The Unified Sovereign-Tenant Cloud: Linear Shared-Nothing Scaling via Per-Tenant SQLite and In-Process Composition

Hanzo AI Research
Hanzo Industries Inc
research@hanzo.ai

Measurement pass: 2026-07-08 (host `evo`, AMD Ryzen AI Max+ 395, Go 1.26.4)

Abstract

Hanzo Cloud runs the entire control- and data-plane of a multi-tenant SaaS as *one* ZAP-native Go binary (56 in-process subsystems, one runtime) over a *shared-nothing* storage substrate: every tenant — org, app, project, or user — is its own embedded SQLite file, replicated to durable object storage as per-tenant post-quantum-encrypted LTX segments, served by a single pinned writer per file, with only the *active* tenants held open in a bounded per-pod LRU. This paper asks one question and answers it with measurement, not marketing: *what does a resident tenant actually cost, and how does the design scale?* We measure the marginal resident-set size (RSS) of opening $N \in \{1, 10, 100, 1000\}$ real base-schema tenant databases through the exact production open path, on the production driver. The answer: **an open, actively-served tenant costs ≈ 0.25 MB of RAM** (251 KB measured incremental slope; cgo/SQLite, warm workload) — roughly 8× below the 2 MB figure the design was previously budgeted against — and an *idle* tenant costs essentially zero live RAM (it is a file on disk plus an encrypted object). The tuned `cache_size` cap is a *ceiling*, not a footprint: for realistic tenants the working set, not the cap, sets the cost. From this one measured constant we build a capacity model: on the standard 128 GB unified-memory node, at 1% peak concurrency, **one billion registered tenants** is served by ≈ 20 nodes (≈ 2.5 TB cluster RAM); at 0.1% concurrency, the 3-node HA floor. Memory scales with *concurrency*, not with registrations. Because a 128 GB box is *unified*-memory, one such box also hosts the shared model weights for GPU inference in the same address space: a single $3,200 commodity box — the very class of hardware these numbers were measured on — runs the entire cloud (per-tenant SQLite *and* inference) and serves tens of millions of registered users from one self-installing binary (§6). We contrast this against the default industry shape — shared Postgres behind a microservice mesh — and quantify the penalties it pays that the unified design does not: a global write bottleneck, a hard connection ceiling, and an N-runtime plus sidecar memory tax we measure at 11–32× the unified binary's footprint. We are explicit about the limits: writer-pinning failover, hot-tenant skew, cross-tenant queries, and the routing directory are real costs, discussed honestly in §10. Every number labelled [measured] is measured on the named host; [modeled] and [cited] numbers are labelled as such; none are invented.

1 Introduction

The dominant shape of a multi-tenant SaaS backend is a fleet of stateless microservices in front of a shared relational database (typically Postgres), with tenancy expressed as a `tenant_id` column and enforced in application code or row-level security. That shape has two structural costs that grow

with the business. First, *writes serialize*: every tenant's mutations funnel through one primary's write-ahead log, so the write path is a single global resource no matter how many read replicas are added. Second, *runtimes multiply*: each microservice is a full process — its own language runtime, heap, connection pool, and (in a service mesh) a sidecar proxy — so the fixed memory cost of the platform is paid N times over, before a single byte of tenant data is stored.

Hanzo Cloud takes the opposite position on both axes, and this paper is the measured defense of that position. On composition: the 56 subsystems (IAM, KMS, Base, Commerce, AI, o11y, VFS, MQ, tasks, audit, ...) are linked into *one* binary and communicate in-process over a ZAP-native transport, with the same symbols exposed as ZAP-RPC only where process isolation is a compliance requirement (payments/vault, PCI CDE) [1, 3, 4]. The unified binary's construction is developed in the companion architecture paper [2]; this paper is its measured memory and scaling counterpart. On storage: there is no shared database. Each tenant is a sovereign, independently encrypted SQLite file [5]; the write path for tenant t is a single writer pinned to t, so writes are *embarrassingly sharded* and there is no global lock, no distributed transaction, and no per-write consensus round. The claim is that this is simultaneously the more *efficient* design (less RAM per tenant, far less fixed overhead) and the more *scalable* one (node count grows linearly in active tenants, with no shared bottleneck to saturate).

Claims about efficiency and scale are cheap; the contribution here is to make them measurable. We (i) establish the real per-active-tenant memory constant by direct measurement through the production open path (§3); (ii) turn that constant into a capacity model from 100 to 10^9 registered tenants (§4); (iii) quantify the memory gap against the shared-Postgres-plus-mesh alternative (§7); and (iv) state the scaling claims as propositions grounded in the measurement, with an honest accounting of where the design pays (§8–§10).

2 Architecture

One binary, 56 in-process subsystems. Per HIP-0106 [1], every Hanzo Go service exposes a top-level `Mount(app, deps)` symbol and is compiled into a single `cloud` binary rather than deployed as an independent service. Subsystems consume each other through Go interfaces that resolve to *in-process* implementations by default (the in-process plugin/composition model of HIP-0116 [3]), falling back to ZAP-RPC only for the small set of subsystems that must stay in a separate process for compliance isolation. HIP-0125 [4] eliminates gRPC from the transport entirely in favor of ZAP-native framing, so there is no per-subsystem gRPC server, no protobuf reflection service, and no sidecar. The consequence for this paper is that the platform's *fixed* memory cost — one Go runtime, one scheduler, one set of shared caches and connection machinery — is paid *once* per node, not once per subsystem.

The sovereign tenant: one encrypted SQLite file each. Tenancy is a composable hierarchy under one org: an org-wide database, a per-app database, a per-app-per-project database, and a per-user database [5]. Each lands on disk at `{DataDir}/orgs/{orgSlug}/{service}.db` and is opened through exactly one code path — the canonical Hanzo SQLite driver with a fixed pragma battery (§3). There is no shared schema and no cross-tenant table: tenant isolation is a *filesystem* boundary, not a `WHERE tenant_id = ?` predicate, which removes an entire class of tenant-bleed bugs by construction [8].

