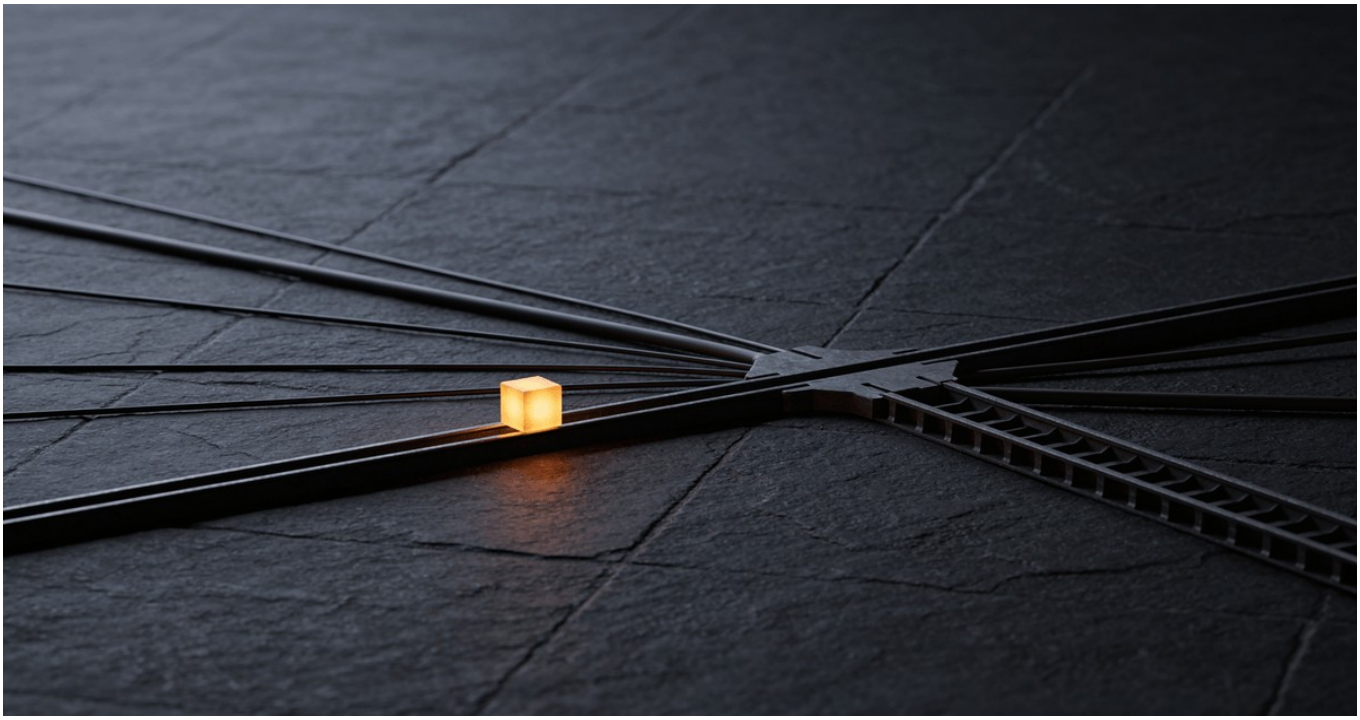
Choosing an Event Mechanism for a Data Platform

Julien Laurenceau | Published 2026-09-30 | Updated 2026-10-21

https://pepitedata.com/choosing-an-event-mechanism/

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



An event mechanism is a contract about delivery, ordering, retention, replay and failure. Product choice comes after that contract. Teams get into trouble when they choose Kafka, RabbitMQ or PostgreSQL first, then discover that the application needed different semantics.

I start with the lightest mechanism that meets the measured requirement. That can be server-sent events for browser updates, PostgreSQL for local signaling or change capture, a queue for work distribution, or a durable log for replay and multiple independent consumers.

