EVENT STORE®

Beginners Guide To **Event Sourcing**

Contents

Introduction

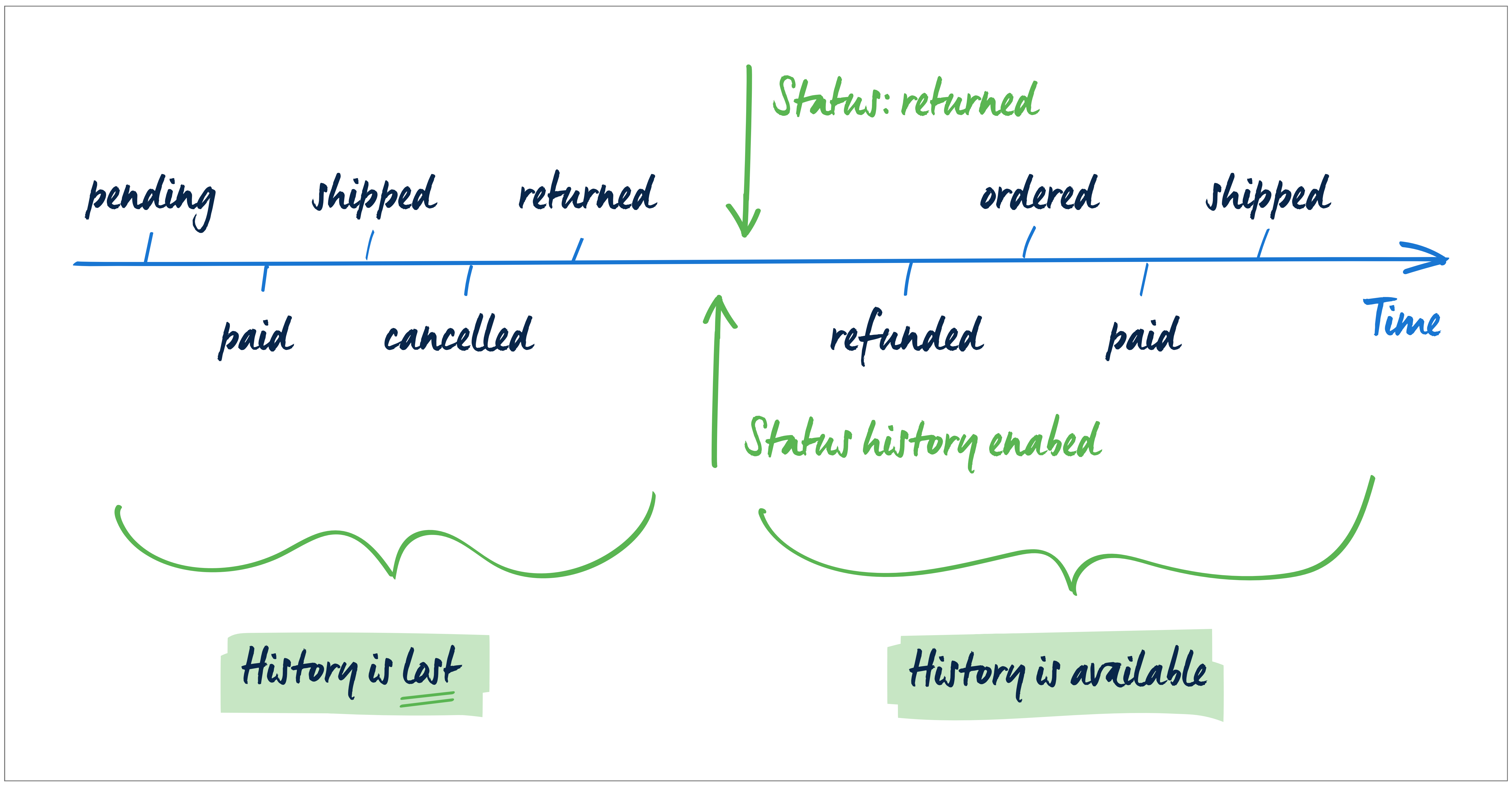
Event Sourcing has been used in production by some of the world's biggest companies (including Netflix and Walmart) for years, providing them with the platform to rapidly scale, iterate and evolve their systems, and establishing a data model that delivers a strong competitive advantage.

In this guide, we discuss what Event Sourcing is, why you'd use it, the range of benefits it provides and we break down the jargon.

What Is Event Sourcing?

Event Sourcing is a pattern where changes that occur in a domain are immutably stored as events in an append-only log.

This provides a business with richer data as each change that occurs within the domain is stored as a sequence of events which can be replayed in the order they occurred. This means you're able to see more than just the current state of your domain - you can see what lead up to the current state.



In addition, as events also contain the context of the change – the 'what', 'when', 'why' and 'who' - an event-sourced system has a wealth of information that can be incredibly valuable to the business.

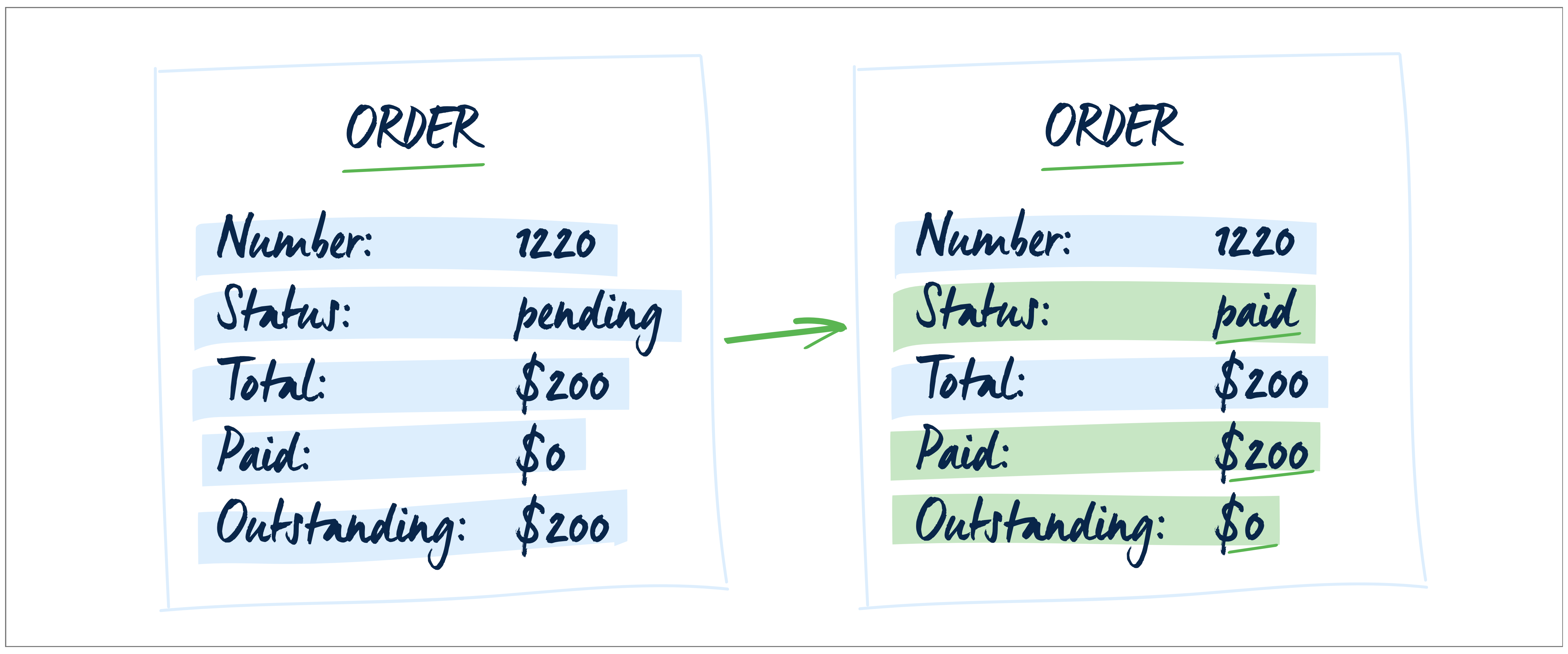
Event Sourcing has a wide variety of applications across many sectors, including finance, logistics, healthcare, retail, government, transport, video game development and many more.

