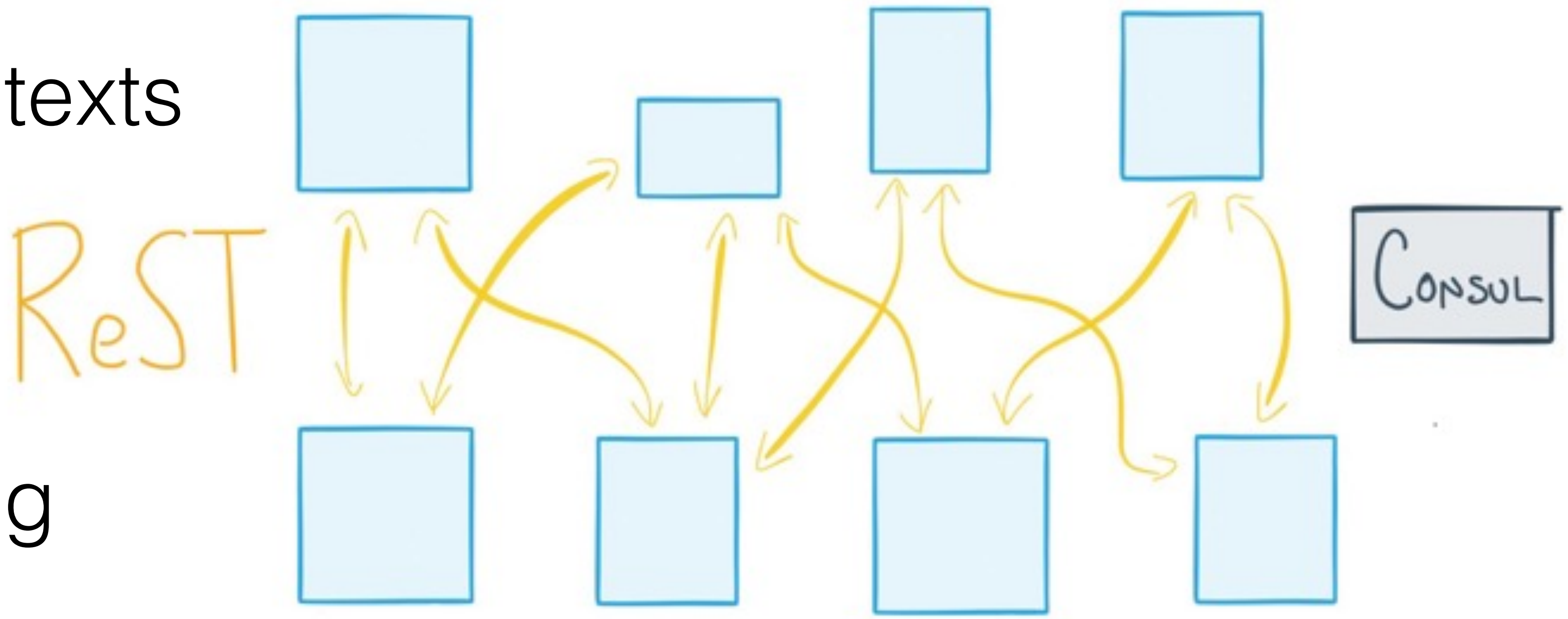
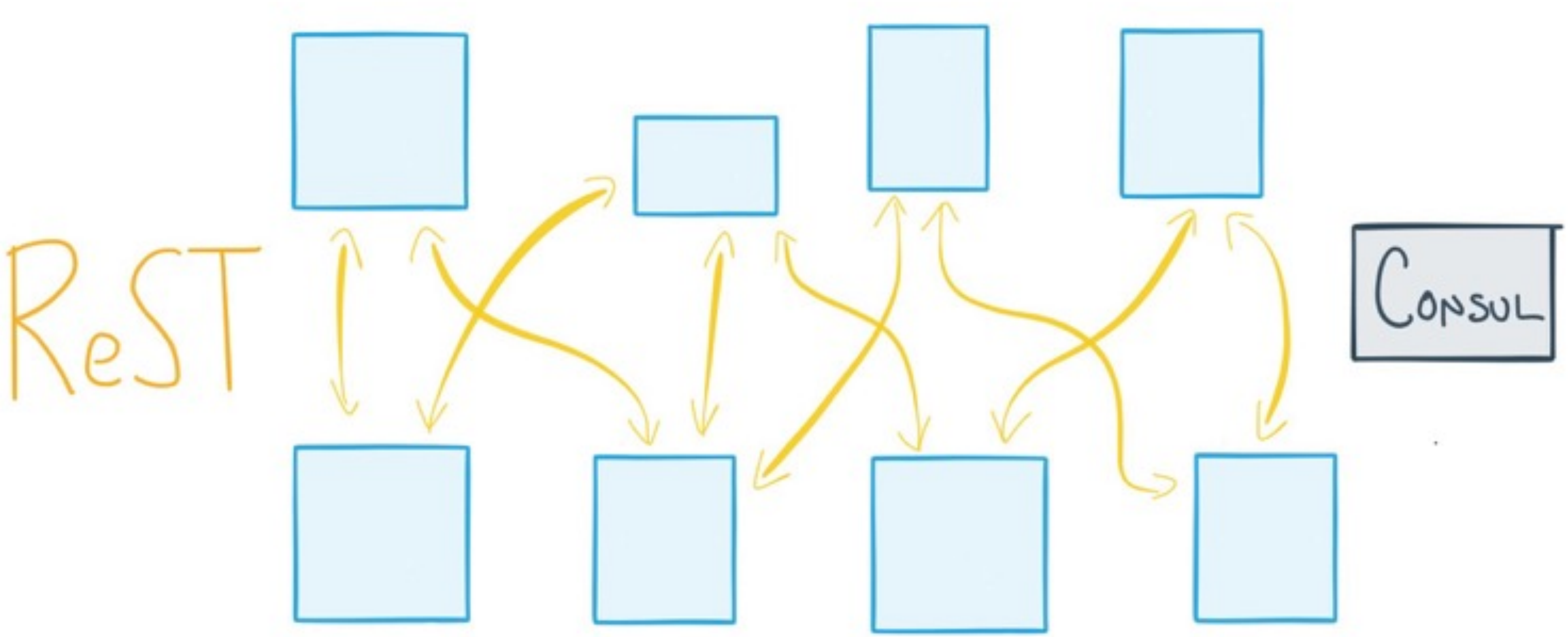
Microservices in a Streaming World

There are many good reasons for building service-based systems

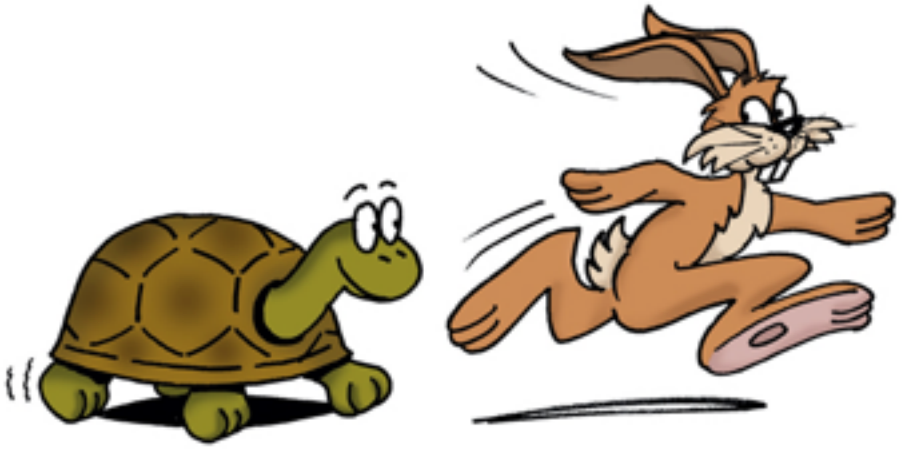
- Loose Coupling
- Bounded Contexts
- Autonomy
- Ease of scaling
- Composability



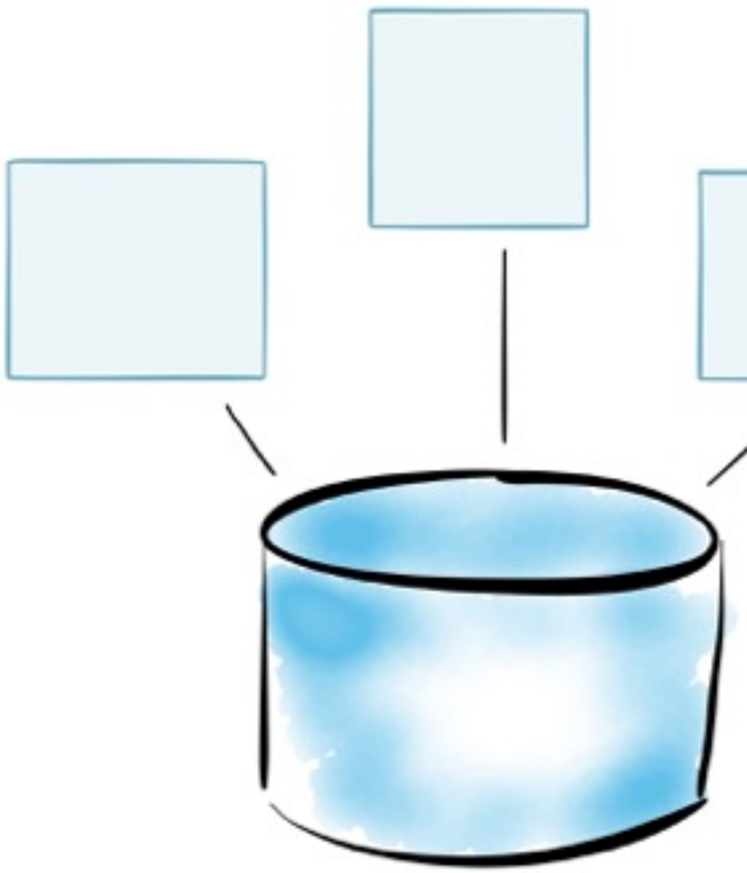
But when we do, we're building a distributed system



This can be a bit tricky

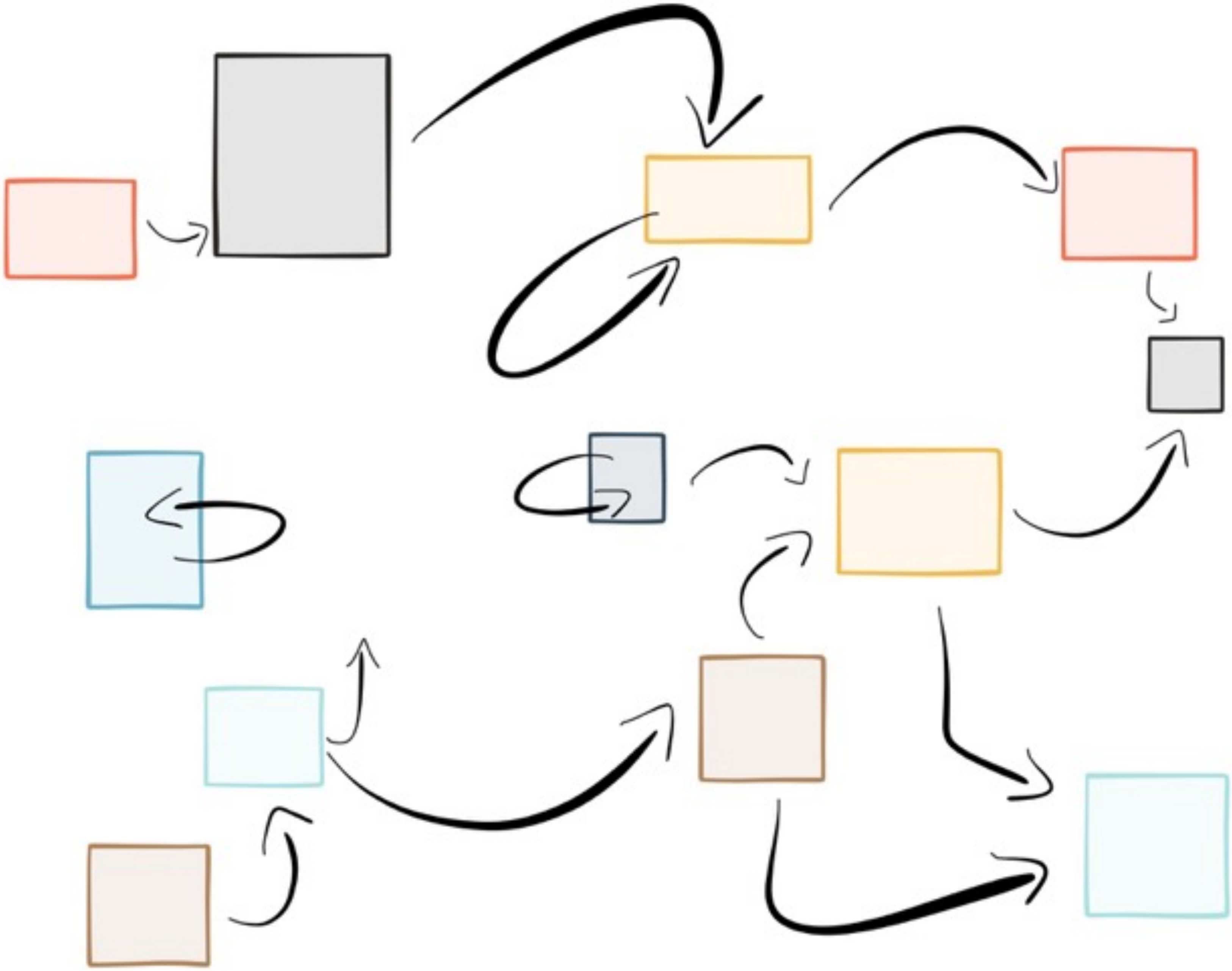


Monolithic & Centralised Approaches



Shared, mutable state

Decentralisation

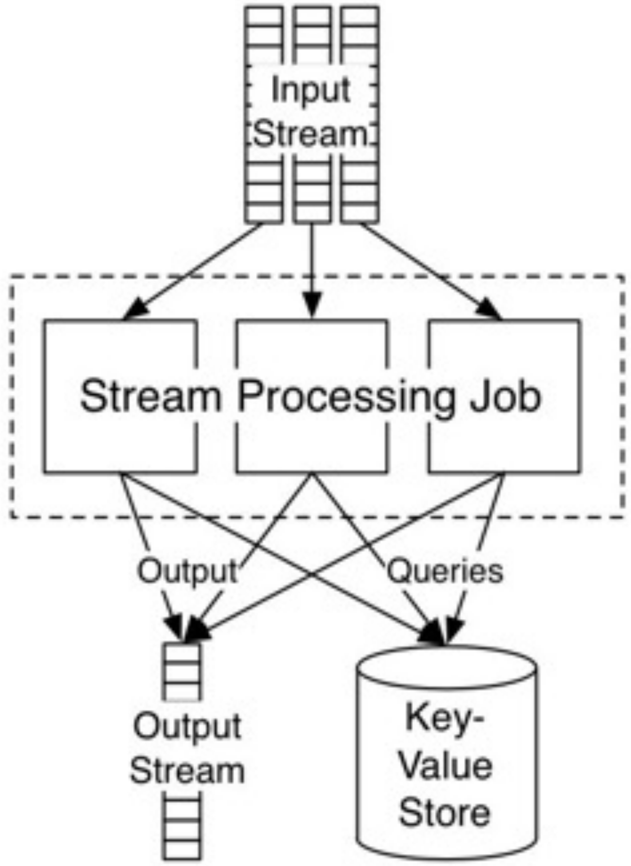


THIS IS ME

- ENGINEER AT CONFLUENT
- Ex THOUGHTWORKS +UK FINANCE

Stream Processing is a bit different

batch analytics => real time => at scale => accurately and comes with an interesting toolset

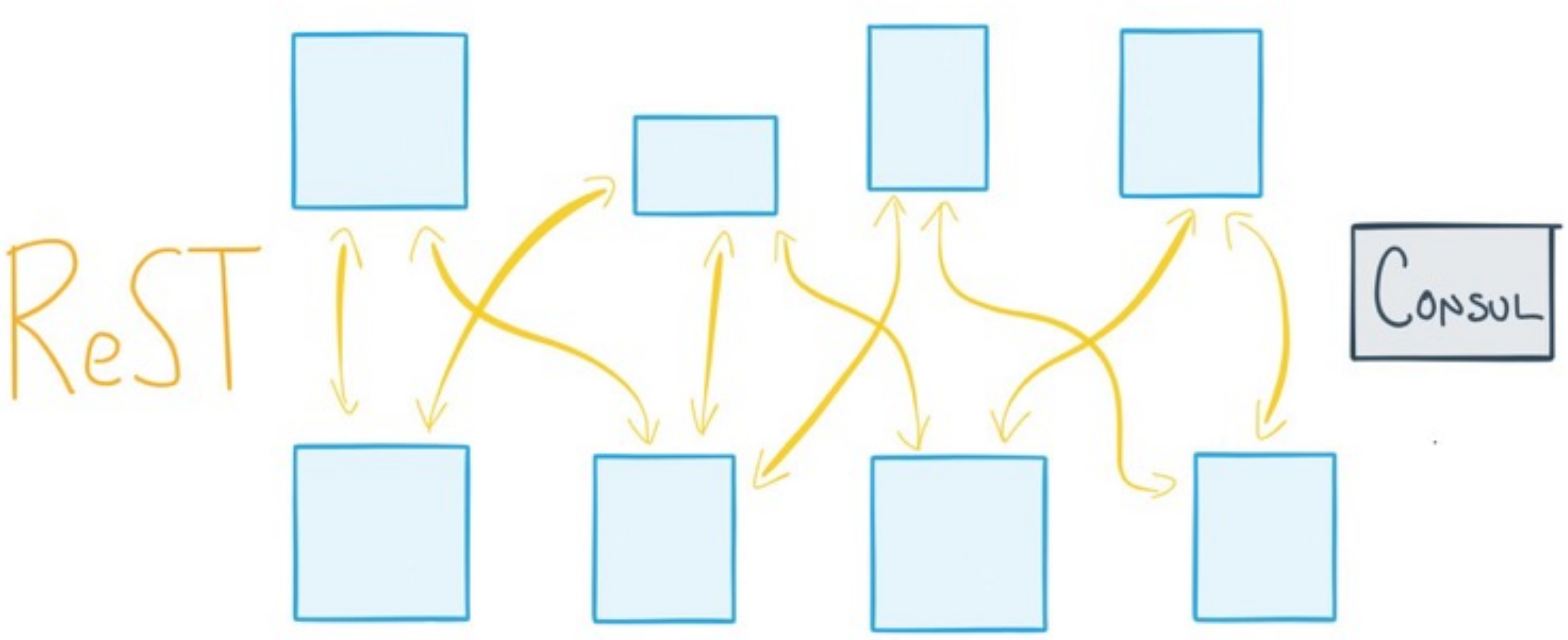


Stream Processing Toolset

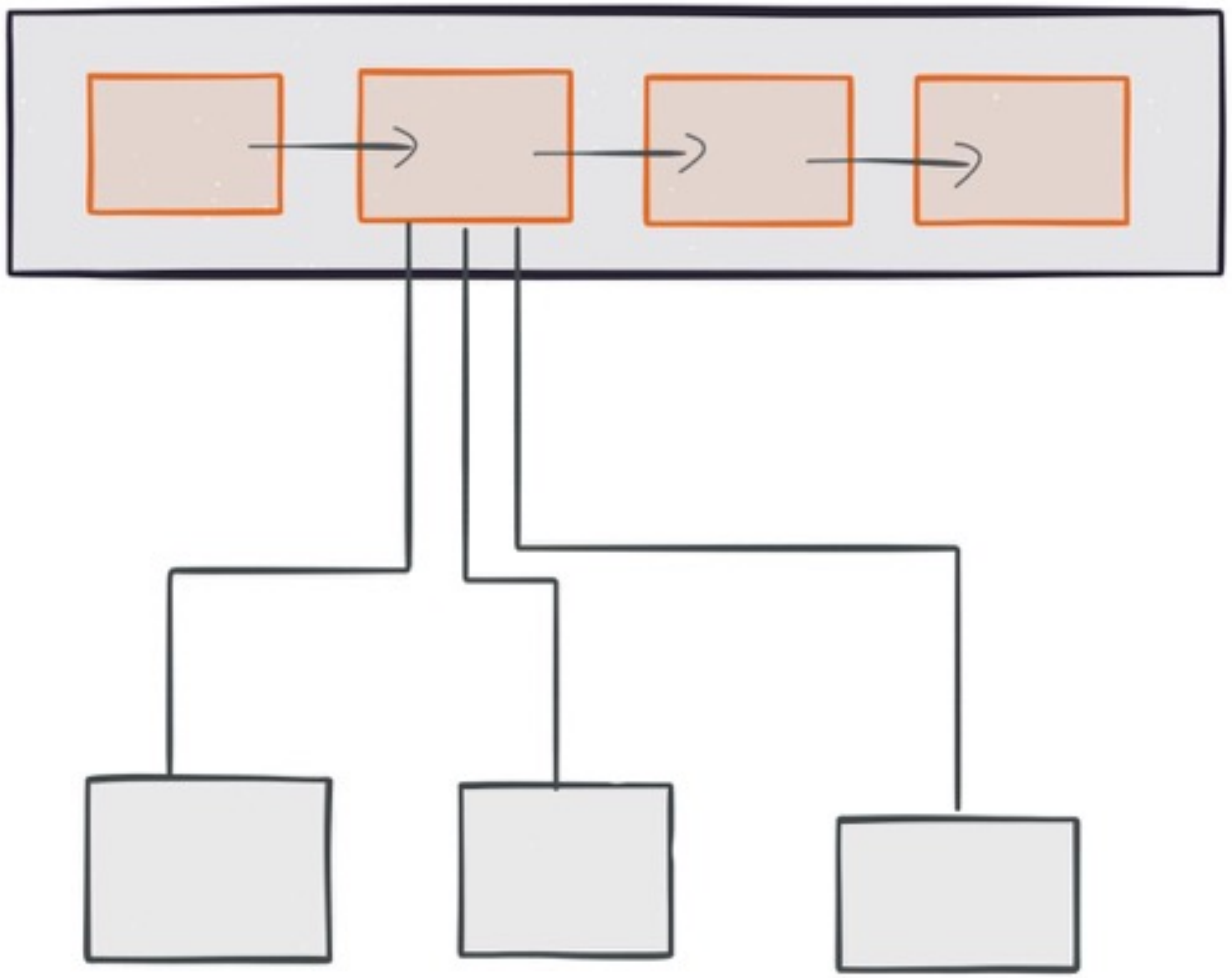
Business Applications

Some fundamental patterns of distributed systems

Request / Response

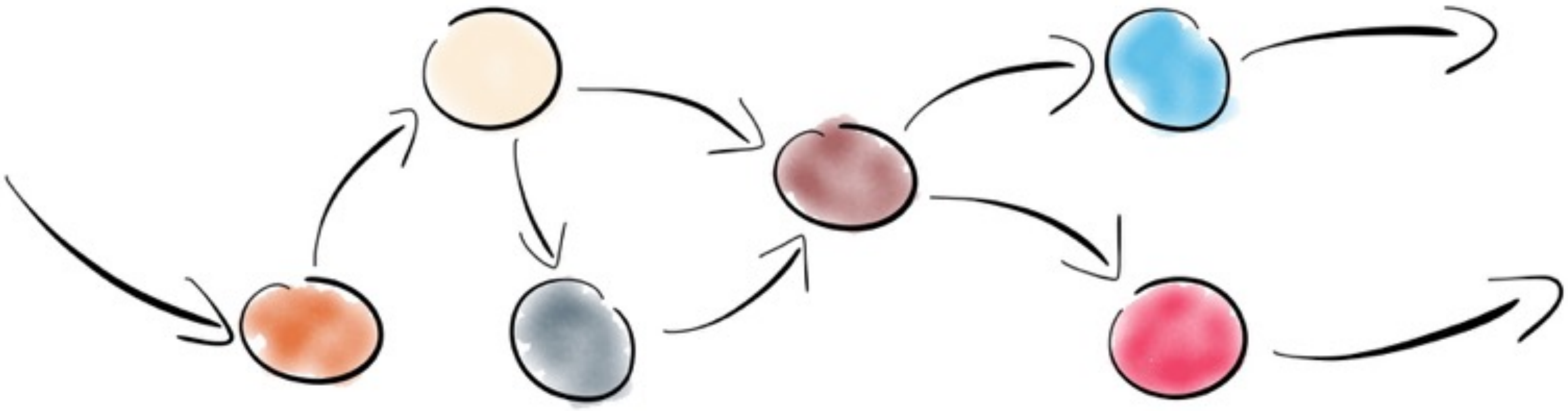


Mediator / Workflow



Request/Response

Event Driven

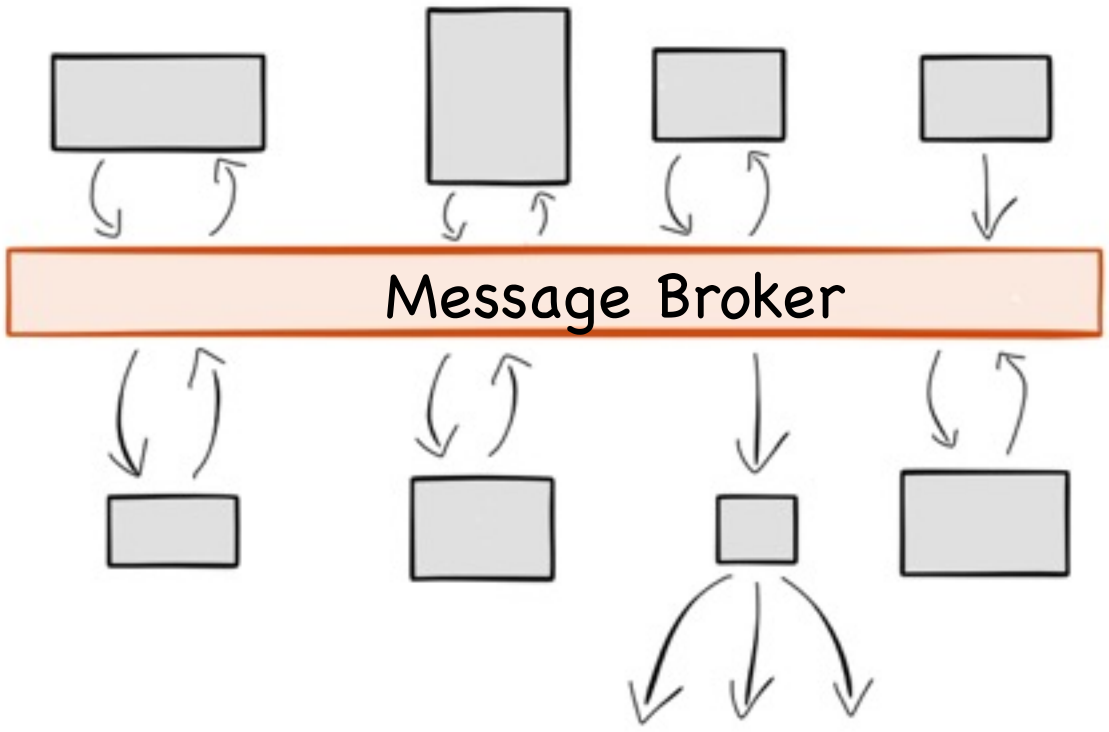


Async / Fire and Forget

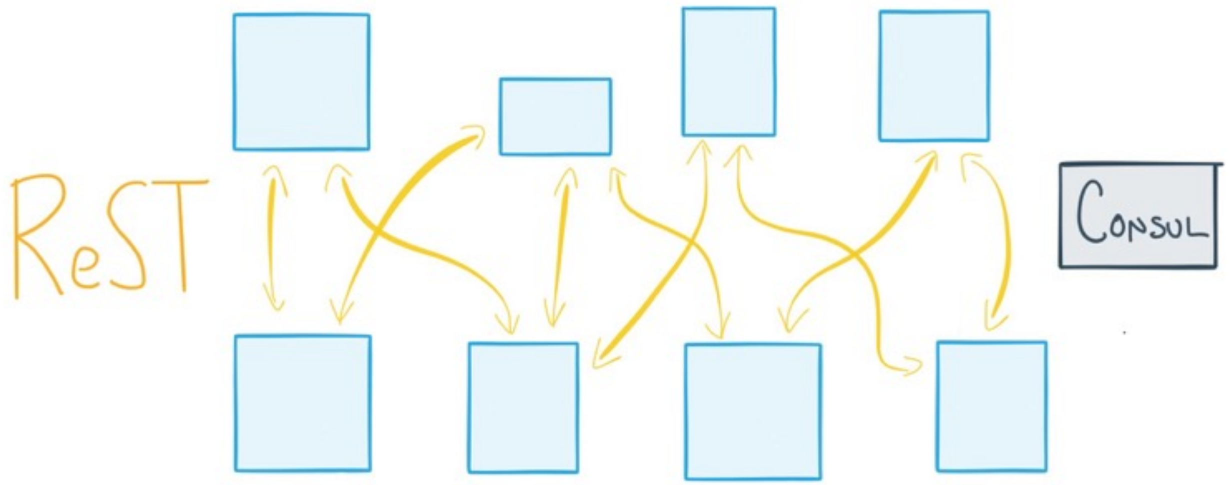
Request/Response vs. Event Based

• Simple	• Requires Broker
• Synchronous	• Fire & Forget
• Event Driven	• Polling
• Good decoupling	• Full decoupling

SOA / Microservices



Event Based

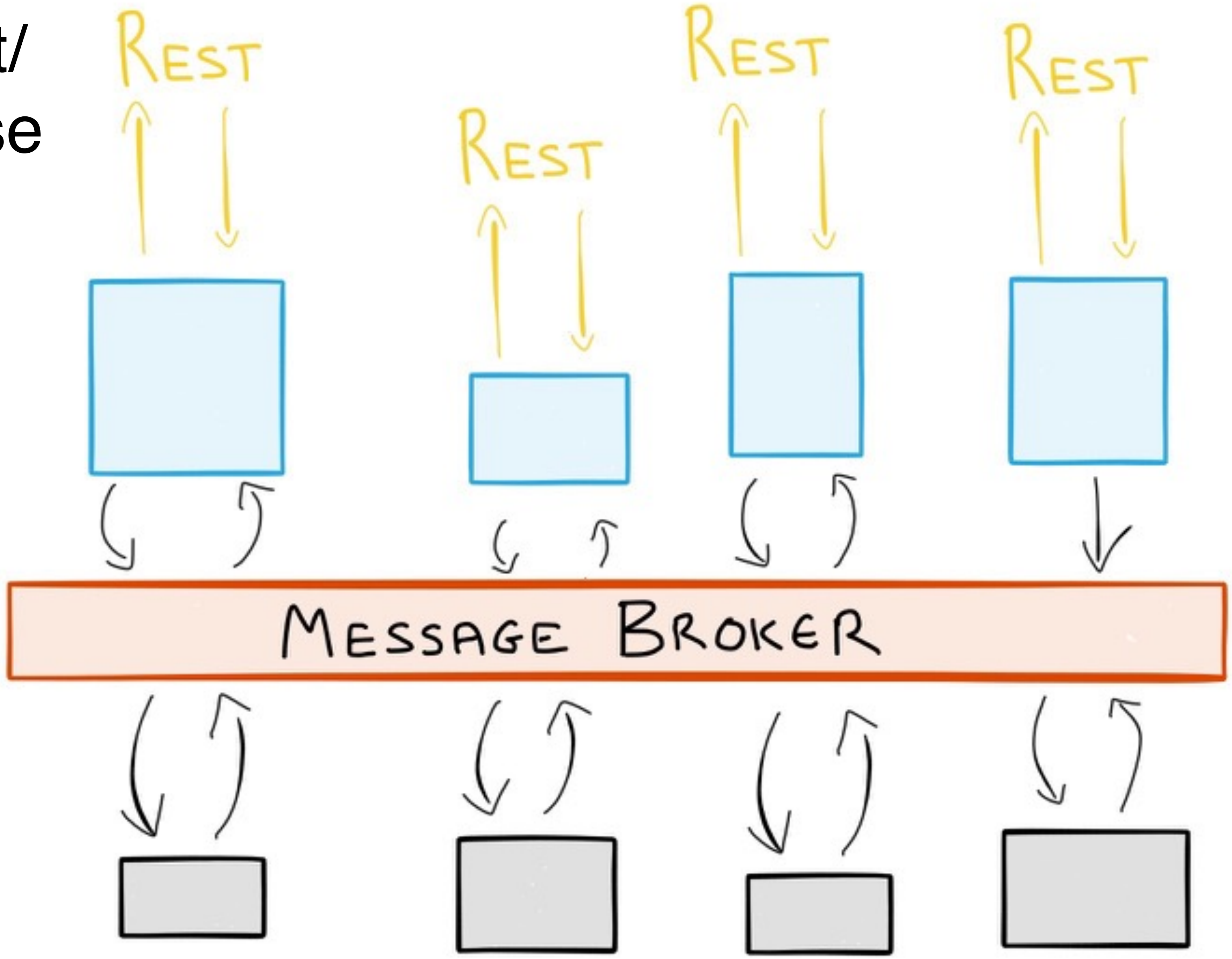


Request/Response

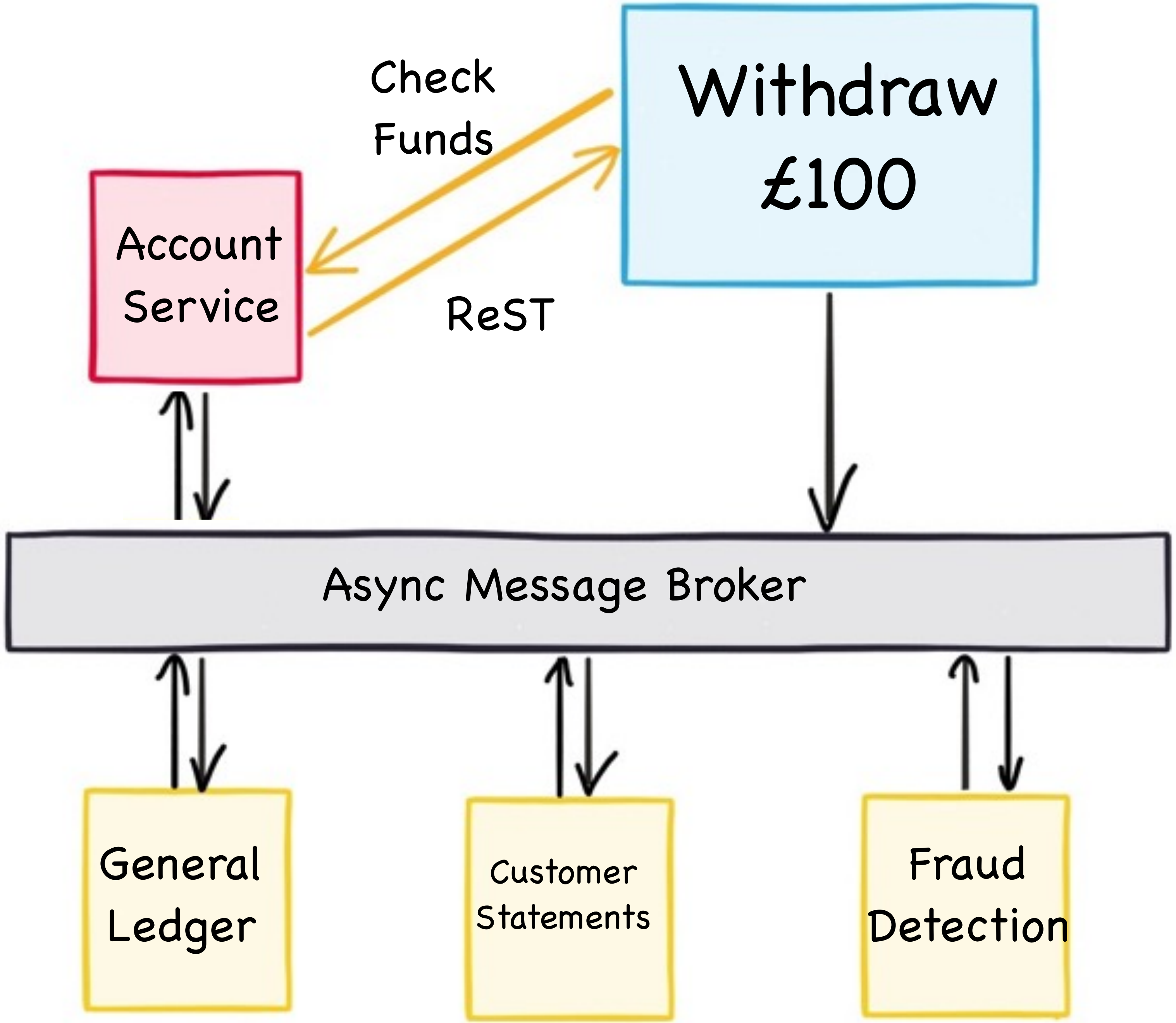
Combinations

Request/
Response

Event-
Based

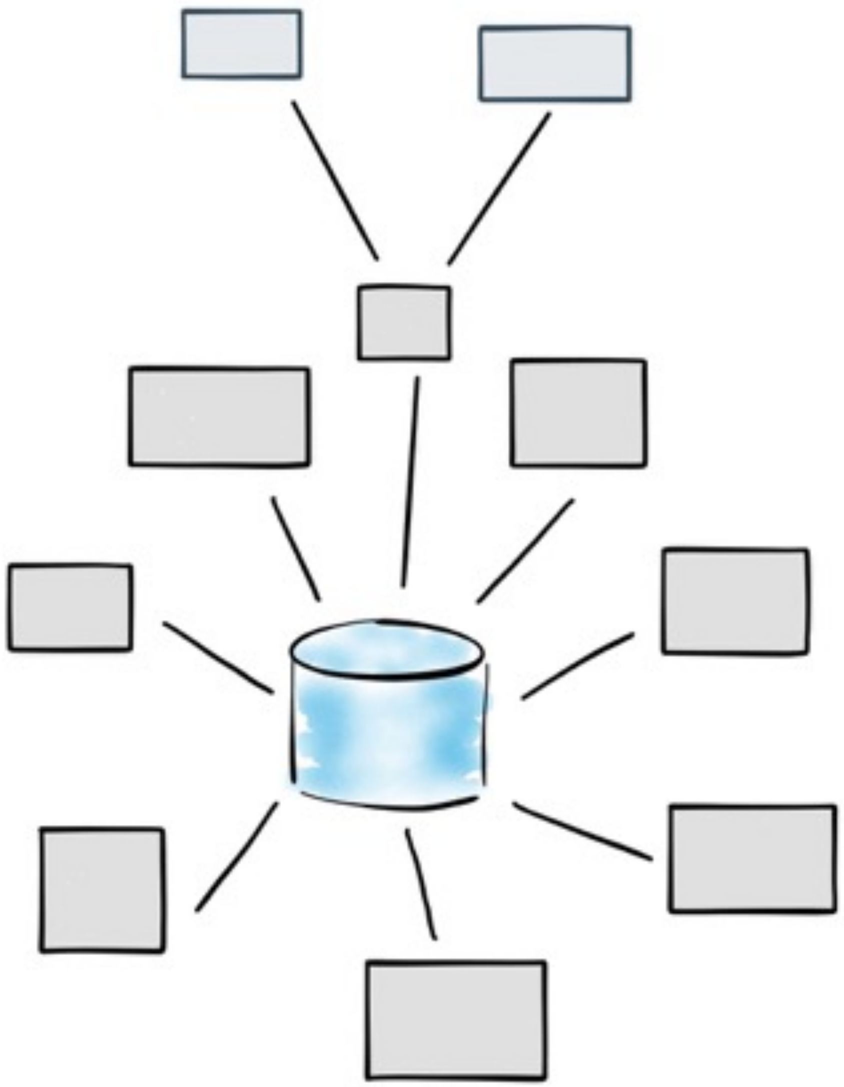


Combinations



I need money

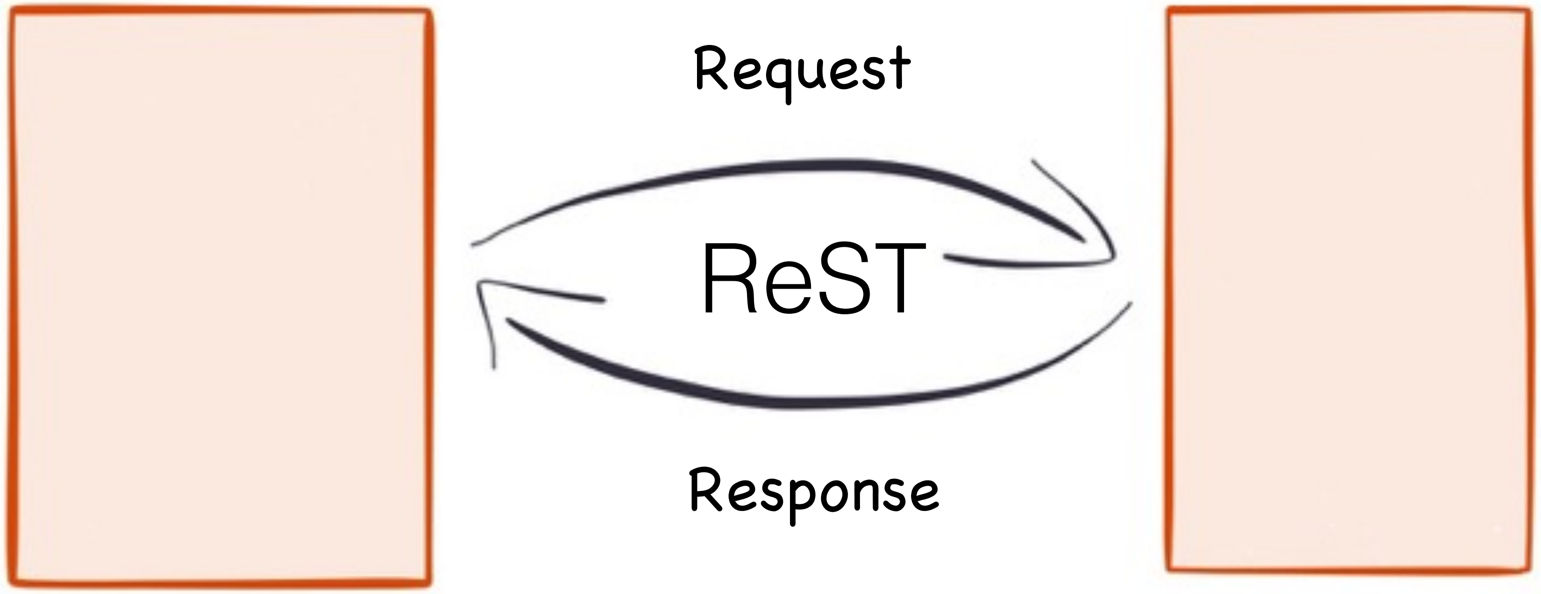
Services generally eschew shared, mutable state



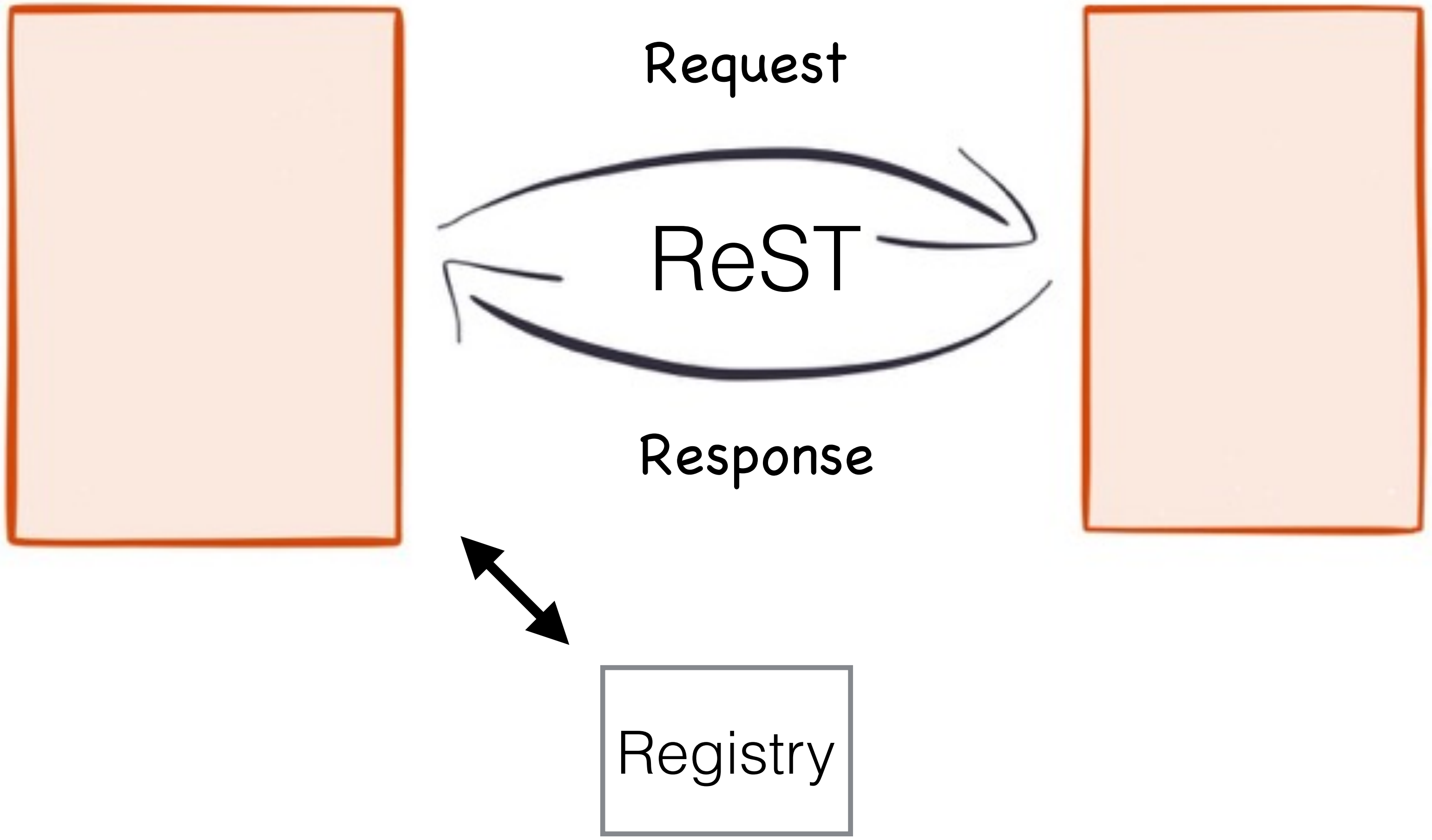
How do we put these things together?

