An Exchange in a Box: Workload-Honest Measurements of Order Books, Kernel I/O Under Microburst, and Tick-to-Trade

Abhishek Aditya

Personal learning project — built on personal time and hardware from public specifications and public sample data; not affiliated with any employer, past or present.

2026-08-28 · code, data manifests, and raw results: https://github.com/Abhishek-Aditya-bs/atlas

Abstract. Public comparisons of limit-order-book data structures almost always drive them with synthetic price processes; real exchange traffic concentrates near the touch over a deep, cold book. We implement five book layouts behind one interface and measure them under both a uniform-random workload and replayed Nasdaq TotalView-ITCH 5.0 sample data, with hardware counters. The synthetic workload's verdict does not survive: a 3.7× median-latency advantage of dense array/bitmap layouts collapses to ≈1.1× under real replay, a sorted vector moves from last place to a statistical tie for second, and the p99.9 ranking inverts outright — the pointer-based treap becomes the best tail. We further measure kernel receive paths under burst schedules extracted from the real opening auction rather than at steady state: the burst triples median queueing delay, recovery takes milliseconds, and coordinated-omission-free accounting through the sender exposes two loopback measurement traps that steady-state means hide. Finally, a per-stage, instrumentation-cost-aware tick-to-trade decomposition of the complete feed-to-order pipeline on a rented 4-core arm64 VM lands at a 5.0 µs median — of which the userspace pipeline is ≈ 400 ns and the order-send syscall is 92 %. All specifications and data are public; every figure regenerates from committed result files.

1 Introduction

Three questions recur in every discussion of trading-system engineering, and the public record answers none of them well.

Which order-book structure is fastest? The folk answer (dense per-tick arrays with a bitmap summary) rests on benchmarks whose price streams look nothing like an exchange's. Uniform-random prices scatter work across the whole structure; the empirical microstructure literature has shown for two decades that real order flow concentrates tightly around the best quotes [1], [2], [3]. If the workload determines cache behaviour and cache behaviour determines the ranking, benchmarks with the wrong workload may crown the wrong structure. Practitioner talks concede this qualitatively [4], [5]; we could find no published controlled measurement of it (§2).

What does the kernel's receive path cost when it matters? Published I/O-stack comparisons report mean throughput at steady state [6], [7]. Market data is violently bursty — in our sample day, the worst 100 ms of the opening auction carries 3.5× the mean rate — and it is the burst, not the steady state, that decides whether a feed handler keeps up. We compare four receive paths **conditioned on the burst**, reporting queueing delay against the intended schedule (coordinated-omission-free [8]) and time-to-recover.

What does tick-to-trade decompose into? Vendor numbers are marketing; desk numbers are confidential. Public, reproducible per-stage decompositions on hardware anyone can rent are nearly nonexistent. We publish one, with the measurement overhead itself measured and reported rather than silently ignored.

The system answering these questions is a complete miniature exchange stack — ITCH 5.0 decoder, five order books, a deterministic matching engine with byte-identical replay, a MoldUDP64 publisher with a gap-recovering book-builder verified digest-identical against the engine, an OUCH 4.2/SoupBinTCP order-entry session, and the receiver ladder — built as a documented learning codebase (every non-obvious C++ construct is explained in-line and in a growing primer). The teaching apparatus is out of scope here but shapes the engineering: nothing below was tuned in a way the repository does not explain.

2 Related work and what is actually novel

Order books and workloads. The empirical result that order placement concentrates near the touch is long established: Zovko and Farmer measure a power-law of placement depth [1], Bouchaud et al. fit the same family

[2], Cont et al. build their generator on level-indexed intensities $\lambda(i) \propto i^{\wedge}(-\alpha)$ [3], and Gould et al. survey the field [9]. Our contribution here is replication on the exact benchmark input (§4.2), not discovery. Practitioner sources describe the candidate structures — Selph's intrusive price-level tree [5], QuantCup's winning flat array [10], sorted-vector books in public engine write-ups, and bitmap/trie indexes with find-first-set navigation in recent practitioner work [11] — and some benchmarks replay real data [12]. What we could not find anywhere is the controlled experiment: **the same implementations, same harness, same counters, synthetic and real workloads side by side, asking whether the ranking survives**. Selph states the ranking is workload-dependent without measurement; Gross concedes the flip qualitatively in Q&A [4]; one recent benchmark runs both arms but with a locality-matched generator, so one structure wins everywhere [11]. We therefore frame Claim 1 as the first controlled measurement of a widely believed folk theorem, and we weaken the folklore premise accordingly: published benchmarks commonly use synthetic price processes whose **depth-of-activity distribution** is unrealistic, and rarely pair the arms on identical implementations.

