In [1]:
```python
# ============================================================
# Notebook setup: run this before everything
# ============================================================

%load_ext autoreload
%autoreload 2

# Control figure size
figsize=(14, 4)

from sklearn.neighbors import KernelDensity
from sklearn.model_selection import GridSearchCV
from util import util
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
import os

# Load data
data_folder = os.path.join('..', 'data', 'nab')
file_name = os.path.join('realKnownCause', 'nyc_taxi.csv')
data, labels, windows = util.load_series(file_name, data_folder)

# Train and validation end
train_end = pd.to_datetime('2014-10-24 00:00:00')
val_end = pd.to_datetime('2014-12-10 00:00:00')

# Cost model parameters
c_alrm = 1 # Cost of investigating a false alarm
c_missed = 10 # Cost of missing an anomaly
c_late = 5 # Cost for late detection

# Build a cost model
cmodel = util.ADSimpleCostModel(c_alrm, c_missed, c_late)

# Separate the training data
data_tr = data[data.index < train_end]

# Apply a sliding window
wdata = util.sliding_window_1D(data, wlen=10)
```

Sequence Input in KDE

Sequence Input in KDE

Can we take sequence input into account in KDE?

There is straightforward approach, using *multivariate* KDE

- Treat each sequence as a *vector variable*

* Learn an estimator as usual

Individual *sequences* in the new dataset are treated as *independent*:

* This is due to the basic assumptions behind KDE
* In practice, for a sufficiently high window length
* ...The dependencies become *negligible*

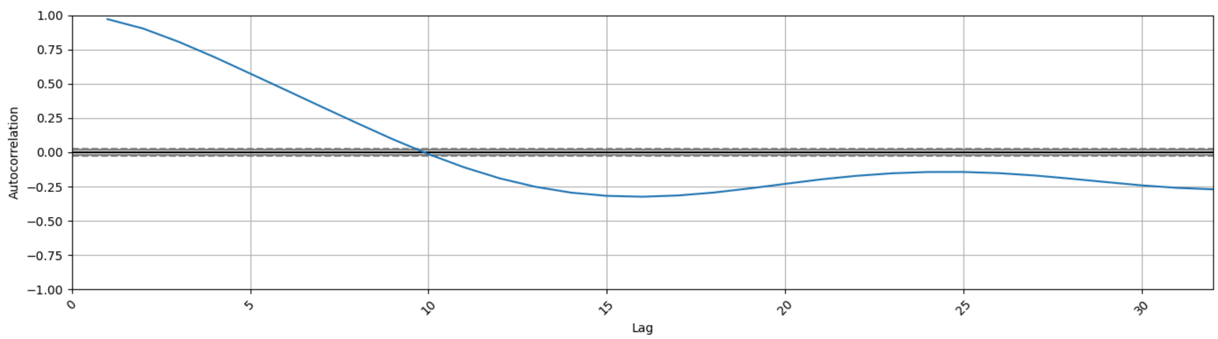
Does it sound familiar?

This is simply the Markov property!

Picking a Window Length

This suggests a way to select the window length

In [3]:
```
util.plot_autocorrelation(data, max_lag=32, figsize=figsize)
```



I.e. we end the window where the correlation becomes too low (e.g. 10 in our case)

Bandwidth Choice in Multivariate KDE

We now need to learn our multivariate KDE estimator

First, we need to choose a bandwidth

* We cannot use the (univariate) rule of thumb
* ...But we can use a more general approach

The basic intuition is that a good bandwidth

...Will make the actual data register as *more likely*

* Therefore we can pick a *validation set*
* ...And tune the bandwidth for *maximum likelihood*

To avoid overfitting, there should be *no overlap with the training data*

Bandwidth Choice in Multivariate KDE

Formally, let x be a *validation* set of m examples:

Assuming independent observations, their *estimated probability* is given by:

$$L(h, x, \bar{x}) = \prod_{i=1}^{m} \hat{f}\left(x_i, \bar{x}_i, h\right)$$

This is a called a *likelihood function*

- The main input of are the *model parameters* (h in our case)
- $\hat{f}$ is the density estimator (which outputs a probability)
- $\bar{x}$ the training set

