What's the difference between a message queue and an event log?

A message queue delivers work items once to workers, then discards them; an event log records an immutable, ordered history of events that multiple consumers can read and replay independently.[1] [2] [3]

Core Concept

- **Message queue**: A **work distribution** mechanism where producers send messages to a queue and consumers pull and process them; once processed, the message is typically removed and not seen again.[4] [5] [1]
- **Event log / event stream / distributed log**: An **append-only history** of events where producers append records to a log and consumers read from positions in that log, often with the ability to replay past events.[2] [6] [3]

Data Lifecycle

- **Queue**:
  - Messages are transient.
  - After a consumer successfully processes and acknowledges a message, it is deleted from the queue.[5] [1] [4]
- **Event log**:
  - Events are immutable and retained for a long time or a configured retention period.
  - Reading an event does not remove it; many consumers can read the same event at different times.[6] [7] [2]

Consumption Model

- **Queue**:
  - Typically "competing consumers": a message is delivered to **one** consumer instance in a consumer group.[1] [4] [5]
  - Focus on reliable, at-least-once or exactly-once delivery to a single worker.
- **Event log**:
  - Publish–subscribe style: **multiple** independent consumers or consumer groups can read the same event.[3] [2] [6]

- Each consumer tracks its own offset in the log, so they can read, rewind, and reprocess as needed.[7] [2]

Ordering and Scaling

- **Queue**:
  - Often guarantees ordering only within a single queue, but parallel consumers can lead to weaker effective ordering in practice.[3] [1]
  - Designed primarily for task throughput and worker scaling rather than strict history semantics.[5] [3]
- **Event log**:
  - Strong ordering **within a partition/topic**; consumers read sequentially in log order.[2] [3]
  - Scaling is usually done via partitions; each partition is an ordered log.[2] [3]

Typical Use Cases

- **Use a message queue when**:
  - You need background job processing (e.g., sending emails, resizing images).
  - You want to decouple services and smooth load with asynchronous task execution.[6] [1] [5]
- **Use an event log when**:
  - You need an audit trail, event sourcing, or the ability to replay history to rebuild state.
  - You do real-time analytics, stream processing, or fan-out events to many consumers.[6] [3] [2]

Technologies Table

Aspect	Message queue (e.g., RabbitMQ, SQS)	Event log (e.g., Kafka, Pulsar, Kinesis)
Primary role	Task/work distribution	Durable event history and streaming
Consumption	One consumer per message	Many consumers per event
Deletion	Removed after processing	Retained for a period / long term
Replay	Generally not supported	Core feature via offsets
Best for	Async jobs, decoupling services	Event sourcing, analytics, CQRS, CDC

⁂

Why do we split topics into partitions?

Topics are split into partitions to scale throughput, enable parallel consumption, and still preserve ordering where needed.[12] [13] [14]

Scalability and Throughput

- Each partition is an independent, append-only log that can live on a different broker, so a topic can use multiple machines' CPU, disk, and network instead of being limited to one.[15] [16] [12]
- Multiple consumers in the same group can read from different partitions in parallel, increasing maximum processing throughput for that topic.[13] [14]

Ordering vs Parallelism

- Kafka preserves record order **within** a partition but not across the whole topic, so partitioning lets you balance "I need ordering for this key" vs "I need parallelism overall."[17] [12]
- By choosing a partition key (like user ID or account ID), all events for that key go to the same partition, giving per-key ordering while still letting other keys be processed concurrently on other partitions.[18] [19] [12]

Fault Tolerance and Data Distribution

- Partitions are the unit of replication; splitting a topic into partitions and replicating them across brokers improves availability if a broker fails.[16] [14]
- Partitioning spreads storage across brokers, allowing topics to hold more data and continue operating even when a single node runs out of capacity or goes down.[15] [16]

Operational Flexibility

- The number of partitions determines the maximum number of consumers in a group that can work in parallel on that topic, giving a direct knob to tune concurrency.[14] [15]
- Partitioning by attributes (e.g., geography, tenant, account) helps co-locate related data, which can improve cache locality, query efficiency, and downstream shard alignment.[12] [13]

⁂

What happens if a Kafka broker (leader) crashes?

When a Kafka leader broker for a partition crashes, one of the in-sync replicas is elected as the new leader and clients transparently fail over to it, as long as replication is configured correctly and enough replicas are still up.[22] [23]

Cluster-side behavior

- Kafka's controller detects that the broker is down (via ZooKeeper in older versions or the Raft-based Quorum Controller in newer ones) and marks its partitions' leaders as unavailable.[24] [22]
- For each affected partition, the controller chooses a new leader from the in-sync replica (ISR) set on other brokers and updates the cluster metadata to reflect the new leader.[23] [22]

Client behavior (producers and consumers)

- Producers and consumers talking to the old leader start seeing transient errors such as "not leader for partition" or "leader not available," then refresh metadata and automatically switch to the new leader broker; this usually stabilizes within a few seconds.[25] [26]
- Messages already committed on replicas are still available, but in-flight requests during the failure window may fail or be retried depending on client settings (acks, retries, timeouts).[26] [25]

Data safety and availability conditions

- If the partition's replication factor is high enough (commonly 3) and at least `min.insync.replicas` replicas remain alive, the partition stays both readable and writable after failover.[26] [22]
- If too many replicas are down (ISR shrinks below `min.insync.replicas`), Kafka can make the partition read-only or fully unavailable to avoid data loss, until brokers recover or an operator intervenes.[24] [26]

When things can still break

- If a topic has replication factor 1, losing the broker that hosts its only replica makes that partition unavailable and its data effectively lost from the cluster's perspective.[22] [23]
- Rapid successive broker failures can cause repeated leader elections and metadata updates, leading to temporary flapping, increased error rates, and lag spikes for consumers until the cluster stabilizes.[26] [24]

⁂

What does ISR (In Sync Replica) stand for and why is it important?