An Event Sourcing Example

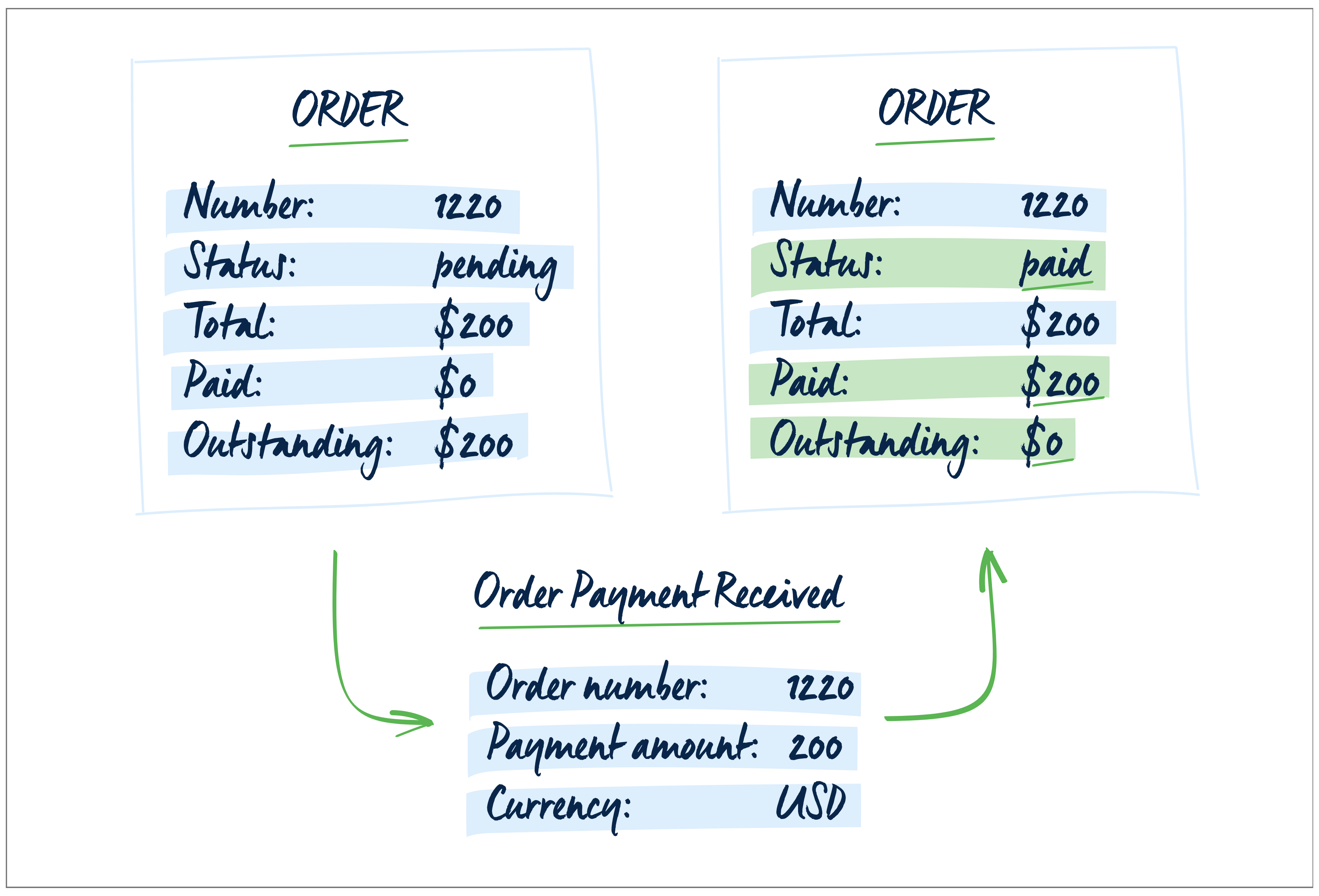
Let's look at an order process as an example to further explain how Event Sourcing works:

A customer places an order and an invoice is raised with the order details. The current status of the domain is 'outstanding' with the amount the customer owes, in this case '$200'.

The customer receives the invoice and pays the bill, the current status is then updated to show the outstanding balance as zero.

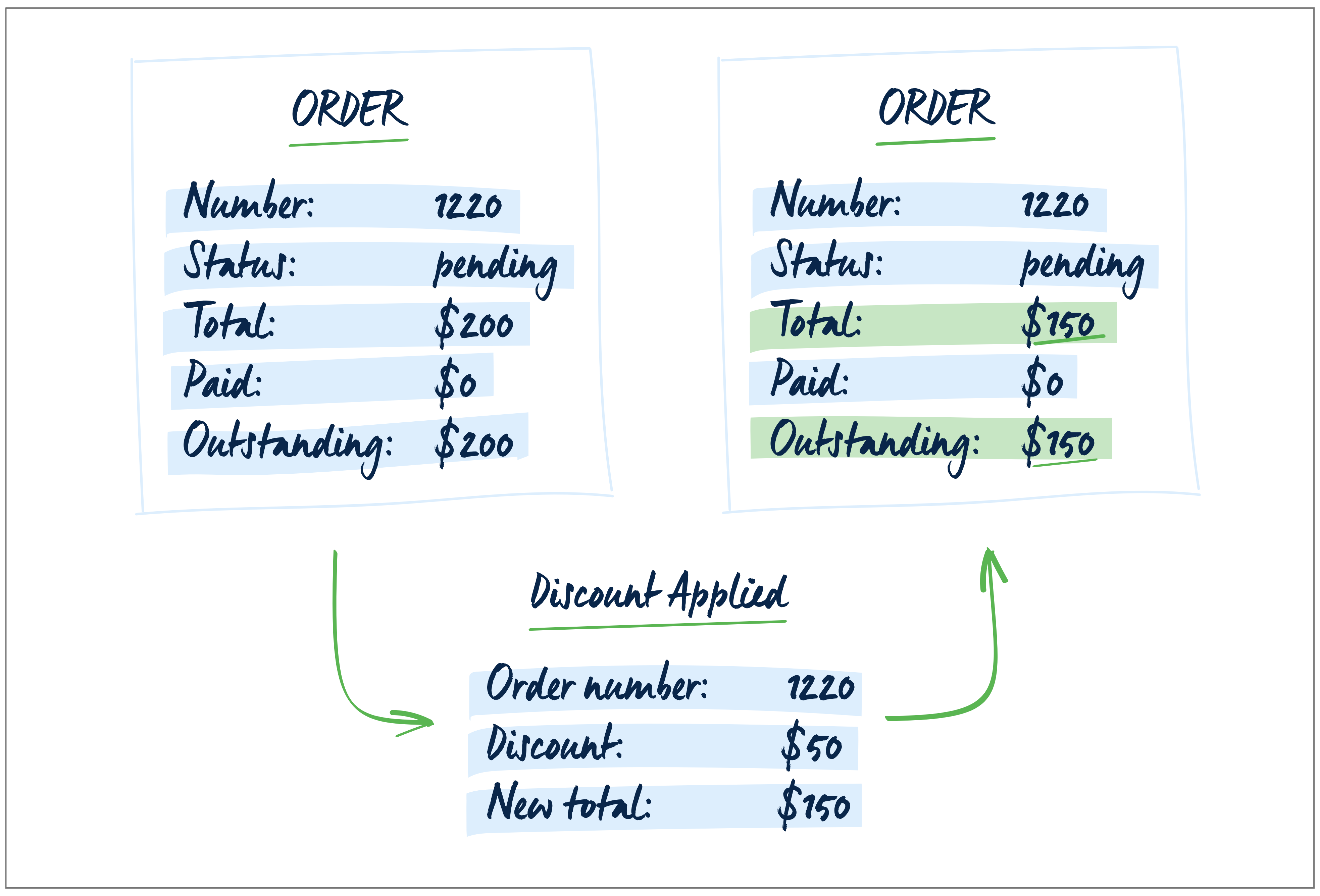


In an event sourced system, the status change would be captured as an event 'OrderPaymentReceived' and stored in the append-only log in the order in which it occurred.

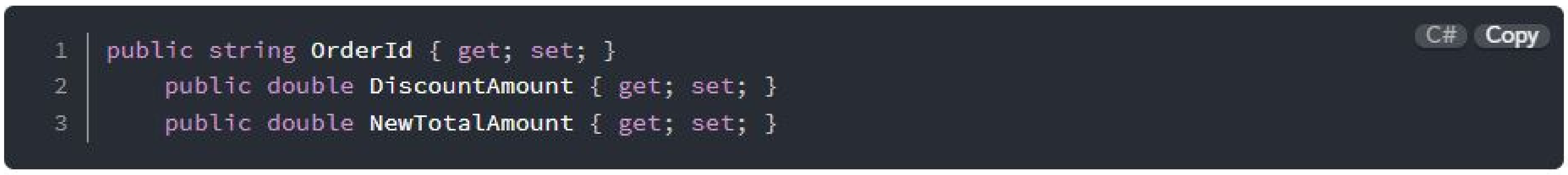


An Event Sourcing Example

Another example, is an order discount. Let's say the customer placed an order, then a discount was applied. In a system that only captures the current state, you'll see the 'Outstanding' amount changed from $200 to $150 and won't know why. In an event sourced system, the change is captured in the event 'DiscountApplied', giving the context of the change - in this case a discount was applied.



