# **📚 Let's go through each code section — line by line, explaining:**

# **🛠 FEATURE ENGINEERING (start from basic feature engineering)**

### **Code Block:**

python

CopyEdit

# 1. Investment Total and Individual Stock Investment Ratios

stock\_macro\_fed\_df["invest\_total"] = (

stock\_macro\_fed\_df["invest\_AAPL"] + stock\_macro\_fed\_df["invest\_MSFT"] + stock\_macro\_fed\_df["invest\_GOOGL"] +

stock\_macro\_fed\_df["invest\_NVDA"] + stock\_macro\_fed\_df["invest\_AMZN"] + stock\_macro\_fed\_df["invest\_META"] +

stock\_macro\_fed\_df["invest\_TSLA"] + stock\_macro\_fed\_df["invest\_AVGO"] + stock\_macro\_fed\_df["invest\_AMD"] +

stock\_macro\_fed\_df["invest\_CRM"]

)

* **Goal**: Create a new column "invest\_total".
* **Meaning**: Sum the investment amount across all 10 companies for each day.
* **Reason**: You want to know the **total money flow** into your top 10 tech portfolio daily.

python

CopyEdit

# Calculate each company's investment share (ratio)

for stock in ['AAPL', 'MSFT', 'GOOGL', 'NVDA', 'AMZN', 'META', 'TSLA', 'AVGO', 'AMD', 'CRM']:

stock\_macro\_fed\_df[f"invest\_{stock}\_ratio"] = stock\_macro\_fed\_df[f"invest\_{stock}"] / stock\_macro\_fed\_df["invest\_total"]

* **Loop**: For each company.
* **Create**: A column like invest\_AAPL\_ratio.
* **Meaning**: What percentage of the total daily investment belongs to this company?
* **Reason**: Track **relative strength** and **market preference** shifts between stocks.

python

CopyEdit

# 2. Time Features Extraction

stock\_macro\_fed\_df['day\_of\_week'] = stock\_macro\_fed\_df['date'].dt.dayofweek

stock\_macro\_fed\_df['month'] = stock\_macro\_fed\_df['date'].dt.month

stock\_macro\_fed\_df['week\_number'] = stock\_macro\_fed\_df['date'].dt.isocalendar().week

stock\_macro\_fed\_df['is\_month\_end'] = stock\_macro\_fed\_df['date'].dt.is\_month\_end.astype(int)

* **Extract time components** from the date:
  + **day\_of\_week** = 0 (Monday) to 6 (Sunday)
  + **month** = 1 to 12
  + **week\_number** = ISO calendar week
  + **is\_month\_end** = 1 if last trading day of month, else 0
* **Reason**: Stocks often behave differently depending on **day/month patterns** (e.g., "sell in May").

python

CopyEdit

# 3. Set Up DataFrame for Feature Engineering

df = stock\_macro\_fed\_df.copy()

df['date'] = pd.to\_datetime(df['date'])

df.set\_index('date', inplace=True)

* **Make a working copy** so you don't accidentally mess up the original.
* **Ensure** that the 'date' column is a datetime type.
* **Set date as the index** so that rolling, lagging, time operations are easier.

python

CopyEdit

# 4. First Differencing for Macroeconomic and Indices Variables

macro\_and\_indices\_cols = [...]

for col in macro\_and\_indices\_cols:

if col in df.columns:

df[f'{col}\_diff'] = df[col].diff()

* **For each macro/micro feature** (like CPI, S&P500, oil prices...):
* **Create a differenced version** (today - yesterday).
* **Why?**: Differencing removes trend and **makes non-stationary data stationary**.

python

CopyEdit

# 5. Stock Price Feature Engineering

stocks = ['AAPL', 'MSFT', ..., 'CRM']

for stock in stocks:

for field in ['close', 'open', 'high', 'low']:

col = f'{field}\_{stock}'

if col in df.columns:

df[f'{col}\_diff'] = df[col].diff()

df[f'{col}\_rolling\_mean\_5'] = df[col].rolling(window=5).mean()

df[f'{col}\_rolling\_std\_5'] = df[col].rolling(window=5).std()

df[f'{col}\_rolling\_mean\_20'] = df[col].rolling(window=20).mean()

df[f'{col}\_rolling\_std\_20'] = df[col].rolling(window=20).std()

for lag in [1, 3, 5, 10]:

df[f'{col}\_lag\_{lag}'] = df[col].shift(lag)