The pinned single writer. A single tenant is served by exactly one pod at a time via gateway-side sticky affinity (consistent hash on `X-Org-Id`). Because SQLite is a single-writer store and

each tenant maps to one file served by one pod, the writer for tenant t is globally unique without any coordination protocol. Reads may be served by read-only replica pods that hydrate from the replication stream and fail *closed* on any mutating verb, so correctness never rides on routing being perfect. This is the mechanism behind the linear-scaling claim: distinct tenants share no write path, so adding tenants adds independent writers rather than contending for one.

Durability: encrypted replication, not a shared database. Per HIP-0302 [5], each tenant's SQLite file is continuously replicated as LTX segments through `encreplica`, which post-quantum-`age`-encrypts every segment with a *per-tenant* key before it touches durable storage (the HIP-0107 replication substrate [6]; local FS for dev, S3 or Hanzo VFS in production [7]). The storage and encryption architecture is developed in the companion post-quantum object-store paper [9]; this paper concerns its memory and scaling behavior. At-rest encryption therefore lives on the replication path, not in the resident page cache: the open handle is plaintext SQLite (fast), and the durable copy is ciphertext keyed to the tenant. This distinction matters for the memory model — the crypto cost is transient (a buffer during flush), not a standing per-connection tax.

Active-set residency. Only *active* tenants are held open. The per-pod store keeps open SQLite handles in a bounded LRU (default cap 1000 handles/pod, tunable) with a 5-minute idle reaper; on eviction it checkpoints the WAL, closes the handle, and flushes the encrypted bytes to the object store. A cold tenant occupies no RAM — it is an encrypted object plus, at most, a file in a local cache. This is the architectural reason the resident memory of a node tracks *concurrency* rather than *registrations*, which §3 then quantifies.

3 Benchmark: the real per-tenant memory constant

Question. What is the marginal resident memory of one open, actively-served tenant database, measured through the exact production open path, on the driver the production binary actually links?

Method. We open tenant databases with the identical call the production store uses,

```
dbx.Open("sqlite", sqlite.PragmaDSN(path, sqlite.DefaultPragmas)),
```

where `DefaultPragmas` is the canonical Hanzo battery: `busy_timeout=10000`, `journal_mode=WAL`, `synchronous=NORMAL`, `foreign_keys=ON`, `temp_store=MEMORY`, `cache_size=-32000` (a 32 MiB page-cache *cap* per connection). The seed is the checked-in base test database (`base/tests/data/data.db`, 344,064 B data + 49,152 B auxiliary $\approx$ 0.38 MB), a realistic small tenant carrying the base system collections plus a little data. For $N \in \{1, 10, 100, 1000\}$ we copy the seed to a fresh path, open it, run a workload, and keep the handle resident; at each checkpoint we force a Go GC and `FreeOSMemory` and read `VmRSS` from `/proc/self/status`. We report the *incremental slope* $\Delta\text{RSS}/\Delta N$ between checkpoints, which cancels the fixed process baseline and yields the true per-tenant constant. RSS (not Go heap statistics) is the honest measure because the cgo SQLite page cache is allocated in the C heap, invisible to Go's allocator. Two workloads: *cold* (open + a single `SELECT 1`) and *warm* (touch every table's pages via `count(*)` then perform a small write, materializing the WAL, the `-shm` index, and the pinned writer connection). The production driver is cgo/`mattn` (the `cloud` binary links it as the active `"sqlite"` driver); we also measure the pure-Go `modernc` backend used on `!cgo` builds. Host: `evo`, AMD Ryzen AI Max+ 395 (32 cores), 62.4 GB, Linux 6.17.0-35, Go 1.26.4.

Result: ≈0.25 MB per active tenant. Table 1 gives the measured constants. On the production driver, a warm resident tenant costs a measured **251 KB** (≈0.245 MB) at the asymptotic (100 → 1000) slope; a cold-but-open tenant costs 235 KB. The pure-Go backend is ≈1.5× heavier at 392 KB. The single most important finding is in the `cache_size` column: sweeping the cap across {−256, −1024, −2000, −32000} (256 KiB to 32 MiB) moves the small-tenant constant by *less than 0.2%* (251.3–251.7 KB). The 32 MiB cap is never approached, because the tenant's working set (≈0.25 MB) — not the cap — sets the footprint. The four cache settings are in effect four independent repeats at equal working set, and their agreement to within 0.4 KB is our repeatability evidence.

Table 1: Measured marginal RSS per open, resident tenant (base-schema seed, 0.38 MB on disk). Incremental slope is $\Delta\text{RSS}/\Delta N$ over 100 → 1000 handles; baseline (runtime + driver, zero tenants) is 7.0–7.2 MB (cgo), 6.3–6.5 MB (modernc). [measured]

Driver	Workload	`cache_size`	Per-tenant RSS
mattn / SQLite (cgo, *production*)	warm	−32000 (32 MiB)	**251.3 KB** (0.245 MB)
mattn / SQLite (cgo, *production*)	warm	−2000 (2 MiB)	251.4 KB
mattn / SQLite (cgo, *production*)	warm	−1024 (1 MiB)	251.3 KB
mattn / SQLite (cgo, *production*)	warm	−256 (256 KiB)	251.4 KB
mattn / SQLite (cgo, *production*)	cold	−32000	235.5 KB (0.230 MB)
modernc (pure-Go, `!cgo`)	warm	−32000	392.0 KB (0.383 MB)

The density knob is real — for *hot, large* tenants only. That `cache_size` is inert for small tenants is not the whole story. We re-ran the sweep against a synthetic 25 MB tenant under a full-scan warm workload (Table 2). Here the cap binds: at −32000 a fully-scanned hot tenant fills its page cache to the working set and costs 25.5 MB; at −1024 it is bounded to 1.18 MB; at −256 to 0.41 MB. So the cap is a per-tenant memory *ceiling* operators can dial to trade page-cache hit rate for density. The realistic operating point is Table 1 (working-set-bound, ≈0.25 MB); Table 2 is the worst case (analytic tenant scanning its whole database every request) and shows it remains bounded and tunable rather than unbounded.