This is how the event 'DiscountApplied' would look when implemented in code using C#.



```csharp
public string OrderId { get; set; }
    public double DiscountAmount { get; set; }
    public double NewTotalAmount { get; set; }
```

Event Sourcing offers a lot of benefits to a business by providing much deeper and richer context to the changes that happen within a domain. Let's take a look at some of those benefits.

Benefits Of Event Sourcing

There are some powerful benefits to building systems using Event Sourcing, and we've gathered our top 12 benefits below.

Audit

An event-sourced system stores your data as a series of immutable events over time, providing one of the strongest audit log options available.

Time Travel

All state changes are kept, so it is possible to move systems backward and forwards in time which is extremely valuable for debugging and “what if” analysis.

Root Cause Analysis

Business events can be tied back to their originating events providing traceability and visibility of entire workflows from start to finish.

Fault Tolerence

Event streams are fundamentally just logs with strong backup and recovery characteristics. Writing just the core “source of record” data to the event stream enables the rebuilding of downstream projections. EventStoreDB is a distributed database technology with failover if a leader fails.

Event-Driven Architecture

Traditional approaches gather data in discrete areas to be called on only when needed. An event-driven approach can be more efficient by immediately reacting to new published information. Event streams can create notifications about new events, and by subscribing to the stream, business can react to these changes in real-time. This allows for easier modelling and building complex business workflows.

Benefits Of Event Sourcing

Asynchronous First

Event sourced systems strive for the minimum amount of synchronous interaction; consistency boundaries are consciously chosen so that business requirements are met, and everything else is eventually consistent. This results in responsive, high performance, scalable systems.

Service Autonomy

If a service goes down, dependent services can “catch up” when the source comes back up. Because events are stored in a sequence in the stream, synchronization can be achieved when each service is back online.

Replay And Reshape

The series of events in a stream can be replayed and transformed to provide new insight and analytics, e.g. the event stream can be replayed to a point in time and a “what if” analysis can be used to project potential future outcomes.

Observability

In event-sourced systems, events flow through queues and streams, allowing for unprecedented observability. What is uniquely powerful is that the events can contain the business context which allows real-time analytics.

Occasionally Connected Data

Since there is a log of all the state changes of an application, it can be used in occasionally connected system scenarios. When a device is disconnected it can continue to work on its own data locally and synchronize upon connection.

Benefits Of Event Sourcing

One Way Data Flow

Data in a CQRS/event-sourced system flows one way, through independent models to update or read information. This brings an improved ability to reason about data and debug as each component in the data flow has a single responsibility.

Legacy Migration

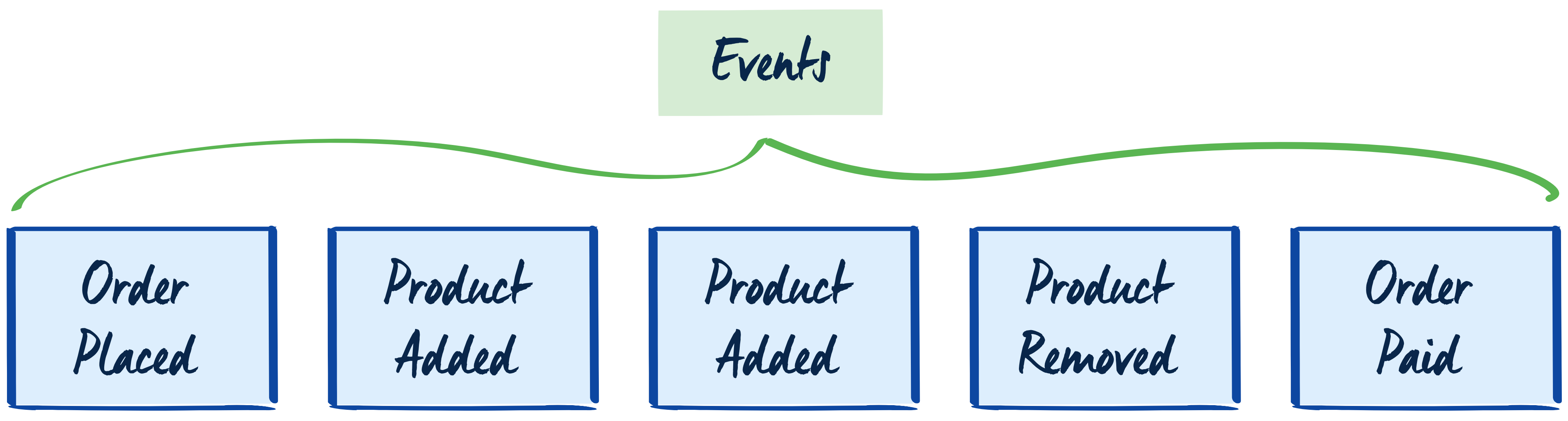
Migration of legacy systems to modern distributed architectures can be carried out incrementally, gradually replacing specific pieces of functionality with event-sourced services. Existing read paths of the legacy system can remain in place while writes are directed to the services.

Core Principles of Event Sourcing

There are some key software architectural concepts and terms that go hand-in-hand with Event Sourcing, and we often mention them within our blog and documentation. It's important to explain these topics at a fundamental level for context, so these are discussed below.

Events

An event represents a fact that took place in the domain. They are the source of truth; your current state is derived from the events. They are immutable, and represent the business facts.



Events are referred to in the past tense and represent the specific business fact. For example, 'InvoiceIssued' shows the state of the invoice has definitely changed, rather than just come into being. It's also a better name than `InvoiceCreated` as it's explicitly describing the fact using the business domain language. The exact definition of an event is going to depend on the business use case, and should reflect your business data.

An event, in terms of Event Sourcing, usually contains unique metadata such as the timestamp of the event, the unique identifier of the subject, etc. The data within the event will be used in the write model to populate the state and make decisions, as well as populate read models.

It's this implicit information in the event name 'InvoiceIssued', along with the metadata and the immutable nature of the event store that makes it an excellent solution for extracting more useful, in-depth insights and context for the business.

Core Principles of Event Sourcing

Event-Native Databases (Event Store)

Each change that took place in the domain is recorded in the database. Event-native databases are natively focused on storing events. Usually, they do that by having the append-only log as the central point.

Event-native databases are a different kind of database from traditional databases (graph, document, relational etc). They are specifically designed to store the history of changes, the state is represented by the append-only log of events. The events are stored in chronological order, and new events are appended to the previous event.

The events are immutable: they cannot be changed. This well-known rule of event stores is often the first defining feature of event stores and Event Sourcing that most people hear, and is absolutely true, from a certain point of view.

The events in the log can't be changed, but their effects can be altered by later events. For example, there may be an 'InvoiceIssued' event appended to the log one day, only for it to be announced that the address the invoice was issued to is incorrect. A new event can be added, with the 'InvoiceVoided' event, then another event with the corrected address and an 'InvoiceIssued' event. All three events are stored, all are still immutable, but the desired outcome is achieved: an invoice has been issued to the correct address. The events are immutable, but that does not mean that log cannot be changed.

This is an example of the business context being kept; every stage the invoice has gone through has been kept, along with the dates and times of all events. Obviously, this information would need to be kept for auditing purposes, but imagine applying this level of information to any and all parts of the business: having an audit-level amount of information for every event in the system. The event store doesn't lose information, and with the right configuration, you can have useful reports and analysis of the data whenever you need it.

