What a Backtest Costs

Throughput, Memory, Scaling and Reproducibility of Four Backtesting Engines on a 64-Core CPU

Chaitanya Palghadmal*

Founder and Principal Systems Architect, Reamer Labs

Preprint • Version 1.1 • 2026-10-26

Abstract

We measure the wall time and memory of one backtest in Backtrader, NautilusTrader, QuantConnect LEAN and Reamer Research. We also measure how both grow with bar count (50,000 and 500,000 bars), ticker breadth (10 to 100 tickers) and parallel copies (1 to 64). Every engine runs the same two strategies on the same data, on one AMD EPYC 9575F. Versions are pinned and the raw results are public. The author builds and sells Reamer Research. Before seeing any result, we stated four outcomes that would count against it.

Reamer Research was fastest in every configuration. On 500,000 bars of the realistic strategy, its C++ path processed 1.39 million bars per second and its Python path 148,000, against 6,500 to 37,000 for the other engines. At 100 tickers its Python path finished in 14.9 s, against 94.8 s for the next fastest engine. With 64 copies in parallel, the C++ path completed 332,000 backtests per hour, about 30 times the next engine's 11,200. The C++ path also used the least memory of any engine in every configuration it ran. Every engine reproduced its results exactly across repetitions. No Reamer Research run failed, while LEAN hung in 14 of 22 multi-ticker attempts.

Two of the four adverse outcomes occurred. Above about 270,000 bars of the minimal strategy, Backtrader used less memory than Reamer Research's Python path: 209 MB against 309 MB at 500,000 bars. The C++ path had the lowest parallel efficiency at 64 copies, 0.53 against 0.63–0.72, although its throughput kept rising. We also report a quadratic cost in NautilusTrader 1.231.0's PnL recording, a multi-instrument cost in its position snapshots, and a start-up hang of LEAN under concurrency.

Keywords: backtesting; backtesting engines; benchmark; throughput; memory; parallel scaling; reproducibility; Backtrader; NautilusTrader; LEAN.

Classification: JEL C63 (Computational Techniques), C88 (Other Computer Software), G17 (Financial Forecasting and Simulation); ACM CCS Software and its engineering → Software performance; General and reference → Cross-computing tools and techniques → Performance.

Statement of interest

The author builds and sells Reamer Research, one of the four engines measured. The study is designed so that a reader need not trust the author:

- The other engines run at their defaults. The one patch to an engine is disclosed, and both the stock and the patched results are reported (§3.8).

*Corresponding author: chaitanya.p@reamerlabs.com. ORCID: 0000-0003-0173-0067.

- The runners, the dataset builders and the raw results are public [14].
- Four outcomes that would count against Reamer Research were stated before the results were seen (§3.12). Two occurred, and both are in the abstract.
- Where another engine does more, the paper says so (§9).

1. Introduction

Why the cost of one backtest matters. A researcher tests an idea, reads the result, and changes the idea. The limit on that loop is turnaround: how many backtests one machine completes per hour, and how much memory each one needs. A single headline speed does not answer that. The answer depends on how the cost of a backtest grows with the length of the data, with the number of instruments, and with the number of copies that run at once.

What published comparisons measure. We found no published comparison of backtesting engines that measures all three directions under aligned fill rules. The comparisons we found are blog posts:

- A 2026 post compares Backtrader and Zipline on 5 to 20 years of daily SPY bars, with speed and peak memory, on a laptop [1]. It reports single runs. It does not align fill timing or commission models across the two engines, and it does not test parallel runs.
- A 2026 survey of five Python engines, NautilusTrader among them, compares design, order modelling and maintenance status [2]. It gives speed only qualitatively and runs no benchmark.

These are useful starting points, and we do not criticise them for being what they are. But a reader cannot use them to say what one backtest costs on a given machine, or why one engine's figure differs from another's.

Contributions.

1. **A like-for-like harness.** The same two strategies and the same data run in all four engines. Every deviation from an engine's defaults is listed. Fill-timing differences are kept and explained, and net PnL is checked within each fill-timing group (§3.9, §4.4).
2. **Scaling in three directions:** bar count (50K to 500K bars, and to 5M for Reamer Research alone), ticker breadth (10 to 100 tickers) and parallel copies (1 to 64 on 64 physical cores).
3. **A reproducibility test.** The full closed-trade ledger is hashed over 20 runs, and every engine's repeat agreement is reported (§8).
4. **A dated, sourced comparison** of what a benchmark does not capture: order model, customisation, path to live trading, maintenance, licence and cost (§9).
5. **Two findings for NautilusTrader users** (§4.3, §6) and one for LEAN users (§3.11), each with its measured cost.

Scope. This paper measures bar-based research backtests. It does not measure tick or order-book simulation, where NautilusTrader and LEAN offer much more than Reamer Research does. It does not measure live trading latency. A companion paper covers the many-core operating envelope of Reamer Labs' execution server [12], and its replication on the release measured here [13].

2. Systems under test

We compare four engines. We chose them because they are widely used, or (for Reamer Research) because the author builds it. Two Reamer Research variants are run. **reamer_cpp** drives the

engine from a compiled C++ strategy. **reamer_py** drives the same engine from a Python strategy through the ctypes binding. Running both separates the engine's own cost from the cost of writing the strategy in Python. Every other engine here is driven from Python.

Engine	Version	Licence	Core	Strategy language here	Installed by
Backtrader	1.9.78.123	GPL-3.0	Python	Python 3.13	`pip` into a uv-managed venv
NautilusTrader	1.231.0	LGPL-3.0	Rust, Cython, Python	Python 3.13	`pip` into the same venv
QuantConnect LEAN	master at f3d841e	Apache-2.0	C# (.NET 10.0.10)	Python 3.11.11 via Python.NET	built from source; QuantConnect's pinned Miniconda for Python
Reamer Research	4.6.0 (commit 68db07a)	proprietary	C++ behind a C ABI	C++ (reamer_cpp); Python 3.13 (reamer_py)	built from source, Release, `-O3 -funroll-loops -march=native`

LEAN. We ran LEAN from source with `dotnet exec` on the launcher, not through the LEAN CLI. The CLI's local backtest path needs a signed-in QuantConnect account, and local coding needs a paid plan (§9.5). The engine itself needs neither. LEAN's Python algorithms run on Python 3.11 because Python.NET loads `libpython3.11.so`. The other Python engines run on 3.13. This difference is part of what a LEAN user gets, so we did not try to equalise it.

Nautilus. NautilusTrader appears twice:

- **nautilus:** the release as shipped.
- **nautilus_p:** one method disabled (§3.8).

---

3. Method

3.1 Hardware and operating system

All measurements come from one rented bare-metal server on 2026-10-25.

CPU	AMD EPYC 9575F, one socket, 64 cores / 128 threads; logical CPUs 0–63 are distinct physical cores, 64–127 their SMT siblings
NUMA	one node
Memory	755 GiB DDR5 ECC
Storage	two NVMe drives in software RAID 1
OS	Ubuntu 26.04.1 LTS, kernel 7.0.0-38-generic, clocksource TSC
CPU policy	`performance` governor on every core; boost on; SMT on
Toolchain	GCC 15.2.0, CMake 4.2.3
Rental price	€2.83 per hour (used in §9)

Engine processes run unpinned and at default priority. Except in the parallel sweep (§3.7), only one backtest runs at a time. Nothing else ran on the machine beyond the operating system. The software RAID finished its initial resync before the first backtest in this paper started.

3.2 Data

Base series. We use 58,942 real GBP/USD 15-minute OHLCV bars, from 2024-01-03 00:00 to 2026-05-15 23:45 UTC, weekends excluded (exported from a broker's MetaTrader 5 server through the desktop client). Closes range from 1.21026 to 1.38534.

Single-ticker tiers. These are 50,000, 500,000, 2,000,000 and 5,000,000 bars.

- Each tier is built by tiling the base series end to end. Every copy's timestamps are shifted by the series' span plus one bar interval, so time increases without overlaps. The result is cut to the exact bar count.
- The 50K tier is a prefix of the real series.
- Larger tiers repeat it, which creates one price discontinuity at each seam, about 8 seams at 500K. Every engine sees the same seams, so the comparison stays like for like. The seams make the series unsuitable for judging a strategy, which this paper does not do.

Portfolio tiers. These are 10, 50 and 100 tickers × 35,064 bars, about one year of 15-minute bars.

- Ticker i is a window of the base series starting at offset $9{,}973 \cdot i \bmod (58{,}942 - 35{,}064 + 1)$.
- Its prices are scaled by a factor between 0.5 and 2.0 that depends on i.
- All tickers share one timestamp grid. Every engine sees a fully aligned panel.

Formats. Each engine reads the format it handles best in this setup:

- Reamer Research and NautilusTrader: the same flat binary file of 48-byte records.
- Backtrader: a pickled pandas DataFrame.
- LEAN: a CSV read through a custom `PythonData` class.

The values are identical across formats except as follows. The LEAN CSV rounds prices to 5 decimals, the instrument's precision, and floors volume at

1. Its timestamps are moved to start at 2000-01-01, for the reason in §3.9.

3.3 Workloads

Two strategies, written once per engine in each engine's own idiom:

- **minimal.** On each bar, read the latest close and do nothing else. This measures the cost of delivering one bar to the strategy.
- **realistic.** A 20-bar Donchian channel breakout.
  - The channel is the high and low of the 20 bars *before* the current one; the current bar never contributes to its own channel.
  - A close above the channel high sets the target position to +1,000 units. A close below the channel low sets it to −1,000.
  - The engine is sent one market order equal to the target minus the current position. A reversal is therefore one order of 2,000 units, not a close followed by an open. This matches netting semantics in all four engines and keeps order counts comparable.
  - The strategy keeps its own 20-bar rolling window rather than calling an engine indicator, except in Backtrader. Backtrader's `Highest` and `Lowest` indicators over the one-bar-lagged series are its idiomatic equivalent.

The portfolio versions apply the same rule to every ticker independently on each step.

3.4 Common configuration and deviations from defaults

All engines use these settings:

- Starting capital: 10,000,000 in the quote currency.
- Leverage: 1.
- No commission, no slippage, no spread.
- Market orders only.

Each change from an engine's out-of-the-box configuration is listed here with the reason for it.

Engine	Change	Reason
NautilusTrader	Instrument `GBP/USD` from `TestInstrumentProvider.default_fx_ccy` (price precision 5, minimum quantity 1,000)	An equity test instrument has 2-decimal prices, which flattens an FX series and removes most signals
NautilusTrader	A fee model that returns zero	The FX test instrument charges a 0.002% taker fee; every other engine runs commission-free
NautilusTrader	Bar volume set to 10^9	Bar execution sizes each simulated price level from bar volume, which split a 1,000-unit order into several partial fills. The other engines fill a market order whole, and no strategy reads volume
NautilusTrader	`run_analysis=False`, `bypass_logging=True`, NETTING OMS, MARGIN account	Skips the post-run report and per-event INFO logging (about 4 KB per bar); the strategy goes both long and short
NautilusTrader (nautilus_p only)	`PortfolioAnalyzer.record_trade` replaced by a no-op	See §3.8
LEAN	`MarginCallModel.Null`; benchmark set to a constant	No margin calls at leverage 1; avoids fetching a benchmark security that no other engine loads
Backtrader	`Cerebro(stdstats=False)`	Turns off the default observers, which the other engines do not compute
Reamer Research	`ohlcv_type=bid`, `rng_seed=42`, zero volatility, spread and slippage	Documented defaults for bar data; the seed fixes the synthetic intrabar path

Fill timing. Fill timing was left at each engine's default:

- Reamer Research and Backtrader fill a market order on the *next* bar. Backtrader fills at that bar's open. Reamer Research fills at the first tick of the bar's synthetic tick path (§3.9).
- LEAN and NautilusTrader fill on the *current* bar.

