QUANTEDGE TECHNOLOGIES

Technical Research Note

Low-Latency Event-Driven Architecture

for Real-Time Options Execution on IBKR

How we eliminated compounding feed lag, isolated execution domains, and achieved stable sub-second bar sealing at SPY market open.

QuantEdge Technologies — March 2026
research@quantedgetechnologies.com

Abstract

Real-time options execution on retail brokerage APIs presents a class of latency challenges that are fundamentally different from those found in institutional co-located environments. The bottleneck is not network round-trip time but the architecture of the Python process that consumes the feed. Python's Global Interpreter Lock, combined with synchronous callback chains, produces compounding feed lag that can reach 40+ seconds at SPY market open rendering any intraday signal based on recent data unreliable.

This note documents the architectural evolution of the QuantEdge execution engine: from a single-threaded callback model where every tick ran the full processing pipeline on the IBKR receiver thread, to a fully decoupled queue-based architecture with isolated executor domains, per-venue DOM processors, and a single source of truth for all technical indicators. The result is stable bar sealing latency of 500–1500ms at market open, down from a growing backlog that reached 45 seconds within five minutes of the bell.

No proprietary signal logic, entry criteria, or scoring parameters are disclosed. This note covers infrastructure only.

1. The Problem: Compounding Feed Lag

1.1 Observed Symptom

During development, the engine produced the following log pattern within minutes of market open:

```
09:35:15  ⏱ [SEAL_1s] SPY  feed_lag_ms=5,200
09:35:20  ⏱ [SEAL_1s] SPY  feed_lag_ms=9,800
09:35:35  ⏱ [SEAL_1s] SPY  feed_lag_ms=23,400
09:36:00  ⏱ [SEAL_1s] SPY  feed_lag_ms=42,700
```

feed_lag_ms is defined as wall_clock_now − tick_timestamp. A value of 42,700ms means the engine is processing ticks whose exchange timestamp is 42.7 seconds in the past. All signals derived from recent bar data are operating on stale information.

The critical observation is that the lag grew monotonically. This is the signature of a producer-consumer imbalance: ticks arrive faster than they are consumed, and the backlog accumulates without bound.

1.2 Root Cause Analysis

The IBKR API delivers market data through callback methods on a single receiver thread. In the original implementation, tickByTickAllLast — called on every trade print — executed the full processing pipeline synchronously before returning:

```
tickByTickAllLast():
    on_tick_by_tick_all_last()   # hygiene, side inference, snapshot
    ingest_event(row)             # route into 6 interval bins
    push_burst(sym, row)          # burst router
    on_trade_row(row)             # exit pipeline callback
    option_telemetry loop         # iterate all contract states
```

At SPY market open, the exchange generates 300–500 trade prints per second. Each callback executing 3–15ms of processing produces a receiver backpressure of 1–15ms per second. Over 5 minutes this becomes tens of seconds of accumulated lag.

The same pattern repeated in the DOM feed: updateMktDepthL2 called DOMFeatureEngine.on_dom_update() synchronously for each of three venues (ARCA, ISLAND, IEX), recomputing the full multi-venue aggregated snapshot on every Level 2 book update.

A third source of receiver-thread blocking was the option chain roll mechanism. When SPY price crossed a strike threshold, on_price_update() called _roll() synchronously, which called qualify_contract_blocking() with a 4-second timeout for each new strike entirely on the IBKR callback thread.

2. The Architecture: Queue-First, Process-Later

2.1 Core Principle

The receiver thread has one job: accept data from IBKR and return as fast as possible. Any processing whether indicator computation, option telemetry, DOM feature aggregation, or file I/O — must happen on a thread or process that the receiver does not wait for.

This is implemented through a queue-first architecture. Every IBKR callback does the minimum possible work: validate the message, extract the payload, and put it in a queue. Dedicated processor threads drain those queues independently of the receiver.

