

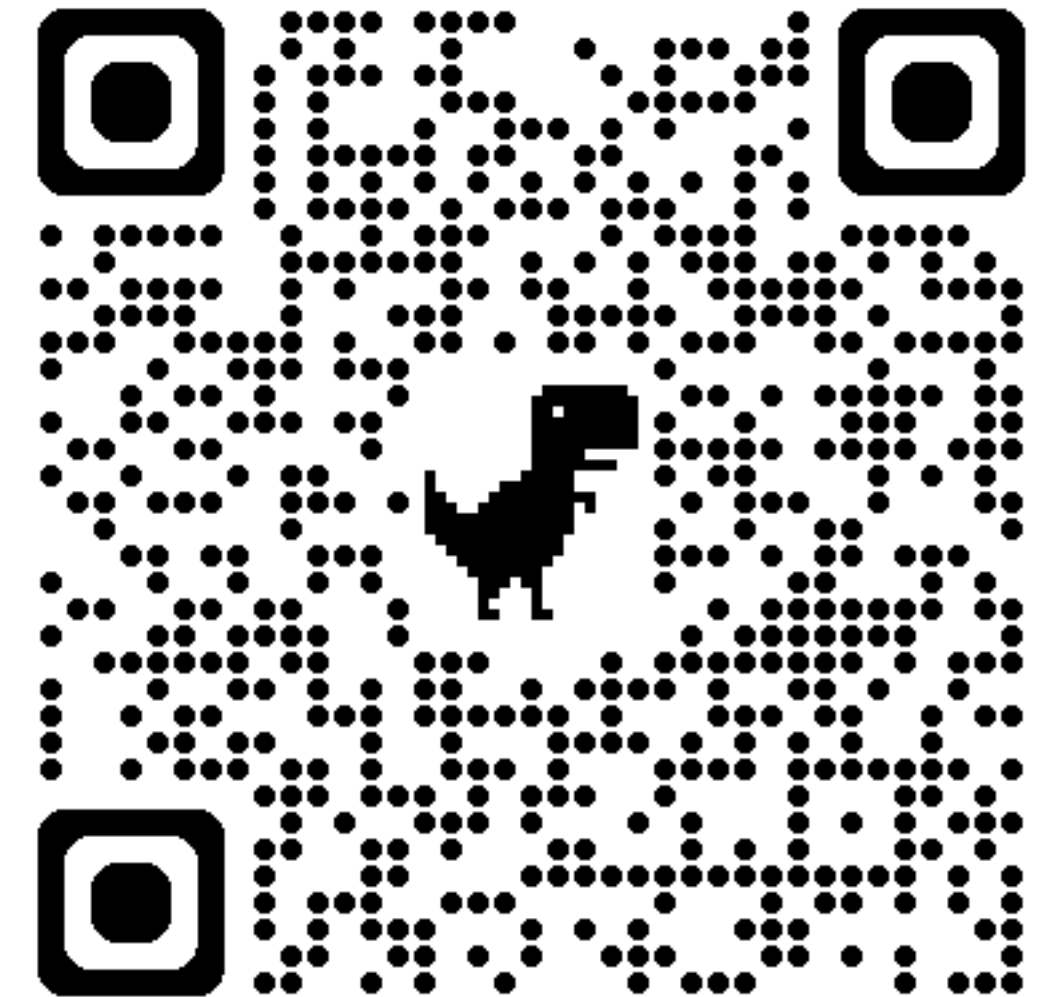
Architectural Patterns Focus: Messaging

Daniel Hinojosa

Design Pattern Messaging Standards

In this Presentation

- Quick introduction to Domain Driven Design
- Event Sourcing / Event Driven Architecture
- Claim Check
- Materialized Views
- CQRS
- Saga Pattern
- Types of Events
- DataMesh



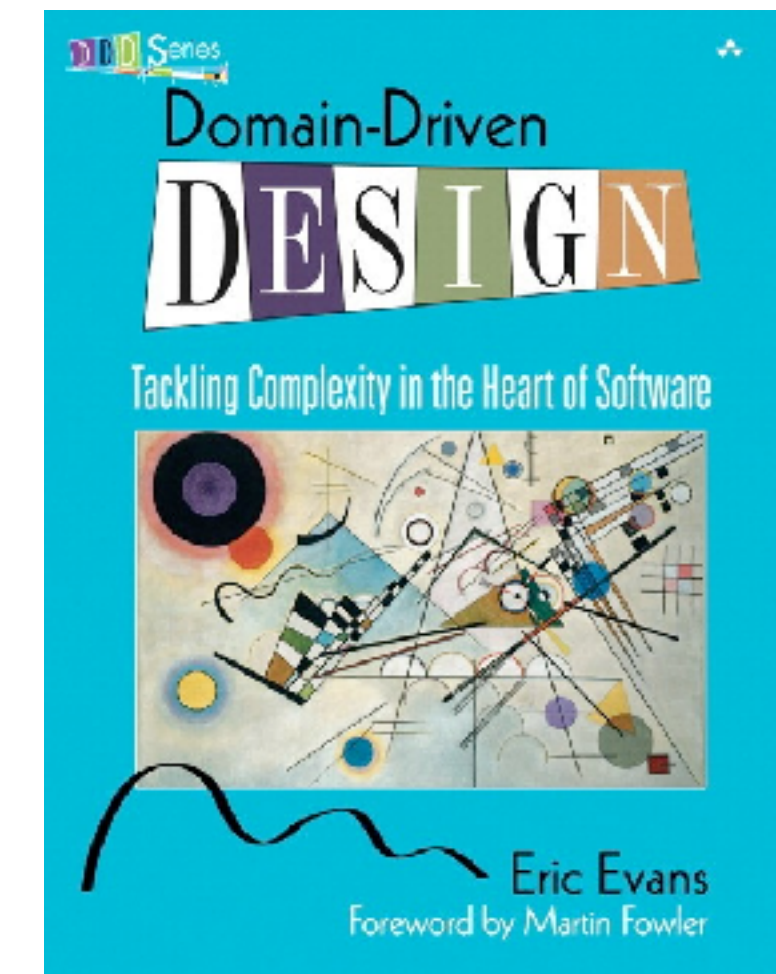
Slides and Material: <https://github.com/dhinojosa/nfjs-architectural-patterns-messaging>

Quick Introduction to DDD



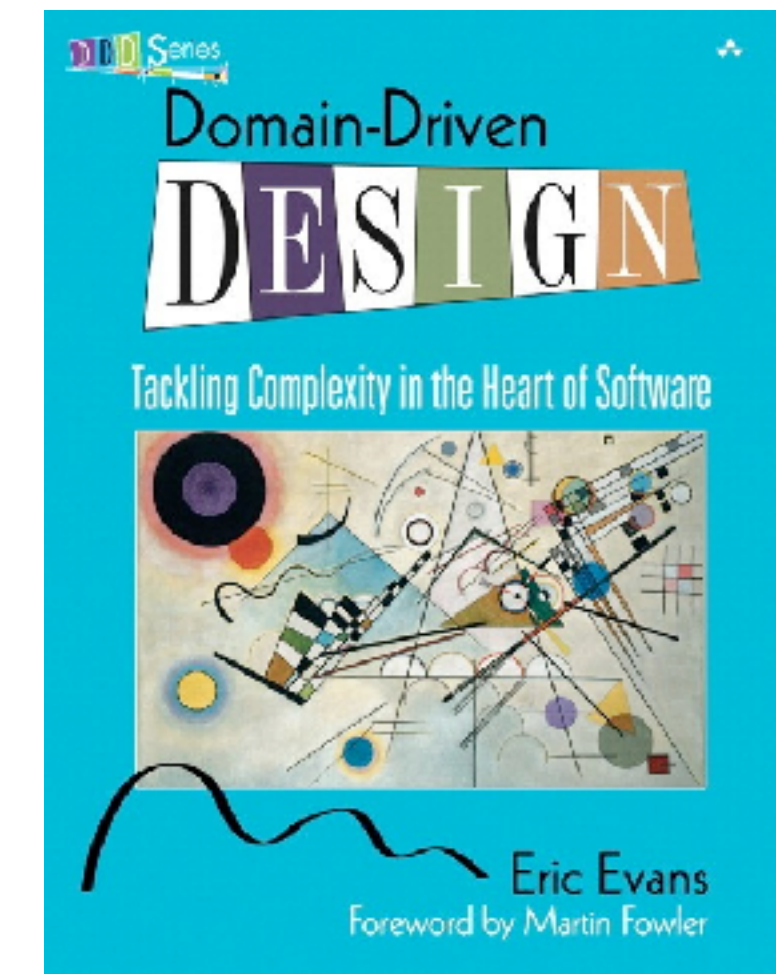
Domain Driven Design

- Methodology focused about the important of the domain
- Build an *ubiquitous language* for domain
- Classifies Objects into Entities, Aggregates, Value Objects, Domain Events, and Service Objects
- One of the other important aspects to DDD is the notion of a bounded context and subdomains



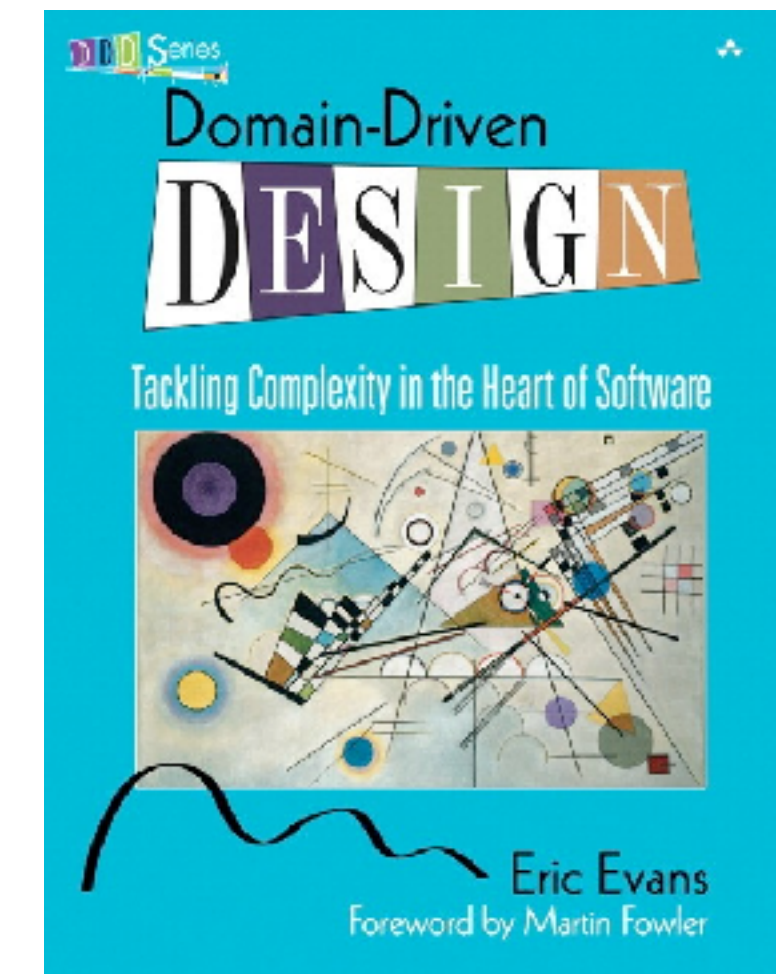
Ubiquitous Language

- Cornerstone of Domain Driven Design
- A dictionary of the same language
- A language for describing your business domain
- Represents both the business domain and the domain experts' mental models.



Bounded Contexts

- Divide the Ubiquitous Language into multiple smaller languages
- Assign each team to the explicit context in which it can be applied: its bounded context



What goes in the bounded context?

- Developers will then develop the following software artifacts:
 - Value Objects
 - Entities
 - Aggregates
 - Domain Services
 - Application Services

Value Objects

- Object that can be identified by the composition of its values
- No explicit identification is required
- Changing the attributes of any of the fields yields a different object
- Value Objects can be used to counter the “Primitive Obsession” code smell
- Typically immutable

```
class Color {  
    int red;  
    int green;  
    int blue;  
}
```


Entities

- Opposite of a value object and requires explicit identification
- Explicit identification is required
- For example, an Employee, just identified by an employee's first name and last name
- Entities are subject to change

```
class Employee {  
    private Name name;  
    //Constructors, equals, hashCode, etc.  
}
```



```
class Employee {  
    private EmployeeId employeeId;  
    private Name name;  
    //Constructors, equals, hashCode, etc.  
}
```


Aggregates

- An entity that manages a cluster of domain objects as a single unit
- Consider an `Order` to `OrderLineItem` relationship
- The root entity ensures the integrity of its parts
- Transactions should not cross aggregate boundaries
- They are domain objects(`order`, `playlist`), they are not collections, like `List`, `Set`, `Map`

```
class Album {  
    private Name name;  
    private List<Track> tracks;  
  
    public void addTrack(Track track) {  
        this.tracks.add(track);  
    }  
  
    public List<Track> getTracks() {  
        return Collections.copy(tracks)  
    }  
}
```


Domain Services

- Stateless object that implements the business logic
- It naturally doesn't belong to any of the domain model's aggregates or value objects

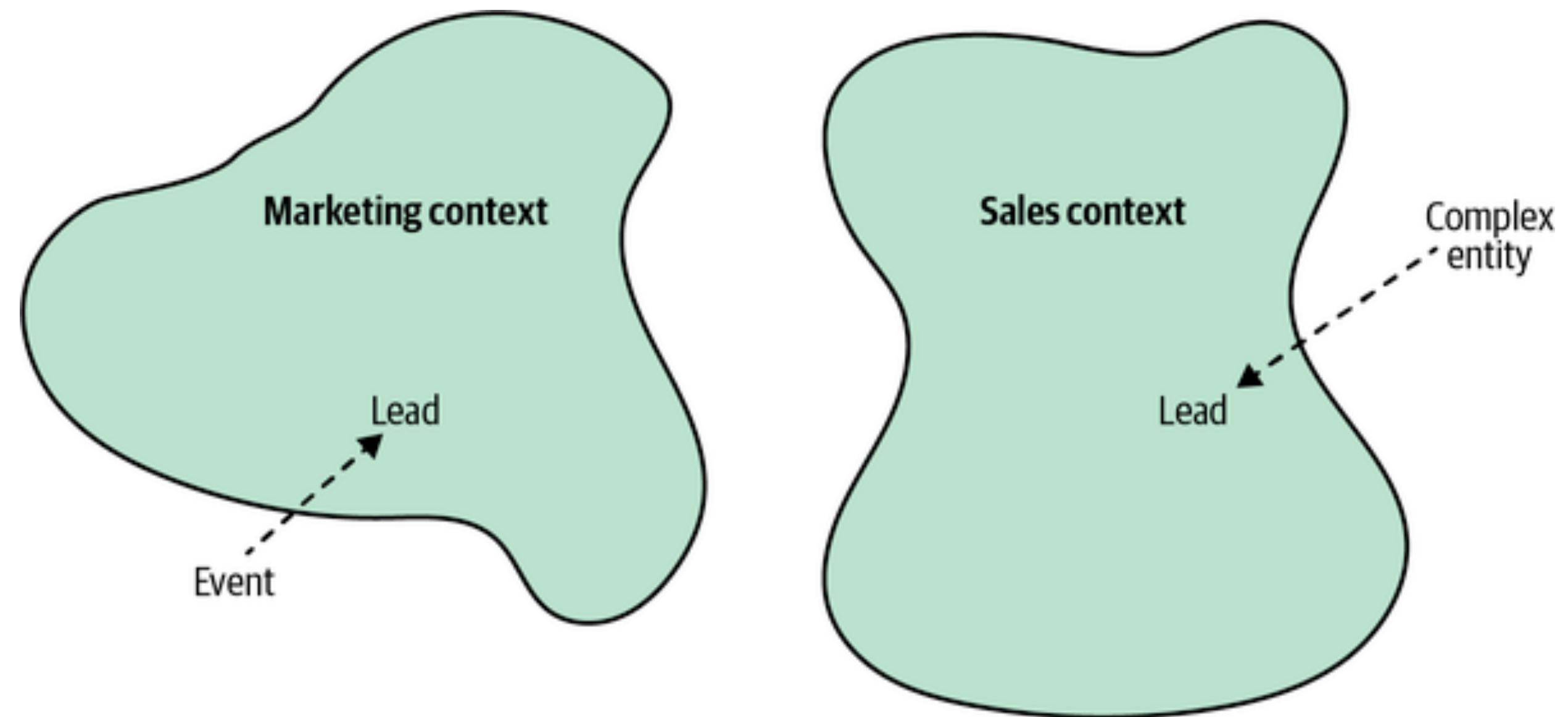
```
class OrderTax {  
    //no state  
    private void calculateTaxWithRate(Order  
        order, TaxRate taxRate) {  
        //...  
    }  
}
```


Application Services

- Makes use of Repositories (Storage Abstractions)
- Aggregate instances and then sent for transforming to a transformed object
- The transformed object will typically be routed to the UI

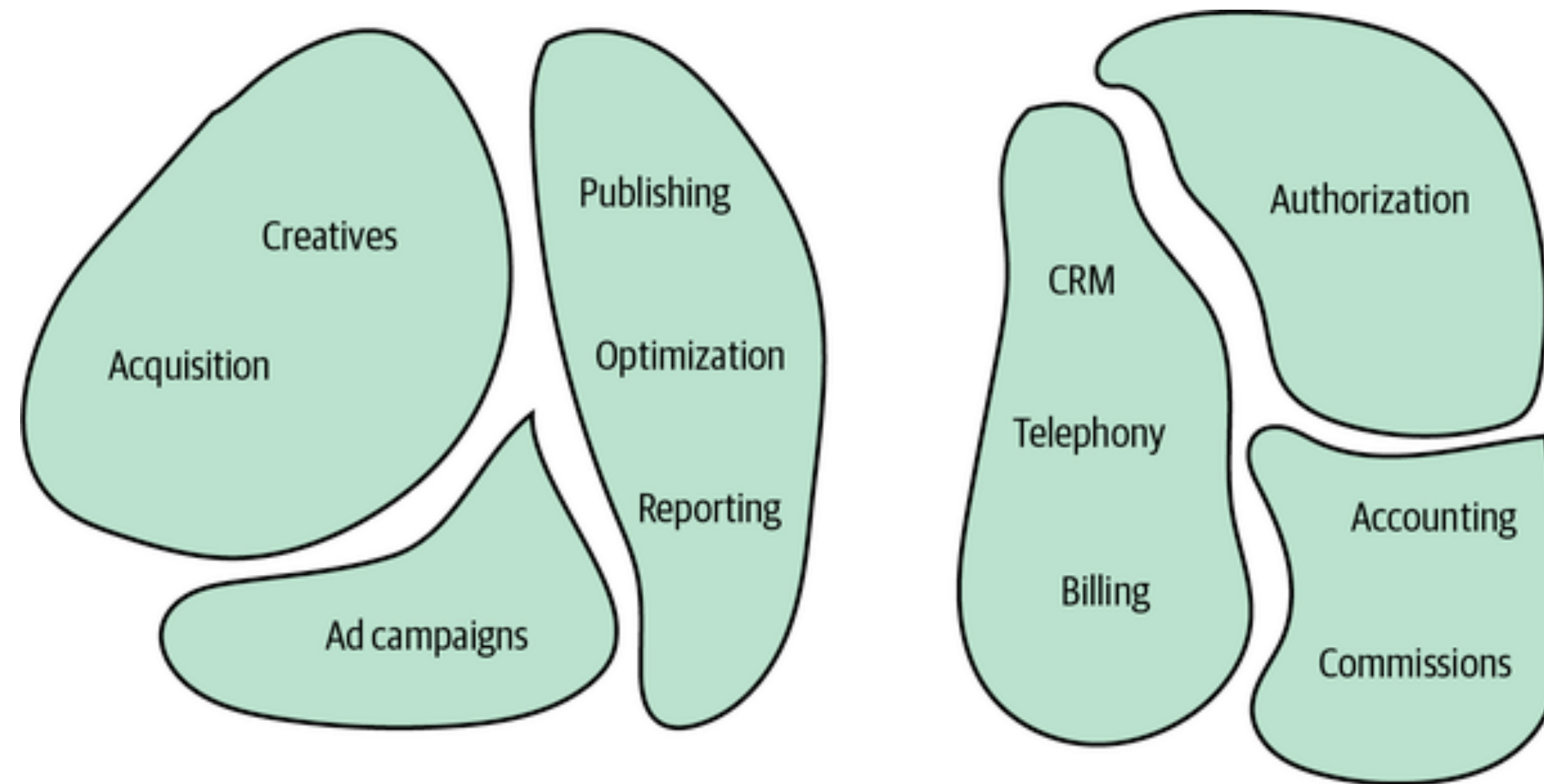
```
class OrderService {  
    private OrderRepository  
        orderRepository;  
    private void persistOrder(Order) {  
        //...  
    }  
}
```


