Tick Data to Signals

Real-Time Crypto Analytics on KDB-X

Draft Version 0.4

Philippe DAMAY

January 2026

Abstract

This paper presents a real-time cryptocurrency analytics system built on kdb+, implementing the reference architecture from Data Intellect's *Building Real-Time Event-Driven KDB-X Systems*. The system The system ingests live Binance market data and computes intraday analytics—VWAP, volatility, correlation, order book imbalance,

Equally important is a second contribution: a documentation methodology designed for AI-assisted comprehension. Structured markdown, Architecture Decision Records, and rewritten reference materials form a queryable knowledge base that enables Large Language Models to reason about system behavior and design intent. All documentation is available on GitHub.

The result is not an academic prototype but a production-oriented reference implementation emphasizing observability, recoverability, and explainability—with documentation engineered for both human comprehension and AI-assisted querying.

Contents

1 Introduction

1.1 Purpose and Positioning

This paper is intentionally not academic. Its objective is to demonstrate how a real-time, event-driven market data system can be implemented in practice using C++ and kdb+, following an established reference architecture, and how such a system can be made understandable, maintainable, and extensible beyond its initial implementation.

The work is based on the reference whitepaper *Building Real-Time Event-Driven KDB-X Systems* by Data Intellect, which describes architectural patterns and design principles . Rather than proposing new theory, this paper focuses on translating those principles into a concrete, running system that ingests live cryptocurrency market data, computes intraday analytics, and exposes operational and analytical visibility in real time.

A central motivation for this work is the observation that many production kdb+ systems, while technically sound, are difficult to reason about once they grow beyond their original scope or team. Architecture is often implicit, decisions are undocumented, and understanding the system requires direct access to the original developers. This paper takes the position that implementation alone is insufficient: design intent, constraints, and trade-offs must be captured explicitly as part of the system.

Documentation is treated as seriously as code. Architecture Decision Records, specifications, and design notes are written in structured formats that work equally well for human developers and Large Language Models. This paper covers both the technical system and the documentation methodology that keeps it queryable and explainable.

This paper is intended for kdb+ practitioners seeking a concrete implementation of established patterns, system developers interested in documentation practices that scale, and newcomers looking for guidance on building real-time data systems that remain understandable beyond their initial implementation.

1.2 System Scope and Constraints

The system ingests real-time market data from Binance, one of the largest cryptocurrency exchanges by volume. Two data streams are consumed:

- A trade stream delivering every individual transaction as it occurs
- A depth stream providing sampled order book snapshots at 100-millisecond intervals

From this data, the system computes a set of intraday analytics: Volume-Weighted Average Price (VWAP) over configurable windows, realized volatility and pairwise correlations derived from a rolling variance-covariance matrix, and Order Book Imbalance (OBI) with exponential smoothing. These metrics are computed in real time as data arrives and are visualized through purpose-built dashboards. The analytics are descriptive, not predictive: the system makes no trading decisions, generates no signals for execution, and maintains no position or PnL state.

The system runs on a WSL environment with 12 CPU threads and 7.6GB of RAM—modest by

any measure. This choice reflects a philosophical position: that thoughtful architecture should achieve production-grade performance without enterprise-scale infrastructure. The constraint also serves a practical purpose, forcing design discipline that would be easy to avoid with unlimited resources.

Several capabilities are explicitly out of scope. There is no historical database or end-of-day persistence beyond log files. There is no integration with execution systems or order management. There is no support for multiple exchanges or asset classes.

1.3 Documentation and Generative AI as Design Constraints

A recurring problem in production kdb+ systems is that they become opaque over time. The original developers understood the design, but that understanding was never externalized. When those developers move on, the system continues to run, but the ability to reason about it—to extend it safely, to debug unexpected behavior, to onboard new contributors—degrades. Documentation, if it exists at all, tends to describe what the code does rather than why it was written that way.

This paper treats documentation not as an afterthought but as a design constraint. Every significant architectural decision is captured in an Architecture Decision Record (ADR) that explains the context, the choice made, and the consequences accepted. Specifications for data schemas, APIs, and latency measurement are maintained in structured markdown files. Together, these artifacts form a knowledge base that can be queried rather than merely searched.

The choice of markdown is deliberate. Large Language Models work best with structured, plain-text formats that preserve semantic relationships. A PDF with embedded diagrams is opaque to an LLM; a markdown file with textual descriptions of those same diagrams is immediately useful. For this reason, the Data Intellect reference whitepaper—originally a formatted PDF—was fully rewritten into LLM-friendly markdown, with every diagram converted to textual description.

The goal is not to replace human understanding but to augment it. When a developer asks "why does the RTE use trade buckets instead of storing raw ticks?", the answer should be retrievable—either by a human reading the relevant ADR or by an AI assistant citing it. When implementation deviates from the reference architecture, that deviation should be flagged, whether by a code reviewer or by an LLM comparing documentation to code. Designing for AI legibility turns out to be designing for long-term human comprehension as well.

1.4 Structure of the Paper

This paper does not follow the conventional structure of architecture first, then implementation, then results. Instead, it begins with the running system and works backward to the decisions that shaped it. This ordering is intentional.

Section 2 presents the system in action: four dashboards that visualize feed handler health, dataflow latency, market data, and computed analytics. Starting here establishes a concrete mental model before any abstraction is introduced. The reader sees what the system produces before learning how it produces it.

Section 3 explains the characteristics and limitations of the input data—why the distinction between true tick data and sampled snapshots matters for analytical accuracy.

Section 4 addresses documentation and AI-assisted comprehension. Placing this before the technical architecture may seem unusual, but it reflects the actual development sequence: documentation was written first, and implementation followed. This section explains the methodology that makes the rest of the paper—and the system itself—queryable by both humans and machines.

Section 5 provides the architecture overview: data flow, component responsibilities, and the design principles that guided construction. Section 6 details the C++ feed handlers, covering the state machine for order book synchronization, cache-efficient storage, latency instrumentation, and reconnection logic. Section 7 covers the kdb+ layer: tickerplant logging, crash recovery via log replay, trade bucketing for VWAP, variance-covariance computation, and telemetry aggregation.

Section 8 presents observed performance metrics. Section 9 concludes with lessons learned.

Throughout, the paper emphasizes not just what was built, but why it was built that way—and how those decisions were captured for future reference.

2 The System in Action

Before examining architecture or code, it is worth seeing what the system delivers. This section walks through four dashboards that together provide both the observability needed to trust the pipeline and the real-time signals it was built to produce.

Each dashboard is designed around a single operational question. The goal is not to expose every available metric, but to surface the specific information needed to answer that question quickly and confidently. In most cases, this means combining a status view (what is the current state?) with a trend view (how is it changing?). The status view enables rapid detection; the trend view enables diagnosis and pattern recognition.

