Running a cluster (active-passive HA)

> **Status: built (Track B).** Single-node operation is the default and is byte-identical whether or not this feature exists — a cluster is opt-in via `[cluster].enabled = true` on a server-DB store (PostgreSQL **or** SQL Server). Clustering is the **active-passive** (leader/standby failover) HA model — the supported HA mode. The horizontal **active-active** scale-out path (the graph running concurrently on every node) was **dropped (2026-06-18) and its code removed**; it is not a planned milestone. Design records: the cluster ADRs 0005 / 0006 (the converged data) and ADR 0008 (the observability API below). Code: `pipeline/cluster.py` (PostgreSQL) + `pipeline/cluster_sqlserver.py` (SQL Server).

MessageFoundry provides HA by running **N identical engine processes against ONE shared server database** (PostgreSQL or SQL Server) in an **active-passive** (one leader, the rest warm standbys) model. There is no separate broker: the durable staged queue, the row leases, leader election, and the config/state convergence all live in the shared database. Single-node stays the no-op default; turning on `[cluster]` makes the nodes coordinate so exactly one — the leader — runs the graph and a standby takes over on failure.

Requirements

A clustered deployment **requires** (enforced at config load):

- `[cluster].enabled = true`
- `[store].backend = "postgres"` **or** `"sqlserver"` — SQLite is single-file/single-node; the cluster needs the shared `nodes` table + row leases a server DB provides. Both backends run the same active-passive leadership lease (`pipeline/cluster.py` for Postgres, `pipeline/cluster_sqlserver.py` for SQL Server).
- `[store].pool_size >= 2` — a clustered node drives concurrent background work against the pool (the membership/lease-renewal maintenance loop + the leader reclaim sweep + the per-stage workers), so it needs headroom over the store's working connections (prefer `>= 3`).

Every node points at the **same** server database (same `[store]` server/database/schema) and runs the **same** config dir.

```
# messagefoundry.toml — identical on every node (the DB password comes from MEFOR_STORE_PASSWORD)
[store]
backend   = "postgres"   # or "sqlserver"
server    = "db.internal"
database  = "messagefoundry"
username  = "mefor"
pool_size = 5          # >= 2

[cluster]
enabled = true
# node_id is auto-derived (host:pid:hex, reusing the store's lease owner-id) — pin it only for a
# stable identity across restarts or in tests.
heartbeat_seconds    = 10.0
node_timeout_seconds = 30.0   # a node is "dead" when last_seen is older than this; must be > heart
# How often the leader runs the RECURRING background expired-lease reclaim sweep (the active-passiv
# background lease-reclaim). It does NOT gate failover speed: on promotion the new leader recovers
# prior leader's stranded rows immediately (owner-scoped, lease-blind; #293), so [store].lease_ttl_
# (the per-row lease TTL, default 60s) is the background-sweep ceiling, NOT the failover-recovery c
reclaim_interval_seconds = 30.0
# Leadership lease (active-passive self-fencing). Timing invariant enforced at load:
#   heartbeat_seconds < leader_fence_timeout_seconds < leader_lease_ttl_seconds
leader_lease_ttl_seconds      = 30.0  # a standby acquires leadership only once the lease has expir
leader_fence_timeout_seconds  = 20.0  # a leader that can't renew within this self-fences (split-br
```

Start the same `serve` command on each host/process — e.g.:

```
python -m messagefoundry serve --service-config messagefoundry.toml --config ./config
```

(For a local DEV Postgres, `scripts/dev/postgres.ps1` sets the `MEFOR_STORE_*` connection env and `MEFOR_ALLOW_INSECURE_TLS=1` for a loopback, no-TLS database — DEV convenience only.)

