Rethinking Services with Stateful Streams

Ben Stopford
@benstopford

THIS IS ME

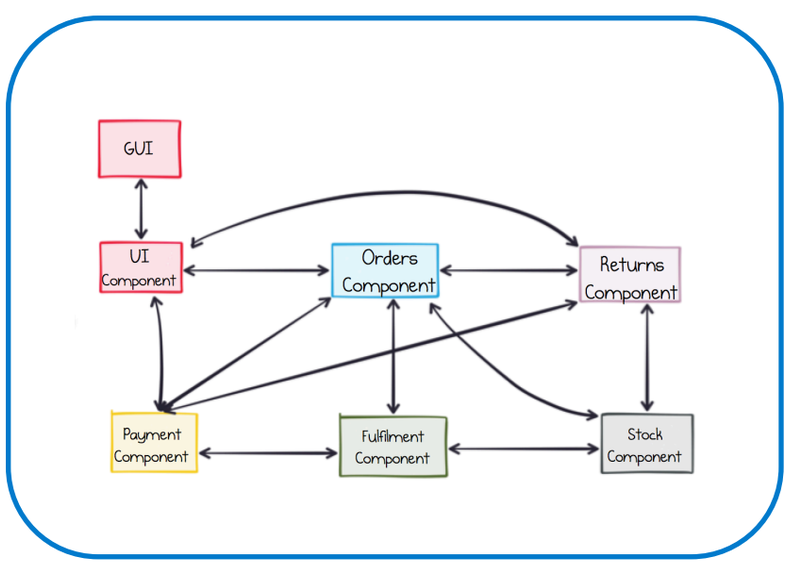
- ENGINEER AT CONFLUENT
- EX THOUGHTWORKS + UK FINANCE

What are microservices really about?

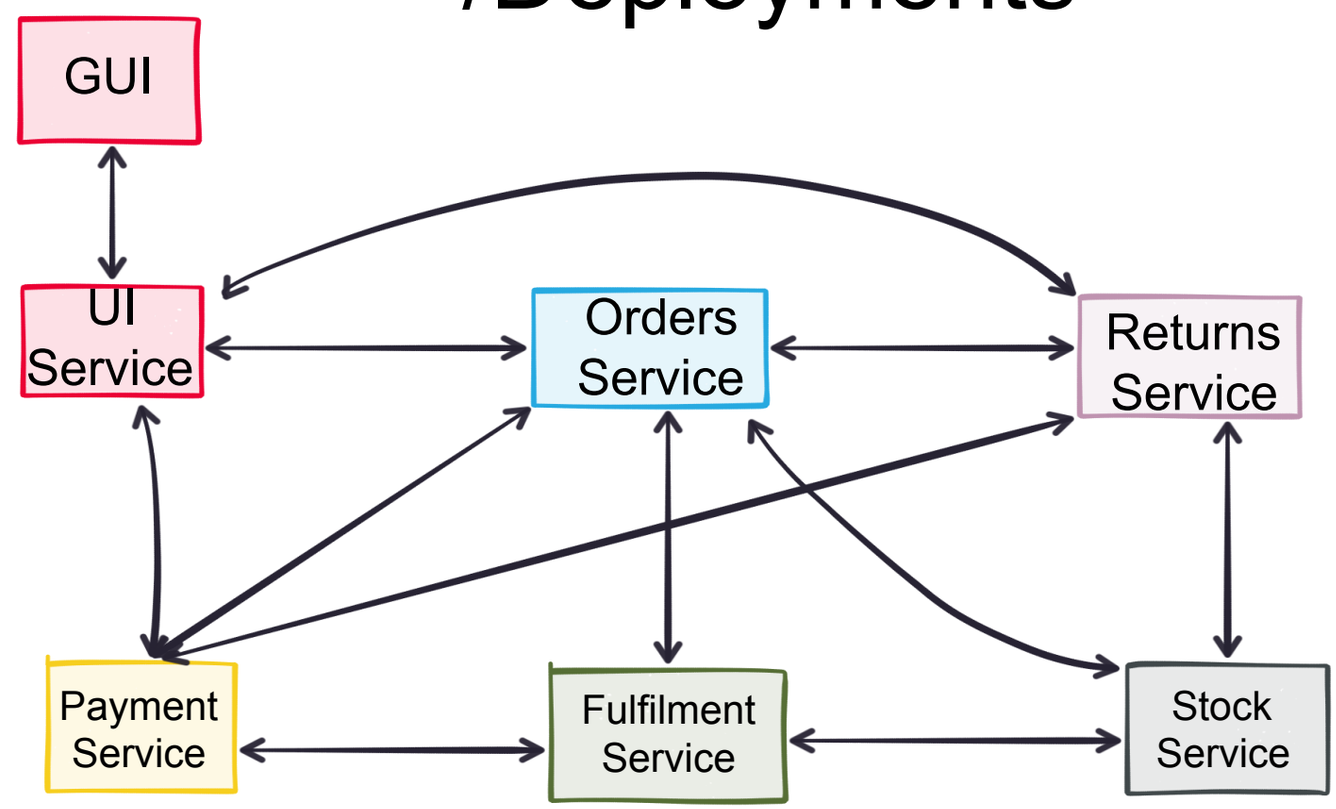
confluent

Splitting the Monolith

Single Process
/Code base
/Deployment

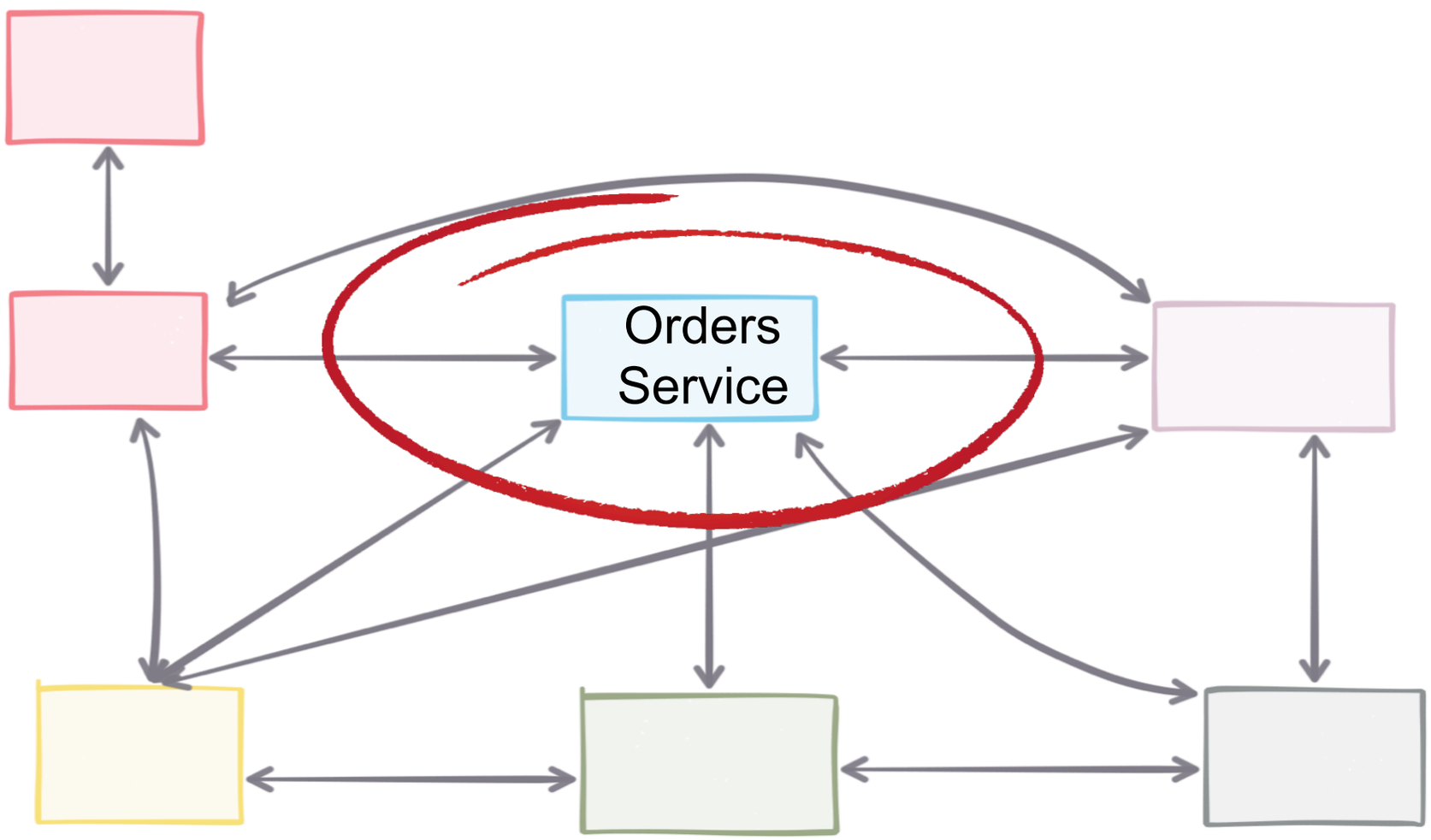


Many Processes
/Code bases
/Deployments

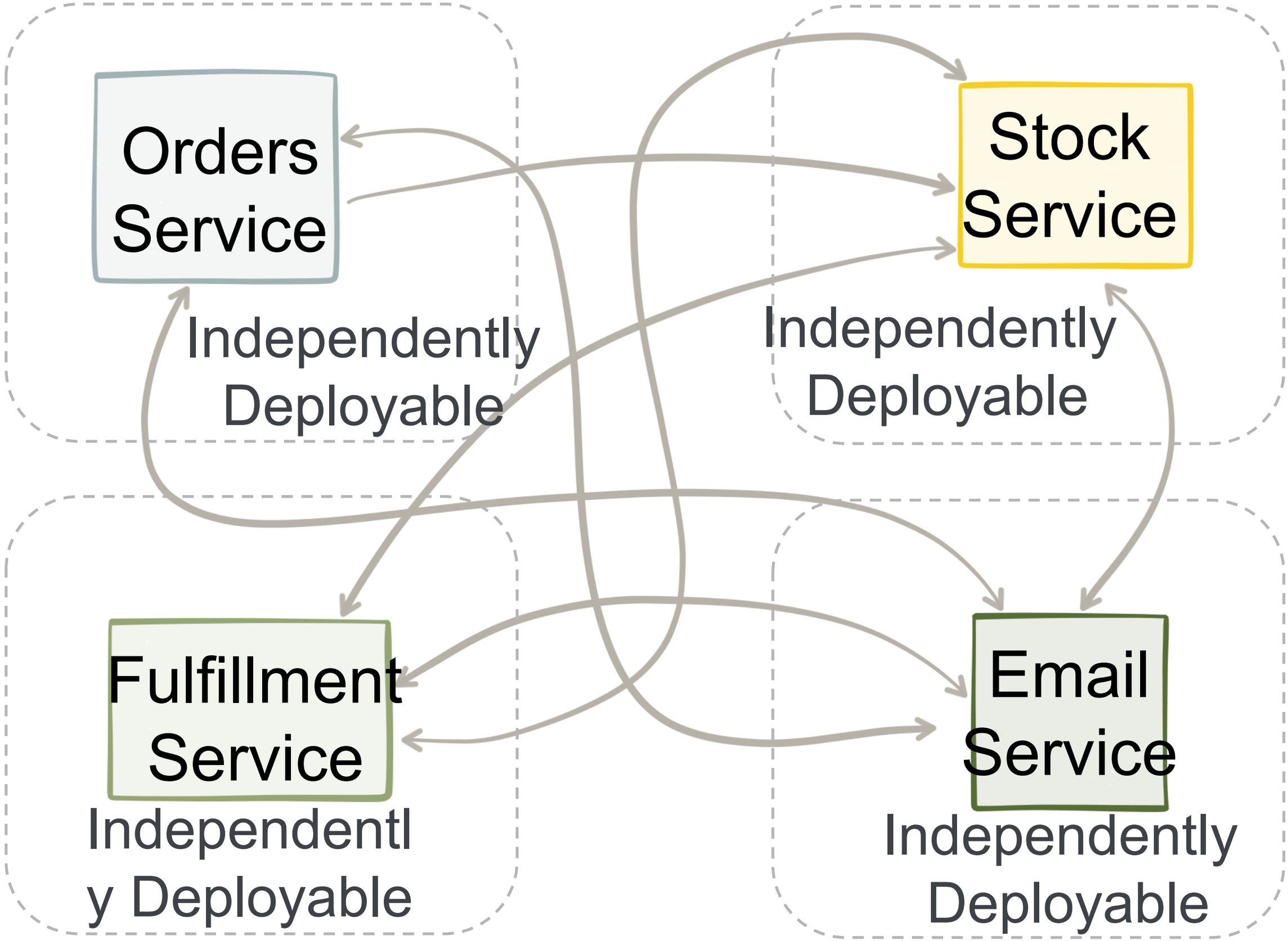


confluent

Autonomy?



confluent


Orders Service
Independently Deployable
Stock Service
Independently Deployable
Fulfillment Service
Independentl y Deployable
Email Service
Independently Deployable

confluent

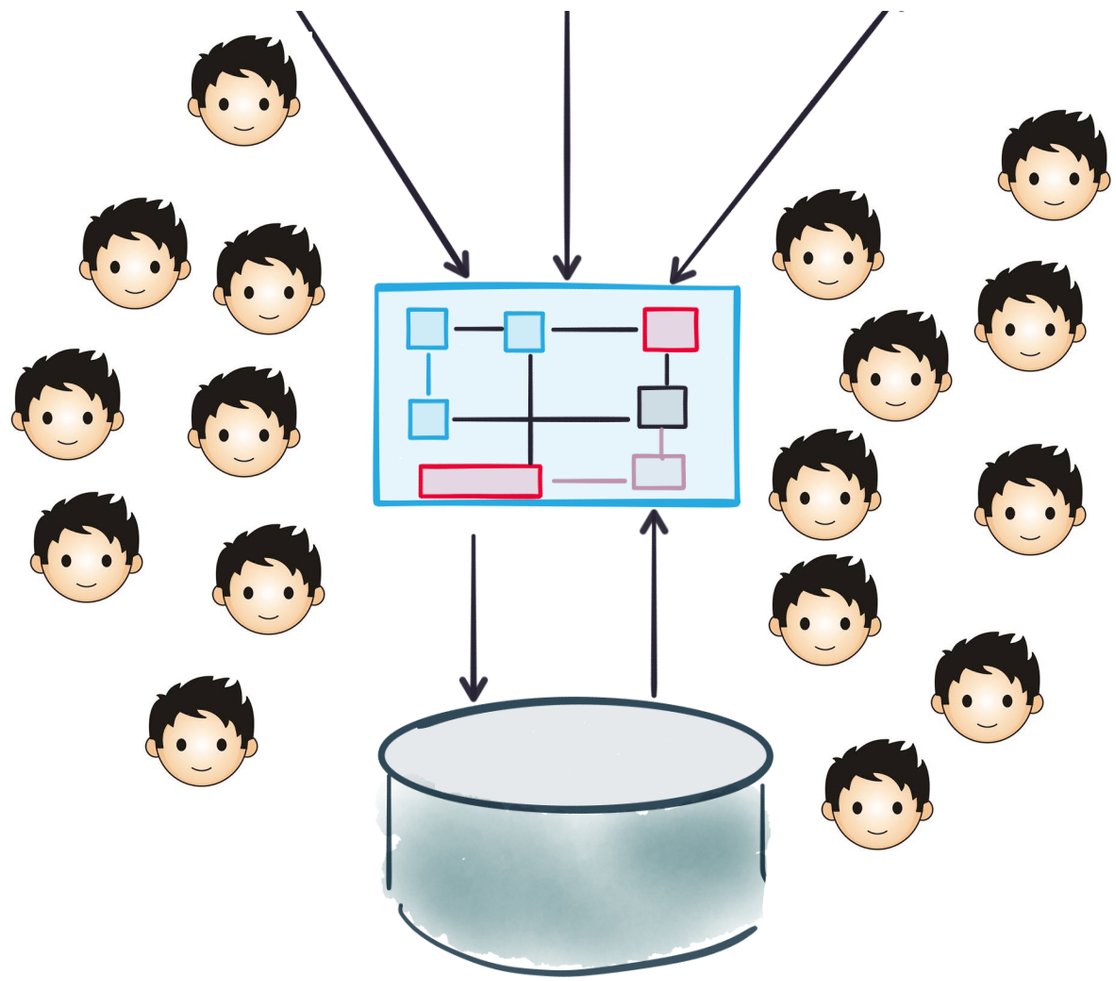
Independence is where services get their value

confluent

Allows Scaling

confluent

Scaling in terms of people

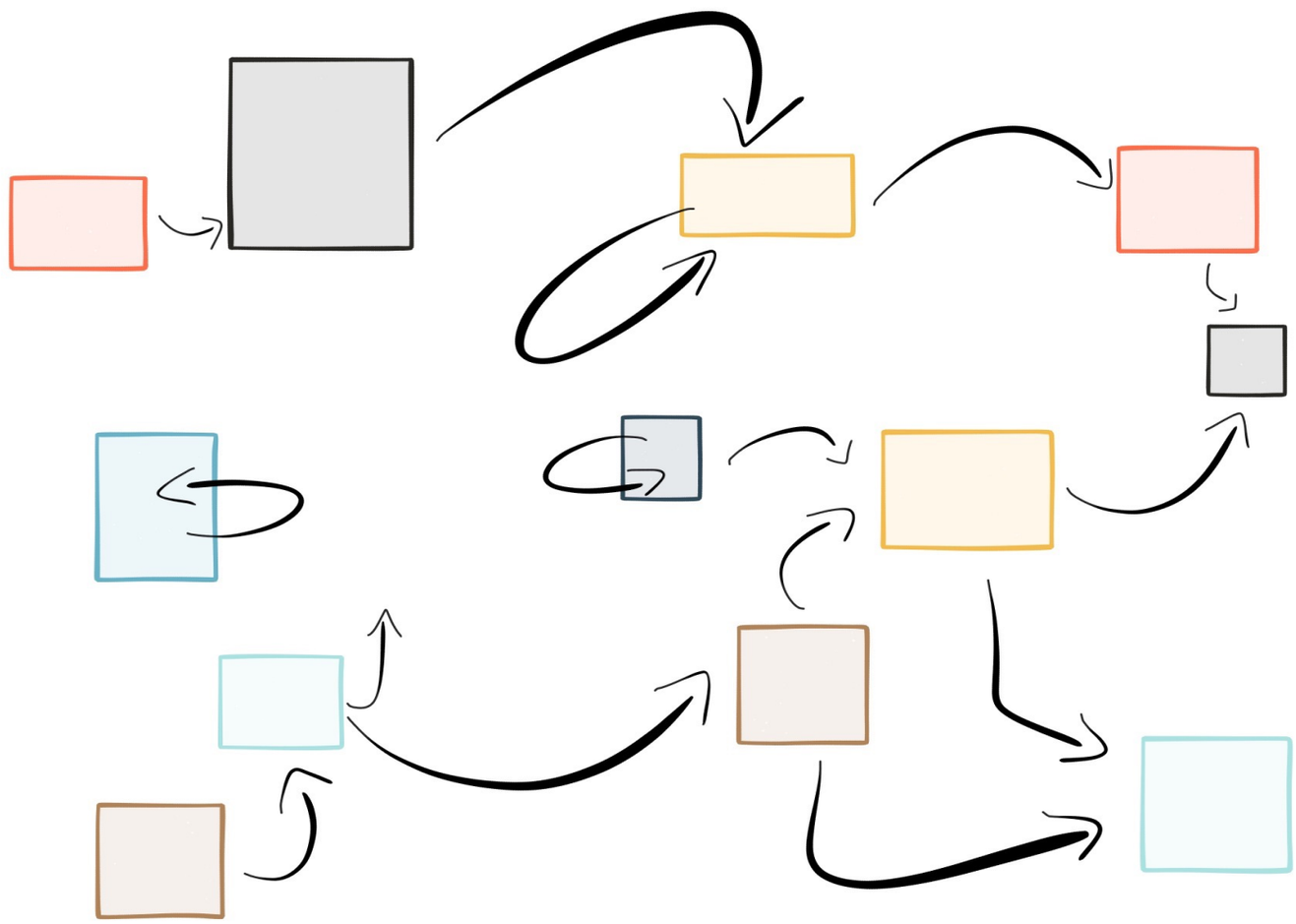


What happens when we grow?