Request/Response

Request/Response



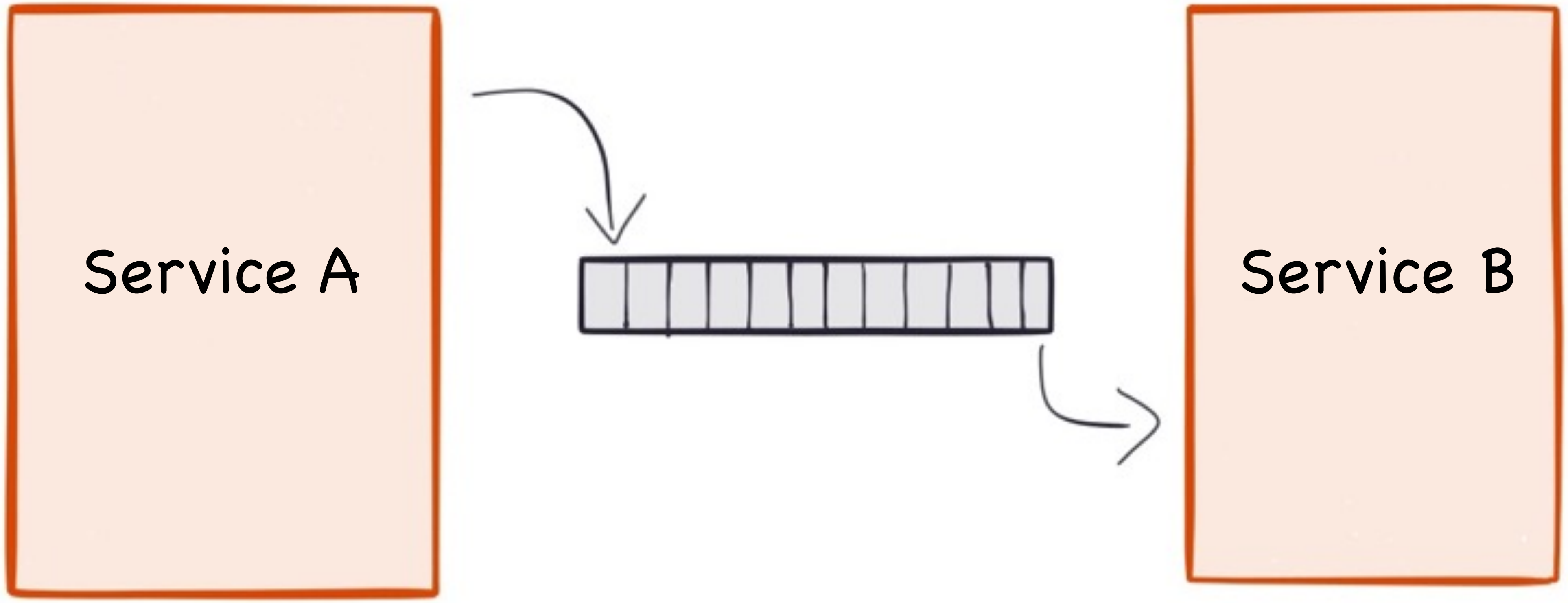
Request/Response + Registry



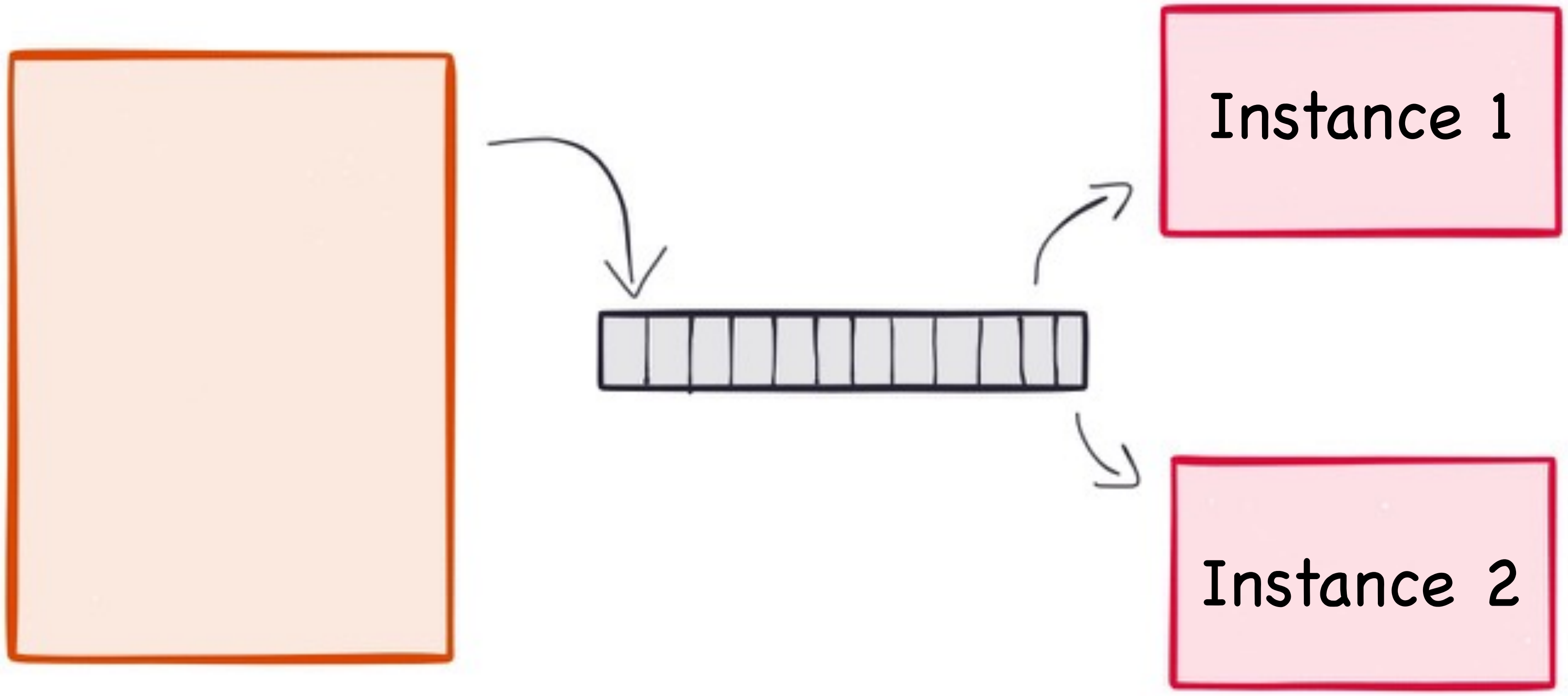
Asynchronous and Event-Based Communication

Queues

Point to Point

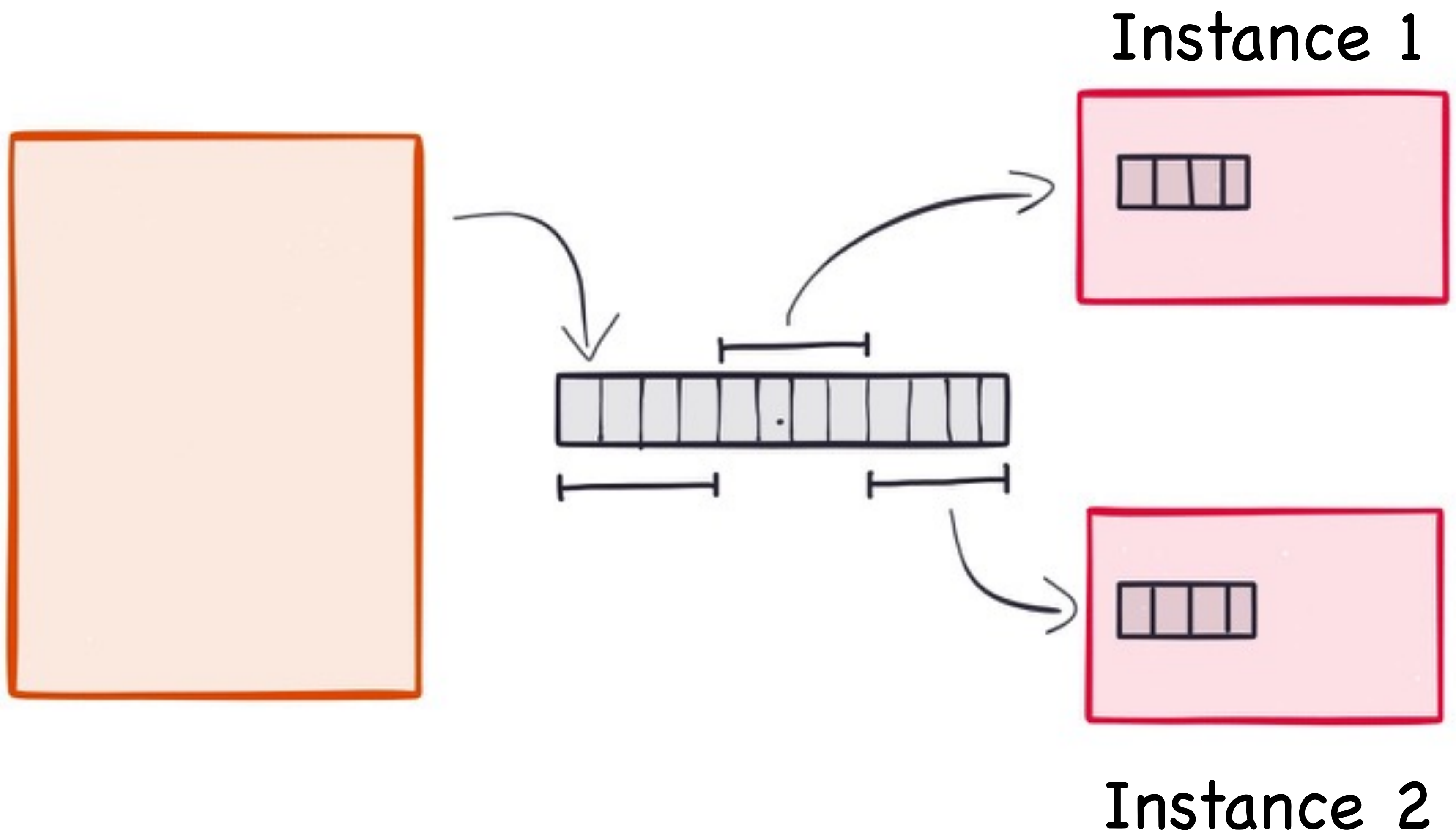


Load Balancing



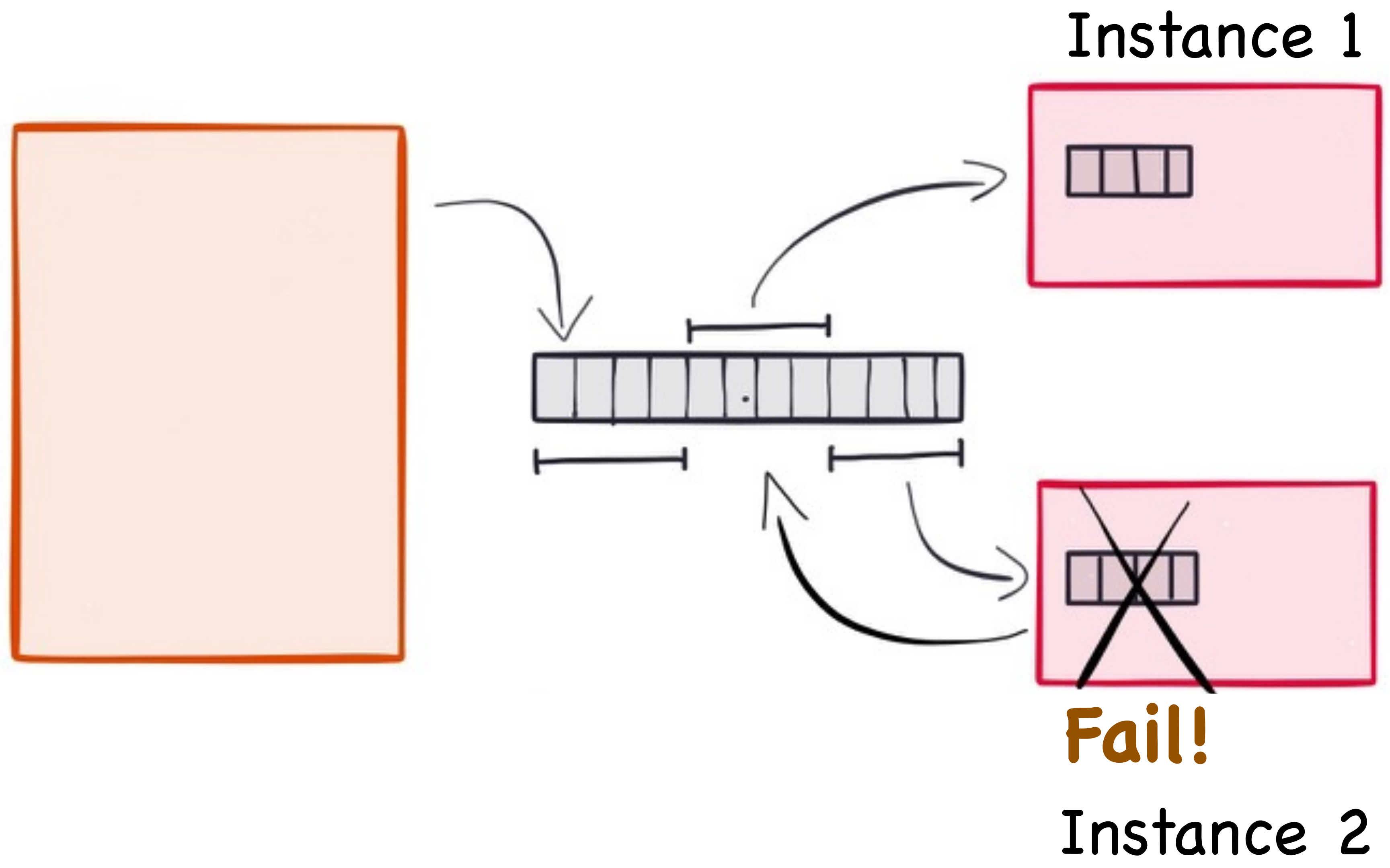
Single message allocation has scalability issues

Batched Allocation



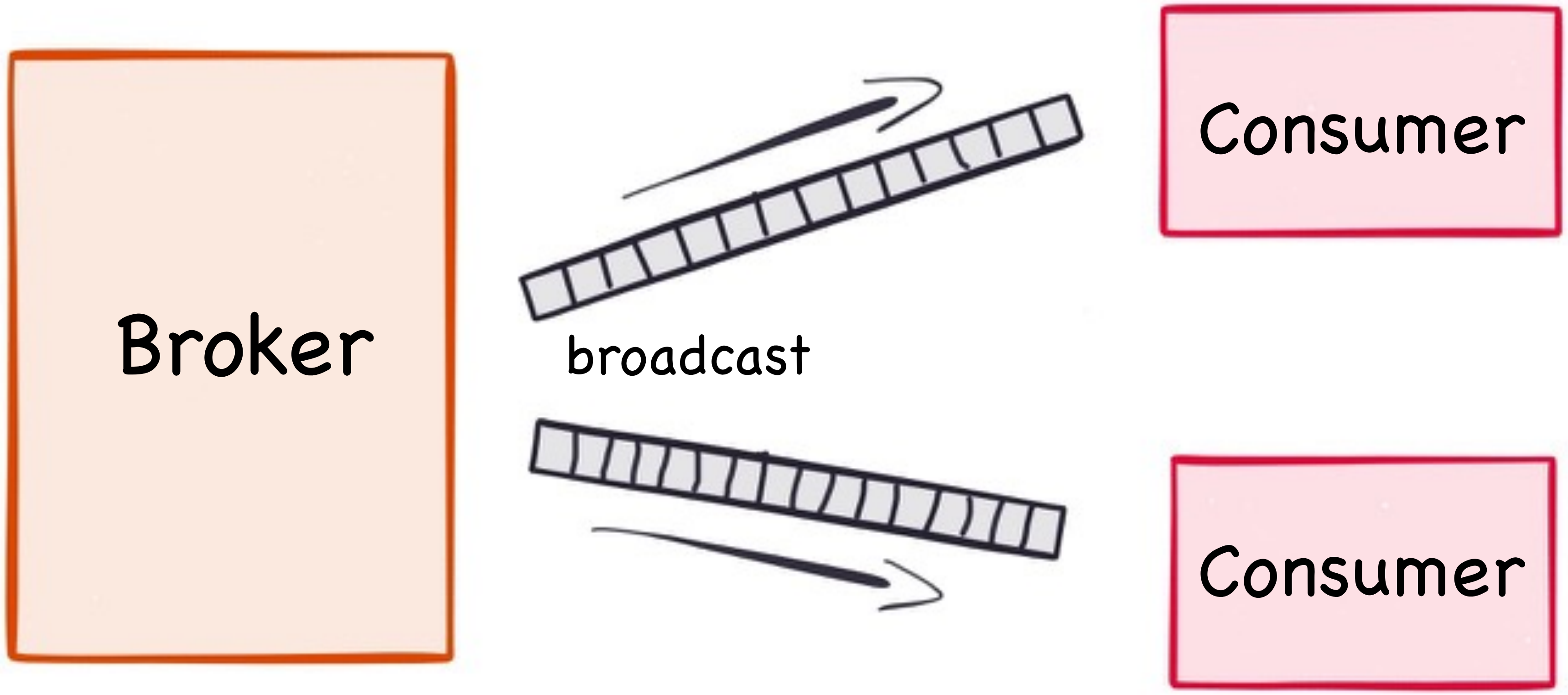
Throughput!

Lose Ordering Guarantees

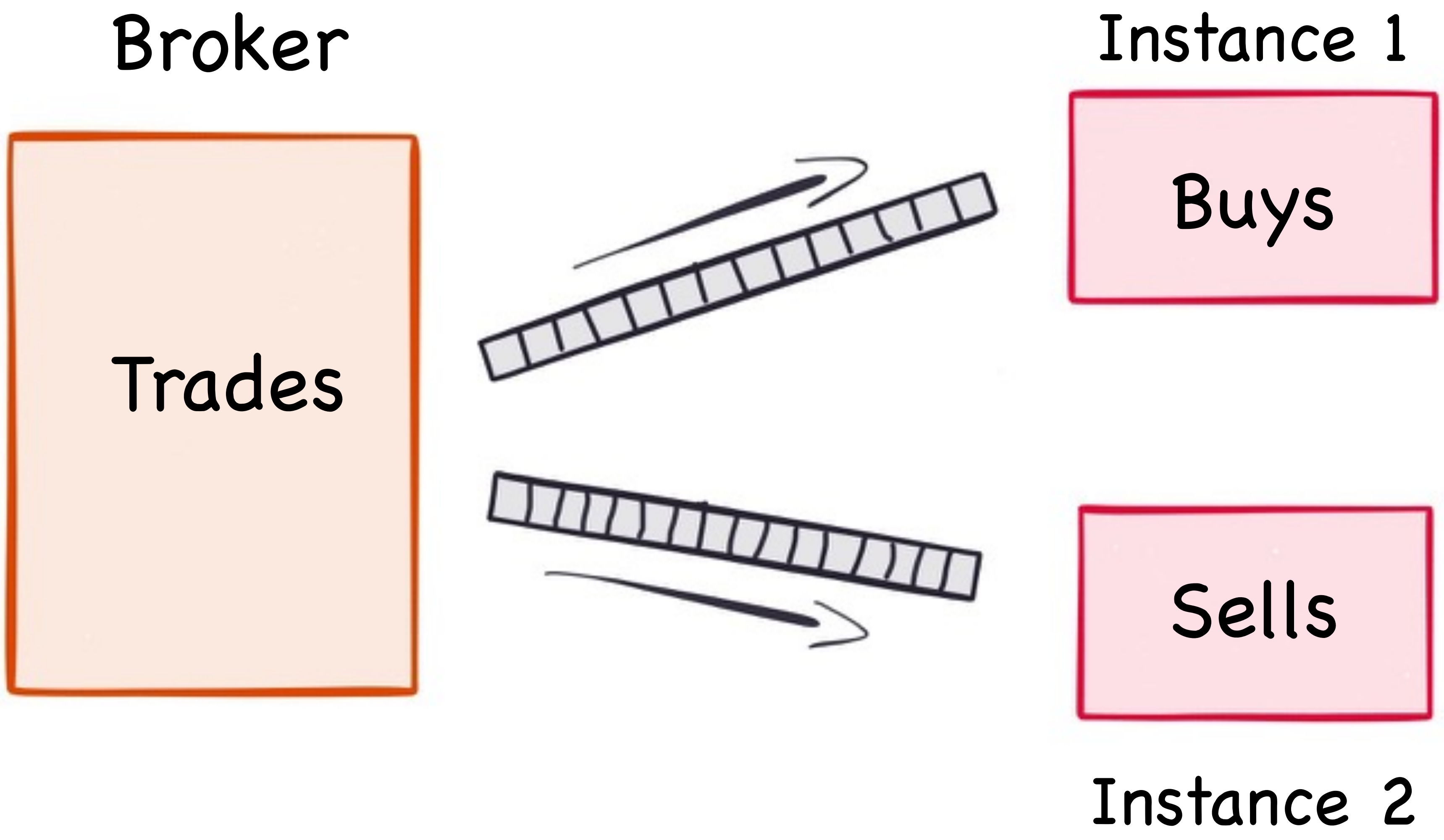


Topics

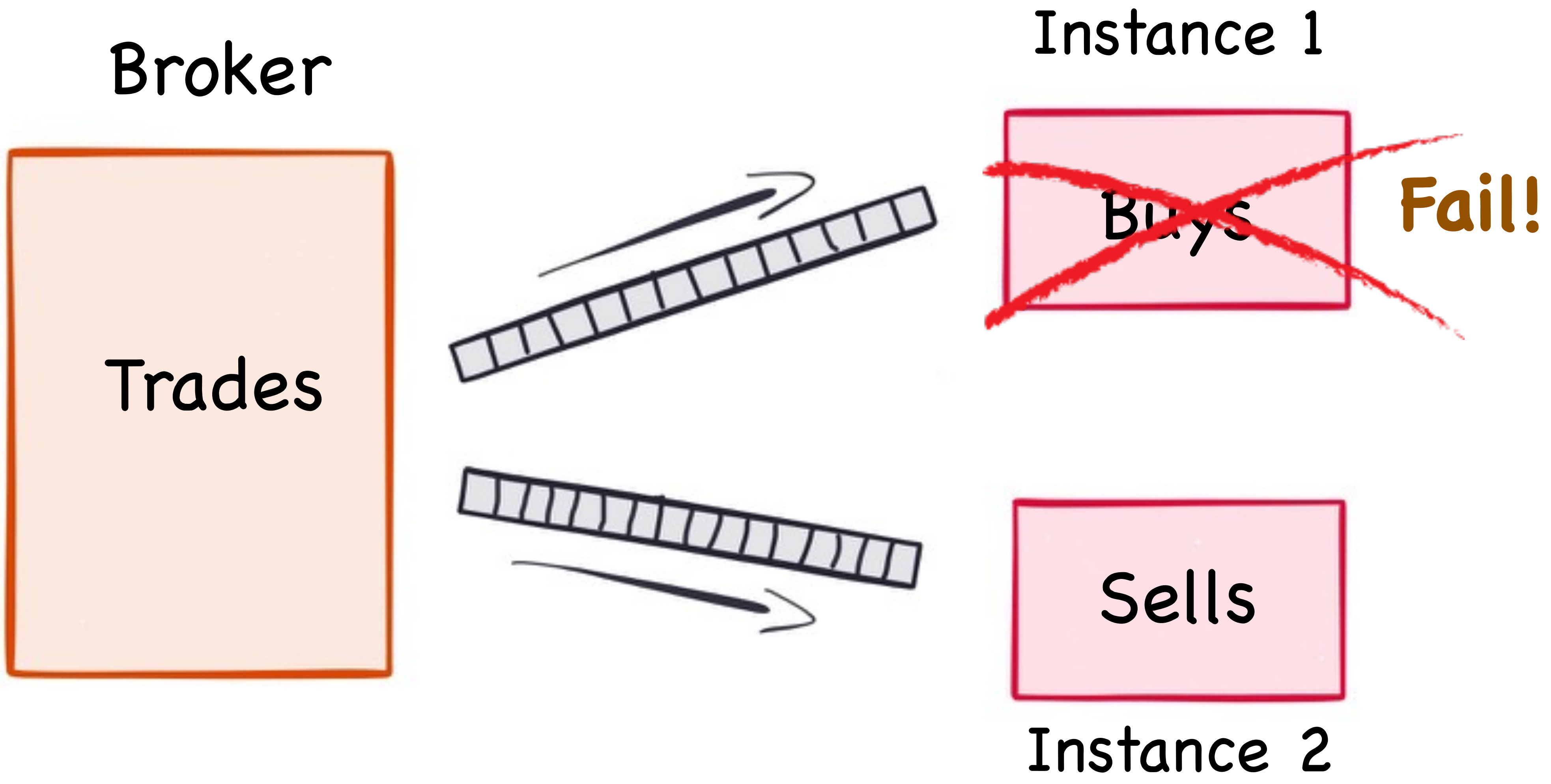
Topics are Broadcast



Topics Retain Ordering



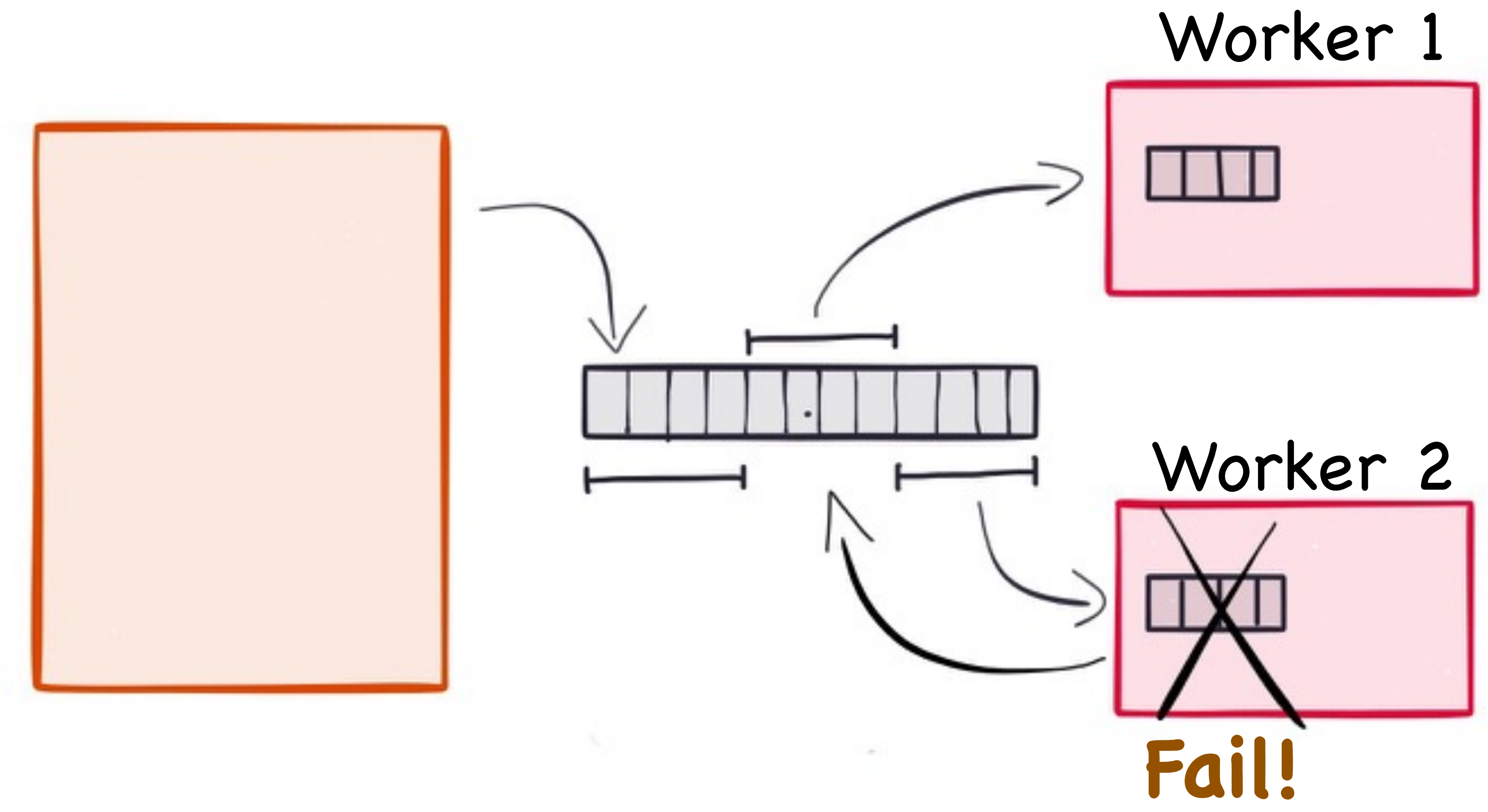
Even when services fail



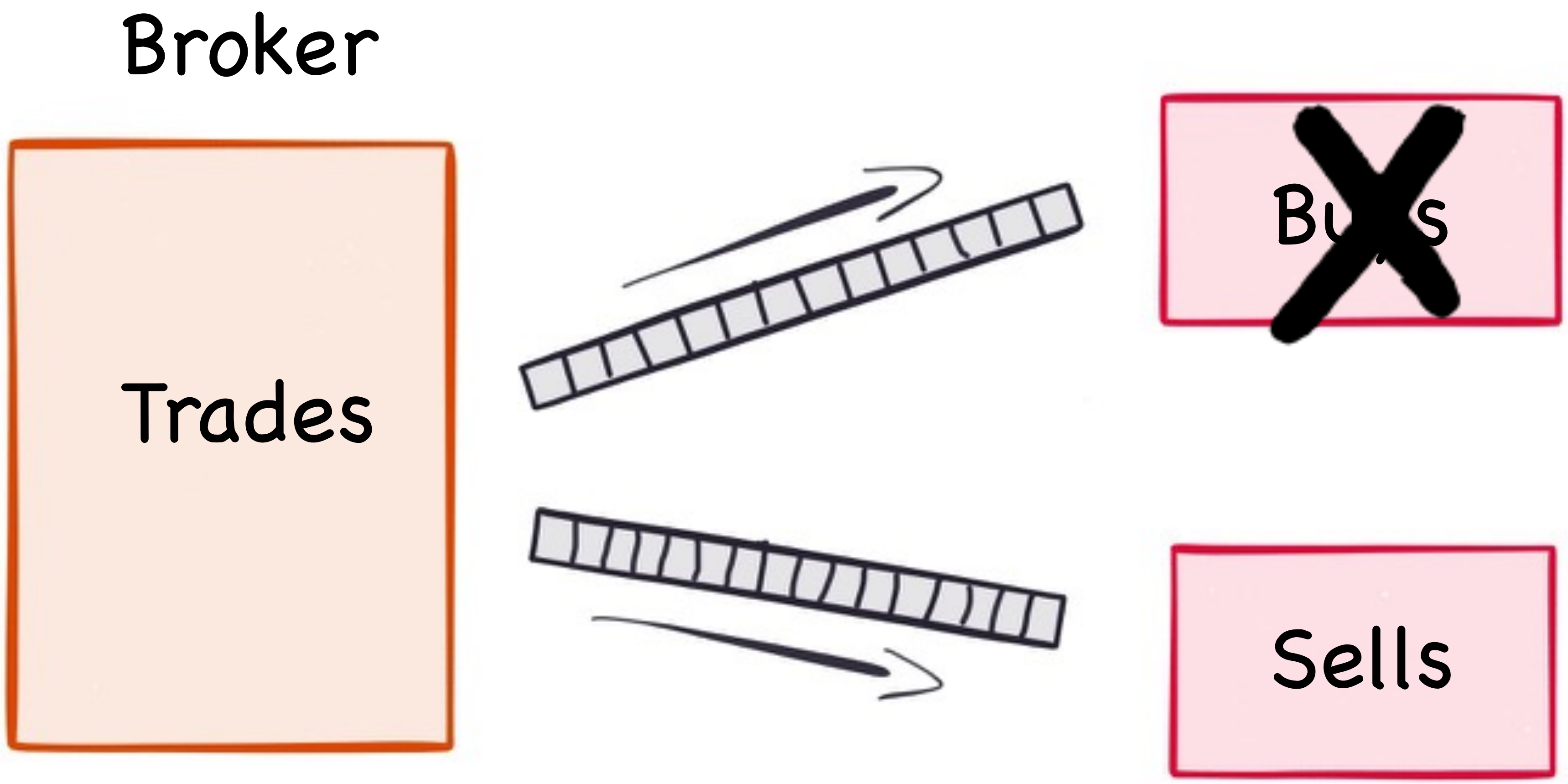
We retain ordering, but we have to detect & reprovision

A Few Implications

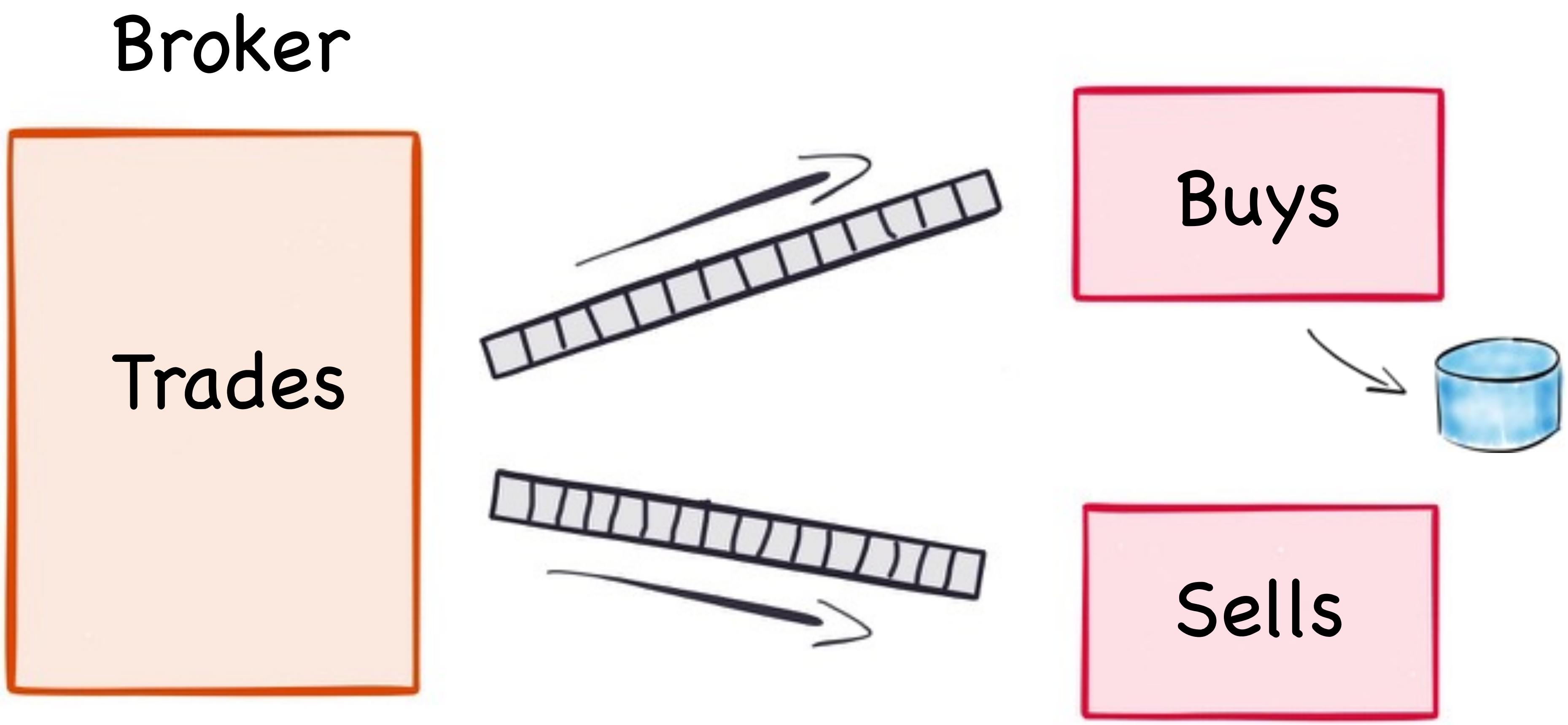
Queues Lose Ordering Guarantees at Scale



Topics don't provide availability



Messages are Transient

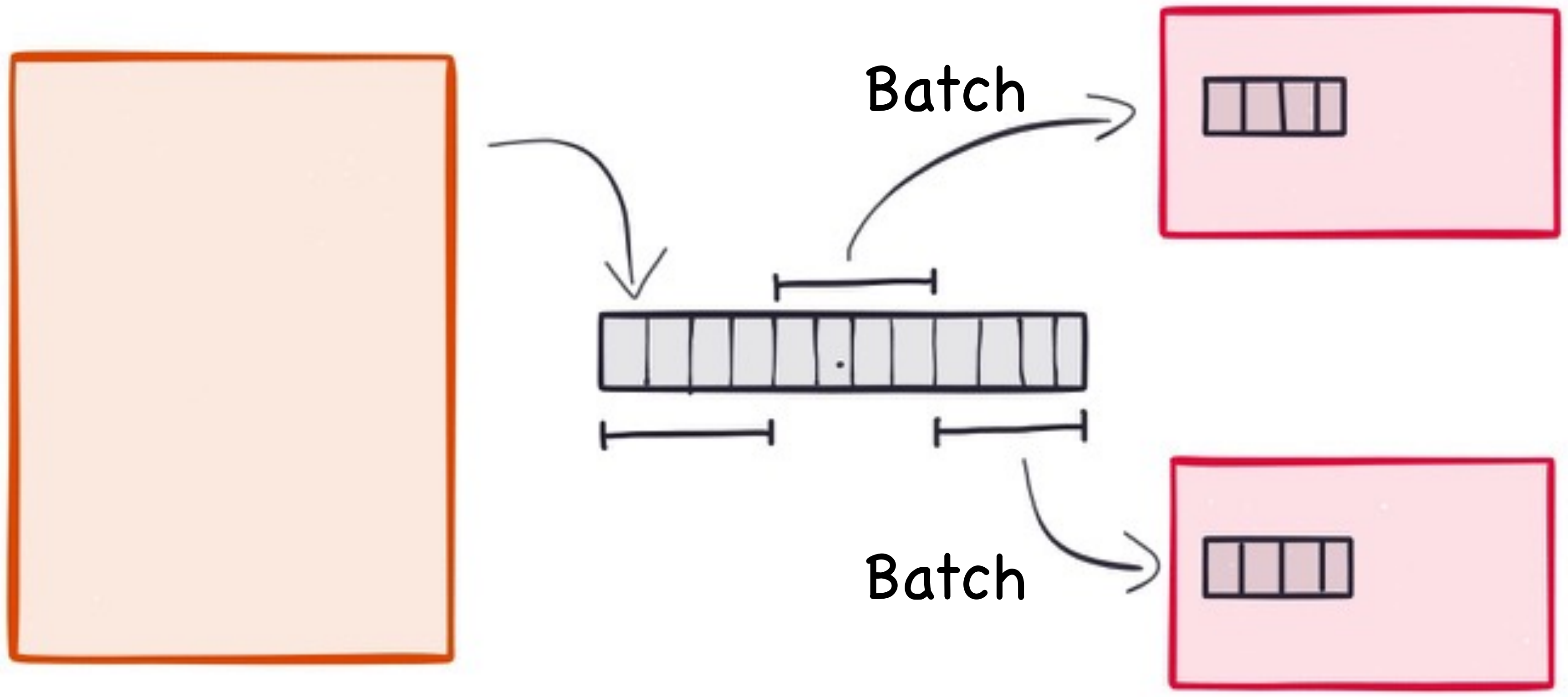


Is there another way?

