DTSA-5511 Introduction to Deep Learning

Final Project: Predicting Stock Prices with Deep Learning

MS-DS, University of Colorado Boulder, peculiar.d@colorado.edu

1. Introduction

It can be challenging for retail investors to participate in the stock market due to the volatility of stock prices and the uncertainty of events, making it difficult to determine the best time to buy or sell. One possible solution is to use a deep learning model to forecast stock price trends (Yuniarty, 2024). Unlike RNNs, which struggle with vanishing gradients, this project applies LSTM and GRU networks, which are more reliable for detecting meaningful patterns in historical stock prices.

2. Project Goal

The main goal of this project is to compare deep learning models such as MLP, LSTM, and GRU using technical indicators like EMA and RSI for stock price prediction. The model with the best performance will undergo hyperparameter tuning and be used to forecast a 14-day stock price. Its performance will be evaluated using various metrics. The aims is to provide more reliable trading signals than traditional methods and improve the return of investments.

3. Project Data

3.1 Data Source

This project uses the SPY fund, which mirrors the S&P 500 index as a measure of overall market performance. Data were downloaded from Yahoo Finance using the `yfinance` Python API and saved locally, allowing the project to restore raw data without API restrictions during development. The dataset covers the period from 2019 to 2025, comprising diverse economic events such as growth, recessions, and the 2021 COVID-19 crash. This five-year dataset allows the model to learn from diverse market conditions, and the recent data provide more reliable predictions than a 30-year dataset.

In [ ]:
```
import numpy as np
import pandas as pd
import yfinance as yf
```

```
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from matplotlib.colors import ListedColormap
from sklearn.preprocessing import MinMaxScaler,StandardScaler
from sklearn.impute import SimpleImputer
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, GRU, Conv1D, Flatten
from sklearn.metrics import mean_squared_error, mean_absolute_error
from tensorflow.keras import Input
```

In [ ]:
```
# Load stock data (Example: S&P 500 stocks)
stocks = ['SPY']

"""
start_date = '2019-01-01'
end_date = '2025-10-23'

##############################################################################################
# This section was commented out as the data was downloaded from Yahoo and saved as
##############################################################################################
# Download stock data
raw_df = yf.download(stocks, start=start_date, end=end_date)


# Save data to CSV so that it can reload without downloading from the source
raw_df.to_csv("stock_data_30_years.csv")

"""
```

3.2 Data Description

- **Rows**: 1,511
- **Columns**: 5 — `Close`, `High`, `Low`, `Open`, `Volume`
- **Data Type**: All columns are stored as objects (strings)
- **Index**: Contains labeled strings like "Ticker" and "Date"
- **Format**: Single-table

There are 1510 non-null rows out of 1511 in the five-year dataset. Conversion is required for price columns to `float64` and `Volume` column to `int`. Non-date items in the index were removed.

In [ ]:
```
# load the data from the save file
# To ensure parsing is consistent, specify index_col=0, parse_dates=True.
spy_data = pd.read_csv('SPY_data_2025_5yrs.csv', index_col=0, parse_dates=True)
```

In [4]:
```
spy_data.info()
spy_data.index
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1511 entries, Ticker to 2025-10-23
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   1510 non-null   object
 1   High    1510 non-null   object
 2   Low     1510 non-null   object
 3   Open    1510 non-null   object
 4   Volume  1510 non-null   object
dtypes: object(5)
memory usage: 70.8+ KB
```

Out[4]:
```
Index(['Ticker', 'Date', '2019-10-23', '2019-10-24', '2019-10-25',
       '2019-10-28', '2019-10-29', '2019-10-30', '2019-10-31', '2019-11-01',
       ...
       '2025-10-10', '2025-10-13', '2025-10-14', '2025-10-15', '2025-10-16',
       '2025-10-17', '2025-10-20', '2025-10-21', '2025-10-22', '2025-10-23'],
      dtype='object', name='Price', length=1511)
```

3.3 Dataset Statistics

The `describe()` function shows the dataset's descriptive statistics. Most prices and volume values are unique over the last five years.

In [5]:
```
spy_data.describe()
```

Out[5]:

	Close	High	Low	Open	Vo
count	1510	1510	1510	1510	
unique	1497	1508	1510	1509	
top	285.3655700683594	283.1889036613753	667.7999877929688	666.8200073242188	656(
freq	2	2	1	2	

3.4 First 5 Rows of the Dataset

Some rows contain the string 'SPY' and null values due to misaligned headers from the 'Ticker' and 'Date' labels. These null rows will be removed during data cleaning.

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

Out[6]:

Price	Close	High	Low	Open	V
Ticker	SPY	SPY	SPY	SPY	
Date	NaN	NaN	NaN	NaN	
2019-10-23	274.2626953125	274.3175675676159	273.0005776236182	273.2109398753011	34
2019-10-24	274.7108459472656	275.3510595128448	273.8785794762566	275.20472418612354	35
2019-10-25	275.8357238769531	276.38447425674264	274.0797282437474	274.134600490666	45

4. Data Cleaning

4.1 Preliminary Findings

- The dataset included both `Ticker` and `Date` labels in the date index column. This resulted in the ticker symbol `"SPY"` and null values appearing in the first and second rows, respectively.
- The `Date`, `Price`, and `Volume` columns were stored as `object` data types instead of `datetime`, `float`, and `int`, respectively.

4.2 Cleaning Steps

4.2.1 Remove Incorrect Header Rows

- The non-data rows were dropped, along with any resulting null rows.

4.2.2 Convert to Numeric

- Price columns (`Open`, `High`, `Low`, `Close`) and `Volume` were converted to numeric using `pd.to_numeric()`.
- The parameter `errors='coerce'` was used to convert non-numeric values into `NaN` for easier handling.

4.2.3 Fix Data Types

- Price columns were converted to `float`.
- `Volume` was converted to `int`.
- `Date` was converted to `datetime` using `pd.to_datetime()`.

In [7]:
```
# 4.2.1 Drop non-numeric index rows 'Ticker' and 'Date'
spy_data = spy_data.iloc[2:]
```

```python
# 4.2.2 Convert all price and volumn columns to numeric
spy_data = spy_data.apply(pd.to_numeric, errors='coerce')

# 4.2.3 Check all the columns data type, and the index again
print(spy_data.columns)
print(spy_data.index)
print(spy_data.head())
print(spy_data.info())
print(spy_data.describe())
print("Check missing values:\n", spy_data.isna().sum())
```

```
Index(['Close', 'High', 'Low', 'Open', 'Volume'], dtype='object')
Index(['2019-10-23', '2019-10-24', '2019-10-25', '2019-10-28', '2019-10-29',
       '2019-10-30', '2019-10-31', '2019-11-01', '2019-11-04', '2019-11-05',
       ...
       '2025-10-10', '2025-10-13', '2025-10-14', '2025-10-15', '2025-10-16',
       '2025-10-17', '2025-10-20', '2025-10-21', '2025-10-22', '2025-10-23'],
      dtype='object', name='Price', length=1509)
                Close        High         Low        Open    Volume
Price
2019-10-23  274.262695  274.317568  273.000578  273.210940  34352200
2019-10-24  274.710846  275.351060  273.878579  275.204724  35453100
2019-10-25  275.835724  276.384474  274.079728  274.134600  45205400
2019-10-28  277.390503  277.893536  277.033833  277.061269  42147000
2019-10-29  277.308167  278.241050  276.988060  277.116114  44284900
<class 'pandas.core.frame.DataFrame'>
Index: 1509 entries, 2019-10-23 to 2025-10-23
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   1509 non-null   float64
 1   High    1509 non-null   float64
 2   Low     1509 non-null   float64
 3   Open    1509 non-null   float64
 4   Volume  1509 non-null   int64
dtypes: float64(4), int64(1)
memory usage: 70.7+ KB
None
             Close         High          Low         Open        Volume
count  1509.000000  1509.000000  1509.000000  1509.000000  1.509000e+03
mean    429.811360   432.228324   426.994531   429.706638  7.932846e+07
std     103.996478   104.031842   103.816145   103.996149  4.008641e+07
min     206.111832   212.333548   201.776039   210.956089  2.027000e+07
25%     363.986877   365.992891   361.386515   364.448089  5.575830e+07
50%     412.011780   414.927223   409.594269   411.879388  7.105290e+07
75%     507.777069   510.742191   503.561679   506.294605  9.078580e+07
max     673.109985   673.950012   669.979980   673.530029  3.922207e+08
Check missing values:
 Close     0
High       0
Low        0
Open       0
Volume     0
dtype: int64
```

4.2.4 Set and Sort Index

- The index column was renamed from `'Price'` to `'Date'` and converted from string to datetime.
- The dataset was sorted by date to ensure chronological order for time-series modeling.

In [8]:
```
# 4.2.4 Convert the index to datetime
spy_data.index = pd.to_datetime(spy_data.index)

# Rename the index to 'Date' for clarity
spy_data.index.name = 'Date'

# Sort index to ensure date is order
spy_data = spy_data.sort_index()

# Check that Date is index now
print(spy_data.columns)
print(spy_data.index)
print(spy_data.head())
print(spy_data.info())
print(spy_data.describe())
print("Check missing values:\n", spy_data.isna().sum())
```

```
Index(['Close', 'High', 'Low', 'Open', 'Volume'], dtype='object')
DatetimeIndex(['2019-10-23', '2019-10-24', '2019-10-25', '2019-10-28',
               '2019-10-29', '2019-10-30', '2019-10-31', '2019-11-01',
               '2019-11-04', '2019-11-05',
               ...
               '2025-10-10', '2025-10-13', '2025-10-14', '2025-10-15',
               '2025-10-16', '2025-10-17', '2025-10-20', '2025-10-21',
               '2025-10-22', '2025-10-23'],
              dtype='datetime64[ns]', name='Date', length=1509, freq=None)
                Close        High         Low        Open    Volume
Date
2019-10-23  274.262695  274.317568  273.000578  273.210940  34352200
2019-10-24  274.710846  275.351060  273.878579  275.204724  35453100
2019-10-25  275.835724  276.384474  274.079728  274.134600  45205400
2019-10-28  277.390503  277.893536  277.033833  277.061269  42147000
2019-10-29  277.308167  278.241050  276.988060  277.116114  44284900
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1509 entries, 2019-10-23 to 2025-10-23
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   1509 non-null   float64
 1   High    1509 non-null   float64
 2   Low     1509 non-null   float64
 3   Open    1509 non-null   float64
 4   Volume  1509 non-null   int64
dtypes: float64(4), int64(1)
memory usage: 70.7 KB
None
             Close         High          Low         Open        Volume
count  1509.000000  1509.000000  1509.000000  1509.000000  1.509000e+03
mean    429.811360   432.228324   426.994531   429.706638  7.932846e+07
std     103.996478   104.031842   103.816145   103.996149  4.008641e+07
min     206.111832   212.333548   201.776039   210.956089  2.027000e+07
25%     363.986877   365.992891   361.386515   364.448089  5.575830e+07
50%     412.011780   414.927223   409.594269   411.879388  7.105290e+07
75%     507.777069   510.742191   503.561679   506.294605  9.078580e+07
max     673.109985   673.950012   669.979980   673.530029  3.922207e+08
Check missing values:
 Close     0
High      0
Low       0
Open      0
Volume    0
dtype: int64
```

4.2.5 Check and Handle Missing Values

- Missing price and volume values usually correspond to non-trading days, such as weekends or holidays.
- Forward-fill (`ffill()`) was applied to ensure no missing rows remained after cleaning.
  - The alternative would be mean imputation; however, since SPY prices have changed from about $250 in 2019 to about 650$ in 2025, the mean value would be unrealistic.

- Forward-fill replaces missing values with the most recent observation, maintaining a more realistic trend.

In [9]:
```
# 4.2.5 Check and Handle Missing Values
print("Check missing values before Forward fill:\n", spy_data.isna().sum())

# Forward fill for consistency
spy_data = spy_data.ffill()

print("Check missing values after Forward Fill:\n", spy_data.isna().sum())
```

```
Check missing values before Forward fill:
 Close     0
High       0
Low        0
Open       0
Volume     0
dtype: int64
Check missing values after Forward Fill:
 Close     0
High       0
Low        0
Open       0
Volume     0
dtype: int64
```

4.2.6 Visual Diagnostics

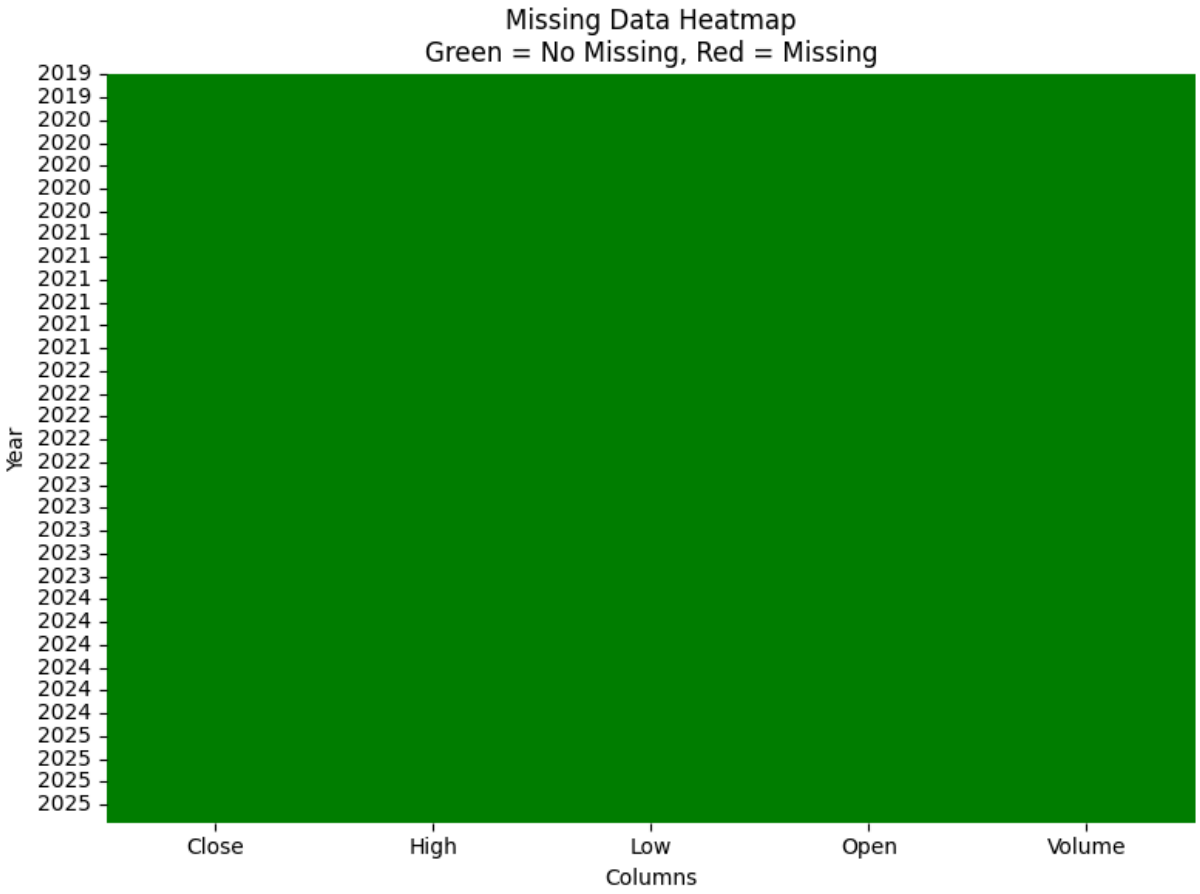
Visual checks were performed to confirm data description and integrity:

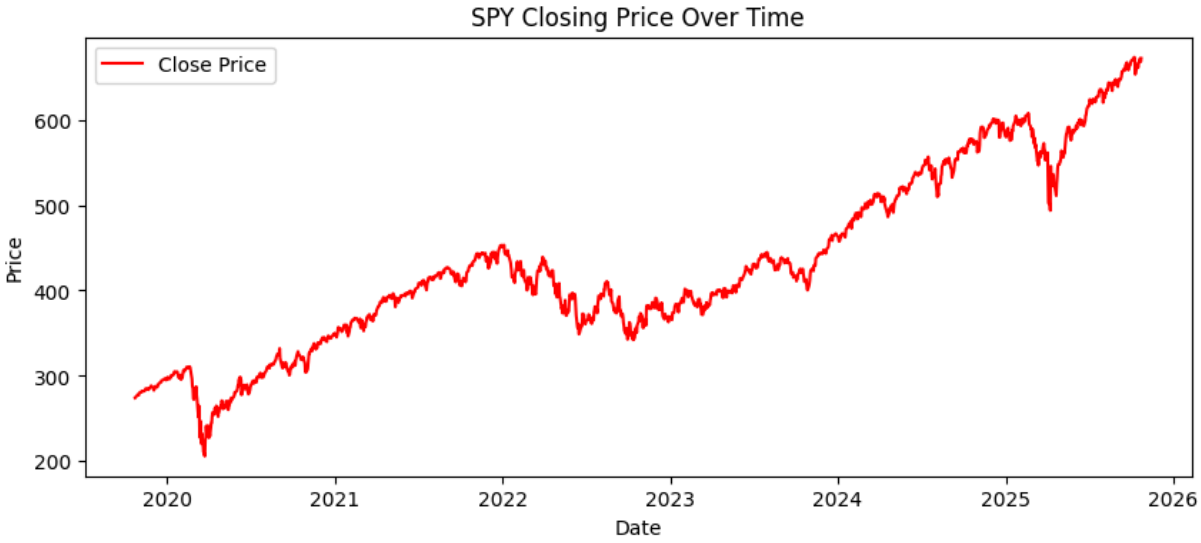
- A tabulated summary showed correct row and column counts with expected data types:
  - Rows: 7,754
  - Columns: 5 — `Close`, `High`, `Low`, `Open`, `Volume`
  - Data Types: All numeric (`float` / `int`), `Date` as `datetime`
  - Index: `Date`
  - Format: Single-table format
- A `sns.heatmap()` of `.isna()` highlighted missing values:
  - Red: Missing data (`True`)
  - Green: No missing data (`False`)
- The SPY price trend plot closely resembled charts from Yahoo Finance (https://finance.yahoo.com/quote/SPY/).

