Volatility Prediction of NASDAQ Composite

Loading the libraries

In [1]:
```
import numpy as np
from scipy.stats import norm
import scipy.optimize as opt
import yfinance as yf
import pandas as pd
import datetime
import time
from arch import arch_model
import matplotlib.pyplot as plt
from numba import jit
from sklearn.metrics import mean_squared_error as mse
import warnings
warnings.filterwarnings('ignore')
plt.rcParams['figure.dpi'] = 300
plt.rcParams['savefig.dpi'] = 300
```

Denoting ticker of NASDAQ Composite and Identifying the start and end dates.

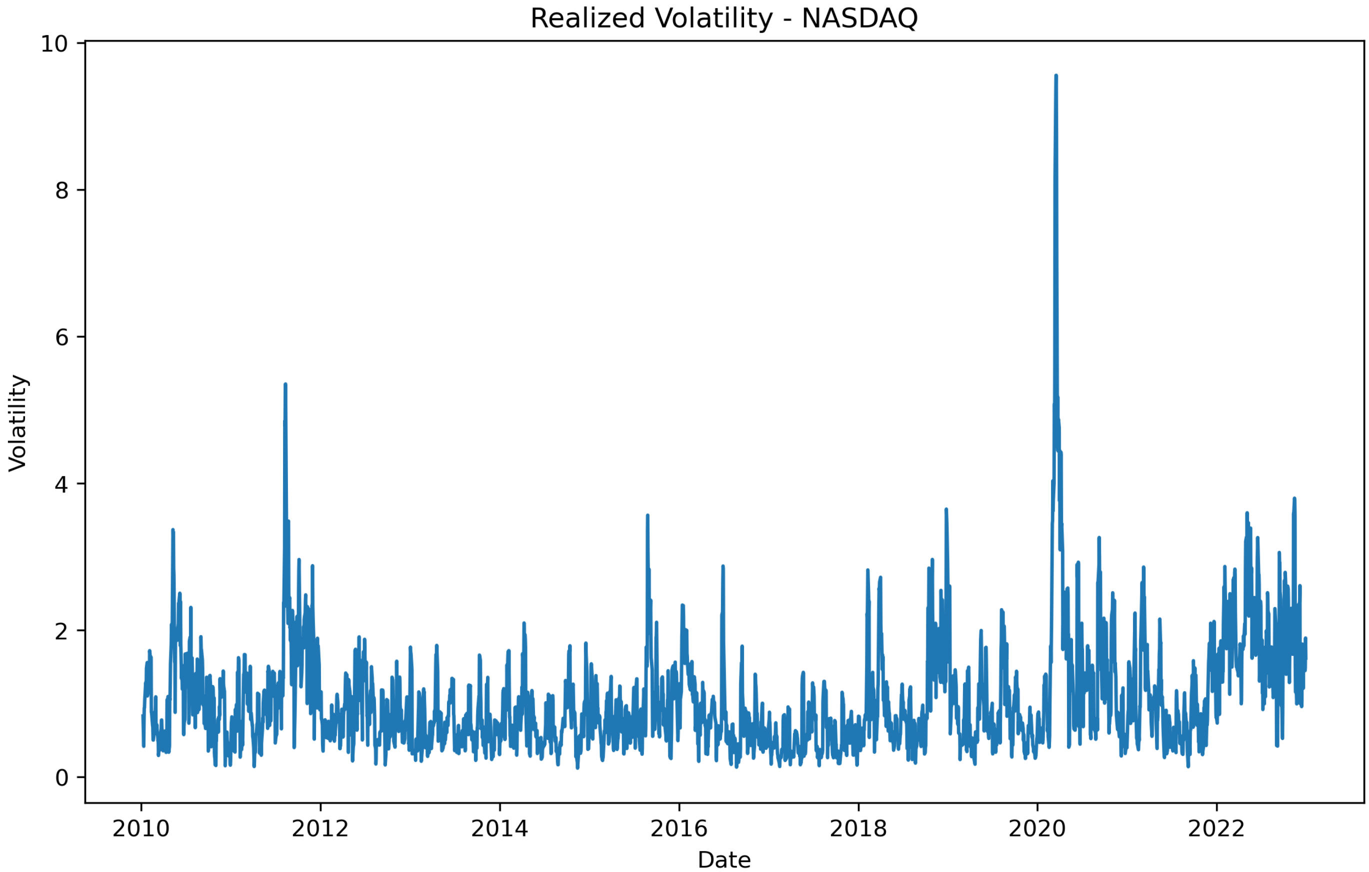
In [2]:
```
stocks = '^IXIC'
start = datetime.datetime(2010, 1, 1)
end = datetime.datetime(2023, 1, 1)
nasdaq = yf.download(stocks, start=start, end = end, interval='1d')
```

```
[*********************100%***********************]  1 of 1 completed
```

Calculating the returns of the NASDAQ Composite based on adjusted closing prices.

In [3]:
```
ret = 100 * (nasdaq.pct_change()[1:]['Adj Close'])
realized_vol = ret.rolling(5).std()
```

In [4]:
```
plt.figure(figsize=(10, 6))
plt.plot(realized_vol.index,realized_vol)
plt.title('Realized Volatility - NASDAQ')
plt.ylabel('Volatility')
plt.xlabel('Date')
plt.show()
```

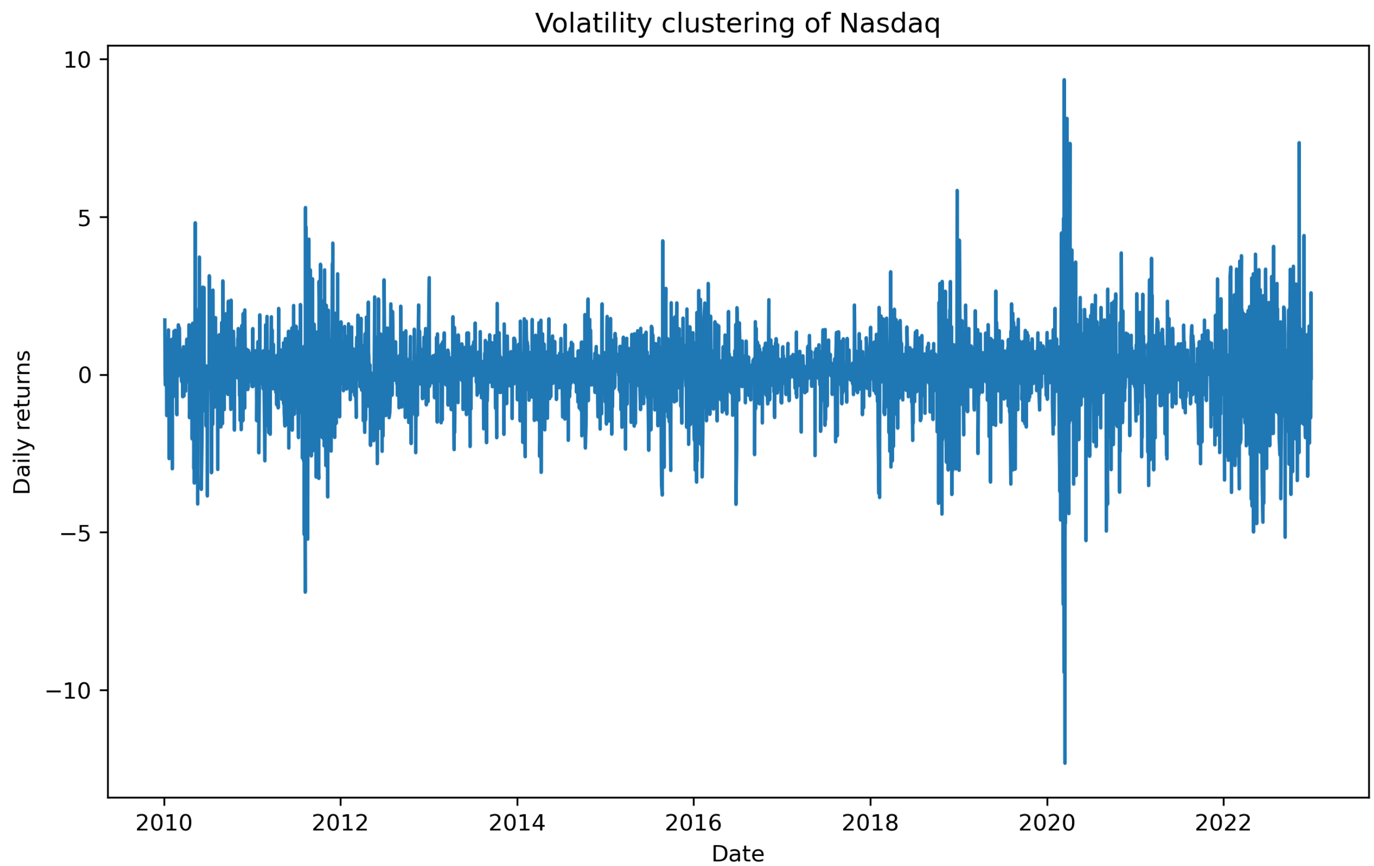


The figure above shows the realized volatility of NASDAQ over the period of 2010–2022.The most striking observation is the spikes around the COVID-19 pandemic.

