CrossConnect
CONNECT. MONITOR. OPTIMIZE.
BY Cybr IQ

Performance & Capacity Planning

How fast CrossConnect goes, the settings you can turn, and a sizing model for a deployment. This guide shows where the cost actually lives, why supporting many operators at once stays cheap, and how much CPU, memory, and storage to set aside for the size of your fleet, the number of operators, and the length of history you keep.

Audience: platform engineers, SREs, capacity planners sizing a deployment

Scope: the request path, caches, the one place config analysis runs one-at-a-time, the connection pool, the meters to watch, and a tier-by-tier sizing model

Method: behavior taken from the shipping build; the numbers are anchored to a measured reference workload

Document: performance & capacity reference, 21 June 2026

Contact: contact_us@cybriq.io

0 How to read this document

This is a sizing reference, not a marketing sheet. Each section explains what is behind a number: the cache and how long it holds (its TTL), the lock in front of the config engine, the pool default, the meter you watch. The sizing tiers are conservative starting points, anchored to the measured reference workload at the back of this guide. Test at your own scale before going live.

DEFAULT the value the build ships with **TUNABLE** an environment variable you set per deployment

SIZING INPUT a number you supply that drives the model

1 Where the cost lives

CrossConnect is a single application process backed by a PostgreSQL database, plus an optional helper service for config analysis (a Batfish sidecar). Sizing it comes down to finding the few places where work is genuinely expensive and confirming the cheap paths stay cheap. Four facts frame every number here:

Reads dominate, and they are cached

Most operator traffic is reads: loading dashboards and lists. The summaries operators ask for repeatedly are served from a short-lived in-memory cache, so adding more simultaneous operators adds CPU, not database load. That is why the measured workload held flat from 90 to 150 concurrent loads.

The config engine is the one place work lines up

Analysis across the whole fleet runs against a single Batfish session, guarded by a fair lock. It is the only place where simultaneous requests wait in line rather than run side by side, which is why it moves onto its own host first as you grow.

Storage is the long-term driver

CPU and memory are sized once per tier and rarely change. The database volume grows steadily as metric and flow history and config snapshots pile up, so the recurring cost decision is the storage plan, not the compute plan.

Background work is capped and off by default

Discovery, webhook delivery, and the intent sweep run on small fixed-size worker pools on a fixed schedule. They do not grow with operator traffic, and discovery ships turned off, so they never surprise a sizing estimate.

Why many operators stay cheap. Dashboard summaries are cached behind a per-key single-flight lock, so only one request at a time computes a given result. The first reader of a cold entry does the work; every other reader asking for the same thing waits on that one result instead of recomputing it. The measured node used about **0.5 GB of heap at 155 devices** and returned **0 errors and 0 throttled requests** under load. Concurrency is limited by CPU and how often the cache misses, not by the database.

2 The request path & sizing model

The figure below is both the architecture and the sizing model. Each arrow is labelled with what makes that path cheap or expensive, because that is what you are sizing for. Cached summaries served from memory keep many operators cheap to support; writes and uncached lists go to PostgreSQL; analysis across the whole fleet is the one path that runs a single request at a time on a single engine session.

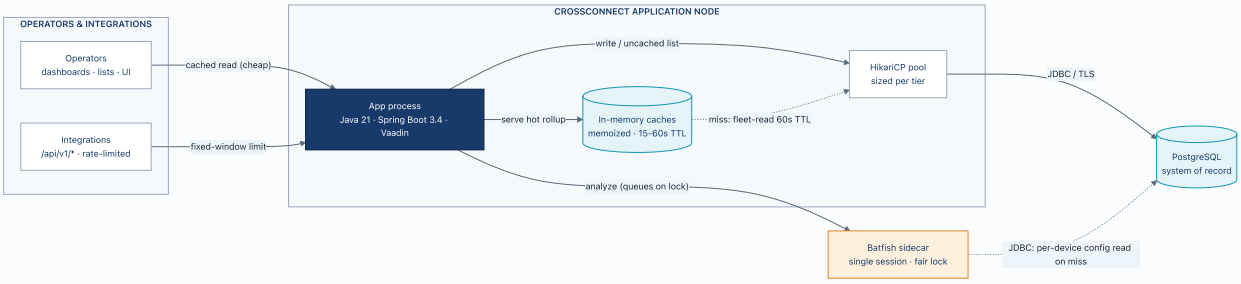

Figure 1. The request path and where to spend. Operator reads are served from memory, so handling more of them at once costs CPU, not database connections. Writes and cache misses cross the HikariCP connection pool to PostgreSQL. Analysis across the whole fleet is the only path that runs a single request at a time, on the one Batfish session behind a fair lock, which is why it is the first component to move onto its own host.

