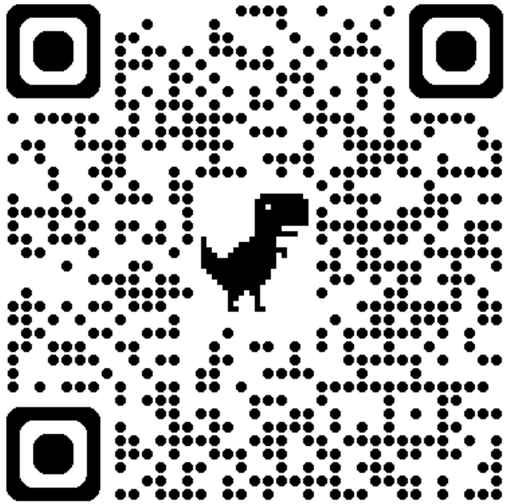
Event Driven Architecture Workshop

Daniel Hinojosa

Architectural Design Pattern Messaging Standards

In this Workshop

- Quick introduction to Domain Driven Design
- Event Sourcing / Event Driven Architecture
- Producers/Consumers
- Materialized Views
- Schema Development
- CQRS
- Saga Pattern
- Types of Events

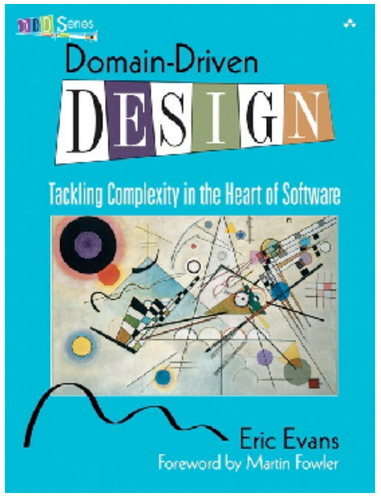


Slides and Material: https://github.com/dhinojosa/nfjs-event-driven-architecture

Quick Introduction to DDD

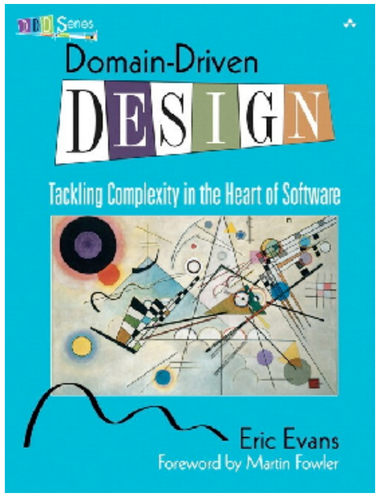
Domain Driven Design

- 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



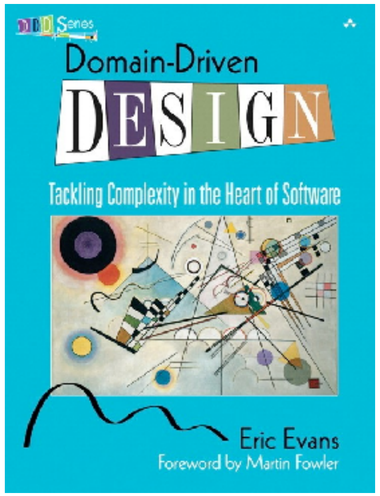
Ubiquitous Language

- 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.

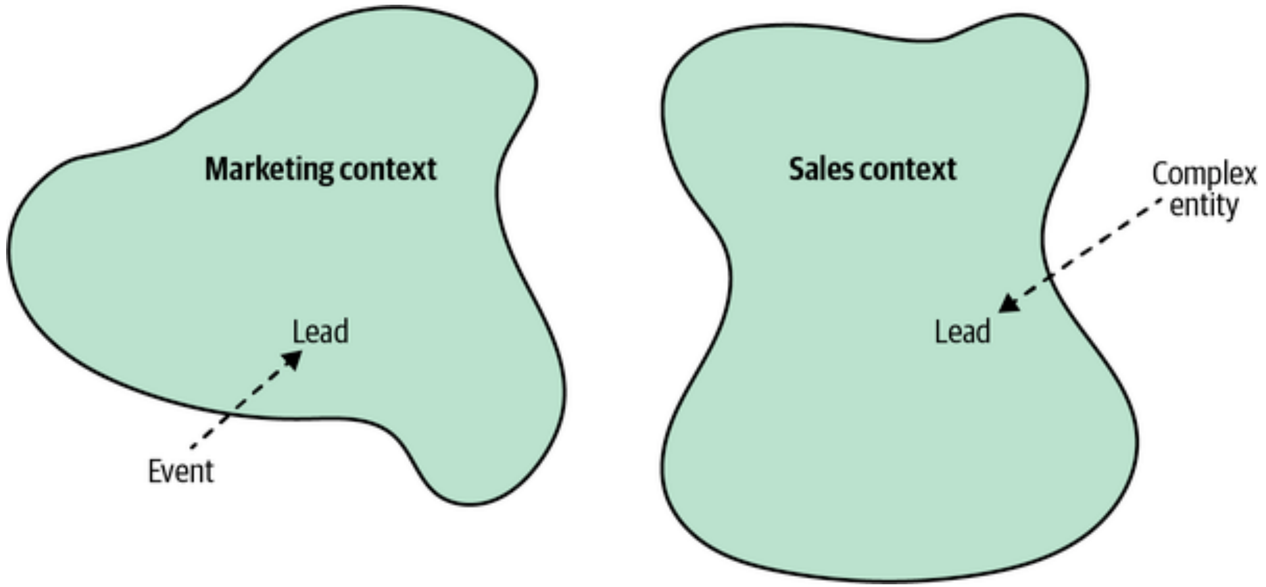


Bounded Contexts

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



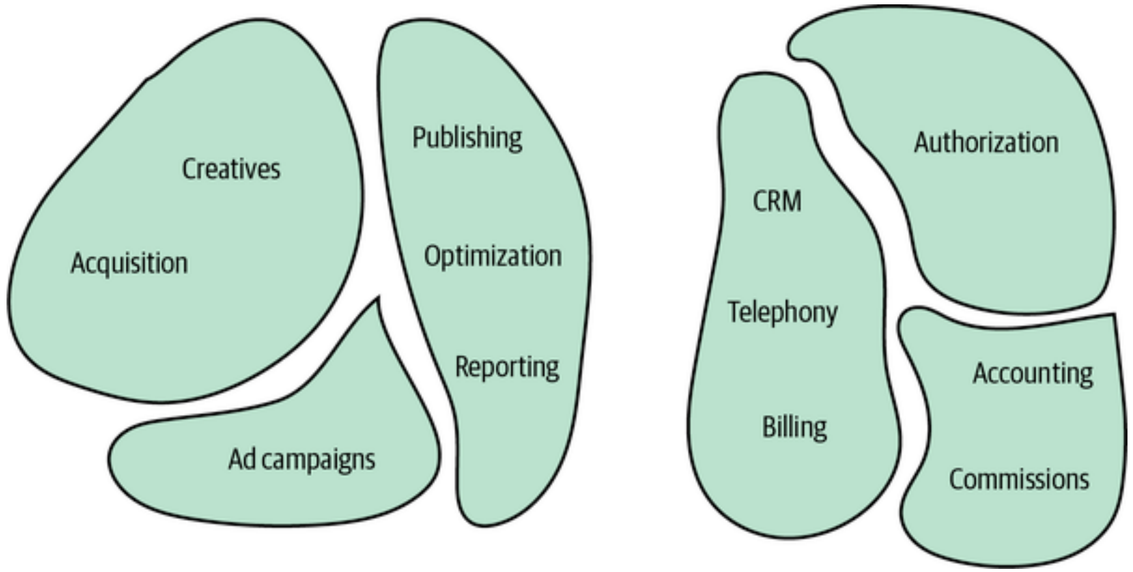
The Diagram



Learning Domain Driven Design - Vlad Khononov

The Diagram

Contexts are neither big nor small, but useful



Learning Domain Driven Design - Vlad Khononov

What goes in the bounded context?

- Developers with 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.
}
```

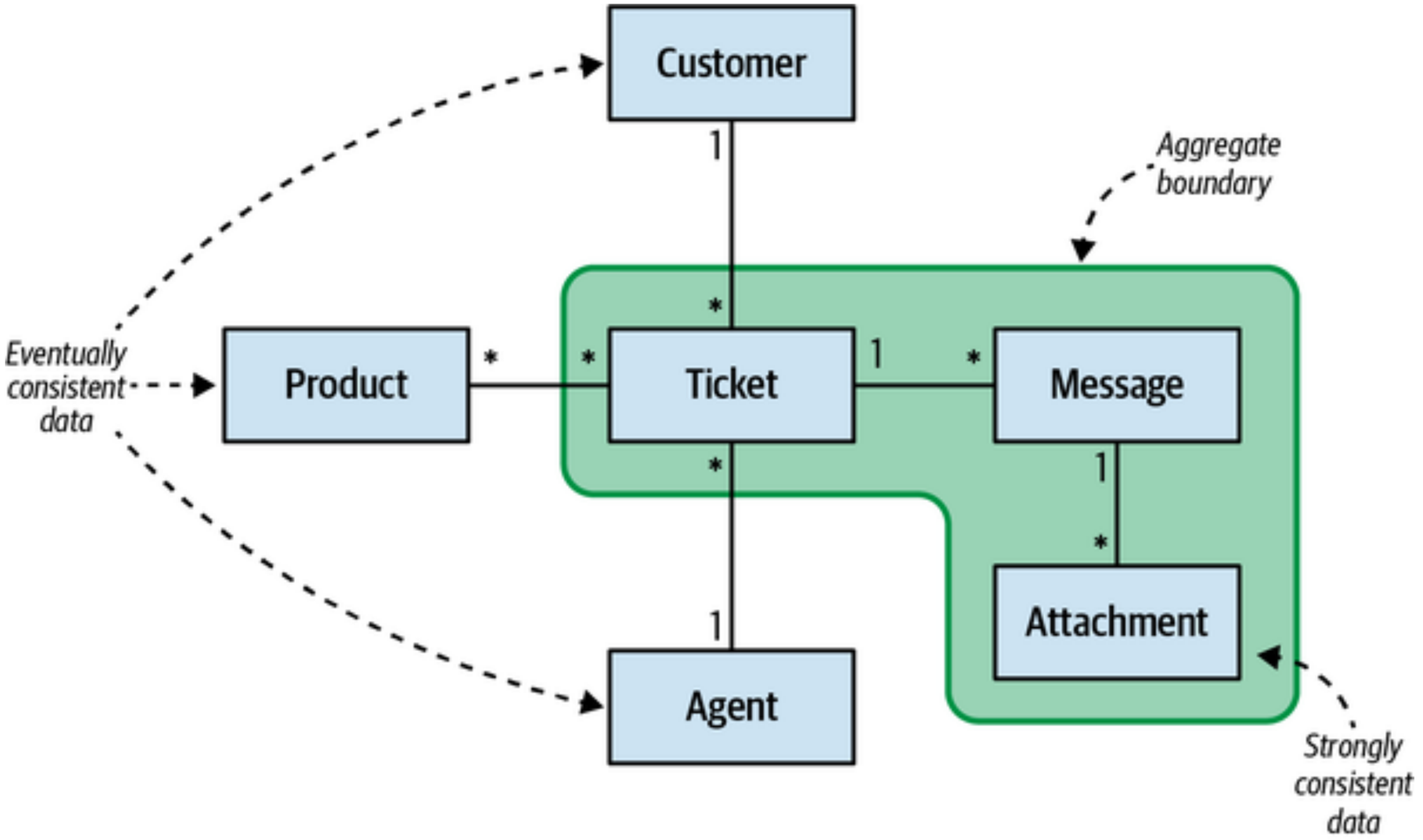
```
class Employee {
   private EmployeeId employeeId;
   private Name name;
   //Constructors, equals, hashCode, etc.
}
```

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)
   }
}
```

Aggregate Boundaries



Learning Domain Driven Design - Vlad Khononov

Keep Aggregates Small

- Keep the aggregates as small as possible and include only objects that are required to be in a strongly consistent state by the aggregate's business logic
- In the following code example, notice that we are not interested in the entire Product nor the entire User, just their identifiers (entities)
- Reasoning behind referencing external aggregates by ID is to reify that these objects do not belong to the aggregate's boundary, and to ensure that each aggregate has its own transactional boundary

```
public class Ticket {
    private UserId           customer;
    private List<ProductId> products;
    private UserId           assignedAgent;
    private List<Message>    messages;
}
```

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) {
       //...
   }
}
```

Domain Events

- Represent significant occurrences in the domain.
- Capture past tense, immutable facts: “Order Placed,” “Payment Processed.”
- Reflect business-relevant changes.
- Decouples business logic from system implementation details.

```
import java.time.Instant;

public record StockLevelUpdated(String
productId, int newStockLevel, Instant
occurredAt) {

    public StockLevelUpdated(
         String productId,
         int newStockLevel) {

