Introduction to Python Programming

Final Project

DeFi Portfolio Analyzer

Master 1 Data Sciences & Business Analytics

Submitted by:
Ya-Sin ZENK
Nicolas COQUELET

Professor:
Fabrice Popineau

December 28, 2025

Contents

1 Introduction

This report presents the **DeFi Portfolio Analyzer**, a Python application we built in three stages (`v0/` → `v1/` → `v2/`). It loads a cryptocurrency portfolio from JSON, fetches market data, computes risk metrics, and in v2, generates exports, plots, and an HTML report. We focused on modular design, clear documentation, logging, configuration, and testing to meet the requirements for a solid Python project.

Key deliverables include:

- Three runnable versions (`v0/`, `v1/`, `v2/`) with dedicated README files,
- A top-level README explaining execution from the repository root,
- Dependencies declared in `requirements.txt`,
- A comprehensive test suite using `pytest` (v1 and v2),
- Exportable outputs (CSV, JSON) and an optional HTML report with visualizations.

2 Context and Motivation

Crypto markets are volatile, and assets tend to move together during stress periods. So a portfolio tool can't just show "current value"—it needs to give a risk-oriented view: volatility (how risky each asset is), correlation (how well diversified you are), and downside exposure (VaR). We designed the application around three principles:

- **Explainability:** short explanations so non-finance users can actually understand the results.
- **Reproducibility:** caching, offline mode, and structured exports (CSV/JSON).
- **Practicality:** a CLI-first workflow (no notebooks required).

3 Problem Statement and Objectives

3.1 Problem statement

Given a portfolio of cryptocurrency assets (symbols and quantities), compute:

1. the total current value and allocation weights (baseline),
2. risk metrics based on historical daily prices over a configurable time window,
3. (optionally) an optimized allocation under user-defined constraints.

The user provides a JSON file describing their holdings. The application fetches live prices from an external API, computes metrics, and outputs results to the console, files, or a visual report.

3.2 Objectives

This project aims to show both *data-science* and *software-engineering* skills:

- **Modular architecture:** Code is split into purpose-specific modules (data loading, API client, risk analytics, optimization, visualization).
- **Object-oriented design:** Core abstractions (`Asset`, `Portfolio`) are implemented as classes with clear responsibilities.
- **Robust data pipeline:** Fetching, caching, aligning, and transforming time-series data reliably.
- **Usable CLI:** We use `argparse` for command-line parsing, `logging` for traceability, and YAML configuration for flexibility.
- **Testability:** A `pytest` suite covering critical logic and mocking external dependencies.
- **Meaningful outputs:** Exports, plots, and an interpretable HTML report.

4 Informal Specifications

4.1 Scope, constraints, and assumptions

Before getting into implementation details, here are the boundaries we set for the project:

- **No Jupyter notebook** is used as a final deliverable. All functionality is accessible via command-line scripts.
- **Three stages** (v0, v1, v2) must remain independently runnable. This ensures we can make progress without breaking earlier versions.
- Historical prices are fetched from the **CryptoCompare API**, a free service with rate limits. We handle rate-limiting via caching and an optional API key.
- Returns are computed as daily simple returns and annualized assuming 365 days/year—a common convention for crypto markets, which operate 24/7.
- Default parameters (historical window, risk-free rate, confidence level) are configurable via YAML or CLI overrides.

4.2 Inputs

The primary input is a portfolio JSON file specifying assets and quantities:

- **v0:** Prices are provided directly in the JSON (static, for testing the data model).
- **v1/v2:** Each asset includes a `crypto_id` field mapping to an API symbol; prices are fetched at runtime.

4.3 Outputs

Depending on the version and subcommand, the application produces:

- Console output summarizing metrics and allocation,
- Log files for debugging and traceability,
- (v2) CSV/JSON exports, PNG plots, and an optional HTML report.

5 What the Program Must Achieve

At a minimum, the program must:

1. Load a portfolio JSON file and validate its structure (required fields, correct types).
2. Compute allocation weights and total portfolio value.
3. (v1/v2) Fetch and align historical prices from an external API.
4. Compute risk metrics: annualized volatility, Sharpe ratio, historical VaR, and correlation matrix.
5. Provide a CLI that works consistently from the repository root.