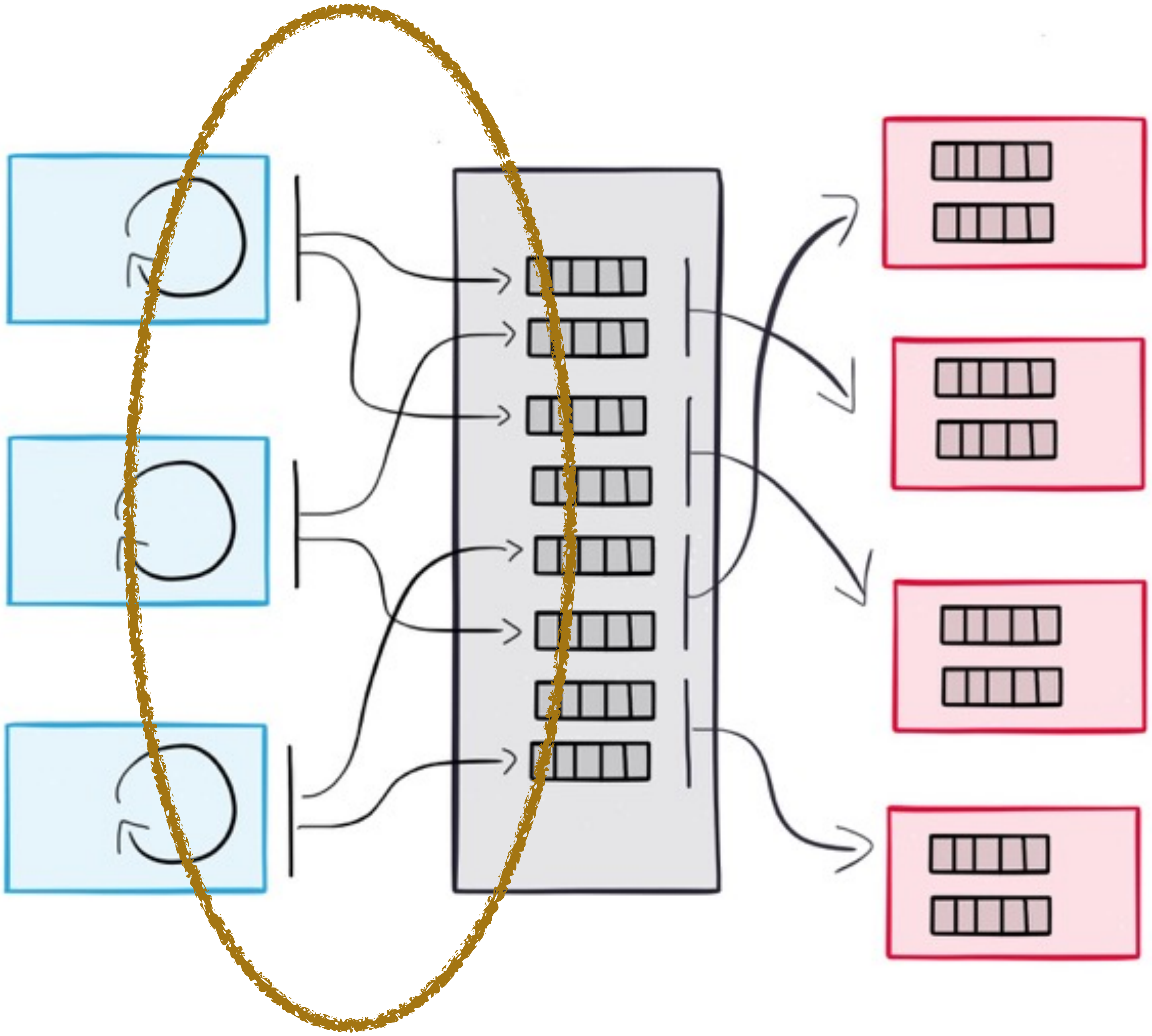
A Distributed Log

Kafka is one example

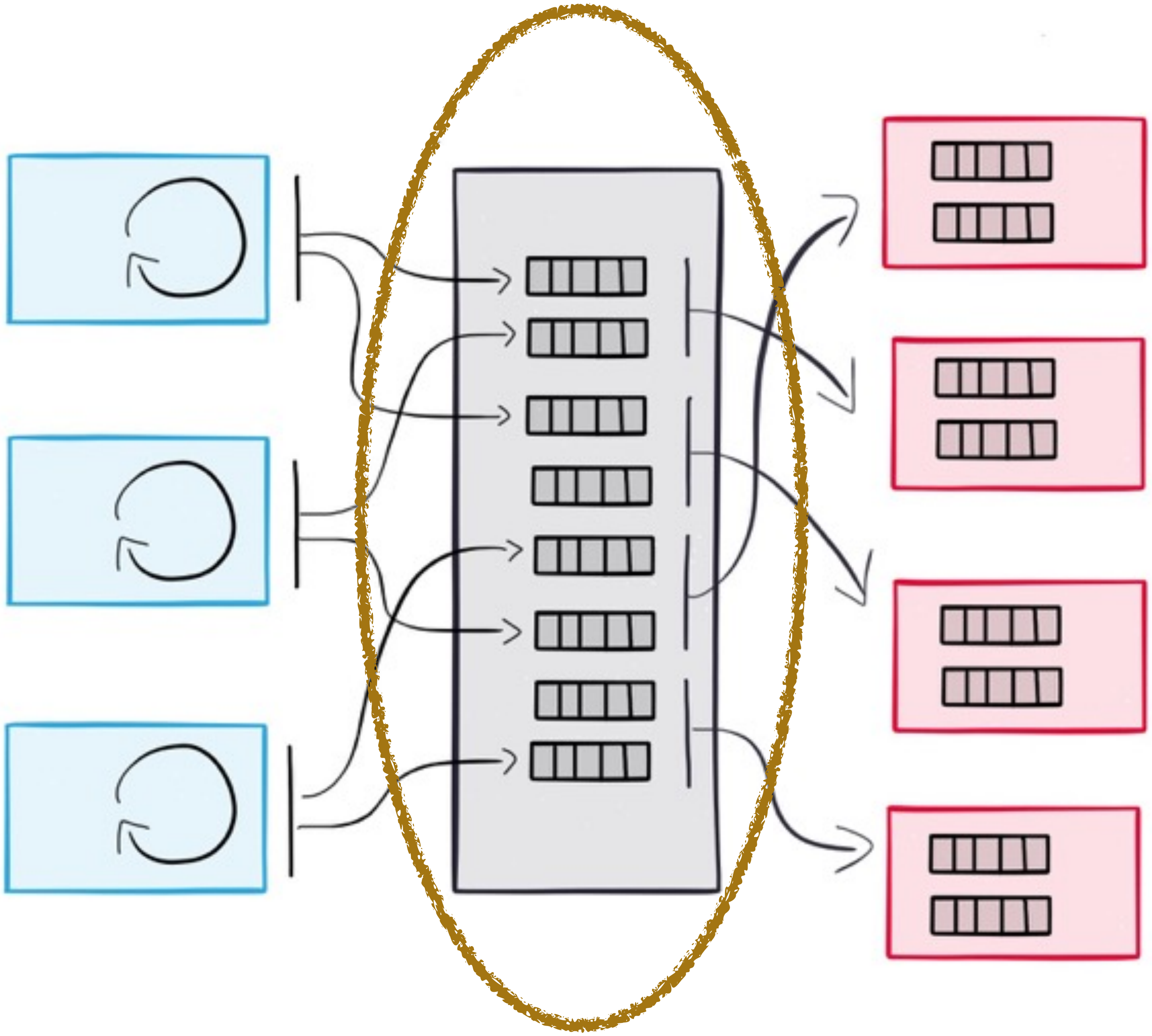
Think back to the queue example



Shard on the way in

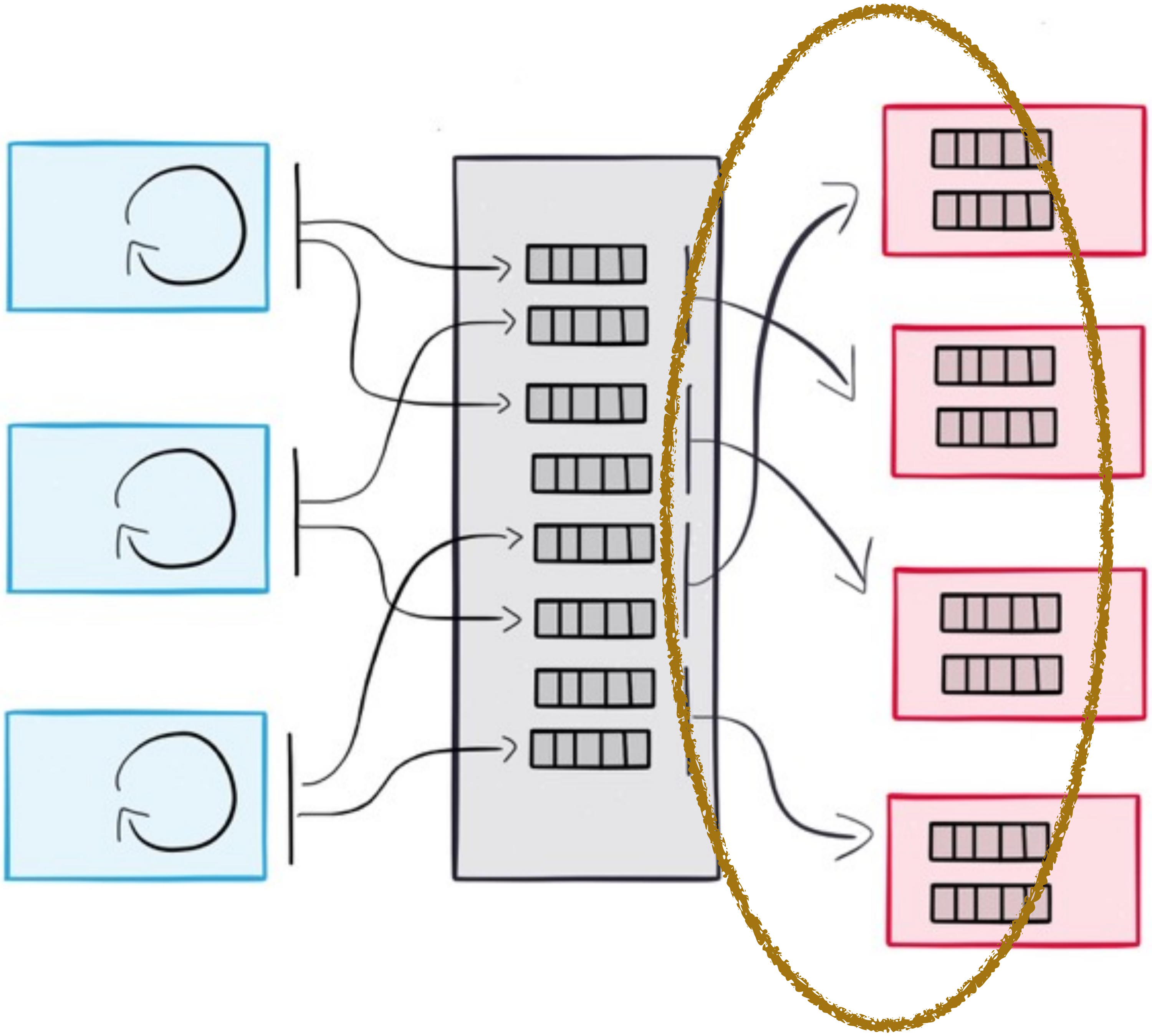


Each shard is a queue



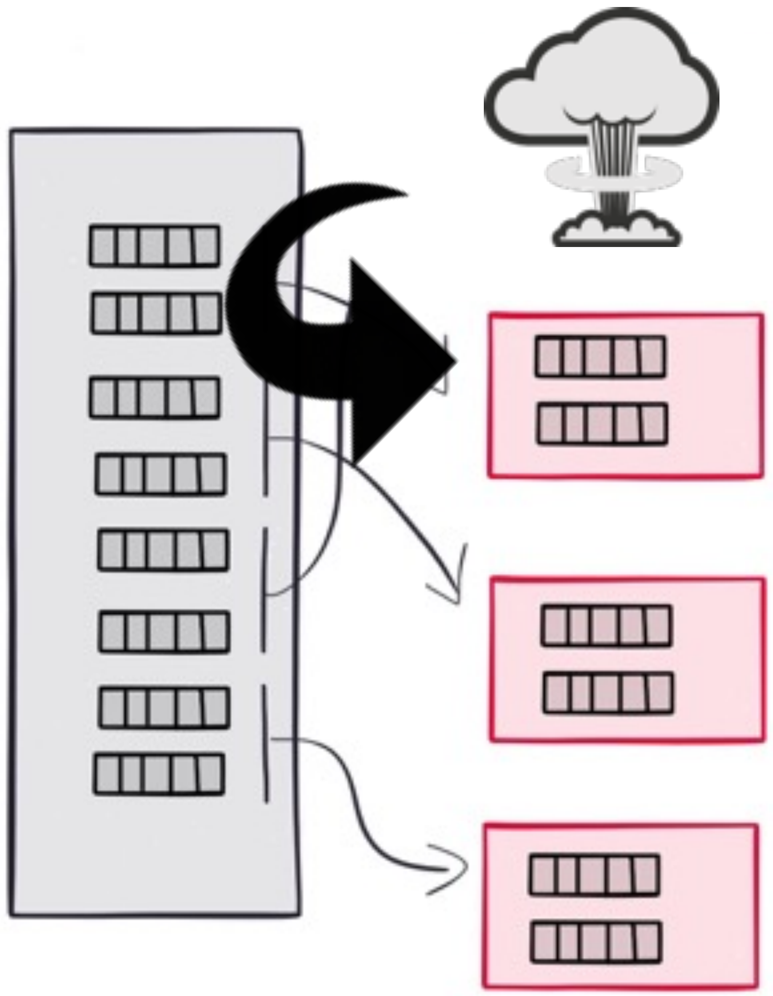
Strong Ordering (in shard). Good concurrency.

Each consuming service is assigned a “personal set” of queues



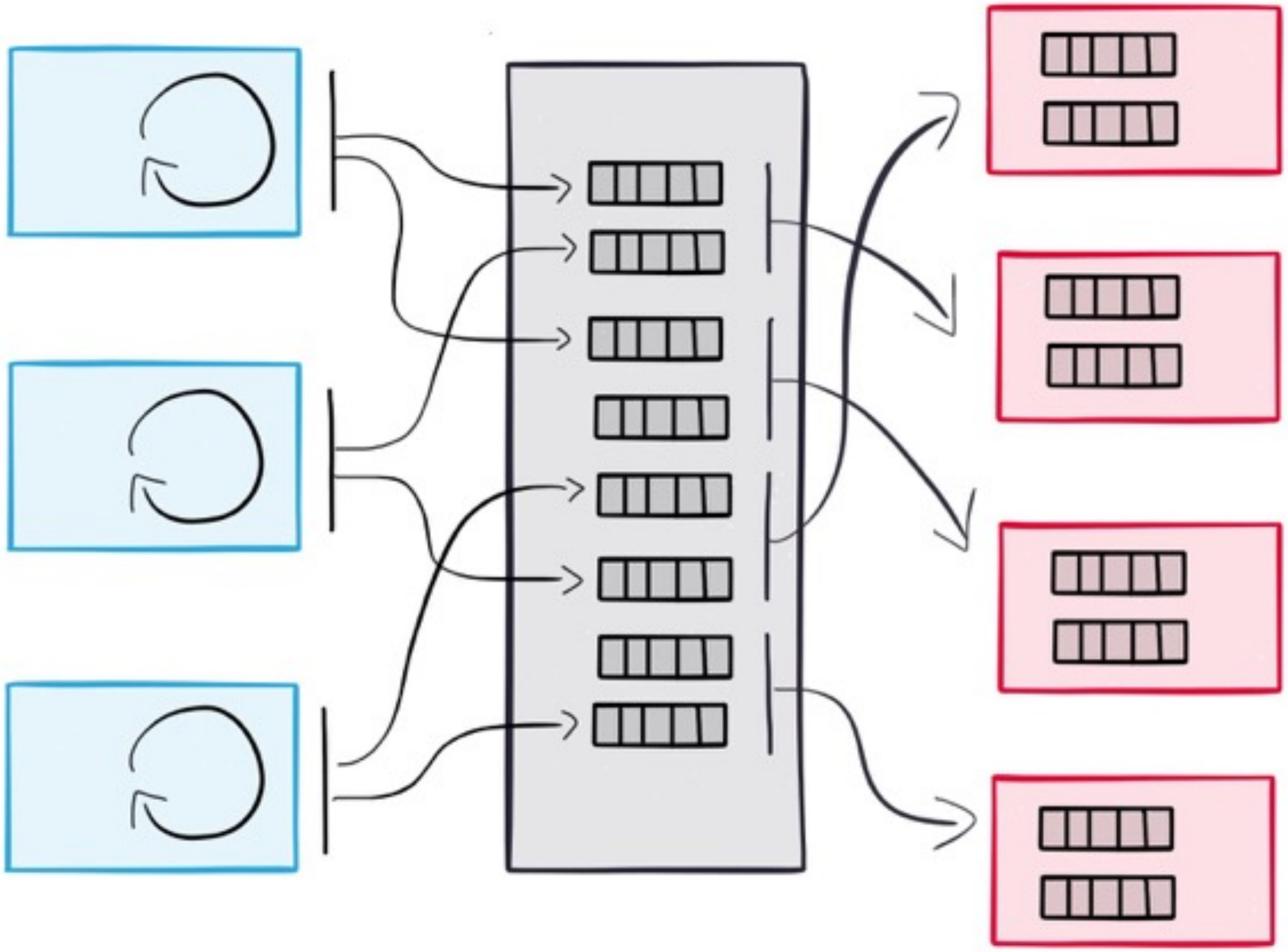
each little queue is sent to only one service in a group

Services instances naturally rebalance on failure



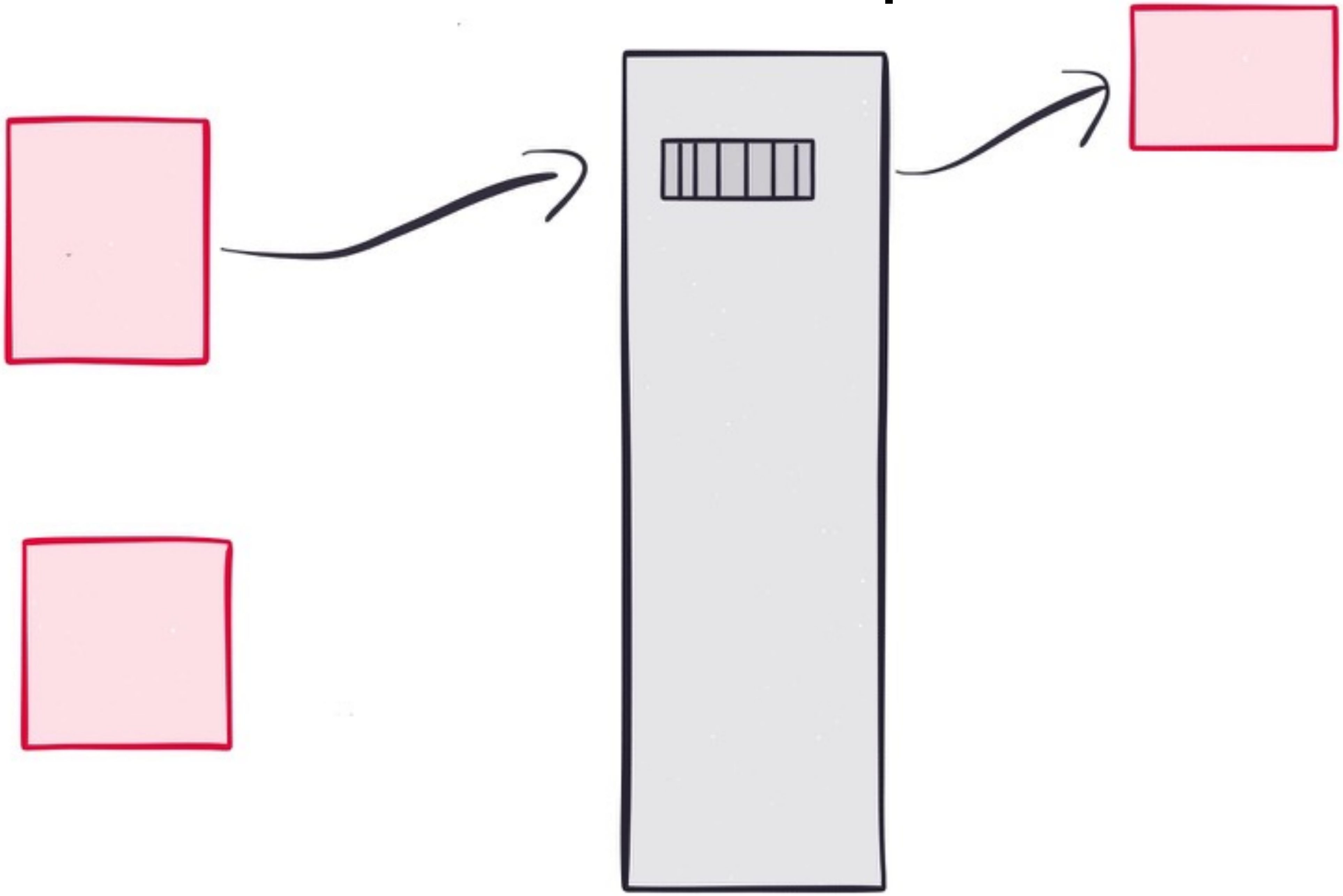
Service instance dies, data is redirected, ordering guarantees remain

Very Scalable, Very High Throughput



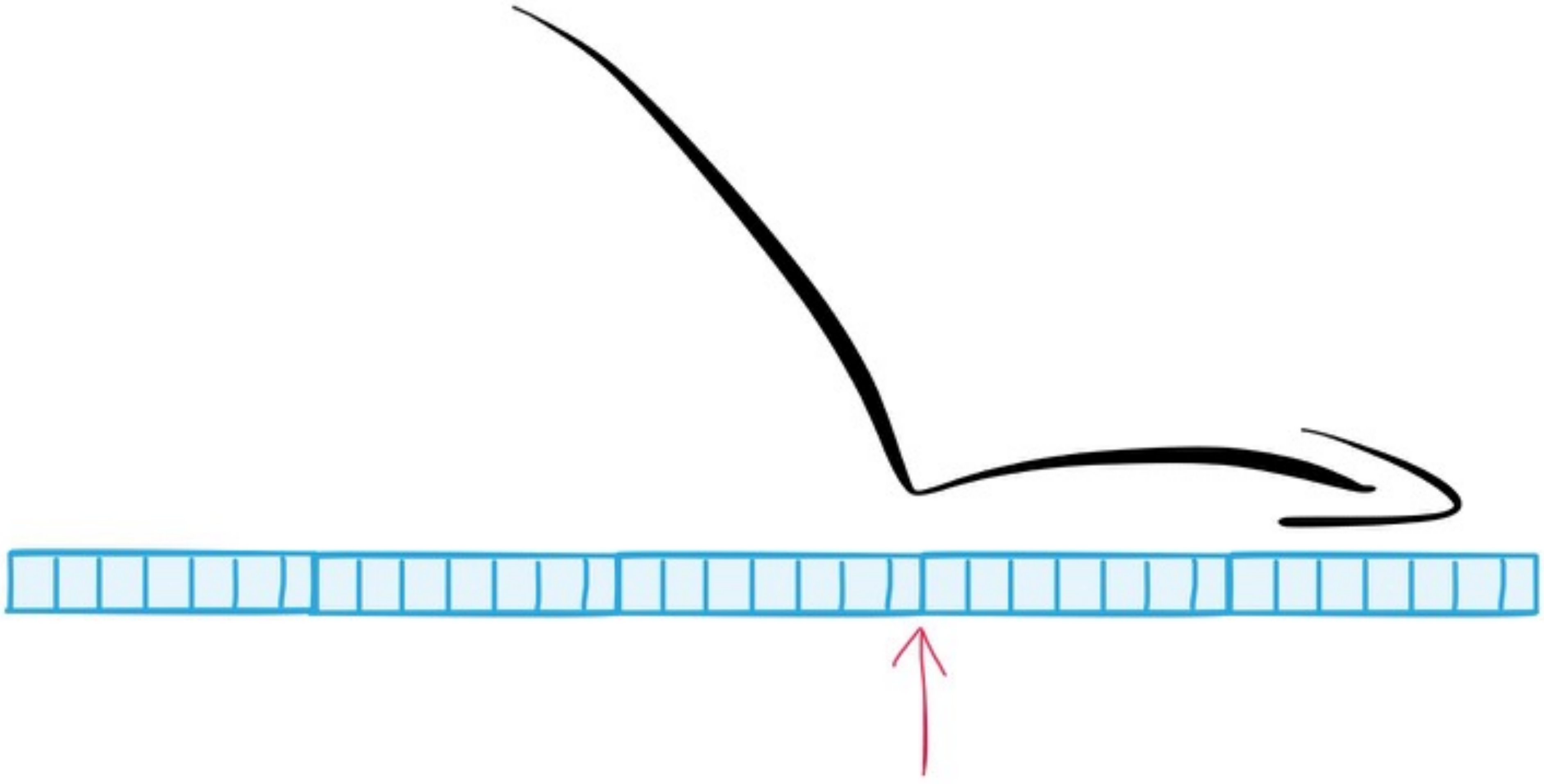
Sharded In, Sharded Out

Reduces to a globally ordered queue



Fault Tolerance

The Log

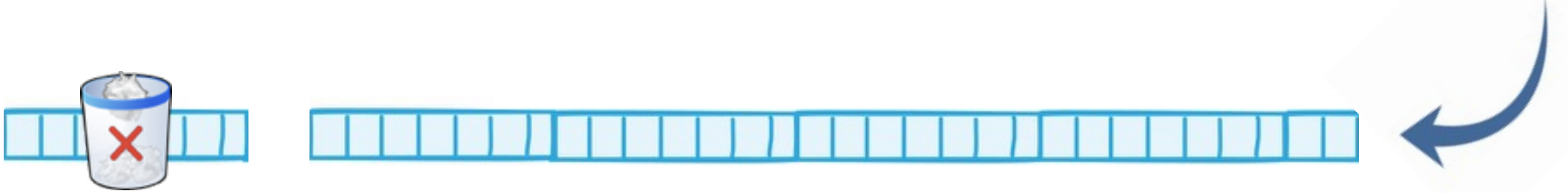


Append only

Single seek & scan

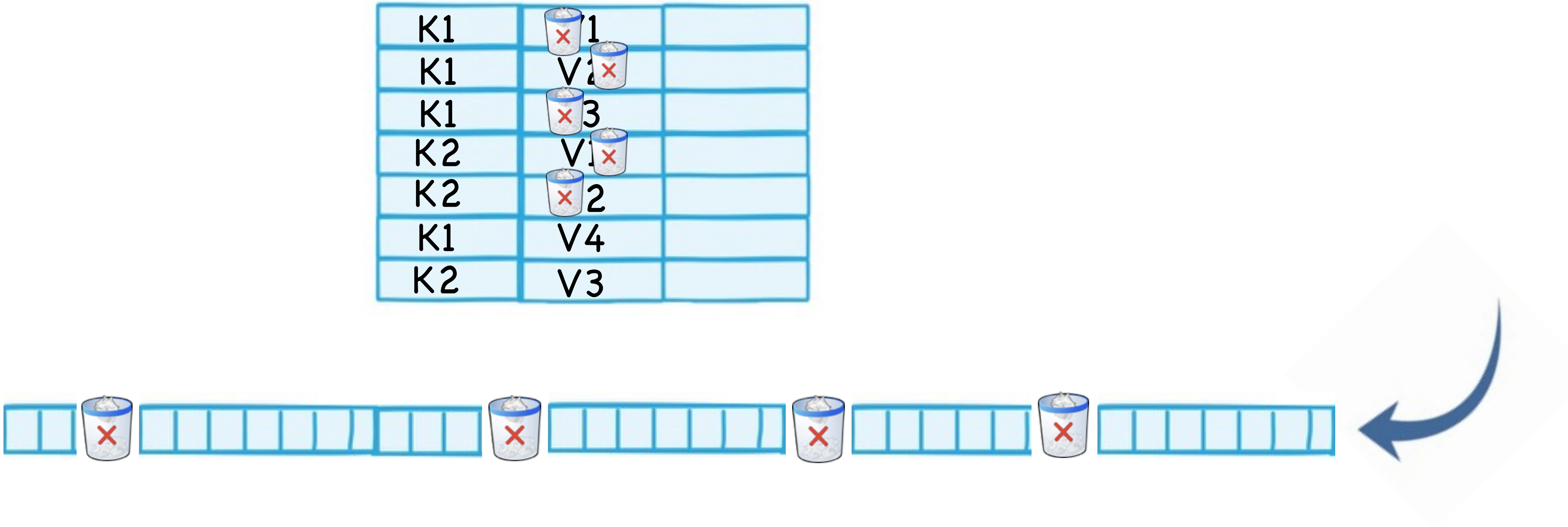
messages don't need to be transient!

Cleaning the Log



Delete old segments

Cleaning the Log



K1	1	
K1	V2	
K1	3	
K2	V1	
K2	2	
K1	V4	
K2	V3	

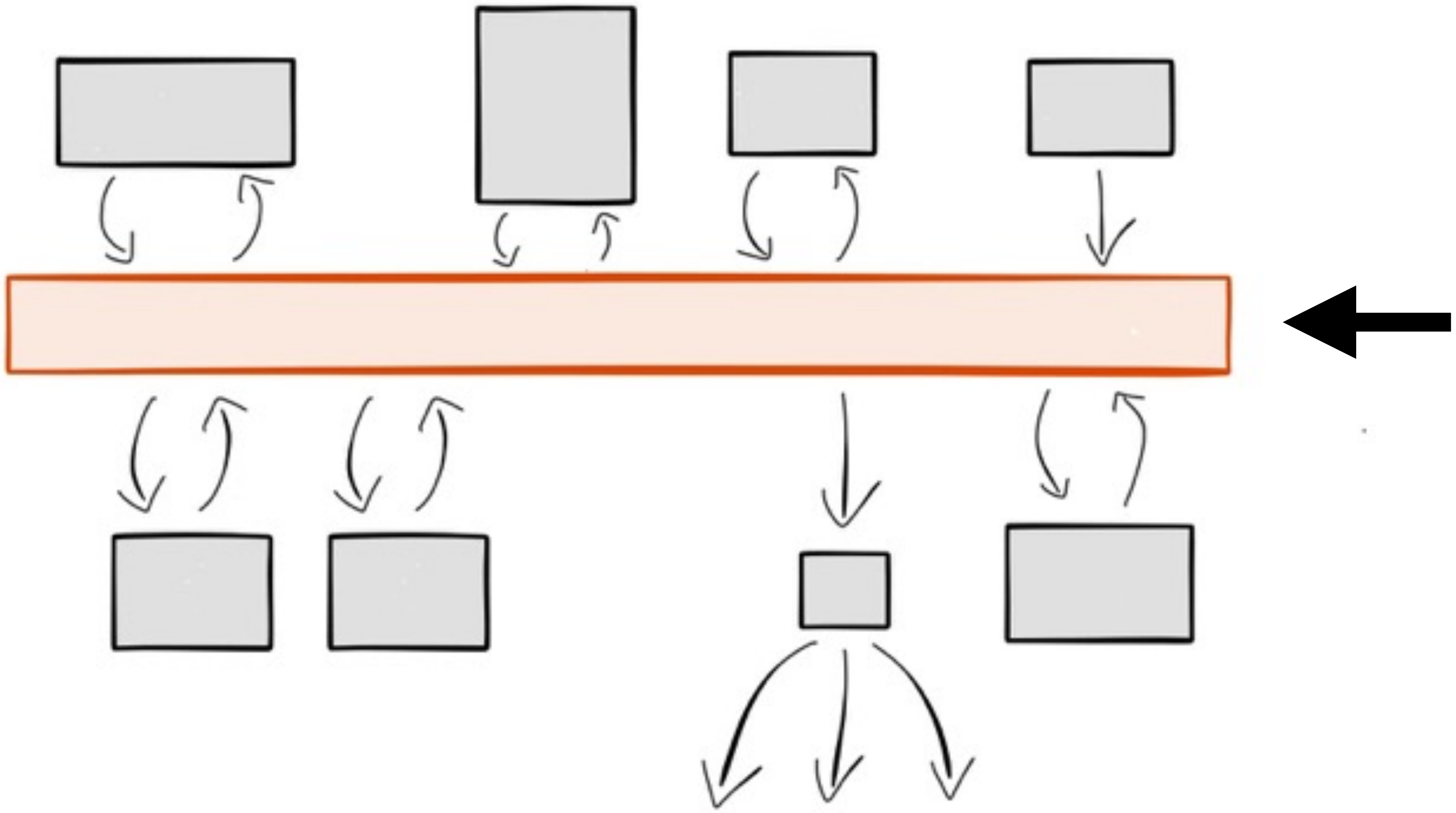
Delete old versions that share the same key

KAFKA

- Scalable multiprocessing ✓
- Strong partition-based ordering ✓
- Efficient data retention ✓
- Always on ✓

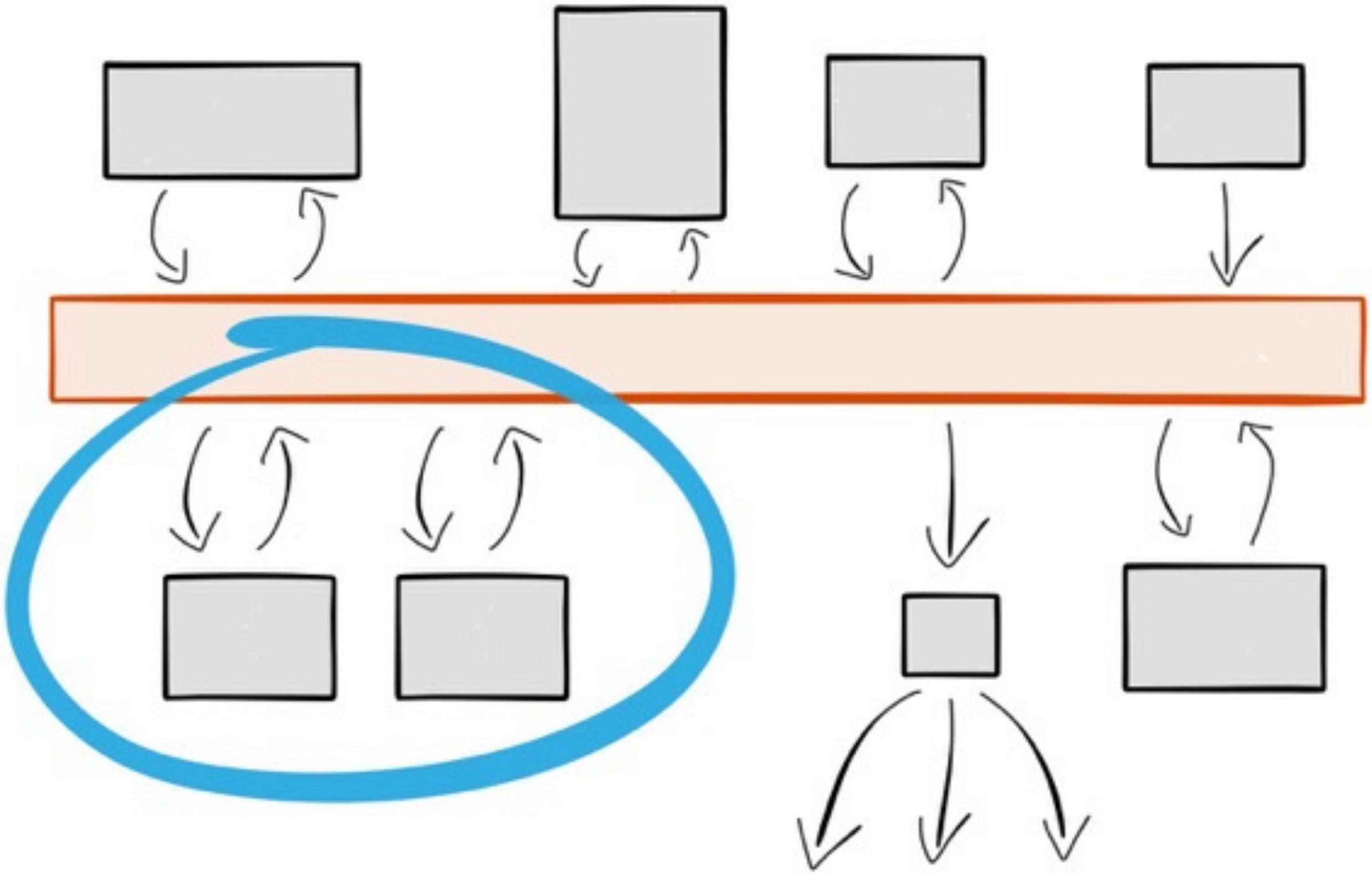
So how is this useful for microservices?

Build ‘Always On’ Services



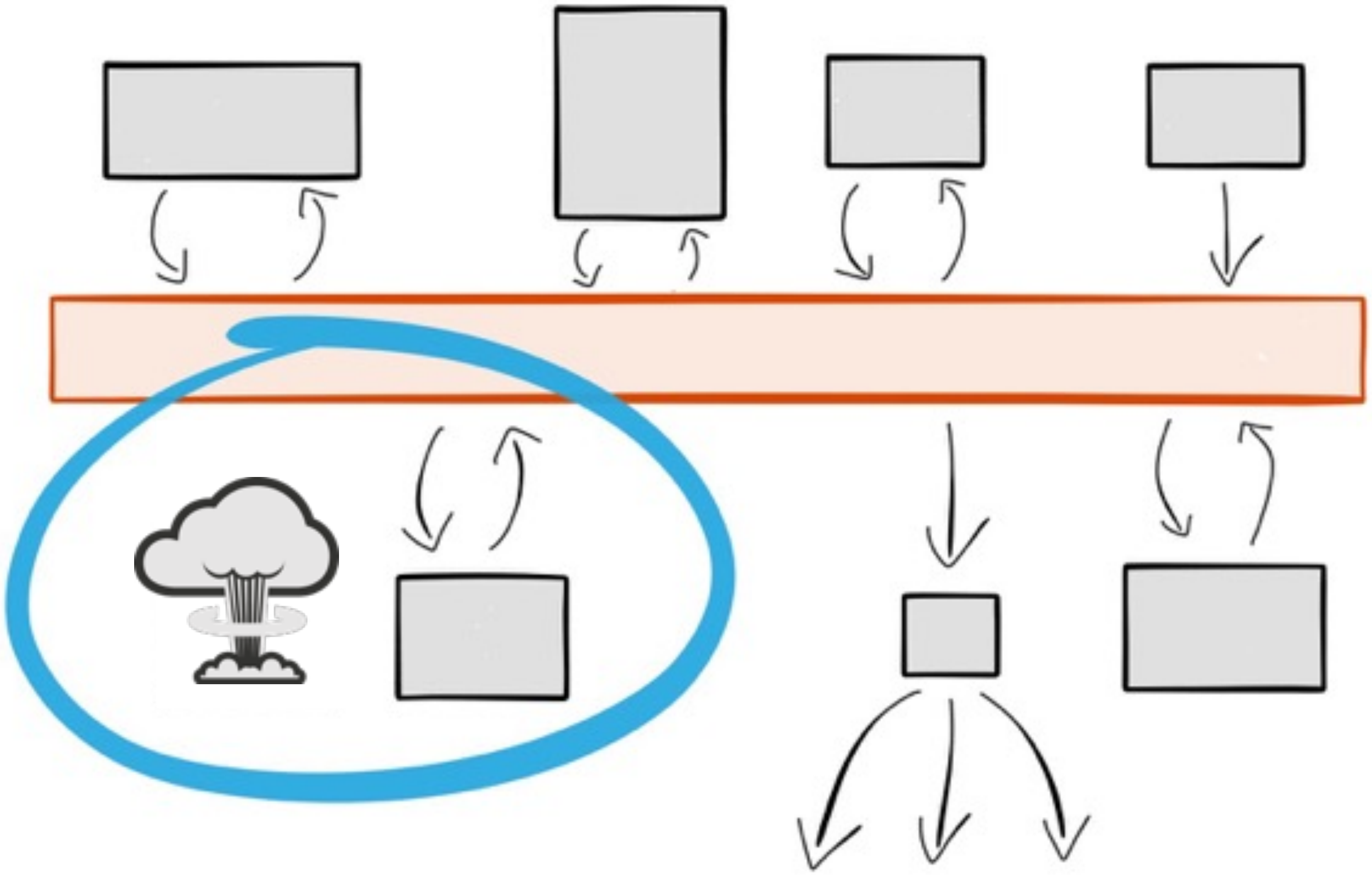
Rely on Fault Tolerant Broker

Load Balance Services



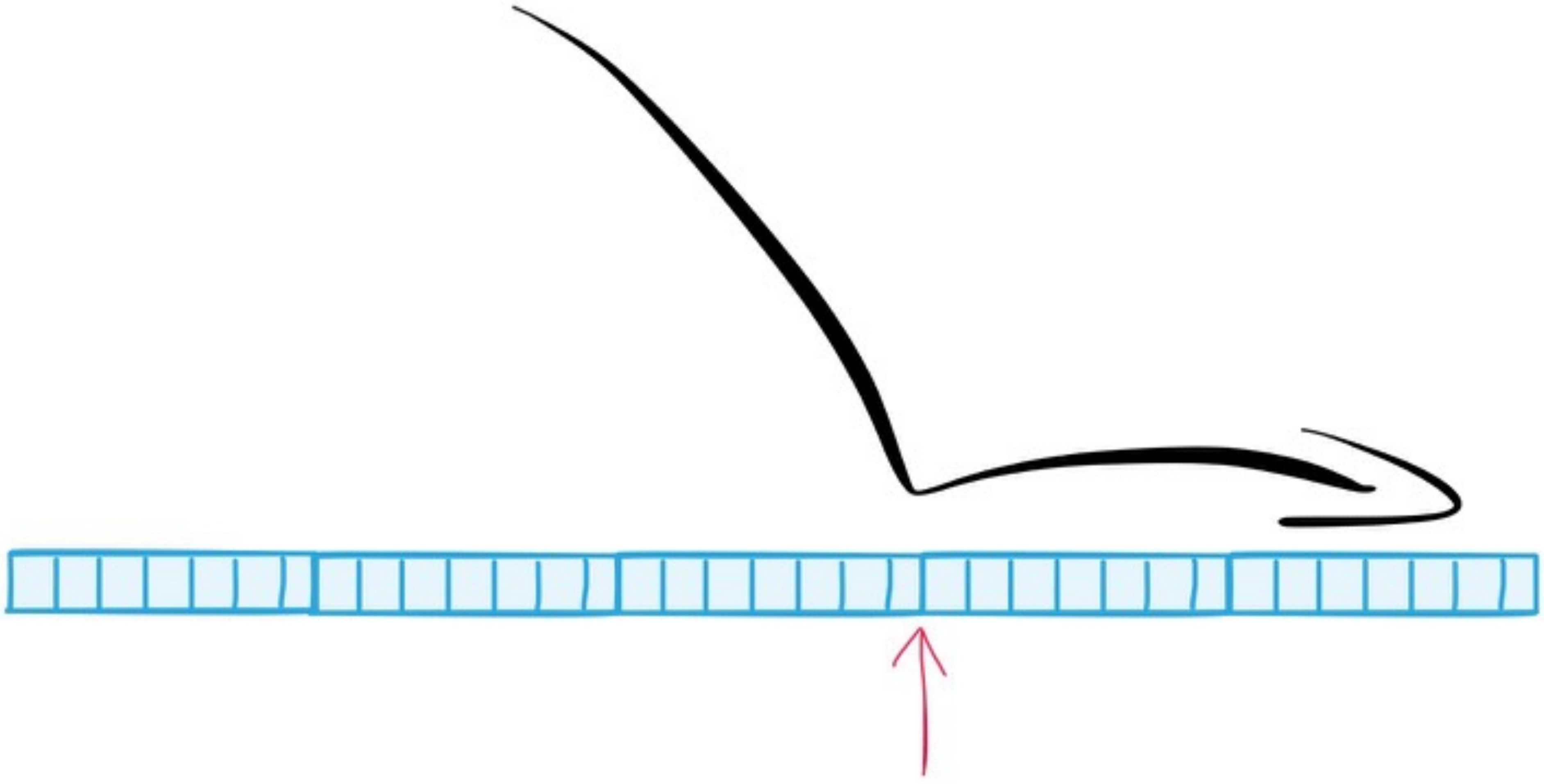
Load Balance Services
(with strong ordering)

Fault Tolerant Services



Services automatically
fail over
(retaining ordering)

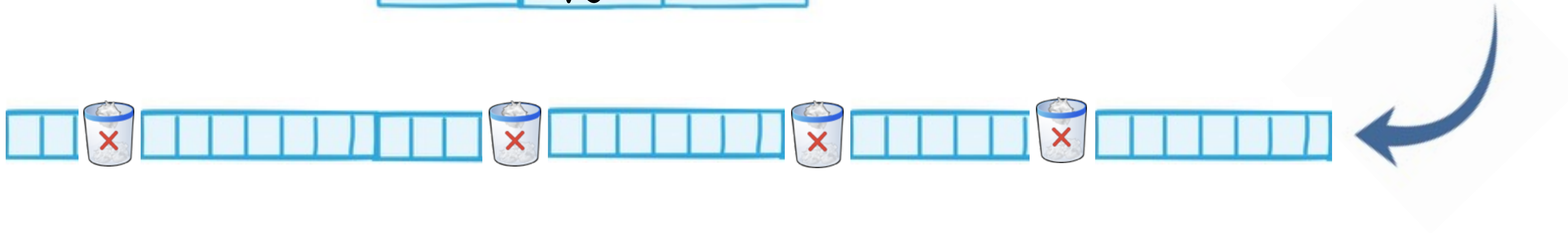
Services can return back to old messages in the log



Rewind & Replay

Compacted Topics are Interesting

K1	V1	
K1	V2	
K1	V3	
K2	V1	
K2	V2	
K1	V4	
K2	V3	



Lets take a little example

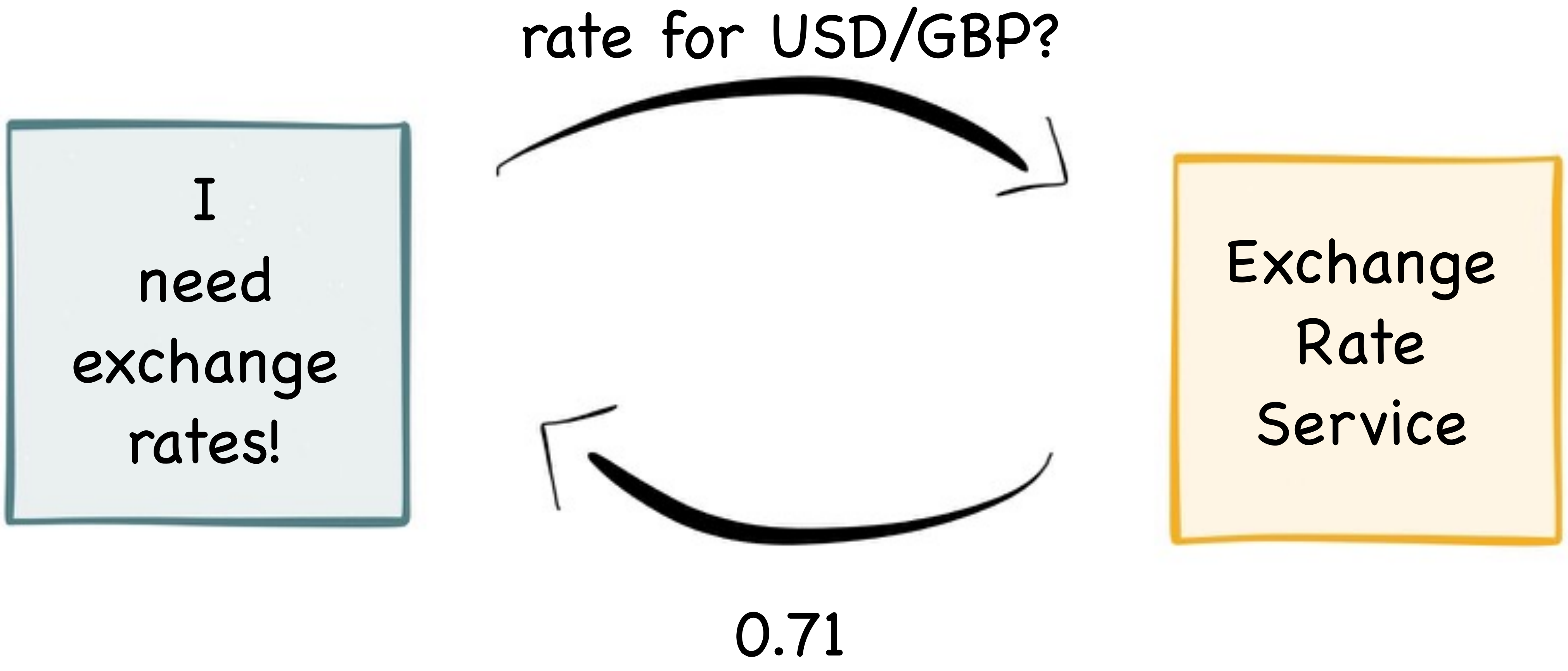
Getting Exchange Rates

I
need
exchange
rates!