The sizing model follows directly from that path. You size four things, in order: the **application node** (CPU for handling many operators at once, memory for the data it keeps in flight), the **database** (connections matched to the pool, RAM for the active working set, disk for how long you keep history), the **config engine** (memory that grows with the size of the fleet, moved off the app node once it gets heavy), and the **storage volume** (the one number that keeps growing over time). The later sections cover each one in turn.

3 Caches: why concurrency is cheap

A Standard node handled 150 dashboard loads at once with memory near half a gigabyte because the expensive summaries are computed once and then reused. Several caches do this work. The dashboard report memos and both Batfish answer caches use a per-key single-flight lock, so even when many requests for the same cold entry arrive together, the result is computed exactly once, not once per request. The fleet-read cache is a plain short-TTL cache: it holds the rebuilt fleet config map for 60 seconds, which collapses the repeated per-question reads inside a single page load.

Cache	What it holds	TTL	Why it matters for sizing
Fleet-read cache	Per-tenant device running-configs, rebuilt for each Batfish question	60 s	A single page asks 5 or more config questions; without this cache the whole fleet would be read from PostgreSQL 5× per page load
Report memos (`TtlMemo`)	Per-tenant dashboard rollups: data-quality, hotspots, maturity, occupancy, mDNS health, AV/QoS reports	15–30 s	The most-loaded dashboard reads; served from memory between refreshes, so more operators add only CPU
Batfish result cache	Config-analysis answers, keyed by tenant + question + config hash	invalidate-on-change	A warm (already-computed) analysis answers in microseconds; the cost is paid once per config version
Batfish finding cache	Two layers: in-memory plus on-disk (keyed by analyzer version + config hash)	30 d on disk	Survives a process restart, so a rolling upgrade does not have to redo cold config analysis

> **How the caches warm up.** The report memos hold for 15 to 30 seconds and the fleet-read cache for 60 seconds. A freshly started replica answers its first read by computing it once under the single-flight lock; a background warmer fills the popular entries ahead of time, so operators rarely hit the slow cold compute. That is why a rolling upgrade done one replica at a time has no downtime, and why the disk-backed finding cache matters: the expensive config analysis does not have to be redone after a restart.

All report memos are per-tenant. A multi-tenant deployment multiplies cache memory by the number of active tenants, but each entry is a small summary, so the total stays modest next to overall heap.

4 The config engine: the one serialization point

Formal analysis across the whole fleet (reachability, ACL, and IP-address-conflict questions) runs against a single Batfish session. This is the only place in the request path where simultaneous requests wait in line rather than run side by side, so it pays to know how it behaves under load and why it is the first thing to move off the app node.

