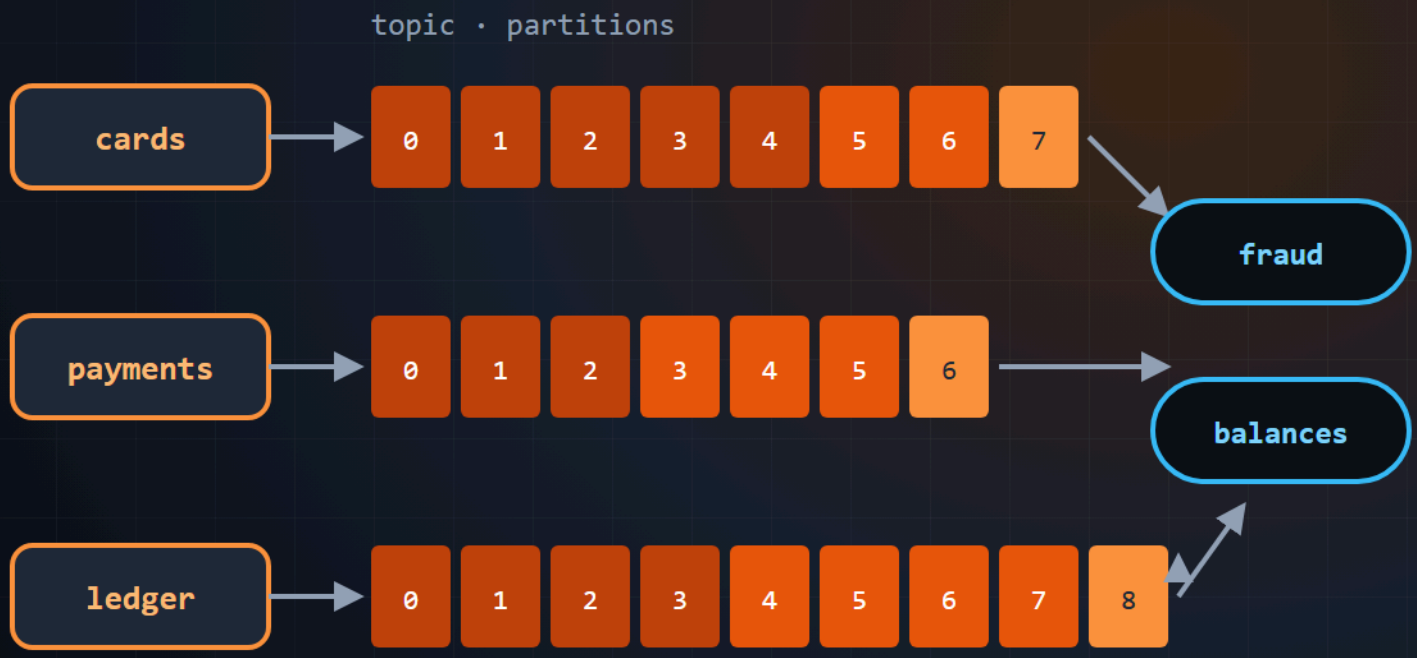


produce · partition · replicate · consume · process

A PRACTICAL GUIDE

Kafka and Event-Driven Architecture

Design, build, and operate reliable event-driven systems — from first topic to production

STREAMS · SCHEMAS · EXACTLY-ONCE · CDC · SAGAS · CQRS

THENEXBOOKS

Kafka and Event-Driven Architecture

A Practical Guide

First Edition

THENEXBOOKS

Kafka and Event-Driven Architecture
A Practical Guide

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. All traffic and capacity figures in the examples are illustrative assumptions, not measurements. Apache Kafka and its ecosystem evolve quickly; always consult the official documentation for the versions you use.

Apache Kafka, Kafka, Apache Flink, Flink, and the Apache feather logo are trademarks of The Apache Software Foundation. Confluent, Schema Registry, and ksqlDB are associated with Confluent, Inc. Debezium, PostgreSQL, Docker, Kubernetes, Java, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by The Apache Software Foundation, Confluent, or any other company or project mentioned.

Brightvault, the running example in this book, is fictional, as are all customers, merchants, partners, and people mentioned in its examples. Any resemblance to real banks, companies, or products is coincidental.

Preface

Most systems start life as a set of services calling each other and sharing a database. It works — until it does not. A slow downstream service makes every caller slow. A new team needs the same data and adds yet another synchronous call. An analytics job hammers the production database. A change in one place ripples unexpectedly through five others. Sooner or later, many teams discover that what they really want is for each part of the system to announce what happened, and for every other part to react in its own time.