Return dataframe into a numpy representation.

In [5]:
```
retv = ret.values
```

In [6]:
```
plt.figure(figsize=(10, 6))
plt.plot(nasdaq.index[1:], ret)
plt.title('Volatility clustering of Nasdaq')
plt.ylabel('Daily returns')
plt.xlabel('Date')
plt.show()
```



ARCH Model

Defining the split location and assigning the split data to split variable.

In [7]:
```
n = 252
split_date = ret.iloc[-n:].index
```

Calculating variance and the kurtosis of the NASDAQ and Identifying the initial value for slope coefficient alpha and the constant term omega.

In [8]:
```
sgm2 = ret.var()
K = ret.kurtosis()
alpha = (-3.0 * sgm2 + np.sqrt(9.0 * sgm2 ** 2 - 12.0 *
                             (3.0 * sgm2 - K) * K)) / (6 * K)
omega = (1 - alpha) * sgm2
initial_parameters = [alpha, omega]
omega, alpha
```

Out[8]:
```
(1.313007878710188, 0.22064177385125494)
```

- Using parallel processing to decrease the processing time.
- Taking absolute values and assigning the initial values into related variables
- Identifying the initial values of volatility
- Iterating the variance of S&P 500
- Calculating the log-likelihood

In [9]:
```
@jit(nopython=True, parallel=True)
def arch_likelihood(initial_parameters, retv):
    omega = abs(initial_parameters[0])
    alpha = abs(initial_parameters[1])
```

```python
    T = len(retv)
    logliks = 0
    sigma2 = np.zeros(T)
    sigma2[0] = np.var(retv)
    for t in range(1, T):
        sigma2[t] = omega + alpha * (retv[t - 1]) ** 2
    logliks = np.sum(0.5 * (np.log(sigma2)+retv ** 2 / sigma2))
    return logliks
```

In [10]:
```python
logliks = arch_likelihood(initial_parameters, retv)
logliks
```

Out[10]:
```
3471.11210291535
```

- Minimizing the log-likelihood function
- Creating a variable params for optimized parameters

In [11]:
```python
def opt_params(x0, retv):
    opt_result = opt.minimize(arch_likelihood, x0=x0, args = (retv),
                              method='Nelder-Mead',
                              options={'maxiter': 5000})
    params = opt_result.x
    print('\nResults of Nelder-Mead minimization\n{}\n{}'
          .format(''.join(['-'] * 28), opt_result))
    print('\nResulting params = {}'.format(params))
    return params
```

In [12]:
```python
params = opt_params(initial_parameters, retv)
```

```
Results of Nelder-Mead minimization
----------------------------
 final_simplex: (array([[1.14008269, 0.32016609],
       [1.14002728, 0.32021089],
       [1.14009332, 0.32020735]]), array([2284.68173741, 2284.68173758, 2284.68173832]))
           fun: 2284.6817374064294
       message: 'Optimization terminated successfully.'
          nfev: 87
           nit: 45
        status: 0
       success: True
             x: array([1.14008269, 0.32016609])

Resulting params = [1.14008269 0.32016609]
```

In [13]:
```python
def arch_apply(ret):
        omega = params[0]
        alpha = params[1]
        T = len(ret)
        sigma2_arch = np.zeros(T + 1)
        sigma2_arch[0] = np.var(ret)
        for t in range(1, T):
            sigma2_arch[t] = omega + alpha * ret[t - 1] ** 2
        return sigma2_arch
```

In [14]:
```python
sigma2_arch = arch_apply(ret)
```

- Well, we modeled volatility via ARCH using our own optimization method and ARCH equation. But how about comparing it with the built-in Python code? This built-in code can be imported from arch library and is extremely easy to apply. The result of the built-in function follows; it turns out that these two results are very similar to each other.

In [15]:
```python
arch = arch_model(ret, mean='zero', vol='ARCH', p=1).fit(disp='off')
print(arch.summary())
```

```
                        Zero Mean - ARCH Model Results                        
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
Vol Model:                       ARCH   Log-Likelihood:               -5291.49
Distribution:                  Normal   AIC:                           10587.0
Method:            Maximum Likelihood   BIC:                           10599.2
                                        No. Observations:                 3272
Date:                Fri, Jan 27 2023   Df Residuals:                     3272
Time:                        16:45:57   Df Model:                            0
                            Volatility Model                            
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
omega          1.1403  6.651e-02     17.145  6.907e-66 [  1.010,  1.271]
alpha[1]       0.3204  4.949e-02      6.474  9.559e-11 [  0.223,  0.417]
========================================================================

Covariance estimator: robust
```

Iterating ARCH model from 1 to 5 lags using Bayesian Information Criteria.

- Iterating ARCH parameter p over specified interval
- Running ARCH model with different p values
- Finding the minimum BIC score to select the best model
- Running ARCH model with the best p value
- Forecasting the volatility based on the optimized ARCH model

In [16]:
```python
bic_arch = []

for p in range(1, 5):
        arch = arch_model(ret, mean='zero', vol='ARCH', p=p)\
                .fit(disp='off')
        bic_arch.append(arch.bic)
        if arch.bic == np.min(bic_arch):
            best_param = p
arch = arch_model(ret, mean='zero', vol='ARCH', p=best_param)\
        .fit(disp='off')
print(arch.summary())
forecast = arch.forecast(start=split_date[0])
forecast_arch = forecast
```

```
                        Zero Mean - ARCH Model Results                        
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
Vol Model:                       ARCH   Log-Likelihood:               -4982.84
Distribution:                  Normal   AIC:                           9975.68
Method:            Maximum Likelihood   BIC:                           10006.1
                                        No. Observations:                 3272
Date:                Fri, Jan 27 2023   Df Residuals:                     3272
Time:                        16:46:01   Df Model:                            0
                              Volatility Model                              
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.4581  3.618e-02     12.663  9.445e-37    [  0.387,  0.529]
```