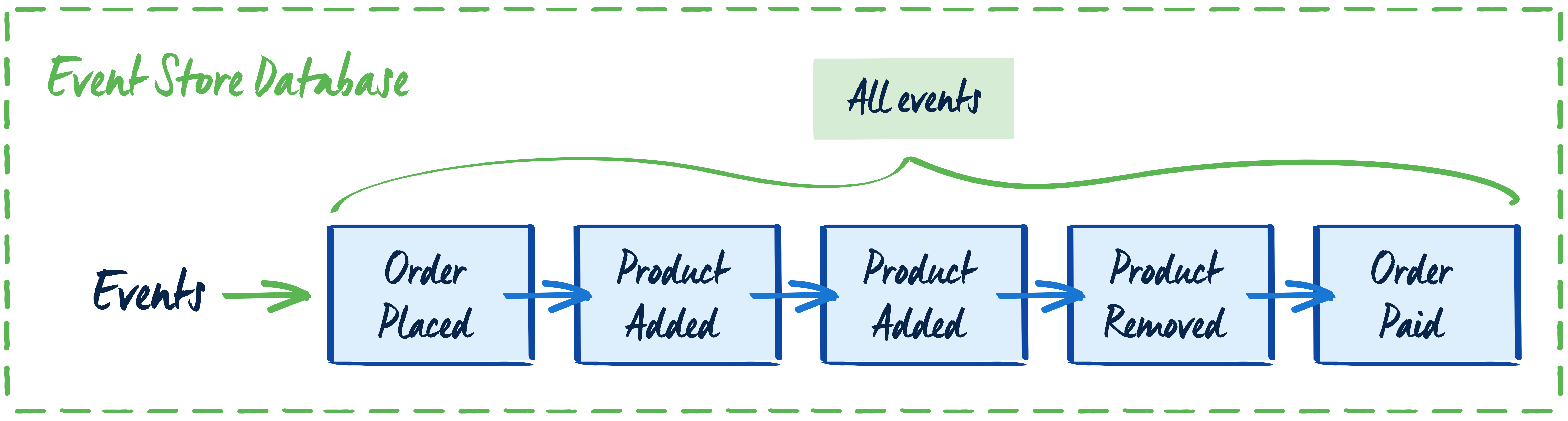
Core Principles of Event Sourcing

Streams

Within the event store, the events referring to a particular domain or domain object are stored in a stream. Event streams are the source of truth for the domain object and contain the full history of the changes. You can retrieve state by reading all the stream events and applying them one by one in the order of appearance.

A stream should have a unique identifier representing the specific object. Each event has its own unique position within a stream. This position is usually represented by a numeric, incremental value. This number can be used to define the order of the events while retrieving the state. It can be also used to detect concurrency issues.

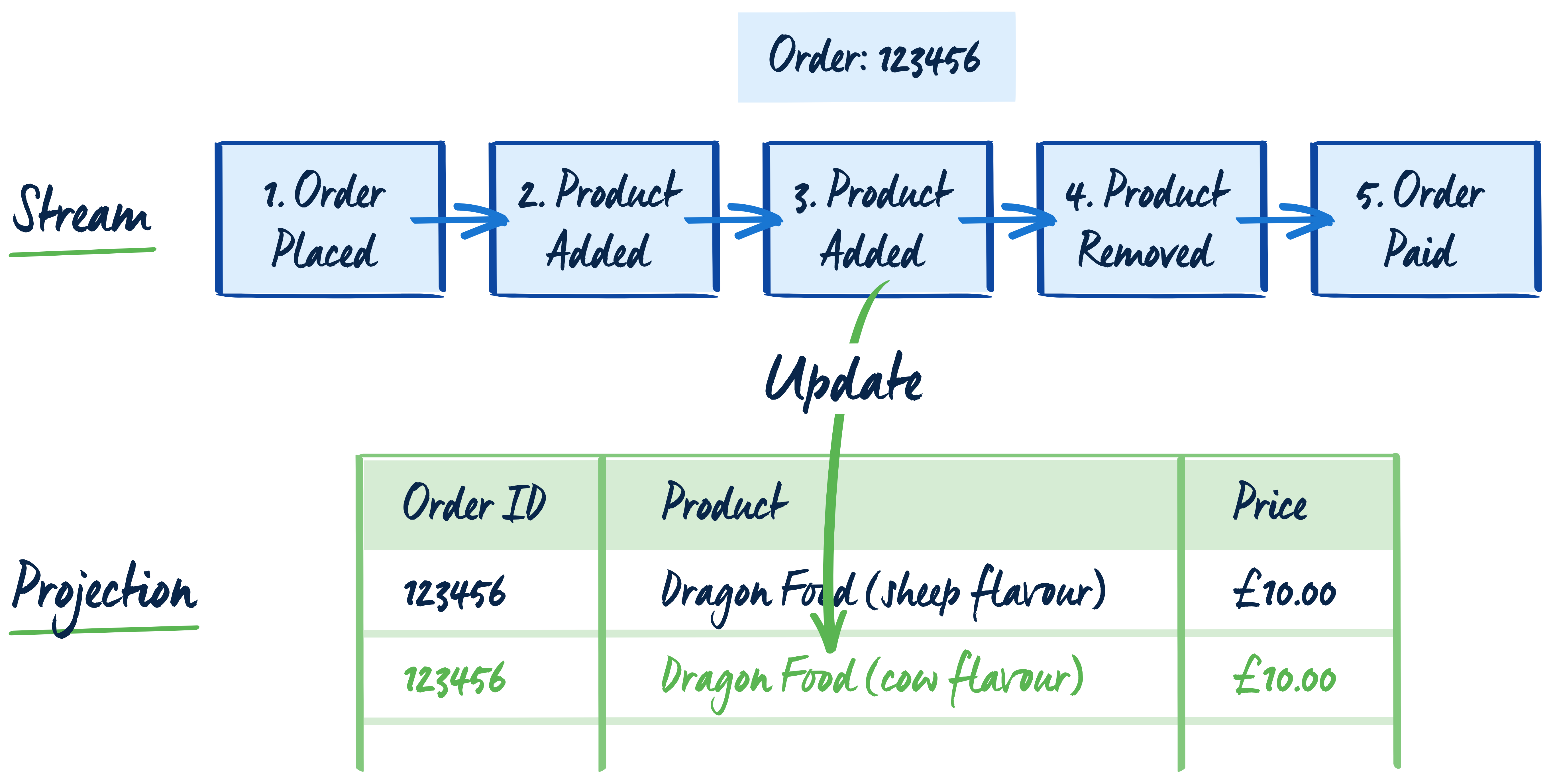
Event stores are built to be able to store a huge number of events efficiently. You don't need to be afraid of creating lots of streams, however, you should watch the number of events in those streams. Streams can be short-lived with lots of events, or long-lived with fewer events. Shorter-lived streams are helpful for maintenance and makes versioning easier.



Core Principles of Event Sourcing

Projections

In Event Sourcing, Projections (also known as View Models or Query Models) provide a view of the underlying event-based data model. Often they represent the logic of translating the source write model into the read model. They are used in both read models and write models.

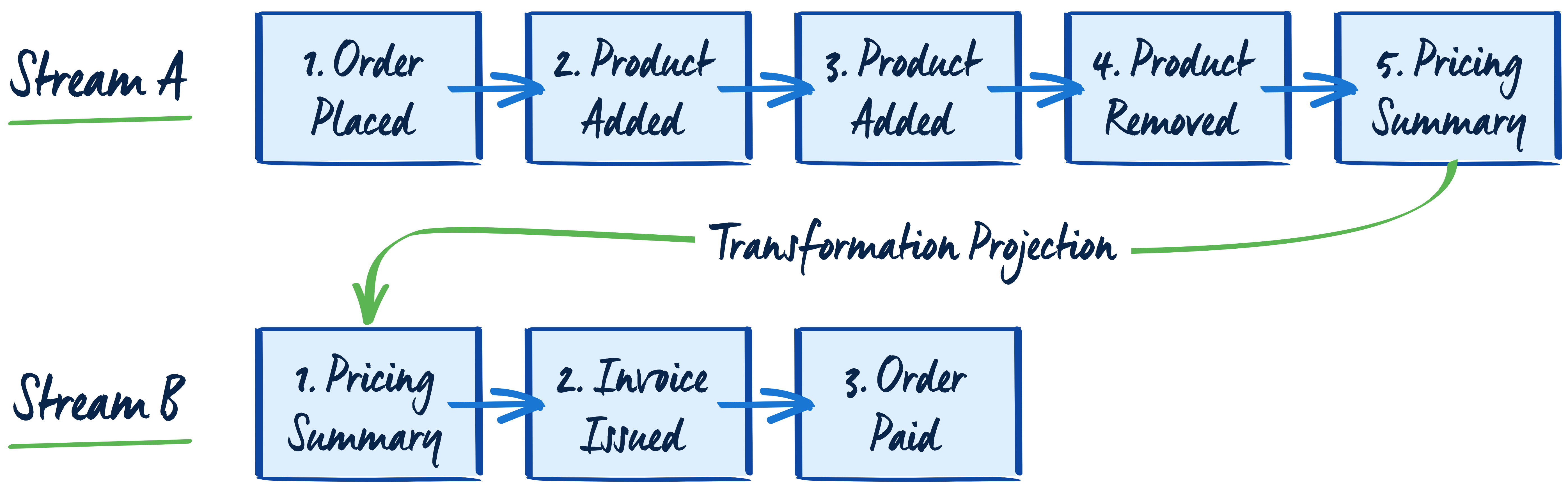


Projections In The Read Model

A common scenario in this context is taking events created in the write model (e.g. InvoiceIssued, OrderPlaced, PaymentSubmitted, OrderItemAdded, InvoicePaid, OrderConfirmed) and calculating a read model view (e.g. an order summary containing the paid invoice number, outstanding invoices items, due date status, etc.). This type of object can be stored in a different database and used for queries.

