MILESTONE – 1

```
# ======================================================================
# MILESTONE 1: CRYPTOCURRENCY DATA PROCESSING WITH PARALLEL TASKS
# ======================================================================
# Objective (as per Milestone 1 PDF):
# 1. Choose at least two cryptocurrency CSV data files
# 2. Store the data in a database
# 3. Calculate metrics and store in a DataFrame
# 4. Apply parallel task concepts to the above steps
# ======================================================================

import pandas as pd
import numpy as np
import sqlite3
from concurrent.futures import ThreadPoolExecutor

# ======================================================================
# 1. PARALLEL DATA LOADING
# ======================================================================
def load_data(file_name, crypto_name):
    """
    Loads each CSV file, keeps only 'date' and 'close' columns,
    renames 'close' to the crypto name.
    """
    df = pd.read_csv(file_name)
    df['date'] = pd.to_datetime(df['date'])
    df = df[['date', 'close']]
    df.rename(columns={'close': crypto_name}, inplace=True)
```

```
    print(f"{crypto_name} data loaded successfully with {len(df)} records.")
    return df

# Dictionary containing all CSV file names
file_names = {
    "BTC": "Binance_BTCUSDT_d.csv",
    "ETH": "Binance_ETHUSDT_d.csv",
    "USDC": "Binance_USDCUSDT_d.csv"
}

print("Loading cryptocurrency data files in parallel...\n")

# Load all crypto data files in parallel using ThreadPoolExecutor
with ThreadPoolExecutor() as executor:
    results = list(executor.map(lambda item: load_data(item[1], item[0]), file_names.items()))

# Merge dataframes on 'date'
data = results[0]
for df in results[1:]:
    data = pd.merge(data, df, on='date', how='inner')

data.set_index('date', inplace=True)

print("\n=== Combined Cryptocurrency Data (First 10 Rows) ===")
print(data.head(10))
print("\nData Columns:", list(data.columns))

# ================================================================
```

```
# 2. STORING RAW DATA INTO DATABASE
# ======================================================================
print("\nStoring combined data into SQLite database...")

# Create SQLite database connection
conn = sqlite3.connect("crypto_data.db")

# Store merged crypto price data
data.to_sql("Crypto_Prices", conn, if_exists="replace", index=True, index_label="Date")

# ======================================================================
# 3. PARALLEL METRIC CALCULATION
# ======================================================================
def calculate_metrics(df, col_name):
    """
    Calculates key statistical metrics for each cryptocurrency:
    Mean, Standard Deviation, Max, Min, and Total Days.
    """
    series = df[col_name]
    metrics = {
        "Currency": col_name,
        "Mean Price": series.mean(),
        "Standard Deviation": series.std(),
        "Maximum Price": series.max(),
        "Minimum Price": series.min(),
        "Total Days": len(series)
    }
    print(f"Metrics calculated for {col_name}")
```

```
    return metrics

print("\nCalculating statistical metrics for each cryptocurrency in parallel...\n")

# Parallel calculation of metrics
with ThreadPoolExecutor() as executor:
    metrics = list(executor.map(lambda col: calculate_metrics(data, col), data.columns))

# Create a DataFrame for metrics
metrics_df = pd.DataFrame(metrics)

print("\n=== Cryptocurrency Metrics DataFrame ===")
print(metrics_df)

# ================================================================
# 4. STORE METRICS IN DATABASE
# ================================================================
print("\nStoring calculated metrics into the database...")

metrics_df.to_sql("Crypto_Metrics", conn, if_exists="replace", index=False)
print("Metrics stored successfully in 'crypto_data.db' (Table: Crypto_Metrics)")

# Close database connection
conn.close()

# ================================================================
# 5. SAVE METRICS TO CSV FILE
# ================================================================
```

```
metrics_df.to_csv("crypto_metrics.csv", index=False)
print("\nMetrics also saved locally to 'crypto_metrics.csv'")
```

OUTPUT:

```
Loading cryptocurrency data files in parallel...

BTC data loaded successfully with 365 records.
ETH data loaded successfully with 365 records.
USDC data loaded successfully with 365 records.

=== Combined Cryptocurrency Data (First 10 Rows) ===
                BTC       ETH    USDC
date
2024-09-25  29783.5726  1794.6118  0.9713
2024-09-26  32398.6906  1713.2078  1.0076
2024-09-27  29450.0596  1754.8628  1.0242
2024-09-28  32121.7937  1897.6015  0.9770
2024-09-29  32662.4059  1744.8421  0.9776
2024-09-30  29676.4571  1922.2990  1.0186
2024-10-01  30879.4024  1874.8512  1.0081
2024-10-02  30946.5220  1812.5524  0.9375
2024-10-03  30121.4021  1924.8931  1.0067
2024-10-04  28948.7407  1898.4636  1.0203

Data Columns: ['BTC', 'ETH', 'USDC']

Storing combined data into SQLite database...

Calculating statistical metrics for each cryptocurrency in parallel...
```

