

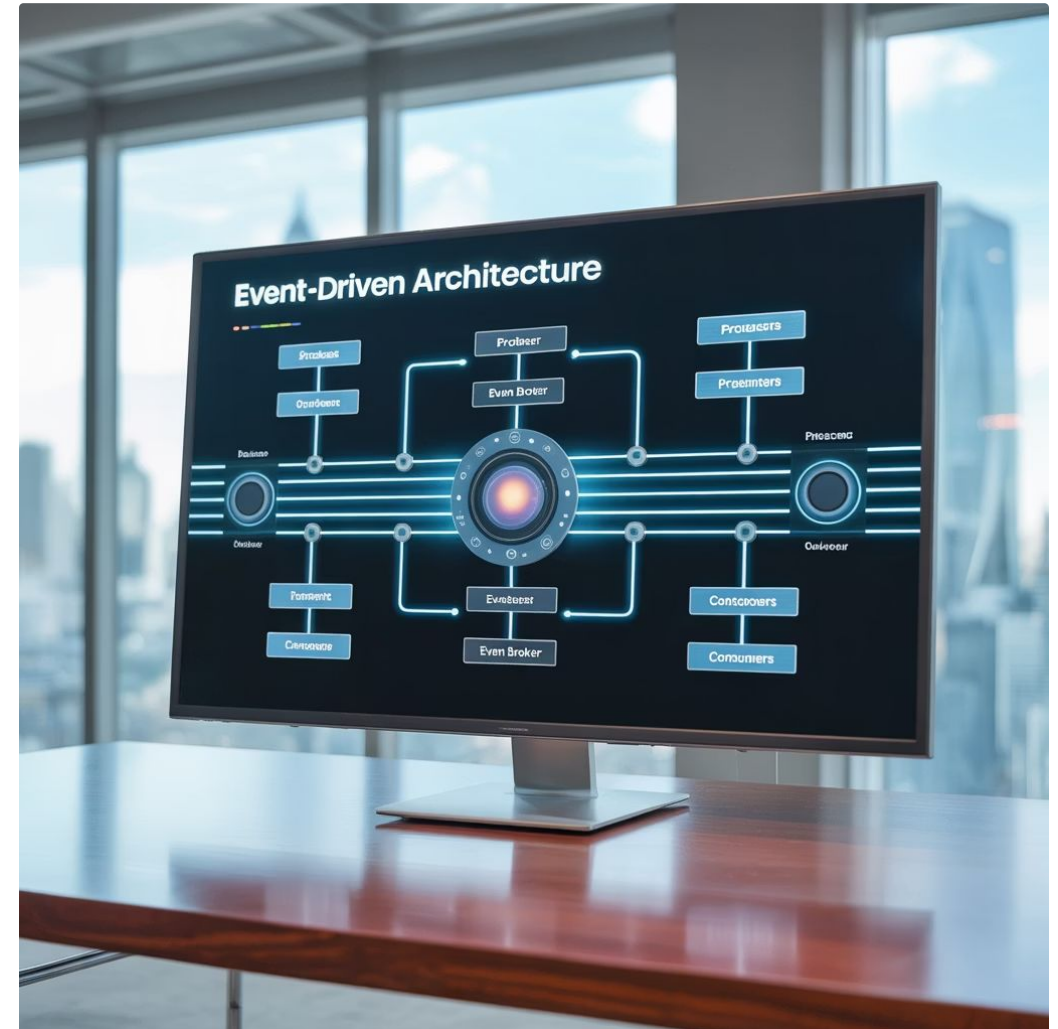
Kiến Trúc Hướng Sự Kiện (Event-Driven Architecture)



Kiến Trúc Hướng Sự Kiện Là Gì?

Kiến trúc hướng sự kiện (*Event-driven architecture - EDA*) là mô hình kiến trúc phần mềm trong đó các thành phần hoặc dịch vụ của hệ thống giao tiếp với nhau chủ yếu thông qua việc sản xuất và tiêu thụ các sự kiện.

Các "sự kiện" (event) này có thể là các hành động người dùng, cập nhật dữ liệu, hoặc các thông báo từ các hệ thống khác.



Lợi Ích Của Kiến Trúc Hướng Sự Kiện



Loose Coupling

Các thành phần hoạt động độc lập, không cần biết về nhau, giúp hệ thống dễ bảo trì và mở rộng.



Xử Lý Thời Gian Thực

Phản ứng nhanh với các sự kiện khi chúng xảy ra, tăng tính tương tác của hệ thống.



Khả Năng Phục Hồi

Hệ thống vẫn có thể hoạt động một phần nếu một số thành phần bị lỗi.