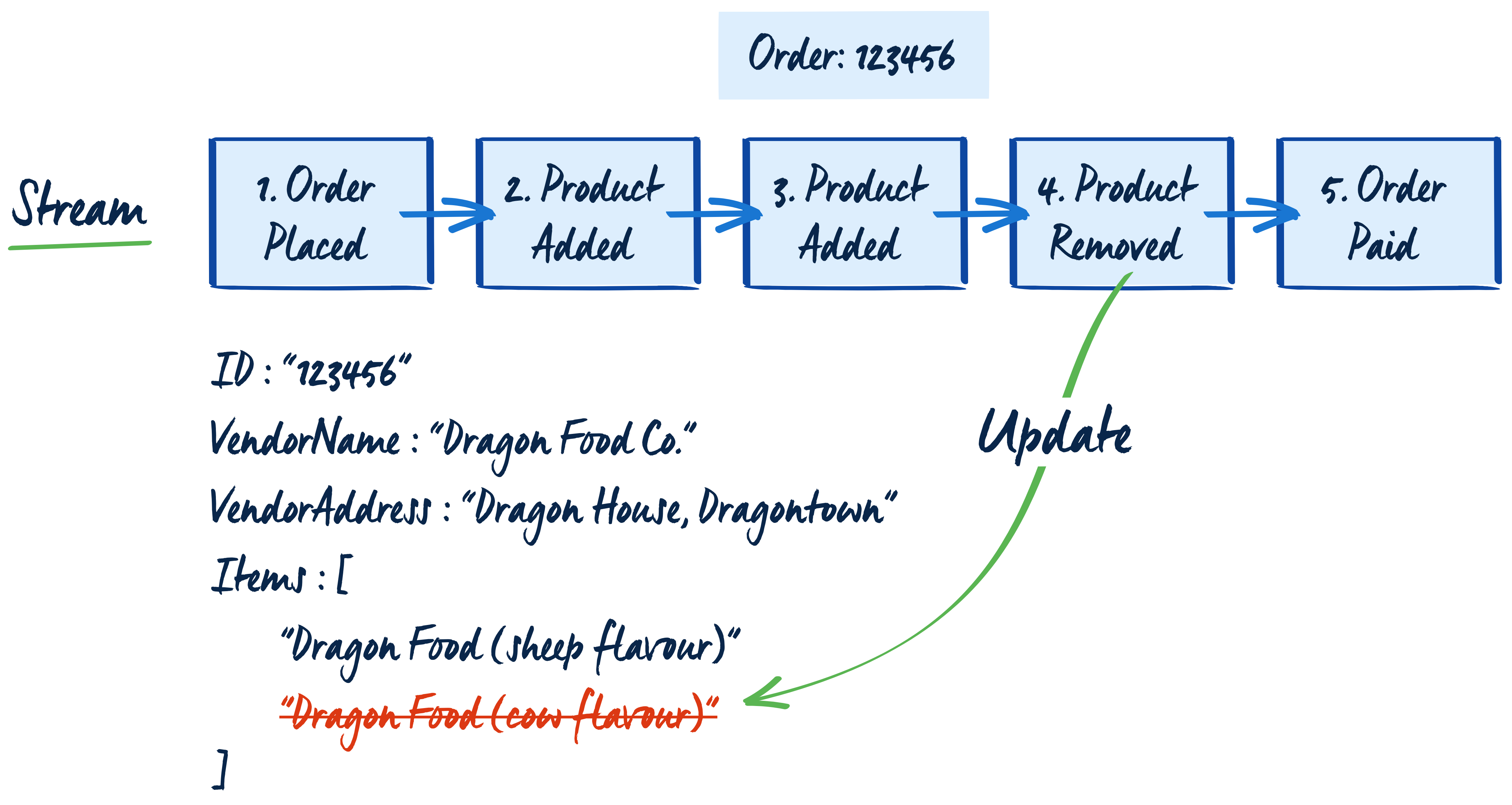
Core Principles of Event Sourcing

A set of events can also be a starting point for generating another set of events. For example, you can take order events, calculate the pricing summary and generate a new order payment event and place it in another stream. This type of projection is also called transformation.



Projections In The Write Model

Another form of projection is called stream aggregation. It's a process of building the current state of the write model from the stream events. During aggregation, events are applied one by one in order of appearance. The main purpose of a stream aggregation is to rebuild the current state to validate an executed command against it.


Order: 123456
Stream
1. Order Placed
2. Product Added
3. Product Added
4. Product Removed
5. Order Paid
Update
ID : "123456"
VendorName : "Dragon Food Co."
VendorAddress : "Dragon House, Dragontown"
Items : [
"Dragon Food (sheep flavour)"
~~"Dragon Food (cow flavour)"~~
]

Core Principles of Event Sourcing

Projections should be treated as temporary and disposable. This is one of their key benefits as they can be destroyed, reimagined and recreated at will; they should not be considered the source of truth.

There is a lot of conceptual crossover between a projection and a read model, and this can lead to some confusion. The simplest way to understand the relationship between projections and read models is that a read model is made of multiple projections. The projections are the method of populating your read model, and represent discrete parts of the whole read model. For example, a projection can be used to create invoices, and another can be used as a financial report, both of which are part of the read model.

A common misconception in explaining Event Sourcing is conflating projections with the state. In Event Sourcing, the source of truth is the events, and the state of the application is derived from these events. Facts are stored in events; projections are an interpretation of that raw data.

Subscriptions

One of the most significant advantages of Event Sourcing is observability. Each action in the system triggers an event, and the event gathers business information about the fact of the action's result. This is a simple but powerful feature, as it allows the building complex business workflows and splitting the work down into smaller chunks.

In Event-Driven Architecture, the responsibilities of services are inverted. The services are decoupled from each other due to the publish/subscribe approach. For example, a reservation service doesn't have to call the invoice service directly: it publishes an event about reservation confirmation, the invoice service subscribes to the reservation event stream, and can issue an invoice right after the notification.