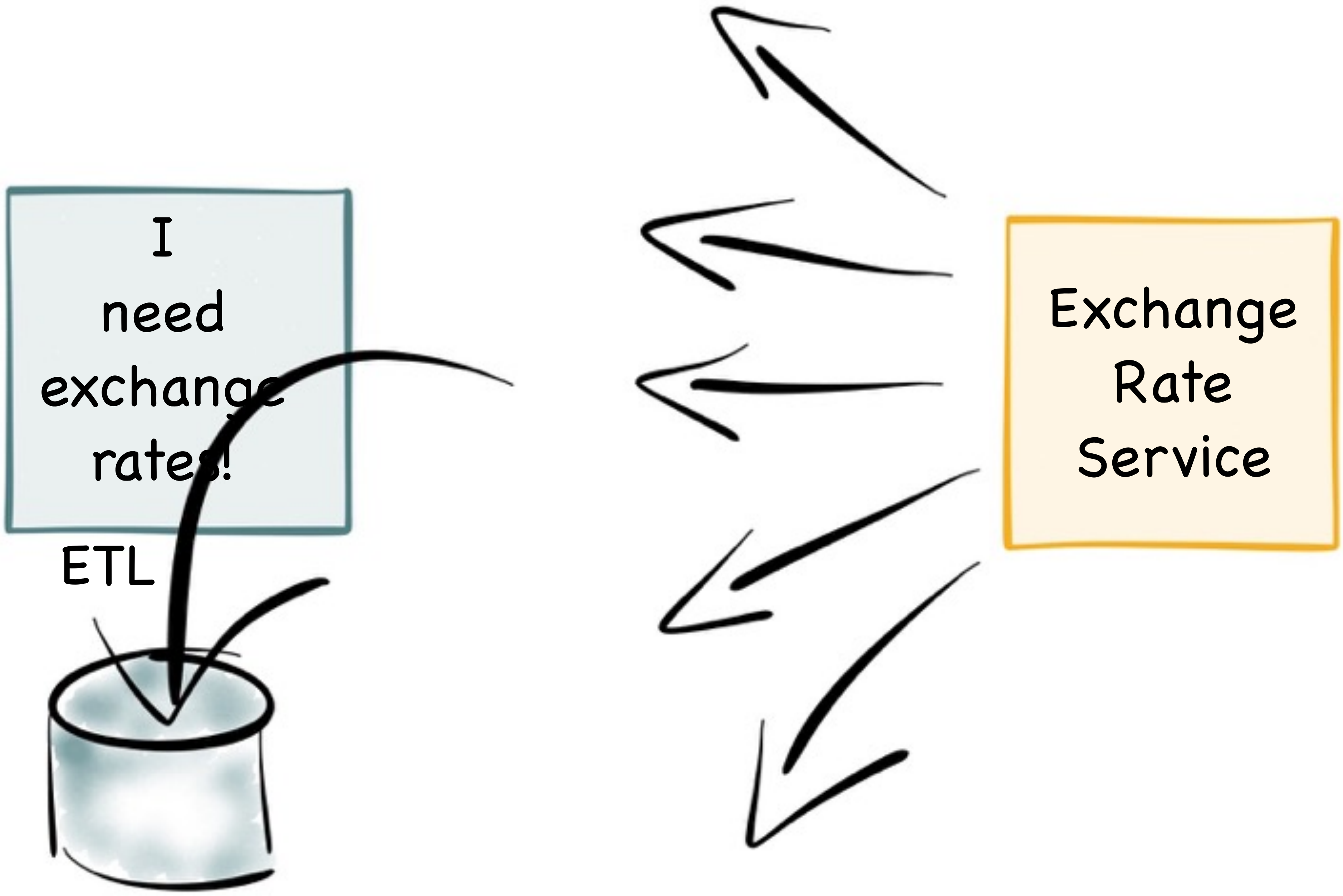
Exchange
Rate
Service

USD/GBP = 0.71
EUR/GBP = 0.77
USD/INR = 67.7
USD/AUD = 1.38
EUR/JPY = 114.41
…

Option1: Request Response

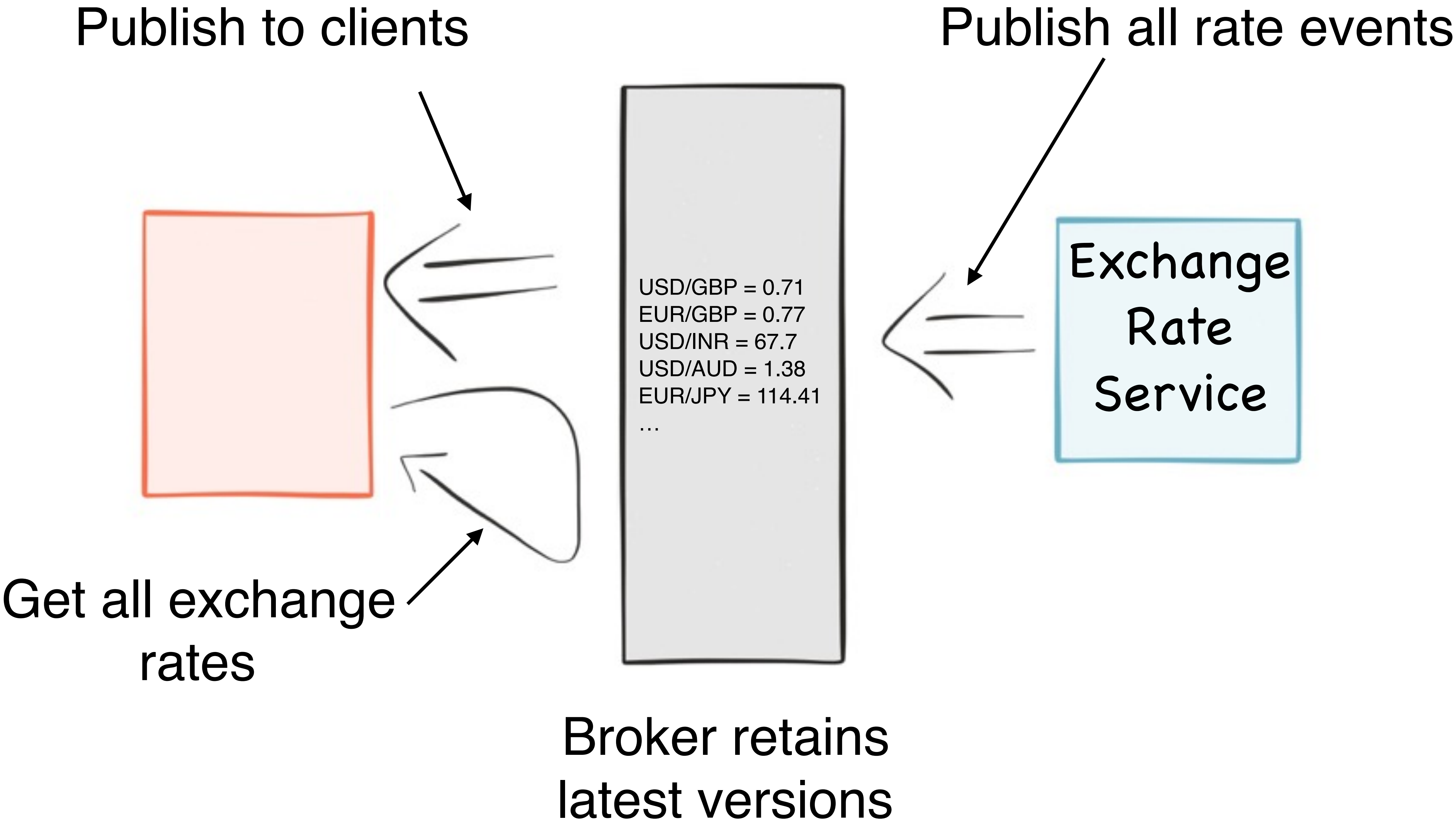


Option 2: Publish Subscribe



Accumulate current state

Option 3: Accumulate in Compacted Stream



Publish to clients

Publish all rate events

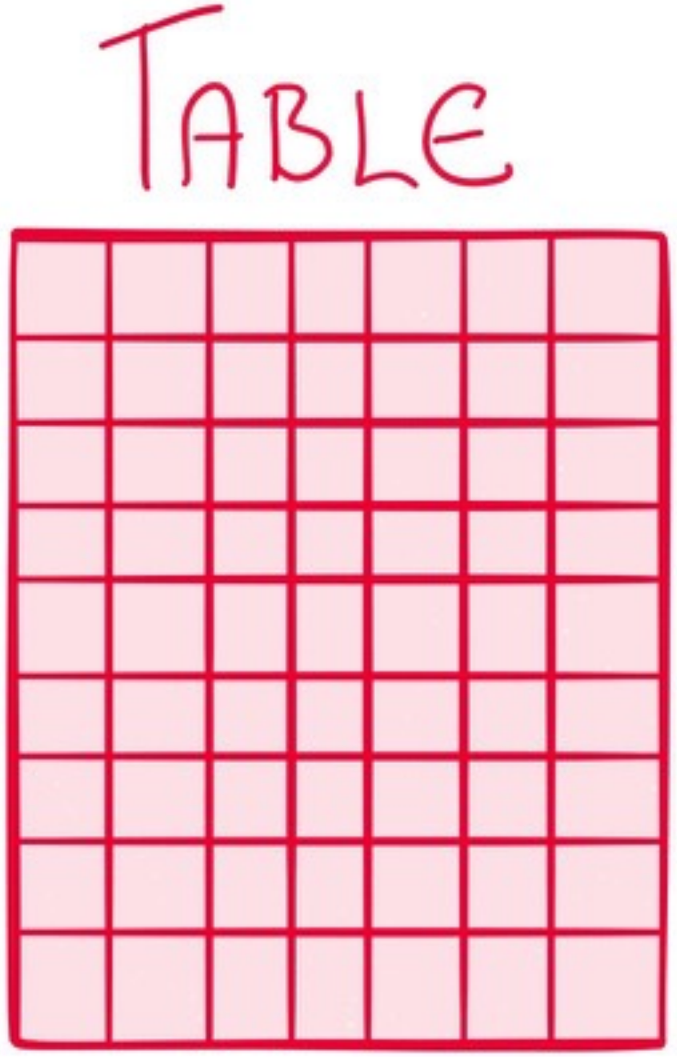
USD/GBP = 0.71
EUR/GBP = 0.77
USD/INR = 67.7
USD/AUD = 1.38
EUR/JPY = 114.41
…

Get all exchange rates

Broker retains latest versions

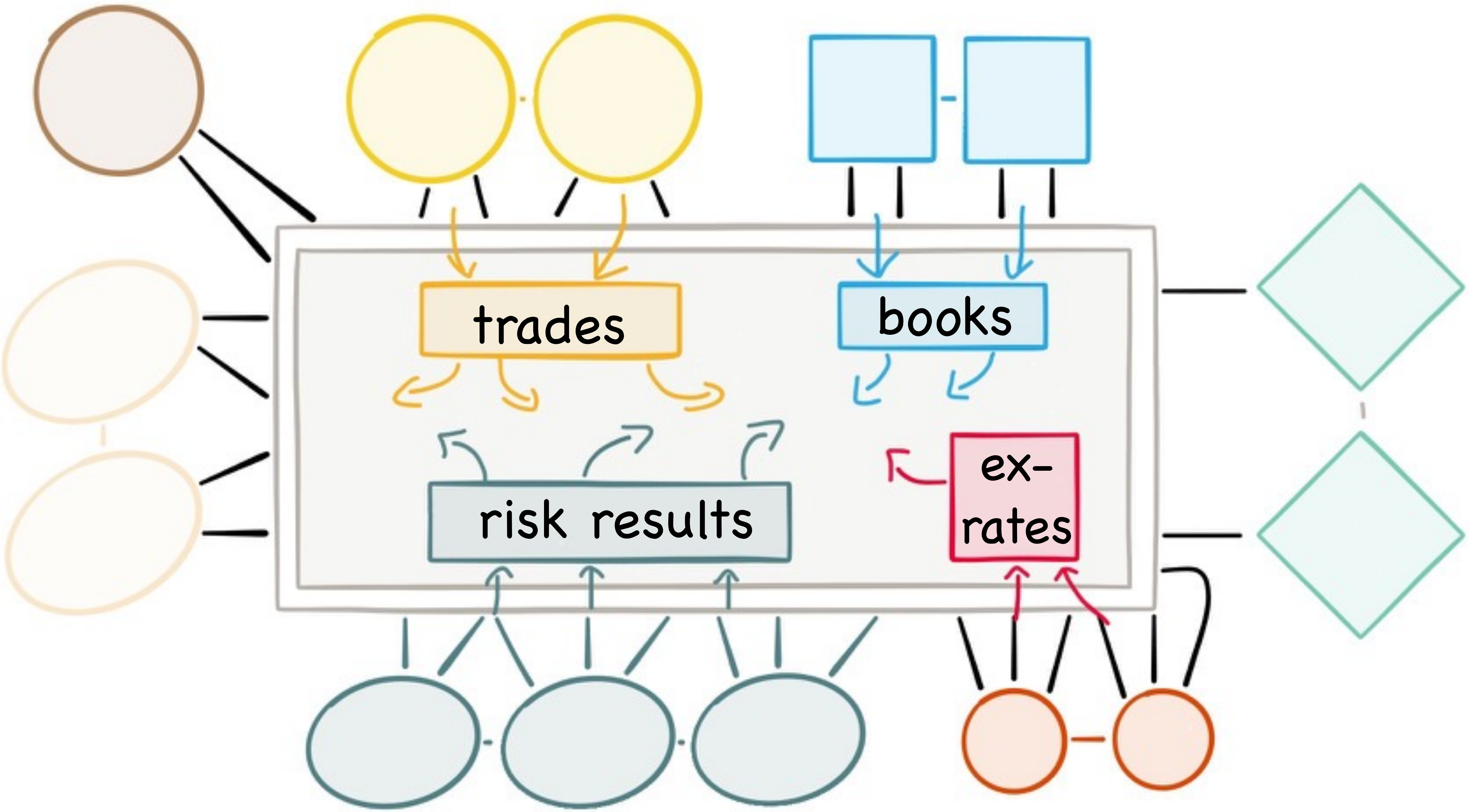
Is it a stream or is it a table?

transitory



stateful

Datasets can live in the broker!

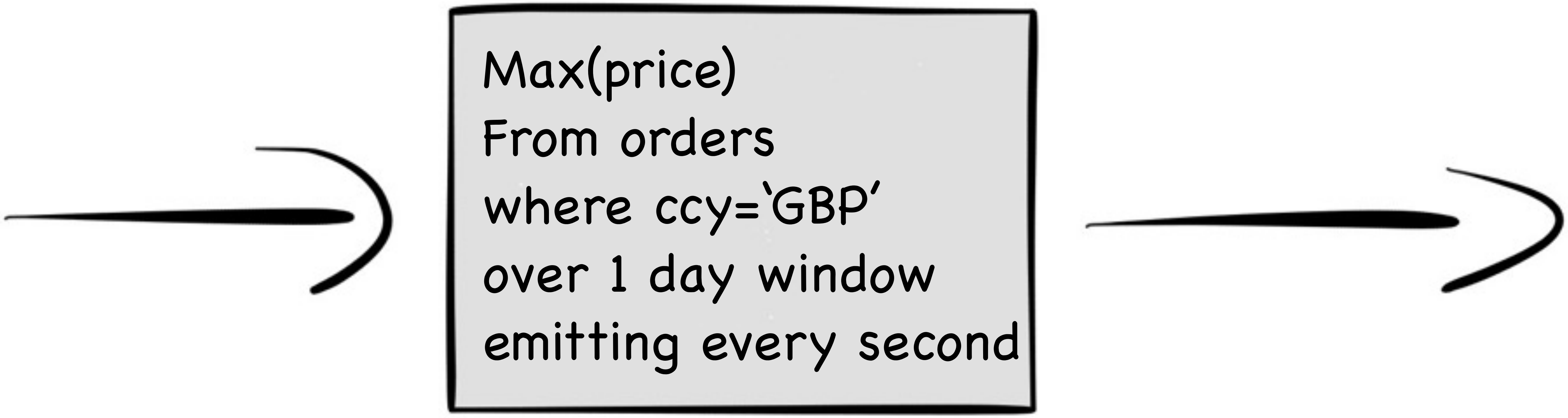


Service Backbone

Scalable, Fault Tolerant, Concurrent, Strongly Ordered, Stateful

… lets add in stream processing

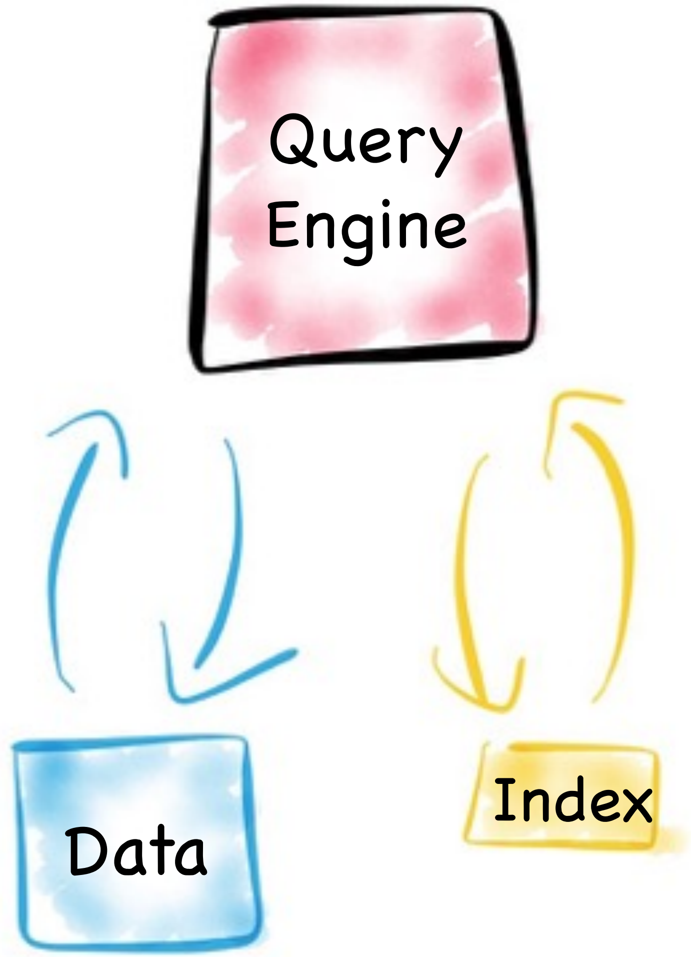
What is stream processing?



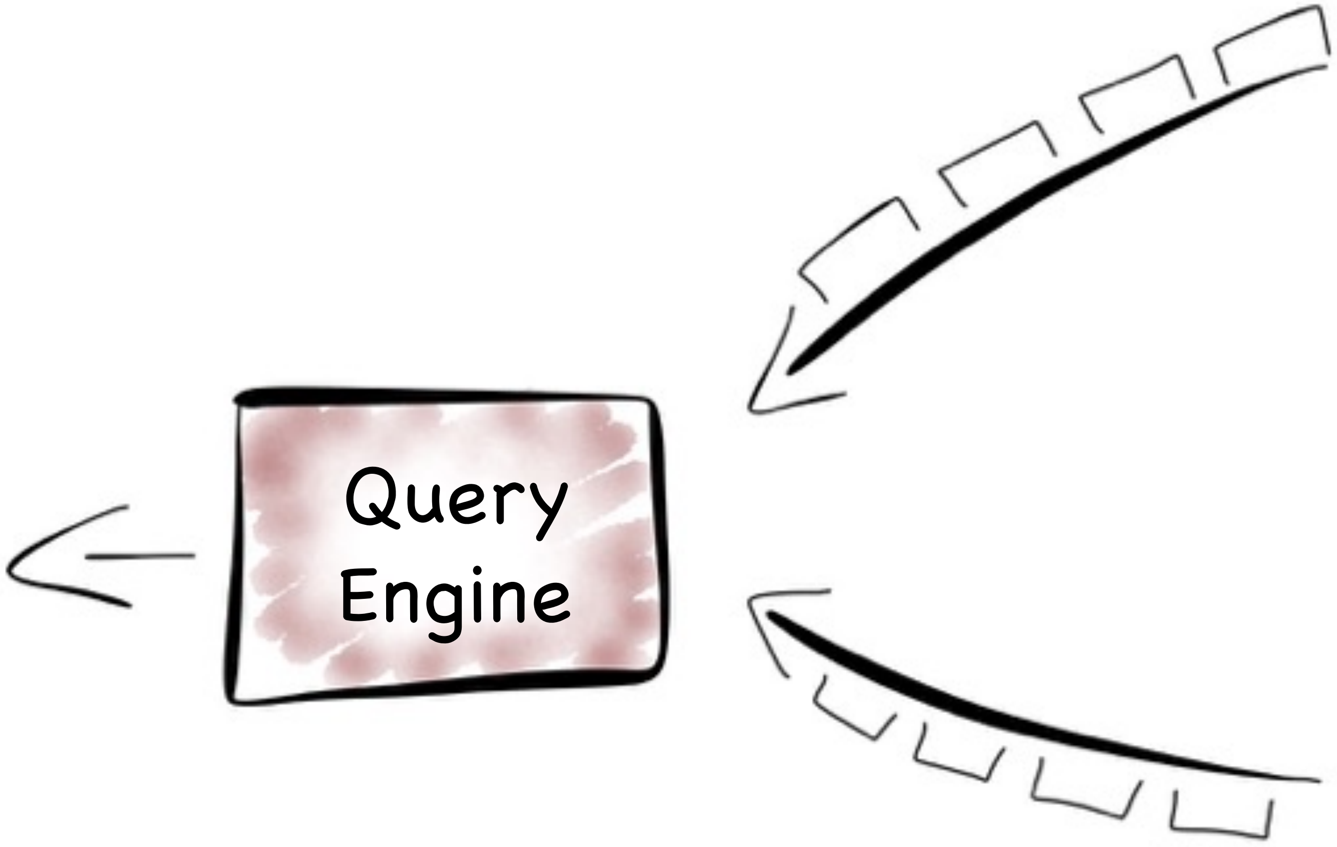
Max(price)
From orders
where ccy='GBP'
over 1 day window
emitting every second

Continuous Queries.

What is stream processing engine?



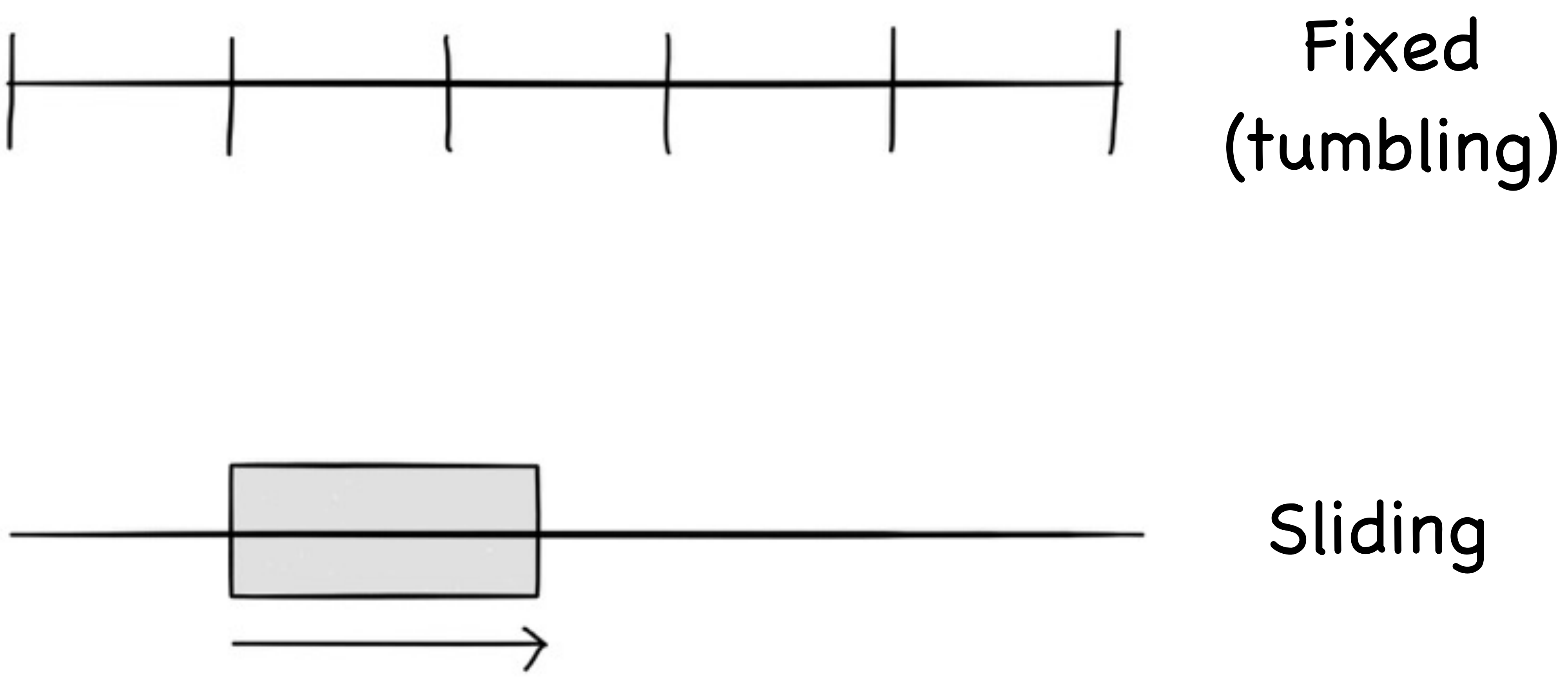
vs



Database
Finite, well defined source

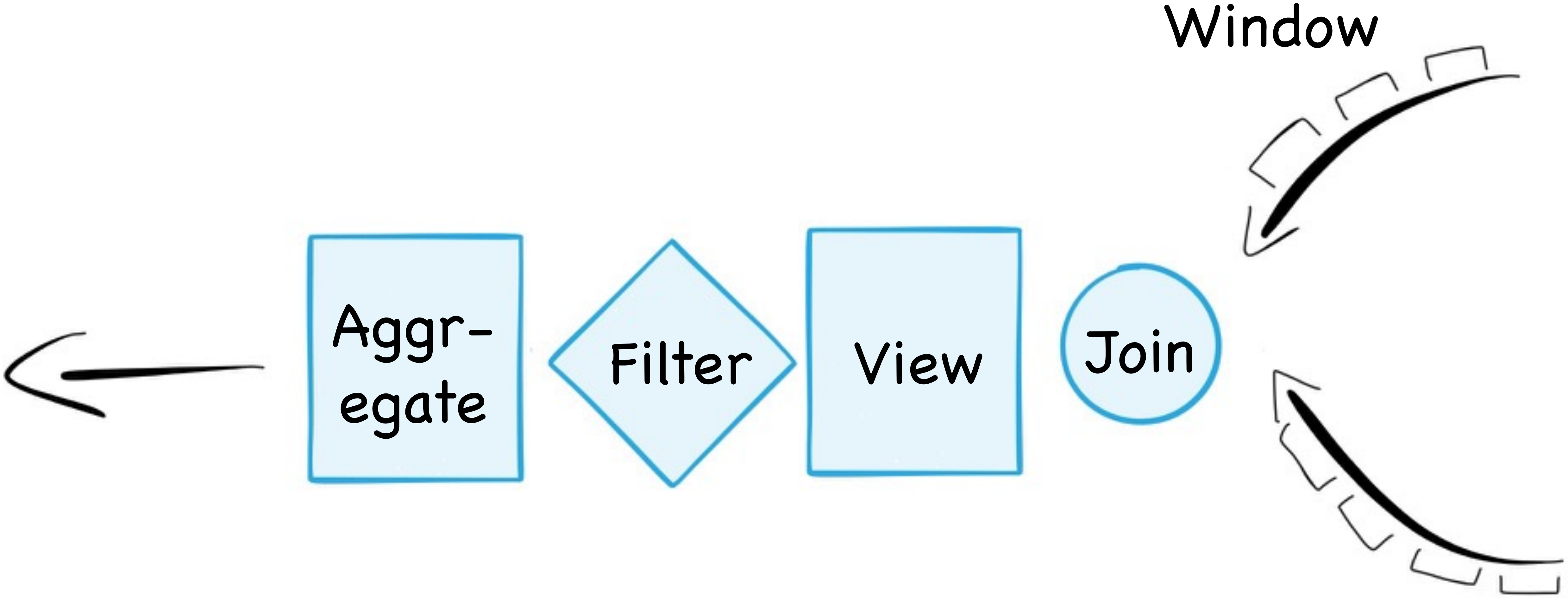
Stream Processor
Infinite, poorly defined source