I/O under burst. The XDP paper [6] and the io_uring literature [7], [13] evaluate throughput at offered steady rates; datacenter measurement work documents microbursts as the dominant loss mechanism [14]; trading-infrastructure write-ups describe burst-driven provisioning [15]. Tene's coordinated-omission critique [8] defines the measurement discipline. We found no public comparison of these specific receive paths driven by real market-data burst shapes and scored on burst-conditioned tails and recovery; Claim 2′s novelty is that conditioning, with the ingredients individually well established.

Deterministic engines and tick-to-trade. The LMAX architecture [16], [17] and Aeron Cluster's replicated-log design [18] document deterministic, journal-replayable engines; open-source matching engines exist [19], [20]. Byte-identical replay is therefore an engineering demonstration, not a research claim, and we label it so. For tick-to-trade, public per-stage decompositions on commodity hardware with stated instrumentation cost are, to our knowledge, not available in citable form; vendor figures do not come with methods sections.

3 The system in one page

The repository implements both sides of the wire. On the exchange side: a single-threaded price-time matching engine whose inputs are journalled before execution and whose event stream and book digest replay byte-identically (500 000 commands → 904 149 events, identical FNV-1a and state digest across replays and across ISAs, since the journal is big-endian); a MoldUDP64 publisher with deliberate datagram-drop injection and a retransmission server; an OUCH 4.2-over-SoupBinTCP gateway whose sequenced stream supports sever-and-relogin recovery, property-tested at every cut point. On the participant side: a zero-copy, schema-driven ITCH 5.0 decoder (all 23 message types; layout static_asserts derived from the specification text; libFuzzer-clean with a real-data seed corpus; 65.6 M real frames replayed with zero unknown types); the five book layouts behind one C++20 concept; a quote-and-cancel strategy; and the four-backend receiver ladder. The strongest correctness argument in the repository is differential: a book rebuilt from the lossy multicast feed — including forced gap recovery at drop rates up to 1-in-7 and an unpaced 12.6 M msg/s overrun — produces a digest identical to direct replay of the same capture; and inside the Claim 1 harness itself, all five layouts must produce identical touch checksums on every run.

Measurement discipline (repo ADR 0005): the arm64 VM's generic timer ticks at 25 MHz (40 ns resolution; read cost ≈ 17.5 ns), so sub-100 ns operations are timed in batches of 32 and every result file embeds the calibration, the machine block, and the exact command line. Nothing below reports a bare mean.

4 Claim 1 — the workload is the result

4.1 Setup

Five layouts behind one interface (mutations: add / reduce / remove / replace; queries: best bid and ask after every mutation, which also defeats dead-code elimination): `std::map` of FIFO levels (the reference, property-tested); a per-side sorted `std::vector` with the best level at the hot end; a flat array of per-tick slots over a 2^{16}-tick window with a cached-best cursor and linear rescan on touch collapse; the same window with a 3-level occupancy bitmap giving best-price via three count-zeros; and an arena-pooled intrusive treap with cached min/max. The windowed layouts route off-grid and out-of-window prices to a counted ordered-map fallback that participates in best-price resolution — hidden escapes would rig the comparison. The order-id index is shared across the four aggregate layouts so it cancels out.

Both workloads compile to the same operation tape and run through the same loop. **Synthetic uniform**: 2 M operations, adds at uniform prices across a 10 000-tick band, balanced randomized removals — the literature's generator. **Real replay**: every order-book message for the most active symbol (QQQ) in Nasdaq's public 2019-12-30 sample day between 09:25 and 10:30 (743 543 operations), applied in exchange order. The tape build also characterises the input: **82.0 % of priced operations land within 5 ticks of the prevailing touch** — our replication of [1], [3] on the exact benchmark input. Five repetitions per cell, one process per layout, pinned to an otherwise idle core, `perf stat` in separate instrumented runs.

