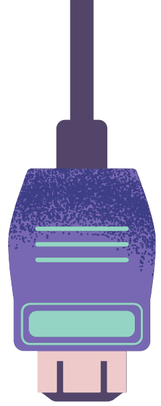
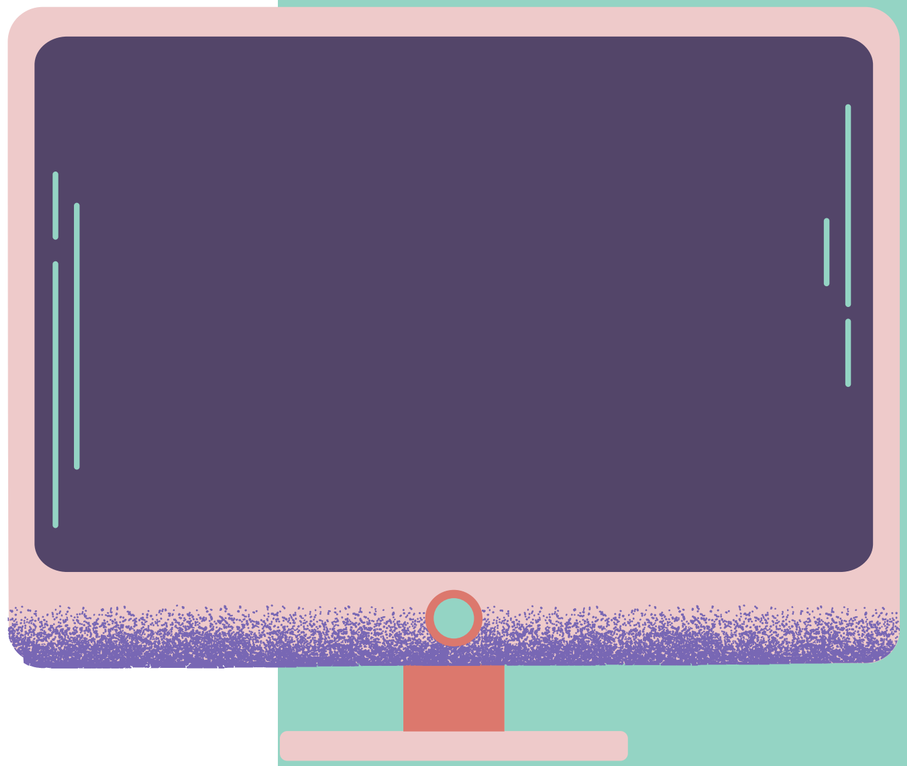
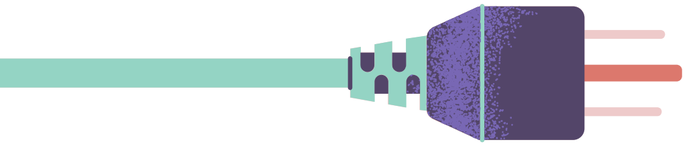
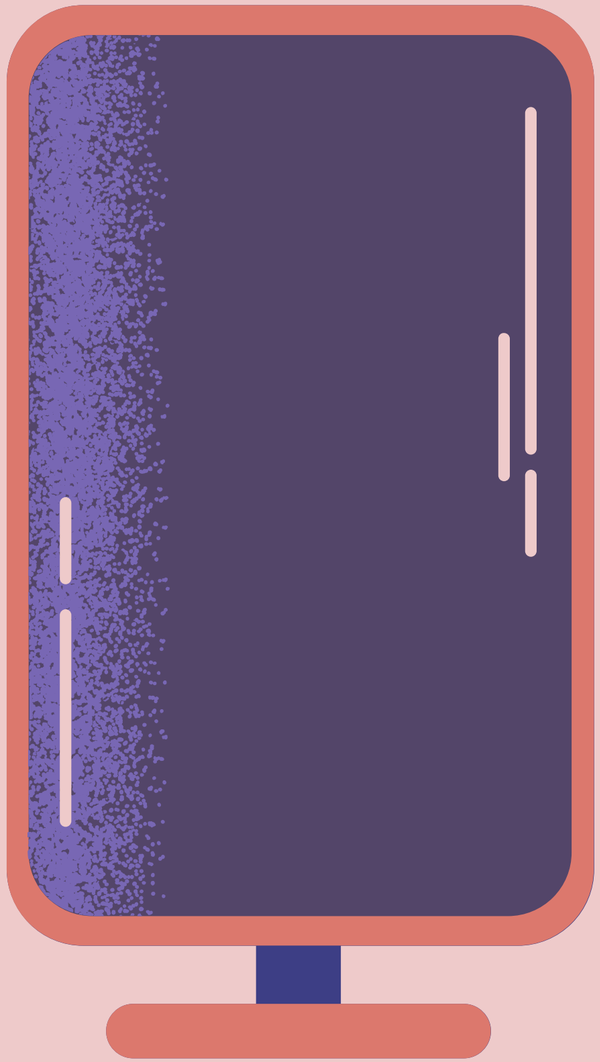


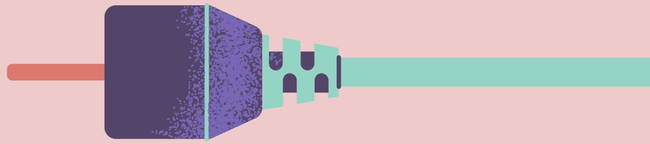
EVENT SOURCING

Neda Peyrone, Ph.D.



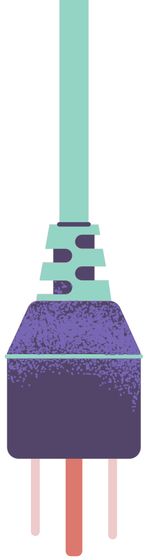






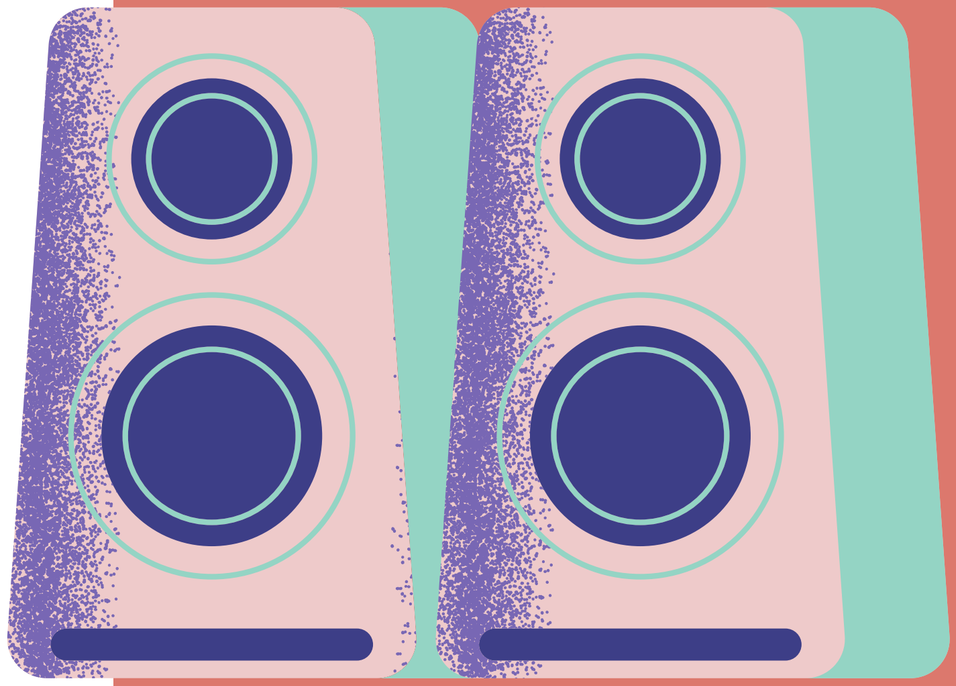
Outline

- Fundamentals of Event Sourcing
- Architecture & Design Patterns
- Hands-on Implementation in Golang
- Operational Considerations
- Benefits & Challenges

