

```
In [8]: # Simple imports for data, models, and plots
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error

from xgboost import XGBRegressor

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

np.random.seed(42)
```

```
In [9]: # Load dataset from CSV
file_path = r"C:\Users\Raed.tattan\Dataset Golden\gold_data_2015_25.csv"
df = pd.read_csv(file_path)
df.head()
```

```
Out[9]:
```

	Date	SPX	GLD	USO	SLV	EUR/USD
0	2015-01-02	2058.199951	114.080002	159.119995	15.11	1.208941
1	2015-01-05	2020.579956	115.800003	150.320007	15.50	1.194643
2	2015-01-06	2002.609985	117.120003	144.399994	15.83	1.193902
3	2015-01-07	2025.900024	116.430000	146.960007	15.85	1.187536
4	2015-01-08	2062.139893	115.940002	148.399994	15.64	1.183600

```
In [10]: # Parse date, sort, and filter study period
df['Date'] = pd.to_datetime(df['Date'])
df = df.sort_values('Date').reset_index(drop=True)

start_date = "2015-01-01"
end_date = "2025-12-31"
mask = (df['Date'] >= start_date) & (df['Date'] <= end_date)
df_period = df.loc[mask].copy().reset_index(drop=True)

df_period.head()
```

```
Out[10]:
```

	Date	SPX	GLD	USO	SLV	EUR/USD
0	2015-01-02	2058.199951	114.080002	159.119995	15.11	1.208941
1	2015-01-05	2020.579956	115.800003	150.320007	15.50	1.194643
2	2015-01-06	2002.609985	117.120003	144.399994	15.83	1.193902
3	2015-01-07	2025.900024	116.430000	146.960007	15.85	1.187536
4	2015-01-08	2062.139893	115.940002	148.399994	15.64	1.183600

```
In [11]: # Fill missing values and rename columns for consistency
df_period = df_period.ffill().bfill()
```

```
df_period.rename(columns={
    'GLD': 'Gold_Price',
    'EUR/USD': 'EUR_USD'
}, inplace=True)

df_period.head()
```

```
Out[11]:
```

	Date	SPX	Gold_Price	USO	SLV	EUR_USD
0	2015-01-02	2058.199951	114.080002	159.119995	15.11	1.208941
1	2015-01-05	2020.579956	115.800003	150.320007	15.50	1.194643
2	2015-01-06	2002.609985	117.120003	144.399994	15.83	1.193902
3	2015-01-07	2025.900024	116.430000	146.960007	15.85	1.187536
4	2015-01-08	2062.139893	115.940002	148.399994	15.64	1.183600

```
In [12]: # Create returns, lags, moving averages, volatility, RSI, and MACD
df_period['Gold_Return_pct'] = (df_period['Gold_Price'] - df_period['Gold_Price'].shift(1)) / df_period['Gold_Price']
df_period['Gold_Return_log'] = np.log(df_period['Gold_Return_pct'] + 1)

for col in ['SPX', 'USO', 'SLV', 'EUR_USD']:
    df_period[f'{col}_Return_log'] = np.log(df_period[col] / df_period[col].shift(1))

df_period['Gold_Lag_1'] = df_period['Gold_Price'].shift(1)
df_period['Gold_Lag_3'] = df_period['Gold_Price'].shift(3)
df_period['Gold_Lag_5'] = df_period['Gold_Price'].shift(5)
df_period['Gold_Lag_10'] = df_period['Gold_Price'].shift(10)

df_period['Gold_Diff_1'] = df_period['Gold_Price'] - df_period['Gold_Price'].shift(1)
df_period['Gold_Diff_5'] = df_period['Gold_Price'] - df_period['Gold_Price'].shift(5)
df_period['Gold_Diff_10'] = df_period['Gold_Price'] - df_period['Gold_Price'].shift(10)

df_period['Gold_Return_log_Lag_1'] = df_period['Gold_Return_log'].shift(1)
df_period['Gold_Return_log_Lag_5'] = df_period['Gold_Return_log'].shift(5)

df_period['Gold_MA_7'] = df_period['Gold_Price'].rolling(window=7).mean()
df_period['Gold_MA_21'] = df_period['Gold_Price'].rolling(window=21).mean()
df_period['Gold_MA_63'] = df_period['Gold_Price'].rolling(window=63).mean()

df_period['Gold_Volatility_21'] = df_period['Gold_Return_log'].rolling(window=21).std()
df_period['Gold_Volatility_63'] = df_period['Gold_Return_log'].rolling(window=63).std()

df_period['Gold_Momentum_5'] = df_period['Gold_Price'] - df_period['Gold_Price'].shift(5)
df_period['Gold_Momentum_10'] = df_period['Gold_Price'] - df_period['Gold_Price'].shift(10)

# RSI(14)
rsi_window = 14
delta = df_period['Gold_Return_log'].diff()
gain = delta.clip(lower=0)
loss = -delta.clip(upper=0)
avg_gain = gain.rolling(window=rsi_window).mean()
avg_loss = loss.rolling(window=rsi_window).mean()
rs = avg_gain / avg_loss
df_period['RSI_14'] = 100 - (100 / (1 + rs))

# MACD(12,26) + Signal(9) + Hist
```

```

ema_12 = df_period['Gold_Price'].ewm(span=12, adjust=False).mean()
ema_26 = df_period['Gold_Price'].ewm(span=26, adjust=False).mean()
df_period['MACD'] = ema_12 - ema_26
df_period['MACD_Signal'] = df_period['MACD'].ewm(span=9, adjust=False).mean()
df_period['MACD_Hist'] = df_period['MACD'] - df_period['MACD_Signal']

df_period = df_period.dropna().reset_index(drop=True)
df_period.head()

