	0.67334
max	97999.000000	4.490521	4.324974	4.560247	4.399373	4.050549	4.71031

8 rows × 30 columns

From the initial inspection, the features have varying ranges, as indicated by the differences between their minimum

and maximum values. This variation suggests that normalization or scaling may be necessary before performing clustering,

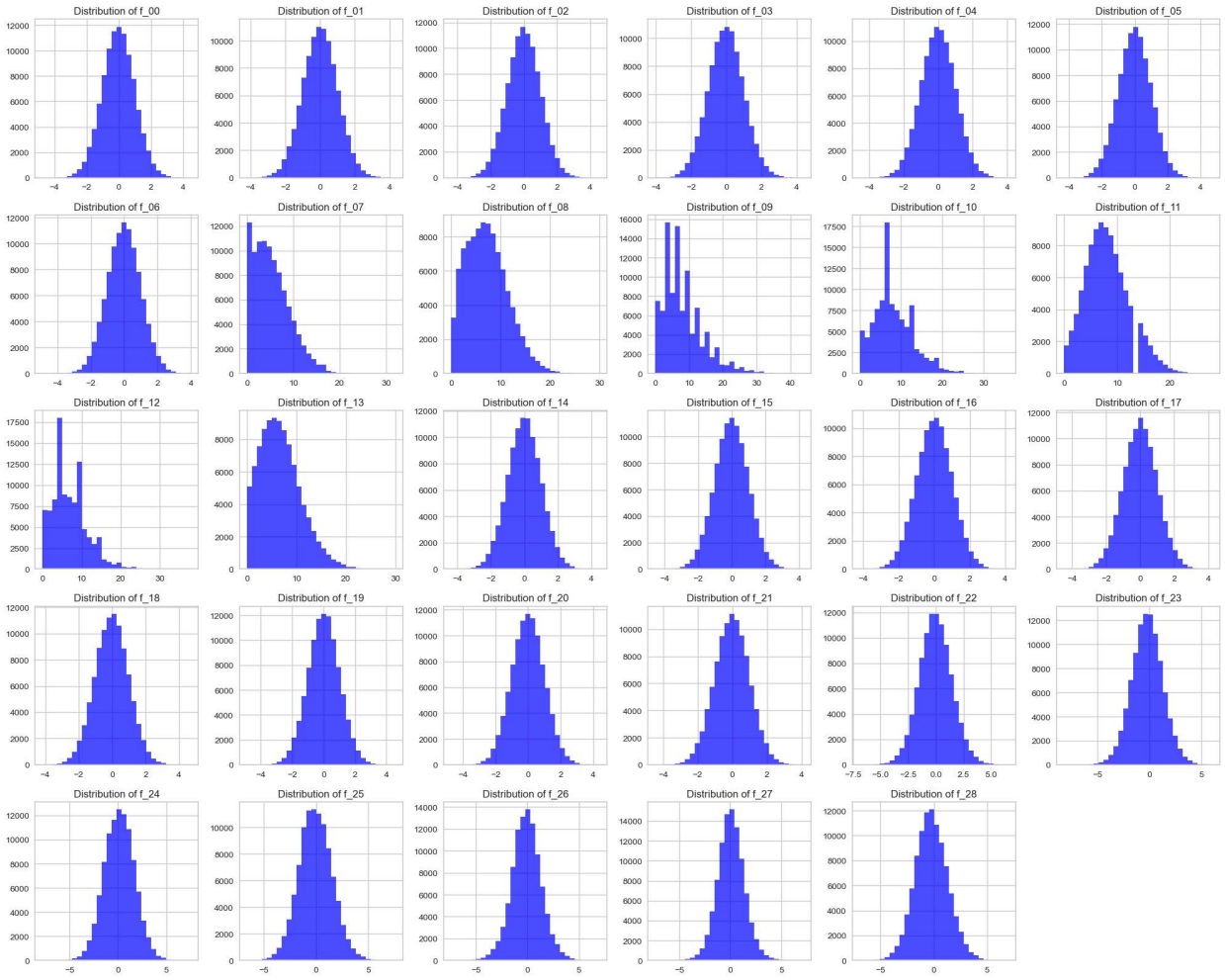
as many clustering algorithms are sensitive to the scale of the data. The standard deviations suggest differing levels of

variance in the features. Features with higher variance might be more influential in the clustering process

Next, we will create histograms for each feature to understand their distributions.

This visual approach will help identify if any features are skewed or have outliers.

In [5]:
```
plt.figure(figsize=(20, 16))

for i in range(29):
    plt.subplot(5, 6, i + 1)
    plt.hist(df[f'f_{str(i).zfill(2)}'], bins=30, color='blue', alpha=0.7)
    plt.title(f'Distribution of f_{str(i).zfill(2)}')
    plt.tight_layout()

plt.show()
```



The histograms provide a visual representation of each feature's distribution.

Here is what we can discern from these plots:

Variety in Distributions: While most of the features appear to have relatively normal distribution,

some features like 'f_06' and 'f_07' seemed to have skewed distribution.

Presence of Outliers: Some histograms like 'f_09' seems to have some potential outliers.

These are visible in features with long tails, indicating values significantly higher or lower than the majority of

the data.

