Chapter 6: Design Considerations

Previous chapters have taken us through the essentials of event streaming and the core concepts of Kafka. By now, the reader should be familiar with the architecture of Kafka, its internal components, as well as the producer and consumer ecosystems. We have set up a Kafka broker, a Kafdrop UI and built basic producer and consumer applications using the Java client APIs.

In essence, the reader should now be equipped with the tools and foundational knowledge required to start building event streaming applications. But it takes time and experience to become proficient in Kafka. This is another way of saying: *To write good software, you need to make lots of mistakes.*

Mistakes need to be made; they are an essential part of learning. But learning from other peoples' mistakes is better than learning from one's own. So this chapter presents a list of considerations that are instructive in the design of performant and sustainable event streaming applications; considerations that have been amassed over years of working with these sorts of systems across a variety of industries.

Roles and responsibilities

Kafka permits a flexible arrangement between producers and consumers, allowing for a host of similar and disparate applications to interact with a topic simultaneously. In coming to terms with this, an often-asked question is: *Which party owns the topic, and who is responsible for its upkeep?*

Event-oriented broadcast

In a broadcast arrangement, where the producer-consumer relationship follows a (multi)point-to-multipoint topology, it is an accepted best-practice for the producer ecosystem to assume custodianship over the topic, and to effectively prescribe the entirety of the topic's configuration and usage semantics. These include —

- The lifecycle of the topic, as well as the associated broker-side configuration, such as the retention period and compaction policy;
- The nature and content of the published data, encodings, record schema, versioning strategy and associated deprecation period; and
- The sizing of the topic with respect to the partition count and the keying of the records.

In no uncertain terms: *the producer is king*. The producer will warrant all existential and behavioural aspects of the topic; the only decision left to the discretion of the consumer is whether to subscribe

to the topic or not. This, rather categorical, approach to role demarcation is essential to preserving the key characteristic of an event-driven architecture — *loose coupling*. The producer cannot be intrinsically aware of the topic's consumers, as doing so would largely defeat the intent of the design. This is not to say that producers should publish on a whim, or that the suitability of the published data is somehow immaterial to the outcome. Naturally, the published data should be correct, complete, and timely; however, the assurance of this lies with the designers of the system and is heavily predicated on the efficacy of domain modelling and stakeholder consultation. It is also evolutionary in nature; feedback from the consuming parties during the design, development, and operation phases should be used to iteratively improve the data quality. Stated otherwise, while the consuming parties are consulted as appropriate, the final decision rights and associated responsibilities rest with the producer.

Peer-to-peer messaging

Kafka may be used in peer-to-peer messaging arrangement, whereby the consumer is effectively responding to specific commands issued by a producer, and in most cases emitting responses back to the initiator. This model sees a role reversal: the consumer plays the role of the service provider, and therefore assumes custody over the lifecycle of the topic and its defining characteristics.

Where the response is sent over a different topic, shared among message initiators, the semantics of the response topic are also fully defined by the consumer. In more elaborate messaging scenarios, the initiator of the request may ask that the response is ferried over a dedicated topic to avoid sharing; in this case, the lifecycle of the topic and its retention will typically be managed by the initiator, while record-related aspects remain within the consumer's remit.