Metrics calculated for BTC

Metrics calculated for ETH

Metrics calculated for USDC

=== Cryptocurrency Metrics DataFrame ===

	Currency	Mean Price	Standard Deviation	Maximum Price	Minimum Price
0	BTC	30093.4803	1156.032383	32733.4210	27302.8802
1	ETH	1804.606035	72.850862	1973.3880	1630.2676
2	USDC	0.999938	0.038041	1.0956	0.9083

	Total Days
0	365
1	365
2	365

Storing calculated metrics into the database...

Metrics stored successfully in 'crypto_data.db' (Table: Crypto_Metrics)

Metrics also saved locally to 'crypto_metrics.csv'

Milestone 2

part 1

```
# portfolio_math.py
import numpy as np
import pandas as pd

# --- Example portfolio data (replace with your DB fetch later)
```

```
assets = ['BTC', 'ETH', 'XRP']
returns = np.array([0.12, 0.08, 0.05])   # expected returns per asset
weights = np.array([0.5, 0.3, 0.2])      # initial weights
cov_matrix = np.array([
    [0.04, 0.006, 0.004],
    [0.006, 0.03, 0.005],
    [0.004, 0.005, 0.02]
])  # covariance matrix

# --- Check weights sum to 1
if not np.isclose(weights.sum(), 1.0):
    raise ValueError("Weights must sum to 1")

# --- Portfolio return
portfolio_return = np.dot(weights, returns)

# --- Portfolio risk (std deviation)
portfolio_variance = np.dot(weights.T, np.dot(cov_matrix, weights))
portfolio_risk = np.sqrt(portfolio_variance)

# --- Put results into DataFrame
df = pd.DataFrame({
    'Asset': assets,
    'Weight': weights,
    'ExpectedReturn': returns
})

print("Portfolio Data:\n", df)
```

```
print("\nPortfolio Expected Return:", round(portfolio_return, 4))
print("Portfolio Risk (Std Dev):", round(portfolio_risk, 4))
```

output:

Portfolio Data:

	Asset	Weight	ExpectedReturn
0	BTC	0.5	0.12
1	ETH	0.3	0.08
2	XRP	0.2	0.05

Portfolio Expected Return**: 0.094**

Portfolio Risk (Std Dev)**: 0.1292**

part 2

```
# db_portfolio.py
import sqlite3
import pandas as pd

# Connect to SQLite DB (creates file if not exists)
conn = sqlite3.connect("portfolio.db")
cursor = conn.cursor()

# --- Create tables ---
cursor.execute("""
CREATE TABLE IF NOT EXISTS portfolio (
    portfolio_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    total_value REAL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
```

```
""")

cursor.execute("""
CREATE TABLE IF NOT EXISTS portfolio_assets (
    asset_id INTEGER PRIMARY KEY AUTOINCREMENT,
    portfolio_id INTEGER,
    currency TEXT,
    weight REAL,
    return REAL,
    risk REAL,
    metrics TEXT,
    FOREIGN KEY (portfolio_id) REFERENCES portfolio(portfolio_id)
)
""")

conn.commit()

# --- Insert sample data ---
cursor.execute("INSERT INTO portfolio (name,total_value) VALUES (?,?)",
               ("My Portfolio", 100000))
portfolio_id = cursor.lastrowid

assets_data = [
    (portfolio_id, 'BTC', 0.5, 0.12, 0.2, 'High Risk'),
    (portfolio_id, 'ETH', 0.3, 0.08, 0.15, 'Medium Risk'),
    (portfolio_id, 'XRP', 0.2, 0.05, 0.1, 'Low Risk')
]
cursor.executemany("""
```

```
INSERT INTO portfolio_assets (portfolio_id, currency, weight, return, risk, metrics)
VALUES (?,?,?,?,?,?)
""", assets_data)
conn.commit()

# --- Fetch Data ---
df_portfolio = pd.read_sql_query("SELECT * FROM portfolio", conn)
df_assets = pd.read_sql_query("SELECT * FROM portfolio_assets", conn)

print("Portfolio Table:\n", df_portfolio)
print("\nPortfolio Assets Table:\n", df_assets)

conn.close()
```

output:

Portfolio Table:

	portfolio_id	name	total_value	created_at
0	1	My Portfolio	100000.0	2025-09-18 06:24:04