Different Ranges: The features have varying ranges (minimum and

maximum values),

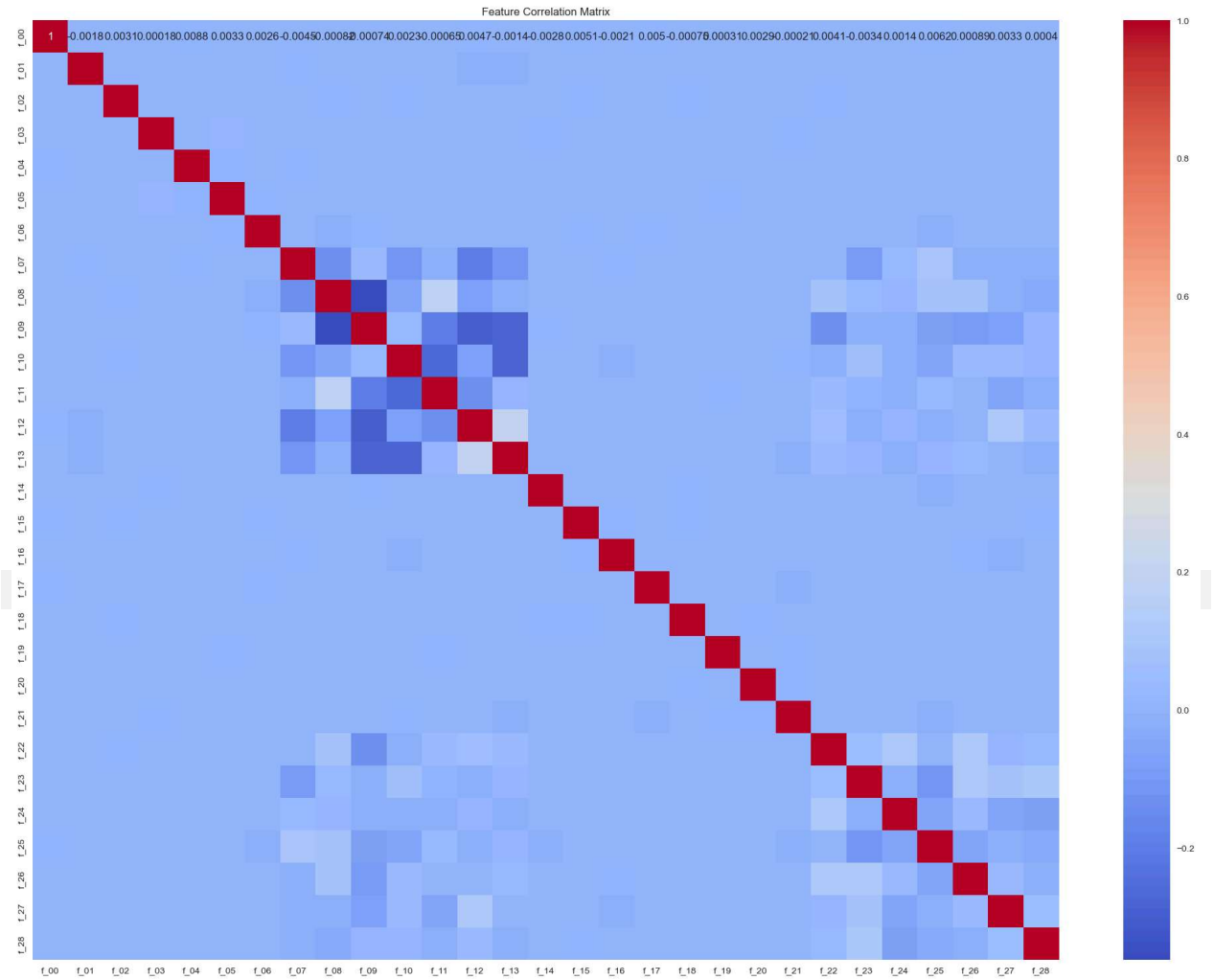
which could affect algorithms sensitive to scale. For instance, 'f_04' has a broader range compared to 'f_22'.

Clearly, the visual inspection suggest the need to delve deeper into outlier detection and formulate a strategy

