```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
```

Fiscal Risk in Europe: Clustering, Resilience, and Forecasting

Goal: Build an interpretable fiscal risk framework for European countries using Eurostat government finance data.

Outputs:

- Clean panel dataset with **Debt-to-GDP**, **Deficit-to-GDP**, **Debt Growth**
- Country clusters (Stable / Moderate Risk / High Risk)
- Fiscal shock resilience index (GFC + COVID)
- Composite fiscal risk score + Europe choropleth + bubble chart
- Simple time-series forecasts (ARIMA, VAR) for a selected country

1) Setup

```
from sklearn.metrics import silhouette_score
from sklearn.decomposition import PCA
from scipy.stats import spearmanr
import itertools
import datetime, json, os
```

```
# Core
import pandas as pd
import numpy as np

# Modeling
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.cluster import KMeans

# Time series
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.api import VAR

# Visualization
import matplotlib.pyplot as plt
import plotly.express as px

# Display helpers (works in Colab; falls back gracefully elsewhere)
try:
```

```
    from google.colab import data_table
    data_table.enable_dataframe_formatter()
    IN_COLAB = True
except Exception:
    IN_COLAB = False

# Paths (Colab usually uses /content; this project uses uploaded files under /i
TSV_PATH = "/mnt/data/estat_gov_10dd_edpt1.tsv"  # uploaded with the notebook
CHOROPLETH_CSV = "/mnt/data/choropleth_data.csv" # optional (if you already ex|
```

2) Load Eurostat TSV (quick inspection)

```
df_raw = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/docx fiscal/estat_
print("Shape:", df_raw.shape)
print("First columns:", list(df_raw.columns[:5]))
display(df_raw.head(5))
```

```
Shape: (2103, 31)
First columns: ['freq,unit,sector,na_item,geo\\TIME_PERIOD', '1995 ', '1996 ',
Warning: Total number of columns (31) exceeds max_columns (20). Falling back t
```

	freq,unit,sector,na_item,geo\TIME_PERIOD	1995	1996	1997	:
0	A,MIO_EUR,S1,B1GQ,AT	183629.2	185944.9	186913.0	1939
1	A,MIO_EUR,S1,B1GQ,BE	220251.5	219965.3	223032.7	2310
2	A,MIO_EUR,S1,B1GQ,BG	14512.8	9829.8	10064.7	134
3	A,MIO_EUR,S1,B1GQ,CY	7596.0	7890.1	8414.3	91
4	A,MIO_EUR,S1,B1GQ,CZ	46333.6	53416.0	55174.6	601

5 rows × 31 columns

If you're in **Google Colab**, the table below is interactive (sort/filter).

```
if IN_COLAB:
    data_table.DataTable(df_raw.head(30))
else:
    display(df_raw.head(30))
```

3) Tidy the dataset into a panel

Eurostat stores multiple dimensions in the first column. We'll split those dimensions, keep the **government sector**, and reshape years into rows.

```
def tidy_eurostat_gov_tsv(df: pd.DataFrame) -> pd.DataFrame:
    """
    Tidy Eurostat gov finance TSV into long format, with strict filtering to av
    artificial duplication across unit/sector combinations.

    Returns long format with columns:
      country, year, na_item, unit, sector, value
    """
    first_col = df.columns[0]

    # Split dimensions from first column (Eurostat TSV standard)
    dims = df[first_col].astype(str).str.split(",", expand=True)
    dims.columns = ["freq", "unit", "sector", "na_item", "geo"]

    out = df.drop(columns=[first_col]).copy()
    out = pd.concat([dims, out], axis=1)

    # Clean geo
    out["geo"] = out["geo"].str.replace(r"\\TIME_PERIOD", "", regex=True).str.

    # Keep annual only
    out = out[out["freq"].eq("A")].copy()

    # --- Strict filters to avoid duplicates ---
    # GDP: B1GQ must be total economy S1 and monetary units MIO_EUR
    gdp_mask = (out["na_item"].eq("B1GQ") & out["unit"].eq("MIO_EUR") & out["s

    # Debt (GD) and Net lending/borrowing (B9): general government S13 and MIO_
    gov_mask = (out["na_item"].isin(["GD", "B9"]) & out["unit"].eq("MIO_EUR") 

    out = out[gdp_mask | gov_mask].copy()

    # Remove non-country aggregates (keep ISO-2 countries only)
    # Eurostat geo includes aggregates like EU27_2020, EA19, EA20 etc.
    # Keep only 2-letter codes PLUS EL (Greece)
    out = out[out["geo"].str.fullmatch(r"[A-Z]{2}")].copy()

    # Identify year columns
    year_cols = [c for c in out.columns if str(c).strip().isdigit()]

    # Melt to long
    long = out.melt(
        id_vars=["geo", "na_item", "unit", "sector"],
        value_vars=year_cols,
        var_name="year",
        value_name="value"
    )

    long["year"] = long["year"].astype(int)

    # Clean values:
    # ":" missing, and remove flags like "123.4 e", "56 p"
    long["value"] = (
        long["value"]
        .astype(str)
        .str.strip()
        .replace({":": np.nan, "": np.nan})
```

```
        .replace({":": np.nan, "": np.nan})
        .str.replace(r"[^0-9\.\-]", "", regex=True)
    )
    long["value"] = pd.to_numeric(long["value"], errors="coerce")

    long = long.dropna(subset=["value"]).rename(columns={"geo": "country"})

    # Uniqueness check: should be one value per (country, year, na_item)
    dup = long.duplicated(subset=["country", "year", "na_item"]).sum()
    if dup > 0:
        raise ValueError(f"Duplicate rows after filtering (count={dup}). Check

    return long
```