The progression of dashboards mirrors the data flow itself: from ingestion at the feed handlers, through the internal pipeline, to the market data and derived analytics that downstream consumers rely on. If the early stages are healthy, the later stages can be trusted. If something degrades upstream, the downstream dashboards reveal the consequences.

2.1 Feed Handler Monitoring

The feed handler is the first point of contact with external reality—and the first place where failures occur. If ingestion degrades, every downstream component continues to operate on incomplete or stale data, often without immediate error.

This view answers: *is market data arriving correctly and on time?*

A compact status grid shows connectivity, message throughput, and recency of received data for each handler. Latency trend charts track the 95th percentile of parsing and publish latencies over a rolling window. The choice of P95 over averages is deliberate: average latency masks the

brief but repeated slowdowns that tend to coincide with market stress, precisely when timely data matters most.

By decomposing latency into parsing and transmission components, the system makes attribution explicit. Elevated parsing latency points to CPU pressure or inefficient deserialization; increased send latency suggests backpressure from the tickerplant or IPC contention. This separation reduces mean time to diagnosis when conditions degrade.

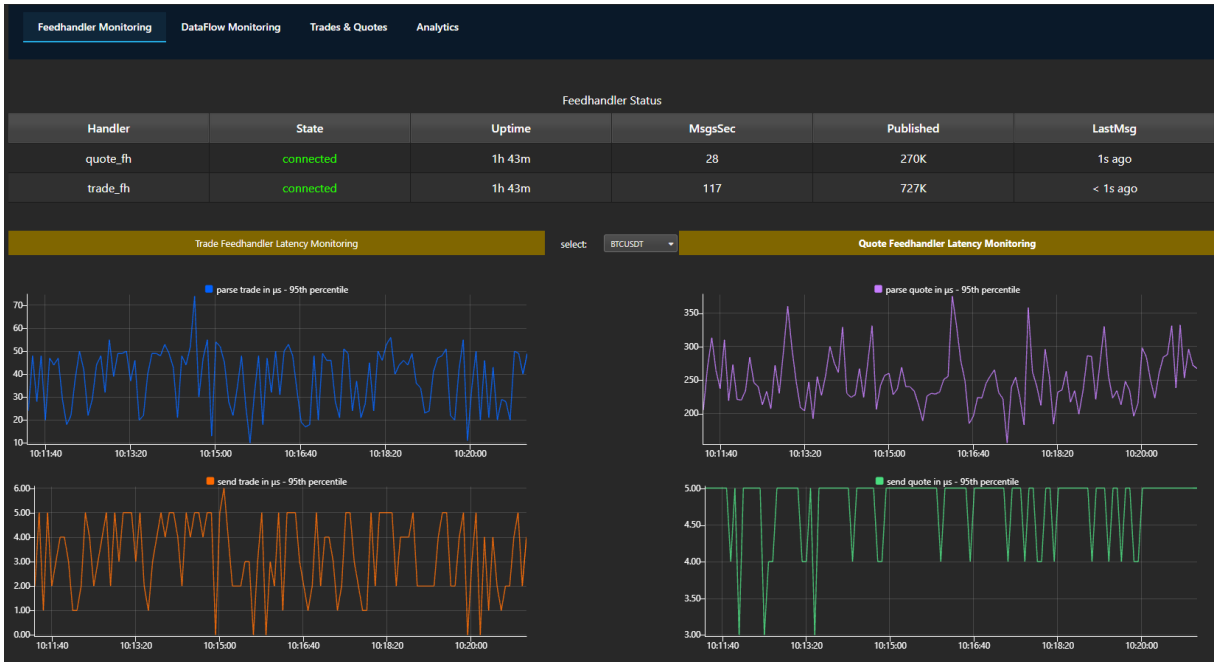

Figure 1: Feed Handler Monitoring dashboard showing health status grid (top) and 95th percentile latency charts (bottom) for trade and quote handlers.

2.2 Dataflow Monitoring

Once data leaves the feed handler, it enters an internal pipeline where each hop—tickerplant, RDB, analytics engine—adds latency and introduces potential failure modes. A message that arrives promptly at ingestion can still reach storage late, or not at all, if any intermediate stage degrades.

This view answers: *where is latency accumulating, and is the pipeline keeping pace with incoming volume?*

Status grids display data volume per symbol and memory usage per kdb+ process. Segment latency charts decompose end-to-end latency into discrete hops: feed handler to tickerplant, tickerplant to RDB. When end-to-end latency spikes, this decomposition narrows the search immediately—if FH→TP is stable but TP→RDB spikes, the issue lies in tickerplant logging or RDB processing, not ingestion.

Trade and quote flows are tracked separately. Quote messages are larger and arrive more frequently; their latency profile differs structurally. Tracking both reveals whether degradation is payload-specific or systemic.

Figure 2: Dataflow Monitoring dashboard showing data volume and system resources (top) with segment latency breakdown for trades and quotes (bottom).

2.3 Trades and Quotes View

Order book grids display current L5 depth for each tracked symbol. Candlestick charts show price evolution over 5-minute intervals, computed directly from the trade stream. The juxtaposition is intentional: order books represent intention (where participants are willing to transact), price charts represent execution (where transactions actually occurred). Seeing both together reveals whether captured data aligns with expected market dynamics.

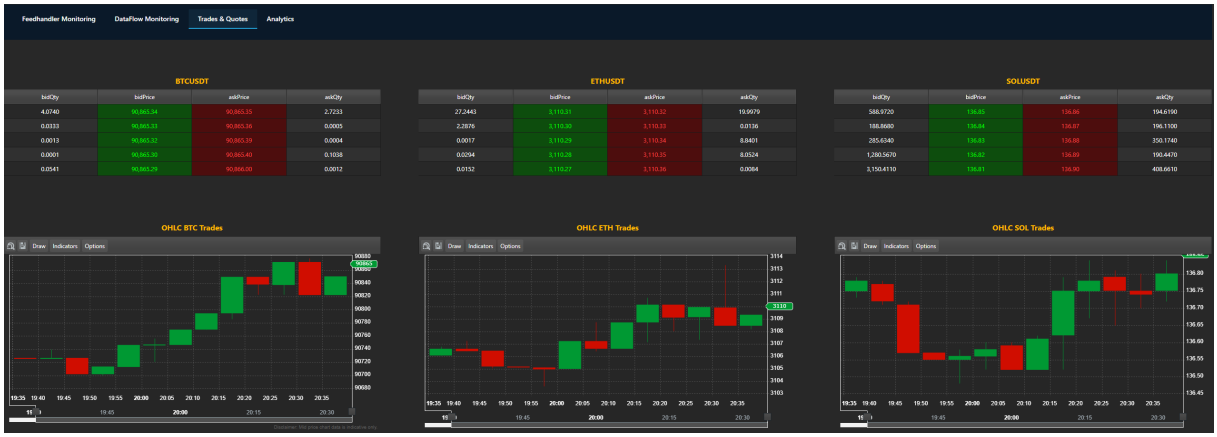

Figure 3: Trades and Quotes dashboard showing L5 order books for BTC, ETH, and SOL (top) with corresponding 5-minute OHLC candlestick charts (bottom).