That is the promise of event-driven architecture, and Apache Kafka has become one of the most widely used foundations for it. Kafka gives you a durable, ordered, replayable log of events that many independent consumers can read at their own pace. On top of that log you can build decoupled microservices, real-time analytics, stream processing, change data capture, event sourcing, and reliable integrations between old and new systems.

Kafka is also easy to use badly. Topics with the wrong keys lose ordering. Consumers that are not idempotent double-charge customers when a rebalance replays messages. Schemas change and silently break downstream teams. A poison message blocks a partition at three in the morning. This book is about getting those details right. We explain how Kafka works, why each setting and pattern exists, and where it breaks — so you can design, build, and operate event-driven systems with confidence.

Who This Book Is For

- **Backend developers** who are starting to use Kafka and want to understand it properly instead of copying configuration.
- **Engineers building microservices** who want to decouple services with events without losing correctness.
- **Architects and tech leads** who need to choose between choreography and orchestration, event sourcing and CRUD, or Kafka Streams and other processors.
- **Platform and operations engineers** who run Kafka clusters and want a solid grounding in sizing, monitoring, security, and multi-region design.

What You Will Learn

- What events are, how to model them, and when event-driven architecture is — and is not — the right choice.
- Kafka's architecture: brokers, topics, partitions, replication, in-sync replicas, and KRaft.
- Producers and consumers in depth: batching, acknowledgments, idempotence, consumer groups, offsets, rebalancing, and lag.

- Schemas and data contracts with Avro, Protobuf, JSON Schema, and a schema registry — and how to evolve them safely.
- Delivery guarantees, transactions, exactly-once processing, retries, dead-letter topics, idempotent consumers, and ordering.
- The transactional outbox, change data capture with Debezium, and Kafka Connect.
- Stream processing with Kafka Streams: stateful aggregations, windowing, joins, and state stores — and when to reach for Flink or ksqlDB.
- Architecture patterns: choreography and orchestration, sagas, event sourcing and CQRS, and the strangler fig.
- Testing event-driven systems, and operating Kafka in production: sizing, monitoring, tuning, upgrades, security, and multi-region replication.

How This Book Is Organized

Part I — Event-Driven Foundations (Chapters 1–5) explains why events matter, how to model them, how Kafka works, and walks you through a first working event flow. **Part II — Producers, Consumers, and Data Contracts** (Chapters 6–10) goes deep on the client APIs, schemas, schema evolution, and topic design. **Part III — Delivery Guarantees and Reliability** (Chapters 11–15) covers delivery semantics, transactions, error handling, the outbox pattern with change data capture, and idempotent consumers. **Part IV — Stream Processing and Integration** (Chapters 16–20) builds real-time processing with Kafka Streams, integrates external systems with Kafka Connect, and compares stream processors. **Part V — Event-Driven Architecture Patterns** (Chapters 21–25) applies everything to microservices, sagas, event sourcing, CQRS, legacy integration, and testing. **Part VI — Operating Kafka in Production** (Chapters 26–28) covers operations, security, multi-region design, and a complete end-to-end case study. The appendices provide a cheat sheet, templates and a configuration reference, a production checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Brightvault

Throughout the book we build the event platform of **Brightvault**, a fictional digital bank. Customers open accounts in a mobile app, pay by card, send transfers, and receive instant notifications. Behind the scenes, card authorizations stream in from the card network, a fraud service scores them in real time, a ledger posts every debit and credit, balances are kept up to date for the app, and a legacy core-banking database is gradually replaced using change data capture.

Banking is a demanding teacher. Money must never be lost or counted twice, events must stay in order per account, audits require long retention, security is non-negotiable, and customers expect answers in seconds. Every concept in this book has a concrete home in Brightvault, which makes the trade-offs real instead of theoretical. All scale figures are assumptions we state explicitly.

Versions and Conventions

Examples use Java 21 with the official Apache Kafka Java clients and Kafka Streams, and run against Kafka in KRaft mode — Apache Kafka 4.0 removed the older ZooKeeper mode. Where a feature is new, in early access, or version-dependent, we say so and point you to the documentation rather than guessing. Command-line examples use the scripts that ship with Kafka, and local setups use Docker Compose.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake or a behavior that can cause data loss, duplicates, or outages in production.

Code appears in `monospace`. Money is always represented as integer minor units with a currency code, events are Java records, and long lines are wrapped to fit the page.

