“We attest that this assignment has been completed in accordance with the academic integrity protocols and honor codes of the McCombs School of Business and the University of Texas. We relied on no unauthorized resources (including material submitted or produced by other students—past or present, individuals not listed below, internet message boards, solution manuals, or other forms of outside assistance). Any assistance from AI tools (e.g., ChatGPT) has been appropriately cited. This work product is the sole result of our efforts and is being submitted as such. **(Aileen Li - aql254 , Luke Hartfield - lgh872, Samay Mohapatra - sdm3888)**.”

Project II (Version B): Portfolio Optimization (IS and OOS Data)

INTRODUCTION

This project involves building a quantitative portfolio optimization tool using real-world stock return data sourced from the WRDS (Wharton Research Data Services) CRSP dataset. We will implement classic and advanced portfolio optimization techniques including mean-variance optimization (MVO) based on the sample covariance matrix and the Ledoit-Wolf covariance shrinkage estimator. Then we will apply these models to in-sample and out-of-sample datasets, compare portfolio metrics, and visualize results through efficient frontier plots.

METHODOLOGY

First, we ask the user for three things: the stock tickers, the dates for the in-sample period and the out-of-sample period, and the original portfolio weights. When accepting tickers from the user, we kept the rules clear. Tickers must be alphanumeric, can be lowercase, and we confirm that each ticker exists in the CRSP library. When obtaining dates, they must be in the form YYYY-MM. We turn each start into the first day of that month and each end into the last day of that month. We also ensure the out-of-sample period must begin after the in-sample period ends and that the out-of-sample dates cannot be beyond the latest CRSP date. Next, we accepted the weights in percentages only or ‘equal’ for equal weights. For weight constraints, we have that the number of weights must match the number of tickers, the weights must add to 100 percent, and they can be negative, which represents shorting a stock.

With these user inputs, we pulled the monthly data. We used crsp.msf for returns, price, and shares and used crsp.msenames so that tickers line up with the right dates. We matched on the name start and name end dates so that we picked the label that was active at that time. Additionally, we joined crsp.msedelist to bring in the delisting return and lined up the delisting month with the same calendar month in the monthly file.

Next, we cleaned the data. CRSP can carry a negative sign on price so we took the absolute value and converted shrout to individual counts by multiplying by one thousand. We filled missing delisting returns with zero since no delist means no extra jump. We kept missing regular returns so that we can easily identify them to remove afterwards. To get a more accurate return, we combined the two into one return called total return with the following formula:

(1 + return) * (1+ delisted return) - 1

If the regular return was missing, we dropped that row. This allowed us to calculate market cap without NaNs and in the correct units. Lastly, we ensured there was only one row per security per month.

We checked coverage in the in-sample window. For each ticker, we kept the ones with at least twenty four months of valid returns and dropped the ones without. That gives us enough history to estimate a stable covariance. With sufficient data, we pivoted the data into a matrix with months as rows and tickers as columns. This was our in-sample return matrix.

After constructing the in-sample return matrix, we built a corresponding one for the out-of-sample period. We recognize that ticker strings can change over time, so to keep the same securities we first looked up the permanent identifiers from CRSP called permnos for the in-sample names. We then pulled the out-of-sample data by those permnos. This kept the link even if the printed ticker changed. Then, we aligned the out-of-sample matrix so that it had the same columns and the same order as the in-sample matrix.

From the in-sample matrix we estimated the average monthly returns and covariances. This was done in two ways. First, we used the sample covariance. Second, we used a Ledoit-Wolf shrinkage estimate to reduce noise and stabilize the inverse. This is helpful for when time periods T over the number of stocks N is not large. These estimates were used for choosing the weights.

With the matrices set, we performed portfolio optimization. Using the user's input, we formed the following portfolios: the original portfolio, the global minimum variance portfolio, the efficient portfolio that matches the original expected return, and the optimal risky portfolio. To calculate these, we used the closed-forms formulas. In addition, we ran an additional optimal risky portfolio using quadratic programming from scipy, which also allows the user to specify lower and upper bounds for how large a weight can be as a parameter in the function.

Next, we took those weights we calculated and measured their out-of-sample performance on the out-of-sample matrix. We reported the average return, risk, and monthly/annual Sharpe in both windows. These metrics show how the portfolio weight choices made with in-sample data held up on the new data. We tested on multiple portfolios with different parameters including different date ranges, tickers, and initial weighting.

Finally, we packaged the results in a clear excel report. The report includes the in-sample efficient frontier with each portfolio marked on it. We added the realized out-of-sample points to the same chart for context. We saved tables for returns, risks, and Sharpe ratios, and we exported the cleaned in-sample and out-of-sample matrices, the weights, and the summary metrics to a single Excel file.

DISCUSSION OF FINDINGS

The results from our in-sample vs out-of-sample shows the sharpe ratios for in-sample being a lot higher than the sharpe ratios from out-of-sample. This shows overfitting which is expected because when training on past data, it will look good during the in-sample but perform poorly when tested on the out-of-sample. This is common with stocks specifically because the market is always changing, so an optimal portfolio over a past date range could change very quickly in the future based on macro-economic or industry trends.

When comparing the portfolios themselves, they behaved how we expected. The global minimum variance portfolio has a lower return and lower risk than any of the other portfolios. The efficient portfolio had the same return but lower risk, leading to a higher sharpe ratio. The ORP had the highest sharpe ratio which we expect because it is the optimal risky portfolio. Out-of-sample, a lot of the sharpe ratios collapsed due to our portfolios performing poorly. However, when looking at the Ledoit-Wolf shrinkage, the in-sample sharpe ratios look similar to the regular without shrinkage, but the out of sample are slightly better. This shows how shrinkage of the covariance of a portfolio can help reduce errors and helps with out-of-sample prediction.

When plugging and trying different portfolio weights and features, one thing that mattered a lot was the weight bounds the user can choose as a parameter in the QP ORP function. If the user wants to find a QP ORP using only positive weights (meaning no shorting allowed), the weights of the portfolio changes drastically from the regular ORP and can constrain it to pick up an extreme portfolio that does not optimize on the efficient frontier or perform well out of sample. However, if the user allows the QP ORP to be unbounded, it behaves like the regular ORP function and gives an almost identical portfolio with identical performance. Additionally, limiting weight sizes changes the portfolio drastically because it forces it to diversify weights across stocks, instead of putting large weights into stocks that perform well or help mitigate risk.

To display this, we ran the entire program to mock-up a portfolio. We used 10 stocks with equal weighting: DE, LMT, DIS, MCD, WMT, NKE, LUV, AMD, AMZN, and TSLA. We made the in-sample date range 2014-01 – 2020-12 and the out of sample date range of 2021-01 – 2023-12. Here is how the regular portfolios performed in-sample vs out-of-sample. As you can see, this highlights the overfitting effect that we talked about earlier and was consistent with multiple tests of different stock portfolios. This is shown by the performance of expected returns collapsing while standard deviation rises, leading to a collapsed sharpe ratio. As you can also see, the GMV has the lowest standard deviation and also expected return. The ORP's also have the highest expected return, standard deviation, and sharpe ratio. The efficient portfolio in-sample matches the return of the original portfolio, and also performs better than the original portfolio out-of-sample, highlighting the lower standard deviation with the same return in-sample.

	Expected Return	Standard Deviation	Sharpe Ratio (Monthly)
IS Orig Port	2.34%	5.21%	0.394
IS GMV	1.23%	3.60%	0.261
IS Efficient Portfolio	2.34%	4.79%	0.428
IS ORP	2.92%	6.02%	0.437
IS QP ORP	2.92%	6.02%	0.437

	Expected Return	Standard Deviation	Sharpe Ratio (Monthly)
OS Orig Port	0.67%	6.28%	0.060
OS GMV	0.73%	4.72%	0.094
OS Efficient Portfolio	1.19%	5.88%	0.153
OS ORP	1.43%	7.44%	0.153
OS QP ORP	1.43%	7.44%	0.153

As you can see below from the results of the Ledoit-Wolf shrinkage, it slightly improves our in-sample results in-sample, but does not really do too much better out-of-sample. In other example portfolios we tested, it varied and usually slightly improved the model, but still fell short out-of-sample due to the fact that it is hard to predict the future from historical returns.

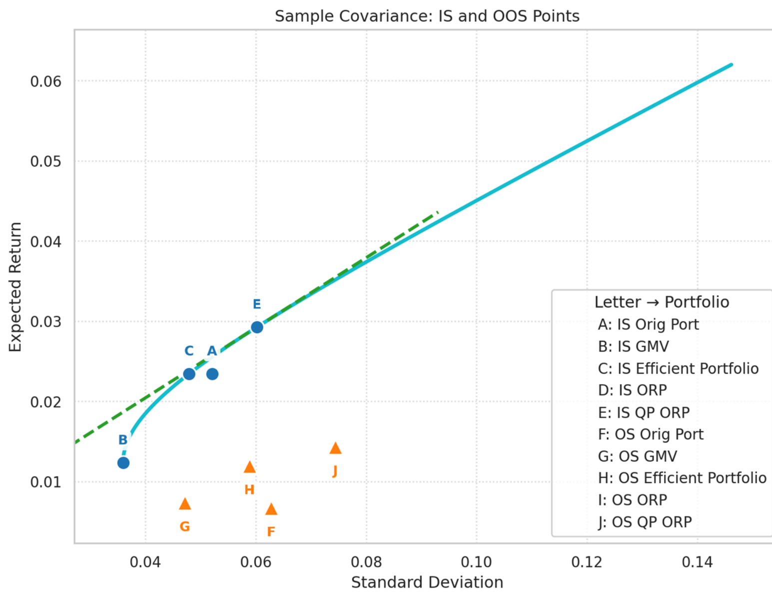
	Expected Return	Standard Deviation	Sharpe Ratio (Monthly)
IS LW Orig Port	2.34%	4.99%	0.411
IS LW GMV	1.32%	3.73%	0.276
IS LW Efficient	2.34%	4.72%	0.434
IS LW ORP	3.00%	6.05%	0.448
IS LW QP ORP	3.00%	6.05%	0.448

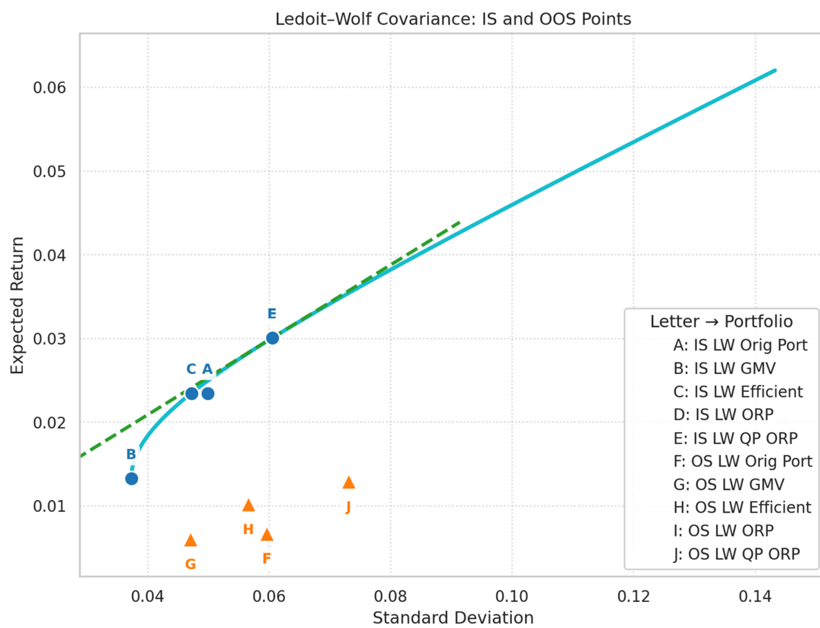
	Expected Return	Standard Deviation	Sharpe Ratio (Monthly)
OS LW Orig Port	0.67%	5.97%	0.063
OS LW GMV	0.60%	4.71%	0.066
OS LW Efficient	1.02%	5.66%	0.128
OS LW ORP	1.29%	7.31%	0.136
OS LW QP ORP	1.29%	7.31%	0.136

The Ledoit-Wolf shrinkage weights for each portfolio changed and are shown below. The weights of the Ledoit-Wolf portfolio change due to the covariance matrix shrinkage in the Ledoit-Wolf portfolio to get closer to the identity matrix, which is the average variance of the original portfolio. This shows that the dispersion of weights in the Ledoit-Wolf portfolio is less than the original portfolio.

