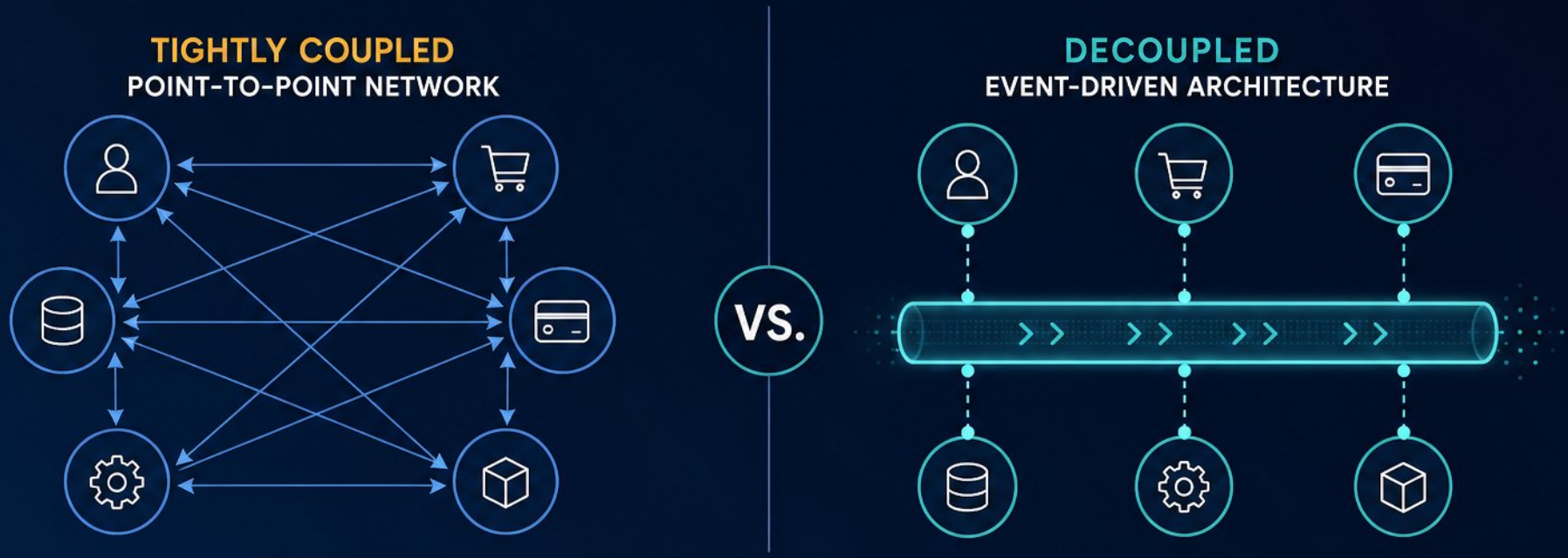
Introduction to Event-Driven Systems:

Events, Producers, and Consumers

- 72% of global organizations now use event-driven architecture, but only 13% reach mature implementation
- Shift from request-response: components collaborate around change, not polling or direct calls
- Real-world analogies: restaurant order flow, news subscriptions, smart home sensors
- Learn events as facts, producers as sources, consumers as reactors, and the decoupling that connects them
- Agenda: core concepts, patterns, anti-patterns, and a complete example flow

Why Event-Driven Thinking?