4.2 Results

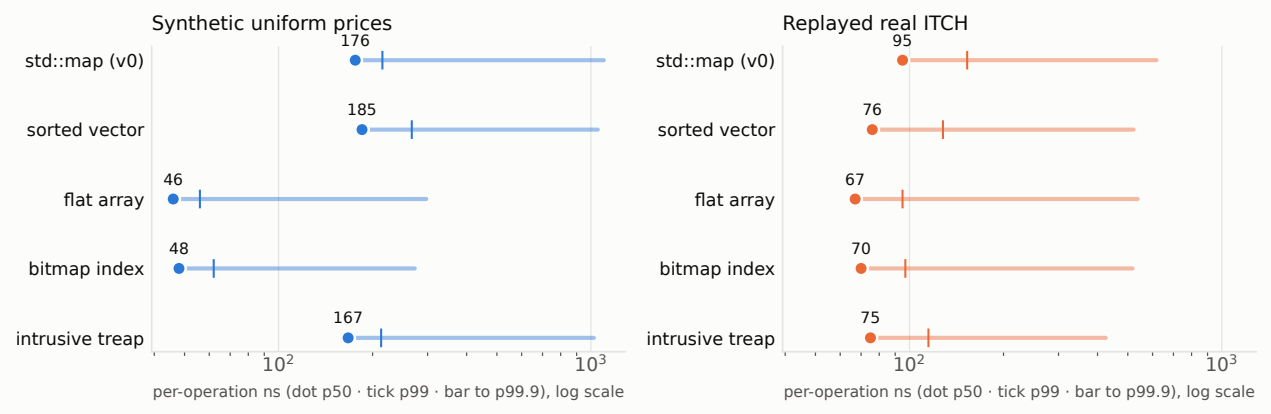

Figure 1: Per-operation latency, five layouts × two workloads: dot = median-of-reps p50, tick = p99, bar end = p99.9 (log scale). Generated from the committed result file by `make_figures.py`.

Medians of per-repetition percentiles, in ns/op (rep-to-rep p50 spread in parentheses); result file `bench/results/claude-code-vm-aarch64-20260828-book.json`:

layout	syn p50	syn p99	syn p99.9	replay p50	rep p99	rep p99.9
std::map	176 (172–183)	215	1100	95 (95–97)	153	618
sorted vector	185 (170–222)	267	1052	76 (75–76)	128	522
flat array	**46** (45–47)	**56**	297	67 (67–67)	**95**	538
bitmap	48 (48–50)	62	**273**	70 (70–70)	97	518
treap	167 (162–175)	213	1022	75 (73–75)	115	**425**

Three observations, each visible in the raw distributions:

1. **The tier structure collapses.** Under synthetic uniform there are two tiers separated by 3.7× (array/bitmap ≈ 46–48 ns; everything else 167–185 ns). Under real replay the five layouts span 67–95 ns: the dense layouts' advantage over the sorted vector and the treap is ≈ 1.1×, within the day-to-day engineering noise of any real system.
2. **Ranks move.** The sorted vector is last under synthetic (185 ns) and statistically tied for second under replay (76 vs 75 ns, rep spreads overlapping). The map improves 1.9× without changing code, because the workload keeps its hot paths resident.
3. **The tail ranking inverts.** At p99.9 the treap is mid-pack under synthetic (1022 ns vs the array's 297 ns) and **best of all five** under replay (425 ns, vs 518–538 ns for the dense layouts). A practitioner choosing by synthetic-workload tails would pick exactly backwards for tail-sensitive real-data use.

The hardware counters give the mechanism. Moving from synthetic to replay, the map's cache-miss rate falls 3.5 % → 1.5 % and its branch-miss rate halves (7.2 % → 3.2 %); the treap behaves the same way — pointer chasing is only expensive when the pointers are cold, and a touch-concentrated workload keeps the walked path hot. The dense layouts' counters barely move (their per-op work was already constant), but their IPC **drops** (1.84 → 1.46): under uniform traffic their advantage was mostly that everyone else was missing cache, not that they got faster. Fallback-escape counts for the windowed layouts were 0 (synthetic, on-grid by construction) and 55 of 743 543 replay operations (0.007 %), published per run in the result file.