Windowing

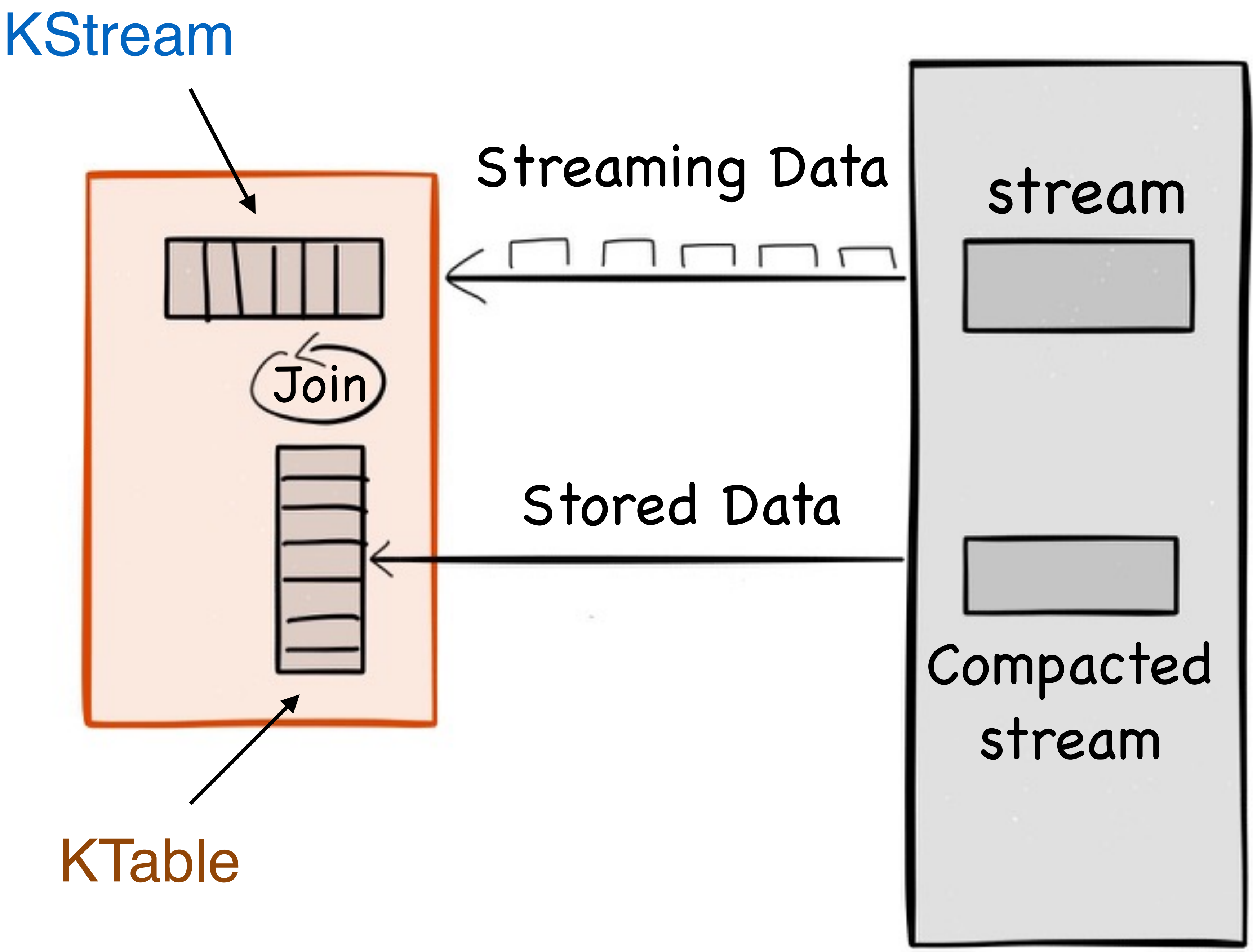


For unordered or unpredictable streams

Features: similar to database query engine

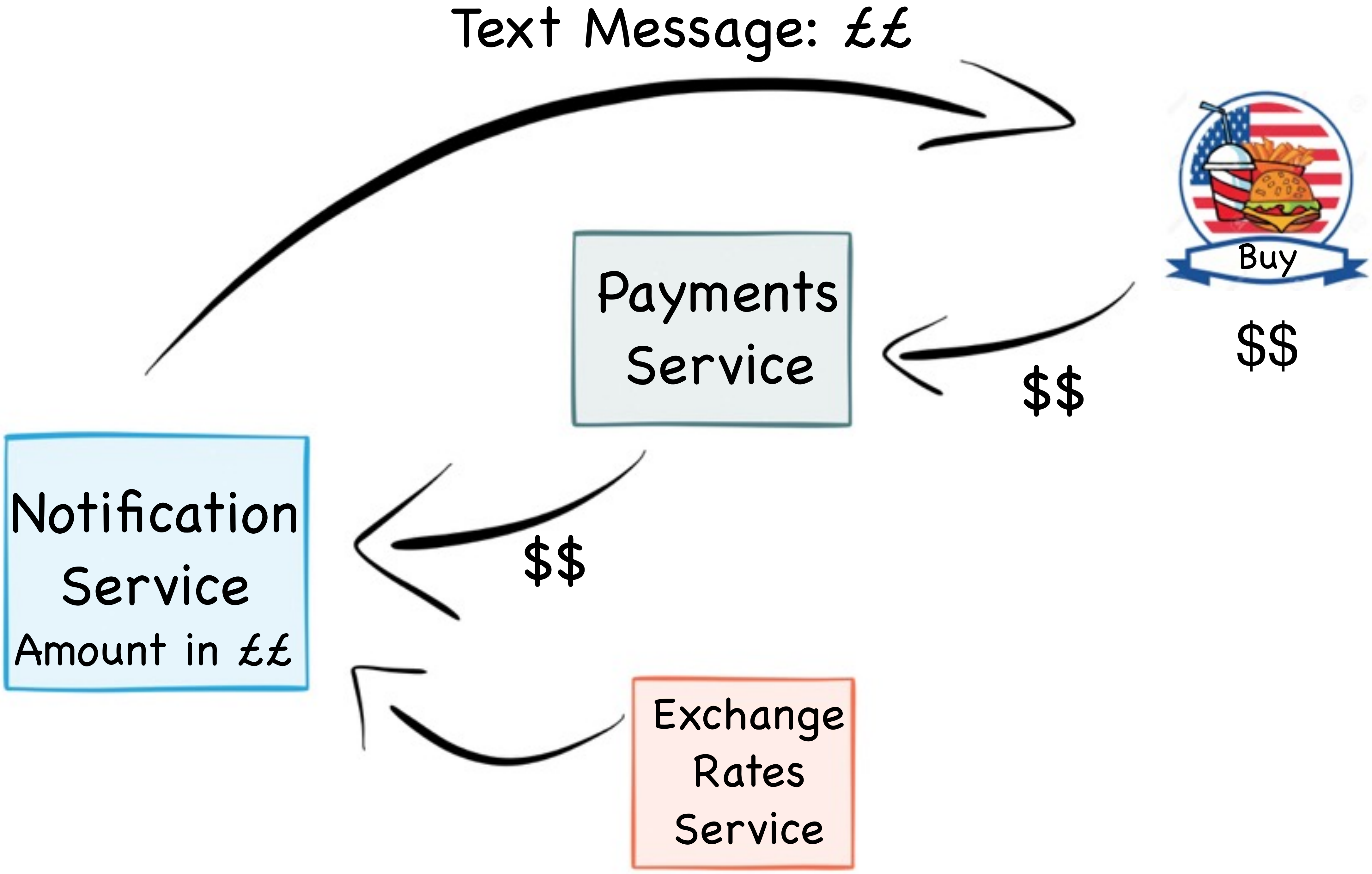


KStreams & KTables



A little example…

Buying Lunch Abroad



Text Message: ££

Buy

$$

Payments Service

$$

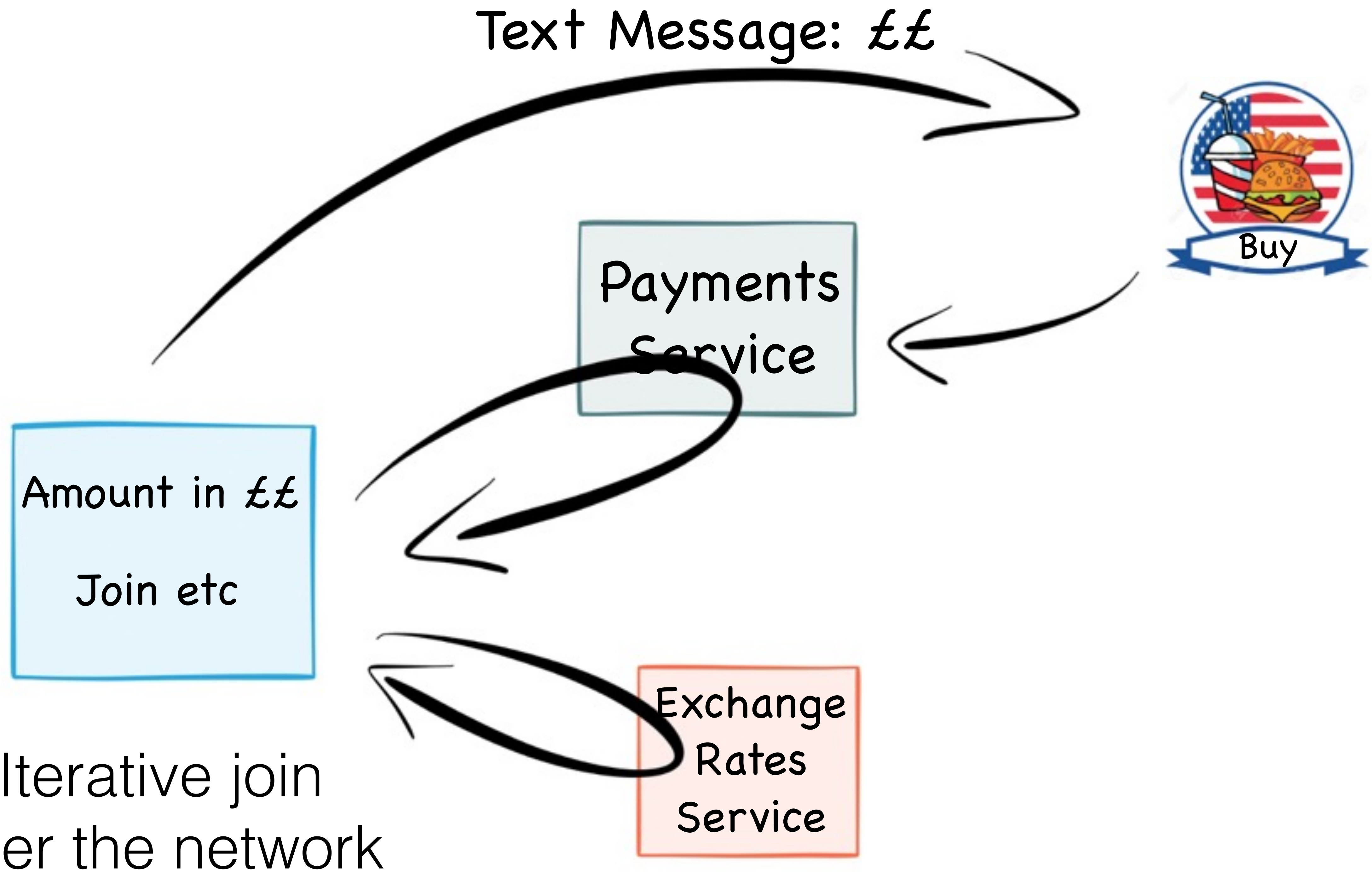
Notification Service

Amount in ££

$$

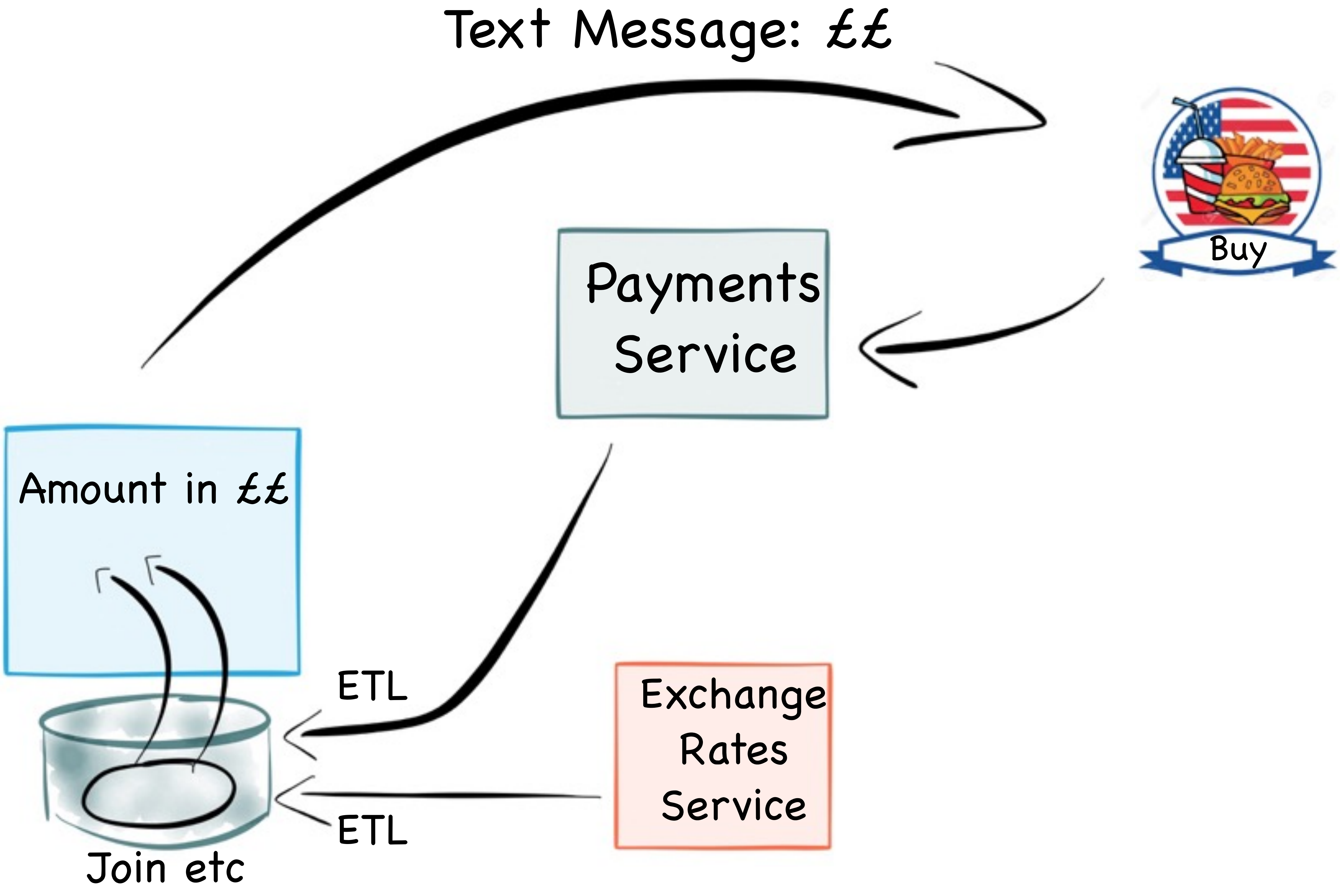
Exchange Rates Service

Request-Response Option

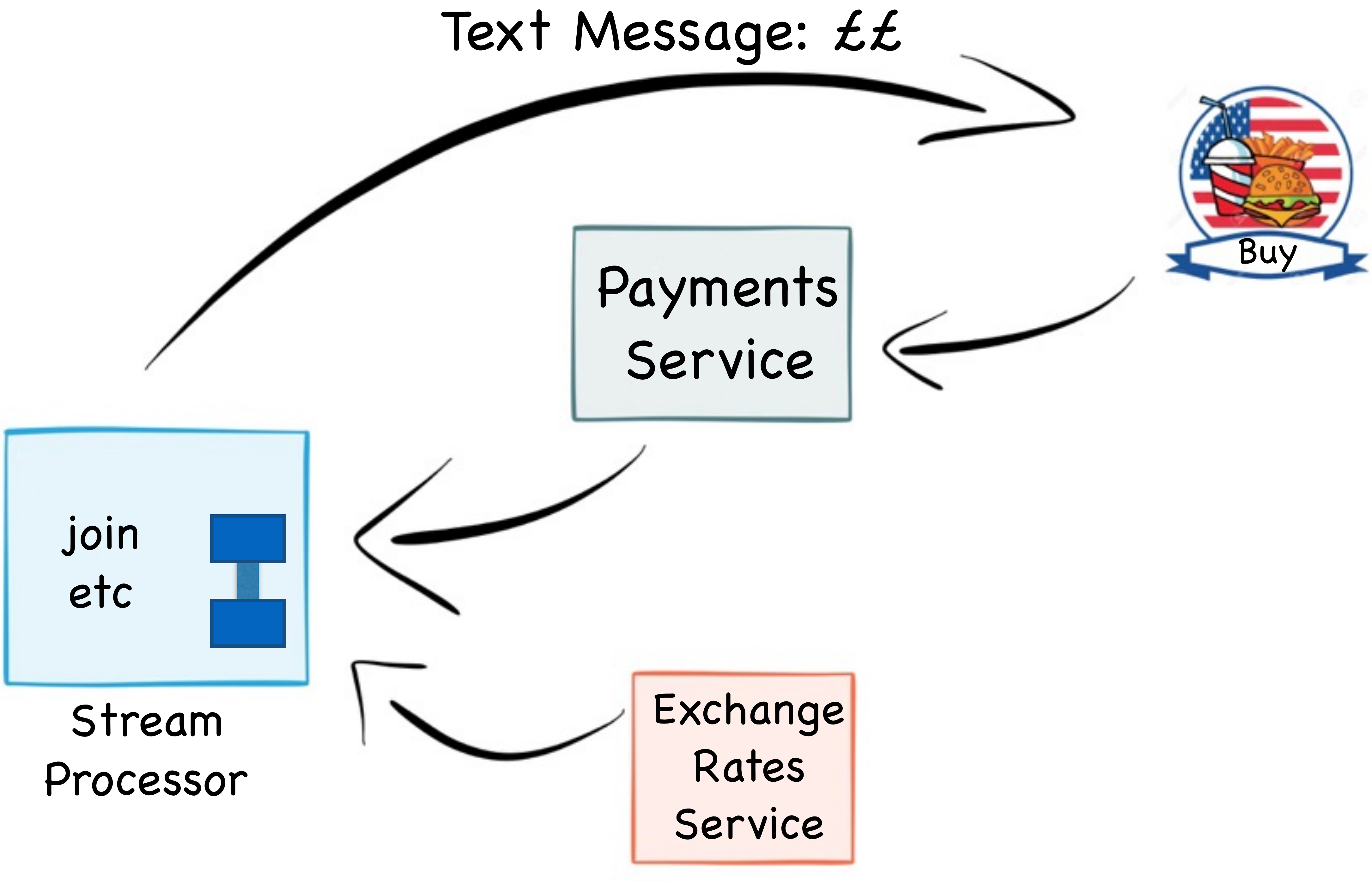


Iterative join
over the network

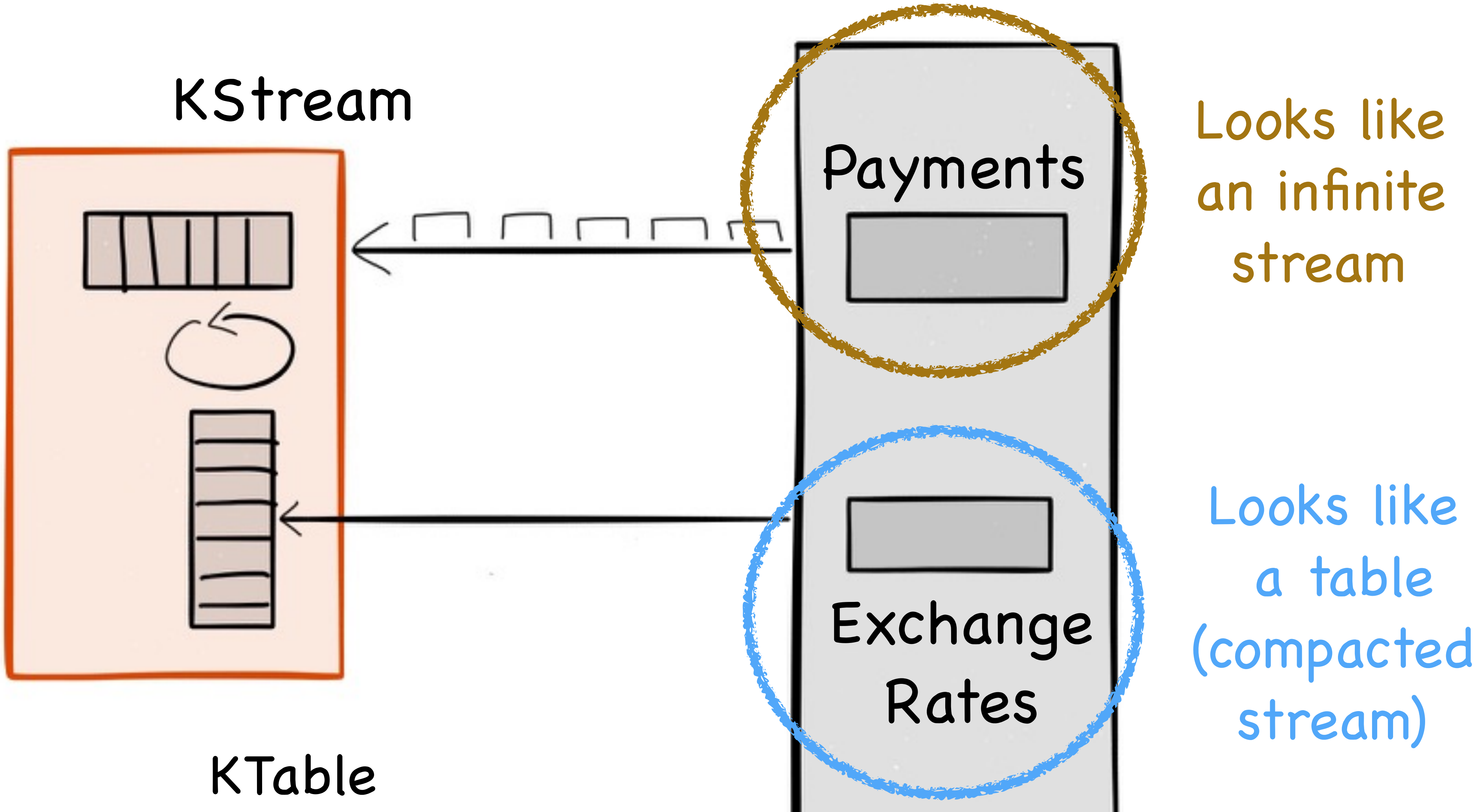
ETL Option


Text Message: ££
Buy
Payments Service
Amount in ££
ETL
ETL
Exchange Rates Service
Join etc

Stream Processor Option

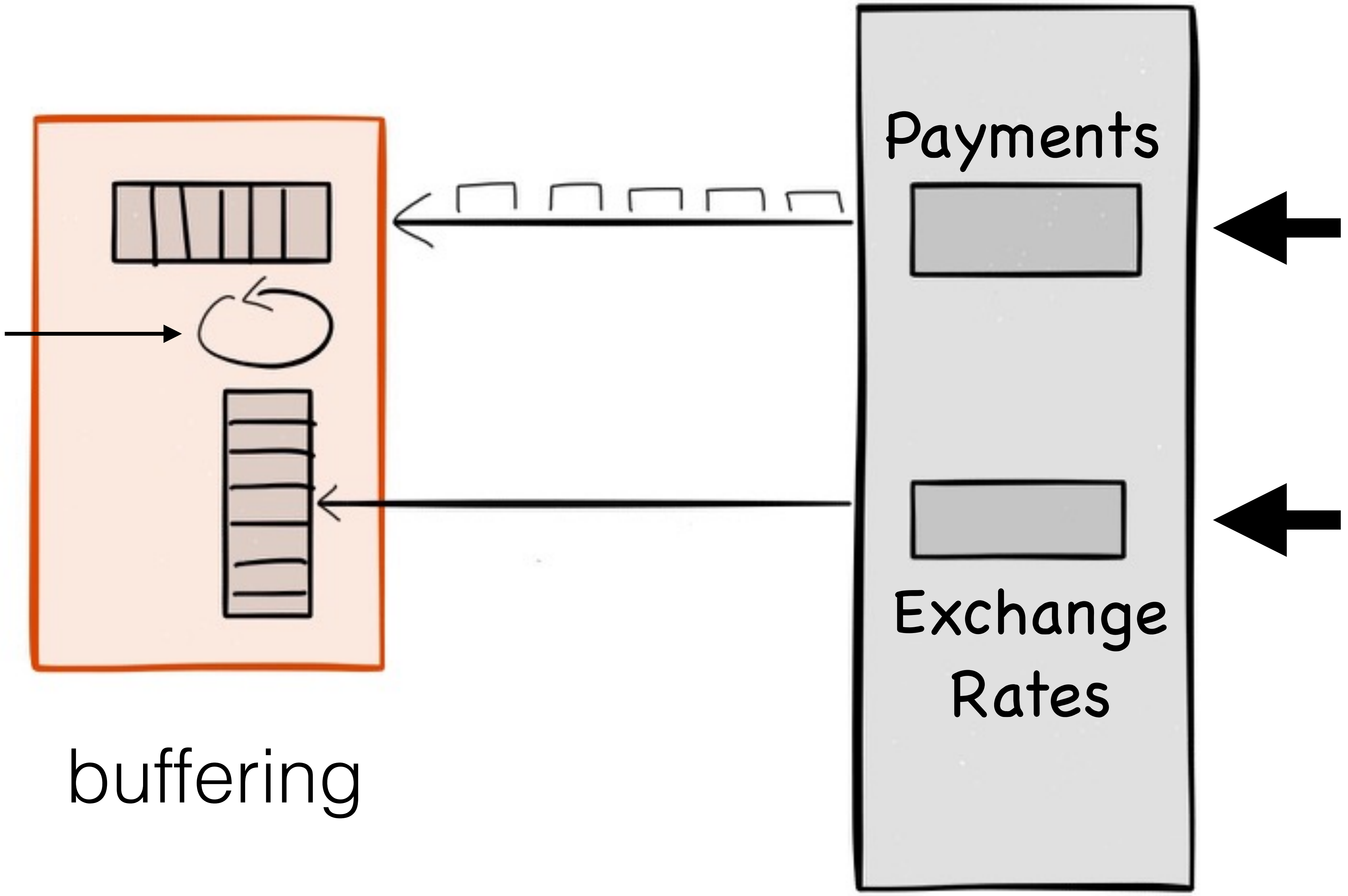


Buying Lunch Abroad



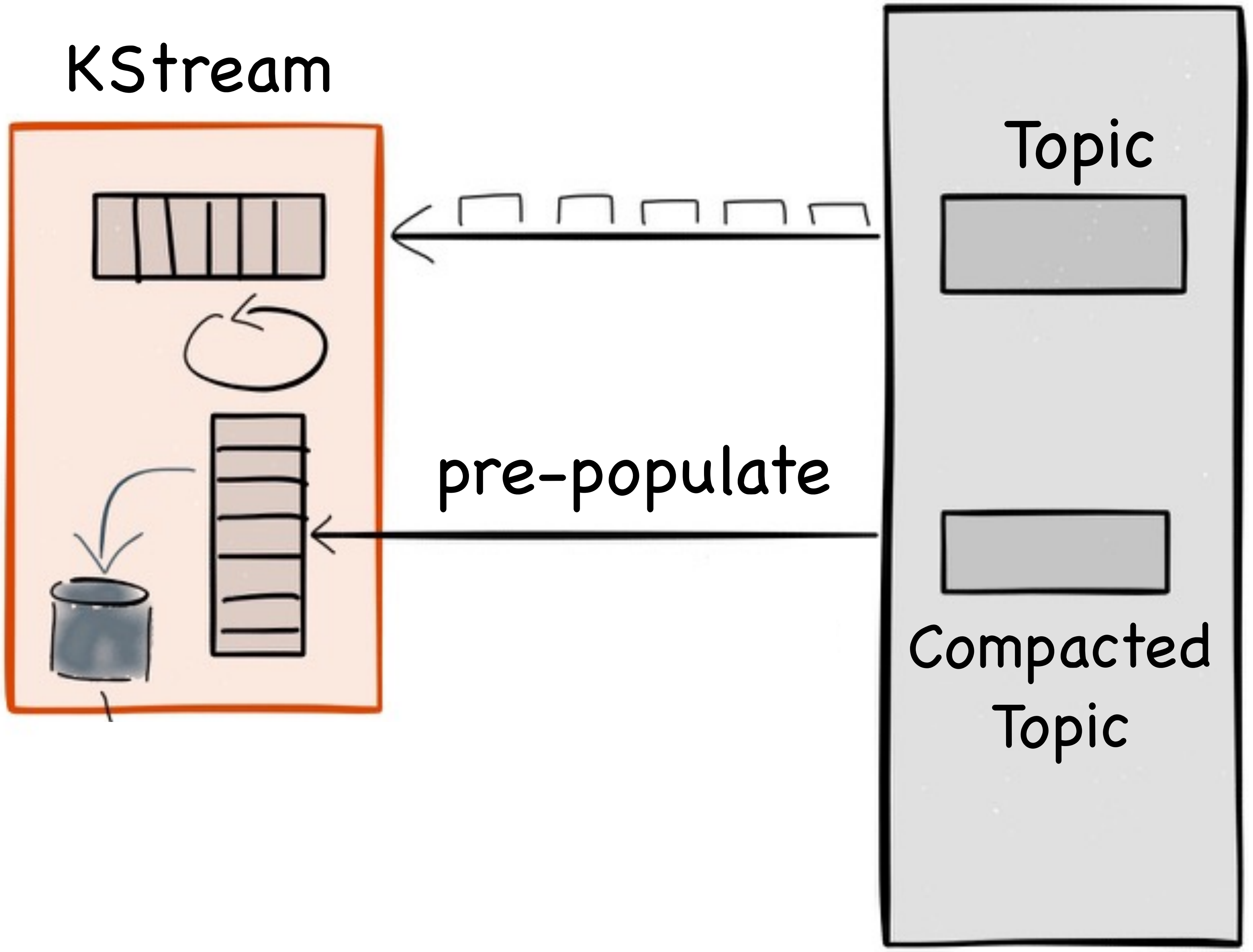
Buying Lunch Abroad

- Filter(ccy<>’GBP’)
- Join on ccy
- Calculate GBP
- Send text message

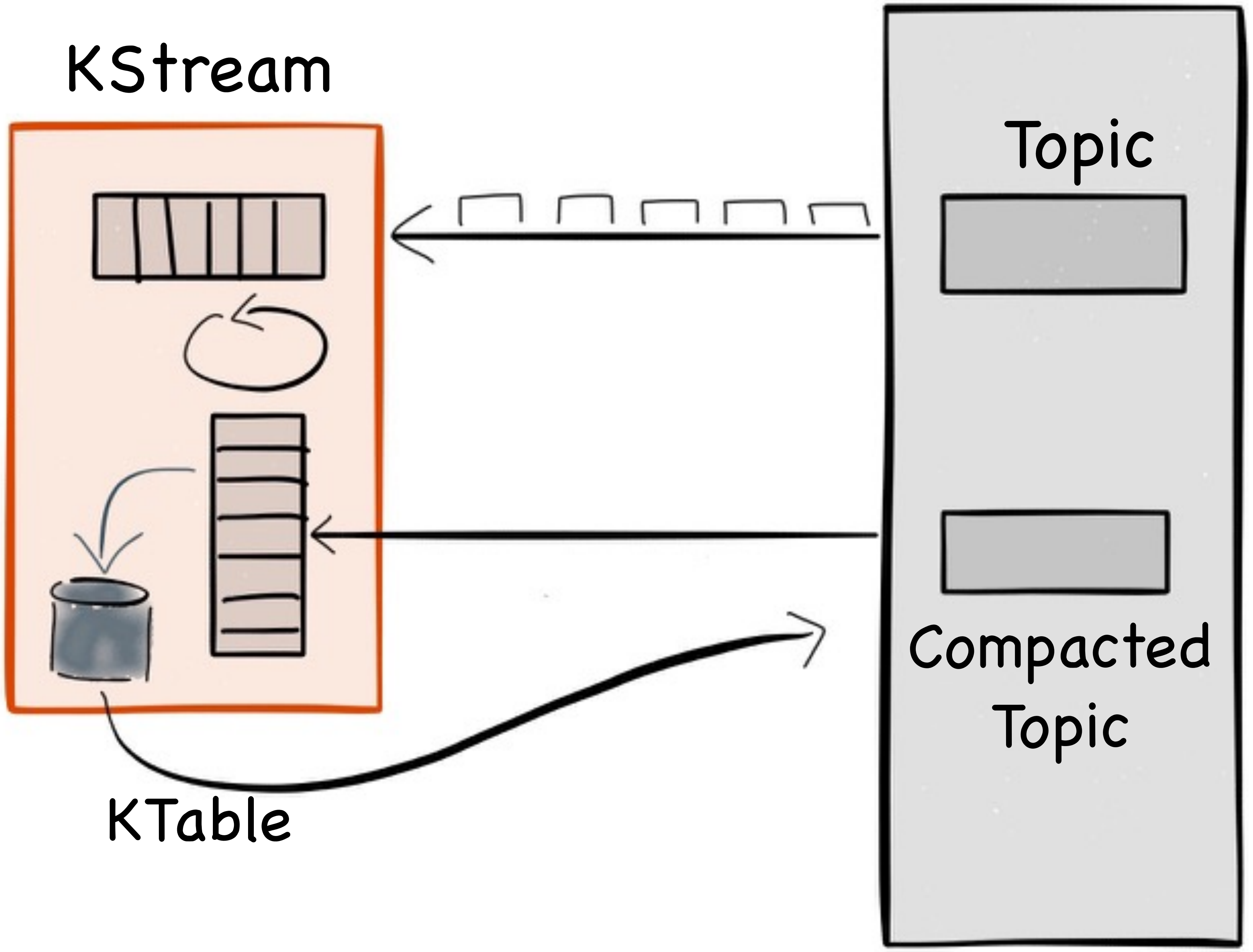


buffering

Local DB (fast joins)

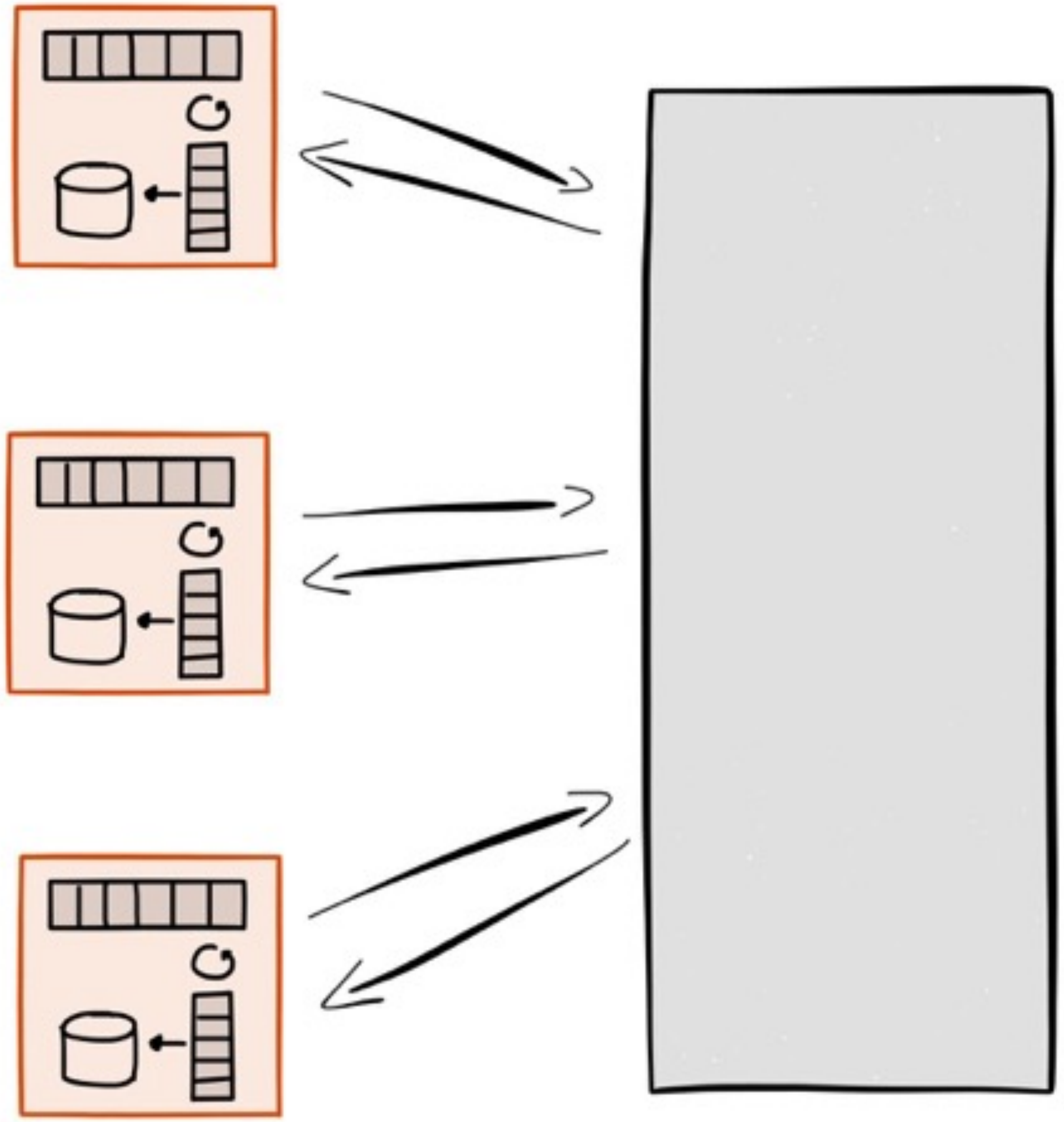


KTables can also be written to - they're backed by the broker



Manage intermediary state

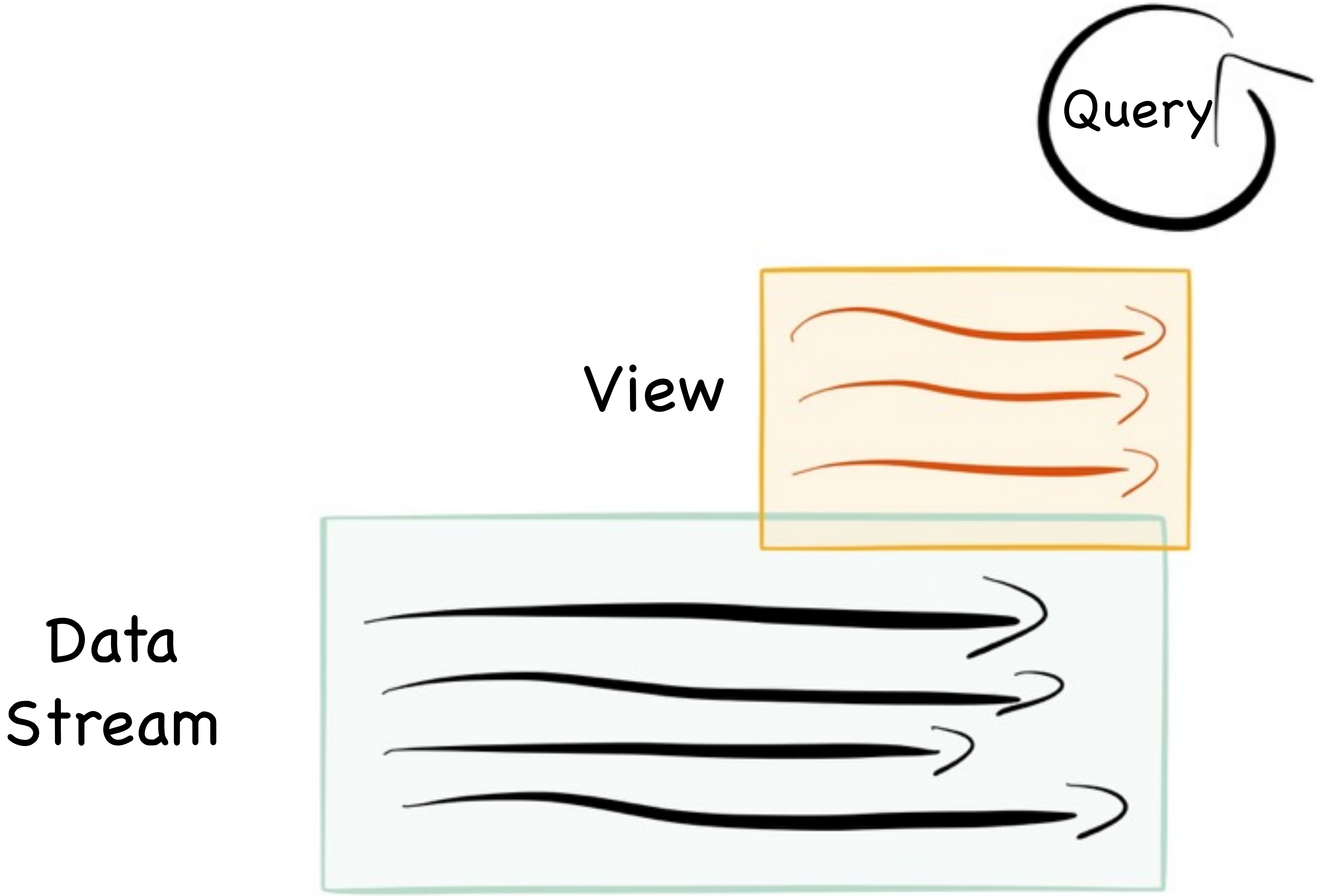
Scales Out (MPP)



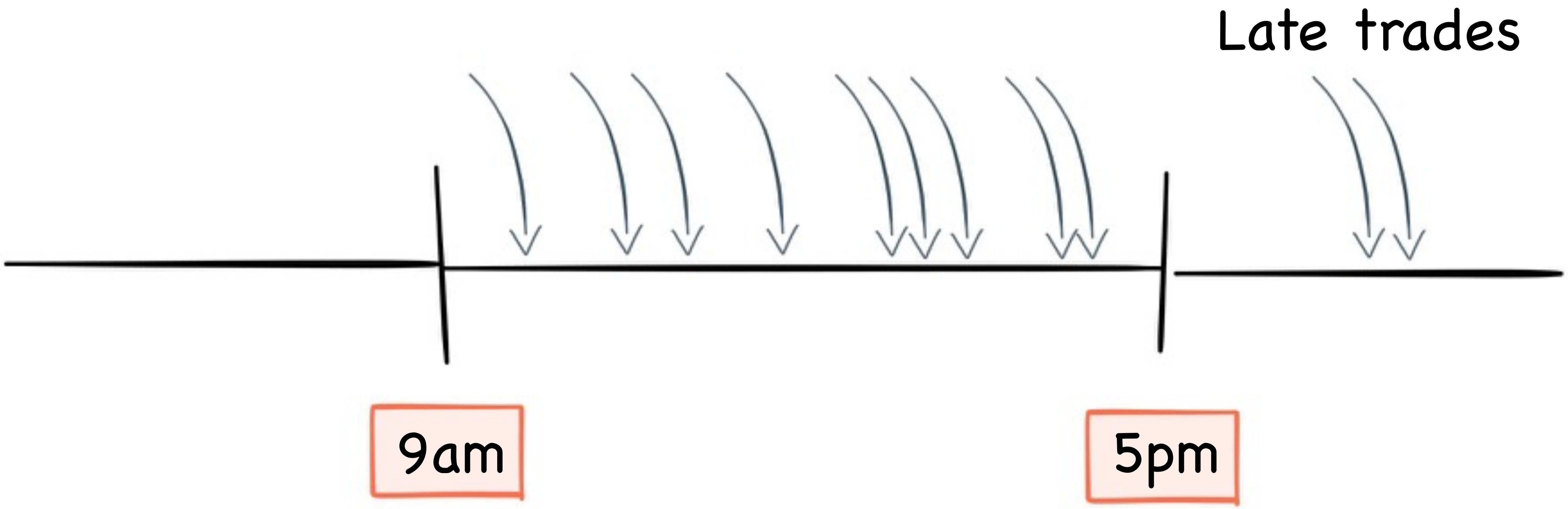
These tools are pretty handy

for managing decentralised services

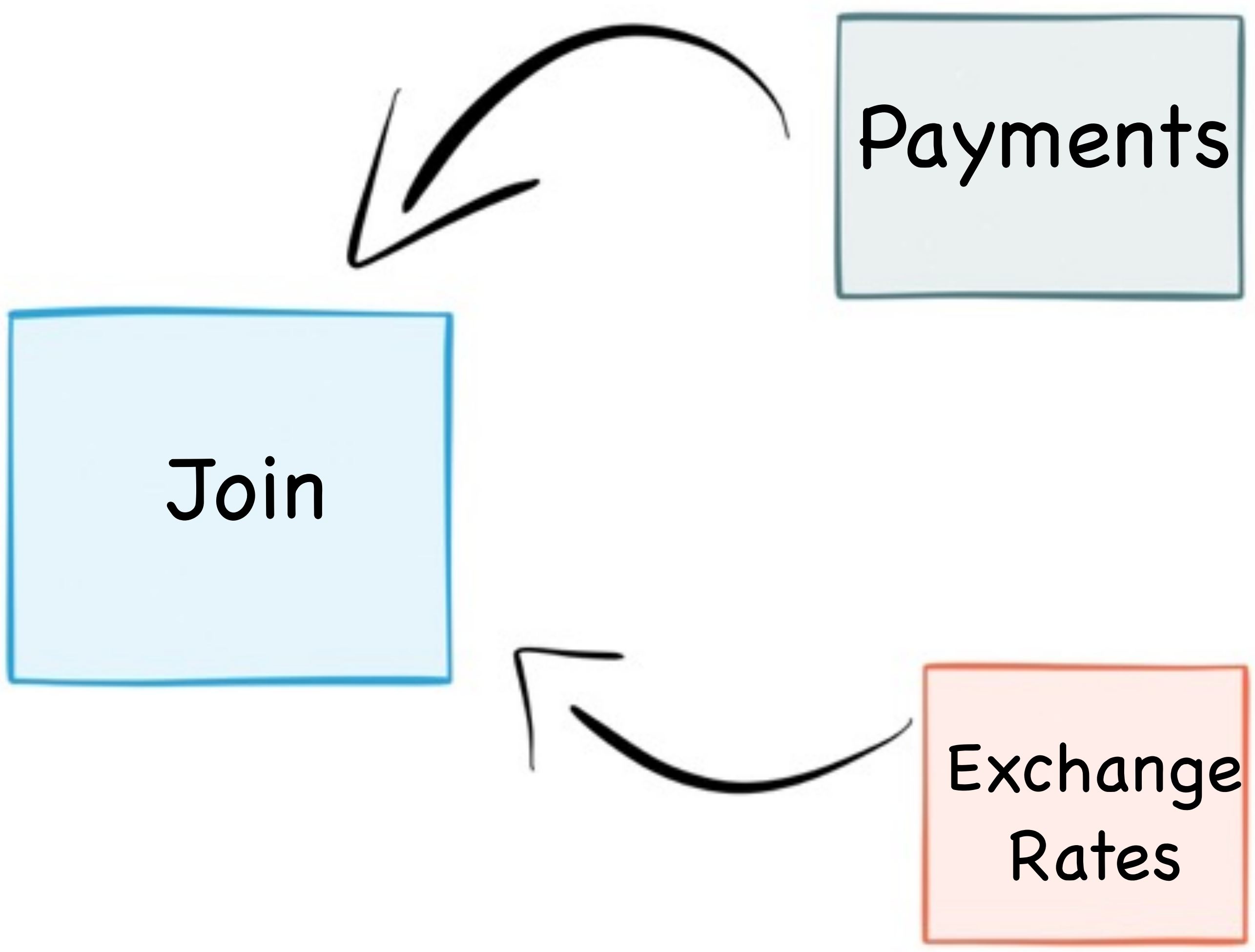
Talk our own data model



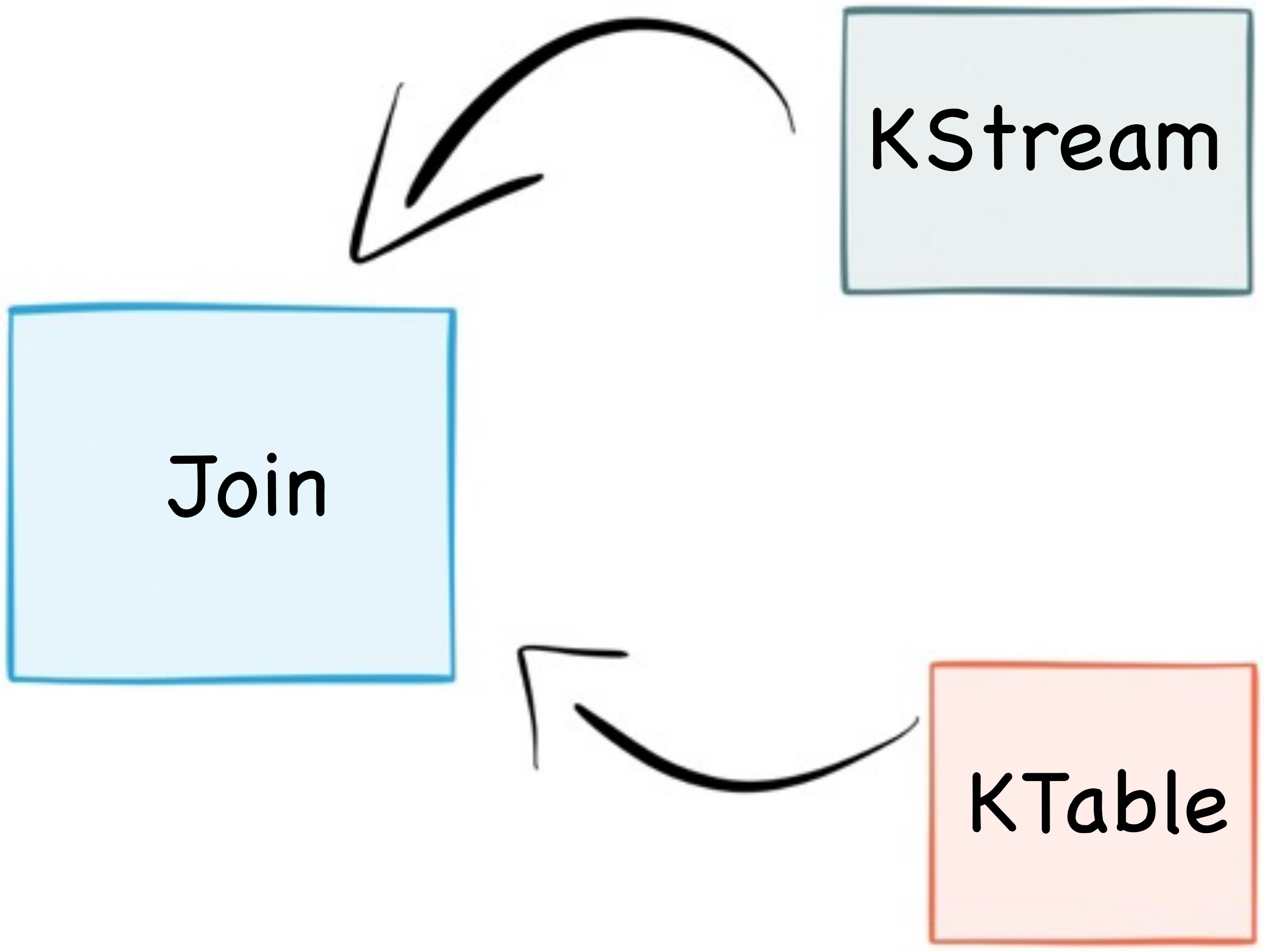
Handle Unpredictability



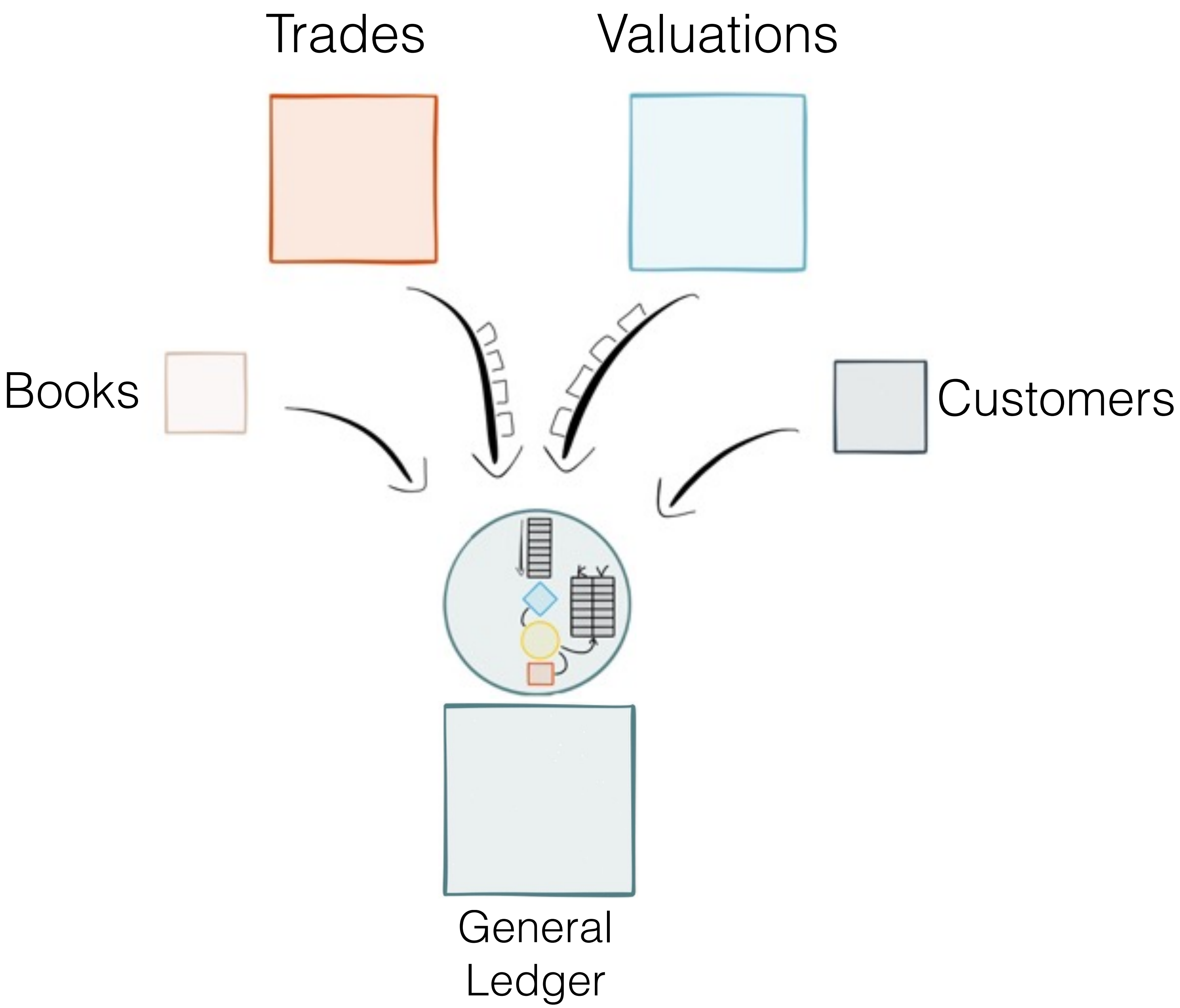
Joining Services



Duality between Stream and Table



More Complex Use Cases



Trades

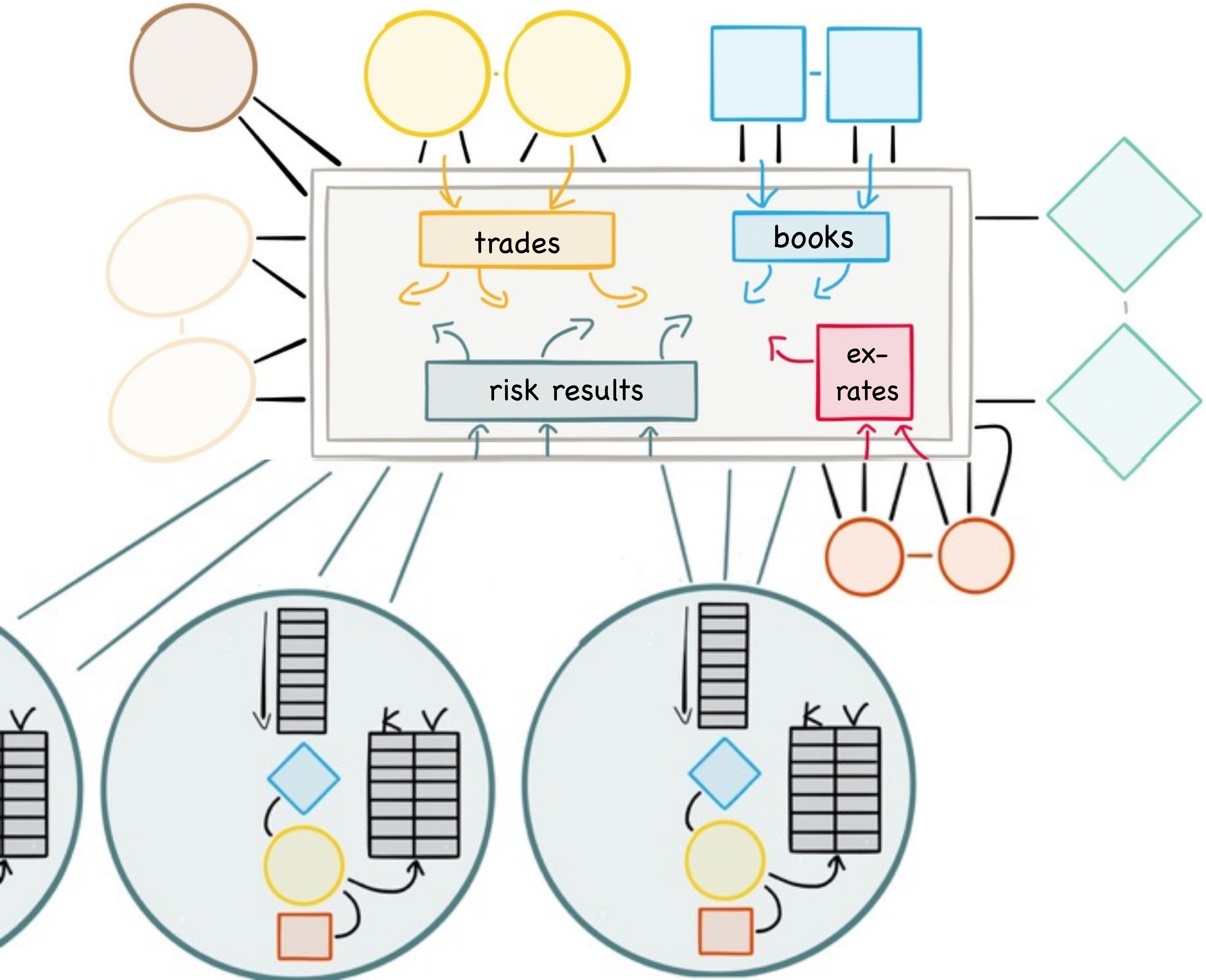
Valuations

Books

Customers

General Ledger

Practical mechanism for managing data intensive, loosely coupled services



- Stateful streams live inside the Log
- Data extracted quickly!
- Fast, local joins, over large datasets
- HA pre-caching
- Manage intermediary state
- Just a simple library (over Kafka)