In [10]:
```
# 4.2.6 Sanity Check After Cleaning
print(spy_data.info())
print("Check missing values:\n", spy_data.isna().sum())

##############################
# Plot missing values as a heatmap
# Create custom colormap: green for no missing (False), Red for missing (True)
cmap = ListedColormap(['green', 'red'])
```

```
plt.figure(figsize=(8, 6))
ax = sns.heatmap(spy_data.isna(), cmap=cmap, cbar=False)

# Customize y-axis to show only years
tick_locs = ax.get_yticks()
tick_labels = spy_data.index[tick_locs.astype(int)].year
ax.set_yticklabels(tick_labels)
tick_labels

plt.title('Missing Data Heatmap\nGreen = No Missing, Red = Missing', fontsize=12)
plt.xlabel('Columns')
plt.ylabel('Year')
plt.tight_layout()
plt.show()

##############################

# Line plot of SPY Price over Time
plt.figure(figsize=(10, 4))
plt.plot(spy_data['Close'], label='Close Price', color='red')
plt.title("SPY Closing Price Over Time")
plt.xlabel("Date")
plt.ylabel("Price")
plt.legend()
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1509 entries, 2019-10-23 to 2025-10-23
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Close   1509 non-null   float64
 1   High    1509 non-null   float64
 2   Low     1509 non-null   float64
 3   Open    1509 non-null   float64
 4   Volume  1509 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 70.7 KB
None
Check missing values:
 Close     0
High      0
Low       0
Open      0
Volume    0
dtype: int64
```





4.2.7 Export Cleaned Data

- The cleaned dataset was exported as a `.csv` file named `SPY_data_cleaned.csv` for use in further analysis and modeling.

In [11]:
```
# 4.2.7 Export Cleaned Data
spy_data.to_csv('SPY_data_cleaned.csv', index=True)
```

In [12]:
```
# open the cleaned dataset
spy_data = pd.read_csv('SPY_data_cleaned.csv', index_col=0, parse_dates=True)
```

```
print(spy_data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1509 entries, 2019-10-23 to 2025-10-23
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Close   1509 non-null   float64
 1   High    1509 non-null   float64
 2   Low     1509 non-null   float64
 3   Open    1509 non-null   float64
 4   Volume  1509 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 70.7 KB
None
```

4.3 Data Cleaning Summary

- The dataset contains no null values, the `Date` index is in datetime format, and the data is sorted chronologically — ready for time-series modeling.
- Simple checks using `.info()` and `.isna().sum()` confirmed consistent data types and the absence of missing values.

5. Targets Variables and Features Engineering

5.1 Target Variables

A prediction target is `Close` price.

5.2 Feature Engineering

The features below are commonly used in technical analysis and provide a solid basis for forecasting stock prices. They are derived from historical prices and capture market trends, momentum, and volatility. Further details on the use of these derived features instead of raw SPY prices are provided in the EDA section.

5.2.1 Exponential Moving Average (EMA)

- The EMA calculates the average with more weight on recent values, allowing it to respond to the rapid changes in market prices.
- The Simple Moving Average (SMA), on the other hand, assigns equal weight to each day's value. That's why the EMA is generally a better choice for volatile financial data.

Common EMAs:

- `EMA_20`: 20-day EMA for short-term trends; `Close` below `EMA_20` indicates a downward trend.
- `EMA_40`: 40-day EMA for mid-term trends; `EMA_20` below `EMA_40` indicates a downward trend.

Formula:

$$\text{EMA}_t = \text{Close}_t \times \alpha + \text{EMA}_{t-1} \times (1 - \alpha)$$

Where:

- EMA_t is the EMA value today.
- Close_t is the closing price today.
- EMA_{t-1} is the EMA value yesterday.

- α is the smoothing factor, calculated as $\alpha = \frac{2}{N+1}$, where N is the period length (e.g., 20 days, 40 days).

Pandas `ewm` (Exponential Weighted Moving) function was used to calculate EMA.

In [13]:
```
# Calculate EMA
spy_data['EMA_20'] = spy_data['Close'].ewm(span=20, adjust=False).mean()
spy_data['EMA_40'] = spy_data['Close'].ewm(span=40, adjust=False).mean()
```

5.2.2 Relative Strength Index (RSI)

- RSI is used to evaluate overbought or oversold conditions by measuring the magnitude of recent price changes.

  Formula (14-day RSI):

  $$RS = \frac{\text{Avg Gain}}{\text{Avg Loss}}$$

  $$\text{RSI}_{14} = 100 - \left(\frac{100}{1+RS}\right)$$

In [14]:
```
########## Calculate RSI ###############
# Calculate the daily closing price difference
delta = spy_data['Close'].diff()

# Group gains and losses:
# use clip to filter off -ve delta for the +ve gain and vice versa
# gain = Positive delta only
# loss = Negative delta only
gain = delta.clip(lower=0)
loss = -delta.clip(upper=0)

# Calculate the average gain and average loss over 14 days
avg_gain = gain.rolling(window=14).mean()
avg_loss = loss.rolling(window=14).mean()

# Calculate the Relative Strength (RS)
rs = avg_gain / avg_loss

# Calculate the Relative Strength Index (RSI)
spy_data['RSI_14'] = 100 - (100 / (1 + rs))
```

5.2.3 Daily Returns

- Daily returns represent the percentage change in stock prices using `pct_change()`.
- Returns provide a better comparison of daily gains or losses.
- Transforming prices into returns reduces scaling bias caused by SPY's non-stationary upward trend.

Formula:

$$R_t = \frac{\text{Close}_t - \text{Close}_{t-1}}{\text{Close}_{t-1}}$$

where

- R_t: Daily Return
- $Close_t$: t day closing price
- $Close_{t-1}$: $t-1$ day closing price

In [15]:
```
# Calculate percentage returns
spy_data['Return_1D'] = spy_data['Close'].pct_change()
```

5.2.4 Volatility

- Volatility measures how much stock prices change over time.
- Higher volatility indicates higher risk, while lower volatility suggests lower risk.
- Including volatility as a feature helps capture the inherent price fluctuation risk of the stock.

Formula (rolling 10-day):

$$\text{Volatility}_t = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(r_{t-i} - \bar{r})^2}$$

Where:

- Volatility_t: 10-day volatility value at day t.
- n: Number of days in the rolling window ($n = 10$).
- r_{t-i}: Daily return on a specific day within the window.
- $\bar{r}$: Average daily return over the n-day window.
- t: Current day index.
- i: Day offset within the window.

In [16]:
```
spy_data['Volatility'] = spy_data['Return_1D'].rolling(window=10).std()
```

5.2.5 Volume

- Trading volume indicates how many shares were traded in a day, helping assess market interest in the stock.
- High volume with rising prices signals strong buying interest, while high volume with falling prices indicates selling pressure.
- Volume adds market context not captured by price alone.
- Since volume is already included in the raw data, no calculation is required.

5.3 Dropped Missing Values

- Missing values (`NaN`) occurred during the calculation of returns, EMA, RSI, and volatility due to rolling window computations.
- All rows containing missing values were removed to ensure a clean dataset for analysis and modeling.

In [17]:
```
# Check missing values after creating the features
print("\n\nCheck missing values after creating features:\n", spy_data.isna().sum())

# Drop rows with NA after computing the features
spy_data.dropna(inplace=True)

# Sanity Check missing values again after dropping NA
print("\n\nCheck missing values after dorpping:\n", spy_data.isna().sum())
```

```
Check missing values after creating features:
 Close          0
High            0
Low             0
Open            0
Volume          0
EMA_20          0
EMA_40          0
RSI_14         14
Return_1D       1
Volatility     10
dtype: int64


Check missing values after dorpping:
 Close          0
High            0
Low             0
Open            0
Volume          0
EMA_20          0
EMA_40          0
RSI_14          0
Return_1D       0
Volatility      0
dtype: int64
```

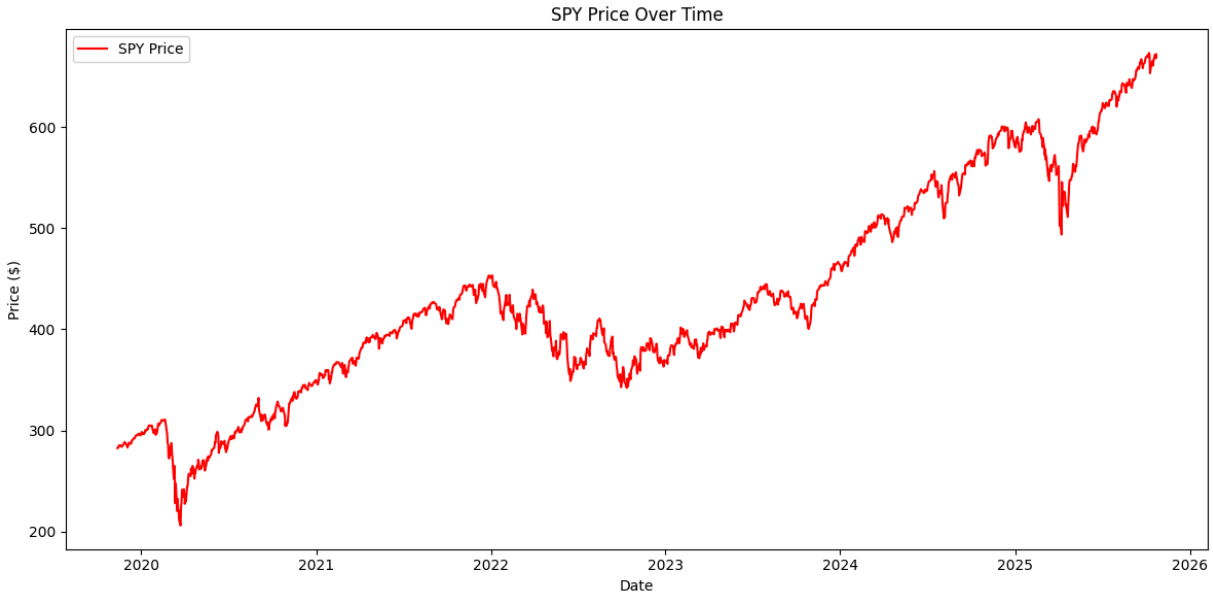
6. Exploratory Data Analysis (EDA)

This section explores the dataset structure, relationships between features, distribution and potential outliers. This ensures the data are relevant and of high quality before input into the machine for training.

6.1 Price Trend Overview

The "SPY Price Over Time" plot shows a strong uptrend of SPY prices with three major dips. This non-stationary data can cause overfitting and unreliable forecasts. To provide more stable patterns for prediction, features like Daily Returns, EMA, and RSI were added. The 'Close' price shows long-term exponential trends, so a log transformation can help smooth it into a more linear curve. This may improve model performance.

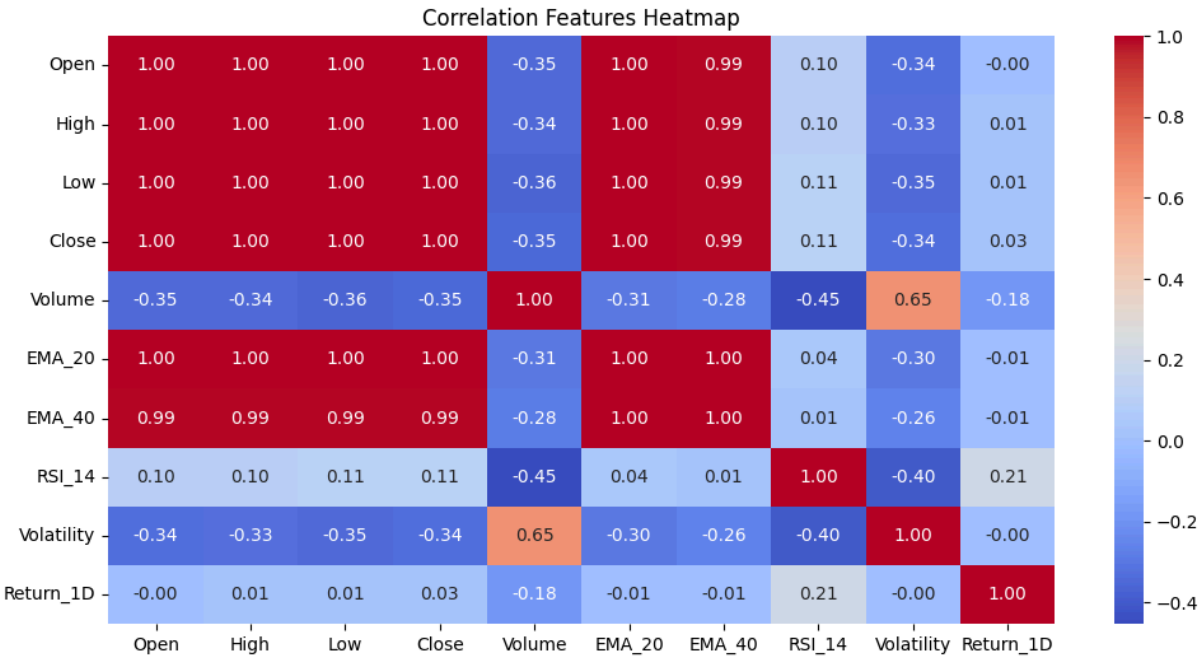
In [18]:
```
# Plot SPY Price Trend Over Time
plt.figure(figsize=(12, 6))
plt.plot(spy_data['Close'], label='SPY Price', color='Red')
plt.title('SPY Price Over Time')
plt.xlabel('Date')
plt.ylabel('Price ($)')
plt.legend()
plt.tight_layout()
plt.show()
```



6.2 Correlation Features Heatmap

A correlation heatmap checked relationships and redundancy among features.