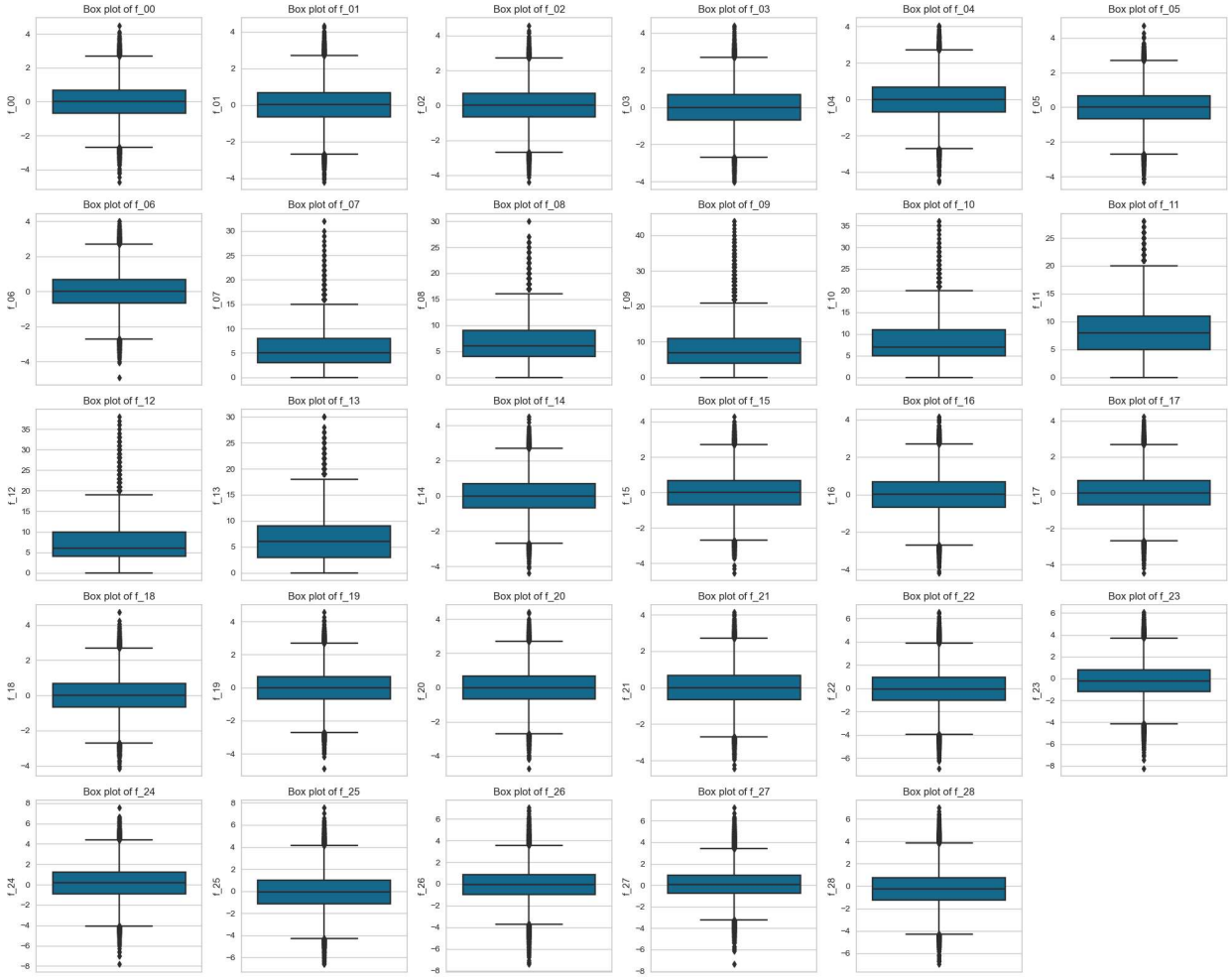
for potentially transforming the data for optimal results in subsequent clustering or predictive modeling.

But before we delve into that, we will examine the features deeper by creating a correlation matrix to see if any

In [18]:
```
import seaborn as sns
correlation_matrix = df.drop(columns=['id']).corr()
plt.figure(figsize=(20,15))
sns.heatmap(correlation_matrix, annot = True ,cmap = 'coolwarm')
plt.title("Feature Correlation Matrix")
plt.tight_layout()
plt.show()
```



In [12]:
```
plt.figure(figsize=(20, 16))

for i in range(29):
    plt.subplot(5, 6, i + 1)
    sns.boxplot(y=df[f'f_{str(i).zfill(2)}'])
    plt.title(f'Box plot of f_{str(i).zfill(2)}')
    plt.tight_layout()

plt.show()
```



The box plots reveal several insights regarding outliers in our dataset:

Presence of Outliers: Most features have numerous outliers, as evidenced by the points located outside the

"whiskers" of the box plots. These outliers could potentially skew our analyses and models subsequently.

Varying Ranges of Feature Values: The box plots also highlight the differing ranges of values that each feature

takes on, confirming what we observed in the histograms. Some features

have a compact interquartile range (IQR),

while others are quite spread out.

First, we will use a common technique for handling outliers, which is to cap them at a certain percentile.

This approach involves setting a threshold (usually the 1st and 99th percentiles) and capping all the outliers

beyond this range. This method preserves the general distribution and range of the data without the extreme values

that can skew analysis. Next, we will apply standard scaling to the data. This process involves transforming the data

such that each feature has a mean of 0 and a standard deviation of 1, to aid

In [15]:
```
for feature in df.columns[1:]:
    lower_threshold = df[feature].quantile(0.01)
    upper_threshold = df[feature].quantile(0.99)
    df[feature] = np.where(df[feature] < lower_threshold, lower_threshold,
                           np.where(df[feature] > upper_threshold, upper_thres

scaler = StandardScaler()
scaled_data = df.copy()
scaled_data.iloc[:, 1:] = scaler.fit_transform(df.iloc[:, 1:])  # excluding 'id'

scaled_data.head()
```

Out[15]:

	id	f_00	f_01	f_02	f_03	f_04	f_05	f_06	f_07	f_08	...
0	0	-0.396426	-0.934976	0.661497	0.600208	-0.840673	0.748036	2.307236	-0.972958	1.522420	...
1	1	-0.700849	-0.467832	0.666811	1.013603	-1.676070	0.880484	-0.089373	-0.972958	-0.912824	...
2	2	0.820439	0.324782	-1.189630	-0.634811	0.110619	0.799234	2.031523	-0.145189	1.035371	...
3	3	-0.509638	0.227534	0.269977	0.236355	0.425150	-1.240811	0.144758	0.130734	-1.156348	...
4	4	-0.682593	-1.064012	-0.273558	-1.861930	-0.291286	-1.883320	0.800780	0.682580	0.061274	...

5 rows × 30 columns

So, outliers have been treated by capping them at the 1st and 99th percentiles.

This approach retains the overall distribution of each feature while

reducing the influence of extreme values.

Features have also been scaled using standard scaling.

This transformation ensures that each feature has a mean of 0 and a standard deviation of 1.