We did not force one convention, because each is a legitimate design and users get the default. §3.9 shows how this splits the results into two groups that agree internally. §4.4 removes the remaining Reamer Research–Backtrader difference by repricing.

3.5 What is measured

Each backtest runs as its own operating-system process. Two clocks are recorded:

- **Process wall time** (outer wall). Measured by GNU `time` from process start to exit. It includes:

  - interpreter or runtime start-up
  - library imports
  - reading the data file
  - building the engine's data objects
  - the backtest itself
  - writing the result

  It is the same measurement for every engine. **We use it for all cross-engine comparisons.** GNU `time` reports it to 10 ms. Runs shorter than about 0.1 s are therefore below useful resolution: the C++ Reamer Research runs at 50K bars take 20–30 ms. For those runs we give engine time as well and flag the row.

- **Engine time** (`run_s`). Measured inside each runner around the engine's run call. What that call includes differs by engine:

Engine	Inside `run_s`	Outside `run_s`
reamer_cpp	`run_backtest`	file read, conversion to the ABI bar struct
reamer_py	`run_backtest`, including the binding's handling of the NumPy array	file read, NumPy struct copy, imports
NautilusTrader	`engine.run()`	constructing one `Bar` object per bar, `add_data`, imports
Backtrader	`cerebro.run()`, which includes loading the DataFrame into Backtrader's lines	pickle read, imports
LEAN	no internal timer; the whole `dotnet` process, which includes CLR and Python.NET start-up and CSV parsing	the Python wrapper's own start-up

  Engine time therefore over-reports for Backtrader and LEAN and under-reports for NautilusTrader. We report it as a secondary figure, for readers who want the cost of the event loop within one engine across tiers. We do not compare it across engines.

- **Peak memory.** The maximum resident set size reported by GNU `time`. For LEAN, whose runner starts `dotnet` as a child, this is the largest single process in the tree, which is the .NET process. The parallel sweep instead sums each copy's self-reported peak RSS (§3.7).
- **Throughput.** Bars per second = tier ÷ process wall time. Engine-time throughput is reported separately, with the caveats above.
- **Correctness outputs:** fills (LEAN reports orders), round trips and net PnL, used in §3.9.

3.6 Repetitions and order of runs

The plan as set before the run. What was cut from it is listed under **What was cut** below.

Step	Tier	Engines	Repetitions
Bars	50K	all six variants	5
Bars	500K	all six variants	3
Bars	2M	reamer_cpp, reamer_py, backtrader, nautilus_p	2
Bars	5M	the same four	1, plus 2 more for reamer_cpp and reamer_py

Step	Tier	Engines	Repetitions
Bars	2M	nautilus (stock), realistic only	1
Breadth	10 / 50 / 100 tickers	reamer_py, backtrader, nautilus_p, lean	2 / 2 / 1
Parallel	500K realistic, K = 1 / 8 / 32 / 64	reamer_cpp, reamer_py, backtrader, nautilus_p, lean	1 batch each

Order of runs.

- Within a step, the repetition loop is outermost. Each repetition runs every engine once, then both strategies. Slow drift, such as thermal state or page cache, is therefore spread across engines rather than landing on one.
- Every engine reads data files that are already in the page cache from the data-build step.
- We report the median across repetitions; the minimum and maximum are in brackets in each table.

Budget. The steps ran in a fixed order chosen before the run, and a 5.5-hour budget was set in advance. A step whose estimate would overrun the budget was skipped and recorded as skipped.

What was cut. At 12:30 UTC, with the run behind its estimates (the LEAN breadth hangs in §3.11 cost about an hour: the 46-minute first hang plus the killed retries), we cut these to keep a two-hour margin for the reruns:

- Backtrader and patched NautilusTrader at 2M and 5M bars.
- The stock NautilusTrader run at 2M.
- LEAN at 100 tickers.

At 13:19 UTC one more followed: LEAN at K = 64 in the parallel sweep, after its K = 32 batch stalled (§3.11).

The cut runs are recorded with a "skipped" message and a non-zero exit code, and are logged as `budget_cut` events in `steps.jsonl`. They are left out of the results, not counted as failures.

The comparison therefore covers 50K and 500K bars for all engines, 10 and 50 tickers for all engines, and 100 tickers without LEAN. The 2M and 5M tiers remain as a scaling check on Reamer Research alone. The other engines are untested above 500K bars, and the results make no claim about them there.

3.7 Parallel parameter sweep

This step asks how many backtests per hour one machine completes when a researcher runs a sweep.

- K identical copies of the 500K-bar realistic backtest start at the same moment, as separate processes. Each has its own working directory: LEAN writes its config and logs there, and its data file is staged once with an atomic rename.
- Processes are not pinned, so the scheduler places them as it would for a user.

Recorded per batch:

- Batch wall time: from launch to the last copy's exit.
- Backtests per hour: successful copies × 3,600 ÷ batch wall.
- Aggregate bars per second: bars processed by all successful copies ÷ batch wall.
- Summed peak RSS of all copies: the memory the sweep needs at once.

- Parallel efficiency: backtests per hour at K ÷ (K × backtests per hour at K = 1).

At K = 64 there is one copy per physical core. We stopped there because the companion paper finds that the operating envelope ends at the physical-core count.

3.8 The NautilusTrader `record_trade` patch

What we found. In NautilusTrader 1.231.0, the portfolio calls `PortfolioAnalyzer.record_trade` every time a position closes. The call appends one row to a pandas Series with `pandas.concat`, so each close costs time proportional to the number of earlier closes. Total cost therefore grows with the square of the trade count. `run_analysis=False` does not turn it off, and we found no configuration option that does.

We re-measured on 1.231.0 with this paper's runner, on the laptop (realistic strategy, single ticker):

- At 40,000 bars (935 trades), `record_trade` takes 27% of profiled wall time; `pandas.concat`, called from it, accounts for most of that.
- Stock minus patched run time is 1.6 s at 40,000 bars (935 trades) and 17.6 s at 160,000 bars (3,768 trades). Four times the trades cost about eleven times as much, as expected for a cost that grows faster than linearly in the trade count.

What we report. No other engine builds an equivalent running PnL series during the timed run. We therefore report:

- **nautilus_p:** the method replaced with a no-op. This is the main figure.
- **nautilus:** the stock engine, at 50K and 500K for both strategies, plus one 2M realistic run with a 2-hour limit.

The difference is reported as a finding (§4). The cost was reported upstream in July 2026 [8]. The maintainers replied that it affects only the 1.x analyser, that 2.0 records PnL with amortised constant-time appends, and that 1.x would not change; the report was closed as not planned. We did not test the 2.0 release candidates.

3.9 Correctness cross-checks

A speed figure means nothing if the engines did different work. Every run is checked as follows:

- **Bar count (LEAN).** LEAN ends a backtest at the current wall-clock date. It drops any later bar without an error.
  - With the base series' real 2024 dates, a 500K-bar run processed only 65,271 bars.
  - Moving the CSV timestamps to start at 2000-01-01 fixes this up to about 900K bars of 15-minute data. LEAN therefore runs only the 50K and 500K tiers, and is absent from 2M and 5M by construction, not by failure.
  - The runner compares bars processed with bars supplied, and fails the run if they differ by more than 0.5%.
  - The first LEAN runs passed that check but still lost bars: 498,352 of 500,000 at 500K, and 49,992 of 50,000 at 50K. We logged every bar LEAN delivered and found two causes, which together account for all 1,648 missing bars:
    - **Time zone (64 bars at 500K, 8 at 50K).** LEAN reads custom data in New York time unless told otherwise. On each US clocks-forward Sunday, 02:00–02:45 does not exist in New York, so LEAN drops those four 15-minute bars without an error.

    - **End date (1,584 bars at 500K).** The base series has weekend gaps, so 500K tiled bars span 2000-01-01 to 2020-01-26, not the 14.26 years that 500,000 × 15 minutes would suggest. The algorithm's end date of 2020-01-01 cut the last 3.5 weeks.

  Both are configuration errors in our LEAN algorithm, not faults in LEAN. We fixed them during the run: the data time zone is set to UTC, as the CSV times are, and the end date to 2021-01-01. With the fix, LEAN processes all 500,000 bars and reports −1,981.89 over 11,818 orders, the same as NautilusTrader to the cent. The fix reached the box before the parallel sweep. The 50K and 500K tiers are rerun with it (§3.11).
- The breadth algorithm has the same time-zone setting and loses 8 bars per ticker, from two clocks-forward Sundays in its 2000–2001 span. Its end date is not reached. We report this and did not rerun the breadth tier.

- **PnL agreement.**
  - The two engines that fill on the current bar, LEAN and NautilusTrader, should report the same net PnL to the cent.
  - The two that fill on the next bar, Reamer Research and Backtrader, should agree with each other, apart from one modelled difference. Backtrader fills a market order at the next bar's open exactly. Reamer Research fills it at the first tick of that bar's synthetic tick path (EXECUTION_SPEC §1, §6). That tick is a deterministic, seeded point inside the bar's low–high range, so it is usually a few pips from the open. With zero spread, slippage and volatility it adds no cost, but it moves each fill price.
  - The two groups should differ only by the fill-timing effect.
  - At 50K bars:
    - LEAN and NautilusTrader: −219.05.
    - Backtrader: −219.67.
    - Reamer Research: −219.33. Repricing its 1,170 fills at the bar's open gives −219.67, matching Backtrader to the cent.

    Speed comparisons are unaffected, because every engine does the same work.

- **Open-fill sensitivity.** To test the claim that the two next-bar engines differ only in fill price, we repriced Reamer Research's fills at the open, at every tier and over several seeds.
  - The engine has no setting for open fills. Tick 0 of each bar's path is pinned to the open, but a bar-origin market order first becomes eligible at tick 1, and every bar has at least four ticks.
  - The strategy's decisions never read a fill price: they use closes, highs, lows and the strategy's own target. At 1,000 units the margin limit never binds. So the same orders fill on the same bars at every seed, and replacing each fill price with that bar's open gives exactly the PnL the engine would report with open fills. The end-of-run force-close is left as is: it fills at the last bar's close, where Backtrader values its open position.
  - An independent check, sharing no code with either engine, simulates the strategy directly: fill at the next open, and value the end position at the last close.
  - Seeds: 20 at 50K and 500K (0–18 and 42), and 5 at 2M and 5M (0–3 and 42).
  - All of this ran on a laptop with the same 4.6.0 library, on tier files byte-identical to the box's. It measures no time, only PnL.
  - Script: `open_fill_sensitivity.py` in the dataset. Results: §4.4.

- **Counting definitions.** Fills, orders and round trips are defined differently across engines:
  - LEAN reports orders submitted.

  - NautilusTrader counts orders, because its fills could be partial before the volume change in §3.4.
  - reamer_cpp counts filled orders, including the engine's end-of-run force-close (1,171 = 1,170 + 1).
  - reamer_py counts round trips through `trades`.

