Vectester v2.0 User's Manual

www.pyquantlab.com

Table of Contents

1. Introduction

Vectester is a desktop trading research and backtesting application designed to help users test, compare, refine, and validate rule-based trading ideas in a structured environment. In practical terms, it gives the user a complete workflow for loading market data, selecting or creating a strategy, configuring its parameters, running a backtest, optimizing those parameters, and then studying the results through charts, metrics, validation tools, comparison views, and exportable reports.

Unlike a simple charting tool or a basic "run strategy" utility, Vectester is built as a research workspace. Its purpose is not only to show whether a strategy made money on historical data, but also to help the user understand *why* it behaved the way it did, how sensitive it is to parameter choices, how stable it remains across different samples, and how much confidence can reasonably be placed in its apparent edge. That distinction is important. A backtest result by itself is often only a first impression. Vectester is meant to support the much more important second step: determining whether that result is meaningful, fragile, or misleading.

The application combines several parts of the strategy-development process into one interface. It supports market data loading from multiple sources, single-run backtesting, exhaustive parameter optimization, walk-forward analysis, Monte Carlo simulation, result comparison, and HTML report export. Just as importantly, it also includes a strategy authoring and editing workflow, allowing users not only to run included strategies, but to study, modify, and create their own.

Because of that, Vectester serves two roles at the same time. It is a backtesting studio for users who want to test ideas quickly, and it is also a strategy development environment for users who want to design, adjust, and iterate on their own trading logic. A proper understanding of both roles is essential for using the platform effectively.

1.1 What Vectester is

Vectester is a professional desktop application for historical strategy testing and trading-system research. It allows the user to answer questions such as:

- What would this trading strategy have done on past data?
- How do its results change when I alter key parameters?
- Is the strategy genuinely robust, or is it merely fitted to one historical sample?
- Does the strategy behave better than alternatives on the same market and timeframe?
- Are the results likely driven by edge, luck, or a fragile combination of settings?

At the most basic level, Vectester takes market data and strategy rules, then simulates how that strategy would have behaved over the selected historical period. It tracks trades, equity growth, drawdowns, returns, and many other performance metrics. But its real value comes from going beyond that single simulation.

A user can define parameter grids and run optimization to search for stronger configurations. The resulting “best” settings can then be challenged with walk-forward analysis, which tests whether optimized settings continue to work outside the data they were tuned on. Monte Carlo analysis goes even further by examining how much the final outcome depends on the order of trades and how widely results can vary even when the underlying trade set stays the same. A comparison page allows the user to place multiple strategy runs side by side, and report export makes it easy to document and share findings.

Vectester also treats the strategy itself as something the user can work on directly. Strategies are not fixed black boxes. They can be studied, edited, duplicated, refined, and extended. This makes the platform especially useful not only for evaluating trading systems, but for *developing* them.

In short, Vectester is not merely a tool for producing backtest numbers. It is a structured environment for researching trading ideas, diagnosing weak logic, improving strategy design, and testing whether good-looking results can survive stricter examination.

1.2 Who it is for

Vectester is intended for users who approach trading systematically and want to evaluate ideas through rules, data, and repeatable testing rather than intuition alone.

It is especially suitable for the following groups.

Strategy developers

These users want to create and refine their own trading logic. For them, Vectester is valuable because it does not stop at running built-in systems. It supports strategy creation and modification, parameter definition, iterative testing, and validation. A developer can begin with a simple idea, test it, observe weaknesses, make adjustments, and repeat the cycle until the strategy becomes clearer and more robust.

Quantitative and semi-systematic traders

Some users do not consider themselves software developers, but they still work in a rule-driven way. They may want to test combinations of indicators, trend filters, momentum rules, stop structures, or volatility conditions. Vectester is appropriate for these users because it presents the strategy process in a visual, workflow-oriented format while still exposing enough structure to support serious experimentation.

Researchers and analysts

These users are often less interested in deploying a trading strategy immediately and more interested in understanding how different methods behave. They may compare trend-following with mean reversion, study different risk controls, or evaluate robustness across time. Vectester supports this type of work through comparison tools, optimization, walk-forward analysis, and report export.

Educators and advanced learners

Vectester is also useful for those learning how systematic trading works. Because the application ties together market data, signals, parameters, performance metrics, and validation tools, it helps the user see how a trading idea becomes a formal testable system. It is particularly helpful for users moving from “indicator watching” to genuine rule-based research.

Who it is not primarily for

Vectester is generally not aimed at users looking for discretionary chart annotation, manual trade journaling, broker integration, or live trade execution. Its core focus is historical testing and strategy research. It is also not meant to provide certainty. A well-tested result can improve confidence, but it cannot eliminate uncertainty or guarantee future performance.

A user will get the most value from Vectester if they are willing to think in terms of hypotheses, rules, evidence, and iteration. The platform rewards careful, disciplined use. It is less useful for users seeking instant confirmation that a favorite idea “works.”

1.3 Main workflow

Although Vectester contains multiple tools and pages, its overall workflow is straightforward. The user typically moves through a sequence that starts with data and ends with interpretation and documentation.

Step 1: Load market data

Every backtest begins with historical price data. The user first loads a dataset for the market they want to study. This dataset is the raw material on which the strategy will operate. If the data is incomplete, misformatted, too short, or inappropriate for the strategy’s timeframe and logic, the rest of the analysis will be compromised from the start.

At this stage, the user is not yet evaluating a strategy. They are defining the historical environment in which the strategy will be tested.

Step 2: Select, study, create, or modify a strategy

Once data is available, the user chooses a strategy. This may be one of the built-in strategies or a user-created strategy. In Vectester, this step is far more important than

simply picking a name from a list. A strategy is the formal expression of a trading hypothesis. It defines what constitutes an entry, what constitutes an exit, and in some cases how stop-losses, take-profits, or trailing behavior should work.

Because Vectester allows users to add and edit strategies, the workflow often includes examining the strategy's parameters, understanding what they do, and adjusting the strategy logic itself when necessary.

Step 3: Configure parameters and portfolio settings

A strategy usually contains tunable parameters such as lookback windows, thresholds, stop distances, momentum filters, or volatility multipliers. The user sets these values for a single test or enters multiple candidate values for optimization.

At the same time, portfolio-level settings must also be chosen. These include items such as direction, initial capital, fees, slippage, and frequency. These settings strongly affect realism and interpretation. A strategy that looks good with zero friction may become much weaker once realistic costs are included.

Step 4: Run a single backtest

The first serious question is usually simple: *How does this strategy behave with these settings on this data?* Running a single backtest provides the initial answer. The user receives charts, metrics, trade records, and summary statistics.

This step is often exploratory. It helps determine whether the idea even produces meaningful trades, whether the behavior broadly matches expectations, and whether there are obvious issues such as too few trades, excessive drawdowns, poor sensitivity to costs, or unrealistic smoothness.

Step 5: Optimize parameters

If the initial run is promising, the user may want to know whether nearby parameter combinations perform better or whether the strategy is highly sensitive to small changes. Optimization addresses this by testing many parameter combinations and identifying the strongest result according to a chosen metric.

This stage is powerful but dangerous. Optimization can reveal useful parameter regions, but it can also produce deceptively strong results by fitting the strategy too closely to one historical sample. In other words, optimization can improve a strategy, but it can just as easily create the *illusion* of improvement.

Step 6: Validate the result

A strong optimized result should never be accepted immediately. Validation is the process of challenging it.

In Vectester, this usually means:

- **Walk-forward analysis**, to test whether optimized settings retain value outside the sample used to find them.
- **Monte Carlo simulation**, to estimate how much luck and trade ordering may affect the outcome.
- **Comparison**, to see whether the result is truly strong relative to alternatives rather than only strong in isolation.

This is the stage where many apparently excellent strategies begin to fail. That is not a flaw in the software. It is one of the most valuable outcomes research can produce.

Step 7: Interpret and document

After testing and validation, the user decides what the evidence actually supports. A result may justify further work, simplification, rejection, or comparison against another idea. The report export feature helps document these findings in a presentable form.

A disciplined workflow in Vectester therefore looks like this:

data → strategy → parameters → backtest → optimization → validation → interpretation

Users who skip the later stages often end up trusting fragile results. Users who treat the full workflow seriously are more likely to separate genuine robustness from attractive but unreliable backtests.

1.4 Core concepts: data, strategy, backtest, optimization, validation

To use Vectester properly, the user must understand five core ideas. Nearly every action in the application belongs to one of these concepts.

Data

Data is the historical market record the strategy is tested on. In most cases this includes price bars with at least open, high, low, and close values, and often volume as well. Data determines the historical environment in which the strategy is evaluated.

Data is not merely an input file. It shapes the entire meaning of the test. The same strategy may look very different on daily data versus intraday data, on one asset versus another, or across one historical period versus another. A short sample can make a poor strategy appear impressive. A noisy or incomplete sample can distort signals. A highly unusual sample can produce misleading conclusions.

For that reason, data selection is never a trivial setup step. It is part of the research question itself.

When a user asks, “Does this strategy work?”, the more precise version is:
“Does this strategy, with these rules and assumptions, behave acceptably on this specific historical dataset?”

That is a much narrower and more honest question.

Strategy

A strategy is a formal set of rules that tells Vectester when to enter and exit trades, and sometimes how to manage them while open. It is the operational form of a trading idea.

A good way to think about a strategy is as an answer to four questions:

1. Under what conditions should a trade begin?
2. Under what conditions should it end?
3. What filters or confirmations must be satisfied?
4. What risk controls should apply while the position is open?

Strategies in Vectester are parameterized. That means they usually do not contain one fixed behavior only. Instead, they expose adjustable values such as moving-average lengths, threshold levels, volatility settings, or stop distances. Those values let the user tune the strategy, but they also create the risk of overfitting if used carelessly.

A strategy is therefore not just “an indicator combination.” It is a full decision system expressed through rules, parameters, and trade-management behavior.

Backtest

A backtest is the historical simulation of a strategy on a chosen dataset under chosen assumptions. It answers the question:
“What would this strategy have done if it had followed its rules across this historical period?”

The backtest applies the strategy’s signals to the data and tracks the resulting trade sequence and portfolio path. From that, Vectester produces outputs such as:

- total return
- risk-adjusted metrics
- drawdown measures
- trade count
- win rate
- expectancy

- monthly returns
- equity growth over time

A backtest is useful because it turns an idea into evidence. But that evidence is limited. A backtest does not prove that a strategy will work in the future. It only shows how the strategy would have behaved under the chosen historical conditions and assumptions.

A backtest can be informative, misleading, or both at the same time. For example, a strategy may show excellent returns but only because of one exceptional market period. It may show a high Sharpe ratio but only with very few trades. It may beat a benchmark only before realistic fees and slippage are included. This is why a backtest should be treated as a starting point for investigation, not a final verdict.

Optimization

Optimization is the process of testing many parameter combinations to find which settings produce the strongest backtest result according to a selected metric.

This is one of the most attractive features in any backtesting platform because it appears to answer a practical question:
“What settings work best?”

However, that question has a hidden danger. The “best” settings often mean only:
“the settings that fit this specific sample best.”

Optimization is useful for:

- locating promising parameter regions
- understanding parameter sensitivity
- comparing broad configurations
- improving obviously weak defaults
- identifying whether a strategy has stable behavior across nearby values

Optimization becomes dangerous when:

- too many parameters are varied at once
- too many values are tested
- the user trusts the top result blindly
- the selected metric is too narrow
- no validation follows afterward

In other words, optimization can help reveal structure, but it can also produce false confidence. In Vectester, it should be treated as a search tool, not as proof.

Validation

Validation is the process of testing whether a promising result remains meaningful when examined more critically. It is what separates exploratory success from credible evidence.

A strategy that performs well in a single backtest may still fail for several reasons:

- it may be overfit to a specific sample
- it may depend heavily on fortunate trade ordering
- it may be overly sensitive to parameters
- it may collapse when applied to unseen portions of data
- it may simply be weaker than alternatives once compared fairly

Validation is therefore the most important stage in serious research. In Vectester, it is supported mainly through walk-forward analysis, Monte Carlo analysis, and side-by-side comparison.

Walk-forward analysis tests whether parameter choices found in one sample continue to work in later out-of-sample segments. Monte Carlo analysis tests how widely outcomes can vary when trade order changes. Comparison tests whether a strategy is attractive in context rather than in isolation.

Validation does not guarantee truth. It does something more realistic and more useful: it reduces the chance that the user mistakes a fragile historical artifact for a robust trading edge.

Why these concepts matter together

These five concepts are not separate features. They are the logic of the entire platform.

Data defines the environment.
Strategy defines the rules.
Backtest measures historical behavior.
Optimization searches for stronger settings.
Validation tests whether those results deserve trust.

A user who understands these concepts will use Vectester as a research instrument. A user who ignores them may still produce impressive charts and metrics, but those outputs will be much easier to misread.

The purpose of the rest of this manual is to show how to use each part of Vectester within that framework, so that results are not only produced, but understood properly.

2. Interface Overview

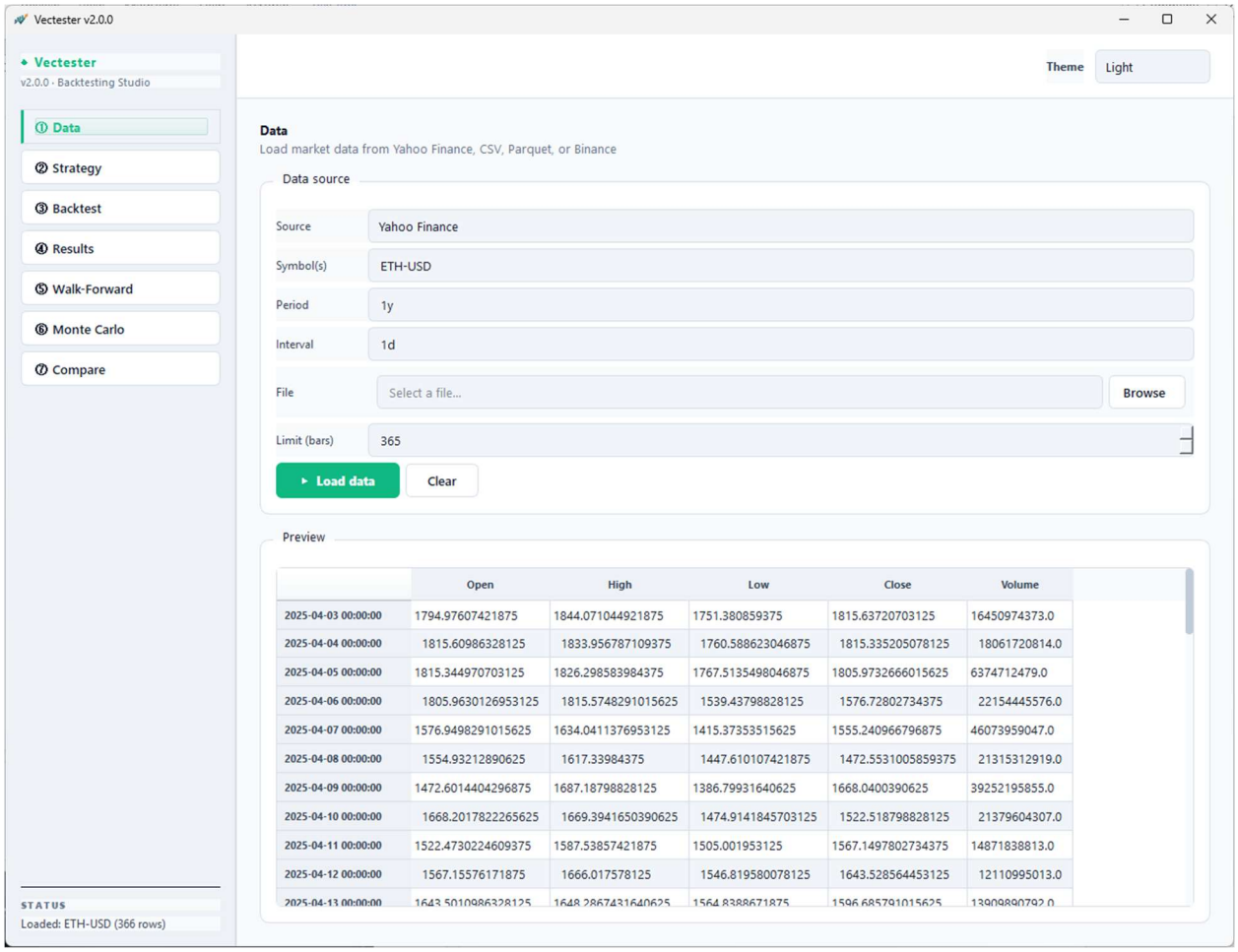
Vectester is organized as a guided research workspace. The interface is built around a left-hand navigation sidebar and a main content area. Each page focuses on one stage of the trading-strategy workflow: loading data, selecting or creating a strategy, configuring the backtest, reviewing results, validating robustness, and comparing alternatives.

The application is designed so that a user can move through the process step by step, but it does not force a rigid sequence. You can return to earlier pages, change settings, rerun tests, and immediately see how those changes affect the output.

2.1 Main Pages

Vectester contains seven main pages, shown in the left sidebar as numbered navigation items.

① Data



The **Data** page is the starting point for every session. This is where you load the market data that the strategy will use.

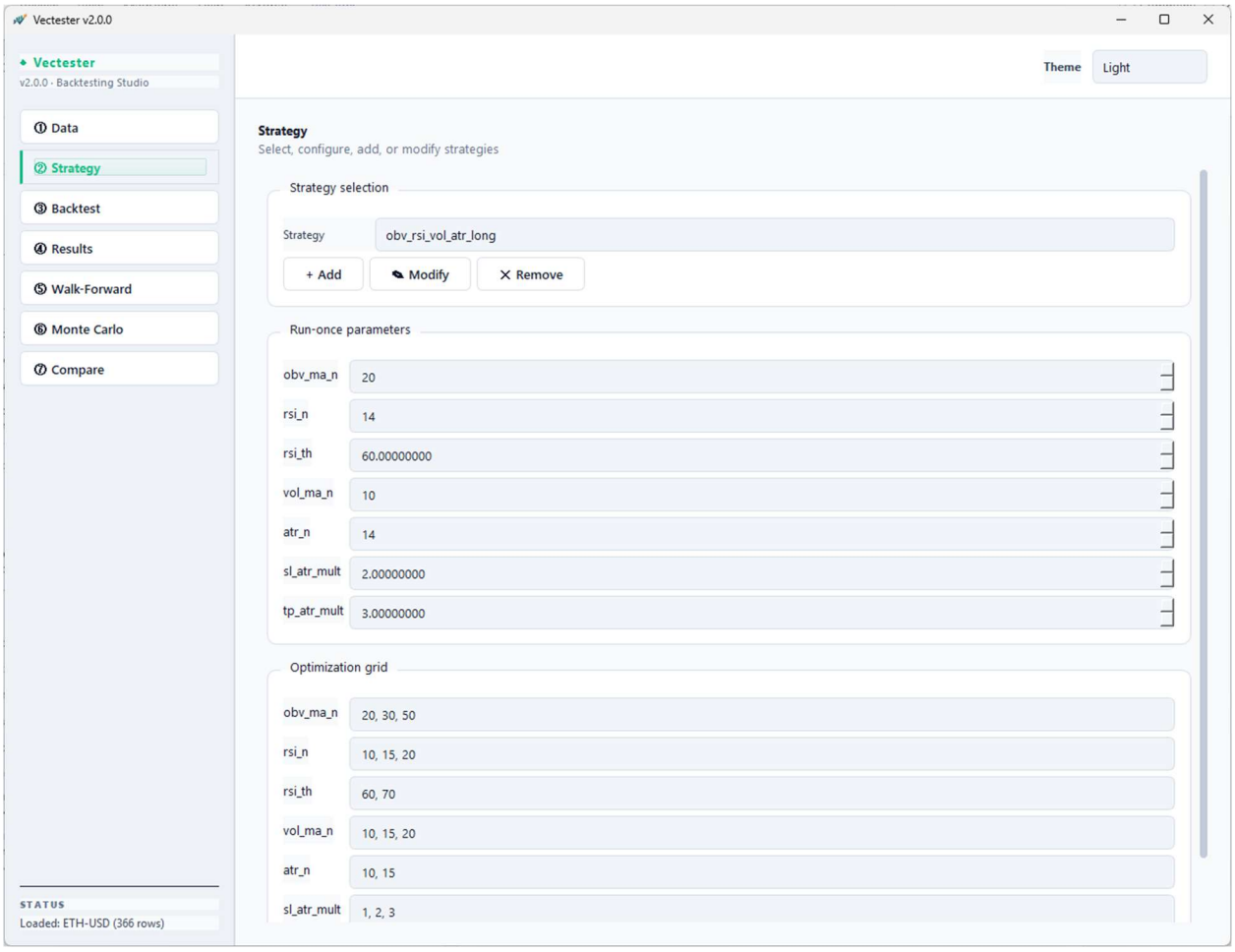
This page allows you to:

- choose the data source
- enter a symbol or symbols
- set the historical period and interval where applicable
- browse to a local file when using file-based data
- preview the loaded dataset before testing

The page includes a preview table that shows the first rows of the loaded data so you can verify that the dataset looks correct before moving on. This is especially useful for checking date formatting, OHLC columns, and general data integrity.

In normal use, once data loads successfully, the application automatically advances to the **Strategy** page so the research flow continues smoothly.

② Strategy



The **Strategy** page is one of the most important pages in the application. It is the control center for selecting, configuring, creating, editing, and removing strategies.

This page allows you to:

- choose a built-in or custom strategy from the strategy list
- view and edit the strategy's run-time parameters
- define optimization values for each parameter
- add a new strategy
- modify an existing strategy
- remove a strategy

The page is divided into three main working areas:

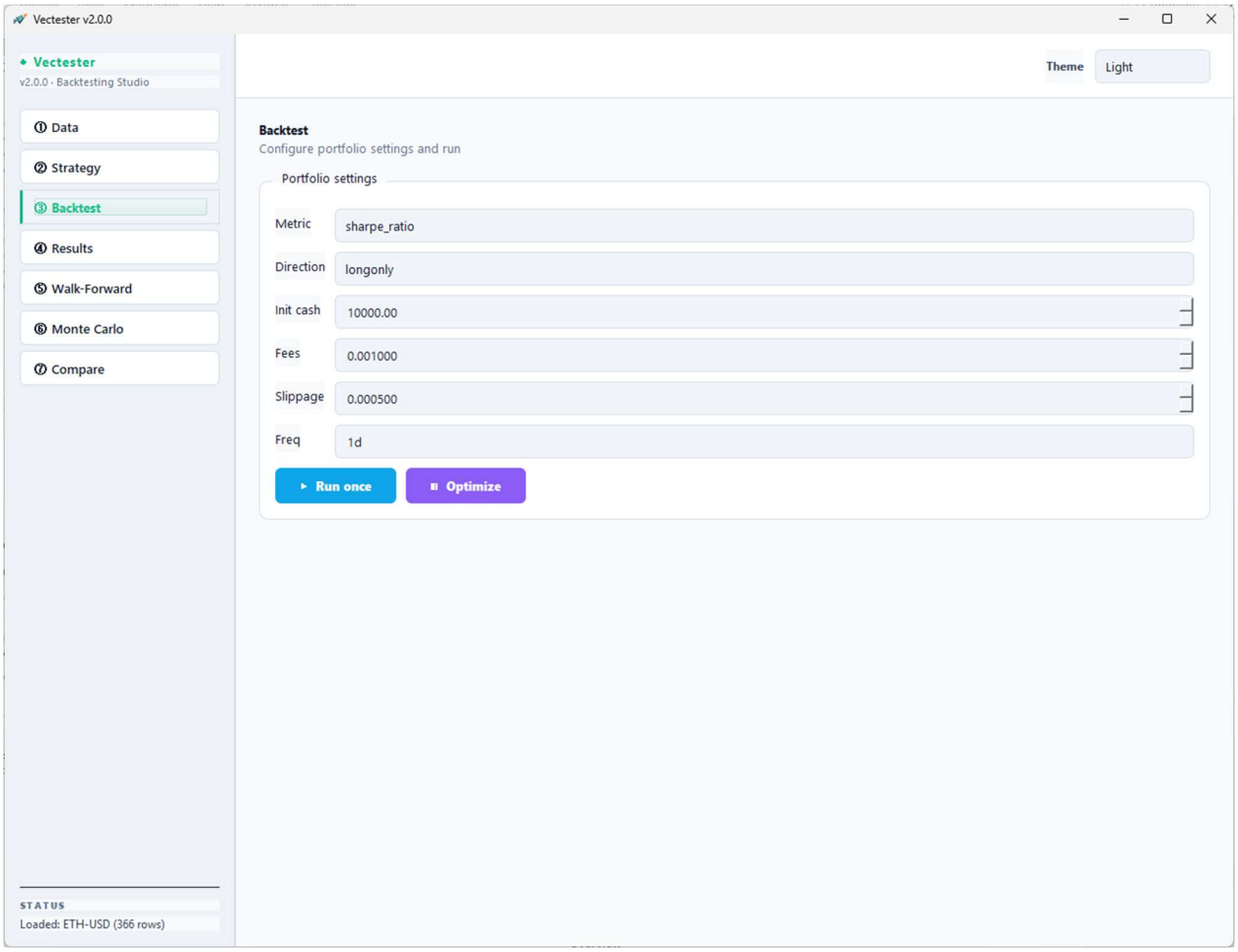
Strategy selection area
This is where you pick the active strategy from the dropdown list.

Run-once parameter area
This section shows the strategy's normal input fields. These are the values used when you run a single backtest.

Optimization grid area
This section shows the same parameters again, but in a format intended for parameter sweeps. Instead of entering one value, you enter one or more comma-separated values so Vectester can test many combinations during optimization.

Because the interface is generated from each strategy's declared parameter definitions, the Strategy page adapts automatically when you switch strategies. Different strategies may show completely different inputs depending on how they were designed.

③ *Backtest*



The **Backtest** page controls the portfolio-level settings and the two main execution modes: a single run or a full optimization.

This page allows you to:

- choose the evaluation metric
- define direction mode
- set initial cash
- set fees
- set slippage
- set data frequency
- run a single backtest
- run a parameter optimization

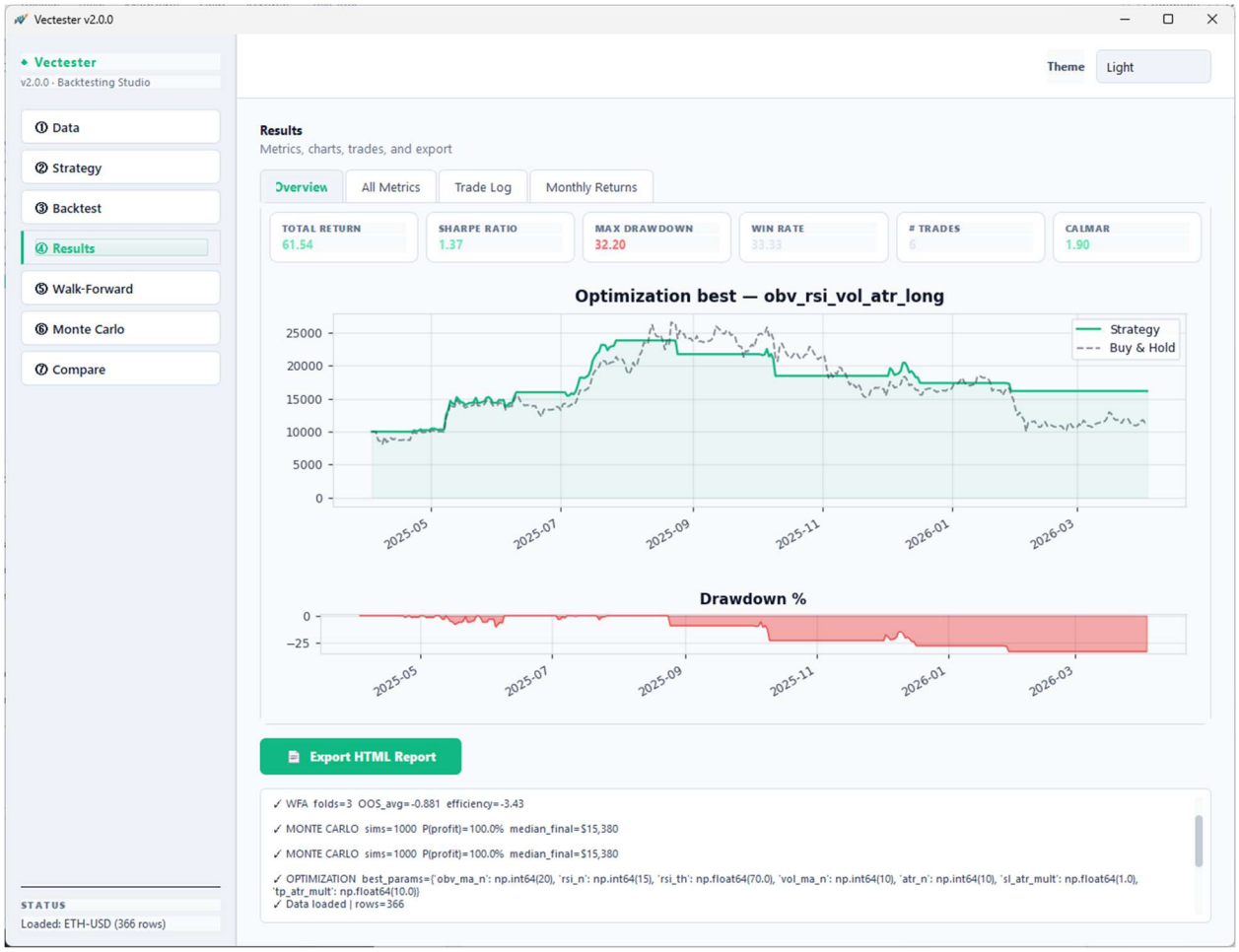
This page answers the question: *How should the strategy be evaluated as a trading system?* While the Strategy page defines the trading logic, the Backtest page defines the portfolio assumptions under which that logic will be tested.

Two main actions are available here:

Run once
Runs the currently selected strategy using the current parameter values.

Optimize
Runs a grid search across the parameter values defined on the Strategy page and selects the best-performing combination according to the chosen metric.

④ Results



The **Results** page is where the output of a backtest or optimization is displayed in full detail.

This page includes four result views:

- **Overview**
- **All Metrics**

- **Trade Log**
- **Monthly Returns**

The Overview tab presents the most important information first:

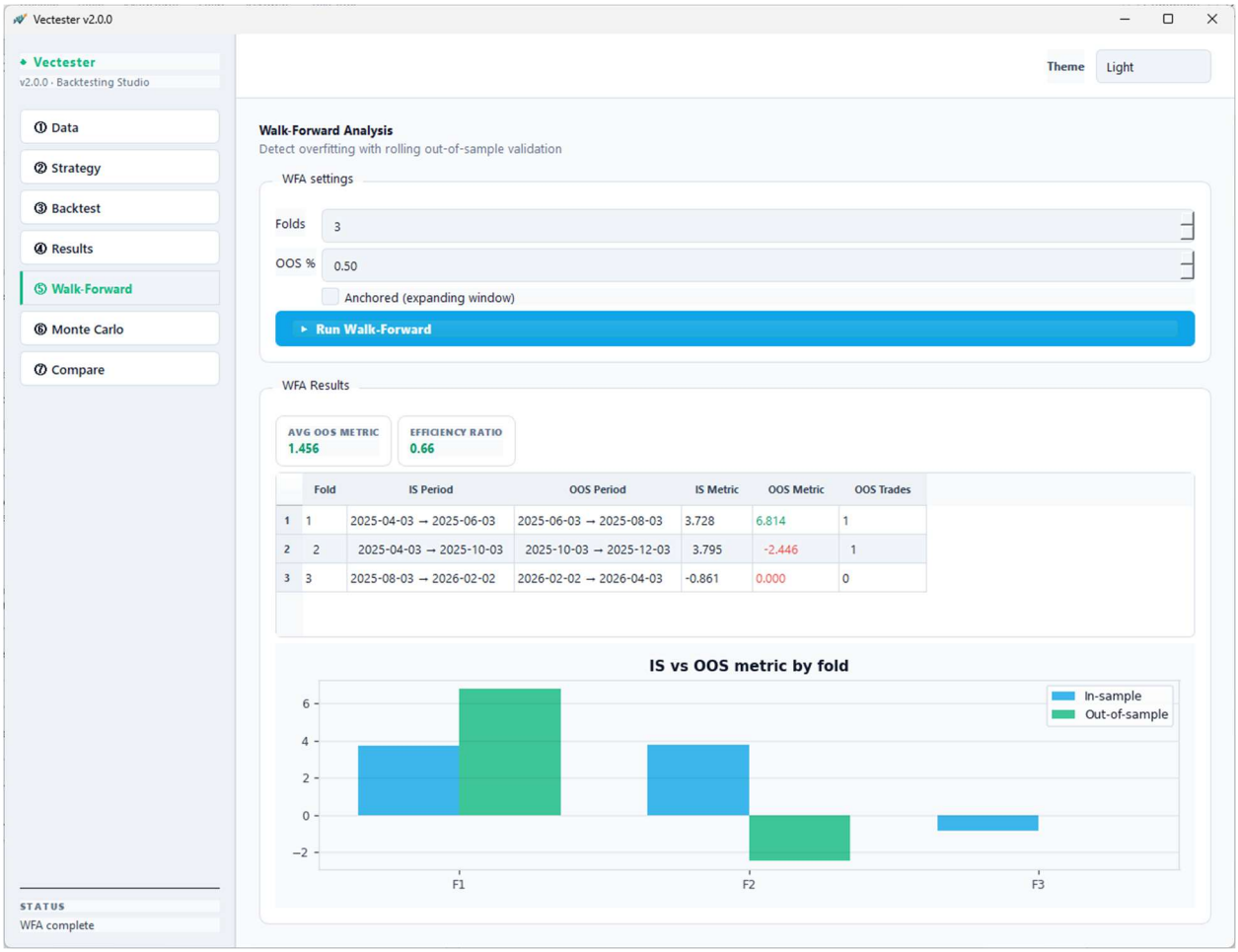
- key metric cards
- equity curve
- drawdown chart

The other tabs allow deeper inspection:

- the full statistics table
- the individual trade list
- monthly return behavior

This page also includes the **Export HTML Report** action and the application log output area. In practice, it becomes the main reference page once you start testing strategies.

⑤ *Walk-Forward*



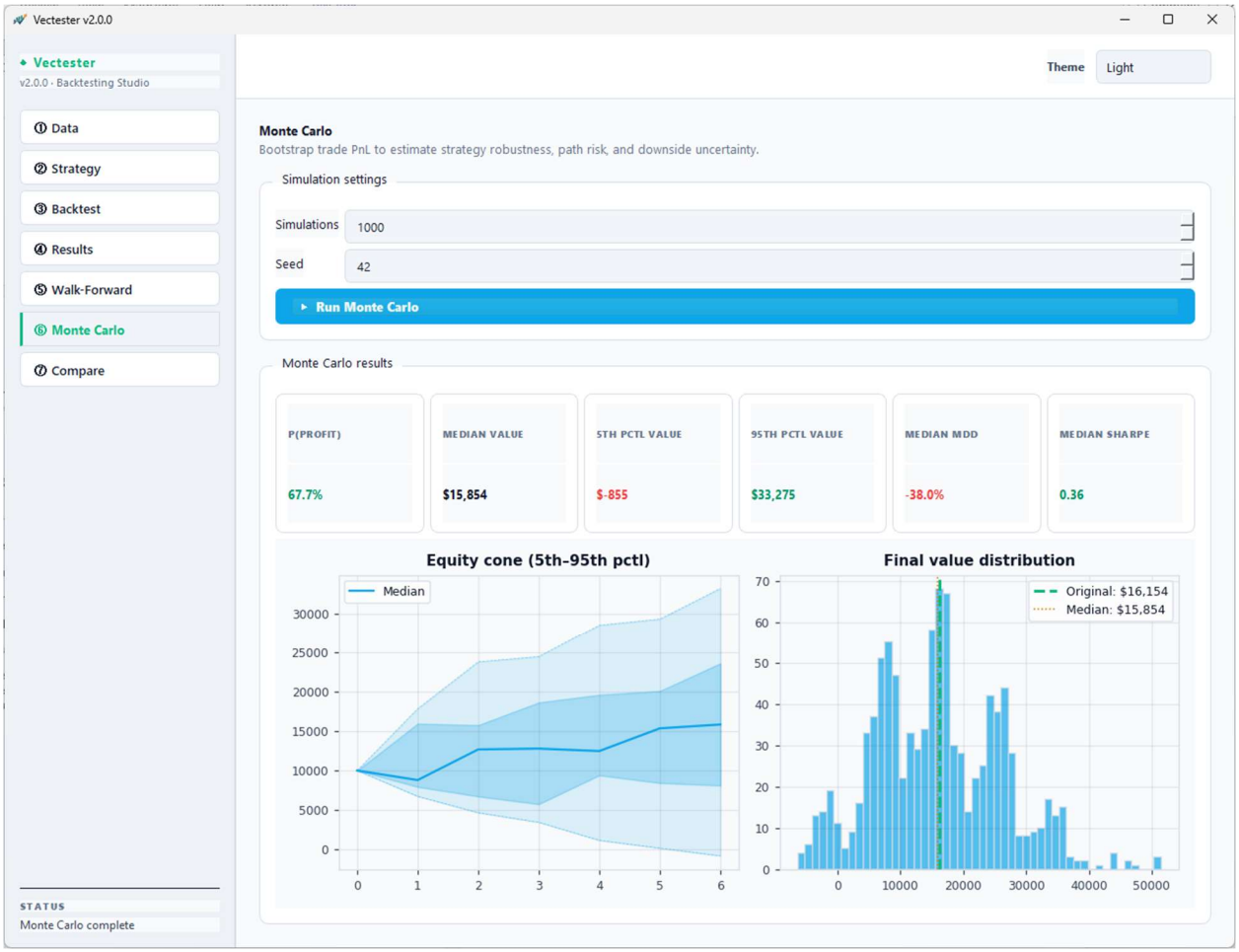
The **Walk-Forward Analysis** page is used to test how well an optimized strategy survives out-of-sample validation.

This page allows you to:

- set the number of folds
- set the out-of-sample percentage
- choose anchored or rolling analysis
- run walk-forward analysis
- review fold-by-fold validation results
- inspect the in-sample versus out-of-sample performance chart

This page is intended for robustness checking, not for initial idea discovery. After finding a promising strategy and parameter set, this page helps determine whether the result is likely to be real or merely overfit to the historical sample.

⑥ *Monte Carlo*



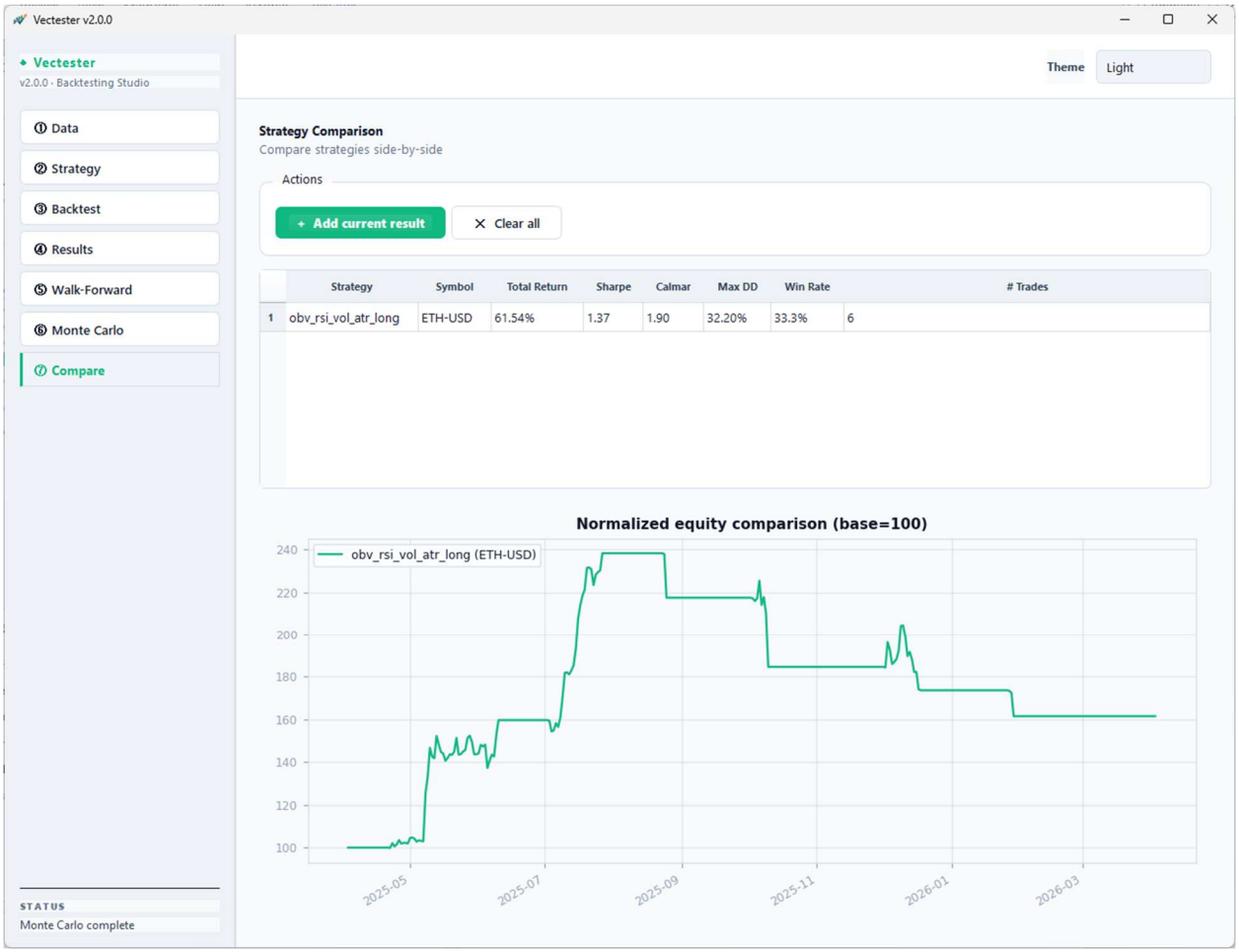
The **Monte Carlo** page estimates how sensitive the backtest result is to trade ordering and path dependency.