Contents

Preface

PART I — EVENT-DRIVEN FOUNDATIONS

1 Why Event-Driven Architecture: From Request/Response to Streams of Events

2 Events, Commands, and Messages: Modeling What Happened

3 Kafka Architecture: Brokers, Topics, Partitions, Replication, and KRaft

4 Getting Started: Running Kafka Locally and Using the Command-Line Tools

5 Your First Event Flow: Producing and Consuming Brightvault Card Events

PART II — PRODUCERS, CONSUMERS, AND DATA CONTRACTS

6 Producers in Depth: Batching, Acknowledgments, Idempotence, and Ordering

7 Consumers in Depth: Consumer Groups, Offsets, Rebalancing, and Lag

8 Serialization and Schemas: JSON, Avro, Protobuf, and Schema Registry

9 Schema Evolution and Data Contracts

10 Topic Design: Keys, Partition Counts, Retention, and Log Compaction

PART III — DELIVERY GUARANTEES AND RELIABILITY

11 Delivery Semantics: At-Most-Once, At-Least-Once, and Exactly-Once

12 Kafka Transactions and Exactly-Once Processing

13 Error Handling: Retries, Dead-Letter Topics, and Poison Messages

14 The Transactional Outbox and Change Data Capture with Debezium

15 Idempotent Consumers, Deduplication, and Ordering Guarantees

PART IV — STREAM PROCESSING AND INTEGRATION

16 Kafka Streams Fundamentals: Topologies, KStream, and KTable

17 Stateful Processing: Aggregations, Windowing, and State Stores

18 Stream Joins and Enrichment: Real-Time Fraud Detection

19 Kafka Connect: Integrating Databases, Object Storage, and Search

20 Beyond Kafka Streams: Apache Flink, ksqlDB, and Choosing a Stream Processor

PART V — EVENT-DRIVEN ARCHITECTURE PATTERNS

21 Event-Driven Microservices: Choreography vs. Orchestration

22 Sagas and Long-Running Business Processes

23 Event Sourcing and CQRS with Kafka

24 Integration Patterns: Webhooks, APIs, Legacy Systems, and the Strangler Fig

25 Testing Event-Driven Systems

PART VI — OPERATING KAFKA IN PRODUCTION

26 Operating Kafka: Sizing, Monitoring, Tuning, and Upgrades

27 Security, Multi-Tenancy, and Multi-Region: TLS, SASL, ACLs, Quotas, and Replication

28 Case Study: Building the Brightvault Event Platform End to End

APPENDICES

A Kafka Cheat Sheet

B Templates and Configuration Reference

C Production Checklist

D Glossary

PART I

Event-Driven Foundations

Why systems move from request/response to streams of events, how to model events well, how Kafka stores and replicates them, and a first working Brightvault event flow from producer to consumer.

1 · Why Event-Driven Architecture
2 · Events, Commands, and Messages
3 · Kafka Architecture
4 · Running Kafka Locally
5 · Your First Event Flow

CHAPTER 1 · PART I — EVENT-DRIVEN FOUNDATIONS

Why Event-Driven Architecture: From Request/Response to Streams of Events

Most backend systems start life as a web of synchronous calls, and most of them eventually strain under that design. This chapter explains why, what changes when services communicate through streams of events instead, and where Apache Kafka fits — using Brightvault, the digital bank we will build throughout this book.

Meet Brightvault

Every chapter in this book returns to the same fictional company, so it is worth getting to know it now. Brightvault is a digital bank that operates in several countries. It has no branches: customers open accounts in a mobile app, receive a debit card, pay in shops and online, send transfers to friends, and expect an instant notification for every movement of money. Behind that simple experience sits a set of backend services, each owned by a small team:

- **accounts** — opens and closes accounts, manages account status and limits.
- **cards** — receives card authorization requests from the card network gateway and approves or declines them.
- **payments** — handles transfers between accounts and to other banks.
- **ledger** — the double-entry book of record; every debit and credit ends up here, and balances are derived from it.
- **fraud** — scores transactions in real time and raises alerts.
- **notifications** — sends push messages, emails, and SMS.
- **customer-profile**, **statements**, and **analytics** — customer data, monthly statements, and reporting.
- **core-banking** — a legacy PostgreSQL database inherited from an earlier platform, which Brightvault is gradually replacing.