           this(productId,
             newStockLevel,
             Instant.now());
    }
}
```

Lab: Writing a Aggregate

- What does an aggregate and even more so, what is a Domain Event?

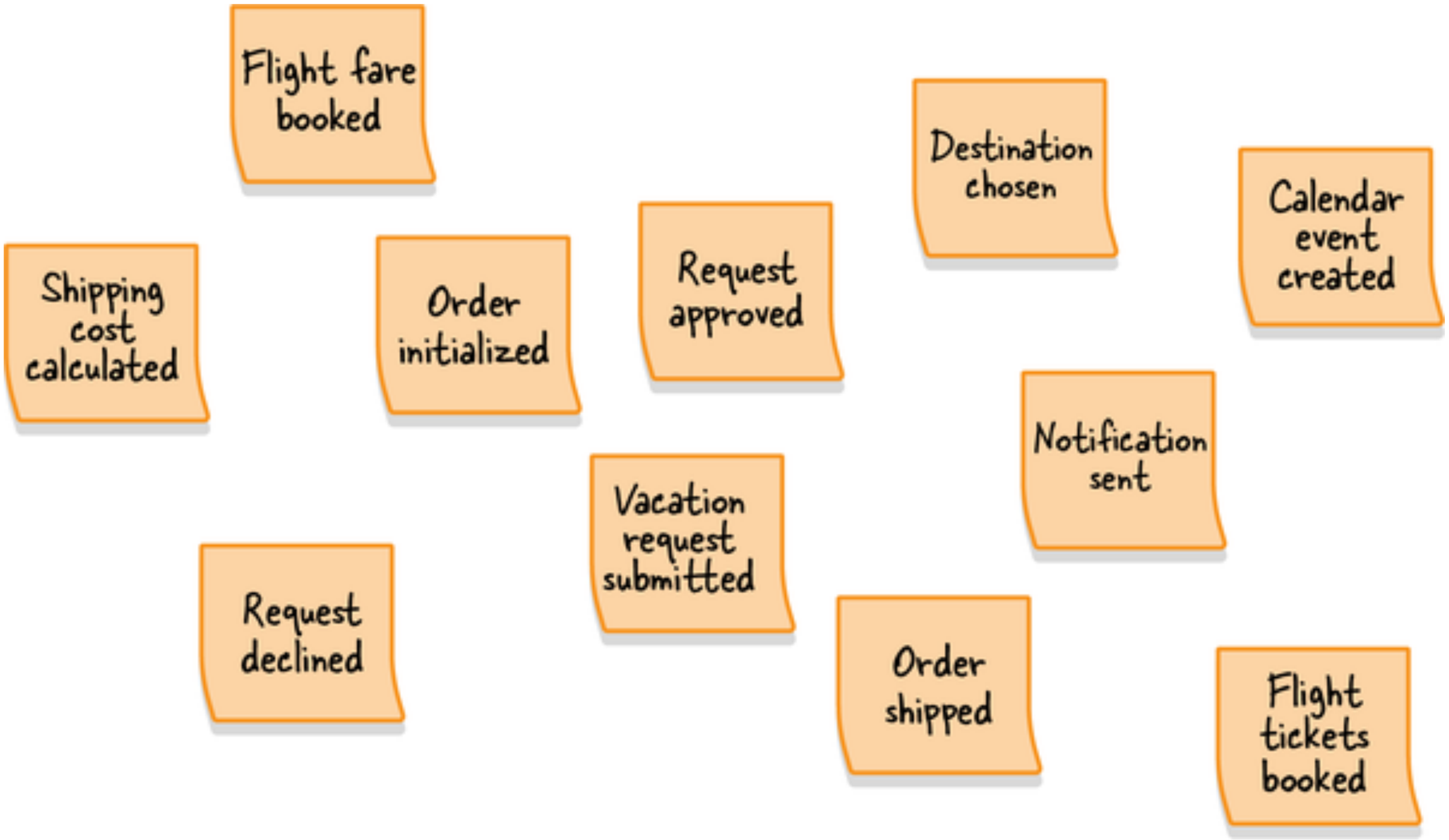
Event Storming

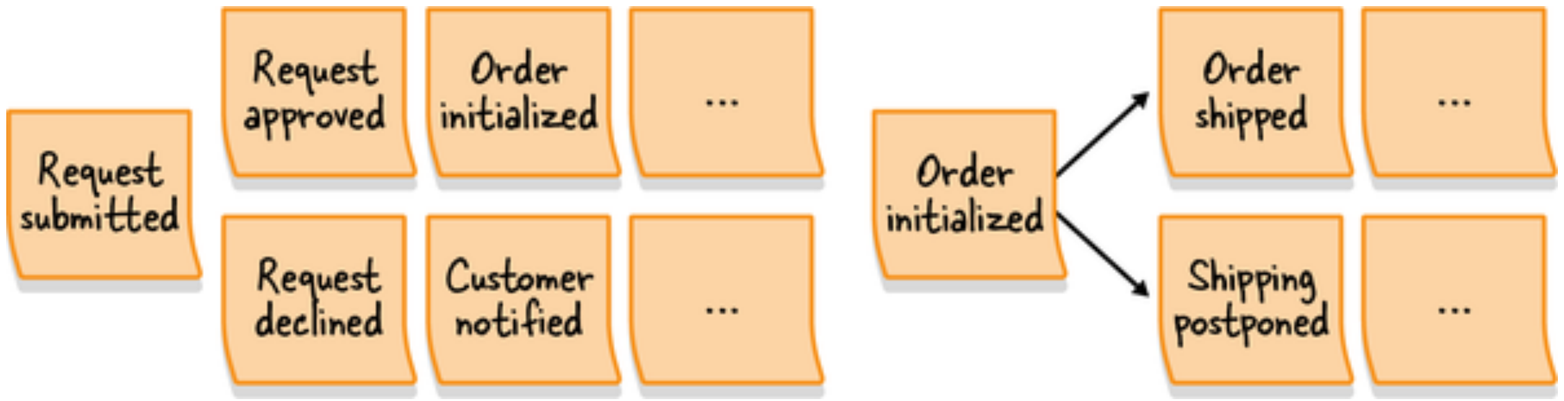
Event Storming

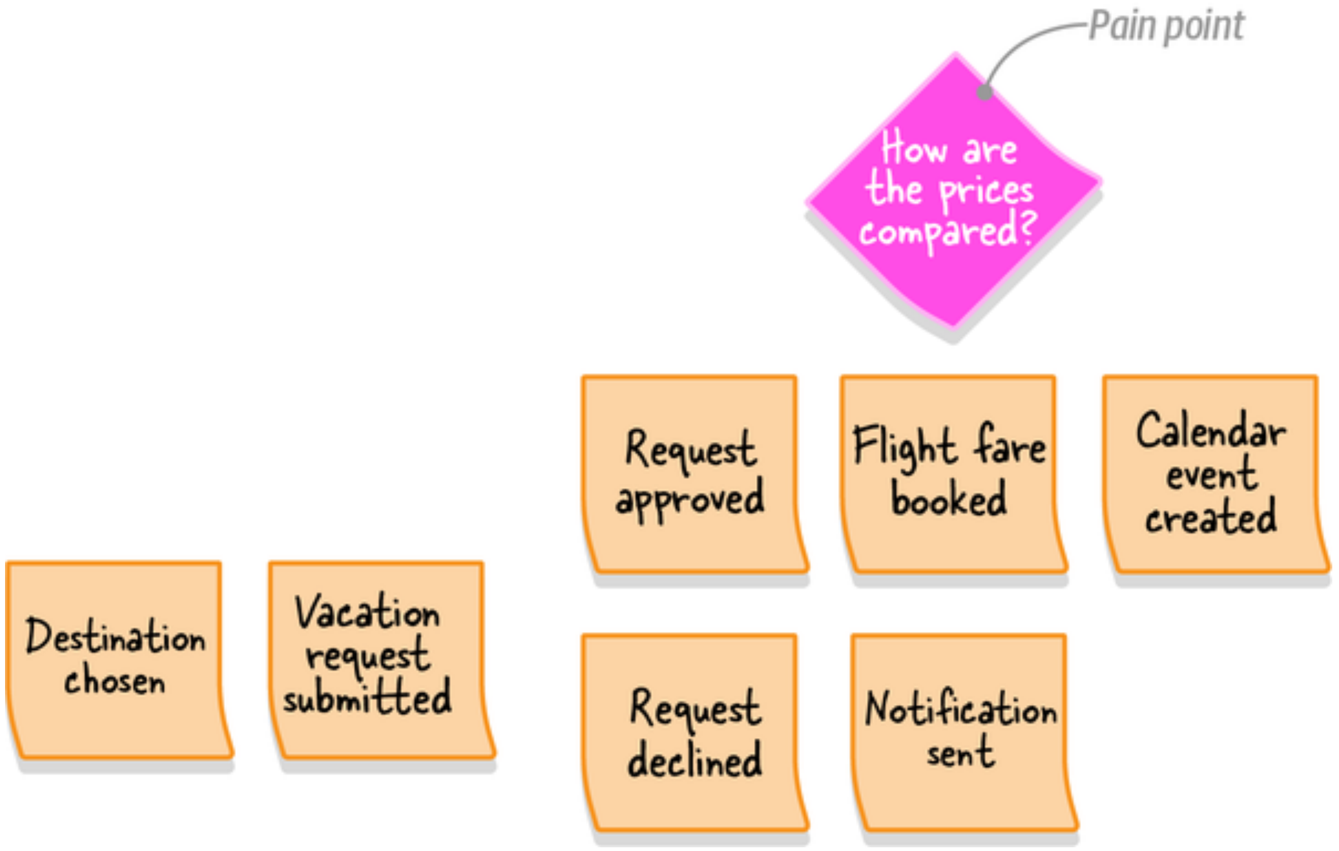
- Low-tech activity for a group of people to brainstorm and rapidly model a business process.
- In a sense, EventStorming is a tactical tool for sharing business domain knowledge.
- An EventStorming session has a scope: the business process that the group is interested in exploring.
- The participants 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.

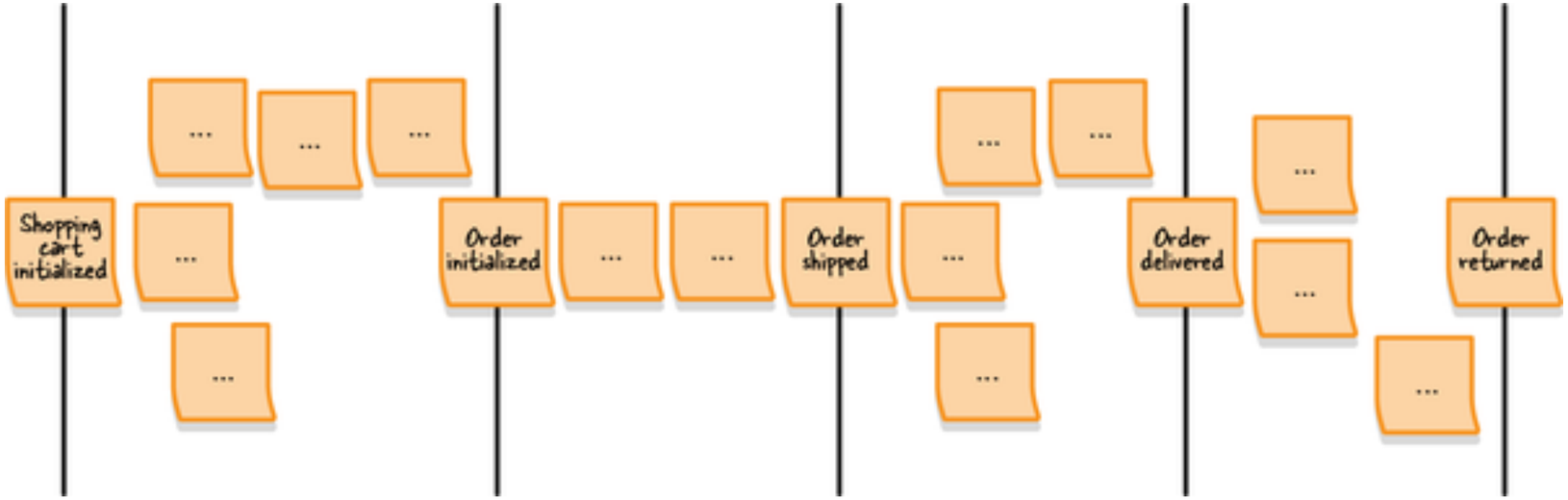
“Just keep in mind that the goal of the workshop is to learn as much as possible in the shortest time possible. We invite key people to the workshop, and we don’t want to waste their valuable time”

— Alberto Brandolini: Creator of the EventStorming workshop

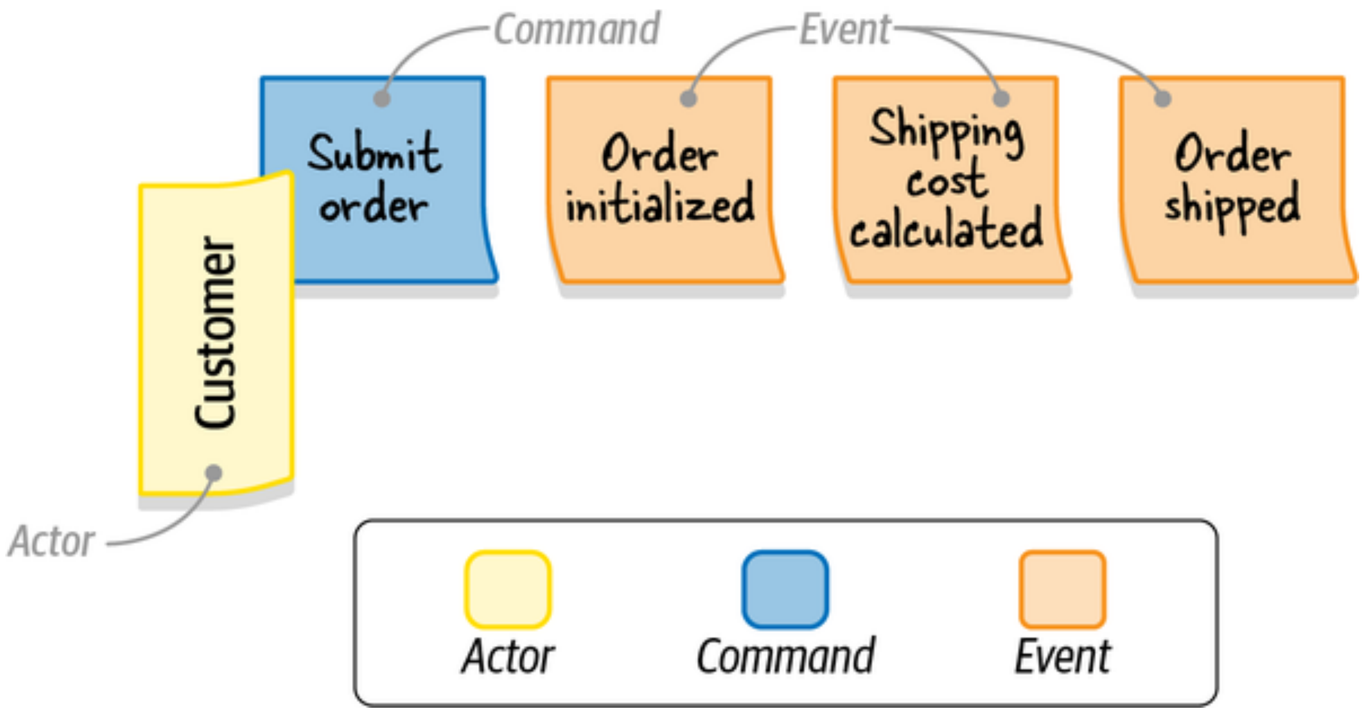

Flight fare booked
Destination chosen
Calendar event created
Shipping cost calculated
Order initialized
Request approved
Notification sent
Vacation request submitted
Request declined
Order shipped
Flight tickets booked

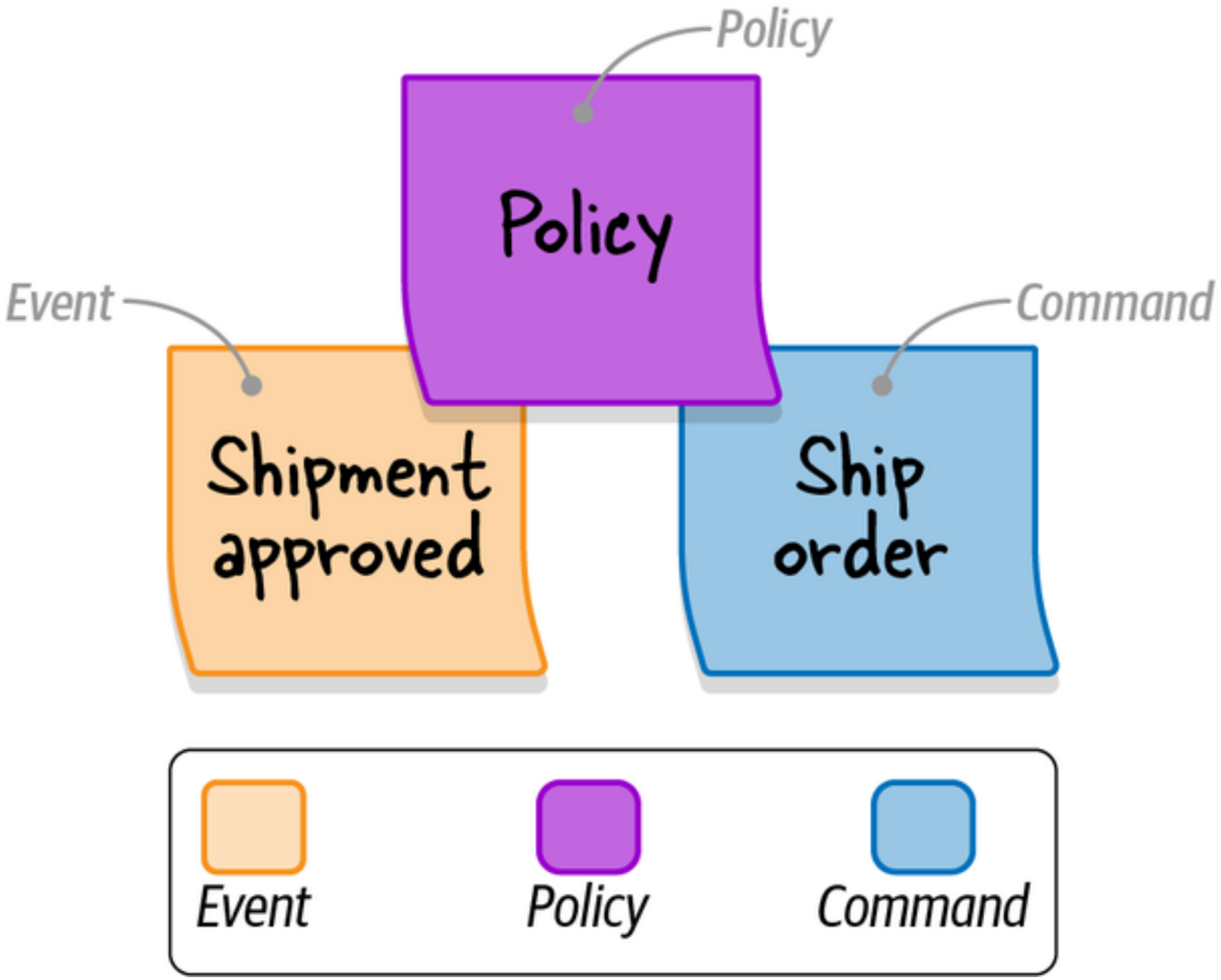

Request submitted
Request approved
Order initialized
...
Request declined
Customer notified
...
Order initialized
Order shipped
...
Shipping postponed
...

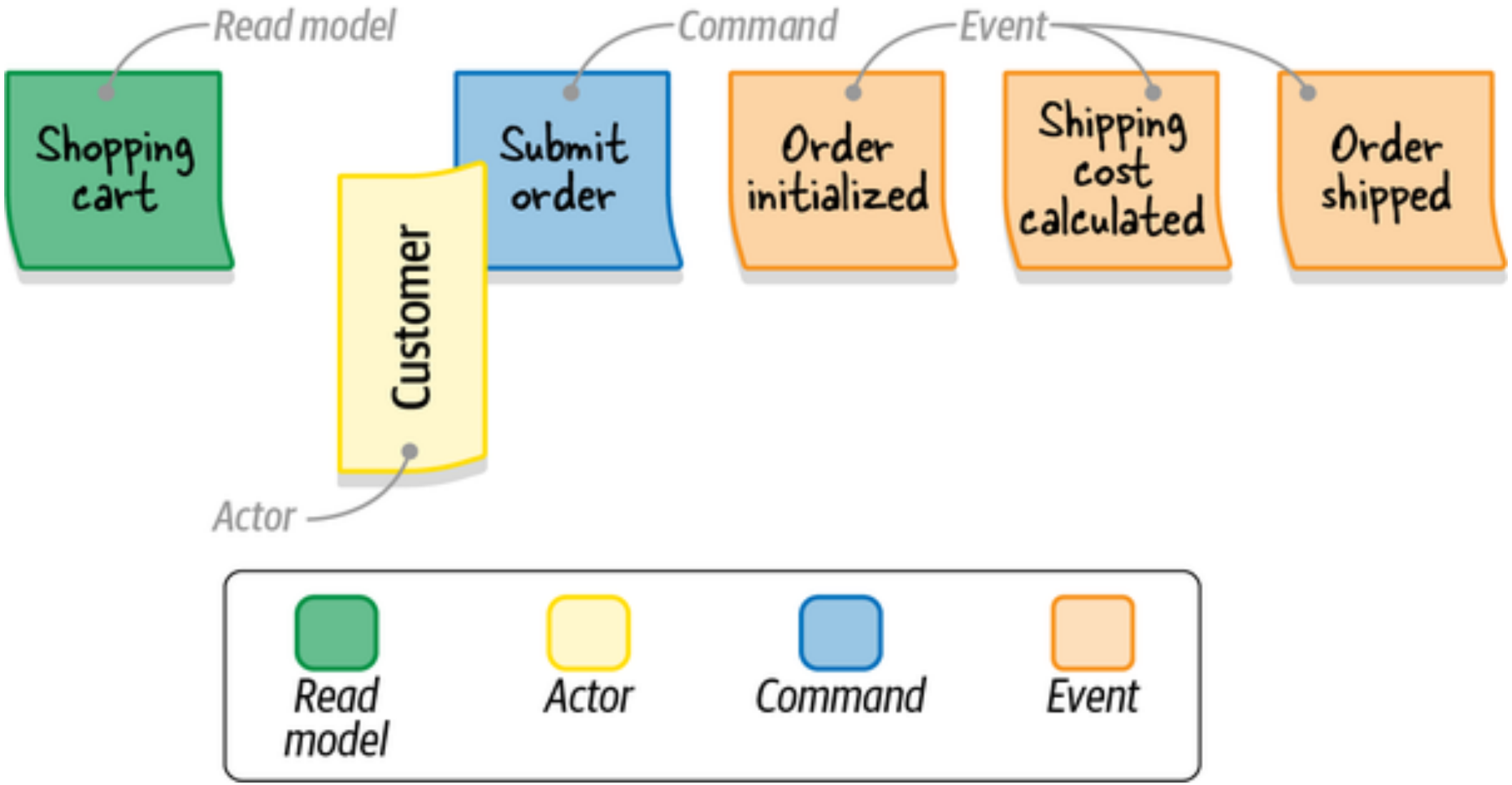

Pain point
How are the prices compared?
Request approved
Flight fare booked
Calendar event created
Destination chosen
Vacation request submitted
Request declined
Notification sent

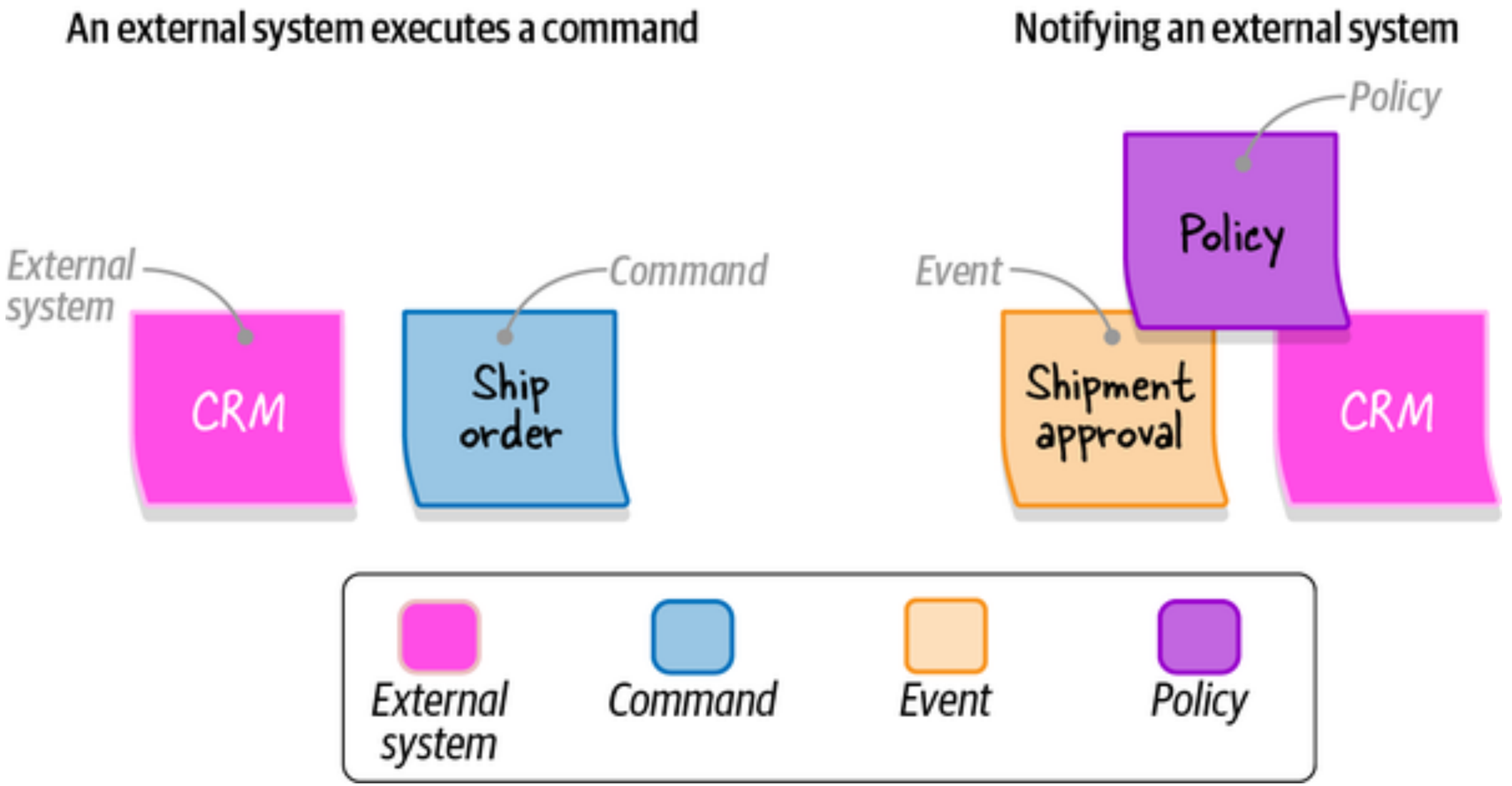


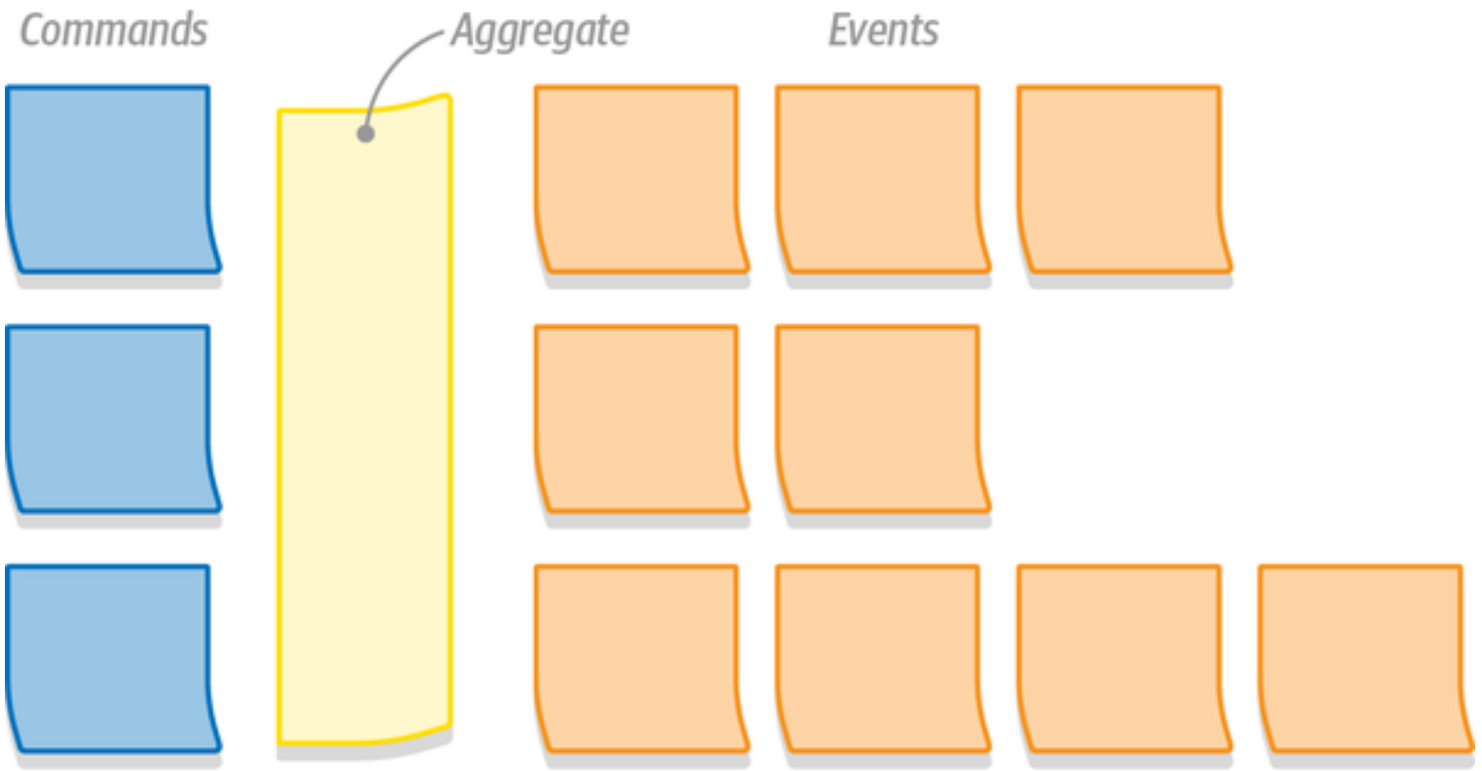
Learning Domain Driven Design - Vlad Khononov


Command
Event
Submit order
Order initialized
Shipping cost calculated
Order shipped
Customer
Actor
Actor
Command
Event

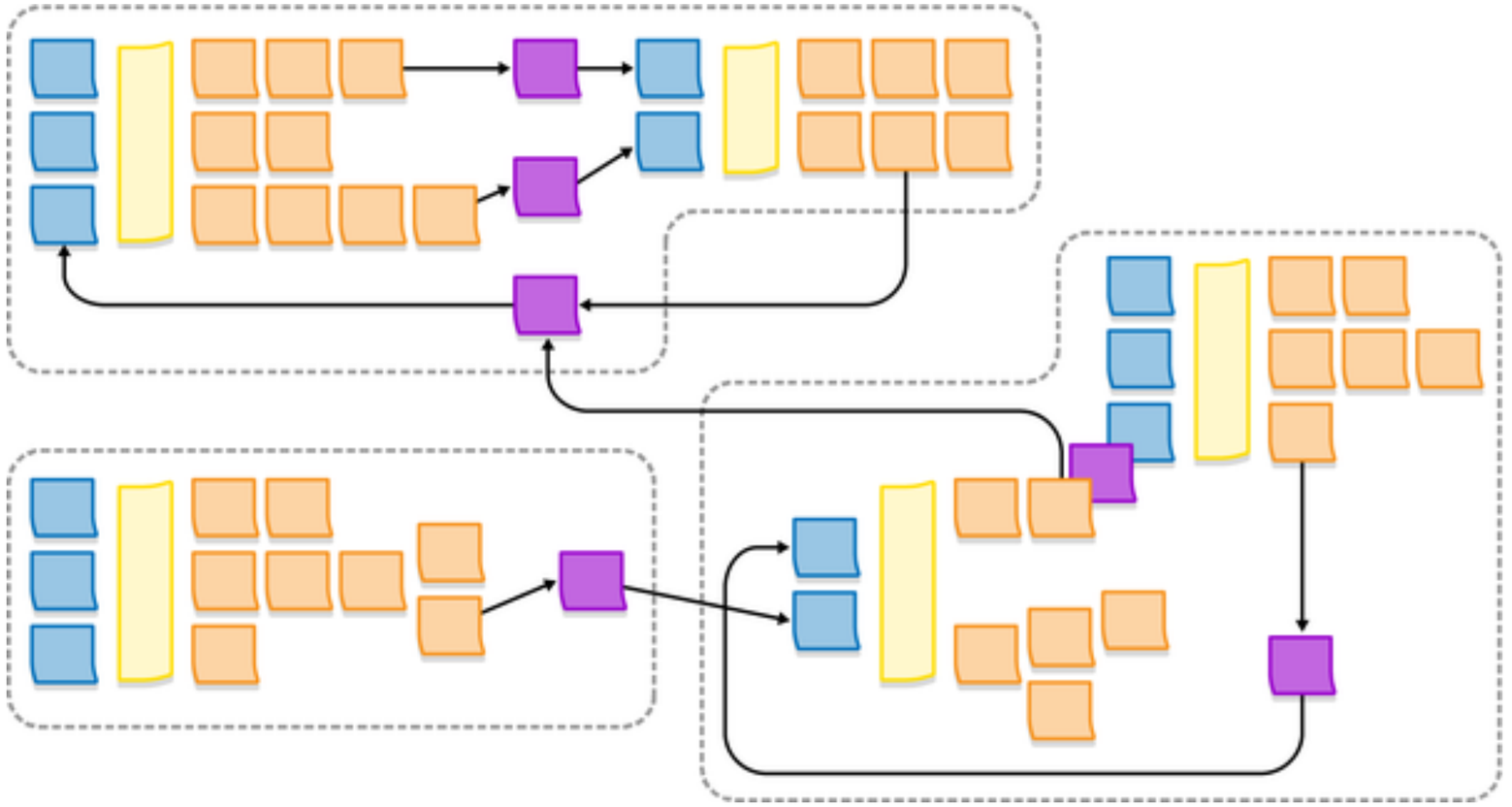

Policy
Policy
Event
Shipment approved
Command
Ship order
Event
Policy
Command


Read model
Command
Event
Shopping cart
Submit order
Order initialized
Shipping cost calculated
Order shipped
Customer
Actor
Read model
Actor
Command
Event


An external system executes a command
External system
CRM
Command
Ship order
Notifying an external system
Policy
Policy
Event
Shipment approval
CRM
External system
Command
Event
Policy

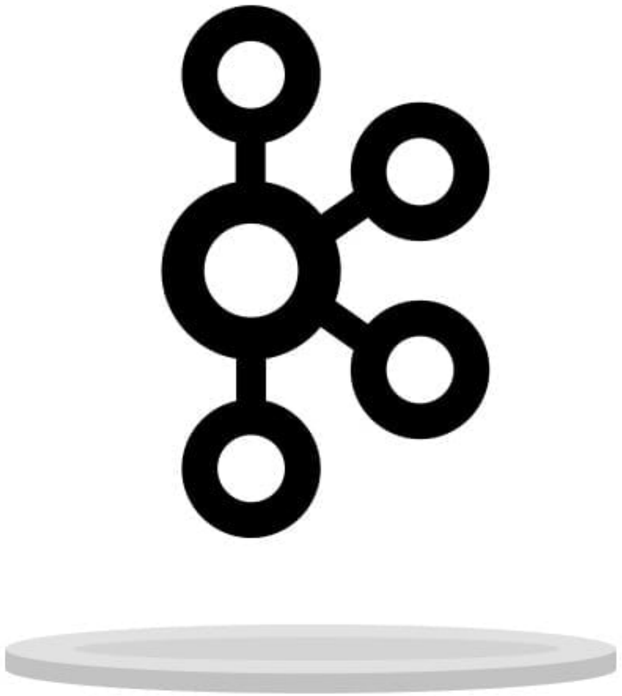


Learning Domain Driven Design - Vlad Khononov



Learning Domain Driven Design - Vlad Khononov

Producers and Consumers



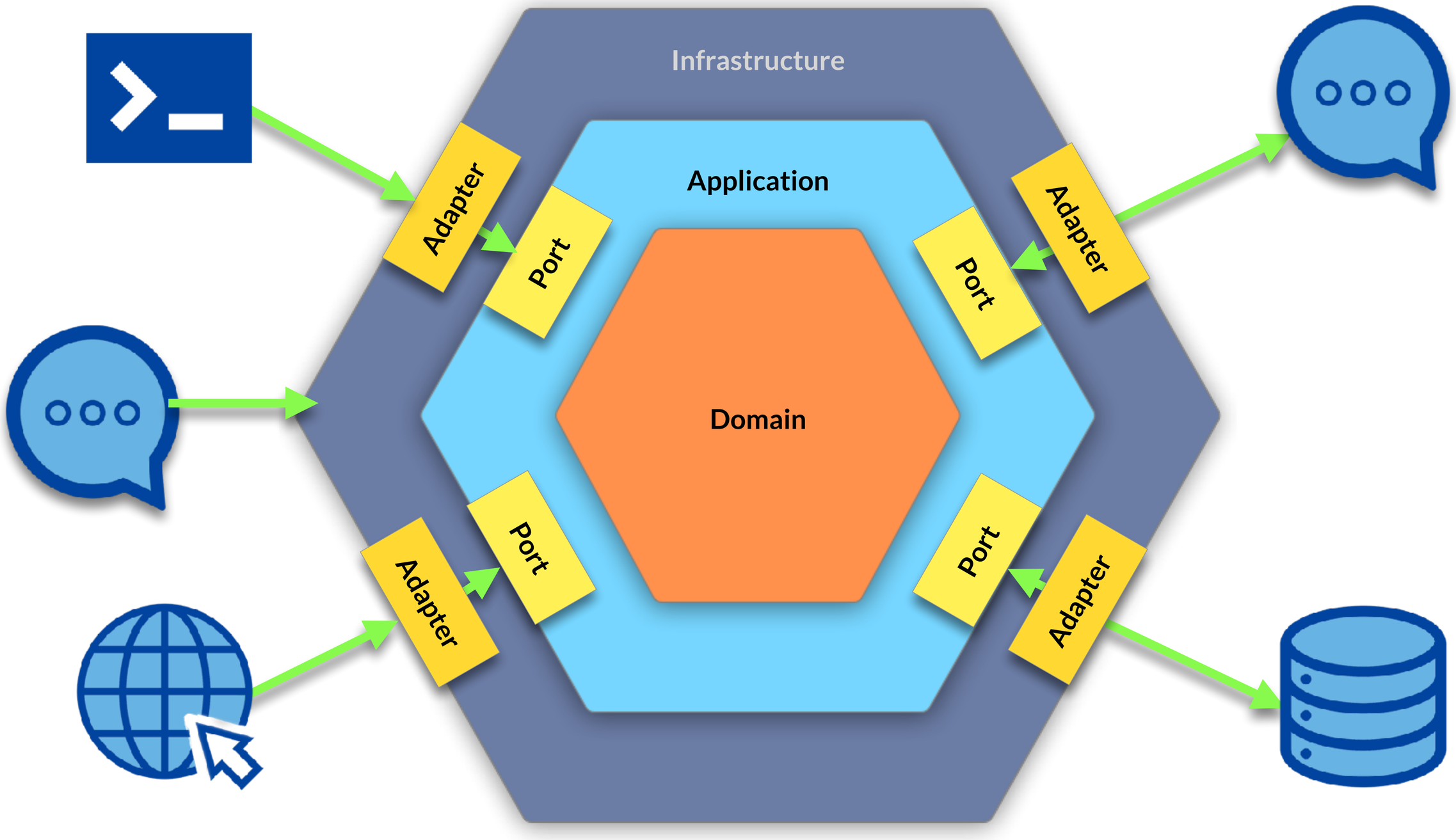
Why Kafka to demonstrate EDA?

- Kafka is a pub/sub designed to handle large volumes of data with minimal latency, making it ideal for real-time event streaming.
- Its distributed nature allows horizontal scaling, which supports high throughput.
- Kafka maintains a durable log of all events, which can be replayed by consumers.
- Kafka integrates seamlessly with a wide range of tools and frameworks such as Kafka Streams, KSQL, and connectors for databases, storage systems, and analytics platforms.
