Sink-Bound: An Empirical Study of Google Cloud Run as the Compute Layer in Streaming Data Pipelines

Ben Jonas Herbertz

Independent · Technical Report TR-2026-01 · July 20, 2026

Experiments executed autonomously by AI coding agents under human direction; see Acknowledgments.

Abstract

Serverless container platforms are marketed and studied primarily as request/response compute, yet a large share of production deployments place them in the middle of streaming pipelines: consuming from queues and logs and writing to storage, warehouses, and lakehouses. We report a two-round empirical study (~140 measured runs, June–July 2026) of Google Cloud Run as pipeline compute. Round 1 varies source protocol and implementation language (Go, Python, Rust) across three pipelines (HTTP→Cloud Storage, Pub/Sub→BigQuery, Kafka→Pub/Sub); round 2 fixes a Kafka source and varies the sink across six write paths, including BigQuery insertAll and Storage Write API, BigQuery-managed Apache Iceberg, external Iceberg via a BigLake REST catalog, and a fully managed Pub/Sub→BigQuery delivery path. Round 2 adopts a skew-free, same-clock primary latency metric (consume→sink-acknowledge) and per-run validity gates covering consumer-group health, drain timeouts, and loss/duplicate accounting. We find the workload is overwhelmingly *sink-bound*: median latency is invariant to a 50× load sweep, an 80× concurrency sweep, and a CPU/RAM doubling, while per-write cost spans four orders of magnitude across sinks (7.9 ms to 95.8 s). Language effects are configuration effects in disguise: one collapse (Python, synchronous inserts at concurrency 80; p50 9.4 s) is fully repaired by batching (25 ms) or concurrency capping (88 ms), and Python wins the streaming-consumer scenario outright. Batching, not hardware, buys throughput: 500-row batches lift a single 1-vCPU instance ~300× to 15,000 msg/s, and the Storage Write API sustains 49,930 rows/s with zero backlog. External Iceberg exhibits a ~2 s commit floor and optimistic-concurrency conflicts that table partitioning does not mitigate. We quantify cold starts, rebalance pauses (24–28 s), and at-least-once duplication under failure injection (9 duplicates per 240,018 messages, zero loss), and derive configuration guidance and cost models. All raw results and code are public.

Keywords: serverless computing, Cloud Run, benchmarking, stream processing, BigQuery, Apache Iceberg, Kafka, lakehouse

1 Introduction

Serverless platforms have been characterized extensively at the function level — cold starts, instance lifecycles, and microbenchmark performance [1, 2, 3] — and the research community has articulated both the promise [4] and the data-systems limitations [5] of the model. Much less attention has been paid to the way serverless *containers* are actually deployed in data-intensive systems: as the compute layer between a source (an API edge, a message bus, a Kafka topic) and a managed sink (object storage, a warehouse, a lakehouse table). In that position, the platform's own performance is only one term in a sum that includes the sink client, the write API, batching strategy, and delivery semantics.

This paper asks a deliberately practical question: *for a realistic set of source→compute→sink pipelines, which decisions actually move latency, throughput, and cost on Google Cloud Run?* We answer it with two benchmark rounds run in June–July 2026 on a production GCP project (europe-west3), totaling ~140 measured runs and ≈$21–29 of cloud spend.

Our contributions:

- **C1 — A sink-inclusive benchmark design.** End-to-end pipelines with counted, timestamped sink contents, rather than isolated platform microbenchmarks; six distinct sink write paths under one fixed Kafka source.
- **C2 — A skew-free primary metric.** Round 2 measures consume→sink-acknowledge from a single instance clock, eliminating cross-machine clock error from headline latencies, with cross-clock metrics reported separately.
- **C3 — Validity gates as first-class protocol.** Per-run consumer-group health gates, drain timeouts with undrained-backlog accounting, and duplicate/loss counting; runs that fail to keep up are results, not retries.
- **C4 — Quantified findings** on latency invariance, the per-write cost ladder, the batching ladder, external-Iceberg commit contention, cold starts, rebalance pauses, and failure semantics.
- **C5 — A public artifact:** all raw per-run JSON, service source in three languages, orchestration, and reports that regenerate byte-identically [13].