Ticker	Δ GMV (LW - Sample)	Δ Efficient (LW - Sample)	Δ ORP (LW - Sample)	Δ QP ORP (LW - Sample)
DE	1.24%	0.84%	0.41%	0.41%
LMT	-0.65%	-0.96%	-1.75%	-1.75%
DIS	3.92%	8.87%	10.01%	10.01%
MCD	-6.80%	-5.12%	-5.23%	-5.24%
WMT	-2.88%	0.63%	1.08%	1.08%
NKE	-0.55%	-6.70%	-9.14%	-9.14%
LUV	2.44%	1.86%	1.64%	1.64%
AMD	0.36%	0.27%	1.66%	1.66%
AMZN	2.34%	0.07%	-0.02%	-0.02%
TSLA	0.59%	0.25%	1.35%	1.35%

VISUALIZATIONS





Looking at the two efficient frontier charts, you can really see the difference between in-sample and out-of-sample results. When you use the weights that worked best in-sample and then apply them to new, out-of-sample returns, the expected return usually drops and the risk goes up. This is because those weights were chosen based on past data, and the future is never exactly the same as the past. This is a good reminder that portfolio optimization can look great in backtests, but may not work as well on new data because of what's called estimation risk. The Ledoit-Wolf (LW) covariance method is supposed to make the results less sensitive to changes in the data, since it uses a shrinkage approach to smooth out the estimates. In practice, though, even this method can't fully get rid of estimation risk, so the improvement is there but not huge. Both frontiers look similar, but the numbers are a bit different because they use different ways to measure risk (covmat vs lwcovmat). If you want out-of-sample results to look more like in-sample, you'd have to keep updating your weights as new data comes in. But doing that too often can mean more trading and higher costs, which can eat into your returns. So, there's always a tradeoff between chasing the best fit for the past and keeping things simple and stable for the future.

LIMITATIONS

One big limitation can come from user inputs. Small errors in user inputs can swing the original weights a lot, which makes returns jump around from run to run. If expected returns are a bit off or if the target return is set wrong, the weights can push to extremes and the final result can look random, so realized performance can end up much higher or lower than planned. When the user picks a tiny list of assets, the math can look good on paper but it is not realistic, and the live returns can be skewed high or low because there is not enough diversification to smooth luck from one or two names. Real markets add noise, costs, and shocks that the simple model does not catch, so small input mistakes grow into big outcome changes.

Robust portfolio methods try to reduce this swing by softening the inputs and spreading bets. Shrinkage of risk (like Ledoit-Wolf), blended views for returns, and limits on position size can make weights more stable and cut the chance of big surprises in realized returns, though they may give up some upside on paper. A practical tool should warn on small portfolios, cap turnover, and show how outputs change when inputs move a little, so users see the range they might get in real life. This keeps results closer to what is possible day to day and can help build trust between managers and clients even when markets are noisy.

CONCLUSION

Overall, the portfolios look amazing in-sample, but underperform greatly when tested out-of-sample. Although portfolio optimization may look promising in-sample, it often does not hold up out-of-sample. Ledoit-Wolf shrinkage helps make it more robust, but shrinkage cannot erase the fundamental noise and issues with constructing expected returns from a historical period. It does improve stability of optimized portfolio and reduce sensitivity to estimation error. The constraints when constructing these portfolios matter greatly and affect performance. Allowing or forbidding shorts fundamentally changes the ORP by changing the weight allocations.

The biggest key take away is that in-sample optimization is not reliable and shouldn't be trusted. It very clearly overfits the data in-sample and performs poorly out-of-sample. Shrinkage is useful for making safer estimates, but it should not be expected to produce better returns out of sample. Going forward, rebalancing to refit allocation more frequently like evaluating rolling out-of-sample instead of one big period can improve estimates. Additionally, more regularization/factors can be included to help take past portfolio returns and optimize a portfolio that better generalizes to the future.

APPENDIX

Exhibit A-1 — Chat Session 1
Tool / Model: ChatGPT5
Date of usage: 9/24
Purpose: Code generation
Prompt: Create code to ask the user to input a list of stock tickers, then turn that into a list. Include validation to ensure that they entered in proper tickers
Response: import re

```
def get_tickers():
    pattern = re.compile(r"^[A-Z0-9]+$")   # basic validation: alphanumeric only
    while True:
        raw = input("Enter stock tickers separated by commas (e.g., AAPL, MSFT, TSLA): ")
        # Step 1: split, strip spaces, force uppercase
        tickers = [t.strip().upper() for t in raw.split(",") if t.strip()]

        # Step 2: validate
        invalid = [t for t in tickers if not pattern.match(t)]
        if not tickers:
            print("✖ You must enter at least one ticker.")
        elif invalid:
            print(f"✖ Invalid ticker(s): {', '.join(invalid)}. Try again.")
        else:
            return tickers
# Example usage
tickers = get_tickers()
print("✔ Final ticker list:", tickers)
```

Exhibit A-1 — Chat Session 2
Tool / Model: ChatGPT5
Date of usage: 9/24
Purpose: Debugging
Prompt: Help me do some of the heavy loading on transforming this to calculate a portfolio mean returns and covariance matrix given a list of tickers, weights, and in sample start and end date. Use validation to make sure the ticker is valid and has returns for that month. Ignore the price weight or value weight, but give the user the option to choose equal weight.: # Steps: # - Set base = 100 for all EW VW PW # - Loop across t-1, t month pairs # - Carry forward rule # - Calculate EW VW PW # - Compound # - return tidy output def build_topn_indexes(panel, top_n): # Organize panel = panel.sort_values(['date', 'permno']) months = sorted(panel['date'].unique()) # A dict containing groups for each date in panel data frame by_month = {d: g for d, g in panel.groupby('date')} ew_level, vw_level, pw_level = [100.0], [100.0], [100.0] # Look from 2nd month for t and t-1; tqdm for progress bar for i in tqdm(range(1, len(months)), desc = "Index Calculation Progress"): t_1, t = months[i-1], months[i] # Stands for group at time t-1, group at time t g_t1 = by_month[t_1].dropna(subset = ['mktcap', 'prc']) g_t = by_month[t] # If there is no data for group at t-1 or at 1, then carry forward last value # ChatGPT -- said this is to keep time series continuous, prevent look-ahead fixes, # prevent NaN cascades, keeps reconstitution pattern intact

(point for each month), # and enables fair comparisions if g_t1.empty or g_t.empty: ew_level.append(ew_level[-1]) vw_level.append(vw_level[-1]) pw_level.append(pw_level[-1]) continue # Top n by mkt cap at t-1 (investable & no look ahead) chosen = g_t1.sort_values('mktcap', ascending = False).head(top_n).set_index('permno') # Check that we have returns at t for the selected/chosen permnos g_t = g_t.set_index('permno') # Stocks(?) in top_n that also exist at t and t_1, hence "common" common = chosen.index.intersection(g_t.index) if len(common) == 0: ew_level.append(ew_level[-1]) vw_level.append(vw_level[-1]) pw_level.append(pw_level[-1]) continue r_t = g_t.loc[common, 'eff_ret'].astype(float) # EW: avg of gross returns over available names r_t_ew = r_t.dropna() # available ew_gross = (1.0 + r_t_ew).mean() if not r_t_ew.empty else 1.0 # avg # VW: weights from t-1 mkt cap # Locate the mktcapc for selected (top_n) that also exist at t (common) w_vw = chosen.loc[common, 'mktcap'].astype(float) # Boolean mask indicating which permnos have effect returns mask_vw = r_t.notna() w_vw = w_vw.loc[mask_vw] # drops names without a t return # Align the t returns to the exact same subset the weights are kept r_vw = r_t.loc[mask_vw] if w_vw.empty or w_vw.sum() <= 0: vw_gross = 1.0 else: w_vw = w_vw / w_vw.sum() vw_gross = ((1.0 + r_vw) * w_vw).sum() # PW: weights from t-1 (abs) price w_pw = chosen.loc[common, 'prc'].abs().astype(float) mask_pw = r_t.notna() w_pw = w_pw.loc[mask_pw] r_pw = r_t.loc[mask_pw] if w_pw.empty or w_pw.sum() <= 0: pw_gross = 1.0 else: w_pw = w_pw / w_pw.sum() pw_gross = ((1.0 + r_pw) * w_pw).sum() # Compound index levels ew_level.append(ew_level[-1] * float(ew_gross)) vw_level.append(vw_level[-1] * float(vw_gross)) pw_level.append(pw_level[-1] * float(pw_gross)) # Convert to tidy data frame index by month t (growth from t-1 to t) idx_months = pd.to_datetime(months[1:]) ew = pd.Series(ew_level[1:], index = idx_months, name = 'Equal-Weighted') vw = pd.Series(vw_level[1:], index = idx_months, name = 'Value-Weighted') pw = pd.Series(pw_level[1:], index = idx_months, name = 'Price-Weighted') return pd.concat([ew, vw, pw], axis=1) idx_df = build_topn_indexes(panel, 100)

Response: import warnings

```
from dataclasses import dataclass
from typing import Iterable, Optional, Dict, List, Tuple

import numpy as np
import pandas as pd

@dataclass(frozen=True)
class PortfolioInputs:
    tickers: Iterable[str]
    start: str  # 'YYYY-MM'
    end: str    # 'YYYY-MM'
    weights: Optional[Iterable[float]] = None  # ignored if equal_weight=True
    equal_weight: bool = False                 # if True, uses 1/N

@dataclass
class PortfolioResult:
    used_tickers: List[str]
    dropped_tickers: Dict[str, str]
    asset_returns: pd.DataFrame      # monthly wide matrix, index MonthEnd, columns tickers
    portfolio_returns: pd.Series     # monthly series
```

```
    mean_vector: pd.Series           # per-asset mean (matching asset_returns columns)
    cov_matrix: pd.DataFrame         # per-asset covariance matrix (sample)

def _monthly_grid(start_ym: str, end_ym: str) -> pd.DatetimeIndex:
    start = pd.Period(start_ym, freq="M").to_timestamp("M")
    end = pd.Period(end_ym, freq="M").to_timestamp("M")
    return pd.date_range(start, end, freq="M")

def _prepare_price_panel(df_prices: pd.DataFrame) -> pd.DataFrame:
    """
    Expect columns: ['date','ticker','price']
    - 'date' can be daily; we will convert to month-end and last price-of-month.
    """
    req = {"date", "ticker", "price"}
    missing = req - set(df_prices.columns)
    if missing:
        raise ValueError(f"df_prices missing required columns: {sorted(missing)}")

    out = df_prices.copy()
    out["date"] = pd.to_datetime(out["date"])
    out["ticker"] = out["ticker"].astype(str).str.upper()

    # month-end price per ticker
    out["month"] = out["date"].dt.to_period("M").dt.to_timestamp("M")
    out = (
        out.sort_values(["ticker", "date"])
          .groupby(["ticker", "month"], as_index=False)
          .last()[["ticker", "month", "price"]]
          .rename(columns={"month": "date"})
    )
    return out

def _prices_to_monthly_returns(monthly_prices: pd.DataFrame) -> pd.DataFrame:
    """
    Input: monthly price panel (one row per ticker-month)
    Output: wide matrix of simple returns, rows=MonthEnd, cols=tickers
    """
    wide = (
        monthly_prices.pivot(index="date", columns="ticker", values="price")
                  .sort_index()
    )
    rets = wide.pct_change().iloc[1:]  # drop first month per series due to diff
    rets.index.name = "date"
    return rets
```

```
def _validate_and_trim(rets: pd.DataFrame,
                       tickers: Iterable[str],
                       grid: pd.DatetimeIndex) -> Tuple[pd.DataFrame, List[str], Dict[str, str]]:
    tickers = [t.upper() for t in tickers]
    dropped: Dict[str, str] = {}

    # Ticker existence
    missing = [t for t in tickers if t not in rets.columns]
    for t in missing:
        dropped[t] = "No price/return history found."
    keep = [t for t in tickers if t not in missing]

    # Align to in-sample grid and enforce monthly coverage
    rets_is = rets.reindex(grid).copy()
# Drop tickers lacking *any* return in the in-sample window
    for t in list(keep):
        col = rets_is[t]
        if col.isna().all():
            dropped[t] = "No returns in the in-sample window."
            keep.remove(t)

    # Optionally enforce full coverage per month; here we require each used ticker
    # to have a non-null return for every month (strict). If you prefer partial,
    # change the mask to allow NaNs and use weights on available names only.
    strict = True
    if strict:
        for t in list(keep):
            if rets_is[t].isna().any():
                dropped[t] = "Missing one or more months in the in-sample window."
                keep.remove(t)

    used = keep
    if not used:
        raise ValueError("No valid tickers with complete in-sample coverage.")

    return rets_is[used], used, dropped

def _resolve_weights(used_tickers: List[str],
                     weights: Optional[Iterable[float]],
                     equal_weight: bool) -> pd.Series:
    n = len(used_tickers)
    if equal_weight or weights is None:
        w = np.repeat(1.0 / n, n)
```

```
        return pd.Series(w, index=used_tickers)

    weights = np.asarray(list(weights), dtype=float)
    if weights.shape[0] != n:
        raise ValueError(f"weights length {weights.shape[0]} != number of valid tickers {n}")
    if np.allclose(weights.sum(), 0.0):
        raise ValueError("Weights sum to zero.")
    if np.any(np.isnan(weights)):
        raise ValueError("Weights contain NaN.")
    # Normalize to sum to 1
    weights = weights / weights.sum()
    return pd.Series(weights, index=used_tickers)

def build_portfolio_stats(df_prices: pd.DataFrame, inputs: PortfolioInputs) -> PortfolioResult:
    """
    Parameters
    ----------
    df_prices : DataFrame with columns ['date','ticker','price'] (daily or monthly)
    inputs    : PortfolioInputs(tickers, start='YYYY-MM', end='YYYY-MM', weights, equal_weight)

    Returns
    -------
    PortfolioResult containing:
      - used_tickers, dropped_tickers
      - asset_returns (monthly simple returns, wide)
      - portfolio_returns (series)
      - mean_vector (per-asset mean returns over in-sample)
      - cov_matrix (per-asset covariance matrix over in-sample)
    """
    # 1) Normalize to monthly price panel
    monthly_prices = _prepare_price_panel(df_prices)

    # 2) Convert to monthly returns matrix
    asset_rets_all = _prices_to_mon_
```

