In [16]:
```python
# Cell 1: Imports and configuration
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

plt.style.use('seaborn-v0_8')
sns.set_context('talk')

DATA_PATH = '../data/raw/XAU_15m_data.csv'  # change to your path (e.g., data/raw/gold.csv)
DT_COL_CANDIDATES = ['datetime', 'time', 'timestamp', 'Date', 'date']
PRICE_COL_CANDIDATES = ['close', 'Close', 'PRICE', 'Price']

# Utility to find datetime and price columns
def detect_columns(df):
    dt_col = None
    for c in DT_COL_CANDIDATES:
        if c in df.columns:
            dt_col = c
            break
    if dt_col is None:
        raise ValueError(f"Could not detect datetime column. Checked: {DT_COL_CANDIDATES}")

    price_col = None
    for c in PRICE_COL_CANDIDATES:
        if c in df.columns:
            price_col = c
            break
    if price_col is None:
        # fallback if there's a typical OHLC layout
        for c in ['close', 'Close']:
            if c in df.columns:
                price_col = c
                break
    if price_col is None:
        raise ValueError(f"Could not detect price column. Checked: {PRICE_COL_CANDIDATES}")

    return dt_col, price_col
```

In [35]:
```python
import pandas as pd

raw = pd.read_csv(DATA_PATH, sep=";", low_memory=False)
print(raw.columns.tolist())
```

```
['Date', 'Open', 'High', 'Low', 'Close', 'Volume']
```

In [58]:
```python
# Cell 2: Load data (robust parsing for large CSV)
chunksize = None  # set to an int like 200_000 if the file is very large

if chunksize:
    chunks = []
    for ch in pd.read_csv(DATA_PATH, sep=";", low_memory=False):
        chunks.append(ch)
    raw = pd.concat(chunks, ignore_index=True)
else:
    raw = pd.read_csv(DATA_PATH,  sep=";", low_memory=False)


dt_col, price_col = detect_columns(raw)
# Explicitly set your column names
#raw.columns = raw.columns.str.strip()   # removes leading/trailing spaces
dt_col = "date"
price_col = "close"

raw.columns = raw.columns.str.strip().str.lower()
#print(raw.columns.tolist())
#Parse datetime, sort, set index
raw[dt_col] = pd.to_datetime(raw['date'], errors='coerce', utc=True)

df = raw.dropna(subset=[dt_col]).sort_values(dt_col).set_index(dt_col)

df = raw.dropna(subset=[dt_col]).sort_values(dt_col).set_index(dt_col)

# Keep common OHLCV if present
cols = [c for c in ['open','high','low','close','volume','Open','High','Low','Close','Volume'] if c in df.columns]
if len(cols) == 0:
    # ensure we at least have price column
    cols = [price_col]
df = df[cols].copy()

# Standardize column names to lowercase for consistency
df.columns = [c.lower() for c in df.columns]
if price_col.lower() not in df.columns:
    # if price exists under close, map it
    if 'close' in df.columns:
        price_col = 'close'
    else:
        price_col = df.columns[0]
```

In [59]:
```python
# Cell 3: Basic cleaning and summary
# Remove duplicate timestamps and non-numeric entries
df = df[~df.index.duplicated(keep='last')]
for c in df.columns:
    df[c] = pd.to_numeric(df[c], errors='coerce')
df = df.dropna()

# Basic overview
print("Shape:", df.shape)
print("\nHead:\n", df.head())
print("\nTail:\n", df.tail())
print("\nSummary stats:\n", df.describe())

# Frequency inference (15-minute expected)
print("\nInferred frequency:", pd.infer_freq(df.index[:200]))  # rough check
```

```
Shape: (485968, 5)

Head:
                            open   high    low  close  volume
date
2004-06-11 07:15:00+00:00  384.0  384.3  383.8  384.3      12
2004-06-11 07:30:00+00:00  383.8  384.3  383.6  383.8      12
2004-06-11 07:45:00+00:00  383.3  383.8  383.3  383.8      20
```

```
2004-06-11 08:00:00+00:00  383.8  384.1  383.6  383.6       8
2004-06-11 08:15:00+00:00  383.6  384.3  383.5  383.5      20

Tail:
                              open     high      low    close  volume
date
2025-10-01 04:30:00+00:00  3864.86  3866.20  3863.59  3865.92    2496
2025-10-01 04:45:00+00:00  3865.93  3868.91  3865.45  3868.81    3277
2025-10-01 05:00:00+00:00  3868.79  3875.57  3867.30  3867.52    7364
2025-10-01 05:15:00+00:00  3867.64  3870.65  3864.68  3866.16    4483
2025-10-01 05:30:00+00:00  3866.19  3866.19  3864.94  3865.88     122

Summary stats:
                 open           high            low          close  \
count  485968.000000  485968.000000  485968.000000  485968.000000
mean     1382.059356    1382.959692    1381.138505    1382.061666
std       582.917032     583.273915     582.552627     582.927162
min       381.300000     381.600000     381.100000     381.300000
25%      1074.417500    1075.110000    1073.660000    1074.407500
50%      1297.900000    1298.520000    1297.220000    1297.900000
75%      1736.700000    1737.860000    1735.502500    1736.710000
max      3869.650000    3875.570000    3867.300000    3869.820000

              volume
count  485968.000000
mean      922.968749
std      1300.473617
min         1.000000
25%       188.000000
50%       617.000000
75%      1193.000000
max     64585.000000

Inferred frequency: None
```

In [60]:

```python
# Cell 4: Feature engineering (returns, intraday/weekly keys)
# Log returns from price column
df['ret'] = df[price_col].pct_change()
df['log_ret'] = np.log1p(df['ret'])

# Rolling volatility (~1 day if 15-min bars: 96 intervals)
WINDOW_1D = 96
df['vol_rolling'] = df['log_ret'].rolling(WINDOW_1D).std()

# Calendar features for pivot/heatmap/boxplots
df['hour'] = df.index.hour
df['dow'] = df.index.dayofweek  # 0=Mon, 6=Sun
df['year'] = df.index.year
```

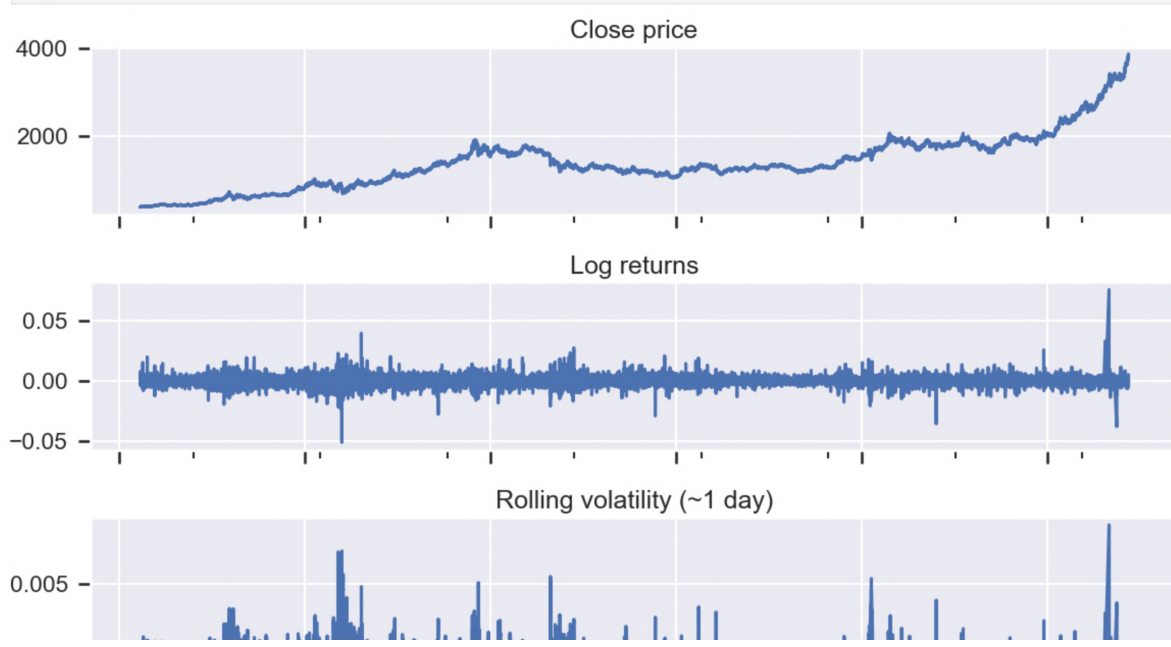
In [63]:

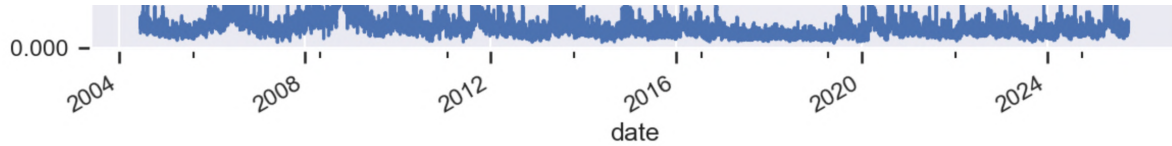
```python
# Cell 5: Yearly changes (positive/negative)
yearly_last = df.resample('YE')[price_col].last()
yearly_pct = yearly_last.pct_change()
yearly_df = pd.DataFrame({
    'year': yearly_pct.index.year,
    'pct_change': yearly_pct.values,
    'direction': np.where(yearly_pct > 0, 'positive', 'negative')
}).dropna()

print(yearly_df.head(10))
```

```
   year  pct_change direction
1  2005    0.177574  positive
2  2006    0.236300  positive
3  2007    0.307450  positive
4  2008    0.042077  positive
5  2009    0.266163  positive
6  2010    0.283022  positive
7  2011    0.110771  positive
8  2012    0.070813  positive
9  2013   -0.278595  negative
10 2014   -0.017736  negative
```

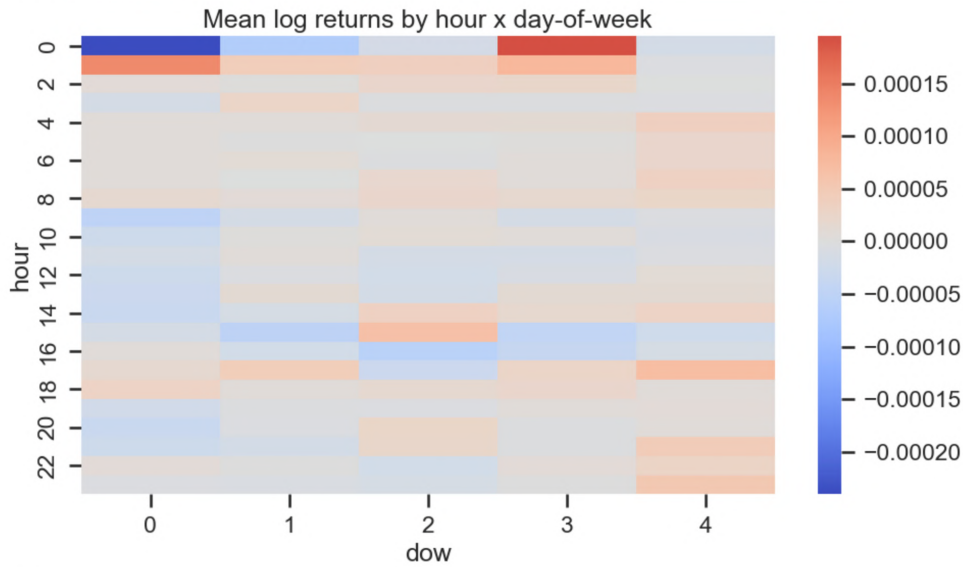
In [64]:

```python
# Cell 6: Overview plots (tight_layout)
fig, axes = plt.subplots(3, 1, figsize=(12, 8), sharex=True)
df[price_col].plot(ax=axes[0], title='Close price')
df['log_ret'].plot(ax=axes[1], title='Log returns')
df['vol_rolling'].plot(ax=axes[2], title='Rolling volatility (~1 day)')
plt.tight_layout()
plt.show()
```



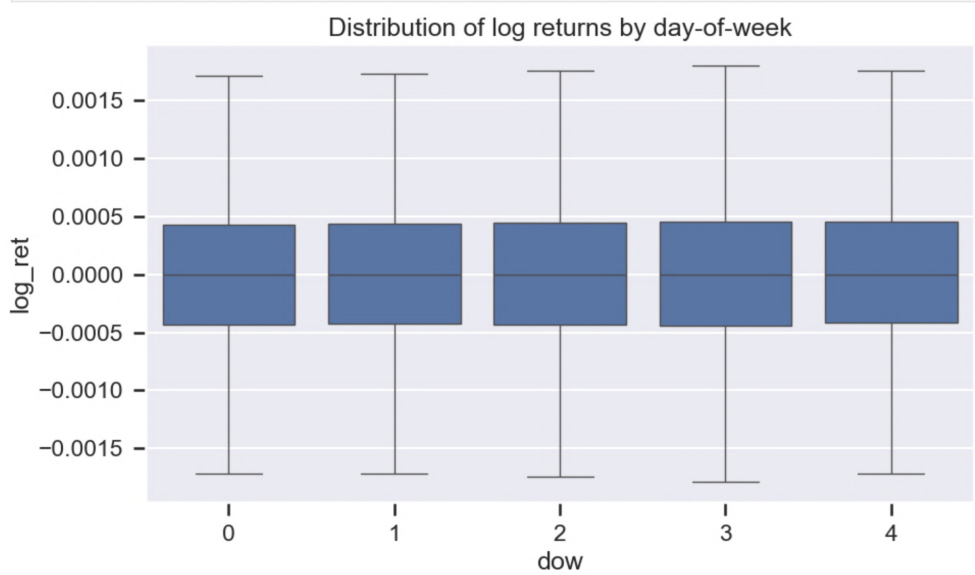


In [65]:
```
# Cell 7: Pivot table and heatmap (hour x day-of-week)
pv = df.pivot_table(index='hour', columns='dow', values='log_ret', aggfunc='mean')
plt.figure(figsize=(10, 6))
sns.heatmap(pv, cmap='coolwarm', center=0)
plt.title('Mean log returns by hour x day-of-week')
plt.tight_layout()
plt.show()

print("Pivot sample:\n", pv.round(6).head())
```

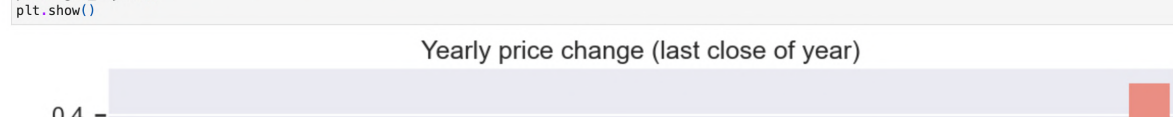



```
Pivot sample:
 dow          0         1         2         3         4
hour
0    -0.000240 -0.000067 -0.000017  0.000195 -0.000018
1     0.000137  0.000041  0.000037  0.000078 -0.000005
2     0.000009  0.000002  0.000023  0.000021 -0.000002
3    -0.000013  0.000028 -0.000003 -0.000003 -0.000005
4     0.000004  0.000006  0.000014  0.000012  0.000037
```

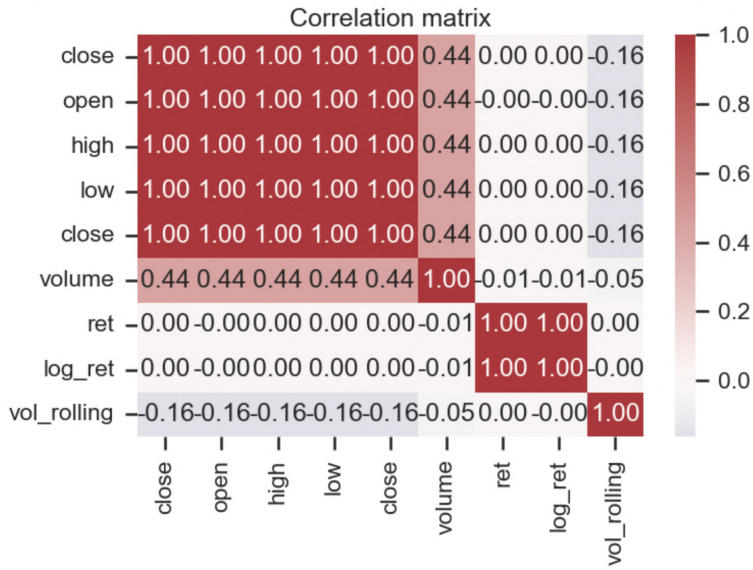
In [66]:
```
# Cell 8: Boxplot by day-of-week
plt.figure(figsize=(10, 6))
sns.boxplot(x='dow', y='log_ret', data=df, showfliers=False)
plt.title('Distribution of log returns by day-of-week')
plt.tight_layout()
plt.show()
```



In [67]:
```
# Cell 9: Bar chart of yearly price changes
plt.figure(figsize=(12, 5))
sns.barplot(x='year', y='pct_change', data=yearly_df, hue='direction', dodge=False, palette=['salmon', 'steelblue'])
plt.axhline(0, color='black', linewidth=1)
plt.title('Yearly price change (last close of year)')
plt.tight_layout()
plt.show()
```



In [68]:
```
# Cell 10: Correlation matrix (OHLCV + engineered features)
corr_cols = [c for c in [price_col, 'open','high','low','close','volume','ret','log_ret','vol_rolling'] if c in df.columns]
corr = df[corr_cols].corr()
plt.figure(figsize=(8, 6))
sns.heatmap(corr, annot=True, fmt=".2f", cmap='vlag', center=0)
plt.title('Correlation matrix')
plt.tight_layout()
plt.show()
print("\nCorrelation matrix:\n", corr.round(3))
```

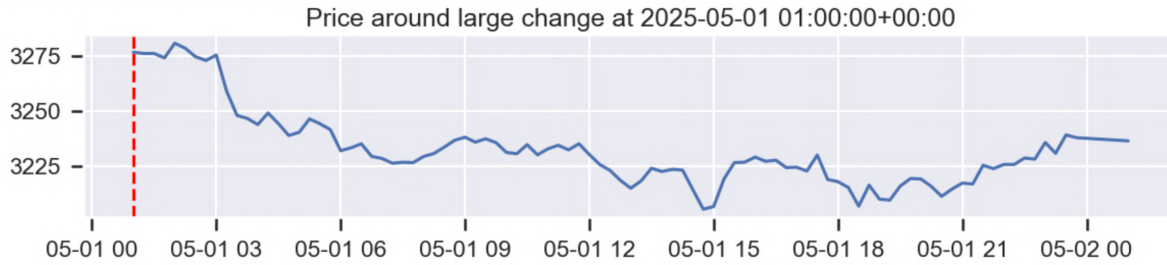


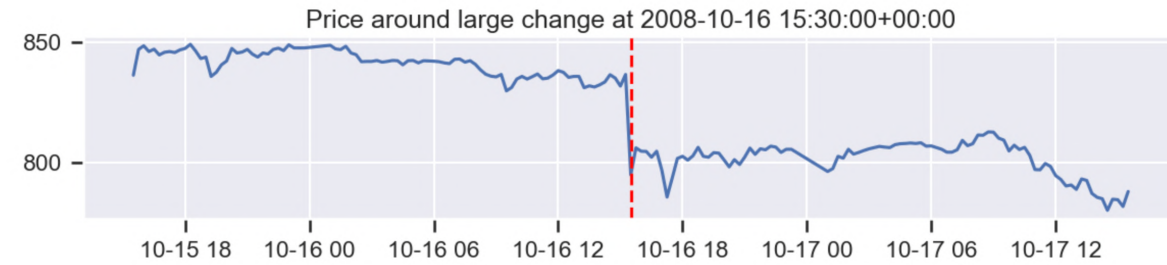

```
Correlation matrix:
               close   open   high    low  close  volume    ret  log_ret  \
close          1.000  1.000  1.000  1.000  1.000   0.439  0.002    0.002
open           1.000  1.000  1.000  1.000  1.000   0.439 -0.000   -0.000
high           1.000  1.000  1.000  1.000  1.000   0.439  0.001    0.001
low            1.000  1.000  1.000  1.000  1.000   0.438  0.001    0.001
close          1.000  1.000  1.000  1.000  1.000   0.439  0.002    0.002
volume         0.439  0.439  0.439  0.438  0.439   1.000 -0.007   -0.007
ret            0.002 -0.000  0.001  0.001  0.002  -0.007  1.000    1.000
log_ret        0.002 -0.000  0.001  0.001  0.002  -0.007  1.000    1.000
vol_rolling   -0.162 -0.162 -0.161 -0.163 -0.162  -0.047  0.000   -0.000

              vol_rolling
close              -0.162
open               -0.162
high               -0.161
low                -0.163
close              -0.162
volume             -0.047
ret                 0.000
log_ret            -0.000
vol_rolling         1.000
```

