Event-Driven Architectures demystified: from Producer to Consumer

30 July 2025 - 8 min. read

Event-Driven Architecture (EDA)

Introduction

This is the first of two articles dedicated to exploring a topic that has recently gained significant attention: Event-Driven and Event-Based Architectures (EDAs).

While there's plenty of discussion surrounding these concepts, increased chatter doesn't necessarily lead to increased clarity. The goal of these blog posts is to cut through the noise and provide a clear, accessible explanation of the fundamental principles behind these architectural paradigms.

At the heart of EDAs lie two primary roles: the producer and the consumer. The producer is responsible for generating and dispatching a message. On the other end, the consumer receives this message and reacts to it, typically by executing a specific business logic or triggering a workflow. This interaction forms the backbone of how information flows in EDAs.

The event semantics

The object transmitted from the producer to the consumer is intentionally referred to using a broad and generic term: **message**. This deliberate generalization allows for flexibility, as a message can represent different types of communication depending on the producer's intent. Most commonly, messages fall into two categories: **commands** and **events**.

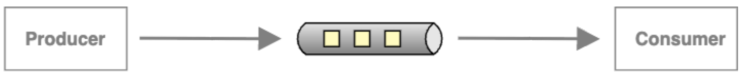
A **command message** is used when the producer intends to request a specific action from the consumer—essentially instructing it to perform a task. Commands are typically imperative and expect a certain outcome or side effect. In contrast, an **event message** serves as a notification that something has already happened—often representing a change in state or the occurrence of a significant business event. Unlike commands, events do not expect any direct response or action from the consumer; the consumer simply reacts if and how it chooses.

This distinction between commands and events is fundamental in understanding how components communicate in EDAs. It also shapes how systems are designed.

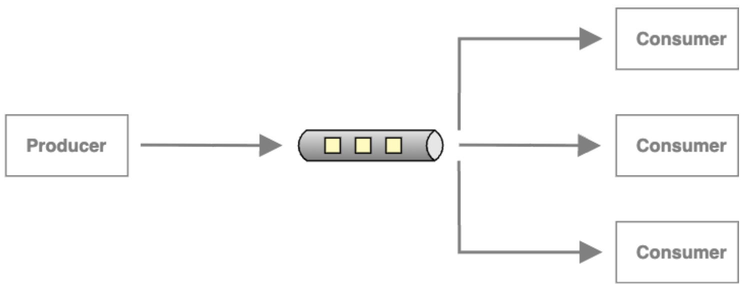
Message channels

A **producer** sends messages through **message channels**, which act as highways to reach consumers. As Gregor Hohpe aptly notes in one of his writings, Event-Driven Architectures are commonly associated with **Publish-Subscribe Channels**, since multiple recipients may be interested in reacting to a single event. This is in contrast to **Point-to-Point Channels**, which are typically used for delivering commands or document-style messages to a single, specific consumer.

In the AWS ecosystem, these two types of channels are well represented: **SQS Queues** exemplify Point-to-Point Channels, while **SNS Topics** are a clear implementation of Publish-Subscribe Channels. With **SQS Queues**, each message is directed to a single consumer—usually a worker or application polling the queue—ensuring that it is processed once and only once. In contrast, **SNS Topics** allow a single message to be delivered to multiple consumers, known as **subscribers**, who have registered their interest in that topic. This model enables decoupled communication where events can propagate to several systems simultaneously, supporting scalability and extensibility.



Point-to-Point Message Channel



Publish-Subscribe Message Channel

Gregor Hohpe's Enterprise Integration Patterns provides canonical definitions for both patterns, and these ideas are foundational for modern EDAs.

The Dimensions of Coupling

When we connect a producer to a consumer via a message channel, we introduce a certain type of coupling. One of the core promises of EDAs is to maximize the independent variability of producers and consumers—achieving what's commonly referred to as loose coupling. This means components (in particular, consumers) should be able to change with little to no impact on the rest of the system.

