Table of Contents

F. Romaric Berger

Base on the work on the EDA, we can work on **recommending actions** to the company management. We can also use models such as ARIMA to **forecast future incomes**.

Here are some extra information to take into account.

1 Summary of Problem Statement

1.1 ❓ How could Olist improve its profit ❓

P&L Rules
Revenues
Sales fees: Olist takes a 10% cut on the product price (excl. freight) of each order delivered
Subscription fees: Olist charges 80 BRL by month per seller

IT costs
Olist's total cumulated IT Costs scale with the square root of the total number of sellers that have ever joined the platform, as well as with the square root of the total cumulated number of items that were ever sold.

$$IT\_costs = \alpha * \sqrt{n\_sellers} + \beta * \sqrt{n\_items}$$

Olist's data team gave us the following values for these scaling parameters:

$\alpha = 3157.27$

$\beta = 978.23$

💡 Both the number of sellers to manage and the number of sales transaction are costly for IT systems.
💡 Yet square roots reflect scale-effects: IT-system are often more efficient as they grow bigger.
💡 Alpha > Beta means that Olist has a lower IT Cost with few sellers selling a lot of items rather than the opposite

with 1000 sellers and a total of 100 items sold, the total IT cost accumulates to 109,624 BRL
with 100 sellers and a total of 1000 items sold, the total IT cost accumulates to 62,507 BRL
Finally, The IT department also told you that since the birth of the marketplace, cumulated IT costs have amounted to 500,000 BRL.

Reputation cost

Estimated **reputation costs** of orders with bad reviews (<= 3 stars)

💡 In the long term, bad customer experience has business implications: low repeat rate, immediate customer support cost, refunds or unfavorable word of mouth communication. We make an assumption about the monetary cost for each bad review:

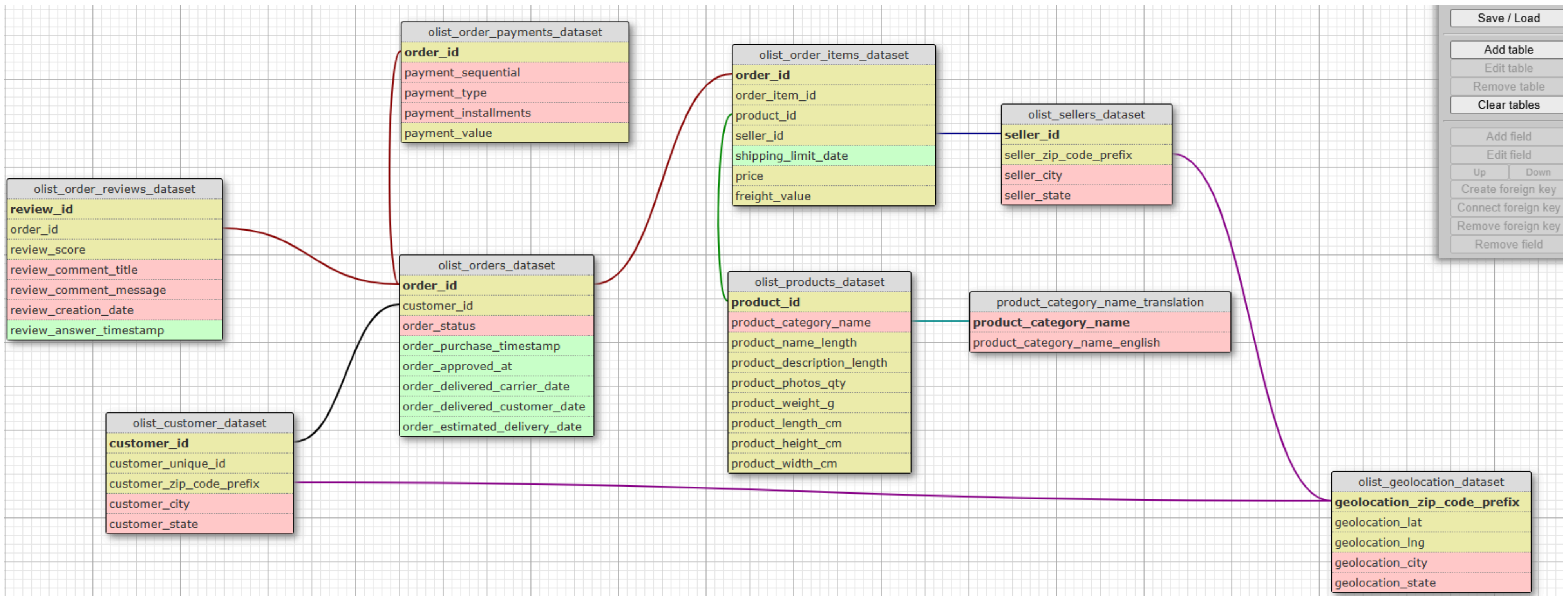
```
# review_score: cost(BRL)
{'1 star': 100
'2 stars': 50
'3 stars': 40
'4 stars': 0
```

In [1]:
```
from IPython.display import Image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from dateutil.relativedelta import relativedelta
from statsmodels.tsa.seasonal  import seasonal_decompose
from statsmodels.tsa.stattools  import adfuller, kpss
from statsmodels.graphics.tsaplots  import plot_acf, plot_pacf
from pmdarima.arima import auto_arima
from sklearn.metrics  import mean_absolute_error, mean_squared_error
from statsmodels.tsa.statespace.sarimax  import SARIMAX
```
executed in 3.95s, finished 10:56:24 2025-05-18

2 Actual situation

In [2]:
```
Image('Image/olist_erd_details.png')
```
executed in 26ms, finished 10:56:24 2025-05-18

Out[2]:



In [3]:
```
# Load CSVs
orders = pd.read_csv('data/olist_orders_dataset.csv')
customers = pd.read_csv('data/olist_customers_dataset.csv')
reviews = pd.read_csv('data/olist_order_reviews_dataset.csv')
order_items = pd.read_csv('data/olist_order_items_dataset.csv')
products = pd.read_csv('data/olist_products_dataset.csv')
translation = pd.read_csv('data/product_category_name_translation.csv')
sellers = pd.read_csv('data/olist_sellers_dataset.csv')

# Merge datasets
data = orders.merge(customers, on='customer_id', how='left') \
    .merge(order_items, on='order_id', how='left') \
    .merge(sellers, on='seller_id', how='left') \
    .merge(reviews, on='order_id', how='left') \
    .merge(products, on='product_id', how='left') \
    .merge(translation, on='product_category_name', how='left')

# Transform order_purchase_timestamp in Datetime
data['order_purchase_timestamp'] = pd.to_datetime(data['order_purchase_timestamp'])

# Cleaning
# Drop columns not inherently related to the objectives, especially in the context of EDA
# drop columns with too many missing values

data = data.drop(columns=['order_delivered_carrier_date','order_approved_at', 'review_id','review_comment_title',
                         'review_answer_timestamp', 'product_name_lenght', 'product_description_lenght',
                         'product_photos_qty','product_weight_g', 'product_length_cm', 'product_height_cm','product_width_cm',
                         'product_category_name', 'customer_id', 'order_delivered_customer_date',
                         'order_estimated_delivery_date','seller_city','seller_zip_code_prefix', 'seller_state','customer_city',
```
executed in 3.15s, finished 10:56:27 2025-05-18

In [4]:
```
data.info()
```
executed in 144ms, finished 10:56:27 2025-05-18

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 114092 entries, 0 to 114091
Data columns (total 15 columns):
 #   Column                         Non-Null Count   Dtype
---  ------                         --------------   -----
 0   order_id                       114092 non-null  object
 1   order_status                   114092 non-null  object
 2   order_purchase_timestamp       114092 non-null  datetime64[ns]
 3   customer_unique_id             114092 non-null  object
 4   customer_zip_code_prefix       114092 non-null  int64
 5   customer_state                 114092 non-null  object
 6   order_item_id                  113314 non-null  float64
 7   product_id                     113314 non-null  object
 8   seller_id                      113314 non-null  object
 9   shipping_limit_date            113314 non-null  object
 10  price                          113314 non-null  float64
 11  freight_value                  113314 non-null  float64
 12  review_score                   113131 non-null  float64
 13  review_creation_date           113131 non-null  object
 14  product_category_name_english  111678 non-null  object
dtypes: datetime64[ns](1), float64(4), int64(1), object(9)
memory usage: 13.1+ MB
```

In [5]:
```
data.isna().sum()
```
executed in 117ms, finished 10:56:27 2025-05-18

Out[5]:
```
order_id                            0
order_status                        0
order_purchase_timestamp            0
customer_unique_id                  0
customer_zip_code_prefix            0
customer_state                      0
order_item_id                     778
product_id                        778
seller_id                         778
shipping_limit_date               778
price                             778
freight_value                     778
review_score                      961
review_creation_date              961
product_category_name_english    2414
dtype: int64
```

2.1 Answer the problem

2.1.1 Maximaze Revenue

Olist's revenue streams:

Sales Fees: 10% of product price (excluding freight) for delivered orders.

Subscription Fees: 80 BRL per month per seller.

In [6]:
```
## Total Sales Fees

# Only consider delivered orders
delivered_orders = data[data['order_status'] == 'delivered']

# Compute total sales fee (10% of price, not freight)
sales_fee = 0.10 * delivered_orders['price'].sum()
print(f'The incomes from the 10% cut on all sales made through the Olist platform is BRL:{sales_fee:,.2f}')
```
executed in 54ms, finished 10:56:27 2025-05-18

```
The incomes from the 10% cut on all sales made through the Olist platform is BRL:1,327,983.66
```

In [7]:
```
## Total Subscription Revenue

# Unique sellers
unique_sellers = data['seller_id'].nunique()

seller_orders = order_items.merge(orders[['order_id', 'order_purchase_timestamp']], on='order_id', how='left')
seller_orders['order_purchase_timestamp'] = pd.to_datetime(seller_orders['order_purchase_timestamp'])


seller_lifespan = seller_orders.groupby('seller_id')['order_purchase_timestamp'] \
    .agg(['min', 'max']) \
    .rename(columns={'min': 'first_sale', 'max': 'last_sale'})

# Function to calculate number of months between two dates
def month_diff(d1, d2):
    rd = relativedelta(d2, d1)
    return rd.years * 12 + rd.months + (1 if rd.days > 0 else 0)

seller_lifespan['months_active'] = seller_lifespan.apply(lambda row: month_diff(row['first_sale'], row['last_sale']), axis=1)

# Subscription revenue = 80 BRL per active month per seller
subscription_revenue = (seller_lifespan['months_active'] * 80).sum()
print(f"Total Subscription Revenue: BRL {subscription_revenue:,.2f}")
```
executed in 445ms, finished 10:56:28 2025-05-18

```
Total Subscription Revenue: BRL 1,532,320.00
```

2.1.2 Total Revenue

In [8]:
```
total_revenue = sales_fee + subscription_revenue
print(f"The Total Revenue of Olist is BRL {total_revenue:,.2f}")
```
executed in 7ms, finished 10:56:28 2025-05-18

```
The Total Revenue of Olist is BRL 2,860,303.66
```

2.1.3 Calculate IT Costs

In [9]:
```
alpha = 3157.27
beta = 978.23

n_sellers = data['seller_id'].nunique()
n_items = data['order_item_id'].count() * 0.92

it_costs = alpha * np.sqrt(n_sellers) + beta * np.sqrt(n_items)
print(f"The Total IT costs of Olist is BRL {it_costs:,.2f}")
```
executed in 14ms, finished 10:56:28 2025-05-18