The Diagram



The Diagram

Contexts are neither big nor small, but useful



Event Storming

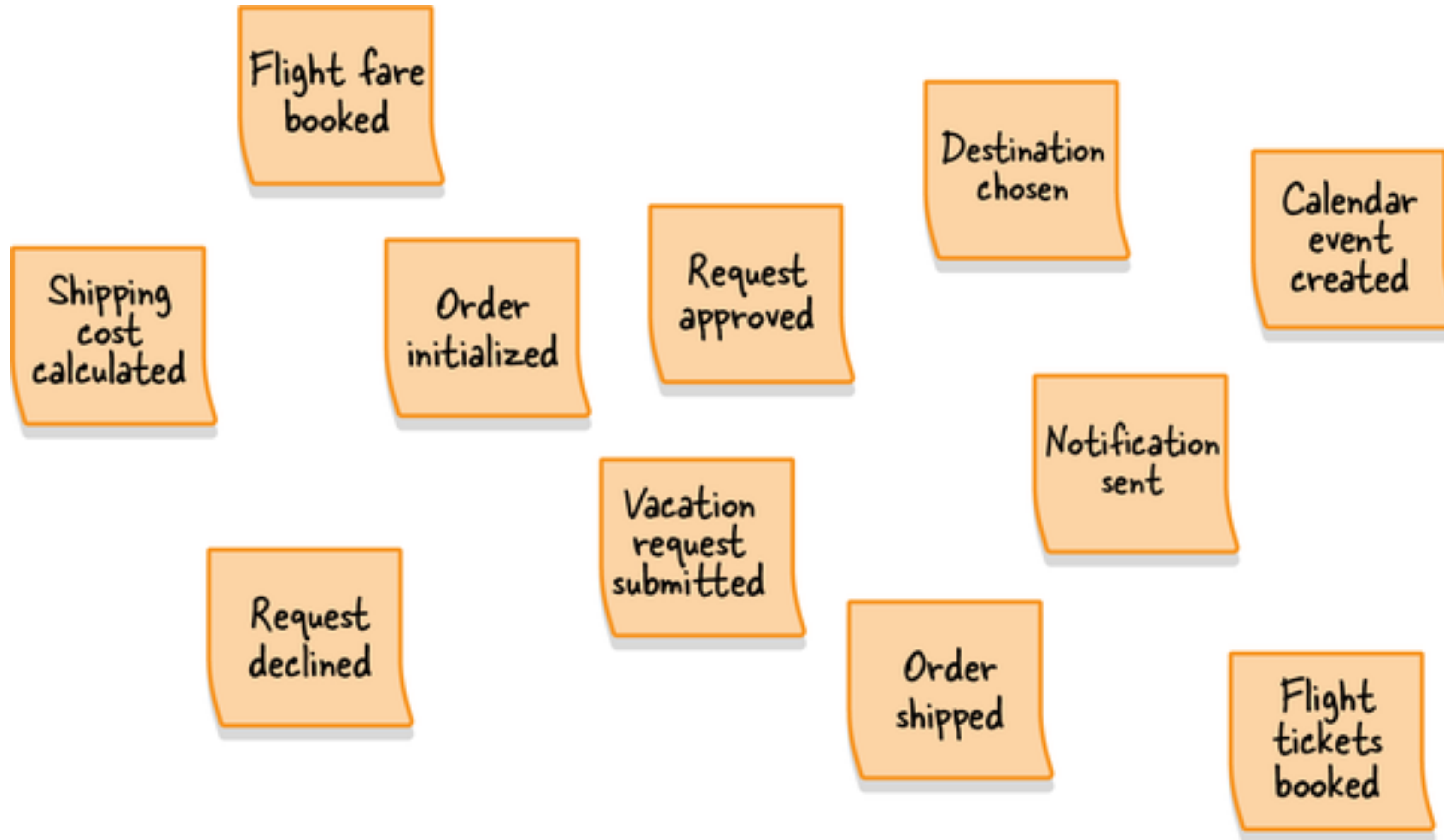


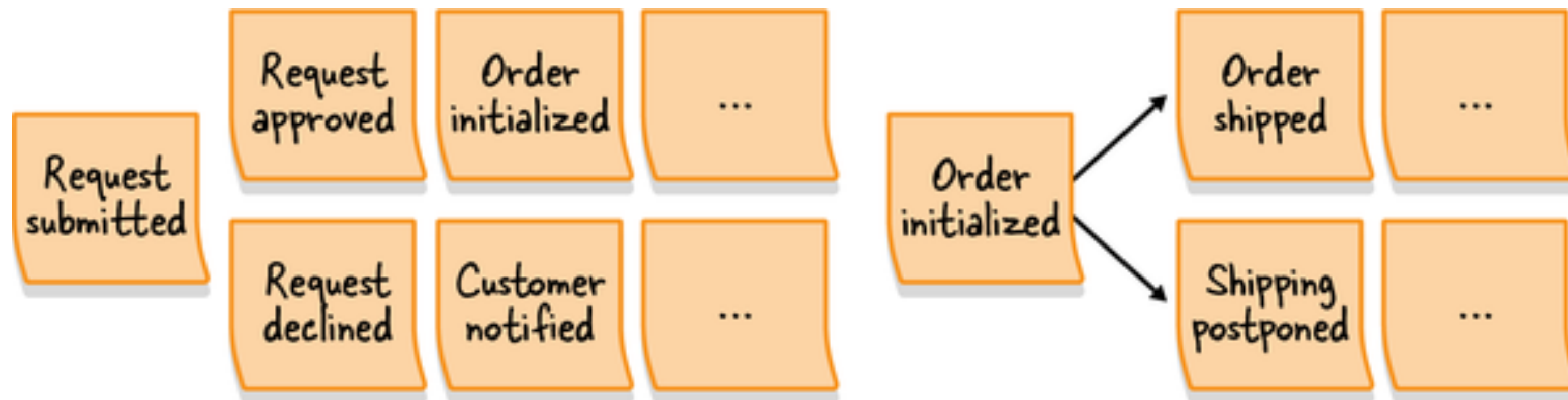
Event Storming

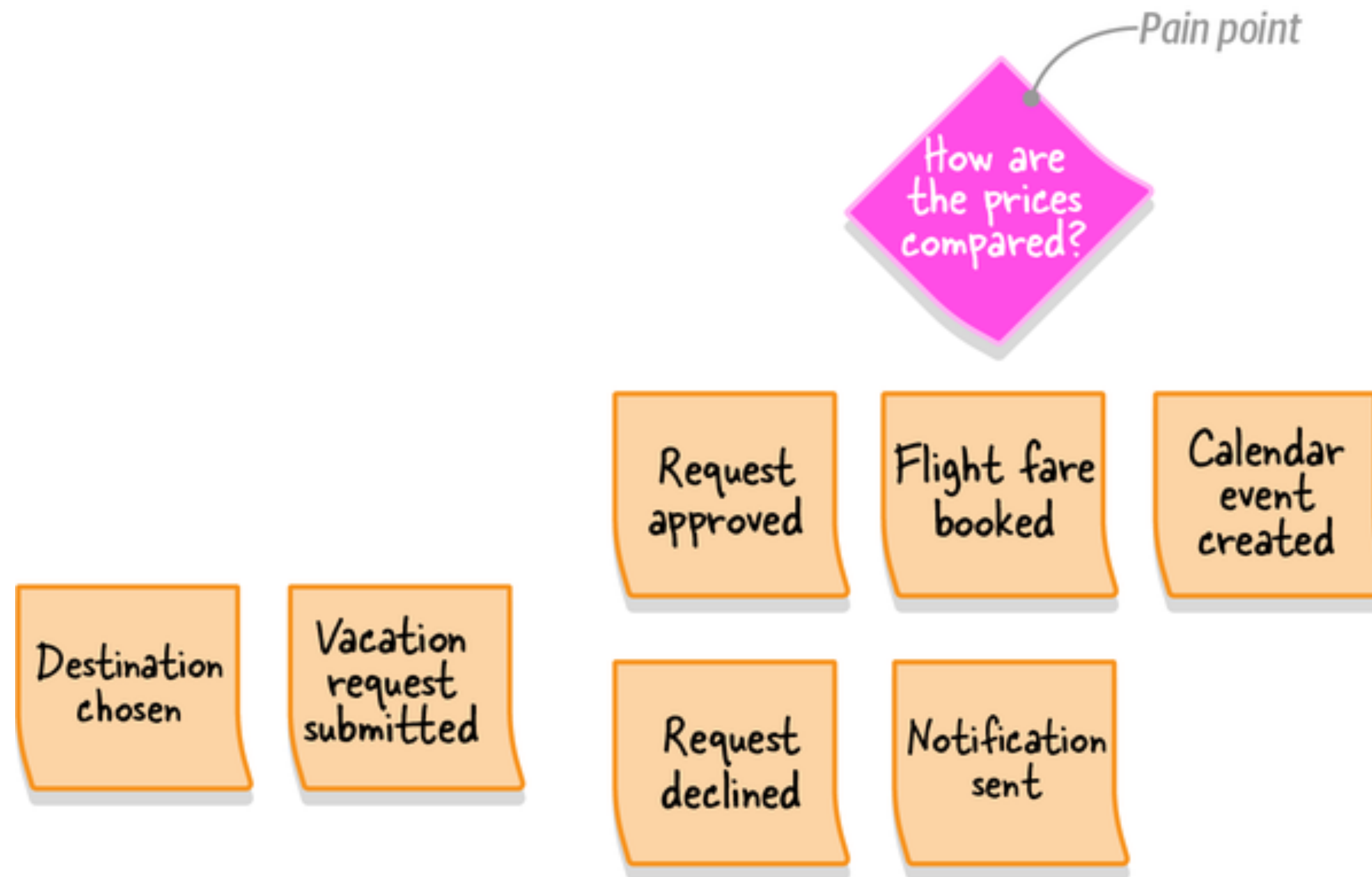
- Low-tech activity for a group of people to brainstorm and rapidly model a business process.
- In a sense, EventStorming is a tactical tool for sharing business domain knowledge.
- An EventStorming session has a scope: the business process that the group is interested in exploring.
- The participants are exploring the process as a series of domain events, represented by sticky notes, over a timeline. Step by step, the model is enhanced with additional concepts—actors, commands, external systems, and others—until all of its elements tell the story of how the business process works.

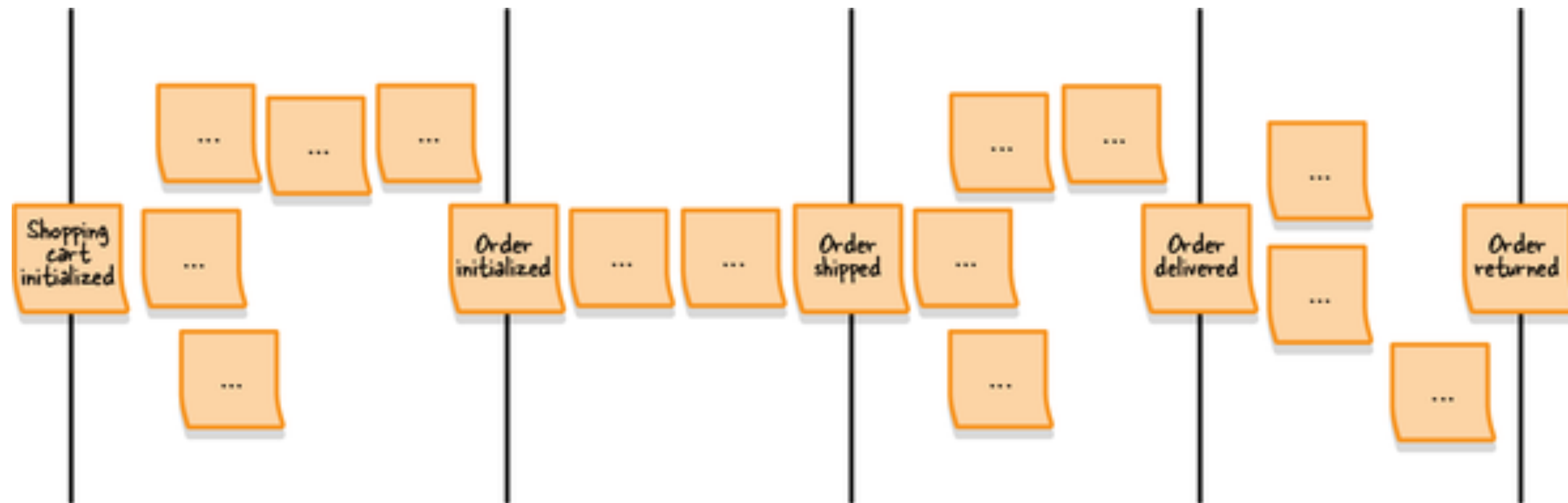
“Just keep in mind that the goal of the workshop is to learn as much as possible in the shortest time possible. We invite key people to the workshop, and we don’t want to waste their valuable time”

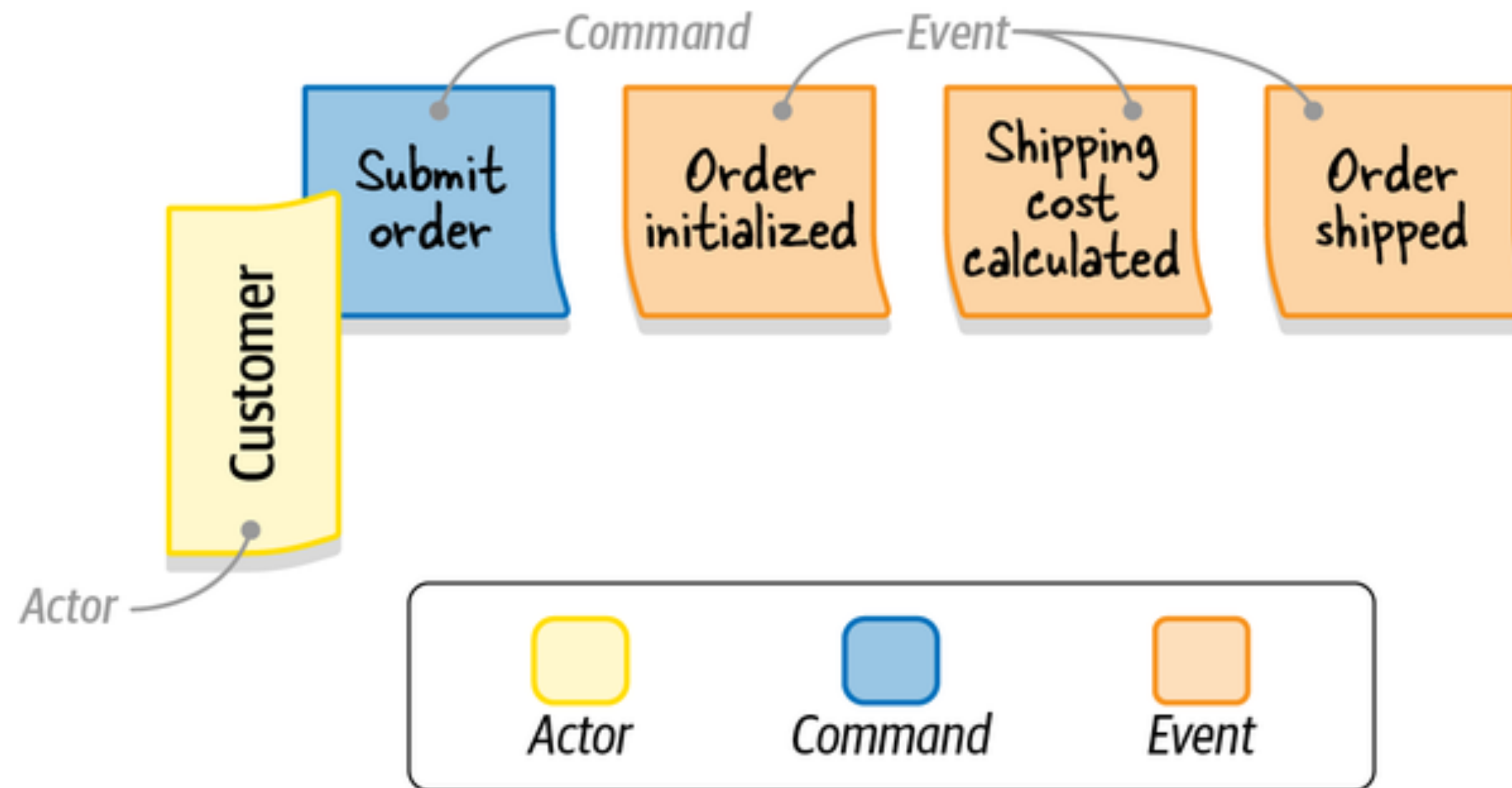
— Alberto Brandolini: Creator of the EventStorming workshop

