An event-driven architecture uses events to trigger and communicate between

decoupled services and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website. Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a notification that an order was shipped).

معماري رويداد محور از رويداد ها براي راه اندازي و ايجاد ارتباط بين سرويس هاي جدا استفاده ميكند و در اپلیکیشن های مدرن ساخته شده با میکرو سرویس ها بسیار رایج هستند . یک رویداد یک تغییر یا بروز رسانی وضعیت است ، مانند یک کالا که درسبد خرید یک وب سایت تجارت الکترونیک قرار دارد . رویدادها می‌توانند دارای وضعیت باشند (کالای خریداری‌شده، قیمت آن و آدرس تحویل) یا رویدادها می‌توانند شناسه (اعلان ارسال سفارش) باشند.

Event-driven architectures have three key components: event producers, event routers, and event consumers. A producer publishes an event to the router, which filters and pushes the events to consumers. Producer services and consumer services are decoupled, which allows them to be scaled, updated, and deployed independently

معماری رویداد محور سه بخش کلیدی دارد : رویداد تولیدکنندگان ، رویداد روترها ، و رویداد مصرف کنندگان . یک تولید کننده رویدادی را به سمت روتر نشر می دهد که آن رویداد را فیلتر کرده و به سمت مصرف کننده می فرستد . سرویس های تولید کننده و سرویس های مصرف کننده از هم جدا هستند که به آن ها اجازه می‌دهد به صورت مستقل مقیاس گذاری ، بروز رسانی و به کار گرفته شوند .

Benefits of an event-driven architecture

مزایای معماری رویداد محور

### Scale and fail independently

### 1-به طور مستقل مقایسه و شکست بخورند

### By decoupling your services, they are only aware of the event router, not each other. This means that your services are interoperable, but if one service has a failure, the rest will keep running. The event router acts as an elastic buffer that will accommodate surges in workloads.

### به دلیل جدا بودن سرویس های شما، آنها فقط از رویداد های روتر آگاه هستند .نه از رویداد های هم این بدان معناست که سرویس‌های شما قابلیت همکاری دارند، اما اگر یکی از سرویس‌ها با مشکل مواجه شود، بقیه به کار خود ادامه می‌دهند. رویداد روتر به عنوان یک بافر قابل ارتجاع عمل میکند که با افزایش بارکاری جلوگیری می‌کند.

### Develop with agility

### 2-توسعه چابک

### You no longer need to write custom code to poll, filter, and route events; the event router will automatically filter and push events to consumers. The router also removes the need for heavy coordination between producer and consumer services, speeding up your development process.

### دیگر نیازی به نوشتن کد سفارشی برای نظر سنجی ، فیلتر و مسیر یابی رویداد ها ندارید . رویداد روتر به طور اتوماتیک فیلتر و ارسال میکند رویداد ها رو به مصرف کنندگان . روتر همچنین نیاز به هماهنگی شدید بین خدمات تولید کننده و مصرف کننده را از بین می برد و روند توسعه شما را سرعت می بخشد.

### 3- Audit with ease

### 3-تسهیل در حسابرسی

### An event router acts as a centralized location to audit your application and define policies. These policies can restrict who can publish and subscribe to a router and control which users and resources have permission to access your data. You can also encrypt your events both in transit and at rest.

### روتر رویداد به عنوان یک مکان متمرکز برای بررسی برنامه شما و تعریف خط مشی ها عمل می کند.این خط مشی ها می توانند انحصار ایجاد کنند که چه کسی . همچنین می توانید رویدادهای خود را هم در حالت حمل و نقل و هم در حالت استراحت رمزگذاری کنید.

### 4-cut costs

### 4-کاهش هزینه ها

### Event-driven architectures are push-based, so everything happens on-demand as the event presents itself in the router. This way, you’re not paying for continuous polling to check for an event. This means less network bandwidth consumption, less CPU utilization, less idle fleet capacity, and less SSL/TLS handshakes.

### معماری رویداد محور مبتنی بر فشار است ، بنابراین همه چیز بر اساس تقاضا اتفاق می‌افتد زیرا رویداد‌ها خود را در روتر ها نمایش می‌دهد . این به معنای کاهش مصرف پهنای باند شبکه ، استفاده کمتر CPU ، ظرفیت ناوگانی کمتر .

### نحوه کارکرد معماری رویداد محور

### نحوه کار این معماری را با ذکر یک مثال توضیح می دهیم

### C:\Users\user\Desktop\للل.jpg.png

### زمان استفاده از این معماری

### Cross-account, cross-region data replication

### تکثیر داده های متقابل، بین منطقه ای

### You can use an event-driven architecture to coordinate systems between teams operating in and deploying across different regions and accounts. By using an event router to transfer data between systems, you can develop, scale, and deploy services independently from other teams.

### شما می توانید از معماری رویداد محور برای هماهنگ کردن سیستم ها بین تیم های عملیاتی و حساب هایی که در ناحیه های متفاوتی قرار گرفته اند استفاد کنید . با اسفتاده از روتر رویداد برای جابجایی داده ها بین سیتم ها ، شما می توانید توسعه دهید ، مقیاس بدهید و سرویس ها را مستقر کنید مستقل از سایر تیم ها .

### Resource state monitoring and alerting

### نظارت و هشدار وضعیت منابع

### Rather than continuously checking on your resources, you can use an event-driven architecture to monitor and receive alerts on any anomalies, changes, and updates. These resources can include storage buckets, database tables, serverless functions, compute nodes, and more.

### به جای چک کردن پیوسته منابع خود ، شما میتوانید از یک معماری رویداد محور برای نظارت و دریافت هشدار در رابطه با هر ناهنجاری ، تغییرات و بروز رسانی ها ، استفاده کنید . منابع می توانند شامل سطل های ذخیره سازی ، جدول پایگاه داده ها ، رویداد ها از دست دادن سرورها ، گره های محاسباتی و غیره باشند .

### Fan-out and parallel processing

### میزان خروجی و پردازش موازی

### If you have a lot of systems that need to operate in response to an event, you can use an event-driven architecture to fan out the event without having to write custom code to push to each consumer. The router will push the event to the systems, each of which can process the event in parallel with a different purpose.

### اگر شما تعداد زیادی سیستم دارید که در پاسخ به یک رویداد نیاز به یک اقدام دارند ، شما میتوانید از معماری رویداد محور برای نمایش رویداد بدون نیاز به نوشتن یک کد اختصاصی برای ارسال به هر مصرف کننده استفاده کنید . روتر رویداد را به سمت سیستم هایی هدایت خواهد کرد که هر یک می توانند رویداد را به طور موازی با اهداف خاص پردازش کنند .

### Integration of heterogeneous systems

### ادغام سیستم‌های ناهمگن

### If you have systems running on different stacks, you can use an event-driven architecture to share information between them without coupling. The event router establishes indirection and interoperability among the systems, so they can exchange messages and data while remaining agnostic.

### اگر شما سیستم هایی دارید که بر روی پشته‌های متفاوتی اجرا میشوند ، شما می‌توایند از معماری رویداد محور برای به اشتراک گذاری اطلاعات بین تیم‌ها بدون گروه شدن استفاده کنید . روتر رویداد غیرمستقیم و قابلیت همکاری بین سیستم ها را ایجاد می کند، بنابراین آن‌ها مي‌توانند پیام‌ها و داده‌ها را مبادله کنند و در عین حال ناشناس باقی بمانند .

### چرا باید از معماری رویداد محور استفاده کنیم ؟

### معماری رویداد محور برای بهبود چابکی و جابجایی سریع بسیار ایده آل است . آن‌ها اغلب در اپلیکیشن‌های مدرنی که از میکروسرویس‌ها استفاده می‌کنند یافته می‌شوند یا هر اپلیکیشنی که اجزاء جدا از هم دارد . زمانی که یک معماری رویداد محور را اتخاذ می‌کنید ، ممکن است لازم باشد در نحوه مشاهده طرح برنامه خود تجدید نظر کنید. برای اینکه خود را برای موفقیت آماده کنید، موارد زیر را در نظر بگیرید:

### دوام منبع رویداد شما منبع رویداد شما باید قابل اعتماد باشد و در صورت نیاز به پردازش تک تک رویدادها، تحویل را تضمین کند.

### الزامات کنترل عملکرد شما برنامه شما باید بتواند ماهیت ناهمزمان روترهای رویداد را مدیریت کند.

### ردیابی جریان رویداد شما غیر جهت معرفی شده توسط یک معماری رویداد محور، امکان ردیابی پویا را از طریق خدمات نظارتی فراهم می کند، اما نه ردیابی ایستا از طریق تجزیه و تحلیل کد.

### داده ها در منبع رویداد شما. اگر نیاز به بازسازی حالت دارید، منبع رویداد شما باید کپی شده و سفارش داده شود.

### رفرنس این بخش https://aws.amazon.com/event-driven-architecture/

**Overview**

### بررسی اجمالی

Event-driven architecture is a [software architecture](https://www.redhat.com/en/topics/cloud-native-apps/what-is-an-application-architecture) and model for application design. With an event-driven system, the capture, communication, processing, and persistence of events are the core structure of the solution. This differs from a traditional request-driven model.

معماری رویداد محور یک معماری نرم افزاری و مدل برای طراحی نرم افزار است . با یک سیستم رویداد محور، ضبط، ارتباط، پردازش و تداوم رویدادها ساختار اصلی راه حل هستند.این متفاوت از مدل سنتی درخواست محور است .

Many modern application designs are event-driven, such as [customer engagement frameworks](https://www.redhat.com/en/solutions/customer-experience) that must utilize customer data in real time. Event-driven apps can be created in any programming language because event-driven is a programming approach, not a language. Event-driven architecture enables minimal coupling, which makes it a good option for modern, distributed application architectures.

بسیای از طراحی های اپلیکیشن مدرن رویداد محور است ، مانند چارچوب های تعامل با مشتری که باید از داده های مشتری در زمان واقعی استفاده کنند . اپلیکیشن های رویداد محور می توانند با تمام زبان های برنامه نویسی به وجود بیایند زیرا معماری رویداد محور یک رویکرد است نه یک زبان . معماری رویداد محور حداقل کوپلینگ را ممکن می کند، که آن را به گزینه خوبی برای معماری های کاربردی مدرن و توزیع شده تبدیل می کند.

An event-driven architecture is loosely coupled because event producers don’t know which event consumers are listening for an event, and the event doesn’t know what the consequences are of its occurrence.

یک معماری رویداد محور به طور ضعیفی پیوند خورده است، زیرا تولیدکنندگان رویداد نمی‌دانند مصرف‌کنندگان رویداد به کدام رویداد گوش می‌دهند، و رویداد نمی‌داند عواقب وقوع آن چیست.

یک رویداد چیست ؟

An event is any significant occurrence or change in state for system hardware or software. An event is not the same as an event notification, which is a message or notification sent by the system to notify another part of the system that an event has taken place.

یک رویداد هر رخداد قابل توجه یا تغییر در وضعیت برای سیستم سخت افزاری یا نرم افزاری است . یک رویداد همان اعلام رویداد نیست، که پیام یا اطلاع بوسیله سیستم برای اطلاع دادن به بخش دیگری از سیستم که یک رویداد صورت گرفته است ، ارسال شود .

The source of an event can be from internal or external inputs. Events can generate from a user, like a mouse click or keystroke, an external source, such as a sensor output, or come from the system, like loading a program.

منبع یک رویداد میتواند ناشی از ورودی های خروجی یا داخلی باشد .رویداد میتواند توسط یک کاربر به وجود آورد ، برای مثال یک کلیک موس یا کیبورد ، یک منبع خارجی مانند سنسور خارجی ، یا از سیستم مانند لود شدن یک برنامه .

