Building a Production Quant Trading System (Free Sample: Preface, Parts I-II)

A practitioner's guide to the unglamorous 90% (validation, risk, plumbing, and operations) that decides whether a trading idea survives contact with a live market.

Dr. Rayan Azhari

Revision 1.27 (2026-07-16)

Building a Production Quant Trading System (Free Sample: Preface, Parts I-II)

Revision 1.27 (2026-07-16).

Parts I and II are free to read at https://www.rayanazhari.co.uk/titan-quant-book/ . The complete book (Parts III to VII and the appendices) is a paid edition.

Educational material. Not investment advice. No warranty. Trading involves a substantial risk of loss; nothing here is a recommendation to trade any instrument, and the example strategies have no expected edge.

Typeset in Source Serif 4 and IBM Plex Mono.

Contents

Preface

A practitioner's guide to the unglamorous 90% (validation, risk, plumbing, and operations) that decides whether a trading idea survives contact with a live market.

By Dr. Rayan Azhari · Source & framework on GitHub · About & licence (see "About") (Parts I & II free; complete book paid; all rights reserved)

Info: Revision 1.27 · last updated 2026-07-16
See the revision history at the foot of this page for what changed.

Most writing about quant trading is about **finding edges**. This book is about everything that happens *after* you think you've found one: proving it's real, wiring it to a broker without lying to yourself, sizing it so a bad month doesn't end you, and running it as a process that survives restarts, data outages, and your own future mistakes.

It is built around a single real system (a multi-strategy, broker-connected book we'll call **Titan**) used as the running case study. Titan is opinionated and battle-scarred: it has shipped look-ahead bugs, mis-sized a leg by a third on a currency assumption, crash-looped on its own data-quality gate, and had an entire quarter of "validated" results deleted because the methodology underneath them couldn't be trusted. Those scars are the point. You'll learn more from *why* each guardrail exists than from any clean architecture diagram.

Note: What this book is, and isn't
It **is** a guide to the engineering and methodology of a production systematic-trading stack: research validation, portfolio construction, risk management, deployment, and operations. It **is not** a source of alpha. Specific parameters, instrument lists, and live performance are deliberately omitted or replaced with clearly-labelled illustrative values: the value here is the *process*, which is exactly what's safe to share. It is educational material, **not investment advice**.

Note: The book and the code
A public companion repo, `titan-quant`, holds the sanitised framework and an educational demo: the validation, risk, and metrics modules the chapters describe, runnable on sample data, with strategy specifics redacted by design. Chapters reference real modules, but

the book is **read-along, not clone-and-copy**: you can run the *mechanisms* (the gates, the bootstraps, the decision function), not lift a live edge. Appendix D: Run it yourself (see "Runnable Walkthrough") is the hands-on walkthrough.

Who this is for

A working engineer or quant-curious developer who can read Python and knows what a Sharpe ratio is, but hasn't yet built the full pipeline from "interesting backtest" to "thing that trades unattended." If you've ever had a strategy look amazing in a notebook and then quietly bleed in production, this book is the missing middle.

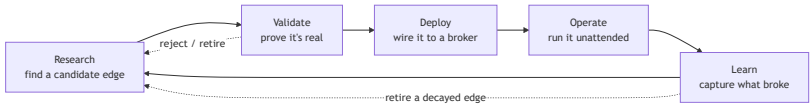
The concrete examples assume a specific stack (**NautilusTrader + Interactive Brokers + Docker + Python**), but the methodology transfers if your broker or framework differs: the discipline is the point, not the brand.

The one idea, if you read nothing else

> **Suspicion over celebration.** When a result looks too good, the correct first reaction is not excitement; it's *"where's the leak?"* Out-of-sample tests, deflated statistics, held-out windows, and parity checks all exist to answer one question: is this real, or am I fooling myself? Every chapter is, in some sense, a different way of not fooling yourself.

The lifecycle this book follows

The whole book is organised around one loop. A strategy is never "done"; it moves around this cycle continuously, and most of the engineering exists to make each arrow trustworthy.



How it's organised

Part	You'll learn to...
I, Foundations & Architecture	Lay out a system whose structure *prevents* whole classes of bugs.
II, Research & Validation	Run a backtest you can actually trust, and decide go/no-go with a function, not a feeling.
III, Data Engineering	Source and store data, and gate the system against silently-regressed data.
IV, From Research to Production	Turn a validated idea into a live strategy that provably computes the same thing the backtest did.
V, Portfolio & Risk	Combine strategies, size positions, and build the layer that keeps a bad day from being a fatal one.
VI, Deployment & Operations	Walk one strategy end-to-end, containerise the stack and run it (deploy, redeploy, monitor, halt, recover), reconcile against the broker, and detect and retire a decayed edge.
VII, Reflections	What's still unfinished, what we'd do differently, and where to read next.

How to read this book

It is **45 chapters across seven parts** (plus a runnable appendix), roughly **90,000 words**, a few evenings front-to-back, or an afternoon for a single part. The weight is not even: Parts II (research & validation) and VI (deployment & operations) are the longest and densest, half a day each if you work the exercises; Parts I and VII are short framing reads you can do in a sitting. Read it as a build log, or take one of three paths:

- **Validation-first** (*you have an edge and you don't trust it*): Part II → Part V → Part IV.
- **Ship-it** (*you have a strategy and you need it live and safe*): Part I → Part VI → Part IV.
- **Reference**: the left-hand nav is the index; each chapter ends with **Exercises** (collapsible answers) and a forward link to the next.

The parts build on each other (Part V assumes Part II's validation vocabulary), so a reference reader jumping mid-book may need to back-fill. And if you read only one chapter, read **A backtest you can trust** (see "Backtest You Can Trust"): it contains the discipline the rest of the book depends on. Once that discipline has sunk in, the recommended second read is **One strategy, end to end** (see "Capstone End To End"), the build-log capstone

that threads a single illustrative sleeve through the whole loop, so the separate disciplines become one chain rather than seven parts. Want to *run* something? Appendix D (see "Runnable Walkthrough") walks you through cloning the companion framework and watching the gates fire.

Tip: Conventions
Code blocks are sanitised, runnable-shaped *patterns*, not copy-paste alpha. Callouts flag the things that bite: `!!! warning` for anything that can corrupt a result or a risk calculation, `!!! danger` for anything that touches live capital. Numbers in examples are **illustrative unless explicitly stated otherwise**. This matters more than it sounds, because mistaking an illustrative figure for a real one is the exact self-deception the book exists to prevent. Illustrative values are flagged in-text (e.g. *"a Sharpe near +2.0 (illustrative)"*, *"`M = 20`, illustrative"*); a number carrying no such flag and no cited source should be read as illustrative, never as live edge.

Revision history

The book tracks the live system, so it gets revised as Titan does. Each revision is dated; the footer of every page carries the current revision number. Found an error? The book is revised as the system changes: open an issue on the companion repo or reach the author via About (see "About").

Revision	Date	What changed
Revision	Date	What changed
1.27	2026-07-16	**Author's pass over the 1.26 audit material, plus the resolution of the one thing 1.26 left contradictory.** The regime gate (see "Layered Safety") is no longer dead config: it is **armed on purpose**, with a passive feed calling `update_vix()` each bar, and it fails **open** (a missing or stale volatility series leaves the scalar at 1.0 rather than de-grossing on bad data), so the rule sharpens from "delete it or mark it dead" to "resolve it explicitly, delete it or arm it deliberately and price what you armed". The drawdown throttle (see "Portfolio Risk Manager") now carries the band arithmetic in full: the corrected ramp flattens at a **13.75%** drawdown, where `(|dd| – heat)/band` reaches 0.75, and a scaled-ramp alternative that lands the floor exactly on the 15% kill line is given, because what is indefensible is calling the first one a cliff-free ramp to 15% when it flattens 1.25 points earlier. Deeper passes on the metric suite (see "Metric Suite") (CVaR/CDaR sign discipline: gate on the magnitude with the sign kept, never `abs()` in alert logic), the correlation dial (see "Allocator Correlation Dial") (a worked scenario table), tail risk and ruin (see

Revision	Date	What changed
1.26	2026-07-16	**The audit revision: what a live risk audit found, and the verifier bugs behind it.** A proactive audit of the running system found four safety controls *silently inert* while every board stayed green, and a parallel audit found four conclusion-flipping bugs in the *verification* code itself. Both are now war-stories. In Part II (free): a new **Verify your verifier (see "Backtest You Can Trust")** section (independently reimplement the load-bearing cell; audit assuming a bug) plus the window that deleted the crash; the **deflated-Sharpe gate that rubber-stamped every winner (see "Deflated Sharpe")** (an annualised Sharpe fed into a daily variance term inflated `z` roughly tenfold and pinned `dsr_prob` near 1.000), with the frequency walkthrough corrected end to end; a vol-matched comparison caveat in the metric suite (see "Metric Suite"); and a public-index margin-call worked example plus a "length is not coverage" bootstrap caveat in tail risk and ruin (see "Tail Risk And Ruin"). In Parts V to VI: the correlation dial's (see "Allocator Correlation Dial") freshness guard firing perfectly for 46 days while the book ran ungoverned (fail-safe as fail-**open**); `regime_scale` named as dead config

Revision	Date	What changed
1.25	2026-06-30	Second-pass review of the items the 1.24 remediation had deliberately deferred: applied the ~10 that were genuinely additive (the rest stay skipped as duplicative, owned by another chapter, or thesis-preserving). Adds the alpha-side-killers scope note + a four-gate “spine” box to Ch. 1 (see “Why Systems Fail”); a “three things a single-snapshot parity test still misses” note (transient, stateful, multi-feed) to live == research (see “Live Equals Research”); partial-fill protective-leg and terminal-vs-transient retry guidance to broker realities (see “Broker Realities”); a per-leg currency-conversion note to per-strategy equity & FX (see “Per Strategy Equity Fx”); self-consistent gate snippets in the data-quality gate (see “Data Quality Gate”); and a “how a bug becomes a catalogue entry” intake note in the failure-mode catalogue (see “Failure Mode Catalogue”).

Revision	Date	What changed
1.24	2026-06-30	**Audit-driven remediation** of the whole improvement backlog (a multi-agent re-read of every chapter against every backlog item). Fixed the 8 remaining high-severity gaps (the capstone as a signposted "second read", a complete A-code/V-era register + a scannable war-story index (see "Failure Mode Catalogue"), the data-gate's (see "Data Quality Gate") operator-facing `Regression` artefact (with the ch.13 overlap compressed), the correlation dial's (see "Allocator Correlation Dial") `1.0`-ceiling-vs-`1.5`-default honesty note, a bar-by-bar de-risk-ladder (see "Layered Safety") trace, and a standalone verdict-mapping on the pre-flight checklist (see "Preflight Checklist")), plus ~130 medium/low improvements folded surgically across 28 chapters (worked Kelly and ruin examples, a closed-form first-passage cross-check, a drawdown-gate summary table, config-validation and units-contract notes, new glossary terms, and many cross-references). No new chapters; existing-chapter depth only.

Revision	Date	What changed
1.23	2026-06-29	Completed the **Part B** existing-chapter pass: surgical worked-number and operational additions across Parts I–VII closing the high-severity gaps from the improvement backlog, without new chapters. Highlights: a fully-numeric de-risk **ladder** and a flatten-selection criterion in layered safety (see “Layered Safety”); a build-time import-check Dockerfile, concrete heartbeat constants + atomic writer, and an `fx_market_likely_open()` sketch in containerizing (see “Containerizing”); a post-redeploy health gate, an escalation/paging table, and the exact `reset_halt` invocation in the live runbook (see “Live Runbook”); an operational “clean soak” definition, a pre-live **slippage-budget** technique, and a data-parity check in paper to live (see “Paper To Live”); explicit **exit criteria** and a severity ranking in caveats (see “Caveats Open Problems”); Sortino, Almgren-Chriss, and survivorship citations in the reading list (see “Reading List”); and operationalised appendices: an **A–Z index** + ops-vocabulary in the glossary (see “Glossary”), halt-persistence + the five-term `scale_factor`

Revision	Date	What changed
1.16–1.22	2026-06-29	Seven Tier-3 additions: new chapters Time, sessions & calendars (see “Time Sessions Calendars”), Order lifecycle & idempotency (see “Order Lifecycle”), When the strategy is a model (the ML lifecycle) (see “Ml Lifecycle”), Live TCA & best-execution evidence (see “Live Tca”), Running an incident & the postmortem (see “Incident Postmortem”), and The operator under fire (see “Operator Under Fire”); plus a covariance-estimation/shrinkage deepening and a realised-vs-assumed correlation-drift band folded into the allocator and layered-safety chapters.

Revision	Date	What changed
1.5–1.15	2026-06-29	Eight new chapters + a runnable appendix completing Tiers 1–2 of the improvement backlog: Factor & beta exposure (see “Factor Beta Exposure”), The execution layer (see “Execution And Costs”), The trade record vs the log (see “Trade Record Audit Trail”), Alerting & the dead-man’s switch (see “Alerting Deadmans Switch”), Solo operator or team (see “Solo Operator Vs Team”), Capacity & crowding (see “Capacity And Crowding”), Combining forecasts (see “Combining Forecasts”), Observability (see “Observability”), and State durability & disaster recovery (see “State Durability Dr”); a Legal & regulatory pre-live bucket; Appendix D: Run it yourself (see “Runnable Walkthrough”); and end-of-chapter **Exercises** (with collapsible answers) throughout.

Revision	Date	What changed
1.4	2026-06-26	New Part VI chapter Reconciliation: making the broker the book of record (see “Reconciliation”). Generalises the boot-time restart-adoption story into a *continuous* discipline: the broker is the book of record and your in-memory state is a cache that drifts. Covers the five break classes (missed fill, out-of-band manual close, corporate-action split, cash/NLV/FX drift, stuck order), the three-way reconciliation (book vs broker vs trade log), the alert→confirm→halt action ladder (and why auto-adopt is forbidden), and reconciliation as a retained evidentiary artifact. Honest about what Titan reconciles today (an internal-book drift watchdog with one auto-halt path) vs what’s still unmodelled (the cross-currency NLV invariant). Closes the dangling “reconciliation orphan trip” the runbook operates but never defined.

Revision	Date	What changed
1.3	2026-06-24	Two new chapters in Part VI. One strategy, end to end: a capstone (see “Capstone End To End”) walks the running `bond→gold` example through all 20 pipeline stages in order, as a build log, surfacing the cross-stage seam bugs (the bracket reject, the silent `fx=1.0`, the leg sized by the *signal* price instead of the *traded* price) that survive every individual chapter. The Learn loop (see “The Learn Loop”) builds the *Operate → Learn → retire* arrow the lifecycle diagram has always promised: detecting a decayed edge (re-running the verdict function on live evidence; the drift-band throttle vs the structural-break detector) and a pre-committed retirement rule, honest about which rungs are armed, inert, or still wiring.

Revision	Date	What changed
1.2	2026-06-22	Added a war-story about a safety counter that silently disarmed itself on every restart. In the failure-mode catalogue (see “Failure Mode Catalogue”): *the guard that reset on every restart*: a minimum-hold counter and a drawdown circuit-breaker that lived only in memory, so a frequently-restarting system silently disarmed both (a position exited a bar early; a breaker cleared mid-drawdown and re-levered). Plus, in layered safety (see “Layered Safety”), the *closing-the-loop* sequel to the fresh-but-wrong drift band: rebuilding the band for the bundle you switched to, with a parameter-parity lock and a plausibility gate on the output.

Revision	Date	What changed
1.1	2026-06-11	Added war-stories about safeguards that agreed with each other while both were wrong, code that had never run until it crashed live, and a "close" that closed nothing. From a full adversarial re-audit: *mutually-reinforcing-wrong safeguards* (two checks validating each other against a phantom roster), *dormant code is unvalidated code* (a strategy whose first live act was a latent crash), and *the close that closed nothing* (strategy-scoped flatten no-oping on framework-reconciled positions) in the failure-mode catalogue (see "Failure Mode Catalogue"); plus, in layered safety (see "Layered Safety"), a fresh-but-wrong drift band (provenance + fail-safe), an out-of-process backstop whose halt never reached the node, and the kill switch that didn't announce itself.
1.0	2026-06	Initial edition: Parts I–VII.

1. Why most retail quant systems fail

Most retail quant systems do not fail in the market. They fail before they ever risk a dollar, in the backtest, in the way the numbers were computed, in the missing layer nobody built. The market just sends the invoice later.

This is the uncomfortable thing to internalise before anything else in this book: a strategy that "works" on your laptop is the *default* outcome, not the achievement. Give a motivated developer a few months, a data feed, and an optimiser, and they will produce a beautiful equity curve almost every time. The equity curve is cheap. What's expensive (and rare) is knowing whether it means anything, and surviving the gap between the curve and the live account.

We have built that gap into a system we call **Titan** (a NautilusTrader-based, multi-strategy stack trading through a retail broker), and we have fallen into all four of the holes below at least once. This chapter is the map of those holes. Each later part of the book is the rope ladder out of one of them.

The four ways a system dies

There are really only four failure modes that account for almost every dead retail quant system. They compound, and they share one nasty property: **each one makes the strategy look *better* than it is.** That asymmetry is the spine of the whole book: errors in this domain are not random noise around the truth, they are biased toward optimism, because the optimistic mistakes are the ones that survive your attention and get funded.

#	Failure mode	What it feels like	Where it actually bites
1	**Overfitting & multiple testing**	"I found a Sharpe of 2."	The edge was selected by luck across many trials; it evaporates live.
2	**Look-ahead bias**	"The backtest is flawless."	The strategy quietly used information it could not have had in time.
3	**No risk layer / no survival math**	"Great returns, why hedge?"	One drawdown or one sizing error ends the account before the edge pays off.

#	Failure mode	What it feels like	Where it actually bites
4	**No operations**	"It runs on my machine."	The order rejects, the container dies, the cost model lied; and nobody notices.

All four are ways the *edge itself* is fake or unsurvivable: engineering failures the builder controls before the market gets a vote. They are deliberately *not* the only ways to lose money: the **alpha-side** killers (capacity limits, crowding into the same trade, and slow alpha decay) are real, but they are problems you only earn once you have a genuine edge to erode, and they get their own treatment in Capacity & crowding (see "Capacity And Crowding"). This chapter is about the four that kill you *first*.

Let's take them in order, with a concrete (sanitised) Titan example for each, and the rule each one bought.

Failure mode 1: Overfitting and the multiple-testing trap

The principle: **if you try enough strategies, parameters, or instruments, one of them will look brilliant by pure chance; and your search process guarantees you will pick exactly that one.** This is not a subtle statistical footnote; it is the single largest source of fake edge in retail quant. The more cells in your grid, the higher the *expected maximum* Sharpe even when every strategy is genuinely worthless. A backtest reported as a point estimate, without accounting for how many things you tried to find it, is not evidence. It's the winner of a lottery you forgot you were running.

There are two tells, and Titan gates on both:

- **Plateau, not spike.** A real edge is robust to small parameter changes: a lookback of `L` and `L±1` should give similar results, say no neighbour more than ~20% below the peak Sharpe (illustrative; the real tolerance is set in Beating your own optimiser (see "Deflated Sharpe")). If the Sharpe collapses the moment you nudge a parameter, you have found a knife-edge in the noise, not a plateau in the signal.
- **The lower bound, not the point estimate.** A bootstrap confidence interval on the Sharpe tells you how bad the strategy could plausibly be. If the 95% lower bound is at or below zero, it's a coin flip with a good story: `unconfirmed`, full stop, regardless of how shiny the point estimate is. (And not just *any* bootstrap: an IID resample destroys the

serial correlation trend and carry strategies live on and biases that lower bound *upward*, the one caveat that makes the gate correct. Use a stationary **block** bootstrap, with a block length on the order of weeks, not days, so each resample preserves the autocorrelation the strategy lives on (illustrative); the mechanism is in a backtest you can trust (see "Backtest You Can Trust").)

Warning: War-story: the +2.0 Sharpe that was a single regime
A trend-following ensemble produced a backtest Sharpe near **+2.0** on a narrow, recent window: a few years on one asset class. It even *passed* the plateau check on that window: nudging the parameters barely moved the result. It looked like a genuine, robust edge.
Then we re-ran the identical specification on a much broader universe and a ~20-year history. The Sharpe **flipped negative.** The "+2.0" was not a trend edge at all; it was one favourable sub-regime, sampled with too few independent folds for the confidence interval to notice. The lower bound on the narrow window had been hovering just below zero the whole time, the framework was already refusing to promote it, but the point estimate was seductive enough that, without the gate, we'd have deployed it.
The rule it bought: **a high Sharpe on a narrow universe and short window must be re-validated on a broader, longer sample before promotion.** Tightening the confidence interval with more data is *necessary but not sufficient*; a sign reversal on broader data is decisive evidence the edge was a regime artefact. The structured form of this re-validation is walk-forward / out-of-sample testing, where the strategy is only ever judged on data the search never touched (see Walk-forward validation (see "Walk Forward")).

The deeper trap is in the *counting*. When you deflate a Sharpe for multiple testing, the number of trials `N` is the size of the *whole search pool*, not the handful of survivors you're proud of. Survivors-only understates the variance of your search and makes the deflated number look better than it is. The honest `N`, the full pool, is exactly the one that kills marginal strategies. That's the point.

The universe itself can be overfit before any parameter is. A backtest run on *today's* index constituents has silently dropped every name that delisted, merged, or fell out of the index along the way, so it only ever traded the survivors: the same upward bias as multiple testing, hidden in the data rather than the search. The discipline is to use a point-in-time universe that includes the dead tickers, so the strategy is tested against the losers it would actually have held; the backtest you can trust (see "Backtest You Can Trust") chapter treats this alongside the leakage discussion.

→ **The fix lives in Beating your own optimiser (see "Deflated Sharpe")**: the Deflated Sharpe Ratio, plateau-vs-spike testing, and why `N` is the pool and not the podium.

Failure mode 2: Look-ahead bias, the silent default

The principle: **a backtest that uses information from the future is not a backtest, it's a memory of the answer.** And look-ahead is the *default* state of careless code, not an exotic edge case. You have to actively work to keep it out; it sneaks back in every time you write a line that touches a return.

The canonical shape: a position is *decided* using bar `t`, then "earns" bar `t`'s return. A decision and the return it earns must live in disjoint time windows, decision first. In code, the only safe pattern lags the decision before it multiplies a return:

```
# A position decided at the close of t can only earn t -> t+1:
strat_returns = asset_returns * position.shift(1).fillna(0.0)
```

That `.shift(1)` is the *research-time* face of look-ahead, and it is worth saying plainly that an event-driven live stack does **not** make you immune: in a live `on_bar` handler you literally cannot see a future bar, but the same bug re-appears as *repainting* indicators, warm-up that peeks, or acting on bar *t*'s close before the bar is confirmed closed. The discipline transfers; the *shape* changes. (This research↔live gap is the whole subject of making live == research (see "Live Equals Research").)

The unsafe version (`position * same_bar_return`, with no shift) reads like perfectly ordinary pandas, which is exactly why it survives code review. Because most signals are autocorrelated, it manufactures a *plausible* fake equity curve. That's the worst kind: an obviously broken curve gets caught, a beautiful one gets deployed.

Danger: War-story: four leaks in one codebase, and a regime model that read the future
One audit of a regime-driven codebase found the same-bar leak, a signal multiplied by a contemporaneous return, in **four separate places**, none flagged in review; collectively they turned a flat strategy stellar (full dissection in A backtest you can trust (see "Backtest You Can Trust")).
The subtler cousin showed up later in a regime model. We labelled market regimes with a hidden Markov model and gated the strategy on the regime. The training was clean: no parameter leakage. But the *decoding* used the standard Viterbi algorithm, which is forward-

backward: the regime it assigns to bar `t` depends on observations both before **and after** `t`. A causality smoke test (corrupt the future, assert the past is unchanged) caught it instantly: corrupting prices *after* date `T` changed the gate *before* `T`. The fix was to replace the smoother with a strictly causal forward filter that only ever looks at data up to `t`.
The rules these bought: **any series that multiplies a return must be `.shift(1)`'d unless you can prove the position was knowable strictly before the return's window opened: guilty until proven innocent.** And **state-space models in a backtest must use the filter, never the smoother.**

Look-ahead has more disguises than the same-bar collect. Normalising a feature with the full series' mean and standard deviation leaks the future into every historical bar. Forward-filling a higher-timeframe signal onto lower-timeframe bars back-dates information that hadn't arrived. The defence is structural, and it's a recurring theme of this book: **make the dangerous operation impossible to call.** Titan's metrics module simply does not offer a full-series z-score, only causal and in-sample-frozen versions exist, and every cross-timeframe merge routes through a primitive that shifts before it fills, wrapped in a causality assertion.

→ **The fix lives in A backtest you can trust (see "Backtest You Can Trust")**: shift discipline, causal normalisation, the corrupt-the-future smoke test, and centralising metrics so the unsafe call can't be written.

Failure mode 3: No risk layer, no survival math

The principle: **a strategy's expected return is irrelevant if a drawdown kills you before the expectation arrives.** Most retail systems optimise the wrong object entirely. They maximise Sharpe, return per unit of volatility, and report it as if it were the verdict. But Sharpe is blind to the two things that actually end accounts: the *path* of the drawdown, and the *tail* of the loss.

This is where the metric suite earns its keep. Sharpe treats an upside spike and a downside crater identically. It says nothing about how deep the worst peak-to-trough trench is, or how long you sit underwater. And it averages over the very tail that ruins you. The numbers that speak to *survival* are different ones:

- **Sortino**: penalises downside deviation only.
- **Calmar**: CAGR over max drawdown, return *per unit of pain*.
- **CVaR / CDaR**: the average loss in the worst slice, not a single quantile.

- **Risk of ruin** and **Kelly / fractional Kelly**: the probability of hitting zero at your deployed size, and the sizing that keeps that probability acceptable.

Warning: War-story: the 1.4-Sharpe strategy nobody could hold
A candidate posted a Sharpe near 1.4 and cleared every statistical gate, yet traced a peak-to-trough drawdown deep enough, and long enough, that no committee would have funded it through the trough. It bought the rule that **Calmar lift, not Sharpe lift, is the primary promotion metric**; full autopsy in Beyond Sharpe: the metric suite (see "Metric Suite").

The risk layer is not just a metric; it's *machinery* that has to behave correctly under stress, and that machinery has its own bugs.

Danger: War-story: the defensive switch that was disabled for the entire crisis
A defensive strategy was supposed to rotate out of a risk asset into a safe one when the risk asset's trailing return weakened, a "don't switch unless the challenger beats the incumbent by a buffer" rule. The bug: it compared the challenger against a **frozen snapshot** of the incumbent's return taken at the last switch, instead of the incumbent's *current* value at each decision.
On real data, the snapshot latched a strongly positive return from late in a bull market. Through the following crisis, the safe asset's actual return never re-cleared that stale high-water mark, so the defensive switch *never fired*. The strategy rode the risk asset all the way down through the worst of the drawdown. The protective logic existed, passed its unit tests, and was effectively turned off precisely when it mattered.
The rule: **a stateful buffer must compare against the incumbent's CURRENT value, never a snapshot: cache only the *identity* of the incumbent, look up its return fresh every bar.** And the regression test that proves it: build a synthetic bull-to-bear path and assert the switch actually fires.

There's a final, structural lesson here about *how you measure* tail risk. A naive Monte Carlo gate, "reject if P(drawdown > some threshold `D`) is too high" (say `D` deep in the double digits, illustrative), is wrong for a long-only equity strategy, because block-bootstrapping 20 years of real crisis bars makes the *underlying itself* fail that gate. The right question is **relative**: does the strategy's risk layer reduce drawdown *versus simply holding the underlying*, on the same path? An absolute gate rejects every honest defensive strategy; a relative gate gives the true verdict.

→ **The fix lives in The portfolio risk manager (see "Portfolio Risk Manager")** (and its neighbours on sizing and tail risk): a real risk layer, survival math at deployed size, and gates that ask the economically correct question.

Failure mode 4: No operations

The principle: **a backtest is research; live trading is a process, and the process fails in ways the research never models.** This mode gets the least respect and causes the most quiet damage, because it doesn't corrupt a number on a slide. It rejects an order, drops a fill, mis-sizes a leg, or silently overwrites your data, and your research never gets a chance to be right or wrong.

Operations fails in three places, each with its own detection mechanism: the **order path** (caught only by verifying live fills), the **cost model** (caught only by auditing against real fills), and the **data path** (caught only by guards and checksums). Deployment reproducibility wraps all three. The war-stories below hang off those three buckets in turn.

The gap between "the backtest is green" and "the system is live and correct" is mostly unglamorous plumbing: order types the broker actually accepts, cost models that match real fills, data pipelines that don't clobber themselves, and deployment that's reproducible rather than "it works on my laptop."

Danger: War-story: the order that rejected itself, silently, every time
A live strategy submitted bracket orders (entry plus take-profit plus stop-loss). Every single trade failed, silently. Two operational bugs stacked. First, a parameter was passed under the wrong name; the order-construction call raised a `TypeError` and the trade never left the building. Second, the take-profit limit leg was being submitted *post-only*, which the broker rejected, and that rejection **cascaded to reject the entry order too.**
Nothing in the research caught this, because research doesn't talk to a broker's order-acceptance rules. The system *appeared* to be running. It was placing zero trades. The fix was specific and unglamorous (the correct parameter name, and `post_only=False` on the take-profit leg), and it could only ever have been found by running against a real (or realistic) broker connection and *verifying fills*, not by re-reading the strategy.

The operational failure mode also poisons research in a way that's easy to miss: **your cost model is part of your edge calculation, and it's almost**

always too optimistic. A flat "X basis points per turnover" model underprices reality whenever notional is small, the broker charges a per-fill commission *floor*, a continuous-futures leg pays for mandatory quarterly rolls, or a vol-target overlay generates many small daily tweaks. The shape of the bug (illustrative magnitudes): a config carried a single low per-turnover rate, while a cost audit against real fills found round-trip costs several times higher, because the model ignored the per-fill commission floor at small notional and applied one rate to legs that should have had several. An edge of a few tens of basis points can be entirely consumed by that gap, so a cost model is a research artefact that must be *calibrated against actual fills*, not assumed.

And commission is only half the cost model. The other half is **slippage**, the gap between the price you *decided* at and the price you actually *filled* at, which a backtest that fills at the bar close ignores entirely. Slippage scales with size and with the thinness of the book, so a strategy that looks fine at research notional can be eaten alive at deployed size; like the commission floor, the only honest calibration is measured fills, not an assumed spread.

And then there are the data hazards. A downloader that writes `SYMBOL_TF.parquet` will happily overwrite a long-history file with a shorter one from a different source. Two vendors timestamp daily bars at different times of day, so a naive merge produces an all-NaN column and a regime gate that *never fires*, and nothing crashes to tell you. These are operations problems, and they are why deployment, containerization, and a disciplined paper-to-live path are a full part of this book and not an afterthought.

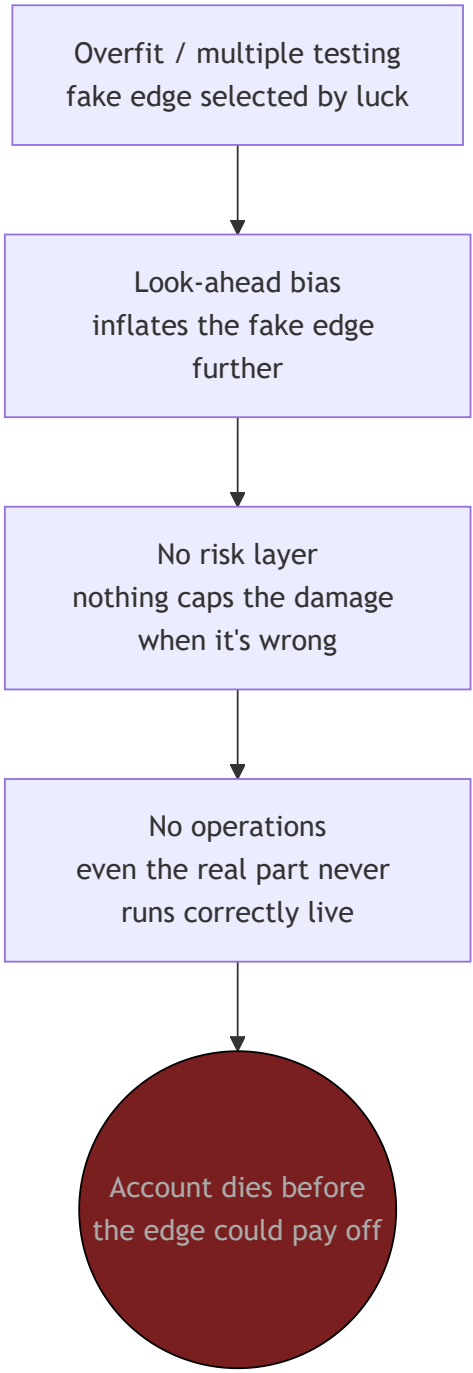
→ **The fix lives in Containerising the stack (see "Containerizing")** (and the live-runbook and paper-to-live chapters): reproducible deployment, realistic cost models, data-overwrite guards, and verifying live behaviour against research instead of assuming it.

Note: A fifth hole, off the modelling axis: the regulatory tripwire
The four modes above are all ways the *edge* is fake or unsurvivable. There is a fifth account-ender with nothing to do with whether your strategy is any good: an operational-legal **bright line** you cross without noticing. A sub-threshold account that day-trades a few times too often gets frozen by the **pattern-day-trader** rule; trading a friend's money can quietly make you an unregistered adviser; the wrong entity or wrapper bakes in liability and tax you can't easily undo. No risk layer in this book predicts a regulatory freeze, because it isn't a market event; it sits on the same shelf as "no operations," the unglamorous plumbing that ends accounts. This book gives no legal advice; it only insists you learn where the lines are *before* you trip one. The map is the

Legal & regulatory bucket of the paper-to-live checklist (see “Paper To Live”).

How the four compound

These failure modes are not independent; they feed each other. The sketch below is *one representative cascade*, not a fixed order; the arrows show how they *can* compound, not a dependency you must defend in sequence. Any mode fires on its own: look-ahead needs no overfitting first, and an ops bug (a too-optimistic cost model) can *manufacture* fake edge upstream of everything. That is exactly why the defence has to be layered rather than prioritised front-to-back:


Overfit / multiple testing
fake edge selected by luck
Look-ahead bias
inflates the fake edge
further
No risk layer
nothing caps the damage
when it's wrong
No operations
even the real part never
runs correctly live
Account dies before
the edge could pay off

A strategy can clear every statistical gate and still die operationally; it can be operationally flawless and still deploy a look-ahead phantom. The defence has to be layered, because the failures are.

Notice what every war-story above has in common: none was caught by re-reading the code. Each fell to an *independent* check (a broader sample, a corrupt-the-future assertion, a fill verification, a synthetic bull-to-bear regression) that asked the question the author's eyes had already skipped. That is why this book leans on external gates and mechanical tests rather than careful self-review: you cannot proofread your way out of a bias you cannot see.

Where each fix lives in this book

This chapter is a promissory note. Every failure mode above is paid off in detail later:

Failure mode	The discipline that fixes it	Chapter
Overfitting & multiple testing	Deflated Sharpe, plateau-vs-spike, full-pool `N`	Beating your own optimiser (see "Deflated Sharpe")
Look-ahead bias	Shift discipline, causal normalisation, corrupt-the-future test	A backtest you can trust (see "Backtest You Can Trust")
No risk layer / no survival math	Calmar-first promotion, relative MC, ruin & Kelly	The portfolio risk manager (see "Portfolio Risk Manager")
No operations	Reproducible deploy, real cost models, data guards	Containerising the stack (see "Containerizing")

The single mental model to carry forward: **every number you produce is biased toward optimism until you've proven otherwise.** Suspicion over celebration. A green backtest is a claim to be stress-tested, never a result to be banked.

> **Tip: The four-gate spine, in one place (you'll build each later)**
> The whole book reduces to one promotion gate, one rung per failure mode. Pin it above the desk; the chapters that follow are how you *implement* each line:
>
> 1. **Overfitting** → the full-pool Deflated Sharpe is positive **and** the winner sits on a plateau, not a spike. (Beating your own opti-

miser (see “Deflated Sharpe”))
2. **Look-ahead** → the corrupt-the-future causality test passes, end to end. (A backtest you can trust (see “Backtest You Can Trust”))
3. **No risk layer** → Calmar-relative-to-the-underlying is positive and risk-of-ruin at deployed size is acceptable. (The portfolio risk manager (see “Portfolio Risk Manager”))
4. **No operations** → fills are verified against a real (or realistic) broker, not assumed from the blotter. (Containerising the stack (see “Containerizing”))

A blank on any line caps the verdict at `unconfirmed`: the result does not exist yet. These are *gates, not advice*; the formal decision function that consumes them is the pre-flight checklist (see “Preflight Checklist”).

Exercises

1. **The shared property.** The four failure modes share one property that is the reason for the book’s entire posture. Name it, and explain why it implies gating on a Sharpe’s *lower bound* rather than its point estimate.

   Answer: Answer
   Each failure mode **flatters** the strategy: the errors are biased toward optimism, not random, because the optimistic versions survive your attention and get funded. So any un-audited number is inflated until proven otherwise, and the honest figure to bet on is the conservative lower bound of the bootstrap interval, not the seductive point estimate.

2. **The fifth hole.** A friend asks you to trade their savings alongside your own. Which account-ending failure does this risk, and why does no risk layer in the book predict it?

   Answer: Answer
   The **regulatory tripwire** (the fifth hole, off the modelling axis): managing someone else’s capital can make you an unregistered adviser. No drawdown or sizing control predicts it because it is not a market event. It is a legal bright line, mapped in the paper-to-live legal bucket (see “Paper To Live”).

Takeaways

- Retail quant systems mostly die *before* the market: in the search, the math, the missing layer, or the plumbing. The four modes compound and each one **flatters** the strategy.
- **Overfitting** is the default when you search hard; gate on the *plateau* and the *lower bound*, and count the *full* trial pool.
- **Look-ahead** is the default of careless code; lag every decision before it earns a return, normalise causally, and make the unsafe operation impossible to call.
- **No risk layer** kills accounts that had a real edge; optimise Calmar and survival, not just Sharpe, and verify the protective logic actually fires.
- **No operations** wastes good research on rejected orders, wrong cost models, and silent data corruption; treat deployment as a first-class engineering problem.
- Carry one rule above all: *every number is optimistic until proven otherwise.*

The rest of Part I builds the foundation that makes those fixes possible: The architecture of a survivable stack (see "Architecture"), Stack choices (see "Stack Choices"), and Project layout (see "Project Layout"). Then Part II turns the research itself into something you can trust, starting with the typology of strategies (see "Typology") and a backtest you can trust (see "Backtest You Can Trust").