This page allows you to:

- set the number of simulations
- set the random seed
- run Monte Carlo analysis on the most recent backtest result
- inspect probability and percentile metrics
- review the equity-cone style chart and final-value distribution

This page is used after a backtest has already been completed. It does not create trades by itself. Instead, it takes the most recent strategy result and resamples its trade outcome sequence to help you judge how much luck may be embedded in the observed result.

⑦ *Compare*



The **Compare** page is used for side-by-side evaluation of different strategy runs.

This page allows you to:

- add the current result to the comparison set
- build up a list of multiple strategies or parameter variants
- compare them in a ranking table
- inspect a normalized equity comparison chart
- clear the comparison set

This page is especially useful when:

- comparing built-in strategies on the same symbol
- comparing parameter variations of the same strategy
- comparing a custom strategy against a benchmark or a simpler alternative

The comparison view helps prevent decision-making based only on memory or impression. It puts competing ideas into one structured view.

2.2 Typical Research Workflow

Vectester is flexible, but the intended workflow follows a clear research sequence. That sequence helps reduce mistakes and keeps the testing process disciplined.

Step 1: Load data

Begin on the **Data** page by choosing a source and loading a dataset. Before proceeding, confirm that:

- the symbol is correct
- the date range is appropriate
- the timeframe matches the strategy idea
- the preview table looks reasonable

Good testing starts with good data. If the underlying data is wrong, incomplete, or too short, every later result becomes unreliable.

Step 2: Select or create a strategy

Go to the **Strategy** page and choose the strategy you want to test. If the built-in strategies do not match your idea, use the strategy editing tools to create a new one or modify an existing one.

At this stage, you should:

- understand what the strategy is supposed to do
- review the parameters carefully
- set sensible run-once defaults
- prepare an optimization grid only if you are ready to explore parameter sensitivity

A strong workflow starts with clear logic, not with brute-force optimization.

Step 3: Configure portfolio assumptions

Move to the **Backtest** page and set the portfolio-level assumptions:

- direction
- initial cash
- fees
- slippage
- frequency
- optimization metric

These settings matter because they change how performance is measured. A strategy that looks attractive with zero friction may weaken materially once realistic fees and slippage are applied.

Step 4: Run a baseline test

Use **Run once** before doing any optimization. This gives you a clean first look at how the strategy behaves with the current parameter values.

This step is important because it tells you:

- whether the strategy is working at all
- whether it produces trades
- whether the logic is broadly sensible
- whether the resulting behavior is worth deeper study

A baseline run often reveals basic problems quickly, such as no entries, too many entries, unstable exits, or unrealistic performance.

Step 5: Review the result in detail

After a run completes, Vectester moves to the **Results** page. Review more than just the headline return.

A proper review should include:

- total return
- risk-adjusted metrics
- drawdown depth
- number of trades
- trade distribution

- monthly consistency
- trade log plausibility

A result is only useful if it is understandable. Extremely high returns with few trades, huge drawdowns, or unstable monthly behavior may be weaker than they first appear.

Step 6: Optimize carefully

If the baseline result is promising, return to the **Strategy** and **Backtest** pages and run an optimization.

Good practice is to:

- start with small, focused parameter ranges
- vary only a few inputs at a time
- avoid enormous grids too early
- choose an evaluation metric that matches your objective

Optimization should be used to explore reasonable variants, not to blindly search for the best-looking number.

Step 7: Validate with walk-forward analysis

Once a promising parameter set has been found, use the **Walk-Forward** page to test whether performance survives out-of-sample.

This is a critical step because optimization alone does not prove robustness. A strategy that performs well only on the sample it was tuned on may fail badly in forward use.

Step 8: Test robustness with Monte Carlo

After obtaining a completed backtest, use the **Monte Carlo** page to see how sensitive the result is to trade ordering.

This helps answer questions such as:

- was the observed outcome unusually lucky?
- how wide is the plausible range of outcomes?
- how severe could drawdowns become under different trade sequences?
- how stable is the apparent edge?

A strategy with a good backtest but weak Monte Carlo characteristics may be less dependable than expected.

Step 9: Add strong candidates to Compare

Whenever a result is worth keeping, add it to the **Compare** page. Repeat this for other strategies or parameter sets.

This allows you to compare:

- aggressive versus conservative variants
- simple versus complex strategies
- built-in versus custom ideas
- alternative instruments or timeframes

Comparison helps impose discipline. Instead of remembering impressions from separate runs, you judge candidates side by side.

Step 10: Export the final report

Once you have a result worth documenting, export it from the **Results** page as an HTML report.

This is useful for:

- record keeping
- sharing findings
- reviewing results later
- documenting research decisions

Practical workflow summary

A typical high-quality session looks like this:

1. Load data
2. Select or create a strategy
3. Set run parameters
4. Run a baseline backtest
5. Review the result carefully
6. Optimize if justified
7. Validate with walk-forward analysis
8. Stress-check with Monte Carlo

9. Compare against alternatives
10. Export the best result

This order helps reduce overfitting and encourages evidence-based strategy development.

2.3 Status Messages, Logs, and Progress Indicators

Vectester provides three main forms of feedback while you work:

- the **status area** in the sidebar
- the **progress indicator**
- the **log output panel**

Together, these help you understand what the application is doing, whether an action succeeded, and where a problem occurred if something goes wrong.

Status area

At the bottom of the left sidebar, Vectester displays a **STATUS** label followed by the current status text.

This status text changes during major actions. Typical examples include:

- `Ready`
- `Loading data...`
- `Running backtest...`
- `Running optimization...`
- `Running walk-forward analysis...`
- `Running Monte Carlo...`
- `Backtest complete`
- `Optimization complete`
- `WFA complete`
- `Monte Carlo complete`
- `Error`

The status area gives a quick high-level summary of the current application state. It is especially useful during longer actions such as optimization, walk-forward analysis, or Monte Carlo simulation.

Progress indicator

Directly beneath the status text is a thin progress bar that appears only while the application is busy.

In practice, this acts as a busy indicator rather than a percentage-complete tracker. It shows that an operation is in progress, but it does not report exact completion percentage or time remaining.

You will typically see it during:

- data loading
- single backtests
- optimization
- walk-forward analysis
- Monte Carlo simulation

Once the action finishes, the progress bar disappears.

Its purpose is simple but important: it confirms that the application is actively working and has not stalled.

Log output panel

The **log output** appears on the Results page beneath the export controls. This is the detailed event record of the current session.

The log records important operations such as:

- successful data loads
- run-once execution
- optimization results
- walk-forward completion
- Monte Carlo completion
- comparison actions
- report export success
- errors and trace details

The log uses visible prefixes that help distinguish message types:

Success messages
These typically begin with a check mark and describe a completed action. Examples include:

- data loaded
- run once completed
- optimization completed
- walk-forward completed
- result added to comparison
- report saved

Warnings or load issues
These may appear when strategies fail to load or when expected resources are missing.

Error messages
These begin with an error mark and usually include the exception information. In serious cases, the detailed traceback is also written to the log.

This makes the log one of the most important troubleshooting tools in the application.

How to use the log effectively

The log should be checked whenever:

- a strategy does not behave as expected
- a run produces no result
- optimization returns nothing useful
- a custom strategy fails to load
- an export does not complete
- comparison actions do not behave as expected

For custom strategy development, the log is especially important because it often reveals:

- syntax problems
- import problems
- invalid parameter handling
- runtime exceptions

- failures during optimization or validation

Relationship between status and log

The **status area** tells you the current state in brief.
The **log output** tells you what happened in detail.

For example:

- the status may say `Running backtest...`
- after completion, it may change to `Backtest complete`
- the log will then show the strategy parameters used and the resulting metric value

This combination is useful because it separates immediate operational feedback from detailed historical trace information.

Error handling behavior

When an operation fails, Vectester typically does three things:

1. changes the status to `Error`
2. stops the busy indicator
3. records the error details in the log

In some cases, it also displays a pop-up message box, especially for more direct user-facing failures such as data-loading errors or missing prerequisites.

This layered feedback makes it easier to identify both:

- what failed
- where to start fixing it

Best practice

When working with Vectester, treat the status area and the log as part of the analysis workflow, not just as technical extras.

A disciplined user should:

- watch the status before and after each major action
- check the log after every optimization or validation run
- review any error text before rerunning
- use the log to diagnose problems in custom strategies

This habit saves time and makes research more reliable, especially when strategy logic becomes more complex.

3. Loading Market Data

The **Data** page is the starting point of every session in Vectester. Before any strategy can be tested, optimized, validated, or compared, the application must load a market dataset into memory. Vectester accepts market data from online sources and local files, normalizes that data into a common OHLCV structure, and displays a preview so you can confirm that the dataset is usable before moving on to strategy work.

In practice, this means the Data page serves two purposes. First, it imports price history. Second, it acts as a validation gate: if the data is incomplete, badly formatted, or missing required fields, Vectester will stop and report the problem before you waste time running a bad backtest.

3.1 Supported data sources

Vectester supports four data sources:

|Source|Typical use|Notes| |||| |**Yahoo Finance**|Fast access to stocks, ETFs, indices, forex proxies, and crypto tickers supported by Yahoo|Best choice for quick research and prototyping| |**CSV file**|Importing your own historical datasets|Useful when you already have exported broker, exchange, or custom data| |**Parquet file**|Importing larger local datasets efficiently|Better than CSV for speed and storage when working with large files| |**Binance data**|Pulling market data for Binance trading pairs|Useful for crypto workflows using exchange-formatted symbols|

All four sources are brought into the same internal format so the rest of the application can work consistently. Regardless of source, Vectester expects the resulting dataset to represent standard price bars with at least:

- **Open**
- **High**
- **Low**
- **Close**

Volume is optional for file-based imports, but it is preferred because some strategies use volume-based logic. If a local file does not contain a Volume column, Vectester fills it with zeros so the dataset can still load.

3.2 Yahoo Finance

Yahoo Finance is the default and most convenient source for most users. It is ideal when you want to quickly load a symbol and start testing without preparing a separate file.

Symbol(s)

The **Symbol(s)** field accepts Yahoo-style tickers such as:

- `AAPL`
- `MSFT`
- `SPY`
- `BTC-USD`
- `ETH-USD`
- `^GSPC`

You can also enter multiple symbols separated by commas.

Example:

```
BTC-USD, ETH-USD, SOL-USD
```

This lets Vectester download several datasets in one step. However, for actual backtesting, the application uses the **first loaded symbol as the active dataset**. The additional symbols are loaded and stored, but they do not create a true multi-asset portfolio test on the main backtest run. This distinction is important and should be understood clearly.

Period

The **Period** field controls how much historical data Yahoo Finance should return. Common examples include:

- `1d`
- `5d`
- `1mo`
- `3mo`
- `6mo`
- `1y`
- `2y`
- `5y`
- `10y`
- `ytd`

- `max`

Shorter periods are useful for intraday testing or recent behavior. Longer periods are usually better for daily-system research because they provide more trades and more varied market conditions.

Interval

The **Interval** field controls the bar size. Examples include:

- `1m`
- `2m`
- `5m`
- `15m`
- `30m`
- `60m`
- `90m`
- `1h`
- `1d`
- `5d`
- `1wk`
- `1mo`
- `3mo`

Smaller intervals create more bars and may increase noise, runtime, and overfitting risk. Daily data is usually the best starting point unless the strategy is explicitly intraday.

Practical guidance

Yahoo Finance is best when you want to:

- test an idea quickly
- compare several symbols
- avoid preparing local files
- build a first research baseline

It is less suitable when you need exact exchange-native data quality, custom preprocessing, or institutional-grade historical archives.

3.3 CSV files

CSV import is the most flexible option when you already have your own historical data.

To use it, select **CSV file** as the source, then browse to a `.csv` file on disk. Vectester reads the file and interprets:

- the **first column** as the datetime index
- the remaining columns as market data fields

Required columns

A CSV file must contain these columns, in any capitalization:

- `Open`
- `High`
- `Low`
- `Close`

These are matched **case-insensitively**, so all of the following are acceptable:

- `Open`
- `open`
- `OPEN`

If the CSV also contains `Volume`, it will be used. If not, Vectester creates a Volume column filled with zeros.

CSV expectations

A good CSV file should look conceptually like this:

```
Date,Open,High,Low,Close,Volume
2024-01-01,100,105,99,104,150000
2024-01-02,104,106,101,102,120000
2024-01-03,102,108,101,107,180000
```

The date column should be the first column, because Vectester uses the first column as the index automatically.

When CSV is useful

CSV is the right choice when you want to:

- import broker-exported data
- use cleaned or custom-built datasets
- test markets not easily available through Yahoo Finance
- maintain your own data pipeline outside the app

Its main downside is that formatting errors are common, so users must pay attention to headers and date structure.

3.4 Parquet files

Parquet import is similar to CSV import but is better suited for larger datasets and more efficient storage.

To use it, select **Parquet file** as the source and browse to a `.parquet` or equivalent compatible file.

Column requirements

Parquet files follow the same logical requirements as CSV files. They must contain:

- `Open`
- `High`
- `Low`
- `Close`

`Volume` is optional. If it is missing, Vectester inserts a zero-filled Volume column.

Column matching for the required OHLC fields is also handled case-insensitively.

Why use Parquet

Parquet is usually preferable when:

- datasets are large
- loading speed matters
- you store many symbols or long histories
- you want a cleaner analytical workflow than plain CSV

For end users, the practical difference is simple: if you already have data in Parquet format, Vectester can use it directly without needing conversion.

3.5 Binance data

Vectester also supports loading market data for Binance trading pairs.

This source is intended for crypto users who want exchange-style pairs such as:

- `BTC/USDT`
- `ETH/USDT`
- `SOL/USDT`

Symbol format

Unlike Yahoo Finance, Binance uses exchange pair notation. That means the symbol should be entered as:

`BASE/QUOTE`

Examples:

- `BTC/USDT`
- `ETH/USDT`
- `BNB/USDT`

Interval

The **Interval** field is used as the Binance timeframe. Common examples include:

- `1h`
- `4h`
- `1d`

Limit (bars)

The **Limit (bars)** control determines how many candles Vectester requests from Binance. This is a direct bar count rather than a “period” expression.

For example:

- `365` with `1d` requests approximately one year of daily bars
- `1000` with `1h` requests 1000 hourly bars

This source is useful when you want exchange-oriented crypto data without relying on Yahoo ticker mappings.

3.6 Data requirements

Regardless of source, Vectester expects the dataset to satisfy several structural requirements.

1. OHLC fields must exist

At minimum, the dataset must contain:

- **Open**
- **High**
- **Low**
- **Close**

Without these, the app cannot build a valid price series for backtesting.

2. Close values must be usable

Rows with missing **Close** values are removed during loading. This is important because the Close series is central to indicator calculations, trade generation, and portfolio valuation.

3. Datetime indexing must make sense

For file-based imports, Vectester assumes the first column represents the time index. If the first column is not actually the date column, the data may load incorrectly or produce confusing results.

4. Volume is optional, but not always irrelevant

Some strategies depend on volume-based conditions. If your file has no Volume column, the app will still load it by inserting zeros, but any strategy that relies on real volume information may behave poorly or become misleading.

5. Clean numeric values are strongly preferred

Columns should contain valid numeric values. Mixed text, malformed cells, or inconsistent formatting can lead to load errors or distorted results.

6. Bars should be chronologically consistent

The application assumes the dataset is a normal ordered time series. If dates are duplicated, missing in odd ways, or out of order, strategy logic and chart interpretation may become unreliable.

3.7 Data preview

After loading data successfully, Vectester fills the **Preview** table on the Data page.

The preview is meant as a quick visual verification step before you proceed to strategies and backtesting.

What the preview shows

The preview displays:

- the **first 50 rows** of the dataset
- up to **12 columns**
- the row index as the vertical labels
- the detected data columns as the horizontal headers

This lets you confirm, at a glance:

- the data loaded correctly
- the index looks like time
- OHLC values appear reasonable
- Volume is present or zero-filled as expected
- there are no obvious formatting issues

Why the preview matters

Users often skip this step, but it is one of the most valuable sanity checks in the entire workflow. A few seconds spent looking at the preview can reveal:

- wrong file selected
- missing date parsing
- incorrect column mapping
- empty rows
- suspicious prices
- broken exports from another platform

After a successful load, Vectester also updates its internal active dataset and automatically advances the workflow toward the next stage.

3.8 Common data issues

Most loading problems come from formatting mistakes, symbol mistakes, or misunderstandings about what the app expects.

Invalid or unsupported symbol

If a Yahoo Finance or Binance symbol is typed incorrectly, the app may fail to download data or return an empty result.

Examples:

- `BTCUSD` may fail where `BTC-USD` is required for Yahoo Finance
- `BTCUSDT` may fail where `BTC/USDT` is required for Binance

Always use the symbol format expected by the selected source.

Missing OHLC columns

For CSV and Parquet imports, the most common error is missing one or more required columns:

- Open
- High
- Low
- Close

Even if the data file contains prices, the column names must still map correctly.

Wrong first column in CSV

Vectester uses the first column of a CSV as the index. If the first column is not the date column, the imported dataset may end up with a wrong or meaningless index.

Missing Volume

A file can still load without Volume, but strategies that depend on true volume behavior may become unreliable because the inserted values are zeros rather than real traded volume.

Bad date formatting

If the date column is malformed, inconsistent, or not actually a datetime field, the resulting time series may be hard to interpret or may not behave properly in charts and analysis.

Empty or sparse data

A file may technically load but still be too short or too incomplete for meaningful research. Very small datasets often lead to:

- too few trades

- unstable optimization
- misleading performance metrics
- weak walk-forward and Monte Carlo results

Intraday limitations from the source

When using online sources, some intervals are only available for limited history ranges. This is a source limitation, not a Vectester problem. If you request too much intraday history, the provider may return less data than expected.

Mixed or dirty local data

If a CSV or Parquet file contains text values inside numeric columns, duplicated headers, extra blank rows, or export artifacts from spreadsheet software, the app may either fail to load the file or load it in a distorted form.

Multi-symbol misunderstanding

When entering multiple Yahoo symbols, users may assume the app will backtest them together as one combined portfolio. In the current workflow, that is not what happens. The first loaded symbol becomes the active dataset for the main run, while the others are only stored as separately loaded data.

A good practical rule is this: before testing a strategy, verify that the loaded dataset has the correct symbol, correct timeframe, valid OHLC columns, sensible dates, and enough history to support the kind of analysis you plan to run.

4. Loading Market Data and the Strategy System

This chapter explains two parts of Vectester that define the entire workflow of the application: how market data is loaded, and how strategies are defined, discovered, displayed, and executed.

Market data is the input. A strategy is the rule set that acts on that data. Once data is loaded and a strategy is selected, Vectester can generate signals, run a backtest, optimize parameters, and validate results with walk-forward analysis and Monte Carlo simulation.

4.0 Loading Market Data

Before any strategy can be tested, Vectester needs a market data set in standard OHLCV form. In practical terms, this means price bars containing at least:

- **Open**
- **High**
- **Low**
- **Close**

Volume is optional, but some strategies use it. If a strategy relies on volume-based filters or indicators, missing or poor-quality volume data can materially affect results.

Vectester supports multiple sources for loading data:

- **Yahoo Finance**
- **CSV files**
- **Parquet files**
- **Binance via CCXT**

After loading, the data is normalized into a common internal format so strategies can work with it consistently.

4.0.1 What the application expects

The strategy engine works on a table of time-indexed market bars. In user terms, that means:

- each row represents one bar or candle
- the index represents the date/time
- the price columns must be present and readable

- the data must be ordered correctly in time
- missing values, especially in `Close`, can invalidate analysis

Most strategies use the `Close` series heavily. Many also use `High`, `Low`, and `Volume`. If any of these are missing or malformed, a strategy may either fail or produce misleading signals.

4.0.2 What happens after data is loaded

Once the data is loaded successfully:

- it becomes the active working data set
- the preview table displays the first rows
- the Strategy page can use it for backtesting
- all strategy calculations will reference that loaded data

If no data has been manually loaded, the application may attempt to fetch Yahoo Finance data automatically using the current symbol, period, and interval fields when a run is started.

4.0.3 Important practical note about multiple symbols

Vectester allows multiple symbols to be entered for Yahoo Finance loading, but that does **not** mean every analysis automatically becomes a true multi-asset portfolio test.

In this build, loading multiple Yahoo symbols mainly helps with data collection and later comparison workflows. The backtest engine still operates on a single active data set for a given run. Users should therefore treat each run as strategy-on-one-data-set unless a future version explicitly changes that behavior.

4.1 What a strategy is in Vectester

In Vectester, a **strategy** is a self-contained rule set that tells the application:

- when to enter a trade
- when to exit a trade
- which user-adjustable parameters the strategy exposes
- whether special portfolio instructions should be passed to the backtest engine

A strategy is not just a name in a dropdown. It is a structured object with a defined format that Vectester understands.

At its core, each strategy provides three things:

1. a **strategy name**
2. a list of **parameter definitions**
3. a **build method** that converts market data and parameter values into executable trading signals

Conceptually, the flow is:

market data → strategy logic → entry/exit signals → portfolio simulation → metrics and charts

This design is what makes Vectester extensible. The application does not need hard-coded forms for each strategy. Instead, it reads each strategy's declared structure and builds the interface dynamically.

What a strategy produces

A valid strategy ultimately produces a result containing:

- **entries**: bars where a position should be opened
- **exits**: bars where a position should be closed, if explicit exits are used
- **portfolio keyword arguments**: optional trade-management instructions such as stop-loss, take-profit, or trailing-stop behavior

This separation is important. A strategy can manage exits in two different ways:

- by generating explicit exit signals
- by returning stop-related settings for the portfolio engine to enforce automatically

That gives strategy authors flexibility. Some strategies are purely signal-based. Others are signal-plus-risk-management systems.

Why this matters for users

For users of the app, this means every strategy in Vectester is more than a preset. It is a formal, editable trading model. That is why strategy creation and modification are core features rather than side utilities.

4.2 How strategies appear in the app

In the application, strategies appear primarily through the **Strategy page**.

When a strategy is selected, Vectester immediately uses that strategy's definition to populate two user-facing sections:

- the **run-once parameter form**

- the **optimization grid form**

This is one of the most important design features in the app. The interface is not manually built per strategy. It is generated automatically from the strategy's declared parameter definitions.

What the user sees

For a selected strategy, the app shows:

- the strategy name in the dropdown
- one input control per declared parameter in the run-once section
- one grid input per declared parameter in the optimization section

The type of control depends on the parameter type:

- **integer parameters** appear as numeric spin boxes
- **float parameters** appear as decimal spin boxes
- **boolean parameters** appear as True/False selectors
- other parameter types fall back to text input

So if a strategy declares parameters like `fast`, `slow`, `sl`, and `tp_ratio`, those become editable fields in the interface automatically.

Why the forms update instantly

When a different strategy is selected, Vectester clears the old parameter widgets and rebuilds the forms from the newly selected strategy's definitions.

That means the Strategy page is always driven by the currently selected strategy's own structure. There is no universal static parameter panel.

What this means in practice

This design has several consequences:

- strategies with more parameters will show larger forms
- very simple strategies may show almost no inputs
- optimization always reflects exactly the same parameter list as run-once testing
- adding or modifying a strategy changes the UI without needing manual form design

This is why Vectester can scale from a trivial buy-and-hold benchmark to complex multi-condition systems without redesigning the interface each time.

4.3 Strategy discovery and loading

Vectester uses a registry-based discovery system to find available strategies.

At startup, and again after strategy reload actions, the application scans the strategy package and identifies all valid strategy classes. A class is considered a valid strategy only if it properly subclasses the base strategy type expected by the app.

How discovery works conceptually

The loading process is:

1. inspect the strategy package
2. import each eligible strategy module
3. look for valid strategy classes
4. read each class's declared strategy name
5. register the strategy so it appears in the app

Any strategy module that fails to import is not silently ignored. Load errors are captured and can be surfaced in the application log.

What makes a strategy discoverable

A strategy must be structured correctly. In practical terms, that means:

- it must live in the expected strategy location
- it must define a valid strategy class
- that class must inherit from the expected strategy base
- the class must be importable without syntax or runtime errors

If any of these conditions are not met, the strategy may not appear in the list.

Why load errors matter

A strategy may fail to load for reasons such as:

- syntax errors
- missing imports
- duplicate strategy names

- invalid class structure
- code that raises an exception during import

When this happens, the application cannot safely expose that strategy to the user. The strategy will be skipped, and the error is recorded.

This is important for manual documentation because from the user's perspective, "the strategy is missing" often really means "the strategy failed to load."

Reloading behavior

After a strategy is edited or a new one is added, Vectester invalidates cached module state, clears previously loaded strategy modules, and rebuilds the registry.

That allows the app to reflect new code without needing a full restart in normal editable workflows.

4.4 Built-in vs user-created strategies

Vectester supports two broad categories of strategies:

- **built-in strategies**
- **user-created strategies**

Built-in strategies

Built-in strategies are the ones that ship with the application. They provide ready-made examples and usable starting points. These strategies are meant to help users:

- understand the system
- test known logic styles
- compare their own ideas against reference implementations
- learn how parameterized strategies are structured

Built-in strategies usually cover a range of styles such as:

- trend-following
- momentum
- mean reversion
- volatility filtering
- stop-managed systems

- composite multi-indicator systems

User-created strategies

User-created strategies are custom strategies written or modified by the user through the strategy editing workflow. These are where Vectester becomes a research platform rather than just a strategy viewer.

A user-created strategy can:

- implement new entry logic
- implement new exit logic
- expose custom parameters
- use different combinations of filters and indicators
- define portfolio-level stop behavior

This is the most important extensibility feature of the application.

The practical distinction inside the app

From the user's perspective, built-in and user-created strategies may look similar in the dropdown, but they differ in how safely they can be edited or removed.

The app checks whether the underlying strategy module is editable in the current environment. If it is not, editing and removal actions are disabled or blocked.

In other words:

- some strategies are packaged assets
- some strategies are file-backed editable modules
- the app decides whether **Modify** and **Remove** are allowed based on whether the corresponding strategy file is actually editable in the current installation context

Why this distinction matters in the manual

A proper user manual must explain that Vectester is not only for selecting strategies. It is also for building a personal strategy library. That library can grow over time, and the platform is designed to regenerate its forms and test flow directly from those authored strategies.

4.5 Strategy names and file names

In Vectester, **strategy names** and **file names** are related, but they are not the same thing.

This distinction matters because it affects how strategies are displayed, discovered, edited, and identified internally.

File name

The file name identifies the physical strategy file in the strategy folder. It determines the module name used when the application imports that strategy.

For a new strategy, the file name must be a valid Python-style module name. That means it must follow identifier rules and cannot be arbitrary text.

Strategy name

The strategy name is the user-facing identity of the strategy inside the app. This is the name shown in the strategy dropdown.

Vectester uses the class attribute called `name` as the strategy's registry key. If that value exists, it takes priority over the class name.

So the visible strategy name is usually:

- the declared `name` field in the strategy class, or
- the class name if no explicit name is provided

Why duplicate names are a problem

Two different files can still declare the same strategy name. When that happens, the registry encounters a collision.

Vectester treats duplicate strategy names as a loading problem. Only unique strategy names can be registered safely, because the dropdown and strategy map need a single unambiguous key per strategy.

So users must understand:

- file names should be valid and clean
- strategy names should be descriptive
- strategy names must be unique across the entire strategy library

Recommended naming practice

A good strategy file name should be stable and code-friendly. A good strategy name should be clear and readable.

For example:

- file name: `ema_pullback_long.py`

- strategy name: `ema_pullback_long`

or, if desired, a more human-readable internal convention can still be used so long as the application can register it cleanly.

The main rule is consistency. The file identifies where the strategy lives. The strategy name identifies what the strategy is inside the app.

4.6 How the app turns strategy definitions into input forms

This is one of the most important mechanisms in Vectester.

The app does not hard-code parameter widgets for each strategy. Instead, it reads the selected strategy's parameter definitions and builds the interface dynamically.

The source of the form: parameter definitions

Each strategy declares a list of parameter definitions. Each parameter definition includes:

- the parameter name
- the default value
- the parameter type
- a description

This declaration acts as a contract between the strategy and the app.

From that single list, Vectester knows:

- which inputs to display
- what each input is called
- what default value it should start with
- what kind of widget should be used

Widget selection rules

When building the run-once form, the app checks each parameter's declared type and chooses the input accordingly.

Integer parameters

Integer parameters become spin boxes with numeric bounds. These are suited for:

- lookback periods
- window sizes

- bar counts
- threshold counts

Examples:

- fast EMA length
- slow EMA length
- RSI period
- ATR period

Float parameters

Float parameters become decimal spin boxes. These are suited for:

- stop-loss percentages
- take-profit ratios
- volatility thresholds
- multiplier values
- fractional cutoffs

Examples:

- `sl = 0.02`
- `tp_ratio = 2.0`
- `atr_mult = 1.5`

Boolean parameters

Boolean parameters become a True/False selector. These are suited for toggles such as:

- enable or disable a filter
- switch between logic variants
- activate trailing-stop behavior

Other parameter types

If the type is not one of the recognized built-in numeric or boolean types, the app falls back to plain text entry.

Run-once form vs optimization form

Vectester creates **two separate interfaces** from the same parameter list.

Run-once form

This is for testing a single parameter set. Each parameter gets one concrete input value.

Optimization form

This is for entering candidate values to search across. In this section, each parameter is shown as a text field containing one or more comma-separated values.

So a parameter definition like:

- `fast`
- `slow`
- `sl`

becomes:

- one run-once input for each
- one optimization-grid input for each

This is a major design strength. It keeps the strategy definition as the single source of truth.

Why this matters

Because the UI is generated from the strategy definition:

- adding a parameter automatically adds new form fields
- removing a parameter automatically removes them
- changing a default value updates both the run-once and optimization starting state
- changing parameter types changes how the UI behaves

This makes Vectester highly maintainable and strategy-centric.

4.7 How strategy parameters drive both testing and optimization

In Vectester, strategy parameters do not just decorate the interface. They drive the entire execution path.

The same declared parameters are used in two different modes:

- **single-run backtesting**

- **grid-based optimization**

That means the strategy's parameter model is shared across both workflows.

Single-run testing

In run-once mode:

1. the user selects a strategy
2. the user enters one value for each parameter
3. the app collects those values from the generated form
4. the app passes them into the strategy
5. the strategy builds entry/exit logic using those values
6. the resulting signals are backtested

This is the cleanest way to test a specific hypothesis.

For example, a user may set:

- `fast = 20`
- `slow = 50`
- `sl = 0.02`

and ask: "How does this exact version behave?"

Optimization

In optimization mode, the same parameters are used differently.

Instead of collecting one value per parameter, the app collects a list of candidate values for each parameter from the grid fields. These values are entered as comma-separated lists.

Example:

- `fast`: `10,20,30`
- `slow`: `50,100`
- `sl`: `0.01,0.02`

Vectester then builds the Cartesian product of those choices, creating every valid parameter combination.

In that example, the combinations would be:

- 3 values for `fast`
- 2 values for `slow`
- 2 values for `sl`

Total: **12 combinations**

Each combination is then tested, and the selected optimization metric is used to rank results.

Why this shared model is powerful

Because the run-once and optimization workflows use the same strategy definition:

- there is no mismatch between what can be tested and what can be optimized
- parameter additions automatically become optimizable
- parameter defaults act as sensible starting points in both workflows
- strategy authors only need to declare parameters once

This avoids one of the most common problems in research tools: separate code paths for testing and optimization that drift apart over time.

Type conversion and parameter meaning

The parameter definition also controls how grid values are interpreted.

For example:

- integer parameters are parsed as integers
- float parameters are parsed as floats
- boolean parameters are interpreted as True/False values

So the strategy definition is not only descriptive. It directly controls how the app parses and executes user input.

Defaults and overrides

When the strategy is run, the app starts from the strategy's default parameter set and then overlays the user-supplied values.

That means:

- defaults define the baseline behavior
- user edits override defaults

- missing fields still fall back to the strategy's own intended baseline

This is useful both for stability and for user understanding. A strategy can be opened and run immediately with sensible defaults even before the user customizes it.

Important implication for strategy authors

If a parameter is declared badly, everything downstream is affected:

- the input form may be awkward
- the default value may mislead users
- optimization grids may parse incorrectly
- the strategy may behave unpredictably

So parameter design is not a cosmetic issue. It is one of the most important parts of strategy authoring in Vectester.

Summary

Vectester's strategy system is built around a simple but powerful principle: the strategy definition is the source of truth.

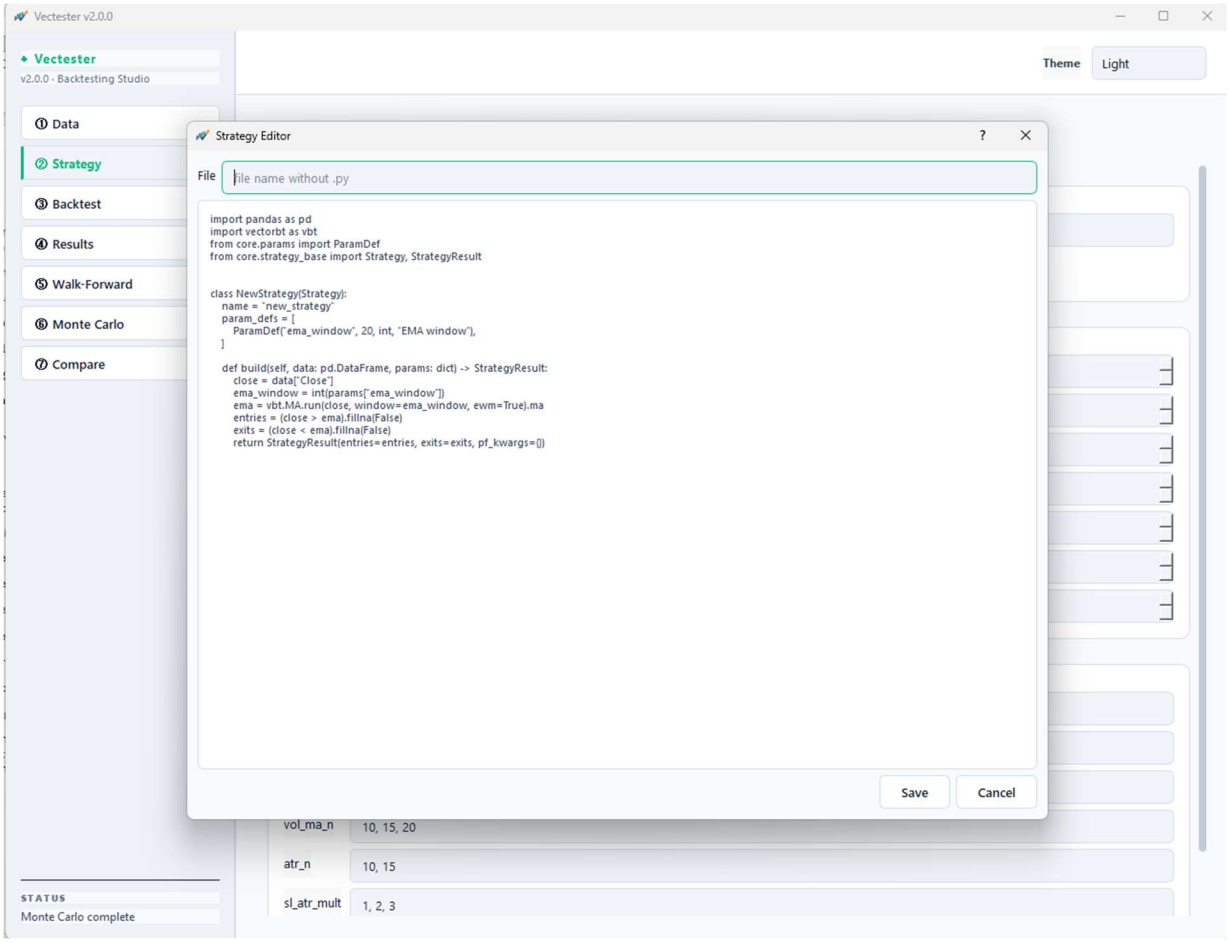
From that one definition, the application determines:

- how the strategy is discovered
- how it is named
- how it appears in the interface
- which parameters the user can edit
- how those parameters are tested
- how they are optimized
- how trading signals are generated

This is why understanding the strategy system is essential to using Vectester properly. The app is not just a backtest runner with a list of presets. It is a strategy-driven research environment in which data, parameters, logic, and validation all begin with the strategy definition.

5. Creating a Strategy

One of Vectester's most important capabilities is that strategies are not fixed. You can create new strategies, study existing ones, modify them, and immediately use them in the application. The strategy system is designed around a simple pattern: each strategy is a Python class that declares its parameters and returns trading signals.



In practice, a strategy in Vectester does three things:

1. It gives the strategy a display name.
2. It declares the parameters the user can edit and optimize.
3. It defines the trading logic that produces entries, exits, and optional portfolio-level controls such as stop-loss or take-profit.

This section explains the full strategy-writing workflow in a user-facing, practical way.

5.1 Opening the Strategy Editor

Vectester includes a built-in **Strategy Editor** dialog for working directly with strategy files.

The editor window contains:

- a **File** field at the top
- a large code editor area
- **Save** and **Cancel** buttons

The **File** field is the Python file name, without the `.py` extension. The main code area is where the strategy source is written or edited.

What the editor is for

The Strategy Editor is used to:

- create a brand-new strategy file
- inspect an existing strategy
- modify an existing strategy
- save the strategy so Vectester can reload it and make it available in the strategy list

What happens when you save

When you save a strategy:

- the code is written to a `.py` file in the strategies folder
- the strategy registry is rebuilt
- the strategy becomes available in the Strategy dropdown if it loads successfully
- its declared parameters are automatically shown in the parameter forms

That means the editor is not just a text editor. It is the main gateway for extending Vectester.

5.2 Starting from a Blank Template

A new strategy should normally begin from a minimal template. The purpose of the template is to give you the exact structure Vectester expects, with the required imports, a class, a name, parameters, and a `build()` method.

A minimal strategy template looks like this:

```
import pandas as pd
import vectorbt as vbt
from core.params import ParamDef
from core.strategy_base import Strategy, StrategyResult
```

```python
class NewStrategy(Strategy):
    name = "new_strategy"
    param_defs = [
        ParamDef("ema_window", 20, int, "EMA window"),
    ]

    def build(self, data: pd.DataFrame, params: dict) -> StrategyResult:
        close = data["Close"]
        ema_window = int(params["ema_window"])
        ema = vbt.MA.run(close, window=ema_window, ewm=True).ma
        entries = (close > ema).fillna(False)
        exits = (close < ema).fillna(False)
        return StrategyResult(entries=entries, exits=exits, pf_kwargs={})
```

This template is deliberately simple. It does four essential things:

- imports what the strategy needs
- defines a class that inherits from `Strategy`
- declares one user-editable parameter
- builds entry and exit signals from the market data

Why starting simple matters

A strategy should begin with the smallest working version possible. This makes it easier to:

- confirm the strategy loads correctly
- verify the parameter form appears correctly in the GUI
- check that the backtest runs and produces trades
- isolate mistakes before adding more filters and complexity

A simple strategy that runs correctly is much more valuable than a complex strategy that fails to load or produces no trades.

5.3 Required Structure of a Strategy

Every Vectester strategy must follow the same basic structure.

Required parts

A valid strategy file needs:

- the required imports
- a class that inherits from `Strategy`

- a `name` attribute
- a `param_defs` list
- a `build()` method
- a return value of type `StrategyResult`

Standard layout

```python
import pandas as pd
import vectorbt as vbt
from core.params import ParamDef
from core.strategy_base import Strategy, StrategyResult


class MyStrategy(Strategy):
    name = "my_strategy"

    param_defs = [
        ParamDef("param1", 10, int, "Example integer parameter"),
        ParamDef("param2", 0.5, float, "Example float parameter"),
    ]

    def build(self, data: pd.DataFrame, params: dict) -> StrategyResult:
        close = data["Close"]

        param1 = int(params["param1"])
        param2 = float(params["param2"])

        # strategy logic here

        entries = ...
        exits = ...

        return StrategyResult(entries=entries, exits=exits, pf_kwargs={})
```

What Vectester expects

Vectester scans the strategy modules, imports them, and looks for classes that are subclasses of the base `Strategy` class. If the file does not define a valid strategy class in the expected form, it will not appear correctly in the app.

So the structure is not optional formatting. It is the contract between your code and the application.

5.4 Defining the Strategy Class

The strategy class is the central object Vectester works with.

A strategy class must inherit from `Strategy`:

```
class MyStrategy(Strategy):
```

This inheritance matters because the base class defines the expected interface. It tells Vectester that this class is meant to behave like a strategy and that it should provide the required fields and methods.

What the class represents

The class is the full definition of the strategy:

- its identity
- its editable parameters
- its trading rules
- any extra portfolio behavior it wants to control

One file, one clear strategy

The cleanest approach is usually one strategy per file, with one main strategy class inside that file. This keeps the registry behavior predictable and makes maintenance much easier.

Good practice

Use a class name that is readable and descriptive. For example:

```
class TripleEmaStrategy(Strategy):
```

The class name itself is less important to the user than the `name` attribute, but it should still be clear and professional.

5.5 Setting the Strategy Name

Every strategy needs a `name` attribute:

```
name = "triple_ema"
```

This is the strategy's public identifier inside Vectester.

What the name is used for

The `name` value is used to:

- identify the strategy in the Strategy dropdown
- distinguish one strategy from another in the registry
- label results and comparisons

Why the name must be unique

Two strategies must not share the same `name`. If two loaded strategies use the same name, Vectester treats that as a duplicate strategy-name problem, and one of them will fail to register correctly.