- Multi-use cases: Event Sourcing, CQRS, Stream Processing

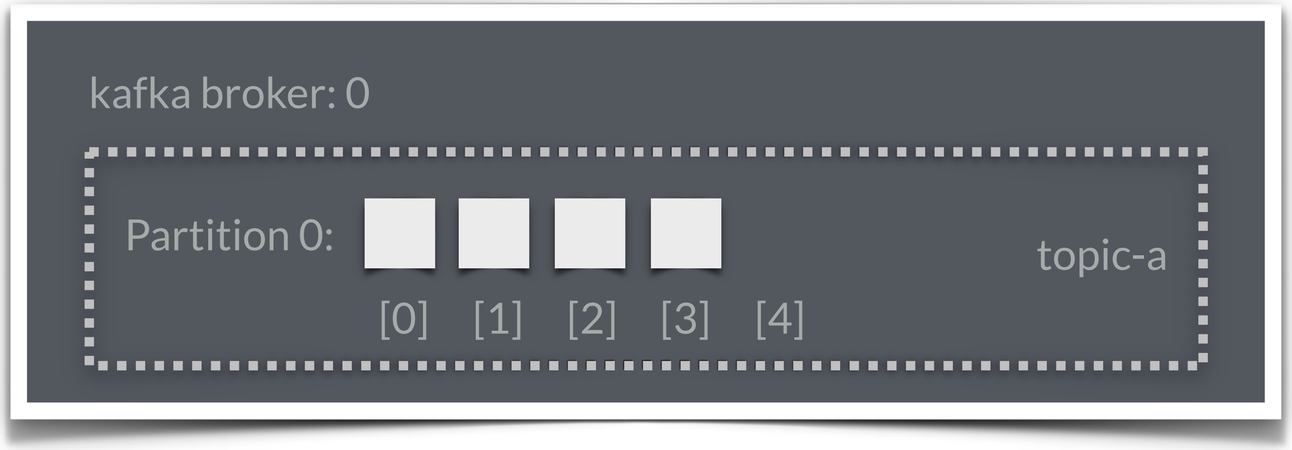
Before we get into EDA though

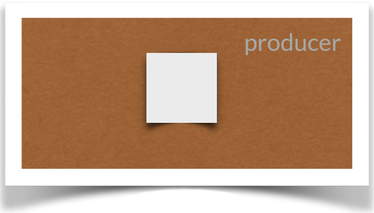
- Let’s just understand how to use Kafka
  - Producers
  - Consumers
- Then we will use these APIs so we can perform
  - Event Sourcing
  - Event Driven Architecture
  - Schema Development
  - CQRS

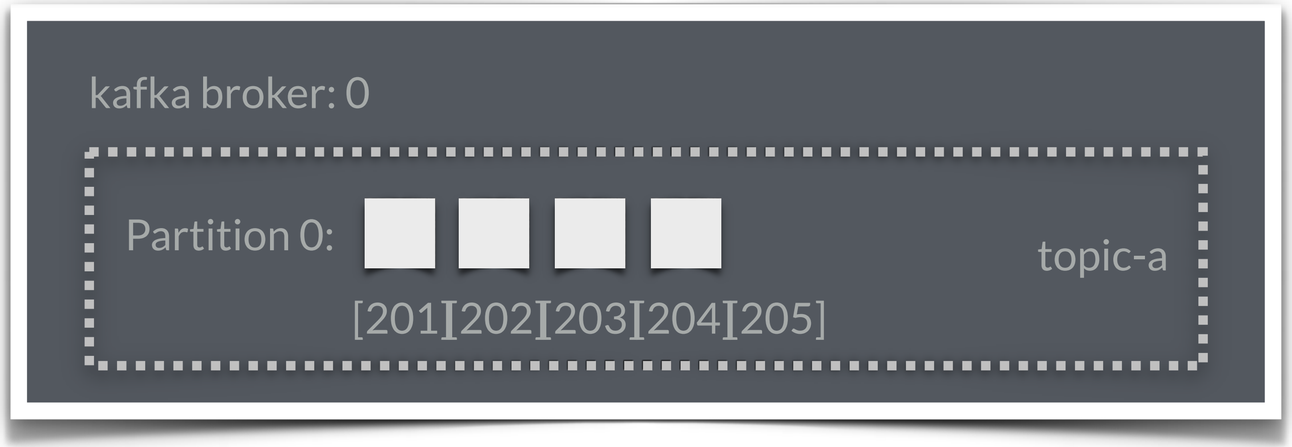
Ports and Adapters

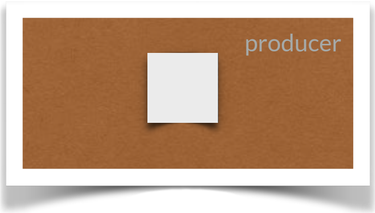

Infrastructure
Application
Domain
Adapter
Port
Adapter
Port
Port
Adapter
Port
Adapter

Producers


kafka broker: 0
Partition 0:
topic-a
[0]
[1]
[2]
[3]
[4]


producer

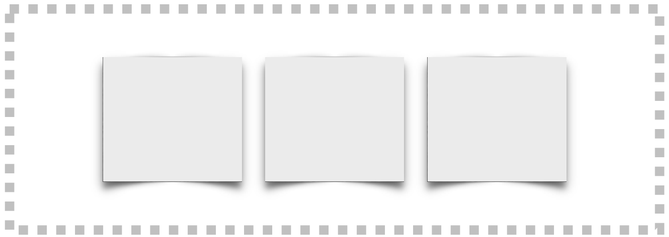


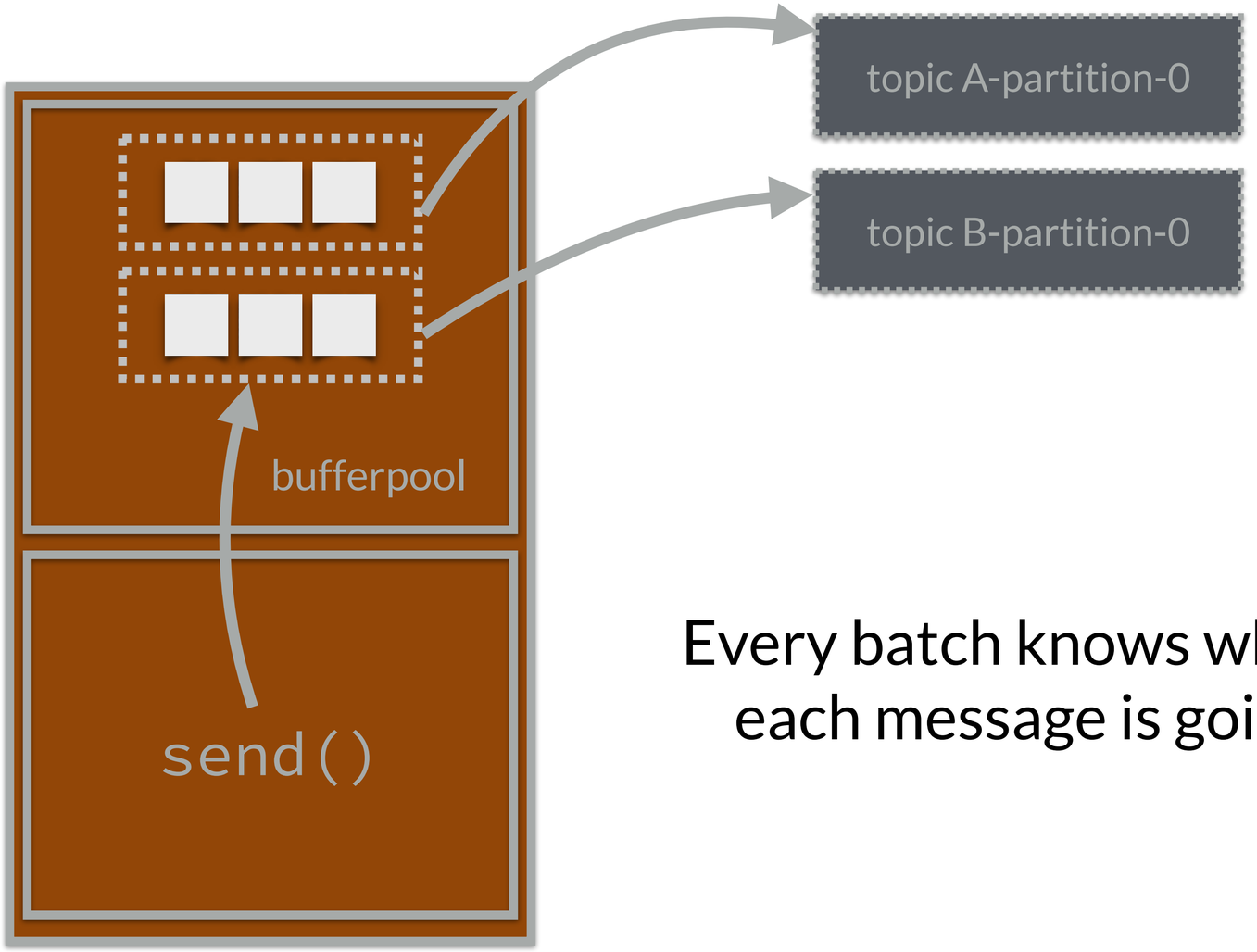


Retention: The **data is temporary**

Kafka Batch

- A collection of messages, that is sent, configured in *bytes*
- Sent to the same *topic* and the same *partition*
- *Avoids overhead of sending multiple message over the wire*





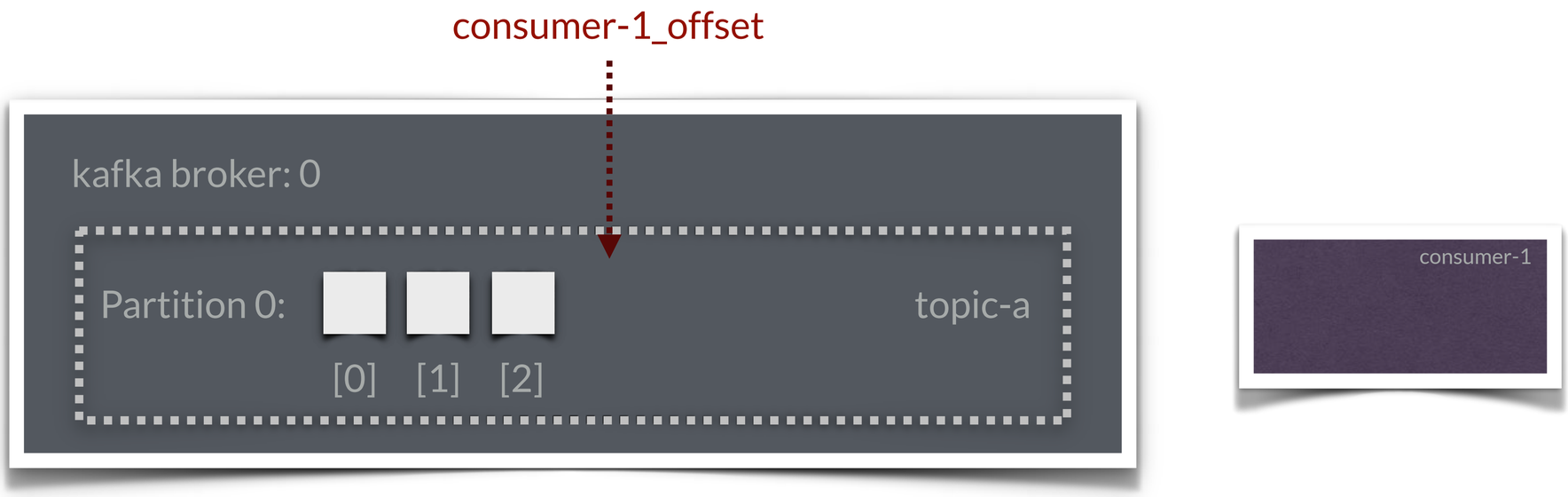
Every batch knows where each message is going

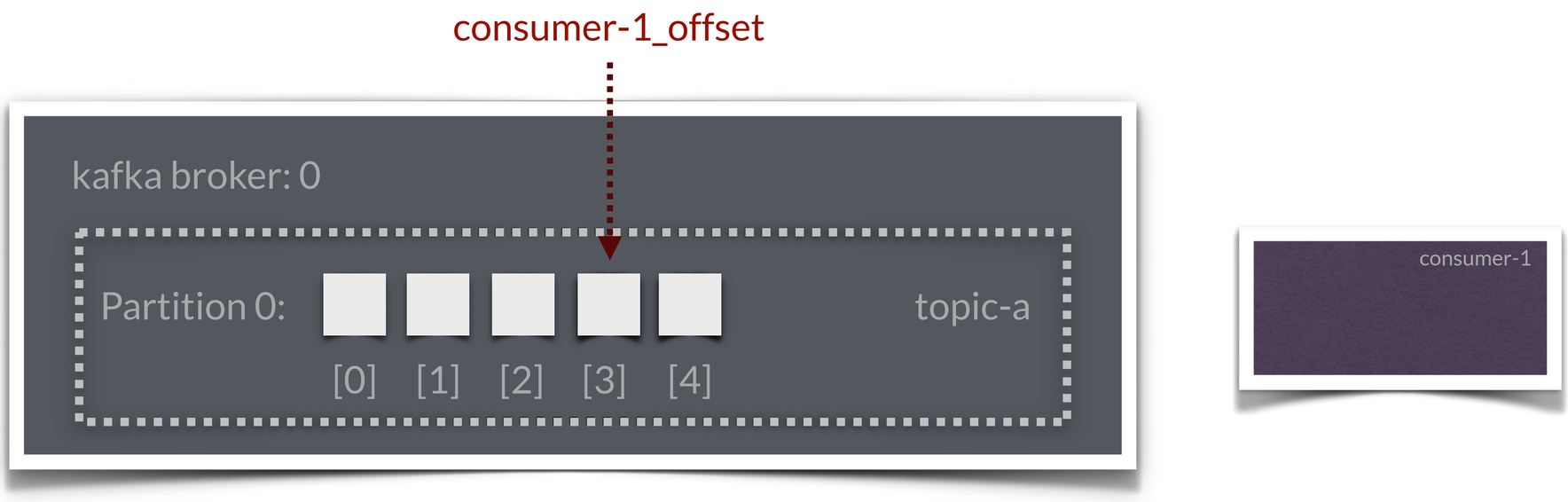
```
murmur2(bytes) % number partitions
```

Lab: Writing a Producer

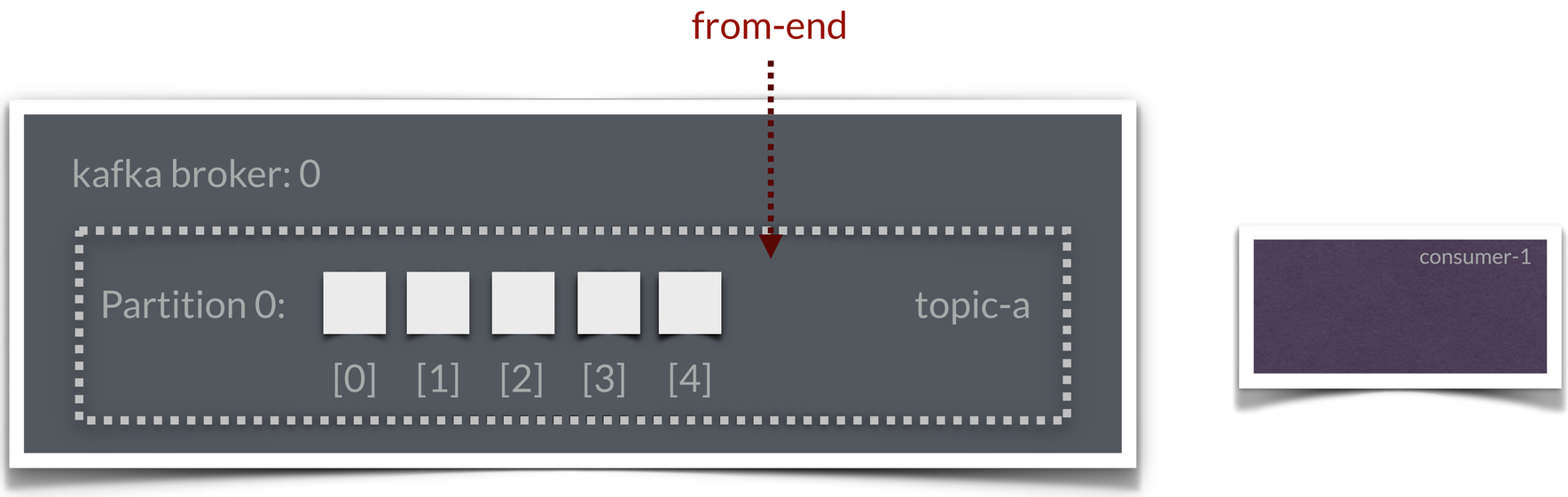
- Let’s create a producer in Kafka

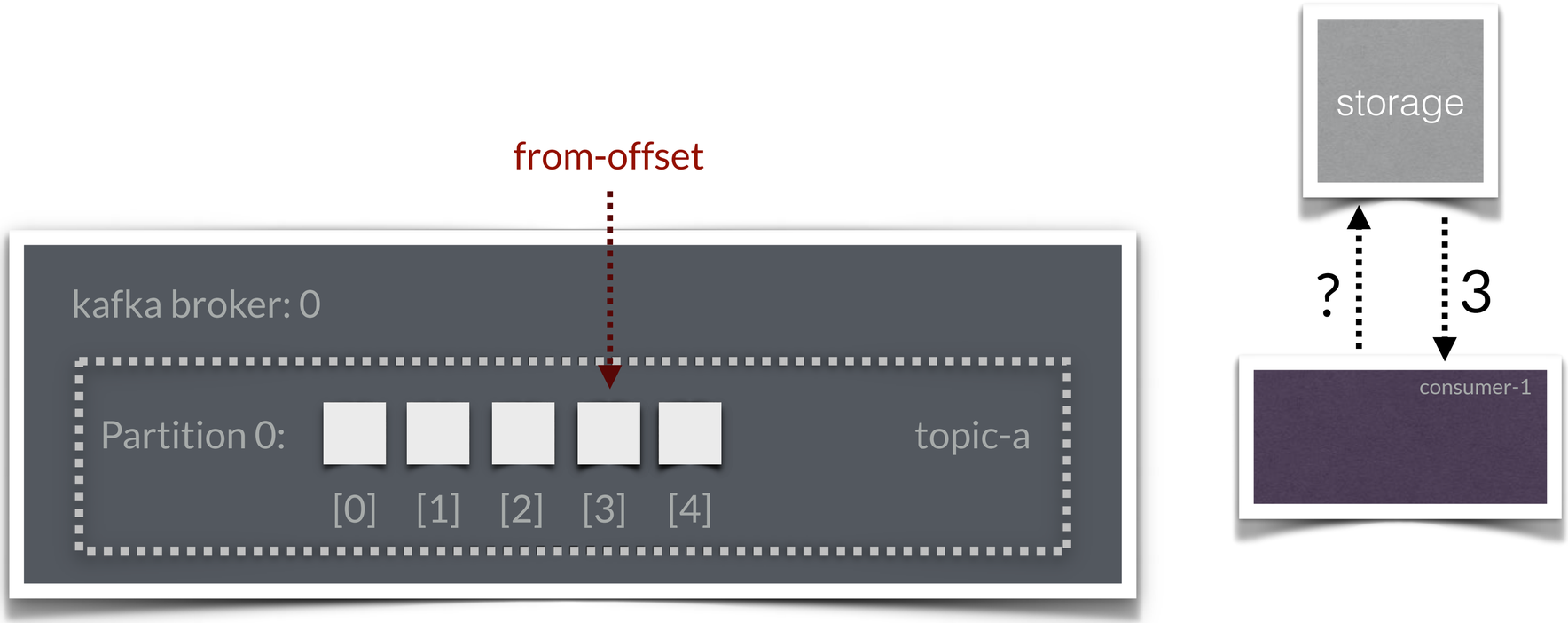
Consumers


consumer-1_offset
kafka broker: 0
Partition 0:
[0]
[1]
[2]
topic-a
consumer-1


consumer-1_offset
kafka broker: 0
Partition 0:
topic-a
[0]
[1]
[2]
[3]
[4]
consumer-1


from-beginning
kafka broker: 0
Partition 0:
topic-a
[0]
[1]
[2]
[3]
[4]
consumer-1

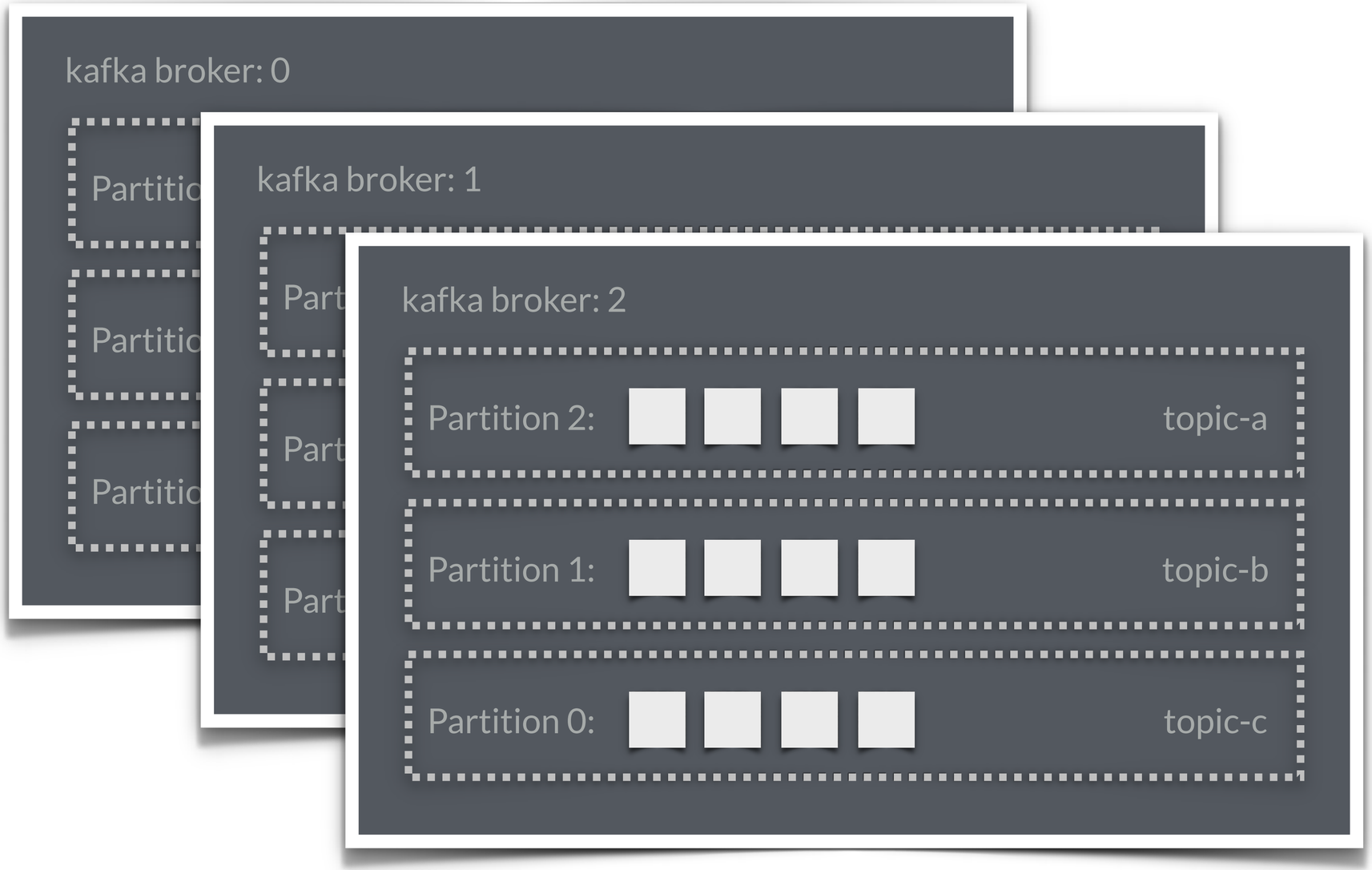

from-end
kafka broker: 0
Partition 0:
topic-a
[0]
[1]
[2]
[3]
[4]
consumer-1


from-offset
kafka broker: 0
Partition 0:
[0]
[1]
[2]
[3]
[4]
topic-a
storage
?
3
consumer-1

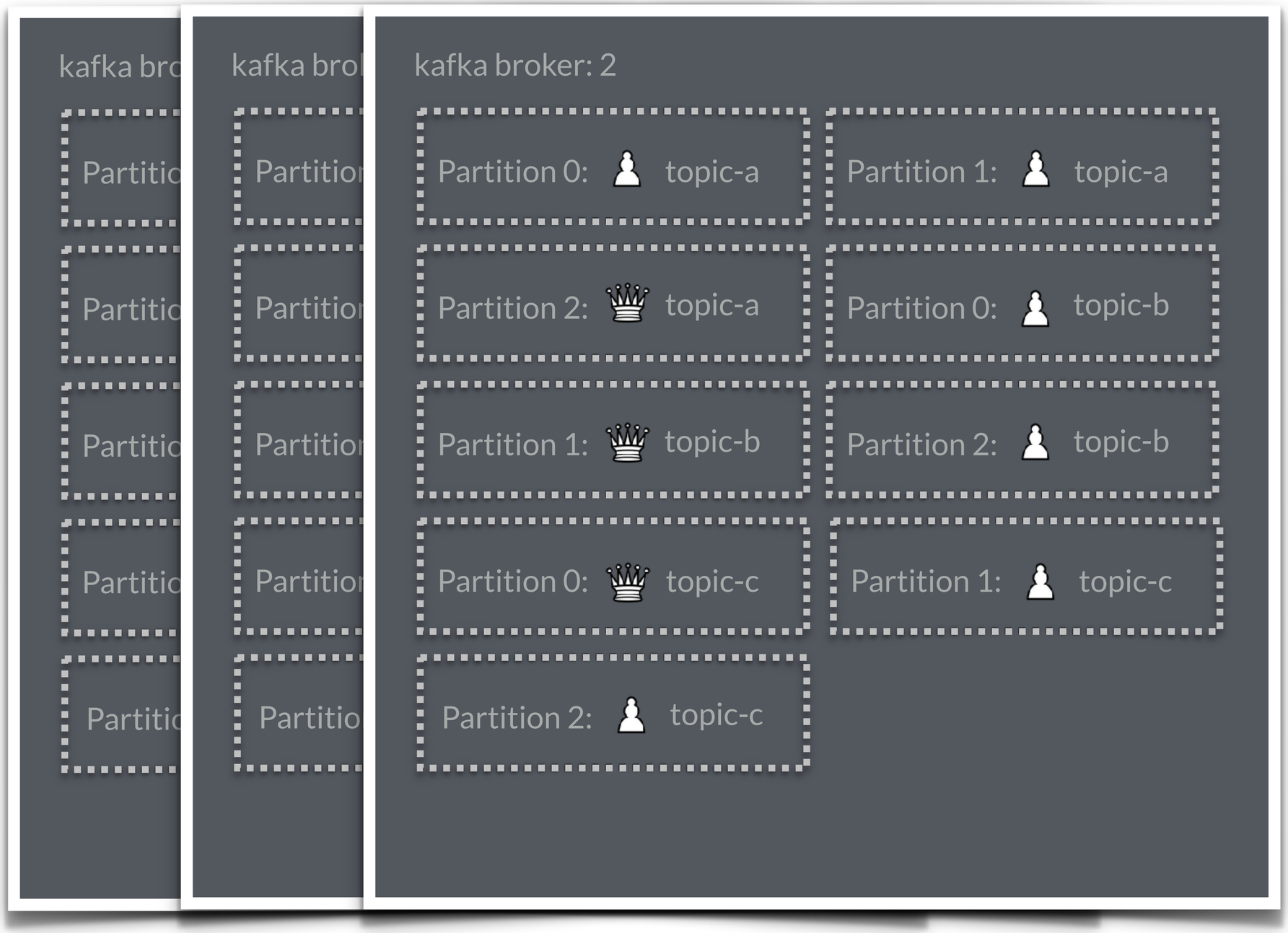
Lab: Writing a Consumer

- Let’s create a consumer in Kafka

Brokers



Each partition is on a different broker,
therefore a single topic is scaled


kafka broker: 2
Partition 0: topic-a
Partition 1: topic-a
Partition 2: topic-a
Partition 0: topic-b
Partition 1: topic-b
Partition 2: topic-b
Partition 0: topic-c
Partition 1: topic-c
Partition 2: topic-c

A-E

F-K

L-S

T-Z

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

The Solution

- Event Sourcing
  - 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?

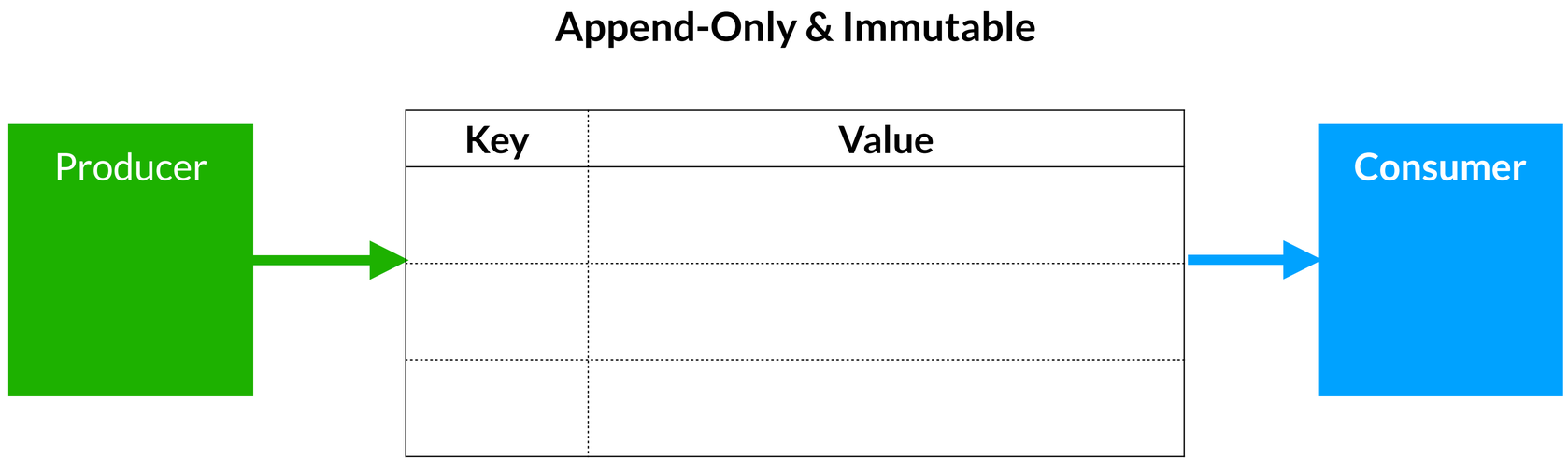
The Ingredients

- Event Store: Central repository for immutable events.
- Event Stream: Ordered sequence of events for each entity.
- Projection: Derived state built from events for read-optimized queries.
- Versioning: Handle changes to event schemas over time.
- Consistency: Requires strong domain modeling to ensure meaningful event data.

The Technology

- **Event Stores:** EventStore, **Kafka**, PostgreSQL with Event Sourcing Extension
- **Frameworks:** Axon Framework (Java), Akka Persistence (Scala/Java), Prooph (PHP).
- **Databases:** DynamoDB for event storage in AWS environments, Cassandra for high scalability.
- **Cloud-Native Options:** AWS EventBridge, Azure Event Grid.

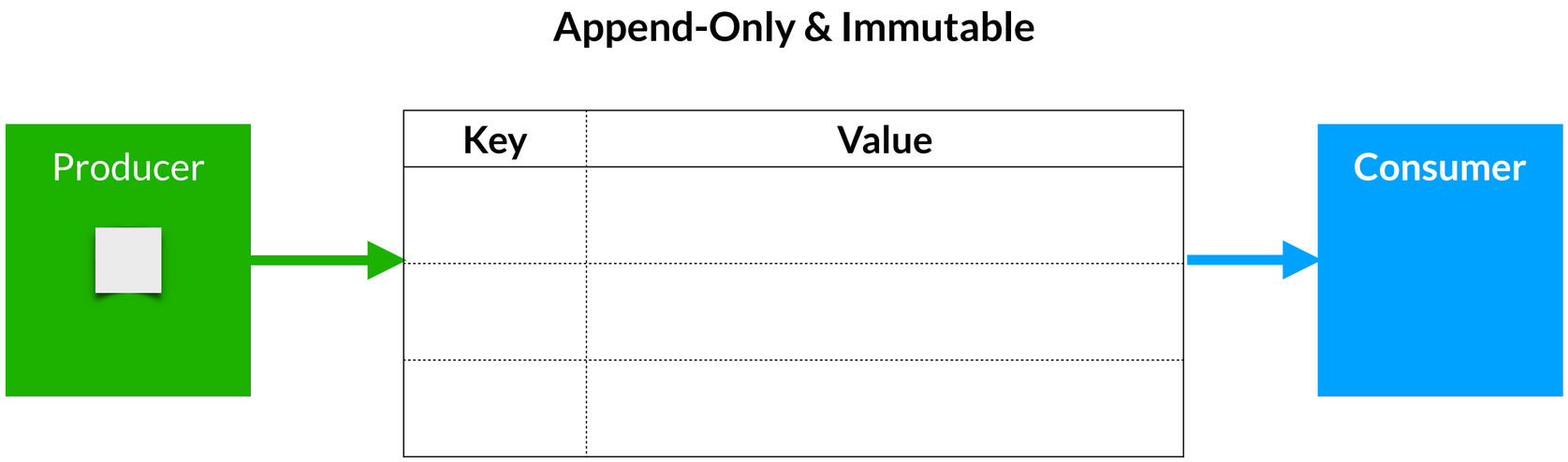
The Diagram



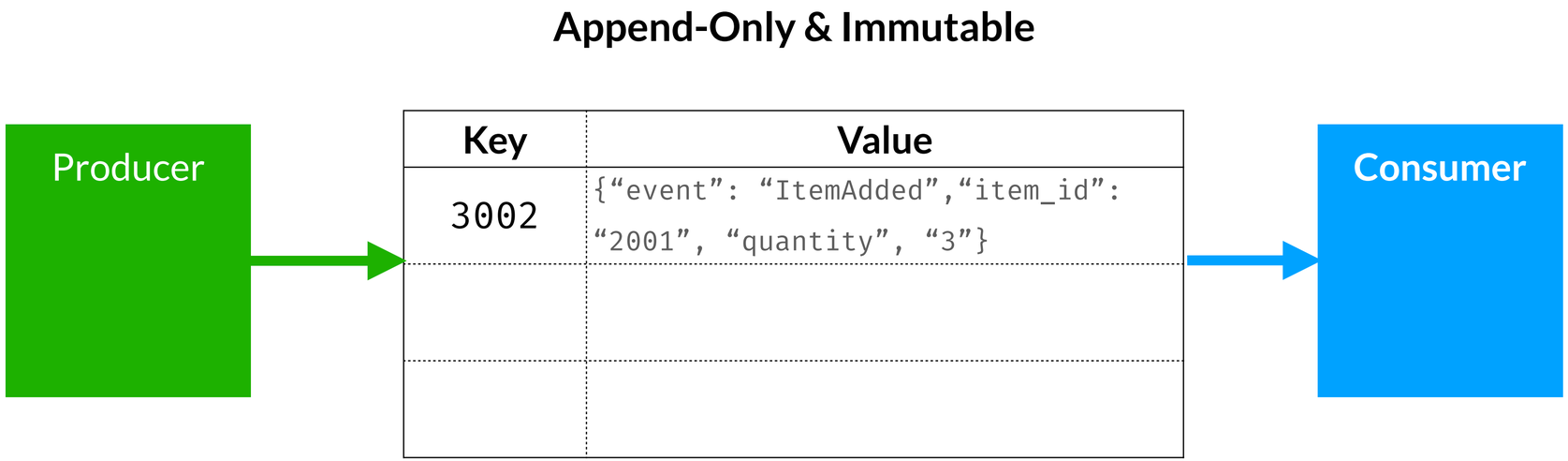