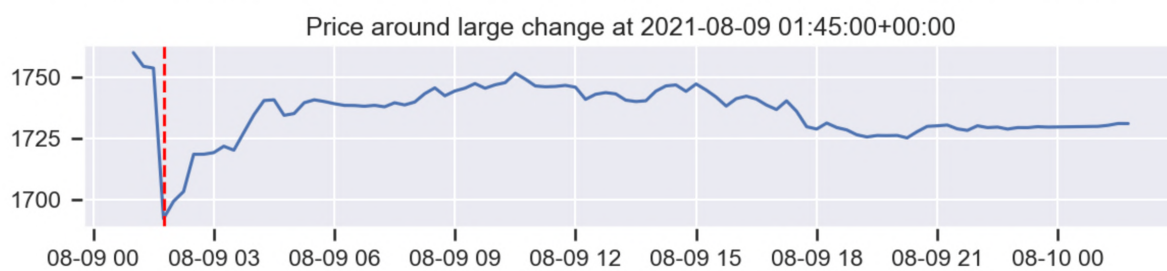
In [69]:
```
# Cell 11: Duration with drastic changes (top-N absolute returns)
N = 25
top_moves = df[['log_ret']].copy()
top_moves['abs_log_ret'] = top_moves['log_ret'].abs()
big_changes = top_moves.sort_values('abs_log_ret', ascending=False).head(N)
print(f"\nTop {N} drastic changes (timestamps and magnitudes):\n", big_changes[['log_ret']])

# Visualize a window around a few extremes
k = 5
timestamps = big_changes.index[:k]
WINDOW = 96  # ~1 day context
fig, axes = plt.subplots(k, 1, figsize=(12, 2*k+4), sharex=False)
for i, ts in enumerate(timestamps):
    w = df.loc[(df.index >= ts - pd.Timedelta(minutes=15*WINDOW)) & (df.index <= ts + pd.Timedelta(minutes=15*WINDOW)), price_col]
    axes[i].plot(w.index, w.values)
    axes[i].axvline(ts, color='red', linestyle='--')
    axes[i].set_title(f'Price around large change at {ts}')
plt.tight_layout()
plt.show()
```

```
Top 25 drastic changes (timestamps and magnitudes):
                              log_ret
date
2025-05-01 01:00:00+00:00  0.075517
2008-10-16 15:30:00+00:00 -0.051010
2009-03-18 19:15:00+00:00  0.039458
```

```
2025-07-01 01:00:00+00:00 -0.037984
2021-08-09 01:45:00+00:00 -0.035594
2025-04-01 01:00:00+00:00  0.032926
2015-07-20 04:15:00+00:00 -0.029015
2010-11-15 01:00:00+00:00 -0.027740
2013-10-17 10:45:00+00:00  0.027297
2023-12-04 01:15:00+00:00  0.025652
2008-10-13 15:45:00+00:00 -0.023794
2013-09-18 21:00:00+00:00  0.023294
2008-09-17 15:30:00+00:00  0.022763
2008-11-21 16:30:00+00:00  0.021897
2008-09-18 21:30:00+00:00 -0.021858
2013-04-15 04:45:00+00:00 -0.020961
2020-03-06 17:15:00+00:00 -0.020654
2015-10-02 15:30:00+00:00  0.020580
2008-09-17 16:00:00+00:00  0.020378
2009-11-04 01:00:00+00:00  0.020100
2008-10-10 19:15:00+00:00 -0.019966
2004-08-06 15:00:00+00:00  0.019737
2006-04-03 01:00:00+00:00 -0.019556
2006-11-27 01:00:00+00:00  0.019400
2006-09-28 01:00:00+00:00  0.019265
```







In [ ]:

Gold time series analysis daily and monthly

This single notebook is self-contained and uses attached monthly CSV. It includes clear, step-by-step explanations and code cells for (1) trend/seasonality via ACF/PACF and STL, (2) long-term cycle, regime and volatility analysis, and (3) baseline models (SARIMA and Holt–Winters) with benchmarking. It also supports daily data if you provide the daily CSV locally.

Setup and imports

In [1]:
```
# Cell: imports and settings
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import STL
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.holtwinters import ExponentialSmoothing

from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import ElasticNet
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

plt.style.use("seaborn-v0_8")
pd.set_option("display.float_format", lambda v: f"{v:,.4f}")
```

Data loading

This cell is responsible for importing the raw gold price data from CSV files (monthly and daily) and preparing it for analysis. It defines helper functions to ensure the datasets are consistently formatted and ready for time series modeling:

File paths: MONTHLY_PATH and DAILY_PATH specify the locations of the monthly and daily gold CSV files. The daily path can be updated if the file is stored elsewhere.

_standardize(df): A utility function that cleans and standardizes the dataset by:

Converting column names to lowercase for consistency.

Setting the date column as the index and sorting chronologically.

Converting numeric fields (open, high, low, close, volume) to proper numeric types, coercing invalid entries to NaN.

Dropping rows that are completely empty.

load_monthly(path): Reads the monthly CSV file using a semicolon separator, parses the Date column into datetime objects, and applies _standardize to return a clean DataFrame.

load_daily(path): Similar to load_monthly, but designed for daily data files. It also parses dates and standardizes the structure.

Loading and validation:

The monthly dataset is loaded into df_m, and the close price series (close_m) is extracted. The code prints the number of rows and the date range to confirm successful loading.

For daily data, the code checks if the file exists. If found, it loads the dataset into df_d, extracts the close series (close_d), and prints summary information. If not found, it prints a message indicating that daily analysis will be skipped.

This preprocessing step ensures that both monthly and daily datasets are clean, consistent, and indexed by date, which is essential for subsequent time series analysis (ACF/PACF, regime detection, and forecasting models).

In [2]:
```
# Cell: data loaders
MONTHLY_PATH = "./XAU_1Month_data.csv"
DAILY_PATH = "./XAU_1d_data.csv"  # change this if your daily file is elsewhere

def _standardize(df: pd.DataFrame) -> pd.DataFrame:
    """
    Standardize columns: lowercase names, datetime index, numeric conversion.
    """
    df.columns = [c.strip().lower() for c in df.columns]
    df = df.set_index("date").sort_index()
    for c in ["open", "high", "low", "close", "volume"]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce")
    df = df.dropna(how="all")
    return df

def load_monthly(path: str) -> pd.DataFrame:
    """
    Load monthly gold CSV with semicolon separator; parses Date column.
    """
    df = pd.read_csv(path, sep=";", parse_dates=["Date"], dayfirst=False)
    return _standardize(df)

def load_daily(path: str) -> pd.DataFrame:
    """
    Load daily gold CSV (semicolon separator by default). Adjust if needed.
    """
    df = pd.read_csv(path, sep=";", parse_dates=["Date"], dayfirst=False)
    return _standardize(df)

# Load monthly
df_m = load_monthly(MONTHLY_PATH)
close_m = df_m["close"].dropna()
print("Monthly rows:", len(close_m), "range:", close_m.index.min(), "→", close_m.index.max())

# Try to load daily; if not available, skip
df_d, close_d = None, None
if os.path.exists(DAILY_PATH):
    df_d = load_daily(DAILY_PATH)
    close_d = df_d["close"].dropna()
    print("Daily rows:", len(close_d), "range:", close_d.index.min(), "→", close_d.index.max())
else:
    print("Daily file not found at", DAILY_PATH, "- daily analysis will be skipped.")
```

```
Monthly rows: 257 range: 2004-06-01 00:00:00 → 2025-12-01 00:00:00
Daily rows: 5462 range: 2004-06-11 00:00:00 → 2025-12-01 00:00:00
```

Helper functions for Stationarity, Diagnostics, and Forecast Evaluation

This cell defines a set of utility functions that support statistical diagnostics and model benchmarking in the gold price forecasting workflow:

prepare_stationary(series, use_log=True, diff_order=1): Transforms a raw time series into a stationary form, which is required for reliable autocorrelation analysis and ARIMA-type models. It optionally applies a logarithmic transformation to stabilize variance and differencing to remove trends.

acf_pacf_plots(series, max_lag, title_prefix): Generates side-by-side plots of the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF). These plots help identify lag structures, seasonal patterns, and guide the selection of ARIMA model orders.

stl_decompose(series, period, title): Performs Seasonal-Trend decomposition using Loess (STL). This separates the series into trend, seasonal, and residual components, allowing clear visualization of long-term cycles and recurring seasonal effects. The period parameter is set according to data frequency (e.g., 12 for monthly, 365 or 7 for daily).

mape(y_true, y_pred): Calculates the Mean Absolute Percentage Error (MAPE), a scale-independent metric that expresses forecast error as a percentage of actual values.

evaluate_forecast(test, pred): Computes three key accuracy metrics — Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and MAPE — to benchmark model performance. These metrics provide complementary views of forecast accuracy, balancing absolute error, squared error sensitivity, and percentage error.

Together, these helpers streamline the analysis pipeline by ensuring data is properly prepared for statistical tests, enabling clear visualization of autocorrelation and seasonality, and providing standardized metrics for comparing forecasting models.

In [3]:
```
# Cell: helpers for stationarity, plots, metrics
def prepare_stationary(series: pd.Series, use_log: bool = True, diff_order: int = 1) -> pd.Series:
    """
    Transform series toward stationarity via optional log and differencing.
    Use this for ACF/PACF where stationarity is needed.
    """
    s = series.copy()
    if use_log:
        s = np.log(s)
    for _ in range(diff_order):
        s = s.diff()
    return s

def acf_pacf_plots(series: pd.Series, max_lag: int, title_prefix: str = ""):
    """
    Plot ACF and PACF for a (preferably stationary) series.
    """
    s = series.dropna()
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))
    plot_acf(s, lags=max_lag, ax=axes[0])
    plot_pacf(s, lags=max_lag, ax=axes[1], method="ywm")
    axes[0].set_title(f"{title_prefix} ACF")
    axes[1].set_title(f"{title_prefix} PACF")
    plt.tight_layout()
    plt.show()

def stl_decompose(series: pd.Series, period: int, title: str):
    """
    STL decomposition to visualize trend and seasonality. Choose period per frequency.
    - Monthly: 12
    - Daily: 365 (yearly), or 7 (weekly), depending on hypothesis
    """
    s = series.dropna()
    res = STL(s, period=period, robust=True).fit()
    fig = res.plot()
    fig.set_size_inches(10, 6)
    fig.suptitle(title, fontsize=12)
    plt.tight_layout()
    plt.show()
    return res

def mape(y_true: pd.Series, y_pred: pd.Series) -> float:
    """
    Mean Absolute Percentage Error in percent.
    """
    y_true = np.array(y_true, dtype=float)
    y_pred = np.array(y_pred, dtype=float)
    eps = 1e-8
    return np.mean(np.abs((y_true - y_pred) / (y_true + eps))) * 100.0

def evaluate_forecast(test: pd.Series, pred: pd.Series) -> dict:
    """
    Compute MAE, RMSE, and MAPE.
    """
    mae = mean_absolute_error(test, pred)
    rmse = np.sqrt(np.mean((np.array(test) - np.array(pred))**2))
    mp = mape(test, pred)
    return {"MAE": mae, "RMSE": rmse, "MAPE": mp}
```

1. Trend and seasonality analysis using ACF/PACF

This cell performs statistical diagnostics to uncover the underlying structure of the monthly gold price series:

STL decomposition (Seasonal-Trend decomposition using Loess): The series is decomposed into three components — trend, seasonality, and residuals.

The period=12 parameter reflects yearly seasonality in monthly data (12 months per cycle).

The trend component highlights long-term upward or downward movements.

The seasonal component captures recurring annual patterns.

The residual component isolates irregular fluctuations not explained by trend or seasonality.

Stationarity preparation: To make the series suitable for autocorrelation analysis, the closing prices are log-transformed (to stabilize variance) and differenced once (to remove trend). This produces a stationary series (close_m_stat).

ACF and PACF plots:

The Autocorrelation Function (ACF) shows correlations between the series and its own lagged values, helping detect repeating cycles or seasonal dependencies.

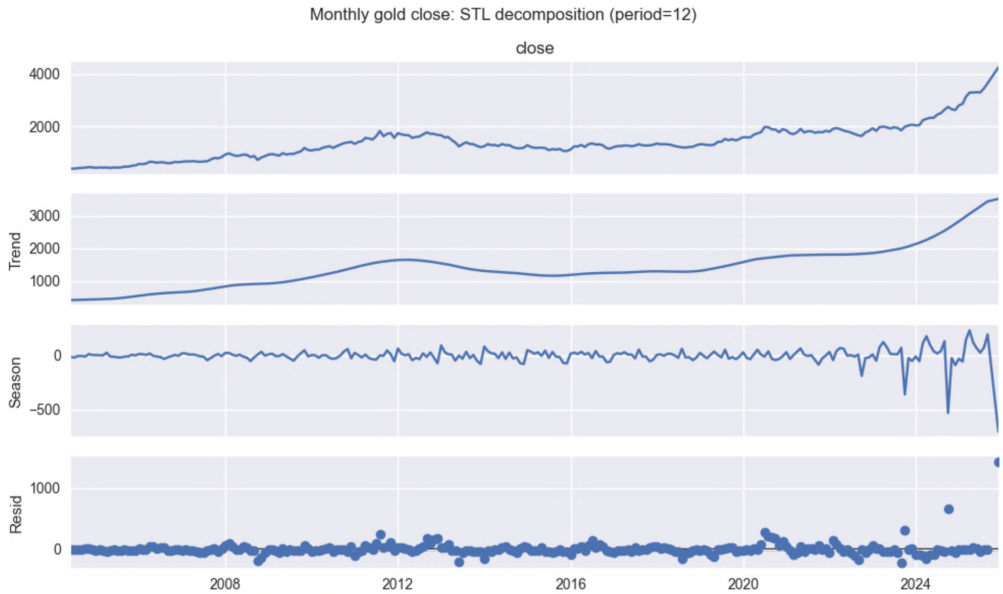
The Partial Autocorrelation Function (PACF) isolates the direct correlation at each lag, guiding the choice of autoregressive (AR) terms in ARIMA/SARIMA models.

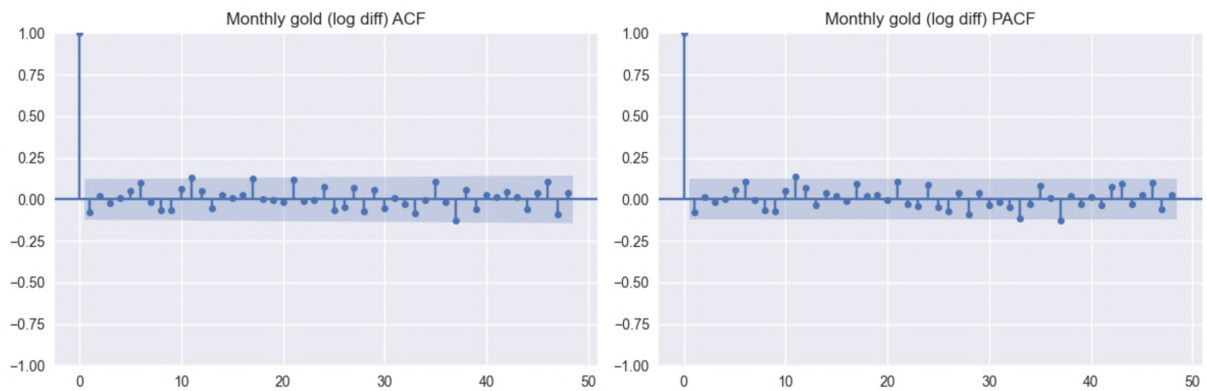
Here, lags up to 48 months (~4 years) are plotted to reveal both short-term and longer seasonal dependencies.

Together, these diagnostics provide a clear view of the gold market's cyclical behavior, seasonal influences, and the lag structure needed for building robust forecasting models.

In [4]:
```
# Cell: monthly STL and ACF/PACF
# STL: yearly seasonality expected in monthly data (period=12)
  = stl_decompose(close_m, period=12, title="Monthly gold close: STL decomposition (period=12)")
```

```
# ACF/PACF: use log-differenced series for stationarity
close_m_stat = prepare_stationary(close_m, use_log=True, diff_order=1)
acf_pacf_plots(close_m_stat, max_lag=48, title_prefix="Monthly gold (log diff)")
```





This cell examines the daily gold price series (if available) to uncover cyclical patterns and lag dependencies that inform forecasting model design:

STL decomposition (Seasonal-Trend decomposition using Loess):

The series is decomposed into trend, seasonal, and residual components.

A yearly period (period=365) is specified to capture annual seasonality in daily data.

The trend component highlights long-term upward or downward movements.

The seasonal component reveals recurring yearly cycles.

The residual captures irregular fluctuations not explained by trend or seasonality.

Stationarity preparation:

The raw daily closing prices are log-transformed to stabilize variance.

A first difference is applied to remove trend and achieve approximate stationarity.

This stationary series (close_d_stat) is suitable for autocorrelation analysis.

ACF and PACF plots:

The Autocorrelation Function (ACF) shows correlations between the series and its lagged values, helping detect repeating cycles or seasonal dependencies.

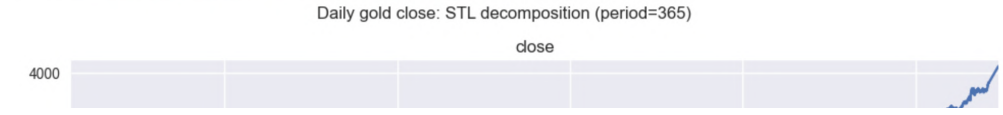
The Partial Autocorrelation Function (PACF) isolates direct correlations at each lag, guiding the choice of autoregressive (AR) terms in ARIMA/SARIMA models.

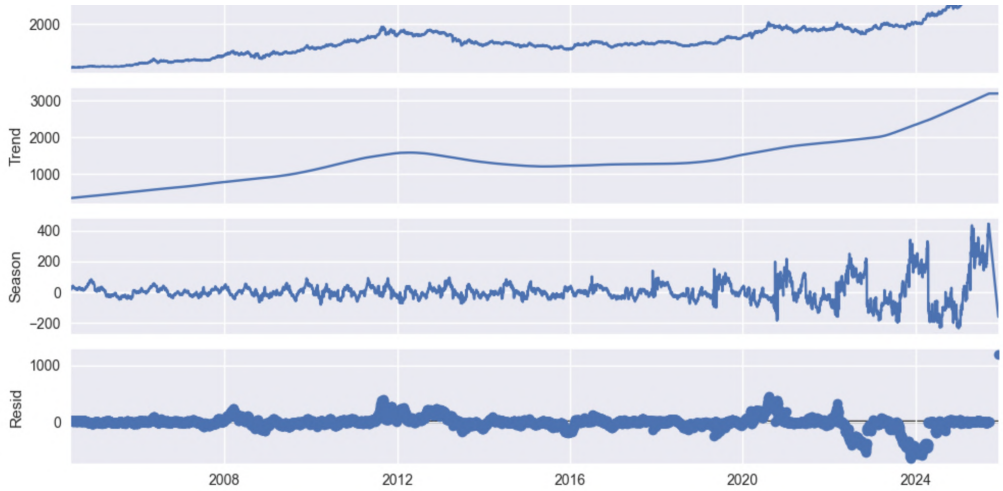
Lags up to 90 days (~3 months) are plotted to reveal both short-term and medium-term dependencies.

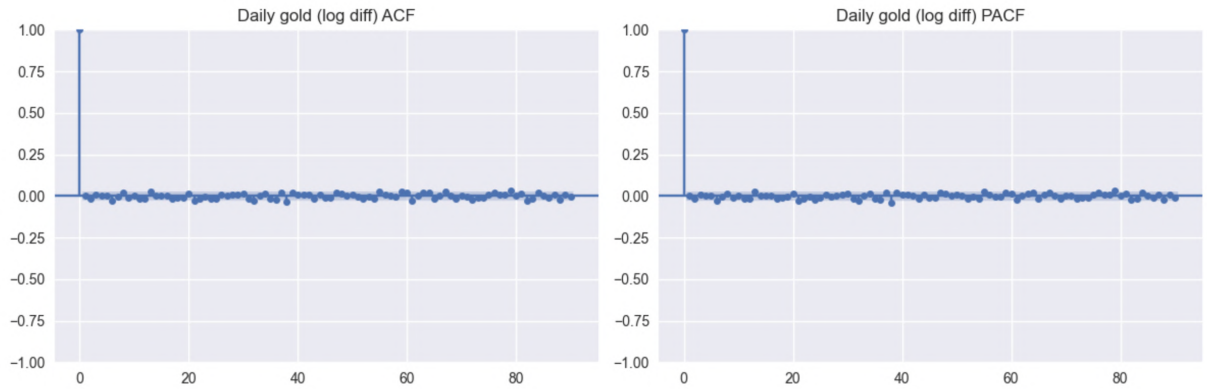
Conditional execution: If the daily dataset is not found, the analysis is skipped gracefully with a message, ensuring the notebook remains robust.