4) Build core fiscal metrics

We extract:

- **GDP** (`B1GQ`)
- **Government debt** (`GD`)
- **Net lending/borrowing** (`B9`) → treated as *deficit* (negative values are deficits)

```
df_long = tidy_eurostat_gov_tsv(df_raw)

# Pivot to wide form: one row per (country, year)
wide = df_long.pivot_table(index=["year", "country"], columns="na_item", valu

needed = ["B1GQ", "GD", "B9"]
wide = wide[needed].dropna().reset_index().rename(columns={"B1GQ":"GDP", "GD"

# Feature engineering
wide = wide.sort_values(["country", "year"])
wide["Debt_to_GDP"] = 100 * wide["Debt"] / wide["GDP"]
wide["Deficit_to_GDP"] = 100 * wide["Deficit"] / wide["GDP"]

# Annual debt growth (%)
wide["Debt_Growth"] = wide.groupby("country")["Debt"].pct_change() * 100

# For most analyses, we drop the first year per country (Debt_Growth is NaN)
df_metrics = wide.dropna(subset=["Debt_Growth"]).copy()

df_metrics = df_metrics[["year","country","Debt_to_GDP","Deficit_to_GDP","Deb
print("Metrics shape:", df_metrics.shape)
display(df_metrics.head())

def validate_metrics(df_metrics: pd.DataFrame):
    if (df_metrics["Debt_to_GDP"] < 0).any():
        raise ValueError("Negative Debt_to_GDP detected.")

    if df_metrics.duplicated(subset=["country", "year"]).any():
        raise ValueError("Duplicate country-year rows in df_metrics.")

    for col in ["GDP", "Debt", "Deficit"]:
        if df_metrics[col].isna().mean() > 0.01:
```

```
            print(f"Warning: {col} has >1% missing after cleaning.")

    extreme_debt = df_metrics["Debt_to_GDP"].max()
    if extreme_debt > 300:
        print("Warning: Debt_to_GDP > 300% detected; confirm unit/sector filt

    print("✅ Validation passed: df_metrics looks consistent.")

validate_metrics(df_metrics)
```

```
Metrics shape: (776, 8)
```

1 to 5 of 5 entries Filter

index	year	country	Debt_to_GDP	Deficit_to_GDP	Debt_Growth	G
25	1996	AT	67.31940483444289	-4.5736667152473665	-0.05469258376548103	185
50	1997	AT	63.54549977797157	-2.622396516026173	-5.114517842734689	186
76	1998	AT	64.78581015132477	-2.753792142290346	5.788601622566403	193
102	1999	AT	67.08008129420693	-2.6317694490776105	8.408979850410114	203
128	2000	AT	66.63779125715374	-2.4064201334420554	3.9107711937135248	212

Show 25 per page

```
✅ Validation passed: df_metrics looks consistent.
```

Notes on sign conventions

- `Deficit_to_GDP` will be **negative** in deficit years (because `B9` is net lending/borrowing).
- If you prefer a *positive deficit number*, use `-Deficit_to_GDP`.

5) Clustering countries by fiscal behavior

We cluster **country-year observations** using standardized features:

- Debt_to_GDP
- Deficit_to_GDP
- Debt_Growth

```
features = ["Debt_to_GDP", "Deficit_to_GDP", "Debt_Growth"]
X = df_metrics[features].copy()

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
def choose_k(X_scaled, k_min=2, k_max=8):
    rows = []
    for k in range(k_min, k_max+1):
        km = KMeans(n_clusters=k, random_state=42, n_init=25)
        labels = km.fit_predict(X_scaled)
        sil = silhouette_score(X_scaled, labels)
        rows.append((k, km.inertia_, sil))
    return pd.DataFrame(rows, columns=["k", "inertia", "silhouette"])
```

```
k_eval = choose_k(X_scaled, 2, 8)
display(k_eval)
# Choose K=3 as an interpretable baseline (Stable / Moderate Risk / High Risk)
kmeans = KMeans(n_clusters=3, random_state=42, n_init=25)
df_metrics["cluster"] = kmeans.fit_predict(X_scaled)

centroids = pd.DataFrame(scaler.inverse_transform(kmeans.cluster_centers_), co
centroids.index.name = "cluster"
print("Cluster centroids (original scale):")
display(centroids)
print("Cluster sizes:")
display(df_metrics["cluster"].value_counts().sort_index())
```

1 to 7 of 7 entries Filter

index	k	inertia	silhouette
0	2	1671.0145667869845	0.3088970762706734
1	3	1204.669981523232	0.35520021734845564
2	4	947.1765794796122	0.3104873030619993
3	5	794.8108061832893	0.29936373748566414
4	6	681.6381229599509	0.3033669668578583
5	7	606.7122484521371	0.3131972850955201
6	8	554.4651293653387	0.3149740421075922