Good naming rules

A good strategy name should be:

- unique
- short
- descriptive
- stable over time

Examples of good names:

```
name = "buy_and_hold"
name = "ema_sltp"
name = "trend_rsi_dip"
name = "dual_momentum_atr_expansion"
```

Avoid vague names such as:

```
name = "strategy1"
name = "test"
name = "my_new_one"
```

Those names become confusing very quickly once you begin comparing many strategies.

File name vs strategy name

The file name and the strategy name are related, but they are not the same thing.

For example:

- file name: `triple_ema.py`
- strategy name: `"triple_ema"`

Keeping them aligned is strongly recommended, because it makes the strategy easier to locate, edit, and troubleshoot.

5.6 Declaring Parameters with `ParamDef`

Parameters are declared in the `param_defs` list.

Example:

```
param_defs = [
    ParamDef("ema_window", 21, int, "EMA window"),
    ParamDef("sl", 0.02, float, "Stop-loss fraction"),
    ParamDef("tp_ratio", 2.0, float, "Take-profit = sl * tp_ratio"),
]
```

This is one of the most important parts of strategy authoring because Vectester uses `param_defs` to automatically build the user interface for the strategy.

What a `ParamDef` contains

Each `ParamDef` defines four things:

- **name**: the parameter key
- **default**: the default value
- **dtype**: the expected data type
- **description**: a human-readable explanation

How each field works

`name`

The parameter name is how the value is stored and retrieved.

```
ParamDef("ema_window", 21, int, "EMA window")
```

Here the name is `ema_window`.

Inside `build()`, you access it with:

```
params["ema_window"]
```

The name should be clear and consistent. It should describe what the parameter actually controls.

`default`

The default value is the value Vectester places into the form when the strategy is selected.

In the example above, the default is `21`.

This value should be a reasonable starting point for the strategy. It should not be random. A bad default can make a strategy look broken even when the logic is fine.

`dtype`

The type tells Vectester what sort of input widget to create.

Common types are:

- `int`
- `float`
- `bool`

These matter because they affect both the run-once form and the optimization grid.

Typical behavior:

- `int` becomes an integer spin box
- `float` becomes a floating-point spin box
- `bool` becomes a True/False selector
- any other type is generally handled as text input

`description`

The description is the human explanation of what the parameter means.

Example:

```
ParamDef("sl", 0.02, float, "Stop-loss fraction")
```

Descriptions should be specific and useful. They are not just comments for programmers. They are part of making the strategy understandable.

Why `param_defs` is so important

`param_defs` is the bridge between strategy code and the GUI. It determines:

- which parameters users can change
- which defaults the user sees
- which parameters can be optimized
- what value types Vectester expects

If your `param_defs` list is wrong, the strategy may load but behave badly, or the user interface may become misleading.

Good parameter design

Good parameters are:

- meaningful
- necessary

- interpretable
- few in number

A smaller set of well-designed parameters is better than a large set of redundant or confusing ones.

5.7 Writing the `build()` Method

The `build()` method is where the strategy logic lives.

Example signature:

```
def build(self, data: pd.DataFrame, params: dict) -> StrategyResult:
```

This method receives:

- `data`: the market data currently loaded in Vectester
- `params`: the values collected from the strategy form

It must return a `StrategyResult`.

Step 1: read the input data

Most strategies begin by reading one or more columns from the data:

```
close = data["Close"]
high = data["High"]
low = data["Low"]
volume = data["Volume"]
```

The most common column is `Close`, but many strategies also use `Open`, `High`, `Low`, and `Volume`.

Step 2: read and cast parameters

Parameters come from the UI and should be converted to the expected types before use:

```
ema_window = int(params["ema_window"])
sl = float(params["sl"])
tp_ratio = float(params["tp_ratio"])
```

This is important even if the form already looks type-safe. Explicit conversion inside the strategy keeps the code predictable and avoids accidental type problems.

Step 3: compute indicators or conditions

This is where the actual trading logic begins.

Example:

```
ema = vbt.MA.run(close, window=ema_window, ewm=True).ma
```

You can compute any indicators, filters, or derived series the strategy needs.

Step 4: define entry logic

Entries are the bars where the strategy opens a position.

Example:

```
entries = close > ema
```

This creates a boolean series. Each bar is either `True` or `False`.

Step 5: define exit logic

Exits are the bars where the strategy closes a position.

Example:

```
exits = close < ema
```

This is also a boolean series.

Step 6: prepare optional portfolio arguments

Some strategies do not rely only on explicit exit signals. They may want stop-loss, take-profit, or trailing-stop behavior.

Those are passed through `pf_kwargs`.

Example:

```
pf_kwargs = {"sl_stop": sl_stop, "tp_stop": tp_stop}
```

Step 7: return the final result

Finally, the method must return:

```
return StrategyResult(entries=entries, exits=exits, pf_kwargs=pf_kwargs)
```

What good `build()` logic looks like

A good `build()` method should be:

- readable
- deterministic
- explicit about parameter casting
- clear about how entries and exits are formed

- careful about missing values
- disciplined about what it returns

The best strategy code is not the most clever code. It is the code that a researcher can understand, trust, and validate.

5.8 Returning a Valid `StrategyResult`

A strategy must return a `StrategyResult`.

The object contains three fields:

- `entries`
- `exits`
- `pf_kwargs`

entries

`entries` must be a boolean pandas Series. It tells Vectester where positions should be opened.

Example:

```
entries = (close > ema).fillna(False)
```

This is a strong pattern because it ensures missing values do not accidentally become invalid signals.

exits

`exits` is either:

- a boolean pandas Series, or
- `None`

Use a boolean series when the strategy has explicit exit rules.

Example:

```
exits = (close < ema).fillna(False)
```

Use `None` when the strategy relies on stop-based management instead of explicit exit signals.

Example:

```
exits = None
```

pf_kwargs

`pf_kwargs` is a dictionary of additional portfolio instructions.

Example:

```
pf_kwargs = {}
```

or:

```
pf_kwargs = {"sl_stop": sl_stop, "tp_stop": tp_stop}
```

This field is essential for strategies that use:

- stop-loss
- take-profit
- trailing stop
- other portfolio signal options

Example: explicit exits

```
return StrategyResult(
    entries=entries,
    exits=exits,
    pf_kwargs={}
)
```

Example: stop-based exits

```
return StrategyResult(
    entries=entries,
    exits=None,
    pf_kwargs={"sl_stop": sl_stop, "tp_stop": tp_stop}
)
```

Static and dynamic stop values

Stops can be given as simple constant values or as full per-bar series.

Static example:

```
sl_stop = 0.02
tp_stop = 0.04
```

Series example:

```
sl_stop = pd.Series(0.02, index=close.index)
tp_stop = pd.Series(0.04, index=close.index)
```

Dynamic series are useful when stop distance depends on volatility or some other changing market measure.

Why return quality matters

Even a strategy with correct-looking logic will fail in practice if it returns malformed objects. For example:

- wrong shapes
- wrong index alignment
- wrong types
- missing required fields

Returning a clean, valid `StrategyResult` is the final responsibility of the strategy author.

5.9 Saving and Reloading

Once the strategy code is complete, it must be saved so Vectester can reload it.

What happens on save

When you press **Save** in the editor:

- Vectester checks that the file name is not empty
- it checks that the file name is a valid Python-style identifier
- it checks that the code is not empty
- it writes the code into the strategies folder as a `.py` file

The file name must follow valid Python naming rules. It cannot contain spaces or invalid characters.

Examples of valid file names:

```
triple_ema
my_strategy
ema_pullback_v2
```

Examples of invalid file names:

```
my strategy
123test
ema-pullback
```

What happens after saving

After a successful save, the strategy system is reloaded. That reload process does the following:

- clears previously loaded strategy modules

- invalidates import caches
- scans the strategies package again
- imports the strategy modules again
- rebuilds the strategy registry

If the strategy loads correctly, it becomes available in the strategy list.

If it does not load correctly, the app records a load error.

Why reloading matters

Without reload behavior, you would have to restart the app after every change. Vectester's reload process makes strategy development much faster because you can:

- edit code
- save
- reload
- test immediately

That tight authoring loop is one of the strongest parts of the platform.

5.10 Common Authoring Mistakes

Most strategy problems come from a small set of repeated mistakes. Knowing them in advance saves a great deal of time.

1. Missing inheritance from `Strategy`

Wrong:

```
class MyStrategy:
```

Correct:

```
class MyStrategy(Strategy):
```

If the class does not inherit from `Strategy`, Vectester will not treat it as a valid strategy.

2. Missing or invalid `name`

Wrong:

```
class MyStrategy(Strategy):
    param_defs = []
```

Correct:

```python
class MyStrategy(Strategy):
    name = "my_strategy"
    param_defs = []
```

Every strategy needs a usable name.

3. Duplicate strategy names

Two different strategies must not use the same `name`. Duplicate names cause registry conflicts and prevent reliable loading.

4. Forgetting `param_defs`

Wrong:

```python
class MyStrategy(Strategy):
    name = "my_strategy"
```

Correct:

```python
class MyStrategy(Strategy):
    name = "my_strategy"
    param_defs = []
```

Even if the strategy has no user parameters, it should still define `param_defs`.

5. Using parameter names that do not match

Wrong:

```python
param_defs = [
    ParamDef("ema_window", 20, int, "EMA window"),
]

ema = vbt.MA.run(close, window=params["window"], ewm=True).ma
```

Correct:

```python
ema = vbt.MA.run(close, window=int(params["ema_window"]), ewm=True).ma
```

The names in `param_defs` and the keys used inside `params[...]` must match exactly.

6. Not converting parameter types

Wrong:

```python
ema_window = params["ema_window"]
```

Better:

```python
ema_window = int(params["ema_window"])
```

Explicit conversion keeps the strategy robust.

7. Returning the wrong object

Wrong:

```
return entries, exits
```

Correct:

```
return StrategyResult(entries=entries, exits=exits, pf_kwargs={})
```

The return type must be `StrategyResult`.

8. Returning malformed signals

Entries and exits should be boolean Series aligned to the data index.

Bad examples include:

- scalar booleans
- lists
- mismatched lengths
- Series with unrelated indices

9. Forgetting to handle missing values

Indicator calculations often create missing values at the beginning of the series. If those are not handled, the strategy may produce confusing behavior.

A safe pattern is:

```
entries = (close > ema).fillna(False)
exits = (close < ema).fillna(False)
```

10. Using stop-based exits incorrectly

If a strategy relies on stop-loss or take-profit logic, it is often cleaner to return `exits=None` and pass stop settings in `pf_kwargs`.

Example:

```
return StrategyResult(
    entries=entries,
    exits=None,
    pf_kwargs={"sl_stop": sl_stop, "tp_stop": tp_stop}
)
```

Trying to mix unclear manual exits with unclear stop logic can make the strategy hard to reason about.

11. Invalid file names

If the file name is invalid, Vectester will reject the save. Use only letters, numbers, and underscores, and do not begin with a number.

12. Empty code or incomplete files

A strategy file that is saved while incomplete may fail to import. That means it will not appear correctly in the strategy list.

13. Syntax or import errors

A simple syntax error is enough to make the strategy fail loading. The same is true for bad imports or names that do not exist.

14. Writing strategies that produce no trades

A strategy can be perfectly valid technically but still generate no trades because the conditions are too strict. When this happens:

- simplify the logic
- relax the thresholds
- test on more data
- verify the conditions are not mutually exclusive

15. Over-parameterizing too early

A strategy with too many knobs becomes difficult to understand and easy to overfit. Start with one or two meaningful parameters and expand only when necessary.

Practical Authoring Advice

When creating a new strategy, use this workflow:

1. Start from the smallest valid template.
2. Add one parameter at a time.
3. Keep the first version visually simple.
4. Make sure the strategy appears in the app.
5. Verify the parameter form looks correct.
6. Run a single backtest.
7. Confirm trades are generated.

8. Only then add stops, filters, or optimization ranges.

That sequence keeps strategy development disciplined and makes errors much easier to diagnose.

Minimal Example

```python
import pandas as pd
import vectorbt as vbt
from core.params import ParamDef
from core.strategy_base import Strategy, StrategyResult


class SimpleEmaStrategy(Strategy):
    name = "simple_ema"

    param_defs = [
        ParamDef("ema_window", 20, int, "EMA lookback window"),
    ]

    def build(self, data: pd.DataFrame, params: dict) -> StrategyResult:
        close = data["Close"]
        ema_window = int(params["ema_window"])

        ema = vbt.MA.run(close, window=ema_window, ewm=True).ma
        entries = (close > ema).fillna(False)
        exits = (close < ema).fillna(False)

        return StrategyResult(
            entries=entries,
            exits=exits,
            pf_kwargs={}
        )
```

Example with Stop-Loss and Take-Profit

```python
import pandas as pd
import vectorbt as vbt
from core.params import ParamDef
from core.strategy_base import Strategy, StrategyResult


class EmaSlTpStrategy(Strategy):
    name = "ema_sltp"

    param_defs = [
        ParamDef("ema_window", 21, int, "EMA window"),
        ParamDef("sl", 0.02, float, "Stop-loss fraction"),
        ParamDef("tp_ratio", 2.0, float, "Take-profit = sl * tp_ratio"),
```

```
    ]

    def build(self, data: pd.DataFrame, params: dict) -> StrategyResult:
        close = data["Close"]

        ema_window = int(params["ema_window"])
        sl = float(params["sl"])
        tp_ratio = float(params["tp_ratio"])
        tp = sl * tp_ratio

        ema = vbt.MA.run(close, window=ema_window, ewm=True).ma
        entries = (close > ema).fillna(False)

        sl_stop = pd.Series(sl, index=close.index)
        tp_stop = pd.Series(tp, index=close.index)

        return StrategyResult(
            entries=entries,
            exits=None,
            pf_kwargs={
                "sl_stop": sl_stop,
                "tp_stop": tp_stop
            }
        )
```

This example is especially important because it shows that a strategy does not always need explicit exit signals. In Vectester, exits can also be delegated to stop-based portfolio rules through `pf_kwargs`.

6. Understanding Strategy Anatomy

Every strategy in Vectester follows the same structural pattern. Once you understand this pattern, you can read built-in strategies more easily, modify them safely, and write your own strategies that integrate correctly with the rest of the application.

A strategy in Vectester is not just a block of trading logic. It is a contract with the application. The app expects each strategy to declare:

- a **display identity**
- a **set of parameters**
- a **build method** that converts market data and parameter values into signals
- an output object containing **entries**, **exits**, and optional **portfolio instructions**

At a high level, the flow is:

1. Vectester loads the selected strategy.
2. It reads that strategy's declared parameters.
3. It builds the parameter input form automatically.
4. It collects the user's chosen values.
5. It calls the strategy's `build(data, params)` method.
6. The strategy returns a `StrategyResult`.
7. Vectester uses that result to construct and run the backtest.

A minimal strategy looks like this:

```
class MyStrategy(Strategy):
    name = "my_strategy"

    param_defs = [
        ParamDef("ema_window", 20, int, "EMA window"),
    ]

    def build(self, data, params) -> StrategyResult:
        close = data["Close"]
        ema_window = int(params["ema_window"])

        ema = vbt.MA.run(close, window=ema_window, ewm=True).ma
        entries = (close > ema).fillna(False)
        exits = (close < ema).fillna(False)
```

```
        return StrategyResult(entries=entries, exits=exits, pf_kwargs={})
```

That simple structure contains all the core pieces Vectester needs.

6.1 The `name` field

The `name` field is the strategy's unique identifier inside Vectester.

```
name = "my_strategy"
```

This name is extremely important because it is what the application uses to register, list, and select the strategy.

What it does

The `name` field is used for:

- the strategy dropdown in the user interface
- internal strategy lookup
- result labeling
- distinguishing one strategy from another during loading

If the strategy does not define a valid `name`, the system falls back to the class name when building the registry. In practice, you should always define `name` explicitly.

What makes a good strategy name

A good strategy name should be:

- unique
- short
- descriptive
- stable over time

Examples:

```
name = "triple_ema"
name = "trend_rsi_dip"
name = "ema_volume_atr_long_short"
```

These are good because they clearly communicate the strategy's logic or style.

Why uniqueness matters

Vectester's strategy registry rejects duplicate names. If two strategy classes use the same `name`, one of them will fail to load and the registry will record a load error.

So this is wrong:

```
class StrategyA(Strategy):
    name = "momentum"

class StrategyB(Strategy):
    name = "momentum"
```

Each strategy must have its own unique name.

Best practice

Treat `name` as the strategy's permanent identity. If you modify the internals of a strategy but want it to remain the same strategy in the app, keep the same name. If you are creating a materially different variation, use a new name.

6.2 The `param_defs` list

The `param_defs` list tells Vectester which parameters your strategy exposes to the user.

```
param_defs = [
    ParamDef("ema_window", 20, int, "EMA window"),
    ParamDef("sl", 0.02, float, "Stop-loss fraction"),
    ParamDef("use_trailing_sl", True, bool, "Use trailing stop-loss"),
]
```

This list is one of the most important parts of the strategy because Vectester uses it to build the strategy interface automatically.

What `param_defs` controls

For every `ParamDef` in the list, Vectester creates:

- one input widget in the **Run Once** parameter form
- one text field in the **Optimization Grid** form

That means `param_defs` defines not only what the strategy needs, but also what the user can edit and optimize.

Structure of a parameter definition

Each item is a `ParamDef` with four parts:

`ParamDef(name, default, dtype, description)`

Where:

- `name` is the parameter key
- `default` is the default value
- `dtype` tells the UI and parser what type the value should be
- `description` is human-readable documentation

Example:

```
ParamDef("ema_window", 21, int, "EMA window")
```

This means:

- parameter name: `ema_window`
- default value: `21`
- expected type: integer
- meaning: the EMA lookback window

How the UI uses `dtype`

Vectester renders the Run Once form differently depending on the type:

- `int` → integer spin box
- `float` → decimal spin box
- `bool` → True/False dropdown
- anything else → text box

So this:

```
ParamDef("fast_ema", 20, int, "Fast EMA")
ParamDef("risk_pct", 0.05, float, "Risk fraction per trade")
ParamDef("use_trailing_sl", True, bool, "Use trailing stop-loss")
```

becomes three different kinds of controls in the app.

Why `param_defs` matters so much

If a parameter is not listed in `param_defs`, the user cannot configure it through the normal interface.

So the strategy author should think carefully about which values should be exposed and which should remain fixed inside the logic.

Good parameter design

Good parameters are:

- meaningful
- independent
- interpretable
- few in number

Bad parameter design usually means:

- too many overlapping parameters
- unclear names
- hidden coupling between settings
- defaults that produce no trades or unrealistic behavior

6.3 The `build(data, params)` method

The `build` method is the heart of the strategy.

```
def build(self, data, params) -> StrategyResult:
```

This method receives market data and resolved parameter values, applies the trading logic, and returns a `StrategyResult`.

What `build()` is responsible for

The method must:

1. read the required columns from the market data
2. read and convert parameter values
3. calculate indicators or filters
4. produce entry and exit signals
5. optionally define extra portfolio instructions
6. return a valid `StrategyResult`

This is the output format Vectester expects:

```
return StrategyResult(entries=entries, exits=exits, pf_kwargs=pf_kwargs)
```

What `build()` should not do

The strategy should not:

- handle GUI logic
- read form widgets directly
- manage optimization itself
- run the portfolio engine directly
- produce charts or reports

Its job is only to convert **data + parameters** into **signals + portfolio instructions**.

Typical internal structure

A well-written `build()` method usually follows this order:

```
def build(self, data, params):
    close = data["Close"]
    high = data["High"]
    low = data["Low"]

    ema_window = int(params["ema_window"])
    sl = float(params["sl"])

    ema = vbt.MA.run(close, window=ema_window, ewm=True).ma

    entries = (close > ema).fillna(False)
    exits = (close < ema).fillna(False)

    pf_kwargs = {
        "sl_stop": sl
    }

    return StrategyResult(entries=entries, exits=exits, pf_kwargs=pf_kwargs)
```

That pattern is easy to read, easy to debug, and easy to extend.

Required output

`build()` must return a `StrategyResult`. Returning anything else breaks the strategy contract.

6.4 What `data` contains

The `data` argument is the market dataset currently being tested.

```
def build(self, data, params):
```

It is passed in as a pandas DataFrame and contains the price series loaded by the app.

What columns are normally available

Vectester's strategies assume standard OHLCV structure:

- `Open`
- `High`
- `Low`
- `Close`
- `Volume` if available

Most strategies use at least `Close`. Many also use `High`, `Low`, and `Volume`.

Examples:

```
close = data["Close"]
high = data["High"]
low = data["Low"]
volume = data["Volume"]
```

What the index contains

The index represents time. It is typically a datetime index produced by the loaded dataset.

This means every derived signal should align to that same index.

Why column names matter

The strategy code is case-sensitive when reading columns. The data loader normalizes and prepares the input data so strategies can usually rely on standard capitalized names like `Close` and `Volume`.

How strategies use `data`

Common uses include:

- price-based trend filters
- volatility measures using `High`, `Low`, and `Close`
- volume filters using `Volume`
- intraday conditions based on time-index behavior
- indicator calculation across one or more columns

Important practical point

A strategy only sees the active dataset passed to it. It should assume `data` is the exact bar-by-bar series it must analyze and generate signals from.

6.5 What `params` contains

The `params` argument is a dictionary containing the strategy's parameter values for the current run.

Example:

```
{
    "ema_window": 21,
    "sl": 0.02,
    "tp_ratio": 2.0
}
```

Where `params` comes from

Vectester builds this dictionary from the strategy's `param_defs` and the current user selections.

For a Run Once operation:

- the app reads the current widgets in the parameter form
- it converts them to the declared types
- it merges them with the strategy defaults

For optimization:

- the optimizer constructs parameter combinations from the optimization grid
- each combination is merged with defaults
- each merged set is passed into `build()`

Important behavior

Before the strategy is built, Vectester applies:

```
params = {**strategy.defaults(), **(params or {})}
```

That means user-supplied values override defaults, but any missing values fall back to the defaults declared in `param_defs`.

Why explicit conversion is still a good idea

Even though Vectester parses values by type, strategy code still commonly converts them explicitly:

```
ema_window = int(params["ema_window"])
sl = float(params["sl"])
use_trailing_sl = bool(params["use_trailing_sl"])
```

This is good practice because it makes the strategy self-explanatory and reduces ambiguity.

What `params` should contain

It should contain one entry for each parameter declared in `param_defs`.

If your strategy logic refers to a parameter not declared there, that parameter will not be managed properly by the UI and may be missing.

6.6 The `entries` signal

The `entries` field tells Vectester when to open positions.

It is the most fundamental output of the strategy.

```
entries = close > ema
```

Then:

```
return StrategyResult(entries=entries, exits=exits, pf_kwargs={})
```

What `entries` should be

`entries` should be a boolean pandas Series aligned with the data index.

Each bar should be either:

- `True` → open a new position here
- `False` → do not open a new position here

Example:

```
entries = (close > ema).fillna(False)
```

Why `.fillna(False)` is often important

Indicators often produce missing values at the beginning of the series because they need lookback periods.

Without cleaning, those missing values can leak into the signal series.
Using `.fillna(False)` ensures the strategy does not try to enter trades on undefined bars.

What `entries` means in practice

It is not a description of bullishness in general. It is a specific instruction to the portfolio engine that this bar is a valid entry event.

That means `entries` should represent actual trade triggers, not vague market opinions.

Entry signal examples

Simple trend entry:

```
entries = close > ema
```

Crossover entry:

```
entries = (ema_fast > ema_slow) & (ema_fast.shift(1) <= ema_slow.shift(1))
```

Dip-buy entry:

```
entries = (close > trend_ma) & (rsi < 30)
```

Good entry signal design

A good entry signal is:

- aligned with the data index
- boolean
- explicit
- stable under missing values
- logically tied to the strategy's hypothesis

6.7 The `exits` signal

The `exits` field tells Vectester when to close positions using explicit signal logic.

```
exits = close < ema
```

What `exits` should be

Like `entries`, `exits` should normally be a boolean pandas Series aligned with the same index.

Each bar should be either:

- `True` → close the position here
- `False` → keep the position open

Exits can also be omitted

`exits` may be set to `None` when the strategy wants to rely on stop-loss or take-profit handling instead of explicit signal-based exits.

Example:

```
return StrategyResult(entries=entries, exits=None, pf_kwargs={"sl_stop": sl_s
top, "tp_stop": tp_stop})
```

This pattern is used in built-in strategies that let stop logic manage position closure.

Exit signal examples

Trend breakdown exit:

```
exits = close < ema
```

Reverse crossover exit:

```
exits = (ema_fast < ema_slow) & (ema_fast.shift(1) >= ema_slow.shift(1))
```

Momentum failure exit:

```
exits = roc_fast < 0
```

What makes a good exit signal

A good exit signal should:

- reflect the logic that invalidates the trade
- be aligned with entry logic
- not create unnecessary churn unless intended
- be simple enough to interpret clearly

6.8 The `pf_kwargs` dictionary

`pf_kwargs` is the strategy's channel for passing advanced portfolio instructions to the backtest engine.

```
pf_kwargs = {
    "sl_stop": sl_stop,
    "tp_stop": tp_stop,
    "sl_trail": True,
}
```

It is returned as part of the `StrategyResult`:

```
return StrategyResult(entries=entries, exits=exits, pf_kwargs=pf_kwargs)
```

What it is used for

`pf_kwargs` can carry strategy-specific execution settings that go beyond simple entry and exit signals.

In the source, this is where strategies pass things like:

- stop-loss values
- take-profit values
- trailing-stop flags
- short-entry signals
- short-exit signals
- position size information
- strategy-level direction mode

How it interacts with global backtest settings

Vectester first creates a base set of portfolio settings from the Backtest page, including:

- direction
- initial cash
- fees
- slippage
- frequency

Then it merges in the strategy's `pf_kwargs`.

The merge logic is effectively:

```
out = dict(base or {})
out.update(extra or {})
```

So if the same key exists in both places, the strategy's `pf_kwargs` wins.

That means strategy-level portfolio instructions can override global settings.

Common `pf_kwargs` fields seen in the source

Stop-loss

```
"sl_stop": 0.02
```

or as a series:

```
"sl_stop": sl_series
```

Take-profit

```
"tp_stop": 0.04
```

or as a series:

```
"tp_stop": tp_series
```

Trailing stop

```
"sl_trail": True
```

Short-side signals

```
"short_entries": short_entries,
"short_exits": short_exits,
```

Position sizing

```
"size": size,
"size_type": "amount",
```

Direction override

```
"direction": "both"
```

Scalar values vs series values

Some `pf_kwargs` values can be simple scalars, while others can be time-varying Series.

For example:

```
"sl_stop": 0.02
```

means a fixed 2% stop.

But:

```
"sl_stop": atr_based_stop_series
```

means the stop distance can change bar by bar.

This is one of the most powerful parts of Vectester's strategy architecture.

6.9 When to use exits vs stop-based exits

This is one of the most important design decisions in strategy writing.

A strategy can close trades in two broad ways:

1. through an explicit `exits` signal
2. through stop-related values in `pf_kwargs`

Sometimes a strategy uses one. Sometimes it uses both.

Use `exits` when the trade should end because the logic changed

Examples:

- trend condition has broken
- crossover has reversed
- momentum has failed
- regime filter is no longer valid

Example:

```
entries = (e1 > e2) & (e2 > e3)
exits = (e1 < e2) & (e2 < e3)
```

This is ideal when the strategy is fundamentally rule-driven.

Use stop-based exits when the trade should end because risk or reward targets were hit

Examples:

- fixed percentage stop-loss
- fixed risk/reward take-profit
- ATR-based trailing stop
- volatility-adaptive protective exit

Example:

```
return StrategyResult(
    entries=entries,
    exits=None,
    pf_kwargs={"sl_stop": 0.02, "tp_stop": 0.04}
)
```

This is ideal when the strategy is fundamentally position-managed rather than rule-exited.

Use both when both logic and risk control matter

Some strategies use:

- explicit exits for logical invalidation
- stop-loss and take-profit for risk control and profit capture

Example pattern:

```
return StrategyResult(
    entries=entries,
    exits=logic_exits,
    pf_kwargs={"sl_stop": sl, "tp_stop": tp}
)
```

This can be very effective, but it also makes the strategy more complex to analyze.

How to choose between them

Use explicit `exits` when:

- your exit rule is part of the strategy's predictive logic
- you want trades to close only when the model says the edge is gone
- you want maximum interpretability of the strategy structure

Use stop-based exits when:

- risk management is central to the strategy
- you want to cap downside mechanically
- your strategy benefits from fixed or dynamic reward/risk targets

Use both when:

- the strategy has a clear signal-based thesis
- but also needs hard risk limits

Practical caution

If a strategy combines weak entry logic with overly optimized stop logic, optimization may produce attractive results that do not generalize well. This is one of the common paths to overfitting.

6.10 How defaults are applied

Defaults in Vectester come directly from the strategy's `param_defs`.

Each `ParamDef` contains a default value:

```
ParamDef("ema_window", 21, int, "EMA window")
```

Here, `21` is the default.

Where defaults are used

Defaults are used in three important places:

1. when the parameter form is first built
2. when the optimization grid is first populated
3. when Vectester merges missing values before running the strategy

Form initialization

When you select a strategy, Vectester rebuilds the parameter forms and sets each widget to the default declared in `param_defs`.

So defaults are not only documentation. They are the actual initial values shown to the user.

Grid initialization

The optimization grid fields are also initialized from the same defaults. Each grid field starts with the default value as text.

That means the same source of truth drives both:

- single-run parameter values
- optimization-grid starting values

Runtime merge behavior

Before `build()` is called, Vectester merges values like this:

```
params = {**strategy.defaults(), **(params or {})}
```

This means:

- start with all defaults
- overlay any user-supplied values
- use the result as the final parameter set

So defaults fill any gaps automatically.

Why this matters

This has several consequences:

- every declared parameter should have a sensible default
- defaults should produce a valid, understandable baseline run whenever possible
- poor defaults make the strategy look broken even if the logic is good
- optimization also depends on these defaults when some parameters are not varied

Best practice for defaults

Good defaults should:

- be realistic
- reflect the intended use of the strategy
- produce trades on appropriate datasets
- not rely on extreme values
- serve as a sensible baseline for optimization

Bad defaults often lead to:

- zero trades
- unstable metrics
- misleading first impressions
- unnecessary user confusion

Summary

A Vectester strategy is built around a small number of core pieces:

- `name` gives the strategy its identity
- `param_defs` declares what the user can configure
- `build(data, params)` contains the trading logic
- `data` provides the market series
- `params` provides the resolved parameter values
- `entries` tells the engine when to open trades
- `exits` tells the engine when to close trades by rule
- `pf_kwargs` adds stop logic, short logic, sizing, and other portfolio instructions
- defaults ensure the strategy always has a complete parameter set

Once these parts are understood, the rest of strategy writing becomes much easier, because every built-in and custom strategy in Vectester ultimately follows this same anatomy.

7. Parameter Definitions in Detail

Vectester uses a small but very important structure called `ParamDef` to describe every strategy parameter. This is the bridge between strategy code and the user interface. When a strategy is loaded, the application reads its list of `ParamDef` entries and uses them to build the parameter controls shown on the **Strategy** page.

In practical terms, this means that if a strategy author defines parameters correctly, Vectester can automatically:

- show the parameter in the **Run Once** form,
- show the same parameter in the **Optimization Grid** form,
- fill in sensible default values,
- choose the right input control for the parameter type,
- parse user input into the correct kind of value.

A `ParamDef` contains four fields:

- **name**
- **default**
- **dtype**
- **description**

A typical example looks like this:

```
ParamDef("ema_window", 20, int, "EMA window")
```

This tells Vectester that the strategy has a parameter called `ema_window`, its default value is `20`, it should be treated as an integer, and the user-facing description is “EMA window”.

7.1 What `ParamDef` Does

`ParamDef` is the formal declaration of a strategy parameter.

Without it, the application does not know:

- what the parameter is called,
- what value it should start with,
- what type of value it expects,
- how to show it in the interface.

Each strategy provides a list of `ParamDef` entries in its `param_defs` definition. Vectester reads that list and builds the parameter area automatically. This is why strategy parameters in the app are not hard-coded page by page. The strategy defines them, and the interface adapts.

`ParamDef` serves five main purposes:

1. It defines the parameter contract

It tells the app what inputs the strategy expects.

2. It controls the initial values

When a strategy is selected, the app fills the form with the `default` value from each parameter definition.

3. It controls the input widget type

The `dtype` determines whether the app shows a numeric spinner, decimal spinner, boolean selector, or plain text field.

4. It feeds both execution modes

The same parameter definition is used for:

- **single backtest runs**, and
- **parameter optimization grids**.

5. It improves readability and maintainability

A well-written `ParamDef` list makes a strategy easier to understand, edit, optimize, and document.

So although `ParamDef` is small, it is one of the most important pieces of the whole strategy system.

7.2 Parameter Name

The `name` field is the identifier of the parameter.

Example:

```
ParamDef("ema_window", 20, int, "EMA window")
```

Here, the parameter name is:

```
"ema_window"
```

This name is important because it is used in several places at once:

- as the row label in the parameter form,

- as the row label in the optimization grid,
- as the key in the `params` dictionary passed into `build(data, params)`,
- as the reference the strategy code uses when reading the value.

For example, inside the strategy:

```
ema_window = int(params["ema_window"])
```

If the parameter is named incorrectly, inconsistently, or ambiguously, the strategy becomes harder to use and easier to break.

Good parameter names should be:

- **clear**
- **specific**
- **consistent**
- **stable**

Good examples

- `ema_window`
- `fast_window`
- `slow_window`
- `rsi_period`
- `atr_mult`
- `sl_pct`
- `tp_ratio`
- `volume_threshold`

Weak examples

- `x`
- `param1`
- `value`
- `thing`
- `n`

- `data2`

A user should be able to look at the name and have a good idea of what it controls.

Naming recommendations

Use names that communicate one of these clearly:

- the indicator being configured,
- the role of the parameter,
- the unit or interpretation.

For example:

- `ema_window` is better than `window`, because it tells the user this window belongs to the EMA.
- `sl_pct` is better than `sl`, because it suggests a percentage stop-loss.
- `atr_multiplier` is better than `mult`, because it is explicit.

Avoid ambiguous abbreviations

Short names are fine if they are standard and obvious, but avoid abbreviations that a normal user cannot decode quickly.

For example:

- `rsi_n` is acceptable if the manual explains the naming style,
- `rn` is not.

Keep names consistent across strategies

If one strategy uses `ema_window` and another uses `ema_n` for the same idea, users have to relearn the interface every time. Consistency matters.

7.3 Default Value

The `default` field is the value Vectester uses when the strategy is first loaded into the interface.

Example:

```
ParamDef("ema_window", 20, int, "EMA window")
```

The default is:

```
20
```

When the strategy is selected:

- the **Run Once** control starts at `20`,
- the **Optimization Grid** field also starts with `20`.

This means the default value acts as both:

- the starting point for ordinary testing,
- the starting point for optimization editing.

Why default values matter

A default value is not just a placeholder. It strongly influences the user experience.

Good defaults:

- let the strategy run immediately,
- produce sensible behavior,
- reduce user confusion,
- serve as examples of reasonable values.

Poor defaults:

- produce no trades,
- produce unrealistic behavior,
- encourage bad optimization ranges,
- make the strategy feel broken even when the logic is fine.

A default should ideally be:

- valid,
- realistic,
- representative,
- safe enough for initial testing.

Example

```
ParamDef("rsi_entry", 30.0, float, "RSI oversold entry threshold")
```

This is much better than a random default like `3.0` or `95.0`, because it matches a commonly understood RSI entry concept.

Defaults also shape optimization behavior

Since the optimization field is initialized with the same default, users often build their search ranges around it.

For example, if the default is:

```
20
```

a user may naturally expand it to:

```
10, 20, 30
```

So the default helps anchor the user's optimization choices.

Best practice for defaults

Choose values that:

- make sense in the strategy's intended timeframe,
- reflect the logic of the strategy,
- help users understand how the strategy is meant to be used.

7.4 Data Type

The `dtype` field tells Vectester what kind of value the parameter is supposed to hold.

Examples:

```
ParamDef("ema_window", 20, int, "EMA window")
ParamDef("sl_pct", 0.02, float, "Stop-loss percentage")
ParamDef("use_filter", True, bool, "Enable volatility filter")
```

The supported core types are:

- `int`
- `float`
- `bool`

Anything else is treated more generically.

The data type affects:

- how the parameter appears in the interface,
- how the app reads the user's input,
- how the optimization grid parses comma-separated values.

Why type matters

A parameter is not just a value. It has a meaning.

For example:

- a lookback window is usually an **integer**,
- a threshold is often a **float**,
- an on/off switch is a **boolean**.

If the wrong type is used, the interface may become awkward or misleading.

Type determines interaction

An integer parameter should not allow decimal values.

A float parameter should not force whole numbers only.

A boolean parameter should not require users to type freeform text.

That is why `dtype` is so important. It tells the app how to create the right control and how to interpret the user's input correctly.

Missing or unsupported type

If a parameter does not use `int`, `float`, or `bool`, the app falls back to a plain text field. This is more flexible, but less guided and more error-prone.

That fallback is useful for advanced cases, but it places more responsibility on the strategy author and user.

7.5 Description

The `description` field is the human explanation of the parameter.

Example:

```
ParamDef("ema_window", 20, int, "EMA window")
```

Here the description is:

```
"EMA window"
```

Its purpose is to explain what the parameter means.

Even though the current form layout mainly uses the parameter name as the visible row label, the description still matters for documentation quality, strategy clarity, future maintainability, and any editor or reporting features that surface parameter information.

Why descriptions matter

A parameter name alone is often not enough.

Compare these:

```
ParamDef("atr_mult", 2.0, float, "")
```

versus:

```
ParamDef("atr_mult", 2.0, float, "ATR multiplier used to size the stop-loss d
istance")
```

The second version is far more helpful.

Descriptions are especially valuable when:

- parameters have abbreviated names,
- several parameters look similar,
- the parameter affects risk management,
- the strategy will be used by someone other than its author.

A good description should explain:

- what the parameter controls,
- what the value represents,
- how it is used in the strategy.

It does not need to be long, but it should be precise.

7.6 How the App Renders `int` Parameters

When `dtype` is `int`, Vectester renders the parameter as an integer spin box.

Example:

```
ParamDef("ema_window", 20, int, "EMA window")
```

This appears as a numeric control that allows whole numbers only.

Behavior of integer parameters in the app

For run-once testing:

- the parameter is shown as an integer spinner,
- the spinner is initialized with the default value,

- decimals are not allowed,
- the value is collected as an integer.

Internally, the app sets a very wide allowed range, so users are not artificially limited for ordinary strategy work.

Why this is useful

An integer control is ideal for values like:

- moving average windows,
- lookback periods,
- bar counts,
- lag lengths,
- discrete regime counts.

These should almost always be whole numbers.

Examples of good integer parameters

- `fast`
- `slow`
- `ema_window`
- `rsi_period`
- `atr_period`
- `lookback_bars`
- `min_bars`

Optimization grid behavior for integers

In the optimization grid, the field is shown as plain text, not as multiple spin boxes. The user enters comma-separated values such as:

```
10, 20, 50
```

The app then parses each item as an integer.

If the text cannot be converted cleanly to integers, optimization will fail for that parameter input.

Practical note

Use `int` only when the parameter truly must be discrete. Do not use `int` for values that conceptually represent percentages, multipliers, thresholds, or coefficients unless they really are meant to be whole numbers.

7.7 How the App Renders `float` Parameters

When `dtype` is `float`, Vectester renders the parameter as a decimal spin box.

Example:

```
ParamDef("sl_pct", 0.02, float, "Stop-loss percentage")
```

This allows decimal values and is the correct choice for most continuous numeric settings.

Behavior of float parameters in the app

For run-once testing:

- the parameter is shown as a decimal spinner,
- the spinner starts at the default value,
- decimal input is allowed,
- the control keeps up to 8 decimal places,
- the value is collected as a floating-point number.

The app also sets a very wide numeric range here, so users can test a broad variety of values.

Why this is useful

A float control is ideal for values like:

- percentages,
- thresholds,
- ratios,
- multipliers,
- volatility filters,
- stop distances,
- take-profit targets.

Examples of good float parameters

- `sl_pct`
- `tp_ratio`
- `atr_mult`
- `volume_multiplier`
- `momentum_threshold`
- `regime_threshold`

Optimization grid behavior for floats

In the optimization grid, the user enters comma-separated decimal values, for example:

```
0.01, 0.02, 0.03
```

The app parses each entry as a float.

This allows users to sweep through a range of continuous values during optimization.

Practical note

Use `float` whenever fractional values are meaningful. This is usually the right choice for stop-loss percentages, take-profit ratios, signal thresholds, and multipliers.

7.8 How the App Renders `bool` Parameters

When `dtype` is `bool`, Vectester renders the parameter as a dropdown selector with two choices:

- `True`
- `False`

Example:

```
ParamDef("use_trend_filter", True, bool, "Enable trend filter")
```

Behavior of boolean parameters in the app

For run-once testing:

- the parameter appears as a two-choice selector,
- the current value is set to `True` or `False` based on the default,
- the collected value becomes a real boolean.

This is useful for turning parts of a strategy on or off.

Common uses for boolean parameters

- enable or disable a filter,
- turn stop logic on or off,
- switch confirmation logic on or off,
- activate optional rules,
- choose between two behaviors in a simple way.

Examples

- `use_trend_filter`
- `use_volume_filter`
- `allow_reentry`
- `use_trailing_stop`

Optimization grid behavior for booleans

In the optimization grid, boolean values are entered as text in a comma-separated field.

Examples:

`True, False`

or

`1, 0`

or

`yes, no`

The parser treats entries such as `1`, `true`, and `yes` as `True`. Anything else becomes `False`.

This is flexible, but it also means strategy authors and users should be careful to enter boolean grids cleanly and consistently.