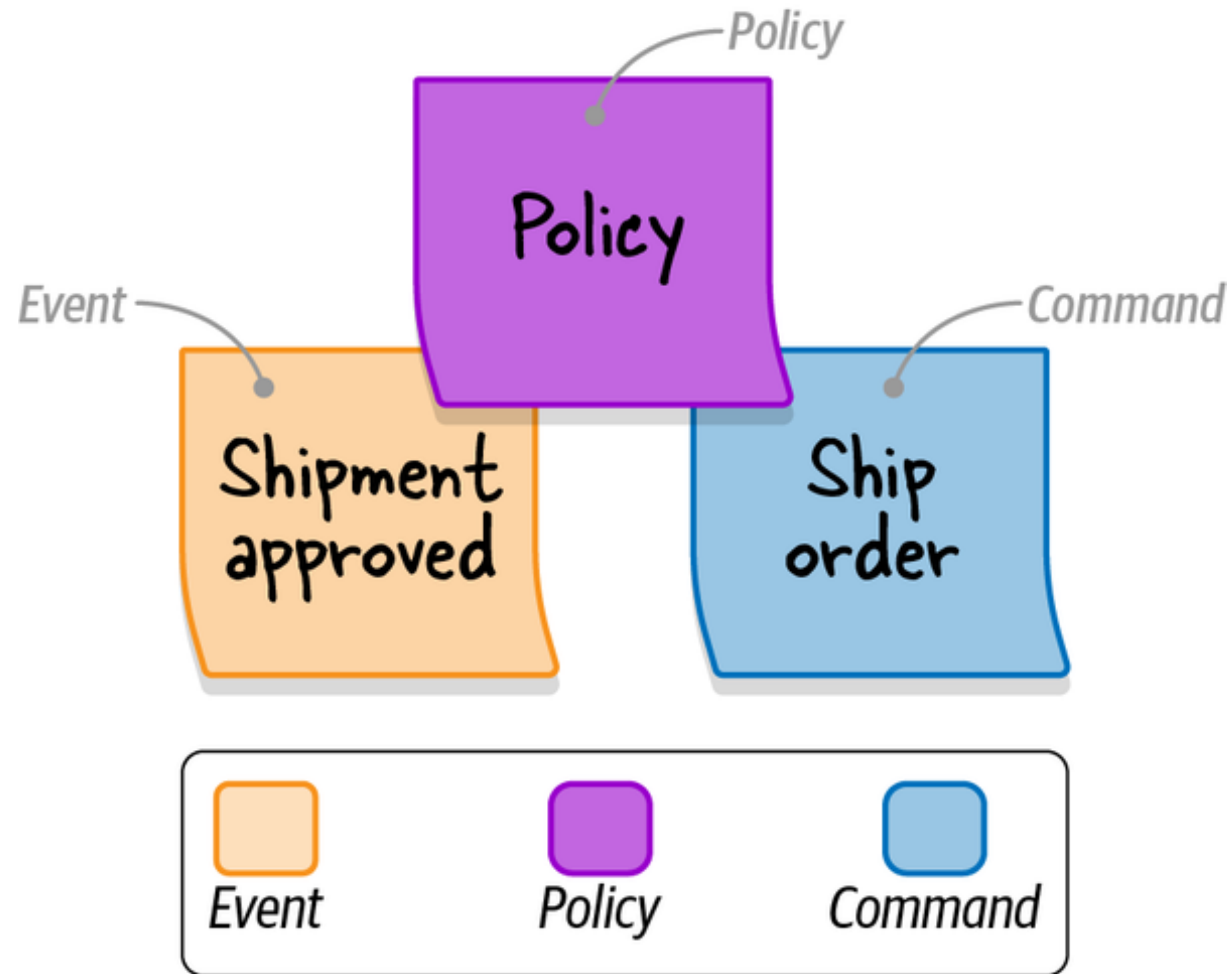


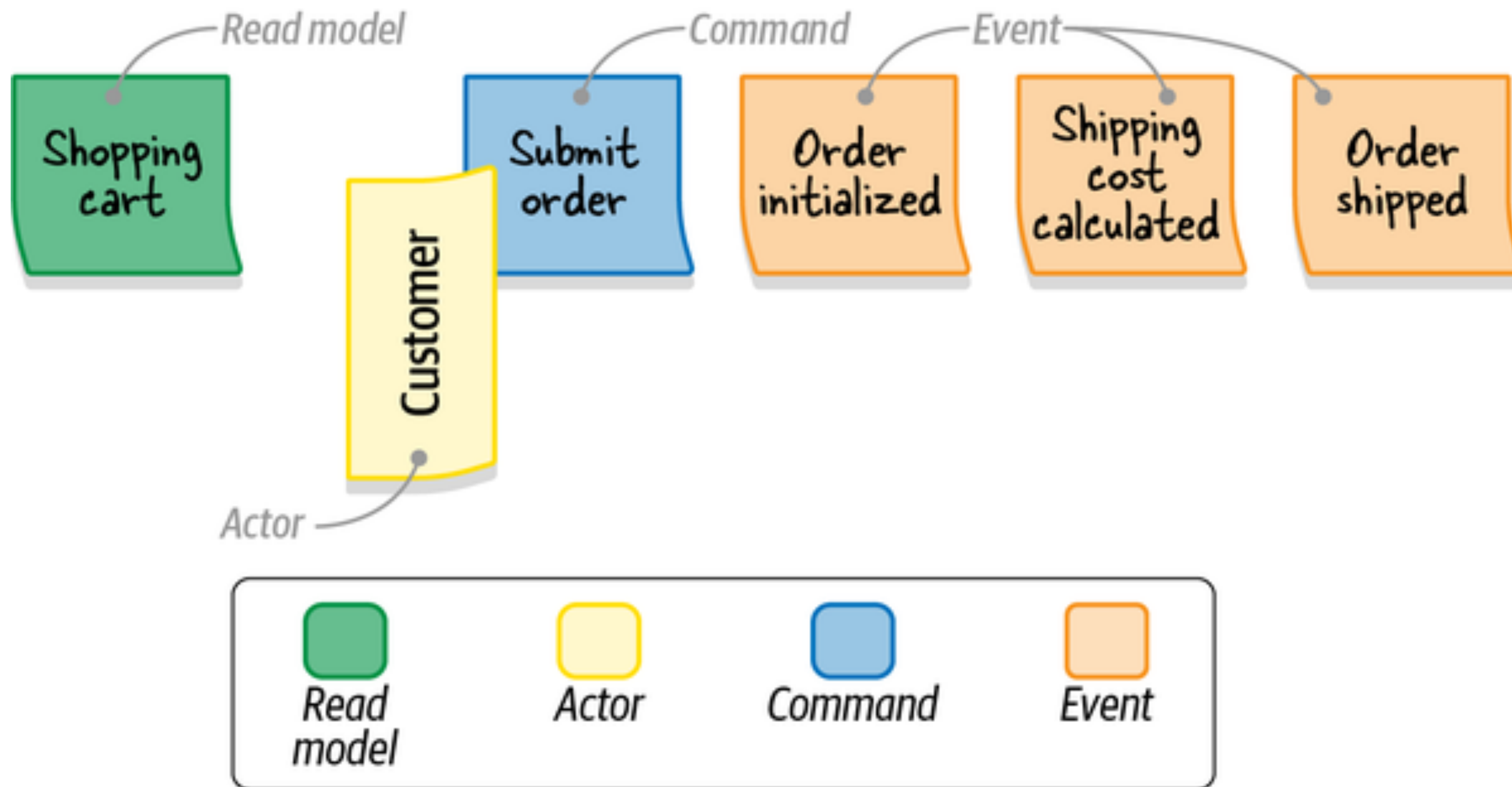








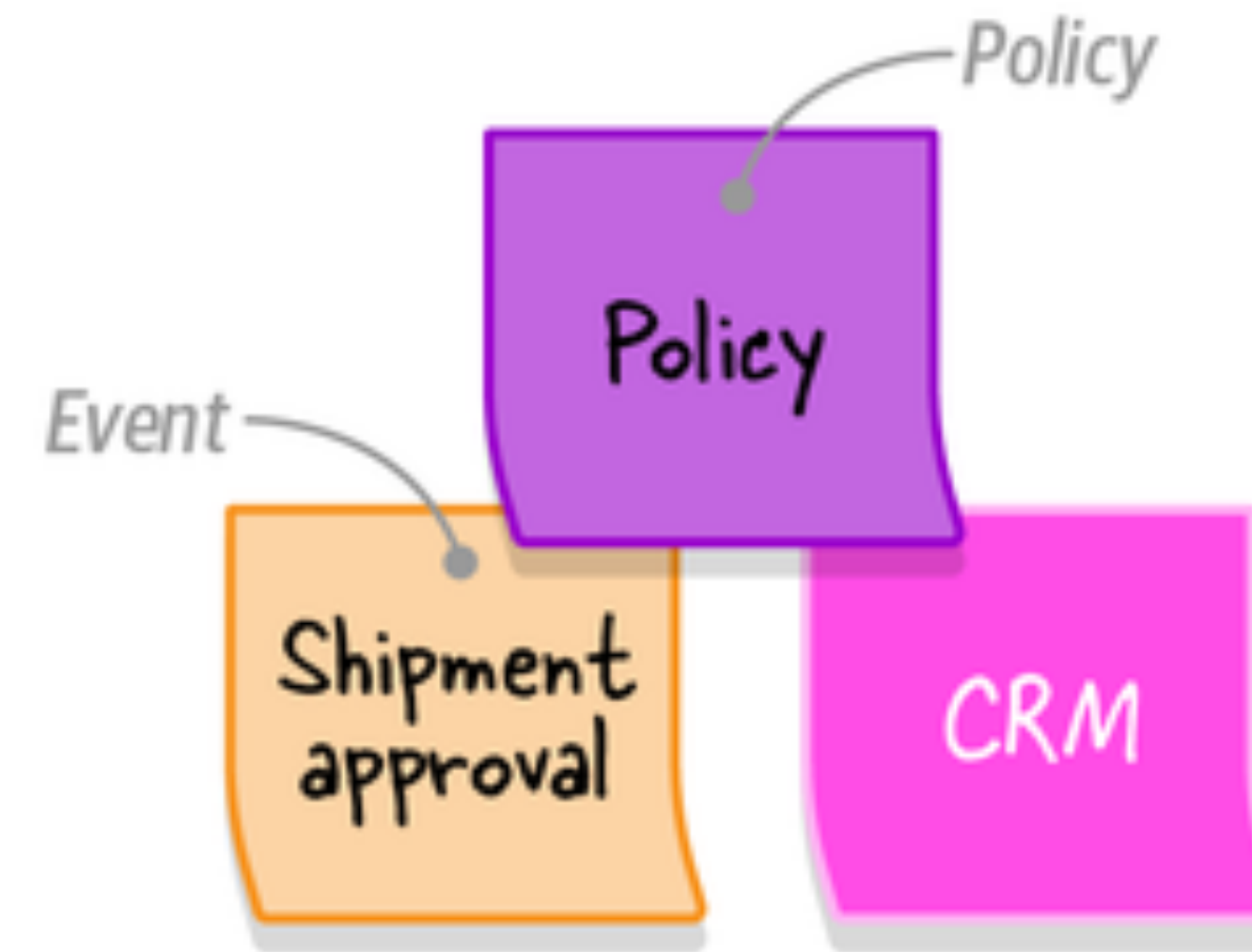


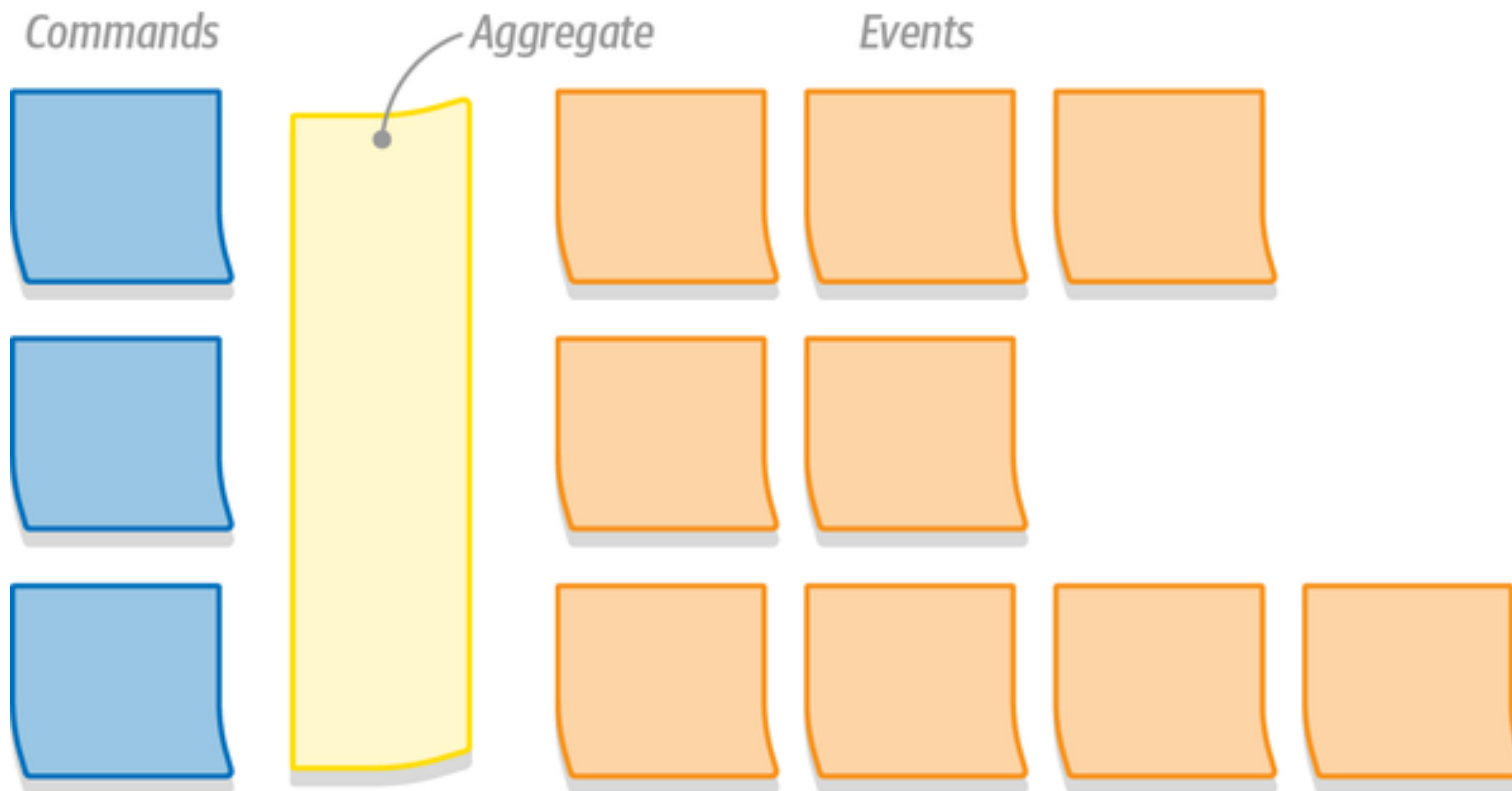


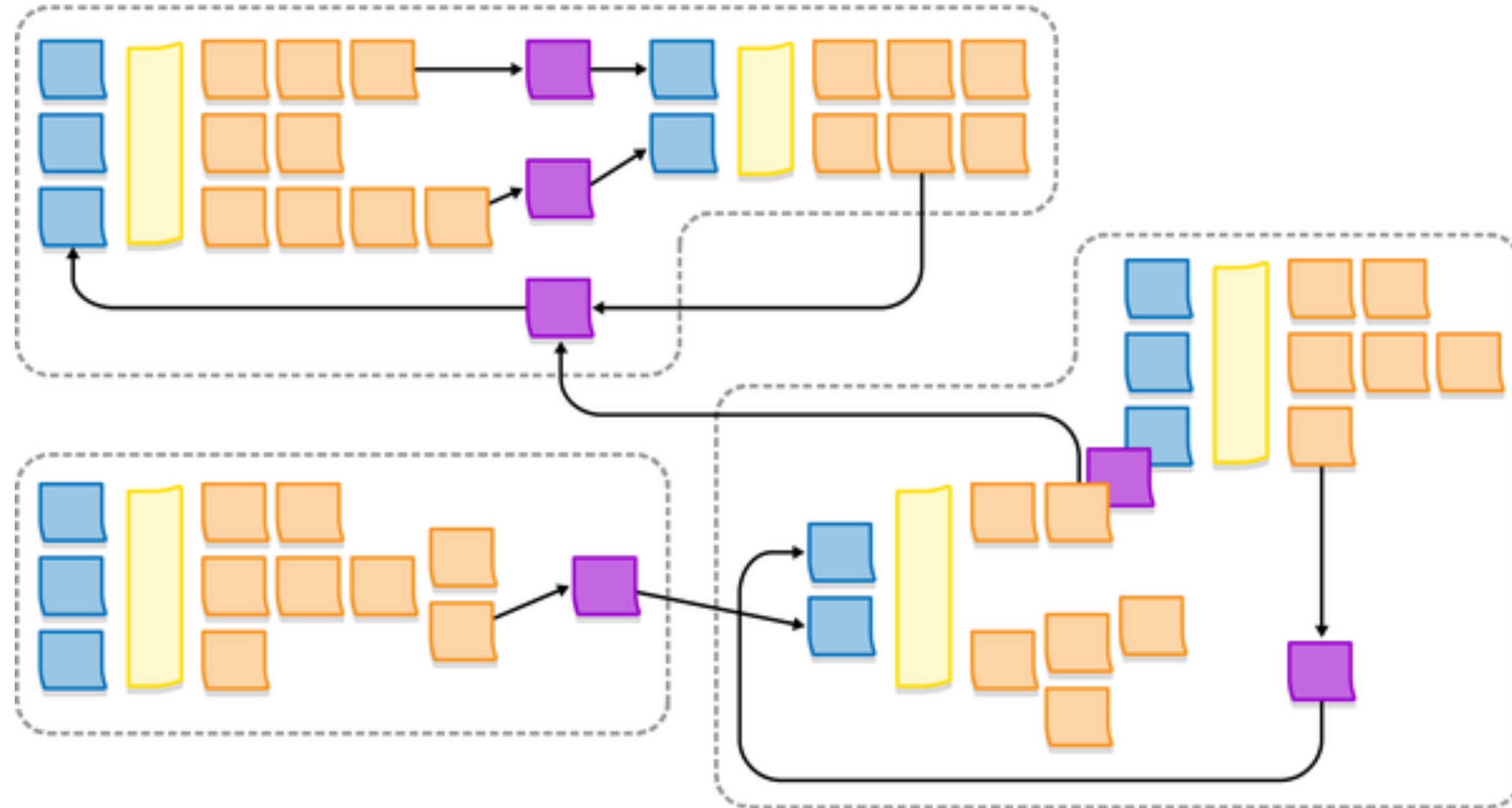
An external system executes a command



Notifying an external system







Event Sourcing



The Problem

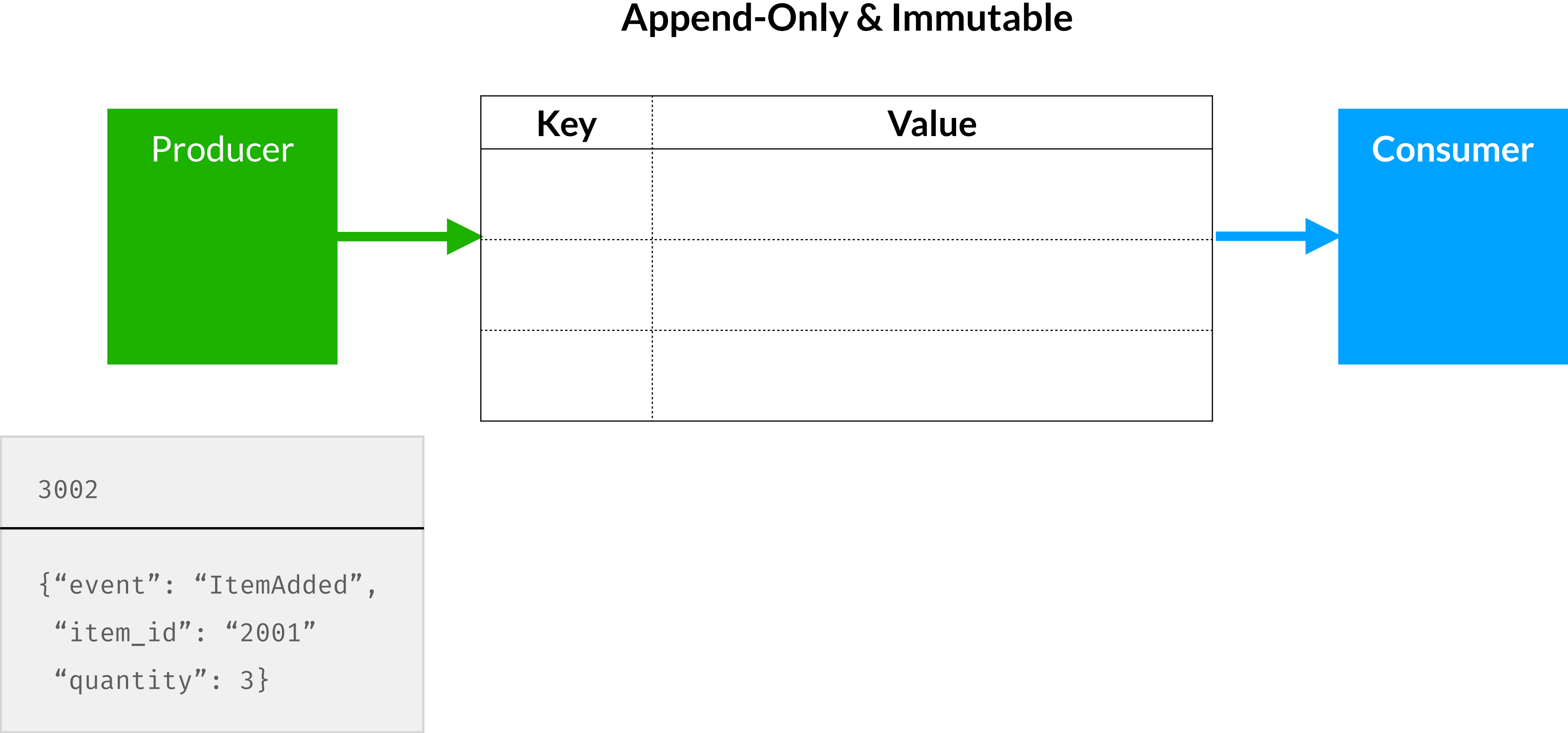
- Typical CRUD Database have issues
 - It is a change log where information get overridden, and not all data is like that
 - Slowdown of performance, when writing to a typical datastore there can be contention and lack of scalability
 - Unless there is an implementation of auditing much of this data is lost
- Event Sourcing compliments CQRS (to be covered soon)

The Solution

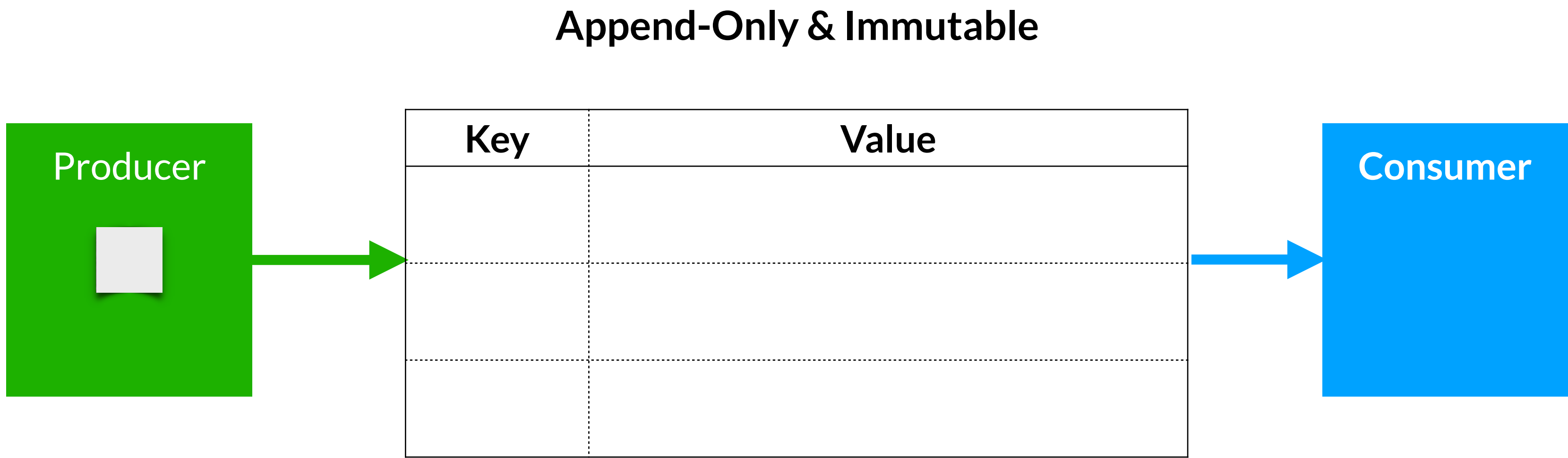
- Events are recorded in an append only store
- Events are described with discrete events like:
 - AddedItemToCart
 - RemovedItemFromCart
 - ClearedCart
- Therefore it can be used to materialize view, prepare the data for the purposes of being efficiently read by a UI or business analytics for consumption
- Remember the orange sticky notes?