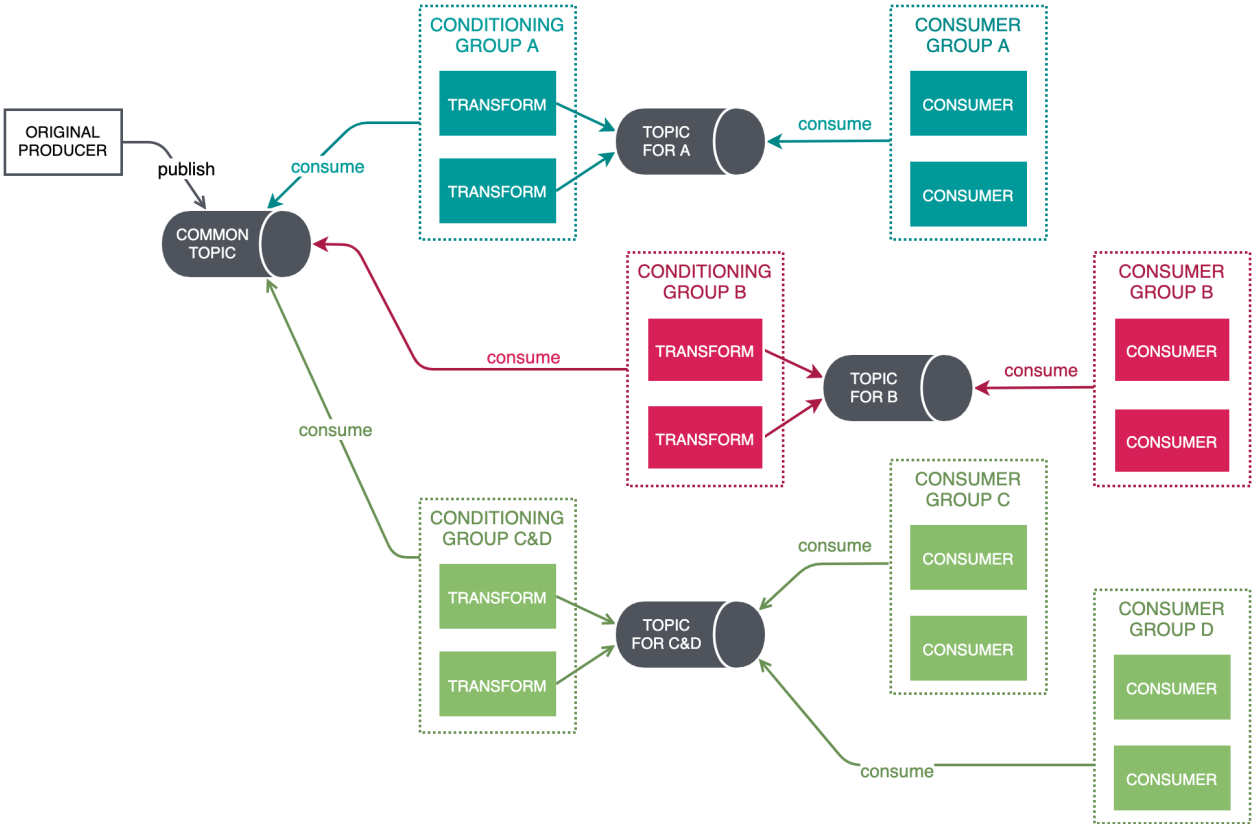
Topic conditioning

Reading the section on producer-driven topic modelling may fail to instill confidence in would-be consumers. The flip side of the coupling argument is the dreaded leap of faith. If the prerogative of the producer is to optimise topics around its domain model, what measures exist to assure the consumers that the upstream decisions do not impact them adversely? Is a compromise possible, and how does one neutralise the apparent bias without negatively impacting all parties.

These are fair questions. In answering, the reader is invited to consider the case where there are multiple disparate consumers. Truly, a single-producer-multiple-consumers is a fairly routine arrangement in contemporary event-driven architecture. And this is precisely the use case that highlights why a compromise is not a viable option. As the number of disagreeing parties grows, the likelihood of striking an effective compromise decreases to the point where the resulting solution is barely tractable for either party. It is the architectural equivalent of children fighting over a stuffed toy, where the inevitable outcome is the tearing of the toy, the dramatic scattering of its plush contents, resulting in discontent but ultimately quiesced children.

So what is one to do?

While this problem might not be trivially solvable, it can be readily compartmentalised. The use of a staged event-driven architecture (SEDA) offers a way of managing the complexity of diverse consumer requirements without negatively impacting the consumer applications directly or coupling the parties. Rather than feeding consumers directly off the producer-driven topic, intermediate processing stages are employed to condition the data to conform to an individual consumer group's expectations. These stages are replicated for each independent set of consumers, as shown in the diagram below.