2 Related Work

Wang et al. [1] systematically measured the resource management of AWS Lambda, Azure Functions, and Google Cloud Functions; Shahrad et al. [2] characterized production FaaS workloads and motivated keep-alive policies; Copik et al. [3] built a portable FaaS benchmark suite. These target function platforms and platform-internal behavior. Our study complements them along two axes: it measures a serverless *container* platform (request-concurrent, gen2 Cloud Run) and it treats the managed sink — GCS, BigQuery's two write APIs, Iceberg catalogs, Pub/Sub — as part of the system under test, which turns out to dominate outcomes.

Hellerstein et al. [5] argued that first-generation serverless is a poor match for data-centric systems, in part because of communication through slow storage; our results give that argument current, quantitative form (a ~2 s commit floor and conflict-bound concurrency for external Iceberg; 71–96 s queued DML), while also showing where the mismatch has closed (49,930 rows/s into BigQuery from pinned serverless containers). Armbrust et al. [7] define the lakehouse architecture our round 2 targets; Kafka [8] and Apache Iceberg [9] are the substrate technologies. Industry measurements exist for fragments of this space — cross-cloud Pub/Sub latency [11], Cloud Run vs. Lambda HTTP benchmarks [12], and Google's own guidance on Kafka autoscaling for Cloud Run worker pools [10] — but we are not aware of a public, loss-accounted, multi-sink comparison of this shape.

3 Experimental Design

3.1 Environment

All experiments ran in one GCP project, region europe-west3, all resources co-located. Cloud Run gen2 with a 1 vCPU / 1 GiB baseline (2 vCPU / 2 GiB variants where stated). Implementations: Go 1.25 (net/http), Python 3.12 (FastAPI, uvicorn, single worker), Rust 1.88 (axum/tokio; round 1 only). Kafka 3.9 (KRaft) on a single e2-medium VM, 12 partitions, Avro binary payloads (1 KB default, 100 KB variants); load generation and collection on an e2-standard-4 VM. HTTP load used vegeta 12.12; Pub/Sub and Kafka producers were custom, rate-controlled, and reported achieved rates. All cloud resources were created for the study and torn down afterward.

3.2 Round 1: vary source and language

Three pipelines, each implemented identically in Go, Python, and Rust: (A) HTTP→Cloud Run→Cloud Storage with synchronous and asynchronous (acknowledge-then-write) modes, payloads 1 KB–1 MB, offered rates 10–500 req/s, concurrency 1–80, and CPU/RAM variants; (B) Pub/Sub push→Cloud Run→BigQuery `tabledata.insertAll`, synchronous per-message vs. 500-row/200 ms batches, 100–3,000 msg/s; (C) Kafka→Cloud Run→Pub/Sub background consumers with pinned instances (1/2/4), 500–5,000 msg/s, plus a scale-from-zero cold-start study (two trials per language × startup-CPU-boost setting). Latency for (A) is client-side; for (B) publish→row-visible; for (C) produce→pulled-from-sink, stamped by the load-generator VM (same clock at both ends).

3.3 Round 2: fix the source, vary the sink

One Kafka source (1 KB Avro, 12 partitions) consumed by Go and Python services writing to six sink paths: S1 BigQuery insertAll (sync and batch); S1b BigQuery Storage Write API (gRPC append streams); S2 BigQuery-managed Apache Iceberg tables (insertAll and DML write paths); S3 external Iceberg — Parquet on GCS behind a BigLake metastore REST catalog, written by pyiceberg (Python) and iceberg-go — under a commit-size × flush-interval × writer-count sweep including a partitioned-table variant; S4 Kafka→Pub/Sub→managed BigQuery subscription (plus S4b, a Pub/Sub pull consumer writing BigQuery directly); S5 the S1 workload deployed as a Cloud Run Worker Pool. A Dataproc Serverless Spark batch into the S3 table serves as a non-streaming reference, and two operational experiments — mid-run instance kill, and rolling restart under eager vs. cooperative-sticky assignors — probe failure and rebalance behavior.