In [19]:
```
#Plot Heatmap Feature Correlation Matrix
plt.figure(figsize=(12, 6))
eval_features = ['Open', 'High', 'Low', 'Close', 'Volume', 'EMA_20', 'EMA_40', 'RSI
sns.heatmap(spy_data[eval_features].corr(), annot=True, fmt=".2f", cmap='coolwarm')
plt.title("Correlation Features Heatmap")
plt.show()
```



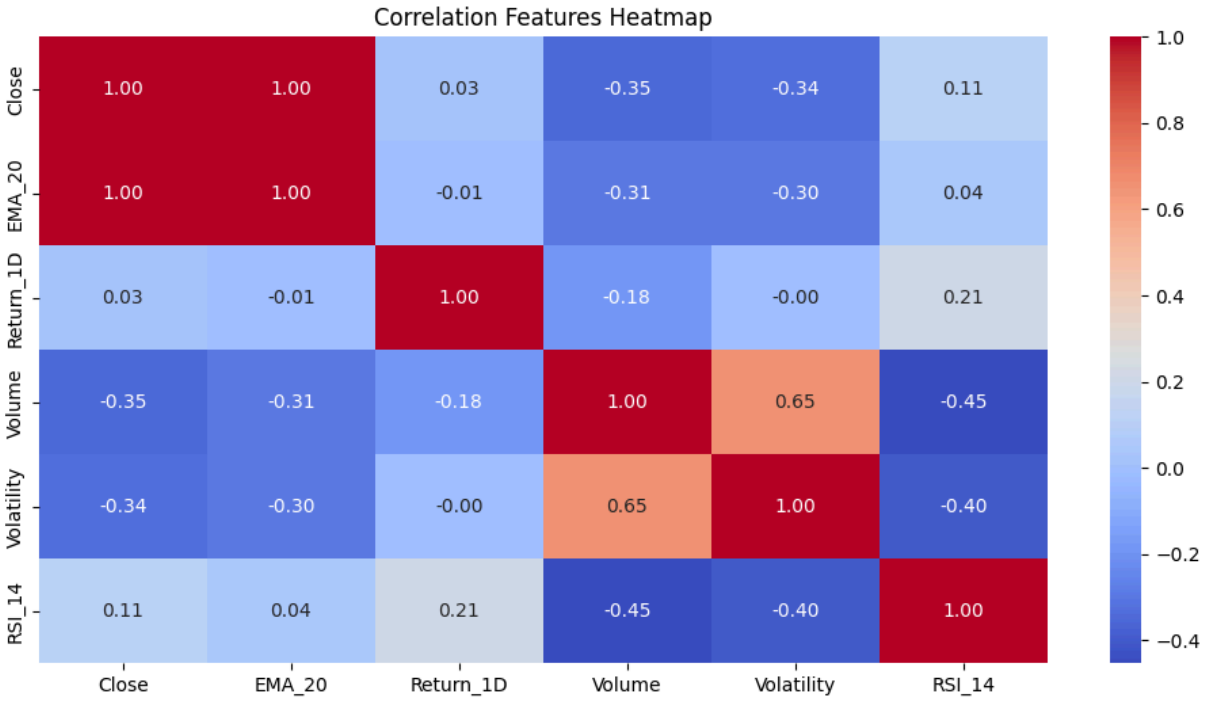
Findings:

- `Open`, `High`, `Low`, `Close`, `EMA_20`, and `EMA_40` are perfectly correlated (1.00) and redundant.
- Compared to linear models, LSTMs and GRUs can handle multicollinearity much better. Using all features is computationally inefficient.
- `Close` is essential for learning sequential changes, and `EMA_20` helps smooth the price trends. Both have been proven important without causing multicollinearity.
- `Return_1D`, `RSI_14`, and `Volatility` have weak correlation with price features, providing independent signals that can improve model generalization and capture patterns missed by price alone.
- `Volume` has a slightly negative correlation with price features, adding unique information for the model.

Selected features are:

- `Close`
- `EMA_20`
- `Return_1D`
- `RSI_14`
- `Volume`
- `Volatility`

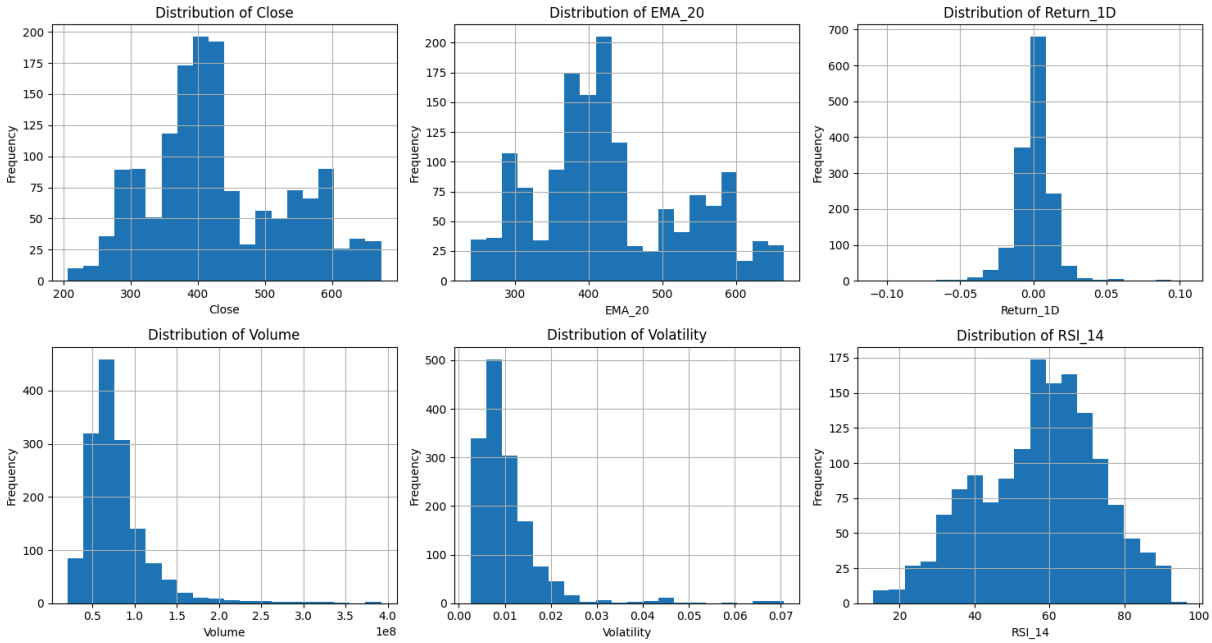
In [20]:
```
#Plot Heatmap Feature Correlation Matrix
plt.figure(figsize=(12, 6))
selected_features = ['Close','EMA_20','Return_1D', 'Volume', 'Volatility', 'RSI_14'
sns.heatmap(spy_data[selected_features].corr(), annot=True, fmt=".2f", cmap='coolwa
plt.title("Correlation Features Heatmap")
plt.show()
```



6.3 Feature Distributions

This section examines feature distributions for skewness and data transformation needs.

In [ ]:
```
# Create Histogram in 2 row and 6 columns

plt.figure(figsize=(15, 8))

plt_cols = (len(selected_features) + 1) //2

for i, feature in enumerate(selected_features):
    plt.subplot(2, plt_cols, i + 1)
    plt.hist(spy_data[feature], bins=20)
    plt.title(f'Distribution of {feature}')
    plt.xlabel(feature)
    plt.ylabel('Frequency')
    plt.grid(True)

# prevent chart overlapping
plt.tight_layout()
plt.show()
```



Findings:

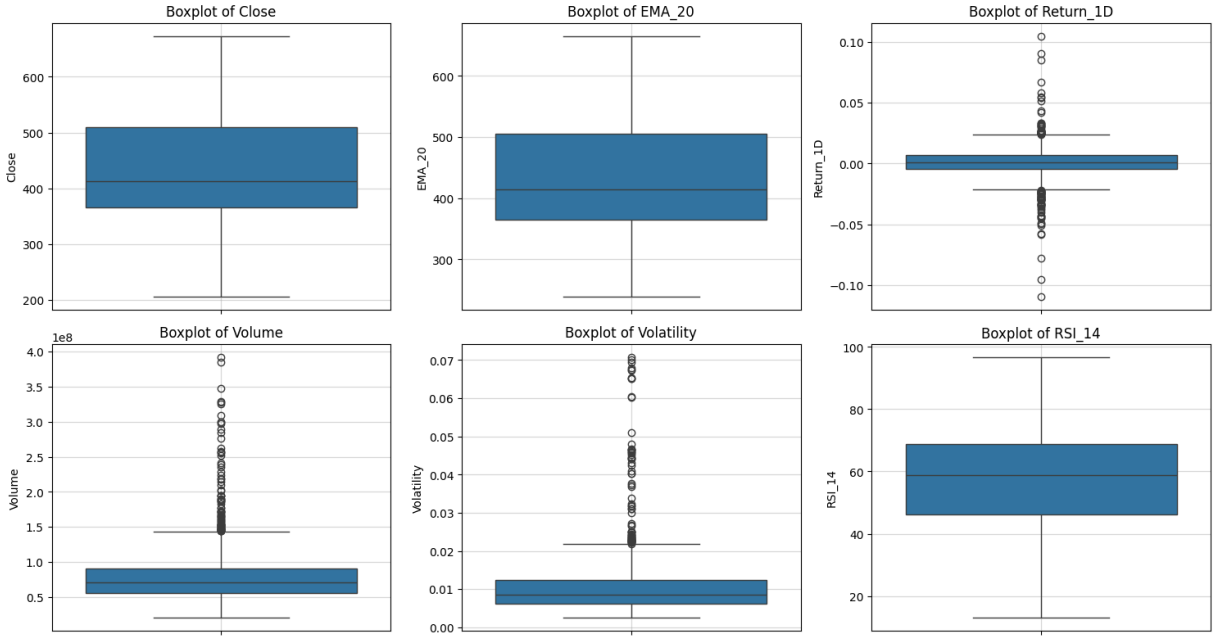
- `Close` and `EMA_20` have three peaks around 300, 400, and 600, showing the stock consolidated at certain levels. Use of MinMax should preserve these peaks while making the data usable for the model.
- `Return_1D` has a narrow, bell-shaped, and symmetrical profile in 0.
- `Volatility` shows a right-skewed distribution but is mostly stable with occasional spikes. Log transformation has minimal effect.
- `Volume` is heavily right-skewed, with most days low but some very high, requiring log transformation to avoid distortion.
- `RSI_14` is roughly bell-shaped and symmetrical, with extremes reflecting overbought or oversold conditions.

6.4 Boxplots for Outlier Detection

This section uses boxplots to identify outliers in selected features that could potentially impact modeling performance.

In [ ]:
```
# create boxplot with subplots
num_features = len(selected_features)
fig, axes = plt.subplots(2, 3, figsize=(15, 8))

# flatten axes for looping
axes = axes.flatten()

for i, feature in enumerate(selected_features):
    sns.boxplot(data=spy_data[feature], ax=axes[i])
```

```
    axes[i].set_title(f'Boxplot of {feature}', fontsize=12)
    axes[i].tick_params(axis='x', rotation=45, labelsize=10)
    axes[i].grid(True, alpha=0.5)

# prevent chart overlapping
plt.tight_layout()
plt.show()
```



Findings:

- `Close` and `EMA_20` boxplots are largely symmetrical around the median, with a slight left skew and no significant outliers.
- `Return_1D` is centered around zero but has significant outliers, representing rare large movements like panic selling.
- `Volume` is usually low, but extreme outliers occur, often linked to major market events; the median is around 0.75×10^8, with outliers above 1.5×10^8.
- `Volatility` is generally stable, with rare spikes beyond 0.02 that indicate important market stress events rather than noise.
- `RSI_14` is symmetrical around the median, with whiskers from roughly 5 to 100 and no significant outliers.

6.5 EDA Summary

- EDA highlighted the dataset's key features and supported the final feature selection.
- Selected features for modeling: `Close`, `EMA_20`, `Return_1D`, `RSI_14`, `Volume`, `Volatility`.

- Long-term growth in SPY prices makes derived indicators like `RSI_14` more suitable than raw prices for accurate predictions.
- Features need preprocessing before training, including log transformation, scaling, and sequencing.
- `Volume` is highly skewed and requires log transformation.
- Apply log transformation to `Close` and `EMA_20` to test if it improves overall performance.

7. Data Preprocessing & Post-Cleaning

Before modeling, selected features were transformed to improve learning and address skew and scale issues. Based on multicollinearity checks, the features retained were `Close`, `EMA_20`, `Return_1D`, `Volume`, `Volatility`, and `RSI_14`.

7.1 Log Transformation

`Volume` was highly right-skewed with extreme outliers. A `log1p` transformation was applied to compress the long tail, improve distribution symmetry, and retain scale integrity.

In [23]:
```
# For sanity check, print out the volumne value before and after Log Transform
print("Before Log Transform:\n ", spy_data['Volume'].describe())

spy_data['Volume_log'] = np.log1p(spy_data['Volume'])

print("\n\nAfter Log Transform:\n ", spy_data['Volume_log'].describe())
```

```
Before Log Transform:
  count    1.495000e+03
mean     7.961621e+07
std      4.014816e+07
min      2.027000e+07
25%      5.600695e+07
50%      7.127560e+07
75%      9.108190e+07
max      3.922207e+08
Name: Volume, dtype: float64


After Log Transform:
  count    1495.000000
mean       18.101341
std         0.409131
min        16.824653
25%        17.840986
50%        18.082065
75%        18.327269
max        19.787335
Name: Volume_log, dtype: float64
```

7.2 Features & Targets Standardization

- Features and targets cover wide value ranges, which can affect model training stability.
- Since LSTMs and GRUs use activation functions like sigmoid and tanh (operating between 0 and 1), all features are scaled using MinMaxScaler.
- Targets are scaled separately to maintain proper scaling and avoid data leakage.
- If the same scaler were used for features and targets, it would cause mismatched distributions between inputs and outputs.

  MinMaxScaler formula:

  $x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$

  where:

  - x': output scaled value
  - x: input value to be transform
  - $x_{\min}$: minimum values
  - $x_{\max}$: maximum values

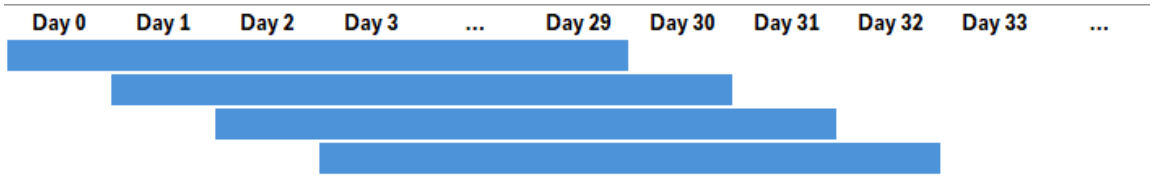
In [ ]:
```
# define features and targets
feature_cols = ['Close', 'EMA_20', 'Return_1D', 'RSI_14', 'Volume_log', 'Volatility
target_cols = ['Close']

# extract raw arrays
X_raw = spy_data[feature_cols].values
y_raw = spy_data[target_cols].values

# split data BEFORE scaling
split_idx = int(len(spy_data) * 0.8)
X_train, X_test = X_raw[:split_idx], X_raw[split_idx:]
y_train, y_test = y_raw[:split_idx], y_raw[split_idx:]

# scale separately
scaler_X = MinMaxScaler(feature_range=(0, 1))
scaler_y = MinMaxScaler(feature_range=(0, 1))

X_train_scaled = scaler_X.fit_transform(X_train)
X_test_scaled  = scaler_X.transform(X_test)

y_train_scaled = scaler_y.fit_transform(y_train)
y_test_scaled  = scaler_y.transform(y_test)
```

7.3 Sliding Window of Data

- LSTM and GRU models learn patterns over short periods within a larger timeframe using a sliding window approach (Brownlee, 2016).
- Each window predicts the next day's return, then moves forward to continue sequential learning.