چگونه یک معماری رویداد محور کار می‌کند؟

Event-driven architecture is made up of event producers and event consumers. An event producer detects or senses an event and represents the event as a message. It does not know the consumer of the event, or the outcome of an event.

معماری رویداد محور از تولید کنندگان رویداد و مصرف کنندگان رویداد تشکیل شده است . یک تولید کننده رویداد یک رویداد را تشخیص و حس می‌کند و رویداد را تحت یک پیام نشان می‌دهند .

After an event has been detected, it is transmitted from the event producer to the event consumers through event channels, where an event processing platform processes the event asynchronously. Event consumers need to be informed when an event has occurred. They might process the event or may only be impacted by it.

بعد از اینکه یک رویداد تشخیص داده می‌شود،توسط کانال‌های رویداد از تولید کننده رویداد به مصرف کننده رویداد منتقل می‌شود ، جایی که یک پلتفرم پردازش رویداد ، رویداد ها را به صورت ناهمزان پردازش می‌شود . مصرف کنندگان رویداد نیازدارند که هنگام رخداد یک رویداد ، اطلاع رسانی شوند.

آنها ممکن است که رویداد را پردازش کنند یا فقط از آن تاثیر بپذیرند .

The event processing platform will execute the correct response to an event and send the activity downstream to the right consumers. This downstream activity is where the outcome of an event is seen.

پلتفرم پردازش رویداد پاسخ صحیح را به یک رویداد اجرا می‌کنند و فعالیت پایین دست را به سمت مصرف کننده مناسب ارسال کند . این فعالیت پایین دست خروجی رویدادی است که دیده شده است .

[Apache Kafka](https://www.redhat.com/en/topics/integration/what-is-apache-kafka) is a distributed data streaming platform that is a popular event processing choice. It can handle publishing, subscribing to, storing, and processing event streams in real time. Apache Kafka supports a range of use cases where high throughput and scalability are vital, and by minimizing the need for point-to-point integrations for data sharing in certain applications, it can reduce latency to milliseconds.

There are other middleware event managers available that can serve as an event processing platform.

مدل های معماری رویداد محور

An event driven architecture may be based on either a pub/sub model or an event stream model.

معماری رویداد محور ممکن است براس یک مدل pub/sub باشد یا بر اساس مدل جریان رویداد باشد .

Pub/sub model

مدل pub/sub

This is a messaging infrastructure based on subscriptions to an event stream. With this model, after an event occurs, or is published, it is sent to subscribers that need to be informed.

این یک زیرساخت پیام رسانی مبتنی بر اشتراک در جریان رویداد است . با این مدل، بغد از رخداد یک رویداد ، یا منتشر شدن آن ، مشترکین نیاز به پیام رسانی دارد .

Event streaming model

مدل جریان رویداد

With an event streaming model, events are written to a log. Event consumers don’t subscribe to an event stream. Instead, they can read from any part of the stream and can join the stream at any time.

با یک مدل جریان رویداد ، رویداد‌ها در یک گزارش نوشته می‌شوند. مصرف کنندگان رویداد در یک جریان رویداد مشترک نمی‌شوند . در عوض، ، آنها می توانند از هر بخشی از جریان بخوانند و در هر زمان می توانند به جریان بپیوندند.

چند نوع مختلف جریان رویداد وجود دارد:

* **Event stream processing** uses a data streaming platform, like Apache Kafka, to ingest events and process or transform the event stream. Event stream processing can be used to detect meaningful patterns in event streams.
* **پردازش جریان رویداد** از یک پلتفرم جریان داده استفاده می‌کنند ، مانند اپاچی کافکا ، برای جذب رویداد ها و پردازش یا جابجایی یک جریان رویداد استفاده می‌کند . پردازش جریان رویداد ميتوانند استفاده شوند براي تشخيص الگوهاي معنادار در رويداد جريان استفاده شوند .
* **Simple event processing** is when an event immediately triggers an action in the event consumer.
* **پردازش رویداد ساده** زمانی است که یک رویداد بلافاصله اقدامی را در مصرف کننده رویداد راه اندازی میکند .
* **Complex event processing** requires an event consumer to process a series of events in order to detect patterns.
* **پردازش رویداد پیچیده** یک مصرف کننده رویداد نیاز دارد تا یک سری از رویداد ها را به ترتیب برای تشخیص الگو پردازش کند .

**رفرنس این بخش (**[**https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture**](https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture) **)**

**تعریف معماری رویداد محور**

Event-driven architecture (EDA) is a software design pattern in which decoupled applications can asynchronously publish and subscribe to events via an event broker (modern messaging-oriented-middleware).

معماری رویداد محور (EDA) یک الگوی طراحی نرم‌افزاری است که در آن برنامه‌های کاربردی جداشده می‌توانند به‌طور ناهمزمان رویدادها را از طریق یک واسطه رویداد (میان‌افزار مبتنی بر پیام‌رسانی مدرن) منتشر کرده و مشترک شوند.

Event-driven architecture is a way of building enterprise IT systems that lets information flow between applications, microservices and connected devices in a real-time manner as events occur throughout your business, instead of periodically polling for updates.

معماری رویداد محور یک راهی برای ساخت سیستم های آیتی پیشرفته است که به جای نظر سنجی دوره ای برای بروز رسانی اجازه میدهد اطلاعات در بین اپلیکیشن ها ، میکروسرویسها و دستگاه های متصل به صورت همزمان جریان داشته باشند ، زیرا رویداد ها در سراسر کسب و کار شما رخ میدهند .

تعریف معماری رویداد محور

**Event-driven architecture** (**EDA**) is a [software architecture](https://en.wikipedia.org/wiki/Software_architecture) paradigm promoting the production, detection, consumption of, and reaction to [events](https://en.wikipedia.org/wiki/Event_(computing)).

معماری رویداد محور (EDA) یک پارادایم معماری نرم افزاری است که تولید، تشخیص، مصرف و واکنش به رویدادها را ترویج می کند.

وب سایت ویکی پدیا

تعریف ساده از معماری رویداد محور

**Event-driven Architecture** (**EDA**) is a software architecture paradigm promoting the production and consumption of **events**.

معماری رویداد محور(EDA ) یک پارادیام معماری نرم‌افزار که تولید و مصرف رویداد را ترویج می‌کند .

**An event represents an action of significant interest.** Often, events correspond to a creation or a change of state of some entity. For example, raising an order in an e-commerce application is an event. Dispatching a product as a result of an earlier order is also an event. A customer submitting a review for a received product is — you guessed it — an event.

یک رویداد نشان دهنده‌ی یک عمل با علاقه قابل توجه است . اغلب، رویدادها مربوط به ايجاد يا تغيير وضعيت بعضي از موجوديت هاست . براي مثال، ايجاد يك سفارش در يك وب سايت تجاري يك رويداد است . ارسال يك سفارش در نتيجه سفارش قبلي نيز يك رويداد است . مشتري كه درحال ارسال نظر درمورد سفارش رسيده است نيز يك رويداد است .

رويدادي كه هرگز اتفاق نيافتاد

The peculiar thing about events is that they are not explicitly communicated to specific parties that might care about them. Events “just happen”. Crucially, they happen irrespective of whether certain parties are interested in them. This might sound like the oft-quoted philosophical thought experiment: *“if a tree falls in a forest and no one is around to hear it, does it make a sound?”*. But that is precisely what makes events so powerful — the fact that an **event translates to a self-contained record** of something occurring means that events and, by extension, their emitters, are **fundamentally decoupled** from their handlers. In fact, **producers** of event records often have no knowledge of who the **consumers** might be, nor whether consumers exist at all.

نکته عجیب در مورد رویدادها این است که آنها به طور صریح به طرف های خاصی که ممکن است به آنها اهمیت دهند اطلاع داده نمی شوند. مهم اين است ، صرف نظر از اينكه بخش هاي از آنها مورد علاقه هستند اتفاق خواهند افتاد . این ممکن است شبیه آزمایش فکری فلسفی که اغلب نقل شده است به نظر برسد. *“if a tree falls in a forest and no one is around to hear it, does it make a sound?”.* اما دقيقا همينطوره چه چيزي رويداد را بسيار قدرتمند مي‌سازد – این واقعیت که یک رویداد به یک رکورد مستقل از یک اتفاق ترجمه می شود به این معنی است که رویدادها و در نتیجه انتشار دهنده های آنها اساساً از کنترل کننده های آنها جدا شده اند. در حقيقت ، توليد كننده‌هاي رويداد بدون اين دانش كه مصرف كننده كيست يا آيا اصلا وجود دارد ركورد ايجاد ميكنند .

**A record typically contains the information necessary to describe an event.** In our earlier example of an order, the corresponding event might be described by a simple JSON document that might look something like this:

يك ركورد شامل اطلاعات ضرروري براي توصيف يك رويداد است . در مثال قبلي ما از يك سفارش،رويداد متناظر بايك سند جيسون ساده كه ممكن است مانند زير باشد توصيف شده :

Admittedly, the example above is probably an oversimplified take on an order, but it will suffice. The application raising the order (say, the *shopping cart service*) has no idea *who* will process the order, *when, how* or even *why*. **A producer ensures that everything that a prospective consumer needs to process the event is captured.** That said, the order record does not strictly need to include every single attribute required for its fulfilment. For example, the dimensions of the product, its stocking location and the shipping address of the customer are not directly specified but can be resolved by following the IDs captured in the order record. The concept of *foreign keys* that you may be familiar with from relational databases also applies to events.

مسلماً، مثال بالا احتمالاً برداشتی بیش از حد ساده از یک سفارش است، اما کافی است.اپلیکیشن سفارشی را اجرا می کند که ایده وجود ندارد که چه کسی سفارش را پردازش می‌کند ،کی یا چطور یا چرا. تولییدکننده اطمینان حاصل میکنه که تمام چیزهای که یک مصرف کننده بالقوه برای پردازش آن نیاز دارد ضبط شده است . بیان میدارد که ، یک سفارش سابقه دار نگاری شده موکدا نیاز ندارد شامل تمام ویژگی های مورد نیاز برای انجام آن باشد . برای مثال، ابعاد محصول ، موقعیت کالا در انبار و آدرس محل دریافت مشتری به طور مستقیم مشخص نشده‌اند ، اما با دنبال کردن شماره های ثبت شده در سابقه سفارش قابل حل است . محتوای کلید های خارجی که ممکن است از پایگاه داده رابطه ای با آن آشنا باشید در اینجا نیز صدق کند .

کانال گذاری رویداد ها

اگر تولید کننده و مصرف کننده رویداد از هم آگاه نیستند پس چگونه با هم ارتباط برقرار می‌کنند ؟

The clue is in the term “record”. Events are usually persisted in a well-known location, called a log. (Sometimes, the term ledger may be used.) Logs are low-level, append-only data structures that allow an event to be saved by producers in a location where other parties (called consumers) can later access it. All manipulations of the log are facilitated by brokers — persistent middleware that resides between producers and consumers. Once an event has been published, anyone and everyone can consume that event.

