

Event Driven

Aayush Bajaj

July 10, 2026

Contents

1	events, commands, queries	1
2	pub/sub and brokers	2
2.1	two semantics, one word	2
2.2	consumer groups and partitions	2
3	delivery guarantees, honestly	3
3.1	idempotency keys	3
4	ordering and partitioning	3
5	event sourcing	3
5.1	state = fold(events)	3
5.2	a worked bank account	4
6	cqrs	6
7	schema evolution	6
8	backpressure	6
9	see also	7

event-driven architecture inverts the direction of knowledge: instead of the caller knowing who must react, the *reactor* knows what it cares about. producers announce facts; consumers subscribe. the whole style is the **observer pattern** with a broker in the middle and a network underneath — which is exactly where the interesting failure modes come from.

1 events, commands, queries

three message species, constantly confused:

- **event**: a fact about the past, named in past tense — **OrderPlaced**, **PaymentFailed**. immutable, owned by the producer, zero expectation about who (if anyone) reacts. broadcasting is safe because nothing is being asked.
- **command**: a request for the future — **PlaceOrder**. imperative mood, addressed to exactly one handler, can be rejected. commands express intent; events record outcome. compare the **command pattern**, which reifies exactly this.
- **query**: a question, side-effect free, wants an answer now.

the classic smell is the *command in event's clothing*: `SendWelcomeEmailRequested` published as an “event” that exactly one consumer must process, or the producer silently depends on a specific reaction — you have built rpc with extra steps and none of rpc’s error handling.

the test: delete every consumer. if the producer's correctness suffers, it was a command.

2 pub/sub and brokers

2.1 two semantics, one word

“message broker” covers two genuinely different machines, and most design errors come from expecting one to behave like the other:

dimension	queue semantics (rabbitmq lineage)	log semantics (kafka lineage)
message lifetime	deleted once acked	retained for a period, regardless
consumer model	competing consumers on one queue	consumer groups over partitions
replay	gone is gone	rewind offset, re-read history
routing	rich (exchanges, topics, keys)	dumb (topic + partition key)
ordering	per-queue-ish, weakened by retries	strict per partition
natural fit	work distribution, tasks	streams, integration, event sourcing

- **queue (rabbitmq-style)**: the broker actively pushes work to competing consumers; an unacked message is redelivered (possibly to another consumer); an acked message is destroyed. the broker tracks per-message state.
- **log (kafka-style)**: the broker is a dumb append-only file per partition. consumers *pull*, and the only per-consumer state is an integer — the **offset**, “i have processed everything before position n ”. committing an offset is a consumer-side act, wholly decoupled from message lifetime — which is what makes replay, late-joining consumers and reprocessing-after-a-bugfix possible.

the log is the same idea as a database write-ahead log promoted to a public interface

2.2 consumer groups and partitions

- a topic is split into p **partitions**; records with the same **key** hash to the same partition, so ordering is guaranteed per key, never per topic.
- a **consumer group** divides partitions among its members: each partition is owned by exactly one consumer in the group at a time. two groups reading the same topic each see everything — pub/sub and work-sharing from one primitive.
- consequence: parallelism is capped at p . a topic with 6 partitions feeds at most 6 active consumers per group; the 7th idles. choose keys to spread load, and expect **hot partitions** when one key (one big customer) dominates.

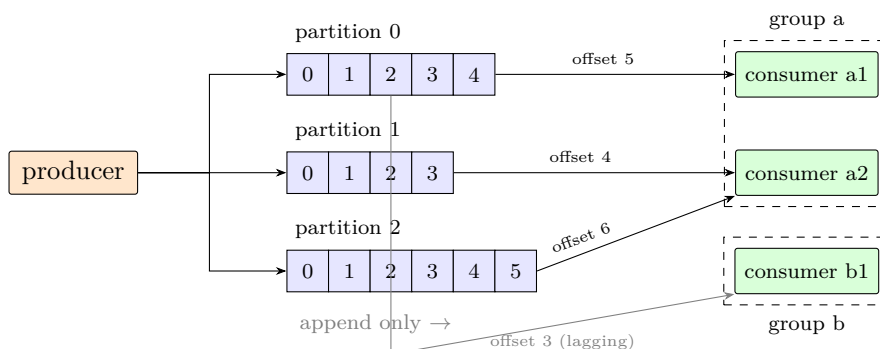


Figure 1: log-style broker topology: keyed records append to partitions; each consumer group tracks its own offsets, so group b can lag or replay without affecting

group a.

