Forecasting Usage of New York City's 311 Service

New York City maintains an information and reporting service for city residents that permit them to source information from and report problems to City Government with greater ease. While the service is an important enhancement to resident life, it imposes an administrative burden on the City as it must receive the requests and reports and refer them to the correct City agency. A model that can accurately forecast usage of the service would allow the City to serve residents more efficiently.

This analysis will search for a model with stronger predictive power than a baseline model. Potential models will include:

- ARIMA of varying orders
- SARIMA (ARIMA with seasonality) of varying orders
- SARIMAX (SARIMA with exogenous variables) with various regressors
- Prophet, a newer and more flexible model with better handling of seasonality
- GARCH and GARCH-X, if heteroskedasticity is an issue
- Neural networks (LSTM)

In [87]:
```
import dill
# uncomment the relevant code to load the full session state

# # Save the entire session state
# filename = '311_session.pkl'
# with open(filename, 'wb') as file:
#     dill.dump_session(file)

# # Load the entire session state
# filename = 'session.pkl'
# with open(filename, 'rb') as file:
#     dill.load_session(file)
```

In [88]:
```
import pandas as pd
import numpy as np

import matplotlib.dates as mdates
import matplotlib.pyplot as plt
import matplotlib.font_manager as fm
from matplotlib.ticker import MultipleLocator, FuncFormatter
import seaborn as sns
from cycler import cycler

import gc
import itertools
from typing import Optional, Tuple, List, Union
import time

from pmdarima.arima import auto_arima
from scipy.stats import boxcox
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import mean_squared_error, root_mean_squared_error
from sklearn.model_selection import TimeSeriesSplit
import statsmodels.api as sm
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.stattools import adfuller
```

In [2]:
```
# Change formatting for images
%matplotlib inline

preferred_font_path = r'C:\Windows\Fonts\GIL_____.ttf'
fallback_fonts = ['Arial', 'DejaVu Sans']
custom_font = None

try:
    fm.fontManager.addfont(preferred_font_path)
    custom_font = fm.FontProperties(fname=preferred_font_path).get_name()
    plt.rcParams['font.family'] = [custom_font] + fallback_fonts
except Exception as e:
    plt.rcParams['font.family'] = fallback_fonts

plt.rcParams['axes.prop_cycle'] = cycler(color=['#6C5B7B'])
background_color = (240/255, 240/255, 240/255)
plt.rcParams['figure.facecolor'] = background_color
plt.rcParams['axes.facecolor'] = background_color
```

Data Preparation

The data sources are:

- NYC OpenData's 311 call database, which includes calls dating from January 1, 2010
- Daily weather for New York City, retrieved from OpenMeteo's API

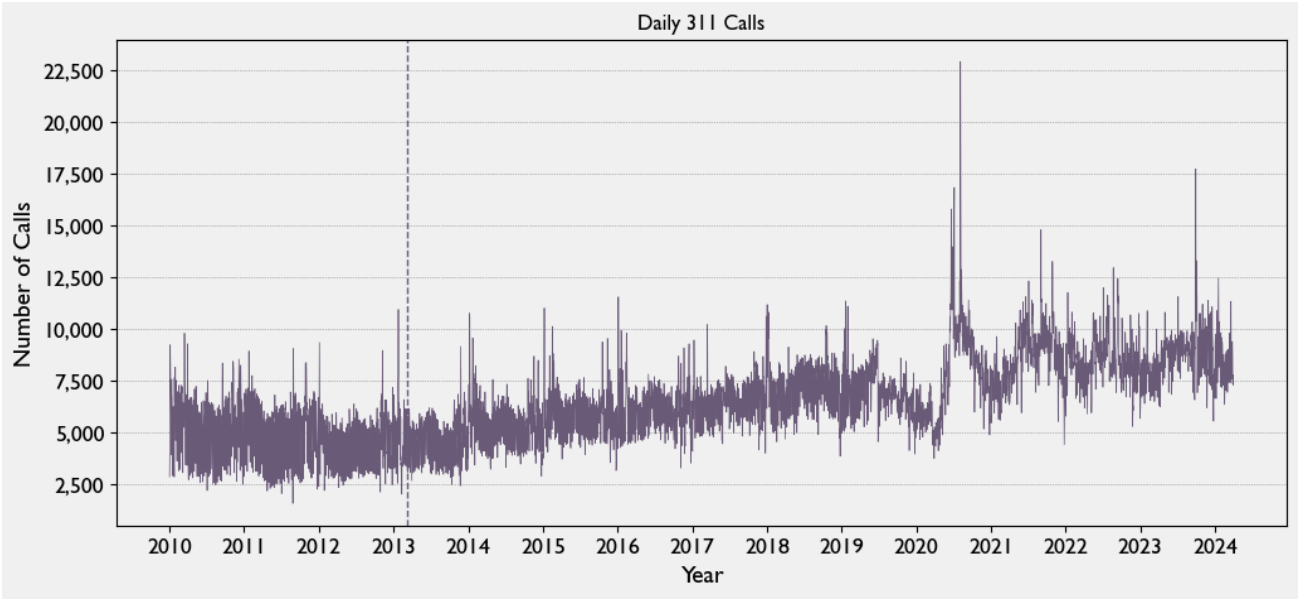
To simplify the functioning of this notebook, the data has been processed and stored in pickle files. To learn more about how, view the notebook titled 'NYC_311_Data_Prep.ipynb' in the Notebooks subfolder.

In [3]:
```
# Read in prepared data
df_311_calls = pd.read_pickle('Data/311_Calls.pickle')
df_zips = pd.read_pickle('Data/NYC_Zips.pickle')
df_weather = pd.read_pickle('Data/NYC_Weather.pickle')
```

Visualize Distributions

The charts in this section illustrate the data distributions and will be used in the presentation

In [4]:
```
daily_plot = df_311_calls.groupby('Date').size()
fig, ax = plt.subplots(figsize=(12,5))
ax.plot(daily_plot, linewidth=.5)
ax.set_xlabel('Year', fontsize=14)
plt.setp(ax.get_xticklabels(), fontsize=13)
ax.set_ylabel('Number of Calls', fontsize=14)
plt.setp(ax.get_yticklabels(), fontsize=13)

ax.xaxis.set_major_locator(mdates.YearLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))

ax.yaxis.set_major_locator(MultipleLocator(2500))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, pos: f'{int(x):,}'))

ax.grid(which='major', axis='y', linestyle=':', linewidth=0.5, color='gray')
ax.axvline(pd.to_datetime('2013-03-09'), linestyle='--', linewidth=1)
output_file = 'Charts/daily_call_volume.jpg'
plt.savefig(output_file, dpi=300, bbox_inches='tight')
ax.set_title('Daily 311 Calls')
plt.show()
```

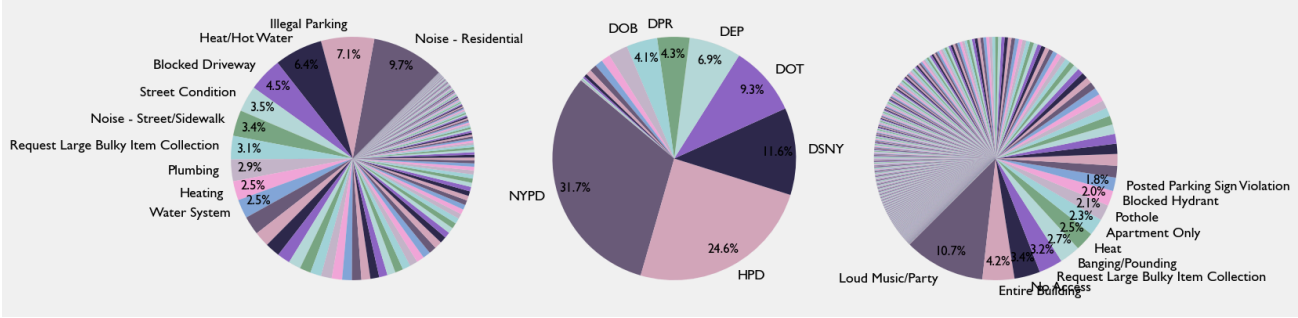


In [5]:
```
# Create sorted series by feature
types = df_311_calls.groupby('Type', observed=True).size().sort_values(ascending=False)
agencies = df_311_calls.groupby('Agency', observed=True).size().sort_values(ascending=False)
subtypes = df_311_calls.groupby('Descriptor', observed=True).size().sort_values(ascending=False)
```

In [6]:
```
# Function to keep labels only for the top specified wedges
def keep_top_labels(labels, sizes, limit):
    top_limit = sizes.argsort()[-limit:][::-1]
    new_labels = [labels[i].title() if i in top_limit else '' for i in range(len(labels))]
    new_autotexts = ['%1.1f%%' % (sizes[i] / sizes.sum() * 100) if i in top_limit else '' for i in range(len(labels))]
    return new_labels, new_autotexts
```

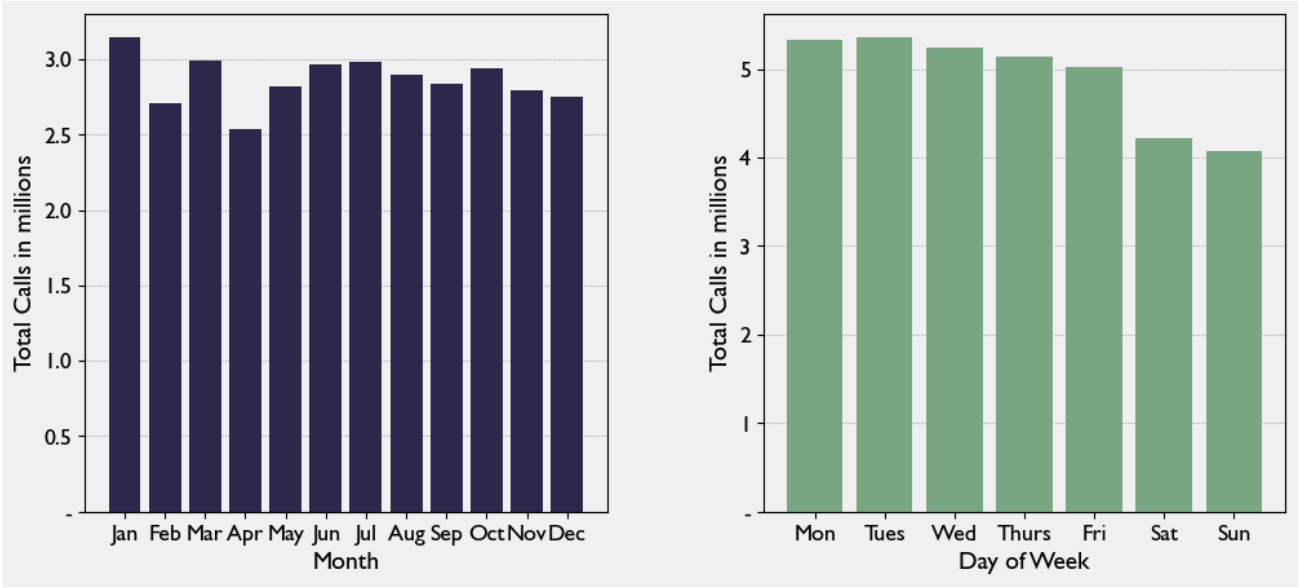
In [7]:
```
labels = types.index.to_list()
sizes = types.values
colors = ['#6C5B7B', '#D5A6BD', '#2E294E', '#9067C6', '#B8D8D8',
          '#7CA982', '#A0D2DB', '#C4B7CB', '#F4A6D7', '#85A6D7']
```

```
labels2 = agencies.index.to_list()
sizes2 = agencies.values
colors2 = ['#6C5B7B', '#D5A6BD', '#2E294E', '#9067C6', '#B8D8D8',
           '#7CA982', '#A0D2DB', '#C4B7CB', '#F4A6D7', '#85A6D7']

labels3 = subtypes.index.to_list()
sizes3 = subtypes.values
colors3 = ['#6C5B7B', '#D5A6BD', '#2E294E', '#9067C6', '#B8D8D8',
           '#7CA982', '#A0D2DB', '#C4B7CB', '#F4A6D7', '#85A6D7']

labels_top, autotexts_top = keep_top_labels(labels, sizes, 10)
labels2_top, autotexts2_top = keep_top_labels(labels2, sizes2, 7)
labels3_top, autotexts3_top = keep_top_labels(labels3, sizes3, 10)
labels2_top = [label.upper() for label in labels2_top]

fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(15, 5))
wedges1, texts1 = ax1.pie(sizes, labels=labels_top, colors=colors, startangle=45)
ax1.axis('equal')
wedges2, texts2 = ax2.pie(sizes2, labels=labels2_top, colors=colors2, startangle=140)
ax2.axis('equal')
wedges3, texts3 = ax3.pie(sizes3, labels=labels3_top, colors=colors3, startangle=225)
ax3.axis('equal')

for i, autotext in enumerate(autotexts_top):
    if autotext:
        angle = (wedges1[i].theta2 + wedges1[i].theta1) / 2
        x = wedges1[i].r * .85 * np.cos(np.radians(angle))
        y = wedges1[i].r * .85 * np.sin(np.radians(angle))
        ax1.text(x, y, autotext, ha='center', va='center', fontsize=12)
for i, autotext in enumerate(autotexts2_top):
    if autotext:
        angle = (wedges2[i].theta2 + wedges2[i].theta1) / 2
        x = wedges2[i].r * .85 * np.cos(np.radians(angle))
        y = wedges2[i].r * .85 * np.sin(np.radians(angle))
        ax2.text(x, y, autotext, ha='center', va='center', fontsize=12)
for i, autotext in enumerate(autotexts3_top):
    if autotext:
        angle = (wedges3[i].theta2 + wedges3[i].theta1) / 2
        x = wedges3[i].r * .85 * np.cos(np.radians(angle))
        y = wedges3[i].r * .85 * np.sin(np.radians(angle))
        ax3.text(x, y, autotext, ha='center', va='center', fontsize=12)
for text in texts1 + texts2 + texts3:
    text.set_fontsize(13)
output_file = 'Charts/jumbled_pies.jpg'
plt.savefig(output_file, dpi=300, bbox_inches='tight')
plt.show()
```



