Does the Operating Envelope Survive an Upgrade?

A Replication of Many-Core Trading-Infrastructure Measurements Across a Release, Kernel and Compiler Change

Chaitanya Palghadmal*

Founder and Principal Systems Architect, Reamer Labs

Preprint • Version 1.1 • 2026-10-26

Abstract

This paper replicates the measurements of *Predictable Trading Infrastructure on Many-Core CPUs* [1], made on an AMD EPYC 9575F on 2026-09-13, on the same CPU model six weeks later. Three things changed together in between: the release (4.2.1 to 4.6.0), the operating system (Ubuntu 24.04 to 26.04.1) and the compiler (GCC 13.3 to 15.2.0). We state five of the earlier findings as claims that the new run could refute.

All five held. The throughput plateau again extends to about one strategy per physical core: its peak reproduced within 0.6% and its level within 2% in two independent sweeps. Pinned run-to-run spread stayed within 1.03×. The ring-size curve reproduced within 3% up to the 256 MiB L3, with the product default again at its peak. The research engine's full closed-trade ledger was bit-identical over 20 runs. The end-to-end order path was 1.3–1.6 µs (11–15%) faster at the median and lower at every percentile through P99.9. The ABI boundary read 80 ns against 73 ns raw, a difference below the machine's 10 ns effective timer resolution. The research engine ran 6.2% more bars per second.

Two results moved, both outside the operating envelope. Between 97 and 128 strategies, beyond one thread per physical core, throughput was 15–19% higher than before, with a worse tail. Above the L3 size, large rings cost 10–11% more. Because the three changes coincide, we attribute every delta to the combined change. New in this run are a concurrency sweep in steps of one strategy, a ring-size sweep with hardware counters, client pinning, and the measured effect of 4.6.0's position-state fix on a strategy that reverses.

Keywords: replication; operating envelope; trading infrastructure; many-core CPU; latency tail; concurrency scaling; CPU affinity; ring buffer sizing; determinism; reproducibility.

Classification: JEL C63 (Computational Techniques), C88 (Other Computer Software), G17 (Financial Forecasting and Simulation); ACM CCS Computer systems organization → Multicore architectures, Software and its engineering → Software performance.

1. Introduction

Why replicate. An operating-envelope claim tells an operator how many strategies a machine can carry before latency degrades, and how to place and size the parts. Such a claim is useful only if it survives routine change: a new release, a new kernel, a new compiler. Most vendor benchmarks are run once and never repeated. This paper repeats one.

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

The question. The earlier paper [1] measured Reamer Labs' trading infrastructure on an AMD EPYC 9575F on 2026-09-13. Six weeks later, three things had changed together: the release (4.2.1 to 4.6.0), the operating system (Ubuntu 24.04 to 26.04.1) and the compiler (GCC 13.3 to 15.2.0). We ran the same measurements again on the same CPU model. The question is which of the earlier findings are properties of the architecture, and which belonged to that build and environment.

The findings under test. We state each of v1's findings as a claim that the new run can refute:

- **H1:** the clean-latency region and the throughput plateau extend to about one strategy per physical core, that is, about 64.
- **H2:** thread placement, not the median, governs run-to-run variance, and pinning recovers it.
- **H3:** a ring buffer does best when it is sized to reader lag and kept in cache.
- **H4:** the research engine is exactly reproducible, even with seeded randomness.
- **H5:** the ABI boundary costs about 53 ns net at the median; the order path takes 11.3 µs at the median and 12.4 µs at P99 end to end.

What is new. The run adds measurements that v1 did not make:

- the concurrency sweep in steps of one strategy from 1 to 128
- a ring-size sweep with hardware counters
- pinning of the client threads
- a hash of the full closed-trade ledger in place of a hash of printed output
- the effect of 4.6.0's position-state fix on a strategy that reverses

Definitions and the benchmark design are those of v1. We cite them rather than restate them. A companion paper measures the research engine against three other backtesting engines on the same machine on the same day [4]; this paper does not repeat that comparison.

2. Method

2.1 What changed between the two runs

Both runs used one rented AMD EPYC 9575F bare-metal server from the same provider and product line. The first run was on 2026-09-13 and is reported in [1]. This one was on 2026-10-25. Both servers had the same public address, but we cannot confirm they were the same physical machine, and we treat them as two units of one model.

	v1 (2026-09-13)	This run (2026-10-25)	Changed
CPU	EPYC 9575F, 64 C / 128 T, 1 NUMA node	same	no
Memory	755 GiB	755 GiB	no
Virtualisation	bare metal	bare metal (`systemd-detect-virt`: none)	no
OS	Ubuntu 24.04 LTS, kernel not recorded	Ubuntu 26.04.1 LTS, kernel 7.0.0-38-generic	**yes**
Compiler	GCC 13.3	GCC 15.2.0	**yes**
CMake	not recorded	4.2.3	unknown
Reamer Server build	CMake Release (`-O3 -DNDEBUG`), `-march=x86-64-v2`	same	no
Reamer Research build	`-O3 -funroll-loops -march=native`	same	no

	v1 (2026-09-13)	This run (2026-10-25)	Changed
CPU policy	`performance` governor, boost on, SMT on	same	no
Clocksource	not recorded	TSC (calibrated 3,300 MHz)	unknown
Release	4.2.1 (tag at 81948a5; commit not recorded, see below)	4.6.0, commit 68db07a	**yes**

v1's build and commit. v1's report labels the Server build "`-O2` Release". That label came from the product's documentation, not from the build. v1's runbook built with `cmake -DCMAKE_BUILD_TYPE=Release`, and the Server's CMakeLists at 4.2.1 is the same as at 4.6.0 in this respect: CMake's default Release flags for GCC (`-O3 -DNDEBUG`) and `-march=x86-64-v2`. Only the load benchmark, which neither paper runs, adds `-O2`. The optimisation level therefore did not change between the runs.

v1's provenance file does not record a commit. 4.2.1 is tagged at 81948a5. The three commits between the tag and the run touch only the reference broker, packaging and documentation, and one benchmark's licence loading. That last one, 3e86f20, was committed four minutes after v1's provenance capture and fixed a harness used in that run. None touches the Server or research engine sources. We describe v1's code as 4.2.1.

