# Stock Investment Recommendation Model

## Business Overview for Stock Investment Recommendation Model

---

**Project Title:** Stock Investment Recommendation System

**Overview:**
This project aims to create a machine learning model that delivers personalized stock investment recommendations based on individual risk appetites. By analyzing historical data from Yahoo Finance, the system will suggest suitable investments across various sectors, empowering investors with data-driven insights to enhance their decision-making and improve investment outcomes.

# Business Understanding

The business questions we will be trying to answer are:

1. How can we categorize stocks into different risk levels that align with investor preferences?
2. How can we personalize stock recommendations based on each customer's risk appetite?
3. What data-driven insights can we provide to customers to increase their confidence in the recommended stocks?
4. What metrics should be used to evaluate the success of the recommendations (e.g., customer satisfaction, portfolio performance, customer retention)?

# Data Understanding

## Overview

The dataset obtained from Yahoo Finance contains historical stock price data for various companies. This data typically includes daily trading information, which is crucial for analyzing stock performance, volatility, and market trends. The dataset spans multiple years, allowing for comprehensive analysis of stock behavior over time.

### Structure of the Dataset

The dataset contains the following columns:

1. **Open**: The price at which the stock opened for trading on a given day.
2. **High**: The highest price reached during the trading session.
3. **Low**: The lowest price recorded during the trading session.
4. **Close**: The price at which the stock closed at the end of the trading day.
5. **Adj Close**: The adjusted closing price that accounts for any corporate actions (like stock splits and dividends) to reflect the stock's true value.
6. **Volume**: The total number of shares traded during the day, providing insights into market activity and liquidity.
7. **Beta**: A measure of a stock's volatility in relation to the market, indicating how much the stock's price is expected to move compared to a broader index (like the S&P 500).

# Data Cleaning

This section reads in a CSV file containing stock data and displays the first 20 rows and summary statistics. This provides an initial overview of the data's structure and basic statistics for each column.

In [1]:

```
# importing  neccesarry libraries
import pandas as pd
import numpy as np
from scipy.stats import zscore
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import pandas as pd
```

## Data Loading

This section of the code is responsible for loading the stock data from a CSV file and providing a quick overview of the dataset.

In [2]:

```
# Load the dataset
try:
    data = pd.read_csv("final_stocks_data.csv")
    print("Data loaded successfully.")

    # Display the first 20 rows of the dataset
    print(data.head(5))

    # Display basic information about the dataset
    print(data.info())
    print(data.describe())
except FileNotFoundError:
    print("Error: The file 'final_stocks_data.csv' was not found.")
except Exception as e:
    print(f"An error occurred: {e}")
```

```
Data loaded successfully.
```

```
Data loaded successfully.
                        Date Symbol        Open        High         Low  \
0  2022-01-03 00:00:00+00:00   AAPL  177.830002  182.880005  177.710007
1  2022-01-03 00:00:00+00:00    XOM   61.240002   63.599998   61.209999
2  2022-01-03 00:00:00+00:00     VZ   52.070000   52.560001   51.980000
3  2022-01-03 00:00:00+00:00      V  217.520004  222.059998  217.009995
4  2022-01-03 00:00:00+00:00   TSLA  382.583344  400.356659  378.679993

        Close   Adj Close       Volume      Beta
0  182.009995  179.273621  104487900.0  1.232782
1   63.540001   57.618000   24282400.0  0.471497
2   52.439999   43.423004   18240100.0  0.337004
3  221.429993  216.793564    7694500.0  0.867907
4  399.926666  399.926666  103931400.0  1.909699
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21450 entries, 0 to 21449
Data columns (total 9 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       21450 non-null  object
 1   Symbol     21450 non-null  object
 2   Open       21450 non-null  float64
 3   High       21450 non-null  float64
 4   Low        21450 non-null  float64
 5   Close      21450 non-null  float64
 6   Adj Close  21450 non-null  float64
 7   Volume     21450 non-null  float64
 8   Beta       21450 non-null  float64
dtypes: float64(7), object(2)
memory usage: 1.5+ MB
None
               Open          High           Low         Close     Adj Close  \
count  21450.000000  21450.000000  21450.000000  21450.000000  21450.000000
mean     165.502818    167.380023    163.595675    165.528565    162.928384
std      144.517485    146.171365    142.767715    144.503328    144.790031
min       10.971000     11.735000     10.813000     11.227000     11.216744
25%       62.304999     62.775001     61.757500     62.312501     59.463318
50%      128.504997    130.000000    127.004749    128.449997    123.171261
75%      198.694996    200.940002    196.529995    199.029999    197.360813
max      957.770020    972.530029    951.580017    960.020020    960.020020

             Volume          Beta
count  2.145000e+04  21450.000000
mean   3.650411e+07      0.936610
std    9.101992e+07      0.509190
min    9.427000e+05      0.253678
25%    5.438000e+06      0.471497
50%    1.176420e+07      0.823051
75%    2.836222e+07      1.336116
max    1.543911e+09      2.300627
```

Lets define a function to remove outliers based on Z-scores. This technique identifies outliers by calculating the Z-score for each value in specified columns. If a Z-score is above the threshold(3), the value is considered an outlier and removed.

In [3]:

```python
# removing for outliers
def remove_outliers_zscore(data, numerical_cols, threshold=3):
    """
    Removes outliers from the given DataFrame using the Z-score method.
```

```
    Parameters:
    -----------
    data : pd.DataFrame
        The input DataFrame containing the data.
    numerical_cols : list
        List of numerical columns to check for outliers.
    threshold : float, optional (default=3)
        The Z-score threshold to use for outlier detection.

    Returns:
    --------
    pd.DataFrame
        A new DataFrame with outliers removed.
    """
    # Calculate Z-scores for the numerical columns
    z_scores = data[numerical_cols].apply(zscore)

    # Create a mask for non-outliers (Z-scores within the threshold)
    mask = (np.abs(z_scores) < threshold).all(axis=1)

    # Return the filtered DataFrame without outliers
    return data[mask]

# calling the function
numerical_cols = ['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']
data_cleaned = remove_outliers_zscore(data, numerical_cols)

print(f"Original data shape: {data.shape}")
print(f"Cleaned data shape: {data_cleaned.shape}")

# Optional: Save the cleaned data to a CSV
```

```
Original data shape: (21450, 9)
Cleaned data shape: (20371, 9)
```

In [4]:

```
data_cleaned.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 20371 entries, 0 to 21449
Data columns (total 9 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Date       20371 non-null  object 
 1   Symbol     20371 non-null  object 
 2   Open       20371 non-null  float64
 3   High       20371 non-null  float64
 4   Low        20371 non-null  float64
 5   Close      20371 non-null  float64
 6   Adj Close  20371 non-null  float64
 7   Volume     20371 non-null  float64
 8   Beta       20371 non-null  float64
dtypes: float64(7), object(2)
memory usage: 1.6+ MB
```

In [5]:

```
data = data_cleaned
```

next we converted the "Date" column to datetime format for consistency, renames columns by removing whitespace and standardizing to lowercase with underscores, and calculates the number of duplicate rows in the dataset.

