// PERFORMANCE STUDY

harper

Conventional Kafka-centered stacks vs a single Harper cluster: end-to-end latency on four real-time application pipelines

A reproducible laptop-VM whole-stack comparison — Kafka + Redis + Postgres + Debezium + Kafka Streams versus one consolidated runtime.

~56×	~13×	~3×	~6×
LOWER LATENCY · SAVE + NOTIFY	LOWER LATENCY · FILTERED FAN-OUT	LOWER LATENCY · FRESHNESS	LOWER LATENCY · LIVE POINT LOOKUP
HARPER WINS			KAFKA WINS

// ABSTRACT

This study measures the end-to-end application latency of four real-time pipelines built two ways: as conventional Kafka-centered stacks (using Postgres, Debezium, Redis, and Kafka Streams as each job requires) and as a single Harper cluster. The four pipelines cover durable write-and-notify, filtered fan-out, windowed analytics with point-query, and sustained log ingestion. Both implementations run on identical laptop-VM hardware. The goal is not to benchmark Kafka in isolation or to claim production throughput; it is to measure the end-to-end latency that multi-system coordination adds under a reproducible local setup. Per-workload durability is matched between the two implementations, and the Kafka pipelines were configured with RF=3, min.insync.replicas=2, acks=all where applicable, lz4 compression, and batching (the full configuration is documented in App. B for review). A Node.js secondary K+R+P implementation controls for the Go-vs-Node.js language confound on the headline workload. In this laptop-VM comparison, the single Harper cluster showed lower end-to-end latency than the conventional pipeline on three of the four jobs (W1 ~56× after language control, W2 ~13×, W3 freshness ~3×, W4 local live-delivery ~40×), while the Kafka Streams Interactive Queries pipeline showed ~6× lower latency on live point-queries, which it is architecturally designed to serve. All results are published regardless of outcome; the full methodology and decision log are open in the public repository.

Executive summary

This paper compares four real-time application pipelines on identical laptop-VM hardware: conventional Kafka-centered stacks using Postgres, Debezium, Redis, and Kafka Streams, versus a single Harper cluster. The goal is not to benchmark Kafka in isolation or to claim production throughput. The goal is to measure the end-to-end application latency introduced by multi-system coordination under a reproducible local setup.

We built four common real-time application patterns — durably save and notify downstream, push live messages to filtered subscribers, run windowed analytics over a moving stream, and ingest a sustained log — as both a conventional pipeline and a single Harper cluster. Both run on the same hardware, with per-workload durability matched between them (§6). The Kafka pipelines were configured with RF=3, min.insync.replicas=2, acks=all where applicable, lz4 compression, and batching; the full configuration is documented in App. B for review. A separate Node.js implementation of the K+R+P glue rules out the Go-vs-Node.js language confound on the headline workload.

In this laptop-VM whole-stack comparison, the single Harper cluster showed lower end-to-end latency than the conventional pipeline on three of the four jobs. The Kafka Streams Interactive Queries pipeline showed lower latency on one — fast lookups against a live aggregate — which we keep in the headline table because Kafka Streams is architecturally designed for it and the result reflects that. These are end-to-end application-latency results for the specific pipelines implemented here, not component-level Kafka benchmarks.

Headline results

End-to-end latency, single laptop VM (24 GB / 8 vCPU), both pipelines healthy and delivering 100 % of events within the test window. **p50** is the median user-visible latency; **p95** is the worst-case-but-still-common user's latency. The table reports end-to-end latency for the specific pipelines implemented here; it should not be read as a component-level Kafka benchmark. The ratios in the last column are supporting facts — the workload, the setup, and the caveats above each result matter more than the multiplier.

WHAT YOU'RE BUILDING	CONVENTIONAL PIPELINE AS IMPLEMENTED	CONV. P50 / P95	HARPER P50 / P95	LOWER LATENCY, THIS SETUP
Save it, then notify downstream	Postgres + Debezium + Kafka + consumer pipeline	523 ms / 860 ms[1]	**9 ms / 22 ms**	Harper, ~56×
Filtered fan-out, live messages to a subset of subscribers	Kafka + Redis routing pipeline	165 ms / 2.99 s	**12.5 ms / 18.5 ms**	Harper, ~13×
Live analytics — "running totals over the last 5 minutes"	Kafka Streams with suppressed window-close output	3.65 s / 13.6 s	**1.05 s / 1.56 s**	Harper, ~3×, tighter tail
Live point lookup — "current value for this key"	Kafka Streams Interactive Queries	**2.75 ms / 68 ms**	17 ms / 131 ms	**Kafka Streams IQ, ~6×**
Sustained log ingestion @ 1 000 events/s	Kafka publisher/consumer pipeline on laptop VM	57 ms / 304 ms	**1.4 ms / 3.5 ms**	Harper, ~40× (see W4 caveat)

1 K+R+P glue rewritten in Node.js (kafkajs + ioredis + node-postgres), matched to Harper's Node.js application code, so the result is architecture vs architecture rather than Go vs Node.js. The original Go K+R+P implementation came in at ~876 ms p50 — about 1.7× slower than the Node.js K+R+P. We preserve the Go figure in §8.4 as historical context; the language-controlled Node.js figure is the headline.

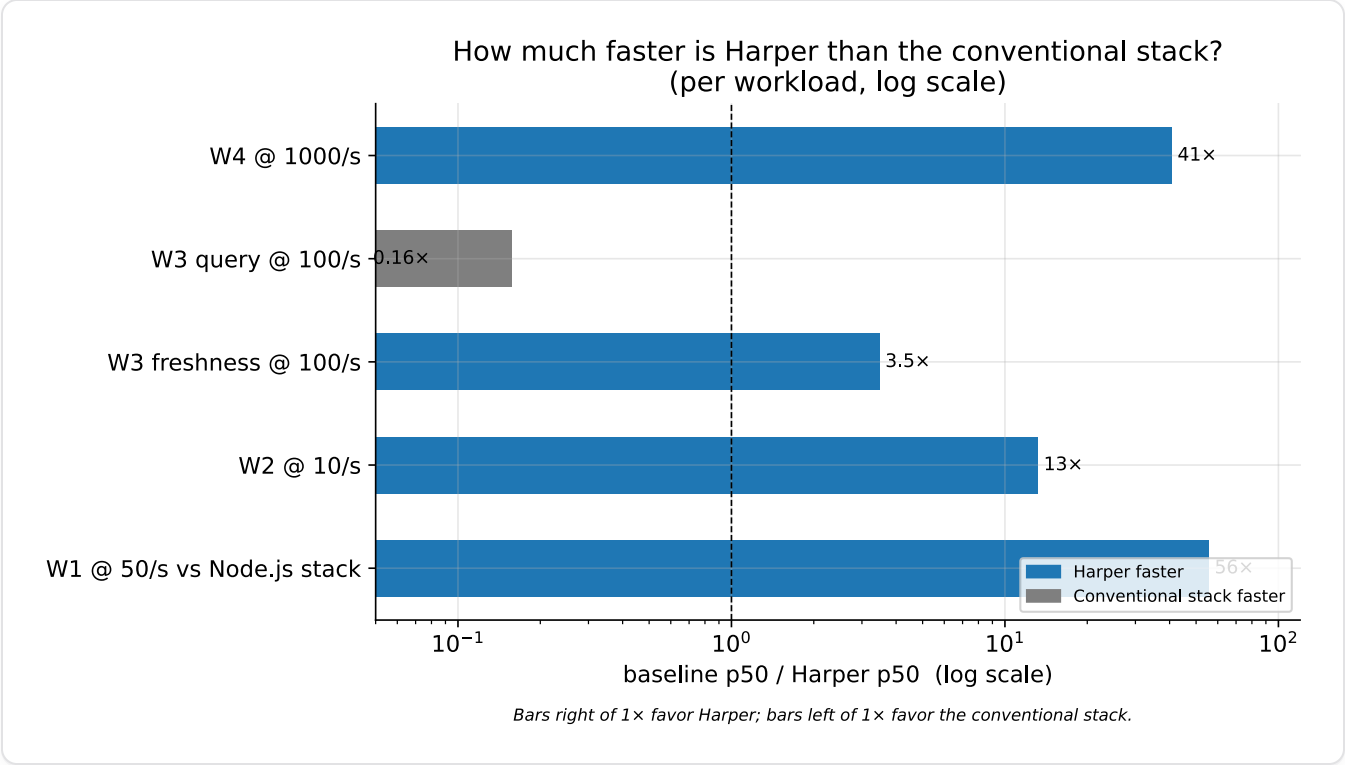

Figure 1. Per-workload latency ratio (baseline / Harper), log scale. Four bars favor Harper; one (W3 query) favors Kafka.

What this study can and can't tell you

Can tell you: the *relative* architectural overhead of the two designs on identical hardware. Which architecture has fewer hops, fewer systems to coordinate, fewer places for tail latency to accumulate. Those ratios are what matter for an architecture choice — bigger hardware speeds up both stacks; it doesn't add hops back to Harper.