These are the core requirements every version must satisfy. v1 and v2 then add more features on top of that.

6 Main Functionalities and Expected Outputs

6.1 Features per version

Table 1 summarizes the user-facing features across versions. We made sure not to break anything: v1 keeps all v0 features, and v2 keeps all v1 features.

Version	**Key features**
v0	Load a portfolio with static prices; compute total value and allocation weights. Output is deterministic and doesn't need network access.
v1	Fetch live and historical prices via API; compute volatility, Sharpe ratio, historical VaR, and correlation; print risk metrics to console.
v2	YAML configuration; local caching + offline mode; CLI subcommands (`analyze`, `optimize`, `visualize`); Markowitz optimizer; structured exports (CSV/JSON); PNG plots; HTML report with glossary; extensive test suite.

Table 1: Project functionalities per version.

6.2 Expected outputs

To give concrete examples:

- **Analyze (v2):** Produces per-asset metrics, allocation weights, and a correlation matrix as CSV/JSON exports (written to the chosen output directory; default: `outputs/`).
- **Optimize (v2):** Produces an optimal allocation export (written to the chosen output directory; default: `outputs/`).
- **Visualize (v2):** Produces plots (risk, correlation, allocation, frontier) and an optional HTML report (written to the chosen output directory; default: `figures/`).

7 Development Plan

7.1 Stages and milestones

The project follows incremental development:

- **v0 (MVP):** Implement core abstractions (`Asset`, `Portfolio`) and a deterministic output path. Goal: validate the data model before introducing external dependencies.
- **v1 (Risk analytics):** Add an HTTP client and compute metrics from historical data. Goal: deliver a working risk analyzer with live prices.
- **v2 (Final):** Add configuration, caching, optimization, visualizations, exports, and tests. Goal: production-quality CLI tool.

Note on Git milestones. The staged structure is captured in dedicated folders (`v0/`, `v1/`, `v2/`). For full traceability, we could tag commits as `v0`, `v1`, `v2` in Git. Right now, we're mainly using the folder-based separation; we can add Git tags before submission if needed.

7.2 Rationale for the order of implementation

We started with stable core data structures (`Asset`/`Portfolio`) and a well-defined input format. Why this order?

1. **Internal model first:** Without a correct data model, everything downstream breaks. v0 let us test parsing and weight computation in isolation.
2. **API calls second:** Network dependencies introduce variability (rate limits, failures). By deferring them to v1, we kept v0 deterministic and easy to debug.
3. **Optimization and visualization last:** These features depend on correct metrics and reliable data alignment. Building them last meant we had a stable foundation.

8 Design & Implementation

Now that we've covered the "what" (requirements) and "when" (development stages), let's look at the "how": architecture, data structures, algorithms, and key implementation choices.

8.1 Architecture overview

Figure 1 shows the v2 pipeline and how modules depend on each other. We followed separation of concerns: each module has a single responsibility.