```
3002
```

```
{“event”: “ItemAdded”,
 “item_id”: “2001”
 “quantity”: 3}
```

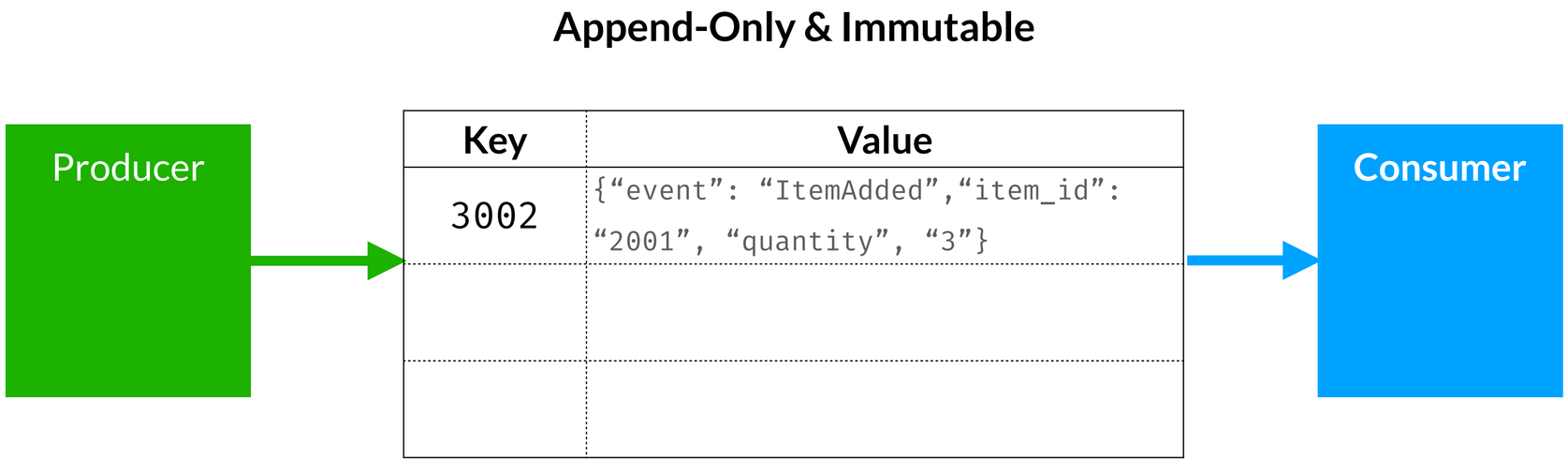
The Diagram



The Diagram


Append-Only & Immutable
Producer
Key
Value
3002
{“event”: “ItemAdded”,“item_id”: “2001”, “quantity”, “3”}
Consumer

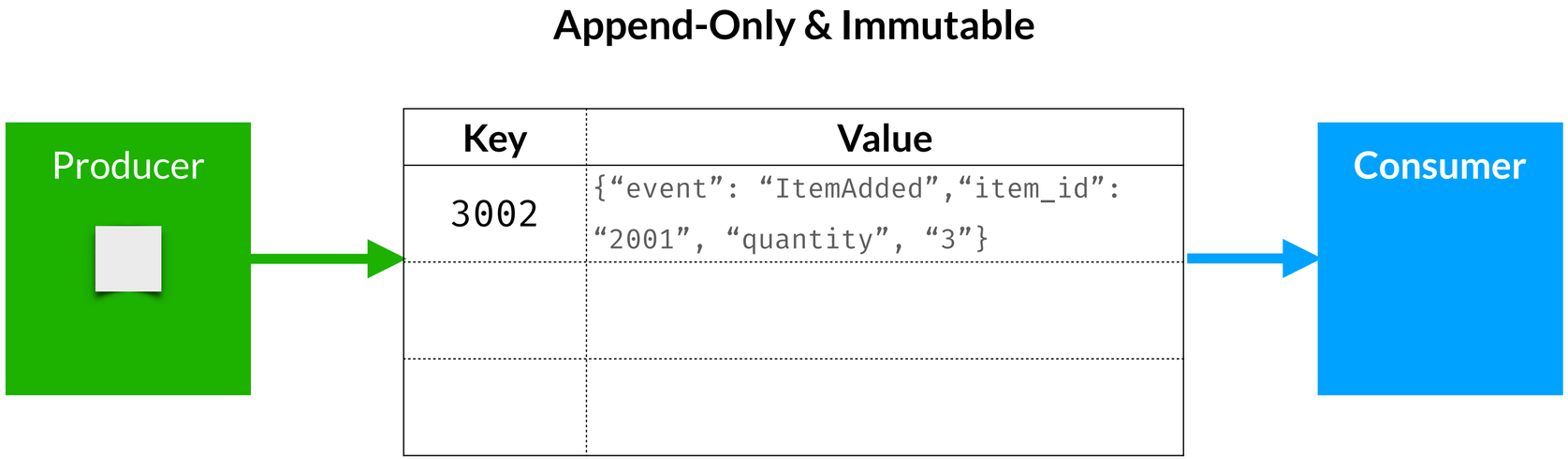
The Diagram



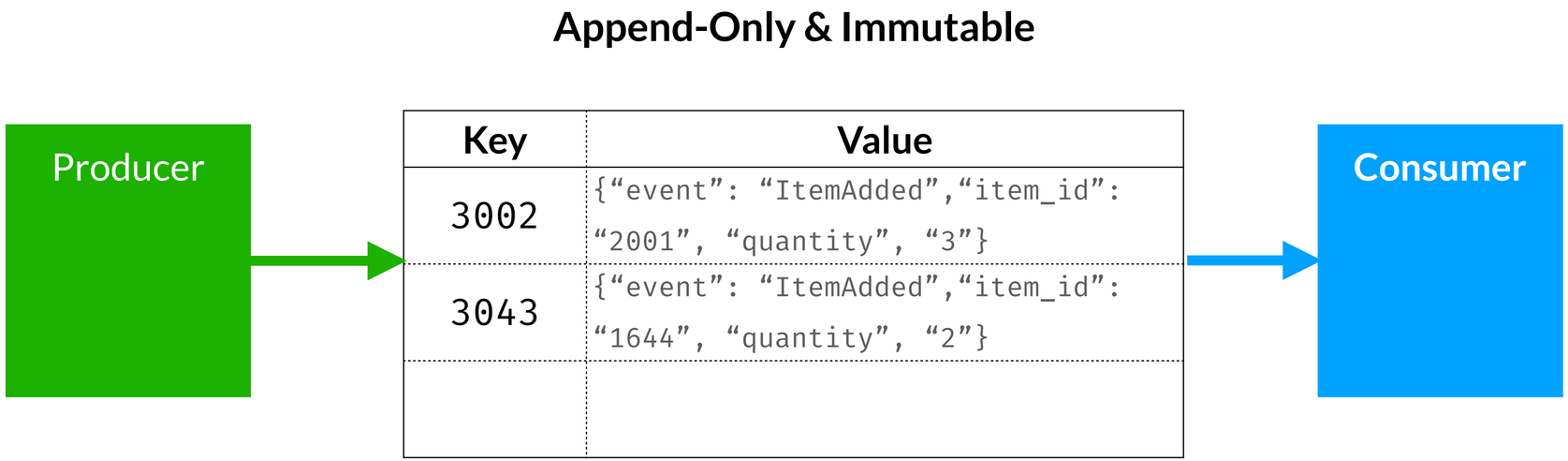