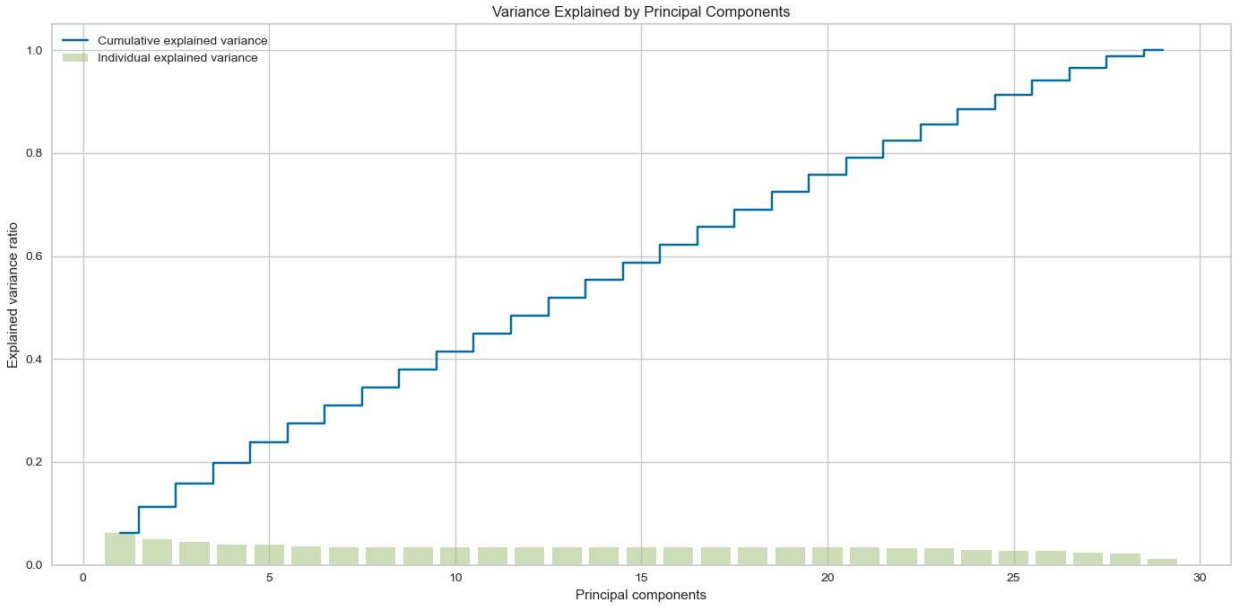
With the data cleaning done, we will proceed with the last step of EDA - feature engineering.

As we have no contextual knowledge of the data, we can utilise dimensionality reduction technique like

Principal Component Analysis (PCA) to select a subset of principal components that capture the most variance in the data.

This approach is not exactly feature selection in the traditional sense but

In [16]:
```
pca = PCA(n_components=min(*scaled_data.iloc[:, 1:].shape))
principal_components = pca.fit_transform(scaled_data.iloc[:, 1:])

explained_variance_ratio = pca.explained_variance_ratio_
cumulative_explained_variance = np.cumsum(explained_variance_ratio)

plt.figure(figsize=(14, 7))

plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio, a
        label='Individual explained variance', color='g')
plt.step(range(1, len(cumulative_explained_variance) + 1), cumulative_explained_
         label='Cumulative explained variance')

plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.title('Variance Explained by Principal Components')
plt.legend(loc='best')
plt.tight_layout()
plt.show()

cumulative_explained_variance
```



Out[16]:
```
array([0.06298837, 0.11320461, 0.15890036, 0.1990685 , 0.23782171,
       0.27441524, 0.30962085, 0.34469816, 0.37972907, 0.41459272,
       0.44941344, 0.48402328, 0.51857926, 0.55307514, 0.58749656,
       0.62182372, 0.65608889, 0.69016479, 0.72415802, 0.75796233,
       0.79166892, 0.82451681, 0.856303  , 0.88525041, 0.91351296,
       0.94076519, 0.96554843, 0.9880914 , 1.        ])
```

The chart and data represent the variance explained by each principal component and the cumulative variance explained.

Individual Explained Variance: Each bar represents the portion of total variance explained by each principal component.

Cumulative Explained Variance: The step line represents the total variance explained up to each component.

And we will use it to find the elbow.

Looking at the cumulative explained variance, we can see that there is no strong presence of very redundant

features and there is no elbow present for feature selection. While we have no contextual knowledge, it seems that

each original feature carries unique information, and therefore, the principal components each explain a portion of

the variance. With this, we will continue our analysis with the full set of features that were scaled and removed for

outliers.