```
The Total IT costs of Olist is BRL 491,494.47
```

The IT department told us that since the birth of the marketplace, cumulated IT costs have amounted to 500,000 BRL. So it confirms our findings.
Note: We already realized that 95 % of the revenue comes from half of the categories, which represent 92% of the products.
The new IT Cost would be BRL 491,494.47, so we won't look further in the idea of cutting products.

2.1.4 Calculate Reputation cost

In [10]:
```
# How does the review scoring works, are these float or int?
data['review_score'].nunique()
```
executed in 9ms, finished 10:56:28 2025-05-18

Out[10]:
```
5
```

In [11]:
```
def review_cost(x):
    if pd.isna(x):
        return 0
    if x == 3:
        return 40
    elif x == 2:
        return 50
    elif x == 1:
        return 100
    else:
        return 0
```
executed in 7ms, finished 10:56:28 2025-05-18

In [12]:
```
data['review_cost']=data['review_score'].apply(lambda x: review_cost(x))
reputation_cost = data['review_cost'].sum()
```
executed in 141ms, finished 10:56:28 2025-05-18

In [13]:
```
print(f'Olist reputation cost is BRL {reputation_cost:,.2f}')
```
executed in 7ms, finished 10:56:28 2025-05-18

```
Olist reputation cost is BRL 2,053,340.00
```

2.1.5 Profit = Revenue - IT Costs - Reputation score

In [14]:
```
profit = total_revenue - it_costs - reputation_cost
print(f'Olist profit is BRL {profit:,.2f}')
```
executed in 8ms, finished 10:56:28 2025-05-18

```
Olist profit is BRL 315,469.19
```

3 Hypothesis

3.1 *What if we remove the sellers that have bad reviews??*

3.1.1 Identify Performant sellers

In [76]:
```
s_unique = data['seller_id'].nunique()
print(f'The number of unique seller is {s_unique}.')
```
executed in 43ms, finished 12:05:44 2025-05-18

```
The number of unique seller is 3095
```

In [16]:
```
# Identify good sellers
good_sellers = data.groupby('seller_id')['review_score'].mean() > 3.0
good_seller_ids = good_sellers[good_sellers].index
```
executed in 22ms, finished 10:56:28 2025-05-18

In [77]:
```
n_good_sellers = good_seller_ids.nunique()
print(f'The number of seller with a review score mean superior to 3 is {n_good_sellers}.')
```
executed in 15ms, finished 12:07:33 2025-05-18

```
The number of seller with a review score mean superior to 3 is 2645.
```

In [18]:
```
# Filter Data
good_data = data[data['seller_id'].isin(good_seller_ids)]
```
executed in 56ms, finished 10:56:28 2025-05-18

3.1.2 Calculate incomes

In [19]:
```
#Delivered orders only
delivered_good_data = good_data[good_data['order_status'] == 'delivered']
```
executed in 57ms, finished 10:56:28 2025-05-18

```
In [20]:
#Sales Fee
good_sales_fee = 0.10 * delivered_good_data['price'].sum()
print(f'The incomes from the 10% cut on all sales made through the Olist platform by Good Sellers Only: BRL {sales_fee:,.2f}')

# Subscription revenue
good_seller_lifespan = seller_lifespan[seller_lifespan.index.isin(good_seller_ids)]
good_subscription_revenue = (good_seller_lifespan['months_active'] * 80).sum()

print(f"Subscription Revenue from Good Sellers Only: BRL {good_subscription_revenue:,.2f}")

gs_total_revenue = good_sales_fee + good_subscription_revenue
print(f"The Total Revenue of Olist is BRL {gs_total_revenue:,.2f}")
```

executed in 14ms, finished 10:56:28 2025-05-18

```
The incomes from the 10% cut on all sales made through the Olist platform by Good Sellers Only: BRL 1,327,983.66
Subscription Revenue from Good Sellers Only: BRL 1,443,440.00
The Total Revenue of Olist is BRL 2,735,287.43
```

3.1.3 Calculate IT Cost

```
In [21]:
gs_it_costs = alpha * np.sqrt(n_good_sellers) + beta * np.sqrt(n_items)
print(f"The Total IT costs of Olist is BRL {gs_it_costs:,.2f}")
```

executed in 8ms, finished 10:56:28 2025-05-18

```
The Total IT costs of Olist is BRL 478,223.96
```

3.1.4 Reputation Cost

The reputation cost now amount to 0

3.1.5 Olist profits with performing sellers only

```
In [22]:
gs_profit = gs_total_revenue - gs_it_costs
print(f'Olist profit with performing sellers only is BRL {gs_profit:,.2f}')
```

executed in 6ms, finished 10:56:28 2025-05-18

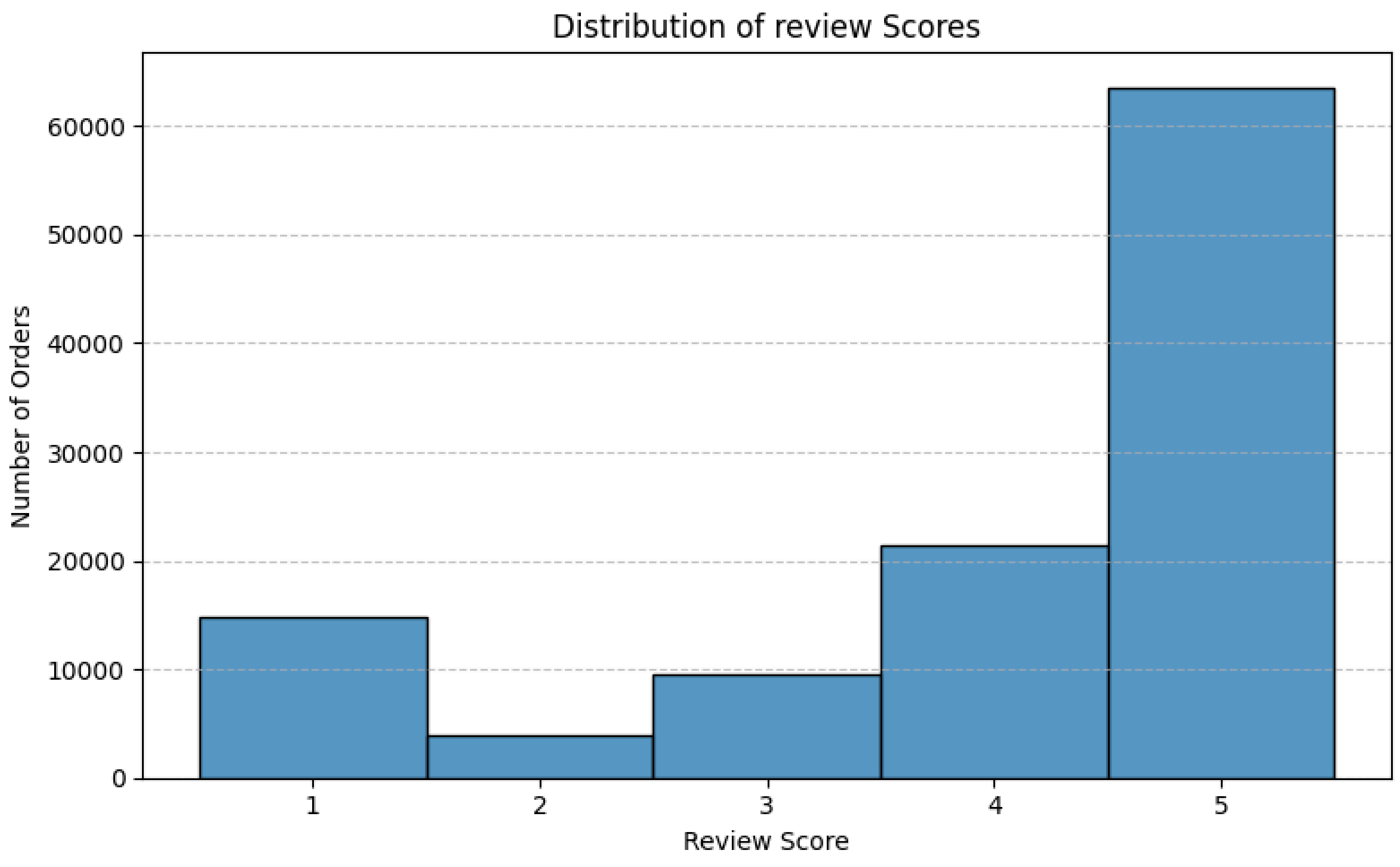
```
Olist profit with performing sellers only is BRL 2,257,063.47
```

3.2 Observation

Intuitively, the cost logic of the impact of reputation cost on Olist's profitability seems disproportionate, a visual analysis can help assessing whether the cost logic is proportionate or possibly inflated.

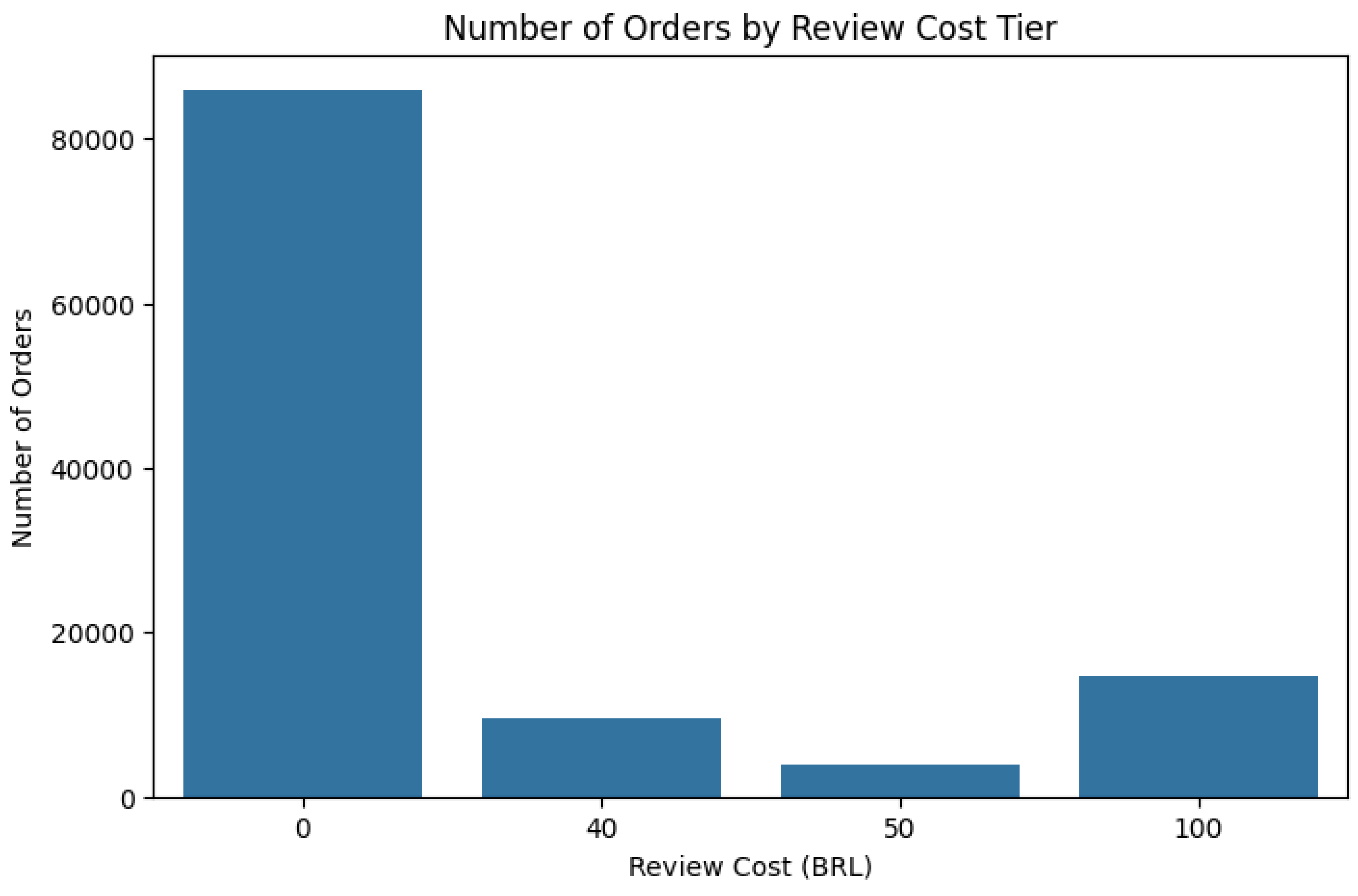
3.2.1 Review Scores Histogram

In [23]:
```
# It helps us see how many low scores are actually driving the costs
plt.figure(figsize=(8,5))
sns.histplot(data['review_score'], bins=range(1, 7), discrete=True)
plt.title('Distribution of review Scores')
plt.xlabel('Review Score')
plt.xticks([1, 2, 3, 4, 5])
plt.ylabel('Number of Orders')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```
executed in 447ms, finished 10:56:29 2025-05-18