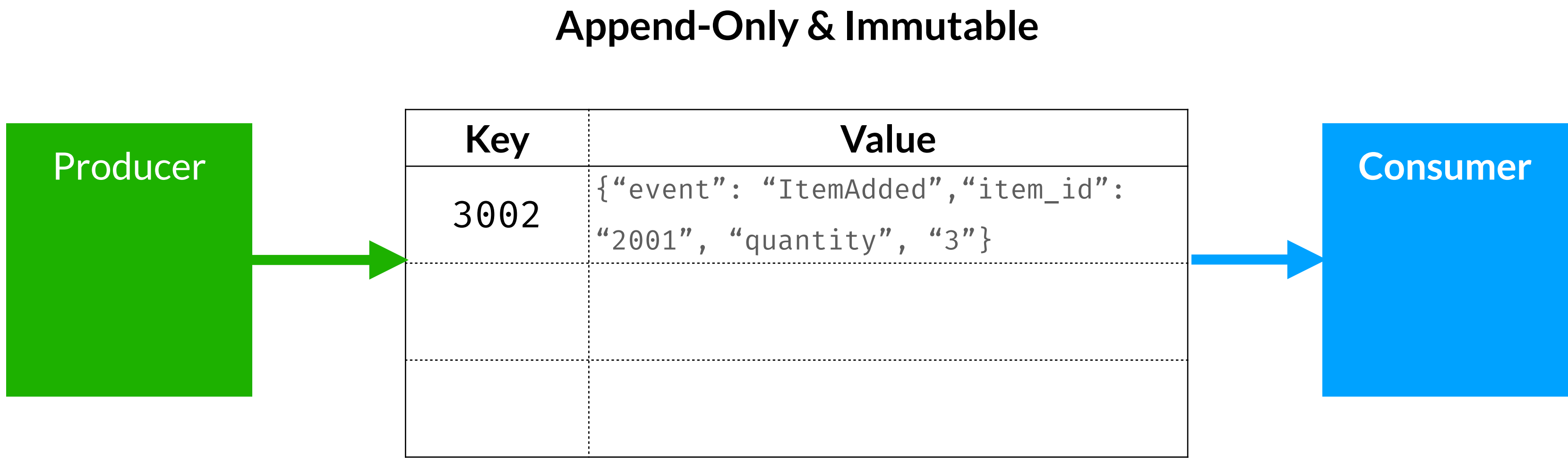
The Diagram



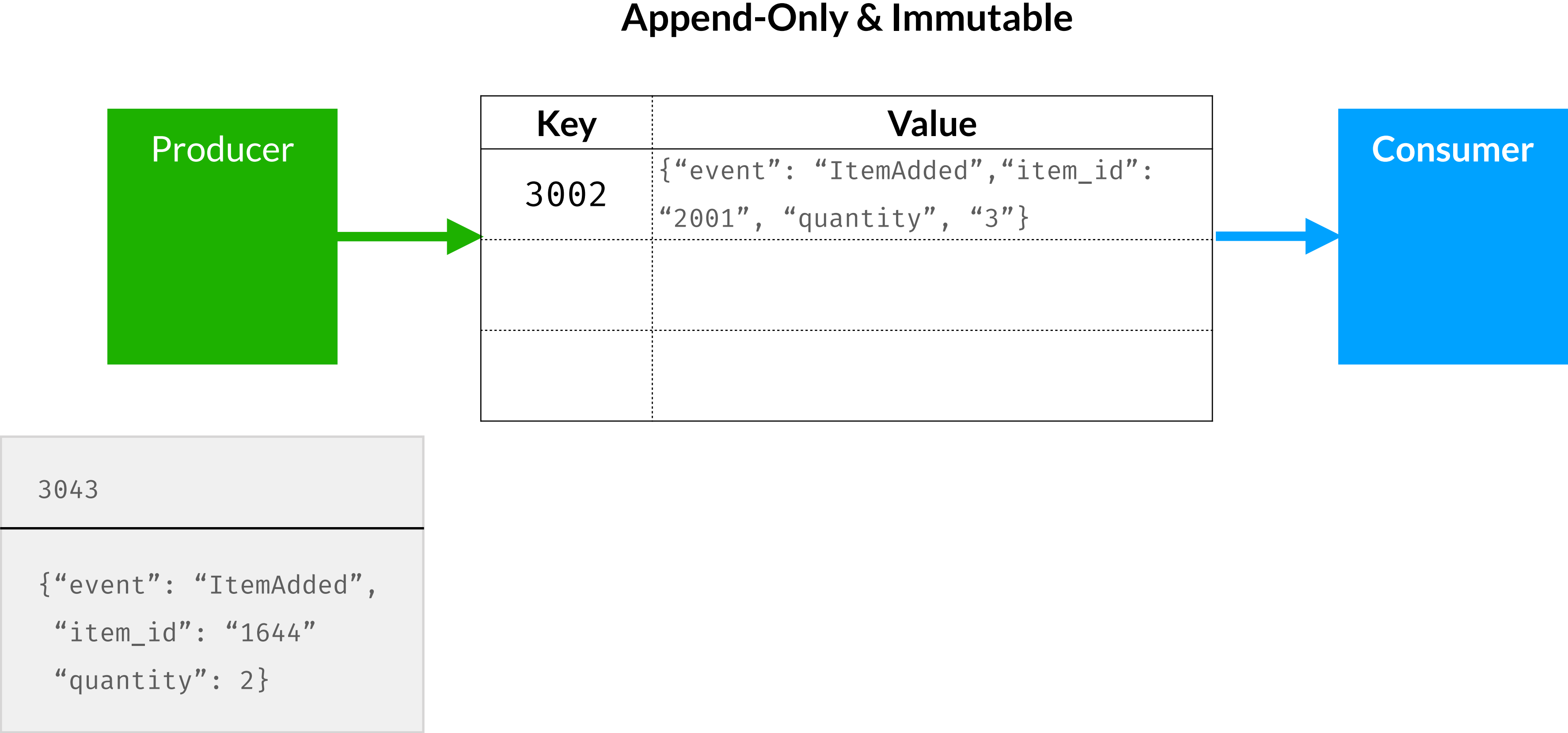
The Diagram



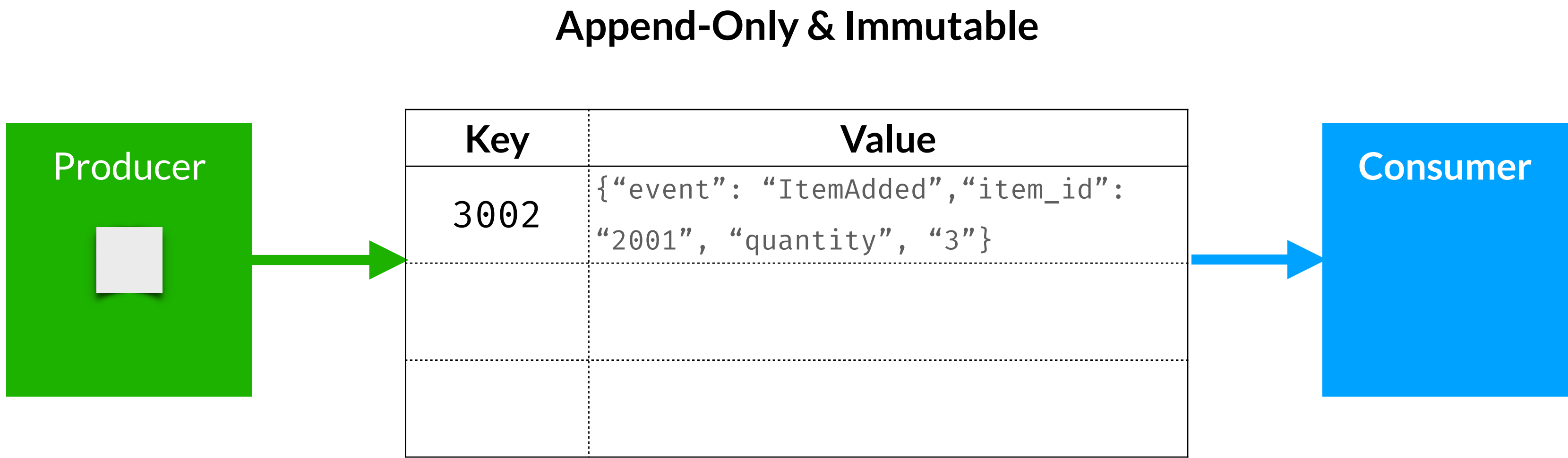
The Diagram



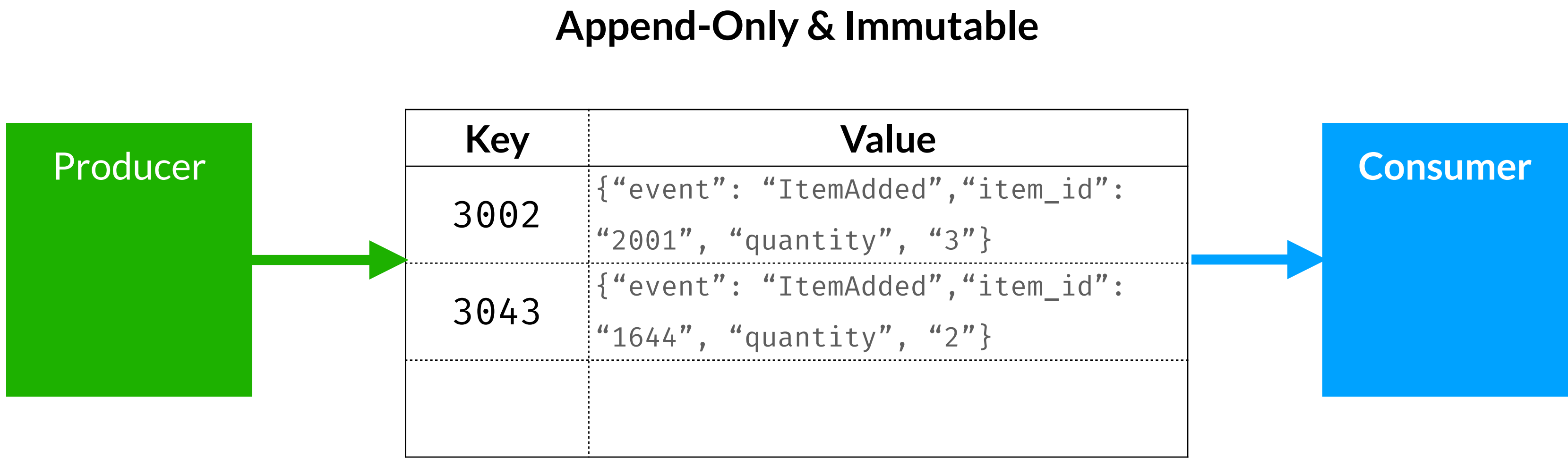
The Diagram



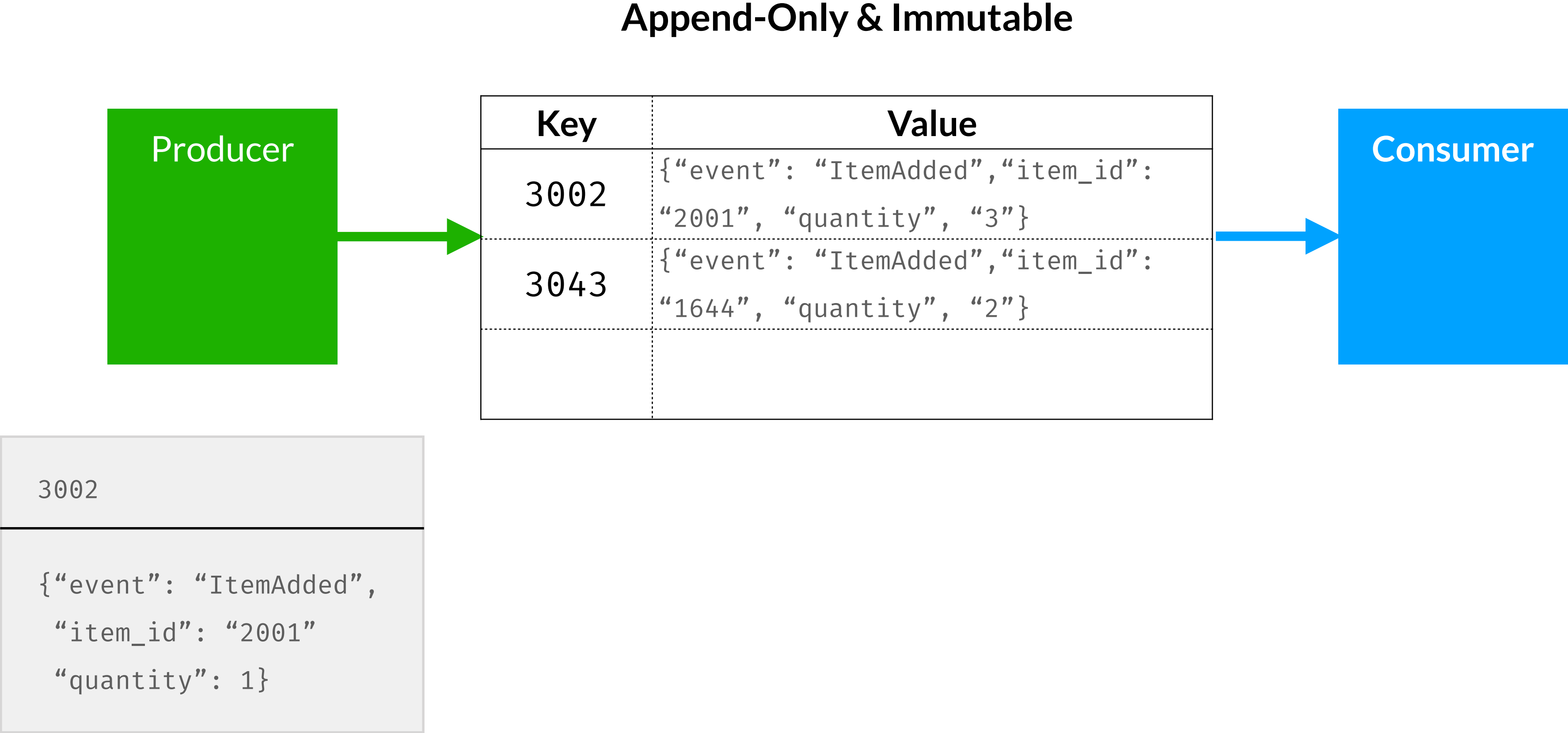
The Diagram



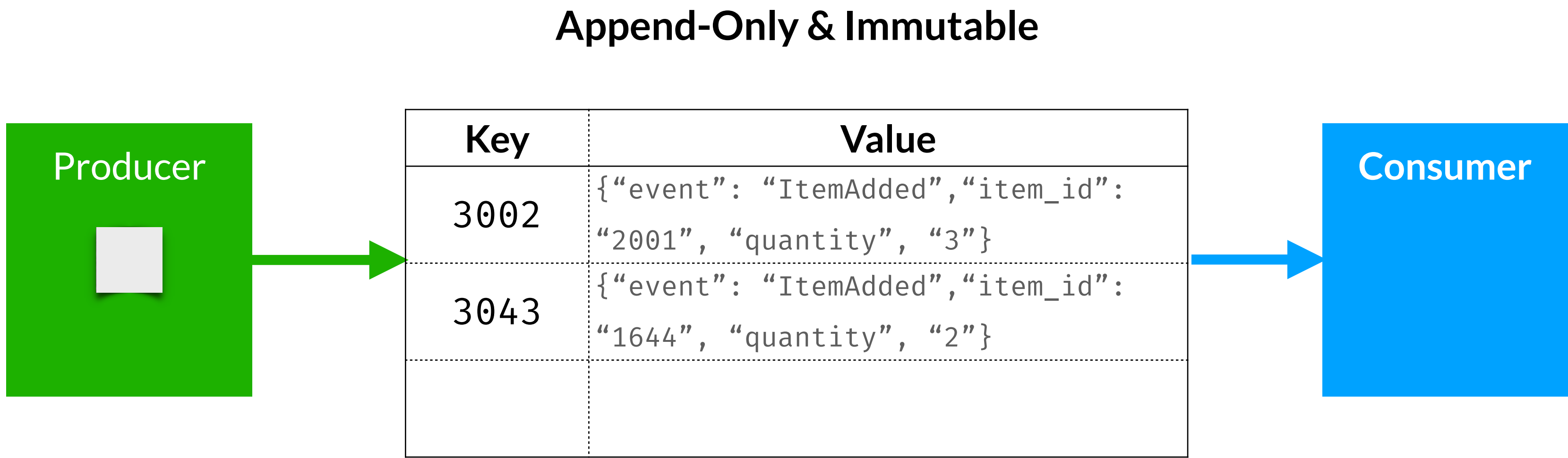
The Diagram



The Diagram



The Diagram



The Diagram

Append-Only & Immutable



The Tradeoffs

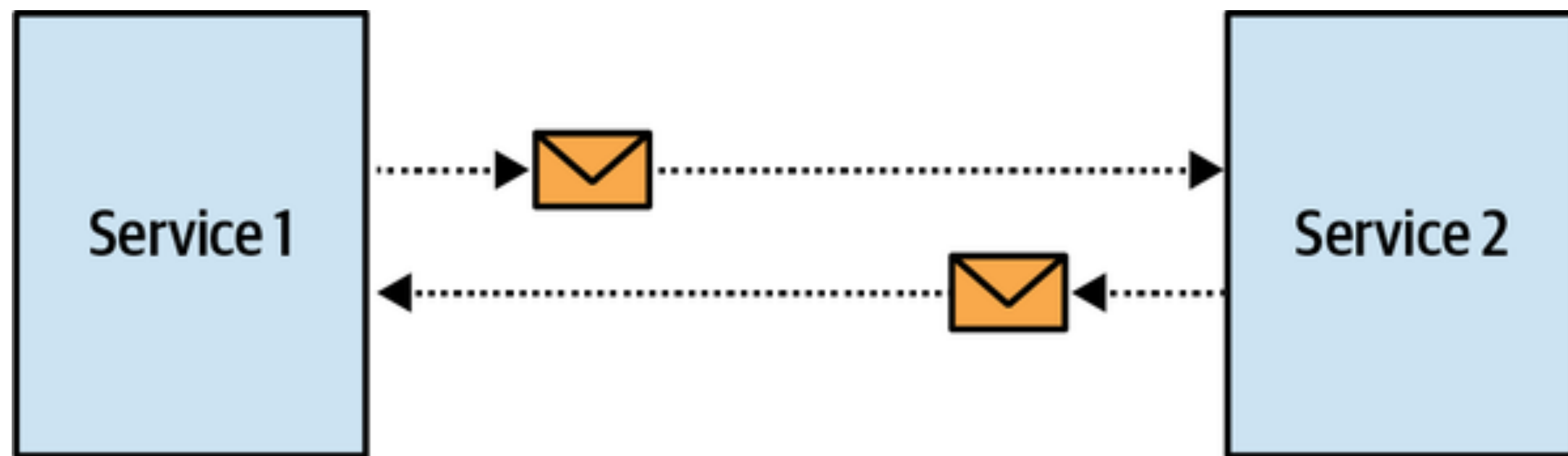
- **Eventually Consistent** - The data that is available to be read will soon be there, but will not be there immediately
- Immediately real-time consistency is not available. It's very close though.
- If you do not require audit trails, history, and roll backs, it might be overkill
- If you do not expect a lot of conflicts, this too will be overkill

Event Driven Architecture



Event Driven Architecture

- Architectural style in which a system's components communicate with one another asynchronously by exchanging event message
- Instead of calling the services' endpoints synchronously, the components publish events to notify other system elements of changes in the system's domain.
- The components can subscribe to events raised in the system and react accordingly



Event Sourcing vs. Event Driven Architecture

- Event Driven Architecture (EDA) refers to the communication between services
- Event sourcing happens inside a service
- The events designed for event sourcing represent state transitions (of aggregates in an event-sourced domain model) implemented in the service and capturing intricacies of the business domain

Two Types of Messages

- **Event**
 - A message describing a change that has already happened
- **Command**
 - A message describing an operation that has to be carried out

Rules for the Messages

- Both events and commands can be communicated asynchronously as messages
- A command can be rejected, and refuse to execute the command
- A recipient of an event, on the other hand, cannot cancel the event

Events are Past Tense

- Since an event describes something that has already happened, an event's name should be formulated in the past tense
 - DeliveryScheduled
 - ShipmentCompleted
 - DeliveryConfirmed

Structure of an Event

- A typical event schema includes the event's metadata and its payload—the information communicated by the event
- An event's payload not only describes the information conveyed by the event, but also defines the event's type

```
{  
  "type": "delivery-confirmed",  
  "event-id": "14101928-4d79-4da6-9486-dbc4837bc612",  
  "correlation-id": "08011958-6066-4815-8dbe-dee6d9e5ebac",  
  "delivery-id": "05011927-a328-4860-a106-737b2929db4e",  
  "timestamp": 1615718833,  
  "payload": {  
    "confirmed-by": "17bc9223-bdd6-4382-954d-f1410fd286bd",  
    "delivery-time": 1615701406  
  }  
}
```


Claim Check



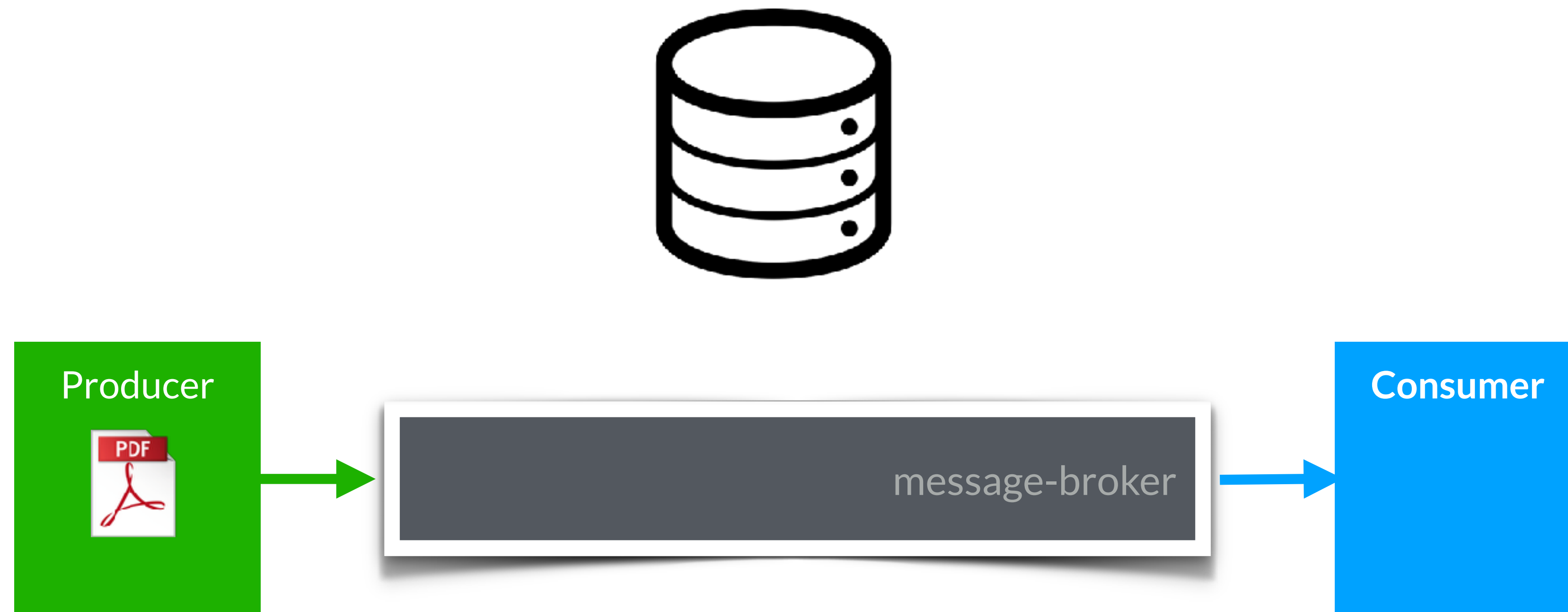
The Context

- Many messaging applications can't accept really large payloads
- Send a claim check to the messaging platform and store that payload to an external service.
- This will prevent the messaging service from being overwhelmed or being slowed down
- Less expensive overall, since storage is cheaper than messaging

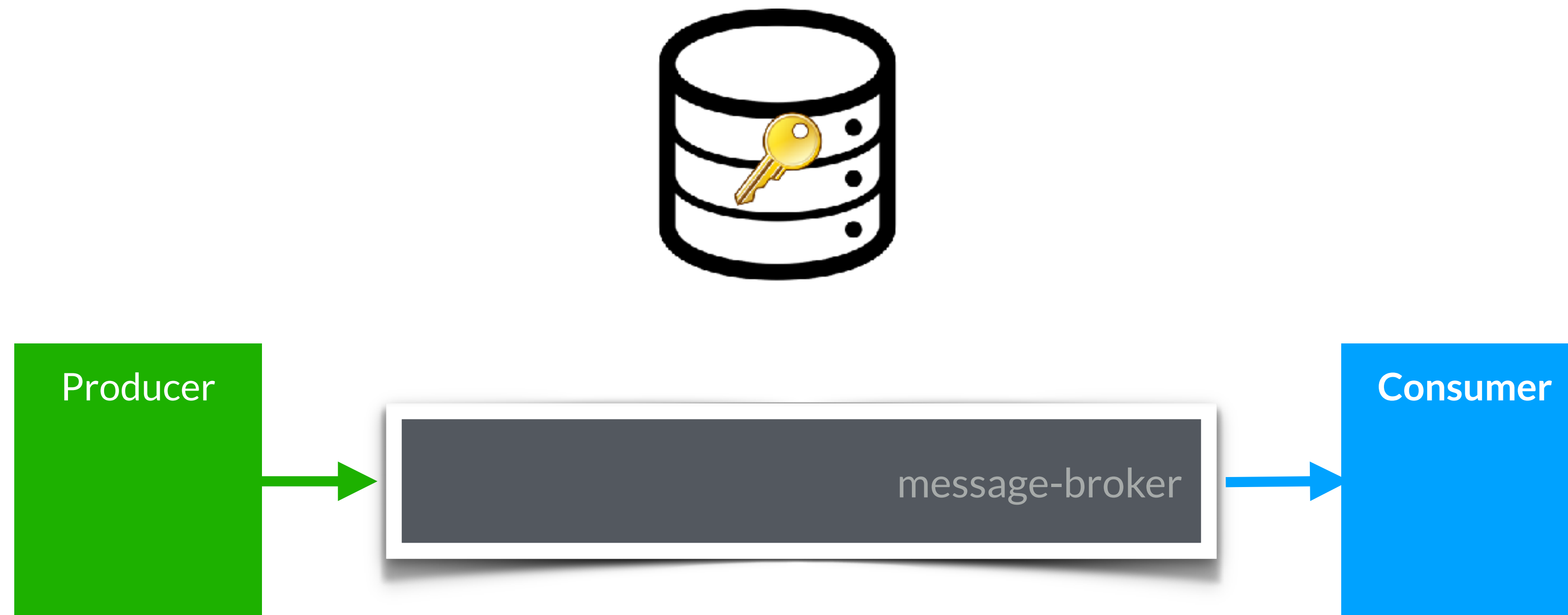
The Solution

- Store the entire message payload into an external service, like a datastore
- Get a reference, like a integer identifier to stored payload
- Attach and send the reference to the message broker *inside of the message*
- The consumer of the message will use the identifier to retrieve the large payload