A column for the actual company names is created because the user may not be familiar with just the symbol. A dictionary is created to map the symbol with the corresponding name of the company.

In [6]:
```
# create dictionary to map the symbols with the company name
company_names = {
    "AAPL": "Apple Inc.",
    "XOM": "Exxon Mobil Corporation",
    "JPM": "JPMorgan Chase & Co.",
    "PG": "Procter & Gamble Co.",
    "JNJ": "Johnson & Johnson",
    "NEE": "NextEra Energy, Inc.",
    "MSFT": "Microsoft Corporation",
    "AMZN": "Amazon.com, Inc.",
    "TSLA": "Tesla Inc.",
    "GOOGL": "Alphabet Inc. (Google)",
    "NVDA": "NVIDIA Corporation",
    "V": "Visa Inc.",
    "MA": "Mastercard Incorporated",
    "DIS": "The Walt Disney Company",
    "NFLX": "Netflix, Inc.",
    "INTC": "Intel Corporation",
    "CSCO": "Cisco Systems, Inc.",
    "PFE": "Pfizer Inc.",
    "KO": "The Coca-Cola Company",
    "PEP": "PepsiCo, Inc.",
    "T": "AT&T Inc.",
    "VZ": "Verizon Communications Inc.",
    "CMCSA": "Comcast Corporation",
    "ADBE": "Adobe Inc.",
    "IBM": "International Business Machines Corporation",
    "NKE": "Nike, Inc.",
    "CRM": "Salesforce, Inc.",
    "LLY": "Eli Lilly and Company",
    "ABT": "Abbott Laboratories",
    "MDT": "Medtronic plc"
}
```

In [7]:
```
# create new column based on the dictionary
data['company_name'] = data['Symbol'].map(company_names)
```

In [8]:
```
# changing columns to correct data types
data["Date"] = pd.to_datetime(data["Date"])
```

In [9]:
```
# removing space and switching to lowercase
data.columns = data.columns.str.strip().str.lower().str.replace(' ', '_')
```

In [10]:
```
# checking for duplicates
duplicates = data.duplicated().sum()
duplicates
```

```
duplicates
```

Out[10]: 0

In [11]:

```
# transfer to another csv file for Tableau for further Visualization
data.to_csv("data_cleaned.csv", index=False)
```

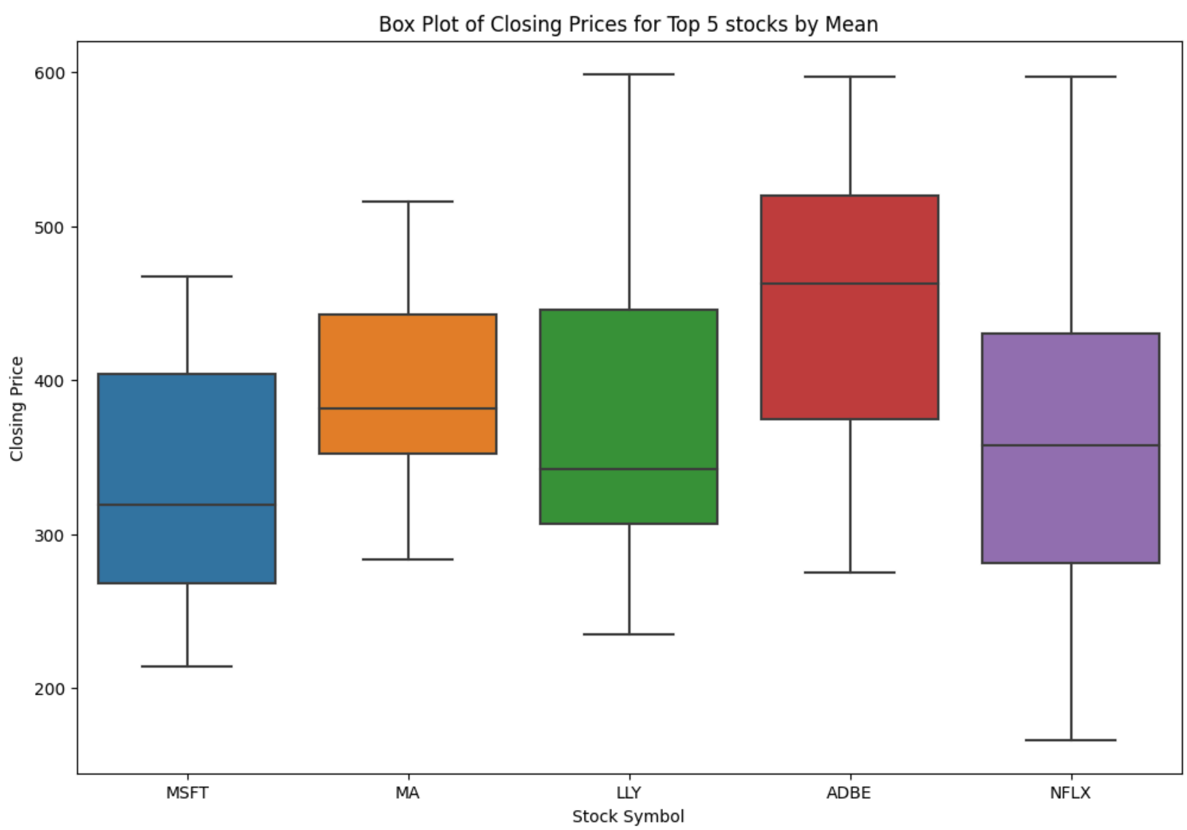
# EDA

In [12]:

```
top_symbols = data.groupby('symbol')['close'].mean().nlargest(5).index
data['symbol'].value_counts().head(5).index

# Filter the data to only include the top 5 symbols
top_symbols_data = data[data['symbol'].isin(top_symbols)]

# Create box plots for 'Close' prices of the top 5 symbols to compare distributi
plt.figure(figsize=(12, 8))
sns.boxplot(x='symbol', y='close', data=top_symbols_data)
plt.title("Box Plot of Closing Prices for Top 5 stocks by Mean")
plt.xlabel("Stock Symbol")
plt.ylabel("Closing Price")
plt.show()
```