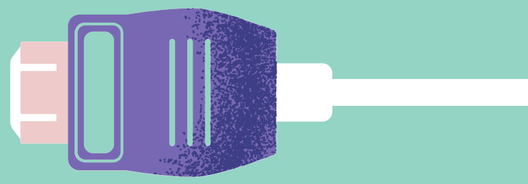


What is Event Sourcing?

Event Sourcing means saving every change as an event, so you can rebuild the full history later.

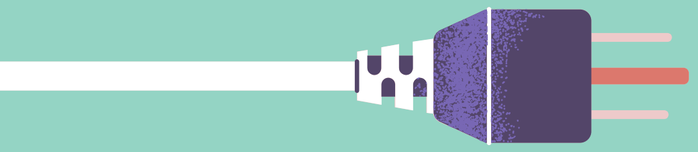


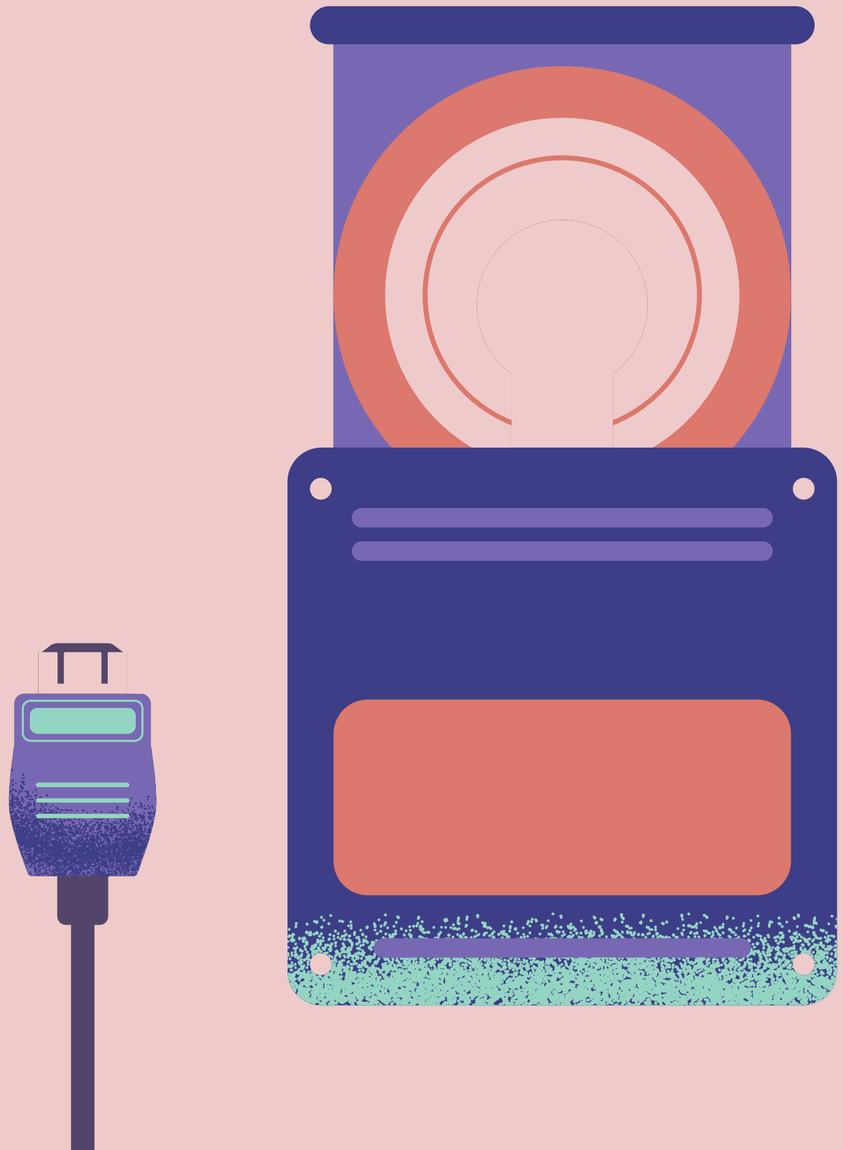
Example



Imagine an online order system:

- **OrderCreated:** when a user makes a new order.
- **ItemAdded:** when they add a product.
- **ItemRemoved:** when they remove a product.
- **OrderCheckedOut:** when they finish payment.

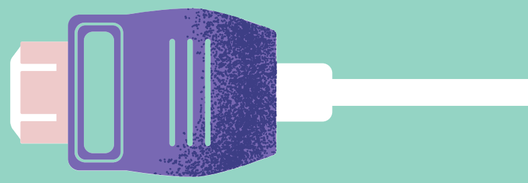




Why Event Sourcing?

Traditional CRUD updates overwrite data, and you lose history. Event Sourcing keeps every change as an event, so nothing is lost.

Example

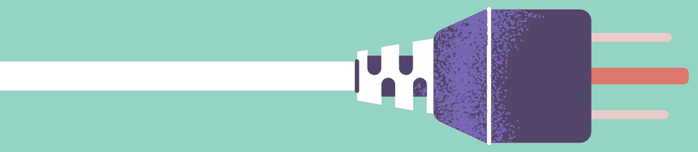


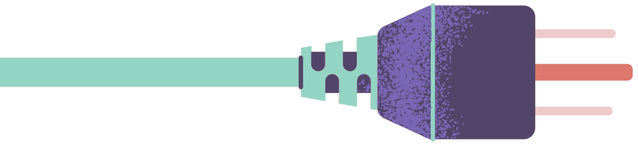
CRUD Operations:

- A user updates their address.
- The old address is replaced, and you can’t see what it was before.

Event Sourcing

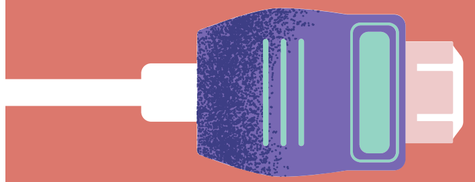
- Record **UserAddressChanged** event with old and new addresses.
- You can track how, when, and why the address changed over time.



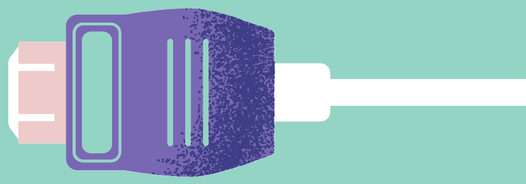


Aggregates and Streams

- An aggregate is one main object, like an order or account.
- Each aggregate has its own list of events called a stream.



