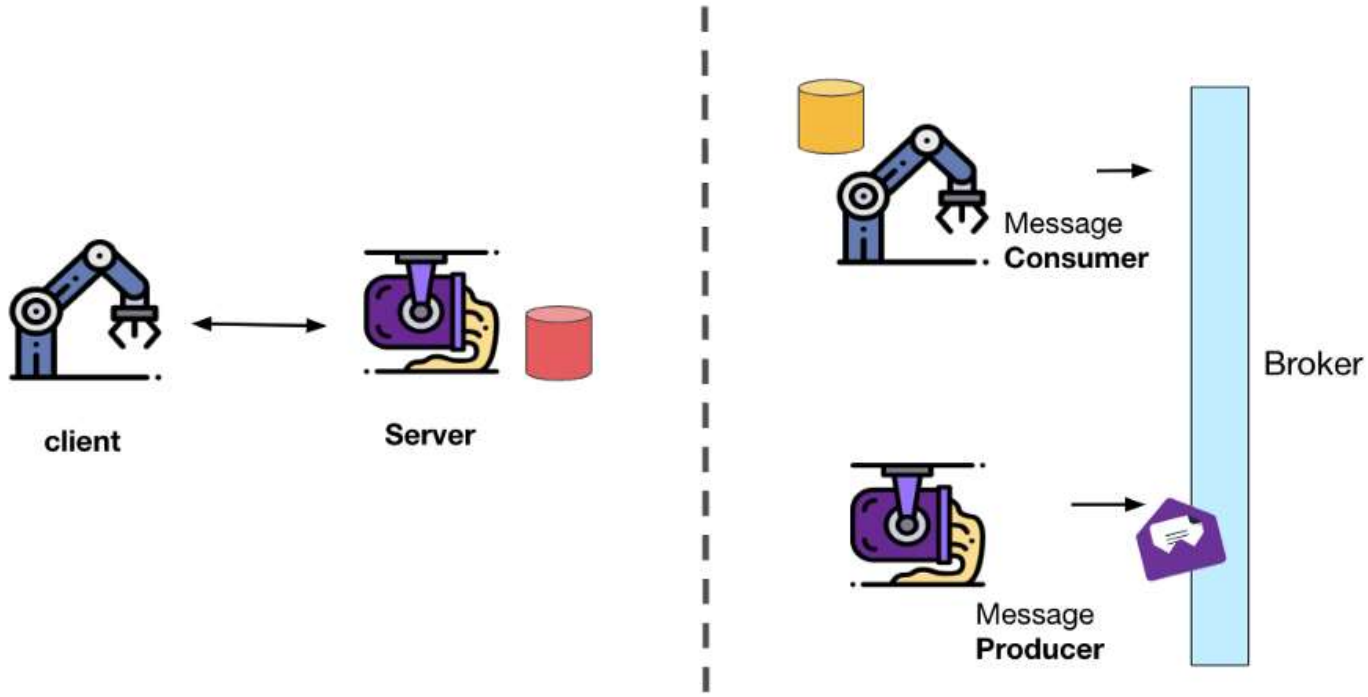
Published in Wix Engineering

Natan Silnitsky Follow
Aug 14 · 10 min read · Listen

Event Driven Architecture — 5 Pitfalls to Avoid

Event driven architecture is very powerful and well suited for a distributed microservices environment. By introducing a broker intermediary, event driven architecture offers a **decoupled** architecture, easier **scalability**, and a much higher degree of **resiliency**.



request reply (client-server) vs. event streaming (pub-sub)

But it's much harder to set up correctly as opposed to request-reply client-server type architecture.

At Wix we have been gradually migrating our growing set of microservices (currently at 2300) from the request-reply pattern to event driven architecture over the last few years. Below there are **5 pitfalls** that Wix engineers have encountered during our experimentation with event driven architecture.

These pitfalls have caused us great pain, in terms of production incidents, required re-writes and steep learning curves. For each pitfall I provide battle-tested proven solutions used at Wix today.

1. Write to db and then fire event without atomicity

Consider for example a simple ecom flow (we will use this example throughout this article)

Once the payment processing is done, the product inventory should be updated to reflect that the product is reserved for the customer.

