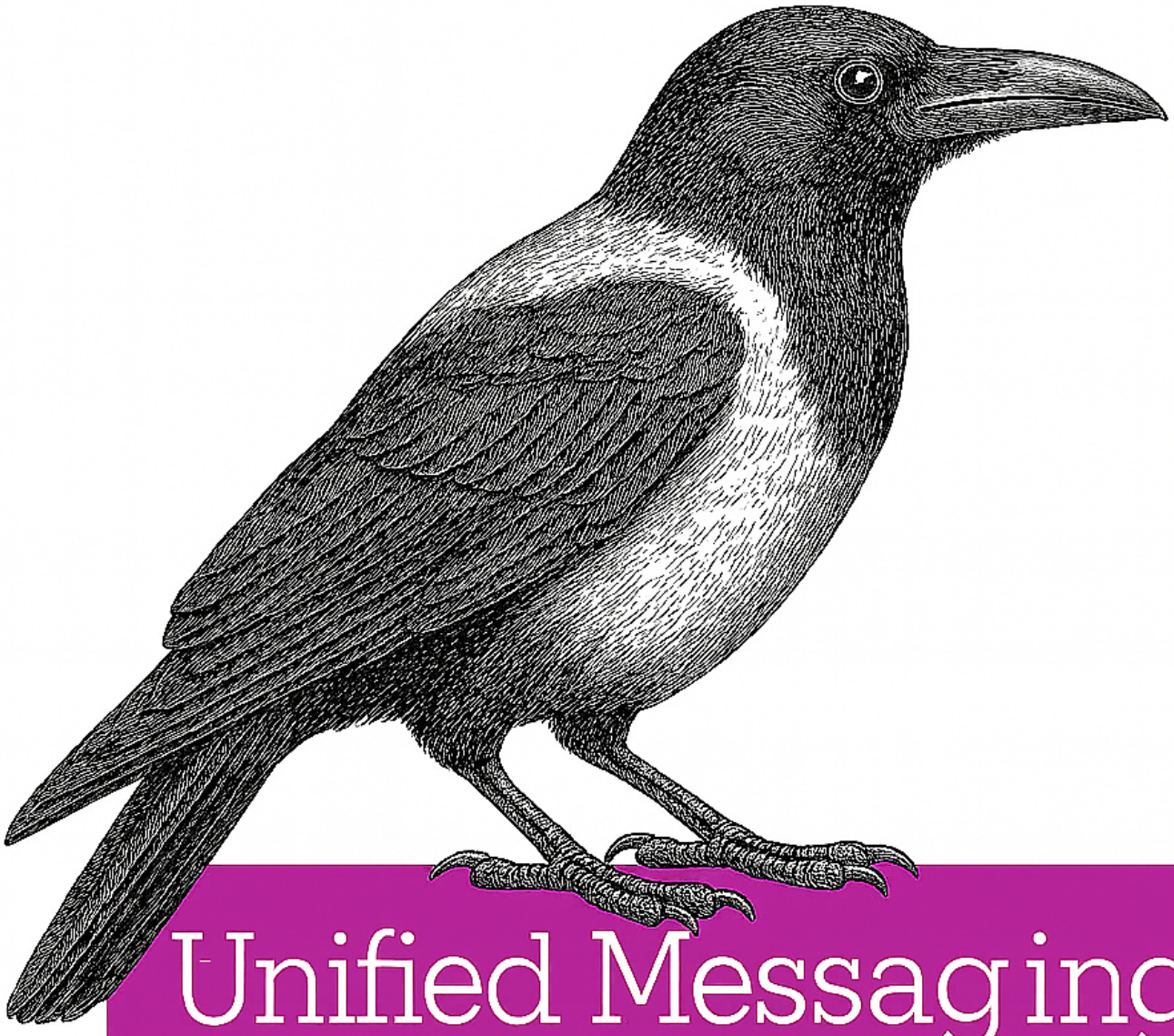

# Unified Messaging with OmniQueue

## A Comprehensive Guide to Broker-Agnostic Messaging Systems and Practical Implementations

Darius Max

# Unified Messaging with OmniQueue

## *A Comprehensive Guide to Broker-Agnostic Messaging Systems and Practical Implementations*

Darius Max

# Table of Contents

# Preface

## Why This Book Exists

In the last decade, messaging systems have become the invisible backbone of modern software. From high-frequency trading platforms to e-commerce checkout flows, from IoT telemetry ingestion to social media fanout, **asynchronous message passing** underpins much of the world's critical software.

But there's a catch: the messaging landscape is **fragmented**. Each broker — RabbitMQ, Kafka, NATS, ActiveMQ, SQS, and so on — comes with its own API, terminology, quirks, and operational mindset. Switching from one broker to another can mean a complete rewrite of integration code, along with a steep re-learning curve for operations.

**OmniQueue** was born from a simple but ambitious idea:

> "What if there was a single, unified API that could speak to any messaging system — brokered or brokerless — without sacrificing the power and unique features of each?"
>
> — Founder's Notes

This book is the culmination of that idea.

It's **both** a deep dive into the theory and practice of messaging systems **and** the definitive guide to OmniQueue — a TypeScript-based, broker-agnostic messaging API.

## Who This Book Is For

This book is written for engineers, architects, and operations teams who:

- Work with messaging systems regularly.
- Need to support multiple brokers across environments.
- Migrate between brokers due to scaling, cost, or vendor lock-in.

- Want to abstract messaging complexity without losing control.

We've structured the content so that:

- **Beginners (0–2 years)** gain a solid grounding in messaging fundamentals.
- **Intermediate practitioners (2–5 years)** learn deep operational patterns and broker-specific optimizations.
- **Senior engineers (5+ years)** explore high-scale, mission-critical messaging architecture and troubleshooting.
- **OmniQueue Mastery Mode** sections give you direct, in-depth knowledge to extend, tune, and contribute to the OmniQueue project.

# What This Book Covers

We will explore:

- Core messaging concepts: queues, topics, routing, delivery semantics, and reliability.
- Detailed broker chapters for RabbitMQ, Kafka, NATS JetStream, ActiveMQ Artemis, ZeroMQ, BullMQ, AWS SQS/SNS, Azure Service Bus, and Apache Pulsar.
- OmniQueue's architecture, design decisions, and implementation patterns.
- Mapping native broker concepts to OmniQueue's unified API.
- Advanced OmniQueue usage: scaling, plugin development, observability, and disaster recovery.
- Real-world case studies, complete with war stories from production incidents.
- How to extend and contribute to OmniQueue.

Throughout the book, we'll balance **conceptual clarity** with **practical hands-on examples**.

# Why OmniQueue?

Traditional messaging integrations often fall into one of two traps:

1. **Tight coupling to a specific broker.** This limits future flexibility, creates vendor lock-in, and complicates migrations.
2. **Overly generic abstractions.** These hide broker-specific features, leading to lowest-common-denominator APIs.

OmniQueue aims to solve both problems:

- **Unified Core API** for sending, receiving, publishing, and subscribing.
- **Broker Adapters** that map directly to native features.

- **Optional Features** for advanced tuning, scaling, and recovery.
- **Plugin Architecture** for extending to new brokers or integrating custom logic.

## A Note on Style

This book is written in the spirit of O'Reilly guides: technical, clear, and actionable — with just enough narrative to keep you awake during the heavy parts.

You'll see **TypeScript code** for OmniQueue, **Mermaid diagrams** for system flows, and **cheat sheets** for quick reference.

## Reader Expectations

- You don't need to be a messaging expert — but some familiarity with distributed systems will help.
- You should be comfortable reading TypeScript and basic CLI commands.
- Expect real-world complexity: we'll talk about failures, quirks, and debugging nightmares.
- We'll show both the "happy path" and the "when it all goes wrong" path.

## How This Book Is Organized

| Part | Focus |
|---|---|
| I | Messaging fundamentals and OmniQueue core API. |
| II | Deep dives into individual brokers with OmniQueue mapping. |
| III | Advanced OmniQueue usage, scaling, and tuning. |
| IV | Real-world implementations and case studies. |
| V | Extending and contributing to OmniQueue. |
| Appendices | Quick references, glossary, and disaster checklists. |

## Our Promise

By the end of this book, you'll be able to:

- Understand the trade-offs between major messaging systems.

- Confidently implement broker-agnostic messaging in your projects.
- Operate and troubleshoot OmniQueue in production.
- Extend OmniQueue to support new brokers and patterns.

> Messaging is not just about moving data. It's about shaping the **lifeblood** of your system — reliably, predictably, and with intent.
>
> — From the Authors

# How to Use This Book

This book is both a **reference** and a **journey**. You can read it cover-to-cover or jump directly to the sections that matter most to your work.

We've designed it with multiple entry points, so whether you are brand-new to messaging or a seasoned architect managing a multi-broker fleet, you'll find value.