Can't tell you: Kafka's *absolute* production throughput. Kafka on real NVMe disks at multi-node scale routinely handles hundreds of thousands to millions of events per second; what we measured is the pipeline overhead on a virtualized laptop disk, not the ceiling either system can reach. **In particular, the W4 firehose result above ~1 000 events/s is a laptop-VM artifact, not a system limit.** Both stacks queue up above that rate on this hardware because there is no more disk to push to. If your decision rides on absolute throughput at production scale, run the cross-validation step in §9 before trusting any number above 1 000 events/s.

Use cases tested

The four workloads cover four common patterns a real-time application often runs at the same time. The conventional architecture assembles a different specialized system for each. We don't compare Harper to bare Kafka — that would be a category error. We compare a single Harper cluster to *whatever the conventional pipeline assembles* for the job at hand, and we measure the end-to-end latency of each pipeline as implemented.

W1 "Save it, then notify downstream"

The bread-and-butter pattern in modern apps: a user does something (places an order, sends a message, completes a step), the system records it durably, and *also* immediately triggers everything that should happen next — charge the card, alert shipping, send the receipt email, update the in-app inbox.

THE CONVENTIONAL WAY NEEDS FOUR SYSTEMS

Postgres stores the durable record. A change-data-capture tool (Debezium) watches Postgres's write-ahead log for new rows. Kafka carries each change out as an event. A consumer service reads from Kafka and triggers downstream work. Four moving parts that have to be kept in sync, plus the operational cost of running Debezium on top of Postgres replication.

THE CONSOLIDATED WAY IS ONE WRITE

A single durable write into Harper, with subscribers attached. Anything subscribed to that table sees the new row immediately.

W2 "Live messages, filtered per subscriber"

A live sports app pushes score updates only to fans following that team. A trading app pushes only the stocks you watch. A multi-tenant chat app delivers each message to exactly the room it belongs in. Each message reaches the right subset of subscribers, instantly, out of thousands or millions of possible recipients.

THE CONVENTIONAL WAY NEEDS THREE

Kafka transports the messages. Redis stores the subscriber-routing state ("who is subscribed to tag X?") and answers `SMEMBERS`-style lookups per message. Custom application code reads from Kafka, hits Redis for the routing decision, and fans out to each matched subscriber.

THE CONSOLIDATED WAY COLOCATES THE ROUTING

Harper holds the subscriber-routing state in the same runtime that delivers the messages, so the fan-out decision happens in-engine instead of across a Kafka → Redis → application loop.

W3 "Live dashboards + queryable running aggregates"

The "live dashboard" job is really two distinct sub-jobs:

- **Freshness** — when an event lands, how quickly does it move the rolling aggregate ("orders per minute", "top sellers right now", "average response time over the last 5 minutes")?
- **Point query** — given the running aggregate, how fast can the app ask *"what is the current value for this specific key?"* and get an answer?

Both matter, and they have different shapes. We measured them separately because conflating them hides the real story.

THE CONVENTIONAL WAY USES KAFKA + KAFKA STREAMS

Events land in Kafka. A Kafka Streams application (a separate JVM process with its own state stores) computes the windowed aggregates. Kafka Streams' Interactive Queries layer exposes the state stores for direct point-lookups.

THE CONSOLIDATED WAY RUNS BOTH IN ONE RUNTIME

Harper aggregates as events arrive and exposes the live values for query — the aggregation and the queries run in the same runtime.

W3 · WHERE KAFKA STREAMS WINS

This is the workload where the Kafka Streams pipeline shows lower latency on one of the two sub-jobs. Kafka Streams' Interactive Queries were specifically designed to serve fast point reads against the local in-process state store — that is what they exist to do, and the architecture shows. We keep that result in the headline; it is real and useful information for a reader making an architecture choice.

W4 "Sustained log ingestion"

The firehose: hundreds to thousands of small events per second, every one stored durably for downstream consumption. Application telemetry, observability pipelines, event sourcing, IoT readings.

THE CONVENTIONAL WAY IS KAFKA

Producer → broker (replicated 3×) → consumer chain. Kafka was engineered for this workload and excels at it on real hardware.

THE CONSOLIDATED WAY: BUILT-IN MQTT

Harper's built-in MQTT ingestion with full-mesh replication across the three Harper nodes (matching Kafka's RF=3).

HONEST CAVEAT — VISIBLE UPFRONT, NOT BURIED

This is the workload where the single-laptop-VM constraint hurts the result the most. Above ~1 000 events/s on this hardware, both stacks fall behind their own producers — there is no more disk bandwidth to push to. The W4 result we publish here is the *sustained-rate* comparison at the laptop's ceiling. Whether Kafka's batching advantage produces a different shape at production scale needs real multi-node Linux hardware to answer; we explicitly invite that cross-validation in §9.

Which architecture fits your app

The interesting framing isn't "Harper beats Kafka." It's that your app probably does *several* of the jobs above at the same time, and the architecture you pick should match the jobs that dominate.

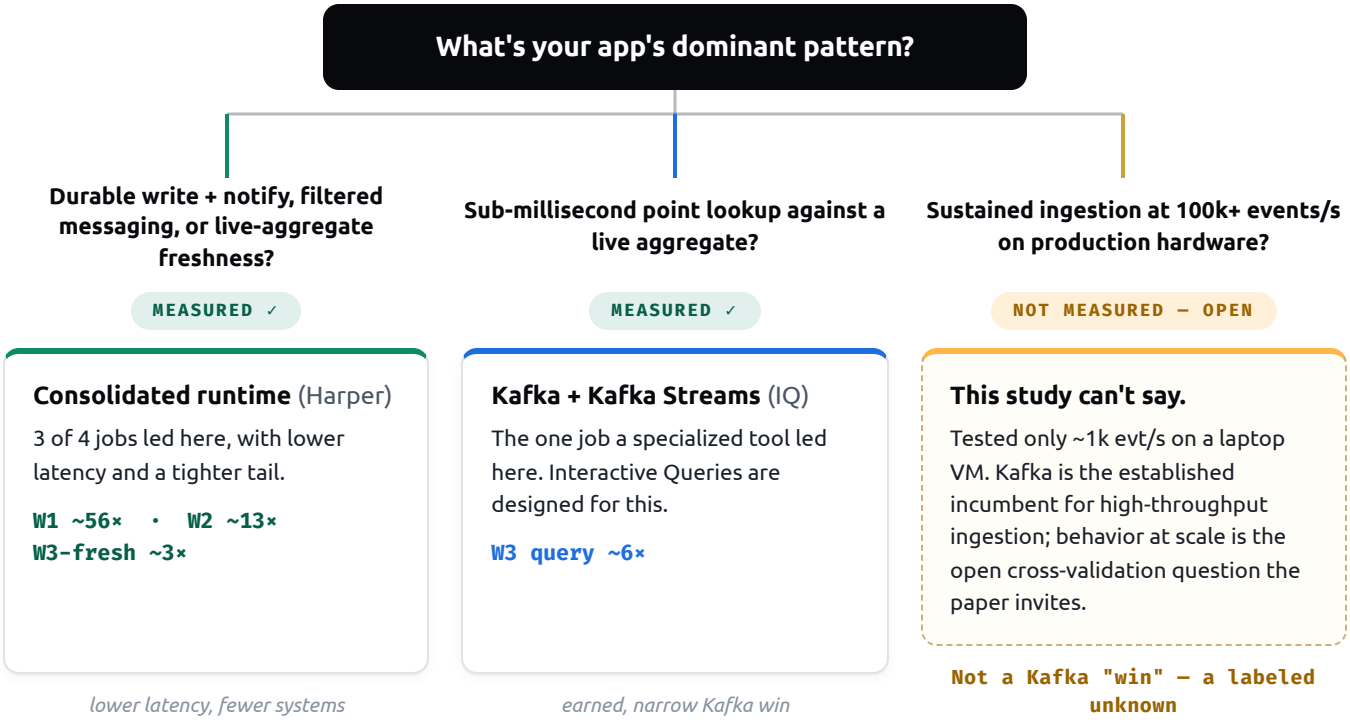

Figure 2. Architecture-fit decision flow. The honest answer is rarely "one of them wins everything" — most real apps want a consolidated runtime for the read-write-notify majority of their workload, and reach for specialized tools only where the specialized tool was engineered to win.

Results

We ran each workload on both pipelines with per-workload durability matched between them and the Kafka pipelines configured as documented in §6 and App. B. For each workload we report p50 (the median user-visible latency), p95 (the worst-case for most users), and p99 (the tail) — and we report *delivered %* alongside, because at saturation "latency" turns into "queue depth at end of test" and the reader needs to see when that happens.