In [8]:
```
months = df_311_calls['Date'].dt.month.value_counts()
day_week = df_311_calls['Date'].dt.day_of_week.value_counts()

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(12,5))
ax1.bar(x=months.index, height=months, color='#2E294E', zorder=3)
ax2.bar(x=day_week.index, height=day_week, color='#7CA982', zorder=3)
plt.subplots_adjust(wspace=0.3)

x1_labels = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
y1_labels = ['-', '0.5', '1.0', '1.5', '2.0', '2.5', '3.0']
x2_labels = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun']
y2_labels = ['-', '1', '2', '3', '4', '5']

ax1.set_xlabel('Month', fontsize=14)
ax1.set_xticks(range(1, 13))
ax1.set_xticklabels(x1_labels, fontsize=13)
ax1.set_ylabel('Total Calls in millions', fontsize=14)
ax1.set_yticks(range(0, 3500000, 500000))
ax1.set_yticklabels(y1_labels, fontsize=13)
ax1.grid(which='major', axis='y', linestyle=':', linewidth=0.5, color='gray', zorder=0)

ax2.set_xlabel('Day of Week', fontsize=14)
```

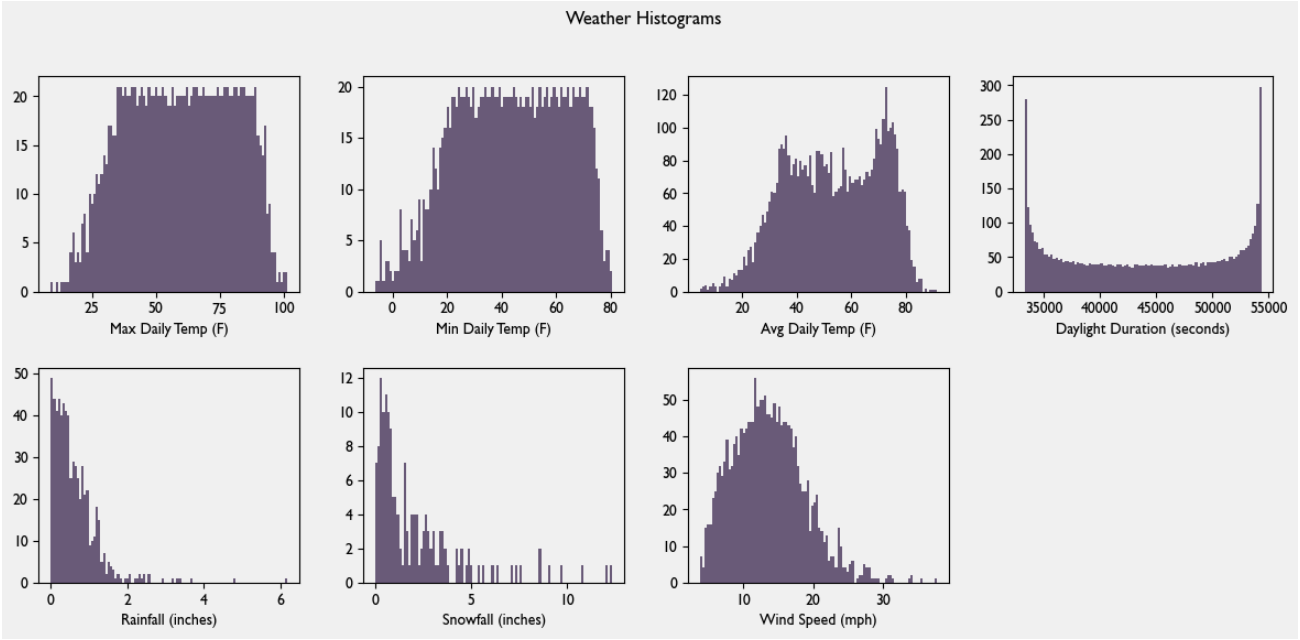
```
ax2.set_xticks(range(7))
ax2.set_xticklabels(x2_labels, fontsize=13)
ax2.set_ylabel('Total Calls in millions', fontsize=14)
ax2.set_yticks(range(0, 6000000, 1000000))
ax2.set_yticklabels(y2_labels, fontsize=13)
ax2.grid(which='major', axis='y', linestyle=':', linewidth=0.5, color='gray', zorder=0)

output_file = 'Charts/calls_by_month_and_day_of_week.jpg'
plt.savefig(output_file, dpi=300, bbox_inches='tight')
plt.show()
```



Weather

The visualization of the weather helps guide the optimal type of scaling.

In [9]:
```
# Visualize weather data
fig, ((ax1, ax2, ax3, ax4), (ax5, ax6, ax7, ax8)) = plt.subplots(2, 4, figsize=(12,6))
fig.suptitle('Weather Histograms')

ax1.hist(df_weather['temperature_2m_max'].value_counts().index, bins=100)
ax1.set_xlabel('Max Daily Temp (F)')
ax2.hist(df_weather['temperature_2m_min'].value_counts().index, bins=100)
ax2.set_xlabel('Min Daily Temp (F)')
ax3.hist(df_weather['temperature_2m_mean'].value_counts().index, bins=100)
ax3.set_xlabel('Avg Daily Temp (F)')
ax4.hist(df_weather['daylight_duration'].value_counts().index, bins=100)
ax4.set_xlabel('Daylight Duration (seconds)')
ax5.hist(df_weather['rain_sum'].value_counts().index, bins=100)
ax5.set_xlabel('Rainfall (inches)')
ax6.hist(df_weather['snowfall_sum'].value_counts().index, bins=100)
ax6.set_xlabel('Snowfall (inches)')
ax7.hist(df_weather['wind_speed_10m_max'].value_counts().index, bins=100)
ax7.set_xlabel('Wind Speed (mph)')
ax8.axis('off')
plt.tight_layout(pad=2.0)
plt.show;
```



Scaling

This data must be scaled for certain of the models, but the distributions vary. The images above suggest the following scaling approaches:

- Rainfall, snowfall and wind speed: Box-Cox transformation, addresses right-skewed distributions
- Temperatures - Temperatures are normally distributed when holding seasonality constant. Seasonal decompose first, then standard scale, then seasonal recompose
- Daylight Duration - Minmax scaling

In [10]:
```
# Scaling functions for weather data

# Log transform with a small constant
def log_transform(series, constant=1e-6):
    return np.log(series + constant)

# Helper function to apply Box-Cox transformation
def boxcox_transform(series):
    series_nonzero = series + 1e-6  # Add small constant to avoid zero values
    transformed_data, _ = boxcox(series_nonzero)
    return pd.Series(transformed_data, index=series.index)

# Seasonal decompose and recompose function
def decompose_and_scale(series, period=365):
    series = series.dropna()  # Drop NaNs if any
    decomposition = seasonal_decompose(series, model='additive', period=period)
    seasonal = decomposition.seasonal
    adjusted_series = series - seasonal  # Adjust the original series by removing the seasonal component

    scaler = StandardScaler()
    scaled_series = scaler.fit_transform(adjusted_series.values.reshape(-1, 1))

    scaled_series_full = pd.Series(scaled_series.flatten(), index=adjusted_series.index)
    scaled_series_full = scaled_series_full.reindex(series.index).bfill().ffill()

    return scaled_series_full
```

In [11]:
```
# Applying transformations

# Temperature columns: seasonal decompose, scale, recompose
df_weather['temperature_2m_max_scaled'] = decompose_and_scale(df_weather['temperature_2m_max'])
df_weather['temperature_2m_min_scaled'] = decompose_and_scale(df_weather['temperature_2m_min'])
df_weather['temperature_2m_mean_scaled'] = decompose_and_scale(df_weather['temperature_2m_mean'])

# Daylight duration: MinMax scale
df_weather['daylight_duration_scaled'] = MinMaxScaler().fit_transform(df_weather['daylight_duration'].values.reshape(-1, 1))

# Rainfall and Snowfall: Log transform with small constant
df_weather['rain_sum_log'] = boxcox_transform(df_weather['rain_sum'])
df_weather['snowfall_sum_log'] = boxcox_transform(df_weather['snowfall_sum'])

# Wind speed: Log transform
df_weather['wind_speed_10m_max_log'] = boxcox_transform(df_weather['wind_speed_10m_max'])
```

```
df_weather.drop(columns=['temperature_2m_max', 'temperature_2m_min', 'temperature_2m_mean',
                         'daylight_duration', 'rain_sum', 'snowfall_sum', 'wind_speed_10m_max'], inplace=True)

df_weather.rename(columns={
    'temperature_2m_max_scaled': 'temperature_2m_max',
    'temperature_2m_min_scaled': 'temperature_2m_min',
    'temperature_2m_mean_scaled': 'temperature_2m_mean',
    'daylight_duration_scaled': 'daylight_duration',
    'rain_sum_log': 'rain_sum',
    'snowfall_sum_log': 'snowfall_sum',
    'wind_speed_10m_max_log': 'wind_speed_10m_max'
}, inplace=True)
```

Modeling

A visual inspection of the dataset suggests that it has a strong autoregressive pattern, high and changing volatility, some weak seasonal patterns and likely some exogenous effects. In order to properly characterize this data, several models will be fitted:

- A baseline model will be established using the best of:
  1. an AR(1) model
  2. a random walk model, or
  3. an MA(1) model
- The first simple model will reflect best guess parameters for an ARIMA model based on a visual inspection of the data and the ACF and PACF graphs
- Using auto_arima, the ARIMA model will be optimized for pdq parameters
- The second model will reflect best guess parameters for a SARIMA model based on visual observation of the results and residuals of prior models and the ACF and PACF graphs of the residuals.
- Using auto_arima, the SARIMA model will be optimized for pdq and PDQ parameters
- Usine the best SARIMA model parameters, several exogenous regressors will be tested to determine whether they can enhance performance
- If heteroskedasticity is present, a GARCH model will be considered
- If strong seasonality is detected, Meta's Prophet model will be considered
- Finally, an LSTM model will be tested to determine whether newer, though less transparent, approaches can detect underlying patterns not found by other models

The value of forecasting to the 311 service and other agencies is to better predict necessary resources to respond to requests. The amount of resources necessary is most directly applicable to mean absolute error. However, the service should place heavier emphasis on outliers. Residents' dissatisfaction with government performance likely follows an exponential pattern, not a linear one. 20 minutes wait time is more than two times worse than 10 minutes. Two weeks for an agency to respond is more than twice as bad as one week. **Root mean squared error** captures both the scale of the problem and the importance of outliers. When using grid search to select parameters, **Akaike Information Criterion** will be used to select the winning combination.