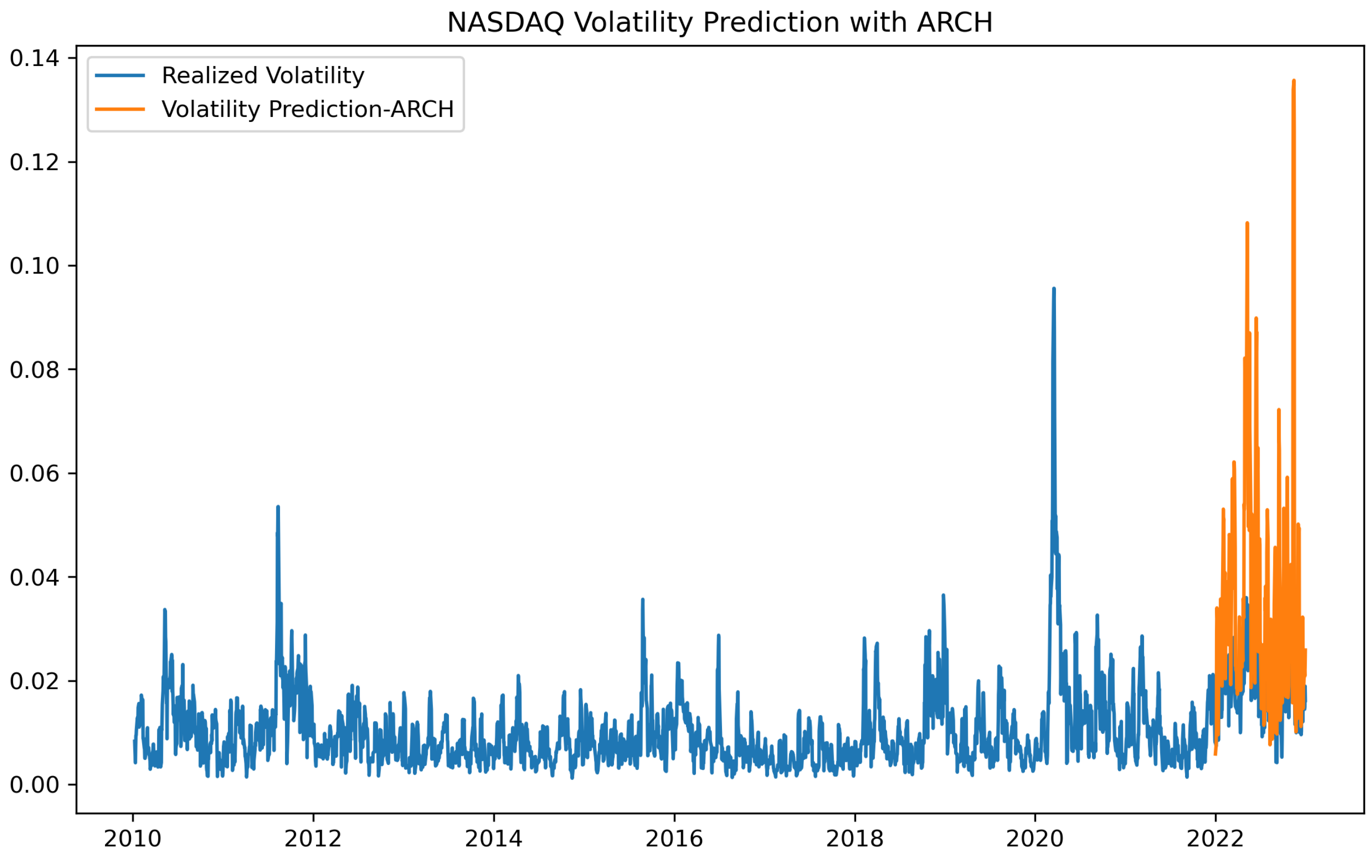
```
alpha[1]        0.1555   3.721e-02      4.178  2.942e-05 [8.253e-02,  0.228]
alpha[2]        0.2052   3.185e-02      6.442  1.181e-10   [  0.143,  0.268]
alpha[3]        0.2028   3.373e-02      6.014  1.810e-09   [  0.137,  0.269]
alpha[4]        0.2001   3.523e-02      5.679  1.352e-08   [  0.131,  0.269]
=============================================================================

Covariance estimator: robust
```

Calculating the root mean square error (RMSE) score.

In [17]:
```
rmse_arch = np.sqrt(mse(realized_vol[-n:] / 100,
                        np.sqrt(forecast_arch\
                        .variance.iloc[-len(split_date):]
                        / 100)))
print('The RMSE value of ARCH model is {:.4f}'.format(rmse_arch))
```

```
The RMSE value of ARCH model is 0.1679
```

In [18]:
```
plt.figure(figsize=(10, 6))
plt.plot(realized_vol / 100, label='Realized Volatility')
plt.plot(forecast_arch.variance.iloc[-len(split_date):] / 100,
         label='Volatility Prediction-ARCH')
plt.title('NASDAQ Volatility Prediction with ARCH', fontsize=12)
plt.legend()
plt.show()
```



GARCH Model (Extension of the ARCH model)

We'll attack the same steps similarly to ARCH model

In [19]:
```
a0 = 0.0001
sgm2 = ret.var()
K = ret.kurtosis()
h = 1 - alpha / sgm2
alpha = np.sqrt(K * (1 - h ** 2) / (2.0 * (K + 3)))
```

```
    beta = np.abs(h - omega)
    omega = (1 - omega) * sgm2
    initial_parameters = np.array([omega, alpha, beta])
    print('Initial parameters for omega, alpha, and beta are \n{}\n{}\n{}'
          .format(omega, alpha, beta))
```

```
Initial parameters for omega, alpha, and beta are
-0.5273336407517452
0.294072939035948
0.4439735676990063
```

In [20]:
```
retv = ret.values
```

In [21]:
```
@jit(nopython=True, parallel=True)
def garch_likelihood(initial_parameters, retv):
    omega = initial_parameters[0]
    alpha = initial_parameters[1]
    beta = initial_parameters[2]
    T =  len(retv)
    logliks = 0
    sigma2 = np.zeros(T)
    sigma2[0] = np.var(retv)
    for t in range(1, T):
        sigma2[t] = omega + alpha * (retv[t - 1]) ** 2 + beta * sigma2[t-1]
    logliks = np.sum(0.5 * (np.log(sigma2) + retv ** 2 / sigma2))
    return logliks
```

In [22]:
```
logliks = garch_likelihood(initial_parameters, retv)
print('The Log likelihood  is {:.4f}'.format(logliks))
```

```
The Log likelihood  is nan
```

In [23]:
```
def garch_constraint(initial_parameters):
    alpha = initial_parameters[0]
    gamma = initial_parameters[1]
    beta = initial_parameters[2]
    return np.array([1 - alpha - beta])
```

In [24]:
```
bounds = [(0.0, 1.0), (0.0, 1.0), (0.0, 1.0)]
```

In [25]:
```
def opt_paramsG(initial_parameters, retv):
    opt_result = opt.minimize(garch_likelihood,
                              x0=initial_parameters,
                              constraints=np.array([1 - alpha - beta]),
                              bounds=bounds, args = (retv),
                              method='Nelder-Mead',
                              options={'maxiter': 5000})
    params = opt_result.x
    print('\nResults of Nelder-Mead minimization\n{}\n{}'\
          .format('-' * 35, opt_result))
    print('-' * 35)
    print('\nResulting parameters = {}'.format(params))
    return params
```

In [26]:
```
params = opt_paramsG(initial_parameters, retv)
```

```
Results of Nelder-Mead minimization
-----------------------------------
 final_simplex: (array([[0.04265915, 0.13115316, 0.84223821],
       [0.0426082 , 0.13111178, 0.84231429],
       [0.04267095, 0.13120978, 0.84217189],
       [0.04267015, 0.13115293, 0.84220692]]), array([1891.77068559, 1891.77069313, 189
1.77069837, 1891.77070055]))
           fun: 1891.77068558753
```

```
        message: 'Optimization terminated successfully.'
           nfev: 164
            nit: 92
         status: 0
        success: True
              x: array([0.04265915, 0.13115316, 0.84223821])
-----------------------------------

Resulting parameters = [0.04265915 0.13115316 0.84223821]
```

In [27]:
```
def garch_apply(ret):
        omega = params[0]
        alpha = params[1]
        beta = params[2]
        T = len(ret)
        sigma2 = np.zeros(T + 1)
        sigma2[0] = np.var(ret)
        for t in range(1, T):
            sigma2[t] = omega + alpha * ret[t - 1] ** 2 +\
                        beta * sigma2[t-1]
        return sigma2
```

The built-in Python GARCH function

In [28]:
```
garch = arch_model(ret, mean='zero', vol='GARCH', p=1, o=0, q=1)\
        .fit(disp='off')
print(garch.summary())
```

```
                       Zero Mean - GARCH Model Results
==============================================================================
Dep. Variable:                Adj Close   R-squared:                       0.000
Mean Model:                   Zero Mean   Adj. R-squared:                  0.000
Vol Model:                        GARCH   Log-Likelihood:               -4898.43
Distribution:                    Normal   AIC:                           9802.87
Method:          Maximum Likelihood   BIC:                           9821.15
                                          No. Observations:                 3272
Date:                  Fri, Jan 27 2023   Df Residuals:                     3272
Time:                          16:46:38   Df Model:                            0
                              Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0427  9.499e-03      4.493  7.028e-06 [2.406e-02,6.130e-02]
alpha[1]       0.1312  1.710e-02      7.672  1.688e-14   [9.771e-02,  0.165]
beta[1]        0.8422  1.902e-02     44.273      0.000     [  0.805,  0.879]
============================================================================

Covariance estimator: robust
```

Iterating GARCH model from 1 to 5 lags using Bayesian Information Criteria.