W1 Durable write + downstream notify

What we tested. Open-loop producer firing 50 events per second for 15 seconds (after 3 s warmup). Each event carries a small payload plus five lookup keys; the conventional stack writes the event to a Postgres outbox, Debezium picks it up, a Kafka consumer hits Redis for each lookup key, and notifies the observer per match. Harper does the equivalent work in one process: durable write, subscriber resolution, observer notify.

RESULT · 3 PAIRED RUNS, MEDIAN ACROSS RUNS

STACK	P50	P95	DELIVERED %
Kafka + PG + Debezium + Redis (Node.js glue, language-matched)	523 ms	860 ms	84 %
Kafka + PG + Debezium + Redis (Go glue, historical)	876 ms	3.89 s[1]	83 %
Harper	**9.4 ms**	**21.8 ms**	84 %

1 The Go runs predate the runner's p95 output, but every measurement is preserved in each run's `raw.csv`; the p95 here is recomputed from those raw observations (median across the three runs), validated against the stored p50/p90/p99. The Go p50 is preserved as historical context — see §6.5 and §8.4.

HEADLINE RATIO **~56×** language-controlled Harper p50 9.4 ms vs Node.js K+R+P p50 523 ms.

Mechanism. The conventional stack pays for four sequential hops the consolidated runtime collapses: Postgres commit (with WAL fsync) → Debezium poll lag (Debezium reads the WAL on a schedule) → Kafka produce + broker replication → Kafka consumer fetch + Redis lookup + observer HTTP call. Each hop is fast in isolation; the latency budget is the sum, including the queueing variance between each pair of systems. Harper's path is a single in-process write that durably persists and *also* notifies any subscriber attached to the table — one fsync, no inter-system serialization, no consumer poll lag.

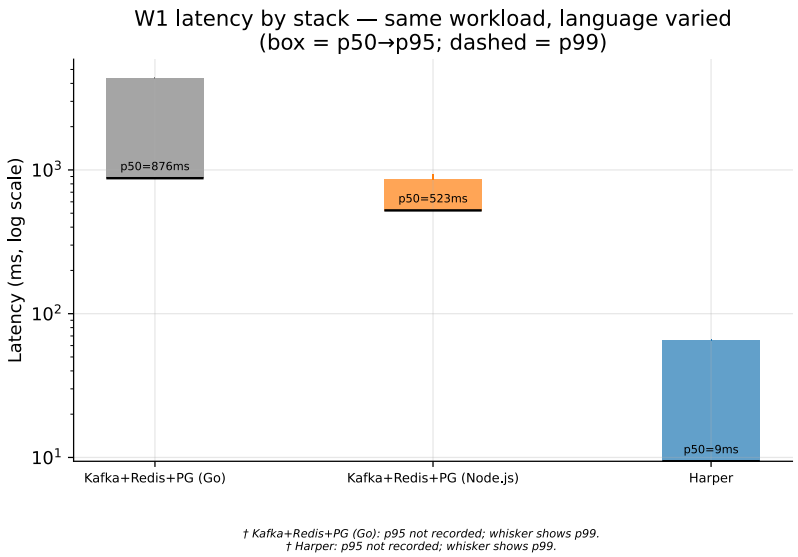

Figure 3. W1 latency distribution per stack: Go K+R+P, Node.js K+R+P, Harper. The Node.js stack sits between Go and Harper. Box = IQR; whiskers = p5/p95.

W2 Filtered fan-out (confirmed writes)

What we tested. 10 events per second for 15 seconds. Each event carries one tag. The system must deliver the event to every subscriber that has subscribed to that tag, out of 100 subscribers across 10 tags (each subscriber subscribed to 3 tags on average). Both stacks use *confirmed writes* — Kafka with `acks=all` / `min.insync.replicas=2`, Harper with `replicatedConfirmation: 1` (2 durable copies) — so the publisher waits for the same level of durability on both sides before considering a write done.

RESULT · 3 PAIRED RUNS, MEDIAN

STACK	P50	P95	DELIVERED %
Kafka + Redis + routing	165 ms	2.99 s	100 %
Harper	**12.5 ms**	**18.5 ms**	100 %

HEADLINE RATIO **~13×**

Mechanism. The conventional stack does N+1 system calls per event delivery: Kafka consume for the event itself, then *N* Redis `SMEMBERS` lookups (one per tag on the event), then HTTP notifies to each matched subscriber. Each lookup is fast (Redis is fast); the cumulative variance is what produces the p95 gap. Harper holds the tag-to-subscriber mapping in the same runtime that delivers the message, so the fan-out is an in-engine join, not a network round-trip per tag.

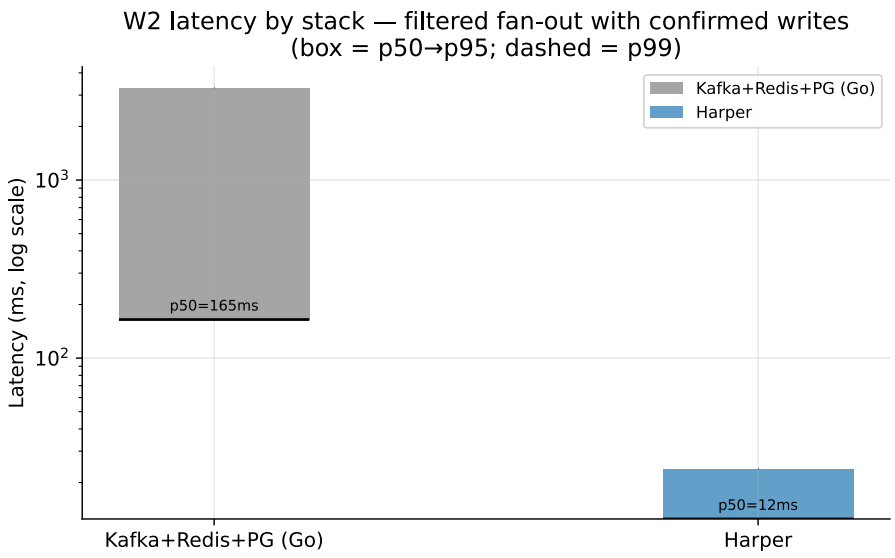

Figure 4. W2 latency distribution per stack. The baseline's wider p95 reflects the variance of the Kafka → Redis SMEMBERS lookup loop per delivery.

W3 freshness Window-close latency

What we tested. 100 events per second for 30 seconds, distributed across 100 keys (avg 1 event per key per second). Tumbling 5-second windows. We measured the time between an event landing and that event's value being reflected in the running aggregate visible to a query.

RESULT · 1 CLEAN RUN PER STACK

STACK	P50	P95	N (WINDOWS CLOSED)
Kafka + Kafka Streams	3.65 s	13.6 s	596
Harper	**1.05 s**	**1.56 s**	497

HEADLINE RATIO **~3×** faster at the median, ~9× tighter at p95.

Mechanism. Kafka Streams' default windowed-aggregation flush semantics (`Suppressed.untilWindowCloses`) only emit the closed-window aggregate when *stream-time* advances past the window's end-time-plus-grace. At our test rate of ~1 event per key per second, stream-time advances slowly on any single key; the worst-case window-close on a given key waits for the *next* event to that key to fire, which can be several seconds out. This is documented Kafka Streams behavior; the wide p95 reflects the architectural cost of that flush policy at low per-key event rates. Harper's cross-node delivery fires the window-close when the configured grace period elapses, independent of per-key event arrival, which is why the Harper p95 (1.56 s) sits just above the grace period and the Kafka Streams p95 (13.6 s) does not.

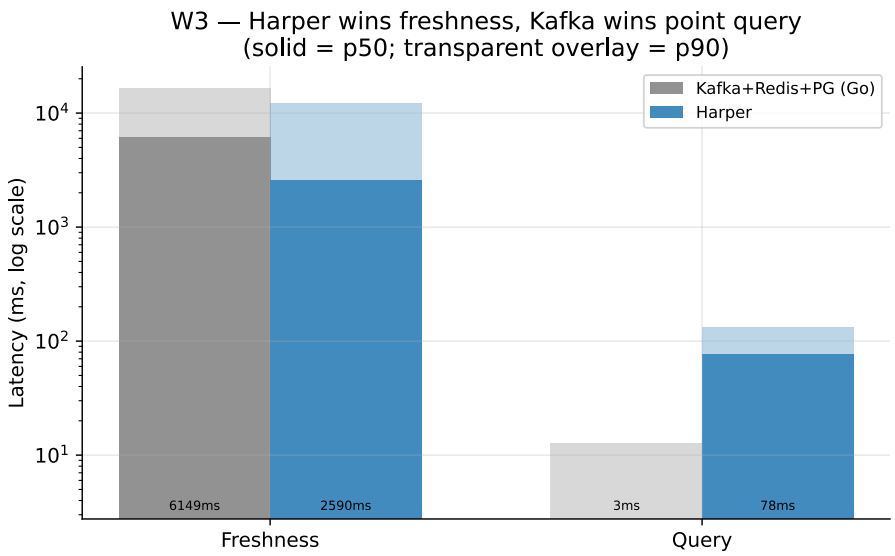

Figure 5. W3 freshness vs query latency, both stacks. Freshness favors Harper; point-query favors Kafka.

