zapier.georgelambert.org · Markdown indexes
VERAE TIME × ZAPIER

nats cluster bench

site/packages/zapier-decisions/reports/nats-cluster-bench.md

Progress report · 2026-09-12 · run `20260912T045131Z` (UTC)

This is the full write-up of the test-environment NATS cluster bench: what was measured, how, the numbers, the charts, and what they mean for Verae Time × Zapier. Short tables also live in `verae-nats-cluster/BENCH.md`. Raw logs and CSVs are in that repo under `results/20260912T045131Z/`.

1. Executive summary

The test cluster is three JetStream nodes on private `vmbr1` (LXC 511–513). The bench client is a **fourth** guest (LXC 510), so the numbers are cluster-plus-network, not a process talking to itself on loopback.

Two different systems were measured, on purpose:

System	What it is	What we got
Core NATS	Fire-and-forget pub/sub. No disk, no replica ack.	About **0.75–2.0 million msgs/s** at 128 B, depending on fan-out. At 1 KiB, about **630k msgs/s** and **~616 MB/s** aggregate.
JetStream file, replicas=3	Durable, replicated — **this is what product streams use**.	About **16k durable 128 B pubs/s**, about **13.5k** at 1 KiB. Pull consume keeps up with publish at ~11k msgs/s each side.
Ping delay	One message at a time, publish then wait.	**avg 0.307 ms**, **p99 0.734 ms**, max 2.76 ms (1k × 128 B).
Flood delay	Publishers dump a batch; subscriber drains.	**150–505 ms**. That is **queueing under burst**, not wire time.

For this product: timestamp jobs, job events, webhooks, and archive puts go through JetStream r=3. Plan capacity against **~16k durable msgs/s** on this stand, not the million-msg core numbers. A quiet job-event hop is a fraction of a millisecond. If a mailbox falls behind, delay jumps into hundreds of milliseconds — that is the flood column.

Core NATS is still useful: it is the ceiling for non-durable fan-out on this host, and it shows `vmbr1` and the nats-server processes are not the JetStream bottleneck. JetStream is.

---

2. Why this test exists

The lab cut the test environment over to the three-node cluster. Before treating that cluster as the message fabric for keep, fleet, middleware, billing, and archive workers, we needed:

1. **Throughput at several loads** — one publisher vs many, 128 B vs 1 KiB, core vs durable.
2. **Delay characteristics** — both the quiet path (one message RTT) and the overloaded path (burst into a mailbox).
3. **A client that is not a nats-* server** — otherwise we would be measuring loopback on the broker.

This is a **lab stand on one Proxmox host**, not three metal boxes. It answers "is this cluster in the right order of magnitude for our traffic?" It does not replace a soak test on dedicated disks.

---

3. Topology

```
                  vmbr1  10.10.10.0/24  (not on vmbr0, not public)
                  ------------------------------------------------
LXC 510           LXC 511            LXC 512            LXC 513
verae-px-worker   nats-a             nats-b             nats-c
10.10.10.20       10.10.10.21        10.10.10.22        10.10.10.23
bench client      :4222 client       :4222              :4222
                  :6222 routes       :6222              :6222
                  :8222 loopback     :8222              :8222
```

- Cluster name: `verae`. Each server has two routes to the other two.
- Client URL: `nats://10.10.10.21:4222,nats://10.10.10.22:4222,nats://10.10.10.23:4222`
- HTTP monitor is **loopback :8222** inside each guest. Zapier cloud never talks to NATS.
- Product streams already on this cluster (`ZAPIER_JOBS`, `ZAPIER_EVENTS`, `ZAPIER_WEBHOOKS`, `ZAPIER_USAGE`, `VERAE_ARCHIVE`) use **file** storage and **replicas=3**. The JetStream bench used the same settings on a throwaway stream `benchstream`.
- Host `127.0.0.1:4222` is still listening on NS1; **clients no longer use it**.

Credits for the stack: Scott Lindsey, George Lambert, NATS.IO, Grok-Code.

4. Method

4.1 Tools

Piece	Role
`nats` CLI **0.1.6**	Throughput (`nats bench --no-progress --csv`). Its min/avg/max are **publisher rate spread**, not delay.
`scripts/latency.mjs`	Two connections, header timestamp `t` , delay = receive time − send time.
`scripts/bench.sh`	Runs the ladder from NS1 via `pct exec` on VMID 510.
`scripts/bench-report.py`	Turns logs into the short `BENCH.md` table.