What changed in the code under test. Between the two runs there were 43 commits and seven releases (4.2.1 to 4.6.0):

- **Server sources:** 22 files changed (+690 / −99 lines). The Server ABI went from version 3 to 4, which added the order fields for relay brackets.
- **Research engine:** 28 files changed (+4,250 / −1,262 lines). The Research ABI went to version 7. This restored memory-mapped data loading, CSV ingest, Monte Carlo and the relay.
- **4.6.0 itself:**
  - The Python binding reads bars from the caller's NumPy array without re-parsing the dtype on every bar. This affects only Python-driven runs.
  - The research engine no longer keeps stale position state on a reversal. Before the fix, a cancel rule that was too broad also cancelled the order that opened the reverse position. Results change only for strategies that reverse.

Appendix A lists every release in between.

Confounding. Several things changed together:

- the code: seven releases
- the OS and kernel
- the compiler

No experiment here can tell their effects apart. When a result moves, we attribute the move to the combined change, not to any one part of it. Where the changelog makes one cause plausible, we say so, and we label that a hypothesis, not a finding.

2.2 How the run was executed

Payload and automation.

- One archive was copied to the server: sha256 `b31125d7...7a95`.
- It held the source at commit 68db07a, with an empty uncommitted diff, plus the benchmark

harness.
- Everything was built on the server.
- One script ran every step in an order fixed before the run, most important first, under a 5.5-hour budget. A step whose estimate exceeded the remaining budget was to be skipped and recorded. None of this paper's steps was skipped. (The budget cuts recorded in the dataset apply to the companion paper's competitor runs only.)
- For each step the script recorded:
  - start time, duration and exit status
  - counts of successful and failed runs
  - each benchmark's raw text output, unedited

Incidents, in order:

1. *First attempt, 10:27–10:35 UTC.* The harness stopped during the test suite. The cause was a shell option left on by a setup step, which turned the first failing test into a fatal error. Two tests had failed only because the system Python lacked `pytest`. We installed it, turned the option off, and restarted at 10:38:33. No benchmark had run, so nothing was lost or repeated. All 43 tests then passed.
2. *Disk resync.* The server's two NVMe drives form a software RAID 1 mirror, which was still on its initial resync after provisioning. The resync ended at 10:40:55. It overlapped the builds, the test suite, the boundary benchmark (§2.3, B1) and the end-to-end benchmark (B2) at 10:39:16–18, the whole stock doubling sweep of B3 (10:39:18–10:40:22), and the first half-minute of the step-by-one sweep. In the doubling sweep, the $N = 256$ point did not account for every order: 1,270,010 of 1,280,000 were matched, at 23,201 orders/s. The same point in the later tail run matched every order, at 71,530 orders/s. We repeated every overlapped benchmark after the main run, with the machine otherwise idle, and report both runs. They differ by no more than repeated runs on the idle machine differ from each other:
   - B1's boundary figures repeat exactly.
   - B2's percentiles move by under 1 µs, and the first run's 141 µs maximum did not recur.
   - The doubling sweep's $N = 256$ point accounted for every order. Results, §3–§4.
3. *Research benchmark licence (B7).* The B7 step reported success but ran none of its 2,000 runs. Each attempt stopped with `license.invalid`.
   - B7's binaries link the licensed research library. Every other research step ran against the kit's test library, which the harness licenses for the run.
   - The benchmark prints the error but exits 0, so the step's status read "ok".
   - We found the empty output at the pull check and reran B7 after the main run, pointing the same binaries at the test library. All 2,000 + 200 runs completed.
   - The original output is kept in the dataset.

2.3 Benchmarks

The labels below are this paper's. The harness step names are in the dataset.

B1. ABI boundary.

- 600,000 calls on one thread, pinned to core 0.
- Each call goes through the risk-gate check path of the stable C ABI: marshalling the order from C++ to the C struct, dispatch through the function table, and return.
- The benchmark also reports, from the same loop:

  - the cost of reading the timer twice with nothing between (*timer floor*)
  - a bare virtual call
  - marshalling alone
- *Net* figures subtract the timer floor's median.

B2. End-to-end order path without a network.

- One thread runs the Server; a second thread on another core submits orders over a Unix-domain socket and watches the shared-memory event bus for the acceptance event.
- The gate accepts everything and the broker is a stub. No Go, Python or FIX code is involved.
- Two runs of 50,000 orders each.
- Latency is from send to observed acceptance.

B3. Concurrent strategies.

- N client threads each connect as a separate strategy. Each sends 5,000 orders back to back, waiting on the event bus for each order's outcome before sending the next.
- The Server is pinned to core 0, and the clients are unpinned unless stated otherwise.
- Reported per N:
  - matched orders
  - aggregate orders per second
  - P50, P90, P99 and maximum latency
  - whether every order was accounted for
- The stock benchmark doubles N from 1 to 512, as in v1.
- To find exactly where behaviour changes, we added a run that steps N by 1 from 1 to 128, then 192, 256, 384 and 512. That run uses 128-strategy rows rather than 512 to stay within budget. Both send 5,000 orders per strategy, so every row expects $N \times 5{,}000$ orders, as in the stock binary.

B4. Repeatability. $N = 32$, run 36 times in a row. The quantity of interest is the spread of P50 and P99 across runs.

B5. Placement.

- $N = 8$, 32, 48 and 62, with 3 repetitions each.
- Each is run twice: once with clients unpinned, once with client i pinned to core $1 + (i \bmod 63)$. The pinned run therefore never shares the Server's core and never uses an SMT sibling.
- At $N = 62$ we also recorded, with `perf stat`: task clock, context switches, CPU migrations, cycles and instructions.
- The main run used the stock 65,536-slot ring. v1 ran B5 at the 4,096-slot product default, so after the main run we repeated the throughput part of B5 at 4,096 slots. That rerun did not repeat `perf stat`.