But coupling isn't binary; systems aren't simply "coupled" or "decoupled." In practice, coupling spans multiple dimensions, each affecting a different aspect of system behavior. Let's explore the five key dimensions of coupling—and how various message channel types support or hinder flexibility across them.

Temporal Coupling

What it is: Time-based dependency between producer and consumer.

Typical consumer changes: Becomes temporarily unavailable, slows down due to latency or load.

In tightly coupled, synchronous systems, a producer must wait for the consumer to respond. This creates fragility—if the consumer is unavailable, the producer is stuck.

Message channel support:

Both point-to-point (e.g., SQS) and pub/sub (e.g., SNS) channels decouple components in time through asynchronous messaging.

The producer emits a message and moves on. Consumers pull or receive messages when ready.

This decoupling improves fault tolerance and resilience, as producers don't depend on the consumer's immediate availability.

Location Coupling

What it is: The producer's awareness of the consumer's network location or address.

Typical consumer changes: Moves to a new host, region, or is scaled horizontally.

In traditional RPC-based systems, producers must know exactly where consumers live. Any network change may require reconfiguration.

Message channel support:

Message brokers (e.g., SQS, SNS, EventBridge) remove the need for direct addressing.

The producer sends to a channel or bus, not to a known endpoint.

This allows consumers to move or scale out without impacting the producer—enabling location transparency and deployment flexibility.

Space Coupling

What it is: Assumes shared infrastructure or physical proximity.

Typical consumer changes: Moves to a different VPC, region, or is separated via intermediary layers (e.g., proxies or gateways).

Space coupling is about assumptions around network topology. If producers and consumers are tightly coupled in space, they must live close—physically or virtually.

Message channel support:

Both point-to-point and pub/sub systems support space decoupling.

Intermediaries like brokers and event buses facilitate cross-network or multi-region communication.

This enables architectural flexibility, especially in distributed systems, hybrid cloud, or multi-account setups.

Topology Coupling

What it is: The ease (or difficulty) of adding new consumers.

Typical consumer changes: A new consumer application needs to receive the same events.

This is where the distinction between channel types becomes critical.

In point-to-point channels like SQS, only one consumer can successfully receive each message. Adding a second consumer to the same queue is an anti-pattern: it creates race conditions, where only one application gets the message, while the other misses it.

In contrast, publish-subscribe channels like SNS are designed to support multiple consumers. Each subscriber receives a full copy of the message, independently.

Message channel implications:

- **Point-to-point:** Not suitable for dynamic or growing topologies.
- **Pub/Sub:** Ideal for extensible architectures where new consumers can be added without modifying the producer.

While these traditional patterns address many use cases, they can be limiting when you want the flexibility to deliver a message to a specific set of consumers—without changing your producer or duplicating infrastructure.

Keep that in mind—we're coming back to it!

Format and Semantic Coupling

What it is: Tight binding between producer and consumer at the data level.

Typical consumer changes: Needs to evolve the schema, rename a field, or redefine a value.

Even if systems are decoupled in time and topology, they may be coupled by the structure or meaning of messages.

- **Format coupling**: The consumer depends on exact field names or types.
- **Semantic coupling**: The consumer depends on the meaning of a field, which may change subtly over time.

