A Backtesting Framework for Systematic Strategy Research

TIC's Quant Department – Risk Management

by

Ivan Khomyanin
Bart van den Akker
Huzaifa Bukhari
Frank Heijnen

Abstract

This document outlines the architecture and rationale of a config-driven backtesting framework designed for collaborative research in the Tilburg Investment Club. The system separates strategy logic from backtest configuration, enabling reproducible experiments, consistent execution, and scalable multi-strategy research. We describe the data layer, strategy interface, configuration files, execution model, and overall workflow.

Contents

1 Motivation

Collaborative research groups typically face the same recurring issues when multiple members implement and test investment strategies:

- Hard-coded parameters inside Python files lead to inconsistent results.
- Strategy code becomes entangled with data loading, execution rules, and dates.
- Experiments cannot be reproduced because configuration drift is untracked.
- Batch testing and walk-forward validation require manual edits to source code.

To solve these problems, we adopt a **config-driven architecture**, where all strategy logic is written in Python, but the environment in which the strategy operates is fully defined by external configuration files. This isolates the *idea* (strategy) from the *experiment* (data, execution settings, dates).

2 Separation of Code and Configuration

The framework enforces a strict separation:

- **Strategies**: implemented as Python classes with clearly defined inputs and outputs.
- **Configs**: YAML files specifying which strategy to run, over what data, using which execution and risk settings.

This design ensures the following:

- The same strategy class can be tested under many environments without editing code.
- Every backtest run is fully reproducible by saving the configuration file.
- Parameter sweeps and sensitivity analyses are possible without code duplication.
- Framework updates do not break user strategies, and user strategies do not break the framework.

3 Folder Structure

A clean folder structure enables multiple teams to work on the framework without conflicts:

```
QIG-RM/
    backtesting/
        strategies/
            <.py files containing a strategy>
        configs/
            <.yml files containing configurations>
    results/
        <auto-generated per run>
    backtesting_framework.py
    data_store.py
```

4 Strategy Interface

Strategies are implemented in Python and produce deterministic signals. Below is a minimal but representative interface:

```python
class Strategy:
    def generate_signals(self, data):
        raise NotImplementedError
```

A concrete example:

```python
class MomentumStrategy(Strategy):
    def __init__(self, lookback, top_quantile, rebalance_frequency):
        self.lookback = lookback
        self.top_quantile = top_quantile
        self.rebalance_frequency = rebalance_frequency

    def generate_signals(self, data):
        returns = data["Close"].pct_change(self.lookback)
        ranks = returns.rank(axis=1, pct=True)
        long  = (ranks >= 1 - self.top_quantile).astype(int)
        short = (ranks <= self.top_quantile).astype(int) * -1
        return long + short
```

Note that:

- All parameters are passed at initialization.
- No dates, data paths, risk limits, or execution rules appear here.
- The class is usable with any config.

5 Configuration Files (YAML)

Configuration files define the environment in which a strategy is evaluated.

An example:

```yaml
strategy:
  class: MomentumStrategy
  parameters:
    lookback: 60
    top_quantile: 0.2
    rebalance_frequency: "1M"

data:
  universe: "sp500"
  freq: "1D"
  source: "local_parquet"
  include_delisted: true

execution:
  slippage_bps: 3
  fees_bps: 1
  trade_at_open: true

risk:
  vol_target: 0.15
  max_leverage: 2.0

backtest:
  start: "2015-01-01"
  end: "2024-12-31"
  initial_capital: 1000000

output:
  path: "results/mom60/"
```

The config file formally encodes:

- **Which strategy class to run**
- **Which data universe and frequency to load**
- **Execution assumptions**: costs, slippage, trading rules
- **Risk overlays**: leverage, volatility targeting
- **Backtest horizon**
- **Output location**

Because of this structure, two backtests differ only by their config files—not by code.