In [13]:

```
data.info()
```

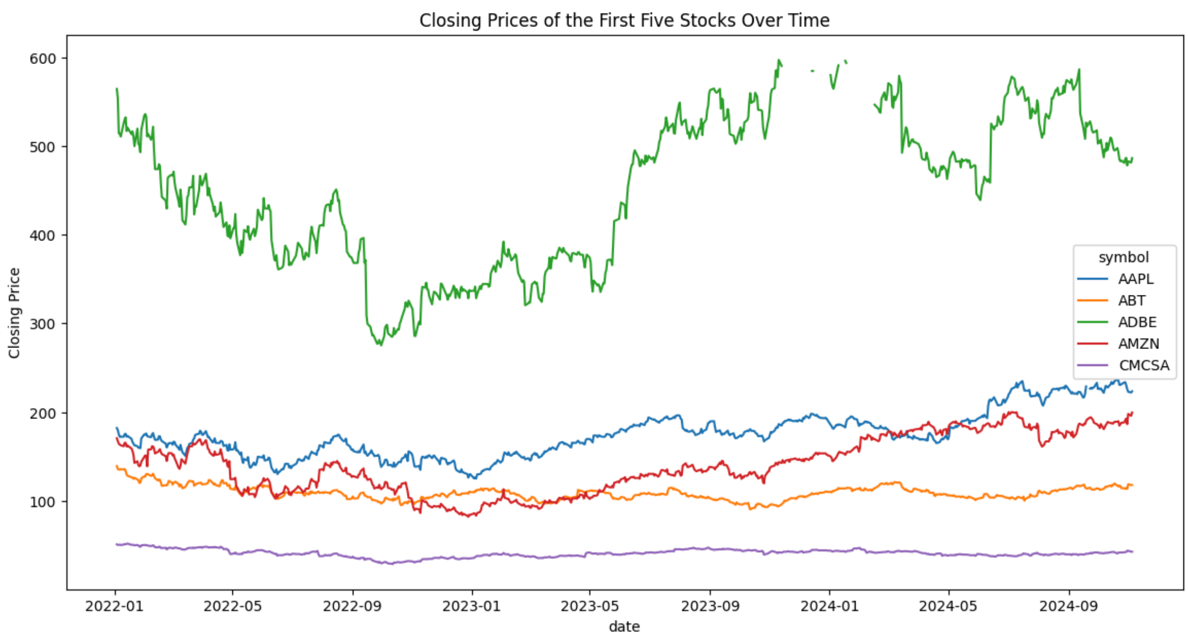
```
<class 'pandas.core.frame.DataFrame'>
Index: 20371 entries, 0 to 21449
Data columns (total 10 columns):
```

```
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   date          20371 non-null  datetime64[ns, UTC]
 1   symbol        20371 non-null  object
 2   open          20371 non-null  float64
 3   high          20371 non-null  float64
 4   low           20371 non-null  float64
 5   close         20371 non-null  float64
 6   adj_close     20371 non-null  float64
 7   volume        20371 non-null  float64
 8   beta          20371 non-null  float64
 9   company_name  20371 non-null  object
dtypes: datetime64[ns, UTC](1), float64(7), object(2)
memory usage: 1.7+ MB
```

In [14]:

```python
#for the first ten stocks
closing_prices = data.pivot(index='date', columns='symbol', values='close')

# Plot the data
plt.figure(figsize=(14, 7))
for stock in closing_prices.columns[:5]:
    plt.plot(closing_prices.index, closing_prices[stock], label=stock)
plt.title('Closing Prices of the First Five Stocks Over Time')
plt.xlabel('date')
plt.ylabel('Closing Price')
plt.legend(title='symbol')
plt.show()
```

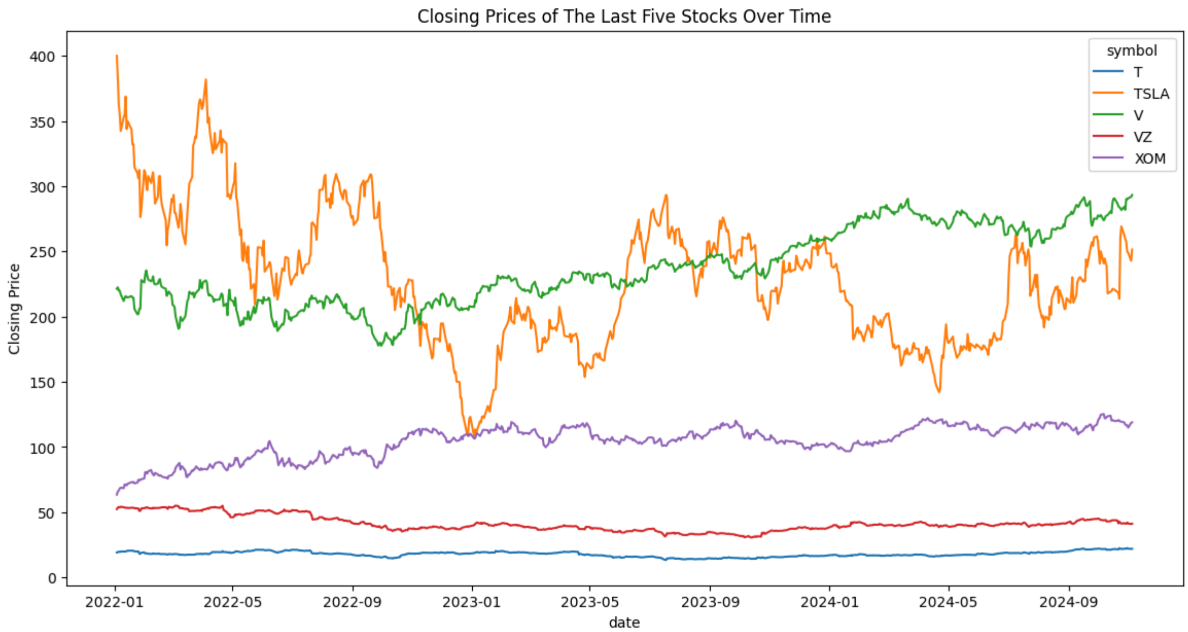



In [15]:

```python
# for the last 5
closing_prices = data.pivot(index='date', columns='symbol', values='close')

# Plot the data
plt.figure(figsize=(14, 7))
for stock in closing_prices.columns[25:30]:
    plt.plot(closing_prices.index, closing_prices[stock], label=stock)
plt.title('Closing Prices of The Last Five Stocks Over Time')
plt.xlabel('date')
plt.ylabel('Closing Price')
```