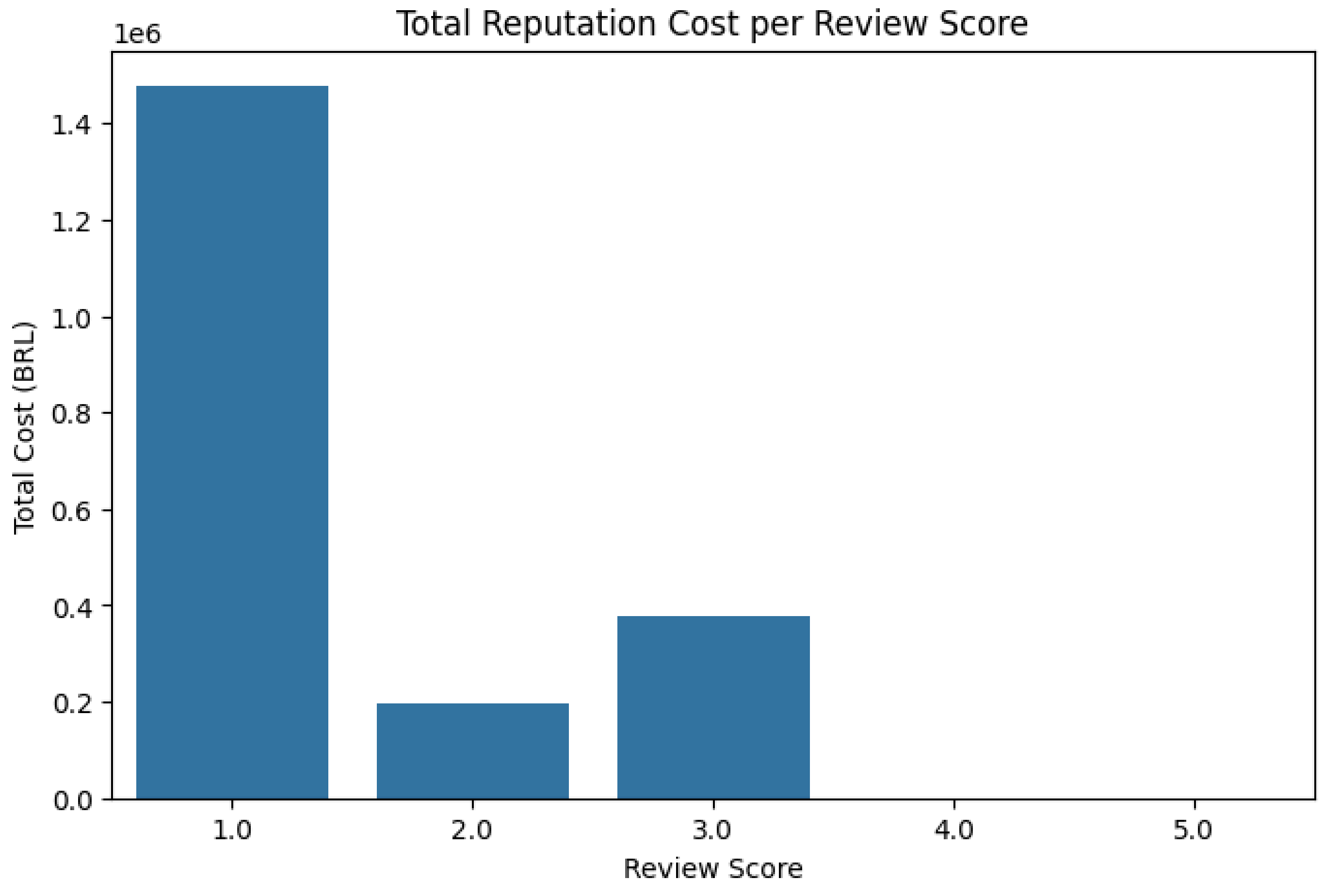
3.2.2 Review Score vs. Review Cost

In [24]:
```
plt.figure(figsize=(8,5))
sns.barplot(
    x=data['review_cost'],
    y=data['review_cost'],
    estimator=lambda x: len(x),
    errorbar=None
)
plt.title('Number of Orders by Review Cost Tier')
plt.xlabel('Review Cost (BRL)')
plt.ylabel('Number of Orders')
plt.show()
```
executed in 376ms, finished 10:56:29 2025-05-18



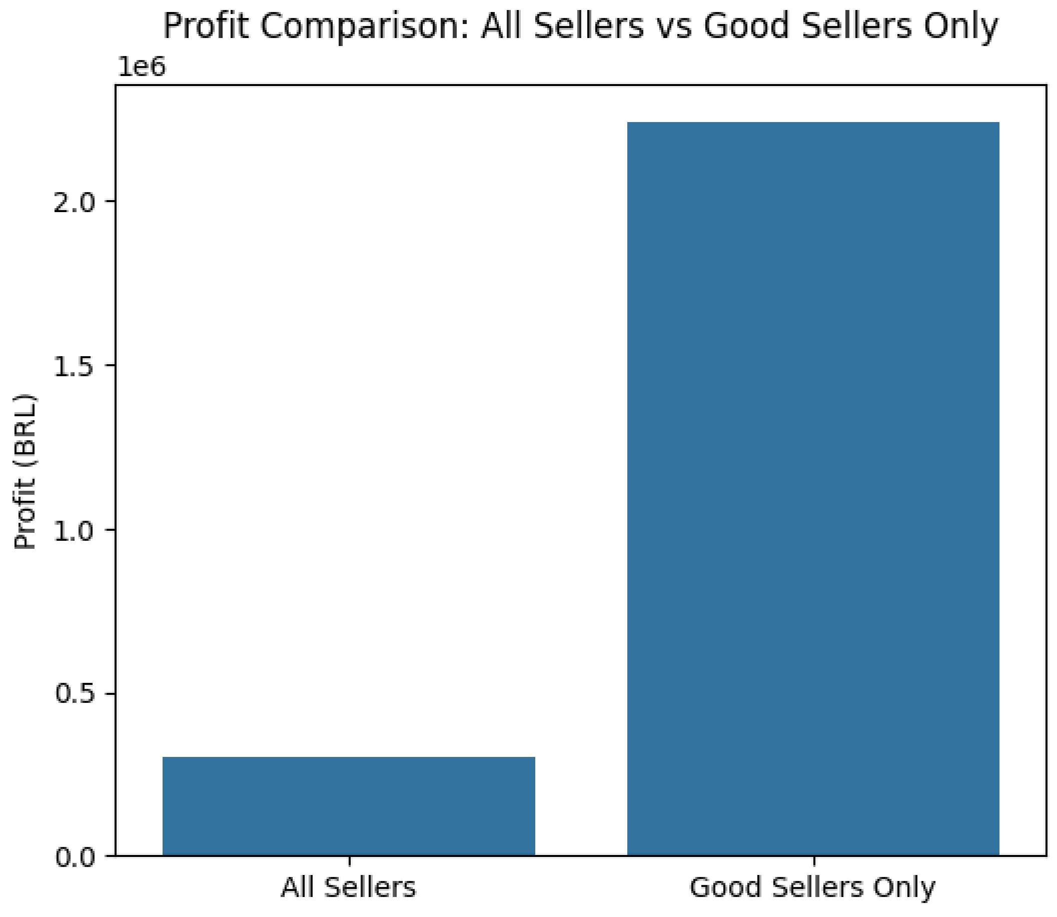
3.2.3 Cumulative Cost by Score

In [25]:
```
cost_by_score = data.groupby('review_score')['review_cost'].sum().reset_index()

plt.figure(figsize=(8,5))
sns.barplot(x='review_score', y='review_cost', data=cost_by_score)
plt.title('Total Reputation Cost per Review Score')
plt.xlabel('Review Score')
plt.ylabel('Total Cost (BRL)')
plt.show()
```
executed in 416ms, finished 10:56:30 2025-05-18



3.2.4 Profit Comparison: All vs. Good Sellers

In [26]:
```
profits = {
    'All Sellers': 302022.93,
    'Good Sellers Only': 2243617.21
}

plt.figure(figsize=(6,5))
sns.barplot(x=list(profits.keys()), y=list(profits.values()))
plt.title('Profit Comparison: All Sellers vs Good Sellers Only')
plt.ylabel('Profit (BRL)')
plt.show()
```
executed in 282ms, finished 10:56:30 2025-05-18



3.2.5 Reputation Cost as % of Revenue

In [27]:
```
revenue_total = total_revenue
rep_cost = reputation_cost

percent_cost = rep_cost / revenue_total * 100
print(f"Reputation Cost as % of Revenue: {percent_cost:.2f}%")
```
executed in 6ms, finished 10:56:30 2025-05-18

```
Reputation Cost as % of Revenue: 71.79%
```

3.3 Conclusion

Improve Customer Satisfaction:
Focus on strategies to improve customer satisfaction to increase the number of high review scores (4 and 5) and reduce the number of low review scores (1 and 2).
Address Negative Reviews:
Implement measures to address the issues leading to negative reviews promptly. This could involve better customer service, quality control, or post-purchase support.
Monitor Reputation Costs:
Regularly monitor the reputation costs and their impact on overall revenue. This will help in identifying trends and taking proactive measures.
Acting on IT-Cost:
It is not efficient as the saving would not be substantial.

3.3.1 Remarks

We are in right to question the estimation of the Cost of Reputation provided by the company considering the difference of profit with or without it. In the same way their ETA (estimated time arrival) seemed to be in much earlier that the actual time of arrival.

4 Sales Prediction

We are going to make a prediction for the future sales of the company 14 days ahead.

4.1 Understanding the Goal

We want to **predict how much money (revenue) Olist will make in the future**, based on patterns from past sales.