Exhibit A-1 — Chat Session 3
Tool / Model: ChatGPT5
Date of usage: 9/24
Purpose: Code generation
Prompt: I want to go function by function. So if you were given monthly returns for 60 months, get the average return for each stock. Then create a separate function creating the demeaned matrix returns
Response: import pandas as pd

```
def mean_returns_monthly(rets: pd.DataFrame) -> pd.Series:
```

```
    """
    Input:  rets = DataFrame of monthly simple returns (60 rows x N tickers).
    Output: Series of average monthly return per ticker (NaNs ignored).
    """
    # If there are non-numeric cols, coerce them (optional; remove if not needed)
    rets_num = rets.apply(pd.to_numeric, errors="coerce")
    return rets_num.mean(axis=0)  # monthly mean per column


def demean_returns(rets: pd.DataFrame, means: pd.Series | None = None) -> pd.DataFrame:
    """
    Input:  rets  = DataFrame of monthly returns (same shape as above)
            means = (optional) Series of per-ticker means; if None, compute from rets
    Output: DataFrame where each column has its mean subtracted (demeaned).
    """
    rets_num = rets.apply(pd.to_numeric, errors="coerce")
    if means is None:
        means = rets_num.mean(axis=0)
    # Broadcast subtract per column; NaNs stay NaN
    return rets_num.subtract(means, axis=1)
```

Exhibit A-4 — Chat Session 4
Tool / Model: ChatGPT5
Date of usage: 9/24
Purpose: Code generation
Prompt: I now want to use my means, and ret_matrix to calculate a covariance matrix. Here is the formulas we use in excel: =MMULT(TRANSPOSE(EMAT),EMAT)/(COUNT(BJ)-1)
Response: import pandas as pd

```
import numpy as np

def cov_from_demeaned(E: pd.DataFrame) -> pd.DataFrame:
    """
    E: DataFrame of demeaned returns (rows = time, cols = tickers), no NaNs.
    Returns the sample covariance matrix using (E' E) / (n - 1).
    """
    # ensure complete cases (Excel usually assumes same n per column here)
    E = E.dropna(axis=0, how="any")
    n = E.shape[0]
    cov_np = (E.T @ E).to_numpy() / (n - 1)
    return pd.DataFrame(cov_np, index=E.columns, columns=E.columns)
```

Exhibit A-1 — Chat Session 5
Tool / Model: ChatGPT5
Date of usage: 9/24

Purpose: Code generation
Prompt: Create a function to calculate the global minimum variance portfolio as follows. Use the numerator: =MMULT(INVCOVMAT,ONEVEC)/ the denominator: =MMULT(MMULT(TRANSPOSE(ONEVEC),INVCOVMAT),ONEVEC)
Response: import numpy as np
import pandas as pd

```
def gmv_weights_from_inv(inv_cov: pd.DataFrame) -> pd.Series:
    """
    Global Minimum Variance (GMV) weights given INVCOVMAT.
    Numerator:   INVCOVMAT * ONEVEC
    Denominator: 1' * INVCOVMAT * 1
    """
    Ainv = inv_cov.to_numpy(dtype=float)
    n = Ainv.shape[0]
    one = np.ones((n, 1))
    num = Ainv @ one
    den = float(one.T @ Ainv @ one)
    w = (num / den).ravel()
    return pd.Series(w, index=inv_cov.index)  # sums to ~1

def gmv_weights_from_cov(cov: pd.DataFrame) -> pd.Series:
    """
    Same as above but starts from COVMAT (computes inverse inside).
    """
    A = cov.to_numpy(dtype=float)
    n = A.shape[0]
    one = np.ones((n, 1))
    Ainv = np.linalg.inv(A)             # use np.linalg.pinv(A) if singular
    num = Ainv @ one
    den = float(one.T @ Ainv @ one)
    w = (num / den).ravel()
    return pd.Series(w, index=cov.index)
```

Exhibit A-1 — Chat Session 6
Tool / Model: ChatGPT5
Date of usage: 9/25
Purpose: Debugging
Prompt: Using the closed form solution I gave you earlier, help me write an excel function that calculates the efficient portfolio.
Response: import numpy as np
import pandas as pd

```
def efficient_weights_from_inv(invcovmat: pd.DataFrame, means: pd.Series, mu0: float) -> pd.Series:
```

```
    """
    Unconstrained efficient-portfolio weights for target return mu0.
    Uses: w = P[(c - b*mu0)/d * 1 + (a*mu0 - b)/d * mu],
    where P = invcovmat, a=1'P1, b=1'Pmu, c=mu'Pmu, d=ac-b^2.
    """
    P   = invcovmat.to_numpy(dtype=float)
    mu  = means.reindex(invcovmat.index).to_numpy(dtype=float).reshape(-1, 1)
    one = np.ones((P.shape[0], 1))

    a = float(one.T @ P @ one)
    b = float(one.T @ P @ mu)
    c = float(mu.T  @ P @ mu)
    d = a * c - b * b

    w = ((c - b*mu0)/d) * (P @ one) + ((a*mu0 - b)/d) * (P @ mu)
    w = w.ravel()
    w = w / w.sum()  # tidy sum-to-one

    return pd.Series(w, index=invcovmat.index, name="Eff_Weight")
```

Exhibit A-1 — Chat Session 7
Tool / Model: ChatGPT5
Date of usage: 9/25
Purpose: Code generation
Prompt: Function to create delta squared: =SUM((covmat-ident)^2)
Response: import numpy as np

```
import pandas as pd

def delta_squared(covmat: pd.DataFrame, ident: pd.DataFrame) -> float:
    """
    Returns SUM((covmat - ident)^2) with index/column alignment.
    """
    # align to common order (and error if mismatch)
    cov = covmat.copy()
    idt = ident.reindex(index=cov.index, columns=cov.columns)
    if idt.isnull().values.any():
        raise ValueError("ident must have the same index/columns as covmat.")

    diff = cov.to_numpy(dtype=float) - idt.to_numpy(dtype=float)
    return float(np.sum(diff * diff))  # Frobenius norm squared
```

Exhibit A-2 — Chat Session 1
Tool / Model: ChatGPT5
Date of usage: 9/28

Purpose: Debugging
Prompt: we only accept weights in percent form so 10 is 10%, we also want to be able to accept 1.2 for 1.2% and be able to return error for weight constraint
Response: Got it. Here's a minimal weights input that accepts percent-only entries, supports decimals like 1.2 for 1.2%, and enforces constraints (no shorting, per-asset bounds, and total = 100%). It returns weights as decimals (sum = 1) for the rest of your pipeline.

```
# === Portfolio weights (percent-only) ===
# Section: User input for portfolio weights with constraints

import re
import pandas as pd

def get_portfolio_weights_percent(
    tickers,
    prompt="Enter weights in PERCENT for each ticker, in order, separated by commas or
spaces.\nExample: 25, 25, 25, 25  -> ",
    min_pct=0.0,     # per-asset lower bound in percent (e.g., 0 for long-only)
    max_pct=100.0,   # per-asset upper bound in percent
    tol=1e-4         # tolerance on total = 100%
):
    """
    Accepts ONLY percent inputs (e.g., 10 means 10%, 1.2 means 1.2%).
    Enforces:
      - number of weights == len(tickers)
      - per-asset bounds: min_pct <= w_i <= max_pct
      - total sum == 100% within tolerance
    Returns:
      pd.Series of decimal weights that sum to 1.0 (for computations).
    """
    tickers = list(tickers)
    n = len(tickers)
    # numeric tokens like 10, 1.2, .5
    token_re = re.compile(r"^[0-9]*\.?[0-9]+$")

    while True:
        raw = input(prompt).strip()
        # allow "equal" shortcut
        if raw.lower() == "equal":
            w_pct = [100.0 / n] * n
        else:
            parts = [p for p in re.split(r"[,\s]+", raw) if p]
            if len(parts) != n:
                print(f"Please provide exactly {n} weights to match: {', '.join(tickers)}")
```

```
            continue
        # validate numeric tokens (percent-only)
        bad = [p for p in parts if not token_re.match(p)]
        if bad:
            print("Invalid weight token(s). Use percent numbers like 10 or 1.2. Offenders: " + ", ".join(bad))
            continue
        w_pct = [float(p) for p in parts]

    # per-asset bounds check in percent
    below = [i for i, v in enumerate(w_pct) if v < min_pct]
    above = [i for i, v in enumerate(w_pct) if v > max_pct]
    if below or above:
        if below:
            print("These tickers are below the minimum percent bound "
                  f"({min_pct}%): " + ", ".join(f"{tickers[i]}={w_pct[i]}%" for i in below))
        if above:
            print("These tickers exceed the maximum percent bound "
                  f"({max_pct}%): " + ", ".join(f"{tickers[i]}={w_pct[i]}%" for i in above))
        continue

    total = sum(w_pct)
    if abs(total - 100.0) > tol:
        print(f"Total must be 100%. Your total is {total:.4f}%. Please adjust and try again.")
        continue

    # convert to decimals for downstream math
    w_dec = [v / 100.0 for v in w_pct]
    return pd.Series(w_dec, index=tickers)
```

How to use

```
# After you have your validated tickers list in the chosen order:
weights = get_portfolio_weights_percent(tickers)
print("Weights (decimals):")
print(weights)
print("Sum =", float(weights.sum()))
```

Notes

Users type only percents. Examples: 10 20 30 40, 10, 20, 30, 40, 1.2 98.8.

Type equal to auto-fill equal weights.

To set bounds like 0% to 30% per asset, call:
weights = get_portfolio_weights_percent(tickers, min_pct=0, max_pct=30)

If any constraint is violated, the function shows a clear error and reprompts.

Exhibit A-2 — Chat Session 2
Tool / Model: ChatGPT5
Date of usage: 9/28
Purpose: Debugging
Prompt: so we do want to accept negative weights as long as it adds up to 100. # add code here to enter original portfolio weights import re import pandas as pd # expects tickers to already be a list def get_portfolio_weights(tickers, prompt="What is your original portfolio weight for each ticker, in order, separated by commas? Enter 10 for 10% or quit to exit\n", min_w=0.0, max_w=100.0, tolerance=1e-4): # ensures within weight constraint and prevents rounding error """Accept percent inputs and returns weights in decimal form, matching the number of tickers while also fitting the weight constraint.""" # tickers = list(tickers) n = len(tickers) token_re = re.compile(r"^[0-9]*\.?[0-9]+$") # 10, 1.2, .5 are allowed while True: raw = input(prompt).strip() user_w = [w for w in re.split(r"[,\s]+", raw) if w] if raw == "quit": print("See you next time.") break # n weights dont match n tickers if len(tickers) != n: print(f"Please provide exactly {n} weights to match: {', '.join(tickers)}") continue bad = [w for w in user_w if not token_re.match(w)] if bad: print("Invalid token(s). Use percent numbers only (e.g., 10 or 1.2). Offenders: " + ", ".join(bad)) continue # to get out of the loop if not raw: print("Please enter your portfolio weights.") w_pct = [float(w) for w in user_w] # per-asset bounds below = [i for i, v in enumerate(w_pct) if v < min_w] above = [i for i, v in enumerate(w_pct) if v > max_w] if below or above: if below: print("Below minimum bound " f"({min_w}%): " + ", ".join(f"{tickers[i]}={w_pct[i]}%" for i in below)) if above: print("Above maximum bound " f"({max_w}%): " + ", ".join(f"{tickers[i]}={w_pct[i]}%" for i in above)) continue total = sum(w_pct) if abs(total - 100.0) > tolerance: print(f"Total must be 100%. Your total is {total:.4f}%. Please adjust and try again.") continue return pd.Series([v/100.0 for v in w_pct], index=tickers) weights = get_portfolio_weights(tickers) weights # 31, 8, -10, 35, 5, 21, 10
Response: Perfect—here are the only changes needed so your weights input:

accepts negative percents (shorts),

checks the right length,

handles quit and empty cleanly,

enforces sum = 100% with a tiny tolerance.

What I changed

Allow negatives in the regex: from ^[0-9]*\.?[0-9]+$ → ^[+-]?[0-9]*\.?[0-9]+$.

Fix the length check: compare len(user_w) to n (not len(tickers) to n).

Handle "quit" early, case-insensitive, and return None.

If the user just presses Enter, prompt again.

Defaults allow shorts per-asset (min_w=-100, max_w=100). Change if you want tighter per-asset bounds.