```

```

Out[12]:
   Date      SPX  Gold_Price  USO  SLV  EUR_USD  Gold_Return_pct  Gold_Return_log  SP
0 2015-04-06  2080.620117  116.690002  147.440002  16.27  1.100134      0.012231      0.012157
1 2015-04-07  2076.330078  116.110001  152.479996  16.15  1.093482     -0.004970     -0.004983
2 2015-04-08  2081.899902  115.470001  144.559998  15.81  1.082415     -0.005512     -0.005527
3 2015-04-09  2091.179932  114.669998  144.240005  15.51  1.078528     -0.006928     -0.006952
4 2015-04-10  2102.060059  115.970001  147.279999  15.77  1.066906      0.011337      0.011273

```

5 rows × 32 columns

```

In [13]: # Use Log-return as target for better stationarity
target_col = 'Gold_Return_log'

feature_cols = [
    'SPX', 'USO', 'SLV', 'EUR_USD',
    'SPX_Return_log', 'USO_Return_log', 'SLV_Return_log', 'EUR_USD_Return_log',
    'Gold_Return_pct', 'Gold_Return_log',
    'Gold_Lag_1', 'Gold_Lag_3', 'Gold_Lag_5', 'Gold_Lag_10',
    'Gold_Diff_1', 'Gold_Diff_5', 'Gold_Diff_10',
    'Gold_Return_log_Lag_1', 'Gold_Return_log_Lag_5',
    'Gold_MA_7', 'Gold_MA_21', 'Gold_MA_63',
    'Gold_Volatility_21', 'Gold_Volatility_63',
    'Gold_Momentum_5', 'Gold_Momentum_10',
    'RSI_14', 'MACD', 'MACD_Signal', 'MACD_Hist'
]

print("Target:", target_col)
print("Number of features:", len(feature_cols))

```

Target: Gold_Return_log
Number of features: 30

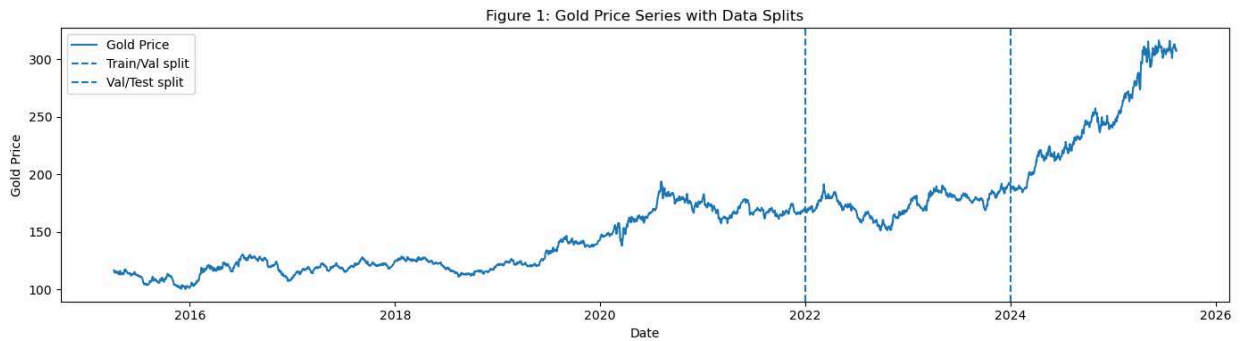
```

In [14]: # Plot gold price and dataset splits for documentation
plt.figure(figsize=(14,4))
plt.plot(df_period['Date'], df_period['Gold_Price'], label='Gold Price')
plt.axvline(pd.to_datetime('2022-01-01'), linestyle='--', label='Train/Val split')
plt.axvline(pd.to_datetime('2024-01-01'), linestyle='--', label='Val/Test split')
plt.title("Figure 1: Gold Price Series with Data Splits")
plt.xlabel("Date")
plt.ylabel("Gold Price")
plt.legend()