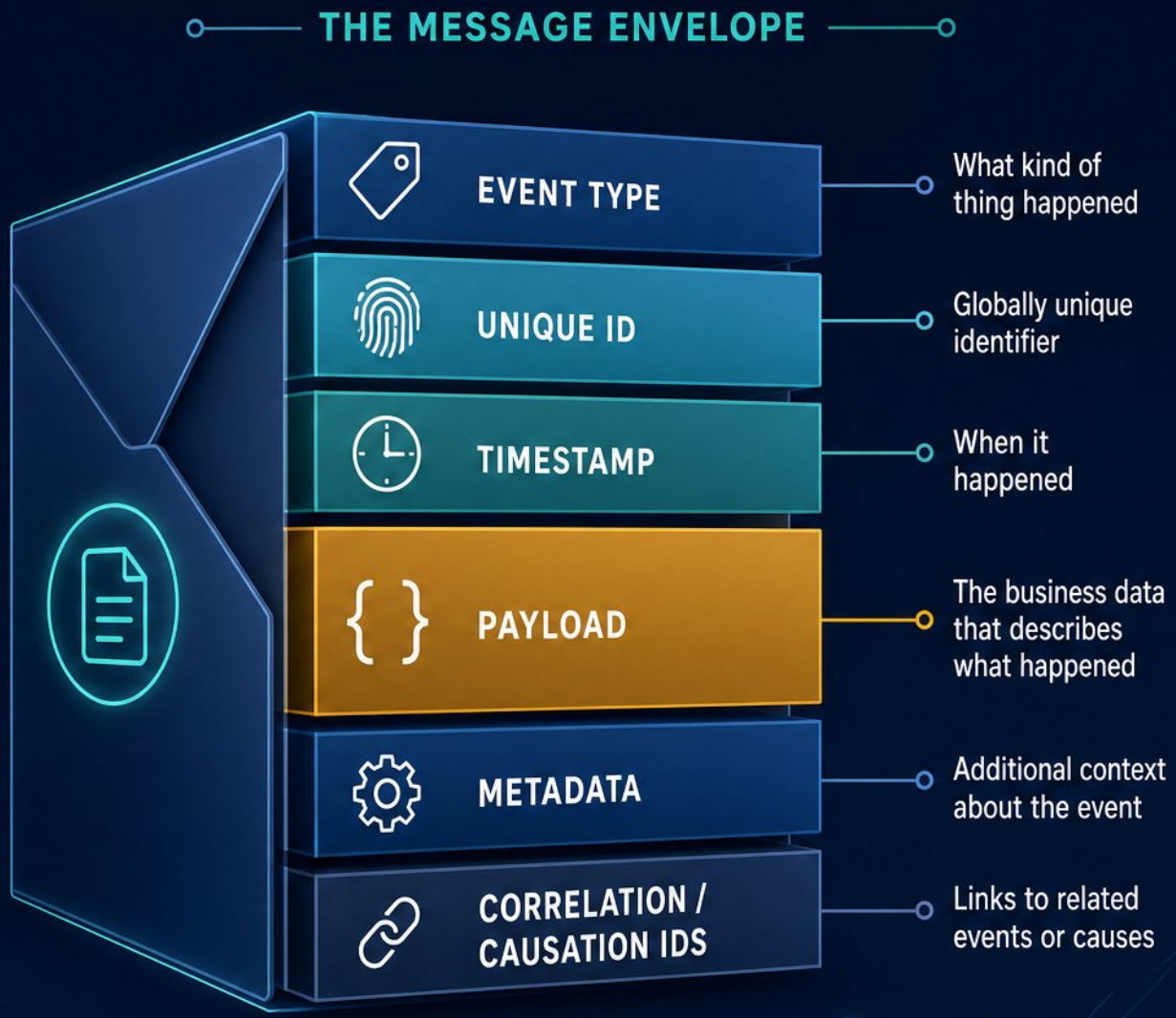
The Motivation and Mindset



- Solves tight coupling, cascading failures, and scaling bottlenecks
- 63% of orgs report better scalability; 52% see fewer incidents (2026 data)
- Enables continuous real-time processing instead of batch or polling
- Core mindset: design around facts that happen, not synchronous requests
- Choose EDA for independent reactions to change—not as a universal default

What Is an Event? Facts, Structure, and Semantics

- An event is an immutable record of something that happened—a fact, not a request
- Structure: event type, unique ID, timestamp, payload, metadata, correlation/causation IDs
- Event vs. command vs. message: events say "this happened," commands say "do this"
- Immutable and append-only: corrections come as new events, never overwrites
- Domain events, system events, and notification events serve different scopes