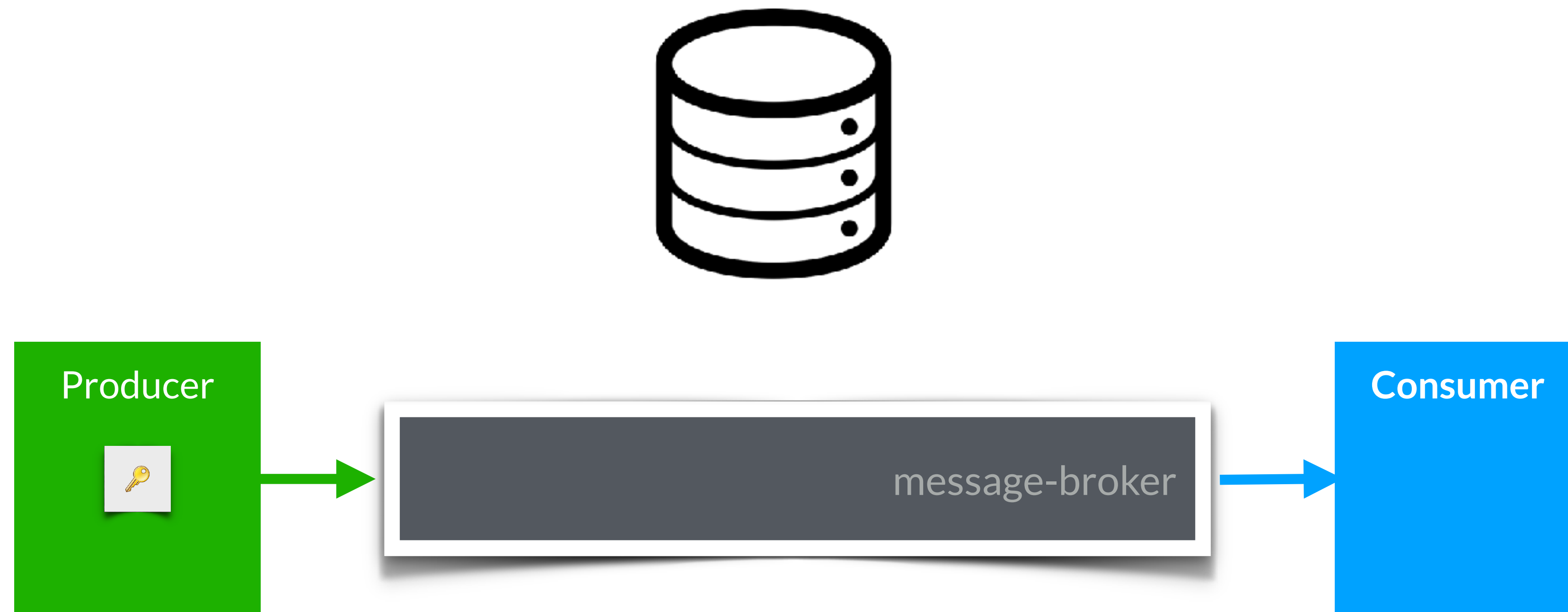
The Diagram



The Diagram



The Diagram



The Diagram



Event Notification



Event Notification

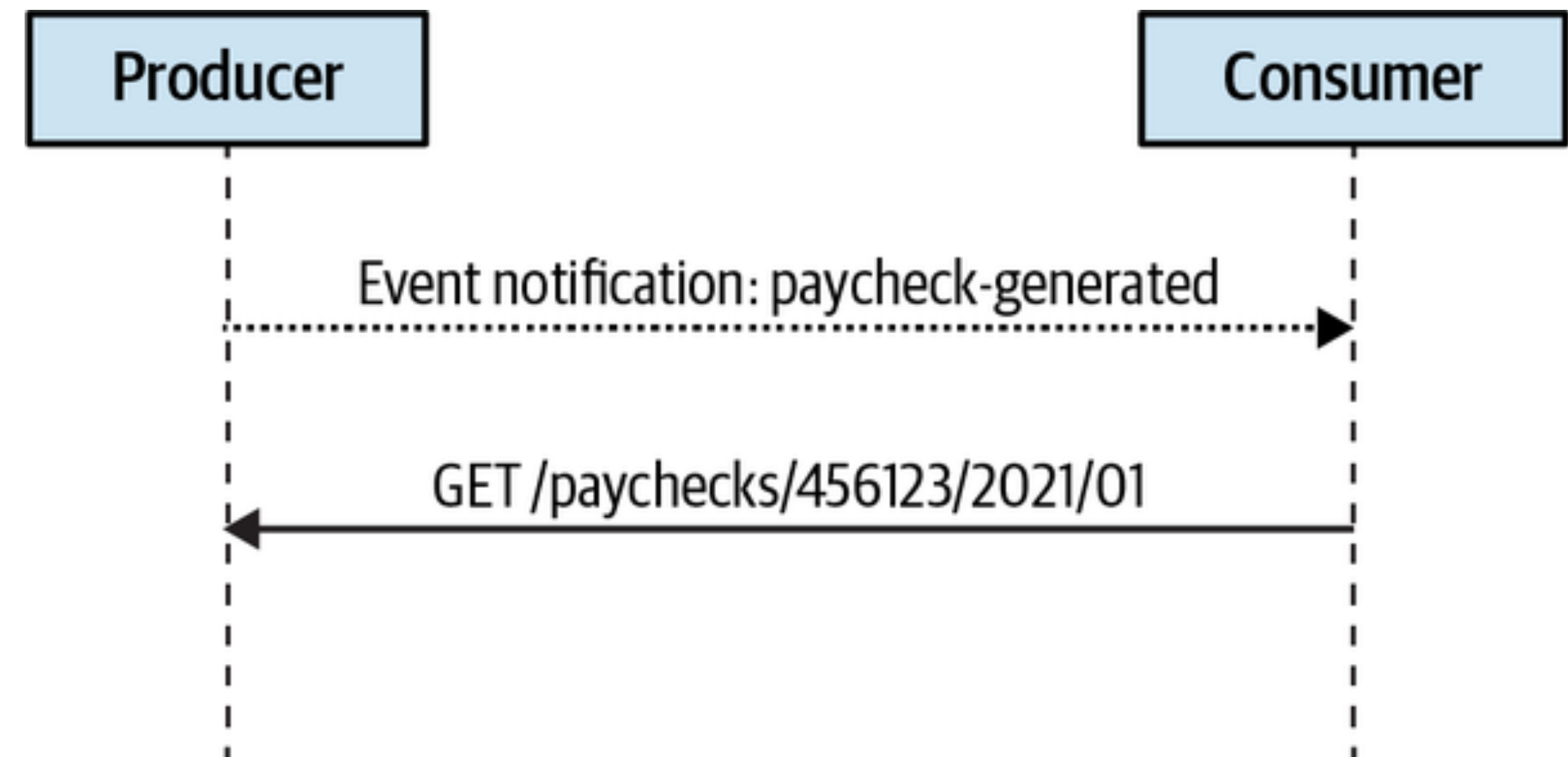
- Message regarding a change in the business domain that other components will react to
- Examples include:
 - PaycheckGenerated
 - CampaignPublished
- Event notification should not be verbose
- The goal is to notify the interested parties about the event and shouldn't carry all the information needed for the subscribers to react to the event

Event Notification

```
{  
  "type": "paycheck-generated",  
  "event-id": "537ec7c2-d1a1-2005-8654-96aee1116b72",  
  "delivery-id": "05011927-a328-4860-a106-737b2929db4e",  
  "timestamp": 1615726445,  
  "payload": {  
    "employee-id": "456123",  
    "link": "/paychecks/456123/2021/01"  
  }  
}
```

Event Notification

- Event notifies the external components of a paycheck that was generated.
- It doesn't carry all the information related to the paycheck.
- Instead, the receiver can use the link to fetch more detailed information



Benefits to Event Notification

- Security
 - Enforcing the recipient to explicitly query for the detailed information prevents sharing sensitive information over the messaging infrastructure and requires additional authorization of the subscribers to access the data.
- Concurrency
 - If information is rendered with the message, the message content will become stale, querying it will retain up-to-date state. This can also include pessimistic locking where no other consumer can process the same message.

Event Carried State Transfer



Event Carried State Transfer

- Event-carried state transfer (ECST) messages notify subscribers about changes in the producer's internal state
- ECST messages on the other hand include all the data reflecting the change in the state.

Event Carried State Transfer

One form is carrying the complete data

```
{
  "type": "customer-updated",
  "event-id": "6b7ce6c6-8587-4e4f-924a-cec028000ce6",
  "customer-id": "01b18d56-b79a-4873-ac99-3d9f767dbe61",
  "timestamp": 1615728520,
  "payload": {
    "first-name": "Carolyn",
    "last-name": "Hayes",
    "phone": "555-1022",
    "status": "follow-up-set",
    "follow-up-date": "2021/05/08",
    "birthday": "1982/04/05",
    "version": 7
  }
}
```

Event Carried State Transfer

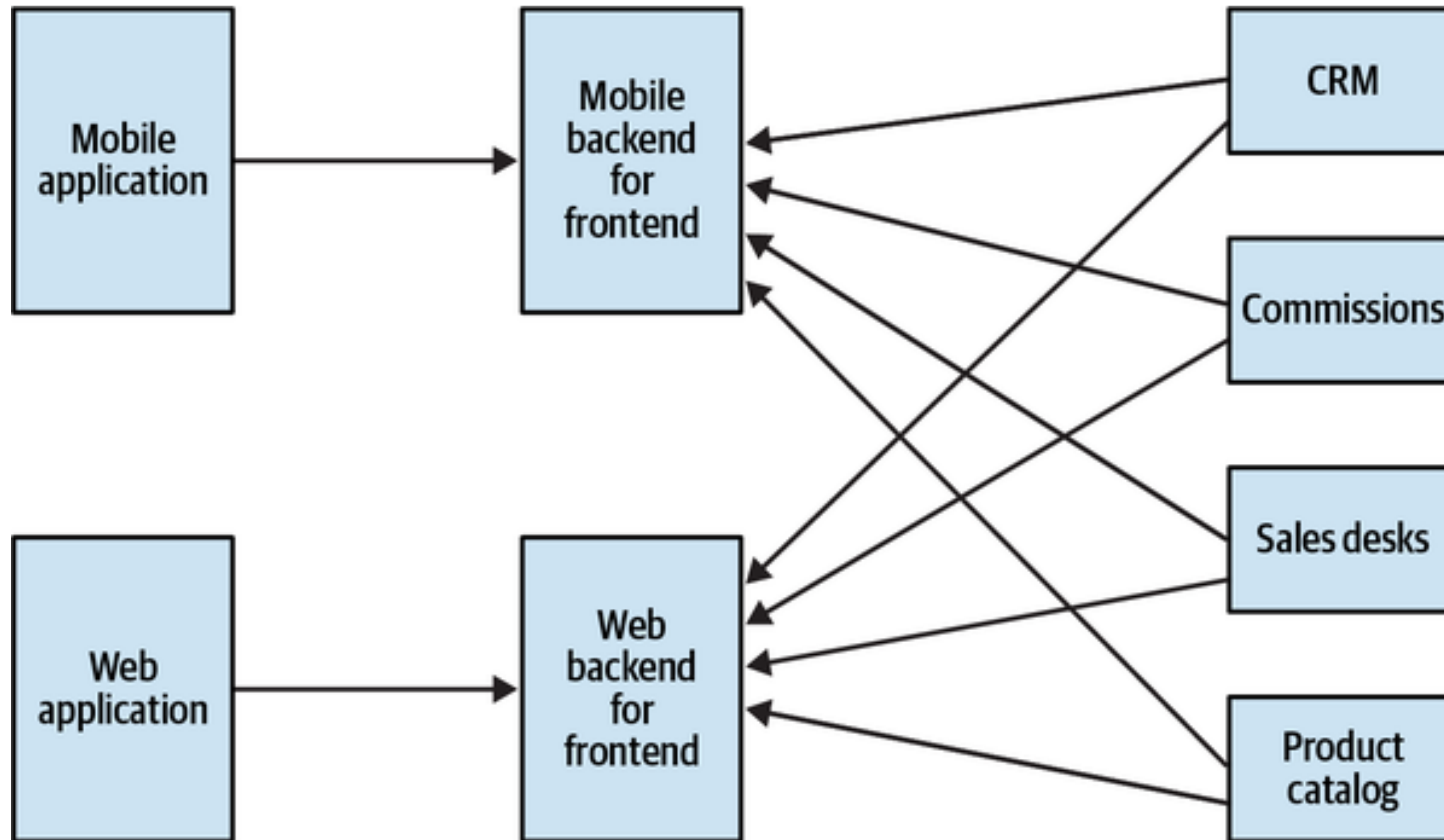
Another form is only carrying the data that has modified

```
{
  "type": "customer-updated",
  "event-id": "6b7ce6c6-8587-4e4f-924a-cec028000ce6",
  "customer-id": "01b18d56-b79a-4873-ac99-3d9f767dbe61",
  "timestamp": 1615728520,
  "payload": {
    "status": "follow-up-set",
    "follow-up-date": "2021/05/10",
    "version": 8
  }
}
```

Caching in Event Driven Architecture

- Whether ECST messages include complete snapshots or only the updated fields, a stream of such events allows consumers to hold a local cache of the entities' states and work with it
- Conceptually, using event-carried state transfer messages is an asynchronous data replication mechanism.
- This approach makes the system more fault tolerant, meaning that the consumers can continue functioning even if the producer is not available

Caching in Event Driven Architecture



Domain Events



Domain Events

- Halfway between event notifications and ECST messages
- Both domain events and event notifications describe changes in the producer's business domain
- Domain events include all the information describing the event
- Domain events are intended to model and describe the business domain
- In event-sourced systems, domain events are used to model all possible state transitions

ECST vs Domain Events

- Event-Carried State Transfer
 - Sufficient Information to hold in a local cache and is often the aggregate
- Domain-Events
 - Not intended to describe any aggregate state, but the event that just happened

Comparing Events

The background of the slide is a vibrant blue sky filled with soft, white clouds. Overlaid on this is a complex network of white dots of varying sizes, connected by thin white lines. This network pattern is most prominent in the lower half of the image, creating a sense of digital connectivity and data flow.

Event Comparison: Event Notification

- Event notification message.
- It contains no information except the fact that the person with the specified ID got married.
- It contains minimal information about the event
- Consumers interested in more details will have to follow the link in the details field.

```
eventNotification = {  
  "type": "marriage-recorded",  
  "person-id": "01b9a761",  
  "payload": {  
    "person-id": "126a7b61",  
    "details": "/01b9a761/marriage-data"  
  }  
};
```


Event Comparison: ECST

- Event Carried State Transfer Message
- Describes the changes in the person's personal details (last name)
- Message doesn't describe the reason for the change

```
ecst = {  
  "type": "personal-details-changed",  
  "person-id": "01b9a761",  
  "payload": {  
    "new-last-name": "Williams"  
  }  
};
```

Event Comparison: Domain Event

- Modeled as close as possible to the nature of the event in the business domain.
- It includes the person's ID and a flag indicating whether the person assumed their partner's name.

```
domainEvent = {  
  "type": "married",  
  "person-id": "01b9a761",  
  "payload": {  
    "person-id": "126a7b61",  
    "assumed-partner-last-name": true  
  }  
};
```

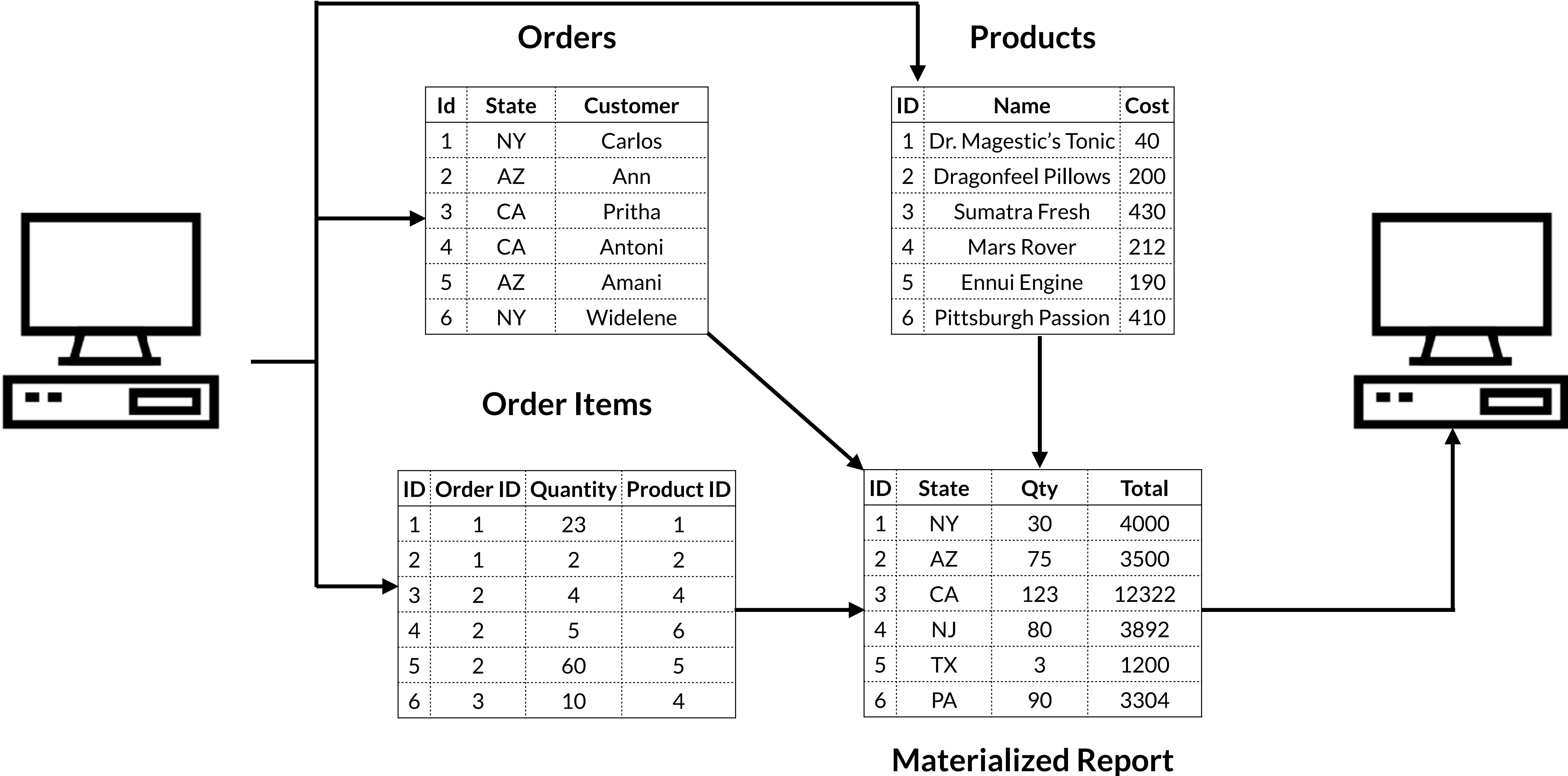

Materialized Views

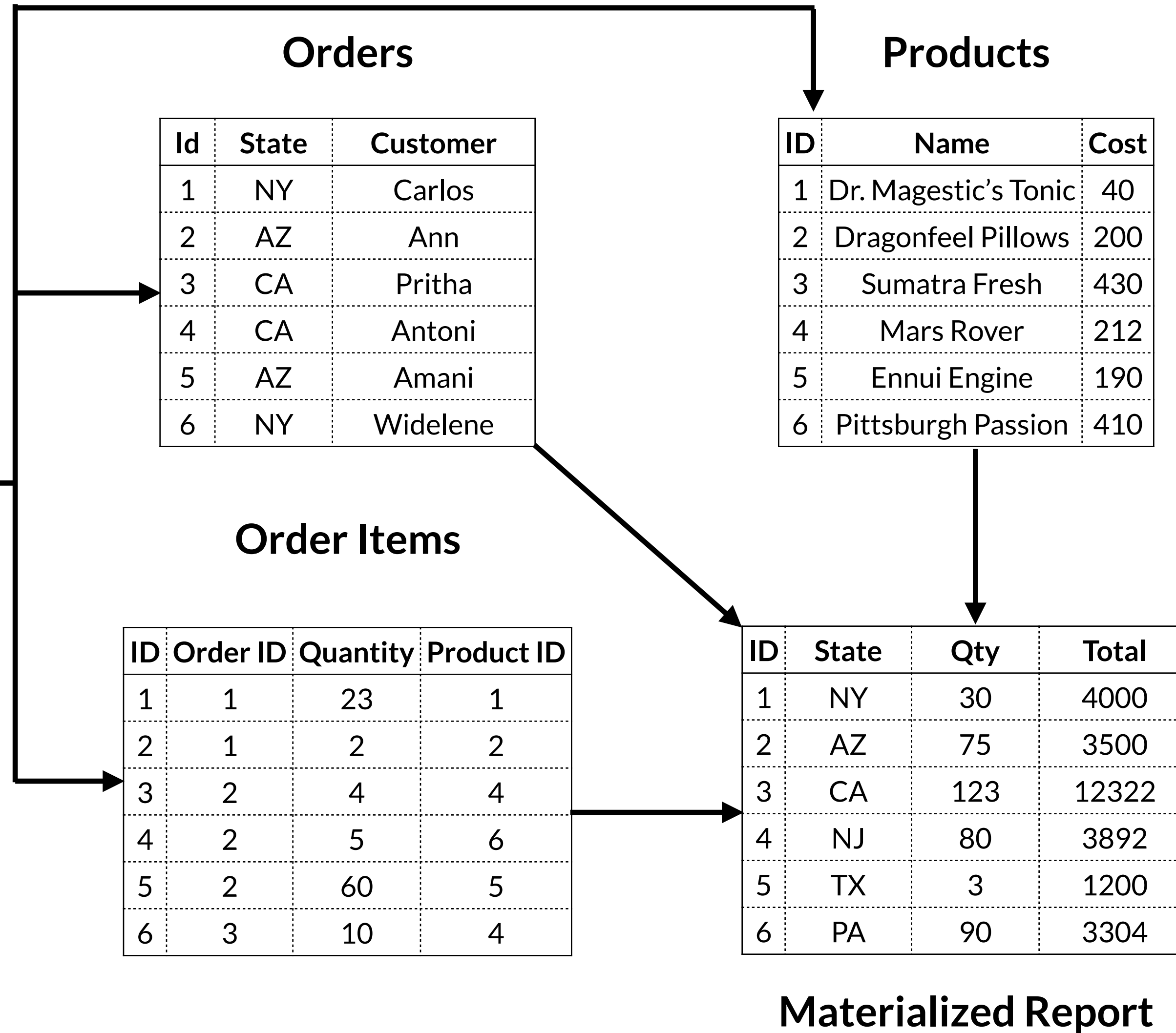


The Problem

- Currently much of the data revolves on how it is stored, not how it is read
- Data when read needs to be transformed and prepared
- Each entity typically has too much information for it to be queryable and usable
- For example, one data entry in a document style database can have other aggregates that are not absolutely necessary
- Data may also need to be joined, cleansed, or engineered for a particular purpose, like machine learning

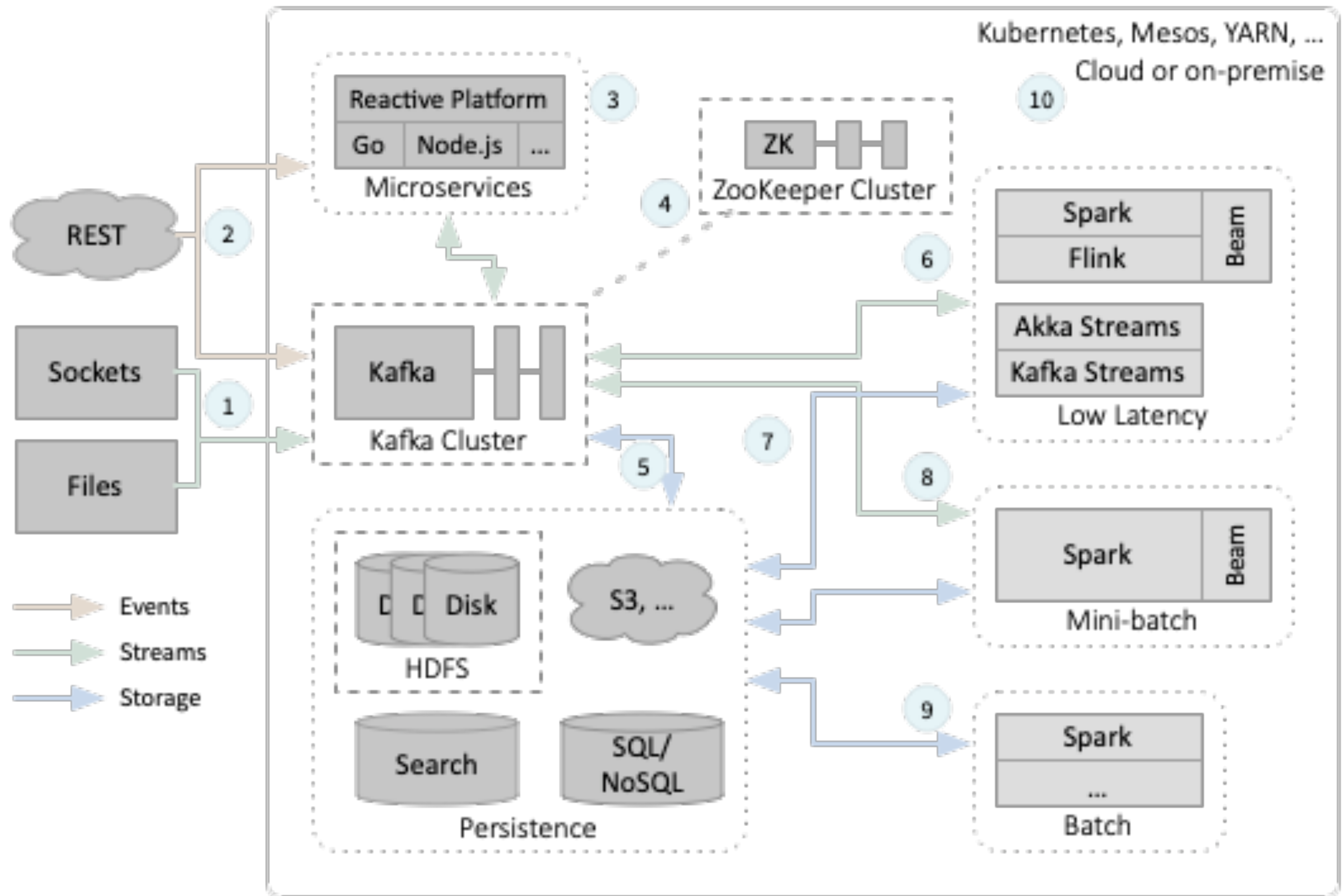
The Diagram





The Solution

- Create a different perspective of the data that you need
- This can be implemented by the datastore itself
 - Oracle
 - PostgreSQL
- Can be performed by Stream Processing Frameworks
 - Kafka
 - Spark
 - Flink