## Reading Paths by Experience Tier

We've tagged content for different levels of experience. If you identify your tier, you can follow the recommended path:

- **Beginner (0–2 years)** Start with Part I: **Messaging Fundamentals** (Chapters 1–3). These chapters introduce core concepts and get you hands-on with OmniQueue quickly. Then, read only the broker chapters (Part II) for the systems you'll use in your current work.
- **Intermediate (2–5 years)** You can skim Part I for a refresher, then dive deep into the broker chapters (Part II). Pay attention to **War Stories** and **Cheat Sheets** — they'll save you pain in production. Follow with Part III to learn scaling, monitoring, and advanced OmniQueue patterns.
- **Senior Engineer / Architect (5+ years)** Jump to the broker chapters you care about most, but don't skip Part III — it's packed with operational patterns for high-scale environments. Use Part IV's case studies for inspiration and validation of architectural decisions.
- **OmniQueue Mastery Mode** These sections are sprinkled throughout the book for readers who want to extend OmniQueue itself. You'll find them in both broker chapters and advanced sections.

## Part-by-Part Guide

| Part | How to Use It |
|---|---|
| I | Foundation for messaging concepts and OmniQueue basics. Best for new readers. |

| Part | How to Use It |
| --- | --- |
| II | Broker-specific knowledge. Each chapter is standalone — read only what you need. |
| III | Advanced OmniQueue usage: scaling, observability, disaster recovery. |
| IV | Real-world case studies — learn from actual deployments and failures. |
| V | Contribution and extension — for those building on or contributing to OmniQueue. |
| Appendices | Quick-reference materials, glossary, and operational checklists. |

## Code Examples

All OmniQueue examples are in **TypeScript**. We assume you have basic familiarity with `npm`, `pnpm`, or `yarn` and Node.js tooling.

Code listings are annotated with comments to explain **why** something is done, not just **how**. For example:

```
// Sending a message with OmniQueue
await omni.send({
  destination: 'orders',
  body: { id: '1234', status: 'confirmed' },
  group: 'order-processing', // Required for consumers
  ensure: true, // Create destination if it doesn't exist
});
```

Whenever we compare OmniQueue to a native broker API, we show them side-by-side for clarity.

## Diagrams and Visuals

We use **Mermaid diagrams** for flows, topologies, and message lifecycles. These are rendered in the book, but if you're reading the raw AsciiDoc, you can copy the code into any Mermaid-compatible renderer to visualize it.

## Cheat Sheets and War Stories

At the end of each broker chapter:

- **Cheat Sheet** — quick reference for commands, configs, metrics, and a **disaster checklist**.

- **War Stories** — short narratives of real-world incidents, failures, and their solutions.

## Cross-References

Throughout the book, cross-references (e.g., “see <<_chapter_3>>”) help you connect concepts. We strongly encourage following them, especially when exploring advanced topics that build on earlier material.

## What You’ll Need

- **Node.js** 18+
- A package manager (`npm`, `pnpm`, or `yarn`)
- Docker (optional, but useful for running brokers locally)
- Basic terminal familiarity

You don’t need to install every broker at once — we’ll give setup instructions per chapter.

## Our Advice

- **Don’t just read — run the examples.** Messaging is best learned by sending and receiving real messages.
- **Experiment with failure.** Kill brokers, drop connections, overload queues — see how systems behave under stress.
- **Adapt patterns to your context.** There’s no universal “right” way; use these patterns as starting points.

# Part I: Messaging Fundamentals

This part lays the essential groundwork for mastering OmniQueue and broker-agnostic messaging systems. Before we dive into advanced patterns, scaling strategies, or broker-specific optimisations, we need to develop a shared vocabulary and mental model.

You'll start by exploring what "messaging" really means in a distributed systems context — moving beyond the abstract to see how messages, queues, topics, and streams actually behave in the wild. We'll break down the trade-offs between brokered and brokerless architectures, delivery semantics, and the key reliability levers at your disposal.

From there, we'll move into the architectural core of OmniQueue itself. You'll learn how its unified API abstracts away broker differences without hiding the critical operational controls you need. We'll also unpack concepts like **mandatory consumer groups** and **work-sharing semantics**, so you can design pipelines that scale cleanly without sacrificing correctness.

By the end of this part, you'll not only understand **how** to send, receive, and manage messages — you'll know **why** these design choices matter, and how to apply them confidently across any broker environment.

# Chapter 1. Chapter 1: Introduction

## 1.1. The Role of Messaging in Modern Systems

Messaging is the circulatory system of distributed software.

Whether you are processing financial transactions, streaming telemetry from millions of IoT devices, or fanning out notifications to mobile users, **messages** are how your system communicates internally and externally.

Unlike synchronous HTTP calls, messaging systems decouple producers and consumers in **time** and **space**. This makes them a cornerstone of:

- **Scalability** — handle bursts by buffering messages in queues, logs, or streams.
- **Resilience** — retry failed deliveries without dropping data.
- **Loose coupling** — evolve services independently.
- **Asynchronous workflows** — free services from waiting on each other.

Modern architectures like **microservices**, **event-driven systems**, and **CQRS/ES** (Command Query Responsibility Segregation / Event Sourcing) rely heavily on messaging.

## 1.2. Brokered vs. Brokerless Messaging

Messaging systems come in two broad categories:

*Table 1. Brokered Vs Brokerless*

| Type | Description | Examples |
|---|---|---|
| Brokered | Messages flow through a central server (the broker) that stores, routes, and delivers them. | RabbitMQ, Kafka, NATS JetStream, ActiveMQ Artemis, Azure Service Bus, Apache Pulsar |

| Type | Description | Examples |
|---|---|---|
| Brokerless | Producers send messages directly to consumers without an intermediary. Often implemented with peer-to-peer sockets. | ZeroMQ, nanomsg |

**Brokered systems** offer durability, routing flexibility, and operational control. **Brokerless systems** excel in speed, simplicity, and deployment independence.

## 1.3. The Fragmentation Problem

Each broker ecosystem has its **own**:

- API shape and naming conventions
- Delivery semantics (at-most-once, at-least-once, exactly-once)
- Terminology: **queue** vs **topic**, **exchange** vs **stream**, **subject** vs **channel**
- Operational model and deployment pattern
- Performance characteristics and tuning knobs

Moving from RabbitMQ to Kafka isn't just a config change — it's often a **rewrite**. Even within the same organization, different teams may adopt different brokers for historical or domain-specific reasons.

This leads to:

- **Code duplication** — multiple code paths for different brokers.
- **Vendor lock-in** — hard to migrate away.
- **Operational complexity** — separate tooling, metrics, and expertise required.

## 1.4. Enter OmniQueue

OmniQueue addresses the fragmentation problem with a **unified, broker-agnostic API** centered on **publish/subscribe with mandatory consumer groups**.

Producers **publish** to a **topic**. Each **consumer group** subscribed to that topic receives a copy of every message; within a given group, only one consumer processes a given message.

Handlers acknowledge with `ack()` or `nack(requeue?: boolean)`.

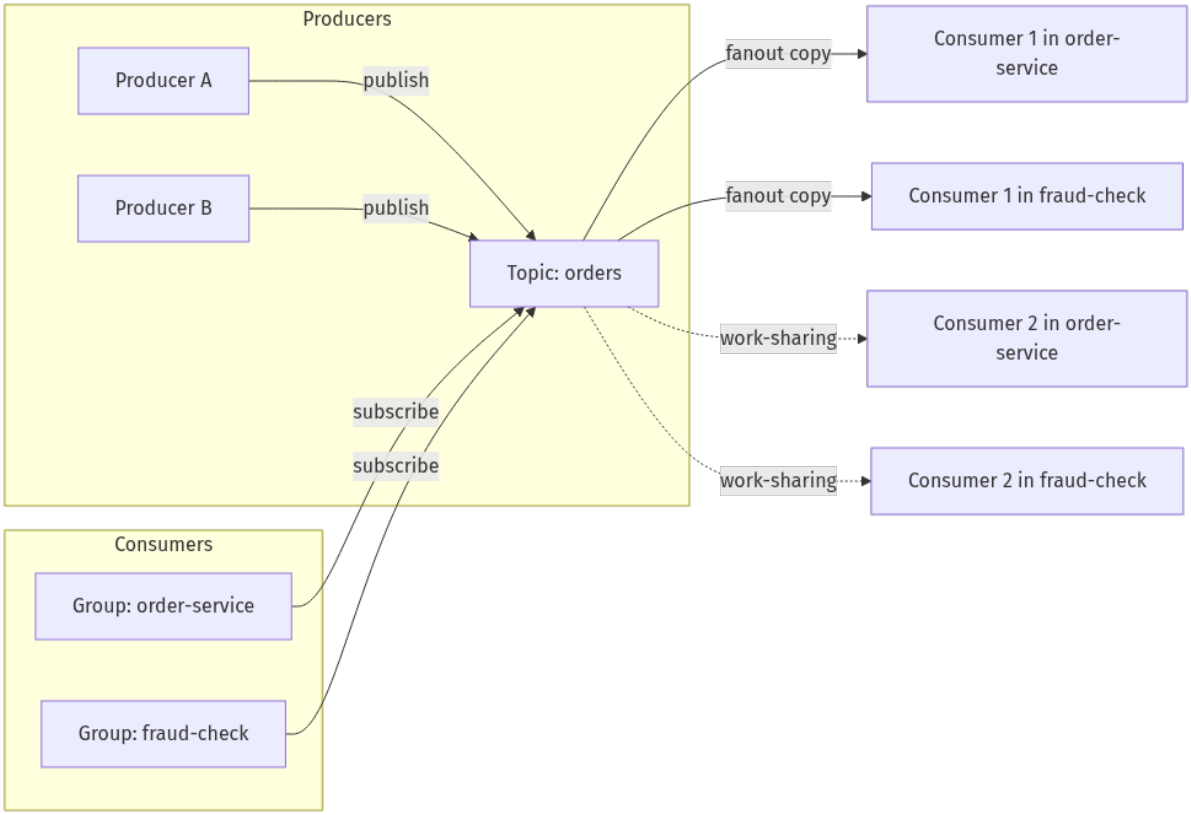


*Figure 1. A look on "topic" messaging*

With OmniQueue, your code can switch between brokers with minimal changes, while still leveraging advanced features through **broker adapters**.