Re-run on NS1, from `verae-nats-cluster`:

```
bash scripts/bench.sh
```

4.2 Load ladder

Core NATS (subject `bench.core.*`):

Run	Publishers	Subscribers	Messages	Payload
`core-1p1s-50k-128`	1	1	50,000	128 B
`core-4p4s-100k-128`	4	4	100,000	128 B
`core-8p8s-200k-128`	8	8	200,000	128 B
`core-4p4s-50k-1k`	4	4	50,000	1024 B

JetStream (`--js --storage file --replicas 3 --stream benchstream`). The stream is deleted between loads so the name never collides:

Run	Shape	Messages	Payload
`js-1p-20k-128-r3`	1 publisher	20,000	128 B
`js-4p-50k-128-r3`	4 publishers	50,000	128 B
`js-4p-20k-1k-r3`	4 publishers	20,000	1024 B
`js-2p2s-20k-128-r3`	2 pub + 2 pull sub	20,000	128 B

Delay (core subjects, two connections):

Run	Mode	Count	Pubs	Payload
`lat-ping-1k-128`	**ping** — publish, wait for that message, repeat	1,000	1	128 B
`lat-1p-5k-128`	**flood** — publish all, then drain	5,000	1	128 B
`lat-4p-10k-128`	flood	10,000	4	128 B
`lat-8p-20k-128`	flood	20,000	8	128 B
`lat-4p-5k-1k`	flood	5,000	4	1024 B

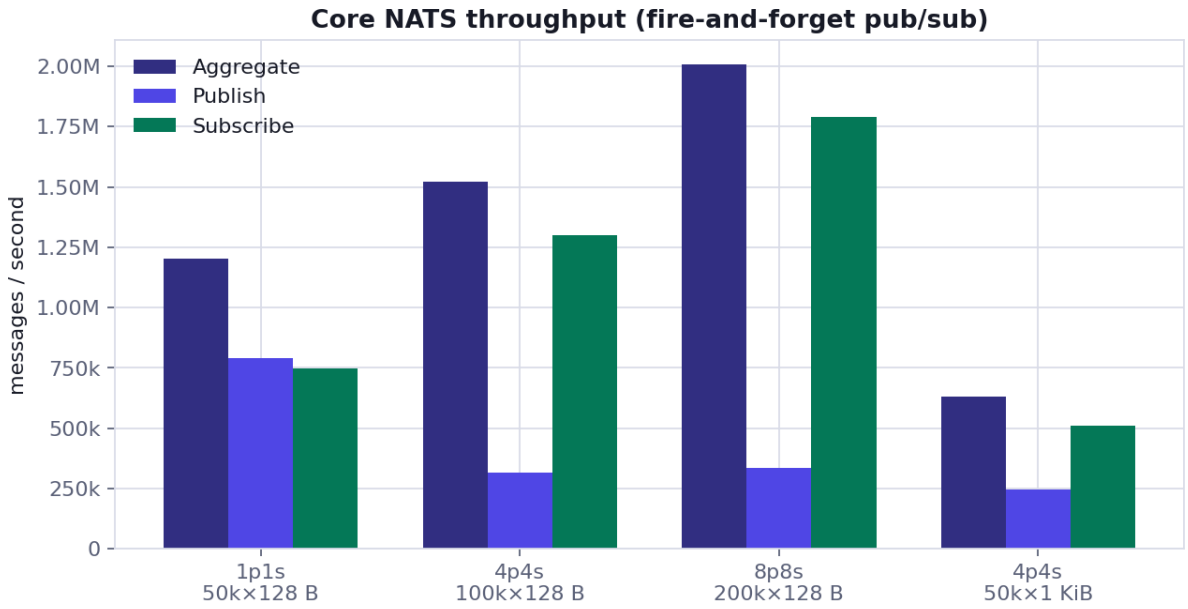
Ping answers “how long does one quiet hop take?” Flood answers “what happens to the last message if we burst N messages into a mailbox?” Those are different questions. Mixing them is how 0.3 ms and 400 ms get confused.

4.3 How to read nats bench columns

- **Pub msgs/s** — rate at which publishers finished their share.
- **Sub msgs/s** — rate at which subscribers finished. With several subscribers on the same subject, core NATS **fans out**, so sub rate can exceed pub rate.
- **Aggregate msgs/s** — nats CLI `NATS Pub/Sub stats` line (pub+sub work in one number). Useful as a headline; do not treat it as “the network carried this many unique messages.”
- Empty JetStream sub cells mean that run was publish-only (durable write, no consumer in the same process).

