Putting the ‘Micro’ into Microservices with Stateful Stream Processing

Ben Stopford

@benstopford

This Is Me

- Engineer at Confluent
- Ex Thoughtworks + UK Finance

Last time: Event Driven Services

- Interact through events, don't talk to services
- Evolve a Cascading Narrative
- Query views, built inside your bounded context.

Related:

-Event Collaboration

-Service Choreography

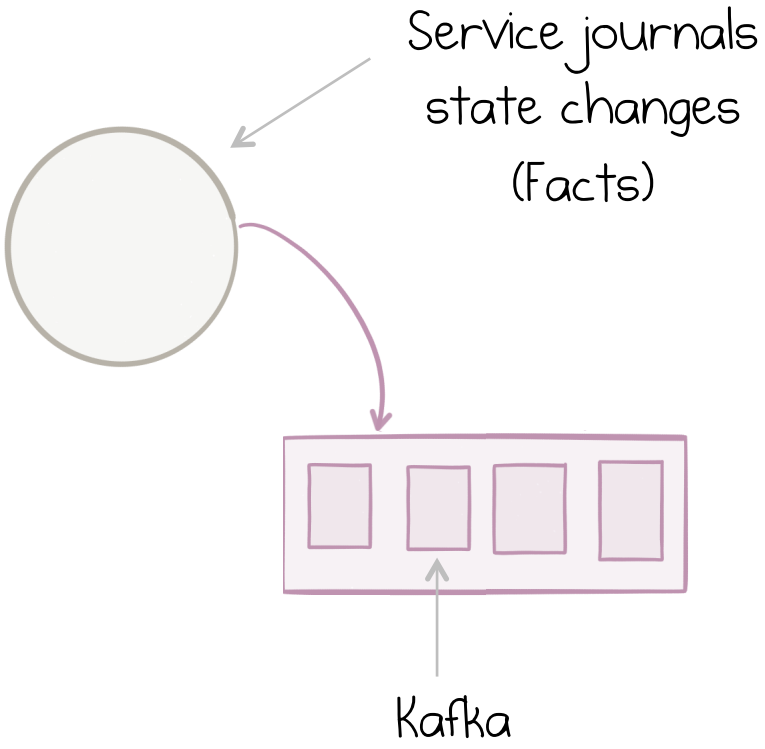
-Event storming

The narrative should contain business facts

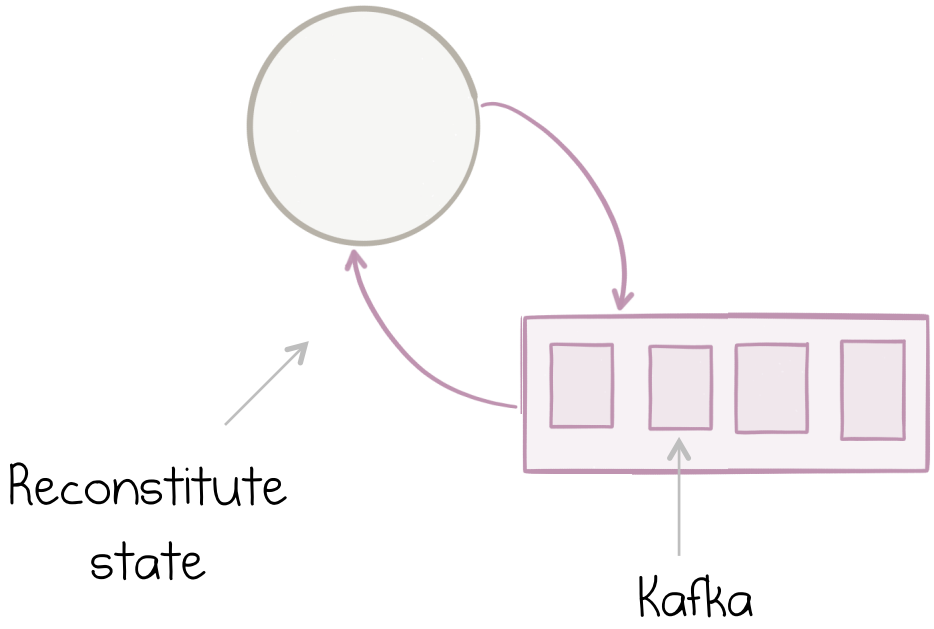
It should read like:

"a day in the life of your business"

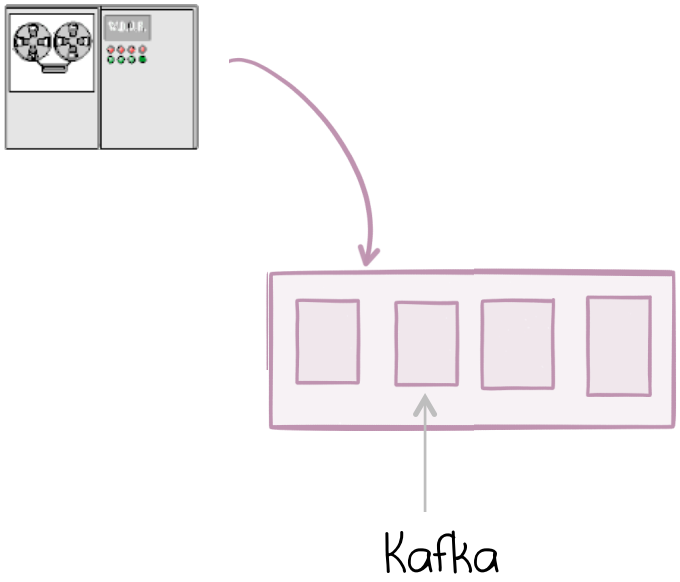
Base services on Event Sourcing



Where stateful, reconstitute from log



Use the narrative to move away from legacy apps



Retain the Narrative

Kafka scales

What is this talk about?

Using SSP to build lightweight services that evolve the shared narrative

1. What's in the SSP toolbox?
2. Patterns for Streaming Services
3. Sewing together the Big & the Small
4. The tricky parts

APPLYING TOOLS FROM STATEFUL STREAM PROCESSING

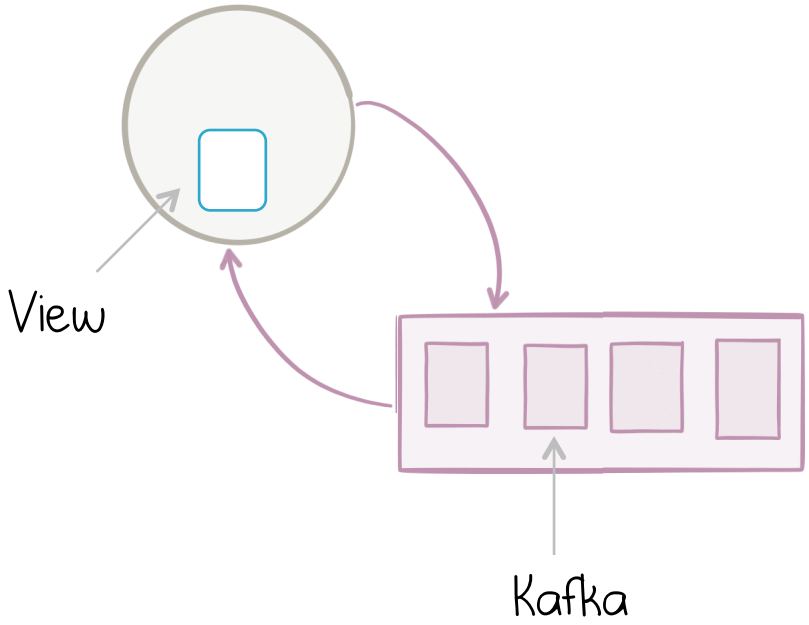
Aside: What is Stateful Stream Processing?

Output a **stream**

or output a **table**

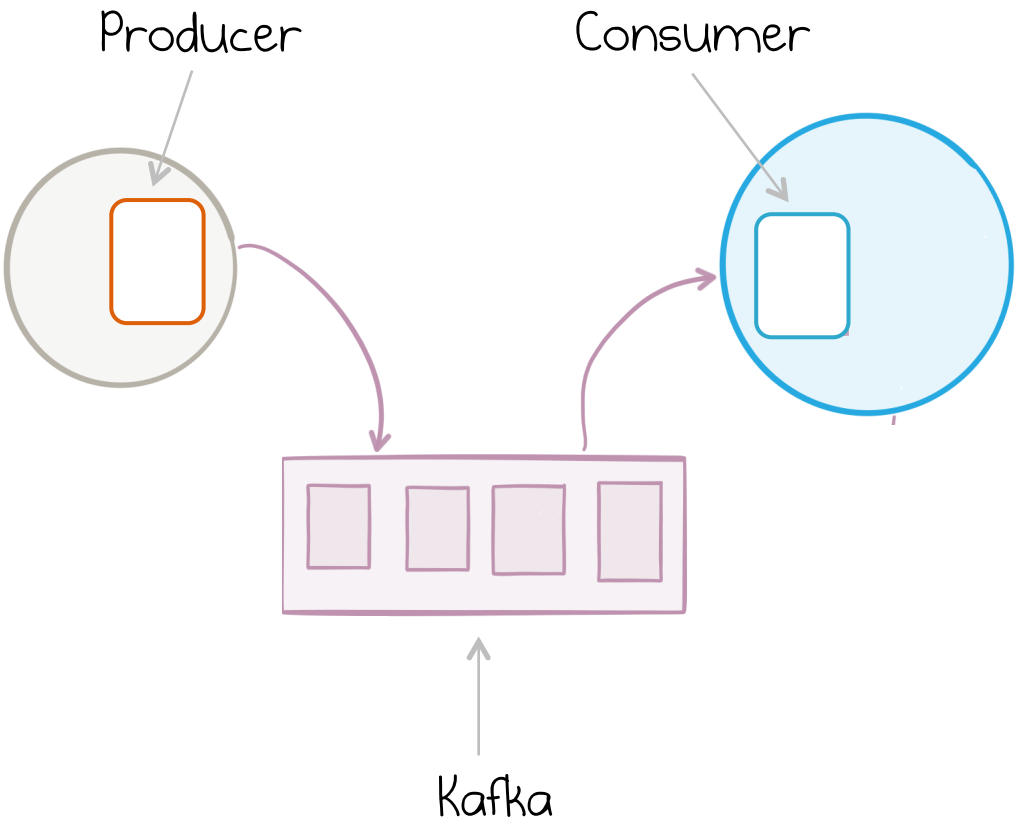
A machine for converting a set of streams into something more useful.