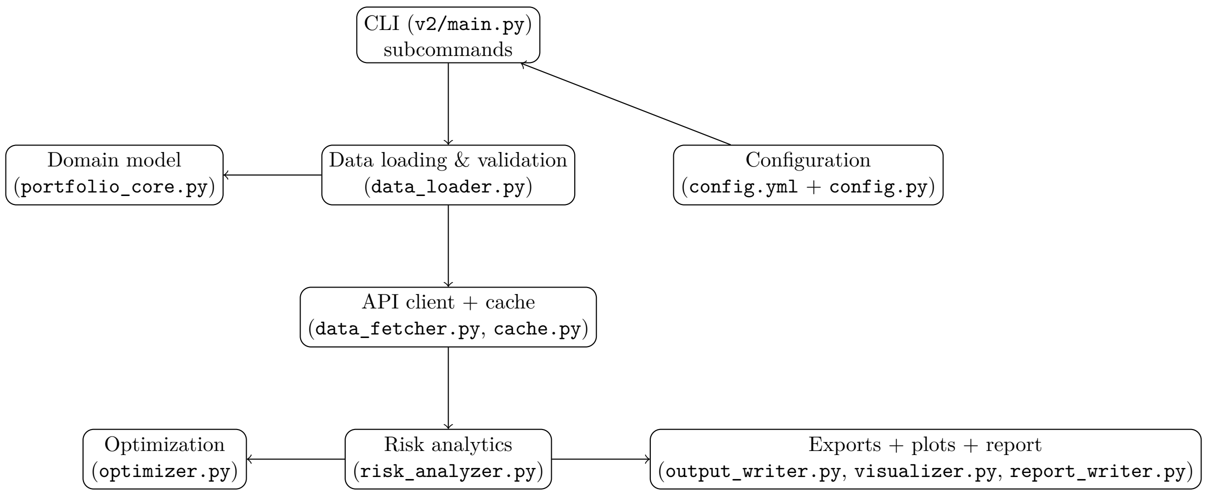

CLI (v2/main.py) subcommands
Domain model (portfolio_core.py)
Data loading & validation (data_loader.py)
Configuration (config.yml + config.py)
API client + cache (data_fetcher.py, cache.py)
Optimization (optimizer.py)
Risk analytics (risk_analyzer.py)
Exports + plots + report (output_writer.py, visualizer.py, report_writer.py)

Figure 1: High-level architecture of v2.

8.2 Key data structures

- **Asset (dataclass, frozen):** Represents a single position with `symbol`, `amount`, and optionally `price` and `crypto_id`. We made it immutable to avoid accidental changes.
- **Portfolio (class):** A collection of `Asset` instances with methods `total_value()` and `weights()`.
- **Configuration dataclasses (e.g., `AppConfig`, `RiskConfig`):** typed parameters loaded from YAML so we don't hardcode values.
- **Cache entries (JSON):** Historical price series stored locally with timestamps, so we can work offline and reproduce results.

8.3 Data pipeline

The pipeline proceeds as follows:

1. **Load and validate:** Parse the portfolio JSON, check required fields, and instantiate `Asset` objects.
2. **Fetch current prices:** Query the API for each asset's latest price.

3. **Fetch historical prices:** Query daily prices over the configured window; cache results locally.
4. **Align and transform:** Align all time series on common dates; compute simple returns.
5. **Compute metrics:** Per-asset volatility, Sharpe, VaR; portfolio-level volatility and correlation matrix.
6. **Export and visualize:** Write CSV/JSON, generate plots, and optionally render an HTML report.

8.4 Understanding the risk metrics

Before we get into formulas, here's what each metric actually means:

Volatility measures how much an asset's price bounces around. Higher volatility means more uncertainty—both upside potential and downside risk.

Sharpe ratio compares return to risk. A higher Sharpe means better "risk-adjusted" performance: you're getting more return per unit of volatility.

Value at Risk (VaR) answers: "What's the worst loss I might expect, 95% of the time?" It quantifies downside exposure.

Correlation shows how assets move together. High correlation means they rise and fall in sync, which reduces diversification benefits.

Efficient frontier shows the best possible risk/return trade-offs. Points on the frontier are "optimal"—no other portfolio gives you higher return for the same risk.

8.5 Risk metrics and formulas

Returns. Simple returns are computed as:

$$r_t = \frac{p_t}{p_{t-1}} - 1.$$

Volatility. If $\sigma(r)$ is the standard deviation of returns, annualized volatility is:

$$\sigma_{\text{ann}} = \sigma(r)\sqrt{365}.$$

Sharpe ratio. With annual risk-free rate r_f, the Sharpe ratio is:

$$\text{Sharpe} = \frac{\mathbb{E}[r - r_f/365]}{\sigma(r - r_f/365)}\sqrt{365}.$$

Historical VaR. For confidence level α (e.g., 0.95), historical VaR is:

$$\text{VaR}_\alpha = Q_{1-\alpha}(r),$$

where Q denotes the empirical quantile. This is a non-parametric approach—we just use the actual distribution of past returns.