Example (30-Day Window):

- Predict Day 31 price using Days 1–30
- Predict Day 32 price using Days 2–31



Dataset Construction:

- Use a 30-day window of historical feature data for each sequence:

  X_i = Features[i - window_size : i]

- Pair each sequence with the target return for the next day:

  y_i = Target[i]

In [ ]:
```python
# sequencer() helps to create blocks of data per window_size
# so that LSTM/GRU can learn short-term dependencies
# to predict the next day's result, then continue the forecasting sequentially.
def sequencer(X, y, window_size):
    x_seq, y_seq = [], []
    for i in range(window_size, len(X)):
        #print("i",i)
        x_seq.append(X[i - window_size:i])
        y_seq.append(y[i])
        #print("x_seqi",x_seq,"y_seq", y_seq)
    return np.array(x_seq), np.array(y_seq)

window_size = 30

X_train_seq, y_train_seq = sequencer(X_train_scaled, y_train_scaled, window_size)
X_test_seq, y_test_seq = sequencer(X_test_scaled, y_test_scaled, window_size)
```

7.4 Post-Preprocessing Sanity Check

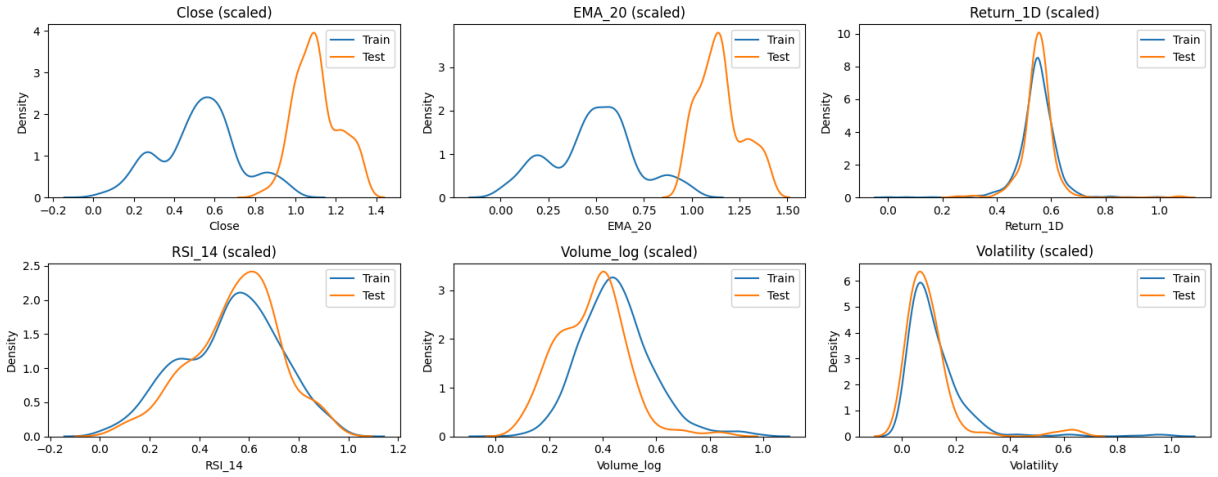
- Visual inspection confirmed no null values, and all features are scaled between 0 and 1.
- `Close` , `EMA_20` Test values slightly exceed Train range, introducing a small positive bias.
- `Return_1D` : Train and Test distributions fully overlap, ideal for stationary series modeling.
- `RSI_14` : Distributions overlap well; Test density slightly wider but peaks align, confirming usability.
- `Volatility` : Nearly identical distributions with minor differences, supporting model stability.

- `Volume_log` : Log transformation normalizes distribution; Test peak higher, reflecting increased trading volume, but shape remains consistent.

In [26]:
```
######################################################################
# Sanity check on result
######################################################################
print("\nSanity check X_train_scaled after Standarization:")
print("Min values:", X_train_scaled.min(axis=0))
print("Max values:", X_train_scaled.max(axis=0))
print("Mean values:", X_train_scaled.mean(axis=0))
print("NaN check:", np.isnan(X_train_scaled).sum())
```

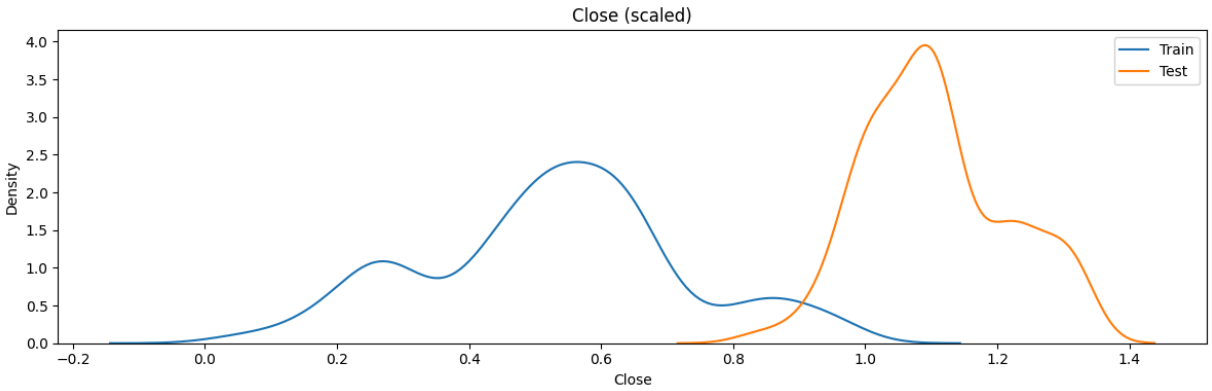
```
Sanity check X_train_scaled after Standarization:
Min values: [0. 0. 0. 0. 0. 0.]
Max values: [1. 1. 1. 1. 1. 1.]
Mean values: [0.52694688 0.4905141  0.55017655 0.5257549  0.44923325 0.12198165]
NaN check: 0
```

In [ ]:
```
# convert to df for easier plotting
X_train_df = pd.DataFrame(X_train_scaled, columns=feature_cols)
X_test_df  = pd.DataFrame(X_test_scaled,  columns=feature_cols)

# plot histogram of scaled features
num_features = len(feature_cols)
rows = 2
cols = (num_features + rows - 1) // rows
fig, axes = plt.subplots(rows, cols, figsize=(15, 6))
axes = axes.flatten()

for i, col in enumerate(feature_cols):
    sns.kdeplot(X_train_df[col], label='Train', ax=axes[i])
    sns.kdeplot(X_test_df[col], label='Test', ax=axes[i])
    axes[i].set_title(f'{col} (scaled)')
    axes[i].legend()

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



In [28]:
```
y_train_df = pd.DataFrame(y_train_scaled, columns=target_cols)
y_test_df  = pd.DataFrame(y_test_scaled,  columns=target_cols)
```

```
fig, axes = plt.subplots(figsize=(12, 4))
for i, col in enumerate(target_cols):
    sns.kdeplot(y_train_df[col], label='Train')
    sns.kdeplot(y_test_df[col], label='Test')
    axes.set_title(f"{col} (scaled)")
    axes.legend()

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



8. Model Architecture

8.1 Model Comparison

Three deep learning models were tested for predicting SPY's daily closing price:

- Multilayer Perceptron (MLP): This model is a baseline model for comparison. This model learns from blocks of past data but does not consider the order of events.
- Long Short-Term Memory (LSTM): This model learns by "memorizing" blocks of data sequentially over time, which allows the model to recognize similar patterns and make more accurate predictions.
- Gated Recurrent Unit (GRU): It is a simplified version of LSTM with the same accuracy of prediction but with fewer parameters, making it faster to train.

In [ ]:
```
######################################################################
# Model Comparison
######################################################################

# mlp model
# dense layer with relu
# optimize with adam to learn faster
# loss with mse to reduce prediction error
window_size = 30
n_features = len(feature_cols)

mlp = Sequential([
```

```
    Flatten(input_shape=(window_size, n_features)),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(1)
])

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

mlp.fit(X_train_seq, y_train_seq, epochs=20, batch_size=16, verbose=0)
mlp_pred = mlp.predict(X_test_seq)

#print(f"MLP Input Shape: {mlp.input_shape}")
#print(f"MLP Output Shape: {mlp.output_shape}")
```

In [ ]:
```
# lstm model
# dense layer output
# optimize with adam to learn faster
# loss with mse to reduce prediction error
lstm = Sequential([
    LSTM(64, return_sequences=False, input_shape=(window_size, n_features)),
    Dense(1)
])
lstm.compile(optimizer='adam', loss='mse')
lstm.fit(X_train_seq, y_train_seq, epochs=20, batch_size=16, verbose=0)
lstm_pred = lstm.predict(X_test_seq)
```

In [ ]:
```
# gru model
# dense layer output
# optimize with adam to learn faster
# loss with mse to reduce prediction error
gru = Sequential([
        Input(shape=(window_size, n_features)),
        GRU(64, return_sequences=False),
        Dense(1)
])
gru.compile(optimizer='adam', loss='mse')
gru.fit(X_train_seq, y_train_seq, epochs=20, batch_size=16, verbose=0)
gru_pred = gru.predict(X_test_seq)
```

8.2 Model Performance Comparison

Selected Features: `['Close', 'EMA_20', 'Return_1D', 'RSI_14', 'Volume_log', 'Volatility']`

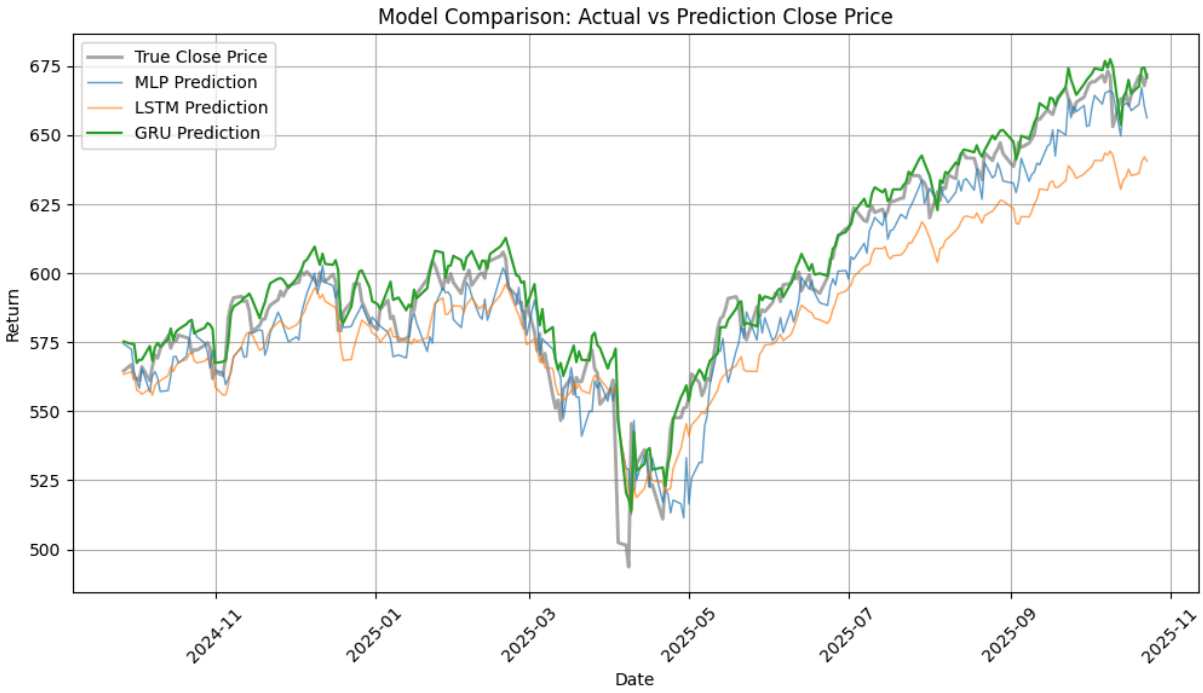
RMSE and MAE metrics were used to measure the model's performance. RMSE is used to detect large errors, and MAE is used to average the prediction error

- Despite the high correlation between `Close` and `EMA_20`, LSTMs and GRUs do not have issues with them.
- GRU model had the lowest RMSE and MAE scores, indicating that it was able to predict with the least amount of errors.

- Actual vs. Predicted charts also confirm that GRU's predictions closely match the real prices.
- Based on these results, GRU will undergo hyperparameter tuning.

In [ ]:
```
# inverse transform the scaled data for plotting the test cycle
y_test_inv = scaler_y.inverse_transform(y_test_scaled)
mlp_pred_inv = scaler_y.inverse_transform(mlp_pred)
lstm_pred_inv = scaler_y.inverse_transform(lstm_pred)
gru_pred_inv = scaler_y.inverse_transform(gru_pred)


# taking in to consideration of the window size
# align actual and predictions to have matching length for plotting
# set y_test_inv wrt window_size
y_test_inv = y_test_inv[window_size:]


# split_idx = int(len(spy_data) * 0.8)
# set test x index wrt y_test_inv
# starting from split_idx + window_size
test_index = spy_data.index[split_idx + window_size : split_idx + window_size + len

#print(len(test_index), len(y_test_inv))
```

In [ ]:
```
# Plot the 3 model for comparison
plt.figure(figsize=(12, 6))
plt.plot(test_index, y_test_inv[:, 0], label='True Close Price', color='gray', line
plt.plot(test_index, mlp_pred_inv[:, 0], label='MLP Prediction', alpha=0.7, linewid
plt.plot(test_index, lstm_pred_inv[:, 0], label='LSTM Prediction', alpha=0.7, linew
plt.plot(test_index, gru_pred_inv[:, 0], label='GRU Prediction')

plt.title('Model Comparison: Actual vs Prediction Close Price')
plt.xticks(rotation=45)
plt.xlabel('Date')
plt.ylabel('Return')
plt.legend()
plt.grid(True)
plt.show()
```



In [ ]:

```python
# calculates the root mean squared error (rmse)
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))

# calculates the mean absolute error (mae)
def mae(y_true, y_pred):
    return mean_absolute_error(y_true, y_pred)

# calculates the mean absolute percentage error (mape)
def mape(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    # prevent division by zero
    epsilon = np.finfo(float).eps
    return np.mean(np.abs((y_true - y_pred) / (y_true + epsilon))) * 100

print(f"Selected Features: {feature_cols}")
print("Model Architecture Comparison (RMSE)")
print(f"MLP RMSE:  {rmse(y_test_inv, mlp_pred_inv):.4f}")
print(f"LSTM RMSE: {rmse(y_test_inv, lstm_pred_inv):.4f}")
print(f"GRU RMSE:  {rmse(y_test_inv, gru_pred_inv):.4f}")

print("\nModel Architecture Comparison (MAE)")
print(f"MLP MAE:   {mae(y_test_inv, mlp_pred_inv):.4f}")
print(f"LSTM MAE:  {mae(y_test_inv, lstm_pred_inv):.4f}")
print(f"GRU MAE:   {mae(y_test_inv, gru_pred_inv):.4f}")
```

```
Selected Features: ['Close', 'EMA_20', 'Return_1D', 'RSI_14', 'Volume_log', 'Volatil
ity']
Model Architecture Comparison (RMSE)
MLP RMSE:  12.8487
LSTM RMSE: 17.0502
GRU RMSE:  8.7818

Model Architecture Comparison (MAE)
MLP MAE:   10.0962
LSTM MAE:  14.6350
GRU MAE:   6.4162
```

8.3 Hyperparameter Tuning

The goal of Hyperparameter Tuning is to test different combinations of parameters and find the configuration that yields the lowest RMSE on unseen data.

- `window_size` and `gru_units` are selected for Hyperparameter Tuning because they are critical for GRU modeling time series data.
- `window_size` defines how many days the model sees to make the next prediction. Testing from short-term 7-day windows up to longer-term 30-day windows.
- `gru_units` is the number of neurons in the hidden layer to memorize and process from the input sequence. 32 and 64 units should be sufficient for capturing basic sequential patterns. Although 128 units can capture more complex patterns, there is a higher risk of overfitting with a smaller dataset.