There is much more to stream processing

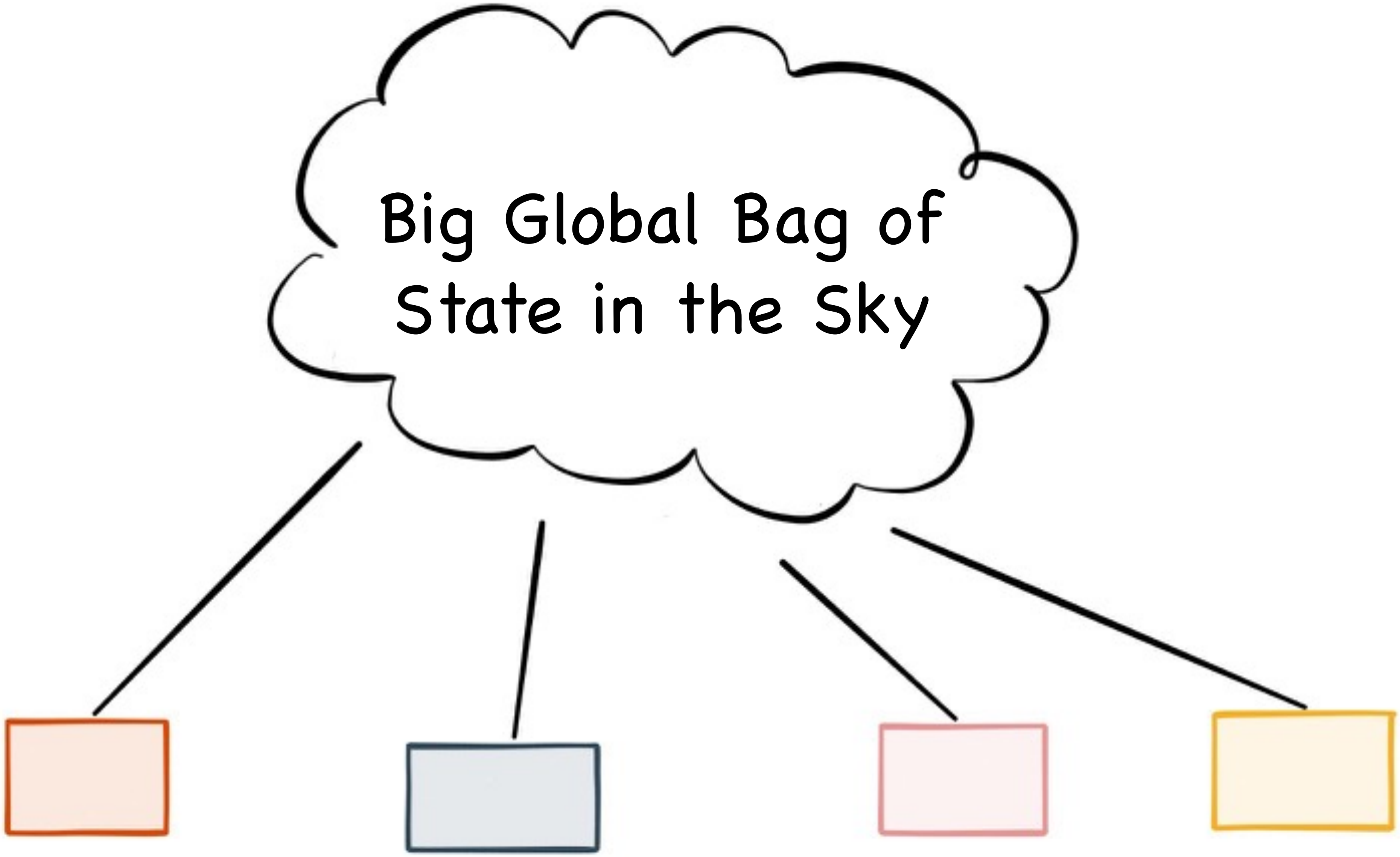
it is grounded in the world of big-data analytics

Simple Approaches

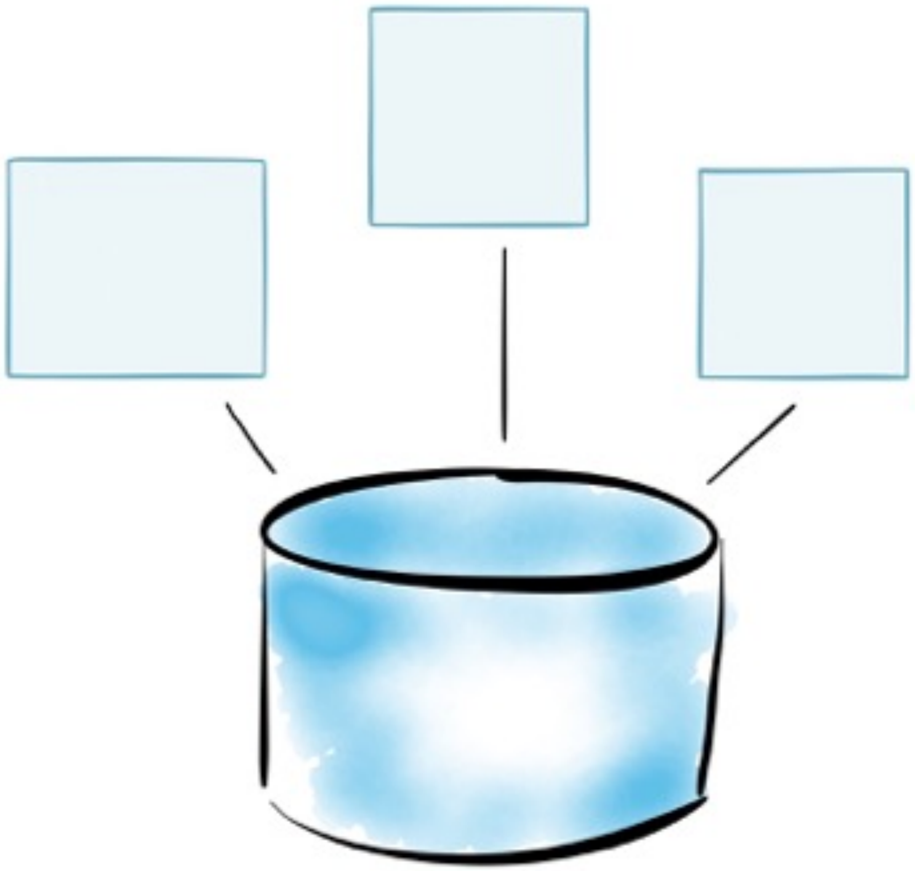
Just a library (over Kafka)

Keeping Services Consistent

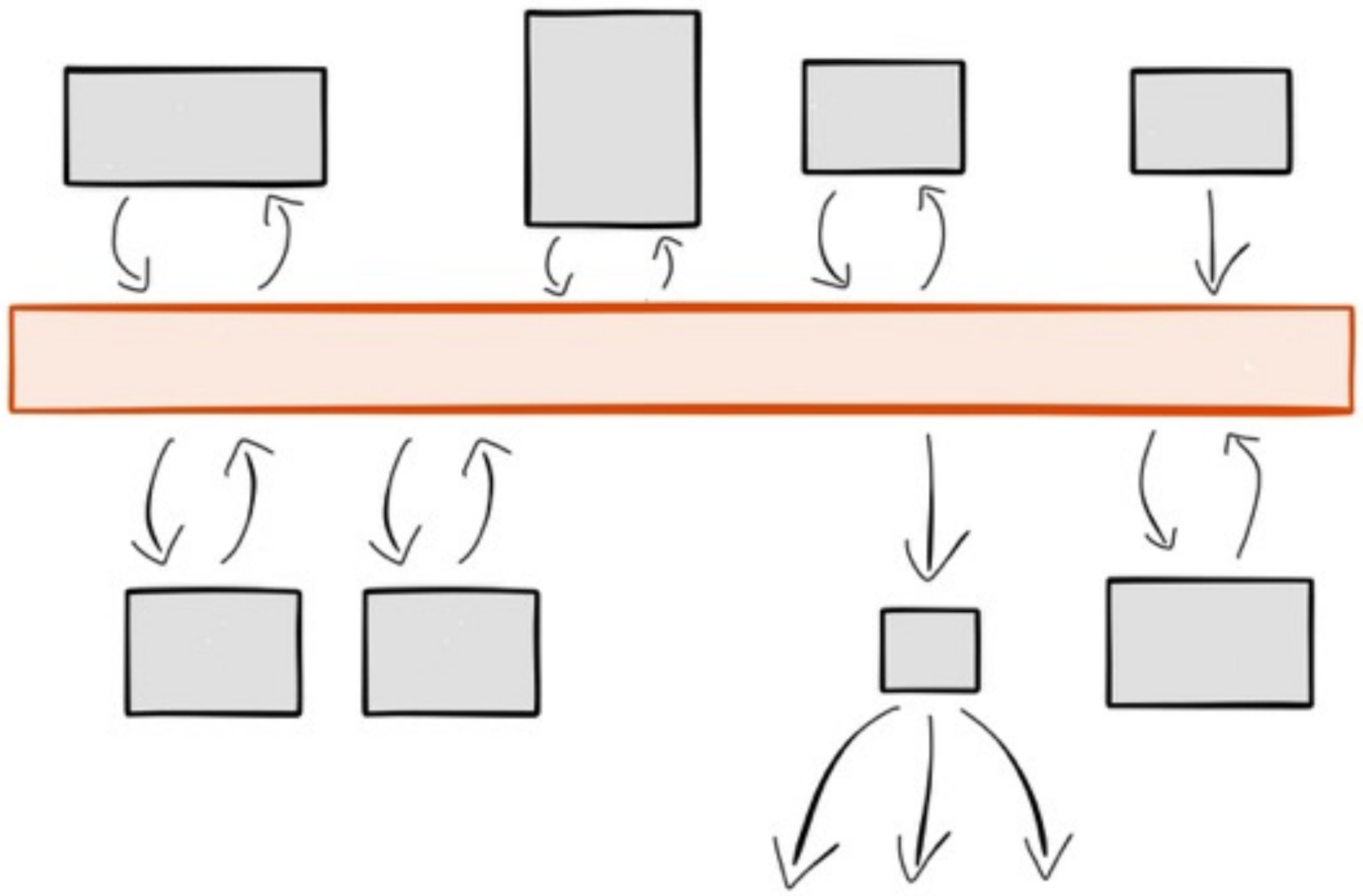
Problem: No BGBSS



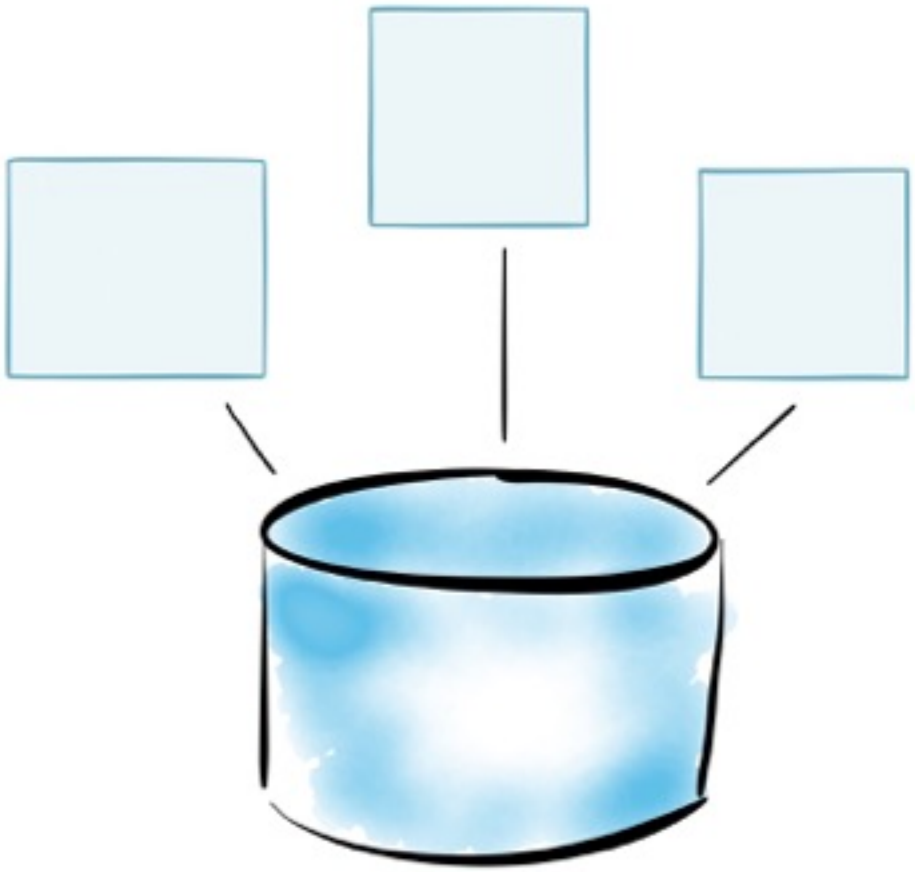
How to you provide the accuracy of this



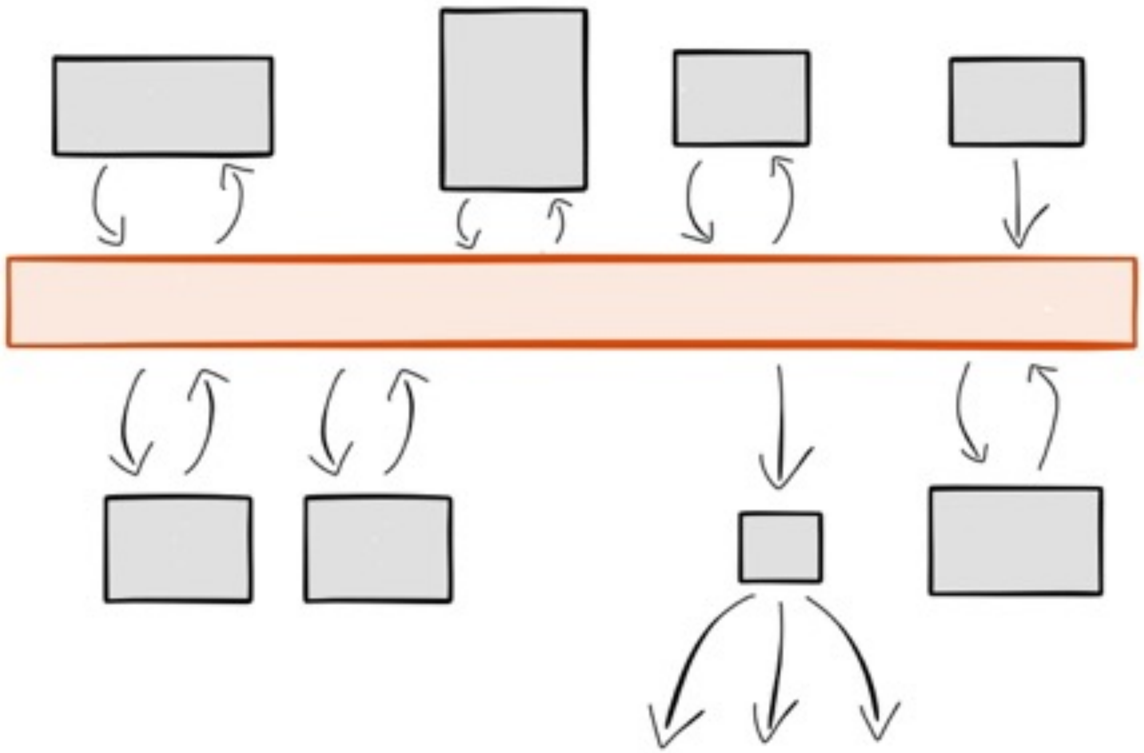
In this?



Centralised vs Federated

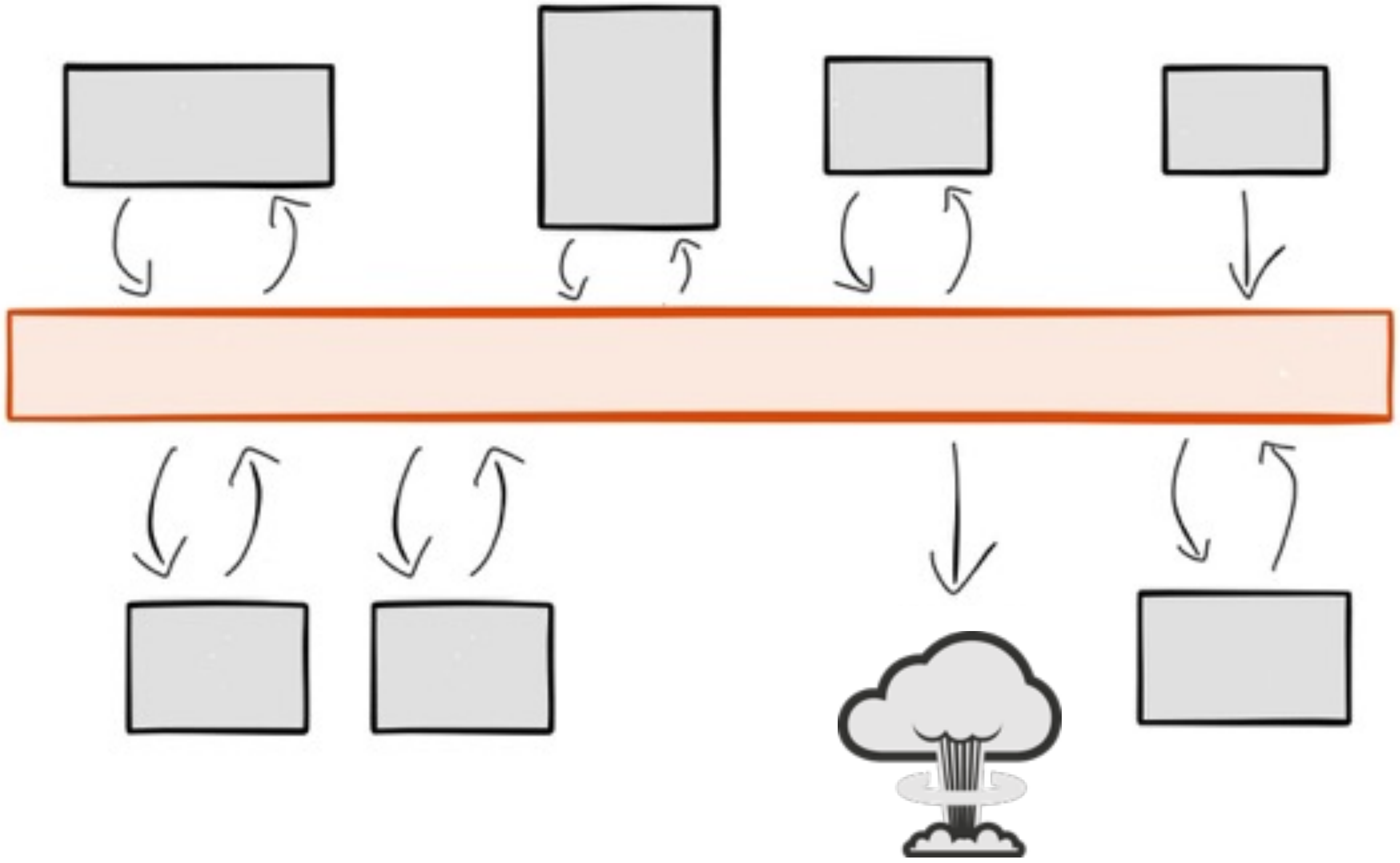


Centralised
consistency model

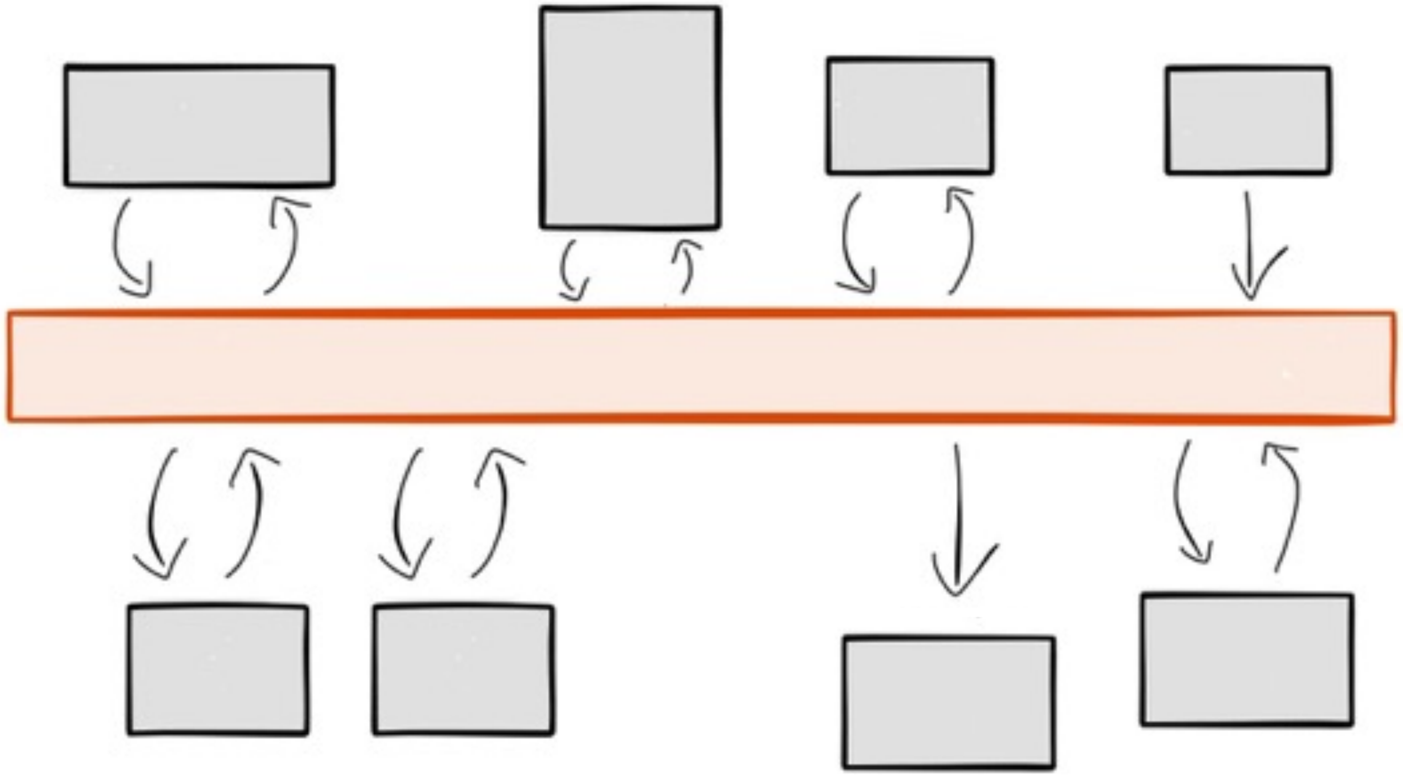


Distributed
consistency model

One problem is failure

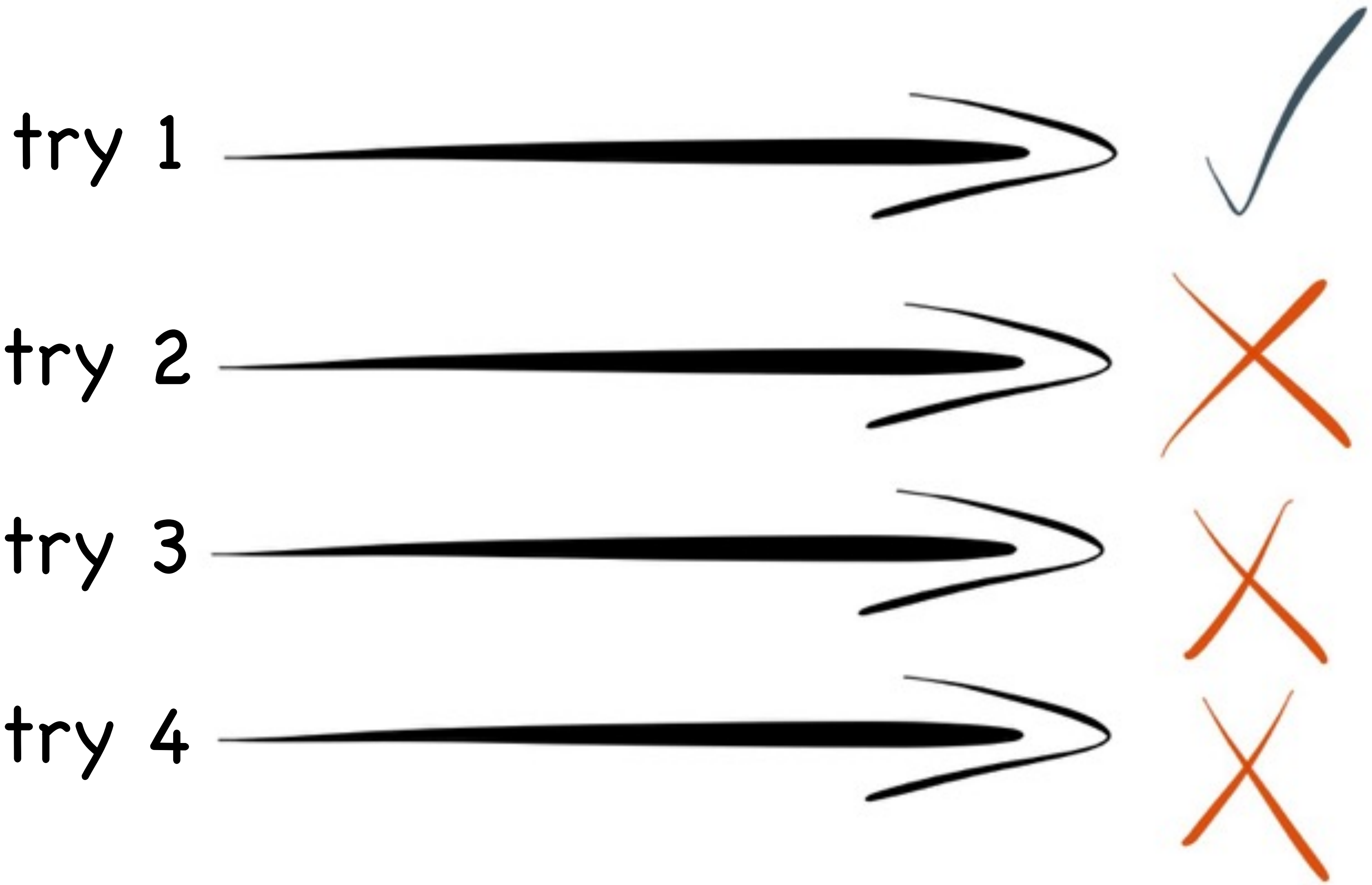


Duplicate messages are inevitable



have I seen
this before?

Make Services Idempotent



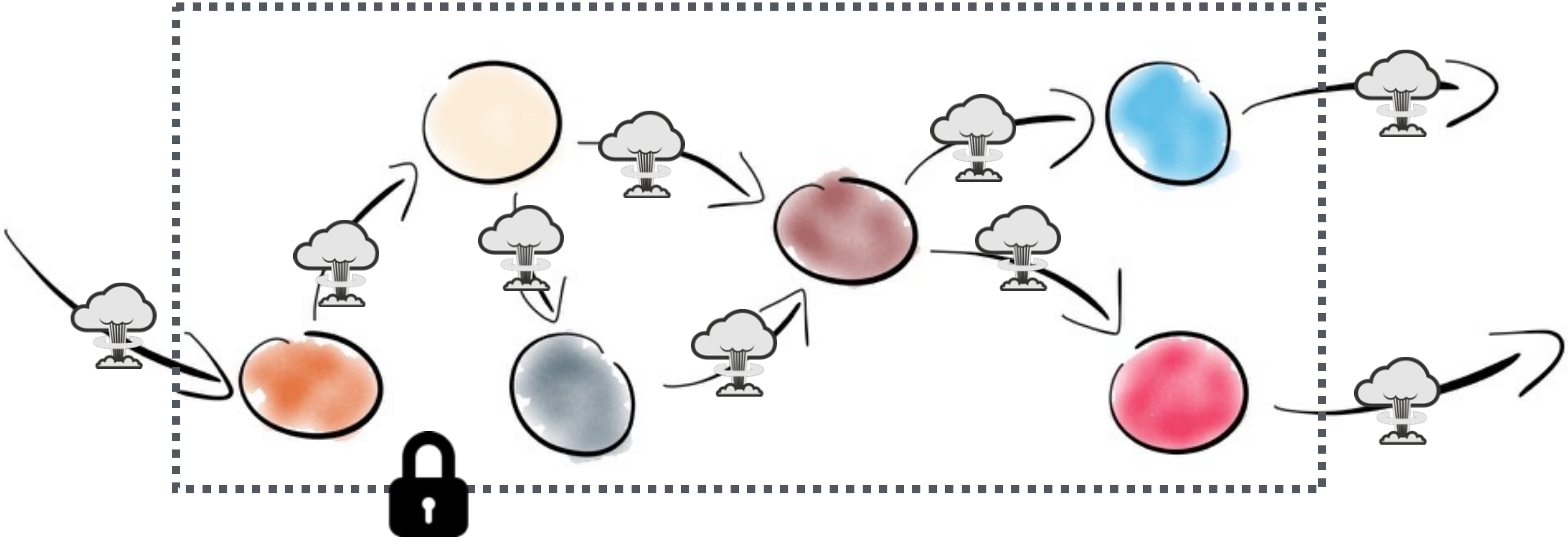
try 1

try 2

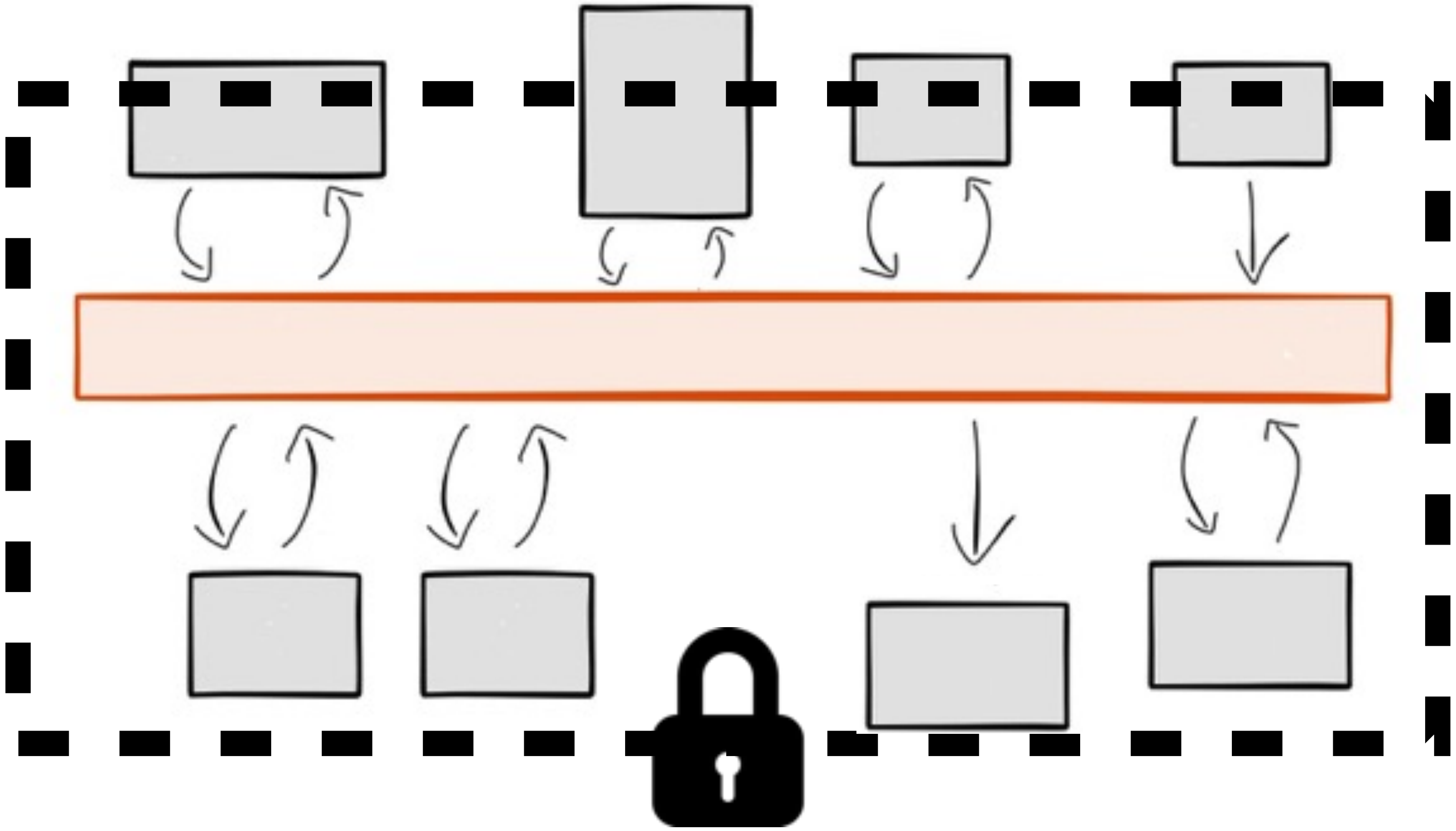
try 3

try 4

Stream processors have to solve this problem



Exactly Once



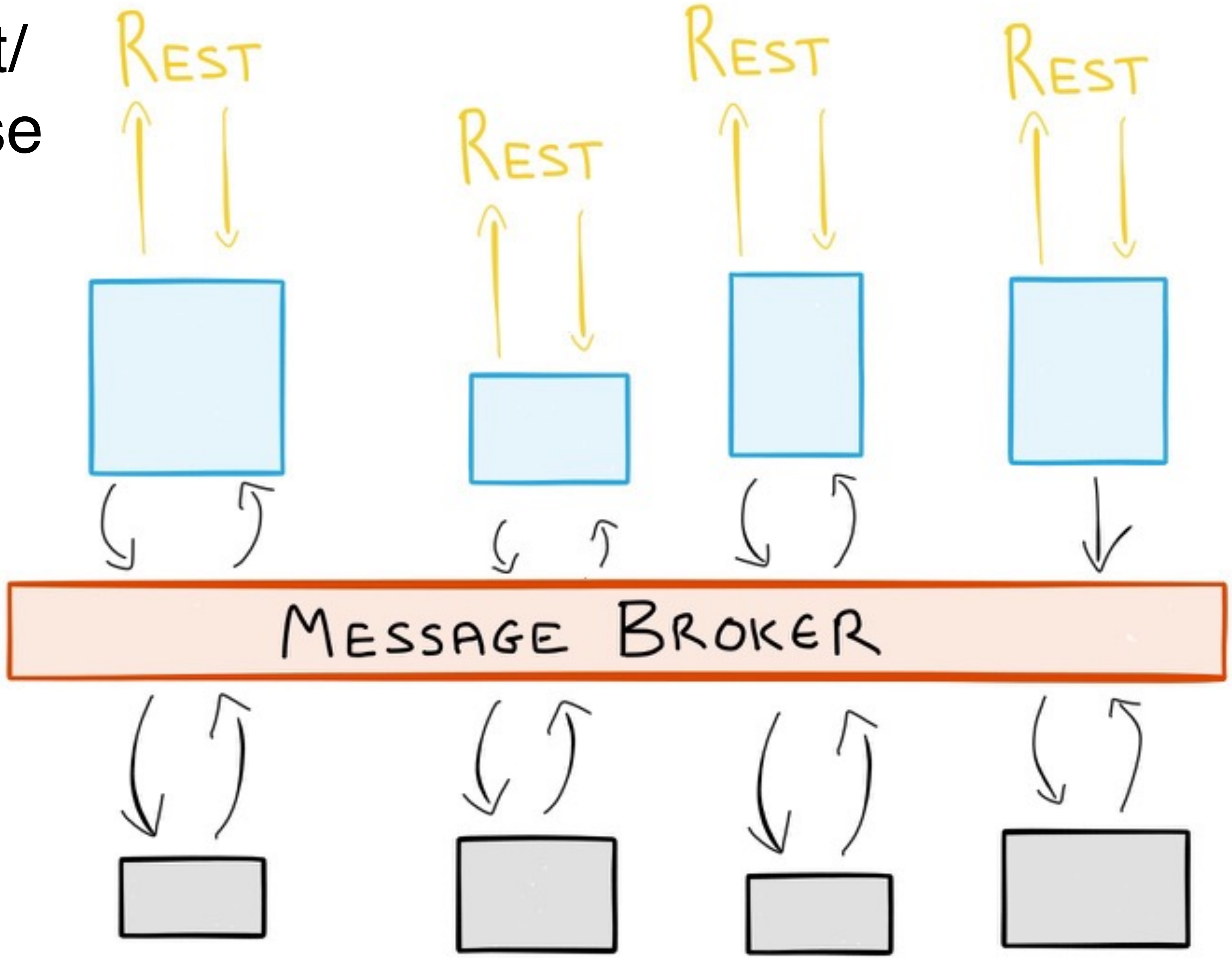
not available in Kafka… yet

So what do we have?