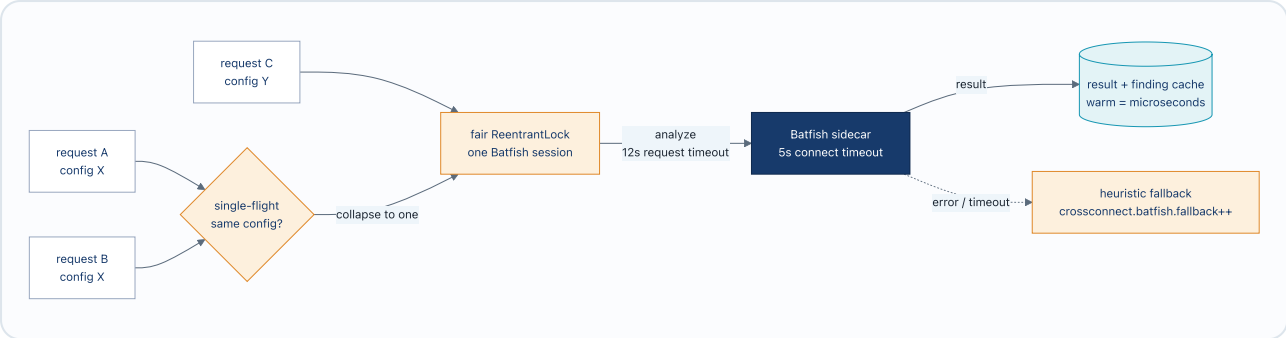

Figure 2. Two layers of merging in front of one session. Simultaneous requests for the same config are merged into a single analyze call (single-flight); requests for different configs take turns on a fair `ReentrantLock` so each finishes in order. A Batfish error or timeout increments `crossconnect.batfish.fallback` and answers from a built-in rule-of-thumb estimate, so the engine degrades gracefully instead of blocking.

- **Single-flight first.** Several first-time requests for the *same* config are merged into one analyze call. Different configs still take turns on the lock.
- **Fair lock second.** A fair `ReentrantLock` lines up distinct analyses so each finishes quickly and the cache fills cleanly. Watch `crossconnect.batfish.analyze.lockwait` to see how deep the queue gets on the shared session.
- **Capped timeouts.** The sidecar HTTP client waits at most 5 seconds to connect and 12 seconds for an analyze request. If the engine is slow or unreachable, the request falls back to a rule-of-thumb answer instead of hanging, and that fallback is counted.
- **The slow first run is paid only once.** The on-disk finding cache (kept 30 days, keyed by analyzer version) means a given config is analysed once across the life of the process; later reads, even after a restart, are served from cache.

> **What this means for sizing.** Because analysis runs one request at a time on one session, config-heavy workloads do not spread across CPU cores the way cached reads do. That is why the tiers move the engine onto its own dedicated host at Large and up: a heavy analysis must not starve the app or the database, and a dedicated host lets the engine's memory grow with the fleet without competing for the app node's memory.

5 The database & the connection pool

PostgreSQL is the single system of record. The application reaches it through a HikariCP connection pool, and the pool size is the main dial between how much the app does at once and how much load lands on the database. The build does not override Hikari, so the framework defaults apply until you set the pool size yourself for each tier.

HikariCP property	Default (unset)	How to set it
maximum-pool-size	10	`SPRING_DATASOURCE_HIKARI_MAXIMUM_POOL_SIZE` (size per tier, §10)
minimum-idle	10 (equals max)	leave at max; the app is steady-state, not bursty
connection-timeout	30 s	framework default; pool waits show up as `hikaricp.connections.pending`

HikariCP property	Default (unset)	How to set it
max-lifetime	30 min	framework default

> **Connection math (do not skip).** Set Postgres `max_connections` to at least **(app pool size × number of app replicas) + 20**, leaving the +20 for admin work and background jobs. With one app replica on the Standard tier (pool 25) that is 25 + 20 = 45, so the Standard recommendation of 150 leaves plenty of room. Setting the pool larger than Postgres can serve does not help; it just moves the bottleneck from the app to the database.

Flyway (version 10.20.1) owns the database schema; Hibernate runs in `validate` mode and never changes the schema on its own. Primary keys use UUIDv7 (time-ordered IDs), which keeps related rows close together in the index on insert. That holds down index bloat and extra write work as history piles up.

6 Discovery, webhooks & background work

Background work runs on small, fixed-size worker pools on a fixed schedule. None of it grows with how many operators are online, and the heaviest job (discovery) ships turned off, so it never inflates a sizing estimate unless you turn it on.

Background job	Cadence / concurrency	Default	Tunable
Discovery sweep	fixed-rate, single-threaded: tenants one at a time, then devices one at a time	5 min interval, 60 s initial delay, **disabled**	`CROSSCONNECT_DISCOVERY_INTERVAL_MS`, `…_ENABLED`
Webhook dispatch	async on a 4-thread worker pool; 6 attempts with exponential backoff (1s→1h cap)	4 worker threads, 3 s per-call timeout	`CROSSCONNECT_WEBHOOKS_WORKER_THREADS`, `…_TIMEOUT_MS`
AI intent sweep	fixed-rate sweep of expired confirm-before-commit intents	60 s rate, 30 s initial delay, 15 min intent TTL	`CROSSCONNECT_AI_INTENT_SWEEP_FIXED_RATE_MS`
Retention purges	scheduled staging / audit / sensor sweeps (chain-aware on the audit trail)	operator-set rolling windows	retention settings

> **Discovery grows with the fleet, not with operators.** A sweep walks tenants one at a time, then devices within a tenant one at a time, with a 20-second SSH read timeout per device. Sweep time grows with the number of devices, not with how many operators are logged in. If a 5-minute interval is not enough time to finish a sweep at your fleet size, widen the interval rather than adding app CPU.

