In [1]:
```
#Import Python Libraries
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt

# Load the data ("hotel_booking.csv").
df = pd.read_csv('hotel_booking.csv')
```

In [2]:
```
df.head()
```

Out[2]:

| | hotel | is_canceled | lead_time | arrival_date_year | arrival_date_month | arriva |
|---|---|---|---|---|---|---|
| **0** | Resort Hotel | 0 | 342 | 2015 | July | |
| **1** | Resort Hotel | 0 | 737 | 2015 | July | |
| **2** | Resort Hotel | 0 | 7 | 2015 | July | |
| **3** | Resort Hotel | 0 | 13 | 2015 | July | |
| **4** | Resort Hotel | 0 | 14 | 2015 | July | |

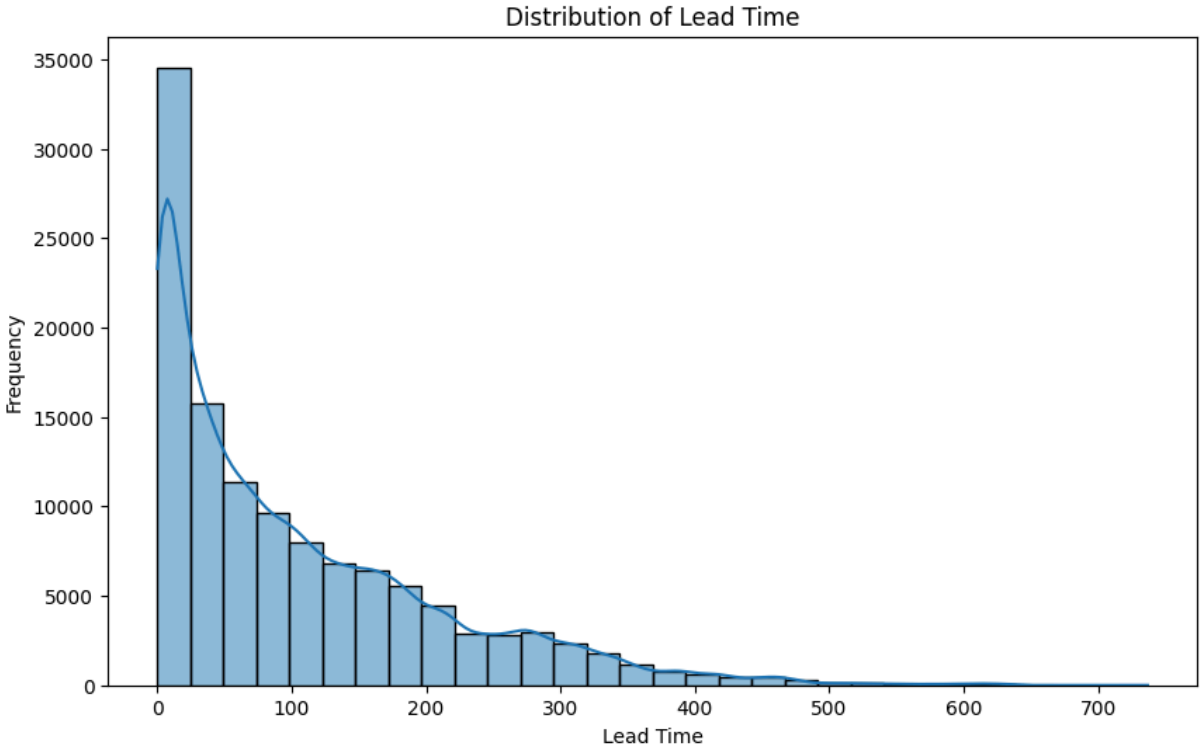
5 rows × 36 columns

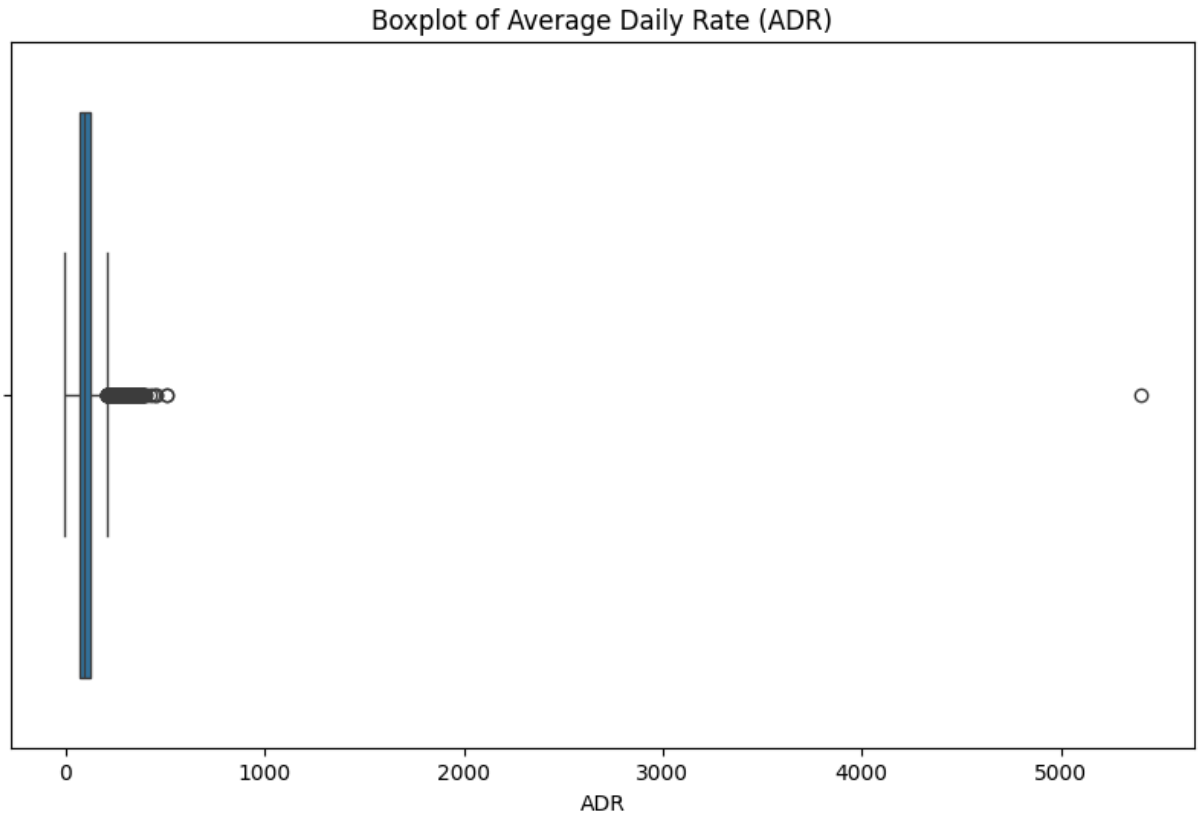
In [3]:
```
# Get statistics from the dataframe's numerical columns
df.describe()
```

Out[3]:

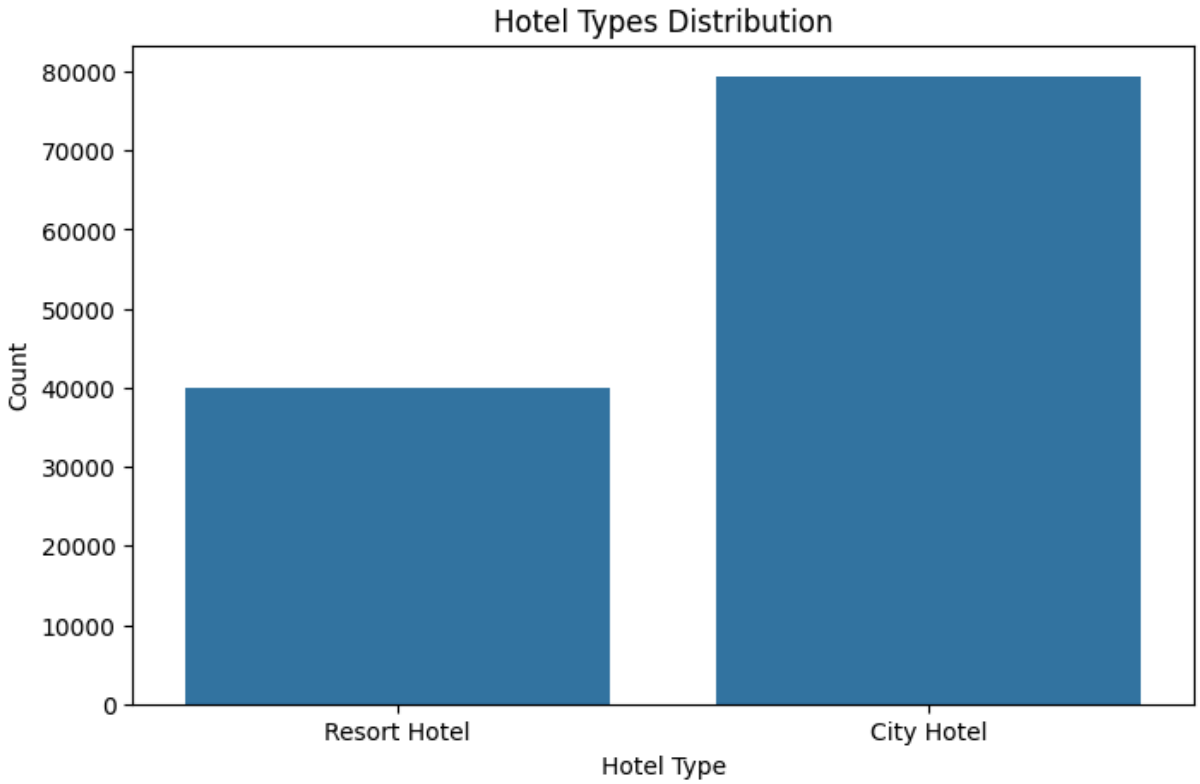
| | is_canceled | lead_time | arrival_date_year | arrival_date_week_nun |
|---|---|---|---|---|
| **count** | 119390.000000 | 119390.000000 | 119390.000000 | 119390.000 |
| **mean** | 0.370416 | 104.011416 | 2016.156554 | 27.165 |
| **std** | 0.482918 | 106.863097 | 0.707476 | 13.605 |
| **min** | 0.000000 | 0.000000 | 2015.000000 | 1.000 |
| **25%** | 0.000000 | 18.000000 | 2016.000000 | 16.000 |
| **50%** | 0.000000 | 69.000000 | 2016.000000 | 28.000 |
| **75%** | 1.000000 | 160.000000 | 2017.000000 | 38.000 |
| **max** | 1.000000 | 737.000000 | 2017.000000 | 53.000 |

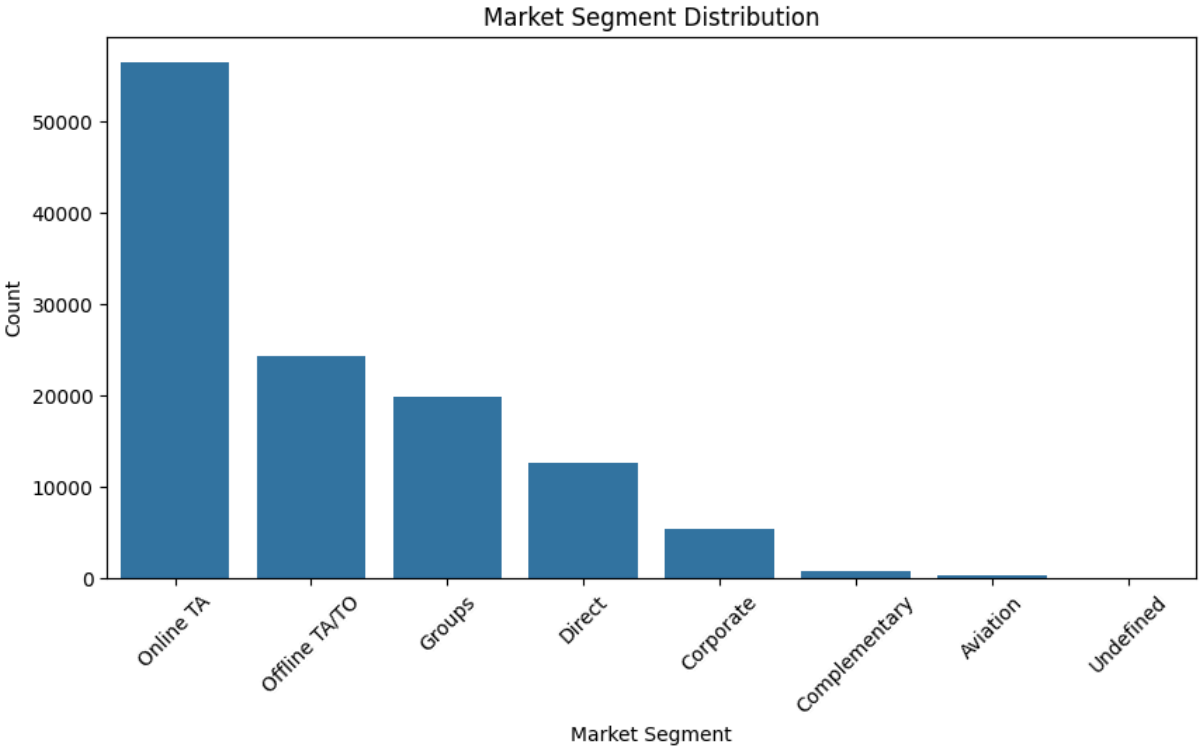
In [4]:
```
# Visualizing Numerical Features (Lead Time and ADR)
import seaborn as sns
import matplotlib.pyplot as plt

# Histogram and KDE for lead_time
plt.figure(figsize=(10, 6))
sns.histplot(df['lead_time'], kde=True, bins=30)
plt.title('Distribution of Lead Time')
plt.xlabel('Lead Time')
plt.ylabel('Frequency')
plt.show()

# Boxplot for ADR
plt.figure(figsize=(10, 6))
sns.boxplot(x=df['adr'])
plt.title('Boxplot of Average Daily Rate (ADR)')
plt.xlabel('ADR')
plt.show()
```

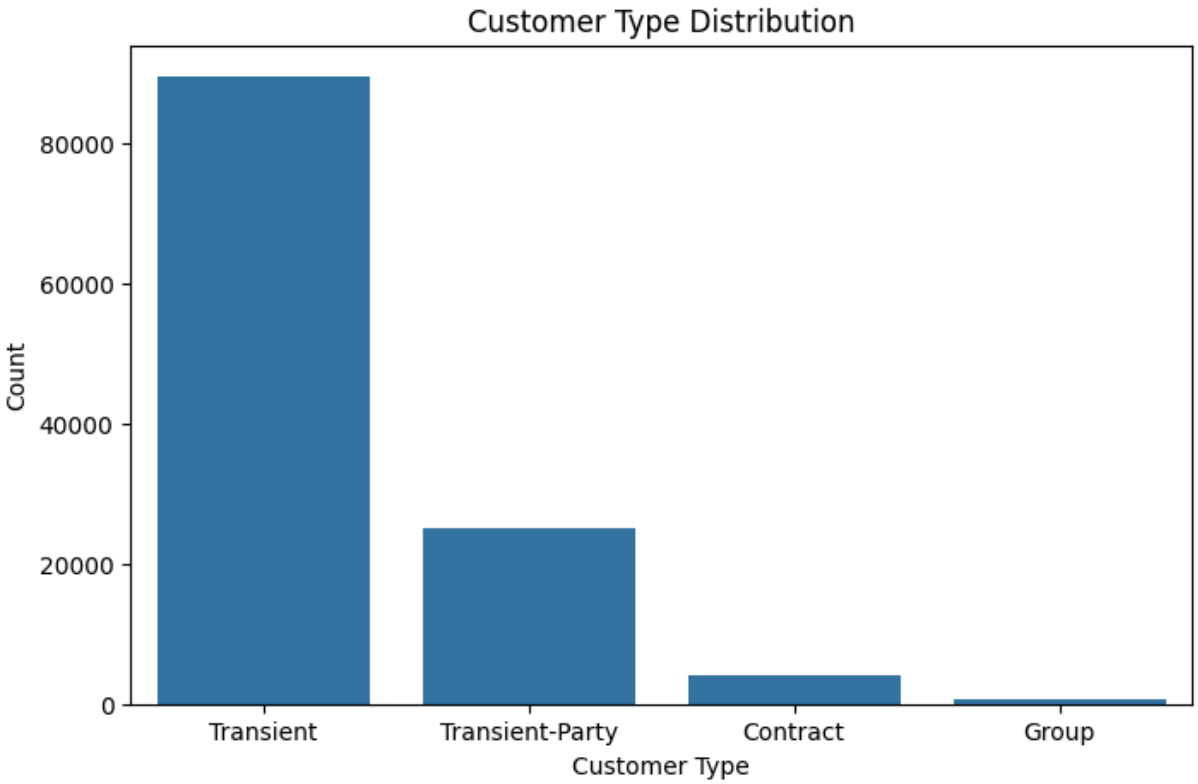





In [5]:
```
# Visualizing Categorical Features (Hotel, Market Segment, Customer Type)
# Hotel Types Distribution
plt.figure(figsize=(8, 5))
sns.countplot(data=df, x='hotel')
plt.title('Hotel Types Distribution')
plt.xlabel('Hotel Type')
plt.ylabel('Count')
plt.show()

# Market Segment Distribution
plt.figure(figsize=(10, 5))
sns.countplot(data=df, x='market_segment', order=df['market_segment'].value_
plt.title('Market Segment Distribution')
plt.xlabel('Market Segment')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()

# Customer Type Distribution
plt.figure(figsize=(8, 5))
sns.countplot(data=df, x='customer_type', order=df['customer_type'].value_co
plt.title('Customer Type Distribution')
plt.xlabel('Customer Type')
plt.ylabel('Count')
plt.show()
```


Hotel Types Distribution
Count
80000
70000
60000
50000
40000
30000
20000
10000
0
Resort Hotel
City Hotel
Hotel Type


Market Segment Distribution
Count
50000
40000
30000
20000
10000
0
Online TA
Offline TA/TO
Groups
Direct
Corporate
Complementary
Aviation
Undefined
Market Segment




- Histograms/KDE: Show the distribution of numerical variables. Lead_time, for instance, can reveal if bookings are made far in advance or close to the arrival date.
- Boxplots: Highlight outliers, especially for variables like adr, where outliers could signify unusually high or low room rates.
- Bar Plots: Show the counts of each category for categorical variables, which is helpful in understanding class imbalances or the general makeup of the data.

# Data Cleaning

In [6]:
```
# Dropping irrelevant columns
# data_cleaned = df.drop(columns=['name', 'email', 'phone-number', 'credit_c
df.drop(columns=['name', 'email', 'phone-number', 'credit_card'])
```

Out[6]:

| | hotel | is_canceled | lead_time | arrival_date_year | arrival_date_month |
|---|---|---|---|---|---|
| **0** | Resort Hotel | 0 | 342 | 2015 | July |
| **1** | Resort Hotel | 0 | 737 | 2015 | July |
| **2** | Resort Hotel | 0 | 7 | 2015 | July |
| **3** | Resort Hotel | 0 | 13 | 2015 | July |
| **4** | Resort Hotel | 0 | 14 | 2015 | July |
| **...** | ... | ... | ... | ... | ... |
| **119385** | City Hotel | 0 | 23 | 2017 | August |
| **119386** | City Hotel | 0 | 102 | 2017 | August |
| **119387** | City Hotel | 0 | 34 | 2017 | August |
| **119388** | City Hotel | 0 | 109 | 2017 | August |
| **119389** | City Hotel | 0 | 205 | 2017 | August |

119390 rows × 32 columns

# Missing Data

In [7]:
```
# Checking for missing values
df.isnull().sum()
```

Out[7]:
```
hotel                               0
is_canceled                         0
lead_time                           0
arrival_date_year                   0
arrival_date_month                  0
arrival_date_week_number            0
arrival_date_day_of_month           0
stays_in_weekend_nights             0
stays_in_week_nights                0
adults                              0
children                            4
babies                              0
meal                                0
country                           488
market_segment                      0
distribution_channel                0
is_repeated_guest                   0
previous_cancellations              0
previous_bookings_not_canceled      0
reserved_room_type                  0
assigned_room_type                  0
booking_changes                     0
deposit_type                        0
agent                           16340
company                        112593
days_in_waiting_list                0
customer_type                       0
adr                                 0
required_car_parking_spaces         0
total_of_special_requests           0
reservation_status                  0
reservation_status_date             0
name                                0
email                               0
phone-number                        0
credit_card                         0
dtype: int64
```

In [8]:
```
# Check for any categorical features that might need encoding
categorical_features = df.select_dtypes(include=['object']).columns
print("Categorical Features for Potential Encoding:", list(categorical_featu
```

```
Categorical Features for Potential Encoding: ['hotel', 'arrival_date_month',
'meal', 'country', 'market_segment', 'distribution_channel', 'reserved_room_
type', 'assigned_room_type', 'deposit_type', 'customer_type', 'reservation_s
tatus', 'reservation_status_date', 'name', 'email', 'phone-number', 'credit_
card']
```

In [9]:
```
from sklearn.preprocessing import StandardScaler

# Identifying numerical features to scale (excluding target if included)
numerical_features = df.select_dtypes(include=['float64', 'int64']).columns
print("Numerical Features for Potential Scaling:", list(numerical_features))
```

```
Numerical Features for Potential Scaling: ['is_canceled', 'lead_time', 'arri
val_date_year', 'arrival_date_week_number', 'arrival_date_day_of_month', 'st
ays_in_weekend_nights', 'stays_in_week_nights', 'adults', 'children', 'babie
s', 'is_repeated_guest', 'previous_cancellations', 'previous_bookings_not_ca
nceled', 'booking_changes', 'agent', 'company', 'days_in_waiting_list', 'ad
r', 'required_car_parking_spaces', 'total_of_special_requests']
```

In [10]:
```
# Changing agent value of "NULL" to "No Agent":
df.replace({'agent': 'NULL'}, 'No Agent', inplace=True)
```

In [11]:
```
# Changing company value of "NULL" to "No Company":
df.replace({'company': 'NULL'}, 'No Company', inplace=True)
```

There are 4 observations with missing values for the children feature. Given that those only represent 0.003% of the data, we think that it is safer not to use those observations in our analysis and modeling. As a result, the 4 observations with missing children values will be dropped.