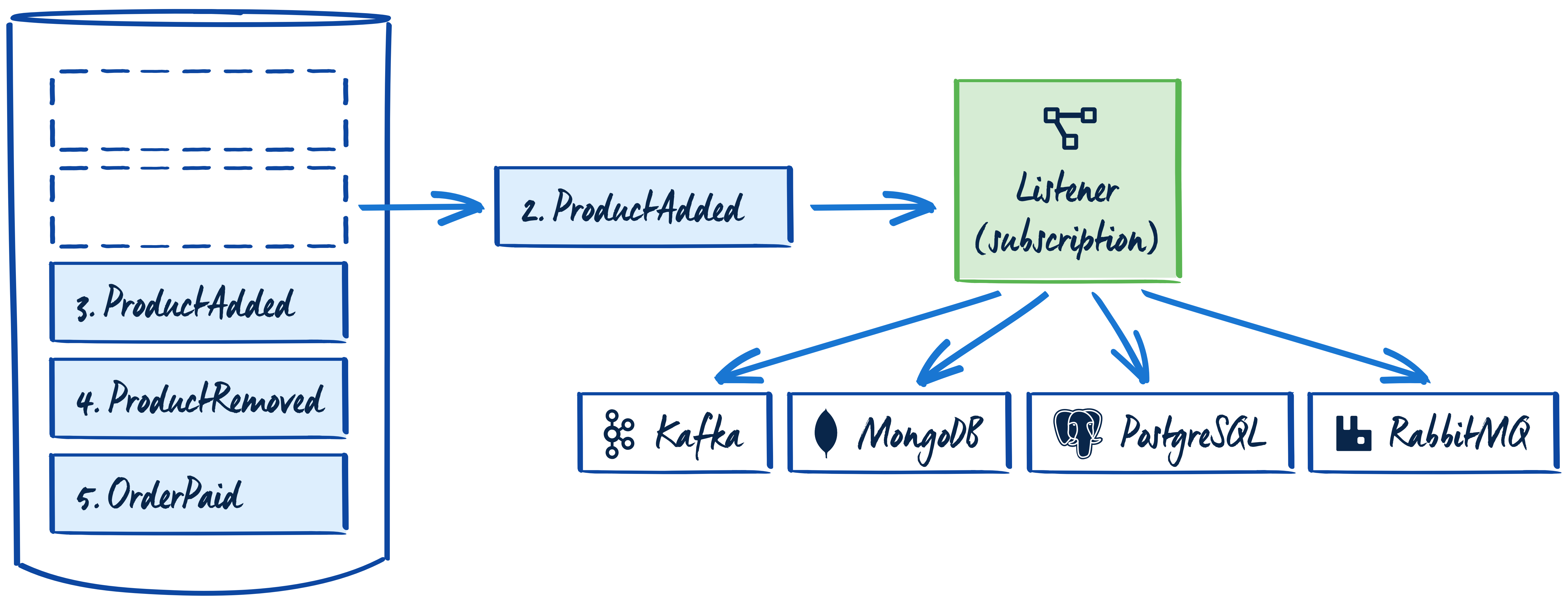
Core Principles of Event Sourcing

Event stores (including EventStoreDB) enable that through the subscriptions functionality. It's a similar concept to the traditional 'Change Data Capture' in relational databases. Each event stored in the database can trigger a notification, and subscribers can listen to those notifications and perform follow-up actions, such as:

- Run a projection to update the read model,
- Perform the next step of the business process,
- Forward the event to the queuing/streaming system to notify external services.

Usually, you can subscribe to either all events (directly from the append-only log) or partitioned/filtered events data (e.g. events from the specific stream, stream type, event type, etc.).

It's important to remember that even though event stores can provide publish/subscribe functionality, they're usually optimised for storage, not transport. For higher throughput needs or cross-service communication, it's worth considering using additional specialised streaming solutions, such as Kafka or Kinesis.



Related Terms

There is a wealth of terms and concepts related to Event Sourcing, some of which have been defined differently by various sources. We've listed some terms here that are related to Event Sourcing and provide useful context to the pattern, but there are misconceptions around.

Eventual Consistency

Eventual consistency is the idea that, in a distributed system, all the different parts of the system will eventually return all the same values. It is a form of weak consistency; not all the parts of the system may return the same values, and certain conditions will need to be met or actions taken in order to make sure all the different parts of the system are consistent. There's a misconception that an eventually consistent system will be inaccurate due to the time delays involved. The time taken to return the same values may not be defined within the system, but the time frames are within the millisecond to seconds range, rather than large spans of time.

No matter what kind of database or structure you use, eventual consistency is something you will have to deal with: it's not a problem specific to Event Sourcing. There will always be a delay between an input being received, being recorded to storage, then called out again. One of the first misconceptions about Event Sourcing is that eventual consistency will be a major problem. This is no more of a problem for Event Sourcing as it is for any other pattern of storing data, and handling it will depend on your use case. Depending on the event store, implementation changes (do not have to be) eventually consistent. Most of the stores allow strong consistency when appending events.

Related Terms

Write Model

The write model is responsible for handling the business logic. If you're using CQRS, then this is the place where commands are handled. The write model has two aspects: physical and logical. The physical aspect relates to how and where data is stored, and the logical aspect reflects the business domain recreated in code structures, and can also be referred to as the Domain Model. Contrary to the traditional anaemic model, it not only contains the data but also the methods to change the data and validate it. The logic within the write model should be as close as possible to the business processes; you should be able to read the code and fully understand the business requirements from the code.

Let's take an example: a write model for issuing invoices. It has data, e.g. amounts, vendor names, invoice numbers, and methods like "Issue Invoice". It has rules such as "each invoice number must be unique", "each invoice must contain a vendor name" etc. By having the design of the system derived from the domain, we can keep all the business logic and data in one place. A write model is essential, but a read model is not always needed.

A useful pattern to consider while implementing the write model is the aggregate. It is responsible for maintaining data consistency, and by using it we're making sure that all related data will be stored in a single, atomic transaction. Aggregates are not necessary, but they are very common.

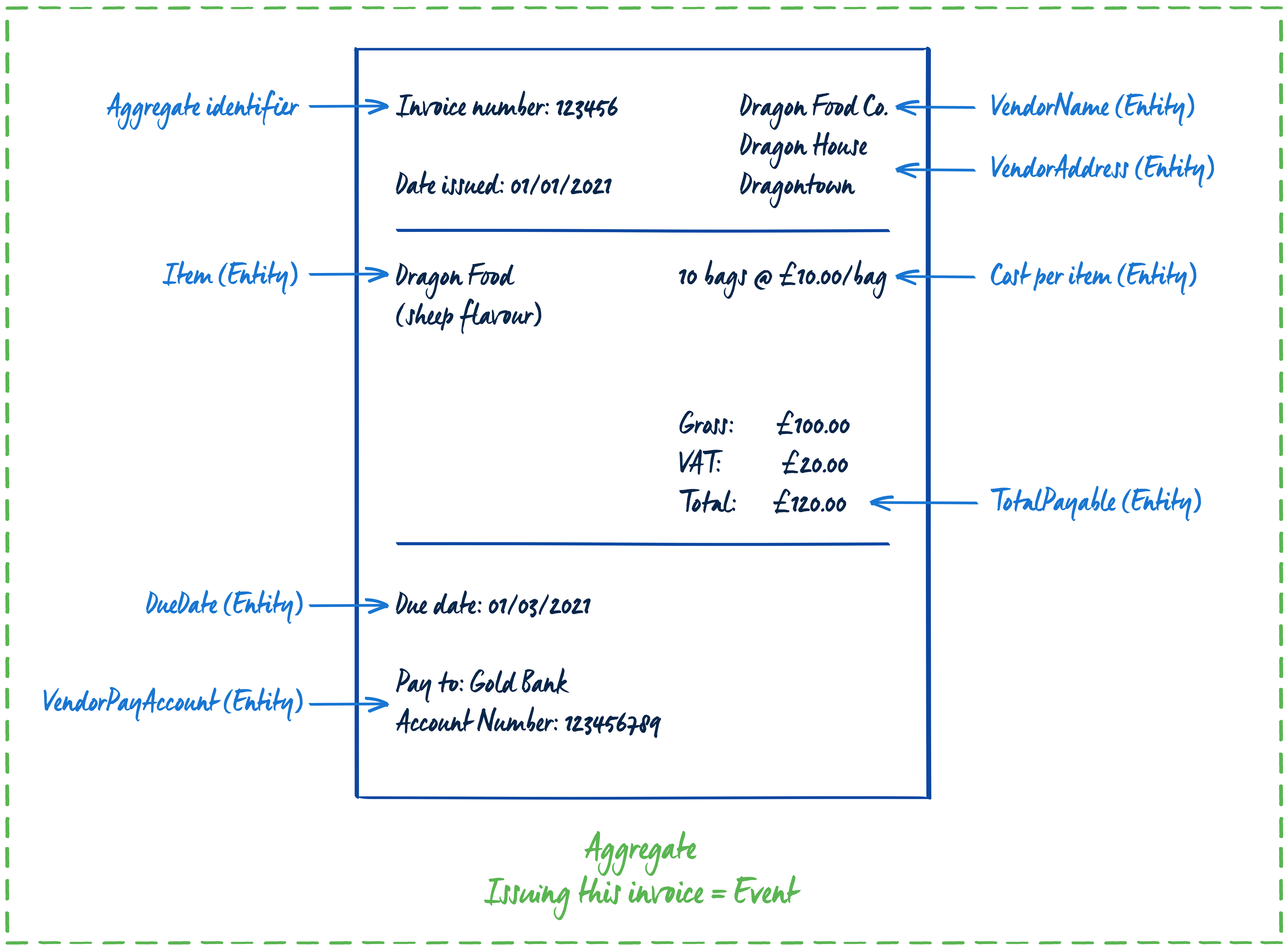
In Domain Driven Design, Eric Evans discusses the granular nature of objects, and provides a definition of an aggregate. From the Domain Driven Design "blue" book:

"An aggregate is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each aggregate has a root and a boundary. The boundary defines what is inside the aggregate. The root is a single, specific entity contained in the aggregate.

Related Terms

The root is the only member of the aggregate that outside objects are allowed to hold reference to, although objects within the boundary may hold references to each other. Entities other than the root have local identity, but that identity needs to be distinguishable only within the aggregate, because no outside object can ever see it out of the context of the root entity."