Drop-in replacement

```
# === Percent-only portfolio weights (allows shorts) ===
# Input example: 31, 8, -10, 35, 5, 21, 10  (sums to 100%)
# Output: pd.Series of decimals (sum = 1.0), indexed by tickers

import re
import pandas as pd

def get_portfolio_weights(
    tickers,
    prompt=("What is your original portfolio weight for each ticker, in order, "
            "separated by commas or spaces? Enter 10 for 10% (type 'quit' to exit)\n"),
    min_w=-100.0,   # allow shorts by default; set to 0.0 if you want long-only
    max_w=100.0,
    tolerance=1e-4  # tolerance for total = 100%
):
    """
    Accept percent inputs (supports negatives) and return decimal weights.
    Enforces:
      - exactly one weight per ticker
      - per-asset bounds [min_w, max_w] in percent
      - total sum == 100% within tolerance
    """
    n = len(tickers)
    token_re = re.compile(r"^[+-]?[0-9]*\.?[0-9]+$")  # allow negatives like -10 or -0.5

    while True:
        raw = input(prompt).strip()
        if raw.lower() == "quit":
            print("See you next time.")
            return None

        if not raw:
            print("Please enter your portfolio weights.")
            continue

        # split on commas or whitespace
        user_w = [w for w in re.split(r"[,\s]+", raw) if w]
```

```
    # must match number of tickers
    if len(user_w) != n:
        print(f"Please provide exactly {n} weights to match: {', '.join(tickers)}")
        continue

    # token validation (percent numbers only, negatives allowed)
    bad = [w for w in user_w if not token_re.match(w)]
    if bad:
        print("Invalid token(s). Use percent numbers only (e.g., 10, 1.2, -5). Offenders: " + ", ".join(bad))
        continue

    w_pct = [float(w) for w in user_w]

    # per-asset bounds (in percent)
    below = [i for i, v in enumerate(w_pct) if v < min_w]
    above = [i for i, v in enumerate(w_pct) if v > max_w]
    if below or above:
        if below:
            print("Below minimum bound "
                  f"({min_w}%): " + ", ".join(f"{tickers[i]}={w_pct[i]}%" for i in below))
        if above:
            print("Above maximum bound "
                  f"({max_w}%): " + ", ".join(f"{tickers[i]}={w_pct[i]}%" for i in above))
        continue

    total = sum(w_pct)
    if abs(total - 100.0) > tolerance:
        print(f"Total must be 100%. Your total is {total:.4f}%. Please adjust and try again.")
        continue

    # convert to decimals for downstream math
    return pd.Series([v / 100.0 for v in w_pct], index=tickers)
```

Exhibit A-2 — Chat Session 3
Tool / Model: ChatGPT5
Date of usage: 9/28
Purpose: Debugging
Prompt: def get_dates(): # In-sample is_start = datetime.strptime(input("In-sample start (YYYY-MM): "), "%Y-%m") is_end = datetime.strptime(input("In-sample end (YYYY-MM): "), "%Y-%m") if is_start > is_end: raise ValueError("In-sample start must be <= end.") # Out-of-sample os_start = datetime.strptime(input("Out-of-sample start (YYYY-MM): "), "%Y-%m") os_end = datetime.strptime(input("Out-of-sample end (YYYY-MM): "), "%Y-%m") if os_start > os_end: raise ValueError("In-sample start must be <= end.") return is_start, is_end, os_start, os_end is_start,is_end,os_start,os_end = get_dates()

Response: Nice start. Here's a single, minimal get_dates() that:

accepts YYYY-MM,

snaps to month start / month end,

validates ordering,

requires OOS to start after IS ends (no overlap),

returns YYYY-MM-DD strings ready for SQL.

```
from datetime import datetime, date
from calendar import monthrange

def get_dates():
    """
    Prompt for YYYY-MM, snap to month boundaries, validate order,
    and ensure OOS starts after IS ends. Returns four YYYY-MM-DD strings.
    """
    # --- collect ---
    is_start_in = input("In-sample start (YYYY-MM): ").strip()
    is_end_in   = input("In-sample end (YYYY-MM): ").strip()
    os_start_in = input("Out-of-sample start (YYYY-MM): ").strip()
    os_end_in   = input("Out-of-sample end (YYYY-MM): ").strip()

    # --- parse as year-month ---
    is_start = datetime.strptime(is_start_in, "%Y-%m").date().replace(day=1)
    is_end_y, is_end_m = datetime.strptime(is_end_in, "%Y-%m").year, datetime.strptime(is_end_in, 
"%Y-%m").month
    is_end   = date(is_end_y, is_end_m, monthrange(is_end_y, is_end_m)[1])  # month end

    os_start = datetime.strptime(os_start_in, "%Y-%m").date().replace(day=1)
    os_end_y, os_end_m = datetime.strptime(os_end_in, "%Y-%m").year, datetime.strptime(os_end_in, 
"%Y-%m").month
    os_end   = date(os_end_y, os_end_m, monthrange(os_end_y, os_end_m)[1])  # month end

    # --- checks ---
    if is_start > is_end:
        raise ValueError(f"In-sample start {is_start} must be <= end {is_end}.")
    if os_start > os_end:
        raise ValueError(f"Out-of-sample start {os_start} must be <= end {os_end}.")
    # enforce non-overlap: OOS must begin AFTER IS ends
    if not (os_start > is_end):
```

```
        raise ValueError(f"Out-of-sample must start AFTER in-sample ends. IS end {is_end}, OOS start
{os_start}.")

    # --- return as strings for SQL BETWEEN ---
    return tuple(d.strftime("%Y-%m-%d") for d in (is_start, is_end, os_start, os_end))

# Example:
# is_start, is_end, os_start, os_end = get_dates()
# print("IS:", is_start, "to", is_end, "| OOS:", os_start, "to", os_end)
```

This keeps it to one function, uses only the standard library, and gives you clean month-boundary strings for your SQL.

Exhibit A-2 — Chat Session 4
Tool / Model: ChatGPT5
Date of usage: 9/28
Purpose: Debugging
Prompt: it doesnt account for if user does not put in a date or puts in date with incorrect format
Response: Gotcha — here's a single, minimal get_dates() that:

Re-prompts on empty input or bad format (must be YYYY-MM)

Snaps to month start / month end

Ensures IS start ≤ IS end, OOS start ≤ OOS end, and OOS starts after IS ends

Returns YYYY-MM-DD strings ready for SQL

```
from datetime import datetime, date
from calendar import monthrange
import re

def get_dates():
    """
    Prompt for YYYY-MM (no blanks, correct format), snap to month boundaries,
    validate ordering and non-overlap. Re-prompts on any error.
    Returns: (is_start, is_end, oos_start, oos_end) as 'YYYY-MM-DD' strings.
    """
    ym_pat = re.compile(r"^\d{4}-(0[1-9]|1[0-2])$")  # strict YYYY-MM

    while True:
        try:
            is_start_in = input("In-sample start (YYYY-MM): ").strip()
```

```
        is_end_in   = input("In-sample end   (YYYY-MM): ").strip()
        oos_start_in = input("Out-of-sample start (YYYY-MM): ").strip()
        oos_end_in   = input("Out-of-sample end   (YYYY-MM): ").strip()

        # empty checks
        if not is_start_in or not is_end_in or not oos_start_in or not oos_end_in:
            print("All four dates are required (format: YYYY-MM). Try again.")
            continue

        # format checks
        bad = [x for x in [("IS start",is_start_in),("IS end",is_end_in),
                           ("OOS start",oos_start_in),("OOS end",oos_end_in)]
               if not ym_pat.match(x[1])]
        if bad:
            print("Incorrect format for: " + ", ".join(f"{label}='{val}'" for label,val in bad)
                  + ". Use YYYY-MM (e.g., 2020-07). Try again.")
            continue

        # parse and snap to month boundaries
        y,m = map(int, is_start_in.split("-"));  is_start = date(y,m,1)
        y,m = map(int, is_end_in.split("-"));    is_end   = date(y,m,monthrange(y,m)[1])
        y,m = map(int, oos_start_in.split("-")); oos_start= date(y,m,1)
        y,m = map(int, oos_end_in.split("-"));   oos_end  = date(y,m,monthrange(y,m)[1])

        # ordering checks
        if is_start > is_end:
            print(f"In-sample start {is_start} must be <= end {is_end}. Try again.")
            continue
        if oos_start > oos_end:
            print(f"Out-of-sample start {oos_start} must be <= end {oos_end}. Try again.")
            continue
        if not (oos_start > is_end):
            print(f"OOS must start after IS ends. IS end {is_end}, OOS start {oos_start}. Try again.")
            continue

        return tuple(d.strftime("%Y-%m-%d") for d in (is_start, is_end, oos_start, oos_end))

    except Exception as e:
        # Just in case any unexpected parse error slips through
        print(f"Invalid input ({e}). Please use YYYY-MM. Try again.")
```

Exhibit A-2 — Chat Session 5
Tool / Model: ChatGPT5
Date of usage: 9/28

Purpose: Understanding
Prompt: runtime error() vs print()
Response: Great question. Short version:

print() just shows a message and keeps going. It does not stop execution.

raise RuntimeError(...) (or ValueError, etc.) stops execution right there unless you catch it with try/except. It's the right tool when the program can't sensibly continue.

When to use which (for your notebook)

Interactive prompts (tickers, dates, weights):
Use print() + loop to show a helpful message and re-prompt. You expect user mistakes here.

Data retrieval / pipeline steps (WRDS query, coverage, cov inverse, Excel write):
If something makes the run invalid (no rows, all tickers dropped, Σ not PD), raise an exception with a clear message. You don't want to silently continue.

Tiny patterns you can copy
1) User input with re-prompt (print + loop)
```
def get_portfolio_weights_percent(tickers):
    while True:
        raw = input("Enter weights in %: ").strip()
        if raw.lower() in {"q", "quit"}:
            return None
        parts = [p for p in re.split(r"[,\s]+", raw) if p]
        if len(parts) != len(tickers):
            print(f"Need {len(tickers)} numbers. Try again.")
            continue
        try:
            vals = [float(p) for p in parts]
        except ValueError:
            print("Use numbers like 10 or 1.2. Try again.")
            continue
        if abs(sum(vals) - 100) > 1e-4:
            print(f"Sum must be 100%. You entered {sum(vals):.4f}%. Try again.")
            continue
        return pd.Series([v/100 for v in vals], index=tickers)
```

2) Non-interactive pipeline step (raise on fatal)
```
def fetch_crsp_returns(db, tickers, start_ymd, end_ymd, min_obs=24):
    if not tickers:
        raise ValueError("No tickers provided.")
    # ... run SQL ...
```

```
    if df.empty:
        raise RuntimeError("CRSP query returned no rows. Check tickers and dates.")
    # ... cleaning ...
    obs_counts = df.groupby("ticker")["ret_total"].apply(lambda s: s.notna().sum())
    kept = obs_counts[obs_counts >= min_obs].index.tolist()
    if not kept:
        raise RuntimeError(f"All tickers dropped by coverage (min_obs={min_obs}).")
    # ... return ret_matrix ...
```

3) Catching a fatal and stopping cleanly at top level

```
try:
    ret_matrix, kept, dropped = fetch_crsp_returns(db, tickers, is_start, is_end, min_obs=24)
except Exception as e:
    print(f"Stopping: {e}")
    raise  # or omit raise if you just want to halt the flow in a notebook cell
```

Quick rule of thumb

If the user can fix input right away → print() and re-prompt.

If the state is invalid for computation → raise (fail fast with a clear message).

Don't mix them in the same function unless it's user-facing input; pipeline helpers should raise so bugs don't get hidden.

If you show me a specific spot you're unsure about, I'll mark it "print+loop" vs "raise" precisely.

Exhibit A-2 — Chat Session 6
Tool / Model: ChatGPT5
Date of usage: 9/28
Purpose: Debugging
Prompt: im getting this error: IS matrix: (68, 7) | kept: ['BJ', 'COST', 'DG', 'DLTR', 'KR', 'PSMT', 'WMT'] | dropped: []

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
/tmp/ipython-input-2570159572.py in <cell line: 0>()
     11
     12 # out of sample - function is giving an error. smth about permnos
---> 13 os_matrix, kept_oos, dropped_oos = fetch_crsp_returns(db, kept_is, oos_start, oos_end, min_obs=1) # change min coverage?
     14
     15 os_ret_matrix = os_matrix.reindex(columns=ret_matrix.columns)

/tmp/ipython-input-1009984672.py in fetch_crsp_returns(db, tickers, start_ymd, end_ymd, min_obs)
```

```
    40     df = db.raw_sql(q, date_cols=["date"])
    41     if df.empty:
---> 42         raise RuntimeError("CRSP query returned no rows. Check tickers and dates.")
    43
    44     # cleanup

RuntimeError: CRSP query returned no rows. Check tickers and dates.
```

this is my code block:

```
# function to retrieve crsp data for date range
import pandas as pd
import numpy as np

# need min obs for coverage
# we felt a minimum observation of 24 seemed good because <2yrs is too noisy

def fetch_crsp_returns(db, tickers, start_ymd, end_ymd, min_obs=24):
    """ Pull monthly CRSP for tickers and date range, clean + combine delisting returns,
    enforce coverage, and create a monthly return matrix (ret_total).
    Returns ret_matrix, kept and dropped tickers."""
    if not tickers:
        raise ValueError("No tickers provided.")

    q = f"""
        SELECT
            msf.permno,
            msf.date,
            msf.ret,
            msf.shrout,
            msf.prc,
            n.shrcd,
            n.exchcd,
            n.ticker,
            dl.dlret
        FROM crsp.msf AS msf
        JOIN crsp.msenames AS n
          ON msf.permno = n.permno
         AND n.namedt <= msf.date
         AND (msf.date <= n.nameendt OR n.nameendt IS NULL)
        LEFT JOIN crsp.msedelist AS dl
          ON msf.permno = dl.permno
         AND date_trunc('month', dl.dlstdt) = date_trunc('month', msf.date)
        WHERE msf.date BETWEEN '{start_ymd}' AND '{end_ymd}'
          AND n.shrcd IN (10, 11, 12)     -- US common shares/close equivalents
```

```
      AND n.exchcd IN (1, 2, 3)       -- NYSE/AMEX/NASDAQ
      AND n.ticker IN ({",".join(f"'{t}'" for t in tickers)})
    ORDER BY msf.permno, msf.date;
"""
df = db.raw_sql(q, date_cols=["date"])
if df.empty:
    raise RuntimeError("CRSP query returned no rows. Check tickers and dates.")