Show 25 per page

```
Cluster centroids (original scale):
```

1 to 3 of 3 entries Filter

cluster	Debt_to_GDP	Deficit_to_GDP	Debt_Growth
0	43.16049725072932	-0.9245423819238889	4.196964684641882
1	41.38640687768245	-6.8626168873986035	32.51848771989783
2	102.00804273779335	-4.553949373481779	5.200763649101975

Show 25 per page

```
Cluster sizes:
```

cluster	count
0	474
1	90
2	212

dtype: int64

Next steps: Generate code with k_eval | New interactive sheet | Generate code with centroi

Country-level summary (dominant cluster + stability metrics)

```
country_summary = (
    df_metrics.groupby("country")
```

```
        .agg(
            avg_debt_to_gdp=("Debt_to_GDP","mean"),
            deficit_volatility=("Deficit_to_GDP","std"),
            avg_debt_growth=("Debt_Growth","mean"),
            dominant_cluster=("cluster", lambda s: s.mode().iat[0]),
            n_years=("year","nunique"),
        )
        .reset_index()
)

display(country_summary.sort_values("avg_debt_to_gdp", ascending=False).head(
```

1 to 10 of 10 entries Filter

index	country	avg_debt_to_gdp	deficit_volatility	avg_debt_growth	dominant_
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	
15	IT	122.2210142351545	2.1484667809029494	3.5220168004627417	
1	BE	104.72244895621103	2.1695760985778634	2.85505313673407З	
22	PT	93.35968408218146	2.918708675773693	5.666596035212437	
11	FR	83.49767513537151	1.838262914486688	5.470336097209943	
9	ES	75.70176586528748	3.8333939702246433	6.268010865230446	
3	CY	75.11146637628339	4.000636261167574	6.592605695549901	
0	AT	74.57648533412697	1.912598082655129	4.124584760842947	
13	HU	67.79934002042377	2.2148397206161863	6.260949305107462	
5	DE	66.02301422161432	1.9750906764111156	3.274001790138665	

Show 25 per page

6) Fiscal shock resilience index

We measure how strongly deficits deteriorate during crisis years.

Definition used here: average `Deficit_to_GDP` in crisis years (more negative = stronger deficit response). You can also compute *delta vs pre-crisis baseline*; both are shown.

```
crisis_years = [2008, 2009, 2020, 2021]

# Average deficit ratio during crisis years
crisis = df_metrics[df_metrics["year"].isin(crisis_years)].copy()
shock_index = crisis.groupby("country")["Deficit_to_GDP"].mean().reset_index(
    columns={"Deficit_to_GDP":"Shock_Response_Index"}
)

# Delta vs baseline: (crisis avg) - (avg of previous 3 years before each cris
def baseline_for_episode(df, episode_years, window=3):
    # baseline uses the years immediately before the episode starts
    start = min(episode_years)
    baseline_years = list(range(start-window, start))
    base = df[df["year"].isin(baseline_years)].groupby("country")["Deficit_tc
```

```
    epi = df[df["year"].isin(episode_years)].groupby("country")["Deficit_to_G
    return (epi - base).rename(f"delta_{start}")

delta_gfc = baseline_for_episode(df_metrics, [2008, 2009], window=3)
delta_covid = baseline_for_episode(df_metrics, [2020, 2021], window=3)

shock_delta = pd.concat([delta_gfc, delta_covid], axis=1).reset_index().renam

df_resilience = shock_index.merge(shock_delta, on="country", how="left")
display(df_resilience.head())
```

1 to 5 of 5 entries Filter

index	country	Shock_Response_Index	delta_2008	delta_2020
0	AT	-5.212604648690544	-1.287406047944506	-6.934216275447582
1	BE	-5.244824979551186	-2.4684332683947527	-5.931533731648292
2	BG	-2.7021497070026417	-2.9319969181801175	-5.731860494386316
3	CY	-3.0311741820203597	-2.4971576927301524	-3.511517236373537
4	CZ	-4.514005456510833	-1.7525929046064472	-6.17533265125353

Show 25 per page

7) Composite fiscal risk score

We combine three normalized components:

1. **Debt level** (avg debt-to-GDP)
2. **Deficit volatility** (std of deficit-to-GDP)
3. **Crisis sensitivity** (Shock Response Index)