This can be applied to the invoice example. An invoice is a domain object, but it is also made of several objects: the amount to be paid, the vendor name, the due date, etc. each one is part of the invoice, and the invoice joins them all together. So in the invoice example, the invoice is the aggregate. Each part of the data needed for the invoice (e.g. issuer information) is an entity. The root needs to be a single, specific entity in the aggregate, so this will be the unique invoice number.



Related Terms

Aggregates are the consistency guards. They take the current state, verify the business rules for the particular operation (e.g. if there is no invoice with the same number) and apply the business logic that returns the new state. The important part of this process is storing all or nothing. All aggregated data needs to be saved successfully. If one rule or operation fails then the whole state change is rejected.

In Event Sourcing, each operation made on the aggregate should result with the new event. For example, issuing the invoice should result in the InvoiceIssued event. Once the event has happened, it is recorded in the event store.

It's easy to assume that because there are aggregates, you can have one mega aggregate. However, this will cause more problems in the long run. Smaller, more concise aggregates that focus on one aspect will make a more efficient system overall and preserve the business context where needed. As an example, consider the invoice aggregate. A VAT validation process would not need all the information contained in the invoice, so this process could be a smaller, more concise aggregate. This preserves the business context where it's needed.

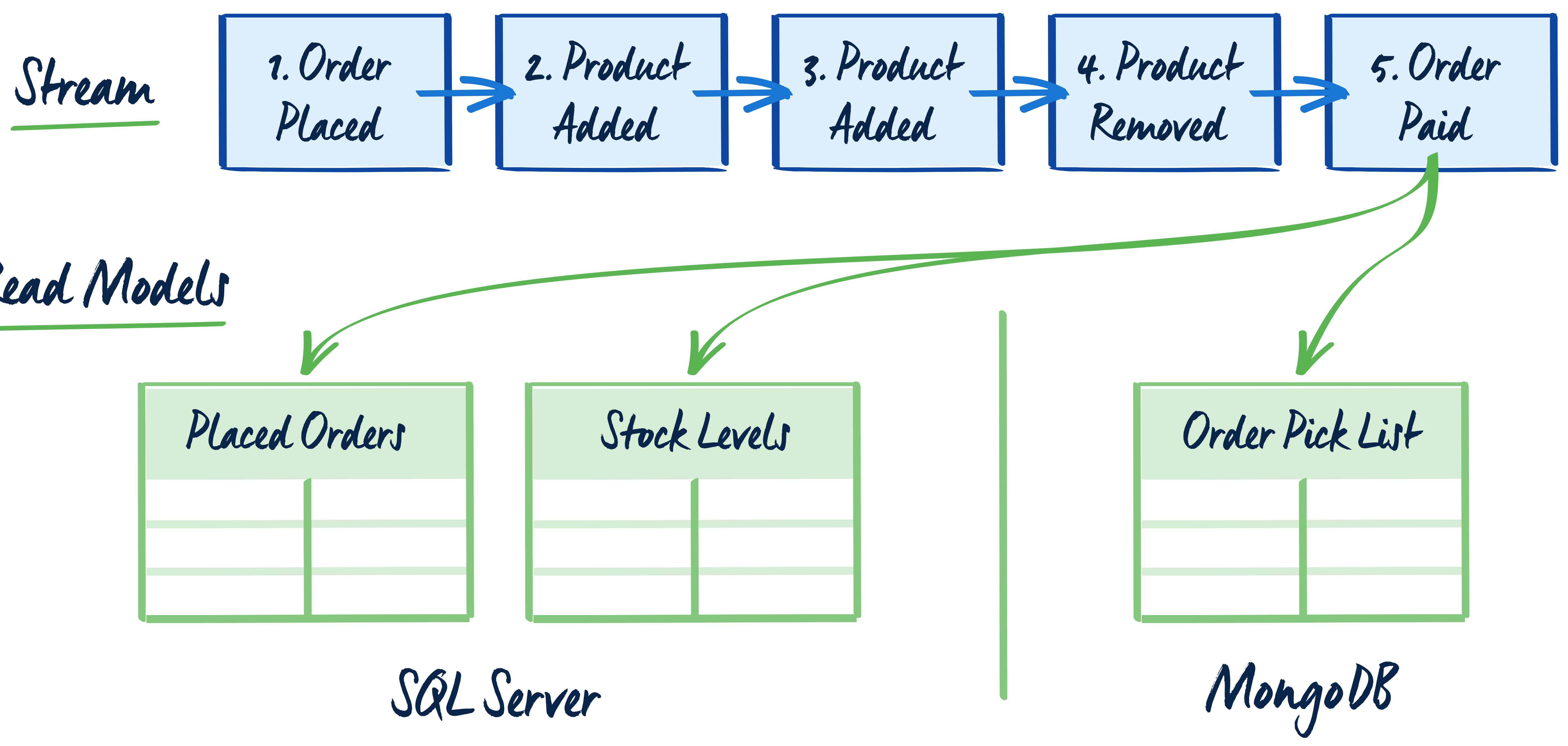
Read Model

Queries in CQRS represent the intention to get data. A read model contains specific information. For example, the read model contains all the invoice information (the amount, the due date, etc) and the query represents the intention to know something about the invoice, such as whether or not it has been paid.

The read model can be, but doesn't have to be, derived from the write model. It's a transformation of the results of the business operation into a readable form.

As stated in the Projections section, read models are created by projections. Events appended in the event store triggers the projection logic that creates or updates the read model.

Related Terms



The read model is specialised for queries. It's modelled to be a straightforward output that can be digested by various readers. It can be presented in many different ways, including a display on a monitor, an API, another dependent service or even to create the PDF of the invoice. In short, a read model is a general concept not tied to any type of storage.

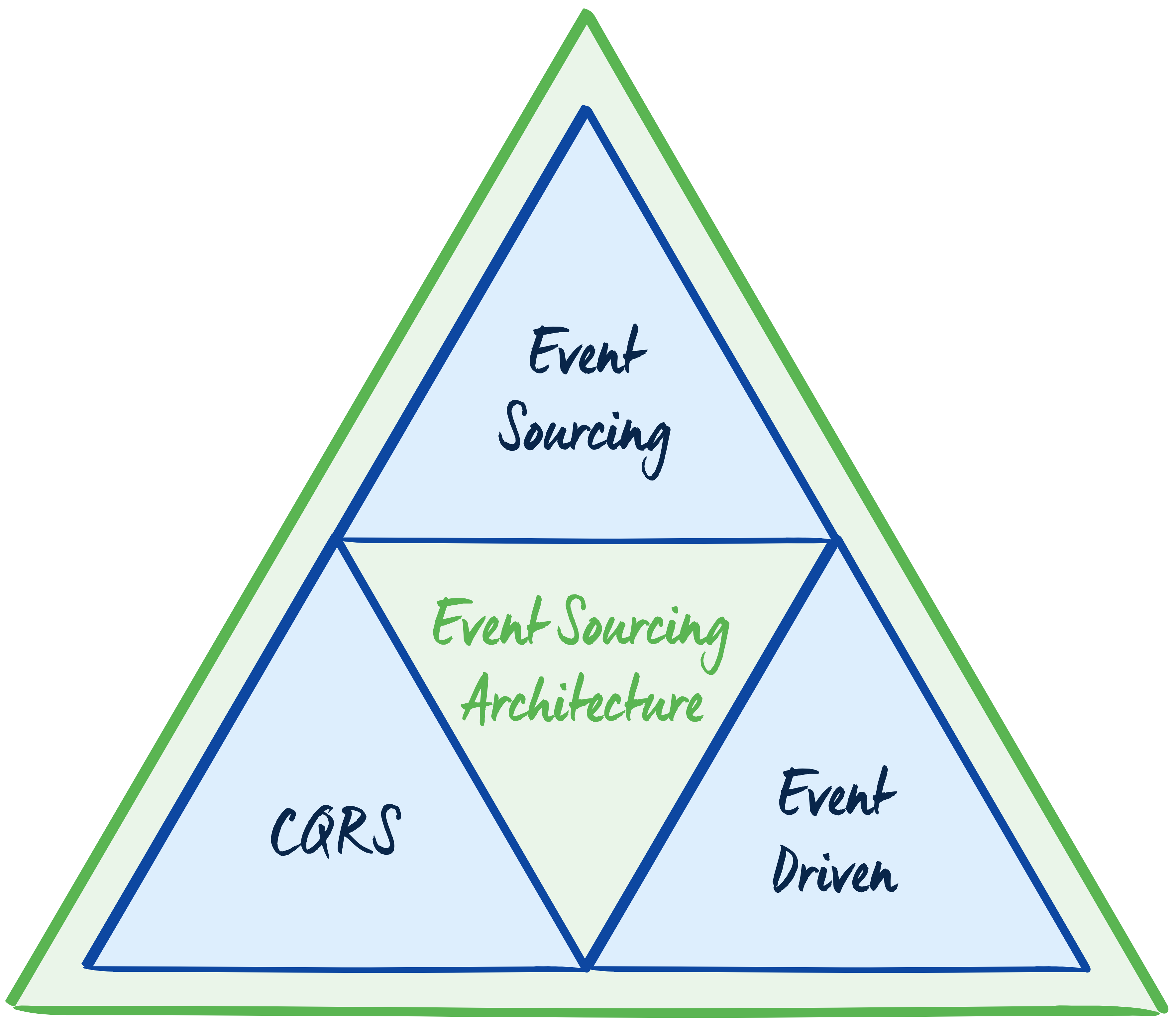
However, the read model does not have to be permanent: new read models can be introduced into the system without impacting the existing ones. Read models can be deleted and recreated without losing the business logic or information, as this is stored in the write model.

