Optimal Asset Allocation for Risk Adjusted Investment Portfolios

Project Report

Master of Technology (M.Tech)

in

Industrial Mathematics and Scientific Computing

Submitted by

Charu Mittal - MA25M008
Ritika Verma - MA25M022

Submitted to

Department of Mathematics
Indian Institute of Technology, Madras
Chennai - 600036, India

November 2025

Contents

Abstract

Portfolio optimization is a fundamental problem in quantitative finance that seeks to construct an investment portfolio balancing the dual objectives of maximizing expected return and minimizing risk. In this project, Object-Oriented Programming (OOP) principles are leveraged to design a modular and extensible system, ensuring clarity, scalability, and maintainability.

Each financial instrument is represented as an object encapsulating its attributes and behavior, enabling clean abstraction and reusability across modules. Data acquisition, portfolio construction, and optimization are implemented as interchangeable components through factory and interface-based architectures. Mathematical optimization models, such as mean-variance or covariance shrinkage methods, are applied to determine the optimal allocation of capital across assets under practical constraints like investment limits and risk tolerance.

This architecture not only supports flexible experimentation with different data sources and algorithms but also promotes a transparent, data-driven framework for efficient asset allocation and risk management.

Chapter 1

Introduction

1.1 Background and Motivation

The rapid growth of financial data and computational capabilities has fundamentally reshaped how investment decisions are made. Modern investors have access to extensive data from diverse sources, including equities, bonds, and digital assets. While this data abundance enables deeper insights, it also introduces challenges such as noise, volatility, and the need for efficient processing.

Traditional portfolio management, which depends on manual judgment and static allocation rules, often struggles to adapt to dynamic market conditions. Human decision-making can be inconsistent and influenced by cognitive biases like overconfidence and loss aversion. As a result, there has been a growing shift toward automated and data-driven investment strategies that rely on objective, quantitative principles.

Algorithmic portfolio optimization provides a structured framework to balance risk and return through mathematical models such as mean-variance optimization and covariance shrinkage. With the rise of Python-based libraries and open financial APIs, it has become feasible to build scalable systems that integrate data collection, analytics, and visualization.

This project aims to leverage these advancements by creating a unified, modular platform that automates data handling, applies optimization algorithms, and provides interactive analysis tools—promoting transparent and reproducible decision-making in portfolio management.

1.2 Objective of the Project

The main objective of this project is to develop a Python-based, modular platform for portfolio management and optimization that bridges theory and practical application. The framework focuses on flexibility, automation, and clarity—making it suitable for both research and real-world financial analytics.

The specific objectives are as follows:

- **Automated Data Integration:** Fetch and preprocess financial data from APIs such as Yahoo Finance, Binance, and FRED, ensuring consistency across asset classes and time horizons.

- **Portfolio Optimization:** Implement classical and advanced optimization strategies, including mean-variance and covariance shrinkage methods, to derive efficient portfolios under defined constraints.
- **Performance Evaluation:** Analyze risk-return trade-offs using metrics like the Sharpe ratio, cumulative returns, and volatility decomposition.
- **Interactive Visualization:** Provide a Streamlit-based interface for intuitive, real-time portfolio construction, analysis, and performance comparison.

Through these objectives, the project contributes to the development of intelligent, adaptive, and data-driven financial decision-making tools that enhance portfolio robustness and transparency.

Chapter 2

Project Architecture and File Structure

2.1 Overall Directory Layout

Project Directory Structure (Part 1)

```
project_root/

 assets/
    asset_interface.py
    asset_factory.py
    asset_collection.py
    stock.py
    bond.py
    etf.py
    crypto.py

 data_fetcher/
    data_fetcher_interface.py
    data_factory.py
    yahoo_fetcher.py
    fred_fetcher.py
    binance_fetcher.py

 optimizer/
    optimizer_interface.py
    optimizer_factory.py
    mean_variance_optimizer.py
    covariance_optimizer.py
```

(Project Directory Structure continued...)

Project Directory Structure (Part 2)

```
portfolio/
   manager.py
   rebalance.py

portfolio_analyzer/
   analyzer_interface.py
   portfolio_analyzer.py
   return_calculator.py
   volatility_calculator.py

visualization/
   streamlit_dashboard.py

app.py
```

2.2 Component Interaction Flow (OOP-Based Design)

The system architecture is developed around key Object-Oriented Programming (OOP) principles — **abstraction**, **inheritance**, **encapsulation**, and **modularity**. Each layer is implemented through interfaces and concrete subclasses, promoting extensibility and code reusability.

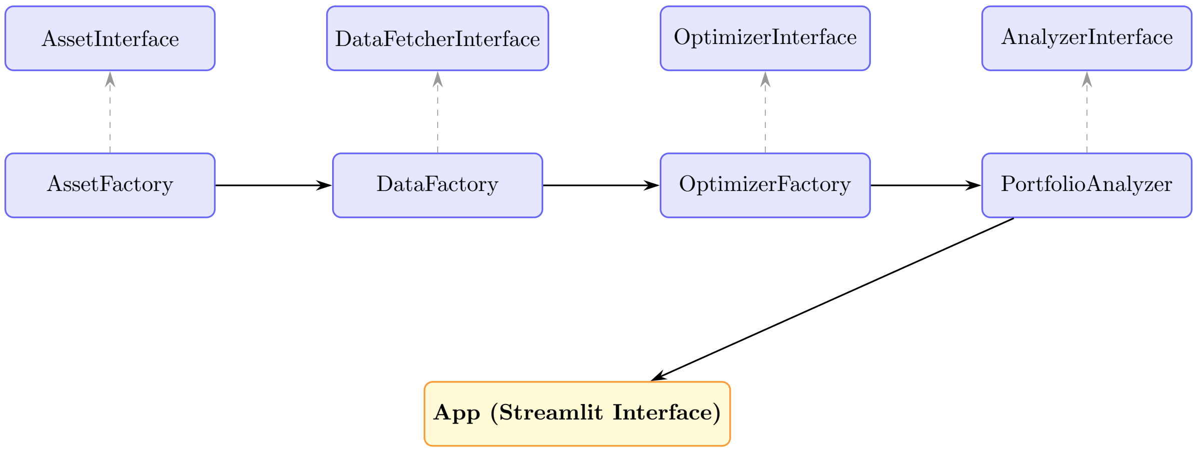


Figure 2.1: Component Interaction Flow with OOP Abstraction.

The OOP-based interaction can be summarized as follows:

- **Interfaces:** Define the abstract behavior for each module (e.g., `AssetInterface`, `DataFetcherInterface`, `OptimizerInterface`).
- **Factories:** Instantiate appropriate subclasses dynamically using the Factory design pattern (e.g., `AssetFactory` creates `Stock`, `Bond`, etc.).

- **Concrete Classes:** Implement specific functionality such as fetching data from Yahoo or optimizing portfolios using covariance shrinkage.
- **Application Layer:** The Streamlit-based `app.py` orchestrates all modules, integrates results, and provides interactive visualization.

Chapter 3

Module Overview and Functionality

3.1 Assets Module

The `assets` module defines the foundational entities representing different financial instruments that can be part of a portfolio, such as stocks, bonds, cryptocurrencies, and exchange-traded funds (ETFs). It follows an object-oriented design based on abstraction and modularity, allowing flexible extension and uniform access to asset metadata throughout the portfolio optimization system.