2. Architecture & the one-way dependency rule

A trading system is not one program. It is at least four, with very different lifespans and very different blast radii. There is the throwaway notebook where you discover an edge. There is the file that says, in plain text, *what you intend to deploy*. There is the tested library that actually does the trading. And there is the thin script that wires the library to a broker and turns it on.

Most systems begin without these seams. Someone discovers a signal in a notebook, the notebook grows a broker connection, the broker connection grows a position-sizer, and one Tuesday the notebook is live with real money, carrying every exploratory shortcut and every hard-coded threshold straight into production. The first chapter, Why systems fail (see "Why Systems Fail"), is largely a tour of what happens next.

This chapter is about the one structural decision that prevents most of it: a **layered architecture where dependencies flow in exactly one direction.** Discovery code may depend on the library; the library may never depend on discovery code. Entry points may depend on the library; the library may never contain an entry point. That single arrow, drawn once and enforced forever, is what makes look-ahead, coupling, and "it works on my machine" *visible* instead of latent.

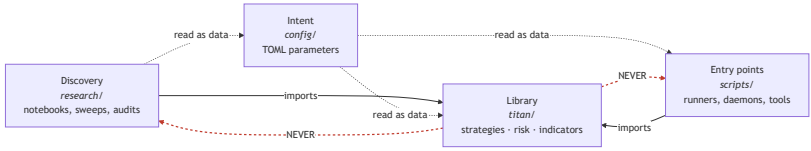
The principle: layers, and an arrow that only points one way

Sort everything you write into four layers by *how long it lives* and *what it is allowed to assume*.

Layer	Lifespan	Job	May import
Discovery	Disposable	Find an edge; iterate fast; be wrong cheaply	The library
Intent	Versioned config	State *what* runs and with which parameters	Nothing (it's data)
Library	The asset	The tested, importable implementation	Only itself + third-party deps
Entry points	Operational glue	Wire the library to a broker/clock and start it	The library + config

The rule is that **the library sits in the middle and depends on nothing above it.** Discovery and entry points both point *inward*, at the library. The

library never points back out. Drawn as a graph, the whole system has to be acyclic with the library as a sink:



Solid arrows are imports: a code dependency. Dotted arrows are read-as-data: config parsed at runtime, not imported, which is why intent can flow into* the library without violating the arrow. The two red edges are the forbidden inversions the rule exists to prevent.*

Why is the *direction* load-bearing, and not just the existence of folders? Three reasons, and each one is a class of bug it prevents.

1. It localises blast radius. A dependency arrow is a contract: "if you change me, you might break the things that point at me." Because the library points at nothing above it, you can rewrite a research notebook, delete a sweep, or rename a config key without any risk of breaking the code that trades. The expensive, well-tested asset is downstream of nothing volatile. Invert one arrow, let the library `import` a research helper, and now an exploratory file is on the critical path to live capital. The most reckless code in the repo is suddenly load-bearing.

2. It makes look-ahead structurally harder to ship. This is the subtle one. Discovery code is *allowed* to see the whole dataset at once: that is what makes it fast and what makes it dangerous. A z-score over the full series, a "winner" chosen with the same bar it earns, a normalisation fit on data that includes the future: these are the five lies (see "Backtest You Can Trust") of an untrustworthy backtest, and they all share one tell. **They assume access to information the live system will not have.** The library lives in the world of *one bar at a time, decisions before returns.* When the boundary is one-directional, look-ahead can't quietly leak from the permissive layer into the strict one, because the strict layer cannot import from the permissive one. The compiler, or the import, is doing your code review.

3. It separates *intent* from *implementation*. The parameters that constitute your edge, the lookbacks, thresholds, the instrument list, belong in versioned config, read as *data*, never baked into the code as literals. A reviewer can diff `config/` and see exactly what changed about a deployment without reading a line of Python. The library asks "what am I told to do?"; the config answers; the entry point delivers the answer.

> **Tip: The layer test: 'what is allowed to assume what?'**
> When you're unsure where a file belongs, don't ask "what does it do"; ask **"what is it allowed to assume?"** Code that may assume it sees the future is discovery. Code that may assume a broker connection exists is an entry point. Code that may assume *neither*, only its inputs and its own tests, is the library. Anything that wants to assume *both* is two files wearing a trenchcoat.

The Titan example: `research → config → titan → scripts`

Titan implements exactly these four layers as four top-level directories, and the dependency arrow runs strictly left to right.

```
research/   discovery   - sweeps, audits, one-off explorations
 ↪  (disposable)
config/     intent      - *.toml: which strategies, which parameters
 ↪  (data)
titan/      library     - the packaged, tested implementation (the asset)
scripts/    entry points - runners, daemons, operational CLIs (glue)
```

The split is not cosmetic; it is encoded where it can be enforced.

Only the library is packaged. `pyproject.toml` declares `packages = ["titan"]`. `research/` and `scripts/` are *not* installable artifacts; they are siblings on the path, not part of the shipped library. The thing you `pip install` and import is precisely the layer that is allowed to touch capital.

The two outer layers are held to a different bar. The linter config is blunt about the asymmetry:

```toml
[tool.ruff.lint.per-file-ignores]
# Standalone scripts: allow sys.path shims + long lines, skip docstrings
"scripts/*"     = ["E402", "E501", "D"]
# Research: relaxed (line length, semicolons, terse var names)
"research/**/*" = ["E501", "E702", "E741", "D", "E402"]
```

This excerpt is trimmed for the page; the real block has more rows (`tests/*`, `execution/*.py`, `resources/**/*`) all granting the same disposable-layer relaxations. `titan/` gets essentially none of these passes. The one exception is `"titan/strategies/**/*" = ["D"]`, which skips docstring rules. Tellingly, that lint-relaxed sub-layer is also exactly where the back-edges cluster (next section): the place held to the lowest bar is the place the arrow bends. Discovery and glue are allowed to be

a little sloppy because they're disposable; the library is held to the strict standard because it's the asset. The ruleset *names* the layers and treats them differently; the architecture is literally in the lint config.

Intent flows in as data, not code. The process-wide risk manager doesn't hard-code its limits; it reads them from `config/risk.toml` at construction:

```
# titan/risk/portfolio_risk_manager.py  (sanitised)
import tomllib
from pathlib import Path

risk_toml = Path(__file__).resolve().parents[2] / "config" / "risk.toml"
if risk_toml.exists():
    with open(risk_toml, "rb") as f:
        cfg = tomllib.load(f)            # parameters arrive as data
    portfolio_risk_manager.load_config(cfg["portfolio"])
```

The drawdown kill-switch threshold, the volatility target, the correlation dial's bounds: all of it lives in a file you can diff, review, and roll back, separate from the FSM that *acts* on it. Changing the deployed risk posture is a config edit plus a restart, not a code change. (For why those numbers are redacted here and what they govern, see the portfolio risk manager (see "Portfolio Risk Manager").)

> **Warning: Intent-as-data needs a turnstile, not just a parser**
> Moving parameters out of code trades a compiler-caught *code* typo for a silent *config* typo no compiler will ever see. A `vol_target = 2.0` where you meant `0.02`, or a key spelled `drawdown_thresold`, parses as perfectly valid TOML and reaches the FSM unchallenged. So the load path has to *validate*, not merely `load`: `load_config` bounds-checks each field (vol target in `(0, cap]`, drawdown threshold negative, the correlation dial inside its documented range) and **raises on an out-of-range value or an unknown key** rather than letting it size a position. The validation cost is the price of the move to TOML; on a system that money-sizes off these numbers, it is not optional.

The entry point is where, and only where, the world gets wired up. `scripts/run_portfolio.py` is a real entry point in the textbook sense: it has a `main()`, an `argparse` parser, an `if __name__ == "__main__":` guard, and it is the single place that constructs the broker-connected `TradingNode`. Nothing in `titan/` does that. The library defines *strategies and risk logic*; the script *selects a bundle, builds the node, and presses go.* That is the difference between a library and a program, and Titan keeps it physical: an entry point is a file in `scripts/`, never a function in `titan/`.

Note: Why entry points must never live in the library
The moment an importable module has top-level "connect and trade" side effects, every test, every research script, and every other importer that so much as `imports` the package risks reaching out to a broker. Entry points must be *inert until called* and must live where nothing imports them. In Titan, importing `titan` builds the risk-manager singleton from config but opens **zero** broker connections; you have to run a `scripts/` file to talk to IBKR. Import-time purity in the library is what lets the test suite, and a thousand research scripts, import it freely.
`tests/` is the fifth inward-pointing consumer the four-layer table doesn't draw, and the primary beneficiary of that import-time purity (the concrete thing "a thousand research scripts" stands in for). Like `research/` and `scripts/` it imports `titan` freely and points inward; the library never imports it. It sits *off* the arrow, below the flow, not on it.

The payoff shows up at every layer boundary. A researcher in `research/` imports `titan` to reuse the *exact* sizing and metrics code the live system uses, so the backtest and the deployment share an implementation rather than two subtly different copies (the whole subject of Live = research (see "Live Equals Research")). The core of the library, risk, indicators, costs, data validation, FX/equity plumbing, imports nothing from `research/` or `scripts/` at all. We checked: the `titan/risk/`, `titan/utils/`, `titan/indicators/`, `titan/costs/`, and `titan/data/` packages have zero back-edges to the outer layers. The arrow holds where it matters most.

When the arrow bends: the honest wrinkle

Here is the part most architecture chapters omit. Titan's arrow is not perfectly clean, and pretending otherwise would teach you the wrong lesson.

A handful of *strategy* modules inside the library import from `research/` (the module name below is illustrative: a generic stand-in for a real strategy):

```
# titan/strategies/<strategy>/strategy.py
from research.<strategy>.signal import StrategyConfig    # library ->
 ↪  research (!)
```

About a dozen `titan/strategies/*` files do some version of this, pulling a config dataclass or a signal function out of the research package that first validated it. This is a genuine back-edge: a library module depending on a discovery module, the exact arrow the rule forbids. And it carries a real operational cost: the disposable layer has acquired a small claim on the critical path, precisely the failure mode the rule exists to prevent, in miniature. The war-story below is what that cost looks like at deploy time.

Warning: War-story: the back-edge that pulled the research tree into the live image
A strategy was promoted to live by having its library wrapper `import` the original signal function straight from the research package, instead of *moving* that function down into the library. It worked, and it passed review, because the import resolved and the tests were green. The cost surfaced at deploy time: the production image would no longer build from `titan/` alone; it had to bundle the entire `research/` tree, and a build-time import check on those classes meant an unrelated edit to an exploratory research file could now break the production build. The shape of the bug is *coupling that points the wrong way*: the asset taking a dependency on the disposable. The fix is mechanical: promote the needed code *down* into `titan/`, leave a thin compatibility shim in research if you must, and let the back-edge die. The rule that bought: **promotion means moving code across the boundary, not importing across it.** An `import research` inside `titan/strategies/` is a TODO, not a design.

The lesson is not "the rule is impractical." It's the opposite: the rule's *value* is exactly that it makes this kind of debt **legible**. One `grep` for `from research` inside `titan/` produces a precise, countable list of the architecture's compromises. You can put a number on your coupling and watch it trend. A system without the rule has the same coupling; it just has no way to *see* it, because everything imports everything and no arrow was ever supposed to point any particular way.

Danger: Three folders that all describe the live strategy, and only one is true
A subtler boundary failure is *intent* fragmenting across files that disagree. In Titan, the running strategy bundle is named in three places: the `Dockerfile`'s baked default, the Compose file's fallback, and an environment file, and they name **three different bundles.** Only the environment file's value actually wins at runtime. Anyone reading just the Dockerfile, or just the Compose file, will confidently misidentify what is live; and on a system that moves real money, "I thought we were running X" is how you fail to flatten the thing that's actually open. The rule it bought: **intent must have exactly one source of truth, and the resolution order must be documented at the point of ambiguity.** Defaults scattered across layers are not redundancy; they are three chances to be wrong. We return to this in the live runbook (see "Live Runbook").

How the boundary makes coupling and look-ahead visible

The reason to pay the ceremony cost of four directories is that the boundary turns invisible problems into greppable ones.

- **Look-ahead.** Discovery code can fit a normaliser on the whole series; the library can't import that code, so the leak cannot ride an import into production. If a strategy *needs* a normalisation, it has to be (re)implemented causally in `titan/`, and now it's reviewable as a library change, with tests, instead of inherited silently from a notebook. The boundary doesn't *detect* look-ahead, but it removes the easiest path for it to travel. Be precise about what that buys: the boundary closes *one* path (leakage riding an import from the permissive layer into the strict one), not all of them. The causal-but-wrong normaliser someone *does* write inside `titan/` is caught downstream, by walk-forward (see "Walk Forward") and the survival-metric suite (see "Metric Suite"), not by the import graph. Layers and validation each close a different door; having layers does not make you safe from leakage on its own.
- **Coupling.** Every illegal dependency is a single grep. `from research` inside `titan/` is your coupling debt, enumerated. `from scripts` inside `titan/` should always be empty (it is); a library that imports an entry point has inverted itself completely.
- **Intent drift.** Because parameters live in `config/` as data, a deployment change is a config diff. Edge-bearing numbers never hide as literals three call-frames deep in the library, where no reviewer would think to look.

Example: A 30-second architecture audit
You can sanity-check a layered system's boundaries with four greps, no deep reading required:

```
grep -rl "from scripts"  titan/     # library importing an entry
 ↪  point: must be EMPTY
grep -rl "from research" titan/     # library coupling to discovery:
 ↪  minimize, track the count
grep -rl "from titan"    research/  # discovery reusing the asset:
 ↪  GOOD, expect many
grep -rl "from titan"    scripts/   # entry points composing the
 ↪  asset: GOOD, expect many
```

The first should be empty. The second is your debt ledger: a small number you can drive toward zero. The third and fourth should be *large*, because reuse-inward is the whole point: research and ops both

compose the same tested library rather than re-deriving it. If the third or fourth come back empty, you have copy-paste, not architecture: two implementations of your edge that will drift apart until live $\neq$ research. One caveat for a repo like Titan's, which *also* has a **packaged** `titan/research/` namespace (the promoted, tested research utilities, metrics and the framework that *are* the library): grep the exact `from research/import research` prefix, not a bare `research`. `from research.…` is the illicit top-level import you're hunting; `from titan.research.…` is a legitimate in-library import and must **not** be counted as a back-edge. The grep audit is only honest if it distinguishes the two; the same directory word means opposite trust levels at the two paths.

Tip: Make the grep a gate, not a habit
A grep you run by hand is a convention, and the war-story above is what happens when the honour system meets a deadline ("it passed review because the import resolved"). To make the arrow a *constraint*, wire it into CI: an **import-linter** contract that declares `titan` may not import `research` or `scripts` and **fails the build** on a violation, or (as a lighter ratchet) a test that asserts the back-edge count (`grep -rl 'from research' titan/ | wc -l`) is `<= N` for a budget `N` you only ever lower. Either way the build, not your memory, holds the boundary, and the "watch the count trend toward zero" the chapter promises becomes a number CI won't let rise.

Exercises

1. **The one-way rule.** State the one-way dependency rule. Which direction may the dependency arrow point between research and library code, and what does the *forbidden* direction let in?

   Answer: Answer
   Discovery and entry points point *inward* at the library; the library depends on nothing above it. The forbidden direction (library importing research) puts the most reckless, look-ahead-permissive code on the critical path to live capital and stops the library from running standalone.

2. **Enforcement, not etiquette.** Why is "everyone agrees to respect the dependency direction" insufficient, and what makes the rule real?

Answer: Answer
A convention nobody enforces drifts the first time a deadline bites. The rule is made real by a CI **import-linter** that fails the build on a backwards import; the compiler does your code review, so look-ahead can't leak from the permissive layer into the strict one.

Takeaways

- **Four layers, one arrow.** Discovery → intent → library → entry points. Dependencies point *inward* at the library; the library depends on nothing above it. Keep the graph acyclic with the library as the sink.
- **Entry points never live in the library.** Anything that connects to a broker or "starts" belongs in `scripts/`, inert until called. Importing the library must open zero connections, or your test suite and your research can't safely import it.
- **Intent is data, not code.** Edge-bearing parameters live in versioned `config/`, read at runtime. A deployment change should be a reviewable config diff, never a literal buried in the library.
- **The boundary makes problems greppable.** One grep finds every illegal dependency; the per-file lint asymmetry encodes which layer is the asset. The rule's real value is that it turns invisible coupling and look-ahead paths into a countable ledger.
- **Promotion means moving code *down*, not importing *up*.** An `import research` inside `titan/` is debt to be paid off, not a design to be defended; and the cost (here, dragging the whole research tree into the production image) is exactly why.

With the layers and the arrow in place, the next two chapters fill them in: Stack choices that won't fight you (see "Stack Choices") covers the tools each layer wants, and Project layout & non-negotiables (see "Project Layout") turns the boundary into coding rules (typed money, fixed seeds, one shared metrics module) that keep the library composable. The *contract* every strategy signs with the risk layer (the interface that makes a portfolio of strategies share one risk manager) is load-bearing enough to be its own chapter: the strategy-class & integration contract (see "Strategy Class Contract") in Part IV.

3. Stack choices & why

Every line of a quant stack is a bet on a trade-off, and the expensive bets are the ones you never noticed you were making. Pick a backtester for its speed and you may have quietly chosen a fill model that will never match your broker. Roll your own execution loop because the frameworks "felt heavy" and you have signed up to re-implement order-state reconciliation, the part that actually loses money when it's wrong. The technology choices in this chapter are not about features. They are about which failures you are willing to own and which you want the tooling to make impossible.

This chapter walks the choices behind Titan, a managed event-driven execution engine, a separate vectorised research layer, a pinned lockfile, and a container as the deployment boundary, and the reasoning that should drive *your* version of each. We name specific tools (NautilusTrader, vectorbt, `uv`, Docker) because concrete examples teach better than abstractions, but the lesson is the trade-off, not the brand.

The principle: buy the engine, build the edge

A trading stack has two halves with opposite economics.

The **edge**, your signals, your sizing, your risk policy, is where differentiation lives and where you must own every line. Nobody else can write it, and a subtle bug there silently corrupts your P&L.

The **plumbing** (order routing, fill simulation, position and cash accounting, reconnection, event scheduling) is undifferentiated and brutally hard to get right. It has no alpha in it. It is pure downside: invisible when correct, catastrophic when wrong, and the bugs surface only in live trading where they cost real money. This is exactly the code to *not* write yourself.

> **Tip: The build/buy heuristic for a quant stack**
> Build what encodes your edge. Buy (or adopt) what merely has to be *correct*, especially anything that talks to the broker or accounts for cash and positions. A managed engine has absorbed thousands of edge-case bug reports you would otherwise discover one margin call at a time.

The single most consequential decision is whether your **backtest and your live system run the same code**. If they don't, you are validating one program and deploying a different one, and the gap between them is precisely where look-ahead, fill-model fantasy, and timezone bugs (see "Time Sessions Calendars") hide. The whole of Live equals research (see "Live Equals

Research") is about defending that equality; the architecture choice here is what makes it *achievable* in the first place.

Choice 1: A managed event-driven engine vs DIY

A backtest can be written two ways, and the difference is not stylistic.

A **vectorised** backtest computes the entire signal and P&L series at once with array operations: `returns = asset_returns * position.shift(1)`. It is fast, seconds for years of data, which is what you want when sweeping parameters. But it models a frictionless world. There is no order that can be rejected, no partial fill, no bracket whose take-profit leg gets refused by the broker, no moment where your position and the broker's position disagree. It assumes you transact at the price you saw.

An **event-driven** engine processes one event at a time (bar, tick, order-accepted, fill, position-changed) exactly as they arrive in live trading. It is slower, but it models the things that actually break: an order sits in `SUBMITTED` until the venue accepts it; a fill arrives late; a reconnect finds a position you didn't know you held. Critically, a mature event engine lets the **same strategy class** drive both a historical backtest and a live session: you swap the data and execution clients, not the logic.

Dimension	Vectorised backtest	Event-driven engine
Speed	Very fast (whole-series array math)	Slower (event-by-event)
Fill realism	Idealised: you get the price you saw	Models acceptance, rejection, partials, latency
Order lifecycle	None: position is just a number	Full state machine (submitted → accepted → filled)
Backtest ↔ live parity	None: live is a *rewrite*	Same strategy code drives both
Best for	Parameter sweeps, idea triage	Final validation and production

The DIY temptation is real: an event loop "is just a `for` over bars." It is not. The hard part is the order-state machine and the accounting: reconnection that re-adopts external positions, cash ledgers that survive partial fills, idempotent handling of duplicate broker events. Get the accounting subtly wrong and your *backtest* still looks fine while your *live* book drifts. We chose to adopt that machinery rather than maintain it.

Example: Titan: one strategy class, two runtimes
Titan runs its production strategies on a managed event-driven engine (NautilusTrader). A `Strategy` subclass implements `on_start`, `on_bar`, and `on_position_closed`; the engine feeds it historical bars in a backtest and live broker bars in production through the *same* handlers. The strategy never knows which world it's in. The risk layer, the sizing, the entry logic, all written once, exercised in both. That parity is the point of adopting the engine; without it, "we tested this" and "we deployed this" describe two different programs.

There is a second, subtler win. A managed engine forces a **strategy-class contract**: a fixed shape every strategy must satisfy (lifecycle hooks, a sizing call, a risk check before any order). That contract is what lets a whole portfolio of strategies share one risk manager and one allocator, the entire subject of Part V. A DIY loop tends to grow per-strategy special cases until no two strategies can be reasoned about together. The contract is documented in The strategy-class contract (see "Strategy Class Contract").

Warning: Buying an engine isn't free: when to build, what it costs, how to pick
The case above is one-sided on purpose; honesty requires the other side. **When DIY wins:** a single-instrument, low-frequency strategy where the engine's learning curve and abstraction tax dwarf its benefit; an exotic asset class no framework models; or a shop that already owns battle-tested execution infrastructure. **The real price of "buy"** is not just "a framework to learn" and "slower": it's the *abstraction tax* (fighting the engine's data and clock model when your need doesn't fit its grain), *upgrade churn* on a fast-moving dependency, *debugging through someone else's event loop*, and *lock-in* if the project is abandoned. Price those before you adopt. **And since the lesson is the property, not the brand,** choose against a rubric, not a logo: does the engine run the *same strategy class* in backtest and live? does it model order rejection, partial fills, and latency? is its cash/position accounting auditable? is it actively maintained? Those four properties are what "buy the engine" actually buys; the name is incidental.

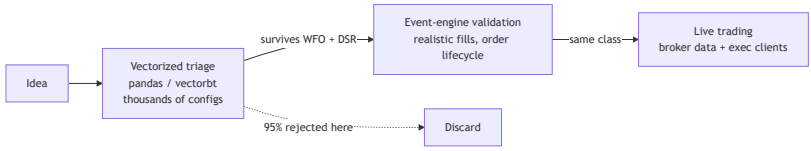
Choice 2: Where vectorised research stops and the engine begins

Adopting an event engine does *not* mean you run every experiment through it. You shouldn't. The two tools have different jobs, and the skill is knowing the handoff.

Titan keeps both, deliberately:

- **Research / triage layer**: `pandas`, `numpy`, `numba`, and vectorised backtesters (`vectorbt`, `backtesting`). This is where ideas are born and most of them die. You want to test thousands of parameter combinations in minutes, compute information coefficients across dozens of instruments, and throw out the 95% that don't survive. Fill realism doesn't matter yet, because a strategy that fails in a frictionless world will only fail harder with costs.
- **Validation / execution layer**: the event-driven engine. Once an idea survives triage, walk-forward, and deflation, it graduates to the engine, where realistic order handling either confirms the edge or reveals that it lived entirely in the idealised fills.

The two passes differ by orders of magnitude, and that asymmetry is the whole reason to keep both. Vectorised triage scores years of a single config in seconds, so a sweep of thousands finishes over coffee; the event engine, replaying every bar, order, and fill, takes minutes to hours per config (*illustrative*). At that ratio you cannot afford to run the engine on the 95% headed for the bin, and you cannot afford *not* to run it on the few percent that survive. So the handoff below is not just a discipline, it is a compute budget: spend cheap vector-seconds finding candidates, spend expensive engine-hours only on the survivors of WFO and deflation.



Warning: War-story: the edge that lived in the fills
A candidate strategy looked excellent in its vectorised backtest: a clean equity curve, a Sharpe well past every gate. When it moved to the event-driven layer with realistic order handling, much of the return evaporated. The vectorised version had been transacting at prices the strategy could never actually have gotten: it sized off a bar's close and implicitly *filled* at that same close, with no spread, no slippage, no rejected leg. The shape of the bug is generic and worth naming: **a vectorised backtest's "fill" is an assumption, not an execution.** The fix wasn't a parameter change; it was treating the vectorised result as a *triage* signal only, never a deployment verdict. Anything headed for capital must clear the engine that models how orders actually behave.

The boundary is a discipline, not a suggestion. Vectorised math is for *finding* candidates; the event engine is for *trusting* them; never let a fast, flattering, frictionless number stand in for a deployment decision. The measurement disciplines that make either number honest in the first place are in A backtest you can trust (see "Backtest You Can Trust").

Choice 3: Broker and data realities constrain the stack

Your strategy logic does not get a vote on what your broker will actually let you trade. The catalogue, the data subscription, and the account jurisdiction are hard constraints that ripple all the way back into research, and ignoring them is how a "validated" strategy turns out to be un-deployable.

Four constraints bite repeatedly:

- **Tradability $\neq$ availability of data.** You can often *stream* an instrument's market data while being *forbidden to trade it*. A signal can legitimately be computed from an instrument you can never hold.
- **Jurisdiction rewrites your universe.** Regulatory rules can block a whole class of instruments for an account, forcing substitutes that correlate with, but are not identical to, what you researched.
- **Currency is not cosmetic.** An instrument quoted in a different currency than your strategy's accounting base means every notional has to cross an FX rate. Get that conversion wrong and your *sizing* is wrong, silently.
- **Cost of carry can invalidate the catalogue.** An edge that clears every gross-return gate can still die once you charge realistic overnight financing, borrow, and spread. These are not a haircut you subtract at the end, they can flip the sign of a strategy that holds positions for days. In one cross-asset migration only a single underlying out of five survived a *cost-aware* walk-forward under live financing; the rest were deployable on paper only (*illustrative*). The cost model is a deployment constraint, not a footnote.

Example: Titan: PRIIPs forces a data-only signal and a substitute leg
Titan's account jurisdiction blocks *trading* certain US-listed ETFs, even though their market data still streams. So a cross-asset sleeve computes its signal from an instrument it cannot hold (data-only) and trades a UCITS substitute instead. The substitute had to be chosen not just for correlation but for *currency*: a USD-quoted line was selected over an otherwise-equivalent local-currency line specifically so the strategy's accounting base and the instrument's quote currency match, keeping the FX conversion factor at exactly 1.0. The broker's rulebook,

not the alpha, dictated the instrument list.

Danger: War-story: a leg mis-sized by a third on a currency assumption
A sizing path divided a base-currency notional by a price quoted in a *different* currency left at an implicit 1.0 – no exception, just a position mis-sized by the FX ratio (about a third, illustrative) on every trade, poisoning every downstream risk number. Broker realities are a correctness property of your sizing math, not a deployment afterthought – full autopsy in Per-strategy equity & FX (see "Per Strategy Equity Fx").

The deeper lesson: do the broker due diligence *before* you fall in love with a backtest, because the catalogue and the cost model can invalidate it. A demo account is not a reliable proxy for the live catalogue or live financing costs; verify against the real account dashboard. The full treatment of these traps is in Broker realities (see "Broker Realities"); how the FX conversion feeds into sizing is in Position sizing: Kelly & vol-targeting (see "Position Sizing Kelly"), and how it feeds back into accounting in Per-strategy equity & FX (see "Per Strategy Equity Fx").

Choice 4: Pinned lockfile: reproducibility is a risk control

A backtest is a measurement (see A backtest you can trust (see "Backtest You Can Trust")), and a measurement you cannot reproduce is an anecdote. If `pip install` today resolves different versions than it did last month, then "the backtest passed" is a claim about a software environment that no longer exists; and the environment your container builds for production may differ again.

The fix is a **lockfile**: pin the *exact* resolved version (and hash) of every transitive dependency, commit it, and build from it everywhere. `pyproject.toml` declares loose intent (`pandas>=2.2`); the lockfile records the precise graph that intent resolved to.

Tip: Loose constraints in the manifest, exact versions in the lockfile
Two files, two jobs. `pyproject.toml` says what you *want* (`numpy>=1.26`). The lockfile says what you *got*, byte-for-byte. The first is for humans choosing dependencies; the second is for machines reproducing an environment. Commit both.

Example: Titan: `uv.lock` + `--frozen` as a build-time tripwire
Titan pins everything in `uv.lock` and the container builds with `uv sync --frozen --no-dev`. The `--frozen` flag is the load-bearing part: it tells the resolver *"do not update anything: install exactly the lockfile, or fail."* If `pyproject.toml` and the lockfile have drifted out of sync, the build *fails* rather than silently resolving fresh versions. That turns dependency drift from a Heisenbug ("it worked on my machine, it backtests differently in the container") into a loud, build-time error you fix before deploy. The Docker layer that copies `pyproject.toml` and `uv.lock` then runs the sync is cached on those two files, so the (slow) dependency install only re-runs when dependencies actually change.

This matters most for numerical libraries. A minor version bump in a math library can change the last bits of a floating-point result, which changes a z-score, which changes whether a signal crossed a threshold, which changes a trade. A pinned lockfile lets you say "this exact strategy, on this exact code, on this exact dependency graph": the only statement a risk reviewer should accept.

Tip: The lesson is the lockfile, not `uv`
`poetry.lock`, pip-tools (`requirements` + `pip-sync`), and `conda-lock` all do the same job, and each has a fail-on-drift mode that is the equivalent of `--frozen` (`poetry install --sync`, `pip-sync`, `conda-lock --check`). Use it *in the build* so drift is a loud error, not a silent re-resolve. The same idea applies one layer down: pin the base and broker-gateway images by **digest** (`FROM image@sha256:…`), not a floating `:latest` or `:3.12` tag, or you have pinned Python on top of a moving OS. The lockfile pins Python; the digest pins everything underneath; both are required, or the environment drifts the moment an upstream tag moves.

Choice 5: The container as the deployment boundary

The final choice is *where research stops and operations begin*. Drawing that line fuzzily (deploying by `git pull` onto a long-lived VM and running scripts by hand) guarantees that production slowly diverges from anything you tested. Some package got upgraded for one experiment; a stray environment variable lingers; the data on disk is a month stale. The "production environment" becomes a unique snowflake nobody can reproduce.

A **container** makes the boundary explicit and reproducible. The image is a frozen artifact: pinned base OS, pinned dependencies (via the lockfile), application code, and a defined entrypoint. What you build is bit-for-bit what runs. The boundary also clarifies *what belongs in the image versus outside it*:

- **Baked into the image** (changes ⇒ rebuild): dependencies, library and strategy code, the entrypoint.
- **Mounted at runtime** (changes ⇒ no rebuild): configuration, market-data files, model artifacts, and any *mutable shared state*, such as halt flags, logs, and heartbeats, that must survive a restart.

A container is not the only way to freeze a deployment, and the honest version of this choice steelmans the alternatives. An immutable VM image built by Packer and pinned by Ansible gets you a reproducible artifact too; so does a unikernel. We chose a container because it is the lightest unit that bundles a pinned base OS, a lockfile-pinned environment, and a single entrypoint while still composing cleanly with the broker gateway and the risk sidecar, whereas a pinned venv on a shared VM freezes Python but leaves the OS and the gateway floating. The honest cost of the container is container literacy and config-precedence hygiene, which the war-story below shows is not free.

Example: Titan: a multi-service compose stack with a clean state channel
Titan ships as a small `docker compose` stack: a broker-gateway container, the portfolio runner that hosts the engine and all strategies, and a sidecar that computes a risk band. Code and pinned deps are baked into the runner image; `config/`, `data/`, and `models/` are bind-mounted read-mostly so a config edit needs only a restart, not a rebuild. Crucially, all *mutable* shared state, including the persisted kill-switch halt file, logs, and the bar-feed heartbeat, lives on a named volume shared between services. That separation is deliberate: the **kill-switch halt must survive a container restart**, so it cannot live inside the ephemeral image. The deployment boundary is also a *safety* boundary.

Danger: War-story: the recreate that typed empty credentials for hours
Configuration precedence is part of the deployment boundary, and getting it wrong is a live-capital problem. In one incident a container recreate pulled broker credentials from the wrong source: an `environment:` block that substituted from an unset shell variable *overrode* the real values in the dedicated secrets file. The gateway sat for hours typing empty credentials into the login dialog; the API port never opened, and the dependent trading node never started. The bug shape is generic: **two config sources, unclear precedence, and a silent empty-string default that looks like "unset" but behaves like "wrong."** The rules it bought: secrets flow from exactly one source,

never re-declared where a shell default can clobber them; and a config that resolves to empty should *fail loudly*, never quietly proceed. Make the deployment boundary boring and explicit, because the alternative is debugging an empty login dialog at 2 a.m.

Containerisation is the subject of Containerizing the stack (see "Containerizing"), and the operational discipline around it in The live runbook (see "Live Runbook").

Putting the choices together

Layer	Titan's choice	What it buys	What it costs
Execution engine	Managed event-driven (adopted, not built)	Backtest ↔ live parity; realistic order lifecycle	Slower than vectorised; a framework to learn
Research tooling	Vectorised (pandas / vectorbt / numba)	Fast triage of thousands of ideas	Idealised fills, never a deployment verdict
Handoff	Triage in vectors, *trust* only after the engine	The frictionless number can't reach capital	Discipline; two layers to maintain
Dependencies	Pinned lockfile, `--frozen` build	Reproducible backtests; drift fails the build	A lockfile to keep in sync
Deployment	Container + compose, state on a volume	Frozen, reproducible prod; safety state survives restarts	Container literacy; explicit config hygiene

None of these is the "right" choice in the abstract. Each is right *given a constraint*: that backtest and live must be the same program; that most ideas must be killed cheaply before the expensive test; that a measurement must be reproducible; that production must not be a snowflake. Name your constraints first, and the stack choices mostly follow.

Tip: Name the constraints before you name the tools
Before choosing anything, write down five answers; the stack mostly falls out of them.

1. **Frequency and instrument count**: drives engine-vs-DIY (Choice 1).

2. **Parameter-sweep throughput**: drives how hard you fence the vectorised layer and where the handoff sits (Choice 2).
3. **Broker catalogue, jurisdiction, financing/borrow costs, and quote currencies**: drives the substitute and sizing work (Choice 3).
4. **Reproducibility-audit requirement**: drives lockfile plus digest pinning (Choice 4).
5. **Restart-safety needs**: drives what mutable state must live *outside* the image (Choice 5).

Answer these honestly and the five choices above stop being a menu and start being a consequence.

Exercises

1. **Buy the engine, build the edge.** Sort these into "build yourself" vs "adopt": signal logic, order-state machine, position sizing, cash/position accounting, risk policy. State the heuristic.

   Answer: Answer
   Build: signal logic, sizing, risk policy (your edge). **Adopt:** the order-state machine and cash/position accounting (undifferentiated plumbing, brutally hard, pure downside). Heuristic: *build what encodes your edge; buy what merely has to be correct, especially anything that talks to the broker or accounts for cash.*

2. **The parity prize.** What is the single most consequential property a managed event-driven engine buys you over a hand-rolled vectorised loop, and why does it matter?

   Answer: Answer
   Backtest and live run the same strategy code (swap the data/execution clients, not the logic). Without it you validate one program and deploy a different one, and the gap is exactly where look-ahead, fill-model fantasy, and timezone bugs hide.

Takeaways

- **Buy the plumbing, build the edge.** Order lifecycle and cash accounting are undifferentiated, hard, and pure downside; adopt a tested engine. Your signals and risk policy are where you own every line.

- **The non-negotiable property is backtest ↔ live parity**: the same strategy class driving both runtimes. An engine that delivers it is worth its overhead; a DIY loop that quietly breaks it is the most expensive shortcut in the stack.
- **Keep vectorised research, but fence it.** Use it to triage thousands of ideas fast; never let its frictionless fills stand in for a deployment verdict. The edge that only exists in idealised fills is no edge.
- **Broker reality is a correctness constraint, not an afterthought.** Tradability, jurisdiction, and quote currency reshape your universe and your sizing math; verify them against the live account before trusting any backtest.
- **A pinned lockfile turns "works on my machine" into a build-time tripwire.** Loose constraints in the manifest, exact versions in the lockfile, `--frozen` on the build.
- **Make the container the deployment boundary**: frozen code and deps baked in, config and *survivable safety state* mounted out. The boundary is also a safety boundary: the kill switch must outlive a restart.

With the stack decided, the next question is how to organise the code *inside* it so that the research-to-production flow stays one-directional and look-ahead stays visible. That's The project layout (see "Project Layout"), which lays out the `research → config → library → scripts` flow this chapter's parity guarantee depends on.

4. Project layout & non-negotiables

Most trading-system bugs are not clever. They are a `float` where a `Decimal` belonged, a Sharpe reimplemented in a notebook with the wrong annualisation factor, a research script that quietly imported from the live runner, or a backtest that gives a different answer every time you run it. None of these is a hard problem. All of them are *recurring* problems; they come back, in a new file, every few weeks, until you make them structurally impossible.

This chapter is about that structure: the repository skeleton, and the handful of rules every line of code obeys. The layout decides what can import what (and therefore which dependency cycles can ever form). The non-negotiables, financial types, full typing, absolute imports, fixed seeds, one shared metrics module, each kill a specific *class* of bug, not a single instance. The point is never the rule itself; it's the bug class the rule retires for good.

The principle: make the wrong thing hard to write

A coding standard you have to *remember* is a coding standard you will eventually forget: under deadline, at 2 a.m., during the trade that matters. The rules that survive are the ones the tooling enforces or the directory structure forbids. So the test for every convention in this chapter is: **does it convert "remember to be careful" into "you literally cannot do it the unsafe way"?**

That principle has three levers, in order of strength:

1. **Make it impossible.** The dangerous operation does not exist in the codebase (no full-series z-score; no money in `float`).
2. **Make it fail loudly.** A required argument with no default; a linter that rejects the import; a type checker that catches the wrong shape before it runs.
3. **Make it conspicuous.** A directory boundary that means a bad import shows up in review as an obviously-wrong line.