(1) Drive processing from Events
(2) Build views that answer questions



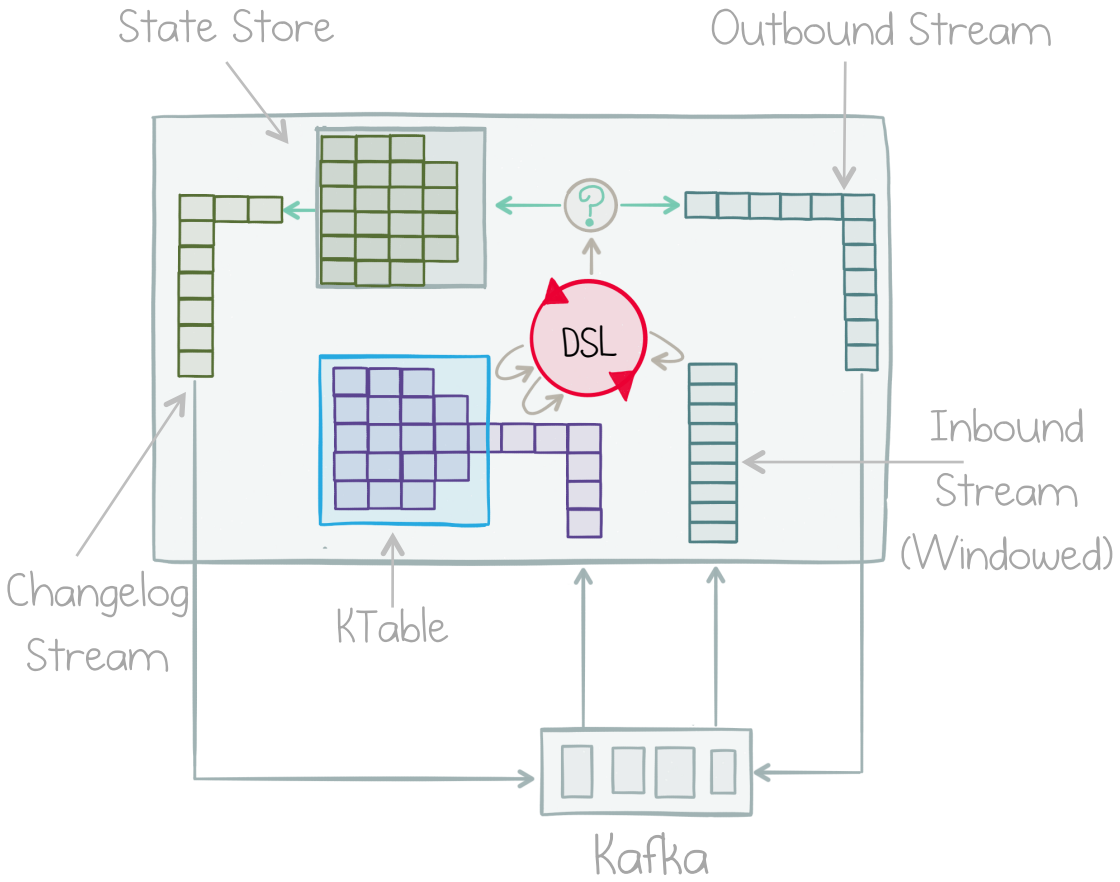
THE TOOLS

(1) Producer/Consumer APIs



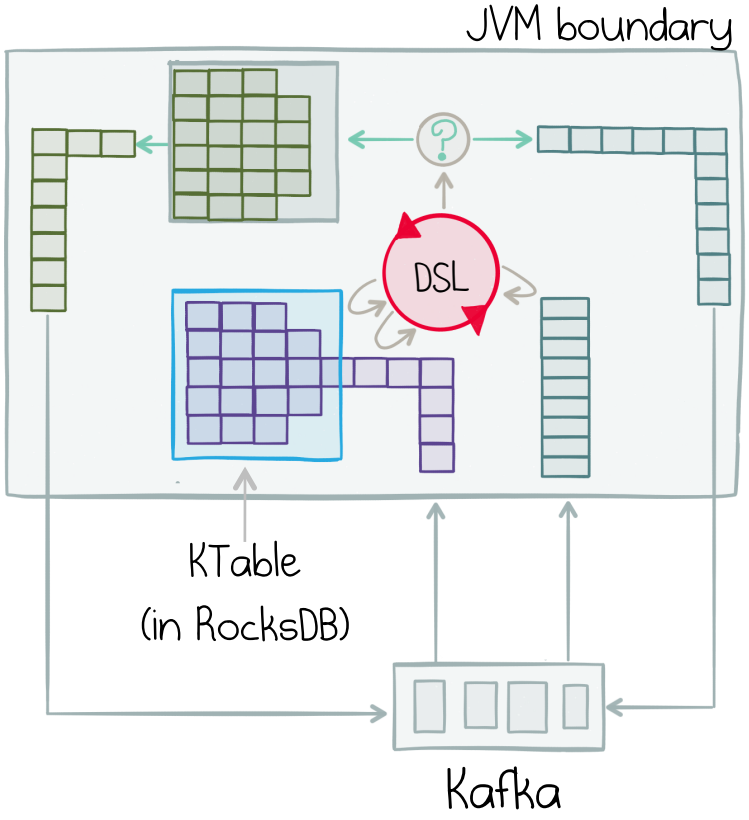
- Simple & easy to understand
- Native in many languages
- Easy to integrate to existing services
- No joins, predicates, groupings or tables
- Includes transactions in Kafka-11.

(2) KStreams DSL



- Joins (including windowing)
- Predicates
- Groupings
- Processor API

(3) The KTable

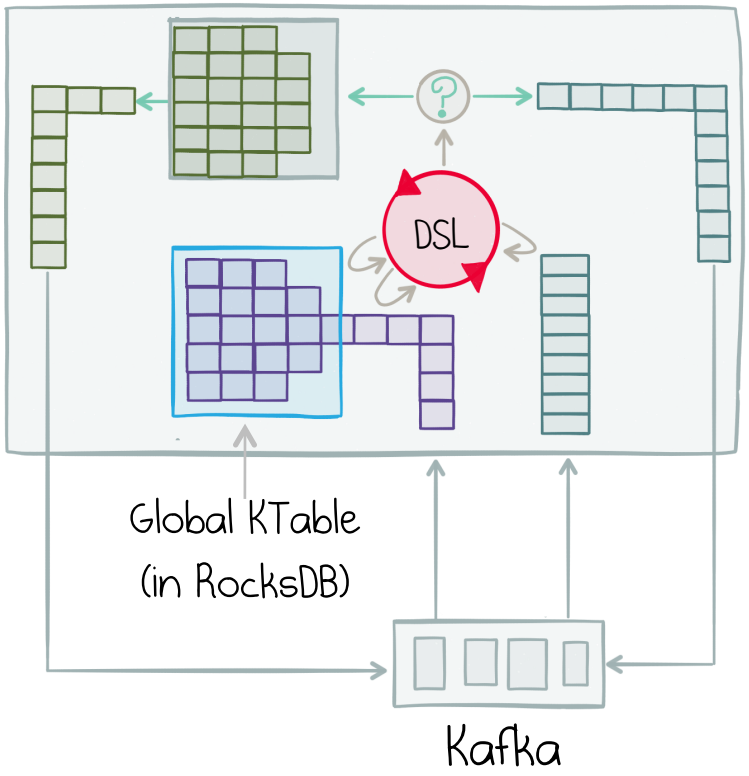


- Recast a stream as a table via a defined key
- Tends to be backed by compacted topic.
- Materialized locally in RocksDB
- Provides guaranteed join semantics (i.e. no windowing concerns)

The KTable

- Behaves like a regular stream but retains the latest value for each key, from offset 0
  - Partitions spread over all available instances
  - Partition counts must match for joins
  - Triggers processing
  - Initializes lazily.