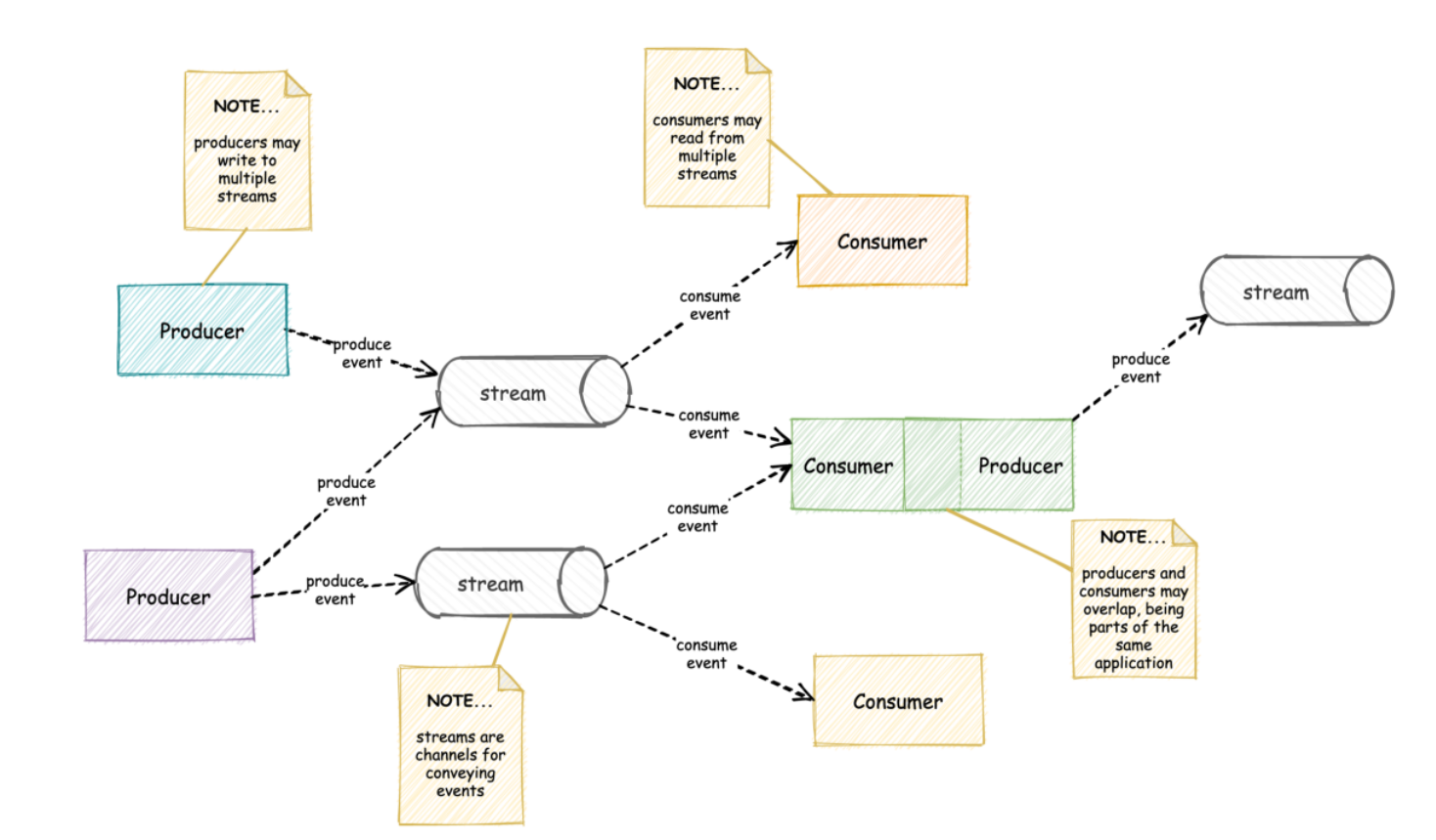
پاسخ در کلمه رکورد است . ". رویدادها معمولاً در یک مکان شناخته شده به نام log ادامه دارند . لاگ ها ساختار داده‌هایی سطح پایین و فقط ضمیمه هستند که اجازه می‌دهند كه يك رويداد توسط توليدد كننده در در موقعيتي ذخيره شوند كه بعدا توسط ساير بخش ها به آن دسترسي وجود داشته باشد . تمامي تغييرات در گزارشات توسط بروكر ها تسهيل مي‌يابد – ميان افزار هاي هميشگي كه بين مشتريان و تولييد كنندكان سكونت دارند . اگر يك رويداد منتشر شوند تمامي مصرف كنندگان ميتوانند آن را مصرف كنند .

When dealing with event-driven systems, we often use the term **stream** to describe an interface to one or more logs. While a log is a physical concept (implemented using files), a stream is a logical construct that represents events as an unbounded sequence of records, subject to certain ordering constraints. Different event streaming platforms might use proprietary names to refer to streams. [Apache Kafka](https://medium.com/swlh/apache-kafka-in-a-nutshell-5782b01d9ffb) — by far the most popular event streaming platform in existence — describes streams in terms of **topics** and **partitions**.

وقتي با يك سيستم رويداد محور معامله مي‌كنيم ، ما اغلب از يك جريان اصلاحي استفاده مي‌كنيم تا يك واسط به يك يا بيش از چند گزارش را توصيف كنيم . زماني كه يك گزارش يك محتواي فيزيكي است ، يك جريان يك ساختار منطقي دارد كه رويداد ها را تحت عنوان دنباله نامحدودي از ركورد ها كه مشروط به محدوديت‌هاي سفارشي خاص است نمايش مي‌دهد . پلتفرم‌های پخش رویدادهای مختلف ممکن است از نام‌های اختصاصی برای اشاره به جریان‌ها استفاده کنند. . آپاچی کافکا - تا حد زیادی محبوب‌ترین پلتفرم استریم رویداد موجود - استریم‌ها را از نظر موضوعات و پارتیشن‌ها توصیف می‌کند.

The relationship between producers, consumers and streams is depicted in the following reference model.

رابطه بین تولیدکنندگان، مصرف کنندگان و جریان ها در مدل مرجع زیر نشان داده شده است.



یک نقطه بازرسی سریع برای کمک به تقویت درک ما:

 **Events are actions of interest that occur at discrete points in time** and may be externally observed and described.

رويداد ها اقدامات مورد علاقه‌اي هستند كه در نقاط مجزايي از زمان اتفاق مي‌افتد و ممكن است به صورت خارجي مشاهده و تفسير شوند .

 **Events are persisted as records.** Events and records, despite being related, are technically different things. An event is an *occurrence* of something (e.g., a state change), and is intangible on its own. A record is an accurate description of that event. We often use the term *event* to refer to its record.

رويداد‌ها به عنوان ركورد ذخيره مي‌شوند . رويدادها و ركورد‌‌‌ها با اينكه با هم مرتبط هستند ولي چيزهاي متفاوتي هستند . یک رویداد، وقوع چیزی است (مثلاً تغییر حالت)، و به خودی خود نامشهود است. يك ركورد يك توصيف واقعي از يك رويداد است . ما اغلب از اصطلاح رویداد برای اشاره به رکورد آن استفاده می کنیم. ف‌اف

Producers are receptors that detect events by publishing corresponding records to a stream.

تولیدکنندگان گیرنده هایی هستند که رویدادها را با انتشار رکوردهای مربوطه در یک جریان شناسایی می کنند.

 **Streams are persistent sequences of records.** They are typically backed by one or more disk-based **logs** under the hood. Equally, streams might be backed by database tables, a distributed consensus protocol, or even a blockchain-style decentralized ledger.

جريان‌ها دنباله‌اي از ركوردها هستند . آنها معمولاً توسط یک یا چند لاگ مبتنی بر دیسک در زیر هود پشتیبانی می شوند.‌اي ‌اي

به همين ترتيب ، به همین ترتیب، جریان ها ممکن است توسط جداول پایگاه داده، یک پروتکل اجماع توزیع شده، یا حتی یک دفتر کل غیرمتمرکز به سبک بلاک چین پشتیبانی شوند.

اندا

 **Brokers govern access to streams**, facilitate the reading and writing operations, handle consumer state and perform various housekeeping tasks on the streams. For example, a broker might truncate the contents of a stream when it overflows with records.

بروكر‌ها دسترسي به جريان‌ها را كنترل مي‌كنند ، عمليات خواندن و نوشتن را تسهيل مي‌بخشند ، وضعيت مصرف كننده‌ها را كنترل مي‌كنند و وظايف نگهداري جريان‌ها را انجام مي‌دهند . برای مثال، یک کارگزار ممکن است محتویات یک جریان را وقتی مملو از رکوردها باشد، کوتاه کند.

 **Consumers read from streams and react to the receipt of records.** A reaction to an event might entail some side-effect; for example, a consumer might persist an entry into its local database — reconstructing the state of a remote entity from its published “update” events.

مصرف كنندگان از جريان ها مي‌خوانند و به دريافت سوابق واكنش نشان مي‌دهند . يك واكنش به يك رويداد ممكن است داراي برخي عوارض جانبي باشد :براي مثال ، يك مصرف كننده ورود به ديتابيس محلي خود را ادامه دهد –

 **Consumers and producers may overlap**; for example, where the reaction to an event might be the production of one or more derived events.

مصرف كنندگان و تولييد كنندگان ممكن است همپوشاني داشته باشند ؛ براي مثال ، برای مثال، جایی که واکنش به یک رویداد ممکن است تولید یک یا چند رویداد مشتق شده باشد.

**جداسازی از طریق ناهمزمانی و عمومیت**

با چرخش به جایی که شروع کردیم، چرا EDA منجر به کاهش قابل توجه سطح کوپلینگ می شود؟

One pragmatic definition of **coupling is the degree to which a component is impacted by others**. Coupling exists in both *space* — whereby components are structurally related, and in *time* — where the notion of time affects the extent of their relationship. A good example of the latter is where one service synchronously invokes another’s REST API. If the called service is down, the callee usually cannot proceed — it is blocked on the response. If both services must be operational at the same time, then there is a degree of *temporal coupling* between them. We say that components are *tightly coupled* if there is a strong interdependency between them, and *loosely coupled* otherwise.

یکی از تعریف‌های عمل‌گرایانه از جفت، میزان تأثیرگذاری یک جزء توسط دیگران است. دوگانگي در هر دو فضا وجود دارد \_ به موجب آن اجزاء از نظر ساختاري به هم مرتبط هستند ، و در زمان \_ جايي كه مفهوم كلمه زمان بر روابط بين آنها تاثير مي‌گذارد . يك مثال خوب از دومي جايي است كه يك سرويس يك REST API را فراخواني مي‌کند . اگر سرویس فراخوانی شده از کار بیفتد ، سرویس فراخوان معمولا نمی‌توانند پردازش شود \_ بلکه درپاسخ بلاک می‌شود . اگر هر دوی سرویس ها در یک زمان یکسان عملیاتی شوند ، سپس درجه ای از جفت شدن زمانی بین آنها پدید می‌آید . ما می گوییم که اگر یک وابستگی متقابل قوی بین آنها وجود داشته باشد، مؤلفه ها به طور محکم جفت می شوند، و در غیر این صورت به طور ضعیف جفت می شوند.

EDA یک رویکرد دو جانبه برای مهار کوپلینگ اتخاذ می کند.

1- Recall, events are not communicated, *they just occur.* The component raising an event (by publishing a record) has no awareness of other components that may or may not exist. Therefore, the producer will not cease working if the consumers are unavailable — provided that the broker can durably buffer the events without imposing back-pressure upon the producer.

به یاد بیاورید که رویداد‌ها ارتباط برقرار نمی‌کنند ، آنها فقط اتفاق می‌افتند . یک جزء یک رویداد را که بالا می‌اورد (بوسیله انتشار یک رکورد ) از سایر اجزاء چه وجود داشته باشند چه نداشته باشند آگاهی ندارند . بنابراین تولید کننده اگر مصرف کننده ای وجود نداشته باشد کار را متوقف نمی‌کند \_ مشروط بر اینکه کارگزار بتواند رویدادها را بدون تحمیل فشار معکوس به تولیدکننده، به‌طور پایدار حفظ کند.

2-The persistence of event records on the broker largely eliminates the notion of time. A producer may publish an event at time *T1*, while a consumer may read it at *T2*, with *T1* and *T2* potentially being separated by milliseconds (if all is well) or hours (if some consumers are down or struggling).

تداوم ثبت رویدادها در کارگزار تا حد زیادی مفهوم زمان را از بین می برد . یک تولیید کننده ممکن است یک رویداد را درزمان T1 منتشر کنند ، زمانی که مصرف کننده آن را در زمان T2 بخواند به طور بلقوه فاصله بین T1 و T2 می‌تواند از چند میلی ثانیه تا چند ساعت به طول بیانجامد .