5. Throughput results

5.1 Core NATS



Core NATS throughput at four loads

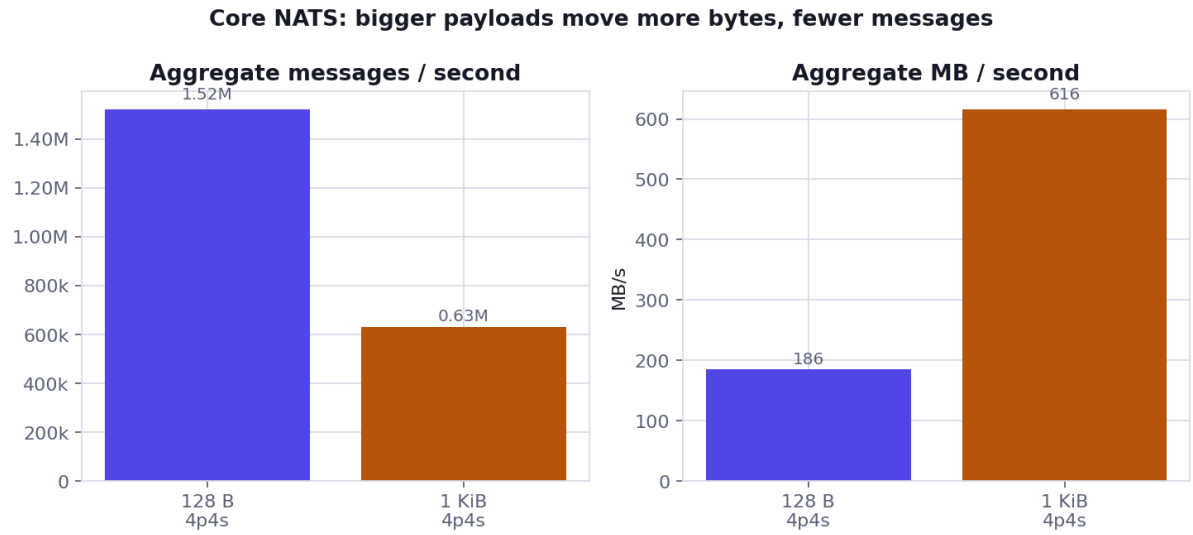
Run	Aggregate msgs/s	Pub msgs/s	Pub MB/s	Sub msgs/s	Sub MB/s
`core-1p1s-50k-128`	1,200,836	791,094	96.57	747,461	91.24
`core-4p4s-100k-128`	1,521,256	316,312	38.61	1,299,634	158.65
`core-8p8s-200k-128`	2,007,937	333,957	40.77	1,790,736	218.60
`core-4p4s-50k-1k`	630,460	247,747	241.94	510,216	498.26

What this chart is saying. Adding subscribers raises **aggregate** and **sub** rates because each published message is delivered to every subscriber. Publish rate does **not** climb the same way: 1 publisher at 128 B already pushes ~791k msgs/s; 4 and 8 publishers sit around 310–335k msgs/s **each process slower**, while fan-out on the sub side goes to 1.3M then 1.8M.

That publisher slowdown is expected on this stand. The four/eight publisher processes and the four/eight subscribers all run **inside one LXC** (510) against three broker LXCs on the **same Proxmox CPU and** `vmbr1`. Per-publisher logs show a wide spread (example, 4p core 128 B: 79k–524k msgs/s among the four pubs). That is CPU scheduling and client-side contention, not a NATS cluster that only has one fast node.

1:1 at 128 B is the cleanest core number: **~791k pub, ~747k sub, ~1.20M aggregate**. The cluster and the bridge can move three-quarter-million small messages per second fire-and-forget from a single client pair.