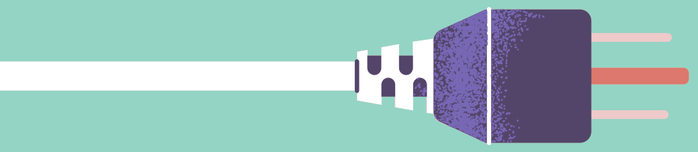
Example



For an Order aggregate:

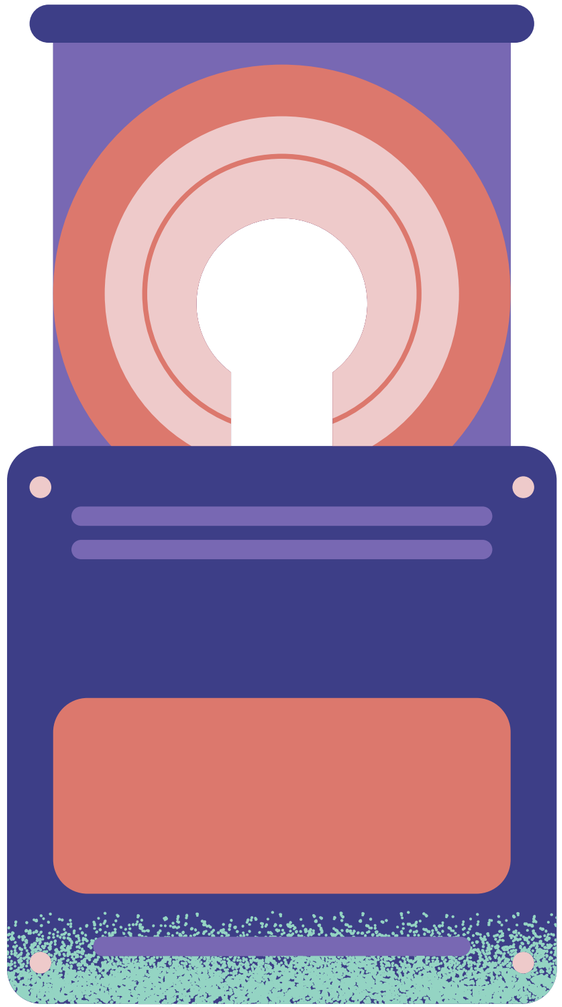
- OrderCreated
- ItemAdded
- ItemRemoved
- OrderCheckedOut

These events together form the **Order’s event stream**.



Projections (Read Models)

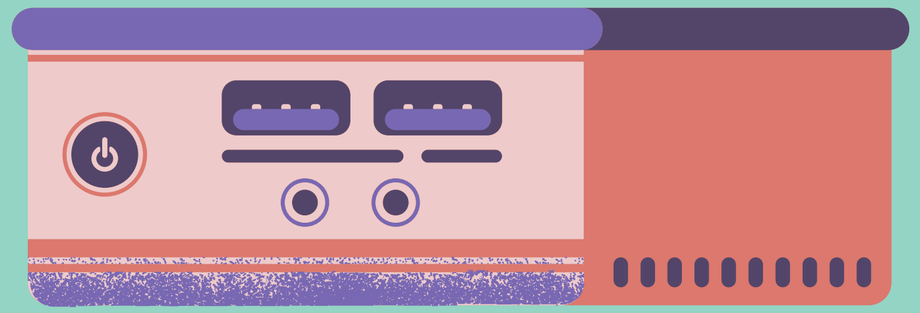
- Turn events into easy-to-read data for queries.
- They show the current state built from all past events.



Example

From order events, create a table that shows each order's total items and status.

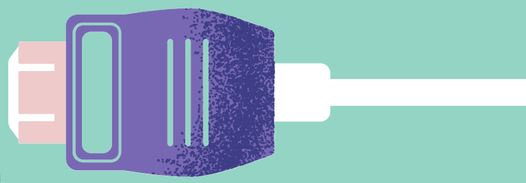
OrderID	Customer	TotalItems	Status
1001	Alice	3	In Progress
1002	Bob	2	Check Out
1003	Carol	5	In Progress



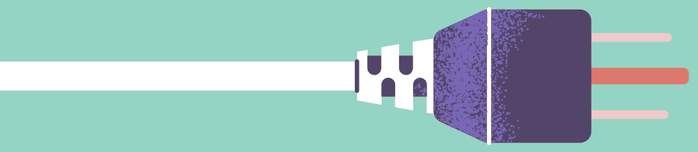
Snapshots (Performance Only)

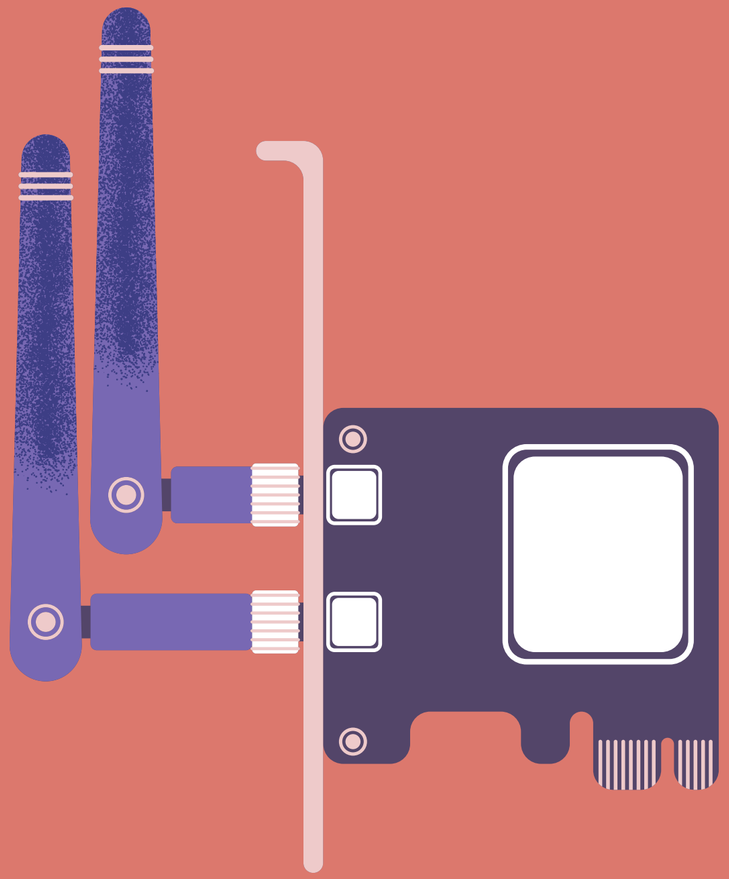
- Save the full state of an object at a point in time.
- This makes loading faster without replaying all events.

Example



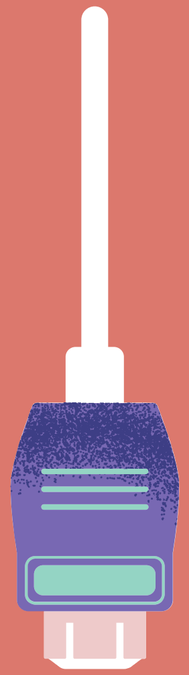
- Instead of replaying 1,000 order events, load the last snapshot and apply only the latest changes.