3 delivery guarantees, honestly

the guarantee is a property of the *loop* (receive, process, acknowledge), not of the broker box on the diagram:

- **at-most-once**: ack (or commit the offset) *before* processing. crash mid-process and the message is lost. acceptable for metrics ticks, disastrous for money.
- **at-least-once**: ack *after* processing. crash between processing and ack, and the message is redelivered — every consumer will eventually see duplicates. this is the default posture of every serious system.
- **exactly-once**: as a transport guarantee across arbitrary systems, this does not exist — the ack itself can be lost, and the broker cannot know whether your side effect happened. what you can build is **exactly-once processing**: at-least-once delivery plus deduplication at the consumer.

kafka's 'exactly-once semantics' is real but scoped: transactions spanning consume-process-produce within kafka, not across your smtp server

3.1 idempotency keys

- producer stamps each logical operation with a unique key (uuid, or a hash of the business operation).
- consumer records processed keys — ideally in the *same transaction* as the side effect — and drops any message whose key it has seen.
- the effect: $f(f(x)) = f(x)$; retries become free. naturally idempotent operations (**set balance = 120**, upserts) need no table; non-idempotent ones (**increment balance**, send email) always do.
- the producer-side twin is the outbox pattern from **microservices**: local transaction + relay gives at-least-once *publication*; idempotency keys give at-most-once *effect*. compose both and the whole pipeline is exactly-once in effect.

4 ordering and partitioning

- global order across a distributed topic is a fiction you pay for with throughput (one partition = one writer's ordering = one consumer's parallelism). the honest contract: **order per key, concurrency across keys**.
- design events so per-key order is all you need: all events for account 42 share key 42, and no invariant spans accounts 42 and 43 within the stream.
- consumers must still tolerate *interleaving* anomalies: a rebalance can hand a partition to a new consumer that re-reads uncommitted records (duplicates again), and cross-partition joins see arrival order, not event-time order. carry timestamps in the payload and reason in event time when it matters.

5 event sourcing

5.1 state = fold(events)

instead of storing current state and losing history, store the history and derive the state. if $e_1, e_2, \dots, e_n$ is the append-only log and a the apply function,

$$s_n = a(s_{n-1}, e_n), \quad s_0 = \text{empty}, \quad \text{state} = \text{fold}(a, s_0, [e_1, \dots, e_n]). \quad (1)$$

the database’s own recovery machinery works exactly this way — replaying a write-ahead log to reconstruct state (Silberschatz, Abraham and Korth, Henry F. and Sudarshan, S., 2019) — event sourcing just makes the log the system of record and the “current state” a cache.

- **you gain:** a perfect audit trail (the ledger *is* the data), time travel (fold a prefix), retroactive bug fixes (fix the fold, replay), new read models for free (new fold over old history).
- **you pay:** schema evolution forever (old events never die), snapshotting machinery once logs get long (fold from the last snapshot, not from genesis), and the up-front cost that queries against “current state” all need a projection.

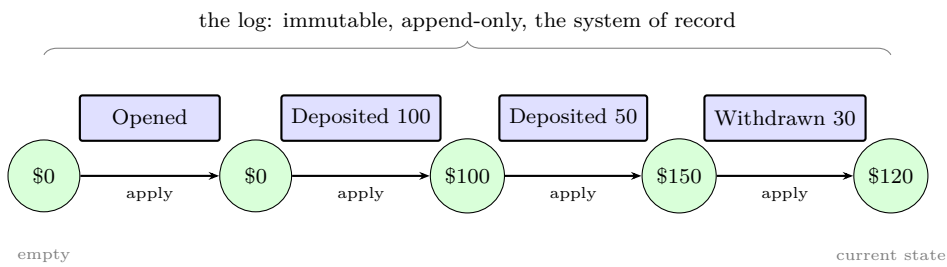


Figure 2: the fold timeline: each event maps the previous state to the next; any prefix of the log is a legitimate historical state.

5.2 a worked bank account

the whole idea in fifty lines: commands validate against folded state, then append facts; state is always derived, never stored.

```

1  from dataclasses import dataclass
2
3  # --- events are facts: past tense, immutable ---
4  @dataclass(frozen=True)
5  class Opened:      owner: str
6  @dataclass(frozen=True)
7  class Deposited:   amount: int
8  @dataclass(frozen=True)
9  class Withdrawn:   amount: int
10
11 # --- state = fold(apply, events, empty) ---
12 @dataclass(frozen=True)
13 class Account:
14     owner: str = ""
15     balance: int = 0
16
17 def apply(state, event):          # the fold step: (state, event) ->
18     ↪ state
19     match event:
20         case Opened(owner):      return Account(owner, 0)
21         case Deposited(amount):  return Account(state.owner, state.balance +
22         ↪ amount)
23         case Withdrawn(amount): return Account(state.owner, state.balance -
24         ↪ amount)
25
26 def replay(events):
27     state = Account()