Everything below is an application of those three levers.

The repository skeleton

> **Note: Recap, then the new material**
> The four-layer skeleton and the one-way arrow below are a deliberate recap of the architecture chapter (see "Architecture"); skim it if it's fresh. The *new* material this chapter owns starts at the non-negotiables (typed money, full typing, fixed seeds, one shared metrics module): the coding rules, distinct from the directory rules. And the *integration contract* the architecture chapter pointed forward to (the interface every strategy signs with the risk layer) is its own chapter, the strategy-class & integration contract (see "Strategy Class Contract"); it isn't here, because it's a Part-IV concern, not a layout one.

A quant repo has four kinds of code, and they must not blur into each other:

- **Research**: exploratory, allowed to be messy, allowed to be wrong, *never* imported by anything live.
- **Library**: the audited core: strategy logic, indicators, risk, the shared metrics. Importable everywhere.
- **Config**: the parameters that research discovered, captured as data (TOML), not code. *Captured as data* means *validated* as data: a config is loaded through a schema (required keys, typed values, bounded ranges) so a missing or out-of-range parameter fails loudly at startup rather than silently defaulting; otherwise "parameters as data" is just a different place for a typo to hide. The loader is the config-layer cousin of the required `periods_per_year` argument you'll meet below: a boundary that refuses to accept a malformed input.
- **Entry points**: the scripts that actually run things: live runners, validators, the kill switch.

Titan lays these out as top-level directories with one rule that does almost all the work:

```
directives/   Specs and methodology docs (read first)
titan/        Audited library - strategies, adapters, indicators, risk,
 ↪  metrics
  research/metrics.py     The shared math module (the subject of half
 ↪  this book)
  research/framework/     The unified backtest framework (WFO, MC, DSR,
 ↪  decision)
  risk/                   PortfolioRiskManager, Allocator,
 ↪  StrategyEquityTracker
research/     Exploratory research - signal defs, regime detectors,
 ↪  audits
```

```
config/       TOML parameters (what research found; re-tuned per audit)
data/         Historical Parquet (gitignored; regenerated by scripts)
scripts/      Entry points - run_live_*, kill switch, watchdogs,
 ↪  validators
tests/        The safety net
```

> **Tip: Code flows one direction**
> `research/` **discovers** → `config/` **captures** → `titan/` **implements** → `scripts/` **executes**. The two hard rules that fall out of it: **never import from `scripts/` inside `titan/`**, and **never put an entry point inside `titan/`.** A library that imports its own runner has a cycle waiting to happen and a test suite that can't isolate the unit under test. If you ever find yourself reaching back up the arrow (a library module importing a script, or a strategy importing a research notebook), the design is telling you the boundary is in the wrong place. Two directories sit *off* the arrow rather than on it: `tests/` may import anything (it lives below the flow, not inside it, so a test reaching into `research/` or `scripts/` is fine), and `data/` is an artifact directory, not a dependency; code writes and reads Parquet there, but no module ever `import`s from it.

The reason `research/` is quarantined from `titan/` is not snobbery about code quality. It's that research code is *allowed to be wrong*; that's its job, to try things that don't work. The moment live code can import it, "allowed to be wrong" becomes "shipped to production." The directory boundary is what lets research stay fast and loose without that looseness ever reaching capital.

Non-negotiable 1: money is never a `float`

Floating point is base-2; money is base-10. `0.1 + 0.2` is not `0.3` in IEEE-754, and the error is not academic: it compounds through position sizing, accumulates across fills, and eventually a quantity that should be a clean integer of shares is `99.99999999998`, which the broker rejects, or worse, silently rounds in a direction you didn't choose. Prices have a tick size and quantities have a lot size; both are *exact* decimal grids, and `float` cannot sit on a decimal grid.

The rule: **anything that is money, a price, a quantity, or a tradable size is `Decimal` or the broker's native typed object, never a raw `float`.** Titan runs on NautilusTrader, whose `Price` and `Quantity` types carry their own precision and are constructed from `Decimal`:

```
from decimal import Decimal

from nautilus_trader.model.objects import Price, Quantity

# A price snapped to the instrument's tick grid at its declared
 ↪  precision:
sl = Price(Decimal(sl_price), precision=instrument.price_precision)
qty = Quantity(Decimal(shares), precision=instrument.size_precision)
```

`float` is fine, preferred, even, for the *statistical* layer: returns, Sharpe, vol, z-scores. Those are dimensionless ratios where base-2 rounding at the 15th digit is irrelevant. The discipline is a boundary, not a blanket ban: **the moment a number becomes an order, it must be a typed money object.** A clean place to enforce that is the conversion at the edge of the strategy: the one function that turns "I want to risk 1% of equity" into a `Quantity` is the only place `float` is allowed to touch a size, and it converts on the way out.

Snapping to the grid is a *policy* choice, not just a precision one: the `99.99999999998` above has to resolve to `99` or `100`, and **which** is a decision you make, not one you inherit. Decide explicitly whether you round half-up, half-even (banker's), or always toward the conservative (smaller) size, and round *once*, at the boundary, where the `float` becomes a `Quantity`, because `Decimal` defaults to `ROUND_HALF_EVEN` while a broker may expect something else, and an unstated tie-break is one more place a wrong assumption hides until a fill lands on it. The full sizing-and-rounding treatment lives in Per-strategy equity & FX (see "Per Strategy Equity Fx"); here the rule is just that the policy is *stated*, not defaulted into.

Danger: War-story: the leg sized a third too large on a currency assumption
A leg quoted in one currency, sized from an account based in another, went on about a third too large because the FX conversion was a plain untyped `float` multiply with the wrong leg: no crash, just a quietly wrong position no test caught. A `Money(amount, currency)` type *can* turn this into a type error. But be precise about which one: a stock `Money` (stdlib or the engine's) carries a currency yet will happily let you multiply it by a bare rate, because FX conversion *is* `money × scalar`, so the compiler catches nothing. The guardrail is a type that **refuses to combine two different currencies without an explicit, rate-checked `convert()`**; then adding USD to JPY, or sizing a JPY leg off a USD budget, is unrepresentable rather than silently wrong. Reach for the stock `Money` expecting the compiler to save you and you're exactly as unprotected as the war-story; the full sizing autopsy is in Per-strategy equity & FX (see "Per Strategy Equity Fx").

Non-negotiable 2: full typing and absolute imports

Two rules that look like style and are really about isolation.

Full type hints, checked. Every function signature is annotated; the return type is explicit. Type hints are not documentation that rots; with a checker in CI they are a test that runs on every line. Most of the bugs they catch are dull: a function that returns `pd.Series` handed to one expecting `float`, a `None` that slips through an `Optional` you forgot to handle. Dull is the point: those are exactly the bugs that survive code review because they read as plausible. The metrics module models the standard well:

```
def sharpe(
    returns: pd.Series | np.ndarray | Iterable[float],
    periods_per_year: int,
    *,
    ddof: int = 1,
) -> float: ...
```

The signature tells you everything: what it accepts, that the annualisation factor is *required* (more on that below), and that it returns a scalar. A reviewer verifies correctness from the signature without reading the body.

Absolute imports only. `from titan.research.metrics import sharpe`, never `from ..metrics import sharpe`. Relative imports break the instant a file moves, hide the dependency direction (a `..` tells you nothing about which layer you're reaching into), and make it possible to construct a deep relative path that quietly crosses a boundary the directory layout was supposed to enforce. An absolute import is self-documenting: `from titan...` is library, `from research...` is exploratory, and a `from research...` line inside `titan/` is a glaring red flag in review. The import path *is* the architecture diagram, restated on every line.

> **Tip: Let the linter own the boring rules**
> Titan's `pyproject.toml` runs Ruff with the import (`I`), pyflakes (`F`), pycodestyle (`E`/`W`), and docstring (`D`) rule sets, line length capped, on a pinned Python target. Unused imports (`F401`), unsorted imports (`I`), undefined names: all caught before a human looks. The per-directory ignore table is itself a policy statement: research and scripts get relaxed line-length and docstring rules (they're allowed to be scrappy), while `titan/` is held to the strict set. The *layout* and the *lint config* encode the same hierarchy of care.

Non-negotiable 3, reproducibility: a fixed seed, always

A backtest that gives a different answer every run is not a measurement; it's a slot machine. Any procedure with randomness (a bootstrap confidence interval, a Monte Carlo path simulation, a model fit, a train/test shuffle) must take an explicit seed and default it to a fixed value, so the same inputs produce the same outputs forever.

```
def bootstrap_sharpe_ci(
    returns, periods_per_year, n_resamples=1000, confidence=0.95,
    *, seed: int = 42, block_size: int | None = None,
) -> tuple[float, float]:
    rng = np.random.default_rng(seed)   # explicit generator, fixed seed
    ...
```

Two non-obvious reasons this matters more than it looks:

- **Audits must be replayable.** When a strategy is rejected because its bootstrapped 95% lower bound came in below zero, that verdict has to be reproducible to the digit, or it's just an opinion. A fixed seed turns "we ran it and it failed" into "anyone who runs this exact code gets this exact number."
- **It separates real instability from RNG noise.** If a result swings when you change *only* the seed, the result is fragile and the seed was hiding it. You *want* to vary the seed deliberately, across a grid, to measure that fragility. But that's a different experiment from your headline run, and it only works if the default is pinned, so a seed change is a *choice* you made, never an accident. So the headline run pins exactly one seed for replayability; the *robustness* claim is a separate experiment, a sweep over a seed grid whose spread you report alongside the point estimate, never the single-seed CI dressed up as robust. The `42` is arbitrary on purpose, any fixed integer works, what matters is that it's recorded next to the result (in the pre-registration or the run log) so the exact number regenerates.

Warning: War-story: the global RNG that made an audit unreproducible
A simulation seeded the process-global random generator once at import, then several modules each drew from it. Run them in a different order, or import one extra module, and the draws shifted, so the same backtest produced subtly different confidence intervals on different machines. A strategy could pass the `CI_lo > 0` gate on the researcher's laptop and fail in CI, with no code change between them.

The bug wasn't the seed value; it was *sharing one mutable global generator across modules*. The fix: every stochastic function constructs its **own** `np.random.default_rng(seed)` from an explicit argument and never touches the global state. Local generator, explicit seed, default pinned: and the result is the same on every machine, every run, in any import order.

Non-negotiable 4, one shared metrics module

This is the rule the rest of the book leans on hardest, so it gets its own section. **Every** Sharpe, Sortino, Calmar, volatility, drawdown, and z-score in the codebase routes through a single module. Not "should." Does. The alternative, each researcher writing `def _sharpe(r): return r.mean()/r.std()*np.sqrt(252)` at the top of their notebook, was tried, and it failed in the most instructive way possible.

Warning: War-story: the same bug, copy-pasted into six files
An audit of Titan found the *identical* Sharpe miscalculation independently reimplemented in at least six places. Several research evaluators filtered out flat (`return == 0`) bars before annualising with `sqrt(252)`, which inflates the Sharpe by roughly `sqrt(1/P)` for a strategy that only trades `P` of the time, a free multiple for any selective strategy. One portfolio metric inlined `sqrt(252)` onto what were sometimes *hourly* bar returns, mis-annualising by `sqrt(24)`. And the same `sqrt(252)`-on-intraday error appeared on the **live** side, in the sizing code of multiple strategies, where it understated annualised vol and therefore systematically *over-sized* every position. The bug wasn't any one researcher's mistake. It was the *absence of a single place to be correct*: every fresh reimplementation was a fresh chance to get it wrong, forever.

The cure is a module designed so the wrong thing is hard or impossible to call. Three concrete design choices show how the levers from the top of this chapter cash out:

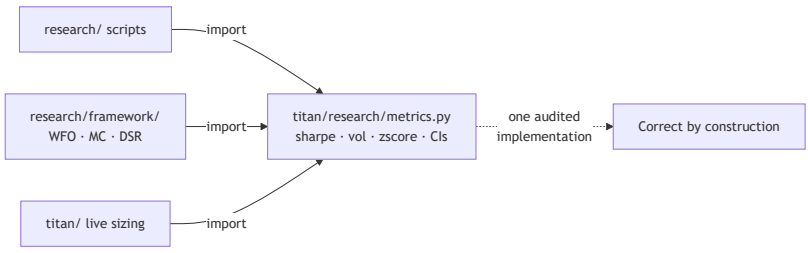
Lever	In the metrics module	Bug class it retires
Make it fail loudly	`periods_per_year` is a *required* argument on every Sharpe/vol call, with no default	Wrong-frequency annualisation (the `sqrt(24)` and `sqrt(252)`-on-hourly errors)

Lever	In the metrics module	Bug class it retires
Make it impossible	No Sharpe filters `returns != 0` internally; per-trade stats live in a *separate* `trade_sharpe`	The selectivity-inflation bug, retired by construction
Make it impossible	Only *causal* (`rolling_zscore`, `expanding_zscore`) and IS-frozen z-scores exist; the full-series version is simply absent	Look-ahead-by-construction normalisation

The required `periods_per_year` argument is the cleanest example of the philosophy. It looks like a papercut: why can't it just default to 252? Because a default is a place a wrong assumption hides. Forcing every caller to *state the frequency at the call site* means a reviewer can verify the annualisation in one glance, and it makes a mid-pipeline resample (the classic "this H1 strategy is secretly daily" trap) impossible to leave implicit. A tiny bit of ceremony at the call site, in exchange for a class of bugs that can no longer recur. That trade is the whole book in miniature.

There's a softer benefit too: edge cases are handled *once*, correctly. The shared functions return `0.0` or `NaN` on empty, constant, or too-short series rather than raising, so a guardrail metric never crashes a batch run; it just declines to flatter you. Get that right in one place and every one of the hundreds of call sites inherits it.

If you're starting from the six-copies state rather than a clean slate (the realistic case; almost no repo arrives with one module already), the migration is mechanical, not heroic: grep the tree for the tells (`sqrt(252)`, `.std() *`, a `returns != 0` filter, any ad-hoc `_sharpe` def), replace each with a call into the shared module, then lock the door behind you with the enforcement grep below so the *seventh* copy fails CI. Retrofit the **live sizing paths first**: that's where a wrong annualisation isn't an inflated number on a report, it's an over-sized real position.



The mechanics of *why* each metric rule prevents its bug, the units, the survivor math, the shift discipline, the future-normalised feature, the error bars, are the subject of **A backtest you can trust** (see "Backtest You Can Trust"), and the full battery beyond Sharpe (Sortino, Calmar, CVaR/CDaR, risk of ruin) gets **its own chapter** (see "Metric Suite"). What matters *here* is the structural decision: there is exactly one module, every Sharpe / Calmar / drawdown / vol in research *and* in live sizing routes through it, and using anything else is the bug.

Tip: Enforcement: three of these four rules are still discipline until you wire them to the build
Only Non-negotiable 2 (typing, imports) is enforced so far; the linter owns it. The other three (typed money, fixed seed, one metrics module) are honour-system until a check fails CI on a violation. Here is the cheapest version that makes each fail loudly; none is more than a few lines, all run in seconds, and like Ch.2's 30-second audit they are deliberately crude: a grep that's occasionally over-eager beats a rule nobody enforces.

```
# (4) No inline annualisation outside the one metrics module.
git grep -nE 'sqrt\(\s*(252|24|365)\b' -- 'titan/**' 'research/**' \
    ':(exclude)titan/research/metrics.py' && echo 'inline
    ↪  annualisation' && exit 1

# (3) No global-RNG seeding or bare global draws outside metrics.
git grep -nE 'np\.random\.(seed|rand|randn|choice|normal)\(' --
 ↪  'titan/**' \
    ':(exclude)titan/research/metrics.py' && echo 'global RNG' &&
    ↪  exit 1
```

The money rule (1) doesn't grep cleanly (a `float` looks like any other `float`), so it gets a unit test instead: assert that the strategy's size-producing call sites return a `Quantity`/`Price`, not a bare `float`, so a raw number can never reach `submit_order`. The grep guards are blunt by design; tighten the excludes as the tree grows, but keep them failing on the *default* unsafe pattern. A rule the build enforces is a rule; a rule in a style doc is a wish.

Exercises

1. **The three levers.** The chapter ranks three ways to enforce a convention, strongest first. Name them, and give the strongest's form for "no look-ahead z-score."

Answer: Answer

(1) **Make it impossible**: the full-series z-score *doesn't exist* in the codebase. (2) **Make it fail loudly**: a required arg with no default, a linter, a type checker. (3) **Make it conspicuous**: a directory boundary so a bad import reads as obviously wrong in review. Strongest is "impossible": the dangerous operation isn't there to call.

2. **One-direction imports.** State the two hard import rules that fall out of "research discovers → config captures → `titan/` implements → `scripts/` executes."

Answer: Answer
Never import from `scripts/` inside `titan/`, and **never put an entry point inside `titan/`.** A library that imports its own runner has a dependency cycle waiting to form and a test suite that can't isolate the unit under test.

Takeaways

- **The layout enforces the architecture.** Four kinds of code (research, library, config, entry points) with a one-directional flow: discover → capture → implement → execute. Never import upstream; never put a runner in the library.
- **Money is never a `float`.** Prices, quantities, and sizes are `Decimal` or typed broker objects that carry precision and currency, so a cross-currency or off-tick error becomes a type error, not a silent mis-size. `float` stays in the statistical layer where base-2 rounding is harmless.
- **Type everything; import absolutely.** Checked type hints are tests that run on every line; absolute imports restate the dependency direction on every line and make a boundary violation glaring. Let the linter own the boring rules.
- **Pin the seed.** Every stochastic procedure takes an explicit seed, defaults it to a fixed value, and constructs its *own* generator, so audits replay to the digit and seed-sweeps measure real fragility instead of hiding it.
- **One shared metrics module, or the same bug six times.** Centralise the math so the unsafe version can't be written: a required `periods_per_year`, no internal zero-filtering, no full-series z-score. The safest API is one where the dangerous operation simply does not exist.

Every rule here is an instance of the same move: convert "be careful" into "can't do it wrong." The next chapter, **A backtest you can trust** (see "Backtest You Can Trust"), takes the shared-metrics module apart function by function and shows exactly which lie each design choice prevents; and why every lie, left unchecked, flatters the strategy. The cross-currency mis-size that opened this chapter's first war-story comes back in **Per-strategy equity & FX** (see "Per Strategy Equity Fx") and the **portfolio risk manager** (see "Portfolio Risk Manager"), where the one audited place to do currency-aware sizing actually lives.

5. Thinking in edges: typology & hypotheses

Most strategies are not killed by a bad idea. They are killed by a good idea tested with the wrong ruler. A breakout strategy that trades nine times a year, measured with a per-bar Sharpe over thousands of flat bars, looks like noise. A trend strategy validated on six months of out-of-sample data dies (or survives) on a single regime that happened to fall in the window. In both cases the *edge* was never the question. The *validation regime* was wrong for the class of thing being validated, and the verdict was therefore meaningless before the first number printed.

This chapter is about the work that happens *before* a backtest: deciding what kind of edge you are claiming, writing it down as something that can be proven false, and pre-committing to how you will judge it. Do this and the rest of Part II (a backtest you can trust (see "Backtest You Can Trust"), walk-forward (see "Walk Forward"), deflation (see "Deflated Sharpe")) has a fixed target to aim at. Skip it and every downstream test inherits a moving goal-post that you, the hopeful author, will move toward "pass" without ever noticing.

The principle: classify before you measure

A trading edge is a hypothesis about *why* a market is mispriced and *how often* that mispricing pays. Those two facts, the mechanism and the cadence, determine almost everything about how the strategy should be measured. Get the cadence wrong and your headline number is off by a multiplicative constant. Get the mechanism wrong and you validate it on a window that can't possibly contain the regime it depends on.

So the first artefact in any research project is not code. It is a **classification**. You assign the candidate to a *strategy class*, and the class carries a pre-decided validation regime: which Sharpe convention is the gate, how the walk-forward is shaped, how the Monte Carlo stress is built, and what drawdown is structural rather than a failure. The point is to bind those decisions *before* you have seen any results, so you cannot tune the ruler to flatter the measurement.

This inverts the natural order of research, which is exactly why it works. Left to instinct, you run the backtest, see a Sharpe, *then* decide whether to annualise per-bar or per-trade, whether six months of OOS is "enough," whether a 30% drawdown is "acceptable for this kind of thing." Every one of those after-the-fact choices is a degree of freedom you will, unconsciously, spend on optimism. Classification spends them in advance, when you have no result to be optimistic about.

Per-bar vs per-trade: the cadence trap

The clearest example is the Sharpe convention. There are three honest ways to annualise a strategy's return-per-risk, and they are *not* interchangeable:

Convention	What it measures	When it's right
per-bar	return per price bar, scaled by bars/year	Continuously-in-market strategies; signal updates every bar
per-day-MTM	daily mark-to-market return, scaled by 252	Position strategies held across many bars; what a daily NAV would show
per-trade	return per round-trip trade, scaled by trades/year	**Sparse** strategies: few, discrete entries with flat time between

The trap is the sparse strategy. Consider a breakout that is in the market a small fraction of the time. Measure it per-bar and you are dividing a handful of trade returns by the standard deviation of *thousands* of mostly-zero bars. Two pathologies follow at once. If you (wrongly) keep the zeros, the volatility is dominated by flat bars and the number is meaningless. If you (wrongly) drop the zeros and still annualise with the full per-bar factor, you inflate the Sharpe by `1/sqrt(fraction-in-market)`: the survivor-math lie from the next chapter (see "Backtest You Can Trust"). The honest measurement for a sparse strategy is **per-trade**: take the round-trip P&L of each discrete trade, compute Sharpe over *those*, and annualise by the number of trades per year. The cadence of the measurement matches the cadence of the edge.

> **Tip: Cadence is a property of the edge, not a knob**
> Ask one question of any candidate: *how often does this thing actually decide?* A trend follower decides every day (per-day-MTM). A microstructure mean-reverter decides every bar it's awake (per-bar). A breakout decides only when a range breaks (per-trade). The answer is fixed by the strategy's logic, so the Sharpe convention should be fixed too: written down before the backtest, not chosen after.

Per-trade buys honesty, but it does not buy a free lunch. A Sharpe computed over nine round-trips a year is a low-power, high-variance estimate with a wide bootstrap CI: swapping per-bar for per-trade trades the zero-dilution sin for a tiny-N sin. That is exactly why the `INTRADAY_BREAKOUT` class does not lean on a point Sharpe alone. It pairs the per-trade convention with a minimum-trade-count gate (too few round-trips and the result

is *unconfirmed*, not "passed"), an expanding walk-forward that accumulates folds so the deflation in beating your own optimiser (see "Deflated Sharpe") has trades to work with, and a bootstrap *lower bound* rather than a headline number. The honest ruler still needs enough marks on it to read.

The middle row earns the same scrutiny, because **per-day-MTM** is the convention most of the example classes default to. For a position strategy held across many bars, per-bar would *over*-count: the same open position is re-measured every bar, and its bar-to-bar autocorrelation understates the volatility a daily NAV actually shows. per-day-MTM marks the book once a day to the daily close, so a multi-day hold contributes one return per calendar trading day, with overnight and weekend gaps landing in the next mark, the way a real account experiences them. The Sharpe then matches what a daily statement would report. It is the right ruler whenever the decision cadence is daily but the holding period spans many bars, which is why trend, carry, vol-carry, and cross-asset momentum all sit on it.

The Titan example: a typology table that ships defaults

Titan encodes this discipline as a literal lookup table. Each candidate is assigned one of a closed set of **strategy classes**, and a class maps to a frozen bundle of defaults: Sharpe convention, walk-forward shape, Monte Carlo design. The enum is the commitment; the lookup is the enforcement.

```
class StrategyClass(Enum):
    INTRADAY_MICROSTRUCTURE        # H1+, sparse-ish, mean-rev / range
    INTRADAY_BREAKOUT              # M5/M15, ORB-style, very sparse
    DAILY_TREND                    # persistent, long-biased
    DAILY_MEAN_REVERSION           # oscillator-based
    DAILY_MEAN_REVERSION_VOL_CARRY # short-vol / VRP harvest
    CROSS_ASSET_MOMENTUM           # always-on, signal from another asset
    PAIRS                          # market-neutral spread
    ML_CLASSIFIER                  # predicts a label, holds until flip
    META_LABELING                  # a primary signal + an ML
↪   accept/reject filter
    CARRY                          # slow FX carry
```

Each class resolves through one function to its defaults:

```
def defaults_for(cls: StrategyClass) -> StrategyClassDefaults:
    """Look up the framework defaults for a strategy class. Raises if
    ↪  missing."""
    if cls not in DEFAULTS:
        raise KeyError(f"No defaults for {cls.value}. Add a row via a "
                       f"pre-registration directive before using this
                        ↪  class.")
```

```
    return DEFAULTS[cls]
```

The bundle it returns carries three decisions a researcher would otherwise make *after* seeing results:

```
@dataclass(frozen=True)
class StrategyClassDefaults:
    sharpe: SharpeReporting   # which Sharpe is the gate, which is
↪   diagnostic
    wfo:    WfoConfig         # IS/OOS window lengths, fold count,
↪   expanding vs rolling
    mc:     McConfig          # block size, paths, the
↪   structural-drawdown threshold
```

Two design choices in that code do real work. First, `defaults_for` **raises** on an unknown class: you cannot validate a strategy that hasn't been classified, and you cannot invent a class on the fly. Adding a class requires a written directive that appends *both* the enum member *and* its defaults row; there are no silent additions. Second, every config is a `frozen=True` dataclass, so the defaults can't be mutated in a notebook halfway through an analysis. The ruler is immutable for the duration of the measurement.

How the class changes the ruler

The same candidate idea, filed under two different classes, is judged by two genuinely different regimes. A few illustrative rows (the exact values are tuned per directive; treat the numbers as *shapes*, not deployable settings):

Class	Primary Sharpe	WFO window shape	Structural MaxDD it tolerates
`INTRADAY_BREAKOUT`	**per-trade**	short IS, expanding	tighter: breakouts shouldn't bleed
`DAILY_TREND`	per-day-MTM	multi-year IS, expanding	wide: trend pays for the drawdown
`CROSS_ASSET_MOMENTUM`	per-day-MTM	shorter IS, **rolling**, overlap allowed	moderate
`DAILY_MEAN_REVERSION`	per-day-MTM	multi-year IS, expanding	**very wide**: short-vol's tail is the edge
`CARRY`	per-day-MTM	**longest** IS, expanding	moderate

Read across that table and the logic is visible. Trend and carry need *long* in-sample windows because their edge only shows up across multiple regimes: a two-year window can miss the very trend or rate cycle the strategy harvests, so the framework demands more history before it will believe a result. Cross-asset momentum uses a *rolling* window with overlap because the relationship it trades (one asset's signal, another's return) drifts, and an expanding window would let ancient, dead correlations dominate. And the drawdown thresholds differ by an order of magnitude on purpose: a breakout that sits in a deep drawdown is broken, whereas a short-volatility carry strategy that *never* takes a large drawdown is almost certainly mis-measured, because absorbing rare, violent losses is precisely how it earns its premium.

Note: The drawdown threshold encodes a belief about the tail
Titan's vol-carry class carries a deliberately wide structural-drawdown tolerance: large peak-to-trough losses are treated as the *cost of the premium*, not as evidence of failure. That number is not generosity; it's a pre-commitment to *not* kill a short-vol strategy at the bottom of the exact drawdown it was always going to have. The inverse error, using that wide tolerance for a class whose deep drawdowns really are failures, is exactly why the threshold is bound to the class, not chosen per-run. And it is a *MaxDD* gate precisely because Sharpe is blind here: two classes with the same Sharpe can have wildly different drawdown geometry and tail shape, which is why the typology also pre-commits the **Calmar** (return-per-drawdown) and **CVaR/CDaR** (tail-loss) lenses each class is judged on. See beyond Sharpe: the metric suite (see "Metric Suite") for the battery, and tail risk & risk of ruin (see "Tail Risk And Ruin") for how the threshold becomes a survival probability.

Falsifiable first: write the obituary before the birth

Classification fixes the ruler. A **hypothesis** fixes the question. Before any backtest, write one sentence of the form:

> *This edge exists because* `<mechanism>`*; it should produce* `<measurable effect>` *under* `<conditions>`*; and it would be falsified by* `<specific observable>`.

The non-negotiable clause is the last one. A hypothesis you cannot imagine *failing* is not a hypothesis, it's a hope, and hope has a 100% in-sample hit rate. "Momentum works" is unfalsifiable. "A short-term momentum signal on this asset class produces a positive per-day-MTM Sharpe whose bootstrap lower bound clears zero out-of-sample, and is falsified if the lower

bound is ≤ 0 across the pre-committed folds" is a claim: it names the metric, the regime, and the death condition, all *before* the data is touched.

This matters because the alternative is the most expensive failure mode in quant research: searching until something passes, then inventing the mechanism afterward. With a pre-written falsification criterion, a failing result is *information*: the hypothesis is dead, you learned something, move on. Without one, a failing result is just a prompt to try another parameter, and you have begun, quietly, to overfit the test set with your own attention. The deflation machinery in beating your own optimiser (see "Deflated Sharpe") exists to correct for exactly this multiple-comparisons drift; pre-registration is the cheaper, upstream defence.

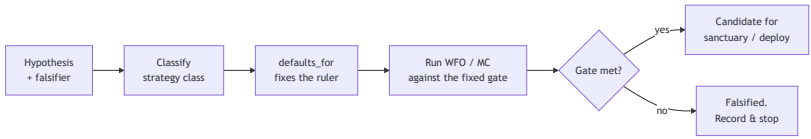
Example: A hypothesis with teeth (illustrative)
Mechanism: credit-market stress leads broad equity by a short lag (credit investors react first). *Effect:* a signal derived from a high-yield credit proxy should predict a related equity sleeve's forward return, yielding a positive per-day-MTM Sharpe whose **bootstrap 95% lower bound is above zero** across pre-committed rolling folds. *Falsified if:* the lower bound is ≤ 0, or the effect only appears with a same-bar (non-lagged) signal, which would mean we're reading the present, not predicting the future. Note what this commits to *before* the run: the class (cross-asset momentum), the metric (per-day-MTM, lower bound), the fold shape (rolling), and a leak-detector built into the falsification clause. The mechanism is textbook; the *deployed* instruments, lookback, and live numbers are not on this page; a hypothesis with teeth doesn't require publishing the edge.

Pre-registration as a commitment device

Put the two together (classification and hypothesis) into a short document written *before the experiment*, and you have a pre-registration. Borrowed from clinical trials, where it exists to stop researchers from quietly redefining the primary endpoint after seeing the data, it does the same job here. The pre-registration names, in advance:

- the **strategy class**, and therefore the Sharpe convention, WFO shape, and MC design that follow from it;
- the **hypothesis** and its falsification criterion;
- the **deployment gate** (e.g. bootstrap lower bound > 0 across the pre-committed folds);
- the **universe and window**: the instruments and the date range you will test on, fixed before you peek.

Note: One pre-registration governs a family, not a single line of code
A pre-registration does not forbid a parameter sweep. You may test fifty variants under `DAILY_TREND`, but they share one pre-committed class, one falsifier, and one gate; the variant count simply *becomes* the trial count that the deflation in beating your own optimiser (see "Deflated Sharpe") discounts for downstream. The two devices split the work cleanly: pre-registration fixes the ruler and the death condition *upstream*, before any result; deflation prices the searching you did *within* them, after. A sweep is not cheating. Pretending fifty trials were a single lucky one is.



The value of writing it down is *psychological*, and that is not a weakness. It is a precommitment against your own future self, who will be staring at a Sharpe of 1.4 and looking for a reason to keep it. If the pre-registration said "rolling folds, per-day-MTM, lower bound > 0," you cannot later switch to per-trade-on-the-active-bars-only because it reads better. The document is the auditor that the excited version of you cannot bribe.

Example: A pre-registration, concretely (illustrative template)
The artefact is small, a committed file, e.g. `research/preregistrations/<strategy>.yml`:

```yaml
strategy_id:    credit_to_equity_v1
strategy_class: CROSS_ASSET_MOMENTUM     # fixes Sharpe convention,
 ↪  WFO shape, MC design
hypothesis:     "Widening HY credit spreads lead equity weakness; a
 ↪  credit-momentum
                 signal should earn positive risk-adjusted return on
 ↪  an equity-ETF leg."
falsifier:      "stitched-OOS Sharpe CI lower bound <= 0, OR the
 ↪  sanctuary year loses money"
gate:           { ci_lo: ">0", dsr_prob: ">=0.95", mc_p_maxdd_gt:
 ↪  "<=class_threshold" }
universe:       ["<signal ETF>", "<traded UCITS ETF>"]
date_range:     ["2008-01-01", "2024-12-31"]
data_sha256:    "a1b2c3…"                # the exact dataset, hashed
author:         rayan
date:           2026-06-29                 # committed BEFORE the
 ↪  data-loading commit
```

Two things make it binding rather than decorative: it is **committed before** the commit that first loads the data (git history timestamps the blindness), and it **hashes the dataset** so the run can't quietly swap in a more flattering window.

Warning: What stops you cheating your own pre-registration?
Be honest about the teeth. Nothing *mechanically* prevents a solo researcher from running the sweep, *then* writing the pre-registration to look blind, or committing it first having already seen results offline. Commit-ordering and a dataset hash *raise the cost* of cheating; they don't make it impossible for a determined author working alone. What makes the device real for a mostly-honest shop is wiring it in: have the WFO harness **refuse to run** unless a committed pre-reg for that strategy class exists, and record the dataset hash *in the result* so a reviewer can diff the registered window against the tested one. It is an honesty aid that makes the easy cheat inconvenient and the deliberate cheat leave fingerprints, not an adversarial proof. Claiming more would be the exact over-confidence the chapter warns against.

Warning: War-story: the breakout that was graded on the wrong curve
A sparse intraday breakout was first evaluated with a per-bar Sharpe over its full bar history: thousands of mostly-flat bars between a handful of trades. The number was unremarkable, and the idea was nearly

shelved. Re-filed under the breakout class, it was measured per-*trade* (round-trip P&L, annualised at trades-per-year), the convention the class mandates, and the picture was completely different, because the per-bar version had been diluting a real per-trade edge into a sea of zeros. The lesson is not "per-trade is better." It's that *the convention is a property of the class, and choosing it after the fact, in either direction, corrupts the verdict.* The fix was structural: bind the Sharpe convention to the class at classification time, so it can never be a post-hoc choice.

Danger: War-story: the validation window that couldn't contain the edge

A trend-following candidate was validated on a short out-of-sample window that happened to be a directionless, choppy regime: the one environment in which trend following is *supposed* to lose. It failed the gate and was discarded. It was, plausibly, a real edge tested where it structurally could not show up. The mirror-image error is worse and touches live capital: validating a slow trend or carry strategy on a *too-short* window that happens to contain one favourable regime, passing it, sizing it for real money, and then meeting the absent regimes live. The rule the typology bakes in: **slow, regime-dependent classes (trend, carry) require long in-sample windows by default**; the framework refuses to certify them on a window too short to contain the regimes their edge depends on. The window length is not a tuning parameter; it's a consequence of the class.

Danger: War-story: the result the falsifier made me bin

A mean-reversion candidate printed an in-sample Sharpe north of 2: the kind of number you start mentally spending. But the pre-registration, written before the data was loaded, had committed the death condition: bootstrap lower bound > 0 across the pre-committed folds, *and* the sanctuary year must not lose money. The lower bound came in negative, and the sanctuary year was red. Under the pre-reg, that is not a near-miss to nurse; it is falsified, and it was binned. The honest part of the story is the temptation: without the pre-written falsifier, the obvious next move was to re-parameterise (widen the look-back, drop the worst fold, swap the sanctuary year) until the lower bound crept positive, and the thing would have "survived" by quiet re-tuning of the test it was supposed to pass blind. The discipline only becomes credible the first time it costs you a result you wanted to keep. This one did, and that is the point.

Note: When an idea doesn't fit a class (and when a taxonomy can be gamed)
The closed enum is a feature, but every reader hits an idea that doesn't cleanly classify in the first week. Three cases worth pre-empting. **A genuinely novel mechanism** that fits no class: don't force it into the nearest flattering bucket; add a class through a pre-registration directive (the same gate the defaults table uses), so the new ruler is committed *before* any result. **A hybrid that spans two classes** (a meta-labelled breakout, `INTRADAY_BREAKOUT` or `META_LABELING`?): pick the class whose *validation regime is stricter*, never the one whose Sharpe convention reads better, and record why. **The gaming risk cuts both ways:** rigid buckets remove the freedom to tune the ruler after the fact, but a too-*coarse* taxonomy lets you smuggle a result into a flattering class, while a too-*fine* one (every idea its own class) defeats the point. The right count is set by *distinct validation regimes*, not by idea variety: two ideas that share a ruler (same Sharpe convention, same WFO shape, same MaxDD tolerance) share a class, and the test for adding a class is whether it would genuinely change the validation regime, not whether the idea *feels* new. A handful of classes is usually right for a single shop. The discipline lives in the borderline cases. Resolve them in writing, in the pre-registration, when you have no result to be optimistic about.

Exercises

1. **Cadence is not a knob.** A breakout that trades nine times a year is being measured with a per-bar Sharpe over thousands of flat bars. Why is that the wrong ruler, and what's the right one?

   Answer: Answer
   Per-bar divides a handful of trade returns by the std of mostly-zero bars: keep the zeros and the number is meaningless; drop them and you inflate Sharpe by `1/sqrt(fraction-in-market)`. The honest ruler is **per-trade**: Sharpe over round-trip P&L, annualised by trades/year. Cadence is fixed by the edge's logic, not chosen after seeing the result.

2. **Classify before you measure.** Why does assigning a candidate to a strategy class *before* running the backtest remove a source of optimism bias?

Answer: Answer
The class binds the validation regime (Sharpe convention, walk-forward shape, MC design, tolerable drawdown) *in advance*, when you have no result to flatter. Decide them after seeing a Sharpe and each is a degree of freedom you'll unconsciously spend toward "pass."