4.2 Daily Revenue: What Happened Before?

We start by adding up all the daily sales (we call it daily_revenue).

📈 We plot this to see trends – Are we selling more over time? Are there ups and downs?

In [28]:
```
# What have been the volume of sales throughout time with the data we have.
monthly_sales = data.set_index('order_purchase_timestamp') \
                    .resample('ME')['price'].sum()
```
executed in 265ms, finished 10:56:30 2025-05-18

In [29]:
```
print(monthly_sales)
```
executed in 10ms, finished 10:56:30 2025-05-18

```
order_purchase_timestamp
2016-09-30        267.36
2016-10-31      49634.35
2016-11-30          0.00
2016-12-31         10.90
2017-01-31     121087.90
2017-02-28     248563.02
2017-03-31     376010.70
2017-04-30     360738.17
2017-05-31     509639.63
2017-06-30     436550.89
2017-07-31     501299.70
2017-08-31     578588.99
2017-09-30     626752.17
2017-10-31     667869.15
2017-11-30    1017758.83
2017-12-31     746717.15
2018-01-31     955658.74
2018-02-28     853591.21
2018-03-31     986867.05
2018-04-30     998893.07
2018-05-31     997066.66
2018-06-30     865956.24
2018-07-31     897496.14
2018-08-31     854760.45
2018-09-30        145.00
2018-10-31          0.00
Freq: ME, Name: price, dtype: float64
```

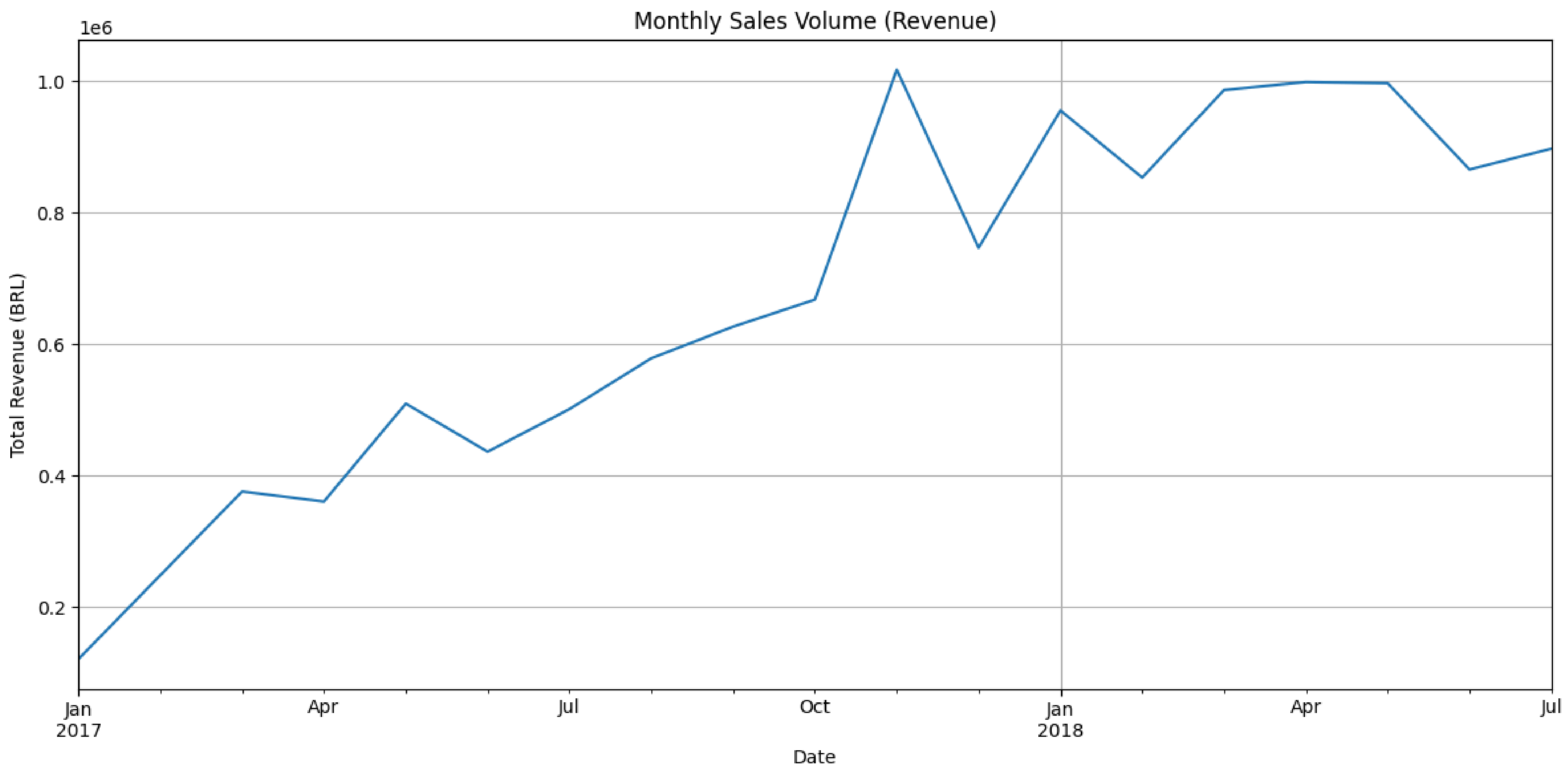
2016 Data: The sales figures for September and October 2016 are significantly lower compared to the rest of the dataset. November and December 2016 have almost negligible sales, which might indicate incomplete data or a period when sales were not fully recorded.

2018 Data: Similarly, the sales figures for September and October 2018 drop drastically compared to the preceding months. This could also indicate incomplete data or a significant change in business operations.

Since there is no way to justify these anomaly (business operation or specific event), we'll treat these data as abnormal (probably no record of the actual sales) and remove them so they do not influence our work.

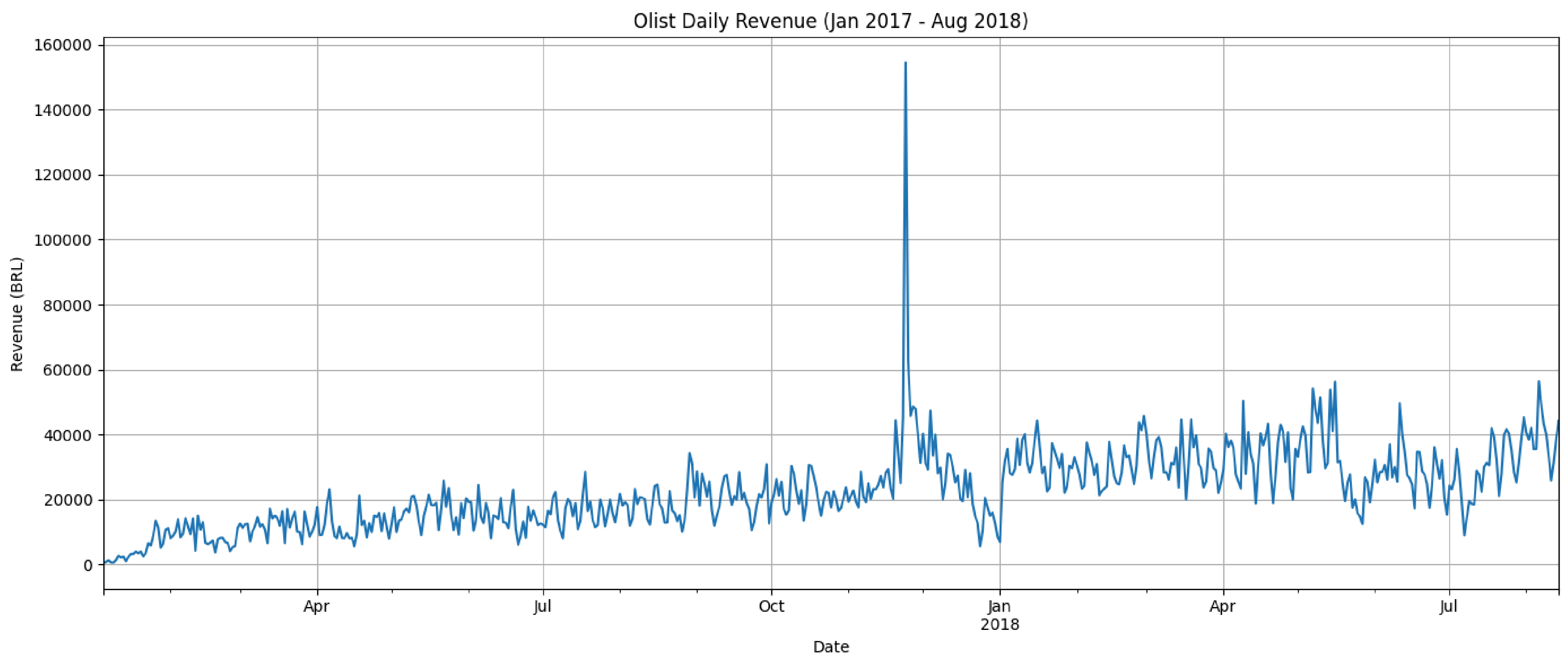
In [30]:
```
monthly_sales.index = pd.to_datetime(monthly_sales.index)
start_date = '2017-01-01'
end_date = '2018-08-15'
monthly_sales = monthly_sales.loc[start_date:end_date]
```
executed in 13ms, finished 10:56:30 2025-05-18

In [31]:
```
monthly_sales.plot(figsize=(12, 6))
plt.title("Monthly Sales Volume (Revenue)")
plt.xlabel("Date")
plt.ylabel("Total Revenue (BRL)")
plt.grid(True)
plt.tight_layout()
plt.show()
```
executed in 287ms, finished 10:56:30 2025-05-18