Phân Loại Kiến Trúc Hướng Sự Kiện

EDA được phân loại dựa trên cấu trúc liên kết (topology) giữa các component trong hệ thống với nhau, bao gồm hai mô hình phổ biến:

Broker Topology

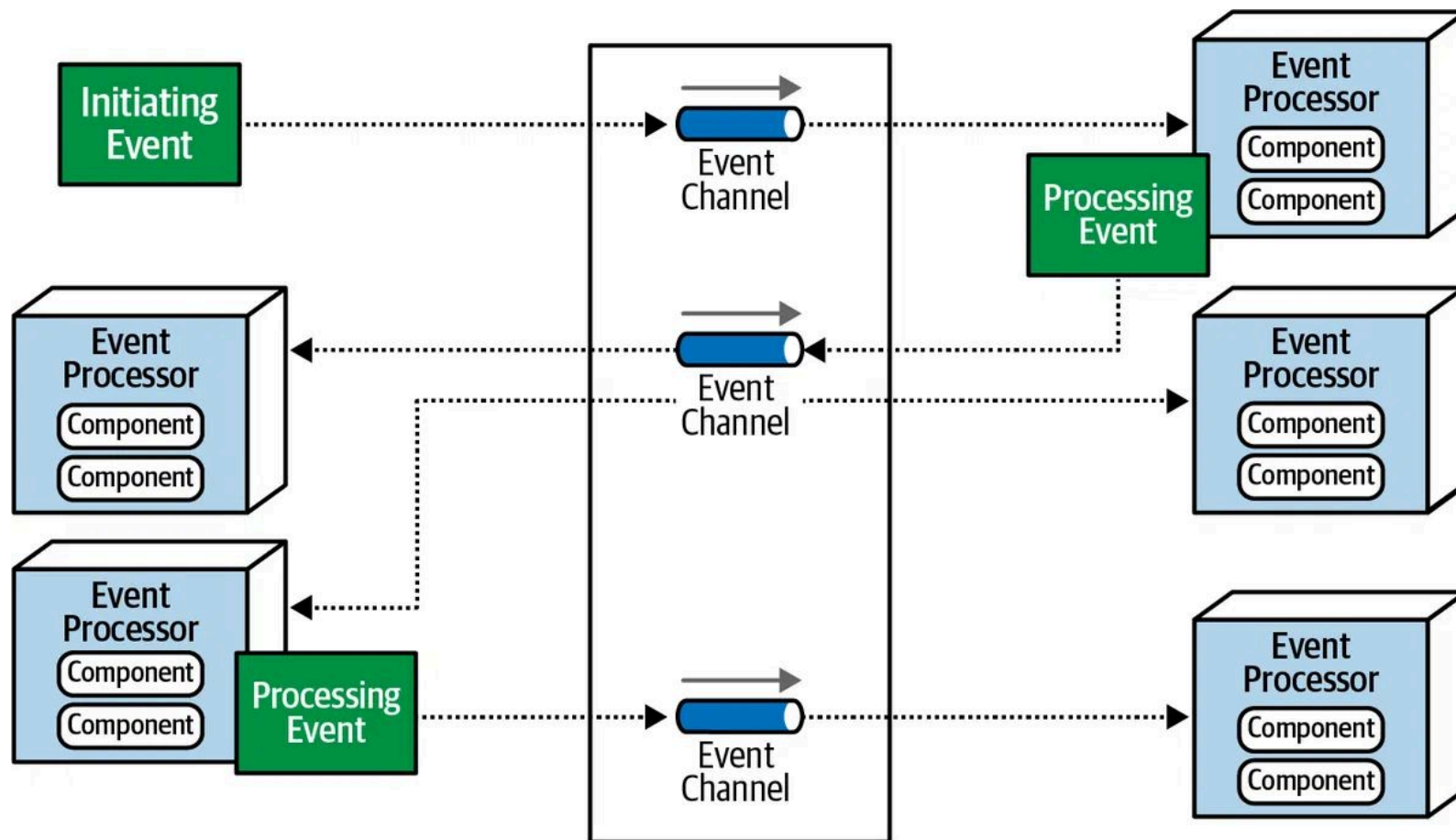
Luồng tin nhắn phân phối đều tới các event processor qua message broker mà không cần tới một trung tâm điều phối event.

Mediator Topology

Có một thành phần trung gian (mediator) được sử dụng để điều phối và quản lý luồng sự kiện giữa các service hoặc component khác nhau.

Broker Topology

Trong Broker topology, luồng tin nhắn sẽ phân phối đều tới các event processor qua message broker mà không cần tới một trung tâm điều phối event.



Các Thành Phần Chính Của Broker Topology

1 Initiating Event

Sự kiện ban đầu bắt đầu toàn bộ luồng sự kiện.

2 Event Channel

Được sử dụng để lưu trữ các sự kiện được tạo ra và phân phối các sự kiện đó đến một service phản hồi chúng. Event channel có thể ở dạng topic hoặc queue.

3 Event Broker

Chứa các event channel tham gia vào một luồng sự kiện.

4 Event Processor

Service chịu trách nhiệm xử lý sự kiện.

5 Processing Event

Sự kiện được tạo khi trạng thái của một service thay đổi và gửi thông báo cho phần còn lại của hệ thống về sự thay đổi trạng thái đó.

Cửa hàng online – khách đặt hàng

Bối cảnh: Khách bấm “Đặt hàng” trên website.