B6. Event-ring size.

- $N = 40$, 3 repetitions per size.
- Ring capacity 1,024, 2,048, 4,096, 8,192, 16,384, 65,536, 262,144 and 1,048,576 slots.
- Each slot is 4,112 bytes, so the sizes run from about 4 MiB to about 4 GiB.
- 4,096 slots (about 16 MiB) is the product default. 65,536 is what the stock benchmark allocates, and what v1 measured.

- At 4,096, 65,536 and 1,048,576 slots we recorded cycles, instructions, cache misses, dTLB loads and dTLB-load misses.
- We also stepped N by 1 from 24 to 48 at 65,536 and 1,048,576 slots, to see whether ring size moves where the knee falls.

B7. Research engine per-call cost.

- The research engine driven through its C ABI by a C++ strategy, on 884,130 bars, no wrapper.
- 2,000 runs of a buy-and-hold strategy, and 200 runs of the realistic breakout strategy from the companion paper.
- Each run reports mean time per bar call and bars per second. We report the distribution across runs.

B8. Determinism. On 500,000 bars of the realistic strategy:

- *Python:* 20 runs through the Python binding. Each hashes its complete closed-trade ledger (every field, keys sorted) with SHA-256.
- *C++:* 20 runs through the C++ runner, each recording fill count, round trips and net PnL.

The claim holds if every hash, and every C++ triple, is identical.

B9. Capacity under stress. The kit's stress suite (`reamer_stress_bench`), run once, as in v1.

- *What it loads.* CSV datasets that the kit's generator derives from the GBP/USD 15-minute series: a deep single ticker (about 1.2M and 4.7M rows), an out-of-order copy, a sparse copy, and 10- and 20-ticker wide sets. The scale cases generate synthetic bars in memory: n tickers over 5 to 40 years of daily bars, 10,000 tickers over 20 years (50M bars), and 10,000-ticker equity-style universes over 5 and 20 years with take-profit and stop-loss brackets.
- *What it covers.* CSV parsing, timeline and aligned builds, single-ticker depth, multi-ticker breadth, tick replay and snapshot restore, Monte Carlo, an end-to-end load-backtest-Monte-Carlo combination, and the scale and capacity cases.
- *What it reports.* For each case: elapsed time, items processed, throughput and resident-memory growth. It ends with a bottleneck summary that flags any subsystem below its threshold. The capacity cases have no threshold.

Changes to the benchmark harness. B3–B6 needed four settings that the stock concurrency benchmark hard-codes. We added them to a *copy* of the benchmark as environment variables; the stock binary was built and run unchanged:

- ring capacity (stock: 65,536 slots)
- first value of N (stock: 1)
- step mode: $N + 1$ instead of $2N$
- client pinning, with the core rule given under B5

Each setting replaces only the constant named. With none set, the copy's output format and order counts match the stock binary's. Throughput at a shared N agrees only to within the run-to-run spread of single unpinned points. For example, at $N = 8$ three runs gave 174K, 209K and 211K orders/s (§4).

2.4 Statistics

- **Latency percentiles.** Every latency percentile is order-statistic, with no interpolation: the sample at index $\lfloor p \cdot (n-1) \rfloor$ of the sorted samples.
- **No samples are removed.** Warm-up is part of the measurement and appears in the maxima.
- **Across repetitions** we report the median of each statistic, with the minimum and maximum beside it.
- **The v1 figures** in the side-by-side tables are taken from [1] and its dataset unchanged; we do not recompute them.
- Δ is this run minus v1, and also as a percentage of v1.
- **Thresholds for B1–B6.** We call a change *within noise* when it falls inside the range of v1's own repetitions, and *moved* otherwise. We set this threshold before comparing.

Timer resolution. Every B1 value in this run is a multiple of 10 ns. The timer floor's median is 20 ns, as in v1, but v1 also recorded values between the 10 ns steps. A probe after the main run measured the clock directly, pinned to core 0:

- `clock_getres(CLOCK_MONOTONIC)` reports 1 ns.
- Over 1,000,000 back-to-back reads, 99.99% of deltas were exactly 10 ns (132,382) or 20 ns (867,489). Values off the 10 ns grid occurred 86 times in total.

The effective resolution on this machine is therefore 10 ns, not the 1 ns the kernel reports. Differences below that resolution are not interpreted. This mainly affects the ABI boundary comparison, where the expected effect is a few nanoseconds.

2.5 Scope

These results describe the Server's own path, not a production deployment:

- The broker is a stub and the gate accepts everything. Real broker round trips and risk logic add latency on top.
- Strategies are threads in one process. Separate processes would add scheduling and connection costs not measured here.
- No network, TLS or FIX session is involved.
- The research figures cover bar-based backtests only.

The companion paper [4] measures the research engine against other backtesting engines; this paper does not repeat that.

2.6 Data availability

Published at https://reamerlabs.com/papers/data/epyc-9575f-4.6.0-raw-artifacts.zip:

- each benchmark's raw text output
- `perf stat` output
- per-step records
- the machine provenance file

No scripts or binaries are included. The v1 dataset is https://reamerlabs.com/papers/data/epyc-9575f-2026-09-raw-artifacts.zip.

3. The integration boundary (H5)

Table 1. ABI boundary and end-to-end order path, v1 vs 4.6.0. Run A overlapped the RAID resync. Run B is the rerun on the idle machine. Δ is run B minus v1.