7 The rate limiter

Integration (API) traffic on `/api/v1/*` is rate-limited: each tenant-and-IP pair gets a fixed number of requests per time window. The operator UI is not rate-limited. The default is 100 requests per 60 seconds; raise it to your measured integration rate plus some headroom.

Setting	Default	Purpose
`CROSSCONNECT_RATELIMIT_REQUESTS_PER_WINDOW`	100	requests allowed per window per (tenant, IP)
`CROSSCONNECT_RATELIMIT_WINDOW_SECONDS`	60	window length in seconds
`crossconnect.ratelimit.overrides`	(none)	comma-separated `tenantId=capacity` per-tenant overrides

A rejected request returns HTTP 429 with an RFC-7807 problem-detail body and a `Retry-After` header. Today the counter lives in memory on each replica, so the first time you run more than one replica it has to become shared state behind a common store (see §13).

8 Gather your sizing inputs

Five numbers drive the model. Fill these in first; everything after derives from them.

Input	What to enter	Drives
A · Managed devices SIZING INPUT	Fleet today plus 12-month growth (switches, routers, APs, firewalls)	Tier, RAM, engine heap, storage
B · Peak concurrent operators SIZING INPUT	The busy-hour peak, not total accounts. If you do not know it, use 5–10% of your operator headcount	App CPU, DB pool
C · History retention SIZING INPUT	How many months of metrics, flows, and change history to keep. If unsure, start at 6	Database storage
D · Integration request rate SIZING INPUT	Automation / API calls per minute; 0 if none	Rate-limit window (§7)
E · Availability target SIZING INPUT	Single node, or no-downtime upgrades (multi-replica)	Deployment shape

9 Pick your tier

Find input **A** (devices) in the first column. That row is your tier for the rest of the model. If input **B** (operators) points you to a higher tier than **A** does, use the higher one: more operators at once add CPU, not memory.

Tier	Devices (A)	Peak operators (B)	Deployment shape	Total RAM	Total vCPU
Pilot	up to 500	up to 10	Single node (all-in-one)	8 GB	4
Small	up to 1,000	up to 15	Single node	16 GB	4
Standard	up to 5,000	up to 25	Single node (larger)	32 GB	8
Large	up to 10,000	up to 50	Two nodes (engine split out)	64 GB	16
X-Large	up to 25,000	up to 75	Three+ nodes by role	128 GB	32
Very large	50,000+	100+	Distributed (engage CybrIQ)	256 GB+	64+

> The reference workload handled **150 dashboard loads at once on the Standard tier** with room to spare and zero errors. Operator headroom is rarely the limit; fleet size (memory and engine heap) and retention (storage) usually are.

10 Provision the three components

For your tier, build each piece to these specs. On Pilot, Small, and Standard they all run together on one node; from Large up they split apart. The figure shows how the layout changes as the config engine, the one place work runs one at a time, is pulled off the shared node.