Use Both Approaches

Request/
Response

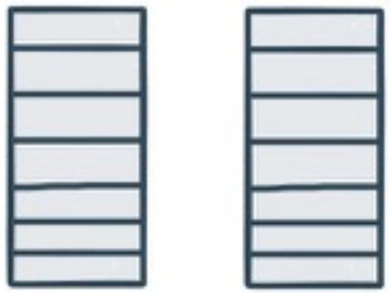
Event-
Based



Queued Delivery System

Ordered queue

Scales Horizontally



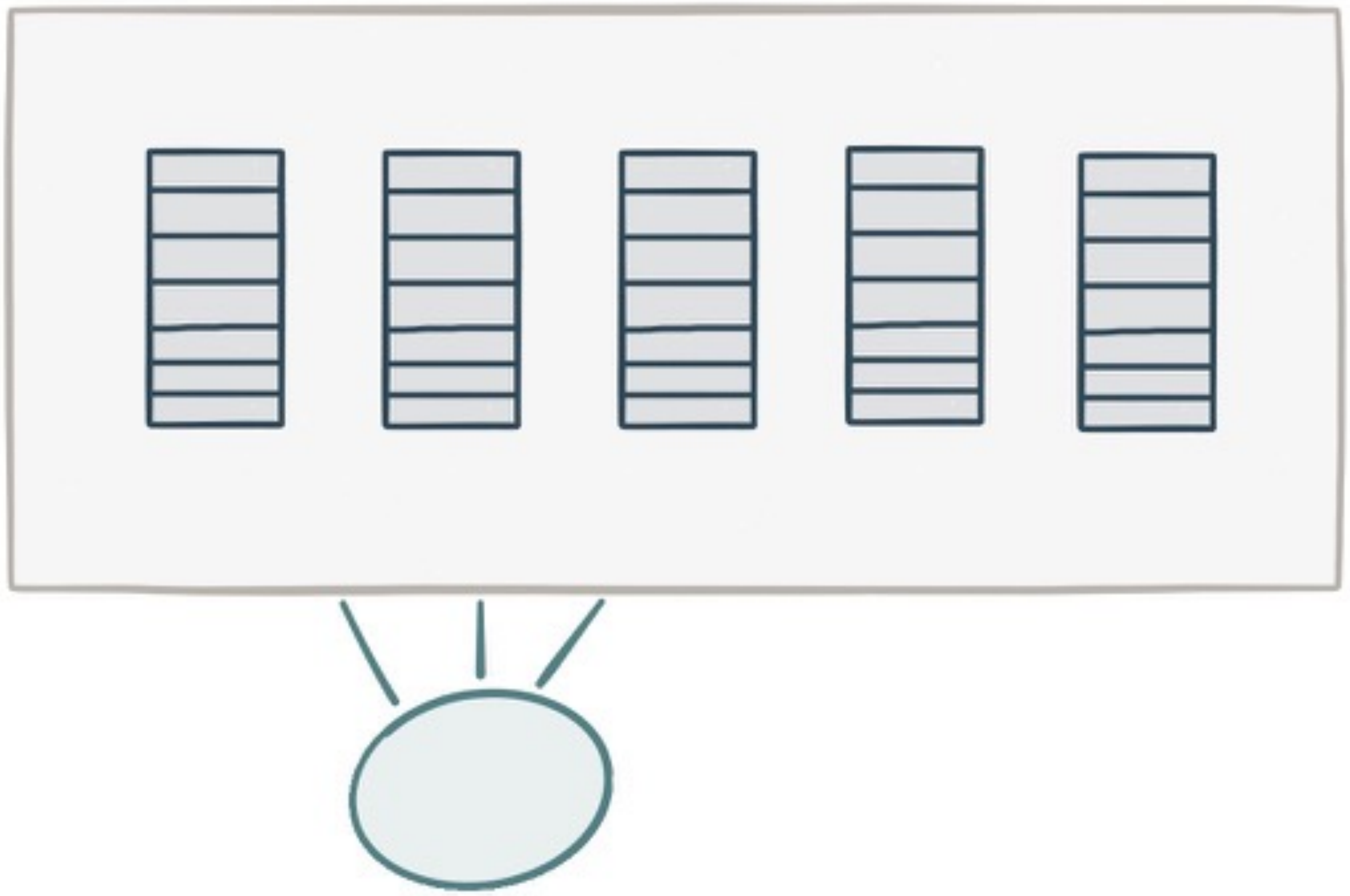
Scales Horizontally

Scales Horizontally

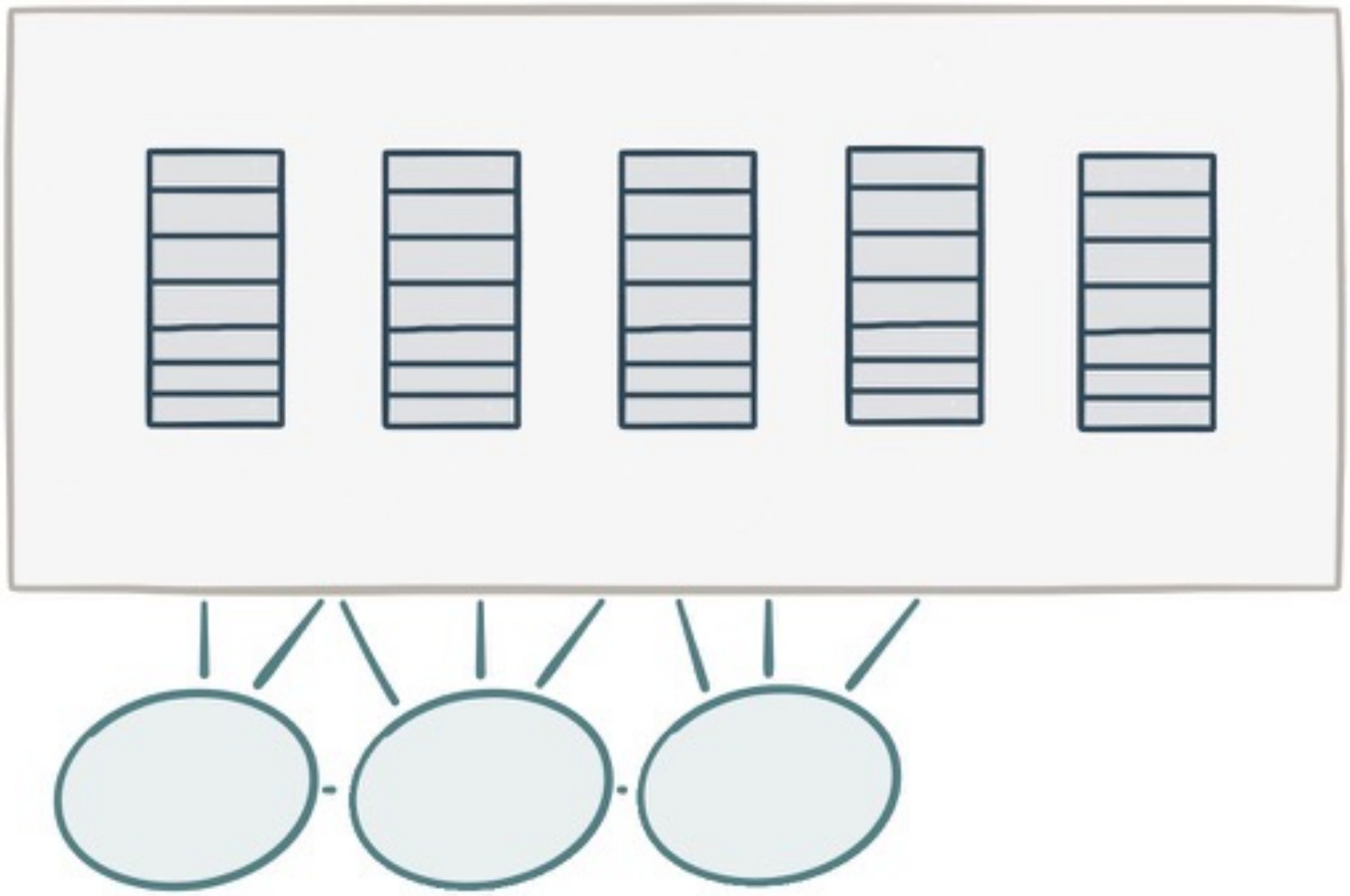
Scales Horizontally

Built In Fault Tolerance

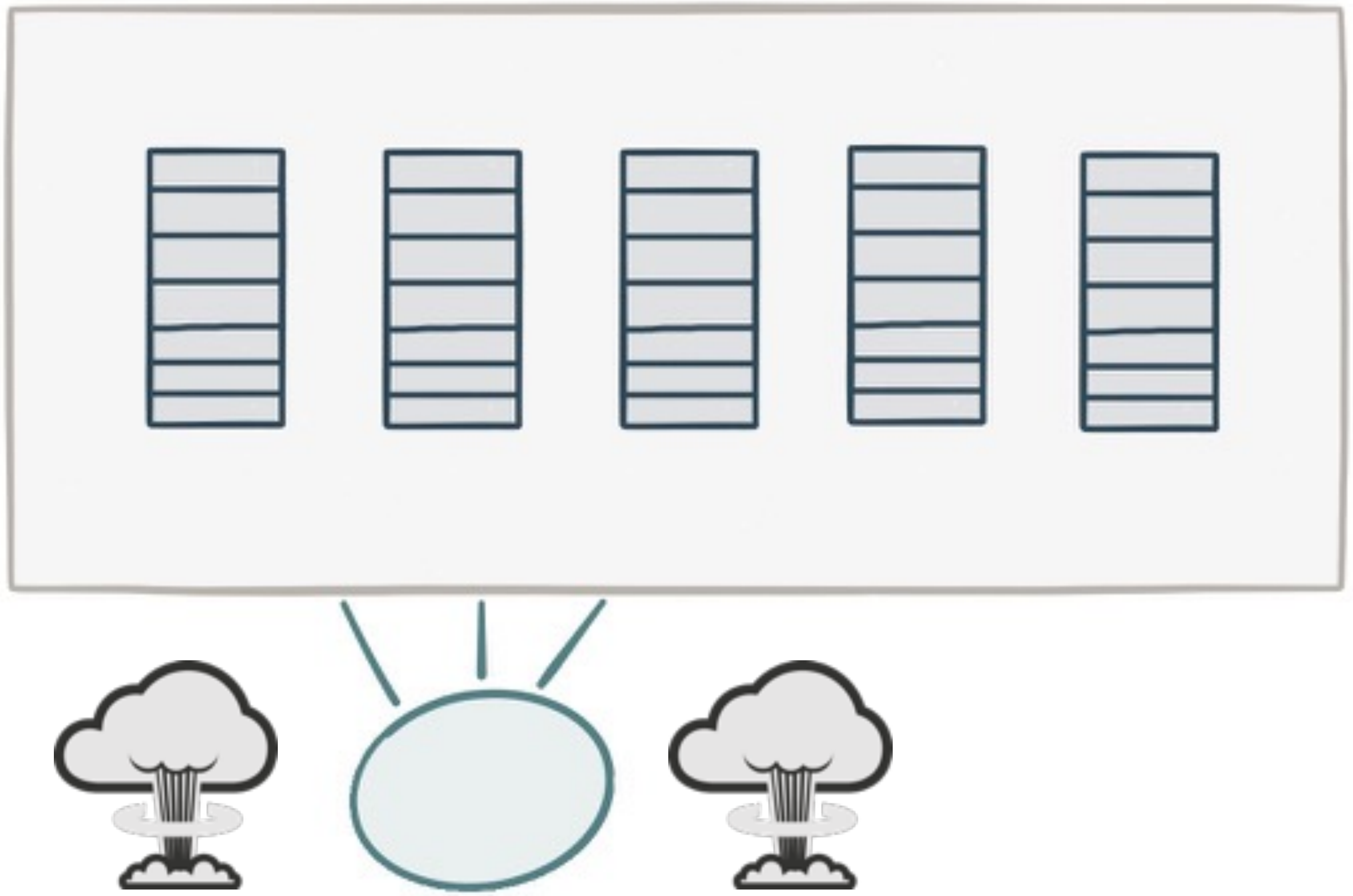
Runs Always On



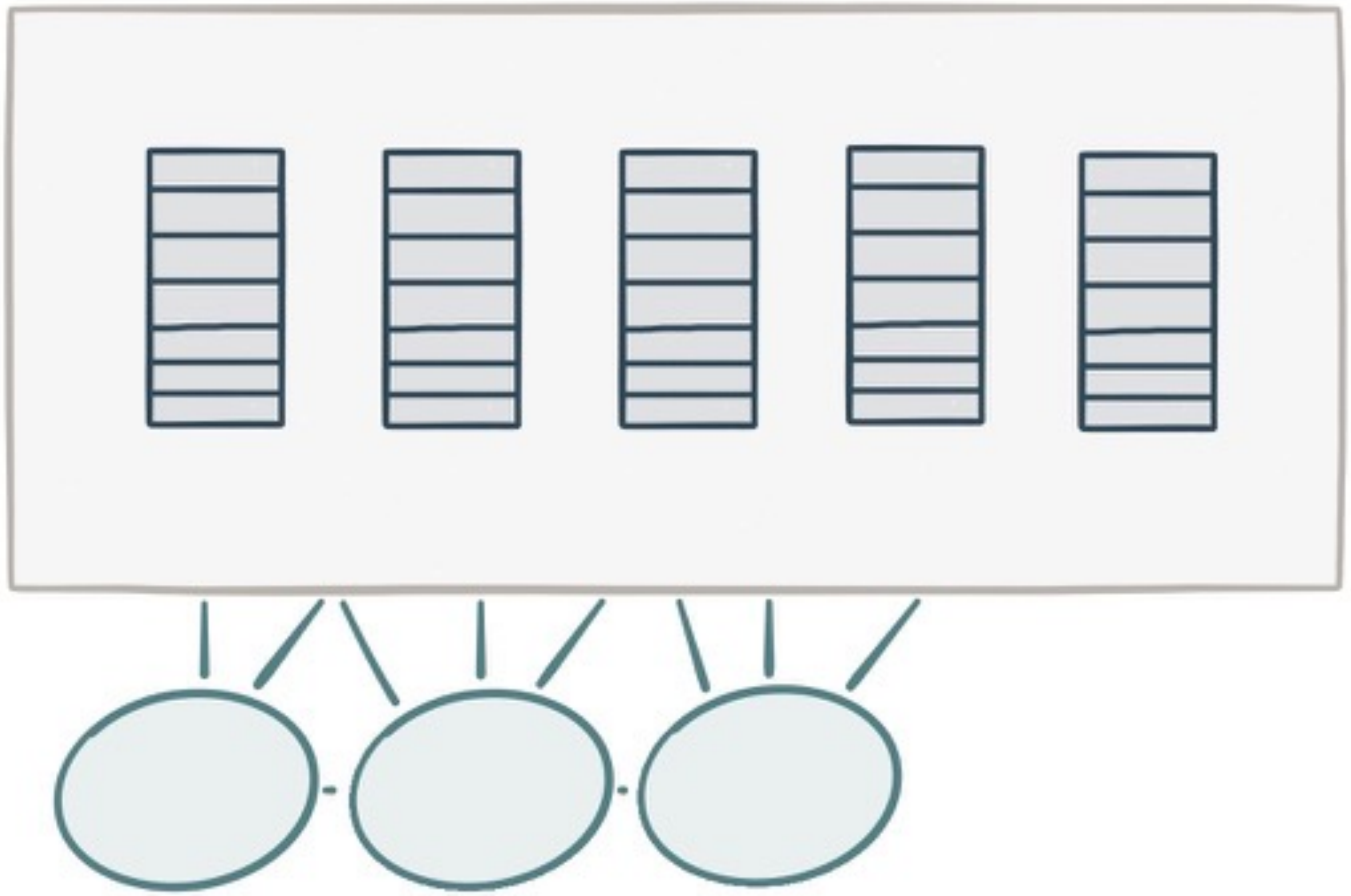
For Services Too



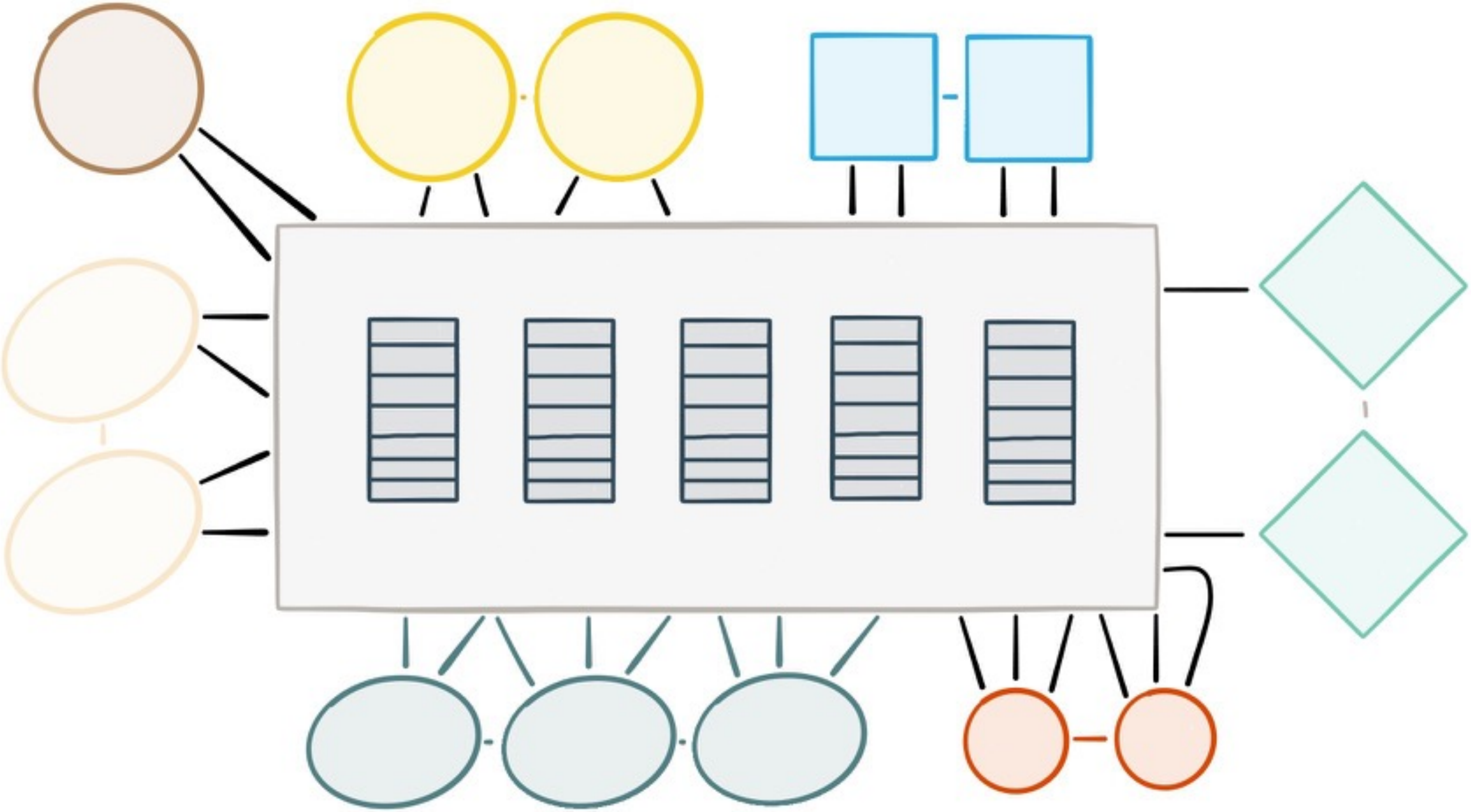
Load Balance



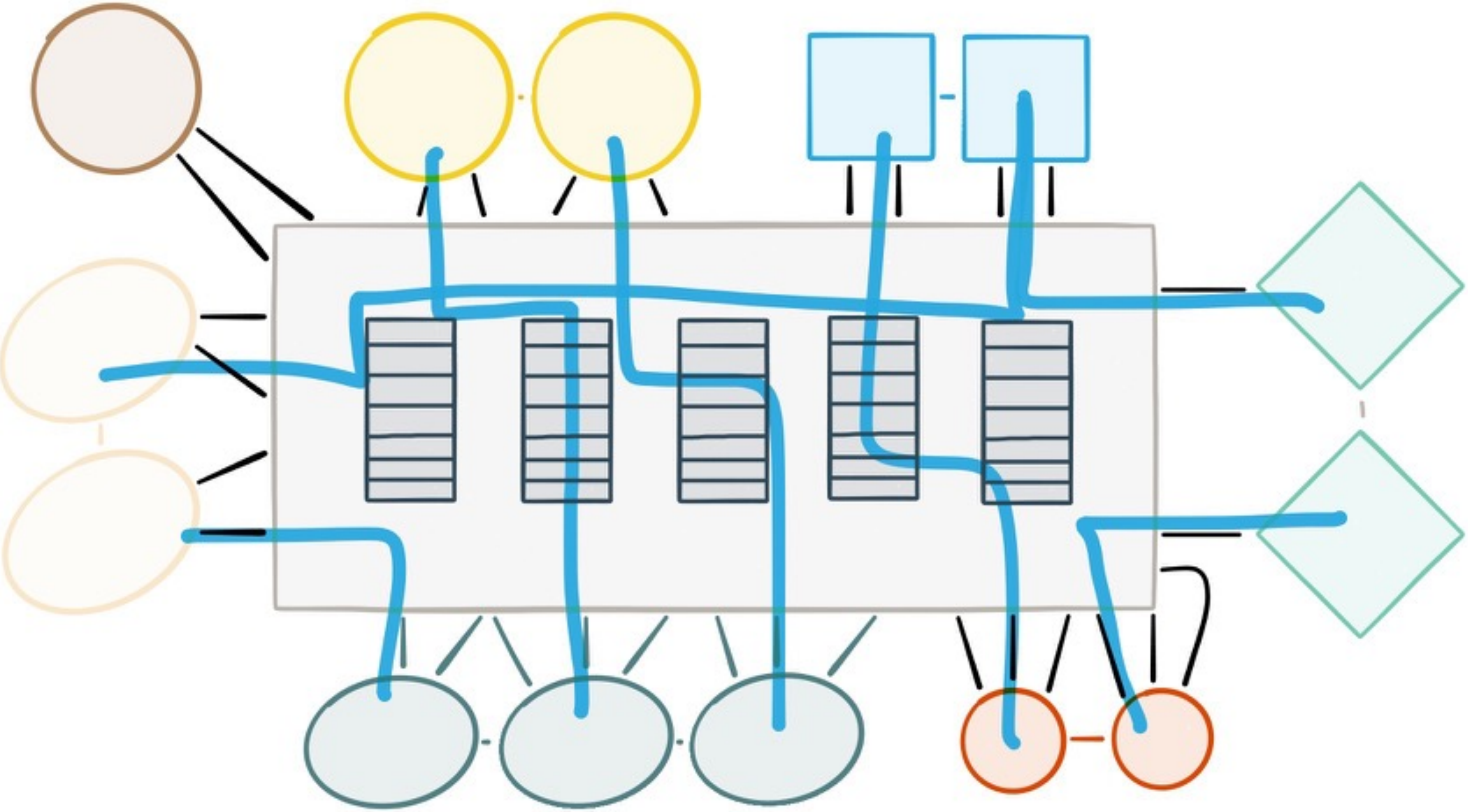
continue through failure



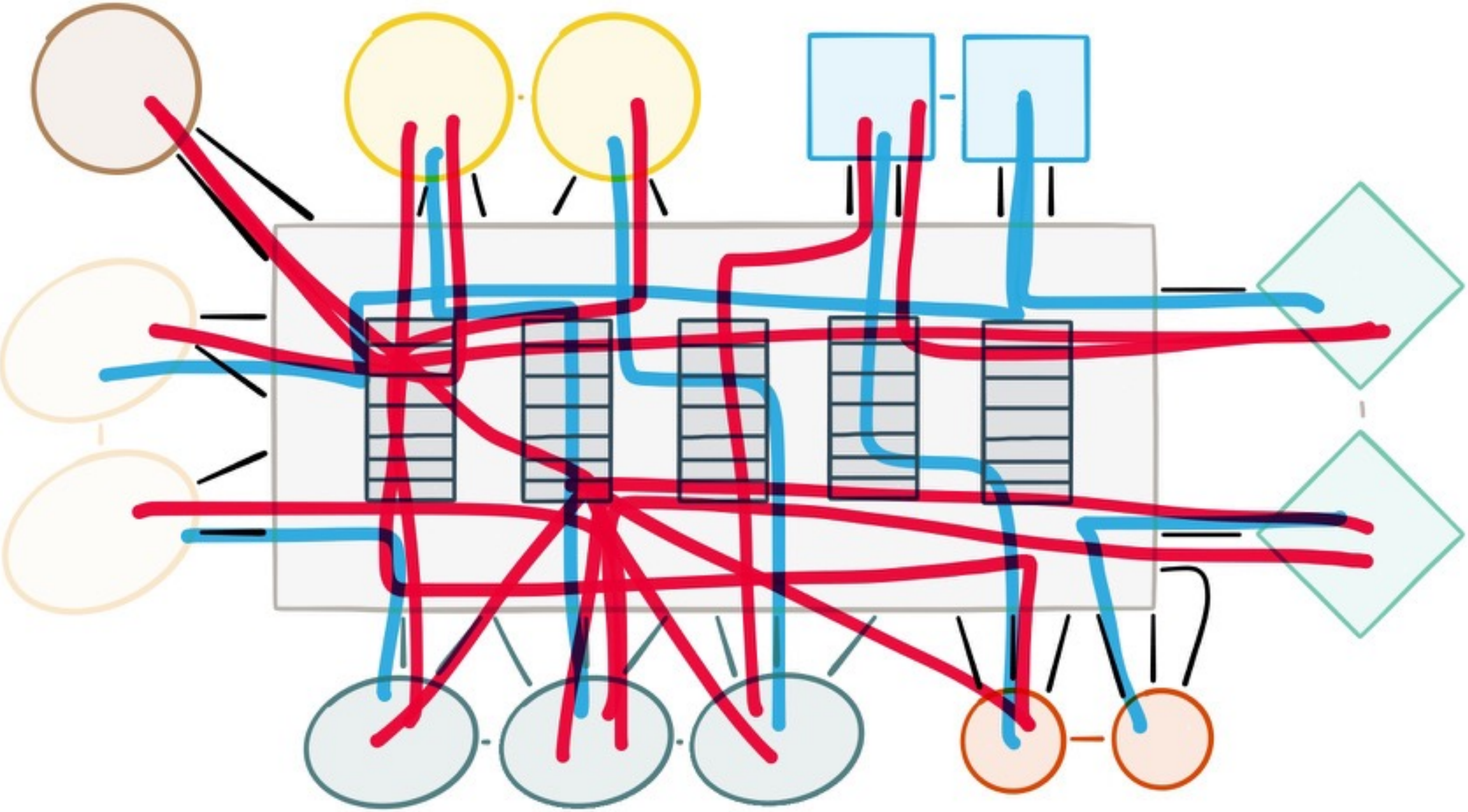
with history stored in the Log



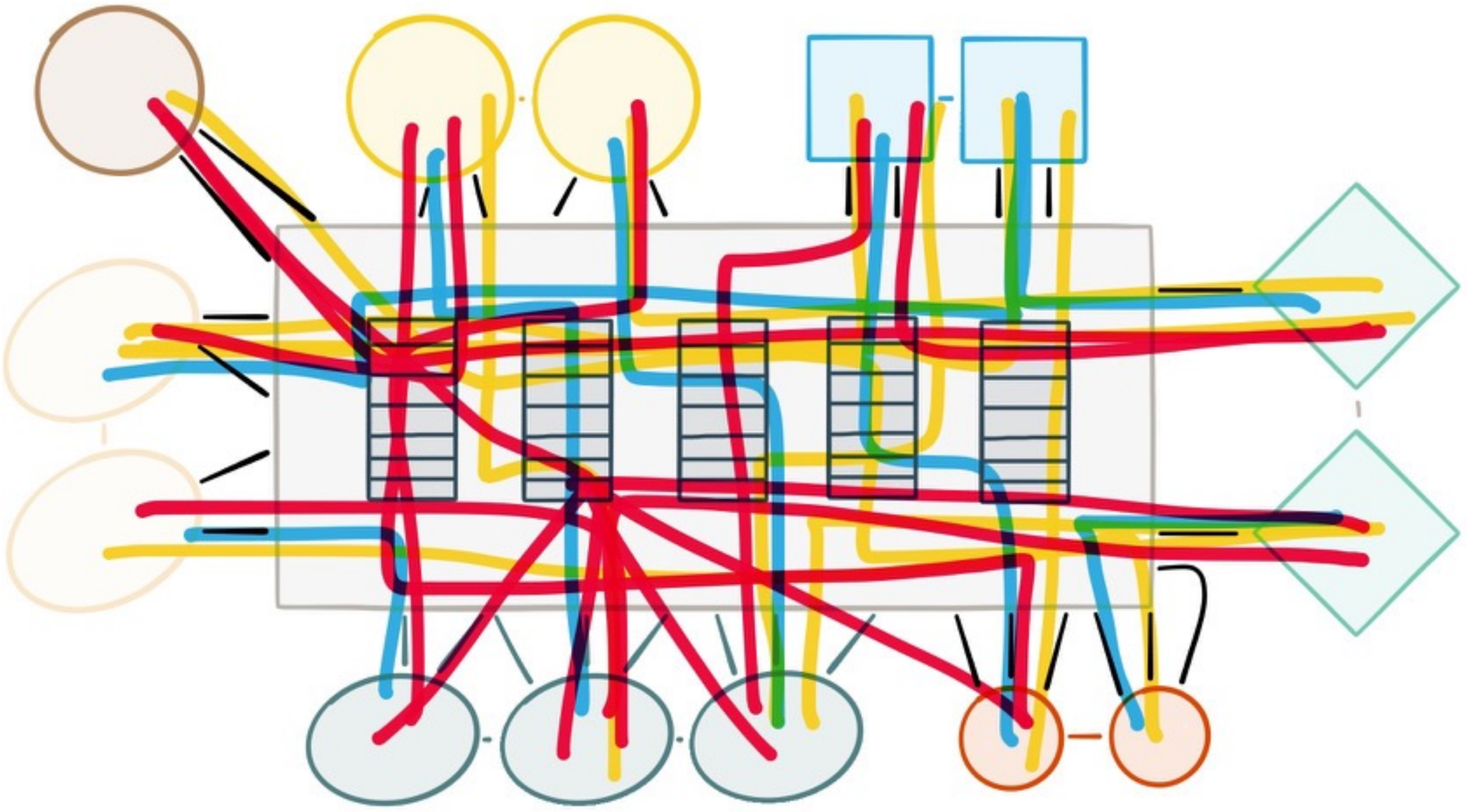
Extending to any number of services



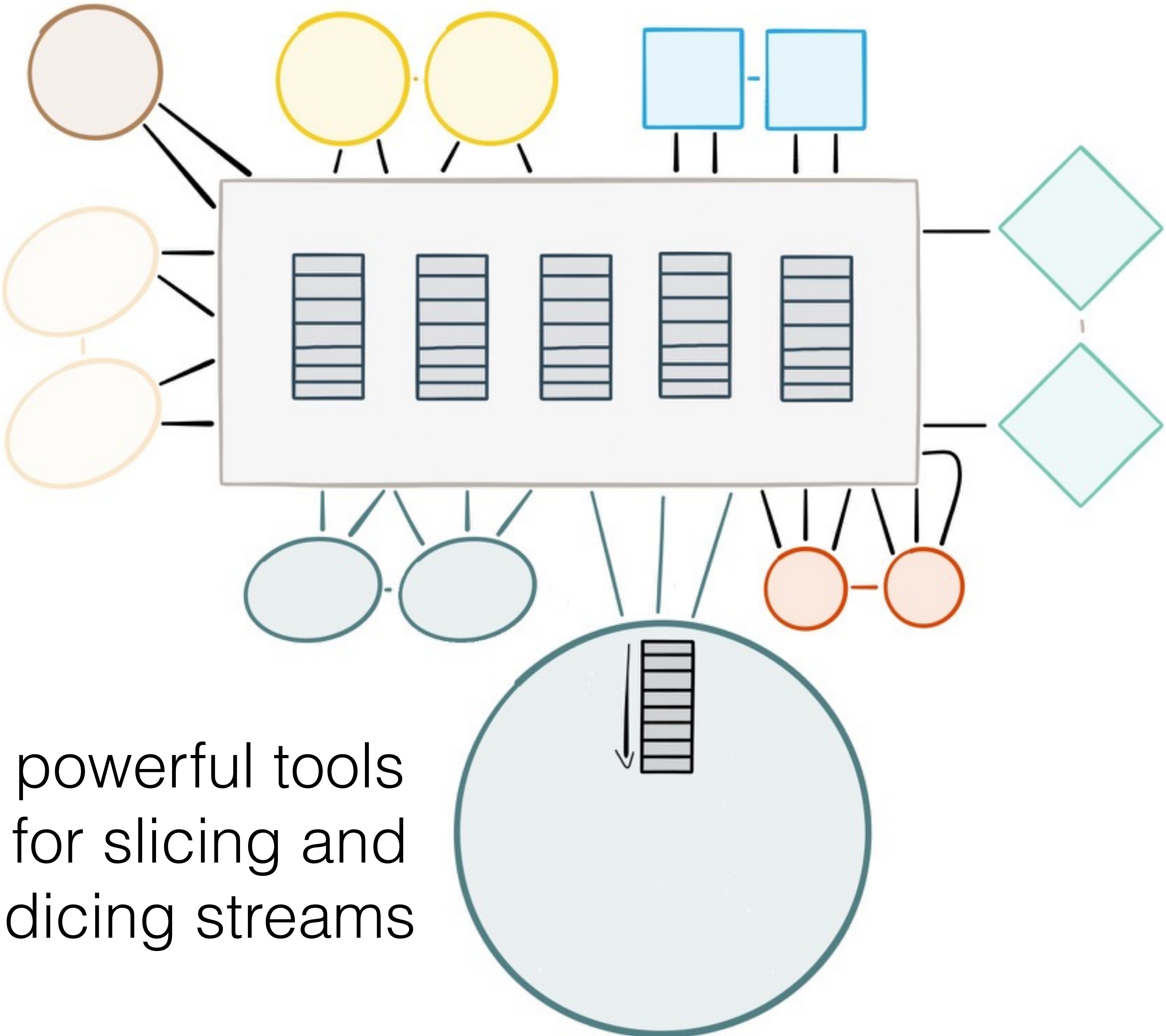
With any data throughput



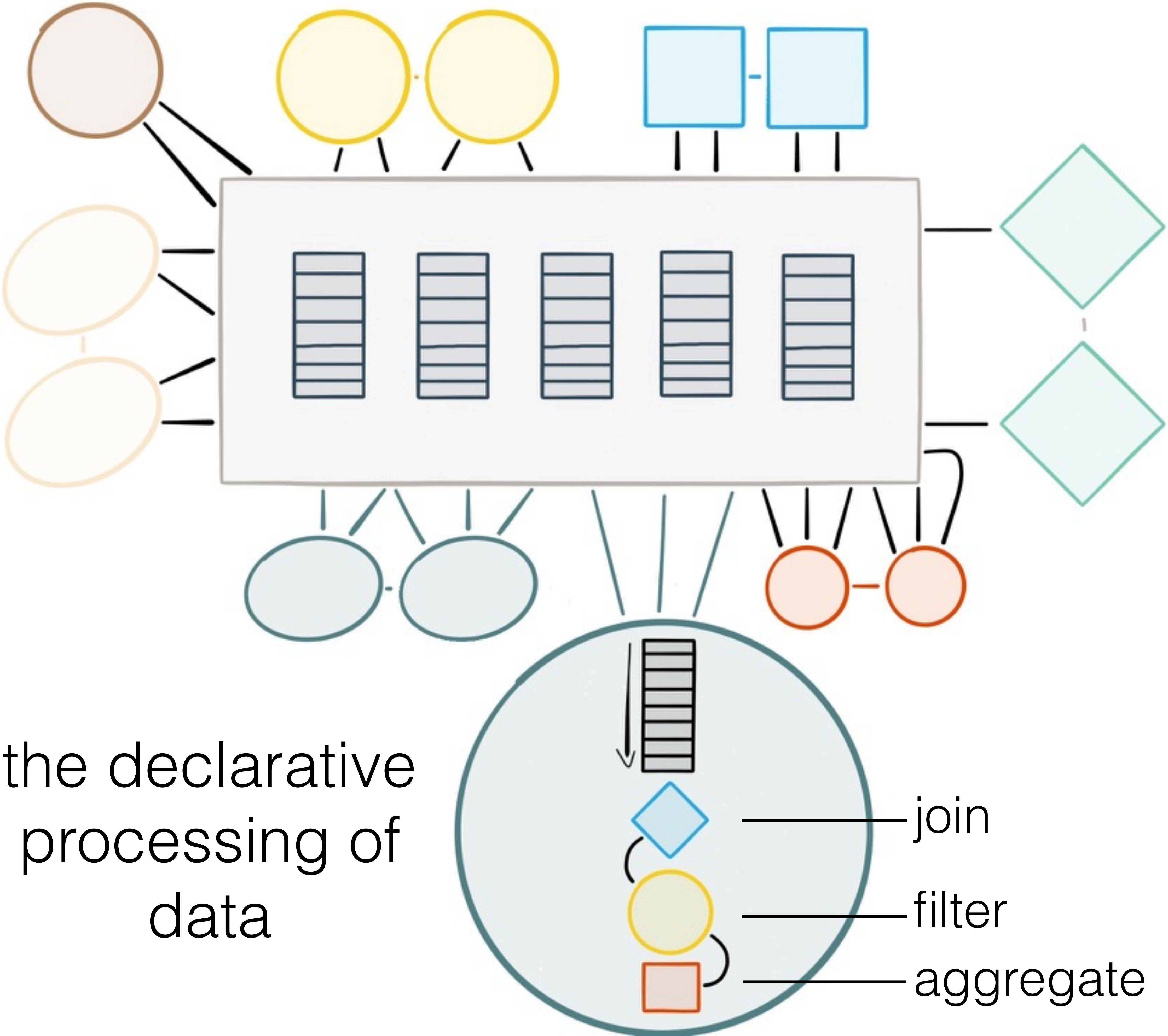
With any data throughput



With any data throughput



powerful tools
for slicing and
dicing streams

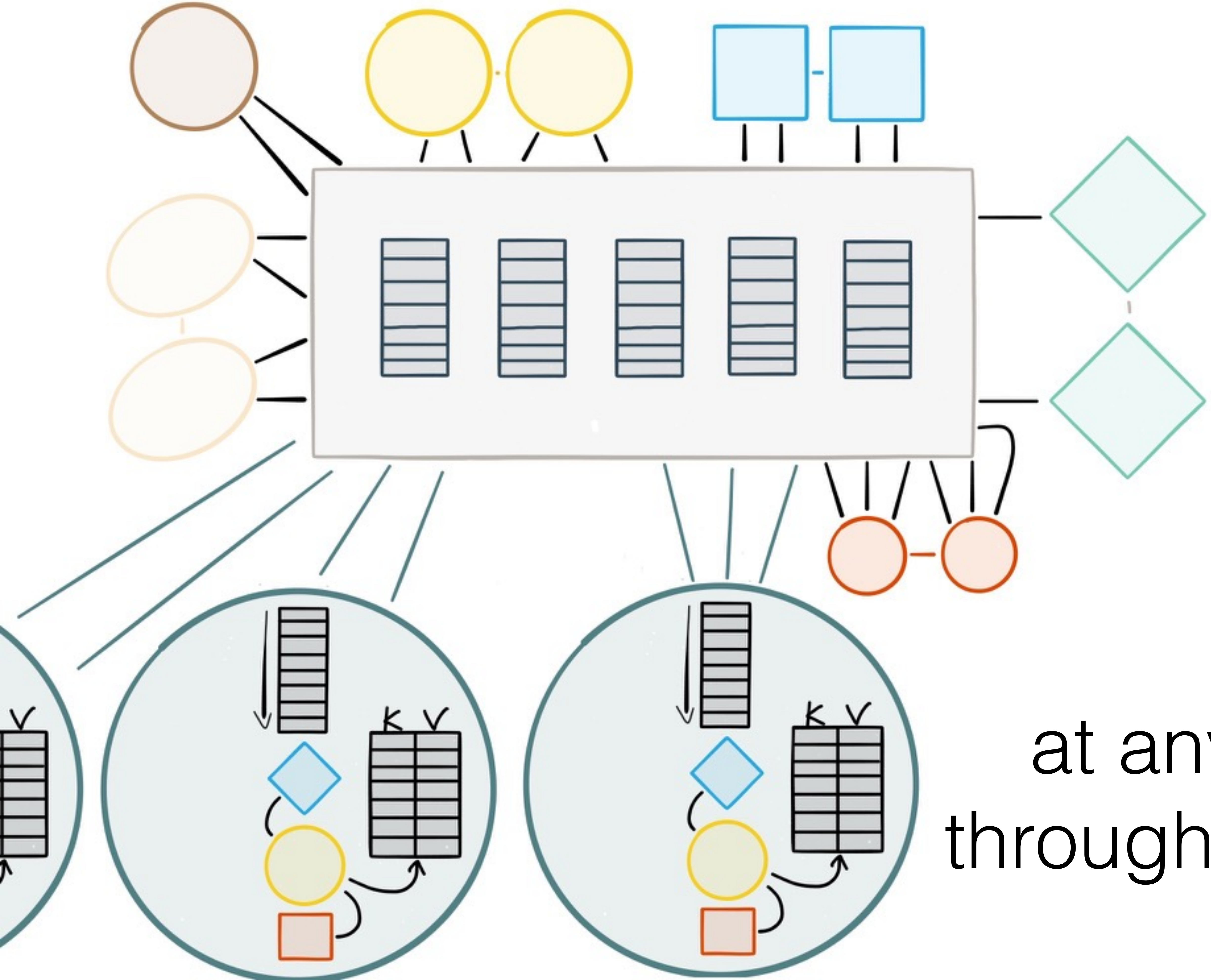


the declarative processing of data

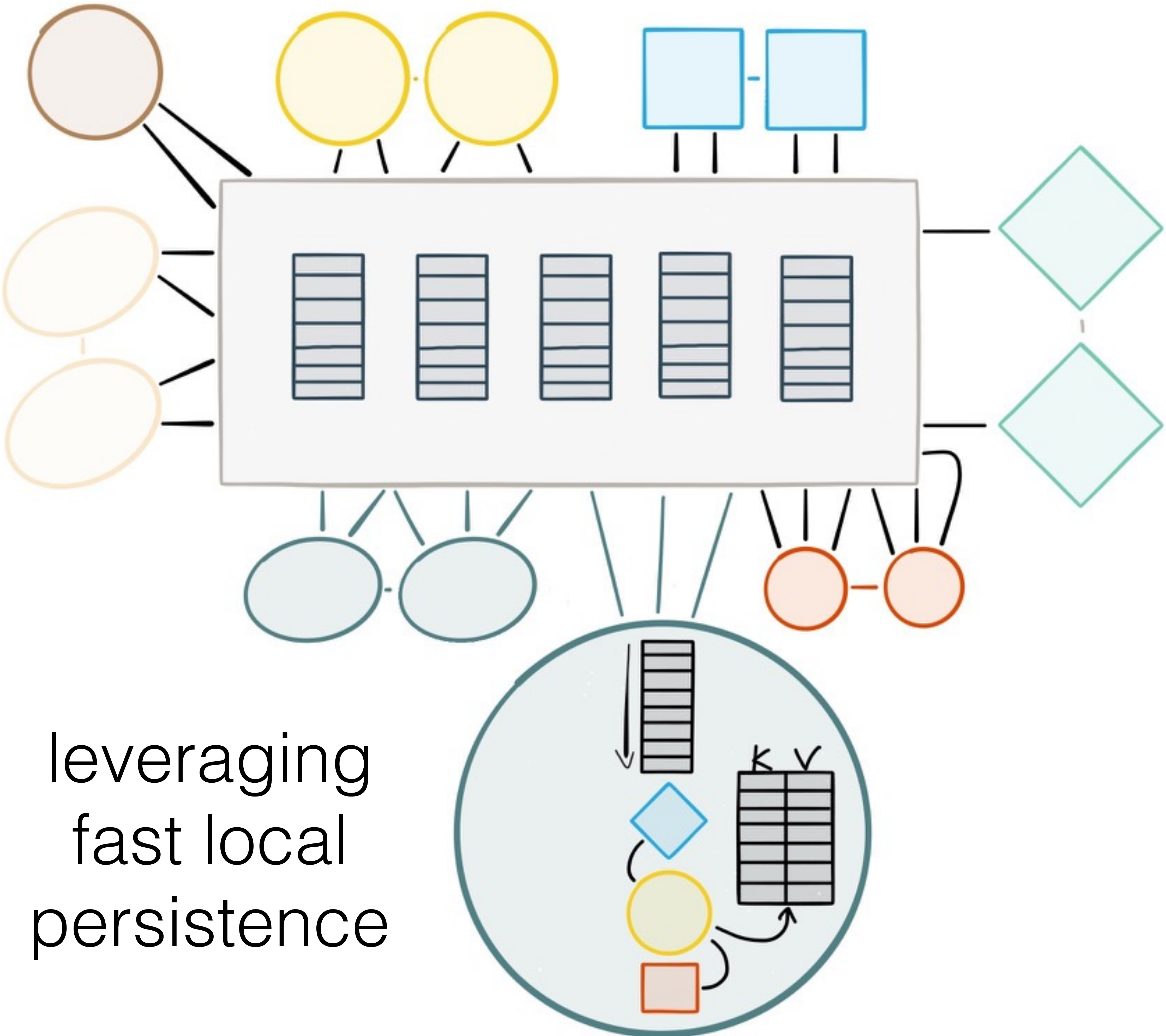
join

filter

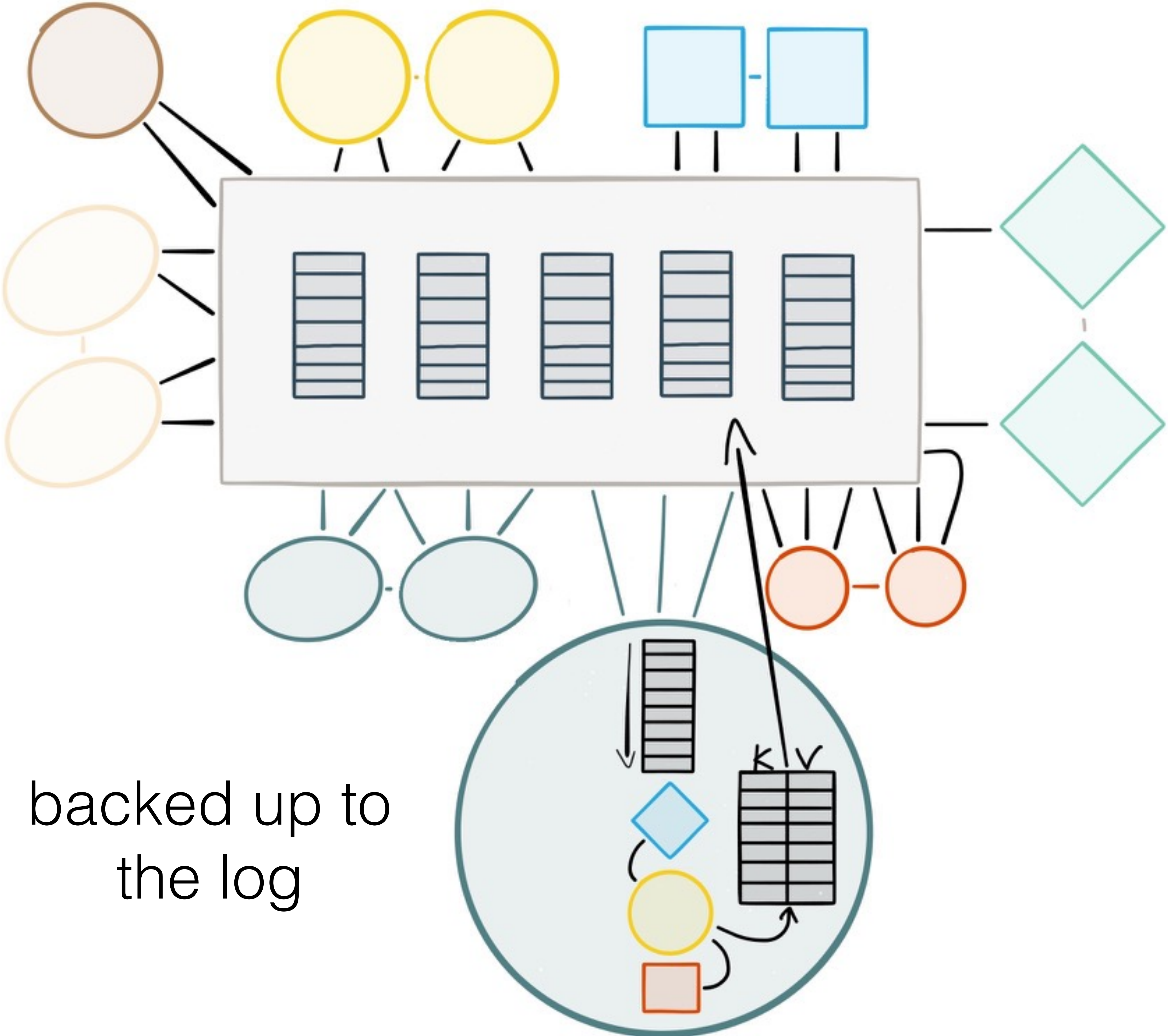
