



Trading Risk Verifier

Proof, not testing, that your risk controls and trading logic hold — on every market move, every fill, every order.

Backtests and simulations check the runs someone thought of. The Giolit Trading Risk Verifier checks **all of them**. It reads your strategy — from its C++, Python or Java code, or from a short specification — with its risk controls and the exchange protocol, and answers, for every rule you state:

PROVEN

The rule holds on every possible run: any market the assumptions allow, any fills, any timing of messages, any number of orders. You get a signed certificate with the proof.

VIOLATED

A concrete, replayable run that breaks the rule — event by event, as a FIX log your team can load and debug. Typically 3 to 15 events long.

0

wrong answers on the 180 trading rules of the release benchmark, each with a known answer

5

open-source strategies verified from their unmodified code; issues found in 4 of them

100 000

simultaneous orders: a position or exposure limit proved in under half a second

100%

in-house engines: no third-party solver, runs fully on your premises

Why trading systems still fail

The costly failures are the runs nobody tested.

A risk check that looks only at the position lets two orders through before either fills. A kill switch that cancels on one venue leaves the other venue's orders working. A hedge that is still pending when the next clip starts. A leg that the exchange rejects after the other leg has filled. A fill that arrives after the cancel was acknowledged.

Each of these needs a particular *interleaving* of market data, acknowledgements, fills and timers. Testing and backtesting replay the sequences someone thought of or the market happened to produce; the dangerous ones are rare by nature, and they are found in production.

The Giolit Trading Risk Verifier explores **every** sequence the exchange protocol and your assumptions allow. It does not sample: when it says PROVEN, there is no run — however unlikely — that breaks the rule. When there is one, it shows you the shortest it can find.

✳ An example: the hedger

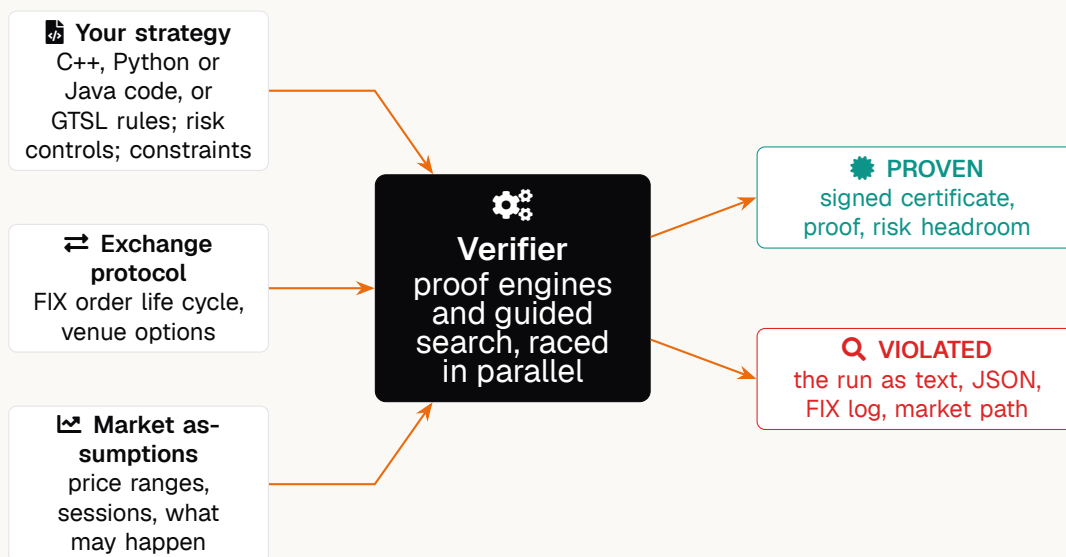
A strategy buys INFY in clips and hedges each fill by selling TCS. Its rule: *never more than one clip unhedged*.

```
1 MarketUpdate INFY
  -> NewOrder 01 BUY 100 INFY
2 Ack 01
3 Fill 01 100
  -> NewOrder 02 SELL 100 TCS
4 MarketUpdate INFY
  -> NewOrder 03 BUY 100 INFY
5 Ack 03
6 PartialFill 03 3
  unhedged = 103 > 100
```

The second clip starts while the first hedge is still pending. The fix — wait for the hedges — is then **proved** for any number of orders in under a second.

How it works

Three inputs, two kinds of answer, evidence for every answer.



Constraints — and, without code, the rules — are written in **GTSL**, a compact language that reads like a risk policy: `reject when position + open_qty(buy) + order.qty > max_long; always position <= max_long; after kill_switch: never new_order;.` Prices, quantities, money and time carry units, so a mix-up between ticks and rupees is an error, not a silent bug.

Features, and what each one does for you

Every feature exists to answer a question a trading desk, a risk team or an auditor actually asks.

Risk controls for any strategy

Certify the pre-trade controls alone: the verifier lets an arbitrary strategy send any order or cancel at any moment.

✓ **Why it matters:** Brokers can vet a client's algorithm without seeing its code or its alpha; a proof holds for every strategy the gateway will ever face.

Several venues, concurrency

Per-venue books, orders and limits; messages from different venues interleave in any order; risk checks on their own thread (`risk_controls.async`).

✓ **Why it matters:** Smart order routers and multi-venue strategies fail on interleavings that single-venue tests never produce.

Signals and lags

A model's output (a z-score, a fair value) is a *signal*: any value of its range at every update. `history(x, k)` reads a value `k` events ago.

