WHITEPAPER · BENCHMARK STUDY · GCP EUROPE-WEST3 · JULY 2026

Cloud Run as Pipeline Compute: What 140 Measured Runs Say About Latency, Throughput, and Sinks

Two benchmark rounds placed Google Cloud Run between real sources and real sinks — an HTTP API in front of Cloud Storage, Pub/Sub push into BigQuery, and Kafka consumers feeding BigQuery, Iceberg lakehouses, and Pub/Sub — in Go, Python, and Rust. This whitepaper condenses ~140 measured runs into the decisions that matter: where latency actually comes from, what throughput costs, when the open lakehouse is worth its toll, and how to configure Cloud Run so the platform works with you rather than against you.

8 ms → 96 s
Per-write cost spread across sinks, same service body (same-clock p50)

50,000 rows/s
Sustained into BigQuery via Storage Write API, zero backlog

300×
One instance's throughput gain from batching alone (50 → 15,000 msg/s)

≈ $21–29
Total cloud spend for both studies, all resources torn down

Executive summary

- **Your sink sets your latency floor, not your language or your hardware.** The identical consumer body pays ~8 ms per BigQuery `insertAll` row, ~40-50 ms per Cloud Storage or batched write, ~2 s per external-Iceberg commit, and 71-96 s per queued BigQuery DML statement under load. Meanwhile a 50× traffic sweep, an 80× concurrency sweep, and a CPU/RAM doubling moved the median by single-digit milliseconds.
- **Language choice is a configuration problem in disguise.** Go, Python, and Rust tie wherever the hot loop is a C-backed client library or the sink dominates; Python even won the streaming-consumer scenario end-to-end. Its one collapse (p50 9.4 s at 1,500 msg/s) came from blocking I/O × concurrency 80 on one event loop — and was fully repaired by batching (25 ms) or by capping concurrency (88 ms).
- **Throughput is bought with batching, not hardware.** Per-message synchronous writes cap near 50 msg/s per instance. 500-row/200 ms batches lift a single 1-vCPU instance to 15,000 msg/s with zero backlog; the Storage Write API sustained 49,930 rows/s across four instances per language. Doubling CPU bought 1.7×; batching bought 300×.
- **Managed Iceberg is free; external Iceberg is a distributed-systems exam.** BigQuery managed Iceberg tables matched native tables on writes and reads. External Iceberg (Parquet on GCS, BigLake REST catalog) has a ~2 s commit floor and optimistic-concurrency conflicts that partitioning does not fix — it demands single-writer funnels with large commits, or scheduled Spark batches (~38k rows/s).
- **Cloud Run is request-shaped; respect that.** It will not autoscale Kafka consumers (pin instances or use Worker Pools), rolling restarts pause a consumer group 24-28 s, cold starts add 0.4-3.5 s unless min-instances ≥ 1, and at-least-once delivery produced 9 duplicates in 240k messages under instance kill

— with zero loss.

> **The one-sentence takeaway:** spend engineering effort in this order — sink client first (batching, connection reuse, write API choice), concurrency second, language last; and let measured write-cost per sink, not platform defaults, drive the architecture.

1 · What was tested, and how

Both rounds share one shape: a load generator with known send timestamps, Cloud Run in the middle, and a sink whose contents can be counted and timed afterward. Round 1 (June-July 2026 codebase, ~60 runs) varied the source and the language across three pipelines. Round 2 (~80 runs) fixed the source — Kafka, 12 partitions, 1 KB Avro — and varied the sink across six write paths, in Go and Python, after an external review added binding measurement amendments.

Round	Pipeline	Variables	Scale
1	HTTP (vegeta) → Cloud Run → Cloud Storage	3 languages · rate 10–500 req/s · payload 1 KB–1 MB · concurrency 1–80 · CPU/RAM · sync vs async writes	17 scenarios × 3 languages
1	Pub/Sub push → Cloud Run → BigQuery insertAll	sync vs 500-row/200 ms batch · 100–3,000 msg/s · concurrency	7 scenarios × 3 languages
1	Kafka → Cloud Run → Pub/Sub	pinned instances 1/2/4 · 500–5,000 msg/s · 1 KB/100 KB Avro · cold starts	6 scenarios × 3 languages + 12 cold-start trials
2	Kafka → Cloud Run → six sink paths	BQ insertAll (sync/batch) · Storage Write API · managed Iceberg (insertAll/DML) · external Iceberg (pyiceberg/iceberg-go) · Pub/Sub→BQ subscription (push + pull) · Worker Pools · Dataproc Spark reference	~80 gated runs × 2 languages