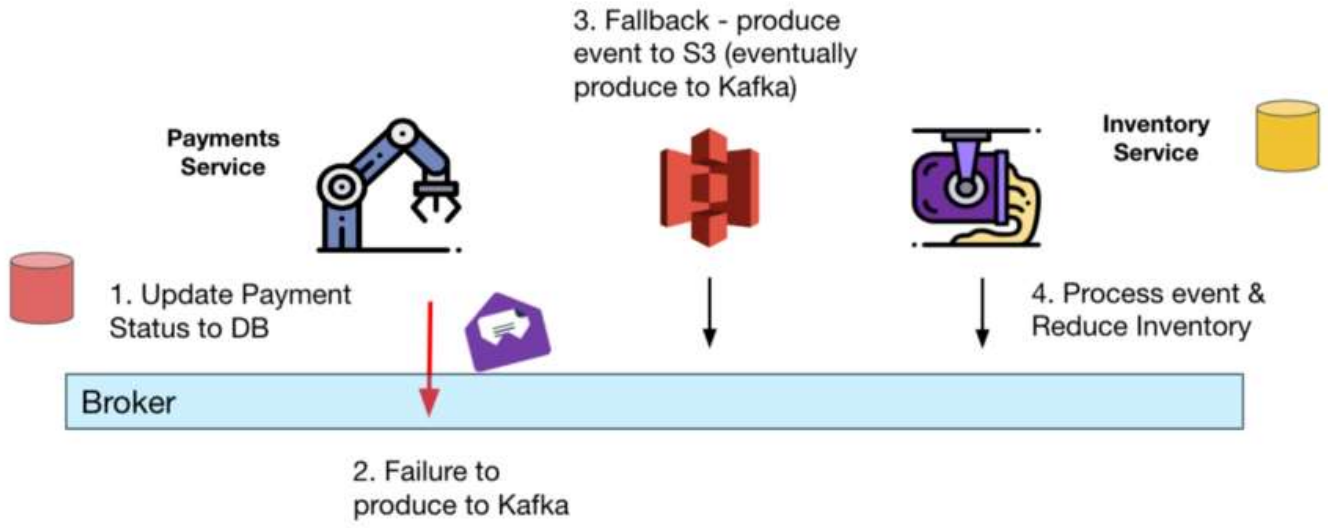
Write to DB and produce event is a non atomic action

Unfortunately, writing the payment-complete status to the database and then producing a "payment completed" event to Kafka (or some other message broker) is not an atomic operation. There could be situations where only one of the actions actually happens.

For example, cases such as the database being unavailable, or Kafka being unavailable, could lead to data inconsistency between different parts of your distributed system. In the case above **Inventory levels** may become **inconsistent** with actual orders.

Atomicity Remedy I — Greyhound resilient producer

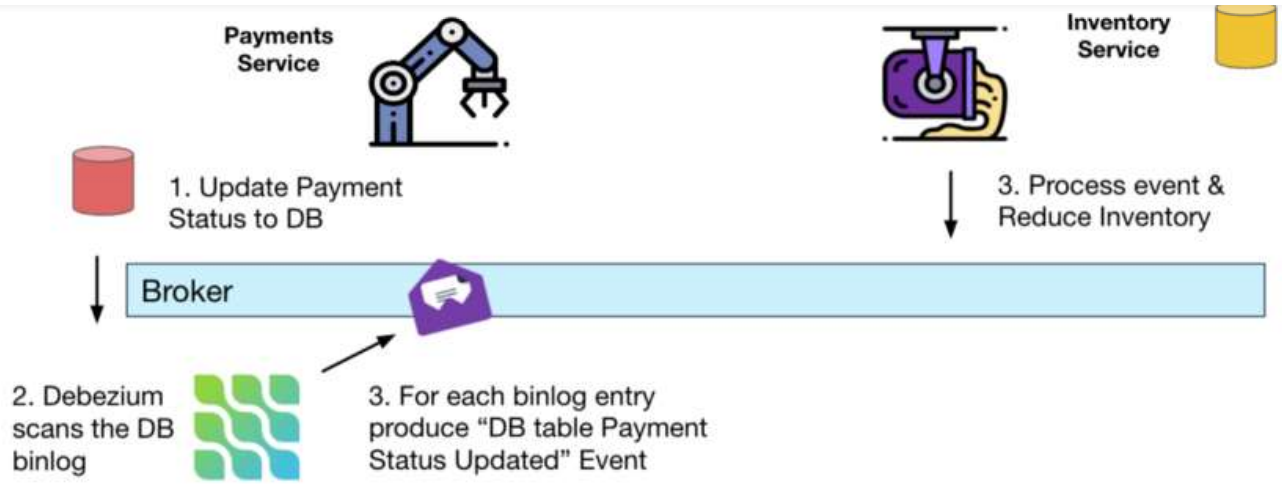
There are several ways to mitigate this issue. At Wix we utilize two ways. The first one is our own messaging platform called Greyhound, that allows us to make sure the event is *eventually* written to Kafka via resilient producer. One downside of this mitigation is that you can end up with out-of-order processing of events downstream.