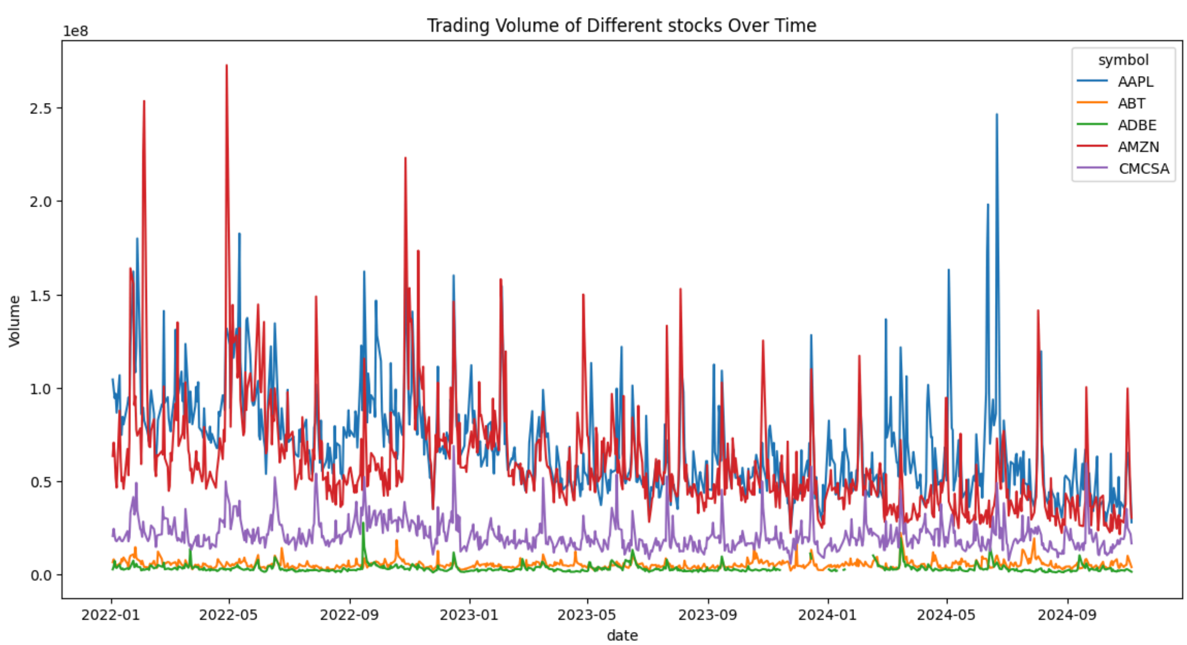
```
plt.legend(title='symbol')
plt.show()
```




In [16]:

```
#stock volume over time
volumes = data.pivot(index='date', columns='symbol', values='volume')

# Plot
plt.figure(figsize=(14, 7))
for symbol in volumes.columns[:5]:
    plt.plot(volumes.index, volumes[symbol], label=symbol)
plt.title('Trading Volume of Different stocks Over Time')
plt.xlabel('date')
plt.ylabel('Volume')
plt.legend(title='symbol')
plt.show()
```

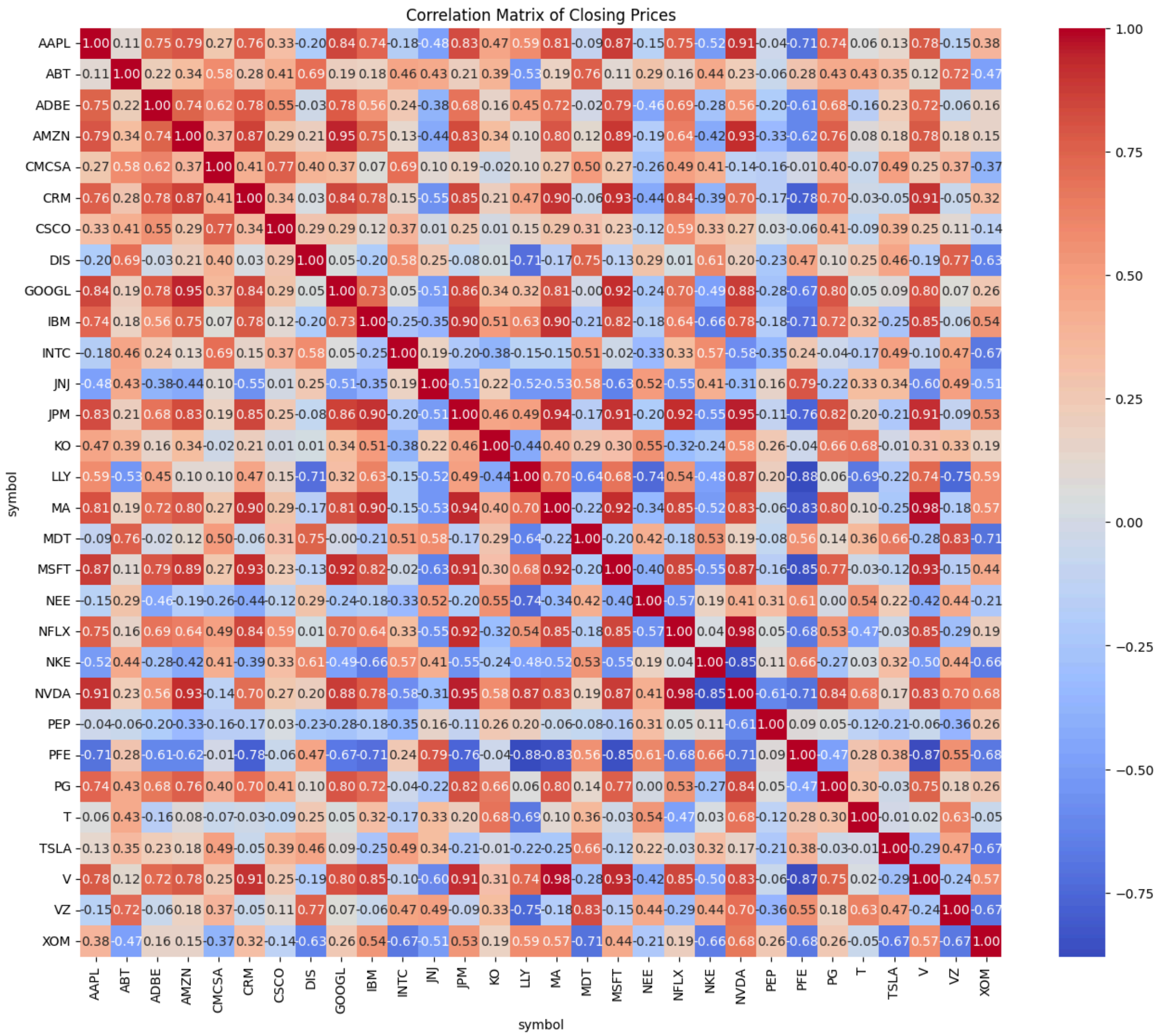



In [17]:

```
# Calculate the correlation matrix
```

```python
corr_matrix = closing_prices.corr()

# Plot heatmap
plt.figure(figsize=(16, 13))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix of Closing Prices')
plt.show()
```




# FEATURE ENGINEERING

## Explanation of Key Features

1. **Daily Returns**: Captures the daily percentage change in closing price, a metric of stock volatility.
2. **Moving Averages (SMA and EMA)**: Smooths price data to highlight trends.
3. **Volatility**: Measures price fluctuations over short periods (7 and 14 days).
4. **Relative Strength Index (RSI)**: Indicates overbought or oversold conditions based on price gains and losses.
5. **Bollinger Bands**: Provides a volatility range using the 20-day SMA and 2 standard deviations.
6. **Time-Based Features**: Adds temporal information (day of the week, month, and

6. **Time-Based Features**: Adds temporal information (day of the week, month, and quarter) for seasonality.
7. **Lag Features**: Captures previous days' prices to model short-term momentum.
8. **MACD**: Measures the difference between 12-day and 26-day EMAs to indicate momentum shifts.
9. **Backfilling Missing Values**: Ensures the dataset is complete for modeling.