Conditioning stages

Under this model, the impedance mismatch is resolved by the intermediate stages, leading to improved maintainability of the overall solution and allowing each of the end-parties to operate strictly within the confines of their respective domain models. The responsibilities of the parties are unchanged; the consumer takes ownership of the conditioning stage, responsible for its development and upkeep. While this might initially appear like a zero-sum transfer, the benefit of this approach is in its modularity. It embraces the single responsibility principle, does not clutter the consumer with transformational logic, and can lead to a more sustainable solution.

The acquired modularity may also lead to opportunities for component reuse. If two (or more) distinct consumer groups share similar data requirements, a common conditioning stage can power both. In the example above, a single conditioning stage powers both consumer groups *C* and *D*.

Parallelism

It was previously stated in Chapter 3: Architecture and Core Concepts that exploiting partial order enables consumer parallelism. The distribution of partition assignments among members of a consumer group is the very mechanism by which this is achieved. While consumer load-balancing is straightforward in theory, the practical implications of aspects such as topic sizing, record key selection, and consumer scaling are not so apparent.

There are several factors one must account for when designing highly performant event streaming applications. The following is an enumeration of some of these factors.

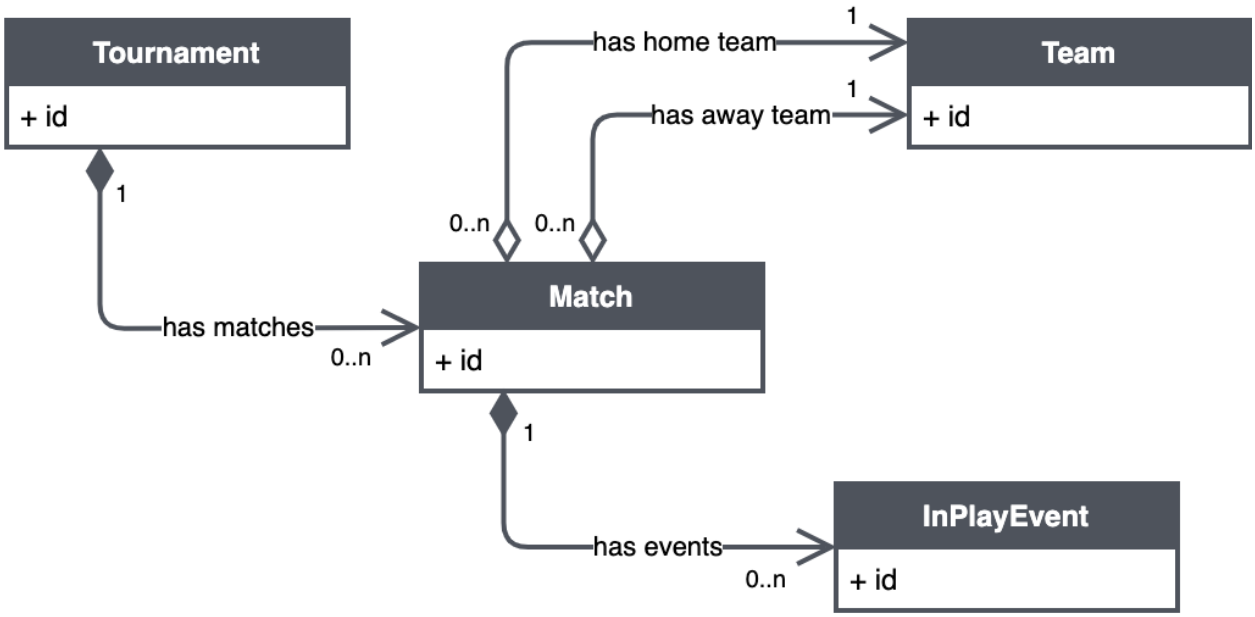
Producer-driven partitioning

Irrespective of the particular messaging topology employed, the responsibility of assigning records to partitions lies solely with the producer. This stems from a fundamental design limitation of Kafka; both topics and partitions are physical constructs, implemented as segmented log files under the hood. The publishing of a record results in the appending of a serialized form of the record to the head-end of an appropriate log file. Once this occurs, the relationship between a record and its encompassing topic-partition is cemented for the lifetime of the record.