```

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

```
# plt.savefig("Figure_1_Gold_Splits.png", dpi=300, bbox_inches="tight")
```



```
In [15]: # Split data into train/validation/test using time boundaries
train_df = df_period[df_period['Date'] < '2022-01-01'].copy()
val_df    = df_period[(df_period['Date'] >= '2022-01-01') & (df_period['Date'] < '2024-01-01')].copy()
test_df   = df_period[df_period['Date'] >= '2024-01-01'].copy()

print("Train:", len(train_df), "Val:", len(val_df), "Test:", len(test_df))
```

Train: 1697 Val: 501 Test: 405

```
In [16]: # Build X/y arrays for XGBoost and LSTM
X_train = train_df[feature_cols].values
y_train = train_df[target_col].values

X_val = val_df[feature_cols].values
y_val = val_df[target_col].values

X_test = test_df[feature_cols].values
y_test = test_df[target_col].values
```

```
In [2]: pip install -U xgboost
```

Requirement already satisfied: xgboost in c:\users\raed.tattan\anaconda3\lib\site-packages (2.1.4)
Requirement already satisfied: numpy in c:\users\raed.tattan\anaconda3\lib\site-packages (from xgboost) (1.24.3)
Requirement already satisfied: scipy in c:\users\raed.tattan\anaconda3\lib\site-packages (from xgboost) (1.9.1)
Note: you may need to restart the kernel to use updated packages.

```
In [17]: import xgboost as xgb
from xgboost import XGBRegressor

xgb_model = XGBRegressor(
    n_estimators=4000,
    max_depth=5,
    learning_rate=0.03,
    subsample=0.85,
    colsample_bytree=0.85,
    reg_lambda=1.0,
    objective='reg:squarederror',
    random_state=42,
    n_jobs=-1
)
```

```
xgb_model.fit(
    X_train, y_train,
    eval_set=[(X_val, y_val)],
    verbose=False
)

print("Best iteration:", getattr(xgb_model, "best_iteration", None))
```

Best iteration: None

```
In [18]: # Scale features and create sequences for LSTM
scaler_X = MinMaxScaler()
X_train_s = scaler_X.fit_transform(X_train)
X_val_s = scaler_X.transform(X_val)
X_test_s = scaler_X.transform(X_test)

scaler_y = MinMaxScaler()
y_train_s = scaler_y.fit_transform(y_train.reshape(-1,1))
y_val_s = scaler_y.transform(y_val.reshape(-1,1))
y_test_s = scaler_y.transform(y_test.reshape(-1,1))

def create_sequences(X, y, window_size):
    X_seq, y_seq = [], []
    for i in range(window_size, len(X)):
        X_seq.append(X[i-window_size:i, :])
        y_seq.append(y[i, :])
    return np.array(X_seq), np.array(y_seq)

window_size = 30
X_train_lstm, y_train_lstm = create_sequences(X_train_s, y_train_s, window_size)
X_val_lstm, y_val_lstm = create_sequences(X_val_s, y_val_s, window_size)
X_test_lstm, y_test_lstm = create_sequences(X_test_s, y_test_s, window_size)

print("LSTM shapes:", X_train_lstm.shape, y_train_lstm.shape)
```

LSTM shapes: (1667, 30, 30) (1667, 1)

```
In [19]: # Train LSTM to predict log-return with early stopping
lstm_model = Sequential()
lstm_model.add(LSTM(64, input_shape=(X_train_lstm.shape[1], X_train_lstm.shape[2])))
lstm_model.add(Dropout(0.2))
lstm_model.add(Dense(1))

lstm_model.compile(loss='mse', optimizer='adam', metrics=['mae'])

es = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

history = lstm_model.fit(
    X_train_lstm, y_train_lstm,
    validation_data=(X_val_lstm, y_val_lstm),
    epochs=100,
    batch_size=32,
    callbacks=[es],
    verbose=1
)
```