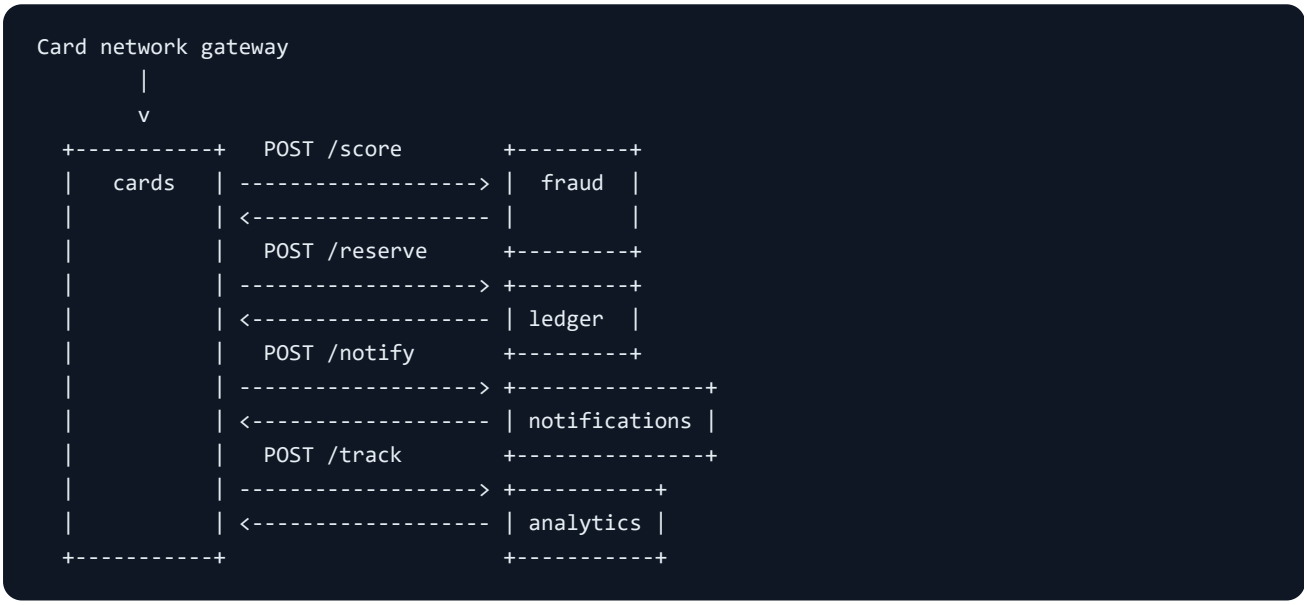
Brightvault is a good teaching example because banking exercises almost everything interesting about event-driven systems. Card authorizations arrive in a high-volume stream with tight latency budgets. Fraud detection needs to correlate events over time windows. Balances must be exactly right, which forces us to think carefully about duplicates and ordering. Regulators want an auditable history of everything that happened. And the legacy database cannot simply be switched off, so we need ways to integrate old and new systems safely.

In this chapter we will look at Brightvault the way many real companies look at their own systems a few years in: built on synchronous request/response calls, working, but increasingly painful to change and operate. Then we will redesign one flow around events and examine what we gain and what we pay.

The Request/Response World

The dominant style of service-to-service communication is the synchronous call: service A sends an HTTP or gRPC request to service B and waits for the response. It is familiar, easy to reason about, and well supported by every framework. For many interactions it is exactly the right tool. When the mobile app asks "what is my balance?", a synchronous query that returns the answer is the natural design.

Problems appear when synchronous calls are used to *propagate facts* through a system — when one service does something and then calls every other service that needs to know about it. Consider what happens at Brightvault when a customer taps their card at Harbor Coffee.



The `cards` service receives the authorization request, calls `fraud` to score it, calls `ledger` to reserve the funds, calls `notifications` to push a message to the customer, and calls `analytics` to record the purchase. Only then does it answer the card network. In code, the handler looks roughly like this:

AuthorizationHandler.java (synchronous version)

```
package com.brightvault.cards;

import java.time.Instant;

public class AuthorizationHandler {

    private final FraudClient fraud;
    private final LedgerClient ledger;
    private final NotificationClient notifications;
    private final AnalyticsClient analytics;
```

```
    public AuthorizationHandler(FraudClient fraud, LedgerClient ledger,
                                NotificationClient notifications,
                                AnalyticsClient analytics) {
        this.fraud = fraud;
        this.ledger = ledger;
        this.notifications = notifications;
        this.analytics = analytics;
    }

    public AuthorizationResult authorize(AuthorizationRequest req) {
        FraudScore score = fraud.score(req);              // blocking call 1
        if (score.value() > 0.9) {
            return AuthorizationResult.declined("SUSPECTED_FRAUD");
        }
        ledger.reserve(req.accountId(), req.amountMinor(),
                       req.currency());                   // blocking call 2
        notifications.send(req.accountId(),
                "Paid " + req.formattedAmount()
                + " at " + req.merchantName());           // blocking call 3
        analytics.track(req, Instant.now());              // blocking call 4
        return AuthorizationResult.approved();
    }
}
```

This code is easy to read, and that is its main virtue. But look at what it quietly implies.

Temporal coupling

Every downstream service must be available at the moment the card is used. If `analytics` is being redeployed, or its database is slow, the purchase at Harbor Coffee either fails or waits. That is absurd from a business point of view: analytics has nothing to do with whether the customer can buy a coffee, yet it sits on the critical path. This is **temporal coupling** — two services must both be up and responsive at the same time for the operation to succeed.