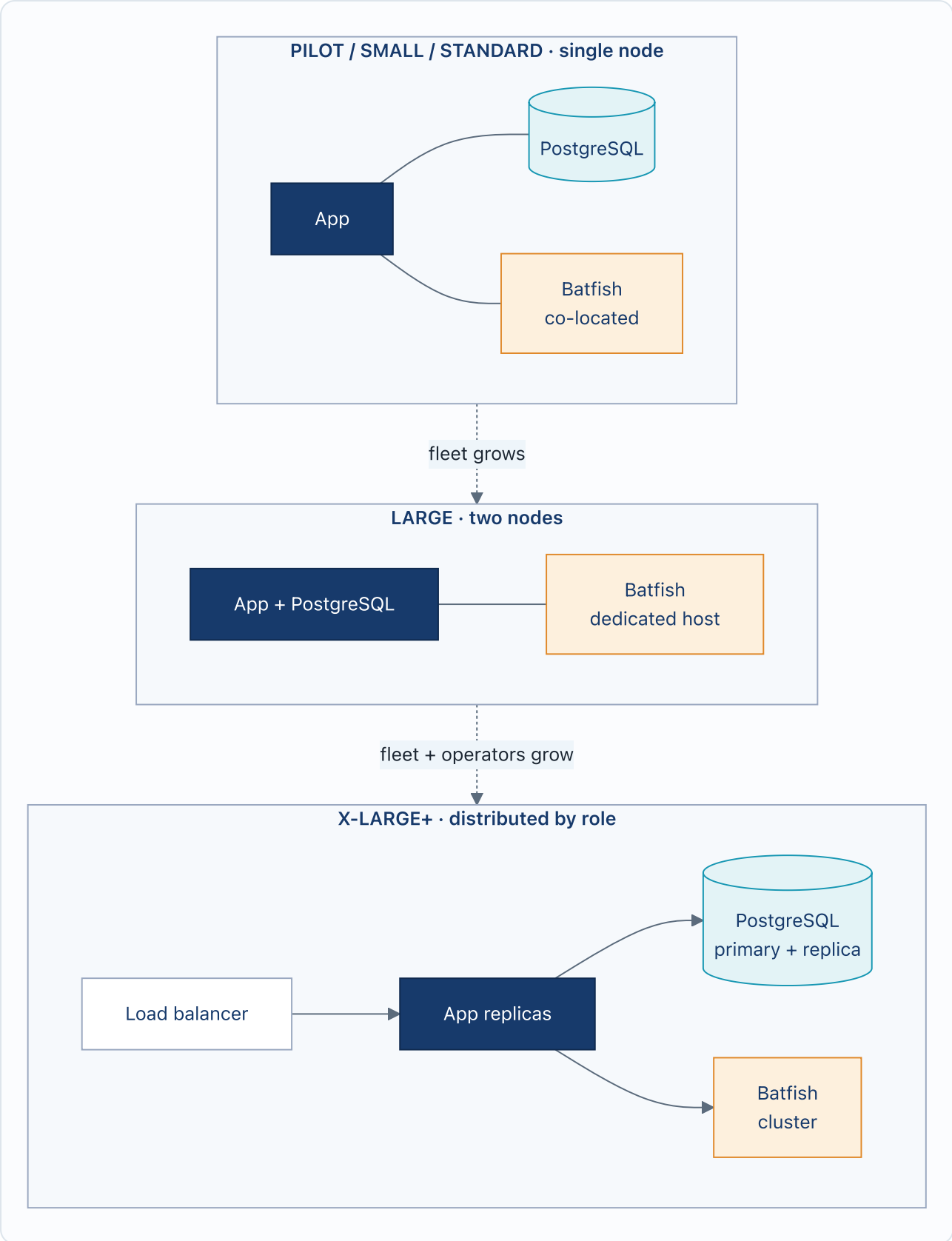

Figure 3. Deployment layout by tier. Components share one node up to Standard. At Large the Batfish engine, the one place work runs one at a time, moves onto its own host so a heavy analysis cannot starve the app or database. At X-Large and up the deployment spreads out: a load balancer, app replicas, a PostgreSQL primary with a replica, and an engine cluster.

10a · Application node

Tier	vCPU	RAM	JVM max heap	DB pool	Notes
Pilot	2	3 GB	`-Xmx2g`	10	Default footprint
Small	2	4 GB	`-Xmx3g`	15	
Standard	4	6 GB	`-Xmx4g`	25	More CPU for concurrent reporting
Large	6	8 GB	`-Xmx6g`	40	Shares a node with the database
X-Large	8	12 GB	`-Xmx8g`	60	Consider 2 replicas behind a load balancer

Leave roughly 30% of node RAM above the heap for off-heap memory, metaspace, and the operating system. The reference node used about 0.5 GB of heap at 155 devices, so these figures are comfortable, not tight.

