Hypercube Circuits: realtime linear algebra with stateful triggers and replay

research@strategynet.ai

August 2026

Abstract

Hypercube calculates factors and scores for a changing set of instruments. This paper adds callbacks that run after each completed generation. A callback can calculate spreads between factor outputs, wait for a threshold to hold across several updates, use a separate exit threshold, and invalidate a signal when an input disappears. It returns current state and entry, exit, or invalidation events for use by monitors, alerts, and strategy services. A bounded Disruptor ring preserves order and reports `RingFull` when callbacks cannot keep up. The same run can be written to JSON Lines and replayed through a fresh engine to check the calculations and callback state. The demo replays 40 generations, 6,400 factor cells, 1,280 trigger-state cells, and 78 transitions without a mismatch. The current implementation excludes Aeron, raw-feed replay, checkpoints, circuit-runtime benchmarks, and execution.

Contents

1 Using Hypercube Across Updates

1.1 Why the circuit exists

Hypercube publishes its memory-mapped output as Slice files. It already calculates fields, factors, ranks, basket discrepancies, and residuals for one complete generation [1]. It does not retain application state between generations. It therefore cannot remember that a condition has held for three updates, distinguish a new entry from an active signal, or rebuild that state from a recording.

Those omissions become visible in an ordinary factor application. Let $f_{i,g}$ and $h_{i,g}$ be two calculated factors for entity i at generation g. A linear Hypercube node can calculate a spread

$$d_{i,g} = f_{i,g} - h_{i,g}.$$

A trading, alerting, or monitoring application rarely wants a new action on every update for which $d_{i,g} > a$. It usually wants a lifecycle:

$$\text{inactive} \rightarrow \text{qualifying} \rightarrow \text{active} \rightarrow \text{exited or invalidated}.$$

That lifecycle requires state across generations. The circuit runs after Hypercube completes a generation, retains the qualifying count and active state for each entity, and reports only genuine state changes.

1.2 Programs in the repository

The repository contains four runnable examples [7]:

Program	Calculation	Output
browser server	stock factor graph over a changing cross-section	Slice files, a JSON snapshot, and an SSE dashboard
ETF monitor	constituent returns, basket fair values, premiums, and ranks	ETF-aligned fair value and premium vectors
pairs monitor	cointegrating residuals, standardized scores, half-life, and rank	pair-aligned residual and opportunity vectors
record/replay	factor calculation plus a persistent threshold callback	a JSON Lines recording, current trigger state, transitions, and a verification report

The ETF and pairs programs show the calculations. The browser shows current output. The replay program adds the stateful callback and recording used in this paper. They run separately but use the same Hypercube update and snapshot types.

1.3 Keep state, recordings, and effects separate

The implementation keeps three outputs separate:

1. **current state**: the latest factor values and callback state;
2. **recording**: the ordered updates needed to rebuild that state;
3. **effects**: alerts, database writes, or orders sent by a separately authorized adapter.

A Disruptor ring only hands work to a callback. A Slice file only exposes the latest value. Neither is a recording. Replay recalculates state but must not resend alerts or orders.

2 Processing Each Hypercube Update

2.1 Process a complete generation

For generation g, the pure engine receives

$$U_g = (g, t_g, m_g, R_g, N_g)$$

and emits a coherent snapshot

$$H_g = E(U_g) = (g, t_g, m_g, n_g, V_g, Q_g).$$

Here R_g is the complete entity-row cross-section, N_g is the declared node graph, V_g is the ordered calculated value set, and Q_g is node status. The circuit does not subscribe to individual cell writes. Its event is the immutable H_g, after the engine has accepted the complete update.

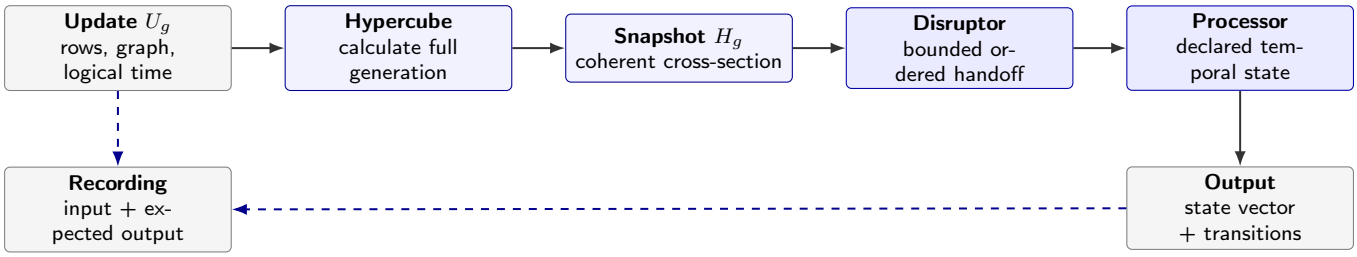

Figure 1: Hypercube finishes a generation before the callback receives it. The input and callback output are written to the recording.