confluent

Companies are inevitably a collection of applications

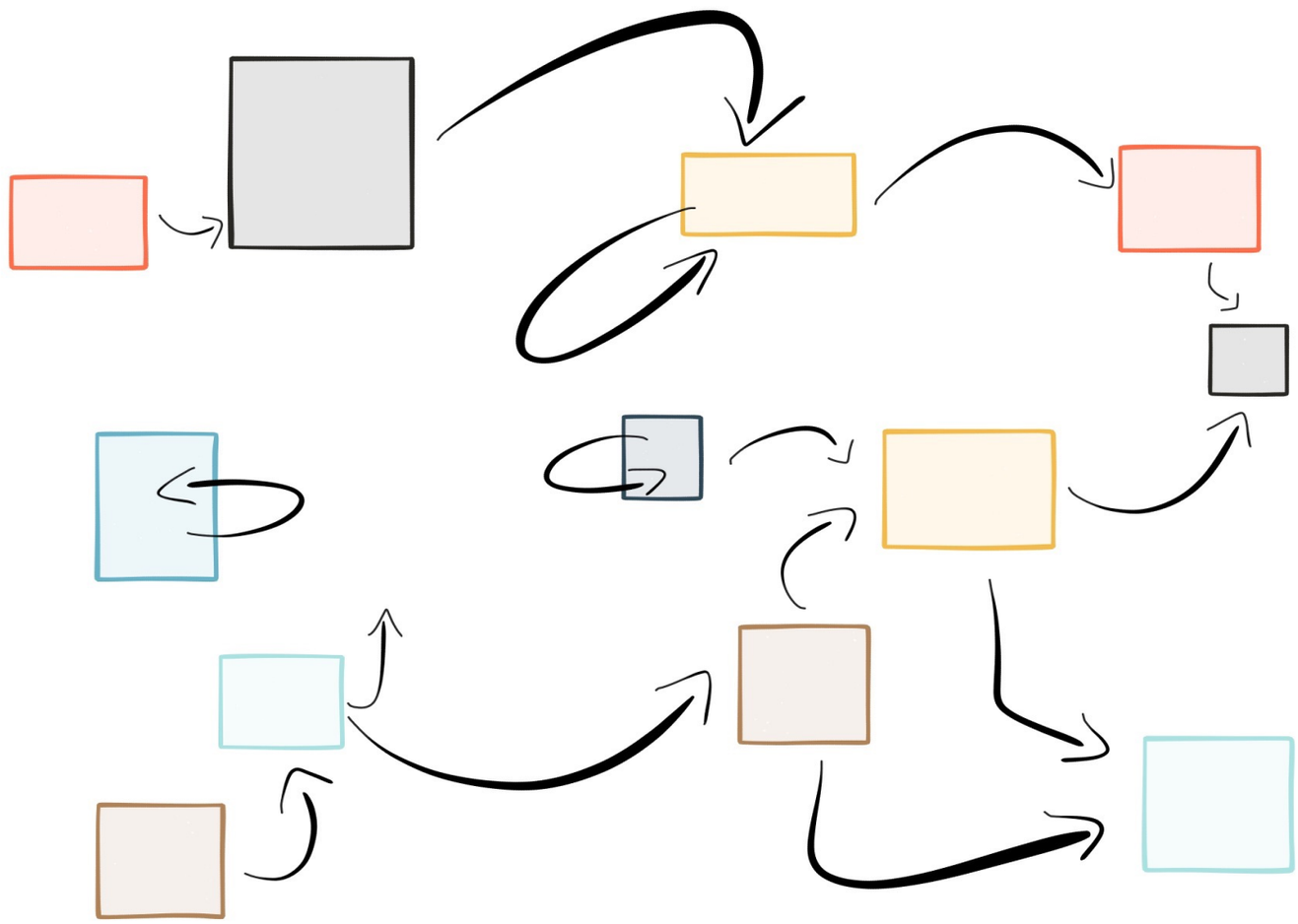
They must work together to some degree



confluent

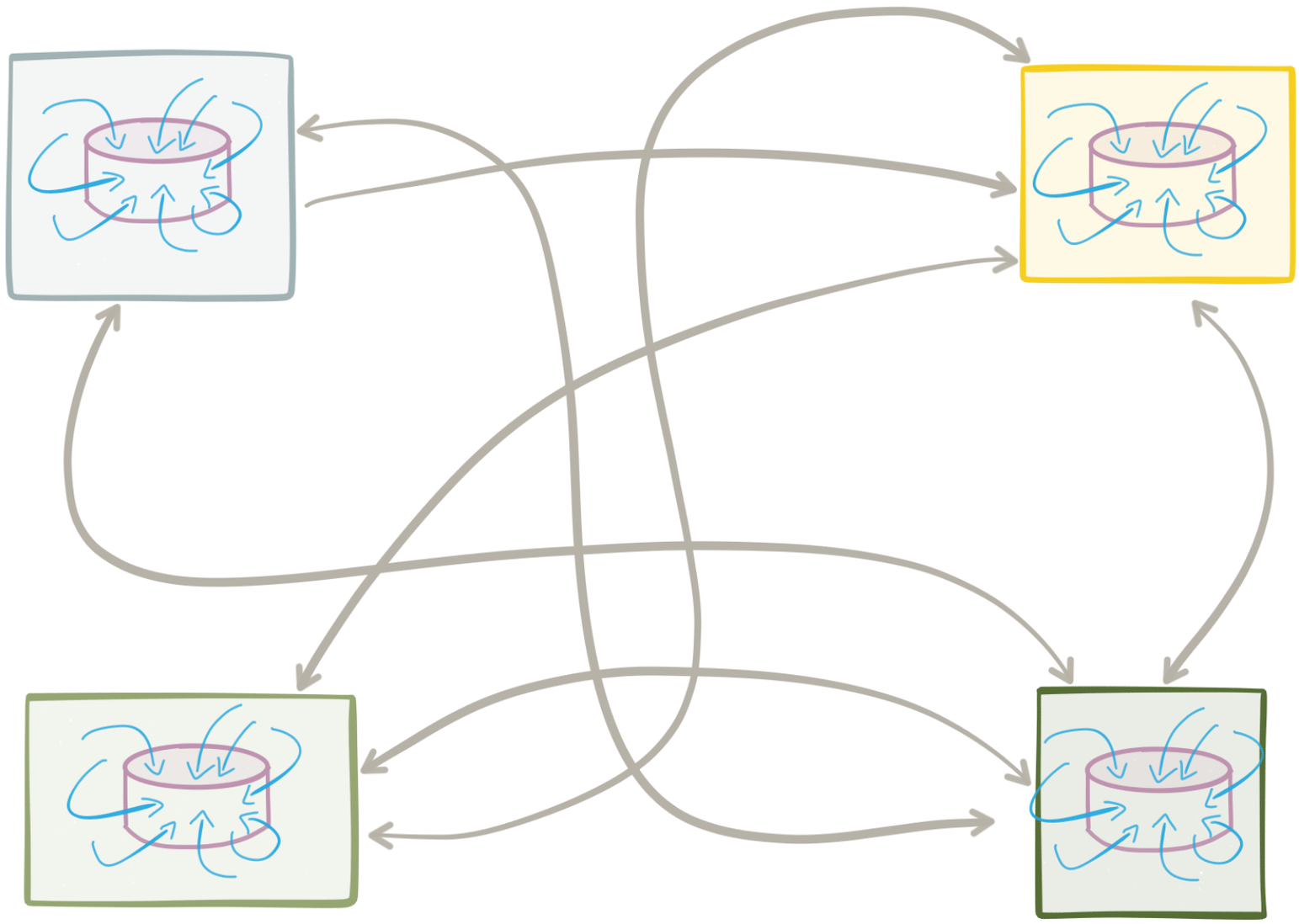
Interconnection is an afterthought

FTP / Enterprise Messaging etc



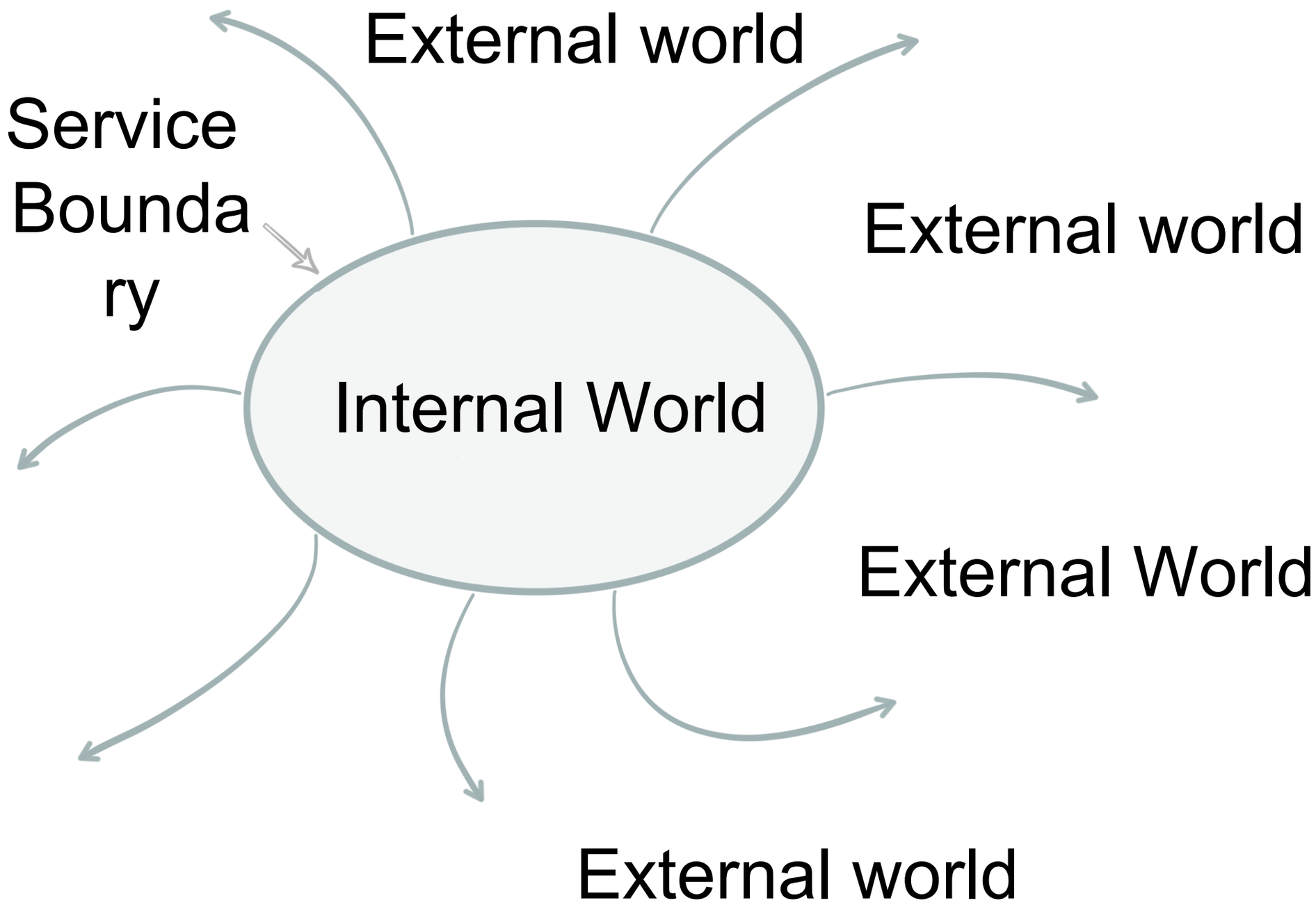
confluent

Microservices / SOA are patterns for multi-team architectures



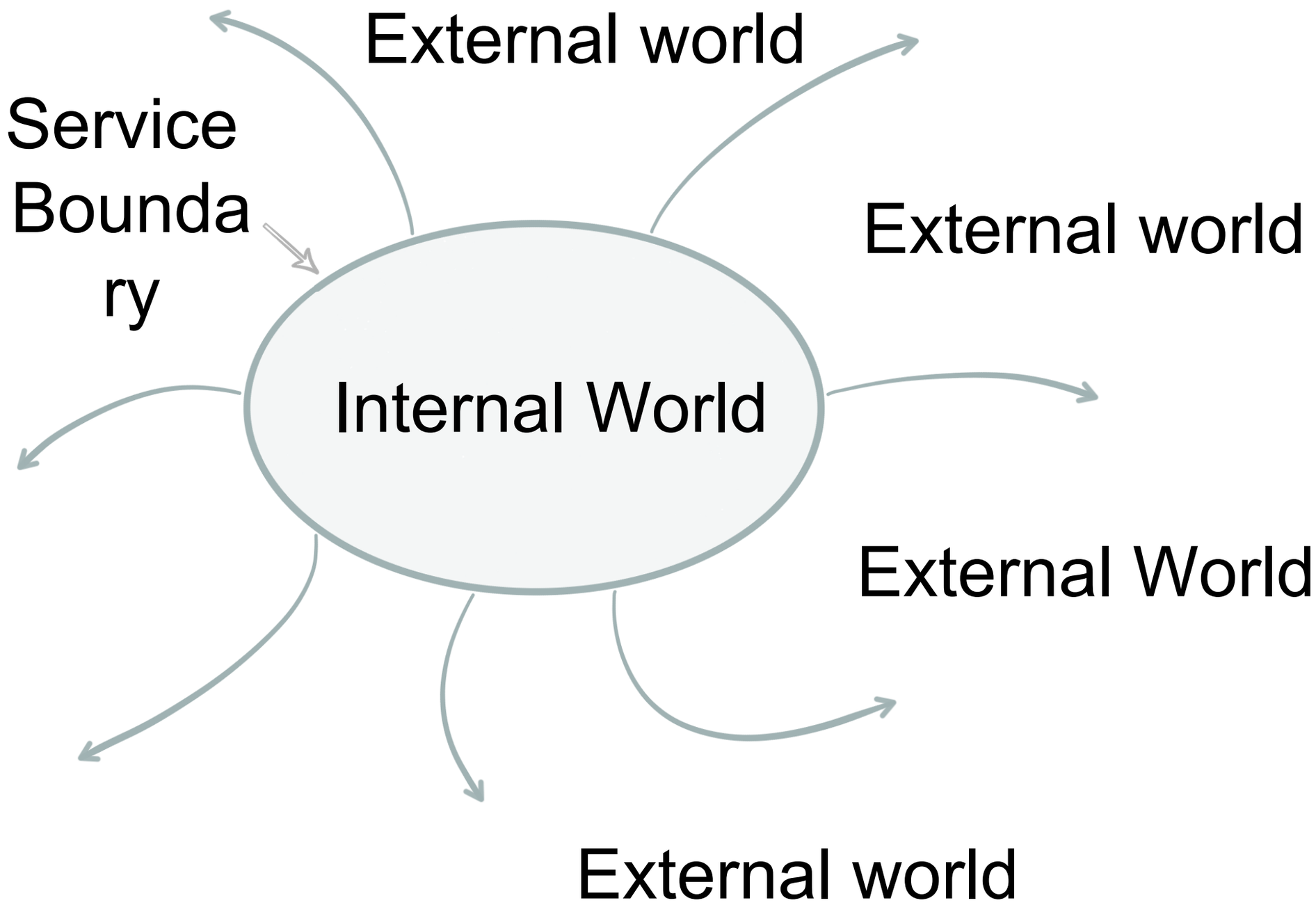
confluent

Services Force Us To Consider The External World



confluent

External World is something we should Design For

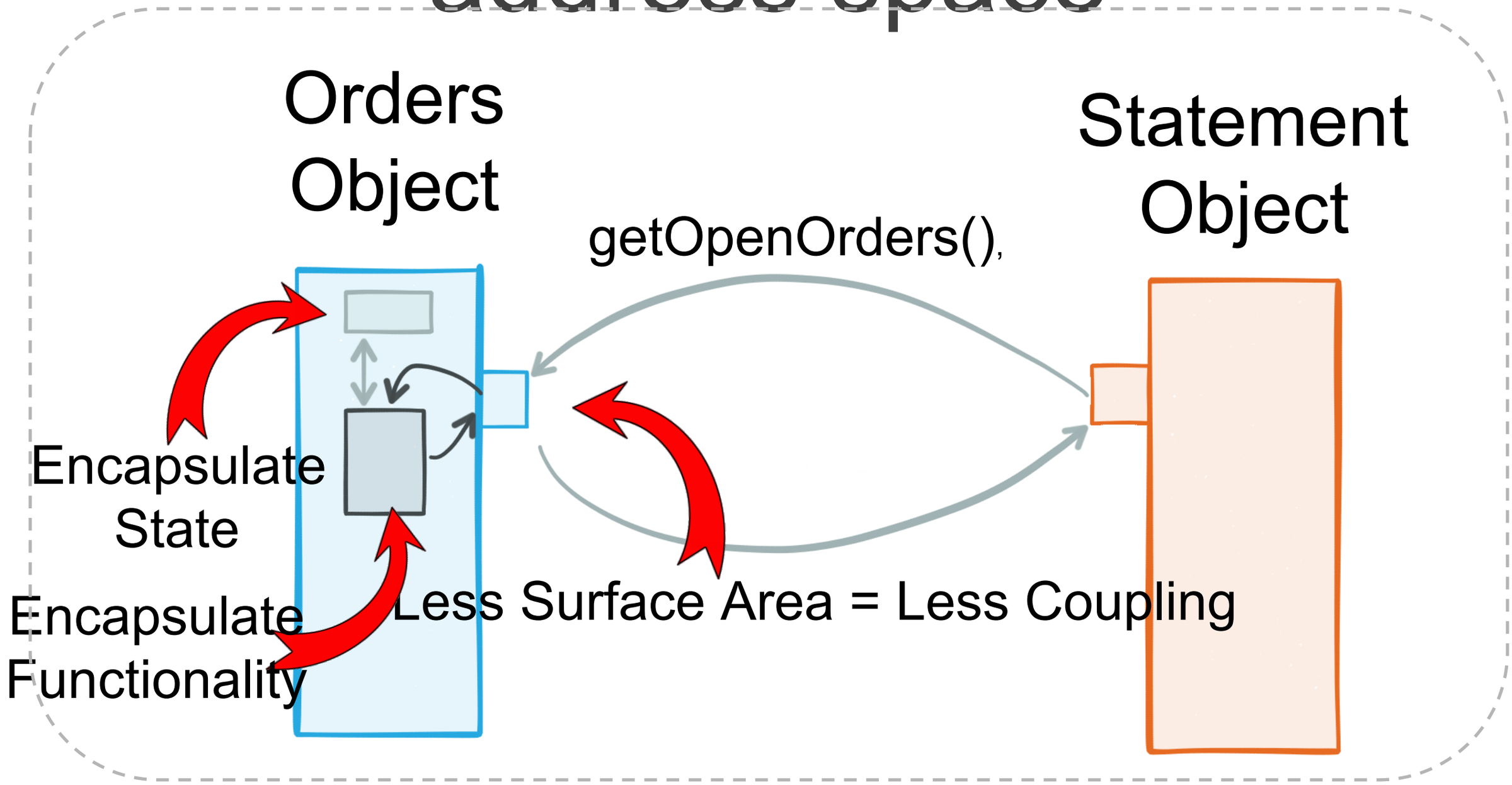


confluent

Independence comes at a cost $$$

confluent

Consider Two Objects in one address space



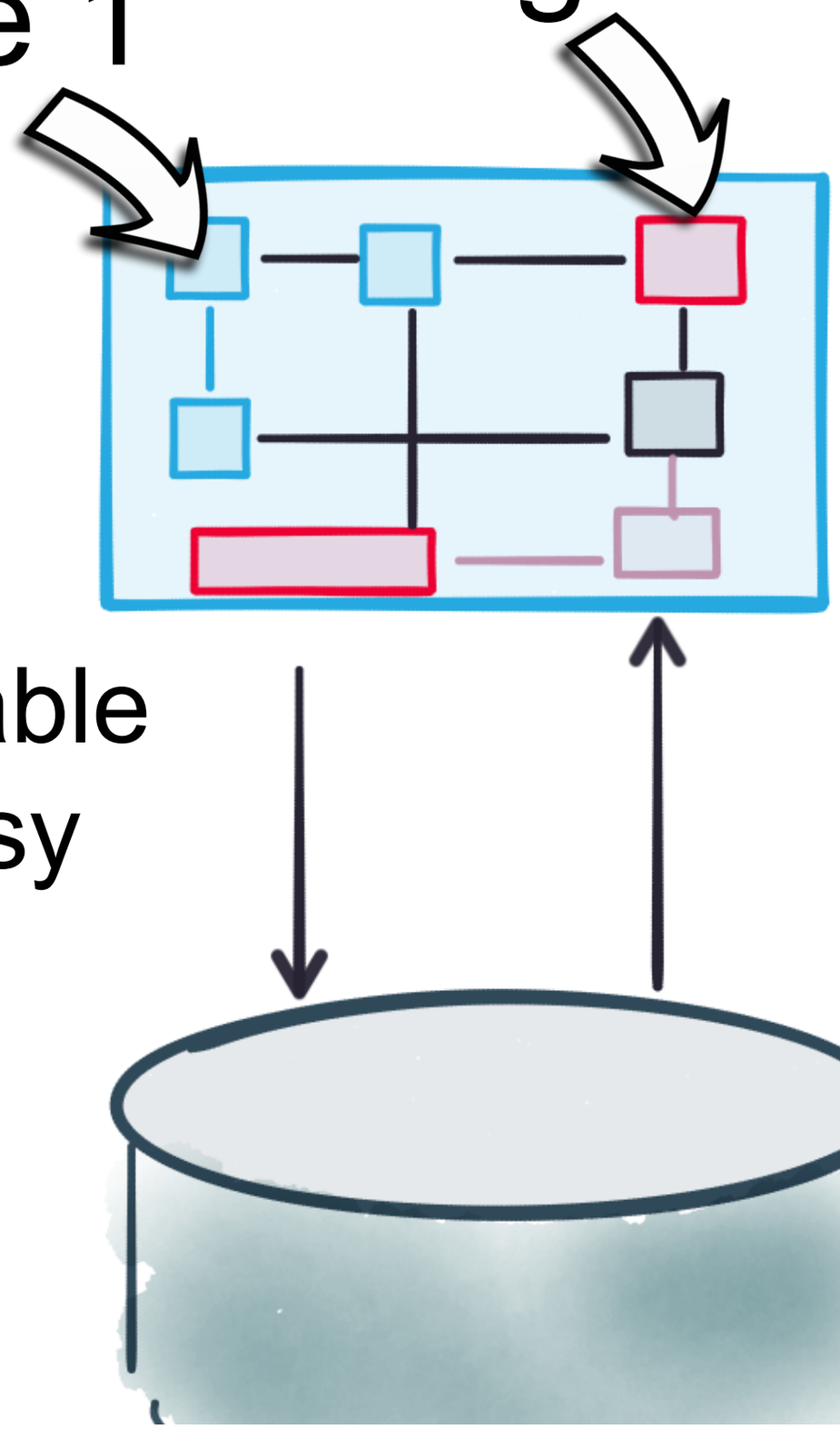
Encapsulation => Loose Coupling

confluent

Change 1

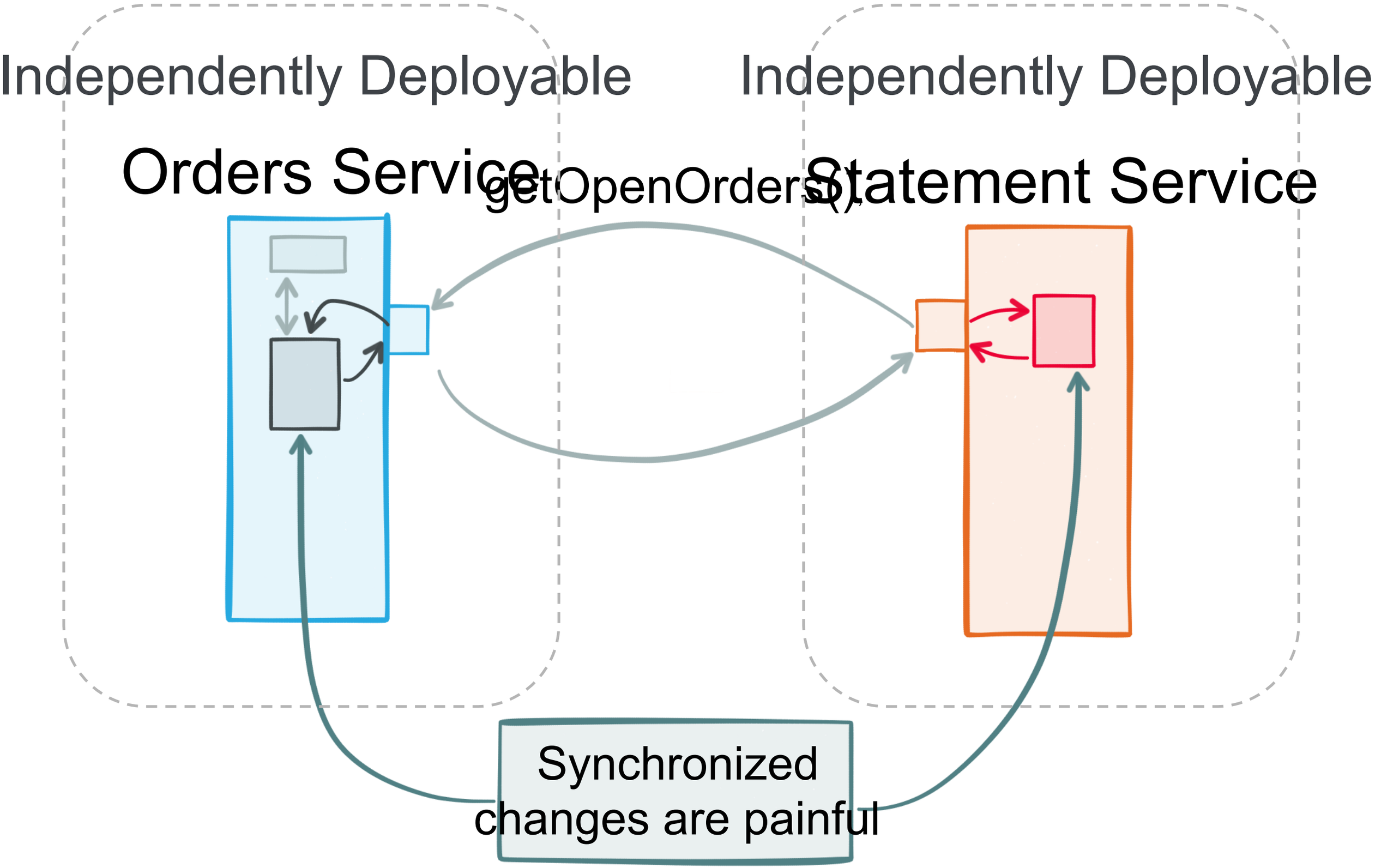
Change 2

Redeploy



Singly-deployable apps are easy

confluent

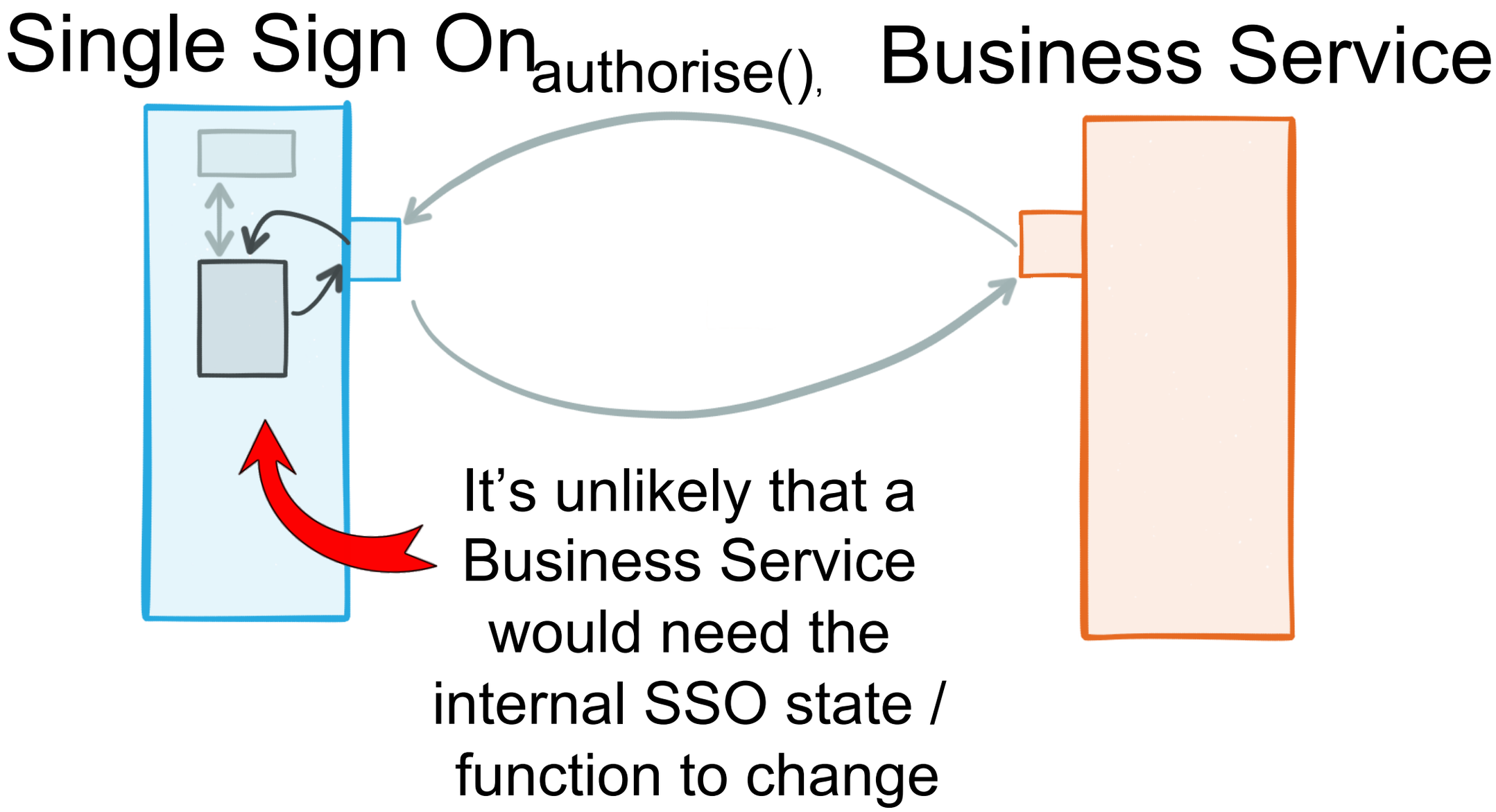

Independently Deployable
Independently Deployable
Orders Service
getOpenOrders()
Statement Service
Synchronized
changes are painful

confluent

Services work best where requirements are isolated in a single bounded context

confluent

Single Sign On



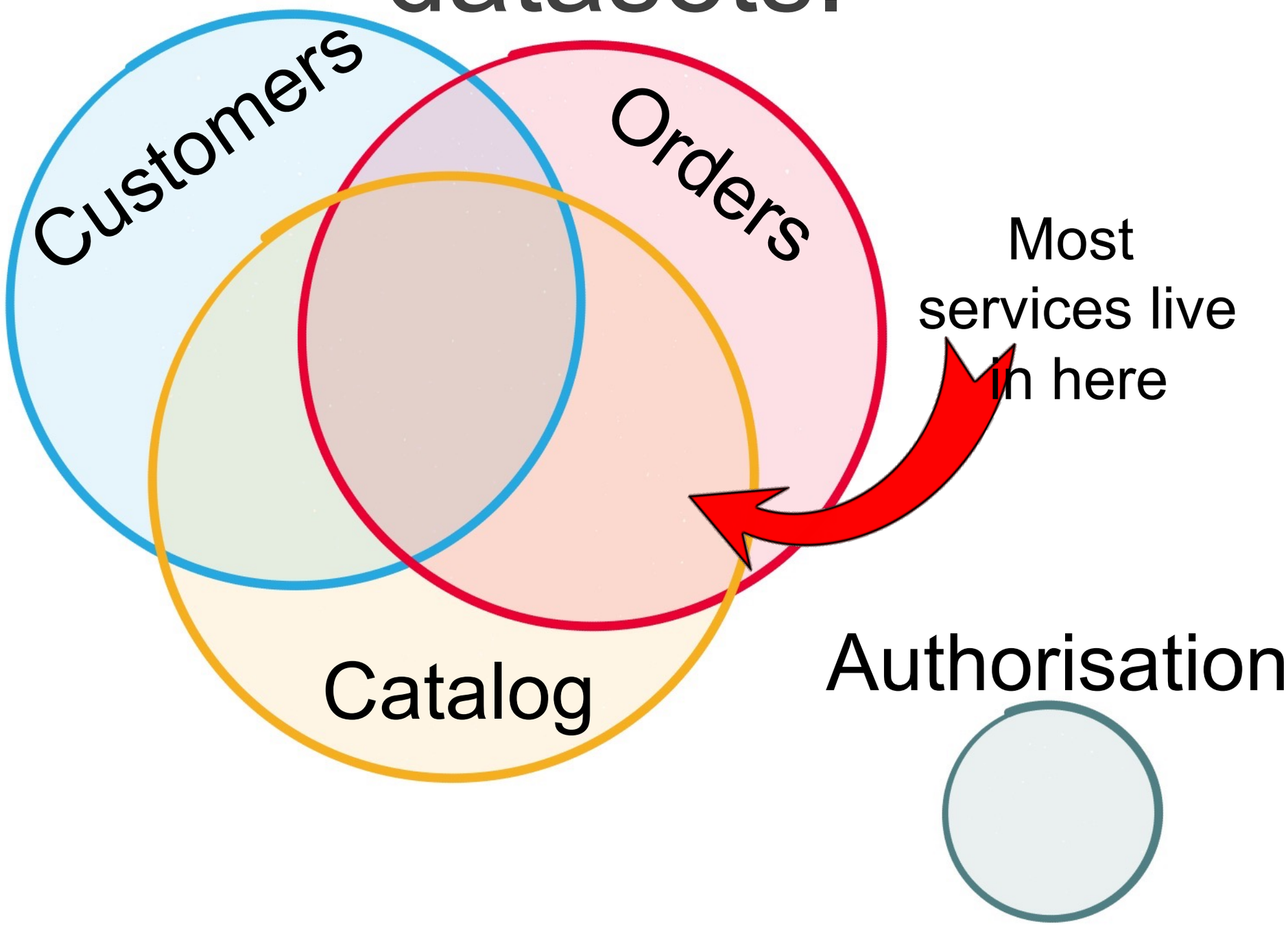
SSO has a tightly bounded context

confluent

But business services are different