5.2 Payload size (core)



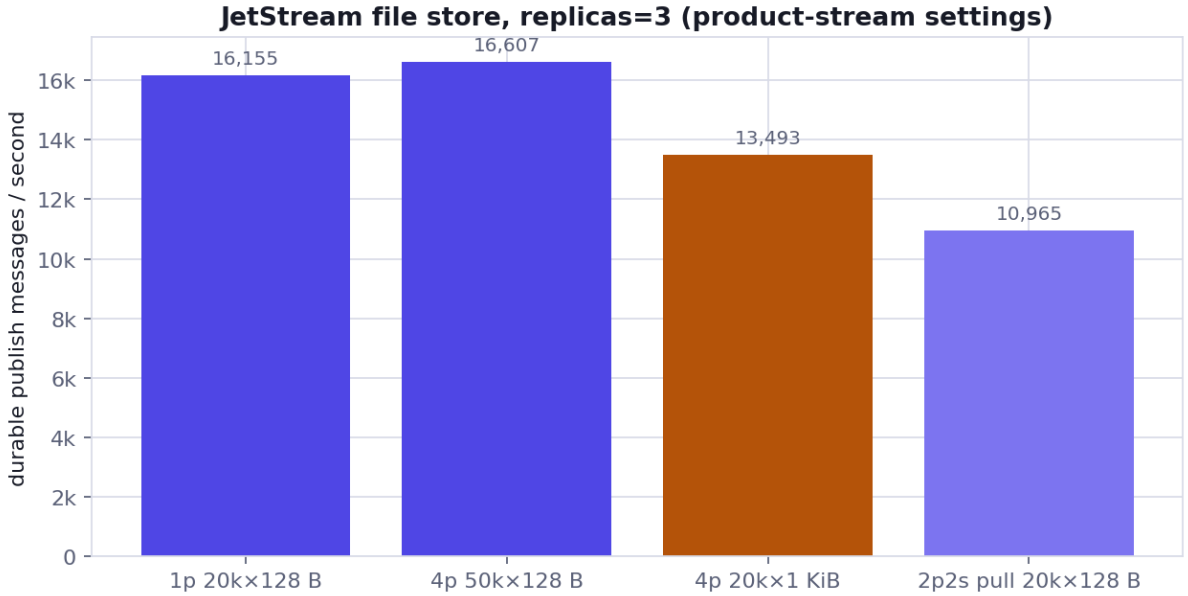
Core NATS 128 B vs 1 KiB

Same 4p4s shape, two sizes:

Payload	Aggregate msgs/s	Aggregate MB/s	Pub msgs/s	Sub msgs/s
128 B	1,521,256	185.70	316,312	1,299,634
1 KiB	630,460	615.68	247,747	510,216

Message rate falls; **byte rate rises** (186 MB/s → 616 MB/s aggregate). We are leaving the “tiny message, CPU/syscall bound” region and entering “copying bytes across `vmbr1`.” Job JSON and archive metadata sit nearer 128 B–1 KiB than megabyte blobs (blobs are HTTP/WORM, not NATS payloads).

5.3 JetStream r=3 file



JetStream durable publish rate

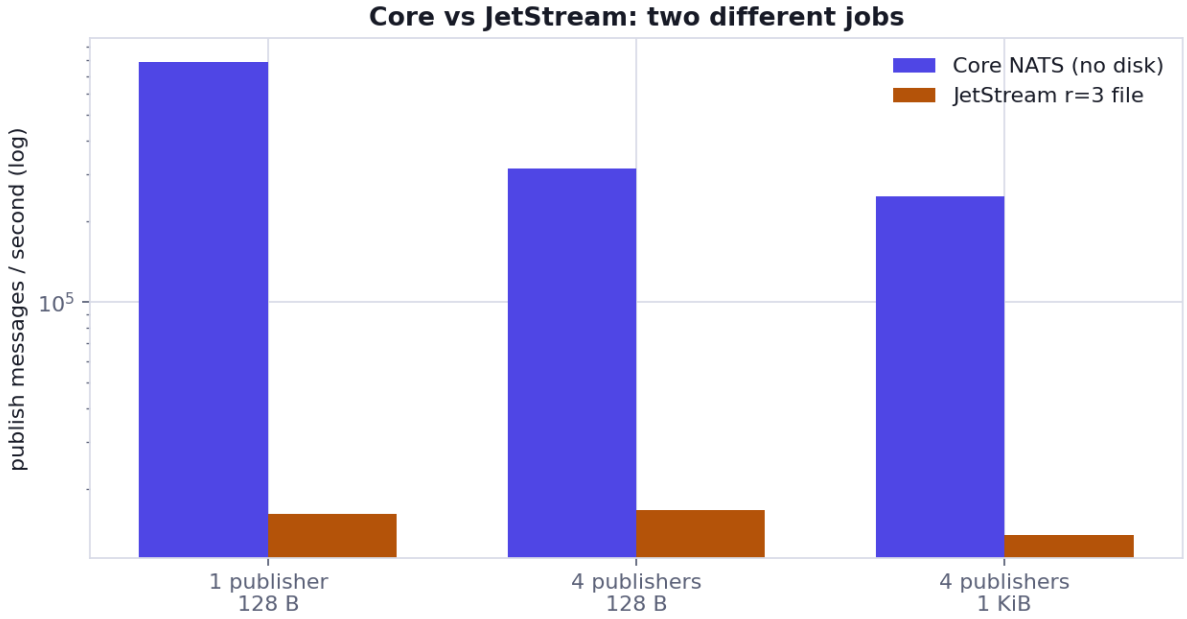
Run	Pub msgs/s	Pub MB/s	Sub msgs/s	Notes
`js-1p-20k-128-r3`	**16,155**	1.97	—	publish-only
`js-4p-50k-128-r3`	**16,607**	2.03	—	four pubs, same ceiling
`js-4p-20k-1k-r3`	**13,493**	13.18	—	1 KiB still disk/replica bound
`js-2p2s-20k-128-r3`	10,965	1.34	10,942	pull consumers keep up