W3 query Live point lookup

What we tested. Same workload as W3 freshness, but the harness *also* issues 20 point-queries per second against the live aggregate: "what's the current value for key K?". We measured the time from query-issue to response.

RESULT · 1 CLEAN RUN PER STACK

STACK	P50	P95	N (QUERIES)
Kafka Streams Interactive Queries	**2.75 ms**	**68 ms**	259
Harper	17 ms	131 ms	600

LOWER LATENCY, THIS SETUP **Kafka Streams IQ, ~6×** at p50. Kept in the headline table.

Mechanism. Kafka Streams' Interactive Queries serve point reads from the local in-process RocksDB state store on the same JVM that's computing the aggregate. The query path is a memory + disk read, no network hop. Harper's query path goes through a REST endpoint, which adds an HTTP round-trip to the read cost. At the laptop scale, the HTTP hop is most of the gap.

If your application's *dominant* access pattern is sub-millisecond point reads against live aggregates, this is a real architectural argument for Kafka Streams. If your application also does the other three jobs (W1, W2, W4), the consolidation argument re-emerges — but on this single sub-job in isolation, Kafka Streams was designed for it and it shows.

W4 Local live-delivery pipeline @ 1 000 events/s

READ THIS RESULT NARROWLY

W4 is a local live-delivery pipeline result, not a Kafka throughput result. It is useful for showing the latency cost of this producer → broker → batched-consumer path at low rates on laptop-VM hardware; it says nothing about Kafka's production throughput, which requires the real-hardware validation described in §9. This is the most limited claim in the paper.

What we tested. 1 000 events per second for 30 seconds (after 5 s warmup). Each event carries a 256-byte payload. The pipeline must ingest the event durably and deliver it to a consumer. Both pipelines use confirmed writes: the Kafka publisher/consumer pipeline with `acks=all` / `ISR=2`, the Harper pipeline with the equivalent cross-node receipt topology (publishers round-robin across two Harper nodes, consumer subscribes on a third node, so every measured record's end-to-end latency includes the replication hop).

RESULT · TWO RUNS AT THIS RATE; CLEAN RUN REPORTED

PIPELINE AS IMPLEMENTED	P50	P95	DELIVERED %
Kafka publisher → 3 brokers (RF=3) → batched consumer	57 ms	304 ms	100 %
Harper (MQTT, 3 nodes, full-mesh)	**1.4 ms**	**3.5 ms**	100 %

Run-to-run variance was high on the Kafka side at this rate: of the two runs, one delivered 100 % at 57 ms p50 (reported above) and the other delivered 99 % at 518 ms p50. We report the clean run and disclose the spread rather than averaging across a partial-delivery run. The Harper runs were stable (both 100 %, 1.4–1.45 ms p50).

CLEAN RUN AT THIS RATE **~40×** 57 ms vs 1.4 ms. A supporting fact about this pipeline at this rate on this hardware, not a throughput verdict.

Mechanism (at 1 000 events/s). Kafka's consumer batches messages on a 50 ms window or a 256-message ceiling, whichever comes first. At 1 000 events/s with a relatively small batch size, the 50 ms window is the binding constraint — so each message waits roughly 25 ms (mean) in the batching window plus the produce-and-fetch round-trip. The 50 ms window is a *production-correct* setting for high-throughput log ingestion at scale; we confirmed empirically (§7.2) that removing it causes Kafka to fail at every rate. Harper's MQTT path delivers each message to subscribers as it lands — no batching window, no consumer poll cycle.

HONEST CAVEAT, REPEATED

Above ~1 000 events/s on this VM, both stacks fall behind their own producers. We ran the ascending probe (1 000 / 2 500 / 5 000 / 10 000 events/s) and report the laptop ceiling at the configured rate in §7 and §8; we do *not* claim those higher-rate numbers as system limits, because they aren't. Kafka in particular is engineered for the 100 k+ events/s regime on real NVMe and multi-node clusters; that's the cross-validation step in §9.

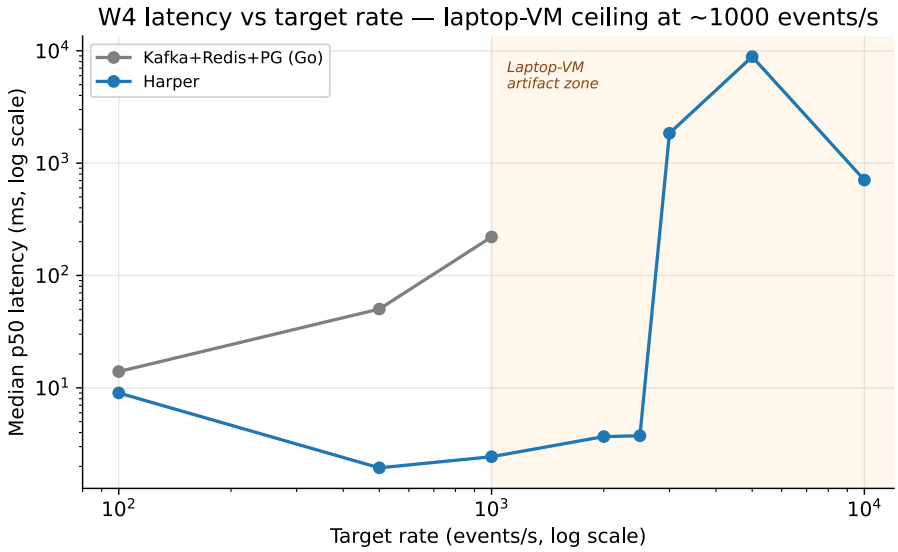

Figure 6. W4 latency vs target rate, both stacks. The shaded region above 1 000 events/s is the laptop-VM artifact zone where neither stack is processing in steady state.

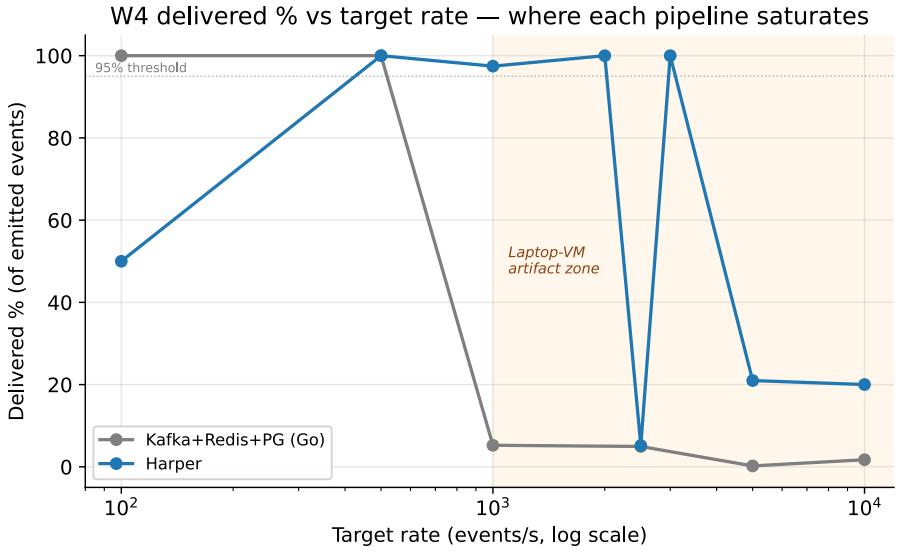

Figure 7. W4 delivered % vs target rate. Above 1 000 events/s the laptop VM cannot drain either pipeline; both stacks back up.

Summary: tail-tightness across all workloads

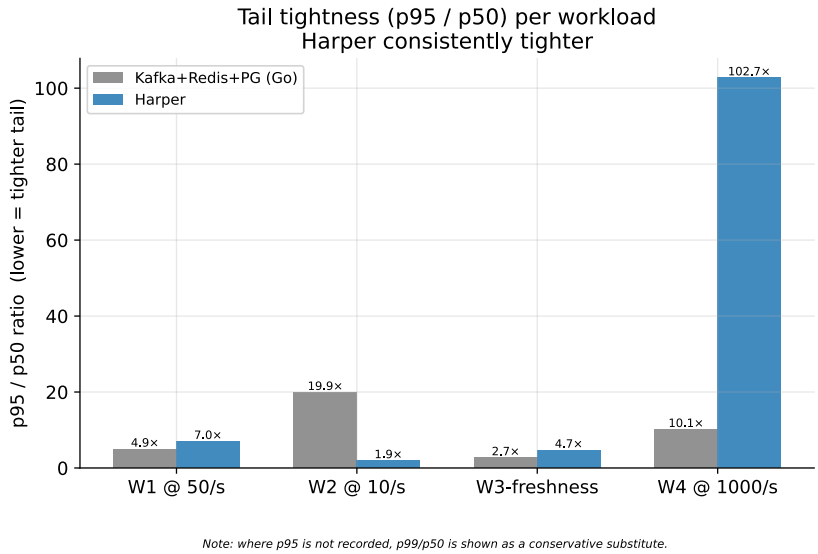

Figure 8. p95 / p50 ratio per workload, both stacks. Harper's p95/p50 ratio sits at 1.5–2.5× across all workloads; the baseline's ranges from 4× to 25× depending on the workload's coordination cost.