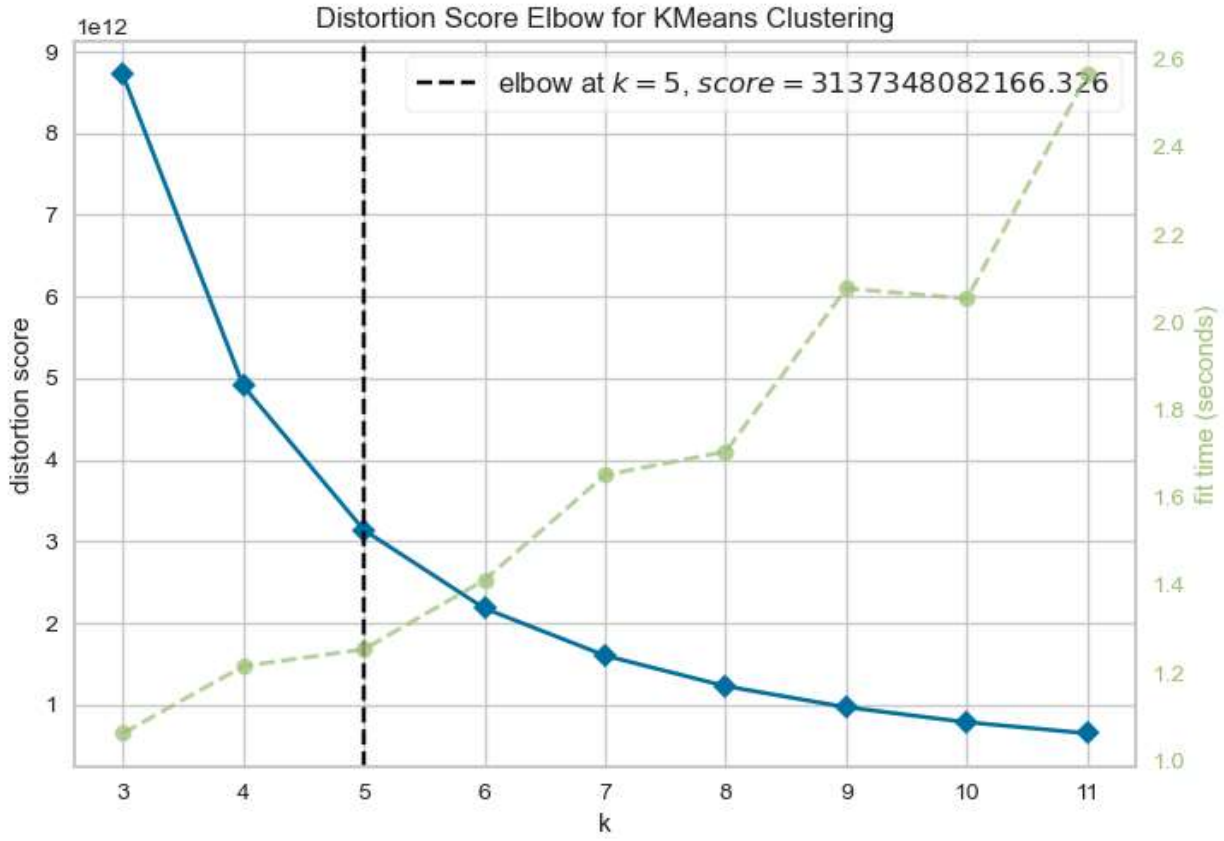
In [ ]:
```
1
```

Models

In [19]:

```
from yellowbrick.cluster import KElbowVisualizer

model = KMeans()
visualizer = KElbowVisualizer(model,n_init=10, k=(3,12))

visualizer.fit(scaled_data)
visualizer.show()
```

```
C:\Users\Sahil\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\Sahil\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\Sahil\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\Sahil\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\Sahil\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\Sahil\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\Sahil\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\Sahil\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\Sahil\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
```



Out[19]: <Axes: title={'center': 'Distortion Score Elbow for KMeans Clustering'}, xlabel='k', ylabel='distortion score'>

after k=5, the decrease starts to become more linear, suggesting that increasing the number of clusters beyond this

doesn't provide much better fitting to the data.

Another way to validate the number of clusters is by using silhouette analysis, which measures how similar an object

is to its own cluster compared to other clusters. The silhouette scores range from -1 (incorrect clustering) to

+1 (highly dense clustering). A high silhouette score indicates that the object is well matched to its own cluster and

poorly matched to neighboring clusters. We will proceed to apply K-means with K=5 and perform silhouette analysis for

further validation

In [20]:
```
km = KMeans(n_clusters=5, n_init=10, random_state=42)
clusters = km.fit_predict(scaled_data.iloc[:, 1:])

silhouette_avg = silhouette_score(scaled_data.iloc[:, 1:], clusters)

print("For n_clusters =", 5, "The average silhouette_score is :", silhouette_avg
```

```
For n_clusters = 5 The average silhouette_score is : 0.036845836183013334
```

In [33]:
```
km.inertia_
```

Out[33]:
```
2543885.163739486
```

The silhouette score = 0.03685. This suggests that the data points are on or very close to the decision boundary

between two neighboring clusters. A score near 0 indicates that clusters are overlapping. Clearly our first model is

unlikely to do very well in the prediction. We will try to submit this model and see how much ARI can we get with our

first K-means clustering model.

In [22]:

```
n_clusters = 5
init_methods = ['k-means++', 'random']
random_states = [0, 42, 100]

results = []

for init in init_methods:
    for random_state in random_states:
        kmeans = KMeans(n_clusters=n_clusters, n_init=10, init=init, random_stat
        cluster_labels = kmeans.fit_predict(scaled_data.iloc[:, 1:])

        silhouette_avg = silhouette_score(scaled_data.iloc[:, 1:], cluster_label
        print(f"For n_clusters {n_clusters}, init {init}, random state {random_s
              f"the average silhouette_score is: {silhouette_avg}")

        results.append({
            'n_clusters': n_clusters,
            'init': init,
            'random_state': random_state,
            'silhouette_score': silhouette_avg
        })

results_df = pd.DataFrame(results)

top_results = results_df.sort_values(by="silhouette_score", ascending=False).hea
print(top_results)
```

```
For n_clusters 5, init k-means++, random state 0, the average silhouette_score is:
0.0368485237365686
For n_clusters 5, init k-means++, random state 42, the average silhouette_score is:
0.036845836183013334
For n_clusters 5, init k-means++, random state 100, the average silhouette_score i
s: 0.03684767214731511
For n_clusters 5, init random, random state 0, the average silhouette_score is: 0.0
36846158830759854
For n_clusters 5, init random, random state 42, the average silhouette_score is: 0.
03684622333793698
For n_clusters 5, init random, random state 100, the average silhouette_score is:
0.03684725389650676
```

	n_clusters	init	random_state	silhouette_score
0	5	k-means++	0	0.036849
2	5	k-means++	100	0.036848
5	5	random	100	0.036847
4	5	random	42	0.036846
3	5	random	0	0.036846