Measure	v1 (4.2.1)	4.6.0 run A (resync)	4.6.0 run B (idle)	Δ (B)
Timer floor, P50	20 ns	20 ns	20 ns	0
Gate check, raw P50	73 ns	80 ns	80 ns	+7 ns, below resolution
Gate check, net P50	53 ns	60 ns	60 ns	+7 ns, below resolution
Gate check, raw mean	not recorded	79.2 ns	79.9 ns	
Gate check, raw P99	not recorded	110 ns	110 ns	
End-to-end P50, run 1 / run 2	11.30 / 11.59 µs	10.39 / 10.55 µs	9.66 / 10.33 µs	−1.64 / −1.26 µs
End-to-end P90	11.63 / 11.99 µs	10.87 / 10.97 µs	10.06 / 10.76 µs	−1.57 / −1.23 µs
End-to-end P99	12.40 / 12.52 µs	12.16 / 12.08 µs	11.00 / 11.78 µs	−1.40 / −0.74 µs
End-to-end P99.9	13.93 / 13.47 µs	14.09 / 13.32 µs	12.13 / 13.16 µs	−1.80 / −0.31 µs
End-to-end max	74.19 / 43.17 µs	140.84 / 36.19 µs	37.38 / 36.46 µs	

End to end.

- On the idle machine the median order path is 1.3–1.6 µs (11–15%) faster than in v1.
- Every percentile through P99.9 is lower than v1's, in both runs.
- The shape of the distribution is unchanged: P99 is 1.14× the median in both runs (v1: 1.08–1.10×), and P99.9 is 1.26–1.27×.
- The resync run's maximum of 141 µs did not recur. Both idle runs peak at 36–37 µs, below v1's 43–74 µs.
- Run A sits between v1 and run B at every percentile through P99. So the resync cost this benchmark up to about 1 µs, without changing the conclusion.
- Two runs back to back differ by 0.7 µs at the median, much as v1's two runs differed by 0.3 µs.

Boundary. The gate check reads 80 ns at the median in both runs, against v1's 73 ns. We do not report that as a change:

- The clock probe (method §2.4) shows an effective timer resolution of 10 ns on this machine: back-to-back reads differ by exactly 10 or 20 ns in 99.99% of a million samples, although `clock_getres` reports 1 ns.
- A 7 ns difference in a median is below that resolution. A true 73 ns measured on a 10 ns grid reads as 70 or 80.
- The two runs repeat exactly: P50 80 ns and P99 110 ns in each, with means of 79.2 and 79.9 ns.
- The mean of many grid-quantised samples estimates the true cost better than the median does, but v1 kept only the medians, so the means cannot be compared. A change of a few nanoseconds in either direction cannot be ruled out without a finer timer.

H5: held for the end-to-end path, which is faster at every percentile. The boundary is **held within resolution**: unchanged at 10 ns resolution, and a single-digit nanosecond change cannot be ruled out.

4. Concurrency (H1)

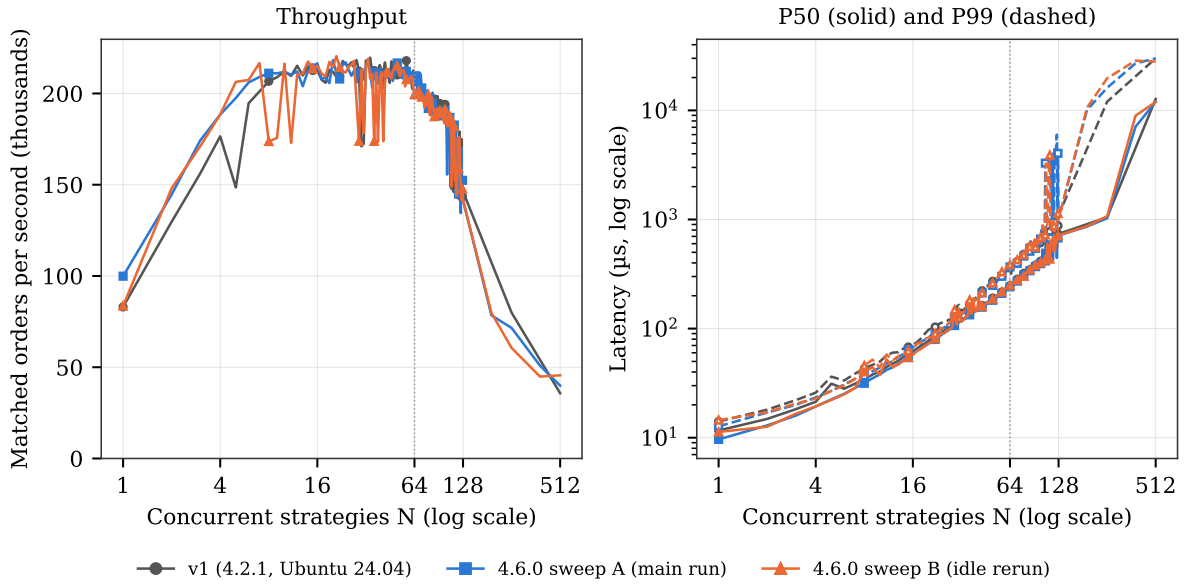


Figure 1. Aggregate matched orders per second (left) and P50 and P99 latency (right) against concurrent strategies N: v1's 130 points and the two 4.6.0 sweeps. The dotted line marks N = 64, one strategy per physical core.

Two 4.6.0 sweeps. The step-by-one sweep ran twice on 4.6.0:

- Sweep A ran in the main run. Its first half-minute overlapped the end of the RAID resync.
- Sweep B ran on the idle machine after the main run.

v1 has one sweep. Each sweep measures each N once, so comparing A with B shows how far a single point moves between runs. The median point moves 1.9%; a few points at low N move by up to 27%.