Practical note

Boolean parameters are powerful, but too many of them can explode the optimization grid. Every `True/False` switch doubles the number of combinations. Use them only when they represent a meaningful design choice.

7.9 How Other Parameter Types Are Handled

If a parameter's `dtype` is not exactly `int`, `float`, or `bool`, Vectester does not create a specialized numeric or boolean control. Instead, it falls back to a plain text input field.

This is the generic handling path.

Example:

```
ParamDef("mode", "aggressive", str, "Trading mode")
```

Since `str` is not one of the app's three special rendering types, the parameter is displayed as a standard text box.

Run-once behavior for other types

For single runs:

- the parameter appears as a text field,
- the default is inserted as text,
- the entered value is collected as text.

This gives the strategy author flexibility, but it also means the strategy code must handle conversion and validation carefully.

For example, if a user enters:

```
aggressive
```

then the strategy receives that text as-is.

Optimization behavior for other types

In the optimization grid:

- values are entered as comma-separated text,
- the app splits them into strings,
- no automatic numeric conversion is applied.

Example:

```
aggressive, balanced, conservative
```

These become a list of strings.

Important implications

This fallback behavior is flexible, but it has trade-offs:

- there is less input guidance,
- the app does less validation,

- strategy code must be more defensive,
- user mistakes become easier.

When this is useful

Fallback text handling can still be useful for:

- mode names,
- symbolic options,
- labels,
- categorical strategy modes,
- custom textual flags.

When to avoid it

Do not rely on generic text fields when a parameter is really numeric or boolean. If the parameter should be an integer, float, or boolean, declare it that way. The user experience is much better, and the app can help more.

7.10 Writing Clear Parameter Descriptions

Clear parameter descriptions are one of the best ways to make a strategy usable.

A good strategy is not just one that works. It is one that another user can read, understand, configure, optimize, and trust. Parameter descriptions play a major role in that.

A strong description should answer at least one of these questions:

- What does this parameter control?
- What unit is it in?
- What happens when it increases or decreases?
- Where is it used in the logic?
- Is it a threshold, window, ratio, or switch?

Weak descriptions

- `Window`
- `Threshold`
- `Value`
- `Multiplier`

- `Parameter`

These are technically labels, but they do not really explain anything.

Better descriptions

- `EMA lookback window used for the trend filter`
- `RSI level below which long entries are allowed`
- `ATR multiplier used to calculate the stop-loss distance`
- `Take-profit target as a multiple of the stop distance`
- `Enable volume filter for entry confirmation`

These give the user a much better mental model.

Good description principles

Be specific

Say what the parameter belongs to.

Better:

- `Fast EMA window`

Worse:

- `Window`

Mention the role

Say whether it affects entries, exits, filtering, or risk.

Better:

- `Minimum volume ratio required for entry`

Worse:

- `Volume threshold`

Include the unit or interpretation

If the value is a fraction, percentage, multiplier, or bar count, make that obvious.

Better:

- `Stop-loss as a fraction of entry price`

Worse:

- `Stop-loss`

Keep it concise

Descriptions should be informative, not bloated.

Better:

- `ATR period used for volatility filter`

Worse:

- `This parameter tells the strategy how many previous bars should be taken into account for the ATR calculation that is later used by the volatility filter logic`

Recommended description patterns

For windows:

- `EMA lookback window`
- `RSI calculation period`
- `Rolling volatility window`

For thresholds:

- `Minimum momentum threshold for entry`
- `RSI oversold threshold`
- `Trend strength threshold`

For risk controls:

- `Stop-loss percentage`
- `Take-profit ratio relative to stop distance`
- `ATR multiplier for trailing stop`

For booleans:

- `Enable trend confirmation filter`
- `Allow short trades`
- `Use trailing stop`

Why this matters so much in Vectester

Because Vectester builds the interface directly from the parameter definitions, a badly written parameter definition becomes a badly explained user interface.

In other words:

- clear `ParamDef` definitions create a clear strategy form,
- weak `ParamDef` definitions create a confusing strategy form.

That is why writing parameters well is not optional. It is part of writing the strategy properly.

Summary

`ParamDef` is the mechanism that makes Vectester's strategy system dynamic and user-friendly. It defines:

- the parameter's identifier,
- its starting value,
- its expected type,
- and its human meaning.

A well-designed parameter definition gives the user:

- the right control,
- the right default,
- the right optimization behavior,
- and a much clearer understanding of the strategy.

A poorly designed parameter definition does the opposite.

For strategy authors, this means parameter design is not a minor detail. It is part of the strategy's interface, documentation, and usability.

8. Strategy Logic Design

A trading strategy in Vectester is not just a formula. It is a structured decision process that answers four questions:

1. **When should a trade begin?**
2. **When should it end?**
3. **What market conditions should be allowed or avoided?**
4. **How should multiple signals be combined without creating noise or contradictions?**

In practical terms, strategy logic is built from conditions. Those conditions are evaluated bar by bar on the loaded market data. The final result is a set of entry signals, exit signals, and optional portfolio instructions such as stop-loss, take-profit, trailing stop, sizing, or long/short behavior.

The built-in strategies in Vectester follow a very consistent design pattern. Most of them are built by combining a **primary signal** with one or more **filters**. The primary signal is the main reason to trade. The filters decide whether the trade is allowed. This keeps strategies readable, controllable, and easier to optimize.

8.1 Building entry rules

An **entry rule** defines the exact conditions under which a new position is opened.

A good entry rule is precise. It should describe a market situation that is meaningful, repeatable, and testable. In Vectester, entry rules are typically based on one of these patterns:

- a **state condition**, such as price being above a moving average
- a **crossing event**, such as a fast EMA moving above a slow EMA
- a **threshold condition**, such as RSI being below 20 or ROC being above 1
- a **breakout event**, such as price moving above a recent channel high
- a **multi-condition setup**, such as trend is up, momentum is positive, and volatility is expanding

A weak entry rule is too vague. For example, “buy when the market looks strong” cannot be tested. A strong entry rule translates that idea into measurable conditions, such as “buy when the fast EMA is above the slow EMA and PPO crosses above zero.”

There are two broad ways to think about entries.

State-based entries

A state-based entry says: enter while a condition is true.
Example: buy whenever price is above the EMA.

This is simple, but it can repeatedly signal the same condition over many bars. In practice, the portfolio engine determines whether additional entries matter while already in a trade.

Event-based entries

An event-based entry says: enter when something changes.
Example: buy only when PPO crosses from negative to positive.

This is usually cleaner because it focuses on transition points rather than persistent states.

Most of the stronger built-in strategies use event-based entries or event-plus-filter entries. For example:

- **Triple EMA** uses alignment of moving averages as a trend structure
- **EMA PPO STDDEV** waits for PPO to cross above zero while trend and volatility conditions support the move
- **Intraday Volatility Breakout** waits for a prior compression condition and then a breakout above the previous upper channel
- **OBV RSI Volume ATR Long** waits for OBV to cross above its average, then requires RSI and volume filters to agree

When designing entry rules, keep these principles in mind.

Use one primary cause for entry

A trade should have a clear reason to exist. That reason might be:

- trend continuation
- pullback within trend
- breakout from compression
- momentum acceleration
- reversal from an extreme

If a strategy has too many equal-priority entry conditions, it becomes harder to understand, harder to optimize, and easier to overfit.

Separate the trigger from the filter

A clean design usually looks like this:

- **Trigger:** the event that starts the trade
- **Filter:** the conditions that must already be favorable

For example:

- Trigger: PPO crosses above zero
- Filters: EMA trend is up, Ultimate Oscillator is above threshold

This is better than mixing all conditions together mentally. It gives the strategy a clearer structure.

Prefer meaningful transitions over noisy states

A raw condition like `ROC > 0` may stay true for many bars. A transition like "ROC crosses above threshold" often better represents a new opportunity.

Avoid entries that are too rare

If an entry rule is too restrictive, the strategy may produce very few trades. Then optimization results become unreliable, and performance metrics become unstable.

Avoid entries that are too frequent

If a strategy enters constantly, the logic is often too loose. That tends to create overtrading, poor selectivity, and higher sensitivity to fees and slippage.

A good entry rule should balance selectivity and opportunity.

8.2 Building exit rules

An **exit rule** defines when a position should be closed.

Exit logic is as important as entry logic. Many weak strategies spend all their design effort on entries and treat exits as an afterthought. In reality, exit design largely controls:

- average trade duration
- realized profit size
- drawdown behavior
- sensitivity to reversals
- robustness across regimes

In Vectester, exits usually take one of two forms:

- **explicit exit signals**

- **risk-management exits through portfolio settings**, such as stop-loss and take-profit

Some strategies use both.

Explicit exit signals

These are logical conditions that tell the strategy to close the trade.

Examples from the built-in strategies include:

- exit when fast EMA falls below slow EMA
- exit when ROC turns negative
- exit when RSI rises above an exit threshold
- exit when a regime filter turns unfavorable
- exit when price climbs back above a regression line after a pullback entry

Explicit exits are useful when the strategy has a clear reason why the trade thesis is no longer valid.

Stop-based exits

These do not depend on a new indicator signal. Instead, they exit because price moves far enough against or in favor of the position.

Examples:

- fixed fractional stop-loss
- ATR-based stop-loss
- trailing stop-loss
- fixed or ATR-based take-profit

Stop-based exits are especially useful when:

- the trade needs hard risk control
- the strategy uses sparse signal logic
- the strategy has no natural indicator-based exit
- the strategy is designed around breakout or pullback patterns where price behavior matters more than indicator reversal

Several built-in strategies return `exits = None` and rely mainly on stop-based trade management.

Good exit design principles

A good exit should answer one of these questions clearly:

- Has the original reason for the trade disappeared?
- Has the market moved enough against the position to invalidate the setup?
- Has the market moved enough in favor of the position to capture the intended payoff?
- Has the regime changed enough that staying in the trade no longer makes sense?

Common exit styles

Trend failure exit
Used in trend systems. Exit when the trend structure breaks.

Momentum failure exit
Used in momentum systems. Exit when momentum weakens or turns negative.

Target-based exit
Used when the strategy seeks a defined move.

Protective stop exit
Used to limit loss size.

Trailing exit
Used to capture longer trends while locking in gains gradually.

Time-style exit
Not heavily used in the current built-in set, but conceptually valid. Exit after a defined holding period or session boundary.

Entry and exit should fit together

The entry and exit logic should reflect the same trading idea.

Examples:

- A breakout strategy usually pairs well with trailing stops and volatility-aware exits.
- A mean-reversion strategy usually pairs well with target exits or “return to average” exits.
- A trend-following strategy usually pairs well with trend failure exits and wider stops.

- A pullback strategy often exits when price returns to the trend line or when the trend itself weakens.

When entry and exit logic are mismatched, the strategy often feels inconsistent and performs erratically.

8.3 Trend-following logic

Trend-following logic tries to participate in sustained directional moves. It assumes that once a market begins trending, it often continues longer than expected.

In Vectester, trend-following is one of the most common foundations for strategy design. Many built-in strategies begin by asking a simple question: **is the market directionally aligned upward, downward, or neutral?**

Typical trend-following tools include:

- moving averages
- moving average crossovers
- moving average slope
- price relative to moving average
- regression line slope
- directional indicators such as Vortex
- regime models that identify favorable directional conditions

Common trend-following patterns

Price above trend line
A simple and common condition.
Example: price above EMA or SMA.

Fast average above slow average
This defines directional alignment.
Example: fast EMA > slow EMA.

Trend line rising
This confirms not just location, but direction.
Example: EMA rising or regression line rising.

Multi-layer trend structure
This uses several averages to confirm a stronger trend state.
Example: EMA1 > EMA2 > EMA3.

Why trend-following works

Trend-following works when markets display persistence. It does not try to buy the exact low or sell the exact high. It tries to capture the middle of a directional move.

Strengths

- simple to understand
- often robust across instruments
- can capture large moves
- pairs well with trailing stops
- works well as a filter for other logic

Weaknesses

- performs poorly in sideways markets
- can enter late
- can suffer repeated whipsaws
- often gives back gains before exit

Built-in examples

Triple EMA
Uses stacked moving averages to define a bullish or bearish structure.

EMA RSI Trend
Uses EMA structure to define trend, then RSI to confirm strength.

GMM Regime EMA Long
Combines EMA trend with a regime filter.

Linear Regression Pullback Long
Uses a rising regression line as a trend definition, then enters on pullbacks.

Design advice

When building trend-following logic:

- decide whether trend is the main trigger or only a filter
- use as few trend definitions as necessary
- avoid stacking several nearly identical trend indicators
- make sure the trend window is appropriate for the data timeframe

- expect drawdowns during sideways phases

Trend-following logic is often strongest when paired with either:

- a momentum trigger, or
- a pullback entry

8.4 Mean-reversion logic

Mean-reversion logic assumes that price often returns toward a central tendency after moving too far away from it.

Instead of buying strength, mean-reversion usually buys weakness in an otherwise acceptable environment, or sells strength when price becomes stretched.

Typical mean-reversion tools include:

- RSI extremes
- distance from moving average
- deviation from regression line
- volatility expansion after an overshoot
- oscillator thresholds

Common mean-reversion patterns

Oversold within trend
Buy when a strong market temporarily pulls back.
This is one of the safest forms of mean reversion because it avoids fighting the broader trend.

Price below reference line by a buffer
Buy when price is sufficiently below EMA, SMA, or regression line.

Oscillator below entry threshold
Buy when RSI or similar indicator enters an extreme zone.

Exit on return to normality
Close when price or oscillator returns to a more neutral zone.

Why mean reversion works

Markets do not move in straight lines. Even strong trends often contain temporary pullbacks. Mean-reversion logic tries to exploit short-lived overextensions.

Strengths

- often good entry prices
- can produce favorable reward-to-risk
- useful for pullback entries within trends
- often generates clearer target-based exits

Weaknesses

- dangerous in collapsing markets
- can keep buying weakness that continues
- very sensitive to regime
- easily overfit with too many thresholds

Built-in examples

Trend RSI Dip
A classic example. It uses a trend filter, then buys when RSI falls below an entry level, and optionally exits when RSI rebounds.

Linear Regression Pullback Long
Buys when price is below a rising regression line by a buffer amount.

EMA NATR Quiet Long
Although not pure mean reversion, it has a similar "wait for favorable conditions" character by requiring quiet volatility before entry.

Design advice

Mean-reversion works best when you clearly answer:

- mean reversion toward what?
- how far is "too far"?
- what confirms the market is still tradable?
- what prevents the strategy from buying every falling market?

A strong design often uses:

- a trend filter first
- a pullback or oversold condition second
- a target or recovery exit third

Without the trend filter, mean-reversion strategies often become fragile.

8.5 Momentum logic

Momentum logic looks for markets that are already moving with force and attempts to join that move.

Momentum is not exactly the same as trend. Trend describes directional structure. Momentum describes speed, strength, or acceleration of movement.

Typical momentum tools include:

- rate of change (ROC)
- PPO
- RSI above a bullish threshold
- breakout above recent highs
- rising volatility accompanying price movement

Common momentum patterns

Positive momentum threshold
Enter when ROC or PPO is above zero or above a chosen threshold.

Fast momentum above slow momentum
This suggests acceleration.
Example: fast ROC > slow ROC.

Momentum crossing upward
A transition signal.
Example: PPO crossing above zero.

Momentum with trend alignment
A strong combination.
Example: fast EMA > slow EMA and PPO cross-up.

Why momentum works

Momentum strategies assume that strong movement tends to continue at least for some period. They try to capture continuation rather than reversal.

Strengths

- can identify strong opportunities early
- often combines well with trend filters
- adapts well to breakout logic

- useful in directional markets

Weaknesses

- often vulnerable to sudden reversals
- can buy near local highs
- may overtrade in noisy environments
- can degrade quickly if thresholds are too loose

Built-in examples

Momentum Long Strategy
Uses ROC above a threshold for entry and ROC below zero for exit.

Dual Momentum Strategy
Uses both fast and slow ROC, requiring slow momentum to be strong and fast momentum to exceed slow momentum.

Dual Momentum ATR Expansion
Adds a volatility expansion requirement on top of dual momentum.

EMA PPO STDDEV
Uses PPO crossing above zero together with trend and rising volatility.

Design advice

Momentum logic is strongest when it is not used blindly. It usually improves when combined with:

- trend filters to avoid countertrend bursts
- volatility filters to avoid dead markets
- regime filters to avoid structurally poor conditions
- sensible exit logic so winning trades are not held after momentum fades

Momentum rules should also be calibrated carefully. If thresholds are too low, the strategy becomes noisy. If they are too high, it becomes too selective.

8.6 Volatility filters

A **volatility filter** controls whether the strategy is allowed to trade based on market activity level.

Volatility does not itself tell you direction, but it often tells you whether the environment is suitable for a given strategy style.

Some strategies prefer:

- **low volatility**, because they are looking for orderly conditions before a breakout
- **high or rising volatility**, because they want evidence of expansion or momentum

Typical volatility measures include:

- ATR
- NATR
- rolling standard deviation
- volatility relative to its own moving average
- range compression and expansion

Common volatility filter patterns

Trade only when volatility is quiet
Used in setups that want calm conditions before entry.

Trade only when volatility is expanding
Used in breakout or momentum logic.

Trade only when volatility is below a dynamic threshold
Used to avoid unstable conditions.

Use volatility to scale stops
Not just a filter, but also a risk control mechanism.

Built-in examples

EMA NATR Quiet Long
Requires normalized ATR to remain below a multiple of its own average. This helps avoid entering during overly noisy periods.

Dual Momentum ATR Expansion
Requires ATR to be above its average, favoring momentum moves accompanied by volatility expansion.

Intraday Volatility Breakout
Looks for volatility compression first, then enters on breakout.

SMA ATR Vortex Long
Uses ATR and a dynamic threshold to require acceptable volatility conditions.

EMA PPO STDDEV
Uses rising standard deviation as part of its entry logic.

Why volatility filters matter

A strategy can fail not because its signal is wrong, but because it is being used in the wrong volatility environment.

Examples:

- trend systems often struggle in choppy, high-noise conditions
- breakout systems struggle if volatility is already exhausted
- mean-reversion systems can be dangerous in extreme volatility spikes

Design advice

Volatility filters should answer:
What kind of market activity does this strategy need?

Use them to define:

- acceptable noise level
- expansion vs compression preference
- adaptive stop distance
- trade selectivity

Avoid using too many volatility measures at once unless each has a distinct purpose.

8.7 Volume filters

A **volume filter** requires unusual or sufficient trading activity before allowing a trade.

Volume can help separate meaningful price movement from weak movement. A move on strong volume is often more significant than the same move on weak volume.

Typical volume filters include:

- current volume above moving average volume
- current volume above moving average times a multiplier
- volume trend confirmation
- volume-based indicators such as OBV

Common volume filter patterns

Current volume above average
A simple confirmation that participation is elevated.

Current volume above average times multiplier
A stricter requirement for stronger conviction.

OBV above its moving average
A confirmation that volume flow is supportive.

Volume plus breakout
A common pattern for validating breakouts.

Built-in examples

EMA Volume ATR Long/Short
Requires volume to exceed its moving average by a multiplier before accepting EMA crossover entries.

OBV RSI Volume ATR Long
Uses both raw volume greater than its moving average and OBV crossing above its average.

Why volume filters matter

Volume filters are useful because price movement without participation can be unreliable. They can reduce false signals, especially in breakout and crossover systems.

Strengths

- improve selectivity
- help validate breakout or crossover events
- can reduce low-quality signals in quiet sessions

Weaknesses

- not all markets have equally meaningful volume data
- some instruments have distorted or synthetic volume
- overly strict volume filters can remove too many trades

Design advice

Use volume filters when participation matters to the trade idea. They are especially useful for:

- breakout logic
- momentum continuation
- crossover confirmation

They are less essential for some slow, higher-timeframe trend systems.

Always make sure the data source provides usable volume. A volume-based strategy becomes unreliable if volume is missing, flat, or unrepresentative.

8.8 Regime filters

A **regime filter** decides whether the broader market environment is suitable for the strategy.

A regime is not a single signal. It is a classification of context, such as:

- trending vs ranging
- calm vs unstable
- favorable vs unfavorable
- risk-on vs risk-off

Regime filters are extremely valuable because many strategies work only in certain environments. A good regime filter prevents the strategy from trading when its underlying assumption is unlikely to hold.

Common regime filter styles

Trend regime
Trade only when the market is directionally structured.

Volatility regime
Trade only when volatility is within a target range.

Statistical regime
Use clustering or classification to decide whether the current market resembles historically favorable periods.

Directional regime
Allow long trades only when the broader state is bullish.

Built-in examples

GMM Regime EMA Long
This is the clearest regime-based strategy in the built-in set. It classifies market states using return and volatility features, then identifies the most favorable cluster and allows trades only in that regime.

EMA NATR Quiet Long
Acts like a simpler volatility regime filter.

SMA ATR Vortex Long
Uses multiple conditions to define a favorable operating regime.

Why regime filters matter

Without a regime filter, a strategy often trades all the time. That is rarely ideal. Most strategies are specialized even if they do not appear so at first.

A regime filter makes the strategy more honest. It says:
“This setup is valid only in this kind of market.”

Design advice

A regime filter should be broad and stable. It should not behave like a second trigger. Its role is to define context, not to micromanage entries.

A good regime filter:

- changes less often than the entry trigger
- reflects the strategy’s actual assumptions
- removes clearly unsuitable environments
- improves quality more than it reduces opportunity

A bad regime filter:

- is too reactive
- duplicates the trigger logic
- adds complexity without improving interpretation

8.9 Confirmation filters

A **confirmation filter** is an additional condition that increases confidence in the entry signal.

It is not the main trigger. It is the supporting evidence.

Examples:

- trend is up, and RSI confirms strength
- breakout occurs, and volume confirms participation
- pullback is detected, and volatility remains controlled
- OBV crosses above average while volume is also above average

Confirmation filters are useful because a single signal often has too many false positives. Confirmation narrows the set of trades to those with stronger supporting evidence.

Common confirmation roles

Indicator agreement
Two different indicators support the same idea.

Market quality check
The market is active enough, quiet enough, or directional enough.

Secondary timing condition
The trade is allowed only when another indicator aligns.

Built-in examples

EMA PPO ULTOSC Long
Trend is defined by EMA structure, PPO supplies the momentum event, and ULTOSC confirms internal strength.

EMA PPO STDDEV Long
Trend is up, PPO turns positive, and standard deviation is rising.

OBV RSI Volume ATR Long
OBV cross-up is the trigger, while RSI and volume confirm quality.

SMA ATR Vortex Long
Trend, volatility condition, and Vortex agreement all work as confirmations.

Good use of confirmation

Confirmation should improve the quality of trades without making the strategy unreadable.

A strong confirmation filter:

- adds information different from the main trigger
- reduces obvious false positives
- still leaves enough trades to evaluate robustly

A weak confirmation filter:

- repeats the same concept in another form
- creates unnecessary parameter complexity
- makes the strategy too selective

If the trigger already says “momentum is strong,” adding three more momentum indicators often adds little real value. It may only add overfitting risk.

8.10 Combining multiple conditions cleanly

The most important design skill in Vectester is not inventing more conditions. It is combining conditions **cleanly**.

A clean strategy is easier to:

- understand
- debug
- optimize
- trust
- explain in a manual or report

The best way to combine conditions is to assign each one a role.

A clean condition hierarchy

Most strong strategies can be described like this:

- **Context:** Is the market environment suitable?
- **Setup:** Is the market forming the kind of opportunity we want?
- **Trigger:** What exact event starts the trade?
- **Risk logic:** How is the trade controlled after entry?
- **Exit logic:** What ends the trade if the idea fails or completes?

This structure prevents condition sprawl.

Example framework

A well-organized strategy might look conceptually like:

- Context: trend is up
- Setup: volatility is quiet
- Trigger: price crosses above EMA or PPO crosses above zero
- Risk logic: 2% stop-loss and 4% take-profit
- Exit logic: close if trend breaks

That is much clearer than mixing everything into one long unexplained boolean expression.

Keep different concepts separate

Try not to confuse:

- trend
- momentum
- volatility
- participation
- regime
- risk control

Each serves a different purpose.

Avoid redundant indicators

If two conditions express nearly the same thing, one may be enough.

Examples of possible redundancy:

- fast EMA > slow EMA
- MACD above zero
- PPO above zero

These are not identical, but they often tell a very similar story. Stacking too many similar conditions can create false confidence.

Use filters to improve quality, not to impress

Every added condition should answer:

- What problem does this solve?
- What bad trades does this remove?
- Does it add distinct information?

If the answer is unclear, the condition may not belong.

Balance simplicity and selectivity

Too few conditions can make a strategy noisy.
Too many conditions can make it brittle.

The goal is not maximum complexity. The goal is a compact set of conditions where each one has a clear job.

Preserve enough trades

A strategy with beautifully precise logic but only a handful of trades is difficult to evaluate. Clean logic should still produce enough observations for meaningful testing.

Make logic readable

Even though the strategy is coded, it should also be explainable in plain language.

A user should be able to summarize the strategy in a few lines such as:

> The strategy trades only in an uptrend. It enters when momentum turns positive during acceptable volatility conditions. It exits when trend fails or when the stop-loss or take-profit is reached.

If the logic cannot be explained clearly, it is usually too complicated.

Practical design template

A very effective way to design strategies in Vectester is to build them in layers:

Layer 1: Define market context

Decide whether the strategy needs:

- bullish trend
- bearish trend
- ranging market
- quiet volatility
- expanding volatility
- favorable regime

Layer 2: Define the setup

Describe what opportunity you want:

- pullback
- breakout
- momentum acceleration
- oversold recovery
- trend continuation

Layer 3: Define the trigger

Choose the exact event that opens the trade:

- crossover
- threshold cross
- breakout above prior range
- oscillator signal
- confirmation cross

Layer 4: Define risk controls

Decide how the trade is protected:

- fixed stop
- ATR stop
- trailing stop
- take-profit
- sizing logic

Layer 5: Define the exit thesis

Decide why the trade ends:

- signal failure
- trend break
- momentum reversal
- target reached
- stop reached
- regime deterioration

This layered approach matches the style of the built-in Vectester strategies and is the best way to build logic that remains understandable after optimization.

Final guidance

Good strategy logic is not about adding more indicators. It is about making each condition serve a clear purpose.

A well-designed Vectester strategy usually has:

- one primary trading idea
- one clear trigger
- a small number of meaningful filters
- explicit risk management
- exit logic that matches the entry thesis

That is the standard to aim for when building or modifying strategies in the application.

9. Signals and Trade Management

This section explains how Vectester strategies tell the backtest engine **when to enter**, **when to exit**, and **how to manage risk after a trade is open**.

In Vectester, every strategy's `build()` method returns a `StrategyResult` with three core outputs:

- `entries`
- `exits`
- `pf_kwargs`

These three outputs work together.

- `entries` says when a position should be opened.
- `exits` says when a position should be closed by signal.
- `pf_kwargs` carries trade-management instructions such as stop-loss, take-profit, trailing stops, short-side signals, position sizing, and direction behavior.

A strategy does not have to use all of them. Some strategies use only `entries` and `exits`. Others use `entries` plus stop-based management and set `exits = None`.

9.1 Entries

An **entry** is the event that tells Vectester to open a new position.

In practice, `entries` is a boolean time series aligned with the price data:

- `True` means "open a trade here"
- `False` means "do not open a trade here"

Most strategies build entries from one or more market conditions, such as:

- price crossing above a moving average
- momentum becoming positive
- volatility contracting before breakout
- volume expanding above normal
- trend and filter conditions aligning together

A simple example is:

- enter when `Close > EMA`

A more selective example is:

- enter when fast EMA crosses above slow EMA **and**
- volume is above its moving average **and**
- volatility conditions are favorable

The more conditions included, the fewer entries the strategy usually generates. That can improve trade quality, but it can also reduce trade count too much and make results unreliable.

What makes a good entry rule

A good entry rule should be:

- logically tied to the market behavior the strategy is trying to exploit
- simple enough to understand
- selective enough to avoid random noise
- not so restrictive that it produces very few trades

Important practical note

In many Vectester strategies, `entries` is not just a one-bar event signal. Sometimes it is a condition that stays true for multiple bars. The portfolio engine still interprets it as an entry signal stream. This means strategy authors should understand whether they are writing:

- a **crossing event**, or
- a **persistent state condition**

A crossing event is usually cleaner and less ambiguous.

9.2 Exits

An **exit** is the event that tells Vectester to close an open position by strategy logic.

Like entries, `exits` is usually a boolean time series:

- `True` means “close the trade here”
- `False` means “keep the trade open”

Typical exit logic includes:

- price crossing below a moving average
- trend reversal

- momentum weakening
- opposite signal appearing
- regime filter turning unfavorable

For example:

- enter when price crosses above EMA
- exit when price crosses below EMA

This is the most direct signal-driven trade lifecycle:
entry signal opens the trade, exit signal closes it.

Exit signals versus stop exits

There are two broad ways a strategy can close trades:

1. **Signal exits**
    - handled through `exits`
    - based on indicator logic
2. **Stop-based exits**
    - handled through `pf_kwargs`
    - based on price movement after entry

A strategy may use:

- only signal exits
- only stop exits
- both together

When both are used, the trade can close whichever way is triggered first.

When to use signal exits

Signal exits are useful when the strategy logic itself defines when the opportunity is over.
For example:

- a trend-following strategy exits when the trend flips
- a mean-reversion strategy exits when price reverts to its target zone

When `exits` can be `None`

Some Vectester strategies deliberately return:

```
exits = None
```

That means the strategy is not using an indicator-based exit signal. In that case, trades are usually managed by:

- stop-loss
- take-profit
- trailing stop
- opposite-side signal, if the strategy also provides short-side behavior

This is common in strategies where the risk framework is more important than an indicator exit.

9.3 Stop-loss

A **stop-loss** is a protective exit that limits the maximum tolerated loss on a trade.

In Vectester, stop-loss is usually passed through `pf_kwargs` using:

```
"sl_stop": value
```

The stop value is generally expressed as a **fraction of price**, not as raw currency.

Examples:

- `0.02` means 2%
- `0.01` means 1%
- `0.05` means 5%

So a stop-loss of `0.02` means the trade should be closed if price moves 2% against the entry.

Why stop-loss matters

Stop-loss exists to:

- cap downside on any single trade
- make risk more predictable
- prevent one trade from dominating the whole backtest
- support more realistic position sizing

Without stop-loss, a strategy may hold losing trades too long, which can create severe drawdowns and distort optimization.

Common stop-loss styles in Vectester strategies

Vectester strategies use several stop styles:

- fixed percentage stop
- ATR-based stop
- dynamic stop derived from a time series
- trailing stop

A fixed stop is simple and stable.

An ATR-based stop is adaptive. It becomes wider when the market is more volatile and narrower when volatility is low.

Choosing stop-loss size

A stop that is too tight may:

- cut off valid trades too early
- increase noise exits
- reduce win rate

A stop that is too wide may:

- allow overly large losses
- worsen drawdown
- reduce capital efficiency

The right stop depends on:

- timeframe
- volatility of the asset
- strategy type
- expected holding period

Trend systems often need wider stops than short-term mean-reversion systems.

9.4 Take-profit

A **take-profit** is an exit that locks in gains once price reaches a favorable target.

In Vectester, take-profit is usually passed through:

```
"tp_stop": value
```

Like stop-loss, it is generally expressed as a fraction of price.

Examples:

- `0.03` means 3%
- `0.05` means 5%

If a trade is opened and price moves that far in the profitable direction, the position is closed.

Why take-profit matters

Take-profit can be useful for:

- converting open profit into realized profit
- enforcing a defined reward target
- keeping trades shorter and more repeatable
- preventing profitable trades from reversing before exit

Fixed reward versus signal-driven exits

Some strategies prefer a fixed take-profit target.

Others prefer to let the trade continue until the signal itself weakens.

A fixed take-profit is often best when the strategy is designed to capture relatively short, repeatable moves.

A signal-driven exit is often better when the strategy is designed to ride extended trends.

Risk-reward structure

A common pattern in Vectester strategies is:

- stop-loss = `sl`
- take-profit = `sl * tp_ratio`

Example:

- `sl = 0.02`
- `tp_ratio = 2.0`
- then `tp = 0.04`

This creates a 1:2 risk-reward structure.

That does **not** guarantee profitability. It only defines the payoff shape. A strategy with a strong reward ratio can still lose if entries are poor.

9.5 Trailing stop

A **trailing stop** is a stop-loss that moves with the trade when price moves in the favorable direction.

In Vectester, this is typically enabled with:

```
"sl_trail": True
```

When trailing is active, the stop is no longer fixed at the original level. Instead, it follows price as the trade becomes more profitable.

Why use a trailing stop

A trailing stop helps:

- protect open gains
- reduce the need for a fixed take-profit
- let strong trends run longer
- close trades automatically when momentum fades

This makes trailing stops especially useful in:

- breakout systems
- trend-following systems
- volatility expansion systems

Trade-off of trailing stops

Trailing stops are not automatically better.

They may:

- preserve gains better than fixed stops
- reduce catastrophic reversals

- but also close trades too early during normal pullbacks

A very tight trailing stop can damage trend strategies because trends often retrace before continuing.

A looser trailing stop gives trades more room, but gives back more open profit.

Typical use in Vectester

Some strategies combine:

- an entry signal
- no explicit exit signal
- a stop-loss
- a trailing stop

This means the trade is opened by logic and then managed mainly by price behavior.

9.6 Static vs dynamic stop values

Stop values in Vectester can be either **static** or **dynamic**.

Static stop values

A static stop uses the same value throughout the whole test.

Example:

```
"sl_stop": 0.02
```

This means every trade uses the same 2% stop.

Static stops are:

- easy to understand
- easy to optimize
- consistent across trades
- less adaptive to changing volatility

They are often appropriate for:

- stable instruments
- simple systems
- early-stage strategy testing

Dynamic stop values

A dynamic stop changes over time.

Example idea:

- stop width expands when ATR rises
- stop width contracts when ATR falls

This is useful because market volatility is not constant. A fixed stop that works in quiet conditions may be too tight in volatile conditions.

Dynamic stops are:

- more adaptive
- often more realistic
- better suited to volatile assets
- harder to interpret and tune

When to prefer each

Use static stops when:

- the strategy is simple
- you want clean comparisons
- you are still validating basic logic

Use dynamic stops when:

- volatility changes materially over time
- the strategy depends on market regime
- fixed stops are clearly inappropriate for the asset

9.7 Series-based stop values

A **series-based stop** is a stop passed as a full time series rather than as a single scalar number.

This is important in Vectester because many strategies compute stop distances bar by bar.

Example:

```
sl_stop = atr * 2 / close
```

This produces a series where each bar has its own stop fraction.

That series can then be passed through `pf_kwargs`:

```
"sl_stop": sl_stop
```

Why series-based stops matter

Series-based stops allow the strategy to express:

- ATR-based risk
- volatility-adaptive stop distances
- regime-sensitive risk control
- bar-by-bar changes in stop logic

This is more powerful than a fixed percentage stop because it lets risk management react to current market structure.

Practical meaning

Suppose ATR rises sharply.

A static 1% stop may become unrealistically tight.

A series-based ATR stop may automatically widen, which can keep the strategy from being stopped out by normal noise.

Important design implication

Once a strategy uses series-based stop values, the stop logic becomes part of the strategy's market model, not just a generic risk setting.

That means users should evaluate it as part of the strategy's core behavior.

9.8 Long-only behavior

A **long-only** strategy opens only long positions.

It profits when price rises and loses when price falls after entry.

This is the simplest and most common behavior in Vectester.

In practice, long-only behavior can be defined in two ways:

1. the backtest is globally configured as `longonly`
2. the strategy itself enforces long-only behavior in `pf_kwargs`

Long-only strategies typically:

- buy during favorable conditions
- close when conditions weaken
- never open inverse positions

When long-only is appropriate

Long-only behavior is usually best for:

- bullish assets over long horizons
- trend-following systems intended to ride upside moves
- users who do not want short exposure
- simpler strategy design and interpretation

Benefits of long-only design

Long-only systems are often:

- easier to understand
- easier to maintain
- easier to validate
- more realistic for many retail and discretionary workflows

Limitation of long-only design

A long-only strategy may:

- sit in cash during bearish periods
- underperform in sideways regimes
- miss opportunities on the short side

Still, for many assets, long-only is the cleanest and most robust starting point.

9.9 Short-only behavior

A **short-only** strategy opens only short positions.

It profits when price falls after entry.

Short-only behavior is less common, but it is important in systems built for bearish conditions or downside momentum.

In Vectester, short-only behavior can come from:

- global direction set to `shortonly`
- a strategy structure built around short-side entries and exits

How short behavior differs conceptually

A short trade is not just a mirrored long trade.

Downside moves often behave differently:

- they can be faster
- more volatile
- more mean-reverting
- more sensitive to news shocks

Because of this, short-only strategies often need different parameter choices and risk control than long-only strategies.

Why short-only is harder

Short-only systems often face:

- sharper reversals
- faster stop-outs
- more unstable trends
- lower tolerance for loose risk control

That makes stop design especially important.

Good use cases

Short-only behavior can be useful when:

- testing bearish regimes specifically
- building a hedging strategy
- evaluating whether a logic block works symmetrically
- combining long and short modules later into a full system

9.10 Both-directions behavior

A **both-directions** strategy can trade long and short.

This is the most flexible form of behavior, but also the most complex.

In Vectester, both-direction behavior usually requires more than the basic `entries` and `exits`. A strategy may also provide:

- `short_entries`
- `short_exits`
- `direction: "both"`

through `pf_kwargs`.

That allows the strategy to define separate logic for:

- opening long positions
- closing long positions
- opening short positions
- closing short positions

Why both-directions is powerful

A bidirectional strategy can:

- participate in bullish and bearish conditions
- reduce idle time
- exploit more market structure
- improve capital usage

Why both-directions is harder to design

It is harder because the long side and short side are often not truly symmetrical.

For example:

- a trend filter that works well for longs may not work well for shorts
- stop widths may need to differ
- entry timing may need to differ
- volatility behavior may differ sharply between up and down moves

A poor both-direction strategy often just copies long logic onto the short side without adapting it. That usually leads to weak or misleading results.

Best practice for bidirectional strategies

Treat long and short logic as related but separate systems.

A good both-direction design should answer:

- what qualifies a long trade?
- what qualifies a short trade?
- how are long exits handled?
- how are short exits handled?
- are the same stops valid for both sides?
- is the same sizing valid for both sides?

If those answers are not clear, the strategy is usually not ready.

Practical design guidance

When writing or evaluating a Vectester strategy, think of trade management as a hierarchy:

1. **Entry logic** decides when an opportunity begins.
2. **Exit logic** decides when the thesis is no longer valid.
3. **Stop-loss** protects against adverse movement.
4. **Take-profit** locks in defined gains.
5. **Trailing stop** protects profits while allowing continuation.
6. **Direction rules** determine whether the strategy participates only on the long side, only on the short side, or both.

A strategy becomes much easier to trust when each of these parts has a clear purpose.

A weak strategy usually shows one or more of these problems:

- entries are vague or overly frequent
- exits are inconsistent with the entry logic
- stops are arbitrary
- take-profit is chosen only to improve backtest results

- trailing stop is too tight or too loose
- long and short behavior are mixed without clear design

A strong strategy usually has:

- a clear market idea
- entry logic tied to that idea
- exit logic tied to invalidation or completion of the move
- stop placement consistent with expected volatility
- direction behavior chosen deliberately, not accidentally

Summary

Signals and trade management define the life cycle of every trade in Vectester.

- **Entries** open positions.
- **Exits** close positions by signal.
- **Stop-loss** limits downside.
- **Take-profit** captures targeted upside.
- **Trailing stop** protects gains while letting trades run.
- **Static stops** use one fixed value.
- **Dynamic stops** adapt over time.
- **Series-based stops** allow bar-by-bar risk control.
- **Long-only**, **short-only**, and **both-directions** determine what side of the market the strategy can trade.

In practice, the quality of a strategy depends not just on its entry logic, but on how all of these pieces work together.

10. Portfolio Keyword Arguments

Portfolio keyword arguments, referred to in Vectester as `pf_kwargs`, are the strategy's direct way of controlling how trades are executed and managed after signals are generated.

A strategy in Vectester does not only decide **when to enter** and **when to exit**. It can also send additional execution instructions back to the backtesting engine. These instructions are returned inside the `pf_kwargs` dictionary as part of the strategy result. In practice, this is how a strategy says things such as:

- use a stop-loss
- use a take-profit
- trail the stop-loss as price moves
- allow only long trades
- allow both long and short trades
- use special position sizing
- provide short-entry and short-exit signals

This makes `pf_kwargs` one of the most important parts of strategy authoring in Vectester. It is the bridge between a strategy's signal logic and the actual portfolio behavior used during the backtest.

10.1 What `pf_kwargs` Is For

Every strategy returns three things:

- `entries`
- `exits`
- `pf_kwargs`

The first two define the trade signals. The third defines how the backtest engine should handle those signals.

Conceptually:

```
return StrategyResult(
    entries=entries,
    exits=exits,
    pf_kwargs={...}
)
```

The `pf_kwargs` dictionary is passed into the portfolio engine together with the signals. This means it can influence the backtest just as strongly as the signals themselves.

Typical uses of `pf_kwargs` in Vectester include:

- stop-loss control with `sl_stop`
- take-profit control with `tp_stop`
- trailing-stop activation with `sl_trail`
- direction control with `direction`
- short trade support with `short_entries` and `short_exits`
- custom position sizing with `size` and `size_type`

Important idea:

A strategy can open a trade with an entry signal, but the actual life of that trade may then be controlled by `pf_kwargs` rather than by a normal exit signal. For example, a strategy may set:

- `entries = ...`
- `exits = None`
- `pf_kwargs = {"sl_stop": ..., "tp_stop": ...}`

In that case, the trade is not closed by a normal rule-based exit signal. It is closed by stop-loss or take-profit logic.

10.2 `sl_stop`

`sl_stop` defines the stop-loss distance.