Epoch 1/100
53/53 [=====] - 4s 22ms/step - loss: 0.0304 - mae: 0.1323 -
val_loss: 0.0081 - val_mae: 0.0685
Epoch 2/100
53/53 [=====] - 1s 12ms/step - loss: 0.0128 - mae: 0.0868 -
val_loss: 0.0092 - val_mae: 0.0740
Epoch 3/100
53/53 [=====] - 1s 23ms/step - loss: 0.0113 - mae: 0.0799 -
val_loss: 0.0095 - val_mae: 0.0760
Epoch 4/100
53/53 [=====] - 1s 22ms/step - loss: 0.0101 - mae: 0.0763 -
val_loss: 0.0088 - val_mae: 0.0723
Epoch 5/100
53/53 [=====] - 1s 24ms/step - loss: 0.0099 - mae: 0.0750 -
val_loss: 0.0093 - val_mae: 0.0750
Epoch 6/100
53/53 [=====] - 1s 25ms/step - loss: 0.0091 - mae: 0.0712 -
val_loss: 0.0088 - val_mae: 0.0726
Epoch 7/100
53/53 [=====] - 1s 24ms/step - loss: 0.0097 - mae: 0.0737 -
val_loss: 0.0087 - val_mae: 0.0722
Epoch 8/100
53/53 [=====] - 1s 19ms/step - loss: 0.0092 - mae: 0.0723 -
val_loss: 0.0080 - val_mae: 0.0689
Epoch 9/100
53/53 [=====] - 1s 16ms/step - loss: 0.0092 - mae: 0.0716 -
val_loss: 0.0083 - val_mae: 0.0701
Epoch 10/100
53/53 [=====] - 1s 14ms/step - loss: 0.0088 - mae: 0.0705 -
val_loss: 0.0081 - val_mae: 0.0690
Epoch 11/100
53/53 [=====] - 1s 19ms/step - loss: 0.0094 - mae: 0.0733 -
val_loss: 0.0086 - val_mae: 0.0710
Epoch 12/100
53/53 [=====] - 1s 12ms/step - loss: 0.0088 - mae: 0.0701 -
val_loss: 0.0079 - val_mae: 0.0684
Epoch 13/100
53/53 [=====] - 1s 15ms/step - loss: 0.0086 - mae: 0.0685 -
val_loss: 0.0079 - val_mae: 0.0685
Epoch 14/100
53/53 [=====] - 1s 18ms/step - loss: 0.0089 - mae: 0.0701 -
val_loss: 0.0082 - val_mae: 0.0694
Epoch 15/100
53/53 [=====] - 1s 13ms/step - loss: 0.0088 - mae: 0.0705 -
val_loss: 0.0084 - val_mae: 0.0699
Epoch 16/100
53/53 [=====] - 1s 19ms/step - loss: 0.0085 - mae: 0.0689 -
val_loss: 0.0079 - val_mae: 0.0680