3.1.1 asset_interface.py

Objective: Defines the abstract interface that all asset classes must follow. It enforces a consistent structure through method contracts such as `get_symbol()`, `get_type()`, and `get_metadata()`, ensuring interoperability across the system.

asset_interface.py

```python
from abc import ABC, abstractmethod
from typing import Dict

class AssetInterface(ABC):
    """Abstract base class for all asset types."""

    @abstractmethod
    def get_symbol(self) -> str:
        """Return the ticker or unique symbol of the asset."""
        pass

    @abstractmethod
    def get_type(self) -> str:
        """Return the asset category (e.g., stock, bond,
          crypto, etc.)."""
        pass

    @abstractmethod
    def get_metadata(self) -> Dict:
        """Return asset-specific metadata as a dictionary."""
        pass
```

Purpose in Hierarchy: Acts as the blueprint for all asset classes, ensuring a standardized interface for the rest of the modules such as the `optimizer` and `portfolio` components.

3.1.2 stock.py

Objective: Implements the `Stock` class derived from `AssetInterface`. It encapsulates attributes like stock name, ticker symbol, and industry sector, representing traditional equity instruments.

stock.py

```python
from .asset_interface import AssetInterface
from typing import Dict, Optional

class Stock(AssetInterface):
    """Represents an equity asset such as a company stock."""

    def __init__(self, name: str, symbol: str, sector:
        Optional[str] = None):
        self.name = name
        self.symbol = symbol
        self.sector = sector

    def get_symbol(self) -> str:
        return self.symbol

    def get_type(self) -> str:
        return "stock"

    def get_metadata(self) -> Dict:
        return {"name": self.name, "sector": self.sector}
```

Purpose in Hierarchy: Provides the representation of stock-type assets, which are the most common inputs to the portfolio optimizer for mean-variance analysis.

3.1.3 bond.py

Objective: Defines the `Bond` class for fixed-income instruments. It captures attributes such as coupon rate and maturity period, which are essential for evaluating return stability and risk.

bond.py

```python
from .asset_interface import AssetInterface
from typing import Dict, Optional

class Bond(AssetInterface):
    """Represents a bond asset with coupon and maturity
       details."""

    def __init__(self, name: str, symbol: str,
                 coupon_rate: Optional[float] = None,
                 maturity_years: Optional[int] = None):
        self.name = name
        self.symbol = symbol
        self.coupon_rate = coupon_rate
        self.maturity_years = maturity_years

    def get_symbol(self) -> str:
        return self.symbol

    def get_type(self) -> str:
        return "bond"

    def get_metadata(self) -> Dict:
        return {
            "name": self.name,
            "coupon_rate": self.coupon_rate,
            "maturity_years": self.maturity_years
        }
```

Purpose in Hierarchy: Enables inclusion of fixed-income assets for diversified portfolio construction and risk mitigation in the optimization framework.

3.1.4 crypto.py

Objective: Implements the `Crypto` class, representing digital assets such as Bitcoin or Ethereum. It includes parameters like exchange name to manage cross-market data consistency.

crypto.py

```python
from .asset_interface import AssetInterface
from typing import Dict

class Crypto(AssetInterface):
    """Represents a cryptocurrency asset."""

    def __init__(self, name: str, symbol: str, exchange: str =
        "Binance"):
        self.name = name
        self.symbol = symbol
        self.exchange = exchange

    def get_symbol(self) -> str:
        return self.symbol

    def get_type(self) -> str:
        return "crypto"

    def get_metadata(self) -> Dict:
        return {"name": self.name, "exchange": self.exchange}
```

Purpose in Hierarchy: Facilitates integration of cryptocurrency data sources and risk modeling for emerging digital assets within the unified asset architecture.

3.1.5 etf.py

Objective: Defines the `ETF` class representing Exchange-Traded Funds. It captures categorical information about the ETF, such as its investment category or index type.

etf.py

```python
from .asset_interface import AssetInterface
from typing import Dict, Optional

class ETF(AssetInterface):
    """Represents an Exchange-Traded Fund (ETF) asset."""

    def __init__(self, name: str, symbol: str, category:
        Optional[str] = None):
        self.name = name
        self.symbol = symbol
        self.category = category

    def get_symbol(self) -> str:
        return self.symbol

    def get_type(self) -> str:
        return "etf"

    def get_metadata(self) -> Dict:
        return {"name": self.name, "category": self.category}
```

Purpose in Hierarchy: Supports index-based or thematic investment assets to achieve broader diversification in the portfolio optimizer.

3.1.6 asset_factory.py

Objective: Implements a factory pattern for dynamic creation of asset objects. The `AssetFactory` class abstracts away class-specific instantiation, allowing creation of any asset type based on a single call.

asset_factory.py

```python
from .stock import Stock
from .bond import Bond
from .crypto import Crypto
from .etf import ETF

class AssetFactory:
    """Factory for creating different asset instances
        dynamically."""

    @staticmethod
    def create(asset_type: str, name: str, symbol: str,
        **kwargs):
        asset_type = asset_type.lower()
        if asset_type == "stock":
            return Stock(name, symbol, kwargs.get("sector"))
        elif asset_type == "bond":
            return Bond(name, symbol,
                kwargs.get("coupon_rate"),
                kwargs.get("maturity_years"))
        elif asset_type == "crypto":
            return Crypto(name, symbol, kwargs.get("exchange",
                "Binance"))
        elif asset_type == "etf":
            return ETF(name, symbol, kwargs.get("category"))
        else:
            raise ValueError(f"Unsupported asset type:
                {asset_type}")
```

Purpose in Hierarchy: Acts as a centralized constructor for assets, promoting modular scalability. It is used by higher-level modules like `data_fetcher` and `portfolio` to dynamically register asset classes without hardcoding dependencies.

3.1.7 asset_collection.py

Objective: Defines a lightweight container class for managing multiple asset instances. It offers methods to add, retrieve, and describe assets, maintaining a cohesive collection.

asset_collection.py

```python
from typing import List, Dict
from .asset_interface import AssetInterface

class AssetCollection:
    """Container for managing multiple asset objects."""

    def __init__(self):
        self.assets: List[AssetInterface] = []

    def add_asset(self, asset: AssetInterface) -> None:
        self.assets.append(asset)

    def get_assets(self) -> List[AssetInterface]:
        return list(self.assets)

    def get_symbols(self) -> List[str]:
        return [asset.get_symbol() for asset in self.assets]

    def describe(self) -> List[Dict]:
        """Return a summary of all assets in the collection."""
        return [
            {
                "symbol": asset.get_symbol(),
                "type": asset.get_type(),
                **asset.get_metadata(),
            }
            for asset in self.assets
        ]
```

Purpose in Hierarchy: Provides structured management of all asset instances in memory. The collection is later utilized by the portfolio construction and optimization modules to access asset metadata and market data efficiently.

OOPS concepts used:

OOP Concepts and Design Principles Used

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Modularity
- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Factory Design Pattern

3.2 Data Fetcher Module

The `data_fetcher` module is responsible for collecting historical financial data from multiple online data sources such as Yahoo Finance, Binance, and the Federal Reserve Economic Data (FRED). This modular structure allows the system to flexibly handle different asset classes — including stocks, bonds, and cryptocurrencies — using a uniform data fetching interface.

By abstracting the fetching logic behind a common interface, the system remains modular, scalable, and easy to extend when introducing new asset classes or data providers.

3.2.1 YahooFetcher (yahoo_fetcher.py)

The `YahooFetcher` class fetches stock and ETF data from Yahoo Finance using the `yfinance` API. It automates the process of downloading historical price data over a defined date range and ensures that the data is clean and formatted consistently.