- **Initiating event (sự kiện khởi phát):**
OrderPlaced – phát sinh khi khách xác nhận giỏ hàng.
- **Event channel (kênh sự kiện):**
orders.topic (topic trên Kafka) dùng để **lưu & phát** sự kiện OrderPlaced đến các dịch vụ quan tâm (Inventory, Payment, Notification...).
- **Event broker (trung gian sự kiện):**
Kafka cluster (hoặc RabbitMQ) **chứa nhiều topic/queue**, ví dụ: orders.topic, inventory.topic, payments.topic. Broker đảm nhiệm lưu bền, phân phối, cân bằng consumer.
- **Event processor (dịch vụ xử lý):**
 - **Inventory Service** (consumer của orders.topic): nhận OrderPlaced → giữ hàng → phát sự kiện mới StockReserved.
 - **Payment Service** (consumer của orders.topic): nhận OrderPlaced → thu tiền → phát PaymentCaptured.
 - **Order Service** (consumer của cả StockReserved & PaymentCaptured): khi đủ điều kiện → cập nhật trạng thái đơn → phát OrderConfirmed.
- **Processing event (sự kiện xử lý tiếp theo):**
StockReserved, PaymentCaptured, OrderConfirmed... là **sự kiện phát sinh khi trạng thái service thay đổi, để thông báo cho hệ thống**. Ví dụ OrderConfirmed sẽ kích hoạt **Notification Service** gửi email.
- **Fire-and-forget broadcasting:**
Mỗi service **publish sự kiện và không cần chờ ai xác nhận**. Ví dụ Inventory publish StockReserved lên inventory.topic là xong – **không đợi** Order/Payment phản hồi. Dịch vụ nào quan tâm tự subscribe.

Tóm tắt:

UI → (publish) OrderPlaced → Kafka(orders.topic) → Inventory/Payment xử lý → (publish) StockReserved/PaymentCaptured → Kafka → Order Service tổng hợp → (publish) OrderConfirmed → Notification gửi mail.

Gọi xe công nghệ – khách đặt chuyến

Bối cảnh: Khách nhấn “Đặt xe”.

- **Initiating event:**

RideRequested – yêu cầu đặt xe mới.

- **Event channel:**

rides.topic (topic) để phân phối RideRequested cho **Dispatch Service** và **Pricing Service**.

- **Event broker:**

RabbitMQ (ví dụ) với các exchange/queue: rides.topic, pricing.topic, drivers.topic.

- **Event processor:**

- **Pricing Service:** nhận RideRequested → tính giá động → publish FareQuoted.
- **Dispatch Service:** nhận RideRequested → tìm tài xế phù hợp → publish DriverAssigned.
- **Trip Service:** nhận DriverAssigned + FareQuoted → tạo chuyến → publish TripStarted/TripCompleted.

- **Processing event:**

FareQuoted, DriverAssigned, TripStarted, TripCompleted... thông báo **trạng thái mới** để các service khác (Billing, Notification, Analytics) tự xử lý.

- **Fire-and-forget broadcasting:**

DriverAssigned được broadcast lên broker; Billing/Notification **tự lắng nghe**, nhà phát hành sự kiện **không đợi** phản hồi.

Luồng Tin Nhắn Trong Broker Topology



Initiating Event

Một initiating event được gửi tới một event channel nằm trong event broker để xử lý.



Event Processor

Event processor lấy event từ event channel để xử lý và thực hiện nhiệm vụ cụ thể.



Processing Event

Khi xử lý xong, event processor gửi processing event tới event channel để thông báo.



Lặp Lại

Các event processor khác lắng nghe và phản ứng lại bằng cách tạo ra event processing khác.

Lưu ý: Event processor luôn gửi processing event dù có ai lắng nghe hay không, giúp dễ dàng mở rộng hệ thống khi có nghiệp vụ mới.

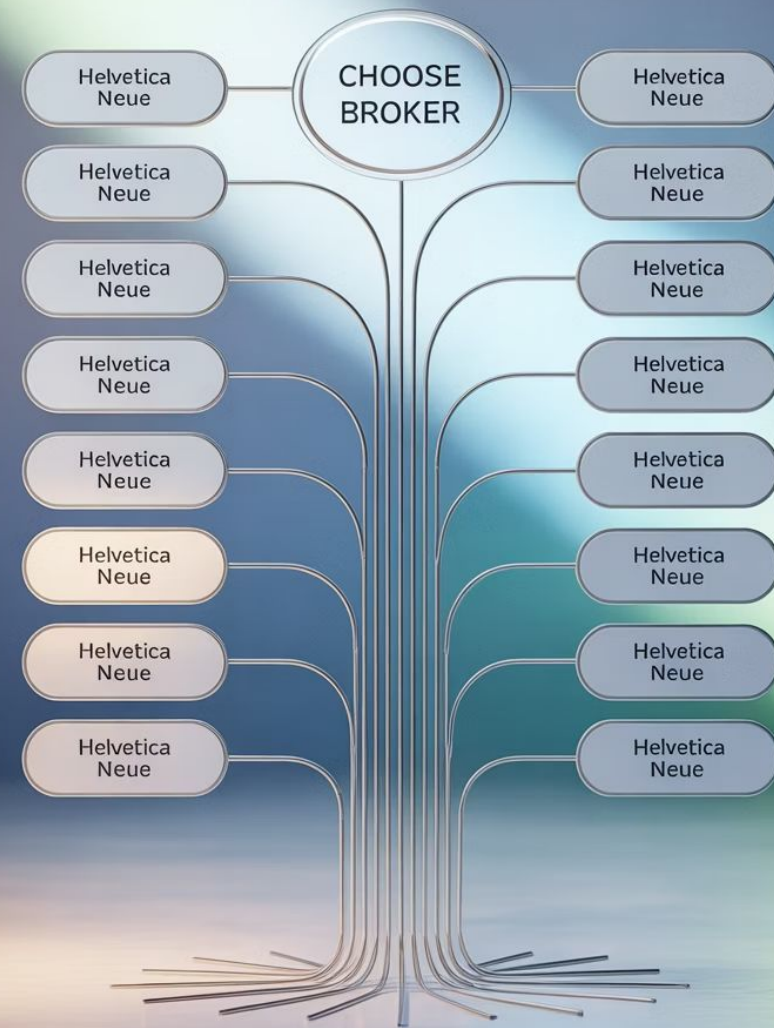
Khi Nào Nên Dùng Broker Topology?