In [32]:
```python
# Ensure the 'order_purchase_timestamp' column is in datetime format
data['order_purchase_timestamp'] = pd.to_datetime(data['order_purchase_timestamp'])

# Filter the data
data_s = data[(data['order_purchase_timestamp'] >= '2017-01-01') &
              (data['order_purchase_timestamp'] < '2018-08-15')]

# Resample and sum the price on a daily basis
daily_revenue = data_s.set_index('order_purchase_timestamp').resample('D')['price'].sum()

# Plot the daily revenue
plt.figure(figsize=(14, 6))
daily_revenue.plot()
plt.title("Olist Daily Revenue (Jan 2017 - Aug 2018)")
plt.xlabel("Date")
plt.ylabel("Revenue (BRL)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

executed in 590ms, finished 10:56:31 2025-05-18



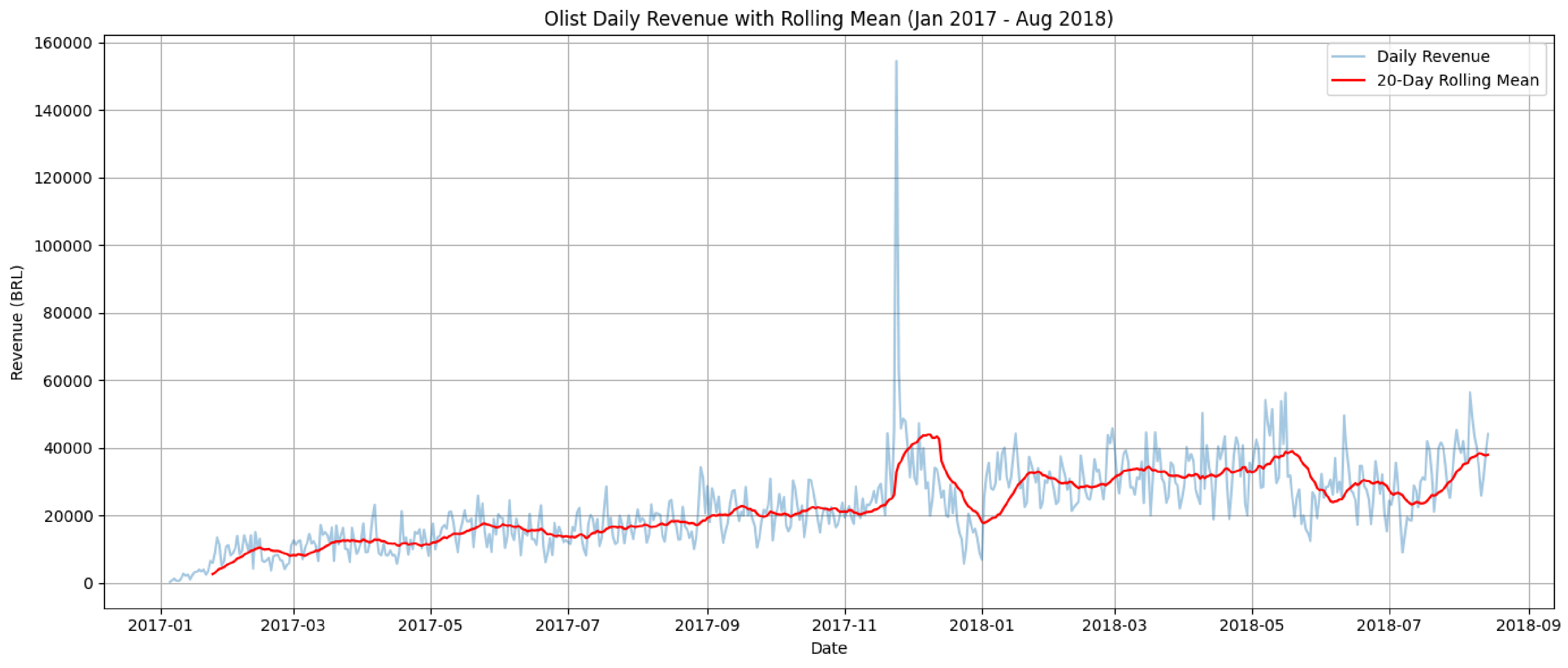
Right away, what catches the most attention in 'Olist Daily Revenue' is the great spike in revenue which actually occurred on the Black Friday of 2017, on November 24th. We can also notice that, in general, 2018 revenues are higher than in 2017.

4.3 Smoothing the Revenue

Sales can be noisy – for example, a big promotion might spike revenue for one day. To better see the general direction, we use a rolling average (like looking at the average of 20 days at a time) to smooth out the noise.

This helps us focus on long-term behavior instead of short-term surprises.

In [33]:
```
window_size = 20

# Compute rolling mean to smooth the curves to reduce the impact of short-terms fluctuation
smooth_daily_revenue = np.convolve(daily_revenue, np.ones(window_size)/window_size, mode='valid')

smoothed_index = daily_revenue.index[window_size - 1:]

plt.figure(figsize=(14, 6))

# Original time series
plt.plot(daily_revenue.index, daily_revenue, label='Daily Revenue', alpha=0.4)

# Smoothed line
plt.plot(smoothed_index, smooth_daily_revenue, label=f'{window_size}-Day Rolling Mean', color='red')

plt.title("Olist Daily Revenue with Rolling Mean (Jan 2017 - Aug 2018)")
plt.xlabel("Date")
plt.ylabel("Revenue (BRL)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```
executed in 370ms, finished 10:56:31 2025-05-18



4.4 Using ARIMA to forecast the 2 next weeks of sales.

Train vs Test Data (What We Know vs What We Predict)
We split the data into:

Training data: what we already know (Jan 2017 to July 2018)

Testing data: the future we're trying to predict (first two weeks of August 2018)

In [34]:
```
# Sort the index just to be safe
daily_revenue = daily_revenue.sort_index()

# Split the data
df_train = daily_revenue['2017-01-01':'2018-07-31']
df_test  = daily_revenue['2018-08-01':'2018-08-14']
```
executed in 13ms, finished 10:56:31 2025-05-18

In [35]:
```
# Describe data
df_train.describe().T
```
executed in 19ms, finished 10:56:31 2025-05-18

Out[35]:
```
count       573.000000
mean      22246.257260
std       12264.036592
min         396.900000
25%       13588.540000
50%       20561.440000
75%       29262.920000
max      154461.880000
Name: price, dtype: float64
```

4.5 Seasonal Decomposition: Finding Hidden Patterns

We use a tool that breaks our revenue into 3 parts:

- **Trend** – Are sales growing or shrinking?
- **Seasonality** – Do sales follow a repeating pattern? (like weekends or monthly cycles)

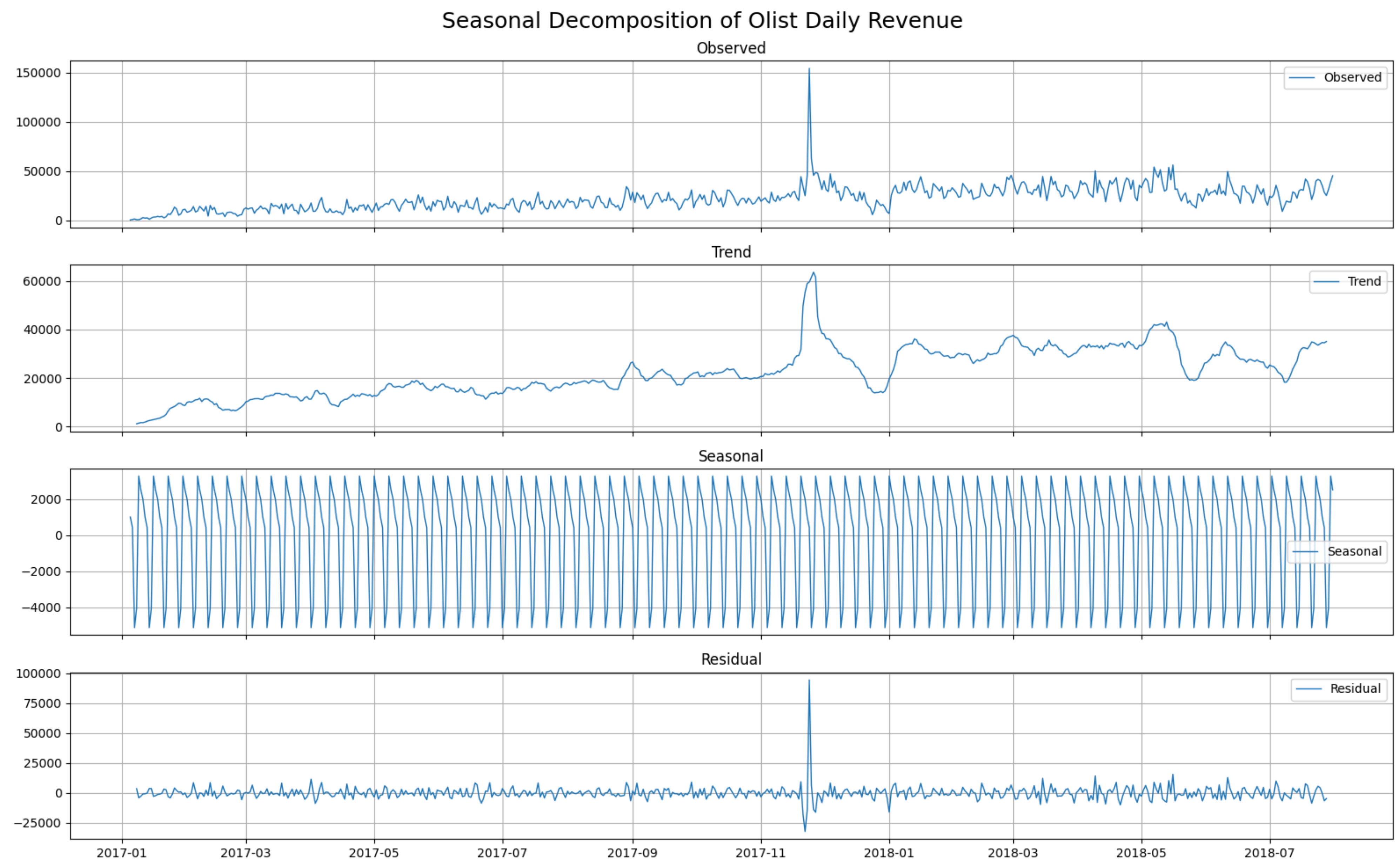
- **Residuals** – Hidden patterns we can't explain

This gives us a deeper look into what's really driving sales over time.

The three main components of a time series: trend, seasonality, and residuals are given by seasonal_decompose.

In [36]:
```python
decomposition = seasonal_decompose(df_train, model='additive')

fig, axes = plt.subplots(4, 1, figsize=(16, 10), sharex=True)

# Observed
axes[0].plot(decomposition.observed, label='Observed', linewidth=1)
axes[0].set_title('Observed')

# Trend
axes[1].plot(decomposition.trend, label='Trend', linewidth=1)
axes[1].set_title('Trend')

# Seasonal
axes[2].plot(decomposition.seasonal, label='Seasonal', linewidth=1)
axes[2].set_title('Seasonal')

# Residual
axes[3].plot(decomposition.resid, label='Residual', linewidth=1)
axes[3].set_title('Residual')

# General plot formatting
for ax in axes:
    ax.legend()
    ax.grid(True)

plt.suptitle("Seasonal Decomposition of Olist Daily Revenue", fontsize=18)
plt.tight_layout()
plt.show()
```

executed in 1.33s, finished 10:56:33 2025-05-18



- **The trend** looks like a rolling mean of about 10 days, it goes up which shows an increase in sales over time.
- **The seasonal component** shows a weekly pattern.
- **The residual** stays close to 0.

4.6 Forecasting with ARIMA

We use a forecasting model called ARIMA that:

- Learns from the trend,
- Adjusts for repeating patterns,
- And handles randomness.

We test different versions of ARIMA to find the best one that fits our past data. Once we have it, we use it to predict what revenue will look like in August.

4.6.1 Stationarity

We check for **stationarity** to make sure our forecast model doesn't get confused by drifting or unstable data. If the data is not stationary, we "clean it up" (usually by differencing) so it's ready to predict the future.

There are two main tests we can run:

ADF Test: If the result is below 0.05, it means the data is stationary.

KPSS Test: If the result is above 0.05, it means the data is stationary.

In real life, sales might slowly grow, fluctuate with the seasons, or have random spikes — so the data often **isn't stationary**.

To fix this, we sometimes **transform the data** (for example, by taking the difference between each day and the day before). This helps us get a more stable series that's easier to model and forecast accurately.