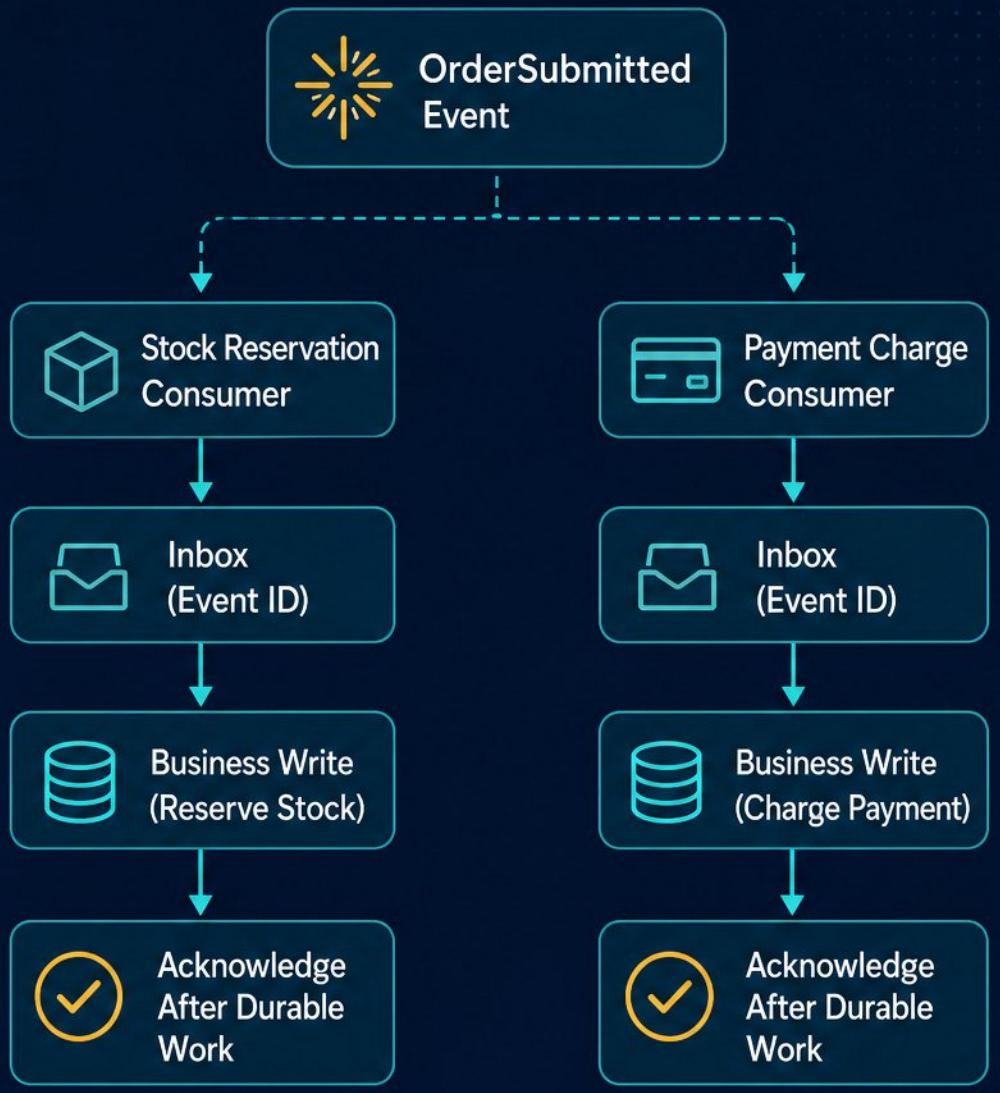


Producers:
Where Events Come From

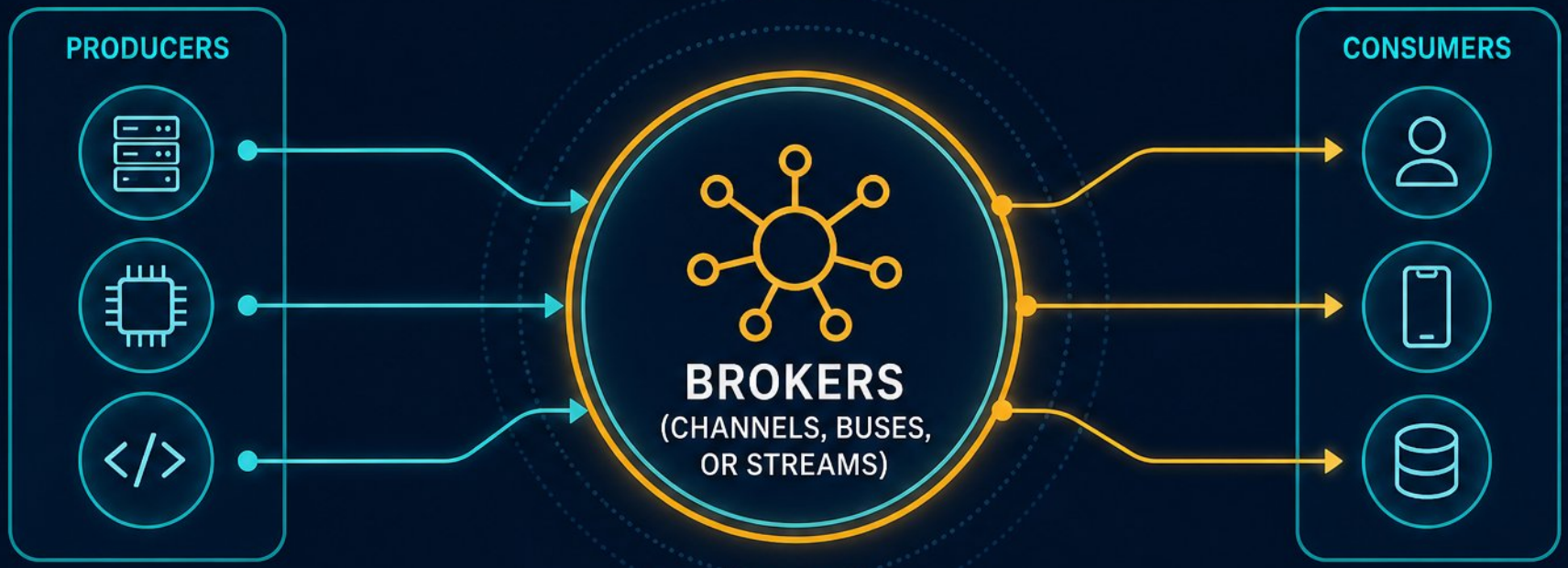
- A producer detects a state change and publishes a fact — services, frontends, IoT, or database CDC
- The dual-write problem: database and broker are two operations that can drift apart
- Transactional outbox: write events to a DB table in the same transaction, then relay to the broker
- Idempotent publishing: ensure the same logical event is never published twice
- Producers don't know who consumes — one-way dependency is the key to loose coupling

Consumers: Reacting to Events

- Consumers subscribe to events and perform reactions: notifications, read models, workflow triggers
- At-least-once delivery is the default; duplicates always happen, so make every handler idempotent
- Use the inbox pattern: record event ID in the same DB transaction as the business write
- Push vs. pull consumption models trade off latency, throughput, and backpressure control
- Acknowledgment occurs only after durable work—never before, or you risk losing events



