Financial Time Series Analysis

There are two columns to be used, log-returns of a stock and its google trends. Let the log-returns of the stock on date t be X_t, and let the google trend (for that stock) on date t be Z_t. Google trend series is not the relative change series.

First, change Z_t to a relative change series, call it $Y_t = (Z_t - Z_{t-1})/Z_{t-1}$. This will reduce the series length by 1. Now you have the training data $(X_t, Y_t), 2 \leq t \leq T$.

Goal: The goal of your analysis is to use $Y_{t-1}, Y_{t-2}, \ldots$, and $X_{t-1}, X_{t-2}, \ldots$ to come up with prediction intervals for X_t, the future.

Output required: Plot the test series $\{X_t\}$ and the prediction intervals obtained. Additionally, provide plots for the width of the conformal prediction intervals and also trailing coverage probabilities over a window of length 20. Comment on the validity and accuracy of prediction intervals reported.

In [4]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
import statsmodels.tsa.api as tsa
from arch import arch_model
from scipy.special import comb

import warnings
warnings.filterwarnings("ignore")
```

In [5]:
```
train_df = pd.read_csv('./synthetic_train_AAPL.csv', index_col=0)
```

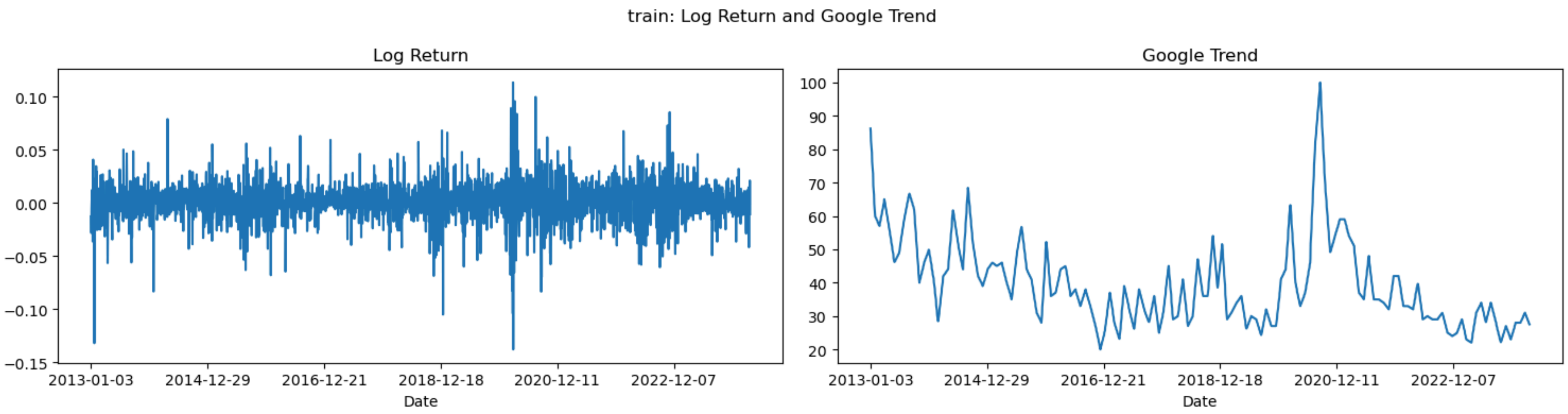
In [6]:
```
train_df.head()
```

Out[6]:

Date	Close	GoogleTrend	log_ret
2013-01-03	16.458612	86.193548	-0.012703
2013-01-04	16.000160	85.290323	-0.028250
2013-01-07	15.906043	82.580645	-0.005900
2013-01-08	15.948854	81.677419	0.002688
2013-01-09	15.699588	80.774194	-0.015753

In [10]:

```
fig, axes = plt.subplots(1, 2, figsize=(15, 4))
train_df['log_ret'].plot(ax=axes[0], title='Log Return')
train_df['GoogleTrend'].plot(ax=axes[1], title='Google Trend')
plt.suptitle('train: Log Return and Google Trend')
fig.tight_layout()
plt.show()
```



Step 1

*Create the dataset $(X_t, Y_{t-1}, Y_{t-2}), 4 \leq t \leq T$. This is a trivariate data. Regress X_t on Y_{t-1}, Y_{t-2} and obtain the coefficients and residuals. So, you get $u_t = X_t - (\hat{\gamma}_0 + \hat{\gamma}_1 Y_{t-1} + \hat{\gamma}_2 Y_{t-2})$. You should have $T - 3$ residual values. Save

$\hat{\gamma}_0$, $\hat{\gamma}_1$, **and** $\hat{\gamma}_2$**. You will need these later.***

In [11]:
```
train_df['Yt'] = train_df['GoogleTrend'].pct_change()
```

In [12]:
```
X = train_df['log_ret'].iloc[3:]
X.name = 'X'
Y_lag1 = train_df['Yt'].shift(1).iloc[3:]
Y_lag1.name = 'Y_lag1'
Y_lag2 = train_df['Yt'].shift(2).iloc[3:]
Y_lag2.name = 'Y_lag2'

assert len(X) == len(train_df) - 3
assert len(Y_lag1) == len(train_df) - 3
assert len(Y_lag2) == len(train_df) - 3
```

In [13]:
```
# regress X on Y_lag1 and Y_lag2
X_regressors = sm.add_constant(pd.concat([Y_lag1, Y_lag2], axis=1))
model = sm.OLS(X, X_regressors).fit()
```

In [14]:
```
gamma_hat_0, gamma_hat_1, gamma_hat_2 = model.params
```

Step 2

Test if $\{u_t\}$ is stationary or not using an appropriate test. (You should justify your choice of test.) If it is stationary, proceed to Step 3. If not, difference the series as many times as needed to obtain a stationary series. If you think necessary, you can apply transformations to the series. Call the resulting stationary series $\{v_t\}$. Depending on the differencing and your test of stationarity, this can be a shorter series than $\{u_t\}$.

In [15]:
```
ut = X - (gamma_hat_0 + gamma_hat_1 * Y_lag1 + gamma_hat_2 * Y_lag2)
```

We use the Augmented Dickey-Fuller (ADF) test to determine whether the residual is stationary. Testing for a unit root is essentially testing for stationarity. The ADF test incorporates lagged differences of the series to account for autocorrelation (serial correlation) in the residuals. Without this adjustment, we might obtain misleading results if the residuals are autocorrelated. Therefore, the ADF test is more robust.

In [16]:
```
def difference_until_stationary(series, max_diff=5, significance=0.05):
    n_diff = 0
    diffed_series = series.copy()
    adf_result = adfuller(diffed_series.dropna())
    p_value = adf_result[1]

    while p_value > significance and n_diff < max_diff:
        diffed_series = diffed_series.diff().dropna()
        n_diff += 1
        adf_result = adfuller(diffed_series)
        p_value = adf_result[1]

    return diffed_series, n_diff, adf_result

vt, n_diff, adf_result = difference_until_stationary(ut)

print(f"Number of differences applied: {n_diff}")
print(f"Final ADF Test p-value: {adf_result[1]}")

if n_diff == 0:
    print("Residuals were already stationary.")
else:
    print(f"Residuals became stationary after {n_diff} differencing(s).")
```