Nên Dùng

- Cần giảm sự phụ thuộc lẫn nhau (loose coupling)
- Định tuyến đơn giản
- Các thành phần cần hoạt động độc lập

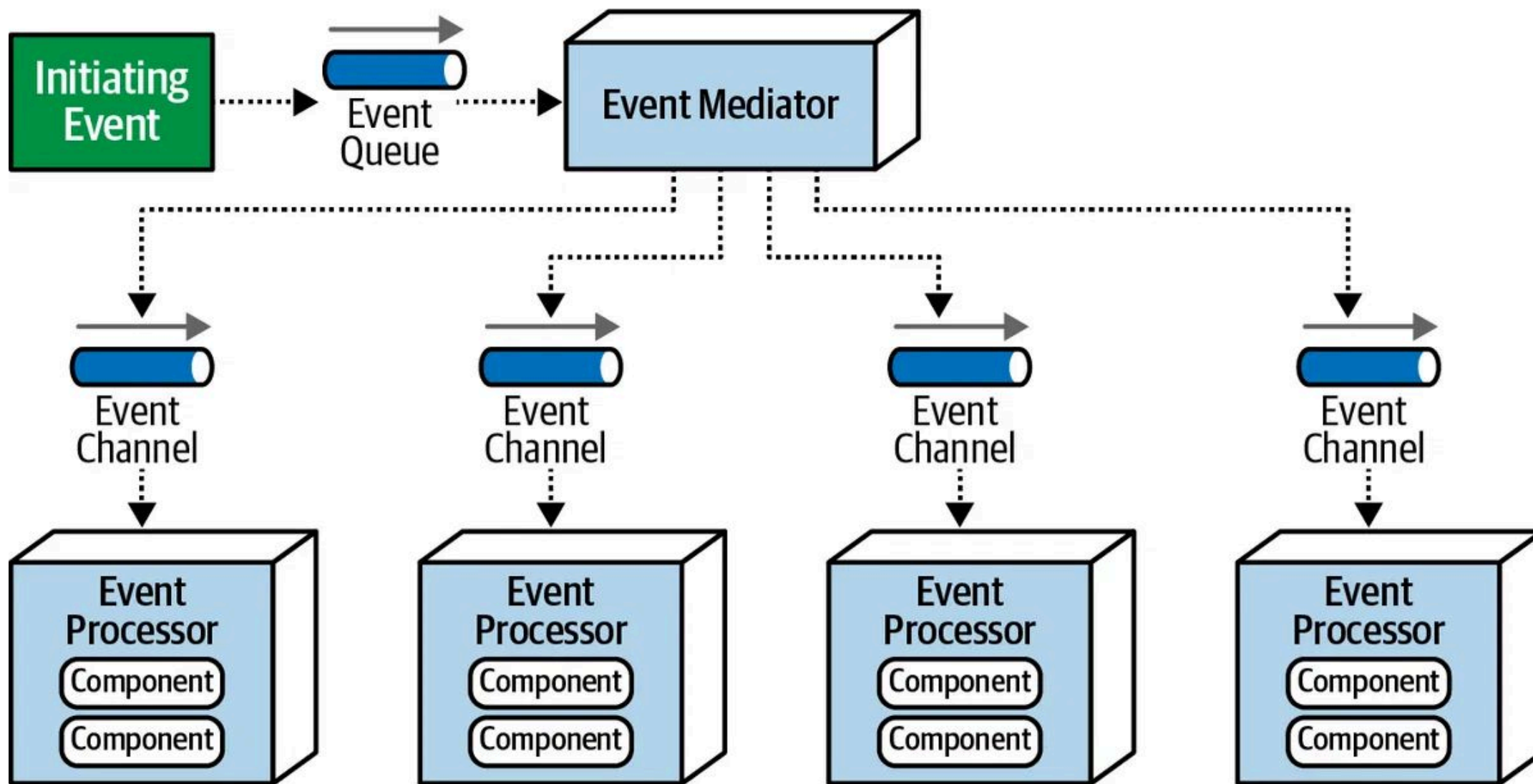
Không Nên Dùng

- Xử lý logic phức tạp giữa các thành phần
- Tích hợp và chuẩn hóa dữ liệu từ nhiều nguồn
- Cần kiểm soát trung tâm và quản lý workflow



Mediator Topology

Trong Mediator topology có một thành phần trung gian, thường được gọi là "mediator", được sử dụng để điều phối và quản lý luồng sự kiện giữa các service hoặc component khác nhau trong hệ thống.



Các Thành Phần Chính Của Mediator Topology

1 Initiating Event

Sự kiện ban đầu bắt đầu toàn bộ luồng sự kiện.

2 Event Queue

Lưu trữ các sự kiện ban đầu, đảm bảo chúng không bị mất mát trong quá trình xử lý.

3 Event Mediator

Nơi điều phối và quản lý luồng sự kiện, là trung tâm của mô hình Mediator.

4 Event Channel

Tương tự broker topology.

5 Event Processor

Tương tự broker topology.