In [29]:
```
bic_garch = []

for p in range(1, 5):
    for q in range(1, 5):
        garch = arch_model(ret, mean='zero',vol='GARCH', p=p, o=0, q=q)\
                .fit(disp='off')
        bic_garch.append(garch.bic)
        if garch.bic == np.min(bic_garch):
            best_param = p, q
garch = arch_model(ret, mean='zero', vol='GARCH',
                   p=best_param[0], o=0, q=best_param[1])\
        .fit(disp='off')
```

```
print(garch.summary())
forecast = garch.forecast(start=split_date[0])
forecast_garch = forecast
```

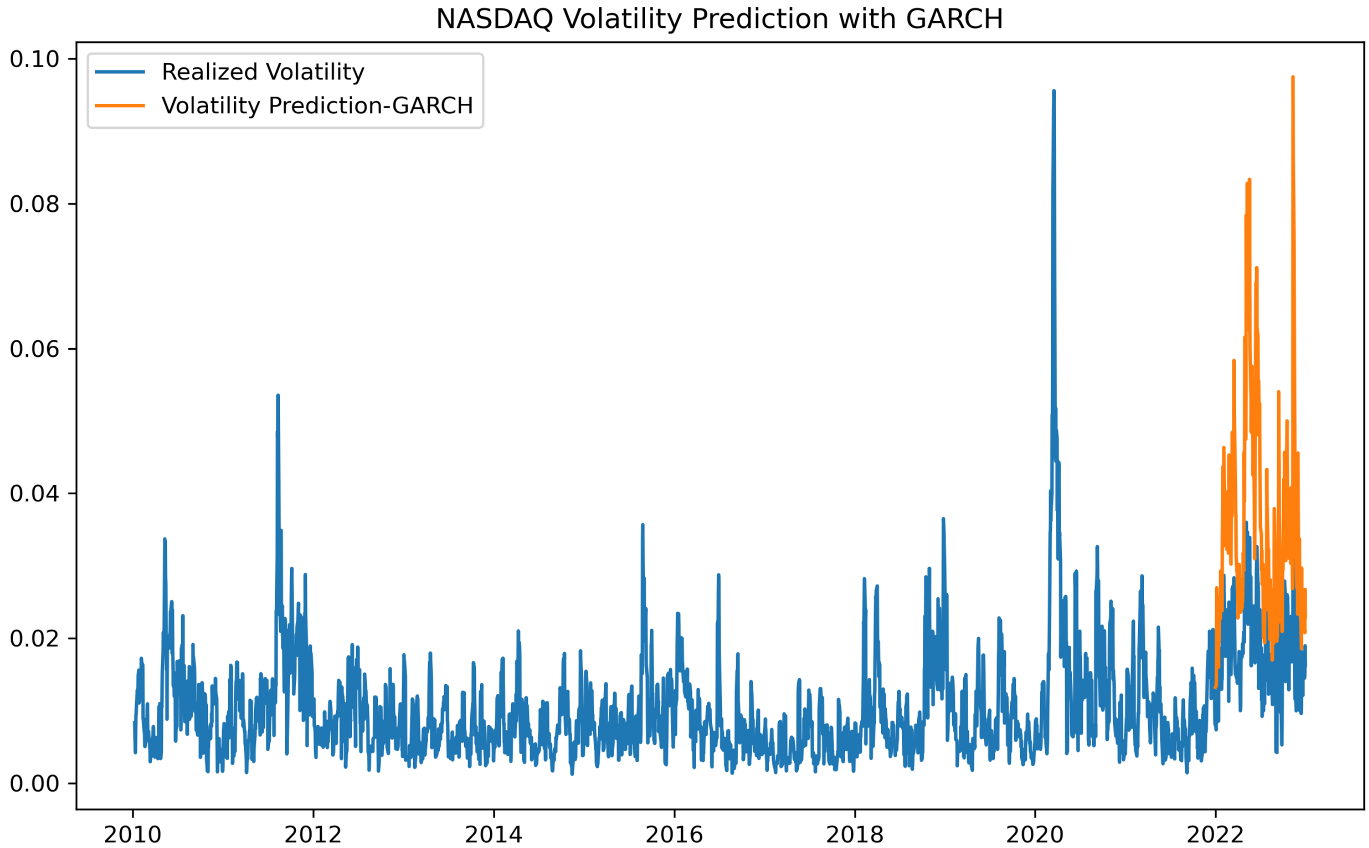
```
                       Zero Mean - GARCH Model Results                        
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
Vol Model:                      GARCH   Log-Likelihood:               -4898.43
Distribution:                  Normal   AIC:                           9802.87
Method:            Maximum Likelihood   BIC:                           9821.15
                                        No. Observations:                 3272
Date:                Fri, Jan 27 2023   Df Residuals:                     3272
Time:                        16:46:43   Df Model:                            0
                              Volatility Model                              
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0427  9.499e-03      4.493  7.028e-06 [2.406e-02,6.130e-02]
alpha[1]       0.1312  1.710e-02      7.672  1.688e-14   [9.771e-02,  0.165]
beta[1]        0.8422  1.902e-02     44.273      0.000     [  0.805,  0.879]
============================================================================

Covariance estimator: robust
```

Calculating the root mean square error (RMSE) score.

In [30]:
```
rmse_garch = np.sqrt(mse(realized_vol[-n:] / 100,
                         np.sqrt(forecast_garch\
                         .variance.iloc[-len(split_date):]
                         / 100)))
print('The RMSE value of GARCH model is {:.4f}'.format(rmse_garch))
```

```
The RMSE value of GARCH model is 0.1702
```

In [31]:
```
plt.figure(figsize=(10,6))
plt.plot(realized_vol / 100, label='Realized Volatility')
plt.plot(forecast_garch.variance.iloc[-len(split_date):] / 100,
         label='Volatility Prediction-GARCH')
plt.title('NASDAQ Volatility Prediction with GARCH', fontsize=12)
plt.legend()
plt.show()
```



GJR-GARCH Model

Iterating GJR-GARCH model from 1 to 5 lags using Bayesian Information Criteria.

In [32]:
```
bic_gjr_garch = []

for p in range(1, 5):
    for q in range(1, 5):
        gjrgarch = arch_model(ret, mean='zero', p=p, o=1, q=q)\
                   .fit(disp='off')
        bic_gjr_garch.append(gjrgarch.bic)
        if gjrgarch.bic == np.min(bic_gjr_garch):
            best_param = p, q
gjrgarch = arch_model(ret,mean='zero', p=best_param[0], o=1,
                      q=best_param[1]).fit(disp='off')
print(gjrgarch.summary())
forecast = gjrgarch.forecast(start=split_date[0])
forecast_gjrgarch = forecast
```

```
                     Zero Mean - GJR-GARCH Model Results
==============================================================================
Dep. Variable:                 Adj Close   R-squared:                       0.000
Mean Model:                    Zero Mean   Adj. R-squared:                  0.000
Vol Model:                     GJR-GARCH   Log-Likelihood:               -4832.54
Distribution:                     Normal   AIC:                           9673.08
Method:               Maximum Likelihood   BIC:                           9697.45
                                           No. Observations:                 3272
Date:                   Fri, Jan 27 2023   Df Residuals:                     3272
Time:                           16:46:59   Df Model:                            0
                              Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
```