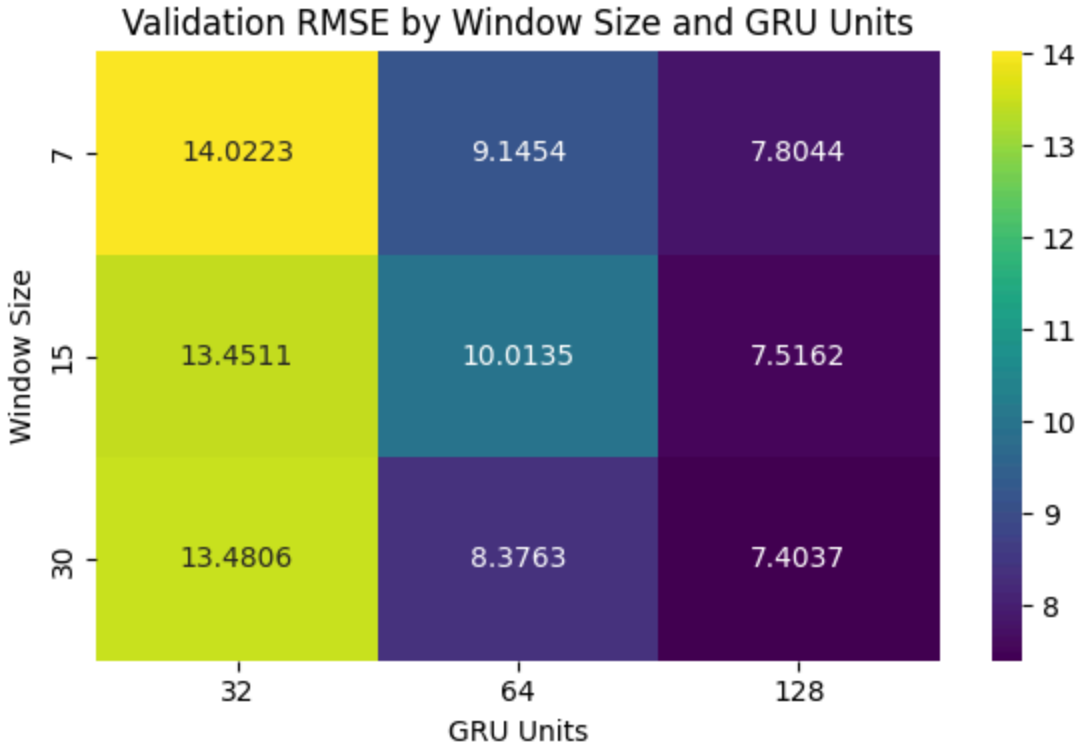
In [ ]:
```
######################################################################
# Hyperparameter Tuning
######################################################################
window_sizes = [7, 15, 30]
gru_units = [32, 64, 128]

results = []
histories = {}

# training loop
for w in window_sizes:
    # get the sequence of scaled data based on windows size
    X_train_seq, y_train_seq = sequencer(X_train_scaled, y_train_scaled, w)
    X_test_seq, y_test_seq   = sequencer(X_test_scaled, y_test_scaled, w)

    print(f"\nwindow {w}: {len(X_test_seq)} test sequences")

    # train gru model
    # dense layer output
    # optimize with adam to learn faster
    # loss with mse to reduce prediction error
    for units in gru_units:
        print(f"Training GRU with window={w}, units={units}")

        model = Sequential([
            Input(shape=(w, n_features)),
```

```
            GRU(units, return_sequences=False),
            Dense(1)
        ])
        model.compile(optimizer='adam', loss='mse')

        # capture history for the plotting
        history = model.fit(
            X_train_seq, y_train_seq,
            epochs=15,
            batch_size=16,
            validation_split=0.1,
            verbose=0
        )

        # save history
        histories[(w, units)] = history

        # predict
        gru_pred_eval = model.predict(X_test_seq, verbose=0)

        # inverse transform scaled values for plotting
        y_test_inv_eval = scaler_y.inverse_transform(y_test_seq)
        gru_pred_inv_eval = scaler_y.inverse_transform(gru_pred_eval)

        # align lengths
        min_len = min(len(y_test_inv_eval), len(gru_pred_inv_eval))
        y_test_inv_eval = y_test_inv_eval[:min_len]
        gru_pred_inv_eval = gru_pred_inv_eval[:min_len]

        # get error score
        training_score_rmse = rmse(y_test_inv_eval, gru_pred_inv_eval)
        training_score_mae = mae(y_test_inv_eval, gru_pred_inv_eval)
        results.append((w, units, training_score_rmse, training_score_mae))
        print(f"Window={w}, Units={units}: RMSE={training_score_rmse:.4f}, MAE={tra

# summarize results
df_results = pd.DataFrame(results, columns=['Window', 'Units', 'RMSE', 'MAE'])
df_results = df_results.sort_values(by=['RMSE','MAE'])

print("\nTop 3 Best Configurations:")
print(df_results.head(3))
```

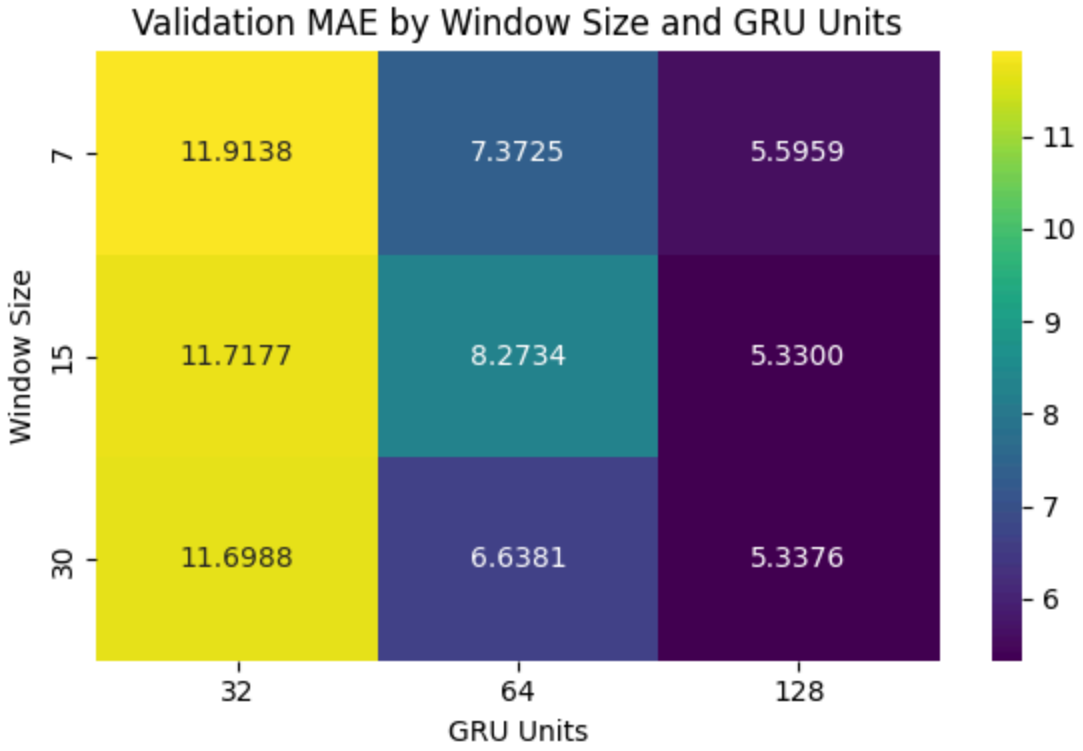
```
window 7: 292 test sequences
Training GRU with window=7, units=32
Window=7, Units=32: RMSE=14.0223, MAE=11.9138
Training GRU with window=7, units=64
Window=7, Units=64: RMSE=9.1454, MAE=7.3725
Training GRU with window=7, units=128
Window=7, Units=128: RMSE=7.8044, MAE=5.5959

window 15: 284 test sequences
Training GRU with window=15, units=32
Window=15, Units=32: RMSE=13.4511, MAE=11.7177
Training GRU with window=15, units=64
Window=15, Units=64: RMSE=10.0135, MAE=8.2734
Training GRU with window=15, units=128
Window=15, Units=128: RMSE=7.5162, MAE=5.3300

window 30: 269 test sequences
Training GRU with window=30, units=32
Window=30, Units=32: RMSE=13.4806, MAE=11.6988
Training GRU with window=30, units=64
Window=30, Units=64: RMSE=8.3763, MAE=6.6381
Training GRU with window=30, units=128
Window=30, Units=128: RMSE=7.4037, MAE=5.3376

Top 3 Best Configurations:
   Window  Units      RMSE       MAE
8      30    128  7.403657  5.337578
5      15    128  7.516152  5.329969
2       7    128  7.804393  5.595905
```

In [ ]:

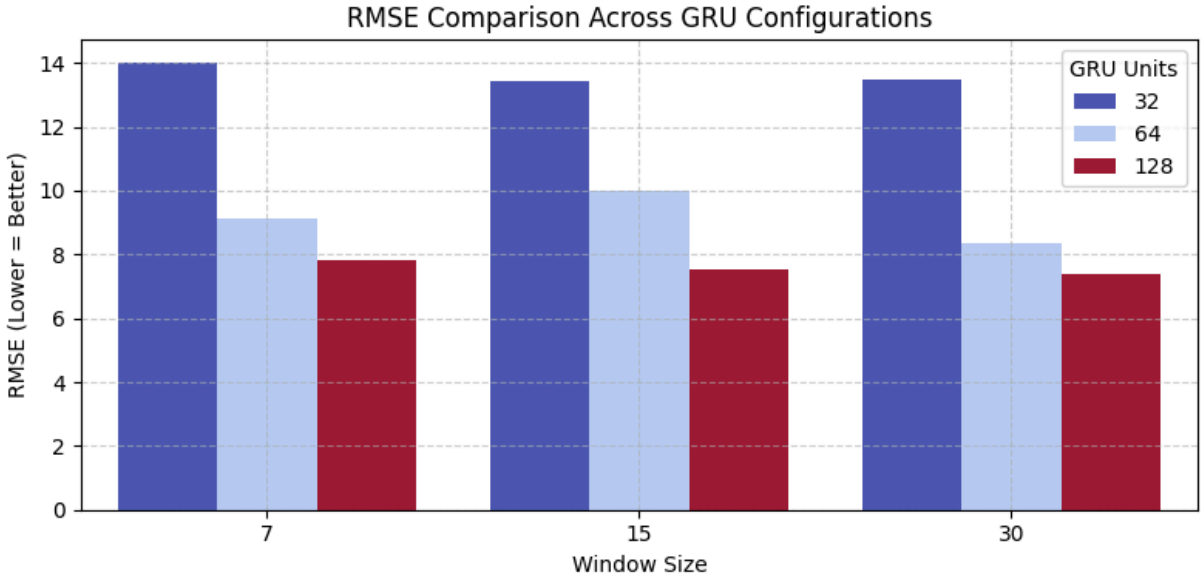
```python
# Validation RMSE Heatmap
pivot = df_results.pivot(index='Window', columns='Units', values='RMSE')
plt.figure(figsize=(6,4))
sns.heatmap(pivot, annot=True, cmap='viridis', fmt=".4f")
plt.title("Validation RMSE by Window Size and GRU Units")
plt.xlabel("GRU Units")
plt.ylabel("Window Size")
plt.tight_layout()
plt.show()
```



In [ ]:
```
# Validation MAE Heatmap
pivot = df_results.pivot(index='Window', columns='Units', values='MAE')
plt.figure(figsize=(6,4))
sns.heatmap(pivot, annot=True, cmap='viridis', fmt=".4f")
plt.title("Validation MAE by Window Size and GRU Units")
plt.xlabel("GRU Units")
plt.ylabel("Window Size")
plt.tight_layout()
plt.show()
```

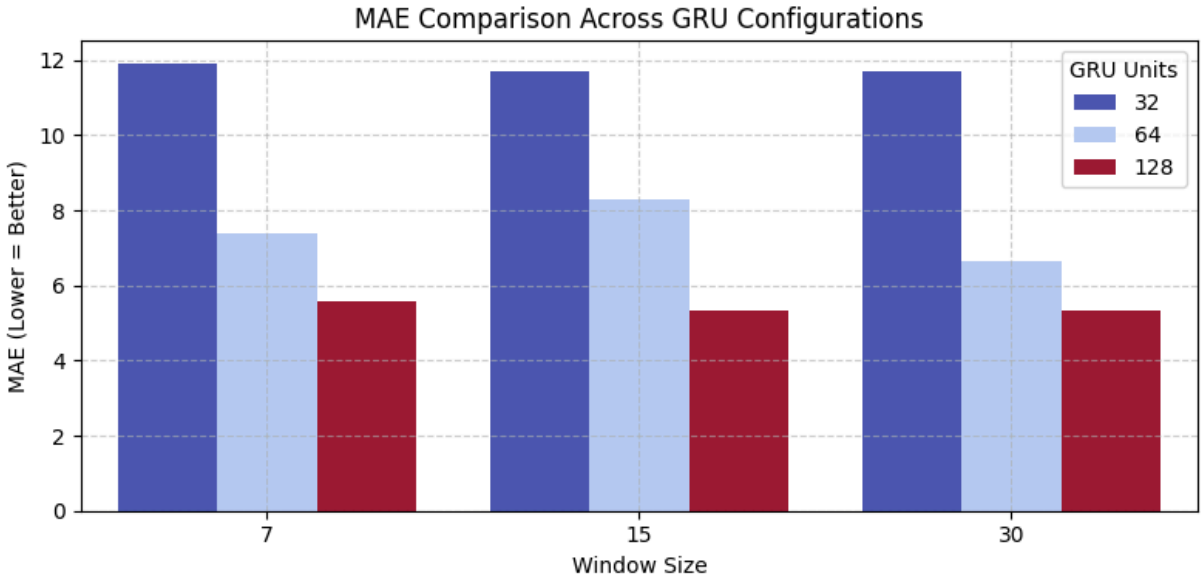


In [ ]:
```
# rmse bar plot
plt.figure(figsize=(8,4))
sns.barplot(data=df_results, x='Window', y='RMSE', hue='Units', palette='coolwarm')
plt.title('RMSE Comparison Across GRU Configurations')
plt.ylabel('RMSE (Lower = Better)')
plt.xlabel('Window Size')
plt.legend(title='GRU Units')
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```



In [ ]:
```
# mae bar plot
plt.figure(figsize=(8,4))
```

```
sns.barplot(data=df_results, x='Window', y='MAE', hue='Units', palette='coolwarm')
plt.title('MAE Comparison Across GRU Configurations')
plt.ylabel('MAE (Lower = Better)')
plt.xlabel('Window Size')
plt.legend(title='GRU Units')
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```

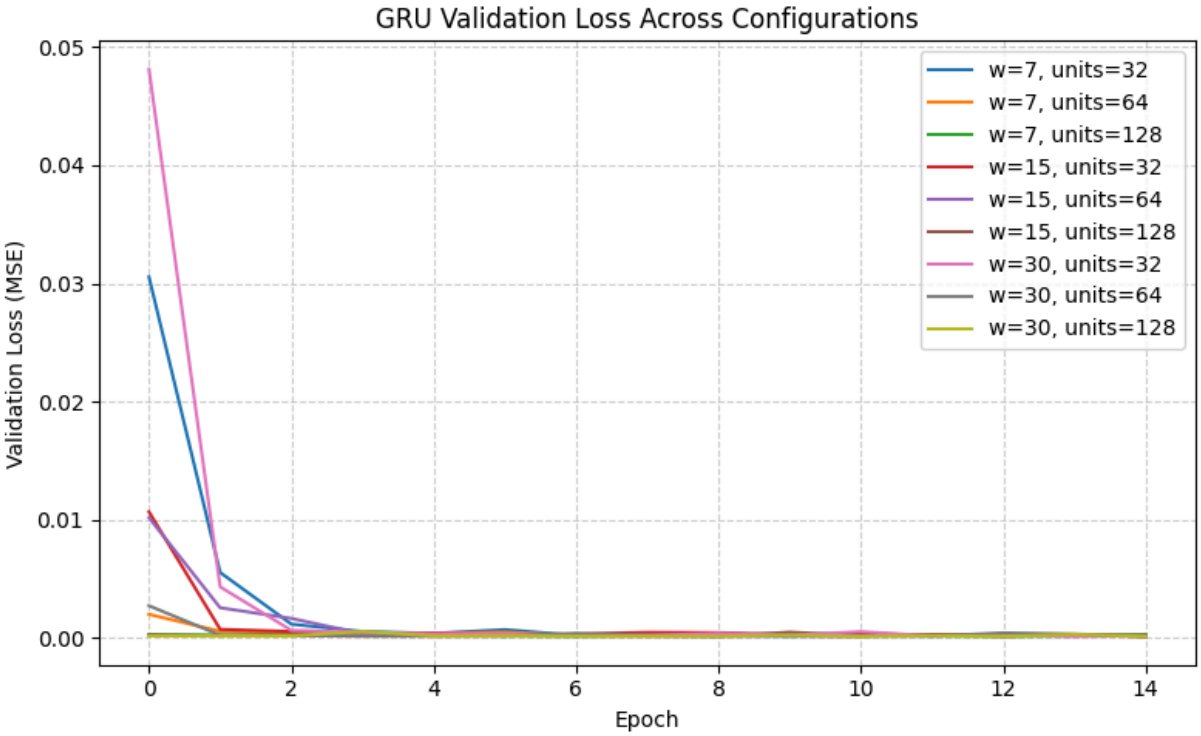


8.4 Validation Loss (MSE)

The Validation Loss plot reveals at what point the training cycle is sufficient and with minimal subsequent improvement:

- All nine GRU configurations demonstrate a sharp decrease in MSE before Epoch 2. This rapid initial learning phase confirms that the model is highly effective at quickly finding a low-error state on the validation data.
- By Epoch 3, all configurations appear to have converged to the lowest possible loss near 0. Training beyond 3 epochs provides negligible improvement and unnecessarily increases training time.