In [37]:
```
# Augmented Dickey-Fuller test on the original data
result = adfuller(df_train)
print(f'ADF Statistic is {result[0]:.2f} and the p-value is {result[1]:.3f}')

# KPSS test on original data
result = kpss(df_train)
print(f'KPSS Statistic is {result[0]:.2f} and the p-value is {result[1]:.3f}')
```
executed in 147ms, finished 10:56:33 2025-05-18

```
ADF Statistic is -2.78 and the p-value is 0.062
KPSS Statistic is 2.78 and the p-value is 0.010

/home/romaric/.pyenv/versions/3.10.6/envs/lewagon/lib/python3.10/site-packages/statsmodels/tsa/stattools.py:2018: InterpolationWa
rning: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is smaller than the p-value returned.

  warnings.warn(
```

We ran two tests to check if our time series data is stationary:

- **ADF Test**: p-value = 0.079 → We fail to reject the null hypothesis, so the series is likely *not* stationary.
- **KPSS Test**: p-value = 0.01 → We reject the null hypothesis of stationarity.

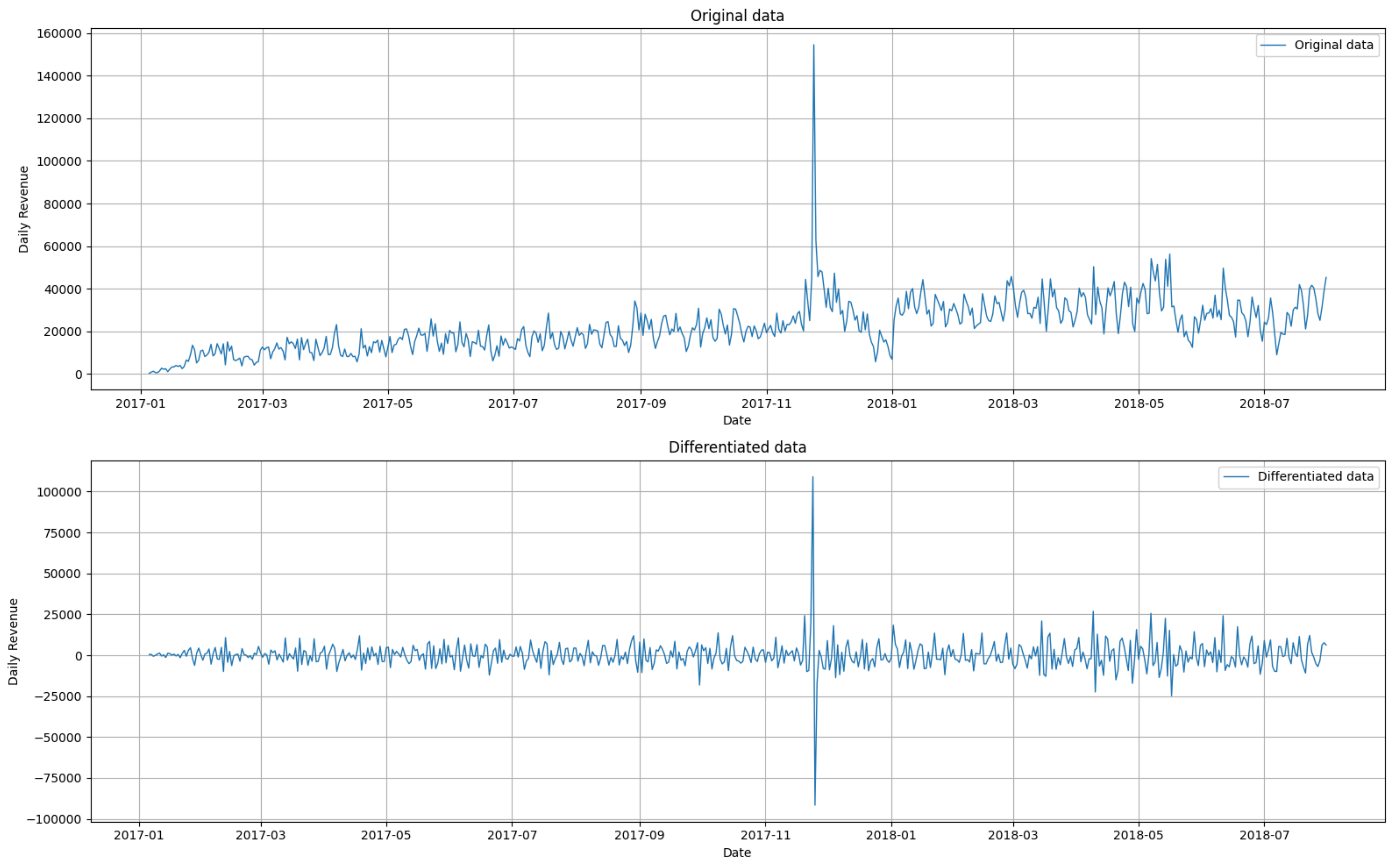
✅ Both tests suggest that our time series is **not stationary**, which means we need to **difference** it before forecasting (d=1).

In [38]:
```
# ADF test on the differenciated data
result = adfuller(df_train.diff().dropna())

print(f'ADF Statistic is {result[0]:.2f} and the p-value is {result[1]:.3f}')

# KPSS test on differenciated data
result = kpss(df_train.diff().dropna())

print(f'KPSS Statistic is {result[0]:.2f} and the p-value is {result[1]:.3f}')
```
executed in 99ms, finished 10:56:33 2025-05-18

```
ADF Statistic is -7.95 and the p-value is 0.000
KPSS Statistic is 0.03 and the p-value is 0.100

/home/romaric/.pyenv/versions/3.10.6/envs/lewagon/lib/python3.10/site-packages/statsmodels/tsa/stattools.py:2022: InterpolationWa
rning: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is greater than the p-value returned.

  warnings.warn(
```

With these results we can confirm that the series **needs to be differentiated one time (d=1) to be stationary** and to be used for forecasting.
See the graphes below

In [39]:
```python
fig, axes = plt.subplots(2, 1, figsize=(16, 10))

# Original data
axes[0].plot(df_train, label='Original data', linewidth=1)
axes[0].set_title('Original data')

# Differentiated data
axes[1].plot(df_train.diff(), label='Differentiated data', linewidth=1)
axes[1].set_title('Differentiated data')

# General plot formatting
for ax in axes:
    ax.set_ylabel('Daily Revenue')
    ax.set_xlabel('Date')
    ax.legend()
    ax.grid(True)

plt.tight_layout()
plt.show()
```
executed in 848ms, finished 10:56:34 2025-05-18

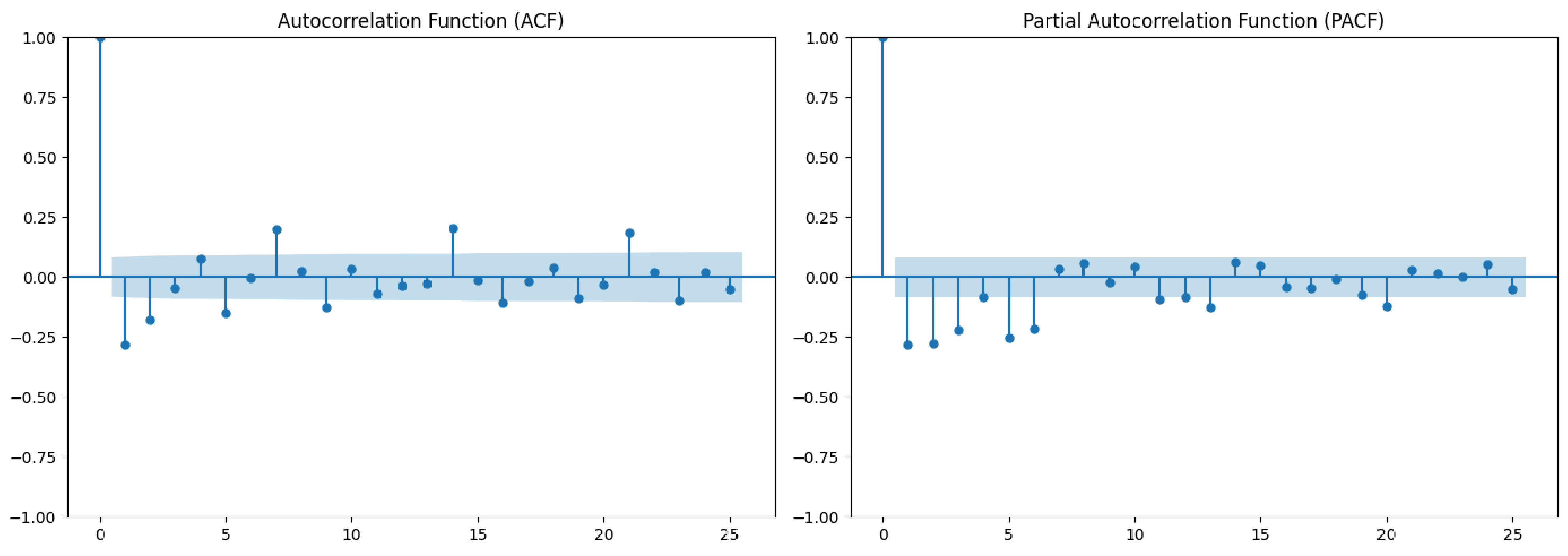


4.6.2 Pre-analysis of ACF and PACF

We run a pre-analysis of Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) on the differentiated series to get the best training indicators for our model.

In [40]:
```python
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))
# ACF
plot_acf(df_train.diff().dropna(), lags=25, alpha=0.05, ax=ax1)
ax1.set_title('Autocorrelation Function (ACF)')

# PACF
plot_pacf(df_train.diff().dropna(), lags=25, alpha=0.05, ax=ax2)
ax2.set_title('Partial Autocorrelation Function (PACF)')

plt.tight_layout()
plt.show()
```
executed in 489ms, finished 10:56:34 2025-05-18

```
/home/romaric/.pyenv/versions/3.10.6/envs/lewagon/lib/python3.10/site-packages/statsmodels/graphics/tsaplots.py:348: FutureWarnin
g: The default method 'yw' can produce PACF values outside of the [-1,1] interval. After 0.13, the default will change tounadjust
ed Yule-Walker ('ywm'). You can use this method now by setting method='ywm'.
  warnings.warn(
```



Looking at the ACF plot, we notice that the spikes at 7, 14, and 21 days stand out. Since our data is tracked daily, this suggests a weekly pattern — meaning what happens this week is often similar to what happened last week.

The PACF plot shows a kind of wavy, repeating pattern (like a sine wave), which often happens when the data has a strong regular cycle — again, hinting at that weekly trend.

This tells us that we're likely working with a time series where past values have an influence over time, especially in cycles of 7 days. Based on this, **it makes sense to explore an ARIMA model**, probably something like an *ARIMA(0, d, q)*.

4.6.3 Auto Arima

It is useful to find the optimal ARIMA model, we'll use it to find the values of p and q.