# cleanup
df["prc"] = df["prc"].abs()
df["ret"] = pd.to_numeric(df["ret"], errors="coerce") # sets to NaN instead of error
df["dlret"] = pd.to_numeric(df["dlret"], errors="coerce").fillna(0.0)

# incl delisting return: ret_total = (1+ret)*(1+dlret) - 1
df["ret_total"] = (1.0 + df["ret"]) * (1.0 + df["dlret"]) - 1.0

# Month-end key for pivot
df["month"] = pd.to_datetime(df["date"]).dt.to_period("M").dt.to_timestamp("M")

# calc Market cap
df["shrout"] = pd.to_numeric(df["shrout"], errors="coerce")
df["shrout_shares"] = df["shrout"] * 1000.0
df["me"] = df["prc"] * df["shrout_shares"]

# drop rows w/missing returns
df = df.dropna(subset=["ret_total"])

# Deduplicate (permno, month)
df = df.sort_values(["permno", "month"])
df = df.drop_duplicates(subset=["permno", "month"], keep="last")

# keep tickers with at least min_obs non-null returns
counts = df.groupby("ticker")["ret_total"].apply(lambda s: s.notna().sum())
kept = counts[counts >= min_obs].index.tolist()
dropped = sorted(set(df["ticker"]) - set(kept))

if not kept:
    raise RuntimeError(f"All tickers dropped by coverage (min_obs={min_obs}).")

df = df[df["ticker"].isin(kept)].copy()

# Pivot to return matrix (rows=month-end, cols=tickers)
ret_matrix = (
    df.pivot_table(index="month", columns="ticker", values="ret_total", aggfunc="last")
```

```
        .sort_index()
    )

    ret_matrix.index = pd.to_datetime(ret_matrix.index).to_period("M").to_timestamp("M")

    return ret_matrix, kept, dropped


# In-sample start (YYYY-MM): 2019-01
# In-sample end (YYYY-MM): 2024-08
# Out-of-sample start (YYYY-MM): 2025-01
# Out-of-sample end (YYYY-MM): 2025-08
# tickers = ('BJ','COST','DG','DLTR','KR','PSMT','WMT')
# (is_start, is_end, oos_start, oos_end)

# in sample
ret_matrix, kept_is, dropped_is = fetch_crsp_returns(db, tickers, is_start, is_end, min_obs=24)
print("IS matrix:", ret_matrix.shape, "| kept:", kept_is, "| dropped:", dropped_is)

# out of sample - function is giving an error. smth about permnos
os_matrix, kept_oos, dropped_oos = fetch_crsp_returns(db, kept_is, oos_start, oos_end, min_obs=1) # change min coverage?

os_ret_matrix = os_matrix.reindex(columns=ret_matrix.columns)
print("OOS matrix:", os_ret_matrix.shape)
```
Response: You're getting "no rows" OOS because filtering by ticker text can miss securities when the ticker label changes between IS and OOS. The robust fix is to pull OOS by permno. Here are the smallest possible edits:

1) Minimal change inside your function

Add an optional permnos arg and swap the ticker filter to use it when provided.

```
# CHANGE 1: add permnos=None in the signature
def fetch_crsp_returns(db, tickers, start_ymd, end_ymd, min_obs=24, permnos=None):
    ...
    # CHANGE 2: build the ID filter once
    if permnos:
        id_filter_sql = "msf.permno IN (" + ",".join(str(int(p)) for p in permnos) + ")"
    else:
        tickers_sql = ",".join("'" + t.replace("'", "''") + "'" for t in tickers)
        id_filter_sql = f"n.ticker IN ({tickers_sql})"
    ...
    q = f"""
```

```
    SELECT
        msf.permno,
        msf.date,
        msf.ret,
        msf.shrout,
        msf.prc,
        n.shrcd,
        n.exchcd,
        n.ticker,
        dl.dlret
    FROM crsp.msf AS msf
    JOIN crsp.msenames AS n
      ON msf.permno = n.permno
     AND n.namedt <= msf.date
     AND (msf.date <= n.nameendt OR n.nameendt IS NULL)
    LEFT JOIN crsp.msedelist AS dl
      ON msf.permno = dl.permno
     AND date_trunc('month', dl.dlstdt) = date_trunc('month', msf.date)
    WHERE msf.date BETWEEN '{start_ymd}' AND '{end_ymd}'
      AND n.shrcd IN (10, 11, 12)
      AND n.exchcd IN (1, 2, 3)
      AND {id_filter_sql}              -- CHANGE 3: use unified filter
    ORDER BY msf.permno, msf.date;
"""
```

Nothing else in the function needs to change.

2) Resolve permnos once from the IS window, then call OOS with them

Insert these lines after your IS call:

```
# Resolve permanent IDs for the kept IS tickers over the IS window
q_perm = f"""
    SELECT DISTINCT msf.permno
    FROM crsp.msf AS msf
    JOIN crsp.msenames AS n
      ON msf.permno = n.permno
     AND n.namedt <= msf.date
     AND (msf.date <= n.nameendt OR n.nameendt IS NULL)
    WHERE msf.date BETWEEN '{is_start}' AND '{is_end}'
      AND n.ticker IN ({",".join("'" + t + "'" for t in kept_is)})
"""
permnos_is = db.raw_sql(q_perm)["permno"].astype(int).tolist()
```

```
print("Resolved permnos:", permnos_is)
```

Then change your OOS call to:

```
os_matrix, kept_oos, dropped_oos = fetch_crsp_returns(
    db, kept_is, oos_start, oos_end, min_obs=1, permnos=permnos_is
)
```

Finally:

```
os_ret_matrix = os_matrix.reindex(columns=ret_matrix.columns)
print("OOS matrix:", os_ret_matrix.shape)
```

Why this fixes it

permno is CRSP's permanent identifier and does not change when the ticker string changes.

Filtering OOS by permno guarantees you pull the same securities across time.

If OOS still returns zero rows after this, double check your OOS dates actually have monthly data available.

Exhibit A-2 — Chat Session 7
Tool / Model: ChatGPT5
Date of usage: 9/28
Purpose: Debugging
Prompt: i just want the ticker names matrices, this is my code so far: # tickers = ('BJ','COST','DG','DLTR','KR','PSMT','WMT') # (is_start, is_end, oos_start, oos_end) # in sample ret_matrix, kept_is, dropped_is, df_is = fetch_crsp_returns(db, tickers, is_start, is_end, min_obs=24) print("IS matrix:", ret_matrix.shape, "| kept:", kept_is, "| dropped:", dropped_is) # Resolve permnos directly from the cleaned IS df permnos_is = sorted(df_is["permno"].dropna().astype(int).unique().tolist()) print("Resolved permnos:", permnos_is) # out of sample - function is giving an error. smth about permnos # coverage? os_matrix, kept_oos, dropped_oos = fetch_crsp_returns(db, kept_is, oos_start, oos_end, min_obs=1, permnos=permnos_is) # align OS to IS os_ret_matrix = os_matrix.reindex(columns=ret_matrix.columns) print("OOS matrix:", os_ret_matrix.shape) # OS matrix is permnos based, so were coverting them to tickers permno_to_ticker = dict(df_is.drop_duplicates(subset=["permno"])[["permno","ticker"]].itertuples(index=False)) # IS matrix: (60, 7) kept all stocks # OOS matrix: (12, 7)
Response: You're super close. Since you only want ticker-named matrices, the safest move is:

Capture df_oos from your OOS call (your function returns 4 things).

Build permno-based IS and OOS matrices from df_is and df_oos.

Rename both with the IS permno→ticker map so OOS uses the same ticker labels even if the live OOS label changed.

Drop this in place of your OOS block (minimal edits, keep everything else you have):

```
# --- In-sample (unchanged) ---
ret_matrix, kept_is, dropped_is, df_is = fetch_crsp_returns(db, tickers, is_start, is_end, min_obs=24)
print("IS matrix:", ret_matrix.shape, "| kept:", kept_is, "| dropped:", dropped_is)

# Resolve permnos from the cleaned IS df
permnos_is = sorted(df_is["permno"].dropna().astype(int).unique().tolist())
print("Resolved permnos:", permnos_is)

# --- Out-of-sample: CAPTURE df_oos too ---
os_matrix, kept_oos, dropped_oos, df_oos = fetch_crsp_returns(
    db, kept_is, oos_start, oos_end, min_obs=1, permnos=permnos_is
)

# --- Build permno-based matrices from the cleaned frames (robust to ticker drift) ---
ret_perm = (df_is.pivot_table(index="month", columns="permno", values="ret_total", aggfunc="last")
            .sort_index())
oos_perm = (df_oos.pivot_table(index="month", columns="permno", values="ret_total", aggfunc="last")
            .sort_index()
            .reindex(columns=ret_perm.columns))  # align OOS to IS permnos

# --- Rename columns to TICKERS using IS mapping (presentation only) ---
# Use the last IS label per permno to be consistent
permno_to_ticker = (df_is.sort_values(["permno","month"])
                    .drop_duplicates(subset=["permno"], keep="last")
                    .set_index("permno")["ticker"]
                    .to_dict())

ret_matrix_tickers = ret_perm.rename(columns=permno_to_ticker)
os_ret_matrix      = oos_perm.rename(columns=permno_to_ticker)  # <- OOS now uses IS ticker labels

print("IS (tickers):", ret_matrix_tickers.shape)
print("OOS (tickers):", os_ret_matrix.shape)

# OPTIONAL: save to CSVs
from pathlib import Path
import pandas as pd
outdir = Path("outputs"); outdir.mkdir(exist_ok=True)
```

```
ts = pd.Timestamp.now().strftime("%Y%m%d_%H%M%S")
ret_matrix_tickers.index.name = "month_end"
os_ret_matrix.index.name = "month_end"
ret_matrix_tickers.to_csv(outdir / f"IS_returns_tickers_{ts}.csv", float_format="%.8f", na_rep="NA")
os_ret_matrix.to_csv(outdir / f"OOS_returns_tickers_{ts}.csv", float_format="%.8f", na_rep="NA")
print("Saved CSVs to:", outdir)
```

Why this works:

All math happens on permno internally, which never changes.

Final column names come from the IS mapping, so your OOS matrix has the same ticker headers as IS, even if the live OOS label differs.

You still end up with exactly what you want: ticker-named IS and OOS matrices.

Exhibit A-2 — Chat Session 8
Tool / Model: ChatGPT5
Date of usage: 9/29
Purpose: Code generation
Prompt: do i dont have everything yet for the worksheet, can u giv eme a set up?
Response: Gotcha. Here's a clean worksheet skeleton you can run now. It creates a short-named Excel file with the key tabs you'll fill in later: README, Inputs, IS, OOS, Coverage, Weights, Metrics, and Plots. If ret_matrix_tickers and os_ret_matrix already exist, it writes them; otherwise it drops placeholders so you can wire things up later.