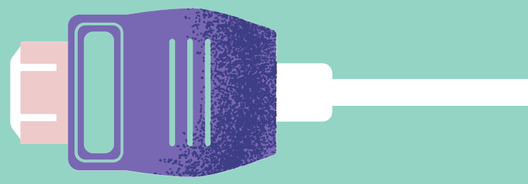


What is CQRS?

- Command Query Responsibility Segregation (CQRS) means separating how you write and read data.
- Write handle commands that change data, and reads use simple views for queries.



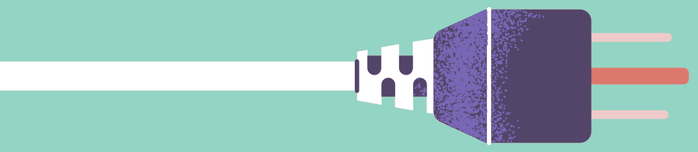
Example

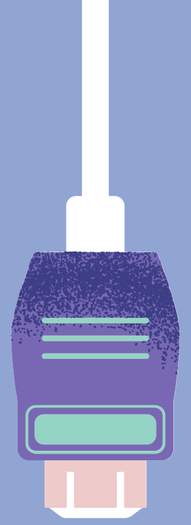


When a user adds an item to an order:

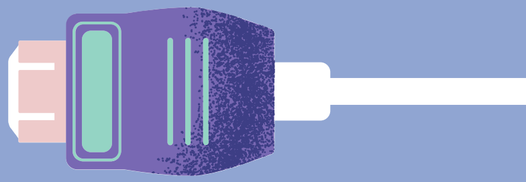
- **Write side:** save ItemAdded event.
- **Read side:** show the updated order summary from the projection.

CQRS helps make event-sourced systems faster and easier to scale.





Questions



Q1

What does “SC Created” mean in the diagram?

Q2

Where are all the events saved?

Q3

What is the job of the EventBus?

Q4

Why do we have a Read Database?

Q5

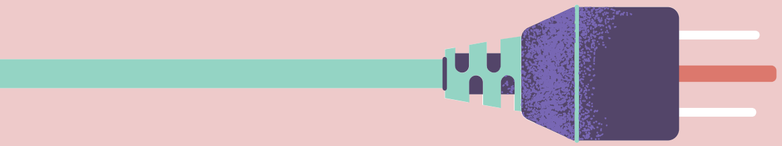
What does “Replay” do?

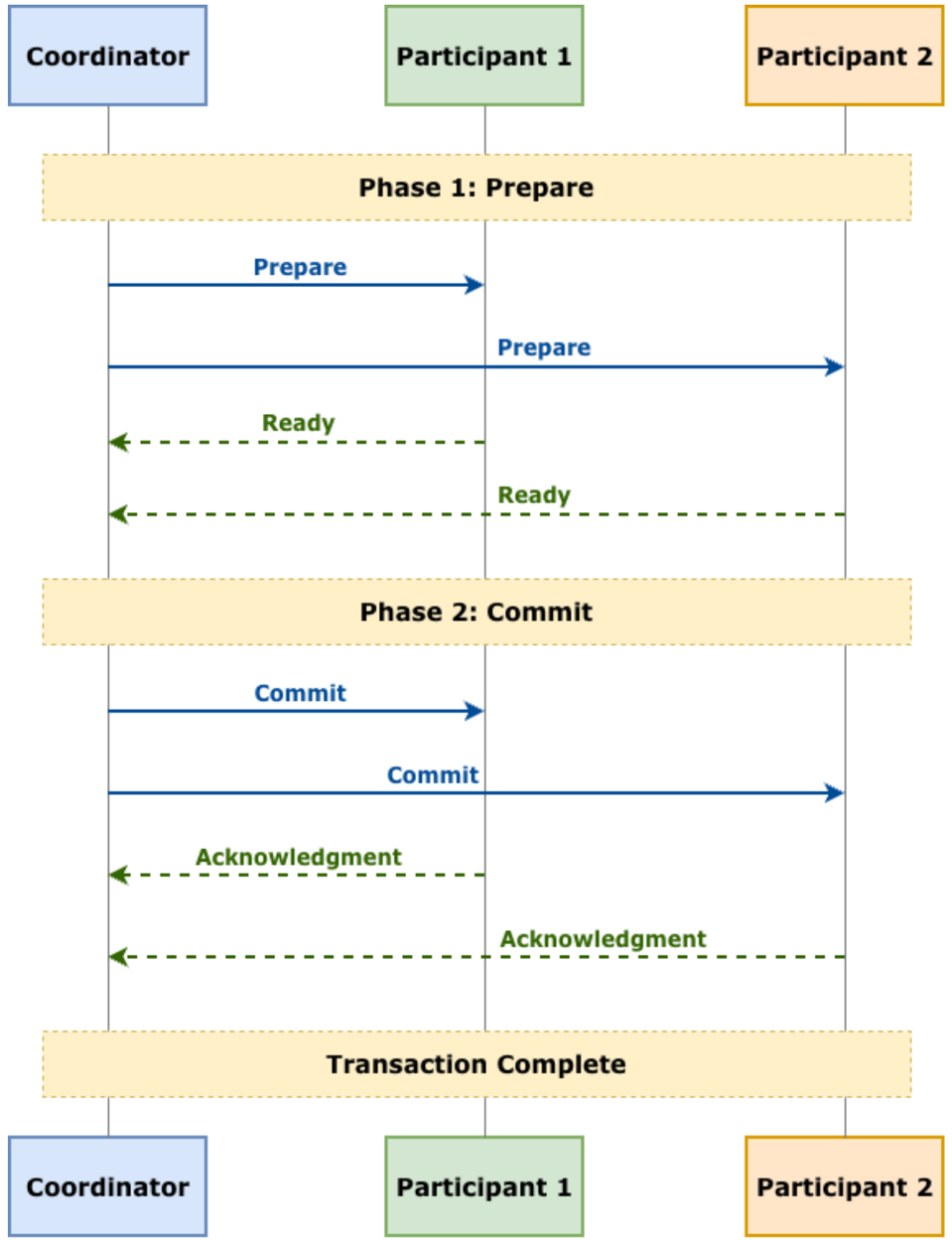
Two Phase Commit (2PC)

2PC is a mechanism that ensures all systems agree before committing a transaction.

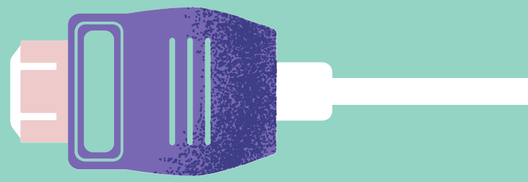
It has two steps:

- **Prepare:** Each system says it is ready to commit.
- **Commit:** The main coordinator tells all systems to finish the commit.