The per-frame context is

$$K_g = (q_g, g, t_g, m_g),$$

where q_g is the circuit sequence. Stateful code uses g and t_g as logical coordinates. The state transition function consumes these logical coordinates. Processor scheduling time, operating-system time, and measured calculation duration are recorded separately as operational observations.

2.2 Ordered callback execution with Disruptor

The LMAX Disruptor separates event storage, sequencing, producer claims, consumer progress, and dependency graphs around a preallocated bounded ring [2]. The Rust `disruptor` crate implements the same family of patterns for low-latency inter-thread communication and recommends nonblocking publication when the application must handle a full ring explicitly [3].

The current circuit chooses a single producer because Hypercube generations already have one ordered owner. One stateful processor consumes them in sequence. The ring offers:

- a preallocated handoff with fixed capacity;
- a circuit sequence independent of wall-clock timestamps;
- nonblocking submission with a visible `RingFull` result;
- configurable sleep, spin-loop, and busy-spin waiting;
- an asynchronous submit/receive API and a lockstep correctness path.

It does *not* make an overloaded consumer current. If producer sequence p, consumer frontier c, and ring capacity B satisfy

$$p - c \geq B,$$

the next nonblocking publication is rejected. That failure is valuable information: the consumer cannot keep up and latency would otherwise grow behind the producer.

2.3 When the callback falls behind

The circuit never silently overwrites an unprocessed correctness-lane generation. On a full ring, an owner must choose one of three policies:

1. backpressure the producer and preserve every accepted generation;
2. retain normalized input durably and catch up from its log position;
3. coalesce or drop only a separately named best-effort view.

`CaptureSession` and `ReplayRunner` use lockstep processing. The engine does not accept the next update until callback processing and the recording step complete. Where an upstream market feed cannot be slowed, the feed should be archived before generation assembly. The circuit is then a deterministic consumer, while the durable archive remains the system of record.

The ring does not provide raw-feed storage, cross-host recovery, multi-stream generation assembly, or duplicate protection for external effects.

2.4 Requirements for repeatable callbacks

A processor is a state transition function

$$(S_g, O_g) = P(S_{g-1}, K_g, H_g),$$

where S_g is private declared state and O_g is the published deterministic output. Replay is meaningful when:

1. P reads only S_{g-1}, K_g, H_g, and immutable configuration;
2. iteration and output ordering are canonical;
3. missing data and reset behavior are specified;
4. nondeterministic I/O, random sources, and wall time stay outside P;
5. irreversible effects consume O_g through another adapter.

Logical ordering has a longer history than this implementation. Lamport showed that event ordering can be represented independently of synchronized physical clocks [4]. Hypercube does not implement a distributed Lamport clock, but it applies the relevant separation: generation and circuit sequence determine processing order, while observation time describes the input.

2.5 Persistent threshold callback

The first processor watches one calculated node for every entity. A trigger specification is

$$T = (\mathrm{id}, \mathrm{node}, a, b, p),$$

with entry threshold a, exit threshold b, required persistence p, and strict hysteresis $a > b$.

For an inactive entity with observed value $x_{i,g}$, the qualifying count is

$$k_{i,g} = \begin{cases} k_{i,g-1} + 1, & x_{i,g} \geq a, \\ 0, & x_{i,g} < a. \end{cases}$$

The state becomes active when $k_{i,g} \geq p$. An active state remains active through the hysteresis band and exits only when

$$x_{i,g} \leq b.$$

If the watched node has no value for an active entity, the state is *invalidated*; exit still requires the declared exit threshold. If a qualifying but inactive entity is missing, its count is removed, so a later value cannot falsely complete a "consecutive" run across a gap.