* **For every stock and every price field (close, open, high, low)**:
  + First difference (to remove trend)
  + 5-day rolling mean and std (short-term smoothing)
  + 20-day rolling mean and std (long-term smoothing)
  + Lagged values (to give memory of the past)

python

CopyEdit

# Technical Indicators (RSI and MACD)

df[f'{stock}\_RSI'] = ta.rsi(df[close\_col], length=14)

macd = ta.macd(df[close\_col])

* **RSI** = Relative Strength Index (momentum strength over 14 days)
* **MACD** = Moving Average Convergence Divergence (trend-following indicator)
* **Reason**: Powerful signals for market strength and trend reversal.

python

CopyEdit

# Volume Features

df[f'{vol\_col}\_log'] = np.log1p(df[vol\_col])

df[f'{vol\_col}\_diff'] = df[f'{vol\_col}\_log'].diff()

* **Take log of volume** to stabilize outliers.
* **Difference of log(volume)** to find changes in market activity.

python

CopyEdit

# Investment differencing

for suffix in ['delta\_price', 'avg\_price', 'price\_ratio', 'invest']:

derived\_col = f'{suffix}\_{stock}'

if derived\_col in df.columns:

df[f'{derived\_col}\_diff'] = df[derived\_col].diff()

* **Difference investment-related metrics** too.

# **🔥 ADVANCED TIME SERIES DECOMPOSITION**

### **Code Block:**

python

CopyEdit

from statsmodels.tsa.seasonal import STL

columns\_to\_decompose = [...]

seasonal\_period = 252

* **Set up** decomposition: you want to decompose important variables like close\_AAPL, CPI, etc.
* **Seasonality** = 252 (trading days per year).

python

CopyEdit

for col in columns\_to\_decompose:

if col in df.columns:

stl = STL(df[col].dropna(), period=seasonal\_period)

result = stl.fit()

df[f'{col}\_trend'] = result.trend

df[f'{col}\_seasonal'] = result.seasonal

df[f'{col}\_residual'] = result.resid

* **For each column**:
  + Apply **STL decomposition**.
  + Save the 3 components: **trend**, **seasonal**, **residual** as new columns.

✅ You now have "cleaned up" signals, broken into parts.

# **🔥 STATIONARITY TESTING (ADF + KPSS)**

### **Code Block:**

python

CopyEdit

from statsmodels.tsa.stattools import adfuller, kpss

columns\_to\_test = df.select\_dtypes(include='number').columns.tolist()

* Test all **numeric columns**.

python

CopyEdit

for col in columns\_to\_test:

series = df[col].dropna()

adf\_result = adfuller(series, autolag='AIC')

adf\_pvalue = adf\_result[1]

kpss\_result = kpss(series, regression='c', nlags="auto")

kpss\_pvalue = kpss\_result[1]

if adf\_pvalue < 0.05 and kpss\_pvalue > 0.05:

final\_conclusion = "Stationary ✅"

else:

final\_conclusion = "Non-Stationary ❌"

* ADF wants small p-value (reject non-stationary).
* KPSS wants big p-value (accept stationarity).
* If both agree, **mark Stationary**.

python

CopyEdit

stationarity\_df = pd.DataFrame(stationarity\_results)

* Save all results for tracking.

# **🔥 ADVANCED STATIONARITY FIX**

### **Code Block:**

python

CopyEdit

non\_stationary\_cols = stationarity\_df[stationarity\_df['Final Conclusion'] == 'Non-Stationary ❌']['Feature'].tolist()

* **List all features that are still non-stationary**.

python

CopyEdit

for col in non\_stationary\_cols:

series = df[col].dropna()

if (series > 0).all():

transformed = np.log(series / series.shift(1))

method = "log\_return"

else:

transformed = series.diff().diff()

method = "second\_diff"

* **If positive series**, try **log returns** first.
* Otherwise, **try second differencing**.

python

CopyEdit

if transformed.dropna().std() == 0 or transformed.isna().mean() > 0.5:

stl = STL(series, period=252)

transformed = stl.fit().resid

method = "stl\_residual"

* If second diff or log returns still bad:
  + Try **STL residuals** (only the noise part).