Greyhound producer fallbacks to S3. A dedicated service recovers the messages to Kafka

Atomicity Remedy II — Debezium Kafka source connector

The second way to make sure both DB update action and Kafka produce action happen and that data remains consistent is using the Debezium Kafka connector. Debezium connector allows to automatically capture all the change events (CDC) that happen in the database (For MySQL via binlog) and produce them as Kafka events.
Kafka Connect together with Debezium DB connectors guarantees that events will eventually be produced to Kafka. In addition there is a guarantee that events order will be kept

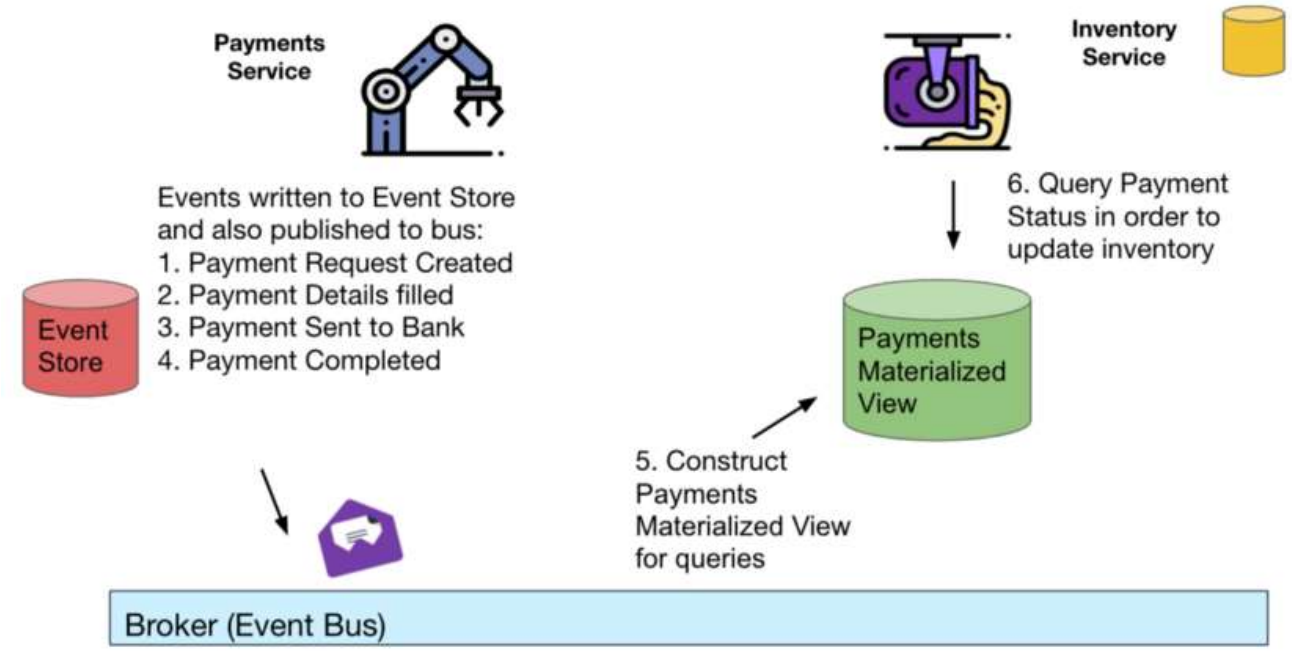


Debezium connector makes sure change events are eventually consistent with DB

Note that Debezium also works with other event streaming platforms such as Apache Pulsar.

2. Using Event sourcing everywhere

Event sourcing is a pattern where instead of updating an entity's state upon a business operation, the service saves an event to its database. The service reconstructs an entity's current state by replaying the events.
These events are also published on an event bus such that other services can also create materialized views on other databases that are optimized for queries by replaying the events.



Event Sourcing — persist change events to Event Store. Play events to reach current state

While there are certain advantages to this pattern (a reliable audit log, performing "time travel" — the ability to get the state of your entity at any point in time, and building multiple views over the same data), it is by far more complex than CRUD services that update the state of an entity stored in a database.