This diagnostic step provides a clear view of daily gold price behavior, highlighting annual seasonality, short-term autocorrelation, and the lag structure needed for building accurate forecasting models.

In [5]:
```
# Cell: daily STL and ACF/PACF (skip if daily file missing)
if close_d is not None:
    # Consider yearly seasonality (365) or weekly (7); we visualize yearly first
    _ = stl_decompose(close_d, period=365, title="Daily gold close: STL decomposition (period=365)")
    close_d_stat = prepare_stationary(close_d, use_log=True, diff_order=1)
    acf_pacf_plots(close_d_stat, max_lag=90, title_prefix="Daily gold (log diff)")
else:
    print("Daily analysis skipped: no daily dataset loaded.")
```







2. Long-term cycles, regime shifts, and volatility patterns

We examine slow-moving trend regimes with rolling means, dispersion changes with rolling std, and volatility clustering via realized volatility from log returns. We also flag extremes with rolling z-scores as a simple regime indicator.

This cell defines and applies utilities to study long-term cycles, regime shifts, and volatility clustering in the gold price series:

rolling_stats(series, windows): Computes rolling means and rolling standard deviations over specified windows.

Rolling means act as proxies for regime trends, showing how the average price evolves over time.

Rolling standard deviations highlight changes in dispersion, useful for spotting volatility regimes.

Suggested windows:

Monthly data → 12 and 24 (1 year, 2 years).

Daily data → 30 and 90 (1 month, 3 months).

realized_vol(series, window): Estimates realized volatility by calculating the rolling standard deviation of log returns, scaled by the square root of the window length. This captures volatility clustering, where periods of high volatility tend to follow each other.

plot_regime_vol(series, windows, title): Produces a composite visualization with three panels:

Price level (raw closing prices).

Rolling means (regime proxies) to highlight long-term cycles.

Rolling std and realized volatility to reveal volatility patterns and clusters. This allows simultaneous inspection of price trends, regime shifts, and volatility dynamics.

zscore_regimes(series, window, threshold): Calculates rolling z-scores to flag outsized deviations from the local mean.

A regime_flag is set to 1 when the absolute z-score exceeds the threshold (default = 1.0).

This provides a simple indicator of potential regime shifts or extreme events relative to recent history.

Application to monthly data: The functions are applied to the monthly gold closing price series with 12- and 24-month windows.

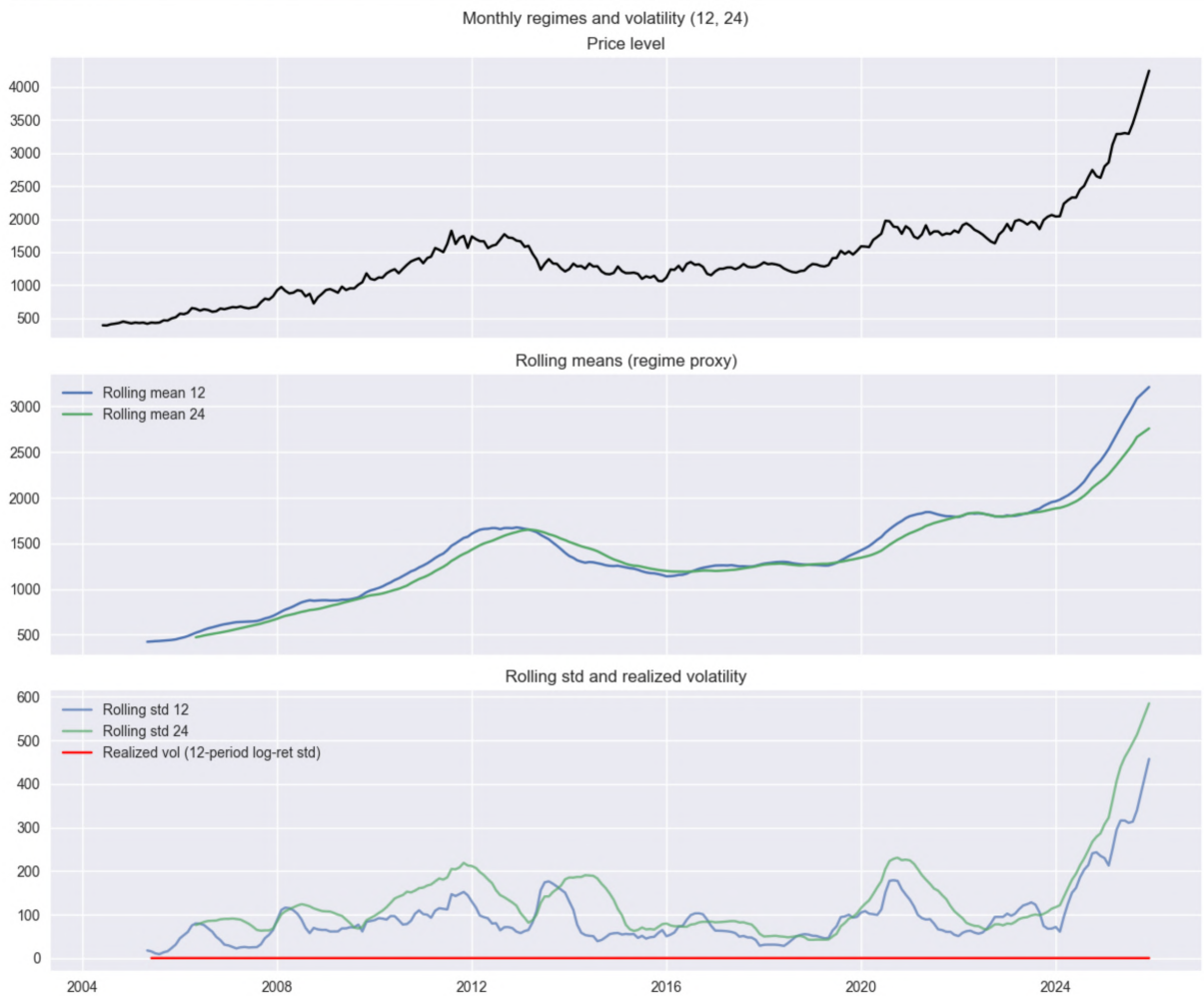
The plot shows how regimes and volatility evolve over time.

The z-score output (reg_m.tail()) lists the most recent flagged regimes, helping identify unusual market conditions.

This analysis step is crucial for understanding how gold prices transition between regimes (e.g., bull vs bear cycles) and how volatility clusters around major economic or geopolitical events.

In [6]:
```
# Cell: regime & volatility utilities and plots
def rolling_stats(series: pd.Series, windows):
    """
    Compute rolling mean/std over specified windows.
    - Monthly suggestion: (12, 24) ~ 1y, 2y
    - Daily suggestion: (30, 90) ~ 1m, 3m
    """
    df = pd.DataFrame({"value": series})
    for w in windows:
        df[f"roll_mean_{w}"] = series.rolling(w).mean()
        df[f"roll_std_{w}"] = series.rolling(w).std()
    return df

def realized_vol(series: pd.Series, window: int):
```

```python
    """
    Realized volatility: rolling std of log returns scaled by sqrt(window).
    """
    log_ret = np.log(series).diff()
    vol = log_ret.rolling(window).std() * np.sqrt(window)
    return vol

def plot_regime_vol(series: pd.Series, windows, title: str):
    """
    Composite visualization: price, rolling means (regimes), rolling std & realized vol (vol clustering).
    """
    df = rolling_stats(series, windows=windows)
    vol_window = windows[0] if isinstance(windows, (list, tuple)) else int(windows)
    vol = realized_vol(series, window=vol_window)

    fig, axes = plt.subplots(3, 1, figsize=(12, 10), sharex=True)

    axes[0].plot(series.index, series.values, label="Close", color="black")
    axes[0].set_title("Price level")

    for w in windows:
        axes[1].plot(df.index, df[f"roll_mean_{w}"], label=f"Rolling mean {w}")
    axes[1].legend()
    axes[1].set_title("Rolling means (regime proxy)")

    for w in windows:
        axes[2].plot(df.index, df[f"roll_std_{w}"], label=f"Rolling std {w}", alpha=0.7)
    axes[2].plot(vol.index, vol.values, label=f"Realized vol ({vol_window}-period log-ret std)", color="red")
    axes[2].legend()
    axes[2].set_title("Rolling std and realized volatility")

    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

def zscore_regimes(series: pd.Series, window: int = 24, threshold: float = 1.0) -> pd.DataFrame:
    """
    Rolling z-scores to flag outsized deviations (simple regime shift indicator).
    """
    roll_mean = series.rolling(window).mean()
    roll_std = series.rolling(window).std()
    z = (series - roll_mean) / roll_std
    regimes = (np.abs(z) > threshold).astype(int)
    return pd.DataFrame({"value": series, "zscore": z, "regime_flag": regimes})

# Monthly regime & volatility
plot_regime_vol(close_m, windows=(12, 24), title="Monthly regimes and volatility (12, 24)")
reg_m = zscore_regimes(close_m, window=24, threshold=1.0)
reg_m.tail()
```



Out[6]:

date	value	zscore	regime_flag
2025-06-01	3,302.9200	1.7993	1
2025-07-01	3,290.6800	1.5988	1
2025-08-01	3,448.1600	1.7315	1

2025-09-01	3,642.5600	1.9037	1
2025-12-01	4,246.3100	2.5403	1

Daily Gold Regime and Volatility Analysis

This cell investigates regime shifts and volatility clustering in the daily gold price series (if available):

Composite visualization (plot_regime_vol)

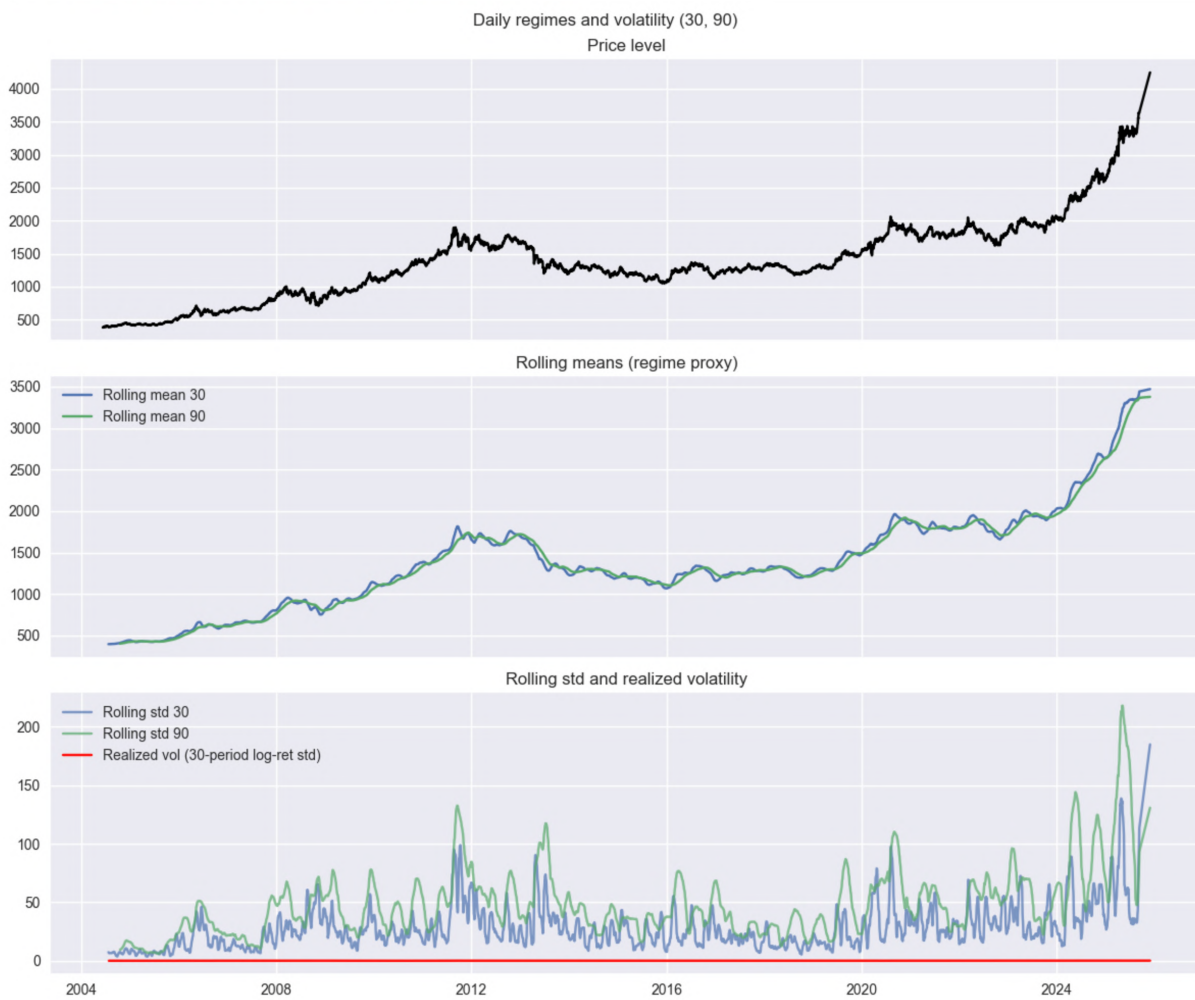
```
Plots the raw daily closing prices.

Adds rolling means (30-day and 90-day windows) as proxies for short-term and medium-term regimes. These highlight how the average price evolves and help
detect regime changes.

Shows rolling standard deviations and realized volatility (from log returns) to capture volatility clustering, where periods of high volatility tend to
follow each other.
```

Regime detection via z-scores (zscore_regimes)

```
Computes rolling z-scores over a 90-day window.

Flags extreme deviations (absolute z-score > 1.0) as potential regime shifts.

Produces a DataFrame with the latest values (reg_d.tail()), showing whether the market is in a normal state or flagged as an extreme regime.
```

Conditional execution If the daily dataset is not found, the analysis is skipped gracefully with a message, ensuring the notebook runs without errors.

In [7]:
```
# Cell: daily regime & volatility (skip if missing)
if close_d is not None:
    plot_regime_vol(close_d, windows=(30, 90), title="Daily regimes and volatility (30, 90)")
    reg_d = zscore_regimes(close_d, window=90, threshold=1.0)
    reg_d.tail()
else:
    print("Daily regime/volatility skipped: no daily dataset loaded.")
```



3. Baseline models: SARIMA and Holt–Winters

We build baselines with SARIMA and Holt–Winters. Monthly models use seasonal period 12. For daily, we start with weekly seasonality (7). We evaluate using MAE, RMSE, and MAPE, and plot forecasts versus actuals.

This cell defines utility functions to build and evaluate baseline forecasting models for the gold price series, using both statistical approaches and visualization:

time_split(series, test_size): Splits the time series chronologically into training and testing sets, reserving the last N observations for out-of-sample evaluation. This ensures forecasts are validated on unseen future data.

fit_sarima_forecast(train, test, order, seasonal_order): Fits a Seasonal ARIMA (SARIMA) model with specified autoregressive (p), differencing (d), moving average (q), and seasonal (P, D, Q, s) parameters.

```
The model captures both short-term dynamics and recurring seasonal patterns.
```

```
The model captures both short-term dynamics and recurring seasonal patterns.

Forecasts are generated for the test horizon and aligned with the test index.
```

fit_holt_winters_forecast(train, test, trend, seasonal, seasonal_periods): Implements Holt–Winters Triple Exponential Smoothing, which models level, trend, and seasonality.

```
Supports additive or multiplicative components depending on the data’s variance behavior.

Produces forecasts for the test horizon, aligned with actual dates.
```

plot_forecasts(train, test, preds, title): Creates a comparative plot showing:

```
Training data (gray).

Actual test values (black).

Forecasts from multiple models (dashed lines). This visual comparison highlights how well each baseline model tracks actual market movements.
```

Together, these functions provide a structured way to benchmark forecasting accuracy, compare statistical baselines, and visualize performance before moving on to more advanced machine learning or hybrid models.

In [8]:
```python
# Cell: baselines and plotting
def time_split(series: pd.Series, test_size: int):
    """
    Chronological train/test split by last N points.
    """
    train = series.iloc[:-test_size]
    test = series.iloc[-test_size:]
    return train, test

def fit_sarima_forecast(train: pd.Series, test: pd.Series,
                        order=(1,1,1), seasonal_order=(0,1,1,12)):
    """
    SARIMA baseline with given (p,d,q) and (P,D,Q,s).
    """
    model = SARIMAX(train, order=order, seasonal_order=seasonal_order,
                    enforce_stationarity=False, enforce_invertibility=False)
    res = model.fit(disp=False)
    pred = res.get_forecast(steps=len(test)).predicted_mean
    pred.index = test.index
    return res, pred

def fit_holt_winters_forecast(train: pd.Series, test: pd.Series,
                              trend="add", seasonal="add", seasonal_periods=12):
    """
    Holt-Winters (Triple Exponential Smoothing); try 'add' vs 'mul' seasonality.
    """
    hw = ExponentialSmoothing(train, trend=trend, seasonal=seasonal,
                              seasonal_periods=seasonal_periods, initialization_method="estimated")
    res = hw.fit()
    pred = res.forecast(len(test))
    pred.index = test.index
    return res, pred

def plot_forecasts(train: pd.Series, test: pd.Series, preds: dict, title: str):
    """
    Comparative plot for multiple forecast series.
    """
    plt.figure(figsize=(12, 6))
    plt.plot(train.index, train.values, label="Train", color="gray")
    plt.plot(test.index, test.values, label="Test (Actual)", color="black")
    for label, series in preds.items():
        plt.plot(series.index, series.values, label=label, linestyle="--")
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Monthly Baseline Forecasting and Benchmarking (SARIMA vs Holt–Winters)

This cell builds and evaluates baseline forecasting models for the monthly gold price series, using the last 24 months as the test set:

Train/Test Split: The series is divided chronologically, with the most recent 24 months reserved for out-of-sample evaluation. This ensures forecasts are validated against unseen data.

SARIMA Model: A Seasonal ARIMA model is fit with parameters (p,d,q) = (1,1,1) and seasonal (P,D,Q,s) = (0,1,1,12).

```
Captures short-term autoregressive and moving average dynamics.

Models yearly seasonality (s=12) in monthly data.

Produces forecasts aligned with the test horizon.
```

Holt–Winters Model: A Triple Exponential Smoothing model with additive trend and seasonality is fit.

```
Decomposes the series into level, trend, and seasonal components.

Seasonal period set to 12 months.