It tells the portfolio engine how far price may move against the position before the trade is closed automatically.

In Vectester strategies, `sl_stop` is usually expressed as a **fraction of price**, not as an absolute currency amount.

Examples:

- `0.01` means 1%
- `0.02` means 2%
- `0.05` means 5%

So this:

```
pf_kwargs={"sl_stop": 0.02}
```

means the trade is closed if price moves 2% against the position.

Common ways `sl_stop` is used

Fixed scalar stop-loss

The same stop-loss percentage is used for every bar and every trade.

```
pf_kwargs={"sl_stop": 0.02}
```

This is the simplest form.

Series-based stop-loss

A strategy can provide a full time series instead of a single number. This allows the stop distance to vary over time.

Example: ATR-based stop.

```
sl_stop = atr * 2 / close
pf_kwargs={"sl_stop": sl_stop}
```

This means the stop distance changes from bar to bar based on volatility.

This is much more adaptive than a fixed percentage stop.

Per-combination stop-loss during optimization

When optimization is run, different parameter combinations may generate different stop values. Vectester builds a stop matrix internally so each tested parameter combination gets its own stop-loss values.

That is why strategies are free to return different `sl_stop` values for different parameter sets.

Practical interpretation

A smaller `sl_stop`:

- cuts losses faster
- reduces damage per losing trade
- may also stop out good trades too early
- usually increases sensitivity to noise

A larger `sl_stop`:

- gives the trade more room

- may work better in volatile markets
- increases potential loss per trade
- may reduce over-trading caused by tight stops

In the built-in strategies

Examples from the source include:

- constant fractional stops such as `sl = 0.02`
- ATR-derived dynamic stops such as `sl_stop = stop_dist / close`
- strategies that return `sl_stop` together with `tp_stop`
- strategies that use `sl_stop` with `sl_trail=True`

Important note

`sl_stop` is not the same thing as an exit signal. It is a portfolio-level risk rule. Even if `exits` is `None`, trades can still close because `sl_stop` is active.

10.3 `tp_stop`

`tp_stop` defines the take-profit distance.

It tells the portfolio engine how far price should move in favor of the trade before the trade is closed automatically with a profit target.

Like `sl_stop`, this is typically expressed as a **fraction of price**.

Examples:

- `0.03` means 3%
- `0.06` means 6%

```
pf_kwargs={"tp_stop": 0.04}
```

means the trade is closed once it reaches a 4% favorable move.

Common ways `tp_stop` is used

Fixed scalar take-profit

```
pf_kwargs={"tp_stop": 0.04}
```

All trades use the same target.

Take-profit based on another parameter

A common pattern is to define take-profit as a multiple of stop-loss.

Example:

```
sl = 0.02
tp = sl * 2.0
pf_kwargs={"sl_stop": sl, "tp_stop": tp}
```

This produces a 2:1 reward-to-risk target.

Series-based take-profit

A strategy can also provide a full series, such as a volatility-adjusted target.

```
tp_stop = atr * 4 / close
pf_kwargs={"tp_stop": tp_stop}
```

This makes the target expand and contract with volatility.

Practical interpretation

A smaller `tp_stop`:

- locks profits more quickly
- increases the frequency of modest winners
- may cap strong trends too early

A larger `tp_stop`:

- gives trades room to run
- may improve payoff on strong trends
- may reduce win rate because targets are harder to reach

Relationship with `sl_stop`

`sl_stop` and `tp_stop` are often designed together.

Their ratio defines the strategy's reward-to-risk structure.

Examples:

- `sl_stop = 0.02`, `tp_stop = 0.02` gives a 1:1 structure
- `sl_stop = 0.02`, `tp_stop = 0.04` gives a 2:1 structure
- `sl_stop = 0.01`, `tp_stop = 0.05` gives a 5:1 structure

A strategy with a wide target may tolerate a lower win rate if winners are much larger than losers.

Important note

Like `sl_stop`, `tp_stop` is a portfolio-level trade-management rule, not a normal signal exit. A strategy can rely entirely on `tp_stop` and `sl_stop` with `exits=None`.

10.4 `sl_trail`

`sl_trail` controls whether the stop-loss trails price.

When `sl_trail` is set to `True`, the stop-loss is no longer just a fixed protective line set once at entry. Instead, it moves in the trade's favor as price moves favorably, helping lock in gains.

Example:

```
pf_kwargs={"sl_stop": 0.01, "sl_trail": True}
```

This means a 1% trailing stop-loss is used.

How trailing stops behave conceptually

For a long trade:

- if price rises, the stop moves up
- if price later falls, the stop does not move back down
- once price hits the trailing stop, the trade is closed

For a short trade, the logic is mirrored.

When `sl_trail` is useful

Trailing stops are useful when a strategy aims to:

- capture trends
- reduce the need for explicit exit logic
- protect profits automatically
- let winners continue while still managing risk

When `sl_trail` can be harmful

Trailing stops can also:

- exit too early in choppy markets
- cut trades during normal pullbacks
- reduce the size of big winners if the trailing distance is too tight

In Vectester's source behavior

Some built-in strategies expose trailing-stop use directly as a strategy parameter, such as:

- `sl_trail = True`
- `use_trailing_sl = True`

That means the user can turn trailing behavior on or off from the strategy parameter panel.

Important limitation during optimization

In the optimization engine, `sl_trail` is treated as a scalar setting, not a per-combination time series. In other words, it must remain a simple single True/False style setting when the engine builds the portfolio.

So `sl_trail` is not handled like a stop matrix. It is handled as a single portfolio argument.

10.5 `direction`

`direction` controls which trade directions the portfolio is allowed to take.

In Vectester, the global backtest page already provides a Direction control with these options:

- `longonly`
- `shortonly`
- `both`

A strategy can also place its own `direction` value inside `pf_kwargs`.

Examples:

```
pf_kwargs={"direction": "longonly"}
```

or

```
pf_kwargs={"direction": "both"}
```

Meaning of each mode

`longonly`

Only long positions are allowed.

The strategy may buy first and profit when price rises.

This is the most common mode for equity-style trend and momentum systems.

`shortonly`

Only short positions are allowed.

The strategy may sell short first and profit when price falls.

This is less common in the built-in set but supported by the backtest engine.

`both`

Both long and short positions are allowed.

This is required for strategies that produce both:

- `entries` and `exits` for long-side behavior
- `short_entries` and `short_exits` for short-side behavior

A built-in example is the EMA/Volume/ATR long-short strategy, which explicitly returns:

- `short_entries`
- `short_exits`
- `direction = "both"`

Why strategy-level direction matters

Some strategies are structurally long-only by design.

For example, a long-only pullback or trend strategy should not depend on the user remembering to choose `longonly` every time. Setting:

```
pf_kwargs={"direction": "longonly"}
```

inside the strategy makes that intent explicit and keeps behavior consistent.

This is safer and clearer than leaving it entirely to the global UI setting.

10.6 Strategy-Level Settings vs Global Backtest Settings

Vectester has two levels of portfolio control:

Global backtest settings

These are selected on the Backtest page and apply generally to the run:

- `direction`
- `init_cash`
- `fees`

- `slippage`
- `freq`

These form the base portfolio settings for the run.

Strategy-level settings

These are returned by the strategy in `pf_kwargs`.

They may include:

- `sl_stop`
- `tp_stop`
- `sl_trail`
- `direction`
- `short_entries`
- `short_exits`
- `size`
- `size_type`

Why both levels exist

The global level is for broad run configuration.

The strategy level is for logic-specific execution behavior.

Examples:

- `fees` usually belongs at the global level because it is part of the trading environment
- `sl_stop` often belongs at the strategy level because it is part of the strategy design
- `direction` may belong to either level depending on whether the strategy is intrinsically long-only or whether the user should be free to choose direction

Good design principle

Use the global settings for assumptions that should apply to any strategy on that run.

Use strategy-level `pf_kwargs` for behavior that is part of the strategy's identity.

That keeps strategy logic self-contained and avoids accidental misuse.

10.7 Merge Behavior and Precedence

This is one of the most important details for advanced users.

Vectester first builds a dictionary from the global backtest settings. Then it merges in the strategy's `pf_kwargs`.

Conceptually, the behavior is:

```
pf_kwargs = dict(global_settings)
pf_kwargs.update(strategy_pf_kwargs)
```

This means:

strategy-level values override global values when the same key appears in both places.

Example: direction override

Suppose the user selects:

```
Direction = shortonly
```

in the Backtest page.

But the strategy returns:

```
pf_kwargs={"direction": "longonly"}
```

The final portfolio will use:

```
direction = "longonly"
```

because the strategy-level value replaces the global one.

Example: adding stops without replacing fees

If global settings are:

```
{
    "direction": "longonly",
    "init_cash": 10000,
    "fees": 0.001,
    "slippage": 0.0005,
    "freq": "1d"
}
```

and the strategy returns:

```
{
    "sl_stop": 0.02,
    "tp_stop": 0.04
}
```

the final merged result becomes:

```
{
    "direction": "longonly",
    "init_cash": 10000,
    "fees": 0.001,
    "slippage": 0.0005,
    "freq": "1d",
    "sl_stop": 0.02,
    "tp_stop": 0.04
}
```

So the strategy adds new behavior while preserving the global assumptions.

Special optimization behavior

During optimization, Vectester handles certain keys specially:

- `sl_stop`
- `tp_stop`
- `sl_trail`

This happens because each parameter combination may need its own stop values.

For optimization:

- `sl_stop` and `tp_stop` may be rebuilt into matrices so each parameter combination gets the right stop values
- `sl_trail` must remain scalar, so it is handled separately

This is why stop-related arguments are not merged in exactly the same simple way as ordinary scalar keys during multi-parameter optimization.

Walk-forward behavior

In walk-forward analysis, the merge is simpler: base backtest settings are copied, then strategy `pf_kwargs` are applied on top. So strategy-level values also take precedence there.

Practical rule

If the same setting exists in both places, assume the strategy wins.

That is the safest mental model for understanding actual run behavior.

10.8 Practical Examples

Example 1: Fixed stop-loss and take-profit

A simple long strategy enters above an EMA and lets stop and target manage exits.

```python
def build(self, data, params):
    close = data["Close"]
    ema = close.ewm(span=20).mean()

    entries = close > ema

    return StrategyResult(
        entries=entries,
        exits=None,
        pf_kwargs={
            "sl_stop": 0.02,
            "tp_stop": 0.04
        }
    )
```

Behavior:

- opens long trades when `close > ema`
- has no normal exit signal
- closes trades at either 2% loss or 4% profit

This is clean and easy to understand.

Example 2: ATR-based dynamic stop and target

A volatility-aware strategy uses wider stops in volatile markets and tighter stops in calm markets.

```python
def build(self, data, params):
    close = data["Close"]
    atr = ...
    entries = ...

    sl_stop = 2.0 * atr / close
    tp_stop = 4.0 * atr / close

    return StrategyResult(
        entries=entries,
        exits=None,
        pf_kwargs={
            "sl_stop": sl_stop,
            "tp_stop": tp_stop
        }
    )
```

Behavior:

- stop and target vary over time

- risk adapts to volatility
- better suited to markets with changing conditions

This pattern appears in several built-in strategies.

Example 3: Trailing-stop breakout strategy

A breakout strategy enters when price escapes a range and uses a trailing stop to stay in the move.

```
def build(self, data, params):
    entries = ...
    return StrategyResult(
        entries=entries,
        exits=None,
        pf_kwargs={
            "sl_stop": 0.01,
            "sl_trail": True
        }
    )
```

Behavior:

- enters on breakout
- uses a 1% trailing stop
- no take-profit cap
- designed to hold trends until they weaken

This is a common pattern for momentum and breakout systems.

Example 4: Long-short strategy with explicit direction

A strategy supports both bullish and bearish trades and provides separate short signals.

```
def build(self, data, params):
    entries = ...
    exits = ...
    short_entries = ...
    short_exits = ...

    return StrategyResult(
        entries=entries,
        exits=exits,
        pf_kwargs={
            "short_entries": short_entries,
            "short_exits": short_exits,
            "direction": "both"
```

```
        }
    )
```

Behavior:

- allows long and short trading
- uses separate signals for each side
- ensures the portfolio engine is allowed to trade both directions

Without `direction="both"`, the short side may not behave as intended.

Example 5: Strategy overrides the global direction

Suppose the user chooses `both` in the Backtest page, but the strategy is intended to be long-only.

```
return StrategyResult(
    entries=entries,
    exits=exits,
    pf_kwargs={"direction": "longonly"}
)
```

Behavior:

- the strategy forces long-only behavior
- the global direction selection is overridden
- this protects the strategy from being run in an unintended mode

This is often a good idea for strategies whose logic was only designed and tested for one side of the market.

Example 6: Custom sizing together with stops

A more advanced strategy can manage not only exits but also position size.

```
return StrategyResult(
    entries=entries,
    exits=exits,
    pf_kwargs={
        "size": size_series,
        "size_type": "amount",
        "sl_stop": sl_stop,
        "tp_stop": tp_stop,
        "direction": "both"
    }
)
```

Behavior:

- trade size changes over time
- stop-loss and take-profit are still active
- long and short trades are both allowed

This shows that `pf_kwargs` is broader than just stop management. It is the strategy's full execution-control layer.

Summary

`pf_kwargs` is where a strategy defines how its signals should be traded, managed, and constrained.

The most important arguments in the current Vectester build are:

- `sl_stop` for stop-loss distance
- `tp_stop` for take-profit distance
- `sl_trail` for trailing stop behavior
- `direction` for allowed trade direction

The key rule to remember is this:

global backtest settings provide the base behavior, but strategy-level `pf_kwargs` can extend or override that behavior.

For serious strategy development in Vectester, understanding `pf_kwargs` is essential. A strategy is not fully defined by its entry and exit signals alone. It is defined by the combination of:

- signals
- stop logic
- direction rules
- position behavior
- portfolio execution settings

That full combination is what the user is actually testing.

11. Editing Existing Strategies

Vectester allows you to modify, rename, and remove strategies directly from the application. This is one of the most powerful parts of the platform, because it lets you evolve trading ideas without leaving the research environment. At the same time, strategy editing changes the logic that generates signals, the parameters exposed in the interface, and sometimes the way optimization behaves. For that reason, edits should be made carefully and methodically.

A strategy in Vectester is not just a label in a dropdown. It is a complete definition containing:

- a strategy class
- a unique displayed name
- a list of declared parameters
- the signal-building logic
- any extra portfolio instructions such as stops or trailing exits

When you edit a strategy, you are potentially changing all of those layers at once.

11.1 Modifying a strategy safely

A safe modification process starts with understanding what kind of change you are making. Not all edits carry the same risk.

Minor changes are usually low risk. These include:

- changing a parameter description
- adjusting a default value
- improving comments or formatting
- slightly refining a filter threshold
- adding a small confirmation condition

Major changes are higher risk. These include:

- changing entry or exit logic
- changing parameter names
- changing parameter types
- adding or removing parameters

- changing stop-loss or take-profit handling
- changing the overall structure of the strategy

A disciplined editing workflow usually follows this order:

1. Review the current strategy logic before changing anything.
2. Decide whether the change is cosmetic, structural, or behavioral.
3. Make one small change at a time.
4. Save the strategy.
5. Confirm that the strategy still loads correctly.
6. Re-check the parameter form in the interface.
7. Run a simple backtest with default settings.
8. Only then run optimization, walk-forward, or Monte Carlo again.

This process matters because a strategy may still load successfully even when its trading behavior has changed dramatically. A syntactically valid strategy is not necessarily a logically correct one.

Good editing habits

Keep the strategy's purpose stable unless you intentionally redesign it.
For example, if a strategy is meant to be a trend-following system, avoid gradually turning it into a mean-reversion system through multiple small edits without also renaming or documenting that shift.

Change one concept at a time.
If you change the EMA period, add an ATR filter, and modify the stop logic all at once, it becomes difficult to understand which edit improved or harmed performance.

Retest default behavior first.
Before optimizing an edited strategy, run it once using the default parameters. This reveals whether the strategy still functions in a basic way.

Watch for "silent failures."
Some edits do not produce an error, but they produce no trades, extremely unstable results, or invalid metrics. These are logic failures rather than load failures.

Treat edited strategies as new research versions.
If a modification changes the strategy idea materially, document it as a new version or new strategy name rather than overwriting the old identity without explanation.

11.2 How changes affect the parameter forms

Vectester builds the strategy parameter forms from the strategy's declared parameter definitions. This means the interface is not manually hard-coded for each strategy. Instead, the application reads the strategy's parameter list and generates the input fields automatically.

Because of that design, any change you make to the declared parameters immediately affects the interface.

This includes changes to:

- parameter names
- default values
- parameter types
- parameter descriptions
- the number of parameters

What happens when you change a parameter name

If you rename a parameter, the old field disappears and a new field appears under the new name. From the application's perspective, this is not the "same parameter with a new label." It is usually treated as a different parameter.

Example:

Before:

- `ema_window`

After:

- `trend_ema_window`

The interface will show the new field name. Any run-once values or optimization grid values previously associated with `ema_window` no longer match automatically.

What happens when you change a default value

If you keep the same parameter name and type but change the default value, the form will usually present the new default. This affects the starting point for new runs and influences how the strategy behaves when the user does not override the value.

Example:

- default RSI entry threshold changes from `30` to `25`

This means fresh runs now begin with a stricter oversold threshold, even if the user does nothing else.

What happens when you change a parameter type

Changing a type is more disruptive than changing a default.

Example:

- changing `atr_mult` from `float` to `int`
- changing `use_filter` from `bool` to `float`

When this happens, the generated form control may change, the way values are parsed may change, and any old saved mental assumptions about how to enter values may no longer apply.

What happens when you add a parameter

A new field appears in the parameter form. The strategy now expects that parameter to exist in its logic. If the new parameter has a sensible default, the strategy may continue to run smoothly. If not, users may immediately experience confusing behavior or poor results.

Good practice is to add new parameters only when they represent a real decision the user should control.

What happens when you remove a parameter

The field disappears from the form. This simplifies the strategy, but it also invalidates any previous workflows that relied on tuning that parameter.

Removing parameters can improve robustness if the strategy had become too complex.

What happens when you change parameter descriptions

Descriptions help users understand what each parameter means. Updating them does not change logic directly, but it changes how users interpret and use the strategy. Good descriptions are especially important when a parameter has a subtle effect, such as controlling a volatility filter, a stop distance, or an entry confirmation threshold.

11.3 When old parameter values stop matching

Old parameter values stop matching when the new strategy definition no longer lines up with the previous form structure or expected input types.

This usually happens in five situations.

1. A parameter is renamed

If a parameter used to be called `fast` and is renamed to `fast_ema`, any previous understanding or entered values for `fast` no longer map directly.

The app sees:

- old name: gone
- new name: new field

This means old run setups are no longer directly compatible in a one-to-one way.

2. A parameter is removed

If a parameter disappears, any previous use of it becomes irrelevant. This is usually harmless, but it means earlier optimization grids or strategy interpretations no longer apply.

3. A parameter is added

If a new parameter is introduced, previous runs did not account for it. Even if the strategy still runs because the new parameter has a default, the behavior is no longer exactly comparable to earlier results.

4. A parameter type changes

Suppose a parameter was previously used as:

- integer window length

and is later changed to:

- floating threshold

Even if the parameter name stays the same, older values may no longer make conceptual or practical sense.

5. The internal meaning changes even if the name stays the same

This is one of the most dangerous situations.

Example:

- `threshold` used to mean “minimum ROC to enter”
- now it means “maximum pullback allowed before entry”

The field name is unchanged, so the form looks familiar, but the meaning has changed completely. Old values may still fit numerically while being logically wrong.

Why this matters

When old parameter assumptions stop matching, several problems can appear:

- backtests generate no trades
- optimization results become meaningless

- the strategy behaves very differently from before
- comparisons across old and new runs become invalid
- users believe they are testing the same system when they are not

Best practice

Whenever you make a breaking change to parameters, do at least one of the following:

- rename the strategy
- document the change clearly
- reset and re-check all run-once values
- rebuild the optimization grid from scratch
- treat the edited strategy as a new version

A useful rule is this: if a user would misunderstand the strategy by relying on the old parameter assumptions, the change is large enough to deserve explicit versioning or renaming.

11.4 Renaming a strategy

A strategy's displayed identity comes from its declared strategy name, not only from the file name. Renaming a strategy changes how it appears in the interface and how users distinguish it from other strategies.

Renaming can be helpful when:

- the strategy has evolved substantially
- you want to preserve the original version conceptually
- you want clearer naming for users
- you are creating variants of a base idea
- you want to signal a different market regime or logic focus

Examples of good reasons to rename:

- `triple_ema` becomes `triple_ema_trend_filter`
- `momentum_long` becomes `momentum_long_v2`
- `ema_sltp` becomes `ema_pullback_sltp`

When renaming is recommended

Renaming is strongly recommended when the strategy's core behavior changes, especially if any of the following are true:

- entry logic changed materially
- exit logic changed materially
- parameter meaning changed
- market regime target changed
- the strategy is no longer directly comparable to earlier results

Strategy name vs file name

These are related but not identical concepts.

The file name helps the strategy live in the strategy folder and affects organization.
The declared strategy name is what the app uses for display and identification inside the interface.

For a clean setup, both should remain sensible and aligned. If they drift too far apart, users may become confused when editing, troubleshooting, or comparing results.

Good naming principles

Use names that are:

- unique
- descriptive
- stable
- easy to distinguish in dropdowns and comparison tables

Avoid names that are:

- vague
- overly generic
- too similar to other strategies
- misleading about what the strategy does

Poor examples:

- `test`

- `mystrategy`
- `new_one`
- `ema2`

Better examples:

- `ema_rsi_trend_long`
- `dual_momentum_atr_expansion`
- `linreg_pullback_long_v2`

Renaming and historical interpretation

After renaming a strategy, earlier results associated with the old name should not be casually treated as identical to the renamed one unless the logic truly remained the same. A new name should communicate whether the strategy is:

- the same concept with better labeling, or
- a genuinely new variant

11.5 Removing a strategy

Removing a strategy deletes it from the active set of available strategies. This is useful when cleaning up experiments, eliminating broken drafts, or reducing clutter in the strategy list.

However, deletion is a stronger action than modification. Once a strategy is removed, it is no longer selectable in the interface unless it is recreated.

Good reasons to remove a strategy

- it is a failed experiment
- it is obsolete
- it duplicates another strategy
- it was only a temporary test
- it contains logic you no longer want users to access
- it causes confusion due to poor naming or incomplete design

Risks of removing a strategy

Removing a strategy can create practical problems if:

- it was still being compared against other runs
- users still rely on its parameter conventions
- it represented an earlier version worth preserving
- it served as a useful template for future development

Better alternatives to immediate removal

Before deleting, consider whether the strategy should instead be:

- renamed as an archive version
- simplified and kept as a teaching example
- duplicated into a cleaner successor
- documented as deprecated rather than erased immediately

Removal discipline

A good rule is:

- remove only when the strategy has no lasting research value
- rename or archive when it still has historical or educational value

For serious research environments, preserving older versions is often preferable to deleting them outright, especially when those versions explain how the current system evolved.

11.6 Avoiding duplicate strategy names

Every strategy should have a unique identity. Duplicate strategy names are one of the most common avoidable problems in a plugin-style strategy system.

If two strategies use the same declared name, the application may not be able to distinguish them reliably. This can lead to confusion in loading, selection, display, editing, or comparison.

Why duplicate names are a problem

The strategy name is used as a user-facing identifier. If two strategies share that same identity:

- the dropdown becomes ambiguous
- edits may seem to apply to the wrong strategy
- users may not know which variant they are running

- comparison results become difficult to interpret
- registry behavior may become unpredictable or error-prone

Even if the files are different, duplicate displayed names are still a problem because the user interacts with names, not only file paths.

Common causes of duplicate names

- copying an existing strategy and forgetting to change its `name`
- creating a new version but leaving the old identifier untouched
- using generic placeholders such as `test_strategy`
- saving variants quickly during experimentation

How to avoid duplicates

Use a naming convention that makes versions and variants explicit.

Examples:

- `ema_rsi_trend_long`
- `ema_rsi_trend_long_v2`
- `ema_rsi_trend_long_atr_filter`
- `ema_rsi_trend_long_fast`

This makes relationships visible without creating ambiguity.

Recommended naming convention

A good strategy name often follows this pattern:

`core_logic` + `filter_or_variant` + `direction_or_version`

Examples:

- `dual_momentum_long`
- `dual_momentum_atr_long`
- `linreg_pullback_long_v2`

This is clearer than generic numbering alone.

Practical rule

Before saving a new or modified strategy, always confirm:

- no other strategy already uses that exact name
- the name still matches the logic
- the name is specific enough to remain clear months later

If two strategies need similar names because they are close relatives, make the distinction explicit in the name itself.

11.7 Recovering from load errors

A load error happens when a strategy cannot be loaded correctly into the application. This usually means the strategy will fail to appear properly, fail to generate a usable parameter form, or fail during execution.

Load errors are normal during development. They do not necessarily mean the strategy idea is wrong. They usually mean the strategy definition is incomplete, inconsistent, or invalid in some way.

Common causes of load errors

Syntax problems
Simple code mistakes can prevent the strategy from loading at all. These include missing punctuation, invalid indentation, incomplete statements, or malformed declarations.

Missing required elements
A valid strategy generally needs a proper class definition, a unique strategy name, declared parameter definitions, and a build method that returns a valid result structure.

Broken parameter declarations
If a parameter is declared incorrectly, the application may fail to build the form or may mis-handle values.

Invalid return structure
If the strategy does not return the expected signal/result object, it may load partially or fail at run time.

Name conflicts
Duplicate strategy names may create registry issues or confusing load behavior.

Logic that produces incompatible signals
Even if the strategy loads, it may fail when run if entries, exits, or stop instructions are not in a usable form.

Symptoms of a load problem

You may see one or more of these symptoms:

- the strategy does not appear in the dropdown

- the strategy appears but the parameter form is wrong or incomplete
- the strategy loads but crashes when run
- the strategy produces immediate errors in the log
- the strategy loads but always returns invalid results

Recovery process

A practical recovery workflow is:

1. Read the log carefully

The log is the first place to look. It often tells you whether the problem is:

- loading
- parameter generation
- execution
- optimization
- result calculation

Do not guess until you read the actual error message.

2. Check the basic structure

Confirm that the strategy still has:

- a valid strategy class
- a unique strategy name
- a proper parameter list
- a valid build method
- a valid returned result

3. Undo the last major edit mentally

Ask what changed immediately before the failure:

- parameter rename?
- new stop logic?
- changed type?

- renamed class?
- renamed strategy?
- removed exit logic?

Most load problems begin with the most recent structural edit.

4. Reduce complexity

If the strategy has become difficult to debug, simplify it temporarily:

- remove optional filters
- remove advanced stop logic
- reduce to one or two parameters
- use simple entry and exit conditions

Once the basic skeleton works again, reintroduce complexity gradually.

5. Test after each fix

Do not apply many fixes at once. After each correction:

- save
- reload
- check the strategy list
- check the parameter form
- run a simple backtest

This makes the true cause easier to identify.

Recovering from specific problem types

Strategy does not appear at all

Likely causes:

- serious load failure
- invalid class definition
- duplicate naming conflict
- broken file content

What to do:

- verify the strategy has a valid name and structure
- ensure the name is unique
- simplify the file to a minimal working version first

Strategy appears but form fields are wrong

Likely causes:

- broken parameter definitions
- changed types
- renamed parameters
- malformed parameter metadata

What to do:

- review every parameter definition
- make sure each parameter has a clear name, default, and correct type
- confirm the form matches the intended user-facing inputs

Strategy runs but fails immediately

Likely causes:

- invalid signal creation
- missing data assumptions
- bad stop configuration
- invalid return object

What to do:

- simplify the build logic
- confirm the returned entries/exits/stops are valid conceptually
- test with default parameters only

Strategy runs but yields no trades

Likely causes:

- overly strict conditions
- broken thresholds
- parameter mismatch after edits
- logical contradiction in entry rules

What to do:

- loosen conditions
- inspect parameter values
- verify that the edited logic can actually trigger in real data

Best recovery principle

When a strategy breaks, do not try to save the whole complex version immediately. First restore a minimal working version. Then add features back one by one until the problem reappears. That is the fastest way to isolate the real cause.

Summary

Editing strategies in Vectester is not just a convenience feature. It is a core research capability. A strategy's identity, parameter forms, optimization behavior, and signal generation are all connected. Small edits can have large effects.

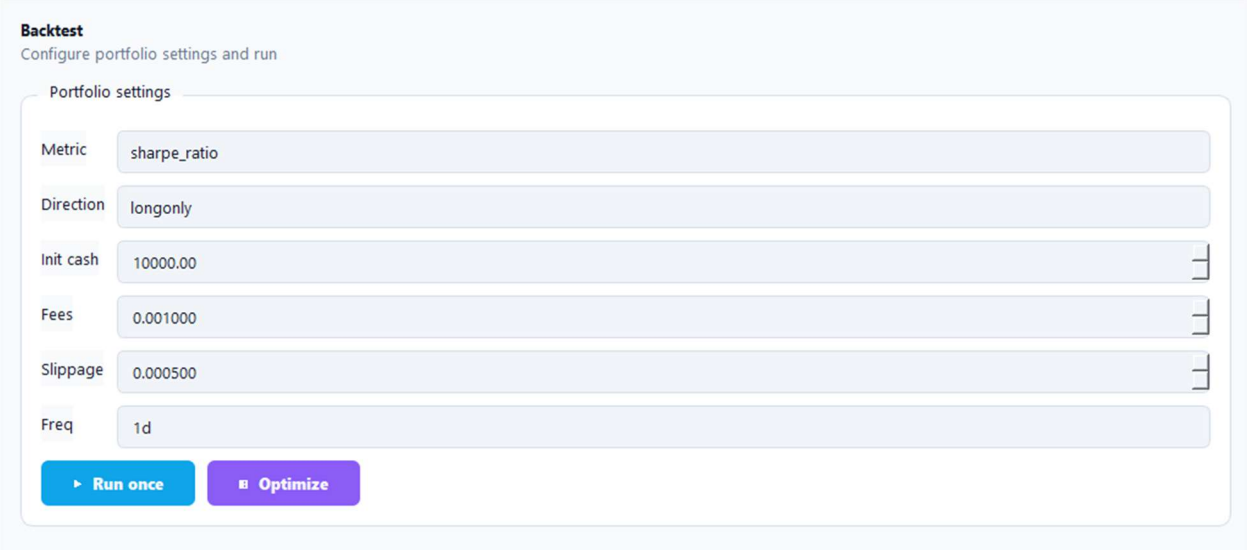
The safest approach is to:

- edit incrementally
- keep names unique and meaningful
- treat parameter changes as potentially breaking
- rename strategies when their behavior changes materially
- remove strategies only when they no longer matter
- use the log and a minimal working version to recover from errors

Handled well, strategy editing turns Vectester into a full strategy-development studio rather than only a backtest viewer.

12. Backtest Settings

The **Backtest** page controls the portfolio-level assumptions used when Vectester converts a strategy's signals into an actual backtest result. These settings do not define the strategy logic itself. Instead, they define the environment in which the strategy is evaluated: what counts as success, how trades are executed, how much capital is available, what trading costs are assumed, and how performance ratios are annualized.



This distinction matters. A strategy may produce the same entry and exit signals under two different backtest configurations, yet the final return, drawdown, Sharpe ratio, and even the apparent robustness of the strategy can change significantly.

In practice, the backtest settings answer questions such as:

- What metric should optimization try to maximize?
- Is the strategy allowed to trade long, short, or both?
- How much starting capital does the portfolio begin with?
- What transaction cost assumptions are applied?
- How much slippage is assumed between signal and execution?
- What time frequency should be used when annualizing risk-adjusted metrics?

These settings should be chosen deliberately. Poor assumptions can make a weak strategy look strong or make a good strategy look worse than it really is.

12.1 Metric Selection

The **Metric** setting determines which performance measure Vectester uses as the main objective when comparing results, especially during optimization.

When running a single backtest, the selected metric is still important because it becomes the primary headline measure used for interpretation. When running an optimization, it becomes even more important because Vectester uses it to decide which parameter combination is "best."

What metric selection does

If you optimize a strategy across many parameter combinations, each combination produces a portfolio result. Vectester calculates the chosen metric for each one, then ranks the combinations according to that metric.

For example:

- If the selected metric is **Sharpe ratio**, Vectester prefers the parameter set with the best risk-adjusted return.
- If the selected metric is **total return**, Vectester prefers the parameter set with the highest raw profit.
- If the selected metric is **max drawdown**, the logic changes: the best result is the one with the smallest drawdown burden, not necessarily the highest return.

This means the selected metric directly shapes the type of strategy you end up favoring.

Common metrics and what they emphasize

Sharpe Ratio

Measures return relative to total volatility.

Use this when you want a balance between growth and stability. A high Sharpe ratio usually indicates a smoother equity curve, but it can also favor strategies that trade less or avoid large swings rather than maximizing profit.

Total Return

Measures the portfolio's total percentage gain over the test period.

Use this when raw profit is the priority. Be careful: total return alone says nothing about how risky or unstable the path was.

Calmar Ratio

Measures annualized return relative to maximum drawdown.

Use this when drawdown control matters a lot. This often gives a more practical view than Sharpe for discretionary trading systems because it directly penalizes deep equity declines.

Sortino Ratio

Similar to Sharpe, but penalizes only downside volatility.

Use this when you care more about harmful volatility than upside movement. This can be useful for trend-following or asymmetric systems.

Omega Ratio

Compares gains and losses above a return threshold.

Use this when you want a broader probability-based quality measure rather than a simple mean/volatility ratio.

Max Drawdown

Measures the deepest peak-to-trough decline.

Use this when the main goal is capital preservation or when testing strategies that must remain within strict drawdown constraints.

Win Rate

Measures the percentage of profitable trades.

Use this carefully. High win rate can be misleading if losses are much larger than gains. It should never be used alone.

Choosing the right metric

There is no universally best metric. The best metric depends on the purpose of the test.

A few practical guidelines:

- Use **Sharpe ratio** when comparing general-purpose strategies.
- Use **Calmar ratio** when drawdown matters heavily.
- Use **total return** only when you are also checking drawdown and trade quality separately.
- Avoid relying on **win rate** as the sole optimization target.
- For strategies with few trades, any metric can become unstable, so trade count must also be considered.

Why metric choice can distort optimization

Optimization always pushes the strategy toward whatever the chosen metric rewards. If you select total return, the optimizer may prefer extreme parameter sets that happened to produce one unusually strong run. If you select Sharpe, the optimizer may prefer more stable but lower-return settings. If you select max drawdown, it may favor overly conservative settings that barely trade.

Because of this, the selected metric should match the actual goal of the strategy, not just the most flattering result.

12.2 Direction

The **Direction** setting controls what type of positions the portfolio is allowed to take.

This is one of the most important settings because it changes how strategy signals are interpreted. The same signals can produce very different results depending on whether the backtest is allowed to go long only, short only, or in both directions.

The usual options are:

- **longonly**
- **shortonly**
- **both**

Long Only

In **long-only** mode, the strategy can only take positions that benefit when price rises.

This is the most common choice for equity and crypto swing systems. It is also the safest default when testing strategies that were clearly designed to buy dips, ride trends, or hold bullish breakouts.

Use long-only mode when:

- the strategy logic was designed for bullish participation
- the asset has a long-term upward drift
- short selling is not realistic or not desired
- you want the cleanest and simplest interpretation

Short Only

In **short-only** mode, the strategy can only take positions that benefit when price falls.

This is useful for bear-market systems, hedging ideas, or strategies specifically designed for short setups. It is less common in retail workflows because shorting has practical limits in real markets, but it is still valuable for research.

Use short-only mode when:

- you are testing a bearish signal model
- you want to isolate the quality of short entries
- you are building a hedge or crisis strategy

Both Directions

In **both** mode, the backtest can take both long and short positions.

This is useful when strategy logic is symmetric or when you want to test whether a signal family works on both sides of the market. It can also reveal whether the strategy has a long bias, a short bias, or genuine two-sided edge.

Use both-directions mode when:

- the strategy explicitly defines logic for both sides
- you are researching directional neutrality
- you want to compare long and short behavior inside one framework

Why direction matters so much

Many strategies are not naturally symmetric. A buy-the-dip logic may work well long-only but behave poorly when inverted for shorts. Likewise, a momentum breakout system may work differently on the short side because downtrends are often faster, more violent, and less persistent than uptrends.

If the backtest direction does not match the strategy design, the result can become misleading.

Direction and realism

Even if a strategy performs well in both mode, that does not automatically mean it is practical to trade both sides in the real world. Markets differ in their shorting constraints, borrow costs, liquidity behavior, and overnight gap risk.

So direction should be chosen not just for performance, but also for realism.

12.3 Initial Cash

The **Initial Cash** setting defines the starting portfolio value used at the beginning of the backtest.

This is the baseline from which all returns, compounding behavior, and equity curves begin.

For example, if initial cash is set to **10,000**, the portfolio starts with 10,000 units of the account currency. All trades, profits, losses, and compounding occur from that starting point.

Why initial cash matters

At first glance, initial cash may look cosmetic, but it affects several important things:

- position sizing behavior
- portfolio growth path
- dollar-denominated P&L values
- comparability across tests
- practical realism

Even when percentage returns remain similar, the absolute dollar amounts change with starting capital.

Smaller vs larger starting capital

A smaller starting balance can make the strategy look more volatile in cash terms. A larger starting balance produces larger nominal gains and losses. If the strategy compounds, the difference becomes more pronounced over time.

This matters especially when:

- comparing different strategies
- comparing different assets with different prices
- evaluating practical tradeability
- interpreting trade logs and P&L charts

Choosing a value

Use an initial cash level that reflects the account size you actually care about.

Examples:

- use a modest amount for small-account feasibility
- use a realistic portfolio size for deployment planning
- use a standardized value when comparing many strategies

The most important thing is consistency. If one strategy is tested with 10,000 and another with 100,000, the comparison becomes less clean.

Initial cash and percentage metrics

Metrics such as total return, Sharpe, Sortino, and Calmar are generally scale-aware in a normalized sense, so changing initial cash may not radically alter them. But nominal profit, trade size interpretation, and practical usability can still change meaningfully.

12.4 Fees

The **Fees** setting applies transaction costs to every trade as a proportional trading fee.

This is critical. A strategy that looks excellent before fees can become mediocre or untradeable after realistic costs are included.

For example:

- `0.001` means **0.1%** fee per trade
- `0.002` means **0.2%**
- `0.0005` means **0.05%**

Fees are usually applied every time the portfolio enters or exits a position. Because of this, frequent-trading strategies are affected far more than low-turnover strategies.

Why fees matter

Fees directly reduce profitability. They:

- lower total return
- reduce Sharpe and other performance ratios
- increase the chance that marginal trades become losers
- punish overtrading
- expose unrealistic parameter sets during optimization

A strategy that depends on tiny edges per trade can disappear entirely once realistic fees are applied.

High-turnover vs low-turnover impact

Fees are especially damaging for:

- intraday systems
- breakout systems with many false starts

- strategies with tight exits
- strategies that flip frequently between long and short

Fees matter less for:

- low-frequency trend-following systems
- position strategies with long holding periods
- systems with large average trade expectancy

Choosing a realistic fee value

The correct fee assumption depends on the market and execution venue.

Examples of factors that affect actual fees:

- broker or exchange fee schedule
- maker vs taker execution
- asset class
- leverage product structure
- hidden costs not included in nominal fee rate

The safest approach is to choose a conservative assumption rather than an optimistic one. Underestimating fees is one of the easiest ways to overstate strategy quality.

Fees and optimization

When fees are included during optimization, Vectester automatically penalizes parameter sets that depend on excessive trading. This is desirable. It helps steer optimization toward more robust settings rather than fragile, cost-sensitive ones.

12.5 Slippage

The **Slippage** setting estimates the difference between the expected trade price and the actual execution price.

This cost is separate from fees. Fees are explicit transaction charges. Slippage is the price friction caused by the market itself.

For example:

- `0.0005` means **0.05%** assumed slippage
- `0.001` means **0.1%**

A signal may occur at one price, but in real trading the actual fill may be slightly worse. Slippage models that execution friction.

Why slippage matters

Slippage is often one of the biggest sources of hidden strategy inflation in backtests. Without it, signals are treated as if they execute perfectly at clean market prices. Real markets rarely behave that way.

Slippage is especially important for:

- intraday systems
- low-timeframe strategies
- volatile assets
- breakout entries
- stop-triggered exits
- illiquid markets

How slippage affects results

Slippage reduces profitability by worsening fills:

- entries tend to be less favorable
- exits tend to be less favorable
- tight-edge systems degrade quickly
- stop-heavy systems can look much worse under realistic assumptions

Like fees, slippage hits frequent traders hardest.

Slippage vs fees

These two settings should both be used.

- **Fees** represent explicit trading costs.
- **Slippage** represents imperfect execution.

Ignoring either one usually makes results look cleaner than they really are.

Conservative use of slippage

It is generally better to slightly overestimate slippage than to ignore it. A strategy that survives realistic slippage assumptions is much more trustworthy than one that only works under perfect fills.

12.6 Frequency

The **Frequency** setting tells Vectester how to interpret the spacing of the data in time.

This is not just a label. It affects how annualized statistics are computed, especially metrics such as:

- Sharpe ratio
- Sortino ratio
- Calmar ratio
- volatility-based measures
- annualized return measures

For example, if the data is daily, frequency should usually be set to **1d**. If the data is hourly, it should reflect the actual bar spacing. If the frequency is wrong, the annualized ratios can become misleading even if the raw trades themselves are unchanged.

Why frequency matters

Many performance metrics are scaled based on how often returns are observed. A daily strategy and an hourly strategy cannot be annualized the same way.

If frequency is set incorrectly:

- Sharpe ratio may look too high or too low
- Sortino ratio may be distorted
- annualized return may be misrepresented
- comparisons across tests may become invalid

Matching frequency to data

The frequency should match the actual dataset being tested.