Convert dataframe to time series

In [12]:
```
# Identify columns to drop
cols_to_drop = ['Agency', 'Type', 'Zip', 'Descriptor']

# Create Master dataset to use for modeling
df_311_dates_drop = df_311_calls.drop(columns=cols_to_drop, axis=1)
time_series = df_311_dates_drop.groupby('Date').size().reset_index(name='Count')
time_series.set_index('Date', inplace=True)

# Dummy for COVID period
lockdown1 = pd.date_range(start='2020-03-21', end='2020-06-06', inclusive='both')
lockdown2 = pd.date_range(start='2021-07-09', end='2021-10-27', inclusive='both')
lockdown3 = pd.date_range(start='2021-02-13', end='2021-02-17', inclusive='both')
lockdown4 = pd.date_range(start='2021-05-28', end='2021-06-10', inclusive='both')
lockdowns = lockdown1.union(lockdown2).union(lockdown3).union(lockdown4)

time_series['covid'] = time_series.index.isin(lockdowns) * 1
time_series['weekend'] = (time_series.index.dayofweek >= 5) * 1
time_series['311_app'] = (time_series.index >= pd.Timestamp('2013-03-19')) * 1

time_series = pd.concat([time_series, df_weather], axis=1)
```

Add population as a potential regressor

In [13]:
```
# Interpolate NYC population using US Census estimates
population_data = {
    'date': pd.to_datetime(['2010-01-01', '2010-07-01', '2011-07-01', '2012-07-01', '2013-07-01', '2014-07-01',
                            '2015-07-01', '2016-07-01', '2017-07-01', '2018-07-01', '2019-07-01',
                            '2020-04-01', '2020-07-01', '2021-07-01', '2022-07-01', '2023-07-01', '2024-03-31']),
    'population': [8175133, 8190209, 8251281, 8312676, 8374527, 8436839, 8499614, 8562857, 8626570, 8690757, 8755421, 8804199, 8740292,
}

# Create DataFrame
population_df = pd.DataFrame(population_data)
```

```
# Set the date as index
population_df.set_index('date', inplace=True)

# Generate a date range from 2010-01-01 to 2024-03-31
date_range = pd.date_range(start='2010-01-01', end='2024-03-31', freq='D')

# Reindex the population data to the full date range, using interpolation to fill in the gaps
population_daily_df = population_df.reindex(date_range)
population_daily_df['population'] = population_daily_df['population'].interpolate(method='linear')

# Scale population numbers
population_daily_df['population'] = MinMaxScaler().fit_transform(population_daily_df['population'].values.reshape(-1, 1))

time_series = pd.concat([time_series, population_daily_df], axis=1)
time_series = time_series.asfreq('D')
```

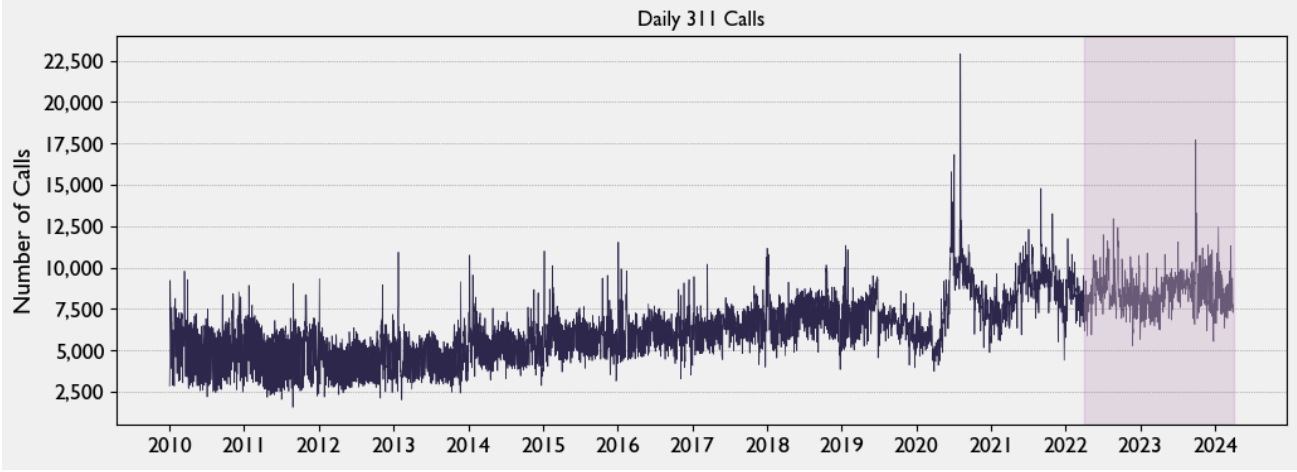
Test train split

The testing set will be limited to 2 full years, which should be sufficient for time series testing

In [14]:
```
test_size = 731         # two years plus one day

split_point = len(time_series) - test_size
train_df = time_series.iloc[:split_point]
test_df = time_series.iloc[split_point:]
```

Create image for time series and testing period

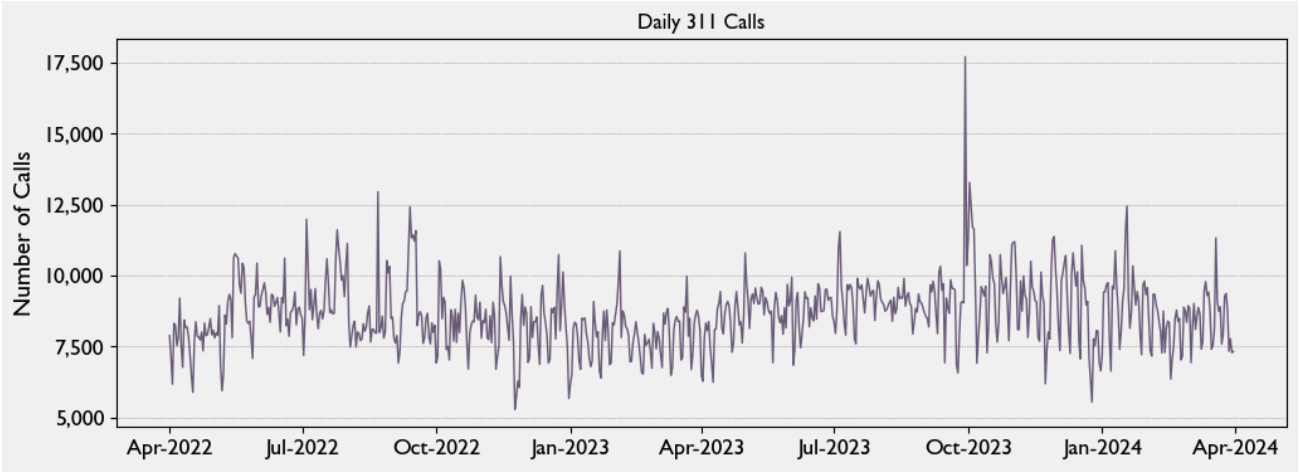
In [15]:
```
# Image: Test period shaded
fig, ax = plt.subplots(figsize=(12,4))
ax.plot(train_df['Count'], linewidth=.5, color='#2E294E')
ax.plot(test_df['Count'], linewidth=.5)
plt.setp(ax.get_xticklabels(), fontsize=13)
ax.set_ylabel('Number of Calls', fontsize=14)
plt.setp(ax.get_yticklabels(), fontsize=13)

ax.xaxis.set_major_locator(mdates.YearLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))

ax.yaxis.set_major_locator(MultipleLocator(2500))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, pos: f'{int(x):,}'))

ax.grid(which='major', axis='y', linestyle=':', linewidth=0.5, color='gray')
ax.axvspan(test_df.index[0], test_df.index[-1], color='purple',alpha=.1)
output_file = 'Charts/daily_call_shaded.jpg'
plt.savefig(output_file, dpi=300, bbox_inches='tight')
ax.set_title('Daily 311 Calls')
plt.show()
```



Create image for test period only

In [16]:
```
# Image: Test period only
fig, ax = plt.subplots(figsize=(12,4))
ax.plot(test_df['Count'], linewidth=1)
plt.setp(ax.get_xticklabels(), fontsize=13)
ax.set_ylabel('Number of Calls', fontsize=14)
plt.setp(ax.get_yticklabels(), fontsize=13)
```

```
ax.xaxis.set_major_formatter(mdates.DateFormatter('%b-%Y'))

ax.yaxis.set_major_locator(MultipleLocator(2500))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, pos: f'{int(x):,}'))

ax.grid(which='major', axis='y', linestyle=':', linewidth=.5, color='gray')
output_file = 'Charts/daily_call_test_period.jpg'
plt.savefig(output_file, dpi=300, bbox_inches='tight')
ax.set_title('Daily 311 Calls')
plt.show()
```



Helper functions for rolling forecasts and plotting

In [17]:
```
# Returns the final model, training/testing predictions and a confidence interval
def rolling_forecast(train_series: pd.Series,
                     test_series: pd.Series,
                     pdq: Tuple[int, int, int],
                     seasonal_pdq: Optional[Tuple[int, int, int, int]] = None,
                     exog_train: Optional[pd.DataFrame] = None,
                     exog_test: Optional[pd.DataFrame] = None,
                     maxiter: int = 50,
                     start_params: Optional[List[float]] = None,
                     forecast_horizon: int = 1,
                     initialization: Optional[str] = None) -> Union[Tuple[pd.Series, List[float], SARIMAX], Tuple[pd.Series, List[float

    history = list(train_series)
    exog_history = exog_train.values.tolist() if exog_train is not None else None
    predictions = []
    conf_intervals = []
    total_steps = len(test_series)
    start_time = time.time()
    train_pred = None

    for t in range(0, total_steps, forecast_horizon):
        if exog_history is not None:
            model = SARIMAX(history, exog=exog_history, order=pdq, seasonal_order=seasonal_pdq, initialization=initialization)
            model_fit = model.fit(disp=False, maxiter=maxiter, start_params=start_params)
            steps = min(forecast_horizon, total_steps - t)
            forecast_results = model_fit.get_forecast(steps=steps, exog=exog_test.iloc[t:t+steps])
            yhat = forecast_results.predicted_mean
            conf_int = forecast_results.conf_int(alpha=0.05)
            exog_history.extend(exog_test.iloc[t:t+steps].values)
            if t == 0:
                train_pred = model_fit.fittedvalues
        else:
            model = SARIMAX(history, order=pdq, seasonal_order=seasonal_pdq, initialization=initialization)
            model_fit = model.fit(disp=False, maxiter=maxiter, start_params=start_params)
            steps = min(forecast_horizon, total_steps - t)
            forecast_results = model_fit.get_forecast(steps=steps)  # Get forecast results object
            yhat = forecast_results.predicted_mean  # Extract the predicted mean (forecasts)
            conf_int = forecast_results.conf_int(alpha=0.05)
            if t == 0:
                train_pred = model_fit.fittedvalues

        predictions.extend(yhat)
        conf_intervals.extend(conf_int)
        history.extend(test_series.iloc[t:t+steps])

        if len(history) > len(train_series) + forecast_horizon:  # Limits training size to the initial length
            history = history[steps:]
```

```
            if exog_history is not None:
                exog_history = exog_history[steps:]

        gc.collect()  # Fx creates hundreds of models; releasing memory as each model is fitted improves computational performance

        if (t // forecast_horizon + 1) % (100 // forecast_horizon) == 0 or t + steps >= total_steps:
            elapsed_time = time.time() - start_time
            print(f"Progress: {min(t + forecast_horizon, total_steps)} out of {total_steps} steps completed in {elapsed_time} seconds")
            start_time = time.time()

    return train_pred, predictions, model_fit, conf_intervals
```

In [18]:
```
# Plots forecasts
def plot_forecasts(test_series, test_pred, conf_intervals):
    forecast_df = pd.DataFrame({
        'Actual': test_series,
        'Predicted': test_pred,
        'Lower CI': [ci[0] for ci in conf_intervals],
        'Upper CI': [ci[1] for ci in conf_intervals]
    }, index=test_series.index)

    fig, ax = plt.subplots(figsize=(12, 4))
    ax.plot(test_series, label='Actual Data', color='#6C5B7B', linewidth=.6)
    ax.scatter(x=test_series.index, y=test_pred, label='Prediction', color='black', marker='x', s=3)
    ax.yaxis.set_major_formatter(FuncFormatter(lambda x, pos: f'{int(x):,}'))
    ax.xaxis.set_major_locator(mdates.AutoDateLocator())
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%b %Y'))
    plt.setp(ax.get_xticklabels(), fontsize=13)
    plt.setp(ax.get_yticklabels(), fontsize=13)
    ax.grid(which='major', axis='y', linestyle=':', linewidth=.5, color='gray')
    ax.fill_between(forecast_df.index, forecast_df['Lower CI'], forecast_df['Upper CI'], color='purple', alpha=0.1, label='95% Confiden
    ax.set_ylabel('Count', fontsize=14)
    ax.legend(loc='upper right')
    return fig, ax
```