Portfolio Assets Table:

	asset_id	portfolio_id	currency	weight	return	risk	metrics
0	1	1	BTC	0.5	0.12	0.20	High Risk
1	2	1	ETH	0.3	0.08	0.15	Medium Risk
2	3	1	XRP	0.2	0.05	0.10	Low Risk

part 3

```
# parallel_execution.py
from concurrent.futures import ThreadPoolExecutor
import time
```

```
# --- Example rule functions ---
def rule_equal_weight():
    time.sleep(1)  # simulate calculation
    return "Equal-weight rule executed"

def rule_risk_based():
    time.sleep(1)
    return "Risk-based rule executed"

def rule_performance_based():
    time.sleep(1)
    return "Performance-based rule executed"

rules = [rule_equal_weight, rule_risk_based, rule_performance_based]

# --- Run in parallel ---
results = []
with ThreadPoolExecutor(max_workers=3) as executor:
    futures = [executor.submit(rule) for rule in rules]
    for f in futures:
        results.append(f.result())

print("Parallel Execution Results:")
for r in results:
    print("-", r)
```

output:

Parallel Execution Results:

- Equal-weight rule executed
- Risk-based rule executed
- Performance-based rule executed

part 4

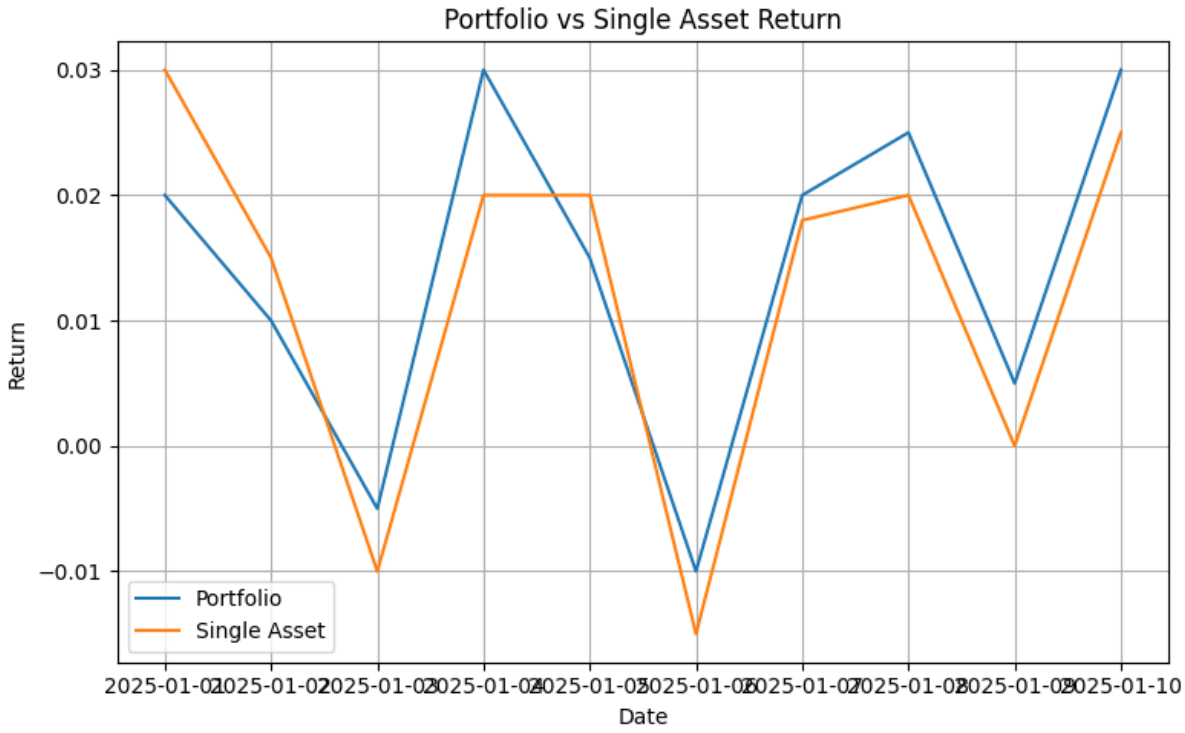
```
# compare_and_export.py
import pandas as pd
import matplotlib.pyplot as plt


# --- Sample Data ---
data = {
    'Date': pd.date_range(start='2025-01-01', periods=10, freq='D'),
    'PortfolioReturn': [0.02, 0.01, -0.005, 0.03, 0.015, -0.01, 0.02, 0.025, 0.005, 0.03],
    'SingleAssetReturn': [0.03, 0.015, -0.01, 0.02, 0.02, -0.015, 0.018, 0.02, 0.0, 0.025]
}
df = pd.DataFrame(data)


# --- Plot ---
plt.figure(figsize=(8,5))
plt.plot(df['Date'], df['PortfolioReturn'], label='Portfolio')
plt.plot(df['Date'], df['SingleAssetReturn'], label='Single Asset')
plt.xlabel('Date')
plt.ylabel('Return')
plt.title('Portfolio vs Single Asset Return')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
# --- Export to CSV ---
df.to_csv('portfolio_comparison.csv', index=False)
print("Data exported to portfolio_comparison.csv")
```

output:



MILESTONE – 3

part 1

```
import numpy as np
import pandas as pd
import sqlite3
import smtplib
from email.mime.text import MIMEText

# ----------------------------
# Email Setup
```

```
# ----------------------------
sender_email = "chanduchatopadyaya@gmail.com"
receiver_email = "chanduchatopadyaya@gmail.com"
# Replace with your Gmail App Password (16 characters)
sender_pass = "hbid vyir hblr tgjq"

# ----------------------------
# Simulated Portfolio Data
# ----------------------------
np.random.seed(42)
n_days = 252

assets = ["Asset_A", "Asset_B", "Asset_C"]
weights = np.array([0.35, 0.35, 0.30])
vols_daily = np.array([0.04, 0.04, 0.04]) / np.sqrt(252)
means_daily = np.array([0.08, 0.06, 0.10]) / 252

market_returns = np.random.normal(0.07/252, 0.15/np.sqrt(252), n_days)
returns = {a: np.random.normal(mu, sig, n_days) for a, mu, sig in zip(assets, means_daily, vols_daily)}
returns_df = pd.DataFrame(returns)
returns_df["Portfolio"] = returns_df.values.dot(weights)

# ----------------------------
# Risk Metrics
# ----------------------------
def annualized_volatility(daily_rets):
    return np.std(daily_rets, ddof=1) * np.sqrt(252)
```

```
def sharpe_ratio(daily_rets, rf=0.03):
    excess = daily_rets - (rf / 252)
    return np.mean(excess) / np.std(daily_rets, ddof=1) * np.sqrt(252)

def max_drawdown(daily_rets):
    cum = np.cumprod(1 + daily_rets)
    highwater = np.maximum.accumulate(cum)
    drawdowns = (cum - highwater) / highwater
    return -np.min(drawdowns)

def sortino_ratio(daily_rets, rf=0.03):
    excess = daily_rets - (rf / 252)
    downside = excess[excess < 0]
    if len(downside) == 0:
        return np.inf
    return np.mean(excess) / np.std(downside, ddof=1) * np.sqrt(252)

def beta(portfolio_rets, market_rets):
    cov = np.cov(portfolio_rets, market_rets)[0, 1]
    var = np.var(market_rets)
    return cov / var

def max_asset_weight(weights):
    return np.max(weights)

# ----------------------------
# Evaluate Rules
```

```
# ----------------------------
portfolio_rets = returns_df["Portfolio"]

rules = {
    "Volatility ≤ 5%": annualized_volatility(portfolio_rets) <= 0.05,
    "Sharpe Ratio ≥ 1": sharpe_ratio(portfolio_rets) >= 1,
    "Max Drawdown ≥ -20%": max_drawdown(portfolio_rets) >= -0.20,
    "Sortino Ratio ≥ 1": sortino_ratio(portfolio_rets) >= 1,
    "Beta ≤ 1.2": beta(portfolio_rets, market_returns) <= 1.2,
    "Max Asset Weight ≤ 40%": max_asset_weight(weights) <= 0.40,
}

# ----------------------------
# Store in Database
# ----------------------------
conn = sqlite3.connect("risk_results.db")
cursor = conn.cursor()
cursor.execute("""
CREATE TABLE IF NOT EXISTS risk_results (
    rule TEXT,
    passed BOOLEAN
)
""")
cursor.executemany(
    "INSERT INTO risk_results (rule, passed) VALUES (?, ?)",
    [(rule, passed) for rule, passed in rules.items()]
)
conn.commit()
```

```
conn.close()


# ----------------------------
# Print Results
# ----------------------------
print("\n--- Risk Rule Results ---")
for rule, passed in rules.items():
    print(f"{rule}: {'PASS' if passed else 'FAIL'}")


# ----------------------------
# Always Send Email
# ----------------------------
msg_body = "\n".join([f"{rule}: {'PASS' if passed else 'FAIL'}" for rule, passed in rules.items()])
msg = MIMEText(msg_body)
msg["Subject"] = "RISK ALERT: Portfolio Check"
msg["From"] = sender_email
msg["To"] = receiver_email


try:
    with smtplib.SMTP_SSL("smtp.gmail.com", 465) as server:
        server.login(sender_email, sender_pass)
        server.send_message(msg)
    print("\nEmail sent successfully ✅")
except smtplib.SMTPAuthenticationError:
    print("\nSMTP Authentication Error: Check your App Password and 2FA settings.")
except Exception as e:
    print(f"\nFailed to send email: {e}")
```