Generates forecasts for the test horizon.
```

Evaluation Metrics: Both models are benchmarked using MAE, RMSE, and MAPE to quantify forecast accuracy. Results are printed for comparison.

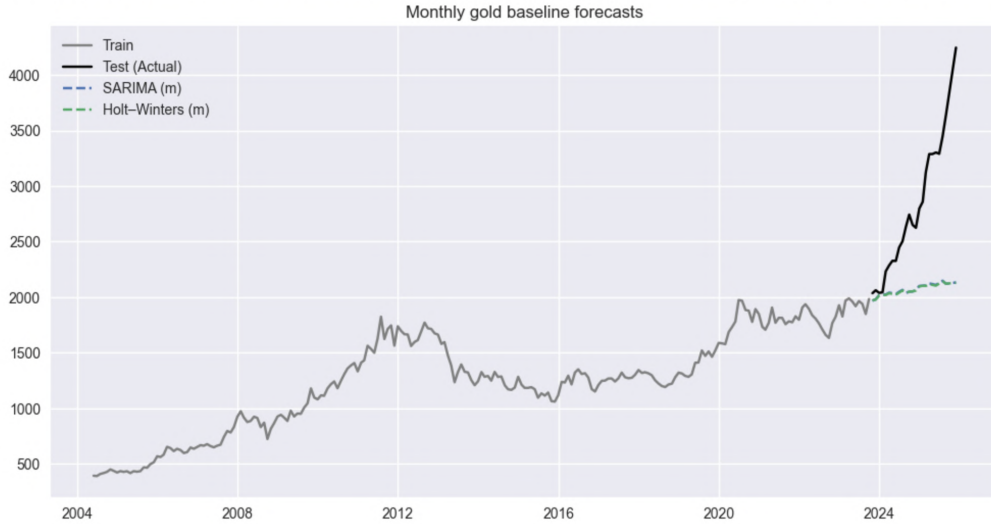
Visualization: Forecasts from SARIMA and Holt–Winters are plotted alongside training and test data. This visual comparison highlights how well each baseline tracks actual gold price movements.

This benchmarking step establishes baseline performance for monthly forecasting, providing a reference point before exploring more advanced or machine learning models.

In [9]:
```python
# Cell: monthly baselines and benchmarking
import warnings
warnings.filterwarnings("ignore")
# Use last 24 months for testing
train_m, test_m = time_split(close_m, test_size=24)

# SARIMA: simple starting point
sarima_m_res, sarima_m_pred = fit_sarima_forecast(train_m, test_m, order=(1,1,1), seasonal_order=(0,1,1,12))
sarima_m_metrics = evaluate_forecast(test_m, sarima_m_pred)

# Holt-Winters (additive)
hw_m_res, hw_m_pred = fit_holt_winters_forecast(train_m, test_m, trend="add", seasonal="add", seasonal_periods=12)
```

```
hw_m_res, hw_m_pred = fit_holt_winters_forecast(train_m, test_m, trend="add", seasonal="add", seasonal_periods=12)
hw_m_metrics = evaluate_forecast(test_m, hw_m_pred)

print("Monthly SARIMA metrics:", sarima_m_metrics)
print("Monthly Holt-Winters metrics:", hw_m_metrics)

plot_forecasts(train_m, test_m,
               preds={"SARIMA (m)": sarima_m_pred, "Holt-Winters (m)": hw_m_pred},
               title="Monthly gold baseline forecasts")
```

```
Monthly SARIMA metrics: {'MAE': 692.0667846482748, 'RMSE': np.float64(870.8979787085452), 'MAPE': np.float64(22.32372789315603)}
Monthly Holt-Winters metrics: {'MAE': 695.812848505943, 'RMSE': np.float64(874.085610197274), 'MAPE': np.float64(22.4607906480793)}
```



Daily Baseline Forecasting and Benchmarking (SARIMA vs Holt–Winters)

This cell benchmarks baseline forecasting models for the daily gold price series, using the most recent ~6 months (180 days) as the test set:

Train/Test Split: The daily series is divided chronologically, with the last 180 days reserved for out-of-sample evaluation. This ensures forecasts are validated against unseen data.

SARIMA Model: A Seasonal ARIMA model is fit with parameters (p,d,q) = (1,1,1) and seasonal (P,D,Q,s) = (0,1,1,7).

```
Weekly seasonality (s=7) is modeled to capture recurring weekly cycles.

Yearly seasonality (s=365) is possible but computationally heavier.

Forecasts are generated for the test horizon and aligned with actual dates.
```

Holt–Winters Model: A Triple Exponential Smoothing model with additive trend and weekly seasonality (seasonal_periods=7) is fit.

```
Decomposes the series into level, trend, and seasonal components.

Produces forecasts for the test horizon, aligned with the test index.
```

Evaluation Metrics: Both models are benchmarked using MAE, RMSE, and MAPE to quantify forecast accuracy. Results are printed for comparison.

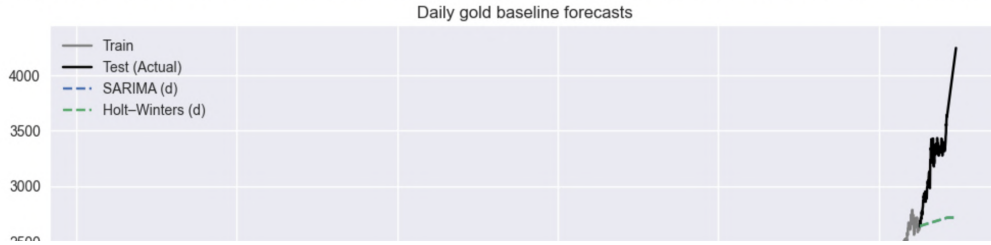
Visualization: Forecasts from SARIMA and Holt–Winters are plotted alongside training and test data. This comparative plot highlights how well each baseline tracks actual daily gold price movements.

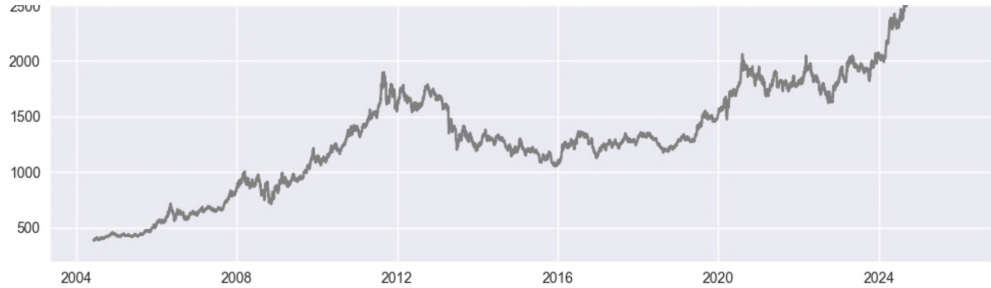
Conditional Execution: If the daily dataset is not available, the analysis is skipped gracefully with a message, ensuring the notebook runs without errors.

This benchmarking step provides a baseline reference for daily forecasting, allowing comparison of statistical models before exploring more advanced approaches.

In [10]:
```
# Cell: daily baselines and benchmarking (skip if missing)
import warnings
warnings.filterwarnings("ignore")
if close_d is not None:
    # Test size ~6 months (adjust to taste)
    train_d, test_d = time_split(close_d, test_size=180)

    # SARIMA with weekly seasonality s=7 (fast); yearly s=365 is heavier
    sarima_d_res, sarima_d_pred = fit_sarima_forecast(train_d, test_d, order=(1,1,1), seasonal_order=(0,1,1,7))
    sarima_d_metrics = evaluate_forecast(test_d, sarima_d_pred)

    # Holt-Winters weekly seasonality
    hw_d_res, hw_d_pred = fit_holt_winters_forecast(train_d, test_d, trend="add", seasonal="add", seasonal_periods=7)
    hw_d_metrics = evaluate_forecast(test_d, hw_d_pred)

    print("Daily SARIMA metrics:", sarima_d_metrics)
    print("Daily Holt-Winters metrics:", hw_d_metrics)

    plot_forecasts(train_d, test_d,
                   preds={"SARIMA (d)": sarima_d_pred, "Holt-Winters (d)": hw_d_pred},
                   title="Daily gold baseline forecasts")
else:
    print("Daily baselines skipped: no daily dataset loaded.")
```

```
Daily SARIMA metrics: {'MAE': 505.8292103493088, 'RMSE': np.float64(561.0741569775636), 'MAPE': np.float64(15.34856366259912)}
Daily Holt-Winters metrics: {'MAE': 505.304613741266, 'RMSE': np.float64(560.5153465575501), 'MAPE': np.float64(15.332519567087811)}
```





Optional ML baseline: elastic net with lag features

A lightweight ML baseline using lag features can complement statistical baselines. We train on lagged monthly closes and predict out-of-sample via a recursive scheme.

Machine Learning Baseline with Lag Features (Monthly Gold Forecasting) This cell introduces a machine learning baseline for forecasting monthly gold prices using lagged features and ElasticNet regression:

Lag Feature Construction (make_lag_features)

```
Builds a supervised learning dataset where each target value yt is predicted from its past values
Creates 12 lag features to capture one full year of historical dependencies.
Drops rows with missing values caused by shifting.
```

ElasticNet Regression (rolling_predict_elasticnet)

```
Fits an ElasticNet model (combining L1 and L2 regularization) on the lagged training data.

Uses a pipeline with StandardScaler for normalization and ElasticNet for regression.

Parameters:

alpha=0.1 controls overall regularization strength.

l1_ratio=0.2 balances between L1 (sparsity) and L2 (ridge) penalties.

Forecasting is recursive: predictions are generated step-by-step, each new forecast fed back into the lag history to predict the next point.
```

Evaluation and Benchmarking

```
Forecasts are generated for the test horizon (last 24 months).

Accuracy is measured using MAE, RMSE, and MAPE.

Results are printed for comparison against statistical baselines.
```

Visualization (plot_forecasts)

```
Plots training data, actual test values, and forecasts from three models:

    ElasticNet with lag features (ML baseline).

    SARIMA statistical baseline.

    Holt-Winters statistical baseline.

Provides a direct visual comparison of machine learning vs traditional time series methods.
```

This step benchmarks a simple machine learning approach against classical statistical baselines, highlighting whether lag-based regression can capture monthly gold price dynamics as effectively as SARIMA or Holt–Winters.

In [11]:

```python
# Cell: ML baseline with lags (monthly)
def make_lag_features(series: pd.Series, lags=12) -> pd.DataFrame:
    """
    Create lag features X_t = [y_{t-1}, ..., y_{t-lags}] and target y_t.
    """
    df = pd.DataFrame({"y": series})
    for i in range(1, lags+1):
        df[f"lag_{i}"] = df["y"].shift(i)
    return df.dropna()

def rolling_predict_elasticnet(train: pd.Series, test: pd.Series, lags=12, alpha=0.1, l1_ratio=0.2):
    """
    Fit ElasticNet on lag features and produce recursive forecasts over the test horizon.
    """
    df_train = make_lag_features(train, lags=lags)
    X_train = df_train[[f"lag_{i}" for i in range(1, lags+1)]].values
    y_train = df_train["y"].values

    model = Pipeline([
        ("scaler", StandardScaler()),
        ("enet", ElasticNet(alpha=alpha, l1_ratio=l1_ratio, max_iter=10000, random_state=42))
    ])
    model.fit(X_train, y_train)

    # Recursive forecasting: start from the last lags in train
    history = list(train.iloc[-lags:].values)
    preds = []
    for _ in range(len(test)):
        x = np.array(history[-lags:])[::-1]  # lag_1 is last value
        x = x.reshape(1, -1)
        yhat = model.predict(x)[0]
        preds.append(yhat)
        history.append(yhat)

    pred_series = pd.Series(preds, index=test.index, name="enet_pred")
    return pred_series

# Run ML baseline on monthly data
enet_pred_m = rolling_predict_elasticnet(train_m, test_m, lags=12, alpha=0.1, l1_ratio=0.2)
enet_m_metrics = evaluate_forecast(test_m, enet_pred_m)
```

```
print("Monthly ElasticNet-lags metrics:", enet_m_metrics)

plot_forecasts(train_m, test_m,
               preds={"ElasticNet-lags (m)": enet_pred_m, "SARIMA (m)": sarima_m_pred, "Holt-Winters (m)": hw_m_pred},
               title="Monthly gold: ML vs statistical baselines")
```

```
Monthly ElasticNet-lags metrics: {'MAE': 846.9435710816934, 'RMSE': np.float64(1027.6133821403564), 'MAPE': np.float64(27.756323171879526)}
```

In [12]:
```
print("=== Monthly Forecast Accuracy ===")
print("SARIMA:", evaluate_forecast(test_m, sarima_m_pred))
print("Holt-Winters:", evaluate_forecast(test_m, hw_m_pred))
print("ElasticNet:", evaluate_forecast(test_m, enet_pred_m))  # if ML model used
print("----------------------------------------------------------------")
print("=== Daily Forecast Accuracy ===")
print("SARIMA:", evaluate_forecast(test_d, sarima_d_pred))
print("Holt-Winters:", evaluate_forecast(test_d, hw_d_pred))
```

```
=== Monthly Forecast Accuracy ===
SARIMA: {'MAE': 692.0667846482748, 'RMSE': np.float64(870.8979787085452), 'MAPE': np.float64(22.32372789315603)}
Holt-Winters: {'MAE': 695.812848505943, 'RMSE': np.float64(874.085610197274), 'MAPE': np.float64(22.4607906480793)}
ElasticNet: {'MAE': 846.9435710816934, 'RMSE': np.float64(1027.6133821403564), 'MAPE': np.float64(27.756323171879526)}
----------------------------------------------------------------
=== Daily Forecast Accuracy ===
SARIMA: {'MAE': 505.8292103493088, 'RMSE': np.float64(561.0741569775636), 'MAPE': np.float64(15.34856366259912)}
Holt-Winters: {'MAE': 505.304613741266, 'RMSE': np.float64(560.5153465575501), 'MAPE': np.float64(15.332519567087811)}
```

In [13]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
import numpy as np

# Actual direction
actual_dir_m = np.where(test_m.diff().fillna(0) > 0, 1, 0)

# Predicted direction
pred_dir_sarima_m = np.where(pd.Series(sarima_m_pred).diff().fillna(0) > 0, 1, 0)
pred_dir_holt_m = np.where(pd.Series(hw_m_pred).diff().fillna(0) > 0, 1, 0)

print("=== Monthly Direction Confusion Matrix (SARIMA) ===")
print(confusion_matrix(actual_dir_m, pred_dir_sarima_m))
print("Accuracy:", accuracy_score(actual_dir_m, pred_dir_sarima_m))

print("\n=== Monthly Direction Confusion Matrix (Holt-Winters) ===")
print(confusion_matrix(actual_dir_m, pred_dir_holt_m))
print("Accuracy:", accuracy_score(actual_dir_m, pred_dir_holt_m))
```

```
=== Monthly Direction Confusion Matrix (SARIMA) ===
[[ 2  4]
 [ 5 13]]
Accuracy: 0.625

=== Monthly Direction Confusion Matrix (Holt-Winters) ===
[[ 2  4]
 [ 7 11]]
Accuracy: 0.5416666666666666
```

In [14]:
```
actual_dir_d = np.where(test_d.diff().fillna(0) > 0, 1, 0)

pred_dir_sarima_d = np.where(pd.Series(sarima_d_pred).diff().fillna(0) > 0, 1, 0)
pred_dir_holt_d = np.where(pd.Series(hw_d_pred).diff().fillna(0) > 0, 1, 0)

print("=== Daily Direction Confusion Matrix (SARIMA) ===")
print(confusion_matrix(actual_dir_d, pred_dir_sarima_d))
print("Accuracy:", accuracy_score(actual_dir_d, pred_dir_sarima_d))

print("\n=== Daily Direction Confusion Matrix (Holt-Winters) ===")
print(confusion_matrix(actual_dir_d, pred_dir_holt_d))
print("Accuracy:", accuracy_score(actual_dir_d, pred_dir_holt_d))
```

```
=== Daily Direction Confusion Matrix (SARIMA) ===
[[28 47]
 [25 80]]
Accuracy: 0.6

=== Daily Direction Confusion Matrix (Holt-Winters) ===
[[28 47]
 [25 80]]
Accuracy: 0.6
```

In [ ]:

In [1]:
```python
"""
Gold Price Forecasting using Deep Learning
Masters in Applied AI Project

This script implements advanced deep learning techniques for time series forecasting:
- Recurrent Neural Networks (RNN)
- Long Short-Term Memory (LSTM)
- Gated Recurrent Units (GRU)
- Bidirectional architectures
- Transfer Learning
- Attention mechanisms
- Ensemble methods

Libraries: TensorFlow, Keras
Concepts: Backpropagation, Gradient Flow, Transfer Learning
"""
```

Out[1]:
```
'\nGold Price Forecasting using Deep Learning\nMasters in Applied AI Project\n\nThis script implements advanced deep learning techniques for time series forecasting:\n- Recur
rent Neural Networks (RNN)\n- Long Short-Term Memory (LSTM)\n- Gated Recurrent Units (GRU)\n- Bidirectional architectures\n- Transfer Learning\n- Attention mechanisms\n- Ense
mble methods\n\nLibraries: TensorFlow, Keras\nConcepts: Backpropagation, Gradient Flow, Transfer Learning\n'
```

In [2]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error, mean_absolute_percentage_error, r2_score
import warnings
warnings.filterwarnings('ignore')

# Deep Learning Libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import (Dense, LSTM, GRU, SimpleRNN, Dropout,
                                     Bidirectional, Conv1D, MaxPooling1D,
                                     Flatten, Input, Concatenate, Attention,
                                     LayerNormalization, MultiHeadAttention)
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau, TensorBoard
from tensorflow.keras.optimizers import Adam, RMSprop
from tensorflow.keras.regularizers import l1_l2
```

In [3]:
```python
# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)
```

In [5]:
```python
# Load the dataset
df = pd.read_excel('C:/Users/ravde/OneDrive/MS AAI/AAI-501/Project/XAU_15m_data.xlsx')
```

In [6]:
```python
# Display dataset information
print("Dataset Shape:", df.shape)
print("\nFirst 10 rows:")
print(df.head(10))
print("\nLast 10 rows:")
print(df.tail(10))
print("\nDataset Info:")
print(df.info())
print("\nStatistical Summary:")
print(df.describe())
```

```
Dataset Shape: (485968, 7)

First 10 rows:
         Date      Time   Open   High    Low  Close  Volume
0  2004.06.11  07:15:00  384.0  384.3  383.8  384.3      12
1  2004.06.11  07:30:00  383.8  384.3  383.6  383.8      12
2  2004.06.11  07:45:00  383.3  383.8  383.3  383.8      20
3  2004.06.11  08:00:00  383.8  384.1  383.6  383.6       8
4  2004.06.11  08:15:00  383.6  384.3  383.5  383.5      20
5  2004.06.11  08:30:00  383.3  383.5  383.3  383.5       7
6  2004.06.11  08:45:00  383.3  383.3  383.1  383.1       6
7  2004.06.11  09:00:00  383.1  384.1  383.1  383.6       9
8  2004.06.11  09:15:00  383.8  384.0  382.8  383.0      15
9  2004.06.11  09:30:00  383.1  383.3  382.8  383.0      20

Last 10 rows:
               Date      Time     Open     High      Low    Close  Volume
485958  2025.10.01  03:15:00  3865.09  3865.29  3861.16  3861.26    2463
485959  2025.10.01  03:30:00  3861.10  3864.84  3858.55  3863.56    3716
485960  2025.10.01  03:45:00  3863.57  3864.17  3860.55  3860.61    2541
485961  2025.10.01  04:00:00  3860.57  3863.06  3858.90  3862.26    3122
485962  2025.10.01  04:15:00  3862.27  3865.14  3861.44  3864.87    3547
485963  2025.10.01  04:30:00  3864.86  3866.20  3863.59  3865.92    2496
485964  2025.10.01  04:45:00  3865.93  3868.91  3865.45  3868.81    3277
485965  2025.10.01  05:00:00  3868.79  3875.57  3867.30  3867.52    7364
485966  2025.10.01  05:15:00  3867.64  3870.65  3864.68  3866.16    4483
485967  2025.10.01  05:30:00  3866.19  3866.19  3864.94  3865.88     122

Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 485968 entries, 0 to 485967
Data columns (total 7 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Date    485968 non-null  object
 1   Time    485968 non-null  object
 2   Open    485968 non-null  float64
 3   High    485968 non-null  float64
 4   Low     485968 non-null  float64
 5   Close   485968 non-null  float64
 6   Volume  485968 non-null  int64
dtypes: float64(4), int64(1), object(2)
memory usage: 26.0+ MB
None

Statistical Summary:
                Open           High            Low          Close  \
count  485968.000000  485968.000000  485968.000000  485968.000000
mean     1382.059356    1382.959692    1381.138505    1382.061666
std       582.917032     583.273915     582.552627     582.927162
min       381.300000     381.600000     381.100000     381.300000
25%      1074.417500    1075.110000    1073.660000    1074.407500
50%      1297.900000    1298.520000    1297.220000    1297.900000
75%      1736.700000    1737.860000    1735.502500    1736.710000
```

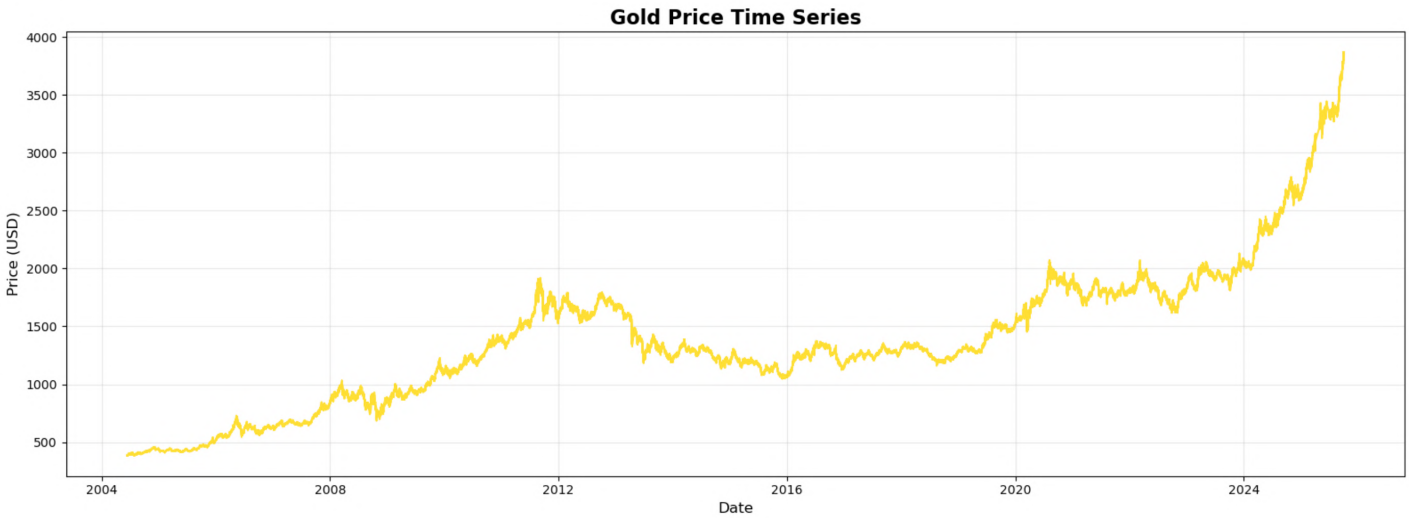
```
75%      1730.700000    1737.800000    1733.302500    1736.710000
max      3869.650000    3875.570000    3867.300000    3869.820000