Choosing a Baseline Model

AR(1) Model

In [19]:
```
# ARIMA (1, 0, 0)
arima_100_train_pred, arima_100_test_pred, arima_100_model, arima_100_ci = rolling_forecast(train_df['Count'], test_df['Count'], (1, 0,
```

```
Progress: 100 out of 731 steps completed in 13.439925909042358 seconds
Progress: 200 out of 731 steps completed in 13.605478525161743 seconds
Progress: 300 out of 731 steps completed in 13.37204623222351 seconds
Progress: 400 out of 731 steps completed in 13.814006567001343 seconds
Progress: 500 out of 731 steps completed in 13.605412483215332 seconds
Progress: 600 out of 731 steps completed in 14.256211042404175 seconds
Progress: 700 out of 731 steps completed in 13.89442753791809 seconds
Progress: 731 out of 731 steps completed in 4.37993860244751 seconds
```

In [20]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], arima_100_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], arima_100_test_pred)}')
```

```
RMSE on training set: 1215.000642192471
RMSE on test set: 1129.6448233396275
```

In [21]:
```
arima_100_model.summary()
```

Out[21]:

SARIMAX Results

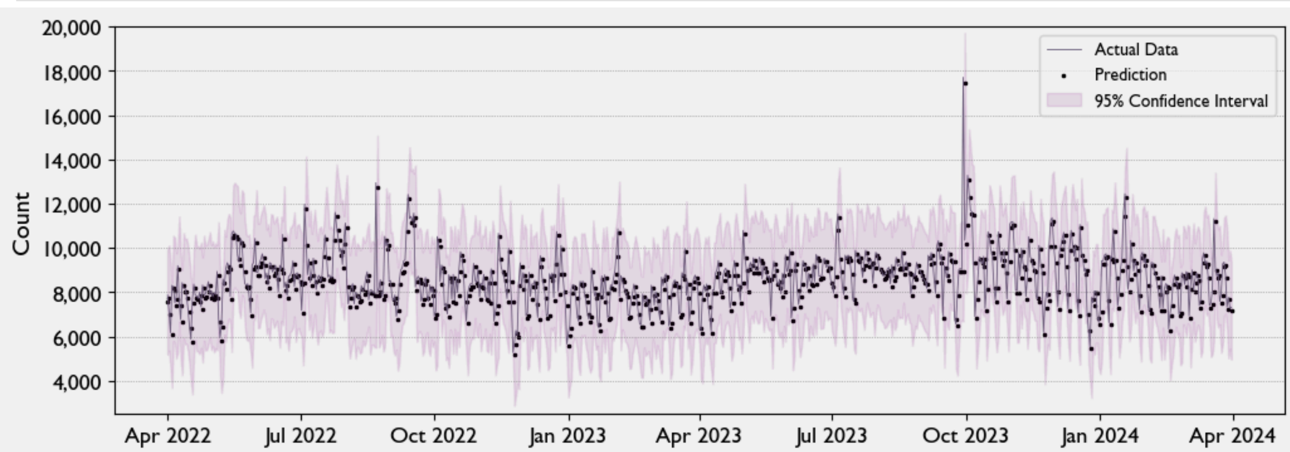
Dep. Variable:	y	**No. Observations:**	4474
Model:	SARIMAX(1, 0, 0)	**Log Likelihood**	-37801.673
Date:	Tue, 28 May 2024	**AIC**	75607.345
Time:	15:07:24	**BIC**	75620.157
Sample:	0	**HQIC**	75611.861
	- 4474		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.9871	0.002	422.386	0.000	0.983	0.992
sigma2	1.277e+06	1.3e+04	97.882	0.000	1.25e+06	1.3e+06

Ljung-Box (L1) (Q):	52.66	**Jarque-Bera (JB):**	9526.01
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	0.93	**Skew:**	0.96
Prob(H) (two-sided):	0.19	**Kurtosis:**	9.88

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

In [22]:
```
fig, ax = plot_forecasts(test_df['Count'],arima_100_test_pred, arima_100_ci)
ax.set_ylim(2500,20000)
output_file = 'Charts/conf_ar1.jpg'
fig.savefig(output_file, dpi=300, bbox_inches='tight')
plt.show()
```



Random Walk

In [23]:
```
# ARIMA (0, 1, 0)
arima_010_train_pred, arima_010_test_pred, arima_010_model, arima_010_ci = rolling_forecast(train_df['Count'], test_df['Count'], (0, 1,
```

```
C:\Users\rickl\AppData\Roaming\Python\Python312\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood opti
mization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
C:\Users\rickl\AppData\Roaming\Python\Python312\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood opti
mization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
C:\Users\rickl\AppData\Roaming\Python\Python312\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood opti
mization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
C:\Users\rickl\AppData\Roaming\Python\Python312\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood opti
mization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

```
Progress: 100 out of 731 steps completed in 20.15434503555298 seconds
Progress: 200 out of 731 steps completed in 16.749308824539185 seconds
Progress: 300 out of 731 steps completed in 16.005462169647217 seconds
Progress: 400 out of 731 steps completed in 15.823643445968628 seconds
Progress: 500 out of 731 steps completed in 16.365745544433594 seconds
Progress: 600 out of 731 steps completed in 16.37676453590393 seconds
Progress: 700 out of 731 steps completed in 16.01613688468933 seconds
Progress: 731 out of 731 steps completed in 5.0195112228393555 seconds
```

In [24]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], arima_010_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], arima_010_test_pred)}')
```

```
RMSE on training set: 1220.2428670532076
RMSE on test set: 1130.171820889745
```

In [25]:
```
arima_010_model.summary()
```

Out[25]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	4474
Model:	SARIMAX(0, 1, 0)	**Log Likelihood**	-37806.031
Date:	Tue, 28 May 2024	**AIC**	75614.061
Time:	15:09:26	**BIC**	75620.467
Sample:	0	**HQIC**	75616.319
	- 4474		
Covariance Type:	opg		

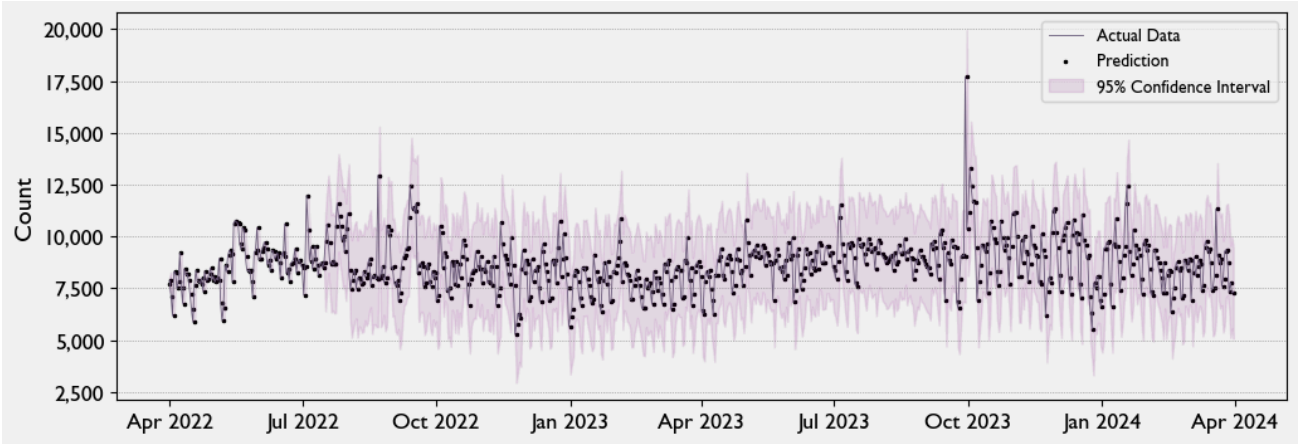
	coef	std err	z	P>|z|	[0.025	0.975]
sigma2	1.272e+06	1.27e+04	100.026	0.000	1.25e+06	1.3e+06

Ljung-Box (L1) (Q):	53.02	**Jarque-Bera (JB):**	9196.07
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	0.93	**Skew:**	0.94
Prob(H) (two-sided):	0.15	**Kurtosis:**	9.77

Warnings:

[1] Covariance matrix calculated using the outer product of gradients (complex-step).

In [26]:
```
fig, ax = plot_forecasts(test_df['Count'],arima_010_test_pred, arima_010_ci)
plt.show()
```



MA(1) Model

In [27]:
```
# ARIMA (0, 0, 1)
arima_001_train_pred, arima_001_test_pred, arima_001_model, arima_001_ci = rolling_forecast(train_df['Count'], test_df['Count'], (0, 0,
```

```
Progress: 100 out of 731 steps completed in 39.12517070770264 seconds
Progress: 200 out of 731 steps completed in 38.53307056427002 seconds
Progress: 300 out of 731 steps completed in 39.20037007331848 seconds
Progress: 400 out of 731 steps completed in 39.28607177734375 seconds
Progress: 500 out of 731 steps completed in 39.62921166419983 seconds
Progress: 600 out of 731 steps completed in 39.546029806137085 seconds
Progress: 700 out of 731 steps completed in 40.500346660614014 seconds
Progress: 731 out of 731 steps completed in 12.990298509597778 seconds
```

In [28]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], arima_001_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], arima_001_test_pred)}')
```

```
RMSE on training set: 3617.0004062264293
RMSE on test set: 4776.865055214732
```

In [29]:
```
arima_001_model.summary()
```

Out[29]:

SARIMAX Results

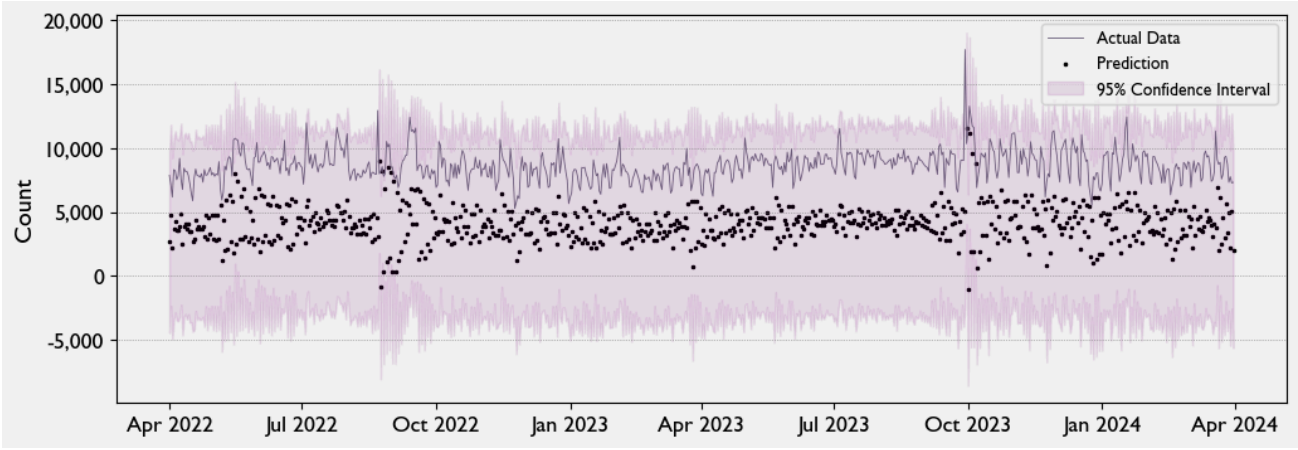
Dep. Variable:	y	**No. Observations:**	4474
Model:	SARIMAX(0, 0, 1)	**Log Likelihood**	-43345.309
Date:	Tue, 28 May 2024	**AIC**	86694.618
Time:	15:14:15	**BIC**	86707.430
Sample:	0	**HQIC**	86699.134
	- 4474		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ma.L1	0.9100	0.006	154.824	0.000	0.898	0.921
sigma2	1.522e+07	3.08e-11	4.94e+17	0.000	1.52e+07	1.52e+07

Ljung-Box (L1) (Q):	156.04	**Jarque-Bera (JB):**	1867.81
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	2.48	**Skew:**	0.66
Prob(H) (two-sided):	0.00	**Kurtosis:**	5.88

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

In [30]:
```
fig, ax = plot_forecasts(test_df['Count'],arima_001_test_pred, arima_001_ci)
plt.show()
```



AR(1) is the baseline: AR(1) has the lowest RMSE ~ 1130 and will serve as the comparison going forward. Also create 7-day and 30-day rolling forecasts for baselines.