Write the requirements before comparing products

Write the delivery requirements before naming a product, because a mechanism comparison only means something against a fixed contract. The questions below pin down who produces and consumes each event, how long a consumer may be offline, whether every consumer must see every event or one worker claims each item, and what ordering, duplicate, rate and outage behavior the application can live with. Answer them in writing, then compare.

- Who produces and consumes the event?
- Can a consumer be offline, and for how long?
- Must every consumer see every event, or should one worker claim each item?
- Do consumers need replay from an offset or point in time?

- What ordering scope matters: global, per key, per partition or none?
- What duplicate behavior can the application tolerate?
- What are the expected and peak event rates, payload sizes and retention periods?
- What happens when the mechanism is unavailable?

A practical mechanism comparison

Compare event mechanisms on semantics rather than throughput: which job each one fits, what its durability and replay model promises, and the trap most likely to bite in production. The eight options below range from HTTP SSE for one-way browser updates to Apache Kafka and AutoMQ for durable logs, and capacity depends on workload, configuration, hardware and failure assumptions, not on the name.

Mechanism	Strong fit	Durability and replay	Main trap
HTTP SSE	One-way browser updates	Reconnect support, but durable replay is application work	Treating a client stream as a system of record
PostgreSQL NOTIFY	Lightweight signaling near existing transactions	No offline consumer backlog	Sending business payloads instead of a row key
PostgreSQL logical decoding	Change data capture from committed database writes	Replay through replication slots and consumer offsets	Unconsumed WAL retention and OLTP coupling
RabbitMQ quorum queue	Work distribution, routing, acknowledgments and dead lettering	Replicated durable queue, replay is not its primary model	Ignoring publisher confirms, consumer acknowledgments or poison messages
RabbitMQ stream	Append-only retention and replay inside RabbitMQ	Offset-based consumption with retention	Assuming queue and stream semantics are identical
Apache Kafka	Durable event logs, long retention and independent consumer groups	Partition offsets and replay	Claiming end-to-end exactly-once behavior outside its transactional boundary
Apache Pulsar	Multi-tenant messaging with stateless brokers and BookKeeper storage	Persistent subscriptions and retained data	Underestimating the number of operating components
AutoMQ	Kafka-compatible workloads designed around object storage	Kafka APIs with a different storage architecture	Accepting vendor efficiency claims without a workload-specific pilot

This is a semantics comparison, not a throughput ranking. Capacity depends on workload, configuration, hardware and failure assumptions.

SSE is a client delivery channel, not a broker

SSE is a client delivery channel, not a broker: the browser `EventSource` API opens a persistent one-way HTTP connection. It reconnects after interruption, and the event format supports an ID that a server can use to resume from a known point. The transport does not create a durable event log for you. If an offline client must recover every event, the application needs retained state and a replay endpoint behind the stream.

SSE is a good fit for dashboards, progress updates and notifications when the server is the only sender. Check proxy buffering, idle timeouts and HTTP connection limits before production. If the client also needs to send a continuous stream, choose a bidirectional protocol.

PostgreSQL offers two different event paths

PostgreSQL offers LISTEN and NOTIFY for lightweight signaling and logical decoding for change data capture, and the two paths promise different things. NOTIFY is a signal that tells a connected process which durable row to read, while logical decoding streams committed changes through logical replication slots so the consumer can resume from its position. Pick the path by whether consumers need replay and resume or only notification that something committed.

LISTEN and NOTIFY for signaling

`NOTIFY` delivers after the transaction commits. PostgreSQL can fold duplicate notifications with the same channel and payload inside one transaction, and the default payload must be shorter than 8,000 bytes. I use it as a signal that tells a connected process which durable row to read. I do not use it as the durable record of a business event.

Logical decoding for change data capture

Logical decoding streams committed database changes through logical replication slots. It is a stronger foundation for CDC because the consumer can resume from its position. It also creates an operational obligation. A stalled slot can retain WAL and consume disk until the consumer catches up or the slot is managed. Monitor slot lag as part of database capacity, not only event-pipeline health.