```
# === Minimal Excel worksheet setup for your project ===
from pathlib import Path
import pandas as pd
from openpyxl import load_workbook
from openpyxl.styles import Font, Alignment, numbers
from openpyxl.utils import get_column_letter

# ---- Inputs you may already have (optional) ----
# ret_matrix_tickers  # IS returns, index=month-end, columns=tickers (decimal)
# os_ret_matrix       # OOS returns, same columns

outdir = Path("outputs"); outdir.mkdir(exist_ok=True)
xlsx_path = outdir / "report.xlsx"   # short name; overwrite each run

# Fallback placeholders if matrices aren't built yet
def _ensure_df(df, name):
    if "DataFrame" in str(type(df)):
```

```
        return df.copy()
    # minimal placeholders
    if name == "IS":
        return pd.DataFrame({"_PLACEHOLDER_TICKER_": []}, index=pd.Index([],
name="month_end"))
    if name == "OOS":
        return pd.DataFrame({"_PLACEHOLDER_TICKER_": []}, index=pd.Index([],
name="month_end"))

try:
    _is_df = _ensure_df(ret_matrix_tickers, "IS")
except NameError:
    _is_df = _ensure_df(None, "IS")
try:
    _oos_df = _ensure_df(os_ret_matrix, "OOS")
except NameError:
    _oos_df = _ensure_df(None, "OOS")

# ---- Create base workbook with the core sheets ----
with pd.ExcelWriter(xlsx_path, engine="openpyxl") as writer:
    # README
    pd.DataFrame({
        "What to do next": [
            "Fill Inputs sheet (tickers, dates, weights).",
            "Paste or generate IS and OOS return matrices as decimals.",
            "Check Coverage tab; drop names with poor coverage if needed.",
            "Enter or paste portfolio weights in Weights tab.",
            "Compute Metrics (IS & OOS returns, stdev, Sharpe).",
            "Insert charts in Plots tab (frontier, perf)."
        ]
    }).to_excel(writer, sheet_name="README", index=False)

    # Inputs (simple scaffold)
    inputs = pd.DataFrame({
        "Field": ["Tickers (comma-separated)",
                  "IS start (YYYY-MM-DD)", "IS end (YYYY-MM-DD)",
                  "OOS start (YYYY-MM-DD)", "OOS end (YYYY-MM-DD)",
                  "Weights entry (10 = 10%, or 'equal')"],
        "Value": [", ".join(_is_df.columns) if len(_is_df.columns)>0 else "",
                  "", "", "", "", ""]
    })
    inputs.to_excel(writer, sheet_name="Inputs", index=False)

    # IS & OOS matrices
```

```
    _is_df.index.name = _is_df.index.name or "month_end"
    _oos_df.index.name = _oos_df.index.name or "month_end"
    _is_df.to_excel(writer, sheet_name="IS")
    _oos_df.to_excel(writer, sheet_name="OOS")

    # Coverage scaffold
    cov = pd.DataFrame({
        "ticker": _is_df.columns,
        "obs": [_is_df[c].notna().sum() for c in _is_df.columns] if _is_df.shape[1]>0 else [],
        "coverage_%": [(_is_df[c].notna().sum()/max(1,len(_is_df))*100) for c in _is_df.columns] if
_is_df.shape[1]>0 else []
    })
    cov.to_excel(writer, sheet_name="Coverage", index=False)

    # Weights scaffold (one row, percent entry)
    weights = pd.DataFrame(columns=_is_df.columns)
    weights.loc["Enter % here"] = ""
    weights.to_excel(writer, sheet_name="Weights")

    # Metrics scaffold
    metrics_rows = ["IS mean %", "IS stdev %", "IS Sharpe", "OOS mean %", "OOS stdev %", "OOS
Sharpe"]
    metrics = pd.DataFrame(index=metrics_rows, columns=["Portfolio", "Notes"])
    metrics.to_excel(writer, sheet_name="Metrics")

    # Plots placeholder
    pd.DataFrame({"Drop charts here": ["Efficient frontier", "Portfolio points", "OOS
performance"]}).to_excel(
        writer, sheet_name="Plots", index=False
    )

# ---- Light formatting (headers, column widths, percent formats on data sheets) ----
wb = load_workbook(xlsx_path)

def style_table(ws, percent=False):
    # bold header
    for cell in ws[1]:
        cell.font = Font(bold=True)
    # freeze header
    ws.freeze_panes = "B2" if ws.max_column > 1 else "A2"
    # filter
    ws.auto_filter.ref = ws.dimensions
    # widths
    for c in range(1, ws.max_column + 1):
```

```python
        ws.column_dimensions[get_column_letter(c)].width = 13 if c == 1 else 10
    # percent format (for data cells only)
    if percent:
        for r in range(2, ws.max_row + 1):
            for c in range(2, ws.max_column + 1):
                ws.cell(row=r, column=c).number_format = numbers.FORMAT_PERCENTAGE_00

# Style each sheet
for name in ["README", "Inputs", "Coverage", "Weights", "Metrics", "Plots"]:
    ws = wb[name]
    style_table(ws, percent=False)

# IS/OOS as percentages (they're decimal returns)
style_table(wb["IS"], percent=True)
style_table(wb["OOS"], percent=True)

# Center some labels
for name in ["README", "Plots"]:
    ws = wb[name]
    for row in ws.iter_rows(min_row=2, max_row=ws.max_row, min_col=1, max_col=ws.max_column):
        for cell in row:
            cell.alignment = Alignment(horizontal="left")

wb.save(xlsx_path)
print(f"Worksheet skeleton saved: {xlsx_path}")
```

What you get:

README tells the story of what to fill next

Inputs has slots for tickers, dates, weights

IS and OOS tabs ready for matrices, formatted as percentages

Coverage prefilled from IS if available

Weights sheet for your original or optimized weights

Metrics placeholders for IS and OOS stats

Plots tab as a drop zone for charts later

Run this now; you can keep rerunning as your matrices and metrics come together.

Exhibit A-2 — Chat Session 9
Tool / Model: ChatGPT5
Date of usage: 9/29
Purpose: Understanding
Prompt: this is my ret_matrix: month_end KR BJ WMT DLTR PSMT COST DG but when i run this function i get this: # get avg returns to demean later def mean_returns_monthly(ret_matrix: pd.DataFrame) -> pd.Series: """ Input: ret_matrix = DataFrame of monthly simple returns (60 rows x N tickers). Output: Series of average monthly return per ticker (NaNs ignored). """ # If there are non-numeric cols, coerce them (optional; remove if not needed) # not sure if necessary rets_num = ret_matrix.apply(pd.to_numeric, errors="coerce") return rets_num.mean(axis=0) # monthly mean per column 0 month_end NaN KR 0.012904 BJ 0.022978 WMT 0.011502 DLTR 0.012306
Response: You're seeing month_end NaN because month_end is a column in your DataFrame, not the index. Your function coerces everything to numeric, so the date column turns into all-NaNs and then shows up in the mean with NaN. The lone 0 you see on top is just the Series name column when pandas prints a Series like a 1-col table.

You can fix this two ways:

A) Structural fix (recommended): set the date as the index once

Do this right after you build or load the matrix:

```
# one-time fix when you construct ret_matrix_tickers
if "month_end" in ret_matrix.columns:
    ret_matrix = ret_matrix.set_index("month_end")
```

Then your original mean function will work.

B) Minimal guard inside the function (drop non-asset cols)

If you want the function to be robust even when month_end is still a column:

```
import numpy as np
import pandas as pd

def mean_returns_monthly(ret_matrix: pd.DataFrame) -> pd.Series:
    """
    Return mean monthly simple returns per ticker. Ignores non-numeric columns like month_end.
    """
    # If month_end leaked as a column, set it as index; otherwise leave as is
    if "month_end" in ret_matrix.columns:
```

```python
        tmp = ret_matrix.set_index("month_end")
    else:
        tmp = ret_matrix

    # Keep only numeric columns (tickers), then take column means
    tmp_num = tmp.select_dtypes(include=[np.number])
    return tmp_num.mean(axis=0, skipna=True)
```

Bonus: your other two functions (matching this approach)

```python
def demean_returns(ret_matrix: pd.DataFrame) -> pd.DataFrame:
    """Return matrix with each column demeaned; preserves index & columns."""
    if "month_end" in ret_matrix.columns:
        tmp = ret_matrix.set_index("month_end")
    else:
        tmp = ret_matrix
    num = tmp.select_dtypes(include=[np.number])
    mu  = num.mean(axis=0, skipna=True)
    out = num.sub(mu, axis=1)
    # keep any non-numeric columns (if you had them) alongside
    return out

def cov_matrix(ret_matrix: pd.DataFrame, ddof: int = 1, min_periods: int = 1, ridge: float = 0.0) -> 
pd.DataFrame:
    """Sample covariance of monthly returns with optional ridge for stability."""
    if "month_end" in ret_matrix.columns:
        tmp = ret_matrix.set_index("month_end")
    else:
        tmp = ret_matrix
    num = tmp.select_dtypes(include=[np.number])
    Sigma = num.cov(min_periods=min_periods, ddof=ddof)
    if ridge and Sigma.shape[0] > 0:
        Sigma = Sigma + ridge * np.eye(Sigma.shape[0])
        Sigma = pd.DataFrame(Sigma, index=num.columns, columns=num.columns)
    return Sigma
```

Either set the index once (cleanest), or keep the guards in the functions so they ignore the date column.

Exhibit A-2 — Chat Session 10
Tool / Model: ChatGPT5
Date of usage: 9/29
Purpose: Understanding

Prompt: why do u have all thise xtra stuff: def minverse(covmat: pd.DataFrame, extra_ridge: float = 1e-6) -> pd.DataFrame: """ Excel-like MINVERSE: returns Σ^{-1} as a DataFrame. Adds a tiny ridge to stabilize, uses inv; falls back to pinv if singular/ill-conditioned. """ A = covmat.to_numpy(dtype=float) if extra_ridge: A = A + extra_ridge * np.eye(A.shape[0]) try: invA = np.linalg.inv(A) except np.linalg.LinAlgError: invA = np.linalg.pinv(A) return pd.DataFrame(invA, index=covmat.index, columns=covmat.columns)

Response: Totally fair question. The "extra stuff" is there to keep you from crashing or getting garbage weights when Σ is singular or nearly singular—which happens a lot with real return matrices.

Why each piece exists

extra_ridge (add εI before inverting)
Covariance matrices from short samples, highly correlated stocks, or duplicated/near-constant columns can be ill-conditioned. Adding a tiny diagonal (e.g., 1e-6) nudges Σ to be invertible and reduces numeric noise. It barely changes results but prevents blow-ups.

try: inv ... except: pinv fallback
If Σ is still singular (exactly or numerically), np.linalg.inv raises LinAlgError. The Moore–Penrose pseudo-inverse (pinv) gives you the best least-squares solution so you can still compute GMV/target-return weights instead of dying with an error.

to_numpy(dtype=float) and wrapping back to DataFrame
Ensures the linear algebra uses clean float arrays, then returns the result with the same row/col labels as Σ so you don't lose tickers.

When would a "simple" inverse be OK?

If you've already:

dropped all-NaN and zero-variance columns,

have T >> N observations,

and your columns aren't highly collinear,

then you might get away with:

```
def minverse_simple(S: pd.DataFrame) -> pd.DataFrame:
    A = S.to_numpy(dtype=float)
    invA = np.linalg.inv(A)  # may fail if S is singular/ill-conditioned
    return pd.DataFrame(invA, index=S.index, columns=S.columns)
```

…but in practice portfolios + CRSP often violate those assumptions. The robust version saves you from edge cases with almost no downside.

Quick sanity check (useful once)

If you're curious whether you "need" the extras for a given Σ:

```
import numpy as np

def cond_number(S: pd.DataFrame) -> float:
    # larger = worse (ill-conditioned)
    return float(np.linalg.cond(S.to_numpy(dtype=float)))

kappa = cond_number(SIGMA_S)  # or your chosen covariance
print("Condition number:", f"{kappa:,.0f}")
```

If the condition number is ~1e6 or higher, use the ridge + pinv safety net.

Keep or trim?

Keep the robust version for reliability on all datasets.

If your instructor insists on matching Excel's MINVERSE behavior exactly (no ridge), set extra_ridge=0.0 and keep the pinv fallback to avoid crashes:

```
invA = np.linalg.pinv(A)  # always safe, Excel-like results when S is singular
```

Exhibit A-3 — Chat Session 1
Tool / Model: ChatGPT5
Date of usage: 9/30
Purpose: Debugging
Prompt: def calc_orp(invcovmat: pd.DataFrame, means: pd.Series, rfrate: float) -> pd.Series:

```
    """
    Optimal Risky Portfolio (tangency) weights.
    Excel analog:
      Numerator   = MMULT(INVCOVMAT, muvec - rfrate)
      Denominator = MMULT(TRANSPOSE(ONEVEC), MMULT(INVCOVMAT, muvec - rfrate))
    Returns weights that sum to 1.
    """
    # Align and cast
    mu = means.reindex(invcovmat.index).to_numpy(dtype=float).reshape(-1, 1)
    P  = invcovmat.to_numpy(dtype=float)
    e  = np.ones((P.shape[0], 1))
```

```
    x = P @ (mu - rfrate * e)           # numerator vector
    denom = float(e.T @ x)              # scalar denominator
    w = (x / denom).ravel()             # normalize to sum to 1
    return pd.Series(w, index=invcovmat.index, name="ORP_Weight")
```

redo this but using scipy.optimize.minimize

Response: Here's how you can redo your Optimal Risky Portfolio (ORP) weights calculation using scipy.optimize.minimize instead of the closed-form solution. This approach numerically finds the weights that maximize the Sharpe ratio (i.e., maximize excess return per unit risk), subject to weights summing to 1.

Plug-and-Play ORP with scipy.optimize.minimize

python