  We compare PnL across engines, never counts. A count is compared only across repetitions of the same engine.
- **Repeat agreement.** Every engine should give the same fills and PnL on every repetition. Any disagreement is flagged in the results tables.

3.10 Determinism

Twenty runs of the 500K realistic backtest each through:

- **reamer_py:** each run hashes the full closed-trade ledger (every field of every trade, serialised with sorted keys) with SHA-256.
- **reamer_cpp:** each run records fills, round trips and net PnL.

Determinism holds if all 20 hashes are identical. The claim covers one build on one machine. A laptop build of the same commit (GCC 15, without `-march=native`) gives a different ledger hash, with the same 11,818 trades and the same net PnL to five decimal places. Bit-identical results across builds are not claimed. For the other engines the evidence is the repeat agreement in §3.9. That is a weaker test: equal PnL and counts do not prove an identical ledger. We say so wherever the two are compared.

3.11 Failures and timeouts

Every run is recorded, whether it succeeded or not, with its exit code and the last lines of stderr. A failed run is never retried silently.

Per-run time limits:

- LEAN: 1 hour.
- NautilusTrader: 2 hours.

A run that hits its limit is reported as not finishing within the limit, with the limit stated.

Incident. On the first attempt at the 50K tier, LEAN's first backtest completed in 2.7 s of engine time, but its `dotnet` process then stopped. .NET restores terminal settings when it exits. Under `timeout`, the process ran in a background process group, so restoring the settings drew SIGTTOU from the kernel, which stopped it indefinitely. We stopped the harness 11 minutes later, fed every run's standard input from `/dev/null`, and restarted the 50K tier from the beginning. The ten records from the aborted attempt (Backtrader, NautilusTrader and Reamer Research, repetition 1) are kept separately in the dataset. They are not used in the results.

Mid-run fixes. During the breadth tier (f03) we found two errors and fixed both on the box before the parallel sweep began. Each is logged in `steps.jsonl` with its time, and each original file is kept beside its replacement.

1. **LEAN algorithm (11:31 UTC).** The time zone and end date described in §3.9.
2. **Reamer Research Python binding (11:42 UTC).** The 4.6.0 binding allocates a fixed buffer for 65,536 closed trades.
   - The 5M realistic backtest produces 118,239, so the binding raises an error and returns no result. Every 5M Python realistic run would have failed.

- The buffer also costs every run about 40 MB of peak memory and about 7 ms to allocate and zero, whether or not trades are returned: 61 MB peak with the fix against 100–104 MB without, at 50K on the laptop.
- The fix sizes the buffer from the trade count the engine has already reported. The ledger is unchanged (same SHA-256 before and after), and the binding's tests pass.
- The fix will ship in 4.6.1. Reamer Research results in this paper therefore describe 4.6.0 with this one binding change, which we state wherever Python memory is reported.

3. **LEAN breadth runner (11:59 UTC).** LEAN's 10-ticker realistic run (repetition 1) processed no bars. It stalled on the first time step at about 1% CPU. LEAN's own isolator stopped the algorithm after 20 minutes, but the `dotnet` process never exited. We killed it at 12:01, 46 minutes in, and it is recorded as failed.
    - **Reproduced on the idle box:** 3 of 6 separate attempts at the same run hung the same way. The other 3, and every later attempt, finished in 16–18 s with identical PnL. Lowering LEAN's data-feed workers from 128 to 8 did not prevent it (1 of 3 still hung).
    - **Stack dump of a hung process:** a data-feed worker thread waits for the Python GIL inside a Python import, while LEAN's main thread waits inside the .NET runtime. This points to a deadlock between Python.NET and LEAN's worker threads during start-up with several Python custom-data feeds. No sequential single-ticker run hung. In the parallel sweep, however, all 32 single-ticker copies at K = 32 stalled on their first time step (item 4). On the development laptop (fewer cores), none of 11 LEAN runs hung: 7 single-ticker runs, 2 run as parallel copies, and 2 five-ticker portfolio runs. That sample is small and uses fewer feeds, so it does not show the hang is specific to the 128-thread machine. A search of the QuantConnect forum and the Python.NET tracker on 2026-10-26 found no report of this pattern, so we describe it as observed behaviour, not as a confirmed bug in either project.
    - **Runner change:** the breadth runner now kills an attempt whose current time step has run for 45 s or more, and retries. The limit was 5 attempts, raised to 10 at 12:07 after one run needed 4. A healthy run never spends more than a second on one step. The result line records every attempt. Reported times come from the successful attempt only, and the number of hung attempts is reported beside them. Under the retry runner, 14 of 22 attempts hung, each killed after about 62 s; no run exhausted its attempts (§6).
    - The failed row is rerun after the main run.
4. **LEAN parallel sweep (13:19 UTC).** At K = 32, all 32 LEAN copies of the single-ticker 500K run stalled on their first time step from 12:54, with 0 bars processed. The single-ticker runner has no retry. We killed the 32 processes by PID at 13:19, after 1,502 s, and recorded the batch as 0 of 32. LEAN at K = 64 was not run (§3.6).

To keep each comparison on one configuration, the LEAN rows recorded before the fix are rerun after the main run, at 50K and 500K. So are the Reamer Research Python rows recorded before the binding fix, at 50K, 500K and in the breadth tier, so that every Python memory figure uses the fixed binding. The original rows are kept unedited in the dataset, and its README lists which rows the reruns supersede.

3.12 Outcomes stated in advance

The author has a conflict of interest (§0). So before any four-way result was seen (2026-10-25, about 10:45 UTC), we wrote down the outcomes that would count against Reamer Research.

We report each one in the abstract if it occurs:

1. Reamer Research is slower or uses more memory than another engine at any tier, for either strategy.
2. reamer_py's Python-strategy overhead removes its advantage over nautilus_p.
3. Reamer Research's parallel efficiency falls well below another engine's, or flattens before K = 64.
4. Any Reamer Research run fails, or two repetitions disagree.

3.13 Availability

The raw results are published on GitHub, https://github.com/j0osman/research-engine-benchmark:

- every per-run JSONL record
- the step log
- the machine provenance file, with every installed Python and conda package and its version

The same dataset holds the runner scripts, the dataset builders and the script that rebuilds every table in this paper. The runners contain the strategies, configurations and deviations described above. Running the Reamer Research runners needs the library from the public 30-day kit; no Reamer Research binary is distributed with the dataset.

3.14 Superseded rows and reading rules

What the tables use. These rows replace superseded ones (§3.11):

- LEAN and Reamer Research (Python) at 50K and 500K come from `fourway_rerun.jsonl`, run with the fixed algorithm and the fixed binding.
- Reamer Research (Python) breadth rows come from `portfolio_rerun.jsonl`.
- LEAN's 10-ticker realistic repetition 1 also comes from `portfolio_rerun.jsonl`.

Rows skipped by the budget cut (§3.6) are left out. They are not counted as failures. No Reamer Research run failed.

Order in every table: alphabetical by engine, then Reamer Research's two variants, as in §2.

Reading rules, written before the remaining results. These are applied whichever way the numbers fall:

- **Headline:** process wall time and peak RSS. Engine time is reported only within one engine.
- **Small tiers:** at 50K bars, fixed start-up cost dominates, especially for LEAN's CLR and Python.NET start-up. We do not headline a ratio from the 50K tier. The 500K tier and above carry the throughput comparison.
- **Fixed vs per-bar cost:** we fit wall $= a + b \cdot$ bars per engine across the tiers it ran, and report a (fixed cost) and $1/b$ (marginal bars per second) separately. This separates "slow to start" from "slow per bar".
- **Agreement:** any PnL disagreement within a fill-timing group is reported in the text and investigated before the paper is released.

4. Bar-count scaling

4.1 The 50K tier (5 repetitions, medians; range in brackets)

Table 4a. Minimal strategy, 50,000 bars

Engine	Process wall (s)	Bars/s (wall)	Engine time (s)	Peak RSS (MB)
Backtrader	1.26 [1.26–1.27]	39,683	1.030	162
LEAN	4.78 [4.13–4.99]	10,460	(whole process)	553 [522–585]
NautilusTrader (patched)	0.97 [0.96–0.98]	51,546	0.439	458
NautilusTrader (stock)	0.97 [0.97–0.98]	51,546	0.438	458
Reamer Research, Python	0.31 [0.31–0.31]	161,290	0.267	71
Reamer Research, C++	0.02 †	†	0.024	19

Table 4b. Realistic strategy (20-bar breakout), 50,000 bars

Engine	Process wall (s)	Bars/s (wall)	Engine time (s)	Peak RSS (MB)	Net PnL	Fills
Backtrader	1.69 [1.68–1.71]	29,586	1.451	171	−219.67	1,170
LEAN	5.71 [5.66–5.74]	8,757	(whole process)	609 [536–620]	−219.05	1,170 orders
NautilusTrader (patched)	1.47 [1.45–1.48]	34,014	0.927	472	−219.05	1,170
NautilusTrader (stock)	2.05 [2.03–2.07]	24,390	1.510	478	−219.05	1,170
Reamer Research, Python	0.38 [0.37–0.38]	131,579	0.333	71	−219.33	1,170
Reamer Research, C++	0.03 †	†	0.033	22	−219.33	1,171 ‡

† Below the 10 ms resolution of process wall time (§3.5). The engine-time rate is 1.5–2.1 M bars/s. ‡ Counts filled orders, including the end-of-run force-close (1,170 + 1). The Python path counts closed trades through the binding. Same PnL (§3.9).

What the 50K tier shows

- **Repeatability.** Every engine repeats tightly: process wall time varies by at most 2% across five repetitions, except LEAN's minimal runs (4.13–4.99 s). Every engine gives identical PnL and fill counts on every repetition.
- **Fixed cost.** Most of a 50K-bar run is the fixed cost of starting the engine:
  - LEAN spends about 4.8 s, against 2.7 s of algorithm time in its own log.
  - NautilusTrader spends about 0.5 s outside `engine.run()`: imports and building 50,000 `Bar` objects.
  - The Python Reamer Research runner spends about 0.04 s.
- **Strategy cost.** The realistic strategy adds this much over minimal, in process wall time:

Engine	Added time
Backtrader	+0.43 s
LEAN	+0.93 s
NautilusTrader (patched)	+0.50 s
NautilusTrader (stock)	+1.08 s
Reamer Research, Python	+0.07 s

This difference is the closest the 50K tier comes to a per-bar comparison. It counts the strategy's own Python work plus each engine's order handling.

- **Memory.**
  - Reamer Research (C++): 19–22 MB.
  - Reamer Research (Python): 71 MB, with the fixed binding (§3.11).
  - Backtrader: 162–171 MB.
  - NautilusTrader: 458–478 MB.
  - LEAN: 522–620 MB.

  All of these are close to each engine's fixed footprint at this tier. The per-bar slope comes from the larger tiers (§5).

PnL agreement.

- LEAN and NautilusTrader, the current-bar group, agree to the cent: −219.05.
- Within the next-bar group, Reamer Research gives −219.33 and Backtrader −219.67, with the same 1,170 fills on the same bars.
- The 0.34 difference comes from the fill price. Backtrader fills at the next bar's open. Reamer Research fills at the first tick of a seeded synthetic path inside that bar (§3.9). That tick sits within ±3.2 pips of the open, and exactly on it in 34 of 1,171 fills.
- Repricing Reamer Research's fills at the open reproduces Backtrader's −219.67 exactly. A hand simulation with open fills, marked at the last close, gives the same figure. Both engines end on the last bar's close, 1.35082: Backtrader marks the open short there, and Reamer Research force-closes it there.
- Over 20 seeds, Reamer Research's PnL runs from −224.16 to −215.83. So for this strategy, intrabar fill modelling moves the result by about ±2% at this tier, which is more than any difference between engines. §4.4 shows this at every tier.