```
Number of differences applied: 0
Final ADF Test p-value: 7.01437200803369e-30
Residuals were already stationary.
```

Step 3

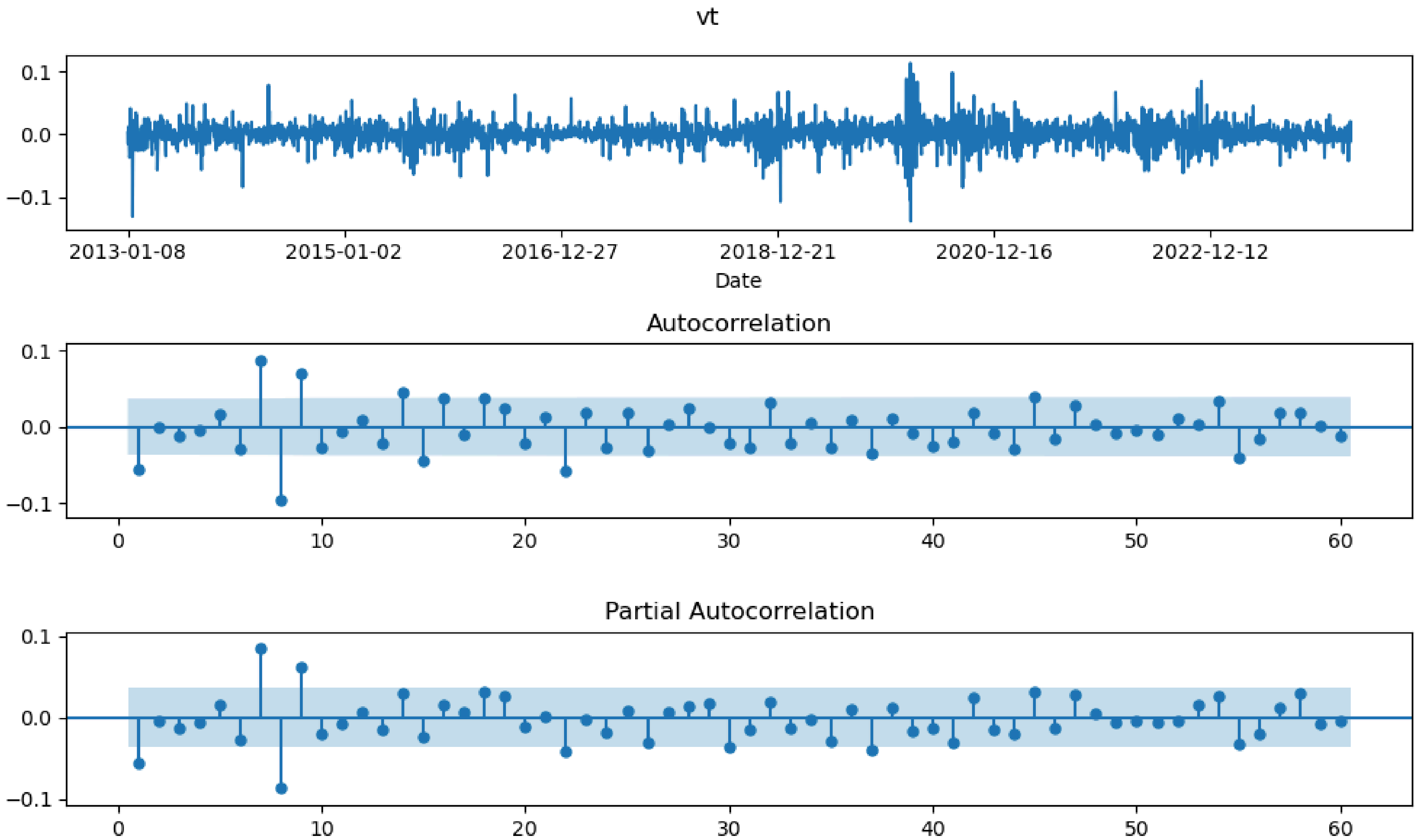
Find an appropriate ARMA model for $\{v_t\}$. Justify your steps, why you choose those specific p, q for the ARMA model. (Some of your justifications can also be computational constraints. For example, if within the time you cannot run a model with $p > 10$, you can use this as a justification to stop at $p = 10$.) Save the coefficient estimates $\hat{\phi}_1, \ldots, \hat{\phi}_p$ and $\hat{\theta}_1, \ldots, \hat{\theta}_q$. Calculate the residuals of the ARMA model: call that series $\{e_t\}$.

In [17]:
```
fig, axes = plt.subplots(3, 1, figsize=(10, 6))

vt.plot(ax=axes[0])
```

```
tsa.graphics.plot_acf(vt, zero=False, auto_ylims=True, lags=60, ax=axes[1])
tsa.graphics.plot_pacf(vt, zero=False, auto_ylims=True, lags=60, ax=axes[2])

plt.suptitle('vt')
fig.tight_layout()
plt.show()
```



The lags for ACF and PACF are significant for up to 10 lags. Due to computation constraints, we could choose an MA(10) and AR(10) model, and search for the optimal coefficietns based on AIC.

In [18]:
```
ar_order = 10
ma_order = 10
d = 0
```

In [19]:
```
arima_model = tsa.arma_order_select_ic(y=vt, max_ar=ar_order, max_ma=ma_order, ic="aic")
```

In [20]:
```
p, q = arima_model.aic_min_order
print(f"ARIMA(p, d, q) = ARIMA({p}, {d}, {q})")
```