Examples:

- daily bars → 1d
- hourly bars → 1h

- 15-minute bars → appropriate intraday equivalent
- weekly bars → `1wk`

The key principle is consistency between the dataset and the frequency setting.

Frequency and interpretation

Suppose two backtests have identical trade outcomes, but one uses the wrong frequency. The equity curve and total return may look the same, but Sharpe and other annualized metrics may differ. This can cause false conclusions, especially during optimization or strategy comparison.

So frequency is not a cosmetic setting. It directly affects how professional-quality metrics are computed.

12.7 How These Interact with Strategy Code

This is where the backtest settings become most important.

A strategy in Vectester usually defines its own signal logic: when to enter, when to exit, and sometimes additional portfolio behavior such as stop-loss or take-profit rules. But the Backtest page supplies the environment in which those signals are interpreted.

In simple terms:

- the **strategy code** produces instructions
- the **backtest settings** determine how those instructions are evaluated

Both are necessary, and both affect the final result.

Strategy code provides signals

A strategy typically defines:

- entry conditions
- exit conditions
- optional stop-loss and take-profit logic
- optional trailing behavior
- optional direction-related portfolio arguments

These are returned through the strategy result structure.

Backtest settings provide portfolio assumptions

The Backtest page then supplies:

- the ranking metric
- the allowed trade direction
- the starting capital
- transaction fees
- slippage assumptions
- the frequency used for annualized metrics

These settings affect every strategy, including custom strategies.

Interaction example: same strategy, different settings

A trend-following strategy may produce identical entries and exits under two tests, but if one test uses:

- higher fees
- higher slippage
- short-only direction
- incorrect frequency

the final result can change dramatically even though the strategy logic itself did not change.

That is why results should never be interpreted without knowing the backtest settings used.

Direction can conflict with strategy intent

A strategy may be written as a bullish long-only system, but if the backtest direction is changed to something inconsistent, the result may become distorted or less meaningful.

This does not necessarily mean the strategy is broken. It may simply mean the portfolio settings no longer match the logic the strategy was designed for.

Fees and slippage expose fragile code

Custom strategies often look strongest before realistic execution costs are applied. Once fees and slippage are added, weak logic is exposed.

This is healthy. It prevents false confidence and helps reveal whether a strategy has a real edge or only an idealized one.

Frequency affects interpretation, not signal generation

The strategy code does not usually change because of frequency. Signals are still generated from the same data. But the interpretation of the result changes because annualized metrics are scaled using the selected frequency.

So frequency affects the reported quality of the strategy even if it does not change the underlying trades.

Strategy-level portfolio behavior vs global settings

Some strategies may include portfolio-specific behavior such as stop-loss, take-profit, or trailing logic. These are strategy outputs. They work together with the global backtest settings.

This means a complete backtest result is the product of both:

1. strategy-defined behavior
2. user-selected backtest assumptions

The best practice is to think of the backtest as a contract between the two. Strategy code defines *what to do*; backtest settings define *under what conditions it is judged*.

Best practice

When evaluating or writing strategies:

- keep backtest settings realistic
- keep them consistent across comparisons
- do not optimize using one set of assumptions and interpret results under another
- always report the settings alongside the results
- treat costs and frequency as core parts of the test, not optional details

A strategy result is only meaningful in the context of the backtest settings used to produce it.

13. Running a Single Backtest

A **single backtest** is the fastest way to evaluate one strategy with one specific set of parameter values on the currently loaded dataset. It is the standard workflow for checking a strategy's basic behavior before moving on to optimization, walk-forward analysis, or Monte Carlo analysis.

In Vectester, **Run once** does not search for the best settings. It simply takes the selected strategy, reads the values currently shown in the strategy parameter form, combines them with the portfolio settings from the Backtest page, runs the backtest one time, and displays the result.

This mode is especially useful when you want to:

- test a newly written strategy
- verify that a built-in strategy behaves as expected
- check whether a parameter change improves or worsens behavior
- confirm that a strategy produces real trades before attempting optimization
- inspect the equity curve, drawdown, trade log, and metrics for one exact setup

13.1 Run-once workflow

The run-once process follows a fixed sequence inside the application.

Step 1: The application starts the backtest

When you click **Run once**, Vectester changes its status to indicate that a backtest is running and switches the interface into a busy state. This is simply visual feedback that the application is working.

Step 2: The active dataset is obtained

Vectester first determines which market data to use.

- If data has already been loaded in the current session, that dataset is used.
- If data has not yet been loaded into memory, the application attempts to load it using the currently selected data-source settings.

This means a single backtest always runs on the **current active dataset**.

Step 3: The selected strategy is instantiated

Vectester reads the strategy currently chosen in the Strategy page dropdown, loads that strategy class from the registry, and creates a fresh instance of it for the run.

This is important because the backtest is always based on the strategy currently selected at the moment you click **Run once**.

Step 4: Run-once parameter values are collected

The application then reads the values shown in the **run-once parameter form** for that strategy. These values come directly from the strategy's declared parameter definitions.

At this stage, Vectester builds a parameter dictionary that will be passed into the strategy.

Step 5: Portfolio settings are collected

Next, Vectester reads the global portfolio settings from the Backtest page, including items such as:

- direction
- initial cash
- fees
- slippage
- frequency

These are general portfolio-level settings that apply to the run.

Step 6: Strategy logic is built

Vectester calls the strategy's `build()` method using:

- the active market data
- the collected parameter dictionary

The strategy must then return a `StrategyResult` containing:

- `entries`
- optional `exits`
- `pf_kwargs`

This is the point where the strategy transforms its logic into actual trading signals.

Step 7: The portfolio is created

Using the strategy signals and the portfolio settings, Vectester builds the backtest portfolio. This is the actual simulation step where positions, entries, exits, and trading costs are applied over time.

Step 8: The chosen metric is calculated

After the portfolio has been created, Vectester calculates the selected metric, such as Sharpe ratio or total return. That metric is used as a summary value for the run and is also written to the log.

Step 9: The result is checked for validity

Vectester verifies that the run produced a meaningful result. In practice, it rejects runs that have:

- no trades
- a non-finite metric value such as `NaN`, positive infinity, or negative infinity

If the result is valid, it is displayed on the Results page. If not, the run is treated as invalid and an error is reported.

Step 10: The Results page is updated

For a valid run, Vectester stores the resulting portfolio as the current active result and updates the Results page, including:

- summary metric cards
- equity curve
- drawdown chart
- monthly returns
- trade log
- full metrics table

The result title is labeled as:

Run once — strategy name

This distinguishes it from optimization results, which are labeled separately.

13.2 Parameter collection

The quality of a single backtest depends heavily on correct parameter collection. Vectester builds the run-once parameter set directly from the selected strategy's declared parameters.

Where the values come from

Every strategy defines its inputs through its parameter definitions. These definitions control the fields shown in the run-once form. When you press **Run once**, Vectester reads the current value of each field and assembles them into a parameter dictionary.

Each parameter is collected by name, so if a strategy defines parameters such as:

- `fast`
- `slow`
- `sl`
- `tp_ratio`

the run-once parameter dictionary will contain those exact names.

How different parameter types are collected

Integer parameters

Integer parameters are collected from integer spin boxes.

These are used for values such as:

- moving-average windows
- lookback periods
- count-based thresholds
- bar lengths

The collected value is converted to an integer before being passed into the strategy.

Float parameters

Float parameters are collected from decimal spin boxes.

These are commonly used for:

- percentage stops
- ratio thresholds
- volatility multipliers
- indicator thresholds

The collected value is converted to a floating-point number.

Boolean parameters

Boolean parameters are collected from a True/False dropdown control.

The stored result becomes an actual Boolean value:

- `True`
- `False`

These are useful for feature toggles such as enabling or disabling a filter, a trailing stop, or an optional confirmation rule.

Text parameters

Any parameter not declared as integer, float, or Boolean is collected as plain text.

These are less common, but they can be used for categorical or symbolic settings.

Why this matters

This design gives Vectester two important advantages:

1. the strategy itself controls what the user can edit
2. the collected values are already converted into the most appropriate basic type before the strategy receives them

Because of this, the run-once form is not generic decoration. It is a direct user-facing representation of the strategy's declared parameter schema.

What the strategy receives

Once collection is finished, the strategy receives a `params` dictionary containing the current run-once values. The strategy then uses that dictionary inside `build()` to compute signals.

For example, if the form contains:

- `ema_window = 20`
- `sl = 0.02`
- `use_filter = True`

the strategy receives those values as named parameters and uses them to construct its indicators and signal conditions.

13.3 Default filling

Default filling is one of the most important parts of the single-backtest flow because it ensures that every parameter has a usable value even if the user does not manually change it.

Where defaults come from

Each strategy declares a default value for every parameter. These defaults are part of the strategy definition and serve two roles:

- they populate the run-once form when the strategy is selected
- they act as fallback values if a parameter is missing from the supplied run dictionary

In other words, defaults are both **UI defaults** and **execution defaults**.

What happens when a strategy is selected

When you choose a strategy from the dropdown, Vectester rebuilds the parameter form and pre-fills each control with the default value declared by the strategy.

Examples:

- an integer parameter may default to `20`
- a float parameter may default to `0.02`
- a Boolean parameter may default to `True`
- a text parameter may default to a specific string

This means a strategy is immediately runnable after selection, even before the user edits anything.

What happens when the run starts

When **Run once** is executed, Vectester merges the strategy defaults with the collected run parameters.

Conceptually, the rule is:

- start with the strategy defaults
- override them with the values currently collected from the form

This means the final parameter set is complete and explicit.

Why default filling is useful

Default filling prevents several common problems:

- missing parameter values
- partially completed forms
- inconsistent execution when a field has not been manually edited

- fragile strategies that only work when every field is explicitly set by the user

It also makes a newly written strategy much easier to test. As long as the declared defaults are sensible, the strategy can be selected and run immediately.

Best practice for strategy authors

For this reason, strategy authors should choose defaults carefully. Good defaults should be:

- valid
- conservative
- easy to understand
- capable of producing at least some trades on typical data

A bad default may cause the strategy to appear broken even when its underlying logic is correct.

13.4 Validation

After the portfolio is created and the selected metric is calculated, Vectester performs validation to decide whether the run should be accepted as meaningful.

What Vectester validates

In the single-backtest path, Vectester mainly checks two things:

1. whether the selected metric is finite
2. whether the strategy generated at least one trade

These checks are deliberately strict. The application does not treat a mathematically undefined or trade-free result as a successful backtest.

Finite-metric validation

The chosen metric is converted into a single numeric value. Vectester then checks whether that number is finite.

A run is considered invalid if the metric is:

- `NaN`
- positive infinity
- negative infinity

This can happen when a metric becomes undefined because of the structure of the result. For example:

- Sharpe ratio may be undefined in some edge cases
- Calmar ratio may become problematic when the denominator is zero or near zero
- some ratios become unstable when there are too few observations

Trade-count validation

Vectester also checks how many trades the portfolio produced. If the trade count is zero, the run is rejected.

This rule is very important. A strategy that never enters a position has not actually been tested in a meaningful trading sense, even if the rest of the pipeline runs without technical error.

Why Vectester rejects such runs

This validation exists for good reasons:

- a no-trade result tells you nothing about the strategy's practical trading behavior
- an undefined metric cannot be meaningfully compared with other runs
- allowing such results would pollute optimization, comparison, and further analysis

Validation therefore acts as a quality gate.

What happens when validation passes

If the result is valid:

- the portfolio is stored as the current active result
- the Results page is updated
- the metric value is written to the log
- the application status changes to indicate completion

What happens when validation fails

If validation fails, the run raises an error and is not treated as a usable backtest result. The error is written to the log output, and the application status changes to **Error**.

A typical invalid-run message contains both:

- the computed metric value

- the trade count

This is useful because it tells you whether the failure came from an undefined metric, zero trades, or both.

13.5 Interpreting invalid runs

An invalid run does **not** necessarily mean the strategy code is broken. Very often it simply means the selected settings did not produce a meaningful trade sequence on the chosen data.

Understanding this distinction is critical.

Invalid does not always mean "bug"

There are two broad classes of invalid runs.

1. Logical but unusable results

These occur when the strategy runs correctly, but the outcome is not usable for evaluation.

Examples:

- the strategy generates no entries
- the metric is undefined
- the rules are so restrictive that no trade completes

In these cases, the strategy may still be technically correct.

2. True execution errors

These occur when something fails during loading, signal creation, or portfolio construction.

Examples:

- a required parameter is missing
- a strategy returns malformed signals
- a strategy references a column that does not exist
- data is missing or corrupted
- the strategy code raises an exception

In these cases, the strategy or setup usually needs correction.

Common invalid-run patterns

Zero trades

This is the most common invalid single-backtest result.

It usually means one of the following:

- entry conditions are too strict
- exits prevent complete trades from forming
- the selected dataset is not suitable for the strategy
- the parameter values are unrealistic
- the strategy is long-only on a market period with no qualifying long setups

Non-finite metric with very few trades

Sometimes a strategy produces only a tiny number of trades, or returns so unstable that the chosen metric cannot be computed reliably.

This often happens with:

- short datasets
- very narrow market windows
- highly restrictive settings
- metrics that rely on stable return distributions

Error after clicking Run once

If the application reports a full error rather than a simple invalid result, inspect the log carefully. That usually indicates a real execution problem, not merely a poor trading outcome.

How to respond to an invalid run

When a run is invalid, work through the problem methodically.

1. Check whether the dataset is correct.
2. Confirm that the strategy is intended for that market and timeframe.
3. Review the current parameter values.
4. Simplify the strategy if necessary.
5. Try a more forgiving metric such as total return.

6. Verify that the strategy is producing entry signals at all.
7. Confirm that exits and stops are not preventing normal trade completion.

When an invalid run is informative

Even an invalid run can be useful.

For example:

- if a strategy only works under one very narrow condition set, that may indicate fragility
- if a small parameter change causes the strategy to stop trading entirely, that may indicate over-sensitivity
- if a strategy repeatedly fails on multiple datasets, its logic may not generalize

So an invalid run should not simply be ignored. It often reveals something important about strategy robustness.

13.6 Why some strategies generate no trades

One of the most common user questions is: **"Why did the backtest run, but no trades were generated?"**

In most cases, the answer lies in the interaction between strategy logic, parameters, and data.

1. Entry conditions are too strict

A strategy may require several conditions to be true at the same time.

For example:

- trend filter must be bullish
- momentum must exceed a threshold
- volatility must be below a limit
- volume must exceed a confirmation level
- price must pull back by a certain amount

Individually, each condition may be reasonable. Combined, they may become so restrictive that no bar ever satisfies all of them simultaneously.

This is one of the most common reasons a strategy does not trade.

2. Parameter values are unrealistic

Even a sound strategy can stop trading if the chosen parameter values are too extreme.

Examples:

- a lookback window is too long for the available dataset
- an RSI threshold is set so low or high that it is rarely reached
- a volatility filter excludes nearly all bars
- a stop or confirmation rule effectively blocks most entries

When a strategy suddenly goes silent after changing one or two values, unrealistic parameter choices are a likely cause.

3. The dataset is too short

Some strategies need substantial history before their indicators become usable.

For example:

- long moving averages
- rolling regressions
- volatility windows
- regime filters
- multi-stage confirmations

If the dataset is too short, the early section may be dominated by missing indicator values, and the remaining usable region may be too small to produce trades.

4. The market regime does not suit the strategy

Strategies are usually designed for specific market behaviors.

Examples:

- trend-following strategies may struggle in flat or choppy markets
- mean-reversion strategies may struggle during persistent breakouts
- breakout strategies may remain inactive during quiet consolidation
- long-only momentum strategies may fail to trigger in weak downward markets

So no-trade behavior may simply mean the current market does not match the strategy's intended environment.

5. Long-only or short-only restrictions

If the strategy logic or the backtest direction limits the system to one side of the market, trades may disappear when the market does not present that type of opportunity.

For example:

- a long-only strategy on a bearish sample may produce nothing useful
- a short-only configuration on a strong uptrend may remain mostly inactive

Direction settings always need to be interpreted together with market context.

6. Exit structure prevents meaningful trade formation

Sometimes entries are present, but the way exits or stop rules are defined prevents usable trades from appearing in the final result.

Possible causes include:

- contradictory exit logic
- immediate exit conditions
- stop settings that close positions too aggressively
- logic that creates unstable entry/exit overlap

This is especially relevant for custom strategies.

7. Indicator warm-up removes too much of the sample

Strategies using rolling indicators often need a warm-up period before the first valid values appear.

If multiple indicators are combined, the effective warm-up can become much longer than expected. On short datasets, this can leave too little usable data for the strategy to trade.

8. Data frequency does not match the strategy design

A strategy written for daily bars may not behave sensibly on intraday data, and an intraday strategy may become ineffective on daily bars.

The same logic may therefore produce:

- many trades on one timeframe
- no trades at all on another

Always make sure the timeframe fits the strategy's intended use.

9. The strategy defaults are too conservative

If a strategy author chooses very cautious defaults, the strategy may look inactive when first selected.

This is not necessarily wrong. Conservative defaults can be safer than aggressive ones. However, they may require adjustment before the strategy becomes practically tradable on a given instrument.

10. The strategy logic is genuinely flawed

Finally, no-trade behavior can also indicate a real design problem.

Examples include:

- entry conditions that can never be satisfied together
- incorrect comparison directions
- a parameter used in the wrong scale
- conditions applied to the wrong series
- logic that unintentionally filters out all entries

When a strategy produces no trades across many instruments, timeframes, and reasonable parameter sets, the logic itself should be inspected.

Practical guidance

When a single backtest produces no trades or an invalid result, the best approach is to simplify and isolate.

Start with this sequence:

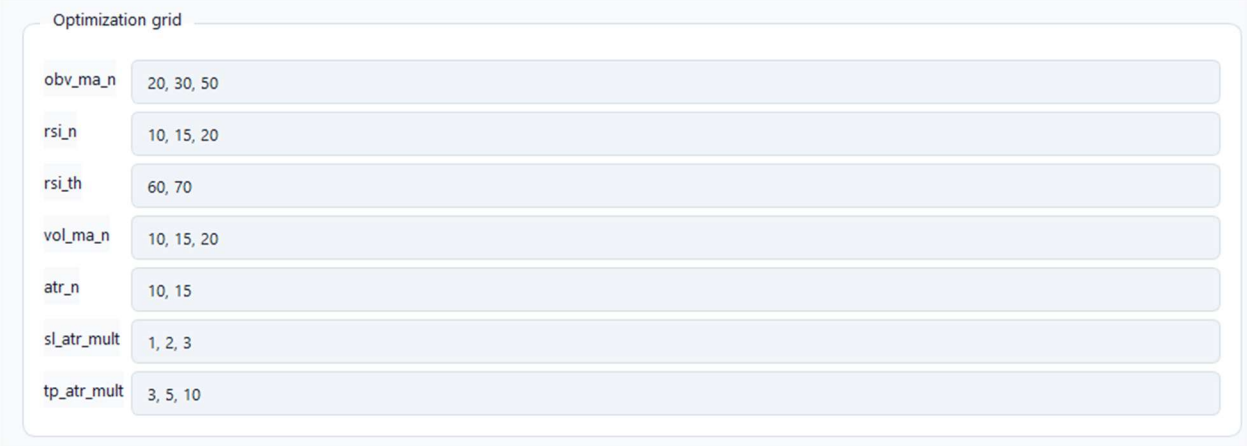
1. use a longer and cleaner dataset
2. test the strategy with its declared defaults
3. reduce the number of filters
4. widen thresholds
5. shorten long lookback windows
6. confirm the strategy is being run on the intended timeframe
7. inspect whether the logic should naturally trade in the current market regime

A single backtest is not just a result screen. It is also a diagnostic tool. If you use it carefully, it will tell you whether a strategy is ready for deeper research or still needs design work.

14. Optimization

Optimization in Vectester tests many parameter combinations for the currently selected strategy and finds the combination that scores best on the metric you choose on the **Backtest** page. It is designed to help you answer a practical question:

> "Which parameter values work best for this strategy on this dataset, under these trading assumptions?"



Optimization is powerful, but it is also easy to misuse. A high optimization result does **not** automatically mean a strategy is good. It only means that, among the parameter combinations tested, one combination scored best on the chosen metric for the selected historical data.

14.1 How optimization works

When you click **Optimize**, Vectester performs these steps:

1. It loads the current dataset.
2. It reads the current strategy.
3. It reads the optimization grid for each declared strategy parameter.
4. It builds every possible parameter combination from those grid values.
5. For each combination, it runs the strategy and generates entry and exit signals.
6. It applies the current portfolio settings such as direction, initial cash, fees, slippage, and frequency.
7. It calculates the selected performance metric for each run.
8. It filters out invalid runs.
9. It ranks the valid runs.

10. It selects the best-performing combination and displays that result in the **Results** page.

So optimization is an **exhaustive grid search**, not a random search and not a machine-learning optimizer. If you enter 3 values for one parameter and 4 values for another, Vectester tests all 12 combinations.

This matters because runtime grows very quickly as you add more parameters and more values per parameter.

14.2 How parameter grids are entered

On the **Strategy** page, every strategy parameter appears twice:

- once in the **run-once** parameter form
- once in the **Optimization grid** form

The **run-once** field is for a single backtest using one value.

The **Optimization grid** field is for one or more candidate values. Vectester reads those values and uses them to build the search space.

Each parameter has its own input line. You enter candidate values directly into that line.

Examples:

```
ema_window: 10, 20, 50
rsi_entry: 25, 30, 35
sl: 0.01, 0.02, 0.03
use_filter: true, false
```

The parameter type is determined by the strategy definition. Vectester interprets the text according to that type:

- integer parameters become integers
- decimal parameters become floating-point values
- boolean parameters become `True` or `False`
- text parameters remain text

The grid is always built from the strategy's declared parameter list. If a parameter is not declared by the strategy, it is not part of the optimization process.

14.3 Comma-separated values

Grid values are entered as **comma-separated lists**.

Example:

```
20, 50, 100
```

This means Vectester will test three candidate values for that parameter.

Whitespace does not matter. These are treated the same:

```
20,50,100
20, 50, 100
 20 , 50 , 100
```

For boolean parameters, Vectester accepts values such as:

```
true, false
yes, no
1, 0
```

In practice, it is best to use:

```
true, false
```

because that is the clearest and least ambiguous format.

For decimal parameters, enter standard decimal notation:

```
0.01, 0.02, 0.03
```

For text parameters, use plain text values separated by commas.

Important behavior:

- a single value is still a valid grid
- if every parameter has only one value, optimization still runs, but it is effectively testing only one combination
- empty values are ignored only if there is still at least one valid value left after parsing

Examples:

```
50
```

means one candidate value.

```
10, 20, 30
```

means three candidate values.

```
10, , 20
```

is read as:

```
10, 20
```

14.4 Parameter combination expansion

Once all grid fields are read, Vectester generates the full set of combinations by taking the **Cartesian product** of all parameter lists.

This means every value of parameter A is combined with every value of parameter B, and so on.

Example:

```
fast = 10, 20
slow = 50, 100
sl = 0.01, 0.02
```

This produces these 8 combinations:

```
fast=10, slow=50,  sl=0.01
fast=10, slow=50,  sl=0.02
fast=10, slow=100, sl=0.01
fast=10, slow=100, sl=0.02
fast=20, slow=50,  sl=0.01
fast=20, slow=50,  sl=0.02
fast=20, slow=100, sl=0.01
fast=20, slow=100, sl=0.02
```

The number of combinations is the product of the number of values in each parameter field.

So:

- 3 parameters with 3 values each = 27 combinations
- 5 parameters with 4 values each = 1,024 combinations
- 8 parameters with 5 values each = 390,625 combinations

This is why optimization can become slow very quickly.

A good rule is to calculate the approximate grid size before running it.

14.5 Metric ranking

After each parameter combination is tested, Vectester evaluates the run using the metric selected on the **Backtest** page.

Typical metrics include things like:

- Sharpe ratio
- total return
- Calmar ratio

- Sortino ratio
- win rate
- max drawdown

The selected metric becomes the optimization objective.

In other words, Vectester asks:

> "Which valid parameter combination gives the highest value for this metric?"

For most metrics, higher is better. So the optimizer ranks combinations from highest to lowest and picks the top one.

This has important consequences:

Choosing Sharpe ratio

Optimization will favor smoother risk-adjusted performance, not necessarily the highest raw return.

Choosing total return

Optimization will favor absolute growth, even if drawdowns are large.

Choosing Calmar ratio

Optimization will favor return relative to drawdown.

Choosing win rate

Optimization may prefer many small wins even if total profitability is weak.

So the "best" parameter set depends entirely on the metric you choose. There is no universally best parameter set independent of objective.

Best practice is to optimize on one metric, then inspect the full result carefully in the **Results**, **Walk-Forward**, and **Monte Carlo** pages before trusting it.

14.6 Invalid parameter sets

Not every parameter combination produces a usable result.

Vectester treats a parameter set as invalid if:

- it produces **zero trades**
- the selected metric is **not finite**

A non-finite metric usually means the result is mathematically unusable for ranking. For example, some risk metrics can become undefined when returns are too sparse or volatility calculations break down.

Examples of invalid combinations:

- an entry threshold so strict that no signals ever occur
- parameter relationships that prevent entries and exits from functioning properly
- a stop or filter setup that blocks all trades
- extreme settings that produce undefined statistics

Invalid combinations are automatically removed from the ranking.

If all combinations are invalid, optimization fails and no best result is produced.

This is an important diagnostic signal. It usually means one of these is true:

- the strategy logic is too restrictive
- the grid values are unrealistic
- the dataset is too short or unsuitable
- the chosen metric is unstable for the generated trades

When this happens, the fix is usually to simplify or widen the search space rather than immediately assuming the strategy is broken.

14.7 Best-parameter selection

After invalid runs are removed, Vectester selects the parameter combination with the highest value for the chosen optimization metric.

That combination becomes the **best parameter set**.

The app then:

- stores the full optimization result internally
- extracts the portfolio corresponding to the best parameter combination
- displays that best run on the **Results** page
- reports the best parameter values in the output log

This is important to understand:

The **Results** page after optimization shows only the **best run**, not the entire grid table.

So when you see the post-optimization charts and metrics, you are looking at the single top-ranked parameter combination, not an average of all tested combinations and not the full optimization landscape.

This means you should treat the result as a candidate, not as proof.

A responsible workflow is:

1. optimize
2. inspect the best result
3. check whether neighboring values also work
4. run walk-forward analysis
5. run Monte Carlo analysis
6. compare against simpler alternatives

If the top result is dramatically better than nearby parameter values, that is often a warning sign of overfitting.

14.8 Optimization pitfalls

Optimization is useful, but there are several common mistakes.

Pitfall 1: Using too many parameters

The more parameters you optimize, the easier it becomes to fit noise rather than real market structure.

A strategy with many knobs can almost always be made to look good in-sample.

Pitfall 2: Using too many values per parameter

Very dense grids massively increase search size and raise the probability of chance fits.

Testing 20 values for 6 parameters is usually excessive for normal research workflows.

Pitfall 3: Optimizing on the wrong metric

A strategy optimized for total return may end up with unacceptable drawdown.
A strategy optimized for win rate may have poor expectancy.
A strategy optimized for Sharpe may trade too little to be meaningful.

Pitfall 4: Ignoring trade count

A parameter set can look outstanding with very few trades. That may not be robust.

Always check:

- number of trades
- distribution of winners and losers
- consistency through time

Pitfall 5: Treating the best result as truth

The top-ranked result is simply the winner of the tested grid on the chosen data. It may not survive new data.

Pitfall 6: Testing unrealistic ranges

If your ranges include impossible or unreasonable values, optimization spends time on useless combinations and may distort interpretation.

Pitfall 7: Building incompatible parameter combinations

Some parameters should logically constrain others.

For example:

- a “fast” moving average should generally stay below a “slow” moving average
- a very tight stop with a very wide trend filter may produce meaningless combinations
- some filters only make sense when another boolean option is enabled

If your grid ignores these relationships, many combinations may be invalid or misleading.

Pitfall 8: Re-optimizing repeatedly until something looks good

This is one of the fastest ways to overfit. Repeated search over many ranges, metrics, and datasets can manufacture attractive results that do not generalize.

Pitfall 9: Ignoring validation

Optimization should never be the end of the process. In Vectester, the next steps should usually be:

- **Walk-Forward**
- **Monte Carlo**
- **Compare**

14.9 Designing good search spaces for strategies you write

If you create your own strategies, the quality of the search space matters almost as much as the quality of the strategy logic.

A good search space is:

- small enough to run efficiently
- broad enough to test the main idea
- structured around plausible values
- easy to interpret
- resistant to overfitting

Start with the strategy idea, not the grid

Do not begin by throwing large ranges at every parameter.

First ask:

- What is this strategy trying to detect?
- Which parameters truly control that behavior?
- Which parameters are secondary?

Only optimize parameters that matter.

Prefer a few meaningful values

For many parameters, 3 to 5 carefully chosen values are better than 20 arbitrary values.

Example:

```
ema_window: 20, 50, 100
```

is often better than:

```
ema_window: 17, 18, 19, 20, 21, 22, 23, 24
```

The first tests distinct regimes.
The second over-samples a narrow neighborhood and increases the chance of fitting noise.

Use sensible spacing

For lookback windows, it is usually better to test values that reflect meaningfully different time horizons.

Examples:

- short / medium / long
- fast / normal / slow

- low / medium / high sensitivity

Good:

```
10, 20, 50
```

Less useful:

```
10, 11, 12, 13, 14
```

unless there is a specific reason to study local sensitivity.

Keep related parameters logically consistent

If your strategy uses paired parameters, define grids that make sense together.

For example, in a fast/slow moving-average strategy, you usually want:

- fast < slow

If your grid allows the reverse, you create combinations that are technically testable but conceptually weak.

Optimize only a few parameters at first

For a new strategy, begin with the most important 1 to 3 parameters.

After you understand those, add others only if needed.

This gives you:

- faster runs
- easier interpretation
- less overfitting risk

Fix some values deliberately

Not every parameter needs to be optimized.

Sometimes it is better to hold one parameter constant and optimize the others. This makes the result easier to understand.

Use broad-to-narrow refinement

A good workflow is:

1. test a coarse grid
2. identify the promising region
3. refine modestly around that region

4. stop before the search becomes hyper-specific

This is much better than starting with a huge fine-grained grid.

Look for stability, not just peaks

A robust strategy usually has a region of good performance, not a single magical point.

After optimization, ask:

- Do nearby values also perform reasonably well?
- Or does performance collapse immediately outside the winner?

Stable plateaus are more trustworthy than sharp spikes.

Match the grid to the dataset

Short datasets cannot support extremely rich optimization.
Sparse-trading strategies also need extra caution, because small changes may produce unstable metrics.

The less data you have, the simpler the grid should be.

Include practical constraints

When designing strategies for real use, choose ranges that are operationally sensible.

Examples:

- stop-loss values that are not absurdly tight or wide
- momentum thresholds that produce enough trades
- lookback windows that make sense for the timeframe being tested

Validate every "good" result

A good search space does not remove the need for validation.

After finding promising parameters, always check:

- trade count
- drawdown
- consistency through time
- walk-forward efficiency
- Monte Carlo robustness

- comparison against simpler strategies

Practical optimization workflow

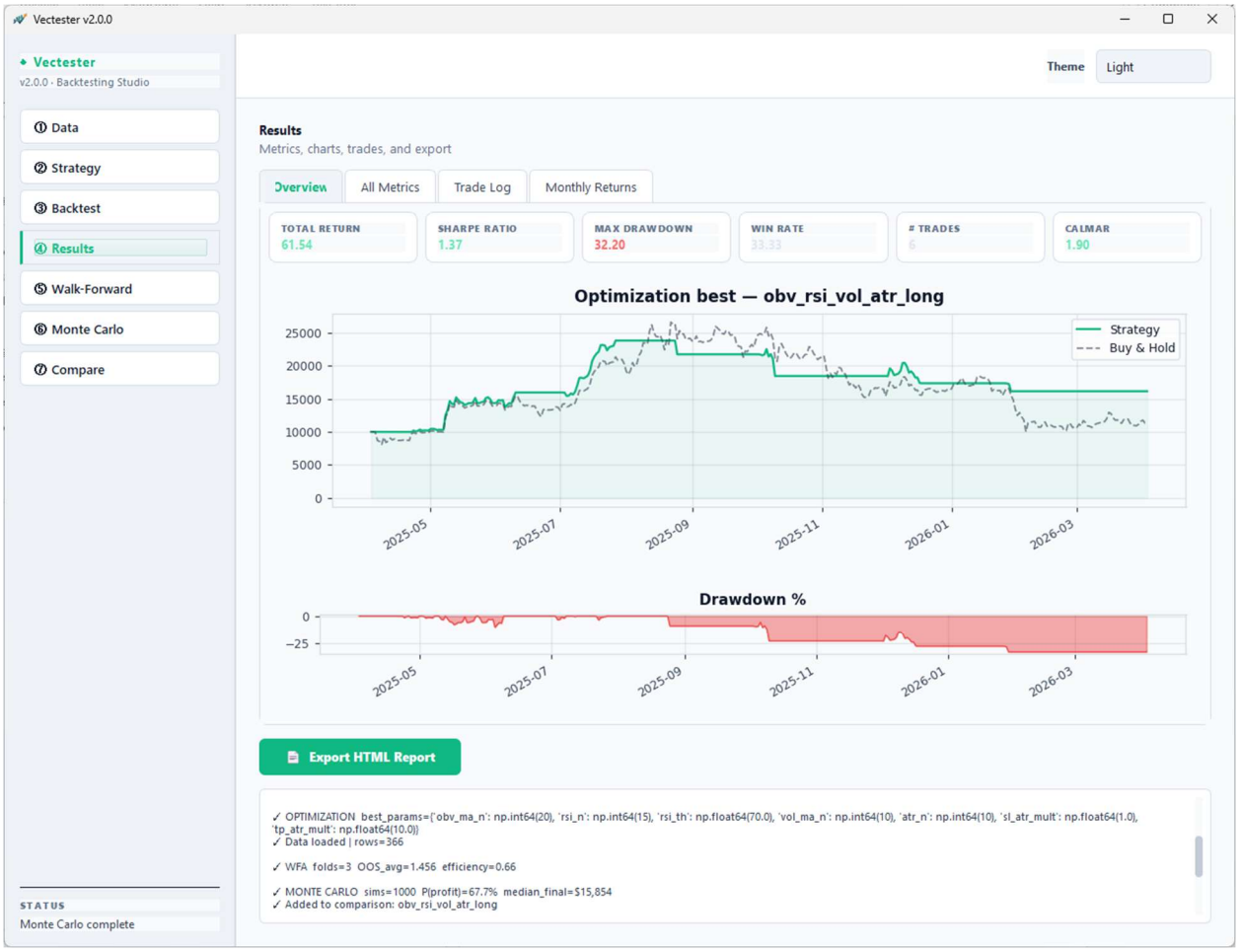
A disciplined workflow for Vectester is:

1. Run the strategy once with default parameters.
2. Confirm that the logic works and produces sensible trades.
3. Optimize only the most important parameters.
4. Keep the first grid small.
5. Review the best result carefully.
6. Check whether nearby values also hold up.
7. Run walk-forward analysis.
8. Run Monte Carlo analysis.
9. Compare with simpler or benchmark strategies.
10. Only then treat the strategy as potentially robust.

Optimization is most useful when it helps you understand a strategy better. It becomes dangerous when it is used only to search for the prettiest historical result.

15. Reading Results

The **Results** page is where Vectester turns a backtest into something you can interpret, question, and compare. A strategy is not judged by one number alone. It is judged by the combined story told by its summary cards, equity behavior, drawdowns, trade distribution, monthly consistency, and the realism of the assumptions behind it.



In Vectester, the Results page is organized into four main views:

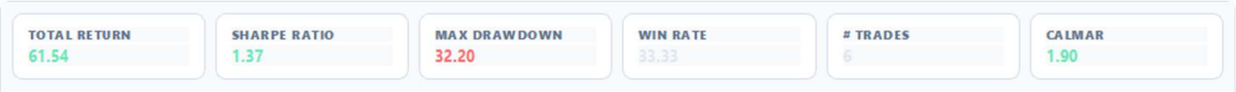
- **Overview**
- **All Metrics**
- **Trade Log**
- **Monthly Returns**

The Overview view also contains the most important visual summary of the strategy: the **summary cards**, the **equity curve**, and the **drawdown chart**.

A good result is not simply “high return.” A good result is a result that is profitable, understandable, repeatable, and robust enough to survive realistic market conditions.

15.1 Summary Cards

The summary cards provide a fast first impression of the strategy. They are useful because they compress the most important performance information into a small set of headline figures.



Vectester shows the following cards:

- **Total Return**
- **Sharpe Ratio**
- **Max Drawdown**
- **Win Rate**
- **# Trades**
- **Calmar**

These cards are useful for quick triage. They help answer the first question every researcher asks:

> Is this result worth deeper inspection, or should I reject it immediately?

Total Return

Total Return shows the net percentage change in portfolio value over the tested period.

It answers:

- Did the strategy make money overall?
- How much did it grow or lose from start to finish?

A high Total Return is attractive, but by itself it proves very little. A strategy can show a high return while also being dangerously unstable, heavily overfit, or exposed to unacceptable drawdowns.

You should always read Total Return together with:

- Max Drawdown
- Sharpe Ratio

- Number of trades
- Equity curve shape

A return figure becomes much more meaningful when you know **how** that return was achieved.

Sharpe Ratio

Sharpe Ratio measures return relative to volatility. In simple terms, it tells you how much reward the strategy generated for the amount of variability it took on.

A higher Sharpe Ratio usually suggests cleaner, more efficient performance. Two strategies may have similar returns, but the one with the higher Sharpe Ratio usually gets there with less noise and less erratic behavior.

Be careful with very high Sharpe values. They can sometimes result from:

- too few trades
- too short a sample period
- very smooth but unrealistic assumptions
- over-optimization

Sharpe is useful, but never final.

Max Drawdown

Max Drawdown shows the largest peak-to-trough decline experienced during the test.

This is one of the most important risk measures in the entire application.

It answers:

- How painful was the worst loss period?
- How much capital did the strategy give back before recovering?
- Could a real user have tolerated this behavior psychologically and financially?

A strategy with high returns but severe drawdown may be much worse than a lower-return strategy with controlled risk.

Drawdown often matters more than users expect, because many strategies fail not by losing forever, but by falling so deeply that they become impossible to hold in real life.

Win Rate

Win Rate is the percentage of trades that closed profitably.

This number is often misunderstood. A high Win Rate does **not** automatically mean a good strategy. Many weak strategies win often but suffer occasional large losses that erase many small gains.

Likewise, a lower Win Rate does **not** automatically mean a bad strategy. Trend-following systems often win less often, but their winners are much larger than their losers.

Always read Win Rate together with:

- average win size
- average loss size
- expectancy
- profit factor
- trade count

Win Rate tells you frequency of success, not quality of success.

Number of Trades

 # Trades shows how many completed trades the strategy produced.

This is essential context for every other metric.

A very strong-looking result with only a handful of trades is not nearly as convincing as a similar result built on a large sample. Small samples can easily produce misleading results by luck alone.

Use trade count to judge statistical credibility:

- **Very low trade count**: weak evidence
- **Moderate trade count**: more informative
- **High trade count**: stronger basis for evaluation, assuming data quality is good

There is no universal "correct" number of trades, because it depends on timeframe, market, and strategy type. But in general, more observations make the result more trustworthy.

Calmar Ratio

Calmar Ratio compares return to maximum drawdown. It asks a practical question:

> How much return did the strategy earn for the worst pain it caused?

This makes it extremely useful for comparing aggressive and conservative strategies. A strategy with a lower headline return may still be superior if it uses risk more efficiently.

Calmar is particularly useful when evaluating strategies that differ sharply in drawdown behavior.

How to Use the Summary Cards Properly

Treat the cards as a screening layer, not as a final verdict.

A sensible first pass is:

1. Check whether Total Return is acceptable.
2. Check whether Max Drawdown is survivable.
3. Check whether Sharpe and Calmar are reasonably strong.
4. Check whether the trade count is large enough to mean anything.
5. Only then move on to the charts and deeper metrics.

A strong summary-card profile does not prove robustness, but a weak one often saves you from wasting time on a bad strategy.

15.2 Equity Curve

The **equity curve** shows how portfolio value changes over time. It is one of the most revealing parts of the Results page because it shows the behavior of the strategy, not just the final score.

Vectester plots:

- the **strategy equity**
- a **buy-and-hold benchmark**
- the portfolio value path across the full backtest period

This chart answers questions such as:

- Did the strategy grow steadily or erratically?

- Did performance come from one lucky period or from repeated success?
- Did the strategy outperform passive holding?
- Were gains achieved early, late, or consistently throughout the sample?

What a Good Equity Curve Looks Like

A healthy equity curve usually has these characteristics:

- a generally upward long-term slope
- controlled setbacks rather than catastrophic collapses
- no extreme dependence on one isolated burst of performance
- reasonable behavior across different market phases

It does **not** need to be perfectly smooth. Real strategies never are. Some volatility is normal. What matters is whether the curve behaves in a way that is understandable and believable.

What to Watch For

1. Late-stage explosion

If most of the profit appears in one short period near the end of the test, the result may be fragile. The strategy may have benefited from a narrow market regime rather than a broad underlying edge.

2. Long flat periods

A strategy that spends most of its life doing nothing may still be valid, but you should understand why. Perhaps it is highly selective. Perhaps it is under-trading. Or perhaps the logic rarely triggers because the parameters are too restrictive.

3. Violent jumps

Sudden large steps up or down may indicate:

- concentrated risk
- oversized winners or losers
- dependence on unusual market events
- potentially unrealistic assumptions

4. Strong underperformance versus benchmark

If the strategy cannot beat buy-and-hold on the same instrument over the same period, it needs a very strong reason to exist. Sometimes lower return is acceptable if the strategy gives meaningfully lower drawdown or better consistency, but that trade-off must be clear.