RabbitMQ queues and streams solve different jobs

RabbitMQ queues and streams solve different jobs: a quorum queue is a replicated work queue based on Raft for competing consumers, acknowledgments, routing, dead lettering and poison-message handling, while a RabbitMQ stream is an append-only retained log with offset-based replay. The same broker can support both, but choosing the wrong data structure changes delivery and retention behavior.

I would choose on semantics before estimating throughput. A queue should answer who owns the next unit of work and what happens after a failed delivery. A stream should answer how long records remain, how consumers address offsets and how replay affects downstream idempotency.

Kafka exactly-once claims need a boundary

Kafka exactly-once claims need a boundary, and Kafka provides idempotent production and transactions. Its documentation supports exactly-once processing when an application reads from Kafka, processes records and writes results back to Kafka with transactions and read-committed consumers. A write to an unrelated database, API or object store requires cooperation from that destination or application-level idempotency. The default Kafka delivery model is at least once.

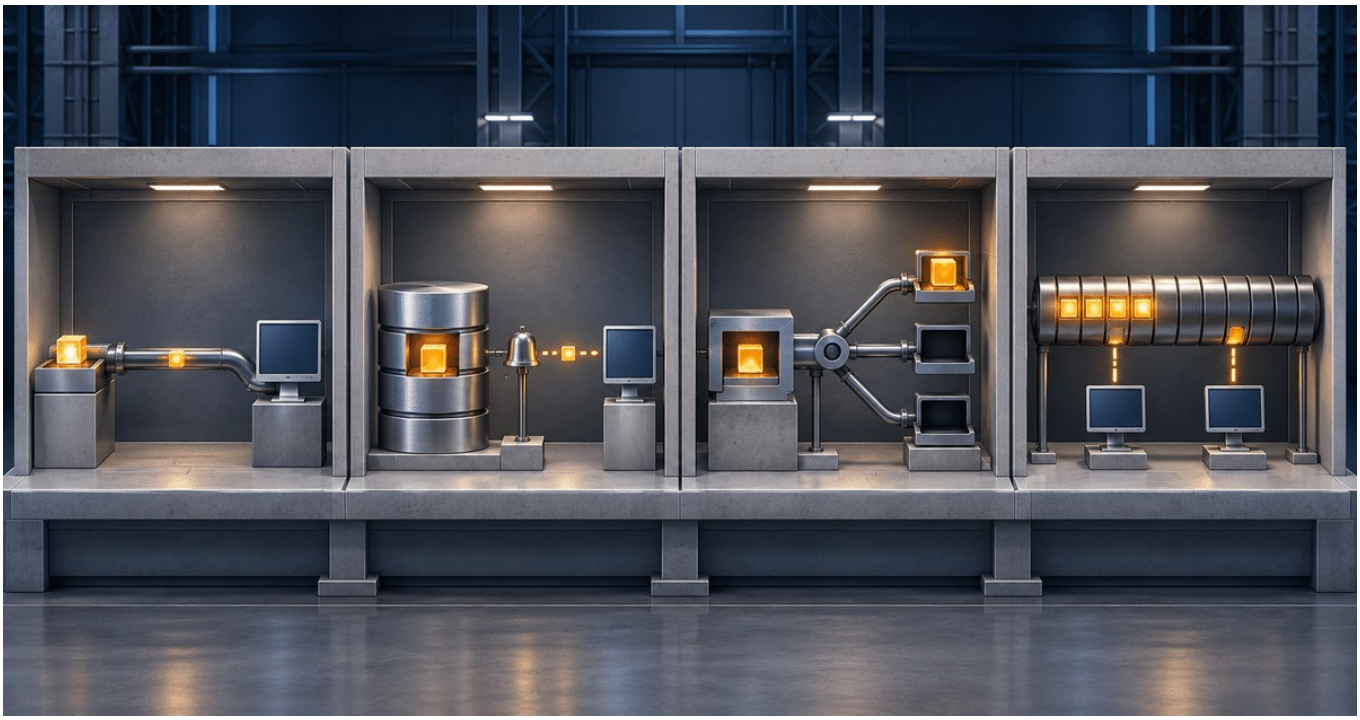
That scope belongs in the architecture decision record. The phrase "Kafka gives exactly once" is too broad to test.