✓ **Why it matters:** Verify the trading logic around a model without verifying — or disclosing — the model itself.

Risk headroom and resilience score

For every proven limit, the worst value it ever reaches, how far the limit could be tightened, and a 1–5 star Giolit Risk Resilience Score.

✓ **Why it matters:** Know whether a limit is barely holding or holds with room to spare — and set limits from evidence.

Conformance replay of FIX logs

Replay a production drop copy or execution log through the specification: every message that cannot happen, every order the rules would not send, every rule the run breaks.

✓ **Why it matters:** Closes the gap between the model and production; catches drift after a release and venue behaviour you did not expect.

Verify the code you deploy

Import the strategy or the risk layer from its C++, Python or Java source: a short binding names the handlers and the order calls; the code itself is read, never run, and never approximated silently.

✓ **Why it matters:** No restating of logic by hand: the proof or the failing run is about the code that trades. 5 open-source strategies (WonderTrader, Hummingbot) are in the benchmark.

Several symbols, multi-leg orders

Pairs, hedges, spreads and index arbitrage: per-symbol positions, net, gross and beta-weighted exposure, orders of several legs that pass the controls together.

✓ **Why it matters:** Legging risk — one leg filled, the other rejected — is found as an exact run; hedge-ratio and exposure limits are proved.

Real arithmetic, exactly as in C++

Strategies written with `double` keep their formulas: IEEE binary64, rounding included, with `exp`, `log`, `sqrt`.

✓ **Why it matters:** Market makers such as Avellaneda–Stoikov are verified as coded; a run that breaks a rule shows the exact floating-point values.

Certificates and evidence bundles

A signed PDF certificate (or violation report), the proofs, every run as text, JSON, FIX messages and a market path, and an offline verification script.

✓ **Why it matters:** Ready-made evidence for algorithm approval, internal model validation and annual self-assessments.

FIXatdl and FIX Orchestra imports

Take client parameters from FIXatdl (verify every value a client can enter) and a venue's order state machine from FIX Orchestra.

✓ **Why it matters:** Verification starts from the documents you already maintain, and covers every parameter combination clients can choose.



Profit and loss guarantees

What can honestly be guaranteed: the edge a market maker locks in and the worst-case loss are proved; “never under water” is shown false with a run.

✔ **Why it matters:** Separates provable P&L properties from wishful ones before capital is at risk.



Feasibility and optimization

Targets (“the parent order can complete by the close”) are reached or proved impossible; an objective such as implementation shortfall gets its best and worst value with proved bounds.

✔ **Why it matters:** Finds schedules that can never complete (an off-by-one in a VWAP schedule is one of the shipped examples) and bounds execution cost.

Start from a verified template

A library of strategy families, each with the constraints it must never break.

Template	Standard constraints, proved as shipped
Market maker	inventory limits, quotes never cross the market, quote size, limit orders only, kill switch
TWAP • VWAP • POV	parent never overfilled, on schedule / on the volume curve, participation cap, child size, one side only
Pairs	legging imbalance at most one pair, leg caps, gross exposure checked, beta-weighted exposure
Lead-lag	trades the follower only, position limits, one order at a time
Delta hedge	every hedge reduces the net delta, never overshoots, hedge limits
Smart order router	no overfill across venues, per-venue exposure, every order cancelling after the kill switch
Pre-trade risk gateway	order size, order value, price band, no market orders, position limits, kill switch, accounting

`trade-verify --template=pairs > my_pairs.gts` — adapt the parameters and rules, keep the standard constraints, add your own. A change that breaks one comes back as a run, before it reaches production.

Fits the way you work

A command, a library, a release gate.

- › **Command line:** `trade-verify spec.gts` — results in seconds to minutes; exit codes for scripts.
- › **C++ library and SDK:** verify, compute headroom, replay logs and issue certificates from your own tools.
- › **CI release gate:** run on every release of the strategy or the risk layer; a violated rule fails the build with the run attached.
- › **JSON, FIX and CSV outputs:** runs load into your FIX tools, your backtester (market path) and your dashboards.
- › **On premises, offline:** all engines are Giolit's own; no specification, code or data leaves your machines.
- › **Licensing:** per-licensee signed licenses; a trial of 1000 runs over 50 days.
- › **Benchmark included:** 44 specifications with 180 rules and their results ship with the trial — rerun any of them yourself.

Supporting your obligations

Evidence that maps onto what regulators and auditors ask for.

Requirement	What the verifier provides
Pre-trade risk controls (e.g. SEC Rule 15c3-5; SEBI algorithmic trading framework; MiFID II RTS 6)	proofs that order size, value, price-band, position and kill-switch controls hold for any strategy, with signed certificates
Algorithm testing and approval	every rule of the strategy checked against every interleaving the protocol allows; violations as replayable FIX logs
Change management	the same specification re-verified on every release; conformance replay of production logs after go-live
Annual self-assessment and audit	an evidence bundle per version: certificate, proofs, headroom, runs, and an offline integrity check

The verifier supplies evidence; it does not by itself make a firm compliant with any regulation.

See your own strategy verified.

Send us a risk policy or a strategy outline: we write the specification with you and show you the first proof — or the first run that breaks it.

 www.giolit.com

 business.giolitlabs@gmail.com

 Giolit Labs, Thrissur,
Kerala, India