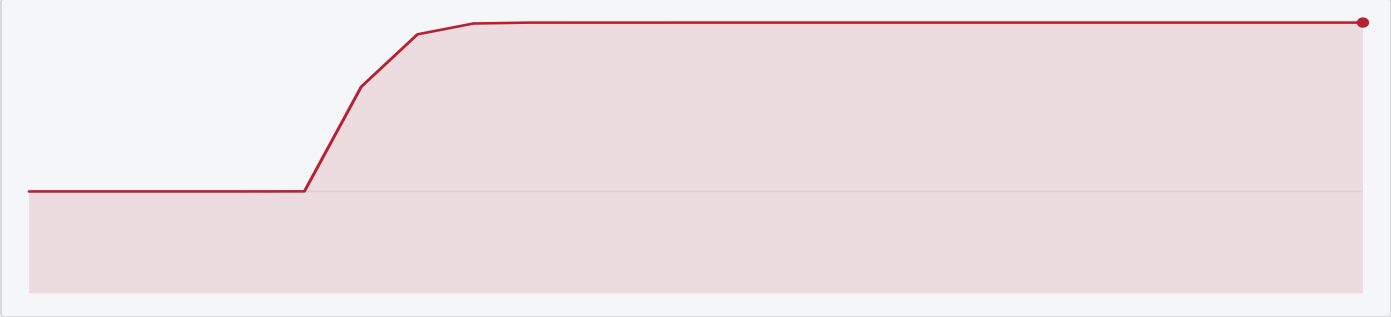
Systems note · single-thread C++17 · reproducible

Low-Latency Order Book

A single-threaded matching engine that keeps the entire hot path inside the CPU cache: contiguous price levels instead of trees, a fixed startup arena instead of the allocator, and a lock-free single-producer/single-consumer ring decoupling ingestion from matching. Every number on this page is emitted by the engine itself.

Replay throughput · ev/s	Median latency · ticks	Data races (TSAN)	Hot-path allocations
6.6M	63	0	0



Cumulative distribution of per-event latency over 5,000,000 add/cancel events; half complete within 63 counter ticks.

01 Protocol 02 Transport 03 Matching 04 Correctness 05 Results

The system is split into two execution domains pinned to separate cores: a **feed handler** that does nothing but push raw bytes, and a **matching engine** that pops them, decodes, routes by symbol, and matches. They share only a lock-free ring buffer — no mutex, no allocation, no shared mutable state on the hot path.

01
Feed handler
core 0 · push bytes

→

02
SPSC ring
atomic cursors · alignas(64)

→

03
Matching engine
flat levels · SIMD scan

→

04
Order book
pooled · price-time

`bounded` → every container is sized at construction. The matching loop never calls `new` or `malloc`, so per-event latency carries no allocator jitter and the working set is known ahead of time.

01 The binary feed

The engine consumes a fixed-width binary feed: every command — limit, market, cancel, modify, replace — is a **40-byte little-endian record**. Fixed width means the decoder never branches on length and the producer never allocates; messages are blitted into the ring as raw bytes and parsed on the far side.

Fig. 1 The wire record. Order-id fields are shaded crimson, the `'OBL1'` magic trailer slate. Decode validates the magic word and the type/side ranges before a message is admitted, so a corrupt frame is rejected rather than matched.

A frame is accepted only when its trailing magic word equals `0x4F424C31` (ASCII `"OBL1"`) and its type and side bytes are in range; otherwise it is dropped at the boundary. The replay benchmark on this page round-trips a recorded stream through this exact codec — `1,100,000` encode / decode / match cycles — with zero rejects.

02 Lock-free transport

Ingestion and matching are joined by a single-producer/single-consumer ring buffer. The only shared state is a pair of 64-bit cursors. The capacity is a power of two, so the slot index is a bit-mask rather than a modulo:

$$\mathrm{slot}(i) = i \bmod N = i \,\&\, (N-1), \qquad N = 2^{16} = 65{,}536 \text{ slots}.$$

The ring is **full** when $\mathrm{tail} - \mathrm{head} = N$ and **empty** when $\mathrm{tail} = \mathrm{head}$. Correctness rests on release/acquire ordering, not sequential consistency: the producer publishes a slot with a `store(tail, release)`; the consumer observes it with a `load(tail, acquire)`. That single edge *synchronizes-with* the read, guaranteeing the buffer write is visible before the slot is dequeued — the minimum barrier the hardware needs, and no more.

N	*align*	*cmd*	*order*
65,536	64	40	40
ring slots (power of two)	byte cursor separation	bytes per queued command	bytes on the wire

Producer and consumer cursors sit on separate 64-byte cache lines via `alignas(64)`, so the two threads never invalidate each other's line (false sharing). There is no compare-and-swap: one producer and one consumer make plain load/store with acquire/release sufficient, which is why an SPSC ring beats a multi-producer queue at the hardware limit.

---

03 The matching engine and its memory

A `std::map` of price levels is asymptotically tidy and cache-hostile: every node is a pointer chase into cold heap. Here both sides are **fixed-size contiguous arrays** of price levels, sorted by price; orders are drawn from a startup object pool and linked intrusively within each level. A `PriceLevel` is exactly 32 bytes — one half cache line, the width the SIMD scan loads at a time.

Resting orders obey strict **price-time priority**. For two resting bids a, b, a executes first exactly when

$$a \prec b \iff p_a > p_b \ \vee \ \big(p_a = p_b \wedge t_a < t_b\big),$$

with the price comparison reversed for asks. A *modify* that only reduces quantity keeps the order's place in line; a *cancel-replace* deliberately forfeits it. Because every container is fixed at construction, the working set is bounded and known ahead of time:

$$M = N_o \cdot \mathrm{sizeof}(\mathrm{Order}) + L \cdot \mathrm{sizeof}(\mathrm{PriceLevel}) + C \cdot \mathrm{sizeof}(\mathrm{Entry}).$$

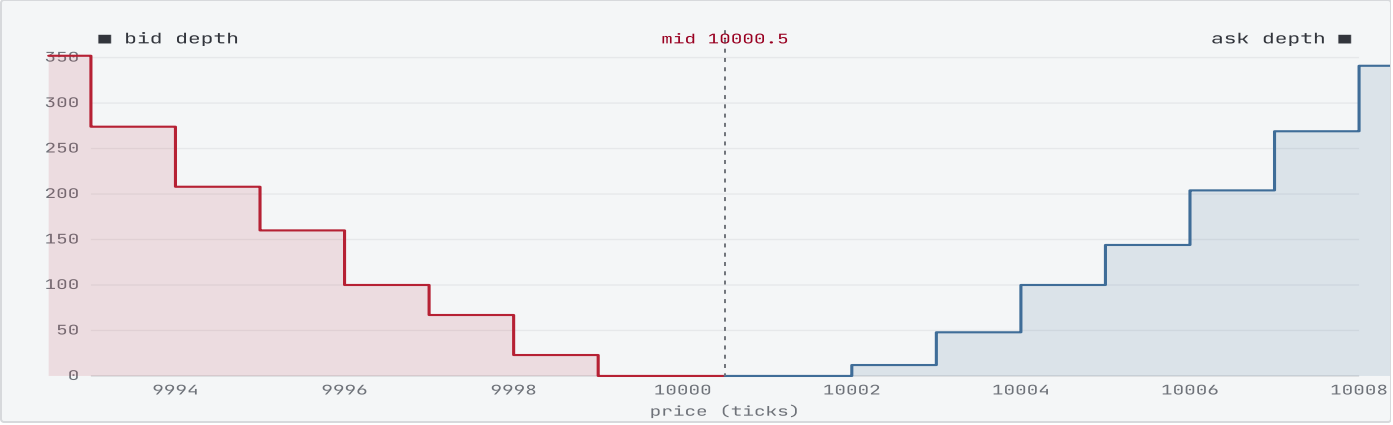


Fig. 2 Cumulative market depth after two crossing market orders consume the touch. Best bid 9,999, best ask 10,002, a 3-tick spread. The staircase is the engine's own `bid_depth()/ask_depth()` snapshot — internal storage is never exposed.

PriceLevel	*Order*	*ring*	*arena*
32 B	40 B	65,536	8.6 MB
256-bit SIMD record	pooled, intrusively linked	power-of-two slots	resident, allocated once

When a marketable order arrives, the engine scans the contiguous price array with a vector compare — AVX2 on x86, NEON on Apple Silicon, scalar elsewhere — testing eight (or four) levels per instruction and reading the first crossing index off the comparison mask. The crossing prints a deterministic trade tape:

seq	aggressor	price	qty
#1	BUY	10,001	8
#2	BUY	10,001	8
#3	BUY	10,002	12
#4	SELL	10,000	10
#5	SELL	10,000	10
#6	SELL	9,999	4

Tbl. 1 The 6 trades generated by the depth scenario, each tagged with the aggressor side, fill price, and quantity.

04 Correctness and race-freedom

Speed claims are only worth as much as the correctness underneath them. The engine ships with a deterministic test suite, a golden-file replay, and a ThreadSanitizer pass over the threaded pipeline. Every check below runs in the build that produced this page:

check	covers	result
● order_book_tests	Matching, partial/full fills, price-time priority, modify, replace, depth, SPSC FIFO	PASS
● protocol_tests	40-byte binary message encode / decode round-trip	PASS
● replay_golden	Deterministic replay matches the committed golden output byte-for-byte	PASS

Tbl. 2 Verification matrix. ● passing check · ○ failing check — status is also spelled out, so the table never relies on color alone.

The threaded pipeline is then rebuilt with `-fsanitize=thread` and driven with `10,000` rapid-fire events across the producer, ring, and consumer. ThreadSanitizer reports `0` data races — empirical backing for the

release/acquire reasoning in §02, not a substitute for it.

---

05 Latency and throughput

Latency is timed per event with the hardware cycle counter — `__rdtsc()` on x86, the virtual counter on ARM — and bucketed into a power-of-two histogram, where bucket b collects every event whose tick count falls in its octave:

$$b(x) = \lfloor \log_2 x \rfloor, \qquad \text{bucket } b \text{ covers } \left(2^{b},\, 2^{b+1}\right].$$

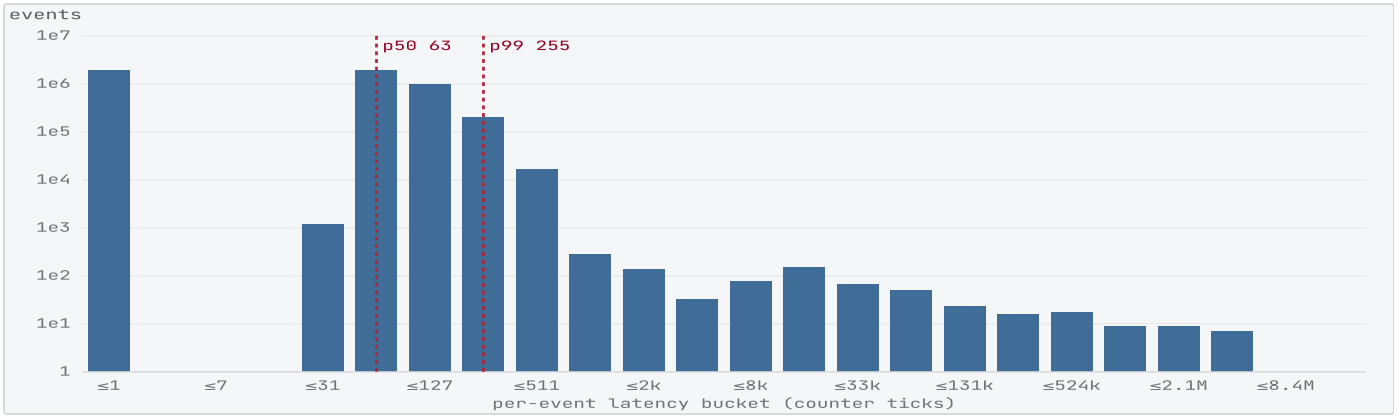


Fig. 3 Per-event add/cancel latency, counts on a log axis. The mass sits in the low-tick octaves — p50 at 63, p99 at 255 ticks — with a thin tail from scheduler interrupts the single thread cannot mask.

Plotting the tail by percentile shows the same story for both the bare hot path and the full decode → route → match replay: flat through the body, with the climb deferred deep into the nines.

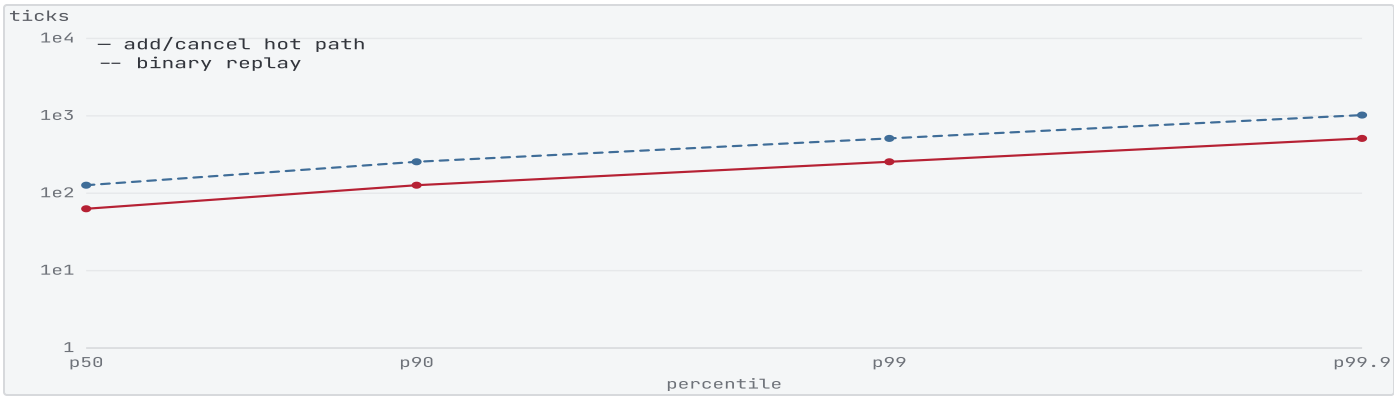


Fig. 4 Tail latency by percentile (log ticks). The replay path (dashed slate) carries decode and symbol routing on top of matching, so it sits above the bare hot path (crimson) — both stay bounded to the p99.9.

6.6M events per second through the full binary pipeline. The design target was 2M/s on one thread; every workload clears it, and pinned, isolated production cores would read higher than this loaded laptop.

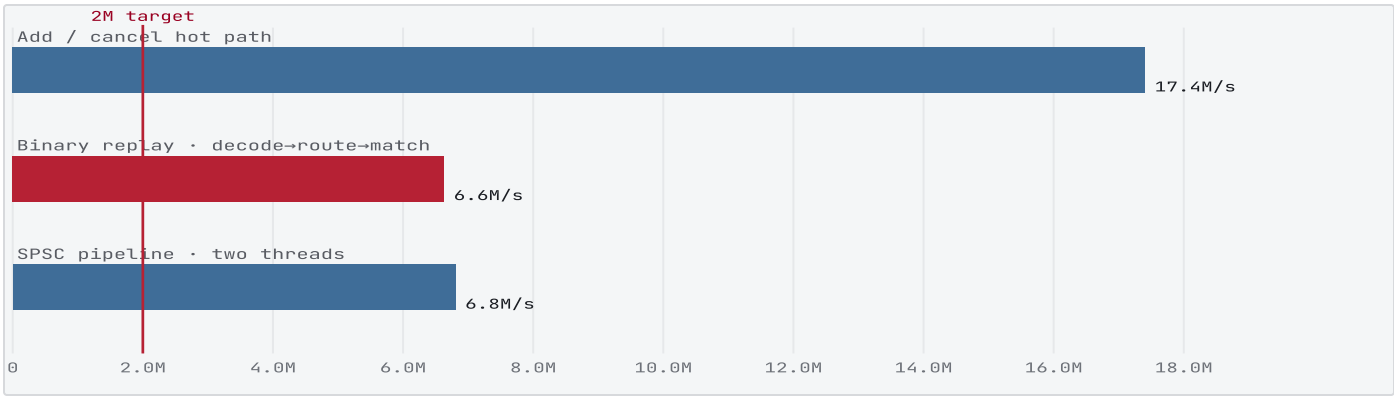


Fig. 5 Throughput by workload against the 2M/s target (crimson). The replay bar (accent) is the headline figure: a full binary pipeline, not an isolated micro-benchmark. The pipeline bar is the median of the size sweep in Fig. 6.

Throughput holds as the workload grows; smaller runs read higher because the working set stays warm in cache, which is itself the point of the flat layout.

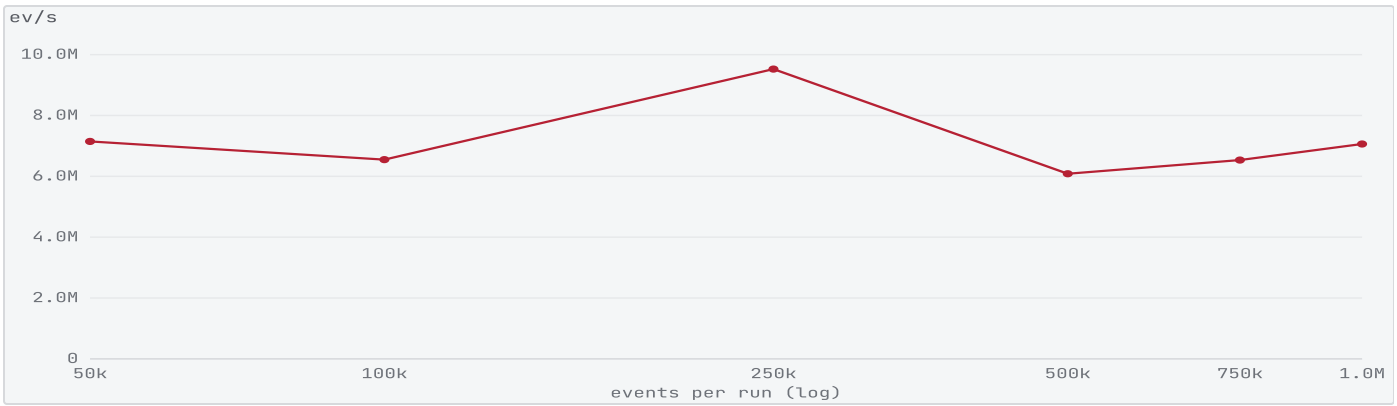


Fig. 6 SPSC pipeline throughput versus run size (log). The engine sustains multi-million-event-per-second matching across two orders of magnitude of batch size.

Replay throughput	6.6M/s	Hot-path throughput	17.4M/s
Pipeline throughput · median	6.8M/s	Events benchmarked	5,000,000
Median latency · p50	63 tk	Mean latency	58 tk

p90 latency	127 tk	p99 latency	255 tk
p99.9 latency	511 tk	Max latency	9.1M tk
Data races (TSAN)	0	TSAN events	10,000
PriceLevel size	32 B	Order size	40 B
Ring capacity	65,536	SIMD scan path	NEON

Tbl. 3 Full performance report — latency percentiles, throughput by workload, and the engine's bounded structural sizes. Latency is in hardware counter ticks (tk), architecture-specific magnitudes rather than nanoseconds.

Colophon

Header-only C++17 — flat price-level arrays, a startup object pool, an open-addressed order index, a lock-free SPSC ring, and SIMD price scans — built with a plain Makefile. The page is printed to PDF through headless Chrome; every figure is produced by a real engine run:

```
make site && PYTHONPATH=src python3 site/export_data.py
```

Built from 34b141c · Apple clang version 17.0.0 (clang-1700.3.19.1) · NEON price scan · Apple M4 · generated 2026-06-20T02:11Z · deterministic build.