Pulsar and AutoMQ change the storage architecture

Pulsar and AutoMQ change the storage architecture: both keep client traffic on brokers while persistent data moves somewhere else. Pulsar uses stateless brokers for client traffic and Apache BookKeeper bookies for persistent storage. That separation supports flexible placement and multi-tenancy, but the platform still has brokers, bookies and metadata coordination to operate.

AutoMQ keeps Kafka-compatible APIs while replacing Kafka's native log storage with its S3Stream layer. Object storage becomes the primary repository and a write-ahead log absorbs latency-sensitive writes. That can change retention economics and reassignment behavior. I would validate those benefits against the actual cloud, object store, failure model and workload rather than copying the vendor's

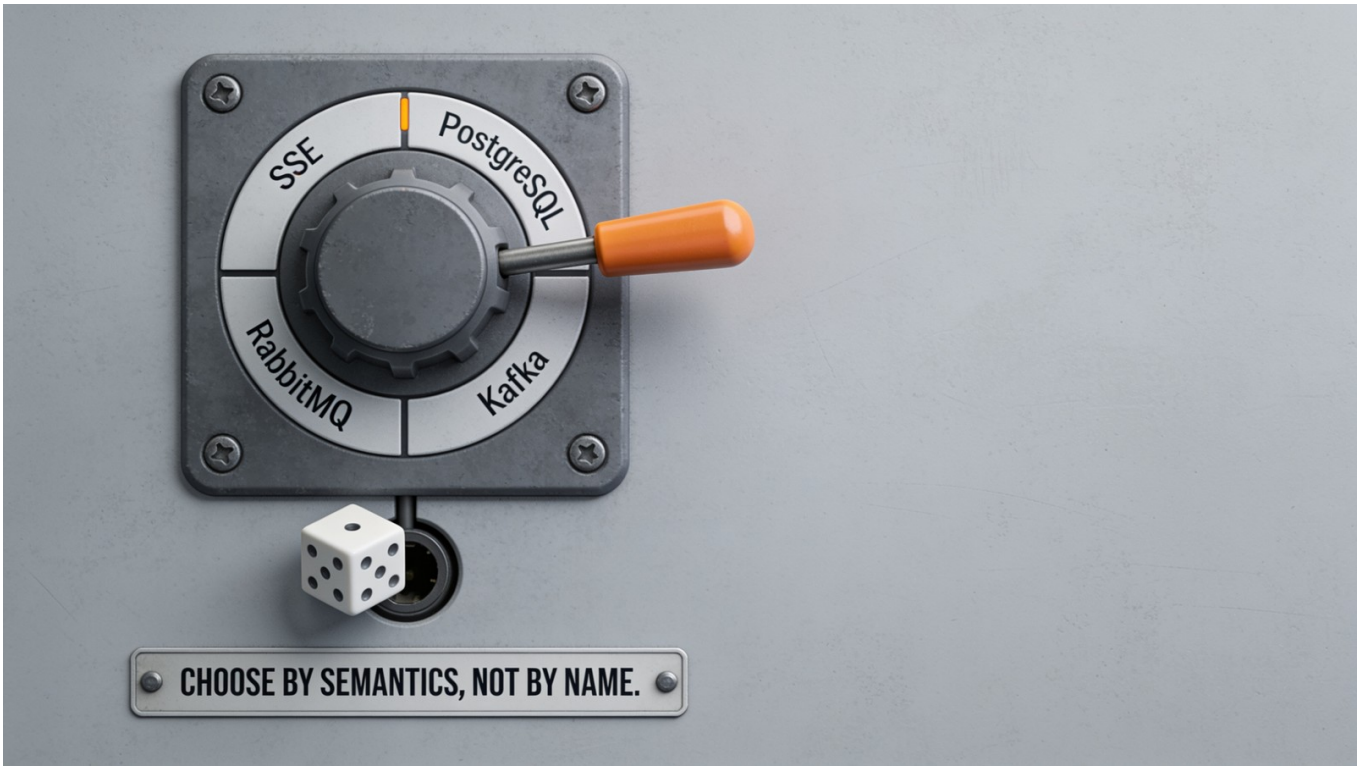
headline ratios into a capacity plan.



