Multi-Replica Reference Architecture

How to run CrossConnect across several application copies (replicas) at once, so the service stays up and carries more load. We cover the load balancer, the application replicas, the shared state they coordinate through, the single PostgreSQL system of record, and the trap that catches every multi-replica rollout: scheduled jobs that fire once on every node.

Audience: SRE, DevOps, and platform engineers building an HA (high-availability) deployment

Scope: topology, shared-state constraints, scheduled-sweep coordination, failover, and what is and is not safe to run at N copies (N-up)

Prerequisite: Performance & Capacity Planning Guide (size one node first)

Document: reference architecture, 21 June 2026

Contact: contact_us@cybriq.io

0 How to read this document

This reference names the mechanism behind each claim, not a vague adjective. It tells you which state lives on a single replica, which is shared, which background job is safe to run at N copies and which is not, and which config key controls each one. Most CrossConnect deployments run a single application node, and the **Performance & Capacity Planning Guide** sizes that one node well into the thousands of devices. You move to more than one replica for two reasons: to survive a node failure or a rolling upgrade with no downtime, and to serve more requests or operators than one node should carry. The rest of this document is the reference shape for that move.

STATELESS safe at N copies, no coordination needed

SHARED one backing store every replica points at

COORDINATE runs once per node; needs a single-runner gate to be safe at N copies

1 Posture at a glance
2 Reference topology
3 The request path is stateless
4 How a request flows
5 Shared state, component by component
6 The rate limiter is per-replica today
7 Scheduled sweeps: the once-per-node hazard
8 The Batfish single-session constraint
9 Caches and single-flight coalescing
10 Database connection math
11 Rolling upgrades & autoscaling
12 Failure modes & recovery
13 Observability across replicas
14 Pre-go-live checklist

Prerequisite. Size a single node first using the Performance & Capacity Planning Guide. Multi-replica is a high-availability and throughput pattern you add on top of that sizing, not a replacement for it. If one correctly sized node already meets your availability target, you do not need this document.

1 Posture at a glance

CrossConnect ships as a set of containers: one application image, PostgreSQL, and an optional Batfish analysis sidecar. You can run them under Docker Compose, Kubernetes, or a managed-database setup such as Cloud Run plus Cloud SQL. The application tier scales out (add more replicas), with one boundary you have to engineer on purpose. Four facts frame every decision below.

The request path is stateless

Dashboards serve from in-memory rollups; reads and writes go to PostgreSQL. No lasting per-user state is kept between calls, so a request can land on any replica. That is what lets the tier scale out.

One database, shared by all replicas

Every replica points at the same PostgreSQL primary, exactly as in a single-node install. The connection budget is the number that bites at scale: the primary has to hold the sum of every replica's pool at once.

Two pieces of state are not yet shared

The API rate-limit counters and signed-in sessions are held in memory on each replica today. Across N replicas that becomes N separate copies, until you put a shared store in front of them or pin each session to one replica (session affinity).

Background sweeps run once per node

Roughly twenty `@Scheduled` jobs (discovery, drift, purges, report delivery) run on every replica on their own. The build has no leader election today, so N replicas means N runs unless you gate them. This is the main trap when you scale out.

> **The single-node invariant.** The shipped build is engineered and tested as a single application instance: rate limiting, sessions, and scheduled work all assume one node. Running several replicas is supported, but it is a *deployment* you assemble rather than a flag you flip. You add three pieces: a shared counter store, session affinity, and a single-runner gate on the sweeps. This document is the assembly guide.

2 Reference topology

Here is the overall shape. A health-checked load balancer spreads traffic across identical application replicas. Every replica shares one PostgreSQL primary and one rate-limit / session store. The primary streams to an optional read replica that absorbs reporting and AI-retrieval reads, keeping that load off the primary. Configuration analysis runs against a Batfish sidecar reached on a separate path (out of band). Every edge below is labeled with its transport.