In [12]:
```
df.dropna(subset=['children'], inplace=True)
```

The missing country values will be changed to unknown. While imputing missing values with unknown resolves the issue for primary data analysis, it does not for modeling. However, in an effort to avoid leaking information about which customers canceled their booking or not, the decision was made not to include the country feature in the model.

In [13]:
```
# Replacing null value for country feature with "unknown"
df.replace({'country': 'nan'}, 'Unknown', inplace=True)
```

In [14]:
```
# Checking that no missing values remain:
df.isnull().sum().sum()
```

Out[14]:
```
np.int64(129415)
```

# Data Types

In [15]:
```
# Checking our features' data types:
df.dtypes
```

Out[15]:
```
hotel                            object
is_canceled                       int64
lead_time                         int64
arrival_date_year                 int64
arrival_date_month               object
arrival_date_week_number          int64
arrival_date_day_of_month         int64
stays_in_weekend_nights           int64
stays_in_week_nights              int64
adults                            int64
children                        float64
babies                            int64
meal                             object
country                          object
market_segment                   object
distribution_channel             object
is_repeated_guest                 int64
previous_cancellations            int64
previous_bookings_not_canceled    int64
reserved_room_type               object
assigned_room_type               object
booking_changes                   int64
deposit_type                     object
agent                           float64
company                         float64
days_in_waiting_list              int64
customer_type                    object
adr                             float64
required_car_parking_spaces       int64
total_of_special_requests         int64
reservation_status               object
reservation_status_date          object
name                             object
email                            object
phone-number                     object
credit_card                      object
dtype: object
```

While comparing the data types to the expected data types (in the data dictionary), 2 discrepancies were found:

The children feature should be of type integer. The reservation status date feature should be of datetime type. So we need to change the type of that feature.

In [16]:
```
# Changing children to integer type:
df['children'] = df['children'].astype(int)

# Changing reservation status date to datetime type:
df['reservation_status_date'] = pd.to_datetime(df['reservation_status_date']
```

Duplicates

It is good practice to ensure our dataset does not have any duplicates.

In [17]:
```
# Checking for duplicates:
df[df.duplicated(keep='first')]
```

Out[17]:

| hotel | is_canceled | lead_time | arrival_date_year | arrival_date_month | arrival_ |
|---|---|---|---|---|---|

0 rows × 36 columns

The original dataset provides a customer's arrival date with day, month, and year each in a separate feature. For analysis purposes, it is easier to have all of those elements combined into one arrival_date_full feature.

In [18]:
```
# Creating the arrival date full feature:
df['arrival_date_full'] = df['arrival_date_year'].astype(str) + "-" + df['ar
df['arrival_date_full'] = pd.to_datetime(df['arrival_date_full'], format="%Y
```

The original dataset provides both the arrival date (date when the customer is supposed to arrive) and the reservation status date (date at which the last status change was made in the property management system). For customers who canceled their booking, the reservation status date represents the date when the booking was canceled. For customers who did not cancel their booking, the reservation status date represents the date when the guest checked-out of the hotel. As a result, calculating the difference between the arrival date and the reservation status date tells us how many days prior to supposed arrival a customer canceled their booking (for canceled reservations) OR how many days a guest stayed at the hotel (for not canceled reservations).

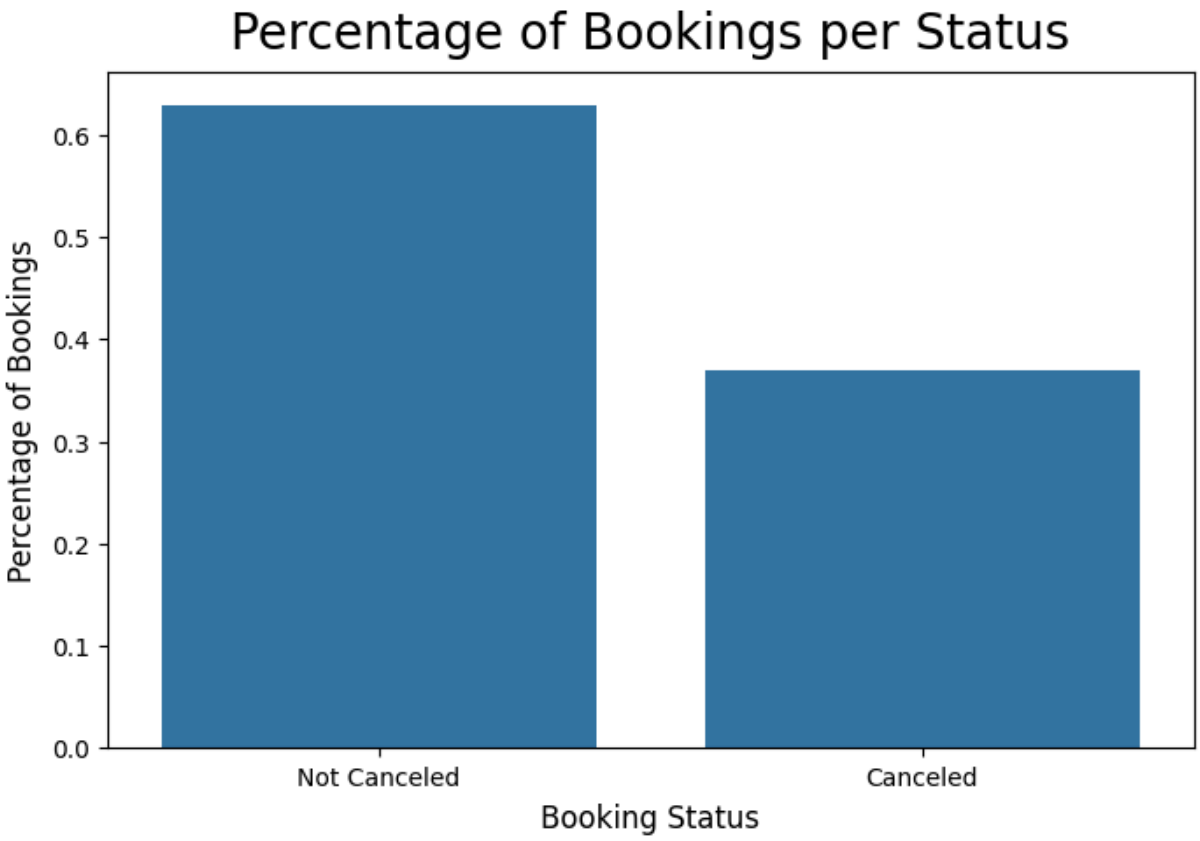
In [19]:
```
# Creating a new feature representing length of stay or how many days before
df['status_minus_arrival_date'] = np.abs(df['arrival_date_full'] - df['reser
# formating the feature
def format_lenght(date):
    return date[0]
df['status_minus_arrival_date'] = df['status_minus_arrival_date'].map(format
```

# Target Variable: Cancellation

In [20]:
```
import seaborn as sns

# Calculate the percentages of canceled vs not canceled bookings
cancel_percentages = df['is_canceled'].value_counts(normalize=True) * 100
print(f"Not Canceled: {cancel_percentages[0]:.2f}%")
print(f"Canceled: {cancel_percentages[1]:.2f}%")

# Visualizing the percentage of canceled vs not canceled bookings:
plt.figure(figsize=(8,5))
plt.title("Percentage of Bookings per Status", fontsize = 20, pad = 10)
sns.barplot(x=df['is_canceled'].unique(), y=df['is_canceled'].value_counts(r
```

```
plt.xlabel("Booking Status", fontsize = 12, labelpad = 5)
plt.ylabel("Percentage of Bookings", fontsize = 12, labelpad = 5)
plt.xticks(ticks=[0, 1], labels=['Not Canceled', 'Canceled']);
```

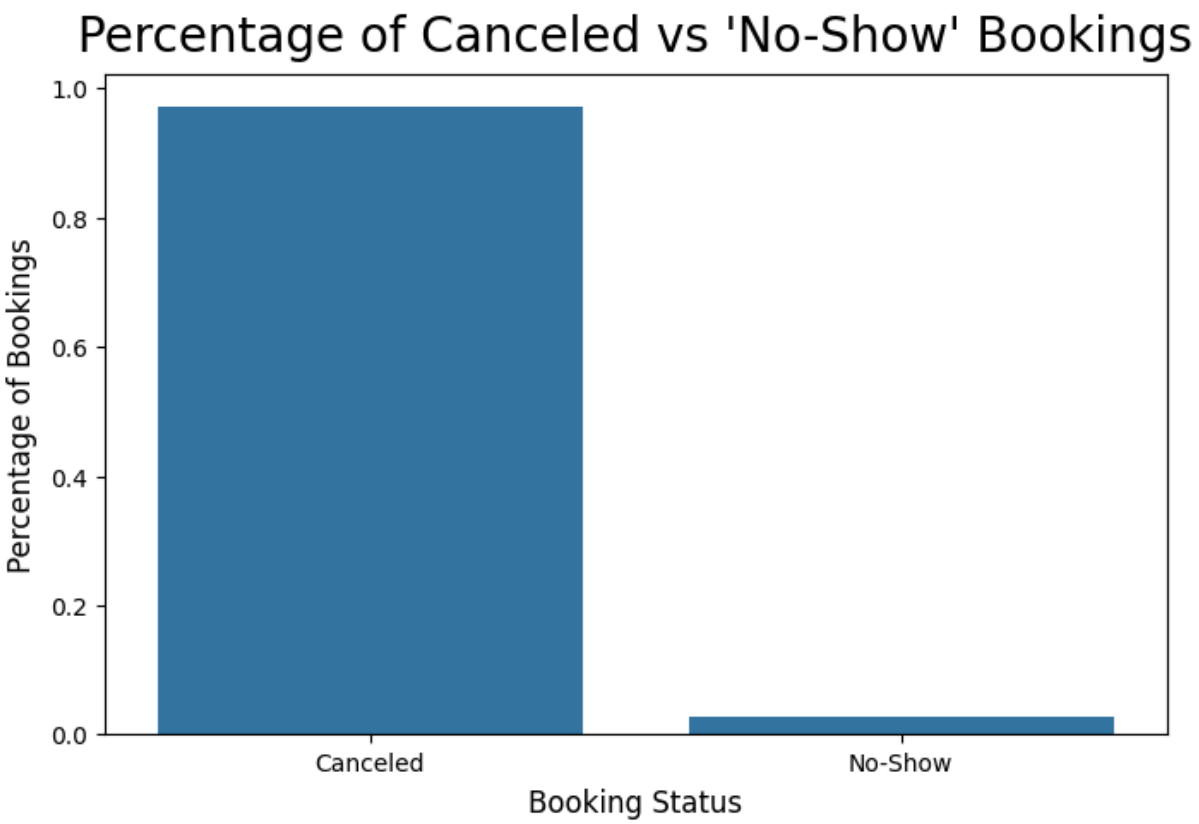
```
Not Canceled: 62.96%
Canceled: 37.04%
```




37% of bookings were canceled.

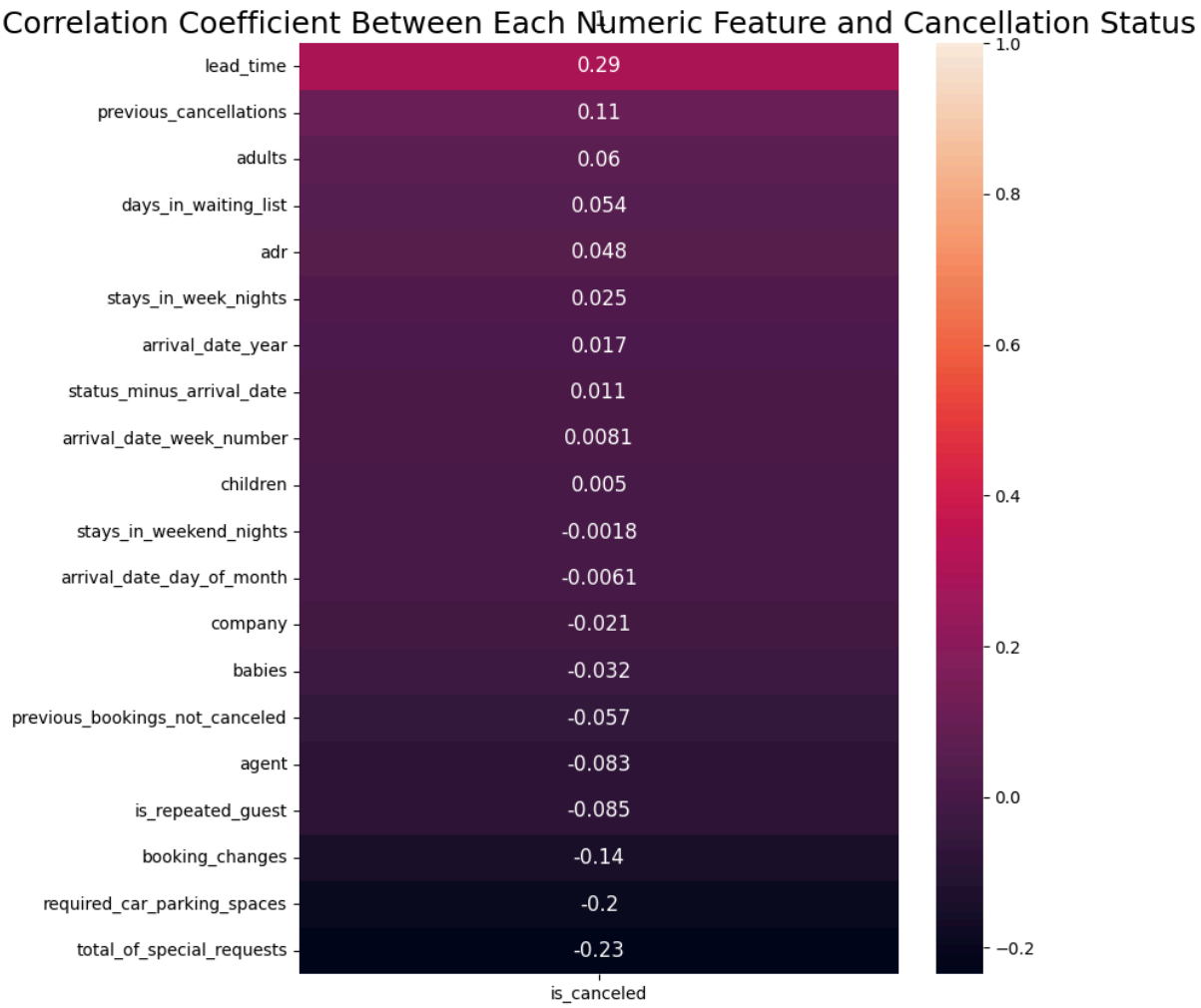
In [21]:
```
# Filter for canceled bookings
canceled_df = df[df['is_canceled'] == 1]

# Calculate the percentage of each reservation status within canceled bookin
reservation_status_percentages = canceled_df['reservation_status'].value_cou

# Print the exact percentages for 'Canceled' and 'No-Show'
for status, percentage in reservation_status_percentages.items():
    print(f"{status}: {percentage:.2f}%")

# Visualizing the percentage of bookings canceled prior to arrival:
plt.figure(figsize=(8,5))
plt.title("Percentage of Canceled vs 'No-Show' Bookings", fontsize = 20, pac
sns.barplot(x=df[df['is_canceled']==1]['reservation_status'].unique(), y=df[
plt.xlabel("Booking Status", fontsize = 12, labelpad = 5)
plt.ylabel("Percentage of Bookings", fontsize = 12, labelpad = 5);
```

```
Canceled: 97.27%
No-Show: 2.73%
```




A vast majority of bookings (97%) are canceled before the client's arrival.

In [22]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Select only numeric columns for correlation -> Error converting hotel to i
numeric_df = df.select_dtypes(include=['float64', 'int64'])

# Calculate correlation with 'is_canceled'
correlation = numeric_df.corr()[['is_canceled']].sort_values('is_canceled',

# Visualizing correlation coefficients between features and cancellation
fig = plt.figure(figsize=(8,10))
ax = sns.heatmap(correlation, annot=True, annot_kws={"size":12})
ax.set_title('Correlation Coefficient Between Each Numeric Feature and Cance

y_min, y_max = ax.get_ylim()
ax.set_ylim(top=y_max+1);
```




Lead time has the strongest correlation with booking cancellations. This makes sense because as the number of days between making the booking and the planned arrival increases, customers have more opportunity to cancel and the chance of unexpected events affecting travel plans also rises.

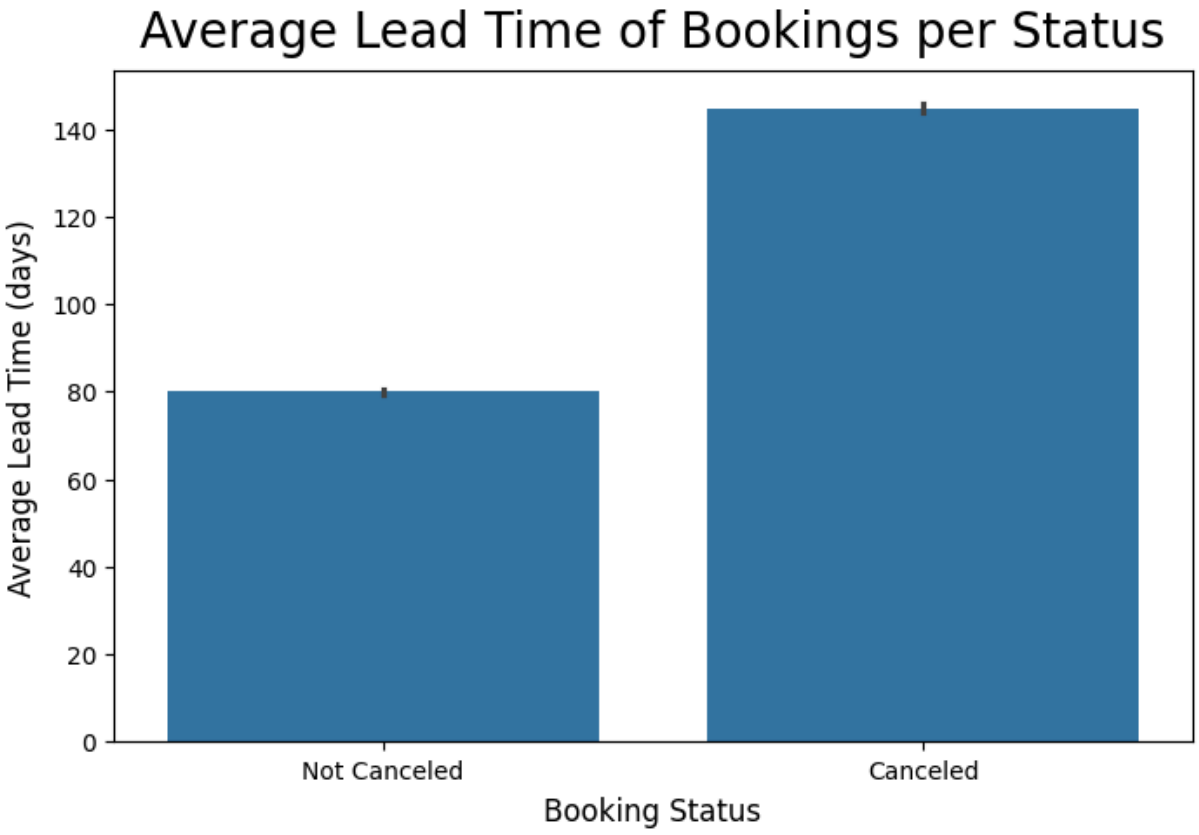
Interestingly, the total number of special requests ranks second in correlation to cancellations. The more special requests a customer makes, the less likely they are to cancel. This implies that engagement with the hotel and feeling that their needs are being met could reduce the likelihood of a cancellation.

Similarly, the number of requested parking spaces is the third most correlated feature. As parking requests increase, the chances of cancellation decrease, and potential explanations for this trend will be explored later.

It's worth noting that a customer’s previous history with the hotel (e.g., number of previous bookings not canceled or repeat guest status) does not strongly correlate with current booking cancellations. However, a history of previous cancellations shows a stronger correlation with the likelihood of canceling the current booking.

In [23]:
```
# Calculate average lead time
average_lead_time = df.groupby('is_canceled')['lead_time'].mean()
print(f"Average Lead Time: {average_lead_time}")