In [41]:
```python
fit_arima = auto_arima(
    df_train,                       # dataset
    max_p=3,                        # limiting the number of autoregressive terms so the model don't overfit
    max_q=3,                        # limiting the number of moving average terms so the model don't overfit
    m=7,                            # number of periods in the seasonal cycle
    seasonal=True,                  # try for seasonal data
    seasonal_test='ocsb',           # use the OCSB test for seasonality
    d=1,                            # number of differencing equals to 1 because of ADF and KPSS results
    trace=False,                    # to not print the progress of the model
    information_criterion='bic',    # choose the best model based on Bayesian information criterion
    stepwise=False                  # testing all possible combinations
)
```
executed in 1m 45.0s, finished 10:58:19 2025-05-18

In [42]:
```
fit_arima.summary()
```
executed in 35ms, finished 10:58:19 2025-05-18

Out[42]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	573
Model:	SARIMAX(3, 1, 0)x(1, 0, [1], 7)	**Log Likelihood**	-5911.870
Date:	Sun, 18 May 2025	**AIC**	11837.740
Time:	10:58:19	**BIC**	11868.184
Sample:	01-05-2017	**HQIC**	11849.616
	- 07-31-2018		
Covariance Type:	opg		

	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
intercept	1.3675	23.681	0.058	0.954	-45.047	47.782
ar.L1	-0.4911	0.018	-26.594	0.000	-0.527	-0.455
ar.L2	-0.3636	0.024	-15.265	0.000	-0.410	-0.317
ar.L3	-0.2269	0.019	-11.726	0.000	-0.265	-0.189
ar.S.L7	0.9907	0.017	59.333	0.000	0.958	1.023
ma.S.L7	-0.9317	0.040	-23.090	0.000	-1.011	-0.853
sigma2	6.485e+07	0.000	1.96e+11	0.000	6.49e+07	6.49e+07

Ljung-Box (L1) (Q):	0.08	**Jarque-Bera (JB):**	252198.80
Prob(Q):	0.78	**Prob(JB):**	0.00
Heteroskedasticity (H):	2.62	**Skew:**	6.47
Prob(H) (two-sided):	0.00	**Kurtosis:**	105.05

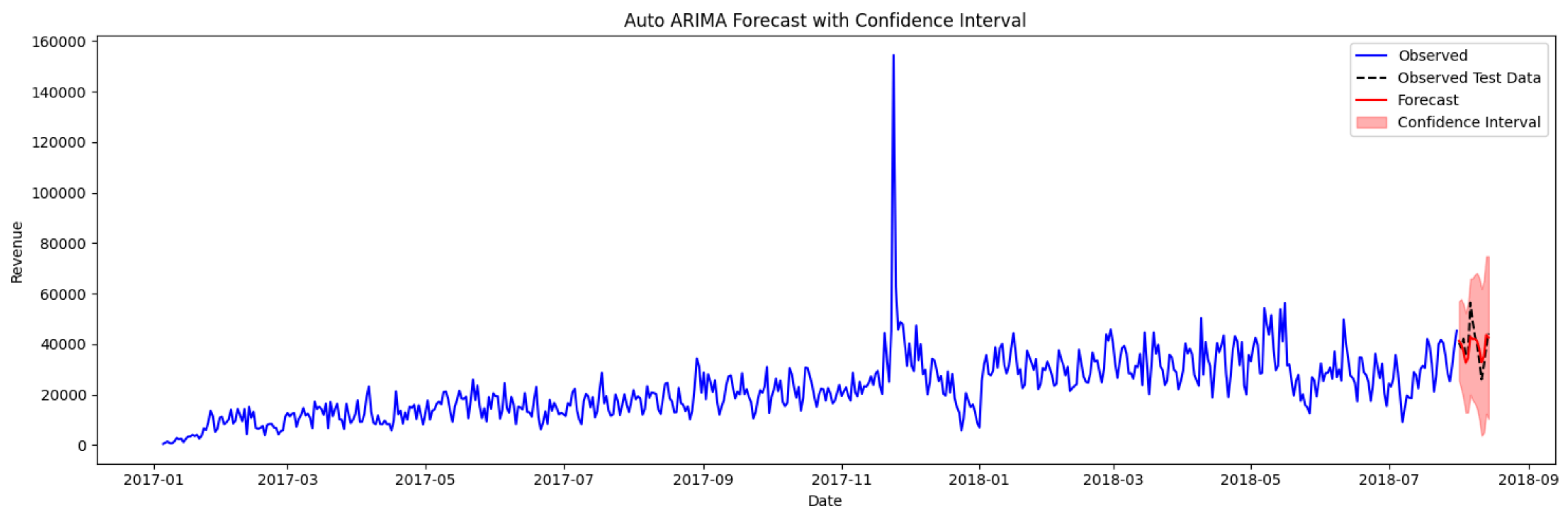
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 4.57e+25. Standard errors may be unstable.

4.6.4 FORECAST

We will use this fitted model to forecast the next 14 days of revenue and confidence intervals limit.

In [43]:
```
forecast, conf_int = fit_arima.predict(n_periods=14, return_conf_int=True)
```
executed in 19ms, finished 10:58:19 2025-05-18

In [44]:
```
forecast_index = pd.date_range(start='2018-08-01', periods=14)

# Convert forecast to series
forecast_series = pd.Series(forecast, index=forecast_index)
lower_series = pd.Series(conf_int[:, 0], index=forecast_index)
upper_series = pd.Series(conf_int[:, 1], index=forecast_index)

# Plot observed, test, forecast
plt.figure(figsize=(15, 5))

plt.plot(df_train.index, df_train, label='Observed', color='blue')  # training data
plt.plot(df_test.index, df_test, label='Observed Test Data', color='black', linestyle='--')  # test data
plt.plot(forecast_series, label='Forecast', color='red')  # forecast
plt.fill_between(forecast_index, lower_series, upper_series, color='red', alpha=0.3, label='Confidence Interval')

plt.title('Auto ARIMA Forecast with Confidence Interval')
plt.xlabel('Date')
plt.ylabel('Revenue')
plt.legend()
plt.tight_layout()
plt.show()
```
executed in 420ms, finished 10:58:20 2025-05-18



4.7 Comparing Prediction to Reality

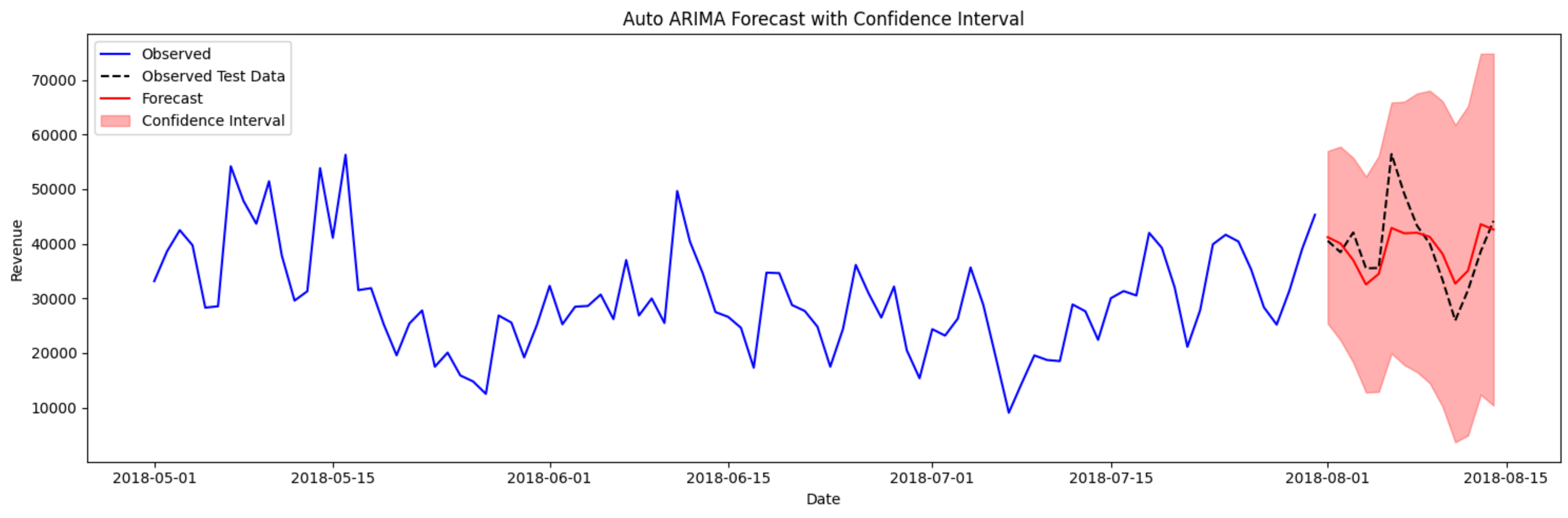
After forecasting, we compare our predicted values with the real sales data from August to see:

- Did we guess well?
- Where were we off?

📉 This helps us improve future predictions.

Let's zoom onto our forecast!

In [45]:
```
zoom_df = daily_revenue['2018-05-01':'2018-07-31']

plt.figure(figsize=(15, 5))

plt.plot(zoom_df.index, zoom_df, label='Observed', color='blue')  # training data
plt.plot(df_test.index, df_test, label='Observed Test Data', color='black', linestyle='--')  # test data
plt.plot(forecast_series, label='Forecast', color='red')  # forecast
plt.fill_between(forecast_index, lower_series, upper_series, color='red', alpha=0.3, label='Confidence Interval')

plt.title('Auto ARIMA Forecast with Confidence Interval')
plt.xlabel('Date')
plt.ylabel('Revenue')
plt.legend()
plt.tight_layout()
plt.show()
```
executed in 317ms, finished 10:58:20 2025-05-18



We need to **Evaluate** these results.

The best indicators here are:

- **MAE** (Mean Absolute Error), the average absolute difference between the predicted values and the actual values
- **RMSE** (Root Mean Squared Error), the average magnitude of the errors, it gives more weight to larger errors.
- **MAPE** (Mean Absolute Percentage Error), it is the percentage of difference between the predicted and the actual values.

In [46]:
```
# Transforming the dataframes to numpy array
y_true = np.array(df_test)
y_pred = np.array(forecast_series)

# Creating metrics
mae  = mean_absolute_error(y_true, y_pred)
rmse = np.sqrt(mean_squared_error(y_true, y_pred))
mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100

print(f'MAE: {mae:.2f}\nRMSE: {rmse:.2f}\nMAPE: {mape:.2f}')
```
executed in 12ms, finished 10:58:20 2025-05-18

```
MAE: 4020.93
RMSE: 5245.65
MAPE: 10.14
```

What does it mean? The model's predictions are off by approximately BRL 4,020.93 on average, approximately 10.14%.

In [49]:
```
real_incomes= y_true.sum()
forecast_incomes = y_pred.sum()
print(f'The ARIMA model predicted incomes of BRL {forecast_incomes:,.2f} when it was in reality BRL {real_incomes:,.2f}')
```
executed in 17ms, finished 11:16:01 2025-05-18

```
The ARIMA model predicted incomes of BRL 545,626.01 when it was in reality BRL 554,882.52
```

In [54]:
```
ARIMA_pct = ((real_incomes-forecast_incomes)/real_incomes)*100
print(f'This is a relative error of {ARIMA_pct:.2f}% !')
```
executed in 8ms, finished 11:18:07 2025-05-18

```
This is a relative error of 1.67% !
```

4.8 Forecast with SARIMA

SARIMA removes the need to '*de-seasonalize*' our data.

However, it is common practice to use a log function on the data to remove the effect of the increasing variance of the data overtime, and then '*re-construct*' the model with an exponential function.

However, we've seen through the decomposition of our data that the variance is pretty stable over time and it does not justify using a log function, it could **even be harmful for our model by distorting our data unnecessarily**.