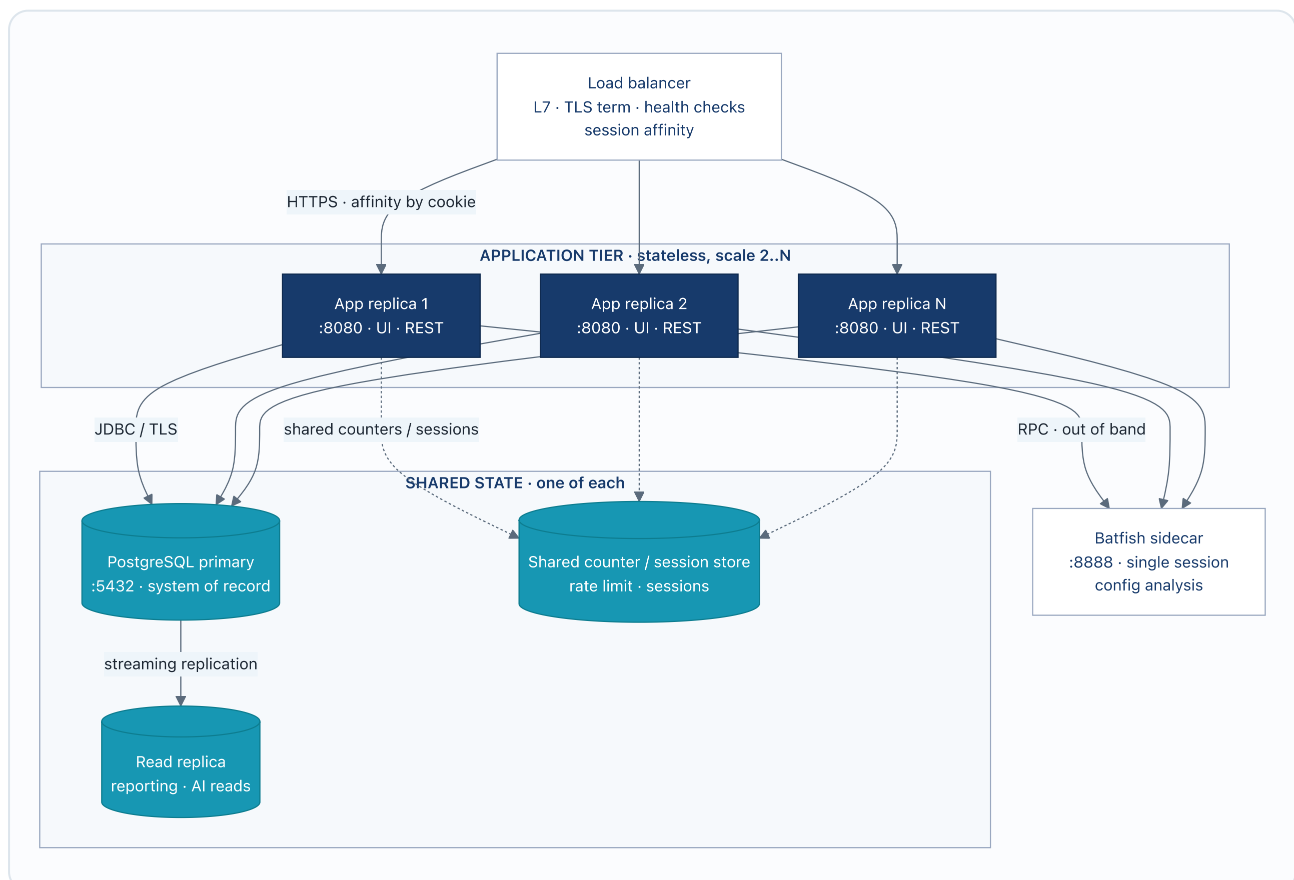


Figure 1. Reference shape. The application tier is stateless and scales from 2 replicas up to N. One PostgreSQL primary is the system of record for every replica; an optional read replica takes the heavy reads. The shared counter / session store and the single-session Batfish sidecar are the two pieces every replica must reach but no replica owns.

Component	Role	Cardinality	State
Load balancer	L7 entry, TLS termination, health checks, session affinity	1 (HA pair)	None (routing only)
App replica	Serves UI (`:8080`) and REST; identical image, identical config	2 to N	Stateless request path
PostgreSQL primary	System of record: all writes and uncached reads (`:5432`)	1	Durable, authoritative
Shared counter / session store	Holds rate-limit counters for the whole fleet, and optionally sessions; not part of the default install, you add it	1 (HA pair)	Ephemeral, shared
PostgreSQL read replica	Reporting, exports, and AI-retrieval reads	0 to N	Replicated, read-only
Batfish sidecar	Config analysis (drift, reachability, change impact); single shared session	1 host or pool	Snapshot-scoped

The default ports come from the shipped configuration: the application listens on `server.port` (`8080`), PostgreSQL on `5432` , and the Batfish sidecar on `8888` via `CROSSCONNECT_BATFISH_URL` . The managed reference (Cloud Run) sets `containerConcurrency: 50` and autoscales from `minScale: 1` to `maxScale: 5` . It keeps one warm replica so requests do not pay the roughly five-second JVM cold start.

3 The request path is stateless; what is not

CrossConnect serves dashboards from in-memory rollups, and reads and writes PostgreSQL as its system of record. The request path itself keeps no lasting per-user state between calls, so a request can land on any replica. But three pieces of state are not local to a single replica, and most of this architecture exists to handle them correctly.

State	Where it lives today	N-up consequence	What you do
System of record	One PostgreSQL primary, shared by every replica	None; it is already shared in a single-node install	Point all replicas at the same primary; size the connection budget (§10)
API rate-limit budget	PER REPLICA in-memory fixed-window counters (`RateLimitFilter`)	The effective limit multiplies by the replica count	Front with a shared counter store, or accept the multiplied limit (§6)
UI sessions	PER REPLICA in-memory; long-lived by default	A session is valid only on the replica that issued it	Session affinity at the LB, a shared session store, or both (§5)

The configuration-analysis engine (Batfish) is shared too, but it is reached on a separate path and holds no per-request state, so it scales as its own pool rather than as part of the request path (§8). The background sweeps are a fourth concern, covered in §7.