Cửa hàng online – khách đặt hàng

- **Initiating event:**
OrderPlaced – khách bấm “Đặt hàng”.
- **Event queue (hàng đợi sự kiện ban đầu):**
orders.incoming.queue lưu OrderPlaced ngay khi phát sinh để không mất sự kiện và xử lý tuần tự (FIFO).
→ Đảm bảo nhất quán trước khi chuyển vào Mediator.
- **Event mediator (trung tâm điều phối):**
OrderFlowMediator đọc từ orders.incoming.queue, ra quyết định tuyến:
 - a. Gửi yêu cầu **reserve stock**
 - b. Gửi yêu cầu **capture payment**Mediator theo dõi trạng thái từng bước, gom kết quả, quyết định **tiến/thoái** (commit/compensate).
- **Event channel (kênh phân phối sau mediator):**
 - inventory.cmd.reserve (channel lệnh đến Inventory)
 - payment.cmd.capture (channel lệnh đến Payment)
 - order.evt.status (kênh sự kiện trạng thái cho các bên quan tâm)→ Giống “broker topology”: Mediator **publish/subscribe** qua các channel.
- **Event processor (dịch vụ xử lý):**
 - **Inventory Service** (consumer inventory.cmd.reserve) → xử lý → phát StockReserved lên inventory.evt
 - **Payment Service** (consumer payment.cmd.capture) → xử lý → phát PaymentCaptured lên payment.evt
 - **Mediator** subscribe inventory.evt & payment.evt, khi đủ điều kiện → phát OrderConfirmed lên order.evt.status.

Luồng tóm tắt:

UI → OrderPlaced → **Event queue** → **Mediator** → (publish) inventory.cmd.reserve & payment.cmd.capture → **Processors** xử lý → (publish) StockReserved & PaymentCaptured → **Mediator** hợp nhất → (publish) OrderConfirmed → Notification/Analytics nhận.

Đặt xe – khách yêu cầu chuyên

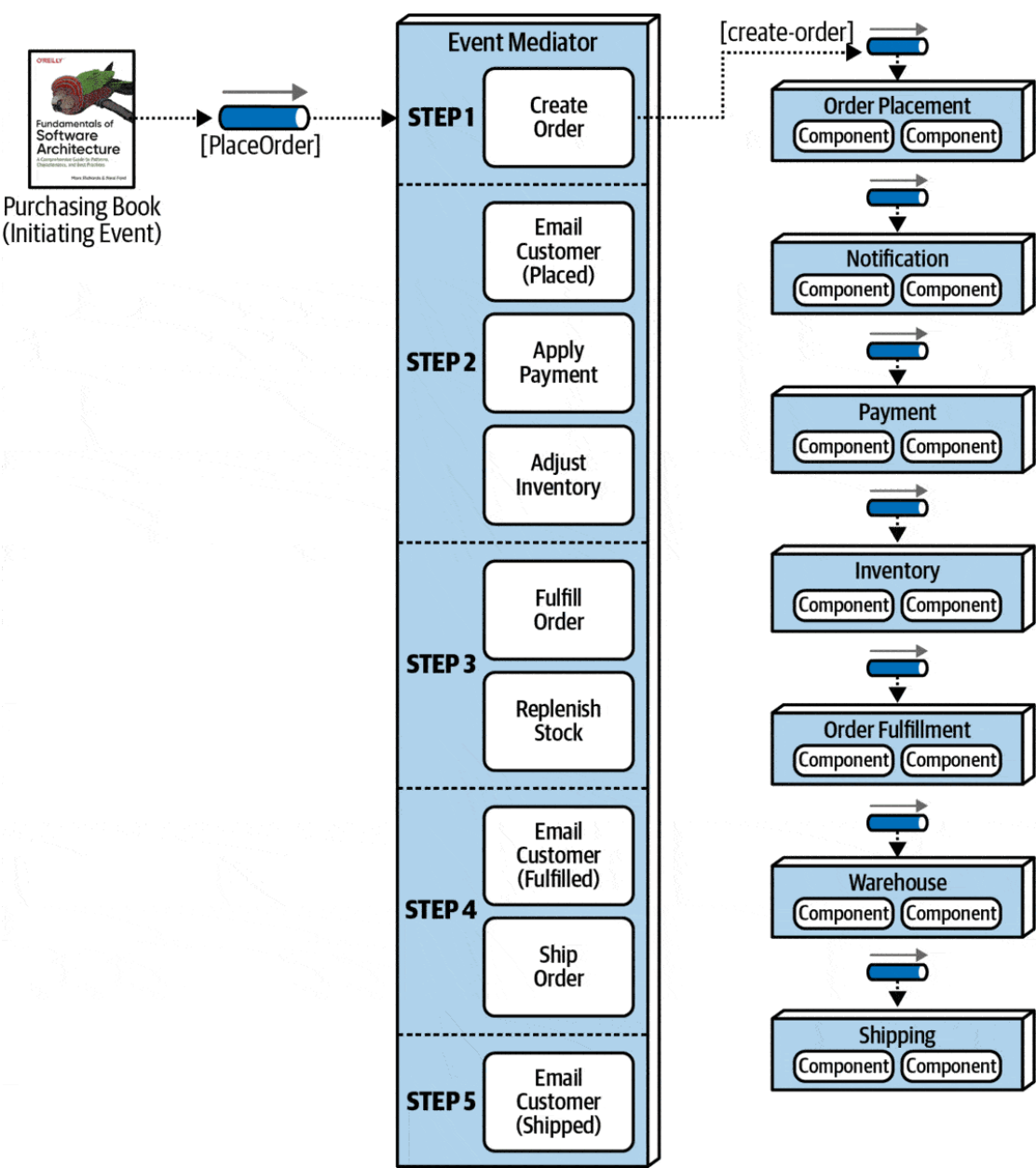
- **Initiating event:**
RideRequested.
- **Event queue:**
rides.incoming.queue nhận RideRequested để **chống mất mát** và **điều tiết tốc độ** (back-pressure).
- **Event mediator:**
RideOrchestratorMediator đọc yêu cầu, quyết định:
 - a. **Quote giá** (Pricing)
 - b. **Gán tài xế** (Dispatch)
Mediator đợi kết quả hai nhánh, xử lý timeouts, **compensation** nếu một nhánh fail.
- **Event channel:**
 - pricing.cmd.quote → Pricing Service
 - dispatch.cmd.assign → Dispatch Service
 - trip.evt.status → broadcast trạng thái chuyển cho các bên khác (Billing/Notif/ETL)
- **Event processor:**
 - **Pricing Service** → FareQuoted (trả về giá)
 - **Dispatch Service** → DriverAssigned (tài xế A)
 - **Mediator** nhận cả hai → phát TripCreated lên trip.evt.status.