6 Backtest Runner

In the framework architecture used by the Tilburg Investment Club, the entire backtesting engine is implemented in a single file: `backtesting/backtesting_framework.py`. This file contains:

- configuration loading and validation,
- strategy instantiation,
- data loading via a `DataStore` interface,
- execution and portfolio accounting,
- result logging and output generation.

The backtest runner follows three steps:

1. Load the YAML configuration file.
2. Dynamically import the strategy class from the `strategies/` directory.
3. Execute the backtest using the shared framework logic.

A minimal version of the runner is shown below:

```python
# backtesting/backtesting_framework.py
import yaml
import importlib

from data_store import DataStore

class BacktestEngine:
    def __init__(self, config):
        self.cfg = config
        self.store = DataStore(config["data"])

    def run(self, strategy):
        data = self.store.load_universe(self.cfg["data"])
        signals = strategy.generate_signals(data)
        # execution + accounting logic here...
        return self._compute_results(signals, data)

def run_from_config(config_path):
    cfg = yaml.safe_load(open(config_path))

    module = importlib.import_module(
        f"backtesting.strategies.{cfg['strategy']['class'].lower()}"
    )
    StratClass = getattr(module, cfg["strategy"]["class"])
    strategy = StratClass(**cfg["strategy"]["parameters"])

    engine = BacktestEngine(cfg)
    return engine.run(strategy)

if __name__ == "__main__":
    import sys
    run_from_config(sys.argv[1])
```

A backtest is launched using:

```
python backtesting/backtesting_framework.py configs/mom_60.yml
```

7 Data Layer Abstraction

All data access is handled by a small file, `data_store.py`, which provides a stable interface for loading market data regardless of where it is physically stored. The group currently uses Parquet files synced through Google Drive, but the design allows seamless migration to SQL or cloud storage without modifying strategy code.

A simplified version of the abstraction:

```python
# backtesting/data_store.py
import pandas as pd
from pathlib import Path

class DataStore:
    def __init__(self, cfg):
        self.root = Path(cfg["path"])

    def load_universe(self, cfg):
        universe = cfg["universe"]
        freq = cfg["freq"]
        folder = self.root / f"{universe}_{freq}"
        frames = [
            pd.read_parquet(folder / f"{asset}.parquet")
            for asset in cfg["assets"]
        ]
        return {a: f for a, f in zip(cfg["assets"], frames)}
```

All strategies call data in the same way:

```
data = store.load_universe(cfg["data"])
```

This removes user dependency on local file paths, storage backends, or metadata formats.

8 Execution and Accounting

The execution model and portfolio accounting logic are implemented directly inside `backtesting_framework.py`. While strategies only generate signals, the engine determines how those signals translate into positions, trades, and P&L.

The execution layer handles:

- slippage and transaction cost assumptions,
- trade-at-open/close behavior,
- position sizing rules,
- rebalancing frequency,
- leverage constraints.

Accounting then computes:

- daily returns,
- exposure and leverage,
- volatility, Sharpe ratio, and drawdowns,
- turnover and trading cost attribution.

Keeping execution and accounting in a single engine file makes the framework easier to maintain and ensures that all strategies share identical assumptions.

9 Reproducibility and Experiment Logging

Every backtest run produces a results folder containing:

- a copy of the configuration file used,
- a small JSON with timestamps and framework version,
- performance metrics (Sharpe, drawdown, turnover),
- time series of portfolio value and exposures,
- strategy-level signals and trades (optional).

Because all parameters live in a standalone YAML configuration file, any experiment can be reproduced exactly by re-running the same config against the same version of the framework and data.

This structure ensures auditability, prevents configuration drift, and enables parameter sweeps or walk-forward validation without editing strategy code.

10 Conclusion

The architecture used by the Tilburg Investment Club keeps the complexity low while preserving the essential features of a professional backtesting environment. A single engine file, a unified data interface, and external configuration files allow members to collaborate effectively without breaking each other's work.

As the number of strategies and users grows, the framework can be split into multiple files or modules, but the current layout strikes the ideal balance between clarity, flexibility, and reproducibility.