Table 2: Density knob: marginal RSS for a *hot* 25 MB tenant under a full-table-scan workload, as a function of the `cache_size` cap. [measured]

`cache_size` cap	Per-tenant RSS	Effect
−32000 (32 MiB)	25.5 MB	cache fills to working set (cap not reached)
−1024 (1 MiB)	1.18 MB	bounded to ≈1 MiB regardless of DB size
−256 (256 KiB)	0.41 MB	bounded to ≈256 KiB + fixed overhead

Idle tenants cost ≈0. We opened and exercised 1000 tenants, then closed every handle. On the pure-Go backend — whose memory frees to the Go heap, which `FreeOSMemory` returns to the OS — RSS fell from 392 MB back to **11.2 MB**, i.e. to within 4.9 MB of baseline over 1000 closed tenants (≈5 KB/tenant of retained runtime bookkeeping). This is the direct measurement behind "an idle tenant is free": closing a handle releases essentially all of its live memory. On the cgo backend RSS settled at 221 MB after close; that residual is *not* live memory but glibc's retention

of freed C arenas (the allocator does not `munmap` them back to the kernel). We confirmed it is reused, not leaked: closing 1000 tenants and opening 1000 fresh ones grew RSS by only ≈110 MB on top of the 222 MB retained (vs. 247 MB for a cold start), so the freed arenas absorb subsequent opens. The operational takeaway: a pod's steady-state RSS tracks its *high-water active set*, and idle registrations contribute no live pages.

On-disk and base-node constants. A realistic tenant occupies ≈0.38 MB on disk (measured seed), which at rest lives as per-tenant-encrypted LTX in object storage [7, 5], not on node-local disk. The fixed per-node cost — the `cloud` binary with all 56 subsystems and one Go runtime — is a working-set RSS of **154–225 MB** from prior production measurement [10]; we use 190 MB as the point estimate in the model below and note it is [cited] (the cluster was not reachable from the measurement host on this pass to re-sample it live). For comparison, a *bare* Go HTTP service (net/http, one handler, no framework) measured [measured] 7.2 MB on the same host — the floor a per-service split cannot go below, used in §7.

4 The capacity model

The measured constant makes the fleet size a closed-form function of registrations and concurrency. Let R be registered tenants, c the peak concurrency fraction, so active tenants $A = cR$. Let m be the measured per-active-tenant RAM (MB), b the per-node base RSS (GB), and U the usable RAM per node (GB) after reserving for the OS, kubelet, and burst headroom. Then

$$\text{nodes} \;=\; \max\!\left(3,\; \left\lceil \frac{A \cdot m}{(U-b)\cdot 1024} \right\rceil\right), \qquad \text{cluster RAM} \;=\; \text{nodes}\cdot b \;+\; \frac{A\cdot m}{1024}\ \text{GB}. \tag{1}$$

The standard Hanzo node is a **128 GB** unified-memory box (§6); we use $U = 125$ GB usable, $b = 0.19$ GB, and the measured $m = 0.25$ MB, giving a per-node density of ≈511,000 active tenants; we also report the conservative $m = 1.0$ MB envelope (larger/hotter tenants, multiple resident connections, extra headroom), which yields ≈128,000 active/node. A 128 GB node holds ≈4× the active tenants of a 32 GB node, so the fleet is ≈1/4 the size at the same load: 10^9 registered tenants at 1% concurrency is ≈20 128 GB nodes versus ≈103 32 GB nodes (both from the *measured* m, not a 2 MB assumption). Disk is $R \times 0.38$ MB total, object-tiered. Table 3 gives the model at 1% peak concurrency; Table 4 shows concurrency sensitivity at the extremes.

Table 3: Capacity at **1% peak concurrency**, **128 GB** nodes ($U = 125$ GB), from Eq. 1 with the [measured] $m = 0.25$ MB constant (and the [modeled] $m = 1.0$ MB conservative envelope). "Cluster RAM" is consumed RAM per Eq. 1, not provisioned. Node counts are floored at 3 for HA.

Registered R	Active @1%	Nodes (m=0.25)	Cluster RAM	Nodes (m=1.0)	Disk (total, tiered)
100	1	3	0.6 GB	3	38 MB
1,000	10	3	0.6 GB	3	380 MB
10,000	100	3	0.6 GB	3	4 GB
100,000	1,000	3	0.8 GB	3	38 GB
1,000,000	10,000	3	3.0 GB	3	380 GB
10,000,000	100,000	3	25.0 GB	3	4 TB
100,000,000	1,000,000	3	244.7 GB	8	38 TB
1,000,000,000	10,000,000	**20**	**2.45 TB**	79	380 TB

Table 4: Concurrency sensitivity for $R = 10^9$ registered tenants ($m = 0.25$ MB, 128 GB nodes). Memory scales with *concurrency*, not registrations: a 50× swing in peak concurrency is a 50× swing in fleet size, while R is held fixed. [measured]-grounded.

Peak concurrency	Active tenants	Nodes (128 GB)	Cluster RAM
0.1%	1,000,000	3	245 GB
1.0%	10,000,000	20	2.45 TB
5.0%	50,000,000	98	12.2 TB

The shape of Table 3 is the thesis in numbers. Up to *one hundred million* registered tenants at 1% concurrency, the fleet is the 3-node HA floor — the platform is base-RSS-bound, not tenant-bound, and the tenants are "free" relative to the fixed cost of running at all. Only past $\sim 10^8$ registrations does tenant memory dominate, and there it grows *linearly*: 10^9 tenants is 20 128 GB nodes, 10^{10} would be ∼200. There is no knee, no bottleneck tier, and no rewrite between the 3-node and the 20-node regime — the same binary and the same per-tenant file, more of them.

