Hypercube: core memory-mapped real time factors

research@strategynet.ai

August 2026

Abstract

A live basket monitor may recalculate thousands of values whenever prices change. The arithmetic is usually cheap; repeatedly finding, joining, serializing, and copying the same current inputs often dominates the cost. When every reader rebuilds that state for itself, a simple vector calculation inherits database latency, extra allocation, and inconsistent observation boundaries. Slice gives local processes a typed, memory-mapped view of current, entity-aligned values. Hypercube evaluates one coherent input generation through a declared node graph and produces cross-sectional outputs that can be published back as aligned slices. The public repository includes the format, readers and writers, calculation engine, deterministic synthetic sources, and live browser and terminal demonstrations. The current implementation also reuses compiled graph plans and publication buffers, distinguishes mapped visibility from durable flush, and supplies fixed-capacity online moments for explicit state owners. A reproducible benchmark suite tests those choices. A separate leakage-controlled experiment compares rolling-z event selection with calibrated point, robust, quantile, and triple-barrier tree models under stationary and regime-switching controls. This paper explains that implemented path and its limits. The current version publishes slices independently and provides no atomic multi-slice generation; vendor feeds, durable persistence, arbitrary function cells, replication, GPU lowering, and trading actions remain outside the current release. Although the examples use markets, the API also works for any stable set of entities.

Contents

Part I
Motivation

1 From events to live analytical state

Markets arrive as events, but many decisions are functions of state. A quote event changes the best bid and offer for one instrument; the next consumer usually wants the latest midpoint, spread, return, risk exposure, or cross-sectional rank. If every consumer independently rebuilds that state from an event stream, the system duplicates joins, symbol resolution, window maintenance, and normalization. If every calculation first queries a durable database, the database becomes part of the latency and availability path even when the working set is only a few megabytes of current vectors.

The distinction concerns time and purpose:

- an event log answers what happened and in what order;
- a durable analytical store answers historical and ad hoc questions;
- a live state plane answers what the current published values are;
- an execution graph answers how declared outputs follow from those values.

These are complementary responsibilities. Slice is the live state plane. The hypercube is the logical coordinate system and calculation graph. A database remains the recovery, research, and audit path; a transport remains the mechanism for moving events or published generations between machines.

> *A current analytical value should be cheap to read because its identity, layout, type, generation, and provenance were established when it was published—not rediscovered by every reader.*

The architecture is motivated by repeated whole-population work. Given entity-aligned weights w, returns r, signals s, and betas β, common calculations are

$$R_p = w^\top r, \qquad E_s = w^\top s, \qquad E_\beta = w^\top \beta, \qquad G = \|w\|_1.$$

The arithmetic is elementary. Most accidental complexity lies in obtaining vectors that refer to the same entities, units, observation cutoff, and generation. The system therefore records alignment and publication semantics as first-class metadata.

2 Design goals

The present architecture has six goals:

1. maintain latest source and derived state in fixed-layout, memory-mapped vectors;
2. permit many local processes to perform point reads and vector scans without taking a shared reader mutex;
3. reject linear algebra across incompatible entity layouts or value schemas;
4. run the same node graph in live, replay, and batch modes with a fixed calculation and mode-specific updates;
5. publish node identity, observation timestamps, coverage, and local compute diagnostics beside values;
6. leave room for typed, composable function cells without allowing calculation code to inherit trading authority.

The first implementation is intentionally narrow. It is little-endian, uses fixed-size pointer-free payloads, and assumes one writer with many readers per slice. Those constraints make the shared ABI explicit enough for Rust today and C++ or Python readers later. A richer type system, distributed replication, global multi-slice generations, and effectful actions require explicit extensions to the current file format.

3 Related work and scope

Memory mapping is the operating-system foundation. POSIX specifies a mapping between a process address space and a file or shared-memory object and defines shared mappings through which writes change the underlying object [1]. Slice adds an application ABI, identity, layout, and consistency protocol to that primitive. It does not assume that mapping a file alone makes concurrent access coherent.

Columnar memory systems establish the advantages of contiguous, typed buffers. Apache Arrow defines a language-independent columnar representation designed for sequential scans, constant-time random access, vectorization, and relocatable zero-copy shared-memory access [2]. Slice shares those physical motivations but serves a narrower purpose: mutable current-state vectors with stable entity slots and explicit writer epochs. Arrow remains the interoperability format and ecosystem.

Array database work treats multidimensional arrays as first-class data and avoids encoding every analytical object as a relation. SciDB, for example, was designed for complex analytics over array-shaped scientific data [3]. The hypercube developed here is closer to a resident execution and publication model than a general array DBMS. Dimensions describe aligned live slices and calculation identities; durable historical queries remain outside the hot path.

The numerical interface follows a long tradition of separating stable vector and matrix operations from their machine-specific implementations. The original BLAS specification included dot products, vector operations, norms, and related kernels [4]. Slice does not define a new numerical algebra. It makes aligned live vectors safe to hand to such an algebra and leaves room for optimized CPU, SIMD, or GPU kernels behind the same logical operation.

For change propagation, adaptive functional programming and differential dataflow show how explicit dependencies allow a system to update only affected results [8, 9]. Those systems are broader and more formal than the current hypercube engine. They nevertheless supply the right design model for future function cells: a declared dependency graph, logical time, deterministic propagation, and controlled state are preferable to hidden callbacks attached to mutable memory.

Part II
Memory and Cube Model

4 Stable entity layout

Let a layout be

$$L = (\lambda, a, n, U, \iota),$$

where λ is a layout identity, a is a normalized namespace (retained as `asset_class` in version-1 JSON), n is allocated capacity, U is the enabled entity set, and $\iota : U \to \{0, \ldots, n-1\}$ is an injective slot mapping. A symbol is a normalized lookup key; `instrument_id` is the legacy field name for the stable namespaced entity identity, and the slot is the dense numerical address.

$$\text{entity key} \longrightarrow \text{stable entity id} \longrightarrow \text{slot id} \longrightarrow S[\text{slot id}].$$

Slots do not move within a layout. New entities consume unused slots; removed entities are disabled or tombstoned. Reordering or compaction creates a new layout. The serialized layout is hashed, and every dependent slice carries that hash. A dot product is rejected unless its inputs agree on layout hash, schema, and active length.

This separation matters because a domain registry and a numerical layout have different rates of change. Venue, currency, multiplier, expiry, strike, and other rich metadata can live in keyed sidecars or services. The layout keeps only the identity and stable dense index required by the shared vectors.

5 Slice as a published vector

For a value field f and a published generation g, a primitive slice is a partial vector

$$S_{f,g} : U \to V_f \cup \{\bot\},$$

where V_f is a fixed physical type and $\bot$ denotes missing or invalid state. Dense analytical slices usually use $V_f = \mathbb{R}$; point-state slices use fixed records such as quote, trade, or quote-at-trade. Validity is explicit in every record, leaving zero with its ordinary numerical meaning.

The version-one file begins with a 256-byte header:

Header group	Fields	Purpose
format	magic, version, header length, value type, slot size	reject unknown physical representations
shape	capacity, active length, layout hash, schema hash	establish vector compatibility
publication	writer epoch, created and updated timestamps	detect stable reads and identify data age
operations	writer flags and heartbeat	distinguish liveness from value change

The current value types are dense $f64$ and fixed quote, trade, and TAQ records. All shared records are fixed-size, pointer-free, naturally aligned, and compatible with a C layout. A schema hash identifies the payload meaning, while a layout hash identifies the entity ordering. Neither substitutes for the other.

6 The logical hypercube

A collection of slices forms a logical cube even when no monolithic multidimensional allocation exists. For one generation,

$$X_g[u, f] = S_{f,g}[\,\iota(u)\,].$$

The implemented engine names four axes directly: generation, entity, primitive field, and calculated node. Applications may add asset class, venue, factor, family, horizon, universe, scenario, portfolio, or model version:

$$X[g, u, f, h, v, s, \ldots].$$

Only a sparse subset of this space is materialized. A slice is dense along the entity axis with the other coordinates fixed. A catalog maps a semantic name such as `signal` to the file, layout, type, and role that materialize one such slice.

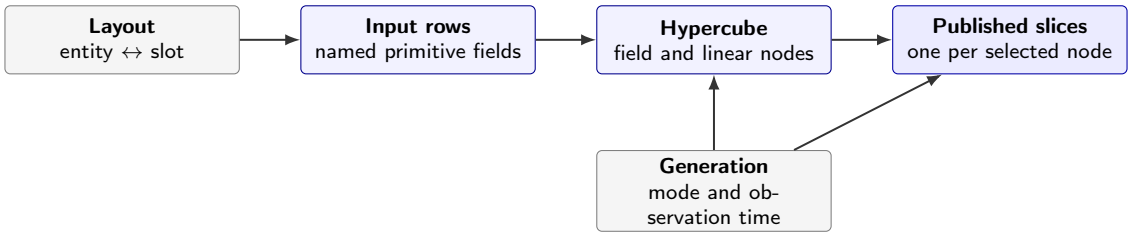

Figure 1: Physical slices are aligned one-dimensional slices. Layout and generation metadata give the collection its hypercube meaning.

This formulation avoids two extremes. It does not flatten all meaning into an untyped matrix, and it does not allocate every possible combination of dimensions. The catalog and calculation registry describe a sparse set of materialized slices; consumers request only the axes and generations required for their operation.

7 Time, observation, and publication

Financial decisions require more temporal detail than "latest." A cell may carry:

$$\tau(c) = (t_{\text{event}}, t_{\text{ingest}}, t_{\text{compute}}, t_{\text{publish}}, g).$$

Event time states when the source says the observation occurred. Ingest time states when the local system received it. Compute and publish time bound the derived value's latency. Generation g states which coherent calculation the value belongs to.

The current quote and trade records retain source and ingest timestamps. TAQ records preserve the quote known locally when a trade was processed, together with quote age and flags for missing, stale, or temporally inverted context. Hypercube `CellValue` records node, entity key, finite value, and the oldest contributing observation time. `NodeStatus` records emitted and missing counts plus local compute time. Snapshot time, slice updated time, and writer heartbeat remain distinct. An application may build a freshness service level with a testable predicate:

$$\mathrm{fresh}(c, t_d) \iff \mathrm{status}(c) = \mathrm{ok} \wedge 0 \le t_d - t_{\mathrm{publish}}(c) \le \Delta_{\max} \wedge t_{\mathrm{compute}} \le C_{\max}.$$

Part III
Consistency and Publication

8 What "nonlocking read" means

The desired property is precise: a reader of an already mapped slice should not acquire a process-shared mutex or wait for a writer-owned critical section. This *reader-lockless* design permits retries and therefore offers no formal wait-free guarantee. An optimistic reader may have to retry, and under a continuous writer it may fail to obtain a stable version within its retry budget.

Sequence counters are a standard form of this tradeoff. The Linux documentation describes lockless readers that accept data only when an even sequence value is unchanged across the read; it also notes that readers can spin while a writer is interrupted and that raw counters do not serialize multiple writers [5]. The current format consequently assumes one writer per slice and bounds reader retries.

9 Point-state protocol

Quote, trade, and TAQ slices use one sequence counter per slot. For slot i, the writer performs

$$\begin{aligned} q_i &\leftarrow 2k+1 && \text{publication in progress,} \\ S[i] &\leftarrow v && \text{copy fixed payload,} \\ q_i &\leftarrow 2k+2 && \text{publication complete.} \end{aligned}$$

The reader loads q_0, rejects an odd value, copies the payload, and loads q_1. It accepts only if

$$q_0 = q_1 \quad \wedge \quad q_1 \bmod 2 = 0.$$

An update to AAPL therefore does not make a reader of MSFT retry. This is the appropriate granularity for independently changing latest-state records.

The point snapshot of an entire struct slice guarantees coherence for each cell independently. Consumers that need every slot at the same market instant require a generation-level snapshot protocol.

10 Vector protocol

Dense $f64$ slices use one epoch for the vector. A writer makes the epoch odd, updates one or many cells, and publishes the next even epoch. A reader loads e_0, scans or copies the vector, and loads e_1; it accepts when

$$e_0 = e_1 \quad \wedge \quad e_1 \bmod 2 = 0.$$

Dot products validate both input epochs before and after the operation. Layout, schema, and active-length checks precede the arithmetic.

This protocol gives a coherent view of one slice without a reader mutex. Its scope is limited:

- a busy writer may force a reader to retry;
- separate slices have separate epochs, so sequentially stable copies may still belong to different logical input generations;
- one writer must own publication for a slice;
- memory-order behavior must be validated on every supported architecture and foreign-language reader implementation.

11 Consistency classes

Different data uses should pay for different consistency:

Class	Example	Read guarantee	Failure behavior
cell-latest	quote display, health probe	one slot, per-cell sequence	retry or report unavailable
vector-stable	exposure, rank, dot product	one slice, stable even epoch	bounded retry
generation-coherent	ETF value or another multi-field decision	named immutable multi-slice generation	reject mixed generation
decision-grade	order or rebalance input	coherent generation plus freshness, authority, and risk checks	fail closed; no action

The table prevents a common design error: imposing a global lock on all state because a small subset of decisions require a global snapshot. Latest quote display can tolerate independently published slots. A cross-sectional rank needs one stable vector. An ETF fair-value calculation may need constituent prices, currencies, holdings, and shares outstanding from one declared generation. A trade requires that coherence plus policy and authority.

12 Published immutable generations

The next publication mechanism is a small multi-version protocol. A writer builds generation $g+1$ in inactive buffers while readers continue to use g. After every slice passes shape, schema, and completeness checks, the writer publishes one compact manifest:

$$M_{g+1} = (g+1, t, \lambda, \{(f, e_f, h_f)\}, H),$$

where e_f and h_f are the final epoch and content identity of each slice and H authenticates the manifest. Publication changes a single active-generation reference. A reader pins M_g, reads only the referenced immutable buffers, and verifies that the active generation did not change before accepting the result.

This is a read-copy-publish design related to multiversion and RCU techniques. RCU separates removal from reclamation so readers can traverse an existing version without acquiring a conventional lock [6]; snapshot isolation likewise gives a transaction a named database version [7]. The slice implementation still needs its own lifecycle rules:

- a generation is immutable after publication;
- buffer reuse waits until no reader can reference the old generation;
- a slow reader is bounded or copied out so it cannot retain generations indefinitely;
- publication uses atomic rename or an aligned atomic generation word;
- crash recovery selects the last complete manifest, never half-written vectors.

No immutable multi-slice manifest or reader-pinning protocol is implemented in version 1. Consumers that need a coherent decision across several nodes should use the in-process `Snapshot`; general multi-slice publication remains an explicit extension and must not be inferred from independent slice epochs.

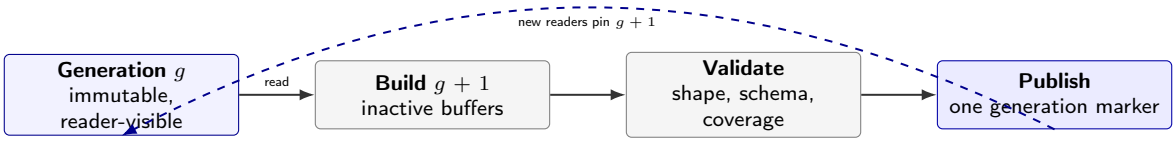

Figure 2: Copy-publish generations keep readers on immutable state while the next coherent set of slices is assembled.

13 Sharding and progress

Double buffering every vector is attractive when the working set is small. For larger or faster-changing cubes, the publication unit can be a shard: entity range, field group, asset class, or semantic family. A generation manifest then references a vector of immutable shard versions. Copy-on-write touches only changed shards while the manifest preserves a coherent global view.

Progress measurements should include writer duration, epoch changes per second, reader retries, snapshot failures, generation publish latency, oldest pinned generation, and reclaimed bytes. If retry tails grow under ingestion load, the system can use smaller shards, longer immutable generations, a copied snapshot, or a stale-but-declared generation according to the consumer's consistency class.

Part IV
Hypercube Execution

14 Runtime inputs and outputs

The current Rust engine receives

$$U_g = (g, t, m, R, N),$$

where g is a strictly increasing generation, t is its Unix-millisecond observation time, m is the execution mode, R contains keyed `InputRow` values and named primitive fields, and N is the complete `NodeSpec` graph. It emits

$$H_g = (g, t, m, n, V, Q),$$

where n is the entity count, V contains ordered `CellValue` records, and Q contains ordered `NodeStatus` diagnostics.

The modes are `Live`, `Replay`, and `Batch`. All three use the same calculation implementation, with the mode recorded as context. Replay supplies its own clock and rows while preserving node semantics. The engine rejects a generation that does not advance beyond the last successful update.

15 Calculation nodes

The implemented engine has two node kinds and five cross-sectional transforms:

Node	Input and operation	Output
`Field`	one named primitive field from every entity row	aligned raw values before the selected transform
`Linear`	required or optional upstream nodes and declared weights	weighted value per entity, optionally normalized by available absolute weight

`Identity`, `ZScore`, `Rank`, `Percentile`, and `RankZScore`

The engine derives execution order topologically and returns values in declaration order. Duplicate rows or nodes, empty identifiers, duplicate inputs, non-finite weights, cycles, and unknown required dependencies fail the update. A missing required value suppresses that entity's linear output; an optional dependency may be absent.

16 Cross-sectional algebra

For node j and entity i, let x_{ij} be a raw current value. The engine can publish raw values, ordinary z-scores, ranks, percentiles, or rank z-scores. A simple linear composite is

$$c_i = \mathcal{N}\left(\frac{\sum_{j \in A_i} w_j x_{ij}}{\sum_{j \in A_i} |w_j|}\right),$$

where A_i is the set of available dependencies for entity i and $\mathcal{N}$ is the declared output transform. The denominator is used only when `normalize_weights` is true; otherwise the raw weighted sum is transformed. A zero available denominator emits no value.

17 Resident state and incremental work

The engine retains the last successfully completed generation as a monotonicity guard and one compiled plan for the most recent node declaration. Every accepted update still supplies complete rows and node specifications and recalculates every value. When the declaration is unchanged, validation has already produced indexed dependency edges and a topological order, so later generations do not rediscover the same schedule. The public `graph_compilations()` counter makes topology churn observable.

This reuses the compiled configuration; every generation still recalculates its values. The pure graph has no built-in price histories, session accumulators, alias registries, or implicit window state. The separate `RollingMoments` primitive gives a simulator or explicitly stateful callback a preallocated circular window with online mean and variance. Its checkpoint and replay schema remain the responsibility of that state owner.

The natural extension is change-directed evaluation:

$$\Delta X_g \longrightarrow \{\phi : \mathrm{deps}(\phi) \cap \Delta X_g \neq \varnothing\} \longrightarrow \Delta Y_g.$$

In a future incremental engine, only nodes whose dependency cones intersect changed cells would need to run. Logical generation and event time must remain part of the cache key; otherwise a fast cache can quietly return a value computed from the wrong observation cutoff.

18 Hot state, persistence, and transport

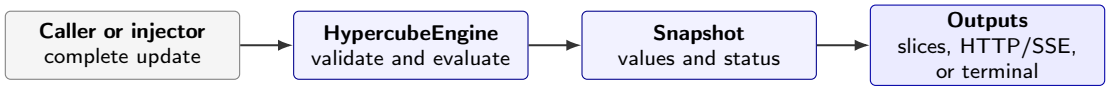

Figure 3: The implemented public path. Vendor feeds, durable persistence, replication, and production gateways are outside the repository.

`SlicePublisher` writes one configured $f64$ slice per selected node and publishes `layout.json` and `catalog.json`. Projection reuses one vector per node, caches entity slots, and scans snapshot cells once. Mapped-memory visibility, asynchronous flush, and synchronous durability are explicit policies. The compatibility `publish()` method remains durable; the synthetic live view deliberately selects visibility. Those slices are independently coherent under every policy; the in-process `Snapshot` is the coherent whole-generation object. The synthetic server also serves a dependency-free browser dashboard, a JSON snapshot, and an SSE stream. The `etf` and `pairs` examples run separate

top-like terminal monitors over basket premiums and cointegrating residuals. These paths make the architecture inspectable without external services.

The version-1 file ABI and Rust readers form a public local API. The example server supplies no authentication, entitlement, backpressure, or production reconnect guarantees. Vendor feeds, durable stores, host-to-host transport, and a governed gateway require separate adapters and policies; none is hidden in the public engine.

Part V
Composable Cells

19 From values to functions

The current cube materializes primitive values and runs a fixed set of node types. The general cell model is

$$c = (\tau, \sigma, \pi, \nu),$$

where τ is a value type, σ is shape and dimension metadata, π is provenance and publication state, and ν is either a materialized value or a typed expression.

For a function cell,

$$c_j := \phi_j(c_{i_1}, \ldots, c_{i_k}; \theta_j),$$

the registry stores the function identity and version, typed inputs, parameters θ_j, output shape, missing-data policy, and permitted state. The dependency graph is explicit. Acyclic pure expressions can be topologically evaluated; windowed or recursive expressions require declared state and logical-time semantics.

Composability comes from retaining the same cell schema at each stage. A quote midpoint can feed a return; returns can feed factors; factors can feed a portfolio or index; the resulting discrepancy can feed a decision rule. Each output remains addressable, versioned, and independently inspectable.

20 Function classes

Not every function should have the same authority:

Class	Examples	Guarantee
pure scalar/vector	midpoint, return, dot product, normalization	deterministic for a generation; no hidden I/O
stateful analytic	VWAP, rolling covariance, session range	declared state, clock, retention, reset, and replay semantics
model inference	learned score, volatility estimate	versioned artifact, features, calibration, and resource limits
decision	threshold, rebalance proposal, hedge intent	produces an intent and explanation; no execution authority
effect adapter	send order, alert, persist, publish	explicit capability, idempotency, audit, timeout, and failure policy

The distinction between a decision and an effect is structural. A function cell may calculate that a discrepancy exceeds a threshold. It should not silently turn a read of that cell into an order. Effect adapters consume published decision intents only after checking generation, freshness, entitlement, position limits, market state, and a unique action key.

21 ETF and index example

Consider an ETF e with constituent quantities q_{ei}, local prices $p_i(t)$, currency conversion rates $x_i(t)$, cash and accrued value $C_e(t)$, liabilities $L_e(t)$, and shares outstanding $N_e(t)$. A simplified computed value per share is

$$\widehat{V}_e(t) = \frac{C_e(t) - L_e(t) + \sum_i q_{ei} p_i(t) x_i(t)}{N_e(t)}.$$

An index uses analogous constituent arithmetic with its declared weights, corporate-action rules, and divisor. The observed ETF midpoint $m_e(t)$ gives a discrepancy

$$\delta_e(t) = m_e(t) - \widehat{V}_e(t), \qquad \delta_e^{\mathrm{bps}}(t) = 10^4 \, \frac{m_e(t) - \widehat{V}_e(t)}{\widehat{V}_e(t)}.$$

In the cube, each input is an aligned slice or a declared keyed join:

$$\begin{aligned} P_g &= \text{constituent price vector}, \\ X_g &= \text{currency conversion vector}, \\ Q_{e,g} &= \text{holdings vector}, \\ \widehat{V}_{e,g} &= \phi_{\mathrm{nav}}(P_g, X_g, Q_{e,g}, C_{e,g}, L_{e,g}, N_{e,g}), \\ \delta_{e,g} &= \phi_{\mathrm{delta}}(m_{e,g}, \widehat{V}_{e,g}). \end{aligned}$$

The calculation is accepted only when price, FX, holdings, cash, liabilities, and share count satisfy their observation and generation policies. Holdings from last night's file may be valid with an explicit as-of time; a stale FX rate may not be. The cell graph makes those choices visible.

21.1 Runnable ETF monitor

The checked-in `etf` example implements a compact version of the basket calculation. It creates stable synthetic constituents and long-only baskets whose weights sum to one. For ETF j, it forms a basket return and advances fair value along the constituent axis:

$$r_{j,g}^{\mathrm{NAV}} = \sum_{i=1}^{N} w_{ji} r_{i,g}, \qquad V_{j,g} = V_{j,g-1}(1 + r_{j,g}^{\mathrm{NAV}}).$$

The simulated ETF price is $P_{j,g} = V_{j,g}(1 + u_{j,g})$, where the fractional premium follows

$$u_{j,g} = \phi u_{j,g-1} + \sigma \eta_{j,g}, \qquad \phi = 0.88, \quad \sigma = 0.00040.$$

The terminal reports premium in basis points and against the stationary model standard deviation:

$$q_{j,g} = 10^4 \frac{P_{j,g} - V_{j,g}}{V_{j,g}}, \qquad z_{j,g} = \frac{u_{j,g}}{\sigma / \sqrt{1 - \phi^2}}.$$

Hypercube publishes ETF-aligned fair values, prices, premiums, model z-scores, and cross-sectional premium ranks. This keeps the two axes explicit: basket arithmetic reduces a constituent vector to one ETF value; a Hypercube node produces another aligned ETF vector.

21.2 Runnable pairs monitor

The `pairs` example gives each pair a stable identity and two price paths. Leg B carries a stochastic trend, while leg A shares it through hedge ratio β_j. Their cointegrating residual is

$$y_{j,g} = \log A_{j,g} - \alpha_j - \beta_j \log B_{j,g}, \qquad y_{j,g} = \phi_j y_{j,g-1} + \sigma_j \eta_{j,g}.$$

The monitor standardizes the current residual against the preceding $W = 20$ residuals. If $\mathcal{W}_{j,g} = \{y_{j,g-W}, \ldots, y_{j,g-1}\}$, then

$$\bar{y}_{j,g} = \frac{1}{W} \sum_{u \in \mathcal{W}_{j,g}} u, \qquad z_{j,g} = \frac{y_{j,g} - \bar{y}_{j,g}}{\sqrt{W^{-1} \sum_{u \in \mathcal{W}_{j,g}} (u - \bar{y}_{j,g})^2}}, \qquad h_j = \frac{\log(1/2)}{\log \phi_j}.$$

Scoring occurs before $y_{j,g}$ enters the circular window, so an observation does not estimate its own center and scale. Until the prior window has nonzero variance the display emits a neutral zero. Hypercube publishes current prices, this rolling residual score, model half-life, and a cross-sectional rank of absolute residual size. The table shows the largest live dislocations and the sign-implied hedged position. Fixed α_j, β_j, and ϕ_j keep the example readable; out-of-sample estimation, stationarity tests, structural breaks, costs, borrow, and exposure controls remain outside it.

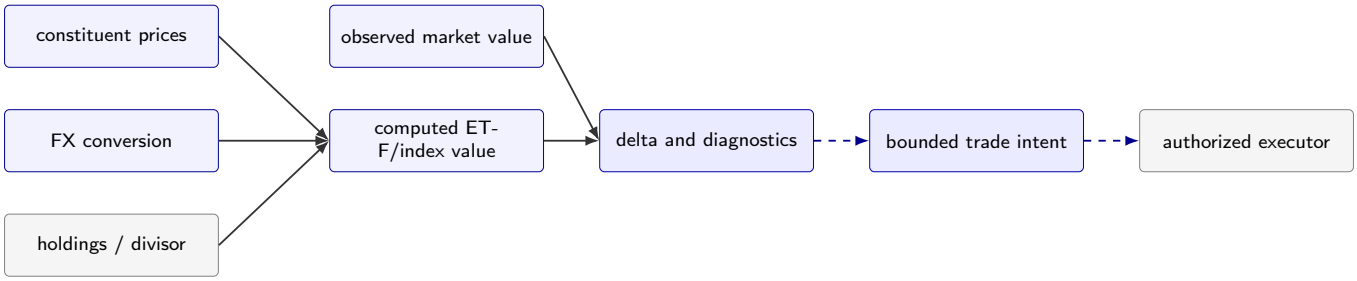

Figure 4: The calculation is already vector-shaped. Solid edges show current slice behavior; dashed edges show proposed decision and effect stages.

The same pattern supports baskets, hedges, factor-mimicking portfolios, risk decompositions, and observed-versus-model residuals. The significant abstraction is the ability to compose versioned values and functions while preserving a coherent generation and keeping effects explicit and separate.

22 Noise-aware statistical-arbitrage selection study

The runnable pairs monitor forms its signal by comparing the current residual with the preceding 20 observations. Our research harness begins with that same signal and the same mean-reversion trade direction, then gives a fitted model one job: rank the candidate trades and decide which ones are worth taking. By defining the economic trade before the model sees it, we can compare every model with the rolling z-score policy while holding the strategy constant. Any improvement then reflects candidate selection.

For candidate time t, let

$$z_t = \frac{y_t - \mu(\mathcal{W}_t)}{\sigma(\mathcal{W}_t)}, \qquad s_t = -\operatorname{sign}(z_t), \qquad |z_t| \geq 1,$$

where $\mathcal{W}_t = \{y_{t-20}, \ldots, y_{t-1}\}$. At future offset h, normalized gross return is

$$g_t(h) = s_t \frac{y_{t+h} - y_t}{\sigma(\mathcal{W}_t)}.$$

The triple-barrier target uses profit barrier $b_+ = 0.75$, stop barrier $b_- = 1.00$, vertical horizon $H = 12$, and normalized round-trip cost $c = 0.08$. Define

$$\tau_+ = \inf\{h \in [1, H] : g_t(h) \geq b_+\}, \qquad \tau_- = \inf\{h \in [1, H] : g_t(h) \leq -b_-\}.$$

When a barrier remains untouched, its first-touch time falls beyond the trading horizon. The profit barrier therefore produces class $+1$, the stop barrier produces class -1, and a trade that reaches the end of the horizon produces class zero. Using the same exit, we also retain the realized return as a continuous target:

$$r_t = g_t(\min(\tau_+, \tau_-, H)) - c.$$

The three labels make triple barrier a natural classification problem, although the label records only which exit occurred. The continuous target preserves the size of any overshoot and the variable return earned when time expires, giving the regression models economic information that the classifier cannot recover from the class label alone [17].

The study moves forward through five consecutive periods so that every modeling choice is made before the final results are observed:

$$\text{fit} \rightarrow \text{validation} \rightarrow \text{predictive calibration} \rightarrow \text{policy selection} \rightarrow \text{untouched test}.$$

The fitting period estimates the trees, while the following validation period determines how many boosting iterations to use. A third period then corrects systematic bias in point forecasts, quantile coverage, or class probabilities. Once those corrections have been fixed, the policy-selection period chooses the lowest score at which the strategy will accept a trade, and the final test period measures that completed policy.

We leave 12 observations between each period, matching the longest possible trade. This gives every candidate opened in one period enough time to finish before data from the next period are used, which keeps its eventual outcome from leaking into a later modeling decision. The simulation also waits for an open position in a pair to close before considering another trade in that pair.

We compare LightGBM and CatBoost using ordinary squared-error, Huber, quantile, and multiclass objectives [18, 19]. Ordinary regression aims at the conditional mean return, whereas Huber loss gradually changes from a quadratic penalty near zero to a linear penalty in the tails, reducing the influence of an extreme outcome [20]. Quantile regression describes several points in the conditional return distribution and retains information beyond its mean [21]. For $q \in \{0.2, 0.5, 0.8\}$, we measure the coverage error during the calibration period and apply the following fixed correction to later forecasts:

$$\Delta_q = \text{Quantile}_q(r - \widehat{Q}_q(x)), \qquad \widetilde{Q}_q(x) = \widehat{Q}_q(x) + \Delta_q.$$

After applying this correction, we sort the three forecasts for each event so that the 20th percentile cannot exceed the median or the 80th percentile. We then rank events with a score that discounts the median whenever the lower tail is wide:

$$u_Q = \widetilde{Q}_{0.5} - 0.25(\widetilde{Q}_{0.5} - \widetilde{Q}_{0.2}).$$

Requiring the corrected 20th percentile to exceed zero would reject almost every event in the reference test because stop outcomes account for more than one fifth of the distribution. The score above still responds to lower-tail risk and lets the strategy compare candidates despite the high rate of stop outcomes.

Class weighting helps the three-way models learn the less frequent outcomes, but it also means that their raw scores should not be read directly as probabilities. We therefore fit an independent one-versus-rest sigmoid calibrator [22], then convert the calibrated probabilities into expected net return:

$$u_C(x) = \sum_{k \in \{-1,0,+1\}} P^{\text{cal}}(Y = k \mid x)\, \bar{r}_k,$$

Here $\bar{r}_k$ is the average realized return for class k, measured in the same calibration period. On the reference seed, the 20th, 50th, and 80th percentile forecasts cover approximately 0.18–0.20, 0.47–0.49, and 0.79–0.80 of the later test outcomes, while the expected calibration error for the profit class is about 0.03. Those diagnostics show that the forecasts have sensible probabilistic meaning; trading performance still depends on whether they rank the economically attractive events ahead of the rest.

We test the models in two synthetic settings. The first is a homogeneous Gaussian Ornstein–Uhlenbeck process, where the rolling score already captures the useful structure and a more flexible model has little reason to improve on it. The second moves among observable volatility and persistence conditions, with stressed periods adding unit-variance Student-$t(4)$ shocks. The models observe a noisy volatility measure but never receive the underlying regime itself. Every reported result combines two expanding-window evaluations over 20 pairs and 3,300 observations, and we repeat the complete experiment with five deterministically derived seeds.

Scenario and policy	Mean trade t	Delta	Seed wins	Total-net delta
stationary OU, rolling z	12.666	—	—	—
stationary OU, closest ML (LightGBM quantile)	12.648	-0.018	2/5	-91.283
regime stress, rolling z	6.626	—	—	—
regime stress, CatBoost Huber	**7.290**	**+0.664**	**5/5**	**+48.176**
regime stress, LightGBM Huber	7.181	+0.555	5/5	+32.926

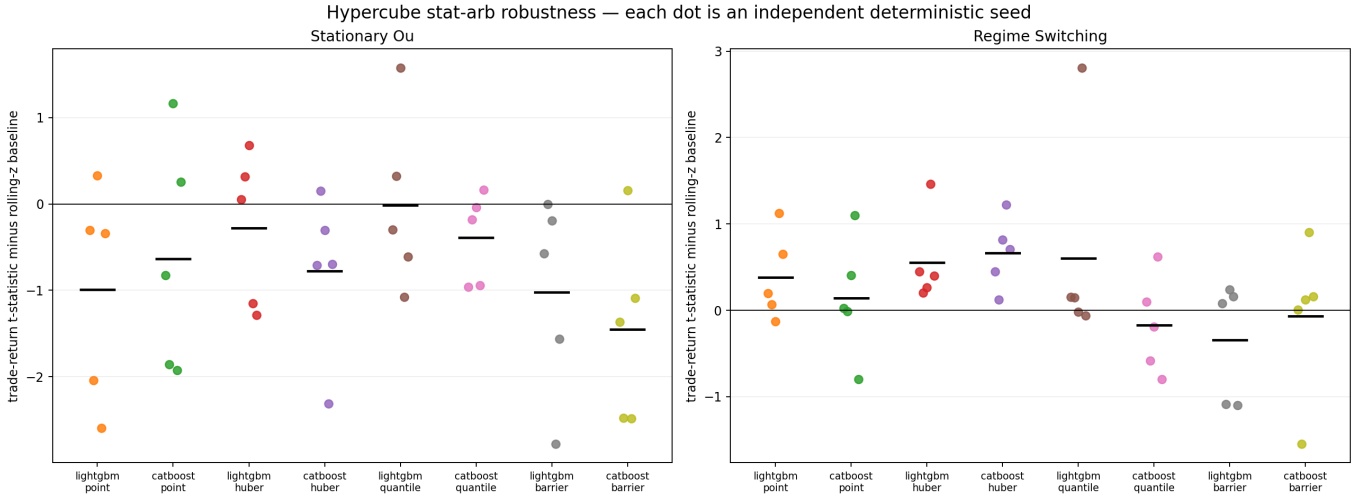

Figure 5: Difference from the rolling-z trade-return t-statistic for five complete deterministic seeds; black segments mark means. The statistic is descriptive over normalized net trade returns. Annualized Sharpe and a dependence-corrected significance test require separate estimates.

In the stationary setting, the simple rolling policy remains marginally better on average. When volatility, persistence, and tail behavior change over time, both Huber models improve on it in all five runs, with CatBoost producing the largest average gain. This result shows that the harness can detect useful conditional structure in a setting where we deliberately put such structure into the data; a claim about market profitability would require the same frozen protocol to survive a study of real, survivorship-controlled prices. Such a study would estimate hedge parameters inside each historical fold and include bid/ask execution, borrow, asynchronous-leg risk, portfolio constraints, and clustered or block-bootstrap uncertainty. For now, the fitted models live in the optional Python research workflow, while the Rust runtime continues to run the rolling statistical monitor.

23 From analysis to controlled execution

A future trading adapter should accept an immutable intent:

$$I = (g, \text{strategy}, \text{instrument}, q, \text{limit}, \text{expiry}, \text{evidence}, k),$$

where k is an idempotency key. Before submission it must verify:

1. generation g is decision-grade and within its freshness budget;
2. the calculation, model, and policy versions are approved;
3. the account has authority for the instrument and action;
4. position, notional, liquidity, price, and loss limits pass;
5. the market is in an allowed session and no kill switch is active;
6. action key k has not already been consumed.

The executor records acknowledgement, rejection, fill, and cancellation events back into the evidence path. These events may become new cube inputs, but the dependency must not create an unbounded accidental feedback loop. Stateful control policies need explicit cycle semantics, rate limits, and termination conditions.

Part VI
Implementation and Evidence

24 Current implementation

The repository contains two principal Rust crates:

Component	Implemented responsibilities	Present limits
`hypercube-slice`	layout and catalog validation; versioned headers; $f64$, quote, trade, and TAQ mappings; point and vector readers/writers; stable snapshots, sums, dot products, and bounded top-absolute selection	little-endian; one writer per slice; local file-backed mappings
`hypercube-engine`	generation-checked updates; field and linear nodes; five cross-sectional transforms; reusable graph plans; deterministic snapshots and status; fixed-capacity rolling moments; buffered output `SlicePublisher` with explicit durability; generic and OU market injectors; HTTP/SSE plus `etf` and `pairs` terminal demonstrations	complete in-process rows per update; no vendor adapter, persistence, general function cells, or production web service

The slice crate supplies fixed quote, trade, and TAQ record schemas; callers handle live-feed ingestion and retain responsibility for record meaning and provenance. Callers provide the engine with ordinary in-process rows; direct input-slice reads are outside the current adapter. Its publisher projects selected calculated nodes into $f64$ slices sharing one entity layout; it does not publish factor-family rollups, portfolio projections, or application-specific summaries.

25 Scale model and reproducible baseline

For F published nodes and N entities, dense $f64$ payloads use approximately

$$8FN \text{ bytes}$$

plus one 256-byte header per slice. The checked-in semi-live smoke run uses eight entities and five nodes. It emits 40 cell values and five 320-byte slices:

$$256 + 8 \cdot 8 = 320 \text{ bytes per slice.}$$

The dashboard, JSON snapshot, and SSE stream expose that same state. This is a functional and packaging baseline. The following benchmark is reported separately so its host and postconditions remain explicit.

26 Design-pattern audit and measured performance

The implementation was cross-checked against the Imperial HFT design-pattern and pairs-trading corpus at commit `5783606` [10]. Its most useful result came from reusing the pairs window, which alone reduced its checked-in benchmark median from 334.7 to 172.9 microseconds; the combined fixed buffer, one-pass moments, and AVX variant reached 66.3 microseconds. Explicit SIMD alone reached 302.9 microseconds, while software prefetching made no material difference in its separate scan. This motivates algorithm and allocation work before architecture-specific instructions.

Hypercube therefore adopted four bounded changes:

1. compile stable dependency declarations once and reuse indexed plans;
2. cache entity slots and reuse one publication vector per node;
3. maintain rolling moments with a fixed circular buffer;
4. select a small top set in linear time and sort only that prefix.

Forced inlining, manual unrolling, prefetching, and explicit SIMD remain benchmark-gated. Rust's portable SIMD API is still experimental, and target features require an explicit deployment configuration [15, 16].

For the rolling window, state is (n, μ, M_2), where $M_2 = \sum_i (x_i - \mu)^2$. Adding x uses the centered Welford update [12]:

$$n' = n + 1, \qquad \delta = x - \mu, \qquad \mu' = \mu + \frac{\delta}{n'}, \qquad M_2' = M_2 + \delta(x - \mu').$$

When a full window evicts oldest value x_o, the removal identity is

$$n^- = n - 1, \qquad \mu^- = \frac{n\mu - x_o}{n^-}, \qquad M_2^- = M_2 - (x_o - \mu)(x_o - \mu^-),$$

then the addition update is applied. These centered formulas avoid the cancellation-prone difference of large raw sums; a bounded periodic corrected two-pass rebuild limits accumulated roundoff [13]. The implementation stores the mean as an offset from a retained window origin, so a large common level does not consume the precision needed for a small residual variance.

The committed Criterion 0.7 harness [11] used 30 samples after a one-second warmup and a three-second measurement target. Before and after lanes used identical benchmark source and tool version. The host was an AMD Ryzen Threadripper PRO 7985WX running Linux 6.8, Rust 1.97.1 and LLVM 22.1.6, with the standard bench profile and no native-CPU or explicit SIMD flags.

Benchmark	Before/reference	Current	Result
stable five-node graph, 128 entities	106.18 μs	96.99 μs	−8.55%
stable five-node graph, 1,024 entities	998.31 μs	888.30 μs	−11.59%
durable five-slice publish, 1,024 entities	9.071 ms	3.494 ms	−62.08%
rolling z-score, 4,064 observations	156.43 μs	44.53 μs	3.51×
top 10 of 16,384 finite values	296.92 μs	47.92 μs	6.20×

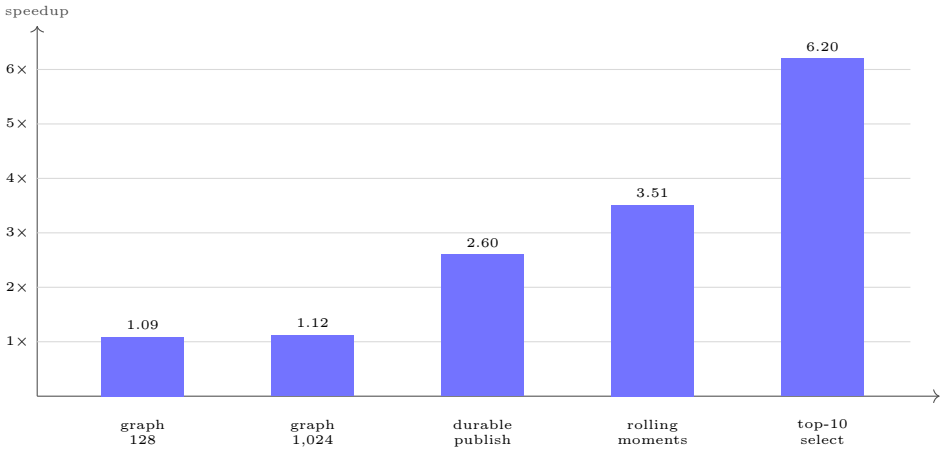

Figure 6: Central-estimate speedup against the matched pre-change or reference lane. The first three are before/after Hypercube results; the final two compare algorithms in the same current harness.

At 128 entities the durable publisher remained statistically unchanged because the synchronous operating-system flush dominates. The explicit mapped-memory policy measured 12.76 microseconds at 128 entities and 109.04 microseconds at 1,024, but it has a weaker postcondition: memory visibility without storage durability [14]. The host ran without isolation, frequency locking, or core pinning. These development-host microbenchmarks provide no evidence about tail latency, contention, NUMA, crash recovery, or production SLAs. Full commands, confidence limits, and the pattern-by-pattern audit are checked in as `docs/performance.md`.

27 Conformance tests

The current workspace has 37 Rust tests:

1. seven engine, publisher, and synthetic-injector tests cover dependency cycles, plan reuse, stale generations, tie-aware ranking, deterministic inputs, and aligned publication;
2. six rolling-moment tests cover eviction, no-look-ahead scoring, non-finite input, large-offset and sub-epsilon variance retention, and agreement with corrected two-pass windows;
3. two financial-example tests cover long-only ETF weights and exact reconstruction of the declared pair residual;
4. eight slice tests cover layouts and catalogs, format compatibility, vector algebra, bounded top selection, heartbeat semantics, and fixed records;
5. fourteen circuit tests cover ring backpressure, trigger state, recording, digests, and fresh-state replay.

The complete all-target suite, including the benchmark smoke harness, also passes under the declared minimum compiler, Rust 1.81.0.

Foreign-language byte fixtures, high-contention stress, interrupted writers, cross-process x86-64 and aarch64 evidence, immutable generation pinning, and crash recovery remain conformance work.

Performance reports should state entity count, slice count, update distribution, writer rate, reader count, operation, page residency, CPU architecture, flush policy, and percentile methodology. Useful measurements include point-read latency, stable-vector latency, retries per accepted read, generation publish time, changed-shard bytes, and end-to-end input-to-node-to-publication latency.

28 Limits

The current implementation covers the following vertical slice and limits:

- published output vectors are stable per slice; global atomicity across the complete output set is absent;
- vector readers use optimistic retry and can fail to progress within a fixed budget under sustained writes;
- shared atomic and payload behavior requires explicit cross-process, cross-language, x86-64, and aarch64 conformance evidence;
- layouts are stable while running; dynamic mutation requires a new layout and coordinated publication;
- replication and general live gateway generation protocols remain separate work;
- vendor ingestion, durable persistence, and direct input-slice adapters are outside the public repository;
- the current typed calculation runtime excludes arbitrary function cells, automatic incremental invalidation, model sandboxing, and GPU lowerings;
- action cells and trading authority require separate architecture; a calculated discrepancy conveys no trading authority.

These limits are part of the contribution. They identify where a simple memory-mapped vector is sufficient and where stronger publication, lifecycle, or authority semantics are required.

29 Open-source research sequence

Starting from the current public version, development can proceed in bounded stages:

1. stabilize the published format, layout and catalog schemas, Rust implementation, synthetic fixture, and current conformance tests;
2. add portable readers and an ABI corpus that proves the same bytes have the same meaning outside Rust;
3. extend the reproducible single-process benchmarks to point counters, vector retries, concurrent writers and readers, tail latency, x86-64, and aarch64;
4. implement a general immutable generation manifest with reader pinning, crash recovery, and optional changed-shard publication;
5. formalize typed pure and stateful function-cell interfaces, dependency identity, change propagation, and replay equivalence;
6. build the ETF/index example through fair value and discrepancy as a deterministic reference application;
7. specify decision intents and a simulated effect adapter before any connection to live execution.

Each stage should be useful independently. A community member may only need a fast stable-layout shared vector, a factor researcher may use the replay/live hypercube interface, and a systems researcher may focus on multi-slice generation consistency. The project does not need to expose trading authority to make its memory and composition model valuable.

30 Conclusion

Slice reduces current analytical state to a small, inspectable design: fixed typed cells, stable entity slots, versioned headers, and reader-lockless publication protocols. The hypercube restores the dimensions that a raw vector alone cannot express—entity, primitive field, calculated node, and generation—while keeping common operations compatible with ordinary linear algebra. Applications may add factor, horizon, universe, and provenance coordinates without changing the version-1 core.

The concurrency design is intentionally plural. Per-slot counters serve independently changing point state and slice epochs serve stable vector scans. Immutable generation manifests are the proposed stronger class for coherent multi-vector decisions. This avoids locking an entire analytical universe for every market change without pretending that all consumers can accept the same consistency.

Function cells would extend the model from shared values to a richer shared computation graph. Pure and explicitly stateful nodes could compose data into returns, factors, portfolios, ETF or index values, and observed-model residuals. Decision nodes may produce bounded intents, but effects remain behind a separately governed adapter. That separation lets composability grow without allowing convenience to erase provenance, freshness, or authority.

The germinal system is therefore already present: a live memory plane, an aligned multidimensional calculation model, and versioned outputs that can be read repeatedly at low cost. The open-source work is to make its guarantees portable, its generation semantics complete, its benchmarks representative across deployment classes, and its function interface explicit.

References

[1] The Open Group, *POSIX.1-2024: mmap—Map Pages of Memory*, https://pubs.opengroup.org/onlinepubs/9799919799/functions/mmap.html.

[2] Apache Arrow Project, *Arrow Columnar Format*, https://arrow.apache.org/docs/format/Columnar.html.

[3] M. Stonebraker, P. Brown, D. Zhang, and J. Becla, *SciDB: A Database Management System for Applications with Complex Analytics*, Computing in Science & Engineering, 15(3), pp. 54–62, 2013, https://doi.org/10.1109/MCSE.2013.19.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN Usage*, ACM Transactions on Mathematical Software, 5(3), pp. 308–323, 1979, https://doi.org/10.1145/355841.355847.

[5] Linux Kernel Documentation, *Sequence Counters and Sequential Locks*, https://www.kernel.org/doc/html/latest/locking/seqlock.html.

[6] Linux Kernel Documentation, *RCU Concepts*, https://docs.kernel.org/RCU/rcu.html.

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, *A Critique of ANSI SQL Isolation Levels*, SIGMOD, pp. 1–10, 1995, https://doi.org/10.1145/223784.223785.

[8] U. A. Acar, G. E. Blelloch, and R. Harper, *Adaptive Functional Programming*, ACM Transactions on Programming Languages and Systems, 28(6), pp. 990–1034, 2006, https://doi.org/10.1145/1186632.1186634.

[9] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, *Differential Dataflow*, CIDR, 2013, https://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.

[10] B. M. Gunduz, *Low-Latency Programming Repository for High-Frequency Trading*, commit `5783606`, 2026, https://github.com/0burak/imperial_hft.

[11] Criterion.rs Project, *Criterion.rs: statistics-driven microbenchmarking*, https://bheisler.github.io/criterion.rs/book/.

[12] B. P. Welford, *Note on a Method for Calculating Corrected Sums of Squares and Products*, Technometrics, 4(3), pp. 419–420, 1962, https://doi.org/10.1080/00401706.1962.10490022.

[13] T. F. Chan, G. H. Golub, and R. J. LeVeque, *Algorithms for Computing the Sample Variance: Analysis and Recommendations*, The American Statistician, 37(3), pp. 242–247, 1983, https://doi.org/10.1080/00031305.1983.10483115.

[14] The memmap2 Project, *MmapMut flush and flush_async contracts*, https://docs.rs/memmap2/latest/memmap2/struct.MmapMut.html.

[15] The Rust Project, *Portable SIMD (nightly experimental API)*, https://doc.rust-lang.org/nightly/std/simd/struct.Simd.html.

[16] The Rust Project, *Code generation target-feature attribute*, https://doc.rust-lang.org/stable/reference/attributes/codegen.html.

[17] M. López de Prado, *Advances in Financial Machine Learning*, Wiley, 2018, https://doi.org/10.1002/9781119482086.

[18] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*, NeurIPS, 2017, https://proceedings.neurips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.

[19] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, *CatBoost: Unbiased Boosting with Categorical Features*, NeurIPS, 2018, https://proceedings.neurips.cc/paper/2018/hash/14491b756b3a51daac41c24863285549-Abstract.html.

[20] P. J. Huber, *Robust Estimation of a Location Parameter*, Annals of Mathematical Statistics, 35(1), pp. 73–101, 1964, https://doi.org/10.1214/aoms/1177703732.

[21] R. Koenker and G. Bassett, *Regression Quantiles*, Econometrica, 46(1), pp. 33–50, 1978, https://doi.org/10.2307/1913643.

[22] Scikit-learn Project, *Probability calibration*, https://scikit-learn.org/stable/modules/calibration.html.