In [ ]:
```
# plot validation loss
plt.figure(figsize=(8,5))
for (w, units), hist in histories.items():
    plt.plot(hist.history['val_loss'], label=f'w={w}, units={units}')
plt.title('GRU Validation Loss Across Configurations')
plt.xlabel('Epoch')
plt.ylabel('Validation Loss (MSE)')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```



8.5 GRU Hyperparameter Tuning Results

Window	Units	RMSE	MAE
30	128	**7.4037**	**5.3376**
15	128	7.5162	5.3300
7	128	7.8044	5.5959

Best Configuration: Window = 30, Units = 128 (lowest RMSE and MAE)

Key Findings:

1. GRU benefits significantly from the longest 30-day window size, suggesting that the model is able to establish the long-term price trend using a larger block of historical data.
2. The best result also indicates that the complexity of price movements requires a high-capacity model with 128 neuron units to effectively map the sequence of the next day's price.

9. Results and Analysis

Selected Features: `Close` , `EMA_20` , `Return_1D` , `RSI_14` , `Volume` , `Volatility`
Targets: `Close`

Summary Model Architecture Comparison

Model	RMSE	MAE
MLP	12.8487	10.0962
LSTM	17.0502	14.6350
GRU	**8.7818**	**6.4162**

Summary of Top 3 Hyperparameter Combination:

Window	Units	RMSE	MAE
30	128	**7.4037**	**5.3376**
15	128	7.5162	5.3300
7	128	7.8044	5.5959

The final model selection is the GRU configured with a 30-day window and 128 units for the final prediction and analysis.

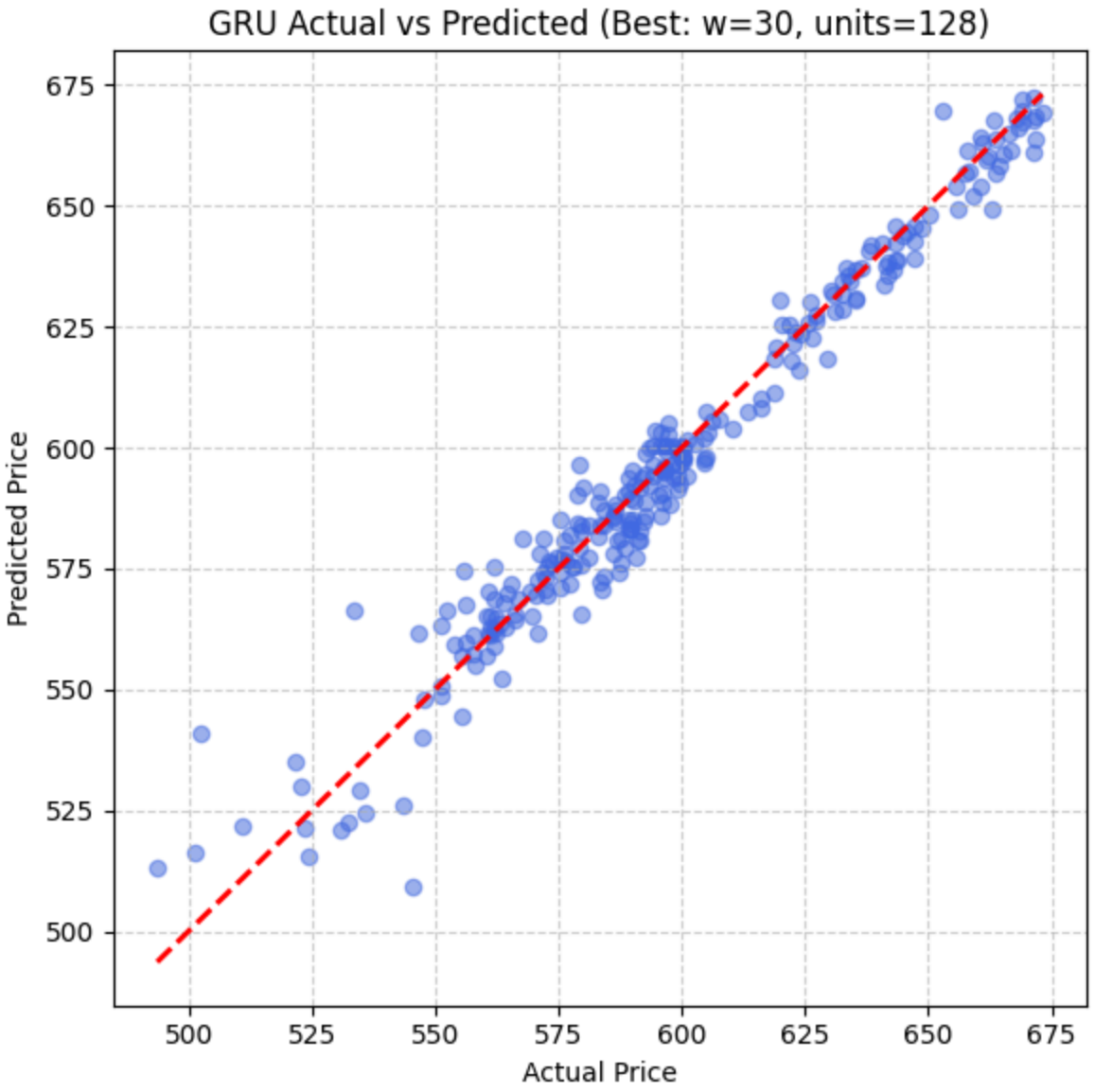
In [ ]:
```
######################################################################
# Train model with best hyperparameter config
######################################################################
# visualization for the best hyperparameter configuration
best_w, best_units, _ ,_ = df_results.iloc[0]
best_w = int(best_w)
best_units = int(best_units)

# train best model with the best hyperparameter configuration
X_train_seq, y_train_seq = sequencer(X_train_scaled, y_train_scaled, int(best_w))
X_test_seq, y_test_seq   = sequencer(X_test_scaled, y_test_scaled, int(best_w))

# train gru model
# dense layer output
# optimize with adam to learn faster
# loss with mse to reduce prediction error
model_best = Sequential([
    Input(shape=(best_w, n_features)),
    GRU(best_units, return_sequences=False),
    Dense(1)
])
model_best.compile(optimizer='adam', loss='mse')
model_best.fit(X_train_seq, y_train_seq, epochs=15, batch_size=16, validation_split

y_pred_best = model_best.predict(X_test_seq, verbose=0)
y_test_inv_best = scaler_y.inverse_transform(y_test_seq)
y_pred_inv_best = scaler_y.inverse_transform(y_pred_best)
```

9.1 Actual vs Predicted Scatter

- The scatter plot visually confirms the RMSE result, showing a high degree of correlation between the predicted and actual stock prices.

- Although some data points are far from the y = x line at the beginning, the points become tighter as the price increases, implying that the prediction is more accurate for higher prices, which are generally more recent than the lower prices.
- Overall, the minimal spread of data points around the y = x line provides evidence that the chosen model architecture and hyperparameters (w = 30, units = 128) deliver an accurate solution for the stock prediction task.
- The scattering of points in the top range appears to be slightly concentrated below the line. This suggests the model tends to predict a value that is slightly lower than the actual price.

In [43]:
```
# Actual vs Predicted Scatter plot
plt.figure(figsize=(6,6))
plt.scatter(y_test_inv_best, y_pred_inv_best, alpha=0.5, color='royalblue')
plt.plot([y_test_inv_best.min(), y_test_inv_best.max()],
         [y_test_inv_best.min(), y_test_inv_best.max()],
         'r--', linewidth=2)
plt.title(f'GRU Actual vs Predicted (Best: w={best_w}, units={best_units})')
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```

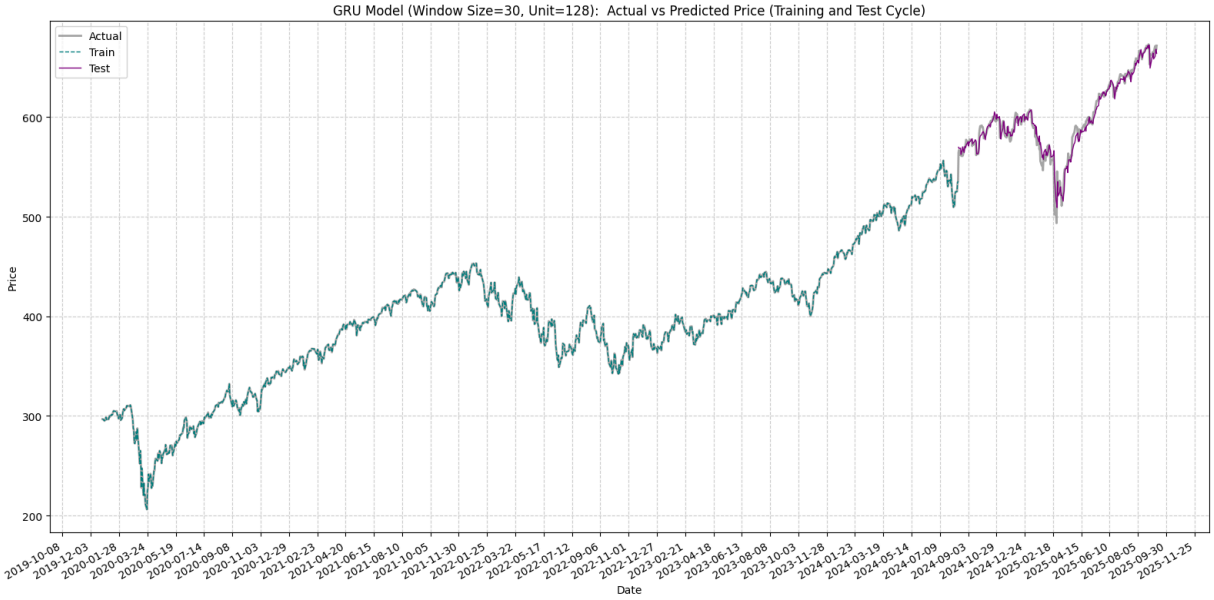


9.2 Actual vs Predicted Price in Full Cycle

- The Actual vs Predicted Price in Training and Test cycle chart provides a full visualization of the GRU model's performance throughout the training and testing cycle.
- Training Phase: The predictions in the training region show a near-perfect alignment with the actual prices, indicating the GRU model is capable of learning the complex patterns from the input features with minimum loss.
- Test Phase: The forecast portion of the chart maintains a high degree of alignment with the actual price trend, validating the low RMSE and MAE achieved on the unseen data.

In [ ]:
```
############################################################
# Plot the Train and Test against the Actual Close Price
############################################################
# inverse transform training predictions
y_train_inv_best = scaler_y.inverse_transform(y_train_seq)

# total number of points to plot (Train + Test)
```

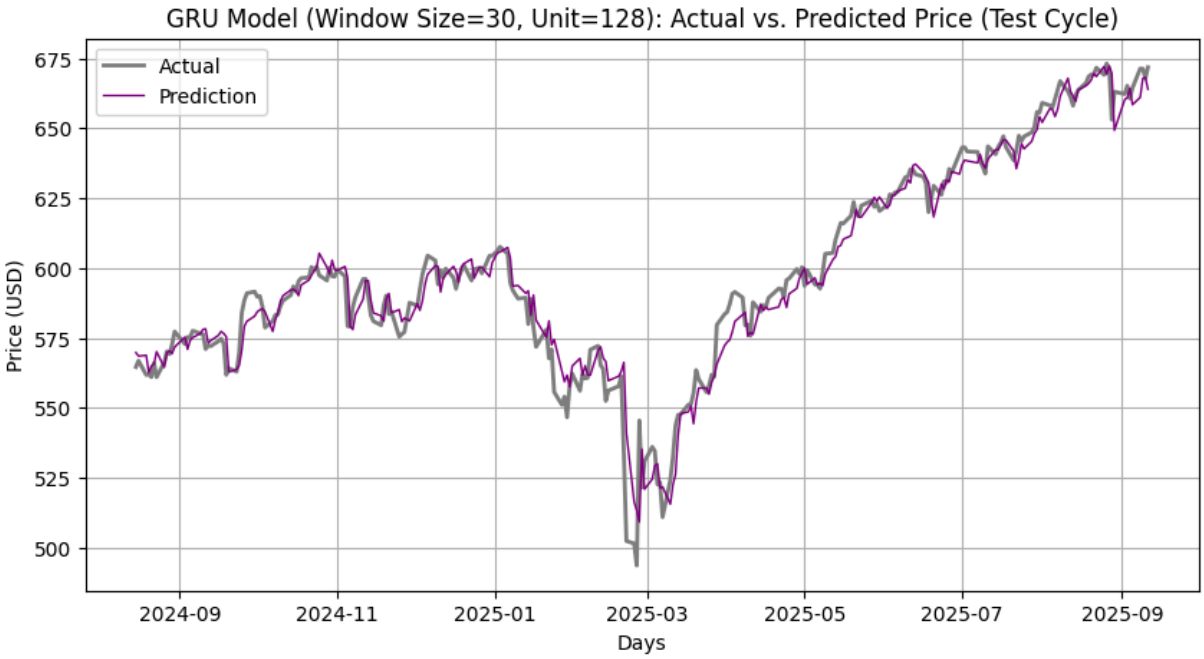
```
train_length = len(y_train_inv_best)
test_length = len(y_test_inv_best)
total_length = train_length + test_length
window_size = best_w

# align dates with the output sequences
full_dates = spy_data.index
sequence_dates = full_dates[window_size : window_size + total_length]

# separate training and testing date ranges
train_dates = sequence_dates[:train_length]
test_dates = sequence_dates[train_length:]

# combine actual prices
y_actual_full = np.concatenate([y_train_inv_best, y_test_inv_best])

# combine predicted prices
y_pred_full = np.concatenate([y_train_inv_best, y_pred_inv_best])


plt.figure(figsize=(16, 8))

# plot the full Actual Close Price
plt.plot(sequence_dates, y_actual_full,
         label='Actual', color='gray', linewidth=2, alpha=0.7)

# plot Predicted Train
plt.plot(train_dates, y_train_inv_best,
         label='Train', color='teal', linestyle='--', linewidth=1)

# plot Predicted Test
plt.plot(test_dates, y_pred_inv_best,
         label='Test', color='purple', linestyle='-', linewidth=1)

# configure the x ticks date interval and date format
plt.gca().xaxis.set_major_locator(mdates.WeekdayLocator(interval=8))
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
plt.gcf().autofmt_xdate()

plt.title(f'GRU Model (Window Size={best_w}, Unit={best_units}):  Actual vs Predict
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()

# show plot
plt.show()
```



9.3 Actual vs. Predicted Price in Test Cycle

- A closer look at the Actual vs. Predicted Price in the Test Cycle chart shows that the prediction line closely tracks the overall trend of the grey actual line throughout the entire test period from 2024 to 2025, correctly capturing periods of both uptrend and downtrend. This implies the model's capability to learn underlying temporal dependencies and predict the long-term direction of stock price movements.
- However, the prediction line appears slightly below and lags behind the actual line. The consistently lower predictions suggest a small negative bias in the model's output, indicating that the model is conservative and tends to underestimate the true values.

In [ ]:
```
############################################################
# Plot actual vs. predicted prices
############################################################
plt.figure(figsize=(10, 5))
plt.plot(test_dates,y_test_inv_best, label='Actual', color='gray', linewidth=2)
plt.plot(test_dates,y_pred_inv_best, label='Prediction', color='purple', linestyle=
plt.title(f'GRU Model (Window Size={best_w}, Unit={best_units}): Actual vs. Predict
plt.xlabel('Days')
plt.ylabel('Price (USD)')
plt.legend()
plt.grid(True)
plt.show()
```



9.4 Performance Score

Selected Features: ['Close', 'EMA_20', 'Return_1D', 'RSI_14', 'Volume_log', 'Volatility']

Root Mean Squared Error (RMSE): 7.2463

Mean Absolute Error (MAE): 5.0449

Mean Absolute Percentage Error (MAPE): 0.8664%

Directional Accuracy (DAC): 48.33%

Key Finding:

- The MAE score shows that the model's predictions can be off by about $5 on average and with errors under 1% of actual prices.
- A low DAC of 48% implies a low probability of accurately predicting the next day's uptrend or downtrend — slightly worse than flipping a coin.
- Based on the DAC score alone, the model is not dependable for reliable trading signals because it consistently predicts the wrong direction.