Luồng tóm tắt:

App → RideRequested → **Event queue** → **Mediator** → (publish) pricing.cmd.quote, dispatch.cmd.assign → **Processors** → (publish) FareQuoted, DriverAssigned → **Mediator** hợp nhất → (publish) TripCreated.

Ví Dụ Về Mediator Topology

Hệ thống bán lẻ sử dụng Mediator topology để xử lý đơn hàng qua các bước: tạo đơn hàng, xử lý đơn hàng, thực hiện đơn hàng, giao hàng và thông báo cho khách hàng.



- **Bước 1: Tạo đơn hàng**

Sau khi nhận được event khởi tạo `PlaceOrder`, Mediator tạo ra event `create-order`, rồi gửi nó tới `order-placement` queue. `OrderPlacement` event processor sẽ nhận event này, xác thực thông tin và tạo order, trả cho Mediator thông báo đã xử lý xong kèm ID của order (`acknowledgment`). Tùy vào nghiệp vụ cụ thể, Mediator có thể gửi lại event cho client để thông báo đã tạo order.

- **Bước 2: Xử lý đơn hàng**

Mediator tạo và gửi 3 event đồng thời là `email-customer`, `apply-payment` và `adjust-inventory` tới các queue tương ứng là `notification`, `payment` và `inventory`. Mediator cần phải chờ cho các event processor xử lý thành công tất cả event trước khi chuyển sang bước 3.

- **Bước 3: Thực hiện đơn hàng**

Tương tự bước 2, các event `fulfill-order` và `order-stock` được gửi tới các queue tương ứng là `order-fulfillment` và `warehouse`.

- **Bước 4: Giao hàng**

Mediator thông báo trạng thái sẵn sàng giao hàng cho khách hàng bằng gửi event `email-customer` và `ship-order` tới các queue `notification` và `shipping`.

- **Bước 5: Thông báo cho khách hàng**

Thông báo cho khách hàng trạng thái đơn hàng qua email bằng event `email-customer`.

Luồng Xử Lý Trong Mediator Topology

Bước 1: Tạo đơn hàng

Mediator tạo event create-order, gửi tới order-placement queue. OrderPlacement processor xử lý và trả về ID đơn hàng.

Bước 2: Xử lý đơn hàng

Mediator gửi 3 event đồng thời: email-customer, apply-payment và adjust-inventory tới các queue tương ứng.

Bước 3: Thực hiện đơn hàng

Các event fulfill-order và order-stock được gửi tới các queue tương ứng.

Bước 4: Giao hàng

Mediator thông báo trạng thái sẵn sàng giao hàng qua event email-customer và ship-order.

Bước 5: Thông báo khách hàng

Thông báo cho khách hàng trạng thái đơn hàng qua email.



Khi Nào Nên Dùng Mediator Topology?

Nên Dùng

- Cần quản lý tập trung
- Xử lý logic phức tạp
- Tích hợp hệ thống đa dạng
- Cần kiểm soát chặt chẽ về quyền truy cập và bảo mật

Không Nên Dùng

- Yêu cầu hiệu suất cao và độ trễ thấp
- Cần tính đơn giản và dễ mở rộng
- Tách biệt và độc lập giữa các thành phần
- Khả năng chịu lỗi và phục hồi cao

So Sánh Broker và Mediator Topology

Tiêu Chí	Broker Topology	Mediator Topology
Điều phối trung tâm	Không	Có
Độ phức tạp	Thấp	Cao
Quản lý workflow	Hạn chế	Mạnh mẽ
Loose coupling	Cao	Trung bình
Điểm lỗi đơn	Không	Có (mediator)
Phù hợp với	Hệ thống đơn giản	Hệ thống phức tạp

Broker vs Mediator

- **Broker topology:** các service **giao tiếp gián tiếp** với nhau qua **broker**; không có “não trung tâm”. Mỗi service tự xử lý nghiệp vụ của mình khi nhận event.
- **Mediator topology:** có một **trung tâm điều phối (Mediator)** quyết định **thứ tự, nhánh, bù trừ (compensation)** và tổng hợp kết quả. Phù hợp **orchestration** (nhạc trưởng).