part 2

```
# predictor_xrp.py
# Advanced prediction module: Predict BTC, ETH, and XRP + Portfolio returns
# using Linear Regression. Also calculates daily returns (percentage change).

import pandas as pd
import numpy as np
import sqlite3
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from datetime import datetime
from pathlib import Path

def load_data_from_csvs():
    """Load Binance or Portfolio CSVs. If not found, generate synthetic demo data."""
    files = {
        "BTC": Path("Binance_BTCUSDT_d.csv"),
        "ETH": Path("Binance_ETHUSDT_d.csv"),
        "XRP": Path("Binance_XRPUSDT_d.csv"),   # changed from LTC → XRP
    }
    df = None
    found = False

    for k, p in files.items():
        if p.exists():
            tmp = pd.read_csv(p)
            cols_lower = [c.lower() for c in tmp.columns]
            if "close" in cols_lower:
```

```
            col = tmp.columns[cols_lower.index("close")]
            prices = pd.to_numeric(tmp[col], errors="coerce").fillna(method="ffill").fillna(0)
        else:
            numeric_cols = tmp.select_dtypes(include=[np.number]).columns
            if len(numeric_cols) > 0:
                prices = pd.to_numeric(tmp[numeric_cols[0]],
errors="coerce").fillna(method="ffill").fillna(0)
            else:
                continue
        series = prices.pct_change().fillna(0)
        colname = f"{k}_pct_change"
        if df is None:
            df = pd.DataFrame({colname: series})
        else:
            df[colname] = series
        found = True

    # Portfolio file check
    portfolio_paths = [
        Path("portfolio_vs_assests.csv"),
        Path("portfolio_vs_assests_15days_equal.csv"),
        Path("Portfolio_vs_assests_(Cumulative Return        ).csv")
    ]
    for p in portfolio_paths:
        if p.exists():
            tmp = pd.read_csv(p)
            candidates = [c for c in tmp.columns if "portfolio" in c.lower() and ("pct" in c.lower()
or "change" in c.lower())]
            if candidates:
```

```
            col = candidates[0]
            df["Portfolio_pct_change"] = pd.to_numeric(tmp[col], errors="coerce").fillna(0)
            found = True

    if not found or df is None:
        rng = np.random.RandomState(42)
        n = 2000
        df = pd.DataFrame({
            "BTC_pct_change": rng.normal(0, 1, size=n).cumsum(),
            "ETH_pct_change": rng.normal(0, 1.2, size=n).cumsum(),
            "XRP_pct_change": rng.normal(0, 1.1, size=n).cumsum(),  # XRP synthetic data
        })
        df["Portfolio_pct_change"] = (
            0.5 * df["BTC_pct_change"] +
            0.3 * df["ETH_pct_change"] +
            0.2 * df["XRP_pct_change"]
        )

    return df

def train_and_predict_series(series, label):
    y = np.asarray(series).astype(float)
    N = len(y)
    if N == 0:
        return None

    X_full = np.arange(N).reshape(-1, 1)
    model = LinearRegression()
```

```
    model.fit(X_full, y)
    y_pred_full = model.predict(X_full)

    mse = mean_squared_error(y, y_pred_full)
    r2 = r2_score(y, y_pred_full)

    last_n = min(10, N)
    actual_last = y[-last_n:]
    pred_last = y_pred_full[-last_n:]

    return {
        "label": label,
        "mse": float(mse),
        "r2": float(r2),
        "actual_last": actual_last.tolist(),
        "pred_last": pred_last.tolist(),
    }

def run_all_predictions(df):
    results = {}
    cols = [c for c in df.columns if c.lower().endswith("_pct_change") or "portfolio" in
c.lower()]
    if not cols:
        print("No suitable columns found.")
        return results

    for c in cols:
        res = train_and_predict_series(df[c].fillna(0), c)
```

```
        if res is None:
            continue
        results[c] = res
        print(f"--- {c} ---")
        print(f"MSE: {res['mse']:.4f} R²: {res['r2']:.4f}")
        display_df = pd.DataFrame({
            "Asset": [c]*len(res["actual_last"]),
            "Actual": np.round(res["actual_last"], 6),
            "Predicted": np.round(res["pred_last"], 6)
        })
        print(display_df.to_string(index=False))
        print()
    return results


def store_predictions(results, db_path="crypto.db"):
    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute("""CREATE TABLE IF NOT EXISTS predictions
                 (id INTEGER PRIMARY KEY AUTOINCREMENT, asset TEXT, mse REAL, r2 REAL, ts
TIMESTAMP)""")
    c.execute("""CREATE TABLE IF NOT EXISTS prediction_rows
                 (id INTEGER PRIMARY KEY AUTOINCREMENT, asset TEXT, actual REAL, predicted
REAL, ts TIMESTAMP)""")
    for asset, vals in results.items():
        c.execute("INSERT INTO predictions(asset, mse, r2, ts) VALUES (?, ?, ?, ?)",
                  (asset, vals["mse"], vals["r2"], datetime.utcnow().isoformat()))
        for a, p in zip(vals["actual_last"], vals["pred_last"]):
            c.execute("INSERT INTO prediction_rows(asset, actual, predicted, ts) VALUES (?, ?, ?,
?)",
```

```
                (asset, float(a), float(p), datetime.utcnow().isoformat()))
    conn.commit()
    conn.close()
    print("Predictions stored in DB.")

if _name_ == "_main_":
    print("Loading data...")
    df = load_data_from_csvs()

    # Show daily returns sample
    print("\n 🔹 Sample Daily Returns (first 5 rows):")
    print(df.head())

    print("\nRunning predictions...")
    results = run_all_predictions(df)

    if results:
        print("Storing predictions to DB...")
        store_predictions(results)
    print("Done.")
```