confluent

Most business services share the same core datasets.



confluent

The futures of business services are far more tightly intertwined

confluent

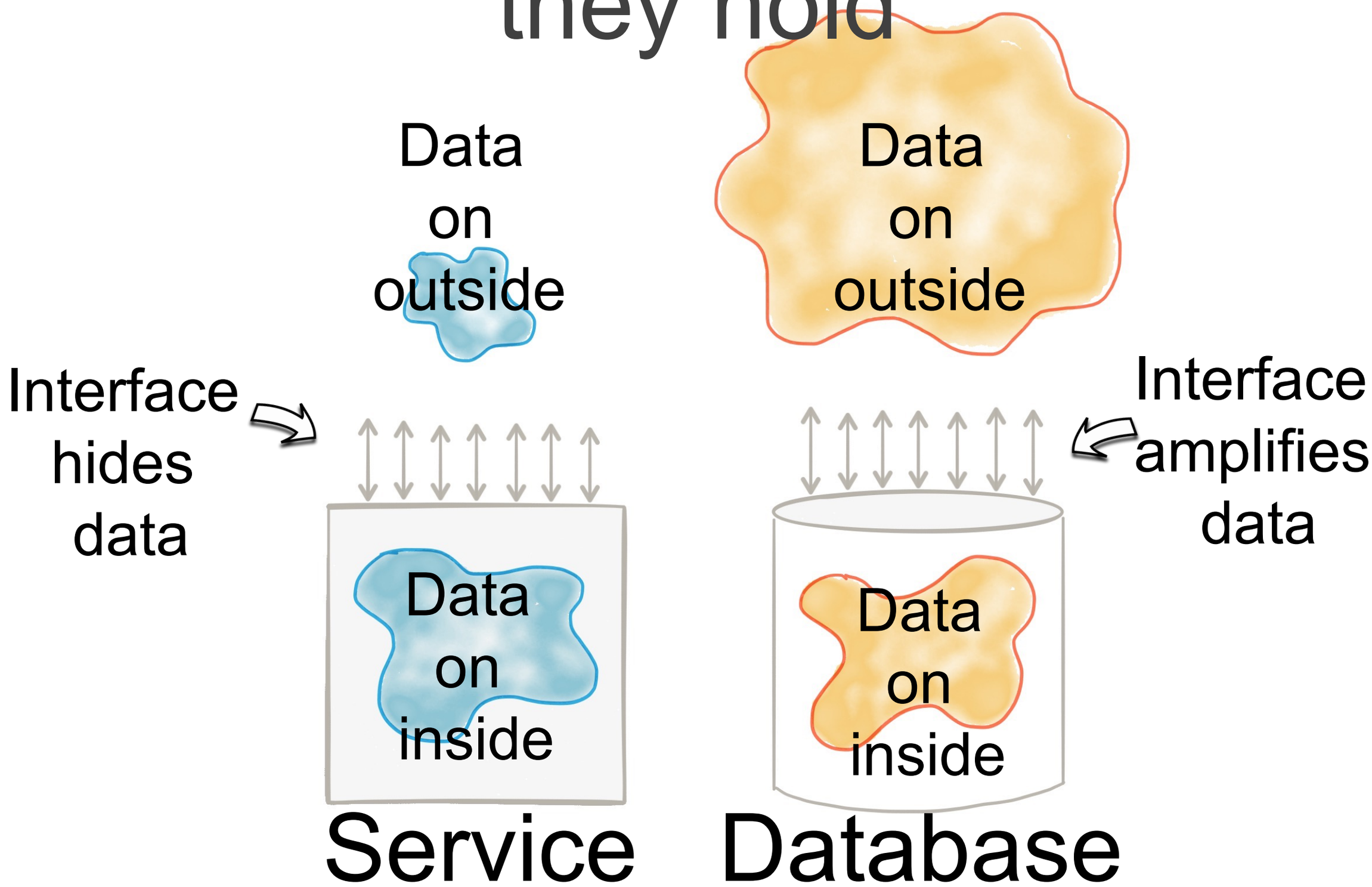
We need encapsulation to hide internal state. Be loosely coupled.

But we need the freedom to slice & dice shared data like any other dataset

But data systems have little to do with encapsulation

confluent

Databases amplify the data they hold



confluent

The data dichotomy

ata systems are about exposing da

Services are about hiding it.

confluent

Microservices shouldn’t share a database

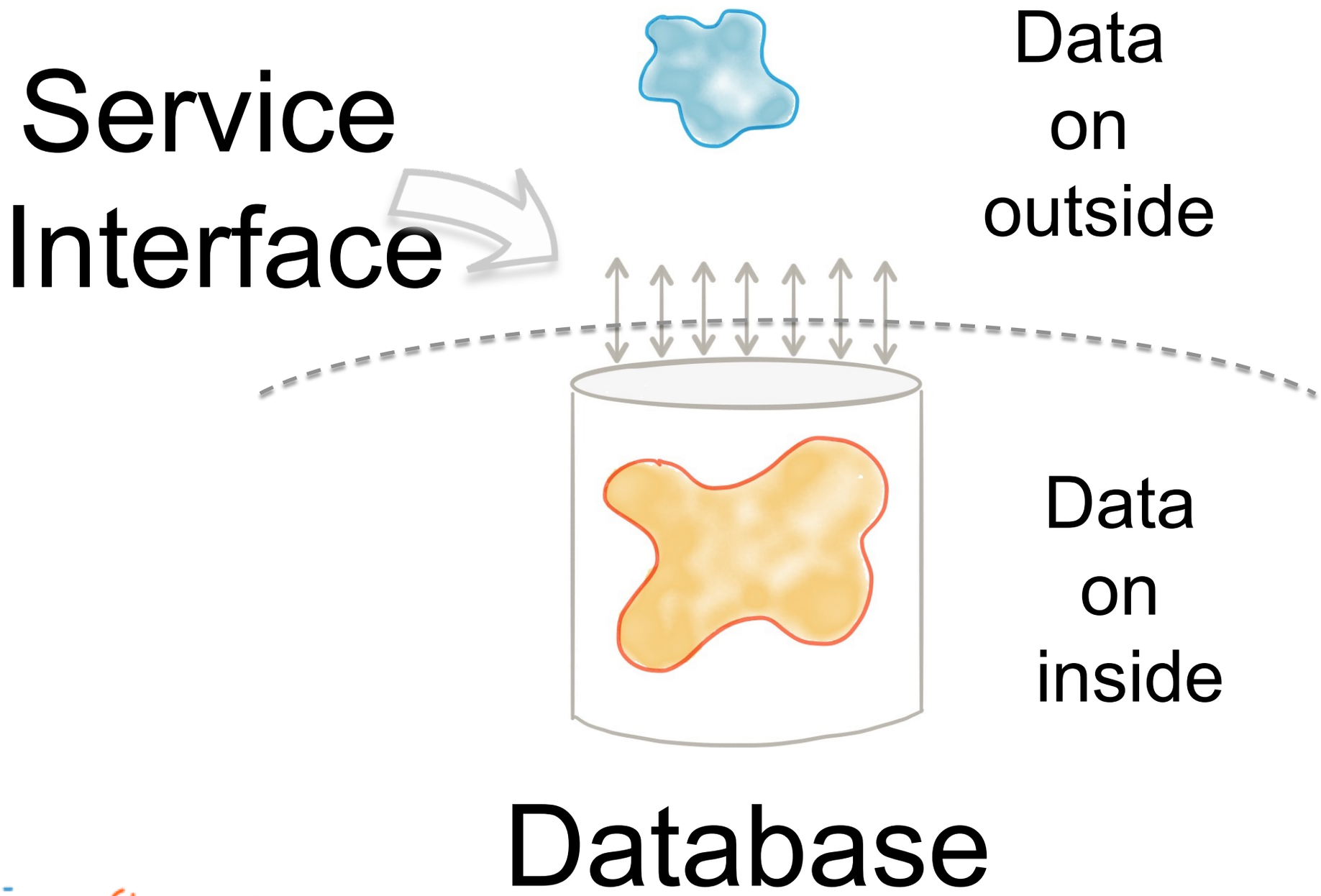
Good Advice!

confluent

So what do we do instead?

confluent

We wrap a database in a service interface



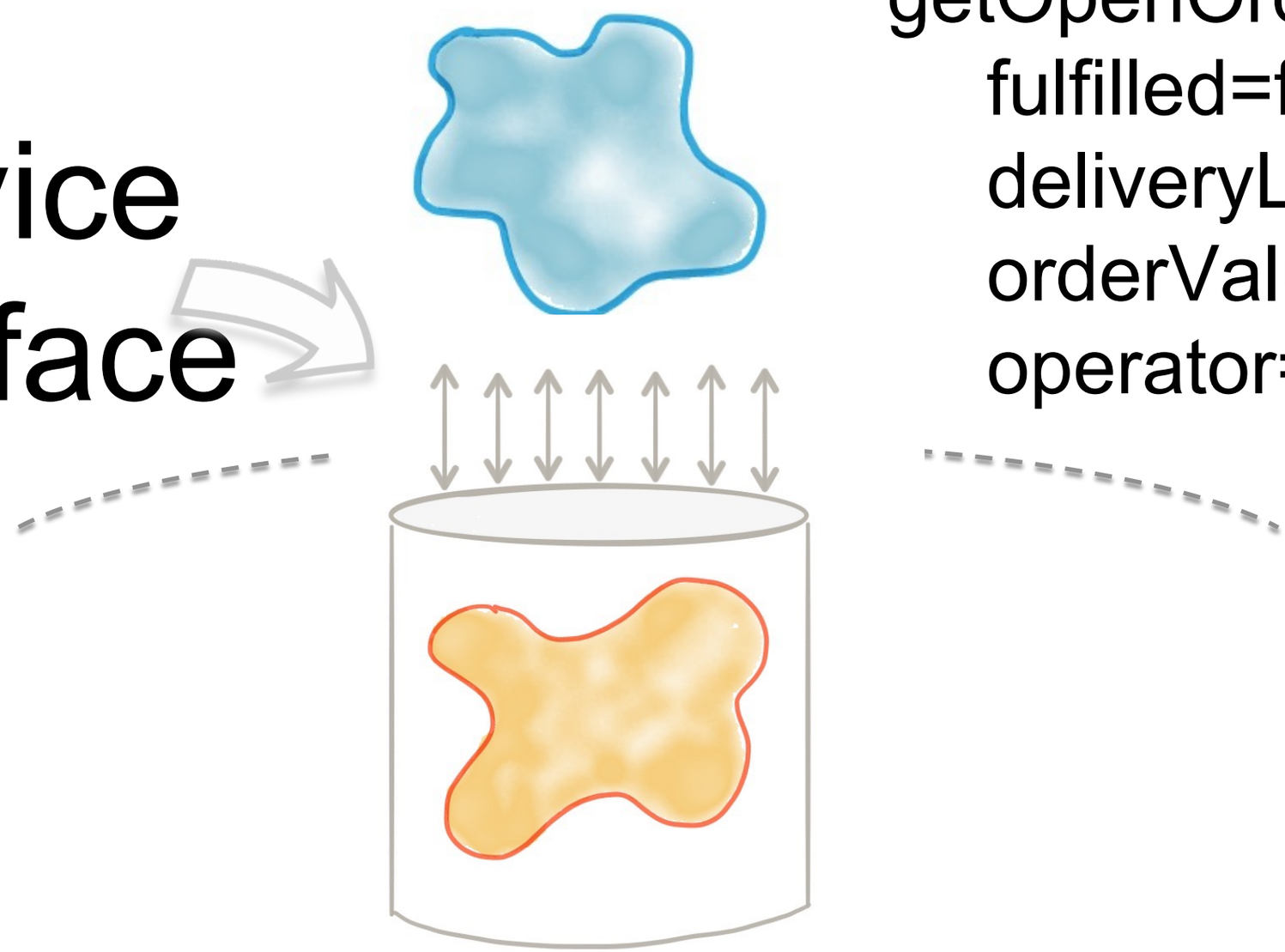
confluent

One of two things happens next

confluent

Either (1) we constantly add to the interface, as datasets grow

Service Interface

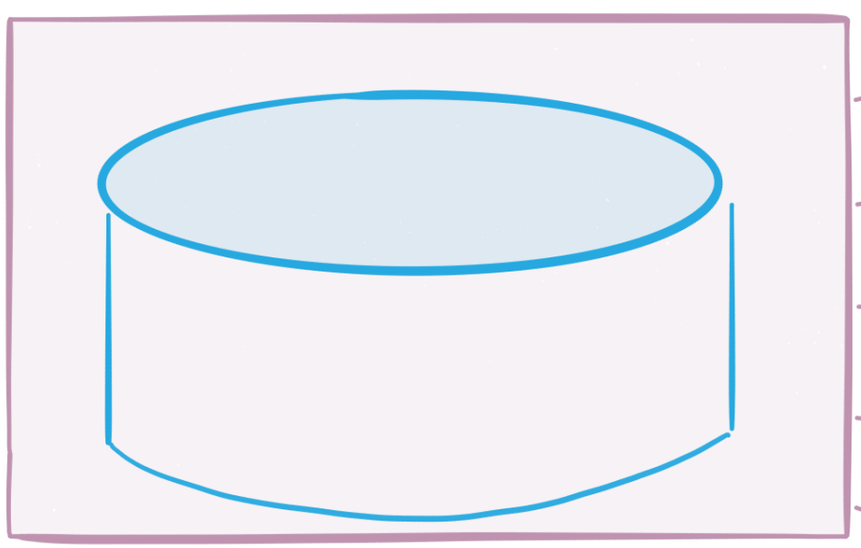


```
getOpenOrders(
    fulfilled=false,
    deliveryLocation=CA,
    orderValue=100,
    operator=GreatherThan)
```

Database

confluent

(1) Services can end up looking like kookie home-grown databases



getOrder(Id)

getOrder(UserId)

getAllOpenOrders()

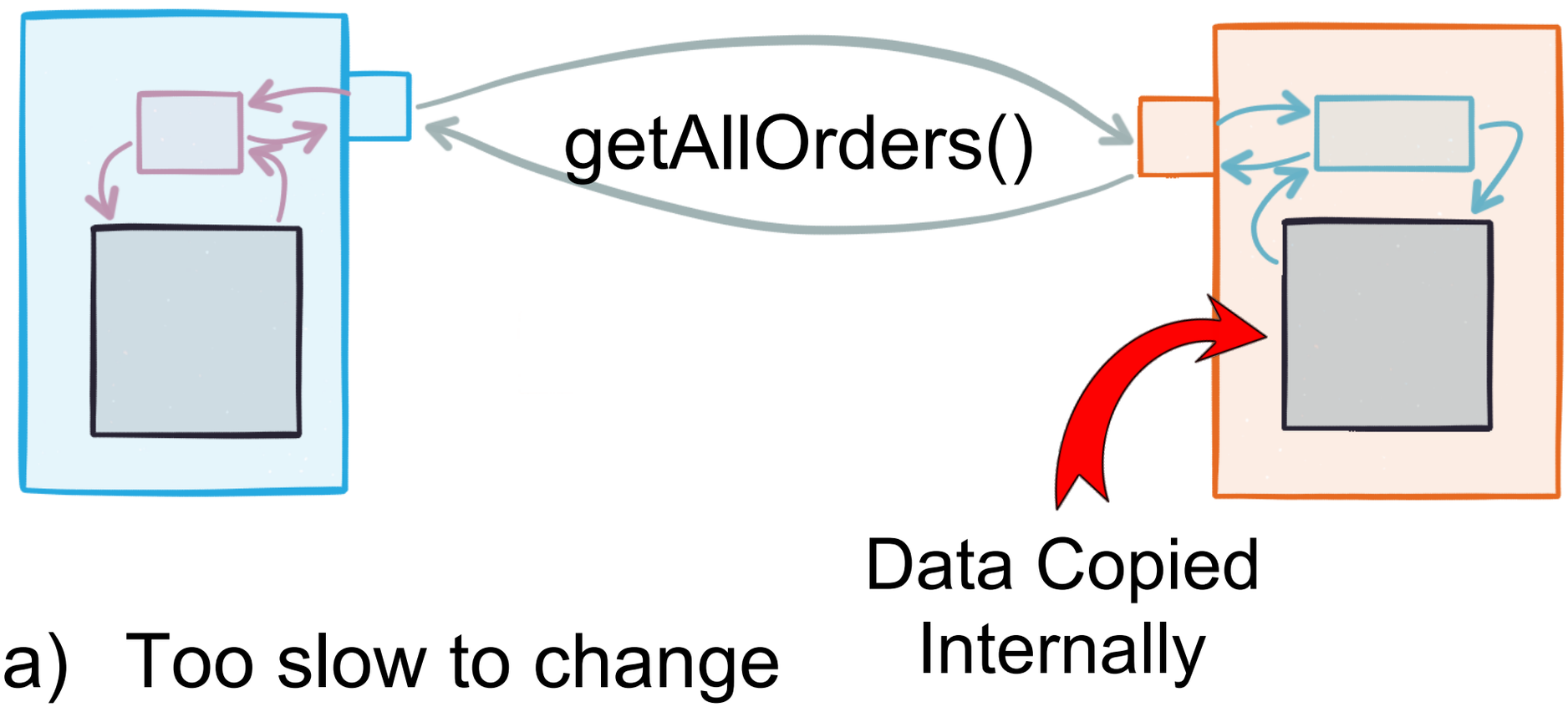
getAllOrdersUnfulfilled(Id)

getAllOrders()