# Visualizing the average lead time for canceled and not canceled bookings:
plt.figure(figsize=(8,5))
plt.title("Average Lead Time of Bookings per Status", fontsize = 20, pad = 1
sns.barplot(x=df['is_canceled'], y=df['lead_time'])
plt.xlabel("Booking Status", fontsize = 12, labelpad = 5)
plt.ylabel("Average Lead Time (days)", fontsize = 12, labelpad = 5)
plt.xticks(ticks=[0, 1], labels=['Not Canceled', 'Canceled']);
```

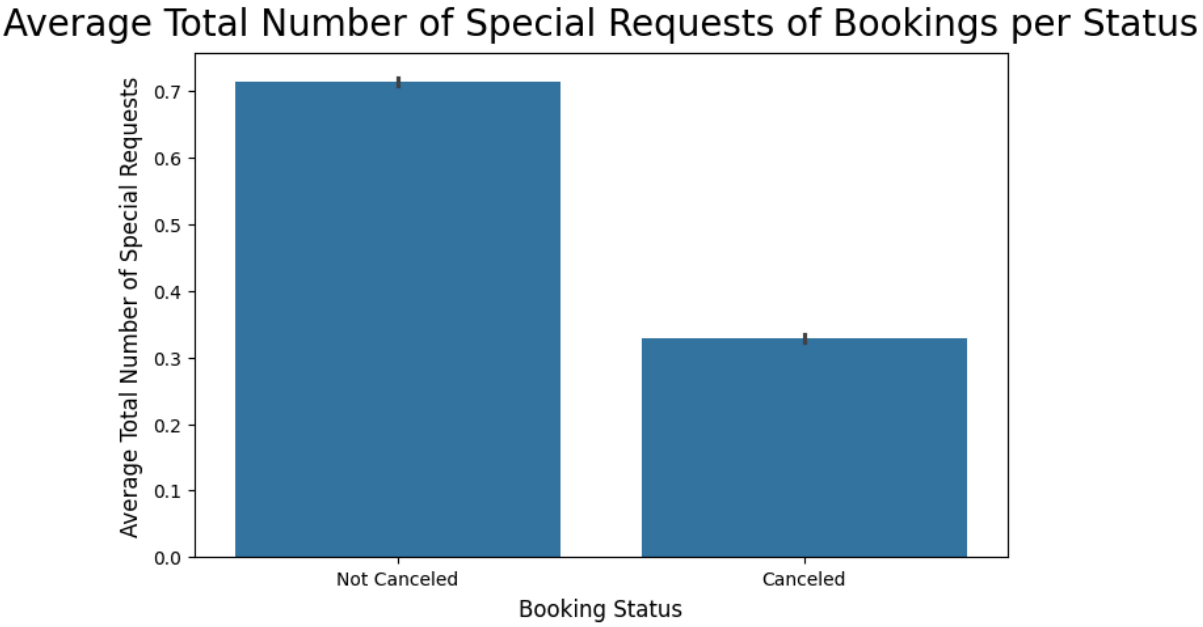
```
Average Lead Time: is_canceled
0     79.984687
1    144.861646
Name: lead_time, dtype: float64
```




Canceled bookings have a longer lead time on average.

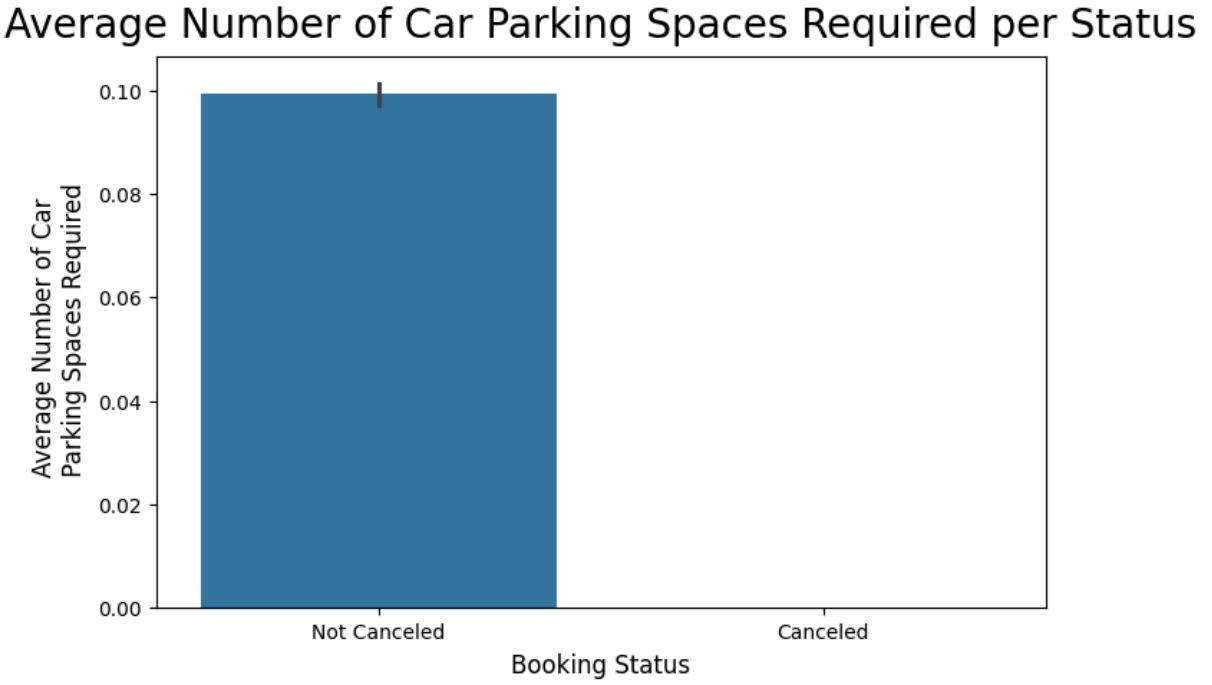
In [24]:
```
# Calculate average total number of special requests
average_special_requests = df.groupby('is_canceled')['total_of_special_reque
print(f"Average Special Requests: {average_special_requests}")

# Visualizing the total number of special requests for canceled and not canc
plt.figure(figsize=(8,5))
plt.title("Average Total Number of Special Requests of Bookings per Status",
sns.barplot(x=df['is_canceled'], y=df['total_of_special_requests'])
plt.xlabel("Booking Status", fontsize = 12, labelpad = 5)
plt.ylabel("Average Total Number of Special Requests", fontsize = 12, labelp
plt.xticks(ticks=[0, 1], labels=['Not Canceled', 'Canceled']);
```

```
Average Special Requests: is_canceled
0    0.714060
1    0.328743
Name: total_of_special_requests, dtype: float64
```




Customers who cancel their bookings make on average fewer special requests.

In [25]:
```
# Visualizing the total number of requested parking spaces for canceled and
plt.figure(figsize=(8,5))
plt.title("Average Number of Car Parking Spaces Required per Status", fontsi
sns.barplot(x=df['is_canceled'], y=df['required_car_parking_spaces'])
plt.xlabel("Booking Status", fontsize = 12, labelpad = 5)
plt.ylabel("Average Number of Car \n Parking Spaces Required", fontsize = 12
plt.xticks(ticks=[0, 1], labels=['Not Canceled', 'Canceled']);
```

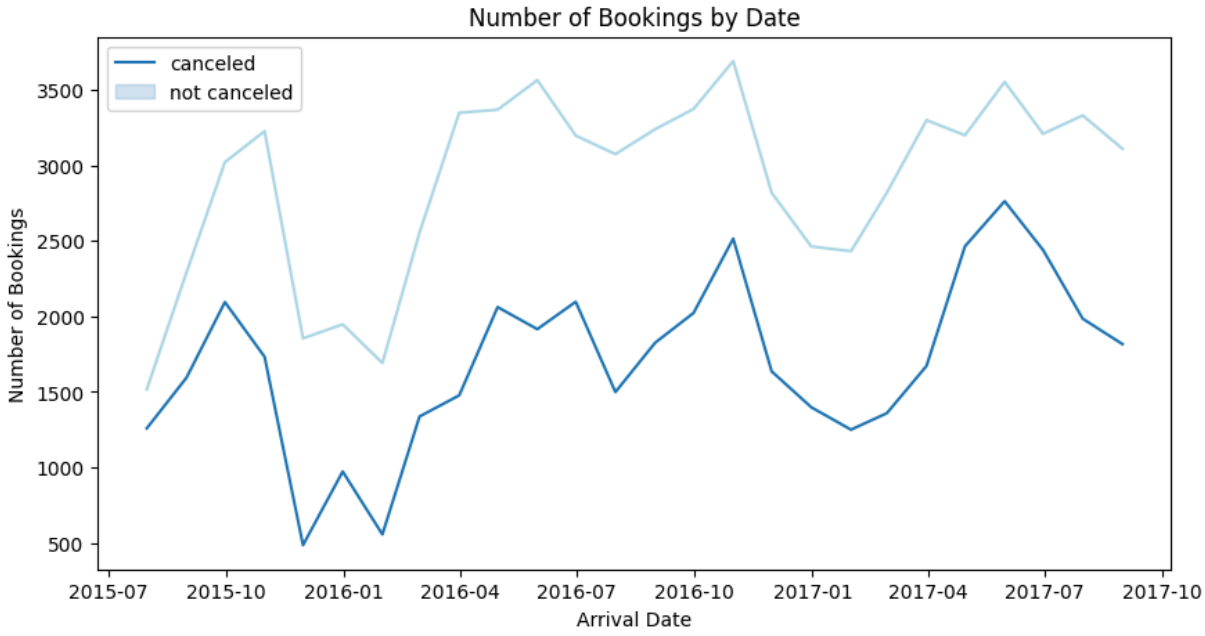


On average, customers who don't cancel their bookings tend to request more parking spaces. Similar to special requests, the more a customer interacts with the hotel (such as requesting a parking spot), the less likely they are to cancel. It's reasonable to assume that once a guest is considering parking logistics, they are already fairly committed to their trip.

From the hotel's perspective, it's possible that parking availability is limited in the area. This would mean that guests requiring parking may have fewer hotel options, making them less likely to cancel. While more information from the hotel is needed to confirm this, if true, it suggests that increasing parking availability could help lower cancellation rates.

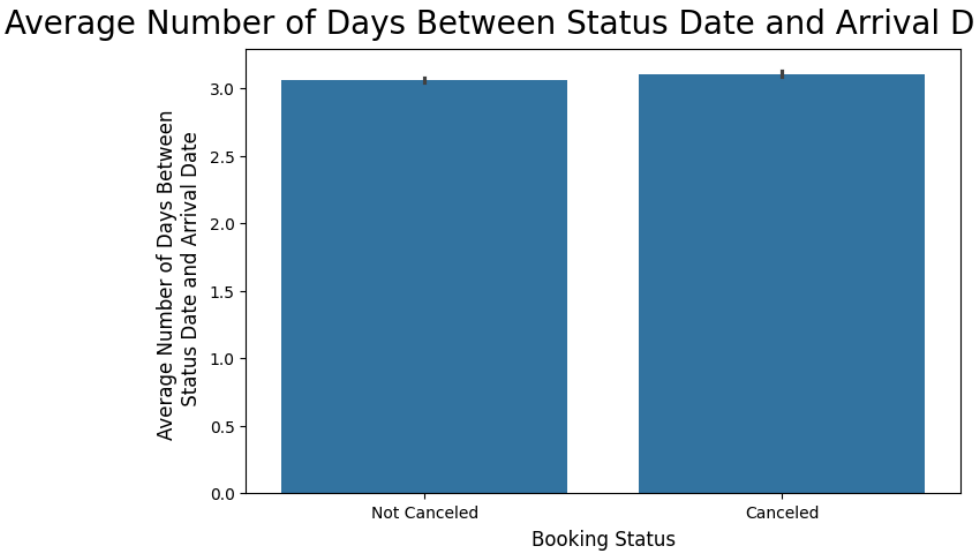
In [26]:
```
# Dataframe of canceled bookings for plotting purposes
canceled = df[df['is_canceled']==1][['arrival_date_full']]
canceled.set_index('arrival_date_full', inplace=True)
canceled['count'] = 1
canceled=canceled.resample('ME').sum()

# Dataframe of not canceled bookings for plotting purposes
not_canceled = df[df['is_canceled']==0][['arrival_date_full']]
not_canceled.set_index('arrival_date_full', inplace=True)
not_canceled['count'] = 1
not_canceled=not_canceled.resample('ME').sum()
```

In [27]:
```
# Visualizing the number of canceled and not canceled bookings by date:
plt.figure(figsize=(10,5))
sns.lineplot(x=canceled.index, y=canceled['count'])
sns.lineplot(x=not_canceled.index, y=not_canceled['count'], color='lightblue
plt.ylabel('Number of Bookings')
plt.xlabel('Arrival Date')
plt.title('Number of Bookings by Date')
plt.legend(['canceled', 'not canceled']);
```



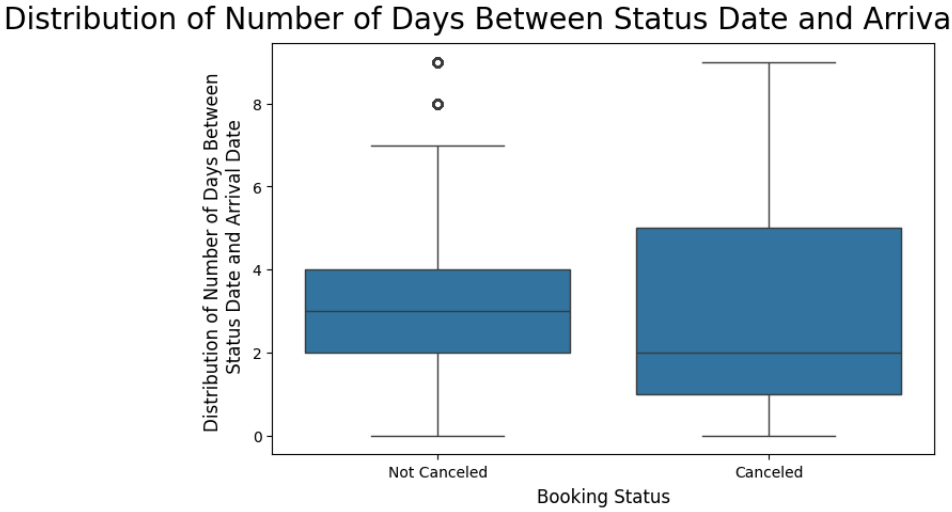
The same pattern can be seen in both canceled and non-canceled bookings. Less bookings are canceled around January. More bookings are canceled in the warmer months between April and July.

In [28]:
```
# Visualizing the average number of days between status date and arrival dat
plt.figure(figsize=(8,5))
plt.title("Average Number of Days Between Status Date and Arrival Date per S
sns.barplot(x=df['is_canceled'], y=df['status_minus_arrival_date'])
plt.xlabel("Booking Status", fontsize = 12, labelpad = 5)
plt.ylabel("Average Number of Days Between \n Status Date and Arrival Date",
plt.xticks(ticks=[0, 1], labels=['Not Canceled', 'Canceled']);
```



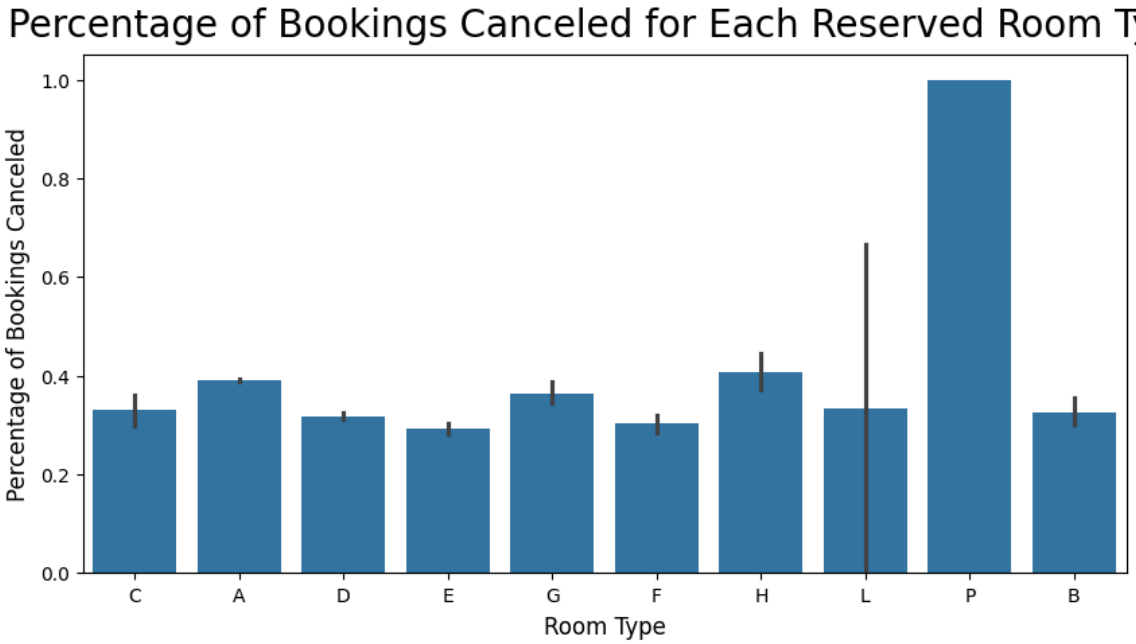

On average customers spend 3 nights in the hotel. On average, customers cancel 3 days before their supposed arrival date. This doesn't give hotels a lot of time to find a new guest or adjust their operations.

In [29]:
```
# Visualizing the distribution of number of days between status date and arr
plt.figure(figsize=(8,5))
plt.title("Distribution of Number of Days Between Status Date and Arrival Da
sns.boxplot(x=df['is_canceled'], y=df['status_minus_arrival_date'])
plt.xlabel("Booking Status", fontsize = 12, labelpad = 5)
plt.ylabel("Distribution of Number of Days Between \n Status Date and Arriva
plt.xticks(ticks=[0, 1], labels=['Not Canceled', 'Canceled']);
```



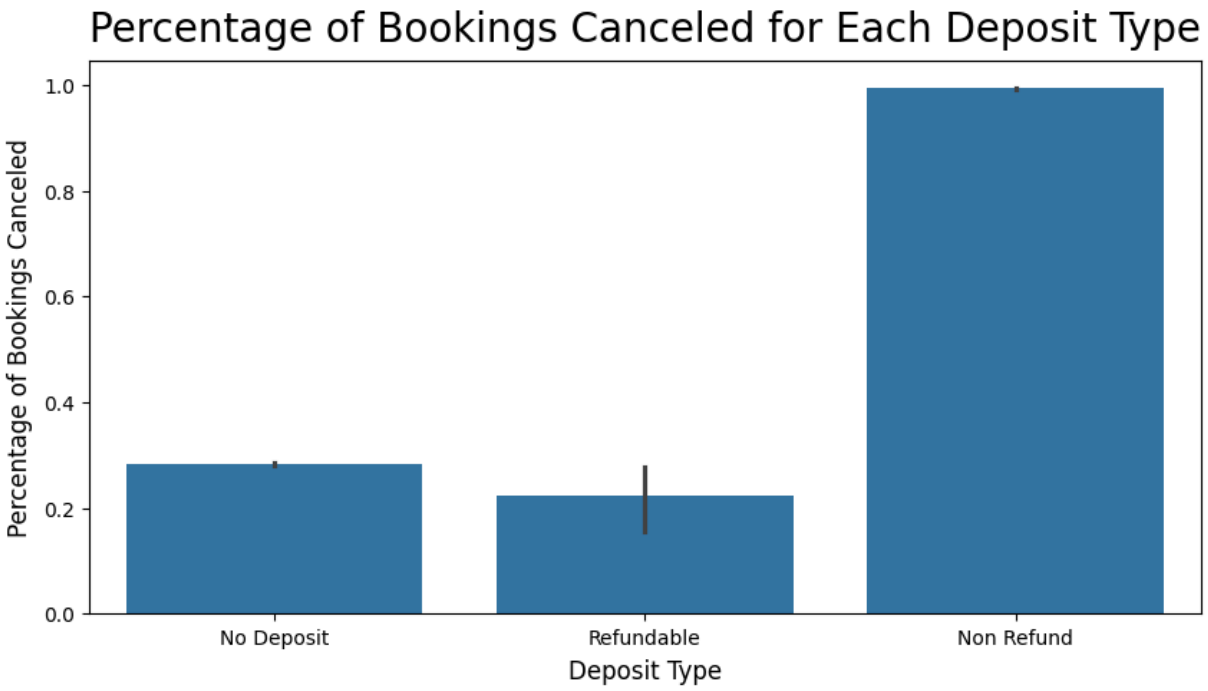

There seems to be more variation in how many days prior to arrival customers cancel their bookings compared to the number of days customers stay at a hotel. We notice 2 outliers in the length of stay, but there'n no reason to believe that stays over 8 days are not valid datapoints. As a result, we will keep those outliers in our model.

In [30]:
```
# Visualizing percentage of bookings canceled for each room type:
plt.figure(figsize=(10,5))
plt.title("Percentage of Bookings Canceled for Each Reserved Room Type", for
sns.barplot(x=df['reserved_room_type'], y=df['is_canceled'])
plt.xlabel("Room Type", fontsize = 12, labelpad = 5)
plt.ylabel("Percentage of Bookings Canceled", fontsize = 12, labelpad = 5);
```




Customers who reserved room type P have the highest percentage booking cancellation with 100% of bookings canceled.

In [31]:
```
# Visualizing percentage of bookings canceled for each deposit type:
plt.figure(figsize=(10,5))
plt.title("Percentage of Bookings Canceled for Each Deposit Type", fontsize
sns.barplot(x=df['deposit_type'], y=df['is_canceled'])
plt.xlabel("Deposit Type", fontsize = 12, labelpad = 5)
plt.ylabel("Percentage of Bookings Canceled", fontsize = 12, labelpad = 5);
```




Surprisingly, customers who pay a non-refundable deposit have a much higher percentage of canceled reservations. As this is a counter-intuitive finding, it is necessary to dig a little deeper into the characteristics of bookings with a non-refundable deposit.

In [32]:
```
df.groupby(df['deposit_type']).describe(include='all')
```

Out[32]:

| | hote | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | count | unique | top | freq | mean | min | 25% | 50% | 75% | ma |
| deposit_type | | | | | | | | | | |
| **No Deposit** | 104637 | 2 | City Hotel | 66438 | NaN | NaN | NaN | NaN | NaN | Nal |
| **Non Refund** | 14587 | 2 | City Hotel | 12868 | NaN | NaN | NaN | NaN | NaN | Nal |
| **Refundable** | 162 | 2 | Resort Hotel | 142 | NaN | NaN | NaN | NaN | NaN | Nal |

3 rows × 407 columns

Non-refundable deposits appear to be more commonly made by transient groups who book through a travel agent. This suggests that hotels may have policies

requiring certain groups to pay a non-refundable deposit, possibly viewing them as "high risk" and more likely to cancel. However, to confirm this theory, more information about the hotels' deposit policies would be necessary.

# Full Dataset

In [33]:
```
df.describe()
```

Out[33]:

| | is_canceled | lead_time | arrival_date_year | arrival_date_week_nun |
|---|---|---|---|---|
| **count** | 119386.000000 | 119386.000000 | 119386.000000 | 119386.000 |
| **mean** | 0.370395 | 104.014801 | 2016.156593 | 27.165 |
| **min** | 0.000000 | 0.000000 | 2015.000000 | 1.000 |
| **25%** | 0.000000 | 18.000000 | 2016.000000 | 16.000 |
| **50%** | 0.000000 | 69.000000 | 2016.000000 | 28.000 |
| **75%** | 1.000000 | 160.000000 | 2017.000000 | 38.000 |
| **max** | 1.000000 | 737.000000 | 2017.000000 | 53.000 |
| **std** | 0.482913 | 106.863286 | 0.707456 | 13.605 |

8 rows × 23 columns

The adr feature (average daily rate) shows some outliers, with a minimum of -6.38 and a maximum of 5400. A negative ADR might occur if a hotel had to compensate a guest for some reason. While these values are unusual, we don’t have enough information to confirm whether these observations are inaccurate or legitimate data points.

In [34]:
```
# Check the distinct values of the attributes to see if there are any correc
for col in df.columns:
    print(f"{col}: {df[col].nunique()} unique values")
    print(df[col].unique())
    print("----")
```