aggregate



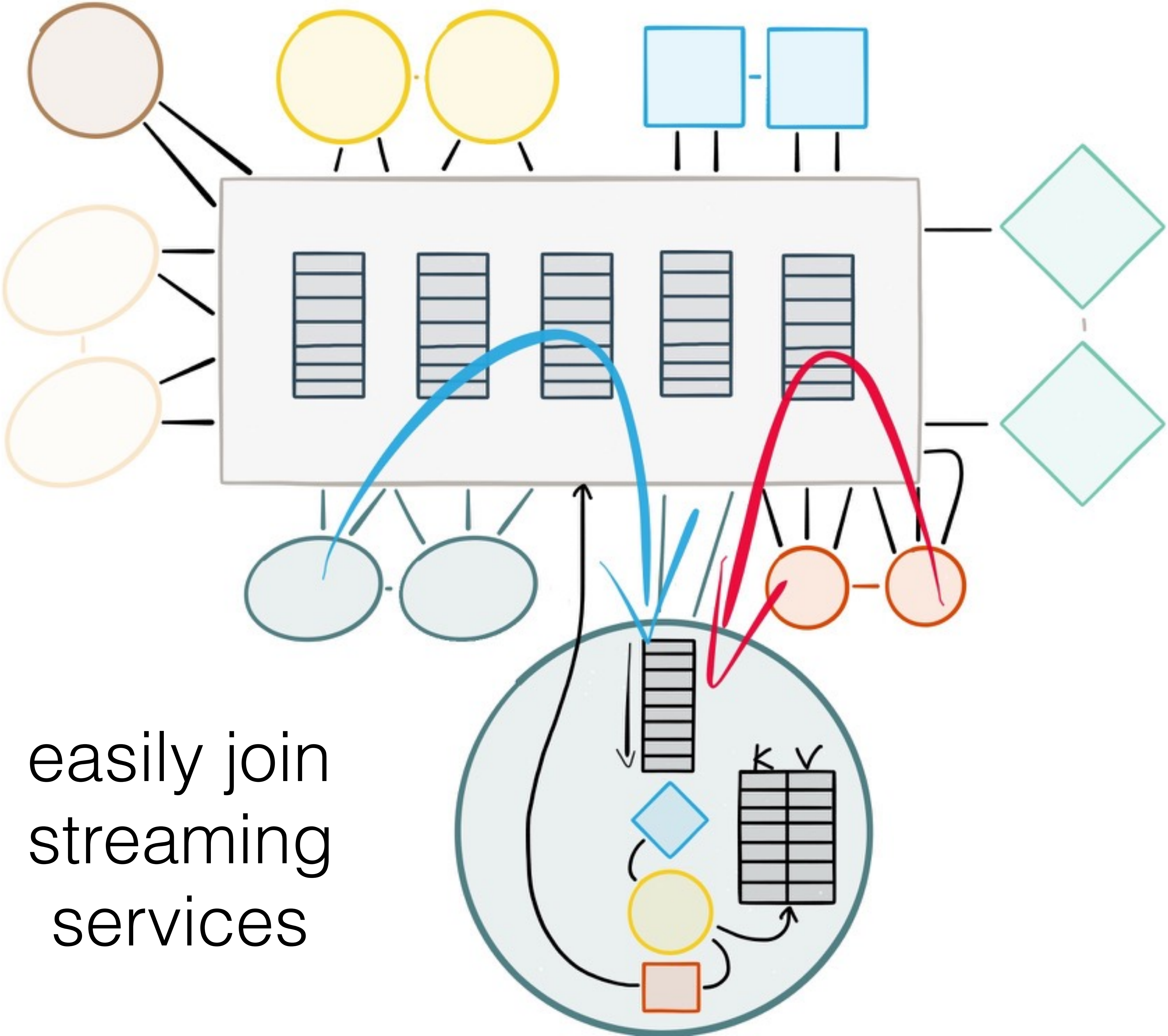
at any throughput



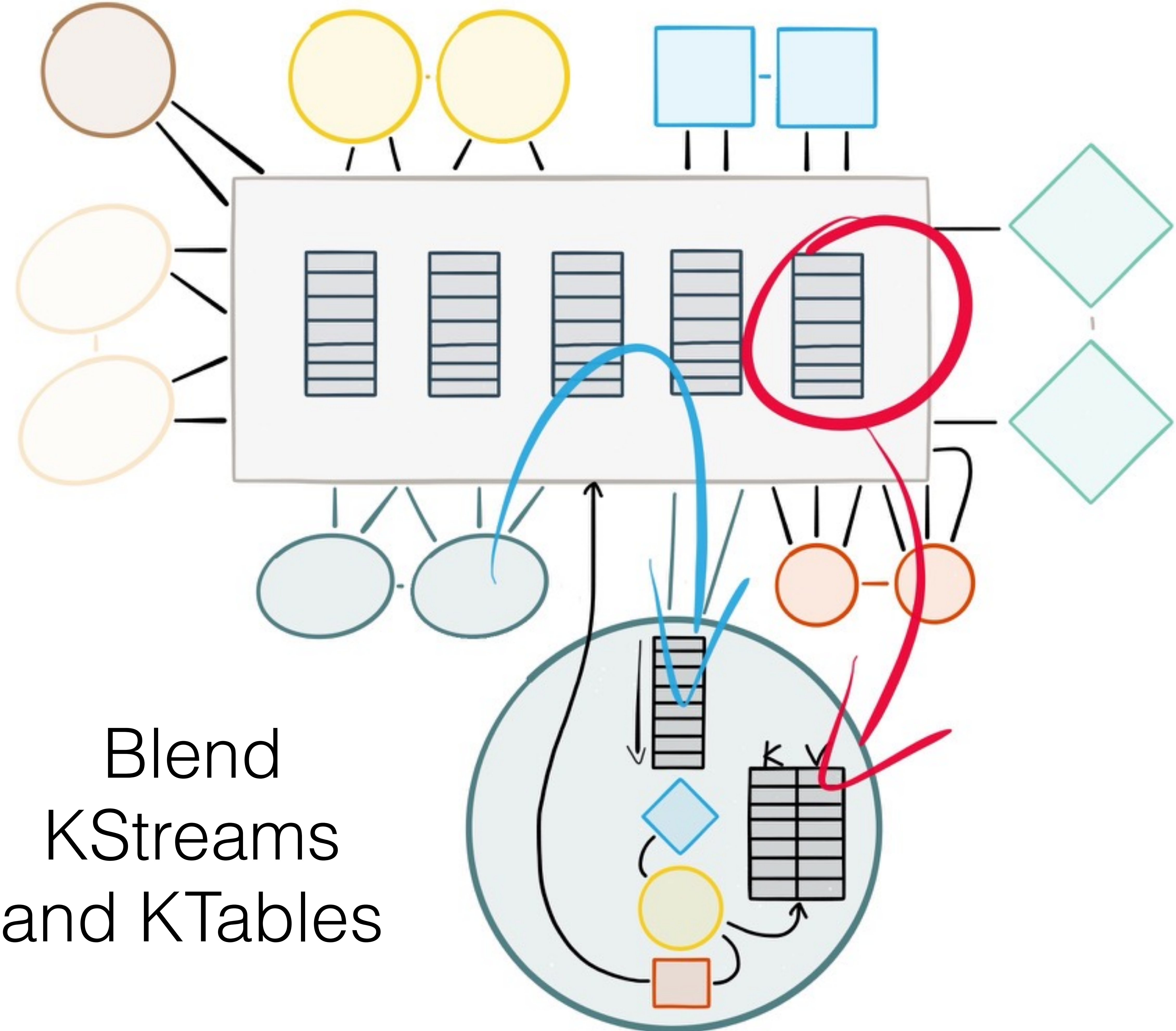
leveraging fast local persistence



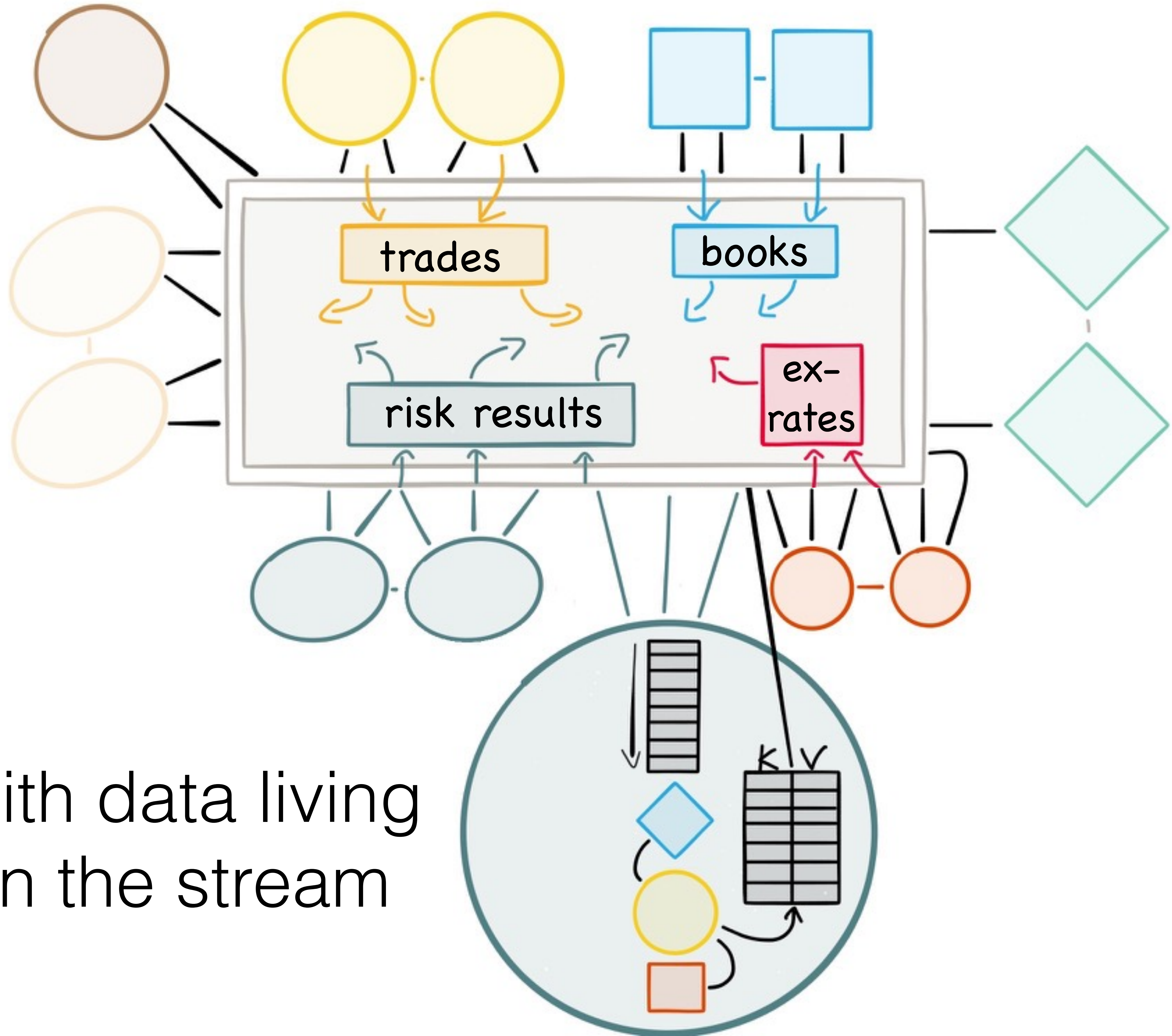
backed up to
the log



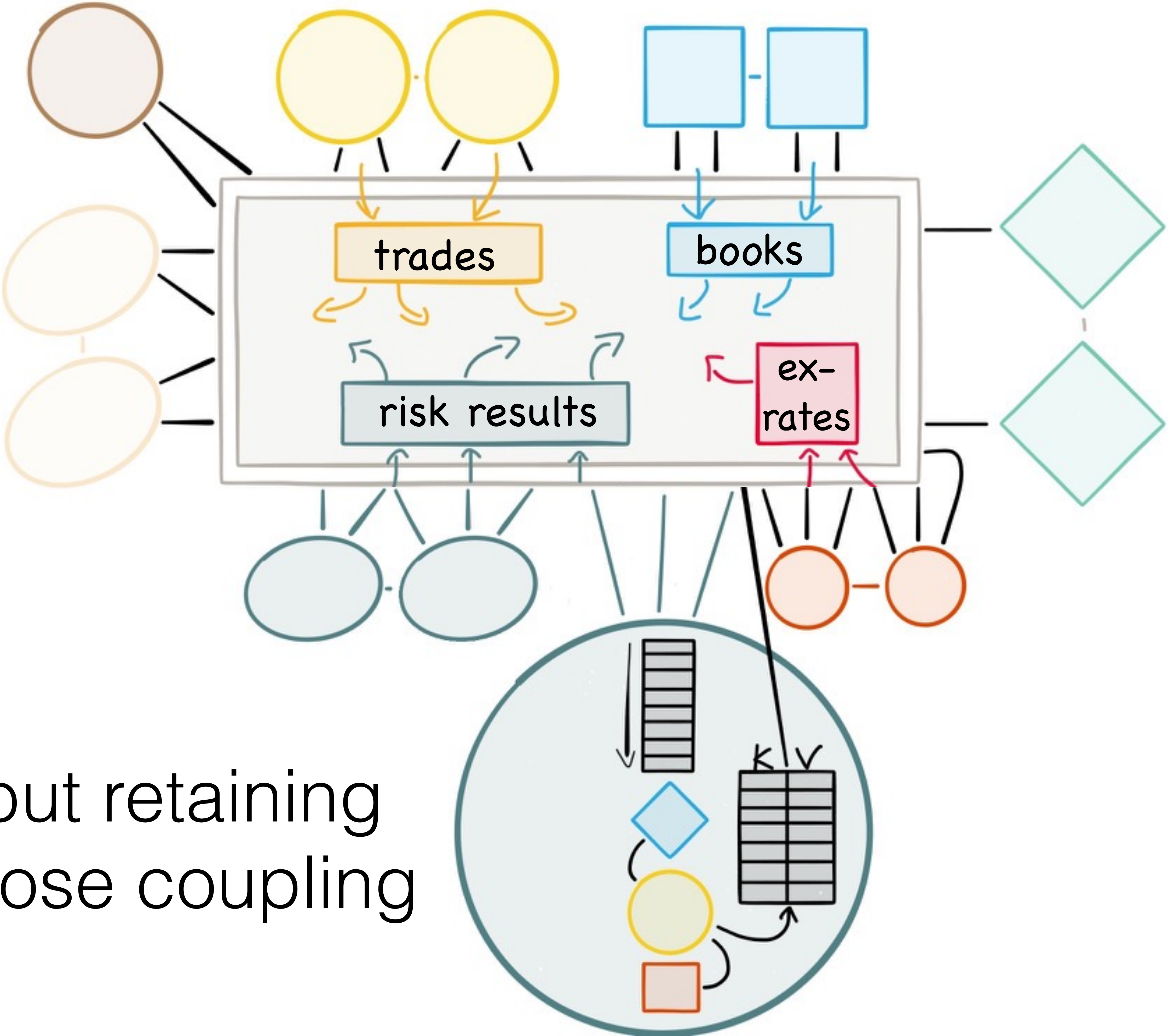
easily join
streaming
services



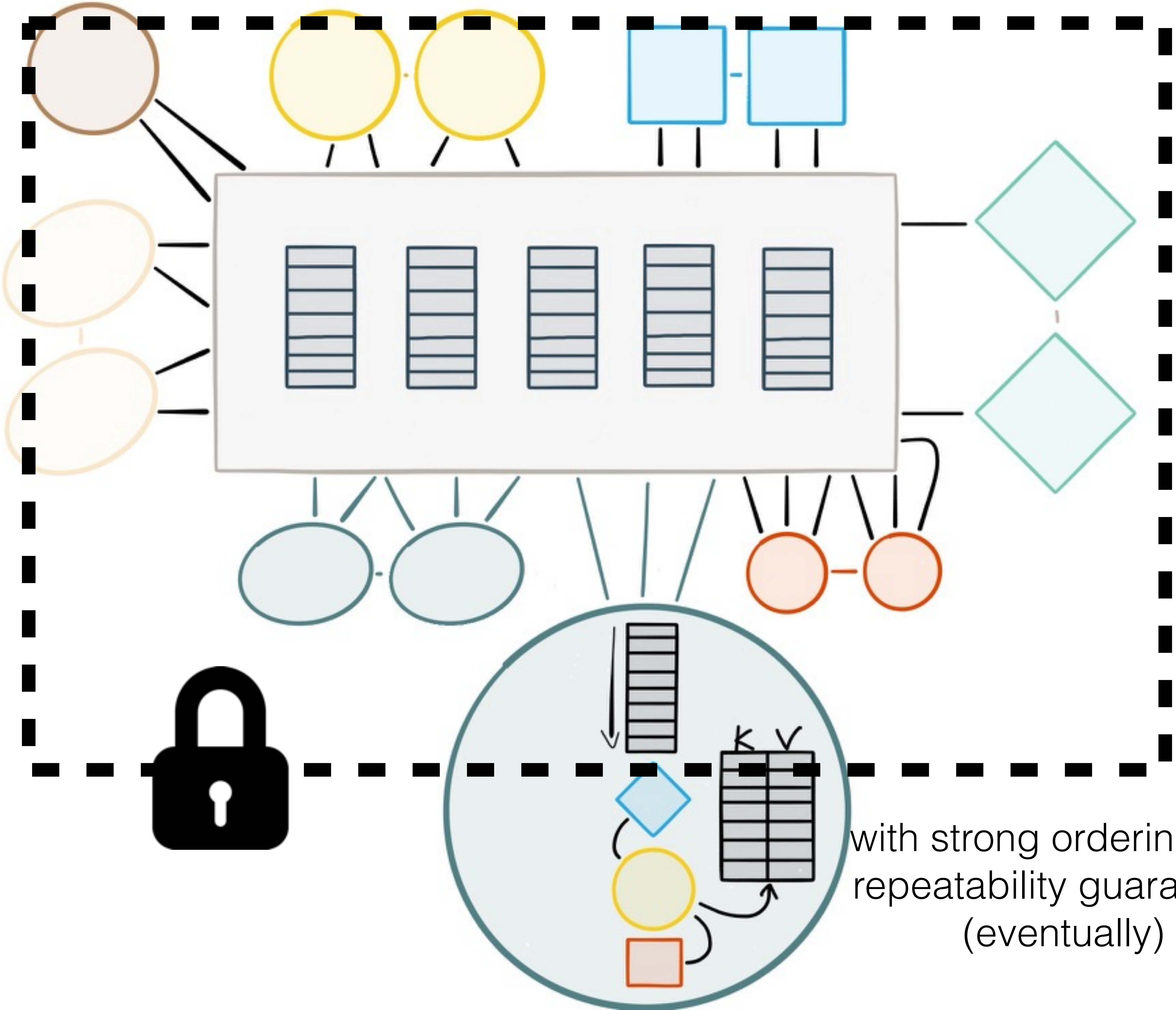
Blend KStreams and KTables



with data living
in the stream



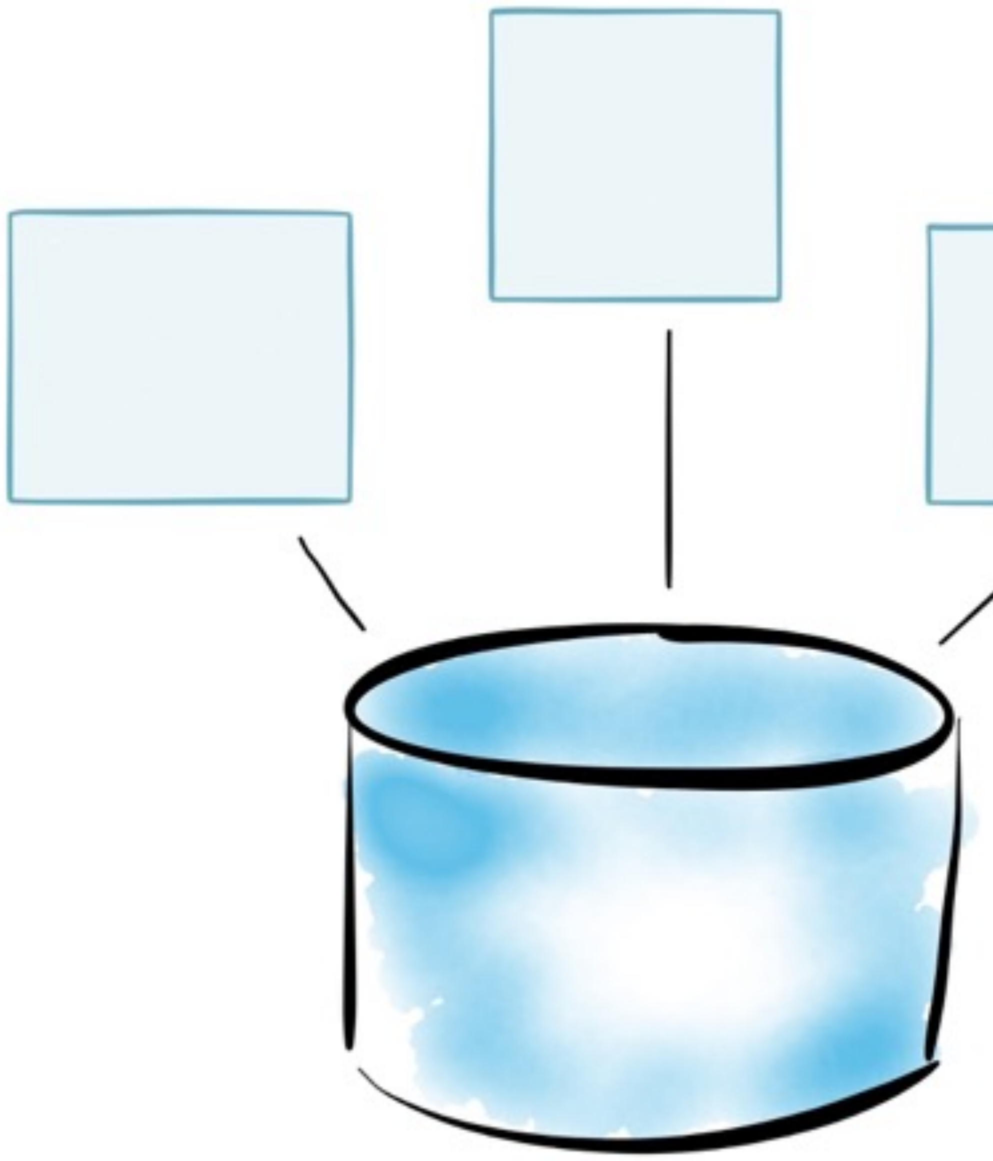
but retaining
loose coupling



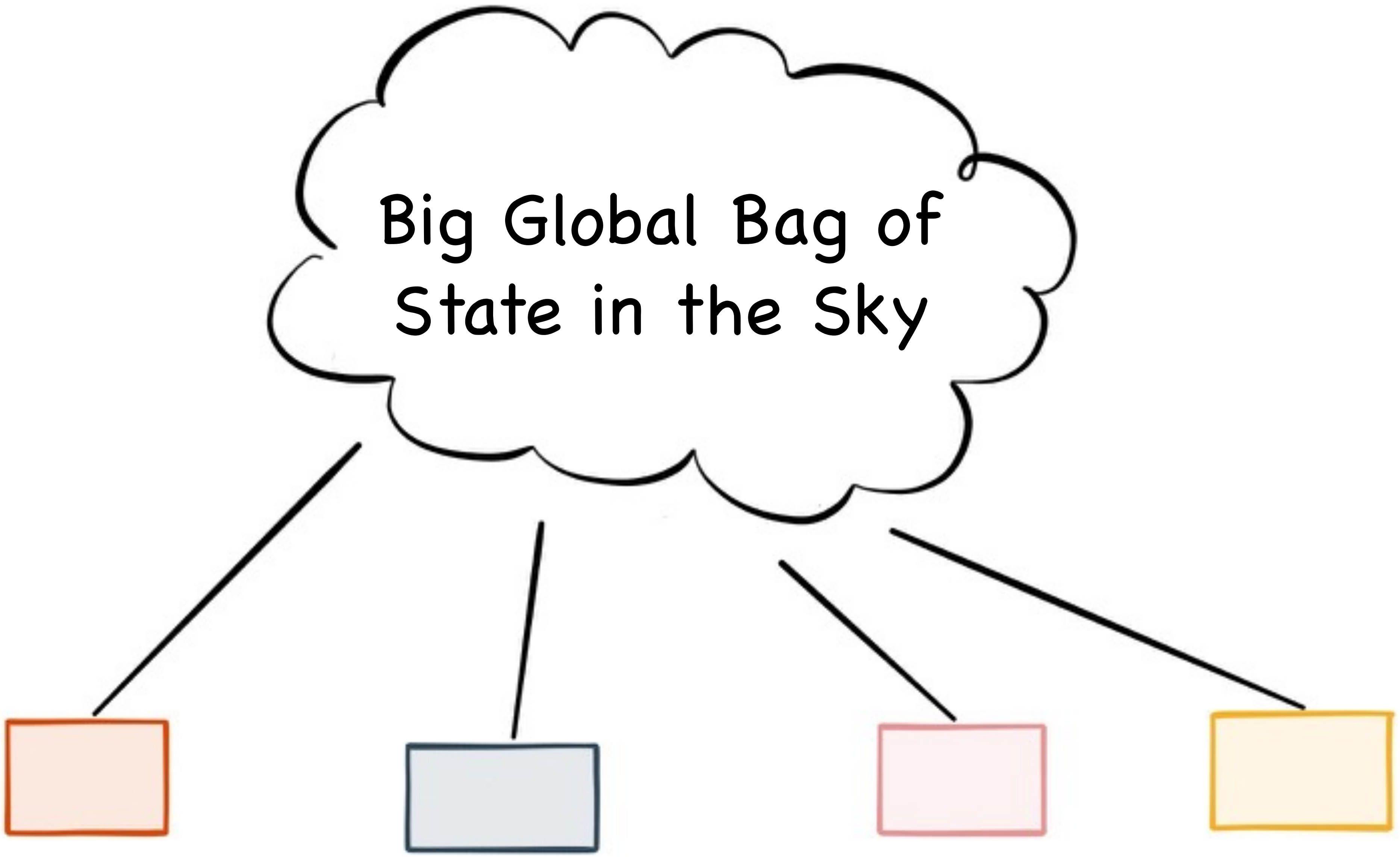
with strong ordering and
repeatability guarantees
(eventually)

SO…

Microservices push us away from shared, mutable state

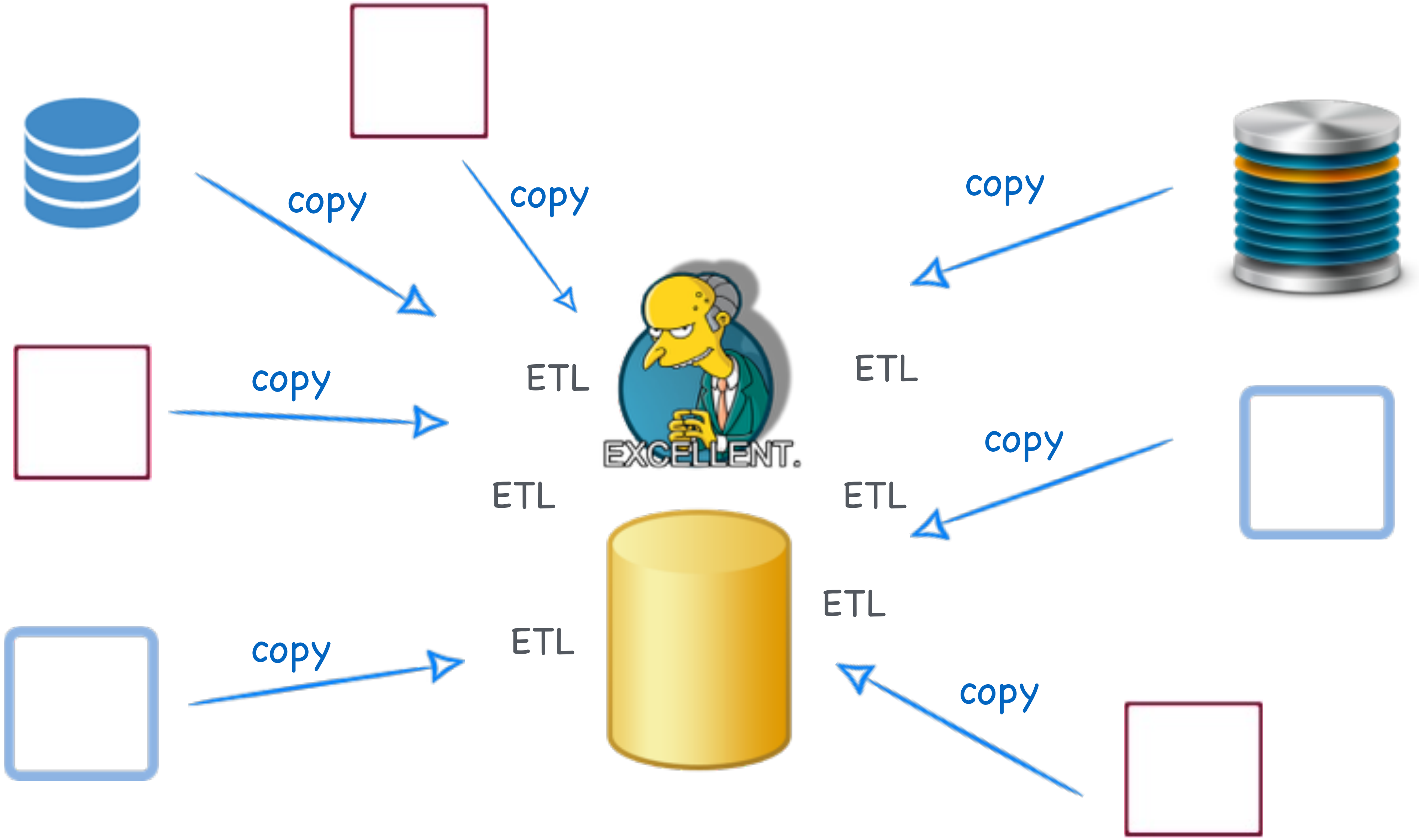


Away from BGBSS's



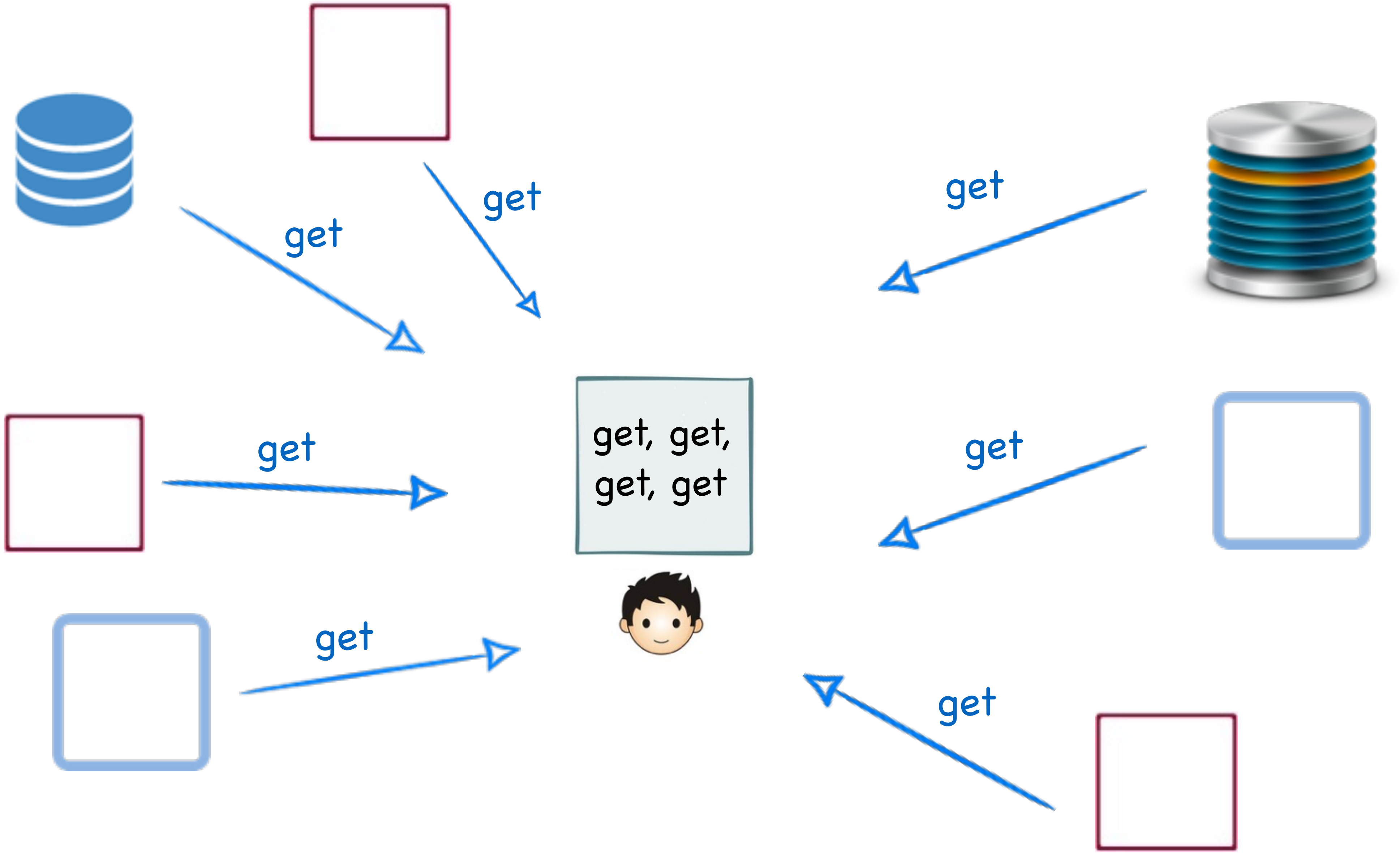
This means data is increasingly remote

Sure, you can collect it all



can be a lot of work

Or you can look it all up

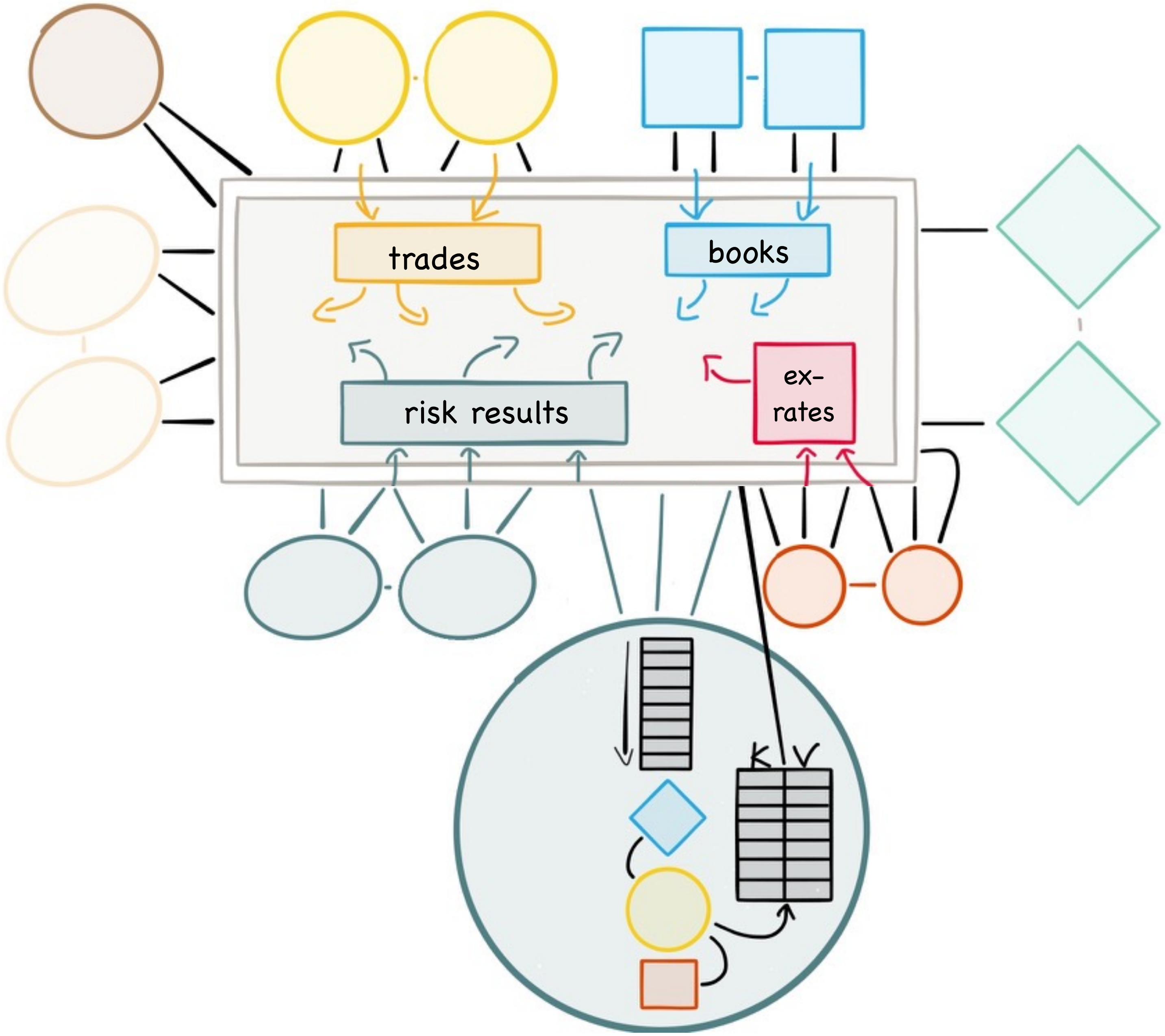


but that doesn’t scale well

(with system complexity or with data throughput)

Better to embrace decentralistion

We need a decentralised toolset to do this



Keep it simple,
Keep it moving