```
ARIMA(p, d, q) = ARIMA(5, 0, 7)
```

In [21]:
```
arima_model_w_fx_degree = tsa.ARIMA(vt, order=(p, d, q))
arima_results = arima_model_w_fx_degree.fit()
```

In [22]:
```
et = arima_results.resid
```

Step 4

Test if $\{e_t\}$ has ARCH effects. If no, then set $\hat{\sigma}_t^2$ as the variance estimate from ARMA model. If yes, then find an appropriate GARCH model for $\{e_t\}$ and save the coefficient estimators $\hat{\alpha}_0, \hat{\alpha}_1, \ldots, \hat{\alpha}_m$ and $\hat{\beta}_1, \ldots, \hat{\beta}_s$. This allows you to compute $\hat{\sigma}_t$.

In [23]:
```
residuals_arma_sq = np.square(et)

sm.stats.acorr_ljungbox(residuals_arma_sq, lags=10, boxpierce=True)
```

Out[23]:

	lb_stat	lb_pvalue	bp_stat	bp_pvalue
1	110.287924	8.474328e-26	110.170887	8.989719e-26
2	200.178486	3.402469e-44	199.934260	3.844387e-44
3	238.072710	2.484709e-51	237.761463	2.901218e-51
4	284.734302	2.124133e-60	284.324020	2.604062e-60
5	370.923689	5.460064e-78	370.299992	7.439461e-78
6	431.561536	4.556422e-90	430.766242	6.756551e-90
7	463.367955	5.974232e-96	462.471403	9.308341e-96
8	530.388383	2.113921e-109	529.254759	3.702331e-109
9	571.394351	2.875108e-117	570.101170	5.445439e-117
10	616.196664	5.953841e-126	614.713307	1.238029e-125

The p-values are very significant, so we reject the null hypothesis that the process is a white noise with no autocorrelation. Therefore, there are ARCH effects, i.e. the volatility at time t is predictive of volatility at time $t + h$.

In [24]:

```
_, p_lagrange, _, p_f = sm.stats.diagnostic.het_arch(et, nlags=10)
print("Engle's test p-value (Lagrange):", p_lagrange)
print("Engle's test p-value (F-test):", p_f)
```

```
Engle's test p-value (Lagrange): 7.003560664248858e-54
Engle's test p-value (F-test): 7.884175905803032e-57
```

The ARCH effects is once again confirmed by the Engle test.

In [25]:

```
model_aic = []
for p in range(1, 5):
    for q in range(1, 5):
        model = arch_model(et, vol='Garch', p=p, q=q, rescale=False)
        result = model.fit(disp="off")
        print(f"GARCH({p}, {q}) AIC: {result.aic}")
        model_aic.append([[p, q], result.aic])
```

```
GARCH(1, 1) AIC: -15244.555001015931
GARCH(1, 2) AIC: -15238.514743023006
GARCH(1, 3) AIC: -15235.94467475155
GARCH(1, 4) AIC: -15257.83243705662
GARCH(2, 1) AIC: -15233.96379426013
GARCH(2, 2) AIC: -15245.614381597621
GARCH(2, 3) AIC: -15252.993116714224
GARCH(2, 4) AIC: -15256.229930535763
GARCH(3, 1) AIC: -15223.2059236598
GARCH(3, 2) AIC: -15243.526128304973
GARCH(3, 3) AIC: -15249.26650769521
GARCH(3, 4) AIC: -15248.735855181254
GARCH(4, 1) AIC: -15225.928987412059
GARCH(4, 2) AIC: -15254.49158099038
GARCH(4, 3) AIC: -15238.05948585339
GARCH(4, 4) AIC: -15237.989003535196
```

In [26]:
```
pd.DataFrame(model_aic, columns=['p,q', 'AIC']).sort_values(by='AIC', ascending=True).head(10)
```

Out[26]:

	p,q	AIC
3	[1, 4]	-15257.832437
7	[2, 4]	-15256.229931
13	[4, 2]	-15254.491581
6	[2, 3]	-15252.993117
10	[3, 3]	-15249.266508
11	[3, 4]	-15248.735855
5	[2, 2]	-15245.614382
0	[1, 1]	-15244.555001
9	[3, 2]	-15243.526128
1	[1, 2]	-15238.514743

According to the AIC, we can choose GARCH(1, 1) to model e_t, but GARCH(1, 1) is alreay sufficient to eliminate the ARCH effect.

In [28]:
```
scale = 100
garch_model = arch_model(scale * et, p=1, q=1)
garch_results = garch_model.fit()
garch_results.summary()
```