Event Sourcing Architecture

Event Sourcing is an architectural design pattern that stores data in an append-only log. It is part of a wider ecosystem of design patterns that work together in various ways to allow developers to create the most effective architecture for their needs.

Related Terms

By deploying an event store as a design pattern within a wider architecture, it allows for the inclusion of other design patterns in the system that are the most suitable for the needs of the domain. For example, an event-sourced system works well within a CQRS architecture, however it's not necessary to use them together. Event Sourcing can be deployed alongside event driven architecture, or in conjunction with CQRS, an aggregate pattern, a command handler pattern, or one of many other patterns. Each individual pattern is a useful tool on its own, that can work in concert with other related patterns to make stronger, more specific patterns for your specific use case.



An event store can be a key element of a system, and that system can be as simple or as complex as the business domain requires it to be. It's useful to consider putting an event-sourced system in a part of the architecture that requires the preservation of context for all events, as this is where Event Sourcing is most effective.

Related Terms

Domain Driven Design

Domain Driven Design (DDD) is a method for optimizing a team's understanding of a problem space, and how to work in that space. At its core, it's about having a ubiquitous language between the terms used by the business users and the development team. This unification of language can be extremely useful when translating the problem concept into functioning software.

Domain Driven Design was first coined by Eric Evans in his seminal work Domain-Driven Design: Tackling Complexity in the Heart of Software. Evans took his experience and fused the business process and the implementation as a first step in the design process, then created a ubiquitous language based on that first step. He realised the importance of using a common language to collaborate efficiently on the project.

Domain Driven Design is important because it requires meaningful and continued collaboration between the architect and the product owner. It takes the developer away from the purely technical and theoretical world, and imposes a reality on their development skills. It also forces the product owner (or customer) to think through what they require from the system and recognise the capabilities of it. By making all the stakeholders cooperate, a shared understanding is created, and progress is more efficient.

The creation of a ubiquitous language involves Knowledge Crunching, the process of taking unrelated terms from business and development and creating something from the scattered terms. It's collaborative, it can be messy, but can also be the beginning of a beautiful friendship.

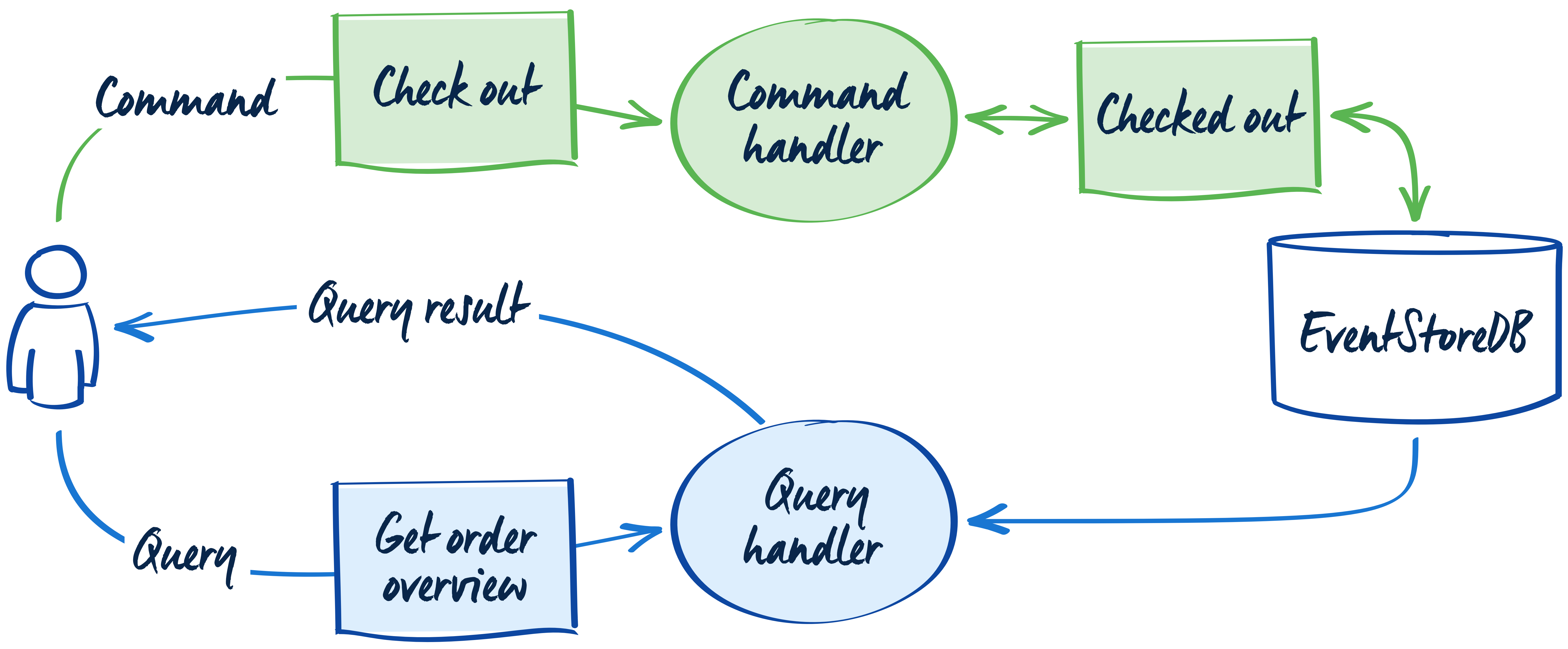
Using Domain Driven Design with Event Sourcing is not mandatory. However, the main concepts such as speaking the same language as the business and proper business process modelling are also good foundations for building an Event Sourcing system. The better understanding we have of the business processed, the more precise the business information will be in our events.

Related Terms

CQRS

CQRS is a development of CQS, which is an architectural pattern, and the acronym stands for Command Query Separation. CQS is the core concept that defines two types of operations handled in a system: a command that executes a task, a query that returns information, and there should never be one function to do both of these jobs. The term was created by Bertrand Meyer in his book 'Object-oriented Software Construction' (1988, Prentice Hall).

CQRS is another architectural pattern acronym, standing for Command Query Responsibility Segregation. It divides a system's actions into commands and queries. The two are similar, but not the same. Oskar Dudycz describes the difference as "CQRS can be interpreted at a higher level, more general than CQS; at an architectural level. CQS defines the general principle of behaviour. CQRS speaks more specifically about implementation".



Related Terms

CQRS

One of the common misconceptions about CQRS is that the commands and queries should be run on separate databases. This isn't necessarily true; only that the behaviours and responsibilities for both should be separated. This can be within the code, within the structure of the database, or (if the situation calls for it), different databases.

Expanding on that concept, CQRS doesn't even have to use a database: It could be run off an Excel spreadsheet, or anything else containing data.

Event sourcing and CQRS seems like a new and interesting design patterns, but they have been around for some time. CQRS can feel like a strange new technique, especially after spending years learning with a foundation based in CRUD. CQRS was described by Greg Young as a 'stepping stone towards Event Sourcing'. It's important to understand event sourcing and CQRS do not rely on each other; you can have an event sourced system without CQRS and you can use CQRS without event sourcing. It's just that the two work best together.

More Resources

You can find more resources about Event Sourcing on our website and YouTube channel, and by following us on LinkedIn and X.

www.eventstore.com/blog

www.youtube.com/eventstoreltd

https://www.linkedin.com/company/eventstore/

https://x.com/eventstore

EVENT STORE®

www.eventstore.com