## 1.5. A Simple Example

```
import { create } from '@omniqueue/core';
import '@omniqueue/rabbitmq'; // registers the "rabbitmq" adapter

// Create and init a broker instance
const broker = await create('rabbitmq', { url: 'amqp://localhost' });
await broker.init();

// Publisher: send one event to the topic "orders"
await broker.publish(
  'orders',
  {
    id: 'ulid-01JABCDEF...',
    body: { orderId: 'A123', status: 'confirmed' },
    headers: { 'x-source': 'checkout' },
  },
  {
    ensure: true, // lazily create the destination if missing
    // createOptions can pass through to the adapter (durable, partitions, etc.)
  }
```

```
);

// Subscriber: each GROUP receives a copy; work is shared inside the group.
await broker.subscribe(
  'orders',
  async (msg) => {
    try {
      console.log('Processing order', msg.body);
      await msg.ack();
    } catch (err) {
      await msg.nack(true); // requeue on error
    }
  },
  'order-service',    // <-- mandatory groupId
  { ensure: true }    // optional ConsumeOptions
);

// ...later, graceful shutdown
await broker.close();
```

Switching to Kafka, NATS, Pulsar, or another supported broker means changing only the adapter import and connection config — your core logic stays the same.

## 1.6. Who Uses OmniQueue?

OmniQueue is designed for:

- **Polyglot microservice environments** — where different teams choose different brokers.
- **Hybrid cloud architectures** — where on-prem and cloud-native brokers must coexist.
- **Migration projects** — moving from one broker to another without downtime.
- **Prototyping and R\&D** — quickly test message flows across multiple systems.

## 1.7. What This Chapter Sets Up

In the chapters ahead:

- **Chapter 2** will cover **Core Messaging Concepts** in detail — topics, routing, delivery guarantees, and reliability.
- **Chapter 3** will dissect **OmniQueue Architecture & Core API**, showing exactly how the abstraction is implemented with `publish`/`subscribe`.
- **Part II** will then dive deep into individual brokers.

The goal of this introduction is simple: Give you the “why” and the “what” before we explore the “how” and “when”. == Core Messaging Concepts

# 1.8. Messages, Queues, Topics, Streams

At the heart of any messaging system is the **message**: a discrete unit of data that moves from producer to consumer.

## 1.8.1. The Message Structure

While brokers differ in terminology and features, most messages contain:

*Table 2. Message components*

| Field | Purpose |
|---|---|
| **ID** | Uniquely identifies the message. For durability and traceability, many teams use ULID or UUID v7. Some brokers provide their own sequence IDs (Kafka offset, Pulsar ledgerId/entryId, etc.). |
| **Body** | The actual payload. Often JSON for ease of debugging, but formats like Avro or Protobuf are preferred in high-performance or schema-validated environments. |
| **Headers / Attributes** | Metadata about the message (e.g., `content-type`, source service, trace IDs for distributed tracing, tenant identifiers). These can influence routing (e.g., header-based exchange in RabbitMQ) or serve for observability. |

In OmniQueue, `body` is always JSON-serialisable to ensure adapter compatibility, but you can base64-encode binary payloads if needed.

## 1.8.2. Queues

A **queue** is a named, ordered collection of messages that delivers each message to exactly one consumer within a given group. When a message is acknowledged (`ack()`), it is removed from the queue (or marked processed).

**Key characteristics:**

- **FIFO order** by default, though some brokers allow priority or sharded ordering.
- **Work sharing** — multiple consumers in the same group share the load.
- **Backpressure** — unacknowledged messages accumulate in the queue.

**Real-world usage:**

- Task processing (image rendering, report generation).

- Decoupling request spikes from downstream processing.
- Buffering workloads when downstream systems are slow.

### 1.8.3. Topics

A **topic** is a named channel to which producers publish and consumers subscribe.

**Key characteristics:**

- **Fanout delivery** — every consumer group subscribed to a topic gets a full copy of the message stream.
- **Group semantics** — within a single group, messages are load-balanced to one consumer at a time.
- **Loose coupling** — producers don't need to know who consumes their messages.

**Real-world usage:**

- Event broadcasting (order events, user activity streams).
- Multi-service pipelines (analytics, fraud detection, notifications).
- Event sourcing feeds.

### 1.8.4. Streams

A **stream** is an append-only, immutable sequence of messages stored in durable storage. Consumers track their own **offset** (position) and can replay older messages at will.

Key characteristics:

- **Messages persist** for a retention period (e.g., 7 days) or indefinitely.
- **Multiple independent consumers** — each with their own offsets.
- **High throughput** by avoiding message deletion.

**Real-world usage:**

- Audit logging with replay.
- Rebuilding projections in CQRS systems.
- Time-series analytics pipelines.

### 1.8.5. Comparing Destinations

*Table 3. Queue Vs Topic Vs Stream*

| Type | Delivery Model | Storage & Retention | Replay Support |
|---|---|---|---|
| Queue | Point-to-point within a group | Message removed after ack | No |

| Type | Delivery Model | Storage & Retention | Replay Support |
|---|---|---|---|
| Topic | Publish/subscribe, fanout to groups | Broker-controlled, may persist temporarily | Sometimes (depends on broker) |
| Stream | Publish/subscribe with offset tracking | Long-term retention in log | Yes |

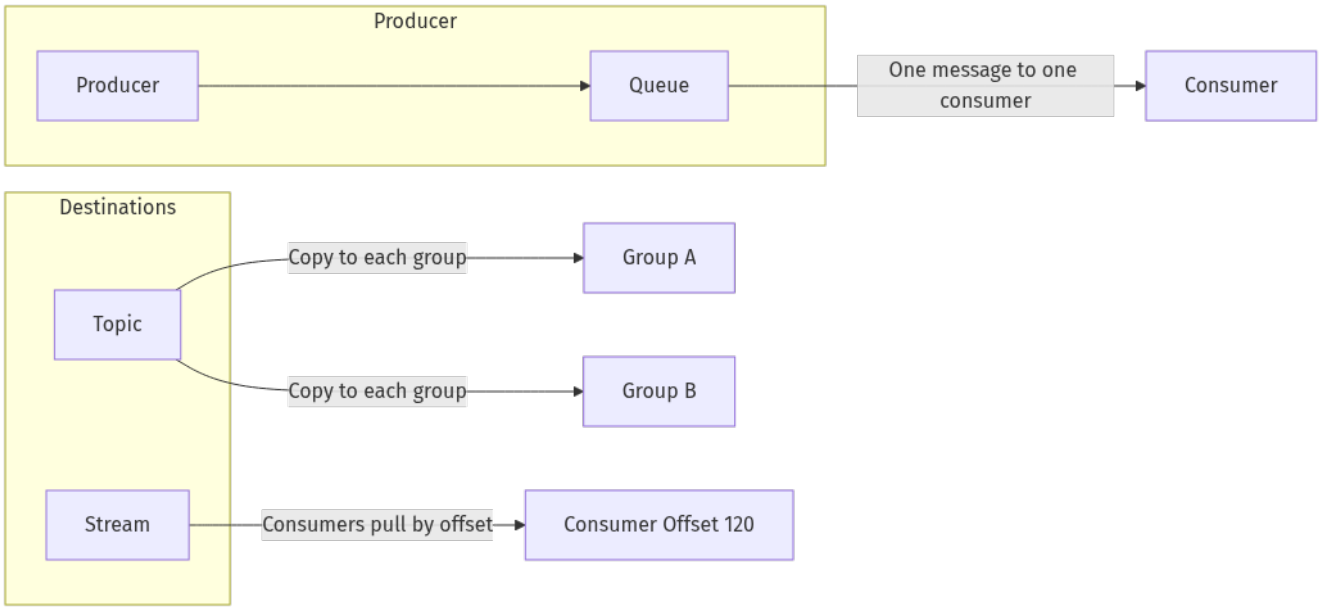


*Figure 2. Queue, Topic, Stream*

# 1.9. Brokers vs. Brokerless

## 1.9.1. What is a Broker?

A **broker** is a dedicated server (or cluster) that:

- Accepts messages from producers via network protocols (AMQP, Kafka protocol, MQTT, HTTP).
- Routes them based on metadata (exchange bindings, partition keys, headers).
- Stores them (in memory, disk, or distributed logs) for durability and later delivery.
- Delivers them to consumers, tracking acknowledgements.

Examples: RabbitMQ, Kafka, NATS JetStream, ActiveMQ Artemis, Azure Service Bus, Apache Pulsar.

**Advantages:**

- **Durability** — messages survive restarts or crashes.
- **Routing flexibility** — fanout, topic filters, header matching, partitioning.
- **Operational visibility** — metrics, queues, consumer lag.
- **Security controls** — authentication, authorization, encryption.