EDA is not a silver bullet. It does not eliminate the notion of coupling altogether — otherwise, components in the system would no longer function collectively. Our attention now turns to the broker: for producers and consumers to be meaningfully decoupled, they must instead rely on (and therefore couple themselves to) a broker. This adds complexity to the architecture of a system and introduces another point of failure. This is why **brokers must be highly performant and fault-tolerant**, otherwise we’ve just traded one set of problems for another.

معماری رویداد محور یک گلوله نقره ای نیست . به طور کلی مفوم جفت شدگی را از بین نمی‌برد \_ در غیر اینصورت اجزاء سیستم دیگر به طور جمعی کار نمی‌کنند . توجه ما الان به سمت کارگزار چرخش می‌کند : برای اینکه تولیید کنندگان و مصرف کنندگان به طور جدی از هم جدا شوند ، انها باید در عوض به یک کارگذار تکیه کنند . منجر به افزودگی پیچیدگی به معماری یک سیستم و ایجاد یک نقطه شکست دیگر می‌شود . به همین دلیل است که کارگزاران باید عملکرد بسیار بالایی داشته باشند و تحمل خطا داشته باشند، در غیر این صورت ما فقط یک سری از مشکلات را با دیگری مبادله کرده ایم.

سبک های پردازش رویداد

Event processing is generally categorized into three nominal styles. These styles are not mutually exclusive, often appearing together in large, event-driven systems.

پردازش رویداد‌ها به طور معمول از نظر اسمی به سه سبک تقیسیم بندی می‌شوند . این سبک‌ها دوبه دو ناسازگار نیستند ، اغلب در سیستم های بزرگ و رویداد محور با هم ظاهر می‌شوند .

پردازش رویداد گستره

The processing of an unbounded stream of related events, where event records appear in some order and are processed with some knowledge of past events. A good example might be the syndication of changes to a business entity. A consumer may apply these changes in a producer-prescribed order, to save a copy of the entity in its local database. Processing these change records discretely might not cut it, as *order matters*. Consumers also need to avoid race conditions, whereby multiple consumer instances might attempt to concurrently apply changes to the same record in a database, resulting in data inconsistencies due to out-of-order updates.

پردازش یک جریان نامحدود از رویدادهای مرتبط، که در آن سوابق رویداد به ترتیبی ظاهر می شوند و با آگاهی از رویدادهای گذشته پردازش می شوند. یک مثال خوب ممکن است پیوند تغییرات در یک نهاد تجاری باشد . یک مصرف کننده ممکن است این تغییرات را طبق دستور تولیید کننده اعمال کند تا یک نسخه از موجودیت را در پایگاه داده محلی خود ذخیره کند . پردازش این سوابق تغییر به طور مجزا ممکن است آن را قطع نکند، زیرا سفارش مهم است. مشتریها نیاز به پرهیز از شرایط رقابتی دارند، . مصرف کنندگان همچنین باید از شرایط مسابقه اجتناب کنند، به طوری که چندین نمونه مصرف کننده ممکن است سعی کنند به طور همزمان تغییراتی را در یک رکورد در یک پایگاه داده اعمال کنند، که منجر به ناسازگاری داده ها به دلیل به روز رسانی های خارج از دستور می شود.

Popular event streaming platforms like Kafka rely on record keying and partitions to preserve the order of updates. Kafka also guarantees that all changes to an entity are processed by one consumer instance, avoiding concurrency races that would result if multiple consumers were to naively process events in parallel.

پلتفرم های رویداد گسترده مشهوری مانند کافکا برای حفظ ترتیب بروزرسانی ها به کلید گذاری رکوردها و پارتیشن ها وابسته است . کافکا تضمین می‌کند که تمامی تغییرات در رابطه با یک موجودیت فقط توسط یک نمونه از مشتری پردازش شود ، و از رقابت‌های همزمانی که در نتیجه پردازش موازی که توسط چندین مشتری رخ می‌دهد جلوگیری می‌کند.

پردازش رویداد پیچیده

Complex event processing (CEP) derives or identifies complex event patterns from a series of simple events. An example of CEP might be monitoring a group of temperature and smoke sensors in a building to infer that a fire has broken out and to track its progress. Individual temperature changes might not be sufficient to raise an alert; however, the clustering of temperature spikes and the rate of change may provide more meaningful insights that could ultimately save lives.

پردازش رویداد پیچیده (CEP) الگوهای رویداد پیچیده را از یک سری رویدادهای ساده استخراج یا شناسایی می کند .یک مثال از پردازش رویداد پیچیده ممکن است که نظارت بر یک گروه از سنسورهای دما و دود در یک ساختمان برای پی بردن به وقوع آتش سوزی و پیشرفت آن باشد . تغییرات یک سنور دما برای نواختن زنگ هشدار کافی نیست ; با این حال، خوشه‌بندی نوسانات دما و نرخ تغییر ممکن است بینش‌های معناداری بیشتری ارائه دهد که در نهایت می‌تواند جان انسان‌ها را نجات دهد.

This sort of processing is usually more involved, requiring the event processor to keep track of prior events and provide an efficient way of querying and aggregating them.

این نوع پردازش معمولاً بیشتر درگیر است و به پردازشگر رویداد نیاز دارد تا رویدادهای قبلی را پیگیری کند و راهی کارآمد برای پرس و جو و جمع آوری آنها ارائه دهد.

چه زمانی از معماری رویداد محور استفاده میکنیم

چندین مورد استفاده وجود دارد که با قدرت معماری رویداد محور بازی می کند:

1. **Opaque consumer ecosystem.** Cases where producers are generally unaware of consumers. The latter might even be ephemeral processes that could come and go with short notice!

1- اکوسیستم مصرف کننده مات. مواردی که تولیدکنندگان عموماً از مصرف کنندگان بی اطلاع هستند. مورد دوم حتی ممکن است فرآیندهای زودگذری باشند که می توانند با اطلاع کوتاهی بیایند و بروند!

1.  **High fan-out.** Scenarios where one event might be processed by multiple, diverse consumers.

2- • فن اوت بالا. سناریوهایی که ممکن است یک رویداد توسط چندین مصرف کننده مختلف پردازش شود.

1.  **Complex pattern matching.** Where events might be strung together to infer more complex events.

3- • تطبیق الگوی پیچیده. جایی که رویدادها ممکن است برای استنباط رویدادهای پیچیده تر در کنار هم قرار گیرند.

1.  **Command-query responsibility segregation**. CQRS is a pattern that separates read and update operations for a data store. Implementing CQRS can improve the scalability and resilience of applications, with some consistency trade-offs. This pattern is commonly associated with EDA.
2. • تفکیک مسئولیت پرس و جو فرمان. CQRS الگویی است که عملیات خواندن و به روز رسانی را برای ذخیره داده جدا می کند. پیاده‌سازی CQRS می‌تواند مقیاس‌پذیری و انعطاف‌پذیری برنامه‌ها را با برخی مبادلات سازگاری بهبود بخشد. این الگو معمولاً با EDA مرتبط است.

مزایای معماری رویداد محور

 **Buffering and fault-tolerance.** Events might be consumed at a different rate to their production and producers mustn’t slow down for the consumers to catch up.

بافرو تحمل خطا . رویداد ها ممکن است با نرخ متفاوتی نسبت به تولیدشان مصرف شود و تولیید کنندگان نباید سرعت خود را کمک کنند تا مصرف کنندگان عقب نمانند .

 **Decoupling of producers and consumers**, avoiding unwieldy point-to-point integrations. It’s easy to add new producers and consumers to the system. It’s also easy to change the implementations of producers and consumers, provided that the contracts/schemas constraining the event records are honoured.

جداسازی تولیید کنندگان و مصرف کنندگان ، اجتناب از ادغام نقطه به نقطه . افزودن تولییدکنندگان و مصرف کنندگان جدید به سیستم آسان است . تغییر درپیاده سازی تولییدکنندگان و مصرف کنندگان بسیار آسان است ، مشروط بر اینکه قراردادها/طرحواره هایی که سوابق رویداد را محدود می کنند رعایت شود.

 **Massive scalability.** It is often possible to partition event streams into unrelated substreams and process these in parallel. We can also scale the number of consumers to meet the load demands if the backlog of events grows. Platforms like Kafka enable the [processing of events in strict order while simultaneously allowing massive parallelism](https://codeburst.io/combining-strict-order-with-massive-parallelism-using-kafka-83dc1ec9be03) across the stream.

مقیاس پذیری زیاد . اغلب می توان جریان های رویداد را به زیرجریان های نامرتبط تقسیم کرد و آنها را به صورت موازی پردازش کرد. همچنین می‌توانیم تعداد مصرف‌کنندگان را برای برآورده کردن نیازهای بار، در صورت افزایش رویدادهای عقب‌افتاده، مقیاس کنیم. پلتفرم‌هایی مانند کافکا پردازش رویدادها را به ترتیب دقیق امکان‌پذیر می‌سازند و در عین حال امکان موازی‌سازی عظیم در سراسر جریان را فراهم می‌کنند.

معایب معماری EDA

 **Limited to asynchronous processing.** While EDA is a powerful pattern for decoupling systems, its application is limited to the asynchronous processing of events. EDA does not work well as a substitute for request-response interactions, where the initiator must wait for a response before continuing.

محدود به پردازش ناهمزمان . اگرچه معماری رویداد محور یک الگوی قوی برای جدا سازی سیستم هاست ، اما کاربرد آن به پردازش های رویداد های نا همزمان محدود می‌شود . EDA به خوبی به عنوان جایگزینی برای تعاملات درخواست-پاسخ کار نمی کند، جایی که آغازگر باید قبل از ادامه منتظر پاسخ باشد.

 **Introduces additional complexity.** Where traditional client-server and request-response style of computing involves just two parties, the adoption of EDA requires a third — a broker to mediate the interactions between producers and consumers.

پیچیدگی اضافی را معرفی می کند. در جایی که روش سنتی محاسبات مشتری-سرور و درخواست-پاسخ فقط دو طرف را شامل می شود، پذیرش EDA به سومی نیاز دارد - یک کارگزار برای میانجیگری تعاملات بین تولیدکنندگان و مصرف کنندگان.

 **Failure masking.** This is a peculiar one as it seems to run contrary to the grain of decoupling systems. When systems are tightly coupled, an error in one system tends to propagate quickly and is brought to the forefront of our attention, often in painful ways. In most cases, this is something we would like to avoid: the failure of one component should have as little effect as possible on the others. The flip side of failure masking is that it inadvertently conceals problems that should otherwise be brought to our attention. This is solved by adding real-time monitoring and logging to each event-driven component, but this comes with added complexity.

پوشش شکست. این یک مورد عجیب است زیرا به نظر می رسد برخلاف اصل سیستم های جداسازی است. هنگامی که سیستم‌ها به‌شدت جفت می‌شوند، یک خطا در یک سیستم به سرعت منتشر می‌شود و اغلب به روش‌های دردناکی در خط مقدم توجه ما قرار می‌گیرد. در بیشتر موارد، این چیزی است که ما دوست داریم از آن اجتناب کنیم: خرابی یک جزء باید کمترین تأثیر ممکن را بر سایرین داشته باشد. طرف دیگر پوشش شکست این است که به طور ناخواسته مشکلاتی را پنهان می کند که در غیر این صورت باید به ما توجه شود. این با افزودن نظارت و ثبت بی‌درنگ به هر مؤلفه رویداد محور حل می‌شود، اما این با پیچیدگی بیشتری همراه است.

چيزهايي كه راجب آن بايد مراقب بود .

EDA is not a panacea, and like any powerful tool, it is prone to misuse. The following list should not be read as the outright disadvantages of EDA, but more as a set of gotchas that prudent developers and architects should be aware of when designing and implementing event-driven systems.