2.4 Analytics View

Raw market data has limited utility without transformation and contextualization. The value of a real-time system lies in transforming continuous data streams into metrics that support interpretation and decision-making.

This view answers: *what is the market telling us right now?*

Summary grids display VWAP-based momentum, realized versus implied volatility, and pairwise

correlations. An order book imbalance section shows the current balance of buying and selling interest, with historical context from a rolling time-series chart.

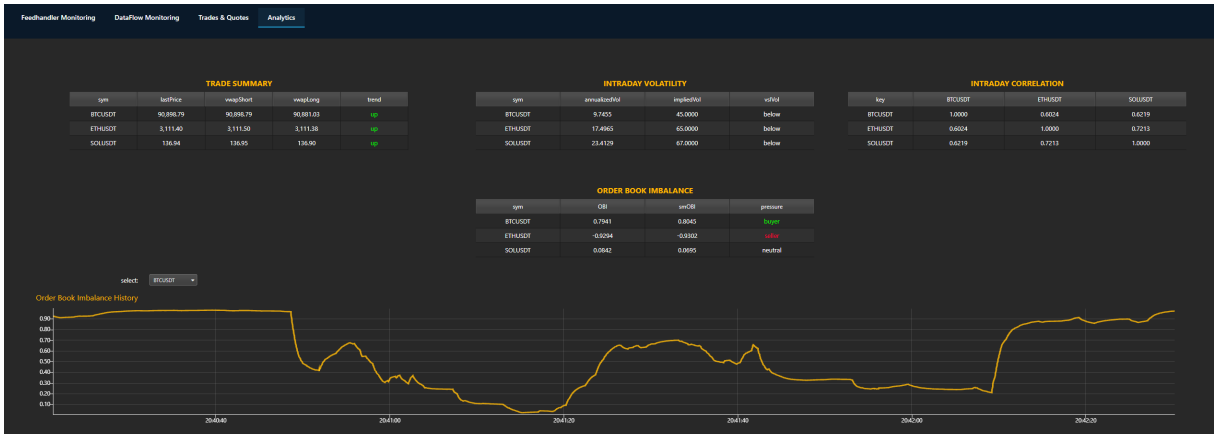

Figure 4: Analytics dashboard showing Trade Summary, Intraday Volatility, and Correlation matrices (top) with Order Book Imbalance grid and 5-minute history chart (bottom).

3 Understanding the Data

It is necessary to understand exactly what data the system processes. This is not a formality. The granularity and characteristics of input data determine what analytics are meaningful, what precision can be claimed, and where interpretation must be cautious. A system that treats all market data as equivalent will produce metrics that appear precise but are not.

3.1 Trade Data: True Tick Data

The trade stream from Binance delivers true tick data—every individual transaction as it occurs, in real time. When a buyer and seller match on the exchange, Binance immediately pushes a message containing the trade ID, price, quantity, and timestamp. Nothing is aggregated, batched, or sampled. If 500 trades execute in one second, the system receives 500 discrete messages.

Listing 1: Binance trade stream payload

```
{
  "e": "trade",       // Event type
  "E": 1672515782136, // Event time (exchange timestamp)
  "s": "BNBBTC",      // Symbol
  "t": 12345,         // Trade ID (unique, sequential)
  "p": "0.001",       // Price
  "q": "100",         // Quantity
  "T": 1672515782136, // Trade time
  "m": true,          // Is the buyer the market maker?
  "M": true           // Ignore
}
```

Each trade message represents an atomic event: a specific quantity changed hands at a specific price at a specific moment. The trade ID field provides a strictly increasing sequence number,

enabling detection of gaps or out-of-order delivery. The `m` flag indicates whether the buyer was the maker (passive side)—equivalently, whether the trade was initiated by a seller hitting a bid. This distinction is useful for analyzing aggressor flow.

This granularity is what makes VWAP calculations exact. The system sees every price and quantity that contributed to volume, not an approximation or sample.

3.2 Order Book Data: Sampled Snapshots

The depth stream operates on fundamentally different principles. Rather than pushing every order book change in real time, Binance aggregates changes over 100-millisecond windows and delivers the net result at each price level.

Listing 2: Binance depth snapshot payload

```
{
  "lastUpdateId": 160,  // Last update ID (for synchronization)
  "bids": [             // Bid levels
    ["0.0024", "10"]    // [price, quantity]
  ],
  "asks": [             // Ask levels
    ["0.0026", "100"]   // [price, quantity]
  ]
}
```

Consider what happens within a single 100ms window at one price level:

Time	Event	Quantity at Level
0ms	Window starts	100
20ms	+50 added	150
45ms	-80 removed	70
70ms	+30 added	100
100ms	Snapshot sent	**100**

Table 1: Order book activity within a 100ms window—only the final state is transmitted.

The system receives only that the level ended at 100. The intermediate states—the +50, -80, +30 sequence—are invisible. If a level changed and then reverted within the window, it may not appear in the update at all.

This is not tick data. It is sampled data at 10Hz. The distinction has practical implications:

- Fleeting liquidity is invisible. An order that appeared and was filled within 100ms leaves no trace.
- Order flow within windows is lost. The sequence of changes cannot be reconstructed.
- Queue position cannot be inferred. Without seeing individual order additions and cancellations, precise queue modeling is impossible.

3.3 Market Data Depth Levels

Market data is commonly categorized by the depth of information provided:

Level	Content	Use Case
L1	Best bid and ask only	Basic spread monitoring, simple execution
L2	Multiple price levels with aggregated depth	Order book imbalance, liquidity analysis
L3	Individual orders at each level	Queue position modeling, full book reconstruction

Table 2: Market data depth levels.

This system captures snapshots of L2 data at 5 levels of depth every 100ms. This is sufficient for order book imbalance calculations and liquidity visualization, but does not support queue-position analysis or precise microstructure modeling that would require L3 data.

3.4 Implications for Analytics

The asymmetry between trade and quote data shapes what analytics are meaningful and what precision can be claimed:

Metric	Data Source	Accuracy
VWAP	Trades (tick)	Exact—every trade captured
Order Book Imbalance	Quotes (sampled)	Approximate—100ms snapshots
Trade Arrival Rate	Trades (tick)	Exact
Spread	Quotes (sampled)	Up to 100ms delayed
Realized Volatility	Trades (tick)	Exact for chosen bucket size

Table 3: Analytics accuracy depends on underlying data granularity.