4 How a request flows

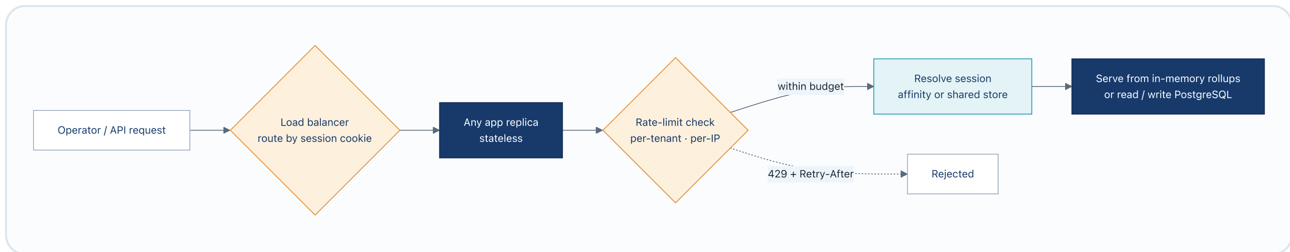


Figure 2. One request. The load balancer uses session affinity to send the request to a replica. That replica checks the rate-limit budget and resolves the session, then serves from its in-memory rollups or reads and writes PostgreSQL. No step depends on which replica was chosen, as long as the budget and the session resolve against shared state rather than memory local to one replica.

When the rate-limit budget and the session both resolve against shared state, the operator sees one consistent limit and one continuous session no matter how the load balancer spreads their requests. Add or remove a replica and none of that changes. When they resolve against replica-local memory instead (the shipped default), the budget and session belong only to whichever replica answered. That is exactly why §5 and §6 exist.

5 Shared state, component by component

Session continuity

Affinity: the load balancer pins each session to one replica by cookie. This is the simplest option, but if that replica is lost, its pinned operators have to log in again. **Shared store:** sessions live off-node, so any replica can serve any session and losing a replica is invisible to users. Affinity plus a shared store gives you both locality and survivability. Sessions are long-lived by default (`close-idle-sessions: false`), so affinity alone strands fewer users than a short timeout would.

Shared rate-limit counter

Point every replica at the same counter store so the budget applies across the whole fleet. The limiter is in-memory per replica in the shipped build. The code is structured so the bucket store can move behind a port, and a shared version plugs in the first time a deployment needs to scale out (§6). Without it, the real limit multiplies by the number of replicas.

One database, sized for the fleet

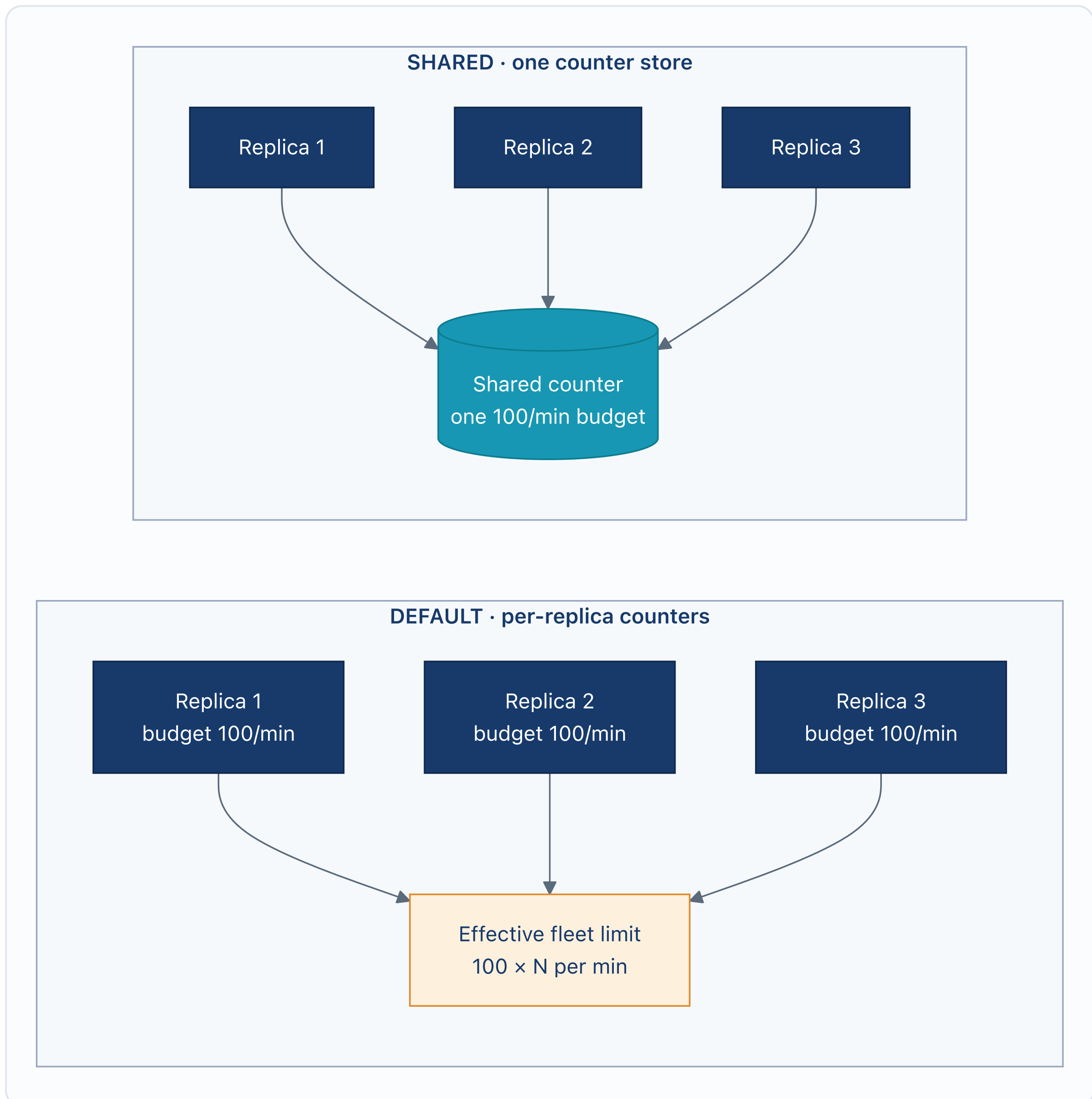
All replicas write to one primary, so they share the connection budget. Size the primary's `max_connections` for the sum of every replica's `SPRING_DATASOURCE_HIKARI_MAXIMUM_POOL_SIZE`, plus room for replication, maintenance, and admin sessions. See the connection math in §10.