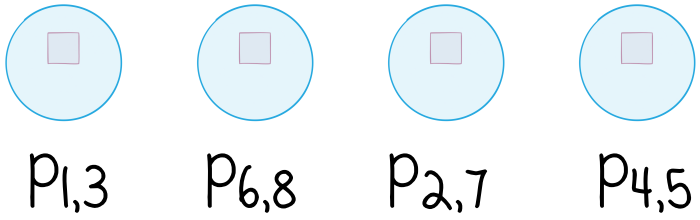
(4) Global KTable



- All partitions replicated to all nodes
- Supports N-way join
- Blocks until initialized
- Doesn't trigger processing (reference resource)

Global-KTables vs. KTables

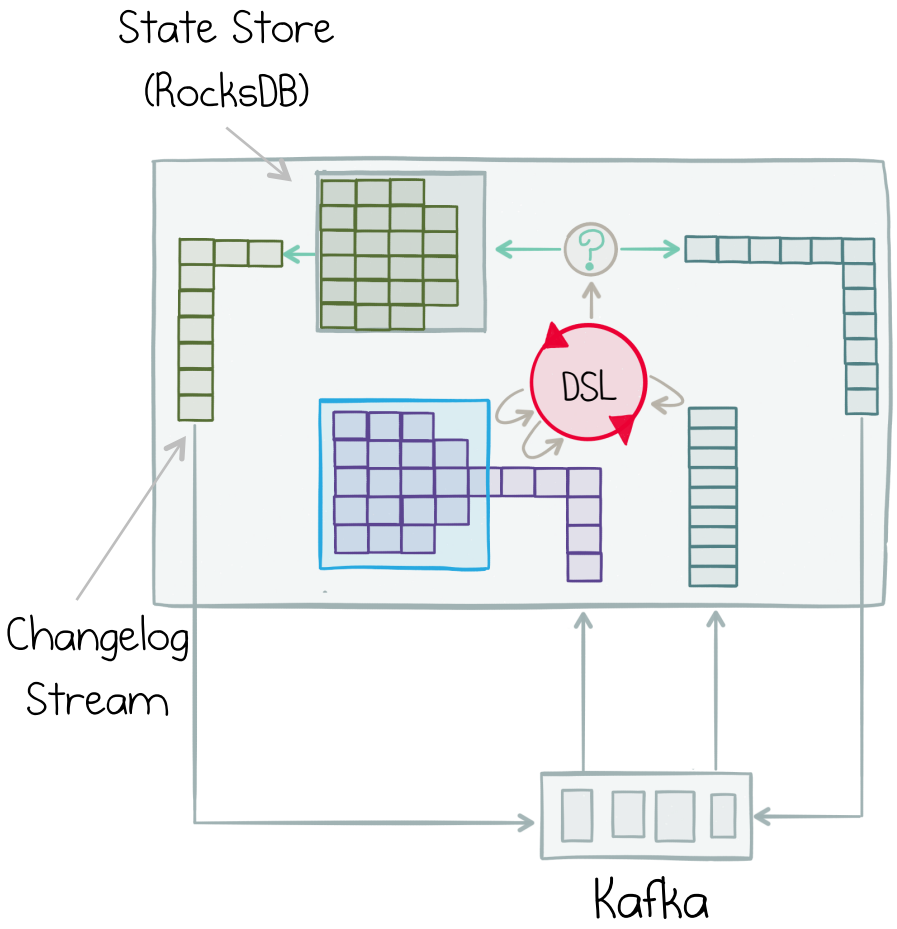
Given a service with four instances:



KTable
(Sharded)

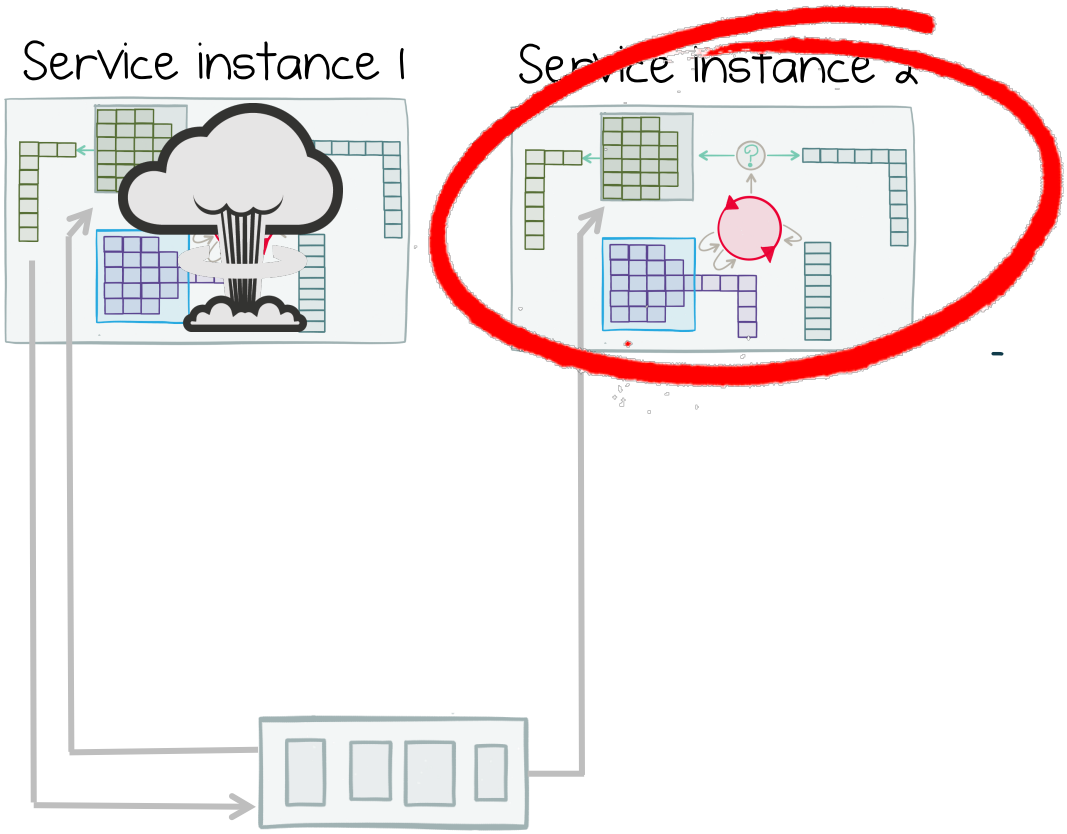
Global KTable
(Replicated)

(5) State Store



- RocksDB instance
- **Mutable**
- Backed by Kafka
- Integrated into DSL
- Can be instantiated independently
- Queryable via Queryable state
- Useful for custom views or stateful services.

(6) Hot Replicas



By default KStreams keeps state backed up on a secondary node.

Single machine failure doesn't result in a re-hydration pause.

=> Highly available, stateful services.

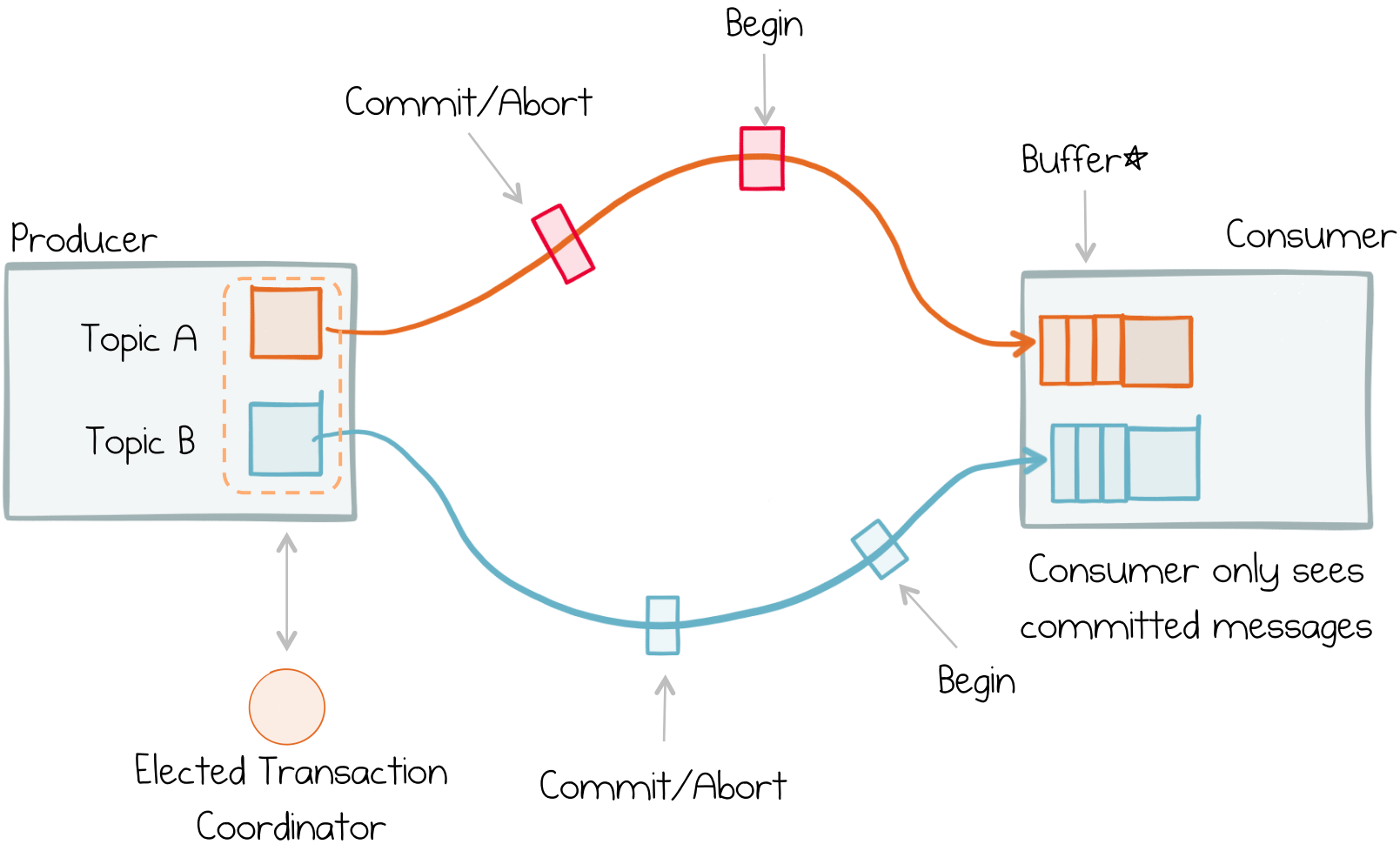
(7) Transactions

Two problems:

1. Failures create duplicates
   - Each message gets a unique ID
   - Deduplicate on write
2. You need to write, atomically, to 2 or more topics
   - Chandy-Lamport Algorithm

NB: Transactions will be in the next Kafka release

Snapshot Marker Model

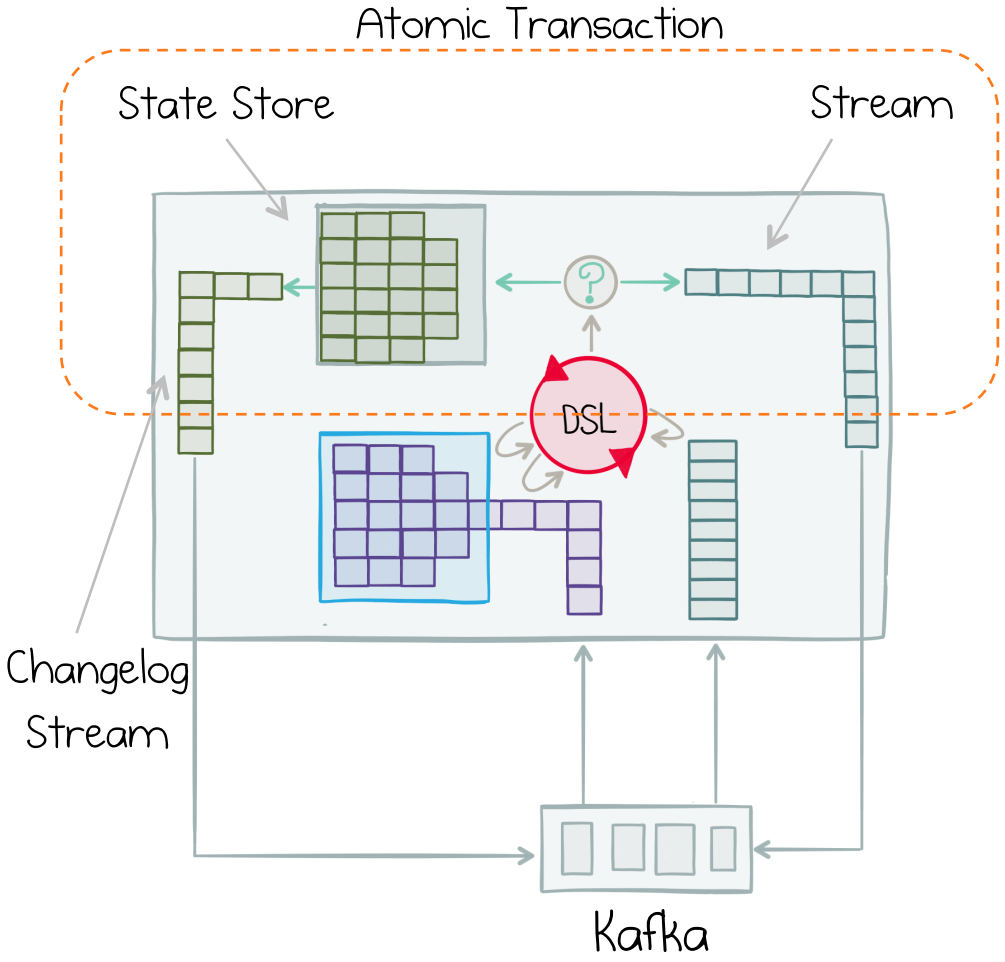


* The buffering is actually done in Kafka no the consumer

Simple Example

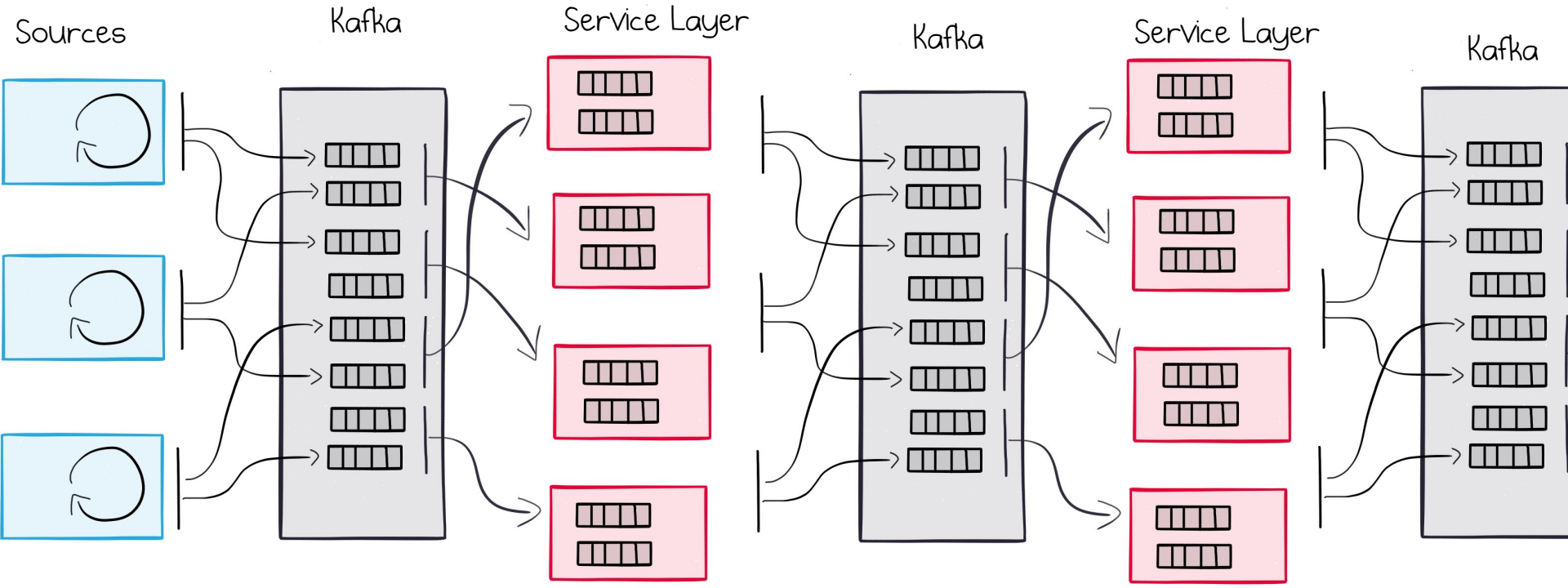
```
//Orders -> Stock Requests and Payments
orders = ordersTopic.poll()
payments = paymentsFor(orders)
stockUpdates = stockUpdatesFor(orders)

//Send & commit atomically
producer.beginTransaction()

producer.send(payments)
producer.send(stockUpdates)
producer.sendOffsetsToTransaction(offsets(orders))

producer.endTransaction()
```

State-Stores & Transactions



- In KStreams Transactions span stream and changelog
- Processing of stream & state is atomic
- Example: counting orders.

All these tools scale horizontally and avoid single points of failure



SERVICE PATTERNS

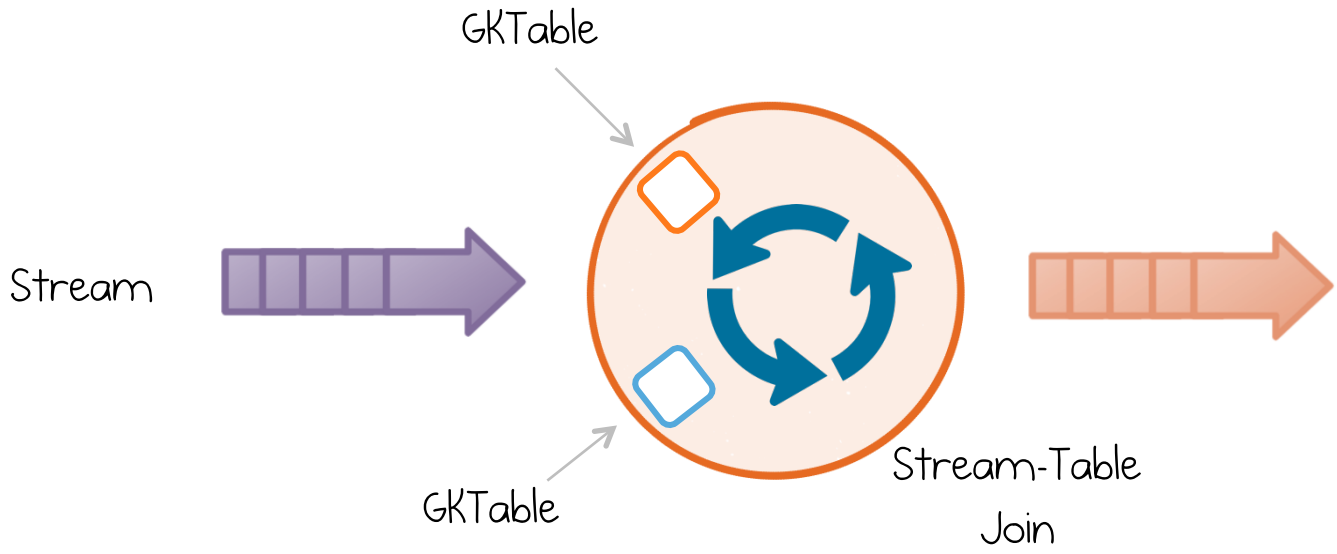
(building blocks)

(1) Stateless Processor

e.g.