```
3043
```

```
{“event”: “ItemAdded”,
 “item_id”: “1644”
 “quantity”: 2}
```

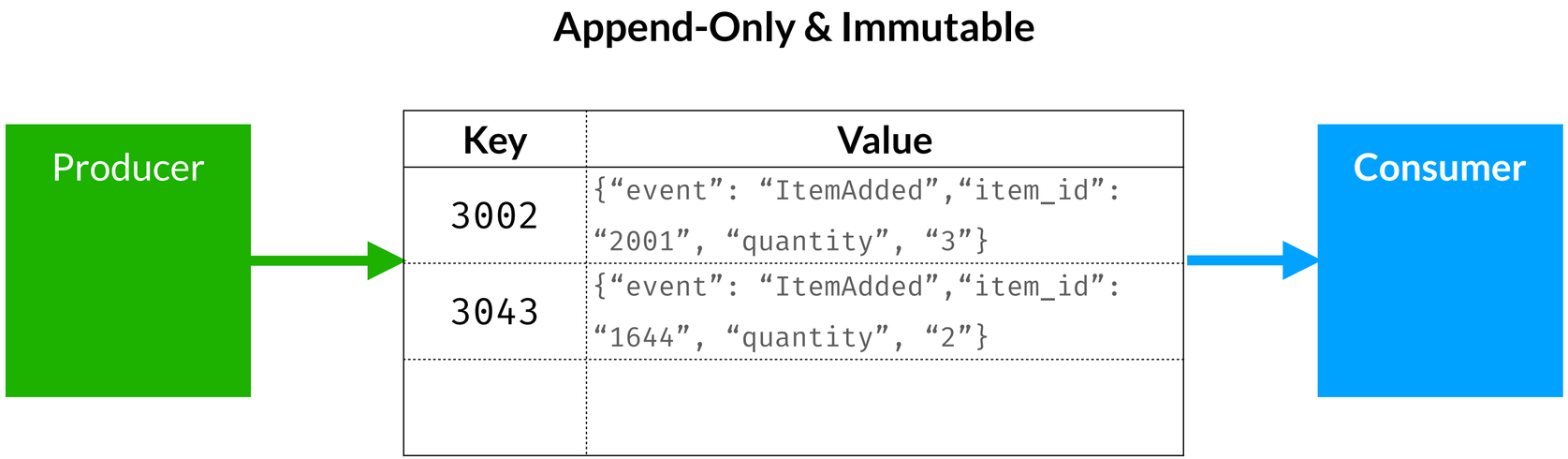
The Diagram


Append-Only & Immutable
Producer
Key
Value
3002
{“event”: “ItemAdded”,“item_id”: “2001”, “quantity”, “3”}
Consumer

The Diagram


Append-Only & Immutable
Producer
Key
Value
3002
{“event”: “ItemAdded”,“item_id”: “2001”, “quantity”, “3”}
3043
{“event”: “ItemAdded”,“item_id”: “1644”, “quantity”, “2”}
Consumer

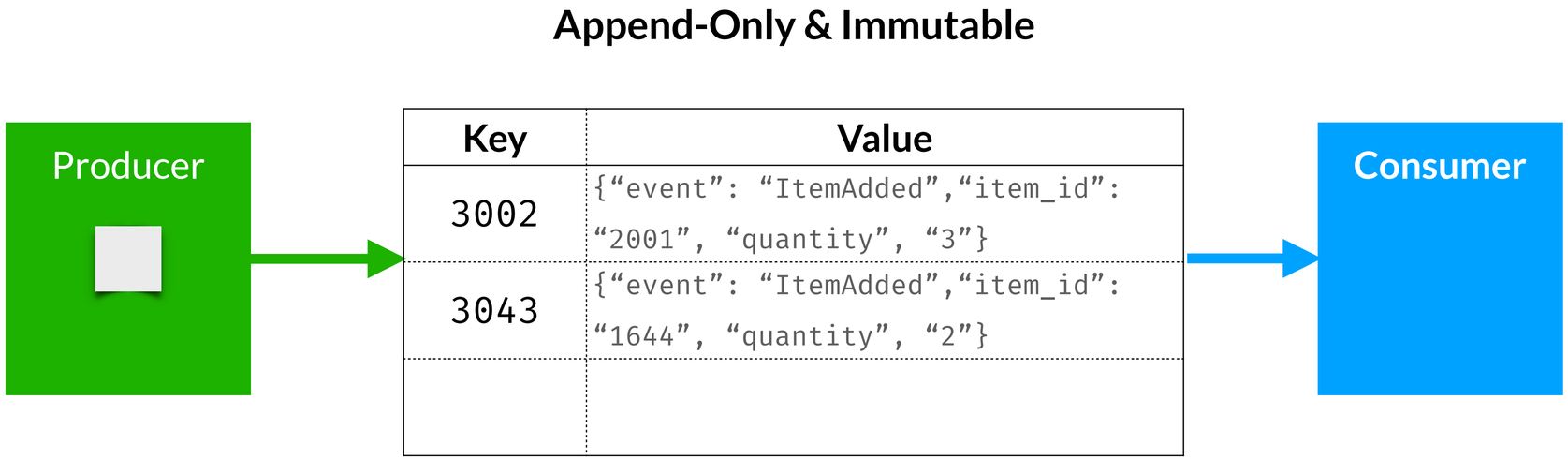
The Diagram



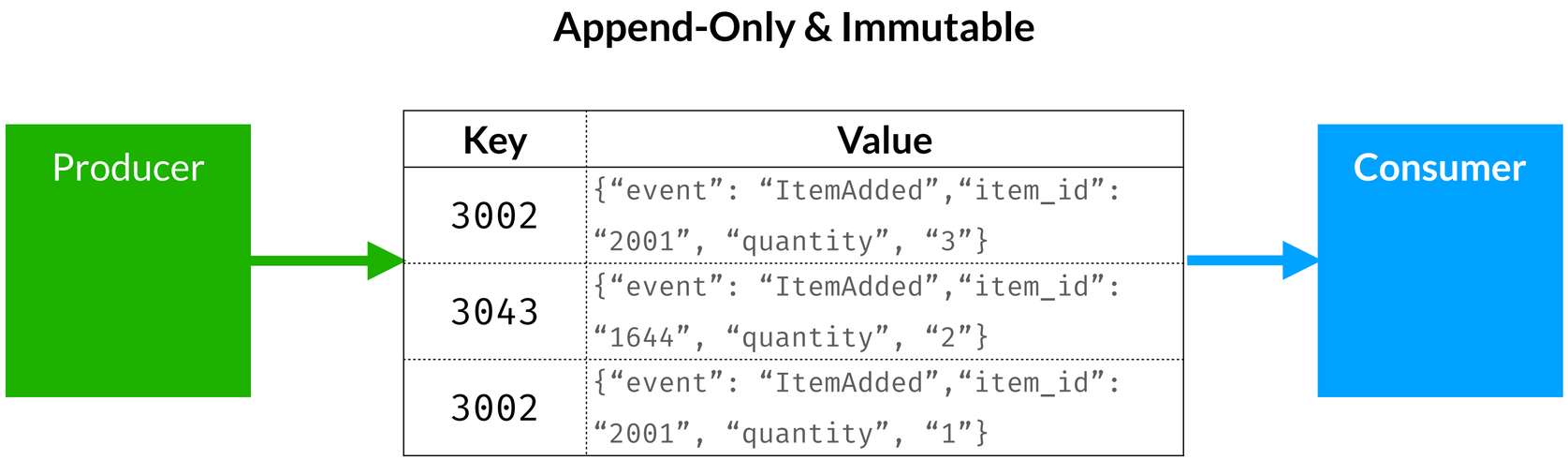

```
3002
```

```
{“event”: “ItemAdded”,
 “item_id”: “2001”
 “quantity”: 1}
```

The Diagram


Append-Only & Immutable
Producer
Key
Value
3002
{“event”: “ItemAdded”,“item_id”: “2001”, “quantity”, “3”}
3043
{“event”: “ItemAdded”,“item_id”: “1644”, “quantity”, “2”}
Consumer

The Diagram


Append-Only & Immutable
Producer
Key
Value
3002
{“event”: “ItemAdded”,“item_id”: “2001”, “quantity”, “3”}
3043
{“event”: “ItemAdded”,“item_id”: “1644”, “quantity”, “2”}
3002
{“event”: “ItemAdded”,“item_id”: “2001”, “quantity”, “1”}
Consumer

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

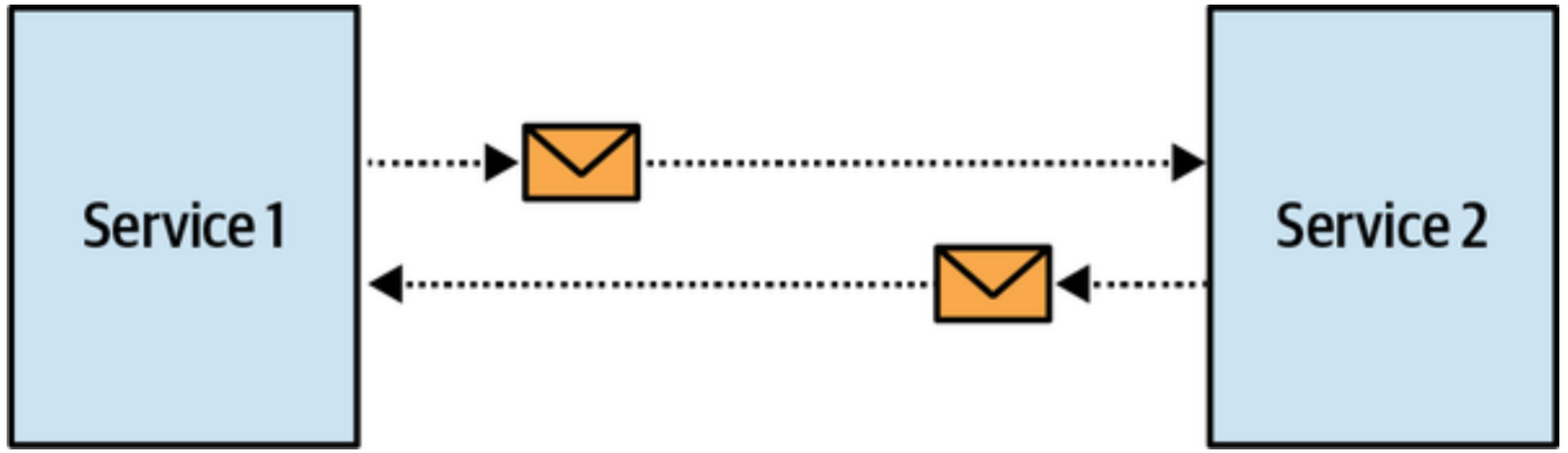
Lab: Go Back in Time

- We should be able to rewind in time to some

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


Service 1
Service 2

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
    }
}
```

Event Notification

Event Notification

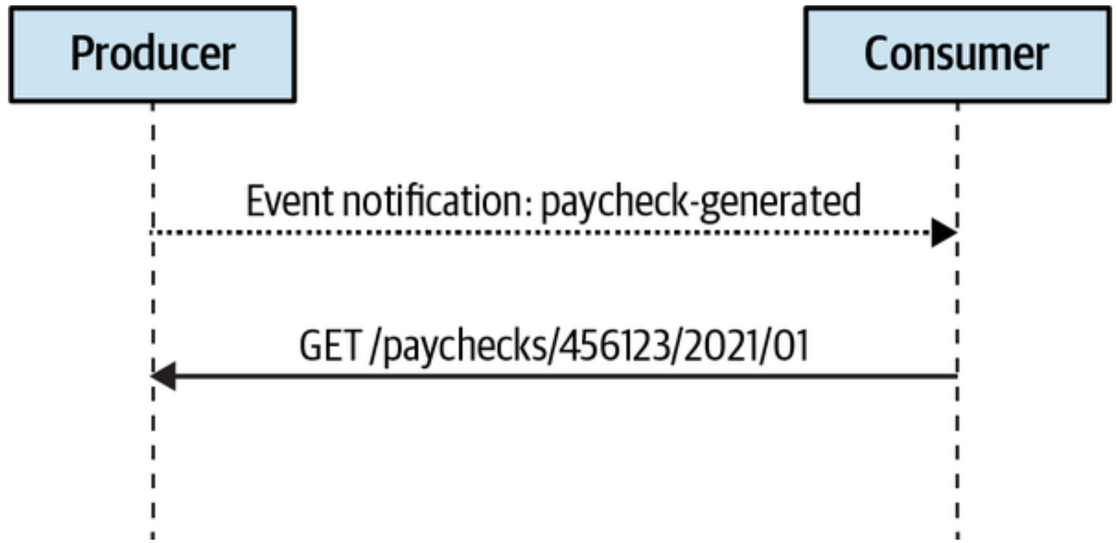
- Message regarding a change in the business domain that other components will react to
- Examples include:
  - `PaycheckGenerated`
  - `CampaignPublished`
- Event notification should not be verbose
- The goal is to notify the interested parties about the event and shouldn’t carry all the information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

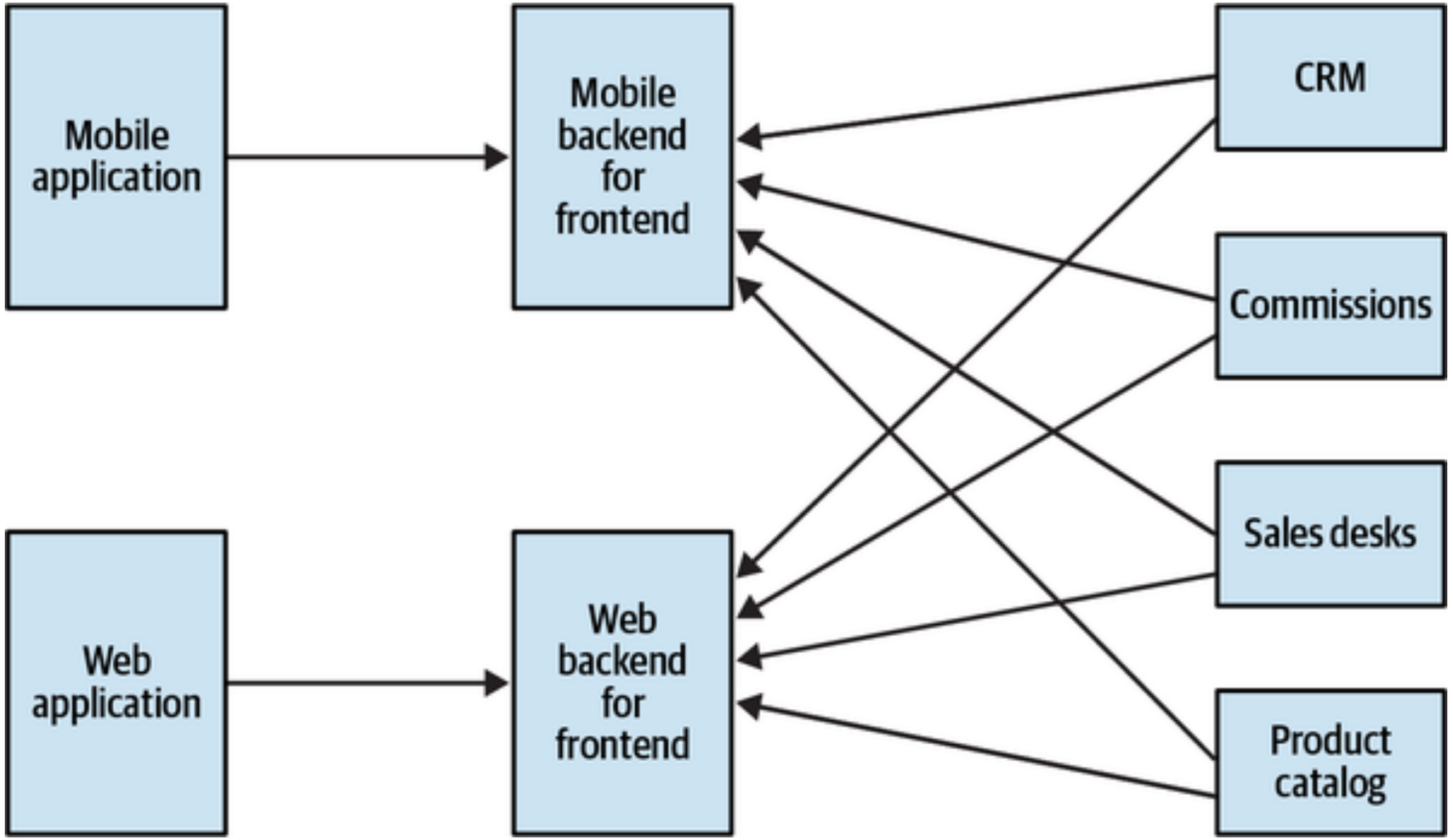
Another form is only carrying the data that has modified

```
{
    "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

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

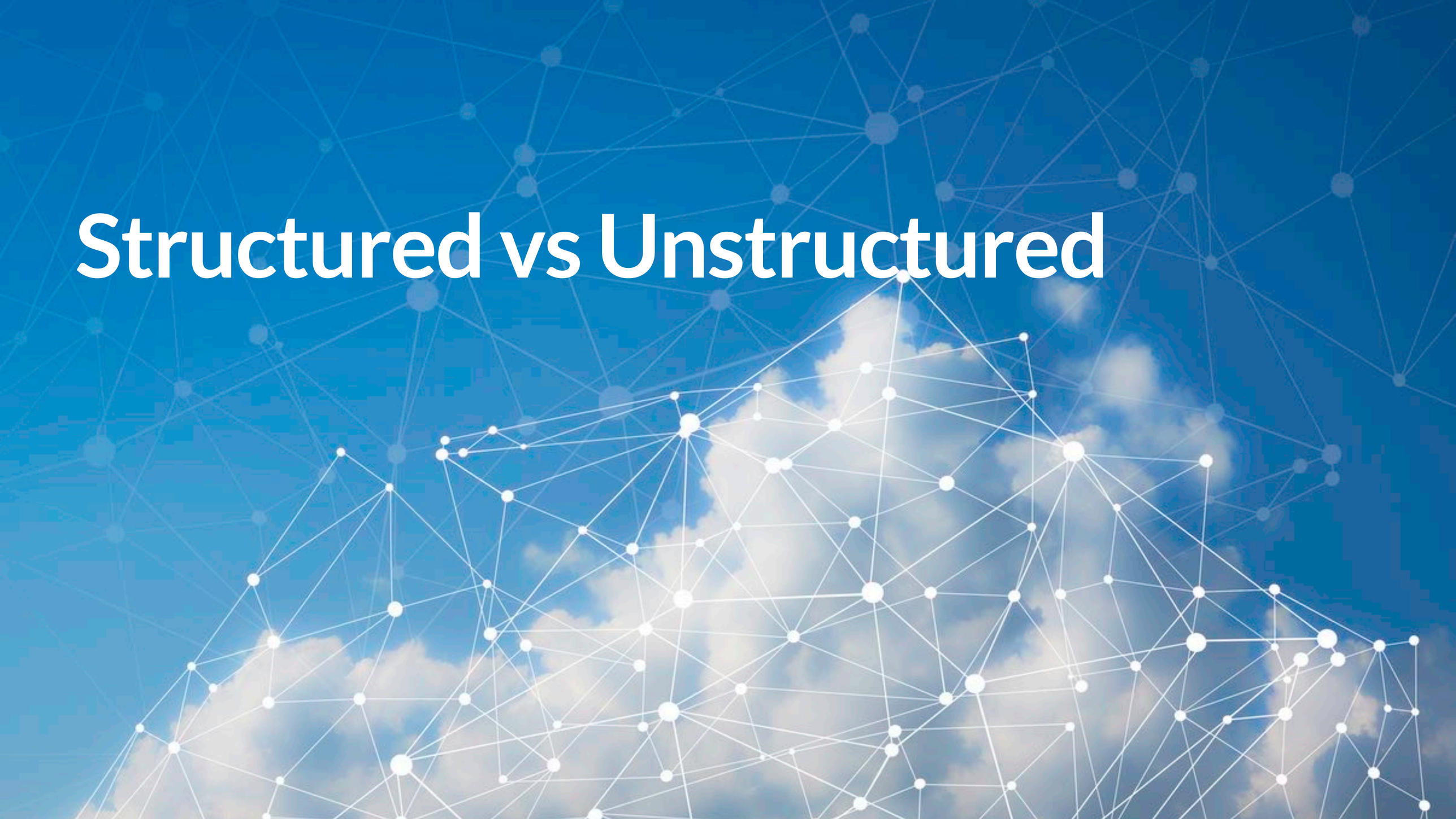
```
ecst = {
    "type": "personal-details-changed",
    "person-id": "01b9a761",
    "payload": {
        "new-last-name": "Williams"
    }
};
```

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
    }
};
```

Structured vs Unstructured



Structured vs Unstructured Data

- **Structured:**
  - Data that conforms to a predefined schema or structure, making it easily searchable and queryable.
  - The schema defines the types, relationships, and constraints of the data.
  - Can exist in non-tabular formats like JSON, XML, or Avro if a schema is defined.
- **Unstructured Data:**
  - Data that lacks a predefined schema or consistent organization.
  - Examples: Text files, images, videos, audio files, and logs without consistent structure.