معماري رويداد محور يك نوشدارو نيست و مانند هر ابزار قدرتمند ديگري مستعد استفاده نادرست است . فهرست زیر نباید به عنوان معایب آشکار EDA خوانده شود، بلکه بیشتر به عنوان مجموعه ای از مشکلات است که توسعه دهندگان و معماران محتاط باید هنگام طراحی و اجرای سیستم های رویداد محور از آن آگاه باشند.

 **Convoluted choreography.** With loosely coupled components, one can get into a situation where the architecture might resemble a Rube Goldburg machine, whereby the entire business logic is implemented as a series of side-effects that are disguised as events: one component might raise an event that triggers a response in another component that raises another event, triggers another component, and so forth. This style of interaction between components can quickly become difficult to understand and reason about.

رقص پیچیده. با مولفه‌هایی که به‌طور ضعیف جفت شده‌اند، می‌توان وارد موقعیتی شد که معماری ممکن است شبیه ماشین روب گلدبورگ باشد، که در آن کل منطق کسب‌وکار به‌عنوان مجموعه‌ای از اثرات جانبی که به‌عنوان رویدادها پنهان می‌شوند پیاده‌سازی می‌شوند: یک جزء ممکن است رویدادی را ایجاد کند که باعث ایجاد یک رویداد شود. پاسخ در مؤلفه دیگری که رویداد دیگری را ایجاد می کند، مؤلفه دیگری را تحریک می کند و غیره. درک و استدلال این سبک از تعامل بین اجزا به سرعت دشوار می شود.

 **Disguising commands as events.** An event is a pure depiction of something that has happened; it does not prescribe how the event should be handled. On the other hand, a **command is a direct instruction addressed to a specific component**. Because both commands and events are messages of sorts, it is easy to get carried away and misrepresent a command as an event.

پنهان کردن دستورات به عنوان رویداد. یک رویداد تصویری ناب از چیزی است که اتفاق افتاده است. نحوه رسیدگی به رویداد را تعیین نمی کند. از سوی دیگر، یک فرمان یک دستور مستقیم است که به یک جزء خاص خطاب می شود. از آنجایی که دستورات و رویدادها هر دو پیام هایی هستند، به راحتی می توان یک فرمان را به عنوان یک رویداد نادرست معرفی کرد.

 **Remaining agnostic of consumers.** Events should capture relevant attributes in a way that does not limit how those events may be processed. This is easier said than done. Sometimes we might be privy to more information that could, in theory, be added to an event record, but it’s not clear whether adding that information to the record is useful or if it just leads to useless bloat.

ناشناس ماندن مصرف کنندگان. رویدادها باید ویژگی‌های مرتبط را به گونه‌ای دربر گیرند که نحوه پردازش آن رویدادها را محدود نکند. گفتن این کار آسان تر از انجام آن است. گاهی اوقات ممکن است اطلاعات بیشتری را در اختیار داشته باشیم که در تئوری می‌تواند به یک رکورد رویداد اضافه شود، اما مشخص نیست که افزودن آن اطلاعات به رکورد مفید است یا اینکه منجر به نفخ بی‌فایده می‌شود.

نتيجه

The microservices architectural paradigm is one piece of the broader puzzle of building more maintainable, scalable and robust software systems. Microservices are terrific from a problem decomposition standpoint, but they leave a lot of prickly problems on the table; one such problem being coupling. A monolith haphazardly decomposed into a handful of microservices could actually leave you in a worse state compared to where you started. We even have a term for that: a “distributed monolith”.

پارادایم معماری میکروسرویس ها بخشی از پازل گسترده تر ساختن سیستم های نرم افزاری قابل نگهداری، مقیاس پذیر و قوی تر است. میکروسرویس ها از نقطه نظر تجزیه مشکل فوق العاده هستند، اما مشکلات خاردار زیادی را روی میز باقی می گذارند. یکی از این مشکلات جفت شدن است. یک مونولیت که به طور تصادفی به تعدادی میکروسرویس تجزیه می شود در واقع می تواند شما را در وضعیت بدتری در مقایسه با جایی که شروع کرده اید قرار دهد. ما حتی یک اصطلاح برای آن داریم: "یکپارچه توزیع شده".

برای کمک به تکمیل پازل و پرداختن به مسئله کوپلینگ، معماری رویداد محور را بررسی کردیم.

EDA is an effective tool for reducing coupling between the components of a system by modelling interactions using the concepts of *producers*, *consumers*, *events* and *streams*. An event represents an action of interest and may be published and consumed asynchronously by components who are not even aware of each other’s existence. EDA allows for components to operate and evolve independently. It is not a silver bullet to slay all demons, but where EDA is an appropriate choice, the benefits it brings significantly outweigh the cost of its adoption. It may be argued that EDA is an essential element of any successful microservices deployment.

EDA ابزاری موثر برای کاهش جفت بین اجزای یک سیستم با مدل سازی تعاملات با استفاده از مفاهیم تولیدکنندگان، مصرف کنندگان، رویدادها و جریان ها است. یک رویداد نشان دهنده یک عمل مورد علاقه است و ممکن است به طور ناهمزمان توسط اجزایی که حتی از وجود یکدیگر آگاه نیستند منتشر و مصرف شود. EDA به اجزا اجازه می دهد تا مستقل عمل کنند و تکامل یابند. کشتن همه شیاطین یک گلوله نقره ای نیست، اما در جایی که EDA انتخاب مناسبی است، مزایایی که به ارمغان می آورد به میزان قابل توجهی از هزینه پذیرش آن بیشتر است. ممکن است استدلال شود که EDA یک عنصر اساسی در استقرار موفقیت آمیز میکروسرویس است.

The event-driven architecture pattern is a popular distributed asynchronous architecture pattern used to produce highly scalable applications. It is also highly adaptable and can be used for small applications and as well as large, complex ones. The event-driven architecture is made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events.

الگوی معماری رویداد محور یک الگوی معماری ناهمزمان توزیع شده محبوب است که برای تولید برنامه های کاربردی بسیار مقیاس پذیر استفاده می شود. همچنين بسيار قابليت سازگاري دارد و ميتواند به خوبي براي نرم افزار هاي كوچك و همچنين نرم افزارهاي بسيار بزرگ و پيچيده اسفتاده شود . معماري رويداد محور از اجزاي پردازش رويداد جداشده و تك منظوره تشكيل شده است كه رويداد ها را به طور ناهمزمان دريافت و پردازش ميكنند .

The event-driven architecture pattern consists of two main topologies, the mediator and the broker. The mediator topology is commonly used when you need to orchestrate multiple steps within an event through a central mediator, whereas the broker topology is used when you want to chain events together without the use of a central mediator. Because the architecture characteristics and implementation strategies differ between these two topologies, it is important to understand each one to know which is best suited for your particular situation.

الگوي معماري رويداد محور از دو توپولوژي اصلي ، ميانجي و واسط تشكيل شده است . توپولوژی واسطه معمولاً زمانی استفاده می شود که شما نیاز دارید چندین مرحله را در یک رویداد از طریق یک واسطه مرکزی هماهنگ کنید، در حالی که توپولوژی واسطه زمانی استفاده می شود که می خواهید رویدادها را بدون استفاده از یک واسطه مرکزی به یکدیگر زنجیره بزنید. از آنجایی که ویژگی‌های معماری و استراتژی‌های پیاده‌سازی بین این دو توپولوژی متفاوت است، مهم است که هر کدام را درک کنید تا بدانید کدام یک برای موقعیت خاص شما مناسب‌تر است.

**Mediator Topology**

توپولوژي مديديتور

The mediator topology is useful for events that have multiple steps and require some level of orchestration to process the event. For example, a single event to place a stock trade might require you to first validate the trade, then check the compliance of that stock trade against various compliance rules, assign the trade to a broker, calculate the commission, and finally place the trade with that broker. All of these steps would require some level of orchestration to determine the order of the steps and which ones can be done serially and in parallel.

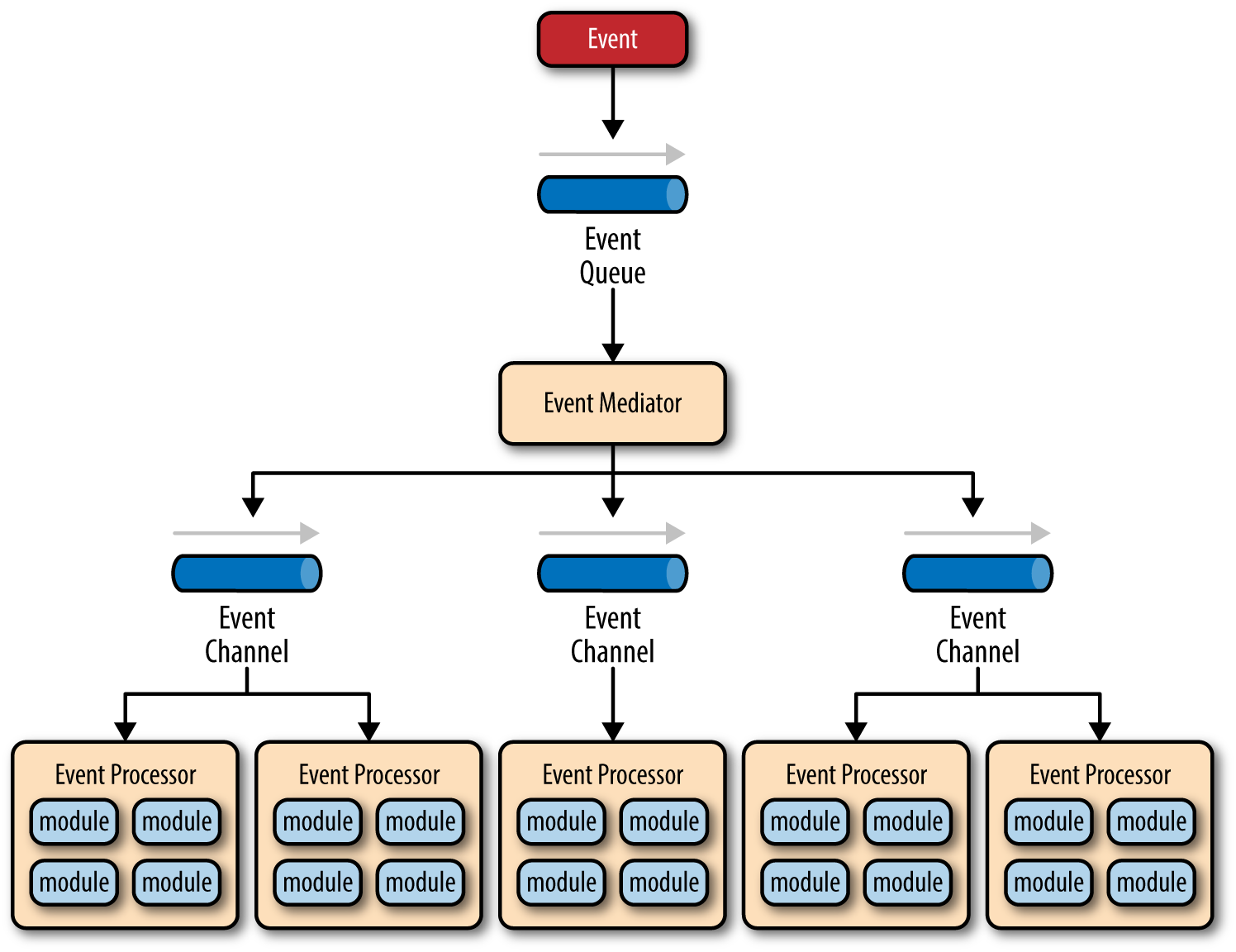
توپولوژی واسطه برای رویدادهایی مفید است که دارای مراحل متعدد هستند و برای پردازش رویداد نیاز به سطحی از هماهنگی دارند. به عنوان مثال، یک رویداد واحد برای انجام معامله سهام ممکن است از شما بخواهد که ابتدا معامله را تأیید کنید، سپس مطابقت آن معامله سهام را با قوانین انطباق مختلف بررسی کنید، معامله را به یک کارگزار واگذار کنید، کمیسیون را محاسبه کنید، و در نهایت معامله را با آن دلال

انجام دهيد . همه این مراحل به سطحی از هماهنگي نیاز دارند تا ترتیب مراحل را مشخص کند و اینکه کدام یک را می توان به صورت سریالی و موازی انجام داد.