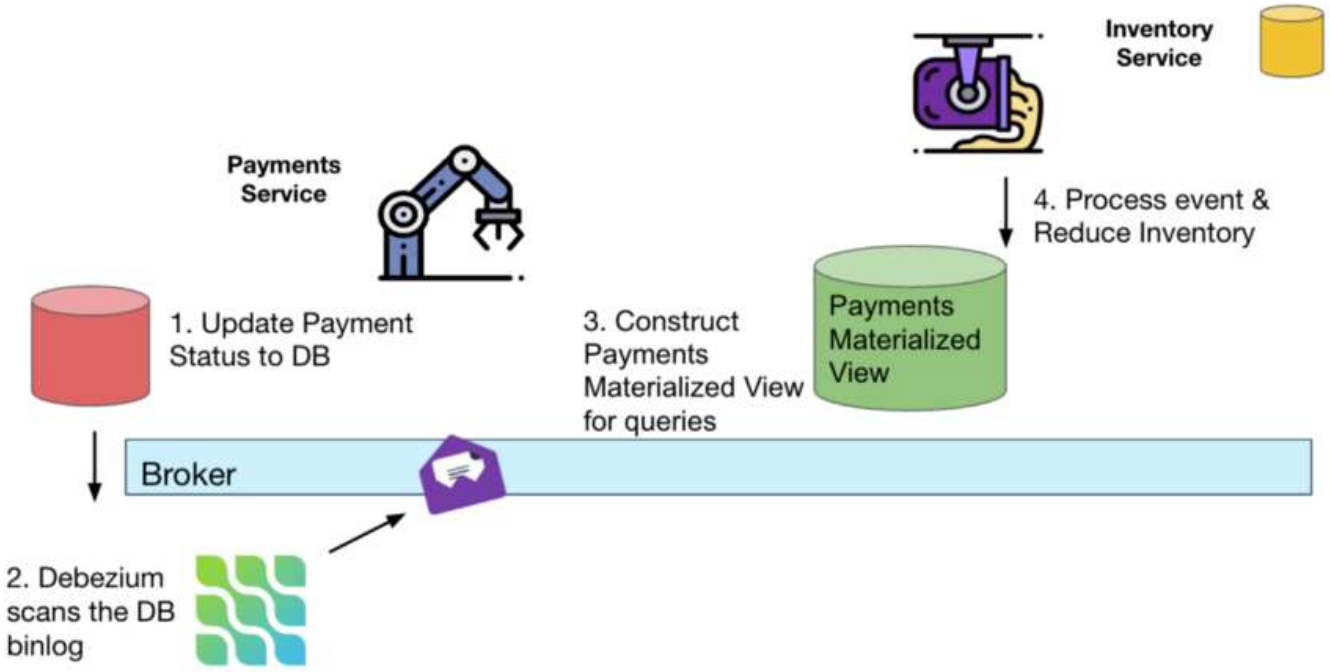
Disadvantages of event sourcing include:

2. Snowflake nature — Unlike CRUD ORM solutions, It's harder to create common libraries and frameworks to ease development that can globally solve snapshotting and read-optimizations that can fit for every single use case.
3. Only supports eventual consistency (problematic for read-after-write use-cases)

Event Sourcing alternative — CRUD+CDC

Utilizing both simple CRUD capabilities and publishing database change events (CDC) for downstream uses (e.g. creation of query-optimized materialized views) can reduce complexity, increase flexibility and still allow for Command-Query Responsibility Segregation (CQRS) for specific use cases.

For most of the use-cases, the service can expose a simple read endpoint that will fetch the entity's current state from the database. As scale increases and more complex queries are needed, the additional published change events can be used to create custom materialized views specifically tailored for complex queries.