The availability arithmetic is unforgiving: the chain is available only when every dependency is, so its combined availability is lower than that of its weakest link. Latency compounds too. Sequential calls add their latencies together, and the card network gives the bank only a limited time to answer. Timeouts, retries, and circuit breakers help, but each is extra code that exists only because the caller knows too much about its callees.

Knowledge coupling and the growing fan-out

The `cards` service must know every consumer of the "card was authorized" fact: their addresses, their APIs, their error semantics. When the `statements` team needs card transactions too, they must ask the cards team to add a fifth call. When the `fraud` team wants to see declined authorizations as well as approved ones, that is another change in `cards`. The producer of a fact becomes a bottleneck for every team that wants to react to it. Organizationally, this is often the most painful problem: teams queue up behind one another's release schedules.

Partial failure and inconsistency

What happens if the ledger reservation succeeds but the notification call times out? Did the notification go out or not? Should `cards` retry, possibly sending a duplicate push message? Should it roll back the ledger reservation? Synchronous chains have no natural answer to partial failure. Each call is a separate network operation with its own failure modes, and there is no transaction spanning them. The typical result is a collection of ad hoc compensations and a steady trickle of inconsistencies that someone reconciles by hand.

Lost history

Finally, once the calls complete, the fact itself is gone. The analytics database has a row, the ledger has a reservation, the notification service has a log line. But there is no single, durable record that says "card `card_51XR` was authorized for 3.40 EUR at Harbor Coffee at 08:14:03 UTC". If a new team wants to build a feature from the last six months of card activity, there is nothing to replay. They must either scrape other services' databases or start collecting from today.

> **NOTE**
>
> None of this means request/response is bad. Queries that need an immediate answer ("is this card blocked?"), user-facing reads, and interactions with external partners that expect HTTP are often best served synchronously. The problem is using synchronous calls as the *only* integration mechanism, especially for broadcasting facts that many parties care about.

Thinking in Events

An **event** is a record of something that happened: a fact, stated in the past tense, that cannot be changed after the fact. "Card authorized." "Transfer requested." "Ledger entry posted." Events are not requests for someone to do something; they are announcements. The service that owns a piece of business logic publishes the event, and any interested service can react to it — now, later, or years from now.

Here is the Harbor Coffee authorization as an event. We will use this envelope structure throughout the book: every event carries an identifier, a type, a timestamp, its source, a schema version, and a correlation identifier that ties related events together, followed by the payload.

CardAuthorized event (JSON)

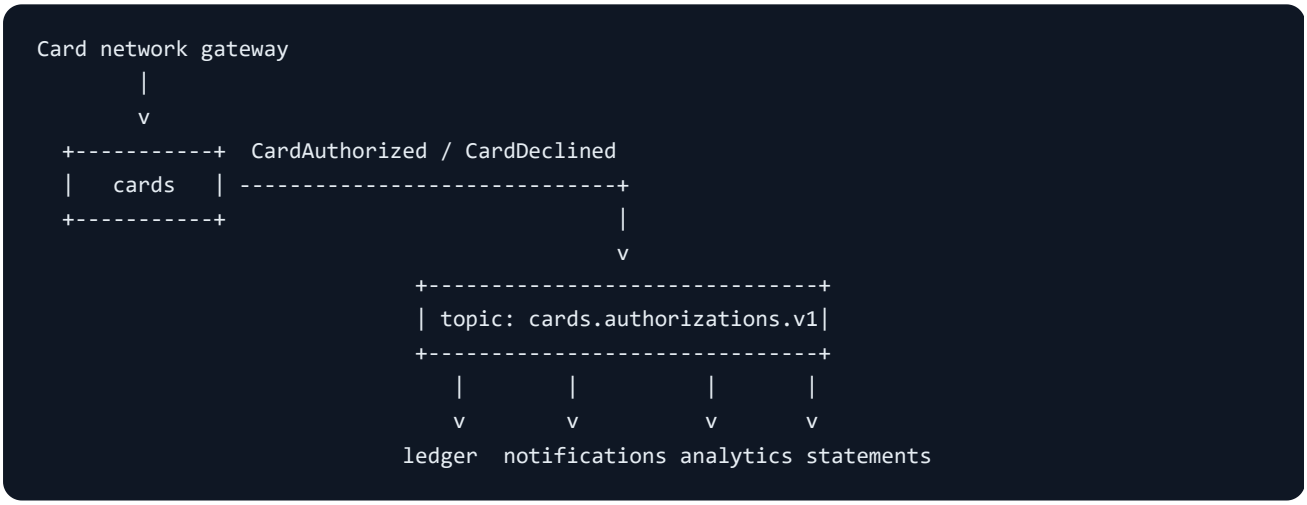
```
{
  "eventId": "6f1c2a9e-3b7d-4c55-9a0e-2d8f4b7e1c90",
  "eventType": "CardAuthorized",
  "occurredAt": "2026-03-14T08:14:03.412Z",
  "source": "cards",
  "schemaVersion": 1,
  "correlationId": "c0a8f3d2-51e4-4b8b-8a57-7f0c9e2b6d11",
  "payload": {
    "cardId": "card_51XR",
```

```
    "accountId": "acc_7Q2M9K",
    "merchantName": "Harbor Coffee",
    "merchantCategory": "5814",
    "amountMinor": 340,
    "currency": "EUR",
    "authorizationCode": "A7K2Q9"
  }
}
```