Table 2. Selected points, v1 vs 4.6.0 (orders/s; latency from sweep B)

N	v1	4.6.0 A	4.6.0 B	Δ A / Δ B	v1 P50 / P99 (µs)	4.6.0 B P50 / P99 (µs)
1	83,035	99,886	83,546	+20.3% / +0.6%	11.6 / 14.1	11.3 / 14.5
8	206,685	211,087	173,667	+2.1% / −16.0%	34.2 / 43.1	40.1 / 45.9
32	218,116	214,286	214,116	−1.8% / −1.8%	122.0 / 144.9	115.5 / 134.5
48	209,059	209,614	213,191	+0.3% / +2.0%	180.8 / 260.3	175.7 / 248.7
64	206,468	209,707	199,285	+1.6% / −3.5%	248.0 / 359.4	249.4 / 385.4
68	200,319	205,402	203,489	+2.5% / +1.6%	277.2 / 389.1	262.7 / 423.7
96	190,236	188,188	186,406	−1.1% / −2.0%	397.0 / 553.7	388.8 / 573.5
112	145,968	183,603	178,416	+25.8% / +22.2%	645.2 / 755.7	442.1 / 3,696.0
128	146,260	141,615	141,544	−3.2% / −3.2%	737.4 / 1,093.9	714.8 / 994.9
256	79,896	71,530	60,568	−10.5% / −24.2%	1,047 / 12,006	1,071 / 19,665
512	35,747	39,906	45,576	+11.6% / +27.5%	12,723 / 29,985	11,957 / 28,012

At $N = 1$ and 8, a third measurement exists: the stock doubling sweep's idle-machine rerun gave 104,476 and 208,995 orders/s. So three runs at $N = 8$ span 174K–211K. Single unpinned points at low N are not comparable to the percent. That is H2's finding, seen here (§5).

The plateau held.

Measure	v1	4.6.0 A	4.6.0 B
Peak	219,652 at $N = 34$	218,294 at $N = 20$	220,446 at $N = 21$
N within 5% of peak	9–69	7–70	7–61
N within 10% of peak	7–89	5–86	5–78
Median, $N = 32$–67	211,668	209,890 (−0.8%)	209,151 (−1.2%)
Median, $N = 68$–96	195,639	194,704 (−0.5%)	191,862 (−1.9%)

- The peak reproduces within 0.6% in both sweeps.
- The plateau's level reproduces within 2%.
- The band edges move by up to 9 strategies between the two 4.6.0 sweeps (70 against 61). With one run per point, a band edge is not resolvable more finely than that. v1's band, 9–69, lies within the range the two sweeps span.
- The plateau still extends to about one strategy per physical core.

Latency inside the envelope held.

- Through $N = 96$, P99 stays below 2× P50 in both sweeps, at most 1.90×

  (A) and 1.81× (B). The median ratio is 1.37–1.40×.

- In sweep B, P50 latency is within 4% of sweep A at every plateau point checked ($N =$ 32–96, in steps of 8).

What moved: 97–128.

- Between 97 and 128 strategies every run is noisy, with one run per point and dips of 15–20% between neighbours.
- Both 4.6.0 sweeps hold higher throughput than v1 here:

Run	Median, $N = 97$–128	First N below 170K/s
v1	152,501	103
4.6.0 A	180,925 (+18.6%)	102
4.6.0 B	175,743 (+15.2%)	109

- That two independent sweeps both show it makes it more than one run's noise. But without repetitions at each point, we do not quantify it more closely than "15–19%".
- In the same range, 4.6.0's P99 first exceeds 2× P50 at $N = 100$ (A) and 102 (B), and 3× at 106 and 107. v1's never did below 129. So 4.6.0 completes more orders in this region with a worse tail.

If the difference is real, the OS change is the likely place to look: this is the region beyond one thread per physical core, where the scheduler decides the outcome. This is a hypothesis.

Beyond 128.

- Every order was accounted for at every point from 192 to 512, in both sweeps. The harness raises the file-descriptor limit to 1,048,576 before the sweep. v1 needed a manual `ulimit` raise at 512.
- At 512 strategies, 2,560,000 orders exceed the Server's default `max_tracked_orders` of 2,000,000. The Server logs `order_tracking.evicted` warnings for the oldest orders but still matches every order.

- Throughput here moves by 20–30% between runs:

N	v1	Three 4.6.0 runs
256	79,896	60,568–81,543
512	35,747	39,906–57,512

  These points are deep in oversubscription, two to four threads per hardware thread. We read the 192–512 range as "same order of magnitude, same shape", not as a comparison to the percent.

- The stock doubling sweep's 256 point, taken during the resync, matched only 1,270,010 of 1,280,000 orders, at 23,201 orders/s. On the idle machine the same point matched all 1,280,000, at 81,543 orders/s. We attribute the shortfall to the resync window, without proof. No idle run dropped an order.

H1: held. The plateau's level reproduces within 2% of v1 in two sweeps. The clean-latency region extends to about one strategy per physical core, as in v1.

5. Placement and variance (H2)

Table 3. Run-to-run variance, unpinned, $N = 32 \times 36$ runs (same stock 65,536-slot ring as v1)

	v1	4.6.0
Mean orders/s	233,327	207,381
Min – max	199,949 – 265,052	195,798 – 212,910
Std. dev.	24,812	3,930
CV	10.6%	1.9%
Max / min	1.33×	1.09×
Shape	bimodal: 22 runs near 252K, 14 near 204K	unimodal around 207K
P50 latency, median (range)	127.0 µs (116.6–155.2)	120.0 µs (116.1–123.5)
P99 latency, median (range)	142.1 µs (123.5–179.8)	134.6 µs (126.1–141.7)

This is the largest change between the two runs, and it changes v1's argument.

- In v1 a single unpinned run was a coin-flip between two bands about 1.24× apart.
- In 4.6.0 all 36 runs fall in one band, with a coefficient of variation of 1.9%.
- That band sits where v1's *slow* band was (204K). The fast band did not appear at all.

So the unpinned server is now about five times more repeatable than in v1, and its mean is 11% lower than v1's, which averaged both bands.

Table 4. Client pinning at the 4,096-slot ring, as in v1. $N = 8$ / 32 / 48 / 62 $\times$ 3 runs, orders/s. Run on the idle machine after the main run.