MILESTONE - 4

```
# rp_portfolio_from_csv.py
import os
import glob
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
import sqlite3
from math import sqrt


# ------------------------
# SETTINGS / PARAMETERS
# ------------------------
DATA_DIR = "/content"          # change if your CSVs are elsewhere
CSV_PATTERN = os.path.join(DATA_DIR, "Binance_*.csv")
OUT_DIR = DATA_DIR
MOM_WINDOW = 90  # not used here but left for reference
TRADING_DAYS = 365  # crypto daily


# ------------------------
# load CSV and extract date & close
# ------------------------
def load_price_series(path):
    df = pd.read_csv(path)
    # detect date column (common names)
    date_cols = [c for c in df.columns if c.lower() in ("date","time","timestamp","datetime")]
    if date_cols:
        date_col = date_cols[0]
        # try to parse milliseconds timestamp if numeric
        if pd.api.types.is_integer_dtype(df[date_col]) or
pd.api.types.is_float_dtype(df[date_col]):
            # assume unix ms if very large; fallback to default parse
            if df[date_col].max() > 1e10:
                df[date_col] = pd.to_datetime(df[date_col], unit='ms', errors='coerce')
            else:
```

```
            df[date_col] = pd.to_datetime(df[date_col], unit='s', errors='coerce')
        else:
            df[date_col] = pd.to_datetime(df[date_col], errors='coerce')
        df = df.rename(columns={date_col: "date"})
    else:
        # fallback to first column
        df = df.rename(columns={df.columns[0]: "date"})
        df["date"] = pd.to_datetime(df["date"], errors='coerce')

    # detect close column
    close_cols = [c for c in df.columns if c.lower() in
("close","close_price","price","last","closeusd")]
    if not close_cols:
        # fallback: first numeric column other than date
        for c in df.columns:
            if c != "date" and pd.api.types.is_numeric_dtype(df[c]):
                close_cols = [c]
                break
    if not close_cols:
        raise ValueError(f"No close/price column found in {path}. Columns:
{df.columns.tolist()}")

    close_col = close_cols[0]
    df = df[["date", close_col]].rename(columns={close_col: "close"}).dropna()
    df = df.sort_values("date").set_index("date")
    df["close"] = pd.to_numeric(df["close"], errors='coerce')
    df = df.dropna(subset=["close"])
    return df[["close"]]
```

```
# ------------------------
# Load all Binance CSVs
# ------------------------
files = glob.glob(CSV_PATTERN)
if not files:
    raise FileNotFoundError(f"No files found matching {CSV_PATTERN}")

series = {}
for f in files:
    # derive ticker name e.g. Binance_BTCUSDT_d.csv -> BTCUSDT
    base = os.path.splitext(os.path.basename(f))[0]
    ticker = base.replace("Binance_","").replace("_d","")
    try:
        s = load_price_series(f)
        series[ticker] = s
        print(f"Loaded {ticker} from {f}, {len(s)} rows")
    except Exception as e:
        print(f"Warning: skipped {f} -> {e}")

if not series:
    raise RuntimeError("No valid series loaded.")

# align by date (inner join to ensure consistent returns)
prices = pd.concat(series.values(), axis=1, join="inner")
prices.columns = list(series.keys())
prices = prices.sort_index()
if prices.empty:
```

```
    raise RuntimeError("Aligned price DataFrame is empty after inner join. Check date ranges.")

# ------------------------
# Compute returns & risk-parity weights
# ------------------------
# Use simple pct_change returns consistent with your friend's code
returns = prices.pct_change().dropna(how="all").dropna()  # drop early NA rows

# Calculate sample volatility (std of returns) and inverse-vol weights
vol = returns.std(ddof=1)               # daily volatility