Notice two conventions that will recur. Money is an integer number of minor units (340 cents, not 3.40) plus an ISO currency code — never a floating-point number, because binary floating point cannot represent most decimal fractions exactly. And the timestamp is ISO 8601 in UTC, because a bank operating in several countries cannot afford ambiguity about when something happened. Chapter 2 examines event modeling in depth; for now, what matters is the shift in responsibility.

Publish once, react many times

In an event-driven design, `cards` does its own job — decide whether to approve — and publishes the outcome. It does not call notifications, analytics, or statements. Those services subscribe to the stream of card events and act on their own.



Adding the `statements` service now requires no change to `cards` at all. The statements team subscribes to `cards.authorizations.v1` and starts consuming. If analytics is down for an hour, card payments are unaffected; analytics catches up when it returns, because the events wait for it. The producer no longer knows or cares how many consumers exist.

Some interactions still need a synchronous answer. The card network wants an approve/decline decision within its deadline, and the fraud score feeds into that decision. A realistic Brightvault design keeps the fraud check on the critical path (possibly as a fast in-process model or a low-latency call) while moving everything that does not affect the decision — notifications, analytics, statements, most ledger bookkeeping — behind the event stream. Event-driven architecture is not all-or-nothing. It is a deliberate decision about which interactions need an immediate answer and which only need to know that something happened.

Three ways to use events

It helps to distinguish three patterns that people loosely call "event-driven", because they have very different consequences.

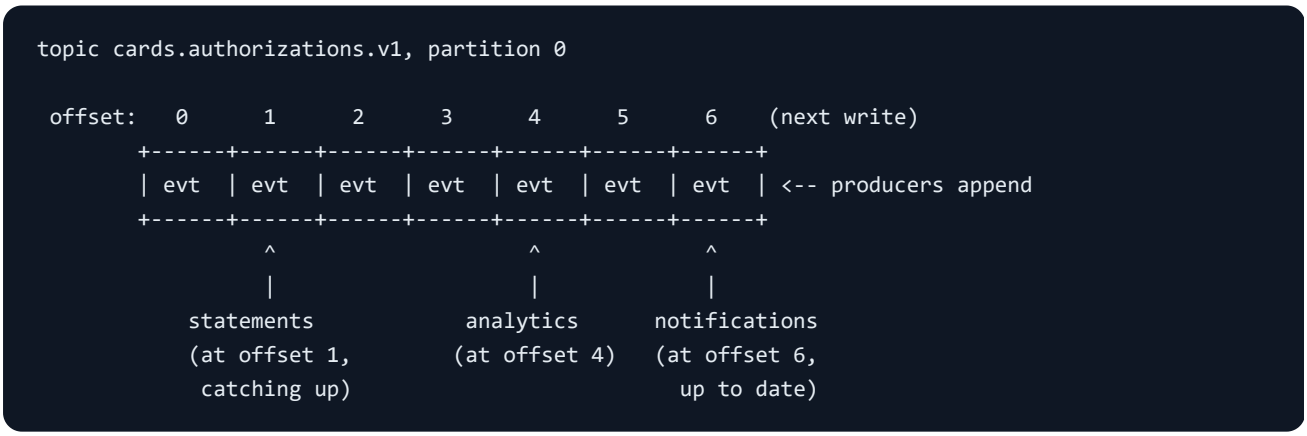
1. **Event notification.** The event is a thin signal — "account `acc_7Q2M9K` changed" — and consumers call back to the source to fetch details. This decouples the trigger but not the data: the source must still serve queries from everyone who reacts, and temporal coupling sneaks back in through the callbacks.
2. **Event-carried state transfer.** The event carries the data consumers need, as in our `CardAuthorized` example. Consumers can build their own local views without calling back. This is the pattern we will favor for most Brightvault events. It costs some duplication of data across services, which is a deliberate trade for autonomy.
3. **Event sourcing.** The events *are* the system of record. The current state of an account is derived by replaying its ledger entries. Chapter 23 covers event sourcing and CQRS in depth; it is powerful but not something every service needs.