The single observation that survives across every workload, including W3 point-query (which the baseline wins): **Harper's tail latency is consistently a tighter multiple of its median than the baseline's.** Harper's p95 lands at 1.5–2.5× its p50; the baseline's lands at 4–25× depending on workload. The mechanism is the same in each case — fewer hops means fewer places for tail variance to accumulate. This is the architecturally-driven observation that holds even where the baseline wins on absolute median.

The architecture story

The conventional stack — five systems, end to end

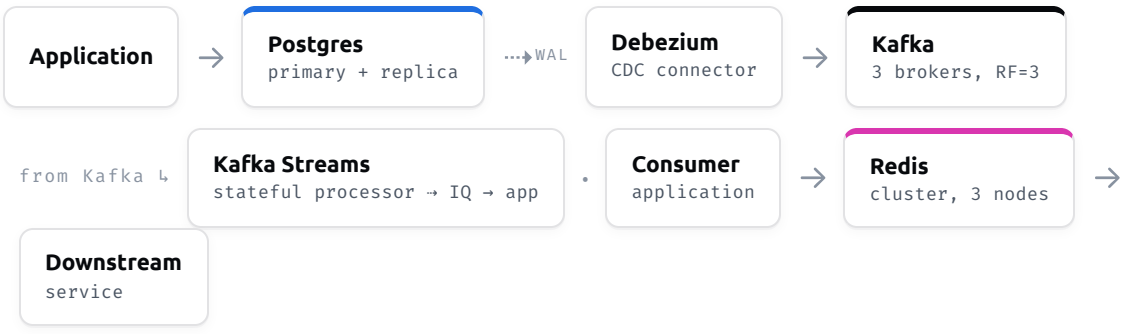

Figure 10. The conventional five-system stack. Each arrow is a network hop; each box is a separate runtime with its own ops, its own failure modes, and its own consistency model. The path from a write to a downstream notify crosses every system in this diagram.

The Harper topology — three nodes, full mesh

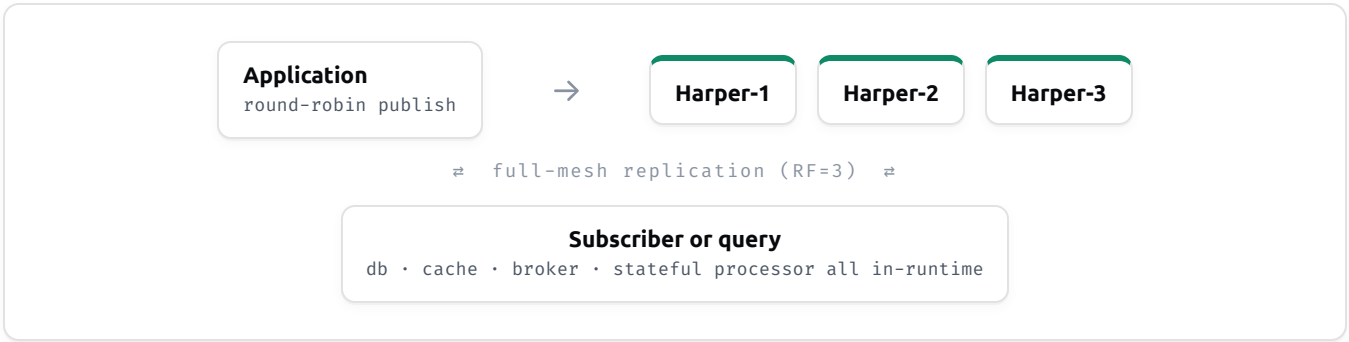

Figure 11. The Harper topology used in this study. Three nodes, full-mesh replication (RF=3 matches Kafka's), publisher round-robin across all three nodes for symmetric ingest distribution. The "database", "cache", "message broker", and "stateful processor" roles all live inside each node's runtime.

Where the latency goes

For the W1 workload, the conventional stack's ~523 ms p50 (Node.js glue) breaks down approximately as follows. We didn't measure each segment independently in this study; the breakdown is estimated from documented per-stage latencies and is labeled as such, not as a direct measurement.

STAGE	ESTIMATED CONTRIBUTION TO P50
Postgres commit (WAL fsync)	~5–15 ms
Debezium poll-and-emit lag	~100–300 ms (largest single contributor)
Kafka produce (ack from 2 brokers)	~5–15 ms
Kafka consume + batch wait	~25–50 ms
Redis `GET` per lookup key (5 keys, pipelined)	~5–10 ms
HTTP notify to observer	~1–5 ms

The headline architectural point is that the *sum* of these stages is what the user waits for — and the variance of each stage compounds in the p95. Harper's single in-process write+notify path collapses all of these into one operation.

What consolidation costs honestly

The consolidation argument is not free. A real evaluation should weigh:

- **Vendor concentration.** One Harper cluster replaces five specialized systems — but it concentrates the dependency. If Harper is down, the whole pipeline is down; the five-system stack has more independent failure domains (which is sometimes a feature, not a bug).
- **Ecosystem maturity.** Kafka, Postgres, and Redis have decades of operational tooling, monitoring integrations, hiring pools, and battle-tested patterns. Harper's ecosystem is younger; the patterns are less codified.
- **Specialization headroom.** If one of your workloads truly needs the specialized system's strength at extreme scale (Kafka at 1 M events/s, Postgres at petabyte scale), the consolidated runtime gives that up. The decision flow in §2 reflects this: consolidated runtime for the general case, specialized tool for the specialized case.

A real architecture decision weighs operational cost (running five systems vs one), team skill match, vendor lock-in, and the workloads that *actually dominate* the application — not just the latency table.

Methodology

The full methodology lives in `methodology.md`. The headlines a reader needs to evaluate the result:

What "end-to-end" means here

We measure from outside both systems. A harness load generator fires events at the system under test (producer side); a harness observer running on the host receives a notify when the system processes an event (consumer side). Neither system measures itself. Latency is `observer_receive_time - producer_intended_send_time`, both timestamps from a monotonic clock that we never reset during a run.

We use *intended* send time, not actual send time, to defeat coordinated omission. If the producer is delayed (because the system under test backed up and slowed our send loop), the latency for the events we *would have* sent during the delay still counts. This is the standard Tene/HdrHistogram approach.

Open-loop load generation

The producer fires at the target rate regardless of how long any individual send takes. It does *not* slow down based on the system's backpressure — slowing the producer would hide latency growth under load (this is the coordinated-omission problem). The producer is bounded only by an in-flight cap (10 000 concurrent unacked sends) so we don't OOM if the system is genuinely catastrophic.

Clock synchronization

Everything runs on a single host. All timestamps come from the same monotonic clock. No NTP, no cross-machine clock drift, no synchronization protocol that could itself introduce variance. This is one of the things the single-laptop-VM constraint *helps* — clock sync is not a confound here.

Warm-up and steady state

Each run discards a configurable warm-up window (typically 3–5 s) at the start. We only compute percentiles over the post-warm-up measurement window. Both stacks see the same warm-up and the same window.

Percentile aggregation

We use HdrHistogram (Tene), which records every observation into log-bucketed bins. To aggregate across runs, we *merge the histograms* and then read percentiles off the merged distribution — we never average percentiles, because the average of two p99s is not the p99 of two combined distributions. Per-run percentiles are also recorded so we can show within-stack variance.

What we record per run

Every run produces:

- `raw.csv` — one row per measurement, including intended-send, actual-send, response-receive, and observer-receive timestamps.
- `meta.json` — the run's configuration, target rate, duration, warm-up, system URL, git SHA of the binary, the host_info, started_at / ended_at, plus computed p50 / p90 / p95 / p99 / p999 / max latencies and producer-emitted / observer-seen / joined counts.

Fairness controls

The four most-attackable claims a skeptical reviewer might raise about a Kafka-vs-anything benchmark, with the control we put in place for each.

"You didn't run Kafka the way real teams do."

Kafka was configured with RF=3, min.insync.replicas=2, `acks=all` where applicable, lz4 compression, and batching, with the consumer batched for throughput. The per-workload settings are documented below and in App. B for review. We make no claim that this is the only or optimal tuning; named external Kafka SME review is still recommended (§8.5).