...and DATA amplifies this "Data-Service" problem

confluent

(2) Give up and move whole datasets en masse

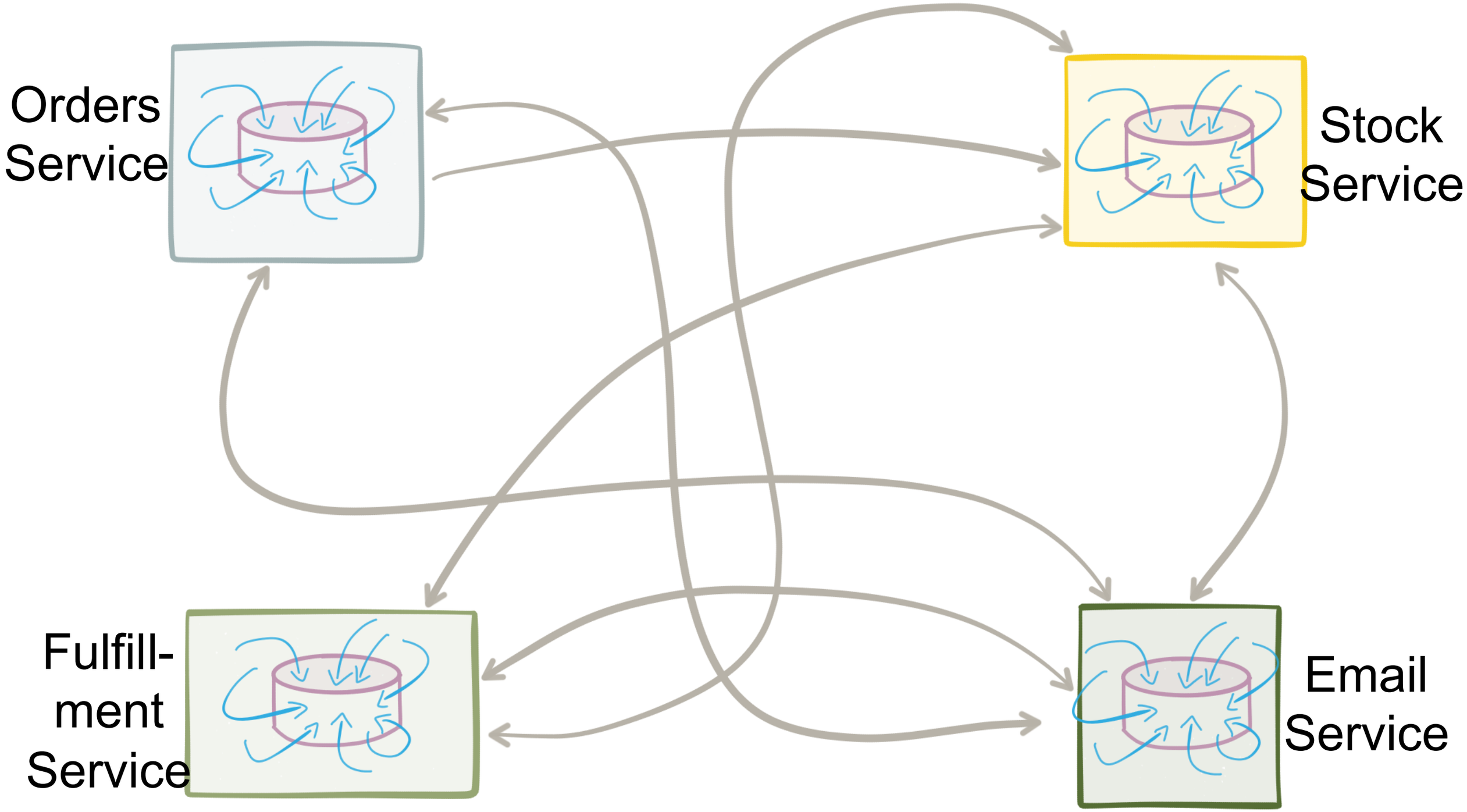


a) Too slow to change
b) To perform joins

confluent

This leads to a different problem

Data diverges over time

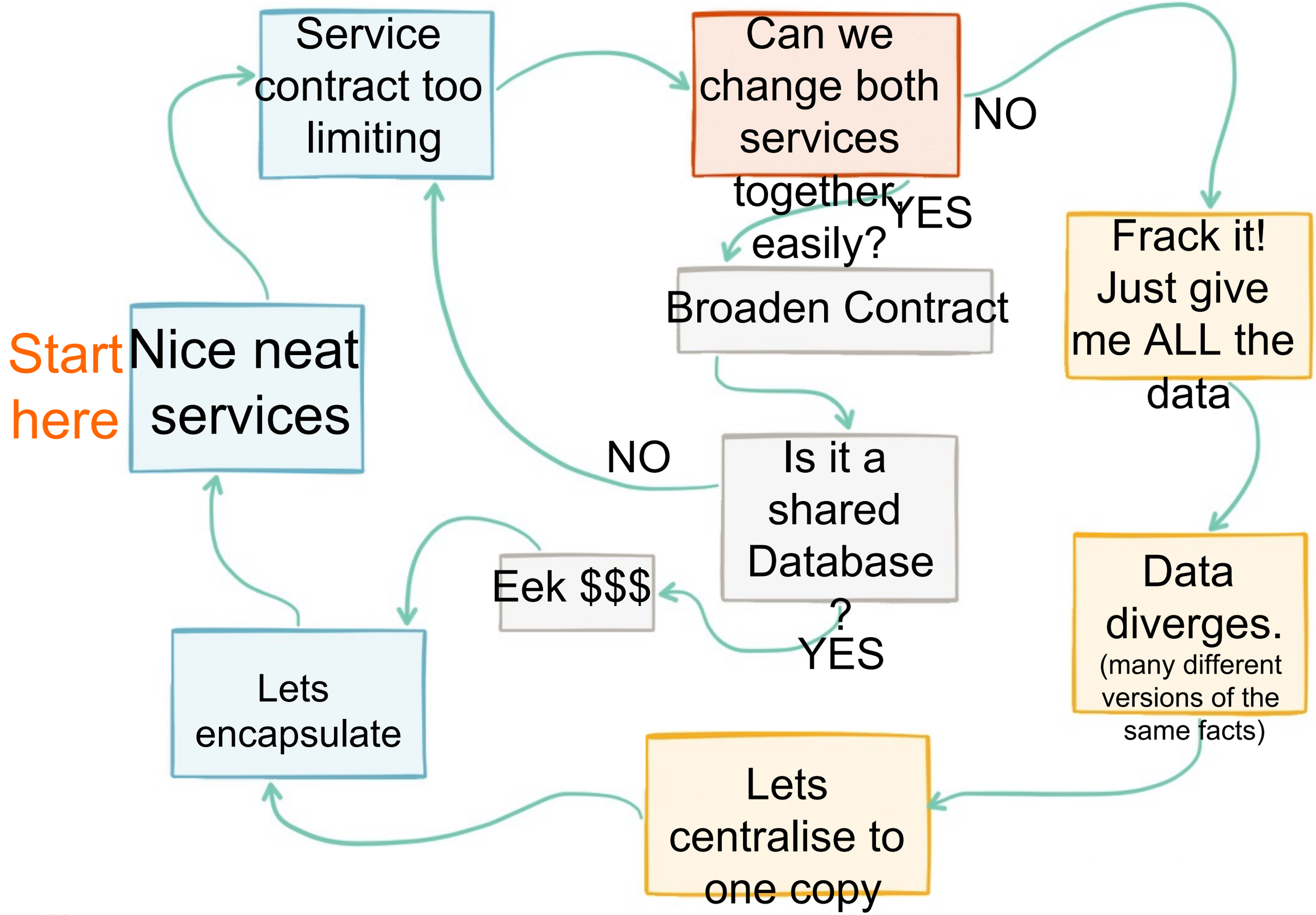


confluent

The more mutable copies, the more data will diverge over time

confluent

ycle of inadequacy:



confluent

These forces compete in the systems we build

Accessibility

VS

Coupling

Divergence

confluent

Is there a better way?

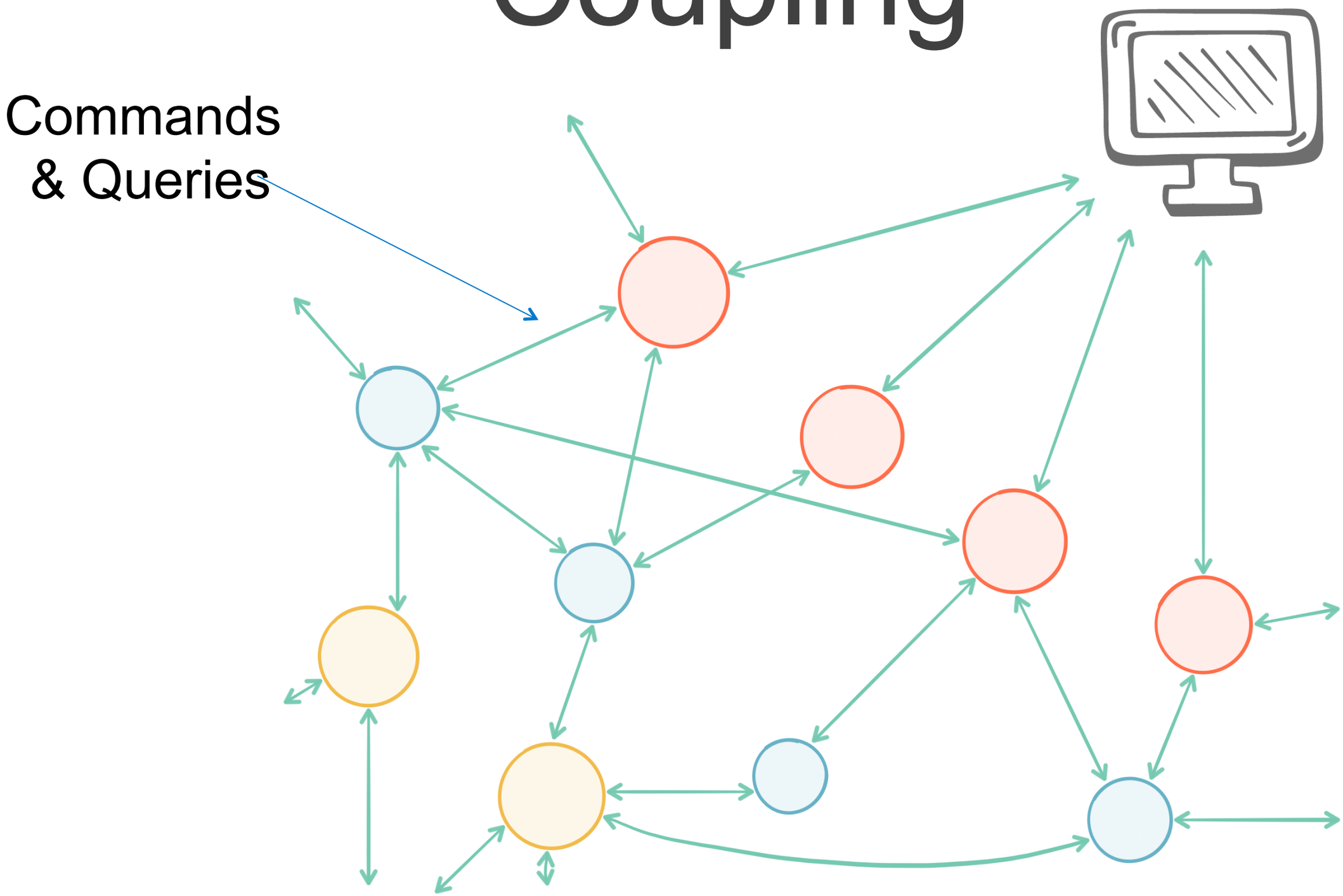
Step 1:

Build on a backbone of events.

Layer in requests where necessary.

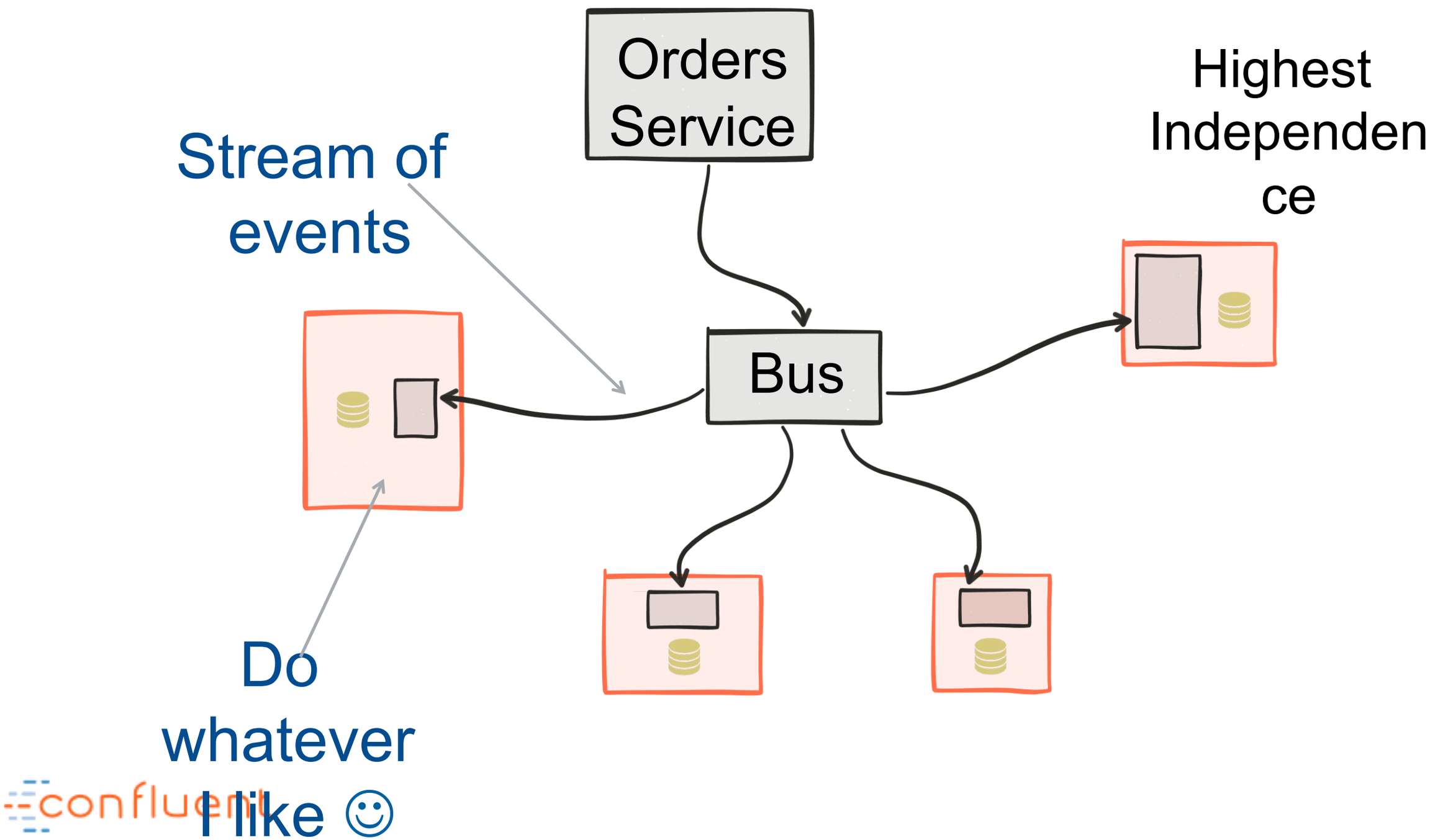
confluent

Request Driven -> highest Coupling

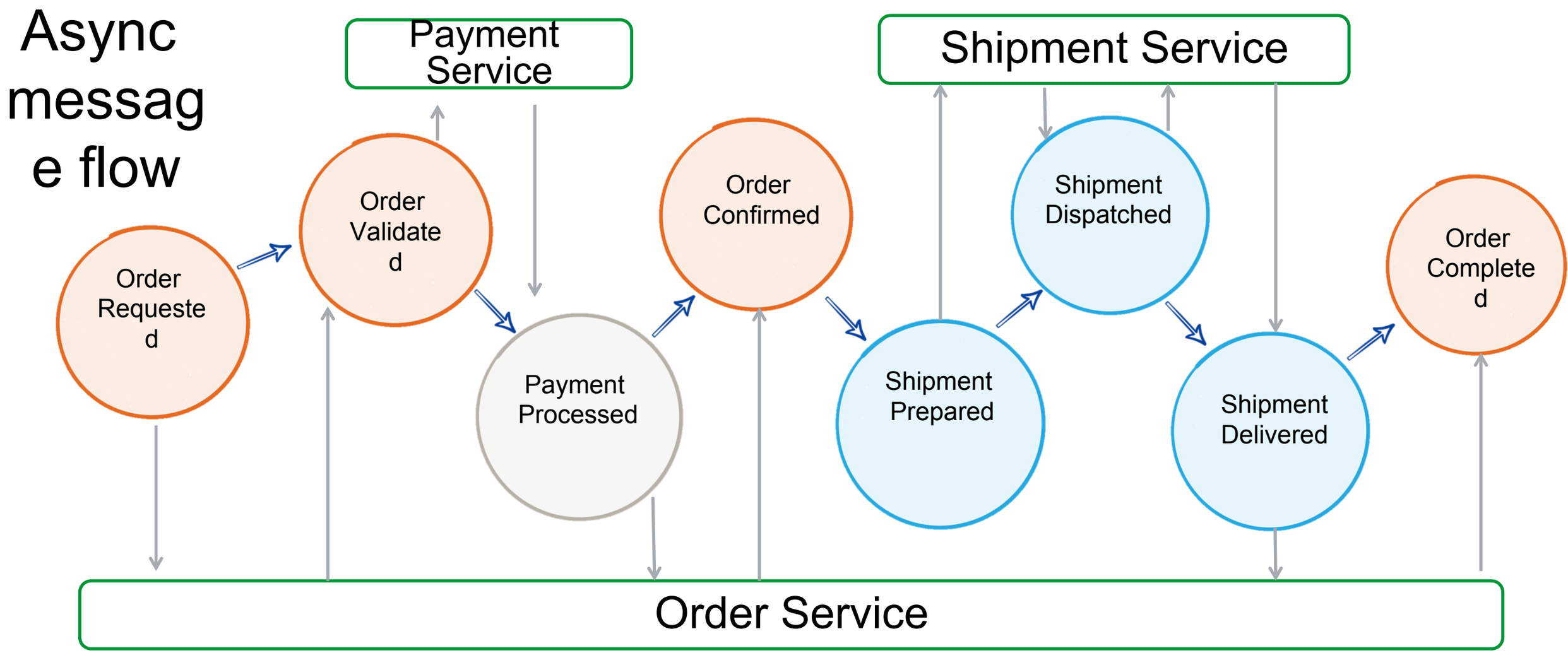


confluent

Event Broadcast => lowest coupling

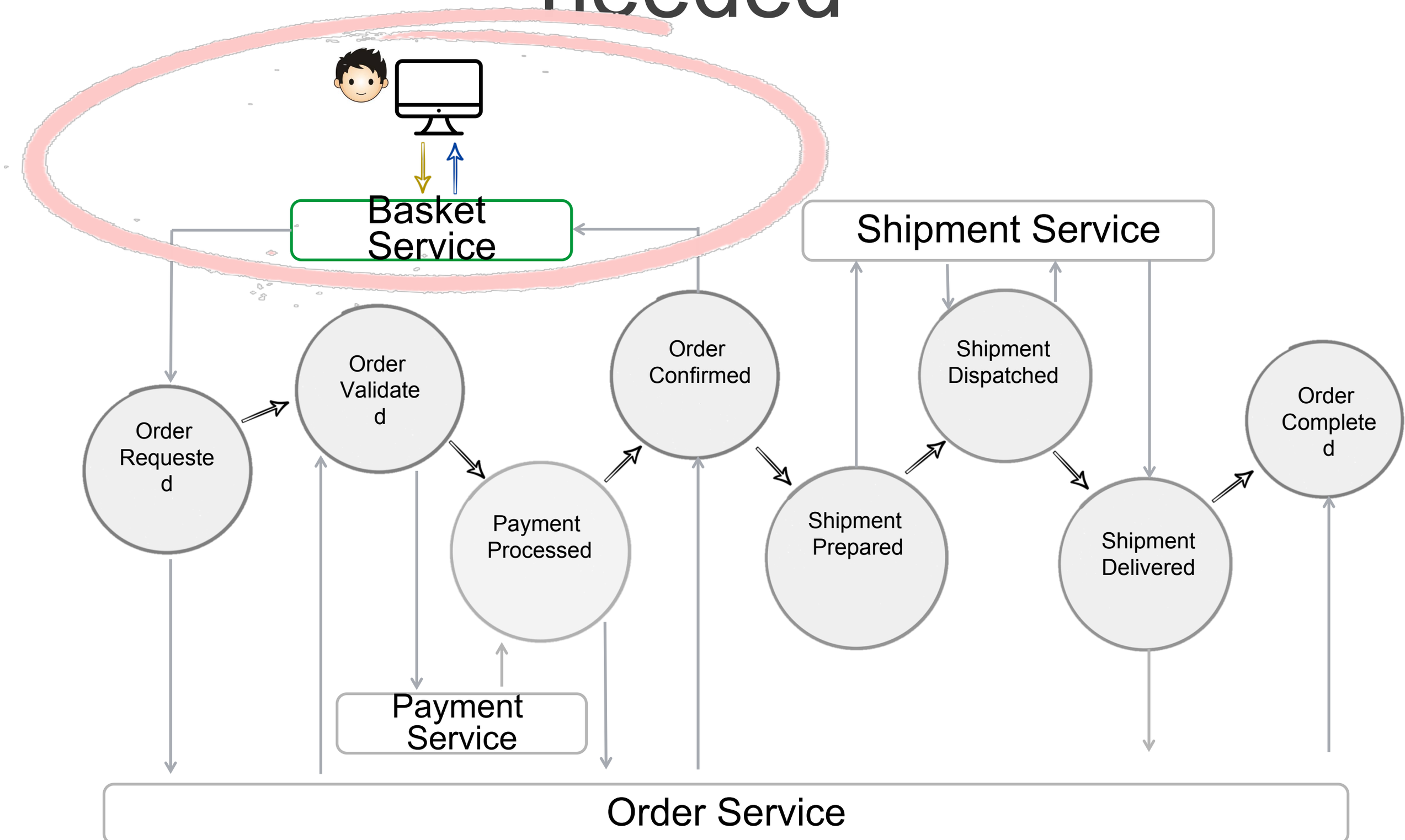


Event Driven Services Chain (Event Collaboration)