There are four main types of architecture components within the mediator topology: event queues, an event mediator, event channels, and event processors. The event flow starts with a client sending an event to an *event queue*, which is used to transport the event to the event mediator. The *event mediator* receives the initial event and orchestrates that event by sending additional asynchronous events to *event channels* to execute each step of the process. *Event processors*, which listen on the event channels, receive the event from the event mediator and execute specific business logic to process the event. [Figure 2-1](https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html#sapr_0201_img) illustrates the general mediator topology of the event-driven architecture pattern.

چهار نوع اصلی از اجزای معماری در توپولوژی میانجی وجود دارد: صف‌های رویداد، میانجی رویداد، کانال‌های رویداد و پردازشگرهای رویداد. جریان رویداد با ارسال یک رویداد به یک صف رویداد توسط مشتری شروع می شود که برای انتقال رویداد به میانجی رویداد استفاده می شود. واسطه رویداد، رویداد اولیه را دریافت می‌کند و با ارسال رویدادهای ناهمزمان اضافی به کانال‌های رویداد برای اجرای هر مرحله از فرآیند، آن رویداد را هماهنگ می‌کند. پردازنده‌های رویداد که در کانال‌های رویداد گوش می‌دهند، رویداد را از میانجی رویداد دریافت می‌کنند و منطق تجاری خاصی را برای پردازش رویداد اجرا می‌کنند.

شکل 2-1 توپولوژی میانجی کلی الگوی معماری رویداد محور را نشان می دهد



It is common to have anywhere from a dozen to several hundred event queues in an event-driven architecture. The pattern does not specify the implementation of the event queue component; it can be a message queue, a web service endpoint, or any combination thereof.

وجود یک دوجین تا چند صد صف رویداد در یک معماری رویداد محور معمول است. الگو اجرای جزء صف رویداد را مشخص نمی کند. این می تواند یک صف پیام، یک نقطه پایانی وب سرویس یا هر ترکیبی از آنها باشد.

There are two types of events within this pattern: an *initial event* and a *processing event*. The initial event is the original event received by the mediator, whereas the processing events are ones that are generated by the mediator and received by the event-processing components.

دو نوع رویداد در این الگو وجود دارد: یک رویداد اولیه و یک رویداد پردازش. رویداد اولیه رویداد اصلی دریافت شده توسط میانجی است، در حالی که رویدادهای پردازش رویدادهایی هستند که توسط میانجی ایجاد می‌شوند و توسط اجزای پردازش رویداد دریافت می‌شوند.

The event-mediator component is responsible for orchestrating the steps contained within the initial event. For each step in the initial event, the event mediator sends out a specific processing event to an event channel, which is then received and processed by the event processor. It is important to note that the event mediator doesn’t actually perform the business logic necessary to process the initial event; rather, it knows of the steps required to process the initial event.

مؤلفه میانجی رویداد مسئول هماهنگ کردن مراحل موجود در رویداد اولیه است. برای هر مرحله در رویداد اولیه، واسطه رویداد یک رویداد پردازشی خاص را به یک کانال رویداد ارسال می‌کند، که سپس توسط پردازشگر رویداد دریافت و پردازش می‌شود. . توجه به این نکته مهم است که واسطه رویداد در واقع منطق تجاری لازم برای پردازش رویداد اولیه را انجام نمی دهد. بلکه از مراحل مورد نیاز برای پردازش رویداد اولیه می داند.

Event channels are used by the event mediator to asynchronously pass specific processing events related to each step in the initial event to the event processors. The event channels can be either message queues or message topics, although message topics are most widely used with the mediator topology so that processing events can be processed by multiple event processors (each performing a different task based on the processing event received).

کانال‌های رویداد توسط واسطه رویداد برای انتقال ناهمزمان رویدادهای پردازش خاص مربوط به هر مرحله در رویداد اولیه به پردازنده‌های رویداد استفاده می‌شوند. کانال‌های رویداد می‌توانند صف‌های پیام یا موضوعات پیام باشند، اگرچه موضوعات پیام بیشتر با توپولوژی واسطه استفاده می‌شوند، به طوری که رویدادهای پردازش می‌توانند توسط چندین پردازشگر رویداد (هر کدام وظیفه متفاوتی را بر اساس رویداد پردازش دریافتی انجام می‌دهند) پردازش کنند.

The event processor components contain the application business logic necessary to process the processing event. Event processors are self-contained, independent, highly decoupled architecture components that perform a specific task in the application or system. While the granularity of the event-processor component can vary from fine-grained (e.g., calculate sales tax on an order) to coarse-grained (e.g., process an insurance claim), it is important to keep in mind that in general, each event-processor component should perform a single business task and not rely on other event processors to complete its specific task.

اجزای پردازشگر رویداد حاوی منطق تجاری برنامه کاربردی است که برای پردازش رویداد پردازشی ضروری است. پردازنده‌های رویداد اجزای معماری مستقل، مستقل و بسیار جداشده‌ای هستند که وظیفه خاصی را در برنامه یا سیستم انجام می‌دهند. در حالی که جزئیات مولفه رویداد-پردازنده می تواند از ریزدانه (مثلاً محاسبه مالیات فروش بر روی یک سفارش) تا درشت دانه (مثلاً رسیدگی به ادعای بیمه) متفاوت باشد، مهم است که به خاطر داشته باشید که به طور کلی، هر جزء رویداد-پردازنده باید یک کار تجاری واحد را انجام دهد و برای تکمیل کار خاص خود به سایر پردازنده های رویداد متکی نباشد.

The event mediator can be implemented in a variety of ways. As an architect, you should understand each of these implementation options to ensure that the solution you choose for the event mediator matches your needs and requirements.

میانجی رویداد را می توان به روش های مختلفی اجرا کرد. به عنوان یک معمار، باید هر یک از این گزینه های پیاده سازی را درک کنید تا مطمئن شوید که راه حلی که برای میانجی رویداد انتخاب می کنید با نیازها و نیازهای شما مطابقت دارد.

The simplest and most common implementation of the event mediator is through open source integration hubs such as Spring Integration, Apache Camel, or Mule ESB. Event flows in these open source integration hubs are typically implemented through Java code or a DSL (domain-specific language). For more sophisticated mediation and orchestration, you can use BPEL (business process execution language) coupled with a BPEL engine such as the open source Apache ODE. BPEL is a standard XML-like language that describes the data and steps required for processing an initial event. For very large applications requiring much more sophisticated orchestration (including steps involving human interactions), you can implement the event mediator using a business process manager (BPM) such as jBPM.

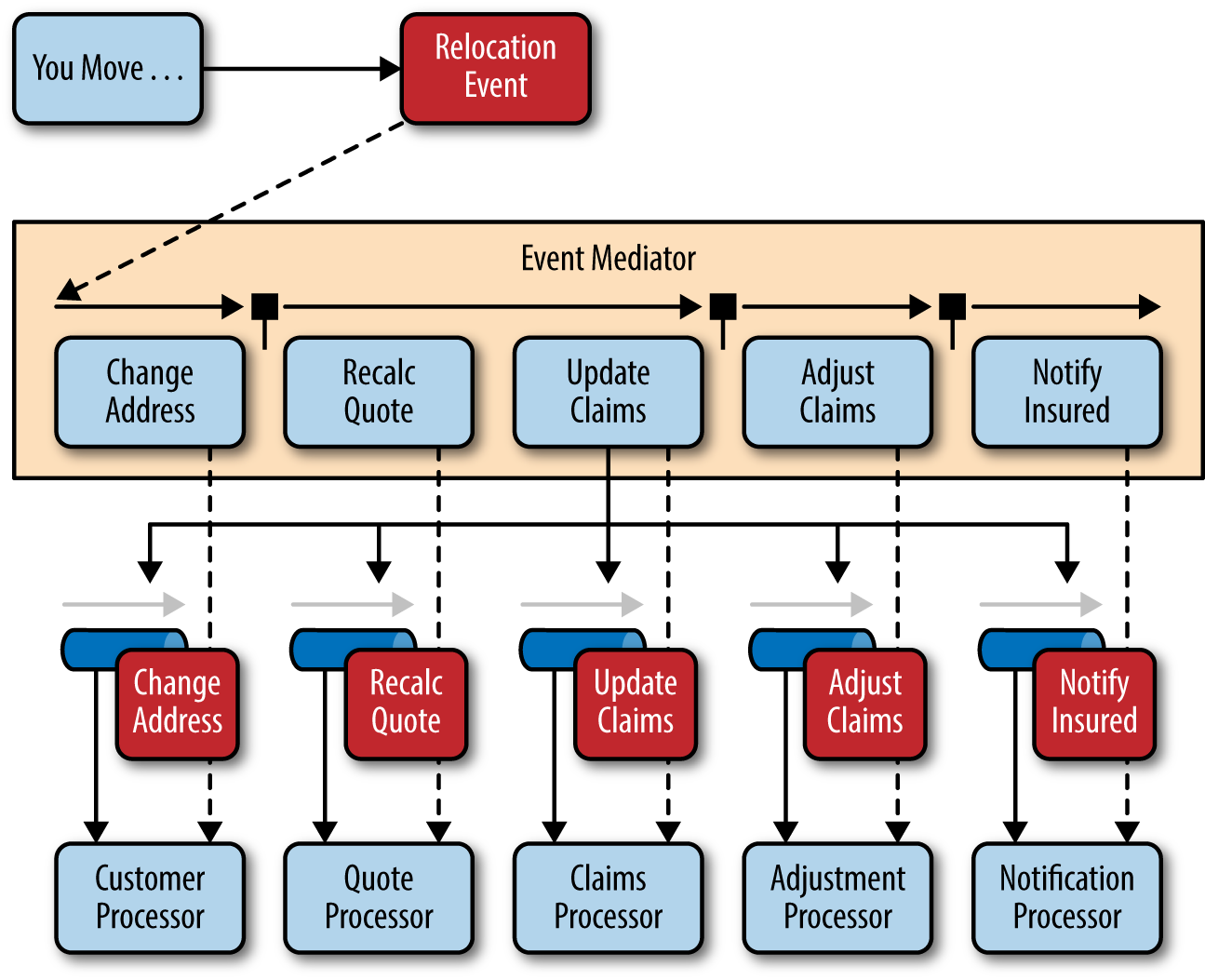
ساده ترین و رایج ترین پیاده سازی میانجی رویداد از طریق هاب های یکپارچه سازی منبع باز مانند Spring Integration، Apache Camel یا Mule ESB است. . جریان رویداد در این هاب های یکپارچه منبع باز معمولاً از طریق کد جاوا یا یک DSL (زبان مخصوص دامنه) پیاده سازی می شود. برای میانجی‌گری و هماهنگ‌سازی پیچیده‌تر، می‌توانید از BPEL (زبان اجرای فرآیند تجاری) همراه با موتور BPEL مانند منبع باز Apache ODE استفاده کنید. BPEL یک زبان استاندارد XML است که داده ها و مراحل مورد نیاز برای پردازش یک رویداد اولیه را توصیف می کند . . برای برنامه های بسیار بزرگ که به هماهنگی بسیار پیچیده تری نیاز دارند (از جمله مراحل مربوط به تعاملات انسانی)، می توانید میانجی رویداد را با استفاده از مدیر فرآیند کسب و کار (BPM) مانند jBPM پیاده سازی کنید.