Handling Multi-Index DataFrames and Symbol Alignment: Yahoo Finance may return MultiIndex DataFrames, especially when fetching multiple symbols. To handle this, the fetcher flattens the columns, aligns symbols properly, and extracts the `Close` price for the given symbol. This ensures compatibility with portfolio computations that rely on properly indexed time series.

yahoo_fetcher.py

```python
import yfinance as yf
import pandas as pd
from .data_fetcher_interface import DataFetcherInterface

class YahooFetcher(DataFetcherInterface):
    def fetch_data(self, symbol: str, start_date: str,
        end_date: str) -> pd.Series:
        df = yf.download(symbol, start=start_date,
            end=end_date, progress=False)
        if df.empty:
            raise ValueError(f"No data returned for {symbol}
                from Yahoo Finance.")

        # Flatten MultiIndex columns
        df.columns = ['_'.join(col).strip() if isinstance(col,
            tuple) else col for col in df.columns]

        # Extract closing price for the symbol
        col_name = f'Close_{symbol}'
        if col_name not in df.columns:
            raise ValueError(f"Column {col_name} not found in
                downloaded data")

        series = df[col_name].dropna().rename(symbol)
        return series
```

Purpose in hierarchy: The `YahooFetcher` acts as the main data provider for stocks and ETFs. It ensures consistent, symbol-aligned datasets that are ready for downstream portfolio analysis and optimization workflows.

3.2.2 BinanceFetcher (binance_fetcher.py)

The `BinanceFetcher` retrieves cryptocurrency price data using the `yfinance` API. It focuses on digital assets such as Bitcoin and Ethereum, represented in symbols like `BTC-USD` and `ETH-USD`.

It attempts to use the adjusted closing price (if available) to ensure accuracy and raises meaningful errors if data retrieval fails.

binance_fetcher.py

```python
import yfinance as yf
import pandas as pd
from .data_fetcher_interface import DataFetcherInterface

class BinanceFetcher(DataFetcherInterface):
    def fetch_data(self, symbol: str, start_date: str,
        end_date: str) -> pd.Series:
        try:
            df = yf.download(symbol, start=start_date,
                end=end_date, progress=False)
            if not df.empty:
                s = df['Adj Close'] if 'Adj Close' in
                    df.columns else df['Close']
                return s.dropna().rename(symbol)
        except Exception:
            pass
        raise ValueError(f"Unable to fetch crypto data for
            {symbol}. Try BTC-USD, ETH-USD, etc.")
```

Purpose in hierarchy: The `BinanceFetcher` extends the data fetching system to support cryptocurrencies, enabling portfolio diversification into digital assets. It adheres to the same interface as other fetchers, ensuring smooth integration into the broader data pipeline.

3.2.3 FredFetcher (fred_fetcher.py)

The `FredFetcher` obtains economic and financial time series data from the FRED (Federal Reserve Economic Data) database. It uses the `pandas_datareader` library to fetch information like interest rates, inflation data, or bond yields, which are essential for fixed-income portfolio construction.

fred_fetcher.py

```python
from .data_fetcher_interface import DataFetcherInterface
from pandas_datareader import data as pdr

class FredFetcher(DataFetcherInterface):
    def fetch_data(self, symbol: str, start_date: str,
        end_date: str):
        df = pdr.DataReader(symbol, 'fred', start_date,
            end_date)
        if df.empty:
            raise ValueError(f"No FRED data for {symbol}")
        return df.iloc[:, 0].dropna().rename(symbol)
```

Purpose in hierarchy: The `FredFetcher` introduces access to macroeconomic and bond-related data, enriching the model's capability to handle multiple asset categories and factor-based analysis.

3.2.4 DataFetcherInterface Implementation

The `DataFetcherInterface` defines the common structure that every fetcher class must follow. It enforces the implementation of the `fetch_data()` method, ensuring that all data fetchers return data in a standardized format — a `pandas.Series` indexed by time.

data_fetcher_interface.py

```python
from abc import ABC, abstractmethod
import pandas as pd

class DataFetcherInterface(ABC):
    @abstractmethod
    def fetch_data(self, symbol: str, start_date: str,
        end_date: str) -> pd.Series:
        """Fetch price data for a symbol."""
        pass
```

Purpose in hierarchy: By defining this abstract base class, the system enforces uniformity among different fetchers, improving maintainability and modular design. New data providers can be easily added by extending this interface.

3.2.5 DataFetcherFactory (data_factory.py)

The `DataFetcherFactory` serves as a centralized component that determines which data fetcher to use based on the asset type. It abstracts away the logic of selecting between `YahooFetcher`, `BinanceFetcher`, and `FredFetcher`, ensuring that the rest of the system does not need to handle this decision-making process.

This approach follows the Factory Design Pattern, promoting clean separation between asset classification and data retrieval.

data_factory.py

```python
from .yahoo_fetcher import YahooFetcher
from .binance_fetcher import BinanceFetcher
from .fred_fetcher import FredFetcher

class DataFetcherFactory:
    @staticmethod
    def get_fetcher_for_asset_type(asset_type: str):
        asset_type = asset_type.lower()  # force lowercase
        if asset_type == 'stock' or asset_type == 'etf':
            return YahooFetcher()
        elif asset_type == 'crypto':
            return BinanceFetcher()
        elif asset_type == 'bond':
            return FredFetcher()
        else:
            return YahooFetcher()
```

Purpose in hierarchy: The `DataFetcherFactory` provides a clean interface for other system components (like portfolio construction or optimization modules) to obtain the appropriate data fetcher. This promotes flexibility and scalability, enabling the addition of new data sources without modifying existing logic.

OOPS concepts used:

OOP Concepts and Design Principles Used

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Modularity
- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Factory Design Pattern

3.3 Optimizer Module

The `optimizer` module forms the computational core of the portfolio management system. It implements different optimization strategies that determine the best allocation of capital across available assets based on risk and return trade-offs. The design follows an abstraction-oriented architecture, ensuring that multiple optimization techniques can coexist and be seamlessly extended in the future.

3.3.1 optimizer_interface.py

Objective: This file defines a formal abstraction for all optimizers through the `OptimizerInterface`. It specifies the contract that every optimizer must fulfill — an `optimize()` method accepting expected returns and the covariance matrix. This ensures that any optimizer (analytical or numerical) can be plugged into the system without altering the higher-level workflow.

Relevant Code Snippet:

optimizer_interface.py

```python
from abc import ABC, abstractmethod
import pandas as pd

class OptimizerInterface(ABC):
    """
    Abstract base class for portfolio optimizers.
    """

    @abstractmethod
    def optimize(self, expected_returns: pd.Series, cov_matrix:
        pd.DataFrame) -> pd.Series:
        """
        Optimize the portfolio based on expected returns and
            covariance matrix.

        Parameters
        ----------
        expected_returns : pd.Series
            Expected annualized returns for each asset.
        cov_matrix : pd.DataFrame
            Annualized covariance matrix of asset returns.

        Returns
        -------
        pd.Series
            Optimal portfolio weights (sum to 1).
        """
        pass
```

Purpose in Hierarchy: Serves as the parent abstraction layer for all optimization methods. By depending on this interface, higher-level modules like `portfolio` and `streamlit_app` can switch optimizers dynamically without code modification, adhering to SOLID design principles.

3.3.2 mean_variance_optimizer.py

Objective: Implements the **Mean-Variance Optimization (MVO)** strategy proposed by Harry Markowitz. This optimizer minimizes the portfolio's volatility while enforcing constraints such as weight bounds and total allocation. An optional target return can also be specified to trace the efficient frontier.