What each node does

MessageFoundry runs **active-passive** (the Corepoint/Rhapsody model): the **leader (primary)** runs the whole message graph; every other node is a **warm standby** that contends for leadership only. The cluster coordinates the parts that must not double-run or interleave:

- **Active-passive graph gating (Workstream A1).** The wired graph — **all** listeners (MLLP/TCP/File/ …) **and** the router/transform/delivery workers — runs **only on the leader**. A standby binds no listeners and runs no workers; it stays warm (membership heartbeat + cache convergence) and brings the graph up the moment it acquires leadership, and tears it down if it loses leadership. The graph supervisor polls leadership on a short interval so a demotion/fence **promptly** stops a node accepting new inbound work and initiating new processing. (The hard guarantee against *concurrent double-processing of a given row* is the **self-fencing leadership lease** + the leader-gated graph: the graph runs only on the leader, and a

partitioned/slow old leader self-fences and lets its leadership lease **expire** before a standby can acquire leadership — so by the time a promoted node acts, the old leader has provably stopped. The store's **row leases** are the additional backstop for the *recurring background* reclaim sweep, which only takes rows whose lease has **expired**.) Clients reconnect to whichever node is currently primary via a **floating VIP / load-balancer health check** (see the deployment doc). On promotion the new leader recovers the prior leader's stranded in-flight rows immediately — an *owner-scoped, lease-blind* on-promotion recovery (it re-pends only rows owned by *another* instance, never its own), so failover delivery resumes at once instead of waiting out the per-row lease TTL (`[store].lease_ttl_seconds` , default 60s — previously the dominant ~60s Postgres failover-recovery delay; #293). This is safe under the self-fencing guarantee above; the *recurring background* sweep stays lease-gated.

- **Leader election (self-fencing lease).** Exactly one node holds the `leader_lease` row and is the **leader**. The leader renews the lease every `heartbeat_seconds` (to `DB_now + leader_lease_ttl_seconds` , measured on the database's own clock, so node clock skew doesn't affect who may hold it); a standby acquires only once that lease has **expired**. A leader that cannot renew within `leader_fence_timeout_seconds` (< the TTL) **self-fences** — it stops acting as leader before the lease can expire and a standby acquire it, so a network-partitioned old leader never double-processes (the split-brain guard). On a clean stop the leader expires its lease so a standby takes over at once.
- **Leader-gated WRITE singletons.** Retention purges and the lease-reclaim sweep run **only on the leader**, so they never double-execute.
- **Leader-gated poll-source intake.** Only the leader polls a **shared** external resource (a watched directory / DB-poll table / remote dir). Under active-passive the standby doesn't run the graph at all, so this is belt-and-suspenders (the poll loop is also internally leader-gated).
- **Per-lane FIFO survives failover.** Because the graph runs on the **leader only**, per-lane FIFO is naturally serialized by that single processor. Across a failover, the ordinary FIFO claim (`claim_next_fifo`) reclaims a crashed/fenced prior leader's **stranded head** — this lane's expired-lease in-flight row, in the same transaction before the head SELECT — so the stranded row blocks the lane and a later row can never deliver ahead of it. (This replaced the dropped active-active per-lane lease mechanism.)
- **Reference / config / transform-state convergence.** The leader materializes each reference set from its source and followers read-through the shared snapshot; an operator config reload on one node bumps a shared version token and every other node reloads its own config dir to converge; transform-state writes propagate the same way via a per-namespace version token.

Observability — `/cluster/status` and `/cluster/nodes`

Two read-only endpoints on the engine API expose membership and leadership. Both require `Permission.MONITORING_READ` (held by VIEWER and up — no PHI, no new permission) and are reachable via the console or any API client. They cost a cheap in-memory read (`/cluster/status`) or a single `nodes` -table read (`/cluster/nodes`).

`GET /cluster/status` — this node's posture

```
{
  "node_id": "node-a:4812:1f9c2a7b",
  "clustered": true,
  "is_leader": false,
  "role": "standby",
  "config_version": 7
}
```

`role` is the active-passive role for operators / a load-balancer health check: `"primary"` when this node is the leader (it runs the graph), `"standby"` when it is a warm follower (no listeners bound, no workers running), or `"single-node"` when not clustered. Single-node (no cluster) reports `clustered: false`, `is_leader: true`, `role: "single-node"`, `config_version: 0`:

```
{ "node_id": "host:1234:ab12cd34", "clustered": false, "is_leader": true,
  "role": "single-node", "config_version": 0 }
```

`GET /cluster/nodes` — all nodes + the derived leader

`leader_node_id` is the single **live** leader; a crashed ex-leader whose row still carries the leader flag is filtered out by a freshness check (`last_seen` within `node_timeout_seconds`), so it is never reported as the leader.