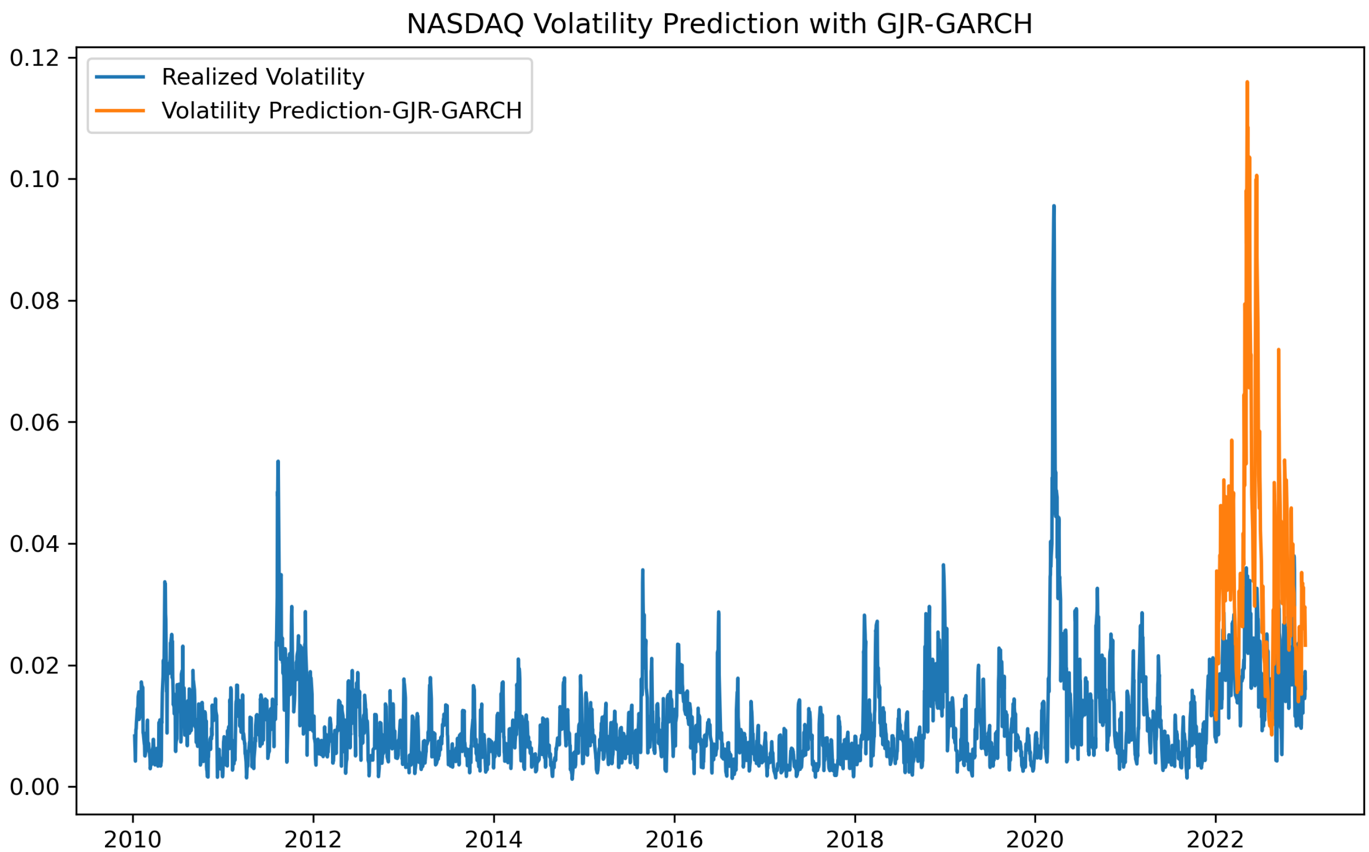
```
omega            0.0459  1.118e-02      4.101  4.111e-05  [2.395e-02,6.779e-02]
alpha[1]     5.0681e-03  1.883e-02      0.269      0.788 [-3.184e-02,4.198e-02]
gamma[1]         0.2027  4.519e-02      4.485  7.292e-06      [  0.114,  0.291]
beta[1]          0.8643  2.475e-02     34.918 3.973e-267      [  0.816,  0.913]
=============================================================================

Covariance estimator: robust
```

Calculating the root mean square error (RMSE) score.

In [33]:
```
rmse_gjr_garch = np.sqrt(mse(realized_vol[-n:] / 100,
                             np.sqrt(forecast_gjrgarch\
                             .variance.iloc[-len(split_date):]
                             / 100)))
print('The RMSE value of GJR-GARCH models is {:.4f}'
      .format(rmse_gjr_garch))
```

```
The RMSE value of GJR-GARCH models is 0.1732
```

In [34]:
```
plt.figure(figsize=(10, 6))
plt.plot(realized_vol / 100, label='Realized Volatility')
plt.plot(forecast_gjrgarch.variance.iloc[-len(split_date):] / 100,
         label='Volatility Prediction-GJR-GARCH')
plt.title('NASDAQ Volatility Prediction with GJR-GARCH', fontsize=12)
plt.legend()
plt.show()
```



EGARCH Model

Iterating EGARCH model from 1 to 5 lags using Bayesian Information Criteria.

In [35]:
```
bic_egarch = []
```

```
for p in range(1, 5):
    for q in range(1, 5):
        egarch = arch_model(ret, mean='zero', vol='EGARCH', p=p, q=q)\
                 .fit(disp='off')
        bic_egarch.append(egarch.bic)
        if egarch.bic == np.min(bic_egarch):
            best_param = p, q
egarch = arch_model(ret, mean='zero', vol='EGARCH',
                    p=best_param[0], q=best_param[1])\
         .fit(disp='off')
print(egarch.summary())
forecast = egarch.forecast(start=split_date[0])
forecast_egarch = forecast
```

```
                       Zero Mean - EGARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
Vol Model:                     EGARCH   Log-Likelihood:               -4910.47
Distribution:                  Normal   AIC:                           9826.93
Method:            Maximum Likelihood   BIC:                           9845.21
                                        No. Observations:                 3272
Date:                Fri, Jan 27 2023   Df Residuals:                     3272
Time:                        16:47:11   Df Model:                            0
                              Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0181  5.323e-03      3.407  6.570e-04 [7.702e-03,2.857e-02]
alpha[1]       0.2624  2.779e-02      9.444  3.575e-21     [  0.208,  0.317]
beta[1]        0.9623  8.155e-03    118.008      0.000     [  0.946,  0.978]
============================================================================

Covariance estimator: robust
```

Calculating the root mean square error (RMSE) score.

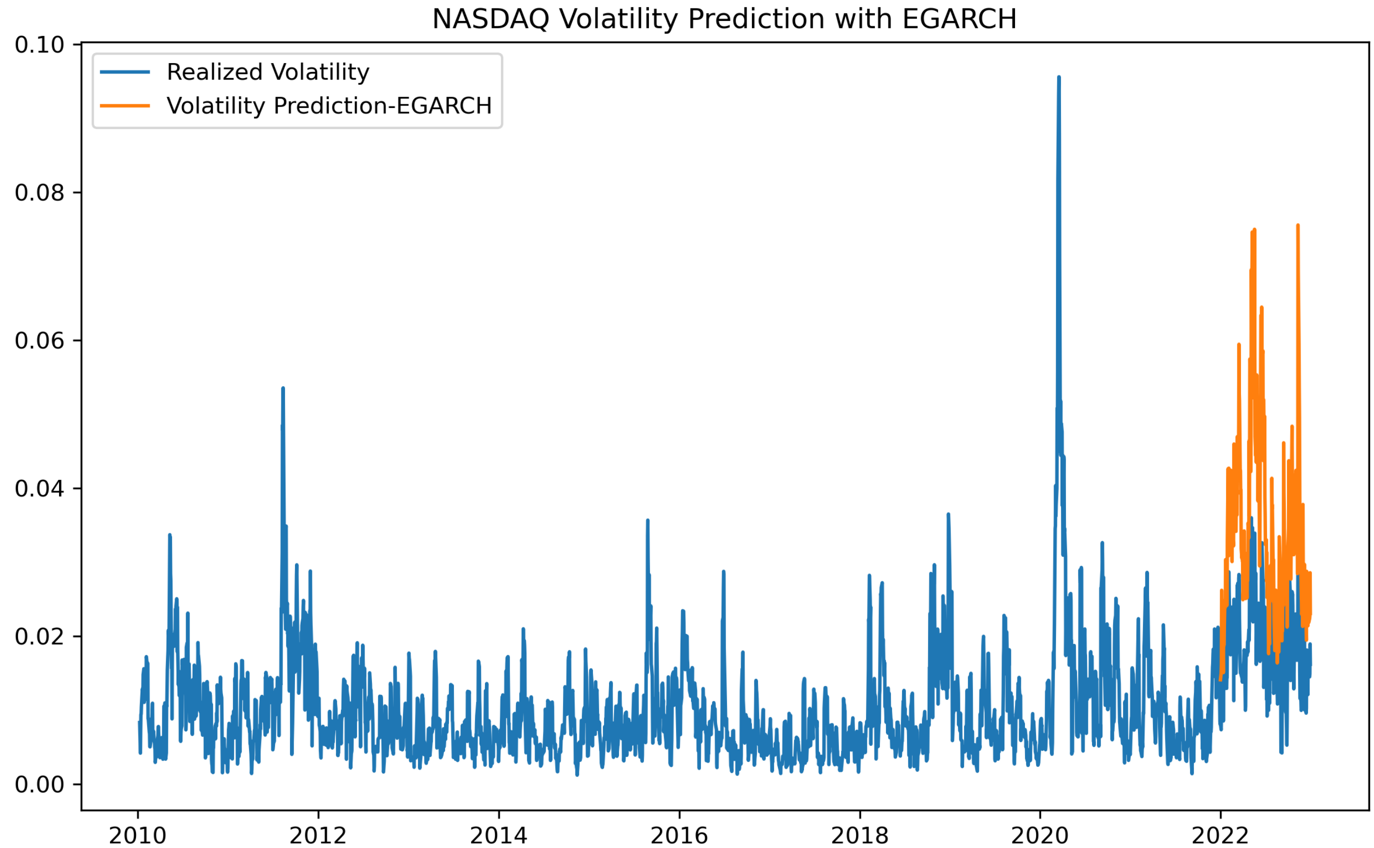
In [36]:

```
rmse_egarch = np.sqrt(mse(realized_vol[-n:] / 100,
                          np.sqrt(forecast_egarch.variance\
                          .iloc[-len(split_date):] / 100)))
print('The RMSE value of EGARCH models is {:.4f}'.format(rmse_egarch))
```

```
The RMSE value of EGARCH models is 0.1660
```