- Filter orders by region
- Convert to local domain model
- Simple rule

(2) Stateless, Data Enabled Processor

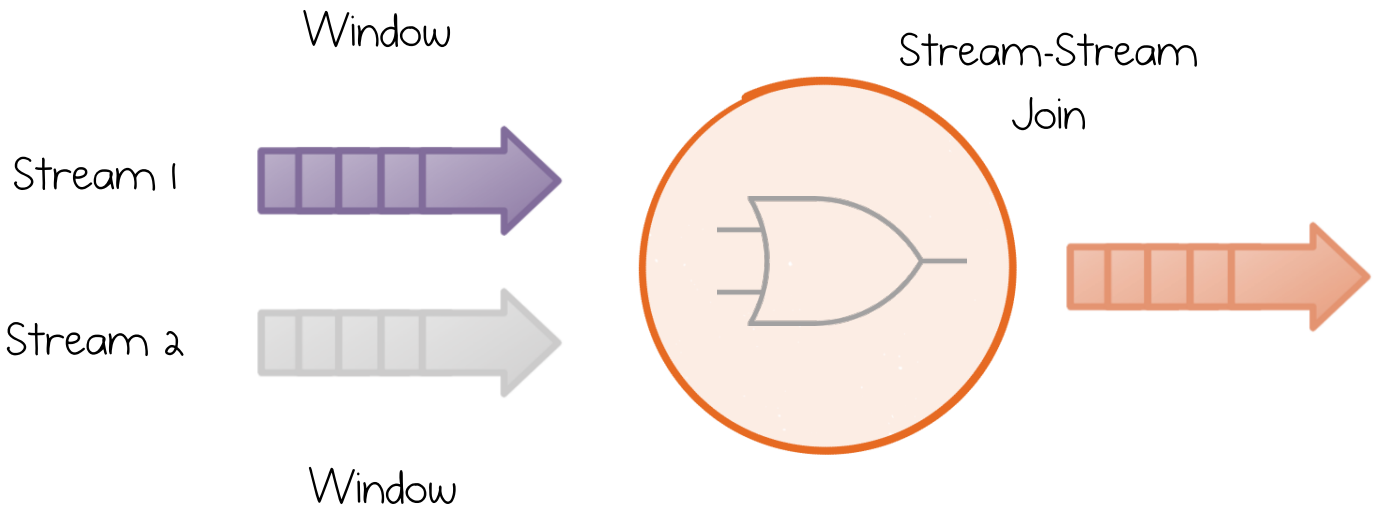


Similar to star schema

- Facts are stream
- Dimensions are GKTables

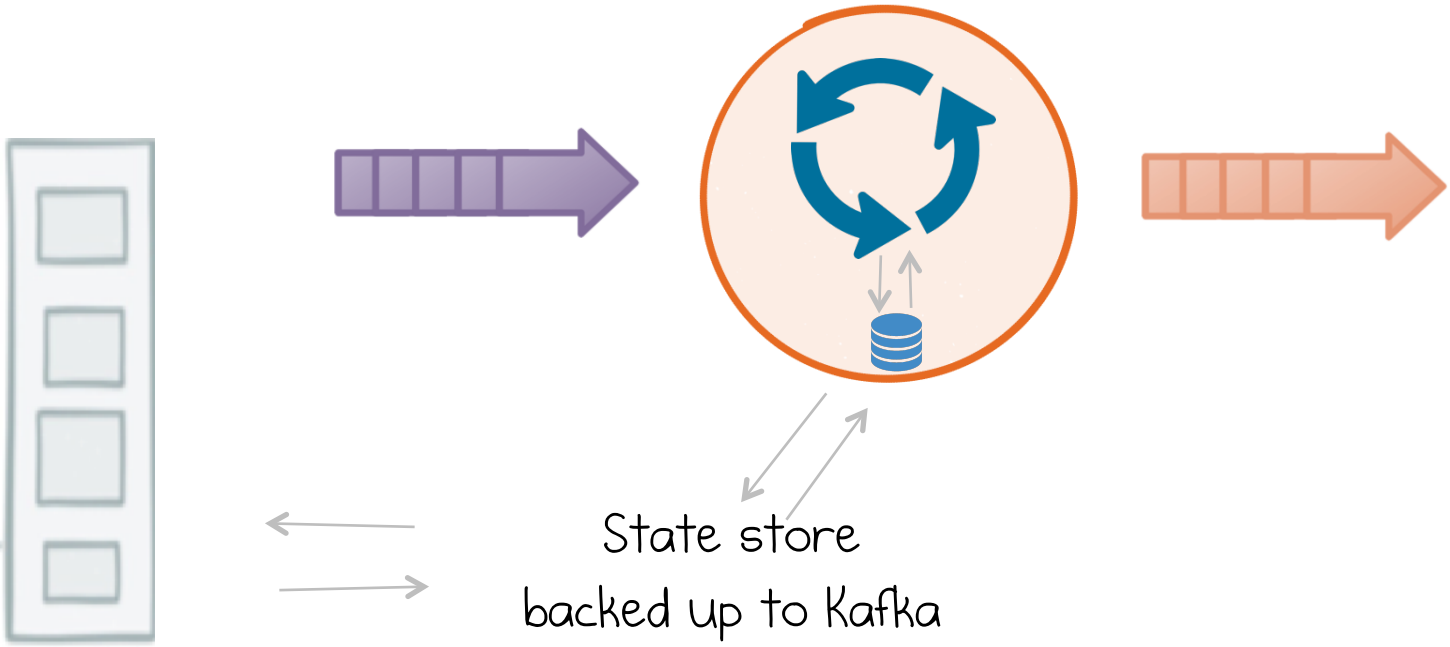
e.g. Enrich Orders with Customer Info & Account details

(3) Gates



e.g. rules engines or event correlation
When Order & Payment then ...