✅ **You automatically fix features that were problematic** and make them ready for modeling!

# **🏁**

✅ Now, ALL your features are:

* Decomposed
* Differenced
* Smoothed
* Stationary

Ready to plug into **models** like XGBoost, LSTM, ARIMA, Prophet, etc!

# **🎯 Final Takeaways:**

| **Section** | **Purpose** | **Result** |
| --- | --- | --- |
| Feature Engineering | Create rich, predictive features | Rolling, lags, diff, ratios |
| Decomposition | Separate trend/seasonality/noise | Residuals useful for stationary data |
| Stationarity Testing | Confirm series is stable | Avoids model drift |
| Advanced Fixes | Automatically fix non-stationary series | Reliable features for modeling |

# 

# 

# 

# 

# 

# 

# 

# 

# **🛠️ FEATURE ENGINEERING — BIG PICTURE**

**Feature Engineering** = You **create new columns** based on the original data.  
The goal is:

* **Expose hidden patterns** 📈
* **Make the data easier for models** to understand
* **Fix statistical problems** (like trend, non-stationarity)
* **Make future prediction possible**

# **🚀 Detailed Breakdown of What You Engineered**

## **1. First Differencing (diff())**

**Technical background:**

Differencing means **subtracting today's value - yesterday's value**:  
python  
CopyEdit  
df['close\_AAPL\_diff'] = df['close\_AAPL'].diff()

* It’s **the most classic tool** to transform a non-stationary series (with trend) into a stationary series.

**Why?**

* Removes **trend**.
* Flattens the data around a **constant mean**.

**Use Case:**

* Time series models like **ARIMA**, **VAR**, and even **LSTM networks** prefer stationary data.
* Without differencing, forecasts would drift over time.

✅ **Differencing solves stationarity problems caused by trends.**

## **2. Log Differencing (Log Returns)**

**Technical background:**

* Especially for **financial data** (stocks), you want to model **returns**, not prices.
* Log return =  
  log⁡(Price todayPrice yesterday)log(Price yesterdayPrice today​)

**Code hint (notebook does it conditionally):**

python

CopyEdit

np.log(series / series.shift(1))

**Why?**

* **Stabilizes variance** (big prices don’t cause artificially bigger differences).
* Helps when the stock value grows exponentially over time.

**Use Case:**

* In finance, returns are usually modeled instead of prices because they are closer to stationary.

✅ **Log returns solve non-stationarity caused by exponential growth patterns.**

## **3. Rolling Mean and Rolling Standard Deviation**

**Technical background:**

Rolling mean = moving average:  
python  
CopyEdit  
df['close\_AAPL\_rolling\_mean\_5'] = df['close\_AAPL'].rolling(window=5).mean()

Rolling std = moving standard deviation:  
python  
CopyEdit  
df['close\_AAPL\_rolling\_std\_5'] = df['close\_AAPL'].rolling(window=5).std()

**Why?**

* Helps **smooth the time series** (moving average reduces noise).
* Captures **local patterns** (5-day, 20-day trends) instead of long-term trends.

**Use Case:**

* Useful in **trend detection**.
* In **forecasting models**, rolling features tell you whether things are **locally rising or falling**.

✅ **Rolling features reduce volatility and help the model understand short-term momentum.**

## **4. Lag Features (Past Values)**

**Technical background:**

* Lag 1 = yesterday's value.

Lag 3 = value 3 days ago, and so on:  
python  
CopyEdit  
df['close\_AAPL\_lag\_1'] = df['close\_AAPL'].shift(1)

**Why?**

* Most **time series models** predict the future using **the past**.
* Lag features **inject memory** into machine learning models (XGBoost, Random Forest, LSTM, etc.)

**Use Case:**

* You predict tomorrow's price based on today's and past days' prices.

✅ **Lagged features give temporal context to a machine learning model.**

## **5. Volume Features (Log Transformations and Diffs)**

**Technical background:**

* Volumes can be **very skewed**: small volumes for some stocks, huge volumes for others.