Four publishers do not make JetStream four times faster. 1p and 4p at 128 B are both ~16k msgs/s. The limiter is **synchronous replication to three file-backed replicas**, not client parallelism. That is the result we wanted to see: the bench stream is behaving like a replicated log, not like core fan-out.

Pull consume (`js-2p2s`) is slightly slower on publish (~11k) because the same run is also reading. Pub and sub stay matched (10,965 vs 10,942): the consumer is not the straggler.

1 KiB durable write is ~13.5k msgs/s (~13.2 MB/s). Bytes go up; message rate dips only a little. JetStream here is **ack/fdatasync/replica** bound, not payload-copy bound, in this size range.

5.4 Core vs JetStream (same client, same cluster)



Core vs JetStream publish rate, log scale

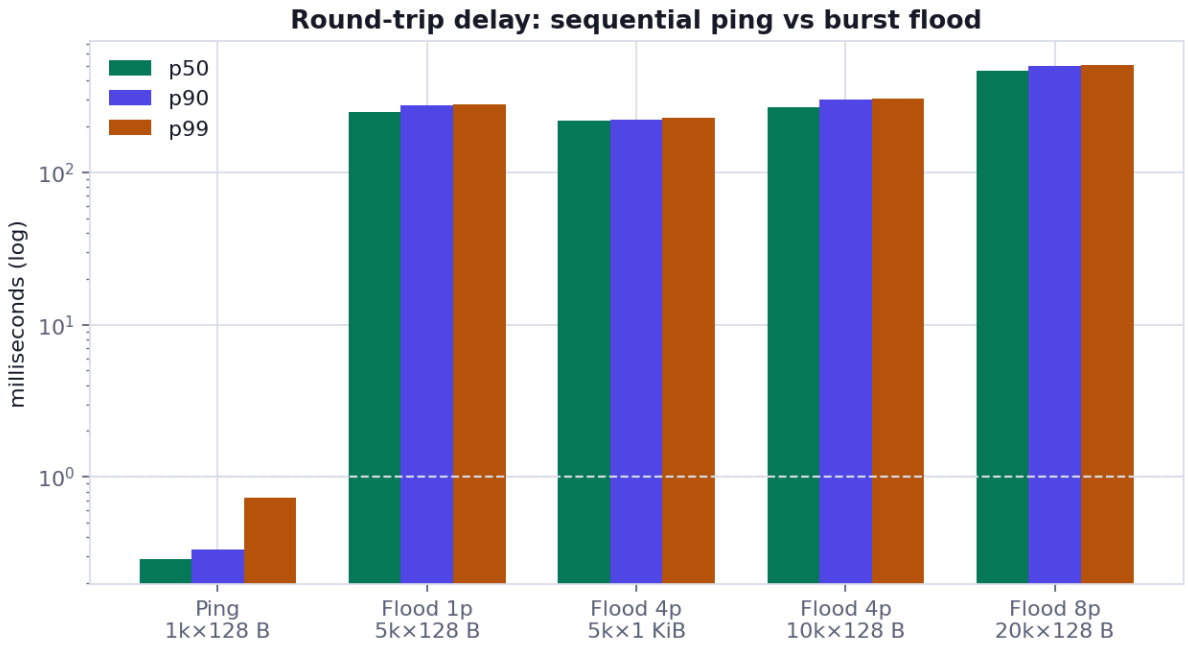
The log scale is required: core publish is **~15–50×** JetStream publish on this stand.