confluent

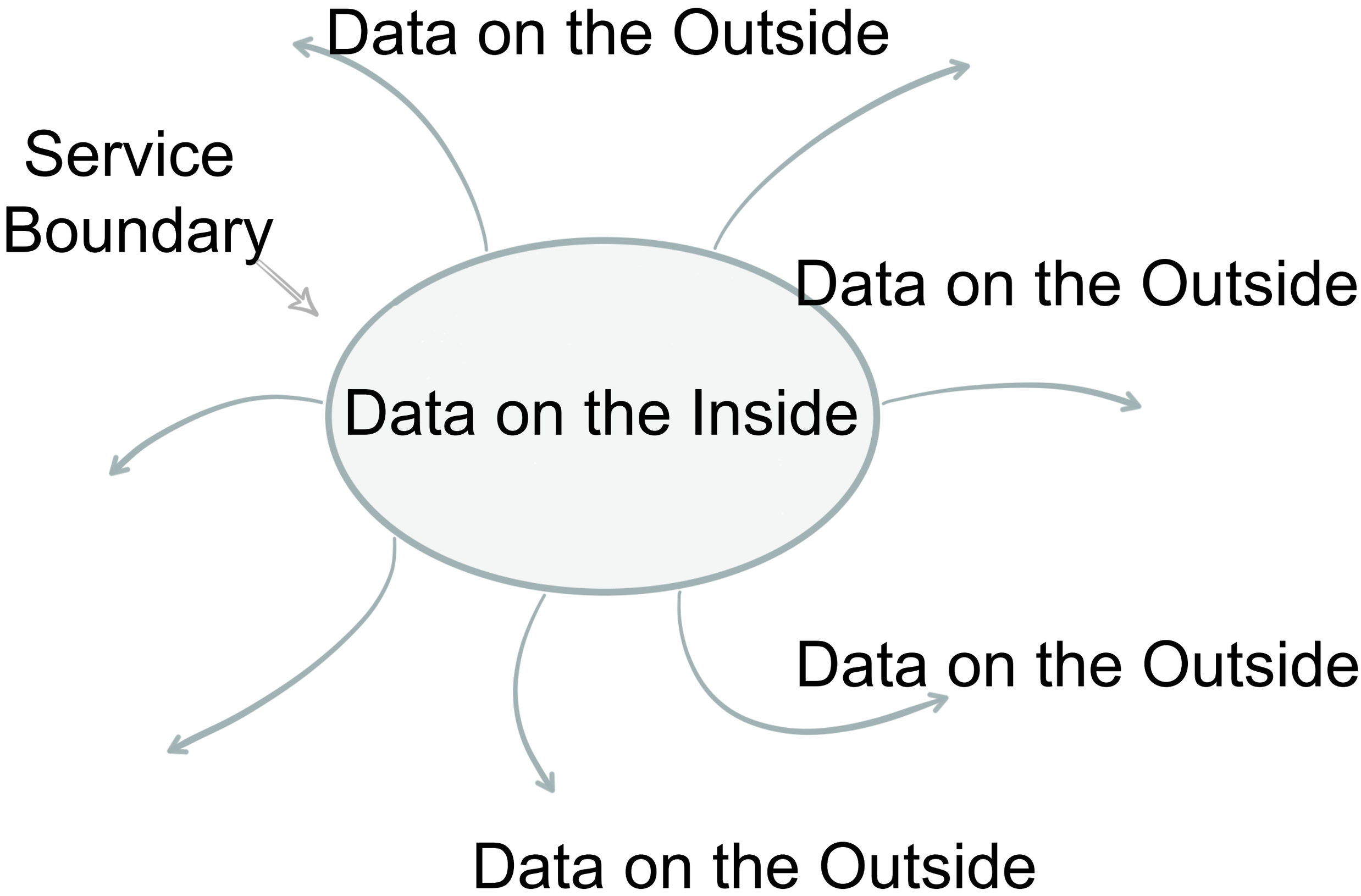
Layer queries/commands where needed



confluent

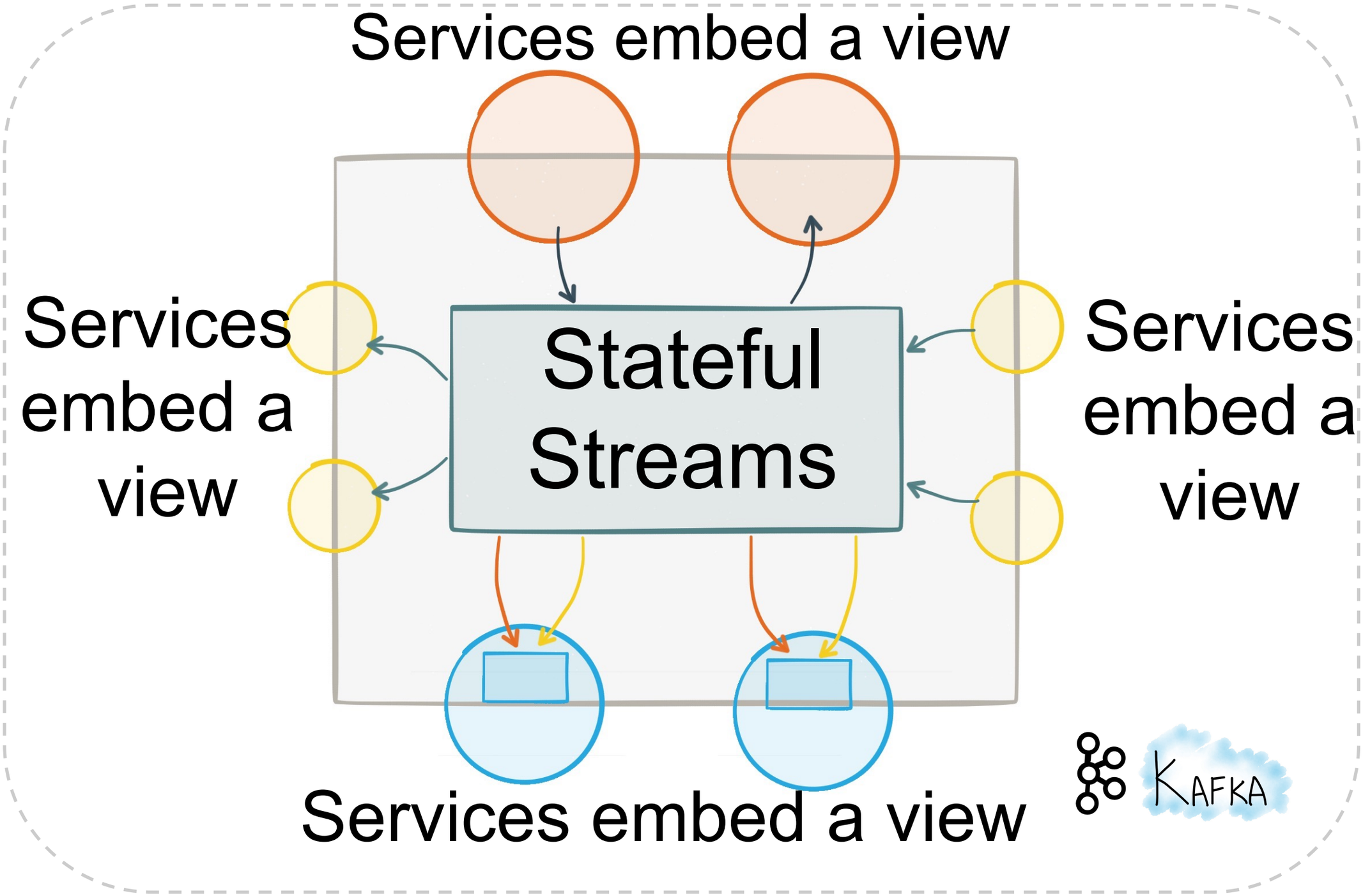
Step 2:
Make data-on-the-outside a 1st class citizen

confluent


Data on the Outside
Service Boundary
Data on the Outside
Data on the Inside
Data on the Outside
Data on the Outside

confluent

Stateful Streams

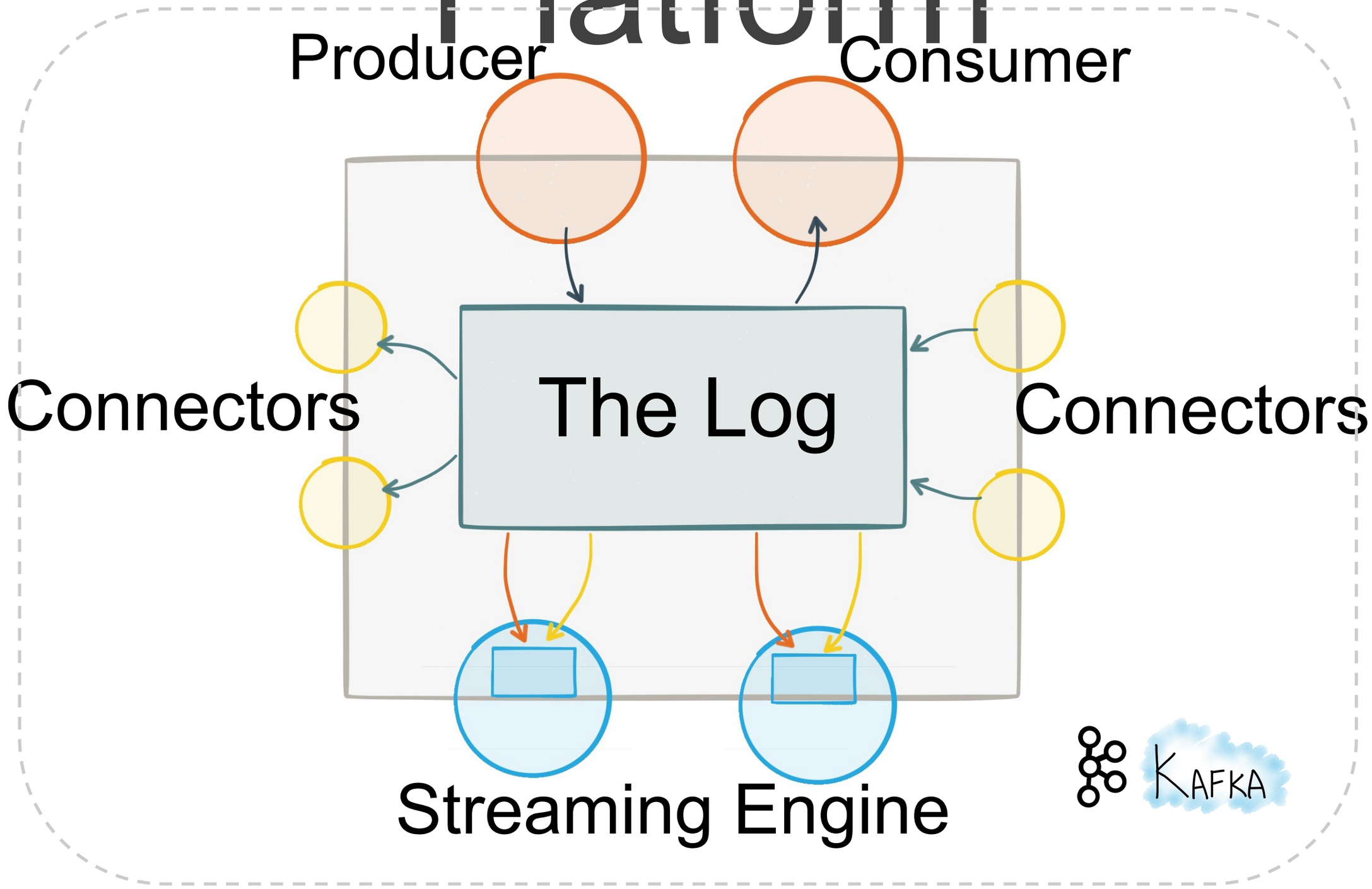


confluent

Kafka helps with this

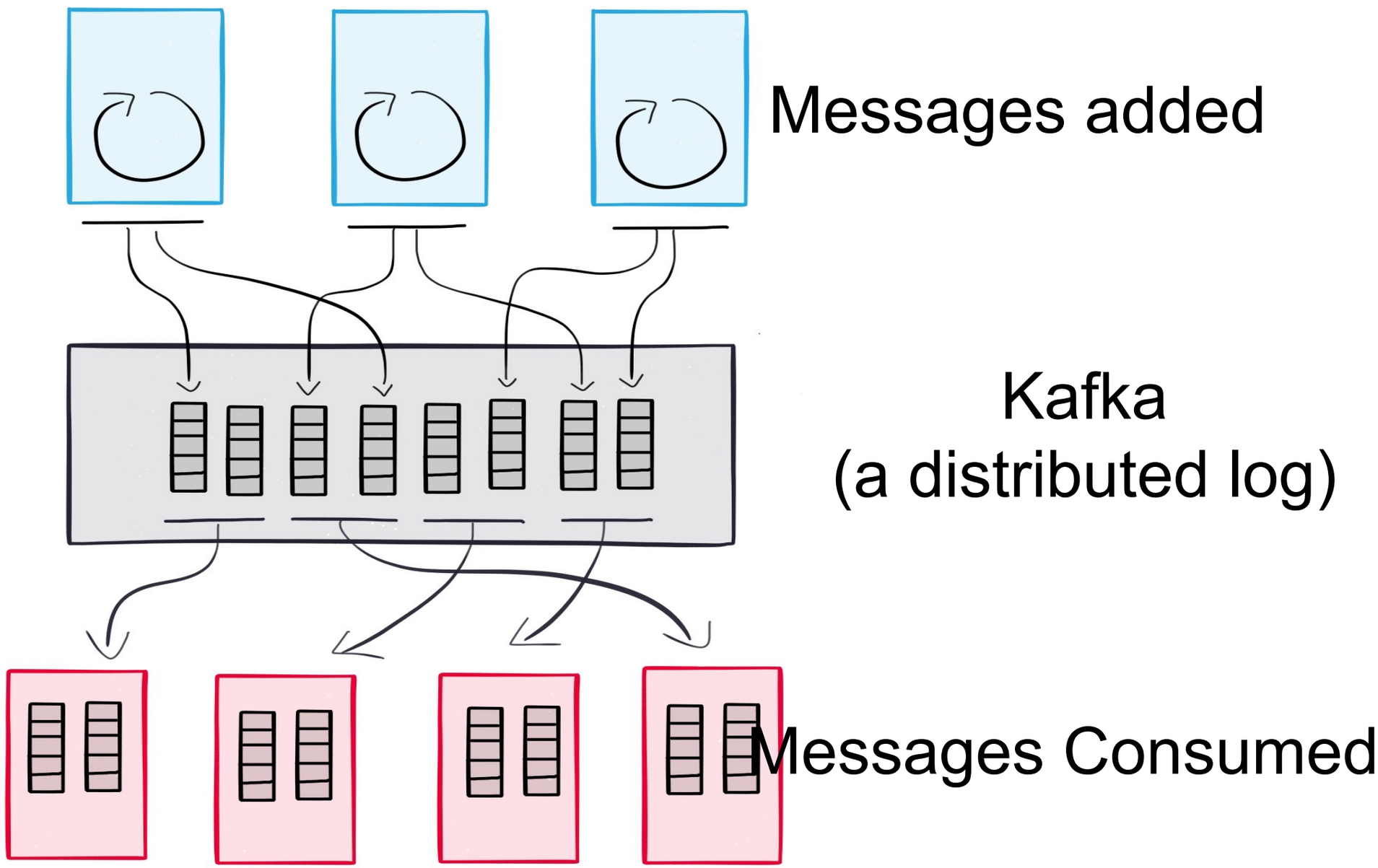
confluent

Kafka: a Streaming Platform



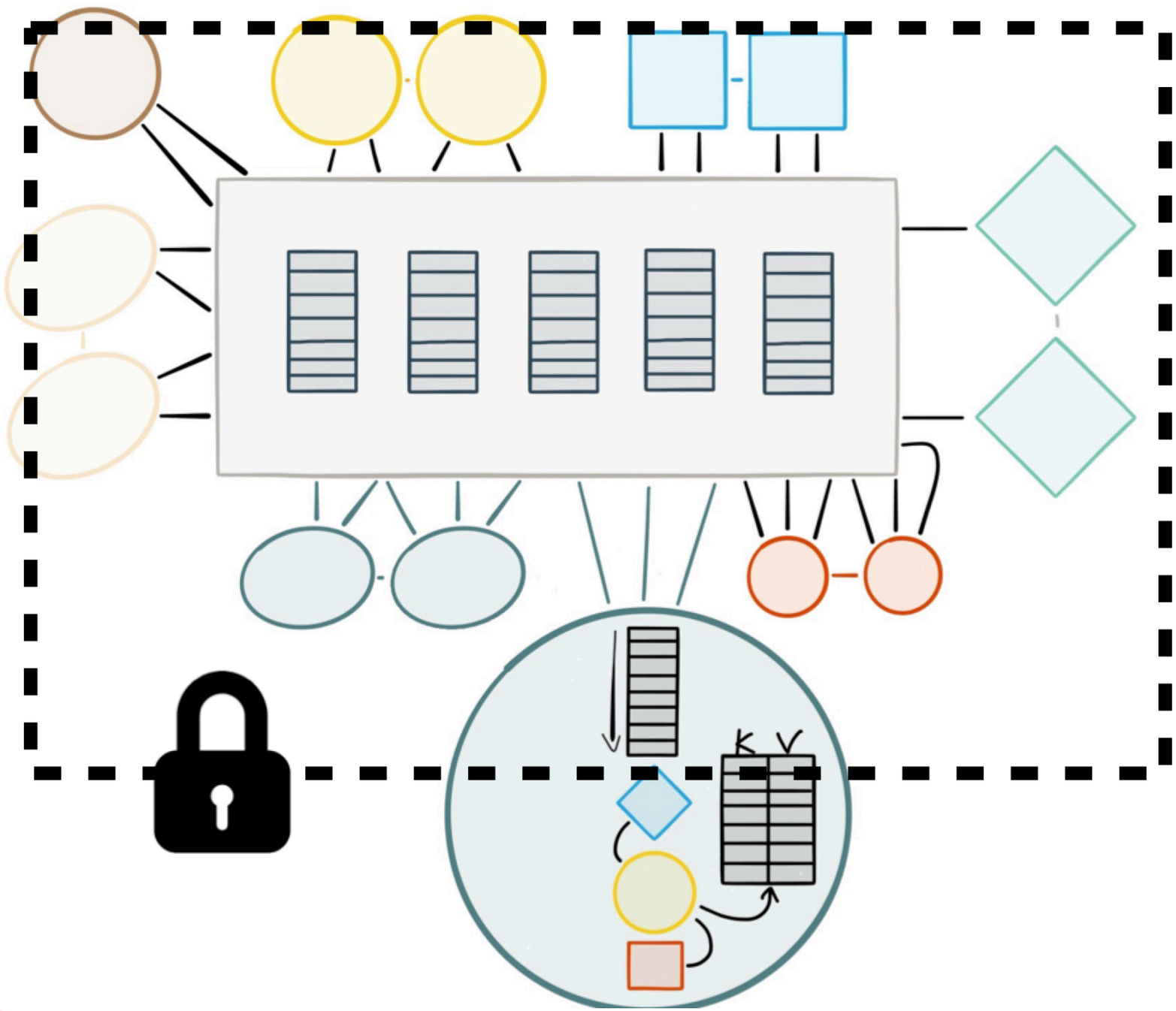
confluent

The Log



confluent

Transactions

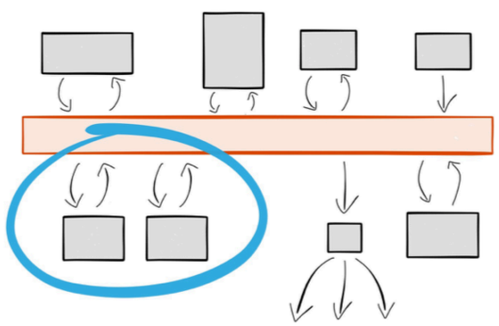


confluent

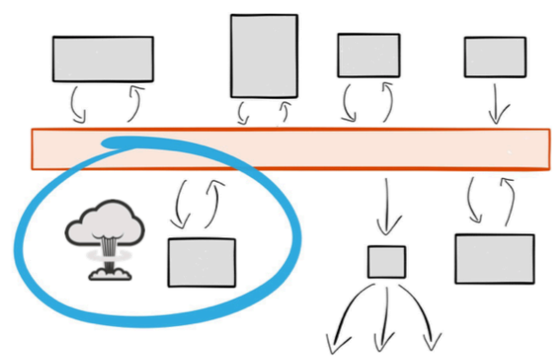
Service Backbone

Scalable, Fault Tolerant, Concurrent, Strongly Ordered, Transactional*, Retentive

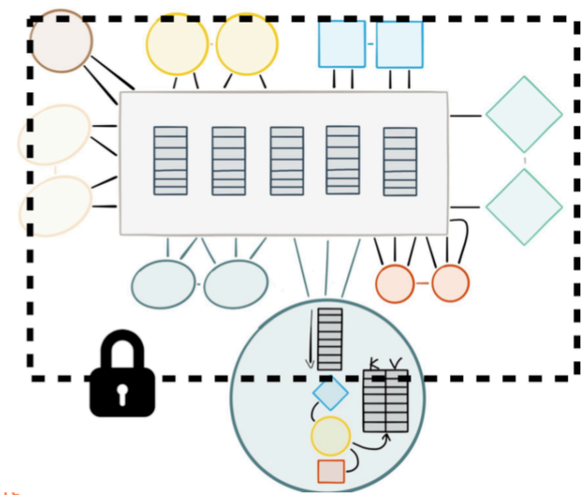
Services Naturally Load Balance



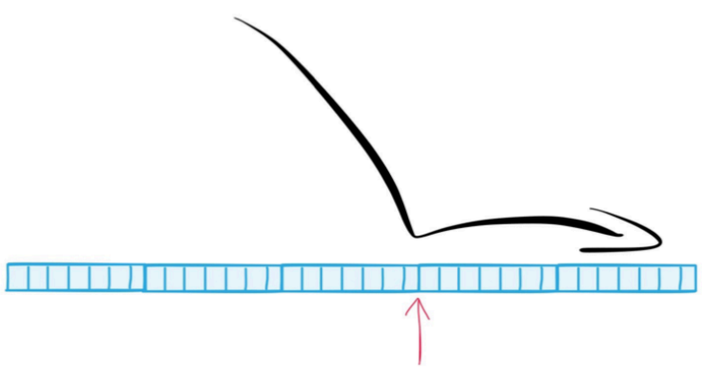
This provides Fault Tolerance



Transactions



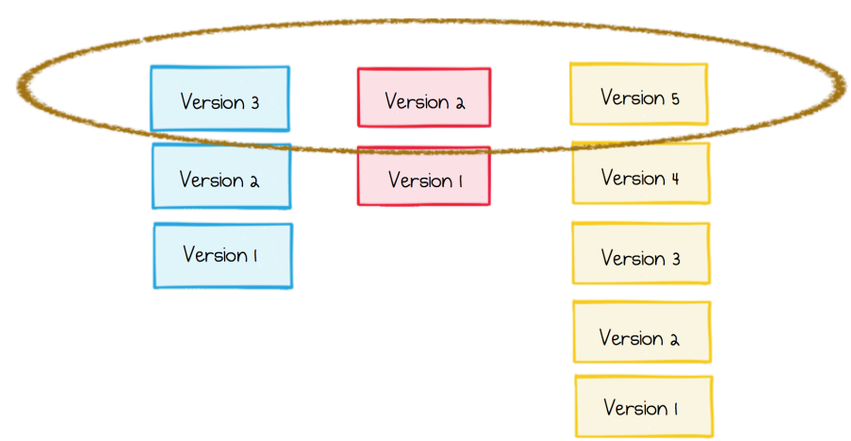
Roll back in "time"