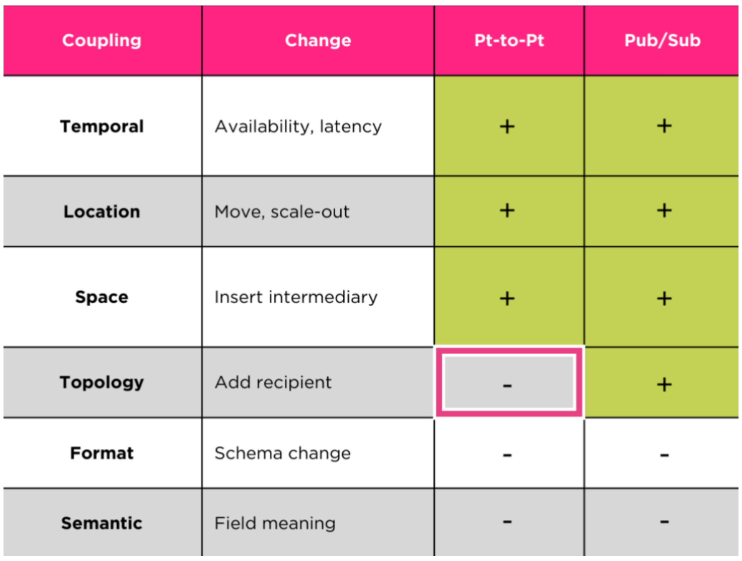
Message channel implications

Most channels—whether SQS, SNS, or EventBridge—don’t enforce structure. It's up to you to manage this coupling explicitly.

Tools like JSON Schema, Avro, or Protocol Buffers, and techniques like schema versioning and backward-compatible changes, are key to minimizing risk.

Coupling dimensions recap

Each dimension of coupling reflects a different kind of dependency between components. Message channels like SQS and SNS help decouple producers and consumers across several of these dimensions—but they aren't perfect for all use cases.



Coupling	Change	Pt-to-Pt	Pub/Sub
Temporal	Availability, latency	+	+
Location	Move, scale-out	+	+
Space	Insert intermediary	+	+
Topology	Add recipient	-	+
Format	Schema change	-	-
Semantic	Field meaning	-	-

Where point-to-point channels fall short—especially in topology flexibility—tools like **Amazon EventBridge** offer a more dynamic and scalable model. EventBridge is a serverless event bus that allows you to route events to one or more consumers based on flexible, content-based rules. Unlike traditional point-to-point messaging—where adding a new consumer can introduce race conditions or require architectural changes—EventBridge decouples routing logic from the producer. This means you can direct the same event to multiple consumers, or selectively to just one, without modifying upstream code or duplicating messages.

By supporting both publish-subscribe and point-to-point communication patterns from a single event stream, EventBridge fills the gap left by conventional messaging tools. With built-in event filtering, targeted delivery, and many-to-many routing, it enables loosely coupled systems that are far easier to evolve and extend.

In the next part of this series, we'll move from theory to practice—diving into how to apply these concepts using EventBridge to build resilient, event-aware systems with clarity and control.

Event-Driven or Event-Based?

By now, you've likely got a solid high-level sense of what it means for a system to work with events, which will help as we move into the practical side of things. In today's software landscape, the terms **event-driven** and **event-based** are often used interchangeably, but they actually refer to different perspectives. At a high level, an **event** is simply a notification that something has happened—an immutable fact that cannot be undone.

The key distinction lies in the type of events systems respond to. **Event-based** systems primarily deal with *technical events*, such as a file being uploaded or a timer firing. On the other hand, **event-driven** systems focus on *business events*—significant occurrences within the domain, like a new customer registration or an order being placed. This difference shapes how we design, build, and communicate about our systems, helping ensure that software not only reacts to change but does so in a way that aligns with business goals.

Wrapping Up

In this first part of our journey into EDAs, we've laid a solid foundation by unpacking the fundamental concepts: from understanding the dual nature of messages as commands or events, to examining the critical role message channels play in shaping communication patterns, and diving into the multiple dimensions of coupling that influence how loosely or tightly systems are connected—often more subtly than we realize.

By gaining clarity on these principles, you're better equipped to appreciate the true essence of event-based and event-driven systems beyond the buzzwords. These concepts form the building blocks for designing flexible, scalable, and resilient architectures that can evolve alongside business needs.

This is just the beginning.

In the next article, we'll get our hands dirty and move from principles to practice—exploring how these concepts come to life using real-world tools like Amazon EventBridge, and how to route, filter, and transform events with confidence.

Stay tuned!

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Eric Villa

Solutions Architect @beSharp | AWS Community Builder | AWS Authorized Instructor

---