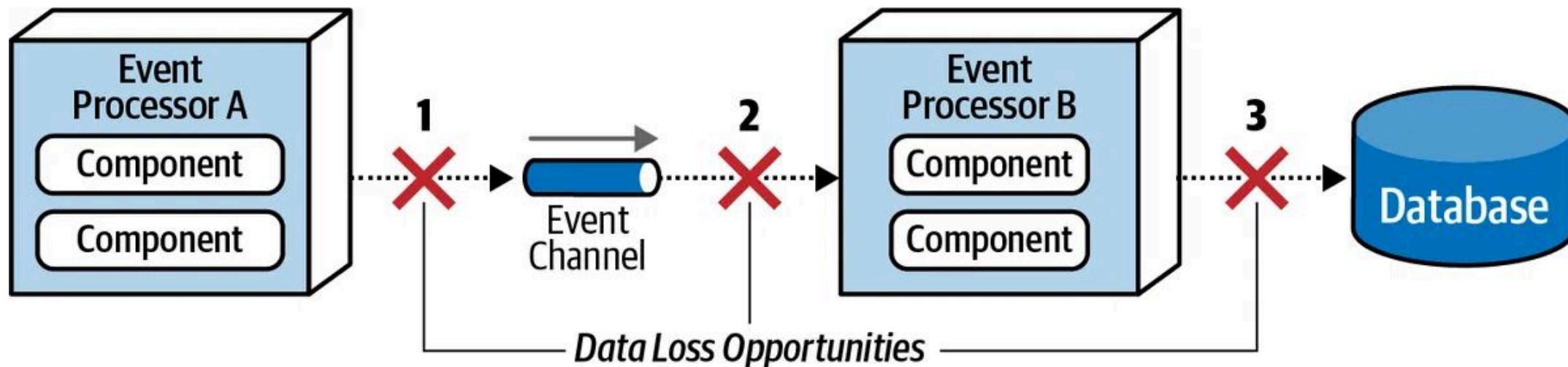
Ảnh dụ:

- **Broker** = dàn nhạc **tự phối hợp** qua tín hiệu (event).
- **Mediator** = có **nhạc trưởng** (orchestrator) ra hiệu từng phần.

Vấn Đề Mất Dữ Liệu Trong EDA

Một trong những vấn đề trọng tâm trong EDA là *mất dữ liệu*. Lỗi này có thể xảy ra ở các thời điểm sau:

1. Event processor đẩy event vào Event channel: Broker lỗi khiến dữ liệu bị mất.
2. Event channel đẩy event tới Event Processor: Event processor lỗi trước khi nhận được event.
3. Event processor lưu trữ dữ liệu đã xử lý vào database: Dữ liệu bị lỗi không lưu được hoặc database bị lỗi.



Giải Pháp Xử Lý Mất Dữ Liệu

Synchronous Send (Gửi đồng bộ)

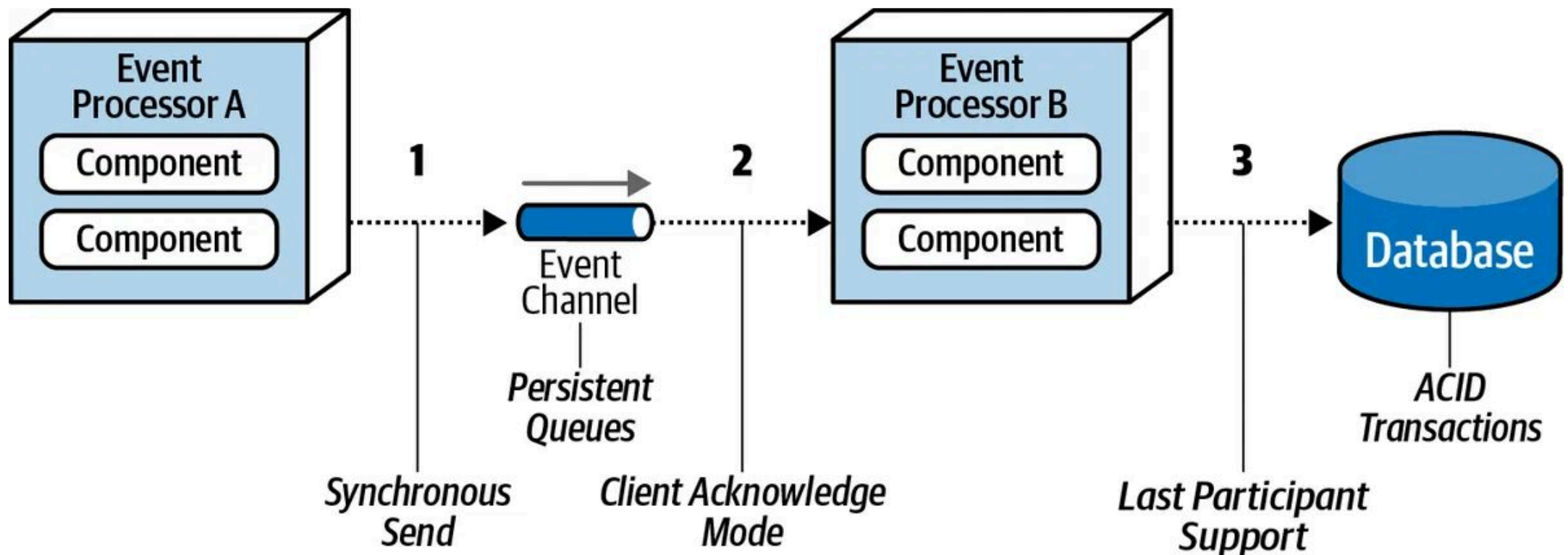
Triển khai Persisted message queues hỗ trợ tính năng guaranteed delivery. Broker lưu dữ liệu cả ở memory và kho chứa vật lý (filesystem/database).