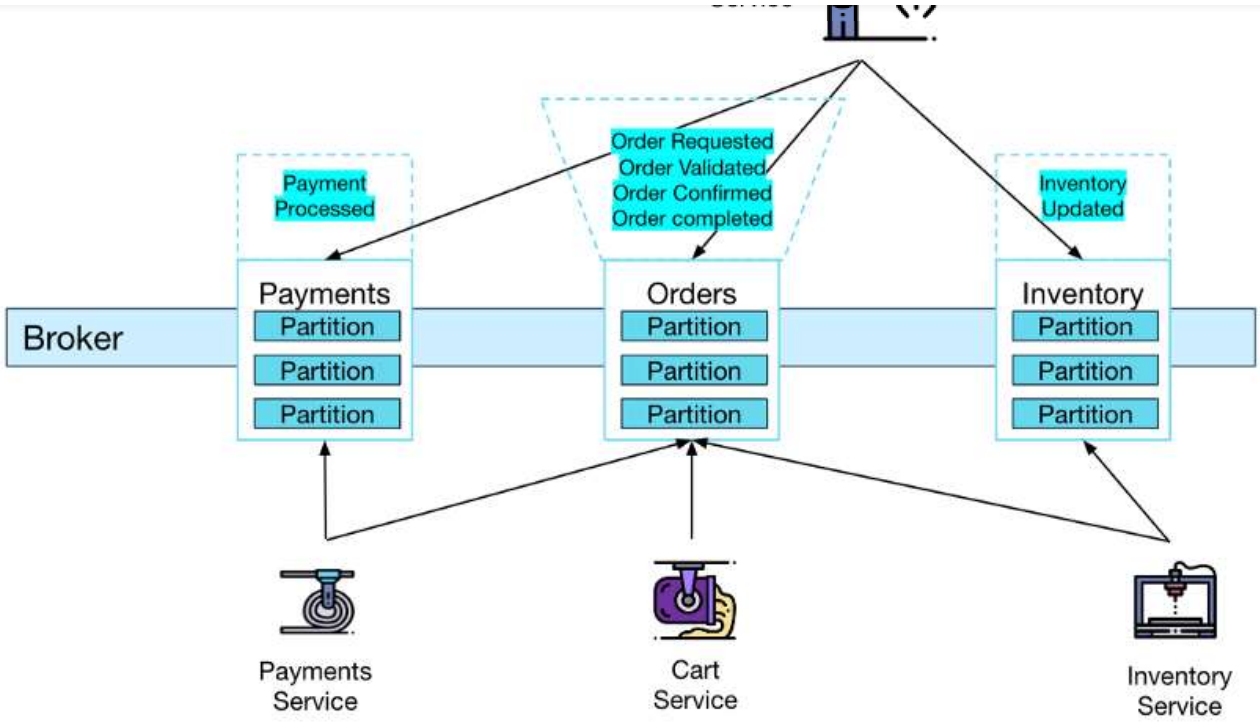
CRUD — simple read from DB + CDC for external materialized views

In order to avoid exposing DB changes as a contract to other services, and creating a coupling between them, the service can consume the CDC topic and produce an "official" API of change events similar to the event stream created in event-sourcing pattern.

3. No Context Propagation

Switching to event driven architecture means developers, devops and SREs potentially have more difficulty to debug production issues and track the processing of end-user requests throughout the system.
Unlike with the request-reply model, there is no explicit chain of HTTP/RPC requests to follow. Debugging code is harder as event handling code is spread out across the service code instead of being sequentially traceable by clicking into function definitions usually found in the same object/module.

Consider for example the ecom flow that I'm using throughout this article. The Orders service has to consume multiple events from 3 different topics all related to the same user action (purchasing items in a webstore).