In [37]:

```
plt.figure(figsize=(10, 6))
plt.plot(realized_vol / 100, label='Realized Volatility')
plt.plot(forecast_egarch.variance.iloc[-len(split_date):] / 100,
         label='Volatility Prediction-EGARCH')
plt.title('NASDAQ Volatility Prediction with EGARCH', fontsize=12)
plt.legend()
plt.show()
```



Conclusion

After comparing the RMSE values, we can say that in this case the best performing model is EGARCH and the worst is GJR-GARCH. But there are no big differences in the performance of the models we have used here.

SVR-GARCH (Support Vector Machine) Model

In [38]:
```
from sklearn.svm import SVR
from scipy.stats import uniform as sp_rand
from sklearn.model_selection import RandomizedSearchCV
```

- Computing realized volatility and assigning a new variable to it.

In [39]:
```
realized_vol = ret.rolling(5).std()
realized_vol = pd.DataFrame(realized_vol)
realized_vol.reset_index(drop=True, inplace=True)
```

In [40]:
```
returns_svm = ret ** 2
returns_svm = returns_svm.reset_index()
del returns_svm['Date']
```

In [41]:
```
X = pd.concat([realized_vol, returns_svm], axis=1, ignore_index=True)
X = X[4:].copy()
X = X.reset_index()
X.drop('index', axis=1, inplace=True)
```

In [42]:
```
realized_vol = realized_vol.dropna().reset_index()
realized_vol.drop('index', axis=1, inplace=True)
```

- Creating new variables for each SVR kernel.

In [43]:
```
svr_poly = SVR(kernel='poly', degree=2)
svr_lin = SVR(kernel='linear')
svr_rbf = SVR(kernel='rbf')
```

SVR-GARCH-Linear

- Identifying the hyperparameter space for tuning
- Applying hyperparameter tuning with RandomizedSearchCV
- Fitting SVR-GARCH with linear kernel to data
- Predicting the volatilities based on the last 252 observations and storing them in the predict_svr_lin

In [45]:
```
para_grid = {'gamma': sp_rand(),
             'C': sp_rand(),
             'epsilon': sp_rand()}
clf = RandomizedSearchCV(svr_lin, para_grid)
clf.fit(X.iloc[:-n].values,
        realized_vol.iloc[1:-(n-1)].values.reshape(-1,))
predict_svr_lin = clf.predict(X.iloc[-n:])
```

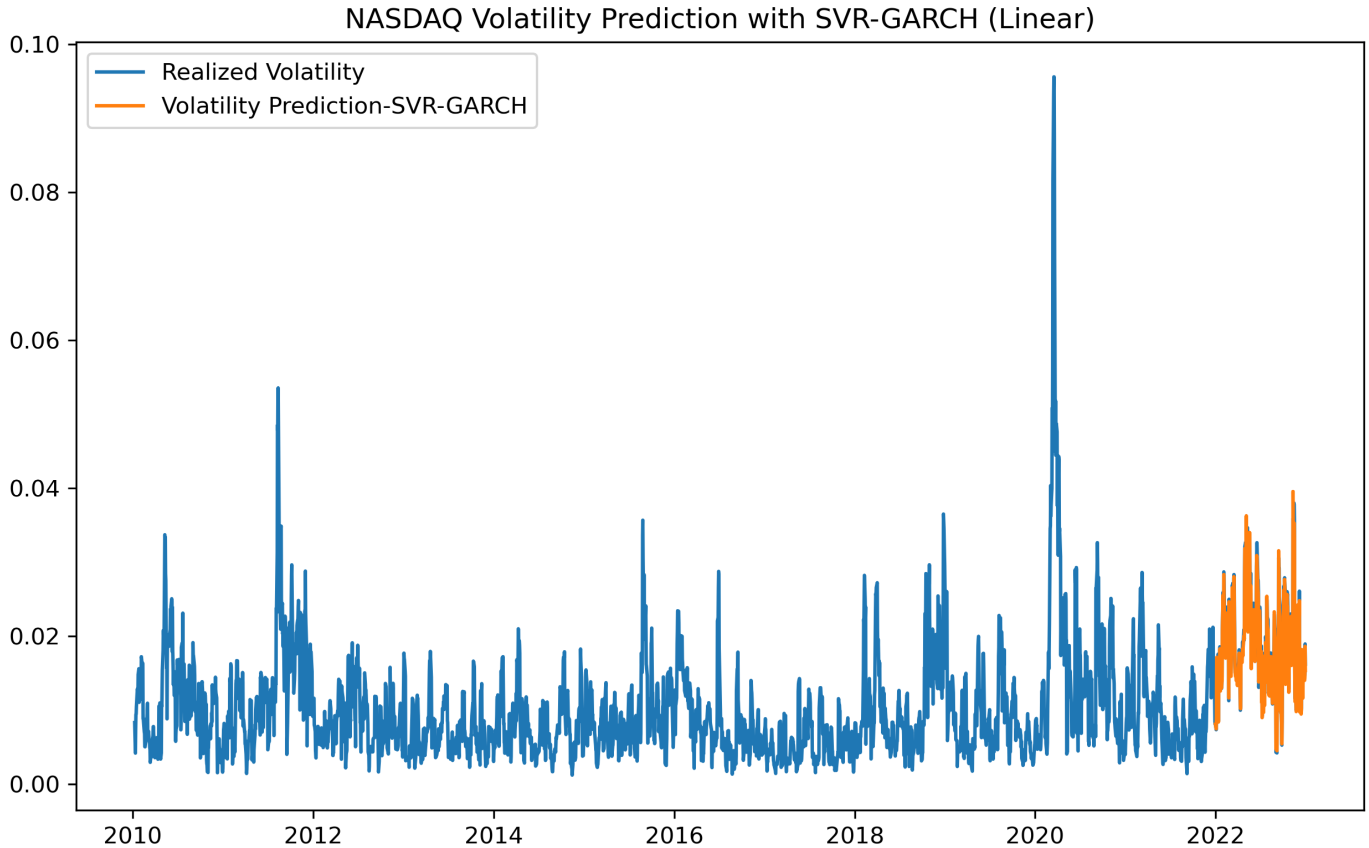
In [46]:
```
predict_svr_lin = pd.DataFrame(predict_svr_lin)
predict_svr_lin.index = ret.iloc[-n:].index
```

In [47]:
```
rmse_svr = np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                       predict_svr_lin / 100))
print('The RMSE value of SVR with Linear Kernel is {:.6f}'
      .format(rmse_svr))
```

```
The RMSE value of SVR with Linear Kernel is 0.000954
```

In [48]:
```
realized_vol.index = ret.iloc[4:].index
```

In [49]:
```
plt.figure(figsize=(10, 6))
plt.plot(realized_vol / 100, label='Realized Volatility')
plt.plot(predict_svr_lin / 100, label='Volatility Prediction-SVR-GARCH')
plt.title('NASDAQ Volatility Prediction with SVR-GARCH (Linear)', fontsize=12)
plt.legend()
plt.show()
```



SVR-GARCH RBF (Radial Basis Function kernel)

