DATA ENGINEERING

Streaming Data Engineering with Kafka and Flink

Real-time pipelines, exactly-once processing and stateful streams

Houssam Kodad

PDF · DATAFORGE BOOKS

Questions about this sample or the full edition: **support@dataforgebooks.com**

CONTENTS

Table of Contents

CHAPTER 01

From Batch to Streams

There is a moment in the life of most data platforms when batch stops being good enough. Fraud needs to be caught before the transaction settles, not in tomorrow's report. A recommendation should reflect what a user did thirty seconds ago. An alert that fires an hour late is an incident review, not a save. Streaming is how you close that gap — and Kafka and Flink are the tools we will use to do it correctly.

The word "correctly" matters more than it looks. Building something that processes events quickly is easy; building something that stays correct when a machine dies mid-computation, when events arrive out of order, or when you need to reprocess a week of history after fixing a bug — that is the real work. This book is about that work.

This first chapter frames the shift from batch to streams. We look at what waiting for a batch actually costs, why the humble append-only log is such a powerful foundation, the event-time problem that trips up almost every newcomer, and the guarantees — at-least-once, exactly-once — that determine whether you can trust your results. By the end you will understand the shape of the system the rest of the book builds.

The Cost of Waiting for a Batch

A nightly batch encodes a hidden assumption: that a decision made on data up to twelve or twenty-four hours old is good enough. For many reports it is. But latency is not free, and the cost is rarely visible on an invoice — it is in the fraud that cleared, the outage detected late, the customer who churned while your model still believed they were happy.

Batch also creates spiky, bursty load. Nothing happens for hours, then everything runs at once, competing for the same resources and turning a midnight failure into a morning emergency. Streaming spreads that work continuously and processes each event as it arrives, trading a simpler mental model for fresher results and smoother, more predictable utilisation.

Logs as the Source of Truth

At the heart of every streaming system is a deceptively simple structure: the append-only log. Events are written in order and never mutated. Consumers read from a position — an offset — and advance at their own pace. This is what Kafka provides, and it quietly changes how you think about data: current state becomes a view derived from a stream of facts, rather than the primary thing you store.

Because the log is durable and replayable, you get capabilities batch systems struggle with. Add a new consumer and it can read history from the beginning. Fix a bug in your processing logic and reprocess from an earlier offset to rebuild corrected results. Two independent systems can derive entirely different state from the same ordered stream without ever coordinating with each other.

Kafka in One Page: Topics, Partitions, Offsets

Kafka organises events into topics, and each topic is split into partitions. A partition is an ordered, append-only sequence; ordering is guaranteed within a partition but not across them. The partition an event lands in is chosen by its key, which is how Kafka gives you both parallelism and per-key ordering at the same time — all events for one customer go to one partition and stay in order.

Consumers track their position in each partition with an offset, a simple integer cursor. Committing an offset says "I have processed up to here." This tiny mechanism is the foundation of every delivery guarantee we will discuss: where and when you commit the offset, relative to when you act on the event, is precisely what separates at-least-once from exactly-once processing.

```
# Producing keyed events: same key -> same partition -> ordered per customer
producer.send(
    topic="orders",
    key=order["customer_id"].encode(),
    value=json.dumps(order).encode(),
)

# A consumer advances an offset cursor per partition
for msg in consumer:
    handle(msg.value)
    consumer.commit()      # "I have processed up to here"
```

Where Flink Fits

Kafka stores and transports events; it does not, by itself, compute much. Flink is the engine that does the heavy processing — joining streams, aggregating over windows, maintaining state, and reacting to patterns — while giving you strong correctness guarantees under failure. If Kafka is the nervous system of a streaming platform, Flink is where the thinking happens.

Flink's defining feature is managed state. A streaming job that counts events per user, or joins clicks to impressions, must remember things between events, and that memory has to survive machine failures. Flink keeps this state, checkpoints it consistently, and restores it on recovery — which is what lets a job resume after a crash as if nothing had happened. Most of this book is, in some sense,

about using that capability well.

The Event-Time Problem

Here is the question that separates people who have run streaming systems from people who have only read about them: when an event says it happened at 12:00 but arrives at your processor at 12:05, which time do you use? Processing time — the clock on the machine — is simple but wrong, because it makes your results depend on network delays and retries. Event time — the timestamp in the event — is correct but forces you to handle data that arrives late or out of order.

Flink builds on event time using watermarks: markers that flow through the stream and assert "we believe we have now seen all events up to time T." Watermarks let a windowed computation decide when it is safe to emit a result, while still giving you explicit control over how long to wait for stragglers. Getting watermarks right is the difference between aggregates you can trust and aggregates that silently drop late data.

Delivery Guarantees: At-Least-Once vs Exactly-Once

Every streaming system makes a promise about what happens when something fails mid-processing. At-least-once means no event is ever lost, but an event may be processed twice after a crash — fine if your operations are idempotent, dangerous if you are incrementing a counter or moving money. At-most-once means you never double-process but may lose events. Exactly-once means each event affects the result once and only once, even across failures.

Exactly-once sounds like obvious table stakes, but it is genuinely hard and not free, because it requires coordinating your state, your input offsets and your output sink atomically. Flink achieves it with checkpoint barriers and transactional sinks, and we will take it apart in detail later. For now, internalise that the guarantee you need is a design decision driven by what your output does — and that choosing it deliberately is the mark of a serious streaming engineer.

```
// Exactly-once requires that state, source offsets and sink writes
// all commit together at a checkpoint — or all roll back on failure.
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(60_000);   // a consistent snapshot every 60s
env.getCheckpointConfig()
   .setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
```

When Streaming Is the Wrong Choice

Streaming is powerful, but it is not free. It is operationally heavier than batch, harder to reason about, and unforgiving of sloppy state management. If your data is consumed once a day, if correctness matters far more than freshness, or if your team has no appetite for running stateful distributed systems on call, batch is very likely the right answer — and choosing it is a sign of judgement, not timidity.

The honest engineer asks what decision this latency actually enables before reaching for Kafka and Flink. When the answer is real and time-sensitive — a fraud block, a live recommendation, an operational alert — streaming earns its complexity. The rest of this book assumes you have asked that question and answered yes, and shows you how to build systems that stay correct under failure rather than merely fast on a good day.

This is a free sample

You've reached the end of the sample chapter.

Get the complete book — every chapter, fully worked — at **dataforgebooks.com**.

FULL EDITION · 336 PAGES · PDF

Read the full title at **dataforgebooks.com**

Questions? support@dataforgebooks.com