Many systems mix all three. What matters is choosing deliberately and knowing which one each topic represents, because consumers make different assumptions about each.

What an Event Log Gives You

You could publish events through many kinds of infrastructure: a traditional message queue, a cloud notification service, even a database table that consumers poll. Apache Kafka takes a specific approach that shapes everything in this book: it stores events in a **durable, append-only, partitioned log**.

A log, in this sense, is the simplest possible storage structure: an ordered sequence of records to which you can only append. Each record gets a sequential position number called an **offset**. Readers do not remove records when they read them; each reader simply remembers the offset it has reached.



This simple structure has far-reaching consequences.

- **Many independent readers.** Because reading does not consume the record, any number of consumers can read the same events at their own pace. The notifications service can be up to date while the statements service is still processing yesterday's batch.
- **Replay.** A consumer can rewind its offset and reprocess history — to rebuild a corrupted database, backfill a new feature, or fix a bug and recompute results. A new service can start from the beginning of the retained log and build its state from scratch.
- **Retention by time or size, not by consumption.** Events stay in the log for a configured period (seven days by default in Kafka, often configured much longer) regardless of whether anyone has read them. Some topics can be retained indefinitely or compacted so that only the latest value per key survives — useful for things like the current balance of every account.
- **Ordering within a partition.** Records in one partition are read in exactly the order they were written. By routing all events for one card or account to the same partition, you get a reliable per-entity order, which is essential when a debit must not be processed before the account was opened.
- **Horizontal scale.** A topic is split into many partitions spread across many servers, so both writes and reads scale by adding partitions and machines. Chapter 3 explains exactly how.

Log versus queue

Developers who know traditional message brokers sometimes expect Kafka to behave like a queue, where a message is delivered to one worker and deleted once acknowledged. The differences are important enough to lay out side by side.

Aspect	Classic message queue	Kafka event log
After a read	Message is removed once acknowledged	Record stays; reader advances its offset
Multiple consumers	Competing workers share messages; fan-out needs extra setup	Any number of independent consumer groups
Replay	Usually not possible after acknowledgment	Rewind offsets within retention
Ordering	Often per queue, weakened by parallel workers	Strict within a partition
Per-message routing	Rich (routing keys, priorities, delays)	Minimal; routing is by topic and key
Scaling consumers	Add workers freely	Bounded by partition count per group

Neither model is universally better. Queues excel at distributing independent tasks; logs excel at broadcasting facts, preserving history, and ordering per key. Kafka can emulate work queues through consumer groups, and recent versions add an early-access feature called share groups (sometimes called "Queues for Kafka") for queue-like consumption; its status varies by version, so check the documentation for yours. Throughout this book we use Kafka primarily as a log, because that is where it is strongest.

Streams, Not Snapshots

There is a deeper idea behind event-driven architecture that is worth making explicit. Traditional systems store *state*: a table of accounts with a balance column. Events describe *changes*: every debit and credit. The two are related by a simple equation — the current state is the result of applying all changes, in order, to an initial state.

```
ledger.entries.v1 (key acc_7Q2M9K)            derived balance
-----------------------------------           ---------------
LedgerEntryPosted  credit  +250000 EUR   ->       250000
LedgerEntryPosted  debit       -340 EUR   ->       249660
LedgerEntryPosted  debit      -1899 EUR   ->       247761
LedgerEntryPosted  credit     +5000 EUR   ->       252761
```

If you have the stream of changes, you can always reconstruct the state. The reverse is not true: a balance of 252761 tells you nothing about how the account got there. This asymmetry is why events are such a valuable foundation. From one stream of ledger entries, Brightvault can derive current balances for the app, monthly statements for customers, aggregated reports for regulators, and features for the fraud model — each one a different view of the same facts, each maintained by a different team.

Instead of one central database whose tables every service queries, the log of changes becomes the shared source of truth, and each service maintains the derived views it needs — and is responsible for keeping them correct, which is what much of Parts III and IV are about.

Redesigning a Brightvault Flow

Let us make this concrete by redesigning card authorizations. The `cards` service keeps its synchronous interaction with the card network gateway — the network expects an answer — but instead of calling four services, it publishes one event per decision to `cards.authorizations.v1`, keyed by `cardId`. Keying by card means every event for one card lands in the same partition and is seen by consumers in order.

We will write fully configured producers and consumers in Chapter 5 and examine every setting in Chapters 6 and 7. Here is a first sketch that shows the shape of the code. The payload is a Java record; for now we serialize it to JSON with a helper and send it as a string.