3.4 Metrics and validity protocol

Round 2's primary latency metric is **sink-write latency**: $t_{ack} - t_{consume}$, both stamped by the same instance immediately around the sink operation, aggregated from per-batch samples — a quantity immune to cross-machine clock skew. Cross-clock metrics (produce→consume; produce→sink-ack) are reported separately and labeled as skew-bearing. Five pre-registered protocol amendments (A1–A5), produced by an independent review of the experiment plan before execution, additionally required: per-sink rate ceilings established in a discovery phase; a 300 s post-production drain timeout with `undrained_backlog`, commit-conflict, retry, failed-batch, and duplicate counts as mandatory result fields; a consumer-group health gate (12/12 partitions assigned and canary messages flowing) before every measured window; visibility-curve (not millisecond) reporting for the polled managed-delivery path; and scripted teardown with a hard spend stop. Producer achieved rates are recorded per run; delivered rows are counted in the sink and reconciled against sent counts and distinct message identifiers. Aggregated datasets regenerate byte-identically from raw per-run JSON.

4 Results: Round 1

4.1 Latency is sink-bound and configuration-invariant

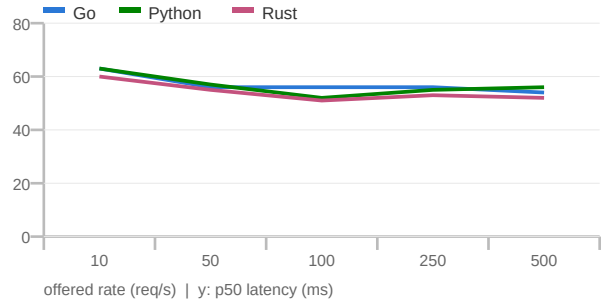

Figure 1. Pipeline A (HTTP→GCS, 1 KB synchronous writes): p50 vs. offered rate per language. The band 51–63 ms holds across a 50× rate sweep.

Median latency in pipeline A was statistically flat across every platform knob we turned: offered rate 10→500 req/s (51.5–63 ms), concurrency 1/10/40/80 (51–59 ms), and 1 vCPU/512 MiB vs. 2 vCPU/2 GiB (no change). A dedicated latency-optimized deployment (min-instances=1, startup CPU boost, tuned concurrency) did not improve the median. The explanation is arithmetic: the server-side GCS write costs ~35–50 ms and dominates the request. Consistently, reads ran 28–32 ms and 1 MB writes 71–78 ms. The single tail anomaly in the sweep — Go p99 of 1,029 ms at 500 req/s with 99.3% success — was mid-run scale-up from min-instances=0, and disappears with a warm floor instance.

4.2 Language effects are configuration effects

Rate (msg/s)	Go	Python	Rust	Py conc10	Py batch
100	133	800	129	—	—
500	20	779	20	88	—
1,500	22	9,413	22	—	22
3,000 (batch)	23	25	24	—	25

Table 1. Pipeline B (Pub/Sub→BigQuery, sync insertAll unless noted): publish→row-visible p50 in ms, concurrency 80. Python's collapse at high rate is repaired by concurrency capping or batching.

The study's only language failure is Python's synchronous insert path under concurrency 80: eighty concurrent push requests multiplexed onto one uvicorn event loop, each blocking on insertAll. At 1,500 msg/s p50 reached 9.4 s (p99 24 s) and ~15% of rows missed the measurement window, while Go and Rust held 22 ms. Two configuration changes each repaired it: concurrency 10

(more instances, less loop contention) yielded 88 ms; 500-row/200 ms batches yielded 25 ms at twice the load, indistinguishable from Go (23 ms) and Rust (24 ms). Conversely, in pipeline C — a pinned streaming consumer whose hot loop is librdkafka, fastavro, and C-backed gRPC — Python was the *fastest* language end-to-end (28.2 ms vs. Go 35.1 ms and Rust 47.0 ms at 500 msg/s; 50.8 vs. 51.3 vs. 88.7 ms at 5,000 msg/s), with identical per-instance throughput ceilings (~2,000 msg/s per pinned instance, scaling to ~5,000 msg/s per language at four instances, broker-capped). Rust's deficit localizes to its Pub/Sub client's REST batching versus streaming gRPC — a sink-client property, not a language property.