**Challenges:**

- Operational cost — needs monitoring, scaling, patching.
- Latency vs. durability trade-offs — fsyncs, replication.
- Complexity — tuning partitions, queues, consumers.

### 1.9.2. Brokerless Messaging

In **brokerless** systems, producers send messages directly to consumers without an intermediary server. Typically, these use **peer-to-peer sockets** or multicast protocols.

Example: ZeroMQ's PUB/SUB, PUSH/PULL patterns.

**Advantages:**

- **Ultra-low latency** — no queue persistence or broker hop.
- **No single point of failure** — as long as consumers are reachable.
- **Lightweight deployment** — often just a library in your app.

**Challenges:**

- No inherent durability — offline consumers miss messages.
- No centralized management — must handle discovery, retries, ordering yourself.
- Difficult scaling for complex topologies.

### 1.9.3. Visual Comparison

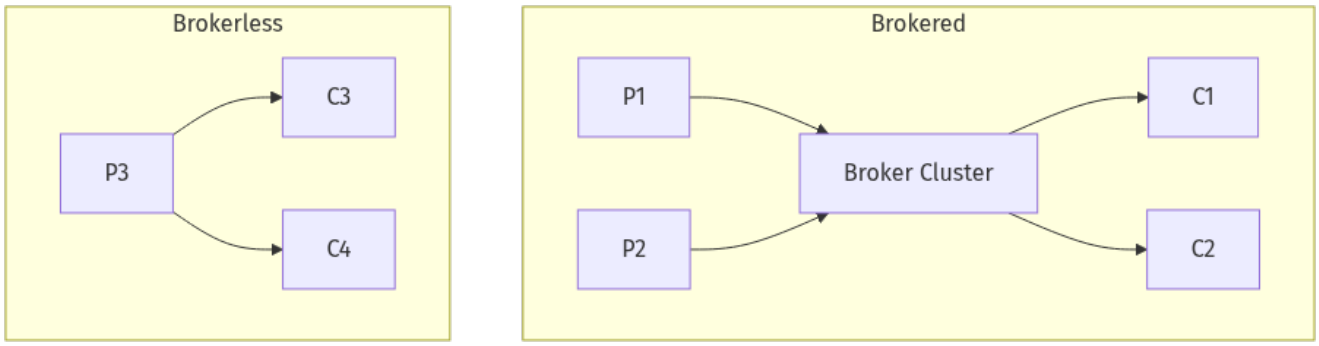


*Figure 3. Brokered Vs Brokerless*

**Brokered systems** shine in reliability and complexity management. **Brokerless systems** shine in simplicity and speed — but require discipline in application design.

OmniQueue supports **both** worlds. A `@omniqueue/zeromq` adapter can connect directly between processes without a broker, while `@omniqueue/kafka`, `@omniqueue/rabbitmq`, and others leverage durable, clustered brokers.

# 1.10. Delivery Semantics

Delivery semantics define **how many times** a message is delivered to a consumer, and under what guarantees. They are a cornerstone of messaging design — the choice affects performance, complexity, and data correctness.

## 1.10.1. At-most-once

**Definition:**

A message is delivered **zero or one time**, but never more than once. If the delivery fails, the message is lost — no retries.

**How it happens:**

1. The broker (or producer in brokerless systems) sends the message without waiting for acknowledgement.
2. No redelivery is attempted if the consumer fails mid-processing.

**Pros:**

- Lowest latency — no ack/nack overhead.
- Simplest implementation.

**Cons:**

- Data loss possible under failures.
- Not suitable for critical workflows.

**Real-world usage:**

- Non-critical telemetry where occasional loss is acceptable (e.g., live UI metrics).
- High-frequency, low-value sensor data.

## 1.10.2. At-least-once

**Definition:**

A message is delivered **one or more times** until it is acknowledged. Duplicates may occur.

**How it happens:**

1. The broker stores the message until it gets an explicit ack from the consumer.
2. If ack is not received within a timeout or connection drops, the broker redelivers.

**Pros:**

- No message loss under normal broker persistence guarantees.
- The default for most brokers (RabbitMQ, SQS, NATS JetStream).

**Cons:**

- Requires **idempotent** consumers — they must handle duplicates gracefully.
- Potential extra load from duplicate processing.

**Real-world usage:**

- Payment processing (with idempotency keys).
- Order event processing.
- Logging pipelines.

### 1.10.3. Exactly-once

**Definition:**

A message is delivered **exactly one time** — no duplicates, no losses.

**How it happens:**

1. Requires transactional coordination between producer, broker, and consumer.
2. The broker and consumer commit offsets/state atomically.

**Pros:**

- Simplest for consumers — no deduplication logic.

**Cons:**

- High complexity, often broker-specific.
- Lower throughput due to transactional overhead.
- Often misunderstood — "exactly-once" guarantees are fragile in distributed systems.

**Real-world usage:**

- Financial transaction settlement (Kafka + idempotent producers + transactions).
- Highly sensitive event processing.

### 1.10.4. OmniQueue Perspective

OmniQueue does not enforce a delivery semantic by itself — it passes through the broker's native behavior. However:

- At-least-once is the most common.
- You can achieve effectively-once delivery by combining at-least-once brokers with idempotent

consumer logic.

- Broker adapters may expose native exactly-once features (e.g., Kafka transactions) via `createOptions` or broker-specific APIs.

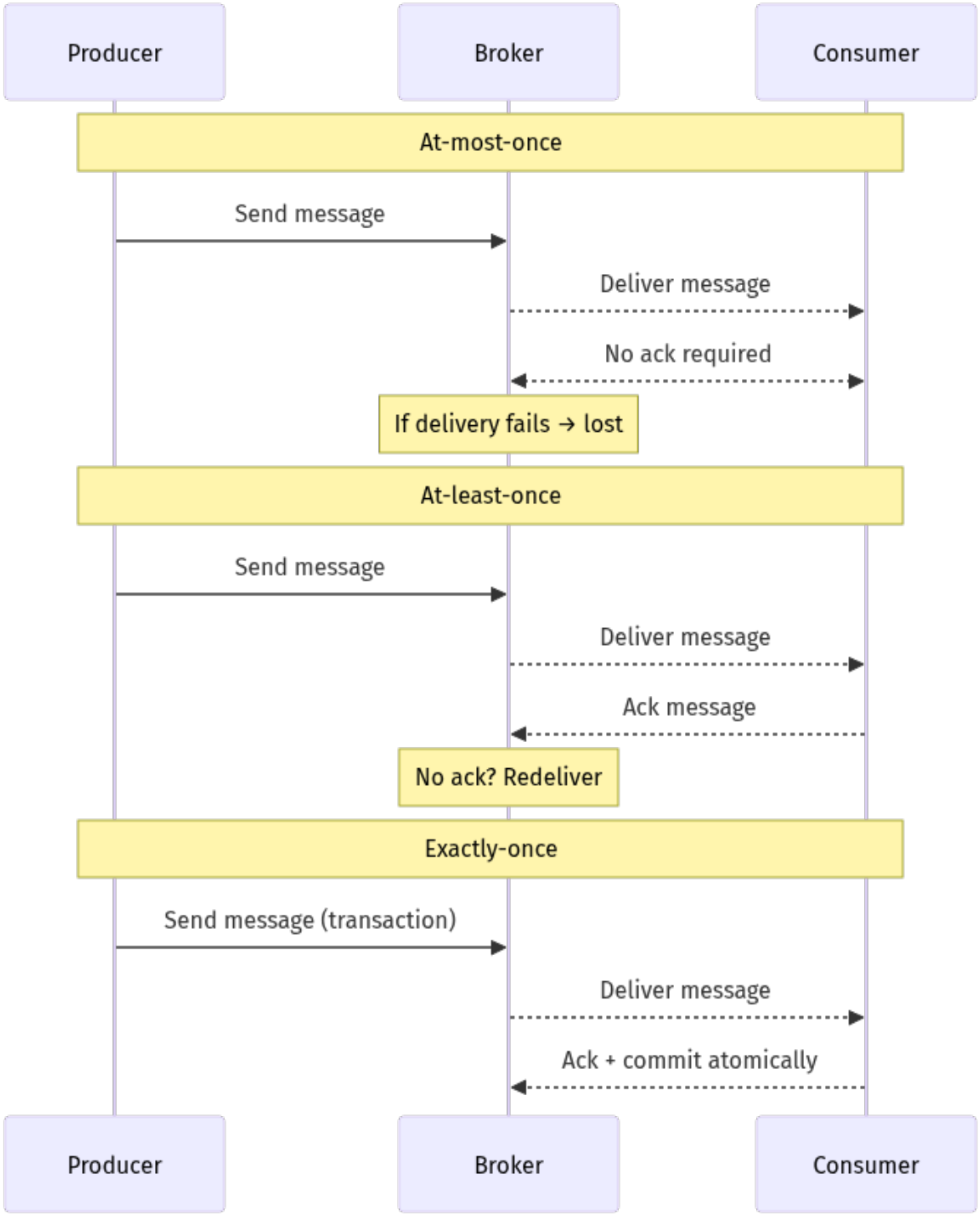


*Figure 4. Delivery Semantics*

## 1.11. Reliability & Durability

### 1.11.1. Reliability

Reliability is the ability of a messaging system to deliver messages as promised, even under faults. It depends on:

- **Acknowledgements** — explicit `ack()`/`nack()` calls.
- **Retries** — redelivery on failure.
- **Consumer groups** — work sharing without loss.
- **Dead-letter queues (DLQ)** — holding unprocessable messages for inspection.