Understanding your needs and matching them to the correct event mediator implementation is critical to the success of any event-driven architecture using this topology. Using an open source integration hub to do very complex business process management orchestration is a recipe for failure, just as is implementing a BPM solution to perform simple routing logic.

درک نیازهای شما و تطبیق آنها با اجرای صحیح میانجی رویداد برای موفقیت هر معماری رویداد محور با استفاده از این توپولوژی حیاتی است. استفاده از یک هاب یکپارچه سازی منبع باز برای انجام هماهنگی مدیریت فرآیندهای تجاری بسیار پیچیده، دستوری برای شکست است، درست مانند پیاده سازی یک راه حل BPM برای اجرای منطق مسیریابی ساده.

To illustrate how the mediator topology works, suppose you are insured through an insurance company and you decide to move. In this case, the initial event might be called something like *relocation event*. The steps involved in processing a *relocation event* are contained within the event mediator as shown in [Figure 2-2](https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html#sapr_0202_img). For each initial event step, the event mediator creates a processing event (e.g., *change address*, *recalc quote*, etc.), sends that processing event to the event channel and waits for the processing event to be processed by the corresponding event processor (e.g., customer process, quote process, etc.). This process continues until all of the steps in the initial event have been processed. The single bar over the recalc quote and update claims steps in the event mediator indicates that these steps can be run at the same time.

برای نشان دادن نحوه عملکرد توپولوژی واسطه، فرض کنید از طریق یک شرکت بیمه بیمه شده اید و تصمیم به نقل مکان دارید. در این مورد، رویداد اولیه ممکن است چیزی شبیه رویداد جابجایی نامیده شود. مراحل دخیل در پردازش یک رویداد جابجایی در میانجی رویداد وجود دارد همانطور که در شکل 2-2 نشان داده شده است. برای هر مرحله اولیه رویداد، واسطه رویداد یک رویداد پردازشی ایجاد می‌کند (مثلاً تغییر آدرس، نقل قول مجدد محاسبه، و غیره)، آن رویداد پردازشی را به کانال رویداد ارسال می‌کند و منتظر می‌ماند تا رویداد پردازش توسط پردازشگر رویداد مربوطه پردازش شود (مثلاً ، فرآیند مشتری، فرآیند نقل قول و غیره). این فرآیند تا زمانی ادامه می یابد که تمام مراحل در رویداد اولیه پردازش شوند. نوار تکی روی مراحل نقل قول مجدد و ادعاهای به روز رسانی در میانجی رویداد نشان می دهد که این مراحل می توانند همزمان اجرا شوند.



**Broker Topology**

The broker topology differs from the mediator topology in that there is no central event mediator; rather, the message flow is distributed across the event processor components in a chain-like fashion through a lightweight message broker (e.g., ActiveMQ, HornetQ, etc.). This topology is useful when you have a relatively simple event processing flow and you do not want (or need) central event orchestration.

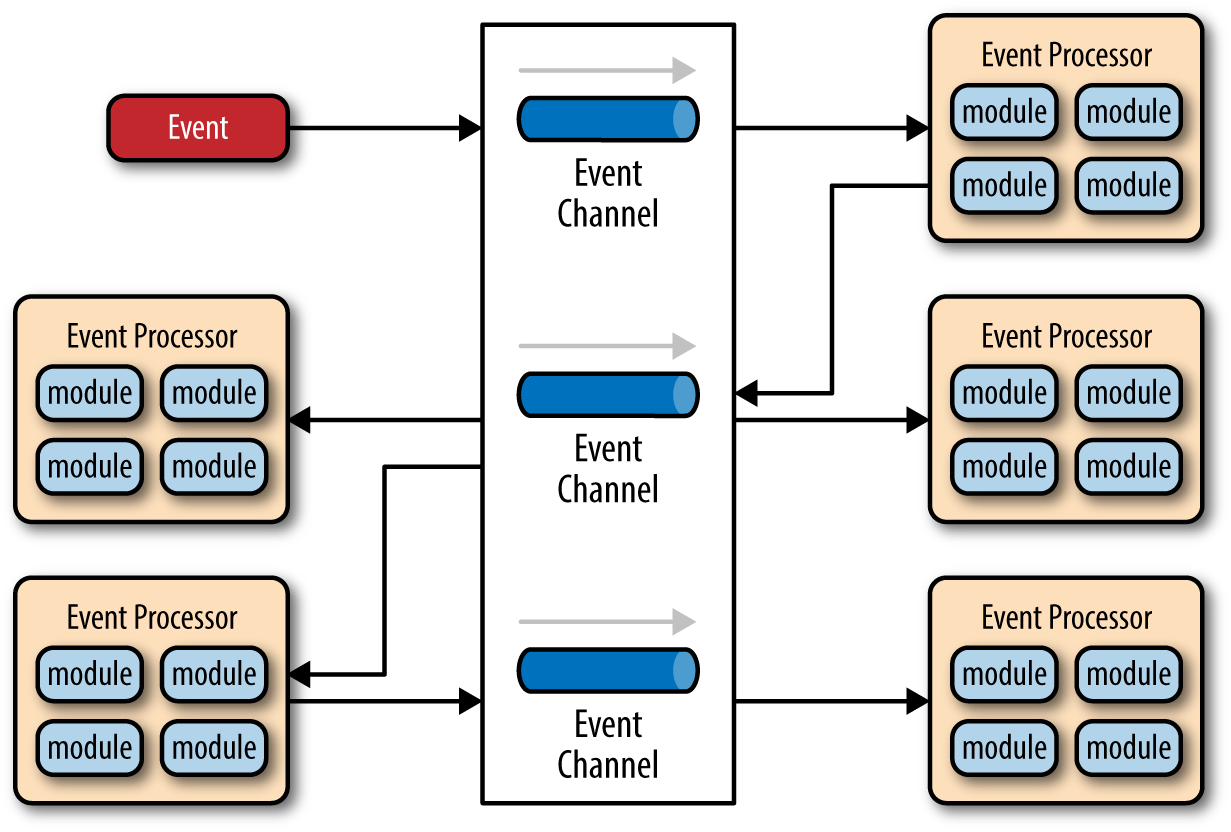
توپولوژی بروکر با توپولوژی واسطه تفاوت دارد زیرا هیچ واسطه رویداد مرکزی وجود ندارد. در عوض، جریان پیام در اجزای پردازشگر رویداد به شکلی زنجیره‌وار از طریق یک واسطه پیام سبک (مانند ActiveMQ، HornetQ و غیره) توزیع می‌شود. این توپولوژی زمانی مفید است که شما یک جریان پردازش رویداد نسبتا ساده دارید و نمی خواهید (یا نیازی به) هماهنگی رویداد مرکزی داشته باشید.

There are two main types of architecture components within the broker topology: a *broker* component and an *event processor* component. The broker component can be centralized or federated and contains all of the event channels that are used within the event flow. The event channels contained within the broker component can be message queues, message topics, or a combination of both.

دو نوع اصلی از اجزای معماری در توپولوژی کارگزار وجود دارد: یک جزء کارگزار و یک جزء پردازشگر رویداد. جزء بروکر می تواند متمرکز یا فدرال باشد و شامل تمام کانال های رویدادی است که در جریان رویداد استفاده می شوند. کانال‌های رویداد موجود در مؤلفه کارگزار می‌توانند صف‌های پیام، موضوعات پیام یا ترکیبی از هر دو باشند.

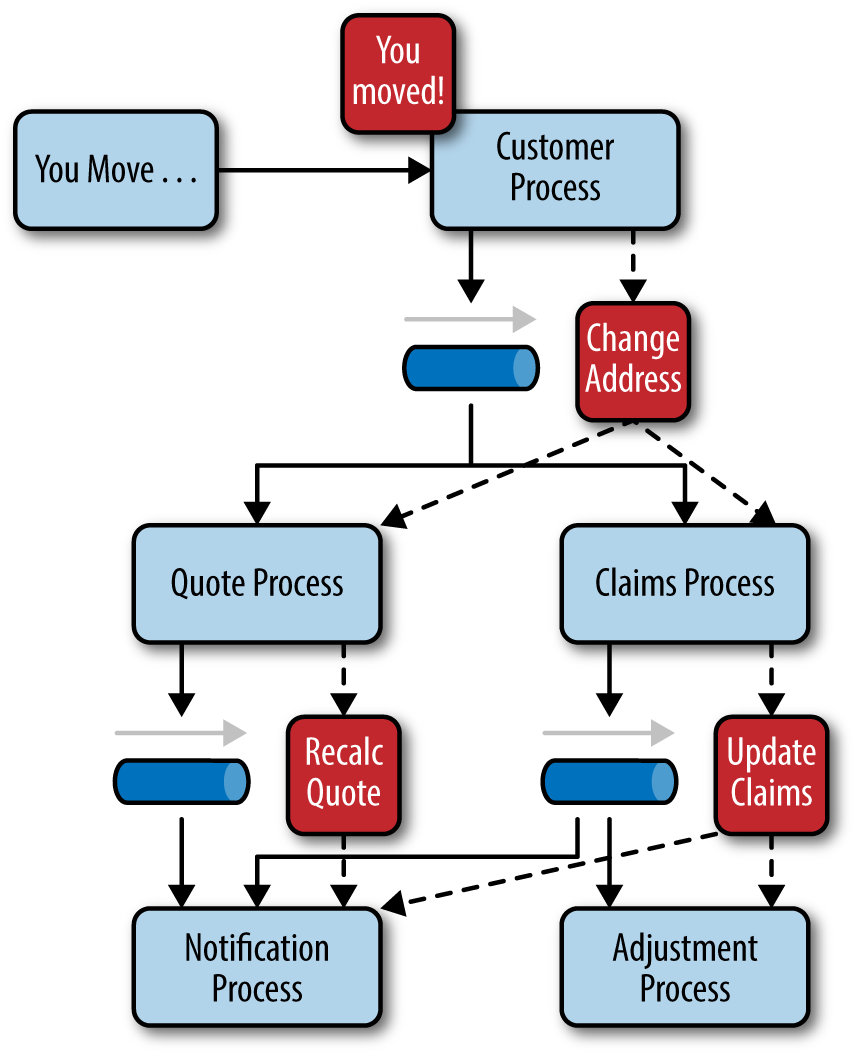
This topology is illustrated in [Figure 2-3](https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html#sapr_0203_img). As you can see from the diagram, there is no central event-mediator component controlling and orchestrating the initial event; rather, each event-processor component is responsible for processing an event and publishing a new event indicating the action it just performed. For example, an event processor that balances a portfolio of stocks may receive an initial event called *stock split*. Based on that initial event, the event processor may do some portfolio rebalancing, and then publish a new event to the broker called *rebalance portfolio*, which would then be picked up by a different event processor. Note that there may be times when an event is published by an event processor but not picked up by any another event processor. This is common when you are evolving an application or providing for future functionality and extensions.

این توپولوژی در شکل 2-3 نشان داده شده است. همانطور که از نمودار می بینید، هیچ جزء میانجی رویداد مرکزی که رویداد اولیه را کنترل و هماهنگ کند وجود ندارد. در عوض، هر جزء رویداد-پردازنده مسئول پردازش یک رویداد و انتشار یک رویداد جدید است که نشان دهنده عملکردی است که به تازگی انجام داده است. به عنوان مثال، یک پردازشگر رویداد که یک سبد سهام را متعادل می کند ممکن است یک رویداد اولیه به نام تقسیم سهام را دریافت کند. بر اساس آن رویداد اولیه، پردازشگر رویداد ممکن است مقداری توازن مجدد پورتفولیو را انجام دهد و سپس یک رویداد جدید به نام rebalance portfolio را برای کارگزار منتشر کند، که سپس توسط یک پردازشگر رویداد دیگر انتخاب می‌شود. توجه داشته باشید که ممکن است زمان‌هایی وجود داشته باشد که یک رویداد توسط یک پردازشگر رویداد منتشر شود اما توسط هیچ پردازشگر رویداد دیگری انتخاب نشود. هنگامی که شما در حال توسعه یک برنامه یا ارائه عملکردها و برنامه های افزودنی آینده هستید، این امر رایج است.