The Tradeoffs

- If your source is already simple and easy to query, there is no need
- If immediate consistency is required, then query direct

CQRS

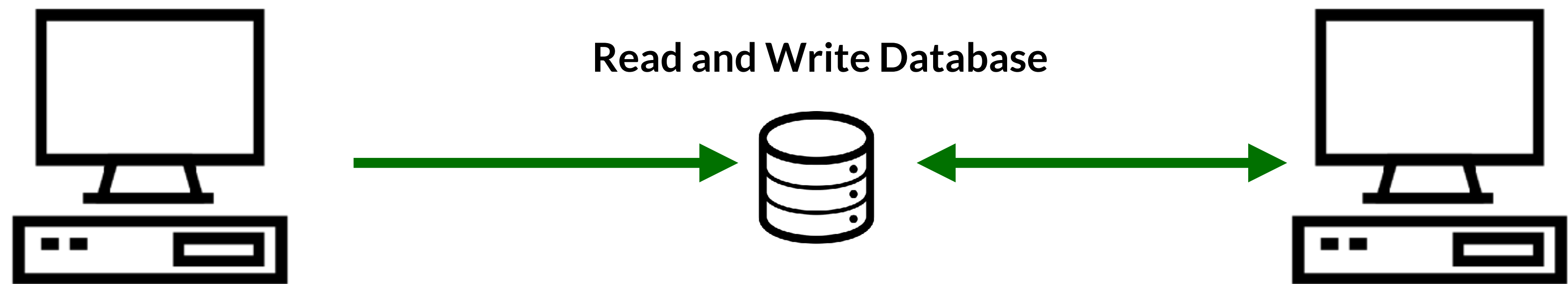


CQRS

- **Command and Query Responsibility Segregation**
- Separating reads, add-updates, for a databases
- Benefits include better performance, scalability, and security
- Better evolution over time
- Prevents Merge Conflicts:
 - Database locking ensures that the updates don't change the same data concurrently, but it doesn't ensure that multiple independent changes result in a consistent data model.

The Diagram

What if we need to query...
SELECT state, count(*)
from orders
group by state,over and
over?



State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani

The Diagram



Write Database



State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani

Read Database



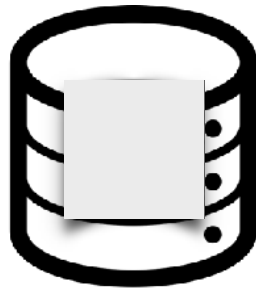
State	Total
NY	4000
AZ	2433
CA	1340



The Diagram



Write Database



State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani
NY	900	Widelene

Read Database



State	Total
NY	4000
AZ	2433
CA	1340



The Diagram



Write Database



State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani
NY	900	Widelene

Read Database



State	Total
NY	4900
AZ	2433
CA	1340



Demo: CQRS



- Next we will use Kafka to implement the following behaviors
 - Event Sourcing
 - Materialized Views
 - CQRS (Command Query Responsibility Segregation)

Message Coupling

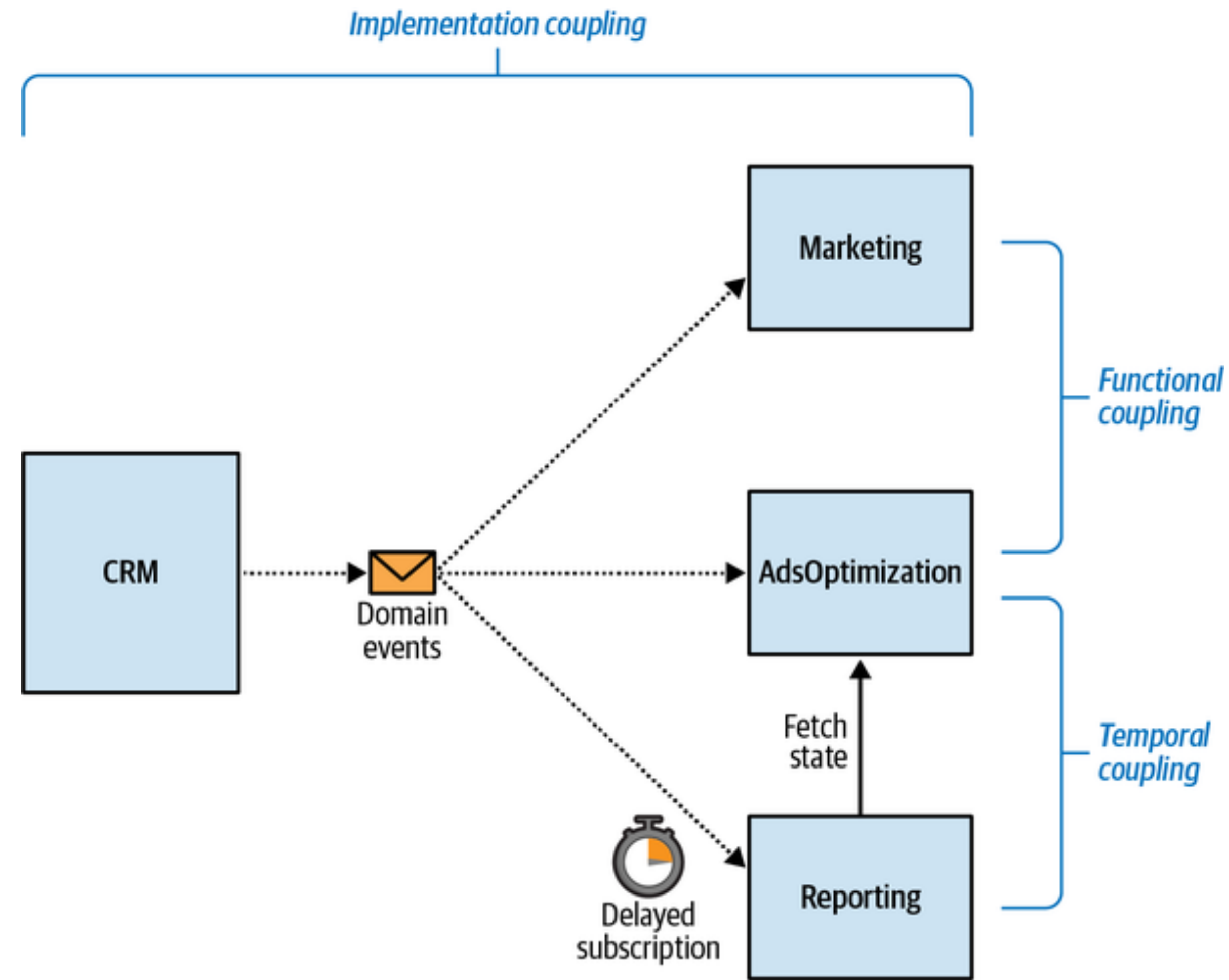


Design Event Driven Integration

- The events in an EDA-based system are first-class design elements
- They affect both how the components are integrated and the components' boundaries themselves.
- Choosing the correct type of event message is what makes (decouples) or breaks (couples) a distributed system

Big Ball of Mud

Problems with this implementation



Design Event Driven Integration

- Marketing bounded context ingests a domain event which is heavy, and has to create a projection or enrichment
- AdsOptimization bounded context ingests a domain event, and say, creates the same projection
- Reporting bounded context also ingests the domain event, but needs to wait for AdOptimization to complete after a certain time, before processing it's payload

Three Coupling Issues

Three Coupling Issues with this architecture

- Temporal Coupling
- Functional Coupling
- Implementation Coupling

Temporal Coupling

- AdsOptimization and Reporting bounded contexts are temporally coupled
- They depend on strict order of execution
- AdsOptimization has to finish before the Reporting module is triggered
- If reporting triggers first then there is inconsistent data, particularly if AdsOptimization is backed up
- The delay may not work all the time, given a 5-minute delay:
 - AdsOptimization may be overloaded and unable to finish
 - A network issue may delay the delivery of incoming messages to the AdsOptimization service
 - AdsOptimization component can experience an outage and stop processing incoming messages.

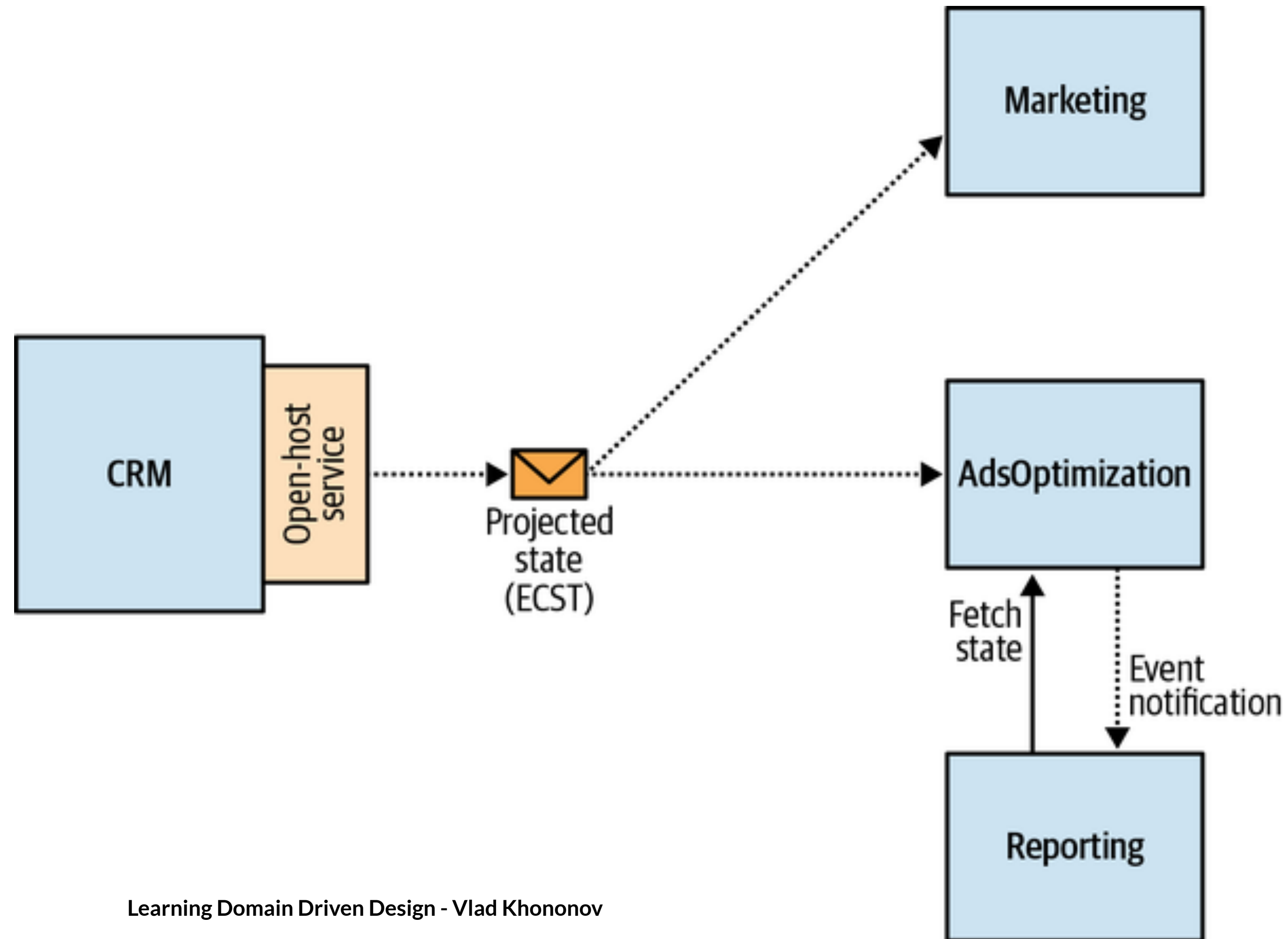
Functional Coupling

- Marketing and AdsOptimization bounded contexts both subscribed to the CRM's domain events and ended up implementing the same projection or enrichment of the customers' data! This is a duplication!
- **If the projection was changed in one of the components, the change had to be replicated in the second bounded context!**

Implementation Coupling

- The Marketing and AdsOptimization bounded contexts are subscribed to all the domain events generated by the CRM's event-sourced model
- Consequently, a change in the CRM's implementation, such as:
 - Adding a new domain event
 - Changing the schema of an existing one
 - Changes has to be reflected in both, Marketing and AdOptimization bounded context
- Ignoring changes can lead to inconsistent data and state

Refactored Event Driven Integration



Refactoring Solution

- Encapsulated the projection/enrichment in the producer, the CRM bounded context
- Message type changed from a domain event to a event carried state transfer message
- Reporting Bounded Context now receives an event notification where it can fetch the state

Modeling Time



Analyzing Tabular Data

- One can analyze data and get a feel for any process by looking at a table
- Table representations only capture one moment in time, not the data that lead up to it
- For example, your checking account. You know your balance, but how did it get there?

View the Data

lead-id	first-name	last-name	status	phone-number	followup-on	created-on	updated-on
1	Sean	Callahan	CONVERTED	555-1246		2019-01-31T10:02:40.32Z	2019-01-31T10:02:40.32Z
2	Sarah	Estrada	NEW_LEAD	555-4395		2019-03-29T22:01:41.44Z	2019-03-29T22:01:41.44Z
3	Stephanie	Brown	CLOSED	555-1176		2019-04-15T23:08:45.59Z	2019-04-15T23:08:45.59Z
4	Sami	Calhoun	PENDING_PAYM ENT	555-1850		2019-04-25T05:42:17.07Z	2019-04-25T05:42:17.07Z
5	Sian	Espinoza	FOLLOWUP_SET	555-6461	2019-12-04T01:49:08.05Z	2019-12-04T01:49:08.05Z	2019-12-04T01:49:08.05Z

What we understood

- The sales flow starts with the potential customer in the NEW_LEAD status.
- A sales call can end with the person not being interested in the offer (the lead is CLOSED), scheduling a follow-up call (FOLLOWUP_SET), or accepting the offer (PENDING_PAYMENT).
- If the payment is successful, the lead is CONVERTED into a customer. Conversely, the payment can fail—PAYMENT_FAILED

Missing Information

Since we don't have the information that lead to this point, we might be missing valuable information!

- Was there a purchase made right away?
- Was there a lengthy sales journey?
- Is it worth trying to contact a person after multiple follow-ups, or is it more efficient to close the lead and move to a more promising prospect?
- None of that information is there. All we know are the leads' current states.

Event Sourcing

- The event sourcing pattern introduces the dimension of time into the data model.
- Instead of the schema reflecting the aggregates' current state, an event sourcing-based system persists events documenting every change in an aggregate's lifecycle

Rebuilding The Data

```
{  
  "lead-id": 12,  
  "event-id": 0,  
  "event-type": "lead-initialized",  
  "first-name": "Casey",  
  "last-name": "David",  
  "phone-number": "555-2951",  
  "timestamp": "2020-05-20T09:52:55.95Z"  
}
```


Rebuilding The Data

```
{  
  "lead-id": 12,  
  "event-id": 1,  
  "event-type": "contacted",  
  "timestamp": "2020-05-20T12:32:08.24Z"  
}
```

Rebuilding The Data

```
{  
  "lead-id": 12,  
  "event-id": 2,  
  "event-type": "followup-set",  
  "followup-on": "2020-05-27T12:00:00.00Z",  
  "timestamp": "2020-05-20T12:32:08.24Z"  
}
```

Rebuilding The Data

```
{  
  "lead-id": 12,  
  "event-id": 3,  
  "event-type": "contact-details-updated",  
  "first-name": "Casey",  
  "last-name": "Davis",  
  "phone-number": "555-8101",  
  "timestamp": "2020-05-20T12:32:08.24Z"  
}
```


Rebuilding The Data

```
{  
  "lead-id": 12,  
  "event-id": 4,  
  "event-type": "contacted",  
  "timestamp": "2020-05-27T12:02:12.51Z"  
}
```

Rebuilding The Data

```
{  
  "lead-id": 12,  
  "event-id": 5,  
  "event-type": "order-submitted",  
  "payment-deadline": "2020-05-30T12:02:12.51Z",  
  "timestamp": "2020-05-27T12:02:12.51Z"  
}
```

Rebuilding The Data

```
{  
  "lead-id": 12,  
  "event-id": 6,  
  "event-type": "payment-confirmed",  
  "status": "converted",  
  "timestamp": "2020-05-27T12:38:44.12Z"  
}
```


OLTP vs. OLAP



OLTP vs. OLAP

- Online analytical processing (OLAP) and online transaction processing (OLTP) are data processing systems that help you store and analyze business data.
- OLAP combines and groups the data so you can analyze it from different points of view
- OLTP stores and updates transactional data reliably and efficiently in high volumes
- OLTP databases can be one among several data sources for an OLAP system.

OLTP vs OLAP

Criteria	OLAP	OLTP
Purpose	OLAP helps you analyze large volumes of data to support decision-making.	OLTP helps you manage and process real-time transactions.
Data source	OLAP uses historical and aggregated data from multiple sources.	OLTP uses real-time and transactional data from a single source.
Data structure	OLAP uses multidimensional (cubes) or relational databases.	OLTP uses relational databases.

OLTP vs OLAP

Criteria	OLAP	OLTP
Data model	OLAP uses star schema, snowflake schema, or other analytical models.	OLTP uses normalized or denormalized models.
Volume of data	OLAP has large storage requirements. Think terabytes (TB) and petabytes (PB).	OLTP has comparatively smaller storage requirements. Think gigabytes (GB).

OLTP vs OLAP

Criteria	OLAP	OLTP
Response Time	OLAP has longer response times, typically in seconds or minutes.	OLTP has shorter response times, typically in milliseconds
Example Applications	OLAP is good for analyzing trends, predicting customer behavior, and identifying profitability.	OLTP is good for processing payments, customer data management, and order processing.

Outbox Pattern



Where will this fail?

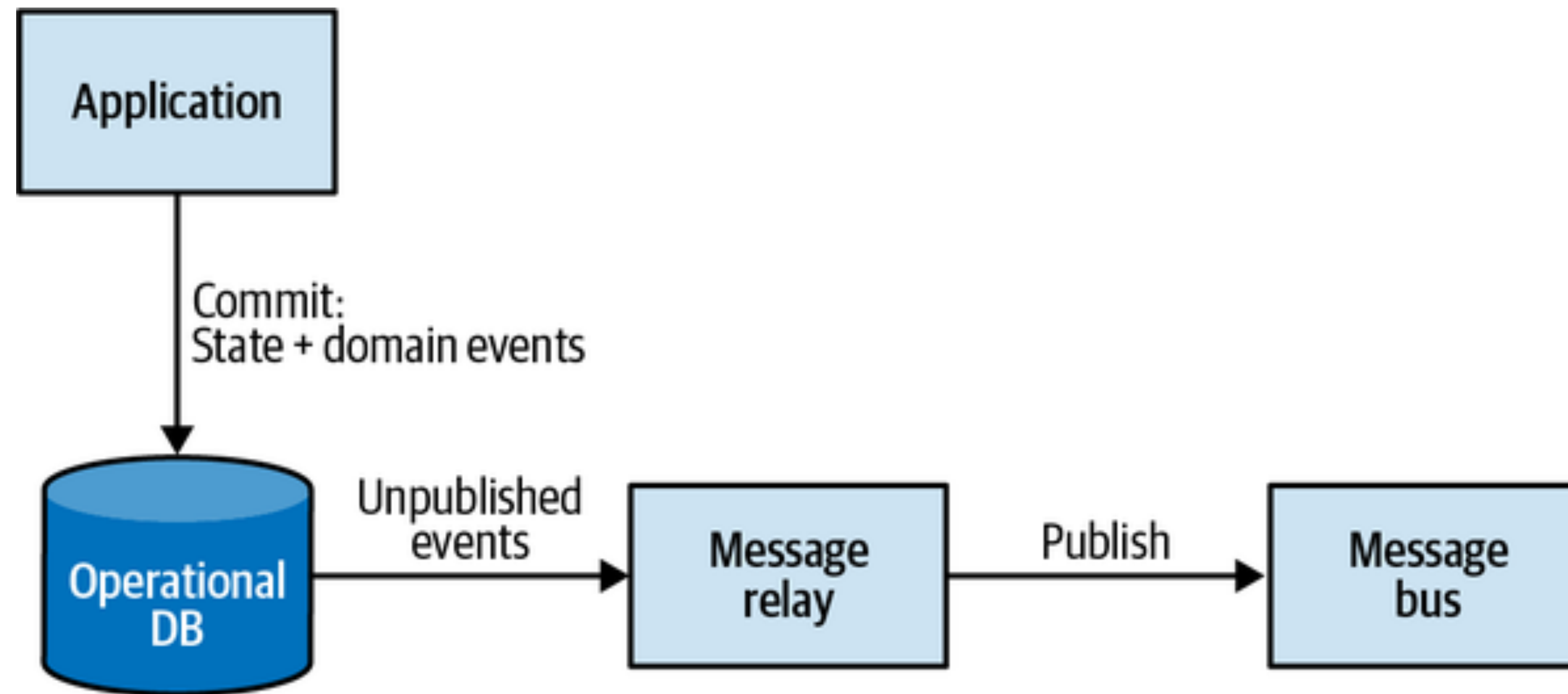
```
public class ManagementAPI {  
    private final MessageBus messageBus;  
    private final CampaignRepository repository;  
  
    public Result deactivateCampaign(CampaignId id, string reason) {  
        //Communicate with the message bus  
        //AND the repository!  
    }  
}
```

- Process running the logic for some reason fails to publish the domain events
- Perhaps the message bus is down
- Server running the code fails right after committing the database transaction, but before publishing the events the system will still end in an inconsistent state

Outbox Pattern

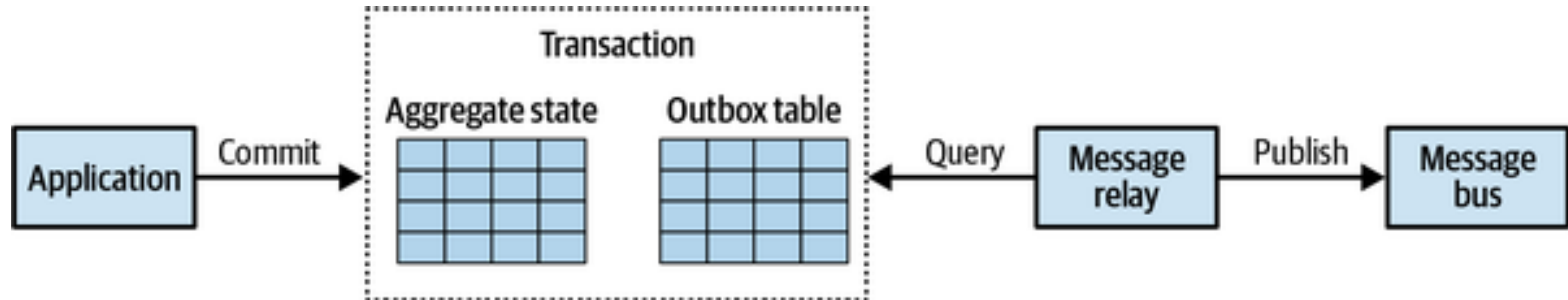
- Both the updated aggregate's state and the new domain events are committed in the same atomic transaction.
- A message relay fetches newly committed domain events from the database.
- The relay publishes the domain events to the message bus.
- Upon successful publishing, the relay either marks the events as published in the database or deletes them completely

Outbox Pattern



Outbox with Relational Databases

When using a relational database, it's convenient to leverage the database's ability to commit to two tables atomically and use a dedicated table for storing the messages



Saga Pattern

The background of the slide features a blue sky with soft, white clouds. Overlaid on this is a complex network of white dots of varying sizes, connected by thin white lines, creating a web-like or molecular structure that spans the entire frame.