N	Unpinned, 3 runs	Spread	Pinned, 3 runs	Spread	Δ median	v1 Δ median
8	236,843 / 226,934 / 228,622	1.04×	285,950 / 283,404 / 284,851	1.01×	+24.6%	+22.9%
32	263,007 / 207,251 / 270,767	1.31×	272,756 / 276,324 / 272,090	1.02×	+3.7%	+5.4%
48	256,723 / 258,640 / 261,104	1.02×	267,185 / 265,298 / 266,950	1.01×	+3.2%	−0.8%
62	193,299 / 191,713 / 254,308	1.33×	256,314 / 253,362 / 260,130	1.03×	+32.6%	+25.1%

Pinned medians, v1 vs 4.6.0 (orders/s): $N = 8$: 262,464 → 284,851 (+8.5%); 32: 266,701

→ 272,756 (+2.3%); 48: 257,048 → 266,950 (+3.9%); 62: 250,238 → 256,314 (+2.4%). Pinned, 4.6.0 is faster than v1 at every N.

Pinning's effect replicates at v1's setting.

- Pinned spread is 1.01–1.03× at every N, against 1.00–1.03× in v1.
- The gain from pinning at each N is within 7 points of v1's.
- Unpinned runs still fall into slow placements. At $N = 32$, one run of three ran at 207K against 263–271K for the others. At $N = 62$, two of three ran at 192–193K against 254K, the same two-of-three pattern as v1.
- Pinning raises the median most where a slow run is the median ($N = 62$) or where the clients are few ($N = 8$). At 32 and 48 it adds 3–4%.

So the single unpinned $N = 32$ configuration of Table 3 is tight, but the slow placements have not gone: they appear at other N, and at this ring size.

The same study at the 65,536-slot ring. The main run measured placement at the stock benchmark's ring, before the rerun above:

N	Unpinned, 3 runs	Spread	Pinned, 3 runs	Spread	Δ median
8	210,239 / 206,484 / 209,030	1.02×	264,263 / 261,914 / 257,655	1.03×	+25.3%
32	210,095 / 174,881 / 208,093	1.20×	232,161 / 232,059 / 225,780	1.03×	+11.5%
48	221,170 / 208,654 / 209,543	1.06×	212,551 / 209,443 / 211,330	1.02×	+0.9%
62	170,001 / 162,312 / 164,261	1.05×	206,280 / 205,209 / 205,018	1.01×	+24.9%

The same pattern holds: pinned spread is at most 1.03×, and slow unpinned placements occur at $N = 32$ and 62. Here all three unpinned runs at 62 are slow. Every figure is lower than at 4,096 slots, which is H3's result (§6).

Table 5. `perf stat` at $N = 62$ (4.6.0 at the 65,536-slot ring; v1 at 4,096)

Counter	v1 unpinned	v1 pinned	4.6.0 unpinned	4.6.0 pinned
CPU migrations	118	63	125	63
Context switches	2,712	1,871	20,348	16,602
IPC	1.53	1.51	1.36	1.39

Two readings:

- **Migrations replicate:** pinning halves them, from 118 to 63 in v1 and from 125 to 63 now.
- **Context switches are 7.5–9× higher** than in v1, pinned or not, and IPC is lower. The rings differ, so part of the IPC difference is the ring (§6). The rerun at 4,096 slots did not repeat `perf stat`, and the machine no longer exists. We therefore cannot say whether the context-switch count comes from the ring, the OS or the code. We report it and leave it unattributed.

H2: held. Placement still governs variance, and pinning still collapses it, at v1's ring size and at the benchmark's.

- The 36-run variance test at $N = 32$ is much tighter than in v1.
- But slow unpinned placements still occur at other N. At $N = 62$, two of three unpinned runs were slow, as in v1.

So the practical advice stands: pin the clients when repeatability matters. What changed is how often an unpinned run is slow in one configuration, not whether it can be.

6. Buffer sizing (H3)

Table 6. Ring capacity at $N = 40$, median of 3

Capacity (slots)	Footprint	v1 orders/s	4.6.0 orders/s	Δ	4.6.0 vs 65,536
1,024	~4 MiB	247,383	249,328	+0.8%	+19.7%
2,048	~8 MiB	256,190	253,263	−1.1%	+21.6%
4,096 (default)	**~16 MiB**	255,195	**257,311**	+0.8%	**+23.5%**
8,192	~32 MiB	250,843	246,399	−1.8%	+18.3%
16,384	~64 MiB	239,716	246,608	+2.9%	+18.4%
65,536	~256 MiB	205,161	208,267	+1.5%	0
262,144	~1 GiB	163,536	146,707	−10.3%	−29.6%
1,048,576	~4 GiB	157,852	140,367	−11.1%	−32.6%

(v1 also measured 32,768 and 131,072 slots; this run did not.)

The curve replicated, and the cliff is steeper.

- Up to 256 MiB every point is within 3% of v1.
- The product default is again at the top of the curve, 23.5% above the benchmark's 65,536-slot setting (v1: 24%).
- Above the 256 MiB L3, 4.6.0 is 10–11% below v1.
- Across the 24–48 strategy window, the 4 GiB ring is a median 28.8% slower than the 256 MiB ring, against 22.4% in v1.

Table 7. `perf stat` at $N = 40$

Counter	16 MiB v1	16 MiB now	256 MiB v1	256 MiB now	4 GiB v1	4 GiB now
Instructions	242.7 B	234.8 B	242.1 B	242.5 B	243.9 B	264.7 B
IPC	1.53	1.44	1.40	1.40	1.19	1.29
Cache misses	57.7 M	72.7 M	98.1 M	99.1 M	190.7 M	183.8 M
dTLB-load-miss rate	18%	15%	44%	31%	43%	27%
System time	not recorded	0.94 s	not recorded	3.83 s	not recorded	12.91 s
Wall	0.96 s	0.98 s	1.17 s	1.17 s	2.34 s	2.56 s

The cache-miss growth from 16 MiB to 4 GiB replicates (2.5× now, 3.3× in v1). What differs is where the extra time goes at 4 GiB:

- 13% more instructions, where v1's instruction count was flat.
- System time 14× the 16 MiB figure.