```
hotel: 2 unique values
['Resort Hotel' 'City Hotel']
----
is_canceled: 2 unique values
[0 1]
----
lead_time: 479 unique values
[342 737   7  13  14   0   9  85  75  23  35  68  18  37  12  72 127  78
  48  60  77  99 118  95  96  69  45  40  15  36  43  70  16 107  47 113
  90  50  93  76   3   1  10   5  17  51  71  63  62 101   2  81 368 364
 324  79  21 109 102   4  98  92  26  73 115  86  52  29  30  33  32   8
 100  44  80  97  64  39  34  27  82  94 110 111  84  66 104  28 258 112
  65  67  55  88  54 292  83 105 280 394  24 103 366 249  22  91  11 108
 106  31  87  41 304 117  59  53  58 116  42 321  38  56  49 317   6  57
  19  25 315 123  46  89  61 312 299 130  74 298 119  20 286 136 129 124
 327 131 460 140 114 139 122 137 126 120 128 135 150 143 151 132 125 157
 147 138 156 164 346 159 160 161 333 381 149 154 297 163 314 155 323 340
 356 142 328 144 336 248 302 175 344 382 146 170 166 338 167 310 148 165
 172 171 145 121 178 305 173 152 354 347 158 185 349 183 352 177 200 192
 361 207 174 330 134 350 334 283 153 197 133 241 193 235 194 261 260 216
 169 209 238 215 141 189 187 223 284 214 202 211 168 230 203 188 232 709
 219 162 196 190 259 228 176 250 201 186 199 180 206 205 224 222 182 210
 275 212 229 218 208 191 181 179 246 255 226 288 253 252 262 236 256 234
 254 468 213 237 198 195 239 263 265 274 217 220 307 221 233 257 227 276
 225 264 311 277 204 290 266 270 294 319 282 251 322 291 269 240 271 184
 231 268 247 273 300 301 267 244 306 293 309 272 242 295 285 243 308 398
 303 245 424 279 331 281 339 434 357 325 329 278 332 343 345 360 348 367
 353 373 374 406 400 326 379 399 316 341 320 385 355 363 358 296 422 390
 335 370 376 375 397 289 542 403 383 384 359 393 337 362 365 435 386 378
 313 351 287 471 462 411 450 318 372 371 454 532 445 389 388 407 443 437
 451 391 405 412 419 420 426 433 440 429 418 447 461 605 457 475 464 482
 626 489 496 503 510 517 524 531 538 545 552 559 566 573 580 587 594 601
 608 615 622 629 396 410 395 423 408 409 448 465 387 414 476 479 467 490
 493 478 504 507 458 518 521 377 444 380 463]
----
arrival_date_year: 3 unique values
[2015 2016 2017]
----
arrival_date_month: 12 unique values
['July' 'August' 'September' 'October' 'November' 'December' 'January'
 'February' 'March' 'April' 'May' 'June']
----
arrival_date_week_number: 53 unique values
[27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
 51 52 53  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
 22 23 24 25 26]
----
arrival_date_day_of_month: 31 unique values
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31]
----
stays_in_weekend_nights: 17 unique values
[ 0  1  2  4  3  6 13  8  5  7 12  9 16 18 19 10 14]
----
stays_in_week_nights: 35 unique values
[ 0  1  2  3  4  5 10 11  8  6  7 15  9 12 33 20 14 16 21 13 30 19 24 40
```

```
 22 42 50 25 17 32 26 18 34 35 41]
----
adults: 14 unique values
[ 2  1  3  4 40 26 50 27 55  0 20  6  5 10]
----
children: 5 unique values
[ 0  1  2 10  3]
----
babies: 5 unique values
[ 0  1  2 10  9]
----
meal: 5 unique values
['BB' 'FB' 'HB' 'SC' 'Undefined']
----
country: 177 unique values
['PRT' 'GBR' 'USA' 'ESP' 'IRL' 'FRA' nan 'ROU' 'NOR' 'OMN' 'ARG' 'POL'
 'DEU' 'BEL' 'CHE' 'CN' 'GRC' 'ITA' 'NLD' 'DNK' 'RUS' 'SWE' 'AUS' 'EST'
 'CZE' 'BRA' 'FIN' 'MOZ' 'BWA' 'LUX' 'SVN' 'ALB' 'IND' 'CHN' 'MEX' 'MAR'
 'UKR' 'SMR' 'LVA' 'PRI' 'SRB' 'CHL' 'AUT' 'BLR' 'LTU' 'TUR' 'ZAF' 'AGO'
 'ISR' 'CYM' 'ZMB' 'CPV' 'ZWE' 'DZA' 'KOR' 'CRI' 'HUN' 'ARE' 'TUN' 'JAM'
 'HRV' 'HKG' 'IRN' 'GEO' 'AND' 'GIB' 'URY' 'JEY' 'CAF' 'CYP' 'COL' 'GGY'
 'KWT' 'NGA' 'MDV' 'VEN' 'SVK' 'FJI' 'KAZ' 'PAK' 'IDN' 'LBN' 'PHL' 'SEN'
 'SYC' 'AZE' 'BHR' 'NZL' 'THA' 'DOM' 'MKD' 'MYS' 'ARM' 'JPN' 'LKA' 'CUB'
 'CMR' 'BIH' 'MUS' 'COM' 'SUR' 'UGA' 'BGR' 'CIV' 'JOR' 'SYR' 'SGP' 'BDI'
 'SAU' 'VNM' 'PLW' 'QAT' 'EGY' 'PER' 'MLT' 'MWI' 'ECU' 'MDG' 'ISL' 'UZB'
 'NPL' 'BHS' 'MAC' 'TGO' 'TWN' 'DJI' 'STP' 'KNA' 'ETH' 'IRQ' 'HND' 'RWA'
 'KHM' 'MCO' 'BGD' 'IMN' 'TJK' 'NIC' 'BEN' 'VGB' 'TZA' 'GAB' 'GHA' 'TMP'
 'GLP' 'KEN' 'LIE' 'GNB' 'MNE' 'UMI' 'MYT' 'FRO' 'MMR' 'PAN' 'BFA' 'LBY'
 'MLI' 'NAM' 'BOL' 'PRY' 'BRB' 'ABW' 'AIA' 'SLV' 'DMA' 'PYF' 'GUY' 'LCA'
 'ATA' 'GTM' 'ASM' 'MRT' 'NCL' 'KIR' 'SDN' 'ATF' 'SLE' 'LAO']
----
market_segment: 7 unique values
['Direct' 'Corporate' 'Online TA' 'Offline TA/TO' 'Complementary' 'Groups'
 'Aviation']
----
distribution_channel: 5 unique values
['Direct' 'Corporate' 'TA/TO' 'Undefined' 'GDS']
----
is_repeated_guest: 2 unique values
[0 1]
----
previous_cancellations: 15 unique values
[ 0  1  2  3 26 25 14  4 24 19  5 21  6 13 11]
----
previous_bookings_not_canceled: 73 unique values
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 20 21 22 23 24
 25 27 28 29 30 19 26 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72]
----
reserved_room_type: 10 unique values
['C' 'A' 'D' 'E' 'G' 'F' 'H' 'L' 'P' 'B']
----
assigned_room_type: 12 unique values
['C' 'A' 'D' 'E' 'G' 'F' 'I' 'B' 'H' 'P' 'L' 'K']
----
```

```
booking_changes: 21 unique values
[ 3  4  0  1  2  5 17  6  8  7 10 16  9 13 12 20 14 15 11 21 18]
----
deposit_type: 3 unique values
['No Deposit' 'Refundable' 'Non Refund']
----
agent: 333 unique values
[ nan 304. 240. 303.  15. 241.   8. 250. 115.   5. 175. 134. 156. 243.
 242.   3. 105.  40. 147. 306. 184.  96.   2. 127.  95. 146.   9. 177.
   6. 143. 244. 149. 167. 300. 171. 305.  67. 196. 152. 142. 261. 104.
  36.  26.  29. 258. 110.  71. 181.  88. 251. 275.  69. 248. 208. 256.
 314. 126. 281. 273. 253. 185. 330. 334. 328. 326. 321. 324. 313.  38.
 155.  68. 335. 308. 332.  94. 348. 310. 339. 375.  66. 327. 387. 298.
  91. 245. 385. 257. 393. 168. 405. 249. 315.  75. 128. 307.  11. 436.
   1. 201. 183. 223. 368. 336. 291. 464. 411. 481.  10. 154. 468. 410.
 390. 440. 495. 492. 493. 434.  57. 531. 420. 483. 526. 472. 429.  16.
 446.  34.  78. 139. 252. 270.  47. 114. 301. 193. 182. 135. 350. 195.
 352. 355. 159. 363. 384. 360. 331. 367.  64. 406. 163. 414. 333. 427.
 431. 430. 426. 438. 433. 418. 441. 282. 432.  72. 450. 180. 454. 455.
  59. 451. 254. 358. 469. 165. 467. 510. 337. 476. 502. 527. 479. 508.
 535. 302. 497. 187.  13.   7.  27.  14.  22.  17.  28.  42.  20.  19.
  45.  37.  61.  39.  21.  24.  41.  50.  30.  54.  52.  12.  44.  31.
  83.  32.  63.  60.  55.  56.  89.  87. 118.  86.  85. 210. 214. 129.
 179. 138. 174. 170. 153.  93. 151. 119.  35. 173.  58.  53. 133.  79.
 235. 192. 191. 236. 162. 215. 157. 287. 132. 234.  98.  77. 103. 107.
 262. 220. 121. 205. 378.  23. 296. 290. 229.  33. 286. 276. 425. 484.
 323. 403. 219. 394. 509. 111. 423.   4.  70.  82.  81.  74.  92.  99.
  90. 112. 117. 106. 148. 158. 144. 211. 213. 216. 232. 150. 267. 227.
 247. 278. 280. 285. 289. 269. 295. 265. 288. 122. 294. 325. 341. 344.
 346. 359. 283. 364. 370. 371.  25. 141. 391. 397. 416. 404. 299. 197.
  73. 354. 444. 408. 461. 388. 453. 459. 474. 475. 480. 449.]
----
company: 352 unique values
[ nan 110. 113. 270. 178. 240. 154. 144. 307. 268.  59. 204. 312. 318.
  94. 174. 274. 195. 223. 317. 281. 118.  53. 286.  12.  47. 324. 342.
 373. 371. 383.  86.  82. 218.  88.  31. 397. 392. 405. 331. 367.  20.
  83. 416.  51. 395. 102.  34.  84. 360. 394. 457. 382. 461. 478. 386.
 112. 486. 421.   9. 308. 135. 224. 504. 269. 356. 498. 390. 513. 203.
 263. 477. 521. 169. 515. 445. 337. 251. 428. 292. 388. 130. 250. 355.
 254. 543. 531. 528.  62. 120.  42.  81. 116. 530. 103.  39.  16.  92.
  61. 501. 165. 291. 290.  43. 325. 192. 108. 200. 465. 287. 297. 490.
 482. 207. 282. 437. 225. 329. 272.  28.  77. 338.  72. 246. 319. 146.
 159. 380. 323. 511. 407. 278.  80. 403. 399.  14. 137. 343. 346. 347.
 349. 289. 351. 353.  54.  99. 358. 361. 362. 366. 372. 365. 277. 109.
 377. 379.  22. 378. 330. 364. 401. 232. 255. 384. 167. 212. 514. 391.
 400. 376. 402. 396. 302. 398.   6. 370. 369. 409. 168. 104. 408. 413.
 148.  10. 333. 419. 415. 424. 425. 423. 422. 435. 439. 442. 448. 443.
 454. 444.  52. 459. 458. 456. 460. 447. 470. 466. 484. 184. 485.  32.
 487. 491. 494. 193. 516. 496. 499.  29.  78. 520. 507. 506. 512. 126.
  64. 242. 518. 523. 539. 534. 436. 525. 541.  40. 455. 410.  45.  38.
  49.  48.  67.  68.  65.  91.  37.   8. 179. 209. 219. 221. 227. 153.
 186. 253. 202. 216. 275. 233. 280. 309. 321.  93. 316.  85. 107. 350.
 279. 334. 348. 150.  73. 385. 418. 197. 450. 452. 115.  46.  76.  96.
 100. 105. 101. 122.  11. 139. 142. 127. 143. 140. 149. 163. 160. 180.
 238. 183. 222. 185. 217. 215. 213. 237. 230. 234.  35. 245. 158. 258.
 259. 260. 411. 257. 271.  18. 106. 210. 273.  71. 284. 301. 305. 293.
```

```
 264. 311. 304. 313. 288. 320. 314. 332. 341. 352. 243. 368. 393. 132.
 220. 412. 420. 426. 417. 429. 433. 446. 357. 479. 483. 489. 229. 481.
 497. 451. 492.]
----
days_in_waiting_list: 128 unique values
[  0  50  47  65 122  75 101 150 125  14  60  34 100  22 121  61  39   5
   1   8 107  43  52   2  11 142 116  13  44  97  83   4 113  18  20 185
  93 109   6  37 105 154  64  99  38  48  33  77  21  80  59  40  58  89
  53  49  69  87  91  57 111  79  98  85  63  15   3  41 224  31  56 187
 176  71  55  96 236 259 207 215 160 120  30  32  27  62  24 108 147 379
  70  35 178 330 223 174 162 391  68 193  10  76  16  28   9 165  17  25
  46   7  84 175 183  23 117  12  54  26  73  45  19  42  72  81  92  74
 167  36]
----
customer_type: 4 unique values
['Transient' 'Contract' 'Transient-Party' 'Group']
----
adr: 8879 unique values
[  0.    75.    98.   ... 266.75 209.25 157.71]
----
required_car_parking_spaces: 5 unique values
[0 1 2 8 3]
----
total_of_special_requests: 6 unique values
[0 1 3 2 4 5]
----
reservation_status: 3 unique values
['Check-Out' 'Canceled' 'No-Show']
----
reservation_status_date: 926 unique values
<DatetimeArray>
['2015-07-01 00:00:00', '2015-07-02 00:00:00', '2015-07-03 00:00:00',
 '2015-05-06 00:00:00', '2015-04-22 00:00:00', '2015-06-23 00:00:00',
 '2015-07-05 00:00:00', '2015-07-06 00:00:00', '2015-07-07 00:00:00',
 '2015-07-08 00:00:00',
 ...
 '2015-03-13 00:00:00', '2015-05-05 00:00:00', '2015-03-29 00:00:00',
 '2015-06-10 00:00:00', '2015-04-27 00:00:00', '2014-10-17 00:00:00',
 '2015-01-20 00:00:00', '2015-02-17 00:00:00', '2015-03-10 00:00:00',
 '2015-03-23 00:00:00']
Length: 926, dtype: datetime64[ns]
----
name: 81501 unique values
['Ernest Barnes' 'Andrea Baker' 'Rebecca Parker' ... 'Wesley Aguilar'
 'Caroline Conley MD' 'Ariana Michael']
----
email: 115885 unique values
['Ernest.Barnes31@outlook.com' 'Andrea_Baker94@aol.com'
 'Rebecca_Parker@comcast.net' ... 'Mary_Morales@hotmail.com'
 'MD_Caroline@comcast.net' 'Ariana_M@xfinity.com']
----
phone-number: 119386 unique values
['669-792-1661' '858-637-6955' '652-885-2745' ... '395-518-4100'
 '531-528-1017' '422-804-6403']
----
credit_card: 9000 unique values
```

```
['************4322' '************9157' '************3734' ...
 '************9170' '************6349' '************7959']
----
arrival_date_full: 793 unique values
<DatetimeArray>
['2015-07-01 00:00:00', '2015-07-02 00:00:00', '2015-07-03 00:00:00',
 '2015-07-04 00:00:00', '2015-07-05 00:00:00', '2015-07-06 00:00:00',
 '2015-07-07 00:00:00', '2015-07-08 00:00:00', '2015-07-09 00:00:00',
 '2015-07-10 00:00:00',
 ...
 '2017-08-31 00:00:00', '2016-01-20 00:00:00', '2015-12-09 00:00:00',
 '2017-03-21 00:00:00', '2016-01-11 00:00:00', '2015-12-16 00:00:00',
 '2015-11-22 00:00:00', '2016-01-24 00:00:00', '2016-03-06 00:00:00',
 '2016-11-13 00:00:00']
Length: 793, dtype: datetime64[ns]
----
status_minus_arrival_date: 10 unique values
[0 1 2 5 7 8 4 6 3 9]
----
```

# Filtering Methods

## ANOVA

In [35]:

```
# Selecting the attributes most related with the target attribute by applyin
# Filter methods: ANOVA

# Define the ANOVA function
from scipy.stats import f_oneway

def FunctionAnova(inpData, TargetVariable, ContinuousPredictorList):
    # Creating an empty list of final selected predictors
    SelectedPredictors = []

    print('##### ANOVA Results ##### \n')
    for predictor in ContinuousPredictorList:
        # Group data by the target variable and extract lists of values for
        CategoryGroupLists = inpData.groupby(TargetVariable)[predictor].appl