NautilusTrader's `record_trade`. On the realistic strategy, the stock release takes 39% more wall time than the patched one (2.05 s vs 1.47 s) and 63% more engine time. It does the same work and gives the same PnL. On the minimal strategy, which closes no positions, the two are identical, as expected. The gap grows with the trade count (§4.3).

4.2 500K, 2M and 5M tiers

Scope. Backtrader, LEAN and NautilusTrader ran at 50K and 500K. Only Reamer Research ran at 2M and 5M.

- The plan had included Backtrader and patched NautilusTrader at 2M and 5M, and stock NautilusTrader once at 2M.
- These were cut during the run to keep the box within its budget (§3.6).
- The four-way comparison does not need them. 500K bars is about 14 years of 15-minute data, which is more than most single-instrument research uses.
- The two larger tiers are kept as a scaling check on Reamer Research alone. They make no claim about the other engines.

Table 4c. Minimal strategy, 500,000 bars (3 repetitions)

Engine	Process wall (s)	Bars/s (wall)	Engine time (s)	Peak RSS (MB)
Backtrader	10.54 [10.43–10.64]	47,438	10.291	209
LEAN	18.52 [18.33–19.19]	26,998	(whole process)	617 [608–620]

Engine	Process wall (s)	Bars/s (wall)	Engine time (s)	Peak RSS (MB)
NautilusTrader (patched)	6.43 [6.41–6.53]	77,760	4.351	603
NautilusTrader (stock)	6.54 [6.46–6.55]	76,453	4.365	602
Reamer Research, Python	2.74 [2.73–2.76]	182,482	2.691	309
Reamer Research, C++	0.26 [0.26–0.26]	1,923,077	0.248	153

Table 4d. Realistic strategy, 500,000 bars (3 repetitions)

Engine	Process wall (s)	Bars/s (wall)	Engine time (s)	Peak RSS (MB)	Net PnL	Fills
Backtrader	15.04 [14.79–15.15]	33,245	14.632	309	−1,956.83	11,818
LEAN	22.10 [21.23–22.28]	22,624	(whole process)	1,126 [1,040–1,143]	−1,981.89	11,818 orders
NautilusTrader (patched)	13.61 [13.51–13.65]	36,738	11.443	717	−1,981.89	11,818
NautilusTrader (stock)	76.85 [76.49–77.32]	6,506	74.743	739	−1,981.89	11,818
Reamer Research, Python	3.38 [3.34–3.39]	147,929	3.326	309	−1,937.16	11,818
Reamer Research, C++	0.36 [0.36–0.36]	1,388,889	0.338	185	−1,937.16	11,819 ‡

LEAN processes all 500,000 bars in every repetition.

Table 4e. Reamer Research alone at 2M and 5M bars (2 repetitions at 2M, 3 at 5M)

Tier	Variant	Strategy	Process wall (s)	Bars/s (wall)	Engine time (s)	Peak RSS (MB)	Net PnL	Fills
2M	Python	minimal	10.79 [10.64–10.93]	185,443	10.699	1,101		
2M	C++	minimal	1.06 [1.05–1.06]	1,895,735	0.999	599		
2M	Python	realistic	13.55 [13.40–13.71]	147,547	13.450	1,101	−7,738.29	47,298
2M	C++	realistic	1.50 [1.50–1.51]	1,328,904	1.395	721	−7,738.29	47,299 ‡
5M	Python	minimal	27.12 [26.92–27.34]	184,366	26.970	2,686		
5M	C++	minimal	2.66 [2.65–2.66]	1,879,699	2.512	1,571		
5M	Python	realistic	33.55 [33.33–33.91]	149,031	33.370	2,686	−19,181.24	118,239
5M	C++	realistic	3.80 [3.77–3.81]	1,315,789	3.526	1,795	−19,181.24	118,240 ‡

The 5M Python realistic run returns all 118,239 closed trades. Under the unfixed 4.6.0 binding it would have failed (§3.11).

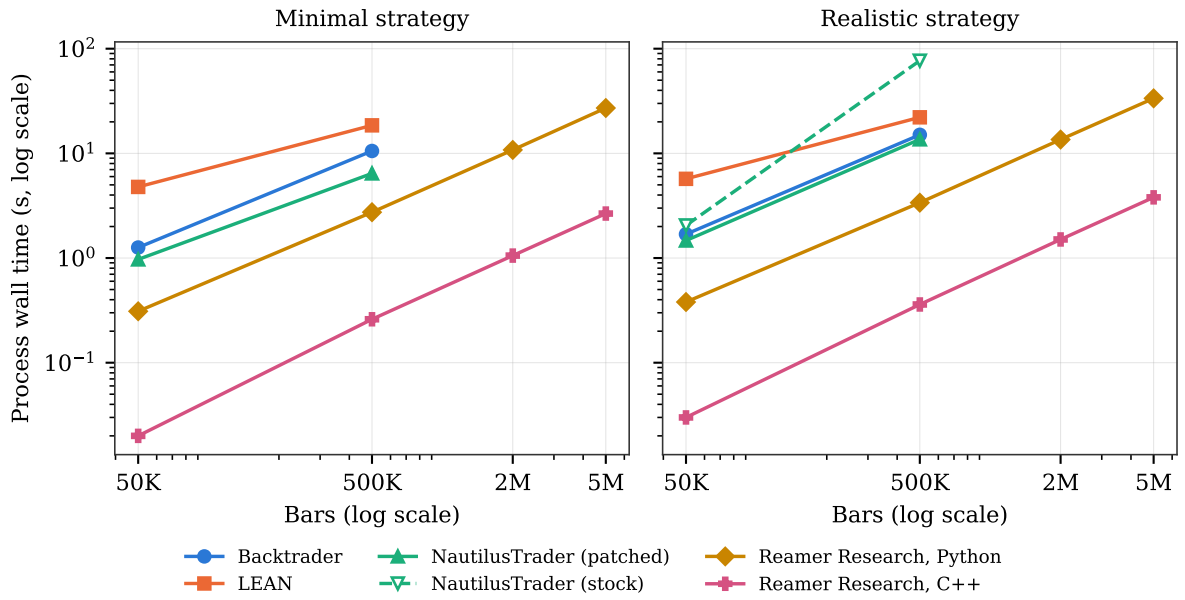


Figure 1. Process wall time against bars, medians. Backtrader, LEAN and NautilusTrader end at 500K bars by the scope of §3.6.

Table 4f. Fixed cost and marginal rate, from wall $= a + b \cdot$ bars over each engine's tiers. With two tiers the line passes through both points, so there is no R^2 to report.

Engine	Strategy	Fixed cost a (s)	Marginal bars/s ($1/b$)	Tiers used	R^2
Backtrader	minimal	0.23	48,491	50K, 500K	—
Backtrader	realistic	0.21	33,708	50K, 500K	—
LEAN	minimal	3.25	32,751	50K, 500K	—
LEAN	realistic	3.89	27,456	50K, 500K	—
NautilusTrader (patched)	minimal	0.36	82,418	50K, 500K	—
NautilusTrader (patched)	realistic	0.12	37,068	50K, 500K	—
NautilusTrader (stock)	realistic	not linear (§4.3)	—	50K, 500K	—
Reamer Research, Python	minimal	0.02	184,632	50K–5M	1.0000
Reamer Research, Python	realistic	0.06	149,181	50K–5M	1.0000
Reamer Research, C++	minimal	$-0.01 \approx 0$	1,875,132	50K–5M	1.0000
Reamer Research, C++	realistic	$-0.02 \approx 0$	1,311,102	50K–5M	1.0000

Reamer Research's small negative intercepts are below the 10 ms resolution of wall time. They mean zero.

What the larger tiers show

- **Marginal rate, realistic strategy, fastest first:**

Engine	Marginal rate
Reamer Research (C++)	1.31 M bars/s
Reamer Research (Python)	149 K
NautilusTrader (patched)	37 K
Backtrader	34 K
LEAN	27 K

The minimal strategy gives the same order. NautilusTrader's lead over Backtrader is wider

there (82 K against 48 K).

- **Fixed cost:**
  - Reamer Research, Backtrader and NautilusTrader all start in under 0.4 s.
  - LEAN takes 3.3–3.9 s to start: the CLR, Python.NET and its own initialisation.
- **Which cost matters to whom:**
  - A researcher running many short backtests pays LEAN's fixed cost each time. At 50K bars it is most of the run.
  - A researcher running a few long backtests is limited by the marginal rate. There, LEAN and Backtrader are close, and patched NautilusTrader is about 10% faster than Backtrader on the realistic strategy.
- **Linearity.** Reamer Research's wall time is linear in bars from 50K to 5M ($R^2 = 1.0000$ on four tiers). The 5M run costs 2.5× the 2M run, for 2.5× the bars.
- **Pre-stated adverse outcome 2 did not occur at 500K.** It asked whether Reamer Research's Python overhead removes its lead over patched NautilusTrader.
  - At 500K realistic, Reamer Research (Python) takes 3.38 s against 13.61 s, so it is 4.0× faster.
  - At 500K minimal it takes 2.74 s against 6.43 s, so it is 2.3× faster.
  - Above 500K the check cannot be made, because NautilusTrader was not run there (scope above).

PnL at 500K.

- LEAN and NautilusTrader (stock and patched) agree to the cent: −1,981.89 over 11,818 fills.
- Backtrader gives −1,956.83 and Reamer Research −1,937.16, with the same 11,818 fills. Repricing Reamer Research's fills at the open gives Backtrader's figure exactly (§4.4).

4.3 NautilusTrader with and without `record_trade`

Stock and patched NautilusTrader were compared at 50K and 500K. The planned 2M run of the stock engine was cut (§4.2).

Tier	Closed trades	Stock: wall / engine (s)	Patched: wall / engine (s)	Extra engine time, stock
50K	1,169	2.05 / 1.510	1.47 / 0.927	0.58 s
500K	11,817	76.85 / 74.743	13.61 / 11.443	63.30 s

The extra time grows with the square of the trade count.

- Ten times the trades (1,169 → 11,817) costs 109 times the extra engine time (0.58 s → 63.30 s).
- That gives a fitted exponent of 2.03, which matches the quadratic mechanism in §3.8: each close appends to a pandas Series with `concat`.
- At 500K bars the stock engine spends 85% of its engine time in this one function.
- On the minimal strategy, which closes no positions, stock and patched are within 2%.