5 The two scaling axes: tenant state vs. GPU inference

The capacity model of §4 sizes *one* resource: per-tenant SQLite state, which is CPU/RAM/IO. A GPU-enabled build of the same binary adds a *second*, orthogonal axis — inference — and it scales by a completely different law. Conflating the two is the standard modeling error; separating them is the point.

The inference engine, hanzo-engine (Rust), is FFI-embeddable into the cloud binary through a stable C ABI (`hanzo_engine_ffi.h`, present in the engine tree), so a GPU node runs inference and embeddings *in the same process* as the app subsystems and the tenant stores — no separate model service, no network hop to a model server (the co-location is the cloud "bring-your-gpu" path). The two axes then scale as follows.

- **Axis A — tenant state (CPU/RAM).** Resident memory is $A \cdot m$ with $A = cR$: it grows with the number of *active* tenants, and each tenant's state is private (shared-nothing). This is §4, measured at $m \approx 0.25$ MB.
- **Axis B — inference (GPU/VRAM).** GPU memory holds *model weights*, which are *shared across all tenants*: one resident copy of a model serves one tenant or one million at the same VRAM cost. Inference memory is therefore O(distinct models hosted), *not* O(tenants). Adding tenants adds no weight memory — only queue depth, absorbed by request *batching*. Axis B scales by (i) model-hosting: VRAM = weights + KV-cache ([cited] ≈8.11 GiB resident for Qwen3-8B-Q8_0), and (ii) throughput: tokens/sec, bounded by memory bandwidth ([cited] ≈231 GB/s and single-stream decode ≈25 t/s on the measured Strix Halo box; an NVIDIA GB10 Grace-Blackwell offers 121 GB unified [11]).

Because the axes are independent, a deployment sizes them independently: tenant-state nodes by Eq. 1, inference nodes by peak token demand ÷ per-node batched throughput. A single node carries *both* roles when it has RAM for tenants and VRAM (or unified memory) for weights — exactly the box of §6. The weight-sharing property is what makes inference cheap to co-locate: the model is a fixed cost amortized across the *entire* tenant population, not a per-tenant tax. (Axis-A numbers are [measured]; the Axis-B weight/throughput figures are [cited] from the companion inference papers [11, 10].)

6 The unified-memory box as the atomic self-hostable unit

Put the two axes in one box with one physical memory pool and the architecture collapses to something new: a complete, self-hostable cloud in a single commodity unit.

The enabling hardware is the *unified-memory* box — a Grace-Blackwell GB10-class system (e.g. NVIDIA DGX Spark, [cited] 128 GB unified LPDDR5X) or an AMD Ryzen AI Max+ 395 "Strix Halo" box (the class this paper's per-tenant constant was *measured* on) at 128 GB unified for ≈$3,200. *Unified* means CPU and GPU address the same physical DRAM: the per-tenant SQLite population (Axis A) and the model weights (Axis B) coexist in one 128 GB space with *no* CPU↔GPU copy — the tenant data the app serves and the model the app calls live in the same address space. One box therefore runs the entire stack: the one binary with all 56 subsystems, per-tenant storage, replication, *and* GPU inference.

Table 5 sizes it: reserve model weights + engine + OS, and the remainder $\div m$ is the tenant capacity. With an 8B-Q8 model (8 GiB) and ≈6 GiB for engine/OS, ≈114 GiB remains for tenant state = a [measured]-grounded 467,000 active tenants; at 1% concurrency that is **≈47 million registered users on one $3,200 box**, at 0.1% concurrency ≈467 million. Even at the conservative $m = 1.0$ MB it is ≈11.7 million registered at 1%.

Table 5: "Cloud in a box": tenant capacity of one 128 GB unified-memory box after reserving model weights + engine + OS. Active-tenant counts use the [measured] m; registered-user counts apply the stated concurrency; weight reservations are [cited]/[modeled]. Capex is the $3,200 box price ÷ registered users at 1% concurrency.

Resident model	Tenant RAM	Active (m=0.25)	Reg. @1%	Reg. @0.1%	Capex/user @1%
8B-Q8 (≈8 GiB)	114 GiB	467,000	**46.7 M**	467 M	$0.000069
≈20B-Q8 (≈20 GiB)	102 GiB	418,000	41.8 M	418 M	$0.000077
70B-Q4 (≈40 GiB)	82 GiB	336,000	33.6 M	336 M	$0.000095

The economics follow directly. At $3,200 per box and ≈47 M registered users at 1% concurrency, the one-time hardware capex is **≈$0.00007 per registered user** ($68 per million). Scale is additive: each box is self-contained (shared-nothing tenants, its own model copy), so N boxes are $N\times$ the capacity with no shared bottleneck. A sovereign, full-stack AI cloud — app, storage, *and* inference — is a rack of commodity boxes, not a hyperscaler contract. (The tenant capacity is [measured]-m-grounded; the weight reservation and inference throughput are [cited]/[modeled]; a real box's capacity depends on the p99 tenant size and the model chosen.)

7 Comparative analysis: shared Postgres + microservice mesh

We compare against the default industry shape on the three axes where the unified design's structure diverges. The comparison is deliberately conservative: where we estimate the alternative we anchor to a measured floor and label the estimate [modeled].