Higher score → higher fiscal risk.

```
df_fiscal = country_summary.merge(df_resilience, on="country", how="left").dr

# Make crisis deficits increase risk (bigger deficit = higher risk)
df_fiscal["CrisisDeficitMagnitude"] = -df_fiscal["Shock_Response_Index"]

risk_cols = ["avg_debt_to_gdp", "deficit_volatility", "CrisisDeficitMagnitude

sc = MinMaxScaler()
norm = pd.DataFrame(
    sc.fit_transform(df_fiscal[risk_cols]),
    columns=risk_cols,
    index=df_fiscal.index
)

weights = {c: 1/3 for c in risk_cols}

# Final corrected score (use the original name everywhere)
df_fiscal["Fiscal_Risk_Score"] = (
    norm["avg_debt_to_gdp"] * weights["avg_debt_to_gdp"]
    + norm["deficit_volatility"] * weights["deficit_volatility"]
    + norm["CrisisDeficitMagnitude"] * weights["CrisisDeficitMagnitude"]
```

```
)

df_fiscal = df_fiscal.sort_values("Fiscal_Risk_Score", ascending=False)
from scipy.stats import spearmanr

def weight_sensitivity(df_fiscal, risk_cols, n=2000, seed=42):
    rng = np.random.default_rng(seed)
    base_rank = df_fiscal["Fiscal_Risk_Score"].rank(ascending=False).values

    X = df_fiscal[risk_cols].values
    sc = MinMaxScaler()
    Xn = sc.fit_transform(X)

    corrs = []
    worst = {"corr": 1.0, "weights": None}

    for _ in range(n):
        w = rng.random(len(risk_cols))
        w = w / w.sum()
        score = (Xn * w).sum(axis=1)
        rank = pd.Series(score).rank(ascending=False).values
        corr = spearmanr(base_rank, rank).correlation
        corrs.append(corr)
        if corr < worst["corr"]:
            worst = {"corr": corr, "weights": w}

    print("Rank robustness vs equal weights (Spearman):")
    print(f"  Mean: {np.mean(corrs):.3f}")
    print(f"  5th percentile: {np.percentile(corrs, 5):.3f}")
    print(f"  Worst: {worst['corr']:.3f} weights={dict(zip(risk_cols, worst['

risk_cols = ["avg_debt_to_gdp", "deficit_volatility", "CrisisDeficitMagnitude
weight_sensitivity(df_fiscal, risk_cols)
from sklearn.decomposition import PCA

def pca_risk_score(df_fiscal, risk_cols):
    X = df_fiscal[risk_cols].copy()
    sc = StandardScaler()
    Xs = sc.fit_transform(X)

    pca = PCA(n_components=1, random_state=42)
    pc1 = pca.fit_transform(Xs).ravel()
    pc1 = (pc1 - pc1.min()) / (pc1.max() - pc1.min())

    df_fiscal["Fiscal_Risk_PCA"] = pc1
    explained = pca.explained_variance_ratio_[0]
    print(f"PCA(1) explained variance: {explained:.2%}")

    corr = spearmanr(
        df_fiscal["Fiscal_Risk_Score"].rank(ascending=False),
        df_fiscal["Fiscal_Risk_PCA"].rank(ascending=False)
    ).correlation
    print(f"Rank corr (Equal-weight vs PCA): {corr:.3f}")

pca_risk_score(df_fiscal, risk_cols)
# Normalized drivers (0-1)
```

```
X = df_fiscal[risk_cols].copy()
sc = MinMaxScaler()
Xn = pd.DataFrame(sc.fit_transform(X), columns=risk_cols, index=df_fiscal.ind

for c in risk_cols:
    df_fiscal[f"norm_{c}"] = Xn[c]

weights = {c: 1/3 for c in risk_cols}
for c in risk_cols:
    df_fiscal[f"contrib_{c}"] = df_fiscal[f"norm_{c}"] * weights[c]

display(
    df_fiscal[["country", "Fiscal_Risk_Score"] +
             [f"contrib_{c}" for c in risk_cols] +
             risk_cols]
    .sort_values("Fiscal_Risk_Score", ascending=False)
    .head(15)
)
display(
    df_fiscal[
        ["country", "Fiscal_Risk_Score",
         "avg_debt_to_gdp", "deficit_volatility",
         "Shock_Response_Index", "CrisisDeficitMagnitude"]
    ].head(15)
)
```

```
Rank robustness vs equal weights (Spearman):
  Mean: 0.942
  5th percentile: 0.844
  Worst: 0.584 weights={'avg_debt_to_gdp': np.float64(0.0019795156082759147),
PCA(1) explained variance: 58.23%
Rank corr (Equal-weight vs PCA): 0.996
```

1 to 15 of 15 entries Filter

index	country	Fiscal_Risk_Score	contrib_avg_debt_to_gdp	contrib_deficit_volatility	c
8	EL	0.8233690250051628	0.3333333333333333	0.15670235833849627	
14	IE	0.67716931361089	0.11869573109894047	0.3333333333333333	
9	ES	0.5524134201165469	0.16418482896847833	0.12621846637390066	
15	IT	0.5243659432909465	0.2796548222676937	0.02784118558831753	
22	PT	0.47232990041866196	0.20801528130019709	0.07281304172829911	
1	BE	0.4478319885394544	0.23621991550737065	0.02907368826573773	
3	CY	0.4193996947089149	0.16271958865552466	0.135983186131281	
11	FR	0.4136464221138875	0.1835358207132417	0.009729416094294821	
19	MT	0.4040853871677616	0.11770142022747859	0.09087221896972558	
13	HU	0.37392182530133045	0.1445694419764342	0.03171648024087893	
0	AT	0.3570995226777335	0.1613916596902531	0.014069598719096688	
23	RO	0.35651177749937235	0.047202263386614485	0.054195525724128904	
12	HR	0.35291803760357604	0.110693881353376	0.0726188193724512	
26	SK	0.344206913750536	0.09026588468872482	0.07048333720128147	
25	SI	0.3232568629672634	0.09456130505053147	0.05579362395670885	