Scope, honestly. This is one symbol-day on one ISA with aggregate (non-FIFO) level books; the result is the measured collapse-and-tail-inversion under a realistic workload, not a claim that any layout is globally best. The synthetic arm is the literature's generator, not a strawman we invented — that is the point of the comparison.

5 Claim 2 — latency under burst, not throughput at steady state

5.1 Setup

A sender thread replays the **actual message-arrival times** of the sample day's opening auction — 928 153 messages in the five seconds after 09:30:00.000, whose worst 100 ms bin (65 332 messages) runs at 3.5× the window's mean rate — as one UDP datagram per message on loopback, busy-waiting on an intended schedule; a control schedule offers the same mean rate (185 k/s) steadily. Each datagram carries its sequence number and **intended** send tick; the receiver records `now − intended` per packet, so sender lateness and receiver queueing are both charged to the distribution rather than silently forgiven [8] (sender lateness is also reported separately). Four backends behind one drain interface: per-datagram `recvfrom`; `recvmmsg` in batches of 64; io_uring with 256 pre-posted receives re-armed by one submit per drain (multishot requires liburing ≥ 2.2 — x86-host follow-up, stated in the result file); AF_XDP, which this VM's virtualised NIC cannot run and which therefore reports **unavailable** rather than a simulated number. Recovery time = from the end of the peak burst bin until measured delay first stays under 2× the quiet-phase median for 50 consecutive packets. Five repetitions per cell.

5.2 Results

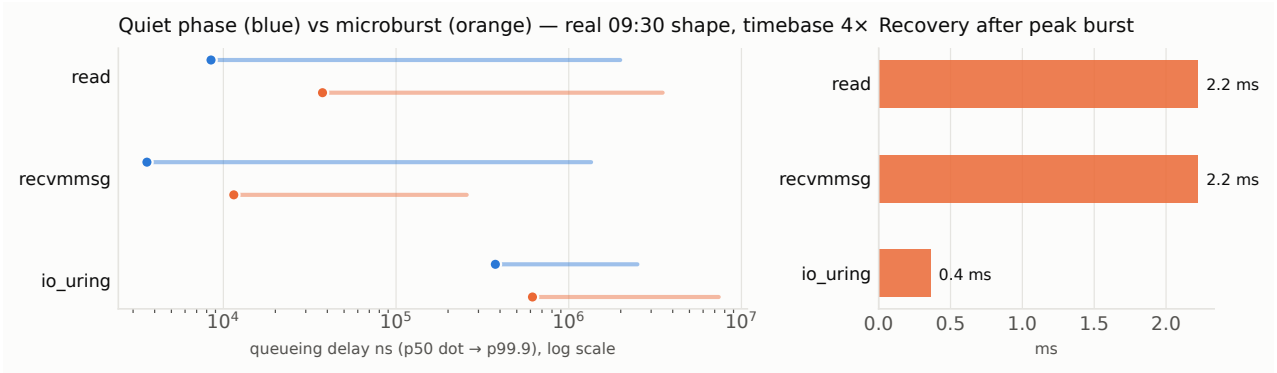

Figure 2: Queueing delay through the replayed 09:30:00 open, per backend: quiet-phase vs in-burst distributions (p50 → p99.9), and recovery time after the peak 100 ms bin.

Medians of per-repetition percentiles (queueing delay vs the intended schedule, ns), from `bench/results/claude-code-vm-aarch64-20260828-burst.json`; the sender held its schedule in every official cell (lateness p50 = 0), and no run lost a datagram:

backend	quiet p50	quiet p99.9	burst p50	burst p99.9	recovery
read	3 368	222 720	10 912	251 392	2.2 ms
recvmmsg	3 608	227 840	11 296	254 464	2.2 ms
io_uring	375 808	2 957 312	624 640	7 815 168	0.36 ms
AF_XDP	unavailable on this host (no XDP driver/privileges) — by design an empty column, not a simulated one				

Findings, in decreasing order of confidence:

1. **Burst conditioning is visible and quantifiable.** For the syscall backends the real burst triples the median delay (3.4 µs → 11 µs) and lifts the p99.9 above 250 µs, with ≈ 2.2 ms to drain back to quiet behaviour after the peak 100 ms bin. A steady-state-only report at the same mean rate (p50 ≈ 3.2 µs, table in the result file) would show none of this.
2. **At single-core-sustainable rates, `recvmmsg` ≈ `read`.** The batching advantage appears only when a backlog exists to batch: in the archived full-rate saturation study (same result-file family, `scale = 1.0`), per-datagram `read` collapsed (sender dragged 347 ms off schedule at a steady 185 k/s) while `recvmmsg` held — at the stretched rates both keep the queue near-empty and pay one syscall per drain either way.