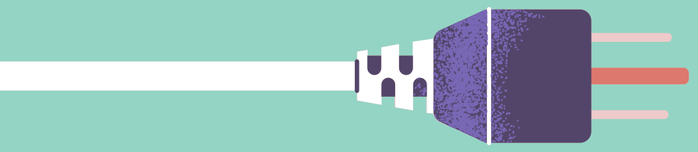


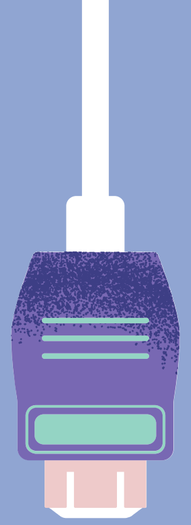


Example

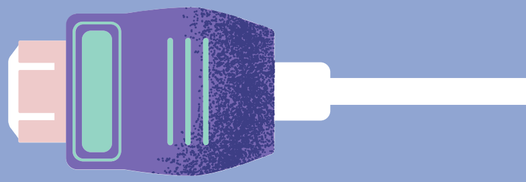


- If an order update affects both Payment and Inventory databases, 2PC waits for both to confirm before saving.
- This keeps data consistent but makes the process slower and more complex.





Questions



Q6

How many steps does 2PC have?

Q7

Why is 2PC used?

Q8

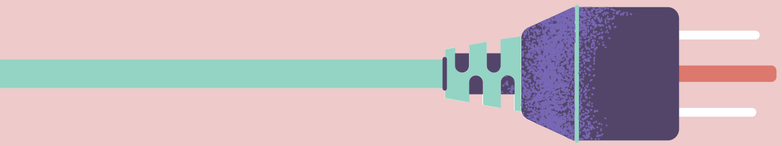
What happens if one system fails during 2PC?

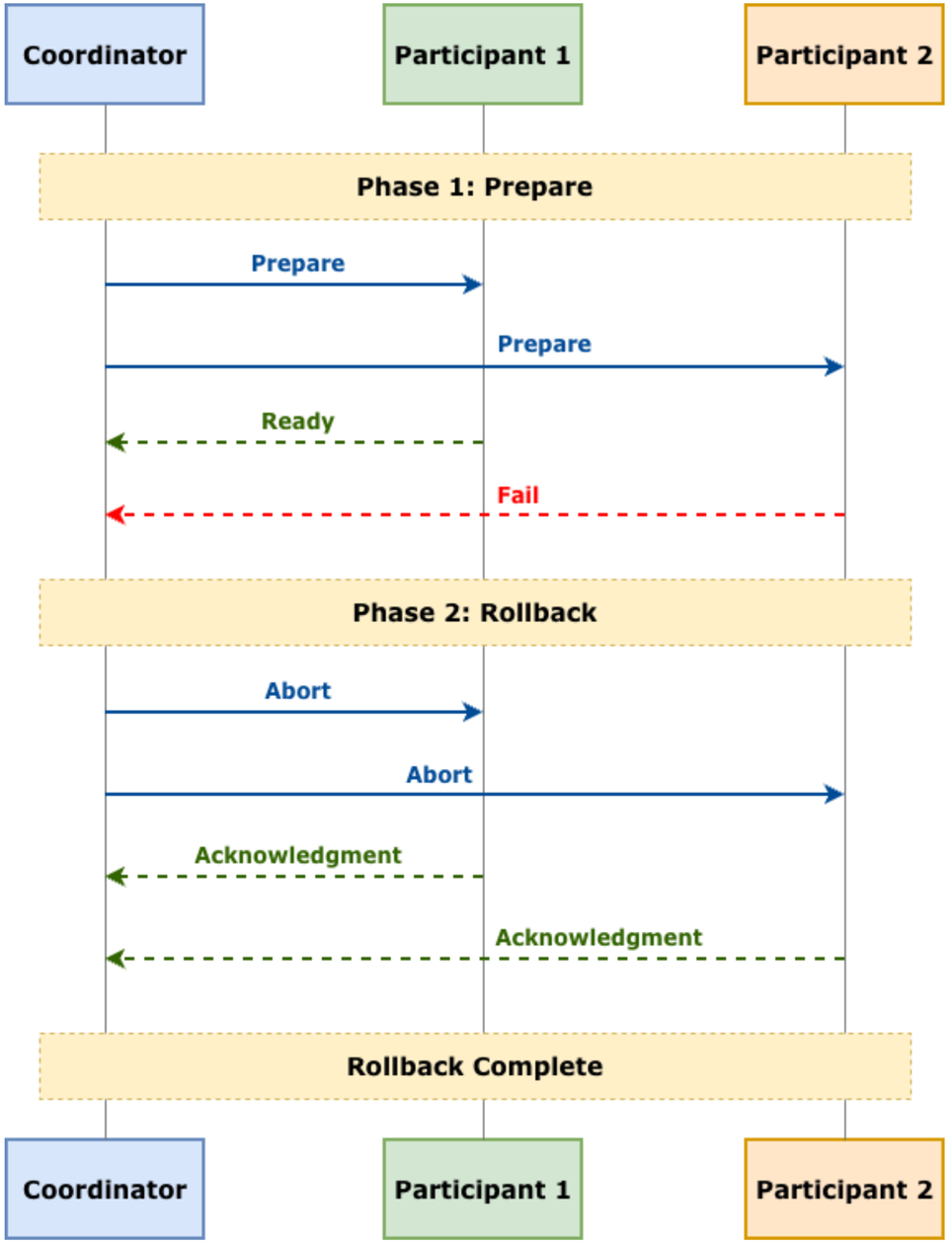
Q9

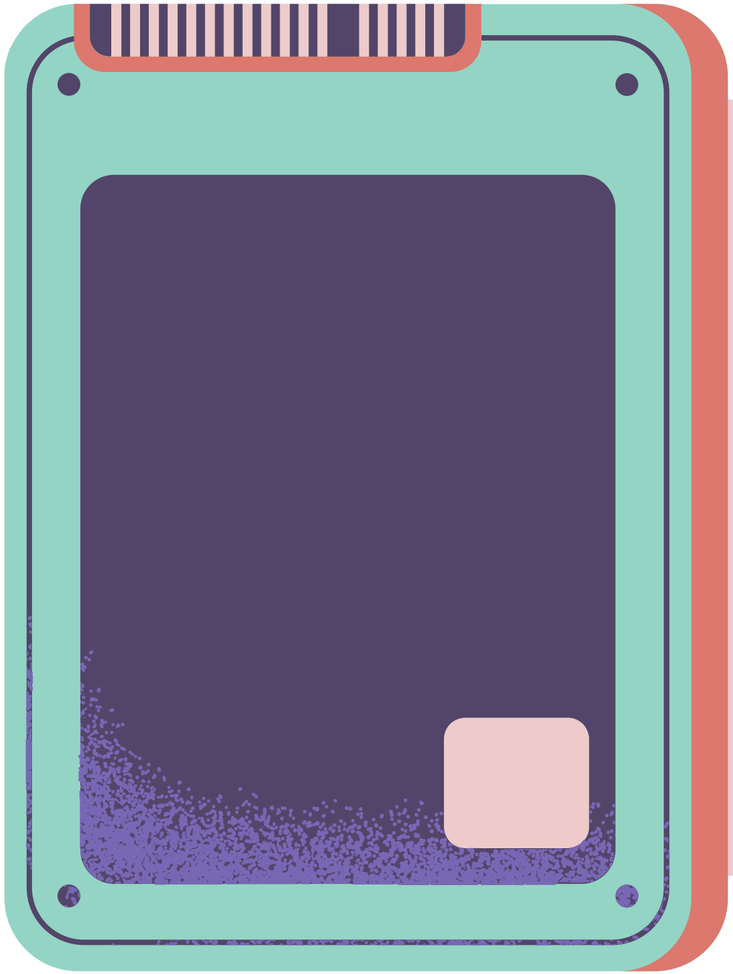
What is one problem with 2PC?