This feature engineering process enriches the dataset with essential indicators, enabling effective clustering based on risk.

In [18]:

```
#  Ensure Date is in datetime format
def preprocess_data(data):
    data['date'] = pd.to_datetime(data['date'])
    data.sort_values(by=['symbol', 'date'], inplace=True)
    return data

# Calculate Daily Returns
def calculate_daily_returns(data):
    data['daily_return'] = data.groupby('symbol')['close'].transform(lambda x: x
    return data

# Calculate Simple Moving Averages (SMA)
def calculate_sma(data, windows=[7, 14, 20]):
    for window in windows:
        data[f'SMA_{window}'] = data.groupby('symbol')['close'].transform(lambda
    return data

# Calculate Exponential Moving Averages (EMA)
def calculate_ema(series, span):
    return series.ewm(span=span, adjust=False).mean()

def add_ema(data, spans=[7, 14, 12, 26]):
    for span in spans:
        data[f'EMA_{span}'] = data.groupby('symbol')['close'].transform(lambda x
    return data

# Calculate Volatility
def calculate_volatility(data, windows=[7, 14]):
    for window in windows:
        data[f'Volatility_{window}d'] = data.groupby('symbol')['daily_return'].t
    return data

# Calculate Relative Strength Index (RSI)
def calculate_rsi(series, window=14):
    delta = series.diff()
    gain = np.where(delta > 0, delta, 0)
    loss = np.where(delta < 0, -delta, 0)

    avg_gain = pd.Series(gain).rolling(window=window).mean()
    avg_loss = pd.Series(loss).rolling(window=window).mean()

    rs = avg_gain / avg_loss
    return 100 - (100 / (1 + rs))

def add_rsi(data, window=14):
    data[f'RSI_{window}'] = data.groupby('symbol')['close'].transform(lambda x:
    return data
```

```
# Calculate Bollinger Bands
def calculate_bollinger_bands(data, window=20):
    data['SMA_20'] = data.groupby('symbol')['close'].transform(lambda x: x.rolli
    rolling_std = data.groupby('symbol')['close'].transform(lambda x: x.rolling(
    data['BB_Upper'] = data['SMA_20'] + (2 * rolling_std)
    data['BB_Lower'] = data['SMA_20'] - (2 * rolling_std)
    return data

# Add Time-Based Features
def add_time_features(data):
    data['Day_of_Week'] = data['date'].dt.dayofweek  # 0 = Monday, 6 = Sunday
    data['Month'] = data['date'].dt.month
    data['Quarter'] = data['date'].dt.quarter
    return data

# Create Lag Features for Open, High, Low, and Close prices
def create_lag_features(data, lags=[1, 2, 3]):
    for lag in lags:
        for col in ['open', 'high', 'low', 'close']:
            data[f'{col.capitalize()}_Lag_{lag}'] = data.groupby('symbol')[col].
    return data

# Calculate MACD
def calculate_macd(data):
    data['MACD'] = data['EMA_12'] - data['EMA_26']
    return data

# Fill NaN values
def fill_missing_values(data):
    data.fillna(method='bfill', inplace=True)
    return data

# Main function to run all feature engineering steps
def feature_engineering_pipeline(data):
    data = preprocess_data(data)
    data = calculate_daily_returns(data)
    data = calculate_sma(data)
    data = add_ema(data)
    data = calculate_volatility(data)
    data = add_rsi(data)
    data = calculate_bollinger_bands(data)
    data = add_time_features(data)
    data = create_lag_features(data)
    data = calculate_macd(data)
    data = fill_missing_values(data)
    return data


# Run the pipeline
data = feature_engineering_pipeline(data)

# Save to CSV
data.to_csv('engineered_stocks_data.csv', index=False)
```

```
C:\Users\jeffr\AppData\Local\Temp\ipykernel_3780\3387600118.py:78: FutureWarning:
DataFrame.fillna with 'method' is deprecated and will raise in a future version. U
se obj.ffill() or obj.bfill() instead.
  data.fillna(method='bfill', inplace=True)
```

In [19]:
```
data.head()
```

Out[19]:

| | date | symbol | open | high | low | close | adj_close |
|---|---|---|---|---|---|---|---|
| **0** | 2022-01-03 00:00:00+00:00 | AAPL | 177.830002 | 182.880005 | 177.710007 | 182.009995 | 179.273621 |
| **53** | 2022-01-04 00:00:00+00:00 | AAPL | 182.630005 | 182.940002 | 179.119995 | 179.699997 | 176.998337 |
| **82** | 2022-01-05 00:00:00+00:00 | AAPL | 179.610001 | 180.169998 | 174.639999 | 174.919998 | 172.290192 |
| **106** | 2022-01-06 00:00:00+00:00 | AAPL | 172.699997 | 175.300003 | 171.639999 | 172.000000 | 169.414124 |
| **138** | 2022-01-07 00:00:00+00:00 | AAPL | 172.889999 | 174.139999 | 171.029999 | 172.169998 | 169.581528 |

5 rows × 39 columns

In [20]:
```
# Display the first few rows of the engineered DataFrame
data.columns
```

Out[20]:
```
Index(['date', 'symbol', 'open', 'high', 'low', 'close', 'adj_close', 'volume',
       'beta', 'company_name', 'daily_return', 'SMA_7', 'SMA_14', 'SMA_20',
       'EMA_7', 'EMA_14', 'EMA_12', 'EMA_26', 'Volatility_7d',
       'Volatility_14d', 'RSI_14', 'BB_Upper', 'BB_Lower', 'Day_of_Week',
       'Month', 'Quarter', 'Open_Lag_1', 'High_Lag_1', 'Low_Lag_1',
       'Close_Lag_1', 'Open_Lag_2', 'High_Lag_2', 'Low_Lag_2', 'Close_Lag_2',
       'Open_Lag_3', 'High_Lag_3', 'Low_Lag_3', 'Close_Lag_3', 'MACD'],
      dtype='object')
```

## feature selection using correlation

In [21]:
```
correlation_matrix = data.drop(columns=['symbol', 'company_name']).corr()

correlation_matrix
```