Epoch 17/100
53/53 [=====] - 1s 22ms/step - loss: 0.0085 - mae: 0.0688 -
val_loss: 0.0080 - val_mae: 0.0684
Epoch 18/100
53/53 [=====] - 1s 24ms/step - loss: 0.0081 - mae: 0.0668 -
val_loss: 0.0079 - val_mae: 0.0683
Epoch 19/100
53/53 [=====] - 1s 17ms/step - loss: 0.0082 - mae: 0.0674 -
val_loss: 0.0088 - val_mae: 0.0731
Epoch 20/100
53/53 [=====] - 1s 23ms/step - loss: 0.0084 - mae: 0.0681 -
val_loss: 0.0091 - val_mae: 0.0732

```

Epoch 21/100
53/53 [=====] - 1s 22ms/step - loss: 0.0085 - mae: 0.0690 -
val_loss: 0.0079 - val_mae: 0.0685
Epoch 22/100
53/53 [=====] - 1s 25ms/step - loss: 0.0083 - mae: 0.0680 -
val_loss: 0.0084 - val_mae: 0.0716
Epoch 23/100
53/53 [=====] - 1s 24ms/step - loss: 0.0090 - mae: 0.0713 -
val_loss: 0.0081 - val_mae: 0.0695
Epoch 24/100
53/53 [=====] - 1s 22ms/step - loss: 0.0083 - mae: 0.0672 -
val_loss: 0.0080 - val_mae: 0.0687
Epoch 25/100
53/53 [=====] - 1s 20ms/step - loss: 0.0082 - mae: 0.0675 -
val_loss: 0.0081 - val_mae: 0.0686
Epoch 26/100
53/53 [=====] - 1s 19ms/step - loss: 0.0081 - mae: 0.0670 -
val_loss: 0.0081 - val_mae: 0.0691
Epoch 27/100
53/53 [=====] - 1s 18ms/step - loss: 0.0082 - mae: 0.0670 -
val_loss: 0.0079 - val_mae: 0.0686
Epoch 28/100
53/53 [=====] - 1s 20ms/step - loss: 0.0077 - mae: 0.0649 -
val_loss: 0.0080 - val_mae: 0.0686

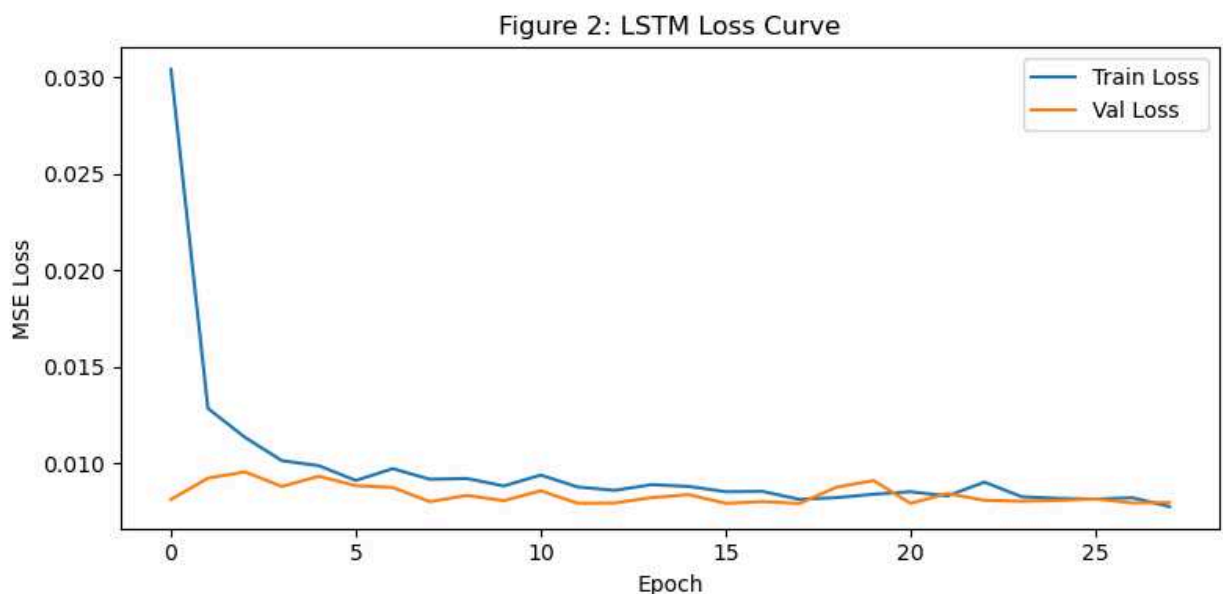
```

```

In [21]: # Plot LSTM training/validation loss for reporting
plt.figure(figsize=(8,4))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title("Figure 2: LSTM Loss Curve")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.legend()
plt.tight_layout()
plt.show()

plt.savefig("Figure_2_LSTM_Loss.png", dpi=300, bbox_inches="tight")