Strategy Equity vs Buy-and-Hold

The benchmark matters because it answers:

> Was the strategy actually useful, or would passive exposure have done better?

A custom strategy should justify its complexity. If buy-and-hold delivers similar or better return with less effort and fewer moving parts, then the custom strategy may not offer enough value.

However, outperforming buy-and-hold is not the only valid goal. A strategy may still be useful if it offers:

- lower drawdown
- smoother behavior
- better capital efficiency
- stronger out-of-sample stability
- more acceptable psychological risk

How to Read Shape, Not Just Endpoint

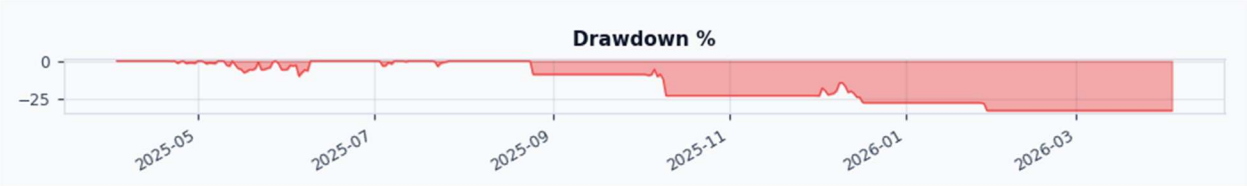
Two strategies can finish at the same final value but be completely different in quality.

One may rise steadily with manageable pullbacks. Another may suffer huge collapses, recover late, and look acceptable only because it happened to end strong.

That is why the **path** matters. The equity curve helps you judge whether the strategy's success is stable, chaotic, concentrated, or fragile.

15.3 Drawdown Chart

The **drawdown chart** shows the percentage decline from the portfolio's previous peak at every point in time.



If the equity curve tells you how the strategy grows, the drawdown chart tells you how painful that growth is.

This view is critical because many strategies look acceptable until you see what had to be endured to achieve the final result.

What Drawdown Means

A drawdown begins when the portfolio falls below its previous high. It ends only when a new high is reached.

The drawdown chart therefore reveals:

- depth of losses
- duration of painful periods
- speed of recovery
- clustering of stress events

Why It Matters So Much

Users often focus too much on return and not enough on drawdown. In practice, drawdown is one of the main reasons strategies fail in real usage.

A system with strong returns but repeated deep drawdowns may be:

- too stressful to follow
- too risky to size properly
- too unstable for real capital
- vulnerable to market regime changes

Drawdown is not just a technical statistic. It is a practical survivability measure.

How to Interpret the Drawdown Chart

Shallow and brief drawdowns

Usually a positive sign. They suggest the strategy recovers efficiently and avoids prolonged damage.

Deep but rare drawdowns

Potentially acceptable, but only if they are understood and intentionally tolerated. Such strategies may still work, but they require stronger discipline and careful sizing.

Frequent repeated drawdowns

This may indicate unstable signal quality, poor exits, or a strategy that is constantly being whipsawed.

Long recovery periods

Even if the drawdown depth is moderate, very long recoveries can be a major weakness. Time trapped below the previous equity high is an important psychological and capital-efficiency cost.

Max Drawdown vs Drawdown Behavior

The summary card gives you **one number**: the worst drawdown.

The chart gives you the **full pattern**:

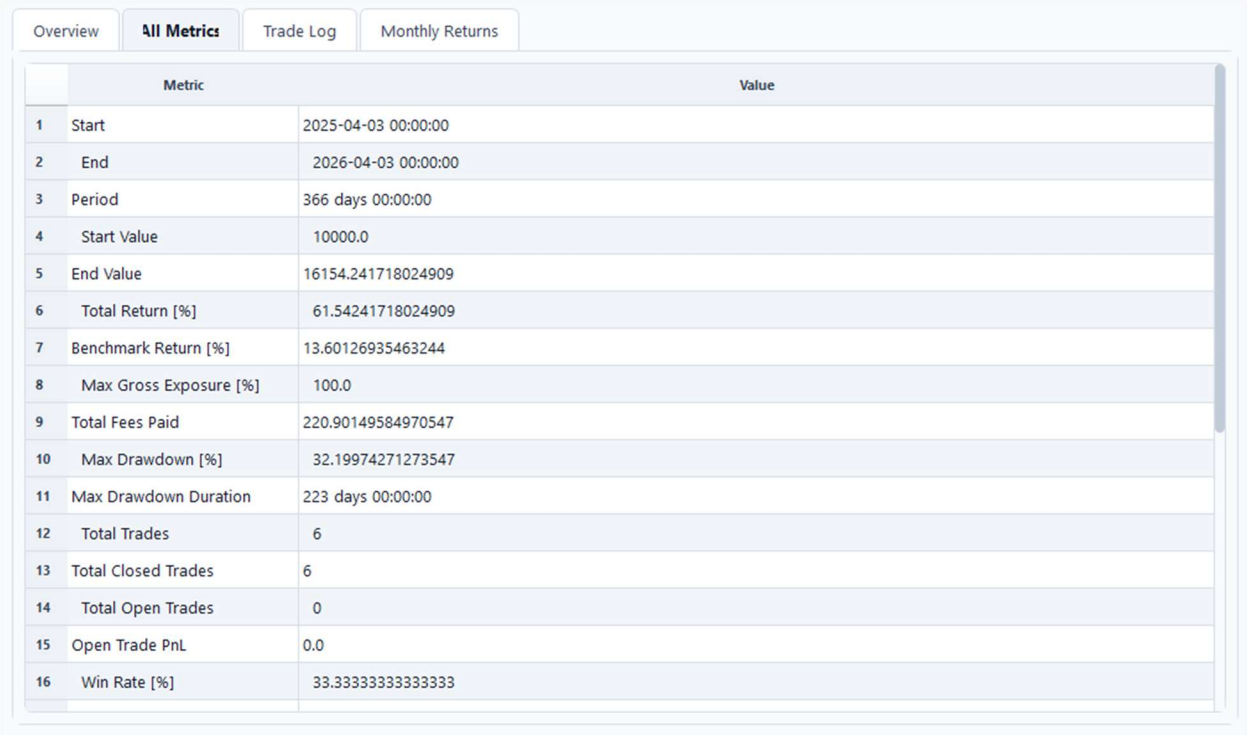
- was the worst drawdown a single event?
- were there many near-worst events?
- does the strategy repeatedly break down?
- are recoveries fast or painfully slow?

The chart is often more informative than the single Max Drawdown number.

15.4 Full Metrics Table

The **Full Metrics Table** gives the complete statistical view of the backtest. This is where you move beyond the headline numbers and examine the structure of the result in depth.

This table is important because good strategies are multidimensional. A custom strategy should not be evaluated by one ratio or one return figure alone.



Overview | All Metrics | Trade Log | Monthly Returns

	Metric	Value
1	Start	2025-04-03 00:00:00
2	End	2026-04-03 00:00:00
3	Period	366 days 00:00:00
4	Start Value	10000.0
5	End Value	16154.241718024909
6	Total Return [%]	61.54241718024909
7	Benchmark Return [%]	13.60126935463244
8	Max Gross Exposure [%]	100.0
9	Total Fees Paid	220.90149584970547
10	Max Drawdown [%]	32.19974271273547
11	Max Drawdown Duration	223 days 00:00:00
12	Total Trades	6
13	Total Closed Trades	6
14	Total Open Trades	0
15	Open Trade PnL	0.0
16	Win Rate [%]	33.33333333333333

Depending on the backtest, the table can include metrics such as:

- total return
- benchmark return
- annualized return
- volatility
- Sharpe Ratio
- Sortino Ratio
- Calmar Ratio
- Omega Ratio
- Max Drawdown
- exposure
- win rate
- average winning trade
- average losing trade

- expectancy
- profit factor
- trade count
- best trade
- worst trade
- average trade duration

The exact contents depend on the portfolio statistics generated by the engine.

How to Read the Table

A good way to read the table is by category.

Return metrics

These tell you how much the strategy made.

Examples:

- Total Return
- Annualized Return
- Benchmark Return

These are useful, but they should be tested against risk and consistency.

Risk-adjusted metrics

These tell you how efficiently the strategy earned its return.

Examples:

- Sharpe Ratio
- Sortino Ratio
- Calmar Ratio
- Omega Ratio

These are often more meaningful than raw return, especially when comparing strategies with very different risk profiles.

Risk and damage metrics

These show what the strategy suffered.

Examples:

- Max Drawdown
- Volatility
- Worst Trade
- Average Losing Trade

These numbers help you judge whether the return came at an acceptable cost.

Trade quality metrics

These describe the internal quality of the trade distribution.

Examples:

- Win Rate
- Profit Factor
- Expectancy
- Average Win
- Average Loss

These are especially useful when evaluating custom strategies, because they reveal whether the strategy has a believable edge or is being held up by a few lucky outcomes.

Activity metrics

These show how active or inactive the strategy is.

Examples:

- Total Trades
- Exposure
- Trade Duration

These metrics help explain whether the system is:

- selective
- hyperactive
- underused

- constantly in the market

Important Metrics to Understand Well

Profit Factor

Profit Factor compares gross profits to gross losses.

- Above 1.0 means profits exceed losses
- Higher values are generally better
- Extremely high values can be suspicious if the sample is small

A good Profit Factor supports the idea that the strategy has real economic edge.

Expectancy

Expectancy estimates the average value generated per trade.

This is one of the most useful concepts in evaluating a strategy honestly because it combines:

- win rate
- win size
- loss size

A strategy with positive expectancy has a better claim to being genuinely useful than one that simply looks good on one test.

Exposure

Exposure tells you how much of the time the strategy is actually invested.

This matters because a strategy that earns strong returns while being in the market only part of the time can be very efficient. On the other hand, a strategy with weak return despite constant exposure may not justify its risk.

How to Use the Full Metrics Table Properly

Use the table to ask:

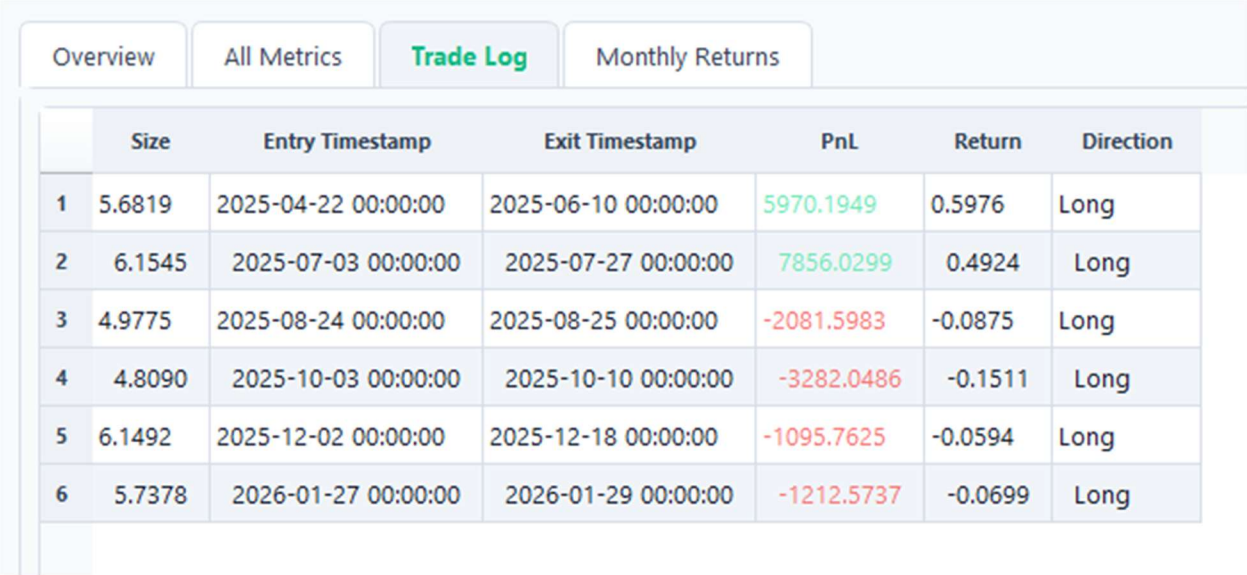
- Are returns high enough?
- Is the risk acceptable?
- Are the trades internally healthy?
- Is the sample size convincing?

- Does the strategy beat the benchmark for a good reason?
- Are there hidden weaknesses behind the headline cards?

The full metrics table is where many “good-looking” strategies begin to fall apart under scrutiny.

15.5 Trade Log

The **Trade Log** displays the actual completed trades produced by the strategy. This is where abstract metrics become real trading behavior.



	Size	Entry Timestamp	Exit Timestamp	PnL	Return	Direction
1	5.6819	2025-04-22 00:00:00	2025-06-10 00:00:00	5970.1949	0.5976	Long
2	6.1545	2025-07-03 00:00:00	2025-07-27 00:00:00	7856.0299	0.4924	Long
3	4.9775	2025-08-24 00:00:00	2025-08-25 00:00:00	-2081.5983	-0.0875	Long
4	4.8090	2025-10-03 00:00:00	2025-10-10 00:00:00	-3282.0486	-0.1511	Long
5	6.1492	2025-12-02 00:00:00	2025-12-18 00:00:00	-1095.7625	-0.0594	Long
6	5.7378	2026-01-27 00:00:00	2026-01-29 00:00:00	-1212.5737	-0.0699	Long

Vectester shows key trade fields such as:

- entry timestamp
- exit timestamp
- direction
- size
- entry price
- exit price
- PnL
- return

This is one of the most important diagnostic views for custom strategy development.

Why the Trade Log Matters

A strategy may look strong at the aggregate level but reveal major problems when you inspect individual trades.

The trade log helps you verify:

- whether entries happen where you expect
- whether exits behave logically
- whether stop-loss and take-profit logic work correctly
- whether trade durations make sense
- whether position sizing looks reasonable
- whether the strategy is dominated by a few exceptional trades

What to Look For

Entry quality

Are trades opening in the kinds of conditions the strategy is supposed to detect?

If not, the signal logic may be too loose, too strict, or simply incorrect.

Exit quality

Are exits happening for understandable reasons?

If exits consistently arrive too late, too early, or at strange points, the strategy may need better management logic.

Trade clustering

Do many trades occur in very similar conditions, or in a short burst?

That may be normal, but it may also reveal over-sensitivity or poor filtering.

Outlier trades

A few giant winners or giant losers can dominate the result.

This is not automatically bad, but it must be understood. If the entire strategy depends on one or two exceptional trades, the result is much less robust than it first appears.

Holding period

Do trades last as long as the logic suggests they should?

An intended swing strategy that exits almost immediately may have a design problem. A supposedly short-term strategy with very long holding times may not be behaving as intended either.

Trade Log as a Debugging Tool

For custom strategies, the trade log is not just a reporting feature. It is a debugging tool.

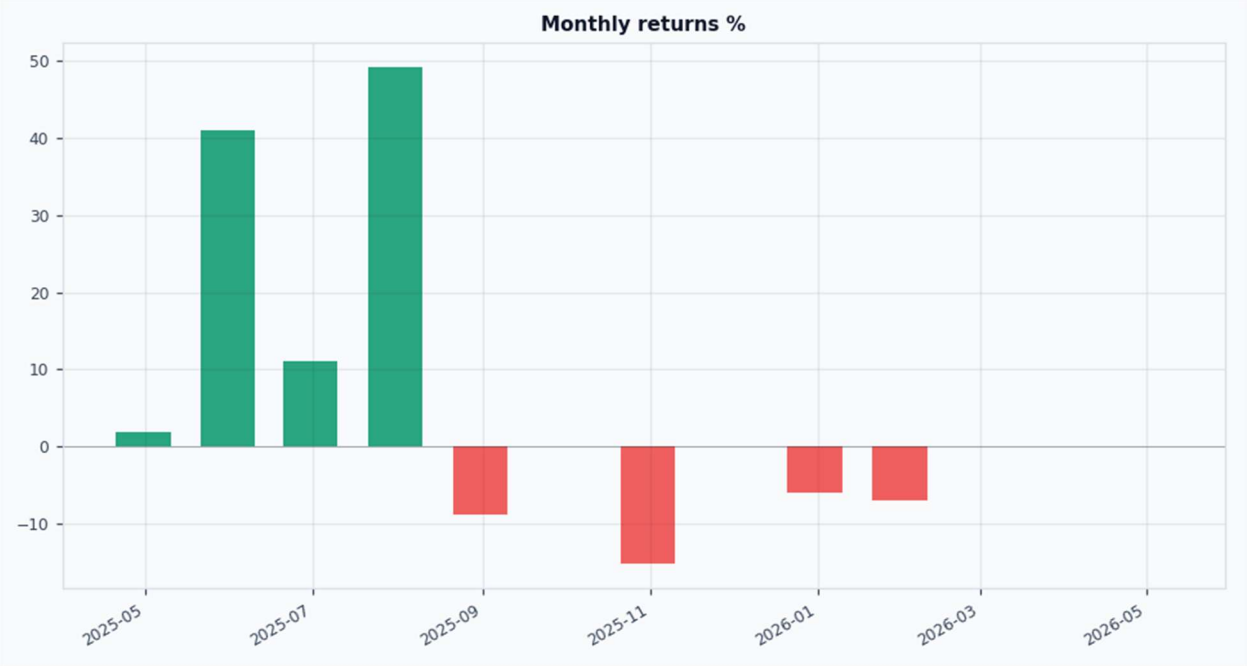
Use it to answer:

- Did my strategy enter where the rules say it should?
- Did it exit when the rules say it should?
- Did stop-based exits override signal-based exits?
- Are trades too rare, too frequent, or badly timed?

Many strategy design problems become obvious only when you inspect actual trades one by one.

15.6 Monthly Returns

The **Monthly Returns** view aggregates performance by calendar month and displays the result as a monthly return chart.



This is extremely useful because it helps you see **distribution and consistency over time**, rather than just the final outcome.

A strategy can end with a strong total return while still being highly inconsistent month to month. Monthly returns expose that pattern.

What Monthly Returns Show

This chart helps answer:

- Does the strategy produce gains consistently?
- Are profits concentrated in a few exceptional months?
- How often does the strategy have losing months?
- Are losing months small and manageable, or severe?
- Does performance appear stable across time?

Why This Matters

Custom strategies often look stronger than they really are when evaluated only by the final portfolio value. Monthly returns force a more disciplined view.

A strategy with many moderately positive months and manageable negative months is often more trustworthy than a strategy with a few huge months and many flat or negative ones.

What to Look For

Consistency

A good strategy often shows a reasonably balanced pattern of positive months, not just occasional large spikes.

Volatility of results

If monthly returns swing wildly between large gains and large losses, the strategy may be unstable or too dependent on particular market conditions.

Negative month severity

A strategy can tolerate losing months. Almost all real strategies have them. The question is whether the negative months are controlled or destructive.

Concentration of profits

If the majority of the total result comes from one or two exceptional months, you should be cautious. The strategy may not generalize well.

Seasonal Interpretation

Monthly returns can also hint at seasonality or regime dependence. For example, some strategies may work especially well in trending environments and poorly in sideways conditions. The chart does not prove why that happens, but it often shows that it does.

15.7 How to Evaluate a Custom Strategy Honestly

This is the most important part of the Results page.

A custom strategy should be judged with skepticism, discipline, and context. The job is not to "prove that it works." The job is to determine whether the apparent edge is credible.

The most common mistake in strategy research is to stop at the first attractive result. Honest evaluation requires resisting that impulse.

Start With the Right Question

Do not ask:

> How good does this result look?

Ask:

> What would make this result misleading?

This shift in mindset leads to much better strategy research.

A Practical Evaluation Framework

1. Check that the result is understandable

You should be able to explain:

- why the strategy enters
- why it exits
- why the parameters make sense
- what market behavior it is trying to exploit

If you cannot explain the logic clearly, you should not trust the backtest even if the performance looks strong.

2. Check the headline numbers

Use the summary cards first.

Ask:

- Is return attractive enough?

- Is drawdown acceptable?
- Is Sharpe reasonably positive?
- Is Calmar sensible?
- Is the trade count large enough?

If these are weak, the strategy likely does not deserve deeper work.

3. Inspect the equity curve shape

Ask:

- Is growth steady or chaotic?
- Does the strategy depend on one small part of the sample?
- Is outperformance broad or concentrated?
- Does it beat the benchmark in a meaningful way?

A believable edge usually has a believable equity path.

4. Study drawdowns seriously

Ask:

- Could this drawdown be tolerated with real money?
- How often do large drawdowns occur?
- How long does recovery take?
- Is the result only acceptable because the final endpoint hides severe pain?

Many strategies that look attractive on return alone fail here.

5. Examine the trade log

Ask:

- Are the trades logical?
- Are entries and exits happening where expected?
- Are a few massive outliers driving everything?
- Does the strategy trade often enough to support its claims?

If the trade-level behavior looks strange, the result probably is strange.

6. Check distribution, not just average

Use the monthly returns and the deeper metrics.

Ask:

- Are returns consistent?
- Are losses concentrated?
- Does the strategy have healthy expectancy?
- Does profit factor support the thesis?
- Is performance broad-based or dependent on rare events?

A strategy with smoother internal structure is usually more trustworthy than one with dramatic but irregular outcomes.

7. Be suspicious of complexity

A strategy with many parameters and excellent in-sample performance is not automatically better. In fact, it is often more dangerous.

More parameters usually mean:

- more flexibility
- more opportunities to fit noise
- less generalization
- harder interpretation

Whenever two strategies are similar in quality, the simpler one usually deserves more trust.

8. Be suspicious of low sample size

A beautiful result built on very few trades, a short history, or one specific market phase is weak evidence.

The smaller the sample, the less you should believe the result.

9. Compare against simpler alternatives

A custom strategy should justify its existence.

Compare it against:

- buy-and-hold
- a simpler version of itself

- a version with fewer filters
- closely related strategies

If the complex version only marginally improves the result, the extra complexity may not be worth it.

10. Separate profitability from robustness

A strategy can be profitable in one backtest and still not be robust.

Robustness means the result is not overly dependent on:

- one date range
- one market regime
- one exact parameter choice
- one lucky trade sequence

This is why the Results page must eventually be followed by:

- optimization sanity checks
- walk-forward analysis
- Monte Carlo analysis
- comparison against alternatives

Signs of a More Trustworthy Custom Strategy

A custom strategy is more credible when:

- its logic is simple and explainable
- its parameters have clear meaning
- it produces a reasonable number of trades
- the equity curve rises in a believable way
- drawdowns are controlled
- trade-level behavior matches the design intent
- monthly results are not wildly erratic
- performance is not dependent on a few extreme outliers

- benchmark comparison is favorable for a good reason

Warning Signs

Be cautious when you see:

- huge return with very few trades
- very high Sharpe from a short sample
- excellent result driven by one short period
- severe drawdowns hidden behind high final return
- strange trade behavior in the log
- many parameters with narrow optimal values
- strong optimization result but weak practical interpretability
- strategy complexity that cannot be clearly justified

A Good Final Discipline

Before accepting any custom strategy, try to state the bearish case against it.

For example:

- Maybe the strategy only works in one market regime.
- Maybe one or two large winners create the illusion of quality.
- Maybe the result depends on one fragile parameter setting.
- Maybe the benchmark would have been simpler and nearly as good.
- Maybe transaction costs or slippage matter more than expected.

If the strategy still looks strong after those objections, it becomes much more interesting.

Final Principle

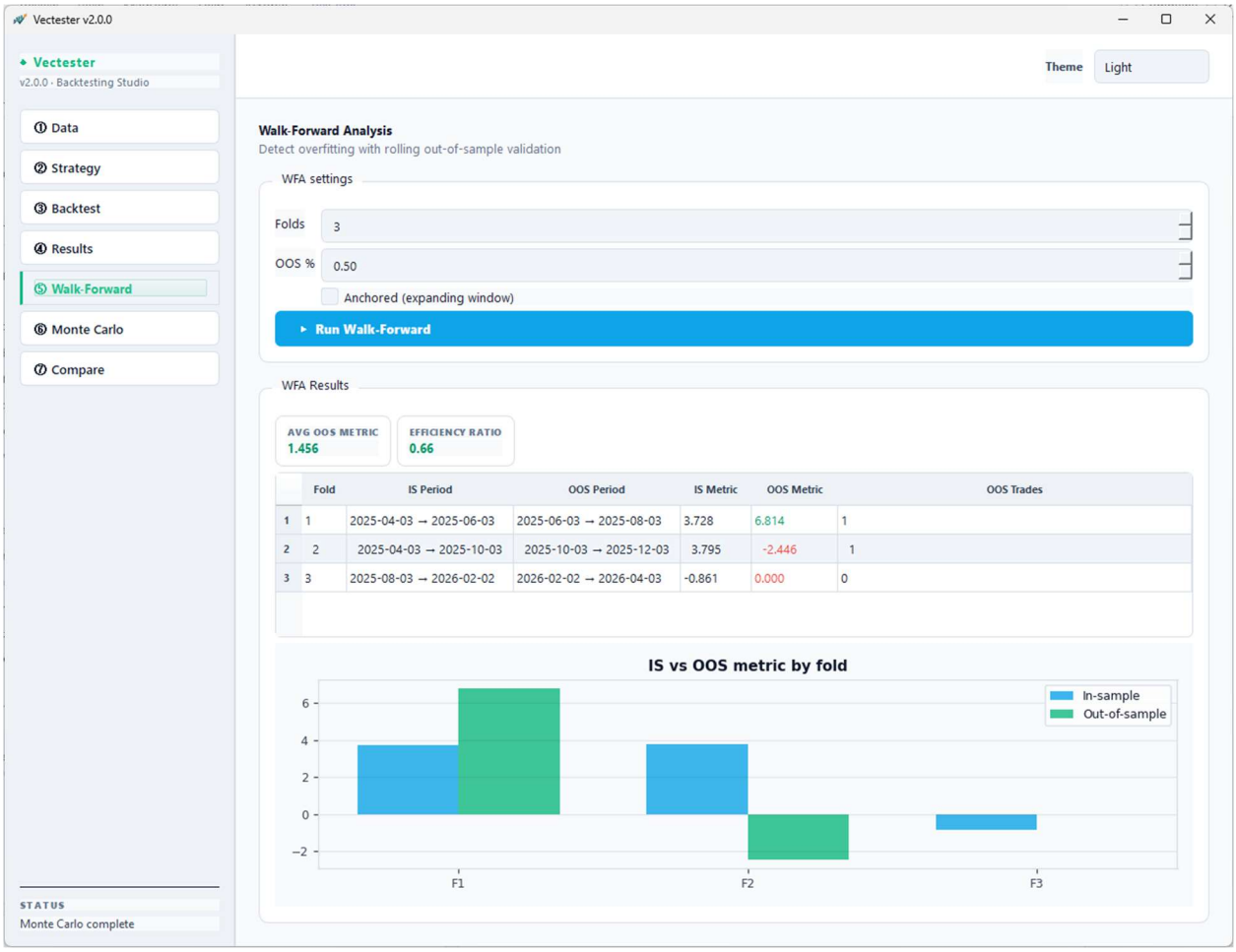
Do not try to confirm that your custom strategy is good.

Try to disprove it.

A strategy that survives serious doubt is far more valuable than one that merely looks impressive at first glance.

16. Walk-Forward Analysis

Walk-Forward Analysis, often shortened to **WFA**, is one of the most important validation tools in Vectester. A normal backtest tells you how a strategy performed on one continuous block of historical data. That is useful, but it does not tell you whether the strategy is genuinely robust or whether it only looks good because its parameters were tuned too closely to that specific history.



Walk-Forward Analysis addresses that problem by repeatedly splitting the available data into two parts:

- an **in-sample** segment, used for parameter selection
- an **out-of-sample** segment, used for validation

This process is repeated across multiple folds. Instead of asking only, “What were the best parameters on the whole dataset?”, WFA asks a much better question:

When the strategy is optimized on past data, does it still perform acceptably on the next unseen segment of data?

That is why WFA is one of the strongest tools in the application for evaluating custom strategies.

16.1 Why it matters for custom strategies

Custom strategies are powerful because they let you encode your own ideas, filters, entry rules, exits, stop logic, and regime conditions. But that freedom also creates the biggest risk in quantitative research: **overfitting**.

A custom strategy can easily become too tailored to the historical sample used during development. This often happens when:

- too many parameters are exposed
- parameter ranges are too wide
- too many filters are combined
- optimization is repeated until the result “looks good”
- the same dataset is used to design, tune, and judge the strategy

In such cases, the strategy may produce a strong backtest while having little or no real predictive value.

Walk-Forward Analysis matters because it tests whether a strategy’s edge survives after optimization. It simulates a more realistic research process:

1. Use historical data available up to a point in time.
2. Optimize the strategy on that past data.
3. Freeze those chosen parameters.
4. Test them on the next unseen segment.

If performance collapses out-of-sample, that is a warning sign that the custom strategy is learning the past too specifically instead of capturing a durable market tendency.

For custom strategies, WFA is especially important because it helps answer questions like these:

- Are my parameters stable, or are they fragile?
- Is my strategy logic general, or is it curve-fit?
- Does the edge survive in unseen data?
- Are the good results coming from real structure or lucky tuning?

A custom strategy that performs well only in-sample is not yet trustworthy. A custom strategy that holds up across repeated out-of-sample tests is far more credible.

16.2 Fold structure

In Vectester, Walk-Forward Analysis divides the dataset into a number of **folds**. Each fold is a time segment used for one cycle of optimization and validation.

Within each fold, the data is split into:

- an **in-sample portion**, used to search for the best parameters
- an **out-of-sample portion**, used to test those chosen parameters

The size of each fold is determined by the total dataset length and the selected number of folds. Then, within each fold, the **OOS %** setting determines how much of that fold is reserved for out-of-sample testing.

Example idea

Suppose you have 1,000 bars of data and choose:

- **5 folds**
- **20% OOS**

Vectester first divides the dataset into 5 segments of roughly equal size. Then, inside each fold:

- the last 20% of the fold becomes the **out-of-sample** section
- the earlier part becomes the **in-sample** section

So each fold becomes a small research cycle:

- optimize on the earlier part
- validate on the later part

This is important because the out-of-sample segment always comes **after** the in-sample segment in time. That preserves the natural order of market history and avoids peeking into the future.

Practical meaning of the fold count

A lower number of folds gives:

- larger segments
- more data per optimization

- fewer repeated validation cycles

A higher number of folds gives:

- more repeated tests
- shorter segments
- less data inside each fold

There is a trade-off. Too few folds may not test enough market regimes. Too many folds may leave too little data inside each fold for meaningful optimization and validation.

In practice:

- **3 to 5 folds** is usually a reasonable starting point
- more folds can be useful on long datasets
- shorter datasets should use fewer folds

If a fold is too small, the in-sample optimization becomes less reliable and the out-of-sample result becomes noisy.

16.3 In-sample optimization

The in-sample phase is where Vectester searches the optimization grid for the best parameter set.

This works the same basic way as normal optimization elsewhere in the app, but here it is done separately inside each fold.

What happens in-sample

For a given fold, Vectester:

1. takes the in-sample portion only
2. tests all parameter combinations from the optimization grid
3. calculates the selected optimization metric for each valid combination
4. selects the parameter set with the best in-sample score

The chosen metric might be, for example:

- Sharpe ratio
- total return
- Calmar ratio

- Sortino ratio
- another supported portfolio metric

The best in-sample parameters are then frozen and carried forward into the out-of-sample test for that fold.

Why this matters

This mirrors real strategy development. In real use, you would only have access to past data at the moment you choose parameters. You would not yet know what comes next. WFA reproduces that discipline.

Important interpretation point

A strong in-sample result is **not** proof that the strategy is good.

It only means:

- under this fold's past data
- with this parameter grid
- these parameters were the best found

That is all.

A strategy can look excellent in-sample for several reasons:

- genuine edge
- lucky parameter fit
- unusual market phase
- too many degrees of freedom
- too few trades
- an unstable metric

That is why the out-of-sample phase is the real test.

Good practice for in-sample optimization

When using WFA on custom strategies, keep the optimization grid realistic. If the grid is too large or too fine, WFA can become a repeated search for lucky parameter combinations. A smaller, more meaningful grid usually provides more honest information.

Good in-sample design usually means:

- only optimize parameters that truly matter
- use sensible ranges
- avoid testing excessive combinations
- prefer interpretable parameters over arbitrary ones

16.4 Out-of-sample validation

After the best parameters are found in-sample, Vectester applies them to the out-of-sample portion of the same fold.

This is the most important step in the entire WFA process.

What happens out-of-sample

For each fold, Vectester:

1. takes the best parameter set found in-sample
2. applies it unchanged to the out-of-sample segment
3. runs the strategy on that unseen data
4. calculates the selected metric
5. records the number of out-of-sample trades

The out-of-sample result shows how well the strategy generalizes beyond the data it was optimized on.

Why this is the real test

A strategy that only works on the data it was tuned on is not robust. A strategy that continues to work on the next unseen segment is much more promising.

Out-of-sample validation reveals whether the optimization found:

- a real and repeatable edge, or
- a parameter set that merely exploited quirks in the in-sample data

What to look for

When reviewing out-of-sample performance, ask:

- Is the metric still positive?
- Is it much weaker than in-sample?

- Does the strategy still trade meaningfully?
- Are the fold results reasonably consistent?
- Does performance collapse in most folds?

A strong strategy does not need to produce identical in-sample and out-of-sample numbers. Some degradation is normal. Markets change, and the best-fit parameters rarely remain equally strong on new data.

The key question is not whether out-of-sample is lower. The key question is whether it is still **good enough** to suggest the strategy has a real edge.

16.5 Anchored vs rolling

Vectester provides two ways to define the in-sample window across folds:

- **anchored**
- **rolling**

This setting changes how the in-sample segment moves through time.

Anchored

With **Anchored** enabled, the in-sample window always starts at the beginning of the dataset and expands as the folds progress.

That means each later fold uses all earlier available history up to the start of its out-of-sample segment.

Conceptually

Fold 1:

- optimize on early history
- test on the next segment

Fold 2:

- optimize on a larger history starting from the very beginning
- test on the next segment

Fold 3:

- optimize on an even larger history from the beginning
- test on the next segment

And so on.

When anchored is useful

Anchored mode is useful when:

- you want the strategy to learn from all prior data
- you believe older history remains relevant
- you want parameter selection to benefit from a growing sample
- you prefer a cumulative research style

Anchored mode tends to be more stable because later folds use more data. However, it may also dilute recent market behavior if older regimes are no longer relevant.

Rolling

With **Anchored** disabled, Vectester uses a **rolling** or **sliding** in-sample window.

In this mode, the in-sample segment moves forward through time instead of always starting from the beginning. Each fold uses a more local training window relative to its out-of-sample test period.

Conceptually

Fold 1:

- optimize on one historical block
- test on the next block

Fold 2:

- move forward
- optimize on the next historical block
- test on the next block after that

Fold 3:

- move forward again
- repeat

When rolling is useful

Rolling mode is useful when:

- you want to emphasize more recent market behavior
- you suspect old data is less relevant
- you want a more adaptive evaluation
- the market regime changes meaningfully over time

Rolling mode can be more realistic for strategies that are expected to adapt to current market conditions rather than rely on very old history.

Anchored vs rolling in practice

Neither mode is universally better.

Use **anchored** when:

- the strategy is based on broad, stable market structure
- long histories are valuable
- you want more data in each later optimization

Use **rolling** when:

- market behavior changes materially over time
- recent history matters more than old history
- you want to test adaptability

A robust custom strategy often looks acceptable under both methods. If it only looks good in one mode and fails badly in the other, that is worth investigating.

16.6 Reading fold results

Vectester presents WFA fold results in a table and a fold-by-fold comparison chart.

Each fold represents one complete optimization-and-validation cycle.

Fold table columns

Fold

This is the fold number. It identifies the sequence of the validation cycle.

IS Period

This shows the date range used for **in-sample optimization** in that fold.

It tells you exactly which historical segment was used to select parameters.

OOS Period

This shows the date range used for **out-of-sample validation** in that fold.

It is the unseen segment used to test the best in-sample parameters.

IS Metric

This is the value of the selected optimization metric achieved by the best parameter set on the in-sample segment.

This number answers:
How good did the strategy look during optimization?

OOS Metric

This is the value of the same metric when those chosen parameters are applied to the out-of-sample segment.

This number answers:
How well did the strategy generalize to unseen data?

OOS Trades

This is the number of trades taken in the out-of-sample portion.

This matters because a fold result based on very few trades is less reliable than one based on a meaningful sample.

How to interpret the table

The most important relationship is the difference between **IS Metric** and **OOS Metric**.

Healthy pattern

A healthy fold usually looks like this:

- IS metric is positive
- OOS metric is also positive
- OOS is somewhat lower than IS, but not dramatically worse
- OOS trade count is reasonable

This suggests the strategy retained some edge after optimization.

Warning pattern

A suspicious fold often looks like this:

- IS metric is very high

- OOS metric is weak, near zero, or negative

This suggests that the parameters may have been tuned too specifically to the in-sample data.

Weak-evidence pattern

Another weak pattern is:

- OOS metric looks strong
- but OOS Trades is extremely low

In that case, the result may not be statistically meaningful. A fold with one or two good trades does not carry the same weight as a fold with a larger trade sample.

Reading the fold chart

Vectester also shows an **IS vs OOS metric by fold** chart.

This chart helps you see consistency quickly.

What the chart is good for

It lets you spot:

- repeated degradation from IS to OOS
- folds with severe collapse
- folds where OOS remains stable
- whether poor results are isolated or systematic

Healthy chart pattern

A healthy chart usually shows:

- OOS bars lower than IS bars, but not dramatically lower
- mostly positive OOS bars
- no repeated catastrophic collapse

Concerning chart pattern

A concerning chart often shows:

- large IS bars and very small OOS bars
- many negative OOS bars

- inconsistent behavior across folds
- one lucky fold carrying the whole result

The chart is valuable because it stops you from being misled by averages alone. A decent average can hide highly unstable fold behavior.

16.7 Efficiency ratio

The **Efficiency Ratio** is Vectester's compact summary of how much in-sample performance survives out-of-sample.

It is calculated as:

```
average out-of-sample metric / average in-sample metric
```

In other words, it compares the average OOS result to the average IS result across all folds.

What it means

This ratio answers a very practical question:

How much of the optimized performance actually carries forward into unseen data?

If the ratio is high, the strategy preserved a meaningful portion of its in-sample strength.

If the ratio is low, the strategy lost most of its apparent edge when tested on new data.

How to interpret it

High efficiency ratio

A higher ratio suggests:

- better parameter stability
- lower degradation after optimization
- stronger evidence of robustness

This does **not** mean the strategy is automatically good. It means the strategy's performance is transferring more successfully from training data to unseen data.

Medium efficiency ratio

A middle-range ratio suggests:

- partial degradation
- possible edge, but with fragility
- some overfitting risk

This requires deeper review of individual folds.

Low efficiency ratio

A low ratio suggests:

- severe degradation
- optimization found something unstable
- high likelihood of overfitting or weak underlying edge

If the ratio is very low, the strategy may be mostly a product of parameter fitting.

Important caveats

The Efficiency Ratio is useful, but it must not be interpreted alone.

A ratio can be misleading when:

- the in-sample average is distorted by one extreme fold
- the out-of-sample sample size is too small
- the number of trades is too low
- the selected metric is unstable
- one fold dominates the average

For that reason, always review:

- the ratio
- the fold table
- the fold chart
- the OOS trade counts

Use the ratio as a summary, not as a replacement for judgment.

Practical interpretation approach

A sensible reading process is:

1. Check whether average OOS performance is positive.
2. Check whether the Efficiency Ratio is acceptable.
3. Inspect fold-by-fold consistency.

4. Look for negative or collapsing OOS folds.
5. Check whether trade counts are meaningful.

A strategy with a moderate ratio and stable folds may be more trustworthy than a strategy with a slightly better ratio but highly erratic fold behavior.

16.8 What WFA says about overfitting

Walk-Forward Analysis is fundamentally an **overfitting detector**.

It does not prove that a strategy will work in live trading, but it does provide strong evidence about whether the strategy's backtest quality is likely to be genuine or artificially optimized.

What overfitting looks like in WFA

Overfitting usually appears as this pattern:

- very strong in-sample performance
- weak or negative out-of-sample performance
- large fold-to-fold instability
- poor efficiency ratio
- results that depend on one or two lucky folds

This means the optimization process found parameters that explain the past well, but those parameters do not transfer effectively into unseen data.

What a healthier result looks like

A healthier WFA profile usually has:

- positive OOS performance in most folds
- degradation from IS to OOS that is noticeable but not catastrophic
- reasonable efficiency ratio
- meaningful OOS trade counts
- no single fold dominating the result

This does not guarantee future profits. It simply means the strategy has passed a more difficult and more honest validation test.

What WFA can tell you about your custom strategy

It can tell you whether your parameters are fragile

If small shifts in time cause performance to collapse, your parameter set may be too sensitive.

It can tell you whether your edge is local or general

If the strategy only works in one historical segment, it may be exploiting a temporary pattern rather than a durable one.

It can tell you whether the optimization grid is too aggressive

If optimization repeatedly finds excellent IS results that fail OOS, the search space may be too broad or too finely tuned.

It can tell you whether the strategy logic is too complicated

Strategies with too many interacting filters often look impressive in-sample and weak out-of-sample.

What WFA cannot tell you

WFA is powerful, but it has limits.

It does **not** tell you:

- that a strategy will definitely succeed live
- that market conditions will remain similar
- that transaction assumptions are perfect
- that the selected metric fully captures risk
- that a positive result is statistically sufficient by itself

WFA improves confidence. It does not eliminate uncertainty.

Best-practice conclusion

For custom strategy development in Vectester, Walk-Forward Analysis should be treated as a required validation step, not an optional extra.

A strong workflow is:

1. Build the strategy.
2. Test default parameters.
3. Optimize on a sensible grid.

4. Run Walk-Forward Analysis.
5. Reject or revise strategies that fail out-of-sample.
6. Only take seriously the strategies that remain reasonably stable across folds.

If a custom strategy looks excellent in a normal optimization but weak in WFA, trust the WFA.