Connecting Producers and Consumers: Channels, Brokers, and Streams



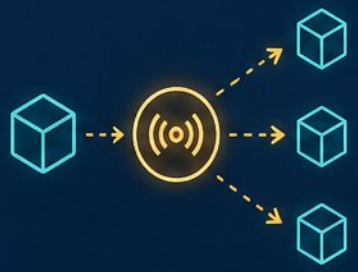
- - Events travel via **channels, buses, or streams**—the intermediary that enables decoupling
- - **Kafka:** log-based streaming for high throughput, message replay, and event sourcing
- - **RabbitMQ:** flexible broker for complex routing, task queues, and low-latency delivery
- - **NATS:** lightweight pub/sub for microservices, edge, and IoT with simple operations
- - Brokers enable temporal and spatial decoupling—producers and consumers don't need to sync

Core Benefits: Why Organizations Adopt Event-Driven Architecture

- **Loose coupling:** add consumers without touching producers; 80% faster integration onboarding (W1-S1)
- **Scalability & resilience:** isolated failures; Kafka handles millions of events/sec; 99.999% uptime proven (W3-S2)
- **Real-time responsiveness:** 80ms p90 latency replacing 8-15 second polling; 100x speed improvement (W6-S3)
- **Auditability:** immutable event log as natural audit trail, enabling replay and compliance (W6-S3, W9-S8)
- **Trade-off awareness:** eventual consistency, operational complexity, schema evolution require discipline (W9-S4)

Common Patterns in Event-Driven Design

1 Event Notification



- Event Notification: fire-and-forget—producers broadcast facts, consumers react independently

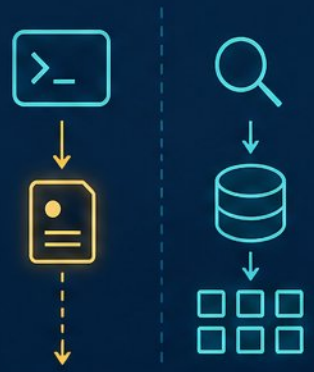
2 Event-Carried State Transfer

- Event-Carried State Transfer: events include enough data so consumers don't need callbacks

3 Event Sourcing

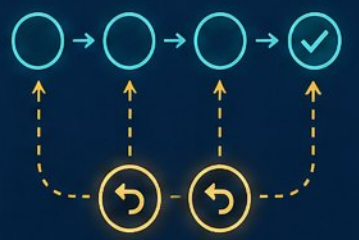
- Event Sourcing: store all changes as immutable events; rebuild state by replaying the log

4 CQRS



- CQRS: separate read and write models—writes via commands/events, reads use optimized projections

5 Saga Pattern



- Saga Pattern: coordinate distributed transactions through local steps with compensating actions for rollback