Operational tips:

- Set sensible retry limits — infinite retries can overload systems.
- Use DLQs to capture poison messages (malformed or consistently failing).
- Monitor consumer lag — large lag indicates bottlenecks.

### 1.11.2. Durability

Durability ensures messages survive broker or system restarts. This requires **persistent storage** at the broker level.

Common strategies:

- **Disk persistence** — messages are written to disk (RabbitMQ durable queues, Kafka logs).
- **Replication** — messages are stored on multiple nodes (Kafka ISR, Pulsar BookKeeper).
- **Acknowledgement after fsync** — broker only acks after writing to stable storage.

Trade-offs:

- Durability often increases latency.
- Replication adds network cost but prevents data loss from node failure.

### 1.11.3. Durability in OmniQueue Context

OmniQueue's durability depends entirely on the broker adapter in use:

- **RabbitMQ Adapter** — durable queues + persistent messages.
- **Kafka Adapter** — log-based storage with configurable replication factor.
- **NATS JetStream Adapter** — file or memory storage with optional replication.
- **ZeroMQ Adapter** — no built-in durability (application must handle persistence).

You can influence durability in OmniQueue via `createOptions` when calling `publish()` or `subscribe()`, passing through broker-specific flags.

Example:

```
await broker.publish(
  'orders',
  { id: '1', body: {...}, headers: {} },
  { ensure: true, createOptions: { durable: true, replicationFactor: 3 } }
);
```

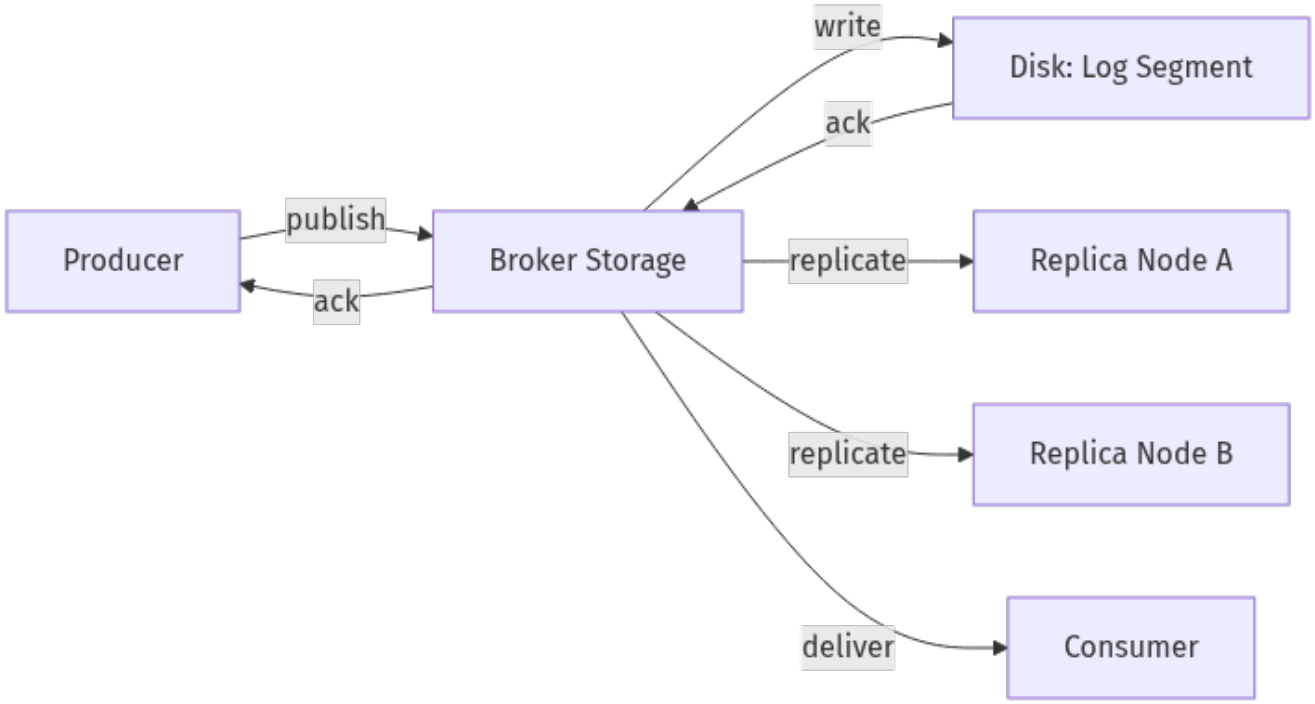


*Figure 5. Durability Lifecycle*

If you don't configure durability in the broker, OmniQueue can't "make it durable" — always set it explicitly in `createOptions` when the workload demands it.

# 1.12. Load Balancing & Consumer Groups

In messaging systems, **load balancing** is about distributing work evenly among multiple consumers to improve throughput and avoid overloading any single consumer.

## 1.12.1. Consumer Groups

A **consumer group** is a named set of consumers that share the work of processing messages from a topic or queue.

Key characteristics:

- **Work-sharing**: Each message is processed by exactly **one** consumer in the group.
- **Isolation between groups**: Multiple groups can subscribe to the same topic, each getting its **own full copy** of the messages.

- **Scaling**: Adding more consumers to a group increases parallelism; removing consumers reduces throughput but not reliability.

OmniQueue enforces **mandatory consumer groups** — every `subscribe()` call must specify a `groupId`. This makes semantics explicit and avoids accidental fanout to unintended consumers.

Example with OmniQueue:

```
await broker.subscribe(
  'orders',
  async (msg) => {
    console.log(`[Worker] Processing order ${msg.body.id}`);
    await msg.ack();
  },
  'order-service',       // groupId
  { ensure: true }
);
```

### 1.12.2. Consumer Group Dynamics

- If a consumer in a group fails mid-processing: The broker reassigns its unacknowledged messages to other consumers in the same group.
- If more consumers join the group: The broker redistributes partitions or message streams to balance the load.

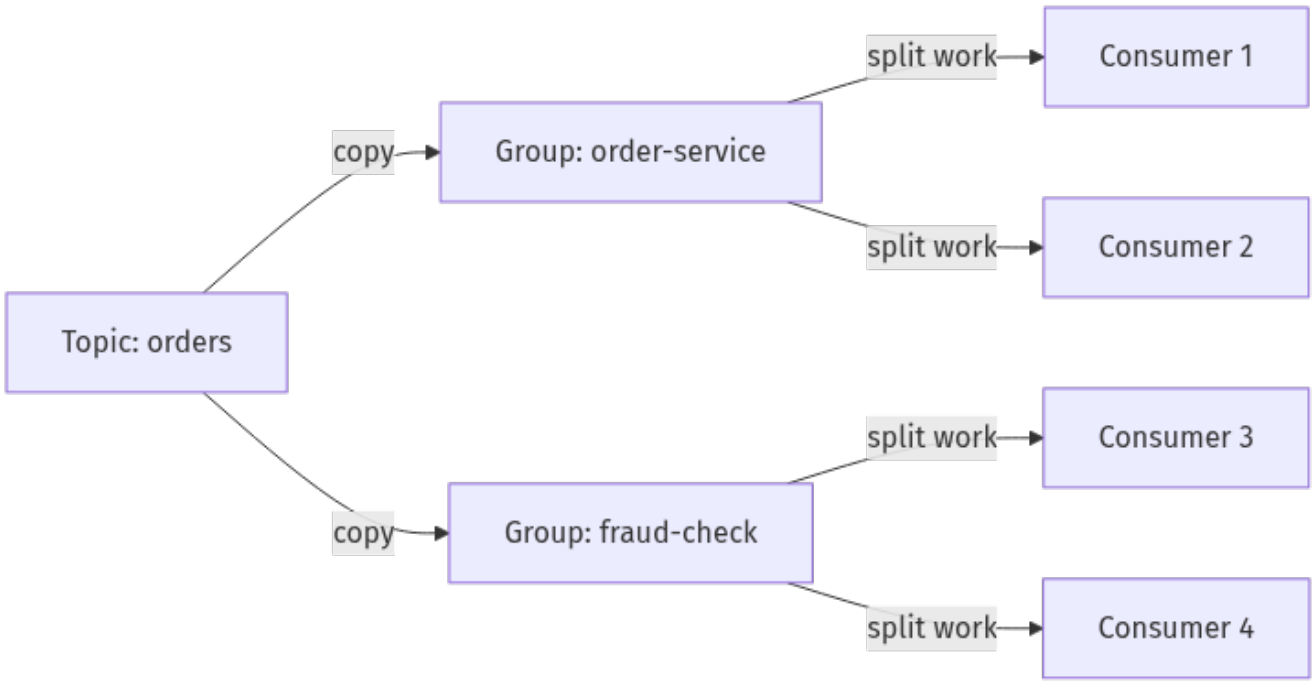


*Figure 6. Consumer Group Dynamics*

### 1.12.3. Load Balancing Patterns

1. **Static partition assignment** (Kafka, Pulsar): Each consumer gets a fixed set of partitions.

2. **Dynamic work stealing** (RabbitMQ, NATS JetStream): Consumers pull messages as they become available.
3. **Broker push with credit flow control**: Broker pushes messages but limits in-flight count per consumer.

