Introduction to Event-Driven Architecture

The essential concepts that every developer should know

Kacey Bui
Follow
Feb 10 · 11 min read

Our inaugural post on the Introduction to Microservices talked about the granularity of services and the need to ensure loose coupling. It was said that services should be autonomous, fully own their dependencies, and minimise synchronous communication. Today we are going to touch on what it means to be loosely coupled, and explore one handy trick of the trade that seems to be increasingly gaining traction in the microservices community — Event-Driven Architecture.

A Simple Definition

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

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.

The Event That Never Happened

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.

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:

```
{
  "orderId": "760b5301-295f-4fec-95f8-6b303a3b824a",
  "customerId": 28623823,
  "productId": 31334,
  "quantity": 1,
  "timestamp": "2021-02-09T11:12:17+0000"
}
```

***Note**: Despite their subtle differences, **records and events are often used interchangeably;** i.e., the term "event" is used to denote a "record" of that event. To make things easier, we'll permit ourselves the same liberty from here on in.*

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.

Channelling Events

If producers and consumers of events are unaware of each other, how do they communicate?

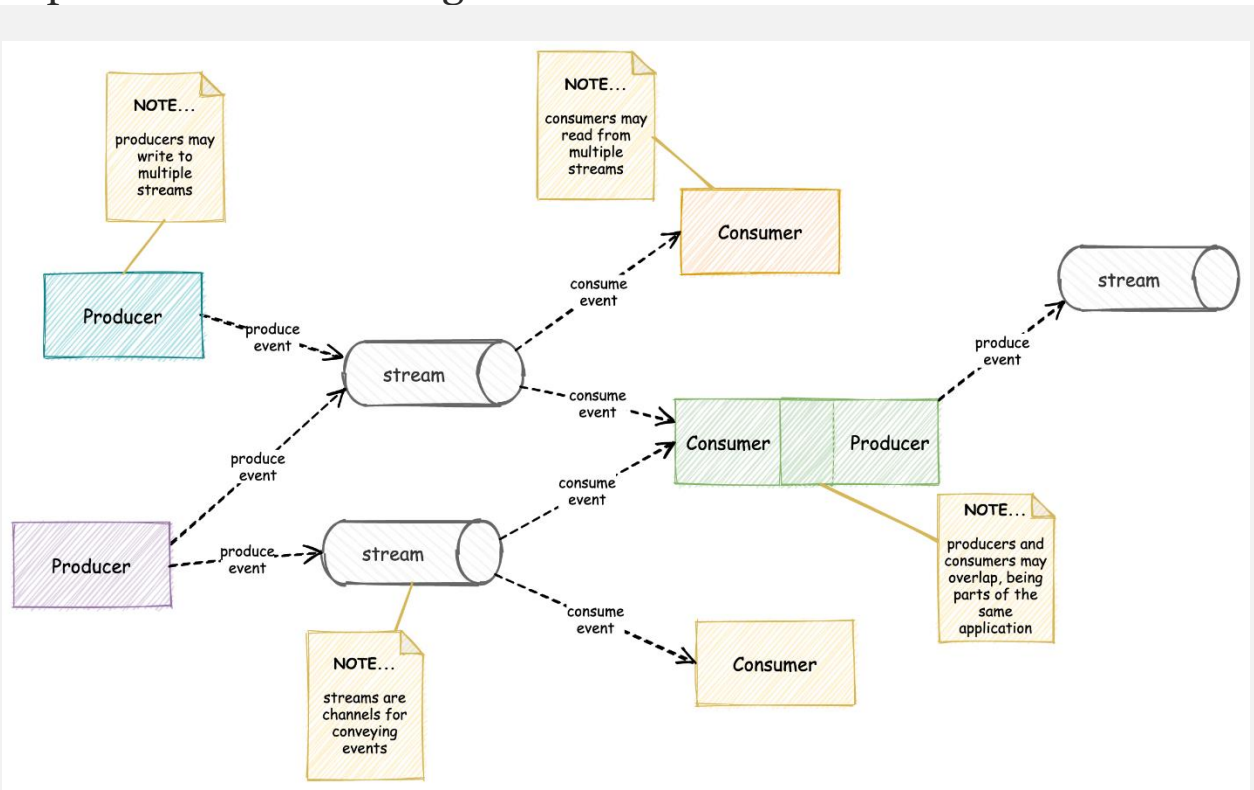
The clue is in the term "record". Events are usually persisted in a well-known location, called a **log**. (Sometimes, the term **ledger** may be used.) Logs are low-level, append-only data

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.