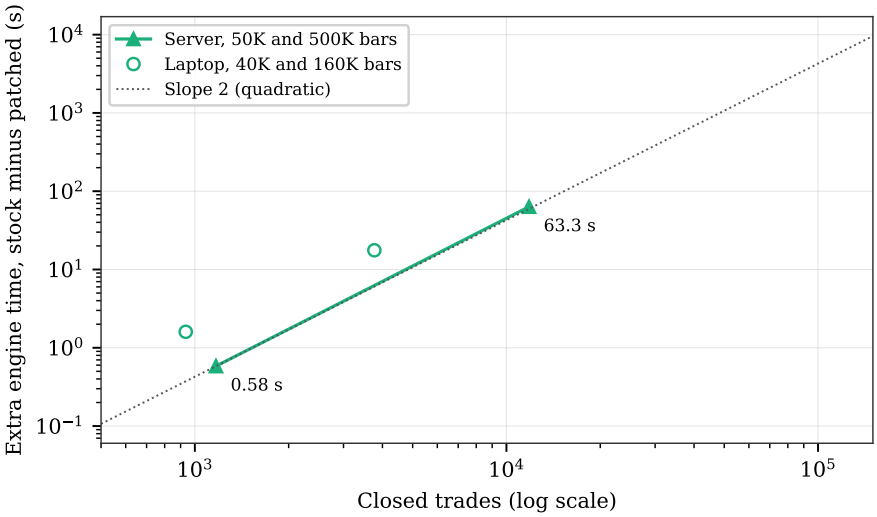


Figure 2. Extra engine time of stock over patched NautilusTrader 1.231.0 against closed trades, realistic strategy, log-log. The dotted line has slope 2.

Extrapolation, not measured. If the cost stays quadratic, the stock engine would add about:

- 17 minutes at 2M bars (47,298 trades).
- 1.8 hours at 5M bars (118,239 trades).

This is the cost a NautilusTrader 1.231.0 user meets with the default portfolio on a strategy that trades often. Turning off analysis does not avoid it (§3.8). The maintainers' answer to the upstream report is in §3.8 [8].

4.4 Open-fill sensitivity

Method: §3.9. Every figure in this section is net PnL on the realistic strategy. None is a timing.

Table 4g. Reamer Research with fills repriced at the open, against an independent simulation and Backtrader

Tier	Fills (excl. force-close)	Repriced at open	Hand simulation	Backtrader (box)
50K	1,170	−219.67	−219.67	−219.67
500K	11,818	−1,956.83	−1,956.83	−1,956.83
2M	47,298	−7,766.67	−7,766.67	not run (§4.2)
5M	118,239	−19,215.60	−19,215.60	not run (§4.2)

The repriced figure is the same at every seed, as expected: the seed moves fill prices, never which bars fill. At 2M and 5M the hand simulation is the only independent check.

Table 4h. Reamer Research engine PnL across seeds (tick-1 fills)

Tier	Seeds	Seed 42 (benchmark)	Median	Min	Max	SD	Repriced at open
50K	20	−219.33	−219.83	−224.16	−215.83	2.55	−219.67
500K	20	−1,937.16	−1,952.24	−1,980.59	−1,933.41	13.12	−1,956.83
2M	5	−7,738.29	−7,746.00	−7,757.04	−7,735.00	9.59	−7,766.67
5M	5	−19,181.24	−19,169.22	−19,187.98	−19,166.99	9.47	−19,215.60

The box's 2M and 5M runs, at seed 42, reproduce the laptop's seed-42 figures to the cent, from

both the C++ and Python paths.

Fill offset from the open, over all seeds:

- The mean offset is about −0.002 pips, so the tick-1 fill is unbiased relative to the open. The mean absolute offset is 0.34 pips.
- About 3.2–3.4% of fills land exactly on the open.
- Range: −6.1 to +5.3 pips at 50K, −7.0 to +8.7 at 500K, −9.4 to +10.5 at 2M, −10.9 to +12.3 at 5M. The range widens with tier only because more bars sample more of the tail.
- At every tier and seed the force-close fills at the last bar's close.

What this shows

- **The two next-bar engines agree exactly.** With fills at the open, Reamer Research reproduces Backtrader to the cent at 50K and 500K, and an independent simulation agrees at every tier. The only difference between them is the modelled fill price.
- **The median seed drifts from the open-fill figure as bars grow.** Median engine PnL minus open-fill PnL: −0.16 at 50K, +4.6 at 500K, +20.7 at 2M, +46.4 at 5M. Above 50K, tick-1 fills come out slightly better than open fills, by about 0.2% of PnL.
- **Why the drift exists.** The offset is unbiased over all fills, so the drift has to come from which side each fill is on. We measured it, splitting engine PnL minus open-fill PnL by side, for seeds 0, 1, 2 and 42 at 50K and 500K, and seed 42 at 2M and 5M:
  - In the bar where a fill happens, price on average moves against the signal that caused it. Buys fill in bars that fall by 0.23 pips from open to close, and sells in bars that rise by 0.23 pips (500K–5M). The breakout strategy enters after a move, and the next bar partly reverses it.
  - Tick 1 of the bridge sits slightly along the bar's open-to-close path. A least-squares fit gives a tick-1 offset of 0.6–1.3% of the bar's move, about 0.8% on average, with a correlation of 0.08–0.17.
  - So a tick-1 buy fills, on average, slightly below the open, and a sell slightly above it.
  - The bar moves against the strategy by $\Sigma(-\text{qty} \times \text{move}) = 561.5$ at 500K, 2,183.5 at 2M and 5,437.4 at 5M. At 0.8% of that, the expected drift is +4.5, +17 and +44. The measured median drifts are +4.6, +20.7 and +46.4. At 50K the expected drift is +0.5, well inside that tier's seed SD of 2.6.
  - Which side contributes more varies from seed to seed. The sum does not.

  The drift is therefore a property of the bridge model meeting this strategy's entries, not a bias in the fill price. It is about 0.2% of PnL. Backtrader's open fills see none of the bar's move.
- **Seed variation is small against the strategy's result, and grows slowly.** The seed SD rises from 2.6 at 50K to about 9–13 above it, while PnL grows almost a hundredfold. At 500K and above it is under 1% of PnL.
- **Using one seed is fair to the other engines.**
  - Seed 42 sits inside the seed range at every tier.
  - At 50K, 500K and 2M it is better than the median, by 0.5, 15.1 and 7.7. At 5M it is 12.0 worse.
  - It is never the best seed. Ranked from best: 9th of 20 at 50K, 3rd of 20 at 500K, 2nd of 5 at 2M and 4th of 5 at 5M.
  - The most it flatters the result is 15.1 at 500K, which is 0.8% of PnL.

5. Memory

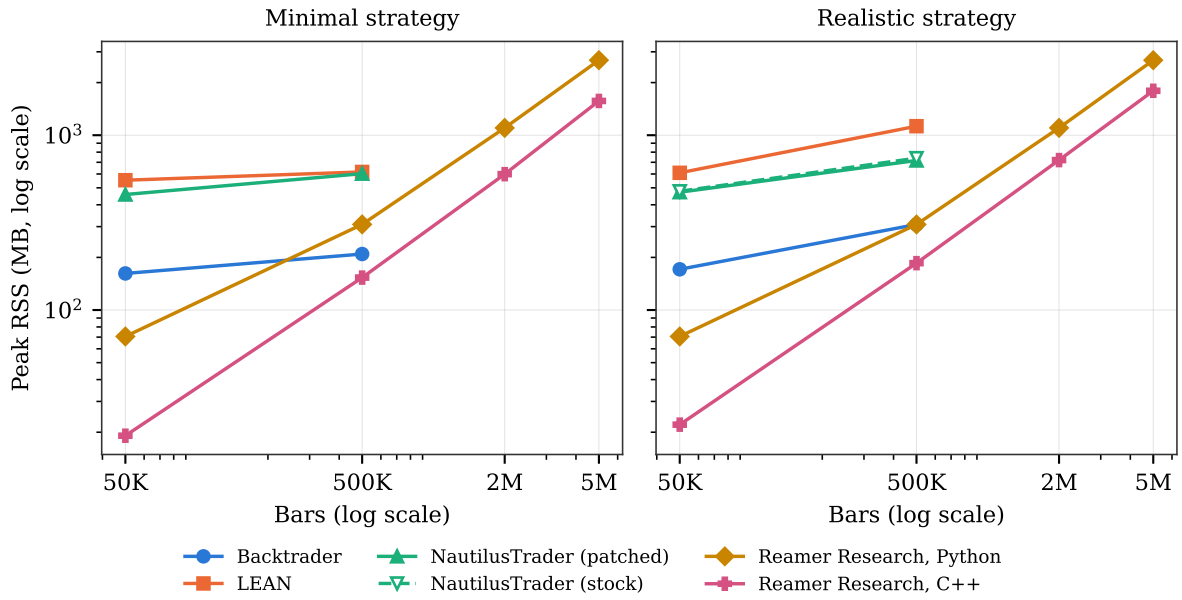


Figure 3. Peak RSS against bars, medians.

Table 5. Memory: fixed footprint and bytes per bar, from peak RSS $= a + b \cdot$ bars over the same tiers as Table 4f

Engine	Strategy	Footprint a (MB)	Bytes per bar (b)
Backtrader	minimal	157	110
Backtrader	realistic	156	321
LEAN	minimal	546	149
LEAN	realistic	552	1,204
NautilusTrader (patched)	minimal	442	337
NautilusTrader (patched)	realistic	445	570
NautilusTrader (stock)	realistic	449	609
Reamer Research, Python	minimal	44	554
Reamer Research, Python	realistic	44	554
Reamer Research, C++	minimal	≈ 0	329
Reamer Research, C++	realistic	5	375

Pre-stated adverse outcome 1, memory: it occurred. Backtrader uses less memory than Reamer Research's Python path in these cases:

Case	Backtrader (MB)	Reamer Research, Python (MB)
500K minimal	209	309
500K realistic	308.6	308.7 (a tie)
Breadth, 50 tickers, minimal	342	401
Breadth, 100 tickers, minimal	513	739
Parallel K = 64, summed	19,697	19,721 (a tie)

The reasons for the gap:

- Reamer Research's Python path has the smallest fixed footprint of the Python engines, at

44 MB.

- But it holds about 554 bytes per bar, which is more than Backtrader's 110–321 bytes. Backtrader's footprint becomes the smaller one above about 270K bars on the minimal strategy, and about 500K on the realistic one.
- Reamer Research's per-bar memory is the same for both strategies. So the cost comes from how bars are held, not from strategy state.
- We traced the per-bar cost on the laptop (RSS by stage at 500K and 2M bars, and a valgrind massif heap profile of the C++ path at 500K). It has three parts:
  - The runner's own arrays: 48 bytes per bar as read from disk, plus a 64-byte copy in the engine's bar format. Both runners hold these.
  - The engine, about 290 bytes per bar of heap at its peak: its own 64-byte copy of the bars, one timestamp string per bar, the portfolio timeline and replay state, plus vector growth slack.
  - In the Python path only, a transient of about 420 bytes per bar. The 4.6.0 binding converts a NumPy structured array by building a Python tuple per bar, then a new array from the list. That list is freed after conversion, but it sets the peak, because it is larger than the engine's own allocation.
- The transient is a binding inefficiency, not a property of the engine. Converting the same array field by field gives identical results (11,818 fills, −1,937.16 at 500K realistic) and lowers the laptop peak at 500K from 299 MB to 216 MB (minimal) and 248 MB (realistic). It also removes about 2.0 s from an 8.8 s laptop run. We report the measured 4.6.0 figures above unchanged; the conversion time is included in every Python-path wall time in this paper.

Reamer Research's C++ path uses the least memory of any engine at every tier it ran. Its per-bar cost, 329–375 bytes, is about the same as patched NautilusTrader's. Its advantage is the footprint: about 0 MB against 442 MB. That advantage shrinks as bars grow. LEAN's realistic strategy uses the most memory per bar, at about 1.2 KB.