4.3 Decoupling the sink: asynchronous acknowledgment

Acknowledging before the sink write cuts caller-visible p50 from ~54 ms to 7–10 ms (Go 10.0 ms at 100% success after one anomalous run; Rust 6.7 ms clean at 500 req/s; Python 6.6 ms while shedding ~18% as deliberate bounded-queue 503 backpressure — its single event loop saturates before GCS does). The semantics change is categorical: durability becomes at-least-once-after-ack with an explicit loss window, appropriate for telemetry and inappropriate for transactional data.

4.4 Cold starts

Language	Boost off (ms)	Boost on (ms)	First-req failures
Go	573, 591	491, 491	0 / 4
Python	3,097, 2,440	3,461, 1,578	0 / 4
Rust	405, 532	3,191, 361	4 / 4 (HTTP 500)

Table 2. First-request latency after ≥15 min scale-to-zero, two trials per cell. Every Rust first request failed: a hand-rolled metadata-token fetch without retry, where SDKs retry transparently — its fast times are times-to-error.

Samples are deliberately small (n=2 per cell) and reported as indicative. The ordering — Go ≈ Rust, with Python 3–6× slower — is consistent with public measurements [12]; the actionable findings are the retry defect pattern and that startup CPU boost did not change the ranking.

5 Results: Round 2

5.1 The per-write cost ladder

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

Table 3. Same-clock sink-write p50 per operation, single instance, 2,000 msg/s (DML row: 500 msg/s; its 71–96 s is dominated by BigQuery's mutating-DML concurrency queue). The identical consumer body spans four orders of magnitude by sink choice alone.

The one systematic language gap in round 2 is the Storage Write API append (Go 16.3 ms vs. Python 49.5 ms) — a client-library asymmetry (proto-over-gRPC wrapper maturity) that was pre-registered as an expected outcome before execution. Managed Iceberg's write cost tracks native BigQuery within ~10 ms, supporting the view that BigQuery-managed Iceberg is a format choice rather than a performance choice; a five-query read benchmark (count, aggregate, windowed scan, point lookup, heavy scan) likewise placed managed Iceberg in the same class as a native table, with occasional cold-scan outliers.

5.2 Throughput: the batching ladder

Write strategy	Sustained	Backlog
Ext. Iceberg, per-msg commits	0.9 rows/s/writer	98%
Sync sink call per message	~50 msg/s/inst	—
Ext. Iceberg, 5k/10 s, 1 writer	~950 rows/s	4.7%
BQ insertAll batch, 1 vCPU	15,000 msg/s	0
BQ insertAll batch, 2 vCPU	25,000 msg/s †	0
Dataproc Spark batch (ref.)	37,856 rows/s	—
BQ Storage Write, 4 inst × 2 langs	49,930 rows/s	0

Table 4. Maximum sustained ingestion by strategy (1 KB messages, 120 s windows, drain-audited). † Broker-capped: the single e2-medium Kafka broker saturated before the 2-vCPU consumer; the same instance also cleared 15,000 msg/s at 1 vCPU. The Storage Write figure was verified by counting 5,991,840 sink rows against sent messages with zero undrained backlog (produce→consume p50 35 ms).

Moving from per-message synchronous writes to 500-row/200 ms client-side batches multiplies one instance's ceiling by ~300×; doubling CPU multiplies it by ≤1.7× (and even that bound is broker-limited). Cloud Run's request-based autoscaler cannot see consumer lag, so all Kafka results require pinned instances (min=max, CPU always allocated) or Worker Pools; the S5 Worker Pool deployment reproduced S1's results (produce→consume p50 7.1 ms at 2,000 msg/s, zero backlog) and removes the vestigial HTTP listener. A lag-based Kafka autoscaler for Worker Pools shipped after these runs [10] and remains future work.

5.3 External Iceberg: a commit-contention regime

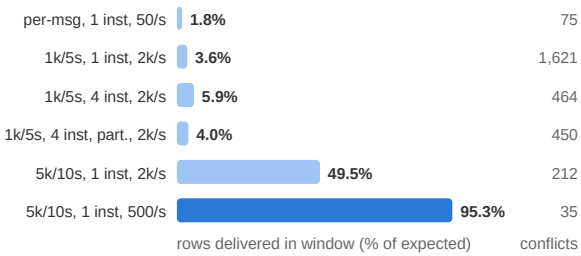

Figure 2. External Iceberg (S3): rows delivered within the audited window as % of expected, with optimistic-concurrency commit conflicts at right. Writers = instances × languages sharing one table. Partitioning does not reduce contention (450 vs. 464 conflicts): commits race on the table's metadata pointer, not on data files.