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 — by far the most popular event streaming platform in existence — describes streams in terms of **topics** and **partitions**.

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



Event-Driven Architecture Reference Model

A quick checkpoint to help cement our understanding:

- **Events are actions of interest that occur at discrete points in time** and may be externally observed and described.
- **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.
- **Producers are receptors that detect events by publishing corresponding records to a stream.**
- **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 decentralised ledger.
- **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.

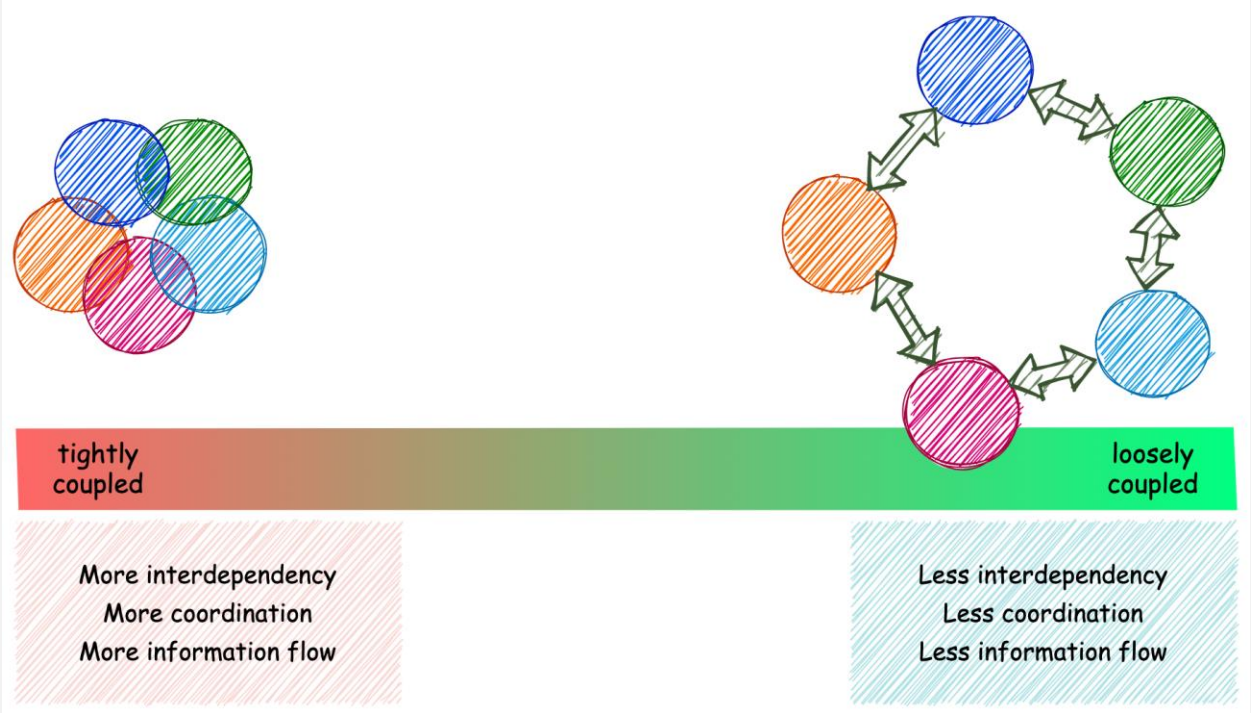
- **Consumers read from streams and react to the receipt of records.** A reaction to an event might entail some side-effect; for example, a consumer might persist an entry into its local database — reconstructing the state of a remote entity from its published "update" events.
- **Consumers and producers may overlap**; for example, where the reaction to an event might be the production of one or more derived events.

Decoupling Through Asynchrony & Generality

Circling back to where we started, *why does EDA lead to a significantly reduced level of coupling?*

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.



Conceptual model of coupling

EDA takes a two-pronged approach to curb coupling.

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.

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).

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.