              Volume
count  485968.000000
mean      922.968749
std      1300.473617
min         1.000000
25%       188.000000
50%       617.000000
75%      1193.000000
max     64585.000000
```

In [7]:
```
# Convert date column to datetime
date_column = df.columns[0]
price_column = df.columns[1] if len(df.columns) > 1 else df.columns[0]

df[date_column] = pd.to_datetime(df[date_column])
df = df.sort_values(date_column)
df = df.reset_index(drop=True)
```

In [11]:
```
# Extract price data
prices = df['Close'].values.reshape(-1, 1)

print(f"\nTotal data points: {len(prices)}")
print(f"Date range: {df[date_column].min()} to {df[date_column].max()}")
print(f"Price range: ${prices.min():.2f} to ${prices.max():.2f}")
```

```
Total data points: 485968
Date range: 2004-06-11 00:00:00 to 2025-10-01 00:00:00
Price range: $381.30 to $3869.82
```

In [13]:
```
# Visualize the gold price time series
plt.figure(figsize=(16, 6))
plt.plot(df['Date'], df['Close'], linewidth=1.5, color='gold', alpha=0.8)
plt.title('Gold Price Time Series', fontsize=16, fontweight='bold')
plt.xlabel('Date', fontsize=12)
plt.ylabel('Price (USD)', fontsize=12)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



In [14]:
```
# Create additional features
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day
df['DayOfWeek'] = df['Date'].dt.dayofweek
df['Quarter'] = df['Date'].dt.quarter

# Technical indicators
df['MA_7'] = df['Close'].rolling(window=7).mean()
df['MA_30'] = df['Close'].rolling(window=30).mean()
df['MA_90'] = df['Close'].rolling(window=90).mean()
df['EMA_7'] = df['Close'].ewm(span=7, adjust=False).mean()
df['EMA_30'] = df['Close'].ewm(span=30, adjust=False).mean()

# Volatility
df['Volatility_7'] = df['Close'].rolling(window=7).std()
df['Volatility_30'] = df['Close'].rolling(window=30).mean()

# Price changes
df['Price_Change'] = df['Close'].diff()
df['Price_Change_Pct'] = df['Close'].pct_change() * 100
```

In [16]:
```
# Lag features
for lag in [1, 7, 30]:
    df[f'Lag_{lag}'] = df['Close'].shift(lag)
```

In [17]:
```
# Remove NaN values created by rolling operations
df = df.dropna().reset_index(drop=True)
```

In [18]:
```
print(f"Dataset shape after feature engineering: {df.shape}")
print("\nFeatures created:")
for col in df.columns:
    print(f"  - {col}")
```

```
Dataset shape after feature engineering: (485879, 24)

Features created:
  - Date
  - Time
  - Open
  - High
  - Low
  - Close
```

```
  - Volume
  - Year
  - Month
  - Day
  - DayOfWeek
  - Quarter
  - MA_7
  - MA_30
  - MA_90
  - EMA_7
  - EMA_30
  - Volatility_7
  - Volatility_30
  - Price_Change
  - Price_Change_Pct
  - Lag_1
  - Lag_7
  - Lag_30
```

In [19]:
```python
# Prepare data for deep learning
def create_sequences(data, lookback=60, forecast_horizon=1):
    """
    Create sequences for time series forecasting

    Args:
        data: Input data array
        lookback: Number of time steps to look back
        forecast_horizon: Number of steps to forecast ahead

    Returns:
        X: Input sequences
        y: Target values
    """
    X, y = [], []
    for i in range(lookback, len(data) - forecast_horizon + 1):
        X.append(data[i-lookback:i])
        y.append(data[i + forecast_horizon - 1])
    return np.array(X), np.array(y)

# Define parameters
LOOKBACK = 60  # Use 60 days of historical data
FORECAST_HORIZON = 1  # Predict 1 day ahead

# Scale the data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_prices = scaler.fit_transform(prices)

print(f"Sequence parameters:")
print(f"  - Lookback window: {LOOKBACK} days")
print(f"  - Forecast horizon: {FORECAST_HORIZON} day(s)")
print(f"  - Scaling method: MinMaxScaler (0-1)")
```

```
Sequence parameters:
  - Lookback window: 60 days
  - Forecast horizon: 1 day(s)
  - Scaling method: MinMaxScaler (0-1)
```

In [20]:
```python
# Create sequences
X, y = create_sequences(scaled_prices, LOOKBACK, FORECAST_HORIZON)

print(f"\nSequence shapes:")
print(f"  - X shape: {X.shape} (samples, timesteps, features)")
print(f"  - y shape: {y.shape}")

# Train-validation-test split (70-15-15)
train_size = int(0.7 * len(X))
val_size = int(0.15 * len(X))

X_train = X[:train_size]
y_train = y[:train_size]

X_val = X[train_size:train_size+val_size]
y_val = y[train_size:train_size+val_size]

X_test = X[train_size+val_size:]
y_test = y[train_size+val_size:]

print(f"\nData split:")
print(f"  - Training set: {X_train.shape[0]} samples ({X_train.shape[0]/len(X)*100:.1f}%)")
print(f"  - Validation set: {X_val.shape[0]} samples ({X_val.shape[0]/len(X)*100:.1f}%)")
print(f"  - Test set: {X_test.shape[0]} samples ({X_test.shape[0]/len(X)*100:.1f}%)")
```

```
Sequence shapes:
  - X shape: (485908, 60, 1) (samples, timesteps, features)
  - y shape: (485908, 1)

Data split:
  - Training set: 340135 samples (70.0%)
  - Validation set: 72886 samples (15.0%)
  - Test set: 72887 samples (15.0%)
```

In [21]:
```python
# Define common parameters
INPUT_SHAPE = (LOOKBACK, 1)
EPOCHS = 50
BATCH_SIZE = 32

# Callbacks
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True,
    verbose=1
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=5,
    min_lr=1e-7,
    verbose=1
)
```

In [22]:
```python
# Model 1: Simple RNN
model_rnn = Sequential([
    SimpleRNN(64, activation='tanh', return_sequences=True, input_shape=INPUT_SHAPE),
    Dropout(0.2),
```

```
    SimpleRNN(32, activation='tanh'),
    Dropout(0.2),
    Dense(16, activation='relu'),
    Dense(1)
], name='Simple_RNN')

model_rnn.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae'])
print(model_rnn.summary())
```

Model: "Simple_RNN"

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 60, 64)	4,224
dropout (Dropout)	(None, 60, 64)	0
simple_rnn_1 (SimpleRNN)	(None, 32)	3,104
dropout_1 (Dropout)	(None, 32)	0
dense (Dense)	(None, 16)	528
dense_1 (Dense)	(None, 1)	17

Total params: 7,873 (30.75 KB)
Trainable params: 7,873 (30.75 KB)
Non-trainable params: 0 (0.00 B)
None

In [24]:
```
# Model 2: LSTM (Long Short-Term Memory)
model_lstm = Sequential([
    LSTM(128, activation='tanh', return_sequences=True, input_shape=INPUT_SHAPE),
    Dropout(0.3),
    LSTM(64, activation='tanh', return_sequences=True),
    Dropout(0.3),
    LSTM(32, activation='tanh'),
    Dropout(0.2),
    Dense(16, activation='relu'),
    Dense(1)
], name='LSTM')

model_lstm.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae'])
print(model_lstm.summary())
```

Model: "LSTM"

Layer (type)	Output Shape	Param #
lstm_3 (LSTM)	(None, 60, 128)	66,560
dropout_5 (Dropout)	(None, 60, 128)	0
lstm_4 (LSTM)	(None, 60, 64)	49,408
dropout_6 (Dropout)	(None, 60, 64)	0
lstm_5 (LSTM)	(None, 32)	12,416
dropout_7 (Dropout)	(None, 32)	0
dense_4 (Dense)	(None, 16)	528
dense_5 (Dense)	(None, 1)	17

Total params: 128,929 (503.63 KB)
Trainable params: 128,929 (503.63 KB)
Non-trainable params: 0 (0.00 B)
None

In [25]:
```
# Model 3: GRU (Gated Recurrent Unit)
model_gru = Sequential([
    GRU(128, activation='tanh', return_sequences=True, input_shape=INPUT_SHAPE),
    Dropout(0.3),
    GRU(64, activation='tanh', return_sequences=True),
    Dropout(0.3),
    GRU(32, activation='tanh'),
    Dropout(0.2),
    Dense(16, activation='relu'),
    Dense(1)
], name='GRU')

model_gru.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae'])
print(model_gru.summary())
```

Model: "GRU"

Layer (type)	Output Shape	Param #
gru (GRU)	(None, 60, 128)	50,304
dropout_8 (Dropout)	(None, 60, 128)	0
gru_1 (GRU)	(None, 60, 64)	37,248
dropout_9 (Dropout)	(None, 60, 64)	0
gru_2 (GRU)	(None, 32)	9,408
dropout_10 (Dropout)	(None, 32)	0
dense_6 (Dense)	(None, 16)	528
dense_7 (Dense)	(None, 1)	17

Total params: 97,505 (380.88 KB)
Trainable params: 97,505 (380.88 KB)
Non-trainable params: 0 (0.00 B)
None

In [26]:
```
# Model 4: Bidirectional LSTM
model_bilstm = Sequential([
    Bidirectional(LSTM(64, activation='tanh', return_sequences=True), input_shape=INPUT_SHAPE),
```

```
    Dropout(0.3),
    Bidirectional(LSTM(32, activation='tanh')),
    Dropout(0.2),
    Dense(16, activation='relu'),
    Dense(1)
], name='Bidirectional_LSTM')

model_bilstm.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae'])
print(model_bilstm.summary())
```

Model: "Bidirectional_LSTM"

Layer (type)	Output Shape	Param #
bidirectional (Bidirectional)	(None, 60, 128)	33,792
dropout_11 (Dropout)	(None, 60, 128)	0
bidirectional_1 (Bidirectional)	(None, 64)	41,216
dropout_12 (Dropout)	(None, 64)	0
dense_8 (Dense)	(None, 16)	1,040
dense_9 (Dense)	(None, 1)	17

Total params: 76,065 (297.13 KB)

Trainable params: 76,065 (297.13 KB)

Non-trainable params: 0 (0.00 B)

None

In [27]:
```
# Model 5: CNN-LSTM Hybrid
model_cnn_lstm = Sequential([
    Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=INPUT_SHAPE),
    MaxPooling1D(pool_size=2),
    Conv1D(filters=32, kernel_size=3, activation='relu'),
    LSTM(64, activation='tanh', return_sequences=True),
    Dropout(0.3),
    LSTM(32, activation='tanh'),
    Dropout(0.2),
    Dense(16, activation='relu'),
    Dense(1)
], name='CNN_LSTM')

model_cnn_lstm.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae'])
print(model_cnn_lstm.summary())
```

Model: "CNN_LSTM"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 58, 64)	256
max_pooling1d (MaxPooling1D)	(None, 29, 64)	0
conv1d_1 (Conv1D)	(None, 27, 32)	6,176
lstm_8 (LSTM)	(None, 27, 64)	24,832
dropout_13 (Dropout)	(None, 27, 64)	0
lstm_9 (LSTM)	(None, 32)	12,416
dropout_14 (Dropout)	(None, 32)	0
dense_10 (Dense)	(None, 16)	528
dense_11 (Dense)	(None, 1)	17

Total params: 44,225 (172.75 KB)

Trainable params: 44,225 (172.75 KB)

Non-trainable params: 0 (0.00 B)

None

In [28]:
```
# Model 6: Stacked LSTM with Attention (Advanced)
# Custom attention layer
class AttentionLayer(keras.layers.Layer):
    def __init__(self, **kwargs):
        super(AttentionLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        self.W = self.add_weight(name='attention_weight',
                                 shape=(input_shape[-1], input_shape[-1]),
                                 initializer='glorot_uniform',
                                 trainable=True)
        self.b = self.add_weight(name='attention_bias',
                                 shape=(input_shape[-1],),
                                 initializer='zeros',
                                 trainable=True)
        super(AttentionLayer, self).build(input_shape)

    def call(self, x):
        e = tf.nn.tanh(tf.matmul(x, self.W) + self.b)
        a = tf.nn.softmax(e, axis=1)
        output = x * a
        return tf.reduce_sum(output, axis=1)

inputs = Input(shape=INPUT_SHAPE)
x = LSTM(128, activation='tanh', return_sequences=True)(inputs)
x = Dropout(0.3)(x)
x = LSTM(64, activation='tanh', return_sequences=True)(x)
x = Dropout(0.3)(x)
x = AttentionLayer()(x)
x = Dense(32, activation='relu')(x)
x = Dropout(0.2)(x)
x = Dense(16, activation='relu')(x)
outputs = Dense(1)(x)

model_attention = Model(inputs=inputs, outputs=outputs, name='LSTM_Attention')
model_attention.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae'])
print(model_attention.summary())
```

```
WARNING:tensorflow:From C:\Users\ravde\anaconda3\Lib\site-packages\keras\src\backend\tensorflow\core.py:232: The name tf.placeholder is deprecated. Please use tf.compat.v1.placeholder instead.
```

Model: "LSTM_Attention"

Layer (type)	Output Shape	Param #
input_layer_6 (InputLayer)	(None, 60, 1)	0
lstm_10 (LSTM)	(None, 60, 128)	66,560
dropout_15 (Dropout)	(None, 60, 128)	0
lstm_11 (LSTM)	(None, 60, 64)	49,408
dropout_16 (Dropout)	(None, 60, 64)	0
attention_layer (AttentionLayer)	(None, 64)	4,160
dense_12 (Dense)	(None, 32)	2,080
dropout_17 (Dropout)	(None, 32)	0
dense_13 (Dense)	(None, 16)	528
dense_14 (Dense)	(None, 1)	17

Total params: 122,753 (479.50 KB)
Trainable params: 122,753 (479.50 KB)
Non-trainable params: 0 (0.00 B)
None

In [29]:
```python
# Dictionary to store models and their histories
models = {
    'Simple RNN': model_rnn,
    'LSTM': model_lstm,
    'GRU': model_gru,
    'Bidirectional LSTM': model_bilstm,
    'CNN-LSTM': model_cnn_lstm,
    'LSTM with Attention': model_attention
}

histories = {}
training_times = {}

import time

for model_name, model in models.items():
    print("\n" + "=" * 80)
    print(f"Training {model_name}...")
    print("=" * 80)

    start_time = time.time()

    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=EPOCHS,
        batch_size=BATCH_SIZE,
        callbacks=[early_stopping, reduce_lr],
        verbose=1
    )

    end_time = time.time()
    training_time = end_time - start_time

    histories[model_name] = history
    training_times[model_name] = training_time

    print(f"\nTraining completed in {training_time:.2f} seconds")
    print(f"Final training loss: {history.history['loss'][-1]:.6f}")
    print(f"Final validation loss: {history.history['val_loss'][-1]:.6f}")
```