Operational considerations:

- Monitor **consumer lag** per group.
- Keep consumer processing times balanced — slow consumers can cause uneven workloads.
- In idempotent systems, you can temporarily run the same consumer logic in multiple groups for migration or testing.

# 1.13. Pub/Sub (Fanout) vs. Point-to-Point

## 1.13.1. Point-to-Point (P2P)

In **point-to-point** messaging:

- A message is sent to a **queue**.
- Exactly one consumer within the target group processes it.
- Once acknowledged, the message is removed.

**Pros:**

- Ensures work is processed only once per group.
- Easy to reason about load balancing.

**Cons:**

- No automatic duplication to other consumers outside the group.

**Real-world usage:**

- Task queues (image processing jobs).
- Distributed workers for batch workloads.

## 1.13.2. Publish/Subscribe (Fanout)

In **publish/subscribe** messaging:

- Producers send to a **topic**.
- Each subscribed group gets a **full copy** of each message.
- Inside each group, messages are load-balanced.

**Pros:**

- Multiple services can independently react to the same event stream.
- Decouples event producers from consumers.

**Cons:**

- More storage and network usage if many groups subscribe.
- Need to manage schema and compatibility carefully.

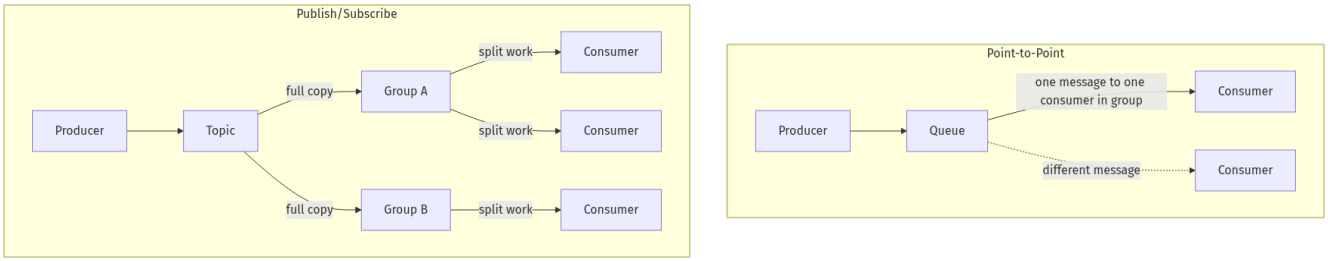


*Figure 7. Pubsub Vs P2P*

### 1.13.3. OmniQueue's Approach

OmniQueue **always** uses publish/subscribe semantics at the API level:

- `publish(topic, message, opts?)` — send to a topic.
- `subscribe(topic, handler, groupId, opts)` — join a group to consume messages.

This allows:

- **Point-to-point** — by having only one group for a topic.
- **Fanout** — by having multiple groups subscribed to the same topic.

This design unifies the two models, and the behavior depends on how you name and organize your groups.

Example:

```
// Point-to-point: single group
await broker.subscribe('orders', handler, 'order-service', { ensure: true });

// Fanout: multiple groups get all events
await broker.subscribe('orders', handler1, 'fraud-check', { ensure: true });
await broker.subscribe('orders', handler2, 'analytics', { ensure: true });
```

# Chapter 2. OmniQueue Architecture & Core API

## 2.1. Overview & Design Principles

OmniQueue is a **broker-agnostic messaging abstraction** written in TypeScript. Its core purpose is to provide a **single, unified API** for producing and consuming messages across a wide range of messaging technologies — both brokered (e.g., RabbitMQ, Kafka, Pulsar) and brokerless (e.g., ZeroMQ).

The architecture is designed to:

- **Unify, not oversimplify**: Provide consistent `publish` and `subscribe` semantics while still exposing broker-specific options via `createOptions` when needed.
- **Preserve advanced features**: Let developers use native capabilities of each broker without breaking the abstraction.
- **Enforce explicitness**: OmniQueue requires `groupId` for every subscription — avoiding accidental broadcast to unintended consumers.
- **Support both high-level productivity and low-level control**: Developers can start quickly but still configure performance, durability, and topology deeply.
- **Be modular**: Each broker is implemented as a separate adapter that can be loaded dynamically.
- **Be lifecycle-conscious**: Connection management (`init`, `close`) is explicit to encourage clean startup/shutdown patterns.

OmniQueue's **core runtime** is small and stable. It handles:

- **Broker registration** (via the plugin registry).
- **Factory creation** of broker instances.
- **Delegation** of operations (`publish`, `subscribe`) to the active adapter.

Adapters handle:

- **Protocol specifics** (AMQP, Kafka wire protocol, ZeroMQ sockets, etc.).

- **Broker-specific topology creation**.
- **Mapping of OmniQueue concepts** (topics, groups) to the broker's native constructs.

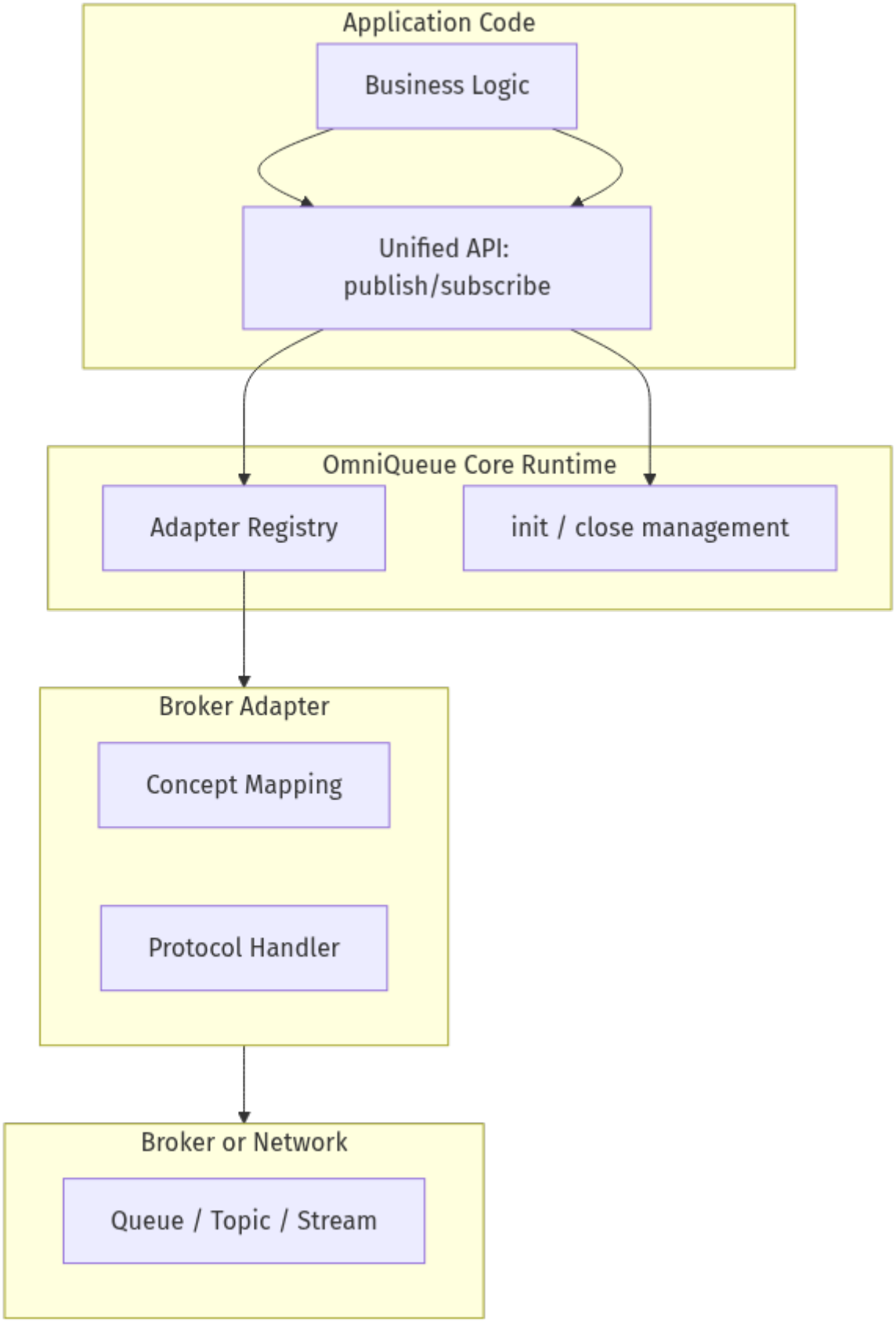


*Figure 8. OmniQueue Design Layers*

This separation means that adding support for a new broker is as simple as writing

> an adapter and registering it with `register('provider-name', factoryFn)`.

## 2.2. Core API: publish and subscribe

OmniQueue's runtime exposes two primary operations for message flow:

- **publish** — send a message to a topic so it can be consumed by one or more groups.
- **subscribe** — join a consumer group to receive messages from a topic.

This section documents their signatures, parameters, and usage patterns.