In [31]:
```
# AR(1): 7-day Horizon
arima_100_7_train_pred, arima_100_7_test_pred, arima_100_7_model, arima_100_7_ci = rolling_forecast(train_df['Count'], test_df['Count']
                                                                                                    (1, 0, 0), forecast_horizon=7)
```

```
Progress: 98 out of 731 steps completed in 1.914280891418457 seconds
Progress: 196 out of 731 steps completed in 1.9516170024871826 seconds
Progress: 294 out of 731 steps completed in 1.9704492092132568 seconds
Progress: 392 out of 731 steps completed in 1.871297836303711 seconds
Progress: 490 out of 731 steps completed in 1.897594690322876 seconds
Progress: 588 out of 731 steps completed in 1.994077205657959 seconds
Progress: 686 out of 731 steps completed in 1.8918859958648682 seconds
Progress: 731 out of 731 steps completed in 0.9606997966766357 seconds
```

In [32]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], arima_100_7_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], arima_100_7_test_pred)}')
```

```
RMSE on training set: 1215.000642192471
RMSE on test set: 1428.025929742037
```

In [33]:
```
arima_100_7_model.summary()
```

Out[33]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	4480
Model:	SARIMAX(1, 0, 0)	**Log Likelihood**	-37855.703
Date:	Tue, 28 May 2024	**AIC**	75715.406
Time:	15:14:30	**BIC**	75728.220
Sample:	0	**HQIC**	75719.922
	- 4480		
Covariance Type:	opg		

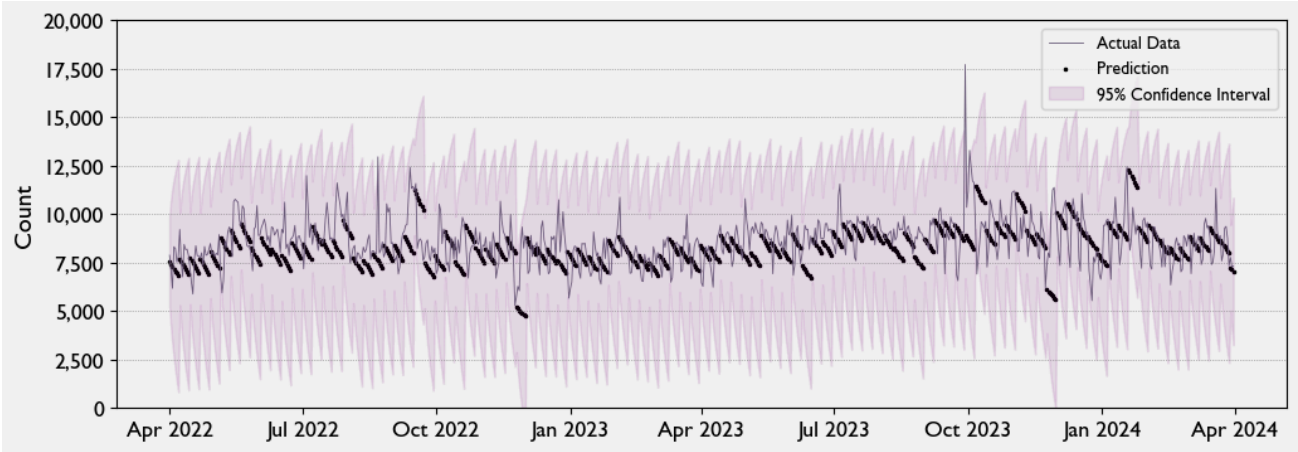
	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.9871	0.002	421.585	0.000	0.983	0.992
sigma2	1.279e+06	1.31e+04	97.774	0.000	1.25e+06	1.3e+06

Ljung-Box (L1) (Q):	51.20	**Jarque-Bera (JB):**	9446.16
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	0.93	**Skew:**	0.96
Prob(H) (two-sided):	0.18	**Kurtosis:**	9.85

Warnings:

[1] Covariance matrix calculated using the outer product of gradients (complex-step).

In [34]:
```
fig, ax = plot_forecasts(test_df['Count'],arima_100_7_test_pred, arima_100_7_ci)
ax.set_ylim(0,20000)
output_file = 'Charts/conf_ar1_7.jpg'
fig.savefig(output_file, dpi=300, bbox_inches='tight')
plt.show()
```



In [35]:
```
# AR(1): 30-day Horizon
arima_100_30_train_pred, arima_100_30_test_pred, arima_100_30_model, arima_100_30_ci = rolling_forecast(train_df['Count'], test_df['Cou
                                                                                                         (1, 0, 0), forecast_horizon=30)
```

```
Progress: 90 out of 731 steps completed in 0.4214968681335449 seconds
Progress: 180 out of 731 steps completed in 0.403456449508667 seconds
Progress: 270 out of 731 steps completed in 0.4310014247894287 seconds
Progress: 360 out of 731 steps completed in 0.4121086597442627 seconds
Progress: 450 out of 731 steps completed in 0.42978668212890625 seconds
Progress: 540 out of 731 steps completed in 0.4490022659301758 seconds
Progress: 630 out of 731 steps completed in 0.45873546600341797 seconds
Progress: 720 out of 731 steps completed in 0.41199827194213867 seconds
Progress: 731 out of 731 steps completed in 0.13300275802612305 seconds
```

In [36]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], arima_100_30_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], arima_100_30_test_pred)}')
```

```
RMSE on training set: 1215.000642192471
RMSE on test set: 2396.3714111558884
```

Note: this model performs badly on test data. Normally this is an indication of overfitting, but in this case, it indicates overforecasting. Typically, an ARIMA model should have at least as many terms as the forecast window. Otherwise, when it reaches the end of the forecast horizon, there is no longer any training data in the lookback period. A more appropriate baseline would be an ARIMA model with 30 terms, but any models that exceed 10 terms seem to be computationally prohibitive.

In [37]:
```
arima_100_30_model.summary()
```

Out[37]:

SARIMAX Results

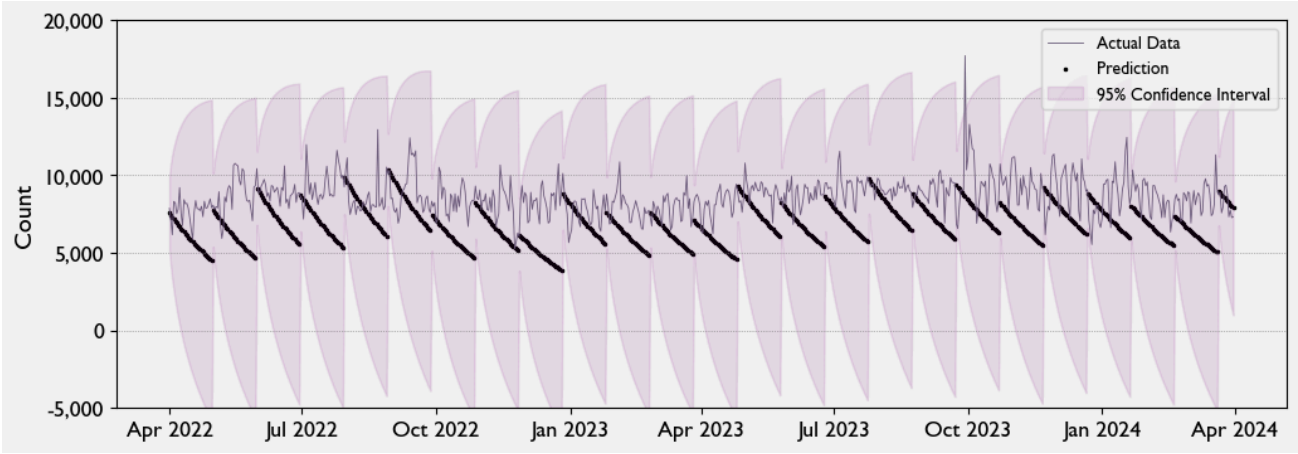
Dep. Variable:	y	**No. Observations:**	4503
Model:	SARIMAX(1, 0, 0)	**Log Likelihood**	-38058.293
Date:	Tue, 28 May 2024	**AIC**	76120.587
Time:	15:14:34	**BIC**	76133.412
Sample:	0	**HQIC**	76125.106
	- 4503		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.9872	0.002	419.910	0.000	0.983	0.992
sigma2	1.283e+06	1.32e+04	97.558	0.000	1.26e+06	1.31e+06

Ljung-Box (L1) (Q):	51.77	**Jarque-Bera (JB):**	9273.03
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	0.95	**Skew:**	0.95
Prob(H) (two-sided):	0.32	**Kurtosis:**	9.77

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

In [38]:
```
fig, ax = plot_forecasts(test_df['Count'],arima_100_30_test_pred, arima_100_30_ci)
ax.set_ylim(-5000,20000)
output_file = 'Charts/conf_ar1_30.jpg'
fig.savefig(output_file, dpi=300, bbox_inches='tight')
plt.show()
```



First Simple Model: ARIMA

ARIMA will look for autoregressive and moving average terms that will lead to improvements against the baseline. ARIMA models are applied to "stationary" data sets. The Augmented Dickey-Fuller test will test to determine whether this data is stationary. Large negative statistics, and p-values under 0.05, imply that the data *is* stationary.

In [39]:
```
result = adfuller(time_series['Count'])
print(f'ADF Statistic: {result[0]}')
print(f'p-value: {result[1]}')
```

```
ADF Statistic: -2.5208859123946588
p-value: 0.11047490316084208
```

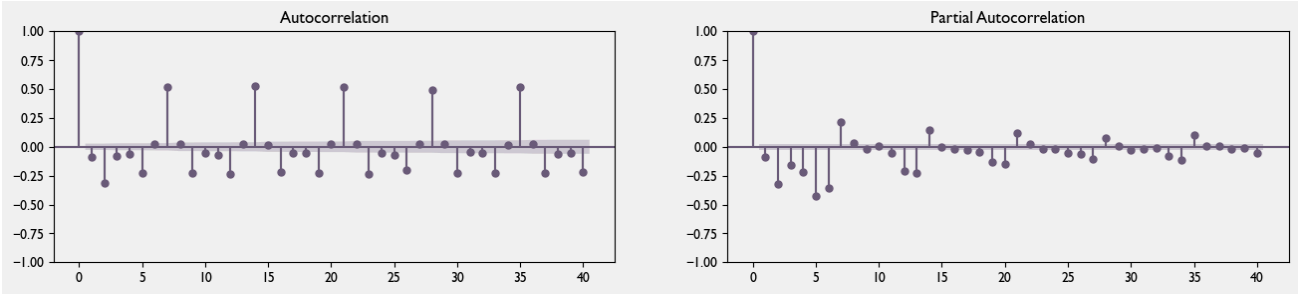
Not stationary. Because the p-value is not less than 0.05, the data is not stationary and must be transformed. The first transformation will use the one-day difference.

In [40]:
```
# Transform the data by taking the one-day difference
time_series['Count_diff'] = time_series['Count'].diff().dropna()
result_diff = adfuller(time_series['Count_diff'].dropna())
print(f'ADF Statistic (1st diff): {result_diff[0]}')
print(f'p-value (1st diff): {result_diff[1]}')
```

```
ADF Statistic (1st diff): -21.56754139806522
p-value (1st diff): 0.0
```

Stationary. The data is now stationary so the future 'd' term will be 1. Now, examine the ACF and PACF to look for good p and q terms, and check for seasonality.

In [41]:
```
fig, axes = plt.subplots(1, 2, figsize=(16, 3))

plot_acf(time_series['Count_diff'].dropna(), lags=40, ax=axes[0])
plot_pacf(time_series['Count_diff'].dropna(), lags=40, ax=axes[1]);
output_file = 'Charts/acf1.jpg'
fig.savefig(output_file, dpi=300, bbox_inches='tight')
```



The drop-off after 1 term in each chart suggests that p=1 and q=1 will lead to better results. However, the oscillation makes it difficult to determine what the right term actually will be. Notably, the 7-day pattern outside of the confidence interval suggests seasonality.