```
Iteration:      1,   Func. Count:      6,   Neg. LLF: 32597641921.415146
Iteration:      2,   Func. Count:     14,   Neg. LLF: 15724446288.951365
Iteration:      3,   Func. Count:     22,   Neg. LLF: 5680.072207534429
Iteration:      4,   Func. Count:     30,   Neg. LLF: 5859.979740295255
Iteration:      5,   Func. Count:     37,   Neg. LLF: 5395.205339422912
Iteration:      6,   Func. Count:     43,   Neg. LLF: 5374.57518892362
Iteration:      7,   Func. Count:     48,   Neg. LLF: 5374.469388062561
Iteration:      8,   Func. Count:     53,   Neg. LLF: 5374.458122707876
Iteration:      9,   Func. Count:     58,   Neg. LLF: 5374.456819457536
Iteration:     10,   Func. Count:     63,   Neg. LLF: 5374.455818368411
Iteration:     11,   Func. Count:     68,   Neg. LLF: 5374.455816908305
Iteration:     12,   Func. Count:     72,   Neg. LLF: 5374.455816908136
Optimization terminated successfully    (Exit mode 0)
            Current function value: 5374.455816908305
            Iterations: 12
            Function evaluations: 72
            Gradient evaluations: 12
```

Out[28]:

Constant Mean - GARCH Model Results

Dep. Variable:	None	**R-squared:**	0.000
Mean Model:	Constant Mean	**Adj. R-squared:**	0.000
Vol Model:	GARCH	**Log-Likelihood:**	-5374.46
Distribution:	Normal	**AIC:**	10756.9
Method:	Maximum Likelihood	**BIC:**	10780.7
		No. Observations:	2825
Date:	Sat, May 03 2025	**Df Residuals:**	2824
Time:	11:32:25	**Df Model:**	1

Mean Model

	coef	**std err**	**t**	**P>|t|**	**95.0% Conf. Int.**
mu	0.0552	3.049e-02	1.812	7.003e-02	[-4.519e-03, 0.115]

Volatility Model

	coef	**std err**	**t**	**P>|t|**	**95.0% Conf. Int.**
omega	0.1291	4.282e-02	3.015	2.572e-03	[4.516e-02, 0.213]
alpha[1]	0.0888	2.099e-02	4.229	2.344e-05	[4.763e-02, 0.130]
beta[1]	0.8679	3.003e-02	28.900	1.210e-183	[0.809, 0.927]

Covariance estimator: robust

In [29]:

```
st_residuals = np.divide(
    garch_results.resid,
    garch_results.conditional_volatility,
)
```

In [30]:
```
sm.stats.acorr_ljungbox(st_residuals**2, lags=10, boxpierce=True)
```

Out[30]:

	lb_stat	lb_pvalue	bp_stat	bp_pvalue
1	0.034811	0.851992	0.034774	0.852070
2	0.102561	0.950012	0.102428	0.950075
3	0.642693	0.886593	0.641605	0.886845
4	1.943934	0.746070	1.940084	0.746778
5	1.952077	0.855736	1.948207	0.856264
6	2.109784	0.909330	2.105468	0.909748
7	2.829758	0.900292	2.823150	0.900866
8	3.296948	0.914365	3.288687	0.914957
9	3.297391	0.951329	3.289128	0.951721
10	3.650717	0.961731	3.640954	0.962094

In [31]:
```
_, p_lagrange, _, p_f = sm.stats.diagnostic.het_arch(
    st_residuals,
    nlags=10,
)
print("Engle's test p-value (Lagrange):", p_lagrange)