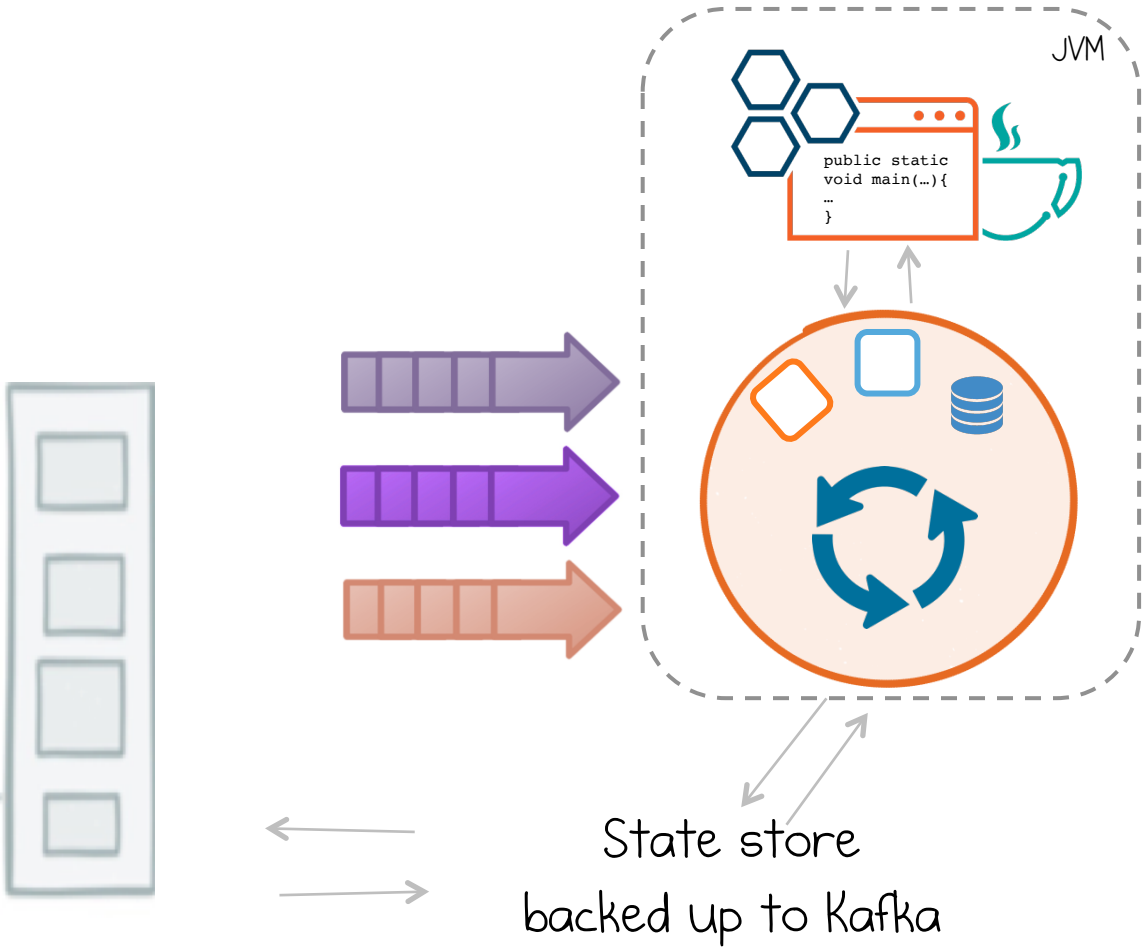
(4) Stateful Processor



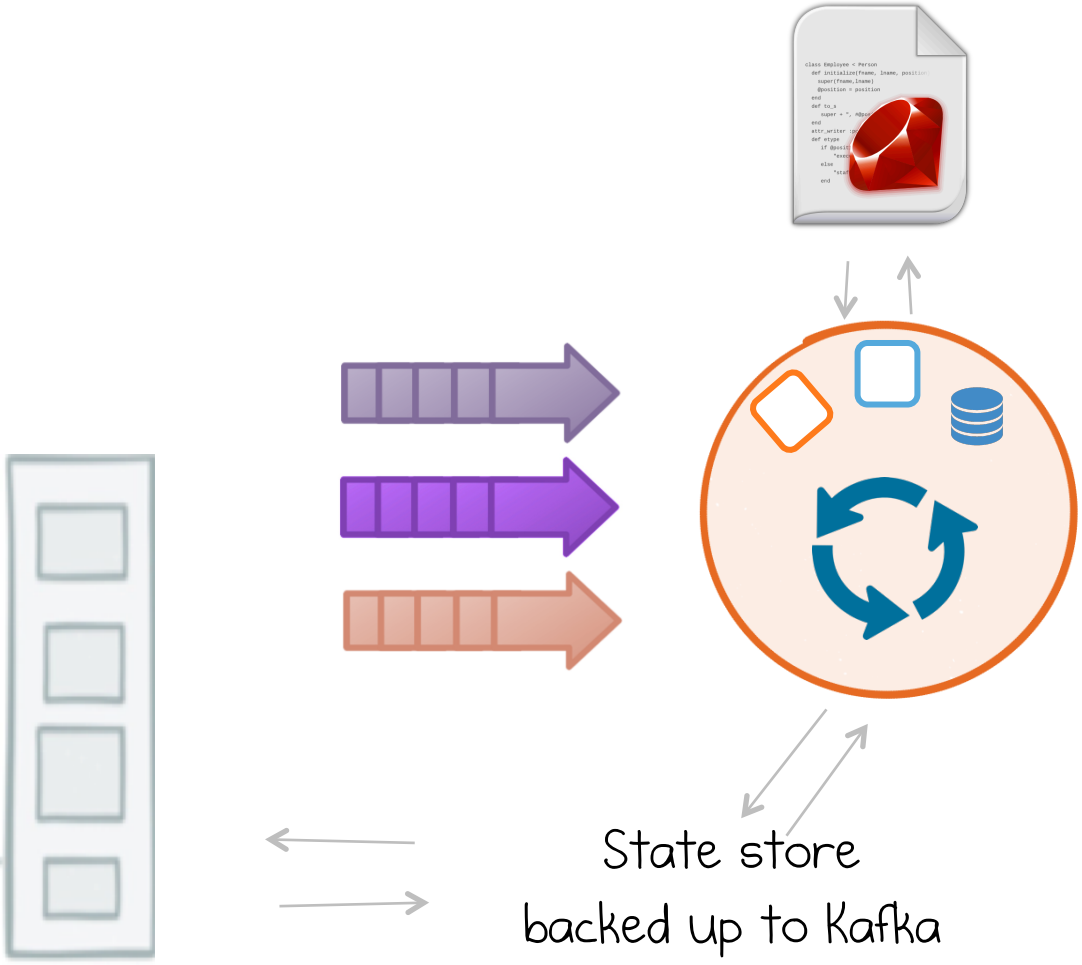
e.g.

- Average order amount by region
- Posting Engine (reverse replace of previous position)

(5) Stream-Aside Pattern

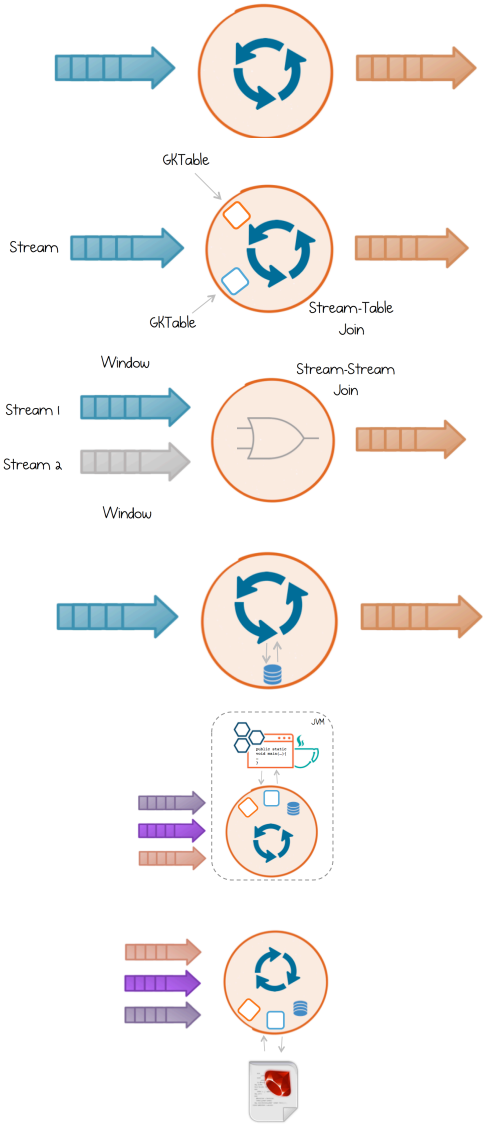


(7) Sidecar Pattern

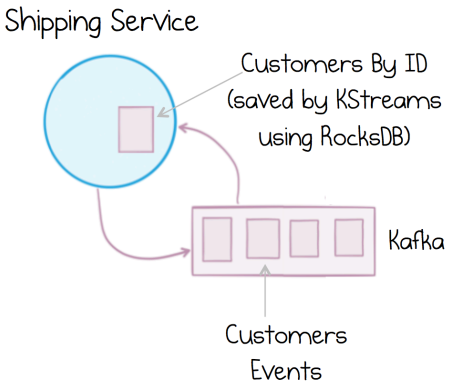


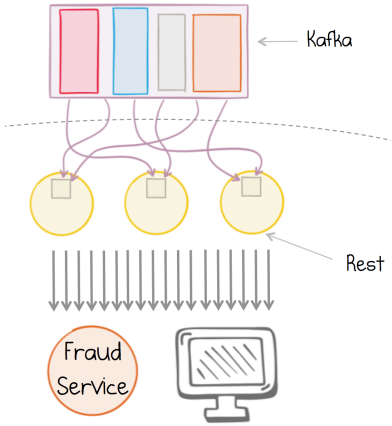
Combine them:

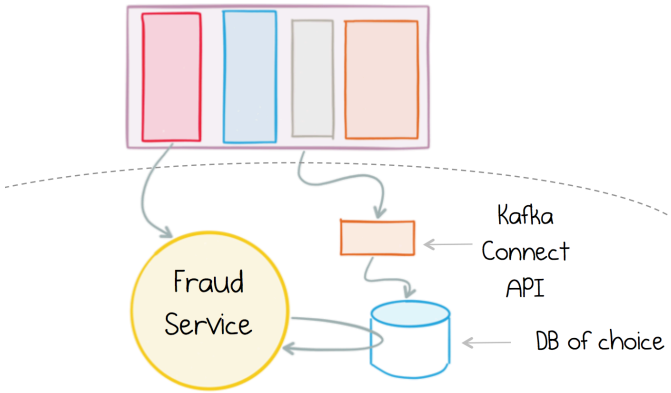
1. Stateless
2. Data enriched
3. Gates
4. Stateful processors
5. Stream-aside
6. Sidecar

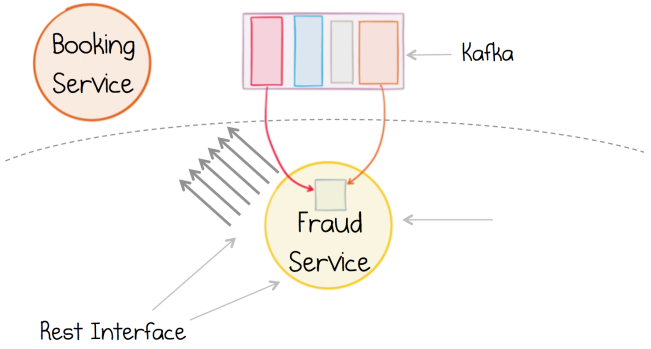


Query patterns covered in last talk









BUILDING ON THE NARRATIVE

THE BIG AND THE SMALL

Evolutionary approach

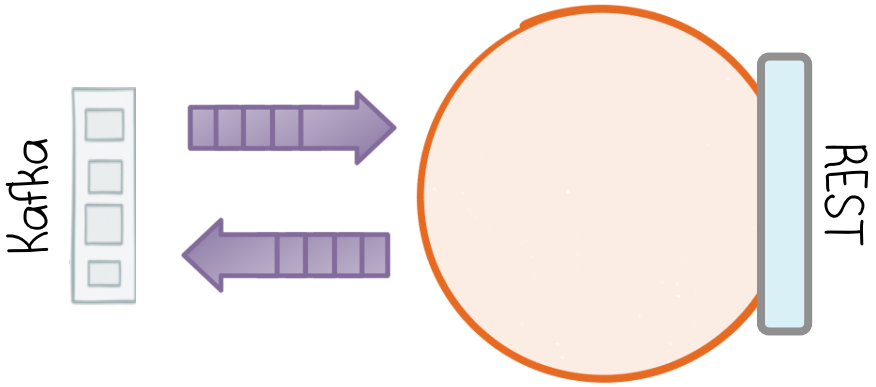
Services should:

- Fit in your head
- Should be quick to stand up & get going

But *sometimes* bigger is the right answer

Event Driven First

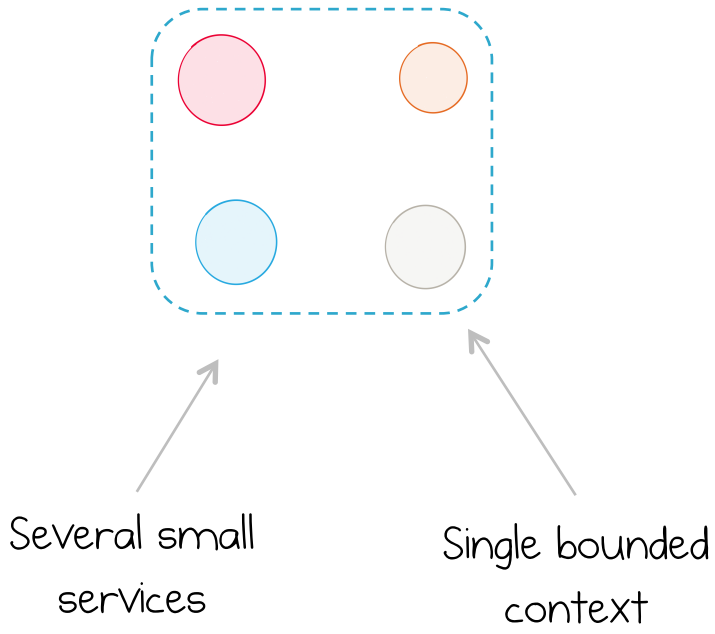
- Build on an event driven backbone
- Add Request-Driven interfaces where needed.
  - Prefer intra-context



Putting the Micro into Microservices

- Combine simple services that develop the shared narrative
- Start stateless, make stateful if necessary

Contexts within Contexts

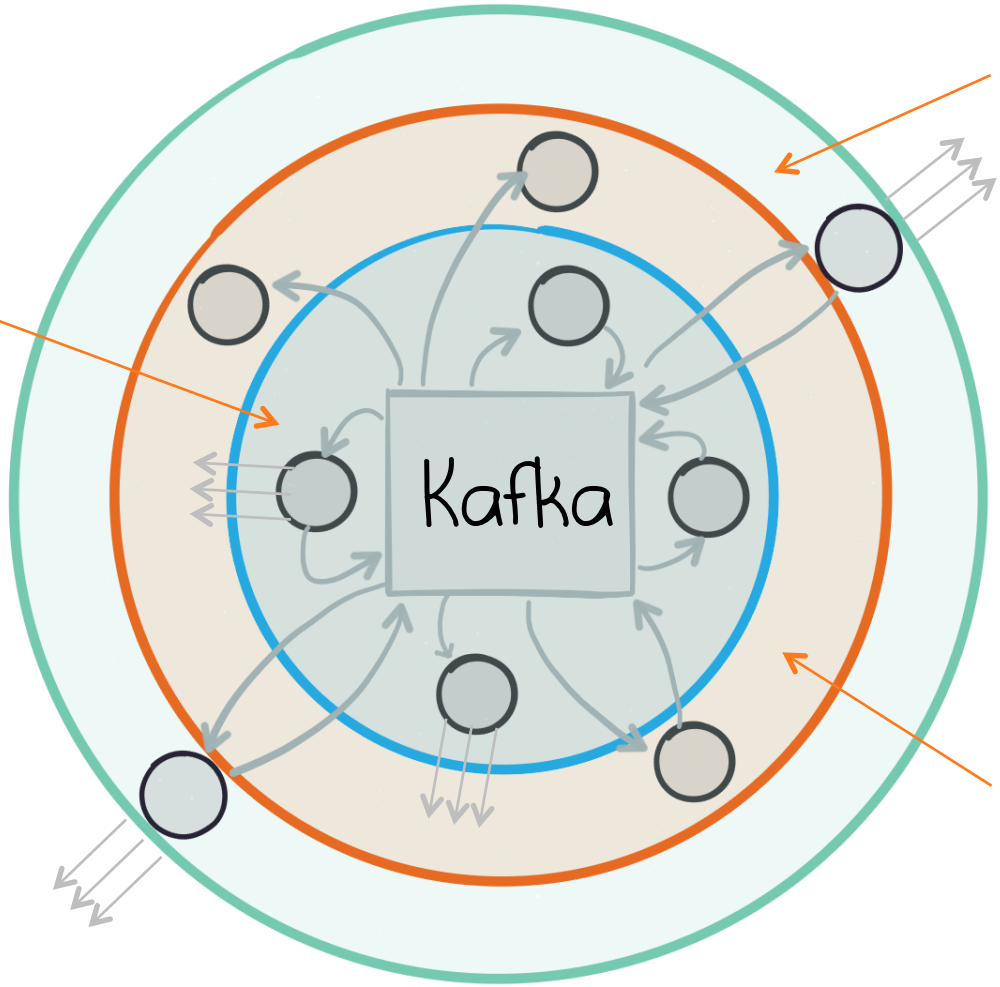


Where services are tiny, they will typically be grouped together.

- Shared domain?
- Shared deployment?
- Shared streams?

Layering within multi-service contexts

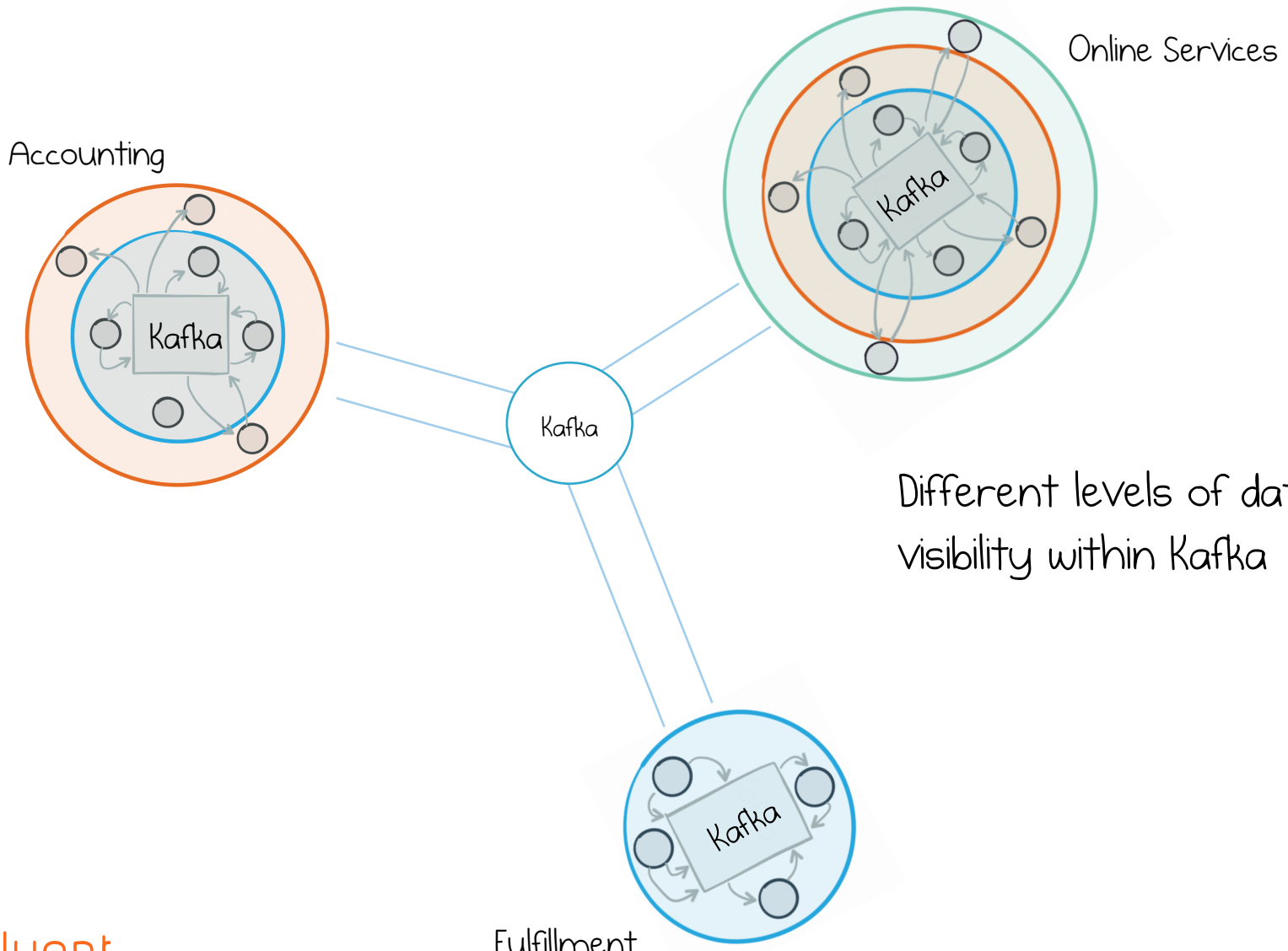
Data layer:
-Stream enrichment
-View creation
-Streams maintenance



Request-Driven layer provides REST / RPC / websotcket etc for UIs etc

Business layer encapsulates logic, integrates with legacy code etc.