Takeaways

- **Classify before you measure.** The strategy class, not the result, should fix the Sharpe convention, the walk-forward shape, and the structural-drawdown tolerance. Decide the ruler before you can see what you're measuring.
- **Cadence picks the Sharpe.** Continuous strategies are per-bar or per-day-MTM; *sparse* strategies (breakouts, meta-labelled trades) are **per-trade**. Measuring a sparse edge per-bar either drowns it in zeros or inflates it by `1/sqrt(fraction-in-market)`. Per-trade is honest but data-hungry, so a sparse class leans on a minimum-trade-count gate and a bootstrap lower bound, never a lone point Sharpe.
- **The window must be able to contain the edge.** Trend and carry need long in-sample windows because their edge only appears across regimes; cross-asset relationships drift, so they get rolling windows. The window is a consequence of the class, never a knob.
- **The drawdown threshold encodes a belief.** A wide structural-MaxDD tolerance is correct for short-vol carry and dangerous everywhere else, which is exactly why it's bound to the class, not chosen per run.
- **Write the falsifier first.** A hypothesis you can't imagine failing is a hope. Name the death condition, and the deployment gate, before the data is touched, so a failing run is information instead of a prompt to overfit.
- **Pre-registration is a commitment device.** It binds these choices in writing against the optimistic future-you who will want to move the goalposts toward "pass."

With the edge classified, the hypothesis falsifiable, and the ruler pre-committed, the next chapter makes that ruler trustworthy: **a backtest you can trust** (see "Backtest You Can Trust") closes the five ways a measurement lies before any heavier machinery runs. From there, **walk-forward**

(see “Walk Forward”) shapes the out-of-sample experiment your class demands, and **beating your own optimiser** (see “Deflated Sharpe”) corrects for the searching you did to find the edge in the first place. The class you assign here also becomes a runtime contract later: see **the strategy-class contract** (see “Strategy Class Contract”).

6. A backtest you can trust

A backtest is a measurement. And like any measurement, it is worthless, worse than worthless, because it's *confident*, unless you know its units, whether the instrument was wired up correctly, and how big the error bars are. Most backtests fail all three checks silently, and the failure always points the same direction: it makes the strategy look better than it is.

This chapter is about closing those leaks before any of the heavier machinery (walk-forward, deflation, Monte Carlo) even runs. Get this layer wrong and everything downstream inherits the lie. Get it right and the rest of Part II is just turning up the rigour.

We'll go through the five ways a backtest lies to you, the single discipline that fixes each, why each fix works, and how Titan bakes those disciplines into one shared module so they can't be quietly skipped. Then, because Sharpe is only the *first* number, we'll look at the rest of the suite you actually need.

The five lies

#	The lie	What it looks like	The fix
1	**Wrong units**	A Sharpe computed on hourly bars, annualised as if daily	An explicit `periods_per_year` on every call
2	**Survivor math**	Dropping flat bars before annualising	Never filter `returns != 0` before a Sharpe
3	**Peeking**	The signal at bar t uses information from bar t	Shift discipline: trade t's signal on $t{+}1$'s return
4	**Future-normalised**	A z-score computed over the whole series	Causal (rolling / expanding) or IS-frozen normalisation
5	**No error bars**	A single Sharpe number, reported to two decimals	A bootstrap confidence interval, gated on the lower bound

None of these is exotic. Every one of them has shipped to production in real systems, including Titan, and every one is invisible unless you go looking.

Let's take them in order, and notice that **each lie flatters the strategy.** That asymmetry is the whole reason for suspicion: errors in a backtest are not random, they are *biased toward optimism*, because the optimistic versions are the ones that survive your attention and get deployed.

> **Note: A sixth lie lives one layer out: the fill**
> The five lies here all corrupt the *measurement* of a return series that already exists. There is a sixth that corrupts the **series itself**: the backtest booked a fill (at a price, a bar, and a cost) the market would never have given you, so every Sharpe, Calmar, DSR and ruin number downstream is measuring an edge that lived in the *fills*, not the signal. It belongs to execution rather than measurement, so it gets its own treatment in **The execution layer** (see "Execution And Costs"): next-bar-open vs same-bar-close, the half-spread you cross each side, stops that fill *past* the trigger, and a constructed cost model you re-gate the edge against. Keep it in mind as the sixth member of this list: the most expensive lie, because it sits upstream of every honest number you are about to learn to compute.

Lie 1, Units: annualisation is a correctness property, not a convention

The Sharpe ratio is a per-period quantity scaled to a year. The scaling factor is the number of bars in a trading year, and it depends entirely on your bar size:

```
BARS_PER_YEAR = {
    "D":  252,            # daily
    "H4": 252 * 6,        # 6 four-hour bars per 24h FX day
    "H1": 252 * 24,       # = 6048
    "M5": 252 * 24 * 12,  # = 72576
}
```

Why does the factor matter so much? Because Sharpe scales with the *square root* of the number of periods. Annualising per-bar Sharpe means multiplying by `sqrt(periods_per_year)`. Get the period count wrong by a factor of 24 (treating hourly data as daily) and your annualised number is off by `sqrt(24)` $\approx$ **4.9**$\times$. Nobody makes that error in the direction that makes the strategy look *worse*. The dangerous version is subtle: a pipeline that resamples mid-way and then annualises with the wrong frequency's factor.

Warning: War-story: the H1 strategy that was secretly daily
A backtest aggregated hourly signals into a once-a-day position, producing a P&L series that was, in substance, **daily**. But the Sharpe helper was still being handed the H1 annualisation factor (`252 * 24`). The strategy reported a Sharpe near 4 and looked like a licence to print money. The real, daily-annualised figure was around 0.8: a fine-but-ordinary strategy. The `sqrt(24)` $\approx 4.9\times$ phantom came entirely from annualising a daily series as if it were hourly. The fix wasn't cleverness; it was making the frequency *impossible to leave implicit*.
The positive rule for a mixed-frequency pipeline: annualise with the factor of the P&L's actual *decision* frequency, not the raw bar frequency. When signals fire on H1 but a position is held for days, the deciding frequency is daily. Confirm it by counting distinct position changes (or collapsing the series to one return per held position) and use *that* bar count, here 252, not `252 * 24`.

The fix is a rule, not vigilance:

Tip: Make the annualisation factor a required argument
Titan's shared metrics module refuses to compute a Sharpe without being told the frequency. There is no default.

```
def sharpe(returns, periods_per_year: int) -> float: ...
```

A missing default looks like a papercut. It's actually the cheapest correctness gate in the system: it forces every caller to *state, at the call site,* what frequency the P&L is, which means a reviewer can verify it in one glance. The first line of any backtest output should be the bar timeframe. If you can't state it, stop.

Lie 2, Survivor math: don't filter the flat bars

A tempting "cleanup" looks innocent:

```
# WRONG: makes a selective strategy look far better than it is
active = returns[returns != 0.0]
sr = sharpe(active, periods_per_year=252)
```

A strategy that's in the market only a fraction `a` of the time has its Sharpe inflated by `1/sqrt(a)` when you drop the flat bars and then annualise with the full-year factor. The intuition: removing the zeros leaves the *same* mean-per-active-bar but a *smaller* standard deviation per bar (you deleted

the calm days), and you still scale by the full year's bar count. A strategy that trades one day in four (`a = 0.25`) gets a free **2**× to its reported Sharpe. The flat days are not noise to be cleaned; they are information about the strategy's *selectivity*, and the annualisation factor already accounts for them.

If you genuinely want per-trade statistics (and for sparse strategies you often should), that's a different measurement with its own helper (`trade_sharpe`), fed a per-trade P&L series, not a zero-filtered bar series. The point is to make the choice explicit, never to silently delete inconvenient zeros.

Warning: This is the single most common Sharpe inflation
It survives code review because the filter reads as hygiene. Titan's rule: **no Sharpe function filters `returns != 0` internally**, and reviewers reject any caller that does it by hand. If the strategy is sparse, that's a fact the Sharpe should reflect, not one the metric should hide.

Warning: War-story: the window that deleted the crash
Zero-filtering is survivorship in the *rows*. Choosing a start date that skips the last crash is survivorship in the *window*. It is the same lie, applied to time, and a nastier one because the code is *correct*: the sample is the fraud, so there's nothing for a reviewer to flag. A 60/40 equity sleeve went to paper advertising a **$**-20.7-**$32.1%**; the excluded window contained the one event the drawdown number was supposed to survive.
And the flattery **grows with leverage**, because leverage's entire downside lives in the crash the window deletes. The S&P leg at 2× draws **$**-37.5-60.5-**$22%**) purely because its data began in **2009-09**. It *is* the post-GFC window, so it should have been trusted least, not most. The rule is unglamorous: **quote the FULL column, and treat every 2010-onward number as marketing** until you've seen what it does through a crisis.

Lie 3, Peeking: shift discipline

This is the one that has cost the most, across the most systems. The signal you compute at the close of bar *t* can only be acted on *after* bar *t*: so it earns the return from *t* to *t+1*, not the return *into t*. The mental model: **a decision and the return it earns must live in disjoint time windows, decision first.** Written as code, the only safe pattern is:

```
# A position decided at the close of t earns t -> t+1:
strat_returns = asset_returns * position.shift(1).fillna(0.0)
```

The bug, "same-bar collect", looks like this, and it is everywhere in regime and cross-sectional code:

```
# WRONG: `winner` is decided using close[t]; `ret` is also the return
↪  into close[t].
winner = momentum.idxmax(axis=1)          # uses information available
↪  only AT close t
strat  = ret.where(columns == winner)     # but `ret` already happened by
↪  close t
```

The position at t was chosen with information that includes bar t itself, then "earns" bar t's return. It is pure look-ahead, and because momentum signals are autocorrelated it manufactures a gorgeous, completely fake equity curve: the worst kind, because it looks *plausible*. The fix is mechanical: lag the decision before it touches a return.

```
winner_lag = winner.shift(1).fillna("CASH")   # decide on yesterday's
↪  info
strat = ret.where(columns == winner_lag)      # earn today's return
```

Danger: War-story: four leaks in one codebase
One audit of one regime-driven codebase found the same-bar pattern in **four separate places**, each one a signal series multiplied by a contemporaneous return. None had been caught in review, because each looked like ordinary pandas. Collectively they turned a flat strategy into a stellar one. The rule Titan now enforces: **any series that multiplies a return must be `.shift(1)`'d first** unless you can *prove* the position was knowable strictly before the return's window opened. Treat same-bar `position * return` as guilty until proven innocent.

Warning: The shift knife cuts both ways: don't over-lag
Lie 3 says lag the decision before it earns a return, but "shift everything that touches a return," applied literally, introduces the *opposite* bug. **Double-lagging** (shifting a signal that was already lagged, or `.shift(1)`-ing a position *and* also filling at next-bar-open) separates the decision and the return by a *gap*, so the position decided at `t` earns the return from `t+1→t+2`: it throws away a bar of real edge and biases the result **pessimistic**, making you discard strategies that actually work. The invariant is tighter than "shift it": **the decision window**

and the return window must be *adjacent and disjoint* (back-to-back, no gap, no overlap). One lag, not zero (look-ahead, which flatters) and not two (lost edge, which buries). Both errors are real; only the adjacent-and-disjoint version is correct.

Lie 4, Future-normalised features

Standardising a feature is routine, and the routine version is look-ahead by construction:

```
# WRONG: mean and std are computed over the ENTIRE series, including the
 ↪  future
z = (x - x.mean()) / x.std()
```

At every historical bar, that z-score "knows" the mean and standard deviation of data that hadn't happened yet. It's a subtle leak because it doesn't touch returns directly; it poisons the *feature*, and the damage flows downstream into whatever signal the feature feeds. The fix is to normalise *causally*, using only data available at each point, or, inside a walk-forward, to **freeze** the normalisation statistics on the in-sample window and apply them unchanged out-of-sample:

```
z_causal    = rolling_zscore(x, window=252)          # only past data at
 ↪  each bar
z_expanding = expanding_zscore(x, min_periods=252)   # all past data,
 ↪  growing window
z_wfo       = is_frozen_zscore(x, is_end=fold.is_end) # IS stats, applied
 ↪  to OOS
```

Titan's metrics module deliberately **does not offer** a full-series z-score. It can't be called by accident because it doesn't exist; the only z-scores available are the causal and IS-frozen ones. That's a recurring pattern in this book: *the safest API is one where the dangerous operation is simply absent.*

Warning: The warmup prefix has to be dropped from *both* series
A windowed z-score is NaN for its first `window` (or `min_periods`) bars: there isn't enough history yet to standardise. That warmup prefix has to be dropped from the signal **and** the aligned P&L series *together*. Drop it from one but not the other and you have silently re-introduced a 1-bar offset, which is Lie 3 wearing a different hat, or a length mismatch that pandas will quietly broadcast around. The safe idiom is

to compute the z-score, then slice the signal and the returns to their common, non-NaN index *before* anything multiplies a return.

Note: Feature discipline beyond leakage: what you should be z-scoring
Lie 4 keeps the *future* out of a feature's normalisation, but two feature pitfalls remain that aren't look-ahead, so the causal check sails right past them. **Stationarity vs memory (over-differencing):** a model wants stationary inputs, and the reflex is to difference a series until it is, but each difference discards *memory*, the slow-moving level a forecast often lives on, so over-differencing strips the signal while making the statistics look clean. Difference only as far as a stationarity test demands (fractional differencing keeps more memory at the same stationarity), and z-score *that*, not the raw level or the over-differenced noise. **Collinearity:** two near-duplicate features don't add information, they add instability; a regression or tree splits the shared signal between them arbitrarily and each one's "importance" becomes noise. Check feature correlation and either drop the redundant one or orthogonalise the predictors (see "Combining Forecasts") before they reach the model, the same residualisation the forecast-combination layer uses. Both are *engineering* failures, not leakage, but they corrupt the forecast just as surely. And feature *search* across many candidates is itself a multiple-testing inflator (see "Deflated Sharpe") that must feed the deflated-Sharpe trial count.

Lie 5, No error bars: a Sharpe is an estimate, not a fact

A backtest Sharpe of, say, 1.1 is a point estimate from one particular history, one draw from the distribution of histories the world could have produced. Reported alone, to two decimals, it implies a precision it does not have. The honest version is an interval, and the cheapest honest interval is a bootstrap: resample the return series many times, recompute the Sharpe on each resample, and read the 2.5th and 97.5th percentiles.

```
lo, hi = bootstrap_sharpe_ci(
    returns,
    periods_per_year=252,
    n_resamples=1000,   # more resamples -> tighter percentile estimates
    block_size=block,   # <- see the warning below
    seed=42,            # reproducible: same input, same interval
)
```

Then the deployment rule is brutally simple and applied everywhere in Titan:

> **If the 95% lower bound of the Sharpe is ≤ 0, the strategy is `unconfirmed` and cannot enter a default deployment registry, regardless of how good the point estimate looks.**

A point estimate of 1.1 with a 95% interval of `[-0.2, 2.4]` is not a 1.1-Sharpe strategy; it's a coin flip with a good story. The lower bound is the number that decides capital, because it is the honest answer to *"how bad could this plausibly be?"*

And because it decides capital, resample it harder than you would the point estimate. The `n_resamples=1000` above is fine for the centre of the distribution but light for the 2.5th-percentile tail you actually gate on: that quantile is the noisiest part of any bootstrap. When the lower bound decides deployment, push to 5,000–10,000 resamples so the gate itself isn't a coin flip.

Note: Gate the sleeve to admit it; gate the book to size it
The rule above runs on one return series, but a real book is several sleeves at once, so apply it at *both* levels. Each sleeve must clear `CI_lo > 0` on its own returns to enter the registry, and the aggregate book equity must clear it too. The two checks are not redundant: correlated sleeves can each pass individually yet stack into an aggregate whose lower bound is *worse* than any single sleeve's, because correlation concentrates the tail rather than diversifying it away. Gate the sleeve to decide what is allowed in; gate the book to decide how much of it you can hold.

Warning: War-story: the bootstrap that lied about itself
The naive (IID) bootstrap resamples individual bars independently, which **destroys** the autocorrelation that trend and carry strategies live on. That narrows the interval and biases the *lower* bound **upward**: exactly the optimism you're trying to avoid, applied to the exact number you gate on. An external audit of Titan flagged this: strategies were passing the `CI_lo > 0` gate partly because the CI was artificially tight. The fix is a **stationary block bootstrap** (Politis & Romano, 1994): resample *blocks* of consecutive bars (geometric-mean length matched to the strategy's autocorrelation) so serial dependence survives the resampling, and the lower bound tells the truth. Pass a `block_size`; don't accept the IID default for a serially-correlated strategy.

Tip: How long a block? Span the decorrelation horizon
Pick the block so one block covers roughly the strategy's decorrelation horizon: order-of-magnitude its holding period, or the lag at which the return autocorrelation first falls below ~0.05. *(Illustrative:* a daily trend sleeve holding ~2–4 weeks wants a block of ~10–20 trading days.) Too short and serial dependence still leaks out across block boundaries, putting you back where the IID bootstrap left you; too long and you have too few independent blocks to resample, so the interval gets noisy. The stationary bootstrap then randomises the block length around that geometric mean, so the answer isn't brittle to one exact number.

Sharpe is necessary, not sufficient

Everything above makes *Sharpe* trustworthy. But Sharpe answers exactly one question, return per unit of *volatility*, and it is blind to the questions that actually decide whether you can live with a strategy:

- It treats upside and downside volatility identically. A strategy that occasionally spikes *up* is penalised the same as one that occasionally craters. **Sortino** fixes this by dividing by downside deviation only.
- It says nothing about the *path*. Two strategies with identical Sharpe can have wildly different worst drawdowns and time-to-recovery. **Calmar** (CAGR over max drawdown) and the **max-drawdown** geometry capture what a human actually experiences holding the thing.
- It is a whole-distribution average, so it under-weights the tail that ends you. **CVaR / CDaR** (the *average* loss in the worst slice, not just a single quantile) and a formal **risk of ruin** speak to survival, not smoothness.

Warning: War-story: the 1.4-Sharpe strategy nobody could hold
A candidate posted a Sharpe around 1.4, past every gate above, yet its drawdown ran deep into the double digits and took over a year to recover: the kind of trough a strategy gets switched off at the bottom of. That is why **Calmar lift, not Sharpe lift, is the primary promotion metric**; the full telling is in Beyond Sharpe: the metric suite (see "Metric Suite").

Crucially, *all* of these metrics are subject to the same five lies: wrong units, survivor math, peeking, future-normalisation, and no error bars. A Calmar computed on a look-ahead equity curve is exactly as worthless as a Sharpe.

So the disciplines in this chapter are not "Sharpe rules"; they are *measurement* rules, and they apply to every number you report.

The full battery, Sortino, Calmar, geometric CAGR, CVaR/CDaR, gets its own treatment in **Beyond Sharpe: the metric suite** (see "Metric Suite"), and turning tail risk into a survival probability at deployed size is the subject of **Tail risk & risk of ruin** (see "Tail Risk And Ruin").

Why Titan puts all of this in one module

Each fix is a one-liner. The reason they hold over hundreds of research scripts is that **none of them is reimplemented locally.** Every Sharpe, Sortino, Calmar, volatility, z-score, and annualisation in the codebase routes through a single shared metrics module, and the module is written so the wrong thing is hard or impossible:

- `sharpe(...)`, `ewm_vol(...)`, `calmar(...)`, `sortino(...)` **require** `periods_per_year`: no default.
- No Sharpe filters zeros internally.
- Only causal and IS-frozen z-scores exist; the full-series version is absent.
- Edge cases (empty, constant, NaN series) return `0.0`/`NaN` rather than raising, so a guardrail never crashes a batch; it just refuses to flatter you.

The alternative, every researcher writing `def _sharpe(r): return r.mean()/r.std()*np.sqrt(252)` at the top of their notebook, guarantees that the lies reappear, independently, forever. A shared module turns "remember to be careful" into "you literally cannot call it the unsafe way." That trade, a tiny bit of ceremony at the call site for a class of bugs that can't recur, is the whole philosophy of this book in miniature.

Example: What 'stating the timeframe' buys you
Put the bar frequency at the top of every backtest report. It sounds trivial. But the act of writing `# P&L frequency: H1` forces you to confirm the annualisation factor (`252 * 24`), which is also the moment you'd notice a mid-pipeline resample, a zero-filter, or a daily factor on hourly data. One disciplined comment catches three of the five lies at once.

Verify your verifier

Everything so far hardens the *backtest*. But the backtest is produced by code, and that code is exactly as fallible as the strategy code it audits, with one cruel twist: **a bug in the verification machinery flatters the very number you deploy on.** The measurement layer needs its own measurement, and it comes in two habits.

Independently reimplement the load-bearing cell. Find the single number the decision actually turns on (the one cell whose *sign* you would bet the account on) and reimplement it from scratch, in code that shares nothing with the original, then demand the two agree *to the basis point*. When a levered-allocation study's entire recommendation rested on one cell (the S&P leg, 2× leverage, full history), an independent from-scratch engine reproduced its CAGR, volatility and max-drawdown to **0.00pp**, and its margin-call count to zero. A separate study's independent engine matched the shared unlevered-60/40 cell to **+0.029pp CAGR / +0.006 Sharpe**. Agreement at that tolerance is the only thing that upgrades "the code says so" into "the number is real"; disagreement means one of the two is wrong and you don't yet know which.

Audit adversarially: assume a bug until proven innocent. Not "does this look right?" but "*where* is this wrong?" One week of that posture on this desk caught **four separate conclusion-flipping bugs**, every one of them flattering:

- A table labelled **"FULL, GFC-inclusive"** of which only 2 of its 5 windows genuinely were: two global indices started *mid-2008*, capturing the crash but *understating* it (they miss the 2007 peak), and a third began in *2009-09*, after the crash entirely. All three printed beside the two genuinely-2002 lines as if comparable. The window bias the study existed to warn about, committed *by* the study.
- A Sharpe computed **raw** (risk-free = 0), which inflates low-vol cells most and so *doubled* the apparent leverage penalty: the drop read as **0.15** of Sharpe when the honest excess-of-cash drop was **0.08**.
- A counterfactual whose headline rested on a branch of **dead code**: the central number was quoted in the writeup but never actually computed.
- A **sleeve-level** simulation whose output was used to make a **book-level** claim, silently conflating one strategy's equity with the whole portfolio's.

And a fifth, found the same week one chapter over: the deflated-Sharpe gate (see "Deflated Sharpe") was fed an *annualised* Sharpe into a variance term that assumes a *daily* (per-observation) one, inflating its z-statistic roughly

tenfold and pinning the pass-probability near **1.000**. The gate that exists to catch overfitting had been rubber-stamping every winner. It is the purest instance of the twist: the flaw lived in the *verifier*, so it silently loosened the exact bar the deployed strategies had to clear.

Danger: The rule: the verifier is guilty until proven innocent
The load-bearing number gets an independent reimplementation that must agree to the basis point, and every audit opens by *assuming* a bug, not hunting for one. Five for five, the bugs pointed the same way the five lies do: toward *passing*. That is not coincidence. A verification bug that made a strategy look *worse* gets found instantly: someone chases the disappointing number until it's explained. A bug that makes it look *better* gets deployed. Selection pressure curates your verifier's bugs for optimism exactly as it curates the strategy's.

Exercises

1. **Spot the lie.** A colleague reports a Sharpe of 4.0 on an *hourly* strategy, annualised with `periods_per_year=252`, after filtering out the flat bars. Which two of the five lies are present, and roughly how inflated is the number?

   Answer: Answer
   Lie 1 (wrong units): hourly data annualised as daily inflates Sharpe by `sqrt(24) ≈ 4.9×`. **Lie 2 (survivor math):** dropping flat bars inflates by `1/sqrt(active fraction)`. Both compound and both flatter: the real figure is a small fraction of 4.0.

2. **The sixth lie.** The five lies are all about measuring a return series. Name the sixth, which corrupts the series *before* any metric runs, and one of its faces.

   Answer: Answer
   The **fill**: the backtest booked a price/timing/cost the market wouldn't have given you. Faces include next-bar-open vs same-bar-close, the half-spread crossed each side, and stops filling ~1.5R *past* the trigger. See the execution layer (see "Execution And Costs").

Takeaways

- A backtest is a measurement: it needs **units** (`periods_per_year`), **causality** (shift discipline + causal normalisation), and **error bars** (a bootstrap CI you gate on).
- The lies all point the same way: they flatter the strategy. Assume any un-audited number is inflated until you've checked all five.
- **Gate on the lower bound, not the point estimate.** `CI_lo ≤ 0` ⇒ unconfirmed, full stop. And use a *serially-aware* bootstrap, or the lower bound lies too.
- **Sharpe is the entry point, not the verdict.** Drawdown geometry (Calmar), downside risk (Sortino), and the tail (CVaR/CDaR, risk of ruin) decide whether a strategy is *livable*, and they obey the same measurement rules.
- Centralise the metrics so the unsafe version can't be written. The safest API is one where the dangerous operation simply doesn't exist.

This chapter fixed the *measurement*. The next chapters harden the *experiment*: **Beyond Sharpe: the metric suite** (see "Metric Suite") builds out the battery of numbers; **Walk-forward that's actually out-of-sample** (see "Walk Forward") makes sure you're not testing on data you trained on; and **Beating your own optimiser** (see "Deflated Sharpe") corrects for the fact that the more parameters you try, the better your best result looks by pure chance.

7. Beyond Sharpe: the metric suite

A Sharpe ratio answers one question, *return per unit of volatility*, and answers it well. The trouble is that it's the only question most pipelines ever ask, and it is silent on the questions that actually decide whether you can hold a strategy through a bad year: How deep does it dig? How long does it stay underwater? When it loses, does it lose a little or does it lose everything? A strategy can ace the Sharpe gate and still be one you'd switch off at the worst possible moment, locking in the loss the average smoothed over.

A backtest you can trust (see "Backtest You Can Trust") made the Sharpe number *honest*. This chapter makes it *sufficient*, by surrounding it with the four metrics that catch what it misses: Sortino for asymmetry, Calmar (over geometric CAGR) for drawdown geometry, and CVaR/CDaR for the tail that ends you. Then it states the rule those metrics bought, **Calmar lift, not Sharpe lift, gates promotion**, and ends with the warning that ties the whole part together: every one of these numbers obeys the same five measurement lies.

The principle: one number cannot describe a path

Two strategies can have an identical Sharpe and feel nothing alike to hold. Sharpe is a property of the *return distribution* (mean over standard deviation) and it throws away the *order* of the returns. But order is exactly what a human (and a risk committee, and a margin engine) experiences. A strategy that earns its return as a smooth ramp and one that earns the same return via a 40% crater followed by a heroic recovery have the same mean and the same volatility. They do not have the same Calmar, the same time-to-recovery, or the same probability of getting switched off at the bottom.

So the discipline is to report a **vector**, not a scalar. Each component answers a different question, and a strategy has to clear all of them, because they fail independently. Here is the suite Titan computes for every candidate, and the question each one is the *right* tool for:

Metric	Answers	Why Sharpe can't	Family
Sharpe	Return per unit of total volatility	n/a (the baseline)	Risk-adjusted return
Sortino	Return per unit of *downside* volatility	Sharpe penalises upside spikes identically to crashes	Risk-adjusted return
Calmar	Return per unit of *worst drawdown*	Sharpe ignores the path entirely	Drawdown geometry

Metric	Answers	Why Sharpe can't	Family
Max drawdown	The single worst peak-to-trough	A distribution average hides the worst point	Drawdown geometry
CVaR (Expected Shortfall)	*Average* loss in the worst α-tail of bars	Sharpe under-weights a fat left tail	Tail risk
CDaR	*Average* depth of the worst α-tail of drawdowns	MaxDD reports only one point on the curve	Tail risk

Note: Where each metric lives in the book
This chapter owns the *return-and-drawdown* battery. Turning the tail into a **survival probability at deployed size** (risk of ruin, and the Monte-Carlo `P(MaxDD > x%)` constraint) is the job of Tail risk & risk of ruin (see "Tail Risk And Ruin"). Turning a Sharpe/Kelly estimate into a *position size* you can survive is Position sizing: Kelly & vol-targeting (see "Position Sizing Kelly"). Read this one for the diagnostics; those two for the decisions they feed.

The suite is curated to roughly one metric per failure mode, not exhaustive. We omit the Ulcer Index and pain ratio (CDaR is our chosen drawdown-tail metric and carries the same path information), MAR and Sterling (Calmar variants that differ only in the drawdown window), and Omega (a threshold-integral that added no decision over Sortino plus CVaR in our use). If a sixth metric does not answer a question the five leave open, it earns nothing but a longer report to read.

Sortino: stop punishing the good surprises

Sharpe divides excess return by the standard deviation of *all* returns. Standard deviation is symmetric: a +5% day and a $-5% day move it by the same amount. But you do not feel them the same way, and you should not penalise them the same way. A strategy whose volatility comes from occasional sharp *gains* is being unfairly marked down by Sharpe for the crime of making money in lumps.

Sortino fixes this by replacing the denominator with **downside deviation**, the standard deviation computed only over returns below a target (usually zero):

```python
def sortino(returns, periods_per_year, *, target=0.0, ddof=1):
    s = _as_series(returns).dropna()
    if len(s) < 20:
        return 0.0
    downside = s[s < target]            # only the bad bars enter the
↪   denominator
    if len(downside) < 5:
        return 0.0
    dsd = downside.std(ddof=ddof)
    if dsd < 1e-12:
        return 0.0
    return (s.mean() - target) / dsd * math.sqrt(periods_per_year)
```

Two design choices in that snippet are worth lifting out, because they recur across the suite:

- **It requires `periods_per_year`.** No default. Sortino is annualised by the same `sqrt(periods_per_year)` factor as Sharpe, and it is just as wrong if you hand H1 returns the daily factor. Lie #1 from the previous chapter applies verbatim.
- **It refuses to lie on thin data.** Fewer than 20 samples, or fewer than 5 downside bars, returns `0.0` rather than a flattering ratio computed from three bad days. A near-empty denominator is how Sortino blows up to a meaningless 14.0 and someone deploys it.

The Sortino/Sharpe *gap* is itself a diagnostic. If Sortino is much higher than Sharpe, the strategy's volatility is mostly upside, which is good. If they're close, volatility is symmetric. If Sortino is *lower*, your "risk" is disproportionately to the downside: a quiet warning that the smooth Sharpe is hiding a left-skewed return stream.

Calmar and the geometric CAGR you must not skip

Calmar is return divided by the worst drawdown: `CAGR / |MaxDD|`. It is the most honest single-number summary of *livability*, because the denominator is the exact quantity that gets a strategy killed mid-trough. A Sharpe of 1.4 tells you nothing about whether you'll have to sit through a 35% hole to collect it. Calmar puts that hole in the denominator.

There is one trap in computing it, and Titan got bitten by it badly enough that the framework now has two separate implementations on purpose.

Warning: Arithmetic mean is not your compounded return
The convenience Calmar in the shared metrics module uses an **arithmetic** annualised return (`mean(returns) * periods_per_year`), which is fine for quick diagnostics. It is **wrong** for the number that gates promotion. Arithmetic mean overstates what you actually compound by roughly `vol² / 2` per year (the volatility drag). For a strategy with non-trivial volatility over a multi-year window, that overstatement is large enough to flip a promotion decision. The fix is a **geometric CAGR**: build the equity curve, then take its true annualised growth rate.

```python
def compute_cagr(returns, periods_per_year):
    s = pd.Series(returns).dropna()
    n = len(s)
    if n < MIN_BARS_FOR_CALMAR:
        return 0.0
    eq_final = float((1.0 + s).prod())   # the equity curve's terminal
↪   value
    if eq_final <= 0.0:
        return -1.0                      # total wipe-out: treat as
        ↪  catastrophic
    return eq_final ** (periods_per_year / n) - 1.0   # GEOMETRIC,
     ↪  compounds correctly
```

The geometric form asks "what constant annual rate would have produced this terminal equity?", the rate you would actually have earned. The arithmetic mean asks "what's the average bar?" and silently ignores that a $-$50% bar needs a +100% bar to undo it. The two diverge in exactly the direction that flatters a volatile strategy, which is why Titan reserves the geometric CAGR for the promotion Calmar and never the other way around.

Example: The arithmetic–geometric gap, in numbers (illustrative)
A volatile candidate over a multi-year window draws down 35%. Its *arithmetic* annual return reads **18%**, so the convenience Calmar is `0.18 / 0.35 ≈ 0.51`, which is respectable, and it clears a `+0.10` lift gate against an incumbent sitting at 0.40. But the volatility drag means the *geometric* CAGR (the rate you actually compounded) is only **11%**, so the true Calmar is `0.11 / 0.35 ≈ 0.31`, *below* both the incumbent and the gate. Same strategy, same hole; the arithmetic mean would have promoted a strategy the geometric truth rejects. (Figures illustrative; the point is the *direction and size* of the gap, which always flatters the volatile candidate.)
Three caveats keep this gate honest. **The denominator drifts with history length:** `|MaxDD|` is a single sample-path extreme, so a longer

backtest tends to find a deeper worst-case and *lower* the Calmar, which is why the promotion compares Calmars over a *fixed rolling window* ($\approx$ one year, illustrative) rather than re-rating every strategy as its history grows; it is the same "one point hides the pattern" weakness the CDaR section below raises, now sitting in your gate's denominator. **The denominator also scales with volatility, so two candidates at a *fixed weight* are being compared at *unequal risk*:** a deeper drawdown is partly just the mechanical consequence of higher vol, and the naive comparison penalises the higher-vol name for it. On one GFC-inclusive window, at a fixed 60/40, the S&P 600 (IJR) ran **14.06%** vol against the S&P 500 (SPY)'s **11.38%**, and looked worse on drawdown (**$**-33.22-31.14-**$27.13%**, now *shallower* than SPY's. The fixed-weight comparison was genuinely unfair. (Note every figure in that sentence comes from the *same* window. An earlier draft of this very paragraph quoted IJR's full-history vol and drawdown against SPY's from a shorter common window, comparing the unfairness on one sample and the fix on another, which is the window splice (see "Backtest You Can Trust") this book spends a chapter condemning. It is remarkably easy to do while writing *about* it.) The decision held anyway, but note *how*: at matched vol SPY's Sharpe is **0.790** against global ACWI's **0.614**, a gap whose bootstrap CI excludes zero, and against IJR's **0.671**, a gap whose CI does *not* (`[-0.077, +0.335]`). Against IJR the honest claim is *"no evidence of rescue"*, not *"rescue disproven"*, which is this chapter's own lower-bound rule, applied to its own verdict. **Re-weight to matched risk before you compare drawdowns: a lower-vol name can look safer purely because it took less risk. And the gate assumes a single, pre-registered candidate:** screen N strategies and the best-of-N clears a `+0.10` lift and `CI_lo > 0` by chance, so both must be deflated for the trial count (see Beating your own optimiser (see "Deflated Sharpe")).

Note the `eq_final <= 0.0` branch: a strategy that wipes out the account returns CAGR `-1.0`, not a `NaN` or a divide-by-zero. The metric is built so the catastrophic case produces a *catastrophic number*, not an exception that gets swallowed in a batch run.

Warning: War-story: the 1.4-Sharpe strategy nobody could hold
A candidate posted a Sharpe around **1.4**, clean past every gate in the previous chapter, with a serially-aware bootstrap lower bound comfortably above zero. On the Sharpe axis it was a clear promote. Then we looked at the *path*. Its peak-to-trough drawdown ran deep into the double digits and stayed underwater for **well over a year** before recovering. The Sharpe never showed it, because Sharpe averages over the path and the path was the problem.

The arithmetic vs geometric trap made it worse: the convenience Calmar, fed an arithmetic annual return, still looked respectable, because the arithmetic mean papered over the volatility drag that the long trough represented. Only the **geometric** CAGR over `|MaxDD|` told the truth: a Calmar low enough that no human or risk committee would have kept funding the strategy through the hole. It would have been switched off at the bottom, converting a paper drawdown into a realised loss.

The rule this bought: **Calmar lift (computed on geometric CAGR) is the primary promotion metric.** Sharpe is retained only as a secondary no-regression check. A strategy that improves portfolio return-per-volatility but worsens return-per-drawdown is not an improvement; it is a future kill-switch event with good marketing.

CVaR and CDaR: average the tail, don't sample it

Value-at-Risk asks "what's the loss at the 5th percentile?", a single threshold. The problem is that VaR tells you nothing about what's *beyond* the threshold. A $-$3% 5%-VaR is the same whether the worst-5% bars average $-$3.1% or $-$30%. The tail past the cutoff is exactly where ruin lives, and VaR is blind to it by construction.

CVaR (Conditional VaR, a.k.a. Expected Shortfall) fixes this by averaging the *whole* bad tail instead of reading a single point on it:

```
def cvar(returns, *, alpha=0.05):
    s = _as_series(returns).dropna()
    n = len(s)
    if n < 20:
        return 0.0
    k = max(1, int(np.ceil(n * alpha)))   # how many bars are "the worst
↪   alpha"
    worst = np.sort(s.to_numpy())[:k]      # the k most-negative returns
    return float(worst.mean())            # their AVERAGE, not their
     ↪  threshold
```

`cvar(returns, alpha=0.05)` is the mean of the worst 5% of bars. It is *coherent* (it respects diversification in a way VaR does not) and it is the number that actually answers "when it goes wrong, how wrong, on average?"

CDaR (Conditional Drawdown-at-Risk) is the same idea applied to the drawdown curve instead of the bar returns. MaxDD is a single point, the deepest hole. But a strategy that visits $-$15% *once* and a strategy that visits $-$12% to $-$15% a *dozen times* have nearly identical MaxDD and wildly different

lived experience. CDaR averages the worst α-fraction of drawdown depths, so the "many medium holes" pattern that MaxDD hides shows up:

```
def cdar(returns, *, alpha=0.05):
    s = _as_series(returns).fillna(0.0)
    eq = pd.concat([pd.Series([1.0]), (1.0 +
↪   s).cumprod().reset_index(drop=True)])
    dd = (eq - eq.cummax()) / eq.cummax()  # the full drawdown path
    k = max(1, int(np.ceil(len(s) * alpha)))
    worst = np.sort(dd.to_numpy())[:k]     # the deepest alpha-fraction
↪   of the path
    return float(worst.mean())
```

Note the equity curve is anchored at `1.0` *before* the first bar, the same convention as `max_drawdown`. A first-bar loss is a real drawdown from the starting peak, not a zero because "the peak was never below 1.0." Skip that anchor and you under-report exactly the early drawdowns a new live strategy is most likely to hit.

> **Tip: Read CVaR and CDaR as a pair, not in isolation**
> CVaR is about the *bars* (how bad is a bad day on average); CDaR is about the *path* (how deep are the deep holes on average). A strategy can have a tame CVaR (no individual day is catastrophic) and an ugly CDaR, because it bleeds in long correlated runs. That combination is the signature of a slow-grind drawdown, the kind that's hardest to hold and easiest to switch off at the worst time. Use the α you actually fear; 5% is a starting convention, not a law.