Show 25 per page

1 to 15 of 15 entries Filter

index	country	Fiscal_Risk_Score	avg_debt_to_gdp	deficit_volatility	Shock_Re
8	EL	0.8233690250051628	143.84640726619497	4.355497630717558	-10.642
14	IE	0.67716931361089	57.37563124405512	7.380691352973958	-6.769
9	ES	0.5524134201165469	75.70176586528748	3.8333939702246433	-8.089
15	IT	0.5243659432909465	122.2210142351545	2.1484667809029494	-6.473
22	PT	0.47232990041866196	93.35968408218146	2.918708675773693	-5.5656
1	BE	0.4478319885394544	104.72244895621103	2.1695760985778634	-5.244
3	CY	0.4193996947089149	75.11146637628339	4.000636261167574	-3.0311
11	FR	0.4136464221138875	83.49767513537151	1.838262914486688	-6.599
19	MT	0.4040853871677616	56.97505449623688	3.2280117842543876	-5.709
13	HU	0.37392182530133045	67.79934002042377	2.2148397206161863	-5.785
0	AT	0.3570995226777335	74.57648533412697	1.912598082655129	-5.212
23	RO	0.35651177749937235	28.573149436382245	2.599842788624114	-7.842
12	HR	0.35291803760357604	54.15193628143664	2.91538219098443	-4.781
26	SK	0.344206913750536	45.92213541811416	2.878807364522639	-5.277
25	SI	0.3232568629672634	47.65262592005849	2.627213733844177	-4.89

Show 25 per page

8) Europe choropleth map

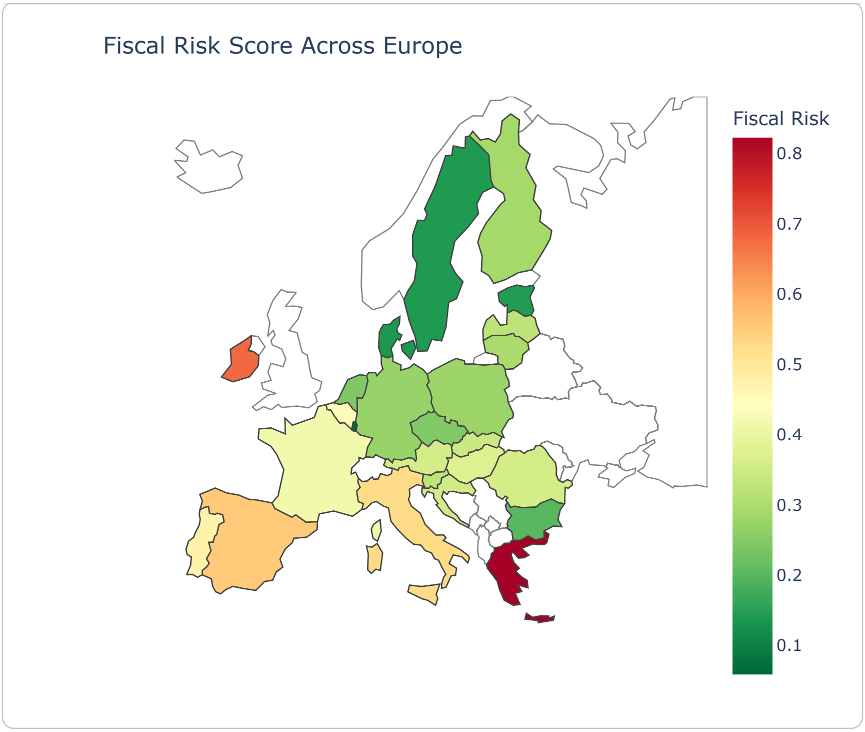
This cell uses a `choropleth_data.csv` with columns:

- `country_name`
- `Fiscal_Risk_Score`

If the CSV isn't present, we'll generate a best-effort one from ISO-2 country codes.

```
import os
import pandas as pd
import plotly.express as px

CHOROPLETH_CSV = "/content/drive/MyDrive/Colab Notebooks/docx fiscal/bubble.c

def build_country_name_map():
    """
    ISO2 -> Country Name map.
    Tries pycountry first. If not available, uses fallback.
    Also fixes Eurostat 'EL' (Greece) to match Greece.
    """
    # --- Always include Eurostat special codes ---
    special = {"EL": "Greece"}  # Eurostat uses EL for Greece

    try:
        import pycountry
        m = {c.alpha_2: c.name for c in pycountry.countries}
        m.update(special)
        # Optional: improve common naming differences
        m["CZ"] = "Czechia"
        m["GB"] = "United Kingdom"
        return m
    except Exception:
        fallback = {
            "AT":"Austria","BE":"Belgium","BG":"Bulgaria","HR":"Croatia","CY"
            "DK":"Denmark","EE":"Estonia","FI":"Finland","FR":"France","DE":"
            "HU":"Hungary","IE":"Ireland","IT":"Italy","LV":"Latvia","LT":"Li
            "MT":"Malta","NL":"Netherlands","PL":"Poland","PT":"Portugal","RO
            "SI":"Slovenia","ES":"Spain","SE":"Sweden",
            "IS":"Iceland","NO":"Norway","LI":"Liechtenstein","CH":"Switzerla
        }
        fallback.update(special)
        return fallback

name_map = build_country_name_map()

# 1) Prefer LIVE df after correction (recommended)
# If you really want CSV, ensure it was regenerated AFTER you fixed the score
if "df_fiscal" in globals() and "Fiscal_Risk_Score" in df_fiscal.columns:
    choropleth_data = df_fiscal[["country", "Fiscal_Risk_Score"]].copy()
elif os.path.exists(CHOROPLETH_CSV):
    choropleth_data = pd.read_csv(CHOROPLETH_CSV)
else:
    raise ValueError("No df_fiscal with Fiscal_Risk_Score found and CSV path

# 2) Ensure country_name exists
choropleth_data["country_name"] = choropleth_data["country"].map(name_map)

# 3) Clean score
choropleth_data["Fiscal_Risk_Score"] = pd.to_numeric(
```

```
    choropleth_data["Fiscal_Risk_Score"], errors="coerce"
)

choropleth_data = choropleth_data.dropna(subset=["country_name", "Fiscal_Risk

# (Optional) If your df still contains aggregates like EU27_2020 / EA19 etc,
# choropleth_data = choropleth_data[~choropleth_data["country"].isin(["EU27_2

# 4) Plot (Europe-only)
fig = px.choropleth(
    choropleth_data,
    locations="country_name",
    locationmode="country names",
    color="Fiscal_Risk_Score",
    scope="europe",
    color_continuous_scale="RdYlGn_r",
    title="Fiscal Risk Score Across Europe"
)

fig.update_traces(
    hovertemplate="<b>%{location}</b><br>Fiscal Risk Score: %{z:.3f}<extra></
)

fig.update_geos(
    fitbounds="locations",
    visible=False,
    showcountries=True,
    countrycolor="rgba(50,50,50,0.6)",
    showland=True,
    landcolor="white",
    showocean=False,
    showlakes=False
)

fig.update_layout(
    template="plotly_white",
    margin=dict(l=20, r=20, t=60, b=20),
    coloraxis_colorbar=dict(title="Fiscal Risk")
)

fig.show()
```