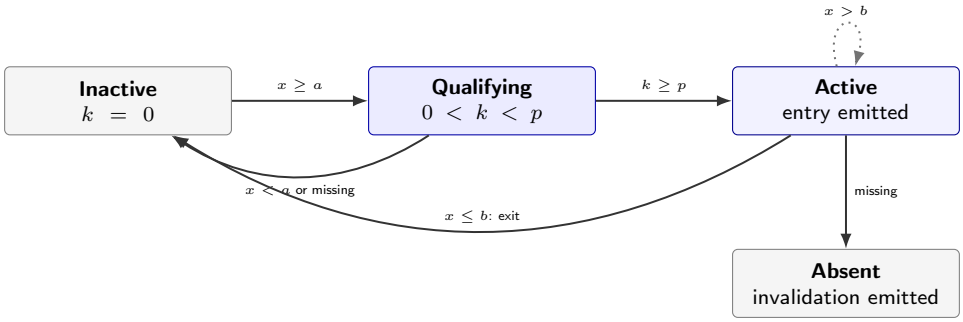

Figure 2: Persistent upper-threshold trigger with hysteresis and explicit missing-data invalidation.

2.6 Publish current state and transitions

For every generation, the processor emits:

$$Z_g = \{(\mathrm{trigger}, \mathrm{node}, i, \mathrm{active}_{i,g}, k_{i,g})\}$$

and

$$\Delta Z_g = \{\mathrm{Entered}, \mathrm{Exited}, \mathrm{Invalidated\ events\ at}\ g\}.$$

Z_g is the complete ordered trigger-state cross-section. It maps naturally to a new Slice vector and lets a late subscriber bootstrap without replaying the entire run. ΔZ_g is the sparse edge stream for a subscriber that already holds Z_{g-1}. Recording both views also exposes latent state divergence even when no transition happens in a generation.

An application can write Z_g to a Slice vector so a new subscriber can read current state immediately. Existing subscribers can consume ΔZ_g to receive only entries, exits, and invalidations. Neither consumer needs to rerun the factor calculation or the threshold callback.

3 Recording and Replay

3.1 What to record

"Replay the system" is incomplete until the recording scope is named.

Recording scope	What replay reconstructs	Tradeoff
published factor frame	callbacks, strategies, and downstream state only	smallest record; cannot verify factor calculation
complete Hypercube update	factor graph, values, callback state, and transitions	implemented compromise; repeats rows and graph per generation
normalized raw events plus generation seals	ingestion, generation assembly, factors, and callbacks	rebuilds the most; requires ordering, watermark, and join rules

The current implementation records complete Hypercube updates. It is more useful than factor-only replay because changes to rows, node graphs, transforms, or arithmetic can be detected. It stops short of claiming that vendor normalization and multi-feed generation assembly are reproducible.

3.2 Recording format

A recording is

$$\mathcal{D} = (M, G_0, G_1, \ldots, G_n).$$

The manifest

$$M = (v, \text{run}, \text{build}, \text{config}, \text{layout}, \text{triggers}, \text{metadata})$$

identifies the schema and non-generation configuration. One generation record is

$$G_g = (q_g, U_g, F_g, D_g, Z_g, \Delta Z_g),$$

where F_g is an optional map from stable upstream stream identity to the last position included in the generation and D_g is the expected semantic snapshot digest.

The development adapter writes one manifest and then one generation per JSON line. It validates:

- exactly one leading version-1 manifest;
- contiguous circuit sequences and increasing generations;
- agreement between update and expected output generation;
- valid trigger configuration and nonempty provenance fields;
- only finite JSON numeric inputs.

JSON parsing enables exact finite $f64$ round trips. NaN and infinity are rejected before the engine advances because JSON would otherwise encode them as `null`; a missing primitive is represented by omitting its field. This inspectable format serves the current recording path; a low-latency wire encoding requires a separate binary schema.

3.3 What replay compares

The semantic snapshot digest includes:

- generation, observation time, and entity count;
- ordered node and entity identities;
- exact IEEE-754 value bits and value observation times;
- ordered status identities and value/missing counts.

It excludes:

- `ExecutionMode`, because `Live` becomes `Replay`;
- measured node compute duration;
- wall time, thread identity, transport delay, and logging details.

The current digest is a versioned streaming xxHash64 over a canonical byte representation. Digest equality is supplemented by exact comparison of the full ordered state and transition outputs. The build and configuration identities remain provenance: a different compiler, target, or implementation may be replayed, and every arithmetic difference is reported without normalization.

Replay compares calculated values and state transitions. It does not compare processor scheduling, compute duration, or transport timing.

3.4 Replay procedure

Replay performs the following bounded procedure:

1. validate and read the recording manifest;
2. construct a new `HypercubeEngine`;
3. construct a new processor from the recorded trigger configuration;
4. set each recorded update mode to `Replay`;
5. calculate the snapshot and process it in lockstep;
6. compare circuit sequence, semantic digest, complete state, and transitions;
7. count every divergent generation and retain the first concrete expected-versus-actual difference.

Formally,

$$H'_g = E'(U_g[m \leftarrow \text{Replay}]),$$

$$(S'_g, O'_g) = P'(S'_{g-1}, K'_g, H'_g).$$

An exact replay requires

$$q'_g = q_g, \qquad D(H'_g) = D_g, \qquad Z'_g = Z_g, \qquad \Delta Z'_g = \Delta Z_g$$

for every recorded generation.

Research on deterministic component replay likewise separates virtual or logical ordering from incidental runtime scheduling [5]. The Hypercube implementation is narrower: it records already coherent application updates and executes one ordered stateful lane.

3.5 Mismatch reporting and exit codes

Malformed files, invalid updates, and processor failures are operational errors. A valid recording that produces a different digest, state, or transition is a semantic result. The CLI therefore uses separate exits:

Exit	Meaning
0	every recorded generation matched
1	operational or recording-format failure
2	replay completed with semantic divergence

This distinction supports two workflows:

- **reconstruction**: expect zero divergence under the same implementation and configuration;
- **counterfactual replay**: intentionally change a node, threshold, or model and analyze the resulting divergence.

3.6 Disable external effects during replay

Replay data must not reach an adapter that can send an order, notification, or database write as though the data were live. The current implementation contains no execution adapter, and callbacks perform calculations only.

Future replay publication should use a namespace such as

`replay/{run_id}/...`

and every effect adapter must reject replay-mode inputs. A live order, notification, or database mutation should also carry an idempotency key and a reference to the generation that produced it.

4 Deployment

4.1 Use Aeron between processes

Inside one compute process, Aeron is unnecessary for the engine-to-callback handoff. The immutable snapshot is already in memory; the Disruptor provides ordered inter-thread transfer without serialization or a media driver.

Aeron becomes useful between processes or hosts:

- factor/state publication to independent services;
- durable capture of normalized inputs or generation envelopes;
- subscriber recovery from a known stream position;
- controlled replay onto a dedicated channel.

Aeron Archive records and replays streams from durable storage and supports replay from a specified recording and position [6]. The current crate does not depend on Aeron. Its public manifest and generation structs are transport-neutral envelopes that a future SBE adapter can encode as:

```
hypercube.recording.manifest.v1
hypercube.recording.generation.v1
```

4.2 Record upstream stream positions

For k upstream streams, generation g may record

$$F_g = (p_{1,g}, p_{2,g}, \ldots, p_{k,g}),$$

where $p_{j,g}$ is the last input position included from stream j. The manifest can carry Archive recording identities and start positions; `source_positions` carries the per-generation frontier.

These positions show which upstream messages were included in a generation. The feed adapter must still define:

- the canonical order when inputs from different streams race;
- event-time and ingestion-time watermark rules;
- late, duplicate, corrected, and missing event behavior;
- the generation seal that says no more input belongs to g.

Without those rules, Hypercube can replay complete updates but cannot rebuild them from raw feeds.

4.3 Deployment options

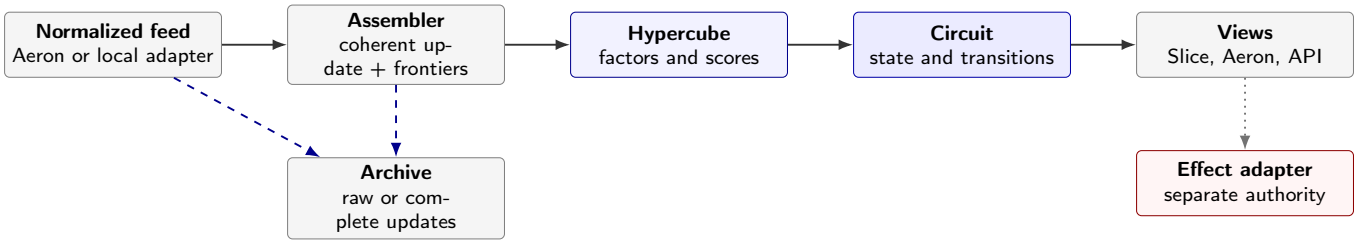

Figure 3: Target deployment: live calculation (solid), recording (dashed), and separately authorized effects (dotted).

All components can run in one process. A split service can keep Hypercube and the callback together while publishing state over Aeron. A deployment that must rebuild from raw input can archive normalized messages before generation assembly and publish replay on a separate channel.

5 Examples

5.1 ETF basket premiums

The ETF monitor reduces 160 correlated constituent returns into 12 basket fair values and then evaluates five ETF-aligned Hypercube nodes. For ETF j,

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

while the simulated market premium follows

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

The graph publishes fair value, market price, premium basis points, model z-score, and cross-sectional premium rank. The terminal therefore displays 60 cells per generation.

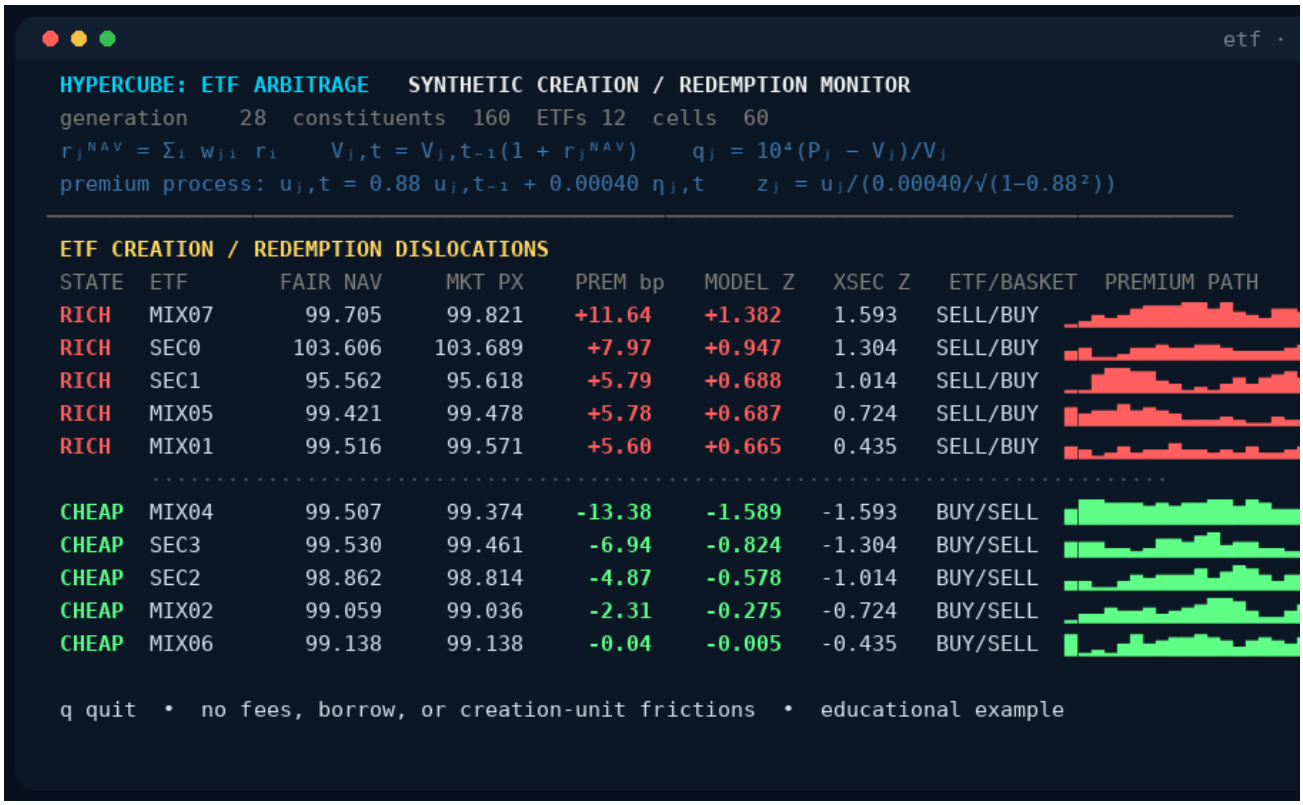

Figure 4: Checked-in ETF recording at generation 28. Labels preserve the premium sign independently of color. Fees, borrow, and creation-unit frictions remain explicit omissions.

A callback can watch `premium_z`, require it to remain above a threshold for several generations, and publish an entry or exit without changing the basket calculation. A trading component would still need inventory, quoted spread, creation-unit size, costs, and execution limits.

5.2 Pairs residuals and triggers

The pairs monitor publishes 120 cells over 24 stable pair entities. Its residual is

$$y_{j,g} = \log A_{j,g} - \alpha_j - \beta_j \log B_{j,g}.$$

It standardizes the current residual against the preceding $W = 20$ observations; the current observation enters only after scoring. With $\mathcal{W}_{j,g} = \{y_{j,g-W}, \ldots, y_{j,g-1}\}$,

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

Scoring occurs before the current residual enters the fixed-capacity rolling window, avoiding look-ahead. Model half-life and cross-sectional opportunity rank remain separate nodes.

Figure 5: Checked-in pairs recording at generation 28. The display ranks standardized dislocations while retaining hedge ratio and half-life context.

The threshold callback can enter only after the z-score remains large for several generations and exit at a lower threshold. If either leg disappears, it reports invalidation; convergence requires both legs. Out-of-sample parameter fitting, structural-break detection, borrow, costs, and portfolio limits remain outside the demo.

5.3 Browser and Slice output

The synthetic server computes five nodes for a changing stock cross-section, publishes them to entity-aligned Slice files, exposes a JSON snapshot, and streams generations through SSE to a dependency-free dashboard.

The callback adds two outputs:

- a dense current-state projection suitable for Slice or a bootstrap response;
- a sparse transition stream suitable for already synchronized subscribers.

The browser stream and Slice files contain current output only. Replay history resides in the JSON Lines recording.

5.4 Replay demo

The executable experiment is:

```
cargo run -p hypercube-circuit --bin hypercube-replay -- \
  record-demo /tmp/hypercube-factor.jsonl 40 32

cargo run -p hypercube-circuit --bin hypercube-replay -- \
  verify /tmp/hypercube-factor.jsonl
```

The synthetic graph has five nodes over 32 entities for 40 generations:

$$5 \cdot 32 \cdot 40 = 6{,}400$$

factor cells. One threshold specification yields

$$32 \cdot 40 = 1{,}280$$

complete trigger-state cells.

Compared object	Recorded	Replayed
generations	40	40
factor cells represented by semantic digests	6,400	6,400
trigger-state cells	1,280	1,280
transitions	78	78
divergent generations		0

The file contains 41 lines: one manifest and 40 generation records. This demonstrates functional exactness. Throughput and storage require separate benchmarks. The repository also includes an animated Hypercube terminal demo that shows qualifying state, entries, exits, missing-data invalidation, and the final replay comparison.

5.5 Failure cases covered by tests

The circuit tests do more than assert the happy path:

- a blocked consumer fills a two-slot ring and the third generation returns `RingFull` without overwriting pending work;
- a missing inactive candidate resets persistence, while a missing active entity emits `Invalidated`;
- sequence gaps and non-increasing generations are rejected;
- non-finite JSON input is rejected before a generation is written;

- live and replay JSON values reproduce exact floating-point bits;
- a changed recorded primitive produces a snapshot-digest divergence;
- unchanged input reconstructs snapshot, state, and transitions.

The tests cover overload, sequence gaps, missing observations, serialization, state reset, and output comparison.

5.6 Test results

The workspace currently runs 37 tests:

Suite	Tests	Result
engine, publisher, and synthetic injectors	7	passed
fixed-capacity rolling moments	6	passed
ETF and pairs example invariants	2	passed
Slice layout, mmap, records, and vector algebra	8	passed
circuit, triggers, recording, digest, and replay	14	passed
Total	**37**	**passed**

The workspace also passes formatting, clippy with warnings denied, public API documentation with warnings denied, and the independently verifiable `hypercube-slice` package check. The engine and circuit follow their local dependencies in registry publication order. These are conformance and packaging checks; the same all-target suite also passes under the declared minimum compiler, Rust 1.81.0. This circuits paper still makes no latency or throughput claim.

6 Practical Use and Next Steps

6.1 Factor-to-signal service

A live service can run the components in this order:

1. a feed adapter assembles one complete Hypercube update;
2. Hypercube calculates fields, spreads, ranks, residuals, and scores;
3. the circuit runs its callbacks on the completed snapshot;
4. the caller writes current callback state to Slice or an API and sends transitions to alerting or strategy consumers;
5. the capture session records the update and callback output for replay.

For example, suppose an ETF premium z-score must be at least 1.5 for three consecutive generations and exits at 0.5. Hypercube calculates the z-score. The callback increments a per-ETF count, emits `Entered` on the third qualifying generation, keeps the state active while the value remains above 0.5, and emits `Invalidated` if the ETF value disappears. A new consumer reads the complete current state; an existing consumer can process only the transition events. The recording lets an operator rerun the same generations when the factor code or trigger code changes.

6.2 Test rule changes against recorded data

After an exact replay passes, another run can intentionally:

- change entry, exit, or persistence parameters;
- replace one factor node or model version;
- compare old and new state transitions by generation;
- inject delayed or missing data according to a declared scenario;
- measure proposed decision intents without enabling live effects.

The current CLI verifies only the configuration stored in the recording. A counterfactual command would accept a different graph or trigger configuration, mark the run as non-exact, and report the first generation whose values or transitions changed.

6.3 Current limits

The current code covers one complete calculation-and-replay path:

- JSON Lines is the only implemented recording adapter; specifications also describe Aeron Archive and SBE mapping;
- replay consumes complete updates; raw vendor events and generation assembly remain upstream;
- one threshold processor is provided; a reusable rolling-moment primitive now exists, but general callback composition, serialized window state, and model artifacts need additional schemas;
- replay begins at generation zero and currently offers no state checkpoints or seek operation;
- xxHash64 detects ordinary divergence; cryptographic authenticity requires an authenticated digest or signature;
- state vectors are public structs but no dedicated trigger-to-Slice publisher is included;
- no circuit benchmark justifies a wait strategy, ring size, or deployment topology for production;
- no external-effect or trading authority exists in the circuit crate.

The `verify` command checks calculations. Production recovery requires checkpoint and seek support. Trigger state records data; order creation requires separate trading authority and execution integration.

6.4 Next steps

The next implementation work is:

1. publish trigger state to Slice and transitions onto a named subscriber channel;
2. add an SBE codec and record the envelopes through Aeron Archive;
3. define multi-stream generation seals and late-event handling;
4. add processor checkpoints and verified seek;
5. add counterfactual configuration and first-mismatch comparison;
6. benchmark circuit latency, backlog, CPU, allocation, and recovery time for each supported deployment;
7. add an idempotent simulated effect adapter before considering live execution.

6.5 Conclusion

Hypercube can now calculate a complete generation, pass it to an ordered stateful callback, return current state and transitions, record the input and output, and verify the result with a fresh process. The included threshold callback handles persistence, separate entry and exit levels, and missing inputs.

The immediate use is a factor-to-signal service for monitors, alerts, and strategy inputs. ETF and pairs provide calculation examples, the browser provides current output, and the replay demo checks 40 recorded generations. Publishing callback state to Slice is the next practical addition. Aeron transport, raw-feed reconstruction, checkpoints, circuit benchmarks, and execution remain future work.

References

[1] StrategyNet, *Hypercube: Current analytical state without rebuilding it for every reader*, Open-source working paper, August 2026, https://github.com/ingvor-restrat/hypercube/blob/main/docs/latex/hypercube_foundations.pdf.

[2] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, *LMAX Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads*, LMAX Exchange, May 2011, https://lmax-exchange.github.io/disruptor/disruptor.html.

[3] *disruptor 4.3.0: Low latency inter-thread communication via a ring buffer*, Rust crate documentation, https://docs.rs/disruptor/4.3.0/disruptor/.

[4] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM, 21(7), pp. 558–565, 1978, https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/.

[5] R. Strom, C. Dorai, T. H. Feng, and Z. Wei, *Deterministic Replay for Transparent Recovery in Component-Oriented Middleware*, IEEE ICDCS, 2009, https://doi.org/10.1109/ICDCS.2009.79.

[6] Aeron Project, *Aeron Archive Overview*, https://aeron.io/docs/aeron-archive/overview/.

[7] StrategyNet, *Hypercube public repository*, https://github.com/ingvor-restrat/hypercube.