In [57]:
```
# Fitting the same orders from Auto ARIMA to compare the models
model   = SARIMAX(df_train, order=(0, 1, 1), seasonal_order=(1, 0, 1, 7))

sarima = model.fit()
```
executed in 1.64s, finished 11:24:54 2025-05-18

```
/home/romaric/.pyenv/versions/3.10.6/envs/lewagon/lib/python3.10/site-packages/statsmodels/base/model.py:604: ConvergenceWarning:
Maximum Likelihood optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

In [61]:
```
sarima.summary()
```
executed in 29ms, finished 11:28:15 2025-05-18

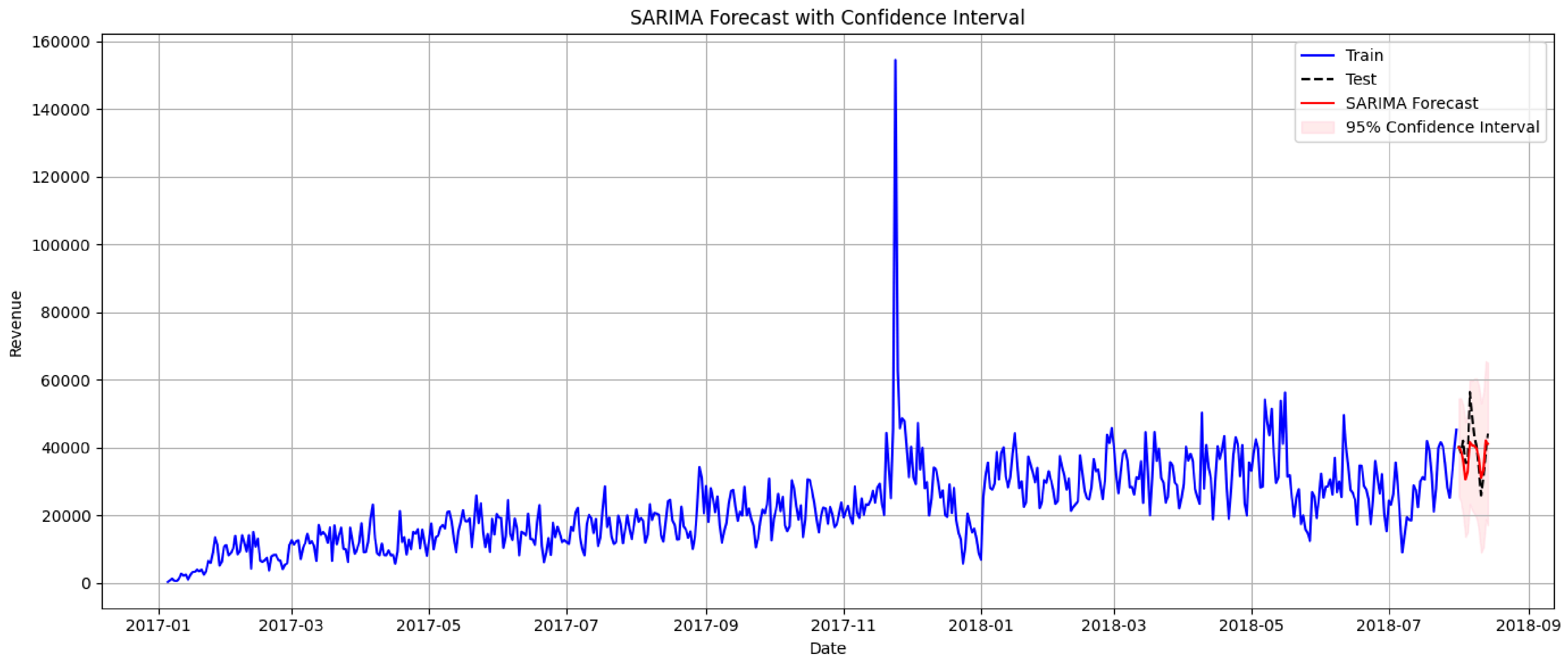
Out[61]:
```
SARIMAX Results

     Dep. Variable:                     price  No. Observations:         573
             Model:  SARIMAX(0, 1, 1)x(1, 0, 1, 7)  Log Likelihood   -5908.421
              Date:          Sun, 18 May 2025               AIC    11824.841
              Time:                  11:28:15               BIC    11842.238
            Sample:                01-05-2017              HQIC    11831.628
                                 - 07-31-2018
  Covariance Type:                        opg

              coef    std err         z   P>|z|    [0.025     0.975]
   ma.L1   -0.6450      0.017   -37.377   0.000    -0.679     -0.611
  ar.S.L7   0.9939      0.012    84.549   0.000     0.971      1.017
  ma.S.L7  -0.9414      0.030   -31.219   0.000    -1.000     -0.882
   sigma2  5.42e+07   4.04e-10  1.34e+17  0.000  5.42e+07   5.42e+07

     Ljung-Box (L1) (Q):  6.34  Jarque-Bera (JB):  356247.00
                Prob(Q):  0.01          Prob(JB):       0.00
Heteroskedasticity (H):  2.67              Skew:       7.52
  Prob(H) (two-sided):   0.00          Kurtosis:     124.33


Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 7.02e+30. Standard errors may be unstable.
```

In [60]:
```
# Forecast
sarima_results = sarima.get_forecast(len(df_test), alpha=0.05)

sarima_forecast = sarima_results.predicted_mean
sar_conf_int = sarima_results.conf_int()
```
executed in 24ms, finished 11:27:34 2025-05-18

In [66]:
```
# Reconstruct

sarima_forecast_recons = pd.Series(sarima_forecast, index = df_test.index)
sarima_low_recons = sar_conf_int['lower price'].values
sarima_up_recons = sar_conf_int['upper price'].values
```
executed in 16ms, finished 11:53:03 2025-05-18

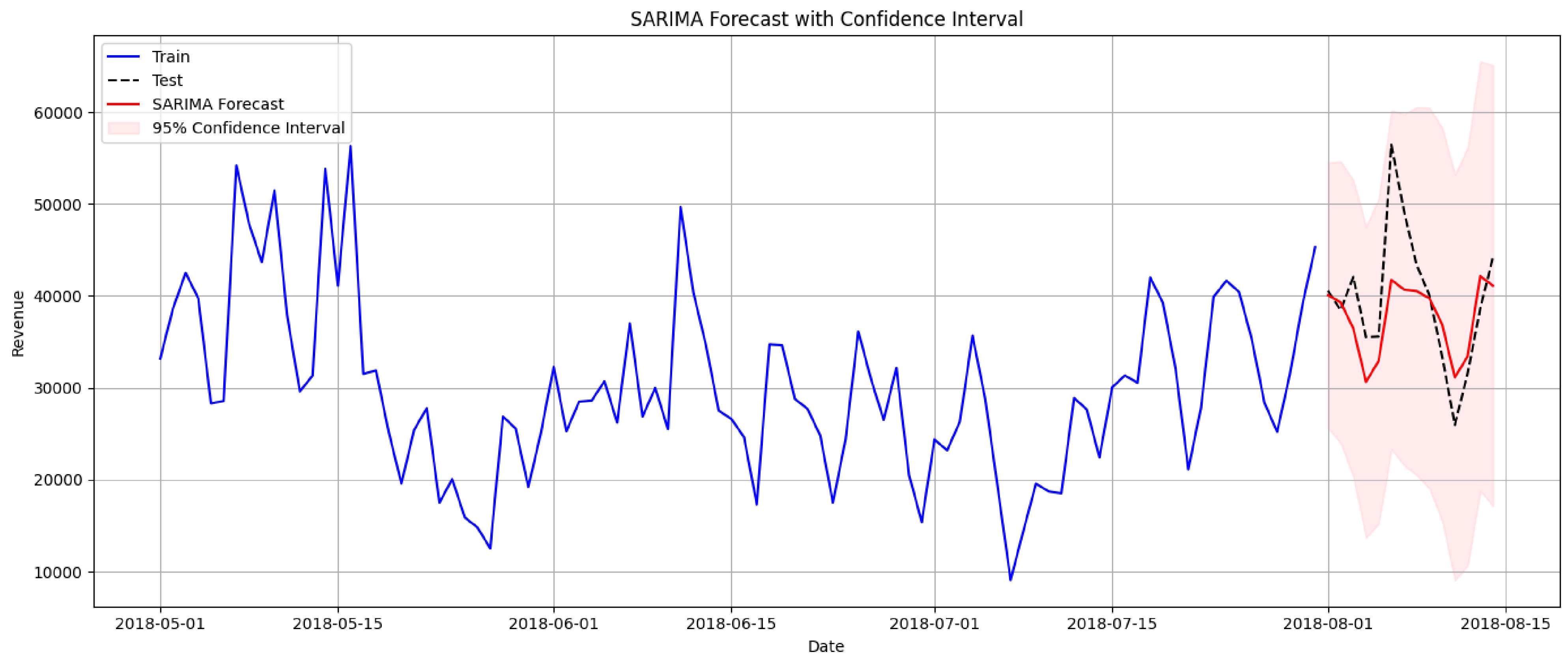
In [69]:
```python
plt.figure(figsize=(14, 6))

# Plot the training data
plt.plot(df_train, label='Train', color='blue')

# Plot the test data
plt.plot(df_test, label='Test', color='black', linestyle='--')

# Plot the forecasted values
plt.plot(sarima_forecast_recons, label='SARIMA Forecast', color='red')

# Plot confidence intervals
plt.fill_between(sarima_forecast_recons.index,
                 sarima_low_recons,
                 sarima_up_recons,
                 color='pink', alpha=0.3, label='95% Confidence Interval')

plt.title('SARIMA Forecast with Confidence Interval')
plt.xlabel('Date')
plt.ylabel('Revenue')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

executed in 545ms, finished 11:56:27 2025-05-18



Let's zoom in!

In [70]:
```python
plt.figure(figsize=(14, 6))

# Plot the training data
plt.plot(zoom_df, label='Train', color='blue')

# Plot the test data
plt.plot(df_test, label='Test', color='black', linestyle='--')

# Plot the forecasted values
plt.plot(sarima_forecast_recons, label='SARIMA Forecast', color='red')

# Plot confidence intervals
plt.fill_between(sarima_forecast_recons.index,
                 sarima_low_recons,
                 sarima_up_recons,
                 color='pink', alpha=0.3, label='95% Confidence Interval')

plt.title('SARIMA Forecast with Confidence Interval')
plt.xlabel('Date')
plt.ylabel('Revenue')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```
executed in 361ms, finished 11:57:39 2025-05-18



In [74]:
```python
# Transforming the dataframes to numpy array
y_true = np.array(df_test)
y_pred = np.array(sarima_forecast_recons)

# Creating metrics
mae  = mean_absolute_error(y_true, y_pred)
rmse = np.sqrt(mean_squared_error(y_true, y_pred))
mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100

print(f'MAE: {mae:.2f}\nRMSE: {rmse:.2f}\nMAPE: {mape:.2f}')

real_incomes= y_true.sum()
forecast_incomes = y_pred.sum()
print(f'The SARIMA model predicted incomes of BRL {forecast_incomes:,.2f} when it was in reality BRL {real_incomes:,.2f}')

SARIMA_pct = ((real_incomes-forecast_incomes)/real_incomes)*100
print(f'This is a relative error of {SARIMA_pct:.2f}%.')
```
executed in 21ms, finished 12:01:25 2025-05-18

```
MAE: 4137.56
RMSE: 5500.93
MAPE: 10.06
The SARIMA model predicted incomes of BRL 526,778.08 when it was in reality BRL 554,882.52
This is a relative error of 5.06%.
```

5 Conclusion: Why This Matters

In general, both models performed well and produced trusted forecast results with very small relative errors for the next fourteen days of revenue.

This whole process helps a company like Olist:

- Plan for the future,
- Prepare stock and deliveries,
- Make smarter decisions.

📊 Forecasting turns historical data into real business value.