### 2.2.1. publish()

Sends a message to a given topic. All consumer groups subscribed to that topic will receive a copy. Within each group, only one consumer processes the message.

```
publish(
  topic: string,
  msg: Omit<BrokerMessage, 'ack' | 'nack'>,
  opts?: SendOptions
): Promise<void>
```

**Parameters:**

- `topic`: Name of the topic (string).
- Naming is case-sensitive for most brokers.
- `msg`: Message to send, without the `ack` and `nack` methods.
- Must include `id`, `body`, and `headers`.
- `opts`: Optional `SendOptions`:
- `prio`: Message priority (broker-dependent).
- `ensure`: Create the topic/queue/stream if it doesn't exist.
- `createOptions`: Broker-specific creation parameters (durability, partitions, etc.).

**Example:**

```
await broker.publish(
  'orders',
  {
    id: '01JABCDE123...',
    body: { orderId: 'A123', status: 'confirmed' },
    headers: { 'x-source': 'checkout-service' },
  },
```

```
  {
    ensure: true,
    createOptions: { durable: true }
  }
);
```

### 2.2.2. subscribe()

Listens for messages on a topic as part of a specific consumer group.

```
subscribe(
  topic: string,
  handler: (m: BrokerMessage) => Promise<void>,
  groupId: string,
  opts: ConsumeOptions
): Promise<void>
```

**Parameters:**

- `topic`: Name of the topic (string).
- `handler`: Async function that processes each message.
- Receives a `BrokerMessage` with `ack()` and `nack()` methods.
- Must call `ack()` on success, or `nack(requeue?: boolean)` on failure.
- `groupId`: Mandatory group identifier for work-sharing.
- `opts`: Optional `ConsumeOptions` (inherits all `SendOptions`).

**Example:**

```
await broker.subscribe(
  'orders',
  async (msg) => {
    try {
      console.log('Processing order', msg.body);
      await msg.ack();
    } catch (err) {
      await msg.nack(true); // requeue on error
    }
  },
  'order-service',
  { ensure: true }
);
```

Use multiple groups to implement **fanout**; use a single group for **point-to-point** load sharing.

## 2.3. Message Structure: `BrokerMessage`

The `BrokerMessage` interface defines the shape of every message flowing through OmniQueue. It represents a single unit of data in transit between producer and consumer.

```
export interface BrokerMessage<T = any> {
   id: string;
   body: T;
   headers: Record<string, any>;

   ack(): Promise<void>;
   nack(requeue?: boolean): Promise<void>;
}
```

*Table 4. Message fields*

| Field | Description |
|---|---|
| `id` | Unique identifier for the message. It should be globally unique to avoid collisions across brokers and environments. Best practice: ULID or UUID v7 for sortable uniqueness. |
| `body` | JSON-serialisable payload of the message. For binary data, encode as Base64 or use a broker-specific binary mode if supported. |
| `headers` | Metadata key/value pairs. Common uses: content type, correlation IDs, trace context, tenant information. |

*Table 5. Message acknowledgement methods*

| Method | Description |
|---|---|
| `ack()` | Positively acknowledges that the message was processed successfully. In at-least-once systems, this will remove the message from the queue or commit the offset. |
| `nack(requeue?: boolean)` | Negatively acknowledges the message. If `requeue` is `true`, the broker will attempt redelivery (immediate or delayed, broker-dependent). If `false`, the message may be routed to a Dead Letter Queue (DLQ) or discarded. |

### 2.3.1. Best Practices

- Always call `ack()` **exactly once** for successfully processed messages.
- Call `nack(true)` for transient errors (e.g., network failures to downstream systems).
- Call `nack(false)` for permanent errors (e.g., invalid schema) to avoid poison-message loops.
- Store `id` and relevant `headers` for idempotent processing in at-least-once systems.

*Example: Handling `BrokerMessage`*

```
await broker.subscribe(
  'payments',
  async (msg) => {
    try {
      if (!isValidPayment(msg.body)) {
        console.warn('Invalid payment', msg.id);
        await msg.nack(false); // send to DLQ
        return;
      }

      await processPayment(msg.body);
      await msg.ack();
    } catch (err) {
      console.error('Temporary error, retrying', err);
      await msg.nack(true); // requeue for retry
    }
  },
  'payment-processor',
  { ensure: true }
);
```

# 2.4. Groups and Work-sharing Semantics

In OmniQueue, **every subscription belongs to a group** — this is not optional. The `groupId` parameter in `subscribe()` is the explicit contract that defines how messages are delivered and balanced among consumers.

## 2.4.1. Why `groupId` is mandatory

- Prevents **accidental global broadcast** to all consumers.
- Makes **intent explicit** in code: you either want fanout (multiple groups) or point-to-point (single group).
- Aligns OmniQueue's semantics with modern brokers like Kafka, NATS JetStream, and Pulsar, where groups (or their equivalents) are first-class.

## 2.4.2. Work-sharing model

When a producer publishes a message to a topic:

- Each **group** subscribed to that topic receives a full copy of the message stream.
- Within each group, **only one consumer** handles a given message.
- The broker decides which consumer in the group gets the next message, based on its load-

balancing strategy.

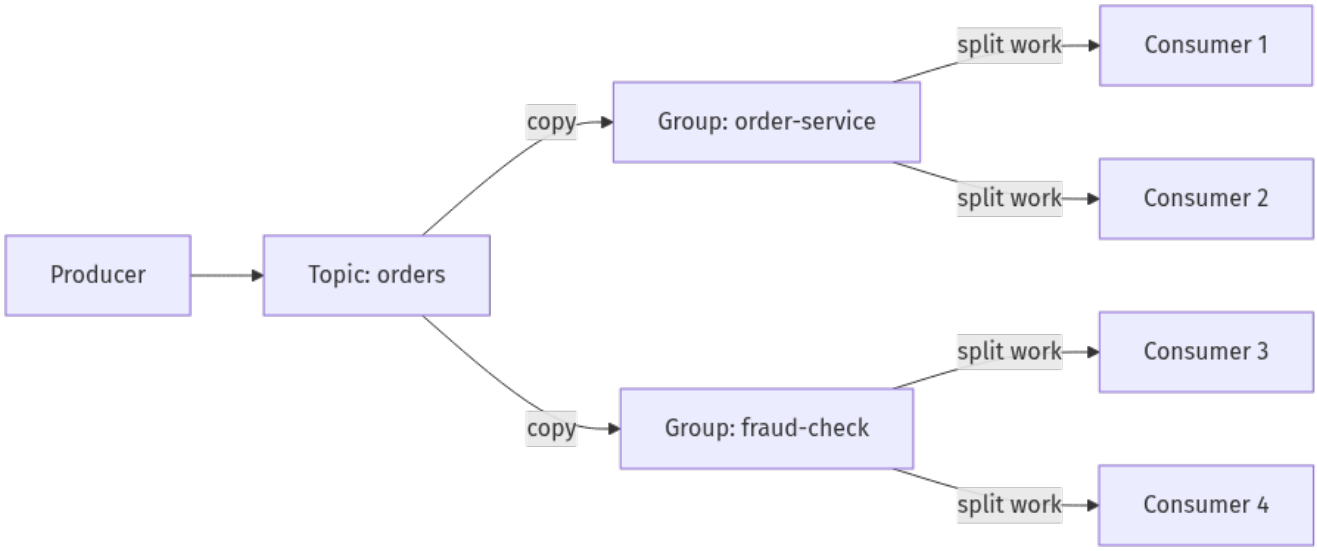


*Figure 9. Consumer group work-sharing*

### 2.4.3. Patterns enabled by `groupId`

- **Point-to-point load sharing**: Use a single group for all consumers that should share the workload.
- **Fanout processing**: Use multiple groups, each representing an independent processing path.

```
await broker.subscribe('orders', handleOrder, 'order-service', { ensure: true });
await broker.subscribe('orders', handleFraud, 'fraud-check', { ensure: true });
await broker.subscribe('orders', handleAnalytics, 'analytics', { ensure: true });
```

### 2.4.4. Operational considerations

- **Scaling throughput**: Add more consumers to the same group to process messages faster.
- **Zero-downtime deployment**: Temporarily run old and new versions of a consumer in the same group to drain messages during rollout.
- **Isolated experiments**: Spin up a separate group for A/B testing without impacting production consumers.
- **Back-pressure management**: Monitor group-level consumer lag to detect bottlenecks.

In some brokers, group identifiers are durable — the broker remembers offsets per group even if no consumers are active. This allows resuming consumption from the last processed message.

## 2.5. Lifecycle Management (`init`, `close`)

OmniQueue brokers follow a **well-defined lifecycle** to ensure connections are properly established, resources are managed, and workloads are gracefully shut down.

### 2.5.1. Initialization (`init()`)

Before you can send or receive messages, you **must** initialize the broker instance.

- Establishes network connections to the broker or peer nodes.
- Performs authentication/authorization (if required).
- Prepares client internals such as connection pools, channel/session objects, and background heartbeat tasks.
- May lazily create broker-side resources (queues, topics, streams) depending on the adapter.

```
import { create } from 'omni-queue-core';

// Create broker instance from provider
const broker = await create('rabbitmq', {
  host: 'localhost',
  port: 5672,
  username: 'guest',
  password: 'guest',
});

// Must be called before publish/subscribe
await broker.init();
```

In production, call `init()` **during service startup** and ensure it completes before accepting external requests. Failing to initialize may cause publish or subscribe calls to throw errors.