```
import numpy as np
import pandas as pd
from scipy.optimize import minimize

def calc_orp_scipy(covmat: pd.DataFrame, means: pd.Series, rfrate: float) -> pd.Series:
    n = len(means)
    # Initial guess: equal weights
    w0 = np.ones(n) / n
    # Objective: negative Sharpe ratio (since minimize finds minimum)
    def neg_sharpe(w):
        port_ret = np.dot(w, means)   # expected portfolio return
        port_vol = np.sqrt(np.dot(w, np.dot(covmat, w)))  # portfolio std dev
        return -(port_ret - rfrate) / port_vol
    # Constraint: weights sum to 1
    cons = {'type': 'eq', 'fun': lambda w: np.sum(w) - 1}
    # Bounds: weights between 0 and 1 (no shorting)
    bounds = [(0, 1)] * n
    result = minimize(neg_sharpe, w0, method='SLSQP', bounds=bounds, constraints=cons)
    # Return as Series with tickers as index
    return pd.Series(result.x, index=means.index, name="ORP_Weight")
```

How it works:

The function neg_sharpe computes the negative Sharpe ratio for a given set of weights.

minimize finds the weights that maximize the Sharpe ratio, subject to weights summing to 1 and being between 0 and 1.

The result is returned as a pandas Series, just like your closed-form version.

You can use this function exactly like your original calc_orp function. Let me know if you want to allow shorting (weights < 0), or set custom bounds!
https://towardsdatascience.com/portfolio-optimization-with-scipy-aa9c02e6b937/
https://www.youtube.com/watch?v=ehZjZdk9iFM
https://blog.mlq.ai/python-for-finance-portfolio-optimization/
https://www.askpython.com/python-modules/scipy-minimize
https://www.youtube.com/watch?v=D3O6riW4zDU
https://www.kaggle.com/code/vijipai/lesson-7-constrained-portfolio-optimization
https://docs.scipy.org/doc/scipy/tutorial/optimize.html
https://builtin.com/data-science/portfolio-optimization-python

Exhibit A-3 — Chat Session 2
Tool / Model: ChatGPT5
Date of usage: 9/30
Purpose: Debugging
Prompt: pfs = pd.DataFrame(columns=["Expected Return","Standard Deviation"])
pfs.loc['IS Orig Port'] = [portfolio_return(weights, means), port_stdev(weights, covmat)]
pfs.loc['OS Orig Port'] = [portfolio_return(weights, os_means), port_stdev(weights, os_covmat)]
pfs.loc['IS GMV'] = [portfolio_return(gmv, means), port_stdev(gmv, covmat)]
pfs.loc['OS GMV'] = [portfolio_return(gmv, os_means), port_stdev(gmv, os_covmat)]
pfs.loc['IS Efficient Portfolio'] = [portfolio_return(efport, means), port_stdev(efport, covmat)]
pfs.loc['OS Efficient Portfolio'] = [portfolio_return(efport, os_means), port_stdev(efport, os_covmat)]
pfs.loc['IS ORP'] = [portfolio_return(orp, means), port_stdev(orp, covmat)]
pfs.loc['OS ORP'] = [portfolio_return(orp, os_means), port_stdev(orp, os_covmat)]
pfs.loc['IS QP ORP'] = [portfolio_return(qporp, means), port_stdev(qporp, covmat)]
pfs.loc['OS QP ORP'] = [portfolio_return(qporp, os_means), port_stdev(qporp, os_covmat)]

pfs['Sharpe Ratio (Monthly)'] = (pfs['Expected Return']-rfrate)/pfs['Standard Deviation']
pfs['Sharpe Ratio (Annualized)'] = (pfs['Sharpe Ratio (Monthly)'])*(12**0.5)
this is the setup of a table i have with different portfolios. i want to make an efficient frontier chart. this chart should plot each portfolio (combination of standard deviation vs expected return). i want the in sample and out of sample (IS vs OS) in different colors. i want each portfolio labeled. also, draw the capital allocaiton line in a diff color, which should intercept at teh y-axis at rfrate, teh risk free rate. also, i want it to intercept the orp for each portfolio. also, draw the efficient frontier for each in its own color. give me the plt python code to do this
Response: import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

Assumptions:

```
# - pfs has index labels like 'IS GMV', 'OS GMV', 'IS ORP', 'OS ORP', etc.
# - pfs has columns: 'Expected Return', 'Standard Deviation'
# - rfrate is a scalar (monthly, consistent with Expected Return units)

assert {'Expected Return','Standard Deviation'}.issubset(pfs.columns)

# Split IS vs OS
is_mask = pfs.index.str.startswith('IS ')
os_mask = pfs.index.str.startswith('OS ')

sns.set(style='whitegrid')
fig, ax = plt.subplots(figsize=(8, 6), dpi=120)

# Scatter: IS vs OS
ax.scatter(pfs.loc[is_mask, 'Standard Deviation'],
        pfs.loc[is_mask, 'Expected Return'],
        c='#1f77b4', marker='o', s=70, label='In-Sample (IS)')
ax.scatter(pfs.loc[os_mask, 'Standard Deviation'],
        pfs.loc[os_mask, 'Expected Return'],
        c='#ff7f0e', marker='^', s=80, label='Out-of-Sample (OS)')

# Labels for each portfolio
for name, row in pfs.iterrows():
    ax.annotate(name,
            (row['Standard Deviation'], row['Expected Return']),
            textcoords='offset points', xytext=(6, 4), fontsize=8)

# Capital Allocation Lines: one for IS and one for OS, through their ORP
cal_colors = {'IS': '#2ca02c', 'OS': '#d62728'}  # green IS, red OS
for regime, mask in [('IS', is_mask), ('OS', os_mask)]:
    # Prefer ORP; fall back to QP ORP if needed
    candidates = [idx for idx in pfs.index[mask] if ('ORP' in idx) and ('QP' not in idx)]
    if not candidates:
        candidates = [idx for idx in pfs.index[mask] if 'QP ORP' in idx]
    if candidates:
        orp_label = candidates[0]
        sd_orp = float(pfs.loc[orp_label, 'Standard Deviation'])
        er_orp = float(pfs.loc[orp_label, 'Expected Return'])
        slope = (er_orp - rfrate) / max(sd_orp, 1e-12)
        x_max = max(float(pfs['Standard Deviation'].max())*1.1, sd_orp*1.15)
        x_vals = np.linspace(0, x_max, 200)
        y_vals = rfrate + slope * x_vals
        ax.plot(x_vals, y_vals, color=cal_colors[regime], linestyle='--', linewidth=1.8,
            label=f'CAL ({regime}) via {orp_label}')
```

```
        # Risk-free intercept marker
        ax.scatter([0], [rfrate], color=cal_colors[regime], edgecolor='k', zorder=3)

# Efficient frontier curves for IS and OS:
# If specific points exist (GMV, Efficient Portfolio), use them; else fit through all regime points.
frontier_specs = [
    ('IS', is_mask, '#17becf', ['IS GMV', 'IS Efficient Portfolio']),
    ('OS', os_mask, '#8c564b', ['OS GMV', 'OS Efficient Portfolio'])
]

for regime, mask, color, pref_labels in frontier_specs:
    regime_df = pfs.loc[mask, ['Standard Deviation','Expected Return']].dropna()
    if len(regime_df) < 2:
        continue
    if all(lbl in regime_df.index for lbl in pref_labels):
        pts = regime_df.loc[pref_labels]
    else:
        pts = regime_df.sort_values('Standard Deviation')
    x = pts['Standard Deviation'].values
    y = pts['Expected Return'].values
    if len(pts) >= 3:
        # Quadratic fit as a smooth approximation to the frontier
        coeffs = np.polyfit(x, y, 2)
        x_fit = np.linspace(x.min(), x.max(), 200)
        y_fit = np.polyval(coeffs, x_fit)
        ax.plot(x_fit, y_fit, color=color, linewidth=2.0, label=f'Efficient Frontier ({regime})')
    else:
        # With only two points, connect them
        ax.plot(x, y, color=color, linewidth=2.0, label=f'Efficient Frontier ({regime})')

ax.set_xlabel('Standard Deviation')
ax.set_ylabel('Expected Return')
ax.set_title('Efficient Frontier, CAL, and Portfolios')
ax.legend(frameon=True, loc='best')
plt.tight_layout()
plt.show()
```

Exhibit A-3 — Chat Session 3
Tool / Model: ChatGPT5
Date of usage: 9/30
Purpose: Debugging
Prompt: i noticed for most of the sheets the first numerical row are not in percentage form
Response: Good catch — that's a slicing bug. When we format columns we used [header_row+1:], but openpyxl gives you a 0-indexed tuple of cells per column ([row1, row2, row3, ...]). So:

Header is at index 0 (row 1).

First numeric row is at index 1 (row 2).

By slicing with +1, we accidentally started at row 3, skipping the first numeric row. Same issue in the Monthly Returns sheets where I used [2:].

Fix (minimal patch)

Change 4 places:

In _format_percent(...) replace:

```
for cell in ws[col_letter][header_row+1:]:
```

with:

```
for cell in ws[col_letter][header_row:]:
```

In _format_ratio(...) replace:

```
for cell in ws[get_column_letter(ci)][header_row+1:]:
```

with:

```
for cell in ws[get_column_letter(ci)][header_row:]:
```

In “Monthly Returns (IS)” percent loop replace:

```
for cell in ws_ris[get_column_letter(col)][2:]:
```

with:

```
for cell in ws_ris[get_column_letter(col)][1:]:
```

In “Monthly Returns (OS)” percent loop replace:

```
for cell in ws_ros[get_column_letter(col)][2:]:
```

with:

```
for cell in ws_ros[get_column_letter(col)][1:]:
```

If you prefer to paste the corrected functions

```
def _format_percent(ws, header_row=1, include_cols=None, exclude_cols=None):
    headers = {}
    for cell in ws[header_row]:
        if cell.value:
            headers[cell.column] = str(cell.value).strip()
    norm = lambda s: s.lower().strip()
    include = {norm(c) for c in include_cols} if include_cols else set()
    exclude = {norm(c) for c in exclude_cols} if exclude_cols else set()
    keywords = ["expected return", "standard deviation", "return", "ret", "stdev", "std dev", "vol",
"volatility"]
    for ci, name in headers.items():
        n = norm(name)
        if n in exclude:
            continue
        use_pct = (n in include) if include else (any(k in n for k in keywords) or n.endswith("(ann.)") or
n.endswith("_a") or n.endswith("_m"))
        col_letter = get_column_letter(ci)
        if use_pct:
            # start at index 1 (row 2), not 2
            for cell in ws[col_letter][header_row:]:
                if isinstance(cell.value, (int, float)):
                    cell.number_format = "0.00%"

def _format_ratio(ws, header_row=1):
    headers = {}
    for cell in ws[header_row]:
        if cell.value:
            headers[cell.column] = str(cell.value).lower()
    for ci, name in headers.items():
        if "sharpe" in name or "ratio" in name:
            col_letter = get_column_letter(ci)
            # start at index 1 (row 2)
            for cell in ws[col_letter][header_row:]:
                if isinstance(cell.value, (int, float)):
                    cell.number_format = "0.000"
```

And the Monthly Returns loops:

```
# Monthly Returns (IS)
for col in range(2, ws_ris.max_column+1):
    for cell in ws_ris[get_column_letter(col)][1:]:  # row2 onward
        if isinstance(cell.value, (int, float)):
            cell.number_format = "0.00%"

# Monthly Returns (OS)
for col in range(2, ws_ros.max_column+1):
    for cell in ws_ros[get_column_letter(col)][1:]:  # row2 onward
        if isinstance(cell.value, (int, float)):
            cell.number_format = "0.00%"
```

Re-run your "build Excel" cell after making these edits and the first numeric row will now be in percent everywhere.

Exhibit A-3 — Chat Session 4
Tool / Model: ChatGPT5
Date of usage: 9/30
Purpose: Debugging
Prompt: the excel doesnt show all the numbers in percent how to fix
Response: I looked through your script

las_traders_project_2_py (2)

. The reason your Excel output doesn't consistently show percentages is the formatter function.

In the first version you had:

```
def _format_percent(ws, header_row=1):
    # Apply percent format to columns that look like returns or stdevs
    headers = {}
    for cell in ws[header_row]:
        if cell.value:
            headers[cell.column] = str(cell.value).lower()
    for col_idx, name in headers.items():
        if any(k in name for k in ["er", "sd", "return", "vol"]) or name.endswith("_m") or
name.endswith("_a"):
            for cell in ws[get_column_letter(col_idx)][header_row+1:]:
                if isinstance(cell.value, (int, float)):
                    cell.number_format = "0.00%"
```

That only catches column headers with "er", "sd", "return", "vol", etc.
But your pfs/pfslw DataFrames only have "Expected Return" and "Standard Deviation" columns (and later Sharpe). So if you add annualized ER/SD columns, those names ("ER_a", "SD_a") don't get picked up.

Fix

Use the improved formatter (which is already at the bottom of your file):

```
def _format_percent(ws, header_row=1, include_cols=None, exclude_cols=None):
    """
    Format selected columns as percentages.
    - If include_cols is provided, ONLY those exact columns are formatted.
    - Otherwise, we use a broad keyword heuristic.
    """
    headers = {}
    for cell in ws[header_row]:
        if cell.value:
            headers[cell.column] = str(cell.value).strip()

    def norm(s): return s.lower().strip()
    include_set = {norm(c) for c in include_cols} if include_cols else set()
    exclude_set = {norm(c) for c in exclude_cols} if exclude_cols else set()