Measurement discipline (round 2)

Round 2 ran under five binding amendments produced by an independent pre-execution review: **(A1)** the primary latency metric is consume→sink-ack, both timestamps from the same instance clock — immune to cross-machine clock skew — with cross-clock produce→consume numbers reported separately; **(A2)** discovery-set rate ceilings per sink, a 300 s drain timeout, and mandatory per-run accounting of undrained backlog, commit conflicts, retries, failed batches, and duplicates; **(A3)** a consumer-group health gate (12/12 partitions assigned plus flowing canary messages) before every measured window; **(A4)** the managed BigQuery-subscription path reported as visibility curves, not millisecond claims; **(A5)** a tested teardown script and a hard spend stop. Aggregated datasets regenerate byte-identically from the raw per-run JSON in both rounds.

Environment: GCP project in europe-west3, all resources co-located; Cloud Run gen2, 1 vCPU / 1 GiB baseline; Go 1.25, Python 3.12 (FastAPI/uvicorn, single worker), Rust 1.88 (round 1 only); single-broker Kafka 3.9 (KRaft) on e2-medium; load generator on e2-standard-4. Total spend: ≈$12–16 (round 1) + ≈$9–13 (round 2).

2 · Latency: the sink is the floor

Round 1 established the negative result that saves the most tuning effort: against a Cloud Storage sink, p50 stayed at 51–63 ms for all three languages across a 50× rate sweep (10→500 req/s), concurrency 1→80,

and a doubling of CPU and memory. A dedicated latency-optimized configuration (min-instances, startup CPU boost, tuned concurrency) bought nothing — because none of it touches the ~35–50 ms server-side GCS write that dominates the request. Reads ran 28–32 ms; 1 MB writes 71–78 ms. The object, not the platform, sets the number.

Round 2 then priced the floors with the skew-free instrument — the same Kafka consumer body, swapping only the write call:

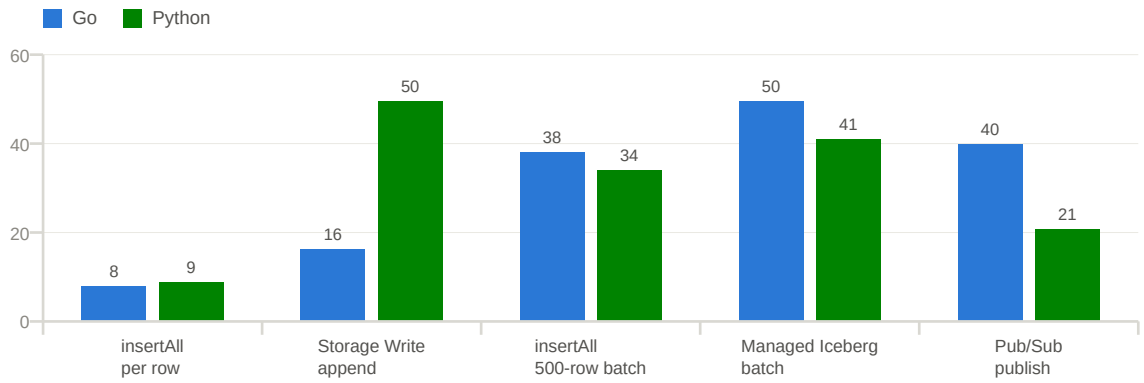

Figure 1 — What one write costs, by sink (round 2, same-clock sink-write p50, single instance at 2,000 msg/s). The chart shows the millisecond-class sinks; external-Iceberg commits (1.9–2.8 s) and overloaded BigQuery DML (71–96 s per statement) are off-scale and appear in the table below. The Storage Write API gap (Go 16 ms vs Python 50 ms) is the one real language difference round 2 found — a client-library maturity gap, pre-registered before the run.

Sink path	Go p50	Python p50	Unit
BigQuery insertAll, per row (sync)	7.9 ms	8.8 ms	per row
BigQuery Storage Write API	16.3 ms	49.5 ms	per batch append
BigQuery insertAll, 500-row batch	38.0 ms	34.0 ms	per batch
Managed Iceberg, insertAll batch	49.6 ms	41.0 ms	per batch
Pub/Sub publish (batched)	39.9 ms	20.8 ms	per batch
External Iceberg commit	1.92 s	2.79 s	per commit
BigQuery DML INSERT (overloaded)	71.0 s	95.8 s	per statement (queue time)

If the caller's response time matters more than durability, the floor can be dodged instead of lowered: round 1's async mode acknowledged first and wrote to GCS in the background, cutting caller-visible p50 from ~54 ms to 7–10 ms. The price is real: an at-least-once-after-ack durability window, mandatory bounded-queue backpressure (Python shed ~18% at 500 req/s on its single event loop), and semantics that suit telemetry but not payments.