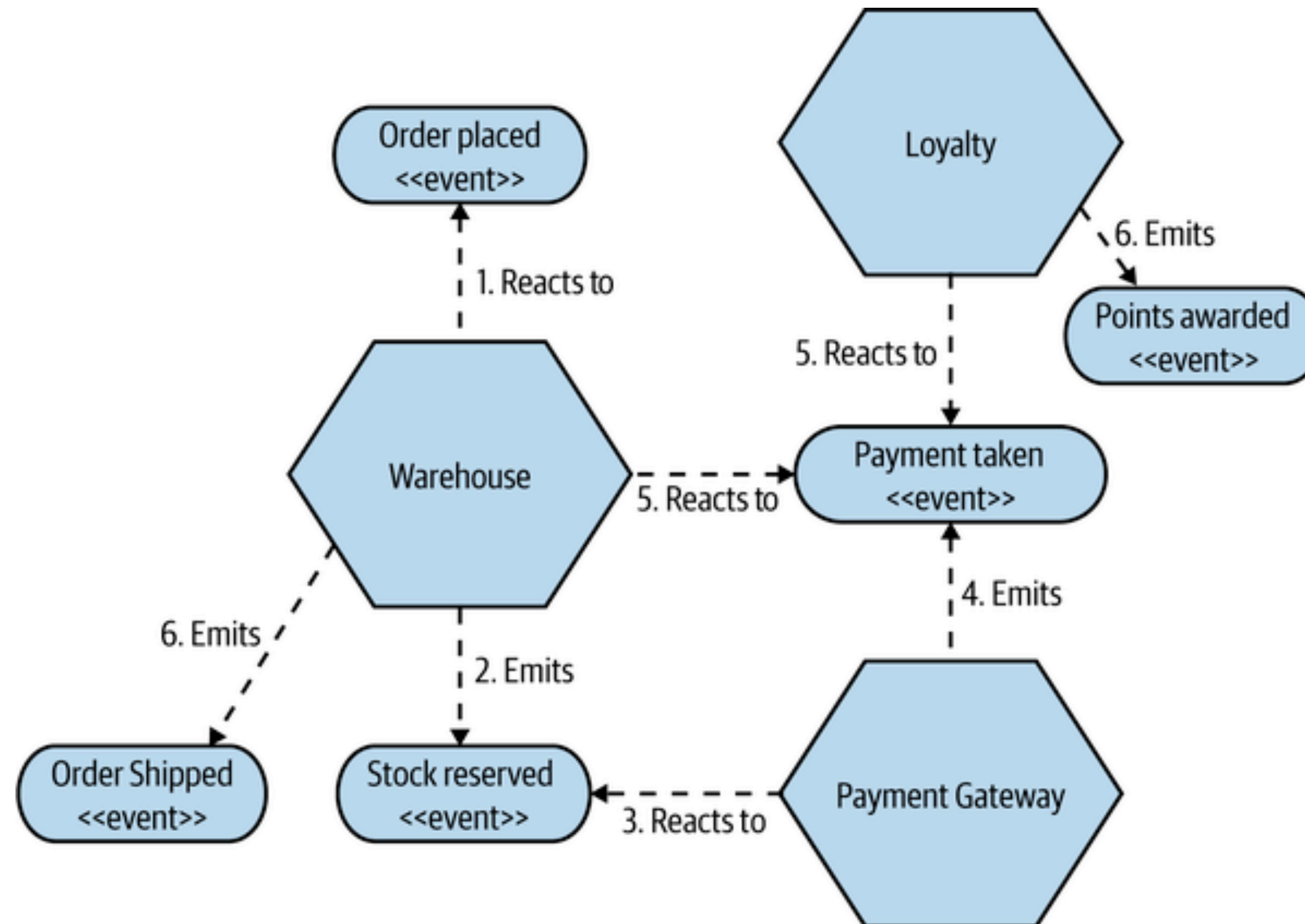
The Context

- We require distributed transactions across through events
- A transaction is a single unit of logic or work, sometimes made up of multiple operations.
- Within a transaction, an event is a state change that occurs to an entity, and a command encapsulates all information needed to perform an action or trigger a later event.
- Transactions must be atomic, consistent, isolated, and durable (ACID). Transactions within a single service are ACID, but cross-service data consistency requires a cross-service transaction management strategy.

ACID Transactions

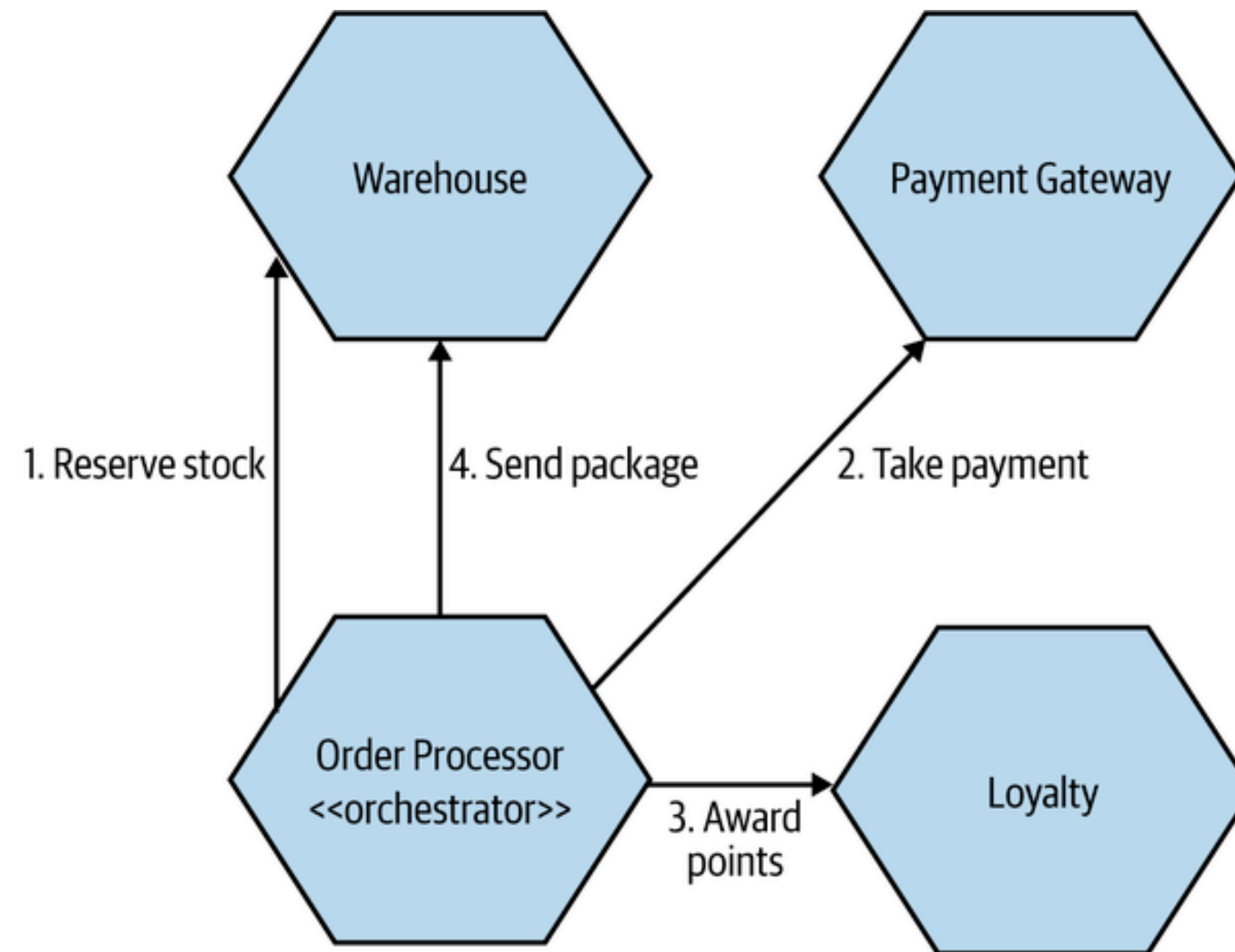
- **Atomicity** is an indivisible and irreducible set of operations that must all occur or none occur.
- **Consistency** means the transaction brings the data only from one valid state to another valid state.
- **Isolation** guarantees that concurrent transactions produce the same data state that sequentially executed transactions would have produced.
- **Durability** ensures that committed transactions remain committed even in case of system failure or power outage.

The Diagram



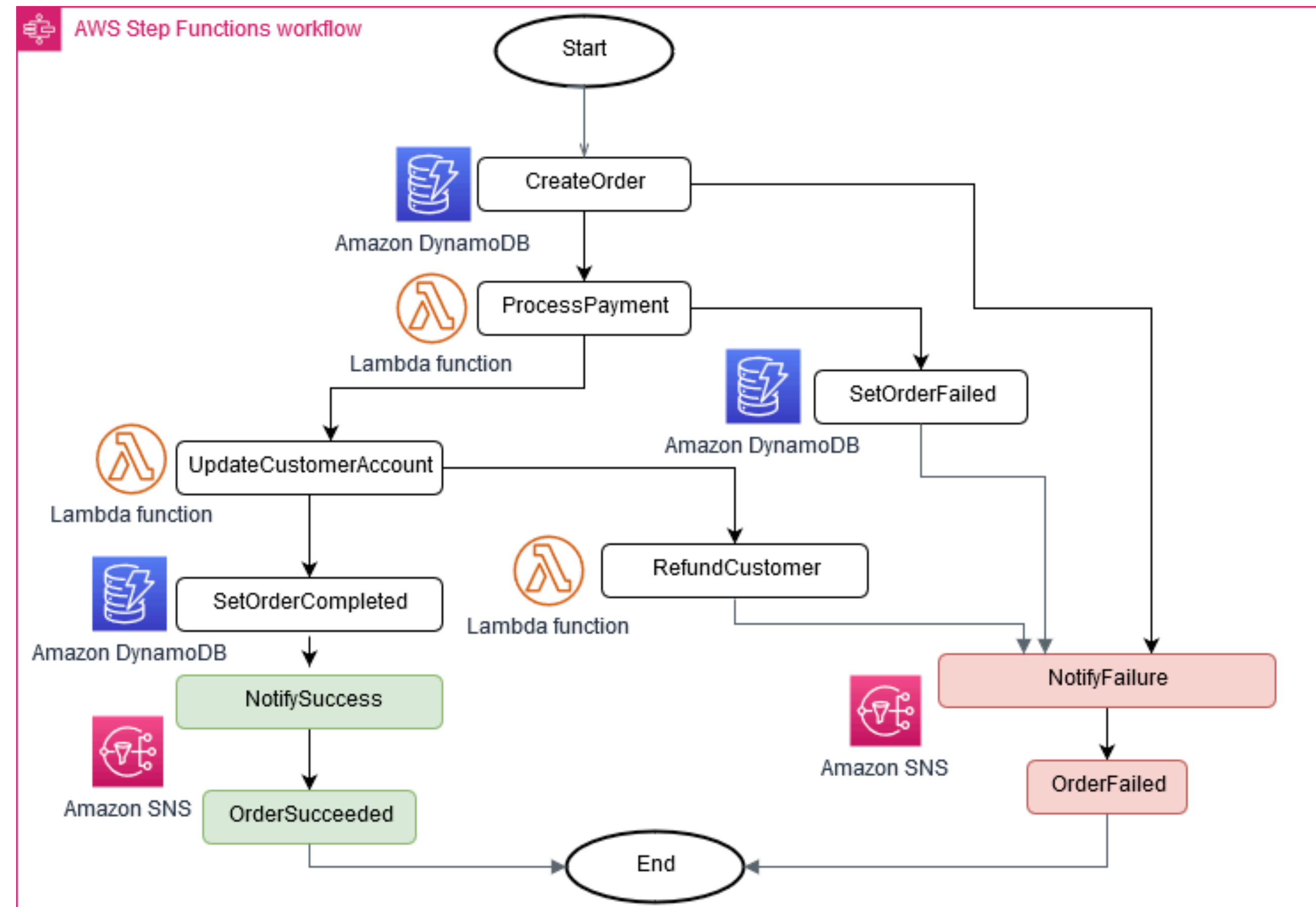
Choreography based pattern

The Diagram



Orchestrator based pattern where state of the transaction is owned by a service

The Diagram



<https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html>

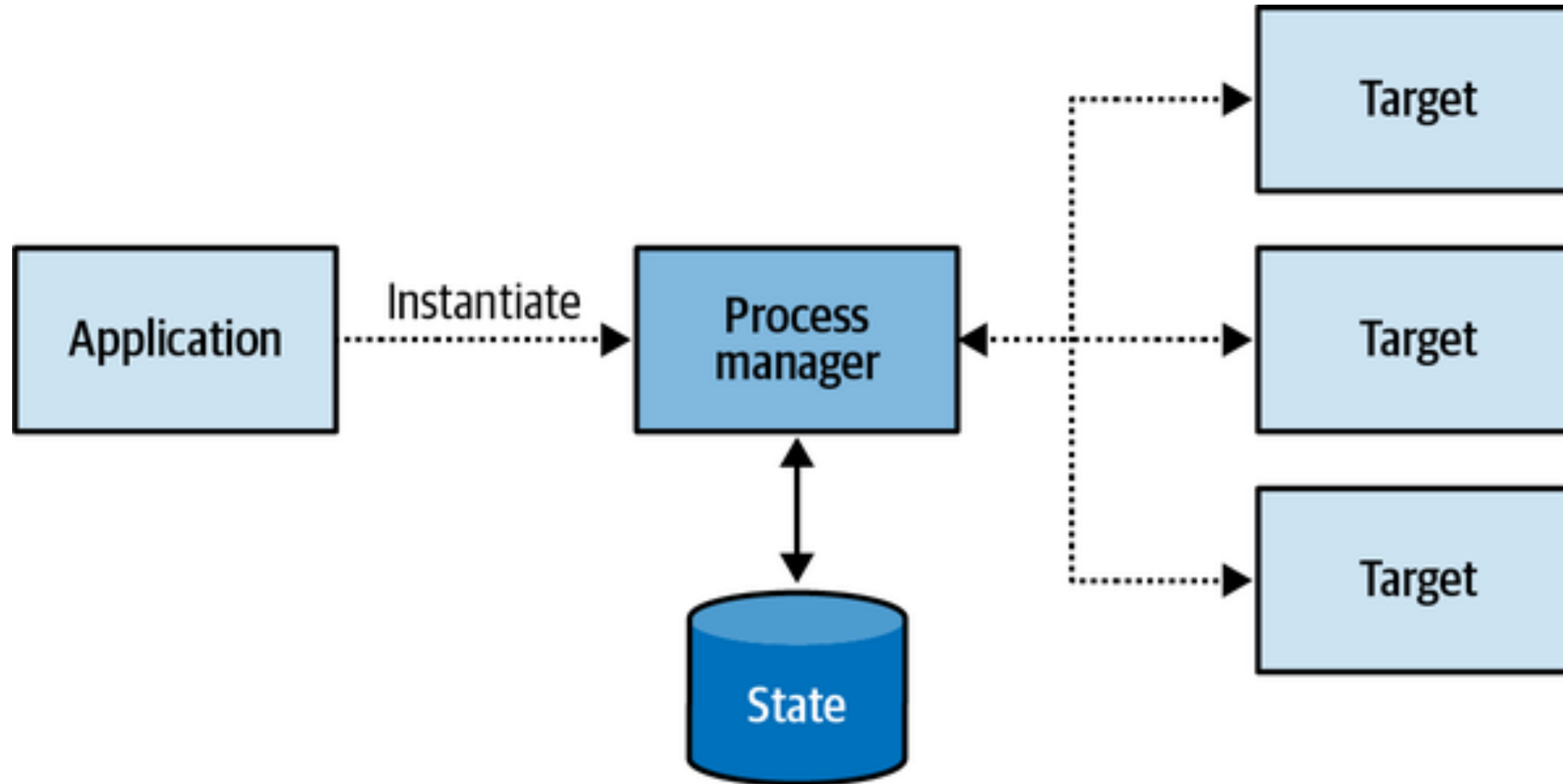
Process Manager

The background of the slide is a vibrant blue sky filled with soft, white clouds. Overlaid on this is a complex network of white dots of varying sizes, connected by thin white lines. This network pattern is most prominent in the lower half of the image, creating a sense of digital connectivity and structure.

Process Manager

- The process manager pattern is intended to implement a business-logic-based process
- It is defined as a central processing unit that maintains the state of the sequence and determines the next processing step
- If a saga contains if-else statements to choose the correct course of action, it is probably a process manager
- Process Manager can be called a Saga Orchestrator where the Orchestrator is the Process Manager

Process Manager



Data Mesh



The Problem

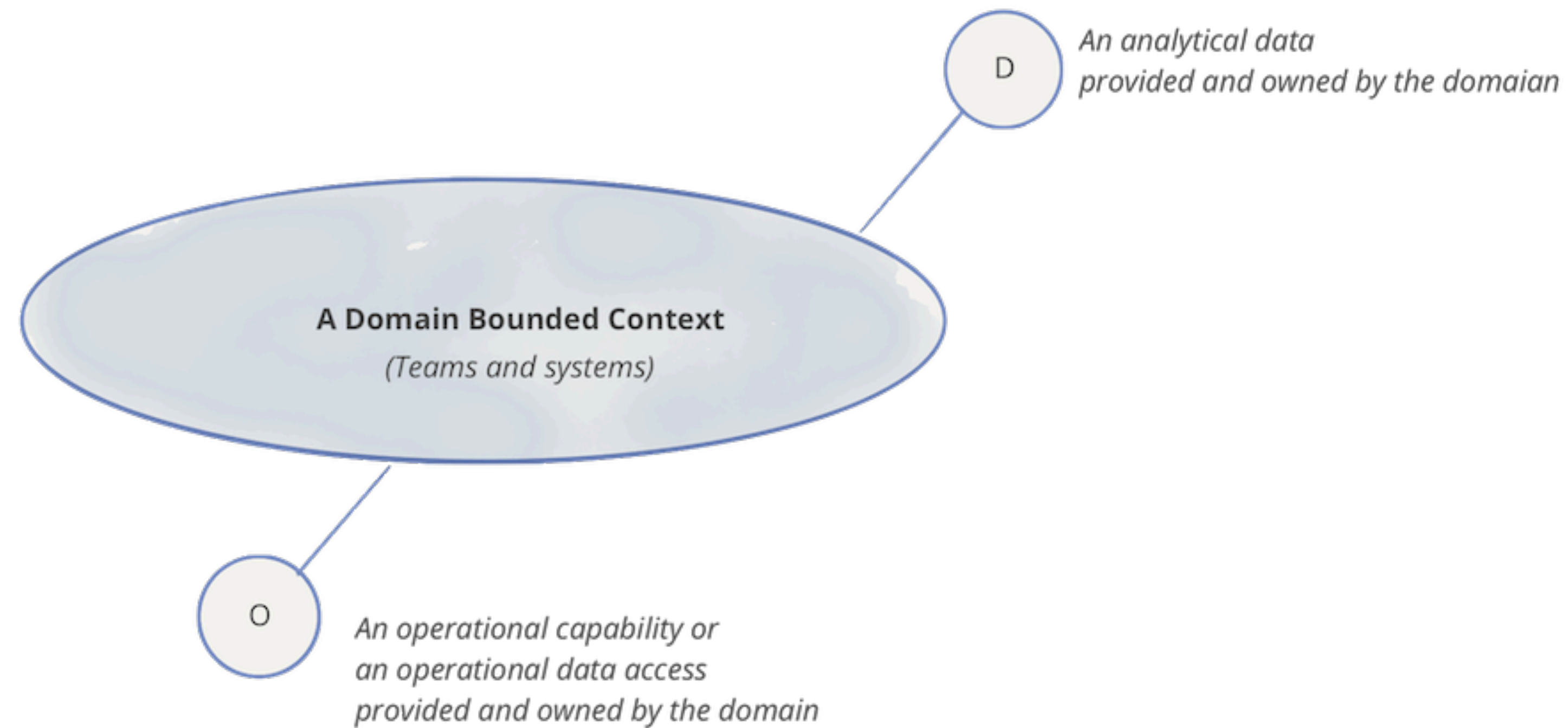
- Data can be siloed Database per Service. To circumvent, we created a data lake or shared database
- Problem is that the data lake, ended up as a data monolith
- By using a shared storage, we lost governance, and ownership
- We wanted to apply bounded context patterns, and did so with services, but lost the point with data.

The Solution

- There are two patterns: Database per Service, and Shared Database
- Data Mesh is an attempt at bringing not only the database but owning all data
- Data Mesh is about providing data ownership per context, and follows these principles:
 - Domain-oriented decentralized data ownership and architecture
 - Data as a product
 - Self-serve data infrastructure as a platform
 - Federated computational governance
 - The accountability of data quality shifts upstream as close to the source of the data as possible.