Correlation and portfolio volatility. The correlation matrix tells us about diversification: low average correlation means losses in one asset might be offset by gains in another. Portfolio volatility takes these correlations into account:

$$\sigma_{\text{portfolio}} = \sqrt{w^\top \Sigma w},$$

where w is the weight vector and Σ is the covariance matrix.

8.6 Optimization (Markowitz)

In v2, optimization is performed under constraints:

$$\sum_{i=1}^{n} w_i = 1, \quad w_i \in [0, w_{\max}],$$

with three modes:

- **min-vol:** minimize variance $w^\top \Sigma w$,
- **max-sharpe:** maximize $(\mu^\top w - r_f)/\sqrt{w^\top \Sigma w}$,
- **target-return:** minimize variance such that $\mu^\top w = r^\star$.

We approximate the efficient frontier by solving multiple target-return problems between the feasible minimum and maximum expected returns. If a target isn't feasible, we just skip it.

8.7 Complexity considerations

If n is the number of assets and T the number of aligned days:

- Return computation: $O(nT)$,
- Covariance/correlation: $O(n^2 T)$,
- Optimization: iterative (SLSQP), practically fast for small n.

Code references (no code listing). For detailed implementation and docstrings, see:

- `v2/risk_analyzer.py` (e.g., `portfolio_volatility()`, `correlation_matrix()`): risk metrics and portfolio aggregation,
- `v2/optimizer.py` (e.g., `max_sharpe()`, `efficient_frontier()`): constraints, objectives, and efficient frontier,
- `v2/data_fetcher.py` and `v2/cache.py`: API client, caching, offline mode,
- `v2/main.py`: CLI subcommands and I/O orchestration.

The v2 codebase is over 2,000 lines of Python across 12 modules, which meets the non-trivial implementation requirement.

9 User Guide

This section provides practical instructions for installing and running the application.

9.1 Installation

From the repository root:

```
python -m venv .venv
source .venv/bin/activate  # macOS / Linux
.venv\Scripts\activate     # Windows
pip install -r requirements.txt
```

v0 uses the standard library only; dependencies in `requirements.txt` are for v1/v2 (pandas, numpy, scipy, matplotlib, requests, pyyaml, pytest).

9.2 Execution (commands)

v0 (static prices, deterministic):

```
python v0/main.py --portfolio data/sample_portfolio.json
```

v1 (live prices, risk metrics):

```
python v1/main.py --portfolio data/sample_portfolio.json --days 30
```

v2 (full CLI with subcommands):

```
python v2/main.py analyze --portfolio data/sample_portfolio.json \
  --outdir outputs --pretty
python v2/main.py optimize --portfolio data/sample_portfolio.json \
  --mode max-sharpe --outdir outputs --pretty
python v2/main.py visualize --portfolio data/sample_portfolio.json \
  --outdir report/assets --report
```

9.3 Running tests

From the repository root:

```
pytest v1/tests/ -v
pytest v2/tests/ -v
```

9.4 CLI options (v2 highlights)

- `--pretty` prints human-readable tables in the console (exports unchanged).

- `--offline` uses cache only (no network calls)—useful for reproducibility.
- `--refresh-cache` bypasses cache and refreshes data from the API.
- `--log-level` / `--quiet` / `--verbose` control console verbosity.

9.5 Configuration examples

v2 reads `v2/config.yml` by default and supports CLI overrides for key parameters (days, risk-free rate, confidence, output directory). This way, you can set defaults once and override them as needed.

API key. CryptoCompare works without an API key, but you're limited to 100 calls/hour. For higher limits, set the environment variable `CRYPTOCOMPARE_API_KEY`.

9.6 Example inputs/outputs

Example portfolio format (v1/v2):

```
{
  "name": "defi_portfolio",
  "assets": [
    {"symbol": "ETH", "crypto_id": "ETH", "amount": 1.5},
    {"symbol": "BTC", "crypto_id": "BTC", "amount": 0.1}
  ]
}
```

10 Example Outputs and Visualizations

This section shows examples of the figures generated by v2 (Figure 2 and Figure 3).