Out[21]:

| | date | open | high | low | close | adj_close | volume |
|---|---|---|---|---|---|---|---|
| **date** | 1.000000 | 0.053200 | 0.050498 | 0.056291 | 0.053491 | 0.066488 | 0.000110 |
| **open** | 0.053200 | 1.000000 | 0.999837 | 0.999821 | 0.999645 | 0.999352 | -0.142111 |
| **high** | 0.050498 | 0.999837 | 1.000000 | 0.999757 | 0.999823 | 0.999568 | -0.138550 |
| **low** | 0.056291 | 0.999821 | 0.999757 | 1.000000 | 0.999837 | 0.999502 | -0.145999 |
| **close** | 0.053491 | 0.999645 | 0.999823 | 0.999837 | 1.000000 | 0.999701 | -0.142217 |
| **adj_close** | 0.066488 | 0.999352 | 0.999568 | 0.999502 | 0.999701 | 1.000000 | -0.135025 |
| **volume** | 0.000110 | -0.142111 | -0.138550 | -0.145999 | -0.142217 | -0.135025 | 1.000000 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **beta** | 0.008132 | 0.385011 | 0.388515 | 0.381310 | 0.384806 | 0.397135 | 0.474434 |
| **daily_return** | 0.032374 | 0.004024 | 0.011419 | 0.012238 | 0.019665 | 0.019798 | 0.037717 |
| **SMA_7** | 0.050718 | 0.999111 | 0.999042 | 0.998925 | 0.998862 | 0.998570 | -0.142538 |
| **SMA_14** | 0.048149 | 0.997778 | 0.997744 | 0.997501 | 0.997488 | 0.997203 | -0.142823 |
| **SMA_20** | 0.046307 | 0.996615 | 0.996610 | 0.996297 | 0.996324 | 0.996044 | -0.143163 |
| **EMA_7** | 0.050240 | 0.999392 | 0.999363 | 0.999233 | 0.999211 | 0.998919 | -0.142574 |
| **EMA_14** | 0.046340 | 0.998413 | 0.998404 | 0.998133 | 0.998146 | 0.997860 | -0.143010 |
| **EMA_12** | 0.047461 | 0.998704 | 0.998685 | 0.998452 | 0.998452 | 0.998164 | -0.142898 |
| **EMA_26** | 0.039601 | 0.996620 | 0.996682 | 0.996210 | 0.996308 | 0.996030 | -0.143472 |
| **Volatility_7d** | -0.173334 | 0.055276 | 0.059774 | 0.050660 | 0.055339 | 0.059601 | 0.488032 |
| **Volatility_14d** | -0.183007 | 0.069127 | 0.073463 | 0.064531 | 0.068981 | 0.073971 | 0.525914 |
| **RSI_14** | -0.001936 | -0.172480 | -0.174806 | -0.170310 | -0.172693 | -0.174776 | -0.170009 |
| **BB_Upper** | 0.031135 | 0.993547 | 0.993899 | 0.992844 | 0.993228 | 0.993275 | -0.118506 |
| **BB_Lower** | 0.063415 | 0.993295 | 0.992880 | 0.993422 | 0.993040 | 0.992385 | -0.170498 |
| **Day_of_Week** | -0.007695 | 0.000033 | 0.000198 | -0.000324 | -0.000170 | -0.000223 | 0.023583 |
| **Month** | 0.233936 | -0.006832 | -0.007460 | -0.006042 | -0.006852 | -0.003336 | -0.013895 |
| **Quarter** | 0.240877 | -0.004965 | -0.005522 | -0.004232 | -0.004939 | -0.001368 | -0.011924 |
| **Open_Lag_1** | 0.052135 | 0.999382 | 0.999259 | 0.999176 | 0.999043 | 0.998753 | -0.141938 |
| **High_Lag_1** | 0.049415 | 0.999564 | 0.999494 | 0.999298 | 0.999221 | 0.998968 | -0.138924 |
| **Low_Lag_1** | 0.055218 | 0.999588 | 0.999382 | 0.999458 | 0.999239 | 0.998908 | -0.145374 |
| **Close_Lag_1** | 0.052384 | 0.999738 | 0.999589 | 0.999549 | 0.999390 | 0.999094 | -0.142163 |
| **Open_Lag_2** | 0.051064 | 0.998783 | 0.998661 | 0.998539 | 0.998431 | 0.998144 | -0.142087 |
| **High_Lag_2** | 0.048346 | 0.998952 | 0.998893 | 0.998657 | 0.998610 | 0.998361 | -0.139172 |
| **Low_Lag_2** | 0.054159 | 0.998974 | 0.998787 | 0.998801 | 0.998619 | 0.998292 | -0.145317 |
| **Close_Lag_2** | 0.051307 | 0.999117 | 0.998987 | 0.998890 | 0.998758 | 0.998466 | -0.142223 |
| **Open_Lag_3** | 0.050000 | 0.998182 | 0.998083 | 0.997925 | 0.997849 | 0.997565 | -0.142159 |
| **High_Lag_3** | 0.047262 | 0.998358 | 0.998302 | 0.998036 | 0.998004 | 0.997757 | -0.139371 |
| **Low_Lag_3** | 0.053098 | 0.998361 | 0.998198 | 0.998174 | 0.998026 | 0.997701 | -0.145337 |
| **Close_Lag_3** | 0.050219 | 0.998502 | 0.998385 | 0.998247 | 0.998136 | 0.997846 | -0.142321 |
| **MACD** | 0.201049 | 0.169336 | 0.167326 | 0.173252 | 0.170791 | 0.170519 | -0.002558 |

37 rows × 37 columns

In [22]:
```
data.isna().sum()
```

Out[22]:
```
date                0
symbol              0
open                0
high                0
low                 0
close               0
adj_close           0
volume              0
beta                0
company_name        0
daily_return        0
SMA_7               0
SMA_14              0
SMA_20              0
EMA_7               0
EMA_14              0
EMA_12              0
EMA_26              0
Volatility_7d       0
Volatility_14d      0
RSI_14            691
BB_Upper            0
BB_Lower            0
Day_of_Week         0
Month               0
Quarter             0
Open_Lag_1          0
High_Lag_1          0
Low_Lag_1           0
Close_Lag_1         0
Open_Lag_2          0
High_Lag_2          0
Low_Lag_2           0
Close_Lag_2         0
Open_Lag_3          0
High_Lag_3          0
Low_Lag_3           0
Close_Lag_3         0
MACD                0
dtype: int64
```

In [23]:
```
data.dropna(subset=['RSI_14'], inplace=True)
```

## Feature Selection Using Correlation Analysis

This code performs feature selection by calculating the correlation of various features with specific target variables. The objective is to identify features that are highly correlated with the target features ( `adj_close` , `BB_Upper` , and `BB_Lower` ) to retain only the most relevant features for further analysis and modeling.

### Code Walkthrough

1. **Select Numeric Features**:

1. **Select Numeric Features**:
   First, we select only the numeric columns from the dataset, as these are suitable for correlation analysis. Non-numeric columns, like dates or categorical variables, are excluded.

2. **Define Targets and Set Correlation Threshold**:
   We define the target features ( `adj_close` , `BB_Upper` , `BB_Lower` ) and set a correlation threshold ( `corr_threshold` ) of 0.5. Only features with an absolute correlation above this threshold will be selected.

3. **Calculate Absolute Correlations**:
   For each target feature, we calculate the absolute correlation of all numeric features with that target. This provides a dictionary of correlations for each target.

4. **Combine Correlations into a DataFrame**:
   The individual correlation results for each target are combined into a single DataFrame for easy comparison across targets.