Order book imbalance computed from 100ms snapshots is an approximation of the instantaneous state, delayed by up to 100ms and blind to intra-window dynamics. This does not make imbalance metrics useless. For signals operating on timescales of seconds or longer, 100ms sampling is adequate. But it would be inappropriate to claim tick-level precision for book-derived metrics when the input data is fundamentally sampled.

3.5 Why This Matters

Understanding data granularity prevents two common errors: over-claiming precision and under-utilizing available information.

When designing indicators that combine trade and quote data, the temporal mismatch requires careful handling. A trade arriving at time T represents exactly what happened at T. A quote "at time T" represents the book state from the most recent 100ms snapshot—potentially up to

100ms stale. Joining these naively can produce misleading results; acknowledging the mismatch allows for appropriate interpretation.

There is a further subtlety: depth snapshots arrive independently per symbol. The 100ms update for BTCUSDT is not synchronized with the 100ms update for ETHUSDT. When comparing order book imbalance across symbols, the readings may reflect book states up to 100ms apart.

This section exists because data quality determines analytical quality. The subsequent sections describe how the system is architected and implemented, but the value of that architecture depends entirely on understanding what the data can and cannot support.

4 Documentation and Generative AI as Design Constraints

Traditional software documentation—inline comments, Word documents, Confluence pages—was designed for human readers navigating linearly. This approach was never ideal (even when documentation was good and up to date, which is unfortunately seldom the case) and becomes increasingly inadequate. Modern projects are too complex for any single developer to hold in memory. Documentation that cannot be queried becomes documentation that is not read or partially read often at best.

Generative AI changes the equation. Large Language Models can ingest entire documentation sets, answer questions about architecture, identify inconsistencies, and explain design decisions in context. But they work best with structured, plain-text formats that preserve semantic relationships. A PDF with embedded diagrams is opaque to an LLM; a markdown file with textual descriptions of those same diagrams is immediately useful. This transformation still requires significant human effort. Automated parsing tools remain unreliable for complex technical documents—they miss what matters, and worse, misinterpret what they do extract.

4.1 The Backbone: A Rewritten Reference

The first task in this project was not writing code—it was rewriting documentation. Data Intellect's whitepaper *Building Real-Time Event-Driven KDB-X Systems* provided the architectural foundation, but as a PDF with diagrams and formatted tables, it was inaccessible to LLMs. The solution was to transform it entirely: converting every diagram into textual descriptions, restructuring content into clean markdown, and ensuring that nothing was lost in translation.

This was not a quick task. Hours went into producing a document that an LLM could parse as reliably as a human expert. the rewritten reference can be found alongside the original whitepaper in the project repository.

The payoff is substantial. This document now serves as the project's architectural backbone—the authoritative source for how the system *should* behave. When implementation deviates from established patterns, the deviation should be intentional and documented. An AI assistant with access to both the reference and the codebase can identify discrepancies automatically.

4.2 Additional Documentation

The rewritten whitepaper was the kickoff, not the finish line. A living system requires living documentation that captures decisions as they are made.

4.2.1 Architecture Decision Records

Ten Architecture Decision Records document specific design choices, organized in the `decisions` folder. Examples include:

- Why flat arrays instead of nested dictionaries for order book state?
- Why 1-second buckets for VWAP rather than storing raw ticks?

Each ADR follows a consistent structure: context, decision, consequences. This format is particularly effective for LLM consumption—the model can retrieve relevant decisions when answering questions about implementation choices, and flag when new code contradicts an established decision.

4.2.2 Specification Notes

Additional markdown files cover specific topics:

- `api-binance` — Binance WebSocket API behavior and edge cases
- `quotes-schema` / `trades-schema` — Field definitions and data types
- `notes` — Practical guidance including how latency measurement is defined and implemented

Together with the ADRs, these form a queryable knowledge base rather than a static document archive. The distinction matters: archives are searched when something breaks; knowledge bases are queried before writing code.

4.3 Practical Usage with Claude Projects

Claude Projects provides a way to create a persistent AI assistant with access to project documentation. Rather than re-explaining context in every conversation, the documentation lives in the project and informs every response. The following walkthrough demonstrates how to set this up.

4.3.1 Creating a Project

From `claude.ai`, navigate to Projects in the left sidebar and click **New Project**. The creation dialog asks for a project name and description—these help organize multiple projects but do not affect the AI's behavior.

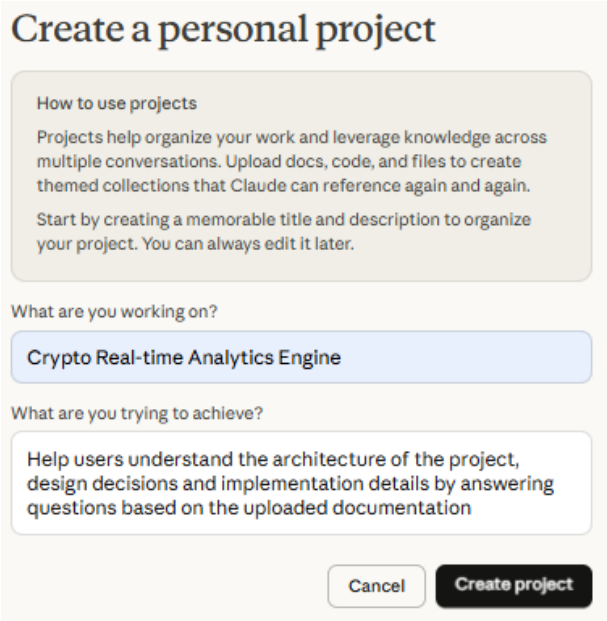

Figure 5: Creating a new project in Claude.

4.3.2 Setting Project Instructions

Project instructions tell the AI how to behave and what role to assume. For this system, the following instructions establish the AI as a project expert with clear guidance on how to respond:

Listing 3: Claude Project instructions for tick-to-signal

```
You are an expert on the tick-to-signal project, a real-time market
    data pipeline for crypto trading analytics.

Architecture:
- C++ feed handlers ingest Binance WebSocket data (trades + L5
   quotes)
- kdb+ tickerplant logs messages and distributes data to
   subscribers
- RDB stores raw trades and quotes, supports log replay
- RTE computes analytics: VWAP, volatility, correlation, OBI
- TEL aggregates latency metrics and monitors feed handler health
- KX Dashboard visualizes real-time metrics across four views

Your knowledge base includes:
- paper/: Academic white paper on the system
- decisions/: Architecture Decision Records (ADR-001 through ADR
   -009)
- specs/: Data schemas and APIs
- reference/: External references (including rewritten Data
   Intellect whitepaper)
- notes/: Implementation notes

When answering:
```

```
- Cite the specific document (e.g., "per ADR-004")
- Cross-reference between documents when relevant
- If information isn't documented, say so explicitly
- When implementation deviates from the reference architecture,
  flag it
```

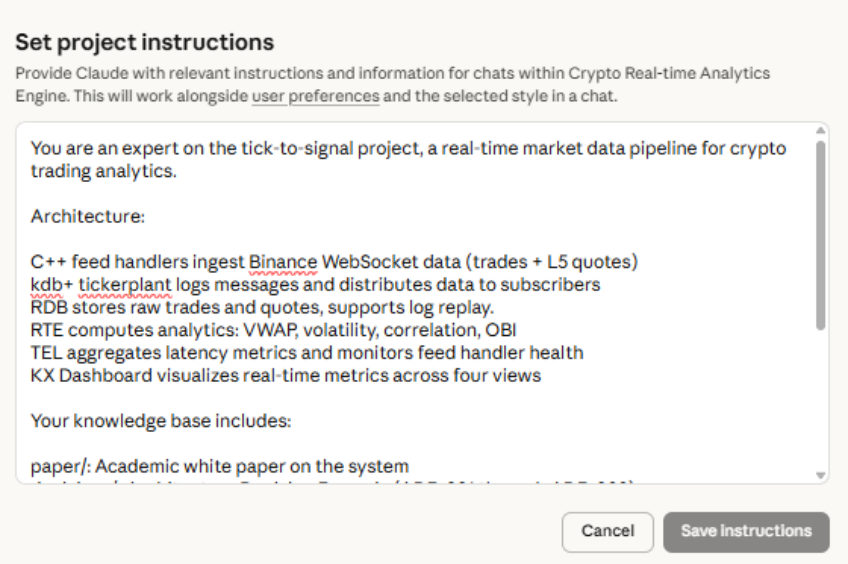

Figure 6: Configuring project instructions to establish the AI's role and behavior.

4.3.3 Attaching Documentation

Upload all markdown files from the repository: the rewritten whitepaper, ADRs, specification notes, and any additional context. Claude indexes these files and references them when responding. The more complete the documentation, the more accurate and grounded the responses.

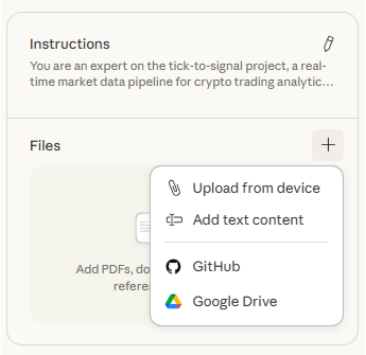

Figure 7: Uploading project documentation to Claude's knowledge base.

4.3.4 Querying the System

With documentation loaded, the assistant can answer architectural questions, explain design decisions, and identify potential issues. Questions like "How does data flow from Binance to the RDB?" or requesting "List all the tables available and the purpose of each" receive grounded

answers as shown below.

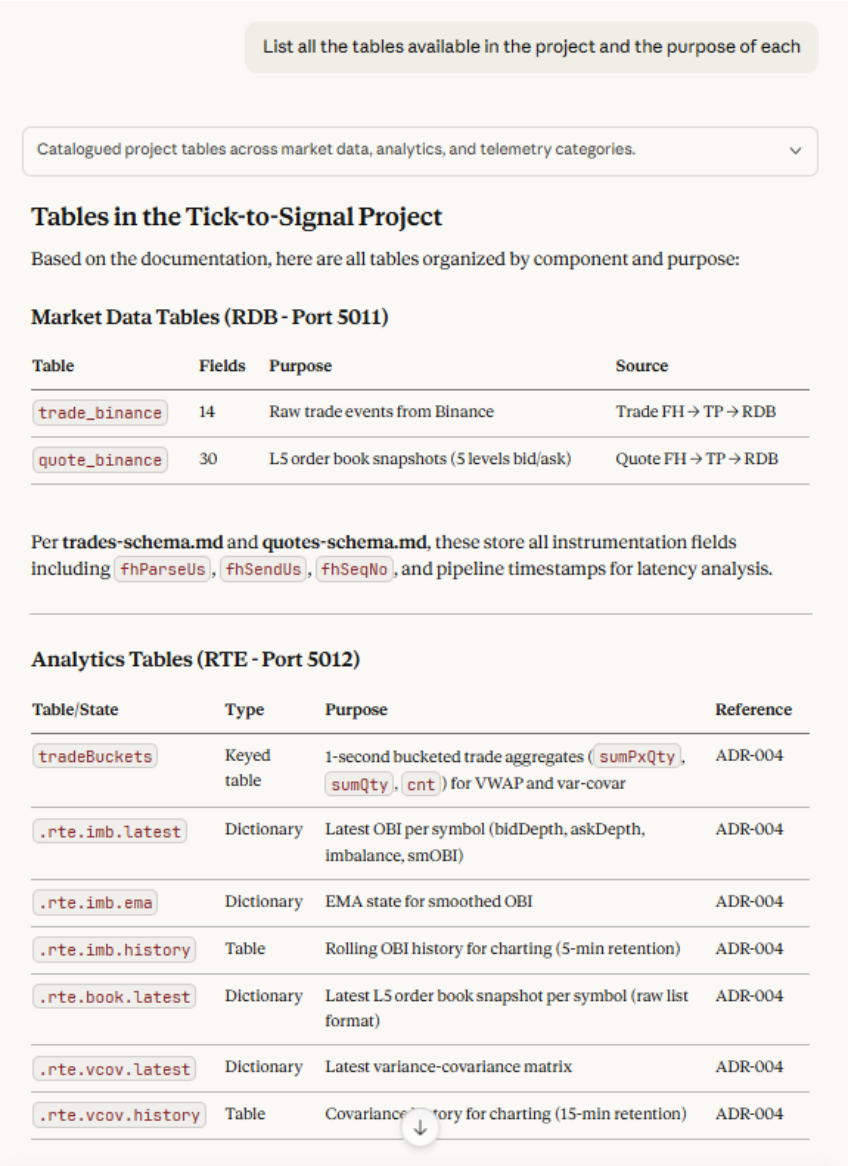

Figure 8: Querying the AI assistant about system architecture.

The engineering payoff is concrete. When implementing new features, describing the approach to the AI often surfaces conflicts with existing decisions before code is written. When debugging unexpected behavior, the AI can trace through documented data flows to identify where assumptions might be violated. When onboarding—or simply returning to the project after time away—a conversation with the AI reconstructs context faster than reading through documentation linearly.

This is not a replacement for understanding the system. It is an augmentation that makes the full context of the project accessible at conversational speed.

5 Architecture Overview

The system follows a classic kdb+ tick architecture: feed handlers capture and normalize external data, a tickerplant logs and distributes it, and downstream processes store, analyze, and monitor.

5.1 Data Flow

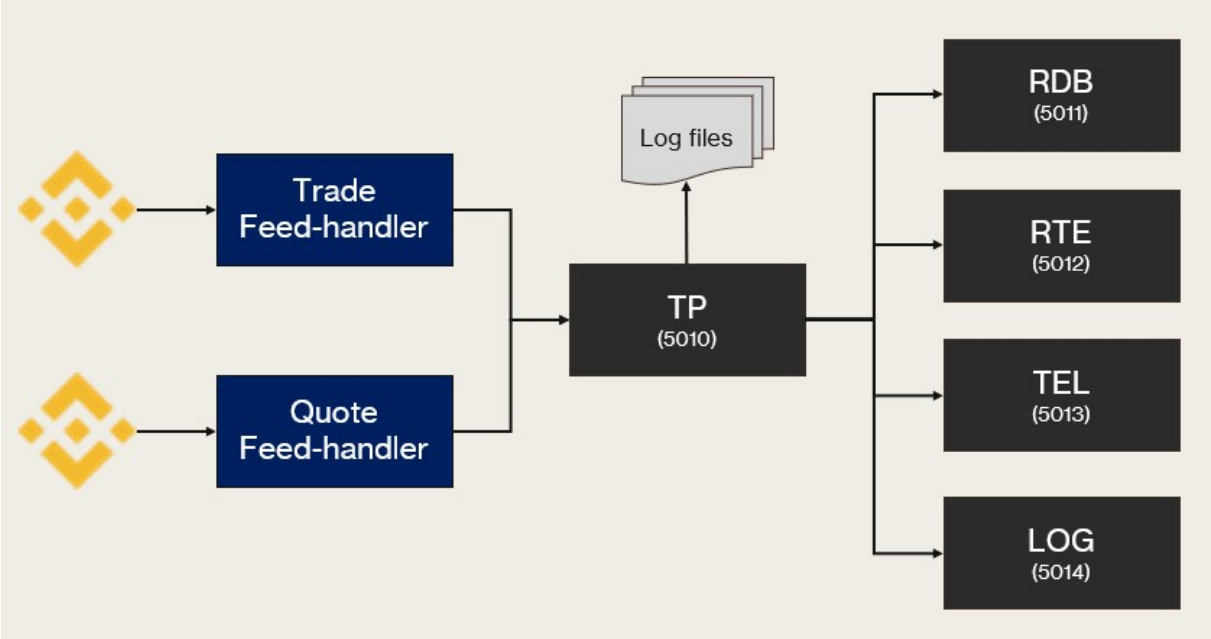

Figure 9: System architecture: data flows from Binance through feed handlers to kdb+ components

Market data enters via two WebSocket connections to Binance—one for trades, one for order book depth updates. Each stream is handled by a dedicated C++ feed handler that normalizes the data, adds timing instrumentation, and publishes to the tickerplant via asynchronous IPC.

The tickerplant serves as the central hub: it logs every message to disk and fans out to subscribers. Downstream, the RDB stores raw events, the RTE computes analytics, and TEL aggregates telemetry. A log manager handles retention and integrity verification.

5.2 Component Responsibilities

Component	Port	Responsibility
Trade FH	–	Captures trades from Binance, adds timestamps, publishes to TP
Quote FH	–	Maintains L5 order book via snapshot + delta reconciliation
TP	5010	Logs messages to disk, distributes to subscribers
RDB	5011	Stores raw trades and quotes, supports replay from logs
RTE	5012	Computes VWAP and order book imbalance in real-time
TEL	5013	Aggregates latency metrics, monitors feed handler health
LOG	5014	Manages log retention, verifies integrity

5.3 Design Principles

Several principles from the Data Intellect whitepaper guided this architecture:

One job only. Each process has a single responsibility. The tickerplant logs and distributes—it does not compute analytics. The RTE computes analytics—it does not store raw data. This separation simplifies debugging and allows independent scaling.

Fail fast. Processes validate their environment on startup and exit immediately if dependencies are missing. A feed handler that cannot connect to the tickerplant does not silently buffer—it logs the failure and retries with backoff.

Event-driven. Analytics are computed on update, not on timers. When a trade arrives, VWAP buckets update immediately. When a quote arrives, order book imbalance recalculates. This minimizes latency between data arrival and derived metric availability.

Recoverable. Both RDB and RTE replay from tickerplant logs on startup. If a process crashes mid-day, it restarts with full state reconstruction—no manual intervention required.

Observable. Every message carries latency instrumentation. TEL aggregates these into percentiles every 5 seconds. Feed handlers publish health metrics. The system is designed to reveal its own performance characteristics.

5.4 Why Two Feed Handlers?

Trades and quotes have fundamentally different semantics:

- **Trades** are discrete events. Each message is final and append-only. Processing is stateless—parse, timestamp, publish.

- **Quotes** represent evolving state. The order book must be reconstructed from a REST snapshot and maintained through delta updates. Processing requires a state machine with sequence validation.

Combining these into a single process would mix simple and complex code paths, couple their failure modes, and complicate debugging. Separate handlers keep each focused and independently recoverable.

6 C++ Feed Handler Implementation

The system comprises two dedicated feed handlers—one for trades, one for L5 order book quotes—that ingest real-time market data from Binance WebSocket streams and publish to a kdb+ tickerplant. The architecture prioritizes correctness, latency observability, and reliable state management to support downstream VWAP and Order Book Imbalance calculations for 20 symbols.

6.1 Order Book State Machine

The quote handler implements a state machine for each symbol to correctly synchronize with Binance's depth stream protocol:

```
enum class BookState {
    INIT,        // Initial state, buffering deltas
    SYNCING,     // Snapshot applied, replaying buffered deltas
    VALID,       // Normal operation, applying live deltas
    INVALID      // Sequence gap detected, needs rebuild
};
```

Upon connection, each symbol enters INIT state and buffers incoming deltas while requesting a REST snapshot. The critical synchronization logic validates that the first delta after snapshot satisfies Binance's reconciliation rule:

```
if (state == BookState::SYNCING) {
    // First delta after snapshot
    // Must satisfy: U <= snapshotUpdateId+1 <= u
    if (firstUpdateId > snapshotUpdateIds_[idx] + 1) {
        invalidate(idx, "Snapshot too old");
        return false;
    }
    if (finalUpdateId < snapshotUpdateIds_[idx] + 1) {
        // Delta is stale, skip it
        return true;
    }
    // Transition to VALID
    states_[idx] = BookState::VALID;
}
```

This approach ensures the published L5 book is never stale or corrupted, which is essential for meaningful Order Book Imbalance calculations.

6.2 Cache-Efficient Flat Array Storage

The OrderBookManager stores all book data in contiguous arrays rather than per-symbol maps or trees:

```
// Price and qty arrays: [numSymbols * BOOK_DEPTH]
// Access: bidPrices_[symIdx * BOOK_DEPTH + level]
std::vector<double> bidPrices_;     // Bids sorted high->low
std::vector<double> bidQtys_;
std::vector<double> askPrices_;     // Asks sorted low->high
std::vector<double> askQtys_;
```

For 100 symbols at L5 depth, the entire dataset occupies approximately 16KB—fitting within CPU L1 cache. L5 extraction becomes a simple offset calculation, eliminating pointer chasing and cache thrashing as the symbol count scales.

6.3 Latency Instrumentation

Each published message includes timing fields captured at distinct points in the processing pipeline:

```
// Capture wall-clock receive time (for cross-process correlation)
auto recvWall = std::chrono::system_clock::now();
long long fhRecvTimeUtcNs =
    std::chrono::duration_cast<std::chrono::nanoseconds>(
        recvWall.time_since_epoch()).count();

// Start monotonic timer for parse latency
auto parseStart = std::chrono::steady_clock::now();
// ... JSON parsing and book update ...
auto parseEnd = std::chrono::steady_clock::now();
long long fhParseUs =
   std::chrono::duration_cast<std::chrono::microseconds>(
    parseEnd - parseStart).count();
```

This granularity allows precise identification of latency sources. When combined with tickerplant-side timestamps, end-to-end latency from exchange to kdb+ table is fully observable.

6.4 Reconnection with Exponential Backoff

Both handlers implement automatic reconnection with exponential backoff capped at 8 seconds. The backoff sleep is interruptible to allow rapid shutdown response:

```
bool TradeFeedHandler::sleepWithBackoff(int attempt) {
    int delay = INITIAL_BACKOFF_MS;
    for (int i = 0; i < attempt && delay < MAX_BACKOFF_MS; ++i) {
```

```
            delay *= BACKOFF_MULTIPLIER;
        }
        delay = std::min(delay, MAX_BACKOFF_MS);

        // Sleep in small increments to allow quick shutdown
        const int checkIntervalMs = 100;
        int slept = 0;
        while (slept < delay && running_) {
            std::this_thread::sleep_for(std::chrono::milliseconds(checkIntervalMs));
            slept += checkIntervalMs;
        }
        return running_;
}
```

This prevents rate-limit violations during exchange instability while maintaining availability.

6.5 Asynchronous IPC Publishing

Publication to kdb+ uses negative handle IPC, which is fire-and-forget:

```
K result = k(-tpHandle_, (S)".u.upd", ks((S)"trade_binance"), row,
    (K)0);
```

The negative handle ensures the handler writes to the socket buffer and continues without waiting for acknowledgment. This decouples feed handler throughput from tickerplant processing speed, preventing backpressure from stalling market data capture.

7 kdb+ Implementation

The kdb+ layer forms the analytical backbone of the system: a tickerplant sequences and persists events, an RDB stores raw market data, an RTE computes real-time analytics, and supporting processes handle telemetry and log management. Each component has a single responsibility, communicates via the standard `u.q` pub/sub protocol, and can recover independently from failures.

7.1 Tickerplant: Log-Then-Publish

The tickerplant uses the standard u.q pub/sub protocol but adds one critical behavior: every market data message is logged to disk *before* being published to subscribers. This ensures durability—if a subscriber crashes, it can replay from the log to recover state.

Listing 4: Log-then-publish in the update handler

```
.u.upd:{[tbl;data]
  / Add TP timestamp
  tpRecvTimeUtcNs:.tp.tsToNs[.z.p];
  data:data,tpRecvTimeUtcNs;
```

```
  / Log to disk FIRST (durability)
  .tp.logHandle enlist (`.u.upd; tbl; data);

  / Then insert and publish
  tbl insert data;
  .u.pub[tbl;data];
  };
```

The log format `enlist (`.u.upd; tbl; data)` is compatible with kdb+'s `-11!` streaming replay.

7.2 Crash Recovery via Log Replay

Both RDB and RTE implement automatic log replay on startup. The `-11!` operator streams through the log file, invoking each recorded `.u.upd` call:

Listing 5: Log replay with error handling

```
.rdb.replayFile:{[f]
  if[not .rdb.logExists[f]; :0j];
  / -11! returns count of chunks replayed
  replayed:.[{-11!x}; enlist f; {[e] -1 "Replay error: ",e; 0j}];
  replayed
  };
```

On startup, each process checks for today's log file and replays it before subscribing to the tickerplant for live updates. A process crash mid-day results in full state reconstruction without manual intervention.

7.3 Trade Buckets: Incremental VWAP Foundation

The RTE aggregates trades into time-bucketed summaries rather than storing individual ticks. Each 1-second bucket holds the running sums needed for VWAP:

Listing 6: Trade bucket structure and O(1) upsert

```
/ Keyed table: sym,bucket -> sumPxQty, sumQty, cnt
`sym`bucket xkey `tradeBuckets;

/ Hot path: update or insert in O(1)
.rte.bucket.add:{[s;time;price;qty]
  bucketTime:`timestamp$.rte.cfg.bucketNs * `long$time div .rte.cfg
      .bucketNs;
  k:(s;bucketTime);
  pxqty:price*qty;
  $[k in key tradeBuckets;
    tradeBuckets[k;`sumPxQty`sumQty`cnt]+:(pxqty; qty; 1j);
    `tradeBuckets upsert (s; bucketTime; pxqty; qty; 1j)
  ];
  };
```

VWAP for any window becomes a simple sum over buckets. This design amortizes storage cost—65 minutes of history at 1-second granularity requires only 3,900 rows per symbol rather than potentially millions of individual trades.

7.4 Variance-Covariance: Log Returns on Resampled Data

Computing a meaningful covariance matrix requires aligned time series. The RTE resamples 1-second trade buckets to 30-second intervals, forward-fills missing prices, and computes log returns:

Listing 7: Building aligned return vectors

```
/ Resample to 30-sec buckets (last price per window)
prices:select last price by sym, bucket:... from prices1s;

/ Build price matrix aligned by bucket, forward-fill gaps
priceMatrix:fills each getPrices[prices;buckets] each syms;

/ Log returns: log(p_t / p_{t-1})
returns:{r:log x % prev x; @[r;0;:;0n]} each priceMatrix;

/ Covariance matrix via nested iteration
matrix:syms!{[syms;R;i]
  syms!{[R;i;j] cov[R i;R j]}[R;i] each til count syms
  }[syms;R] each til count syms;
```

The `fills` function propagates the last known price forward when a symbol has no trades in a bucket—essential for maintaining alignment across assets with different trading frequencies.

7.5 Order Book Imbalance with EMA Smoothing

Raw OBI fluctuates rapidly. The RTE applies exponential smoothing to produce a stable signal:

Listing 8: OBI calculation with EMA

```
.rte.imb.update:{[s;bidDepth;askDepth;time]
  total:bidDepth + askDepth;
  imb:$[total > 0f; (bidDepth - askDepth) % total; 0n];

  / EMA: alpha * current + (1-alpha) * previous
  prevSmOBI:$[s in key .rte.imb.ema; .rte.imb.ema[s]; imb];
  smOBI:(.rte.cfg.obiAlpha * imb) + (1 - .rte.cfg.obiAlpha) *
     prevSmOBI;

  .rte.imb.ema[s]:smOBI;
  `.rte.imb.history insert (time; s; imb; smOBI);
  };
```

With an alpha of 0.05, each new reading contributes only 5% to the smoothed value—effectively a 20-sample exponential window that filters noise while remaining responsive to genuine shifts.

7.6 Telemetry: Percentile Aggregation

The TEL process computes latency percentiles every 5 seconds:

Listing 9: Percentile function and latency aggregation

```
.tel.percentile:{[p;x]
  if[0 = n:count x; :0n];
  idx:0 | ("j"$p * n - 1) & n - 1;
  `float$(asc x) idx
  };

/ Aggregate parse and send latencies by symbol
fhStats:select
  parseUs_p50:.tel.percentile[0.5; fhParseUs],
  parseUs_p95:.tel.percentile[0.95; fhParseUs],
  sendUs_p50:.tel.percentile[0.5; fhSendUs],
  sendUs_p95:.tel.percentile[0.95; fhSendUs],
  cnt:count i
  by sym from trades;
```

This produces the P50, P95, and max values displayed in the monitoring dashboards.

7.7 Summary

The kdb+ layer leverages q's strengths: vector operations for analytics, keyed tables for O(1) updates, and `-11!` for crash recovery. Each process remains focused—the RDB stores, the RTE computes, TEL monitors—communicating through the standard pub/sub protocol. The result is a system where any component can restart independently and recover its state from the tickerplant log.

8 Observed Performance

This section reports metrics observed during a representative session. All measurements were taken on the target hardware: WSL environment with 12 CPU threads and 7.6GB RAM.

8.1 Test Conditions

Parameter	Value
Duration	180 minutes
Symbols	3
Environment	WSL, 12 CPU threads, 7.6GB RAM
Market conditions	normal

Table 4: Test session parameters.

8.2 Data Volume

Symbol	Trades	Quotes	Total
BTCUSDT	240k	164k	404k
ETHUSDT	271k	160k	431k
SOLUSD	113k	104k	217k
Total	624k	428k	1052k

Table 5: Messages received per symbol.

Metric	Value
Trades per second (avg)	63
Trades per second (peak)	2374
Quotes per second (avg)	40
Quotes per second (peak)	110
Log file size	215 MB

Table 6: Throughput and storage.

Quote update rates observed: BTCUSDT 16/sec, ETHUSDT 15/sec, SOLUSDT 10/sec. High-liquidity pairs seem to receive updates faster than the nominal 100ms interval

8.3 Feed Handler Latency

Metric	P50	Max P95	Max P99
Trade FH parse (μs)	15	115	245
Trade FH send (μs)	1	25	50
Quote FH parse (μs)	109	754	998
Quote FH send (μs)	1	13	90

Table 7: Feed handler latency (microseconds).

8.4 Pipeline Latency

Segment	P50	Max P95	Max P99
Trades			
FH → TP (ms)	0.40	2.90	5.70
TP → RDB (ms)	0.14	0.52	0.82
End-to-end (ms)	0.54	3.10	6.90
Quotes			
FH → TP (ms)	0.40	1.22	3.19
TP → RDB (ms)	0.19	0.95	4.22
End-to-end (ms)	0.60	1.78	4.53

Table 8: Pipeline segment latency (milliseconds).

9 Conclusion

This project set out to build a real-time crypto analytics pipeline on modest hardware. It succeeded—but the more interesting outcome may be the approach rather than the system itself.

The technical architecture follows established patterns: C++ feed handlers for low-latency ingestion, a kdb+ tickerplant for sequencing and durability, downstream processes for storage and analytics. What distinguishes this implementation is the discipline applied throughout: single-responsibility processes, latency instrumentation at every hop, automatic crash recovery, and observability built into the foundation.

The system runs on a WSL environment with 12 CPU threads and 7.6GB of RAM—processing trades and L5 order book updates for 3 cryptocurrency pairs, computing VWAP, variance-covariance matrices, and order book imbalance in real-time, visualized through four dashboards. Thoughtful architecture accomplishes more than abundant resources poorly applied.

Equally significant is the documentation methodology. Traditional documentation fails modern projects: it goes stale, gets ignored, and cannot be queried programmatically. This project inverted the approach—documentation written first, structured for machine comprehension, maintained as the authoritative specification. The result is a codebase that can be understood conversationally. An AI assistant with access to the ADRs and specifications can answer design questions and onboard contributors faster than linear reading ever could.

Three lessons emerged:

First, observability is not optional. Every message carries timestamps from multiple pipeline points. When something degrades, the dashboards show exactly where.

Second, separation of concerns pays compound interest. Feed handlers do not compute analytics. The RTE does not store data. The tickerplant holds no state. This discipline complicates initial development but simplifies everything after.

Third, documentation written for AI is documentation that gets read. The effort invested in structured ADRs and machine-parseable specifications pays back every time someone queries the assistant instead of searching files.

The system is complete, functional, and extensible. The combination—a well-instrumented pipeline with AI-queryable documentation—represents a template for how complex technical projects should be built going forward.

References

1. Data Intellect. *Building Real-Time Event-Driven KDB-X Systems*. November 2025. dataintellect.com/blog/building-real-time-event-driven-kdb-x-systems-whitepaper
2. Official Documentation for the Binance APIs and Streams. https://developers.binance.com/docs/binance-spot-api-docs/README
3. Coulter, N. "Order Book: A kdb+ Intraday Storage and Access Methodology." KX Whitepapers. https://code.kx.com/q/wp/order-book/
4. Lucid, J. "C API for kdb+." KX Whitepapers. https://code.kx.com/q/wp/capi/
5. KX Systems. "Using Log Files: Logging, Recovery and Replication." KX Knowledge Base. https://code.kx.com/q/kb/logging/