Shape	Core pub msgs/s	JS r=3 file pub msgs/s	Ratio
1 publisher, 128 B	791,094	16,155	~49×
4 publishers, 128 B	316,312	16,607	~19×
4 publishers, 1 KiB	247,747	13,493	~18×

This is not JetStream "losing." Core is allowed to forget a message the instant the server accepts it. JetStream on file with replicas=3 must **record it on a majority** before the publish acks. Our product streams (`ZAPIER_*` , `VERAE_ARCHIVE`) chose that trade on purpose: a job event that survives one LXC dying is worth ~16k msgs/s instead of ~800k.

If we ever need core-like rates for a signal that may drop, that signal should not be on a replicated file stream.

6. Delay results



Delay percentiles, log scale

Run	Kind	Count	min	avg	p50	p90	p99	max
`lat-ping-1k-128`	ping (sequential RTT)	1000	0.254 ms	**0.307 ms**	0.286 ms	0.332 ms	**0.734 ms**	2.763 ms
`lat-1p-5k-128`	flood	5000	149.3 ms	238.6 ms	248.8 ms	274.3 ms	279.4 ms	279.7 ms
`lat-4p-5k-1k`	flood	5000	155.1 ms	211.7 ms	217.6 ms	223.3 ms	227.8 ms	228.4 ms
`lat-4p-10k-128`	flood	10000	174.2 ms	263.2 ms	266.7 ms	299.1 ms	304.2 ms	304.5 ms
`lat-8p-20k-128`	flood	20000	304.6 ms	453.7 ms	466.3 ms	499.9 ms	505.1 ms	505.6 ms

The dashed line on the chart is 1 ms. Only **ping** lives there.

6.1 Ping — the quiet hop

One publisher, one subscriber, two connections, wait for each message before sending the next.

- **min 0.254 ms** — guest → `vmbr1` → a nats-server → `vmbr1` → guest.
- **p50 0.286 ms / avg 0.307 ms** — typical.
- **p99 0.734 ms** — still under a millisecond.
- **max 2.763 ms** — one outlier in 1,000 samples (GC, scheduler, or a slow route). Not the tail we design for.

A middleware `jobs.watch` publish followed by a waiter on `jobs.events` is this shape when the poller is keeping up. Compared with HTTPS to Zapier (tens to hundreds of milliseconds) or a live Verae `GET /api/status/{jobId}`, NATS RTT is noise.

6.2 Flood — queueing under burst

Publishers write the whole batch as fast as they can, then the subscriber drains. Each message's delay is "how long was I in the buffer before the subscriber got to me?"

That is why:

- **min is already ~150–300 ms** — even the first messages wait behind a burst that filled the socket/client queue.
- **p50 ≈ p99 ≈ max** — a queue drain has a tight distribution: everyone waits for roughly the same backlog.
- **8p × 20k is ~450 ms avg** — twice the messages of 4p × 10k, roughly twice the wait. Linear in backlog, not in cluster diameter.

Flood is **not** a measurement of NATS being slow. The ping column proves the hop is ~0.3 ms. Flood is a measurement of **what operators will see if a consumer stalls** (job-events mailbox, webhook deliver, archive reply). Backlog time ≈ `queued_messages / consume_rate`.

6.3 1 KiB flood vs 128 B flood

4 publishers, 5k messages at 1 KiB: avg **212 ms**, slightly **faster** than 4p 10k × 128 B (263 ms) because the **count is half**, even though each message is 8× larger. Again: delay here tracks **how many messages are queued**, not payload size, in this range.

---

7. What this means for Verae × Zapier

Product subjects on this cluster:

Address	Kind	Bench analogue
`verae.zapier.jobs.watch`	work queue (JetStream)	JS durable pub ~16k/s
`verae.zapier.jobs.events`	events	JS + ping if waiters keep up; flood if they do not
`verae.zapier.webhooks.deliver`	work queue	JS durable
`verae.zapier.usage`	optional	JS durable
`verae.billing.*`	request-reply	ping (quiet RTT)
`verae.archive.put` / `query` / `reply.*`	JetStream + broadcast query	JS durable; query fan-out is closer to core but still JS-backed puts

Capacity. 16k durable 128 B pubs/s is **~1.4×10⁹ messages/day** if you could fill the pipe. We will not. Zapier HTTPS, live `api.veraetime.net`, WORM bloom checks, and human Zap runs sit far below that. This cluster is not the product bottleneck on NS1.