The simple first model, will use ARIMA (1,1,1), based on visual inspection of the ACF and PACF. Then a grid search will recommend the best set of 'pdq' to use.

In [43]:
```
# ARIMA (1, 1, 1)
arima_111_train_pred, arima_111_test_pred, arima_111_model, arima_111_ci = rolling_forecast(train_df['Count'], test_df['Count'], (1, 1,
```

```
Progress: 100 out of 731 steps completed in 50.458789348602295 seconds
Progress: 200 out of 731 steps completed in 50.33251905441284 seconds
Progress: 300 out of 731 steps completed in 52.18584394454956 seconds
Progress: 400 out of 731 steps completed in 51.40514326095581 seconds
Progress: 500 out of 731 steps completed in 52.44851207733154 seconds
Progress: 600 out of 731 steps completed in 50.29031157493591 seconds
Progress: 700 out of 731 steps completed in 51.97581076622009 seconds
Progress: 731 out of 731 steps completed in 16.74971652030945 seconds
```

In [44]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], arima_111_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], arima_111_test_pred)}')
```

```
RMSE on training set: 1067.694664773531
RMSE on test set: 989.4160484574766
```

In [45]:
```
arima_111_model.summary()
```

Out[45]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	4474
Model:	SARIMAX(1, 1, 1)	**Log Likelihood**	-37235.882
Date:	Tue, 28 May 2024	**AIC**	74477.765
Time:	15:33:37	**BIC**	74496.982
Sample:	0	**HQIC**	74484.538
	- 4474		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.4779	0.012	40.893	0.000	0.455	0.501
ma.L1	-0.9522	0.004	-239.891	0.000	-0.960	-0.944
sigma2	9.95e+05	9102.446	109.311	0.000	9.77e+05	1.01e+06

Ljung-Box (L1) (Q):	43.09	**Jarque-Bera (JB):**	16712.19
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.01	**Skew:**	1.20
Prob(H) (two-sided):	0.91	**Kurtosis:**	12.16

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

ARIMA(1, 1, 1) is the new best model. At RMSE = 989, this model is a significant improvement. Grid search will look for a better combination.

In [46]:
```
arima_model = auto_arima(train_df['Count'], seasonal=False, trace=True, error_action='ignore', suppress_warnings=True, stepwise=True)
```

```
Performing stepwise search to minimize aic
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=74701.430, Time=2.37 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=76253.783, Time=0.03 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=76231.363, Time=0.08 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=75948.244, Time=0.80 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=76251.787, Time=0.02 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=74749.518, Time=1.42 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=74696.394, Time=1.42 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=75058.083, Time=1.18 sec
 ARIMA(2,1,0)(0,0,0)[0] intercept   : AIC=75711.968, Time=0.16 sec
 ARIMA(3,1,1)(0,0,0)[0] intercept   : AIC=74696.933, Time=2.00 sec
 ARIMA(3,1,0)(0,0,0)[0] intercept   : AIC=75602.503, Time=0.22 sec
 ARIMA(3,1,2)(0,0,0)[0] intercept   : AIC=74685.534, Time=2.55 sec
 ARIMA(4,1,2)(0,0,0)[0] intercept   : AIC=74673.564, Time=2.93 sec
 ARIMA(4,1,1)(0,0,0)[0] intercept   : AIC=74535.786, Time=1.80 sec
 ARIMA(4,1,0)(0,0,0)[0] intercept   : AIC=75366.610, Time=0.32 sec
 ARIMA(5,1,1)(0,0,0)[0] intercept   : AIC=74037.657, Time=0.73 sec
 ARIMA(5,1,0)(0,0,0)[0] intercept   : AIC=74353.513, Time=0.52 sec
 ARIMA(5,1,2)(0,0,0)[0] intercept   : AIC=73623.808, Time=3.48 sec
 ARIMA(5,1,3)(0,0,0)[0] intercept   : AIC=73418.219, Time=4.04 sec
 ARIMA(4,1,3)(0,0,0)[0] intercept   : AIC=73836.056, Time=3.75 sec
 ARIMA(5,1,4)(0,0,0)[0] intercept   : AIC=73186.144, Time=4.86 sec
 ARIMA(4,1,4)(0,0,0)[0] intercept   : AIC=73654.457, Time=3.89 sec
 ARIMA(5,1,5)(0,0,0)[0] intercept   : AIC=73514.618, Time=5.60 sec
 ARIMA(4,1,5)(0,0,0)[0] intercept   : AIC=73126.563, Time=4.85 sec
 ARIMA(3,1,5)(0,0,0)[0] intercept   : AIC=73926.457, Time=4.35 sec
 ARIMA(3,1,4)(0,0,0)[0] intercept   : AIC=73808.655, Time=4.02 sec
 ARIMA(4,1,5)(0,0,0)[0]             : AIC=73126.294, Time=4.35 sec
 ARIMA(3,1,5)(0,0,0)[0]             : AIC=73918.501, Time=2.68 sec
 ARIMA(4,1,4)(0,0,0)[0]             : AIC=73654.777, Time=2.91 sec
 ARIMA(5,1,5)(0,0,0)[0]             : AIC=73521.969, Time=2.66 sec
 ARIMA(3,1,4)(0,0,0)[0]             : AIC=73842.944, Time=2.31 sec
 ARIMA(5,1,4)(0,0,0)[0]             : AIC=73185.083, Time=3.46 sec

Best model:  ARIMA(4,1,5)(0,0,0)[0]
Total fit time: 75.767 seconds
```

In [47]:
```
# ARIMA (4, 1, 5)
arima_415_train_pred, arima_415_test_pred, arima_415_model, arima_415_ci = rolling_forecast(train_df['Count'], test_df['Count'], (4, 1,
```

```
C:\Users\rickl\AppData\Roaming\Python\Python312\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood opti
mization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
C:\Users\rickl\AppData\Roaming\Python\Python312\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood opti
mization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
Progress: 100 out of 731 steps completed in 1439.6041560173035 seconds
Progress: 200 out of 731 steps completed in 1220.1436567306519 seconds
C:\Users\rickl\AppData\Roaming\Python\Python312\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood opti
mization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
C:\Users\rickl\AppData\Roaming\Python\Python312\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood opti
mization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
Progress: 300 out of 731 steps completed in 1099.3279211521149 seconds
Progress: 400 out of 731 steps completed in 1263.8792612552643 seconds
Progress: 500 out of 731 steps completed in 1233.1007583141327 seconds
Progress: 600 out of 731 steps completed in 1043.312388420105 seconds
Progress: 700 out of 731 steps completed in 1259.5967481136322 seconds
Progress: 731 out of 731 steps completed in 378.21850752830505 seconds
```

In [48]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], arima_415_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], arima_415_test_pred)}')
```

```
RMSE on training set: 826.3183011874227
RMSE on test set: 924.4766061513161
```

In [49]:
```
arima_415_model.summary()
```

Out[49]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	4474
Model:	SARIMAX(4, 1, 5)	**Log Likelihood**	-36534.019
Date:	Tue, 28 May 2024	**AIC**	73088.038
Time:	18:03:50	**BIC**	73152.096
Sample:	0	**HQIC**	73110.616
	- 4474		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.8019	0.003	309.094	0.000	0.797	0.807
ar.L2	-1.4403	0.003	-498.358	0.000	-1.446	-1.435
ar.L3	0.7960	0.003	235.301	0.000	0.789	0.803
ar.L4	-0.9948	0.002	-503.762	0.000	-0.999	-0.991
ma.L1	-1.3215	0.010	-127.362	0.000	-1.342	-1.301
ma.L2	1.7877	0.012	143.239	0.000	1.763	1.812
ma.L3	-1.5130	0.019	-79.504	0.000	-1.550	-1.476
ma.L4	1.3277	0.014	95.660	0.000	1.301	1.355
ma.L5	-0.5169	0.012	-43.333	0.000	-0.540	-0.494
sigma2	9.313e+05	8608.808	108.185	0.000	9.14e+05	9.48e+05

Ljung-Box (L1) (Q):	42.41	**Jarque-Bera (JB):**	95364.61
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.77	**Skew:**	1.59
Prob(H) (two-sided):	0.00	**Kurtosis:**	25.40

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 6.88e+14. Standard errors may be unstable.

Significant improvement. At RMSE = 924, the optimal ARIMA model is modestly better than ARIMA(1,1,1). It seems as if there is a little overfitting. The next model will attempt to improve performance by taking account of seasonality.

SARIMA for seasonality

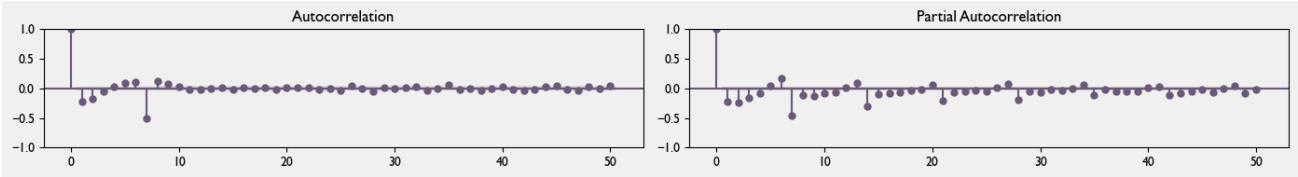
The ACF/PACF plots suggested seasonality. SARIMA is a form of ARIMA model that detects and models seasonality as well (though only one form). Like ARIMA, SARIMA requires stationary data.

In [50]:
```
time_series['week_diff'] = time_series['Count_diff'].diff(7)
results_adf_7d = adfuller(time_series['week_diff'].dropna())
print(f'ADF Statistic (1st diff): {results_adf_7d[0]}')
print(f'p-value (1st diff): {results_adf_7d[1]}')
```

```
ADF Statistic (1st diff): -21.428598158127986
p-value (1st diff): 0.0
```

Stationary. 7-day differencing results in stationarity. 'D' term will be 1 as well.

In [51]:
```
# Plot ACF and PACF of seasonally differenced residuals
fig, ax = plt.subplots(1, 2, figsize=(16, 3))
plot_acf(time_series['week_diff'].dropna(), ax=ax[0], lags=50)
plot_pacf(time_series['week_diff'].dropna(), ax=ax[1], lags=50)
plt.tight_layout()
output_file = 'Charts/acf2.jpg'
plt.savefig(output_file, dpi=300, bbox_inches='tight')
plt.show()
```



MA(Q) = 1: The ACF chart rebounds to zero by the second week.
AR(P) = 1: The PACF chart appears to rebound to zero by the sixth week, which is a very high order. The P term cannot be determined from this chart.

To be conservative, the first SARIMA model will be (1,1,1) (1,1,1) with 7-day seasonality.