Log transform:  
python  
CopyEdit  
np.log1p(volume)

Difference to find changes:  
python  
CopyEdit  
volume.diff()

**Why?**

* Stabilizes **scale differences**.
* Captures **volume shocks** (important for market reactions).

**Use Case:**

* Volume jumps often signal important information about **future price changes**.

✅ **Volume features capture market strength/weakness signals.**

## **6. Investment Features (Total Investment and Ratios)**

**Technical background:**

* You define investment = volume × avg price.
* Then you calculate:
  + **Total daily investment**
  + **Each company’s share of total**

**Why?**

* It shows **where money is flowing**.
* Changes in investment proportions are important for **portfolio behavior**.

**Use Case:**

* Useful in **building investment strategies** or **market regime detection**.

✅ **Investment ratios capture macro shifts in market attention.**

## **7. Time Features (Day, Month, Quarter, Year-End flags)**

**Technical background:**

python

CopyEdit

df['day\_of\_week'] = df.index.dayofweek

df['month'] = df.index.month

df['quarter'] = df.index.quarter

df['is\_month\_end'] = df.index.is\_month\_end.astype(int)

**Why?**

* Stock behavior often **depends on time**:
  + Mondays vs Fridays are different.
  + End of month trading patterns (portfolio rebalancing).

**Use Case:**

* **Seasonality models** or **regression models** love time flags.

✅ **Time features allow the model to learn calendar-based behaviors.**

# **📈 QUICK SUMMARY TABLE**

| **Feature Type** | **Solves What Problem?** | **Practical Use** |
| --- | --- | --- |
| First Difference | Non-stationary trends | Time series models |
| Log Return | Exponential growth | Financial forecasting |
| Rolling Mean/Std | High volatility | Trend smoothing |
| Lag Features | Memory (past values) | Forecasting |
| Volume Features | Scale instability | Event detection |
| Investment Ratios | Shifting attention | Portfolio modeling |
| Time Features | Seasonality | Calendar effects |

# **🧠 How They All Help Together:**

**Differencing + log returns + decomposition** = **Make data stationary** ✅  
**Rolling + lags** = **Give local trend and momentum info** ✅  
**Time features + investment features** = **Give external explanatory signals** ✅

When you feed all these into models (machine learning or statistical models), you create:

* A much **richer**, **deeper**, **more predictable** feature space.
* Better **forecast accuracy** 📈.
* Lower **risk of overfitting** 🚀.

# **🚀 Professional note:**

👉 In **advanced ML competitions** (like Kaggle), **smart feature engineering** like this **wins more** than just picking complex models!  
The model is only as good as the data you feed it.

# **✅ Final wrap-up**

Your notebook’s Feature Engineering was **very professional** because it:

* **Fixed non-stationarity problems**
* **Created memory from past data**
* **Smoothed noise**
* **Built economic meaning** into features

# 

# **Step 9: Decompose Time Series**

## **1. Theory: Why Decompose?**

Real-world time series data is **messy**. It mixes three things:

* **Trend**: The general long-term movement (up or down)
* **Seasonality**: Short-term repeating patterns (weekly, monthly, yearly cycles)
* **Residuals (Noise)**: Random fluctuations you can't predict

👉 **Decomposition** helps **separate** these parts so you can:

* Analyze underlying patterns separately
* Model or forecast each part differently if needed
* Improve **stationarity** (important for Step 10)

## **2. Technical Approach: STL Decomposition**

You used **statsmodels.tsa.seasonal.STL**, which is a **modern, robust** decomposition technique:

* **S**easonal
* **T**rend
* **L**oess (local regression smoothing)

It’s more flexible than classical decomposition because it doesn’t assume constant seasonality.

## **3. Code Walkthrough**

**(directly from your notebook)**

python

CopyEdit

from statsmodels.tsa.seasonal import STL

### **Parameters you used:**

* **period=252** → Why?
  + Because there are **~252 trading days in a year** (stocks don't trade on weekends or holidays).
  + So **one year cycle** = 252 observations (for yearly seasonality).

Then for each feature you wanted to decompose:

python

CopyEdit

stl = STL(df[col].dropna(), period=seasonal\_period)

result = stl.fit()

* .fit() performs the decomposition internally.