(1) The write path: sharded vs. global. In shared Postgres, every tenant's writes commit through one primary's WAL. Read replicas scale reads but not writes; the write ceiling is one node's fsync throughput, and it is *shared* across all tenants, so a write-heavy tenant degrades every other tenant's write latency. Scaling writes past one primary requires sharding (Citus, Vitess-style) or multi-primary consensus (Raft/Paxos per write) — either a re-architecture or a per-commit coordination round. In the unified design the write path for tenant t is a private SQLite writer; N

tenants are N independent writers with *no shared lock and no consensus round*. Aggregate write throughput is the sum of per-tenant throughputs and grows linearly with nodes. The cost paid for this is that a single tenant's writes are bounded by one file's single-writer throughput (§10); the benefit is that there is no global write resource to saturate.

(2) The connection ceiling. A Postgres primary allocates a backend process per connection (∼5–10 MB each), so practical `max_connections` is in the hundreds; thousands of tenant-affine connections require a pooler (PgBouncer) and still contend for the one primary. Per-tenant SQLite has no connection ceiling: a "connection" is an in-process handle into a memory-mapped file, and the active-set LRU (§2) bounds the resident count per pod directly. There is no network round trip and no server-side backend process on the data path at all.

(3) The N-runtime + sidecar memory tax. This is the axis we can quantify most directly. The unified binary runs all 56 subsystems in one Go runtime at a measured/cited base of 190 MB; at 3 replicas for HA that is 570 MB of control-plane RAM before tenant data. A microservice mesh runs each subsystem as its own process with its own runtime heap and, under a service mesh, a sidecar proxy. Even at the *measured floor* of a bare Go service (7.2 MB [measured]) plus a single Istio/Envoy sidecar (≈50 MB [cited] [19]) at 2 replicas, 56 subsystems cost 6.4 GB; at a realistic 60 MB/service ([modeled]: framework + OTEL SDK + DB pool + business logic) plus 50 MB sidecar, 12.3 GB. Table 6 gives the multiples: the mesh spends 11–32× the unified binary's fixed memory, and adds one network hop (plus TLS and serialization) to every inter-subsystem call that the unified design makes as an in-process function call.

Table 6: Fixed control-plane memory: unified binary vs. 56-subsystem microservice mesh, before any tenant data. Unified base 190 MB is [measured]/[cited]; bare-Go 7.2 MB is [measured]; per-service framework RSS and sidecar RSS are [modeled]/[cited]. Multiple is mesh ÷ unified.

Deployment	Per-service	Fixed RAM	× unified (3-rep, 570 MB)
Unified binary, 3 replicas	–	570 MB	1.0×
Mesh, floor (7.2 MB svc + 50 MB sidecar, 2 rep)	57.2 MB	6.4 GB	11.2×
Mesh, realistic (60 MB svc + 50 MB sidecar, 2 rep)	110 MB	12.3 GB	21.6×
Mesh, heavy (120 MB svc + 70 MB sidecar, 2 rep)	190 MB	21.3 GB	37.3×

Composed with the capacity model, the gap compounds: the mesh's 6–21 GB fixed tax is paid on *every node*, so at the 3-node HA floor — where the unified platform serves up to 10^8 registered tenants at 1% concurrency in <1 GB of control-plane RAM (Table 3) — the mesh alternative already needs 19–64 GB just to exist.

8 Claims, grounded in the measurement

We state the design's guarantees as propositions, each tied to a measured or structural fact rather than an assertion.

Proposition 1 (memory ∝ concurrency, not registrations). Resident memory is $\text{nodes} \cdot b + A \cdot m$ with $A = cR$ (Eq. 1). Because an idle tenant contributes ≈0 live RAM (measured: 1000 closed handles return to within 4.9 MB of baseline, §3) and only the active-set LRU is resident, RAM is a function of A, and R enters only through $A = cR$. Doubling registrations at fixed concurrency

does not change resident memory; it changes only object-storage bytes. Table 4 is the empirical shape.

Proposition 2 (linear node scaling from shared-nothing writers). Distinct tenants share no write path: each is a private single-writer SQLite file pinned to one pod (§2). There is no global lock, no distributed transaction, and no per-write consensus, so adding a node adds independent write and read capacity with no coordination overhead that grows with the fleet. Fleet size is therefore $\Theta(A)$ — on 128 GB nodes, 20 at 10^7 active, $\sim$196 at 10^8 — with a constant factor set by the measured m. This is linear, not merely sublinear-with-a-bottleneck.

Proposition 3 (the one-binary memory advantage is multiplicative). Composing S subsystems into one runtime pays the fixed runtime + sidecar cost once rather than S times. Measured against a 56-subsystem mesh, the fixed-memory ratio is 11–32$\times$ (Table 6), and it is paid per node, so it multiplies the per-node base b in Eq. 1. Lowering b is what keeps small deployments in the sub-gigabyte regime in Table 3.

9 hanzo.network: the sovereign decentralized confidential compute fabric

Everything above describes *one* operator's fleet. The culminating step makes it a public, decentralized, confidential cloud: a fabric of independently-owned boxes coordinated leaderlessly by a BFT chain, where anyone can contribute compute and confidential workloads run on hardware the operator cannot inspect. The primitives below are real and cited to their source files; the assembly into a single fabric is stated honestly as roadmap where it is roadmap (§9.1).

The control plane is a consensus chain. hanzo.network is a `luxfi/consensus` BFT chain [12] — the engine is real (`engine/bft`, `engine/chain`, `engine/common`) with the Quasar protocol (`protocol/quasar`) [13] — whose VM *state is* the compute-fabric control plane: node membership and attestation, the tenant→writer-node pinning map, replication assignments, and workload placement. The chain *is* the scheduler — leaderless BFT, no central coordinator, no single point of control or failure.

The data plane is the per-tenant SQLite substrate — unchanged. Each node runs `hanzod` (the node daemon; `hanzo/node`) which spawns the full cloud binary: a consensus participant *plus* the 56 app subsystems *plus* GPU inference (§5). Tenant writes stay exactly as measured — a private, writer-pinned SQLite file, WAL-replicated to consensus-agreed replicas. The chain coordinates *placement*, not individual writes.