In [52]:
```
sarima_111x111_train_pred, sarima_111x111_test_pred, sarima_111x111_model, sarima_111x111_ci = rolling_forecast(train_df['Count'], test
                                                                                                                 (1, 1, 1), (1, 1, 1, 7)
```

```
C:\Users\rickl\AppData\Roaming\Python\Python312\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood opti
mization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
Progress: 100 out of 731 steps completed in 420.5768482685089 seconds
Progress: 200 out of 731 steps completed in 414.9523684978485 seconds
Progress: 300 out of 731 steps completed in 373.4073123931885 seconds
Progress: 400 out of 731 steps completed in 386.60623836517334 seconds
Progress: 500 out of 731 steps completed in 359.1750192642212 seconds
Progress: 600 out of 731 steps completed in 389.8274521827698 seconds
Progress: 700 out of 731 steps completed in 375.50733733177185 seconds
Progress: 731 out of 731 steps completed in 122.92383003234863 seconds
```

In [53]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], sarima_111x111_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], sarima_111x111_test_pred)}')
```

```
RMSE on training set: 779.2310117438527
RMSE on test set: 1477931576.2864265
```

Glitch. This test predicted 40 billion calls for one date, which was certainly a technical glitch. Rather than rerun the model, this value was removed from the predictions and the actuals for this test alone.

In [56]:
```
max_pred_index = np.argmax(sarima_111x111_test_pred)
max_pred_value = sarima_111x111_test_pred[max_pred_index]
max_pred_date = test_df.index[max_pred_index]
max_actual_value = test_df['Count'].iloc[max_pred_index]
test_df_filtered = test_df.drop(index=max_pred_date)
sarima_111x111_test_pred_filtered = np.delete(sarima_111x111_test_pred, max_pred_index)

print(f"Removed value for {max_pred_date}: {sarima_111x111_test_pred[max_pred_index]}")

# Recalculate RMSE
rmse_test_filtered = root_mean_squared_error(test_df_filtered['Count'], sarima_111x111_test_pred_filtered)
print(f'RMSE on test set (without problematic value): {rmse_test_filtered}')
```

```
Removed value for 2022-06-12 00:00:00: 39958862192.31543
RMSE on test set (without problematic value): 896.8257420747296
```

In [57]:
```
sarima_111x111_model.summary()
```

Out[57]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	4474
Model:	SARIMAX(1, 1, 1)x(1, 1, 1, 7)	**Log Likelihood**	-36115.146
Date:	Tue, 28 May 2024	**AIC**	72240.293
Time:	20:24:12	**BIC**	72272.314
Sample:	0	**HQIC**	72251.580
	- 4474		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.5246	0.010	51.037	0.000	0.504	0.545
ma.L1	-0.9437	0.005	-203.191	0.000	-0.953	-0.935
ar.S.L7	0.0598	0.012	4.947	0.000	0.036	0.083
ma.S.L7	-0.9554	0.004	-244.505	0.000	-0.963	-0.948
sigma2	6.165e+05	3594.651	171.496	0.000	6.09e+05	6.24e+05

Ljung-Box (L1) (Q):	0.46	**Jarque-Bera (JB):**	117042.71
Prob(Q):	0.50	**Prob(JB):**	0.00
Heteroskedasticity (H):	2.12	**Skew:**	1.84
Prob(H) (two-sided):	0.00	**Kurtosis:**	27.81

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

SARIMA(1,1,1) (1,1,1,7). At RMSE=896, SARIMA (1,1,1) x (1,1,1,7) improves modestly on ARIMA. There is still a little overfitting. A grid search will look for better parameters.

In [58]:
```
sarima_opt_model = auto_arima(train_df['Count'],
                              seasonal=True,
                              m=7,
                              max_P = 5,
                              max_Q = 5,
                              trace=True,
                              error_action='ignore',
                              suppress_warnings=True,
                              stepwise=True)
```

```
Performing stepwise search to minimize aic
 ARIMA(2,1,2)(1,0,1)[7] intercept   : AIC=inf, Time=4.95 sec
 ARIMA(0,1,0)(0,0,0)[7] intercept   : AIC=76253.783, Time=0.03 sec
 ARIMA(1,1,0)(1,0,0)[7] intercept   : AIC=74468.999, Time=1.20 sec
 ARIMA(0,1,1)(0,0,1)[7] intercept   : AIC=75101.537, Time=1.29 sec
 ARIMA(0,1,0)(0,0,0)[7]             : AIC=76251.787, Time=0.02 sec
 ARIMA(1,1,0)(0,0,0)[7] intercept   : AIC=76231.363, Time=0.08 sec
 ARIMA(1,1,0)(2,0,0)[7] intercept   : AIC=73787.767, Time=1.92 sec
 ARIMA(1,1,0)(3,0,0)[7] intercept   : AIC=73496.393, Time=4.93 sec
 ARIMA(1,1,0)(4,0,0)[7] intercept   : AIC=73365.741, Time=8.41 sec
 ARIMA(1,1,0)(5,0,0)[7] intercept   : AIC=inf, Time=24.16 sec
 ARIMA(1,1,0)(4,0,1)[7] intercept   : AIC=73825.052, Time=18.06 sec
 ARIMA(1,1,0)(3,0,1)[7] intercept   : AIC=inf, Time=11.51 sec
 ARIMA(1,1,0)(5,0,1)[7] intercept   : AIC=inf, Time=40.08 sec
 ARIMA(0,1,0)(4,0,0)[7] intercept   : AIC=73553.214, Time=6.06 sec
 ARIMA(2,1,0)(4,0,0)[7] intercept   : AIC=73085.325, Time=19.24 sec
 ARIMA(2,1,0)(3,0,0)[7] intercept   : AIC=73220.229, Time=6.44 sec
 ARIMA(2,1,0)(5,0,0)[7] intercept   : AIC=inf, Time=35.93 sec
 ARIMA(2,1,0)(4,0,1)[7] intercept   : AIC=72644.026, Time=27.22 sec
 ARIMA(2,1,0)(3,0,1)[7] intercept   : AIC=73416.894, Time=11.78 sec
 ARIMA(2,1,0)(5,0,1)[7] intercept   : AIC=inf, Time=nan sec
 ARIMA(2,1,0)(4,0,2)[7] intercept   : AIC=73354.772, Time=24.27 sec
 ARIMA(2,1,0)(3,0,2)[7] intercept   : AIC=73061.812, Time=13.32 sec
 ARIMA(2,1,0)(5,0,2)[7] intercept   : AIC=73248.181, Time=38.70 sec
 ARIMA(3,1,0)(4,0,1)[7] intercept   : AIC=73165.174, Time=24.39 sec
 ARIMA(2,1,1)(4,0,1)[7] intercept   : AIC=inf, Time=28.92 sec
 ARIMA(1,1,1)(4,0,1)[7] intercept   : AIC=inf, Time=22.76 sec
 ARIMA(3,1,1)(4,0,1)[7] intercept   : AIC=inf, Time=23.82 sec
 ARIMA(2,1,0)(4,0,1)[7]             : AIC=inf, Time=16.40 sec

Best model:  ARIMA(2,1,0)(4,0,1)[7] intercept
Total fit time: 450.290 seconds
```

In [59]:
```
sarima_210x401_model = sarima_opt_model
sarima_210x401_train_pred, sarima_210x401_test_pred, sarima_210x401_model, sarima_210x401_ci = rolling_forecast(train_df['Count'], test
                                                                                                                (2, 1, 0), (4, 0, 1, 7)
                                                                                                                 initialization='approx
```

```
Progress: 100 out of 731 steps completed in 1182.3575057983398 seconds
Progress: 200 out of 731 steps completed in 1238.791585445404 seconds
Progress: 300 out of 731 steps completed in 1299.5681116580963 seconds
Progress: 400 out of 731 steps completed in 1289.4275753498077 seconds
Progress: 500 out of 731 steps completed in 1420.9871108531952 seconds
Progress: 600 out of 731 steps completed in 1407.107075214386 seconds
Progress: 700 out of 731 steps completed in 1308.4783263206482 seconds
Progress: 731 out of 731 steps completed in 398.8554172515869 seconds
```

In [60]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], sarima_210x401_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], sarima_210x401_test_pred)}')
```

```
RMSE on training set: 808.5525694348487
RMSE on test set: 943.0612756521538
```

In [ ]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], sarima_203x111_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], sarima_203x111_test_pred)}')
```

SARIMAX (1,1,1) x (1,1,1,7) still best. The grid search found parameters that performed very well on the training set, but performed the same as our lower order SARIMAX (1,1,1) x (1,1,1,7) model. The additional terms likely result in overfitting. For simplicity, the lower order model will remain the one to beat.

SARIMAX for Exogenous Regressors

The SARIMAX model is an extension of the SARIMA model that accounts for exogenous variables. This calculation will include variables for COVID lockdowns, population, weekends, the introduction of the 311 App, winter months and different weather metrics

The first iteration will include the winning SARIMA model and will add all meaningful exogenous variables. All variables have been scaled:

- Daily minimum, maximum and average temperatures in Fahrenheit
- Daily rainfall and snowfall in inches
- Daily sunlight in seconds
- Daily wind speed in mph
- Whether the day is a weekend
- Whether there was a COVID lockdown occurring

In [62]:
```
exog_cols = ['covid', 'rain_sum', 'temperature_2m_min', 'wind_speed_10m_max', 'daylight_duration', 'temperature_2m_mean']
model = SARIMAX(train_df['Count'], train_df[exog_cols], order=(1,1,1), seasonal_order=(1,1,1,7))
```

```
model_fit = model.fit(disp=False, maxiter=500)
model_fit.summary()
```

Out[62]:

SARIMAX Results

Dep. Variable:	Count	**No. Observations:**	4473
Model:	SARIMAX(1, 1, 1)x(1, 1, 1, 7)	**Log Likelihood**	-35839.682
Date:	Tue, 28 May 2024	**AIC**	71701.364
Time:	23:33:14	**BIC**	71771.808
Sample:	01-01-2010	**HQIC**	71726.195
	- 03-31-2022		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
covid	-420.6041	131.870	-3.190	0.001	-679.065	-162.143
rain_sum	-8.1612	1.363	-5.990	0.000	-10.832	-5.491
temperature_2m_min	-101.4405	30.871	-3.286	0.001	-161.947	-40.934
wind_speed_10m_max	161.6836	36.765	4.398	0.000	89.625	233.742
daylight_duration	826.1255	305.659	2.703	0.007	227.044	1425.207
temperature_2m_mean	-47.6936	27.976	-1.705	0.088	-102.526	7.139
ar.L1	0.5052	0.011	46.573	0.000	0.484	0.526
ma.L1	-0.9378	0.005	-178.603	0.000	-0.948	-0.927
ar.S.L7	0.0444	0.013	3.336	0.001	0.018	0.070
ma.S.L7	-0.9503	0.004	-230.772	0.000	-0.958	-0.942
sigma2	5.461e+05	4089.207	133.546	0.000	5.38e+05	5.54e+05

Ljung-Box (L1) (Q):	1.28	**Jarque-Bera (JB):**	125253.28
Prob(Q):	0.26	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.55	**Skew:**	1.57
Prob(H) (two-sided):	0.00	**Kurtosis:**	28.76

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

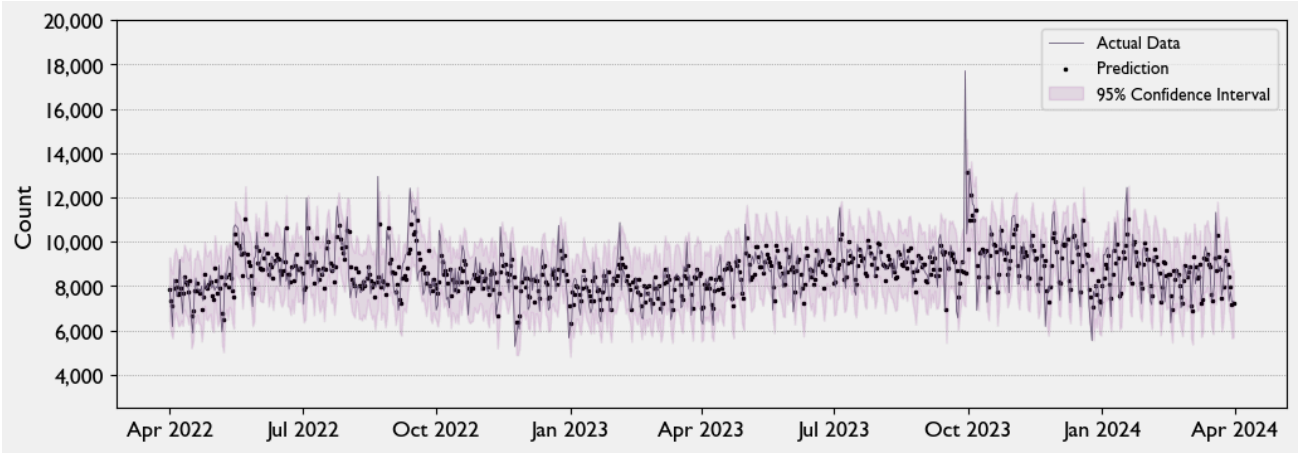
In [63]:
```
sarimax_train_pred, sarimax_test_pred, sarimax_model, sarimax_ci = rolling_forecast(train_df['Count'], test_df['Count'],
                                                                                    (1,1,1), (1,1,1,7),
                                                                                    train_df[exog_cols], test_df[exog_cols],
                                                                                    maxiter=500)
```