Two-node cluster:

```
{
  "nodes": [
    { "node_id": "node-a:4812:1f9c2a7b", "host": "node-a", "pid": 4812,
      "status": "active", "started_at": 1750000000.0, "last_seen": 1750000123.4, "is_leader": true
    { "node_id": "node-b:5210:7c3e9d10", "host": "node-b", "pid": 5210,
      "status": "active", "started_at": 1750000005.0, "last_seen": 1750000124.1, "is_leader": false
  ],
  "leader_node_id": "node-a:4812:1f9c2a7b",
  "lease_owner": "node-a:4812:1f9c2a7b",
  "lease_expires_at": 1750000153.4
}
```

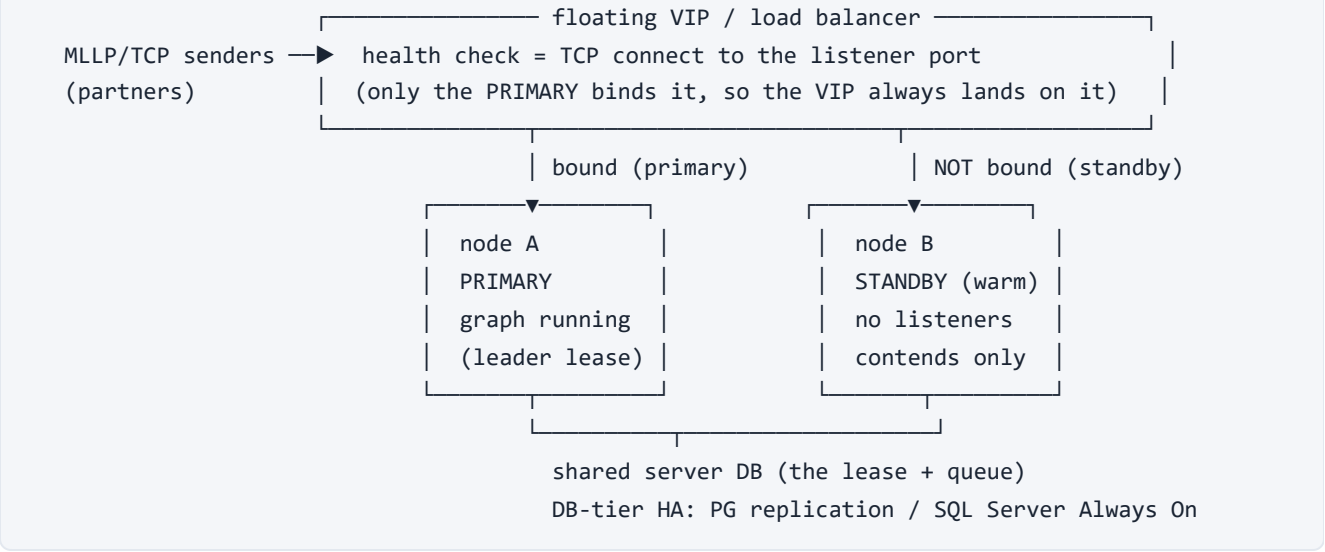
`lease_owner` / `lease_expires_at` are the **authoritative** leadership-lease state read from the `leader_lease` row: who holds the self-fencing lease and the DB-clock epoch at which it expires (the instant a standby could acquire if the leader stops renewing). `lease_owner` normally equals `leader_node_id` (the heartbeat-flag-derived leader); a brief divergence during failover is expected — the lease is the source of truth for who may process. Single node (synthetic self-entry — no heartbeat history, so `started_at` / `last_seen` are `null`; permanently leader, so `lease_expires_at` is `null`):

```
{
  "nodes": [
    { "node_id": "host:1234:ab12cd34", "host": "host", "pid": 1234,
      "status": "active", "started_at": null, "last_seen": null, "is_leader": true }
  ],
  "leader_node_id": "host:1234:ab12cd34",
  "lease_owner": "host:1234:ab12cd34",
  "lease_expires_at": null
}
```

A cleanly stopped node leaves a `status: "left"` tombstone (and its leader flag cleared); a crashed node's row goes stale (its `last_seen` stops advancing) and the freshness filter stops counting it as the leader. `leader_node_id` is always **at most one** node — during a failover window (an old leader's flag not yet cleared while the new leader's flag is already set) the freshest still-beating node wins, so the array never shows two leaders and never names a dead node.