```



<Figure size 640x480 with 0 Axes>

```
In [22]: # Predict log-returns and align lengths due to window_size
y_pred_xgb_test = xgb_model.predict(X_test)
y_pred_xgb_aligned = y_pred_xgb_test[window_size:]

y_pred_lstm_test_s = lstm_model.predict(X_test_lstm)
y_pred_lstm_test = scaler_y.inverse_transform(y_pred_lstm_test_s).flatten()

y_true_aligned = y_test[window_size:]

print("Aligned lengths:", len(y_true_aligned), len(y_pred_xgb_aligned), len(y_pred_lstm_test))

12/12 [=====] - 0s 4ms/step
Aligned lengths: 375 375 375
```

```
In [23]: # Select best alpha on validation set for weighted hybrid
y_pred_xgb_val = xgb_model.predict(X_val)
y_pred_xgb_val_aligned = y_pred_xgb_val[window_size:]
y_true_val_aligned = y_val[window_size:]

y_pred_lstm_val_s = lstm_model.predict(X_val_lstm)
y_pred_lstm_val = scaler_y.inverse_transform(y_pred_lstm_val_s).flatten()

alphas = np.arange(0.0, 1.01, 0.1)
best_alpha, best_rmse = None, np.inf

for a in alphas:
    y_val_h = a * y_pred_xgb_val_aligned + (1 - a) * y_pred_lstm_val
    rmse = mean_squared_error(y_true_val_aligned, y_val_h, squared=False)
    if rmse < best_rmse:
        best_rmse, best_alpha = rmse, a

print("Best alpha:", best_alpha, "Val RMSE:", round(best_rmse, 6))

y_pred_hybrid_alpha = best_alpha * y_pred_xgb_aligned + (1 - best_alpha) * y_pred_lstm_val

15/15 [=====] - 0s 5ms/step
Best alpha: 1.0 Val RMSE: 0.000539
```

```
In [24]: # Compute residuals from XGBoost predictions (true - predicted)
residual_train = y_train - xgb_model.predict(X_train)
residual_val = y_val - xgb_model.predict(X_val)
residual_test = y_test - xgb_model.predict(X_test)
```

```
In [25]: # Scale residuals separately for residual LSTM
scaler_res = MinMaxScaler()

res_train_s = scaler_res.fit_transform(residual_train.reshape(-1,1))
res_val_s = scaler_res.transform(residual_val.reshape(-1,1))
res_test_s = scaler_res.transform(residual_test.reshape(-1,1))

# Create sequences for residual LSTM
X_train_res, y_train_res = create_sequences(X_train_s, res_train_s, window_size)
X_val_res, y_val_res = create_sequences(X_val_s, res_val_s, window_size)
X_test_res, y_test_res = create_sequences(X_test_s, res_test_s, window_size)

print("Residual LSTM shapes:", X_train_res.shape, y_train_res.shape)

Residual LSTM shapes: (1667, 30, 30) (1667, 1)
```

```
In [26]: # Train LSTM to predict residuals of XGBoost
res_lstm = Sequential()
res_lstm.add(LSTM(64, input_shape=(X_train_res.shape[1], X_train_res.shape[2])))
res_lstm.add(Dropout(0.2))
res_lstm.add(Dense(1))

res_lstm.compile(loss='mse', optimizer='adam')

es_res = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

history_res = res_lstm.fit(
    X_train_res, y_train_res,
    validation_data=(X_val_res, y_val_res),
    epochs=100,
    batch_size=32,
    callbacks=[es_res],
    verbose=1
)
```