Rewind & Replay

Compacted Log

(tables & streams)

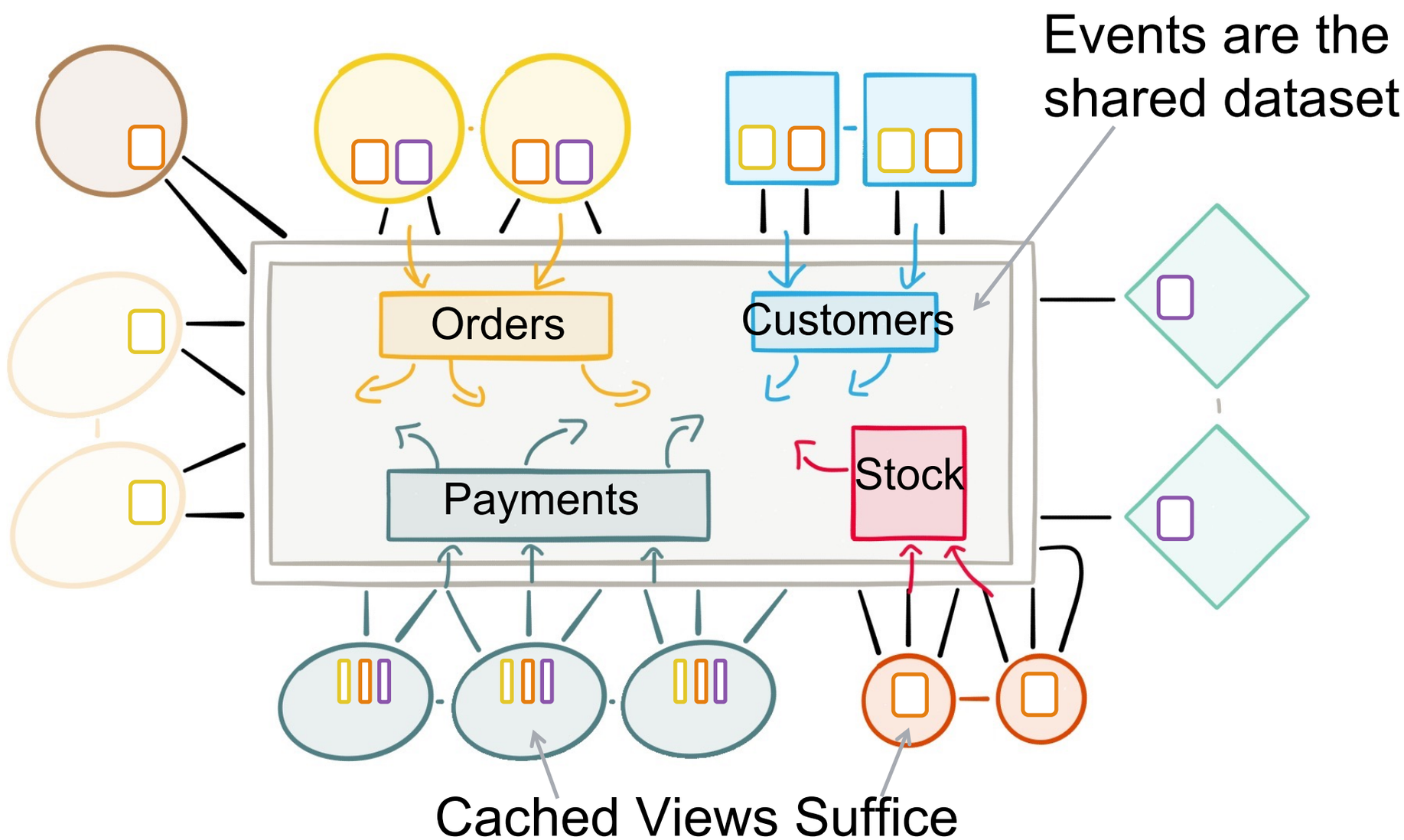


confluent

A place to keep the data-on-the-outside

confluent

Leave data in Kafka
-> Services only need caches



confluent

Now add Stream Processing

confluent

What is Stream Processing?

A machine for combining and processing streams of events

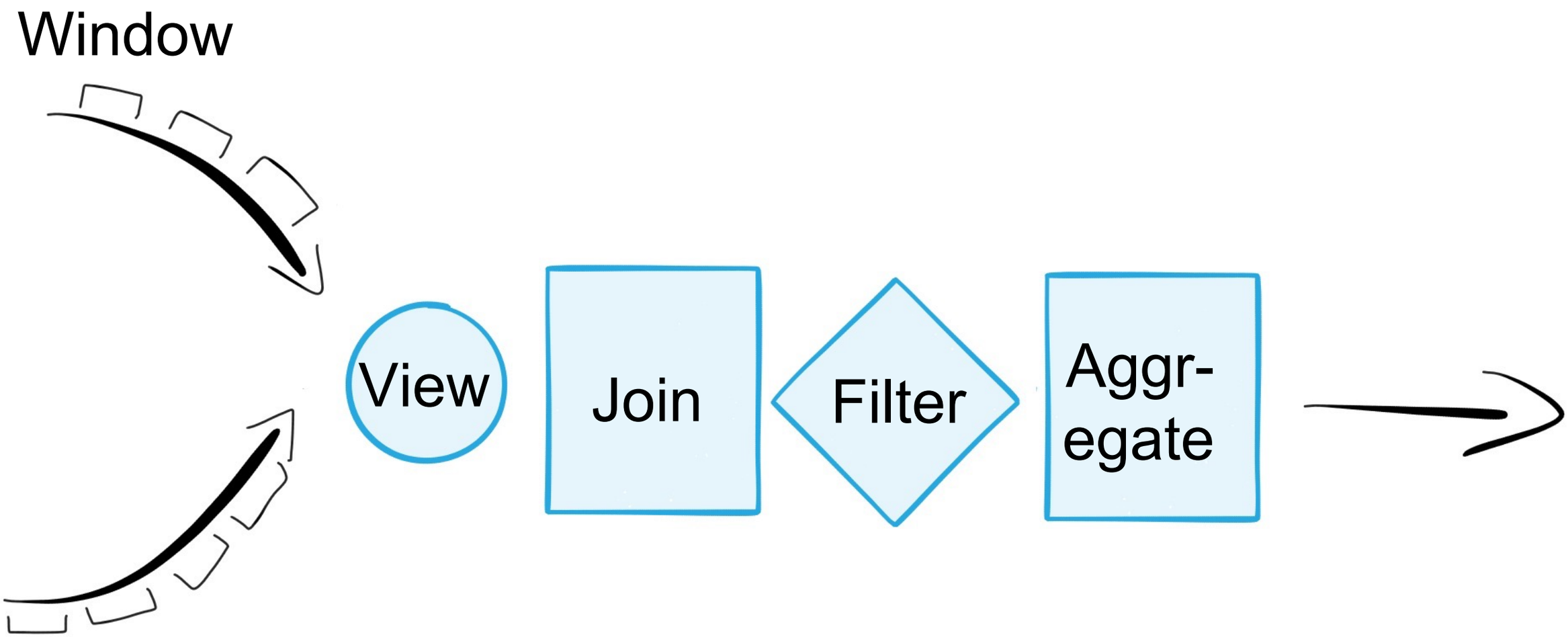
confluent

Stateful stream processing

- Data constantly updated
- Some data accumulated in each service
  - Accumulated via a window
  - Accumulated via a key

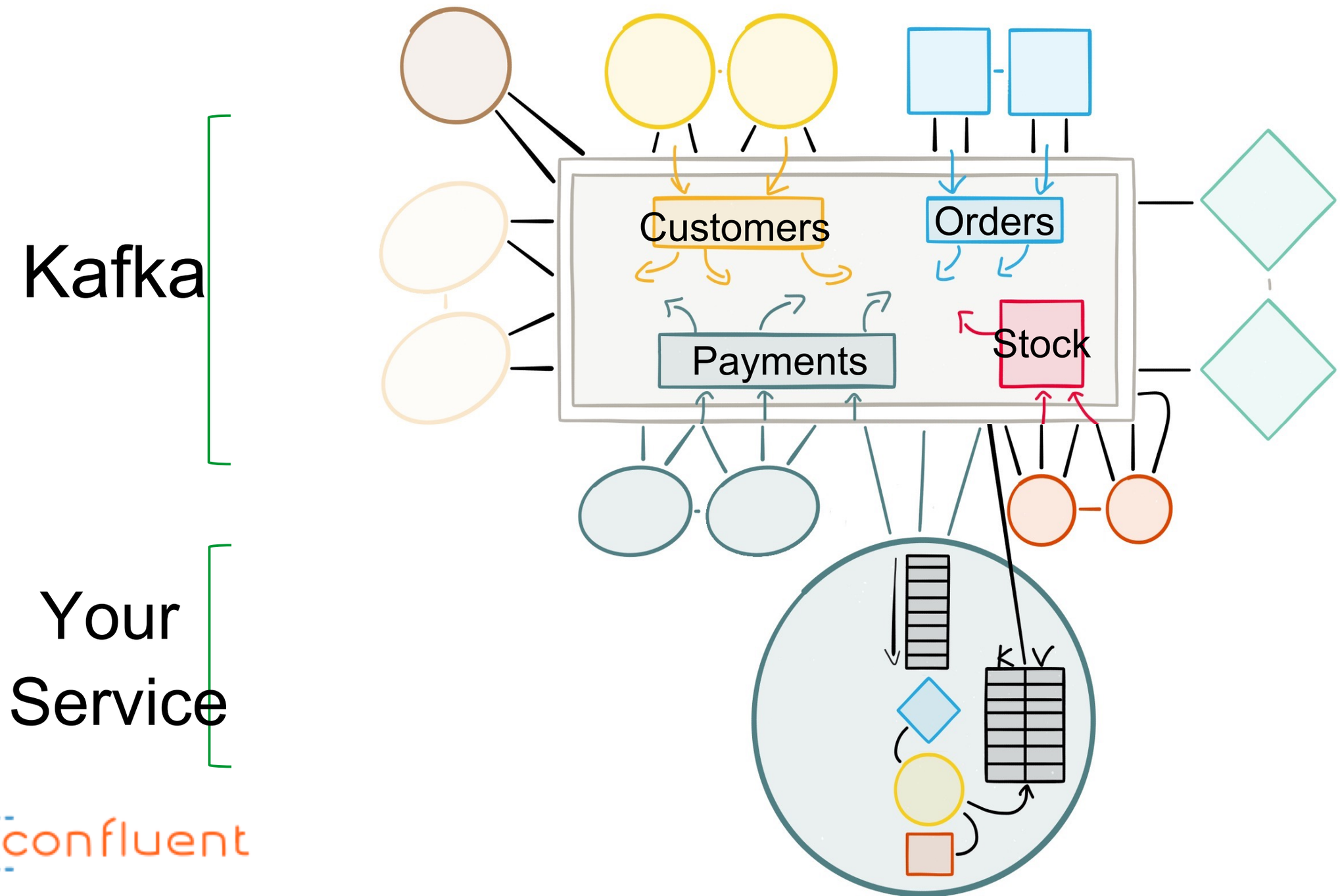
confluent

A Query Engine inside your service



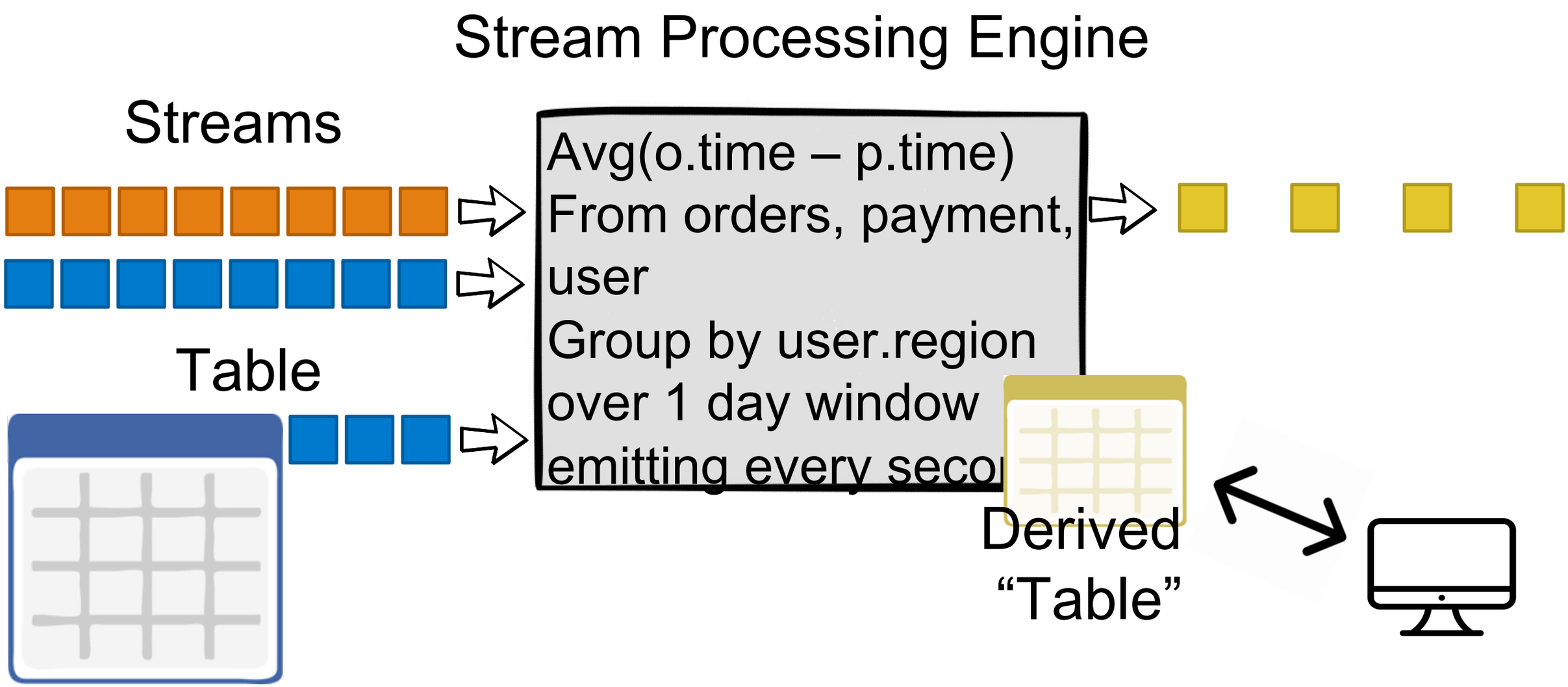
confluent

A database embedded in your service. One designed for handing streams.



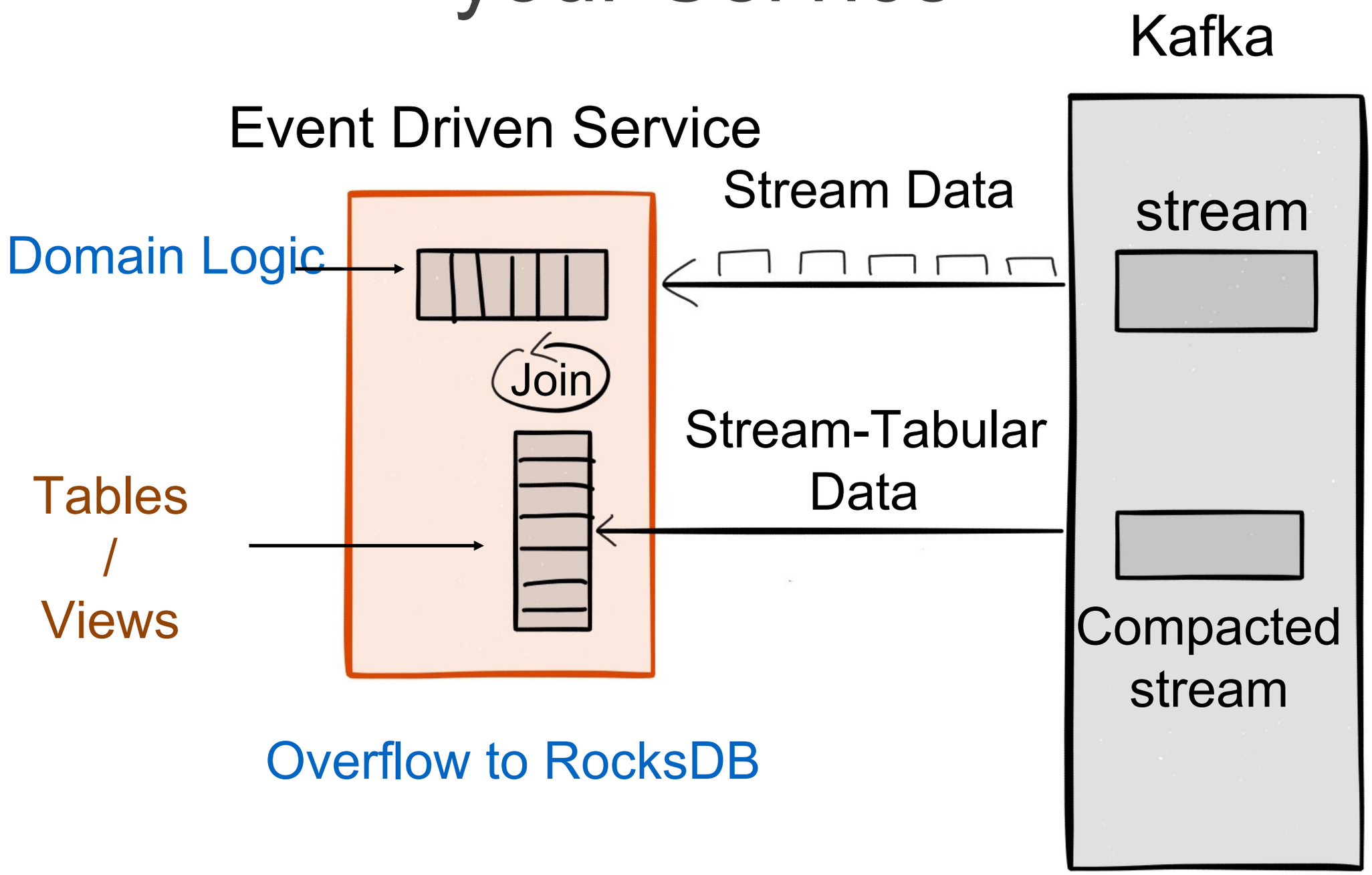
confluent

View as a Stream or a Table (in & out)



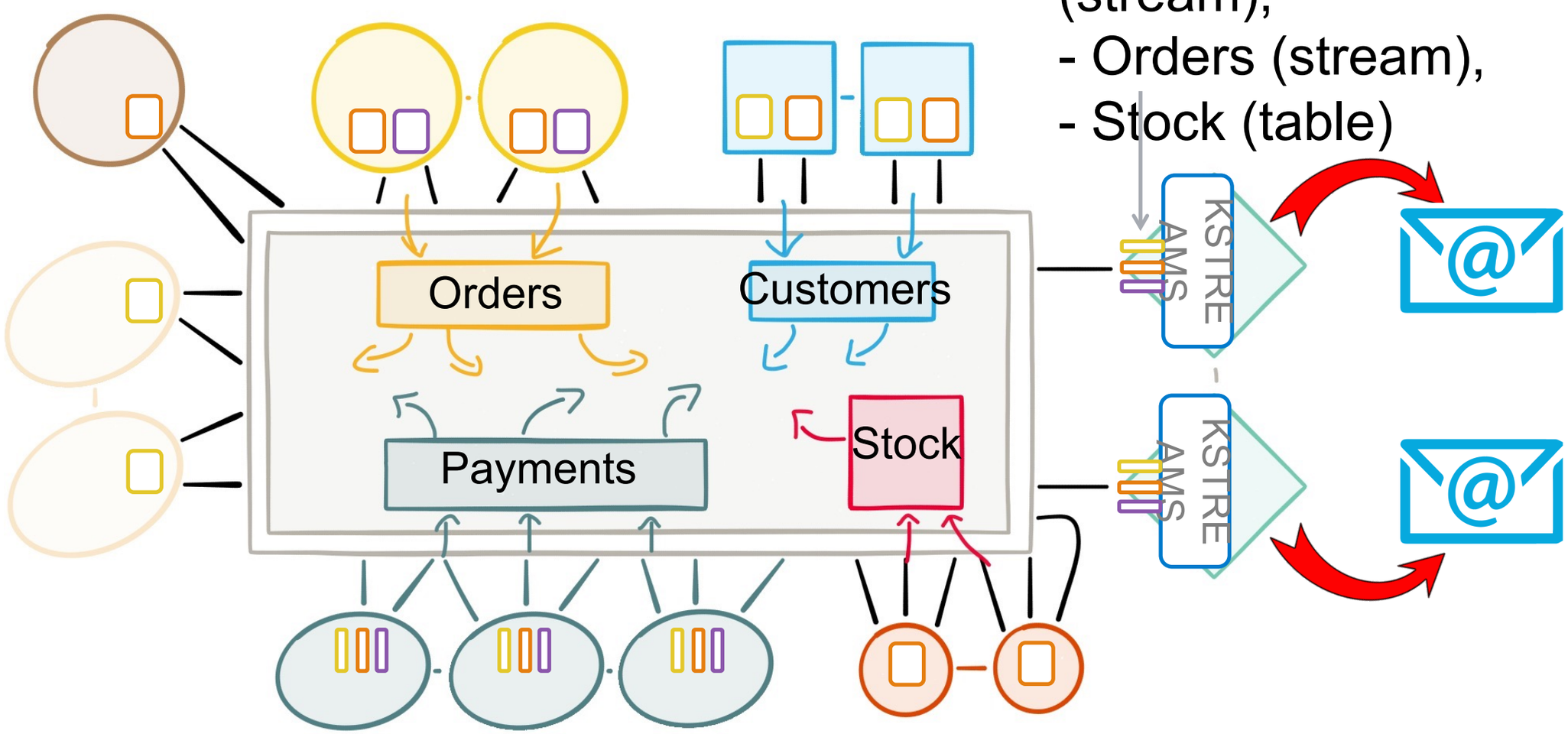
confluent

Window / Tables **Cached on disk** in your Service



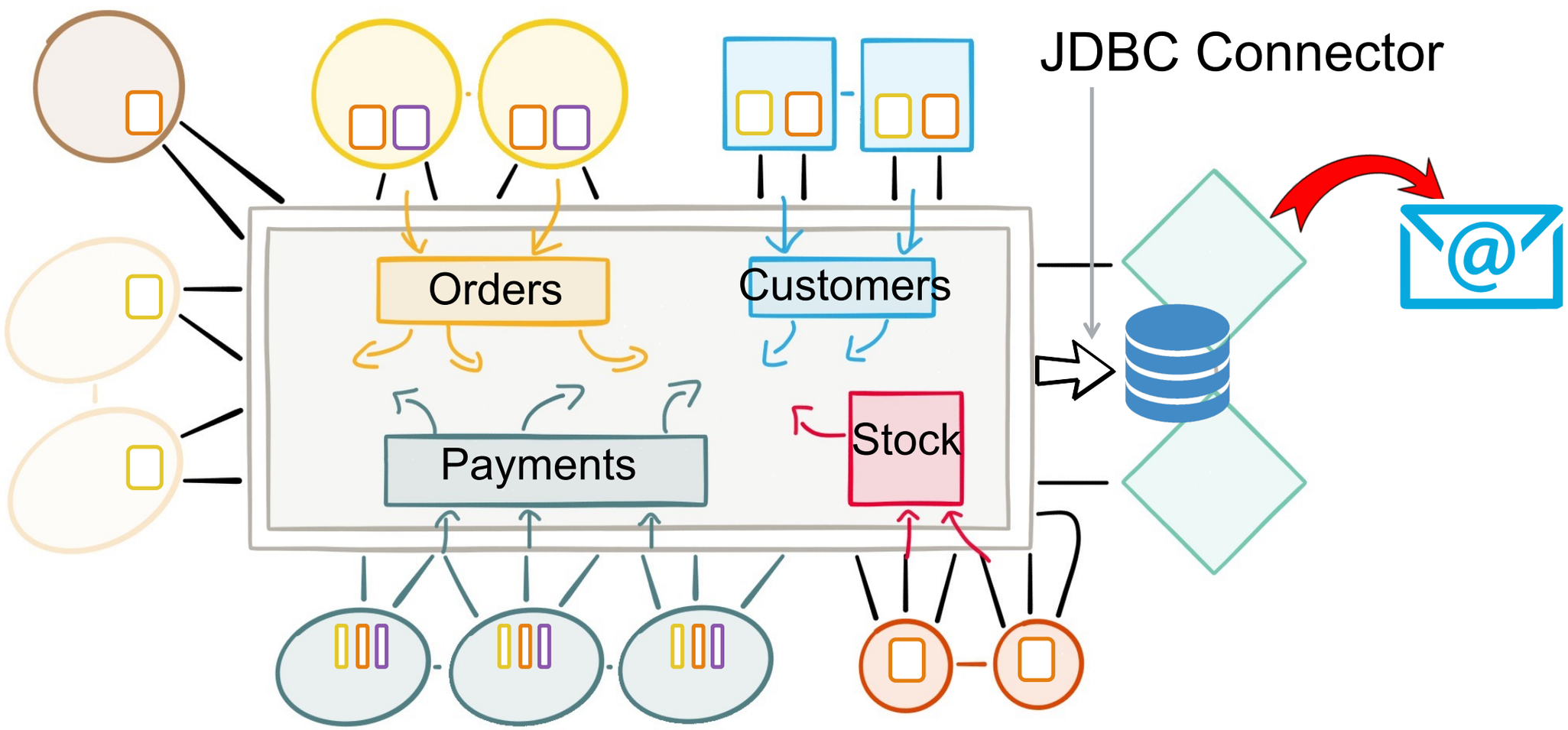
Example: Trigger an email when a payment is confirmed. Include Order & Stock description