In Kafka terminology, the most recent records are considered to be at the 'head' end of a partition.

The implication of this constraint is that the producer should take the utmost care in keying the records such as to preserve the essential causal relations, without overly constraining the record order. In practice, events will relate to some stable entity; the identifier of that entity can serve as the key of the corresponding record.

By way of example, consider a hypothetical content syndication system catering to football fans. Our system integrates with various real-time content providers, listening to significant in-play events from football matches as they unfold, then publishes a consolidated event stream to power multiple downstream consumers — mobile apps, scoreboards, player and match stats, video stream overlays, social networks such as Twitter, and other subscribers. True to the principles of event-driven architecture, we try to remain agnostic of what's downstream, focusing instead on the completeness and correctness of the emitted event stream — the data content of the records, their timing, and granularity. A simplified domain model for this contrived scenario is illustrated below.



Football domain model

Mimicking the real-life order of events is a good starting point for designing a streaming application. With this in mind, it makes sense to appoint the football match as the stable entity — its identifier will act as a record key to induce the partial order. This means that goals, corners, penalties, and so forth, will appear in the order they occurred within a match; records *across* matches will not be ordered by virtue of the matches being unrelated.

The basic requirement for the *stability* of the chosen entities is that they should persist for the lifetime of all causally related events. This does not imply that the entities should be long-lived, only that they exist long enough to survive the causal chains that depend on them. To our earlier example, the football match survives all of its in-play events.

Record keys bound to some stable identity can be thought of as the equivalent of *foreign keys* in database parlance, and the stability characteristic can be likened to a *referential integrity* constraint. Exploring the relational database analogy, the entity in the linked table would persist for as long as it is being referenced by one or more foreign keys. Likewise, the linked stable entity persists for the duration of time from the point when the first record is published to when the last record leaves the producer. Thereafter, once the causal order is materialised in Kafka, the linked entity becomes largely irrelevant.

Assuming no special predecessor-successor relationship between the chosen stable entities, the partial order of the resulting records will generally be sufficient for most, if not all, downstream consumers. Conversely, if the entities themselves exhibit causal relationships, then the resulting stream may fail to capture the complete causality. Reverting to our earlier example, suppose one downstream subscriber is a tournament leaderboard application, needing to capture and present the relative standing of the teams as they progress through a tournament. Match-centric order may be insufficient, as goals would need to be collated in tournament order.

Where a predecessor-successor relationship exists between stable entities, and that relationship is significant to downstream subscribers, there are generally two approaches one may take. The first is to coarsen the granularity of event ordering, for example, using the tournament identifier is the record key. This preserves the chronological order of in-play events within a tournament, and by extension, within a match. The second approach is to transfer the responsibility of event reordering to the downstream subscriber.

Coarsening of causal chains is generally preferred when said order is an intrinsic characteristic of the publisher's domain. In our example, the publisher is well aware of the relationship between tournaments and matches, so why not depict this?

The main drawback of this approach is the reduced opportunity for consumer parallelism, as coarsening leads to a reduction in the cardinality of the partitioning key. Subscribers will not be able to utilise Kafka's load-balancing capabilities to process match-level substreams in parallel; only tournaments will be subject to parallelism. This may satisfy some subscribers, but penalise others.

An extension of this model is to introduce conditioning stages for those subscribers that would benefit from the finer granularity. A conditioning stage would consume a record from a coarse-grained input topic, then republish the same value to an output topic with a different key — for example, switching the key from a tournament ID to a match ID. This model nicely dovetails into the broader principle of producer-driven domain modelling, while satisfying individual subscriber needs with the help of SEDA.

The second approach — fine-grained causal chains with consumer-side reordering — assumes that the consumer, or some intermediate stage acting on its behalf, is responsible for coarsening the granularity of the event substreams to fit some bespoke processing need. The conditioning stage will need to be stateful, maintaining a staging datastore for incoming events so that they can be reordered. In practice, this is much more difficult than it appears. In some cases, there simply isn't enough data available for a downstream stage to reconstruct the original event order. In short, consumer-side reordering may not always a viable option.