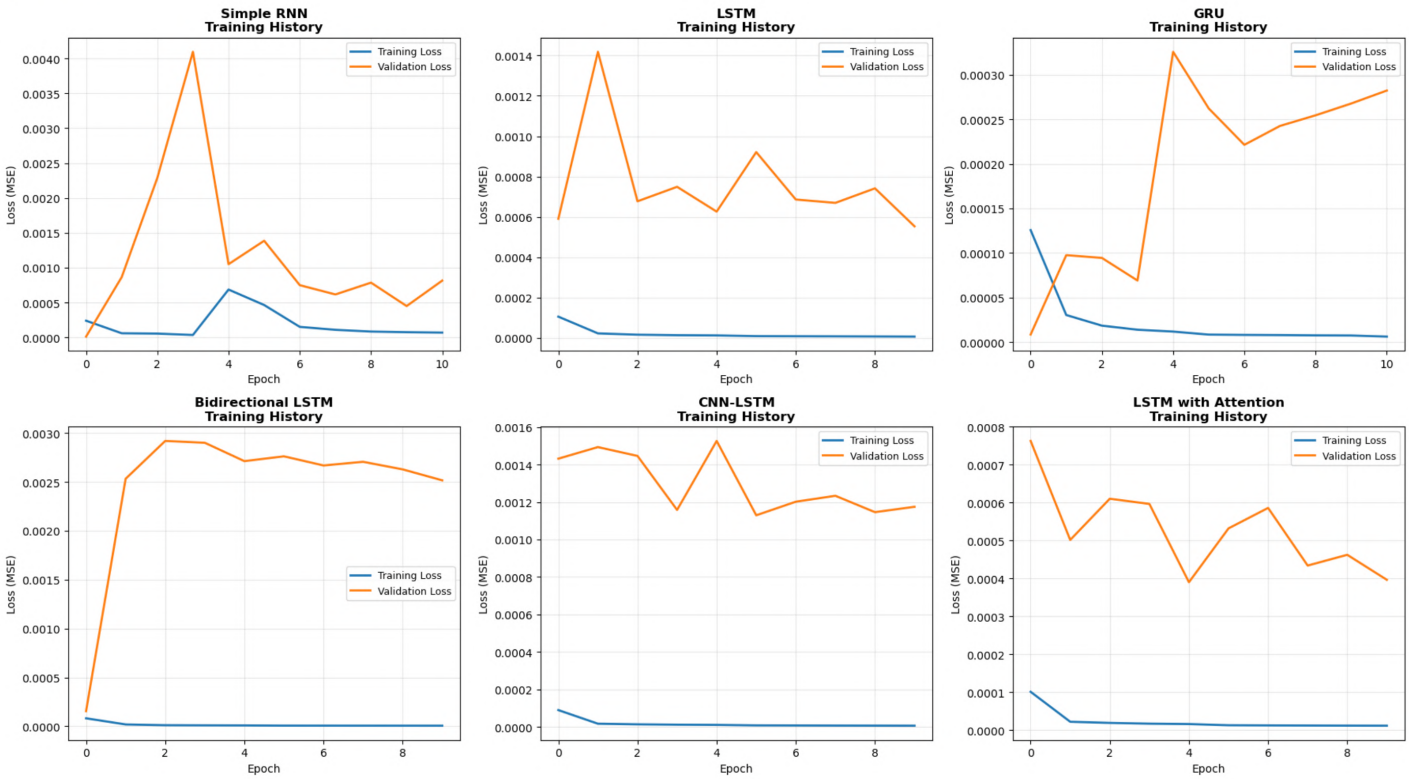
```
================================================================================
Training Simple RNN...
================================================================================
Epoch 1/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 214s 20ms/step - loss: 2.3939e-04 - mae: 0.0092 - val_loss: 1.1554e-05 - val_mae: 0.0027 - learning_rate: 0.0010
Epoch 2/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 210s 20ms/step - loss: 5.9178e-05 - mae: 0.0056 - val_loss: 8.6385e-04 - val_mae: 0.0288 - learning_rate: 0.0010
Epoch 3/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 209s 20ms/step - loss: 5.4650e-05 - mae: 0.0049 - val_loss: 0.0023 - val_mae: 0.0469 - learning_rate: 0.0010
Epoch 4/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 210s 20ms/step - loss: 3.4781e-05 - mae: 0.0042 - val_loss: 0.0041 - val_mae: 0.0631 - learning_rate: 0.0010
Epoch 5/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 209s 20ms/step - loss: 6.8622e-04 - mae: 0.0133 - val_loss: 0.0011 - val_mae: 0.0275 - learning_rate: 0.0010
Epoch 6/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - loss: 5.3086e-04 - mae: 0.0157
Epoch 6: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 207s 19ms/step - loss: 4.6435e-04 - mae: 0.0146 - val_loss: 0.0014 - val_mae: 0.0337 - learning_rate: 0.0010
Epoch 7/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 204s 19ms/step - loss: 1.5098e-04 - mae: 0.0092 - val_loss: 7.4903e-04 - val_mae: 0.0214 - learning_rate: 5.0000e-04
Epoch 8/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 206s 19ms/step - loss: 1.0877e-04 - mae: 0.0078 - val_loss: 6.1600e-04 - val_mae: 0.0197 - learning_rate: 5.0000e-04
Epoch 9/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 208s 20ms/step - loss: 8.3682e-05 - mae: 0.0068 - val_loss: 7.8522e-04 - val_mae: 0.0231 - learning_rate: 5.0000e-04
Epoch 10/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 208s 20ms/step - loss: 7.4881e-05 - mae: 0.0064 - val_loss: 4.4942e-04 - val_mae: 0.0159 - learning_rate: 5.0000e-04
Epoch 11/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - loss: 7.1085e-05 - mae: 0.0063
Epoch 11: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 208s 20ms/step - loss: 6.8827e-05 - mae: 0.0061 - val_loss: 8.1473e-04 - val_mae: 0.0258 - learning_rate: 5.0000e-04
Epoch 11: early stopping
Restoring model weights from the end of the best epoch: 1.

Training completed in 2293.86 seconds
Final training loss: 0.000069
Final validation loss: 0.000815

================================================================================
Training LSTM...
================================================================================
Epoch 1/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 980s 92ms/step - loss: 1.0583e-04 - mae: 0.0064 - val_loss: 5.9076e-04 - val_mae: 0.0226 - learning_rate: 0.0010
Epoch 2/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 968s 91ms/step - loss: 2.2932e-05 - mae: 0.0035 - val_loss: 0.0014 - val_mae: 0.0355 - learning_rate: 0.0010
```

```
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 968s 91ms/step - loss: 2.2952e-05 - mae: 0.0035 - val_loss: 0.0014 - val_mae: 0.0355 - learning_rate: 0.0010
Epoch 3/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 966s 91ms/step - loss: 1.6439e-05 - mae: 0.0030 - val_loss: 6.7728e-04 - val_mae: 0.0248 - learning_rate: 0.0010
Epoch 4/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 970s 91ms/step - loss: 1.3885e-05 - mae: 0.0027 - val_loss: 7.4902e-04 - val_mae: 0.0250 - learning_rate: 0.0010
Epoch 5/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - loss: 1.3161e-05 - mae: 0.0026
Epoch 5: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 1005s 95ms/step - loss: 1.2860e-05 - mae: 0.0026 - val_loss: 6.2612e-04 - val_mae: 0.0235 - learning_rate: 0.0010
Epoch 6/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 1009s 95ms/step - loss: 9.4200e-06 - mae: 0.0022 - val_loss: 9.2102e-04 - val_mae: 0.0288 - learning_rate: 5.0000e-04
Epoch 7/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 951s 89ms/step - loss: 8.8964e-06 - mae: 0.0021 - val_loss: 6.8605e-04 - val_mae: 0.0246 - learning_rate: 5.0000e-04
Epoch 8/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 951s 89ms/step - loss: 8.4161e-06 - mae: 0.0021 - val_loss: 6.6943e-04 - val_mae: 0.0248 - learning_rate: 5.0000e-04
Epoch 9/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 954s 90ms/step - loss: 7.7524e-06 - mae: 0.0020 - val_loss: 7.4144e-04 - val_mae: 0.0257 - learning_rate: 5.0000e-04
Epoch 10/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 82ms/step - loss: 7.2610e-06 - mae: 0.0019
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 959s 90ms/step - loss: 7.0426e-06 - mae: 0.0019 - val_loss: 5.5324e-04 - val_mae: 0.0226 - learning_rate: 5.0000e-04
Epoch 10: early stopping
Restoring model weights from the end of the best epoch: 1.

Training completed in 9712.76 seconds
Final training loss: 0.000007
Final validation loss: 0.000553

================================================================================
Training GRU...
================================================================================
Epoch 1/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 949s 89ms/step - loss: 1.2562e-04 - mae: 0.0070 - val_loss: 8.4185e-06 - val_mae: 0.0021 - learning_rate: 0.0010
Epoch 2/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 935s 88ms/step - loss: 3.0318e-05 - mae: 0.0040 - val_loss: 9.7522e-05 - val_mae: 0.0086 - learning_rate: 0.0010
Epoch 3/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 932s 88ms/step - loss: 1.8395e-05 - mae: 0.0031 - val_loss: 9.4421e-05 - val_mae: 0.0081 - learning_rate: 0.0010
Epoch 4/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 933s 88ms/step - loss: 1.3806e-05 - mae: 0.0027 - val_loss: 6.9012e-05 - val_mae: 0.0068 - learning_rate: 0.0010
Epoch 5/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 82ms/step - loss: 1.2158e-05 - mae: 0.0025
Epoch 5: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 931s 88ms/step - loss: 1.1720e-05 - mae: 0.0025 - val_loss: 3.2578e-04 - val_mae: 0.0150 - learning_rate: 0.0010
Epoch 6/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 932s 88ms/step - loss: 8.3358e-06 - mae: 0.0020 - val_loss: 2.6221e-04 - val_mae: 0.0135 - learning_rate: 5.0000e-04
Epoch 7/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 932s 88ms/step - loss: 7.9863e-06 - mae: 0.0020 - val_loss: 2.2137e-04 - val_mae: 0.0125 - learning_rate: 5.0000e-04
Epoch 8/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 933s 88ms/step - loss: 7.7822e-06 - mae: 0.0020 - val_loss: 2.4256e-04 - val_mae: 0.0132 - learning_rate: 5.0000e-04
Epoch 9/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 933s 88ms/step - loss: 7.4682e-06 - mae: 0.0019 - val_loss: 2.5450e-04 - val_mae: 0.0135 - learning_rate: 5.0000e-04
Epoch 10/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 82ms/step - loss: 7.3601e-06 - mae: 0.0019
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 933s 88ms/step - loss: 7.3349e-06 - mae: 0.0019 - val_loss: 2.6774e-04 - val_mae: 0.0139 - learning_rate: 5.0000e-04
Epoch 11/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 932s 88ms/step - loss: 6.0950e-06 - mae: 0.0017 - val_loss: 2.8231e-04 - val_mae: 0.0143 - learning_rate: 2.5000e-04
Epoch 11: early stopping
Restoring model weights from the end of the best epoch: 1.

Training completed in 10273.69 seconds
Final training loss: 0.000006
Final validation loss: 0.000282

================================================================================
Training Bidirectional LSTM...
================================================================================
Epoch 1/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 552s 51ms/step - loss: 8.2049e-05 - mae: 0.0056 - val_loss: 1.5556e-04 - val_mae: 0.0113 - learning_rate: 0.0010
Epoch 2/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 538s 51ms/step - loss: 1.9468e-05 - mae: 0.0032 - val_loss: 0.0025 - val_mae: 0.0493 - learning_rate: 0.0010
Epoch 3/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 537s 50ms/step - loss: 1.1697e-05 - mae: 0.0025 - val_loss: 0.0029 - val_mae: 0.0529 - learning_rate: 0.0010
Epoch 4/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 534s 50ms/step - loss: 1.0207e-05 - mae: 0.0023 - val_loss: 0.0029 - val_mae: 0.0527 - learning_rate: 0.0010
Epoch 5/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step - loss: 9.4227e-06 - mae: 0.0022
Epoch 5: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 534s 50ms/step - loss: 9.2602e-06 - mae: 0.0022 - val_loss: 0.0027 - val_mae: 0.0511 - learning_rate: 0.0010
Epoch 6/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 535s 50ms/step - loss: 7.1464e-06 - mae: 0.0019 - val_loss: 0.0028 - val_mae: 0.0515 - learning_rate: 5.0000e-04
Epoch 7/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 535s 50ms/step - loss: 6.9601e-06 - mae: 0.0019 - val_loss: 0.0027 - val_mae: 0.0506 - learning_rate: 5.0000e-04
Epoch 8/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 535s 50ms/step - loss: 6.7565e-06 - mae: 0.0018 - val_loss: 0.0027 - val_mae: 0.0508 - learning_rate: 5.0000e-04
Epoch 9/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 539s 51ms/step - loss: 6.5389e-06 - mae: 0.0018 - val_loss: 0.0026 - val_mae: 0.0499 - learning_rate: 5.0000e-04
Epoch 10/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step - loss: 6.4655e-06 - mae: 0.0018
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 536s 50ms/step - loss: 6.4029e-06 - mae: 0.0018 - val_loss: 0.0025 - val_mae: 0.0487 - learning_rate: 5.0000e-04
Epoch 10: early stopping
Restoring model weights from the end of the best epoch: 1.

Training completed in 5375.60 seconds
Final training loss: 0.000006
Final validation loss: 0.002515

================================================================================
Training CNN-LSTM...
================================================================================
Epoch 1/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 272s 25ms/step - loss: 8.9095e-05 - mae: 0.0052 - val_loss: 0.0014 - val_mae: 0.0362 - learning_rate: 0.0010
Epoch 2/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 262s 25ms/step - loss: 1.6437e-05 - mae: 0.0029 - val_loss: 0.0015 - val_mae: 0.0363 - learning_rate: 0.0010
Epoch 3/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 262s 25ms/step - loss: 1.3188e-05 - mae: 0.0026 - val_loss: 0.0014 - val_mae: 0.0358 - learning_rate: 0.0010
Epoch 4/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 261s 25ms/step - loss: 1.1375e-05 - mae: 0.0024 - val_loss: 0.0012 - val_mae: 0.0323 - learning_rate: 0.0010
Epoch 5/50
10628/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 1.0600e-05 - mae: 0.0023
Epoch 5: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 261s 25ms/step - loss: 1.0277e-05 - mae: 0.0023 - val_loss: 0.0015 - val_mae: 0.0373 - learning_rate: 0.0010
Epoch 6/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 262s 25ms/step - loss: 7.5157e-06 - mae: 0.0019 - val_loss: 0.0011 - val_mae: 0.0316 - learning_rate: 5.0000e-04
Epoch 7/50
```

```
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 262s 25ms/step - loss: 7.0564e-06 - mae: 0.0018 - val_loss: 0.0012 - val_mae: 0.0328 - learning_rate: 5.0000e-04
Epoch 8/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 262s 25ms/step - loss: 6.5032e-06 - mae: 0.0018 - val_loss: 0.0012 - val_mae: 0.0329 - learning_rate: 5.0000e-04
Epoch 9/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 262s 25ms/step - loss: 6.1812e-06 - mae: 0.0017 - val_loss: 0.0011 - val_mae: 0.0318 - learning_rate: 5.0000e-04
Epoch 10/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 5.8744e-06 - mae: 0.0017
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 263s 25ms/step - loss: 5.7623e-06 - mae: 0.0017 - val_loss: 0.0012 - val_mae: 0.0324 - learning_rate: 5.0000e-04
Epoch 10: early stopping
Restoring model weights from the end of the best epoch: 1.

Training completed in 2628.60 seconds
Final training loss: 0.000006
Final validation loss: 0.001174

================================================================================
Training LSTM with Attention...
================================================================================
Epoch 1/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 813s 76ms/step - loss: 1.0112e-04 - mae: 0.0054 - val_loss: 7.6257e-04 - val_mae: 0.0253 - learning_rate: 0.0010
Epoch 2/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 803s 76ms/step - loss: 2.2118e-05 - mae: 0.0034 - val_loss: 5.0157e-04 - val_mae: 0.0206 - learning_rate: 0.0010
Epoch 3/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 807s 76ms/step - loss: 1.9160e-05 - mae: 0.0031 - val_loss: 6.1017e-04 - val_mae: 0.0230 - learning_rate: 0.0010
Epoch 4/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 842s 79ms/step - loss: 1.7098e-05 - mae: 0.0029 - val_loss: 5.9649e-04 - val_mae: 0.0223 - learning_rate: 0.0010
Epoch 5/50
10629/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 72ms/step - loss: 1.6239e-05 - mae: 0.0029
Epoch 5: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 834s 78ms/step - loss: 1.6102e-05 - mae: 0.0028 - val_loss: 3.9006e-04 - val_mae: 0.0178 - learning_rate: 0.0010
Epoch 6/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 803s 76ms/step - loss: 1.2965e-05 - mae: 0.0025 - val_loss: 5.3197e-04 - val_mae: 0.0209 - learning_rate: 5.0000e-04
Epoch 7/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 798s 75ms/step - loss: 1.2519e-05 - mae: 0.0025 - val_loss: 5.8592e-04 - val_mae: 0.0222 - learning_rate: 5.0000e-04
Epoch 8/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 789s 74ms/step - loss: 1.2182e-05 - mae: 0.0024 - val_loss: 4.3397e-04 - val_mae: 0.0188 - learning_rate: 5.0000e-04
Epoch 9/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 793s 75ms/step - loss: 1.1910e-05 - mae: 0.0024 - val_loss: 4.6232e-04 - val_mae: 0.0193 - learning_rate: 5.0000e-04
Epoch 10/50
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step - loss: 1.1774e-05 - mae: 0.0024
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
10630/10630 ━━━━━━━━━━━━━━━━━━━━ 794s 75ms/step - loss: 1.1725e-05 - mae: 0.0024 - val_loss: 3.9644e-04 - val_mae: 0.0180 - learning_rate: 5.0000e-04
Epoch 10: early stopping
Restoring model weights from the end of the best epoch: 1.

Training completed in 8075.83 seconds
Final training loss: 0.000012
Final validation loss: 0.000396
```

In [30]:

```python
# Visualize training histories
fig, axes = plt.subplots(2, 3, figsize=(18, 10))
axes = axes.ravel()

for idx, (model_name, history) in enumerate(histories.items()):
    ax = axes[idx]
    ax.plot(history.history['loss'], label='Training Loss', linewidth=2)
    ax.plot(history.history['val_loss'], label='Validation Loss', linewidth=2)
    ax.set_title(f'{model_name}\nTraining History', fontsize=12, fontweight='bold')
    ax.set_xlabel('Epoch', fontsize=10)
    ax.set_ylabel('Loss (MSE)', fontsize=10)
    ax.legend(fontsize=9)
    ax.grid(True, alpha=0.3)

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



In [31]:

```python
# Evaluate all models
results = []
```

```python
for model_name, model in models.items():
    print(f"\n{'-' * 80}")
    print(f"Evaluating {model_name}...")
    print('-' * 80)

    # Make predictions
    y_train_pred = model.predict(X_train, verbose=0)
    y_val_pred = model.predict(X_val, verbose=0)
    y_test_pred = model.predict(X_test, verbose=0)

    # Inverse transform predictions
    y_train_actual = scaler.inverse_transform(y_train)
    y_train_pred_actual = scaler.inverse_transform(y_train_pred)

    y_val_actual = scaler.inverse_transform(y_val)
    y_val_pred_actual = scaler.inverse_transform(y_val_pred)

    y_test_actual = scaler.inverse_transform(y_test)
    y_test_pred_actual = scaler.inverse_transform(y_test_pred)

    # Calculate metrics for test set
    mae = mean_absolute_error(y_test_actual, y_test_pred_actual)
    rmse = np.sqrt(mean_squared_error(y_test_actual, y_test_pred_actual))
    mape = mean_absolute_percentage_error(y_test_actual, y_test_pred_actual) * 100
    r2 = r2_score(y_test_actual, y_test_pred_actual)

    print(f"\nTest Set Performance:")
    print(f"  MAE:  ${mae:.2f}")
    print(f"  RMSE: ${rmse:.2f}")
    print(f"  MAPE: {mape:.2f}%")
    print(f"  R² Score: {r2:.4f}")
    print(f"  Training Time: {training_times[model_name]:.2f} seconds")

    results.append({
        'Model': model_name,
        'MAE': mae,
        'RMSE': rmse,
        'MAPE': mape,
        'R2': r2,
        'Training Time (s)': training_times[model_name],
        'Predictions': y_test_pred_actual
    })
```

```
--------------------------------------------------------------------------------
Evaluating Simple RNN...
--------------------------------------------------------------------------------

Test Set Performance:
  MAE:  $97.81
  RMSE: $159.05
  MAPE: 3.41%
  R² Score: 0.9127
  Training Time: 2293.86 seconds

--------------------------------------------------------------------------------
Evaluating LSTM...
--------------------------------------------------------------------------------

Test Set Performance:
  MAE:  $299.93
  RMSE: $394.26
  MAPE: 11.40%
  R² Score: 0.4639
  Training Time: 9712.76 seconds

--------------------------------------------------------------------------------
Evaluating GRU...
--------------------------------------------------------------------------------

Test Set Performance:
  MAE:  $135.29
  RMSE: $236.20
  MAPE: 4.58%
  R² Score: 0.8076
  Training Time: 10273.69 seconds

--------------------------------------------------------------------------------
Evaluating Bidirectional LSTM...
--------------------------------------------------------------------------------

Test Set Performance:
  MAE:  $192.26
  RMSE: $267.68
  MAPE: 7.13%
  R² Score: 0.7529
  Training Time: 5375.60 seconds

--------------------------------------------------------------------------------
Evaluating CNN-LSTM...
--------------------------------------------------------------------------------

Test Set Performance:
  MAE:  $336.74
  RMSE: $385.67
  MAPE: 13.61%
  R² Score: 0.4870
  Training Time: 2628.60 seconds

--------------------------------------------------------------------------------
Evaluating LSTM with Attention...
--------------------------------------------------------------------------------

Test Set Performance:
  MAE:  $179.49
  RMSE: $194.71
  MAPE: 7.48%
  R² Score: 0.8692
  Training Time: 8075.83 seconds
```

In [32]:

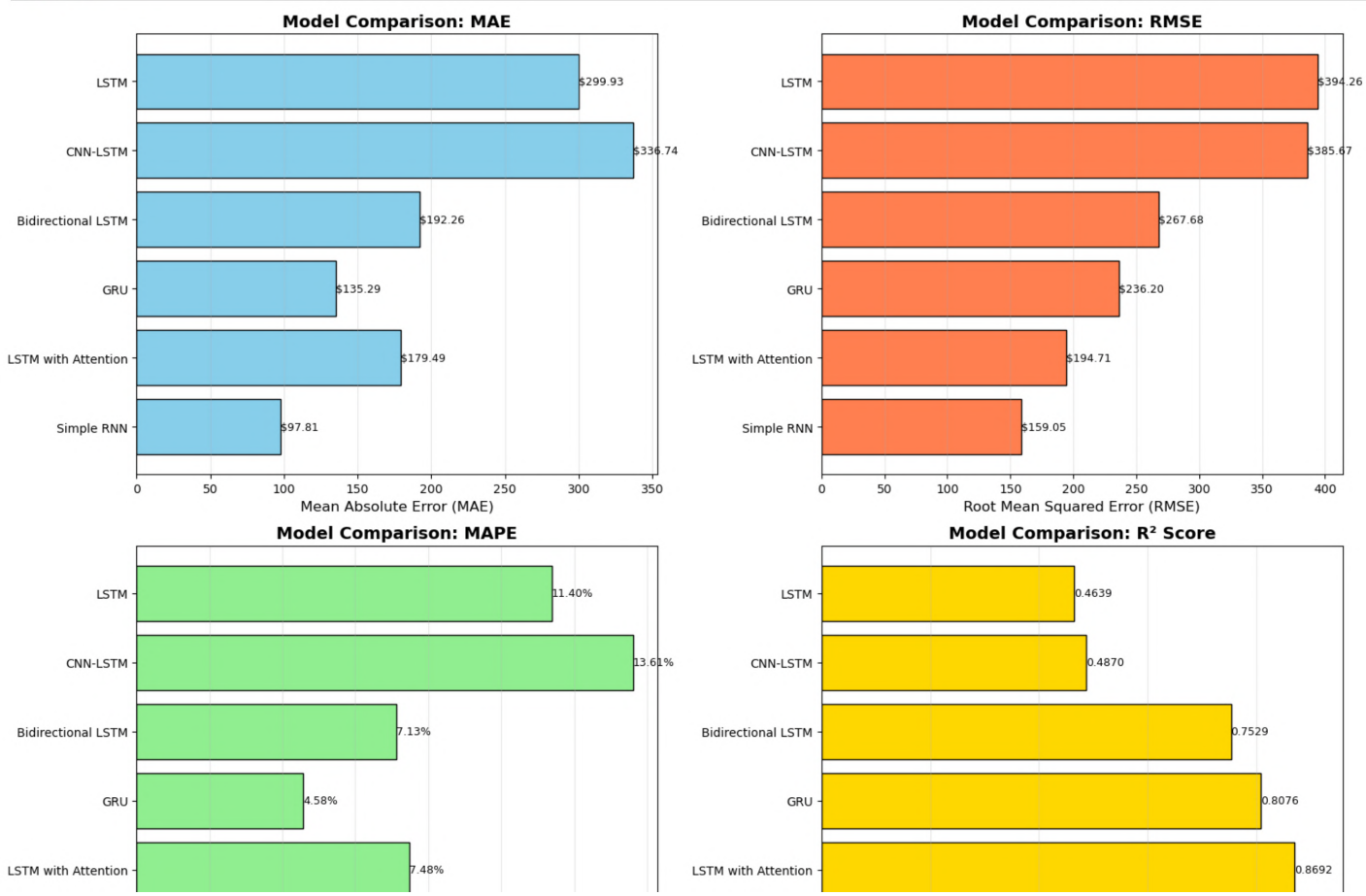
```python
# Create results DataFrame
results_df = pd.DataFrame(results)
results_df = results_df.sort_values('RMSE')

print("\n" + "=" * 80)
print("MODEL COMPARISON SUMMARY")
print("=" * 80)
```

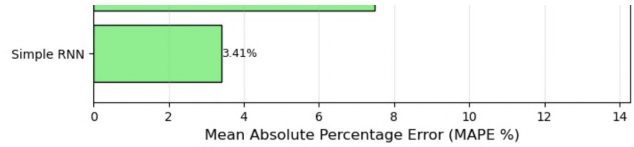
```
print("\n", results_df[['Model', 'MAE', 'RMSE', 'MAPE', 'R2', 'Training Time (s)']].to_string(index=False))
```

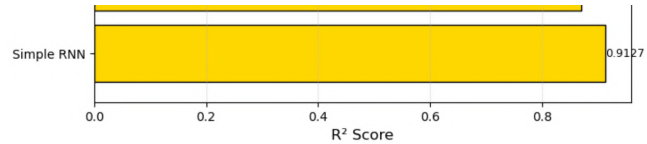
```
================================================================================
MODEL COMPARISON SUMMARY
================================================================================