Structured vs Unstructured Data

Unstructured Logging

```
6:01:00 accepted connection on port 80 from 10.0.0.3:63349
6:01:03 basic authentication accepted for user foo
6:01:15 processing request for /super/slow/server
6:01:18 request succeeded, sent response code 200
6:01:19 closed connection to 10.0.0.3:63349
```

Structured Logging

```
time="6:01:00" msg="accepted connection" port="80" authority="10.0.0.3:63349"
time="6:01:03" msg="basic authentication accepted" user="foo"
time="6:01:15" msg="processing request" path="/super/slow/server"
time="6:01:18" msg="sent response code" status="200"
time="6:01:19" msg="closed connection" authority="10.0.0.3:63349"
```

Structured vs Unstructured Data

Structured vs. Unstructured

Aspect	Structured Data	Unstructured Data
Storage	SQL/NoSQL databases (e.g., MySQL, JSON, Parquet)	Object stores or file systems (e.g., S3, HDFS)
Querying	SQL or tools like Presto/Spark with schema	Advanced NLP, computer vision, or ML
Format	Tabular, JSON, XML with schema	Free-form (e.g., raw text, videos, images)
Schema Dependency	Schema-defined before or during use	No schema or schema inferred dynamically
Cost to Store	Higher per unit due to indexing, schema enforcement, and redundancy requirements	Lower per unit because raw data requires minimal processing or indexing

Schema Development

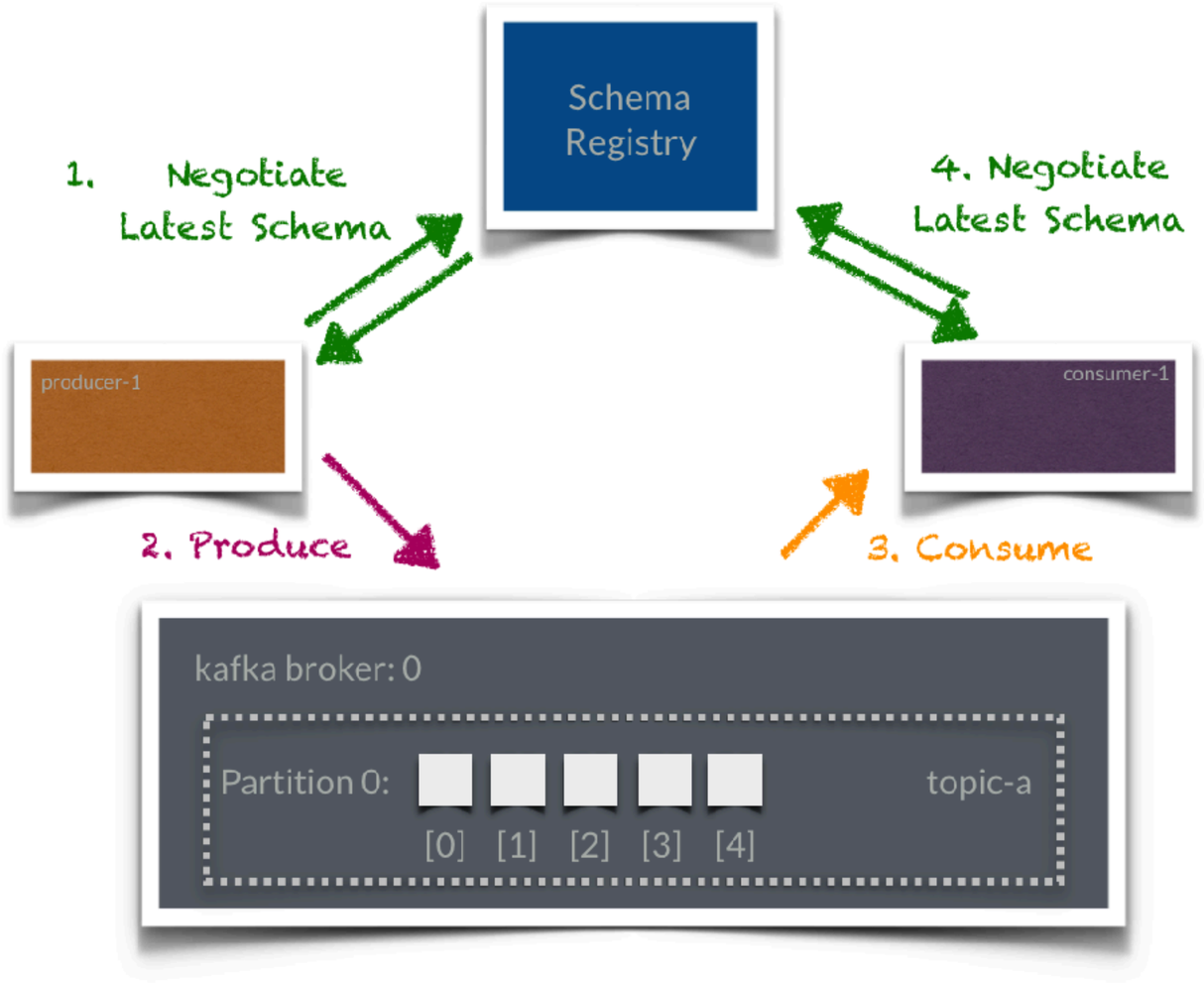
Schema Deployment

- Define the structure and format of data in events.
- Act as a contract between event producers and consumers.
- Ensure data integrity and understandability across systems.

Schema Technologies

- Avro: Compact, fast, and widely used in Kafka and other event systems.
- Protocol Buffers (Protobuf): Efficient serialization with strong type safety.
- JSON Schema: Human-readable, widely adopted for JSON payloads.
- OpenAPI: Defines RESTful services with JSON payload specifications.
- Apache Thrift: Cross-language schema and RPC support.
- Confluent Schema Registry: Manages Avro, Protobuf, or JSON schemas in Kafka ecosystems.

Schema Registry



Backward vs Forward Compatibility

Backward vs Forward Compatibility

- **Backward Compatibility:**
  - Ensures that new versions of a system or API can handle data or requests from older versions.
  - **Key Goal:** Allow older clients to work seamlessly with newer versions.
  - **Example:** A new database schema still supports queries written for the old schema.
- **Forward Compatibility:**
  - Ensures that older versions of a system or API can handle data or requests from newer versions, typically by ignoring unknown fields or extensions.
  - **Key Goal:** Allow older systems to understand or tolerate future changes.
  - **Example:** A legacy system processes JSON with extra fields added by a new client.
- **Both Compatibility:**
  - A system that maintains both backward and forward compatibility, enabling robust interoperability between versions in both directions.

Backward vs Forward Compatibility

1980s	1990s	Today
• company name • contact person • physical address • phone • fax number	• company name • contact person • physical address • phone • fax number • **email address**	• company name • contact person • physical address • phone • ~~fax number~~ • email address • **social media**

Transitive Compatibility

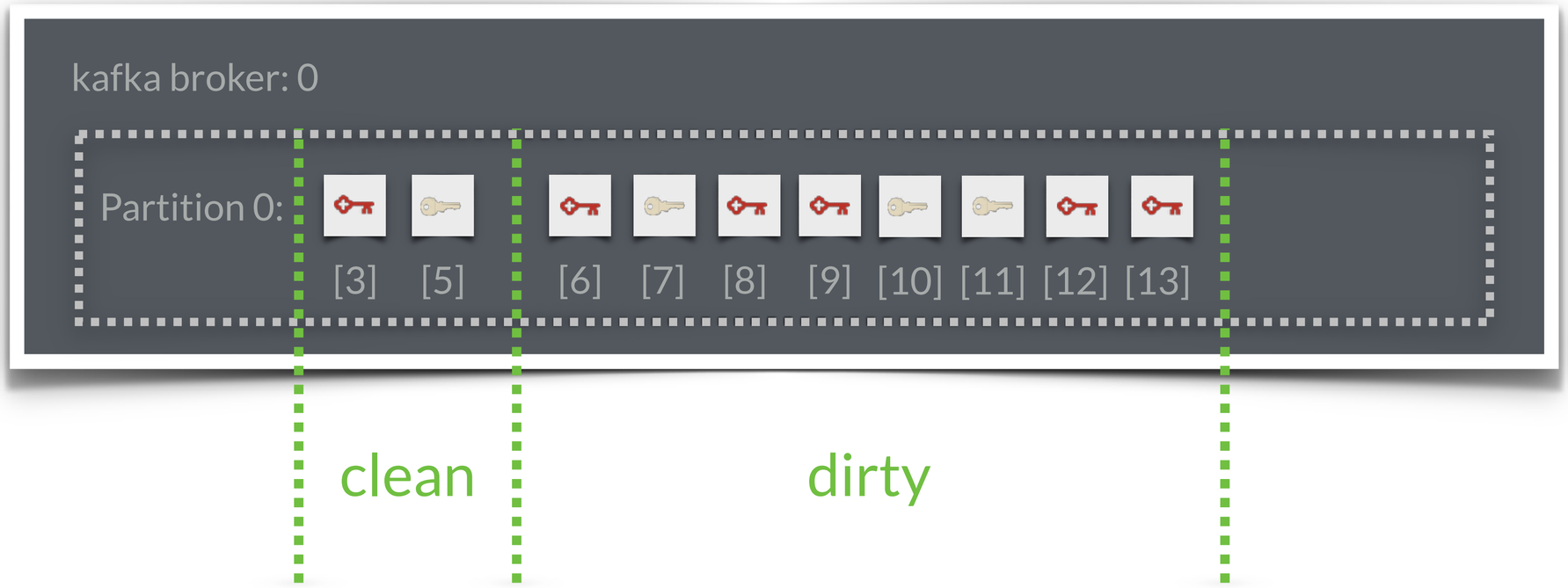
- **Transitive compatibility** ensures that multiple versions of a system can interoperate through a chain of compatibility.
- If version **N** is compatible with **N-1**, and **N+1** is compatible with **N**, then **N+1** is transitively compatible with **N-1**.
- **Forward Transitive:** Older systems or APIs (e.g., **N-1**) can interact with multiple newer versions (e.g., **N** and **N+1**) by ignoring unknown fields or gracefully handling new functionality.
- **Backward Transitive:** A newer system or API (e.g., **N+1**) can interact with multiple older versions (e.g., **N** and **N-1**) by preserving legacy functionality.

Compaction

Compaction

- A form of retention where only messages of the latest key in a partition will be retained
- Compaction is performed by a *cleaner thread*


kafka broker: 0
Partition 0:
[0]
[1]
[2]
[3]
[4]
[5]
kafka broker: 0
Partition 0:
[3]
[5]



Kafka will start compacting when 50% of the topic contains dirty records

Streaming


TK45
115m
ÚNIK.C.

Stream Processing

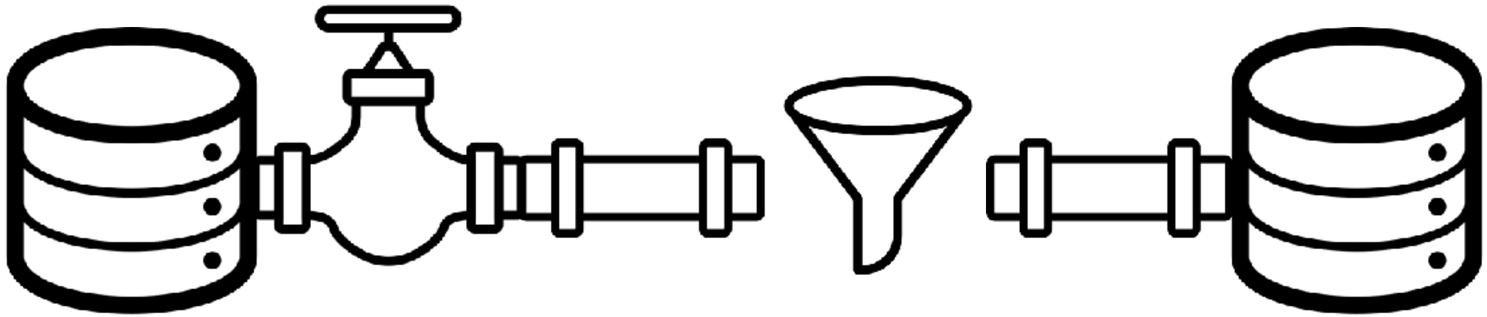
- Endless supply of data
- “Replayable"
- Real Time Processing for Fraud Detection, High End Sales Detection, Internet of Things, and More.
- Will require built-in accumulator table to perform real time data processing, like counting, and grouping