Relevant Code Snippet:

mean_variance_optimizer.py

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

class MeanVarianceOptimizer(OptimizerInterface):
    def optimize(self, expected_returns: pd.Series, cov_matrix:
        pd.DataFrame, target_return: float | None = None):
        n = len(expected_returns)
        mu = expected_returns.values
        Sigma = cov_matrix.values

        def port_vol(w):
            return np.sqrt(w.T @ Sigma @ w)

        x0 = np.ones(n) / n
        bounds = tuple((0.0, 0.7) for _ in range(n))
        cons = ({'type': 'eq', 'fun': lambda w: np.sum(w) -
            1.0},)

        if target_return is not None:
            cons = (
                {'type': 'eq', 'fun': lambda w: np.sum(w) -
                    1.0},
                {'type': 'eq', 'fun': lambda w: w @ mu -
                    target_return}
            )

        res = minimize(lambda w: port_vol(w), x0,
            method='trust-constr', bounds=bounds,
            constraints=cons)
        if not res.success:
            raise RuntimeError('Optimization failed: ' +
                str(res.message))

        weights = res.x
        return pd.Series(weights, index=expected_returns.index)
```

Purpose in Hierarchy: Acts as the primary numerical optimization engine. It provides a mathematically rigorous way to derive optimal portfolio weights based on covariance structure and desired return targets, forming the backbone for efficient frontier visualization.

3.3.3 covariance_optimizer.py

Objective: Implements a simplified analytical optimizer based on the inverse covariance matrix. Instead of iterative numerical methods, this optimizer directly allocates weights proportional to risk-adjusted returns using the formula:

$$w \propto \Sigma^{-1}\mu$$

This provides a quick, closed-form approximation to mean-variance optimization.

Relevant Code Snippet:

covariance_optimizer.py

```python
import numpy as np
import pandas as pd
from .optimizer_interface import OptimizerInterface

class CovarianceOptimizer(OptimizerInterface):
    """
    Analytical optimizer based on inverse covariance weighting.
    """

    def optimize(self, expected_returns: pd.Series, cov_matrix:
        pd.DataFrame) -> pd.Series:
        Sigma = cov_matrix.values
        mu = expected_returns.values

        inv_cov = np.linalg.pinv(Sigma)
        raw_weights = inv_cov @ mu

        weights = raw_weights / np.sum(raw_weights)

        return pd.Series(weights, index=expected_returns.index,
            name="weights")
```

Purpose in Hierarchy: Provides a fast analytical alternative to the Markowitz optimizer, ideal for exploratory analysis and cases where computational efficiency is critical. It also serves as a benchmark model for comparing performance across different optimization strategies.

3.3.4 optimizer_factory.py

Objective: Defines a centralized factory class that returns the appropriate optimizer instance based on the user's choice. This eliminates the need for conditional logic across the project and makes adding new optimization techniques seamless.

Relevant Code Snippet:

optimizer_factory.py

```python
from .mean_variance_optimizer import MeanVarianceOptimizer
from .covariance_optimizer import CovarianceOptimizer

class OptimizerFactory:
    """
    Factory for creating optimizer instances.
    """

    @staticmethod
    def get(method: str):
        method = method.lower()
        if method == "mean_variance":
            return MeanVarianceOptimizer()
        elif method == "covariance":
            return CovarianceOptimizer()
        else:
            raise ValueError(f"Unknown optimizer method:
                {method}")
```

Purpose in Hierarchy: Acts as the decision layer that abstracts away the creation of optimizer objects. It upholds the Open/Closed Principle by allowing new optimizers (like `Black-Litterman` or `Hierarchical Risk Parity`) to be integrated without modifying existing code, thus ensuring modular scalability.

OOPS concepts used:

OOP Concepts and Design Principles Used

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Modularity
- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Factory Design Pattern

3.4 Portfolio Module

The `portfolio` module serves as the core orchestrator of the entire portfolio optimization process. It bridges the gap between data fetching, asset creation, optimization, and analysis. By managing all major interactions among components, this module ensures that the system operates cohesively and efficiently.

3.4.1 Portfolio Manager (manager.py)

The `PortfolioManager` class acts as the central controller that coordinates every stage of portfolio construction and evaluation. It takes advantage of the modular design established in other parts of the project—such as the asset, data fetcher, optimizer, and analyzer modules—and ties them together in a unified workflow.

This class is designed to:

- Build a collection of assets based on provided specifications.
- Fetch and align price data from multiple sources.
- Compute statistical measures like expected returns and covariance matrices.
- Run optimization routines to determine optimal portfolio weights.
- Perform post-optimization analysis of portfolio performance.

manager.py (Part 1)

```python
import pandas as pd
import numpy as np
from typing import List, Dict, Optional

class PortfolioManager:
    """
    Orchestration class for building asset collections, fetching prices,
    computing returns/covariance, and running optimization & analysis.
    """

    def __init__(self, asset_factory, data_factory, optimizer_factory, analyzer):
        self.asset_factory = asset_factory
        self.data_factory = data_factory
        self.optimizer_factory = optimizer_factory
        self.analyzer = analyzer

    def build_collection_from_specs(self, specs: List[Dict]) -> "AssetCollection":
        from assets.asset_collection import AssetCollection
        collection = AssetCollection()
        for s in specs:
            asset = self.asset_factory.create(
                s.get("asset_type") or s.get("type") or s.get("asset"),
                s.get("name"),
                s.get("symbol"),
                **{k: v for k, v in s.items() if k not in ("asset_type", "name", "symbol")}
            )
            collection.add_asset(asset)
        return collection
```

(Code continued on next page)

manager.py (Part 2)

```python
    def fetch_prices(self, asset_collection, start_date: str, end_date: str) -> pd.DataFrame:
        series_list = []
        for asset in asset_collection.get_assets():
            fetcher = self.data_factory.get_fetcher_for_asset_type
            (asset.get_type())
            symbol = asset.get_symbol().strip()
            print(f"Fetching {symbol} ({asset.get_type()}) using {fetcher.__class__.__name__}...")
            try:
                s = fetcher.fetch_data(symbol, start_date, end_date)
                if not s.empty:
                    series_list.append(s)
            except Exception as e:
                print(f"WARNING: failed to fetch {symbol}: {e}")

        if not series_list:
            raise RuntimeError("No price series fetched for any asset.")
        price_df = pd.concat(series_list, axis=1, join='outer').ffill().bfill()
        return price_df

    def compute_expected_returns_covariance(self, price_df: pd.DataFrame):
        daily_returns = price_df.pct_change().dropna()
        expected_returns = daily_returns.mean() * 252
        covariance = daily_returns.cov() * 252
        covariance += np.eye(len(covariance)) * 1e-6
        return expected_returns, covariance, daily_returns

    def optimize(self, expected_returns, covariance, method: Optional[str] = "mean_variance", target_return: Optional[float] = None):
        optimizer = self.optimizer_factory.get(method)
        try:
            weights = optimizer.optimize(expected_returns, covariance, target_return)
        except TypeError:
            weights = optimizer.optimize(expected_returns, covariance)
        except RuntimeError as e:
            print(f"Optimization failed: {e}. Using equal weights as fallback.")
            weights = pd.Series(1 / len(expected_returns), index=expected_returns.index)
        return weights / weights.sum()