    KEYWORDS = [
        "expected return", "return", "ret",
        "standard deviation", "stdev", "std dev", "vol", "volatility"
    ]

    for col_idx, name in headers.items():
        n = norm(name)
        if n in exclude_set:
            continue
        to_percent = (n in include_set) if include_set else any(k in n for k in KEYWORDS)
        if not to_percent:
            continue
        col_letter = get_column_letter(col_idx)
        for cell in ws[col_letter][header_row+1:]:
            if isinstance(cell.value, (int, float)):
                cell.number_format = "0.00%"
```

How to use it

When writing your metrics sheets, call with explicit includes:

```
_format_percent(ws_is, header_row=1,
                include_cols=["Expected Return","Standard Deviation","ER_a","SD_a"],
                exclude_cols=["Sharpe Ratio (Monthly)","Sharpe Ratio (Annualized)"])
```

This way:

Returns and SD (monthly + annualized) show as percent.

Sharpe Ratios stay as plain numbers with 3 decimals.

✅ So the fix is: either broaden your keyword list to cover "ER_a" and "SD_a", or explicitly pass include_cols when you call _format_percent. That will make all expected return and standard deviation numbers show up as percentages in Excel.

Do you want me to edit the builder so that it always treats any column ending in _a or _m as percent except Sharpe? That would cover both monthly and annualized stats without you having to pass include_cols every time.

Exhibit A-3 — Chat Session 5
Tool / Model: ChatGPT5
Date of usage: 9/30
Purpose: Code generation
Prompt: im tasked with creating te excel
Response: Perfect—since you're on the Excel deliverable, here's a drop-in "report builder" you can paste into your notebook (or a helper .py). It creates all required sheets, formats numbers, embeds your frontier plots, and freezes headers.

Copy this whole block first:

```
# ===== Excel Report Builder (FIN 684 Project 2) =====
from openpyxl import Workbook
from openpyxl.utils import get_column_letter
from openpyxl.styles import Font, Alignment, PatternFill
from openpyxl.drawing.image import Image as XLImage
import re
import pandas as pd
import numpy as np

def _autosize(ws, max_width=48):
    for col in ws.columns:
```

```
        col_letter = get_column_letter(col[0].column)
        width = 0
        for cell in col:
            text = str(cell.value) if cell.value is not None else ""
            width = max(width, len(text))
        ws.column_dimensions[col_letter].width = min(max(10, width + 2), max_width)

def _write_df(ws, df: pd.DataFrame, start_row=1, start_col=1, header=True, index_as_col=True):
    r = start_row
    c = start_col
    # header
    if header:
        if index_as_col:
            ws.cell(r, c).value = ""
            ws.cell(r, c).font = Font(bold=True)
            for j, col in enumerate(df.columns, start=c+1):
                ws.cell(r, j).value = str(col)
                ws.cell(r, j).font = Font(bold=True)
        else:
            for j, col in enumerate(df.columns, start=c):
                ws.cell(r, j).value = str(col)
                ws.cell(r, j).font = Font(bold=True)
        r += 1
    # rows
    for idx, row in df.iterrows():
        if index_as_col:
            ws.cell(r, c).value = str(idx)
            ws.cell(r, c).font = Font(bold=True)
            for j, col in enumerate(df.columns, start=c+1):
                val = row[col]
                ws.cell(r, j).value = float(val) if pd.api.types.is_number(val) else (None if pd.isna(val) else
str(val))
        else:
            for j, col in enumerate(df.columns, start=c):
                val = row[col]
                ws.cell(r, j).value = float(val) if pd.api.types.is_number(val) else (None if pd.isna(val) else
str(val))
        r += 1
    return r  # next empty row

def _format_percent(ws, header_row=1):
    # Apply percent format to columns that look like returns or stdevs
    # Detect by header text containing ER, SD, Return, Vol, or ending with _m/_a
    headers = {}
```

```
    for cell in ws[header_row]:
        if cell.value:
            headers[cell.column] = str(cell.value).lower()
    for col_idx, name in headers.items():
        if any(k in name for k in ["er", "sd", "return", "vol"]) or name.endswith("_m") or
name.endswith("_a"):
            for cell in ws[get_column_letter(col_idx)][header_row+1:]:
                if isinstance(cell.value, (int, float)):
                    cell.number_format = "0.00%"

def _format_ratio(ws, header_row=1):
    # Put 3 decimals on ratio-like columns (Sharpe)
    headers = {}
    for cell in ws[header_row]:
        if cell.value:
            headers[cell.column] = str(cell.value).lower()
    for col_idx, name in headers.items():
        if "sharpe" in name or "ratio" in name:
            for cell in ws[get_column_letter(col_idx)][header_row+1:]:
                if isinstance(cell.value, (int, float)):
                    cell.number_format = "0.000"

def _freeze(ws, row=2, col=2):
    ws.freeze_panes = ws.cell(row=row, column=col)

def _stripe(ws, header_row=1):
    fill = PatternFill(start_color="FFF5F5F5", end_color="FFF5F5F5", fill_type="solid")
    i = 0
    for r in range(header_row+1, ws.max_row+1):
        i += 1
        if i % 2 == 0:
            for c in range(1, ws.max_column+1):
                ws.cell(r, c).fill = fill

def build_excel_report(
    path_xlsx: str,
    inputs: dict,
    # metrics frames should already contain both monthly and annualized stats
    pfs: pd.DataFrame,      # sample-cov metrics (IS* and OS*)
    pfslw: pd.DataFrame,    # LW metrics (IS* and OS*)
    is_returns: pd.DataFrame,
    os_returns: pd.DataFrame,
    is_covrep: pd.DataFrame = None,
    os_covrep: pd.DataFrame = None,
```

```
    image_paths: list = None
):
    wb = Workbook()

    # Inputs
    ws = wb.active
    ws.title = "Inputs"
    ws["A1"].value = "Parameter"; ws["A1"].font = Font(bold=True)
    ws["B1"].value = "Value";     ws["B1"].font = Font(bold=True)
    r = 2
    for k, v in inputs.items():
        ws[f"A{r}"] = str(k)
        ws[f"B{r}"] = str(v)
        r += 1
    _autosize(ws)
    _freeze(ws, row=2, col=2)

    # IS Metrics (sample Σ)
    ws_is = wb.create_sheet("IS Metrics")
    _write_df(ws_is, pfs.loc[pfs.index.str.startswith("IS")], start_row=1, start_col=1, header=True,
index_as_col=True)
    _format_percent(ws_is, header_row=1)
    _format_ratio(ws_is, header_row=1)
    _stripe(ws_is); _autosize(ws_is); _freeze(ws_is)

    # OS Metrics (sample Σ)
    ws_os = wb.create_sheet("OS Metrics")
    _write_df(ws_os, pfs.loc[pfs.index.str.startswith("OS")], start_row=1, start_col=1, header=True,
index_as_col=True)
    _format_percent(ws_os, header_row=1)
    _format_ratio(ws_os, header_row=1)
    _stripe(ws_os); _autosize(ws_os); _freeze(ws_os)

    # IS–LW Metrics
    ws_is_lw = wb.create_sheet("IS–LW Metrics")
    _write_df(ws_is_lw, pfslw.loc[pfslw.index.str.startswith("IS")], start_row=1, start_col=1, header=True,
index_as_col=True)
    _format_percent(ws_is_lw, header_row=1)
    _format_ratio(ws_is_lw, header_row=1)
    _stripe(ws_is_lw); _autosize(ws_is_lw); _freeze(ws_is_lw)

    # OS–LW Metrics
    ws_os_lw = wb.create_sheet("OS–LW Metrics")
```

```
    _write_df(ws_os_lw, pfslw.loc[pfslw.index.str.startswith("OS")], start_row=1, start_col=1,
header=True, index_as_col=True)
    _format_percent(ws_os_lw, header_row=1)
    _format_ratio(ws_os_lw, header_row=1)
    _stripe(ws_os_lw); _autosize(ws_os_lw); _freeze(ws_os_lw)

    # Monthly Returns (IS)
    ws_ris = wb.create_sheet("Monthly Returns (IS)")
    df_is = is_returns.copy()
    df_is.index = pd.to_datetime(df_is.index)  # ensure datetime index
    df_out = df_is.sort_index()
    table = df_out.reset_index()
    table.rename(columns={"index": "Date"}, inplace=True)
    table["Date"] = table["Date"].dt.strftime("%Y-%m")
    _write_df(ws_ris, table, start_row=1, start_col=1, header=True, index_as_col=False)
    # percent format for all numeric cols
    for col in range(2, ws_ris.max_column+1):
        for cell in ws_ris[get_column_letter(col)][2:]:
            if isinstance(cell.value, (int, float)):
                cell.number_format = "0.00%"
    _autosize(ws_ris); _freeze(ws_ris)

    # Monthly Returns (OS)
    ws_ros = wb.create_sheet("Monthly Returns (OS)")
    df_os = os_returns.copy()
    df_os.index = pd.to_datetime(df_os.index)
    df_out = df_os.sort_index()
    table = df_out.reset_index()
    table.rename(columns={"index": "Date"}, inplace=True)
    table["Date"] = table["Date"].dt.strftime("%Y-%m")
    _write_df(ws_ros, table, start_row=1, start_col=1, header=True, index_as_col=False)
    for col in range(2, ws_ros.max_column+1):
        for cell in ws_ros[get_column_letter(col)][2:]:
            if isinstance(cell.value, (int, float)):
                cell.number_format = "0.00%"
    _autosize(ws_ros); _freeze(ws_ros)

    # Coverage sheets (optional)
    if is_covrep is not None:
        ws_cov_is = wb.create_sheet("Coverage (IS)")
        _write_df(ws_cov_is, is_covrep, start_row=1, start_col=1, header=True, index_as_col=True)
        _format_percent(ws_cov_is, header_row=1)
        _stripe(ws_cov_is); _autosize(ws_cov_is); _freeze(ws_cov_is)
```

```
    if os_covrep is not None:
        ws_cov_os = wb.create_sheet("Coverage (OS)")
        _write_df(ws_cov_os, os_covrep, start_row=1, start_col=1, header=True, index_as_col=True)
        _format_percent(ws_cov_os, header_row=1)
        _stripe(ws_cov_os); _autosize(ws_cov_os); _freeze(ws_cov_os)

    # Plots
    ws_plots = wb.create_sheet("Plots")
    cur_row = 1
    if image_paths:
        for p in image_paths:
            try:
                img = XLImage(p)
                img.anchor = f"A{cur_row}"
                ws_plots.add_image(img)
                # advance by image height in rows (rough estimate)
                advance = max(int(img.height / 18) + 2, 20)
                cur_row += advance
            except Exception as e:
                ws_plots.cell(cur_row, 1).value = f"Could not embed {p}: {e}"
                cur_row += 2
    _autosize(ws_plots)

    wb.save(path_xlsx)
    print(f"[Excel] Saved {path_xlsx}")
# ===== end builder =====
```

How to call it with your variables

Assuming your notebook has these names already:

pfs and pfslw (metrics for sample Σ and LW)

ret_matrix (IS returns, monthly)

os_ret_matrix (OS returns, monthly)

is_covrep, os_covrep (from coverage validation, optional)

four saved plot files:
is_sample_frontier.png, is_lw_frontier.png, os_sample_frontier.png, os_lw_frontier.png

Then run:

```
inputs = {
    "Tickers": ", ".join(list(ret_matrix.columns)),
    "IS Range": f"{ret_matrix.index.min():%Y-%m} – {ret_matrix.index.max():%Y-%m}",
    "OS Range": f"{os_ret_matrix.index.min():%Y-%m} – {os_ret_matrix.index.max():%Y-%m}",
    "Risk-free (monthly)": f"{rfrate:.6f}",
    "Long-only": str(False),          # set based on your run
    "Bounds (QP)": "[0, 1] or user-set"
}

image_paths = [
    "is_sample_frontier.png",
    "is_lw_frontier.png",
    "os_sample_frontier.png",
    "os_lw_frontier.png",
]

build_excel_report(
    path_xlsx="Project2_Report.xlsx",
    inputs=inputs,
    pfs=pfs,
    pfslw=pfslw,
    is_returns=ret_matrix,
    os_returns=os_ret_matrix,
    is_covrep=is_covrep,          # or None
    os_covrep=os_covrep,          # or None
    image_paths=image_paths
)
```

Quick checklist your grader will look for

Sheets for IS, OS, IS–LW, OS–LW with ER, SD, Sharpe in monthly and annual terms.

Returns tabs for IS and OS with dates in YYYY-MM and percent formatting.

Coverage tabs that show nonmissing counts and fractions (optional but nice).

Plots tab with all four images embedded.

Inputs tab that records tickers, date ranges, r_f, and key options.

Clear header row, frozen panes, readable column widths, alternating row stripes.

If your pfs and pfslw column names differ, you can still use the same builder. It auto-detects which columns should be percent vs ratio based on header text. If you want me to tailor formats to your exact column names, paste the pfs.head() and pfslw.head() and I will adjust the detection logic.