Read replica for heavy reads

Reporting, exports, and AI retrieval read a lot of data and can tolerate some delay. Stream them to a read replica to keep that load off the primary's write path. Start without it, and add it once reporting or AI reads begin to compete with interactive traffic. OPTIONAL

6 The rate limiter is per-replica today

This is the shared-state trap most teams hit first. The API rate limiter (`RateLimitFilter`, applied to `/api/v1/*`) is a fixed-window counter kept per (tenant, IP) in an in-memory map on each replica. The default is `crossconnect.ratelimit.requests-per-window: 100` over `crossconnect.ratelimit.window-seconds: 60`, which is 100 requests per minute per key. The source code itself is clear that this is a single-instance design: *"In-memory for v1, single-instance only. When the first deploy needs horizontal scale, the bucket store moves behind a port and a Redis impl plugs in."* The shared-counter implementation is a defined extension point, not yet shipped.



`Figure 3.` **Per-replica vs shared counters.** Left: each replica enforces its own budget, so a client whose requests get spread across replicas can burst up to 100 × N requests per minute. Right: one shared counter restores a single budget for the whole fleet. The same per-replica pattern applies to the occupancy API limiter (`crossconnect.occupancy.api.rate-per-minute`, default 120 per tenant).

Choose one before go-live. Either put a shared counter store in front of the limiter so the 100/min budget covers the whole fleet, or accept that the real limit is 100 × N and set `requests-per-window` to one replica's share of the total you want. Per-tenant overrides (`crossconnect.ratelimit.overrides`) apply per replica in the same way. Do not leave the default unexamined at N replicas: the protection quietly weakens as you scale out.

7 Scheduled sweeps: the once-per-node hazard

CrossConnect runs roughly twenty `@Scheduled` background jobs: discovery, reachability probing, golden-config drift, report delivery, retention purges, and AI-intent expiry among them. Each one is a Spring scheduled method that fires on a `fixedRate` or `fixedDelay` timer. **The shipped build has no leader election, ShedLock, or PostgreSQL advisory lock.** On a single node that is correct and simple. At N replicas, every timer fires on every replica, so each job runs N times per interval. This is the single most important thing to engineer before you go multi-replica.