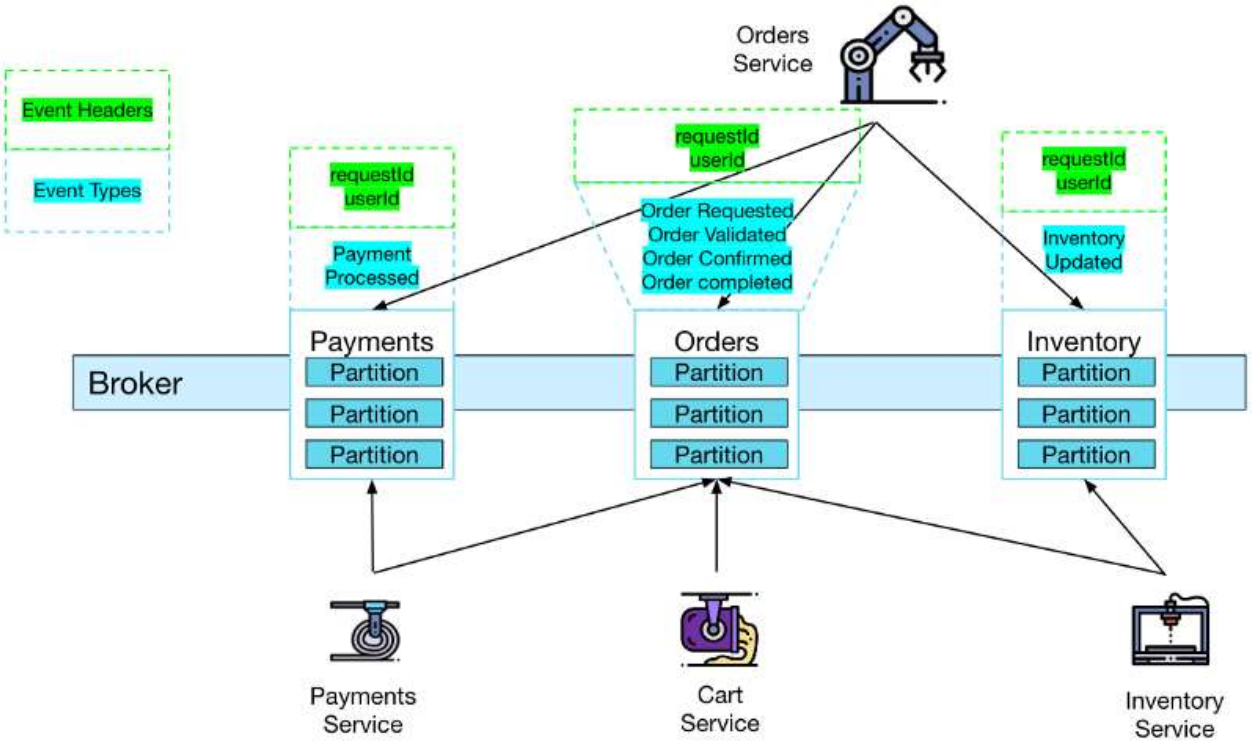


Fully event driven micro-services with hard to follow request flow

The other services also consume multiple events from one or more topics. Let's assume that it was discovered that some inventory level is incorrect. Being able to investigate all related order handling events is crucial. Otherwise it will take a long time to go to individual services logs and try to manually connect the different pieces of evidence into one cohesive narrative.

Automatic Context Propagation

Automatically adding identification of the broader request context for all of the events, makes it really simple to filter for all events related to the end-user request. In our ecom example, 2 event headers were added — requestId and userId. Both of these IDs can greatly help with investigations.



Automatically attach user request context for each event for easier tracing & debugging

4. Publishing Events with Large Payloads

When processing large event payloads (payloads bigger than 5MB, e.g. Image Recognition, Video Analytics, etc…) it may be tempting to publish them to Kafka (or Pulsar) but there is a risk of greatly increasing latency, reducing throughput and increasing memory pressure (especially when tiered storage is not used)

Fortunately, there are a few ways to overcome this issue. Including introducing compression, splitting payloads to chunks, and putting the payload in object store and just passing a reference in the streaming platform.

Large Payloads Remedy I — Compression

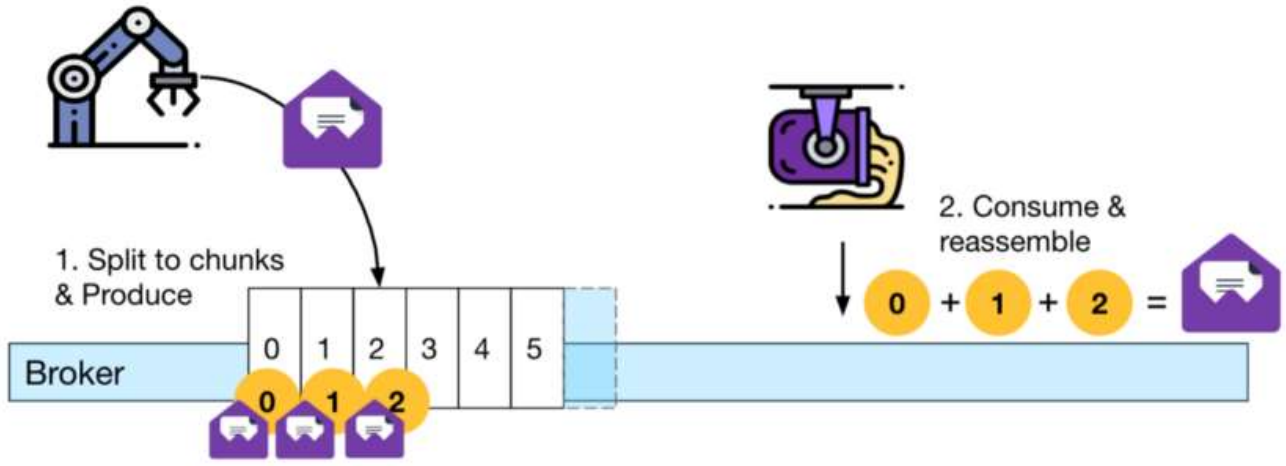
Both Kafka and Pulsar allow compression of payloads. You can try several compression types (lz4, snappy, etc.) to find the one best suited for your payload type. If your payload is a bit large (up to 5MB), compression of 50% can help make sure you maintain good performance of your Message broker clusters.

Compression on Kafka level is usually better than application level, as payloads can be compressed in batches and thus improve compression ratio.