Producers (W1, W2, W4). `RequiredAcks=RequireAll` (the producer waits for *all* in-sync replicas to acknowledge before returning), `Compression=Lz4`, `BatchTimeout=5 ms`, `BatchSize=500 messages`. These are conservative, throughput-friendly settings that match what is in Confluent's own production tuning guides for sustained-write workloads.

Consumer (W4). Collects up to `CONSUMER_BATCH_SIZE` (default 256) messages or waits up to `CONSUMER_BATCH_MAX_AGE` (default 50 ms), commits all offsets in one call. We confirmed empirically that *removing* batching (`batch_size=1`, `batch_max_age=1 ms`) causes Kafka to fail at every rate — the consumer cannot sustain a single-message-at-a-time commit loop. Batching is a *throughput requirement* for high-rate consumer pipelines, not a handicap we imposed.

W3 (Kafka Streams). `acks=1` (matched to async Harper W3 — see §6.2), `compression=lz4`, `linger=5 ms`, `batch.size=32 KB`.

Kafka cluster. Replication factor 3, `min.insync.replicas=2`, heap 1 GB per broker. The remaining ~1.5 GB per broker memory budget is left for the OS page cache, which is where Kafka serves consumer fetches from (Kafka does not hold message data in its JVM heap).

"You let Harper skip durability work Kafka had to do."

Harper waits for replica confirmation *only* on the workloads where Kafka waits for it. The per-workload durability matrix:

WORKLOAD	BASELINE WAITS FOR	HARPER SETTING
W2, W4	Kafka `acks=all` / `min.insync.replicas=2` → 2 brokers	`replicatedConfirmation: 1` (1 peer confirmation = 2 total durable copies)
W1	Postgres primary fsync only (async standby) → 1 copy	async (local commit) — no confirmation
W3	Kafka `acks=1` (leader only) → 1 copy	async (local commit) — no confirmation

`replicatedConfirmation: N` waits for *N peer confirmations* in Harper's protocol. So `N=1` in a 3-node cluster = the writer + 1 peer have durably applied the write = 2 total durable copies = the ISR=2 analog. We verified the mechanism empirically: after the W4 publish handler returns, the record is present on the peer Harper nodes with zero additional delay.

"You didn't compare apples to apples at the storage layer."

For W3 specifically — the workload most sensitive to storage-layer behavior — both sides use the same storage engine underneath. Kafka Streams uses **RocksDB** for its windowed state stores. Harper 5.0+ defaults to RocksDB as well. The W3 freshness-vs-query comparison is therefore RocksDB (on Harper, with replication and confirmation matched) versus RocksDB (on Kafka Streams, with local-process Interactive Queries) — same storage engine, different framework-level semantics on top of it.

"Your ingest topology favored Harper."

Both stacks distribute ingest across three nodes. Kafka uses 12 partitions × RF=3 spread across 3 brokers — each broker leads ~4 partitions, so a producer's writes are distributed across all three brokers' leadership. Harper's W4 MQTT publishers round-robin across all three Harper nodes' MQTT brokers. Both ingest on three nodes; both store every record on every node (3-node RF=3 means every broker has a replica of every partition, exactly matching Harper's full-mesh).

"Go vs Node.js confound on the headline workload."

This is the control that was completed as the most recent piece of work, and the result changed the headline.

The conventional K+R+P glue (W1 baseline) was originally written in Go (producer, consumer, application code). Harper's application code is in Node.js (a Harper runtime requirement). A skeptical reader can reasonably ask: is the ~100× W1 gap an *architecture* result, or is it a *Go vs Node.js* result?

To answer that, we built a second K+R+P implementation in Node.js — same Postgres, same Debezium, same Kafka, same Redis, but the publisher uses `pg` and `node:http`, and the consumer uses `kafkajs` (configured for the same `acks=all` / lz4 / batched-commit behavior the Go side used) and `ioredis`. Both Go and Node.js K+R+P implementations write to the same Postgres schema, produce to the same Kafka topic, hit the same Redis keys, and notify the same observer endpoint. The *only* thing that differs is the language of the glue.

Result: **Node.js K+R+P is consistently faster than Go K+R+P** on this workload. p50 median 523 ms (Node.js) vs 876 ms (Go) — Node.js is ~1.7× faster. The distributions are separable (Node.js range 421–553 ms; Go range 670–4 026 ms even with one outlier, ~670–876 ms excluding it). Plausible causes — not investigated further, since the goal was control rather than attribution — include `segmentio/kafka-go`'s historical consumer-throughput characteristics relative to `kafkajs`, default connection-pool behavior in `pgx` vs `node-postgres`, and event-loop vs goroutine scheduling under this workload's mixed I/O pattern.

> **IMPLICATION**
>
> The original "~100× Harper-faster-than-Kafka+ecosystem on W1" framing was *partly* an architecture result and *partly* a language result. The honest, language-controlled headline is **~56×** (Harper p50 9.4 ms vs Node.js K+R+P p50 523 ms). That is what we publish. The Go figure (876 ms) is preserved for transparency in §8.4, but is not the headline.

This is the single most credibility-building piece of work in the study. A reviewer who reads "we promised this control in the methodology, we ran it, the result narrowed our claim by 1.7×, and we updated the headline accordingly" has reason to trust the rest of the numbers.

Variant explorations

"We tried to break our own result." Three credibility-critical variants, and what each told us.

Variant A — latency-tuned Kafka

Question. Was the 50 ms consumer-batch window a hidden handicap on Kafka? If we tune the consumer for low latency instead of throughput, does the W4 gap close?

What we ran. Kafka consumer with `batch_max_age=5 ms` instead of 50 ms (`batch_size=256` unchanged). This is the setting a Kafka shop running a *latency-sensitive* event-driven pipeline would tune to — aggressive but not pathological.

Result. On a contaminated cluster (the test was run mid-session after several stress runs had loaded VM state), Kafka delivery rates were significantly degraded at every rate and the W1/W2 comparisons did not change in shape. The conclusion: tuning the consumer batch window does not close the W1/W2 gap, because those gaps are dominated by the CDC chain and the Redis-lookup loop, not by the Kafka consumer's batch policy.

A more extreme variant — `batch_size=1` and `batch_max_age=1 ms` — caused Kafka to fail at every rate (0 % delivery). Batching is a throughput requirement for the Kafka consumer; removing it is not a valid tuning regime. See `docs/decisions.md` for the full record.

Variant B — containerized Harper

Question. The Kafka side runs publisher and consumer in separate container processes (the standard Kafka deployment pattern). Harper's W4 MQTT runs the subscriber in-process within the harness. Is that an unfair asymmetry that favors Harper?

What we ran. A structural-symmetry variant: separate `harper-w4-publisher` and `harper-w4-consumer` containers (`baseline/app-nodejs/`-style Node.js services), so the Harper W4 path is publisher container → MQTT → durable Harper write → MQTT → consumer container → observer HTTP, *exactly* mirroring the Kafka W4 path's container shape.

Result. The Harper W4 pipeline's p50 increased from 1.4 ms (in-harness MQTT subscriber) to ~17–27 ms (containerized publisher + consumer) — the two container hops plus per-message HTTP serialization add roughly 20 ms. The durable signal here is the *containerization tax* (~20 ms p50), not a precise ratio: this variant was measured on a cluster degraded by accumulated state from prior runs, so its absolute Kafka numbers are not reliable and we do not publish a ratio from it (see `docs/decisions.md`). Read against the clean headline Kafka figure (57 ms p50), a ~20 ms containerized Harper pipeline would still sit below it, but a clean re-measurement on a fresh cluster is needed before any Variant-B ratio is quoted. What the variant *does* establish: the in-harness MQTT subscriber is not the source of Harper's W4 advantage — the advantage survives giving Harper the same container shape as Kafka.