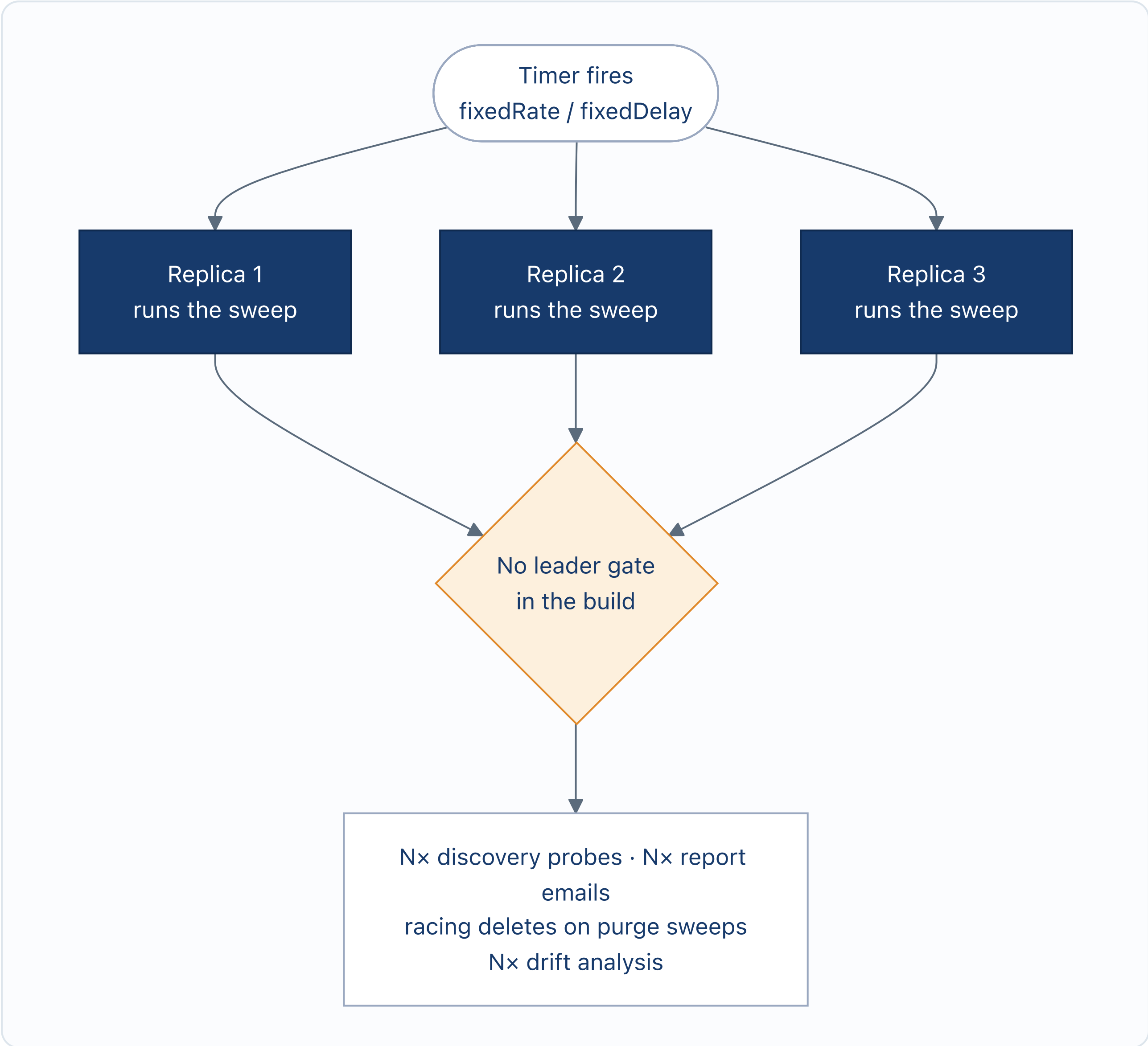


Figure 4. Why a sweep needs a single-runner gate. One timer, N replicas, and no coordination in the build means the same job runs N times. Discovery probes the fleet N times over (a probing storm), report subscriptions send N copies, and the daily purge sweeps race each other to delete the same rows. The fix is a single-runner gate (a distributed lock or a dedicated worker role) that you add at deployment.

Sweep	Default cadence	N-up effect if ungated
Discovery worker	`discovery.interval-ms` 5 min (off by default)	N-fold probing of every device; redundant load on the network
Reachability collector	`health.reachability.interval-ms` 2 min	N-fold ICMP/TCP probes; N observations per device per tick
Golden-config drift sweep	`goldenconfig.drift-sweep-fixed-rate-ms` 15 min	Duplicate analysis; N hits on the shared Batfish session
Scheduled report sweep	`reporting.sweep-fixed-rate-ms` 60 s	Duplicate report deliveries (N copies of each email)
AI write-intent sweep	`ai.intent-sweep-fixed-rate-ms` 60 s	Harmless duplication; idempotent expiry of proposals past TTL
Staging / audit / webhook / device purges	24 h each	Replicas race to delete the same rows; redundant, not corrupting
Batfish health probe	`fixedDelay` 60 s	N pings of the sidecar; benign, read-only
Multicast / AV-drift / occupancy sweeps	5 min to 1 h	Duplicate scans and snapshot writes per tick

> **Three ways to gate the sweeps, best first.** (1) **Dedicated worker role:** run the sweeps on exactly one replica. This is a separate deployment of the same image with discovery and timers turned on, while the operator-facing replicas run with them off. Cleanest option, and it adds no new dependency. (2) **Distributed lock:** wrap each sweep in a leader gate (a PostgreSQL advisory lock or a ShedLock table in the database you already run), so only the lock-holder runs a given interval. (3) **Accept harmless duplication:** for jobs that only re-derive or re-delete data that is already bounded (purges, intent expiry, the health probe), N runs are redundant but do no harm. Discovery and report delivery are *not* in that category, so gate those.