Large Payloads Remedy II — Chunking

Another way to reduce the pressure on brokers and override message size limitations is to split the messages into chunks.
While chunking is already a built-in feature of Pulsar (with some limitations), for Kafka chunking has to happen on the application level.

Examples of how to implement chunking on application level can be found here and here. The basic premise is for producers to send out the chunks with additional metadata that helps consumers to re-assemble them.



producer splits to chunks, consumer figures out how to assemble

The two examples approaches are different in how they assemble the chunks back to the original payload. The first example keeps the chunks in some persistent storage and the consumer fetches them once all chunks have been produced. The second example makes the consumer seek backwards in the topic partition to the first chunk, once all chunks have arrived.

Large Payloads Remedy III — Reference to Object store

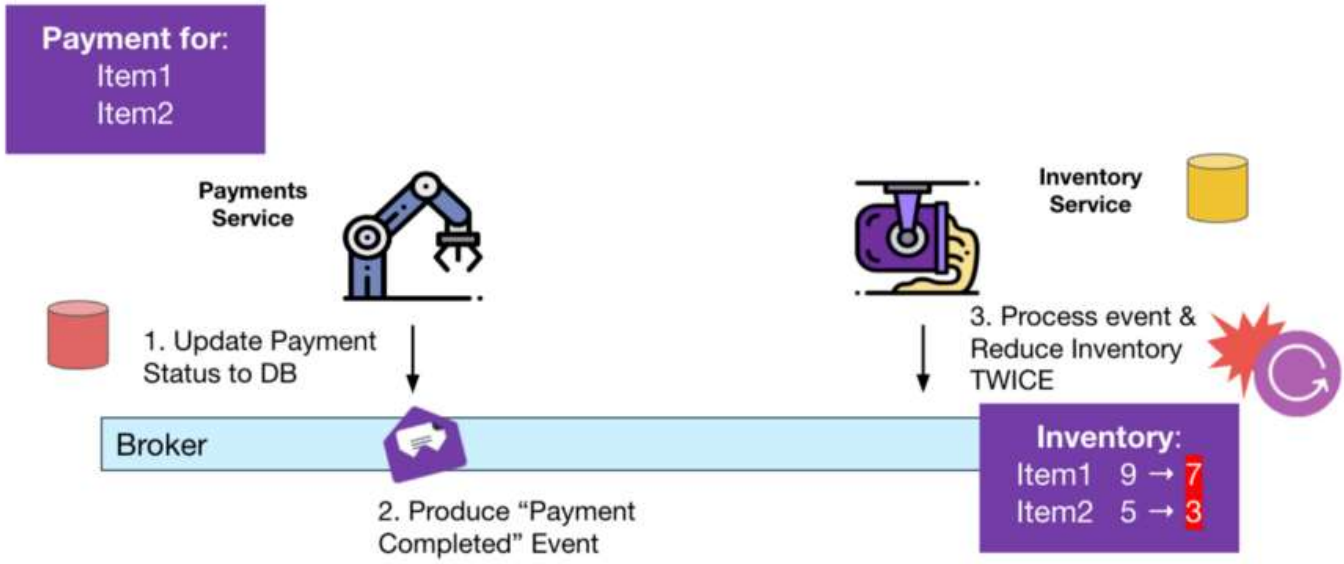
The final approach is to simply store the payload in an object store (such as S3) and pass a reference (a URL usually) to the object in the event payload. These object stores allow to persist any required size without impacting first byte latency.

It's important to make sure the payload is fully uploaded to the object storage before the link is produced, or else the consumer will need to keep retrying until it can start downloading it.

5. Not handling duplicate events

Most message brokers and event streaming platforms guarantee at least once delivery by default. Meaning that some events are

Consider the simple ecom flow that I've been using throughout this article. In case of duplicate processing due to some processing error, inventory levels recorded into the Inventory database for purchased items may drop more than they actually should.