Epoch 1/100
53/53 [=====] - 4s 23ms/step - loss: 0.0532 - val_loss: 0.0365

Epoch 2/100
53/53 [=====] - 1s 13ms/step - loss: 0.0130 - val_loss: 0.0351

Epoch 3/100
53/53 [=====] - 1s 16ms/step - loss: 0.0085 - val_loss: 0.0348

Epoch 4/100
53/53 [=====] - 1s 13ms/step - loss: 0.0062 - val_loss: 0.0347

Epoch 5/100
53/53 [=====] - 1s 12ms/step - loss: 0.0052 - val_loss: 0.0351

Epoch 6/100
53/53 [=====] - 1s 16ms/step - loss: 0.0046 - val_loss: 0.0346

Epoch 7/100
53/53 [=====] - 1s 18ms/step - loss: 0.0044 - val_loss: 0.0352

Epoch 8/100
53/53 [=====] - 1s 18ms/step - loss: 0.0040 - val_loss: 0.0345

Epoch 9/100
53/53 [=====] - 1s 23ms/step - loss: 0.0036 - val_loss: 0.0353

Epoch 10/100
53/53 [=====] - 1s 19ms/step - loss: 0.0037 - val_loss: 0.0345

Epoch 11/100
53/53 [=====] - 1s 13ms/step - loss: 0.0043 - val_loss: 0.0346

Epoch 12/100
53/53 [=====] - 1s 16ms/step - loss: 0.0033 - val_loss: 0.0345

Epoch 13/100
53/53 [=====] - 1s 20ms/step - loss: 0.0034 - val_loss: 0.0348

Epoch 14/100
53/53 [=====] - 1s 17ms/step - loss: 0.0034 - val_loss: 0.0345

Epoch 15/100
53/53 [=====] - 1s 20ms/step - loss: 0.0033 - val_loss: 0.0345

Epoch 16/100
53/53 [=====] - 1s 15ms/step - loss: 0.0031 - val_loss: 0.0346

Epoch 17/100
53/53 [=====] - 1s 15ms/step - loss: 0.0030 - val_loss: 0.0352

Epoch 18/100
53/53 [=====] - 1s 25ms/step - loss: 0.0031 - val_loss: 0.0347

Epoch 19/100
53/53 [=====] - 1s 24ms/step - loss: 0.0032 - val_loss: 0.0348

Epoch 20/100
53/53 [=====] - 1s 22ms/step - loss: 0.0030 - val_loss: 0.0348

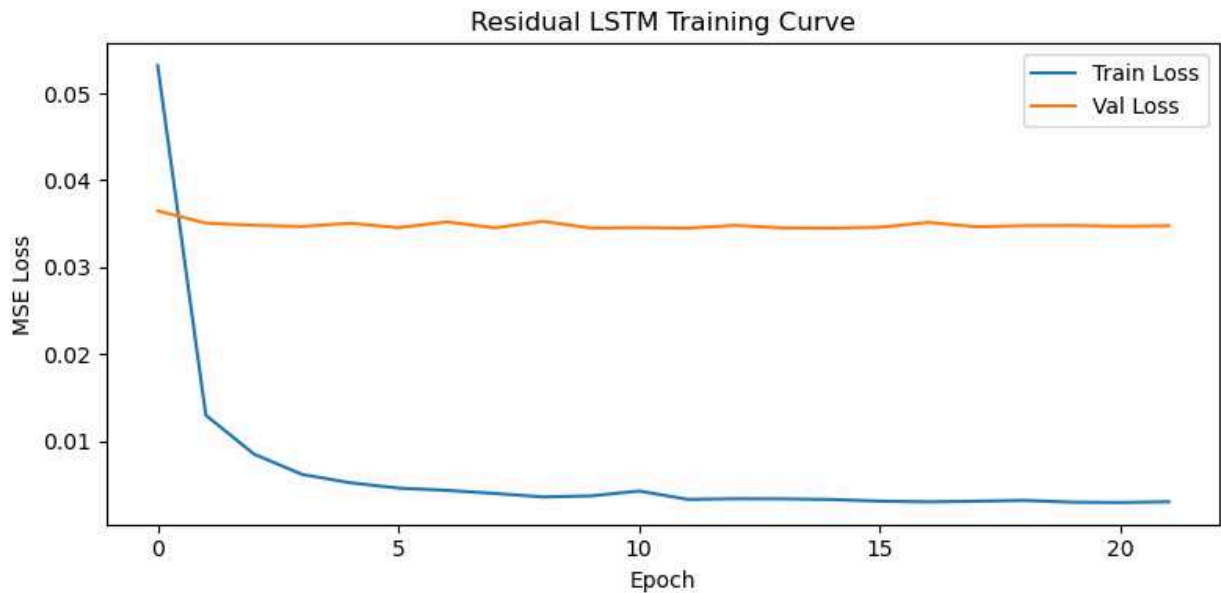
Epoch 21/100

53/53 [=====] - 1s 19ms/step - loss: 0.0030 - val_loss: 0.0347

Epoch 22/100

53/53 [=====] - 1s 23ms/step - loss: 0.0031 - val_loss: 0.0348

```
In [27]: # Plot residual LSTM learning curve
plt.figure(figsize=(8,4))
plt.plot(history_res.history['loss'], label='Train Loss')
plt.plot(history_res.history['val_loss'], label='Val Loss')
plt.title("Residual LSTM Training Curve")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.legend()
plt.tight_layout()
plt.show()
```



```
In [28]: # Predict residuals on test set
residual_pred_s = res_lstm.predict(X_test_res)
residual_pred = scaler_res.inverse_transform(residual_pred_s).flatten()

# Align XGBoost returns with residual predictions
xgb_ret_aligned = y_pred_xgb_aligned

# Final hybrid return = XGBoost + Residual LSTM
hybrid_return = xgb_ret_aligned + residual_pred
```