THE APPEND-ONLY LOG

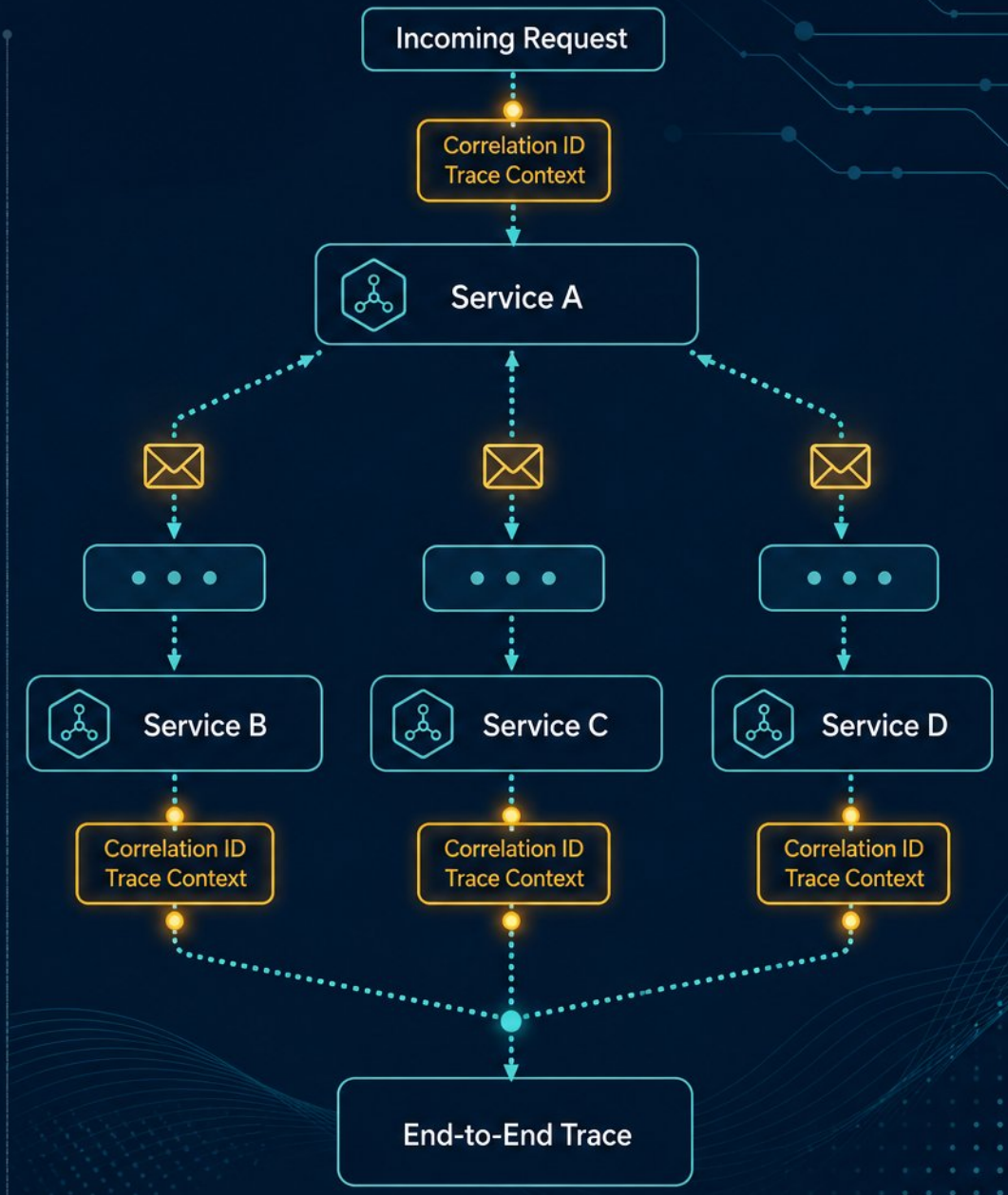
Anti-Patterns:

What to Avoid from Day One

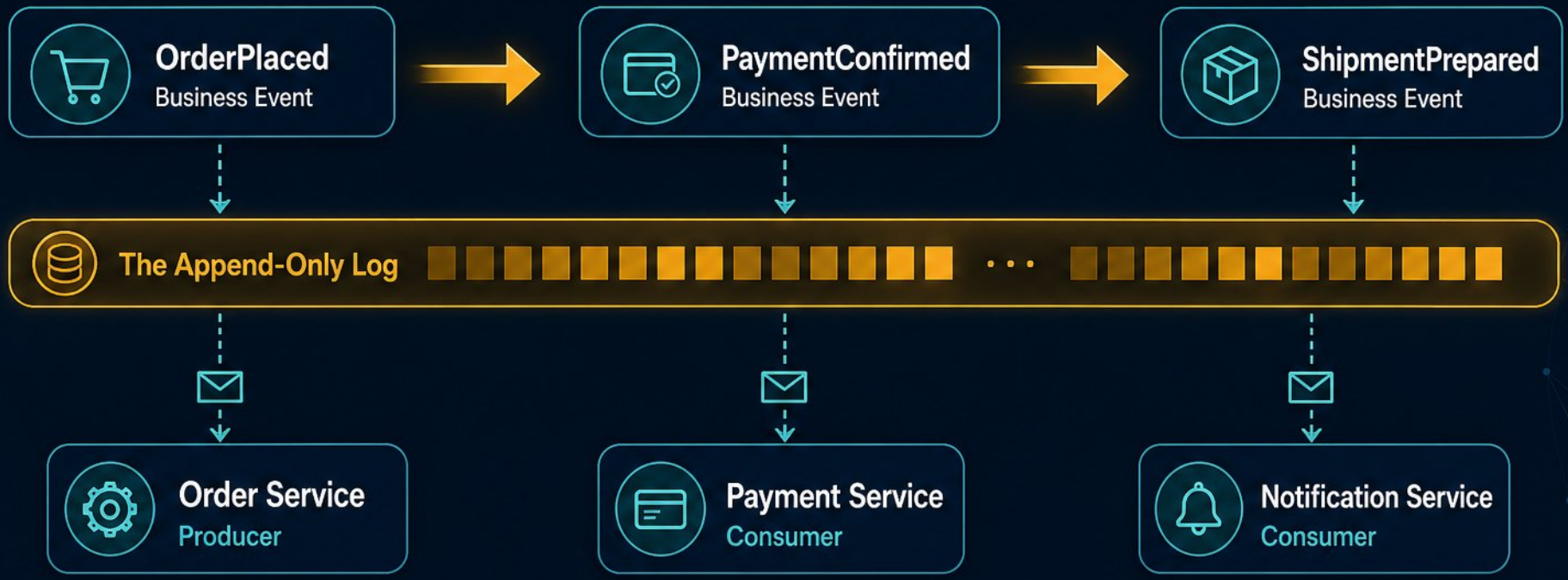
- Using events as commands: if the producer expects a specific outcome, it's a command
- God events and event soup: monolithic payloads create hidden, unmanageable coupling
- Event chains and domino effects: uncontrolled cascading reactions without explicit process managers
- Assuming exactly-once delivery: at-least-once is the reality; idempotency is mandatory for every consumer
- Ignoring dead-letter queues: silent failures pile up—DLQs need monitoring, alerting, and replay tools from day one

Key Challenges: Ordering, Consistency, and Observability

- Order is per-partition, not global—key events by entity ID (e.g., order_id) for correct per-entity sequence
- At-least-once delivery is reality; every consumer must be idempotent using event IDs and inbox tables
- Design for eventual consistency: consumers may see slightly stale data and must tolerate that lag
- Propagate correlation IDs and W3C trace context through message headers for end-to-end distributed tracing
- Treat event schemas as API contracts—use backward-compatible changes, schema registries, and explicit versioning