inv_vol = 1.0 / vol.replace(0, np.nan)   # avoid divide-by-zero
weights_rp = inv_vol / inv_vol.sum()
weights_rp = weights_rp.fillna(0)

print("\n=== Risk-Parity Weights (from CSV data) ===")
print(weights_rp.round(6))

# ------------------------
# Historical portfolio returns using risk-parity weights
# ------------------------
historical_portfolio_returns = returns.dot(weights_rp)
historical_df = returns.copy()
historical_df['Portfolio_Return_RiskParity'] = historical_portfolio_returns

# Save historical returns CSV
hist_out = os.path.join(OUT_DIR, "historical_portfolio_returns_risk_parity.csv")
historical_df[['Portfolio_Return_RiskParity']].to_csv(hist_out, index=True)
```

```
print(f"\nSaved historical portfolio returns to '{hist_out}'")


# ------------------------
# Stress test scenarios (10 days each) - same approach as friend
# ------------------------
np.random.seed(42)

# Build per-asset random returns ensuring same columns order
assets = list(weights_rp.index)

stress_scenarios = {
    "Bull Market": pd.DataFrame({
        a: np.random.uniform(0.03, 0.06, 10) for a in assets
    }),
    "Bear Market": pd.DataFrame({
        a: np.random.uniform(-0.06, -0.03, 10) for a in assets
    }),
    "Volatile Market": pd.DataFrame({
        a: np.random.uniform(-0.10, 0.12, 10) for a in assets
    }),
}

# Apply weights (ensure columns match order)
stress_test_rp = {}
for scenario, df in stress_scenarios.items():
    df = df[assets]  # reorder if necessary
    stress_test_rp[scenario] = df.dot(weights_rp)
```

```
stress_test_rp_df = pd.DataFrame(stress_test_rp)
stress_test_rp_df.index = np.arange(1, len(stress_test_rp_df) + 1)  # Day 1..10

print("\n=== Stress Test Portfolio Returns (Risk-Parity) ===")
print(stress_test_rp_df.round(6))

# Save stress test CSV
stress_out = os.path.join(OUT_DIR, "stress_test_results_risk_parity.csv")
stress_test_rp_df.to_csv(stress_out, index_label="Day")
print(f"\nSaved stress-test results to '{stress_out}'")

# ------------------------
# Save to SQLite DB
# ------------------------
db_path = os.path.join(OUT_DIR, "crypto_portfolio.db")
conn = sqlite3.connect(db_path)
historical_df[['Portfolio_Return_RiskParity']].to_sql(
    "Portfolio_Returns_RiskParity", conn, if_exists="replace", index=True, index_label="Date"
)
stress_test_rp_df.to_sql(
    "StressTest_RiskParity", conn, if_exists="replace", index=True, index_label="Day"
)
conn.close()
print(f"\nSaved results to SQLite DB '{db_path}' (tables: Portfolio_Returns_RiskParity, StressTest_RiskParity)")

# ------------------------
# Plot Stress Test Results
```

```
# ------------------------

plt.figure(figsize=(9, 5))


# avoid using seaborn style implicitly — follow your friend's plotting but keep simple

plt.plot(stress_test_rp_df.index, stress_test_rp_df["Bull Market"], marker='o', linewidth=2.5, label='Bull Market')

plt.plot(stress_test_rp_df.index, stress_test_rp_df["Bear Market"], marker='s', linewidth=2.5, label='Bear Market')

plt.plot(stress_test_rp_df.index, stress_test_rp_df["Volatile Market"], marker='^', linewidth=2.5, label='Volatile Market')


plt.title("Risk-Parity Portfolio Returns (10-Day Random Stress Test)", fontsize=13, fontweight='bold')

plt.xlabel("Day", fontsize=11)

plt.ylabel("Portfolio Return", fontsize=11)

plt.xticks(stress_test_rp_df.index)

plt.grid(True, linestyle='--', alpha=0.6)

plt.legend(frameon=True, fontsize=5, loc='upper right')

plt.tight_layout()


# Save the figure

fig_out = os.path.join(OUT_DIR, "stress_test_rp_plot.png")

plt.savefig(fig_out, dpi=150)

print(f"\nSaved stress test plot to '{fig_out}'")

plt.show()
```