Leaderless HA and durability — the sweet spot. The design sits deliberately between two failure modes. *Global consensus per write* (every mutation is a BFT round) gives durability but does not scale — the write path becomes one global bottleneck, the very thing §7 faults Postgres for, made worse. *Replication without consensus* scales writes but has no agreed failover — a split brain can double-open a tenant or lose its tail. hanzo.network takes neither: per-tenant single-writer for the *data* path (writes shard perfectly), and BFT consensus only for *placement* — who holds the writer pin, where the replicas live, and, on writer-node death, which replica is promoted. The writer pin (`cloud/writerpin`) already guarantees at-most-one writer; the writer/reader roles

(`cloud/role`, `middleware_reader.go`, fail-closed reads — shipped in the reader-HA work) already give safe promotion. The chain makes that promotion decision *deterministic and Byzantine-fault-tolerant across untrusted operators*. Writes never touch consensus; only the rare, slow-changing placement map does. That is why the fabric scales linearly *and* survives node loss.

Trust tiers make it trustless-public. A public fabric of unknown operators cannot be trusted with plaintext tenant data unless the hardware enforces it, so hanzo.network schedules by *trust tier*, recorded on-chain. *Confidential tier*: nodes with a hardware TEE — NVIDIA H100/H200 Confidential Computing, Intel TDX, or AMD SEV-SNP [14] — run tenant workloads inside an enclave the node *operator* cannot read; the attestation is verified and recorded on-chain, and confidential workloads schedule *only* onto attested nodes. *Public tier*: any commodity box (a phone, a laptop, the $3,200 unified box) takes public/research workloads where the operator is trusted enough for public data. Anyone can run `hanzod` and contribute, and the chain routes each workload to hardware whose trust tier matches its sensitivity — the difference between a genuinely *trustless* decentralized cloud and "decentralized, but trust the operators."

Post-quantum from the identity layer up. A sovereign cloud is a decades-long artifact, so its security horizon must not be capped by Shor's algorithm. hanzo.network is post-quantum *end-to-end*, not classical-with-a-PQ-bolt-on: consensus runs a PQ security profile (`luxfi/consensus config/pq_mode.go`, `security_profile.go`) and PQ auth schemes (`protocol/auth`: ML-DSA-44/65 signatures, ML-KEM key exchange, legacy ECDSA *forbidden* under strict PQ) [15]; Quasar is a PQ BFT (BLS + Ringtail threshold + ML-DSA), and the primitives ship in `lux/crypto/{mldsa,mlkem}`. Node identity is an ML-DSA key; capability identity on the app side is PQ as well (`iam/capauth/{keys,verify}.go`). This is a durability argument as much as a security one: node identity, consensus, and key exchange all outlive the quantum transition.

Frictionless, auto-provisioned wallet. Every node and user has a PQ wallet, and if none is configured one is provisioned automatically as part of joining — there is no "install a browser extension, back up a seed phrase" barrier. The wallet subsystem is real (`cloud/clients/wallets`: `wallets.go`, `custody.go`, `anchor.go`, `store.go`, `safeclient.go`, over `clients/mpcseal`), MPC-custodied so there is no single seed to lose. The wallet is simultaneously the node's PQ *identity* and its economic *key*, so `hanzod up` yields identity, wallet, and fabric membership in one step — essential to the viral model below.

Economics: the AI-coin incentive. The wallet stakes to join the fabric, earns for contributed compute — priced by trust tier, so a TEE-confidential node earns more than a public one — and pays for consumed workloads, all in the native AI coin (HIP-0001 [16]). This is the incentive that makes a viral node network *self-sustaining*: contributing compute earns, so nodes join and stay. Against the §6 box economics — a $3,200 128 GB box whose measured per-tenant capex is ≈$0.00007/user — a node's contributed compute earns against a trivial fixed cost, which is what closes the loop.

9.1 Viral deployment: `hanzod` is the unit, `cloud` is what it runs

The reason this can actually spread is the deployment model. Every prior "self-hostable cloud" gates on operator expertise — stand up Kubernetes, then deploy — which restricts the operator population to specialists. hanzo.network inverts it: you run *one* daemon. `hanzod` (`hanzo/node`,

a systemd-managed node service) is the viral unit; the cloud binary is the state machine it runs — the relationship is `hanzod : cloud :: bitcoind : the ledger`. The daemon (i) *joins hanzo.network* as a leaderless `luxfi/consensus` node; (ii) *manages its own runtime adaptively* — on a laptop or small box it spawns the cloud binary bare (no Kubernetes), and on a GPU/large node it bootstraps k3s *internally* for workload isolation and GPU scheduling and runs cloud + workloads inside it, so Kubernetes is an internal implementation detail of the node (the HIP-0117 [17] `cluster init` mechanic wrapped by the daemon), invisible to the operator; (iii) *contributes compute by trust tier* (TEE→confidential, bare→public); and (iv) *self-installs, self-updates, self-heals.* The whole onboarding is `download hanzod` → `hanzod up` → `you are an open-AI-cloud node`: one command, no Kubernetes expertise, PQ identity + wallet + membership included.

The virality argument is then structural: **(a) no expertise barrier** — one download and one command remove the k8s-operator gate that every prior decentralized-cloud project tripped on; **(b) monotone network effect** — each joining node strictly increases fabric capacity, resilience, *and* trust-tier coverage, so growth compounds; **(c) universal addressable population** — trust-tiered participation makes the eligible node set *everyone* (a phone serves public inference; a TEE box serves confidential tenants), not just datacenter operators; **(d) trivial per-node cost** — a $3,200 128 GB unified box is a consumer purchase whose measured per-tenant economics make contributed compute effectively free at the margin, and the AI-coin incentive pays it back. Against hyperscaler centralization (one owner, metered rent, plaintext trust) and against prior decentralized-cloud efforts (still operator-expert, still trust-the-operator), the combination — one-command node, leaderless BFT control, per-tenant shared-nothing data, hardware-enforced confidential tiers, PQ identity, an auto-wallet, and commodity unified-memory economics — is what turns a measured efficiency result into a growth model.