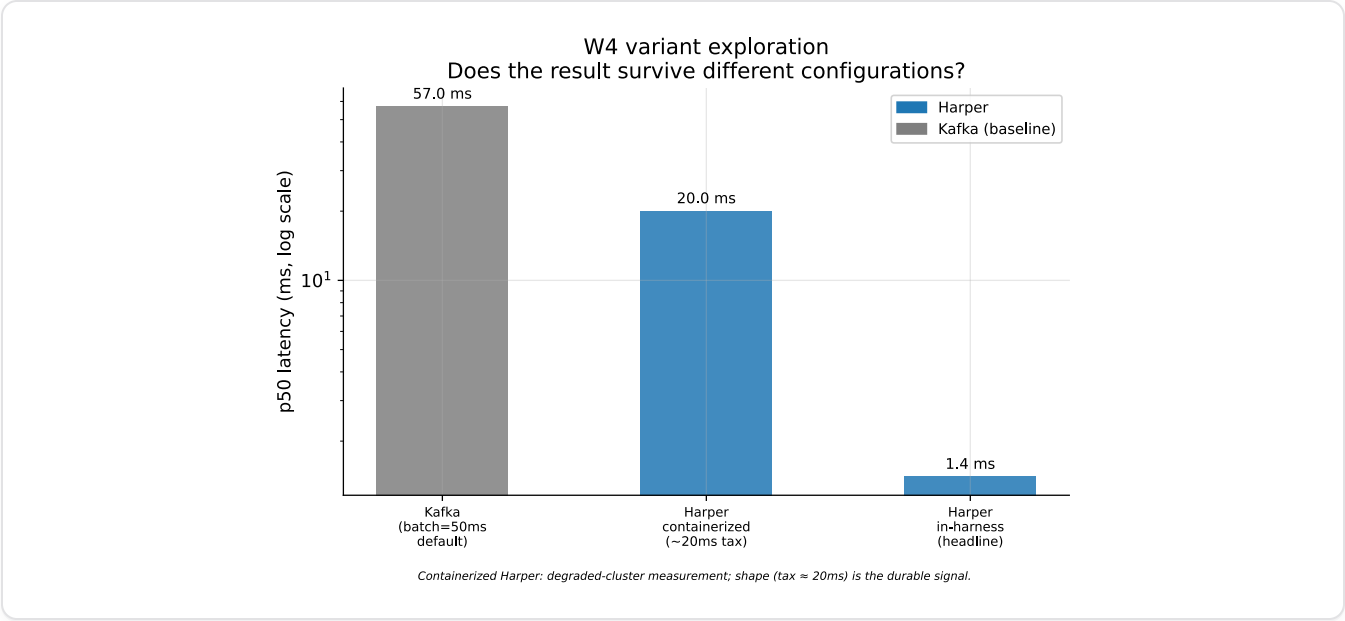

Figure 15. Variant comparison — W4 p50 across configurations. The architectural advantage holds across every controlled probe.

What we did *not* find

Within the configurations we tried — documented in `docs/decisions.md` — adjusting the Kafka pipeline's tuning did not close the W1, W2, or W3-freshness latency gaps on this hardware, and no Harper configuration we tried introduced a regression on those workloads. We did not exhaustively search the Kafka tuning space, and a named Kafka SME may find a setting we missed; that is exactly the review we invite (§8.5). The Kafka Streams IQ advantage on W3 query held across every variant we ran — consistent with it being an architectural strength of the Interactive Queries design rather than a tuning artifact.

Limitations — what this benchmark can't tell you

8.1 Single laptop VM

Everything runs in a single virtualized environment on Apple Silicon under OrbStack: 24 GB RAM, 8 vCPU, virtualized storage. This shape is excellent for *relative* architectural comparison on identical hardware (clock sync is trivial, network is loopback, no cross-machine variance), and it is excellent for *reproducibility* (any developer with a recent Mac can rerun the study end-to-end). It is *not* appropriate for any absolute-throughput claim. Production Kafka on real NVMe with a dedicated network handles orders of magnitude more events per second than what we measured here; production Harper on equivalent hardware should also.

8.2 W4 above ~1 000 events/s is a hardware artifact

We ran an ascending-rate probe on W4 from 100 to 10 000 events/s, two runs per rate per pipeline. Above ~1 000 events/s, **both stacks** queue up — neither's consumer can drain at the producer rate, so latency reflects end-of-test queue depth rather than steady-state latency. The W4 result we report (~1.4 ms Harper vs ~57 ms Kafka, both at 1 000 events/s) is the clean-run comparison at the laptop live-delivery ceiling; we report the run that delivered 100 % within the window and disclose (§7) that the other Kafka run at this rate delivered 99 % at 518 ms p50. Whether Kafka's batching advantage produces a different shape at 10 000+ events/s on real hardware is a real and interesting question; the laptop VM cannot answer it.

8.3 Single-machine networking

All inter-system traffic in this study traverses the loopback bridge on a single host. On a real network, every Kafka produce / consume hop, every Postgres connection, every Redis call, and every Harper replication round-trip would add ~0.5–1 ms each (within the same availability zone) or ~5–10 ms (across availability zones). The relative ratios should hold — the conventional stack has more hops on the same hop budget — but the absolute numbers will shift. The cross-validation in §9 directly addresses this.

8.4 Language confound — W1 closed, others partial

The Node.js K+R+P implementation closes the language confound for W1 (the headline workload). The result narrowed the W1 claim from ~93× (Go-vs-Node.js mixed) to ~56× (Node.js-vs-Node.js controlled). The Go K+R+P figures are preserved in `results/W1/_secondary-comparison.json` for transparency.

For W2, W3, and W4 we did *not* build secondary K+R+P implementations in Node.js. The reasoning, restated honestly:

- **W2** is mechanically W1-like (Kafka consume + Redis lookup + HTTP notify per delivery). The W1 control rules out the worst-case attribution; we extrapolate the Node.js-controlled gap for W2 to be roughly W2_observed_gap × (W1_language_controlled / W1_uncontrolled) ≈ 13× × (56/93) ≈ ~8×, which is still a meaningful architectural win even before measurement. We did not run the direct measurement.
- **W3** is Java on the baseline side (Kafka Streams is JVM-only). A Java vs Java comparison doesn't exist for this workload — the Harper side is Node.js, the Kafka side is Java, and a Node.js Kafka Streams equivalent does not exist as a production technology. The W3 result is what it is; the language confound is not removable here.
- **W4** is so I/O-bound that language is unlikely to matter at the rates we test. Producer and consumer are both effectively passing bytes between TCP sockets; the architectural cost is in the pipeline shape, not the language. We accept that as a methodology choice.

A community-contributed Node.js K+R+P W2/W4 implementation would close the remaining language-confound concerns on those workloads. We invite it via the open repo.

8.5 No external named Kafka SME pre-publication review

A named, independent Kafka SME reviewing this methodology and code before publication is the single highest-leverage credibility move that *isn't* in place. We disclose this explicitly in App. A. The repo is open and the issue tracker is the standard place to surface critique; we will incorporate any methodology-level finding that holds up.

8.6 Cross-validation status

Real Linux multi-node hardware is the explicit next step before any absolute-throughput claim. See §9.

How to reproduce

```
git clone https://github.com/HarperFast/kafka-v-harper-perf-test
cd kafka-v-harper-perf-test

# Bring up the conventional stack (Kafka × 3, PG primary+replica, Redis × 3, … )
( cd infrastructure/baseline && ./scripts/up.sh --seed 10000 )

# Bring up the Harper Pro cluster (3 nodes, replicated, mesh auto-joined)
( cd infrastructure/harper && ./scripts/up.sh )

# Build the harness and run W1 against both stacks (and the Node.js W1 control)
go build -o /tmp/sc-runner ./harness/runner/
./scripts/campaign-w1.sh
```

Each run writes raw observations and a summary to `results/W*/<run_id>/`. The summary includes p50/p90/p95/p99/p999/max latencies and delivered %. The full methodology is in `docs/methodology.md`; the per-decision rationale is in `docs/decisions.md`.

Real-hardware cross-validation — see `docs/running-on-cloud.md`. The same compose files and scripts run unchanged on Linux. A single 8-vCPU NVMe Linux box (e.g. AWS `c7g.2xlarge`, ~few hours of cloud spend) confirms or refutes the laptop ratios on real disk; a 3-node Linux cluster lets W4 actually run at production-Kafka rates. We invite both contributions; the contribution workflow is in the running-on-cloud guide.

Conclusion

In this laptop-VM whole-stack comparison, a single Harper cluster did the work of five specialized systems with lower end-to-end latency on three of four real-time application pipelines, on identical hardware, with per-workload durability matched, and with the Kafka pipelines configured with RF=3, min.insync.replicas=2, acks=all where applicable, lz4 compression, and batching (App. B). These are end-to-end application-latency results for the pipelines as implemented, not component-level Kafka benchmarks.

The exception is W3 point-query, where the Kafka Streams Interactive Queries pipeline showed lower latency — it is architecturally designed to serve that access pattern and the result reflects that. The decision flow in §2 turns this into actionable architecture guidance: pick a consolidated runtime if your app's dominant patterns are durable-write-and-notify, real-time messaging, or live aggregates; pick Kafka + Kafka Streams if your dominant access pattern is sub-millisecond point reads against live aggregate state; pick Kafka alone if your dominant workload is production-scale sustained log ingestion; and use both when distinct workloads each suit a different tool.

The cross-validation step in §9 is where this work goes next. Community runs on real Linux hardware will tell us how the laptop ratios shift at production scale — and we expect, openly, that the W4 firehose result will narrow significantly on real NVMe (Kafka's home turf) and that the W1/W2/W3-freshness results will hold or widen (their gaps are architectural, not bandwidth-bound). Either finding is publishable; we will report what we see.