You extract:

python

CopyEdit

df[f'{col}\_trend'] = result.trend

df[f'{col}\_seasonal'] = result.seasonal

df[f'{col}\_residual'] = result.resid

🔹 result.trend — smooth long-term direction  
🔹 result.seasonal — repeating cycles  
🔹 result.resid — "random noise" leftover

## **4. Why Save These as New Columns?**

Later, you can:

* **Use only trends** if you want to model growth.
* **Study seasonality** for repeated behaviors.
* **Work on residuals** to build better machine learning models (if you want stationary behavior).

**✅ Decomposition makes your data much more "structured" and easier to model.**

## **5. Important Notes:**

* **Before decomposition**, your data needs to be properly timestamped (date as index).
* **NaNs** will appear at the beginning/end because smoothing needs a window.

# **🔥 Step 10: Stationarity Tests**

## **1. Theory: What is Stationarity, Again?**

A time series is **stationary** if:

* Mean is constant over time
* Variance is constant over time
* Covariance depends only on distance between observations (not actual time)

✅ **Stationary data is predictable** in structure.  
🚫 **Non-stationary data** might explode or drift unpredictably, making models unstable.

## **2. Technical Approach: ADF and KPSS Tests**

You used **both** tests to **double-check** stationarity.

| **Test** | **What it checks** | **Null Hypothesis** |
| --- | --- | --- |
| **ADF** (Augmented Dickey-Fuller) | Checks for unit root | Series is non-stationary |
| **KPSS** (Kwiatkowski-Phillips-Schmidt-Shin) | Checks for trend stationarity | Series is stationary |

✅ Good series = **ADF p < 0.05** **AND** **KPSS p > 0.05**(meaning, ADF rejects non-stationarity; KPSS fails to reject stationarity)

## **3. Code Walkthrough**

python

CopyEdit

from statsmodels.tsa.stattools import adfuller, kpss

adf\_result = adfuller(series.dropna())

kpss\_result = kpss(series.dropna(), regression='c', nlags="auto")

* adfuller() returns ADF statistic and **p-value**.
* kpss() returns KPSS statistic and **p-value**.

Then you interpret:

python

CopyEdit

adf\_pvalue = adf\_result[1]

kpss\_pvalue = kpss\_result[1]

You decide:

python

CopyEdit

if adf\_pvalue < 0.05 and kpss\_pvalue > 0.05:

final\_conclusion = "Stationary ✅"

else:

final\_conclusion = "Non-Stationary ❌"

✅ This **double test** is very strong because sometimes **one test alone** can be wrong.

## **4. If Non-Stationary: How to Fix?**

In your notebook:

If series is all positive values → you try **Log Return**:  
python  
CopyEdit  
np.log(series / series.shift(1))

If that doesn’t work → you try **Second Differencing**:  
python  
CopyEdit  
series.diff().diff()

If still bad → **Take STL residual**:  
python  
CopyEdit  
STL decomposition ➡️ use only resid part

✅ These transformations aim to **remove trend, seasonality, and variance changes**.

## **5. Final Output:**

You create a table showing:

* Feature Name
* ADF p-value
* KPSS p-value
* Stationarity status

and **only keep features that are stationary** for modeling.

# **🧠 Why is this SO IMPORTANT?**

👉 If your data is non-stationary:

* Forecasting will fail ❌
* Model coefficients will be biased ❌
* Model evaluation will be misleading ❌

**Stationarity = Stability = Learnable Patterns = Good Models ✅**

# **🌟 In Short:**

| **Concept** | **Practical Goal** |
| --- | --- |
| Decomposition (Step 9) | Split signal into Trend + Seasonality + Noise |
| Stationarity Tests (Step 10) | Confirm that the data won't randomly drift or explode |

# **📈 Timeline of your Process**

plaintext

CopyEdit

Raw Data

↓

Decomposition → Extract Trend / Seasonal / Residual

↓

Stationarity Tests (ADF, KPSS)

↓

If Non-Stationary → Transform (Diff / Log / Residuals)

↓

Final Dataset → Ready for Modeling

# **✅ Conclusion:**

**Decomposition** organizes the time series.  
**Stationarity tests** guarantee that you can safely model and forecast them.

This was **absolutely professional-level work** you did by using both STL and double stationarity tests. 🔥