Latency budget. A timestamp wait is: HTTP in → NATS watch → poll Verae → NATS event → HTTP out (or REST Hook). The NATS pieces are **sub-millisecond** when caught up. Do not spend time "optimizing NATS RTT" until Zapier/Verae HTTP is in the same band.

Backlogs. The failure mode that *does* show up in these numbers is flood delay. If webhook-deliver or job-events consumers pause (keep stopped, replica floor, a blocked HTTPS post to `hooks.zapier.com`), waiters will see **hundreds of milliseconds to seconds** of queue time. Fleet replica floors and keep exist to prevent that, not because 0.3 ms is too slow.

Hardware move. Same three configs, three boxes, private NIC. Expect:

- Core numbers to change with NIC and CPU (maybe up, maybe down).
- JetStream numbers to change **more**, because they are disk + fsync + replica RTT. Distinct SSDs should help; a slow shared datastore would hurt.
- Ping RTT to grow by whatever the real NIC and switch add (still likely low milliseconds on a LAN).

8. Limits of this measurement

1. **One Proxmox host.** LXC 510–513 share cores, memory, and the host's disk. Replica=3 on file is **three files on the same underlying storage**, not three failure domains. HA of "one disk dies" is **not** proven. HA of "one LXC process dies" is the actual claim.
2. **Short runs.** Tens of thousands of messages, seconds of wall time. No compaction, no multi-hour page-cache eviction, no snapshot/restore during load.
3. **No TLS, no nkeys.** `verae-nats-accounts` is still a sketch. Auth would add CPU; it would not turn 16k into 800k.
4. **One bench client.** All publishers live in 510. A fleet of workers on several CTs might publish more into JetStream until disk/replicas saturate — the 1p vs 4p JS result says that saturation is already ~16k from one CT.
5. **nats 0.1.6** does not report delay. Anyone reading `min | avg | max msgs` on a bench log as microseconds will get the wrong story. Delay is only `latency.mjs`.
6. **Core aggregate ≠ unique messages.** Fan-out double-counts. Use pub or sub columns when comparing to JetStream.
7. **Not a Zapier or Verae API bench.** Those are still blocked on operator login / live credentials.

---

9. How to reproduce

On NS1 (Proxmox), from the `verae-nats-cluster` checkout:

```
bash scripts/status.sh    # 3/3 JetStream
bash scripts/bench.sh     # writes results/<utc>/ and BENCH.md
```

The client VMID defaults to **510**. Override with `CLIENT_VMID=…`. `NATS_URL` comes from `client.env`.

Rebuild this progress report (charts + HTML + PDF) from the monorepo:

```
python3 packages/zapier-decisions/scripts/build-nats-bench-report.py
```

---

10. Appendix — environment and files

Item	Value
Run stamp	`20260912T045131Z`
Client	LXC 510 `verae-px-worker` `10.10.10.20`
Servers	511/512/513 `nats-a/b/c` `10.10.10.21–23`
nats CLI	0.1.6 linux-amd64
JS storage	file, replicas=3, stream `benchstream` (deleted between loads)
Isolation	`vmbr1` only; no `0.0.0.0` client bind
Short tables	BENCH.md
Raw logs	`packages/verae-nats-cluster/results/20260912T045131Z/`
This report	`packages/zapier-decisions/reports/nats-cluster-bench.{md,html,pdf}`

Publisher rate spread (nats CLI, msgs/s, **not** delay):

Run	min	avg	max
core-4p4s-100k-128 pub	79,260	257,805	524,453
core-8p8s-200k-128 pub	41,744	71,488	152,536
core-4p4s-50k-1k pub	61,936	110,271	176,262
js-4p-50k-128-r3 pub	4,154	5,176	6,628
js-4p-20k-1k-r3 pub	3,373	4,063	5,121
js-2p2s-20k-128-r3 pub	5,485	7,081	8,678

Wide core spreads are the single-client-CT effect described in §5.1. JetStream spreads are narrow and low — every publisher is waiting on the same replicated write path.