OUTPUT:

```
Loaded USDCUSDT from /content/Binance_USDCUSDT_d.csv, 365 rows

Loaded BTCUSDT from /content/Binance_BTCUSDT_d.csv, 365 rows

Loaded ETHUSDT from /content/Binance_ETHUSDT_d.csv, 365 rows
```

=== Risk-Parity Weights (from CSV data) ===

USDCUSDT 0.344253

BTCUSDT 0.327135

ETHUSDT 0.328612

dtype: float64

Saved historical portfolio returns to '/content/historical_portfolio_returns_risk_parity.csv'

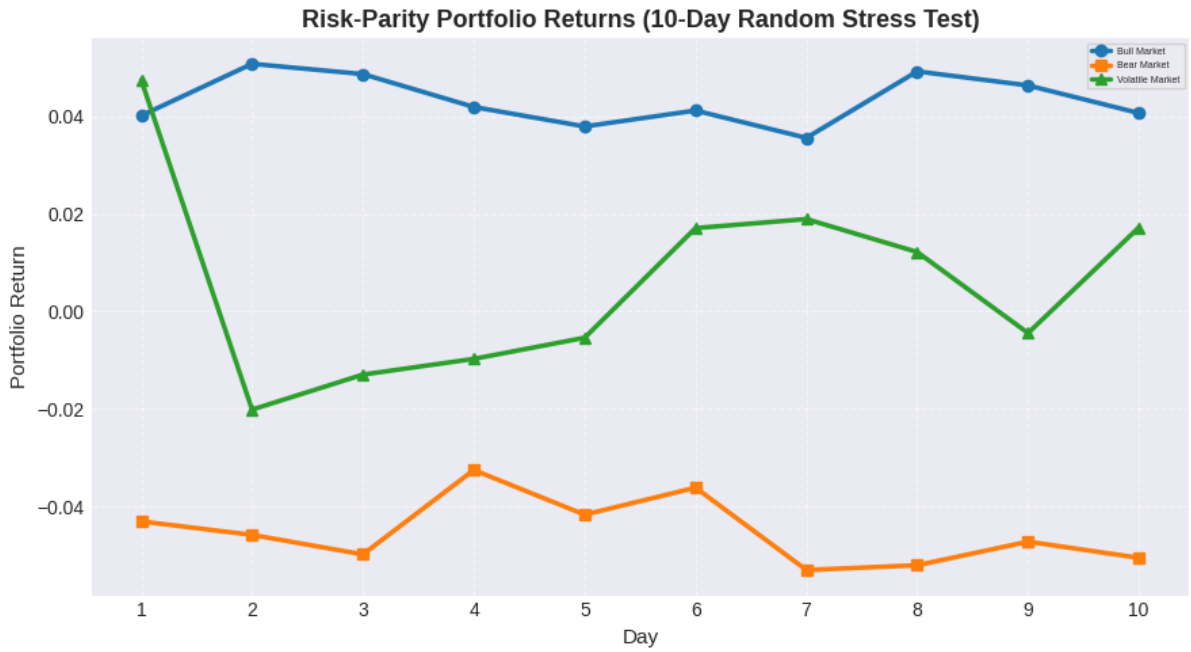
=== Stress Test Portfolio Returns (Risk-Parity) ===

	Bull Market	Bear Market	Volatile Market
1	0.040102	-0.042969	0.047413
2	0.050713	-0.045738	-0.020087
3	0.048609	-0.049729	-0.012916
4	0.041878	-0.032455	-0.009698
5	0.037892	-0.041593	-0.005369
6	0.041152	-0.036061	0.017077
7	0.035554	-0.052923	0.018928
8	0.049165	-0.051955	0.012176
9	0.046287	-0.047122	-0.004414
10	0.040629	-0.050433	0.017220

Saved stress-test results to '/content/stress_test_results_risk_parity.csv'

Saved results to SQLite DB '/content/crypto_portfolio.db' (tables: Portfolio_Returns_RiskParity, StressTest_RiskParity)

Saved stress test plot to '/content/stress_test_rp_plot.png'


Risk-Parity Portfolio Returns (10-Day Random Stress Test)
Bull Market
Bear Market
Volatile Market
Portfolio Return
0.04
0.02
0.00
−0.02
−0.04
1
2
3
4
5
6
7
8
9
10
Day