        # Perform ANOVA
        AnovaResults = f_oneway(*CategoryGroupLists)

        # If the ANOVA P-Value is <0.05, that means we reject the null hypot
        if (AnovaResults[1] < 0.05):
            print(predictor, 'is correlated with', TargetVariable, '| P-Valu
            SelectedPredictors.append(predictor)
        else:
            print(predictor, 'is NOT correlated with', TargetVariable, '| P-

    return SelectedPredictors

# Assuming the churn data is already loaded in a DataFrame 'df'

# Specify the target variable and continuous predictors
```

```
target_variable = 'is_canceled'
continuous_variables = df.select_dtypes(include=['int64', 'float64']).columr

# Apply the ANOVA function
selected_predictors_anova = FunctionAnova(df, target_variable, continuous_va

# Print selected predictors based on ANOVA
print("Selected predictors based on ANOVA:", selected_predictors_anova)
```

```
##### ANOVA Results #####

is_canceled is correlated with is_canceled | P-Value: 0.0
lead_time is correlated with is_canceled | P-Value: 0.0
arrival_date_year is correlated with is_canceled | P-Value: 7.39046497604589
05e-09
arrival_date_week_number is correlated with is_canceled | P-Value: 0.0049571
4971703516
arrival_date_day_of_month is correlated with is_canceled | P-Value: 0.035535
45701304093
stays_in_weekend_nights is NOT correlated with is_canceled | P-Value: 0.5377
670410936434
stays_in_week_nights is correlated with is_canceled | P-Value: 1.12555722552
96351e-17
adults is correlated with is_canceled | P-Value: 1.319220870630564e-95
children is NOT correlated with is_canceled | P-Value: 0.08113827981548259
babies is correlated with is_canceled | P-Value: 2.9596104105688555e-29
C:\Python312\Lib\site-packages\scipy\stats\_axis_nan_policy.py:573: Constant
InputWarning: Each of the input arrays is constant; the F statistic is not d
efined or infinite
  res = hypotest_fun_out(*samples, **kwds)
is_repeated_guest is correlated with is_canceled | P-Value: 2.48109289221877
58e-189
previous_cancellations is correlated with is_canceled | P-Value: 8.2654e-319
previous_bookings_not_canceled is correlated with is_canceled | P-Value: 1.5
238877962617572e-87
booking_changes is correlated with is_canceled | P-Value: 0.0
agent is NOT correlated with is_canceled | P-Value: nan
company is NOT correlated with is_canceled | P-Value: nan
days_in_waiting_list is correlated with is_canceled | P-Value: 2.40121779355
3042e-78
adr is correlated with is_canceled | P-Value: 6.684831825750079e-61
required_car_parking_spaces is correlated with is_canceled | P-Value: 0.0
total_of_special_requests is correlated with is_canceled | P-Value: 0.0
status_minus_arrival_date is correlated with is_canceled | P-Value: 0.000272
5231156768738
Selected predictors based on ANOVA: ['is_canceled', 'lead_time', 'arrival_da
te_year', 'arrival_date_week_number', 'arrival_date_day_of_month', 'stays_in
_week_nights', 'adults', 'babies', 'is_repeated_guest', 'previous_cancellati
ons', 'previous_bookings_not_canceled', 'booking_changes', 'days_in_waiting_
list', 'adr', 'required_car_parking_spaces', 'total_of_special_requests', 's
tatus_minus_arrival_date']
```

## Simplified ANOVA Results Analysis

This analysis looks at how different features are related to booking cancellations ( `is_canceled` ) based on their P-values. A lower P-value indicates a stronger relationship with cancellations.

## Strongly Correlated Features

The following features have very low P-values, showing a significant correlation with cancellations:

- Lead Time (P-Value: 0.0): The longer the lead time, the more likely a booking is to be canceled.
- Stays in Week Nights (P-Value: 1.13e-17): Bookings that include more weekday nights tend to have a higher chance of cancellation.
- Adults (P-Value: 1.32e-95): The number of adults in a booking is highly correlated with cancellations.
- Babies (P-Value: 2.96e-29): Bookings that involve babies are also more likely to be canceled.
- Is Repeated Guest (P-Value: 2.48e-189): Repeat guests are significantly less likely to cancel their bookings.
- Previous Cancellations (P-Value: 8.27e-319): Guests with a history of cancellations are much more likely to cancel again.
- Booking Changes (P-Value: 0.0): More changes to a booking are linked to higher cancellation rates.
- Days in Waiting List (P-Value: 2.40e-78): Longer waiting times increase the likelihood of cancellations.
- ADR (Average Daily Rate) (P-Value: 6.68e-61): The pricing of the booking affects cancellation behavior.
- Required Car Parking Spaces (P-Value: 0.0): More requests for parking spaces correlate with lower cancellation rates.
- Total of Special Requests (P-Value: 0.0): Higher engagement through special requests also leads to lower cancellation likelihood.

## Moderately Correlated Features

Some features show moderate correlation with cancellations:

- Arrival Date Year (P-Value: 7.39e-09), Arrival Date Week Number (P-Value: 0.0049), and Arrival Date Day of Month (P-Value: 0.035): These features indicate that the timing of a booking can influence cancellations.

## Non-Correlated Features

Certain features do not show significant relationships with cancellations:

- Stays in Weekend Nights (P-Value: 0.538): The number of weekend nights does not significantly affect cancellations.

- Children (P-Value: 0.081): The presence of children in a booking has a weaker correlation with cancellations.
- Agent and Company (P-Value: nan): These features have no clear relationship with cancellations, possibly due to insufficient data.

Some features may influence the cancellation rates:

- Customer Behavior: Guests with a history of cancellations or who are first-time guests are more likely to cancel.
- Booking Details: Features like lead time, the number of adults or babies, and the number of booking changes are important indicators of cancellation likelihood.
- Engagement Matters: Higher customer engagement (like special requests or parking space needs) reduces the chances of cancellations.

## Chi-Square

In [36]:
```
import pandas as pd
from scipy.stats import chi2_contingency

# Function to perform the Chi-Square test
def FunctionChisq(inpData, TargetVariable, CategoricalVariablesList):
    SelectedPredictors = []  # List to hold selected predictors

    for predictor in CategoricalVariablesList:
        CrossTabResult = pd.crosstab(index=inpData[TargetVariable], columns=
        ChiSqResult = chi2_contingency(CrossTabResult)

        # If the P-Value is < 0.05, it indicates correlation
        if ChiSqResult[1] < 0.05:
            print(f'{predictor} is correlated with {TargetVariable} | P-Valu
            SelectedPredictors.append(predictor)
        else:
            print(f'{predictor} is NOT correlated with {TargetVariable} | P-

    return SelectedPredictors

# Specify the target variable
target_variable = 'is_canceled'

# Specify categorical variables relevant to the project, excluding irrelevar
categorical_variables = ['hotel', 'meal', 'country', 'market_segment',
                         'distribution_channel', 'reserved_room_type', 'assi
                         'deposit_type', 'customer_type', 'reservation_statu

# Apply the Chi-Square test function
selected_predictors = FunctionChisq(df, target_variable, categorical_variabl

# Print selected predictors
print("Selected predictors based on Chi-Square test:", selected_predictors)
```

```
hotel is correlated with is_canceled | P-Value: 0.0000
meal is correlated with is_canceled | P-Value: 0.0000
country is correlated with is_canceled | P-Value: 0.0000
market_segment is correlated with is_canceled | P-Value: 0.0000
distribution_channel is correlated with is_canceled | P-Value: 0.0000
reserved_room_type is correlated with is_canceled | P-Value: 0.0000
assigned_room_type is correlated with is_canceled | P-Value: 0.0000
deposit_type is correlated with is_canceled | P-Value: 0.0000
customer_type is correlated with is_canceled | P-Value: 0.0000
reservation_status is correlated with is_canceled | P-Value: 0.0000
Selected predictors based on Chi-Square test: ['hotel', 'meal', 'country',
'market_segment', 'distribution_channel', 'reserved_room_type', 'assigned_ro
om_type', 'deposit_type', 'customer_type', 'reservation_status']
```

The results show that several categorical variables are significantly correlated with the cancellation status ( `is_canceled` ). For instance, the type of hotel ( `hotel` ) plays an important role, as different hotels (e.g., resort vs. city hotels) tend to attract different types of guests, leading to variations in cancellation patterns. Similarly, the choice of meal plan ( `meal` ) influences cancellations, suggesting that guests who select different meal options may have varying preferences or levels of commitment to their bookings.

The guest's country ( `country` ) is another key factor, likely because guests from different regions have different travel habits, economic situations, or booking behaviors. The market segment ( `market_segment` ) and distribution channel ( `distribution_channel` ) also play a role, as these variables reflect how and through which platform guests make their bookings, which could impact their likelihood of canceling.

Room selection is crucial, with both the reserved room type ( `reserved_room_type` ) and the assigned room type ( `assigned_room_type` ) showing correlation with cancellations. This might indicate that mismatches between the type of room booked and the one assigned upon arrival affect guest satisfaction and lead to higher cancellation rates.

The deposit type ( `deposit_type` ) is another significant predictor, as the type of deposit policy (e.g., non-refundable or refundable) strongly influences a guest’s decision to cancel. Similarly, the customer type ( `customer_type` ), which distinguishes between new and repeat customers, affects cancellation rates, possibly due to differing levels of loyalty or booking patterns.

Finally, the reservation status ( `reservation_status` ), which indicates whether a booking was canceled, checked-in, or a no-show, is clearly related to the cancellation outcome.

# Customer Segmentation

The goal of this segmentation is to understand the possible types of hotel guests into clusters and analyze those clusters in order to gain a better understanding of them.

## Model Preparation

To avoid leaking information about our target variable (cancellation) into the clusters, we need to remove certain features from our X variable, including is_canceled, reservation_status, and country. These clusters will later be used as features in our predictive model, so this step is crucial.

Additionally, the agent and company features contain a large amount of categorical data that has been de-identified, making it difficult to interpret. For this reason, they were also excluded from the clustering model. Since models cannot process datetime objects directly, the reservation_status_date and arrival_date_full features were removed as well. However, information about the arrival date is still included through features like arrival_date_year, arrival_date_month, and arrival_date_day_of_month.

In [37]:
```
X = df.drop(columns=['is_canceled', 'reservation_status', 'agent', 'company'
```

In [38]:
```
from sklearn.model_selection import train_test_split, cross_val_score

df = pd.get_dummies(df, columns=['hotel', 'arrival_date_month', 'meal', 'mar

X = df.drop(columns=['is_canceled', 'reservation_status', 'agent', 'company'

y = df['is_canceled']
```

In [39]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, s
```

In [40]:
```
X.head()
```

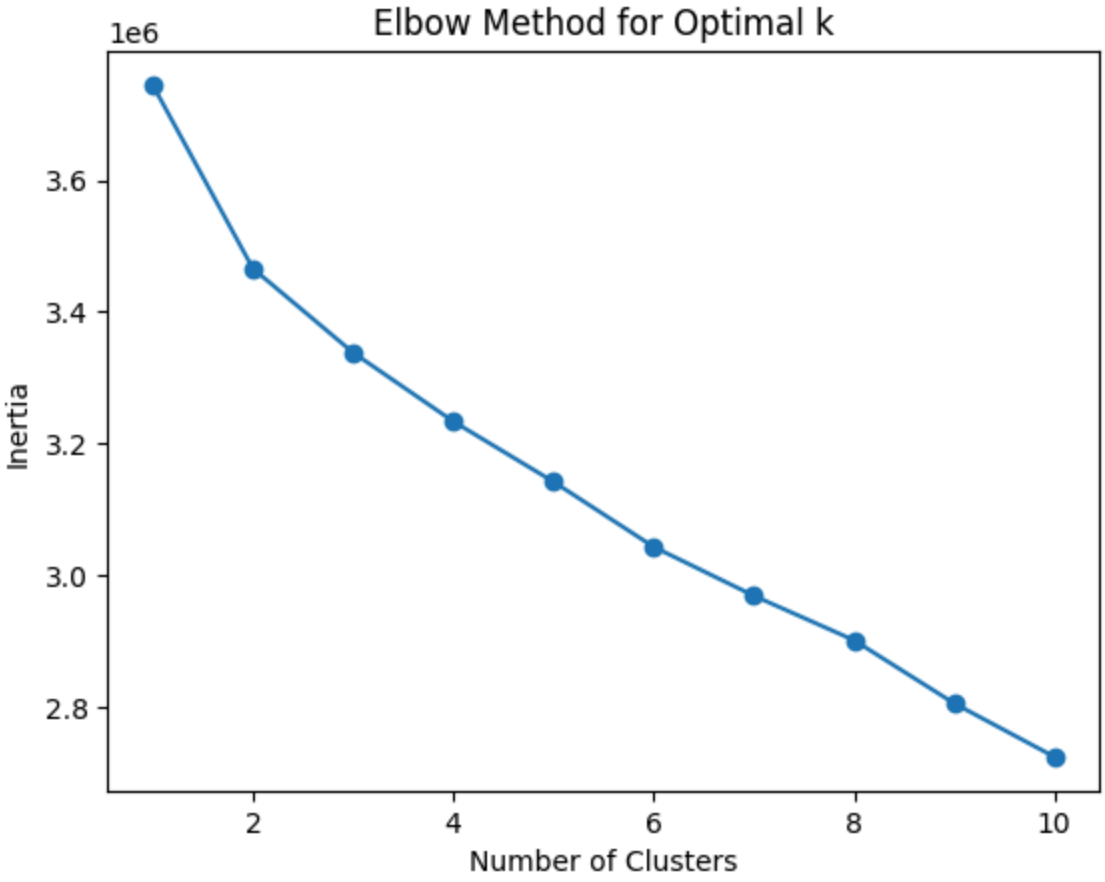
Out[40]:

| | lead_time | arrival_date_year | arrival_date_week_number | arrival_date_day_of |
|---|---|---|---|---|
| **0** | 342 | 2015 | 27 | |
| **1** | 737 | 2015 | 27 | |
| **2** | 7 | 2015 | 27 | |
| **3** | 13 | 2015 | 27 | |
| **4** | 14 | 2015 | 27 | |

5 rows × 78 columns

# Clustering

In [42]:
```
from sklearn.impute import SimpleImputer
import pandas as pd
from sklearn.preprocessing import StandardScaler, OrdinalEncoder
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Identifying and removing datetime columns
datetime_cols = df.select_dtypes(include=['datetime64']).columns
df = df.drop(columns=datetime_cols)  # Drop datetime columns

# Separating numeric and categorical columns
numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns
categorical_cols = df.select_dtypes(include=['object']).columns

# Ordinal encoding categorical columns
ordinal_encoder = OrdinalEncoder()
df[categorical_cols] = ordinal_encoder.fit_transform(df[categorical_cols])

# Imputing missing values with the mean for numeric columns
imputer = SimpleImputer(strategy='mean')
df[numeric_cols] = imputer.fit_transform(df[numeric_cols])

# Standardizing the numerical features
scaler = StandardScaler()
df[numeric_cols] = scaler.fit_transform(df[numeric_cols])

# Clustering the data
X_scaled = df

# Determining the optimal number of clusters using the Elbow Method
inertia = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_scaled)
    inertia.append(kmeans.inertia_)

# Plotting the Elbow Method graph
plt.plot(range(1, 11), inertia, marker='o')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.title('Elbow Method for Optimal k')
plt.show()

# Choosing k based on the elbow point and fitting the final model
optimal_k = 4  # Replace with your choice based on the elbow plot
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
cluster_labels = kmeans.fit_predict(X_scaled)

# Display cluster labels
print("Cluster labels:", cluster_labels)
```




```
Cluster labels: [0 0 0 ... 1 1 1]
```

Cluster Labels: The array [1, 1, 1, ... 0, 0, 0] represents the cluster labels assigned to each data point by the K-Means algorithm. Each number (e.g., 0, 1) corresponds to a specific cluster. For instance, data points with a label of 0 belong to one cluster, while those with 1 belong to another.

In [43]:
```
# Analyze the clusters
import numpy as np
unique, counts = np.unique(cluster_labels, return_counts=True)
cluster_counts = dict(zip(unique, counts))
print("Cluster Counts:", cluster_counts)
```

```
Cluster Counts: {np.int32(0): np.int64(57292), np.int32(1): np.int64(21632),
np.int32(2): np.int64(39924), np.int32(3): np.int64(538)}
```

In [44]:
```
df['Cluster'] = cluster_labels
```

## Analyze Cluster Characteristics

We will calculate the mean of each feature within each cluster to understand the typical values of features for each group.

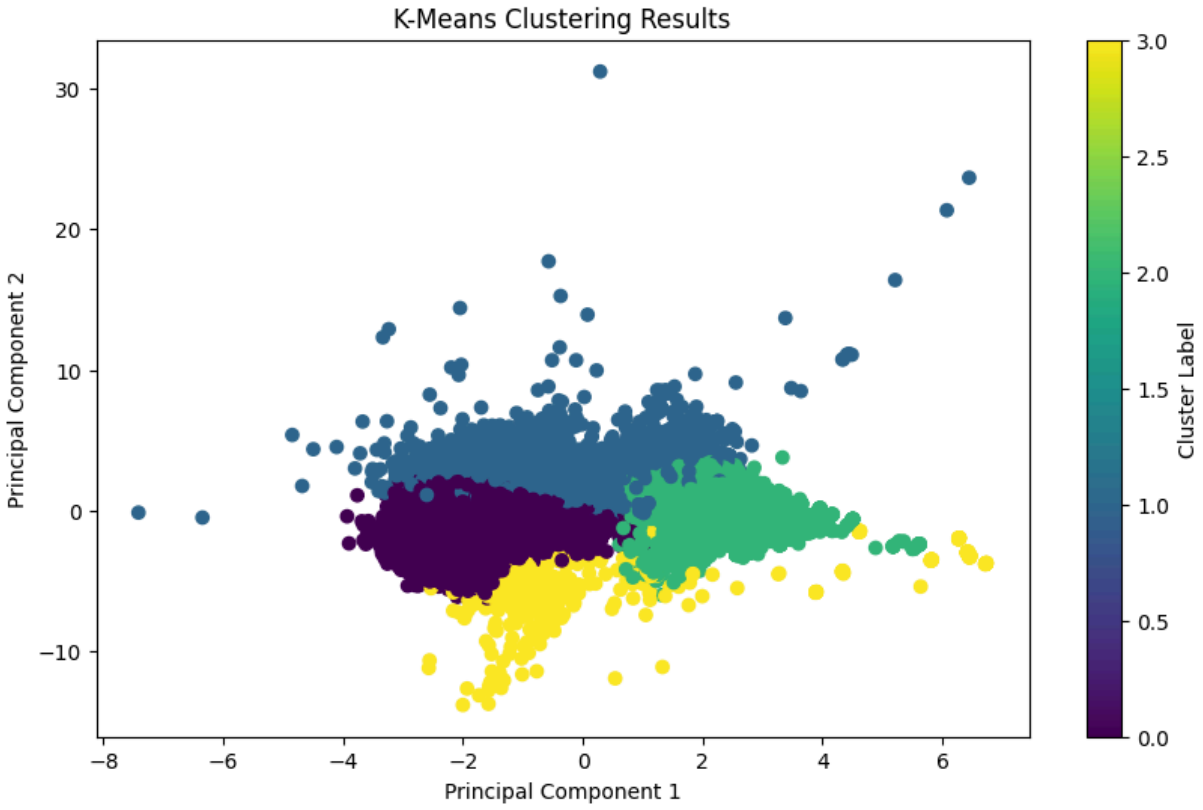
In [45]:
```
cluster_means = df.groupby('Cluster').mean()
print(cluster_means)
```

```
         is_canceled  lead_time  arrival_date_year  arrival_date_week_number
\
Cluster
0          -0.728693  -0.348433          -0.036182                 -0.041047
1          -0.471591   0.166660           0.084593                  0.088776
2           1.302111   0.413299           0.008008                  0.012593
3          -0.066483  -0.266453          -0.142526                 -0.132899

         arrival_date_day_of_month  stays_in_weekend_nights  \
Cluster
0                         0.007373                -0.317820
1                        -0.002934                 1.085873
2                        -0.006031                -0.127686
3                        -0.219647                -0.340716

         stays_in_week_nights    adults  children    babies  ...  \
Cluster                                                      ...
0                   -0.377792 -0.139552 -0.109356 -0.039663  ...
1                    1.210426  0.275054  0.432244  0.227752  ...
2                   -0.108528  0.062460 -0.073824 -0.065386  ...
3                   -0.383933 -0.833452 -0.256000 -0.081581  ...

         assigned_room_type_K  assigned_room_type_L  assigned_room_type_P  \
Cluster
0                    0.003962              0.000000              0.000000
1                    0.001942              0.000000              0.000046
2                    0.000225              0.000025              0.000276
3                    0.001859              0.000000              0.000000

         deposit_type_No Deposit  deposit_type_Non Refund  \
Cluster
0                       0.996125                 0.002025
1                       0.994360                 0.004345
2                       0.642070                 0.357229
3                       0.786245                 0.213755

         deposit_type_Refundable  customer_type_Contract  customer_type_Grou
p  \
Cluster
0                       0.001850                0.021644             0.00775
0
1                       0.001294                0.076877             0.00411
4
2                       0.000701                0.029381             0.00100
2
3                       0.000000                0.000000             0.00743
5

         customer_type_Transient  customer_type_Transient-Party
Cluster
0                       0.692331                       0.278276
1                       0.773992                       0.145017
2                       0.820083                       0.149534
3                       0.862454                       0.130112

[4 rows x 87 columns]
```

## PCA for Visualization

We'll reduce to two dimensions using Principal Component Analysis (PCA) to visualize the clusters.