NY
100.00

Filtering

NY

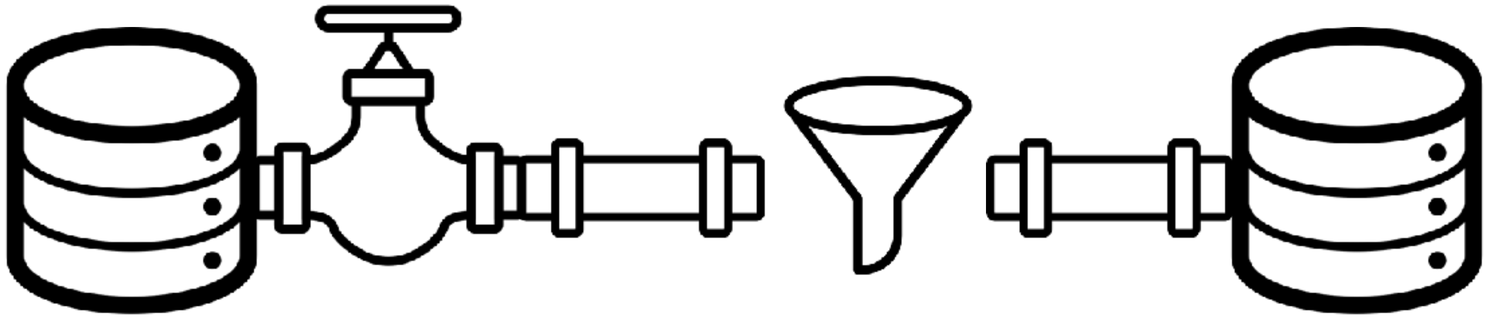
100.00



```
value > 10000
```

OH

20000

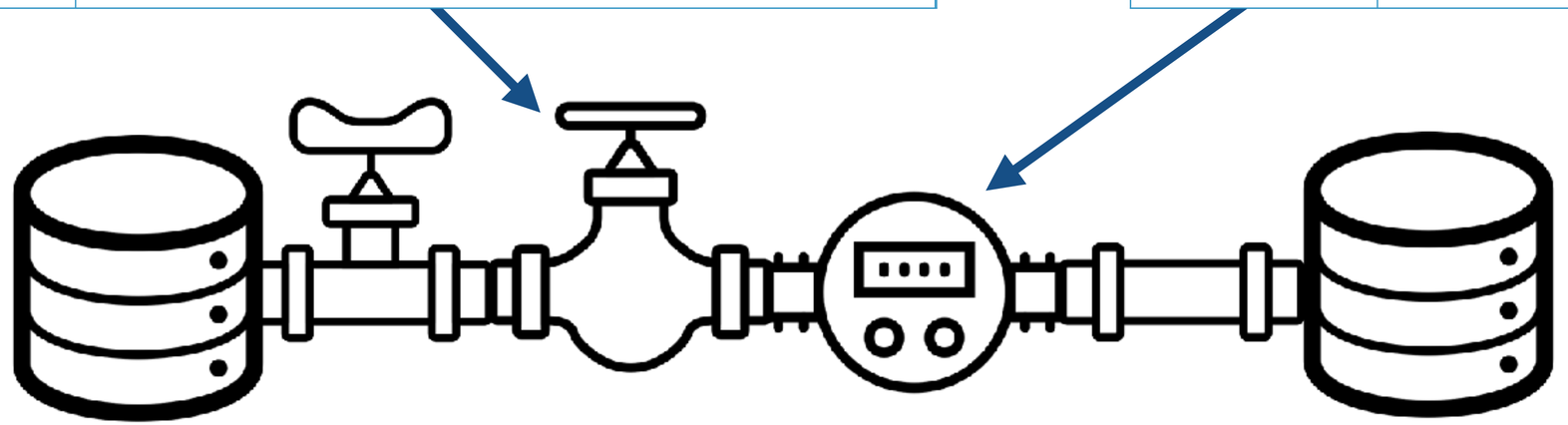


value > 10000

Aggregating

Key	Value
OH	100.0

Key	Value
OH	100.0



OH

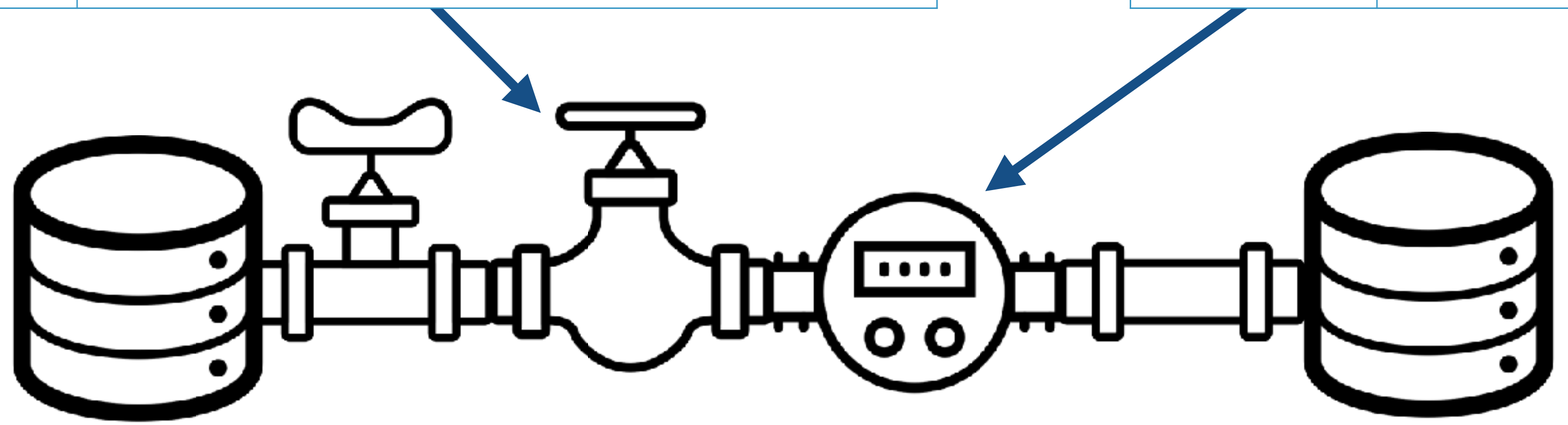
100.00

OH

100.00

Key	Value
OH	100.0
NY	20.0

Key	Value
OH	100.0
NY	20.0



NY

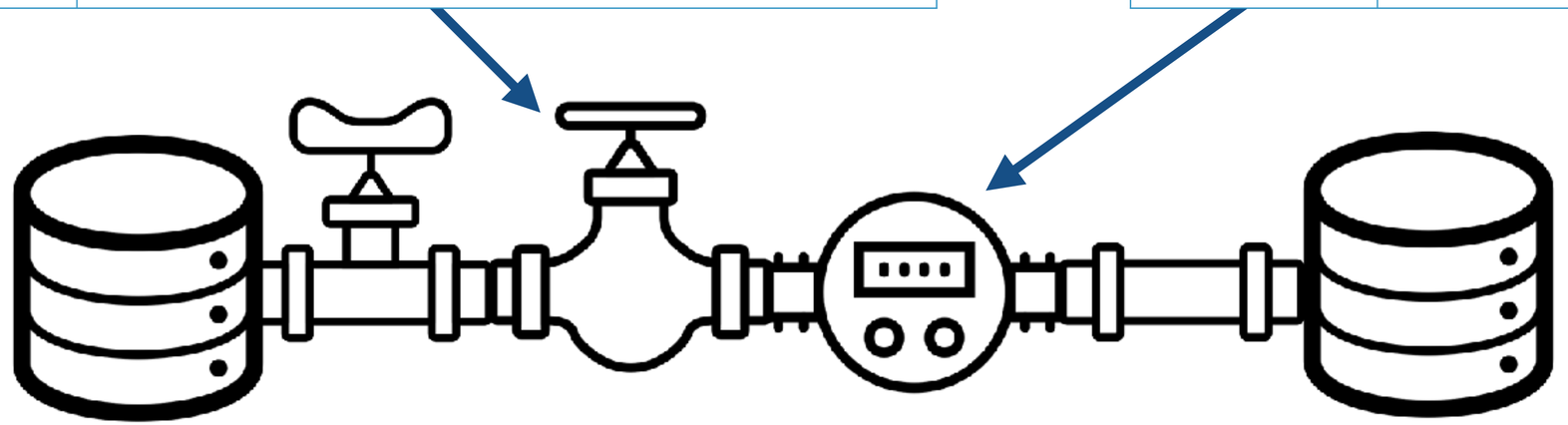
20

NY

20

Key	Value
OH	100.0
NY	20.0, 60.0

Key	Value
OH	100.0
NY	80.0

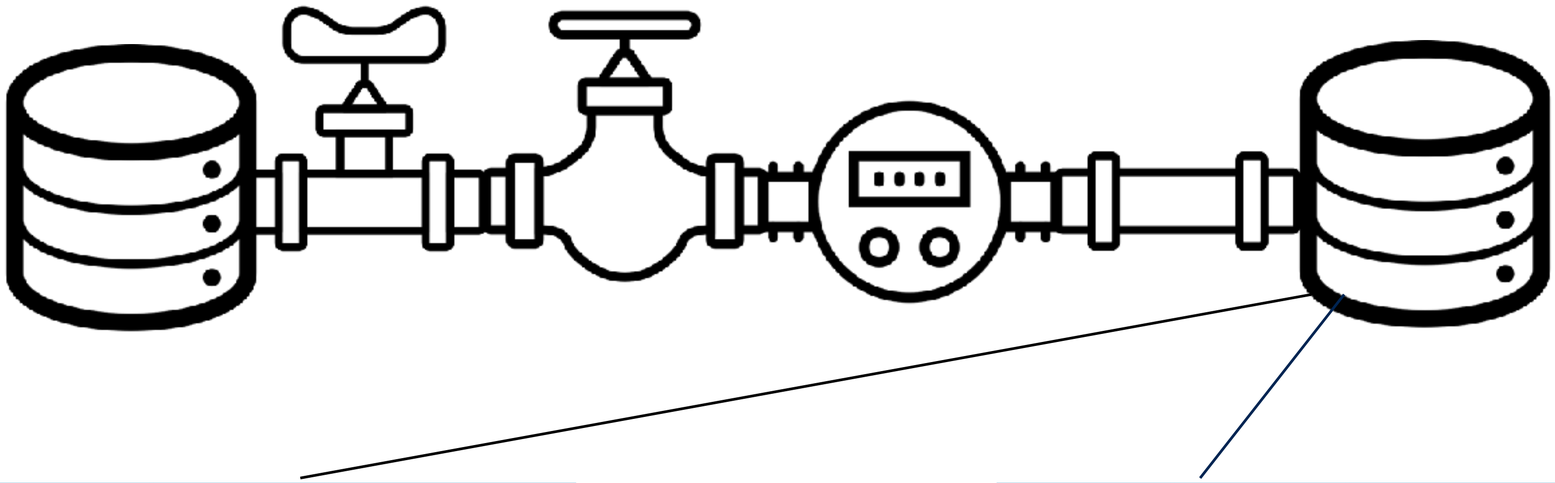


NY

60

NY

80.0

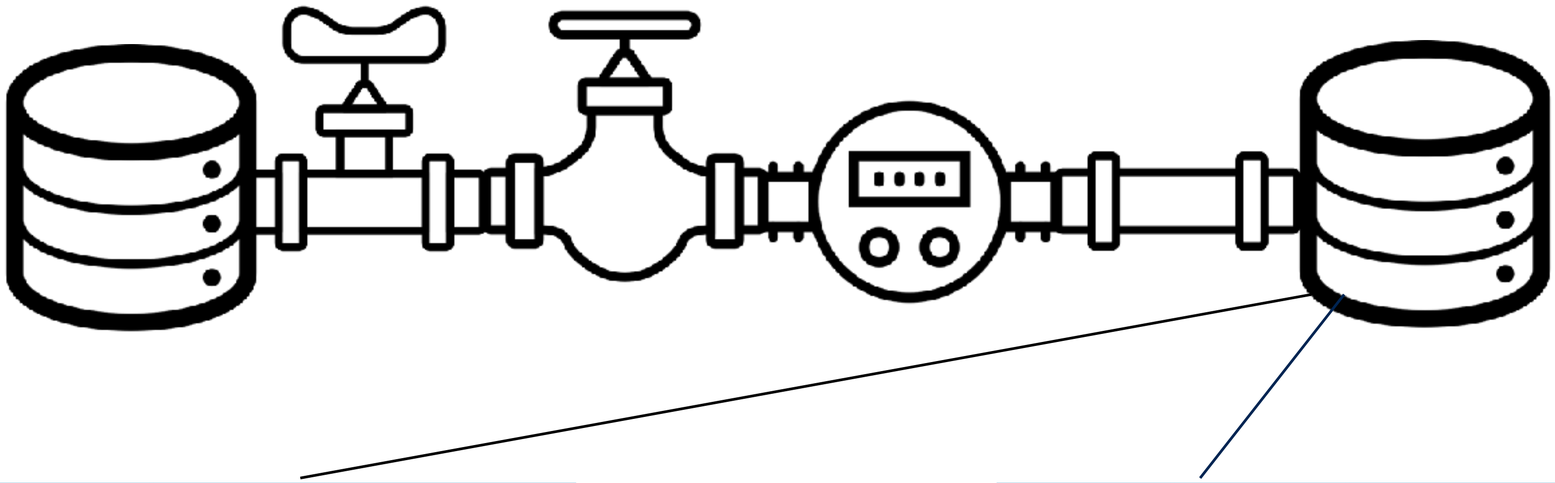


Offset	Key	Value
0	**OH**	100.0

Partition 0

Offset	Key	Value
0	**NY**	20.0
1	**NY**	80.0

Partition 1



Offset	Key	Value
0	**OH**	100.0

Partition 0

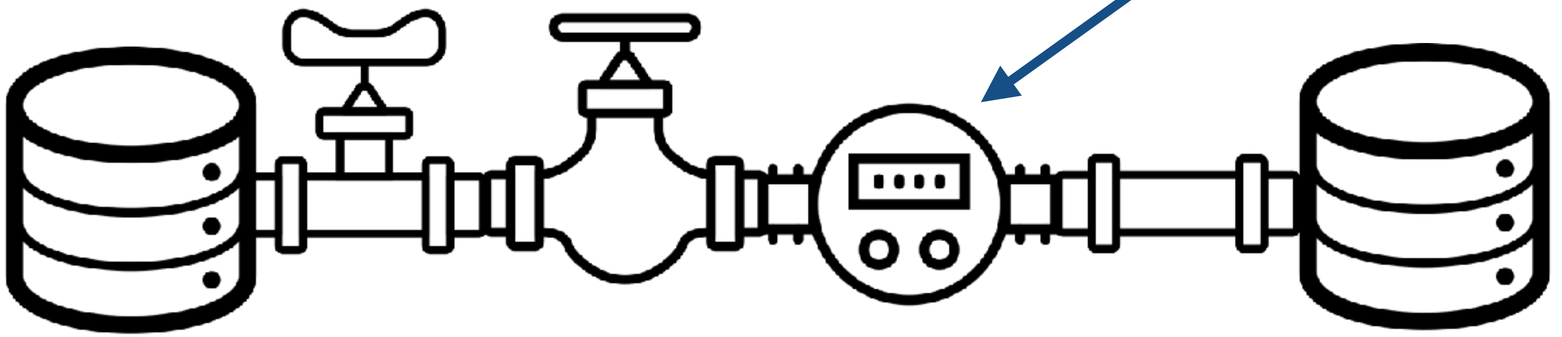
Offset	Key	Value
1	**NY**	80.0

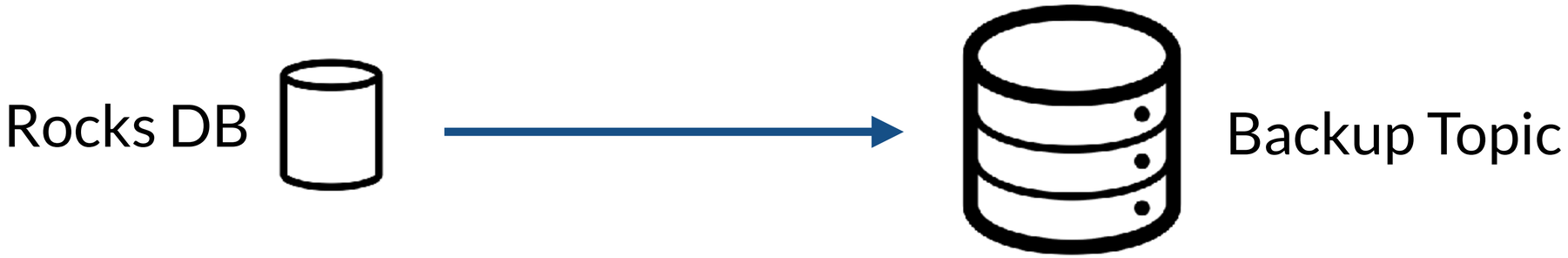
Partition 1

After Compaction

Rocks DB

Key	Value
OH	100.0
NY	80.0




Rocks DB
Backup Topic

Windowing

Tumbling Window

Hopping Window

Session Window

Session Window

the last element has a occurred; now we wait

Session Window

waited 5 minutes = make it a window

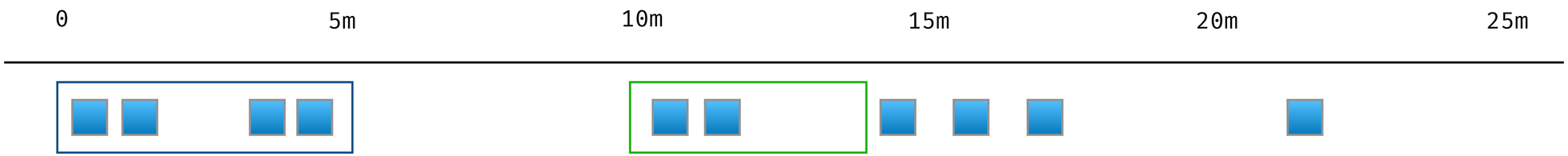
Session Window

we have a new element; we start a new window

Session Window

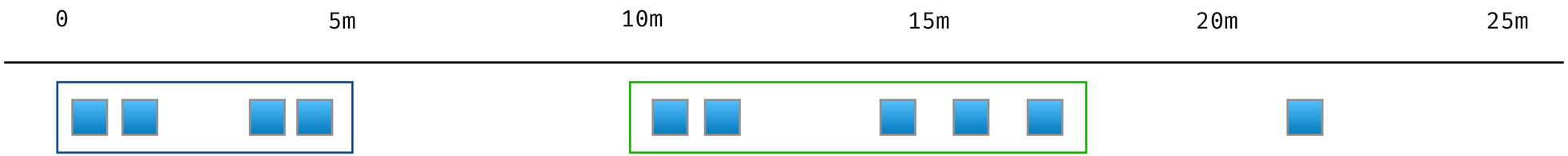
we wait for 5 minutes

Session Window



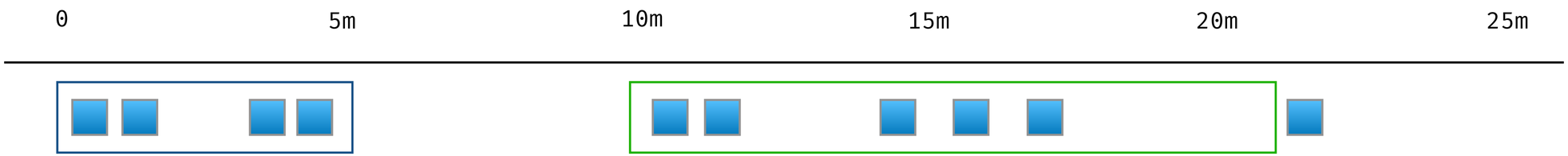
2 minutes elapsed; no windowing yet

Session Window



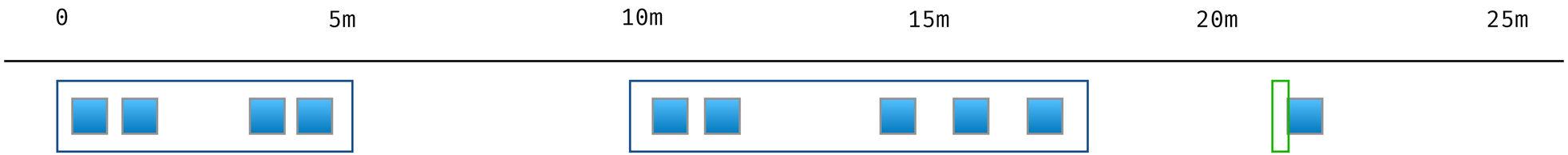
we wait five minutes

Session Window



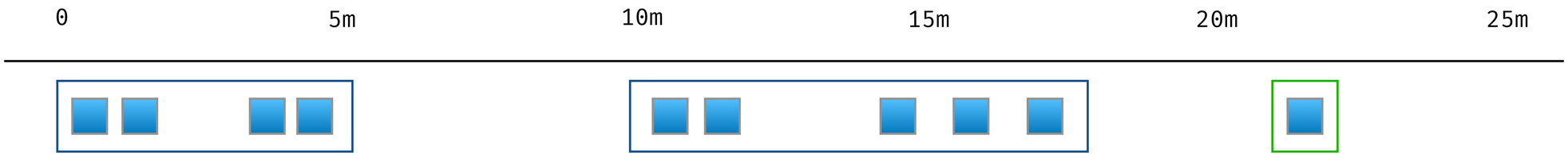
no new elements; we make it a window

Session Window



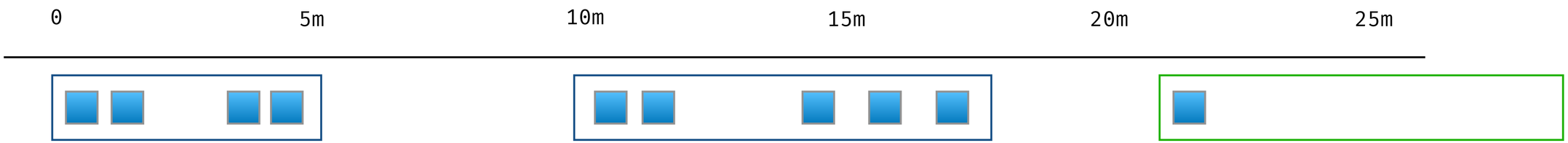
we see something new

Session Window



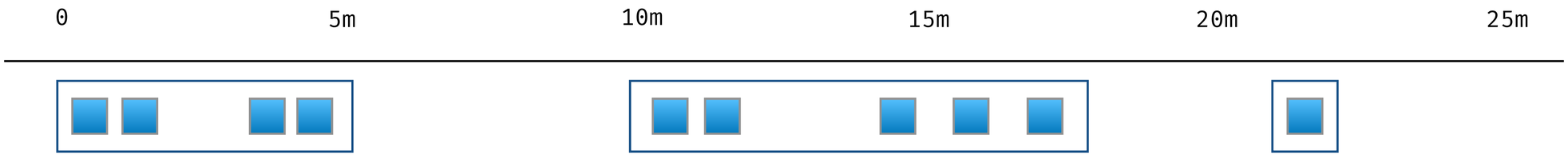
last element; we wait five minutes

Session Window



nothing else; we wait 5 minutes

Session Window



that's a new window

Lab: Using Avro and KSQLDB

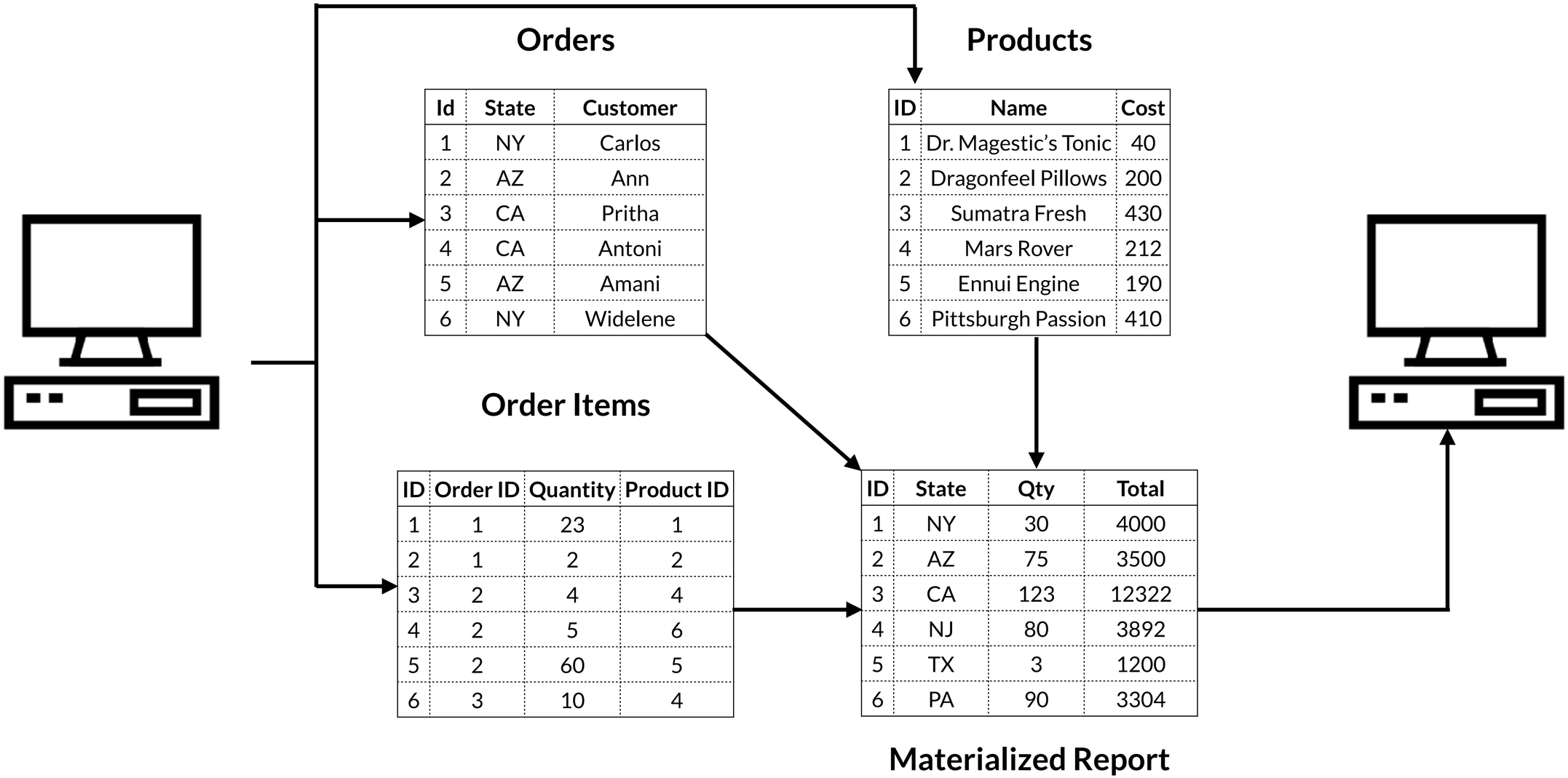
- Let’s do streaming but with well-structured data and SQL based streaming

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 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

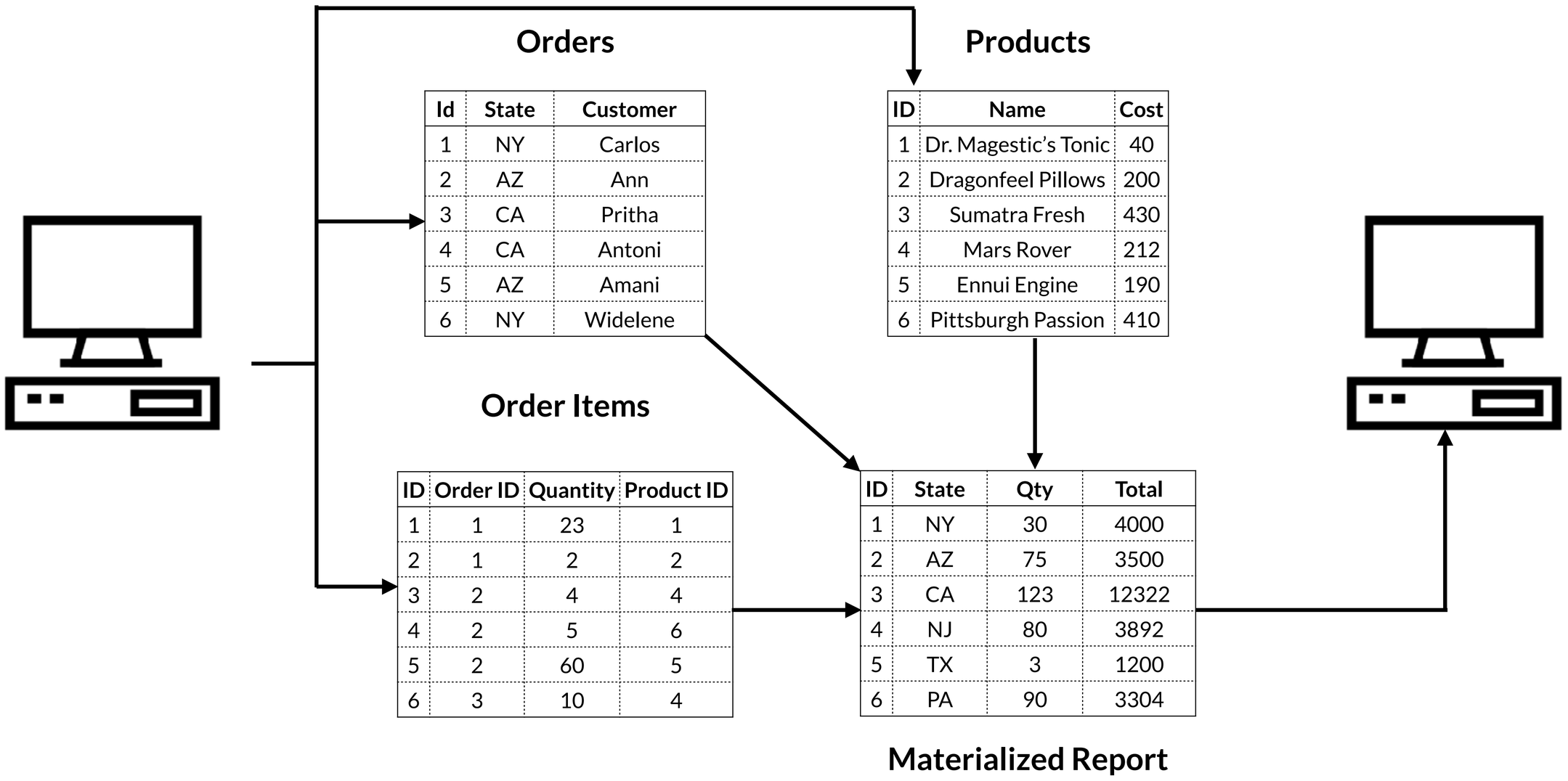


Id	State	Customer
1	NY	Carlos
2	AZ	Ann
3	CA	Pritha
4	CA	Antoni
5	AZ	Amani
6	NY	Widelene

ID	Name	Cost
1	Dr. Magestic's Tonic	40
2	Dragonfeel Pillows	200
3	Sumatra Fresh	430
4	Mars Rover	212
5	Ennui Engine	190
6	Pittsburgh Passion	410

ID	Order ID	Quantity	Product ID
1	1	23	1
2	1	2	2
3	2	4	4
4	2	5	6
5	2	60	5
6	3	10	4

ID	State	Qty	Total
1	NY	30	4000
2	AZ	75	3500
3	CA	123	12322
4	NJ	80	3892
5	TX	3	1200
6	PA	90	3304



Id	State	Customer
1	NY	Carlos
2	AZ	Ann
3	CA	Pritha
4	CA	Antoni
5	AZ	Amani
6	NY	Widelene

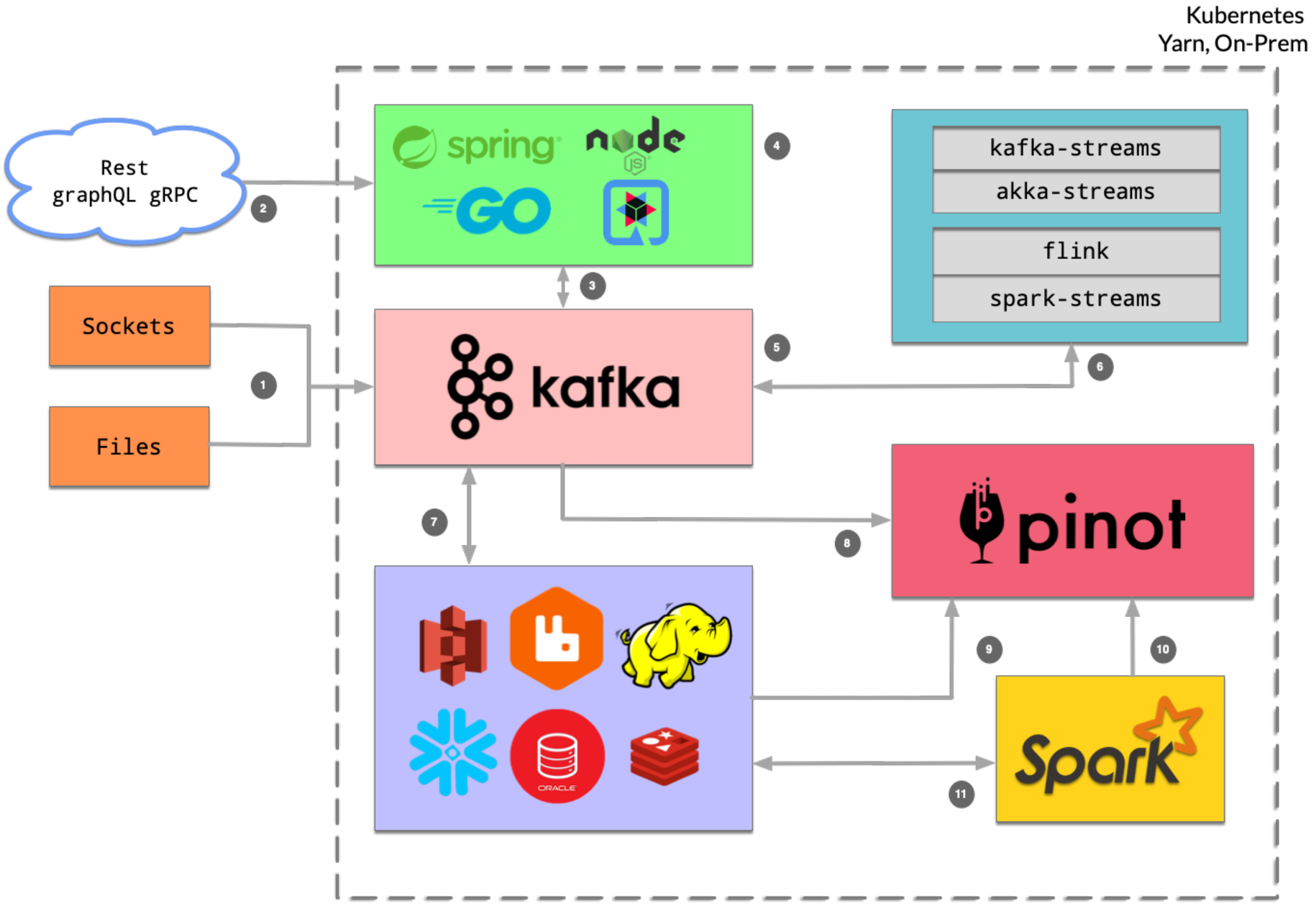
ID	Name	Cost
1	Dr. Magestic's Tonic	40
2	Dragonfeel Pillows	200
3	Sumatra Fresh	430
4	Mars Rover	212
5	Ennui Engine	190
6	Pittsburgh Passion	410

ID	Order ID	Quantity	Product ID
1	1	23	1
2	1	2	2
3	2	4	4
4	2	5	6
5	2	60	5
6	3	10	4

ID	State	Qty	Total
1	NY	30	4000
2	AZ	75	3500
3	CA	123	12322
4	NJ	80	3892
5	TX	3	1200
6	PA	90	3304

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


Kubernetes
Yarn, On-Prem
Rest
graphQL gRPC
Sockets
Files
spring
node
GO
kafka
kafka-streams
akka-streams
flink
spark-streams
pinot
ORACLE
Spark
1
2
3
4
5
6
7
8
9
10
11

The Tradeoffs

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

Outbox Pattern

Where will this fail?