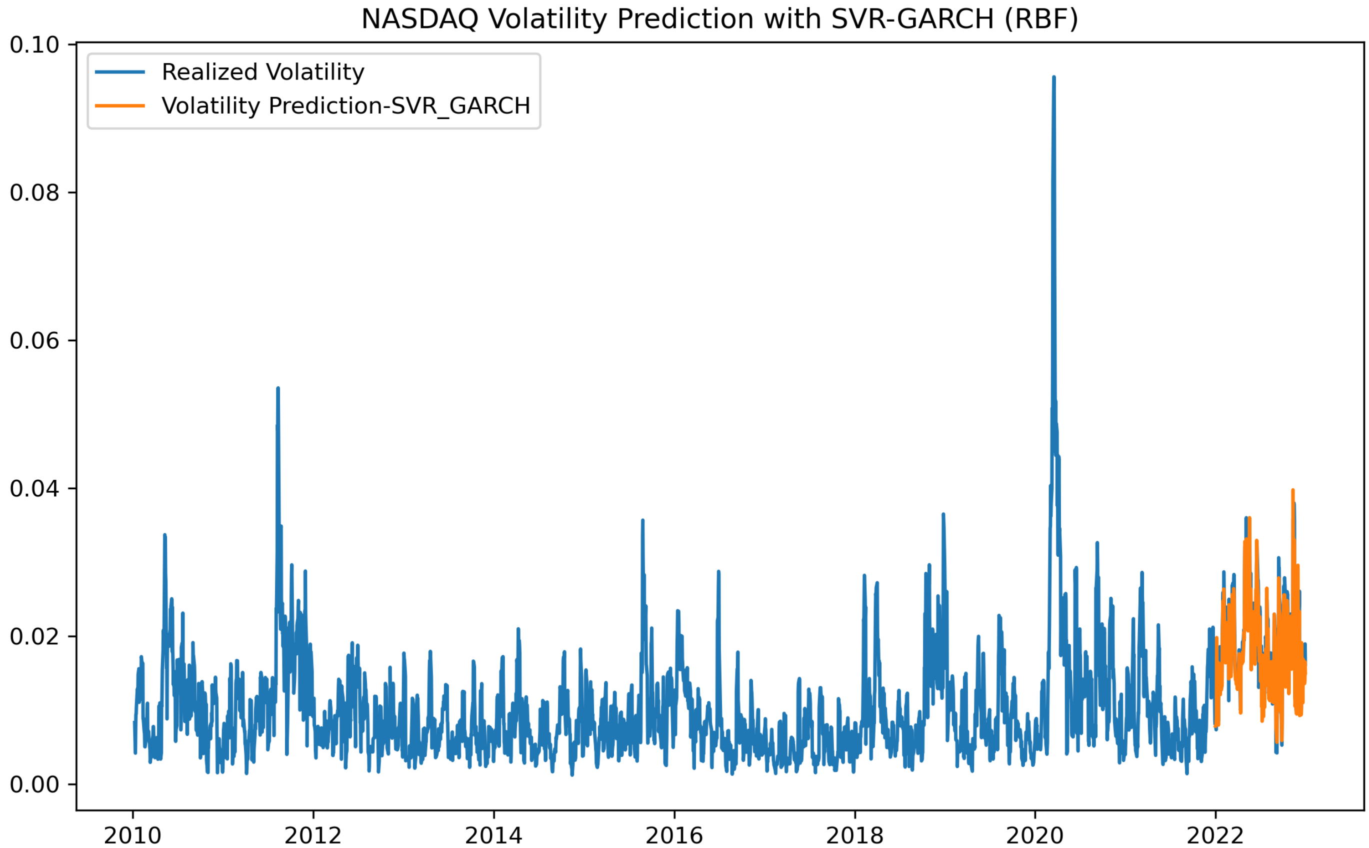
In [50]:
```
para_grid ={'gamma': sp_rand(),
            'C': sp_rand(),
            'epsilon': sp_rand()}
clf = RandomizedSearchCV(svr_rbf, para_grid)
clf.fit(X.iloc[:-n].values,
        realized_vol.iloc[1:-(n-1)].values.reshape(-1,))
predict_svr_rbf = clf.predict(X.iloc[-n:])
```

In [51]:
```
predict_svr_rbf = pd.DataFrame(predict_svr_rbf)
predict_svr_rbf.index = ret.iloc[-n:].index
```

In [52]:
```
rmse_svr_rbf = np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                           predict_svr_rbf / 100))
print('The RMSE value of SVR with RBF Kernel is  {:.6f}'
      .format(rmse_svr_rbf))
```

```
The RMSE value of SVR with RBF Kernel is  0.001675
```

In [53]:
```
plt.figure(figsize=(10, 6))
plt.plot(realized_vol / 100, label='Realized Volatility')
plt.plot(predict_svr_rbf / 100, label='Volatility Prediction-SVR_GARCH')
plt.title('NASDAQ Volatility Prediction with SVR-GARCH (RBF)', fontsize=12)
plt.legend()
plt.show()
```



SVR-GARCH Polynomial (The lightest one)

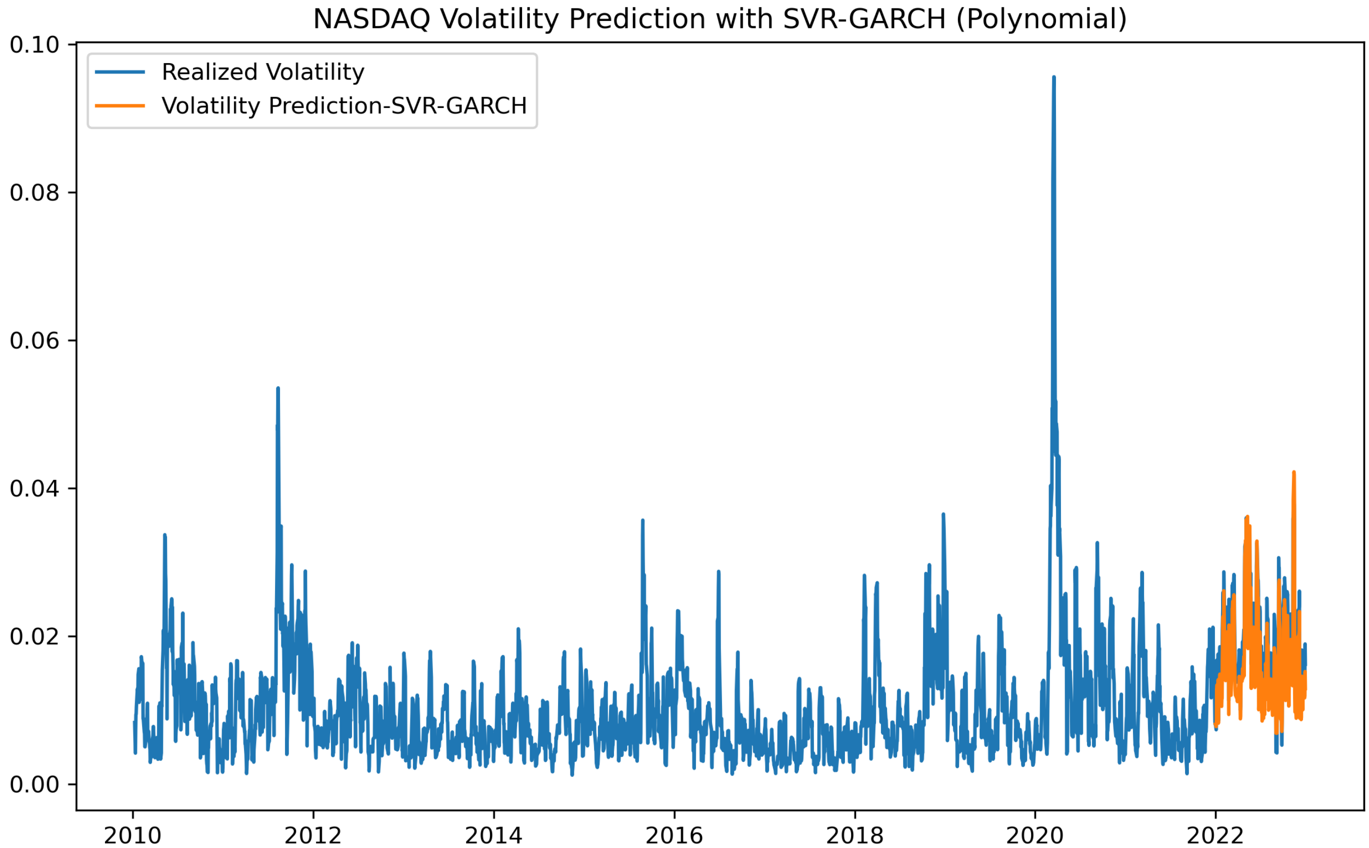
In [54]:
```
para_grid = {'gamma': sp_rand(),
             'C': sp_rand(),
             'epsilon': sp_rand()}
clf = RandomizedSearchCV(svr_poly, para_grid)
clf.fit(X.iloc[:-n].values,
        realized_vol.iloc[1:-(n-1)].values.reshape(-1,))
predict_svr_poly = clf.predict(X.iloc[-n:])
```

In [55]:
```
predict_svr_poly = pd.DataFrame(predict_svr_poly)
predict_svr_poly.index = ret.iloc[-n:].index
```

In [56]:
```
rmse_svr_poly = np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                            predict_svr_poly / 100))
print('The RMSE value of SVR with Polynomial Kernel is {:.6f}'\
      .format(rmse_svr_poly))
```

```
The RMSE value of SVR with Polynomial Kernel is 0.003156
```

In [58]:
```
plt.figure(figsize=(10, 6))
plt.plot(realized_vol/100, label='Realized Volatility')
plt.plot(predict_svr_poly/100, label='Volatility Prediction-SVR-GARCH')
plt.title('NASDAQ Volatility Prediction with SVR-GARCH (Polynomial)',
          fontsize=12)
plt.legend()
plt.show()
```



NN-GARCH (Neural Networks)

- Importing the MLPRegressor module

In [59]:
```
from sklearn.neural_network import MLPRegressor
```

- Configuring the NN model with three hidden layers and varying neuron numbers.
- Fitting the NN model to the training data.
- Predicting the volatilities based on the last 252 observations and storing them in a variable.