The best model turn out to be the one that we had trained earlier

In [ ]:
```
1
```

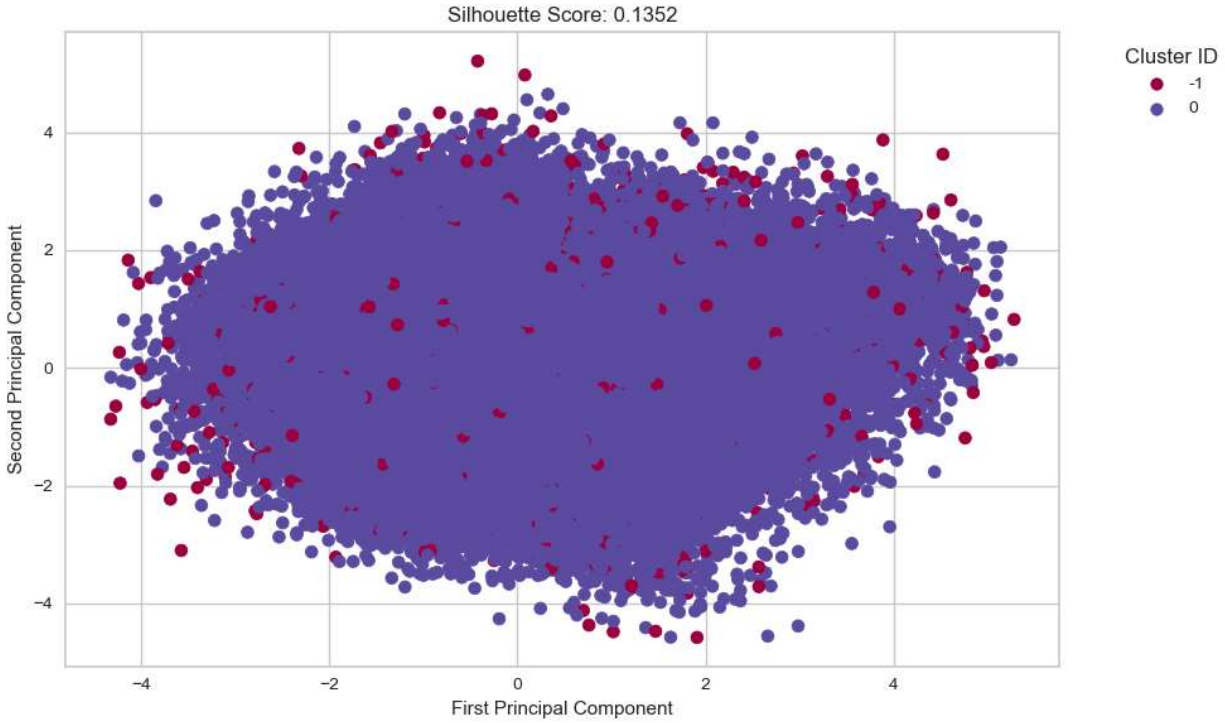
DBSCAN

In [23]:

```
eps_values = [3.9, 4.1, 4.3, 5]
min_samples_value = 500

cluster_results = []

for eps in eps_values:
    model = DBSCAN(eps=eps, min_samples=min_samples_value, metric='euclidean', n
    labels = model.fit_predict(scaled_data.iloc[:, 1:])

    unique_labels, label_counts = np.unique(labels, return_counts=True)

    cluster_results.append({
        'eps': eps,
        'number_of_clusters': len(unique_labels),
        'cluster_sizes': label_counts
    })

for result in cluster_results:
    print(f"\n### DBSCAN with eps value: {result['eps']} ###")
    print(f"Total clusters found: {result['number_of_clusters']}")
    print(f"Sizes of clusters: {result['cluster_sizes']}")
```

```
### DBSCAN with eps value: 3.9 ###
Total clusters found: 1
Sizes of clusters: [98000]

### DBSCAN with eps value: 4.1 ###
Total clusters found: 2
Sizes of clusters: [95261  2739]

### DBSCAN with eps value: 4.3 ###
Total clusters found: 2
Sizes of clusters: [74476 23524]

### DBSCAN with eps value: 5 ###
Total clusters found: 2
Sizes of clusters: [ 1787 96213]
```

In [24]:
```
pca_df = pd.DataFrame(data=principal_components, columns=[f'pca_{i}' for i in ra

pca_df["dbscan_labels"] = labels

silhouette_avg = silhouette_score(scaled_data.iloc[:, 1:], labels)

plt.figure(figsize=(10, 6))
sns.scatterplot(data=pca_df, x="pca_1", y="pca_2", hue="dbscan_labels", palette=
plt.title(f"Silhouette Score: {silhouette_avg:.4f}")
plt.xlabel("First Principal Component")
plt.ylabel("Second Principal Component")
plt.legend(title='Cluster ID', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.tight_layout()
plt.show()
```



We can see that there are not much segregation of clusters. Here, there isnt even much clusters formed (2 only).