At 5M bars, Reamer Research uses 1.6–1.8 GB (C++) and 2.7 GB (Python).

6. Ticker breadth

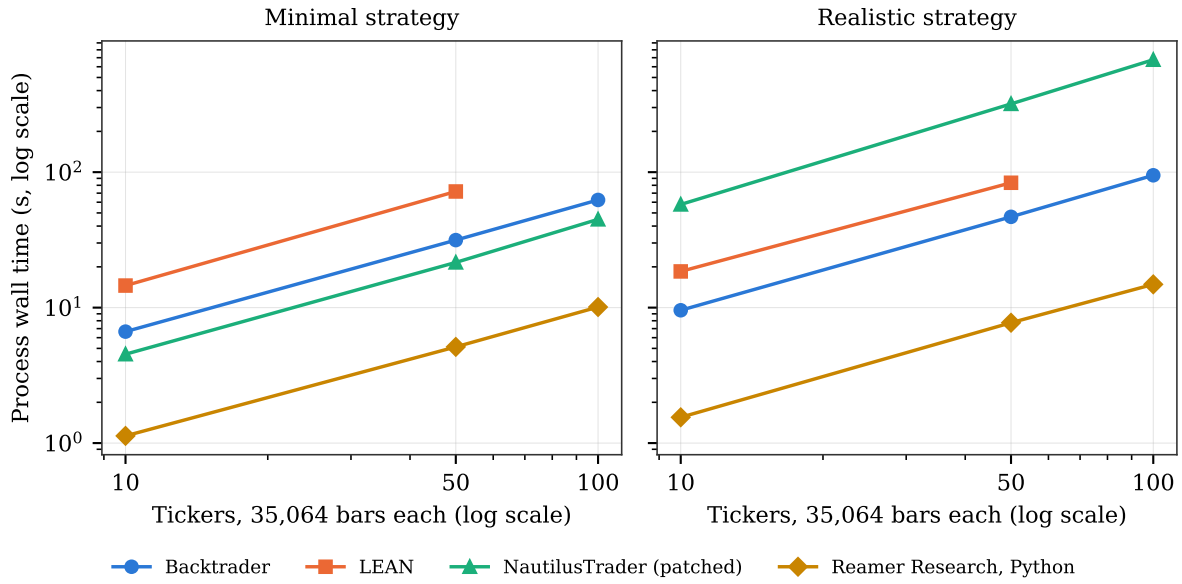


Figure 4. Process wall time against ticker count, medians (LEAN: run time, §3.5).

Each ticker has 35,064 bars. Each engine advances a multi-asset clock in its own way:

- Backtrader: synchronised data feeds.
- LEAN: time slices.
- NautilusTrader: one event stream across instruments.
- Reamer Research: an aligned panel.

Scope.

- LEAN at 100 tickers was cut for the box budget (§3.6).
- Reamer Research's breadth runner is Python only.
- Each cell has 2 repetitions at 10 and 50 tickers, and 1 at 100.

Table 6. Ticker breadth (medians)

Tickers	Engine	Minimal: wall (s)	Minimal: RSS (MB)	Realistic: wall (s)	Realistic: RSS (MB)	Net PnL	Fills
10	Backtrader	6.65	196	9.56	258	−1,583.02	8,160
10	LEAN §	14.55	788	18.50	788	−1,511.98	8,139 orders
10	NautilusTrader (patched)	4.53	553	57.76	661	−1,511.28	8,139
10	Reamer Research, Python	1.13	132	1.55	150	−1,577.19	8,160
50	Backtrader	31.52	342	46.80	670	−9,058.68	40,870
50	LEAN §	72.01	1,198	83.51	1,812	−8,805.54	40,823 orders
50	NautilusTrader (patched)	21.56	1,064	319.12	1,977	−8,811.30	40,831
50	Reamer Research, Python	5.13	401	7.74	503	−9,058.94	40,870
100	Backtrader	62.35	513	94.82	1,190	−17,863.73	81,744
100	LEAN	cut		cut			
100	NautilusTrader (patched)	44.84	1,702	675.92	4,710	−17,366.93	81,647

Tickers	Engine	Minimal: wall (s)	Minimal: RSS (MB)	Realistic: wall (s)	Realistic: RSS (MB)	Net PnL	Fills
100	Reamer Research, Python	10.10	739	14.85	942	−17,917.53	81,744

§ LEAN's wall time is the successful attempt's `dotnet` process. Hung attempts that were killed and retried are excluded and reported below. The runner polled the attempt once a second, so these times have about 1 s resolution.

Scaling with ticker count. Wall time $\propto$ tickers$\hat{}k$:

Engine	Minimal k	Realistic k
Backtrader	0.97	1.00
LEAN (10 → 50)	0.99	0.94
NautilusTrader (patched)	1.00	1.07
Reamer Research, Python	0.95	0.98

Every engine scales close to linearly in total bars. The ranking is the same at every ticker count.

What breadth shows

- **Speed.** Reamer Research (Python) is the fastest engine at every ticker count. On the realistic strategy it is 6.0–6.4× faster than Backtrader, the next fastest there.
- **Minimal strategy.** Patched NautilusTrader is the second fastest, about 4–4.5× slower than Reamer Research.
- **NautilusTrader's realistic breadth cost.** Its realistic strategy costs 12.7–15× its minimal strategy in breadth. In the single-ticker test, the same ratio is 2.1× at 500K.
  - Per bar of wall time, that is about 165 µs at 10 tickers, against about 27 µs for one ticker at 500K.
  - `record_trade` is patched out in these runs, so it is not the cause.
  - We profiled the 10-ticker realistic run on the laptop, where it reproduces (180 s against 9.4 s minimal). About three quarters of the samples fall inside NautilusTrader's `Portfolio`, called from the execution engine on order events. Our strategy code is not the cause: it accounts for about 2% of samples.
  - The cause is the closed-position snapshot cache in `Portfolio._ensure_snapshot_pnls_cached_for` (1.231.0). In NETTING mode, every close and reopen stores a pickled snapshot. The method is meant to unpickle only the new snapshots for one instrument. But its pruning step drops cached entries for every position not belonging to *that* instrument. With two or more instruments, each update therefore erases the other instruments' progress, and their next update unpickles their whole snapshot history again.
  - Measured by counting `pickle.loads` at 8,766 bars per ticker: 629 unpickles for one ticker (210 round trips), 25,453 for two (400) and 95,541 for five (1,018). Per-bar cost grows with run length: 206, 304 and 513 µs per bar at 8,766, 17,532 and 35,064 bars per ticker on ten tickers. One ticker at 35,064 bars takes 2.1 s; two take 22.4 s.
  - Distinct real base currencies instead of the runner's synthetic tickers give the same times (173 s at 10 tickers) and the same PnL, so the instrument setup is not the cause.
  - This is a property of NautilusTrader that any multi-instrument NETTING strategy pays, so we report the measured numbers unchanged. No matching upstream report was found as of 2026-10-26. The 2.0 release candidates were not tested.

- **Memory.**
  - Reamer Research uses the least memory at 10 tickers on both strategies, and at 50 and 100 tickers on the realistic strategy.
  - Backtrader uses less on the minimal strategy at 50 and 100 tickers (§5).
  - Patched NautilusTrader reaches 4.7 GB at 100 tickers, realistic.

PnL agreement.

- Next-bar group: Reamer Research and Backtrader fill the same number of orders on the same bars at every ticker count. Their PnL differs by the fill-price effect of §4.4: 0.37% at 10 tickers, 0.003% at 50 and 0.30% at

  100.

- Fills against round trips:
  - Reamer Research reports round trips equal to fills, because it force-closes at the end.
  - Backtrader reports one fewer per ticker, because positions still open are marked, not closed.

- Current-bar group: LEAN and NautilusTrader agree on fills at 10 tickers (8,139). Their PnL differs by 0.70.
  - At 50 tickers, LEAN has 8 fewer orders than NautilusTrader, and a PnL 5.76 better.
  - LEAN processes 35,056 of 35,064 time steps per ticker in every repetition. These are the 8 bars per ticker lost to the clocks-forward Sundays (§3.9). The breadth algorithm was not rerun with the fix.
  - We checked this on the laptop. We dropped the same 8 bars per ticker from NautilusTrader's input and reran it. It then matched LEAN exactly at both breadths: 8,139 orders and −1,511.98 at 10 tickers, and 40,823 orders and −8,805.54 at 50. Both differences are therefore the 8 bars, not an engine difference.

LEAN start-up hangs (§3.11). Every retry is listed here, so the cost of the hang is visible.

Run	Attempts	Hung attempts
10 tickers, rep 1, minimal	1	0 (before the retry runner)
10 tickers, rep 1, realistic	1 (rerun)	0. The original was killed after 46 minutes.
10 tickers, rep 2, minimal	2	1
10 tickers, rep 2, realistic	4	3
50 tickers, rep 1, minimal	4	3
50 tickers, rep 1, realistic	4	3
50 tickers, rep 2, minimal	4	3
50 tickers, rep 2, realistic	2	1

- Under the retry runner, 14 of 22 attempts hung, about 62 s each before the kill. On the idle box, the same test hung 3 of 6 times.
- Every attempt that started processing finished, with the same PnL as the other repetition.

7. Parallel sweeps

K identical copies of the 500K realistic backtest, launched at once.

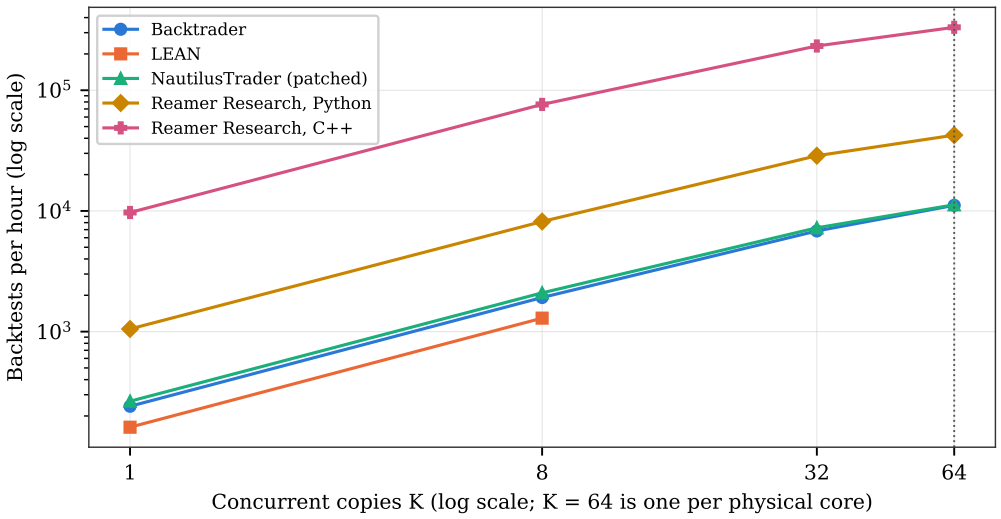


Figure 5. Backtests per hour against concurrent copies K, 500K-bar realistic backtest. LEAN completed at K = 1 and 8 only (§3.11).