`/cluster/status` is the **per-node authoritative** leadership signal (it reads that node's own in-memory lock gate); `/cluster/nodes` derives leadership from the heartbeat flag and so can lag it by up to one `heartbeat_seconds` interval. So immediately after a clean failover the freshly-promoted node can report `is_leader: true` on `/cluster/status` for one beat before `/cluster/nodes` folds its flag in and surfaces it as `leader_node_id` — a transient `leader_node_id: null` there is the one-tick fold-in lag, not a lost-leader incident.

Deployment topology (active-passive)



One primary processes; the rest are warm standbys. All nodes point at the **same** server DB and run the **same** config dir; the `leader_lease` row elects exactly one primary, which alone binds listeners and runs workers (A standby binds nothing). DB-tier high availability (a replica / failover) is **delegated to the**

database (PostgreSQL streaming replication, SQL Server Always On) — MessageFoundry does not replicate the store itself.

Client reconnect — a floating VIP / LB health check is REQUIRED

Like Rhapsody/Corepoint, clients reach "the engine" through a **floating VIP or load balancer**, not a fixed node — so a failover is transparent to senders (modulo a reconnect):

- **MLLP / TCP inbound (per listener).** Use a VIP per inbound port whose health check is a **TCP connect to that port**. Because only the **primary** binds the port (the active-passive graph gating), the check passes only on the primary, so the VIP routes inbound traffic to it automatically; on failover the new primary binds the port, the old one's closes, and the VIP follows. MLLP senders see a connection drop and reconnect through the VIP — make partners **reconnect on drop** (standard MLLP client behavior).
- **Engine API edge (console / IDE).** The API is a control/read plane over the shared DB and is up on **every** node, so an API VIP can health-check the unauthenticated `GET /health` (liveness). To pin operations to the primary, read `GET /cluster/status` → `role` (`"primary"` / `"standby"`), or `GET /cluster/nodes` → `leader_node_id` + `lease_owner` (the console surfaces the live primary).

Failover is not instantaneous

There is a promotion window, as in Rhapsody (minutes-class) — quantify it from the Workstream-D failover benchmark, don't assume zero-downtime:

- **Clean stop** (graceful shutdown / planned switchover): the leaving primary **expires its lease**, so a standby acquires on its next heartbeat — failover is prompt (≈ one `heartbeat_seconds`).
- **Crash / partition**: the primary's lease **ages out**, so a standby acquires after up to `leader_lease_ttl_seconds`. A partitioned old primary **self-fences** within `leader_fence_timeout_seconds` (< the TTL), so it stops processing before the standby takes over.
- During the window, in-flight rows are protected by the **row leases** (a standby reclaims only *expired* leases); the new primary runs an owner-scoped recovery **once on promotion** to recover the dead primary's in-flight rows promptly (and the ordinary FIFO claim reclaims a stranded lane head, so order survives). At-least-once delivery + idempotent re-runs mean a row interrupted mid-delivery is re-delivered after its lease expires (so downstream connections must stay idempotent).

Tune the lease timings to your network

The defaults (`heartbeat_seconds=10`, `leader_fence_timeout_seconds=20`, `leader_lease_ttl_seconds=30`) trade a ~30 s crash-failover for ample margin. Lower all three proportionally (keeping `heartbeat < fence < ttl`) for faster failover at the cost of less tolerance for a slow DB / GC pause. Because the **leadership** lease is evaluated on the **database's** clock, node clock skew does not affect who may hold leadership; the **row** leases, however, use node wall-clock — see below.

Operational assumptions (honor these)

1. **Clock sync (NTP).** Row leases are wall-clock — keep node clocks reasonably synced so a lease expiry isn't mistimed across nodes.

2. **Identical config on every node.** Each node loads the graph (Connections / Routers / Handlers) from its **own** config dir; convergence coordinates the reload *version*, not the files. Deploy the same config dir to all nodes.
3. **Coordinated config changes.** Apply a config change as a **coordinated (not rolling) restart**, so nodes don't run divergent graphs across the change window.

Related

- ADR 0008 — the observability API design.
- docs/adr/ — the cluster ADRs and the staged-pipeline / store architecture they build on.
- docs/CONFIGURATION.md — the full `[store]` / `[cluster]` settings catalog.