In [46]:
```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Reduce to 2D
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Plot
plt.figure(figsize=(10, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=cluster_labels, cmap='viridis', mark
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('K-Means Clustering Results')
plt.colorbar(label='Cluster Label')
plt.show()
```




Each color represents a distinct cluster. The clusters are somewhat overlapping, suggesting that the data might not have perfectly distinct clusters. However, we can stil observe groupings.

We noticed 4 clusters groups (labeled 0, 1, 2, and 3), represented by different colors.

The x-axis and y-axis represent the two main principal components (PC1 and PC2) derived from PCA. These are the two most significant directions of variance in the data. Although these components are a transformation of the original features, they help visualize how clusters are separated in the most significant directions.

In [47]:
```
# Cluster caracteristics
cluster_means = df.groupby('Cluster').mean()
print(cluster_means)
```

```
         is_canceled  lead_time  arrival_date_year  arrival_date_week_number
\
Cluster
0          -0.728693  -0.348433          -0.036182                 -0.041047
1          -0.471591   0.166660           0.084593                  0.088776
2           1.302111   0.413299           0.008008                  0.012593
3          -0.066483  -0.266453          -0.142526                 -0.132899

         arrival_date_day_of_month  stays_in_weekend_nights  \
Cluster
0                         0.007373                -0.317820
1                        -0.002934                 1.085873
2                        -0.006031                -0.127686
3                        -0.219647                -0.340716

         stays_in_week_nights    adults  children    babies  ...  \
Cluster                                                      ...
0                   -0.377792 -0.139552 -0.109356 -0.039663  ...
1                    1.210426  0.275054  0.432244  0.227752  ...
2                   -0.108528  0.062460 -0.073824 -0.065386  ...
3                   -0.383933 -0.833452 -0.256000 -0.081581  ...

         assigned_room_type_K  assigned_room_type_L  assigned_room_type_P  \
Cluster
0                    0.003962              0.000000              0.000000
1                    0.001942              0.000000              0.000046
2                    0.000225              0.000025              0.000276
3                    0.001859              0.000000              0.000000

         deposit_type_No Deposit  deposit_type_Non Refund  \
Cluster
0                       0.996125                 0.002025
1                       0.994360                 0.004345
2                       0.642070                 0.357229
3                       0.786245                 0.213755

         deposit_type_Refundable  customer_type_Contract  customer_type_Grou
p  \
Cluster
0                       0.001850                0.021644             0.00775
0
1                       0.001294                0.076877             0.00411
4
2                       0.000701                0.029381             0.00100
2
3                       0.000000                0.000000             0.00743
5

         customer_type_Transient  customer_type_Transient-Party
Cluster
0                       0.692331                       0.278276
1                       0.773992                       0.145017
2                       0.820083                       0.149534
3                       0.862454                       0.130112

[4 rows x 87 columns]
```

In [48]:
```
# Evaluate Cluster Quality
from sklearn.metrics import silhouette_score
silhouette_avg = silhouette_score(X_scaled, cluster_labels)
print("Silhouette Score:", silhouette_avg)
```

```
Silhouette Score: 0.10741323342781463
```

In [49]:
```
# To find the best number of clusters, we are calculating the Silhouette Sco
# Few number of clusters (2, 3) and bigger cluster number (5, 6)
for k in [2, 3, 5, 6]:
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(X_scaled)
    score = silhouette_score(X_scaled, labels)
    print(f"Silhouette Score for k={k}: {score}")
```

```
Silhouette Score for k=2: 0.10825808889566212
Silhouette Score for k=3: 0.10709019184713382
Silhouette Score for k=5: 0.08392269438412117
Silhouette Score for k=6: 0.05877172372703965
```

## DBSCAN

In [50]:
```
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.5, min_samples=5)  # Adjust eps and min_samples
dbscan_labels = dbscan.fit_predict(X_scaled)
dbscan_score = silhouette_score(X_scaled, dbscan_labels)
print("Silhouette Score for DBSCAN:", dbscan_score)
```

```
Silhouette Score for DBSCAN: -0.17425827849996053
```

DBSCAN can detect clusters of varying shapes and densities and also handles outliers well

In this case, its not good since the results are negative.

## Gaussian Mixture Model (GMM)

In [51]:
```
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=4, random_state=42)  # Adjust different v
gmm_labels = gmm.fit_predict(X_scaled)
gmm_score = silhouette_score(X_scaled, gmm_labels)
print("Silhouette Score for GMM:", gmm_score)
```

```
Silhouette Score for GMM: 0.08714533712578033
```

- Ideal model : GMM

## Silhouette Score for GMM

In [52]:
```
from sklearn.mixture import GaussianMixture
from sklearn.metrics import silhouette_score

# Try different numbers of components
for n in range(2, 11):
    gmm = GaussianMixture(n_components=n, random_state=42)
    labels = gmm.fit_predict(X_scaled)
    score = silhouette_score(X_scaled, labels)
    print(f"Silhouette Score for GMM with {n} components: {score}")
```

```
Silhouette Score for GMM with 2 components: 0.10797003212395127
Silhouette Score for GMM with 3 components: 0.05367926475492533
Silhouette Score for GMM with 4 components: 0.08714533712578033
Silhouette Score for GMM with 5 components: -0.007426534228595482
Silhouette Score for GMM with 6 components: 0.008773285497541939
Silhouette Score for GMM with 7 components: 0.01430446614236575
Silhouette Score for GMM with 8 components: 0.04950221334048227
Silhouette Score for GMM with 9 components: 0.0016351070305528853
Silhouette Score for GMM with 10 components: 0.029213433493921324
```

## Silhouette Score for PCA

In [53]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=0.95)  # Retain 95% of variance
X_pca = pca.fit_transform(X_scaled)

kmeans = KMeans(n_clusters=4, random_state=42)
pca_labels = kmeans.fit_predict(X_pca)
pca_silhouette = silhouette_score(X_pca, pca_labels)
print("Silhouette Score after PCA:", pca_silhouette)
```

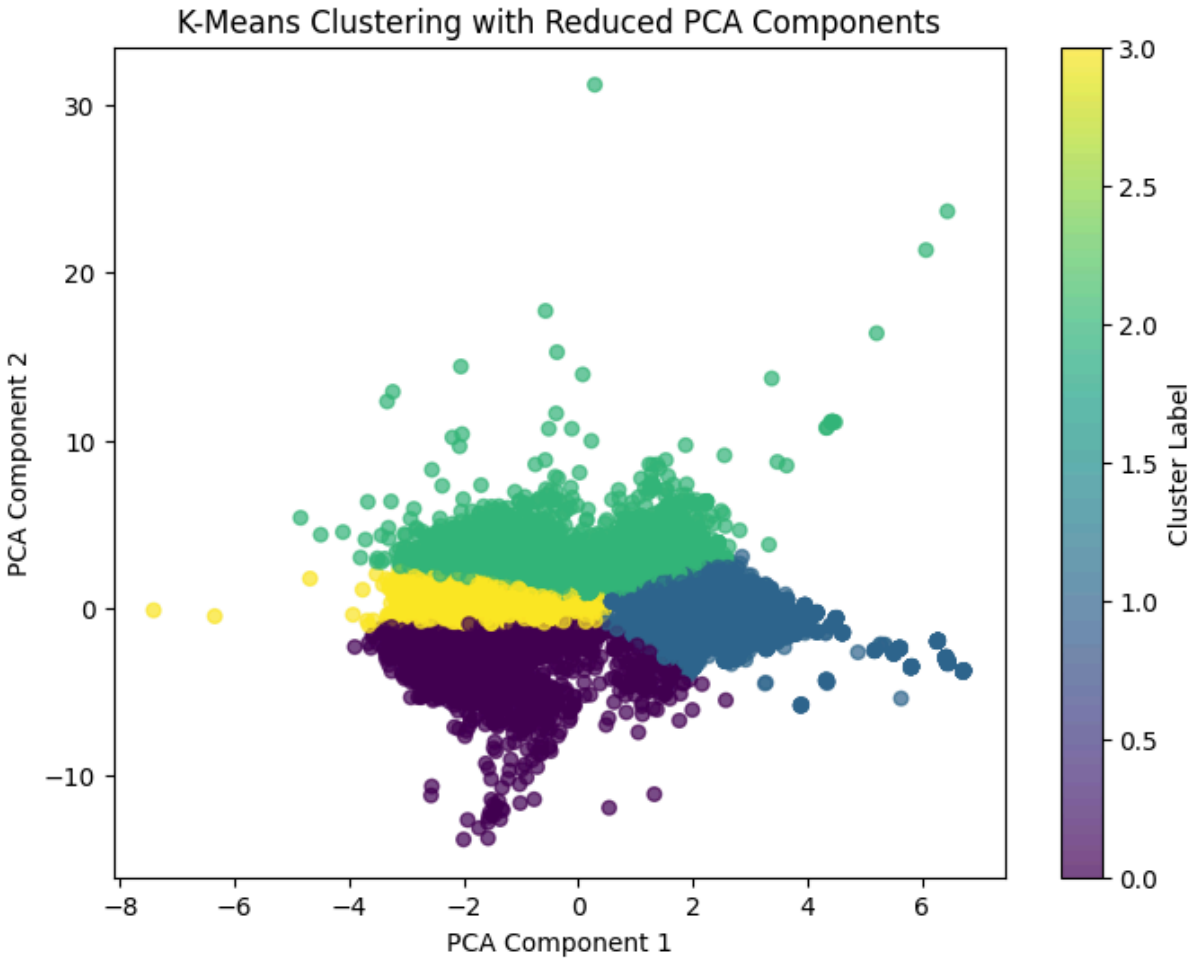
```
Silhouette Score after PCA: 0.11636098287528675
```

## Plot Visualization

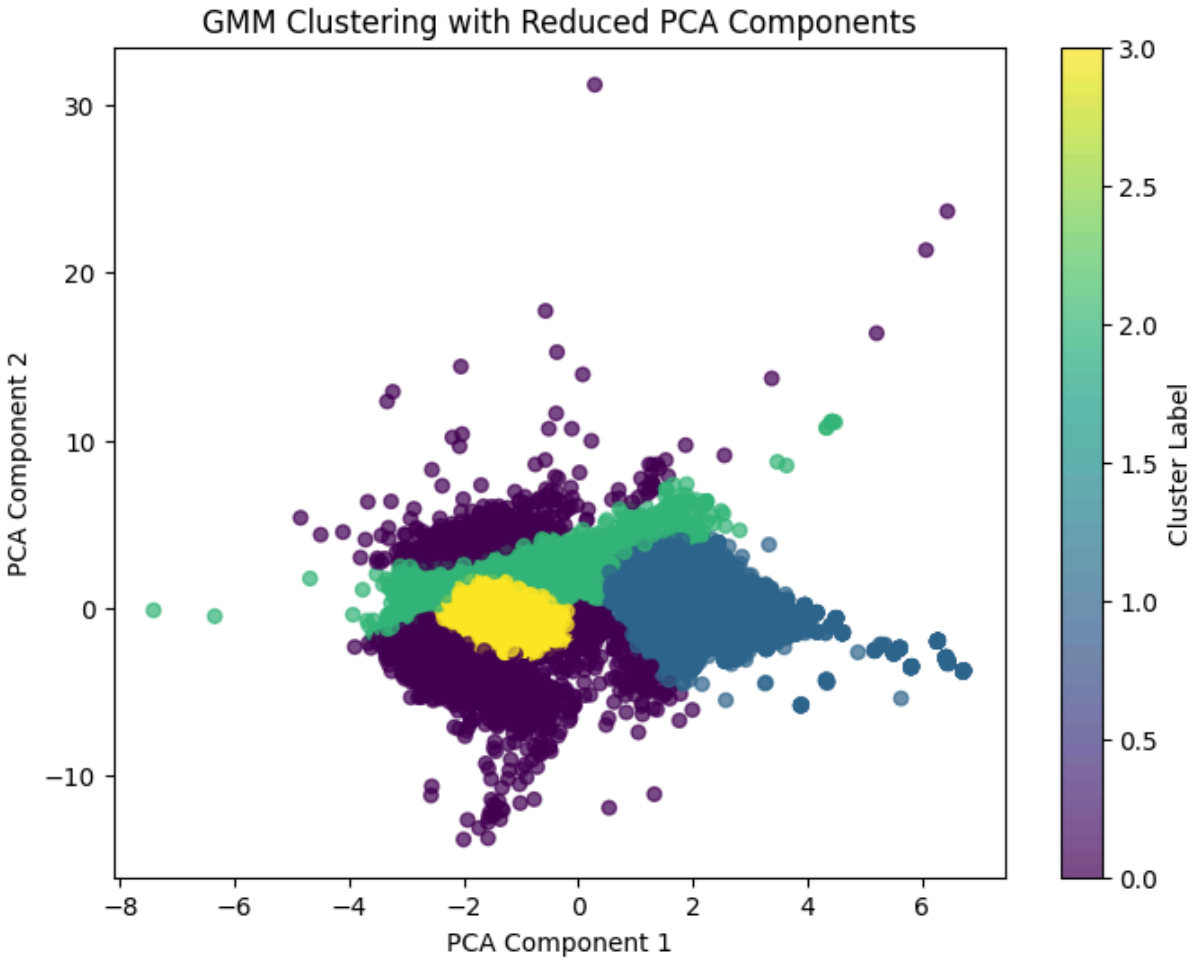
In [ ]:
```
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.mixture import GaussianMixture
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Reducing to 2 PCA components
pca = PCA(n_components=2)
X_pca_reduced = pca.fit_transform(X_scaled)

# Function to plot clusters
def plot_clusters(X, labels, title):
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=30, alpha=0.7)
    plt.xlabel('PCA Component 1')
    plt.ylabel('PCA Component 2')
    plt.title(title)
```

```
    plt.colorbar(label='Cluster Label')
    plt.show()

# K-Means Clustering on PCA-reduced data
kmeans = KMeans(n_clusters=4, random_state=42)
labels_kmeans = kmeans.fit_predict(X_pca_reduced)
silhouette_kmeans = silhouette_score(X_pca_reduced, labels_kmeans)
print("Silhouette Score for K-Means with reduced PCA components:", silhouett

# Plot K-Means clusters
plot_clusters(X_pca_reduced, labels_kmeans, "K-Means Clustering with Reduced

# Gaussian Mixture Model (GMM) on PCA-reduced data
gmm = GaussianMixture(n_components=4, random_state=42)
labels_gmm = gmm.fit_predict(X_pca_reduced)
silhouette_gmm = silhouette_score(X_pca_reduced, labels_gmm)
print("Silhouette Score for GMM with reduced PCA components:", silhouette_gm

# Plot GMM clusters
plot_clusters(X_pca_reduced, labels_gmm, "GMM Clustering with Reduced PCA Co

# Taking a smaller sample for Agglomerative Clustering to avoid memory issue
_, X_sampled, _, y_sampled = train_test_split(X_pca_reduced, range(len(X_pca

# Agglomerative Clustering on the sampled data
agglomerative = AgglomerativeClustering(n_clusters=4)
labels_agg = agglomerative.fit_predict(X_sampled)
silhouette_agg = silhouette_score(X_sampled, labels_agg)
print("Silhouette Score for Agglomerative Clustering with sampled data:", si

# Plot Agglomerative Clustering clusters on sampled data
plot_clusters(X_sampled, labels_agg, "Agglomerative Clustering with Sampled
```

```
Silhouette Score for K-Means with reduced PCA components: 0.4470023124255576
```




Silhouette Score for GMM with reduced PCA components: 0.41932779951929655




## PCA Reduction

We reduced the dataset to 2 or 3 PCA components to simplify the structure and emphasize key variance patterns.

## Clustering Methods

We applied three clustering methods:

- K-Means: Standard clustering method, re-evaluated on reduced data.
- Gaussian Mixture Model (GMM): Probabilistic clustering to handle elliptical clusters.
- Agglomerative Clustering: Hierarchical method, useful for identifying inherent grouping structures.

## Silhouette Score

We calculated the Silhouette Score for each method to evaluate the quality of clustering.

## Cluster Plotting

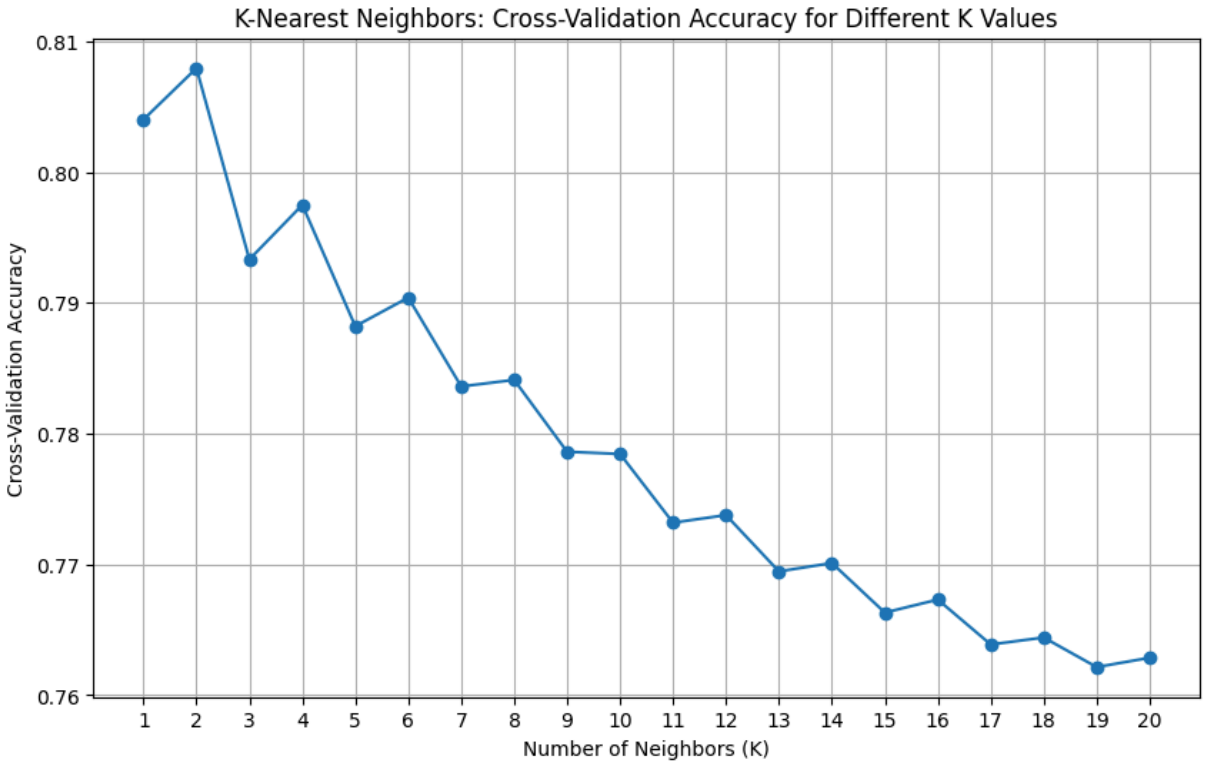
We visualized the clusters in the reduced 2D space for easy comparison.

# Data Classification

## K-Neighbours

In [55]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt

# Split the dataset into training and testing sets (optional, as we're using
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, s

# test_size=0.2 means 20% of the data will be used as the test set, while 80
# random_state=42 ensures reproducibility. The same data split will be gener

# Range of 'K' values to test
k_values = list(range(1, 21))

# Store the cross-validation scores for each 'K' value
cv_scores = []

# Perform cross-validation for each 'K' value
for k in k_values:
    knn_classifier = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn_classifier, X_train, y_train, cv=5)  # 5-fo
    cv_scores.append(np.mean(scores))

# Find the optimal 'K' value with the highest cross-validation score
optimal_k = k_values[np.argmax(cv_scores)]
print(f"Optimal K value: {optimal_k}")

# Train and evaluate the KNN classifier with the optimal 'K' value on the te
best_knn_classifier = KNeighborsClassifier(n_neighbors=optimal_k)
best_knn_classifier.fit(X_train, y_train)
test_accuracy = best_knn_classifier.score(X_test, y_test)
print(f"Test accuracy with optimal K: {test_accuracy:.2f}")

# Plot the cross-validation scores for each K value
plt.figure(figsize=(10, 6))
plt.plot(k_values, cv_scores, marker='o', linestyle='-')
plt.xlabel('Number of Neighbors (K)')
plt.ylabel('Cross-Validation Accuracy')
plt.title('K-Nearest Neighbors: Cross-Validation Accuracy for Different K Va
plt.xticks(k_values)
plt.grid(True)
plt.show()
```

```
Optimal K value: 2
Test accuracy with optimal K: 0.82
```




The optimal k value for this KNN model is 2, with a cross-validation accuracy of approximately 81% and a test accuracy of 82%. Lower values of k (e.g., 1–4) perform better, suggesting that this dataset benefits from a more locally sensitive KNN model. Higher values of k lead to lower accuracy, possibly due to underfitting, as the model becomes too generalized.