Three things will bite you wiring these two into a dashboard or an alert, and none is visible from the formula alone:

- **Both return negative decimals; more negative is worse.** `cvar` lives in per-bar return space, `cdar` in drawdown-depth space, and both report the *average* of the most-negative slice, so a CVaR of `-0.03` is a worse tail than `-0.01`. Gate on the magnitude *with the sign kept*; never `abs()` them in alert logic, or your worst-case alarm fires backwards, treating the calmest tail as the most dangerous one.
- **The `n < 20` guard is far too loose for a tail average.** It is calibrated for whole-sample moments (mean, std), but CVaR and CDaR average only `ceil(n·α)` observations. At α = 5% a 250-bar year is the mean of ~13 bars, and a short live window collapses to the average of two bad days, almost as point-like as the VaR this section just criticised, and the same way Sortino blows up to a "meaningless 14.0" on a near-empty denominator. Treat a 5% tail as unstable below a few hundred bars: widen α (10–20%) on short windows, or report the worst-`k`

count alongside the value so a thin tail can't masquerade as a stable estimate.

- **Everything here is per-bar.** On a sparse-trade strategy most bars are flat zeros, which dilutes the tail: a per-bar CVaR can read benign while the per-trade loss distribution is brutal. Compute these on the per-trade return series for selective strategies (the source ships a separate `trade_sharpe` for exactly this reason; see the cadence trap (see "Typology")), or the flat zeros will flatter the tail the same way they flatter a per-bar Sharpe.

The promotion gate: Calmar lift is the verdict

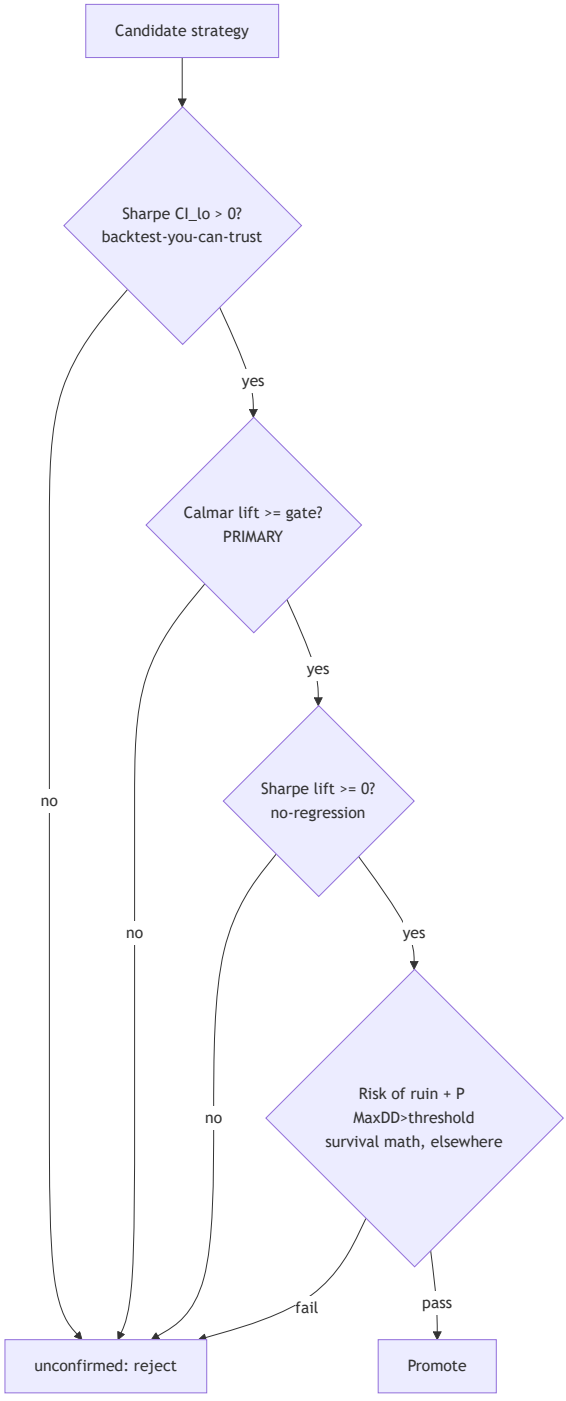
Here is where the suite stops being diagnostic and starts deciding capital. In Titan, a new strategy is never promoted on its standalone metrics. It is promoted on whether it makes the *existing portfolio* better, measured primarily by **Calmar lift**, the change in the combined book's return-per-drawdown.

```
def evaluate_promotion(proposed_returns, current_returns,
↪  periods_per_year, *,
                       calmar_lift_gate=0.10, sharpe_lift_gate=0.0):
    proposed_calmar = compute_calmar(proposed_returns, periods_per_year)
↪  # geometric CAGR
    current_calmar  = compute_calmar(current_returns,  periods_per_year)
    proposed_sharpe = sharpe(proposed_returns,
↪  periods_per_year=periods_per_year)
    current_sharpe  = sharpe(current_returns,
↪  periods_per_year=periods_per_year)

    cal_lift = proposed_calmar.calmar - current_calmar.calmar
    shp_lift = proposed_sharpe - current_sharpe

    passes_cal = cal_lift >= calmar_lift_gate   # PRIMARY: must improve
↪  drawdown-adj return
    passes_shp = shp_lift >= sharpe_lift_gate   # SECONDARY: must not
↪  regress Sharpe

    reasons = []
    if not passes_cal:
        reasons.append(f"calmar lift {cal_lift:+.3f} < gate
↪  {calmar_lift_gate}")
    if not passes_shp:
        reasons.append(f"sharpe regressed by {shp_lift:+.3f}")
    return PromotionVerdict(passes=passes_cal and passes_shp,
                            cal_lift=cal_lift, shp_lift=shp_lift,
↪  reasons=reasons)
```

Note the `reasons` list. The booleans decide; the strings are what an operator actually reads. At 3am, "promotion rejected" is useless and `calmar lift +0.04 < gate 0.10` is actionable. A gate that can only say *no* without saying *why* gets overridden by the first person who is tired enough, which is how a discipline quietly dies. Make every reject carry its own explanation.

The asymmetry is the whole point. Calmar lift is the *primary* gate, with a real positive threshold (`+0.10` here is illustrative; pick the lift you actually require). Sharpe lift is *secondary*, gated only at `>= 0`, a no-regression check, not a promotion driver. A candidate that raises Sharpe but flattens Calmar fails. A candidate that raises Calmar without hurting Sharpe passes. We deliberately put the burden of proof on the metric that maps to *getting switched off*, not the one that maps to *looking smooth*.

And, this is critical, these two conditions are **not the whole gate.** They are the framework-level slice. Full promotion eligibility additionally requires the joint **risk-of-ruin** bound and a Monte-Carlo `P(MaxDD > threshold)` constraint cleared elsewhere; the orchestrator combines all four. The metric suite proposes; survival math disposes. We come back to those two survival constraints in Tail risk & risk of ruin (see "Tail Risk And Ruin"), and to how the whole promote/reject decision is structured in The sanctuary decision matrix (see "Sanctuary Decision Matrix").


Candidate strategy
Sharpe CI_lo > 0?
backtest-you-can-trust
yes
no
Calmar lift >= gate?
PRIMARY
yes
no
Sharpe lift >= 0?
no-regression
yes
no
Risk of ruin + P
MaxDD>threshold
survival math, elsewhere
fail
pass
unconfirmed: reject
Promote

All of these obey the same five lies

The disciplines from the previous chapter (see "Backtest You Can Trust") were not "Sharpe rules." They were *measurement* rules, and every metric in this chapter inherits them. A Calmar computed on a look-ahead equity curve is exactly as worthless as a look-ahead Sharpe; arguably more dangerous, because "we also checked drawdown" buys false confidence.

The lie	How it corrupts the *suite* (not just Sharpe)
Wrong units	Sortino and Calmar are annualised by `sqrt(periods_per_year)` and `periods_per_year` respectively: *two different exponents, so two different-sized lies.* Hand them an hourly series with the daily factor and Sortino is off by `sqrt(24) ≈ 4.9×`, while Calmar is off by roughly an **order of magnitude**: its CAGR numerator scales ~linearly in `periods_per_year`, not by its square root ($\approx 24\times$ in the small-rate limit; exactly how much depends on terminal equity and bar count). The bigger lie lands on the drawdown gate.
Survivor math	Drop the flat bars and you shrink the denominator of Sortino and inflate CAGR; drawdown depth changes too, because you deleted the calm recovery days.
Peeking	A `position * same-bar-return` leak manufactures a smooth, plausible equity curve. Calmar, CDaR, CVaR are all computed *from that curve*; they will happily report a beautiful, fake drawdown profile.
Future-normalised	A full-series z-score feeding the signal poisons every downstream metric identically. The damage is in the returns; the suite just measures it.
No error bars	A point-estimate Calmar of, say, 2.1 from one history is no more a fact than a point Sharpe. The framework bootstraps a Calmar CI for exactly this reason.

That last row deserves its own emphasis, because the Calmar bootstrap has

a known limitation Titan documents in the source rather than hiding, one of several catalogued in the failure-mode catalogue (see "Failure Mode Catalogue").

Warning: War-story: the IID Calmar CI that under-states its own error
Titan's `bootstrap_calmar_ci` uses an **IID (block-of-1) bootstrap**: it resamples individual bars independently. That destroys serial correlation, which means for trend and carry strategies it *narrows* the confidence interval and biases the lower bound *upward*, the same optimism the Sharpe bootstrap audit (see "Backtest You Can Trust") flagged. The honest move is to document the limitation at the point of use and route the *decision* through the serially-aware path: the joint MC ruin/drawdown gate runs a **block** Monte-Carlo that preserves dependence. The IID Calmar CI is acceptable for the cheap "does Calmar lift include zero?" screen; it is **not** the number you'd gate ruin on. The fix wasn't to delete the IID version; it was to write down, in the docstring, exactly which question it's allowed to answer.

Exercises

1. **Why Sharpe isn't the verdict.** Two strategies post an identical Sharpe of 1.2. Name two questions Sharpe is structurally blind to, and the metric that answers each.

   Answer: Answer
   The *path* (worst-drawdown depth and recovery time → **Calmar** / max-drawdown geometry) and the *downside tail* (average loss in the worst slice → **CVaR / CDaR**). Sharpe also marks an upside spike identically to a crash → **Sortino** divides by downside deviation only.

2. **The promotion metric.** Why does the book promote on *Calmar lift*, not *Sharpe lift*, and why must Calmar use a **geometric** CAGR?

   Answer: Answer
   Drawdown, not volatility, is what gets a strategy switched off at the bottom, so Calmar (return per unit of pain) is the livability gate. Arithmetic CAGR overstates compounded return by ~`vol²/2` per year and flips decisions on volatile strategies, so Calmar must use the geometric `eq_final**(ppy/n) - 1`.

Takeaways

- **Sharpe is necessary, never sufficient.** It describes a distribution; you live a path. Report a vector (Sortino, Calmar, MaxDD, CVaR, CDaR) and require a candidate to clear all of them, because they fail independently.
- **Sortino** rewards strategies whose volatility is upside; a Sortino *below* Sharpe is a quiet left-skew warning.
- **Use geometric CAGR for Calmar.** The arithmetic mean overstates compounded return by $\sim \mathrm{vol}^2/2$ per year and flips promotion decisions on volatile strategies. Titan keeps a fast arithmetic Calmar for diagnostics and a geometric one for the gate.
- **CVaR/CDaR average the tail, they don't sample it.** VaR/MaxDD report one point; the average of the worst α-slice is the number that speaks to survival. Read CVaR (bad bars) and CDaR (deep holes) as a pair.
- **Calmar lift gates promotion; Sharpe lift is only a no-regression check.** The primary gate maps to *getting switched off mid-trough*; and even passing it is not the whole story: ruin and `P(MaxDD)` survival math run downstream.
- **Every metric obeys the same five lies.** A look-ahead Calmar is as worthless as a look-ahead Sharpe. Centralise the suite in one audited module so the unsafe version can't be written, and document, at the point of use, which question each estimator is allowed to answer.

The suite tells you how bad a strategy *could* be. The next step is turning that into a survival probability at the size you actually intend to trade: Tail risk & risk of ruin (see "Tail Risk And Ruin") converts CVaR/CDaR and the drawdown geometry into a formal risk of ruin, and The sanctuary decision matrix (see "Sanctuary Decision Matrix") shows how all of these gates combine into a single promote/reject verdict you can defend.

8. Walk-forward that's actually out-of-sample

A walk-forward optimisation (WFO) is supposed to be the experiment that earns a strategy its deployment ticket: train on the past, test on the future you couldn't see, slide the window, repeat. Done right, it's the closest a backtest comes to honesty. Done wrong (and the wrong way is *easier* and *prettier*), it's a stability scan wearing the costume of an out-of-sample test, and it will hand you a glowing curve built entirely on data the parameters already saw.

The trap is not subtle once you name it, but it is almost invisible in practice, because both procedures produce the same artefact: folds, a stitched equity curve, a confidence interval. The difference is *when* the parameters were chosen relative to the data. Get that wrong and every downstream gate (the bootstrap CI, the deflated Sharpe, the Monte Carlo) inherits a number that was never out-of-sample to begin with. This chapter is about the one question that decides whether your WFO means anything: **were the parameters chosen without looking at the data they're now being tested on?**

The principle: OOS is a property of *provenance*, not of *partitioning*

Splitting your data into in-sample (IS) and out-of-sample (OOS) windows feels like it makes the OOS window out-of-sample. It does not. A held-out window is only genuinely out-of-sample if the thing you're testing on it, the parameter set, was fixed *before* anyone looked at that window. Partitioning is necessary; it is nowhere near sufficient.

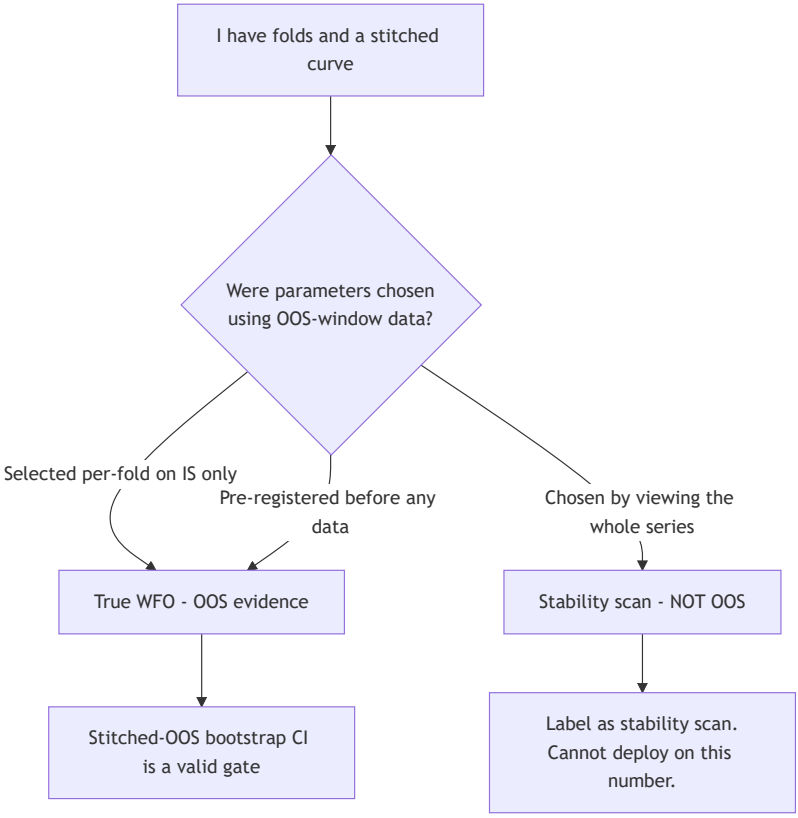
There are exactly two ways to make a walk-forward legitimately out-of-sample:

1. **Per-fold parameter selection.** Inside each fold, you fit/choose parameters using *only* that fold's IS window, then apply them, unchanged, to that fold's OOS window. The OOS bars never touch the selection. Slide the window and re-select from scratch. Each OOS segment is a genuine prediction.
2. **Pre-registration.** You write the exact parameters (or the exact grid and the exact rule for picking the winner) into a directive, commit it, and *then* run the data. The whole series can be a single OOS test because the choice was provably made blind.

If you did **neither** (if you eyeballed the full series, picked the lookback that looked best across the whole history, and *then* drew fold boundaries around

it), you have run a **stability scan**, not a walk-forward. A stability scan is a useful thing: it tells you whether one fixed configuration holds up across sub-periods. But it carries zero out-of-sample evidence, because the configuration was chosen with knowledge of every sub-period it's now being "validated" on. The cardinal sin is presenting the second as the first.

Note: Stability scan is not an insult
A stability scan answers a real question (*"is this one config robust across regimes?"*), and it's cheaper than a true WFO. The rule isn't *don't run stability scans*; it's *don't call them out-of-sample, and don't let their numbers through a deployment gate.* Label honestly and they're a legitimate part of the toolkit.



Example: How big is the lie? (illustrative)
Take a pure random walk: *no edge, by construction*. Sweep a grid of

lookbacks over the **whole** series, keep the cell with the best full-history Sharpe, then draw folds around that winner and report the stitched-OOS number: you can land a stitched-OOS Sharpe near **1.2** with a bootstrap `CI_lo` ≈ `0.4` (a confident *deploy*, on data with zero edge). Run the *identical* grid under honest per-fold selection (choose the lookback on each fold's IS window only) and the same random walk collapses to a stitched-OOS Sharpe near **0.0**, `CI_lo` straddling zero: a correct *reject*. The gap between 1.2 and 0.0 is the entire lie, and it is invisible in the artefacts: both runs produce folds, a stitched curve, and a CI. Only the *provenance* of the parameters tells them apart. (Magnitudes illustrative; the sign is not: full-series selection manufactures a positive lower bound out of pure noise.)

How Titan builds folds

Titan standardises fold construction in one module so every strategy class is tested the same way and cross-comparison is sound. A `WfoConfig` (per strategy class) declares the window geometry; `build_folds` turns it into concrete index boundaries; `iter_folds` yields `(fold, is_df, oos_df)` slices. Two modes cover the cases:

Mode	IS window	OOS placement	Used for
`expanding`	Anchored at the start, grows each fold	Non-overlapping, sequential	Slow daily strategies (trend, cross-asset momentum)
`rolling`	Fixed length, slides forward	Stride = OOS length (or half, if overlap allowed)	Faster / data-rich classes (ML, mean-reversion)

The `Fold` object carries both integer positions *and* wall-clock timestamps, so an audit log can state exactly which calendar window each fold trained and tested on:

```
@dataclass(frozen=True)
class Fold:
    fold_id: int
    is_start: int
    is_end_excl: int        # exclusive
    oos_start: int          # == is_end_excl: OOS begins where IS ends
    oos_end_excl: int
    is_start_ts: pd.Timestamp   # wall-clock, for the audit log
    oos_end_ts: pd.Timestamp
```

Two design choices in `build_folds` are worth calling out because they're easy to get wrong:

- **OOS starts exactly where IS ends** (`oos_start == is_end_excl`). There is no gap and no overlap by default. A gap throws away data; an overlap leaks the same bars into both windows of adjacent folds.
- **Fold count auto-scales to the available history.** A fixed fold count (say, five) is fine for short series but leaves most of a long history untested: your "OOS" ends up covering a small fraction of the data. The `auto_fold_count` logic derives the number of folds from the visible window so the stitched OOS spans (close to) the whole series, capped at a ceiling. The configured `fold_count` becomes a *floor*, not a fixed value. In **expanding** mode the "(close to)" is structural, not a bug: the first fold's IS window (several years, per the defaults below) can never itself become OOS, so the achievable OOS fraction is bounded below 100% by that first IS length; chasing full coverage in expanding mode is chasing a plateau, not a misconfiguration. Rolling mode has no such anchored floor.

Tip: Make 'how much was actually OOS' a reported number
Under conservative WFO defaults, the stitched OOS can cover only a few years of a decade of data. Titan's dashboard plots the *full* strategy history for visual sense-checking but draws the headline metrics **only** from the stitched OOS, with the OOS folds highlighted as bands. If a reviewer can't see what fraction of the history the gate number actually rests on, the number is doing work it hasn't earned.
Give the fraction a band to be judged against. As a rough guide (illustrative), a healthy rolling-mode WFO stitches OOS over a clear majority of the visible history (call it the high half), and once the gate number rests on much less than half the data, treat the CI as resting on too little to trust and widen the window or gather more history before deploying on it. Judge expanding mode against a lower target, since its first IS window is structurally excluded (above).

The normalisation boundary: freeze IS stats, don't peek across the split

Even a structurally correct WFO leaks if you normalise features across the IS/OOS boundary. Standardising a feature with statistics computed over the *whole* fold, IS plus OOS, quietly feeds the future into the past: every IS bar's z-score now "knows" the OOS mean and standard deviation. The fold partitioning is intact; the leak is in the feature pipeline.

The fix is to **freeze the normalisation statistics on the IS window** and apply them, unchanged, to the OOS window. Titan's metrics module exposes exactly this and deliberately offers no full-series alternative:

```
def is_frozen_zscore(x, is_end_idx: int, *, ddof: int = 1) -> pd.Series:
    """Z-score using mean/std computed only on x[:is_end_idx].
    The whole series is then z-scored with those frozen IS stats,
    so OOS bars see no OOS statistics."""
```

The guarantee is testable, and Titan tests it: append OOS bars drawn from a wildly different distribution, and the IS slice's z-scores must be *byte-identical* to what they were before the append. If appending the future changes the past, you have a leak. The same discipline applies to anything fit on a fold, whether a scaler, a regression, or an ML model: fit on `is_df`, transform `oos_df`, never the reverse, never the union.

```
for fold, is_df, oos_df in iter_folds(visible, cfg, bars_per_year=bpy):
    model = fit(is_df)                 # calibrate on IS only
    oos_returns = predict(model, oos_df)   # apply, unchanged, to OOS
    parts.append(oos_returns)
stitched_oos = pd.concat(parts)        # one OOS path across all folds
```

That `stitched_oos` series, the concatenation of every fold's OOS returns, is the only return series that carries out-of-sample evidence. Everything that decides capital is computed on it.

Warning: Per-fold *selection* is an inner grid search, not one fit
The snippet shows `model = fit(is_df)` (a single fit per fold), but the "cleaner route" the chapter recommends is per-fold **selection**: inside each fold you run the *whole grid* on the IS window, pick the winner, and score only that winner on OOS (`best = argmax_p score(fit(is_df, p), is_df)`, then `predict(fit(is_df, best), oos_df)`). Two consequences trip readers up. First, you are optimising *inside every fold*, so the trial count fed to the deflated Sharpe (see "Deflated Sharpe") is **cells** $\times$ **folds**, not cells: per-fold selection *raises* `N`, it doesn't launder it away. Second, the winner can differ per fold (lookback 5 in fold 1, 40 in fold 3), so per-fold selection produces **no single deployable config**: a stitched-OOS pass under wildly varying per-fold parameters may correspond to *no* fixed strategy you could ship. Treat large per-fold instability as a red flag (the edge is fragile to the parameter you keep re-choosing), and deploy a *fixed* config only after confirming it holds across folds; then re-validate *that* config out-of-sample, not the per-fold envelope. (Costs apply inside the OOS path, per fold, or the CI

is meaningless; see the execution layer (see "Execution And Costs").) Re-selecting per fold also multiplies compute by roughly *cells* $\times$ *folds* (and for an ML classifier, a full retrain inside every fold's IS window), which is precisely why the corner gets cut to the dishonest sweep-then-fold version this chapter warns against. The principled answers are caching the IS fits or moving to the purged/embargoed combinatorial CV below, not surrendering the provenance guarantee to save a few CPU-hours.

The gate: the stitched-OOS bootstrap CI, and nothing prettier

Once you have a legitimate stitched-OOS series, the deployment question is *not* "is the point Sharpe high?" A single number from one history is a point estimate with error bars you haven't drawn. The gate is the **95% bootstrap confidence interval on the stitched-OOS Sharpe**, and specifically its **lower bound**:

> If the 95% lower bound of the stitched-OOS Sharpe is ≤ 0, the strategy is `unconfirmed` and cannot enter the default deployment registry, regardless of the point estimate.

This is the one axis in Titan's decision matrix that can *veto* a strategy on its own: a "worst" reading on CI_lo (the interval consistent with a clearly-negative edge) caps the verdict at SUSPECT no matter how the other axes score. The lower bound is the honest answer to *"how bad could this plausibly be out-of-sample?"*, and it is computed on the stitched OOS, never the in-sample fit, never the full-series fit. This is not the only CI lower-bound gate in the book: the sanctuary chapter (see "Sanctuary Decision Matrix") adds a *second* one on a window held back from the research entirely (never touched while the parameters were being chosen), so the two are complementary, not redundant. This stitched-OOS bound asks "is this survivor's OOS edge real?"; the sanctuary bound asks "does it still hold on data the search never saw at all?"

The CI is the *veto*, not the verdict. A Sharpe whose lower bound clears zero has earned the right to be evaluated, not the right to be deployed. Everything that decides whether a surviving edge is *livable* is also computed on the stitched OOS and obeys the same provenance rule: drawdown geometry (**Calmar** and max-drawdown depth/duration), downside-only risk (**Sortino**), and the tail that ends an account (**CVaR / CDaR**, and a formal **risk of ruin** at deployed size). A walk-forward that's genuinely out-of-sample but

only ever reports one Sharpe is still under-reporting; the OOS path is the input to the whole battery in Beyond Sharpe: the metric suite (see "Metric Suite").

Two refinements matter:

- **Use a serially-aware bootstrap.** The naive IID bootstrap resamples individual bars, which destroys the autocorrelation trend and carry strategies live on, narrows the interval, and biases the lower bound *upward*: exactly the optimism the gate exists to catch. Use a stationary block bootstrap with a block length matched to the strategy's persistence. The block length is the one knob that decides whether the correction actually de-biases the lower bound, so don't leave it implicit: tie it to where the strategy's return autocorrelation decays toward zero (a slow daily trend wants a longer block than a per-trade breakout), and when you'd rather not hand-pick it, use the Politis–White (2009) automatic block-length selection as the default. (This is its own war-story in A backtest you can trust (see "Backtest You Can Trust").)
- **Deflate for the search.** The stitched-OOS CI answers "is this configuration's OOS edge real?" It does *not* correct for the fact that you tried many configurations and kept the best. That correction, the deflated Sharpe at the *registered* number of trials, is Beating your own optimizer (see "Deflated Sharpe"). The two gates are complementary: CI_lo on the survivor, DSR over the whole sweep.

Warning: War-story: the glowing WFO whose parameters had seen the whole series
A strategy arrived with a beautiful walk-forward writeup: folds drawn neatly, a stitched equity curve sloping up and to the right, a stitched-OOS Sharpe well clear of every threshold. It looked like textbook out-of-sample validation. It wasn't. The lookback and entry threshold had been chosen *first*, by sweeping the **entire** price series and picking the cell with the best full-history Sharpe, and the folds were drawn around those already-chosen parameters *afterward*. Every "OOS" window was being tested with parameters that had been optimised on data including that very window. It was a stability scan of a curve-fit, painted to look like a walk-forward. The number was real arithmetic on fake provenance: the OOS bars had no information the parameters hadn't already exploited. The lesson, call it **WFO honesty**, became a hard rule: *a walk-forward is out-of-sample only if (a) parameters are selected per-fold on IS data, or (b) they were pre-registered in a committed directive before any data was examined. Otherwise it is a stability scan and must be labelled as one.* "Looks out-of-sample" is not a category; provenance is.

Danger: Why this one is dangerous, not just wrong
A mislabelled WFO doesn't just produce an inflated number; it produces an *inflated number that passes every honest gate downstream.* The bootstrap CI on a curve-fit-then-fold series is genuinely tight and genuinely positive, because the returns it's resampling really did happen under parameters that really did fit. Sized on that confidence, real capital goes live against an edge that exists only in hindsight, and the first regime it hasn't seen takes it apart. The gate isn't "compute a CI"; it's "compute a CI **on a series whose provenance you can prove**."

Pre-registration: making "we chose blind" verifiable

Per-fold selection is the cleaner route, but some strategies have a single intended configuration you want to validate on the whole series at once. That's legitimate, *if* the choice was genuinely blind. The problem is that "we picked the parameters before we looked" is unfalsifiable after the fact. An audit of Titan's own directives found that 27 of 29 pre-registrations had been committed in the *same* commit as their result, so there was no timestamp evidence the gate-defining choices were made before the data was examined. The deflation defence was, technically, unverifiable.

Titan's fix is to make pre-registration cryptographically anchored. The gate-defining fields (strategy class, universe, the grid, the trial count `N`, the canonical cell, the decision rule) go in a machine-readable block inside the directive. A SHA-256 over the *canonical* form of just those fields is the registration hash: prose edits don't change it, but a post-hoc tweak to `N` or a threshold does. CI then verifies two things:

```
def verify_ordering(*, same_commit, prereg_committed_at,
                    result_committed_at, hash_recorded, hash_now):
    # Fails if pre-reg and result share a commit (no blind-selection
    ↪  evidence),
    # if the result doesn't strictly post-date the pre-reg,
    # or if the gate-defining block changed since registration.
```

- The result commit must **strictly post-date** the pre-registration commit (and not share it; same commit means no proof of ordering).
- The gate-defining block must be **byte-identical** between registration and result (no quiet edit to `N` or the winning cell after seeing the outcome).

This is the discipline that lets a *single* OOS pass count as out-of-sample: not because the data was partitioned, but because the choice is provably older

than the data look. Legacy prose-only directives are grandfathered out of the gate; new ones opt in by including the block.

Note: Commit-ordering raises the cost of cheating, it doesn't make it impossible
Be honest about what the hash proves. A determined author can run the sweep first and *then* write and commit the pre-registration to look blind; commit-ordering can't see the sweep that happened in a scratch notebook before the directive existed. So this is an honesty aid, not adversarial proof: it makes the dishonest path *deliberate* rather than accidental, which is most of the value for a mostly-honest solo or small-team author gating their own work. Per-fold selection is the stronger guarantee precisely because it removes the human from the loop; reach for pre-registration when the strategy genuinely has one intended config, and treat the hash as a tripwire against drift, not a proof of virtue.

Example: Two honest WFOs, one dishonest one

- **Honest A (per-fold):** each fold re-selects its lookback from a grid using only its IS window; the OOS segments are stitched; the gate runs on the stitched series. Provenance: the OOS never touched selection. ✓
- **Honest B (pre-registered):** the directive commits one parameter set and a decision rule; the data is run *after*; CI confirms the result post-dates the registration and the block is unchanged. ✓
- **Dishonest:** the whole series is swept, the best cell is kept, folds are drawn afterward, and it's reported as "WFO Sharpe X." Same folds, same arithmetic, no provenance. × This is the war-story above.

Picking honest defaults per strategy class

Fold geometry is itself a researcher degree of freedom: too many short folds and the IS windows are too small to fit anything stable; too few long folds and the OOS barely exists. Titan removes that freedom from the per-experiment author by setting it *per strategy class* and committing it as a framework default (illustrative values; the real ones live in the typology and aren't edge-bearing):

Class (illustrative)	IS min	OOS / fold	Mode
Daily trend / cross-asset	several years	~1 year	expanding
Bond-equity / mean-reversion	a couple of years	~half a year	rolling, overlap allowed
ML classifier	a couple of years	~half a year	rolling, overlap allowed

The point isn't the specific numbers; it's that the geometry is *pinned before the experiment runs*, so an author can't (consciously or not) tune the fold structure until the OOS looks good. That tuning is just curve-fitting moved up one level of abstraction, and it's exactly as invalidating. For label-overlapping problems (multi-bar ML labels), Titan also offers purged + embargoed combinatorial cross-validation, which produces a *distribution* of OOS paths rather than one, more robust, and the input to the probability-of-backtest-overfitting estimate. (Why label overlap leaks and how purge + embargo fix it is the first gate of When the strategy is a model (see "Ml Lifecycle").) But the honesty rule is identical: the OOS, however you slice it, must not have informed the parameters.

Exercises

1. **OOS is provenance, not partitioning.** Someone sweeps a grid over the *whole* history, picks the best cell, then draws walk-forward folds around it and reports the OOS Sharpe. Why is that not out-of-sample, and what is it?

   Answer: Answer
   The parameters were chosen *after* seeing the whole series, including the "OOS" windows, so the selection peeked. It is a **stability scan**, not OOS, and it cannot deploy. True OOS needs the parameters fixed *before* the window they're scored on was examined: per-fold selection on IS only, or a committed pre-registration.

2. **The two legitimate routes.** Name the only two ways to make a walk-forward genuinely out-of-sample.

   Answer: Answer

   (1) **Per-fold selection**: choose parameters using only each fold's IS window, apply unchanged to its OOS window,

re-select from scratch as the window slides. (2) **Pre-registration**: commit the exact grid and winner-rule *before* running the data, so the whole series is one blind OOS test.

Takeaways

- **OOS is provenance, not partitioning.** A held-out window is out-of-sample only if the parameters were fixed before that window was examined. Folds alone prove nothing.
- **Two legitimate routes:** per-fold selection on IS data, or pre-registration committed before the data was run. Anything else is a **stability scan**: useful, but it carries no OOS evidence and must be labelled honestly.
- **Freeze normalisation at the IS/OOS boundary.** Use IS-frozen statistics (and IS-fit models) on the OOS window; offer no full-series version that can leak the future by accident.
- **Gate on the stitched-OOS bootstrap CI lower bound**, computed with a serially-aware bootstrap. `CI_lo` $\leq$ `0` $\Rightarrow$ unconfirmed. It is the one axis that can veto on its own.
- **Make blindness verifiable.** Hash the gate-defining fields and require the result to strictly post-date the registration; "we chose before we looked" must be falsifiable, or it isn't a defence.
- A mislabelled WFO is dangerous precisely because its number passes every downstream gate: the gate must run on a series whose provenance you can *prove*, not merely on a series shaped like an OOS test.

A walk-forward tells you whether *one* configuration's OOS edge is real. It says nothing about the fact that you tried many configurations and kept the luckiest survivor, and the more you tried, the better your best result looks by pure chance. Correcting for that search is the next chapter: **Beating your own optimizer** (see "Deflated Sharpe"). From there, **Tail risk & risk of ruin** (see "Tail Risk And Ruin") turns the surviving edge into a survival probability at deployed size.

9. Beating your own optimiser

You ran a parameter sweep. A hundred-and-something combinations of lookback, threshold, and stop. One of them posted a Sharpe of 1.3, and now you want to deploy it. Here is the uncomfortable question this chapter answers: **how much of that 1.3 did you earn, and how much did you simply *find* by looking at enough cells?**

The answer is almost never "all of it." Searching a grid is not a neutral act of measurement. Every cell you try is another draw from a distribution, and the maximum of many draws drifts up even when none of the draws has any edge at all. The best cell in a sweep is *selected for being lucky*: that is the literal definition of "best." So the headline Sharpe you report is biased high by an amount that grows with the number of things you tried, and that bias has a name and a correction: the **Deflated Sharpe Ratio** (Bailey & López de Prado, 2014).

This chapter is the multiple-testing layer of Part II. A backtest you can trust (see "Backtest You Can Trust") made a *single* Sharpe honest: right units, no peeking, error bars. Walk-forward (see "Walk Forward") made sure you weren't scoring on data you fit. DSR closes the last optimism leak: the one that comes not from any single backtest being wrong, but from *picking the winner out of many right ones.*

The principle: the max of N draws is not the mean

Suppose every parameter cell in your grid is genuinely worthless: true Sharpe zero, every one. Run the backtest on each and you still get a *distribution* of estimated Sharpes, because each is computed on a finite, noisy sample. Some land at $-$0.4, some at +0.5, and, purely from sampling variance, one lands highest. Report that one and you've manufactured a positive number from nothing.

The size of the manufactured number depends on two things, and only two:

1. **How many cells you tried (`N`).** More draws, higher max. This is the part everyone underestimates.
2. **How spread-out the cells' Sharpes are (the cross-trial Sharpe standard deviation, `σ_SR`).** Wider spread, higher max.

Bailey & López de Prado give the expected maximum Sharpe *under the null of zero true edge* in closed form:

$$\mathbb{E}[\max SR_N] \approx \sigma_{SR}\left[(1-\gamma)\,\Phi^{-1}\!\left(1-\tfrac{1}{N}\right) + \gamma\,\Phi^{-1}\!\left(1-\tfrac{1}{Ne}\right)\right]$$

where γ is the Euler-Mascheroni constant and Φ^{-1} is the inverse normal CDF. You don't need to love the algebra; you need the shape of it. The bracketed term is a slowly-growing function of N, call it the **noise ceiling multiplier**. Multiply it by your sweep's actual Sharpe spread σ_SR and you get the Sharpe you'd expect to see *from the best cell of a totally dead grid*.

Here is that multiplier, computed straight from Titan's `deflated_sharpe`:

Cells tried N	Noise-ceiling multiplier	E[max SR] if σ_SR ≈ 0.30 (illustrative)
6	1.30	≈ 0.39
20	1.90	≈ 0.57
120	2.59	≈ 0.78
240	2.82	≈ 0.85
500	3.05	≈ 0.92

Read the `N = 120` row. With a perfectly ordinary cross-trial spread, the *expected* best-of-grid Sharpe is around **0.8, from variance alone, with zero real edge anywhere in the grid.** That is the whole intuition of this chapter in one number. If your 120-cell sweep produced a winner at 0.9, you have found essentially nothing; you've found the noise ceiling with a thin coat of paint. A naive bootstrap CI won't save you here either: it asks "is *this one* series significant?" and never knows it was the survivor of a beauty contest.

Note: The fifth lie's bigger sibling
The backtest chapter (see "Backtest You Can Trust") ended on *"a Sharpe is an estimate, not a fact"* and prescribed a bootstrap confidence interval. DSR is the version of that lesson for a *family* of estimates. The bootstrap CI controls the error on one number; DSR controls the error on the *selection* of that number from many. You need **both**, and that ordering matters: DSR is applied *in addition to*, never *instead of*, the CI gate.
DSR is also one of several overfitting controls in the same López de Prado toolkit, and they answer different questions. DSR haircuts a *single declared winner* into `P(true SR > 0)` given N trials: the form that maps cleanly onto a per-strategy deploy/no-deploy threshold, which is why Titan gates on it. PBO/CSCV (probability of backtest overfitting, via combinatorially-symmetric cross-validation) instead estimates the probability that your *selection procedure* overfits, by reshuffling which folds are in- and out-of-sample. The Sharpe *haircut* reports the same deflation as a shrunk Sharpe rather than a probability. The latter two are complementary diagnostics, not substitutes for the gate.

How DSR deflates

The Deflated Sharpe Ratio converts the gap between your observed Sharpe and the noise ceiling into a probability:

> **`DSR = P(true Sharpe > 0)`**, after accounting for `N` trials and the return distribution's shape.

