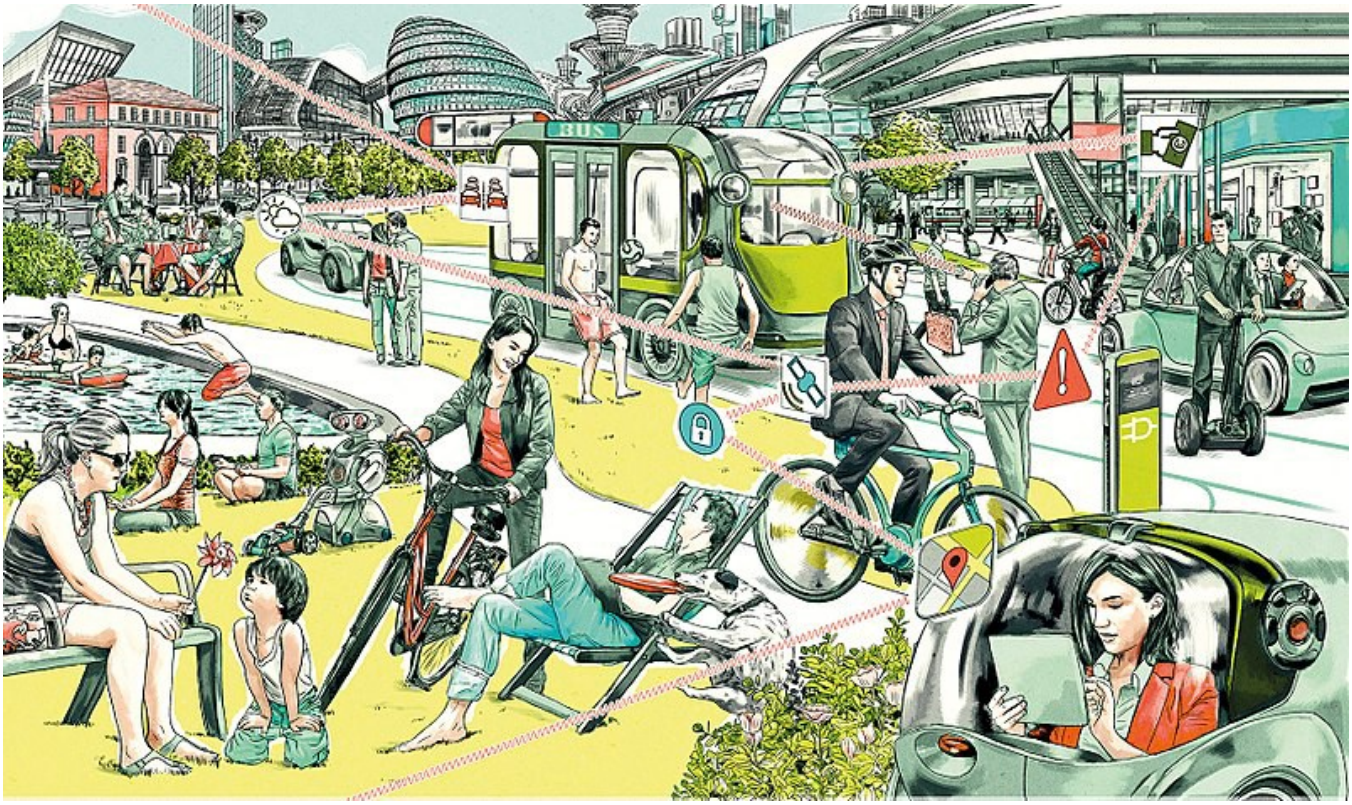
What Is Event Driven Architecture, and When Should I Use It?

By Craig Godden-Payne@beardy.digital

Craig Godden-Payne
Mar 21 · 4 min read ★



Event-driven architecture is a software architecture paradigm promoting the production, detection, consumption of, and reaction to events.

How is an event-driven architecture implemented, and when would you use it?

Event driven architecture goes hand in hand with Microservices. When an action occurs, an event is created and this event is then used to drive decisions across anything that is waiting for that event to occur.

Services are no longer tied together, because in a publish subscribe type model, the caller is no longer calling the callee synchronously. Instead the callee acts upon an event in an eventually consistent way.

Because of this, event driven architectures can be more reliable, because they do not have to act upon a service call immediately (allowing services to be able to fail until they succeed) but are less predictable to know when an action has been performed (for the very same reason).

A simple example of an event driven architecture would be Amazon. If you have ever shopped at Amazon on a busy period, such as Black Friday, it is possible to order an item, but only to be sent an email afterwards to say the item you ordered is actually out of stock.

If you were to think of this process in terms of an event driven architecture, it likely works the following way:

- The customer orders the item, and publishes an OrderPlaced event
- The Stock service subscribes to the event, but by the time it processes event, it checks the stock, and it is now at 0
- The Stock service then publishes a OutOfStock event
- The Email service subscribes to this event and sends an email to the customer explaining that the item is out of stock.

Event driven architectures can use a queue as a backing model in combination with a publish subscribe model. This is to guarantee delivery of event messages, in the event that some service in the chain is unable to process an event.

Photo by Ed 259 on Unsplash

Pros and Cons of Event-Driven Architecture

There are a few reasons why using an event driven architecture can be an advantage over alternative architectures.

Loose coupling — Services do not need to be dependent on each other. This applies different factors such as transport protocol, availability (the service being online) and the data being sent. The consumer will still need to know how to interpret an event or message so a strict contract should still be used between the two services, but implementation details of the contract should not matter.

Scalability — Since the services are no longer coupled, the throughput of service 1, no longer needs to meet the throughput of service 2. This can help reduce costs as services

no longer need to be online 24/7 and it is possible to take advantage of serverless computing with infinite scaling.

Asynchronicity — Since services are no longer dependent on a result being returned synchronously, a fire and forget model can be used, which can greatly speed up a process. This can have a downside, which will be outlined below.

Point in time recovery — If events are backed by a queue or maintaining some kind of history, it is possible to replay events, or even go back in time and recover state.

There are drawbacks to using an event driven architecture as well.

Over-engineering of processes — Sometimes a simple call from one service to another is enough. If a process uses event driven architecture, it usually requires much more infrastructure to support it, which will add costs (as it will need a queueing system)

Inconsistencies — Because processes now rely on eventual consistency, it is not typical to support ACID (atomicity, consistency, isolation, durability) transactions, so handling of duplications, or out of sequence events can make service code more complicated, and harder to test and debug all situations.

What happens when events change?

Imagine a situation when I want to add or change the contract of an event, e.g. in an OrderPlaced event, changing the quantity from an integer, to a floating point number.

This is a situation that would break the subscriber, if it was not expecting this change.

You need to version the contract of the event, to prevent this breakage. A good rule of thumb is usually any additions to a contract would be fine, but for something to be removed, or changed a new version of the event would have to be published, with the subscriber also expecting the new event contract.