10b · PostgreSQL (the system of record)

Tier	vCPU	RAM	max_connections	shared_buffers	work_mem
Pilot	2	3 GB	50	1 GB	16 MB
Small	2	6 GB	75	2 GB	24 MB
Standard	4	12 GB	150	4 GB	48 MB
Large	6	16 GB	200	6 GB	64 MB
X-Large	8	32 GB	300	10 GB	96 MB

Connection math from §5: `max_connections` ≥ (pool × replicas) + 20. Use JDBC over TLS (an encrypted connection) for any database link that is not on the same machine.

10c · Batfish config-analysis engine

Tier	Placement	Heap	Why
Pilot / Small	Co-located on the node	2 to 4 GB	Light analysis load
Standard	Co-located, fixed heap	8 GB (fixed)	Snapshots grow with the fleet
Large +	Dedicated host	16 GB+	A heavy analysis must not starve the app or database, and it runs one request at a time on a single session

The engine is optional. If no sidecar is reachable, config-analysis answers fall back to a built-in rule-of-thumb estimate (counted on `crossconnect.batfish.fallback`) instead of failing.

11 Apply the settings

Set these on the application container. The first three are sized from §10; the rest are the same for every production deployment. The full tunables list is in Appendix A.

Setting (environment variable)	Value	Purpose
`JAVA_TOOL_OPTIONS`	`-Xmx<heap>g` (from 10a)	Application heap ceiling
`SPRING_DATASOURCE_HIKARI_MAXIMUM_POOL_SIZE`	DB pool (from 10a)	Concurrent database connections
`CROSSCONNECT_RATELIMIT_REQUESTS_PER_WINDOW`	100 default, or input **D** + headroom	API requests per window per tenant per IP
`CROSSCONNECT_SECURITY_REQUIRE_SECRETS`	`true`	Refuse to start in prod without secrets set
`CROSSCONNECT_AUTH_ADMIN_SECRET`	a strong secret	Locks the admin API
`CROSSCONNECT_AUTH_SIGNING_SECRET`	a strong secret	Stable login tokens and audit-chain signing across restarts
`SPRING_DATASOURCE_URL / USERNAME / PASSWORD`	your database	System-of-record connection

> **Set the heap for the container.** Left alone, the JVM only uses 25% of the container's memory limit, so set the heap on purpose: either `-Xmx<heap>g` from 10a, or `-XX:MaxRAMPercentage=70` with the container memory limit set to the 10a RAM. Keep about 30% of the limit free for off-heap memory, metaspace, and threads. If the limit is too small, the system's out-of-memory killer ends the process; you do not get a tidy heap dump.

Distributed tracing is always built in, but exporting the data is a runtime switch that is off by default, so you do not need a collector to run. When you want traces, turn it on and point `OTEL_EXPORTER_OTLP_ENDPOINT` at your collector (OpenTelemetry exporter 1.43.0).

12 Plan storage & retention

The database storage in §10b is a starting point. It grows mainly with three things you control, and over time storage is the biggest cost driver.

Driver	Effect	Lever
Metric & flow history (input C)	The fastest grower over time	Set a retention window; older samples are removed on the scheduled purge sweep
Change & audit history	Tamper-evident, append-only, hash-chained	A chain-aware purge trims data past the window while keeping the hash chain intact
Config snapshots	Grow with fleet size × how often configs change	Held in check by snapshot retention

Starting estimate (validate against your own data). Budget roughly **25 GB per 1,000 devices per month of retention**; confirm against your own interfaces-per-device and flow volume. For the worked example (~1,800 devices, 9 months) that is 1.8 × 9 × 25 ≈ 400 GB at the full window, which is why the node starts at 250 GB SSD (about the first half-year accumulated) and grows toward ~400 GB. Dense interface counts and heavy flow capture push it up; sparse fleets pull it down.

Size the volume for your full retention window (input **C**), not just today's footprint. Use SSD storage at Standard and below, and faster NVMe storage from Large up.

13 Probes, orchestration & the meters to alert on

The endpoints and settings to wire into Kubernetes, Cloud Run, or your scheduler, plus the meters that warn you when a component is running out of headroom.