There is no one-size-fits-all approach to partitioning domain events. As a rule of thumb, the producer should publish at the finest level of granularity that makes sense in its respective domain, while supporting a diverse subscriber base. A crucial point: being agnostic of subscribers does not equate to being ignorant of their needs. The effective modelling of the domain and the associated event streams fall on the shoulders of architects and senior technologists; stakeholder consultation and understanding of the overall landscape are essential in constructing a sustainable solution.

In the absence of consumer awareness, one litmus test for formulating partial order is to ascertain that the resulting stream can be used to reconstitute the source domain. Ask the question: Can a hypothetical subscriber rebuild an identical replica of the domain purely from the emitted events while maintaining causal consistency? If the answer is 'yes', then the event stream should be suitable for any downstream subscriber. Otherwise, if the answer is 'no', then there is a gap in the condition of the emitted events that requires attention.

Topic width

With the correct keying granularity in place, the next consideration is the 'width' of the topic — the number of partitions it encompasses. Assuming a fair distribution of keys, increasing the number of partitions creates more opportunities for the consumer ecosystem to process records in parallel. Getting the topic width right is essential to a performant event streaming architecture.

Regrettably, Kafka does not make this easy for us. Kafka only permits non-destructive resizing of topics when *increasing* the partition count. Decreasing the number of partitions is a destructive operation — requiring the topic to be created anew, and manually repopulated.

One of the main gotchas of resizing topics is the effect they have on record order. As stated in Chapter 3: Architecture and Core Concepts, a Kafka producer hashes the record's key to arrive at the partition number. The hashing scheme is not *consistent* — two records with the same key hashed at different points in time will correspond to an identical partition number *if and only if* the number of partitions has not changed in that time. Increasing the partition count results in the two records occupying potentially different partitions — a clear breach of Kafka's key-centric ordering guarantee. Kafka will not rehash records as part of resizing, as this would be prohibitively expensive in the absence of consistent hashing. (To be clear, consistent hashing would not eliminate the need for rehashing records *per se*, but it would dramatically reduce the number of affected hashes when the topic is widened.)

When the correctness of a system is predicated on the key-centric ordering of records, avoid resizing the topic as this will effectively void any ordering guarantees that the consumer ecosystem may have come to rely upon.

One approach to dealing with prospective growth is to start with a sufficiently over-provisioned topic, perhaps an order of magnitude more partitions than one would reasonably expect — thereby avoiding the hashing skew problem down the track. On the flip side, increasing the number of partitions may increase the load on the brokers and consumers.

A partition is backed by log files, which require additional file descriptors. (At minimum, there is one log segment and one index file per partition.) Inbound writes flow to dedicated buffers, which are allocated per partition on the broker. Therefore, increasing the number of partitions, in addition to consuming extra file handles, will result in increased memory utilisation on the brokers. A similar impact will be felt on consumers, sans the file handles, which also employ per-partition buffers for fetching records.

A further impact of wide topics may be felt due to the limitations of the inter-broker replication process and its underlying threading model. Specifically, a broker will allocate one thread for every other broker that it maintains a connection with, which covers the set of replicated partitions — where the two peers have a leader-follower relationship. The replication threads may act as a bottleneck when shuttling records from the partition leader to in-sync replicas, and thereby impact the publishing latency when the producer requests all replicas to acknowledge the writes. This problem is ameliorated when the number of brokers increases, as the growth of the cluster has

a downward effect on the ratio of partitions to brokers, taking the pressure off each connection. Confluent — one of the major contributors to Apache Kafka — recommends limiting the number of partitions per broker to $100 \times b \times r$, where b is the number of brokers in a Kafka cluster and r is the replication factor.

So while a wider topic provides for greater theoretical throughput, it does carry practical and immediate implications on the client and broker performance. The impacts of widening individual topics may not be substantial, but they may be felt in aggregate. This is not to say that topics should not be over-provisioned; rather, decisions regarding the sizing of the topics and the extent of over-provisioning should not be taken on a whim. The broker topology and the overall capacity of the cluster play a crucial role; these have to be adequately specified and taken into consideration when sizing the topics.