3. **io_uring's pre-posted receives are pathological on loopback** (quiet p50 376 µs): loopback delivery runs in the sender's context, so the completion work (copy into the posted buffer, CQE publication) lands on the sender's core and the CQ cachelines ping-pong between cores. This is an artifact of the transport, reported as such — on a NIC that work belongs to the driver — and it is exactly why the x86/NIC follow-up column exists. The harness evolution that led here (a `poll(fd)` sleep that can never wake for ring-consumed sockets; a spin loop whose empty-drain syscalls live-locked the peer through the socket lock; the bounded-backoff spin that resolved both) is documented in ADR 0011 — each wrong method produced confidently wrong numbers that the sender-lateness channel caught.

6 Claim 3 — an honest tick-to-trade budget

6.1 Setup

The full participant loop in one process: MoldUDP64-framed real ITCH replayed at an offered 185 k msg/s; the hot path unframes, decodes, filters to QQQ, updates the bitmap book, runs a quote-and-cancel strategy (requote one tick inside whenever the touch moves), encodes real OUCH 4.2 Enter/Cancel messages, and sends them on a connected UDP socket. Seven counter reads stamp the stage boundaries of every book-mutating message; the read cost (17.5 ns each, measured in-process at startup) is **reported next to every figure, not silently subtracted**. Two populations are kept separate — mutations that fired at least one order message (the "action path", the number that matters) and quiet book updates — because averaging them together would bury the action tail. Two configurations: the single-core hot loop, and a two-core handoff through a bounded SPSC ring (the interface stub for `spindle`, this portfolio's IPC project), which prices the inter-thread hop as its own line item. Percentiles quantise to the 40 ns tick; means recover sub-tick information because stage boundaries are asynchronous to the counter (both are reported).

6.2 Results

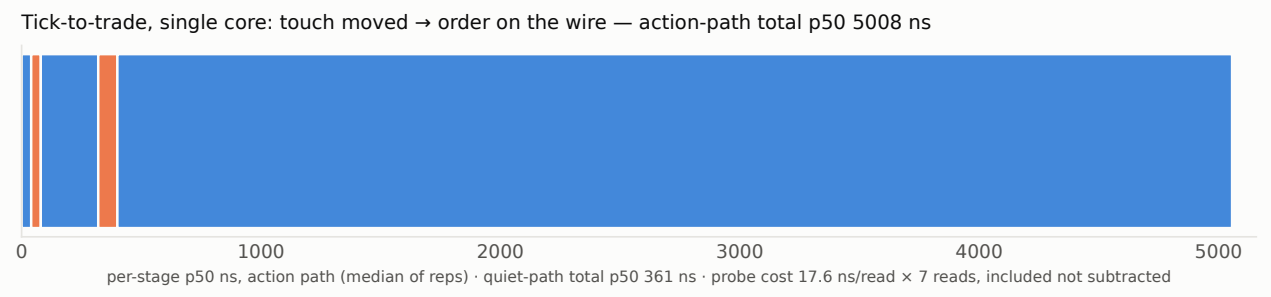

Figure 3: Per-stage attribution of the action path (touch moved → order on the wire), single-core configuration.

Medians of per-repetition percentiles (single-core configuration; 5 reps × 2 M messages at an offered 185 k msg/s; result file `bench/results/claude-code-vm-aarch64-20260828-t2t.json`):

stage	p50	p90	p99	p99.9	mean
unframe	40	40	40	40	20
ITCH decode	40	40	40	160	22
book update	240	1 644	3 768	10 336	685
decide + OUCH encode	80	200	842	1 284	125
send (1–3 order msgs)	4 656	23 616	31 808	49 280	8 181
tick-to-trade, action path	**5 008**	25 280	35 200	199 168	9 038
quiet path (no order fired)	361	2 568	4 752	7 056	884

The decomposition's message is blunt: everything a C++ hot path controls — unframe, decode, book, decide, encode — totals ≈ 400 ns at the median, and the **UDP send syscalls are 4.7 µs: 92 % of the 5.0 µs tick-to-trade median on a stock kernel path**. Optimisation applied where the data pointed: the first harness draft used unconnected `sendto()` and paid a per-call route lookup (8.7 µs median send); connecting the socket once roughly halved it — the kind of stage-attributable delta this instrument exists to produce. Book-update tails (p99.9 ≈ 10 µs, max ≈ 200 µs) are hash-map rehashes and price-window anchor faults, named in ADR 0015.

The instrumentation itself: 7 × 17.5 ns of counter reads per message, included in every number above, never subtracted.

The two-core configuration through the spindle-interface ring prices the cross-core hop at ≈ 680 ns (median) — but on this 4-core VM the strategy core is contested and the action-path median doubles (11.3 µs) with hundred-microsecond tails: the single-core loop wins on this host, and the ring column holds the slot `spindle`'s shared-memory measurements will fill.

7 Experimental setup

One Oracle Cloud VM.Standard.A1 instance: Ampere Neoverse-N1 (aarch64), 4 cores, no SMT, one NUMA node, 64 KiB L1d / 1 MiB L2 per core, kernel 6.8.0-1049-oracle, g++-12.3.0 `-O3 -std=c++20`, governor and machine state captured in each result file's header by `scripts/machine-state.sh`. Generic timer: 25 MHz (40 ns resolution), invariant by architecture; calibration (frequency, read overhead, resolution) embedded per run. Threads pinned; benchmark processes run one layout or one backend per process so `perf stat` counters are attributable. Data: Nasdaq's public sample day 2019-12-30 (`emi.nasdaq.com`, sha256-pinned in `data/checksums.sha256`), never redistributed. LOBSTER's sample files were evaluated as a second dataset and **rejected**: their 2026 terms forbid publishing derived results without a paid licence (`docs/SOURCES.md`, Rejected).

8 Threats to validity

One day, one venue, one symbol class. The workload characterisation (82 % within 5 ticks) is one liquid ETF on one Nasdaq day; less liquid names have sparser books and the collapse may be smaller. The claim is calibrated to what was measured. **Loopback is not a NIC.** Claim 2′s absolute numbers include no wire, no NIC ring, no interrupt path; the comparison across backends shares that floor, and the burst conditioning — the claim's substance — survives it. AF_XDP and multishot io_uring require the x86 host; publishing their column empty is deliberate. **Clock coarseness.** 40 ns quantisation means sub-tick per-stage percentiles are bucketed; batching and reported means address this and the text never claims below-resolution precision. **A shared cloud VM.** Neighbours exist; five repetitions with published spreads and an idle-load precondition bound the noise, and every result file records the load state. **Aggregate books.** Claim 1′s layouts maintain aggregate levels (as a feed handler does), not per-order FIFO queues (as an engine does); v0 with FIFO queues is the semantic reference but the four contenders answer the feed-handler question only. **What we did not tune.** No layout received workload-specific tuning after results were first seen; window sizes and batch constants were fixed by the pre-registered ADR hypotheses.

9 Conclusion

The benchmark you run is the answer you get. Under the literature's synthetic workload, dense windowed order books dominate by 3.7×; under the workload the market actually produces, four of five layouts are within ≈ 15 % at the median and the tail ranking flips in favour of the pointer structure the folk wisdom dismisses. Burst-conditioned I/O measurement and an instrumentation-honest tick-to-trade budget follow the same principle: measure the case that breaks systems, publish the cost of measuring, and let the distributions — not the means — carry the conclusion.

10 Reproduction

```
git clone https://github.com/Abhishek-Aditya-bs/atlas && cd atlas
cmake --preset release && cmake --build --preset release && ctest --preset release
scripts/fetch-data.sh                      # public Nasdaq sample day + specs, sha256-verified
scripts/bench_book.sh data/itch/rth-open.itch50 QQQ     # Claim 1 (results + perf counters)
scripts/bench_burst.sh data/itch/rth-open.itch50        # Claim 2
scripts/bench_t2t.sh data/itch/rth-open.itch50          # Claim 3
python3 paper/figures/make_figures.py       # regenerate every figure from bench/results/
typst compile paper/atlas.typ paper/atlas.pdf
```

Sources. Every specification, dataset, tool, and idea used — with URL, retrieval date, terms, and how it is used — is logged in `docs/SOURCES.md`; candidates whose terms forbade published derived results are logged as rejected. Related-work reading notes with per-claim novelty assessments: `docs/related-work/`.

Bibliography

[1] I. Zovko and J. D. Farmer, "The power of patience: a behavioural regularity in limit-order placement," *Quantitative Finance*, vol. 2, no. 5, 2002, [Online]. Available: https://arxiv.org/abs/cond-mat/0206280

[2] J.-P. Bouchaud, M. Mézard, and M. Potters, "Statistical properties of stock order books: empirical results and models," *Quantitative Finance*, vol. 2, no. 4, 2002, [Online]. Available: https://arxiv.org/abs/cond-mat/0203511

[3] R. Cont, S. Stoikov, and R. Talreja, "A stochastic model for order book dynamics," *Operations Research*, vol. 58, no. 3, 2010, [Online]. Available: https://www.columbia.edu/~ww2040/orderbook.pdf

[4] D. Gross, "Trading at light speed: designing low-latency systems in C++ (Meeting C++ keynote)." [Online]. Available: https://www.youtube.com/watch?v=sX2nF1fW7kI

[5] W. Selph, "How to build a fast limit order book." [Online]. Available: https://web.archive.org/web/20120811163842/http://www.quantcup.org/home/howtohft_howtobuildafastlimitorderbook

[6] T. Høiland-Jørgensen *et al.*, "The eXpress Data Path: fast programmable packet processing in the operating system kernel," in *ACM CoNEXT 2018*, 2018. [Online]. Available: https://raw.githubusercontent.com/xdp-project/xdp-paper/master/xdp-the-express-data-path.pdf

[7] D. Didona, J. Pfefferle, N. Ioannou, B. Metzler, and A. Trivedi, "Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring," in *ACM SYSTOR 2022*, 2022. [Online]. Available: https://atlarge-research.com/pdfs/2022-systor-apis.pdf

[8] G. Tene, "How NOT to measure latency." [Online]. Available: https://www.youtube.com/watch?v=lJ8ydIuPFeU

[9] M. D. Gould, M. A. Porter, S. Williams, M. McDonald, D. J. Fenn, and S. D. Howison, "Limit order books," *Quantitative Finance*, vol. 13, no. 11, 2013, [Online]. Available: https://arxiv.org/abs/1012.0349

[10] "QuantCup: the price-time matching engine programming contest (specification and winning entries)." [Online]. Available: https://web.archive.org/web/20120723043435/http://www.quantcup.org/

[11] shdown, "glass: a high-performance limit order book with cached paths and find-first-set navigation (paper and benchmarks)," 2025. [Online]. Available: https://github.com/shdown/glass-paper

[12] Chronoxor (Ivan Shynkarenka), "CppTrader: high-performance trading platform components with NASDAQ ITCH replay benchmarks." [Online]. Available: https://github.com/chronoxor/CppTrader

[13] J. Axboe, "Efficient IO with io_uring." [Online]. Available: https://kernel.dk/io_uring.pdf

[14] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *ACM IMC 2017*, 2017. [Online]. Available: https://conferences.sigcomm.org/imc/2017/papers/imc17-final60.pdf

[15] A. Myers, J. Nigito, and B. Foster, "Network design considerations for trading systems," in *ACM HotNets 2024*, 2024. [Online]. Available: https://conferences.sigcomm.org/hotnets/2024/papers/hotnets24-262.pdf

[16] M. Fowler, "The LMAX architecture." [Online]. Available: https://martinfowler.com/articles/lmax.html

[17] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "Disruptor: high performance alternative to bounded queues for exchanging data between concurrent threads," 2011. [Online]. Available: https://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf

[18] "Aeron Cluster: replicated state machines over a raft-consensus log." [Online]. Available: https://aeron.io/docs/cluster-quickstart/replicated-state-machines/

[19] "exchange-core: an open-source matching engine with LMAX Disruptor pipelines." [Online]. Available: https://github.com/exchange-core/exchange-core

[20] I. Jericevich, D. Sing, and T. Gebbie, "CoinTossX: an open-source low-latency high-throughput matching engine," *SoftwareX*, 2022, [Online]. Available: https://arxiv.org/pdf/2102.10925