What is built vs. what is the assembly. Built (cited to source): the BFT + Quasar consensus engine (`luxfi/consensus`, PQ mode + ML-DSA/ML-KEM auth); the per-tenant writer pin (`cloud/writerpin`); the writer/reader HA roles (`cloud/role` + `middleware_reader.go`); the node daemon (`hanzod`); the unified cloud binary (HIP-0106) and in-process plugin/VM model (HIP-0116); the wallet/custody subsystem (`cloud/clients/wallets` + `mpcseal`); PQ capability identity (`iam/capauth`); the PQ crypto primitives (`lux/crypto/{mldsa,mlkem}`); the GPU/engine FFI (`hanzo_engine_ffi.h`) and `lux/node/vms`; and the specs HIP-0117 (BYO compute fleet), HIP-0005 (PQ security), HIP-0001 (AI coin). **Roadmap (the assembly):** the fabric-VM that makes consensus state the live scheduler; the on-chain tenant→writer map with consensus-driven failover; on-chain TEE attestation and confidential trust-tier scheduling; the `hanzod` auto-join handshake, adaptive-k3s bootstrap, and self-heal orchestration; the auto-provision-on-join wallet flow; and the compute-earning settlement with cross-node trust-tier pricing. This paper's contribution is the *measured foundation*; the fabric is the composition of shipped primitives, marked as design where it is design.

10 Limitations

The rigor is the persuasion, so the costs are stated as plainly as the benefits.

- **Writer-pinning failover.** One writer per tenant means a pod loss requires re-pinning that tenant to a new pod and replaying the tail of the encrypted replication stream before writes resume. HIP-0302's consistency model is at-most-once under rebalance: a short single-writer overlap window can exist during an HPA move, and operators must drain before scaling. There is

a bounded write-unavailability window per affected tenant on failover; the CAS (ETag If-Match) follow-up is required for generation-checked writes across a rebalance.

- **Hot-tenant skew.** Capacity is modeled on an *average* tenant. One tenant that is simultaneously large and hot can, at the default 32 MiB cap, cost up to 25.5 MB resident (Table 2) and is bounded to one pod's single-writer throughput. The mitigations — a smaller per-tenant `cache_size`, splitting a heavy org across the app/project/user hierarchy, and per-tenant rate ceilings — are real levers but they are operational work, not free. The model's m should be chosen for the p99 tenant, not the mean, in skewed populations.

- **Cross-tenant queries.** A filesystem isolation boundary is exactly the wrong shape for a query that must span tenants (fleet-wide analytics, admin search, billing rollups). These require fan-out plus aggregation, or a separate columnar sink fed from the replication stream — neither is a single SQL statement. The design optimizes the 99% per-tenant path at the cost of the cross-tenant path.

- **The routing directory.** Sticky single-writer routing needs a consistent, durable map from tenant to pod. That directory is itself a piece of shared state that must be highly available and consistent; it is small and changes slowly, but it is a coordination point the pure shared-nothing story would prefer not to have.

- **Measurement scope.** The per-tenant constant is measured on one host, one seed schema, single-stream warm/cold workloads, with a bounded 2-connection pool. It does not sweep concurrent query load per tenant (which materializes more connections, each with its own page cache), nor multi-hour steady-state fragmentation. The 190 MB base RSS is [cited] from prior production sampling, not re-measured on this pass. These are the honest edges of the numbers.

- **The fabric is an assembly of shipped primitives, not a shipped fabric.** The per-tenant measurement, the unified binary, the writer pin, the writer/reader HA roles, the PQ consensus engine, the wallet subsystem, and the node daemon exist and are cited to source (§9.1). The single decentralized confidential fabric of §9 — the fabric-VM as live scheduler, on-chain tenant→writer placement with consensus failover, on-chain TEE attestation and confidential scheduling, the `hanzod` auto-join/adaptive-k3s/ self-heal orchestration, and the compute-earning settlement — is design/roadmap. The efficiency and scaling results are measured; the fabric is the argued composition on top of them, and is labelled as such throughout.

- **Inference axis not measured on this pass.** Axis-B (GPU) figures — model resident size, bandwidth, decode t/s — are [cited] from the companion inference papers, and the box capacity of Table 5 treats the model reservation and throughput as [cited]/[modeled]. The tenant capacity (Axis A) is the measured part; the inference co-location is architecturally grounded (FFI, unified memory) but its throughput is not re-measured here.

11 Conclusion

The unified sovereign-tenant design is efficient and scalable for the same structural reason: it refuses to share the two resources that normally become bottlenecks. It does not share a write path — each tenant is a private single-writer SQLite file — so writes shard perfectly and node count grows linearly in active tenants. It does not share N runtimes — 56 subsystems live in one binary — so the fixed platform cost is paid once, at 11–32× less memory than the equivalent mesh. The measured per-active-tenant cost is ≈0.25 MB, an idle tenant is free, and the `cache_size` cap is a tunable

ceiling rather than a footprint. The capacity model that follows from that one constant puts a billion registered tenants on ≈20 standard 128 GB nodes at 1% concurrency, and on the 3-node HA floor at 0.1% — with no bottleneck tier and no rewrite between the small and the large regime. Two orthogonal axes compose in one process: per-tenant state (CPU/RAM, O(active)) and shared-weight GPU inference (VRAM, O(models)), which on a $3,200 128 GB *unified*-memory box collapse into a complete cloud-in-a-box serving tens of millions of users. Extended over `luxfi/consensus`, a leaderless PQ BFT chain schedules placement (not writes) across independently-owned, trust-tiered nodes — the shipped writer-pin, reader-HA, wallet, and PQ identity primitives composed into `hanzo.network`, a sovereign decentralized confidential cloud that grows one `hanzod up` at a time. The design is not free of cost: writer failover, hot-tenant skew, and cross-tenant queries are genuine prices, and the fabric layer is an argued assembly of measured primitives, all stated in §10. But on the axis it is built for — serving an enormous number of independent tenants cheaply and scaling their concurrency linearly — the measurement supports the architecture.