If dealing with an existing topic that has saturated its capacity for consumer parallelism, consider a staged destructive resize. Create a new topic of the desired size, using a tool such as *MirrorMaker* to replicate the topic contents onto the wider topic. When the replication catches up, switch the producers to double-publish to both the old and the new topics. There may need to be some producer downtime to allow for topic parity. Individual consumer groups can start migrating to the new topic at their discretion; however, the misalignment of partitions may present a challenge with persisted offsets. Assuming that consumers have been designed with idempotency in mind, one should be able to set the `auto.offset.reset` property to `earliest` to force the reprocessing of the records from the beginning. Depending on the retention of the topic, this may take some time, which will also delay the consumers' ability to process new records. Alternatively, one can reset the offsets to a specific timestamp, which will substantially reduce the quantity of replayed records. Some reprocessing will likely be unavoidable; such is the price for resizing strongly-ordered topics. (The double-publishing code can be removed when all consumers have been migrated.)

Scaling of the consumer group

Kafka will allocate partitions approximately evenly among members of a consumer group, up to the width of the topic. So to increase parallelism, one must ensure sufficient consumer instances in the group. Allocating a fixed number of instances to the group is usually not economical, as it may result in idle capacity, particularly for event streams that exhibit cyclic or bursty loading. The recommended approach is to employ an automated horizontal scaling technique to dynamically expand or contract the population of the group in response to load demand. For example, if deploying the consumer group within a public cloud environment like AWS, one may use an autoscaling group to provision additional instances based on CPU utilisation metrics. Alternatively, if the consumer application is containerised, the use of a container orchestration platform is recommended. For example, when deployed in Kubernetes, one would employ horizontal pod autoscaling to dynamically size the consumer group.

Internal consumer parallelism

An alternate way of increasing consumer throughput, without widening the topic or scaling the consumer group, is to exploit parallelism within the consumer process. This can be achieved by partitioning the workload among a pool of threads by independently hashing the record keys to maintain local order. This strategy would be classed as vertical scaling, requiring increased parallelism on each consumer node in exchange for reducing the number of consumers, and hence the number of partitions.

Idempotence and exactly-once delivery

Chapter 3: Architecture and Core Concepts had introduced the concepts of delivery guarantees, stating that Kafka allows for two different modes of delivery by simply shifting the point when the consumer commits its offsets. *At-least once* and *at-most-once* guarantees were named, but there was no mention of an *exactly-once* guarantee. This might be a good segue to discuss the differences between delivery modes; ultimately, it will help us understand what it means to do something 'exactly once'.

The role of messaging middleware is to decouple communications between collaborating parties. When a sender publishes a message, there is an assumption that the receiver (or receivers, as there may be multiple such parties) will eventually consume and process the message. Messaging middleware is generally divided into two categories: those that offer *at-most-once* and those that offer *at-least-once* guarantees. And then we have Kafka, which has a foot in each camp.

The at-most-once guarantee simply means that a message is never redelivered to its recipient, no matter the contingency. The consumer might read the record, but then fail for whatever reason before processing the record. If the offsets for the said record were committed before the record was processed, then the reassignment of the partition following the consumer's failure will result in the skipping of the record by the new consumer.

Some messaging systems are truly *fire-and-forget*, in the sense that there might not even be an initial attempt to deliver the message in the first place. For example, some message brokers support load shedding, in which case a message might be purged from the queue to avoid accumulating a backlog of stale messages. Other messaging systems, such as ZeroMQ, might be purely in-memory, in which case the loss of a node will result in the loss of undelivered messages. Some systems use the term 'maybe-once' as a stand-in for 'at-most-once', which seems more fitting in some cases. Kafka's take on at-most-once processing is slightly different from some of its counterparts. Provided that a record has been stably persisted to a topic, and is within the retention period, Kafka will always allow the consumer to read the record at least once, no matter what. So the term 'at-most-once' applies to the processing of the record, rather than to the mere act of reading the record.