```
Progress: 100 out of 731 steps completed in 1019.5230305194855 seconds
Progress: 200 out of 731 steps completed in 1020.7225739955902 seconds
Progress: 300 out of 731 steps completed in 970.3168959617615 seconds
Progress: 400 out of 731 steps completed in 1055.8779208660126 seconds
Progress: 500 out of 731 steps completed in 1066.437783241272 seconds
Progress: 600 out of 731 steps completed in 1078.0359163284302 seconds
Progress: 700 out of 731 steps completed in 1087.6814856529236 seconds
Progress: 731 out of 731 steps completed in 319.03681683540344 seconds
```

In [64]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], sarimax_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], sarimax_test_pred)}')
```

```
RMSE on training set: 746.9508779031821
RMSE on test set: 878.9548576248286
```

In [65]:
```
fig, ax = plot_forecasts(test_df['Count'],sarimax_test_pred, sarimax_ci)
ax.set_ylim(2500,20000)
output_file = 'Charts/conf_sarimax.jpg'
fig.savefig(output_file, dpi=300, bbox_inches='tight')
plt.show()
```



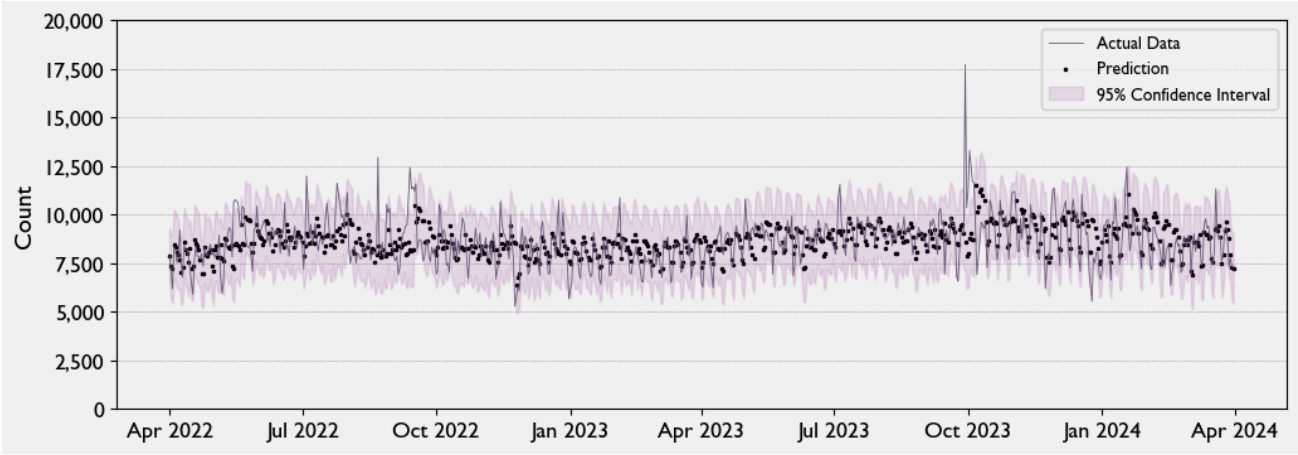
In [66]:
```
sarimax_7_train_pred, sarimax_7_test_pred, sarimax_7_model, sarimax_7_ci = rolling_forecast(train_df['Count'], test_df['Count'],
                                                                                            (1,1,1), (1,1,1,7),
                                                                                            train_df[exog_cols], test_df[exog_cols],
                                                                                            maxiter=500, forecast_horizon=7)
```

```
Progress: 98 out of 731 steps completed in 139.20409440994263 seconds
Progress: 196 out of 731 steps completed in 153.78534603118896 seconds
Progress: 294 out of 731 steps completed in 133.81015610694885 seconds
Progress: 392 out of 731 steps completed in 141.61634421348572 seconds
Progress: 490 out of 731 steps completed in 144.23890590667725 seconds
Progress: 588 out of 731 steps completed in 157.63952589035034 seconds
Progress: 686 out of 731 steps completed in 147.4912359714508 seconds
Progress: 731 out of 731 steps completed in 72.22724986076355 seconds
```

In [67]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], sarimax_7_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], sarimax_7_test_pred)}')
```

```
RMSE on training set: 746.9508779031821
RMSE on test set: 1053.1236930365046
```

In [68]:
```
fig, ax = plot_forecasts(test_df['Count'],sarimax_7_test_pred, sarimax_7_ci)
ax.set_ylim(0, 20000)
output_file = 'Charts/conf_sarimax_7.jpg'
fig.savefig(output_file, dpi=300, bbox_inches='tight')
plt.show()
```



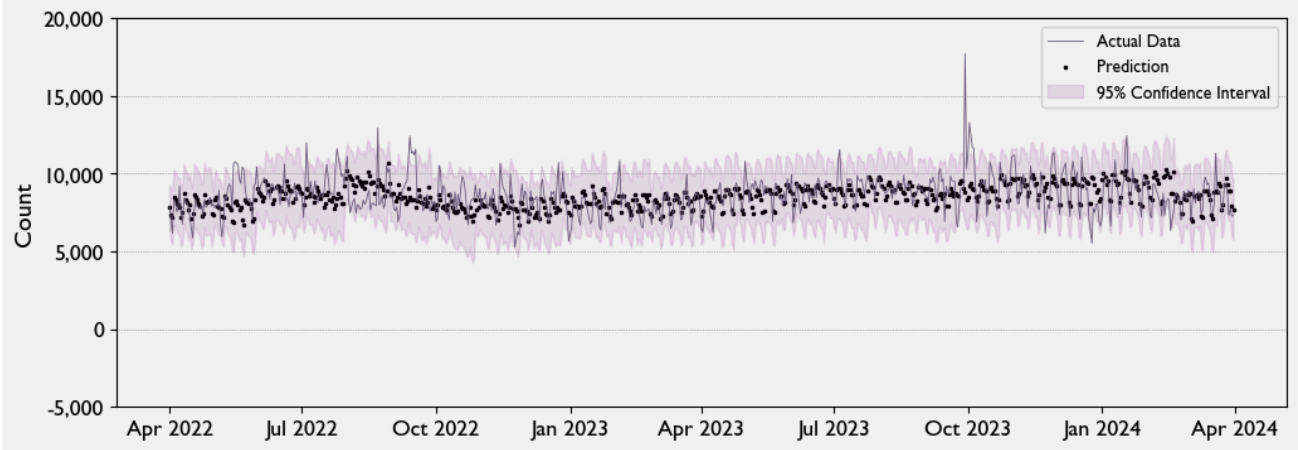
In [69]:
```
sarimax_30_train_pred, sarimax_30_test_pred, sarimax_30_model, sarimax_30_ci = rolling_forecast(train_df['Count'], test_df['Count'],
                                                                                                (1,1,1), (1,1,1,7),
                                                                                                train_df[exog_cols], test_df[exog_cols]
                                                                                                maxiter=500, forecast_horizon=30)
```

```
Progress: 90 out of 731 steps completed in 28.50430941581726 seconds
Progress: 180 out of 731 steps completed in 31.82515859603882 seconds
Progress: 270 out of 731 steps completed in 27.454691410064697 seconds
Progress: 360 out of 731 steps completed in 36.7955961227417 seconds
Progress: 450 out of 731 steps completed in 33.52565097808838 seconds
Progress: 540 out of 731 steps completed in 32.6707558631897 seconds
Progress: 630 out of 731 steps completed in 33.23807740211487 seconds
Progress: 720 out of 731 steps completed in 31.327019453048706 seconds
Progress: 731 out of 731 steps completed in 9.936856508255005 seconds
```

In [70]:
```
print(f'RMSE on training set: {root_mean_squared_error(train_df['Count'], sarimax_30_train_pred)}')
print(f'RMSE on test set: {root_mean_squared_error(test_df['Count'], sarimax_30_test_pred)}')
```

```
RMSE on training set: 746.9508779031821
RMSE on test set: 1095.1801155128749
```

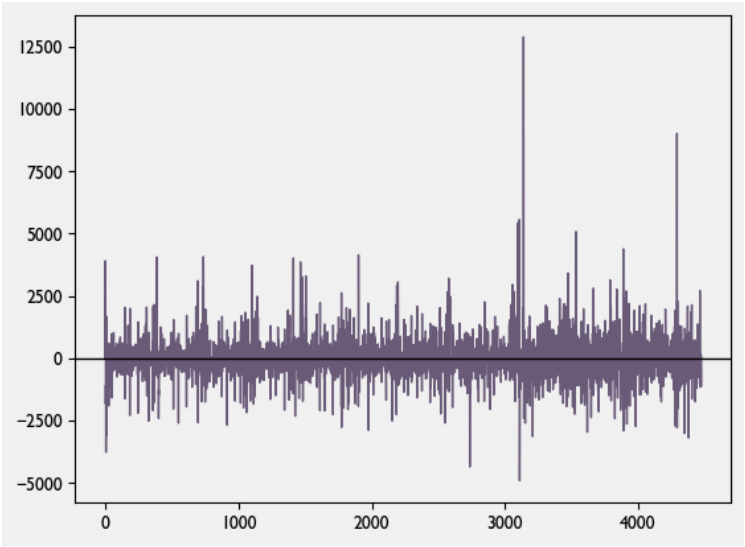
In [71]:
```
fig, ax = plot_forecasts(test_df['Count'],sarimax_30_test_pred, sarimax_30_ci)
ax.set_ylim(-5000,20000)
output_file = 'Charts/conf_sarimax_30.jpg'
fig.savefig(output_file, dpi=300, bbox_inches='tight')
plt.show()
```



SARIMAX(1,1,1)x(1,1,1,7) is the winning model. Other orders seem only to degrade performance and overfit. Further improvements would require either significantly higher orders, better detection of seasonalities, or additional exogenous information. LSTM will check for additional patterns not captured by SARIMAX. Prophet will test whether additional seasonalities and holiday effects can improve performance.

The summary statistics show strong evidence of heteroskedasticity. A plot of the residuals demonstrates that as well. In the future, using a GARCH model would create better predictions for the confidence interval in particular, since it can capture 'momentum' in time series.

In [72]:
```
fig, ax = plt.subplots()
ax.axhline(0,linewidth=1, color='black', zorder=3)
ax.plot(sarimax_model.resid, linestyle='-', linewidth=1);
```



Other Models. In addition to the models above, Meta's Prophet model, an LSTM model and GARCH were tested. These notes will maintain a record to assist in future inquiry:

Prophet is a good model when there is strong seasonality or multiple seasonalities. While there is some seasonality in this data, it is week relative to the types of business problem at which Prophet excels. Prophet is also not as strong when dealing with data that has a strong autoregressive element, or data which is heteroskedastic, both of which characterize this data. Nonetheless, Prophet was fitted to see if it might provide value, particularly in light of its ability to model several seasonalities and its ability to handle holiday effects. While it did not perform terribly (RMSE = 1203), it did not perform as well as SARIMA.

Long Short-Term Memory is a recurrent neural network designed to process sequential data such as time series. Neural networks in general typically do not handle time series well, but LSTMs do because they are structured to maintain important information from earlier in the series (long memory) while taking into account brand new patterns (short-term memory), all while assessing which older information is acceptable to forget. These are powerful models that typically require a significant amount of data to function well and do not alway perform as well on stationary data or for short-term forecasting. This model performed badly on this time series. While the original dataset here is very large, it has been transformed into a time series of

5,204 days which may be too small for LSTM to extract meaningful information. Adding features or transforming the time series into an hourly or continuous time series may lead to better results.

Generalized Autoregressive Conditional Heteroskedasticity is a model to predict variance, not the likely value. This data is quite heteroskedastic, and GARCH was tested and found *very* suitable to refining future predictions. However, because this analysis focused on reducing measured errors, it was beyond scope. A more fully developed forecast model would certainly include it, particularly by fitting the residuals of the winning model which would lead to much more accuracy on the confidence intervals.

Conclusion

SARIMAX (1,1,1)x(1,1,1,7) provides the best improvement over baseline of the tested models.

Future Inquiry

1. The agencies that handle these requests may have specific patterns idiosyncratic to them. Decompose the series into different pools to model those patterns and roll them up into one composite model would be powerful.
2. Use geolocation to discover what might happen next. For instance, pest problems that originate in one area of the city may migrate to others. Street noise that spikes in one area may predict spikes elsewhere. This would uncover previously unknown associations and improve your forecasting.
3. The City has successfully pushed to broaden 311 in the past. New campaigns to add features and broaden usage, particularly in areas with lower engagement, may seem counterproductive at first, but these are needs that citizens actually have that are not being met, and it's constructive for the system and the City in the long run to have more data on what New Yorkers need.