    def analyze_portfolio(self, price_df: pd.DataFrame, weights):
        return self.analyzer.analyze(price_df, weights)
```

Purpose in hierarchy: The `PortfolioManager` represents the integration layer of the entire system. It brings together asset modeling, data fetching, statistical computation, and optimization into a single workflow. By centralizing these operations, it simplifies experimentation, testing, and real-time analysis, allowing for flexible portfolio strategies and clear separation of responsibilities among different modules. This class is key to transforming raw financial data into actionable insights for portfolio optimization and performance evaluation.

OOPS concepts used:

OOP Concepts and Design Principles Used

- Encapsulation
- Modularity
- Abstraction
- Single Responsibility Principle (SRP)
- Dependency Inversion Principle (DIP)

3.5 Portfolio Analyzer

The **Portfolio Analyzer** module acts as the evaluation layer in the overall system. While the optimizer determines the best possible asset weights, this module quantifies how well those weights perform. It computes key performance indicators such as portfolio returns, volatility, and the Sharpe ratio—offering a balanced understanding of profitability and risk.

3.5.1 Analyzer_Interface.py

Meaningful Explanation

This file defines an abstract interface that every analyzer must implement. It establishes a clear contract for analyzing portfolios based on asset prices and corresponding weights. By enforcing this structure, it ensures modularity, extensibility, and compliance with object-oriented design principles, particularly the *Interface Segregation Principle.*

Relevant Code Snippet:

analyzer_interface.py

```python
from abc import ABC, abstractmethod
import pandas as pd

class AnalyzerInterface(ABC):
    """
    Abstract base class for all portfolio analysis modules.
    """

    @abstractmethod
    def analyze(self, price_data: pd.DataFrame, weights:
        pd.Series) -> dict:
        """
        Analyze the portfolio given asset prices and weights.
        Returns metrics like returns, volatility, and Sharpe
            ratio.
        """
        pass
```

Purpose in Hierarchy

Analyzer_Interface.py serves as the foundational layer in the analyzer hierarchy. All concrete analyzer implementations (e.g., PortfolioAnalyzer) inherit from this interface, guaranteeing structural consistency across different analytical modules.

3.5.2 Portfolio_Analyzer.py

Meaningful Explanation

This file defines the **PortfolioAnalyzer** class, the central engine of performance evaluation. It integrates multiple components—return calculation, volatility computation, and risk-adjusted performance metrics—to offer a comprehensive assessment of portfolio efficiency. By modularizing analytical steps, it ensures scalability and ease of debugging.

Relevant Code Snippet:

portfolio_analyzer.py

```python
import numpy as np
import pandas as pd
from portfolio_analyzer.analyzer_interface import
    AnalyzerInterface
from portfolio_analyzer.return_calculator import
    ReturnCalculator
from portfolio_analyzer.volatility_calculator import
    VolatilityCalculator

class PortfolioAnalyzer(AnalyzerInterface):
    def __init__(self, risk_free_rate: float = 0.02):
        self.risk_free_rate = risk_free_rate
        self.return_calculator = ReturnCalculator()
        self.volatility_calculator = VolatilityCalculator()

    def analyze(self, price_data: pd.DataFrame, weights:
        pd.Series) -> dict:
        daily_returns =
            self.return_calculator.calculate_daily_returns
        (price_data)
        portfolio_returns =
            self.return_calculator.calculate_portfolio_return
        (daily_returns, weights)
        cumulative_return =
            self.return_calculator.calculate_cumulative_return
        (portfolio_returns)
        portfolio_volatility =
            self.volatility_calculator.calculate_portfolio_
        volatility(daily_returns, weights)
        sharpe_ratio =
            self._calculate_sharpe_ratio(portfolio_returns,
            portfolio_volatility)

        return {
            "Cumulative Return": round(cumulative_return * 100,
                2),
            "Portfolio Volatility": round(portfolio_volatility
                * np.sqrt(252) * 100, 2),
            "Sharpe Ratio": round(sharpe_ratio, 3)
        }
```

Purpose in Hierarchy

Portfolio_Analyzer.py functions as the core executor within the analyzer module. It coordinates subcomponents such as the *ReturnCalculator* and *VolatilityCalculator* to compute complete portfolio metrics. This structure embodies separation of concerns and ensures analytical precision.

3.5.3 Return_Calculator.py

Meaningful Explanation

The **ReturnCalculator** handles all return-related computations. It converts raw price data into meaningful insights such as daily percentage changes, cumulative returns, and weighted portfolio returns. This modular design allows analysts to reuse return calculations across different evaluation or optimization pipelines.

Relevant Code Snippet:

return_calculator.py

```python
import pandas as pd

class ReturnCalculator:
    """
    Calculates asset-level and portfolio-level returns.
    """

    def calculate_daily_returns(self, price_data: pd.DataFrame)
       -> pd.DataFrame:
        return price_data.pct_change().dropna()

    def calculate_portfolio_return(self, daily_returns:
       pd.DataFrame, weights: pd.Series) -> pd.Series:
        return (daily_returns * weights).sum(axis=1)

    def calculate_cumulative_return(self, portfolio_returns:
       pd.Series) -> float:
        return (1 + portfolio_returns).prod() - 1
```

Purpose in Hierarchy

Return_Calculator.py serves as a utility submodule under the *PortfolioAnalyzer*. It isolates the logic for return computation, ensuring that the main analyzer can focus purely on interpretation and metric integration.

3.5.4 Volatility_Calculator.py

Meaningful Explanation

The **VolatilityCalculator** quantifies portfolio and asset-level risk. It computes the volatility using covariance matrices and standard deviation. This measurement is critical for understanding the uncertainty in portfolio returns and for computing risk-adjusted metrics like the Sharpe ratio.

Relevant Code Snippet:

volatility_calculator.py

```python
import numpy as np
import pandas as pd

class VolatilityCalculator:
    """
    Computes portfolio volatility and related metrics.
    """

    def calculate_portfolio_volatility(self, daily_returns:
        pd.DataFrame, weights: pd.Series) -> float:
        cov_matrix = daily_returns.cov()
        return np.sqrt(weights.T @ cov_matrix @ weights)