That pattern fits the kernel spending longer faulting in and zeroing the 4 GiB shared-memory segment. Neither run recorded page faults or the transparent-huge-page setting, so this is a hypothesis. Either way, it points to the environment, not to a change in the ring's access pattern.

H3: held. Size the ring to reader lag and keep it in cache. The penalty for ignoring this advice grew on this OS.

7. Research engine (H4, and what changed)

Per-call cost (B7). Run on the idle machine after the main run (method §2.2).

Table 7b. Research engine through the C ABI, 884,130 bars per run

Measure	v1, buy and hold (2,000 runs)	4.6.0, buy and hold (2,000 runs)	4.6.0, realistic (200 runs)
Per-call P50	0.58 µs	0.55 µs	0.77 µs
Per-call P99	0.59 µs	0.56 µs	0.80 µs
Per-call max	0.60 µs	0.56 µs	0.84 µs
Bars/s, mean across runs	1,721,199	1,828,578 (+6.2%)	1,288,699
Bars/s, aggregate	1,722,628	1,830,134 (+6.2%)	1,289,333
Bars/s, min – max	1,672,210 – 1,739,181	1,780,426 – 1,864,362	1,182,051 – 1,313,341
CV across runs	0.59%	0.66%	1.20% (1.05% without run 1)
Max / min	1.04×	1.05×	1.11× (1.05× without run 1)
Trades per run			40,091, every run

- The per-call cost of the buy-and-hold strategy is 0.03 µs lower than in v1 at every percentile, and throughput is 6.2% higher. The slowest 4.6.0 run is faster than the fastest v1 run.
- Run-to-run variation is unchanged: CV 0.66% against 0.59%.
- The per-call figures are means over 884,130 calls, reported to 0.01 µs, so a 0.03 µs difference is three steps of the printed resolution.
- The realistic strategy, which v1 did not run, costs 0.22 µs more per bar than buy and hold. That is the strategy's own work: indicators, orders and fills.
- Its first run is the slowest (1.18M bars/s); without it the spread is 1.05×, as for buy and hold.

Determinism (B8).

- 20 of 20 Python runs produced the same SHA-256 of the full closed-trade ledger, `b69821c6cb69...`, covering 11,818 closed trades.
- 20 of 20 C++ runs produced the same net PnL, −1,937.15947.
- The two paths agree with each other to five decimal places.

v1's test hashed the standard output of a quickstart script. This run hashes every field of every trade, so it is a stricter test, passed.

H4: held, on a stricter test.

The reversal fix. We ran the benchmark's realistic strategy (a 20-bar breakout that reverses between +1,000 and −1,000) at 50K bars on 4.5.1 and on 4.6.0, on the laptop. Each version used its own library and Python binding; the strategy and data were identical.

	4.5.1	4.6.0
Closed trades	1,178	1,170
Net PnL	−223.02	−219.33
Trades identical in times and prices	1,165	1,165
Trades that differ	13	5
PnL of the differing trades	−15.12	−11.42

The whole PnL difference, 3.70, comes from the differing trades. They fall in three episodes:

- **The first entry (2024-01-03, 05:45–12:30).** 4.6.0 buys 1,000 and holds it until the reversal at 12:30. 4.5.1 builds the position to 2,000, sells it back to flat over the next bars, and repeats that cycle four times before reversing from a 2,000 long. That is 9 trades in place of 1, at −6.21 against −1.49. This is the pattern the stale position state produces: on the bar after a fill, the strategy saw the position from before the fill and ordered again.
- **2024-03-20 and 2024-04-11.** In both, a short is closed and reversed to long within a few bars. 4.5.1 reverses one bar later than 4.6.0. The result is better on one date and worse on the other (−3.15 against −4.96 on 03-20; −5.76 against −4.97 on 04-11), so the fix does not favour the strategy's PnL.

The strategy is written to the documented contract that `position()` is the state after the current bar's fills. 4.5.1 broke that contract; 4.6.0 keeps it. Outside these three episodes, the two versions trade identically.

8. Capacity headroom

Table 8. Stress suite, v1 vs 4.6.0 (single run each, as in v1)

Case	v1	4.6.0	Δ
CSV parse, 4.7M rows → 884,130 bars	0.398 s	0.408 s	+2.5%
Monte Carlo streaming, 10^9 ops	0.094 s (+61 MB)	0.074 s (+5 MB)	−21%
End-to-end combo, 5 × 1.2M	0.562 s	0.072 s	−87%
Scale backtest, $n = 500$, largest	0.744 s	0.614 s	−17%
10,000 tickers × 20 years, 50M bars	16.65 s (+937 MB)	9.92 s (+931 MB)	−40%
US equity universe, 10k × 5 yr	7.19 s	3.74 s	−48%
US equity universe, 10k × 20 yr	29.11 s	15.80 s	−46%

- The suite again reports no bottleneck.
- Small cases are unchanged within a few percent, and CSV parsing is 2.5% slower.
- The large-universe cases, and the end-to-end combination, take half the time or less.

The code changed most in the research engine between the two runs. It restored memory-mapped loading, among other changes. That is the likely cause of the large-case gains. The compiler change is the alternative, and it cannot be excluded. Two runs that differ in both cannot separate the code from the compiler.