Note:

- DAC (aka Mean Directional Accuracy (MDA)) calculates the percentage of times the predicted and actual values move in the same up-or-down direction (Mohamad, 2019).

$$\text{MDA} = \frac{1}{N} \sum_{t=2}^{T} \mathbf{1}\big(\text{sign}(y_t - y_{t-1}) = \text{sign}(\hat{y}_t - y_{t-1})\big)$$

In [46]:
```
last_train_price = y_train_inv_best[-1]
# true array
```

```
y_true_with_start = np.insert(y_test_inv_best, 0, last_train_price)

# prediction array
y_pred_with_start = np.insert(y_pred_inv_best, 0, last_train_price)

# calculating the sign of the difference between consecutive value
def cal_dir(series):
        return np.sign(series[1:] - series[:-1])


# compare the predicted direction with the actual direction
def dac(y_true, y_pred):

        true_dir = cal_dir(y_true)
        pred_dir = cal_dir(y_pred)

        # Compare the signs
        correct_pred = (true_dir == pred_dir)

        # Calculate accuracy
        return np.mean(correct_pred.astype(int)) * 100


final_rmse_score = rmse(y_test_inv_best, y_pred_inv_best)
final_mae_score = mae(y_test_inv_best, y_pred_inv_best)
final_mape_score = mape(y_test_inv_best, y_pred_inv_best)
da_score = dac(y_true_with_start, y_pred_with_start)

print(f"Selected Features: {feature_cols}")
print(f"Root Mean Squared Error: {final_rmse_score:.4f}")
print(f"Mean Absolute Error: {final_mae_score:.4f}")
print(f"Mean Absolute Percentage Error: {final_mape_score:.4f}%")
print(f"Directional Accuracy: {da_score:.2f}%")
```

```
Selected Features: ['Close', 'EMA_20', 'Return_1D', 'RSI_14', 'Volume_log', 'Volatil
ity']
Root Mean Squared Error: 7.2463
Mean Absolute Error: 5.0449
Mean Absolute Percentage Error: 0.8664%
Directional Accuracy: 48.33%
```

9.5 Forecasting the Next 14 Business Days

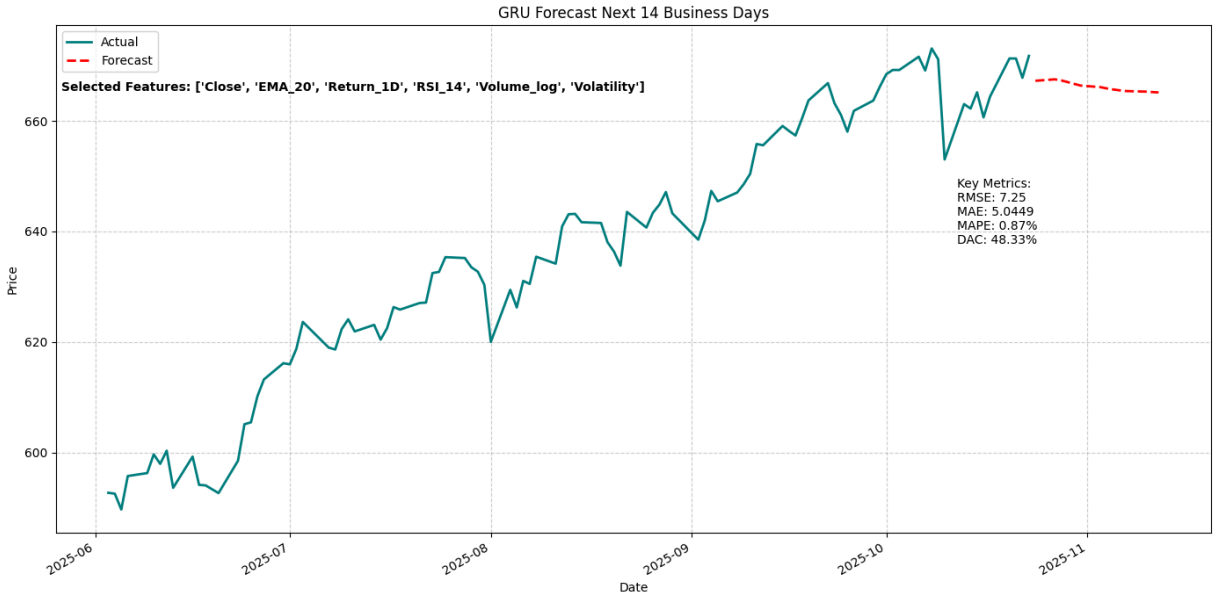
This section evaluates the model's capability to generate a 14-business-day price forecast.

- An autoregressive method produces the forecast (Brownlee, 2021; Hyndman & Athanasopoulos, 2021), allowing the model to predict the price for the next day (Day N+1) without new external input. The predicted price for Day N+1 is then fed back into the model as part of the input for predicting Day N+2, and this process continues for 14 steps.
- The plotted forecasting chart shows a dip followed by a flat, non-volatile line.
- As `Return_1D`, `RSI_14`, `Volume`, and `Volatility` have no future variation beyond the training data, this contributes to the lack of volatility.

In [47]:

```
############################################################
# Forecasting 14 days price
############################################################
forecast_days = 14

# get the last windows of data from the test set
last_windows = X_test_scaled[-best_w:]

# reshapes 2D last_windows into 3D for GRU model
current_windows = last_windows.reshape(1, best_w, len(feature_cols)).copy()

scaled_final_pred = []

# autoregressive Loop repeats for each day
# automatically feed each day prediction back in the input for next day prediction
for _ in range(forecast_days):

    # predict the next day scaled value
    scaled_next_pred = model_best.predict(current_windows, verbose=0)[0, 0]

    # saved the next day prediction
    scaled_final_pred.append(scaled_next_pred)

    # copy all last day features data into temporary input
    temp_input = current_windows[0, -1, :].copy()

    CLOSE_FEATURE_INDEX = 0

    # !!! IMPORTANT !!!!
    # replace the last 'Close' price with new predicted price
    temp_input[CLOSE_FEATURE_INDEX] = scaled_next_pred

    # !!! VERY IMPORTANT !!!!
    # convert the 1D temp_input into 3D with 1 day, 1 time step, N features
    temp_3d_ds = temp_input.reshape(1, 1, n_features)

    # !!! SUPER IMPORTANT !!!!
    # slide the current windows data by 1 day
    # by selected only the index 1 to n set of data
    # dropping index 0 (oldest data point)
    # then update the current window with
    # the new predicted price to the newest data point (n+1)
    # while maintaining the window size
    current_windows = np.concatenate([current_windows[:, 1:, :], temp_3d_ds], axis=

#convert scale data to actual value
non_scaled_prediction = scaler_y.inverse_transform(np.array(scaled_final_pred).resh


last_date = pd.to_datetime(spy_data.index[-1])

#calculate the future date based on the forecast days
future_dates = pd.date_range(start=last_date + pd.Timedelta(days=1),
                             periods=forecast_days, freq='B')
```

```python
# plot the forecasting price
plt.figure(figsize=(14, 7))
recent_days_to_plot = 100

plt.plot(spy_data.index[-recent_days_to_plot:],
         spy_data['Close'][-recent_days_to_plot:],
         label='Actual', color='teal', linewidth=2)

plt.plot(future_dates,
         non_scaled_prediction,
         label=f'Forecast',
         color='red', linestyle='--', linewidth=2)


# add Key Metrics comment
plt.text(0.005, 0.89,
         f'Selected Features: {feature_cols}',
         transform=plt.gca().transAxes,
         fontsize=10,
         fontweight='bold',
         verticalalignment='top')
plt.text(0.78, 0.70,
         f'Key Metrics:\nRMSE: {final_rmse_score:.2f}\nMAE: {final_mae_score:.4f}\n
         transform=plt.gca().transAxes,
         fontsize=10,
         verticalalignment='top',
         horizontalalignment='left')


plt.title(f'GRU Forecast Next {forecast_days} Business Days')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend(loc='upper left')
plt.grid(True, linestyle='--', alpha=0.6)
plt.gcf().autofmt_xdate()
plt.tight_layout()
plt.show()
```



9.6 Feature Comparison in Forecasting and Performance Scores

The feature comparison table provides insight into the project:

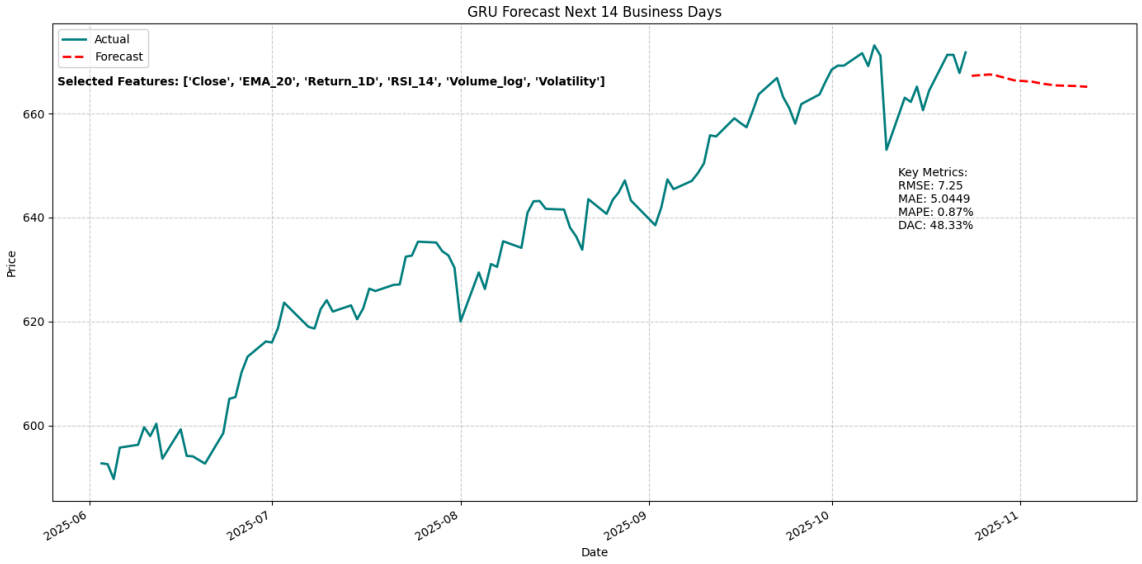
- The results showed that the `Return_1D`, `RSI_14`, `Volume`, and `Volatility` features were unhelpful. The model could not incorporate the data from these features due to the noise that contradicts the price action. The model performs better when focusing on price action.
- The log-transformed `Close` and `EMA_20` produced a DAC score similar to the non-log-transformed `Close` and `EMA_20`. This is a surprising finding, as the distributions of these features were not skewed but multimodal, as shown in the EDA.
- Log-transformed features' RMSE, MAE, and MAPE scores are for reference and should not be compared with non-log-transformed features, as they require exponential inverse-transform first before comparison.
- Visually, Feature Groups 1, 2, 3, and 6 appear more realistic, without the dramatic sharp dips or spikes.

Feature Group	Selected Features	RMSE	MAE	MAPE	Directional Accuracy
1	['Close', 'EMA_20', 'Return_1D', 'RSI_14', 'Volume_log', 'Volatility']	7.2463	5.0449	0.8664%	48.33%
2	['Close']	7.0570	4.7534	0.8173%	53.73%
3	['Close', 'EMA_20']	7.6136	5.6016	0.9596%	53.36%

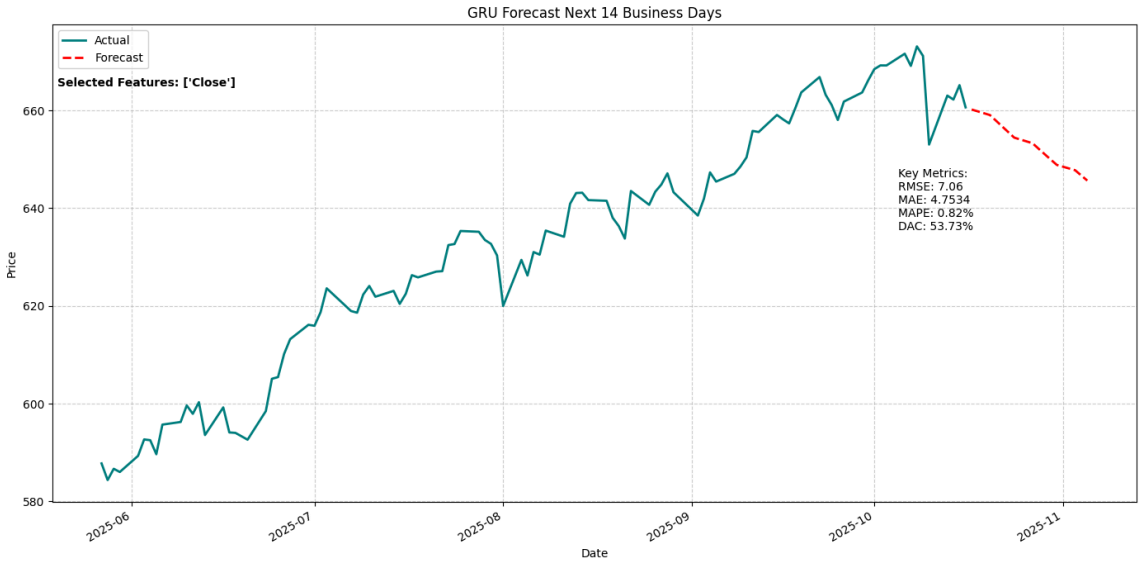
Feature Group	Selected Features	RMSE	MAE	MAPE	Directional Accuracy
4	['Close_log', 'EMA_20_log', 'Return_1D', 'RSI_14', 'Volume_log', 'Volatility']	0.0155	0.0123	0.1930%	47.01%
5	['Close_log', 'EMA_20_log']	0.0178	0.0139	0.2172%	**54.77%**
6	['Close_log']	**0.0133**	**0.0083**	**0.1310%**	53.71%