5. **Select Highly Correlated Features**:
   We identify features that meet the correlation threshold for all target features by filtering the `correlation_df` . Only features with a correlation above the threshold (0.5) across all targets are selected.

6. **Filter the Original DataFrame**:
   Finally, we filter the original data to retain only the selected high-correlation features, creating a new DataFrame ( `filtered_df` ) for further analysis.

7. **Output the Selected Features**:
   The selected features are printed to verify the features chosen based on the correlation criteria.

In [24]:

```
# Step 1: Select only numeric columns for correlation analysis
data_numeric = data.select_dtypes(include=[float, int])

# Step 2: Define target features for correlation check and set threshold
targets = ['adj_close', 'BB_Upper', 'BB_Lower']
corr_threshold = 0.7

# Step 3: Calculate absolute correlations for each target feature with all other
correlations = {target: data_numeric.corr()[target].abs() for target in targets}

# Step 4: Combine individual target correlations into a single DataFrame
correlation_df = pd.DataFrame(correlations)

# Step 5: Identify features that have a high correlation (> threshold) with all 
selected_features = correlation_df[(correlation_df > corr_threshold).all(axis=1)

# Step 6: Filter the original data to include only these selected high-correlati
filtered_df = data[selected_features]

# Output the selected features for verification
print("Selected features with high correlation to targets:", selected_features.t
```

```
Selected features with high correlation to targets: ['open', 'high', 'low', 'clos
e', 'adj_close', 'SMA_7', 'SMA_14', 'SMA_20', 'EMA_7', 'EMA_14', 'EMA_12', 'EMA_2
6', 'BB_Upper', 'BB_Lower', 'Open_Lag_1', 'High_Lag_1', 'Low_Lag_1', 'Close_Lag_
1', 'Open_Lag_2', 'High_Lag_2', 'Low_Lag_2', 'Close_Lag_2', 'Open_Lag_3', 'High_La
g_3', 'Low_Lag_3', 'Close_Lag_3']
```

## Scaling and PCA Transformation

This code performs two key steps—scaling and Principal Component Analysis (PCA)—to prepare the data for dimensionality reduction. By reducing the dataset to its most significant components, PCA allows us to capture the most important information in a lower-dimensional space, which can help simplify analysis and improve model performance.

### Code Walkthrough

1. **Scaling the Data**:
   We use `StandardScaler` to standardize the features, ensuring each feature has a mean of 0 and a standard deviation of 1. This step is crucial for PCA because it removes scale disparities among features, allowing PCA to correctly capture variance across them.

2. **Applying PCA**:
   We apply PCA to reduce the dataset to a specified number of principal components ( `n_components` ). Here, we use 3 components, though this can be adjusted. PCA helps to capture the maximum variance in fewer components, making the data more manageable.

3. **Creating the PCA DataFrame**:
   The transformed data is then converted into a DataFrame ( `pca_df` ), with columns labeled as `PC1` , `PC2` , etc., based on the number of components. This labeled DataFrame makes it easy to interpret and visualize the principal components.

4. **Outputting the Results**:
   Finally, we print the PCA-transformed DataFrame to verify the results.

In [25]:

```
# Step 1: Scale the Data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(filtered_df)  # Fit and transform the selecte

# Step 2: Apply PCA
n_components = 3  # Define the number of principal components to retain
pca = PCA(n_components=n_components)
pca_data = pca.fit_transform(scaled_data)  # Fit PCA model and transform scaled 

# Step 3: Convert PCA Results to DataFrame
pca_df = pd.DataFrame(
    data=pca_data,
```

```
    columns=[f'PC{i+1}' for i in range(n_components)]
)

# Output the resulting DataFrame with principal components
print("PCA-transformed Data:")
print(pca_df)
```

```
PCA-transformed Data:
            PC1       PC2       PC3
0      0.850246 -0.094033  0.057763
1      0.850717 -0.097687  0.056536
2      0.818438 -0.070280  0.053164
3      0.758683 -0.020077  0.047811
4      0.703182  0.012705  0.043185
...         ...       ...       ...
19675 -3.578577 -0.053657  0.037715
19676 -3.530860 -0.073116  0.042235
19677 -3.486196 -0.089970  0.045755
19678 -3.446883 -0.091683  0.045255
19679 -3.394188 -0.094401  0.050033

[19680 rows x 3 columns]
```

# Modelling

## K-Means Clustering and 3D Visualization

## 1. Apply K-Means Clustering

In this step, we perform K-Means clustering to group the stocks into three distinct risk categories based on their principal components (PCs) derived from PCA. The K-Means algorithm assigns each data point to a cluster based on the similarities of their feature values.

In [ ]:

```
from mpl_toolkits.mplot3d import Axes3D
from sklearn.cluster import KMeans, AgglomerativeClustering

# Create a KMeans model with 3 clusters
kmeans = KMeans(n_clusters=3, random_state=42)
data['Risk_Cluster'] = kmeans.fit_predict(pca_data)

# Map the cluster labels
cluster_mapping = {0: 'Low Risk', 1: 'Medium Risk', 2: 'High Risk'}
data['Risk_Label'] = data['Risk_Cluster'].map(cluster_mapping)
```

```
c:\Users\jeffr\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureW
arning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set th
e value of `n_init` explicitly to suppress the warning
  warnings.warn(
```

- We specify 3 clusters for the risk categories: Low, Medium, and High.
- fit_predict(pca_df): This method fits the K-Means model on the pca_df and assigns the