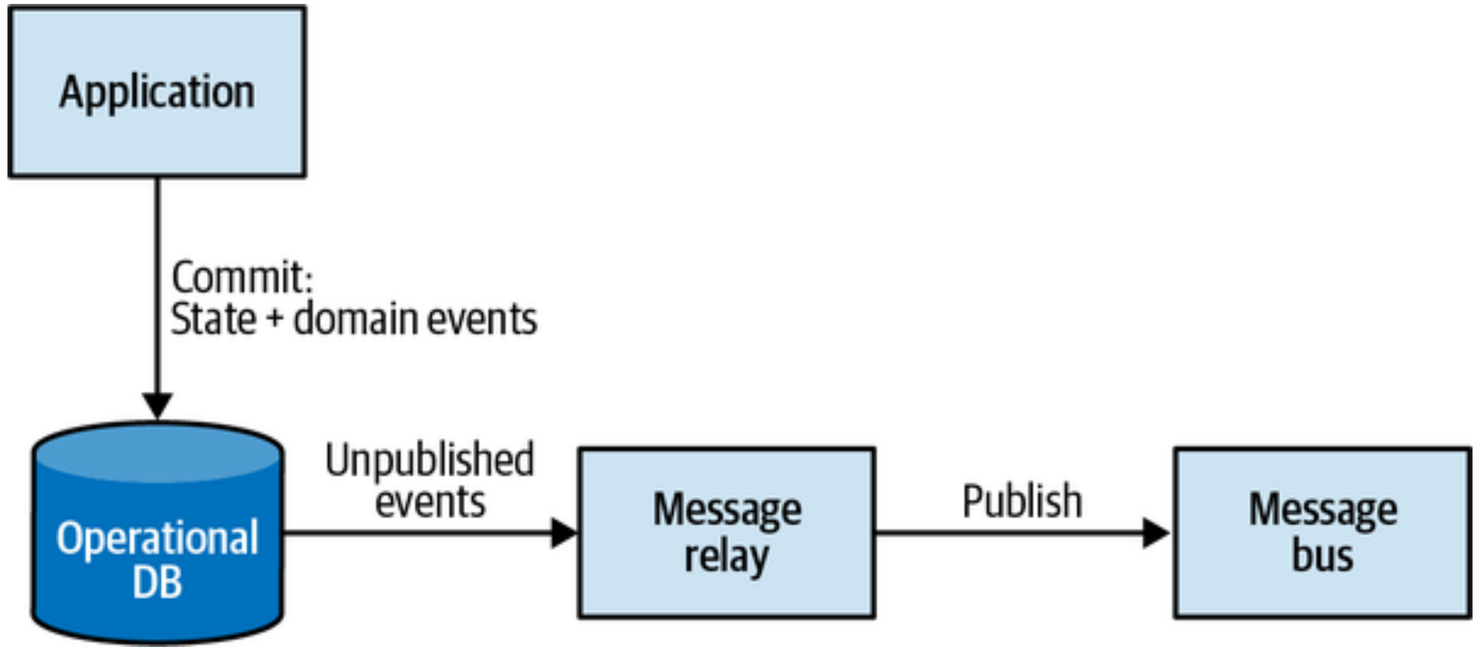
```
public class ManagementAPI {
     private final MessageBus messageBus;
     private final CampaignRepository repository;

     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

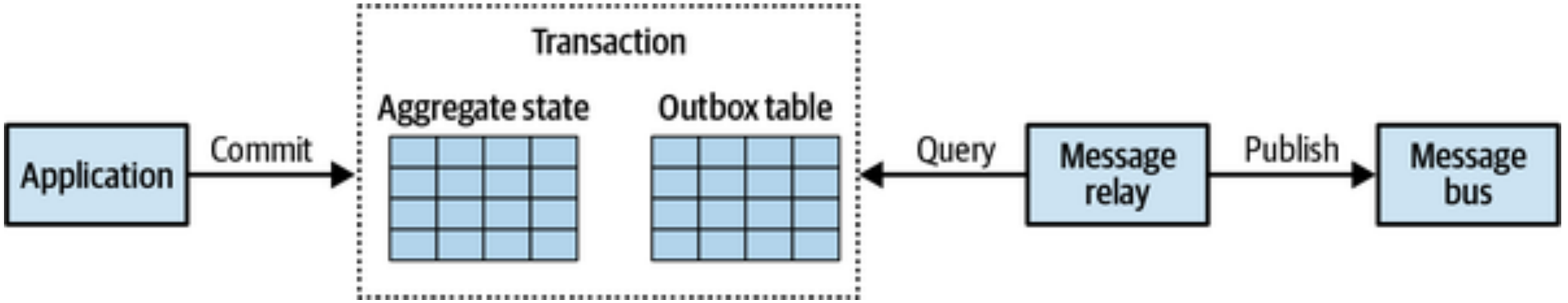
- Both the updated aggregate’s state and the new domain events are committed in the same atomic transaction.
- A message relay fetches newly committed domain events from the database.
- The relay publishes the domain events to the message bus.
- Upon successful publishing, the relay either marks the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



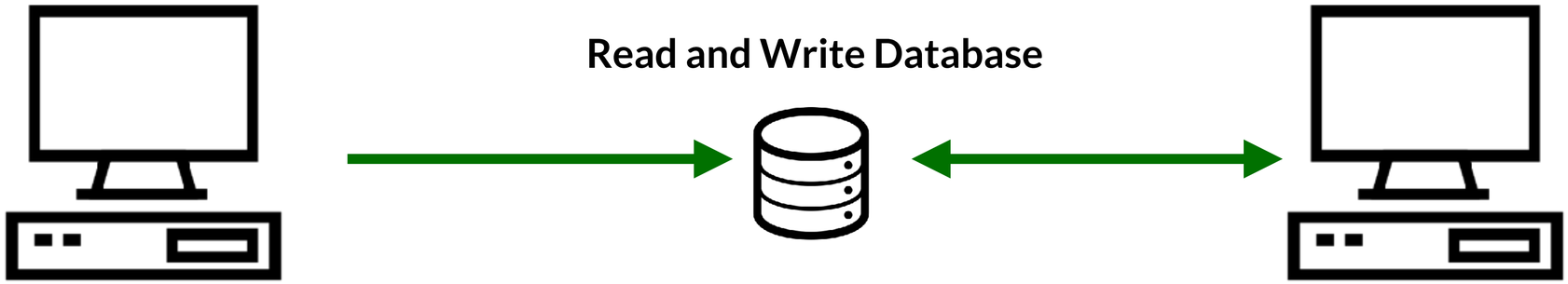
CQRS

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.

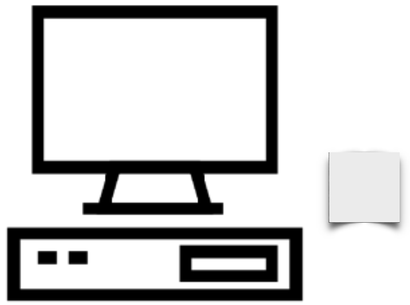
The Diagram

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

Lab: CQRS

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

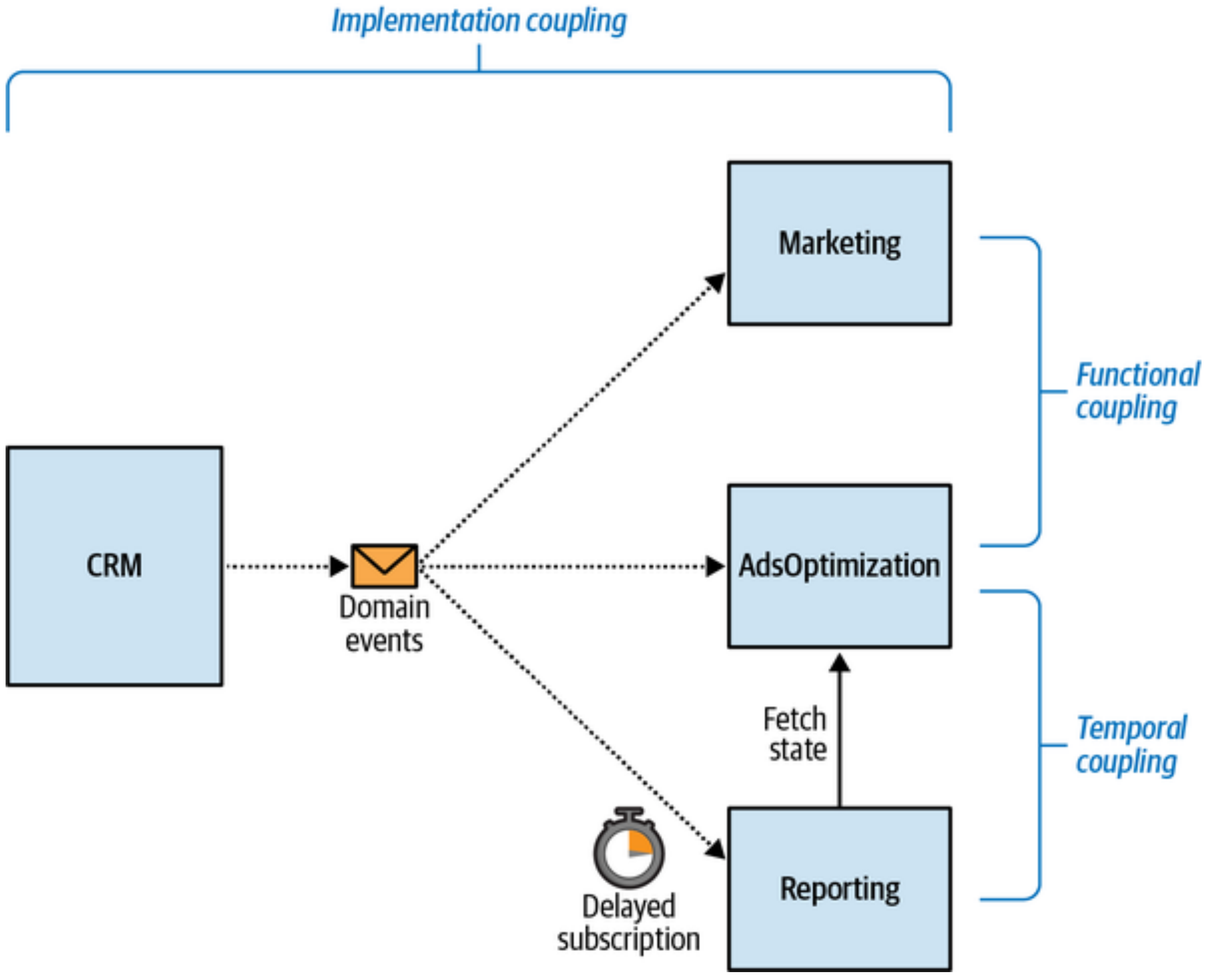
Message Coupling

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.

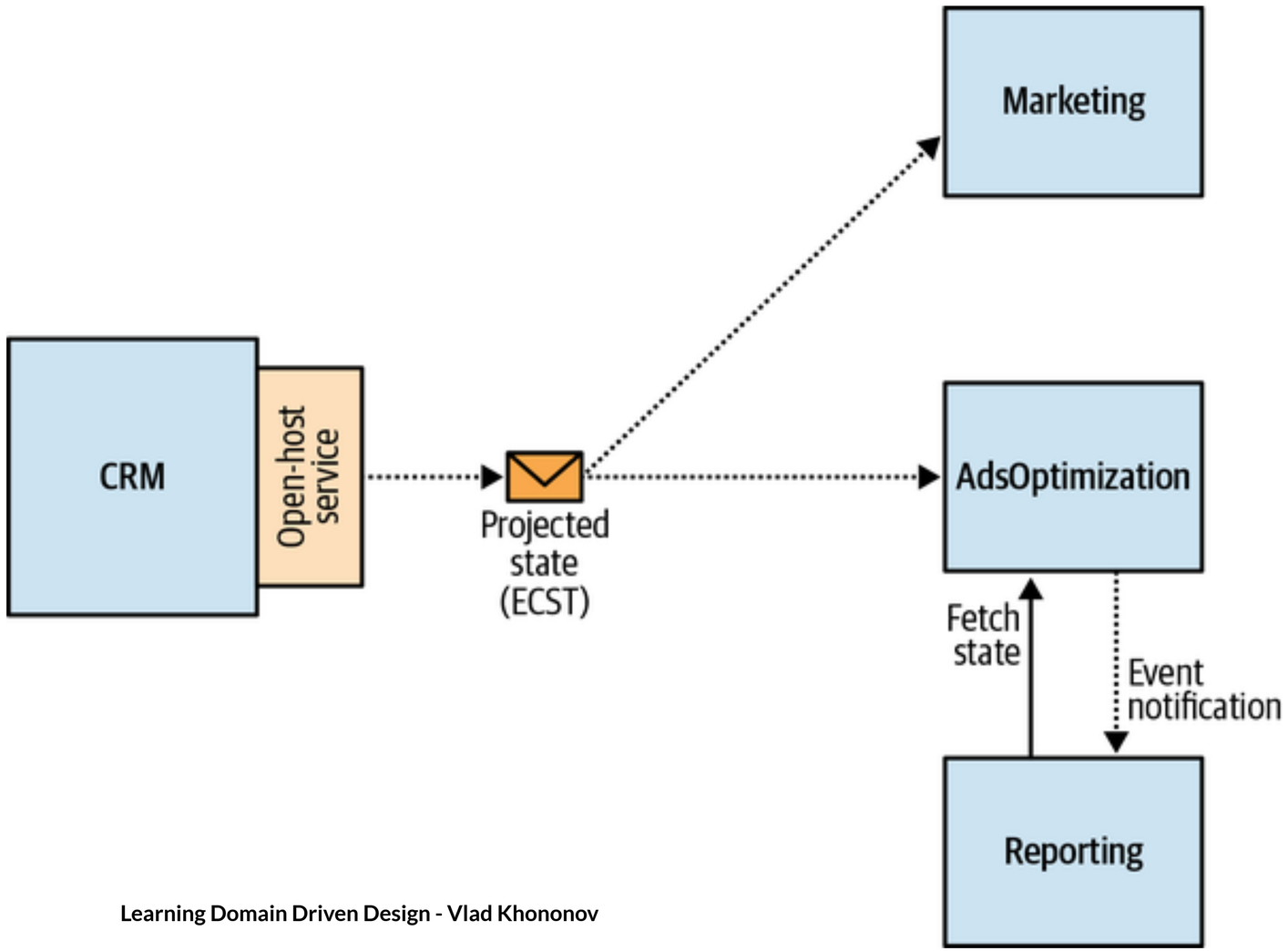
Functional Coupling

- 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_PAYMENT	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"
}
```

Saga Pattern

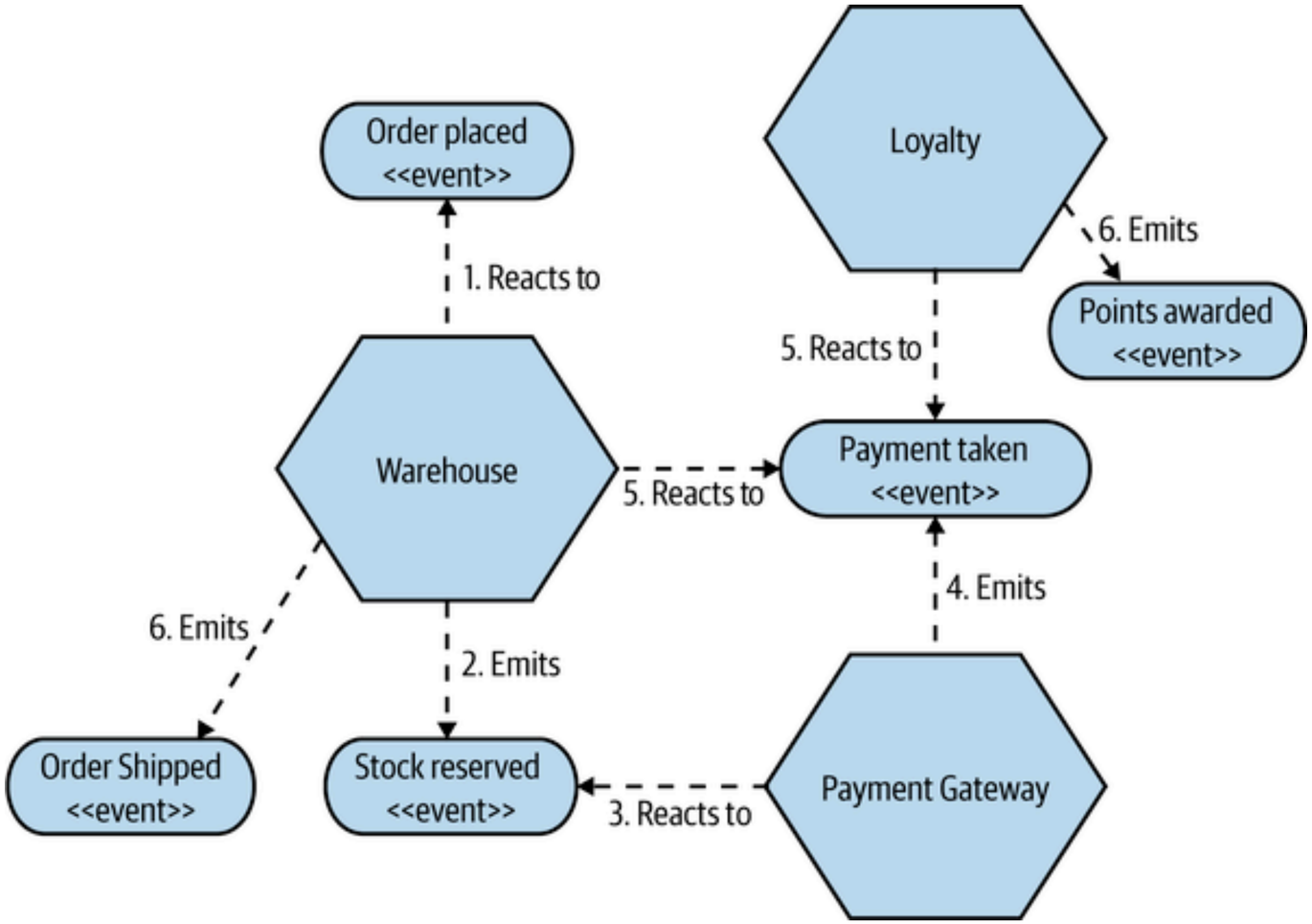
The Context

- 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.

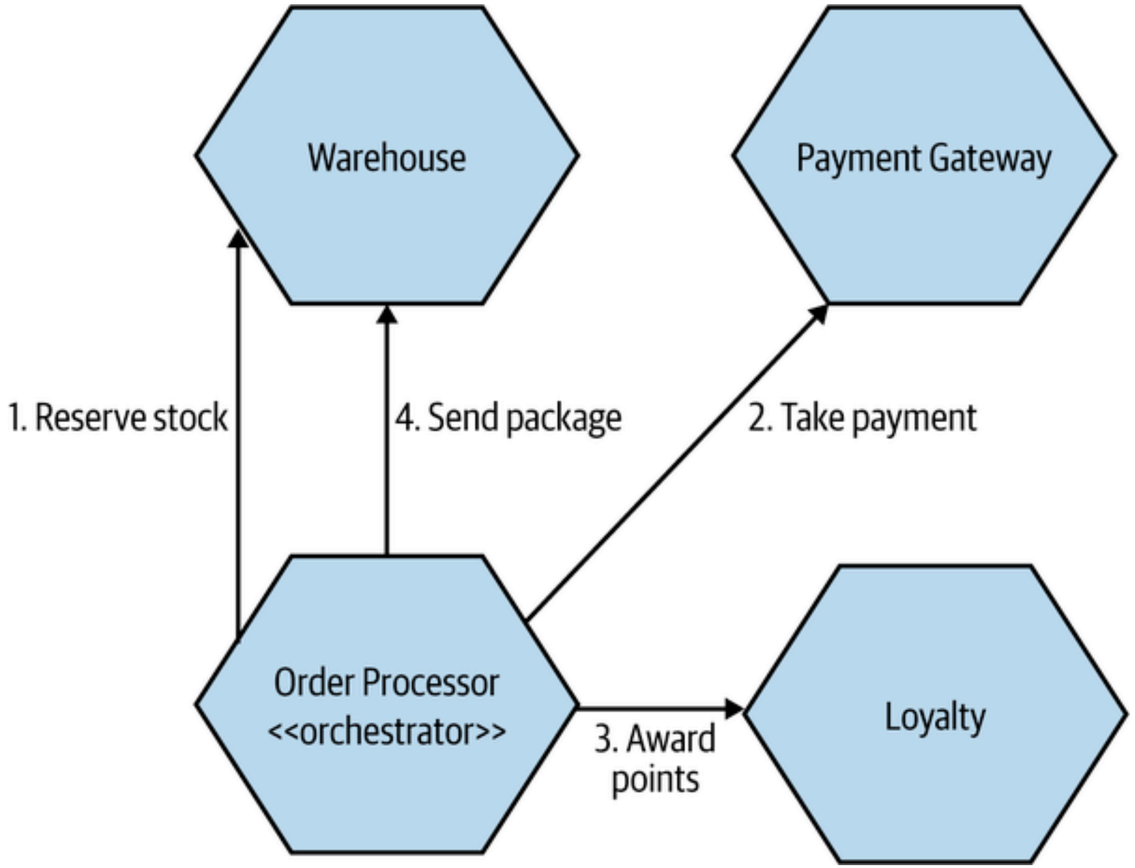
The Diagram



Choreography based pattern

Source: Building Microservices, 2nd Edition Sam Newman Published by O'Reilly Media, Inc.

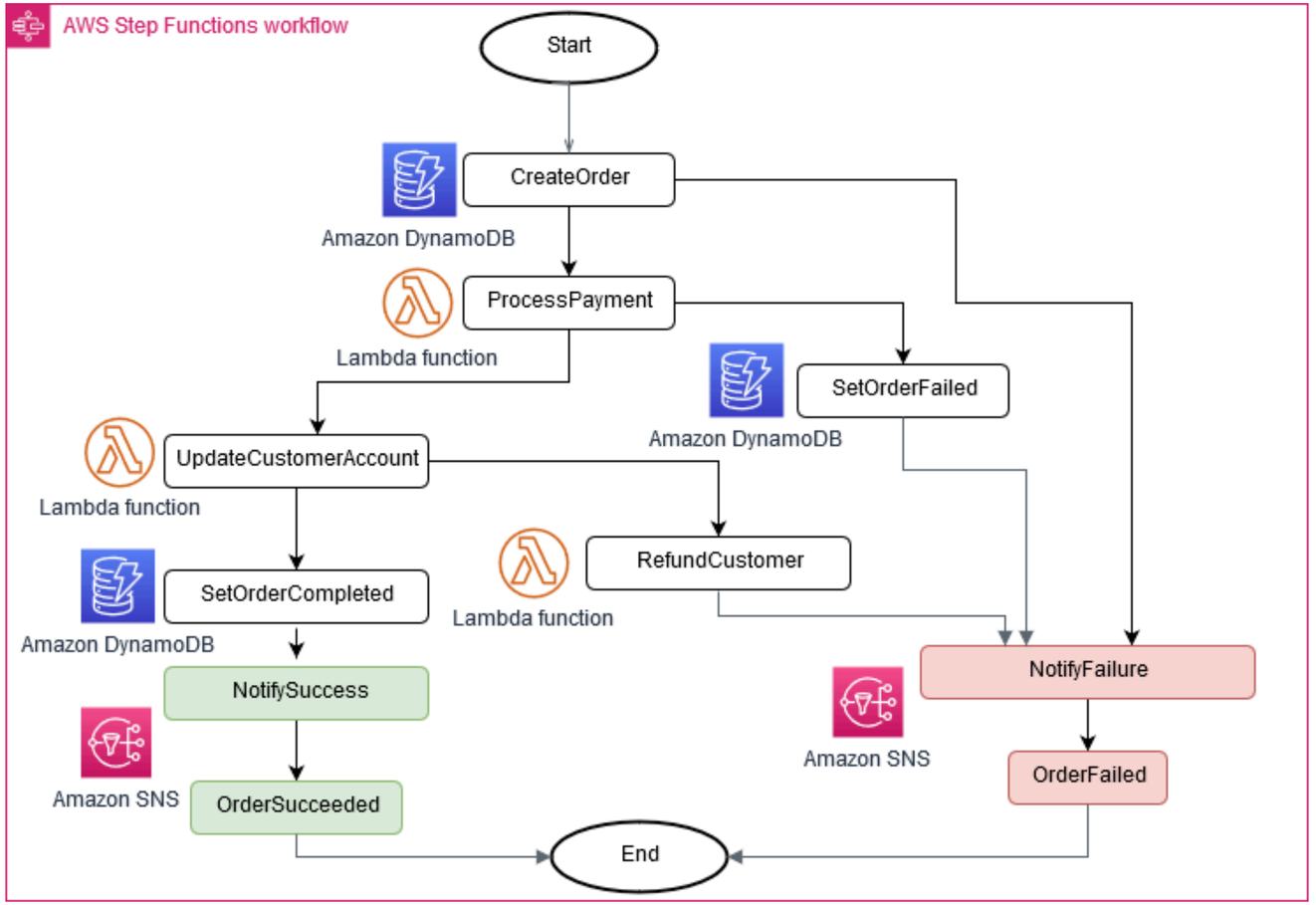
The Diagram



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

Source: Building Microservices, 2nd Edition Sam Newman Published by O'Reilly Media, Inc.

The Diagram



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

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