Best Directional Accuracy: Feature Group 5

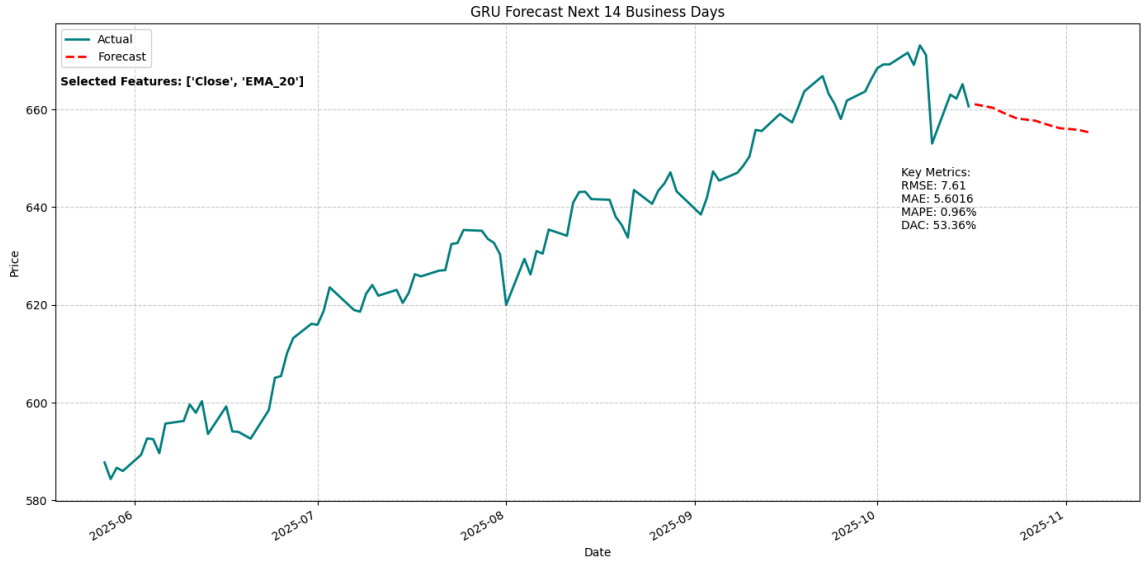
Feature Group 1



Feature Group 2

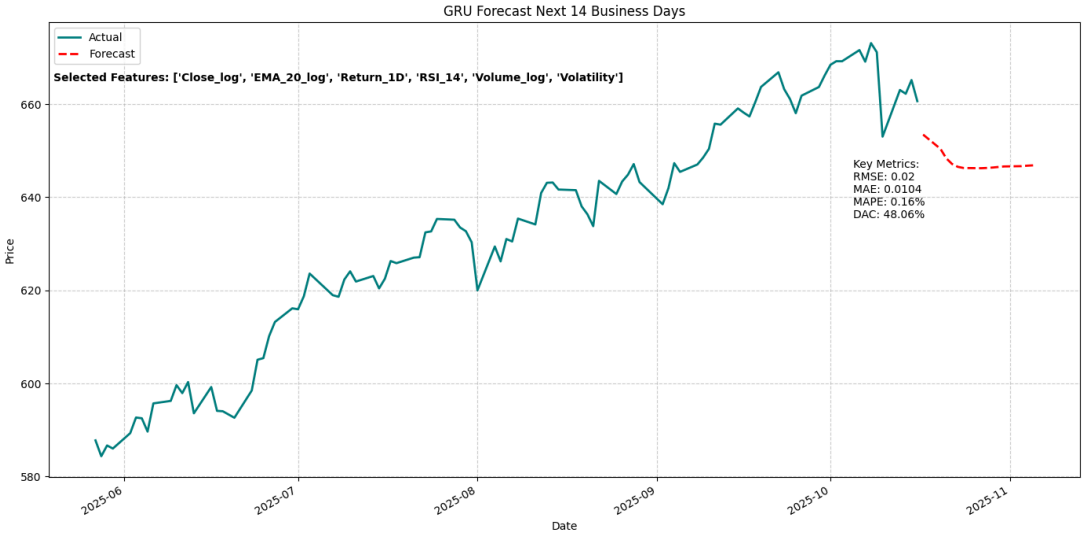


Feature Group 3



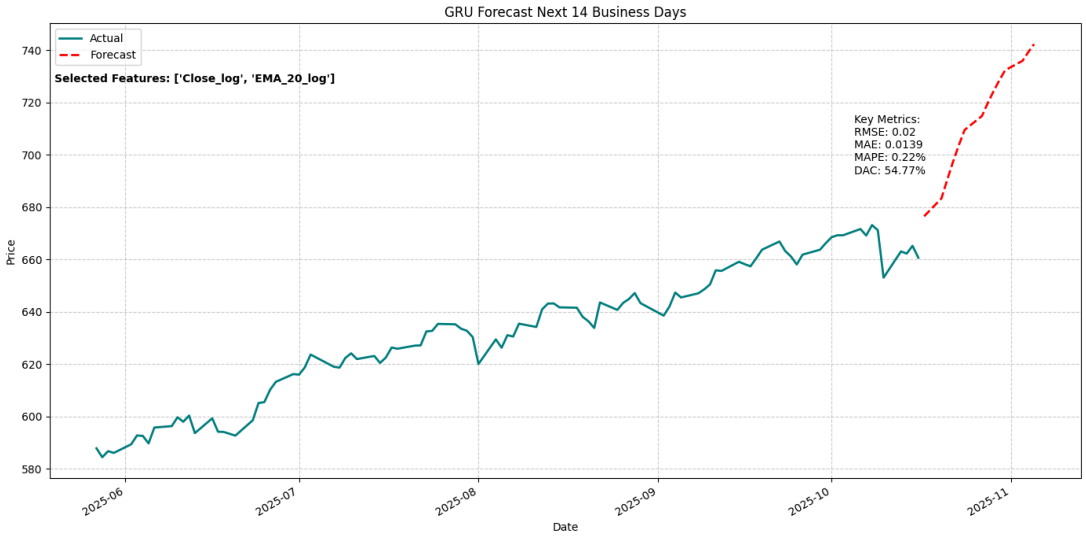
Feature Group 4

- Feature Groups 4 and 5 show price forecasts that are unrealistic and unlikely to occur, given the current price action trend and external technical analysis.
- Feature Group 4 shows a sharp drop followed by a flat, stable line. The model appears to overcorrect and fails to capture realistic short-term movements.

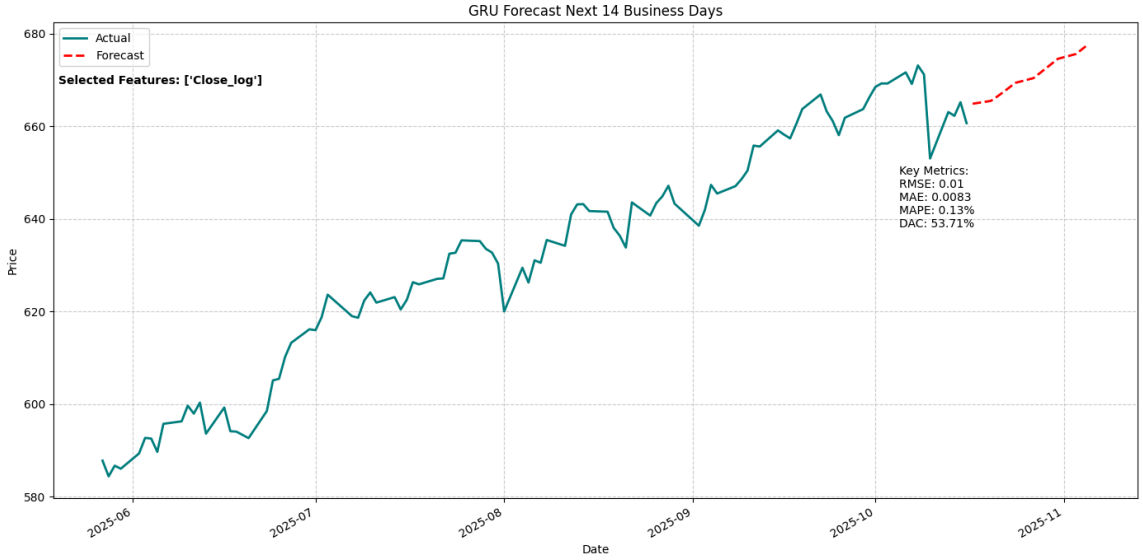


Feature Group 5

- Despite having the best DAC score, Feature Group 5 shows a sudden price spike and steep climb that do not align with the stock's typical trend



Feature Group 6



10. Discussion and Conclusion

10.1 Key Takeaways

- Dynamics of Price Prediction:

  Financial markets are affected by many factors that cannot be fully measured through price alone, such as politics, wars, natural disasters, and human emotions like greed and fear. To improve stock market predictions, future research should consider using multi-step forecasting models and integrating insights from human technical analysis.

- Directional Accuracy (DAC):

  The primary metric for financial forecasting should be DAC, as it provides a more meaningful measure than traditional error metrics like RMSE or MAE. Therefore, the usability of a model for daily trading is defined not by the lowest error score but by its

DAC. If a model has a probability of being correct more than 50% of the time, the odds of profit increase significantly over the long term (Yin & Zhao, 2023) — even by as little as 0.1%.

- Data Leakage:

  One concern in this project is data leakage, which can distort prediction results. Strict separation between training and testing datasets is essential in time-series forecasting to avoid misleading outcomes. This issue arises when data is split sequentially after creating sliding windows — if the windows are generated first, overlap occurs, leading to data leakage and unrealistically inflated performance.

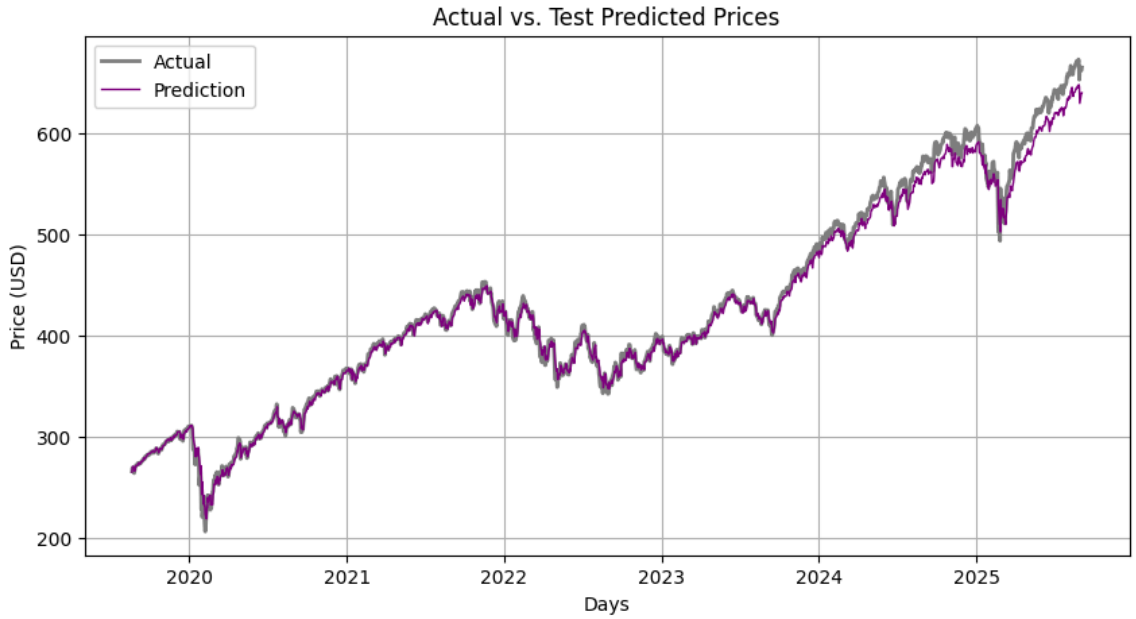
10.2 Why Something Didn't Work

- Feature Selection:

  `Return_1D`, `RSI_14`, `Volume`, and `Volatility` are technical indicators valuable for human technical analysis because they provide useful trading signals. However, models trained with the full set of six technical features showed lower DAC and higher MAE compared to models using only `Close` and `EMA_20`. The GRU model was unable to effectively integrate `Return_1D`, `RSI_14`, `Volume`, and `Volatility` into its learning process because GRUs are designed to learn sequential trends, while these features do not follow the same temporal patterns as price data, making them harder to relate directly to future prices. Unless the model can learn these non-price features separately and combine their predictions in the final output, their inclusion may only add noise. If the GRU primarily predicts prices based on historical price patterns alone, this could lead to weaker performance on unseen data.

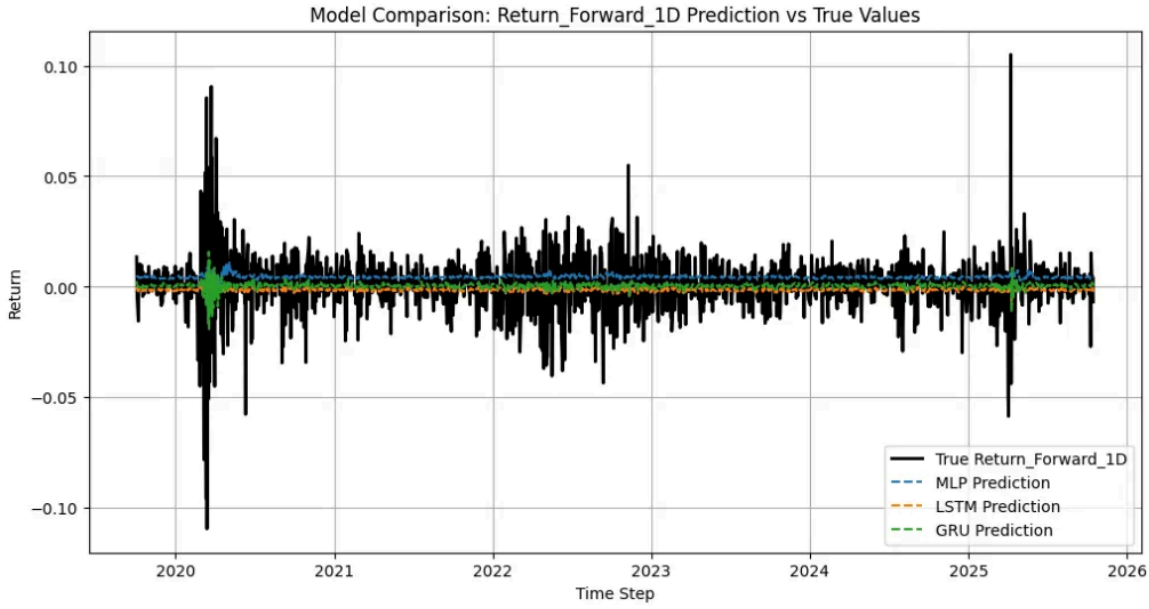
- Irrelevant Data:

  Testing with 30 years of historical data did not improve model performance — in fact, it led to poorer results. As shown in the chart below, the forecast deviates more significantly as the prediction period extends further from the training window. Market patterns from 20 years ago (e.g., the 2000 tech bubble and 2008 financial crisis) are very different from today's market. Therefore, for short-term forecasting, using more recent datasets (e.g., the past 5–7 years) is more effective.



- Future Single-Day Returns:

  When testing the model to predict the next day's single-day returns using the current feature set, the result was essentially zero, indicating no unprofitable returns, and showed no meaningful variation regardless of the input features. It is likely that the models learned that the safest prediction was the mean value, approximately 0. This suggests the models were unable to capture the true variance of returns in such a noisy and non-linear time series.



10.3 Future Improvements

- Hybrid Models:

  Based on observations from this project, the GRU model struggled to effectively integrate additional features and had difficulty with next-day predictions, resulting in a

noticeable “lag” in its outputs. By leveraging the `Volatility` feature, a GRU-Attention hybrid model could assign greater weight to the most recent data (Zhu, 2025). This approach could also be extended to other features, such as `Return_1D`, `RSI_14`, and `Volume`.

- Deep Reinforcement Learning:

  A potential future improvement involves training an AI agent to learn actions to buy or sell, based on a defined policy that works in conjunction with the price prediction model (Liu, 2022).

10.4 Conclusion

This project compared three deep learning models for stock price prediction and found the GRU model to be the most effective. Traditional indicators like RSI added little value compared to price-focused data. While this project demonstrated the potential of deep learning for price prediction, short-term markets remain unpredictable, highlighting the need for more context-aware and adaptive models.

References

Brownlee, J. (2016, August 4). Time series prediction with LSTM recurrent neural networks in Python with Keras. Machine Learning Mastery. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

Brownlee, J. (2020, July 15). Classification versus regression in machine learning. Machine Learning Mastery. https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/

Brownlee, J. (2021, September 7). Autoregression Models for Time Series Forecasting With Python. Machine Learning Mastery. https://machinelearningmastery.com/autoregression-models-time-series-forecasting-python/

Buturac, G. (2022). Measurement of economic forecast accuracy: A systematic overview of the empirical literature. Journal of Risk and Financial Management, 15 (1), 1. https://doi.org/10.3390/jrfm15010001

GRU layer. (n.d.). Keras. Retrieved October 28, 2025, from https://keras.io/api/layers/recurrent_layers/gru/

Hyndman, R. J., & Athanasopoulos, G. (2021). Forecasting: Principles and Practice (3rd ed.). OTexts. https://otexts.com/fpp3/

Image classification using an MLP. (n.d.). Keras. Retrieved October 28, 2025, from https://keras.io/examples/vision/mlp_image_classification/

Leitch, G., & Tanner, J. E. (1991). Economic forecast evaluation: Profits versus the conventional error measures. The American Economic Review, 81(3), 580-590.

Liu, X. Y., Yang, H., Chen, Q., Yang, R., Zhang, R., Yang, L., Xiao, B., & Wang, C. D. (2022). FinRL: A deep reinforcement learning library for automated trading in quantitative finance. Applied Soft Computing, 118, Article 108486.

LSTM layer. (n.d.). Keras. Retrieved October 28, 2025, from https://keras.io/api/layers/recurrent_layers/lstm/

Mohamad. (2019, June 14). MDA – Mean directional accuracy. NumXL Help Center. Retrieved from https://support.numxl.com/hc/en-us/articles/360029220972-MDA-Mean-Directional-Accuracy

Ohliati, J., & Yuniarty. (2024). Deep Learning for Stock Market Prediction: A Review. In 2024 International Conference on Information Management and Technology (ICIMTech). IEEE. https://ieeexplore.ieee.org/document/10780931

Yin, T., & Zhao, C. (2023). Enhancing directional accuracy in stock closing price value prediction using a direction-integrated MSE loss function. Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023), 2, 77-88.

Yulistiani, R., & Kurniadi, F. I. (2024). Stock Price Prediction With the Informer Model. In 2024 International Conference on Information Management and Technology (ICIMTech). IEEE. https://ieeexplore.ieee.org/document/10780913

Zhu, P., Li, Y., Hu, Y., Xiang, S., Liu, Q., Cheng, D., & Liang, Y. (2025). MCI-GRU: Stock prediction model based on multi-head cross-attention and improved GRU. arXiv preprint arXiv:2410.20679.

Wikipedia contributors. (n.d.). *Long short-term memory*. Wikipedia. https://en.wikipedia.org/wiki/Long_short-term_memory

Data Source

Yahoo Finance. (n.d.). *SPDR S&P 500 ETF Trust (SPY) stock historical prices & data*. https://finance.yahoo.com/quote/SPY/history

GitHub Repository Link

https://github.com/peculiardatabits/DTSA-5511-Deep-Learning-Final-Project.git