print("Engle's test p-value (F-test):", p_f)
```

```
Engle's test p-value (Lagrange): 0.9636849645218315
Engle's test p-value (F-test): 0.9639246202427323
```

There is no more ARCH effect.

Step 5

Using the conformity score $s_t(v) = |v - \hat{v}_t|/\hat{\sigma}_t$ compute sequential conformal prediction intervals based on the test data. Convert these prediction intervals for $\{v_t\}$ to prediction intervals for $\{X_t\}$ using your calculations in Steps 1 and 2.

```
In [32]: test_df = pd.read_csv('./synthetic_test_AAPL.csv', index_col=0)
         test_df = pd.concat([train_df.iloc[-3:], test_df])
```

```
In [33]: test_df['Yt'] = test_df['GoogleTrend'].pct_change()
         vt_true = (test_df['log_ret'] - (gamma_hat_0 + gamma_hat_1 * test_df['Yt'].shift(1) \
                                     + gamma_hat_2 * test_df['Yt'].shift(2))).dropna()
```

```
In [34]: def get_conformal_pred_interval(ts, ts_pred, cond_vol):
             st = np.abs(ts - ts_pred) / cond_vol

             B = max(5 * ts.iloc[0], 10)
             C = 6
             pt, qt = np.array([0]), np.array([0])

             for t in range(1, len(ts)):
                 ind = 1 if st.iloc[t] > qt[-1] else 0

                 p_new = pt[-1] + ind - 0.05
                 pt = np.r_[pt, p_new]

                 q_new = B * np.tan(min(max(np.log(t) * p_new / (t * C), -np.pi/2), np.pi/2))
                 qt = np.r_[qt, q_new]

             return qt
```

```
In [35]: garch_fct_results = garch_results.forecast(horizon=len(test_df), reindex=False,)
         vol_forecasts = np.sqrt(garch_fct_results.variance) / scale
```

```
In [36]: def reconstruct_u_from_v(v_pred, u_last_values, N):
             if N == 0:
                 return v_pred

             u_pred = []
             buffer = list(u_last_values)

             for v_hat in v_pred:
                 u_hat = 0.0
                 for k in range(1, N+1):
```

```
            coeff = comb(N, k) * (-1)**(k+1)
            u_hat += coeff * buffer[-k]

        u_hat += v_hat
        u_pred.append(u_hat)

        buffer.append(u_hat)
        buffer.pop(0)

    return np.array(u_pred)
```

In [39]:
```
pred_df = pd.DataFrame()
pred_df['vt_true'] = vt_true
pred_df["cond_vol"] = vol_forecasts.values[-1, 3:]
pred_df["vt_pred"] = arima_results.forecast(steps=len(test_df)-3).values
pred_df["log_ret_true"] = test_df['log_ret'].iloc[3:]
pred_df.index = test_df.index[3:]
```

In [40]:
```
ut_pred = reconstruct_u_from_v(pred_df["vt_pred"], ut[-n_diff:], n_diff)
pred_df["log_ret_pred"] = gamma_hat_0 + gamma_hat_1 * test_df['Yt'].shift(1) \
                        + gamma_hat_2 * test_df['Yt'].shift(2) + ut_pred

qt = get_conformal_pred_interval(pred_df['vt_true'], pred_df['vt_pred'], pred_df['cond_vol'])
pred_df['conf_low'] = pred_df["log_ret_pred"] - qt * pred_df['cond_vol']
pred_df['conf_high'] = pred_df["log_ret_pred"] + qt * pred_df['cond_vol']