Every commit through the BigLake REST catalog costs ~2–3 s regardless of row count (p50 1.92 s Go, 2.79 s Python), which imposes both a latency floor and a concurrency regime: at one committing writer per language and 5,000-row/10 s commits, 95.3% of rows land in-window at 500 msg/s with 35 conflicts; every configuration with more writers or smaller commits degrades — down to 1.8% delivery for per-message commits. Conflicted commits retry (all failures were retried; the drain protocol records the shortfall rather than hanging). The practical regimes are a single-writer funnel with large commits (~10–15 s effective sink-ack) or scheduled batches: the Spark reference wrote 9,790,553 rows in 258.6 s (37,856 rows/s) plus 79 s provisioning, outrunning every streaming configuration into the same table.

5.4 Delivery semantics, failure, and rebalance

Failure injection. Killing one of two consumers mid-run at 2,000 msg/s (at-least-once, 500-row batches, 500 ms offset auto-commit) produced 9 duplicate rows and 0 lost messages out of 240,018 (0.0019% duplication), measured by distinct-identifier reconciliation; the kill is visible as a 22.5 s p99 in the window. **Rebalance.** A rolling restart paused the consumer group for 27.7 s (eager/range assignor) vs. 23.7 s (cooperative-sticky) — a pause, not a slowdown (inter-message spacing p50 ~0.01 ms between pauses). The worst observed failure mode remains silent: in round 1, a post-deploy librdkafka rebalance wedged with zero assigned partitions and no logged error; round 2's health gate (§3.4) exists because of it. **Managed delivery.** The Pub/Sub→BigQuery-subscription path costs 25–34 ms p50 of Kafka→Pub/Sub ingress and delivers at full rate to 5,000 msg/s per language with zero sink code; per protocol A4 its delivery is characterized by 5 s-polled visibility curves rather than millisecond claims. The pull alternative (S4b) reaches BigQuery in 7.0–9.0 ms p50 at the price of owning batching and retry logic.

6 Cost Model

At europe-west3 list prices (vCPU $0.000024/s; memory $0.0000025/GiB-s; requests $0.40/M), Cloud Run compute is a minor term for request-shaped paths: ≈$0.42 per million HTTP requests at concurrency 80 (the sink dominates: GCS Class-A writes ≈$5.00/M), and ≈$0.05–0.10 per million messages for batched Kafka consumers (15,000 msg/s per 1-vCPU instance ⇒ ~67 instance-seconds per million). Commit-bound external Iceberg inverts the ratio (≈$1.50–6.00/M of compute, throughput-limited) before storage-side costs. Pinned consumers bill continuously — ≈$24.50/month per 1 vCPU/1 GiB instance — making steady high-volume streams GKE-shaped economically, while bursty or idle streams retain the serverless advantage. Concurrency acts as a discount for I/O-bound request paths (~30% lower compute per request at concurrency 80 vs. 10, identical latency). The two study rounds themselves cost ≈$12–16 and ≈$9–13.

7 Discussion

Six design rules summarize the evidence. **D1:** optimize the sink client first, concurrency second, language last; no platform knob moved a sink-dominated median. **D2:** climb the batching ladder before the hardware ladder (300× vs. 1.7×). **D3:** treat language choice as an event-loop/concurrency architecture question; C-backed clients equalize languages, and one blocking sink call per request under high per-instance concurrency is the failure signature. **D4:** for pull-based consumption, pin instances or use Worker Pools, gate on group assignment, prefer cooperative-sticky, and commit offsets on SIGTERM. **D5:** choose managed Iceberg for format openness at native performance; enter external Iceberg only with single-writer funnels and large commits, or via scheduled batch jobs. **D6:** design for at-least-once everywhere (duplication is measurable and small; loss was not observed).