Health & metrics endpoints

Endpoint	Use	Notes
`GET /actuator/health/liveness`	Liveness probe	Says the process is alive; fails only on a broken JVM. Do not tie this to dependencies.
`GET /actuator/health/readiness`	Readiness probe	Says it can serve right now, including that the database is reachable. Send traffic only when this passes.
`GET /actuator/metrics/{name}`	Read one meter	Readable with no extra infrastructure; the quick way to spot-check the Batfish meters.
`GET /actuator/prometheus`	Metrics scrape	Prometheus-format metrics to scrape into your time-series database. Optional.

Orchestration (Kubernetes / Cloud Run)

- Set CPU and memory **requests = limits** to the §10a application spec; the app runs at a steady level, not in bursts.
- **Probes:** a `startupProbe` on `/actuator/health/readiness` with about a 30 s budget (boot takes about 9 s), then separate `readinessProbe` and `livenessProbe`. Keeping them separate stops the orchestrator from killing a healthy pod that is still warming up.
- **Rolling upgrades:** `maxUnavailable: 0`, `maxSurge: 1`. A new replica serves right away; its first dashboard read fills the cache under the single-flight memo (§3), the background warmer fills it ahead of time, and the disk-backed finding cache means config analysis does not have to be redone.
- **Autoscaling (HPA):** scale the app tier on CPU at about 70% for workloads heavy on uncached reads or reports. Cached dashboard traffic rarely triggers it. Before running more than one replica, plan for the two shared-state items in the HA note below.

What to alert on

Meter	Healthy	Alert when	Action
`jvm.memory.used` / `max`	< 70%	> 85% sustained	Raise heap / node RAM
`hikaricp.connections.pending`	~0	> 0 sustained	Raise pool + Postgres `max_connections`
`hikaricp.connections.active`	< max	at max sustained	Pool saturated; add connections
`http.server.requests` p95	at baseline	> 2× baseline	Add CPU or a replica
`system.cpu.usage`	< 80%	> 90% sustained	Add CPU / scale out
`crossconnect.batfish.analyze.lockwait`	low	rising sustained	Requests are queueing on the engine session; move the engine onto its own host
`crossconnect.batfish.fallback`	0	> 0	Config engine is degraded; check that Batfish is reachable and has enough heap

> **Scaling out (more than one app replica).** The request path keeps no state of its own, but two things do need shared state: the per-tenant API rate limiter (today it lives in memory on each replica, §7) needs a shared store, and the UI needs either sticky sessions or a shared session store. Engage CybrIQ for the multi-replica reference architecture.

14 Validate, then go live

- Stand up the tier, load your real inventory (or a representative sample of it), and run a short load test at your **peak operator count (B)**. CybrIQ provides the test harness.
- Watch four signals: application heap usage (stay under about 70% of `-Xmx`), how long requests wait for a database connection (`hikaricp.connections.pending` near zero), p95 response time on the dashboards and your busiest lists, and `crossconnect.batfish.fallback` (should stay at 0).
- Set up **liveness** and **readiness** probes; the app becomes ready in about 9 seconds and warms its caches in the background, so roll upgrades one replica at a time for zero downtime.
- If any signal is tight, move up one tier on the constrained component only (usually CPU for operators, RAM and storage for fleet size and retention, and a dedicated engine host if lock-wait is rising).

15 Worked example

Inputs: 1,200 devices growing to ~1,800 (A), 20 peak operators (B), 9 months retention (C), an integration sync at ~200 req/min (D), single node acceptable (E).

Tier: ~1,800 devices and 20 operators → **Standard** (covers up to 5,000 / 25).

The deploy (§§10–12):

- **One node:** 8 vCPU, 32 GB RAM, 250 GB SSD (grow to ~400 GB for 9 months of history).
- **App:** 4 vCPU, 6 GB, `-Xmx4g`, DB pool 25.
- **PostgreSQL:** 4 vCPU, 12 GB, `max_connections=150`, `shared_buffers=4GB`, `work_mem=48MB`.
- **Batfish:** co-located, 8 GB fixed heap.
- **Settings:** rate limit raised to `300` (200 for the sync + headroom), require-secrets on, admin + signing secrets set.

Validate: load test at 20 concurrent operators plus the 300/min sync; expect sub-100 ms dashboards and near-zero pool waits, with `crossconnect.batfish.fallback` at 0, then go live.