Most timers are already held back by configuration that is off by default: discovery (`crossconnect.discovery.enabled: false`) and automation (`crossconnect.automation.enabled: false`) do not run until you enable them. So a worker-role split is often just a matter of turning them on for the worker replica and leaving them off everywhere else.

8 The Batfish single-session constraint

The Batfish sidecar runs a single shared analysis session. If several `/analyze` calls hit one session at once, they line up and thrash a single snapshot, so each call can run past its timeout. CrossConnect handles this inside each replica with two mechanisms in `BatfishConfigBackend`. A fair `ReentrantLock` queues analyze calls so each one finishes quickly and returns real results, rather than falling back to a rough estimate after a timeout. A `SingleFlight` coalescer merges

identical-config requests that arrive at the same time into one computation. Results are cached and addressed by the config's SHA-256 hash in a persistent finding cache, so repeat questions skip the engine entirely.

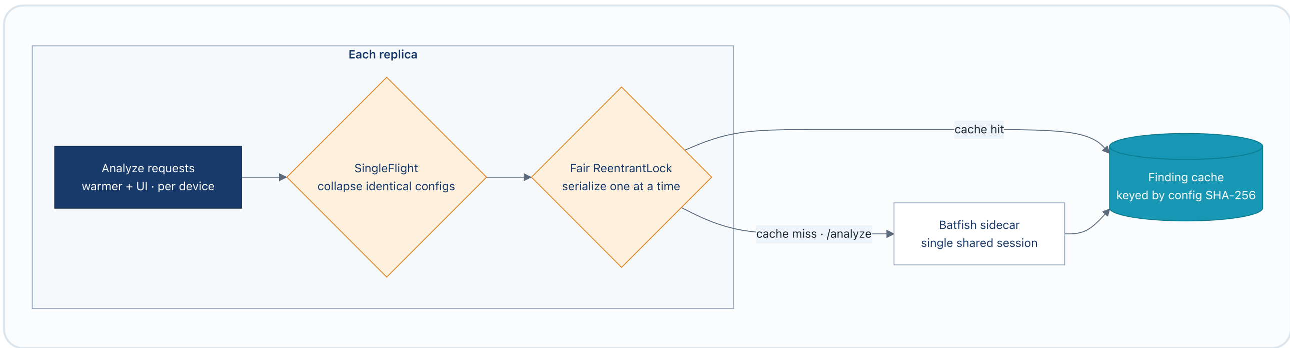


Figure 5. Single-session discipline. Inside a replica, the lock and the single-flight coalescer keep the sidecar's one session from thrashing. The lock is per-replica, so N replicas hold N separate queues. If several replicas call `/analyze` for the same config at once, the sidecar still runs them one at a time, but there is no merging of identical requests across replicas.

> **The multi-replica edge.** The lock and the coalescer live *inside* each replica, so they order that replica's own concurrent calls but do not coordinate across replicas. At N replicas the safe pattern is to keep the heavy analyze callers in one place: run the Batfish warmer and drift sweep on a single worker replica (§7), and let the operator-facing replicas read from the shared hash-addressed cache. Either point all replicas at one sidecar (which then orders the whole fleet's calls on its single session) or give a busy worker role its own sidecar. Watch `crossconnect.batfish.analyze.lockwait` and the `crossconnect.batfish.fallback` counter to spot contention before users feel it.

9 Caches and single-flight coalescing

Every in-memory cache in CrossConnect is per-replica; the build has no distributed cache. That is on purpose and almost always fine, because each cache is either time-bounded (it expires on a TTL) or hash-addressed: two replicas may briefly hold different values, but neither holds a wrong one for long.

Cache	What it holds	Keying / TTL	N-up behaviour
Dashboard rollups (`TtlMemo`)	Network atlas, hotspots, data-quality scorecard	TTL-bounded, single-flight per key	N independent caches; bounded staleness, can disagree for the TTL window
Batfish finding cache	Analysis results	Content-addressed by config SHA-256; no TTL, purged at 30 days	Stable across replicas and restarts for identical config; safe to share on disk
SingleFlight coalescer	In-flight expensive computations	Cleared on completion	Coalesces within a replica only; N replicas may each compute the same miss once

Cache	What it holds	Keying / TTL	N-up behaviour
Discovery run history	Recent discovery runs (UI view)	In-memory deque	Each replica shows only its own runs; expected once sweeps are worker-gated

Because the finding cache keys on a plain hash of the configuration text, it stays stable across replicas and restarts: the same config always produces the same key. So a shared on-disk cache directory lets replicas reuse each other's analysis results without any coordination. The time-bounded dashboard rollups are the only caches where replicas can visibly disagree, and only for the length of the TTL.