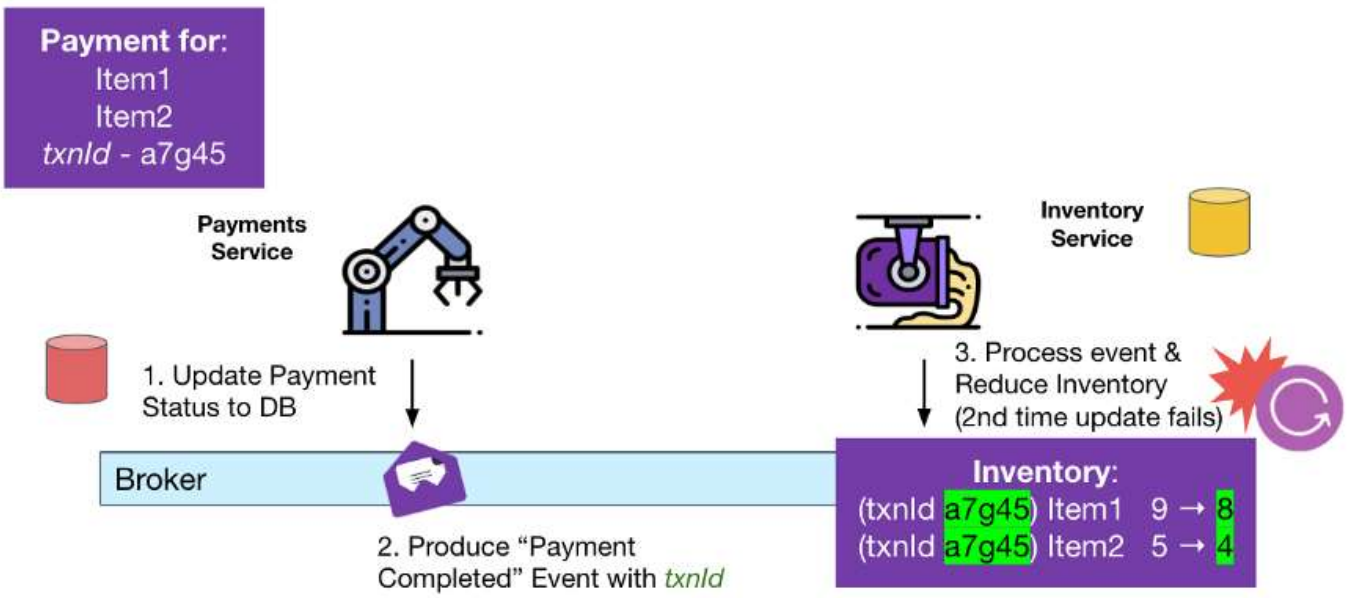
double consumer processing causes inventory level to become incorrect

Other side-effects include calling a 3rd party api more than once (In our ecom case, it could mean calling inventory service with reduce-level twice for same event and item)

Idempotency Remedy — revisionId (versioning)

Optimistic locking technique can serve as inspiration in cases where idempotency of event processing is needed. With this technique, the current revisionId (or version) of the stored entity is first read before any update occurs. If more than one party tries to update the entity (while incrementing the version) at the same time (concurrently), the 2nd attempt will fail as the version will no longer match with what it read before.

In the case of idempotent handling of duplicate events, the revisionId has to be **unique** and **part of the event itself** in order to make sure that two events don't share the same id and that 2nd updates on the same revisionId will (silently) fail even if does not happen concurrently.



* For more on Exactly once delivery in Kafka you can watch my talk from DevOpsDays Tel Aviv Conference

I would like to thank Oded Apel, Dalia Simons and Evgeny Krasik for their great feedback!

Summary

A migration to event-driven architecture can be gradual in order to reduce risks involved with it including harder debugging and mental complexity. Microservices architecture allows flexibility in the choice of pattern for each of the different services. HTTP/RPC endpoints can be called as part of an event's processing, and vice versa.

As a consequence of this gradual migration approach I strongly recommend to adopt the CDC pattern (Database changes streamed as events) as a way to both ensure data consistency (pitfall #1) and avoid the complexity and risks associated with full blown event-sourcing (pitfall #2). The CDC pattern still allows to have the request-reply pattern in place side by side with the event processing pattern.

Fixing Pitfall #3 (propagation user request context throughout your event streaming flows) will greatly improve your ability to find root causes of production incidents quickly.

Remediations for Pitfalls #4 and #5 are for more specific uses cases — *very large payloads* in case of pitfall #4 and non-idempotent side-effects in case of #5. No need to perform the recommended changes in case there is no need for them. Although compression (#4) and transactionIds (#5) are best-practices that you can add by default.

Thank you for reading!

If you'd like to get updates on my future software engineering blog posts, follow me on Twitter and Medium.

You can also visit my website, where you will find my previous blog posts, talks I gave in conferences and open-source projects I'm involved with.

If anything is unclear or you want to point out something, please comment down below.

Further reading:

- 6 Event-Driven Architecture Patterns
- 4 Microservices Caching Patterns at Wix
- Leader election and sharding practices at Wix Microservices
- Gwen Shapira on Event Driven Architecture and Cloud Native Kafka