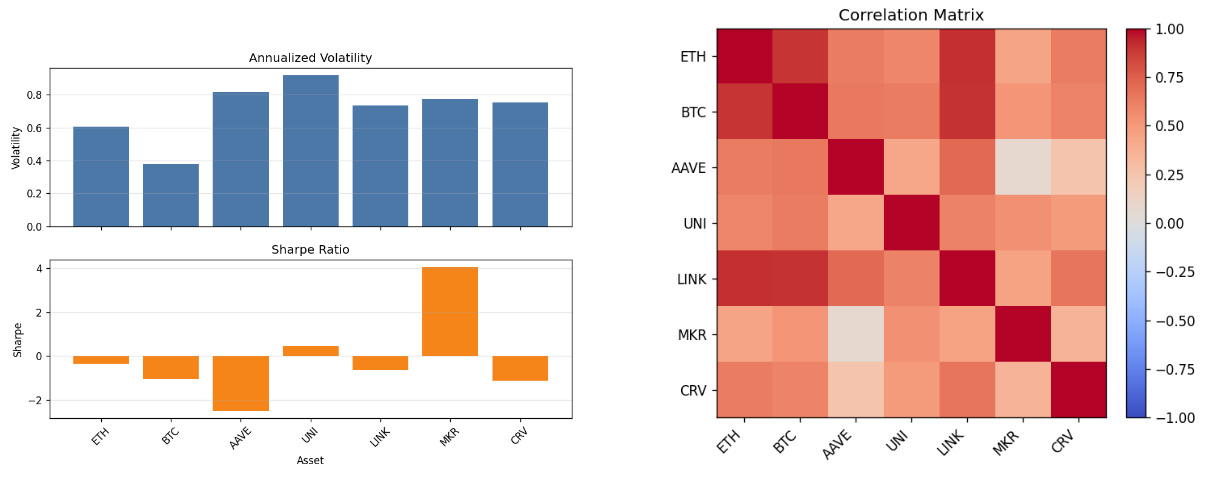
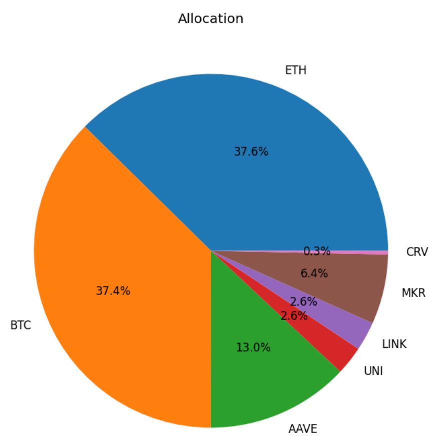


Figure 2: (Left) Per-asset annualized volatility (risk bars). (Right) Correlation heatmap between assets; values close to 1 indicate assets that move together.



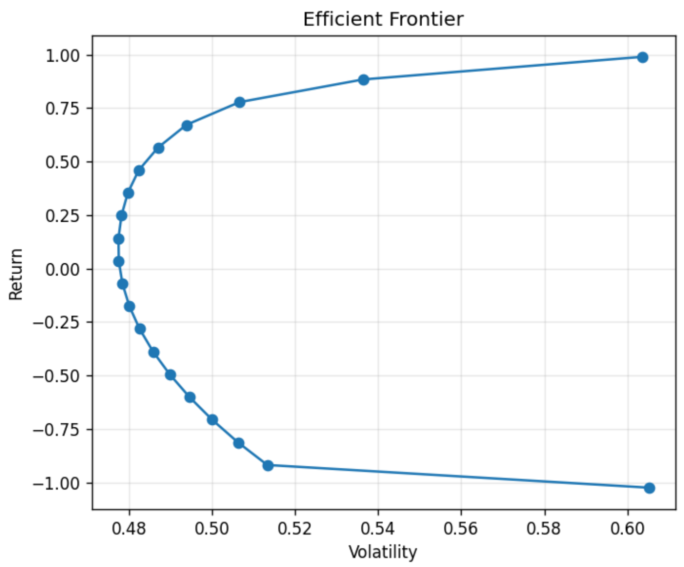


Figure 3: (Left) Current portfolio allocation (weights by market value). (Right) Efficient frontier under the configured constraints (risk/return trade-offs).