ISR stands for **In-Sync Replica**, which is the set of replicas of a Kafka partition that are fully caught up with the leader's data and are considered safe candidates for committing records and leader election.[32] [33]

What ISR means

- An In-Sync Replica is a follower replica whose log has all records that the leader has marked as committed (i.e., it is not lagging beyond configured thresholds like `replica.lag.time.max.ms`).[34] [33] [32]
- The ISR set always includes the leader plus all followers that are sufficiently up to date; replicas that fall too far behind are removed from this set until they catch up again.[35] [36] [32]

Why ISR is important

- **Data durability**: A record is typically considered committed only after it has been replicated to all or a required minimum of ISR members, which protects against data loss if the leader broker fails.[37] [38] [39]
- **Safe failover**: On leader failure, Kafka elects a new leader from the ISR set, ensuring the new leader has all committed data and preventing the cluster from rolling back acknowledged messages.[38] [37] [34]

Relation to `acks` and `min.insync.replicas`

- Producer settings like `acks=all` mean the leader waits for acknowledgements from the ISR (subject to `min.insync.replicas`) before confirming the write to the client.[37] [35]
- If the number of in-sync replicas drops below `min.insync.replicas`, Kafka can reject writes to maintain guarantees, trading availability for stronger consistency and durability.[39] [38]

⁂

1. https://www.geeksforgeeks.org/system-design/message-queues-vs-event-streams-in-system-design/
2. https://jack-vanlightly.com/blog/2018/5/20/event-driven-architectures-the-queue-vs-the-log
3. https://milvus.io/ai-quick-reference/how-does-a-distributed-log-differ-from-a-message-queue
4. https://dev.to/varadasunandaibm/message-queues-vs-event-streams-key-differences-1097
5. https://github.com/AutoMQ/automq/wiki/Differences-Between-Event-Streaming-and-Message-Queuing
6. https://www.svix.com/resources/faq/event-streaming-vs-message-queue/
7. https://dev.to/oleg_potapov/message-brokers-queue-based-vs-log-based-2f21
8. https://nordicapis.com/whats-the-difference-between-event-brokers-and-message-queues/
9. https://stackoverflow.com/questions/22713303/difference-between-event-queue-and-message-queue
10. https://doc.akka.io/libraries/guide/concepts/message-driven-event-driven.html
11. https://www.linkedin.com/posts/viktoriia-kostak-b555152aa_message-queues-vs-event-streams-in-system-activity-7354152136689000449-863Z
12. https://newrelic.com/blog/observability/effective-strategies-kafka-topic-partitioning
13. https://www.graphapp.ai/blog/kafka-topic-partition-best-practices-optimizing-performance-and-scalability
14. https://axual.com/blog/kafka-topics-and-partitions-real-time-data-streaming
15. https://www.groundcover.com/blog/kafka-logging

16. https://hevodata.com/learn/kafka-partitions/
17. https://www.confluent.io/learn/kafka-partition-strategy/
18. https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-features
19. https://www.confluent.io/blog/put-several-event-types-kafka-topic/
20. https://docs.oracle.com/en/middleware/standalone/coherence/14.1.1.2206/develop-applications/working-partitions.html
21. https://www.baeldung.com/kafka-topics-partitions
22. https://www.instaclustr.com/education/apache-kafka/apache-kafka-broker-key-components-tutorial-and-best-practices/
23. https://www.redpanda.com/guides/kafka-architecture-kafka-broker
24. https://dev.to/devopsfundamentals/kafka-fundamentals-kafka-leader-election-3l2i
25. https://docs.aws.amazon.com/msk/latest/developerguide/troubleshooting-offlinebroker-clientfailover.html
26. https://github.com/karafka/karafka/wiki/Broker-Failures-and-Fault-Tolerance
27. https://www.confluent.io/blog/how-to-survive-a-kafka-outage/
28. https://karafka.io/docs/Broker-Failures-and-Fault-Tolerance/
29. https://stackoverflow.com/questions/40183133/how-to-handle-kafka-publishing-failure-in-robust-way
30. https://www.meshiq.com/blog/common-kafka-errors-and-how-to-resolve-them/
31. https://codemia.io/knowledge-hub/path/kafka_broker_failover_-_consumer_not_working_when_first_node_is_down
32. https://www.geeksforgeeks.org/apache-kafka/understanding-in-sync-replicas-isr-in-apache-kafka/
33. https://cwiki.apache.org/confluence/display/kafka/kafka+replication
34. https://www.codestudy.net/blog/kafka-topic-replication-and-insync-replicas/
35. https://blog.2minutestreaming.com/p/kafka-acks-min-insync-replicas-explained
36. https://www.youtube.com/watch?v=sX28i9WMhhM
37. https://docs.confluent.io/kafka/design/replication.html
38. https://datafloq.com/understanding-in-sync-replicas-isr-in-apache-kafka/
39. https://www.linkedin.com/pulse/kafka-replica-mininsyncreplicas-attributes-vincent-vauban-uvyle
40. https://www.codestudy.net/blog/what-is-in-sync-replica-in-kafka/
41. https://www.linkedin.com/posts/punitmaster_understanding-apache-kafka-in-sync-replicas-activity-7309085599112552448-OlyW