Bandwidth Choice in Multivariate KDE

We can then choose h so as to *maximize the likelihood*

Meaning that the training problem is given by:

$$\arg\max_h \mathbb{E}_{x \sim f(x), \bar{x} \sim f(x)} \left[L(h, x, \bar{x})\right]$$

- Where $f(x)$ is the true distribution

As many training problem, it cannot be solved in an exact fashion

- Instead we will approximate $\mathbb{E}$ by sampling multiple x and $\bar{x}$
- ...I.e. multiple validation and training sets
- Then we pick the bandwidth h^* leading to the maximum average likelihood

In a pinch, we could even use a single $x, \bar{x}$ pair

Bandwidth Choice in Multivariate KDE

A simple approach consist in combining *grid search*

- It's the same approach that we used for optimizing the threshold
- scikit learn provides a convenient implementation
- ...Which resorts to cross-fold validation to define $x, \bar{x}$

First, we separate the training set as usual:

In [4]:
```
wdata_tr = wdata[wdata.index < train_end]
```

Then we specify the values we want to consider for each parameter:

```
In [5]: params = {'bandwidth': np.linspace(400, 800, 20)}
```

Training Multivariate KDE

Finally, we can run the grid search routine

```
In [6]: gs_kde = GridSearchCV(KernelDensity(kernel='gaussian'), params, cv = 5)
        gs_kde.fit(wdata_tr)
        gs_kde.best_params_
```

```
Out[6]: {'bandwidth': np.float64(568.421052631579)}
```

- `cv` is the number of folds
- After training, `GridSearchCV` acts as a proxy for the best estimator

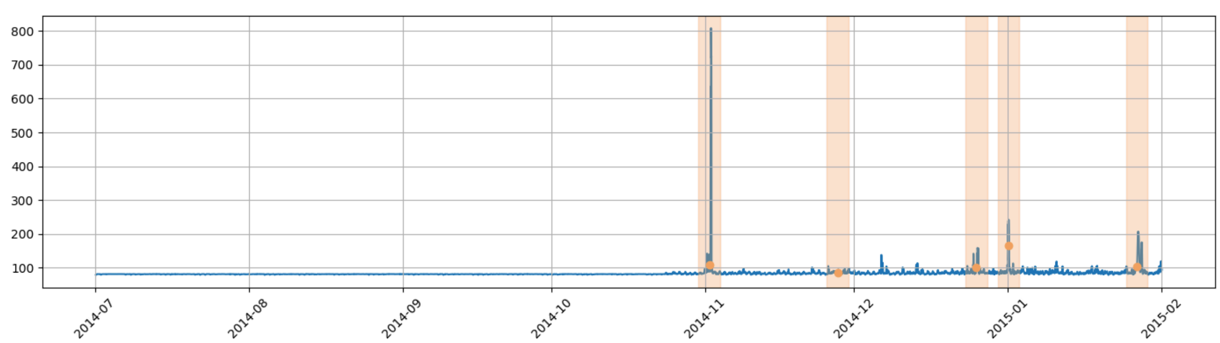
This is an *expensive operation*

- We need to test multiple bandwidth values
- For each one, we need to perform cross-validation
- ...And finally *adding dimensions makes KDE slower*

Sequences via Multivariate KDE

Now we can use the best estimator to generate the alarm signal

```
In [7]: ldens = gs_kde.score_samples(wdata)
        signal = pd.Series(index=wdata.index, data=-ldens)
        util.plot_series(signal, labels, windows, figsize=figsize)
```



- The signal seems *visibly better* than before (but a bit noisy)

Threshold Optimization

Finally, we can do threshold optimization as usual

In [8]:
```
signal_opt = signal[signal.index < val_end]
labels_opt = labels[labels < val_end]
windows_opt = windows[windows['end'] < val_end]
thr_range = np.linspace(50, 200, 100)

best_thr, best_cost = util.opt_thr(signal_opt, labels_opt, windows_opt,  cmc
print(f'Best threshold: {best_thr:.3f}, corresponding cost: {best_cost:.3f}'
```

```
Best threshold: 104.545, corresponding cost: 7.000
```

Cost on the whole dataset

In [9]:
```
ctst = cmodel.cost(signal, labels, windows, best_thr)
print(f'Cost on the whole dataset {ctst}')
```

```
Cost on the whole dataset 30
```