Join:
- Payments (stream),
- Orders (stream),
- Stock (table)



confluent

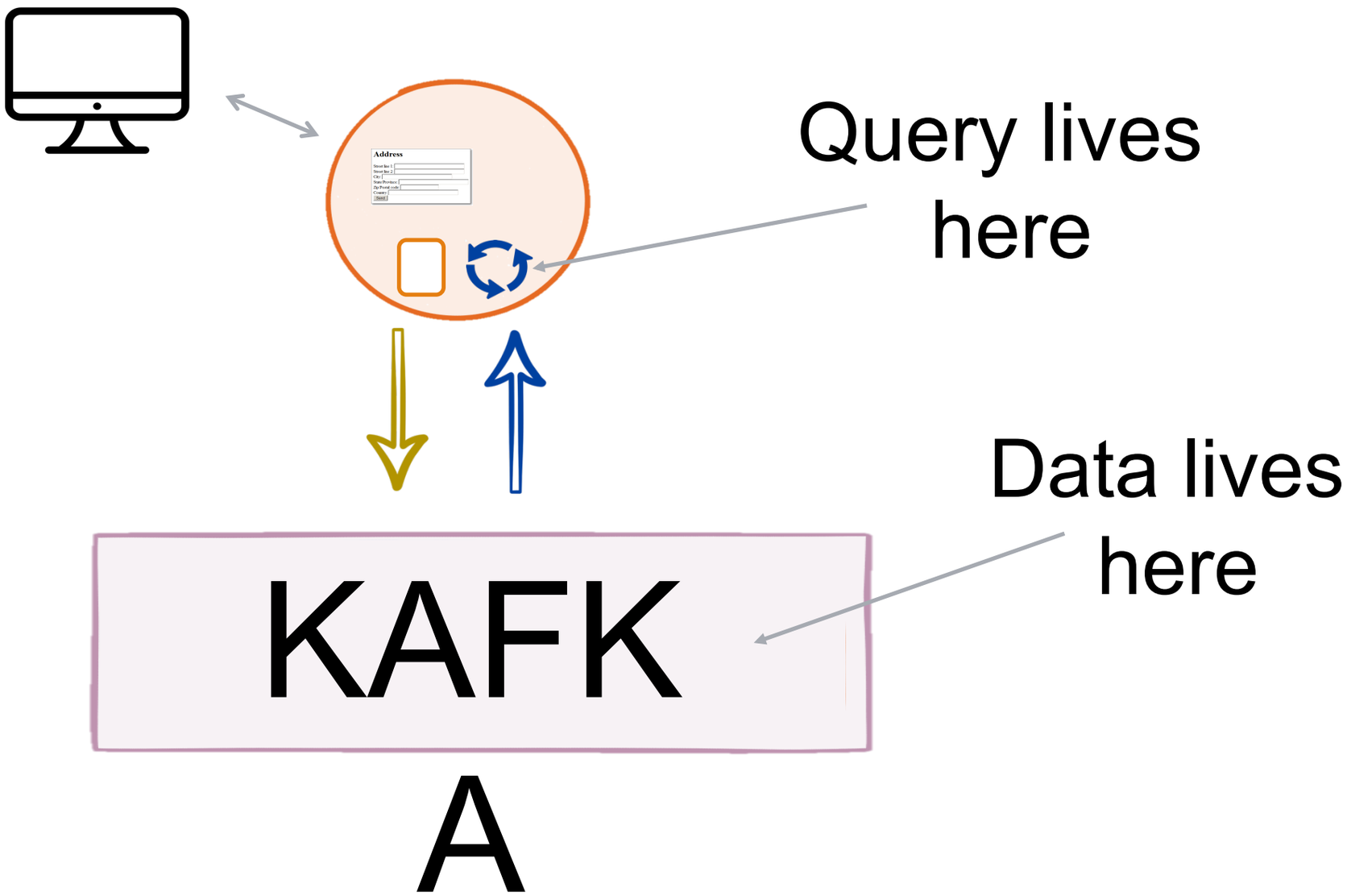
If a streaming engine doesn't cut it, branch out to database but keep it *ephemeral*



confluent

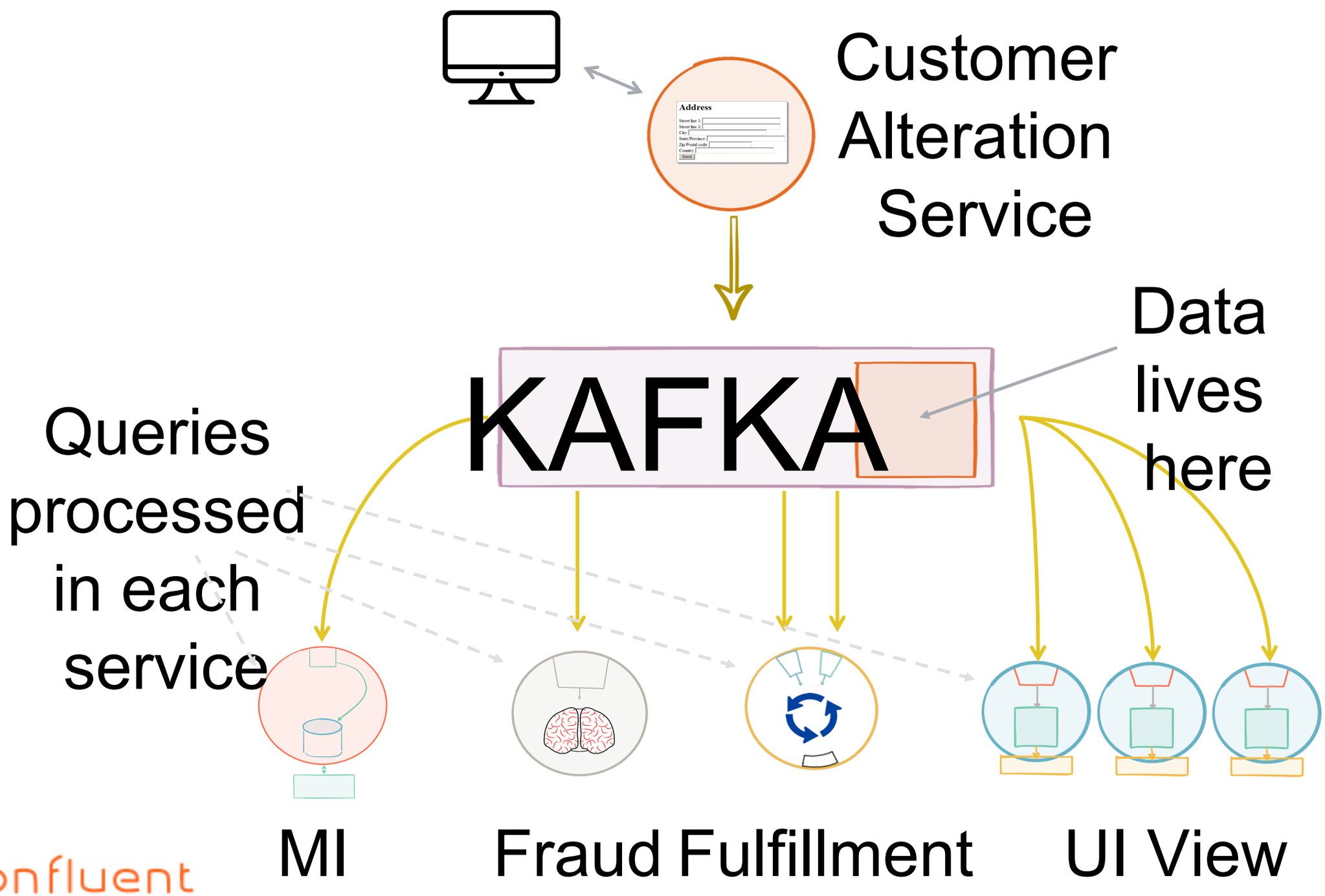
So we have shared storage in the Log, and a query engine layered on top

Data Storage + Query Engine == Database?



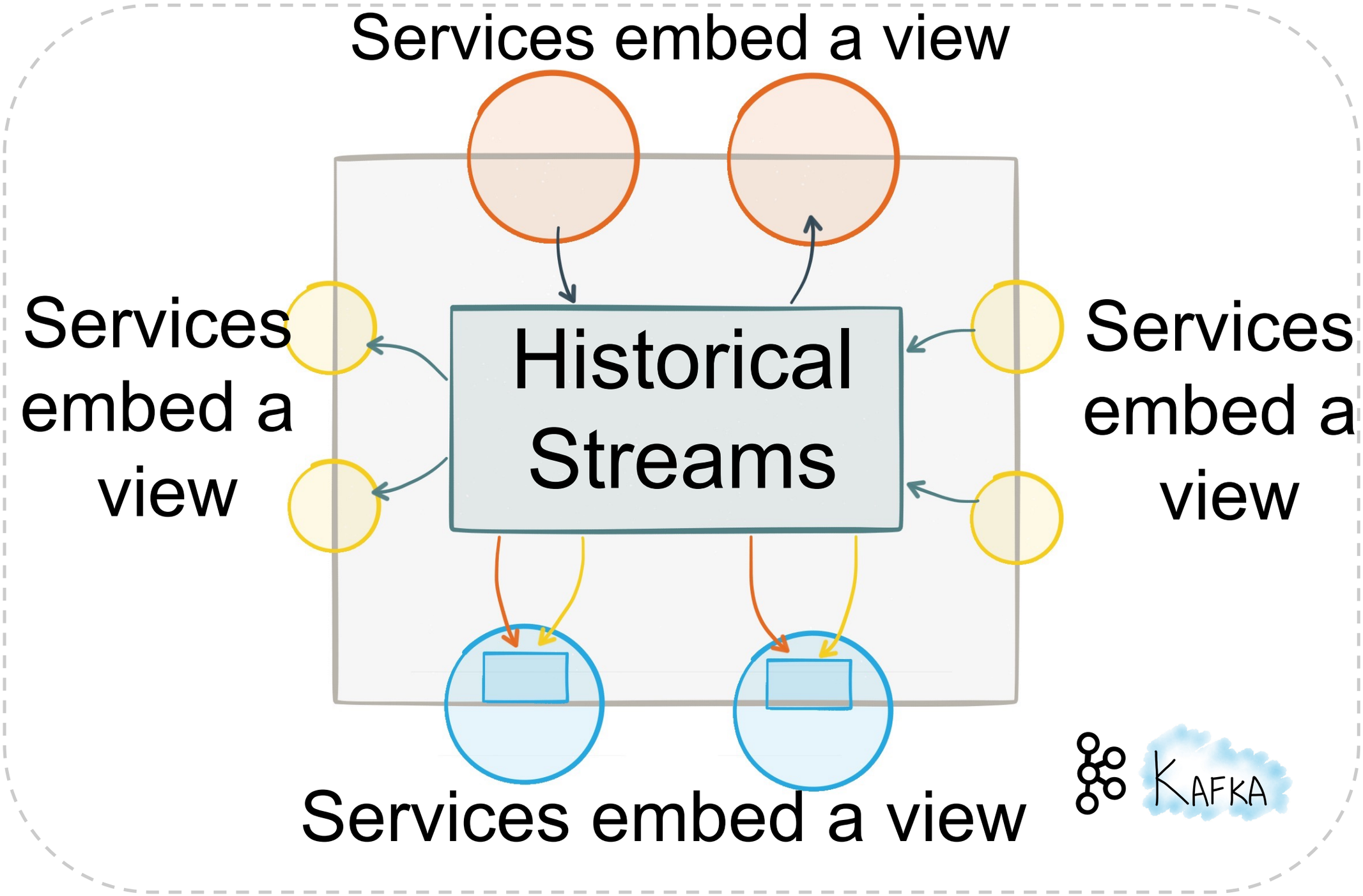
confluent

1 x Data Storage + n x Query == Shared Database?



confluent

A Database, Inside Out



confluent

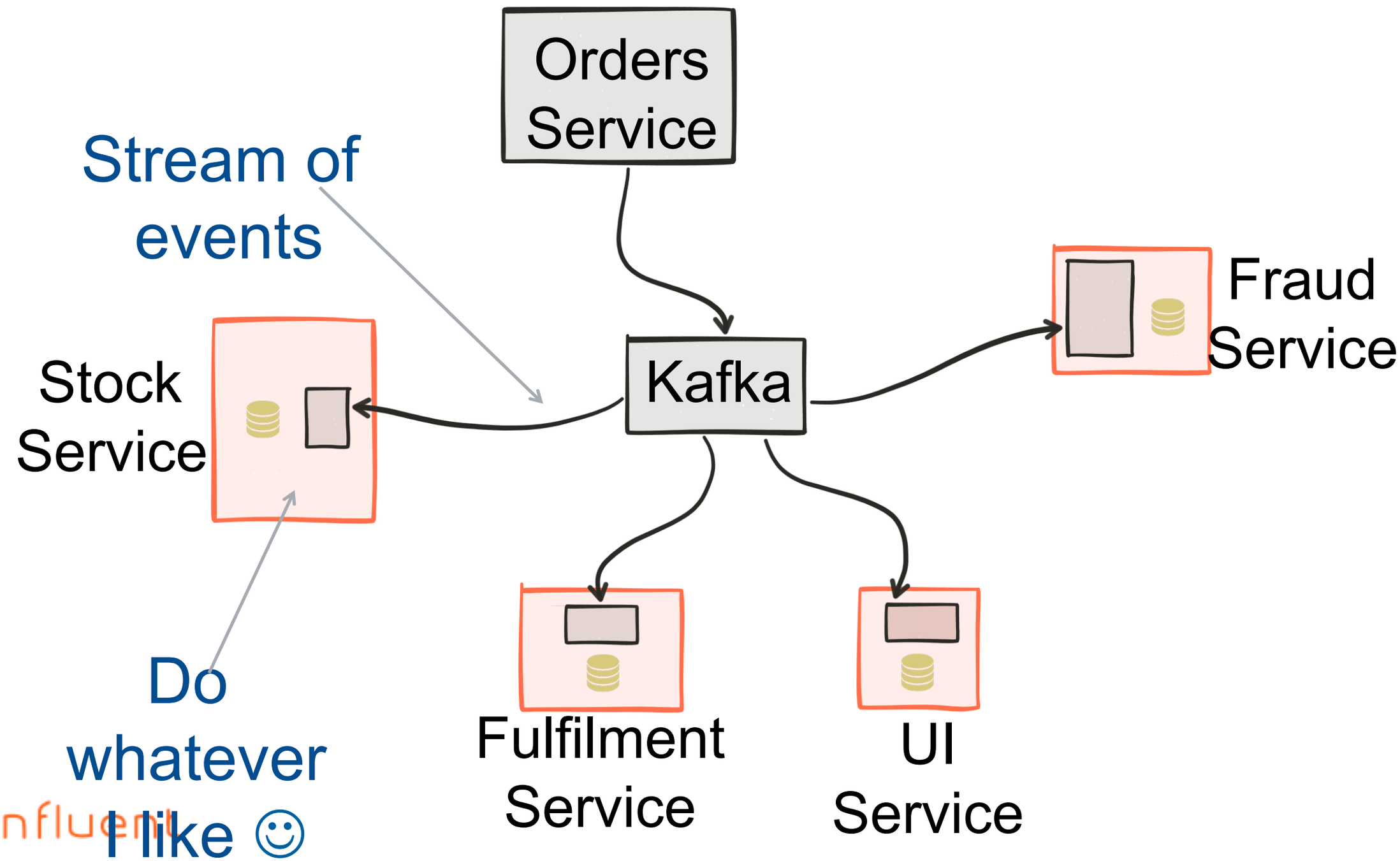
Microservices shouldn’t share a database

confluent

But this isn’t a normal database

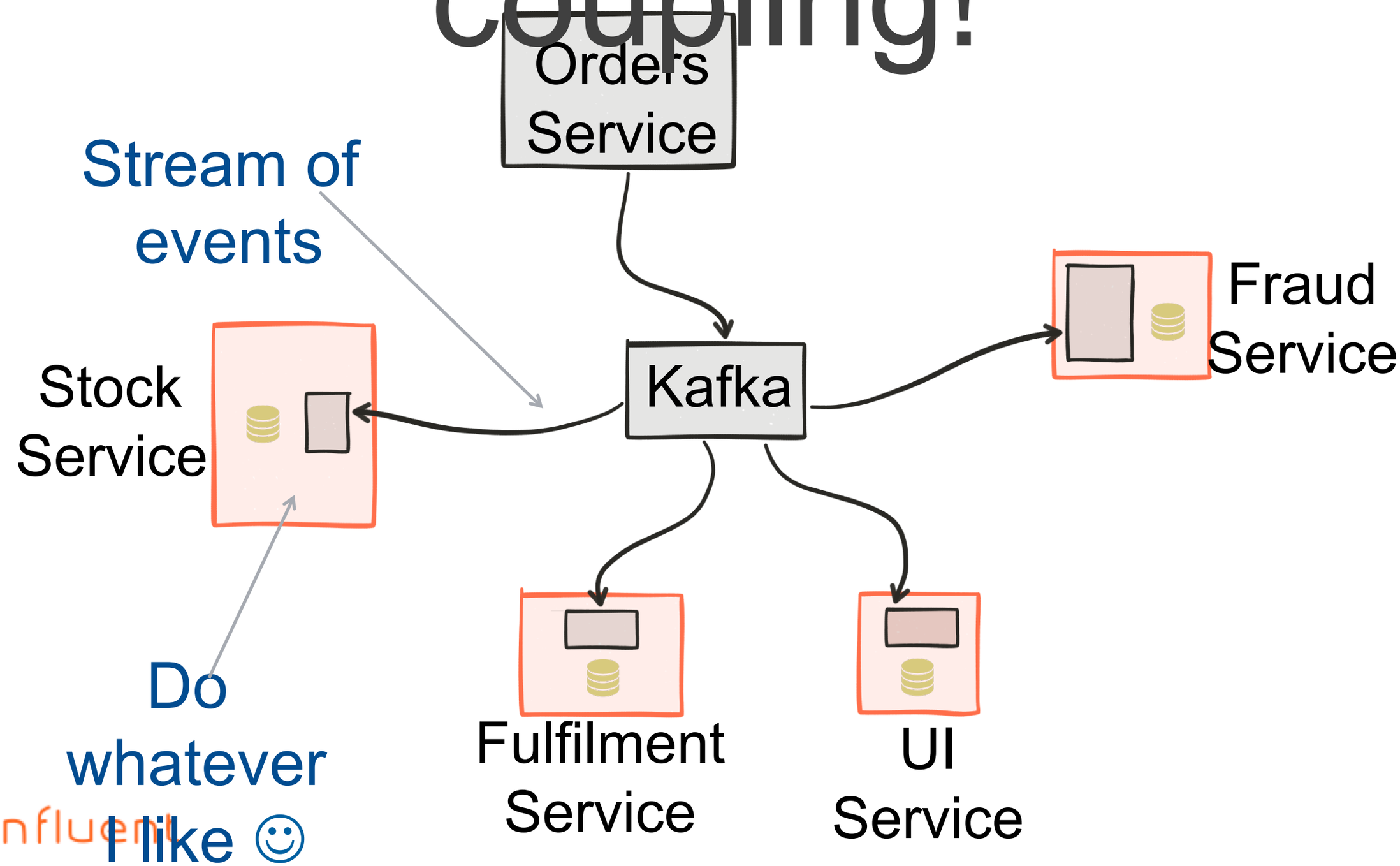
confluent

Event Broadcast has the lowest coupling



confluent

Centralizing immutable data doesn't affect coupling!