This is likely due to no distinctive shape or region of higher density for DBSCAN to work effectively.

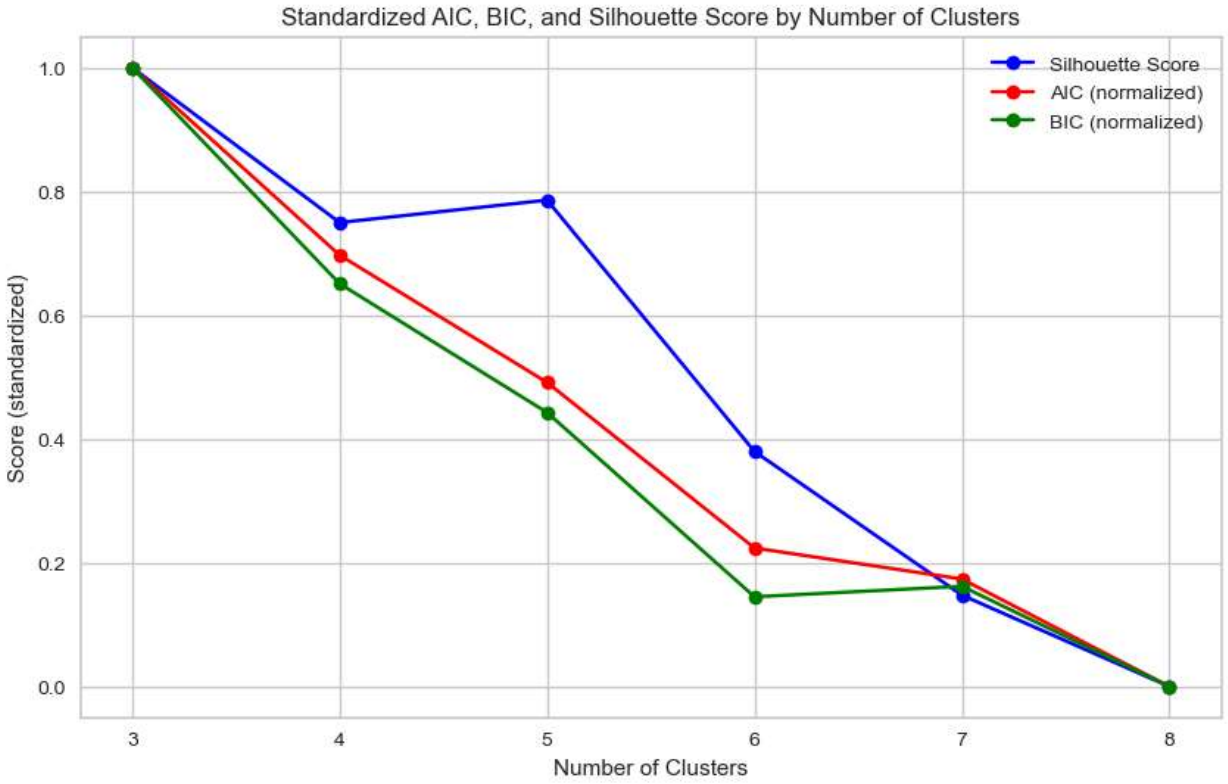
On hindsight, this might have been obvious if we have inferred from our earlier PCA analysis

In [ ]:
```
1
```

Gaussian Mixture Model

In [25]:

```
cluster_range = range(3, 9)

sil_scores = []
aic_scores = []
bic_scores = []

all_principal_components = pca.transform(scaled_data.iloc[:, 1:])

for n_clusters in cluster_range:
    gm = GaussianMixture(n_components=n_clusters, random_state=42)
    gm_preds = gm.fit_predict(all_principal_components)

    sil = silhouette_score(all_principal_components, gm_preds)
    sil_scores.append(sil)

    aic = gm.aic(all_principal_components)
    bic = gm.bic(all_principal_components)
    aic_scores.append(aic)
    bic_scores.append(bic)

max_aic, min_aic = max(aic_scores), min(aic_scores)
max_bic, min_bic = max(bic_scores), min(bic_scores)
max_sil, min_sil = max(sil_scores), min(sil_scores)

aic_scores = [(x-min_aic)/(max_aic-min_aic) for x in aic_scores]
bic_scores = [(x-min_bic)/(max_bic-min_bic) for x in bic_scores]
sil_scores = [(x-min_sil)/(max_sil-min_sil) for x in sil_scores]

plt.figure(figsize=(10, 6))
plt.plot(cluster_range, sil_scores, marker='o', label='Silhouette Score', color=
plt.plot(cluster_range, aic_scores, marker='o', label='AIC (normalized)', color=
plt.plot(cluster_range, bic_scores, marker='o', label='BIC (normalized)', color=
plt.xlabel('Number of Clusters')
plt.ylabel('Score (standardized)')
plt.title('Standardized AIC, BIC, and Silhouette Score by Number of Clusters')
plt.legend()
plt.grid(True)
plt.show()
```


Standardized AIC, BIC, and Silhouette Score by Number of Clusters
Silhouette Score
AIC (normalized)
BIC (normalized)
Score (standardized)
1.0
0.8
0.6
0.4
0.2
0.0
3
4
5
6
7
8
Number of Clusters

As seen above, the standardized plot of AIC, BIC, and silhouette score help us to visualise how different metrics can

aid in decision-making. The AIC and BIC values are typically much larger than silhouette scores, so normalization helps

in visualizing these metrics together.