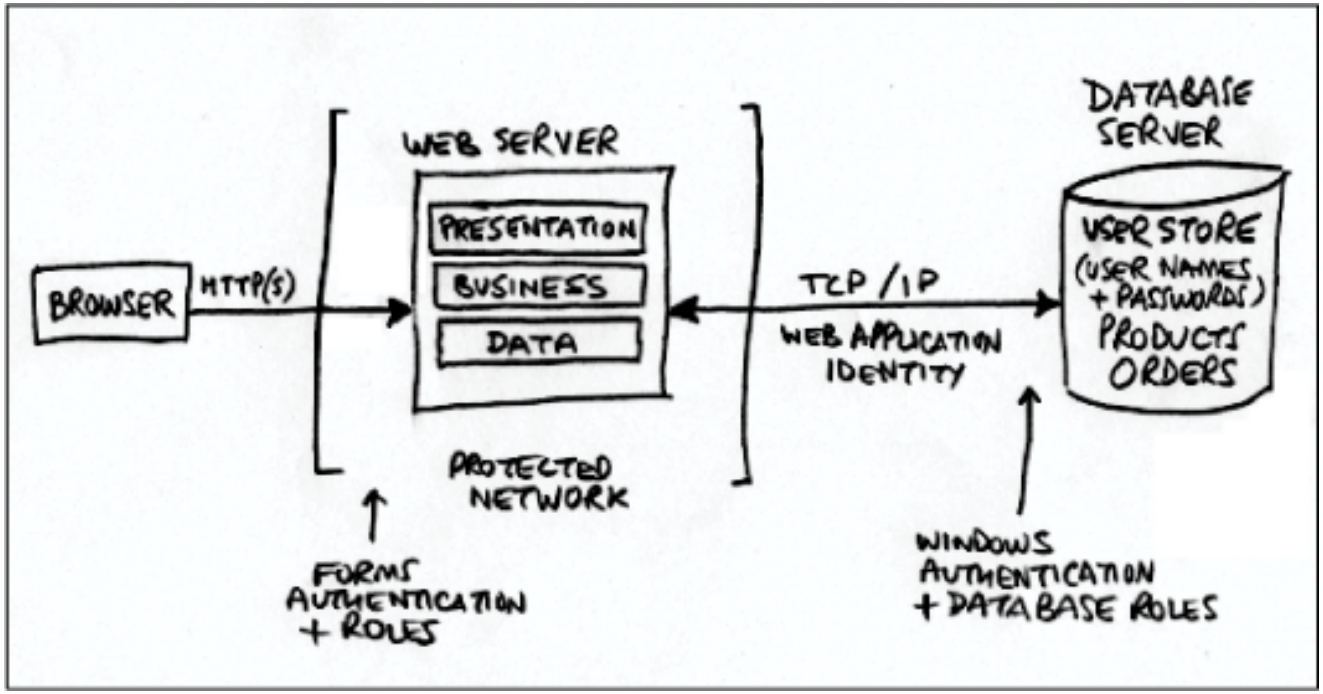
Contexts within Contexts



Different levels of data visibility within Kafka

BRINGING IT TOGETHER

Good architectures have little to do with this:

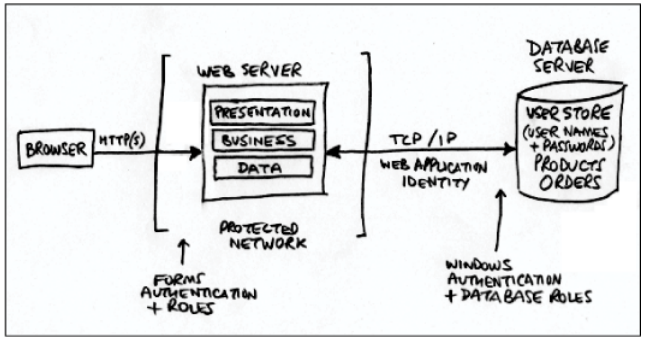


It's more about how systems evolves over time

Sunk Cost vs Future Cost

Balance

Architecture: Sunk Cost



Change: Future Cost

Part 1: Data dichotomy

Request-Driven Services don't handle shared data well

- God service problem
- The data erosion problem

Request-Driven Services struggle with complex flows

- the service entanglement problem
- the batch processing problem.

Part 2: Building Event Driven Services

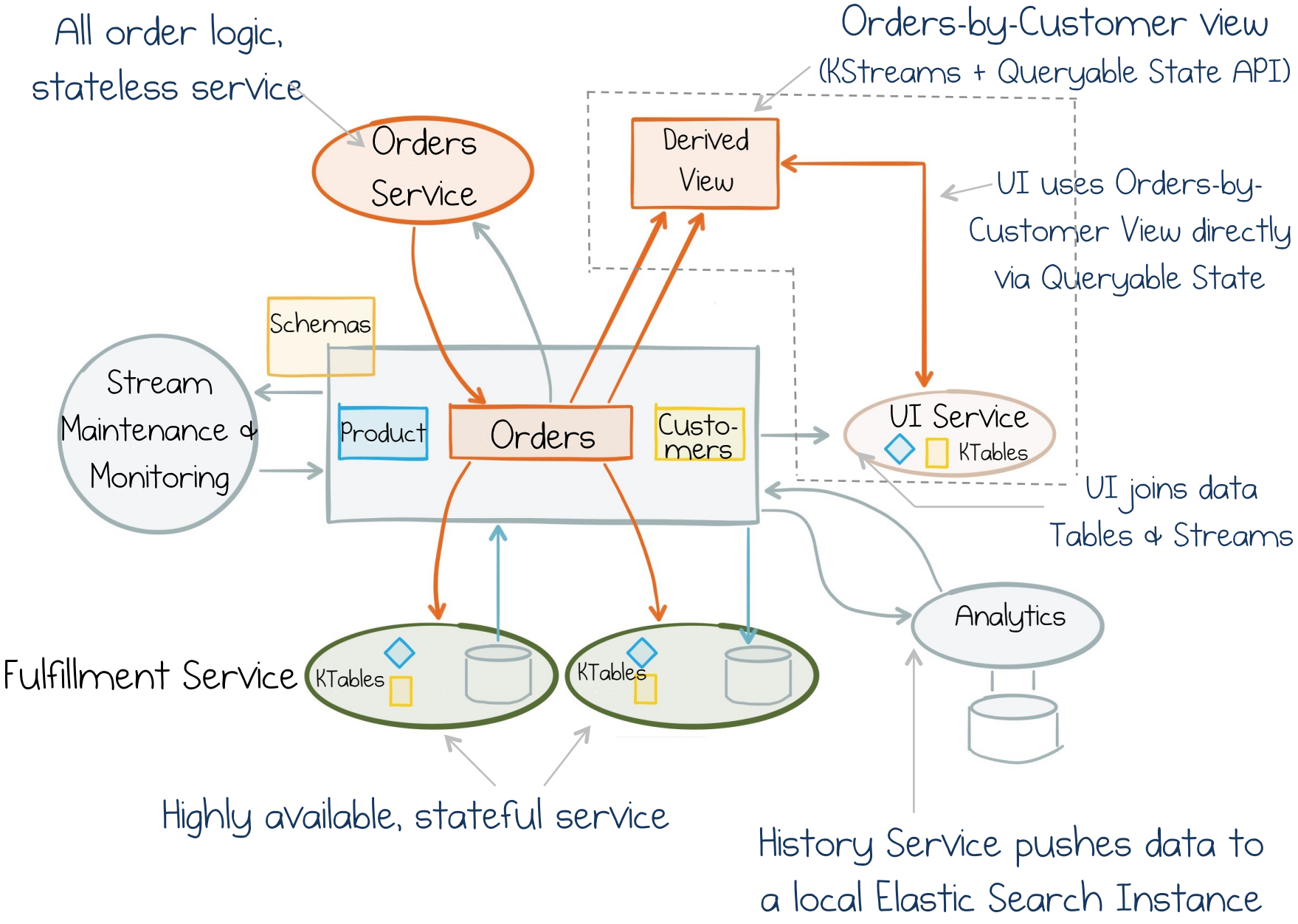
Solution is to part 1:

- Event source **facts** into a Cascading Narrative
- Communication & State transfer **converge**
- **Decentralize** the role of turning facts into queryable views
- Keep these views as lightweight and **disposable** as possible.

Putting the Micro in Microservices

- Start lightweight, stateless functions which compose and evolve
- Leverage the stateful stream processing toolkit to blend streams, state, views, transactions.
- Use layered contexts that segregate groups of streams **and** services.

HIGHLY SCALABLE ARCHITECTURE


All order logic,
stateless service
Orders-by-Customer view
(KStreams + Queryable State API)
Orders
Service
Derived
View
UI uses Orders-by-
Customer View directly
via Queryable State
Schemas
Stream
Maintenance &
Monitoring
Product
Orders
Custo-
mers
UI Service
KTables
UI joins data
Tables & Streams
Analytics
Fulfillment Service
KTables
KTables
Highly available, stateful service
History Service pushes data to
a local Elastic Search Instance

Tips & Tricky parts

- Reasoning about service choreography
  - Debugging multi-stage topologies
  - Monitoring/debug via the narrative
  - Segregate facts from intermediary streams
- Retaining data comes at a cost
  - Schema migration etc
- Accuracy over non trivial topologies
  - Often requires transactions
- Transactions, causality & the flow of events
  - Avoid side effects

Event Driven Services

Create a cascading narrative of events, then sew them together with stream processing.


kafka summit
Discount code: kafcom17
Use the Apache Kafka community discount code to get $50 off
www.kafka-summit.org
Kafka Summit New York: May 8
Kafka Summit San Francisco: August 28
Presented by confluent

Thank You!

@benstopford