In [61]:
```
NN_vol = MLPRegressor(learning_rate_init=0.001, random_state=1)
para_grid_NN = {'hidden_layer_sizes': [(100, 50), (50, 50), (10, 100)],
               'max_iter': [500, 1000],
               'alpha': [0.00005, 0.0005 ]}
clf = RandomizedSearchCV(NN_vol, para_grid_NN)
clf.fit(X.iloc[:-n].values,
        realized_vol.iloc[1:-(n-1)].values.reshape(-1, ))
NN_predictions = clf.predict(X.iloc[-n:])
```

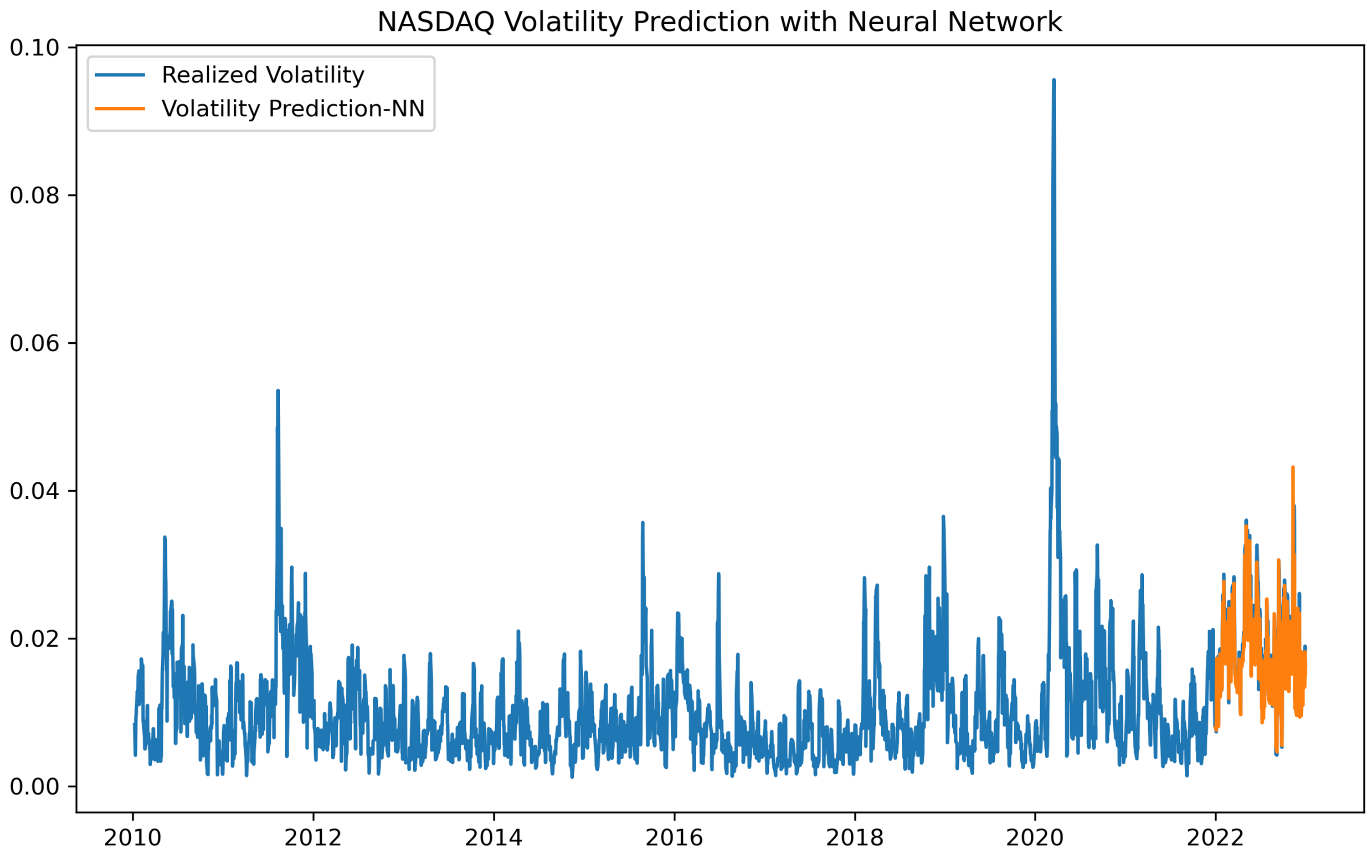
In [62]:
```
NN_predictions = pd.DataFrame(NN_predictions)
NN_predictions.index = ret.iloc[-n:].index
```

In [63]:
```
rmse_NN = np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                      NN_predictions / 100))
print('The RMSE value of NN is {:.6f}'.format(rmse_NN))
```

```
The RMSE value of NN is 0.001817
```

In [65]:
```
plt.figure(figsize=(10, 6))
plt.plot(realized_vol / 100, label='Realized Volatility')
plt.plot(NN_predictions / 100, label='Volatility Prediction-NN')
```

```
plt.title('NASDAQ Volatility Prediction with Neural Network', fontsize=12)
plt.legend()
plt.show()
```



DL-GARCH (Deep Learning Based on Keras)

In [66]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

- Configuring the network structure by deciding number of layers and neurons.

In [67]:
```
model = keras.Sequential(
    [layers.Dense(256, activation="relu"),
     layers.Dense(128, activation="relu"),
     layers.Dense(1, activation="linear"),])
```

- Compiling the model with loss and optimizer.

In [68]:
```
model.compile(loss='mse', optimizer='rmsprop')
```

- Deciding the epoch and batch size.
- Fitting the deep learning model.
- Predicting the volatility based on the weights obtained from the training phase.
- Calculating the RMSE score by flattening the predictions.

In [69]:
```
epochs_trial = np.arange(100, 400, 4)
batch_trial = np.arange(100, 400, 4)
DL_pred = []
```

```
DL_RMSE = []
for i, j, k in zip(range(4), epochs_trial, batch_trial):
    model.fit(X.iloc[:-n].values,
              realized_vol.iloc[1:-(n-1)].values.reshape(-1,),
              batch_size=k, epochs=j, verbose=False)
    DL_predict = model.predict(np.asarray(X.iloc[-n:]))
    DL_RMSE.append(np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                            DL_predict.flatten() / 100)))
    DL_pred.append(DL_predict)
    print('DL_RMSE_{}:{:.6f}'.format(i+1, DL_RMSE[i]))
```

```
8/8 [==============================] - 0s 5ms/step
DL_RMSE_1:0.001594
8/8 [==============================] - 0s 7ms/step
DL_RMSE_2:0.001300
8/8 [==============================] - 0s 5ms/step
DL_RMSE_3:0.001823
8/8 [==============================] - 0s 4ms/step
DL_RMSE_4:0.001289
```

In [70]:
```
DL_predict = pd.DataFrame(DL_pred[DL_RMSE.index(min(DL_RMSE))])
DL_predict.index = ret.iloc[-n:].index
```

In [71]:
```
plt.figure(figsize=(10, 6))
plt.plot(realized_vol / 100,label='Realized Volatility')
plt.plot(DL_predict / 100,label='Volatility Prediction-DL')
plt.title('Volatility Prediction with Deep Learning',  fontsize=12)
plt.legend()
plt.show()
```