               Model        MAE       RMSE      MAPE       R2  Training Time (s)
          Simple RNN  97.807249 159.049063  3.414331 0.912749        2293.864767
LSTM with Attention 179.489486 194.711464  7.482627 0.869235        8075.827173
                 GRU 135.292925 236.199881  4.578223 0.807572       10273.685806
 Bidirectional LSTM 192.256517 267.678291  7.127831 0.752865        5375.604988
           CNN-LSTM 336.741814 385.666388 13.611087 0.486982        2628.595854
               LSTM 299.934131 394.255638 11.395797 0.463877        9712.762239
```

In [33]:

```
# Visualize model comparison
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# Plot 1: MAE Comparison
ax1 = axes[0, 0]
bars1 = ax1.barh(results_df['Model'], results_df['MAE'], color='skyblue', edgecolor='black')
ax1.set_xlabel('Mean Absolute Error (MAE)', fontsize=12)
ax1.set_title('Model Comparison: MAE', fontsize=14, fontweight='bold')
ax1.grid(True, alpha=0.3, axis='x')
for i, bar in enumerate(bars1):
    width = bar.get_width()
    ax1.text(width, bar.get_y() + bar.get_height()/2, f'${width:.2f}',
             ha='left', va='center', fontsize=9)

# Plot 2: RMSE Comparison
ax2 = axes[0, 1]
bars2 = ax2.barh(results_df['Model'], results_df['RMSE'], color='coral', edgecolor='black')
ax2.set_xlabel('Root Mean Squared Error (RMSE)', fontsize=12)
ax2.set_title('Model Comparison: RMSE', fontsize=14, fontweight='bold')
ax2.grid(True, alpha=0.3, axis='x')
for i, bar in enumerate(bars2):
    width = bar.get_width()
    ax2.text(width, bar.get_y() + bar.get_height()/2, f'${width:.2f}',
             ha='left', va='center', fontsize=9)

# Plot 3: MAPE Comparison
ax3 = axes[1, 0]
bars3 = ax3.barh(results_df['Model'], results_df['MAPE'], color='lightgreen', edgecolor='black')
ax3.set_xlabel('Mean Absolute Percentage Error (MAPE %)', fontsize=12)
ax3.set_title('Model Comparison: MAPE', fontsize=14, fontweight='bold')
ax3.grid(True, alpha=0.3, axis='x')
for i, bar in enumerate(bars3):
    width = bar.get_width()
    ax3.text(width, bar.get_y() + bar.get_height()/2, f'{width:.2f}%',
             ha='left', va='center', fontsize=9)

# Plot 4: R² Score Comparison
ax4 = axes[1, 1]
bars4 = ax4.barh(results_df['Model'], results_df['R2'], color='gold', edgecolor='black')
ax4.set_xlabel('R² Score', fontsize=12)
ax4.set_title('Model Comparison: R² Score', fontsize=14, fontweight='bold')
ax4.grid(True, alpha=0.3, axis='x')
for i, bar in enumerate(bars4):
    width = bar.get_width()
    ax4.text(width, bar.get_y() + bar.get_height()/2, f'{width:.4f}',
             ha='left', va='center', fontsize=9)

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

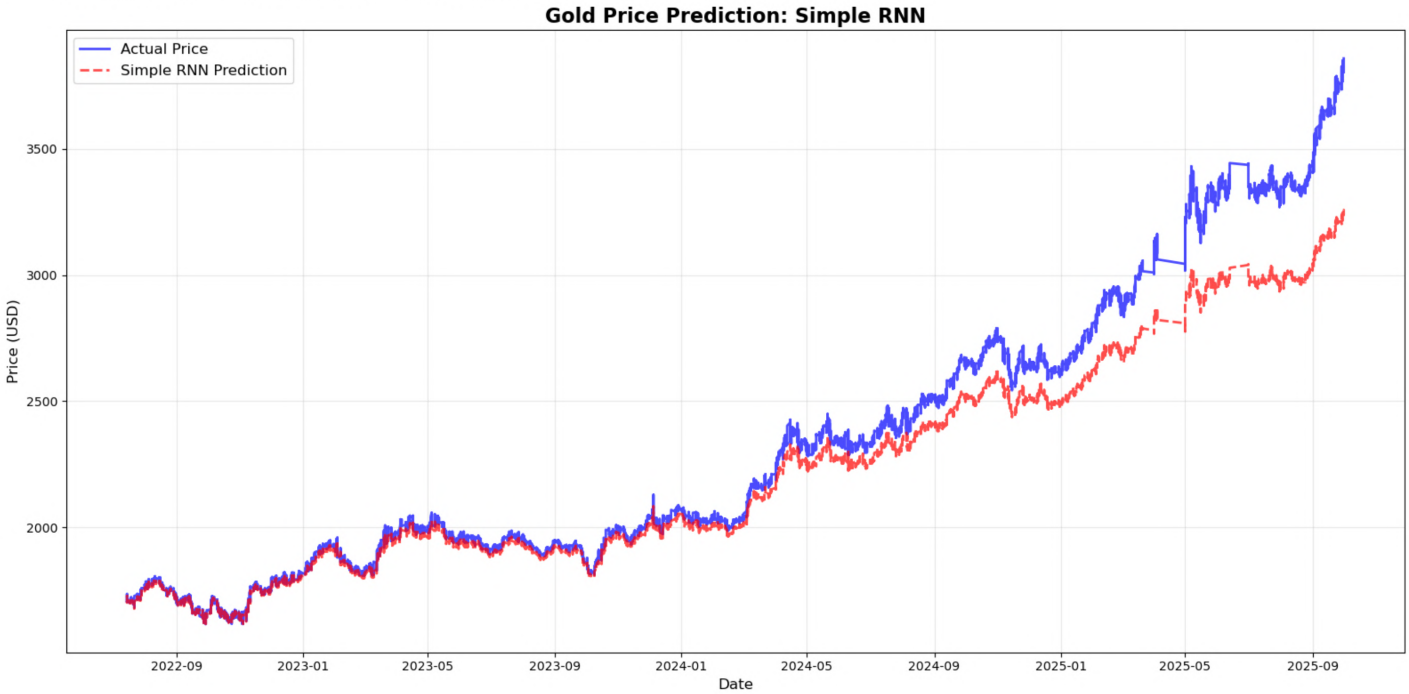






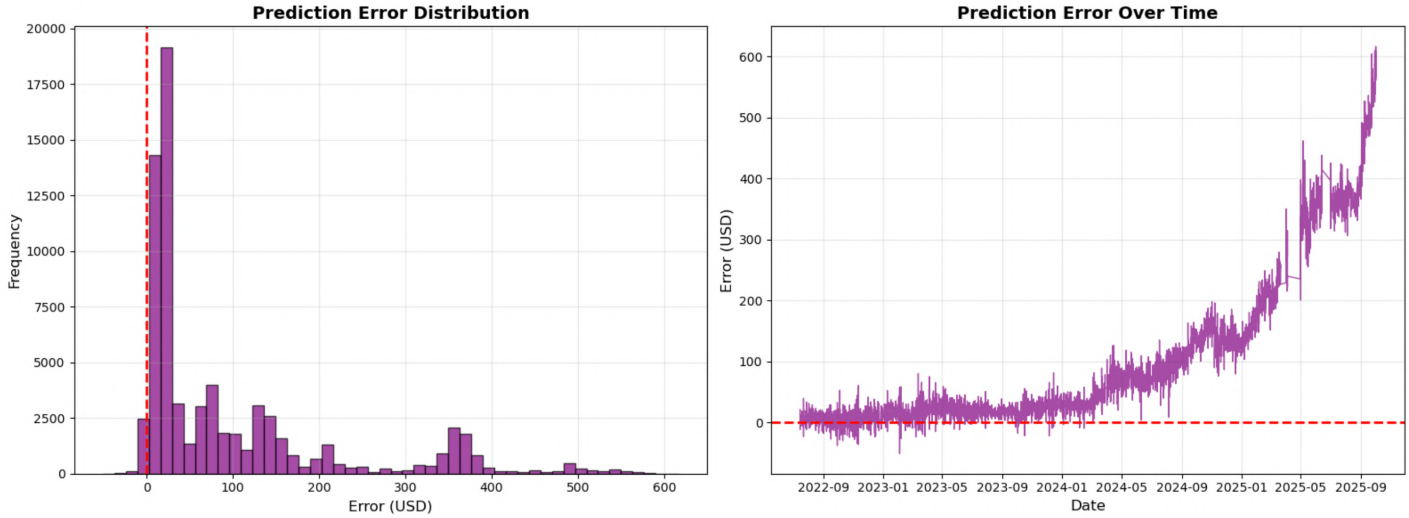
In [36]:
```
# Visualize predictions for best model
best_model_name = results_df.iloc[0]['Model']
best_model = models[best_model_name]
best_predictions = results_df.iloc[0]['Predictions']

print(f"\n{'=' * 80}")
print(f"BEST MODEL: {best_model_name}")
print(f"{'=' * 80}")

# Plot actual vs predicted
plt.figure(figsize=(16, 8))

# Plot test set predictions
test_dates = df[date_column].iloc[train_size+val_size+LOOKBACK:]
test_dates = test_dates[:len(best_predictions)]
y_test_actual = y_test_actual[:len(best_predictions)]

# Reshape predictions if needed
if hasattr(best_predictions, 'shape') and len(best_predictions.shape) > 1:
    best_predictions = best_predictions.flatten()  # Convert from 2D to 1D if needed

# Check and ensure all arrays have the same length
min_length = min(len(test_dates), len(y_test_actual), len(best_predictions))
test_dates = test_dates[:min_length]
y_test_actual = y_test_actual[:min_length]
best_predictions = best_predictions[:min_length]


plt.plot(test_dates, y_test_actual, label='Actual Price', linewidth=2, color='blue', alpha=0.7)
plt.plot(test_dates, best_predictions, label=f'{best_model_name} Prediction',
         linewidth=2, color='red', alpha=0.7, linestyle='--')

plt.title(f'Gold Price Prediction: {best_model_name}', fontsize=16, fontweight='bold')
plt.xlabel('Date', fontsize=12)
plt.ylabel('Price (USD)', fontsize=12)
plt.legend(fontsize=12)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
================================================================================
BEST MODEL: Simple RNN
================================================================================
```



In [37]:
```
# Plot prediction errors
errors = y_test_actual.flatten() - best_predictions.flatten()

fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Error distribution
axes[0].hist(errors, bins=50, color='purple', alpha=0.7, edgecolor='black')
axes[0].axvline(x=0, color='red', linestyle='--', linewidth=2)
axes[0].set_title('Prediction Error Distribution', fontsize=14, fontweight='bold')
axes[0].set_xlabel('Error (USD)', fontsize=12)
axes[0].set_ylabel('Frequency', fontsize=12)
axes[0].grid(True, alpha=0.3)

# Error over time
axes[1].plot(test_dates, errors, color='purple', alpha=0.7, linewidth=1)
axes[1].axhline(y=0, color='red', linestyle='--', linewidth=2)
axes[1].set_title('Prediction Error Over Time', fontsize=14, fontweight='bold')
axes[1].set_xlabel('Date', fontsize=12)
axes[1].set_ylabel('Error (USD)', fontsize=12)
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
```

```
plt.show()
```



In [ ]:
```
print("Transfer Learning Approach:")
print("Using pre-trained LSTM model as feature extractor for fine-tuning")

# Use the best LSTM model as base
base_model = model_lstm

# Freeze initial layers
for layer in base_model.layers[:2]:
    layer.trainable = False

print("\nFrozen layers:")
for layer in base_model.layers:
    print(f"  {layer.name}: trainable = {layer.trainable}")

# Fine-tune the model
print("\nFine-tuning the model...")
history_transfer = base_model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=20,
    batch_size=BATCH_SIZE,
    callbacks=[early_stopping],
    verbose=1
)
```

```
Transfer Learning Approach:
Using pre-trained LSTM model as feature extractor for fine-tuning

Frozen layers:
  lstm_3: trainable = False
  dropout_5: trainable = False
  lstm_4: trainable = True
  dropout_6: trainable = True
  lstm_5: trainable = True
  dropout_7: trainable = True
  dense_4: trainable = True
  dense_5: trainable = True

Fine-tuning the model...
Epoch 1/20
  338/10630 ━━━━━━━━━━━━━━━━━━━━ 16:46 98ms/step - loss: 3.3459e-05 - mae: 0.0042
```

In [ ]:
```
# Evaluate transfer learning model
y_test_pred_transfer = base_model.predict(X_test, verbose=0)
y_test_pred_transfer_actual = scaler.inverse_transform(y_test_pred_transfer)

mae_transfer = mean_absolute_error(y_test_actual, y_test_pred_transfer_actual)
rmse_transfer = np.sqrt(mean_squared_error(y_test_actual, y_test_pred_transfer_actual))
mape_transfer = mean_absolute_percentage_error(y_test_actual, y_test_pred_transfer_actual) * 100

print(f"Transfer Learning Model Performance:")
print(f"  MAE:  ${mae_transfer:.2f}")
print(f"  RMSE: ${rmse_transfer:.2f}")
print(f"  MAPE: {mape_transfer:.2f}%")
```

In [ ]:
```
#Creating ensemble prediction by averaging top 3 models

# Get predictions from all models
all_predictions = []
for result in results:
    all_predictions.append(result['Predictions'].flatten())
```

In [ ]:
```
# Average ensemble
ensemble_pred = np.mean(all_predictions, axis=0).reshape(-1, 1)
```

In [ ]:
```
# Weighted ensemble (weight by inverse RMSE)
weights = 1 / results_df['RMSE'].values
weights = weights / weights.sum()
weighted_ensemble_pred = np.average(all_predictions, axis=0, weights=weights).reshape(-1, 1)
```

In [ ]:
```
# Evaluate ensemble
mae_ensemble = mean_absolute_error(y_test_actual, ensemble_pred)
rmse_ensemble = np.sqrt(mean_squared_error(y_test_actual, ensemble_pred))
mape_ensemble = mean_absolute_percentage_error(y_test_actual, ensemble_pred) * 100

mae_weighted = mean_absolute_error(y_test_actual, weighted_ensemble_pred)
rmse_weighted = np.sqrt(mean_squared_error(y_test_actual, weighted_ensemble_pred))
mape_weighted = mean_absolute_percentage_error(y_test_actual, weighted_ensemble_pred) * 100

print(f"Average Ensemble Performance:")
print(f"  MAE:  ${mae_ensemble:.2f}")
print(f"  RMSE: ${rmse_ensemble:.2f}")
```

```
print(f"  RMSE: ${rmse_ensemble:.2f}")
print(f"  MAPE: {mape_ensemble:.2f}%")

print(f"\nWeighted Ensemble Performance:")
print(f"  MAE:  ${mae_weighted:.2f}")
print(f"  RMSE: ${rmse_weighted:.2f}")
print(f"  MAPE: {mape_weighted:.2f}%")
```

In [ ]:

```
# Final comparison plot
plt.figure(figsize=(16, 8))
plt.plot(test_dates, y_test_actual, label='Actual Price', linewidth=2.5, color='blue')
plt.plot(test_dates, best_predictions, label=f'Best Model ({best_model_name})',
         linewidth=2, color='red', alpha=0.7, linestyle='--')
plt.plot(test_dates, weighted_ensemble_pred, label='Weighted Ensemble',
         linewidth=2, color='green', alpha=0.7, linestyle=':')

plt.title('Gold Price Forecasting: Final Comparison', fontsize=16, fontweight='bold')
plt.xlabel('Date', fontsize=12)
plt.ylabel('Price (USD)', fontsize=12)
plt.legend(fontsize=12)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

In [ ]:

```
print("\n" + "=" * 80)
print("DEEP LEARNING ANALYSIS COMPLETE!")
print("=" * 80)
print("\nKey Findings:")
print(f"  - Best Individual Model: {best_model_name}")
print(f"  - Best RMSE: ${results_df.iloc[0]['RMSE']:.2f}")
print(f"  - Ensemble RMSE: ${rmse_weighted:.2f}")
print(f"  - All models successfully trained using backpropagation")
print(f"  - Transfer learning demonstrated with layer freezing")
print(f"  - Advanced architectures (LSTM, GRU, Attention) implemented")
```

In [ ]:

```
.
```

jupyter Final Project Shanmug Last Checkpoint: 2 days ago

File Edit View Run Kernel Settings Help Trusted

Code JupyterLab Python [conda env:base] *

[33]:
```python
#Installing the 'ta' library that I am using for technical indicators
import sys

print("[Cell 1] Checking if 'ta' is available in this environment...")
```

```
[Cell 1] I am checking if 'ta' is available in this environment...
```

[49]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

sns.set(style="whitegrid")
plt.rcParams["figure.figsize"] = (10, 6)
```

[34]:
```python
# importing core libraries that I am using for data handling, modeling, and evaluation
import os
import warnings

import numpy as np
import pandas as pd

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, random_split

from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.utils.class_weight import compute_class_weight

from ta.trend import SMAIndicator, EMAIndicator, MACD
from ta.momentum import RSIIndicator
from ta.volatility import BollingerBands
from ta.volume import OnBalanceVolumeIndicator

from pandas.errors import SettingWithCopyWarning

# simplifying warning output so that I am focusing on important information
warnings.filterwarnings(
    "ignore",
    message="invalid value encountered in",
    category=RuntimeWarning
)
warnings.filterwarnings("ignore", category=SettingWithCopyWarning)
pd.set_option("compute.use_numexpr", False)

# configuring local data paths that I am using for loading the Kaggle CSV files
DATA_DIR = "data"
FILES = {
    "5m":  "XAU_5m_data.csv",
    "15m": "XAU_15m_data.csv",
    "1h":  "XAU_1h_data.csv",
}

# defining core hyperparameters that I am using for the time-series model
WINDOW_SIZE     = 64          # sequence length
LABEL_THRESHOLD = 0.003       # 0.3% move threshold for Up/Down vs Neutral
BATCH_SIZE      = 128
NUM_EPOCHS      = 10
LR              = 1e-4

# detecting the compute device that I am using for training (GPU if available, otherwise CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(f"[Cell 2] I am using device: {device}")

# checking that the expected CSV files are present in the data directory
missing_files = []
for tf, fname in FILES.items():
    path = os.path.join(DATA_DIR, fname)
    if not os.path.isfile(path):
        missing_files.append(path)

if missing_files:
    print("[Cell 2] ⚠️ I am not finding the following required data files:")
    for p in missing_files:
        print("   -", p)
    print("[Cell 2] Please make sure I am seeing these files inside the 'data' folder next to the notebook.")
else:
    print("[Cell 2] ✅ I am finding all expected CSV files in the 'data' directory:")
    for tf, fname in FILES.items():
        print(f"   - {tf} timeframe -> {os.path.join(DATA_DIR, fname)}")

print("[Cell 2] Finishing import and configuration setup.")
```

```
[Cell 2] I am using device: cpu
[Cell 2] ✅ I am finding all expected CSV files in the 'data' directory:
   - 5m timeframe -> data/XAU_5m_data.csv
   - 15m timeframe -> data/XAU_15m_data.csv
   - 1h timeframe -> data/XAU_1h_data.csv
[Cell 2] I am finishing import and configuration setup.
```

