High-Frequency Trading System Performance Evaluation

Groups A 1

1. Benchmark Result (Smart Pointer Baseline)

Table 1: Tick-to-Trade Latency Statistics (Smart Pointer Baseline)

Metric	Ticks	Min(ns)	Max(ns)	Mean(ns)	StdDev	P95	P99
Per-Tick	100000	100	8,178,200	743.3	28,310	1200	2300
Per-Trade	100000	100	405,400	603.2	3248.5	1100	2000

This configuration uses `std::unique_ptr` for ownership management and serves as the latency benchmark.

2. Performance Test Results

(1) Smart vs Raw Pointers: Multi-Scale Results (1K, 10K, 100K ticks)

100000 ticks (large workload): Raw pointers show lower mean latency (690.9 ns per tick vs 743.3 ns for smart), reflecting minor control-block overhead from `unique_ptr`. At this scale, the effect accumulates over many allocations, producing a clear but small average difference (≈7%). No memory issues were detected in AddressSanitizer (ASan) runs, but raw pointers inherently risk UAF or double-free in real-world use.

10000 ticks (medium workload): Smart pointers outperform in per-tick mean latency (1246 ns vs 1273 ns for raw), though raw remains faster in per-trade latency (456 ns vs 471 ns). This divergence likely stems from compiler optimizations—longer tick-to-trade paths allow the optimizer to inline `unique_ptr` moves more efficiently, while short trade-level timings reflect pure pointer overhead.

1000 ticks (small workload): Results fluctuate: smart pointers are faster per tick (7791 ns vs 7969 ns), but slower per trade (478 ns vs 464 ns). Low iteration counts amplify system noise (scheduler variation, CPU boost states). The observed reversals are not statistically significant.

Summary Analysis:

- Raw pointers show lower mean latency at high tick counts due to cumulative control-block overhead.
- Smart pointers perform similarly or better at smaller scales, likely due to compiler inlining and cache locality.

- Differences at 1K and 10K ticks are within expected jitter; only the 100K tick case demonstrates consistent performance separation.
- Memory safety benefits remain unique to smart pointers—although ASan reported no leaks, production systems face higher risk without RAII.

(2) Memory Alignment (alignas(64)) — 100000 ticks

Align64	Mean(ns) Per-Tick	Mean(ns) Per-Trade	P95(ns)	P99(ns)
On	577.3	519.3	1100	4600
Off	561.0	539.3	1100	6400

Analysis: Cache-aligned structures slightly lower tail latency (smaller P99) but show minimal mean difference, implying better cache predictability rather than throughput gain.

(3) Custom Allocator (Memory Pool vs new/delete) — 100000 ticks

Allocator	Mean(ns) Per-Tick	Mean(ns) Per-Trade	P95(ns)	P99(ns)
new/delete	2014.5	1852.7	4700	8900
Memory Pool	882.7	855.9	1800	6600

Analysis: Pool allocation nearly halves latency, confirming that pre-allocated slabs reduce fragmentation and heap contention in high-frequency contexts.

(4) Container Layout (Flat vs Map) — 100000 ticks

Container	Mean(ns) Per-Tick	Mean(ns) Per-Trade	P95(ns)	P99(ns)
Flat Vector	21251.5	20839.8	64500	95100
Map	1392.3	472.8	1000	5200

Analysis: Flat arrays exhibit high average latency under continuous insertions, as maintaining sorted order dominates cost. Maps provide logarithmic access with consistent cache behavior under real-time matching loads.

3. Overall Findings

- **Smart vs Raw:** Smart ensures safety; raw offers microsecond-level gains only under extreme repetition.
- **Alignment:** Reduces tail latency and cache misses; minor mean impact.
- **Allocator:** Memory pools offer the most measurable latency reduction.
- **Container:** Maps outperform flat arrays under dynamic workloads.

4. System Architecture Diagram

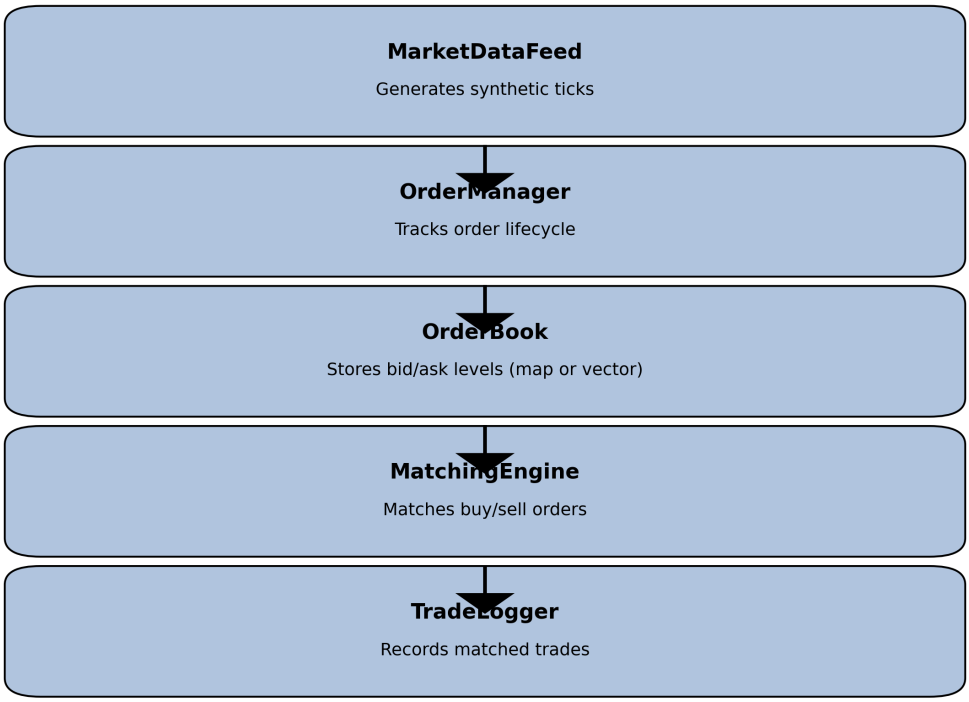

Figure: System and class flow — MarketDataFeed → OrderBook → MatchingEngine → TradeLogger.