12/12 [=====] - 0s 5ms/step

```
In [29]: # Reconstruct prices from hybrid returns
actual_prices_aligned = test_df['Gold_Price'].values[window_size:]

clip_min, clip_max = -0.03, 0.03
hybrid_ret_clip = np.clip(hybrid_return, clip_min, clip_max)
xgb_ret_clip = np.clip(xgb_ret_aligned, clip_min, clip_max)

true_prices_final = actual_prices_aligned[1:]
```

```
hybrid_price_pred = actual_prices_aligned[:-1] * np.exp(hybrid_ret_clip[1:])
xgb_price_pred = actual_prices_aligned[:-1] * np.exp(xgb_ret_clip[1:])
```

```
In [31]: import numpy as np

def regression_metrics(y_true, y_pred, label="Model"):
    y_true = np.asarray(y_true).reshape(-1)
    y_pred = np.asarray(y_pred).reshape(-1)

    rmse = np.sqrt(np.mean((y_true - y_pred) ** 2))
    mae = np.mean(np.abs(y_true - y_pred))

    print(f"{label} -> RMSE: {rmse:.4f}, MAE: {mae:.4f}")
    return {"rmse": rmse, "mae": mae}

def directional_accuracy(y_true, y_pred):
    y_true = np.asarray(y_true).reshape(-1)
    y_pred = np.asarray(y_pred).reshape(-1)

    # Compare direction of change (t vs t-1)
    true_dir = np.sign(y_true[1:] - y_true[:-1])
    pred_dir = np.sign(y_pred[1:] - y_pred[:-1])

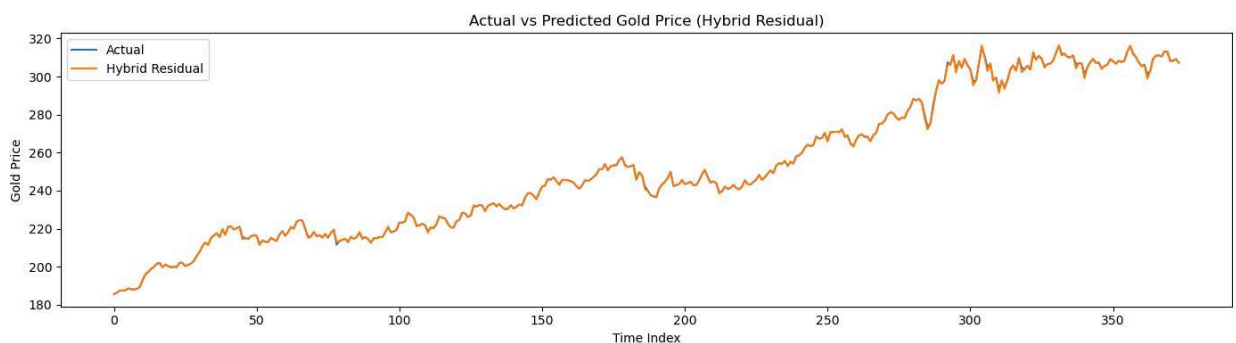
    return float(np.mean(true_dir == pred_dir))
```

```
In [32]: # Final evaluation on price scale (no USD scaling)
regression_metrics(true_prices_final, xgb_price_pred, "XGBoost Final")
regression_metrics(true_prices_final, hybrid_price_pred, "Hybrid Residual Final")

print("DA (XGBoost):", directional_accuracy(true_prices_final, xgb_price_pred))
print("DA (Hybrid Residual):", directional_accuracy(true_prices_final, hybrid_price_pred))

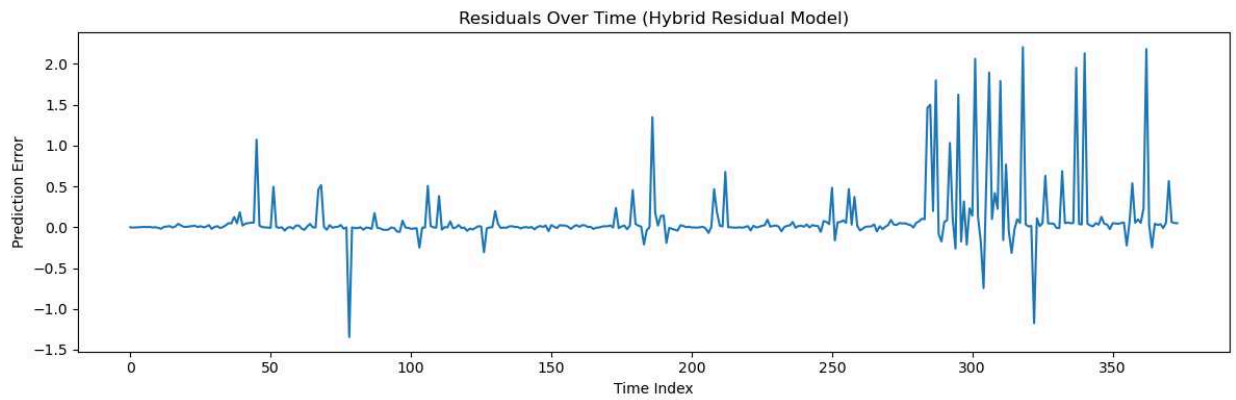
XGBoost Final -> RMSE: 0.3723, MAE: 0.1261
Hybrid Residual Final -> RMSE: 0.3800, MAE: 0.1350
DA (XGBoost): 0.9865951742627346
DA (Hybrid Residual): 0.9865951742627346
```

```
In [33]: # Plot actual vs predicted prices (Hybrid)
plt.figure(figsize=(14,4))
plt.plot(true_prices_final, label='Actual')
plt.plot(hybrid_price_pred, label='Hybrid Residual')
plt.title("Actual vs Predicted Gold Price (Hybrid Residual)")
plt.xlabel("Time Index")
plt.ylabel("Gold Price")
plt.legend()
plt.tight_layout()
plt.show()
```



```
In [34]: # Plot residuals over time
residual_price = true_prices_final - hybrid_price_pred

plt.figure(figsize=(12,4))
plt.plot(residual_price)
plt.title("Residuals Over Time (Hybrid Residual Model)")
plt.xlabel("Time Index")
plt.ylabel("Prediction Error")
plt.tight_layout()
plt.show()
```



```
In [ ]:
```