2.2 Tick Processing

tickByTickAllLast was reduced to a single queue operation:

```
def tickByTickAllLast(self, reqId, tickType, time_, price, size, ...):
    # HOT PATH — one operation, return immediately
    _TICK_Q.put_nowait((reqId, tickType, time_, price, size, ...))
```

A dedicated TickProcessor thread drains _TICK_Q and runs the full pipeline: tick hygiene, quote latch merge, side inference, ingest_event routing into the event-time bin sealer, burst router dispatch, and option telemetry updates. The receiver thread now spends approximately 100 nanoseconds per tick regardless of downstream processing time.

2.3 DOM Processing

Three exchange venues deliver simultaneous Level 2 updates during a sweep: ARCA, ISLAND, and IEX. In the original single-queue model, venue updates serialized behind each other. During a sweep, ISLAND's updates would wait for ARCA's DOMFeatureEngine computation to complete before being processed.

The solution is per-venue queues with dedicated processor threads:

```
_DOM_QS = {
    "ARCA":    Queue(maxsize=50_000),
    "ISLAND":  Queue(maxsize=50_000),
    "IEX":     Queue(maxsize=50_000),
    "DEFAULT": Queue(maxsize=50_000),
}
```

Each venue routes to its own queue. Four DOMProcessor threads drain in parallel. During a sweep, all three venues compute simultaneously rather than sequentially. The DOMFeatureEngine.on_dom_update() implementation reads all venue books under a single lock and computes the merged multi-venue snapshot so each venue trigger always produces a complete, up-to-date aggregated view.

2.4 Option Chain Roll

Option chain rolling resubscribing to new strikes when price moves requires contract qualification via IBKR's reqContractDetails. In the original implementation this ran synchronously on the price-update callback with a 4-second timeout per strike.

Two changes were applied. First, the roll hook was throttled: the _price_roll_hook fires on every trade print, but the chain manager's on_price_update is called at most 4 times per second regardless of tick rate. Second, when a roll is triggered, _roll() is dispatched to a background thread. The callback returns immediately after updating the anchor ATM; qualification and subscription happen asynchronously.

The startup qualification path was also optimized. build_qc_contracts_from_occ() qualifies 8 contracts at startup. Previously these qualified contracts were discarded, and OptionChainManager.initialize() re-qualified the same contracts. A seed_qual_cache() call now pre-populates the chain manager's cache before initialize() runs, eliminating redundant reqContractDetails requests at the most congested moment.

2.5 Executor Isolation

The engine uses three dedicated ThreadPoolExecutor instances, each owning a distinct domain of work:

Executor	Owns	Isolation Benefit
`EXEC_BINS`	Market data feed, order checks, bundle prefetch	Infrastructure never starves trading logic
`EXEC_ENTRY`	Scheduler, entry/exit evaluation, micro-entry loop	Trading decisions always get dedicated threads
`EXEC_TELEMETRY`	Stop-loss checks, watchdog, metrics arm loop	Monitoring cannot block execution
`TickProcessor`	Full tick pipeline off IBKR receiver thread	Receiver returns in nanoseconds per tick
`DOMProcessor×4`	Per-venue DOM feature computation (ARCA/ISLAND/IEX/DEFAULT)	Venues process in parallel during sweeps

Table 1. Executor domains and their isolation benefits.

Before this separation, a slow stop-loss check or a prefetch DataFrame rebuild could starve an entry evaluation. With isolated executors, each domain has guaranteed thread capacity and cannot be blocked by activity in another domain.

3. Event-Time Binning and the Partial Bar

3.1 Why Wall-Clock Bars Fail at Open

Traditional bar construction uses wall-clock boundaries: a 1-minute bar closes at 09:31:00 regardless of when ticks arrived. Under feed pressure, a wall-clock bar may contain ticks that were delivered late, producing a bar whose timestamp says 09:30 but whose most recent tick arrived at 09:31:45. Indicator values computed on such a bar are meaningless for intraday decision-making.

The QuantEdge engine uses event-time binning: each bar's boundary is defined by tick timestamps, not wall clock. A 1-second bar assigned to 09:30:15 contains only ticks with exchange timestamps in [09:30:15, 09:30:16). The bar seals when the engine's watermark defined as last_received_tick_timestamp − WATERMARK_LAG_MS advances past the bar boundary.

3.2 Six Independent Intervals

The bin sealer maintains six independent interval bins simultaneously: 1s, 5s, 15s, 30s, 1m, and 5m. Every incoming tick is routed into all six bins. Each interval has its own dedicated seal worker thread

and its own in-memory bar store. A 1m bar and a 5s bar are computed from the same raw ticks independently; no resampling or aggregation from finer intervals is performed.

This is a critical property for microstructure analysis. Aggregating a 1m bar from twelve 5s bars produces correct OHLC but incorrect flow metrics: cumulative delta, volume-at-aggressor, and tape speed cannot be correctly derived from already-aggregated data. Each interval must be computed from the raw tick stream.

3.3 Live Partial Bar Snapshots

The get_partial_bin_df() function returns a DataFrame containing the last N sealed bars plus the currently-forming partial bar, computed in real time from the ticks accumulated so far in the open bin. This means that at any moment during a 1-minute bar whether 5 seconds or 55 seconds into it the engine has a live snapshot of the bar's current state with all indicators applied.

The system is event-driven, not time-driven. Entry evaluations are triggered by bar close events from the seal workers, not by timers. When a 15s bar seals, the scheduler receives a signal via new_bar_queue and evaluates entries using the freshest data available across all six intervals, including their current partial bars.

4. Single Source of Truth for Indicators

4.1 The Recomputation Problem

In early versions of the engine, technical indicators were recomputed in multiple places: in the seal path, in the bundle prefetch threads, in combine_data_5s() called from the heartbeat loop, and in the entry evaluation functions. Each recomputation ran ewm() across 120 bars the EMA, MACD, and eight flow metric series. With the heartbeat running at 4Hz, this produced approximately 40 pandas ewm() calls per second, each competing with the bin sealer for Python's GIL.

The consequence was periodic GIL starvation: the seal worker would acquire the GIL to process a sealed bar, find it immediately contested by a heartbeat thread running ewm(), and stall. This produced the characteristic pattern of lag spikes that recovered between heartbeat cycles.

4.2 _enrich_df as the Single Computation Point

All indicator computation was consolidated into a single function, _enrich_df(), which runs once per sealed bar on the full historical window at read time. _enrich_df computes:

- EMA-9 and EMA-21 on Close_filled
- MACD (12, 26, 9) Line, Signal, and Histogram
- RSI-14 using Wilder smoothing

- EMA-9 and EMA-21 for Dom_HitRate, DirectionalPressure, DeltaPrice, and five additional flow series
- VWAP (session accumulator from bin_core, or cumsum fallback)

_enrich_df is called once: when get_5s_df() or get_1s_df() reads bars from the in-memory store. Every consumer of this data the entry gate, the stop-loss check, the bundle prefetch, the plot reads already-enriched data. No consumer recomputes any indicator. The combine_data_5s() and combine_data_1s() functions are pure pass-throughs that slice the last 120 bars and add Close_filled via forward-fill, nothing more.

4.3 Indicator Consistency

Because all indicators are computed from the same function on the same data store, there is no possibility of two parts of the engine seeing different EMA values for the same bar. The 5s EMA-9 visible to the entry gate is the same value visible to the stop-loss monitor and the plot.

This property is particularly important for multi-timeframe confluence checks, where the engine must compare EMA alignment across 1s, 5s, 15s, 30s, 1m, and 5m simultaneously. Inconsistent indicator values across timeframes would produce false confluence signals.

5. Observed Results

5.1 Seal Latency

The primary measure of pipeline health is seal_delay_ms: the elapsed time from a bar's close boundary to the moment the seal worker writes the bar to the in-memory store. Before the architectural changes, this metric grew monotonically from approximately 5 seconds at open to 45 seconds within five minutes. After the changes, seal_delay_ms stabilized in the range of 500–1500ms throughout the trading session, including at the open bell.

A residual 4–5 second delay was observed consistently at open and attributed to IBKR's own feed delivery latency the exchange timestamp on ticks was 4–5 seconds behind wall clock during the opening auction. This is an upstream characteristic of the IBKR tick-by-tick feed at open and is not addressable in the client application.

5.2 Scheduler Cycle Times

Entry evaluation cycles the time from receiving a new bar signal to completing all entry and exit checks dropped from 0.75–1.92 seconds to a consistent 0.00–0.25 seconds. The previous cycle times were dominated by combine_data() rebuilding DataFrames with 30-bar synthetic backfills and recomputing all indicators on every cycle. With cached bundles and pre-enriched data, cycles complete in milliseconds.

5.3 Latency Summary

Component	Before (ms)	After (ms)
tickByTickAllLast callback	3–15 ms per tick	< 0.1 ms (queue put)
DOM feature recompute	Serial across 3 venues	Parallel, per-venue
Roll qualify (on sweep)	4s per strike, blocking	Async background thread
Roll hook call rate	Every tick (~20/s at open)	4 Hz throttled
combine_data_5s EMAs	10 ewm() calls/second	Zero (read from store)
Seal delay at open	Growing: 5s → 45s+	Stable: 500–1500 ms

Table 2. Per-component latency before and after architectural changes.

6. Architecture Summary

The full data flow from exchange to execution decision:

```
IBKR Exchange
    │
    ├─ tickByTickAllLast  →  _TICK_Q.put_nowait()    [~100ns]
    └─ updateMktDepthL2   →  _DOM_QS[venue].put_nowait()  [~100ns]

TickProcessor (thread)
    └─ tick hygiene → ingest_event → push_burst → option telemetry

ibkr_event_bins
    └─ 6 SealWorker threads (1s/5s/15s/30s/1m/5m, independent)
        └─ _aggregate(records) → _process_sealed_bar()
            ├─ append_Xs_bar()    →  in-memory store
            ├─ _IO_Q              →  CSV writes (async)
            ├─ _NOTIFY_Q          →  SR builder
            └─ new_bar_queue      →  scheduler trigger

DOMProcessor-ARCA/ISLAND/IEX/DEFAULT (4 threads)
    └─ DOMFeatureEngine._compute() → merged multi-venue snapshot

EXEC_BINS  →  market data | prefetch | order management
EXEC_ENTRY →  scheduler | entry gates | micro-entry loop
EXEC_TEL   →  stop-loss | watchdog | metrics

get_partial_bin_df() → _enrich_df() → all indicators (one call)
get_cached_bundle()  → all 6 intervals with partial bars
```

The architecture is observable at every layer. Seal latency, cycle times, and queue depths are logged continuously. The system is designed so that any future performance regression produces a clear, attributable signal rather than a diffuse slowdown.

7. Conclusion

The architectural improvements documented here were driven entirely by observing production behavior under live market conditions and tracing each bottleneck to its root cause. No speculative optimization was applied; every change addressed a measured problem with a specific, testable fix.

The resulting system processes SPY at market open with stable latency, correct multi-timeframe data, and full indicator consistency across all execution paths. This infrastructure forms the foundation for the continued development of the QuantEdge execution strategy research program.

Future notes in this series will address regime detection under live microstructure conditions, signal robustness across intraday volatility environments, and multi-instrument extension of the current SPY/options stack.

Disclosure

This document describes infrastructure and systems architecture only. No entry signals, scoring parameters, option selection criteria, risk thresholds, or proprietary execution logic are disclosed. QuantEdge Technologies does not provide investment advice. Past system performance does not guarantee future results.