### 2.5.2. Graceful shutdown (`close()`)

When your service is terminating, **always close the broker connection**:

- Ensures in-flight messages are acknowledged before disconnection.
- Releases network sockets, file handles, and background workers.
- Helps the broker redistribute work quickly to other consumers in the same group.

```
process.on('SIGTERM', async () => {
  console.log('Shutting down…');
  await broker.close();
```

```
  process.exit(0);
});
```

Pair `init()` and `close()` in a lifecycle manager, such as:

- Node.js process hooks (`SIGTERM`, `SIGINT`).
- HTTP server start/stop callbacks.
- Framework lifecycle hooks (NestJS `OnModuleInit` / `OnModuleDestroy`).

### 2.5.3. Common pitfalls

- **Skipping `close()`**: Can lead to stale connections or message redelivery delays.
- **Closing too early**: If you close before acknowledging all messages, some brokers will redeliver them.
- **Multiple `init()` calls**: Avoid re-initializing the same broker instance; instead, reuse the connection.

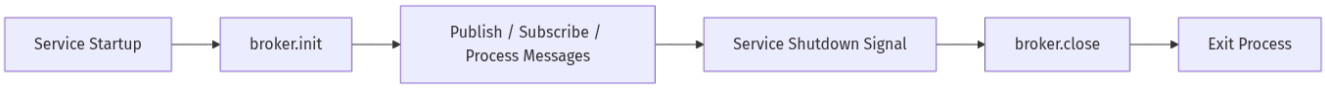


*Figure 10. Lifecycle overview*

## 2.6. Error Handling (`ack`, `nack`)

OmniQueue enforces explicit acknowledgement semantics for all consumed messages. This makes error handling **predictable** and **adapter-agnostic**, regardless of broker type.

### 2.6.1. Acknowledgement (`ack()`)

Calling `ack()` tells the broker:

- The message has been processed successfully.
- It can be removed from the queue or marked as complete in the log.

```
await broker.subscribe('orders', async (msg) => {
  try {
    await processOrder(msg.body);
    await msg.ack();
  } catch (err) {
    console.error('Processing failed', err);
    await msg.nack(true); // retry
  }
```

```
}, 'order-service', { ensure: true });
```

Always `ack()` **after** completing the business logic, not before. Acknowledging too early risks losing messages if the process crashes mid-task.

### 2.6.2. Negative acknowledgement (`nack(requeue?: boolean)`)

Calling `nack()` signals:

- The message **could not** be processed.
- The broker may requeue the message for retry (if `requeue` is `true`) or discard/send to DLQ (if `false`).

**Usage patterns:**

- `nack(true)` → retry later (transient failure).
- `nack(false)` → drop or route to DLQ (poison message).

### 2.6.3. Idempotency for at-least-once delivery

Since most brokers operate with **at-least-once** guarantees, your consumer logic must be **idempotent**:

- Ensure repeating the same message does not cause incorrect results.
- Common strategies:
- Use a **message ID store** (database, Redis) to track processed IDs.
- Apply **upserts** instead of inserts.
- Wrap business logic in transactional boundaries.

### 2.6.4. Retry & DLQ patterns

**Retry loop:** Many brokers allow message redelivery after a delay, often via dead-lettering with a TTL and re-queue policy.

**DLQ (Dead Letter Queue):** Capture messages that consistently fail to process. This enables:

- Debugging message payloads.
- Backfilling after code fixes.
- Alerting when thresholds are reached.

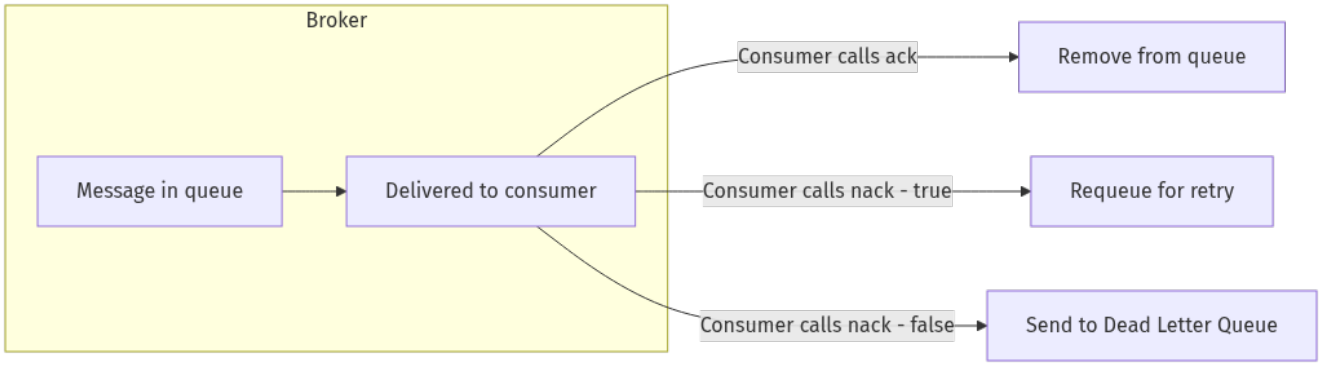


*Figure 11. ack/nack lifecycle*

### 2.6.5. OmniQueue adapter mapping

Different brokers use different APIs, but OmniQueue normalizes them:

- **RabbitMQ** → `channel.ack()` / `channel.nack()`
- **Kafka** → commit offset / seek & reprocess
- **NATS JetStream** → `msg.ack()` / `msg.nak({ delay })`
- **ZeroMQ** → application-level ack handling

You **always** use `msg.ack()` and `msg.nack()` in OmniQueue, regardless of broker.

When integrating with multiple brokers, always assume **at-least-once** and make consumers idempotent. This ensures consistent behavior even if you switch adapters.

## 2.7. OmniQueue Core Flow Diagram

The OmniQueue core flow represents the **end-to-end lifecycle** of a message — from creation to final acknowledgement — in a **broker-agnostic** manner. This view is essential for both developers and operators to understand how OmniQueue mediates between application code and underlying messaging infrastructure.

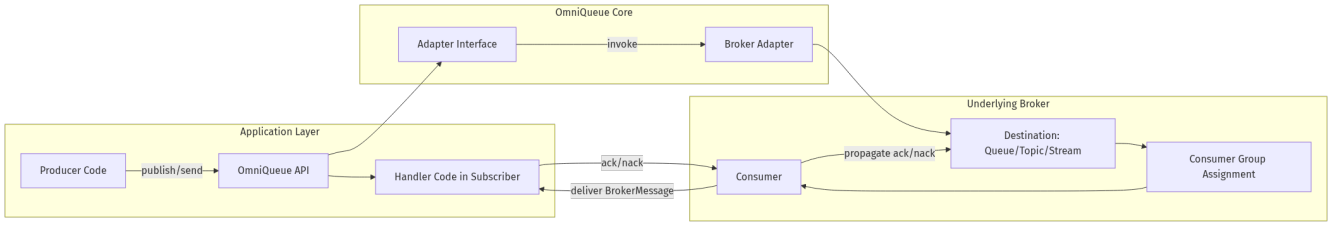


*Figure 12. Core Flow Overview*

### 2.7.1. How to Read This Diagram

- **Application Layer** — Your service's producer and consumer code.

- **OmniQueue Core** — The broker-agnostic interface (`publish`, `subscribe`) and internal adapter interface.
- **Broker Adapter** — Provider-specific implementation (RabbitMQ, Kafka, NATS, etc.).
- **Broker** — The actual messaging infrastructure.
- **Destination** — The queue, topic, or stream created or ensured.
- **Consumer Group Assignment** — The broker's work-sharing or partition assignment process.

### 2.7.2. Flow Breakdown

1. **Producer Code**
   - Calls `publish()` (topic-based API) or `send()` (if adapter supports queue semantics) with a `BrokerMessage` payload.
   - Can set `ensure` and `createOptions` for destination management.
2. **OmniQueue API**
   - Validates parameters.
   - Wraps the payload in OmniQueue's standard format.
   - Passes request to the registered adapter via the adapter interface.
3. **Broker Adapter**
   - Maps OmniQueue's API calls to the broker's native client library calls.
   - Ensures destination exists if `ensure` is `true`.
   - Serialises the message body and headers into broker-specific format.
4. **Destination Creation/Lookup**
   - Broker creates or verifies the target queue/topic/stream.
   - Applies `createOptions` (e.g., partitions, durability, replication).
5. **Delivery to Consumer Group**
   - Broker assigns messages to a consumer in the target group.
   - In **point-to-point**, exactly one consumer in the group processes each message.
   - In **fanout**, all groups subscribed receive the message.
6. **Handler Execution**
   - Message arrives at the subscriber's handler.
   - Handler processes and calls `ack()` (success) or `nack()` (failure).
7. **Ack/Nack Propagation**
   - `ack()` removes or commits the message in the broker.
   - `nack()` either discards or requeues the message (depending on `requeue`).

### 2.7.3. Operational Notes

- **Backpressure Management** — Most adapters expose prefetch or flow control to prevent overwhelming consumers.
- **Tracing** — Add correlation IDs in `headers` for distributed tracing.
- **Error Containment** — Use dead-letter queues to capture repeatedly failed messages.

Understanding this flow helps in debugging end-to-end issues, such as “messages not being delivered” or “duplicates appearing” — often the cause lies at a specific hand-off in this chain.