Three inputs go in, and getting any of them wrong corrupts the answer:

- **The gap.** `SR_hat − E[max SR_N]`. If your winner doesn't clear the noise ceiling, the gap is negative and the deflated probability collapses toward zero. This is the multiple-testing penalty made concrete.
- **The trial count `N`.** Drives the ceiling. Honest counting is the entire game (next section).
- **The return distribution's skew and kurtosis.** The Sharpe estimator's *own* sampling error is wider for skewed, fat-tailed returns. A strategy whose edge is "win small often, lose big rarely" (negative skew, high kurtosis) has a noisier Sharpe than its point estimate suggests, so the same observed Sharpe earns a *lower* deflated probability. Titan's implementation reads these moments from the actual return series rather than assuming normality:

```
from titan.research.framework.dsr import deflated_sharpe,
 ↪  sr_var_from_sweep

# σ_SR² estimated across the FULL sweep, not the survivors (see below).
# all_cell_sharpes is the OOS-Sharpe column the audit harness persists
 ↪  for
# EVERY cell it scored -- survivors AND failures -- precisely so this
 ↪  variance
# is computed over the whole grid. Reconstructing it from the deployed
# survivors alone is the survivors_only trap flagged below.
all_cell_sharpes = sweep_results["oos_sharpe"].to_numpy()
sr_var = sr_var_from_sweep(all_cell_sharpes)

res = deflated_sharpe(
    sr_hat=canonical_sharpe,          # the cell you actually want to
 ↪  deploy
    sr_var_across_trials=sr_var,      # spread of Sharpes across all N
 ↪  cells
    returns=canonical_oos_returns,    # this cell's OOS series -> skew +
 ↪  kurt
    n_trials=N,                       # the HONEST trial count
)
print(res.dsr_prob, res.e_max_sr)     # gate: dsr_prob >= 0.95
```

The result object carries its own diagnostics, `e_max_sr`, the skew/kurt it used, the variance-stabilised gap `z`, and a `survivors_only` flag, so a reviewer can see *why* a cell passed or failed, not just the verdict. The deployment gate is `dsr_prob >= 0.95`.

Note: Degenerate inputs fail safe
The implementation refuses to manufacture a pass out of a missing input. With fewer than 2 trials, or a non-positive cross-trial variance, there is nothing to deflate against, so `dsr_prob` is forced to `0.0`; you cannot clear a DSR gate you never actually ran a sweep for. And below ~30 return observations the skew and kurtosis estimates are too noisy to trust, so the moment-reader falls back to the normal assumption (`skew = 0`, `kurt = 3`): a very short sample *silently loses the fat-tail penalty*. That is one more reason a sparse strategy needs per-trade `T` (next section) and a hard look before deployment, not a per-bar series long enough to dodge the guard but lying about its cadence.

Tip: Report `e_max_SR` next to every sweep winner
The single most clarifying habit: print the noise ceiling beside the headline number. "Best cell SR = 0.9, `e_max_SR` = 0.85" tells the whole story at a glance: the winner is *barely* above what an empty grid would have produced. A reader who only sees "0.9" thinks you have a strategy. A reader who sees both knows you have a coin flip.

Example: One DSR, end to end (illustrative)
The `z` the result object reports is the gap divided by the estimator's *own* noise (Bailey–López de Prado's variance-stabilised statistic):

$$z = \frac{\hat{SR} - \mathbb{E}[\max SR_N]}{\sqrt{\left(1 - \hat{\gamma}_3\,\hat{SR} + \frac{\hat{\gamma}_4 - 1}{4}\,\hat{SR}^2\right) / (T-1)}}, \qquad \texttt{dsr\_prob} = \Phi(z)$$

where γ_3 is skew, γ_4 kurtosis, and `T` the number of return observations. Negative skew and high kurtosis *inflate the denominator*; that is mechanically how "fat tails earn a lower `dsr_prob`," and it's the one piece of the formula the chapter otherwise leaves as a black box. Walk one to a verdict, and watch the **frequency**, because every quantity in the `z` must live at *one* cadence: `SR_hat`, `E[max SR_N]`, and the `T` in the denominator. Our sweep reports **annualised** Sharpes, so the winner is `SR_hat = 0.85`, drawn from `N = 120` cells whose annualised Sharpes spread at `σ_SR = 0.30`, giving an annualised ceiling `E[max SR_N] ≈ 0.78`: an annualised **gap of only 0.07**, the best of 120 barely clearing

an empty grid. But the SE's `/(T-1)` counts the **daily** return series (`T = 750` bars), so the Sharpe in the numerator has to be daily too. De-annualise both by `√252`: `SR_hat ≈ 0.054`, `E[max] ≈ 0.049`, gap `≈ 0.0044`. With skew `−0.4`, kurtosis `6`, the denominator is `sqrt((1 + 0.021 + 0.004)/749) ≈ 0.037`, so `z ≈ 0.0044 / 0.037 ≈ 0.12` and `dsr_prob = Φ(0.12) ≈ 0.55`: **a decisive fail of the 0.95 gate.** Feed the *annualised* 0.85 straight into that daily `/(749)` instead (mixing frequencies) and you get `z ≈ 1.28`, `dsr_prob ≈ 0.90`, a `z` inflated ~10× that flatters a coin flip almost to the gate. That is not a hypothetical slip; it is the war-story below. And note the fat tails barely moved this answer (Gaussian returns give `≈ 0.55` too): at daily cadence the per-bar Sharpe is ~0.05, and the skew/kurtosis terms scale with it, so the haircut here is tiny; the fat-tail penalty bites hardest on a **sparse, per-trade** series where the per-period Sharpe is large (score 20 round-trips on their 20 trade returns, never on 750 daily bars).

The most subjective input is `σ_SR`, and it matters most: `E[max SR_N]` scales roughly *linearly* in it, so a 2× mis-estimate nearly doubles the ceiling and can flip the verdict on its own. A *measured* cross-trial spread also conflates genuine dispersion of edges with pure estimation noise, so anchor on a small, stable value ($≈$0.1–0.4 for nearby grid cells) and don't let a noisy estimate quietly inflate your ceiling.

Danger: War-story: the deflation gate that rubber-stamped every winner

For a long stretch, Titan's `deflated_sharpe` made exactly the units slip in the example above, in production, on the real gate. It fed the **annualised** Sharpe into the variance-stabilised `z` whose `/(T-1)` counts **daily** bars. Numerator and denominator lived at different frequencies, so `z` came out inflated by roughly ten-fold: any winner that cleared its noise ceiling by a real margin was pushed to a `dsr_prob` of essentially `1.000`, and only a candidate as marginal as the example above (a 0.07 gap, scoring `0.90`) still tripped the gate. The `dsr_prob >= 0.95` deployment gate was therefore **near-unfailable in practice**: "passed DSR" certified almost nothing. When the units were fixed (de-annualise the Sharpe to per-observation, `÷ √periods_per_year`, before forming `z`), a representative overfit winner fell from `0.98` (a clean pass) to `0.56` (a clean fail), and **every strategy that had ever cleared the gate failed on recompute.** The meta-lesson is the sharp one: a bug in the *deflation gate itself* is worse than having no gate at all. A missing gate lets a fake winner through and at least you *know* you're unprotected; a broken gate **launders** every overfit winner with an official stamp, so the worse your discipline the more it reassures you. This is the verify-your-verifier

(see "Backtest You Can Trust") discipline turned on the very tool this chapter sells: the thing that grades your Sharpes needs its own independent check, to the basis point, before you trust a single verdict it prints.

Counting `N` honestly is the hard part

The formula is mechanical. The judgement is in `N`, and it is almost always *larger* than the number you instinctively report.

What you tried	Naive `N`	Honest `N`
A 5×4×6 grid, ran once	120	120
"I only swept 6 cells", after 4 earlier exploratory grids you abandoned	6	the cumulative total across all the grids you looked at
Screened a universe of ~500 instruments, kept the handful that looked good	a handful	≈ 500
Tweaked the stop "by hand a few times" until it looked right	1	every variant your eyes evaluated
Searched 30 candidate features, kept the 5 that predicted	5	every feature tried, times the cells per feature

Feature search is the dominant, most-often-omitted N-inflator. A machine-learning or multi-factor workflow that screens dozens of candidate predictors and keeps the few that "worked" has run a selection at least as wide as any parameter grid (usually wider, because each feature is itself swept over lookbacks and transforms), yet feature selection is almost never counted in `N`. It is the same sin as keeping the screener's survivors: the selection pressure was applied across *every feature you evaluated*, so they all belong in the pool. A forecast built from selected features (combining forecasts (see "Combining Forecasts")) inherits this inflated `N`, and deflating against the survivors alone is how an overfit feature set passes a significance test it should fail.

The rule Titan enforces: **`N` is the size of the candidate pool you selected from, not the number of survivors.** If a strategy passed through a several-hundred-name screener, `N` is the whole pool (even if you only carried a handful of names forward) because the selection pressure was applied across every name. Using the survivor count understates the ceiling and inflates `dsr_prob`. Worse, the *variance* term has the same trap: estimate `σ_SR` from the survivors only and you get a too-small spread (survivors are, by construction, the cells that clustered high), which *also* biases the probability optimistic. The framework lets you do it when full-pool data is unavailable, but it forces you to admit it:

```
# Only have the survivors' Sharpes? You may pass them, but the result
# is FLAGGED optimistic -- the true ceiling is higher than this.
res = deflated_sharpe(..., survivors_only=True)
assert res.survivors_only  # documented as a LOWER BOUND on the penalty
```

There is one force pushing the other way, and honesty means naming it. The closed form assumes `N` *independent* draws, but adjacent grid cells are anything but: a lookback of 19 and 20 trade almost the same bars, so the *effective* number of independent trials in a smooth sweep is well below the nominal cell count. Feed the nominal `N` and you therefore **over**-deflate: the ceiling is too high, the penalty too harsh. We keep nominal `N` anyway, because a conservative ceiling is the *safe* error: it can only reject a real edge, never wave through a fake one. But two cases break the symmetry and are worth holding in mind. A feature or instrument *screen* is far closer to independent than a parameter grid (distinct instruments are not near-duplicates of each other), so count those at full nominal `N` without apology. And a winner sitting on a broad *plateau* of correlated neighbours is precisely the regime where nominal `N` most overstates the true penalty; that is no coincidence, it is why the plateau check and DSR are complementary rather than redundant: one rewards exactly the correlation the other conservatively double-counts.

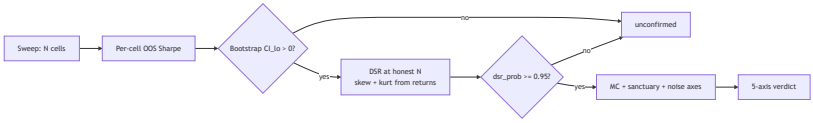
The deepest version of honest counting is **pre-registration**: write down the grid and the trial count *before* you run it, commit it to git, and let the audit read `N` from that committed manifest. Titan's audit wrapper refuses to run against an uncommitted pre-reg precisely so that `N` cannot be quietly shrunk after the winner is known. That machinery lives in the sanctuary & decision matrix chapter (see "Sanctuary Decision Matrix"); here the point is narrower: **the trial count is an input to a significance test, so inventing it after the fact is the same sin as p-hacking.**

Danger: War-story: the screener winner that was indistinguishable from noise

A breakout strategy was run across a large universe (a few hundred instruments) and the handful that survived posted eye-catching annualised Sharpes, some in the high single digits on short, sparse samples. The instinct was to deploy the survivors. Then we ran DSR at the *true* trial count (the full screener pool, not the survivors) and the picture inverted. Because the per-instrument samples were short and the cross-trial Sharpe spread was large, the noise ceiling `e_max_SR` came out *enormous*, well above every survivor's observed Sharpe. Even using the deliberately *optimistic* survivors-only variance, **every surviving cell failed the DSR gate.** The "winners" were exactly what a dead universe of that size and sample length produces by chance. Two compounding mistakes had hidden it: scoring sparse-trade strategies on a per-bar Sharpe annualised by a huge factor (which inflated the raw numbers), and never deflating by the real `N`. The rule it bought: **a screener's `N` is the pool, not the podium; and a sweep with more than a handful of cells is unconfirmed until DSR is run at that `N`, on top of the bootstrap CI.**

Where DSR sits in the gate

DSR is not a standalone verdict. In Titan it is one of five axes in the decision matrix, alongside the bootstrap CI lower bound, a Monte-Carlo drawdown test, a held-out sanctuary year, and a noise-robustness check. A cell must clear `dsr_prob >= 0.95` to score "best" on its axis; below ~0.5 it scores "worst." No single axis deploys a strategy, but a failing DSR caps the verdict hard.



The ordering is deliberate and cheap. CI and DSR are seconds of compute; the Monte-Carlo and sanctuary axes are minutes-to-hours. So the sweep gate runs **CI_lo, then DSR, first**, and a strategy that can't clear the noise ceiling never reaches the expensive machinery. Many decayed published edges die right here, in under a minute, because their best-of-grid cell sits below `e_max_SR`.

Warning: War-story: the plateau that was a single lucky spike
A defensive overlay was being tuned by sweeping its kill-and-re-entry thresholds. One specific cell posted a beautiful drawdown profile:

`P(MaxDD > 50%)` of half a percent in Monte Carlo. The team nearly committed it. But the neighbouring cells told a different story: nudge the kill threshold one step and the same metric jumped to 25 to 30%. The "safe" cell wasn't a robust region of the parameter space; it was a **single spike surrounded by cliffs**, the textbook signature of fitting noise, not signal. We added a *plateau* pre-flight that runs before the audit: take the winner and its grid neighbours, and if their relative Sharpe spread is too wide, abort; there is no robust region to deploy. DSR is the quantitative form of this concern (the max of `N` draws); the plateau check is its structural form (a real edge survives small parameter perturbations, a fitted one doesn't). The rule: **a sweep winner must be a plateau, not a peak; and it must clear DSR at the full `N` before any compute is spent past the gate.** A lone lucky spike fails both tests, and it should.

DSR is a *selection* control, not a quality stamp

One caveat, because it is the way DSR gets over-trusted. A high `dsr_prob` says "this Sharpe is unlikely to be a multiple-testing artifact." It does **not** say the strategy is deployable, or even good. A signal can clear DSR and still die at the next gate, most commonly when realistic costs eat the edge, or when the held-out sanctuary year diverges from the WFO. DSR also can't see selection it wasn't told about: the abandoned grids, the eyeballed stop tweaks, the four earlier experiments you don't count. **It deflates the trials you declare. Declaring them honestly is on you.**

And like every number in Part II, the inputs to DSR obey the same measurement rules. A `dsr_prob` computed on a look-ahead equity curve, or on a per-bar Sharpe that should have been per-trade, is exactly as worthless as the Sharpe that fed it. The affirmative rule at the API: **feed `deflated_sharpe(returns=...)` the series at the cadence the strategy actually trades.** For a sparse strategy (a handful of round-trips a year), pass the *per-trade* return series and set `T` to the trade count, never a per-bar series annualised by a huge factor, which both inflates `sr_hat` and lies about `T`. Use a daily or per-bar series only when the strategy is genuinely active every bar. DSR is a correction *on top of* a trustworthy Sharpe, never a substitute for one. The full battery of livability metrics (Sortino, Calmar, CVaR/CDaR, and a formal risk of ruin at deployed size) is the subject of the metric suite (see "Metric Suite") and position sizing (see "Position Sizing Kelly"); DSR just makes sure the Sharpe you carry into them was *earned*, not *mined*.

Exercises

1. **Count `N` honestly.** You screened 500 instruments, swept a 5$×$4 grid on the survivors, and hand-tweaked a stop "a few times." A colleague reports `N = 20`. What should `N` be, and why does under-counting flatter the result?

   > **Answer: Answer**
   > `N` is the whole **candidate pool** the selection pressure acted across (the ~500-name screen, *times* the grid cells, *plus* every hand-tweak your eyes evaluated), easily in the thousands, not 20. A too-small `N` understates the expected null maximum Sharpe, so the deflated probability comes out optimistically high.

2. **Plateau vs spike.** Two winning cells both post Sharpe 1.5. One's grid neighbours post ~1.4; the other's post ~0.2. Which do you trust, and what does the other one indicate?

   > **Answer: Answer**
   > Trust the **plateau** (neighbours ~1.4): a real edge is robust to small parameter changes. The lone **spike** (neighbours ~0.2) is a knife-edge in the noise, a fitted artefact, and should be rejected even at the same point estimate.

Takeaways

- **Searching a grid inflates your best result.** The maximum of `N` noisy estimates drifts up even when every cell has zero true edge; the more cells, the higher the drift.
- **Deflate it.** The Deflated Sharpe Ratio subtracts the expected null max (driven by `N` and the cross-trial Sharpe spread) and returns `P(true Sharpe > 0)`. Gate at `dsr_prob >= 0.95`. Any sweep beyond a handful of cells needs it.
- **Count `N` honestly.** It's the candidate *pool*, not the survivors; it includes abandoned grids and hand-tweaks; pre-register it so it can't shrink after the winner is known. Survivors-only variance and `N` both bias the probability optimistic; flag them when you must use them.
- **Use the actual distribution's skew and kurtosis.** Fat-tailed, skewed returns make the Sharpe noisier than a normal assumption implies; DSR should penalise them, and Titan's does.
- **DSR sits on top of the bootstrap CI, not instead of it**: CI controls the error on one estimate, DSR controls the error from selecting it. Run both, cheaply, before any expensive gate.

- **A high `dsr_prob` is a selection clearance, not a deployment stamp.** Costs, drawdown paths, and the sanctuary year still get a vote.

DSR closes the multiple-testing leak in a *single* candidate's evaluation. Two chapters take the rigour further: Tail risk & risk of ruin (see "Tail Risk And Ruin") turns drawdown geometry into a survival probability at deployed size, and the Sanctuary decision matrix (see "Sanctuary Decision Matrix") shows how DSR becomes one binding axis among five, including the pre-registration discipline that keeps `N` honest in the first place.

10. Tail risk & risk of ruin

A Sharpe ratio tells you whether a strategy is good on an average day. It is silent on the only question that ends a trading business: *can a plausible bad year kill the account?* Survival is not the average outcome. It lives in the left tail, the worst path the world could have dealt you, not the median one, and the tail is exactly where most backtest machinery is weakest, because there are so few extreme observations to learn from.

This chapter is about turning vague tail fear into a number you can gate on. We do it in two moves. First, simulate worst-case paths *honestly*, which, as we'll see, almost nobody does, because the obvious way to bootstrap a strategy quietly deletes most of its tail risk. Second, translate the resulting drawdown distribution into a **risk of ruin**: the probability that a strategy, deployed at a specific weight, trips a portfolio kill switch over a real horizon. And because you never run one strategy alone, we extend it to **joint ruin** across everything deployed at once, including the case where your diversification evaporates precisely when you need it.

The principle: simulate the cause, not the effect

You want the distribution of a strategy's max drawdown across the many histories the world *could* have produced, not just the one it did. The standard tool is a block bootstrap: resample the time series in chunks (to preserve serial correlation), recompute the metric on each resample, and read the percentiles. The question is *what series you resample.* There are three choices, and they are not close to equivalent.

Approach	What you resample	What it preserves	Tail honesty
Strategy-return bootstrap	the strategy's realised P&L series	the historical strategy distribution	**Badly understated**
Price bootstrap	the underlying price levels directly	almost nothing useful	Noise
Underlying-return bootstrap	the underlying's *returns*, then re-run the strategy	the data-generating process	**Honest**

The trap is the first one, and it is seductive because it's cheap: you already have the strategy's return series from the backtest, so why not just resample *that?* Because the strategy's realised returns are an *effect*. They are

what happened after the strategy's logic (entries, exits, stops, position sizing, regime gates) interacted with one particular price path. Resample the effect and you are sampling from a distribution your strategy already survived. The blocks you draw are blocks the strategy already navigated; its stops already fired, its filters already sidestepped the worst entries. You learn how variable the *outcome* was, not how variable the *world* is.

The honest approach resamples the **cause**: bootstrap the underlying instrument's returns, `cumprod` them back into a synthetic price path, and **re-run the full strategy on each synthetic path.** Now the strategy meets price sequences it never saw, including bad ones where a stop gets gapped through, a filter lets a loser in, or a trend reverses one bar after entry. The drawdown distribution you get out is the distribution of what the strategy *would do under stress*, not a reshuffling of how it already performed.

> **Warning: War-story: the tail that was many times too thin**
> An external audit caught Titan's first Monte Carlo doing the cheap thing: it bootstrapped the **strategy's** return series. Across the strategies on the bench, the resulting `P(MaxDD > threshold)` was understating true tail risk by a large multiple, comfortably an order of magnitude on some strategies (the exact factors are strategy-specific and not published here). The mechanism is exactly the "effect, not cause" error above: resampling realised P&L cannot manufacture a drawdown the strategy never produced, so the worst synthetic path is bounded by the worst *historical* path. The rebuild-and-re-run version can hand the strategy a never-before-seen sequence that walks straight through its stops. The fix changed one thing and everything downstream: **resample the underlying returns at shared block indices, `cumprod` to a synthetic price, and re-run `strategy_fn` on it.** Every drawdown gate that had been "passing" was re-derived; some strategies that looked tail-safe were not. (This is the canonical telling of catalogue entry A6 in the failure-mode catalogue (see "Failure Mode Catalogue").)

The Titan example: rebuild the path, then re-run the strategy

Titan's Monte Carlo (`titan/research/framework/mc.py`) is built around that one discipline. The core of a path is three lines: resample block indices, rebuild a price, run the strategy.

```
def _rebuild_path(log_returns, indices, initial_price):
    resampled = log_returns[indices]                        # (1)!
    return initial_price * np.exp(np.cumsum(resampled))     # (2)!
```

```
# ... per path:
indices       = _resample_indices(n_bars, block_size, rng)  # (3)!
synth_price   = _rebuild_path(primary_log_returns, indices, p0)
df            = pd.DataFrame({"close": synth_price}, index=...)
strat_returns = strategy_fn(df)                  # (4)!
mdd           = max_drawdown(strat_returns)      # measured on the
 ↪  strategy, on a new world
```

1. We resample the **underlying's** log returns, the cause, not the strategy's P&L.
2. `cumprod` (here as `exp(cumsum)` of log returns) rebuilds a synthetic *price path*, which is what a strategy actually consumes.
3. Block resampling preserves serial structure; an IID shuffle would destroy the autocorrelation that trend and carry strategies trade on (see the bootstrap war-story in A backtest you can trust (see "Backtest You Can Trust")).
4. The strategy runs on the synthetic price exactly as it would in the backtest (same entries, stops, filters), so its drawdown reflects how it behaves in a world it has not seen.

One thing the single-`close` sketch hides: `strategy_fn` on a synthetic path must apply the **same cost and financing model as the backtest** (per-trade commission and slippage, plus carry/borrow for anything held overnight). On a synthetic price those costs are *modelled, not observed*, so they inherit the backtest's cost assumptions (see The execution layer (see "Execution And Costs")). Skip them and you feed a *gross*-of-cost drawdown distribution into a ruin gate whose input is net of cost; the two sit on different footings and the run understates ruin.

Danger: Units contract: log returns in, simple returns out
Two adjacent modules use *opposite* return conventions, and mixing them is silently mis-scaled, not an error. `_rebuild_path` above expects **log** returns: it does `exp(cumsum)` to rebuild the price. The ruin module (`assess_strategy_ruin`, next section) expects **simple** returns: it does `cumprod(1 + r)` to build equity. Convert at the boundary (`simple = exp(log) - 1`) and never hand one module the other's convention; a reimplementer who pipes log returns straight into the ruin call gets a number that looks plausible and is wrong.

Warning: Re-running the strategy 1,000× is the practical wall
The honest bootstrap's whole prescription (rebuild a synthetic price and **re-run `strategy_fn` on it**) quietly assumes `strategy_fn(df)` is

cheap and consumes only a price path. Neither is free. Re-running a real strategy (indicators, regime gates, sizing) across `n_paths=1000` multi-year synthetic histories can take minutes to hours, and a strategy that depends on volume, spread, fills, borrow, or *multiple* instruments isn't a function of one `close` column at all. Two consequences: structure `strategy_fn` so it *can* be re-run headless on a synthetic frame (no live-only dependencies, cache what's reusable), and for a cross-asset (see "Factor Beta Exposure") sleeve, `shared_block`-resample *all* legs at the same indices and re-run a strategy that consumes the multi-series synthetic. The single-`close` sketch above is the easy case, not the common one. And be honest about the assumption the technique buys in exchange for fixing "effect, not cause": that the *price path determines the P&L*. For a strategy where path-dependent execution dominates, that is its own false precision.

Three bootstrap flavours are offered, and the choice matters for multi-asset strategies:

- **`block`**: fixed-length blocks from one series; extras resampled independently.
- **`shared_block`**: two or more series drawn at the *same* block indices, so a bond/equity pair keeps its cross-correlation through the resample. Essential for cross-asset strategies; resample the legs independently and you'd invent diversification that isn't there.
- **`stationary`**: Politis & Romano (1994) with geometric (random) block lengths and circular wrap. Fixed blocks have hard boundaries that under-represent *clustered* drawdowns, the multi-week grind that actually hurts, so the stationary scheme is the conservative default when the deploy decision hinges on the tail.

Reading `P(MaxDD > threshold)`

The headline output is `p_maxdd_gt_threshold`: the fraction of synthetic paths whose max drawdown exceeded a stated level. You gate on it directly: *the probability of a worse-than-X drawdown must stay under some small p.* It is a far more honest number than a single backtest MaxDD, because the backtest gives you one draw and this gives you the distribution.

But one subtlety, learned the hard way, deserves its own gate:

Note: Absolute MaxDD gates are wrong for long-only sleeves
For a long-only equity or "ballast" strategy, the block bootstrap shuffles real crisis bars into most synthetic paths, so the **underlying itself**

fails an absolute `P(MaxDD > X%)` gate, which means the gate is testing the market, not the strategy. The economically correct question is *relative*: "does the strategy draw down **less than buy-and-hold on the same synthetic path?**" Titan's `run_relative_block_mc` runs both the strategy and a benchmark on each path and gates on the *ratio* of their drawdowns (median ratio below 1, and the strategy no-worse on a majority of paths). Use the absolute gate for market-neutral and tactical strategies; use the relative gate for anything whose thesis is "I add defensive value over the underlying."

From drawdown to ruin: a survival probability at deployed size

`P(MaxDD > X)` is a tail-risk *proxy*. It is computed at full strategy size and says nothing about the two facts that actually decide survival: **how much capital you deploy into the strategy**, and **the level at which the portfolio's kill switch fires.** A strategy with a scary standalone drawdown can be perfectly safe at a small weight; a mild one can be lethal if it's the whole book and the kill switch sits just below its typical trough. Risk of ruin closes that gap.

A word on the term, because it carries two meanings. The classical "risk of ruin" (the gambler's-ruin and Kelly sense) is the *asymptotic* probability of ever hitting an absorbing barrier as time goes to infinity, a property of the edge and the bet size alone. That is not the quantity here. We mean the narrower, operationally useful event: that a portfolio kill switch trips within a *specific deployment horizon* `H`, a number that depends on both `H` and the weight rather than an as-time-goes-to-infinity limit. The Kelly fraction sets the weight (Position sizing (see "Position Sizing Kelly")); this chapter prices what that weight does to survival over the horizon you actually deploy across.

Example: Worked example: the broker never ends you (a public-index confirmation)
A rare case where the numbers are *public*, so they can be shown in full. Take the boring workhorse: a static 60/40 (SPY / IEF) with the equity leg levered to 2×, run as a full path-dependent **margin account** across 2002–2026 (GFC-inclusive) plus 1,000 block-bootstrap re-simulations of the underlying. Ask the *classical* ruin question first: does the broker ever force-liquidate? **It never does.** `P(margin call) = P(ruin) = 0.0%` in every cell, every window, and every bootstrap path. Minimum equity-to-LMV on the whole 24-year path is **0.575** against a 0.25 maintenance floor; gross leverage peaks at **1.74**× against a 1.60× target. A

call would need a **> 57.5%** maintenance requirement, one no broker imposes. A prior fear ("it gets margin-called in 2008 and never recovers") was simply **falsified.**
Now ask the *operational* question. At 2×, a bootstrap on the common 2008-onward window puts `P(MaxDD < −35%)` at **89.6%** (a worse-than-35% drawdown is **near-certain**), and on the full 2002-onward path the realised draw reaches **$-$60.5%.** (Two windows, named as two, because the probability and the realised depth are not measured on the same sample, and this chapter is about to make window choice the decisive point.) That sits far below any kill line a real book runs, well past the illustrative $-15-$20% NAV levels this chapter simulates against. So the leverage produces not *margin* ruin but a *near-certain* drawdown that the portfolio's own kill switch force-liquidates long before the backtested CAGR is ever earned. **The broker never ends you; your own kill switch does**: exactly the classical-vs-operational distinction just drawn, confirmed on public numbers. And note that *falsifying* the margin-call fear **strengthens** the point rather than softening it: the absorbing barrier that ends the account is not the broker's, it is the one *you* set.

Titan's `assess_strategy_ruin` (`titan/research/framework/ruin.py`) takes the strategy's **net OOS returns**, a **deployment weight**, and a **portfolio kill threshold**, then forward-simulates over a real horizon. The values below are *illustrative*: the live horizon, kill level, and deploy weights are doctrine settings and are not published here:

```
res = assess_strategy_ruin(
    strategy_returns=stitched_oos_returns,   # net of cost, OOS
    deployment_weight=W,                      # fraction of full size
↪   (illustrative)
    portfolio_kill_threshold=K,               # kill switch at K% NAV DD
↪   (illustrative)
    horizon_bars=H,                           # the horizon you actually
↪   deploy over
    block_size=B,                             # ~1 month, preserves
↪   clustering
    n_paths=1000,
    seed=42,
)
```

Each path is a block bootstrap of the strategy's returns out to the horizon; the returns are scaled by `deployment_weight`; the path's max drawdown is compared to the kill threshold. The probability that the scaled drawdown crosses the kill line is `p_kill_trip`: your **risk of ruin over that horizon at that weight.** The assessment also reports `median_recovery_bars` (a deep

drawdown you recover from in a month is not the same animal as one that takes two years) and a separate `p_dd_50pct_strategy`, a catastrophic-strategy guard measured at *full* size, independent of how cautiously you deploy.

By now four distinct drawdown gates have been introduced across two sections; it helps to see them in one place, because each measures size differently and protects against a different thing:

Gate	Size measured at	Applies to	What it protects
Absolute `P(MaxDD > X)`	full strategy size	market-neutral / tactical sleeves	standalone tail against an absolute ceiling
Relative DD ratio (`run_relative_block_mc`)	full size, vs B&H on the *same* path	long-only / ballast sleeves	that the strategy beats its own underlying's drawdown
`p_kill_trip`	scaled to deployment weight	every deployed sleeve	survival of the portfolio kill switch over the labelled horizon
`p_dd_50pct_strategy`	full size, weight-independent	every sleeve	a catastrophic-strategy guard no cautious weight can mask

The first two are a **choose-one** pair: the absolute gate for tactical/market-neutral sleeves, the relative gate for anything whose thesis is "I add defensive value over the underlying" (above). The last two are not a choice: `p_kill_trip` and `p_dd_50pct_strategy` **both** must pass, because a weight gentle enough to keep `p_kill` tiny still does nothing about a strategy that can halve at full size.

One deliberate divergence from the earlier advice: the ruin module bootstraps with **fixed**-length blocks (`block_size=B`), not the stationary scheme named above as the conservative tail default. The trade is reproducibility and a directly interpretable ~one-month clustering window, at the cost of fixed boundaries that slightly *under*-represent clustered drawdowns. So if anything a near-threshold ruin number from this module should be read as mildly optimistic, not conservative: one more reason not to deploy on a `p_kill` decided by a single block boundary.

Example: A risk-of-ruin read, end to end (illustrative)
Numbers to anchor the mechanism, all illustrative; the real horizon, kill level, and weights are doctrine. A strategy's net OOS returns are

block-bootstrapped out to a **252-bar (one-year) horizon**, scaled to a **deployment weight of 0.15** (15% of full size), and each path's max drawdown is compared to a **portfolio kill threshold of $-$20% NAV**. Say 1,000 paths produce 4 that breach the kill line: `p_kill_trip = 4/1000 = 0.4%`, your one-year risk of ruin *at that weight*. The same strategy at a **0.40** weight might trip ~60 paths → 6%: an order of magnitude worse from the *same edge*, because ruin scales with deployed size, not just the strategy. And mind the Monte-Carlo error on the number itself: 4 events in 1,000 paths is a wide binomial interval (roughly 0.1%–1%), so a gate at `1e-3` decided by one or two paths is a coin flip. Re-run across seeds and treat a near-threshold `p_kill` as *un-measured*, not passed.

Two design decisions in that module are worth lifting out, because they are the kind of thing that silently corrupts a risk number.

Warning: War-story: additive drawdown on a fractional threshold
An audit found the ruin module computing drawdown with `np.cumsum` (*additive*) and then comparing that cumulative-return delta against fractional thresholds like `-0.15`. Drawdown is a fraction of *equity*, which compounds geometrically; an additive proxy is not a fraction of anything and is internally inconsistent with the rest of the framework's `max_drawdown`. The fix prepends a starting equity of `1.0` and uses `np.cumprod(1 + r)`, so a first-bar loss is a real drawdown from par. The deeper lesson: a risk number computed with the wrong arithmetic is *worse* than no number, because it carries a false precision into a deploy decision. And there's a sharp edge for the caller: the function expects **simple** returns; hand it log returns and it is silently mis-scaled.

The second decision is about pairing parameters with thresholds so doctrine can't drift.

Danger: War-story: the gate that drifted out from under the threshold
The first ruin gate hard-coded its pass thresholds (a short horizon, a kill level, a `P(kill)` tolerance). When doctrine moved to a tougher standard, a much longer horizon with stricter `P(DD)` and `P(MaxDD)` ceilings, every *caller* still simulated under the old, short horizon while *claiming* to apply the new gate. The numbers looked compliant; they were answering a short-horizon question with a long-horizon label. The fix bundles the simulation parameters and the pass tolerances into one `RuinGate` object, and `passes_gate()` **raises** if the assessment's

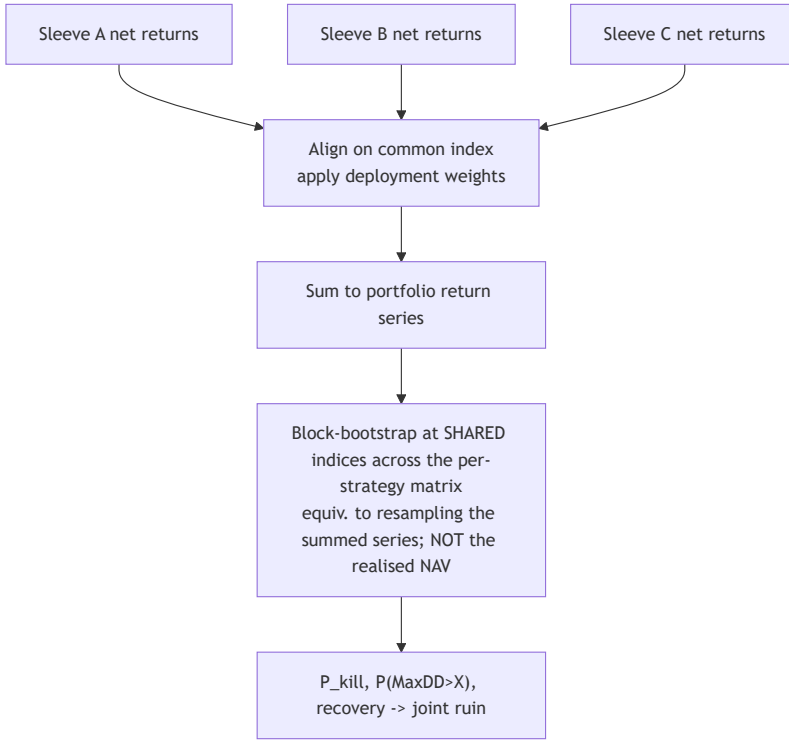
horizon or kill level doesn't match the gate's. You cannot evaluate a long-horizon gate against a short-horizon simulation anymore; the API refuses. (The specific horizons and ceilings are illustrative; the lesson is the binding.) When a risk threshold is a magic number floating free of the simulation that produced it, it *will* eventually describe a different experiment than the one you ran.

The pattern is worth stating as a rule, because it generalises past risk of ruin: **bind every threshold to the experiment that produced it.** A pass/fail tolerance that lives in a different object from the simulation parameters is one refactor away from grading the wrong exam.

Joint ruin: you never deploy one strategy

A portfolio's risk of ruin is not the sum of its sleeves' individual risks; it's smaller when sleeves diversify, larger when they don't. `assess_joint_ruin` aligns every deployed strategy on a common index, applies each one's deployment weight, sums to a single portfolio return series, and block-bootstraps **that**. Bootstrapping the summed series (rather than each sleeve independently) is the point: it preserves the *historically realised* contemporaneous co-movement between sleeves, so the joint drawdown reflects the diversification you actually had.

One subtlety worth pinning down, because it is easy to get backwards. Summing at deployment weights and *then* block-bootstrapping is identical to drawing the **same** block indices across the per-strategy return matrix and weighting after, a fixed-weight sum commutes with a shared-index resample. The word that carries the safety is *shared*. The cheap-but-wrong version resamples the realised portfolio **NAV** series directly: that bakes in whatever time-varying weights the book happened to carry and dilutes exactly the synchronised bad-regime blocks you most need to see. So the live de-risk monitor (the sidecar of Containerising the stack (see "Containerizing")) resamples the per-strategy matrix at shared indices and applies *current* weights, never the NAV series; it is the same construction as `assess_joint_ruin`, moved out of the backtest and onto the live book.



There is a deliberate hole in that picture, and naming it is more important than the method.

Danger: Diversification evaporates in the crash
Summing-then-bootstrapping preserves only the **calm-sample** correlation structure. It certifies the book as safe on a diversification benefit that historically held, and historical correlations are exactly what collapse toward 1 in a real crisis, when everything sells off together. So the base joint-ruin number is *optimistic about the tail by construction.* Titan's answer is `assess_joint_ruin_stressed`: with some probability a bootstrap block is replaced by a **crisis block** in which every sleeve is forced to co-move at a high pairwise correlation via a one-factor model, with the common factor centred on a negative drift (a coordinated drawdown). These are **stress assumptions, not estimates** (the operator's deliberate "what if correlation goes to 1 in a crash" scenario), and the stressed run should report a *higher* ruin probability than the calm one. If a portfolio only survives under calm correlations, it does not survive.