The at-least-once guarantee means that a message will only be marked as delivered when it completes its entire journey within the consumer application. Failure prior to this point is treated as non-delivery and a retry will ensue.

The term 'delivery' may seem somewhat confusing, especially if one perceives delivery in a postal sense. Delivery is not just leaving a record at the consumer's doorstep, but seeing the consumer 'sign' for the delivery by committing the record's offset.

Using the at-most-once approach for delivery is acceptable in many cases, especially where the occasional loss of a record does not leave a system in a perpetually inconsistent state. At-most-once delivery is useful where the source of the record is continuously, within a fixed interval, emitting updates to some entity of interest, such that the loss of one record can be recovered from in bounded time. Conversely, the at-least-once approach is more fitting where the loss of a record constitutes an irreversible loss of data, violating some fundamental invariant of the system. But the flip side is that processing a record multiple times may introduce undesirable side-effects. This is where the notion of *exactly-once* processing enters the scene. In fact, when contrasting at-least-once with at-most-once delivery semantics, an often-asked question is: *Why can't we just have it once?*

Without delving into the academic details, which involve conjectures and impossibility proofs, it is sufficient to say that exactly-once semantics are not possible without tight-knit collaboration with the consumer application. As disappointing as it may sound, a messaging platform cannot offer exactly-once guarantees on its own. What does this mean in practice?

To achieve the coveted *exactly-once* semantics, consumers in event streaming applications must be *idempotent*. In other words, processing the same record repeatedly should have no net effect on the consumer ecosystem. If a record has no additive effects, the consumer is inherently idempotent. (For example, if the consumer simply overwrites an existing database entry with a new one, then the update is naturally idempotent.) Otherwise, the consumer must check whether a record has already been processed, and to what extent, prior to processing the record. *The combination of at-least-once delivery and consumer idempotence collectively leads to exactly-once semantics.*

The design of an idempotent consumer mandates that all effects of processing a record must be traceable back to the record. For example, a record might require updating a database, invoking some service API, or publishing one or more records to a set of downstream topics. The latter is particularly common in SEDA systems, which are essentially graphs of processing nodes joined by topics. When a consumer processes a record, it will have no awareness of whether the record is being processed for the first time, or whether the given record is a repeat attempt of an earlier failed delivery. As such, *the consumer must always assume that a record is a duplicate*, and handle it accordingly. Every potential side-effect must be checked to ensure that it hasn't already occurred, before attempting it a second time. When a side-effect is itself idempotent, then it can be repeated unconditionally.

In some cases, there may not be an easy way to determine whether a potential side-effect had already occurred as a result of a previous action. For example, the side-effect might be to publish a record on another topic; there is often no practical way of querying for the presence of a prior record. Kafka offers an advanced mechanism for correlating the records consumed from input topics with the resulting records on output topics — this is discussed in Chapter 18: Transactions. Transactions can create joint atomicity and isolation around the consumption and production of records, such that either all scoped actions appear to have occurred, or none.

Where the target endpoint is a (non-Kafka) message queue, the downstream receiver must be made idempotent. In other words, two (or more) identical records with different offsets must not result in material duplication somewhere down the track. This is called *end-to-end idempotence*. As the name suggests, this guarantee spans the entirety of an event-streaming graph, covering all nodes and edges. In practice, this is achieved by ensuring that any two neighbouring nodes have an established mechanism for idempotent communication.

This chapter has explored some of the fundamental considerations pertinent to the design and construction of safe and performant event streaming applications.

We started by covering the roles and responsibilities of the various parties collaborating in the construction of distributed event-driven applications. The key takeaway is that the parties publishing or consuming events can be likened to service providers and invokers, and their roles vary depending on the messaging topology. We also explored scenarios where producers and consumers might disagree on the domain model, and the methods by which this can be resolved.

The concept of key-centric record parallelism — Kafka's trademark performance enhancer — has been explored. We looked at the factors that constrain the consumers' ability to process events in parallel, and the design considerations that impact the producing party.

Finally, we contrasted at-most-once and at-least-once delivery guarantees and arrived at the design requirements for exactly-once — namely, a combination of at-least-once delivery and consumer idempotence.