2PC - Fail Case

- When rollback is done, all systems are back to a consistent state.
- 2PC ensures everyone agrees before saving data, but it can be slow if one participant delays.

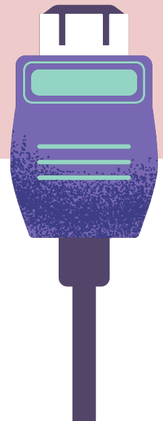




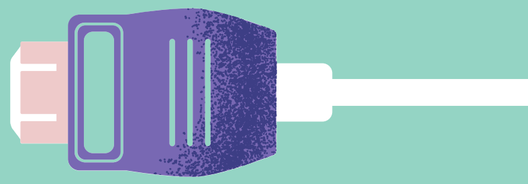


Atomicity Without 2PC

- Saving one event is a single, complete action (atomic).
- This means the system does not need a 2PC to stay consistent.



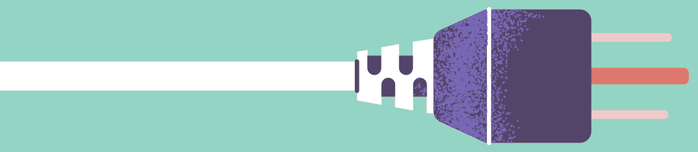
Example



When a payment is completed:

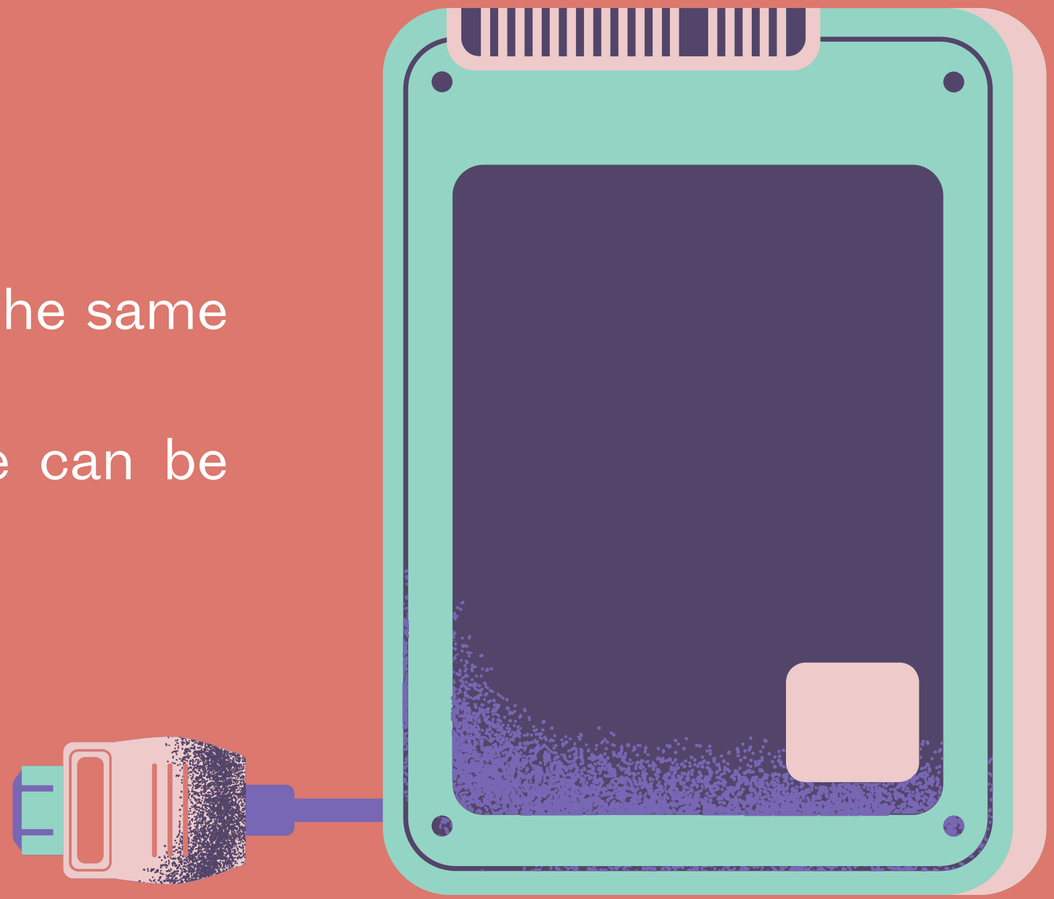
- The system saves a PaymentCompleted event in the event store.
- Other services read this event from the store and update their data.

All services stay consistent without complex transactions.

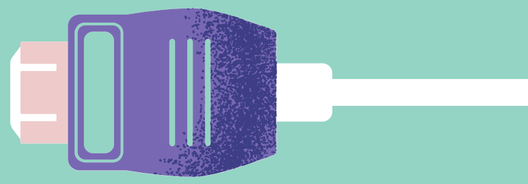


Ordering Guarantees

- Events must happen and be read in the same order they were saved.
- If the order changes, the final state can be wrong.



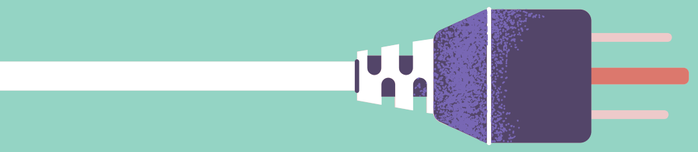
Example



For an order:

- ItemAdded
- ItemRemoved
- OrderCheckedOut

If these events are processed out of order (for example, checkout before add), the system will show the wrong result.