Client Acknowledge Mode

Event không bị xóa khỏi queue khi được lấy ra, mà được gắn client ID. Nếu processor bị lỗi, event vẫn còn trong queue.

Last Participant Support (LPS)

Tận dụng ACID để đảm bảo dữ liệu được lưu. Các event được xác nhận sau khi data được lưu thành công vào database.





Ưu Điểm Của Kiến Trúc Hướng Sự Kiện

Khả Năng Mở Rộng

Dễ dàng thêm các thành phần mới mà không ảnh hưởng đến hệ thống hiện tại.

Phản Ứng Thời Gian Thực

Xử lý sự kiện ngay khi chúng xảy ra, tăng tính tương tác của hệ thống.

Tách Biệt Các Thành Phần

Giảm sự phụ thuộc giữa các thành phần, giúp dễ bảo trì và phát triển.

Khả Năng Phục Hồi

Hệ thống vẫn hoạt động một phần nếu một số thành phần bị lỗi.



Nhược Điểm Của Kiến Trúc Hướng Sự Kiện

Khó Theo Dõi Luồng

Khó theo dõi luồng xử lý khi hệ thống phức tạp với nhiều event và processor.

Xử Lý Lỗi Phức Tạp

Việc xử lý lỗi và đảm bảo tính nhất quán dữ liệu trở nên phức tạp hơn.

Khó Kiểm Thử

Việc kiểm thử toàn bộ hệ thống trở nên khó khăn do tính bất đồng bộ.

Độ Trễ Tiềm Ẩn

Có thể gây ra độ trễ trong xử lý, đặc biệt khi có nhiều lớp trung gian.

Ứng Dụng Thực Tế Của EDA



Thương Mại Điện Tử

Xử lý đơn hàng, quản lý kho, thông báo cho khách hàng về trạng thái đơn hàng.



Ngân Hàng

Xử lý giao dịch, phát hiện gian lận, cập nhật thông tin tài khoản theo thời gian thực.



IoT

Xử lý dữ liệu từ các thiết bị IoT, phản ứng với các sự kiện từ cảm biến.



Mạng Xã Hội

Xử lý thông báo, cập nhật feed, phân tích hành vi người dùng.

Tổng Kết

Kiến trúc hướng sự kiện (EDA) là mô hình kiến trúc phần mềm trong đó các thành phần giao tiếp thông qua việc sản xuất và tiêu thụ các sự kiện.

EDA bao gồm hai mô hình chính là **broker topology** và **mediator topology**.

EDA đem lại khả năng phản ứng nhanh và linh hoạt, tăng hiệu suất xử lý trong các hệ thống phức tạp.

Tuy nhiên, việc áp dụng EDA cần cân nhắc kỹ lưỡng về:

- Quản lý sự phụ thuộc
- Đảm bảo tính nhất quán dữ liệu
- Xử lý mất dữ liệu
- Khả năng theo dõi và gỡ lỗi

Lựa chọn mô hình phù hợp dựa trên yêu cầu cụ thể của hệ thống.

Tài liệu tham khảo

Architecture Pattern - Phần 3: Kiến trúc hướng sự kiện (Event-driven architecture)

[1] [2] [3] [4] [10] [31] Event-driven architecture: Everything you need to know in 2024

<https://ably.com/topic/event-driven-architecture>

[5] [8] Event Driven Architecture and Kafka Explained: Pros and Cons

<https://www.prodyna.com/insights/event-driven-architecture-and-kafka>

[6] [7] [25] Introduction to event-driven architecture

<https://aiven.io/blog/introduction-to-event-driven-architecture>

[9] [11] Publisher-Subscriber pattern - Azure Architecture Center | Microsoft Learn

<https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber>

[12] [13] [14] Event Sourcing pattern - Azure Architecture Center | Microsoft Learn

<https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing>

[15] [16] CQRS Pattern - Azure Architecture Center | Microsoft Learn

<https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs>

[17] [18] [19] Orchestration vs. Choreography in Microservices - GeeksforGeeks

<https://www.geeksforgeeks.org/system-design/orchestration-vs-choreography/>

[20] [21] [23] [24] Mastering Event-Driven Architecture: When and Why to Use It ⚡ - DEV Community

<https://dev.to/tejastn10/mastering-event-driven-architecture-when-and-why-to-use-it-268l>

[22] Best practices for monitoring event-driven architectures | Datadog

<https://www.datadoghq.com/blog/monitor-event-driven-architectures/>

[26] [29] [30] 10 Event-Driven Architecture Examples: Real-World Use Cases | Estuary

<https://estuary.dev/blog/event-driven-architecture-examples/>

[27] [28] System Design of Uber App | Uber System Architecture - GeeksforGeeks

<https://www.geeksforgeeks.org/system-design/system-design-of-uber-app-uber-system-architecture/>