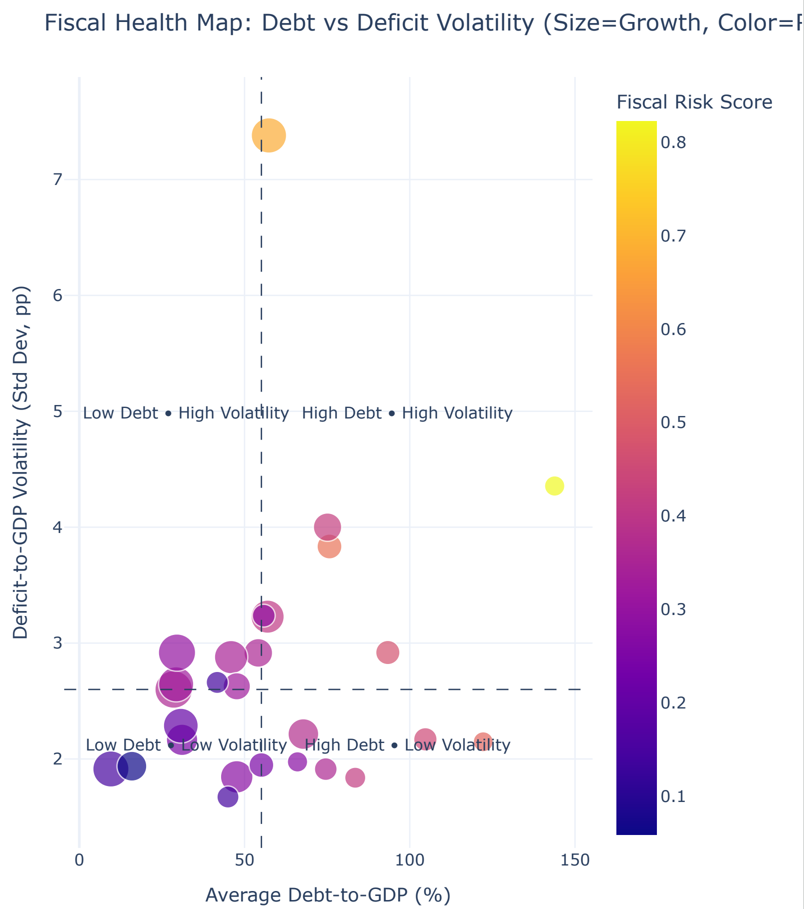
9) Executive bubble chart

X = Average Debt-to-GDP, Y = Deficit volatility, bubble size = Avg GDP growth, color = Fiscal risk score.

```
import numpy as np
import plotly.express as px

# --- Country full-name map ---
country_name_map = {
    "AT":"Austria","BE":"Belgium","BG":"Bulgaria","HR":"Croatia","CY":"Cyprus
    "DK":"Denmark","EE":"Estonia","FI":"Finland","FR":"France","DE":"Germany"
    "HU":"Hungary","IE":"Ireland","IT":"Italy","LV":"Latvia","LT":"Lithuania"
    "MT":"Malta","NL":"Netherlands","PL":"Poland","PT":"Portugal","RO":"Romar
    "SI":"Slovenia","ES":"Spain","SE":"Sweden",
    "IS":"Iceland","NO":"Norway","LI":"Liechtenstein","CH":"Switzerland","GB"
}

# --- Average GDP growth per country ---
tmp = df_metrics.sort_values(["country", "year"]).copy()
tmp["GDP_Growth"] = tmp.groupby("country")["GDP"].pct_change() * 100
avg_gdp_growth = (
    tmp.groupby("country")["GDP_Growth"]
    .mean()
```

```
    .reset_index()
    .rename(columns={"GDP_Growth": "avg_gdp_growth"})
)

# --- Build bubble df ---
bubble = df_fiscal.merge(avg_gdp_growth, on="country", how="left").dropna()

bubble["country_name"] = bubble["country"].map(country_name_map)
bubble = bubble.dropna(subset=["country_name"]).copy()

# --- Quadrant split points (medians) ---
x_split = bubble["avg_debt_to_gdp"].median()
y_split = bubble["deficit_volatility"].median()

# --- Plot ---
fig = px.scatter(
    bubble,
    x="avg_debt_to_gdp",
    y="deficit_volatility",
    size="avg_gdp_growth",
    color="Fiscal_Risk_Score",
    hover_name="country_name",
    color_continuous_scale=px.colors.sequential.Plasma,
    title="Fiscal Health Map: Debt vs Deficit Volatility (Size=Growth, Color=
    labels={
        "avg_debt_to_gdp": "Average Debt-to-GDP (%)",
        "deficit_volatility": "Deficit-to-GDP Volatility (Std Dev, pp)",
        "avg_gdp_growth": "Average GDP Growth (%)",
        "Fiscal_Risk_Score": "Fiscal Risk Score"
    },
    height=700
)

# --- Executive hover card (clean + consistent) ---
fig.update_traces(
    customdata=np.stack([
        bubble["country_name"],
        bubble["avg_debt_to_gdp"],
        bubble["deficit_volatility"],
        bubble["avg_gdp_growth"],
        bubble["Fiscal_Risk_Score"]
    ], axis=-1),
    hovertemplate=(
        "<b>%{customdata[0]}</b><br>"
        "Avg Debt-to-GDP: %{customdata[1]:.1f}%<br>"
        "Deficit Volatility: %{customdata[2]:.2f} pp<br>"
        "Avg GDP Growth: %{customdata[3]:.2f}%<br>"
        "Fiscal Risk Score: %{customdata[4]:.3f}"
        "<extra></extra>"
    )
)

# --- Quadrant lines ---
fig.add_vline(x=x_split, line_width=1, line_dash="dash")
fig.add_hline(y=y_split, line_width=1, line_dash="dash")
```

```
# --- Quadrant labels ---
x_min, x_max = bubble["avg_debt_to_gdp"].min(), bubble["avg_debt_to_gdp"].max
y_min, y_max = bubble["deficit_volatility"].min(), bubble["deficit_volatility

fig.add_annotation(x=(x_min + x_split)/2, y=(y_split + y_max)/2,
                   text="Low Debt • High Volatility", showarrow=False)
fig.add_annotation(x=(x_split + x_max)/2, y=(y_split + y_max)/2,
                   text="High Debt • High Volatility", showarrow=False)
fig.add_annotation(x=(x_min + x_split)/2, y=(y_min + y_split)/2,
                   text="Low Debt • Low Volatility", showarrow=False)
fig.add_annotation(x=(x_split + x_max)/2, y=(y_min + y_split)/2,
                   text="High Debt • Low Volatility", showarrow=False)

fig.update_layout(
    template="plotly_white",
    coloraxis_colorbar=dict(title="Fiscal Risk Score"),
    margin=dict(l=30, r=30, t=70, b=30)
)

fig.show()
```



10) Time-series forecasting (example country)

We keep forecasting as a **demo module** (not the core of the risk score). Use it to illustrate dynamics.

```
selected_country = "DE"  # change freely

# Build country time series from df_metrics/wide base
country_ts = wide[wide["country"] == selected_country].copy()
country_ts["year"] = pd.to_datetime(country_ts["year"], format="%Y")
country_ts = country_ts.set_index("year").sort_index().asfreq("YS-JAN")

country_ts["Debt_to_GDP"] = 100 * country_ts["Debt"] / country_ts["GDP"]
country_ts["Deficit_to_GDP"] = 100 * country_ts["Deficit"] / country_ts["GDP"
```

```
country_ts["GDP_Growth"] = country_ts["GDP"].pct_change() * 100

display(country_ts[["Debt_to_GDP","Deficit_to_GDP","GDP_Growth"]].dropna().he
```