Four representative contracts: live delivery, transaction signals, work distribution and retained logs for replay.

A real-time transport is a separate decision

A real-time transport is a separate decision from a durable event platform: control loops and device links need a transport protocol chosen on latency and link behavior, while business events need retention and replay. The two are different architecture layers, so keep the low-latency path separate from the retained business event path and define how one feeds the other.



Pick the lightest mechanism whose semantics cover the requirement, not the heaviest by name.

Some control loops and device links need a transport protocol rather than a durable event platform. That is a different architecture layer. Keep the low-latency path separate from the retained business event path, then define how one feeds the other. I cover that boundary in Which Message Broker for IoT and Real-Time Control.

Pilot the failure semantics, not only throughput

Pilot the failure semantics before you trust any throughput number, because the failures are where the mechanisms differ. A useful pilot stops a consumer until a realistic backlog builds, kills a producer between send and acknowledgment to expose duplicates, restarts a broker under sustained writes, and checks what operators must do to diagnose lag and recover service. Throughput measured on the happy path is the least informative number in the set.

- Stop a consumer long enough to create a realistic backlog, then measure catch-up time and storage growth.
- Kill a producer after send but before acknowledgment, then inspect duplicates.
- Restart a broker or database node during sustained writes.
- Inject skew so one key or partition receives a disproportionate share of traffic.
- Measure p50, p95 and p99 end-to-end latency with the required durability settings enabled.
- Test retention expiry, replay and downstream idempotency.
- Record the operator steps needed to diagnose lag and recover service.

The practical takeaway

Choose the contract first. SSE pushes updates to connected clients. PostgreSQL can signal or expose committed changes. RabbitMQ can distribute work or retain streams. Kafka and Pulsar provide durable logs for independent consumers. AutoMQ changes Kafka's storage model. None is the default answer for every event.

If an event-platform choice will set your reliability, latency or infrastructure cost for years, I can review the requirements, failure semantics, benchmark plan and capacity assumptions through a Flash Architecture Review or Data Platform Audit. Book a 15-min intro call to discuss the decision.

FAQ

What is the first decision when choosing an event mechanism?

The delivery contract, before any product name: at-most-once, at-least-once or exactly-once within a named boundary; what durability a successful acknowledgment promises; and whether consumers need replay from history or only live tailing. Those three answers eliminate most of the field before a benchmark runs.

Is server-sent events a message broker?

No. Server-sent events is a client delivery channel: one-way HTTP push to connected browsers or clients, with no persistence model, no consumer groups and no replay history. It solves fan-out to live clients and pairs naturally with a real broker or database behind it for durability.

When does PostgreSQL beat a dedicated broker?

When the signaling volume and fan-out are modest and the data already lives in PostgreSQL. LISTEN and NOTIFY is enough for wake-up signals, and logical decoding turns committed changes into a change stream for downstream consumers. Move to a broker when consumers need independent replay, sustained high throughput, or isolation from the primary database's workload.

Does exactly-once delivery remove the need for idempotent consumers?

No. Exactly-once claims are bounded by the system's own transaction boundary; side effects outside that boundary (writes to another store, calls to another service) can still be repeated after a failure and retry. Idempotent consumers or deduplication keys remain necessary, and the boundary itself must be named in writing.

Related posts

- Which message broker for IoT and real-time control

PDF snapshot of the public article. The online version is authoritative; links remain clickable.