The ordinary optimization tells you how good the strategy can be made to look on the past.

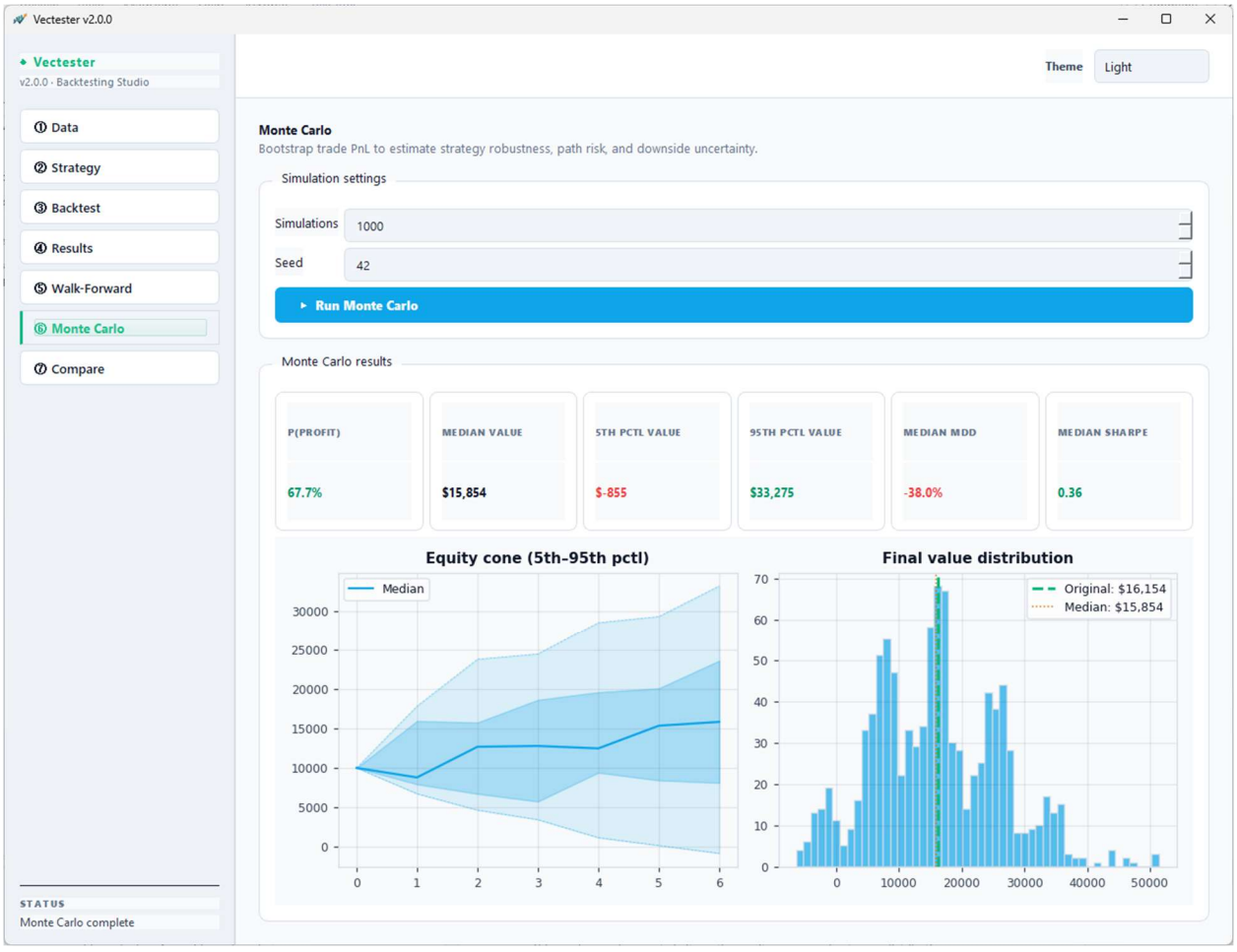
Walk-Forward Analysis tells you how much of that result survives when the past is no longer allowed to help.

17. Monte Carlo Analysis

Monte Carlo Analysis helps you answer a question that every backtest leaves open:

How much of this result comes from genuine strategy edge, and how much comes from the particular order in which trades happened?

A backtest shows one historical path. Monte Carlo analysis takes the trades generated by that backtest and creates many alternative paths from them. This lets you see whether the strategy remains healthy across many plausible variations, or whether the original result depends too heavily on a favorable sequence of wins and losses.



In Vectester, Monte Carlo is a **robustness test**. It does not create new market data, and it does not re-run your strategy on synthetic price series. Instead, it works from the **actual closed trades produced by your backtest** and simulates alternative equity paths from those trade outcomes.

17.1 Why it matters for custom strategies

This section is especially important for user-created strategies.

When you design your own strategy, it is easy to produce a backtest that looks strong on the surface:

- high total return
- attractive Sharpe ratio
- moderate drawdown
- good-looking equity curve

But a single historical run can be misleading. A strategy may look excellent simply because:

- its largest winning trades occurred early
- its losses happened in a forgiving order
- the historical sequence happened to smooth the equity curve
- a small number of trades dominated the final result

Monte Carlo analysis helps expose that kind of fragility.

For custom strategies, Monte Carlo is useful because it helps you judge whether:

- the result is **path-dependent**
- the strategy remains acceptable under less favorable trade ordering
- the original backtest is unusually lucky
- the drawdown profile is stable or unstable
- the strategy's edge is broad and repeatable, or narrow and fragile

A strategy that performs well only when trades happen in one especially favorable order is not robust. A strategy that still looks acceptable across many simulated paths is much more trustworthy.

In practical terms, Monte Carlo gives you a second layer of validation after the backtest:

- the backtest asks, "What happened?"
- Monte Carlo asks, "How sensitive was that result to trade ordering?"

That makes it one of the best tools for stress-checking a custom strategy before placing any confidence in it.

17.2 Trade-sequence simulation

Vectester's Monte Carlo engine works at the **trade level**.

After you run a backtest, the application takes the realized **trade P&L sequence** from that backtest. It then builds many simulated equity paths by reusing those trade outcomes in different ways.

This is not the same as randomizing candles or generating artificial prices. The simulation starts from the actual trades your strategy produced. That means Monte Carlo is analyzing the behavior of the strategy's realized outcome structure rather than inventing a new market.

In the current application workflow, Monte Carlo is run from the completed backtest result. If there are no trades, Monte Carlo cannot run.

What is being simulated

The simulation uses the strategy's closed-trade P&L values and reconstructs portfolio equity from them. For each simulation, Vectester:

1. starts with the current initial cash value
2. creates a new sequence of trade outcomes
3. cumulatively adds those trade P&L values to equity
4. measures the resulting final value, drawdown, Sharpe, and related statistics

This produces a large set of alternative equity curves and outcome distributions.

What this means conceptually

The purpose is to test how much the performance depends on the **order and sampling of trade outcomes**, not on the exact historical chart path.

This is useful because many strategies do not fail due to average profitability alone. They fail because:

- losing streaks arrive at the wrong time
- a bad run creates intolerable drawdown
- capital drops too far before recovery
- the realized historical sequence was much luckier than typical

Monte Carlo makes those risks visible.

17.3 Simulations

The **Simulations** setting controls how many alternative paths Vectester generates.

Each simulation is one complete hypothetical path built from the strategy's historical trades. The more simulations you run, the more stable and informative the output becomes.

What the setting does

If you choose:

- **100 simulations**, you get a fast but rough estimate
- **1,000 simulations**, you get a solid practical assessment
- **10,000 simulations**, you get a more stable distribution, but with longer runtime

Vectester's default value is **1,000**, which is a sensible balance between speed and statistical usefulness.

How to think about simulation count

A low number of simulations can produce noisy percentile values and unstable estimates. For example, the 5th percentile value may move noticeably from one run to another simply because the sample of simulations is too small.

A higher number of simulations gives:

- smoother percentile bands
- more stable probability estimates
- a more reliable view of downside risk
- a better estimate of how unusual the original result really was

Practical guidance

Use:

- **100 to 300** for quick testing
- **1,000** for normal research
- **2,000 to 5,000** when validating a promising strategy more carefully
- **10,000** only when you want maximum stability and do not mind the extra runtime

Increasing the simulation count does not improve the strategy itself. It only improves the quality of the estimate.

17.4 Seed

The **Seed** controls the random number generator used during simulation.

This matters for one reason: **reproducibility**.

Monte Carlo contains randomness. If you run the same simulation count twice with different random sequences, the exact results will differ slightly. The seed fixes that randomness so the same setup can be reproduced.

What the seed does

When the seed stays the same, and all other settings stay the same, Vectester generates the same sequence of random resampling decisions and therefore the same Monte Carlo output.

When the seed changes, the simulation uses a different random path generator, so:

- percentile values may shift slightly
- the histogram shape may change slightly
- probability estimates may vary slightly

Why this is useful

A fixed seed is useful when:

- documenting research
- comparing strategies fairly
- repeating a previous test
- sharing results with someone else
- exporting a report and wanting the numbers to remain identical

Best practice

Keep the seed fixed while comparing different strategies or parameter sets. Change it only if you intentionally want to check whether the conclusions are stable across different random draws.

If a strategy looks good only under one seed and weak under another, that usually suggests the strategy is fragile or the simulation count is too low.

17.5 Equity cone

The **Equity Cone** is one of the most useful visual outputs in Monte Carlo analysis.

It shows the range of possible equity paths across all simulations, rather than just one historical curve.

In Vectester, the cone displays percentile bands across simulated equity paths:

- **5th to 95th percentile band**
- **25th to 75th percentile band**
- **median path**

This lets you see both the likely range of outcomes and the concentration of typical outcomes.

How to read it

The median line represents the middle simulated path. Half of the simulations finish above it and half below it.

The outer band shows a broad range of plausible outcomes. The inner band shows a tighter range containing the more typical paths.

A **narrow cone** usually suggests:

- more consistent trade behavior
- lower path sensitivity
- more stable outcome structure

A **wide cone** usually suggests:

- stronger dependence on trade order
- greater uncertainty
- more variable drawdown behavior
- less predictable outcome quality

What the cone tells you

The cone helps you answer:

- How much can the equity path vary even with the same trade outcomes?
- Does the strategy remain acceptable across most paths?
- Could an unlucky ordering of trades create a much worse experience than the original backtest suggests?

A strategy with a very wide cone may still be profitable on average, but hard to live with in practice because the path to that result can vary dramatically.

Important interpretation point

The equity cone is not a forecast of future prices. It is a **distribution of possible equity paths implied by the trade sequence behavior of the backtest**.

It is best read as a robustness picture, not as a market prediction.

17.6 Final value distribution

The **Final Value Distribution** shows how often different ending portfolio values appear across all simulations.

This is the second major Monte Carlo visualization. While the equity cone shows the path through time, the final value distribution focuses only on the ending result.

What it shows

Each simulation ends with a final portfolio value. Vectester groups these final values into a distribution so you can see:

- the center of the result set
- the spread of possible endings
- the downside tail
- the upside tail
- where the original backtest result sits relative to all simulations

The chart usually includes markers for:

- the **original backtest final value**
- the **median final value**

How to interpret it

If the original final value sits near the middle of the distribution, the historical outcome is fairly typical relative to the simulated alternatives.

If the original final value is far to the right of the distribution, the backtest may have been unusually lucky.

If the distribution is very wide, the strategy's ending result is highly sensitive to sequencing and sampling effects.

If the lower tail reaches poor outcomes or losses, that signals meaningful downside uncertainty even if the historical backtest looked strong.

Why it matters

Two strategies can have the same original backtest result but very different final value distributions:

- one may cluster tightly around acceptable outcomes
- the other may have a huge spread with many weak or losing simulations

The first is generally more robust. The second is more fragile.

17.7 Probability metrics

Monte Carlo becomes especially useful when the output is reduced to clear probability-based measures.

Vectester computes several probability-style and percentile-based robustness statistics from the simulations. These convert a cloud of simulated outcomes into something you can evaluate directly.

P(Profit)

P(Profit) is the percentage of simulations that finish above the initial cash level.

This is one of the clearest summary measures.

A higher value means the strategy remains profitable across most simulated paths. A lower value means the strategy is much more dependent on favorable sequencing or sampling.

General interpretation:

- **very high**: the strategy has broad resilience across simulations
- **moderate**: the edge may exist, but outcomes are unstable
- **low**: the strategy may not be dependable

P(Loss)

P(Loss) is the percentage of simulations that finish below the initial cash level.

This is the mirror image of P(Profit). It highlights how often the strategy could plausibly end underwater given the same trade-level evidence.

A high loss probability is a warning sign, especially if the original backtest looked very attractive.

Probability of ruin

Vectester also tracks **probability of ruin**.

In practical terms, this measures how often simulated equity falls below a severe loss threshold during the path, not just at the end. This is important because many strategies fail operationally long before the final result is reached.

A strategy can still finish profitable in some cases while experiencing unacceptable interim collapse. Probability of ruin helps capture that kind of path risk.

This metric matters most for:

- leveraged strategies
- high-volatility strategies
- strategies with clustered losses
- systems that are hard to continue trading during deep drawdowns

Probability of beating the original result

Vectester also measures the proportion of simulations that finish **better than the original backtest result**.

This helps answer a subtle but important question:

Was the historical backtest typical, or was it unusually favorable?

If only a small fraction of simulations beat the original result, the original backtest may have been exceptionally lucky.

If many simulations beat the original result, the original may actually have been average or even below average.

Percentile values

Vectester also reports percentile values such as:

- **5th percentile final value**
- **50th percentile final value (median)**
- **95th percentile final value**

These are often easier to interpret than raw probabilities.

The:

- **5th percentile** represents a pessimistic but not extreme outcome
- **50th percentile** represents the middle outcome
- **95th percentile** represents a strong upside outcome

These values help define a realistic range of expected behavior.

Median drawdown

Vectester reports the **median maximum drawdown** across simulations.

This is important because your historical drawdown may have been unusually mild or unusually harsh. Median drawdown shows what a more typical worst-case decline looks like across many simulated paths.

A strategy with acceptable historical drawdown but terrible median simulated drawdown is less robust than it first appears.

Median Sharpe

Vectester also reports the **median Sharpe ratio** across simulations.

This gives a more distribution-aware view of risk-adjusted performance. Instead of trusting the Sharpe from one historical path, you can see what Sharpe looks like across many plausible trade-order variations.

17.8 What Monte Carlo says about robustness

Monte Carlo does not tell you whether a strategy is profitable in the future. It tells you whether the historical backtest appears **structurally stable or fragile**.

That is why it is best understood as a robustness test.

A robust strategy usually shows

A more robust strategy typically has several of these qualities:

- high probability of profit
- limited probability of loss
- low or moderate ruin probability
- a reasonably tight equity cone
- a final value distribution centered near acceptable outcomes
- an original backtest result that is not wildly better than the simulation median
- moderate simulated drawdowns rather than extreme ones

This does not guarantee future success, but it suggests the backtest is not entirely dependent on one lucky path.

A fragile strategy usually shows

A weaker or more fragile strategy often shows one or more of the following:

- low probability of profit
- wide dispersion of final values
- large gap between the original result and the simulation median
- large downside tail
- high ruin probability
- highly variable drawdowns
- a very wide equity cone

This suggests that the strategy may be too sensitive to sequencing effects, concentrated winners, or unstable trade behavior.

How to use Monte Carlo correctly

Monte Carlo should be used together with the rest of the Vectester workflow:

1. run the backtest
2. inspect the core metrics
3. review the trade log
4. run optimization if needed
5. validate with walk-forward analysis
6. run Monte Carlo to test path robustness

A strong strategy should not only backtest well. It should also survive this kind of stress check reasonably well.

What Monte Carlo does not prove

It is important not to overstate what Monte Carlo can do.

Monte Carlo does **not**:

- predict future market prices
- create new strategy logic
- guarantee future profitability

- fix overfitting
- replace walk-forward validation
- prove that the historical trades are representative of the future

It only tells you how sensitive the result is to reordering or resampling the observed trade outcomes.

Final takeaway

For custom strategies, Monte Carlo answers a critical practical question:

If this strategy had experienced the same kinds of trades in a less favorable order, would the result still look acceptable?

If the answer is yes, confidence increases.

If the answer is no, the strategy may need:

- simpler logic
- fewer parameters
- better risk control
- more validation
- less reliance on optimization

Practical reading checklist

When reviewing Monte Carlo results in Vectester, ask:

- Is **P(Profit)** convincingly high?
- Is the **5th percentile final value** still acceptable?
- Is the **median drawdown** tolerable?
- Is the **equity cone** reasonably tight?
- Is the **original result** near the middle of the distribution, or unusually lucky?
- Does the strategy still look tradable under less favorable sequencing?

If those answers are mostly positive, Monte Carlo supports the case that the strategy is reasonably robust.

If they are mostly negative, the backtest may look much stronger than the underlying strategy really is.

18. Comparing Strategies

The **Compare** page is designed to help you evaluate multiple strategy runs side by side in a structured and repeatable way. Instead of relying on memory, screenshots, or scattered notes, it gives you a single place to review how different strategies, parameter sets, and trading ideas perform on the same market data.

This is especially useful when you are developing your own strategies. A custom strategy may look impressive on its own, but its value becomes much clearer when it is placed next to simpler built-in systems, alternative parameter sets, or a baseline such as buy-and-hold.

Comparison should not be treated as a beauty contest where the highest return always wins. The goal is to understand **which strategy is stronger, more stable, and more usable**, and under what conditions.

18.1 Adding Current Result

Before a strategy can be compared, it must first be run as a normal backtest.

A typical workflow is:

1. Load your market data.
2. Select a strategy.
3. Configure its parameters.
4. Run the backtest.
5. Review the result.
6. Add that result to the comparison set.

When you click **Add current result**, Vectester takes the currently active backtest result and stores it in the comparison list. That stored result becomes one entry in the Compare page.

Each saved comparison entry represents a specific tested setup. In practice, that usually includes:

- the strategy name
- the active symbol or dataset
- the parameter set used
- the resulting performance metrics
- the equity curve of that run

This allows you to build a comparison set gradually. For example, you might:

- test a built-in trend-following strategy
- add it to comparison
- switch to your custom strategy
- test it on the same data
- add that result too
- then compare both side by side

You can repeat this as many times as needed. This makes the Compare page useful not only for comparing entirely different strategies, but also for comparing:

- different parameter versions of the same strategy
- optimized vs non-optimized results
- conservative vs aggressive risk settings
- long-only vs long/short variants
- built-in vs custom implementations

A good habit is to add only results that you have already inspected. If a backtest is clearly invalid, unrealistic, or based on a bad parameter combination, it is better not to add it, because poor entries make the comparison table less meaningful.

What counts as the "current result"

The "current result" is whatever backtest result is presently active in the application. That may be:

- the output of a **Run once**
- the best result from an **Optimize** operation

So if you optimize a strategy and then add the current result, you are usually adding the **best parameter combination selected by the optimizer**, not the entire optimization grid.

That distinction matters. The comparison page compares **finished backtest results**, not optimization searches.

Why this matters

Saving comparison entries lets you move from isolated testing to structured evaluation. Without comparison, it is easy to overvalue the most recent strategy you tested. With

comparison, you can see whether the new idea is actually better than what you already had.

18.2 Comparing Custom vs Built-In Strategies

One of the most valuable uses of the Compare page is testing your own custom strategies against the built-in strategies included with Vectester.

This matters because a custom strategy often feels more sophisticated simply because you created it. That can create bias. A fair comparison helps answer the real question:

Does the custom strategy actually improve on simpler existing approaches?

Why built-in strategies are important benchmarks

Built-in strategies provide reference points. They help you judge whether your custom logic adds genuine value or merely adds complexity.

For example:

- A custom momentum strategy should be compared against simpler momentum-based built-ins.
- A custom trend system should be compared against basic moving-average systems.
- A complex multi-filter strategy should be compared against a simpler version with fewer conditions.

If your custom strategy is not clearly better than a simpler built-in alternative, that is an important finding. It may mean:

- the custom logic does not add much edge
- the strategy is overcomplicated
- the strategy is overfit
- the added filters reduce trade quality instead of improving it
- the extra parameters make optimization look stronger than the true underlying signal

What to compare

When comparing custom and built-in strategies, focus on the following questions:

- Does the custom strategy improve return meaningfully?
- Does it improve risk-adjusted performance?

- Does it reduce drawdown?
- Does it trade more efficiently?
- Does it remain understandable?
- Does it require many more parameters to get only a small improvement?

A custom strategy should ideally justify its added complexity. If it uses more indicators, more rules, and more parameters, then the improvement should be meaningful, not marginal.

Compare on the same conditions

For this type of comparison to be valid, both strategies should be tested under the same conditions:

- same symbol or dataset
- same date range
- same timeframe
- same initial cash
- same fees
- same slippage
- same direction setting
- same evaluation metric context

If those conditions differ, then the comparison becomes harder to interpret. A strategy tested on different data or with lower fees is not directly comparable.

Use comparison to improve custom strategy design

The Compare page is not just for declaring winners. It is also a diagnostic tool.

For example, comparing a custom strategy against a built-in one may reveal that:

- your strategy improves Sharpe ratio but lowers total return
- your strategy raises return but causes much deeper drawdowns
- your strategy trades less often and depends heavily on a few winners
- your strategy is only better after optimization, but not with default values
- your strategy's equity curve is much less stable over time

Those findings help you refine your design. In many cases, comparison leads to simplification. You may discover that one of your filters contributes little and can be removed.

18.3 Ranking by Different Metrics

A strategy can look strong under one metric and weak under another. That is why comparison should never rely on a single number.

Different metrics answer different questions.

Total Return

Total Return shows how much the portfolio grew or declined over the test period.

This is often the first metric people look at because it is intuitive. A higher total return usually looks attractive. However, total return alone can be misleading because it says nothing about:

- risk taken to achieve that return
- size of drawdowns
- consistency of returns
- number of trades
- volatility of the equity curve

A strategy with the highest total return may also be the most fragile.

Sharpe Ratio

Sharpe Ratio measures return relative to volatility. It helps identify strategies that generate smoother performance instead of simply aiming for raw growth.

This is often one of the best metrics for ranking broadly diversified strategy ideas, because it rewards efficiency rather than aggression.

A higher Sharpe usually suggests:

- better consistency
- less noisy performance
- stronger return per unit of variability

But Sharpe can still be misleading if the strategy has very few trades or highly unusual return distribution.

Calmar Ratio

Calmar Ratio compares annualized return to maximum drawdown. It is especially useful when you care about how much pain a strategy creates on the way to its return.

A strategy with a strong Calmar ratio usually combines:

- respectable growth
- controlled drawdown

This is often more practical than pure return ranking, especially for real trading use.

Max Drawdown

Max Drawdown shows the worst peak-to-trough loss during the backtest.

This metric matters because even profitable strategies can become unusable if the drawdowns are too severe.

A strategy with a lower drawdown may be preferable even if its return is lower, because it may be:

- easier to hold psychologically
- safer to size
- more robust in live conditions

When ranking by drawdown, remember that lower magnitude is better.

Win Rate

Win Rate shows the percentage of profitable trades.

This metric is often misunderstood. A higher win rate does not automatically mean a better strategy. A strategy can have a high win rate and still perform badly if losers are much larger than winners.

Win rate is most useful when interpreted together with:

- average win
- average loss
- profit factor
- expectancy
- drawdown

Number of Trades

Trade count helps you judge whether the result is statistically meaningful.

A strategy with excellent metrics but only a handful of trades may not provide enough evidence. A larger trade sample generally makes performance more credible.

This does not mean high trade count is always better, only that very low trade counts should be treated carefully.

Why no single ranking is enough

A strategy may rank first in one metric and much lower in another. For example:

- highest return, but worst drawdown
- highest Sharpe, but lower total return
- highest win rate, but mediocre profitability
- lowest drawdown, but weak growth

That is normal.

A strong comparison process asks:

- Which metric matters most for this use case?
- Which weaknesses are acceptable?
- Which strengths are sustainable?
- Is the strategy balanced?

The best strategy is usually not the one that dominates one metric at all costs, but the one with the best overall profile.

Practical ranking approach

A sound method is:

1. Start with return and Sharpe.
2. Then examine drawdown and Calmar.
3. Check trade count for credibility.
4. Review equity shape for stability.
5. Use win rate only as supporting context.

This reduces the risk of being misled by one flattering number.

18.4 Normalized Equity Comparison

The normalized equity chart is one of the most powerful visual tools in the Compare page.

Instead of showing each strategy using its raw portfolio value, Vectester scales all compared equity curves to a common starting level. This allows you to compare their growth paths fairly, even if the original runs differ in scale or absolute value.

In simple terms, normalized equity answers this question:

If all strategies had started from the same base value, how would their growth paths compare over time?

Why normalization is useful

Raw portfolio curves can be difficult to compare directly because absolute account values may distract from the real issue, which is relative performance behavior.

Normalization removes that distraction and makes it easier to see:

- which strategy compounds faster
- which strategy experiences deeper setbacks
- which strategy is smoother
- which strategy spends long periods stagnating
- which strategy recovers better after losses

This is especially helpful when comparing:

- different strategies on the same asset
- different parameter versions of one strategy
- custom strategies vs built-in benchmarks

What to look for in the chart

A good normalized equity curve is not just one that ends highest. You should pay attention to the full shape.

Smoother upward progress

A strategy that climbs steadily is often more attractive than one that reaches a similar final value through violent swings.

Depth and duration of setbacks

Look at how far the curve falls and how long it stays below prior highs. Long flat or underwater periods can signal a less practical strategy.

Consistency across the sample

A curve that performs well only during one short period but stagnates elsewhere may be less reliable than a curve that advances in multiple market phases.

Relative behavior vs alternatives

The chart makes it easy to see whether a strategy leads consistently, briefly surges, or only catches up late.

Why this matters more than final value alone

Two strategies may finish at similar final values but behave very differently:

- one may rise steadily
- the other may depend on one large lucky period
- one may recover quickly after losses
- the other may remain weak for long stretches

Normalized equity exposes those differences immediately.

What normalization does not solve

Normalization does not make all comparisons valid automatically. It only standardizes the visual starting point. A comparison can still be unfair if the strategies were tested under different assumptions.

So the normalized chart should always be interpreted together with:

- the metrics table
- the same-data requirement
- trade count
- drawdown statistics
- the strategy logic itself

18.5 Fair Comparison Practices

The Compare page is only as useful as the quality of the comparison process. Poor comparison habits lead to misleading conclusions.

A fair comparison means that the tested strategies are evaluated under conditions that make the outcome meaningful.

Use the same dataset

This is the most important rule.

If one strategy is tested on different market data than another, then the comparison is weak. Strategies should generally be compared on:

- the same symbol
- the same timeframe
- the same date range
- the same bar history

Otherwise, performance differences may come from the data rather than the strategy.

Keep portfolio settings consistent

Use the same:

- initial cash
- fees
- slippage
- direction mode
- frequency assumptions

Even small changes in fees or slippage can materially affect results, especially for higher-turnover strategies.

Compare like with like

A strategy should be judged against relevant alternatives.

Examples:

- compare momentum with momentum
- compare trend-following with trend-following
- compare long-only systems with other long-only systems
- compare optimized versions against other optimized versions, not against untuned defaults

This does not mean cross-style comparison is useless, but comparisons are strongest when the systems serve similar purposes.

Be careful with optimized results

Optimization can make one strategy look much better than another simply because one has more tunable parameters.

A more complex strategy often has more chances to fit historical noise.

So when comparing optimized strategies:

- note how many parameters each strategy has
- consider how large the search grid was
- prefer strategies whose results remain strong in walk-forward analysis
- distrust large improvements that appear only after heavy optimization

A simpler strategy with slightly lower return but stronger validation may be the better choice.

Do not trust one metric

A fair comparison uses multiple perspectives:

- return
- Sharpe
- drawdown
- Calmar
- trade count
- equity shape
- validation results

Single-metric thinking often favors fragile strategies.

Give weight to robustness

A strategy is not strong just because it had the best historical result. It is stronger if it also shows:

- acceptable drawdown
- reasonable trade count

- stable equity progression
- good walk-forward behavior
- healthy Monte Carlo profile

This is especially important for custom strategies, where overfitting risk is higher.

Prefer understandable improvements

If a custom strategy beats a built-in strategy slightly but requires many more parameters and much more optimization effort, the improvement may not be worth the added complexity.

A fair comparison asks whether the gain is:

- meaningful
- robust
- explainable
- likely repeatable

Keep the comparison set clean

Avoid filling the Compare page with random or low-quality runs. A smaller set of carefully chosen comparison entries is much easier to interpret than a large set of noisy results.

A good comparison set usually includes:

- one or more baseline strategies
- your best built-in alternatives
- your custom strategy candidates
- different versions only when each version tests a real design idea

Use comparison as a research tool, not a scoreboard

The purpose of comparison is not simply to crown a winner. It is to learn:

- which ideas are stronger
- which improvements are real
- which risks are hidden
- which strategies deserve more validation

- which strategies should be rejected

That is what makes the Compare page valuable in serious research.

Practical Example

A sensible comparison workflow might look like this:

1. Test a built-in trend strategy on BTC-USD daily data.
2. Add the result to comparison.
3. Test your custom trend-plus-volatility strategy on the same data.
4. Add that result.
5. Test a simpler version of your custom strategy with one filter removed.
6. Add that result.
7. Open the Compare page and review:
    - return
    - Sharpe
    - drawdown
    - trade count
    - normalized equity curves
8. Keep the most balanced candidate.
9. Validate that candidate further with walk-forward and Monte Carlo.

This turns comparison into part of a disciplined strategy-development process.

Key Takeaways

- Use **Add current result** to build a structured set of tested runs.
- Compare **custom strategies against built-in baselines** to judge whether added complexity is justified.
- Rank strategies using **multiple metrics**, not just total return.
- Use **normalized equity charts** to compare growth path, consistency, and recovery behavior.

- Follow **fair comparison practices** so conclusions reflect strategy quality rather than inconsistent test conditions.

19. Strategy Writing Guide

This chapter explains how to design strategies that are useful, understandable, testable, and less likely to fail when market conditions change. Vectester makes it easy to create and modify strategies, but ease of creation does not guarantee quality. A strategy can look impressive in a backtest and still be fragile, overfit, or logically weak.

A good strategy is not just one that performs well on past data. A good strategy is one whose behavior can be explained clearly, whose parameters have a real purpose, and whose results remain reasonably stable when tested under different conditions.

19.1 Start simple

The best strategy designs usually begin with a small number of clear rules. A simple strategy is easier to understand, easier to debug, easier to optimize, and much easier to trust.

A common mistake is to begin with a highly complex idea: several indicators, many conditions, multiple confirmation layers, and advanced exits all at once. This often creates confusion. When such a strategy performs well, it becomes difficult to tell which part is actually useful. When it performs poorly, it becomes difficult to know what should be fixed.

A better approach is to begin with a minimal working concept. For example:

- a moving-average trend filter plus one entry rule
- a momentum threshold plus one exit rule
- a pullback entry inside an existing trend
- a breakout rule with a volatility filter

Starting simple gives you a clean baseline. Once that baseline exists, every later change can be judged against it. If a new condition improves robustness, that is useful. If it only improves one historical backtest while making the strategy harder to understand, it is probably not worth keeping.

Simple strategies also help reveal whether the underlying trading idea has real merit. If the core concept cannot produce reasonable behavior on its own, adding more complexity often hides the weakness instead of solving it.

Why simplicity matters

A simple strategy helps you:

- identify the true source of performance
- understand why trades occur

- spot logic errors quickly
- interpret optimization results more honestly
- reduce the risk of curve-fitting

Practical guidance

When creating a new strategy, begin with:

1. one market idea
2. one entry concept
3. one exit concept
4. a small number of parameters

Only after that baseline works should you consider adding confirmation filters, risk controls, or more specialized logic.

19.2 Use few parameters first

Parameters control strategy behavior. They determine things such as lookback periods, thresholds, stop distances, momentum requirements, and volatility filters. Parameters are powerful, but too many parameters quickly make a strategy unstable.

Every parameter adds freedom. More freedom means more ways to fit the past. This can make a backtest look better, but it also increases the chance that the strategy is merely adapting to noise rather than capturing a real market tendency.

A strategy with two or three meaningful parameters is often more reliable than one with eight or ten adjustable inputs. When many parameters are introduced too early, several problems appear:

- optimization grids become very large
- results become harder to interpret
- the strategy becomes sensitive to small changes
- the best settings may be accidental rather than meaningful

A good early goal is to keep the strategy as compact as possible. Ask whether each parameter has a distinct purpose. If two parameters influence nearly the same behavior, one of them may be unnecessary.

Good parameter discipline

In the early version of a strategy, prefer parameters like:

- one trend lookback
- one entry threshold
- one risk control value

Avoid adding many optional toggles or several overlapping thresholds before the basic idea has proven itself.

Questions to ask about every parameter

Before keeping a parameter, ask:

- What exactly does this parameter control?
- Why does the strategy need it?
- Can the strategy work without it?
- Does changing it produce understandable changes in behavior?
- Is it measuring something different from the other parameters?

If the answer is unclear, the parameter may not belong in the design.

19.3 Prefer interpretable logic

A strategy should be understandable in plain language. You should be able to explain what it does, why it enters, why it exits, and what market behavior it is trying to exploit.

Interpretable logic means the strategy rules make sense as a coherent idea. For example:

- buy when price is above a long-term trend filter and short-term weakness creates a pullback entry
- enter when momentum strengthens after a quiet volatility period
- avoid trades unless trend, participation, and price structure align

This kind of logic is easier to trust because it can be described clearly. It also helps you detect whether the strategy behaves as intended.

A strategy becomes hard to trust when its logic turns into a collection of unrelated conditions. For example, if one indicator suggests trend-following, another suggests mean reversion, and a third is added only because it improved a past result, the overall design may stop making sense.

Interpretable logic is especially important when results disappoint. If the strategy is understandable, you can improve it intelligently. If the strategy is opaque, any change becomes guesswork.

Signs of interpretable logic

A strategy is more interpretable when:

- every rule has a clear role
- the rules support the same market idea
- entries and exits are explainable in normal language
- parameter changes produce predictable effects
- the strategy can be summarized briefly and accurately

Why this matters

Interpretable strategies are easier to:

- maintain
- explain to others
- compare against alternatives
- improve over time
- reject when they are not truly working

A strategy does not need to be simplistic, but it should always remain understandable.

19.4 Avoid redundant filters

Many strategies become bloated because extra filters are added without enough justification. A redundant filter is a rule that adds little real information because it overlaps heavily with conditions already in the strategy.

For example, using multiple trend indicators that all measure nearly the same thing may not improve the strategy in a meaningful way. It may simply make entries rarer and the historical fit tighter. The result can look cleaner in-sample while becoming weaker out-of-sample.

Redundant filters often appear when a developer keeps adding conditions to remove losing trades from past data. This is dangerous. Removing bad-looking trades from history is not the same as improving future performance.

Common forms of redundancy

Redundancy often appears as:

- multiple trend filters that all confirm the same direction

- several momentum indicators with very similar behavior
- multiple volatility rules that block the same periods
- confirmation conditions added only because they improve one metric
- extra boolean conditions that reduce trades without improving robustness

How to identify a redundant filter

A filter may be redundant if:

- removing it changes very little
- it only improves results on one specific dataset
- it sharply reduces the number of trades without clear benefit
- it measures nearly the same concept as another rule
- it makes the strategy harder to explain

Better practice

Before keeping an added filter, compare the strategy with and without it. Ask:

- Does it improve out-of-sample behavior?
- Does it improve robustness under walk-forward analysis?
- Does it improve drawdown or consistency meaningfully?
- Does it still make sense conceptually?

If not, the filter is probably unnecessary.

A strategy should contain only rules that clearly contribute something distinct and useful.

19.5 Avoid overfitting

Overfitting is one of the most serious risks in strategy development. It happens when a strategy is shaped too closely to past data. An overfit strategy often performs extremely well in the historical sample used to develop it, but much worse when exposed to new data.

Overfitting can happen in several ways:

- too many parameters
- too many optimization combinations
- repeatedly adjusting rules until one dataset looks good

- adding filters that target specific historical losses
- trusting one best result without checking stability

An overfit strategy often has attractive headline metrics, but weak real reliability. Its success comes from matching the quirks of the sample rather than capturing a durable pattern.

Warning signs of overfitting

Be cautious when you see:

- very high performance from a highly complex rule set
- strong results concentrated in a small number of trades
- large changes in results from small parameter changes
- one “best” optimization point surrounded by poor nearby settings
- excellent in-sample results and weak out-of-sample results
- a walk-forward efficiency ratio that collapses
- Monte Carlo results that show wide fragility

How to reduce overfitting risk

You can reduce overfitting by:

- keeping the strategy simple
- limiting the number of parameters
- using focused optimization ranges
- preferring broad stable regions over sharp peaks
- testing on multiple symbols or market conditions
- using walk-forward analysis
- using Monte Carlo analysis
- comparing against simpler alternatives

A strategy does not need to be perfect. It needs to be believable. A moderate result that remains stable is often more valuable than a spectacular result that depends on exact settings.

Stability matters more than perfection

When reviewing optimized results, do not ask only, “What is the highest Sharpe ratio?” Also ask:

- Are nearby parameter values similar?
- Does the strategy still work with slightly different settings?
- Does the logic remain consistent across periods?
- Does out-of-sample behavior remain acceptable?

Robustness is more important than the single best backtest.

19.6 Build strategies in stages

Good strategy development is usually incremental. Instead of trying to write the final version immediately, build the strategy step by step and evaluate each stage.

This staged approach makes development more controlled and much easier to understand. You know what changed, why it changed, and whether the change was truly helpful.

A useful sequence is:

Stage 1: Core entry idea

Start with the central signal only. This should represent the main market hypothesis. Ignore advanced filters at first.

Stage 2: Basic exit logic

Add a reasonable exit approach. This might be an indicator-based exit, a stop-loss/take-profit structure, or a simple opposite-signal exit.

Stage 3: Risk controls

Introduce stop-losses, take-profits, trailing stops, or position constraints only after the core signal behavior is clear.

Stage 4: One meaningful filter

If needed, add one filter that clearly improves the logic, such as a trend regime or volatility condition.

Stage 5: Refinement

Only after the earlier stages are understood should you tune parameters or test more specialized additions.

Benefits of staged development

Building in stages helps you:

- isolate the effect of each component
- avoid unnecessary complexity
- debug more efficiently
- understand which parts create value
- reject weak additions early

What to record during staged development

It is useful to track:

- what changed
- why it was added
- how trade count changed
- how drawdown changed
- how risk-adjusted metrics changed
- whether out-of-sample stability improved

A strategy should evolve for clear reasons, not by random accumulation of ideas.

19.7 Test defaults before optimizing

Default values are not just placeholders. They represent the strategy's initial design assumptions. Before running optimization, you should always test the strategy using its default parameter values.

This is important because optimization should improve an already coherent design, not rescue a broken one. If a strategy produces nonsense, no trades, or obviously weak behavior under its defaults, immediate optimization often wastes time and encourages overfitting.

Testing defaults first tells you whether the strategy makes sense in its natural form. It also gives you a baseline for comparison.

What defaults should achieve

A reasonable set of defaults should:

- produce understandable trades

- reflect the intended logic of the strategy
- avoid extreme or unrealistic assumptions
- provide a useful baseline even if not optimal
- show whether the core idea is viable

A default configuration does not need to be the best performer. It should be a credible starting point.

Why optimization should come later

If you optimize too early, you risk:

- hiding logic problems
- fitting noise instead of fixing design
- selecting parameters that only work historically
- losing sight of what the strategy was meant to do

Optimization is most useful when the defaults already produce a sensible structure and you want to refine, compare, and validate behavior.

Practical default-testing checklist

Before optimizing, check whether the default strategy:

- runs without errors
- produces a reasonable number of trades
- behaves in line with its stated idea
- has interpretable entries and exits
- avoids obviously pathological results
- shows at least some promise compared with simpler baselines

If the defaults fail badly, revise the strategy logic before exploring parameter sweeps.

19.8 Validate before trusting results

A backtest result is not proof. It is evidence, and that evidence must be tested. Even a strategy with good metrics should not be trusted until it has survived validation.

Validation means checking whether the result is robust, not just attractive. In Vectester, this means going beyond a single run and using the available analysis tools to challenge the strategy.

Why validation matters

A single strong result can be misleading for many reasons:

- lucky trade ordering
- unusually favorable market period
- excessive parameter tuning
- low trade count
- hidden dependence on one symbol or regime

Validation helps answer the real question: does this strategy have a believable edge, or did it merely perform well in one historical setup?

Core validation steps

1. Compare against a simple baseline

Check whether the strategy meaningfully improves on basic alternatives, such as buy-and-hold or a simpler rule set. If it does not, the added complexity may not be justified.

2. Review trade count and behavior

A result based on too few trades is hard to trust. Examine whether the trade sample is large enough to support conclusions.

3. Run optimization carefully

Look for stable regions, not just the single best point. A robust strategy should not collapse when parameters move slightly.

4. Use walk-forward analysis

Walk-forward analysis shows whether in-sample success survives in unseen out-of-sample segments. This is one of the most important checks against overfitting.

5. Use Monte Carlo analysis

Monte Carlo helps measure how much of the result may depend on trade ordering. A fragile result often looks much weaker when randomized.

6. Compare variations

Test nearby versions of the same idea. If tiny changes destroy performance, the strategy may be too delicate.

7. Check for conceptual consistency

Make sure the results still match the logic of the strategy. If a strategy makes money for reasons you cannot explain, caution is warranted.

What stronger evidence looks like

Confidence is higher when:

- the strategy works with sensible defaults
- optimized regions are stable
- walk-forward results remain acceptable
- Monte Carlo outcomes remain reasonable
- trade count is not too small
- results hold across more than one market condition
- the logic remains understandable throughout

Final principle

Do not trust a strategy because it has impressive numbers. Trust it only after it remains credible under scrutiny.

A strong development process is not about finding the most beautiful historical chart. It is about building a strategy that still makes sense after you try to disprove it.