10 Database connection math

This is the one number that bites at scale. Each replica opens its own HikariCP connection pool (`SPRING_DATASOURCE_HIKARI_MAXIMUM_POOL_SIZE`), and the primary has to hold all of those pools at once, plus a reserve for replication, maintenance, and admin sessions. Keep each replica's pool to what one node needs (10 to 25, per the Capacity Planning tiers), not what the whole fleet needs.

Replicas	Pool per replica	App connections	Reserve (repl. + admin)	Primary max_connections
2	20	40	20	100
3	20	60	20	100
4	20	80	25	150
6	15	90	30	150
8	15	120	40	200

> Past roughly 6 replicas, keep each replica's pool small and put a connection pooler (PgBouncer in transaction mode) in front of the primary, rather than raising `max_connections` without limit. A few hundred real PostgreSQL connections is a practical ceiling, and the pooler lets many replica pools share them. The Capacity Planning Guide's per-tier `max_connections` (150 at the Standard tier) is the single-node figure; this table is what replaces it once you scale out.

11 Rolling upgrades and autoscaling

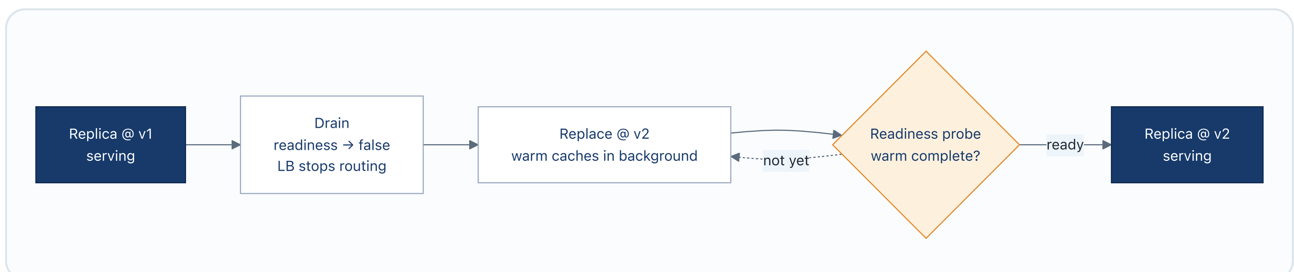


Figure 6. Rolling upgrade. One replica drains (readiness goes false, so the load balancer stops sending it traffic), is replaced, warms its caches in the background, and only then rejoins. Do one replica at a time so capacity never drops below the fleet minus one.

- **Readiness gating.** A replica reports ready only after it has started up and warmed its cache. The shipped `/actuator/health/readiness` probe includes a check that the database is reachable; `/actuator/health/liveness` reports process health. The load balancer and orchestrator both watch readiness, so traffic never reaches a cold or draining replica.
- **Surge policy.** `maxUnavailable: 0, maxSurge: 1` brings a new replica up before retiring an old one, so capacity stays flat through the rollout.
- **Autoscaling.** Scale on CPU, since interactive load is CPU-bound once the cache is warm. A target near 60% CPU leaves room for the warm-up cost of a newly added replica. The Cloud Run reference caps at `maxScale: 5` with `containerConcurrency: 50`.
- **Warm-up cost.** A new replica builds its in-memory rollups on its first traffic, using the startup warmers (`DashboardCacheWarmer` and, where Batfish is reachable, `BatfishWarmer`). Readiness gating hides that from users; time to readiness on restart is roughly nine seconds at the reference workload.

> **Warmers run on every replica.** The startup warmers fire once per node, so each new replica warms its own dashboard rollups, and every replica's `BatfishWarmer` queues against the shared sidecar session at the same time. A guard inside each replica keeps a warm from overlapping itself, but there is no coordination across replicas. That is another reason to concentrate the heavy Batfish callers on a single worker role (§7, §8).