Ballast can be load-bearing for survival

One result from running these tools repeatedly reframes how we think about the boring sleeves. A defensive allocation (bonds, a low-vol line, anything that drifts up while the risk sleeves crater) contributes almost nothing to headline Sharpe. By the *return* metrics it looks like dead weight you could prune for a punchier number. But when you re-run joint ruin with that sleeve removed, `p_kill_trip` jumps: the ballast was holding the portfolio's drawdown below the kill threshold in precisely the crisis paths where the risk sleeves all drew down together.

> **Warning: War-story: pruning the ballast that was holding the floor** A portfolio-trim pass dropped a low-return defensive sleeve because it dragged the aggregate Sharpe and "wasn't earning its slot." Joint-ruin re-assessment told a different story: removing it pushed `P(kill)` over the horizon up by a large multiple, because the sleeve's negative crisis correlation was the thing keeping coordinated-drawdown paths off the kill line. The lesson is a metric mismatch: **you cannot evaluate ballast on a return metric.** Its job is to bend the *left tail*, so judge it on `p_kill_trip` and the stressed joint MaxDD, where it earns its slot many times over. We keep ballast in the book on a survival argument, not a return argument, and we never let a Sharpe-trim pass touch it without re-running ruin. (This is the canonical telling of the "ballast that was load-bearing" entry in the failure-mode catalogue (see "Failure Mode Catalogue"), which there is reached from a different angle, an ablation under selection discipline.)

Choosing horizon, block size, and sample length

These simulations have failure modes of their own. The honest constraint is that you cannot bootstrap a 10-year tail from two years of data and pretend the result is robust; each block gets re-used many times, so the tail estimate is *bootstrap-bound*, not data-rich. Titan's ruin module emits a warning when the horizon dwarfs the sample for exactly this reason. Pick parameters with the data-generating process in mind:

Parameter	Rule of thumb	Why
Block size	Long enough to span the strategy's autocorrelation ($\approx$ one month of bars for a multi-week edge)	Too short destroys serial structure; too long leaves too few distinct blocks

Parameter	Rule of thumb	Why
Horizon	The decision horizon you actually deploy over	A 1-year `P(kill)` and a 10-year `P(kill)` are different questions; label which one
Sample length	Comfortably longer than the horizon (prefer $\geq$ several years)	A horizon `>` `4×` the sample means blocks are re-used so heavily the tail is an artifact
`n_paths`	Enough that the small `p` you gate on is stable across seeds	Gating on a `1e-3` probability with 1,000 paths is reading ~1 event; cross-check seeds

That last row deserves emphasis: if your gate is "`P(kill) ≤ 1e-3`," a 1,000-path run is estimating that probability from roughly a single tail event. Re-run with several seeds, and treat any gate decided by one or two paths as a coin flip, not a measurement. When the horizon is long relative to the sample, pair the bootstrap with an analytic first-passage cross-check rather than trusting the simulation alone.

Warning: A long sample can still be tail-blind: length is not coverage
The rule above guards against a sample too *short* for the horizon. A subtler failure survives it: a sample long *enough* but **crash-free**. The block bootstrap only reshuffles blocks that exist in the data, so a tail the window omits is absent from *every* synthetic path; no resampling can manufacture a crash the sample never contained. Concretely: a $2\times$-levered SPY 60/40 bootstrapped on *post-2010* data tops out near a **$**-37.5-**$60.5%**. The post-2010 run was not merely lucky. It was *structurally incapable* of seeing the crash its window excluded. So length is necessary but not sufficient: the sample must actually *span* the regimes whose tail you are trying to price, or the honest bootstrap of the previous caveat quietly certifies a tail it could never have drawn.

That cross-check has a closed form worth carrying. Model the deployment-weighted log-equity as an arithmetic Brownian motion with the OOS drift `μ` and vol `σ` per bar, and let the kill at `K` be an absorbing barrier `b = ln(1 − K)` (negative). The probability of first-passage to `b` before horizon `H` follows from the reflection principle:

$$P(\text{hit } b \text{ before } H) \;\approx\; \Phi\left(\frac{b-\mu H}{\sigma\sqrt{H}}\right) + \exp\left(\frac{2\mu b}{\sigma^2}\right)\,\Phi\left(\frac{b+\mu H}{\sigma\sqrt{H}}\right).$$

This assumes **IID Gaussian increments**, so it deliberately ignores the serial clustering the block bootstrap exists to preserve; it is a *lower-bound sanity check*, not a replacement. Use it one way only: if the bootstrap `p_kill` comes back far *below* this analytic floor, suspect a too-short sample re-using blocks rather than a genuinely safer strategy, and widen the sample before you trust the gate.

These knobs are not independent under a re-run budget. With the honest bootstrap, total wall-clock is roughly `n_paths × horizon × per-bar strategy cost`, so doubling the horizon or the path count multiplies runtime; you cannot tighten the Monte-Carlo error and lengthen the horizon for free. The practical order: fix the **horizon** at the real decision horizon first (it is a question about the world, not a tuning knob), then trade `n_paths` against the binomial CI on `kill_trips / n_paths` until the small `p` you gate on is stable across seeds, caching reusable indicator computations across paths wherever `strategy_fn` allows.

Exercises

1. **Bootstrap the cause, not the effect.** Why is resampling a strategy's own realised P&L the wrong way to estimate its tail, and what should you resample instead?

   > **Answer: Answer**
   > Realised P&L is an *effect*: the strategy already navigated those bars (stops fired, filters dodged losers), so resampling it only reorders drawdowns it already survived; the worst synthetic path is bounded by the worst *historical* one. Resample the **underlying's returns**, `cumprod` to a synthetic price, and **re-run the strategy** on it. (Resampling the effect understated Titan's tail by up to ~an order of magnitude.)

2. **Judge ballast on the right metric.** A defensive sleeve drags aggregate Sharpe. Why might pruning it still be a mistake, and what metric judges it?

   > **Answer: Answer**
   > Ballast earns its slot on *survival*, not return: removing it can push joint `P(kill)` up by a large multiple, because its negative crisis correlation kept coordinated-drawdown paths off the kill line. Judge it on `p_kill_trip` and stressed joint MaxDD, never on a return metric.

Takeaways

- **Bootstrap the cause, not the effect.** Resample the *underlying's* returns, `cumprod` to a synthetic price, and re-run the strategy on it. Resampling the strategy's realised P&L understated Titan's tail risk by a large multiple (on some strategies an order of magnitude) because it can only reshuffle drawdowns the strategy already survived.
- **Preserve structure.** Block (or stationary) bootstrap to keep serial correlation; `shared_block` to keep cross-asset correlation. IID shuffles flatter trend, carry, and any cross-asset thesis.
- **Use the right MaxDD gate.** Absolute `P(MaxDD > X)` for market-neutral/tactical strategies; a *relative* (vs buy-and-hold) gate for long-only and ballast sleeves, where the underlying itself fails the absolute version.
- **Risk of ruin is the real verdict.** Translate the drawdown distribution into `P(kill)` at a *specific deployment weight* against a *specific kill threshold*, over a *labelled horizon*, and bind those parameters to the gate so doctrine can't drift out from under the threshold.
- **Joint ruin, then stress it.** Assess the whole book together, then re-run with crisis blocks that force correlations toward 1, because real-crisis diversification is exactly what calm-sample correlations cannot see.
- **Judge ballast on the tail.** Defensive sleeves contribute little to Sharpe and a lot to survival; evaluate them on `p_kill_trip` and stressed MaxDD, never on a return metric, or you'll prune the thing holding the floor.

This chapter turned tail fear into a survival probability. The next steps put that probability to work: Capacity, crowding & the size at which the edge stops being yours (see "Capacity And Crowding") asks whether the deployment weight ruin is evaluated at is even *achievable* in the market before any of it counts; The sanctuary window & the decision matrix (see "Sanctuary Decision Matrix") shows how ruin sits alongside walk-forward and deflation in the deploy/reject decision; Position sizing: Kelly & vol-targeting (see "Position Sizing Kelly") sets the deployment weight that risk of ruin is evaluated *at*; and Layered safety (see "Layered Safety") builds the live kill switch whose threshold this chapter has been simulating against. The full ledger of bugs that bought these rules lives in the failure-mode catalogue (see "Failure Mode Catalogue").

11. Capacity, crowding & the size at which the edge stops being yours

Every chapter so far has sized a strategy as if the market would hand you any quantity at the price on your screen. Kelly (see "Position Sizing Kelly") computes a leverage; vol-targeting scales it; the allocator (see "Allocator Correlation Dial") splits the book by risk; risk of ruin (see "Tail Risk And Ruin") evaluates survival at a deployment weight. Not one of them asks the question that decides whether the edge survives contact with real size: **can the market actually absorb the notional this math just produced, without your own trading moving the price against you?** It cannot, beyond some size, and that size is the most under-discussed number in retail quant. A "production" book that never asks *how much can this take?* has a hole exactly where a paper book ramped toward live size is most exposed.

This chapter is about **capacity**: the size at which your own market impact, or a crowd trading the same signal, consumes the edge you measured at toy size. It has three moving parts: a capacity *ceiling* you impose on yourself through market impact, the cost-aware re-validation that turns that ceiling into a deploy/no-deploy verdict *at deployed size*, and **crowding**, the ceiling other people impose on you by trading the same thing. The unifying identity is simple and ruthless: **turnover** $\times$ **size = impact**, so capacity is never a property of the edge alone; it is a property of the edge, the turnover, and the liquidity, together.

The capacity ceiling: ADV, participation, and the square-root wall

The execution chapter (see "Execution And Costs") introduced market impact as a cost term: slippage that *grows with participation*, where participation is your order size as a fraction of the instrument's average daily volume (ADV). The form is sub-linear (roughly the **square root** of participation) and that shape is the whole story of capacity:

```
# from cost_realism.apply_sqrt_impact_slippage: slip GROWS with
 ↪  participation = order / ADV
slip = coef * np.sqrt(np.clip(participation, 0.0, None))   #
 ↪  participation = clip_notional / ADV
cost = turnover * slip                                      # paid every
 ↪  time you trade
```

At toy size, participation is a rounding error and impact is invisible, which is exactly why a backtest run at notional ≈ 0 reports an edge that looks

free. As you scale up, participation rises and the per-unit impact rises with its square root, so the *total* impact you pay scales faster than linearly with the size you're trying to deploy. Somewhere on that curve, **the impact cost equals the gross edge**, and every dollar beyond that point trades at a loss. That crossover is your capacity ceiling: the most you can run before your own footprint eats the alpha.

Two consequences follow immediately, and both are routinely missed:

- **Capacity is a function of turnover, not just AUM.** Because impact is paid *per trade* (`turnover × slip`), a sleeve that turns over weekly hits its ceiling at a far smaller book than one that turns over yearly, even at the same edge and the same ADV. A fast signal is a small-capacity signal, structurally. This is why the allocator's turnover discipline (see "Allocator Correlation Dial") (a no-trade band, a turnover penalty) is not just a cost saving; it is a *capacity* lever: every basis point of turnover you don't spend is capacity you buy back.
- **The sizing chain has no ceiling in it.** Kelly hands back a leverage that can be 75× (see "Position Sizing Kelly"); vol-targeting converts it to a notional; the allocator weights it. And none of those steps knows the ADV of the thing being traded. The math will cheerfully hand you a clip the market cannot absorb in a day, with no error and no warning. Capacity is the cap the sizing chain is missing.

Turn the ceiling into a verdict: re-run the cost-aware WFO at deployed size

A capacity ceiling is only useful if it gates a decision. The mechanism already exists: the execution chapter's (see "Execution And Costs") realistic-cost gate re-runs the bootstrap-Sharpe CI on the *net-of-cost* series and requires the lower bound to stay positive. Capacity makes one change to it: **evaluate the impact term at the participation your deployed size actually implies**, not at a fixed bps.

```
# Capacity-aware cost gate: the slippage term is evaluated at the
↪  participation
# implied by the DEPLOYED clip, so the verdict becomes size-conditional.
for clip in [size_small, size_target, size_2x]:
    participation = clip / adv
    net = apply_sqrt_impact_slippage(gross_returns, turnover,
↪  participation)
    net = apply_cost_model(net, turnover, cost_model)        #
↪  spread/commission too
    ci_lo, _ = bootstrap_sharpe_ci(net, periods_per_year)
    print(clip, "deployable" if ci_lo > 0 else "OVER CAPACITY")
```

The result is the uncomfortable truth a single backtest hides: **deploy/no-deploy is not a property of the strategy, it is a property of the strategy at a size.** A sleeve can be a clean pass at the minimum size that clears commission and an outright reject at the size someone actually wants to run, because the impact term that was a rounding error at the small clip is the entire edge at the large one. The honest response to "it fails at the size I want" is *deploy it smaller* (at or below its capacity ceiling), not to wish the impact away. An edge run past its capacity isn't a smaller edge; it's a negative one wearing the backtest's optimism.

> **Note: Capacity is the sixth decision axis, and it lives at deployment, not research**
> The decision matrix (see "Sanctuary Decision Matrix") scores five axes (CI_lo, DSR, Monte-Carlo drawdown, sanctuary, noise) and every one of them is a property of the *return series*, computed with no reference to how much capital you'll deploy. Capacity is structurally different: it is the one axis you **cannot** evaluate in research, because it only exists once you fix a size and an ADV. So treat it as a **sixth axis evaluated at deployment**, a gate that sits alongside the research verdict rather than inside its count-of-best ladder. A strategy can be a five-axis `DEPLOY` and still fail the capacity axis at your intended size, and that combination means "deploy, but smaller," not "deploy as planned." Check it *before every ramp tranche* (see paper to live (see "Paper To Live")), because participation rises with each step up and a clean small-size replay never guarantees a clean larger one.

Crowding: the capacity ceiling other people impose

ADV-participation is the ceiling *your own* footprint imposes. **Crowding** is the ceiling imposed by everyone else trading the same edge, and it is both more dangerous and harder to measure, because you cannot see other people's positions.

Two distinct harms hide under the one word:

- **Alpha decay.** When capital crowds into a published or widely-rediscovered signal, the signal is arbitraged down: more money chasing the same forecast compresses the very mispricing the forecast exploits. A real edge has a half-life, and crowding shortens it. You cannot observe the crowd directly, but you *can* observe the symptom (a live edge decaying against its research baseline (see "The Learn Loop")), which is why the Learn-loop's edge-decay detection is also a crowding detector.

- **Correlated exit.** The more dangerous harm is in the tail. A crowded trade is one where everyone holds the same position and everyone's risk model says "reduce" at the same moment, so the unwind is a stampede: the de-grossing *is* the crash, the exact mechanism the correlation dial (see "Allocator Correlation Dial") exists to fade. Crowding therefore makes a strategy's drawdowns fatter-tailed and more synchronised with everyone else's *precisely* when diversification is supposed to save you: the same blindness the stressed joint-ruin sim (see "Tail Risk And Ruin") and the factor-exposure decomposition (see "Factor Beta Exposure") are built to surface.

The defensive posture is humility about what you can know: assume any edge you found from a public source (a paper, a forum, a popular indicator) is more crowded than your backtest reflects, treat capacity estimates on it as optimistic, and lean on a *diversity* of low-correlation edges rather than scaling a single crowded one. Crowding is a capacity sibling because it does the same thing capacity does (shrink the size at which the edge is yours), just on a clock you don't control.

Warning: War-story: the small-edge sleeve that was deployable only at toy size
A slow cross-asset sleeve cleared every research gate with a positive lower bound and went to paper. Its backtest, like all backtests, was run at negligible notional, so its impact cost was effectively zero and its net edge looked clean. The first time anyone asked "what's the capacity?", the answer was sobering: re-running the cost gate with the slippage term evaluated at the *participation* a real allocation would imply (a meaningful fraction of the substitute instrument's ADV, on a leg that wasn't especially liquid) pushed the net lower bound toward zero an order of magnitude below the book size someone had in mind. The edge was real; it was just *small and shallow-pooled*, deployable at a fraction of the intended size and a reject above it. The rule it bought: **a backtest's edge is quoted at zero participation; the deployable edge is quoted at your size, and for a small-edge sleeve those are different strategies.** Titan models the sqrt-impact slippage and can run the size-conditional cost gate, but it has **no explicit capacity-ceiling estimate, no ADV cap in the sizing chain, and no crowding monitor**; so this is, as ever, a designed defence the primitives support, not a shipped control.

The Titan example: primitives present, the ceiling absent

Honesty first, as every chapter in this part insists.

Piece	Status	What it does / doesn't
Sqrt-impact slippage model	**shipped** (`cost_realism.apply_sqrt_impact_slippage`)	impact = `turnover × impact_coef × sqrt(participation)`; the curve a capacity ceiling reads off
Cost-aware deploy gate	**shipped** (`realistic_cost_gate`)	re-runs net `CI_lo > 0`; capacity just feeds it a *size-dependent* slippage
Explicit capacity ceiling (the deployable-size estimate)	**not built**	nothing computes "the size at which impact eats the edge"
ADV / participation cap in sizing	**not built**	Kelly/vol-target/allocator hand back a notional with no liquidity ceiling
Capacity axis in the decision matrix	**not built**	the verdict is size-independent; deployment size is chosen by other means
Crowding / signal-decay monitor	**partial**	the Learn-loop (see "The Learn Loop") edge-decay detector is the nearest tell; no positioning/crowding measure

The honest summary: Titan has the *cost* primitive (sqrt-impact) and the *gate* primitive (the realistic-cost CI), so it can answer "is this deployable at size X?" for a given X, but it does not yet *estimate the ceiling*, *cap the sizing chain at it*, or *score it as an axis*, and it has no direct read on crowding. The pieces are in the repo; assembling them into a capacity ceiling that the sizing chain and the deploy gate both respect is the work this chapter specifies.

Exercises

1. **Why deploy/no-deploy depends on size.** A sleeve passes the cost gate at minimum size and fails at 10× the size, with an identical edge. What changed?

   Answer: Answer
   The **market-impact** cost grows with the $\sqrt{\ }$ of participation (clip/ADV), so the impact term that was a rounding error at small size becomes the whole edge at large size. The verdict is *size-conditional*; the honest fix is to deploy *smaller* (at or below the

capacity ceiling), not to wish the impact away.

2. **Turnover and capacity.** Two sleeves share an edge and an ADV; one turns over weekly, one yearly. Which has the smaller capacity, and why?

Answer: Answer
The **weekly** one: impact is paid *per trade* (`turnover × slip`), so a fast sleeve hits its ceiling at a far smaller book. A no-trade band and a turnover penalty are therefore *capacity* levers: turnover you don't spend is capacity you keep.

Takeaways

- **The sizing chain has no liquidity ceiling in it.** Kelly, vol-targeting, and the allocator all hand back a notional with no reference to ADV. Capacity is the cap they're missing: the size at which your own market impact equals the edge.
- **Capacity = edge × turnover × liquidity, not AUM alone.** Impact is paid per trade and grows with the $\sqrt{\ }$ of participation, so a fast-turnover sleeve hits its ceiling at a far smaller book than a slow one. Turnover discipline (a no-trade band, a turnover penalty) is a capacity lever, not just a cost saving.
- **Deploy/no-deploy is a property of the strategy *at a size*.** Re-run the cost-aware WFO with the slippage term evaluated at your deployed participation; a clean small-size pass can be an over-capacity reject at the size you want. The fix is to deploy *smaller*, never to wish the impact away.
- **Capacity is the sixth decision axis, and it lives at deployment.** The five research axes are size-independent; capacity can only be scored once a size is fixed, so gate it alongside the matrix and re-check it before every ramp tranche.
- **Crowding is the capacity ceiling you don't control.** Others trading your edge decays the alpha (watch the live-vs-research drift) and synchronises the exit (fattens the tail, the de-gross-into-the-same-move trap). Treat any public-source edge as more crowded than the backtest shows, and prefer a diversity of edges over scaling one.

All participation coefficients, ADV fractions, and capacity thresholds here are illustrative; the real liquidity of each traded instrument and the deployed sizes are configuration, not edge. The transferable part is

the *method*: read the ceiling off the $\sqrt{}$-impact curve, turn it into a size-conditional cost gate, score it as a deployment axis, and respect crowding as the ceiling you can't measure.

This chapter put a ceiling on the size the sizing chapters compute. It feeds forward into the allocator (see "Allocator Correlation Dial"), where turnover-aware construction (a no-trade band, a turnover penalty) is the lever that buys capacity back, and into paper to live (see "Paper To Live"), where the capacity check gates each ramp tranche. It reads its impact curve from the execution layer (see "Execution And Costs") and its crowding tell from the Learn loop (see "The Learn Loop"); next, the sanctuary & decision matrix (see "Sanctuary Decision Matrix") is where the research verdict this capacity axis sits beside is actually computed.

12. The sanctuary window & the decision matrix

You have walked the data forward, deflated for the optimiser's tries, and stressed the path with Monte Carlo. Every gate is green. Now someone has to say a single word out loud, *deploy* or *don't*, and stake real money on it. This is the moment most research processes quietly fall apart, because the decision is made by a person, in a good mood, looking at a number they spent three weeks producing.

This chapter is about two disciplines that take that decision out of the researcher's hands. The first is a **sanctuary**: a held-out window the research loop is structurally forbidden from seeing, spent once, at the very end. The second is a **decision function**: a total mapping from evidence to a verdict, so the go/no-go is computed, logged, and reproducible, not argued. Together they answer the question every prior chapter set up but none of them closed: *given all this evidence, do we ship it?*

The principle: a held-out window beats one more fold

Walk-forward analysis (see Walk-forward that's actually out-of-sample (see "Walk Forward")) is the workhorse of honest backtesting. But it has a leak that no amount of fold discipline can plug: **you, the researcher, see the out-of-sample results.** You run the WFO, the OOS Sharpe disappoints, you change the lookback, you re-run. Each iteration uses OOS performance as a training signal, for *you*. After fifty iterations the "out-of-sample" window has been fit as thoroughly as the in-sample one, just by a slower optimiser with a worse loss function (your judgement). Beating your own optimiser (see "Deflated Sharpe") corrects the *count* of explicit trials; it can't correct the trials you ran with your eyes.

The fix is not statistical, it is procedural. Carve off the most recent stretch of history (a year of calendar time is a reasonable default, long enough to span a regime or two) *before any fold is constructed*, and put it in a vault. The research loop never touches it. Every parameter sweep, every CV fold, every "let me just try one more thing" happens on the **visible** portion. When a candidate is finally finished, frozen, no more knobs, you spend the sanctuary exactly once, as a final-validation pass.

> **Note: Why once, and why last**
> The sanctuary is a one-shot instrument. The moment you look at it, react to it, and re-tune, it stops being held out; it becomes another fold, and you are back where you started. The discipline only works if "spend the sanctuary" is the *last* step before the verdict, and if a failed

sanctuary sends the candidate to the bin, not back to the lab. If you cannot resist re-tuning, the sanctuary has no statistical meaning; it is theatre.

A held-out window beats one more CV fold for a reason worth internalising: a CV fold tests *whether the model generalises across resamples of the data you've been staring at*. A sanctuary tests *whether the edge survives into time you could never react to*. The first measures overfitting to a sample; the second, overfitting to a process, which includes you. Live trading is exactly that: data you cannot react to before it prints. The sanctuary is the closest thing to a dry run of deployment that history allows.

The Titan example: slicing the vault

Titan's sanctuary slice is deliberately boring. Calendar-time, by the last timestamp in the frame, done before anything else (the `months` default below is illustrative, a year, and is the kind of value a strategy class can override in its pre-registration directive (see "Typology"), not a tuned edge parameter):

```
@dataclass(frozen=True)
class SanctuarySlice:
    visible: pd.DataFrame          # everything the research loop may see
    sanctuary: pd.DataFrame        # the vault - spent once, at the end
    sanctuary_start: pd.Timestamp  # boundary timestamp, logged for audit
    sanctuary_end: pd.Timestamp
    months_held_out: int

def slice_sanctuary(df, months: int = 12) -> SanctuarySlice:
    end = df.index[-1]
    sanctuary_start = end - pd.DateOffset(months=months)
    visible   = df[df.index <  sanctuary_start]
    sanctuary = df[df.index >= sanctuary_start]
    ...
```

Two design choices carry weight. It slices by **calendar months**, not bar count: twelve months is twelve months whether the asset trades hourly or daily, so the held-out *regime exposure* is comparable across strategy classes. And it records the exact `sanctuary_start` timestamp on the result, so an auditor can later confirm that the visible/sanctuary boundary was fixed before fold construction and never moved. A boundary that drifts between runs is a boundary that has been peeked at.

The twelve-month default is a luxury a long series can afford; a short one cannot. A one-shot twelve-month sanctuary on five years of data

spends roughly **20% of training history** and leaves the divergence distribution only a handful of effective draws once windows overlap; the $K = 3$ disjoint-window version below consumes three years outright. On short or post-regime-break history that is a real cost, so shrink the window (with the overlap caveat noted in the divergence test) or drop K before you starve the training set. And if you cannot afford a sanctuary that spans at least one regime change, say so *in the verdict*: a held-out window too short or too calm to contain a regime has not validated much, and the honest move is to flag it, not to spend it and pretend.

The lucky-flag divergence test

A passing sanctuary is necessary but not sufficient, because of a subtler trap: the held-out window might have been an *easy* twelve months. A trend strategy validated over a year that happened to trend beautifully has told you it works in trending regimes, which you already knew, not that it works. The sanctuary's positive Sharpe could be regime luck wearing the costume of out-of-sample evidence.

So Titan asks a second question: **is the sanctuary's Sharpe unusual relative to history?** Build the distribution of every same-length rolling-window Sharpe over the visible portion, then locate the sanctuary in that distribution. These windows are *overlapping*, stepped at roughly a quarter of the window length (~75% overlap), not disjoint: the overlap buys sample size for the distribution at the cost of independence, which is fine here because we only want a percentile rank, not a CI. (The disjoint construction is the separate K-window hold-out below; don't conflate the two.)

```
arr = np.asarray(historical_window_sharpes)   # rolling 12-mo Sharpes,
 ↪  visible portion
sh_sanc = sharpe(sanctuary_returns, periods_per_year=ppy)
pct = float((arr < sh_sanc).mean())           # percentile of the
 ↪  held-out window

return DivergenceTest(
    sanctuary_sharpe=sh_sanc,
    percentile=pct,
    lucky_flag=pct >= 0.95,    # top 5% of historical windows ->
 ↪  regime-specific
    unlucky_flag=pct <= 0.05,  # bottom 5% -> adverse regime, NOT a veto
)
```

Warning: A NaN percentile is a no-op, not a pass
The test needs a distribution to rank against, and a short or newly-

listed series may not give it one. If fewer than two window-lengths of history survive, or fewer than four rolling windows can be formed, the code cannot build the distribution and returns a **NaN percentile with `lucky_flag` silently `False`**. That is not the strategy passing the luck check; it is the luck check failing to run, which means the sanctuary credit is *unguarded*. Treat a NaN percentile as a reason to shrink the window or defer the verdict, never as a clean bill: a candidate that quietly skipped its own divergence test should not collect the credit for surviving it.

The asymmetry between the two flags is the point, and it follows directly from *suspicion over celebration*:

- **`lucky_flag`** (top 5%) is bad news. If the held-out year ranks among the best twelve-month windows the strategy ever had, its OOS "validation" is likely measuring a favourable regime, not a durable edge. This *downgrades* the verdict.
- **`unlucky_flag`** (bottom 5%) is *not* a veto. A strategy that survived its worst regime in the sanctuary, with the long-run edge still intact in the visible history, may be more trustworthy, not less. It is information for the human, not an automatic kill.

A single twelve-month window is one draw, and one draw can be lucky or unlucky by pure regime. So Titan also runs a multi-window version: hold out the K most-recent disjoint windows, divergence-test each against *its own* preceding history, and take a majority vote.

```
result = multi_window_sanctuary_test(
    strategy_returns, periods_per_year=ppy,
    n_windows=3, months=12,
    block_size=block,   # stationary bootstrap -> honest pooled CI
)
# result.lucky_flag fires only if a MAJORITY of windows landed in the top
 ↪  5%
# result.sharpe_ci_lo is the bootstrap lower bound on the POOLED held-out
 ↪  Sharpe
```

This turns an $n = 1$ luck check into a K-of-K vote, and the bootstrapped CI on the pooled sanctuary Sharpe (use the stationary block bootstrap; see A backtest you can trust (see "Backtest You Can Trust") on why the IID version lies) tells you how firmly the held-out edge clears zero. The aggregate `lucky_flag` is what feeds the decision function as a downgrade.

Danger: The sanctuary defends a candidate against itself, not a factory against itself
Everything above defends *one* candidate against its own re-tuning. It does nothing about selection at the *candidate* level. Run forty candidates each through an independent 5% sanctuary and you should expect a couple to clear a positive held-out window by chance alone. The 5% thresholds are per-candidate, not family-wise corrected, and beating your own optimiser (see "Deflated Sharpe") only deflates the *optimiser's* trials within a candidate, not the count of candidates you funnelled through the vault. A research program that screens many ideas re-introduces exactly the multiple-comparisons problem the sanctuary was built to kill, one level up. A portfolio-level program needs a portfolio-level correction (or a far stricter per-candidate bar) layered on top of the sanctuary, and nothing in this chapter's matrix supplies it for you.

Tip: Sharpe is the gate, not the whole read on the sanctuary
We gate the held-out window on Sharpe because it is the axis the matrix scores, but Sharpe alone hides the path. When you spend the sanctuary, look at the held-out window's **max drawdown and recovery time (Calmar)**, its **Sortino** (did the disappointment come from downside, or from missed upside?), and its **worst-slice tail (CVaR)**: the same battery from Beyond Sharpe: the metric suite (see "Metric Suite"). A sanctuary that clears the Sharpe gate but carves a drawdown deeper than the strategy class tolerates is still a problem the single number won't surface.

Warning: War-story: the candidate that passed every gate, then the sanctuary collapsed it
A cross-asset signal cleared the whole battery on the visible history: the bootstrap CI lower bound on stitched OOS Sharpe was comfortably positive, the deflated-Sharpe probability was high, and the Monte Carlo drawdown distribution was inside the class threshold. By every number we had, it was a deploy. Then we spent the sanctuary. The held-out year did not just disappoint; it *lost money*, and the divergence test put the strategy's strong visible-history windows in the top tail: the "edge" had been concentrated in a regime that the held-out year simply didn't repeat. Had the sanctuary been just one more CV fold we'd have re-tuned around it and shipped. Because it was a one-shot vault, the loss was disqualifying. The rule it bought: **an out-of-sample hold-out loss is not fungible with a hygiene-axis miss.** A strategy that

lost money on data it never saw coming cannot be a DEPLOY, no matter how clean the rest of the card. We encoded that as a hard guard, below.

Making the verdict a function, not a judgement call

Here is the load-bearing idea of the chapter. The output of all this evidence-gathering should not be a meeting. It should be a **function**, total, deterministic, and logged, that maps an evidence vector to one of a small set of verdicts. Total means it is defined for *every* possible input; there is no combination of evidence that produces a shrug. The word `UNDETERMINED` should be impossible to return, because "we couldn't decide" is exactly the gap where a researcher's optimism leaks back in.

Titan's `decide()` takes five axes of evidence and returns one of five verdicts.

Axis	What it measures	"best" means	Source chapter
CI_lo	95% bootstrap lower bound on stitched OOS Sharpe	lower bound > 0	Backtest you can trust (see "Backtest You Can Trust")
DSR	Deflated-Sharpe probability (corrected for trials)	prob $\geq$ ~0.95 (illustrative)	Beating your own optimiser (see "Deflated Sharpe")
MC	P(MaxDD > class threshold) from block Monte Carlo	P $\leq$ the class pass-threshold	Tail risk & risk of ruin (see "Tail Risk And Ruin")
Sanctuary	Realised Sharpe on the held-out window	Sharpe > 0	this chapter
Noise	Degradation under input-price noise injection	passes mean *and* worst-case	Failure-mode catalogue (see "Failure Mode Catalogue")

Note: A sixth axis the matrix can't score: capacity
Every axis above is a property of the *return series*; none of them knows how much capital you will deploy. There is a sixth axis that decides whether the verdict survives contact with real money (**capacity**), and it is deliberately *absent* from `decide()` because it cannot be evaluated in research: it exists only once you fix a deployment size and the instrument's liquidity. Treat it as a deployment-stage gate that sits *beside* this matrix. A clean five-axis `DEPLOY` can still be over capacity at the size you want, and that combination means "deploy smaller," not

"deploy as planned." The full treatment (reading the ceiling off the market-impact curve and re-running the cost gate at deployed size) is Capacity, crowding & the size at which the edge stops being yours (see "Capacity And Crowding").

Each axis is classified into three levels, `best`, `mid`, `worst`, by thresholds that live in one place (`GateThresholds`) and can be overridden per strategy class via a pre-registration directive (see "Typology"), never edited ad hoc after seeing a result. The directive is committed *before* the data-loading commit (the same blindness that fixes the visible/sanctuary boundary), so the thresholds and the slice are frozen by the same act; an override that appears after the boundary was set is, by construction, peeking. The verdict then comes from a count-of-best ladder:

```
verdict_by_n = {
    5: Verdict.DEPLOY,                  # all five axes at best
    4: Verdict.CONDITIONAL_WATCHPOINT,  # one weak axis
    3: Verdict.TIER_UNCONFIRMED,        # promising, not proven
    2: Verdict.SUSPECT,
    1: Verdict.RETIRE,
    0: Verdict.RETIRE,
}
base = verdict_by_n[n_best]
```

Note: Why five verdicts, not two
A binary deploy/reject throws away information. `CONDITIONAL_WATCHPOINT` says *ship it, but instrument the one weak axis and watch it in production*. `TIER_UNCONFIRMED` says *the edge is plausible but unproven: paper-trade it, don't fund it*. `SUSPECT` and `RETIRE` separate *needs work* from *bin it*. These map onto operational actions (see Paper-to-live promotion (see "Paper To Live")), which a yes/no can't. The values are illustrative labels; what matters is that each is a distinct, pre-agreed action.

The ladder isn't enough: guards

A pure count-of-best has a flaw the audit caught: it cannot tell a *catastrophe* from a *mediocrity*. A strategy that lost money on the sanctuary scores the same "not best" on that axis as one that was merely not-best on the noise hygiene check. Counting treats them as fungible. They are not. So two guards sit on top of the ladder and take the **more conservative** of the ladder verdict and a cap:

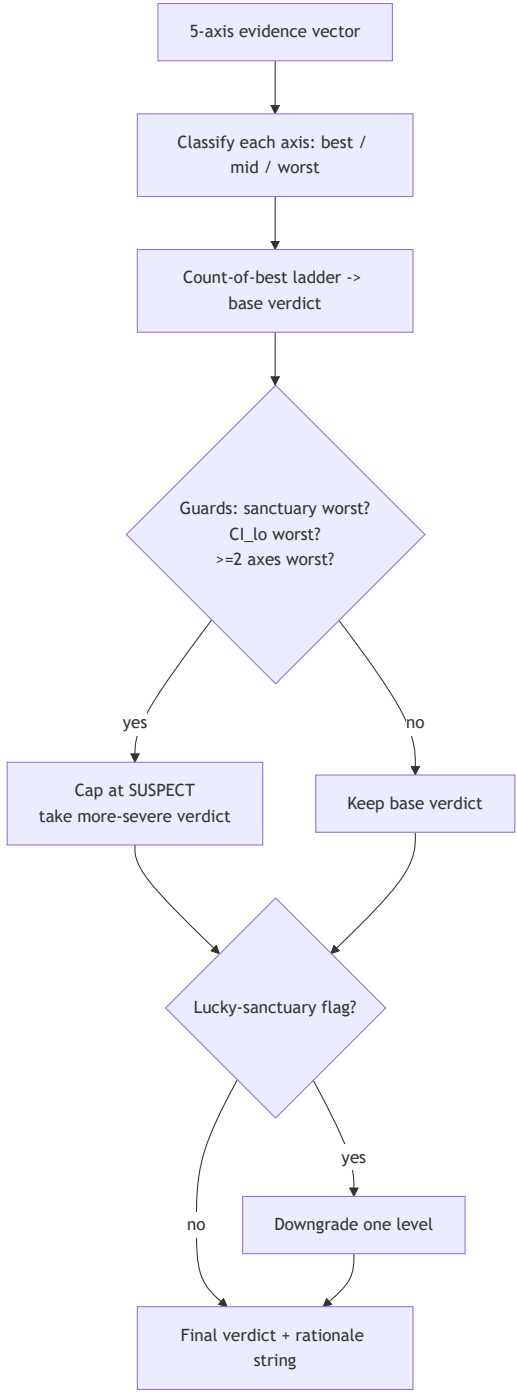
```
veto_reasons = []
if n_worst >= 2:                        # multiple catastrophic axes
    veto_reasons.append(">=2 axes worst")
if sanc == "worst":                     # lost money on the held-out year
    veto_reasons.append("sanctuary worst (OOS hold-out loss)")
if ci == "worst":                       # edge CI strongly negative
    veto_reasons.append("CI_lo worst")
verdict = _more_severe(base, Verdict.SUSPECT) if veto_reasons else base
```

The two dealbreaker axes, sanctuary and CI_lo, are the dominant *out-of-sample edge* signals. A `worst` on either caps the verdict at `SUSPECT` regardless of how clean the other four are. A single `worst` on the noise axis, by contrast, is deliberately *not* a dealbreaker: a four-best card with a noise miss stays `CONDITIONAL_WATCHPOINT`, because noise fragility is a hygiene concern, not evidence the edge is fake. The guards encode a value judgement (*which failures are which kind*) once, in code, instead of re-litigating it per candidate.