CardEventPublisher.java

```
package com.brightvault.cards;

import java.util.Properties;
import java.util.UUID;
import java.time.Instant;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;
```

```
public class CardEventPublisher implements AutoCloseable {

    public record CardAuthorized(String cardId, String accountId,
                                 String merchantName, long amountMinor,
                                 String currency, String authorizationCode) {}

    private static final String TOPIC = "cards.authorizations.v1";
    private final KafkaProducer<String, String> producer;
    private final JsonEnvelope json = new JsonEnvelope("cards", 1);

    public CardEventPublisher(String bootstrapServers) {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                  StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                  StringSerializer.class.getName());
        // acks=all and idempotence are already the defaults since Kafka 3.0;
        // we state them explicitly so the intent is visible in code review.
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
        this.producer = new KafkaProducer<>(props);
    }

    public void publish(CardAuthorized event, String correlationId) {
        String value = json.wrap(UUID.randomUUID(), "CardAuthorized",
                                 Instant.now(), correlationId, event);
        ProducerRecord<String, String> record =
                new ProducerRecord<>(TOPIC, event.cardId(), value);
        producer.send(record, (metadata, error) -> {
            if (error != null) {
                // Real code: alert, and decide whether to fail the request.
                System.err.println("Publish failed: " + error.getMessage());
            }
        });
    }

    @Override
    public void close() {
        producer.close();
    }
}
```

And here is the notifications side. It subscribes to the topic as part of a **consumer group** named `notifications-service`. Kafka tracks, per group, which offset has been processed in each partition. If we run three instances of the notifications service with the same group id, Kafka divides the partitions among them; if one instance crashes, its partitions move to the survivors. Chapter 7 covers the mechanics.

CardNotificationConsumer.java

```
package com.brightvault.notifications;

import java.time.Duration;
import java.util.List;
import java.util.Properties;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

public class CardNotificationConsumer {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "notifications-service");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                  StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                  StringDeserializer.class.getName());
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");

        PushSender push = new PushSender();
        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props)) {
            consumer.subscribe(List.of("cards.authorizations.v1"));
            while (true) {
                ConsumerRecords<String, String> records =
                        consumer.poll(Duration.ofMillis(500));
                for (ConsumerRecord<String, String> rec : records) {
                    push.sendForCardEvent(rec.key(), rec.value());
                }
                consumer.commitSync(); // mark the batch as processed
            }
        }
    }
}
```

The analytics and statements services look almost identical, each with its own group id. None of them required a change in `cards`. Each one can be deployed, scaled, paused, and replayed independently.

> **TIP**
>
> Give every consumer group a stable, descriptive name tied to the service and purpose, such as `notifications-service` or `statements-card-ingest`. Group ids appear in monitoring dashboards, lag alerts, and offset reset commands. A name like `group1` or a randomly generated id makes operations much harder and can leave orphaned offsets behind every time a service restarts.

What we gained

- **Resilience.** The card authorization path depends on Kafka and on the fraud check, not on notifications, analytics, or statements. A slow consumer causes lag, not declined payments.
- **Independent evolution.** New consumers are added without changing or redeploying the producer. The cards team is no longer on everyone's critical path.
- **Buffering.** A burst of purchases at lunchtime is absorbed by the log. Consumers process at their own rate and catch up.
- **History.** The topic is a durable record of every authorization within its retention period. A new team can replay it to build a feature.

What we paid

It would be dishonest to present this as free. Event-driven designs replace some problems with others, and you should adopt them with eyes open.

- **Eventual consistency.** Immediately after the authorization, the notification may not have been sent and the statements view may not show the purchase. The system converges, typically within a short time, but there are moments when different services disagree. User interfaces and business processes must tolerate this.
- **Duplicates.** Most practical configurations deliver each event *at least once*. A consumer that crashes after sending a push notification but before committing its offset will see the event again on restart. Consumers must be idempotent — safe to run twice — or deduplicate. Chapters 11 and 15 go deep on this.
- **The dual-write problem.** If `cards` writes to its own database and then publishes to Kafka, a crash between the two leaves them inconsistent. The transactional outbox pattern in Chapter 14 solves this properly.
- **Harder end-to-end reasoning.** In the synchronous version, you could read one method to see the whole flow. Now the flow is spread across services that react to events. You need correlation ids, tracing, and good documentation of who consumes what.
- **Contracts that outlive code.** Events are stored, replayed, and consumed by teams you may not know. Changing an event's shape carelessly can break consumers or make old events unreadable. Chapters 8 and 9 cover schemas and evolution.
- **New infrastructure.** Kafka is a distributed system that must be sized, monitored, secured, and upgraded — whether you run it yourself or use a managed service.