Based on cross-validation, the best model is one with k=2, as it achieves the highest accuracy on unseen data.

- Low values of k (e.g., 1 or 2) make the model more sensitive to local patterns in the data, which can improve performance but also risks overfitting in some cases.
- Higher values of k reduce the model's sensitivity to individual data points but may lead to underfitting by overly smoothing the decision boundaries.

## Naïve Bayes Classification

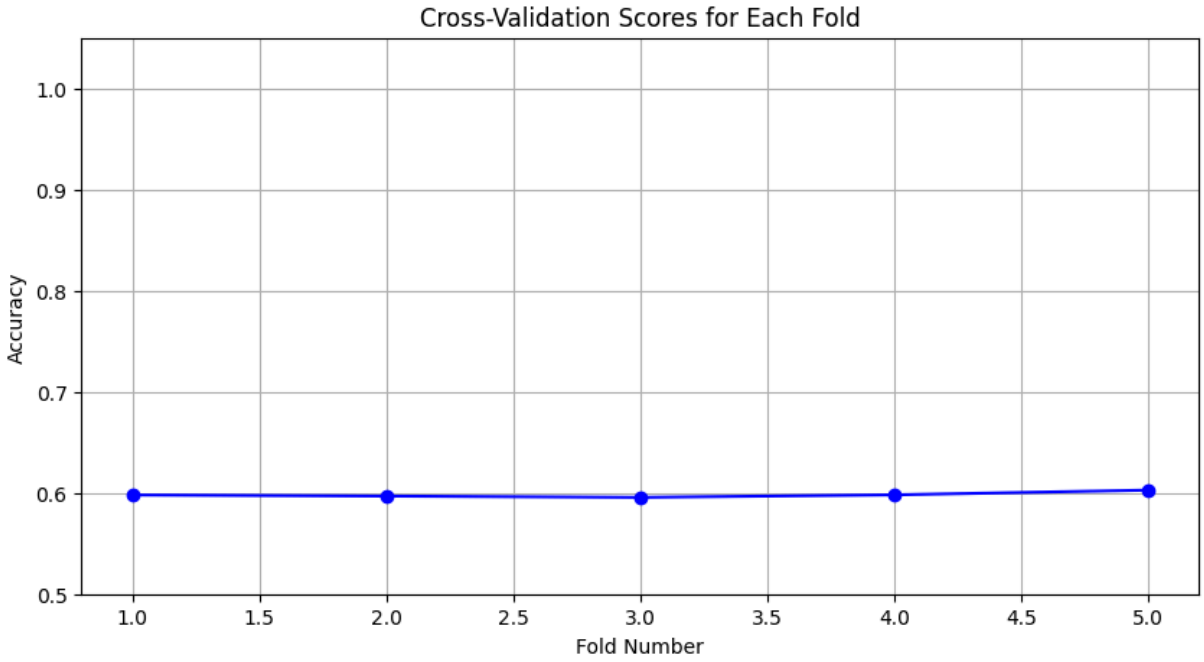
In [56]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import accuracy_score, classification_report

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar

# Train Naïve Bayes Model
nb_classifier = GaussianNB()
nb_classifier.fit(X_train, y_train)
```

```
# Cross-Validation
cv_scores = cross_val_score(nb_classifier, X_train, y_train, cv=5)  # 5-fold
print("Cross-Validation Scores:", cv_scores)
print("Mean Cross-Validation Score:", cv_scores.mean())

# Evaluate on Test Set
y_pred_test = nb_classifier.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred_test)
print(f"Test accuracy: {test_accuracy:.2f}")
print("Classification Report:\n", classification_report(y_test, y_pred_test))
```

```
Cross-Validation Scores: [0.59794786 0.5968485  0.59559208 0.59808387 0.6027
4331]
Mean Cross-Validation Score: 0.5982431245647767
Test accuracy: 0.60
Classification Report:
               precision    recall  f1-score   support

           0       0.82      0.46      0.59     14973
           1       0.48      0.83      0.61      8905

    accuracy                           0.60     23878
   macro avg       0.65      0.65      0.60     23878
weighted avg       0.69      0.60      0.60     23878
```

The Naive Bayes model is achieving 60% accuracy and has mixed performance on different classes, with good recall for class 1 but poor recall for class 0.

In [57]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Cross-Validation Scores
plt.figure(figsize=(10, 5))
plt.plot(range(1, len(cv_scores) + 1), cv_scores, marker='o', linestyle='-',
plt.title("Cross-Validation Scores for Each Fold")
plt.xlabel("Fold Number")
plt.ylabel("Accuracy")
plt.ylim(0.5, 1.05)  # Set y-axis limit for clarity
plt.grid(True)
plt.show()
```




## Cross-Validation Scores

The cross-validation accuracies are all around 0.59 to 0.60, indicating that the model consistently achieves around 59.9% accuracy during training on different subsets of the data. This score is fairly low, which suggests that the model may not be capturing strong patterns in the data.

## Test Accuracy

Test Accuracy: 0.60 (or 60%) The model achieves an accuracy of 60% on the test set, which is consistent with the cross-validation scores.

This indicates that the model's performance on unseen data is similar to its performance on training data, suggesting that the model is not overfitting but may still be underperforming overall.

## Classification Report:

- Precision: 0.82. Precision for class 0 is 0.82, which means that when the model predicts class 0, it's correct 82% of the time.
- Recall: 0.46. Recall for class 0 is 0.46, indicating that the model only identifies 46% of actual class 0 instances correctly. This suggests a high number of false negatives for class 0.
- F1-Score: 0.59 The F1-score, a harmonic mean of precision and recall, is 0.59. This is relatively low due to the lower recall.

- Precision: 0.48 Precision for class 1 is 0.48, meaning that when the model predicts class 1, it's only correct 48% of the time.
- Recall: 0.83 Recall for class 1 is 0.83, indicating that the model identifies 83% of actual class 1 instances correctly. This suggests that the model is better at detecting class 1 but often misclassifies other instances as class 1.
- F1-Score: 0.61 The F1-score for class 1 is 0.61, which is slightly better than for class 0 but still not ideal.

The overall accuracy of the model on the test set is 60%.

## Neural Networks

In [58]:
```
X.shape
```

Out[58]:
```
(119386, 78)
```

In [59]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
#from tensorflow.keras.wrappers.scikit_learn import KerasRegressor
from scikeras.wrappers import KerasClassifier
from tensorflow.keras.callbacks import EarlyStopping

# Scaling the data
ss = StandardScaler()

# Crie uma lista de colunas dummies para X_train
X_train_dummies = pd.get_dummies(X_train, drop_first=True)

# Crie uma lista de colunas dummies para X_test com as mesmas colunas
X_test_dummies = pd.get_dummies(X_test, drop_first=True)

# Alinhe as colunas para que tenham o mesmo número de colunas
X_train_final, X_test_final = X_train_dummies.align(X_test_dummies, join='le

# X_train_sc = ss.fit_transform(X_train)
# X_test_sc = ss.transform(X_test)

X_train_sc = ss.fit_transform(X_train_final)
X_test_sc = ss.transform(X_test_final)

print(X_train_sc.shape[1])
print(X_test_sc.shape[1])

# Creating our model's structure
model = Sequential()
#model.add(Dense(64, activation='relu', input_shape=(81,)))
model.add(Dense(64, activation='relu', input_shape=(X_train_sc.shape[1],)))
model.add(Dropout(0.18))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.15))
```

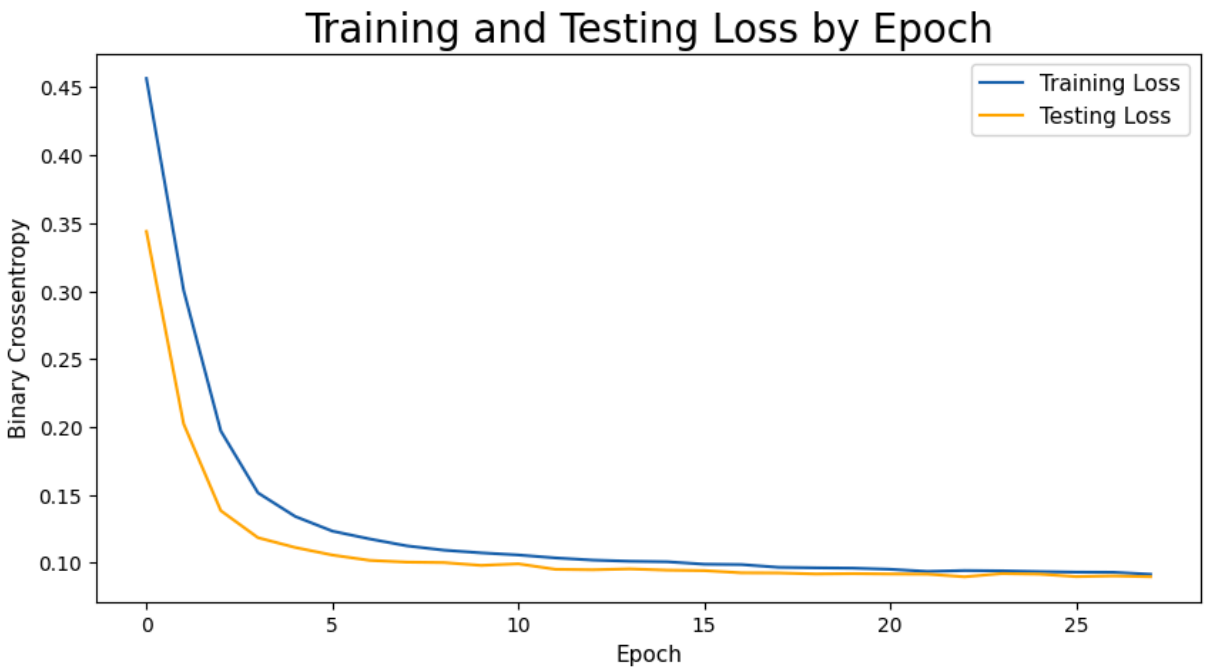
```
model.add(Dense(1, activation='sigmoid'))
es = EarlyStopping(monitor='val_loss', patience=5)

# Compiling the model
model.compile(loss='bce',
              optimizer='adam',
              metrics=['binary_accuracy'])

# Fitting the model
history = model.fit(X_train_sc,
                    y_train,
                    batch_size = 256,
                    validation_data =(X_test_sc, y_test),
                    epochs = 500,
                    verbose = 0,
                    callbacks=[es])
```

```
78
78
```

```
C:\Python312\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarnin
g: Do not pass an `input_shape`/`input_dim` argument to a layer. When using
Sequential models, prefer using an `Input(shape)` object as the first layer
in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

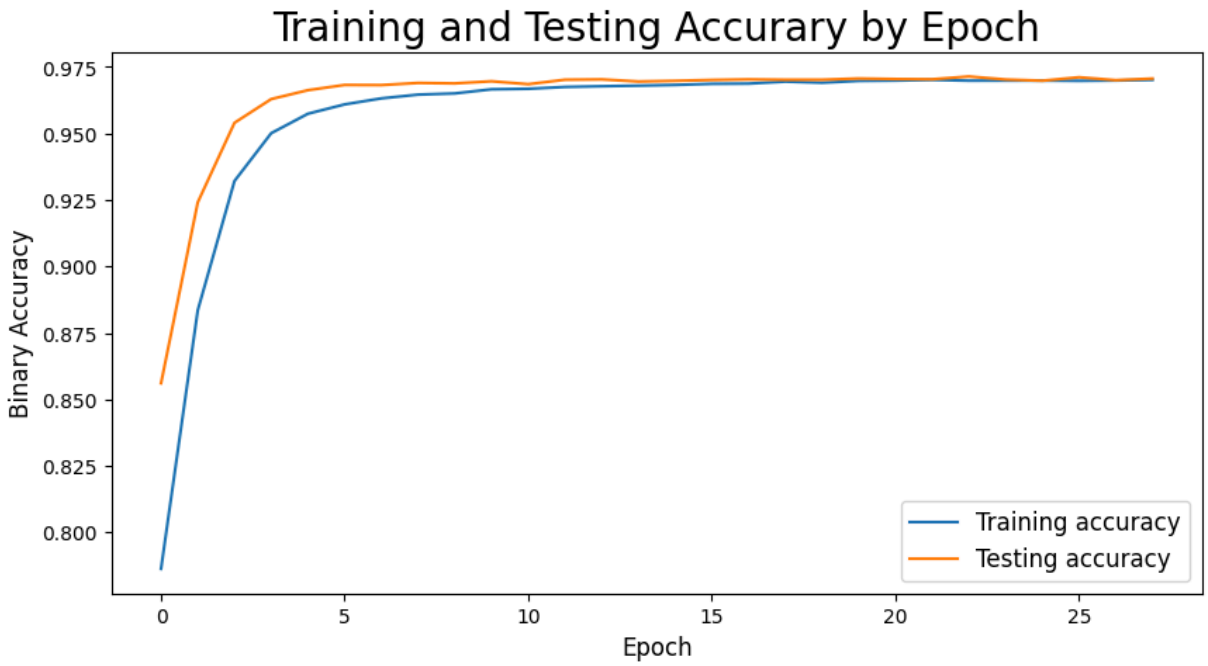
In [60]:

```
# Check out our train loss and test loss over epochs.
train_loss = history.history['loss']
test_loss = history.history['val_loss']

# Visualizing our training and testing loss by epoch
plt.figure(figsize=(10, 5))
plt.plot(train_loss, label='Training Loss', color='#185fad')
plt.plot(test_loss, label='Testing Loss', color='orange')
plt.title('Training and Testing Loss by Epoch', fontsize = 20)
plt.xlabel('Epoch', fontsize = 11)
plt.ylabel('Binary Crossentropy', fontsize = 11)
plt.legend(fontsize = 11);
```




The training and testing loss lines converge towards one another. This indicates that our model has stopped after approximately 34 epochs in order to avoid overfitting.

In [61]:
```
# Visualizing our training and testing accuracy by epoch:
plt.figure(figsize=(10, 5))
plt.plot(history.history['binary_accuracy'], label='Training accuracy')
plt.plot(history.history['val_binary_accuracy'], label='Testing accuracy')
plt.title('Training and Testing Accurary by Epoch', fontsize = 20)
plt.xlabel('Epoch', fontsize = 12)
plt.ylabel('Binary Accuracy', fontsize = 12)
plt.legend(fontsize = 12);
```



While neural network models are built by minimizing loss, we must also look at the accuracy since this is the metric we are evaluating our models on.

We see that both training and testing accuracy increase as the number of epochs increase. Both lines then converge towards one another, suggesting that our model is not overfit.

In [62]:
```
# Scoring
train_score = model.evaluate(X_train_sc,
                        y_train,
                        verbose=1)
test_score = model.evaluate(X_test_sc,
                        y_test,
                        verbose=1)
labels = model.metrics_names

print('')
print(f'Training Accuracy: {train_score[1]}')
print(f'Testing Accuracy: {test_score[1]}')
```

```
2985/2985 ━━━━━━━━━━━━━━━━━━━━ 2s 682us/step - binary_accuracy: 0.9734 - los
s: 0.0817
747/747 ━━━━━━━━━━━━━━━━━━━━ 1s 785us/step - binary_accuracy: 0.9704 - loss:
0.0903

Training Accuracy: 0.9731122255325317
Testing Accuracy: 0.9706424474716187
```

Our model's training score of 97.34% is very close to our 97.02% accuracy testing score meaning that our model is not overfit or underfit. The high testing score indicates that this model provides high predictive power.

# Logistic Regression and Decision Trees

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.impute import SimpleImputer

# Drop irrelevant columns for the prediction model
columns_to_drop = ['name', 'email', 'phone-number', 'credit_card', 'reservat
hotel_data_cleaned = df.drop(columns=columns_to_drop)

# Handle missing values
# Use median for numeric columns and most frequent for categorical columns
numeric_imputer = SimpleImputer(strategy='median')
categorical_imputer = SimpleImputer(strategy='most_frequent')

# Separate numeric and categorical columns
numeric_cols = hotel_data_cleaned.select_dtypes(include=['float64', 'int64']
categorical_cols = hotel_data_cleaned.select_dtypes(include=['object']).colu

# Apply imputers
```

```
hotel_data_cleaned[numeric_cols] = numeric_imputer.fit_transform(hotel_data_
hotel_data_cleaned[categorical_cols] = categorical_imputer.fit_transform(hot

# Encode categorical variables
encoder = LabelEncoder()
for col in categorical_cols:
    hotel_data_cleaned[col] = encoder.fit_transform(hotel_data_cleaned[col])

# Define target and features
X = hotel_data_cleaned.drop('is_canceled', axis=1)
y = hotel_data_cleaned['is_canceled']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rar

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

In [ ]:

```
# Remove any datetime columns
datetime_cols = hotel_data_cleaned.select_dtypes(include=['datetime64']).col
# Drop datetime columns if any are found
hotel_data_cleaned = hotel_data_cleaned.drop(columns=datetime_cols)

# Redefine X and y after updating columns
X = hotel_data_cleaned.drop('is_canceled', axis=1)
y = hotel_data_cleaned['is_canceled']

# Re-split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rar

# Retry training the models
logistic_model.fit(X_train, y_train)
y_pred_logistic = logistic_model.predict(X_test)

decision_tree_model.fit(X_train, y_train)
y_pred_decision_tree = decision_tree_model.predict(X_test)

# Calculate accuracy and classification reports for both models
logistic_accuracy = accuracy_score(y_test, y_pred_logistic)
decision_tree_accuracy = accuracy_score(y_test, y_pred_decision_tree)
logistic_report = classification_report(y_test, y_pred_logistic)
decision_tree_report = classification_report(y_test, y_pred_decision_tree)

logistic_accuracy, decision_tree_accuracy, logistic_report, decision_tree_re
```

## Logistic regression

Accuracy: 78.23%

## Classification Report:

Class 0 (not canceled): precision of 79%, recall of 88%, f1-score of 84% Class 1 (canceled): precision of 76%, recall of 61%, f1-score of 68%

## Decision Tree:

Accuracy: 85.61%

Classification Report: Class 0: precision of 89%, recall of 88%, f1-score of 89%
Class 1: precision of 80%, recall of 81%, f1-score of 81%

Decision Tree maintained a superior performance, especially in terms of precision and f1-score for both classes

In [ ]:

```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay, roc_curve, auc

# Confusion Matrices for both models
fig, ax = plt.subplots(1, 2, figsize=(14, 6))
ConfusionMatrixDisplay.from_estimator(logistic_model, X_test, y_test, ax=ax[
ax[0].set_title("Logistic Regression - Confusion Matrix")
ConfusionMatrixDisplay.from_estimator(decision_tree_model, X_test, y_test, a
ax[1].set_title("Decision Tree - Confusion Matrix")
plt.show()

# Classification Report Bar Graphs
import numpy as np

# Extract values for plotting
logistic_metrics = [float(value) for value in logistic_report.split() if val
decision_tree_metrics = [float(value) for value in decision_tree_report.spli

metrics_labels = ["Precision (Class 0)", "Recall (Class 0)", "F1-score (Clas
                  "Precision (Class 1)", "Recall (Class 1)", "F1-score (Clas

# Create bar plot for classification report comparison
x = np.arange(len(metrics_labels))
width = 0.35

fig, ax = plt.subplots(figsize=(12, 6))
ax.bar(x - width/2, logistic_metrics, width, label='Logistic Regression')
ax.bar(x + width/2, decision_tree_metrics, width, label='Decision Tree')
ax.set_xlabel("Metrics")
ax.set_ylabel("Scores")
ax.set_title("Classification Report Comparison")
ax.set_xticks(x)
ax.set_xticklabels(metrics_labels, rotation=45)
ax.legend()
plt.show()

# ROC Curves
y_test_binary = y_test.apply(lambda x: 1 if x == 1 else 0)  # Convert y_test
logistic_fpr, logistic_tpr, _ = roc_curve(y_test_binary, logistic_model.prec
decision_tree_fpr, decision_tree_tpr, _ = roc_curve(y_test_binary, decision_

logistic_auc = auc(logistic_fpr, logistic_tpr)
decision_tree_auc = auc(decision_tree_fpr, decision_tree_tpr)

plt.figure(figsize=(10, 6))
plt.plot(logistic_fpr, logistic_tpr, label=f'Logistic Regression (AUC = {log
```

```
plt.plot(decision_tree_fpr, decision_tree_tpr, label=f'Decision Tree (AUC =
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve Comparison")
plt.legend()
plt.show()
```

## Confusion Matrices

Show the distribution of true positives, true negatives, false positives, and false negatives for each model, providing a clear view of how each model performs in correctly and incorrectly classifying cancellations.

Classification Report Comparison: A bar chart comparing precision, recall, and F1-score for both classes (0 and 1) across Logistic Regression and Decision Tree models, highlighting differences in metric scores.