1 to 5 of 5 entries Filter

year	Debt_to_GDP	Deficit_to_GDP	GDP_Growth
1996-01-01 00:00:00	56.6243671500304	-3.6384158801994744	-0.39567717555004656
1997-01-01 00:00:00	58.47030428399902	-3.0272269636252735	-0.8448951460889531
1998-01-01 00:00:00	59.84240584686296	-2.6495464722332356	2.5544657467915233
1999-01-01 00:00:00	60.349208565211526	-1.8664670428068013	3.4145665364235134
2000-01-01 00:00:00	59.23878459472404	-1.71431120460543	2.523540852284767

Show 25 per page

10.1 ARIMA forecast for Debt-to-GDP

```
import itertools
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error

# ----------------------------
# 1) Grid Search for Best ARIMA Order (by AIC)
# ----------------------------
def arima_grid_search(ts, p=range(0,4), d=range(0,2), q=range(0,4)):
    best = {"aic": np.inf, "order": None, "model": None}

    for order in itertools.product(p, d, q):
        try:
            model = ARIMA(ts, order=order).fit()
            if model.aic < best["aic"]:
                best = {"aic": model.aic, "order": order, "model": model}
        except:
            continue

    return best

ts = country_ts["Debt_to_GDP"].dropna()

train_size = int(len(ts) * 0.8)
train, test = ts.iloc[:train_size], ts.iloc[train_size:]

best = arima_grid_search(train)

print("Best ARIMA order:", best["order"])
print("Best AIC:", round(best["aic"], 2))

model = best["model"]

# ----------------------------
# 2) Forecast with Confidence Intervals
# ----------------------------
forecast_res = model.get_forecast(steps=len(test))
```

```
mean_forecast = forecast_res.predicted_mean
conf_int = forecast_res.conf_int()

# ----------------------------
# 3) Evaluate Performance
# ----------------------------
rmse = np.sqrt(mean_squared_error(test, mean_forecast))
print(f"Tuned ARIMA{best['order']} RMSE: {rmse:.3f}")

# ----------------------------
# 4) Plot with Uncertainty Bands
# ----------------------------
plt.figure(figsize=(10,4))

plt.plot(train.index, train, label="Train")
plt.plot(test.index, test, label="Test")
plt.plot(mean_forecast.index, mean_forecast, linestyle="--", label="Forecast"

plt.fill_between(
    conf_int.index,
    conf_int.iloc[:, 0],
    conf_int.iloc[:, 1],
    alpha=0.25,
    label="95% Confidence Interval"
)

plt.title(f"{selected_country} — Debt-to-GDP ARIMA Forecast (Tuned)")
plt.xlabel("Year")
plt.ylabel("Debt-to-GDP (%)")
plt.grid(True)
plt.legend()
plt.show()
def rolling_backtest_arima(ts, order, initial_train=0.7):
    n = len(ts)
    start = int(n * initial_train)

    preds, actuals, idxs = [], [], []

    for i in range(start, n):
        train = ts.iloc[:i]
        actual = ts.iloc[i]

        try:
            m = ARIMA(train, order=order).fit()
            fc = m.forecast(steps=1).iloc[0]

            preds.append(fc)
            actuals.append(actual)
            idxs.append(ts.index[i])
        except:
            continue

    pred_s = pd.Series(preds, index=idxs)
    act_s = pd.Series(actuals, index=idxs)

    rmse = np.sqrt(mean_squared_error(act_s, pred_s))
```

```
    return rmse

rolling_rmse = rolling_backtest_arima(ts, best["order"])
print(f"Rolling-origin RMSE: {rolling_rmse:.3f}")
```

```
/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py

Non-invertible starting MA parameters found. Using zeros as starting paramete

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py

Non-invertible starting MA parameters found. Using zeros as starting paramete

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py

Non-invertible starting MA parameters found. Using zeros as starting paramete

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py

Non-stationary starting autoregressive parameters found. Using zeros as star

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py

Non-stationary starting autoregressive parameters found. Using zeros as star

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py

Non-invertible starting MA parameters found. Using zeros as starting paramete

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py

Non-stationary starting autoregressive parameters found. Using zeros as star

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py

Non-invertible starting MA parameters found. Using zeros as starting paramete

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py
```

10.2 VAR model for joint dynamics (Debt-to-GDP, Deficit-to-GDP, GDP Growth)

```
from statsmodels.tsa.api import VAR
from sklearn.metrics import mean_squared_error

# --- Build VAR dataframe ---
var_df = country_ts[["Debt_to_GDP","Deficit_to_GDP","GDP_Growth"]].dropna()

# OPTIONAL but recommended: enforce stationarity by differencing
# Comment this out if you want levels, but differencing is more defendable
var_df_diff = var_df.diff().dropna()

# Train/test split
train_size = int(len(var_df_diff) * 0.8)
train_v, test_v = var_df_diff.iloc[:train_size], var_df_diff.iloc[train_size:

# Fit VAR
model = VAR(train_v)

maxlags = min(4, len(train_v)//3) if len(train_v) >= 9 else 1
order_results = model.select_order(maxlags=maxlags)

# ✅ Correct way to pick lag from AIC
lag = order_results.selected_orders.get("aic", 1)
lag = 1 if (lag is None or lag < 1) else lag
```