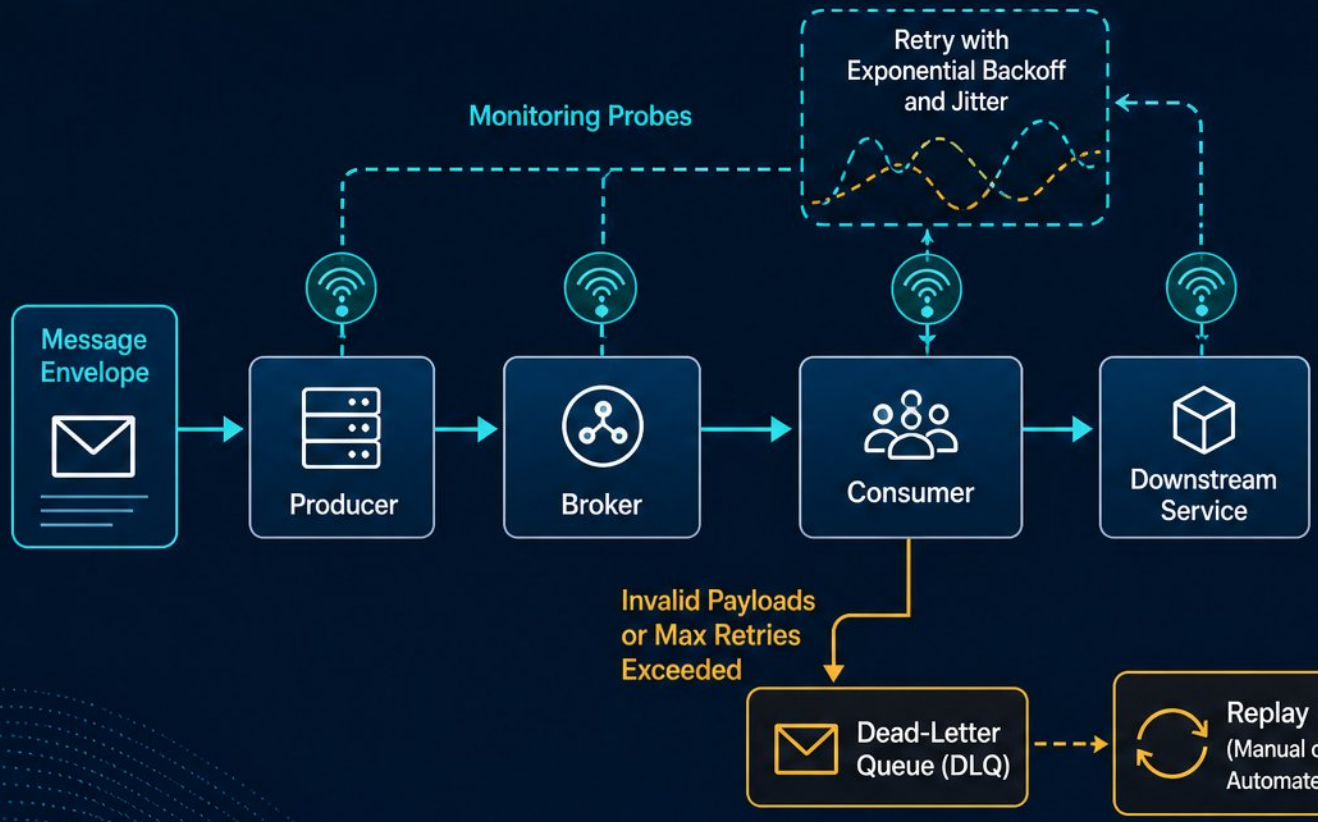


Your First Event-Driven Flow: A Step-by-Step Example



- Scenario: simplified order management for an e-commerce system
- Step 1: Identify business events — OrderPlaced, PaymentConfirmed, ShipmentPrepared
- Step 2: Define schemas, choose a broker (start with lightweight options)
- Step 3: Build producer (order service) and consumers (payment, notification)
- Step 4: Trace OrderPlaced end-to-end — no direct service-to-service calls
- Start small: in-process event bus first, graduate to managed brokers when needed

Operational Readiness: Dead Letters, Retries, and Monitoring



- Dead-letter queues (DLQs) need alerting and a replay plan—monitor depth and age
- Use exponential backoff with jitter for transient failures; no retry for invalid payloads
- Consumer lag is a leading signal—growing lag indicates processing bottlenecks or bugs
- Propagate correlation IDs and W3C trace context through every event header and log line
- Non-negotiables before production: idempotent consumers, DLQ with alerts, versioned schemas, correlation IDs

Wrap-Up, Key Takeaways, and Next Steps

- Four pillars: events as facts, producers as sources, consumers as reactors, channels as connectors
- Core shift: design around change, not synchronous requests — publish facts, react independently
- Start with logical boundaries, then choose communication patterns, then deploy (don't let infrastructure drive design)
- Next: event storming workshops, hands-on with a lightweight broker (RabbitMQ/NATS), then event sourcing and CQRS
- Resources: Confluent EDA guide, Azure Architecture Center, community forums, and tools like Encore

PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/0b518f7c/public