16 Reference: measured performance

The sizing above is anchored to a real, measured reference workload (a Standard-class node, 4 vCPU, 155 devices):

Measure	Result
Dashboard home, 90 concurrent loads	~15 ms (median)
Dashboard home, 150 concurrent loads	~34 ms (median)
Device list (hundreds of rows), 60 concurrent	~66 ms
Errors / throttled requests under load	0
Heap used at 155 devices	~0.5 GB
Time to readiness on restart	~9 s

The full method and complete result set are in the CrossConnect Performance Report.

How to read this guide. The tiers and settings are conservative starting points, not contractual limits. Your real numbers depend on how many interfaces each device has, how long you keep history, your integration load, and database tuning. Always run the §14 load validation at your real fleet and operator counts before going live, and engage CybrIQ for X-Large and distributed deployments.

A Appendix · tunables & defaults reference

The settings that matter for sizing, with the values the build ships with. The defaults are chosen to work out of the box; a production deployment sets the heap, pool, secrets, and rate limit explicitly.

Setting	Default	Notes
Server port	`8080`	`SERVER_PORT`; response compression on, 1 KB min
HikariCP max pool	10 (framework)	`SPRING_DATASOURCE_HIKARI_MAXIMUM_POOL_SIZE`; size per tier
Hibernate ddl-auto	`validate`	Flyway 10.20.1 owns schema; Hibernate never mutates it
Fleet-read cache TTL	60 s	per-tenant device configs; plain short-TTL cache
Report memo TTL	15–30 s	per-tenant dashboard rollups; single-flight on miss
Batfish finding cache (disk)	30 d	`crossconnect.batfish.cache-dir` (default temp dir)
Batfish connect / analyze timeout	5 s / 12 s	fair-lock-serialized single session; fallback on timeout
Discovery sweep	5 min, disabled	`CROSSCONNECT_DISCOVERY_INTERVAL_MS` / `…_ENABLED`; 20 s SSH timeout
Webhook workers / timeout	4 / 3 s	`CROSSCONNECT_WEBHOOKS_WORKER_THREADS` / `…_TIMEOUT_MS`; 6 attempts, backoff to 1h
Rate limit	100 / 60 s	`CROSSCONNECT_RATELIMIT_REQUESTS_PER_WINDOW` / `…_WINDOW_SECONDS`, per (tenant, IP)
AI intent TTL / sweep	15 min / 60 s	confirm-before-commit intents; `CROSSCONNECT_AI_INTENT_SWEEP_FIXED_RATE_MS`
Session token TTL	8 h	`CROSSCONNECT_AUTH_TOKEN_TTL_HOURS`
Tracing export	off	`OTEL_EXPORTER_OTLP_ENDPOINT` when enabled (OTel 1.43.0)

B Appendix · meters reference

The custom meters most useful for capacity work. Read them at `GET /actuator/metrics/{name}` with no extra infrastructure, or scrape them from `/actuator/prometheus`. The standard framework meters (`jvm.*`, `hikaricp.*`, `http.server.requests`, `system.cpu.usage`) are exposed too.

Meter	Type	What it tells you
`crossconnect.batfish.analyze`	Timer	How long a per-config analyze takes end to end, including timeouts; the engine's latency profile

Meter	Type	What it tells you
`crossconnect.batfish.analyze.lockwait`	Timer	Time spent waiting in line for the shared session before analyze; rising means the engine is contended, so move it to its own host
`crossconnect.batfish.fallback`	Counter	Analyses that fell back to the rule-of-thumb estimate (engine error or timeout); a sign of degradation, should be 0
`hikaricp.connections.pending`	Gauge	Requests waiting for a database connection; above 0 for a sustained period means the pool is too small
`hikaricp.connections.active`	Gauge	Connections currently in use; at the max for a sustained period means the pool is full
`jvm.memory.used` / `max`	Gauge	How full the heap is; alert above 85% of `-Xmx`

CrossConnect by CybrIQ · Performance & Capacity Planning · 21 June 2026 · Tunables and meters reflect the shipped operator-preview build; sizing tiers are conservative starting points validated against a measured reference workload, not contractual limits. · contact_us@cybriq.io