```

```

25     for e in events:
26         state = apply(state, e)
27     return state
28
29 # --- the command side: validate against current state, append new facts ---
30 class EventStore:
31     def __init__(self):
32         self.log = []                                # the only mutable thing in the
33         ↪ system
34
35     def state(self, upto=None):
36         return replay(self.log[:upto])
37
38     def execute(self, command, arg):
39         s = self.state()
40         if command == "withdraw" and arg > s.balance:
41             raise ValueError(f"insufficient funds: balance {s.balance}, asked
42             ↪ {arg}")
43         event = {"open": Opened, "deposit": Deposited, "withdraw":
44         ↪ Withdrawn}[command](arg)
45         self.log.append(event)
46         return event
47
48 store = EventStore()
49 for cmd, arg in [("open", "aayush"), ("deposit", 100), ("deposit", 50),
50 ↪ ("withdraw", 30)]:
51     e = store.execute(cmd, arg)
52     print(f"appended {e!r:<22} -> state {store.state()}")
53
54 try:
55     store.execute("withdraw", 1000)
56 except ValueError as err:
57     print(f"command rejected: {err}")
58
59 print("\ntime travel - state after each prefix of the log:")
60 for i in range(len(store.log) + 1):
61     print(f"  events[:{i}] -> {store.state(upto=i)}")

```

```

appended Opened(owner='aayush') -> state Account(owner='aayush', balance=0)
appended Deposited(amount=100) -> state Account(owner='aayush', balance=100)
appended Deposited(amount=50) -> state Account(owner='aayush', balance=150)
appended Withdrawn(amount=30) -> state Account(owner='aayush', balance=120)
command rejected: insufficient funds: balance 120, asked 1000

```

```

time travel - state after each prefix of the log:
events[:0] -> Account(owner='', balance=0)
events[:1] -> Account(owner='aayush', balance=0)
events[:2] -> Account(owner='aayush', balance=100)
events[:3] -> Account(owner='aayush', balance=150)
events[:4] -> Account(owner='aayush', balance=120)

```

note what fell out for free: the rejected overdraft never touched the log (commands are validated, events are facts), and “time travel” is just folding a prefix — this is the **memento pattern** where the mementos happen to be the data itself. a production store adds snapshots (persist s_k every k events, replay from there) and optimistic concurrency (append fails if the log grew since you read it).

6 cqrs

command query responsibility segregation: split the write model from the read model(s).

- the **write side** accepts commands, enforces invariants, emits events — normalised, consistent, small.
- the **read side** subscribes to those events and maintains denormalised **projections**: one table shaped exactly like each screen or query. no joins at read time; the join happened at projection time.
- the price is **eventual consistency** between the sides: a user can write and then read a stale projection. mitigations — read-your-own-writes from the write side, or return the new state in the command response.
- cqrs does not require event sourcing (a plain db plus change-events works) and event sourcing does not require cqrs — but they compose naturally: the fold above *is* a projection, and every projection is just another fold with a different apply function.
- adopt asymmetrically: most systems need cqrs on two or three hot aggregates, not everywhere. full-system cqrs + event sourcing is an architecture astronautics smell — see [code smells](#) for the in-process analogues.

7 schema evolution

events outlive the code that wrote them; a five-year-old **OrderPlaced** v1 must still fold correctly today.

- **compatibility direction**: backward compatibility = new code reads old events (mandatory for event sourcing — history is forever); forward compatibility = old code reads new events (needed during rolling deploys, when producers upgrade first).
- safe changes: add optional fields with defaults; never rename, never retype, never reuse a field number/name for a different meaning.
- mechanics: schema'd formats (avro/protobuf/json-schema) plus a **schema registry** that rejects incompatible producer schemas at publish time — turning a 3 a.m. consumer crash into a ci failure.
- when a change is genuinely breaking: **upcasters** (translate v1 to v2 at read time, chainable) or publish to a new versioned topic and migrate consumers deliberately.

8 backpressure

producers and consumers are decoupled in time, not in arithmetic: if arrival rate exceeds service rate for long enough, the queue grows without bound — the broker just decides *where* the pile of unprocessed work sits.

- **bounded queues everywhere**: an unbounded buffer converts overload into an out-of-memory crash at maximum distance from the cause. bound the queue and choose a policy — block the producer (backpressure proper), shed load (drop with a metric), or spill to disk (a log broker's native move).

- **pull beats push under load:** pull-based consumers (kafka-style) take work at their own pace, so pressure surfaces as consumer **lag** — one number (log head minus committed offset) that tells you precisely how far behind reality each group is. alert on lag, not on cpu.
- **slow-consumer handling:** scale consumers up to the partition cap, then scale partitions; make handlers faster (batch the db writes); or admit the stream has two speeds and split it (hot path folds counters, cold path does the heavy enrichment).
- reactive-streams credit protocols (consumer grants n message credits upstream) are the same idea formalised in-process.

9 see also

- [microservices](#) — sagas and the outbox pattern, the transactional glue for all of this
- [api design](#) — the synchronous counterpart: request/response contracts and web-hooks
- [design patterns](#) — observer, command and memento, before the network got involved
- [databases](#) — write-ahead logs, transactions and the machinery event stores borrow
- [lan messenger](#) — a from-scratch pub/sub-ish messaging system over raw sockets

references

Silberschatz, Abraham and Korth, Henry F. and Sudarshan, S. (2019). *Database System Concepts*, McGraw-Hill Education.