To illustrate how the broker topology works, we’ll use the same example as in the mediator topology (an insured person moves). Since there is no central event mediator to receive the initial event in the broker topology, the customer-process component receives the event directly, changes the customer address, and sends out an event saying it changed a customer’s address (e.g., *change address*event*)*. In this example, there are two event processors that are interested in the *change address* event: the quote process and the claims process. The quote processor component recalculates the new auto-insurance rates based on the address change and publishes an event to the rest of the system indicating what it did (e.g., *recalc quote* event). The claims processing component, on the other hand, receives the same *change address* event, but in this case, it updates an outstanding insurance claim and publishes an event to the system as an *update claim* event. These new events are then picked up by other event processor components, and the event chain continues through the system until there are no more events are published for that particular initiating event.

برای توضیح نحوه عملکرد توپولوژی کارگزار، از همان مثالی که در توپولوژی واسطه (یک شخص بیمه شده حرکت می کند) استفاده می کنیم. از آنجایی که هیچ واسطه رویداد مرکزی برای دریافت رویداد اولیه در توپولوژی کارگزار وجود ندارد، مؤلفه فرآیند مشتری رویداد را مستقیماً دریافت می‌کند، آدرس مشتری را تغییر می‌دهد و رویدادی را ارسال می‌کند که نشانی مشتری را تغییر داده است (به عنوان مثال، تغییر رویداد آدرس) . در این مثال، دو پردازشگر رویداد وجود دارد که به رویداد آدرس تغییر علاقه مند هستند: فرآیند نقل قول و فرآیند ادعا. جزء پردازشگر مظنه نرخ‌های جدید بیمه خودکار را بر اساس تغییر آدرس مجدداً محاسبه می‌کند و رویدادی را برای بقیه سیستم منتشر می‌کند که نشان می‌دهد چه کاری انجام داده است (به عنوان مثال، رویداد نقل قول مجدد محاسبه شود). از طرف دیگر، مؤلفه رسیدگی به خسارت، همان رویداد آدرس تغییر را دریافت می‌کند، اما در این مورد، ادعای بیمه معوق را به‌روزرسانی می‌کند و رویدادی را به عنوان رویداد ادعای به‌روزرسانی در سیستم منتشر می‌کند. این رویدادهای جدید سپس توسط سایر اجزای پردازشگر رویداد انتخاب می‌شوند و زنجیره رویداد از طریق سیستم ادامه می‌یابد تا زمانی که هیچ رویداد دیگری برای آن رویداد آغازگر خاص منتشر نشود.



As you can see from [Figure 2-4](https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html#sapr_0204_img), the broker topology is all about the chaining of events to perform a business function. The best way to understand the broker topology is to think about it as a relay race. In a relay race, runners hold a baton and run for a certain distance, then hand off the baton to the next runner, and so on down the chain until the last runner crosses the finish line. In relay races, once a runner hands off the baton, she is done with the race. This is also true with the broker topology: once an event processor hands off the event, it is no longer involved with the processing of that specific event.

همانطور که از شکل 2-4 می بینید، توپولوژی کارگزار همه چیز در مورد زنجیره رویدادها برای انجام یک عملکرد تجاری است. بهترین راه برای درک توپولوژی بروکر این است که در مورد آن به عنوان یک مسابقه رله فکر کنید. در مسابقه امدادی، دوندگان یک باتوم را در دست می گیرند و برای یک مسافت مشخص می دوند، سپس باتوم را به دونده بعدی تحویل می دهند و به همین ترتیب زنجیره را ادامه می دهند تا آخرین دونده از خط پایان عبور کند. در مسابقات رله، زمانی که یک دونده باتوم را از دست می دهد، مسابقه را تمام می کند. این در مورد توپولوژی بروکر نیز صادق است: هنگامی که یک پردازشگر رویداد رویداد را متوقف می کند، دیگر درگیر پردازش آن رویداد خاص نیست.

ملاحظات

The event-driven architecture pattern is a relatively complex pattern to implement, primarily due to its asynchronous distributed nature. When implementing this pattern, you must address various distributed architecture issues, such as remote process availability, lack of responsiveness, and broker reconnection logic in the event of a broker or mediator failure.

الگوی معماری رویداد محور یک الگوی نسبتاً پیچیده برای پیاده سازی است، در درجه اول به دلیل ماهیت توزیع ناهمزمان آن. هنگام اجرای این الگو، باید به مسائل مختلف معماری توزیع شده، مانند در دسترس بودن فرآیند از راه دور، عدم پاسخگویی و منطق اتصال مجدد کارگزار در صورت خرابی کارگزار یا میانجی رسیدگی کنید.

One consideration to take into account when choosing this architecture pattern is the lack of atomic transactions for a single business process. Because event processor components are highly decoupled and distributed, it is very difficult to maintain a transactional unit of work across them. For this reason, when designing your application using this pattern, you must continuously think about which events can and can’t run independently and plan the granularity of your event processors accordingly. If you find that you need to split a single unit of work across event processors—that is, if you are using separate processors for something that should be an undivided transaction—this is probably not the right pattern for your application.

یکی از ملاحظاتی که باید در انتخاب این الگوی معماری در نظر گرفت، عدم وجود معاملات اتمی برای یک فرآیند تجاری واحد است. از آنجایی که اجزای پردازشگر رویداد به شدت جدا و توزیع شده اند، حفظ یک واحد تراکنشی کار در بین آنها بسیار دشوار است. به همین دلیل، هنگام طراحی برنامه خود با استفاده از این الگو، باید به طور مداوم به این فکر کنید که کدام رویدادها می توانند و کدام رویدادها نمی توانند به طور مستقل اجرا شوند و بر اساس آن، جزئیات پردازشگرهای رویداد خود را برنامه ریزی کنید. اگر متوجه شدید که باید یک واحد کار را بین پردازنده‌های رویداد تقسیم کنید - یعنی اگر از پردازنده‌های جداگانه برای چیزی استفاده می‌کنید که باید یک تراکنش تقسیم نشده باشد - احتمالاً این الگوی مناسب برای برنامه شما نیست.

Perhaps one of the most difficult aspects of the event-driven architecture pattern is the creation, maintenance, and governance of the event-processor component contracts. Each event usually has a specific contract associated with it (e.g., the data values and data format being passed to the event processor). It is vitally important when using this pattern to settle on a standard data format (e.g., XML, JSON, Java Object, etc.) and establish a contract versioning policy right from the start

شاید یکی از سخت‌ترین جنبه‌های الگوی معماری رویداد محور، ایجاد، نگهداری و حاکمیت قراردادهای اجزای رویداد-پردازنده باشد. هر رویداد معمولاً یک قرارداد خاص مرتبط با خود دارد (به عنوان مثال، مقادیر داده و قالب داده به پردازشگر رویداد منتقل می شود). هنگام استفاده از این الگو، تعیین یک قالب داده استاندارد (مانند XML، JSON، Java Object، و غیره) و ایجاد یک خط مشی نسخه‌سازی قرارداد از همان ابتدا بسیار مهم است.

تحليل و اناليز

جدول زیر شامل رتبه‌بندی و تحلیل ویژگی‌های معماری رایج برای الگوی معماری رویداد محور است. درجه بندی برای هر مشخصه بر اساس تمایل طبیعی برای آن مشخصه به عنوان یک قابلیت مبتنی بر اجرای معمولی الگو، و همچنین آنچه که الگو به طور کلی برای آن شناخته شده است، است. برای مقایسه کنار هم در مورد چگونگی ارتباط این الگو با سایر الگوهای این گزارش، لطفاً به پیوست A در انتهای این گزارش مراجعه کنید.

**Overall agility چابكي كلي**

رتبه بندي :بالا

تحليل: چابکی کلی توانایی واکنش سریع به یک محیط دائما در حال تغییر است. از آنجایی که اجزای رویداد-پردازنده تک منظوره هستند و به طور کامل از سایر اجزای پردازشگر رویداد جدا شده اند، تغییرات به طور کلی به یک یا چند پردازنده رویداد جدا می شوند و می توانند به سرعت بدون تأثیر بر سایر مؤلفه ها انجام شوند.

سهولت استقرار ease of deployment

رتبه بندي :بالا

Overall this pattern is relatively easy to deploy due to the decoupled nature of the event-processor components. The broker topology tends to be easier to deploy than the mediator topology, primarily because the event mediator component is somewhat tightly coupled to the event processors: a change in an event processor component might also require a change in the event mediator, requiring both to be deployed for any given change.

به طور کلی این الگو به دلیل ماهیت جداشده اجزای پردازشگر رویداد نسبتاً آسان است. استقرار توپولوژی بروکر نسبت به توپولوژی میانجی آسان‌تر است، در درجه اول به این دلیل که جزء میانجی رویداد تا حدودی با پردازنده‌های رویداد مرتبط است: تغییر در یک جزء پردازشگر رویداد ممکن است نیاز به تغییر در میانجی رویداد نیز داشته باشد، که نیاز به هر دوی آنها باشد. برای هر تغییری مستقر شده است.

**Testability قابليت تست**

**رتبه بندي : پايين**

While individual unit testing is not overly difficult, it does require some sort of specialized testing client or testing tool to generate events. Testing is also complicated by the asynchronous nature of this pattern.

در حالی که آزمایش واحدهای فردی بیش از حد دشوار نیست، به نوعی از مشتری تست تخصصی یا ابزار تست برای ایجاد رویدادها نیاز دارد. آزمایش نیز به دلیل ماهیت ناهمزمان این الگو پیچیده است.

كارايي (performance )

رتبه بندي:بالا

While it is certainly possible to implement an event-driven architecture that does not perform well due to all the messaging infrastructure involved, in general, the pattern achieves high performance through its asynchronous capabilities; in other words, the ability to perform decoupled, parallel asynchronous operations outweighs the cost of queuing and dequeuing messages.

در حالی که مطمئناً می توان یک معماری رویداد محور را پیاده سازی کرد که به دلیل همه زیرساخت های پیام رسانی درگیر، عملکرد خوبی ندارد، به طور کلی، این الگو از طریق قابلیت های ناهمزمان خود به عملکرد بالایی دست می یابد. به عبارت دیگر، توانایی انجام عملیات ناهمزمان جدا شده و موازی بر هزینه صف بندی و جداسازی پیام ها بیشتر است.

**Scalability مقياس پذيري**

رتبه بندي : بالا

Scalability is naturally achieved in this pattern through highly independent and decoupled event processors. Each event processor can be scaled separately, allowing for fine-grained scalability.

مقیاس پذیری به طور طبیعی در این الگو از طریق پردازنده های رویداد بسیار مستقل و جدا شده به دست می آید. هر پردازنده رویداد را می توان به طور جداگانه مقیاس بندی کرد، که امکان مقیاس پذیری دقیق را فراهم می کند.

سهولت توسعه ease of development

پايين

Development can be somewhat complicated due to the asynchronous nature of the pattern as well as contract creation and the need for more advanced error handling conditions within the code for unresponsive event processors and failed brokers.

به دلیل ماهیت ناهمزمان الگو و همچنین ایجاد قرارداد و نیاز به شرایط مدیریت خطای پیشرفته‌تر در کد برای پردازنده‌های رویدادی که پاسخگو نیستند و کارگزاران ناموفق، توسعه می‌تواند تا حدودی پیچیده باشد.