We hypothesize the qualitative results — sink dominance, the batching ladder, commit-contention regimes — transfer to other serverless container platforms (AWS App Runner, Azure Container Apps) because their causes lie in sink write APIs and client libraries rather than in Cloud Run internals; verifying this is future work, as is measuring the post-study lag-based Kafka autoscaler [10], BigQuery's Storage Write API at multi-broker Kafka scale, and multi-worker Python deployments.

8 Threats to Validity

Internal. Cross-machine timestamps carry NTP-level uncertainty; round 2's primary metric is same-clock by construction, and cross-clock metrics are labeled. A per-run clock probe recorded offsets, but its estimator conflates offset with one-way delay (offset ≈ probe RTT); we therefore do not clock-correct cross-clock metrics and treat them as skew-bearing. Round 1's batch mode acknowledges before durability; round 2's sink-ack metric closes that construct gap. One anomalous Go async run was superseded by a clean rerun; both are preserved in the artifact. **External.** One region, one month (July 2026), one GCP project; a single e2-medium Kafka broker caps consumer ceilings (25,000 msg/s is a lower bound, and round 1's 5,000 msg/s plateau is a broker property); Python ran a single uvicorn worker; library versions (July 2026) matter, particularly for the Storage Write Python client and iceberg-go. **Construct.** We report medians with p90/p99 in the artifact; cold-start cells are n=2; managed-delivery latency is visibility-polled at 5 s granularity and deliberately excluded from millisecond comparisons. **Conclusion.** Findings are engineering-scale effects (orders of magnitude, or repaired collapses), not small-percentage claims; we avoid statistical tests on n=2 cells.

9 Conclusion

Placed in the middle of real pipelines, Google Cloud Run is a capable and inexpensive streaming compute layer whose performance story is written almost entirely by the sink: what one write costs (7.9 ms to 95.8 s), how writes are batched (a 300× ladder), and how the sink handles concurrent committers (external Iceberg's conflict regime). Language choice collapses to configuration; hardware barely matters; and the platform's request-shaped autoscaler must be bypassed for pull-based work. We publish all artifacts [13] so that each number here can be recomputed, and so that the study can be rerun as the platform evolves.

Acknowledgments

The benchmark implementation and both measurement campaigns were executed autonomously by an AI coding agent (Kimi K3) under the author's direction and budget approval. An independent AI system (Claude, Anthropic) reviewed the round-2 experiment plan before execution — contributing the binding amendments A1–A5 of §3.4 — and subsequently validated both rounds by recomputing headline results from raw run artifacts. Residual errors are the author's.

Artifact Availability

Service source (Go/Python/Rust), load generators, orchestration, raw per-run JSON for all ~140 runs, both self-contained HTML reports, and an interactive cross-round writeup are available at github.com/ben-j-herbertz/cloudrun-latency-throughput-benchmark. Aggregated datasets regenerate byte-identically from raw results (`report/aggregate.py` in each round folder).

References

[1] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *USENIX ATC*, 2018.

[2] M. Shahrad et al. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX ATC*, 2020.

[3] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler. SeBS: A serverless benchmark suite for function-as-a-service computing. In *ACM/IFIP Middleware*, 2021.

[4] E. Jonas et al. Cloud programming simplified: A Berkeley view on serverless computing. arXiv:1902.03383, 2019.

[5] J. M. Hellerstein et al. Serverless computing: One step forward, two steps back. In *CIDR*, 2019.

[6] I. Baldini et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, Springer, 2017.

[7] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *CIDR*, 2021.

[8] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *NetDB*, 2011.

[9] Apache Software Foundation. Apache Iceberg. iceberg.apache.org, accessed July 2026.

[10] Google Cloud. Autoscale your Kafka consumer workloads (Cloud Run worker pools). cloud.google.com/run/docs/configuring/workerpools/kafka-autoscaler, accessed July 2026.

[11] D. Xia. Benchmarking Kafka and Google Cloud Pub/Sub latencies. davidxia.com, 2021.

[12] IOD. Google Cloud Run vs. AWS Lambda: Performance benchmarks, part 2. iamondemand.com, accessed July 2026.

[13] B. J. Herbertz. Cloud Run latency & throughput benchmark: artifact repository. github.com/ben-j-herbertz/cloudrun-latency-throughput-benchmark, 2026.