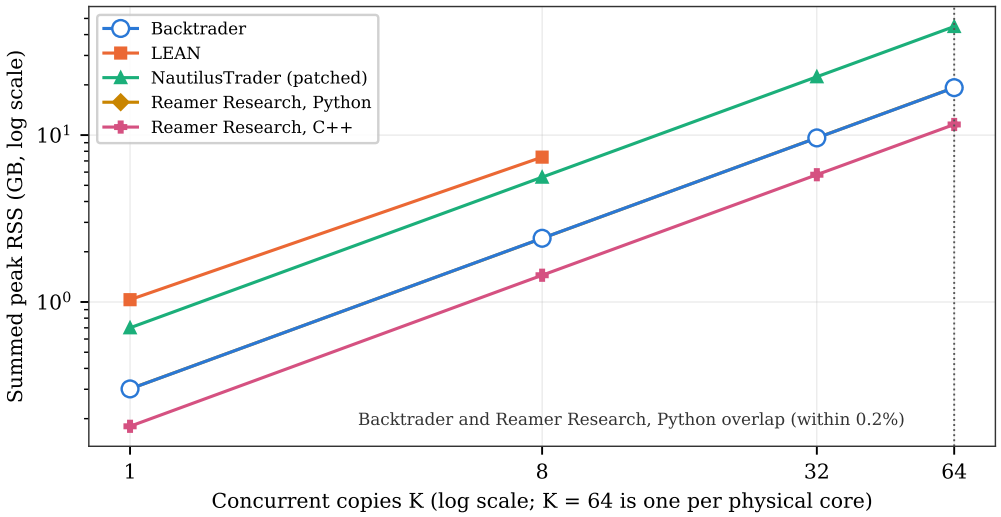


Figure 6. Summed peak RSS against K.

Table 7. Parallel sweep

Engine	K	Copies OK	Batch wall (s)	Median copy run (s)	Backtests per hour	Efficiency	Aggregate bars/s	Summed RSS (GB)
Backtrader	1	1/1	14.96	14.55	241	1.00	33 K	0.3
Backtrader	8	8/8	15.02	14.42	1,917	1.00	266 K	2.5
Backtrader	32	32/32	16.88	15.95	6,826	0.89	948 K	9.8
Backtrader	64	64/64	20.71	19.58	11,125	0.72	1.55 M	19.7
LEAN	1	1/1	22.30	22.26	161	1.00	22 K	1.1
LEAN	8	8/8	22.27	19.45	1,293	1.00	180 K	7.6
LEAN	32	0/32	killed at 1,502	—	0	—	—	—
LEAN	64	not run						
NautilusTrader (patched)	1	1/1	13.59	11.46	265	1.00	37 K	0.7
NautilusTrader (patched)	8	8/8	13.75	11.50	2,095	0.99	291 K	5.7

Engine	K	Copies OK	Batch wall (s)	Median copy run (s)	Backtests per hour	Efficiency	Aggregate bars/s	Summed RSS (GB)
NautilusTrader (patched)	32	32/32	15.90	12.67	7,246	0.85	1.01 M	22.9
NautilusTrader (patched)	64	64/64	20.50	16.58	11,242	0.66	1.56 M	45.8
Reamer Research, Python	1	1/1	3.42	3.36	1,052	1.00	146 K	0.3
Reamer Research, Python	8	8/8	3.53	3.36	8,169	0.97	1.13 M	2.5
Reamer Research, Python	32	32/32	4.02	3.73	28,673	0.85	3.98 M	9.9
Reamer Research, Python	64	64/64	5.42	4.60	42,468	0.63	5.90 M	19.7
Reamer Research, C++	1	1/1	0.37	0.34	9,709	1.00	1.35 M	0.2
Reamer Research, C++	8	8/8	0.38	0.34	76,296	0.98	10.6 M	1.5
Reamer Research, C++	32	32/32	0.49	0.40	232,730	0.75	32.3 M	5.9
Reamer Research, C++	64	64/64	0.69	0.52	332,249	0.53	46.1 M	11.9

Every successful copy of every engine gives its engine's single-run PnL:

- Reamer Research: −1,937.15947.
- Backtrader: −1,956.83.
- LEAN and NautilusTrader: −1,981.89.

LEAN at K = 32 and 64.

- All 32 LEAN copies stalled on their first time step from 12:54 UTC, with 0 bars processed. This is the same start-up hang as in breadth (§6), now in single-ticker runs under concurrency.
- The single-ticker runner has no retry. We killed the 32 processes by PID at 13:19, after 1,502 s.
- K = 64 was not run, to stay within the box budget (§3.6).
- So LEAN's parallel figures stop at K = 8, where its efficiency was still 1.00.

Pre-stated adverse outcome 3: it occurred, for the C++ path.

- At K = 64, Reamer Research's C++ path has the lowest parallel efficiency of any engine.

Engine	Efficiency at K = 64
Reamer Research (C++)	0.53
Reamer Research (Python)	0.63
NautilusTrader	0.66
Backtrader	0.72

- At K = 32 it is also the lowest: 0.75, against 0.85–0.89.
- Its throughput does not flatten before K = 64: it rises from 232,730 to 332,249 backtests per hour.
- At K = 64 it still completes about 30× as many backtests per hour as any other engine:

332,249 against NautilusTrader's 11,242.

What the data shows about the drop:

- **Part of it is fixed overhead in a very short batch.** A C++ copy takes 0.34 s. At K = 64 the batch wall is 0.15 s longer than the slowest copy's run, from launching and reaping 64 processes. That overhead is proportionally much larger for a 0.34 s batch than for a 15 s one.
- **Part of it happens inside the run.** The median copy's own run time rises 53% from K = 1 to K = 64. For the other engines it rises:

Engine	Rise in median copy run time
Reamer Research (Python)	37%
Backtrader	35%
NautilusTrader	45%

- We did not measure the cause. Memory bandwidth and the shared L3 are candidates, since the C++ path moves the most bytes per second. Neither was tested.

Against the companion paper. The companion paper finds that the envelope ends at one workload per physical core. Every engine here still gains throughput up to K = 64, one copy per physical core. Every engine also loses efficiency between K = 32 and K = 64. We did not test beyond K = 64.

Memory at K = 64, measured:

Engine	Summed RSS at K = 64
Reamer Research (C++)	11.9 GB
Reamer Research (Python)	19.7 GB
Backtrader	19.7 GB
NautilusTrader	45.8 GB
LEAN, by its K = 8 figure	about 60 GB (not measured)

These are higher than the estimates from the 50K footprints, because at 500K the per-bar memory dominates (§5). Every engine that ran at K = 64 fits easily in the box's 755 GiB of memory.

8. Reproducibility

Table 8. Distinct outcomes over repeated realistic runs of the same backtest. Runs are counted across single-ticker tiers, breadth, the parallel copies and, for Reamer Research, the determinism step. Outcomes are counted per configuration.

Engine	Runs	Distinct PnL per configuration	Distinct fill counts per configuration	Distinct ledger hashes
Backtrader	118	1	1	not measured
LEAN	21	1	1	not measured
NautilusTrader (patched)	118	1	1	not measured
NautilusTrader (stock)	8	1	1	not measured
Reamer Research, C++	138	1	1	—
Reamer Research, Python	143	1	1	**1** across 20 runs (`b69821c6cb69...`)

Engine	Runs	Distinct PnL per configuration	Distinct fill counts per configuration	Distinct ledger hashes

The LEAN count covers only completed runs. The 32 stalled parallel copies and 15 hung breadth attempts produced no result: the 14 attempts above plus the original 10-ticker run.

What can be claimed:

- Reamer Research's full trade ledger is bit-identical across 20 runs. Its C++ and Python paths agree on PnL in every run: −1,937.15947 at 500K.
- No engine gave two different outcomes for the same configuration in any run that completed. That includes 64 simultaneous copies.
- For the other engines, only PnL and counts were compared across repetitions. That is consistent with determinism but does not prove it. We say this in the text rather than leaving the table to imply equality.
- None of the four engines uses randomness in this configuration, so determinism is expected of all of them. The table records it rather than ranking it.
- LEAN's start-up hang is a completion problem, not a reproducibility one. Every LEAN run that started processing bars finished with the same result.

9. Beyond speed: capability and cost

Speed and memory are only part of the choice of an engine. This section compares what the benchmarks above do not measure. Competitor facts come from each project's own documentation, repository or price page, fetched on 2026-10-25 and 2026-10-26 [9, 10, 6, 7, 4, 5, 3]. Reamer Research facts come from its ABI header and published documentation [11]. Prices change, and a reader should check them again.

9.1 Order model and data granularity

Competitor rows come from each project's order documentation [10, 6] and Backtrader's order reference [3]. Reamer Research rows come from its 4.6 ABI header and execution specification [11].

	Backtrader	LEAN	NautilusTrader	Reamer Research
Order kinds	market, close, limit, stop, stop limit, stop trail, stop trail limit	market, limit, limit-if-touched, stop market, stop limit, trailing stop, market-on-open, market-on-close, combo orders, option exercise	market, limit, stop market, stop limit, market-to-limit, market-if-touched, limit-if-touched, trailing stop market and limit	market, limit, stop; a close that the engine resolves to the open side, full or partial; a reversal in one order
Time in force	until cancelled, or until a date or time (`valid`)	good till cancelled, day, good till date	GTC, IOC, FOK, GTD, DAY, at the open, at the close	GTC, IOC, GTD
Contingent orders	OCO, bracket	contingent orders	OTO, OCO, OUO, brackets	take profit and stop loss on any order and on the open position; when both are touched in one bar, the first in intrabar tick order wins

	Backtrader	LEAN	NautilusTrader	Reamer Research
Changes to working orders	cancel	update quantity and prices (not market orders); cancel; liquidate	modify quantity, price and trigger price; cancel one or all; close one or all positions	modify a resting order's price and brackets; modify a position's take profit and stop loss, including a stop moved to breakeven or into profit; cancel one or all; close all; every change reports whether it was applied and why not
Data granularity	bars	tick to daily, many asset classes	tick, quote, trade, bar and order book, nanosecond timestamps	OHLCV bars of any aggregation (time, tick, volume, dollar), many tickers on one union timeline, attached non-price series, memory-mapped files larger than RAM
Default fill timing on bars (measured, §3.4)	next bar's open	current bar	current bar	next bar, first tick of a seeded intrabar path

LEAN and NautilusTrader have more native order types, more time-in-force options and tick and order-book data. Reamer Research is designed for mid-frequency research on bars, and it keeps the native set small. The other behaviours are written from these parts:

- **Time in force.** Every order fills in full; fills are not limited by bar volume. So FOK behaves as IOC does. DAY is GTD with the expiry set to the session close.
- **Trailing stops.** A position-bracket change, applied once per bar.
- **Combo and contingent orders** (OCO, OTO, multi-leg). A few lines of strategy code that cancel, modify or submit orders from `on_bar`. These act at bar granularity: two orders that both trigger inside one bar both fill before the strategy can react, and the legs of a multi-leg order fill independently. Exits are an exception: take profit and stop loss are native and act as one-cancels-other at tick level.
- **Tick and order-book data.** Not used. High-quality tick and order-book history is large and expensive, and mid-frequency strategies seldom use it directly. Reamer Research instead takes OHLCV bars, time or activity (tick, volume, dollar), and replays a seeded intrabar tick path for each bar. The path has as many ticks as the bar's `tick_count`, or one per second of the bar when `tick_count` is not given [11]. The path is synthetic: it does not model queue position or book depth.