9. Scorecard

Hypothesis	Verdict	Evidence	Note
H1: the envelope ends at about one strategy per physical core	**Held**	Two sweeps: peak within 0.6% of v1, plateau median within 2%; 95%-of-peak band 7–70 and 7–61 vs 9–69	97–128 is 15–19% higher than v1 in both sweeps, with a worse tail
H2: placement governs variance, and pinning fixes it	**Held**	Pinned spread ≤1.03× at both ring sizes; slow unpinned runs at $N = 32$ and 62	The 36-run test at $N = 32$ is unimodal (CV 1.9% vs 10.6%)
H3: size the ring to reader lag and keep it in cache	**Held**	≤3% from v1 up to 256 MiB; default still at the peak	The beyond-L3 penalty is 10% larger
H4: exact reproducibility	**Held**	20/20 ledger hashes; C++ and Python agree	Stricter test than v1

Hypothesis	Verdict	Evidence	Note
H5: 53 ns boundary; 11.3 µs P50 and 12.4 µs P99 end to end	**Held**; boundary **held within resolution**	End to end on the idle machine: P50 −1.3 to −1.6 µs, P99 −0.7 to −1.4 µs; boundary +7 ns on a 10 ns timer grid	v1 kept only boundary medians
B7: 0.58 µs per call, 1.72M bars/s	**Held, faster**	0.55 µs; 1.83M bars/s; CV 0.66%	Realistic strategy 0.77 µs per bar

10. Threats to validity

- **Three changes at once.** The release, the operating system and the compiler changed together. No measurement here separates them. Every delta is a delta of the combined change; where we name a likely cause, we label it a hypothesis.
- **One machine, one date, per run.** Each run is one rented server on one day. The two servers had the same public address and the same model, but we cannot confirm they were one physical unit (§2.1). Unit-to-unit variation of the same model is therefore part of every delta.
- **The RAID resync.** The server's software RAID resync overlapped the start of the run. Every benchmark that it overlapped was run again on the idle machine, and both results are reported (§2.2).
- **One run per point.** The step-by-one concurrency sweep has one run per point, as v1's sweep did. Band edges and the 97–128 region are therefore resolved only as far as two independent sweeps allow (§4).
- **Timer resolution.** The effective timer step on this machine is 10 ns (§2.4). Boundary costs that differ by less than that are not resolved.
- **Measured bounds.** The bounds of v1 apply unchanged: a stub broker, an accept-all gate, threads standing in for processes, and no network, TLS or FIX session (§2.5).

11. Conclusion

Which findings are architectural. All five of v1's findings held after a new release, a new kernel and a new compiler on the same CPU model:

- **H1.** The throughput plateau still extends to about one strategy per physical core. Its peak reproduced within 0.6% and its level within 2%.
- **H2.** Pinning still controls run-to-run variance. Pinned spread stayed within 1.03×.
- **H3.** The ring-size curve reproduced within 3% up to the 256 MiB L3, and the product default is still at its peak.
- **H4.** The research engine's ledger was bit-identical over 20 runs, on a stricter test than v1's.
- **H5.** The order path is faster than in v1 at every percentile through P99.9. The boundary cost is unchanged within the timer's resolution.

What moved. Two results moved, both outside the operating envelope:

- Between 97 and 128 strategies, beyond one thread per physical core, throughput was 15–19% higher than v1's, with a worse tail.
- Above the L3 size, large rings cost 10–11% more than in v1.

Neither changes the advice of v1. Inside the envelope, the numbers an operator would plan with

are the same or better.

What an operator should re-measure after an upgrade. The checklist this paper used:

1. The timer floor and the effective timer resolution, before any nanosecond-scale comparison.
2. The end-to-end order path at the median, P99 and P99.9, twice.
3. A concurrency sweep through one strategy per physical core and beyond, to find the plateau's edges.
4. Run-to-run spread at a fixed load, pinned and unpinned.
5. The ring-size curve around the L3 size.
6. A ledger hash of a fixed backtest, to confirm exact reproducibility on the new build.
7. Any background work on the machine, such as a RAID resync, before the run starts; and a rerun of any benchmark it overlapped.

Appendix A. Releases between the two runs

From the product changelog [5]. ABI numbers are the research engine's unless stated.

Release	Date	Summary
4.3.0	2026-09-13	Benchmark documentation only. ABI 5.
4.3.1	2026-09-17	ABI stability policy documented.
4.3.2	2026-10-03	Licensing documentation; a fix to the Server's licence-expiry notice.
4.4.0	2026-10-09	ABI 5 to 6. Functions renamed to `reamer_*`; the v1 and v2 entry points removed. Restored memory-mapped backtests from files, the bar-file writer, the CSV builder, Monte Carlo and the equity curve. Fixes to the open-orders list and to a ticker shift on zero-bar tickers.
4.5.0	2026-10-09	ABI 6 to 7. Position states, open orders, modify and close-all actions, and a progress callback. Relay to Reamer Server, with brackets (Server ABI 4). A cancel takes effect after the bar replays, at a cost of about 1–2%.
4.5.1	2026-10-15	Security hardening: test switches only in test builds, stripped binaries.
4.6.0	2026-10-25	No ABI change. The strategy sees the post-fill state on the next bar; a full close cancels only closing orders; the Python per-bar call is about 4× cheaper.

Release 4.6.1 (2026-10-26) followed the run and is not measured here.

Declarations

Data availability. The raw outputs of this run are openly available under CC BY 4.0 at `https://reamerlabs.com/papers/data/epyc-9575f-4.6.0-raw-artifacts.zip` [3]. The earlier run's outputs are at `https://reamerlabs.com/papers/data/epyc-9575f-2026-09-raw-artifacts.zip` [2].

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

[2] Palghadmal, C. Raw benchmark artifacts for "Predictable Trading Infrastructure on Many-Core CPUs" (AMD EPYC 9575F). Dataset, Reamer Labs, 2026. https://reamerlabs.com/papers/data/epyc-9575f-2026-09-raw-artifacts.zip.

[3] Palghadmal, C. Raw benchmark artifacts for "Does the Operating Envelope Survive an Upgrade?" (Reamer Server and Reamer Research 4.6.0, AMD EPYC 9575F). Dataset, Reamer Labs, 2026. https://reamerlabs.com/papers/data/epyc-9575f-4.6.0-raw-artifacts.zip.

[4] Palghadmal, C. What a Backtest Costs: Throughput, Memory, Scaling and Reproducibility of Four Backtesting Engines on a 64-Core CPU. Preprint, 2026-10-26. https://reamerlabs.com/papers/what-a-backtest-costs.pdf.

[5] Reamer Labs. Reamer Research changelog. https://reamerlabs.com/docs/research-changelog. Accessed 2026-10-26.

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.*