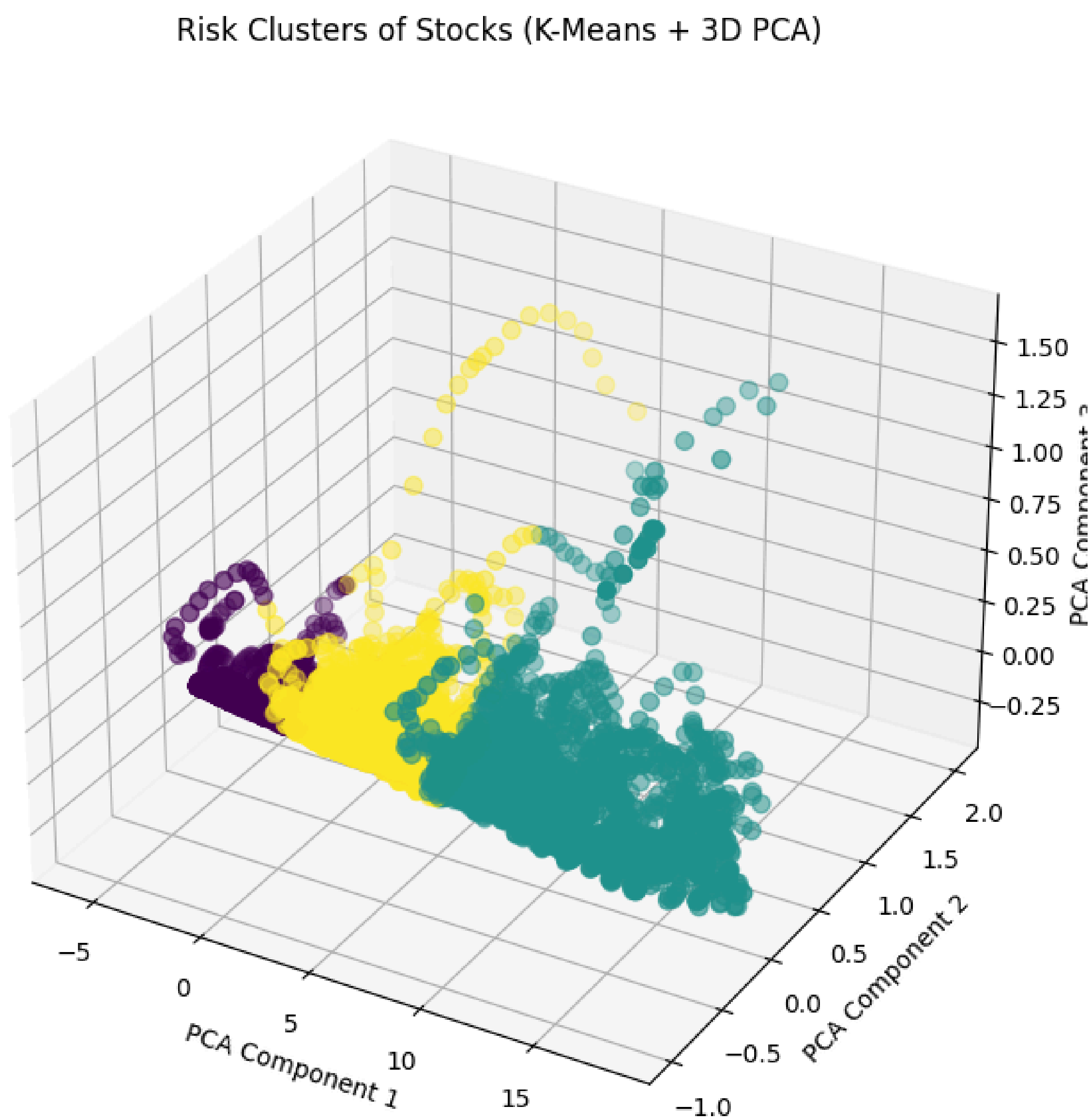
predicted cluster labels to a new column Risk_Cluster in the data.

In [29]:

```
# Create a 3D scatter plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(pca_data[:, 0], pca_data[:, 1], pca_data[:, 2],
           c=data['Risk_Cluster'], cmap='viridis', s=50)

ax.set_xlabel('PCA Component 1')
ax.set_ylabel('PCA Component 2')
ax.set_zlabel('PCA Component 3')
ax.set_title('Risk Clusters of Stocks (K-Means + 3D PCA)')
plt.show()
```




An alternative means of clustering is using the Agglomerative clustering method.

In [33]:

```
# use AgglomerativeClustering
hierarchical = AgglomerativeClustering(n_clusters=3)
```

```
hierarchical_labels = hierarchical.fit_predict(pca_data)
```

In [44]:

```
# save new dataframe for use in recommendations
data.to_csv('stock_flask.csv', index=False)
```

In [39]:

```
# save results
data[['date', 'symbol', 'Risk_Label']].to_csv('stock_risk_clusters.csv', index=F
```

## Silhouette Score Calculation

After applying K-Means clustering to group the stocks into risk categories, we can evaluate the quality of the clustering using the **Silhouette Score**. The Silhouette Score measures how well each data point has been clustered. A higher Silhouette Score indicates that the data points are well matched to their own cluster and poorly matched to neighboring clusters.

### 1. Calculate the Silhouette Score

The `silhouette_score` function from `sklearn.metrics` calculates the Silhouette Score, which is based on two factors:

- **Cohesion**: How similar a data point is to other points within the same cluster.
- **Separation**: How different a data point is from points in other clusters.

In [48]:

```
from sklearn.metrics import silhouette_score

silhouette_avg = silhouette_score(pca_df, data['Risk_Cluster'])
print(f'Silhouette Score: {silhouette_avg:.3f}')
```

```
Silhouette Score: 0.595
```

In [ ]:

```
# calculate silhouette score
hierarchical_silhouette = silhouette_score(pca_data, hierarchical_labels)
hierarchical_silhouette
```

Out[ ]:

```
0.5540925985901247
```

Based on the silhouette score we saw that a higher silhouette score is achieved using k-means compared to Agglomerative clustering. The model of choice therefore will be k-means because the clusters formed are more distinct and not as similar.

In [41]:

```
data['Risk_Label'].value_counts()
```

Out[41]:

```
Risk_Label
Low Risk       9120
High Risk      7829
Medium Risk    2731
```

```
Name: count, dtype: int64
```

In [42]:

```python
df = pd.read_csv('stock_risk_clusters.csv')
```

In [43]:

```python
df.tail(20)
```

Out[43]:

| | date | symbol | Risk_Label |
|---|---|---|---|
| **19660** | 2022-01-07 00:00:00+00:00 | XOM | Low Risk |
| **19661** | 2022-01-10 00:00:00+00:00 | XOM | Low Risk |
| **19662** | 2022-01-11 00:00:00+00:00 | XOM | Low Risk |
| **19663** | 2022-01-12 00:00:00+00:00 | XOM | Low Risk |
| **19664** | 2022-01-13 00:00:00+00:00 | XOM | Low Risk |
| **19665** | 2022-01-14 00:00:00+00:00 | XOM | Low Risk |
| **19666** | 2022-01-18 00:00:00+00:00 | XOM | Low Risk |
| **19667** | 2022-01-19 00:00:00+00:00 | XOM | Low Risk |
| **19668** | 2022-01-20 00:00:00+00:00 | XOM | Low Risk |
| **19669** | 2022-01-21 00:00:00+00:00 | XOM | Low Risk |
| **19670** | 2022-01-24 00:00:00+00:00 | XOM | Low Risk |
| **19671** | 2022-01-25 00:00:00+00:00 | XOM | Low Risk |
| **19672** | 2022-01-26 00:00:00+00:00 | XOM | Low Risk |
| **19673** | 2022-01-27 00:00:00+00:00 | XOM | Low Risk |
| **19674** | 2022-01-28 00:00:00+00:00 | XOM | Low Risk |
| **19675** | 2022-01-31 00:00:00+00:00 | XOM | Low Risk |
| **19676** | 2022-02-01 00:00:00+00:00 | XOM | Low Risk |
| **19677** | 2022-02-02 00:00:00+00:00 | XOM | Low Risk |
| **19678** | 2022-02-03 00:00:00+00:00 | XOM | Low Risk |
| **19679** | 2022-02-04 00:00:00+00:00 | XOM | Low Risk |