[35]:
```python
# defining a function that is computing technical indicators for one timeframe
def compute_technical_indicators(df: pd.DataFrame) -> pd.DataFrame:
    # Computing basic returns that I am using as a primitive feature
    df['return'] = df['Close'].pct_change()
    # computing moving averages that I am using to capture trend
    df['sma_20'] = SMAIndicator(df['Close'], window=20).sma_indicator()
    df['ema_20'] = EMAIndicator(df['Close'], window=20).ema_indicator()
    # computing RSI that I am using to capture momentum
```

```python
    df['rsi'] = RSIIndicator(df['Close'], window=14).rsi()
    # computing MACD that I am using as a trend-momentum hybrid signal
    df['macd'] = MACD(df['Close']).macd_diff()

    # computing Bollinger Bands that I am using to represent volatility
    bb = BollingerBands(df['Close'], window=20, window_dev=2)
    df['bb_bbm'] = bb.bollinger_mavg()
    df['bb_bbh'] = bb.bollinger_hband()
    df['bb_bbl'] = bb.bollinger_lband()

    # computing OBV that I am using to mix price with volume
    df['obv'] = OnBalanceVolumeIndicator(df['Close'], df['Volume']).on_balance_volume()

    # dropping rows with missing indicator values so that the ML model is seeing clean data
    df = df.dropna()
    return df


# defining a function that is normalizing numeric features into [0, 1] using MinMax scaling
def normalize_features(df: pd.DataFrame, feature_cols) -> pd.DataFrame:
    scaler = MinMaxScaler()
    df[feature_cols] = scaler.fit_transform(df[feature_cols])
    return df


# defining a helper that is loading and preparing one timeframe CSV from local disk
def load_and_prepare(file_path: str, timeframe: str):
    df = pd.read_csv(
        file_path,
        sep=';',
        names=['Date', 'Open', 'High', 'Low', 'Close', 'Volume'],
        header=0
    )

    df['Date'] = pd.to_datetime(df['Date'])
    df = df.sort_values('Date').reset_index(drop=True)

    # interpolating missing numeric values over time so that the series is becoming continuous
    df = df.set_index('Date').interpolate(method='linear').reset_index()

    # computing technical indicators for this timeframe
    df = compute_technical_indicators(df)

    # defining the feature set that I am feeding into the neural network
    feature_cols = [
        'Open', 'High', 'Low', 'Close', 'Volume',
        'return', 'sma_20', 'ema_20',
        'rsi', 'macd', 'bb_bbm', 'bb_bbh', 'bb_bbl', 'obv'
    ]

    # scaling the features so that the optimizer is seeing more stable gradients
    df = normalize_features(df, feature_cols)

    return df, feature_cols


print("[Cell 3] I am defining helper functions for technical indicators, normalization, and CSV loading.")
print("[Cell 3] When I call 'load_and_prepare()', I am turning raw candle data into a normalized feature matrix.")
```

```
[Cell 3] I am defining helper functions for technical indicators, normalization, and CSV loading.
[Cell 3] When I call 'load_and_prepare()', I am turning raw candle data into a normalized feature matrix.
```

[36]:

```python
# defining a function that is creating 3-class labels from future 1-step returns
def create_labels(df: pd.DataFrame, threshold: float = LABEL_THRESHOLD) -> pd.DataFrame:
    """
    3-class labeling based on future 1-step Close return:
    0 = Down, 1 = Neutral, 2 = Up
    """
    df = df.copy()

    # computing the future return and shifting it back so that each row is predicting the next step
    df['target'] = df['Close'].pct_change(periods=1).shift(-1)
    df['target'] = df['target'].replace([np.inf, -np.inf], np.nan)

    target = df['target'].to_numpy()
    labels = np.full_like(target, np.nan, dtype=float)

    valid_mask = ~np.isnan(target)
    valid_target = target[valid_mask]

    labels_valid = np.full_like(valid_target, np.nan, dtype=float)

    # mapping strong positive moves to the 'Up' class
    labels_valid[valid_target >  threshold] = 2
    # mapping strong negative moves to the 'Down' class
    labels_valid[valid_target < -threshold] = 0
    # mapping small moves into the 'Neutral' middle class
    neutral_mask = np.isnan(labels_valid)
    labels_valid[neutral_mask] = 1

    labels[valid_mask] = labels_valid
    df['Label'] = labels

    # dropping rows with missing labels so that the model is seeing only valid targets
    df = df.dropna(subset=['Label'])
    df['Label'] = df['Label'].astype(int)
    return df


# defining a function that is building a unified multi-timeframe dataset from 5m, 15m, and 1h data
def build_dataset_lazy():
    """Load 5m/15m/1h CSVs from local disk and merge into one DataFrame with labels."""
    path_5m  = os.path.join(DATA_DIR, FILES["5m"])
    path_15m = os.path.join(DATA_DIR, FILES["15m"])
    path_1h  = os.path.join(DATA_DIR, FILES["1h"])

    # loading and preparing each timeframe independently
    df_5m,  feats_5m  = load_and_prepare(path_5m,  '5m')
```

```python
    df_15m, feats_15m = load_and_prepare(path_15m, '15m')
    df_1h,  feats_1h  = load_and_prepare(path_1h,  '1h')

    # creating classification labels on the 5m anchor timeframe
    df_5m = create_labels(df_5m, threshold=LABEL_THRESHOLD)

    # aligning minimal dates so that each timeframe is having at least WINDOW_SIZE history
    min_date = max(
        df_5m['Date'].min()  + pd.Timedelta(minutes=WINDOW_SIZE * 5),
        df_15m['Date'].min() + pd.Timedelta(minutes=WINDOW_SIZE * 15),
        df_1h['Date'].min()  + pd.Timedelta(hours=WINDOW_SIZE)
    )

    df_5m = df_5m[df_5m['Date'] >= min_date]

    df_15m = df_15m.set_index('Date')
    df_1h  = df_1h.set_index('Date')
    df     = df_5m.set_index('Date')

    # I am joining 15m and 1h features on the 5m timestamps
    df = df.join(df_15m[feats_15m], on='Date', rsuffix='_15m')
    df = df.join(df_1h[feats_1h],  on='Date', rsuffix='_1h')

    # I am dropping remaining NaNs to keep the training matrix dense
    df = df.dropna()

    # I am constructing a flattened feature list that I am feeding into the model
    all_features = (
        feats_5m +
        [f"{f}_15m" for f in feats_15m] +
        [f"{f}_1h"  for f in feats_1h]
    )

    return df.reset_index(), all_features


print("[Cell 4] I am defining the labeling logic and the multi-timeframe dataset builder.")
print("[Cell 4] When I call 'build_dataset_lazy()', I am fusing 5m/15m/1h candles into one labeled feature table.")
```

```
[Cell 4] I am defining the labeling logic and the multi-timeframe dataset builder.
[Cell 4] When I call 'build_dataset_lazy()', I am fusing 5m/15m/1h candles into one labeled feature table.
```

[37]:
```python
# I am defining a Dataset class that is generating sliding windows lazily for sequence modeling
class LazyGoldDataset(Dataset):
    def __init__(self, df: pd.DataFrame, features, label_col='Label', window_size: int = WINDOW_SIZE):
        self.features    = features
        self.label_col   = label_col
        self.window_size = window_size

        # I am caching the underlying arrays for efficient sampling
        self.df = df.reset_index(drop=True)
        self.X  = self.df[features].astype(np.float32).values
        self.y  = self.df[label_col].astype(np.int64).values

        # I am defining valid indices where a full window is available
        self.valid_range = range(window_size, len(self.df))

    def __len__(self):
        return len(self.valid_range)

    def __getitem__(self, idx):
        idx = idx + self.window_size
        x_window = self.X[idx - self.window_size:idx]
        y_target = self.y[idx]

        return torch.tensor(x_window), torch.tensor(y_target)


print("[Cell 5] I am defining a lazy PyTorch Dataset that is turning the table into [window_size, features] sequences.")
print("[Cell 5] Each sample that I am yielding is a small time-series window with a single classification target.")
```

```
[Cell 5] I am defining a lazy PyTorch Dataset that is turning the table into [window_size, features] sequences.
[Cell 5] Each sample that I am yielding is a small time-series window with a single classification target.
```

[38]:
```python
# I am defining a hybrid neural network that is combining CNN, Transformer, and attention for time-series classification
class GoldHybridModel(nn.Module):
    def __init__(self, input_dim, d_model=64, nhead=4, n_classes=3):
        super().__init__()

        # I am using a temporal CNN that is extracting local patterns across the sequence
        self.cnn = nn.Sequential(
            nn.Conv1d(input_dim, d_model, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv1d(d_model, d_model, kernel_size=3, padding=1),
            nn.ReLU()
        )

        # I am creating learnable positional embeddings for each time step in the window
        self.pos = nn.Parameter(torch.randn(WINDOW_SIZE, d_model))

        # I am building a Transformer encoder that is modeling long-range dependencies
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            batch_first=True
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=2)

        # I am defining an attention pooling layer that is learning which time steps matter most
        self.pool = nn.Sequential(
            nn.Linear(d_model, 1),
            nn.Softmax(dim=1)
        )

        # I am defining a classifier head that is mapping the pooled representation into class logits
        self.classifier = nn.Sequential(
            nn.Linear(d_model, 64),
            nn.ReLU()
```

```python
            nn.ReLU(),
            nn.Linear(64, n_classes)
        )

    def forward(self, x):
        # x: [batch, seq_len, input_dim]
        # I am applying the CNN over the feature dimension
        x = self.cnn(x.permute(0, 2, 1))     # [batch, d_model, seq_len]
        x = x.permute(0, 2, 1)               # [batch, seq_len, d_model]

        # I am adding positional embeddings so that the Transformer is seeing ordering information
        x = x + self.pos

        # I am passing the sequence through the Transformer encoder
        x = self.transformer(x)

        # I am computing attention weights and pooling across the time dimension
        weights = self.pool(x)               # [batch, seq_len, 1]
        x = torch.sum(weights * x, dim=1)    # [batch, d_model]

        # I am computing class logits for the three movement categories
        logits = self.classifier(x)
        return logits, None


# I am defining Focal Loss that I am using to focus learning on harder, less frequent classes
class FocalLoss(nn.Module):
    def __init__(self, gamma=2.0, weight=None):
        super().__init__()
        self.gamma  = gamma
        self.weight = weight

    def forward(self, inputs, targets):
        ce_loss = F.cross_entropy(
            inputs,
            targets,
            weight=self.weight,
            reduction='none'
        )
        pt = torch.exp(-ce_loss)
        return ((1 - pt) ** self.gamma * ce_loss).mean()


print("[Cell 6] I am defining the GoldHybridModel (CNN + Transformer + attention) and FocalLoss.")
print("[Cell 6] This model that I am using is treating each window as a short time-series and learning to classify moves.")
```

```
[Cell 6] I am defining the GoldHybridModel (CNN + Transformer + attention) and FocalLoss.
[Cell 6] This model that I am using is treating each window as a short time-series and learning to classify moves.
```

[39]:

```python
# I am defining a hybrid neural network that is combining CNN, Transformer, and attention for time-series classification
class GoldHybridModel(nn.Module):
    def __init__(self, input_dim, d_model=64, nhead=4, n_classes=3):
        super().__init__()

        # I am using a temporal CNN that is extracting local patterns across the sequence
        self.cnn = nn.Sequential(
            nn.Conv1d(input_dim, d_model, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv1d(d_model, d_model, kernel_size=3, padding=1),
            nn.ReLU()
        )

        # I am creating learnable positional embeddings for each time step in the window
        self.pos = nn.Parameter(torch.randn(WINDOW_SIZE, d_model))

        # I am building a Transformer encoder that is modeling long-range dependencies
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            batch_first=True
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=2)

        # I am defining an attention pooling layer that is learning which time steps matter most
        self.pool = nn.Sequential(
            nn.Linear(d_model, 1),
            nn.Softmax(dim=1)
        )

        # I am defining a classifier head that is mapping the pooled representation into class logits
        self.classifier = nn.Sequential(
            nn.Linear(d_model, 64),
            nn.ReLU(),
            nn.Linear(64, n_classes)
        )

    def forward(self, x):
        # x: [batch, seq_len, input_dim]
        # I am applying the CNN over the feature dimension
        x = self.cnn(x.permute(0, 2, 1))     # [batch, d_model, seq_len]
        x = x.permute(0, 2, 1)               # [batch, seq_len, d_model]

        # I am adding positional embeddings so that the Transformer is seeing ordering information
        x = x + self.pos

        # I am passing the sequence through the Transformer encoder
        x = self.transformer(x)

        # I am computing attention weights and pooling across the time dimension
        weights = self.pool(x)               # [batch, seq_len, 1]
        x = torch.sum(weights * x, dim=1)    # [batch, d_model]

        # I am computing class logits for the three movement categories
        logits = self.classifier(x)
        return logits, None


# I am defining Focal Loss that I am using to focus learning on harder, less frequent classes
class FocalLoss(nn.Module):
```

```python
class FocalLoss(nn.Module):
    def __init__(self, gamma=2.0, weight=None):
        super().__init__()
        self.gamma  = gamma
        self.weight = weight

    def forward(self, inputs, targets):
        ce_loss = F.cross_entropy(
            inputs,
            targets,
            weight=self.weight,
            reduction='none'
        )
        pt = torch.exp(-ce_loss)
        return ((1 - pt) ** self.gamma * ce_loss).mean()


print("[Cell 6] I am defining the GoldHybridModel (CNN + Transformer + attention) and FocalLoss.")
print("[Cell 6] This model that I am using is treating each window as a short time-series and learning to classify moves.")
```

```
[Cell 6] I am defining the GoldHybridModel (CNN + Transformer + attention) and FocalLoss.
[Cell 6] This model that I am using is treating each window as a short time-series and learning to classify moves.
```

•[40]:

```python
# building the merged multi-timeframe dataset from the local CSV files
df, features = build_dataset_lazy()

print(f"[Cell 7] I am building the unified dataset and I am ending up with {len(df)} rows.")
print("[Cell 7] I am inspecting the label distribution (0=Down, 1=Neutral, 2=Up):")
print(df['Label'].value_counts().sort_index())

# wrapping the dataset into a lazy sliding-window Dataset object
dataset = LazyGoldDataset(df, features, window_size=WINDOW_SIZE)
print(f"[Cell 7] I am creating the LazyGoldDataset and I am getting {len(dataset)} time-series samples.")
print(f"[Cell 7] I am using {len(features)} features per time step.")

input_dim = len(features)

# splitting the dataset into train and validation subsets
train_size = int(0.8 * len(dataset))
val_size   = len(dataset) - train_size
train_set, val_set = random_split(dataset, [train_size, val_size])

print(f"[Cell 7] splitting the dataset into {train_size} training samples and {val_size} validation samples.")

# creating DataLoaders that I am using for mini-batch training
train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True)
val_loader   = DataLoader(val_set,   batch_size=BATCH_SIZE)

print(f"[Cell 7] I am using batch size {BATCH_SIZE} for training and validation.")

# instantiating the hybrid model and moving it to the chosen device
model = GoldHybridModel(input_dim=input_dim).to(device)
print(f"[Cell 7] initializing GoldHybridModel with input_dim={input_dim} and I am placing it on {device}.")

# computing class weights for Focal Loss to handle any class imbalance
labels_for_weights = df['Label'].iloc[WINDOW_SIZE:].values
unique_classes = np.unique(labels_for_weights)
class_weights = compute_class_weight(
    class_weight='balanced',
    classes=unique_classes,
    y=labels_for_weights
)

print("[Cell 7] I am computing class weights for the observed classes:")
for c, w in zip(unique_classes, class_weights):
    print(f"   class {c}: weight {w:.4f}")

criterion = FocalLoss(
    weight=torch.tensor(class_weights, dtype=torch.float32).to(device)
)
optimizer = torch.optim.Adam(model.parameters(), lr=LR)

print(f"[Cell 7] I am creating the optimizer (Adam) with learning rate {LR}.")
print("[Cell 7] The training pipeline that I am using is now fully initialized.")
```

```
[Cell 7] I am building the unified dataset and I am ending up with 116354 rows.
[Cell 7] I am inspecting the label distribution (0=Down, 1=Neutral, 2=Up):
Label
0      2821
1    110599
2      2934
Name: count, dtype: int64
[Cell 7] I am creating the LazyGoldDataset and I am getting 116290 time-series samples.
[Cell 7] I am using 42 features per time step.
[Cell 7] I am splitting the dataset into 93032 training samples and 23258 validation samples.
[Cell 7] I am using batch size 128 for training and validation.
[Cell 7] I am initializing GoldHybridModel with input_dim=42 and I am placing it on cpu.
[Cell 7] I am computing class weights for the observed classes:
   class 0: weight 13.8391
   class 1: weight 0.3505
   class 2: weight 13.3345
[Cell 7] I am creating the optimizer (Adam) with learning rate 0.0001.
[Cell 7] The training pipeline that I am using is now fully initialized.
```

•[41]:

```python
# training the model for a fixed number of epochs using mini-batch gradient descent
for epoch in range(1, NUM_EPOCHS + 1):
    model.train()
    total_loss = 0.0
    batch_count = 0

    for xb, yb in train_loader:
        xb, yb = xb.to(device), yb.to(device)

        # performing the forward pass and computing the loss
        logits, _ = model(xb)
        loss = criterion(logits, yb)

        # clearing old gradients, backpropagating, and updating parameters
        optimizer.zero_grad()
```

```python
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        batch_count += 1

    avg_loss = total_loss / max(batch_count, 1)
    print(f"[Cell 8] Epoch {epoch}/{NUM_EPOCHS} - observing average training loss: {avg_loss:.4f}")

print("[Cell 8] I am finishing the training phase with the configured number of epochs.")
```

```
[Cell 8] Epoch 1/10 - I am observing average training loss: 0.6333
[Cell 8] Epoch 2/10 - I am observing average training loss: 0.6182
[Cell 8] Epoch 3/10 - I am observing average training loss: 0.6162
[Cell 8] Epoch 4/10 - I am observing average training loss: 0.6157
[Cell 8] Epoch 5/10 - I am observing average training loss: 0.6142
[Cell 8] Epoch 6/10 - I am observing average training loss: 0.6132
[Cell 8] Epoch 7/10 - I am observing average training loss: 0.6125
[Cell 8] Epoch 8/10 - I am observing average training loss: 0.6119
[Cell 8] Epoch 9/10 - I am observing average training loss: 0.6106
[Cell 8] Epoch 10/10 - I am observing average training loss: 0.6110
[Cell 8] I am finishing the training phase with the configured number of epochs.
```

•[42]:

```python
# Evaluating the trained model on the validation set and computing classification metrics
model.eval()
y_true, y_pred = [], []

with torch.no_grad():
    for xb, yb in val_loader:
        xb = xb.to(device)
        logits, _ = model(xb)
        preds = torch.argmax(logits, dim=1).cpu().numpy()

        y_pred.extend(preds)
        y_true.extend(yb.numpy())

acc = accuracy_score(y_true, y_pred)
print(f"[Cell 9] Computed validation accuracy: {acc:.4f}")

# defining labels and names explicitly so that the report is staying stable even if some classes are rare
labels_order = [0, 1, 2]
target_names = ['Down', 'Neutral', 'Up']

print("\n[Cell 9] Generating a detailed classification report (precision, recall, f1, support):")
print(classification_report(
    y_true,
    y_pred,
    labels=labels_order,
    target_names=target_names,
    zero_division=0
))

print("[Cell 9] Computing the confusion matrix that I am using to inspect class-wise predictions:")
cm = confusion_matrix(y_true, y_pred, labels=labels_order)
print(cm)

print("[Cell 9] Finishing evaluation. we now have a trained multi-timeframe classifier on gold price movements.")
```

```
[Cell 9] Computed validation accuracy: 0.7747

[Cell 9] Generating a detailed classification report (precision, recall, f1, support):
              precision    recall  f1-score   support

        Down       0.27      0.46      0.34       581
     Neutral       0.99      0.79      0.88     22126
          Up       0.05      0.39      0.08       551

    accuracy                           0.77     23258
   macro avg       0.43      0.55      0.43     23258
weighted avg       0.95      0.77      0.85     23258

[Cell 9] Computing the confusion matrix that I am using to inspect class-wise predictions:
[[  265    87   229]
 [  470 17537  4119]
 [  262    72   217]]
[Cell 9] Finishing evaluation. we now have a trained multi-timeframe classifier on gold price movements.
```

[ ]: