Print this page to PDF (Ctrl/Cmd+P) — hand to CTO prospects Print to PDF [Full case study]

Sevendyne
Engineering firm · Since 2016
Technical one-pager

Trading System API — Qt/C++ operator desk

Engagement: End-to-end product build **Stack:** C++ · Qt · native trading libs **Complexity:** Concurrency · UI latency · order state

Native Qt desk wired into proprietary C++ trading libraries — not a web shell. Three-thread model: ingest, strategy, UI via queued snapshots.

C++QtQueuedConnectionMulti-window desk
Architecture

How data moves

Ingest thread normalizes ticks → strategy thread mutates books and emits order intents → UI thread renders from immutable snapshots. UI never holds locks on algo memory.

Design decisions

Why Qt/C++ over Electron

Decision	Rejected	Chosen because
Desktop shell	Electron, WPF	Direct C++ lib link; dense grid perf; no IPC latency
UI updates	Mutex every tick	Queued signals; coalesce bursts before repaint
Order state	Global lock	Per-instrument single writer
Widgets	Live pointers	Snapshot copies — safe during realloc

Code shape

Strategy → UI (representative)

```
void BookEngine::onTick(const Tick& t) {
  applyToBook(t);
  if (shouldRepaint(t.symbol))
    emit bookUpdated(t.symbol, snapshot(t.symbol));
}
// UI thread: render immutable BookSnap — no locks
```

Reliability

Failure handling

On feed disconnect: stale banner + block new orders until resync. Operators never trade on partial book state.

Production outcomes

Results

- Sub-second refresh on active symbols during peak tape
- Zero UI-thread blocking from market I/O post-refactor
- Foundation for later Qt/embedded programmes

Client names anonymized where required. Metrics from production monitoring during engagement.

contact@sevendyne.com · sevendyne.com/case-studies