What the engine also offers is measured in this paper: bit-identical ledgers (§8), published fill rules [11], a seeded intrabar path (§4.4) and throughput (§4–§7). Neither strategy in this paper uses the order-management features; they were not benchmarked.

9.2 Customisation

- **Backtrader:** open source, all Python, easy to subclass. It has had no release since 2023, so any change is the user's to maintain.
- **LEAN:** open source. Every model can be replaced: fill, slippage, fee, brokerage, buying power, universe selection and data. Deep changes need C# and a .NET build.
- **NautilusTrader:** open source, built from components (actors, strategies, execution algorithms, adapters, fill and fee models). Core changes need Rust.

- **Reamer Research:** closed source. The extension surface is a stable C ABI. A strategy can be written in any language with a C foreign-function interface; this paper uses C++ and Python. Configuration covers capital, leverage, commission (round trip, open only or close only), slippage, spread, intrabar volatility, price basis (bid, ask or midpoint), overnight swaps by weekday and futures roll rebasing, each overridable per ticker, plus exogenous series per ticker, Monte Carlo resampling and memory-mapped bar files. The engine's internals cannot be changed.

9.3 Path to live trading

- **Backtrader:** integrations for Interactive Brokers, Oanda and Visual Chart, built on client libraries that are themselves unmaintained.
- **LEAN:** live trading through QuantConnect's cloud live nodes or the LEAN CLI, with QuantConnect's brokerage integrations.
- **NautilusTrader:** built-in adapters for Interactive Brokers, several crypto venues, Betfair and others. Backtest and live code paths are the same.
- **Reamer Research:** the strategy sends the same order requests through a relay to Reamer Server, which holds order state, runs the user's pre-trade checks, and routes through a broker connector that the user writes. No broker connector is included. The orders are the same in backtest and live, but the user writes the connector and the deployment.

9.4 Maintenance

Engine	Latest release at 2026-10-25
Backtrader	1.9.78.123, 2023-04-19
LEAN	rolling master; f3d841e on 2026-10-21
NautilusTrader	1.231.0, 2026-08-02; 2.0 release candidates on pre-release
Reamer Research	4.6.0, 2026-10-25

The NautilusTrader 2.0 release candidates were not tested (§10).

9.5 Licence and cost

	Backtrader	LEAN	NautilusTrader	Reamer Research
Licence	GPL-3.0	Apache-2.0	LGPL-3.0	proprietary, per seat
Self-hosted licence cost	none	none for the engine; the LEAN CLI's local coding needs a paid QuantConnect plan	none	US$20 per seat per month, cancel any time
Hosted offer	none	QuantConnect cloud: from US$888 per user per year	"Pro" and "Cloud" offers without public prices	none
Source available	yes	yes	yes	no

QuantConnect cloud prices four plans per user [9]. The pricing page gives a recommended setup for each plan:

Plan	Minimum users	US$ per user per month, billed monthly	US$ per user per year, billed annually
Researcher	1	84	888
Team	2	168	1,728
Trading Firm	2	480	4,992
Institution	5	1,272	12,912

The plans differ in how many compute nodes can be added: up to 2 (Researcher), up to 10 (Team) or unlimited (Trading Firm and Institution). Only Institution includes on-premises deployment. We did not measure QuantConnect's cloud, so we give no cost per backtest for it.

Cost per 1,000 backtests on the test machine. The machine was rented at €2.83 per hour. At each engine's best measured backtests per hour (Table 7), the hardware cost of 1,000 runs of the 500K-bar realistic backtest is 2.83 × 1,000 ÷ (backtests per hour):

Engine	Backtests per hour (K)	€ per 1,000 backtests
Backtrader	11,125 (64)	0.25
LEAN	1,293 (8)	2.19
NautilusTrader (patched)	11,242 (64)	0.25
Reamer Research, Python	42,468 (64)	0.067
Reamer Research, C++	332,249 (64)	0.0085

LEAN's figure is at K = 8, the largest K at which it completed (§7). The table counts hardware only. It excludes the researcher's time, data and licences. A Reamer Research seat adds US$20 a month (US$240 a year); spread over the hours it is used, that is, for example, US$0.12 per hour at 2,000 hours a year. Readers can apply their own utilisation.

This arithmetic is not a verdict. Three of the engines cost nothing to license, and for many users the order model, data and live path of §9.1 to §9.3 decide more than hardware cost does.

9.6 Setup effort observed

- **Backtrader and NautilusTrader:** one `pip install` each.
- **LEAN from source:** the .NET 10 runtime; a Python 3.11 environment, because Python.NET loads `libpython3.11.so`; OpenSSL 1.1 compatibility libraries on current Ubuntu; a custom data class; timestamps moved to 2000 because of the end-date limit (§3.9); the time-zone and end-date settings found during the run (§3.11).
- **Reamer Research:** extract the kit archive, link the library, and activate a licence (here, a test licence).

10. Threats to validity

- **One machine, one date, one instrument.** Every figure comes from one AMD EPYC 9575F on 2026-10-25, and from one GBP/USD series. The larger tiers repeat that series (§3.2). A different CPU, memory system or operating system can change the ratios.
- **Two strategy shapes.** A strategy with heavy indicators, many instruments per step or a model in the loop spends more of its time in user code. Engine differences then matter less, and the ranking can change.
- **Defaults.** Every engine runs at its defaults plus the listed changes (§3.4). Tuning, for example NautilusTrader's configuration or LEAN's data handling, could change the results.

We invite replications with tuned configurations, and the runners are published for that purpose.

- **Scope cuts.** The other engines were not run above 500K bars, LEAN was not run at 100 tickers or above K = 8, and Reamer Research's breadth runner is Python only (§3.6).
- **Versions.** LEAN runs Python 3.11 through Python.NET; the other Python engines run Python 3.13. NautilusTrader 2.0, whose maintainers state that it records PnL without the quadratic cost of §4.3 [8], was not tested.
- **One Reamer Research change.** The Python results use 4.6.0 with one binding fix that ships in 4.6.1 (§3.11).
- **The author's interest.** The author builds and sells Reamer Research. The mitigations are listed in the conflict-of-interest statement: open runners and raw data, outcomes stated in advance, and both stock and patched results for the one patched engine.

11. Conclusion

What does one backtest cost? At 500K bars of the realistic strategy, process wall time ranged from 0.36 s (Reamer Research, C++) through 3.4 s (Reamer Research, Python) to 13.6–22.1 s for the other engines, with NautilusTrader at 76.9 s when stock. Peak memory ranged from 185 MB to 1.1 GB.

How does it grow with bars? Linearly, for every engine except stock NautilusTrader, whose `record_trade` cost grows with the square of the trade count. LEAN pays the largest fixed cost, 3.3–3.9 s per start.

How does it grow with tickers? Close to linearly for every engine. On the realistic strategy, NautilusTrader 1.231.0 pays an extra multi-instrument cost in its position snapshot cache.

How does it grow with parallel copies? Every engine that completed gained throughput up to one copy per physical core. Efficiency fell between K = 32 and K = 64 for all of them, most for Reamer Research's C++ path. LEAN stalled at K = 32.

Is it reproducible? Every completed run of every engine repeated its result. Reamer Research's full ledger is bit-identical across 20 runs on one build.

Where each engine fits.

- **Backtrader:** pure Python, easy to read and change, with the lowest memory of the Python engines on long single-ticker runs; unmaintained since 2023.
- **LEAN:** the broadest asset, data and order model, and a hosted path to live trading; the highest start-up cost, and start-up hangs under concurrency on this machine.
- **NautilusTrader:** a rich order and data model and built-in live adapters with one code path; two costs, measured here, that grow with trade count and instrument count in 1.231.0.
- **Reamer Research:** the highest throughput and lowest memory per run in this study, and a bit-identical ledger; a small native order set that strategies extend in code, bars with simulated intrabar ticks rather than tick or order-book data, closed source and paid.

Declarations

Data availability. The raw results, the runner scripts, the dataset builders and the script that rebuilds every table are openly available under CC BY 4.0 at `https://github.com/j0osman/research-engine-benchmark` [14]. No Reamer Research binary is in the deposit.

Competing interests. The author is the founder of Reamer Labs and has a financial interest

in the software measured. The measurement method, the caveats, and the raw artifacts are stated so that a reader can verify the findings independently.

Funding. This work received no external funding.

Author contributions. C. Palghadmal designed the study, built the benchmark harness, ran the measurements, and wrote the paper.

References

[1] TildAlice. Backtrader vs Zipline: 5-year S&P 500 speed and memory test. Blog post, 2026-03-09. https://tildalice.io/backtrader-zipline-5-year-sp500-speed-memory-test/. Accessed 2026-10-26.

[2] AlgoLab. Python backtesting library comparison 2026 (in Korean). Blog post, 2026-08-01. https://algolab.co.kr/blog/python-backtesting-library-comparison-2026. Accessed 2026-10-26.

[3] Rodriguez, D. (mementum). Backtrader, version 1.9.78.123. https://github.com/mementum/backtrader. Accessed 2026-10-25.

[4] QuantConnect. LEAN algorithmic trading engine, commit f3d841e. https://github.com/QuantConnect/Lean. Accessed 2026-10-25.

[5] Nautech Systems. NautilusTrader, version 1.231.0. https://github.com/nautechsystems/nautilus_trader. Accessed 2026-10-25.

[6] Nautech Systems. NautilusTrader documentation: Orders. https://nautilustrader.io/docs/latest/concepts/orders/. Accessed 2026-10-25.

[7] Nautech Systems. NautilusTrader documentation: Installation. https://nautilustrader.io/docs/latest/getting_started/installation/. Accessed 2026-10-25.

[8] NautilusTrader issue #4434. PortfolioAnalyzer._append_pnl_record: pd.concat per closed trade makes backtests O(n^2) in trade count. Opened 2026-07-10, closed as not planned 2026-07-13. https://github.com/nautechsystems/nautilus_trader/issues/4434.

[9] QuantConnect. Pricing. https://www.quantconnect.com/pricing/. Accessed 2026-10-26.

[10] QuantConnect. Documentation: Order types. https://www.quantconnect.com/docs/v2/writing-algorithms/trading-and-orders/order-types. Accessed 2026-10-25.

[11] Reamer Labs. Reamer Research execution specification. https://reamerlabs.com/docs/execution-spec. Accessed 2026-10-26.

[12] Palghadmal, C. Predictable Trading Infrastructure on Many-Core CPUs. Preprint, version 1.0, 2026-09-16. https://reamerlabs.com/papers/predictable-trading-infrastructure.pdf.

[13] Palghadmal, C. Does the Operating Envelope Survive an Upgrade? A Replication of Many-Core Trading-Infrastructure Measurements Across a Release, Kernel and Compiler Change. Preprint, 2026-10-26. https://reamerlabs.com/papers/operating-envelope-replication.pdf.

[14] Palghadmal, C. Raw artifacts for "What a Backtest Costs": LEAN, NautilusTrader, Backtrader and Reamer Research on an AMD EPYC 9575F. Dataset, Reamer Labs, 2026. https://github.com/j0osman/research-engine-benchmark.

Preprint. The figures come from a single test machine on a single date. The author reports them with their caveats. Reproduce on target hardware before you rely on any specific number. *© 2026 Chaitanya Palghadmal / Reamer Labs. Licensed under CC BY 4.0. Reamer Labs, Reamer Research and Reamer Server are names of Reamer Labs; other product names belong to their owners.*