3 · Languages: one collapse, two cheap repairs

Across ~140 runs there is exactly one scenario family where a language failed: Python's synchronous BigQuery path under Pub/Sub push at concurrency 80 — eighty concurrent requests multiplexed onto one uvicorn event loop, each blocking on an insertAll call.

Publish rate	Go p50	Python p50	Rust p50	Python, conc 10	Python, batched
100 msg/s	133 ms	800 ms	129 ms	—	—
500 msg/s	20 ms	779 ms	20 ms	88 ms	—
1,500 msg/s	22 ms	9,413 ms	22 ms	—	22 ms

Publish rate	Go p50	Python p50	Rust p50	Python, conc 10	Python, batched
3,000 msg/s (batch)	23 ms	25 ms	24 ms	—	25 ms

Table 2 — Round 1, Pub/Sub → BigQuery, publish→row-visible p50. At 1,500 msg/s Python also delivered ~15% fewer rows in-window. Both repairs are configuration, not rewrites: concurrency 10 gives the event loop room (88 ms); 500-row/200 ms batches erase the gap entirely at double the load.

Everywhere else the languages converged — or inverted expectations. In round 1's streaming-consumer pipeline (Kafka → Pub/Sub), Python was the fastest end-to-end (28 ms vs Go 35 ms and Rust 47 ms at 500 msg/s) at identical throughput ceilings: its hot loop is librdkafka, fastavro, and C-backed gRPC, and Rust lost its margin to a slower REST-batching Pub/Sub client — a sink-client effect, not a language-speed effect. In round 2, Go and Python tied within noise on five of six sinks; the exception was the Storage Write API client gap noted above. Cold starts remain the one clean compiled-language win (Section 6).

> **Rule extracted:** judge a language by what is inside its hot loop, not by its label. C-backed clients and pinned instances make Python fully competitive; many concurrent synchronous requests per instance make it the worst of the three unless concurrency is capped, the sink is batched, or multiple workers are run.

4 · Throughput: a batching ladder, not a hardware ladder

The measured ceilings span five orders of magnitude, and every rung is a write-strategy decision:

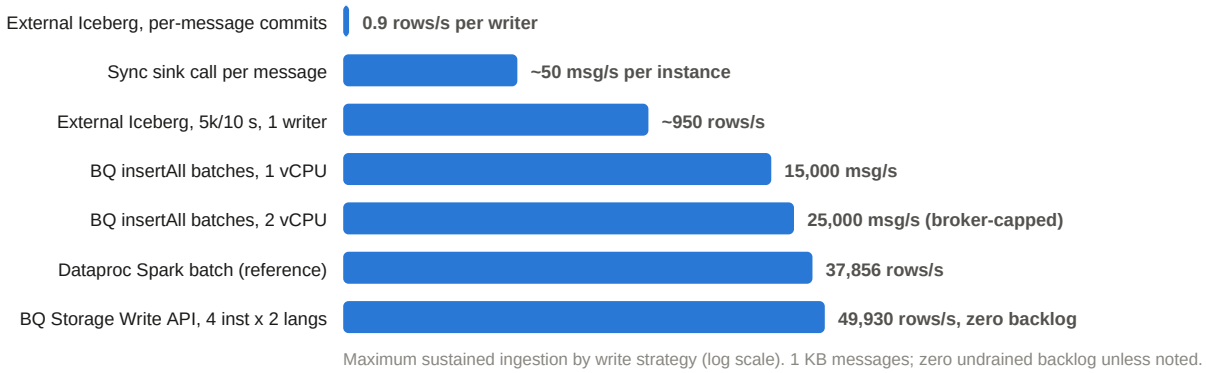

Figure 2 — Sustained ingestion by write strategy (round 2). “Broker-capped”: the single e2-medium Kafka broker saturated before the 2-vCPU consumer did; 25,000 msg/s is a lower bound. The Storage Write figure is the two-language total (24,965 msg/s each, in parallel), verified by counting 5,991,840 rows in BigQuery against sent messages with zero undrained backlog.