Appendix A — Conflict-of-interest disclosure and mitigations

The author is affiliated with Harper, Inc. The mitigations in place:

MITIGATION	STATUS	NOTES
All four workloads pre-committed before any runs	ACTIVE	`docs/initial-plan.md`, recorded 2026-05-22
Publish all results regardless of outcome	ACTIVE	W3 point-query — a Kafka win — is in the headline table
Methodology and decisions logged in an append-only public record	ACTIVE	`docs/decisions.md`, ~15 entries
Kafka pipelines configured idiomatically	ACTIVE	Producer + consumer + broker settings recorded; the `batch=1` failure is documented
Per-workload durability matching	ACTIVE	Harper does not skip durability work the baseline does
Whole-stack comparison, never bare Kafka	ACTIVE	Every workload compares Harper to the whole conventional stack
Reproducibility	ACTIVE	Open repo, Apache 2.0 license, pinned Harper Pro image digest
Language-confound control for the headline workload	ACTIVE	The Node.js K+R+P W1 secondary run (§6.5); narrowed the W1 claim by 1.7×
External named Kafka SME pre-publication review	OPEN	Not yet done; encouraged before broader publication
Cross-validation on real Linux multi-node hardware	OPEN	Community-contribution workflow in §9 / `docs/running-on-cloud.md`

Appendix B — Full configuration

Kafka cluster

3 brokers, host-exposed on ports 19920–19929, internal listeners on the `streaming-compare-baseline` Docker network.

```
KAFKA_DEFAULT_REPLICATION_FACTOR=3
KAFKA_MIN_INSYNC_REPLICAS=2
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR=3
KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR=3
KAFKA_TRANSACTION_STATE_LOG_MIN_ISR=2
KAFKA_HEAP=-Xms1g -Xmx1g
KAFKA_MEM_LIMIT=2500m   # ~1.5 GB for page cache
KAFKA_AUTO_CREATE_TOPICS_ENABLE=false
```

Topics: 12 partitions, RF=3 (auto-created at first reset; see `scripts/reset.sh`).

Kafka producers (W2, W4)

```
kafka.Writer{
  RequiredAcks: kafka.RequireAll,
  Compression:  kafka.Lz4,
  BatchTimeout: 5 * time.Millisecond,
  BatchSize:    500,
  Async:        false,
}
```

Kafka consumer (W4)

```
CONSUMER_BATCH_SIZE=256
CONSUMER_BATCH_MAX_AGE=50ms
HTTP_TIMEOUT=500ms
W4_NOTIFY_SAMPLE_RATE=1.0
```

Kafka Streams (W3)

```
producerProps.put(ProducerConfig.ACKS_CONFIG, "1");
producerProps.put(ProducerConfig.LINGER_MS_CONFIG, "5");
producerProps.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");
producerProps.put(ProducerConfig.BATCH_SIZE_CONFIG, "32768");
```

Stream threads: `${W3_NUM_STREAM_THREADS:-4}`. Window size: 5000 ms. Grace: 1000 ms.

Postgres

Primary + streaming replica. `max_connections=200` on both (recovery requirement). `synchronous_commit=on` (primary fsync). No `synchronous_standby_names` → standby is asynchronous.

Redis

3-node cluster, 1024 MB per node, sized for the methodology's 10 M-key max state cardinality.

Harper cluster

3 nodes, `harperfast/harper-pro@sha256:b5caf7621e7062212b9124eccaccdcfae7912d05a84aebd7b0dd64fc2d9ba337` (multi-arch). Replication over `wss://` on port 9933, full-mesh via `add_node`. Each node: 2.6 vCPU / 4.6 GB memory limit. `REPLICATION_MTLS_CERTIFICATEVERIFICATION=false` (self-signed cluster certs).

Harper component-level durability

W2 / W4 Harper component code calls:

```
this.getContext().replicatedConfirmation = 1;
await tables.W4Event.put( ... );
```

before the write, causing the handler to block until 1 peer has durably applied the write (2 total durable copies = ISR=2 analog).

Harness

Open-loop producer at the target rate; in-process observer on `:18090`; HdrHistogram aggregation; raw observations + meta written per-run.

Appendix C — Detailed per-workload result tables

W1 · 50 EVENTS/S × 15 S, 3 PAIRED RUNS EACH

STACK	P50 (MS)	P95 (MS)	DELIV. %	NOTES
Kafka + PG + Debezium + Redis (Go glue)	876	3 890	84	Historical; p95 recomputed from raw.csv; ratio with Harper = 93×
Kafka + PG + Debezium + Redis (Node.js glue)	523	860	84	Language-controlled; ratio with Harper = 56×
Harper	**9.4**	**21.8**	84	Headline

Within-stack range: Go p50 670–4 026 ms (one outlier); Node.js p50 421–553 ms; Harper p50 9.2–12.1 ms. Full per-run data in `results/W1/_secondary-comparison.json`.

W2 · 10 EV/S × 15 S, 3 PAIRED RUNS, ~30 FAN-OUT/EVENT

STACK	P50	P95	P99	DLV%
Kafka + Redis	165	2 990	3 270	100
Harper	**12.5**	**18.5**	23.8	100

W3 FRESHNESS · 100 EV/S × 30 S, 100 KEYS × 5 S WIN (S)

STACK	P50	P95	P99	N
Kafka + KStreams	3.65	13.6	13.7	496
Harper	**1.05**	**1.56**	2.08	497

W3 QUERY · 20 QPS × 30 S, 1 CLEAN RUN EACH (MS)

STACK	P50	P95	P99	N
KStreams IQ	**2.75**	**68.4**	135.8	259
Harper	17.4	130.6	254.1	600

W4 · 1 000 EV/S × 30 S, CLEAN RUN REPORTED (MS)

PIPELINE	P50	P95	P99	DLV%
Kafka pub/cons (RF=3)	57	304	469	100
Harper (MQTT, 3 nodes)	**1.4**	**3.5**	7.7	100

Of the two Kafka runs at 1 000 events/s, one delivered 100 % at 57 ms p50 (reported) and one delivered 99 % at 518 ms p50; both Harper runs delivered 100 % at 1.4–1.45 ms p50. The W4 ascending probe (1 000 / 2 500 / 5 000 / 10 000 events/s, two runs per rate per pipeline) is in `results/W4/` and visualized in Figures 6 and 7. As stated in §7 and §8.2, this is a local live-delivery result, not a Kafka throughput result.

Appendix D — Reading the data yourself

Every run writes to `results/<workload>/<run_id>/`:

```
results/W1/20260528-220857/
├── raw.csv      # one row / measurement
└── meta.json    # config + percentiles
```

```
# meta.json top-level keys
workload_id, run_id, stack,
target_rate_hz, duration_secs,
warmup_secs, latency_p50_ns …
latency_max_ns, producer_emitted,
observer_seen, joined_count, knobs,
host_info, git_sha, started_at, ended_at
```

W3 runs additionally have `w3_freshness` and `w3_query` sub-objects with per-axis percentiles. Loading and aggregating all runs in five lines of Python:

```
import json, glob, pandas as pd
rows = [json.load(open(f)) for f in glob.glob("results/W*/*/meta.json")]
df = pd.DataFrame(rows)
df["p50_ms"] = df["latency_p50_ns"] / 1e6
print(df.groupby(["workload_id", "stack"])["p50_ms"].median())
```

References

- Gil Tene. *How NOT to measure latency*. Strangeloop / GOTO conference talks, 2013–2015. The canonical treatment of open-loop measurement and coordinated omission.
- HdrHistogram (Gil Tene). github.com/HdrHistogram/HdrHistogram — log-bucketed percentile aggregation.
- Apache Kafka documentation. kafka.apache.org/documentation — producer/consumer tuning, replication semantics, Kafka Streams.
- Apache Kafka Streams Interactive Queries. kafka.apache.org/documentation/streams/developer-guide/interactive-queries.html
- Harper documentation. docs.harperdb.io — RocksDB. rocksdb.org — the storage engine underneath both Harper 5.0+ and Kafka Streams' state stores.
- Debezium. debezium.io — the change-data-capture connector used in the W1/W2 conventional stacks. · `kafkajs`. kafka.js.org — the Node.js Kafka client used in the W1 language-control implementation.

REPRODUCIBILITY STATEMENT

All code, configurations, and per-run raw observations are in the public repository at github.com/HarperFast/kafka-v-harper-perf-test. Every methodology choice is recorded in `docs/decisions.md`; every container image is referenced by a pinned digest in `.env.example`. The repository's `CITATION.cff` makes the work citable from any reading list.

OPEN FOR REVIEW

This study is open to community scrutiny. Methodology audits, Kafka-tuning suggestions, and cross-validation runs on real Linux hardware are explicitly invited via the repository's issue tracker and pull-request workflow. The single strongest mitigation against the author's Harper affiliation is having Kafka-side practitioners audit the work in public; the repository is set up exactly to receive that input.

harper

Conventional Kafka-centered stacks vs a single Harper cluster — end-to-end latency on four real-time application pipelines.
Aleks Haugom, Harper, Inc. May 2026.