Styles of Event Processing

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

Discrete event processing

The processing of discrete events; for example, the publishing of a post in a social media platform. Discrete event processing is

characterized by the presence of events that are generally unrelated to one another and may be handled independently.

Event stream processing

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.

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.

Complex event processing

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.

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.

When to use EDA

There are several use cases that play to the strength of event-driven architecture:

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!
2. **High fan-out.** Scenarios where one event might be processed by multiple, diverse consumers.
3. **Complex pattern matching.** Where events might be strung together to infer more complex events.

4. **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.

Benefits of EDA

1. **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.

2. **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.

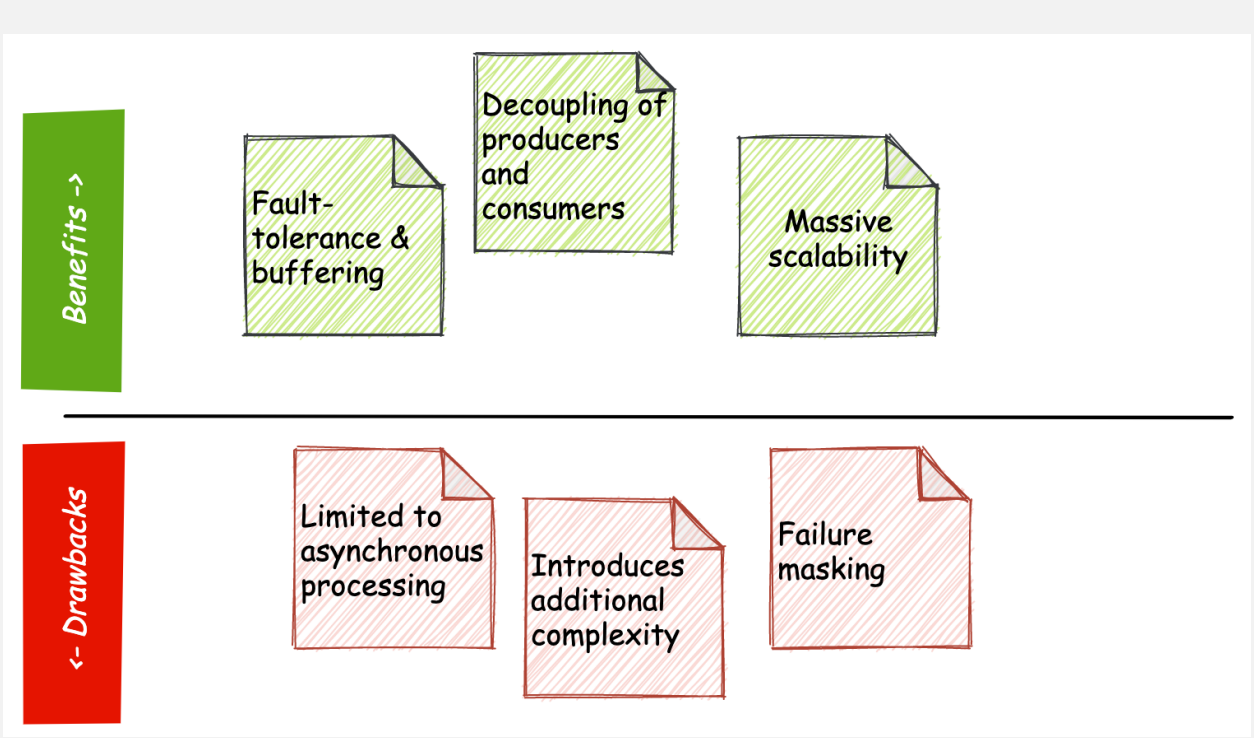
3. **Massive scalability.** It is often possible to partition event streams into unrelated substreams and process these in parallel. We can also scale the number of consumers to meet the load demands if the backlog of events grows. Platforms like Kafka enable the processing of events in strict order while

simultaneously allowing massive parallelism across the stream.

Drawbacks of EDA

1. **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.
2. **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.
3. **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.



Benefits and drawbacks of event-driven architecture

Things to watch out for

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.

1. **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.

2. **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.

3. **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.

Conclusion

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”.

To help complete the puzzle and address the issue of coupling, we looked into Event-Driven Architecture.

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.