    def calculate_asset_volatility(self, daily_returns:
        pd.DataFrame) -> pd.Series:
        return daily_returns.std() * np.sqrt(252)
```

Purpose in Hierarchy

Volatility_Calculator.py complements the *ReturnCalculator* by addressing the risk dimension of portfolio performance. Together, these two calculators provide the foundation for balanced financial analysis, enabling the *PortfolioAnalyzer* to compute a reliable Sharpe ratio and assess overall stability.

OOPS concepts used:

OOP Concepts and Design Principles Used

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Modularity
- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)

Chapter 4

Optimization Algorithms

4.1 Mean-Variance Optimization (Markowitz Model)

4.1.1 Introduction

The Mean-Variance Optimization (MVO) model, proposed by Harry Markowitz in 1952, forms the cornerstone of modern portfolio theory. It provides a systematic approach to balancing risk and return when constructing an investment portfolio. The primary goal is to determine optimal asset weights that either:

- Minimize portfolio risk (variance) for a given expected return, or
- Maximize expected return for a given level of risk.

Risk is quantified as the variance (or standard deviation) of portfolio returns, while expected return is modeled as the weighted sum of the expected returns of individual assets.

4.1.2 Theory of the Algorithm

Let there be n assets with:

- Expected return vector $\mu = [\mu_1, \mu_2, ..., \mu_n]^T$
- Covariance matrix of asset returns Σ
- Portfolio weights $w = [w_1, w_2, ..., w_n]^T$

The portfolio's expected return and variance are given by:

Portfolio Statistics

$$R_p = w^T \mu, \quad \sigma_p^2 = w^T \Sigma w$$

The Mean-Variance Optimization problem can be expressed as a **Quadratic Programming (QP)** problem:

Optimization Problem Formulation

$$\begin{aligned} \min_{w} \quad & w^T \Sigma w \\ \text{s.t.} \quad & w^T \mu = R_t, \\ & \sum_i w_i = 1, \\ & w_i \geq 0 \quad \forall i \end{aligned}$$

Here:

- $w^T \Sigma w$ represents portfolio variance (total risk),
- $w^T \mu = R_t$ enforces a target expected return,
- $\sum_i w_i = 1$ ensures full investment,
- $w_i \geq 0$ prevents short selling (long-only constraint).

This model identifies the *efficient frontier* — a curve representing the set of optimal portfolios that offer the highest return for a defined level of risk.

4.1.3 Step-by-Step Algorithm

1. **Input Data:** Collect historical price data for n assets and compute periodic returns.
2. **Compute Statistical Measures:** Derive the expected return vector μ and covariance matrix Σ of returns.
3. **Formulate the Optimization Problem:** Define the mean-variance objective and constraints as described above.
4. **Solve Using Quadratic Programming:** Use optimization solvers like `cvxopt` or `scipy.optimize.minimize` to compute optimal weights w^*.
5. **Evaluate and Visualize:** Compute key metrics — expected return, volatility, and Sharpe ratio — and visualize the efficient frontier.

4.1.4 Time Complexity Analysis

The MVO algorithm involves matrix computations and solving a constrained quadratic optimization problem.

Computational Complexity

- Covariance matrix computation: $\mathcal{O}(n^2 T)$, where T = number of time observations.
- Quadratic optimization solving: $\mathcal{O}(n^3)$ (for interior-point or Newton-based solvers).

Overall Complexity: $\mathcal{O}(n^3)$

Thus, MVO is computationally efficient for portfolios with a moderate number of assets but can become expensive for very large asset universes.

4.1.5 Implementation Insights in Our Project

In our system, the Mean-Variance Optimization was implemented under the file `mean_variance_optimizer.py`. The design follows a modular, object-oriented approach, making it easy to integrate with the broader portfolio management framework.

- Expected returns and covariance matrices are computed from market data fetched via the `YahooFetcher` class.
- The optimization is solved using `scipy.optimize.minimize`, ensuring numerical stability and proper constraint handling.
- The system supports both:
  - **Long-only portfolios:** $w_i \geq 0$
  - **Short-allowed portfolios:** w_i unconstrained
- The resulting optimal weights are passed to the `PortfolioAnalyzer` for risk-return analysis and visualization.

Relevant Code Snippet:

mean_variance_optimizer.py

```python
import numpy as np
from scipy.optimize import minimize

class MeanVarianceOptimizer:
    """
    Implements the classic Markowitz Mean-Variance Optimization.
    """

    def optimize(self, mu, Sigma, R_target, allow_short=False):
        n = len(mu)
        init_w = np.ones(n) / n

        constraints = [
            {'type': 'eq', 'fun': lambda w: np.sum(w) - 1},
            {'type': 'eq', 'fun': lambda w: np.dot(w, mu) -
                R_target}
        ]
        bounds = None if allow_short else [(0, 1) for _ in
           range(n)]

        result = minimize(
            lambda w: np.dot(w.T, np.dot(Sigma, w)),
            init_w,
            constraints=constraints,
            bounds=bounds
        )
        return result.x
```

4.2 Covariance Shrinkage Optimization

Covariance shrinkage optimization is a regularization technique designed to improve the stability of the sample covariance matrix, which is often ill-conditioned or singular when the number of assets is comparable to or exceeds the number of historical observations. The goal is to combine the sample covariance with a structured target (such as a diagonal matrix or identity matrix) to produce a more reliable estimate for optimization.

4.2.1 Theory of Algorithm

Let Σ denote the true covariance matrix, and $\hat{\Sigma}$ denote the sample covariance matrix computed from historical returns.

We define the **shrinkage estimator** as a convex combination of $\hat{\Sigma}$ and a target matrix T:

mean_variance_optimizer.py

$$\Sigma_{\text{shrink}} = \lambda T + (1 - \lambda)\hat{\Sigma}$$

where:

- T is a structured target matrix (e.g., identity, constant correlation, or diagonal matrix).
- $\lambda \in [0, 1]$ is the shrinkage intensity controlling the trade-off between bias and variance.

To minimize the mean squared error between the estimator and the true covariance, we solve:

mean_variance_optimizer.py

$$\lambda^* = \arg \min_{\lambda \in [0,1]} \mathbb{E}\left[\|\Sigma_{\text{shrink}} - \Sigma\|_F^2\right]$$

Here, $\|\cdot\|_F$ denotes the Frobenius norm.

The optimized covariance matrix is then used in the **Mean-Variance Optimization** framework to find the optimal weights:

mean_variance_optimizer.py

$$\min_{w} \ w^T \Sigma_{\text{shrink}} w \quad \text{subject to} \quad \begin{cases} \mu^T w = \mu_{\text{target}} \\ \mathbf{1}^T w = 1 \end{cases}$$

where w represents portfolio weights and μ_{target} is the desired portfolio return.

4.2.2 Algorithm

1. Compute daily returns from historical price data.
2. Calculate the sample covariance matrix $\hat{\Sigma}$.
3. Define a structured target covariance matrix T (e.g., diagonal of $\hat{\Sigma}$).
4. Estimate the optimal shrinkage intensity λ^* using Ledoit–Wolf or Oracle approximations.
5. Construct the shrinkage covariance matrix:

$$\Sigma_{\text{shrink}} = \lambda^* T + (1 - \lambda^*)\hat{\Sigma}$$

6. Use Σ_{shrink} in optimization to compute optimal portfolio weights.

4.2.3 Time and Space Complexity

Computational Complexity

- **Covariance computation:** $O(n^2 T)$, where n is the number of assets and T is the number of time samples.
- **Shrinkage and optimization:** $O(n^3)$ (dominated by matrix inversion).
- **Overall space complexity:** $O(n^2)$ for storing the covariance matrices.

4.2.4 Implementation Insights in Our Project

In our project, the covariance shrinkage method was implemented within the `optimizer` module under the `cov_shrinkage.py` file. The implementation uses the Ledoit–Wolf shrinkage approach from `sklearn.covariance.LedoitWolf` to estimate λ^* automatically and integrate it into the portfolio optimization pipeline.

The adjusted covariance matrix is seamlessly passed to the optimization routines defined in the `optimizer_factory`, ensuring stability even with limited or noisy data. This integration significantly improves the robustness of the optimizer compared to using a raw sample covariance matrix.

mean_variance_optimizer.py