pred_df["is_covered"] = (pred_df['log_ret_true'] >= pred_df['conf_low']) \
                        * (pred_df['log_ret_true'] <= pred_df['conf_high']) * 1
```

In [41]:
```
pred_df.head()
```

Out[41]:

Date	vt_true	cond_vol	vt_pred	log_ret_true	log_ret_pred	conf_low	conf_high	is_covered
2024-04-01	-0.009141	0.016315	0.001560	-0.008492	0.002209	0.002209	0.002209	0
2024-04-02	-0.007594	0.016357	-0.003786	-0.007023	-0.003216	-0.003216	-0.003216	0
2024-04-03	0.004623	0.016397	0.003886	0.004786	0.004049	-0.014019	0.022118	1
2024-04-04	-0.006030	0.016436	-0.002657	-0.004904	-0.001532	-0.020169	0.017105	1
2024-04-05	0.003368	0.016472	0.000298	0.004492	0.001422	-0.015767	0.018610	1

In [42]:

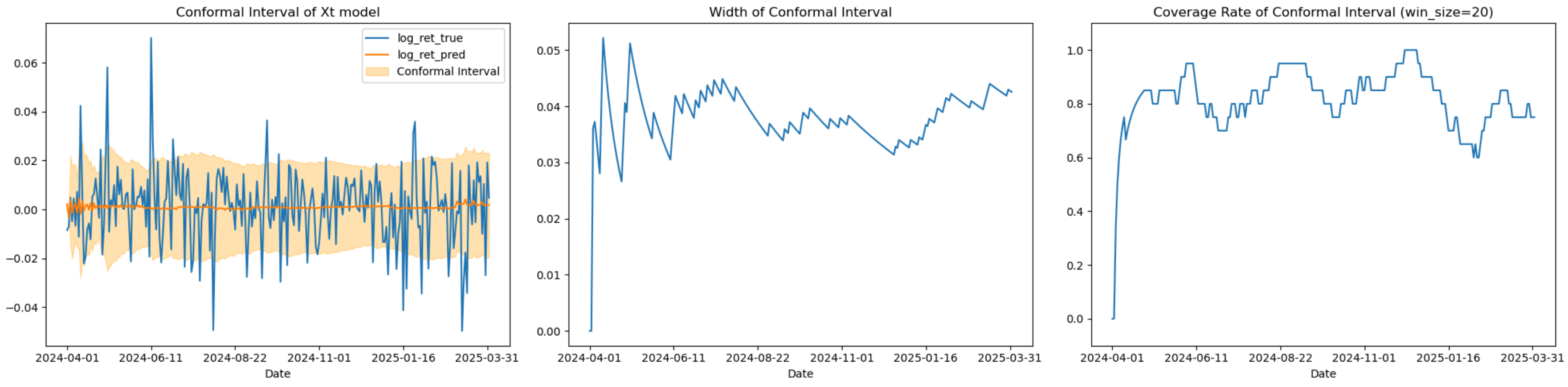
```
fig, axes = plt.subplots(1, 3, figsize=(20, 5))

pred_df[['log_ret_true', 'log_ret_pred']].plot(ax=axes[0])
axes[0].fill_between(pred_df.index, pred_df['conf_low'], pred_df['conf_high'],
                        alpha=0.3, color='orange', label='Conformal Interval')
axes[0].set_title(f"Conformal Interval of Xt model")
axes[0].legend()

(pred_df["conf_high"] - pred_df["conf_low"]).plot(ax=axes[1])
axes[1].set_title(f"Width of Conformal Interval")

pred_df["is_covered"].rolling(20, min_periods=1).mean().plot(ax=axes[2])
axes[2].set_ylim(-0.1, 1.1)
axes[2].set_title(f"Coverage Rate of Conformal Interval (win_size=20)")

fig.tight_layout()
plt.show()
```



- Validity:
  - Intervals are valid most of the time — the true X_t falls inside the predicted bands.
  - Empirical coverage matches the intended target reasonably well.
  - Adaptivity to volatility is clearly working.
- Accuracy:
  - Predicted point values track the general movement of true values but remain centered.
  - Widths react dynamically to local uncertainty — not static.
- Limitations:
  - Sharp market shocks are harder to catch immediately — occasional undercoverage spikes.
  - Slight early instability (first few days) when model has limited training history.

In summary, the constructed conformal prediction intervals achieve near-nominal coverage and adapt to local volatility dynamics, validating the modeling approach. Some transient undercoverage periods may be addressed by enhanced volatility modeling or dynamic recalibration.