A Reproducibility

Host. `evo`: AMD Ryzen AI Max+ 395 (32 cores), 62.4 GB RAM, Linux 6.17.0-35, Go 1.26.4, glibc malloc, TeX Live 2023.
Open path. `github.com/hanzoai/dbx v1.16.0` + `github.com/hanzoai/sqlite v0.2.1`; production driver cgo/`mattn` (`go-sqlite3 v1.14.47`); pure-Go control `modernc.org/sqlite v1.51.0`. Pragmas exactly `sqlite.DefaultPragmas`: `busy_timeout=10000`, `journal_mode=WAL`, `synchronous=NORMAL`, `foreign_keys=ON`, `temp_store=MEMORY`, `cache_size=-32000`.
Seed. `base/tests/data/data.db` (344,064 B) + `auxiliary.db` (49,152 B), the checked-in base-schema test database.
Procedure. Copy seed to a fresh path per tenant; open through the production path; run the *warm* workload (`count(*)` over every non-`sqlite_` table, then `CREATE TABLE IF NOT EXISTS` + five `INSERT`s) or the *cold* workload (`SELECT 1`); at each of $N \in \{1, 10, 100, 1000\}$ force `runtime.GC()` + `debug.FreeOSMemory()` and read `VmRSS` from `/proc/self/status`. Report the $100 \to 1000$ incremental slope. Idle test: close all handles, re-measure; reuse test: close all, open 1000 fresh, re-measure. Large-tenant knob: inflate the seed to 25 MB (60,000 padded rows) and repeat the `cache_size` sweep under a full-scan warm workload.
Capacity model. Eq. 1 with $U = 24$ GB, $b = 0.19$ GB, $m \in \{0.25, 1.0\}$ MB, 32 GB nodes, HA floor 3; disk $R \times 0.38$ MB object-tiered.

References

[1] Hanzo AI. *HIP-0106: Cloud — Unified Hanzo Binary*. https://github.com/hanzoai/HIPs.

[2] Z. Kelling. *The Unified Cloud Binary: Collapsing a Microservices Control Plane into One Go Process*. Hanzo Industries white paper, 2026 (companion architecture paper; `cloud-unified-binary`).

[3] Hanzo AI. *HIP-0116: Plugin VM / In-Process Composition Model*. https://github.com/hanzoai/HIPs.

[4] Hanzo AI. *HIP-0125: ZAP-Native Transport & gRPC Elimination* (renumbered from HIP-0116 to avoid collision with the plugin-vm model). https://github.com/hanzoai/HIPs.

[5] Hanzo AI. *HIP-0302: Hanzo Replicate — Encrypted SQLite Durability for Base Services*. https://github.com/hanzoai/HIPs.

[6] Hanzo AI. *HIP-0107: Replication and Durability Substrate*. https://github.com/hanzoai/HIPs.

[7] Hanzo AI. *HIP-0032: Object Storage Standard*. https://github.com/hanzoai/HIPs.

[8] Hanzo AI. *HIP-0118: SuperAdmin and Tenant Isolation Model*. https://github.com/hanzoai/HIPs.

[9] Z. Kelling. *A Post-Quantum Object Store: Per-Tenant SQLite and a Dropbox-Class Filesystem over One Encrypted Block Substrate*. Hanzo Industries white paper, 2026 (companion; `hanzo-pq-object-store`).

[10] Hanzo AI. *Edge LLM Inference Across Commodity Accelerators* (companion white paper; source of the prior 154–225 MB `cloud` working-set measurement).

[11] Hanzo AI. *Edge LLM Inference Across Commodity Accelerators: Measured AMD, NVIDIA, and Apple Performance* (companion; source of the cited 8.11 GiB / 231 GB/s / decode-t/s inference figures; `hanzo-cross-backend-inference`).

[12] Lux Industries. *HIP-0116: Plugin VM / In-Process Composition* and the `luxfi/consensus` BFT engine (`engine/bft`, `engine/chain`, `engine/common`). https://github.com/luxfi/consensus.

[13] Lux Industries. *Quasar: Post-Quantum BFT Consensus* (LP-020 / Lux Quasar consensus white paper; `luxfi/consensus/protocol/quasar`). https://github.com/luxfi/consensus.

[14] Confidential computing hardware: NVIDIA H100/H200 Confidential Computing; Intel Trust Domain Extensions (TDX); AMD Secure Encrypted Virtualization — Secure Nested Paging (SEV-SNP). Vendor documentation.

[15] Hanzo AI. *HIP-0005: Post-Quantum Security for AI Infrastructure*. https://github.com/hanzoai/HIPs.

[16] Hanzo AI. *HIP-0001: AI Token — Hanzo's Native Currency*. https://github.com/hanzoai/HIPs.

[17] Hanzo AI. *HIP-0117: BYO Compute Fleet and Metered Billing*. https://github.com/hanzoai/HIPs.

[18] SQLite. *PRAGMA* `cache_size` and *Write-Ahead Logging*. https://www.sqlite.org/pragma.html#pragma_cache_size, https://www.sqlite.org/wal.html.

[19] Istio. *Sidecar resource footprint* (Envoy proxy memory, per-proxy). https://istio.io/latest/docs/ops/deployment/performance-and-scalability/.