Finally, the lucky-sanctuary downgrade. If the multi-window divergence test returned a majority `lucky_flag`, the OOS validation is regime-specific, so it does not earn full credit: apply a one-level downgrade on top of the ladder and guards:

```
if lucky_flag:                          # majority of held-out windows in
↪  top 5%
    verdict = _downgrade_one(verdict)  # DEPLOY ->
↪  CONDITIONAL_WATCHPOINT, ...
```

A would-be `DEPLOY` whose held-out year was suspiciously good becomes `CONDITIONAL_WATCHPOINT`: ship it, but treat the validation as provisional and watch it. The whole pipeline, from evidence to verdict:


5-axis evidence vector
Classify each axis: best / mid / worst
Count-of-best ladder -> base verdict
Guards: sanctuary worst? CI_lo worst? >=2 axes worst?
yes
no
Cap at SUSPECT take more-severe verdict
Keep base verdict
Lucky-sanctuary flag?
yes
no
Downgrade one level
Final verdict + rationale string

Example: One verdict, traced end to end (illustrative)
A cross-asset candidate posts five axes: `CI_lo = +0.05` → **best**; `DSR = 0.91` → **mid** (below the ~0.95 best bar); `MC` drawdown inside its class threshold → **best**; `sanctuary Sharpe = +0.6` → **best**; `noise` fragile under perturbation → **worst**. That is **n_best = 3, n_worst = 1**, so the count-of-best ladder hands back a base verdict of `TIER_UNCONFIRMED`. The guards check next: sanctuary `worst`? no. `CI_lo` `worst`? no. Two-or-more `worst`? no. So no guard fires, the base stands. Finally the lucky-sanctuary flag: the held-out year landed in the top 5% of historical windows, so a one-level downgrade applies. The function emits the whole chain as a rationale string: `"n_best=3 → TIER_UNCONFIRMED; guards: none; lucky_downgrade applied; CI_lo=best DSR=mid MC=best SANC=best(lucky) NOISE=worst"`. *That string*, not a meeting, is the deploy artefact: reproducible, logged, explainable six months later. (Note `DSR=mid` contributes nothing to the count: the ladder only counts `best`, and `mid` is inert except that it keeps an axis out of the `worst` bucket the guards police.)
The cutoffs the guards hinge on are real numbers, not vibes: the "worst" band is roughly **$**-$0.3 ** for the sanctuary Sharpe and ** -**$0.2** for `CI_lo` (illustrative `GateThresholds`). They're worth seeing, because a guard is only as meaningful as its cutoff: "sanctuary worst" means *lost real money on the held-out year*, not merely underperformed.

Note: An unlucky sanctuary that *lost money* is still a cap, on purpose
A sharp objection: the divergence test calls `unlucky_flag` (a held-out year in an adverse regime) "information, not a veto", yet a *negative* sanctuary Sharpe trips the `sanctuary == worst` guard and caps the verdict at SUSPECT regardless. Both are true, and the tension is a deliberate value choice, not a contradiction. The `unlucky_flag` only softens how much *credit* a merely-disappointing window earns; the guard fires on a *realised loss* on data the strategy never saw, whatever the regime. The framework would rather eat a **false negative** (bin a good strategy that drew an unlucky-and-losing year) than risk a false positive (deploy on a hold-out that actually lost money). That asymmetry (suspicion over celebration, applied to the one window you can't re-tune against) is the sanctuary's whole purpose.

Why a *total* function matters

Two properties make this worth the ceremony. First, **explainability**: `decide()` returns not just a verdict but a rationale string, which axes passed, which failed, whether a guard fired, whether the lucky downgrade applied,

written straight into the audit log. Six months later, when someone asks why a strategy was retired, the answer is a record, not a memory. Second, **reproducibility**: the same evidence vector always yields the same verdict: no Friday-afternoon DEPLOY that would have been a Monday-morning SUSPECT. The function is the contract; the human's job is to produce honest inputs, not to overrule the output.

Danger: War-story: the verdict that needed a human override, and the human was wrong
Before the matrix was total, the framework could return `UNDETERMINED` when the axes disagreed, kicking the decision to a person. On one borderline candidate, the in-sample numbers were seductive and the disagreement was on a single axis. The reviewer "used judgement," read the disagreement as noise, and waved it through to a small live allocation. The axis it had quietly failed was the one that mattered, and the position bled until a routine review caught it. The lesson was not "trust people less"; it was that **`UNDETERMINED` is a design defect, not a state of the world.** Every evidence vector must map to an action. We deleted the escape hatch: the matrix became total by construction, disagreement became a *defined* low verdict (`SUSPECT` or `TIER_UNCONFIRMED`, paper-only), and the human lost the ability to override the function with a hunch. The override that touched live capital was the bug; removing the override was the fix.

Exercises

1. **Why a sanctuary beats one more fold.** What does a held-out sanctuary test that a cross-validation fold cannot?

   Answer: Answer
   Overfitting to your *process*, including your own re-tuning. A CV fold tests generalisation across resamples you've been staring at; the sanctuary, spent once and *last*, tests survival into time you could never react to. It only works if a *failed* sanctuary bins the candidate, never feeds back into the lab.

2. **The verdict is a total function.** Why must `decide()` never return `UNDETERMINED`, and what caps an otherwise-clean card at SUSPECT?

   Answer: Answer
   `UNDETERMINED` kicks the call to a hopeful human, exactly where optimism leaks back in, so every evidence vector must map to

an action. A `worst` on the **sanctuary** (an OOS hold-out loss) or on **CI_lo** caps the verdict at SUSPECT regardless of the other four axes; a hygiene (noise) miss deliberately does not.

Takeaways

- **A sanctuary is a held-out window the research loop never sees, spent once, last.** It beats one more CV fold because it tests overfitting to your *process*, including your own re-tuning, not just to a sample. The discipline only works if a failed sanctuary bins the candidate instead of feeding back into the lab.
- **A good sanctuary Sharpe can still be regime luck.** The divergence test ranks the held-out window against the historical distribution; a top-5% `lucky_flag` downgrades the verdict, while a bottom-5% `unlucky_flag` is information, not a veto. One window is one draw: use a K-window majority vote and a stationary-bootstrap CI on the pooled held-out Sharpe.
- **Make the go/no-go a total function, not a meeting.** Five axes (CI_lo, DSR, Monte Carlo drawdown, sanctuary, noise) $\rightarrow$ five verdicts. `UNDETERMINED` must be impossible; "we couldn't decide" is where optimism leaks back in.
- **Counting passes isn't enough: encode which failures are catastrophic.** An out-of-sample hold-out loss or a strongly-negative edge CI caps the verdict regardless of the rest; a hygiene miss does not. Put that value judgement in code once, not in every review.
- **The function's output is auditable and reproducible.** A rationale string and a fixed mapping mean the same evidence always yields the same verdict, with a logged reason: no mood-dependent deploys.

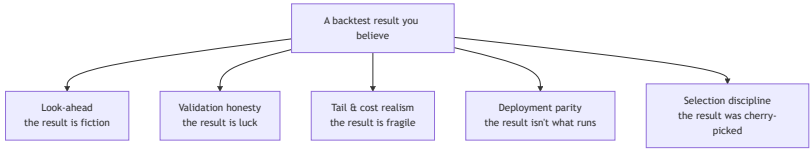
A clean verdict is still only a research verdict. The next chapter, The failure-mode catalogue (see "Failure Mode Catalogue"), collects the bugs that taught every gate in Part II its threshold; and when a candidate does clear the matrix, The strategy-class contract (see "Strategy Class Contract") governs how it crosses from research into production without quietly changing what was validated.

13. War-stories: the failure-mode catalogue

Every rule in this book was bought with a bug. The disciplines in the preceding chapters (shift the signal, gate on the lower bound, deflate for the sweep) did not arrive as wisdom; they arrived as post-mortems. This chapter is the ledger: it collects the failures, sanitised, and for each one names the consequence and the rule it forced into the framework.

The reason to keep a catalogue rather than a tidy list of best practices is that **the same bug class recurs forever** if you only fix instances. A researcher who never saw the same-bar leak will write it again next quarter, in different pandas, and it will look like ordinary code. Best practices are advice; a catalogued failure mode is a gate. And note who caught these: mostly not the author. You do not find your own look-ahead bug by re-reading the code that contains it; you find it with an independent path that disagrees.

We group the catalogue into five families, because the root causes cluster:



Note: How to read the A-codes and V-eras
Two stable labels recur across the book, and they're worth defining once, here, since this catalogue is where they originate. An **A-code** (`A1`, `A6`, `A11`, ...) is a stable identifier for a catalogued *audit finding / failure mode* (an entry in this chapter), so a fix, a test, or another chapter can cite the exact lesson without re-telling the story. A **V-era** (`V1-era`, `V2`, `V3.x`) is a *methodology version*: the framework's gates get stricter over time (a serially-aware bootstrap, deflation-for-`N`, portfolio-level promotion), so a result is tagged with the framework version that blessed it; a "V1-era" Sharpe predates rules a "V3" result had to clear, which is why the caveats chapter (see "Caveats Open Problems") treats un-retagged numbers as `unconfirmed`. The codes this book actually cites:

Code	Failure mode → the rule it bought	Enforcement today
A1 / A2	Same-bar look-ahead: lag any series before it multiplies a return	**CI gate** (value-corruption causality test)
A3	Total-return vs price-only confusion: make TR/price-only explicit in every cost & return signature, and write data hypotheses before believing a surprising RETIRE	prose + typed signatures
A4	"OOS" by partition, not provenance: per-fold selection or genuine pre-registration, *plus* a hard sanctuary / `CI_lo` veto in the decision function	API (decision-fn veto) + process
A5	Deflation at the *survivors'* count: DSR's `N` and trial-variance come from the full screener pool; survivors-only mode self-flags optimistic; family-wise / FDR across a program	prose + API flag
A6	The tail bootstrap resampled the *effect*: resample the *underlying's* returns and re-run the strategy on each path	prose / safe-by-construction
A7	A scaling overlay post-multiplied onto P&L: wire the overlay *into* the engine so it changes the trades, costs, and fills	prose

A8	"Validated against X" with no artifact: require a checked-in artifact *and* a reproducible test that regenerates it	prose (target: CI)
A9	Strategy guide $\neq$ deployed config: match parameter-for-parameter, verified	prose (target: CI)
A10	Parity tested against a *mirror*: compare the live signal against an *independent* reference, never the research code re-used as its own oracle	parity harness
A11	Parity not run to the integer position: parity across the *whole* chain (data $\rightarrow$ feature $\rightarrow$ signal $\rightarrow$ size $\rightarrow$ units), including native-leverage tier scaling	parity harness
V1-era / V2 / V3.x	A methodology *version*: gates get stricter over time, so an un-retagged number is `unconfirmed` until re-run	verdict tagging
V3.1 / V3.2	Pre-commit the selection rule; select the *plateau*, not the peak	process
V3.4	Prove which components carry the edge by ablation; judge ballast on `p_kill`, not a return metric	process
V3.5	Drawdown breakers are *failsafes*, not the primary control (the continuous size haircut is primary)	design

V3.6	Document every dead end so it isn't silently re-run without fresh pre-registration	process

The codes are stable: a newly-catalogued failure takes the next number and keeps it, so "A6" means the same thing in a backtest you can trust (see "Backtest You Can Trust"), the preflight checklist (see "Preflight Checklist"), and here. The **Enforcement** column is the same prose → safe-API → CI-gate strength ladder the end of this chapter defines: most lessons are still prose (memory), and maturing the stack means dragging each one rightward to "the build won't let you."

Note: How a bug becomes a catalogue entry
The codes don't assign themselves, so it's worth stating the intake path once (generically, because it transfers to any team keeping its own ledger). An entry is minted when an **independent path disagrees**: an internal re-derivation that won't reproduce a number, or an external adversarial audit that surfaces a failure the original author's tests missed (the recurring theme of this chapter, since you don't catalogue your own blind spot by re-reading the code that has it). The finding takes the **next free A-code** and keeps it; what gets recorded is the *lesson and its enforcement state*, not just the incident, so the row can later be dragged from `prose` toward `CI gate`. The discipline that matters is not the bureaucracy of *who* assigns the number; it's that the register is **append-only and single-source**: one place every chapter, test, and fix cites, so "A6" can never quietly come to mean two things.

Family 1. Look-ahead: the result is fiction

These are the bugs where the backtest measured something that could not have been traded. They are the most dangerous family because the equity curves they produce are not merely good, they are *plausible*: a same-bar leak on an autocorrelated signal manufactures exactly the smooth, persistent P&L a real edge would.

Danger: War-story: the signal that earned the bar it was looking at
A regime-rotation strategy chose its position at the close of bar t using a momentum rank computed from data including bar t, then collected bar t's return on that position. Decision and reward lived in the

same time window. Because momentum is serially correlated, the leak didn't add noise; it added a beautiful, fake trend. An audit found the identical pattern in **four separate places** in one codebase, each a signal series multiplied by a contemporaneous return, none caught in review because each read as routine pandas (`ret.where(cols == winner)`).

Consequence. The stitched out-of-sample inflation was strongly instrument-dependent, small and negative on a noisy series, materially positive on a trending one (illustrative: order of a few percent per year), enough to flip a flat strategy to a deployable one. **The rule it bought (A1/A2):** any series that multiplies a return must be `.shift(1)`'d first, unless you can *prove* the position was knowable strictly before the return's window opened. Treat same-bar `position * return` as guilty until proven innocent, and verify with a value-corruption causality test, not by eye; the shift discipline is worked end-to-end in A backtest you can trust (see "Backtest You Can Trust").

The subtler cousin of same-bar collect is the *bounded* leak. That same audit found the inflation depends on **position persistence**: a strategy that holds until its prediction flips has few transition bars, so the leak is small; a fresh-decision-every-bar strategy can collapse when fixed. The lesson is not "the leak is sometimes harmless" (fix it regardless) but *quantify it before you assume it dominates*, because the size of the bias tells you whether there was ever an edge underneath.

A second look-ahead shape hides in feature construction. A z-score over the *whole* series (`(x - x.mean()) / x.std()`) lets every historical bar "know" statistics from bars that hadn't happened yet; it never touches a return directly, which is why it survives review. The fix is causal or in-sample-frozen normalisation, and the deeper pattern is to make the dangerous version *absent from the API*: Titan's metrics module offers no full-series z-score, so it cannot be called by accident. Cross-timeframe aggregation is the same trap in a different coat: forward-filling a higher-timeframe signal onto past lower-timeframe bars once turned a true-zero-edge strategy into a phantom with a Sharpe near +2 (illustrative), all of which evaporated once the fill was made causal. The discipline: `.shift(1)` *then* reindex-with-ffill, wrapped in a causality assertion.

Look-ahead bug	What it looks like	The rule it bought
Same-bar collect (A1/A2)	`position * return` with no lag	Lag any series before it multiplies a return; value-corruption causality test in CI

Look-ahead bug	What it looks like	The rule it bought
Future-normalised feature	`(x - x.mean()) / x.std()` over all time	Only causal / IS-frozen normalisation exists in the API
Cross-TF ffill	higher-TF signal ffilled onto past bars	`.shift(1)` then reindex; wrap in `assert_causal`

Warning: War-story: when 'is it a data bug?' is the right first question
A cross-sectional momentum audit came back uniformly negative, contradicting a published edge. Before blaming the strategy, we asked the cheaper question: *is the data wrong?* It was: the download kept price-only `close` and dropped total-return `adj_close`, systematically under-ranking dividend payers. We re-downloaded with total returns and re-ran; the result got *more* negative, not less. **The rule it bought (A3 + the falsification discipline):** total-return vs price-only must be explicit in every cost-model and return signature, and for any surprising RETIRE you write two or three data-construction hypotheses and test the cheapest first. Here the fix confirmed the verdict, but had it flipped the sign, we'd have caught a false retirement. The fifteen "wasted" minutes are the insurance that makes a negative result trustworthy.

Family 2. Validation honesty: the result is luck

The look-ahead family is about whether the number is *real*. This family is about whether a real number is *durable*: whether it survives being one of many tries, a thin sample, or a single lucky regime.

The headline failure is the **co-committed pre-registration**. A scripted git scan of one program found that almost every pre-registration file was first committed *in the same commit as its own verdict*: no timestamp evidence that the canonical cell, the fold count, or the thresholds were fixed before the run. That doesn't make the verdicts wrong; it makes the entire deflation defence (multiple-testing correction, cross-selection penalty) **unverifiable**, in both the RETIRE and DEPLOY directions. A pre-registration you can edit after seeing the result is a diary, not a contract.

Warning: War-story: the deflation that deflated nothing
A *deploying* audit fed its Deflated-Sharpe calculation a hand-picked four-cell "plateau" as both the trial count `N` and the variance-across-trials. The full sweep had been far larger; the deflation was computed as if only four hypotheses had ever been tried, and the optimism went **unflagged** because the survivors-only mode defaulted to silent. The strategy carried real weight in the live book.
Consequence. Too-weak deflation on a strategy certifying capital. **The rule it bought (A5):** for any sweep with more than a handful of cells, the DSR's `N` and trial-variance come from the *full screener pool*, not the survivors; survivors-only mode must self-flag as optimistic; and across a research program you need a family-wise / FDR control, because fifty independent audits each tested against a fresh 0.95 gate will eventually deploy a luck-survivor.

Two quieter failures sit underneath. First, the **lower bound that lied**: an IID bootstrap resamples bars independently, destroying the serial correlation that trend and carry strategies live on, which narrows the interval and biases the *lower bound upward*, the exact number you gate on. A stationary block bootstrap fixes it. Second, **the sanctuary that couldn't rescue, and the matrix that let it try**: holding out the most recent twelve months is mandatory because that year is often anomalously strong, but a positive sanctuary can never lift a strategy whose walk-forward lower bound is negative. A *count-of-best-axes* decision matrix has no hard veto: a catastrophic hold-out Sharpe of $-$0.5 scored identically to a mediocre axis, so a strategy that *lost money on the hold-out year* still earned a non-RETIRE tier. The rule (A4 + the veto): out-of-sample requires per-fold selection or genuine pre-registration, and any "worst" on the sanctuary or CI-lower-bound axis caps the verdict regardless of the other axes.

Note: Negative results are output, not waste
A failed audit is data. One research cycle produced dozens of nulls: IC censuses with zero survivors, strategies retired at the plateau gate, confluence tests where gating *destroyed* the signal, each logged with its failure *mechanism* and the decision rule applied. **The rule it bought (V3.6):** document the dead end so the next researcher (or you, in six months) doesn't re-run it without fresh pre-registration. A catalogue of what *doesn't* work is as load-bearing as the list of what does. Whole families of pre-2014 published edges now get framed as *falsification tests*, not replication targets, precisely because the catalogue made the decay pattern visible.

Family 3. Tail and cost realism: the result is fragile

A Sharpe can be real, durable, and still describe a strategy you cannot survive. This family is about the numbers that decide *livability*: drawdown geometry, tail loss, and the costs that quietly eat the edge.

A Sharpe past every statistical gate still hid a double-digit, year-plus drawdown no committee would hold: it bought **Calmar lift (not Sharpe lift) as the primary promotion metric**; full telling in the metric suite (see "Metric Suite") and tail risk & ruin (see "Tail Risk And Ruin").

The cost side is its own graveyard. A pure `bps_per_turnover` model underprices reality the moment notional is small, the broker charges a per-fill commission floor, or a vol-target overlay produces many tiny daily rebalances. A live cost audit on a real paper account found the *true* drag was several times the modelled drag (illustrative: a low-double-digit bps/yr estimate that grew to roughly five times that once the gaps were closed): the model had missed the commission floor, mis-calibrated the ETF leg, and counted sub-threshold rebalances the live class would have skipped. It didn't flip the verdict, but it turned a comfortable margin into a thin one. Continuous futures legs are worse: an always-on position pays nothing in most models for mandatory quarterly rolls (on the order of tens of bps/yr per instrument), and research exits at the clean close while live stops gap through it.

A tail gate resampled the strategy's own returns instead of the underlyings, re-confirming the realised path; **(A6) Monte Carlo perturbs inputs, not outputs**, and long-only sleeves need a relative (vs buy-and-hold) MaxDD gate; full mechanism in tail risk & risk of ruin (see "Tail Risk And Ruin").

A close relative of the MC bug is the **overlay applied to the wrong layer**. It is tempting to model a position-scaling overlay (vol-targeting, a Kelly haircut, a regime gate) by computing the base strategy's returns and then *post-multiplying* by a scale series. That is wrong whenever scaling changes turnover, costs, or the interaction with stops, which it almost always does. A $0.5\times$ scale isn't half the return; it's a different trade with different fills.

> **Tip: Position scaling changes the engine, not the output**
> **The rule (A7):** a scaling overlay must be wired *into* the backtest engine so it changes the positions the engine actually trades (and therefore the costs, the fills, and the drawdown path), not bolted on as a multiplier after the P&L is computed. The same discipline catches a related class of fragility: bare-threshold regime gates (`signal >= K`) flip on noise near the boundary and fail an input-noise robustness test. Prefer percentile gates, ensembles, or continuous (sigmoid) scaling, and remember that input-noise robustness and parameter-plateau robust-

ness are *different* tests; a strategy can pass one and fail the other.

Family 4. Deployment parity: the result isn't what runs

The most expensive gap in a quant stack is not in the research; it is between research and the live code. You can pass every gate in Part II and still lose money because the thing that trades is not the thing you validated.

Danger: War-story: the strategy guide that didn't match the deployed config
A live strategy's user guide documented one set of parameters; the deployed configuration ran another, unreconciled after a tuning pass. The guide, the document an operator reaches for at 2 a.m. during an incident, described a system that wasn't running. **The rule it bought (A9):** the strategy guide must match the deployed config *parameter-for-parameter*, verified, not trusted. A guide that disagrees with the live `.toml` is worse than no guide; it actively misleads the person trying to fix a live position.

The deeper parity problem is that "validated" often means "validated *something*." An external audit found a live equity sleeve built on best-of-N *current* index constituents, the exact survivorship-plus-selection construction a previous strategy had been *retired* for, in production with docstring Sharpes the audit itself called "implausibly high." The validation existed; it just didn't validate the thing that shipped.

Warning: War-story: 'validated against X' with no artifact
A claim that a strategy had been "validated against" a reference appeared in a docstring with no checked-in artifact and no reproducible test behind it. Nothing to re-run, nothing to diff, nothing to fail in CI. The claim was load-bearing for a deployment decision and unfalsifiable.
Consequence. A deployment justified by a sentence rather than a test.
The rule it bought (A8): "validated against X" requires a checked-in artifact plus a reproducible test that regenerates it. If you cannot point to a file and a command that re-derives the comparison, the validation does not exist. Same instinct as the frozen-ML-artefact rule: a model file is a *build artefact* of its feature pipeline: embed the feature names, assert compatibility on load, and put a one-row prediction in CI, or the

only warning you get is a crash on the first live bar.

The parity test itself has a precise shape, and getting it wrong gives false comfort. It is not enough to check that the live signal *exists*; you must check that the live `on_bar` signal at t equals the vectorised research signal at t, computed by an **independent reference** down the **full chain** (data load, feature build, signal, sizing), with a **causality test** confirming the live path can't see the future either. A parity test that re-uses the research code as its own reference proves only that the code equals itself.

Danger: War-story: the leveraged instrument sized as if it weren't
A live strategy class sized its position in instrument units without accounting for an instrument that carried **native leverage**: the contract already embedded a multiplier the research-side sizing had folded in differently. The live class needed to *scale by tier* to match the validated exposure; it didn't, so the live position didn't match the size the research had risk-checked.
Consequence. Live exposure diverging from validated exposure on a leveraged instrument, the most dangerous place for a sizing error, because leverage scales the loss and the ruin probability by the same factor. **The rule it bought (A10/A11):** parity tests run end-to-end through the full chain with an independent reference and an explicit causality test, and the live class must scale by tier whenever the instrument has native leverage. Sizing is not a detail you eyeball; it is a contract the parity test enforces.

Danger: War-story: the two safeguards that agreed with each other and were both wrong
To stop the live system from running the wrong strategy set, two structural checks were added: a pre-flight gate and a monitoring dashboard, each carrying its own copy of "what the live bundle contains." A later audit found that copy had drifted from the authoritative registry: both the gate and the dashboard listed strategies that the runner had retired months earlier. Crucially, the two checks were **mutually consistent** (they validated against each other), so both reported green while describing a portfolio that wasn't running. A safeguard that checks one belief against a second copy of the same belief proves only that you wrote it down twice.
Consequence. The operator-facing tooling (and an audit reading it) asserted strategies were trading that weren't, and could not tell which were live. **The rule it bought:** every derived view of the deployment (pre-flight gate, dashboard, docs) must validate against the *single au-*

thoritative source (the registry the runner actually reads), enforced by one test that fails CI when any of them diverge. Two safeguards that agree with each other are one safeguard with redundancy theatre. The same audit found a pre-flight check parsing the *wrong schema* for the halt file, so an active kill-switch halt passed the check as "no halt": your safety checks are code, and untested safety-check code fails exactly when you need it.

Danger: War-story: the dormant strategy whose first live act was a crash
A strategy sat in the live bundle but never traded: a warm-up gate it could not satisfy (no historical backfill) kept it short of the bar count its indicators needed, so for roughly a year its `on_bar` returned early every day. It looked deployed. When the gate was finally fixed and the strategy reached its trading path for the first time, it crashed on the first bar: a method called on a framework type that didn't carry it, a latent `AttributeError` that *every* live signal would have hit, sitting undisturbed because no live signal had ever arrived. The validation suite was green throughout; it tested the signal math, not the order path the strategy had never executed.
Consequence. A strategy believed deployed-and-validated whose entire trading path was unproven, primed to fail on activation. **The rule it bought:** *code that never runs is not validated*; a strategy that cannot reach its order path is in an unknown state, not a safe one. Treat activation (or un-dormanting) as a deployment event with its own gate, and unit-test the order path directly rather than inferring it from the signal test. A green dashboard is not the same as an exercised code path.

Danger: War-story: the close that closed nothing
After a restart, the trading framework reconciled the broker's open positions back into its cache and tagged each one with an *internal* owner id (a reconciliation placeholder), not the strategy that had opened it. The strategy's exit, halt-flatten, and shutdown paths all called a strategy-scoped "close my positions" helper, which filters by the strategy's own id, found nothing under it, and logged a cheerful "no positions to close" while the inventory rode the market unmanaged. The position was visible everywhere; it just wasn't *owned* by the code trying to close it. The strategy then saw itself as flat-but-still-long and could neither exit nor re-enter.
Consequence. On every restart with an open position, the kill-switch

flatten and the signal exit became silent no-ops: the worst failure mode for a control you are counting on. **The rule it bought:** claim ownership of reconciled positions explicitly (the framework offered an unused `external_order_claims` hook that re-tags them to the right strategy on reconciliation), and assert post-restart that your flatten path actually reaches the broker's net position. A close path that filters by owner must be tested against the way the *framework* labels positions after a restart, not the way your code labelled them before one.

Danger: War-story: the guard that reset on every restart
A daily momentum sleeve held each position for a minimum number of bars before it would consider an exit: a hold guard that lived as an in-memory bar counter. On restart the framework correctly re-adopted the open position (the fix from the previous war-story), but seeded that counter to *already past the minimum*, on the reasoning that a re-adopted position of unknown age should at least be exitable. The reasoning was local and the consequence was global: the system restarted constantly (a watchdog on every wedged data feed, a redeploy on every change), so **each restart reset the hold clock to "satisfied."** A position entered four bars ago, against a five-bar minimum, became exit-eligible the instant a restart intervened, and the next sub-threshold bar closed it a day early. The guard read as armed in every code review and was a no-op in production, defeated not by a logic error but by the operational reality of uptime.
The deeper version of the same bug lived in a regime-tiered strategy's **drawdown circuit breaker**. Its state (throttled-or-killed, the consecutive-quiet-bars recovery counter, and the high-water mark the drawdown was measured against) was all in memory, and a fresh process reset it to normal, recovery zero, high-water-mark back to seed capital. A restart while the breaker was throttling exposure in a real drawdown therefore *cleared the breaker and blinded it to the loss at once*: equity reset to par, drawdown read as zero, and the strategy re-levered straight back into the decline the breaker existed to escape.
Consequence. Two risk controls (a minimum-hold and a drawdown breaker) that looked live and were silently disarmed by the one event that happens most often: a restart. **The rule it bought:** any guard whose decision depends on elapsed time or a path-dependent counter (minimum-hold, cooldown, re-entry-quiet, circuit-breaker dwell, the high-water mark a breaker measures against) must be **persisted and restored across restarts**, reconstructed from a durable store rather than re-seeded to a default. An in-memory guard in a system that restarts frequently is not a guard; it is a guard-shaped object that works

only as long as the process does. Persist the true state on each bar, restore it on reconciliation, and fall back to the conservative default only when there is genuinely nothing to restore. Concretely, on each bar write the guard's state (entry bar, bars held, high-water mark) to a durable store, and read it back during reconciliation before the strategy acts. The persisted halt file and durable hold-state that implement this pattern in Titan live in Layered safety (see "Layered Safety").

For the full treatment, the contract a strategy class must satisfy and the replay harness that certifies the live path against the proven-causal research path, see Live equals research (see "Live Equals Research") and The strategy-class contract (see "Strategy Class Contract").

Family 5. Selection discipline: the result was cherry-picked

The final family is about the choices you make *around* the backtest: which parameter, which instrument, which control is doing the real work. These are bugs of an honest researcher's optimism rather than a coding mistake, which makes them the hardest to see.

A canonical cell posted a strong Sharpe while a one-step neighbour in the same grid dropped by half: a lucky coordinate, not an edge. **The rules it bought (V3.1 + V3.2): pre-commit the selection rule and select the plateau, not the peak**, run cheaply before any Monte Carlo. This is the qualitative twin of Beating your own optimiser (see "Deflated Sharpe"), the quantitative form of the same concern.

The single-instrument version is the **unrecorded search**. When a legacy config names one instrument from a class of plausible candidates (trend on this ticker, carry on that pair), the chosen instrument is almost certainly the survivor of a search nobody wrote down, and its in-sample Sharpe overstates the true cross-sectional edge by an unknown order statistic. The rule: always run a multi-instrument robustness panel, because a single named instrument is a confession of selection bias until proven otherwise.

A V3.4 ablation (turning each component off one at a time) caught a "ballast" piece, a defensive switch assumed to be along for the ride, actually carrying the edge: **prove which components matter with an ablation rather than assuming, and because its value is in the left tail, judge ballast on `p_kill_trip`, not a return metric** (see tail risk & risk of ruin (see "Tail Risk And Ruin")). The same lens reframed our drawdown breakers: they are *failsafes, not primary controls* (V3.5). A breaker stacked on a vol-targeted

strategy fires rarely and occasionally clips a recovery: useful as a last line, harmful as the main risk system. The primary control is the continuous size haircut; the breaker is the seatbelt, not the steering.

Together this family is the difference between *we chose this* and *this is what survived our suspicion*: a pre-committed selection rule (V3.1), a plateau over a peak (V3.2), a recorded instrument search, an ablation to find the load-bearing parts (V3.4), failsafes kept in their place (V3.5), and every dead end documented (V3.6).

The catalogue at a glance

The chapter is a set of stories; a catalogue should also be *scannable*. Here is every war-story above in one table: symptom, family, the rule it bought, and how strongly that rule is enforced today (the prose → API → CI ladder the next section formalises). IDs reuse the A-code / V-era from the register at the top of this chapter where one exists, and take a local `C`-tag otherwise. Scan the **Enforcement** column for the work that remains: every `prose` row is a lesson still waiting to become a gate.

ID	War-story (symptom)	Family	Rule it bought	Enforcement
A1/A2	signal earned the bar it was looking at: a smooth, *fake* trend	1 Look-ahead	lag any series before it multiplies a return	**CI gate**
A3	uniform-negative IC was a price-only **data** bug, not a dead edge	1 Look-ahead	TR vs price-only explicit; test the cheap data hypothesis first	prose + signatures
Z1	full-series z-score let every bar see the future	1 Look-ahead	only causal / IS-frozen normalisation exists in the API	safe-by-construction

ID	War-story (symptom)	Family	Rule it bought	Enforcement
A5	DSR fed a hand-picked 4-cell plateau, not the full sweep	2 Honesty	`N` and trial-variance from the full pool; survivors-only self-flags	prose + API flag
A4	a `−0.5` sanctuary year still earned a non-RETIRE tier	2 Honesty	per-fold OOS + a hard sanctuary / `CI_lo` veto	API veto + process
V3.6	dozens of nulls nearly re-run for want of a record	2 Honesty	document every dead end	process
A7	a scaling overlay post-multiplied onto the P&L	3 Fragility	wire the overlay into the engine so it changes the trades	prose
A6	tail gate resampled the strategy's own realised returns	3 Fragility	resample the *underlying* and re-run the strategy	prose
A9	strategy guide described a config that wasn't running	4 Parity	guide matches deployed config, verified	prose (→CI)
A8	"validated against X" with no artifact behind it	4 Parity	artifact + reproducible test, or the validation doesn't exist	prose (→CI)
A10/A11	a leveraged instrument sized as if it weren't	4 Parity	parity end-to-end to integer units; tier-scale native leverage	parity harness

ID	War-story (symptom)	Family	Rule it bought	Enforcement
C1	two safeguards validated each other, both months stale	4 Parity	every derived view checks the *authoritative* source; one CI test	prose (→CI)
C2	a dormant strategy crashed on its first-ever live bar	4 Parity	code that never runs is unvalidated; gate activation; test the order path	prose
C3	strategy-scoped flatten no-oped on reconciled positions	4 Parity	claim ownership of reconciled positions; assert flatten reaches the broker	prose
C4	in-memory min-hold and breaker reset on every restart	4 Parity	persist & restore any time- or path-dependent guard state	prose
V3.1/V3.2	a canonical cell beat its one-step grid neighbour	5 Selection	pre-commit the rule; pick the plateau, not the peak	process
C5	a single named instrument = an unrecorded search	5 Selection	always run a multi-instrument robustness panel	process
V3.4	a "ballast" component was secretly carrying the edge	5 Selection	ablate to find the load-bearing parts; judge ballast on `p_kill`	process

The shape of the work is the right-hand column: four `CI gate` / `API` rows are *closed* (the bug cannot recur), and a dozen `prose` rows are *open* (known, named, but still relying on someone remembering). That gradient is the

chapter's real thesis, made literal in the next section.

How the catalogue becomes a gate

A war-story you only tell at the bar changes nothing. The point of cataloguing each failure is to convert it into something that can't recur silently. The conversions take three forms, in order of strength:

Form	Strength	Example
A prose lesson with a mechanism	weak: relies on memory	"remember the carry premium is `yield × time-in-market / vol`, not `yield / vol`"
A safe-by-construction API	strong: the wrong call doesn't exist	no full-series z-score; `periods_per_year` has no default
An automated CI gate	strongest: the bug fails the build	AST check rejecting `sqrt(252)`; value-corruption causality test

The honest assessment, delivered by an external auditor, is that most lessons in a young program are still *prose*, the weakest form. And that claim is itself backed by a number, not a hand-wave: of the war-stories in the catalogue table above, four are closed by a safe-by-construction API or a CI gate (the bug cannot recur) and roughly a dozen still live only as prose or process. The work of maturing a stack is dragging each catalogued failure up that table: from "we know not to do that" to "you literally cannot do that." Every bug here that became an absent API call or a CI gate cannot reappear; every one still living as a paragraph in a directive is waiting to be rediscovered by whoever didn't read it.

Warning: The discipline is not free, and you cannot build it all at once
Cataloguing every failure as a gate makes the framework strict, and strict gates have a cost the war-stories above understate. Some of them will veto a marginal-but-real edge: a full-screener-pool DSR `N` deflates an honest result alongside the lucky ones, a hard relative-MaxDD veto can bin a strategy whose tail is ugly but survivable, and the sanctuary cap will RETIRE something that genuinely earned its keep in every other year. That asymmetry is deliberate (a false retirement costs opportunity, a false deployment costs capital), but pretending the gates only ever catch fakes is its own optimism. And the persistence ma-

chinery from the parity family is real engineering work, not a one-liner: every path-dependent guard needs a durable store, a restore-on-reconciliation path, and a test that the restore actually fires. A solo builder cannot stand all of this up at once, so adopt it in order of leverage: **shift/causality and parity first** (the bugs that make the number outright fiction), **then deflation and the tail gates** (the ones that decide whether a real number is durable and survivable), **then the persistence and ownership machinery** (the ops failures that only bite once you are live). Each layer earns the next.

Exercises

1. **Why a catalogue, not a checklist.** Why does the book keep a catalogue of failure *families* rather than a tidy list of best practices?

   Answer: Answer
   Best practices are advice; a catalogued failure mode is a gate. The same bug *class* recurs forever if you only fix instances: a researcher who never saw the same-bar leak writes it again next quarter in different pandas. And note who catches them: mostly not the author, because you don't find your own look-ahead by re-reading the code that contains it.

2. **The bounded leak.** A same-bar leak inflated one strategy a lot and another barely. What explains the difference, and what's the lesson?

   Answer: Answer
   Position persistence: a hold-until-flip strategy has few transition bars, so the leak is small; a fresh-decision-every-bar strategy collapses when fixed. Lesson: fix the leak regardless, but *quantify* the bias; its size tells you whether there was ever an edge underneath.

Takeaways

- **Bugs cluster into five families:** look-ahead (fiction), validation honesty (luck), tail/cost realism (fragile), deployment parity (not what runs), selection discipline (cherry-picked). Diagnose by family, not by symptom.

- **Every error here flattered the strategy.** That asymmetry is why suspicion beats celebration: optimistic versions are the ones that survive your attention and reach production.
- **You don't catch your own leak by re-reading your code**; you catch it with an independent path that disagrees. Internal re-derivations and external adversarial audits found most of these, not the authors.
- **Parity is where research meets capital.** A guide that doesn't match the config, a "validation" with no artifact, a leveraged instrument sized as if it weren't: these lose money even when the research was perfect.
- **Your safeguards are code too.** A pre-flight gate that validates against a second copy of the belief it's checking (instead of the authoritative source), a halt-check that parses the wrong schema, a backstop that exists in the runbook but not the deployment: each fails green exactly when relied upon. Test the safety layer; tie every derived view to one source of truth.
- **Code that never runs is not validated.** A dormant strategy, a never-exercised order path, a close helper never tested against post-restart position ownership: "green dashboard" is not "exercised path." Activation is a deployment event.
- **A guard that lives only in memory dies on the restart.** Minimum-holds, cooldowns, and drawdown-breaker state must be persisted and restored, or the most frequent event in your system silently disarms them. Reconstruct the true elapsed state from a durable store; never re-seed it to a default.
- **A catalogued failure is only as good as its enforcement.** Prose relies on memory; a safe-by-construction API or a CI gate cannot be skipped. Drag every lesson up that table.

The next part turns the strongest of these rules into machinery: The strategy-class contract (see "Strategy Class Contract") defines what a strategy must satisfy to deploy at all, and Live equals research (see "Live Equals Research") builds the parity harness that makes the deployment-parity family impossible to ship. For the statistical machinery behind Families 2 and 3, see Walk-forward that's actually out-of-sample (see "Walk Forward"), Beating your own optimiser (see "Deflated Sharpe"), and Tail risk & risk of ruin (see "Tail Risk And Ruin").