Three points deserve emphasis. First, the bottom rung is catastrophic rather than slow — per-message commits into external Iceberg delivered 1.8% of expected rows. Second, the middle rungs are nearly free: moving from per-message writes to 500-row/200 ms batches multiplies one instance's ceiling ~300× with no new infrastructure, while doubling CPU bought only 1.7×. Third, none of this scales itself for pull-based consumers: Cloud Run's autoscaler watches HTTP, not consumer lag, so Kafka consumers only performed when pinned (min = max instances, CPU always allocated) — or deployed as Worker Pools, which measured identically to pinned services and are the cleaner primitive (no HTTP shim). Google now also ships a lag-based Kafka autoscaler for Worker Pools; it postdates these runs and is the obvious next thing to measure.

5 · The lakehouse toll booth

“Iceberg” is one word for two very different write paths. **BigQuery managed Iceberg tables are a free choice:** writes go through the same insertAll client at 41–50 ms per batch (vs 34–38 ms native) and read benchmarks landed in the same class as native tables — pick the open format for interoperability and pay approximately nothing. The one trap is writing them with per-event DML statements, which queue behind a concurrency quota measured in tens and collapse to 71–96 s per statement.

External Iceberg — Parquet on GCS behind a BigLake REST catalog, written by pyiceberg / iceberg-go — is a different sport. Every commit costs ~2–3 s regardless of size, and concurrent writers contend optimistically on the table's metadata pointer:

Configuration (rate)	Delivered in window	Commit conflicts
Per-message commits, 1 inst (50 msg/s)	1.8%	75
1k rows / 5 s, 1 inst (2,000 msg/s)	3.6%	1,621
1k rows / 5 s, 4 inst (2,000 msg/s)	5.9%	464
1k / 5 s, 4 inst, partitioned (2,000 msg/s)	4.0%	450
5k rows / 10 s, 1 inst (2,000 msg/s)	49.5%	212
5k rows / 10 s, 1 inst (500 msg/s)	95.3%	35

Table 3 — External Iceberg delivery vs commit strategy (round 2; expected rows = sent × 2 languages sharing one table). Partitioning did not reduce contention (450 vs 464 conflicts) because commits race on the table metadata pointer, not on data files.

The recipes that fall out of the data: for streaming, funnel to a single committing writer (conflicts ≈ 0), commit 5,000+ rows every 5–10 s, accept ~10–15 s sink-ack latency, and schedule compaction. For volume, do not stream at all — one Dataproc Serverless Spark batch wrote 9.79M rows in 258.6 s (~37,856 rows/s) with 79 s of provisioning, outrunning every streaming-writer configuration into the same table with zero conflict management.

6 · Operational physics

Cold starts

First-request latency after scale-to-zero (round 1, n = 2 per cell): Go 0.49–0.59 s; Python 1.6–3.5 s; Rust 0.36–0.53 s — except that every Rust first request failed (4/4, HTTP 500) because its hand-rolled metadata-token fetch had no retry, where the Go/Python SDKs retry transparently. Startup CPU boost did not change the ranking. The boring fixes work: retry startup dependencies, and set min-instances ≥ 1 on latency-sensitive paths — which also removed the only p99 anomaly in the round-1 HTTP ladder (a 1,029 ms scale-up spike at 500 req/s).

Redeploys pause consumers

A rolling restart of a live consumer group paused consumption for 27.7 s (eager assignor) vs 23.7 s (cooperative-sticky) — a pause, not a slowdown; between pauses consume spacing was unchanged. Graceful shutdown (commit offsets on SIGTERM) matters more than assignor choice. Separately, the nastiest observed failure mode: after one redeploy, librdkafka wedged mid-rebalance holding zero partitions and logging nothing, at half throughput — the reason round 2 gated every run on “12/12 partitions assigned + canaries flowing.” Monitor consumer-group assignment and lag yourself; Cloud Run will not.

Failure semantics, measured

Killing 1 of 2 consumers mid-run at 2,000 msg/s (at-least-once inserts, 500-row batches, 500 ms offset commits) produced 9 duplicate rows and 0 lost messages out of 240,018 (0.0019% duplication). Plan for

idempotency keys downstream, not for loss. Delivery counts also routinely exceed sent counts slightly under Pub/Sub redelivery — treat every count as at-least-once.

Gotchas that cost real debugging time

- `--vpc-egress=all-traffic` without Cloud NAT silently breaks Google API access from the service.
- A single-broker Kafka VM will fill its disk mid-benchmark unless retention is bounded (fixed at 4 GB / 1 h here).
- Pinned min = max instances with CPU always allocated bill ~$24.50/month each (1 vCPU / 1 GiB) — a subscription, traffic or not.