## ROC Curve

- This curve illustrates the trade-off between true positive rate (sensitivity) and false positive rate for both models.

The Decision Tree model achieves a higher Area Under the Curve (AUC), indicating better overall performance in distinguishing between canceled and non-canceled bookings.

## Confusion Matrices

Logistic Regression and Decision Tree both classify the data into four classes:

- True Negatives (TN): Correctly predicting non-canceled bookings.
- False Positives (FP): Incorrectly predicting a canceled booking when it wasn’t.
- False Negatives (FN): Incorrectly predicting a non-canceled booking when it was canceled.
- True Positives (TP): Correctly predicting canceled bookings.

## Observations

The Decision Tree model has higher values in the TP and TN cells, indicating better accuracy in predicting both canceled and non-canceled bookings compared to Logistic Regression.

The False Positives and False Negatives are fewer in the Decision Tree model, showing that it’s making fewer errors overall.

## Classification Report Comparison (Precision, Recall, F1-Score)

## Precision

Measures how many of the bookings predicted as canceled (Class 1) were actually canceled. A high precision for Class 1 in the Decision Tree model shows it is better at avoiding false positives (incorrectly labeling non-canceled as canceled).

### Recall

Measures the model’s ability to find all actual canceled bookings. Higher recall for Class 1 in the Decision Tree model indicates it’s capturing more actual cancellations than Logistic Regression.

### F1-Score

Balances precision and recall. The Decision Tree consistently achieves a higher F1-score for both classes, suggesting that it has a more balanced performance in terms of precision and recall.

The Decision Tree model shows higher precision, recall, and F1-score, especially for the canceled bookings class, indicating it’s better at identifying cancellations accurately and balancing between precision and recall.

### ROC Curve and AUC (Area Under the Curve)

The ROC Curve plots the True Positive Rate (sensitivity) against the False Positive Rate, providing a view of each model’s classification threshold trade-offs. The AUC (Area Under the Curve) is a single score summarizing the model's ability to distinguish between classes (canceled vs. non-canceled). Higher AUC values suggest a better model.

The Decision Tree model’s AUC is higher than the Logistic Regression’s, suggesting it more effectively distinguishes between canceled and non-canceled bookings across various thresholds. Logistic Regression’s curve is closer to the diagonal line, indicating it has less discrimination ability between classes than the Decision Tree.

Overall, the Decision Tree model outperforms Logistic Regression across all metrics:

- It has better accuracy in predicting both classes, as seen in the confusion matrix.
- Achieves higher precision, recall, and F1-scores, especially in detecting cancellations.
- Shows a better ROC and higher AUC, meaning it’s more robust in distinguishing between canceled and non-canceled bookings.

# Support Vector Machines (SVM)

In [ ]:
```
from sklearn.svm import SVC

# Initialize the SVM model with a probability output for ROC curve compatibi
svm_model = SVC(probability=True, random_state=42)

# Train the SVM model
svm_model.fit(X_train, y_train)

# Make predictions
y_pred_svm = svm_model.predict(X_test)

# Calculate accuracy and classification report for SVM model
svm_accuracy = accuracy_score(y_test, y_pred_svm)
svm_report = classification_report(y_test, y_pred_svm)

# Generate ROC curve for SVM model
svm_fpr, svm_tpr, _ = roc_curve(y_test_binary, svm_model.predict_proba(X_tes
svm_auc = auc(svm_fpr, svm_tpr)

svm_accuracy, svm_report, svm_auc
```

The results provided are metrics to assess the performance of the Support Vector Machine (SVM) model on a classification task. Let's break down each part:

## **Overall Accuracy** (0.7203484476211749)

- **Accuracy** represents the proportion of correct predictions made by the model out of all predictions.
- In this case, the accuracy of **0.72** (or 72%) means the SVM model correctly classified about 72% of the samples in the test set.

## Classification Report

- This report provides detailed metrics for each class (e.g., `0.0` and `1.0`).
- It includes **precision**, **recall**, and **f1-score** for each class, as well as overall averages.

**Metrics Breakdown:**

- **Precision**: The proportion of true positive predictions out of all positive predictions made for a class.
  - For class `0.0`: Precision is **0.72**, meaning 72% of all samples predicted as `0` were correctly classified as `0`.
  - For class `1.0`: Precision is **0.73**, meaning 73% of all samples predicted as `1` were correct.

- **Recall**: The proportion of true positive predictions out of all actual positives for a class.
  - For class `0.0`: Recall is **0.91**, meaning 91% of all actual `0` samples were correctly identified as `0`.
  - For class `1.0`: Recall is **0.40**, meaning only 40% of actual `1` samples were correctly classified as `1`.
- **F1-Score**: The harmonic mean of precision and recall, giving an overall effectiveness score for each class.
  - For class `0.0`: F1-score is **0.80**.
  - For class `1.0`: F1-score is **0.52**.

**Averages**:

- **Macro Average**: Averages metrics for each class equally, without considering class imbalance.
  - Macro avg F1-score is **0.66**, which suggests moderate balance across classes but indicates room for improvement, especially in class `1.0`.
- **Weighted Average**: Averages metrics considering class support (number of samples in each class).
  - Weighted avg F1-score is **0.70**, indicating the model performs slightly better when class distribution is considered.

## **AUC Score** (0.7507153408321707)

- **AUC** (Area Under the Curve) measures the ability of the model to distinguish between classes, with values ranging from 0.5 (no better than random) to 1.0 (perfect classification).
- In this case, the **AUC score of 0.75** indicates that the SVM model has a fair level of separability for distinguishing between classes but is not highly accurate.

The model has a moderate **overall accuracy of 72%** but shows an imbalance in recall between classes: It performs better in detecting class `0.0` (with high recall of 91%) compared to class `1.0` (only 40% recall). The **AUC of 0.75** suggests that the model can differentiate between classes reasonably well, though there is room for improvement, particularly in improving recall for class `1.0`.

# Ensemble Learning

## Bagging

In [ ]:
```python
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score

# Initialize the Bagging classifier with Decision Trees as base models
bagging_model = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=50,  # Number of base models
    random_state=42
)

# Train and evaluate using cross-validation
bagging_scores = cross_val_score(bagging_model, X_train, y_train, cv=5)
print("Bagging Model Accuracy:", bagging_scores.mean())
```

In [ ]:
```python
# Random Forest
from sklearn.ensemble import RandomForestClassifier

# Initialize RandomForest classifier
random_forest_model = RandomForestClassifier(
    n_estimators=100,  # Number of trees
    max_depth=15,  # Limit depth of each tree for faster performance
    random_state=42
)

# Train and evaluate using cross-validation
rf_scores = cross_val_score(random_forest_model, X_train, y_train, cv=5)
print("Random Forest Model Accuracy:", rf_scores.mean())
```

# Stacking

In [ ]:
```python
# from sklearn.ensemble import RandomForestClassifier, GradientBoostingClass
# from sklearn.linear_model import LogisticRegression
# from sklearn.svm import SVC

# Define base models for stacking
# estimators = [
#     ('rf', RandomForestClassifier(n_estimators=100, random_state=42)),
#     ('gb', GradientBoostingClassifier(n_estimators=100, random_state=42)),
#     ('svc', SVC(probability=True, random_state=42))
#]

# Define meta-model (Logistic Regression)
#stacking_model = StackingClassifier(
#    estimators=estimators,
#    final_estimator=LogisticRegression(),
#    cv=5
#)

# Train and evaluate using cross-validation
# stacking_scores = cross_val_score(stacking_model, X_train, y_train, cv=5)
# print("Stacking Model Accuracy:", stacking_scores.mean())
```

Stacking took 400+ minutes to run due to SVM model combined with Cross-validation

# Boosting

In [ ]:
```
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, AdaB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import xgboost as xgb
import lightgbm as lgb

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran

# Dictionary to store models and their results
results = {}

# AdaBoostClassifier
ada_clf = AdaBoostClassifier(n_estimators=100, random_state=42)
ada_clf.fit(X_train, y_train)
ada_pred = ada_clf.predict(X_test)
results['AdaBoost'] = accuracy_score(y_test, ada_pred)
```

In [ ]:
```
# GradientBoostingClassifier
gb_clf = GradientBoostingClassifier(n_estimators=100, random_state=42)
gb_clf.fit(X_train, y_train)
gb_pred = gb_clf.predict(X_test)
results['GradientBoosting'] = accuracy_score(y_test, gb_pred)
```

In [ ]:
```
# XGBoost
xgb_clf = xgb.XGBClassifier(n_estimators=100, use_label_encoder=False, eval_
xgb_clf.fit(X_train, y_train)
xgb_pred = xgb_clf.predict(X_test)
results['XGBoost'] = accuracy_score(y_test, xgb_pred)
```

In [ ]:
```
# LightGBM
lgb_clf = lgb.LGBMClassifier(n_estimators=100, random_state=42)
lgb_clf.fit(X_train, y_train)
lgb_pred = lgb_clf.predict(X_test)
results['LightGBM'] = accuracy_score(y_test, lgb_pred)
```

In [ ]:
```
# Print the results
for model, accuracy in results.items():
    print(f"{model} Accuracy: {accuracy:.4f}")
```

AdaBoost achieved an accuracy of about 85.17%, which is the lowest among the models tested. Gradient Boosting performed better than AdaBoost, with an accuracy of 90.61%. XGBoost achieved the highest accuracy among the models, at 96.49%. LightGBM's accuracy is slightly lower than XGBoost's but still very high at 95.54%.

XGBoost (96.49%) slightly outperforms LightGBM (95.54%), indicating that XGBoost's optimizations, like regularization, might give it a slight edge in capturing complex patterns in this data.

Based on the accuracy results, XGBoost is likely the best choice. However, LightGBM would also be a good alternative, especially if the training speed or the computational efficiency is a concern.

# Model Selection

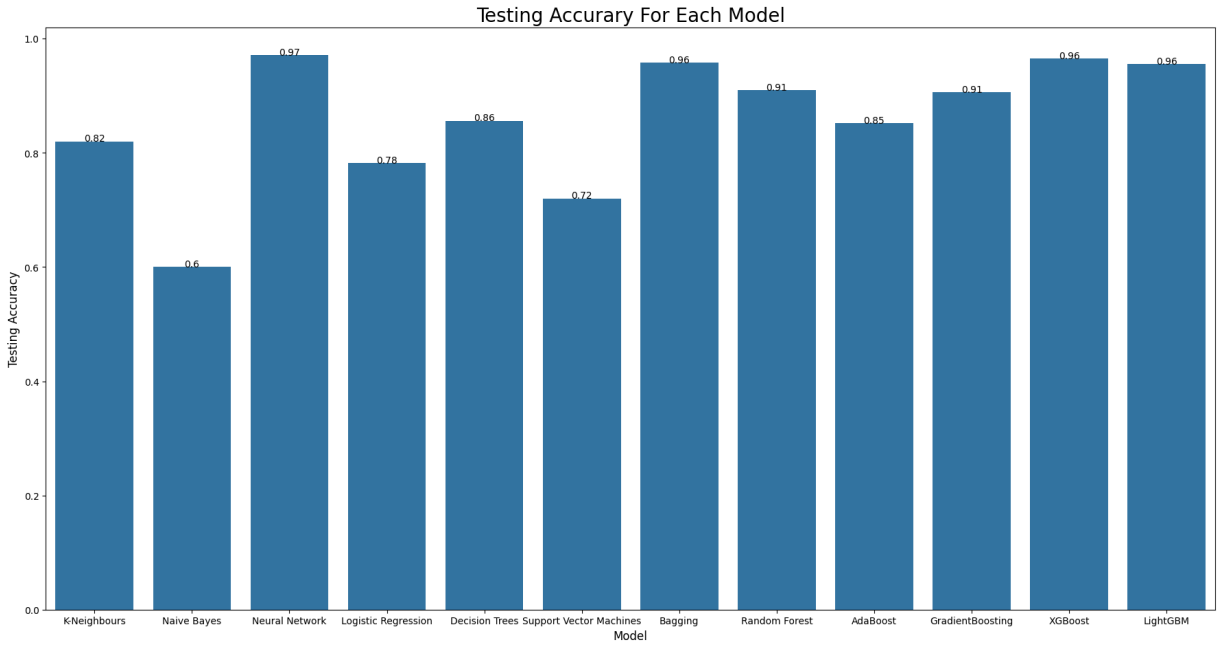
The goal of the project is to analyze the model that provides the highest accuracy for our target. More specifically, we are looking for the model with the highest testing accuracy.

In [63]:
```
# Creating two dataframes to compare our models' performances:
predictive_model_scores = pd.DataFrame(data=[
                                            ('K-Neighbours',0.81, 0.82), ('
                                            ('Neural Network',0.9745, 0.971
                                            ('Decision Trees',0.8561, 0.856
                                            ('Bagging', 0.958, 0.958), ('Ra
                                            ('AdaBoost',0.8517, 0.8517), ('
                                            ('XGBoost',0.9649, 0.9649), ('L
                                           ],
                                      columns=['model', 'training_accuracy'

predictive_model_scores
```

Out[63]:

| | model | training_accuracy | testing_accuracy |
|---|---|---|---|
| **0** | K-Neighbours | 0.8100 | 0.8200 |
| **1** | Naive Bayes | 0.5990 | 0.6000 |
| **2** | Neural Network | 0.9745 | 0.9714 |
| **3** | Logistic Regression | 0.7823 | 0.7823 |
| **4** | Decision Trees | 0.8561 | 0.8561 |
| **5** | Support Vector Machines | 0.7300 | 0.7200 |
| **6** | Bagging | 0.9580 | 0.9580 |
| **7** | Random Forest | 0.9098 | 0.9098 |
| **8** | AdaBoost | 0.8517 | 0.8517 |
| **9** | GradientBoosting | 0.9061 | 0.9061 |
| **10** | XGBoost | 0.9649 | 0.9649 |
| **11** | LightGBM | 0.9554 | 0.9554 |

In [64]:
```
# Visualizing testing accuracy of each model:
plt.style.use('default')
plt.figure(figsize=(22,11))
p=sns.barplot(x='model', y='testing_accuracy', data=predictive_model_scores)
plt.title('Testing Accurary For Each Model', fontsize = 20)
plt.xlabel('Model', fontsize = 12)
plt.ylabel('Testing Accuracy', fontsize = 12)
for index, row in predictive_model_scores.iterrows():
    p.text(x=row.name, y=row.testing_accuracy, s=round(row.testing_accuracy,
```



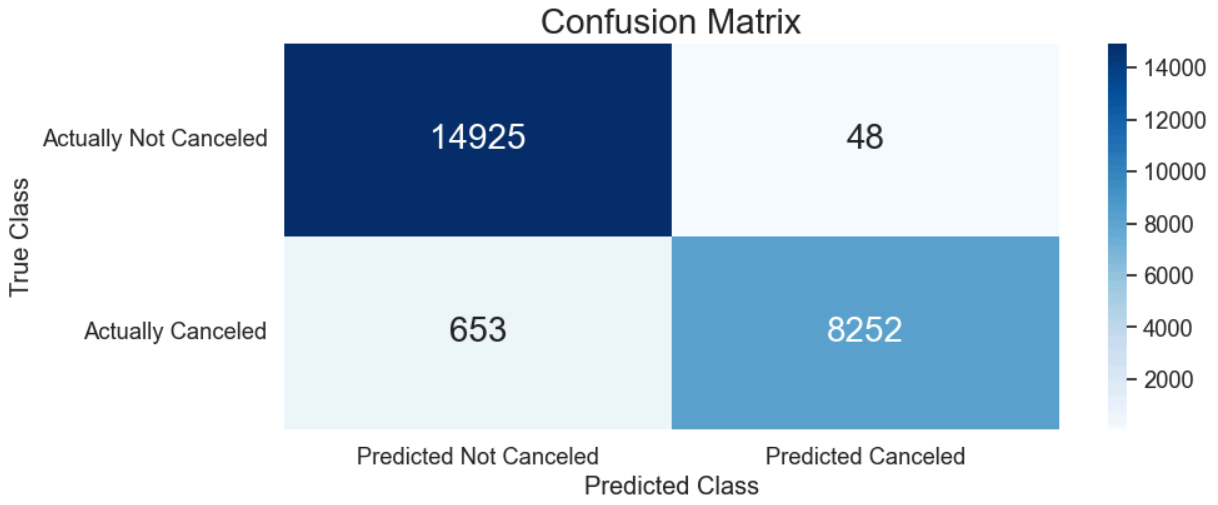

The model providing the highest test accurary score is our Neural Network model. Although our main goal is predictability, neural networks are "black boxes" and having a model that is interpretable can bring valuable insights. Indeed, we mentioned the importance of engaging with the reasons why customers are cancelling. As a result, we will also evaluate our Logistic Regression model for interpretability purposes.

# Confusion Matrix

In [66]:
```
# Making predictions
nn_probabilities = model.predict(X_test_sc)

# Convert the probabilities in classes (0 or 1)
nn_predictions = (nn_probabilities > 0.5).astype("int32")

# Creating confusion matrix
cm = confusion_matrix(y_test, nn_predictions)

# putting the matrix a dataframe form
cm_df = pd.DataFrame(cm, index=['Actually Not Canceled', 'Actually Canceled'
                  columns=['Predicted Not Canceled', 'Predicted Canceled'])
```

```
747/747 ━━━━━━━━━━━━━━━━━━━━ 1s 713us/step
```

In [67]:
```
# visualizing the confusion matrix
sns.set(font_scale=1.2)
plt.figure(figsize=(10,4))

sns.heatmap(cm, annot=True, fmt='g', cmap="Blues",xticklabels=cm_df.columns,
plt.title("Confusion Matrix", size=20)
plt.xlabel('Predicted Class')
plt.ylabel('True Class');
```




In [68]:
```
# True Positives:
TP = 8252
# True Negatives:
TN = 14925
# False Posititives:
FP = 48
# False Negatives:
FN = 653
total = 8252+653+14925+48

print(f'Correctly classified: {np.round((TP+TN)/total*100)}%')
print(f'Canceled bookings correctly classified: {np.round(TP/(TP+FN)*100)}%'
print(f'Not canceled bookings correctly classified: {np.round(TN/(TN+FP)*100
print(f'Bookings predicted canceled that are actually canceled: {np.round(TP
print(f'Bookings predicted not canceled that are actually not canceled: {np.
```

```
Correctly classified: 97.0%
Canceled bookings correctly classified: 93.0%
Not canceled bookings correctly classified: 100.0%
Bookings predicted canceled that are actually canceled: 99.0%
Bookings predicted not canceled that are actually not canceled: 96.0%
```

**Overall Model Accuracy (97.0%)**

The model correctly classified 97% of all bookings, indicating high overall accuracy. This suggests it is effective at distinguishing between canceled and non-canceled bookings.

**Correctly Classified Cancellations (93.0%)**
This metric shows that the model correctly identified 93% of bookings that were actually canceled. In other words, 93% of bookings that should have been classified as canceled were indeed predicted as such. This reflects a good sensitivity or "recall" for the cancellation class, though it also indicates that a small percentage of cancellations were incorrectly classified as non-canceled (false negatives).

**Correctly Classified Non-Canceled Bookings (100.0%)**
The model correctly classified all bookings that were not canceled, achieving 100% recall for the non-canceled class. This means the model is very effective at recognizing bookings that will indeed proceed as planned and did not produce any false positives for this class.

**Predicted Cancellations that Are Actually Canceled (99.0%)**
This metric shows that 99% of bookings predicted as canceled were actually canceled, indicating an excellent "precision" for the cancellation class.

**Predicted Non-Cancellations that Are Actually Not Canceled (96.0%)**
For bookings the model predicted as not canceled, 96% were indeed not canceled. This high percentage suggests good precision in predicting non-cancelled bookings, reducing the risk of incorrectly classifying bookings that should proceed as cancellations.

# Conclusion

We identified a Neural Network as the most accurate model for predicting booking cancellations, achieving an impressive 97% accuracy. This accuracy enables hotels to forecast occupancy more effectively, optimize resource allocation, and potentially increase revenue by planning for guest arrivals and departures with greater precision.

The model’s strength lies in its ability to correctly identify 93% of actual cancellations and achieve 100% accuracy for confirmed bookings, ensuring minimal disruptions and a high level of preparedness for guest arrivals.

With 99% precision in predicting cancellations and 96% precision for confirmed bookings, the model provides hotels with strong confidence in its predictions, effectively reducing the risk of overbooking or missed opportunities for replacement bookings. This high predictive accuracy enables hotels to manage bookings proactively, enhance operational efficiency, and improve overall guest satisfaction by reducing unexpected outcomes.

Ultimately, while the model is highly effective, small adjustments could be made to further reduce the 0.4% and 2.4% misclassification rates, making it an even more precise tool for occupancy management and revenue optimization.

# References


- https://www.geeksforgeeks.org/naive-bayes-classifiers/
- https://medium.com/@mertsukrupehlivan/exploring-the-k-nearest-neighbors-algorithm-in-machine-learning-e3b90a3c9e0d
- https://xgboost.readthedocs.io/en/stable/install.html#conda


In [ ]: