**Final Plan:  SevenDB**
draft the deterministic emission contract and the client reconnect handshake (compact spec + example protobuf/messages).

# 1. Vision

**We want a reactive database that:**

- Pushes updates to clients continuously, not just answers queries once.
- Replicates safely (linearizable semantics, no timing anomalies).
- Scales compute beyond one machine by sharding subscription workloads.
- Handles failures gracefully with deterministic recovery and instant failover.
- Current systems (e.g. Materialize, Flink, Postgres replication) solve parts of this problem, but nobody integrates replication, determinism, and compute sharding at the database core.

# 2. Core Problems to Solve

**Timing anomalies in replication**

- Subscribes/unsubscribes may interleave differently at replicas.
- Without a strict order, replicas disagree on client state.

**Compute bottlenecks**

- Reactive queries consume CPU continuously.
- If every replica recomputes everything, cost explodes.
- If only one replica computes, failover is slow.

**Leader overload**

- Making the Raft leader also responsible for client emissions risks hot spots.
- Snapshots & re-sharding
- Large arrangements must move between nodes without blocking or downtime.

**Determinism fragility**

- Floating-point differences, planner nondeterminism, or hash iteration order can cause divergent results.

# 3. Key Design Decisions

## 3.1 Replicated Buckets

- Partition subscriptions + data into buckets.
- Each bucket is managed by a small Raft group (RF=3 typical).
- The bucket log records everything: data updates, SUBSCRIBE, UNSUBSCRIBE, config changes, notifier leases.
- This log is the single source of truth ⟶ all replicas replay in the same order.

## 3.2 Deterministic Subscription Semantics

- Sub/Unsub are events in the log with their own commit index.
- A client is considered subscribed starting at that index across all replicas.
- Emissions carry emitSeq = commit_index.
- Clients dedupe based on (sub_id, emitSeq).

## 3.3 Notifier Election (Decoupled)

- Notifier ≠ Raft leader.
- Election is deterministic (e.g., rendezvous hash) + confirmed by replicated lease written to the log.
- This prevents Raft leaders from being overloaded by client traffic.

### 3.4 Hot vs Cold Replication Modes

- **Hot mode:** all replicas shadow-evaluate ⟶ instant failover, higher CPU cost.
- **Cold mode:** only notifier evaluates; followers store checkpoints + replay on failover.
- Policy is per-bucket, not mixed within a bucket (simplifies SLA enforcement).

## 3.5 Deterministic Execution Rules

- Canonical traversal (sort keys, fixed hash functions).
- Disallow nondeterministic SQL functions, or log their seeds.
- Planner must emit identical query plans across replicas (deterministic rules, no random tie-breaking).

## 3.6 Snapshots & Migration

- Periodic non-blocking checkpoints (zig-zag / copy-on-write).
- For re-sharding buckets:
    - **Option A:** Raft membership changes (like CockroachDB).
    - **Option B:** Transaction crabbing — dual execution during migration.
- Migration must not block client updates.

# 4. System Components

### Cluster Manager

- Assigns buckets to replica groups.
- Manages configuration epochs.
- Runs autoscaler.

### Bucket Raft Log

- Appends all events (data + control).
- Defines commit order = global truth per bucket.

### Replica Node

- Replays bucket log deterministically.
- Maintains arrangements (incremental indexes).
- Applies SUBSCRIBE/UNSUBSCRIBE.
- Shadow-evaluates in hot mode or stores checkpoints in cold mode.

### Notifier (per bucket)

- Holds lease, emits deltas to clients.
- Lease expiration ⟶ deterministic re-election.

### Arrangements (Shared Indexes)

- Incremental differential structures reused across subscriptions.
- Multi-query optimization ⟶ compute once, fan out.

### Checkpoint Store

- Durable snapshots of arrangements and subscription state.

### Client Gateway

- Routes clients to the correct notifier.
- Handles reconnection + dedup.

# 5. Runtime Flow

### SUBSCRIBE

1. Client → gateway → chosen bucket.
2. SUBSCRIBE appended to bucket log.
3. Commit confirmed; replicas install subscription state.
4. Notifier with lease sends SUB_ACK(sub_id, commit_index).

### DATA_EVENT

1. Update appended to log.
2. Replicas apply update deterministically.
3. Notifier computes delta and emits with emitSeq=commit_index.
4. Followers shadow-evaluate (hot) or just record checkpoint (cold).

### UNSUBSCRIBE

1. Appended to log.
2. Commit → notifier stops emitting deltas after that index.

### Failover

1. Notifier lease expires or node crashes.
2. Replicas deterministically pick new notifier.
3. New notifier resumes emission from last committed index.

# 6. Optimizations

**Arrangements:** Share indices across subscriptions.

**Delta batching:** Group small changes before emitting.

**Adaptive placement:** Re-shard hot buckets dynamically.
Hotness score = f(event rate, CPU per sub, arrangement memory).

**Approximate subs (optional):** sample results for cost-sensitive clients.

# 7. Testing Strategy

### Simulation harness (deterministic environment):

- Controls time, scheduling, network partitions.
- Replay logs in randomized orders to catch nondeterminism.

### Core tests:

- Determinism test: two replicas replay same log → identical outputs.
- Subscribe/unsubscribe race tests.
- Failover correctness (no gaps, no duplicates).
- Snapshot + recovery test.
- Re-sharding test under load.

# 8. Roadmap

### Phase 0 – Simulation First

1. Deterministic harness.
2. Single-node engine runs inside it.

### Phase 1 – Local Reactive Engine

1. Incremental arrangements, SUBSCRIBE/UNSUBSCRIBE.
2. Local WAL + checkpoint.

### Phase 2 – Replicated Buckets

1. Raft per bucket.
2. Append SUBSCRIBE/UNSUBSCRIBE/DATA events.
3. Commit order = emission order.

### Phase 3 – Notifier Election (Lease-based)

1. Decouple notifier from Raft leader.
2. Implement replicated lease system.

### Phase 4 – Hot vs Cold Replication

1. Shadow-evaluation mode.
2. Cold mode with checkpointing.

### Phase 5 – Migration

1. **Non-blocking checkpoints.**
2. **Experiment with Raft membership vs transaction crabbing.**

### Phase 6 – Optimizations

1. Operator sharing, delta batching, hotness scores.

### Phase 7 – Autoscaling & Hardening

1. Re-sharding, chaos tests, metrics.

## 9. Evaluation Plan

- **Correctness**: determinism, linearizable sub semantics.
- **Performance**: throughput (events/sec), latency (commit→emit), failover time.
- **Scalability**: #subs handled vs cluster size (near-linear growth).
- **Comparisons:**
    - Materialize (streaming SQL, external Kafka).
    - Flink/Kafka Streams.
- **Novelty proof**: show DiceDB handles replication + reactive subs natively, unlike prior systems.

## 10. Why This Is Novel

- Replication inside the DB, not outsourced.
- Deterministic subscription semantics with log-indexed subscribe/unsubscribe.
- Decoupled notifier election for scalability.
- Hot vs cold replication tradeoff as a tunable knob.
- Operator sharing for efficient multi-sub workloads.
- This hasn't been done in one system — that's our unique contribution.

# MUST-FIX (things that should be in the plan before coding heavy parts)

### Precise semantics for delivery guarantees.

Explicitly state if client emissions are at-most-once, at-least-once, or effectively-once (deduped). our (sub_id, emitSeq) dedupe implies at-most-once from client POV, but make it explicit and show how we handle retries, network duplicates, and partial acks.

### Commit index monotonicity & index source-of-truth.

Define whether commit_index is per-bucket monotonic (sounds like it) and how we compare indices across re-sharding/migration boundaries. Re-sharding may change index spaces — plan a canonical global ordering or a mapping to preserve emitSeq semantics.

Log retention / compaction / tombstones.

How long will the bucket log be kept? Define compaction and snapshot policies, and how they interact with followers in cold mode and with clients reconnecting from old checkpoints.

### Client reconnection & exactly-once/dedup edge cases.

Define the reconnect handshake and how clients convey last-seen emitSeq so notifiers can resume. Handle the case where a client reconnects and ignores older SUB_ACK or duplicates.

### Determinism plan for floats & non-deterministic libs.

Decide: ban floats, use deterministic FP library (rounding-mode control), or convert critical calculations to fixed-point. Put explicit rules in 3.5 and plan tests that vary hardware/compilation flags.

### Planner determinism enforcement.

Specify how the planner’s determinism is validated: plan hash with canonical serialization, CI gate that runs planner in multiple process seeds and compares output. No “we hope it’s deterministic.”

# HIGHLY RECOMMENDED (important but can be iterative)

### Exactly-Once-ish emission contract + formal proof sketch.

Add a short formal model: logs → commit indices → emitSeq mapping → client handshake → dedupe. This makes our "novelty" claim defensible.

### Lease & election timing bounds and fallbacks.

Define lease TTLs, renewal strategy, and deterministic backoff for split-brain cases. Add a safety mechanism: if lease write to log fails repeatedly, enter a safe-suspend mode to avoid double-emits.

### Failure modes matrix & SLOs.

For each failure (notifier crash, network partition, slow client storm, leader partition), add expected outcome, invariants preserved (no gaps, no duplicates), and SLOs (e.g., failover ≤ X ms in hot mode, ≤ Y ms after cold mode replay).

### Instrumentation & tracing plan.

Add distributed tracing correlation (trace_id through bucket log entries), per-sub metrics (buffer_occupancy, ack_latency), and per-notifier CPU/egress metrics. Make metrics part of acceptance tests.

### Admission control / multi-tenancy quotas.

Add tenant/tenant-group quotas so one tenant's best-effort storm can't starve others. Tie into Cluster Manager decisions.

### Security & auth model.

Outline client auth, TLS for gateway→notifier and inter-node RPC, and per-tenant isolation to avoid noisy-neighbor attacks.

### Migration invariants & tests.

For Option A vs B, codify invariants: no lost subscriptions, no double-emits, and client-visible continuity. Add small, deterministic migration experiments before full re-shard implementation.

### Precise backpressure policy language.

Convert the backpressure ladder into a small state-machine and policy DSL (e.g., when occupancy > 0.9 ⟶ start coalescing; >0.95 ⟶ micro-batching; >0.99 ⟶ suspend). Makes testing deterministic.

# Phased Backpressure Plan (mapped to our milestones)

### Phase BP-A — Essential v1: Client & Per-Sub Flow Control

**Where to add:** during Milestone 4 ⟶ 6 (Raft per bucket, Notifier election, Hot/Cold modes).
**Goal**: make the system safe and predictable for simple slow clients with tiny implementation overhead.

### What to implement

- Client modes: ack and window (client chooses on SUBSCRIBE).
- ack: notifier waits for explicit ACK for each batch.
- window: client advertises window_size = max in-flight batches.
- Per-subscription queue (bounded): simple FIFO + batch coalescing for same-entity updates.
- Simple semantic coalescing: if multiple updates for same row/key exist in queue, keep latest (or oldest+latest policy).
- Backpressure signals: FLOW_STATUS (periodic) and ACK messages from client.
- Server behaviour under overflow: drop best-effort messages first; apply policy per subscription: critical|normal|best_effort.

### API snippets (compact)

```
SUBSCRIBE(query, params, flow_control:{
  mode: "ack" | "window",
  window_size?: int,
  buffer_size: int,
  priority: "critical"|"normal"|"best_effort"
```

```
})

FLOW_STATUS(sub_id, {
  buffer_occupancy: float,     // 0.0 - 1.0
  processing_rate: float,      // updates/sec
  last_processed_seq: uint64
})

ACK(sub_id, last_seq_processed)
```

### Acceptance criteria

- system never allocates unbounded memory for a subscription.
- slow client with window=1 receives updates in-order and no faster than it can ACK.
- coalescing reduces queue length for high-churn keys by ≥X% in synthetic tests.

### Tests

- simulated clients at varying speeds; verify no OOM and correct per-client delivery rate.
- inject hot key churn; verify coalescing behaves as policy.

## Phase BP-B — Notifier-Level Load Management

**Where to add**: during Milestone 6 → 8 (Notifier lease, Hot/Cold modes, Failover, Chaos tests).
Goal: let each notifier protect the cluster and behave reasonably under mixed client mixes.

### What to implement

- Client classification (fast, slow, unreliable, critical) inferred from recent FLOW_STATUS + ACK latency.
- **Adaptive batching**: increase batch size for slow clients, reduce for fast ones.
- **Circuit breaker**: per-client counters; if ACKs drop or RTT explodes, mark suspended and attempt periodic reconnects.
- **Notifier resource thresholds making decisions**: when CPU/memory/egress > thresholds, shift to:

    - more aggressive coalescing,
    - micro-batching (lower message frequency),
    - suspend best-effort clients.
- **Metrics & alerts**: per-notifier load, active circuit breakers, drop counts.

### Acceptance criteria

- single slow client cannot reduce throughput for >95% of other clients.
- notifier recovers & resumes suspended clients without state loss.
- under synthetic overload, best-effort clients are trimmed first.

### Tests

- multi-client storms with controlled slowdowns; assert isolation property.
- induce notifier resource saturation; assert graceful degradation ladder triggers.

## Phase BP-C — Cross-Bucket & Cluster-Level Backpressure

**Where to add**: during Milestone 9 → 11 (Optimizations, Autoscaling, Production hardening).
Goal: coordinate across notifiers and influence placement/resizing decisions.

### What to implement

- **Global backpressure signal**: notifiers periodically publish LoadReport(bucket_id, load_metrics) to Cluster Manager.
- **Cluster Manager reactions (policy-driven):** re-shard hot buckets, spin up cold replicas, or reduce update frequency for non-critical subscriptions.
- **Cascading backpressure handling**: allow downstream proxies to inject backpressure signals that propagate upstream.
- **SLA-based admission control**: deny new best-effort subscriptions if cluster load above threshold for that tenant.

### Acceptance criteria

- global action reduces overloaded notifiers' CPU by target percent (configurable) during spikes.
- re-sharding or autoscale triggered within configured policy boundaries and without violating deterministic semantics.

### Tests

- multi-bucket overload simulations; validate cluster manager choices and resulting stabilization.
- network-region slowdown (many clients in one region slow) — ensure controlled mitigation.

## Phase BP-D — Advanced Semantic & Adaptive Strategies

**Where to add**: later in Milestone 9–11 optimizations and research extensions.
Goal: squeeze efficiency and adaptivity while preserving determinism.

### What to implement

- Intelligent dropping policies: semantic-aware policies (e.g., keep oldest+latest, per-entity sampling, significance-based).
- Approximate subscriptions: optional mode for clients that accept sampled/approx updates for cost savings.

- Machine-learned capacity models (optional): predict client processing capability to preempt backpressure.
- Formalizing backpressure state in replicated logs (if needed) so failover preserves temporary client throttle state deterministically.

### Acceptance criteria

- measured egress reduction vs accuracy loss for approximate modes — expose tradeoff curve.
- ML models show measurable improvement over heuristics in production traces (if used).

### Tests

- A/B experiments on real-ish workload traces.
- Formal verification stubs for correctness when backpressure state is replicated.

### How it fits our milestone map (concise)

**Milestone 4–6**: Implement BP-A (basic client flow control + per-sub queues). This prevents OOMs and gives predictable behavior while we stabilize replication and notifier leases.

**Milestone 6–8:** Implement BP-B (notifier protection, circuit breakers) while we do failover tests and chaos engineering.

**Milestone 9–11**: Implement BP-C and BP-D (cluster-wide signals, adaptive placement, semantic dropping, autoscaling integration).

### Compact Test Matrix (must-have)

- **Determinism × Backpressure**: replay same logs across replicas with mix of slow clients ⟶ outputs and emitted emitSeq must be identical.
- **Failover × Backpressure**: notifier under backpressure fails; new notifier resumes at last commit_index and respects throttles.
- **Isolation**: single slow client cannot reduce throughput of unrelated high-priority clients.
- **Graceful degradation ladder**: micro-batching ⟶ coalescing ⟶ sampling ⟶ suspend. Verify order and correctness.

### Minimal incremental deliverable (ships fast)

If we want one small, high-leverage deliverable to include in our prototype: implement window-based flow control + bounded per-sub queues + simplest semantic coalescing. That size of work drastically reduces early operational risk and is easy to test in our deterministic harness.

**Backpressure is the "survival kit" for reactive push systems**: implement it early and iteratively. Start lean (per-sub window & bounded queues), prove it in the simulation harness, then layer notifier protections and cluster coordination as we harden the system. we already have the right milestones — this phased plan puts concrete, testable backpressure steps into them so we don't reinvent the hose while the house is on fire.

# Journey in milestones

### Milestone 1 — Deterministic Simulation Harness

- **Aim**: Build the playground where all correctness claims can be tested.
- Deterministic log replay with controlled time & randomized delivery.
- Inject events (data updates, sub/unsub) and replay them.
- **Verify**: two replicas replay → identical outputs.

**Success signal**: determinism test suite passes consistently.

### Milestone 2 — Local Reactive Engine

- **Aim**: Run subscriptions/reactive queries on a single node.
- Implement incremental arrangements (shared indexes).
- SUBSCRIBE / UNSUBSCRIBE as log events.
- Local WAL + checkpoint + recovery.

**Success signal**: can crash + restart a single node and subscriptions pick up exactly where they left off.

### Milestone 3 — Replicated Buckets (Raft Integration)

- **Aim**: Introduce real replication with linearizable semantics.
- Partition into buckets; Raft group per bucket.
- Log includes SUB/UNSUB/DATA.
- Commit index defines subscription state + emitSeq.

**Success signal**: two replicas, one crashes → remaining continues; new replica replays log and produces byte-identical outputs.

### Milestone 4 — Notifier Election (Decoupled from Leader)

- **Aim**: Separate client emission from Raft leadership.
- Deterministic notifier election (rendezvous hash + replicated lease).
- Lease written to Raft log before emission begins.
- **Success signal**: kill notifier → replicas deterministically elect same successor; client sees continuous stream with no duplicates/gaps.

### Milestone 5 — Backpressure Essentials (BP-A)

- **Aim**: Prevent OOM and uncontrolled client flooding.
- Per-sub bounded queue.
- **Client flow-control**: ack or window mode.
- Simple coalescing of hot keys.

**Success signal**: one slow client cannot crash server or block others; window=1 client gets in-order delivery at its own pace.

### Milestone 6 — Hot vs Cold Replication Modes

- **Aim**: Offer a tradeoff between CPU cost and failover speed.
- **Hot**: followers shadow-evaluate.
- **Cold**: notifier-only evaluation + checkpoint replay.
- **Success signal**: failover time measured in hot mode = instant; cold mode = replay time bounded and predictable.

### Milestone 7 — Backpressure Expansion (BP-B)

- **Aim**: Protect notifiers under mixed client loads.
- Client classification (fast, slow, best-effort).
- Circuit breakers + adaptive batching.
- Graceful degradation ladder.

**Success signal**: 100 slow clients can't harm 10 fast/critical ones. Metrics prove isolation.

### Milestone 8 — Snapshots & Migration (Re-sharding)

- **Aim**: Move large arrangements between nodes without downtime.
- Non-blocking checkpoints (zig-zag/copy-on-write).
- Experiment with Raft membership changes vs transaction crabbing.

**Success signal**: migrate a hot bucket under load; no lost subs, no duplicate events, client sees continuity.

### Milestone 9 — Backpressure Cluster-Level (BP-C)

- **Aim**: Scale protection from per-notifier to whole cluster.
- Notifiers publish LoadReports.
- Cluster Manager re-shards hot buckets / spins replicas.
- SLA-aware admission control.

**Success signal**: cluster recovers automatically from regional overload, trimming best-effort clients first.

### Milestone 10 — Optimizations Layer

- **Aim**: Make the system efficient, not just correct.
- Operator sharing (multi-query optimization).
- Delta batching.
- Hotness scoring for adaptive placement.

**Success signal**: throughput per node improves >2× vs naïve baseline.

### Milestone 11 — Backpressure Advanced (BP-D)

- **Aim**: Explore research-y extensions.
- Semantic dropping (oldest+latest, significance-aware).
- Approximate subscriptions for cost-sensitive clients.
- Optional ML-driven capacity prediction.

**Success signal**: can demo approximate vs exact mode tradeoff curves.

### Milestone 12 — Autoscaling & Hardening

- **Aim**: Turn prototype into something production-worthy.
- Cluster Manager autoscaling policies.
- Chaos tests (kill nodes, partitions, overload storms).
- Monitoring/metrics/alerts dashboards.

**Success signal**: chaos monkey runs don't break determinism, failover meets SLA, and metrics are clear enough to debug live.

### Milestone 13 — Publication & Demonstration

- **Aim**: Package novelty and show results.
- Formalize deterministic subscription semantics.
- Benchmarks vs Materialize/Flink/Postgres logical replication.
- Write paper/blog: "SevenDB: Deterministic Reactive Database with Native Replication & Backpressure."

**Success signal**: reproducible experiments + novelty claim stand up to scrutiny.

# Emission Contract & Reconnect Handshake (Improved)

### 1. Deterministic Emission Contract

**Guarantee**

At-least-once delivery, deduplicated to effective-once at the client.
All emissions are derived deterministically from the replicated log.

**Mechanism**

Every emission for a subscription is tagged with an emit_seq, defined as a composite:
emit_seq = (epoch_id, commit_index)

**epoch_id**: a monotonically increasing identifier for each bucket lifetime (e.g., after re-shard or bucket migration).

**commit_index**: the log index within that epoch.

**Retries**: The notifier retries unacked emissions until it receives a ClientAck or the subscription is terminated.

**Deduplication**: Clients track last_processed_emit_seq per subscription. Any emission with emit_seq <= last_processed_emit_seq is ignored.

**Gap Detection**: Clients must monitor emit_seq. If gaps are observed, the reconnect protocol will fill them.

## Formal Sketch

1. Let L be the ordered log for a bucket epoch.
2. For subscription S starting at (epoch_id, i_s), evaluation of log L[i_s..N] yields a deterministic stream of deltas $D_1$, $D_2$, ....
3. Each $D_x$ is tagged with (epoch_id, j) where j is the commit index that caused it.
4. Clients process ($D_x$, emit_seq) in ascending emit_seq, discarding duplicates.

### Non-Goals

- Cross-subscription atomicity is not guaranteed. Subscriptions are independent. Clients requiring aligned multi-sub views must use higher-level transactions.
- In-order delivery over the wire is not guaranteed. Clients must reorder by emit_seq.

## 2. Client Reconnect Handshake

**Goal**: Allow clients to resume after disconnection with no gaps or duplicates.

#### Assumptions

- sub_id is a stable UUID returned in SubAck.
- Clients persist (sub_id, last_processed_emit_seq) for durable subs.

#### Protocol

```
Reconnect Request
message ReconnectRequest {
  string sub_id = 1;
  uint64 epoch_id = 2;
  uint64 last_processed_commit_index = 3;
}
```

### Notifier Processing

- **Validate subscription**: Check if sub_id exists and is active.
- **Validate position**:
  - If (epoch_id, commit_index) is behind current state: resume.
  - If (epoch_id, commit_index) is too far ahead (client > notifier): reject with INVALID_SEQUENCE.
  - If (epoch_id, commit_index) is too old (log compacted past this point): reject with STALE_SEQUENCE.

#### Response

```
message ReconnectAck {
  Status status = 1;
  uint64 epoch_id = 2;         // epoch the notifier is serving
  uint64 next_commit_index = 3; // first index that will be replayed
}
enum Status {
  OK = 0;
  SUBSCRIPTION_NOT_FOUND = 1;
  INVALID_SEQUENCE = 2; // client > notifier
  STALE_SEQUENCE = 3;   // client < notifier, data compacted        }
```

- **OK**: Resumes from last_processed_commit_index + 1.
- **INVALID_SEQUENCE**: Client state is ahead. Recovery: auto-resubscribe (fresh subscription, continuity lost).
- **STALE_SEQUENCE**: Client fell too far behind. Recovery: auto-resubscribe with snapshot at current head.
- **SUBSCRIPTION_NOT_FOUND**: Client must issue a new SubscribeRequest.

### 3. Log Retention Policy

1. The log must be retained at least until the slowest active client's last acked emit_seq and the oldest cold replica's checkpoint.
2. To prevent unbounded growth, define a maximum client stall window (e.g., 24h).
3. Clients reconnecting beyond this window get STALE_SEQUENCE and must resubscribe from snapshot.

### 4. Edge Cases

- **Lost ACKs**: If client processed (epoch_id, j) but ACK was lost, on reconnect it replays (epoch_id, j+1…). Old (epoch_id, j) may be resent; dedupe makes this safe.

- **In-flight data on disconnect**: Any emission sent but unacked before disconnect will be replayed after reconnect.

- **Notifier crash + rollback**: If notifier replays from an older checkpoint, any client "ahead" will trigger INVALID_SEQUENCE and must resubscribe.

### 5. Message Examples

```
// Existing
message SubscribeRequest { ... }
message SubAck {
  string sub_id = 1;
  uint64 epoch_id = 2;
  uint64 commit_index = 3; // first emit_seq
}

message DataEvent {
  string sub_id = 1;
  uint64 epoch_id = 2;
```

```
  uint64 commit_index = 3;
  bytes delta = 4;
}

message ClientAck {
  string sub_id = 1;
  uint64 epoch_id = 2;
  uint64 last_processed_commit_index = 3;
}
```

**Improvements over previous version**

- emit_seq is epoch-aware (handles migration).
- Log retention bounded by a max stall window (prevents compaction deadlock).
- INVALID_SEQUENCE and STALE_SEQUENCE have explicit recovery paths (auto-resubscribe).
- Explicitly documented non-goals (cross-sub atomicity, wire ordering).
- Edge cases (lost ACKs, crash rollback) spelled out so they don't become hidden bugs.

# Epochs, Migration & EmitSeq — Final Spec

**1. Epoch identifier (what epoch_id is)**

An epoch_id must be:

- Globally unique for a particular logical bucket lifetime.
- Monotonic across successive incarnations of that bucket so clients and notifiers can compare "which epoch is newer."

**Representation (recommended):**

epoch_id := {bucket_uuid: UUIDv4, epoch_counter: uint64}

**bucket_uuid** — stable identifier for the logical bucket (doesn't change across migrations that preserve the logical bucket).
**epoch_counter** — monotonically increasing counter assigned by the Cluster Manager when a new epoch is created for that bucket_uuid. Starts at 1.
(If we allow bucket splits/ merges to create new logical buckets, create new bucket_uuid values for the new logical buckets.)

### Why this form?

- **Deterministic comparisons**: first compare bucket_uuid; if equal, compare epoch_counter.
- No dependence on wall-clock time.
- Simple to show ordering and to persist.

## 2. Epoch allocation rules

A single authority (Cluster Manager) is responsible for allocating the next epoch_counter for a bucket and writing a durable EpochCreate / EpochAdvance event into the source bucket log before the epoch becomes active.

Allocation must be deterministic and durable: the EpochCreate entry is committed to the bucket log and is part of the bucket's single source-of-truth. Replicas only accept and act on a new epoch after seeing that committed entry.

**Event (appended to log):**

*EpochCreate { epoch_id, target_group, start_commit_index }*

**start_commit_index** = the commit index in the old epoch after which the new epoch will begin serving (see migration protocols).

### 3. How emit_seq is formed & interpreted

**emit_seq** := (epoch_id, commit_index) where commit_index is the log index within that epoch.

Clients compare emit_seq lexicographically by (bucket_uuid, epoch_counter, commit_index).

When epoch changes, commit_index may reset or start from a chosen base for the new epoch — but since epoch_counter increases, (epoch_id, commit_index) remains totally ordered.

## 4. Migration modes (two supported approaches)

we must pick or support both migration modes. Both begin by creating a new epoch_id via EpochCreate and transferring state, but they differ in the cutover algorithm.

### A — Raft Membership Change (single-log approach)

Best when: we can change raft group membership (like CockroachDB). Simpler to reason about because the bucket keeps one logical log; epoch change is an administrative marker.

**Steps:**

1. Cluster Manager decides migration; calls Raft joint-consensus membership change to move replicas to target group.

2. Once joint consensus settles (new replicas in group), the new group writes an EpochCreate{epoch_id, start_commit_index=current_commit_index} entry as the first entry of the new epoch (or we can write an EpochAdvance marker).

3. The system continues appending events to the same logical log (commit_index continues increasing). epoch_counter increments to denote the new epoch; emit_seq uses the new epoch.

**Option**: start_commit_index can simply be set to the next commit index; clients see a new epoch id but commit indices remain unique because epoch changed.

Old replicas can garbage collect old epoch state once retention rules pass.

**Pros**: Single log continuity; minimal dual-write complexity.
Cons: Requires Raft membership change support and coordination.

### B — Transaction Crabbing / Dual-Write (safe non-blocking migration)

**Best when**: we want to avoid changing raft groups or need zero-downtime migration across groups that cannot be joint-consensed.

**High-level idea**: For a brief window, duplicate incoming client events to both the source epoch log and the target epoch log, in the same deterministic order. When the target is caught up, cutover and stop writing to the source.

### Events used

1. MigrationStart { new_epoch_id, target_group, source_cut_index } appended to source log
2. SnapshotTransfer (out-of-band but logged as SnapshotTransferred in target)
3. MigrationReady { new_epoch_id, ready_at_commit_index } appended to target log
4. MigrationCutover { new_epoch_id, cutover_index } appended to both logs as transaction boundaries
5. MigrationComplete { new_epoch_id } appended to the target log when migration is finalized

## Detailed Steps

**Plan**: Cluster Manager allocates new_epoch_id. Append MigrationStart{new_epoch_id, target_group, source_cut_index = current_commit_index} to source log. This marks the point from which the target will start replaying.

**Snapshot**: Create a non-blocking checkpoint/snapshot at source_cut_index. Transfer to target (via shared storage or RPC). Log SnapshotTransferred{new_epoch_id, snapshot_ref} in target.

**Target bootstrap**: Target installs snapshot; creates a fresh empty log (own epoch) and appends a MigrationReady{new_epoch_id, ready_at_commit_index = source_cut_index} entry once it has applied the snapshot.

**Dual-write phase (crabbing):** For each incoming client event after source_cut_index, the Cluster Manager/coordinator writes the same event to both source and target logs in the same order (coordinator serializes these writes). This ensures both logs contain the identical sequence of events from that point forward.

Both Raft groups independently replicate those events; because the coordinator enforces identical order, deterministic replay yields identical effects.

**Catch-up**: Wait until the target's commit_index ≥ source's commit_index (or until the target has applied up to some pre-agreed cutover_index).

**Cutover**: Append MigrationCutover{new_epoch_id, cutover_index} to both logs (same logical event). Notifiers check for MigrationCutover and atomically switch to the notifier responsible for the new epoch starting at cutover_index + 1. After this point the target's notifier emits and the source's notifier stops emitting.

Tear down: Stop writing to source, let the target be owner. Eventually append MigrationComplete{new_epoch_id} to target log.

**GC**: Source epoch can be compacted after retention guarantees are satisfied and no clients are referencing old emit_seq.

**Pros**: Zero-downtime; target can be in a different failure domain.
Cons: Requires deterministic dual-write coordination (single ordering coordinator), a brief period where two logs must be kept consistent, and more complex implementation.

## 5. Invariants & correctness knobs

During any migration the system must maintain these invariants:

**Monotonic visibility**: Clients must never observe emissions with a lower (epoch_counter, commit_index) after seeing a higher one for the same logical bucket. The ordered tuple must be increasing over time per client.

**Continuity at cutover**: The new epoch must start replaying from cutover_index + 1 where cutover_index equals the last commit index emitted by the old epoch. The cutover must guarantee no-gaps between last emitted old-epoch index and first emitted new-epoch index.

**Deterministic order across logs (dual-write):** When dual-writing, a single deterministic ordering coordinator must serialize events to both logs. Both logs must receive identical sequences (same event payload + same logical order).

Durable epoch creation: EpochCreate / MigrationStart entries must be committed to source log before target begins serving. This avoids races where target believes it's new owner before source acknowledged migration.

**Client reconnection compatibility**: A reconnect request carries (epoch_id, commit_index); the notifier must validate lexicographic position relative to current epoch.

## 6. Client & Notifier behavior during migration (summary)

**Client perspective**: client sees emits with emit_seq possibly switching from (old_epoch, i) to (new_epoch, j) after migration. Clients must treat epoch boundary as usual: compare epoch first, then commit_index. If STALE_SEQUENCE or INVALID_SEQUENCE returned on reconnect, auto-resubscribe fallback is attempted (see previous recontract spec).

**Notifier perspective**: notifier stops emitting old-epoch events after MigrationCutover and starts emitting from target epoch at cutover_index + 1. The notifier must ensure SubAck and ReconnectAck include the epoch context.

## 7. Garbage collection (compaction) rules

**Cannot compact log beyond min_retention_position:**

*min_retention_position = min( min(last_processed_emit_seq across active clients),*
*oldest_checkpointed_position among cold replicas,*
*migration_hold_post_cutover )*

**Maximum stall window**: A configurable bound (e.g., 24h) after which clients that haven't acknowledged are considered abandoned and the system can compact past their last ack, returning STALE_SEQUENCE upon later reconnects.

**Post-migration GC:** Old epoch data can be GC'd once no active client references it and all replicas have checkpoints beyond the cutover index.

## 8. Events / Log entries (summary)

EpochCreate{epoch_id, start_commit_index}

MigrationStart{new_epoch_id, target_group, source_cut_index}

SnapshotTransferred{new_epoch_id, snapshot_ref} (target log)

MigrationReady{new_epoch_id, ready_at_commit_index} (target log)

MigrationCutover{new_epoch_id, cutover_index} (written to both logs)

MigrationComplete{new_epoch_id} (target log)

Make sure each of these is committed to the corresponding Raft log(s) to form durable markers.

### 9. Tests (must-run)

**Epoch ordering test**: create a subscription across migration; assert client sees strictly increasing (epoch_id, commit_index) with no gaps.

**Cutover continuity test**: under load, perform migration; verify last emitted old-epoch commit_index + 1 == first emitted new-epoch commit_index.

**Dual-write deterministic parity**: inject identical randomized writes during dual-write and verify both logs lead to identical state hashes after replay.

**STALE_SEQUENCE test**: compact source log past client's ack and ensure reconnect returns STALE_SEQUENCE and client can resubscribe to snapshot.

**Failover during migration**: kill source/target at different points to verify no double-emits and no lost emits (or deterministic handling per INVALID/STALE statuses).

### 10. Short checklist for implementation

1. Implement EpochCreate allocator in Cluster Manager (durable write to source log).
2. Add epoch_id field to SubAck, DataEvent, ClientAck, Reconnect messages.
3. Implement MigrationStart / SnapshotTransfer / MigrationReady / MigrationCutover events in logs.
4. Implement dual-write coordinator (optional, required for transaction crabbing).
5. Enforce retention and max-stall-window config and code paths for STALE_SEQUENCE.

*Add tests above into deterministic harness.*

### Example scenario (compact)

Cluster Manager writes MigrationStart{new_epoch=(uuidX,2), source_cut=1000} to source log.
Source takes snapshot at 1000; transfers to target. Target installs snapshot, writes MigrationReady{(uuidX,2), ready_at=1000} to its log.

Coordinator dual-writes events 1001..1100 to both logs, ensuring identical order.
Once target commit_index >= 1100, coordinator writes MigrationCutover{(uuidX,2), cutover_index=1100} to both logs. Target notifier begins emitting starting at (uuidX,2):1101.
Source notifier stops at 1100.
Old epoch garbage collected later once retention rules satisfied.

# Immediate work needed to rise above determinism flaws:

**3. Determinism & Correctness**: Needs More Specificity( GEMINI research review)
This is the area that requires the most immediate attention, as highlighted in the plan's "MUST-FIX" list.

**Emission Contract & Reconnect Handshake**: The proposed emit_seq = (epoch_id, commit_index) is a robust mechanism for providing at-least-once delivery with client-side deduplication, achieving "effective-once" semantics. The defined reconnect handshake is logical and covers the main recovery paths. The explicit handling of STALE_SEQUENCE by forcing a resubscribe from a snapshot is the correct, practical choice for clients that fall too far behind.

**Enforcing Determinism (MUST-FIX):** The plan correctly identifies the need for deterministic planners and canonical traversal but lacks concrete enforcement strategies.

**Planner**: The planner must be forced to produce identical plans for identical queries. This can be achieved by disabling cost-based optimizations that rely on volatile statistics and using plan hashing to verify that a query's plan structure does not change unexpectedly. This should be an explicit goal.

**Floating-Point Numbers**: The plan must make a firm decision. Banning floats is too restrictive. The recommended path is to use a fixed-point decimal type for all numerical calculations that require determinism. Non-deterministic floats should only be allowed via explicit opt-in functions (FLOAT_APPROX(...)) that taint the query as non-deterministic.

## A Foolproof Plan for Determinism & Correctness:

Here's how to elevate the existing plan to be more rigorous and resilient against subtle bugs and edge cases.

**3.1 Airtight Delivery Semantics & Formal Specification**
This section moves from a protocol description to a provable contract.

**Notifier Outbox Pattern**: To prevent data loss if a notifier crashes after computing a delta but before sending it, every emission must be transactional.

**Compute Delta**: A transaction begins.

**Write to Outbox**: The (sub_id, emit_seq, delta) is written to a durable, replicated "Notifier Outbox" within the bucket's state.

**Commit**: The transaction commits.

**Send to Client**: The notifier reads from its Outbox and sends the message.

**Receive Ack & Purge**: Upon receiving a ClientAck for a given emit_seq, the corresponding entry is purged from the Outbox. On failover, the new notifier simply resumes sending from its own replicated Outbox.

**Formal Specification (TLA+):** Before implementation, the entire emission and reconnect protocol (including the Notifier Outbox) will be formally specified using a language like TLA+. This allows for exhaustive model checking of all possible states, including crashes, network partitions, and message reordering. This process will mathematically verify that the following invariants are always maintained:

**No Gaps**: A client never misses a message in the sequence.

**No Duplicates (Effective-Once):** A client never processes the same emit_seq twice.

**No Regressions**: A client never receives an emit_seq older than one it has already processed.

**Stale Sequence Recovery**: The STALE_SEQUENCE response is correct. The foolproof addition is to include the current head emit_seq in the reply. This allows the client to intelligently decide if it wants to re-subscribe from a new snapshot or abandon the subscription, rather than blindly retrying.

### 3.2 Provably Deterministic Planning & Execution

This section turns plan determinism from a hopeful goal into a verifiable, consensus-driven fact.

**Plan Hash Committed to Raft Log**: This is the most critical improvement. The query plan is not just an implementation detail; it is part of the replicated state.

When a SUBSCRIBE request is received, the deterministic planner generates a query plan and a canonical hash of that plan.

The SUBSCRIBE event written to the Raft log includes this plan hash.

All replicas (hot and cold) consume this log entry. Before instantiating the subscription, each replica must generate its own plan for the query and verify its hash matches the one in the log.

A hash mismatch is a fatal, consensus-level error. It means a replica has diverged. The replica must halt, report the error, and is not allowed to serve data for that subscription. This prevents any possibility of non-deterministic evaluation.

Deterministic Cost Model & Tie-Breaking: Instead of completely disabling cost-based optimization, we define a simple, deterministic cost model.

**Costs**: Costs are derived from properties known at a specific log index (e.g., table cardinality).

**Tie-Breaking**: If two plans have the same cost, the tie is broken by a strict, arbitrary rule, such as sorting operators lexicographically by name.

**Continuous Integration (CI) Gate**: A CI job will run a suite of queries on multiple architectures (e.g., x86, ARM) and with different compiler flags. The CI gate will fail if the plan hashes or query results ever diverge.

### 3.3 Strict Determinism for Data & Functions

This expands the rules to cover all possible computational loopholes.

Sandboxed, Deterministic UDFs (User-Defined Functions): UDFs are a common source of non-determinism. All UDFs must be executed in a sandbox (e.g., WebAssembly/WASM).

The sandbox environment will provide no access to non-deterministic system calls like network, file I/O, random number generators, or high-precision timers.

The only input to a UDF is its arguments; the only output is its return value.

**Explicitly Banned System Dependencies**: The core engine will be built to avoid dependencies that are known sources of divergence:

**Hash Maps**: Use map implementations that guarantee a fixed iteration order.

**Timezones & Locales**: All timestamps are stored and processed as UTC (TIMESTAMP_TZ). Any timezone conversion or locale-specific string sorting must be done via explicit, deterministic functions whose parameters (e.g., timezone name) are part of the query.

**Regular Expressions**: Use a regex library that guarantees deterministic evaluation times and results.

### 3.4 Resilient State Management & Compaction

This section adds safety valves to prevent misbehaving clients from harming the cluster.

**Maximum Client Stall Window**: A client cannot block log compaction indefinitely. The cluster will have a configurable maximum stall window (e.g., 24 hours).

If a client has not acknowledged a new emit_seq within this window, its subscription is marked as "stale" and its position is no longer considered for log retention calculations.

If the client later reconnects, it will receive a STALE_SEQUENCE error and must re-subscribe. This prevents a single dead or buggy client from causing unbounded log growth.

**Log-Indexed Snapshots**: Snapshots are not just backups; they are a core part of the deterministic state.

Every snapshot is uniquely identified by the emit_seq at which it was taken.

When a client re-subscribes after a STALE_SEQUENCE, the SUB_ACK will specify the emit_seq of the snapshot it is receiving, ensuring the client knows exactly where in the global timeline its new state begins.

**Configuration as Code in Log**: To prevent divergence from operational errors, all critical configuration changes (e.g., changing a bucket's replication mode from cold to hot, changing retention policies) are themselves events that are written to and replicated via the Raft log. The system deterministically reconfigures itself only after committing that log entry.

# GOTO Summary:

## The Core Problem We're Solving

we're building a **reactive database**. Clients subscribe to data, and the database pushes updates to them. That's scarier than request/response because:

- we can't let clients miss updates.
- we can't let them see duplicates or "backwards" data.
- Multiple replicas must behave *exactly the same way*; otherwise clients will get conflicting results.

So everything we've added is about **guaranteeing correctness under chaos**: crashes, restarts, network splits, and replication.

---

## The Moving Pieces (Intuitively)

#### 1. emit_seq (epoch_id, commit_index)

- Think of this as the **global timestamp** for each update.
- It's unique and monotonic (always moves forward).
- Epoch part = "which shard/log I belong to." Commit index part = "my position in that log."
- Clients use it to dedupe, reorder, and detect gaps.

**Why needed:** Without it, clients couldn't tell "is this update new, old, or duplicate?"

---

#### 2. Notifier Outbox

- Every update we send to a client is first written to a durable, replicated outbox.
- Only after it's logged do we actually send it.
- If the notifier crashes mid-send, the next notifier can just replay the outbox.

**Why needed:** Without it, we risk losing updates during a crash window (compute delta but crash before sending = client starves forever).

---

**3. Reconnect Handshake**

- When a client reconnects, it says: *"last I saw was emit_seq = 123."*
- The server checks:
    - If 123 is still in log ⟶ continue from 124.
    - If too old ⟶ tell the client: "Sorry, we're stale. Start fresh from snapshot."

**Why needed:** Without it, reconnections would either duplicate updates or skip them.

---

**4. Deterministic Planner + Plan Hash**

- When a client subscribes, the system generates a query plan.
- That plan (or its hash) is written into the Raft log.
- All replicas must generate the exact same plan ⟶ same hash. If not, it's a fatal error.

**Why needed:** Without this, two replicas might interpret the same query differently ⟶ one emits A then B, the other emits B then A. That breaks correctness.

---

**5. Numerical Determinism (Fixed-Point + Controlled Floats)**

- Use fixed-point decimals for all important math ⟶ guarantees identical results across hardware.
- If floats are allowed, they're opt-in and explicitly marked as "approximate."

**Why needed:** Floating-point math can diverge between CPU architectures ⟶ replicas disagree on deltas. That's silent poison.

---

**6. Sandboxed UDFs + Deterministic Libraries**

- User-defined functions run in a safe box (like WASM).
- No sneaky calls to rand(), now(), or system-local quirks like "hash iteration order."

**Why needed:** Non-deterministic UDFs are a backdoor for chaos. Sandboxing closes it.

---

**7. Snapshots + STALE_SEQUENCE Recovery**

- System regularly cuts snapshots (full database image at a certain emit_seq).
- If a client falls too far behind and its requested seq is compacted, we give it a snapshot instead of replaying a million old deltas.

**Why needed:** Keeps log storage bounded and gives lagging clients a clean way to catch up.

---

**8. Config in the Log**

- Even operational changes (e.g., retention timeouts, bucket lawet) are events written into the log.
- That way, the whole system reconfigures deterministically.

**Why needed:** Prevents human ops from introducing divergence by flipping configs differently across nodes.

---

## Putting It Together: The Intuitive Flow

1. Client subscribes → planner makes a deterministic plan → plan hash logged → all replicas agree.
2. Updates happen → each update gets a global `emit_seq` → written to outbox → pushed to client.
3. Client ACKs → outbox entry purged.
4. If client disconnects, reconnect handshake uses `emit_seq` to resume safely.
5. If client is too stale, system hands them a snapshot at a known .
6. Every replica processes queries, math, and functions deterministically, guaranteed by design choices.
7. Even system config changes go through the same log → everyone stays in sync.

---

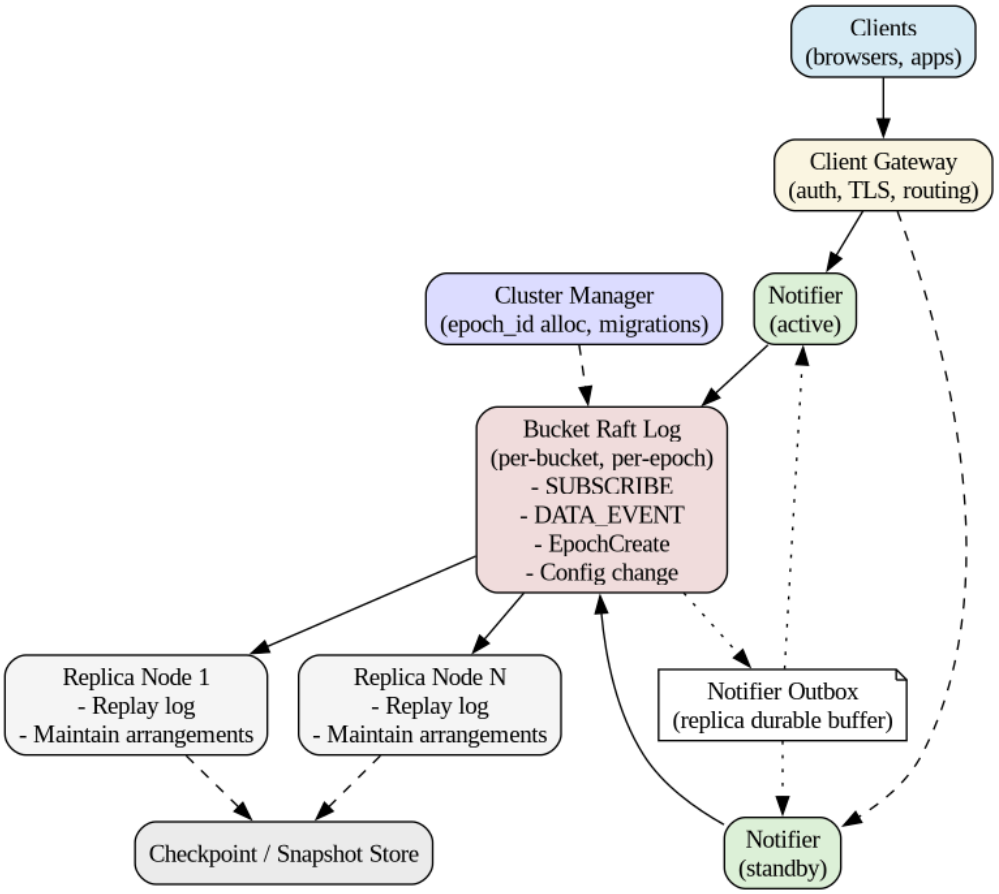
## The Utility of All This

Every piece we added is a brick in the same wall:

- **emit_seq** → lets clients know *where* they are.
- **outbox** → ensures nothing is lost.
- **handshake** → ensures resuming is exact.

- **plan hash** → ensures replicas agree.
- **fixed-point + sandbox** → ensures computations agree.
- **snapshots** → ensures storage stays bounded and recovery is fast.
- **config-in-log** → ensures ops don't cause drift.

Together, they make wer system *provably correct* under replication, failure, and recovery.



# Version V1(Prototype)

### What gets deferred / removed from v1

- Dual-write (transaction-crabbing) migration — **defer** (pick Raft membership approach later).
- Hot vs Cold modes both in full — implement **hot mode only** for instant failover; cold mode later.
- Autoscaling, ML-based backpressure, approximate subscriptions, semantic dropping.
- Sandboxed WASM UDF infra (allow only builtin deterministic functions in v1).
- Full planner CI across multiple CPU archs (do architecture test stubs only).
- Production-grade security/tenant quotas (add minimal TLS/auth stubs).

---

# Minimal v1 feature set (must-have)

1. **Deterministic simulation harness** (Phase 0): deterministic scheduler, controlled time, inject partitions and reordered delivery. (non-negotiable)

2. **Single-node reactive engine** (Phase 1): incremental arrangements, SUBSCRIBE/UNSUBSCRIBE events, local WAL + snapshot.

3. **Replicated buckets (Raft per bucket)** (Phase 2): RF=3 group for a bucket, log contains DATA/SUB/UNSUB/plan-hash entries.

4. **Deterministic planner + plan-hash-in-log**: planner outputs canonical plan; its hash is written with SUBSCRIBE event; replicas verify.

5. **emit_seq = (epoch_id, commit_index)**: epoch creation mechanism (simple epoch_counter allocated by Cluster Manager stub).

6. **Notifier Outbox** (durable, replicated state): write emission to outbox in same log before sending.

7. **Reconnect handshake & reconnect semantics**: ReconnectRequest/ReconnAck; STALE/INVALID/OK handling.

8. **Basic backpressure (BP-A)**: per-sub bounded queue, window/ack flow control, simple coalescing (latest wins for same key).

9. **Core tests in harness**: determinism, subscribe/unsub races, failover with notifier crash, reconnect with stale/invalid sequences.

10. **Minimal metrics & traces**: emit_seq in logs, basic per-sub ACK latency metric (so tests can assert invariants).

---

# Lean milestone breakdown (concrete MVP, testable steps)

## Milestone 0 — Deterministic harness (must pass)

- Implement harness that can: control time, reorder messages, simulate node crashes, and replay logs deterministically.
- Test: replay same input → identical outputs across runs.
- Success signal: harness determinism tests pass.

## Milestone 1 — Local engine + planner

- Engine: in-memory arrangements, incremental update engine, local WAL, snapshot/load.
- Planner: deterministic (simple rule-based planner to start), canonical serialization + plan-hash.
- SUBSCRIBE writes plan-hash to WAL; engine instantiates subscription from WAL on recovery.
- Test: restart node → subscriptions resume with same outputs.

## Milestone 2 — Single-bucket Raft (RF=3) + replication

- Integrate a minimal Raft library (embedded) per bucket; log entries: DATA, SUBSCRIBE(plan_hash), UNSUBSCRIBE, EpochCreate.
- Ensure commit_index is defined and deterministic.
- Test: one replica crash → remaining replica continues; new replica joins and catches up producing byte-identical outputs.

## Milestone 3 — Notifier + Notifier Outbox + emit_seq

- Implement notifier that:
    - Reads committed events, computes deltas deterministically.
    - Writes (sub_id, emit_seq, delta) to replicated Outbox (log-backed or stored in durable state tied to log index).
    - Sends emissions from Outbox; waits for ClientAck to purge.

- Implement emit_seq formatting and include epoch_id in SubAck.
- Test: notifier crash after computing delta but before send → after failover new notifier replays outbox and client sees no loss.

## Milestone 4 — Reconnect handshake + STALE/INVALID behavior

- Implement ReconnectRequest/ReconnectAck. Define log retention/min_stall window config.
- On reconnect: validate (epoch_id, commit_index), return OK/INVALID/STALE. On STALE return snapshot head info.

- Test: client reconnect scenarios (normal resume, lost ACKs, too-old reconnection → STALE → resubscribe snapshot).

## Milestone 5 — Backpressure essentials (BP-A)

- Per-sub bounded FIFO queue + window/ack modes; simple coalescing for same key in queue.
- Implement server behavior under overflow (drop best-effort first).
- Compose harness tests: slow clients with window=1 can't OOM server; coalescing reduces queue under hot key churn.

## Milestone 6 — Determinism hardening & plan-hash CI stub

- Add plan-hash verification across replicas at SUBSCRIBE time; mismatch halts replica (fatal error).
- Add deterministic numeric rule: use fixed-point decimal type in engine core. Provide opt-in float_approx that marks subscription non-deterministic (but block such subs in v1).
- Run deterministic harness tests that include random scheduling and different ordering; assert identical results.

### Milestone 7 — Demo + reproducible benchmark + paper sketch

- Reproduce a neat benchmark: N subs, steady updates, failover test; show correctness (no gaps/duplicates) + basic performance stats (events/sec, commit→emit latency).
- Produce short writeup (2–4 pages) with formal sketch of emission contract, key invariants, and results.

---

## Concrete acceptance criteria (what must be true to call v1 “done”)

- Determinism: two replicas replay identical committed logs (byte-for-byte outputs) in harness across 1000 randomized trials.
- Recovery: kill notifier mid-send and verify client receives every delta exactly once (dedup via emit_seq).
- Reconnect: client that reconnects with last_seen_emit_seq within retention resumes seamlessly; reconnect beyond retention receives STALE and can resubscribe to snapshot.
- Backpressure: slow window=1 client cannot cause OOM; under hot-key churn, coalescing reduces queue length by measurable % vs no-coalesce.
- Plan-hash invariant: any plan-hash mismatch triggers fatal replica quarantine (no silent divergence).

---

# Minimal tech stack & concrete choices (keeps dev friction low)

- Raft: use an existing, well-tested embedded library (for Go: etcd/raft). Keep integration minimal.

- WAL & snapshots: use file-backed append-only log with deterministic serialization (protobuf with canonical serialization).

- Planner: start with a tiny deterministic planner (rule-based operator tree) — no cost-based heuristics. Produce canonical JSON for plan-hash.
- Numerics: fixed-point decimal library (Go: shopspring/decimal).
- Tests & harness: use the same runtime with deterministic scheduler (single-threaded simulated time) — run harness tests in CI.
- Client protocol: protobuf/gRPC for messages (ReconnectRequest/ReconnAck/DataEvent/ClientAck). Keep wire unstable-order tolerant (clients reorder by emit_seq).

---

# Quick risk table + mitigations

- Risk: subtle nondeterminism (hash maps, iteration order).
  Mitigation: use ordered containers only; canonicalize all maps before serialization; CI harness with randomized shuffles.
- Risk: notifier outbox complexity (ordering vs durability).
  Mitigation: implement outbox as log entries (no separate store) so outbox state is replicated by Raft. Simpler and robust.
- Risk: log retention blow-up due to a hung client.
  Mitigation: set strict max_stall_window (configurable) and return STALE beyond it. Instrument metrics and alerts.
- Risk: scope creep (trying to implement both migration modes).
  Mitigation: pick Raft membership change migration for v1; implement dual-write later if needed.

---

## Minimal API sketch (compact) — include in README/proto

- *Subscribe(SubscribeRequest) -> SubAck{ sub_id, epoch_id, commit_index }*
- *DataEvent{ sub_id, epoch_id, commit_index, delta } (emitted by notifier)*

- *ClientAck{ sub_id, epoch_id, last_processed_commit_index }*
- *ReconnectRequest{ sub_id, epoch_id, last_processed_commit_index } -> ReconnectAck{ status, epoch_id, next_commit_index }*

(Implement these as protobuf messages with canonical serialization.)

# Concrete Design Decisions for smooth engineering

## Top-priority (must lock before heavy coding)

1. **Decide one deterministic numeric policy (now).**
    - Pick **fixed-point/decimal** as the canonical in-engine numeric type for v1.
    - Make FLOAT opaque/forbidden in planner v1; allow an explicit FLOAT_APPROX() opt-in that marks a sub non-deterministic (blocked in v1).
    - Implement unit test matrix across architectures for arithmetic determinism.

2. **Pick one migration mode for v1 — *Raft membership change*** (simpler).
    - Defer dual-write (transaction crabbing) to a research extension.
    - Update roadmap/milestones and remove dual-write tasks from v1 to cut complexity.

3. **Make the plan-hash enforcement operational.**
    - SUBSCRIBE must include canonical plan JSON + plan_hash in the log.
    - Replica on startup/subscribe verifies its planner generates same plan_hash; if mismatch → quarantine replica (fail fast).
    - Add CI job: run planner on a canonical query corpus, compare hashes across multiple seeds/processes.

4. **Finalize emission semantics: explicit "effective-once" API and client contract.**
    - Declare in README/API: at-least-once delivery with client-side dedupe using `emit_seq`.
    - Add **Notifier Outbox as log entries** (not a separate store) — implement as `OUTBOX_WRITE(commit_index, sub_id, delta)` so replay is simple.

## High-impact engineering choices (reduce future bugs)

5. **Log retention and max-stall window — explicit config + metric enforcement.**
    - Default `max_stall_window = 24h` (configurable).
    - Expose metrics: `min_acked_emit_seq`, `oldest_checkpoint_index`, `current_compaction_safe_index`.
    - Garbage-collection action plan for STALE clients (automated marking → compact).

6. **Make the reconnect handshake exact and testable.**
    - Add `ReconnectRequest` → include `last_seen_emit_seq` and optional client snapshot version.
    - ReconnectAck must include: `current_head_emit_seq`, `next_commit_index`, `snapshot_ref_if_stale`.
    - Add clear client-side pseudocode in README for reconnect flows (resume / resubscribe).

7. **Backpressure: ship minimal BP-A and test it aggressively.**
    - Implement per-sub bounded queue, ack/window modes, latest-wins coalescing for same key.
    - Add chaos tests: 1 slow client with `window=1` and 1000 fast clients; assert isolation metrics.

## Determinism & verification (formalize)

8. **Plan-hash CI and cross-arch smoke tests.**
    - Run planner on x86 and ARM build runners (or at least different compiler flags). Fail CI if plan_hash changes.
    - Add small suite of canonical queries as "golden plans".

9. **Formal spec sketch (short) + TLA+/PlusCal stub for emission+reconnect.**
    - Not full model-checking now — a 2–4 page TLA+ sketch covering outbox, reconnect, and migration invariants is high ROI for reviewers and for wer mental model.

10. **Deterministic UDF policy.**
    - For v1: only builtin deterministic functions allowed. UDFs disabled.
    - Future: WASM sandbox + deterministic syscall surface.

## Performance / operational decisions

11. **Outbox performance plan.**

    - Since Outbox is log-backed, prepare microbenchmarks: write amplification, commit latency impact.
    - Design: batch outbox writes with data events when possible (coalesce an emit entry with the data event log entry) to reduce RAFT writes.

12. **Metrics & tracing (make observability first-class).**

    - Required metrics: `emit_latency` (commit→emit), `ack_lag_per_sub`, `notifier_cpu, egress_bytes, per-bucket hotness`.
    - Add `trace_id` to log entries to correlate client ACK flow with commit events.

13. **Define SLOs & failure-mode matrix (simple).**

    - e.g., hot-mode failover ≤ 50 ms, cold-mode recovery ≤ X seconds per MB of state.
    - For each failure (notifier crash, network partition, slow client) write expected invariant and acceptable outcome.

## Tests & harness expansions (practical)

14. **Extend simulation harness with these test cases** (must-run):

    - Plan-hash mismatch injection (replica should halt).
    - Lost-ACK recovery: notifier crash after commit before send.
    - Migration cutover under load (single migration scenario — Raft membership).
    - Determinism cross-run: feed same randomized input → byte-identical outputs 1000×.

15. **Add small fuzzing for nondeterminism sources.**

    - Randomize map iteration orders, scheduler interleavings, and assert outputs match canonical runs.

## Documentation / productization (low friction wins)

16. **Add a compact "operational playbook" to README** — how to handle STALE, INVALID, replica quarantine, and how to debug plan_hash mismatches. This is gold for early adopters and reviewers.
17. **Shrink the proto doc to a one-page quick-reference** (Subscribe, SubAck, DataEvent, ClientAck, ReconnectRequest/Ack statuses) — copy-paste friendly for clients.

# Nice-to-have (if time permits)

- Dual-write migration research plan (separate RFC).
- WASM UDF prototype in a research branch.
- Approximate subscriptions / semantic dropping experiments.

## Quick prioritized checklist to turn into issues

1. Decide numeric policy + block FLOAT in planner .
2. Choose migration mode = Raft membership.
3. Implement plan-hash in SUBSCRIBE + replica verification.
4. Implement outbox as log entries and a simple commit→send flow.
5. Add reconnect handshake fields + client example code.
6. Add harness tests: lost-ACK + reconnect.
7. Add basic metrics + one dashboard.

# Potential Pitfalls to keep in check

### 1. Numeric determinism is underspecified

- we mention floats may cause divergence and propose banning or using fixed-point, but don't commit.
- Without a hard decision, replicas on different hardware (x86 vs ARM) will diverge silently.
  **Problem**: correctness bug waiting to happen.

---

### 2. Planner determinism enforcement is incomplete

- The plan says "planner must emit identical query plans" but doesn't specify *how enforcement is verified*.
- There's no mechanism in v1 for replicas to cross-check plan output (e.g., plan-hash written to log).
- **Problem**: nondeterminism could creep in unnoticed → silent divergence.

---

### 3. Migration mode ambiguity

- Both Raft membership change and dual-write (transaction crabbing) are described.
- No clear choice of which to implement first. Supporting both is huge complexity.
- **Problem**: scope creep; easy to sink months into migration logic before MVP works.

---

### 4. Emission semantics not crisply declared

- we imply "effective-once" semantics (at-least-once delivery with dedupe), but don't state it formally in the API.
- Clients don't have a one-line contract like: *"we will never miss an update, but we may see duplicates; dedupe by .*"
- **Problem**: ambiguity for client developers.

---

### 5. Notifier outbox is underspecified

- we propose an outbox, but not where it lives. Is it:
    1. Inside Raft log?
    2. Separate replicated store?
- Without deciding, we risk implementing a fragile ephemeral buffer.
- **Problem**: notifier crash between compute and send could still lose deltas.

---

### 6. Reconnect handshake edge cases only partly covered

- we handle STALE and INVALID sequences, but:
    - What happens if a client reconnects mid-migration?
    - What if the client has unacked deltas in-flight during crash?
- **Problem**: reconnect semantics could fork state if not fully pinned down.

---

### 7. Log retention policy needs firm rules

- we mention “max stall window” (24h default) but don’t define:
    - Who enforces it?
    - What exact metrics decide safe compaction?
- **Problem**: risk of log growth exploding if one client never ACKs.