11 Evaluation & Discussion

Now that we've covered the design and implementation, let's reflect on what we achieved, what was difficult, and what the limitations are.

11.1 Level of completion per stage

- **v0:** Complete baseline with deterministic output. We validated the data model without any external dependencies.
- **v1:** Complete analytics pipeline with live prices, aligned returns, and standard metrics. All core risk features work.
- **v2:** Complete final stage with configuration, caching/offline mode, optimization, exports, plots, HTML report, and 9 test files.

11.2 Testing strategy

The test suite focuses on correctness and robustness:

- Unit tests for portfolio computations and risk metrics (edge cases, zero values).
- Unit tests for cache round-trip and TTL behavior.
- Optimizer tests to make sure weights are valid and constraints are satisfied.
- CLI tests using mocking to avoid non-deterministic API calls.

We focused on deterministic and critical paths (parsing, metrics, cache integrity), and we check that plotting routines generate files without errors.

11.3 Main difficulties encountered and mitigations

We ran into several challenges during development:

- **API rate limits and variability:** We added caching with TTL, a refresh option, and offline mode; tests mock API calls.
- **Time-series alignment:** We align on common dates and fail fast if the aligned series is too short.
- **Cross-platform execution:** We resolve paths so root-level commands work consistently.
- **Optimization feasibility:** We restrict targets to feasible ranges and skip infeasible points on the frontier.
- **Interpretability:** We provide an HTML report with a glossary and data-driven interpretation for non-finance users.

11.4 Reproducibility and offline mode

Reproducibility is critical for any data-driven analysis. The application supports:

- **Local caching + offline mode:** historical series are cached with timestamps and can be re-used without network calls.
- **Structured exports:** CSV/JSON outputs can be archived and compared across runs.

11.5 Known limitations

There are several limitations we're aware of:

- **Short samples:** With only 30 days of data, estimates of expected returns and covariance can be unstable. Increasing the window helps, but assumes stationarity.
- **Historical VaR is backward-looking:** It uses past returns and doesn't extrapolate tail risk beyond the sample. Extreme events not in the sample won't be captured.
- **Optimization sensitivity:** Markowitz optimization is sensitive to estimation noise and ignores transaction costs; results are indicative, not something you should follow blindly.

12 Possible Extensions or Improvements

The current implementation meets the project requirements, but several extensions could make it more useful:

- **Alternative risk models:** parametric VaR or CVaR (expected shortfall) for tail risk.
- **Benchmark comparison:** compare against BTC-only or equal-weight baselines.
- **Realistic constraints:** turnover limits and transaction costs in optimization.

We left these as future work to keep the current scope manageable and stable.

13 Conclusion

The DeFi Portfolio Analyzer shows a complete staged development workflow and a clean modular architecture applied to a data-driven problem. We started from a minimal prototype (v0), then incrementally added API integration (v1) and advanced features (v2) without breaking anything.

The final version is a configurable and testable CLI tool with reproducible outputs, visualizations, and an HTML report that non-finance users can understand. The codebase is over 2,000 lines of Python organized into 12 modules, with a pytest suite of 14 test files.

Key learnings (technical and methodological):

- **Modular architecture:** separation of concerns makes the code easier to maintain and test.
- **CLI + logging:** argparse subcommands and consistent logs make the tool easier to use and debug.
- **Configuration + reproducibility:** YAML + typed config, caching, and exports let you repeat runs reliably.
- **Testing:** unit and CLI tests with mocking give you fast, deterministic feedback.

We believe this project meets the requirements for a non-trivial Python application: it tackles a clearly defined problem with a well-structured codebase, follows best practices from the course (OOP, logging, argparse, configuration, testing, visualization), and produces meaningful, reproducible outputs.