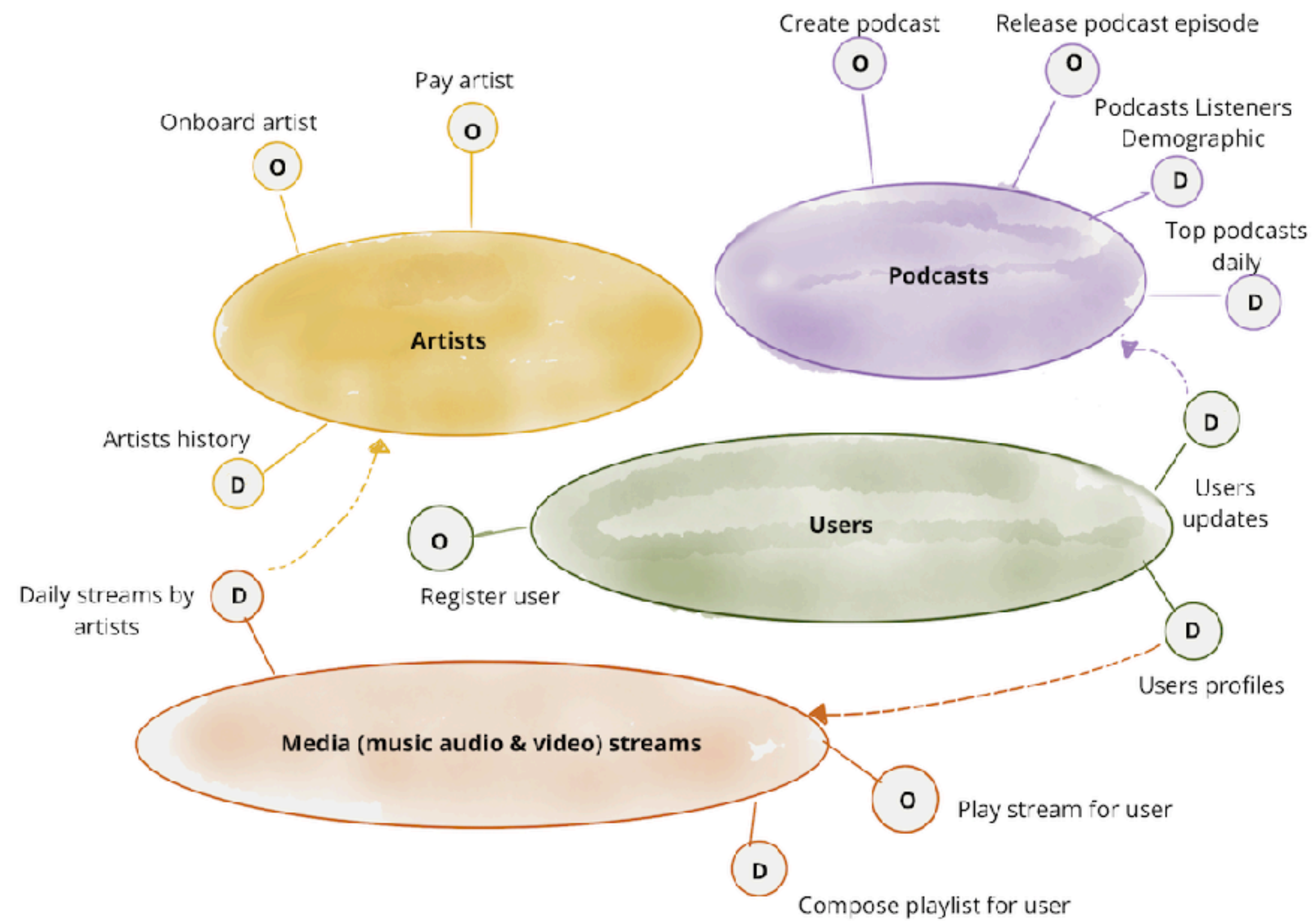
The corps was composed of all arms of the service, was **self-sustaining**, and could fight on its own until other corps could join in the battle. The corps itself was a headquarters to which units could be attached. **It might have attached two to four divisions of infantry with their organic artillery, it had its own cavalry division and corps artillery, plus support units.** With this organization a corps was expected to be able to hold its ground against, or fight off an enemy army for at least a day, when neighboring corps could come to its aid. "Well handled, it can fight or alternatively avoid action, and maneuver according to circumstances without any harm coming to it, because an opponent cannot force it to accept an engagement but if it chooses to do so it can fight alone for a long time."

The Diagram



<https://martinfowler.com/articles/data-mesh-principles.html>

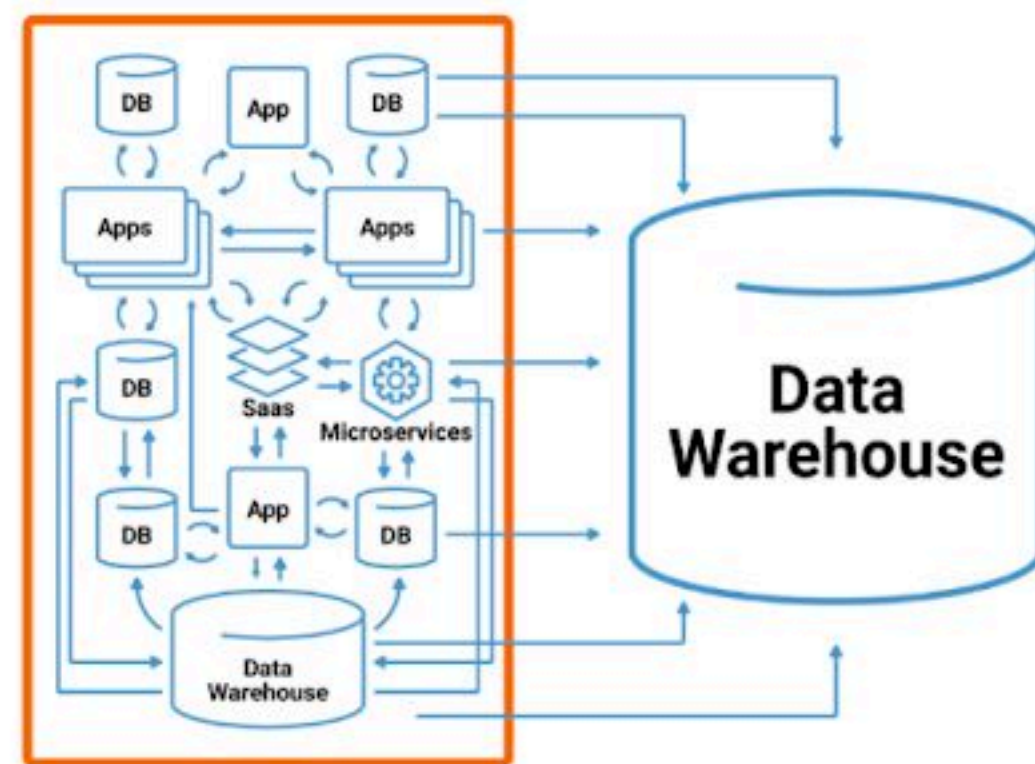
The Diagram



<https://martinfowler.com/articles/data-mesh-principles.html>

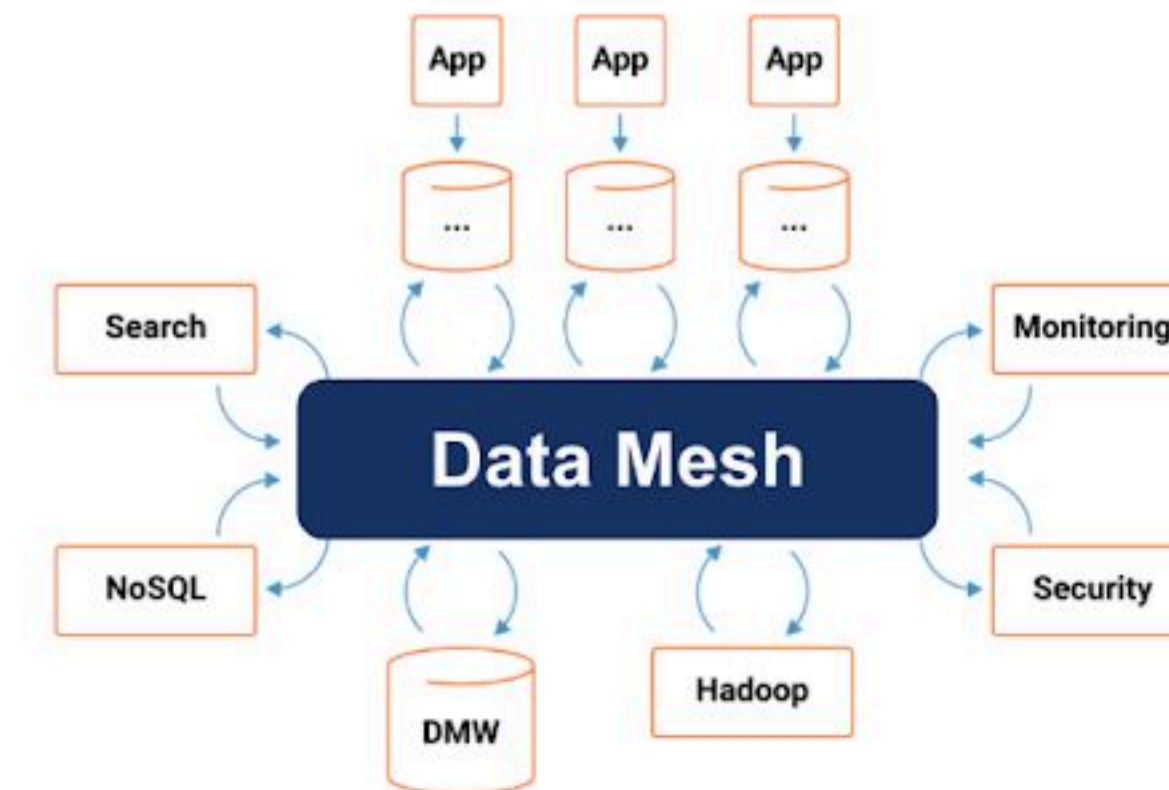
Getting out of Warehouse

Anti-pattern: responsibility for data becomes the domain of the DWH team



Centralized
Data Ownership

Pattern: Ownership of a data asset given to the “local” team that is most familiar with it



Decentralized
Data Ownership

<https://www.confluent.io/blog/how-to-build-a-data-mesh-using-event-streams/>

The Diagram

Domain-oriented decentralized data ownership and architecture

So that the ecosystem creating and consuming data can scale out as the number of sources of data, number of use cases, and diversity of access models to the data increases; simply increase the autonomous nodes on the mesh.

Data as a product

So that data users can easily discover, understand and securely use high quality data with a delightful experience; data that is distributed across many domains.

Self-serve data infrastructure as a platform

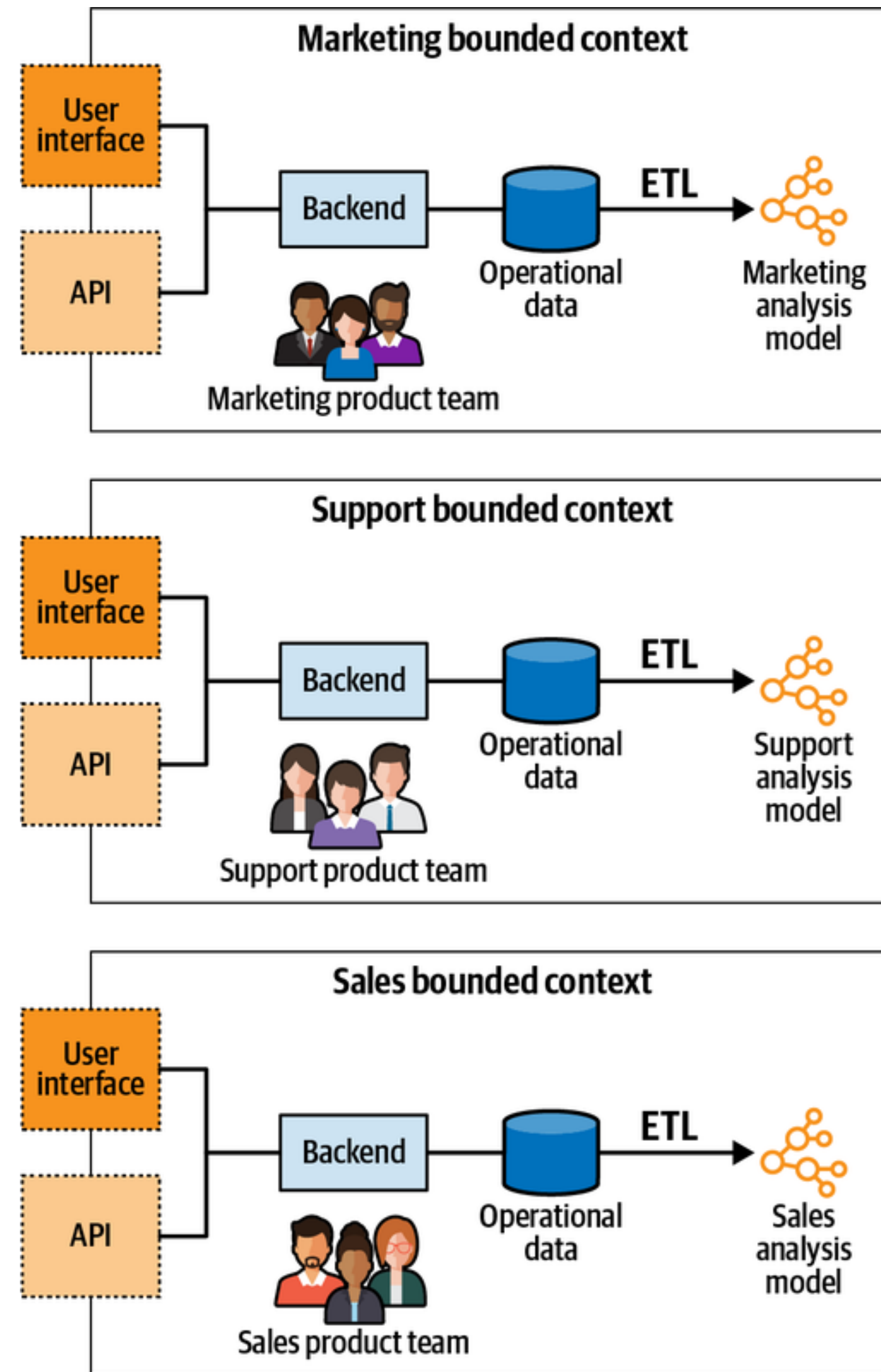
So that the domain teams can create and consume data products autonomously using the platform abstractions, hiding the complexity of building, executing and maintaining secure and interoperable data products.

Federated computational governance

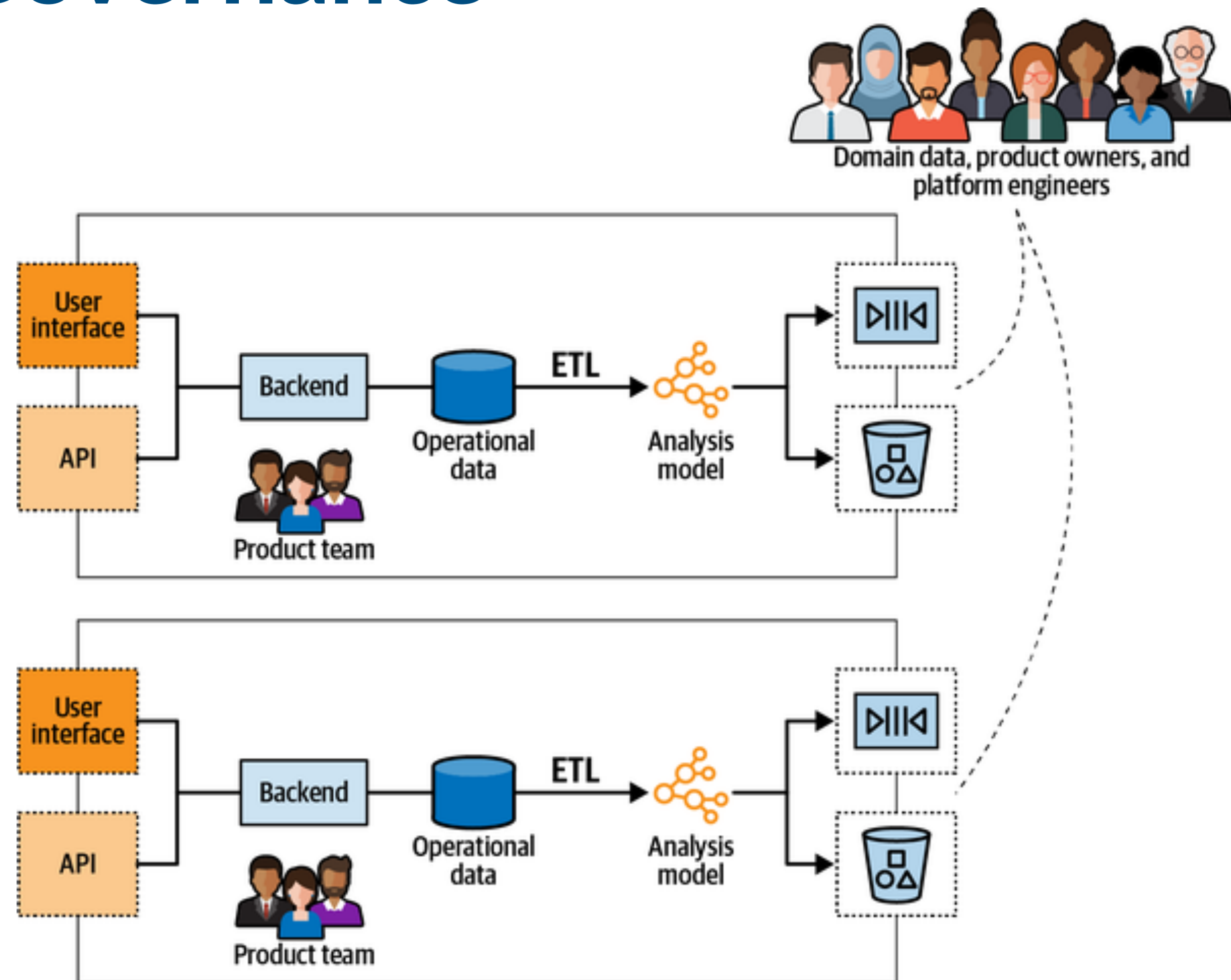
So that data users can get value from aggregation and correlation of independent data products - the mesh is behaving as an ecosystem following global interoperability standards; standards that are baked computationally into the platform.

<https://martinfowler.com/articles/data-mesh-principles.html>

The Diagram



Federated Governance



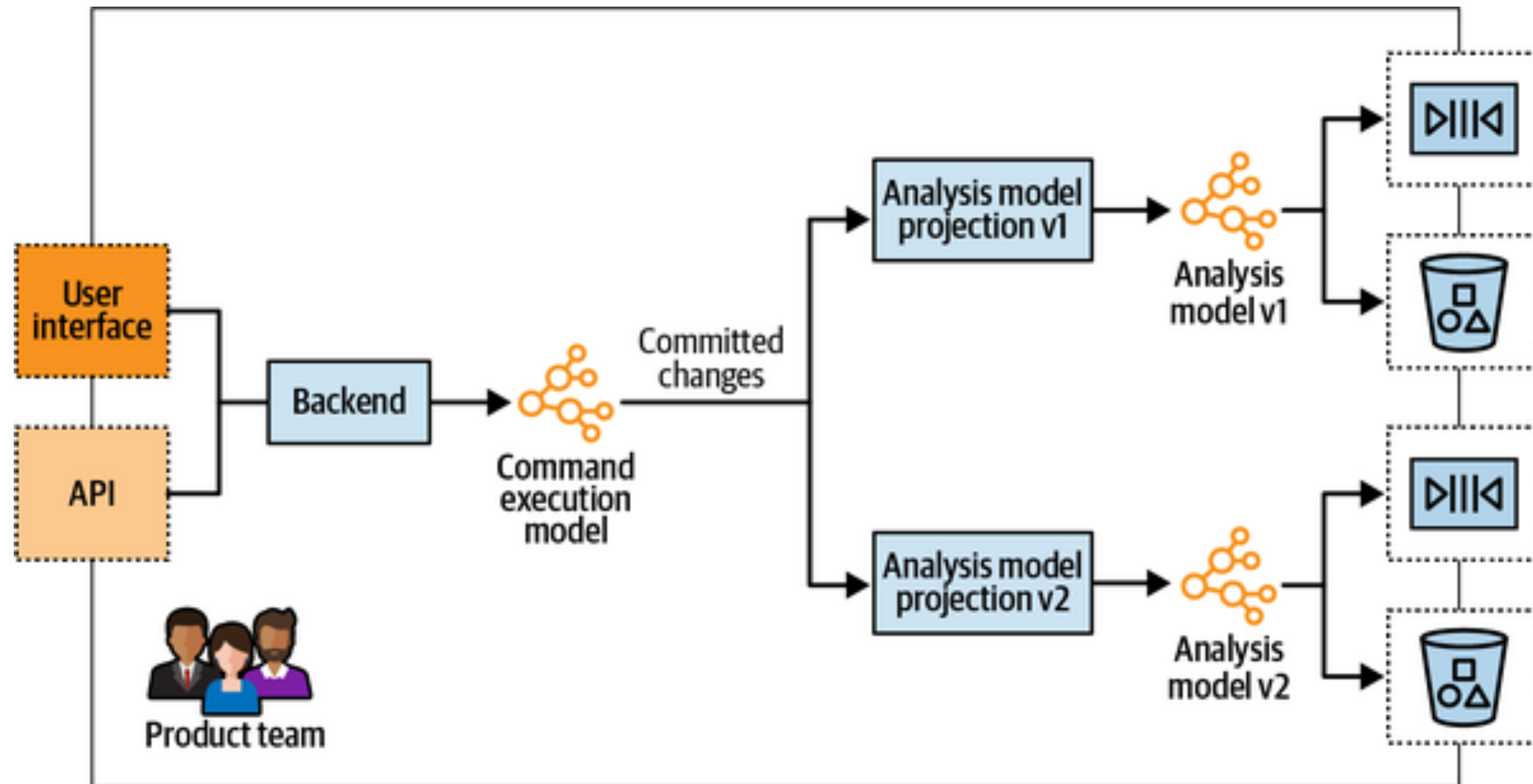
Federated Governance Duty

- The governance group is in charge of defining the rules to ensure a healthy and interoperable ecosystem.
- The rules have to be applied to all data products and their interfaces, and it's the group's responsibility to ensure adherence to the rules throughout the enterprise.

Data Mesh with Domain Driven Design

- Data mesh architecture is based on the same reasoning as domain-driven design
- Ubiquitous language and the resultant domain knowledge are essential for designing analytical models
- Exposing a bounded context's data in a model that is different from its operational model is the open-host pattern
- The CQRS pattern makes it easy to generate multiple models of the same data. It can be leveraged to transform the operational model into an analytical model.

Having different models of the data



Demo: Pinot



- Let's use Pinot to expose OLAP data from various sources
- Apache Pinot would be a great product to own data and expose that data to product owners

Thank You



- Email: dhinojosa@evolutionnext.com
- Github: <https://www.github.com/dhinojosa>
- Mastodon: <https://mastodon.social/@dhinojosa>
- Linked In: <http://www.linkedin.com/in/dhevolutionnext>