12 Failure modes and recovery

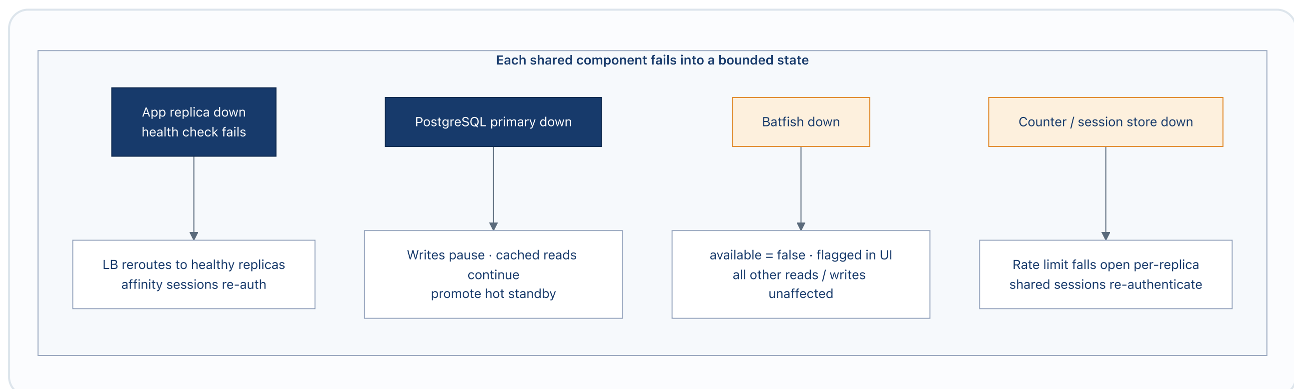


Figure 7. Bounded failure. Each shared component fails into a defined, contained state. None of these failures takes the whole deployment down, and the request path keeps serving reads throughout.

What fails	What happens	What you do
An app replica	Its health check fails, so the load balancer stops routing to it. Affinity-pinned sessions log in again, or continue without interruption if sessions are in a shared store.	The orchestrator replaces it, and capacity restores automatically.
PostgreSQL primary	Writes pause until a standby is promoted. In the meantime, cached reads keep serving from replica memory.	Promote the hot standby and repoint the replicas. Use managed failover where available.

What fails	What happens	What you do
Batfish sidecar	Config analysis reports `available = false` and is flagged in the UI, and analyze calls fall back to a rough estimate (counted on `crossconnect.batfish.fallback`). All other reads and writes are unaffected.	Restart the engine, and analysis resumes. No data is lost.
Counter / session store	Rate limiting falls back to the per-replica in-memory counters so requests are still served; shared sessions fall back to logging in again.	Restore the store (an HA pair makes this rare), and counters and sessions go back to sharing.

13 Observability across replicas

Every replica emits the same metrics it does on a single node, exported at `/actuator/prometheus`, with OpenTelemetry tracing always wired and span export gated by a runtime toggle (`CROSSCONNECT_TRACING_EXPORT_ENABLED`, off by default). The extra work at scale is aggregation: tag each series by replica so you can see both the fleet total and any single replica drifting away from the pack.

- **Per replica:** CPU, heap (`jvm.memory.used` vs `max`), request latency (`http.server.requests` p50/p95/p99), cache hit rate, DB pool in-use (`hikaricp.connections.active` / `pending`), and time to readiness on restart.
- **Fleet aggregate:** total request rate, rate-limit rejections, error rate, and the spread of latency across replicas. A widening spread points to one unhealthy replica before users feel it.
- **Shared dependencies:** PostgreSQL connection count against `max_connections`, replication lag to the read replica, counter/session store availability, and Batfish contention (`crossconnect.batfish.analyze`, `…analyze.lockwait`, `…fallback`).
- **Sweep duplication:** once you are multi-replica, watch for the once-per-node signature: discovery and report volume that scales with the replica count is the tell that a sweep is running ungated.

14 Pre-go-live checklist

Area	Confirm before go-live
Single-node sizing	Each replica is sized per the Capacity Planning Guide for its share of the fleet.
Rate limiter	Either a shared counter fronts `RateLimitFilter` so the fleet limit measures correctly, or `requests-per-window` is set to the per-replica share of the intended total.
Sessions	Affinity and/or a shared session store configured; a replica loss does not strand signed-in operators beyond your tolerance.
Scheduled sweeps	A single-runner gate is in place: a dedicated worker role or a distributed lock. Discovery and report delivery do **not** run on more than one replica.

Area	Confirm before go-live
Batfish	Heavy analyze callers (warmer, drift sweep) concentrated on one worker role; operator replicas read the shared cache; `lockwait` and `fallback` are within tolerance.
Database	`max_connections` covers the sum of replica pools plus reserve; a pooler is in place past ~6 replicas.
Readiness	The readiness probe gates traffic; a draining or cold replica receives none.
Rolling upgrade	Surge policy holds capacity flat; a full rollout completes with no measured downtime.
Failure drills	Replica kill, primary failover, Batfish stop, and counter-store stop each behave as §12 describes.
Observability	Per-replica and aggregate dashboards are live; shared-dependency and sweep-duplication metrics are alerting.

Engage CybrIQ. The shape above covers the great majority of high-availability and throughput needs. For very large or geographically spread deployments, active/active across regions, a built-in shared-counter / leader-election layer, or strict recovery-time and recovery-point objectives (RTO/RPO, how fast you must recover and how much recent data you can afford to lose), contact us at contact_us@cybriq.io and we will design the deployment with you.

CrossConnect by CybrIQ · Multi-Replica Reference Architecture · Technical reference · 21 June 2026 · The single-instance behaviours described (per-replica rate limiting and sessions, once-per-node scheduled sweeps, the single-session Batfish constraint) reflect the shipped operator-preview build; the shared-counter and leader-gate layers are deployment patterns you assemble yourself. · contact_us@cybriq.io