7 · Cost: Cloud Run is rarely the expensive line item

Path	Cloud Run compute / 1M	Dominant other cost
HTTP API → GCS (conc 80, 55 ms)	≈ $0.42	GCS Class-A writes ≈ $5.00 / 1M
Pub/Sub push → BQ insertAll	≈ $0.42	Pub/Sub throughput ($40 / TiB)
Kafka → BQ insertAll batch (15k msg/s/inst)	≈ $0.05–0.10	≈ nothing (ingestion free tier)
Kafka → BQ Storage Write API	≈ $0.05–0.10	> 2 TiB/mo: ~$25 / TiB
Kafka → external Iceberg (commit-bound)	≈ $1.50–6.00	GCS ops + compaction jobs
Kafka → Pub/Sub → BQ subscription	≈ $0.05–0.10	+ ~$0.04 / 1M Pub/Sub (1 KB)
Dataproc Spark → Iceberg (batch)	—	≈ $0.10–0.30 / 100M rows (DCU-h)

Two structural rules. **Concurrency is a discount:** at concurrency 80 vs 10, I/O-bound requests share instances and per-request compute drops ~30% at identical latency. **Pinning is a subscription:** the request-driven billing story inverts for pull-based consumers — four pinned instances ≈ $98/month whether or not traffic flows, which is GKE-shaped economics at small absolute numbers. Steady high-volume streams eventually favor GKE; bursty or idle streams favor Cloud Run.

8 · Recommendations by requirement

You need	Build	Measured result
Lowest-latency HTTP API	Go/Rust, sync write to a fast sink, min-instances=1, concurrency 10–20	p50 ≈ 20 ms (BQ-class sink); ~52 ms if GCS
Lowest latency into analytics	Kafka → insertAll 500-row/200 ms batches on a Worker Pool, instances = rate ÷ 10k	e2e p50 ≈ 15–20 ms at 5,000 msg/s
Maximum throughput into BigQuery	Storage Write API, 2 vCPU, 4 pinned instances per language (Go client)	49,930 rows/s, 0 backlog, p50 35 ms
Open lakehouse, streaming	External Iceberg via 1 committing writer, 5k-row/10 s commits + compaction cron	95% in-window at 500 msg/s; conflicts ≈ 0
Open lakehouse, volume	Dataproc Serverless Spark micro-batches every 5–15 min	~37,856 rows/s per job
Zero-ops decoupling	Kafka → Pub/Sub → BigQuery subscription (managed delivery)	+25–34 ms ingress p50, zero sink code
Open format without the pain	BigQuery managed Iceberg, same insertAll client (never per-event DML)	41–50 ms/batch ≈ native; reads ≈ native

> **Default architecture** for “events into analytics on Cloud Run”: Worker Pool consumers, pinned to rate ÷ 10k instances, 500-row/200 ms insertAll batches, at-least-once with downstream dedup keys, cooperative-sticky assignor, offsets committed on SIGTERM, consumer-lag alerting — and the Storage Write API swapped in once sustained volume passes ~5k msg/s.

9 · Limitations

- Single region (europe-west3), single month (July 2026), all resources co-located. Cross-region latencies will differ substantially.
- Kafka was one e2-medium broker — a protocol-realistic harness, not a capacity statement; consumer ceilings quoted are broker-capped lower bounds.
- Python ran a single uvicorn worker throughout round 1; its collapse and async shedding are floors that multi-worker configs would likely raise.
- Cold-start samples are small (n = 2 per language × boost setting); round 2 deliberately excluded Rust and cold starts.
- Round 1's batch mode acknowledges before sink durability; round 2's same-clock sink-ack metric closes exactly that gap.
- Cross-machine timestamps carry NTP-level uncertainty; round 2's primary metric avoids cross-clock arithmetic entirely, and its clock-probe offset estimate (~65 ms ≈ probe RTT) is a measurement artifact, not real skew.
- The managed BigQuery-subscription path is visibility-polled every 5 s and is reported as delivery-lag curves, not millisecond latencies.

All raw per-run JSON, service source, orchestration code, per-round HTML reports, and an interactive cross-round writeup are public in the artifact repository: `github.com/ben-j-herbertz/cloudrun-latency-throughput-benchmark`. Reports regenerate byte-identically from the raw results. Experiments were executed autonomously by an AI coding agent under human direction; the measurement plan was independently reviewed (amendments A1–A5) and the results independently validated by a second AI system, with every headline number recomputed from raw run artifacts.