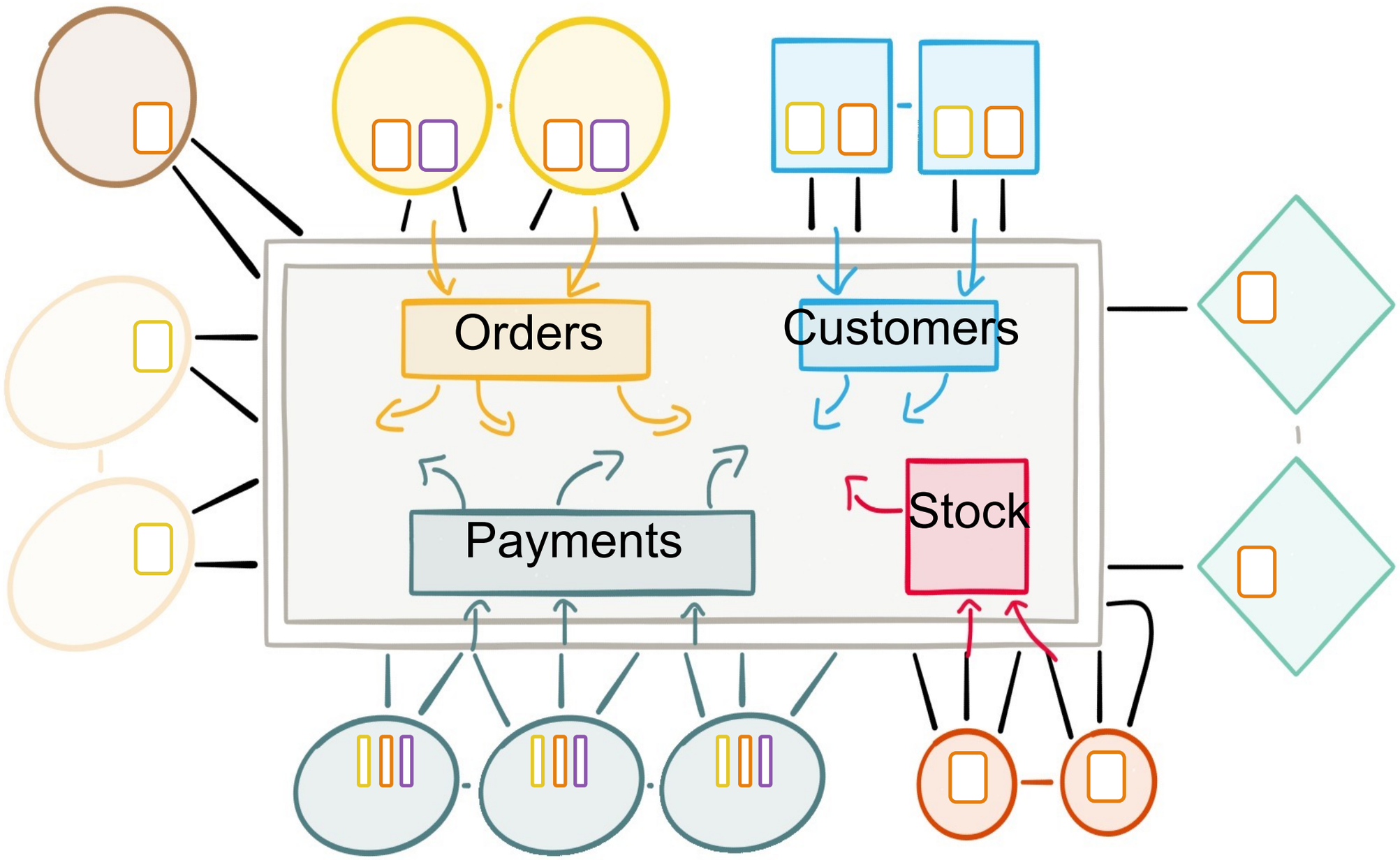


confluent

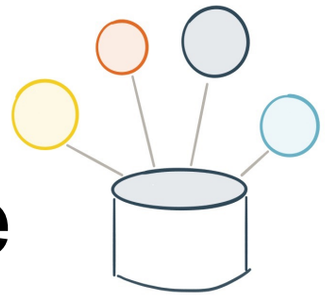
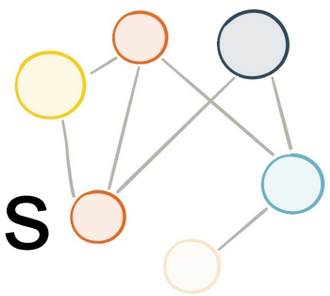
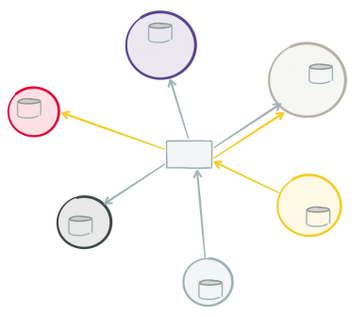
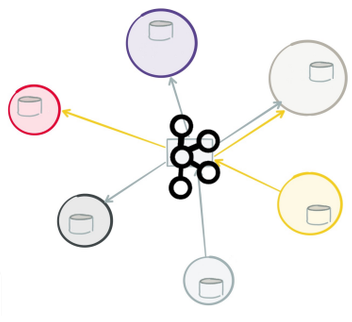
To share a database, turn it inside out!

confluent

AKA: a machine for creating views



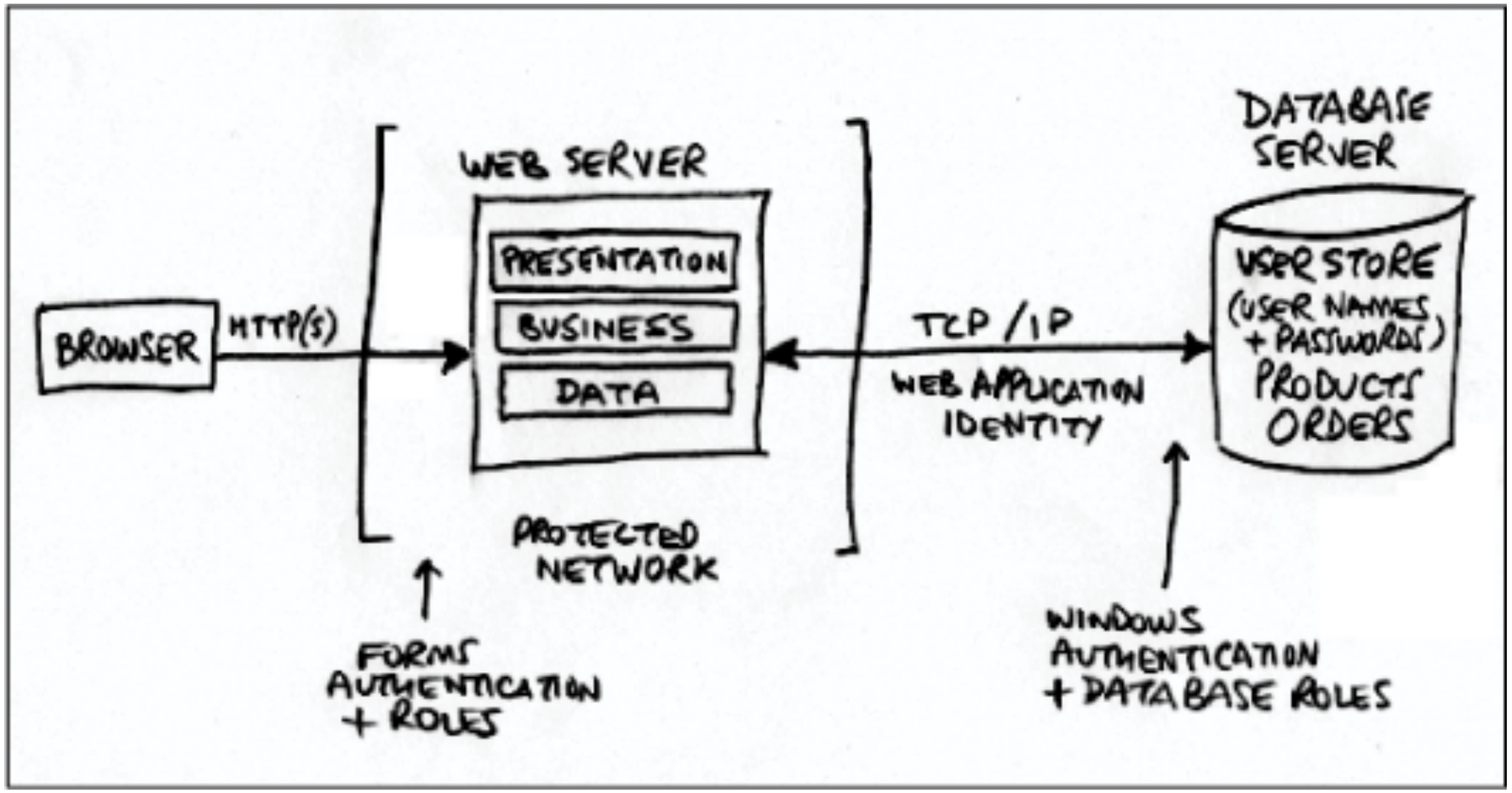
confluent

	Ease of change	Data Accessibility	Data Erosion
Shared database 	☹ ☹	☺	☺
Service Interfaces 	☹	😐	☺
Event Broadcast 	☺	☹	☹
Stream Data Platform 	☺	☺	☺

confluent

So...

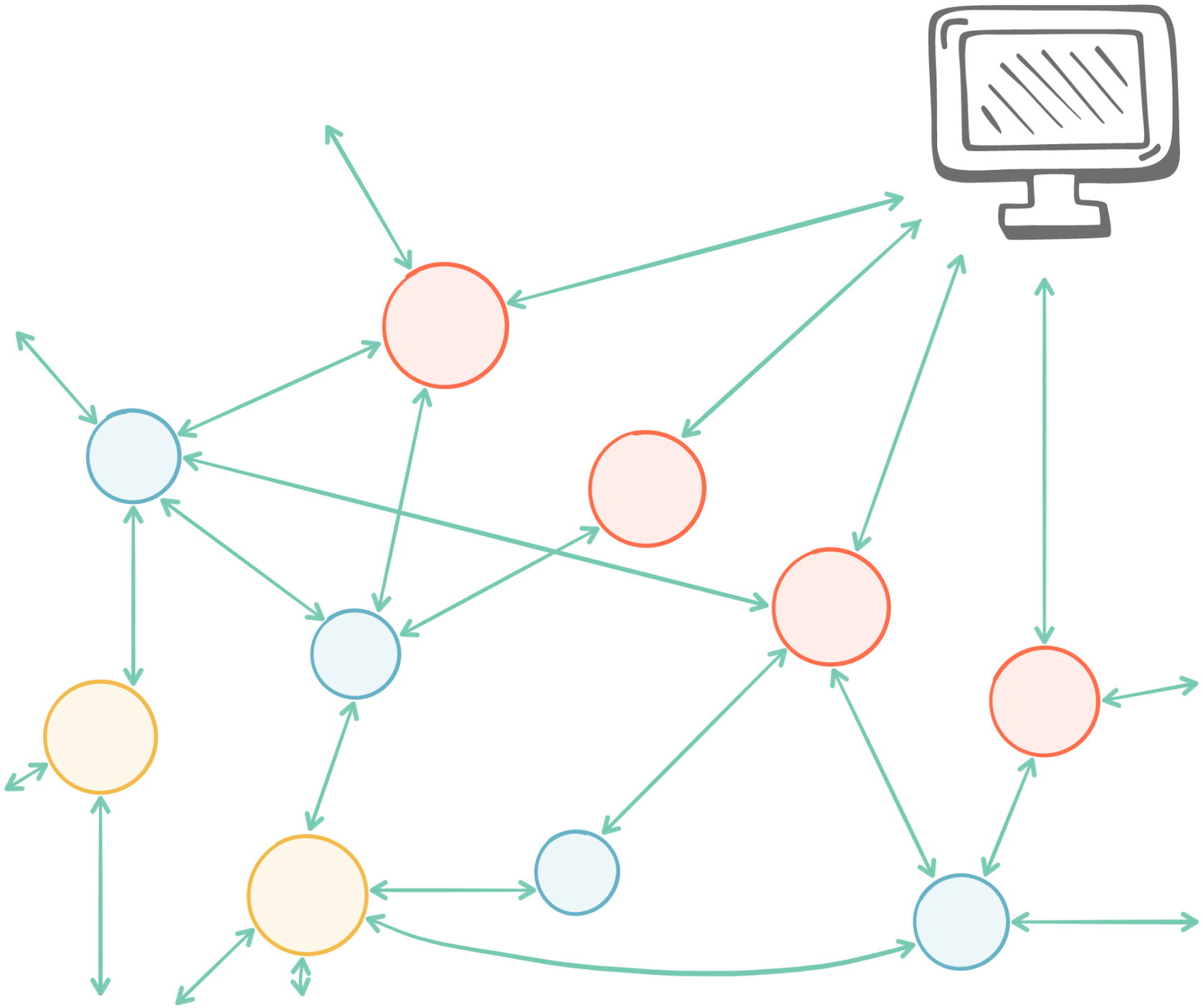
Good architectures have little to do with this:



It’s about how systems evolves over time

confluent

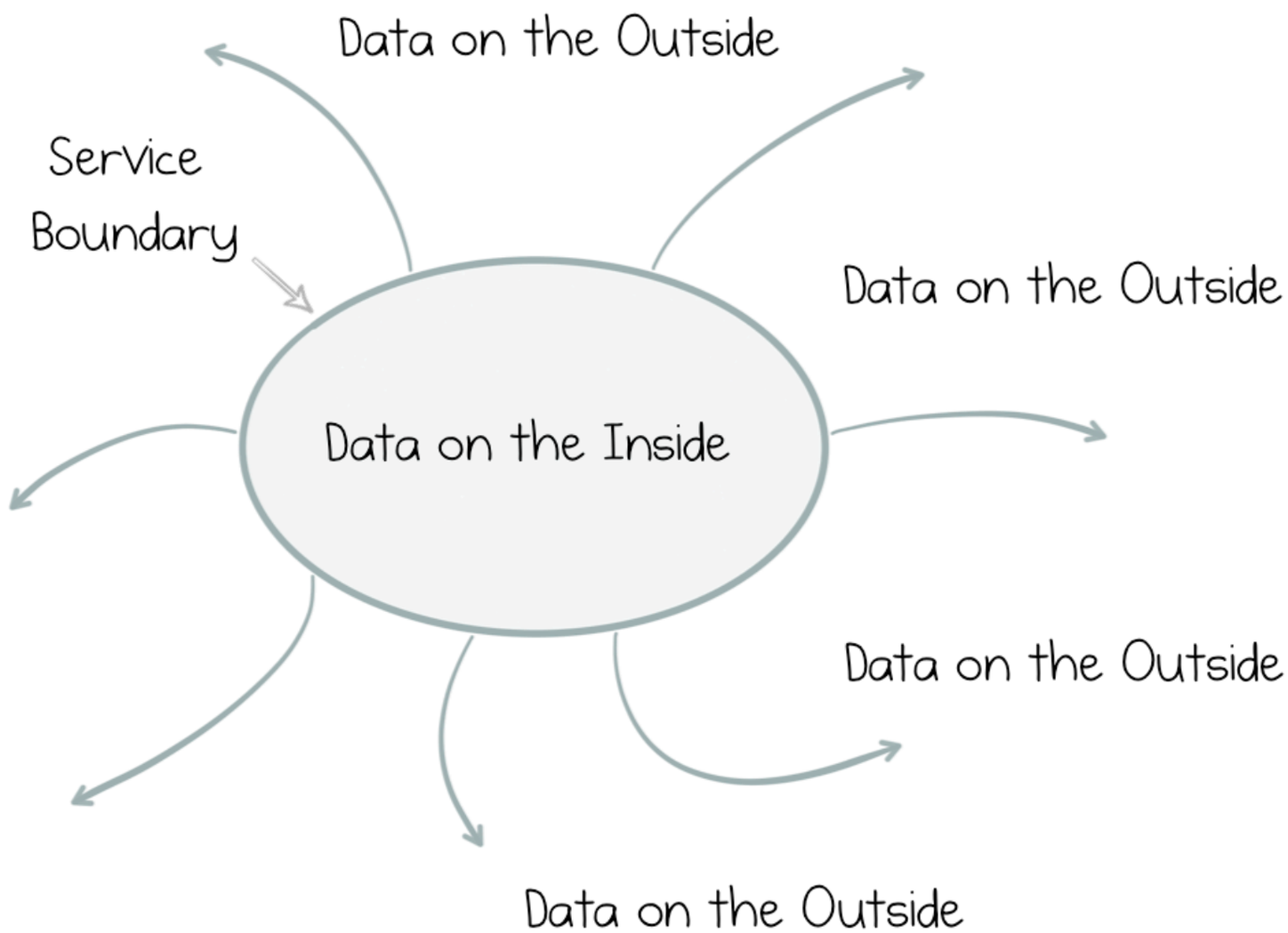
Request driven isn’t enough



- High coupling
- Hard to handle async flows
- Hard to move and join datasets.

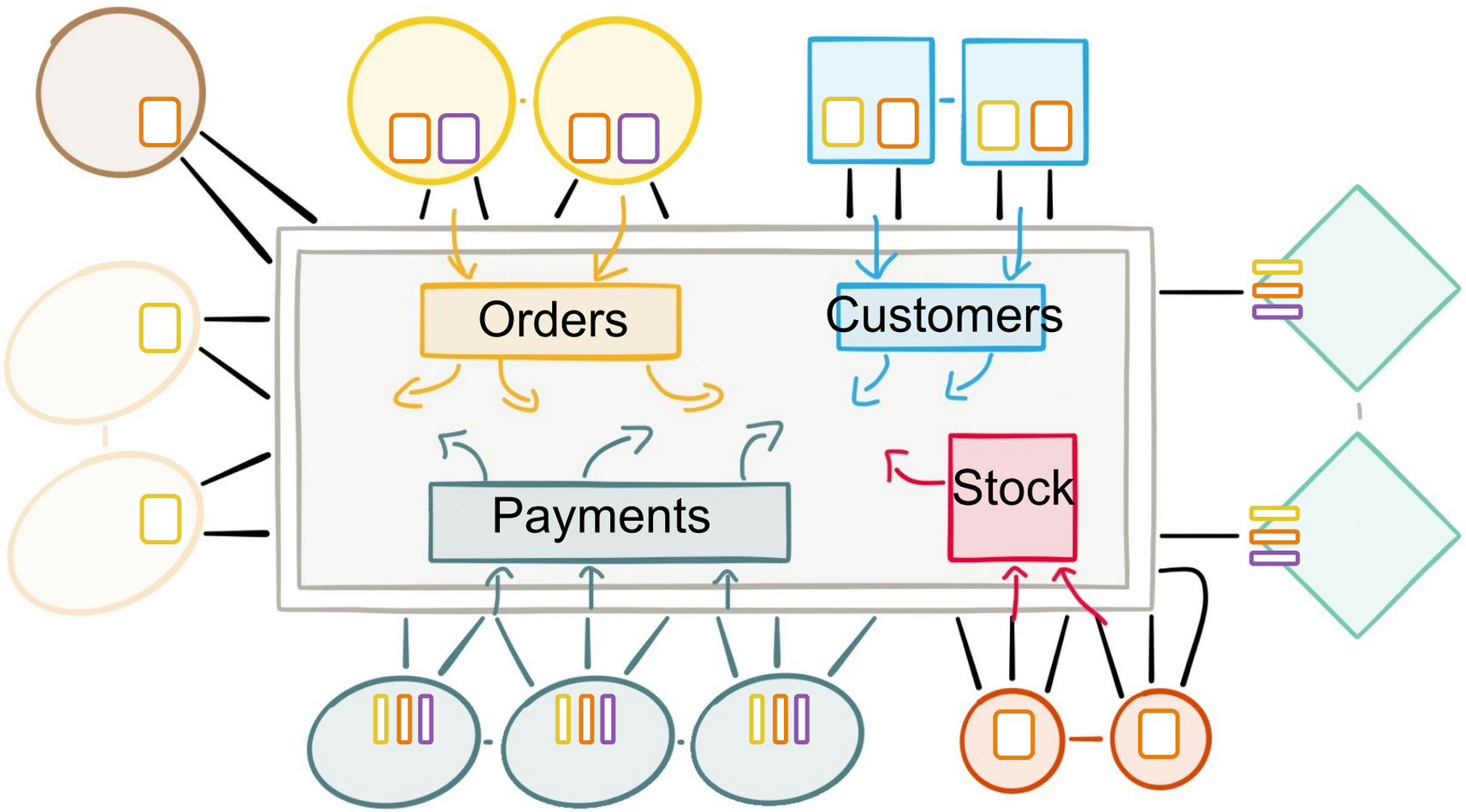
confluent

Embrace data that lives and flows between services



confluent

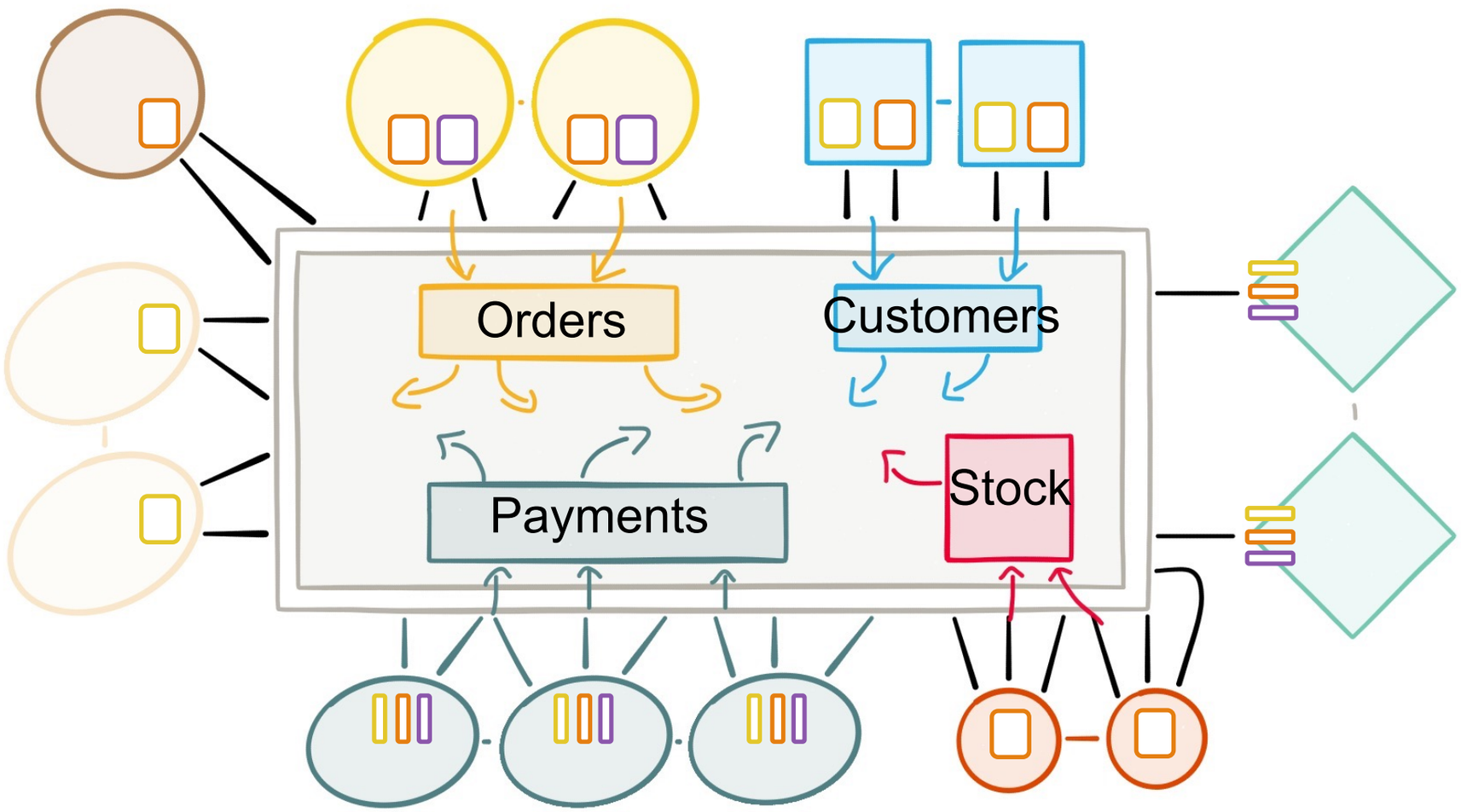
Give services independence



eedom to tap into and manage shared d

confluent

But build on single stream data platform everyone can access and share



confluent

Mechanism for evolving an architecture efficiently over time

confluent

WIRED Principals

- Wimpy: Start simple, lightweight and fault tolerant.
- Immutable: Build a retentive, shared narrative.
- Reactive: Leverage Asynchronicity. Focus on the now.
- Evolutionary: Use only the data you need today.
- Decentralized: Receiver driven.

confluent

References

- Stopford: The Data Dichotomy: https://www.confluent.io/blog/data-dichotomy-rethinking-the-way-we-treat-data-and-services/

- Kleppmann: Turning the Database Inside Out: https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/

- Helland: Immutability Changes Everything: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf

- Helland: Data on the Inside vs Data on the Outside: http://cidrdb.org/cidr2005/papers/P12.pdf

- Kreps: The Log: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

- Narkhede: Event Sourcing, CQRS & Stream Processing: https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/

confluent

Twitter:
@benstopford

Blog series at
http://confluent.io/blog

More coming very shortly