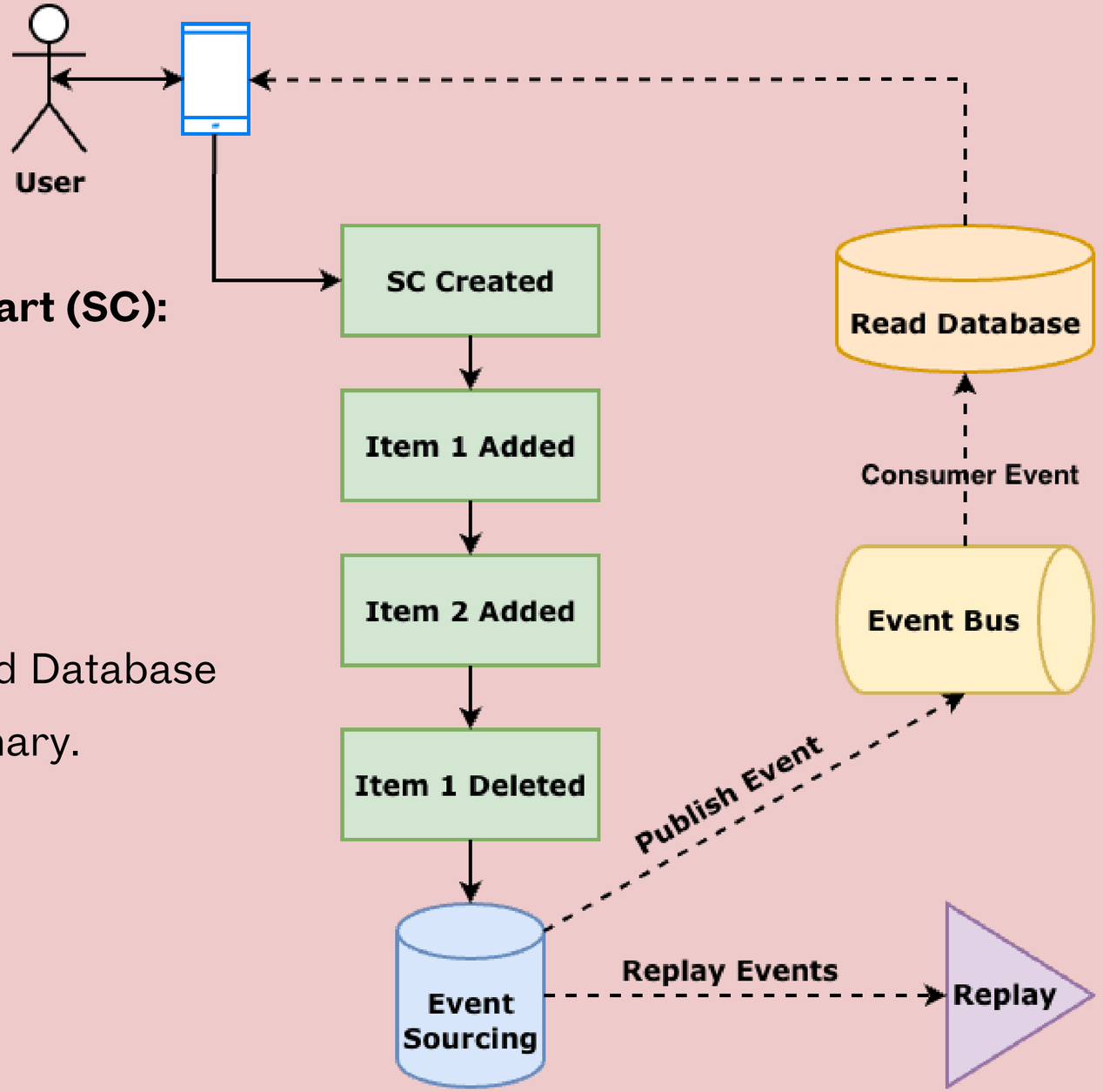


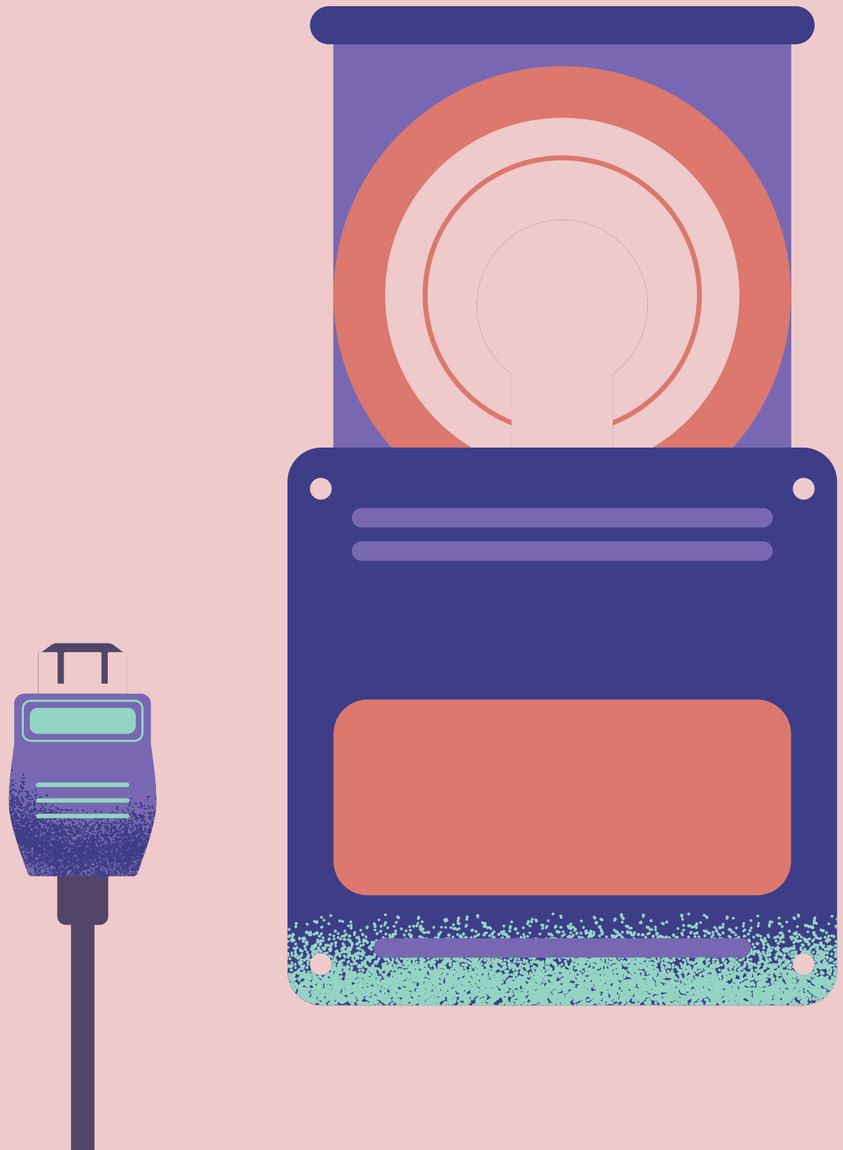
Event Sourcing Pattern

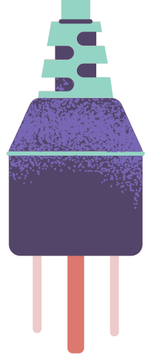
A user adds and removes items from a shopping cart (SC):

- SCCreated
- ItemAdded
- ItemAdded
- ItemRemoved

All these are stored in the Event Store, and the Read Database (projection) will always show the current cart summary.