```python
import numpy as np
import pandas as pd
from sklearn.covariance import LedoitWolf

class CovarianceShrinkageOptimizer:
    """
    Ledoit-Wolf covariance shrinkage-based portfolio optimizer.
    Improves stability of covariance estimation in small
        samples.
    """

    def __init__(self):
        self.model = LedoitWolf()

    def shrink_covariance(self, returns: pd.DataFrame) ->
        np.ndarray:
        """
        Estimate shrunk covariance matrix using Ledoit-Wolf
            shrinkage.
        """
        self.model.fit(returns)
        return self.model.covariance_

    def optimize(self, expected_returns: pd.Series, returns:
        pd.DataFrame):
        """
        Optimize portfolio using shrunk covariance matrix.
        """
        cov = self.shrink_covariance(returns)
        inv_cov = np.linalg.pinv(cov)
        weights = inv_cov @ expected_returns
        weights /= np.sum(weights)
        return pd.Series(weights, index=expected_returns.index)
```

4.3 Implementation Details

In this project, both the Mean-Variance Optimization (Markowitz Model) and the Covariance-Based Optimization techniques were seamlessly integrated into the portfolio management workflow.

- The integration was managed through the `optimizer_factory.py` module, which dynamically selects the optimization strategy based on user input (either `"mean_variance"` or `"covariance"`).
- Before optimization, historical asset prices are fetched and transformed into log returns, ensuring data consistency and eliminating missing values.
- The selected optimizer then computes the optimal portfolio weights according to its respective objective function — variance minimization in the Markowitz model or covariance structure analysis in the Covariance Optimizer.

- Both optimizers share a uniform interface, allowing easy comparison of their results, such as portfolio risk, expected return, and Sharpe ratio.
- Finally, the comparative analysis of optimization outcomes helps identify which model performs better under specific market conditions and risk preferences.

Chapter 5

Streamlit-Based Portfolio Optimization Interface

5.1 Overview

The developed Streamlit interface serves as the front-end layer, seamlessly integrating all backend modules asset creation, data fetching, optimization, and portfolio analysis into an interactive, browser-based environment. It allows users to configure input parameters, select optimization techniques, and visualize performance metrics dynamically without requiring manual script execution.

5.2 Integration Architecture

The application internally connects various modular components following an OOP-driven design:

- **Asset Creation:** Handled through the `AssetFactory`, which dynamically constructs instances of stocks, bonds, ETFs, or crypto assets.
- **Data Fetching:** Managed via the `DataFetcherFactory`, which interfaces with APIs such as Yahoo Finance, FRED, or Binance.
- **Optimization:** Executed using the `OptimizerFactory`, supporting models like mean-variance and covariance shrinkage optimization.
- **Analysis:** Conducted by the `PortfolioAnalyzer`, which computes metrics such as expected return, volatility, and Sharpe ratio.

5.3 User Workflow

The user interacts with the application through a clean and intuitive interface, designed for both quantitative researchers and general investors:

1. Select the desired **assets** (custom/from table) and specify the **date range**.
2. Choose an **optimization method** (e.g., Mean-Variance, Covariance Shrinkage).

3. Click **Run Optimization** to execute the computation.
4. View the **optimized weights, portfolio returns, and risk metrics**.
5. Optionally, **download the results** as a CSV file.

5.4 Execution Instructions

Execution Steps

1. Install dependencies:

```
pip install -r requirements.txt
```

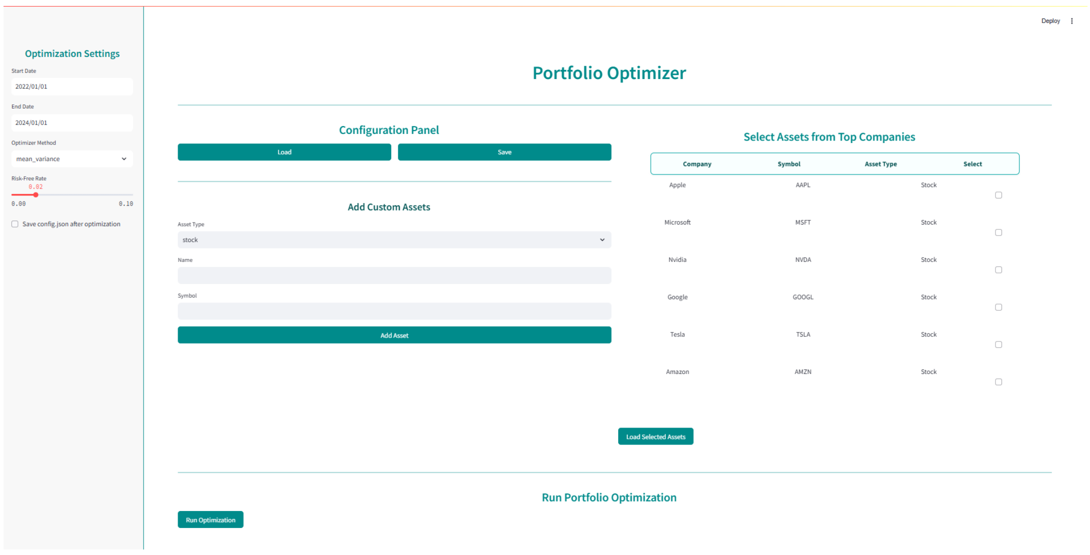
2. Launch the Streamlit interface:

```
streamlit run app.py
```

5.5 Interface Demonstration

5.5.1 Input Configuration Screen

The input screen allows users to select assets, choose data sources, and define the analysis period interactively.

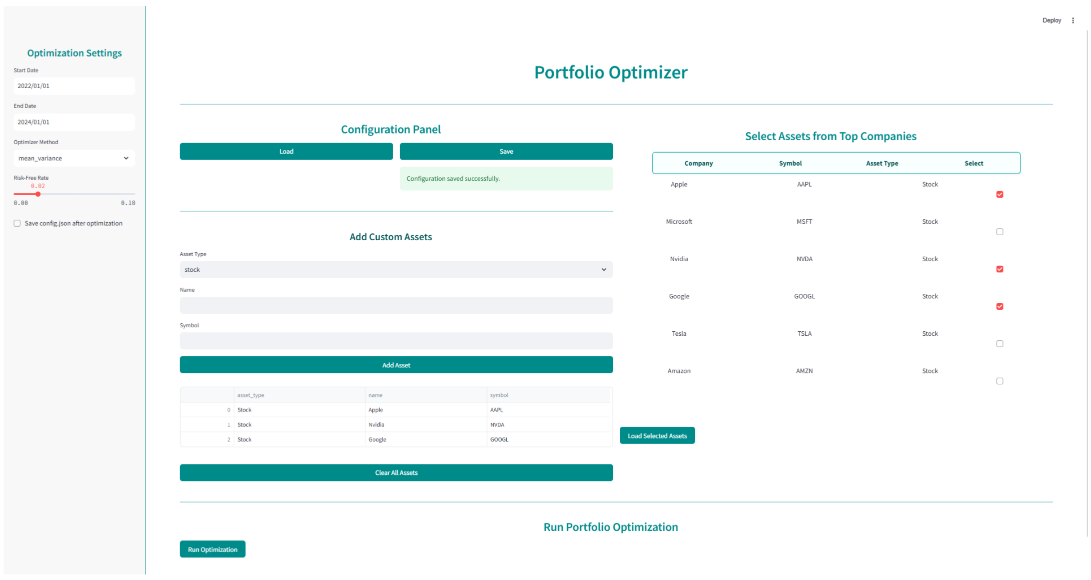


Start Date: 2022/01/01
End Date: 2024/01/01
Optimizer Method: mean_variance
Risk-Free Rate: 0.02 (0.00 – 0.10)
☐ Save config.json after optimization

Asset Type: stock
Name:
Symbol:

Company	Symbol	Asset Type	Select
Apple	AAPL	Stock	☐
Microsoft	MSFT	Stock	☐
Nvidia	NVDA	Stock	☐
Google	GOOGL	Stock	☐
Tesla	TSLA	Stock	☐
Amazon	AMZN	Stock	☐

Figure 5.1: Streamlit Input Configuration Interface.


Deploy

Optimization Settings

Start Date: 2022/01/01

End Date: 2024/01/01

Optimizer Method: mean_variance

Risk-Free Rate: 0.02 (0.00 – 0.10)

Save config.json after optimization

Portfolio Optimizer

Configuration Panel

Configuration saved successfully.

Add Custom Assets

Asset Type: stock

Name:

Symbol:

	asset_type	name	symbol
0	Stock	Apple	AAPL
1	Stock	Nvidia	NVDA
2	Stock	Google	GOOGL

Select Assets from Top Companies

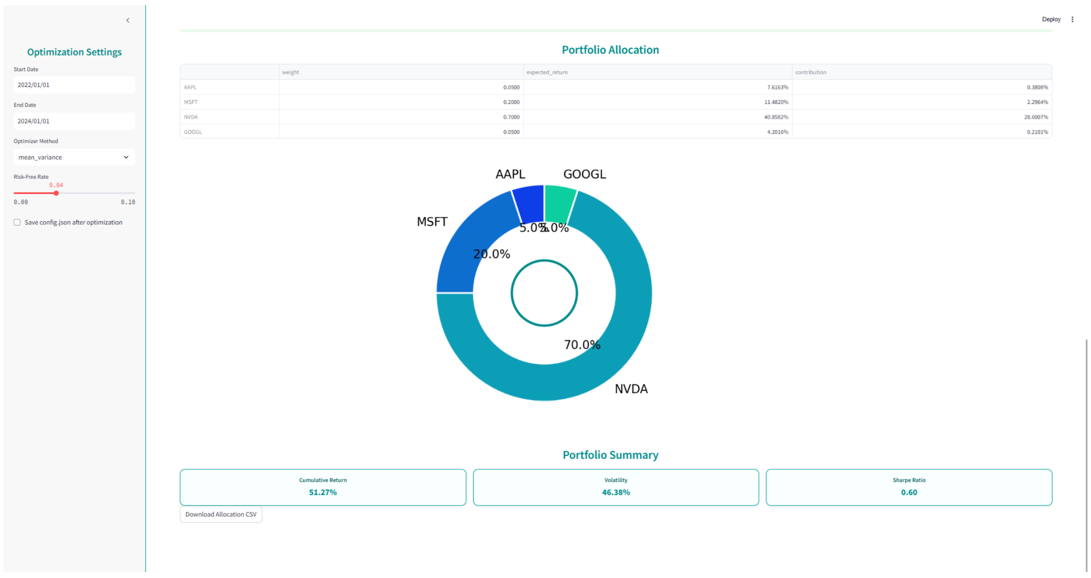
Company	Symbol	Asset Type	Select
Apple	AAPL	Stock	✓
Microsoft	MSFT	Stock	☐
Nvidia	NVDA	Stock	✓
Google	GOOGL	Stock	✓
Tesla	TSLA	Stock	☐
Amazon	AMZN	Stock	☐

Run Portfolio Optimization

Figure 5.2: Asset Selection and Data Loading Interface.

5.5.2 Optimization Results and Analysis

After running the optimization, the system displays the calculated portfolio weights, expected return, volatility, and Sharpe ratio, along with visual plots.


Deploy

Optimization Settings

Start Date: 2022/01/01

End Date: 2024/01/01

Optimizer Method: mean_variance

Risk-Free Rate: 0.04 (0.00 – 0.10)

Save config.json after optimization

Portfolio Allocation

	weight	expected_return	contribution
AAPL	0.0500	7.6163%	0.3808%
MSFT	0.2000	11.4820%	2.2964%
NVDA	0.7000	40.8582%	28.6007%
GOOGL	0.0500	4.2016%	0.2101%

Portfolio Summary

Cumulative Return	Volatility	Sharpe Ratio
51.27%	46.38%	0.60

Figure 5.3: Optimization Output and Portfolio Metrics Visualization.

Chapter 6

Results and Analysis

6.1 Object-Oriented Concepts Applied

OOP / Design Concept	**File / Module**	**Utility / Benefit**
Abstraction	`asset_interface.py`, `optimizer_interface.py`, `analyzer_interface.py`	Provides standardized contracts for assets, optimizers, and analyzers ensuring consistent usage across modules
Encapsulation	`stock.py`, `bond.py`, `crypto.py`, `ETF.py`, `PortfolioAnalyzer`	Keeps asset properties and analysis logic internal, exposing only necessary interfaces; supports maintainability
Single Responsibility Principle (SRP)	All individual asset classes, `PortfolioAnalyzer`	Each class has a focused responsibility, making the system modular and easier to maintain
Composition over Inheritance & Open/Closed Principle (OCP)	`AssetCollection`, `PortfolioManager`	Enables aggregation of assets and flexible portfolio-level operations without modifying existing code

Table 6.1: OOP and SOLID Concepts Applied (Part 1)

OOP / Design Concept	**File / Module**	**Utility / Benefit**
Dependency Inversion Principle (DIP) & Liskov Substitution Principle (LSP)	`DataFetcherFactory`, `OptimizerFactory`, `PortfolioAnalyzer`	Allows interchangeable data sources and optimizers, promoting extensibility and adherence to abstraction
Polymorphism	Factories (`AssetFactory`, `OptimizerFactory`)	Allows uniform handling of different asset types and optimization strategies without modifying calling code
Modularity & Maintainability	All modules (assets, data_fetcher, optimizer, analyzer)	Facilitates independent development, testing, and scalable system design

Table 6.2: OOP and SOLID Concepts Applied (Part 2)

Chapter 7

Future Work

- In the current implementation, the system optimizes asset weights purely based on return maximization without incorporating any explicit constraints on the user's available capital. In future extensions, we aim to integrate the user's initial investment amount into the optimization framework, enabling more realistic and personalized portfolio recommendations.
- Integration of reinforcement learning-based trading strategies to allow adaptive decision-making in dynamic market environments.
- Implementation of real-time market updates and an automated alert system for responsive portfolio adjustments.
- Deployment of the system on cloud infrastructure to support multi-user access and scalability.
- Enhancement of visual analytics through interactive, multi-page dashboards using Streamlit for improved user experience and transparency.

Bibliography

[1] Markowitz, H. (1952). Portfolio Selection. *The Journal of Finance*, 7(1), 77–91. Wiley. 10.1111/j.1540-6261.1952.tb01525.x

[2] Grinold, R. C., & Kahn, R. N. (1999). *Active Portfolio Management: A Quantitative Approach for Producing Superior Returns and Controlling Risk* (2nd ed.). McGraw-Hill Education. ISBN: 9780070248823.

[3] Booch, G. (2007). *Object-Oriented Analysis and Design with Applications* (3rd ed.). Addison-Wesley Professional. ISBN: 9780201895513.

[4] Martin, R. C. (2000). Design Principles and Design Patterns. Retrieved from https://objectmentor.com/resources/articles/Principles_and_Patterns.pdf. Introduced the SOLID principles for object-oriented software design.

[5] Streamlit Inc. (2025). *Streamlit — The fastest way to build data apps*. Retrieved from https://streamlit.io. Used for developing the interactive portfolio optimization user interface.

[6] OpenAI. (2025). *ChatGPT: Large Language Model for Conversational AI (GPT-5)*. Retrieved from https://chat.openai.com. Used to support documentation drafting and concept clarification.

[7] Liskov, B., & Wing, J. (1994). A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1811–1841. 10.1145/197320.197383

[8] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. ISBN: 9780201633610.