Interestingly, the results with additional scoring criteria changes our perspective on number of clusters.

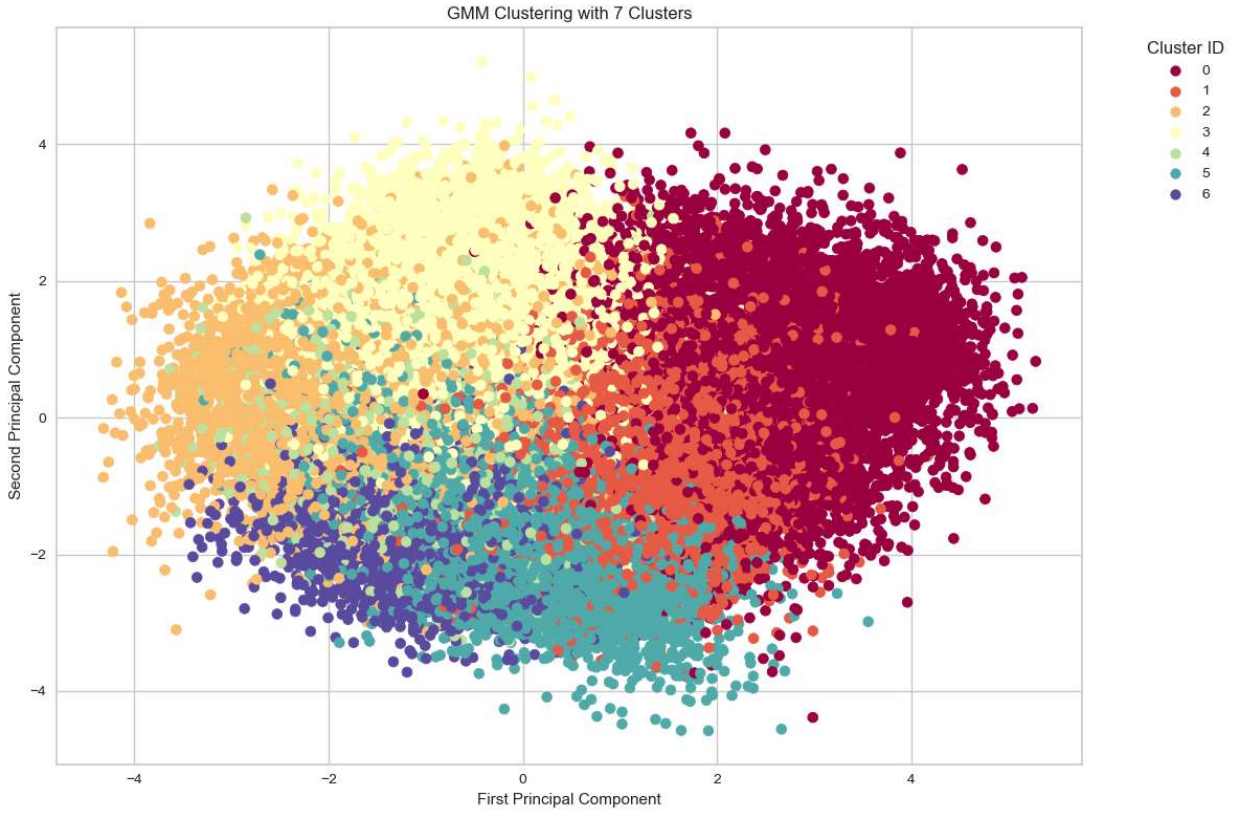
As AIC/BIC may be a better scoring measurement of GMM, we can see that the trade-off occur at n=7 when we standardised

the 3 metrics. Perhaps if we had included these metrics in our k selection in Kmeans clustering, we might have yielded

better predictions. Nonetheless, we will proceed to fit n=7

In [26]:
```
optimal_clusters = 7
gm = GaussianMixture(n_components=optimal_clusters, random_state=42)
gm.fit(all_principal_components)

labels = gm.predict(all_principal_components)

submission_df = pd.DataFrame({
    'ID': scaled_data.index,
    'Predicted': labels
})
```

In [27]:
```
pca_df = pd.DataFrame(all_principal_components, columns=[f"pca_{i+1}" for i in r
pca_df['gmm_labels'] = labels

plt.figure(figsize=(12, 8))
sns.scatterplot(data=pca_df, x="pca_1", y="pca_2", hue="gmm_labels", palette="Sp
plt.title(f"GMM Clustering with {optimal_clusters} Clusters")
plt.xlabel("First Principal Component")
plt.ylabel("Second Principal Component")
plt.legend(title='Cluster ID', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.tight_layout()
plt.show()
```



In [ ]:
```
1
```

Kmeans(n=7)

In [30]:
```
km_7 = KMeans(n_clusters=7, n_init=10, random_state=42)
clusters = km_7.fit_predict(scaled_data.iloc[:, 1:])

silhouette_avg_7 = silhouette_score(scaled_data.iloc[:, 1:], clusters)

print("For n_clusters =", 7, "The average silhouette_score is :", silhouette_avg
```

```
For n_clusters = 7 The average silhouette_score is : 0.03345035641299725
```

In [34]:
```
print(km_7.inertia_)
print(km.inertia_)
```

```
2487229.992053091
2543885.163739486
```

Here we can see that the performance of Kmeans with 7 clusters is better than 5, so it is important to consider multiple

metrics for model selection

In [ ]:
```

```