Benefits

01 Every change is stored as an event, so nothing is lost.

02 Rebuild the system state at any point in time.

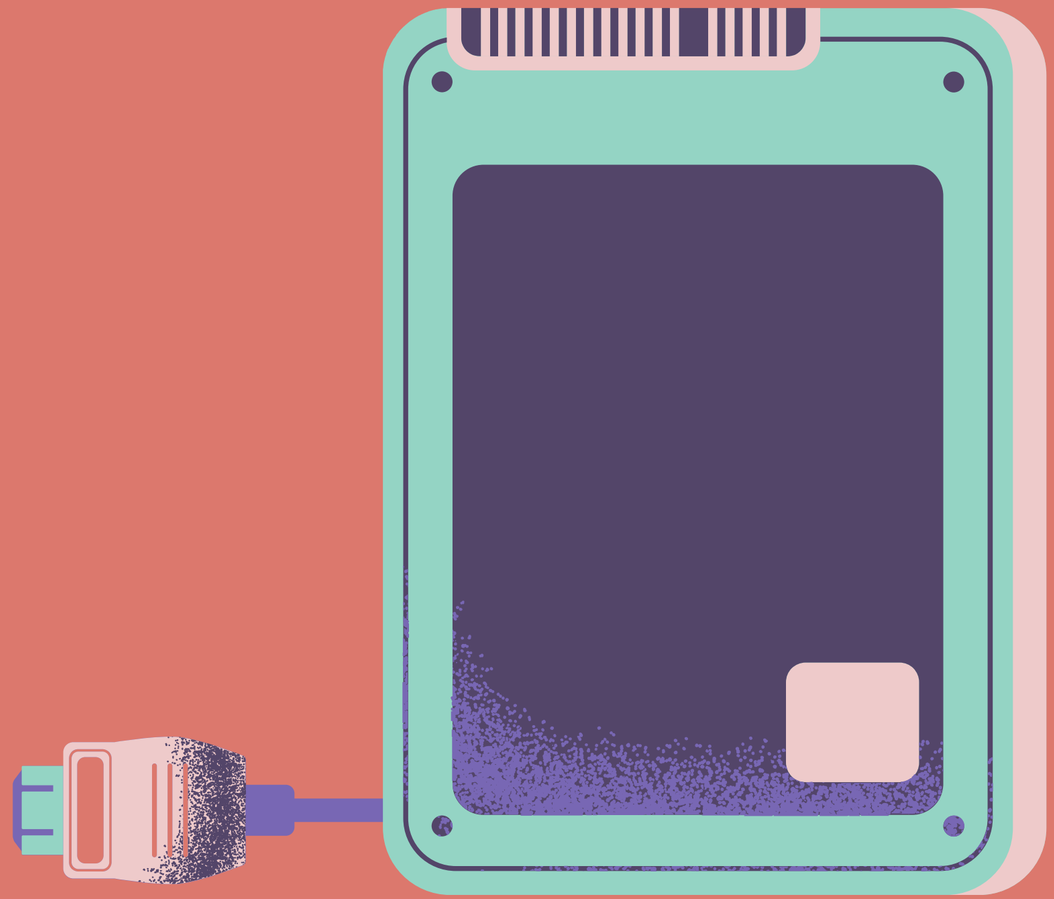
03 Easy to separate write (events) and read (projections) models.

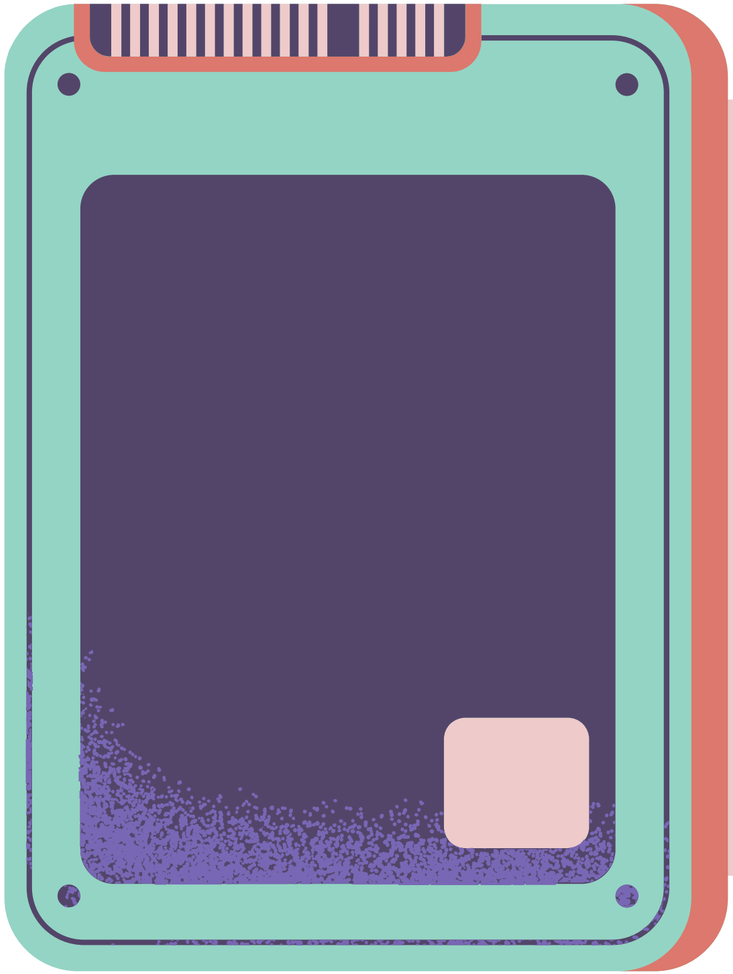
04 Events can be shared across microservices easily.

05 Historical events help find patterns and insights.

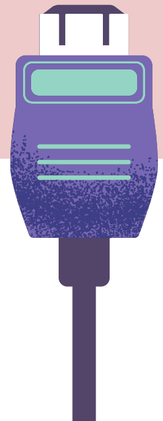
Disadvantages

- Harder to learn and implement.
- Complex to query current state.
- Needs extra patterns like CQRS.
- Data may be eventually consistent.





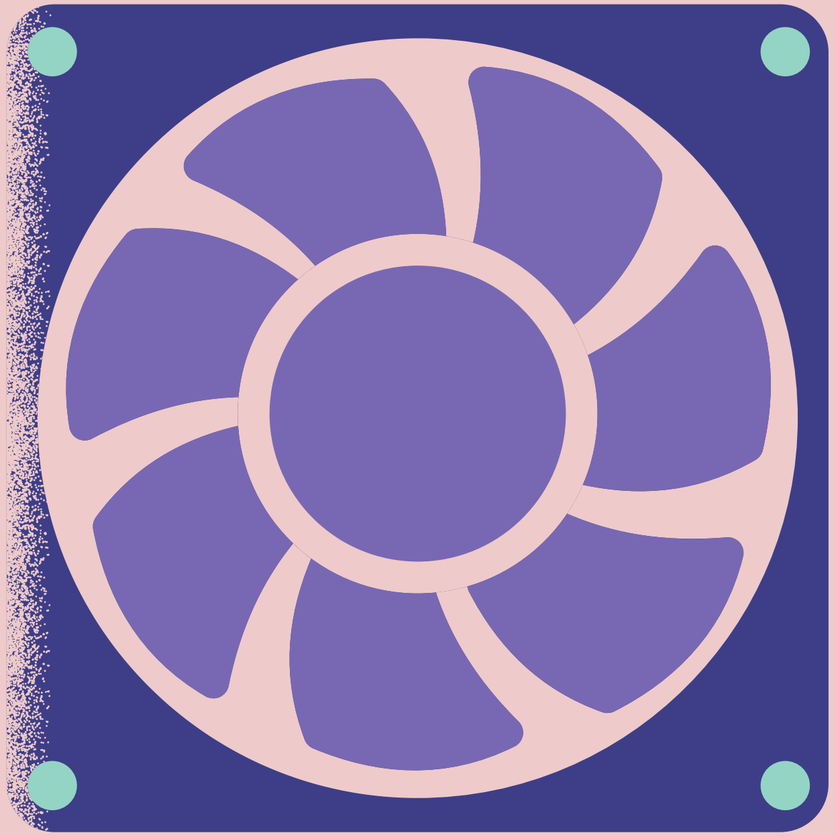
What types of applications should use Event Sourcing?



Hands-on Example

GitHub Repository:

- https://github.com/peyrone/go-event-sourcing



Q&A