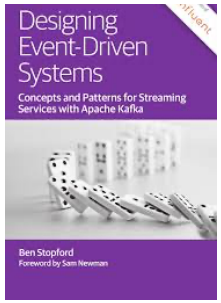
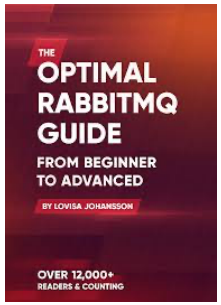
Event-driven systems

UA.DETI.IES - 2019/20

Resources & Credits



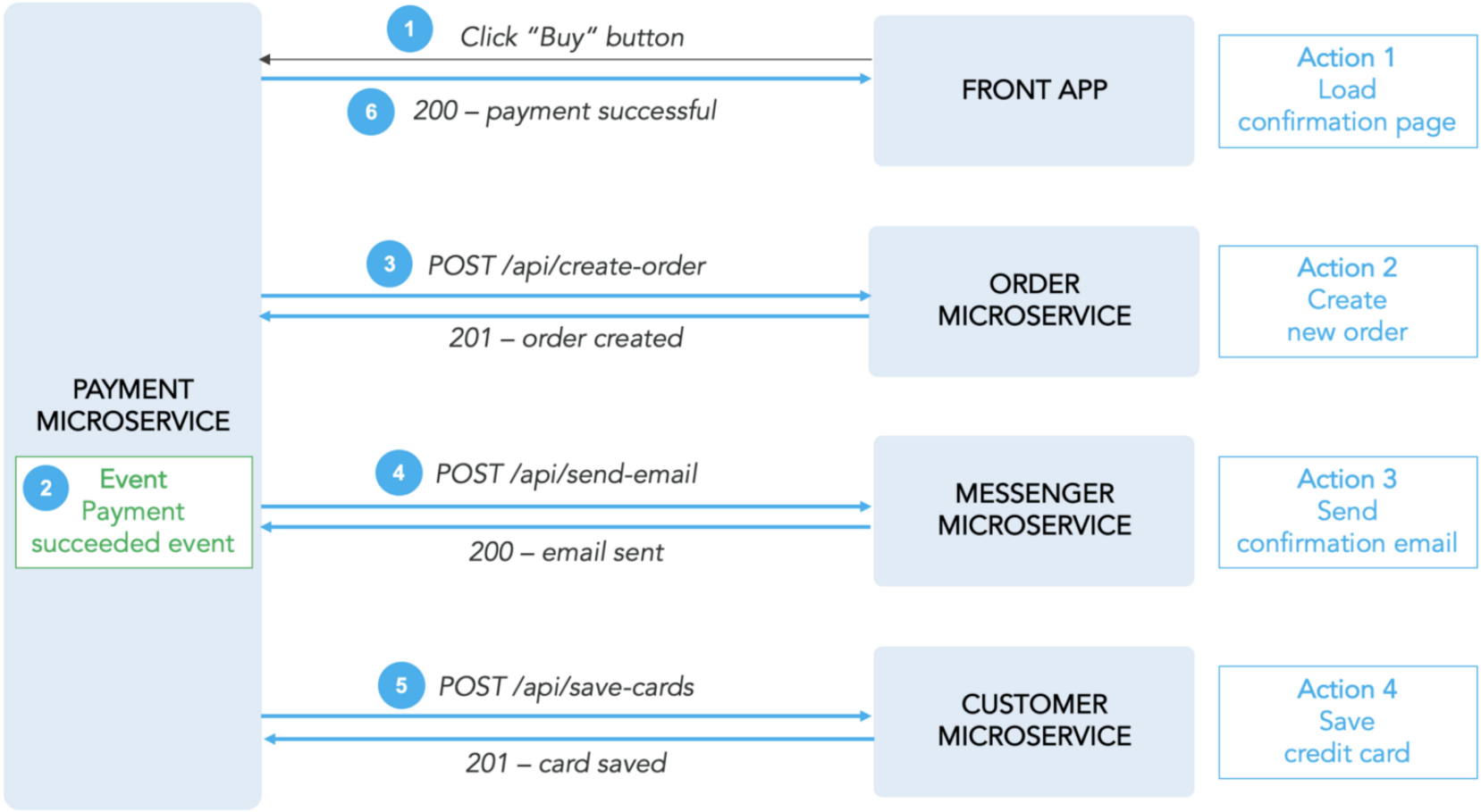


- Designing Event-Driven Systems
  Ben Stopford, O'Reilly
- The Optimal RabbitMQ Guide, Lovisa Johansson, CloudAMQP
- Spring Cloud Data Flow
  - https://dataflow.spring.io

Scaling with Microservices

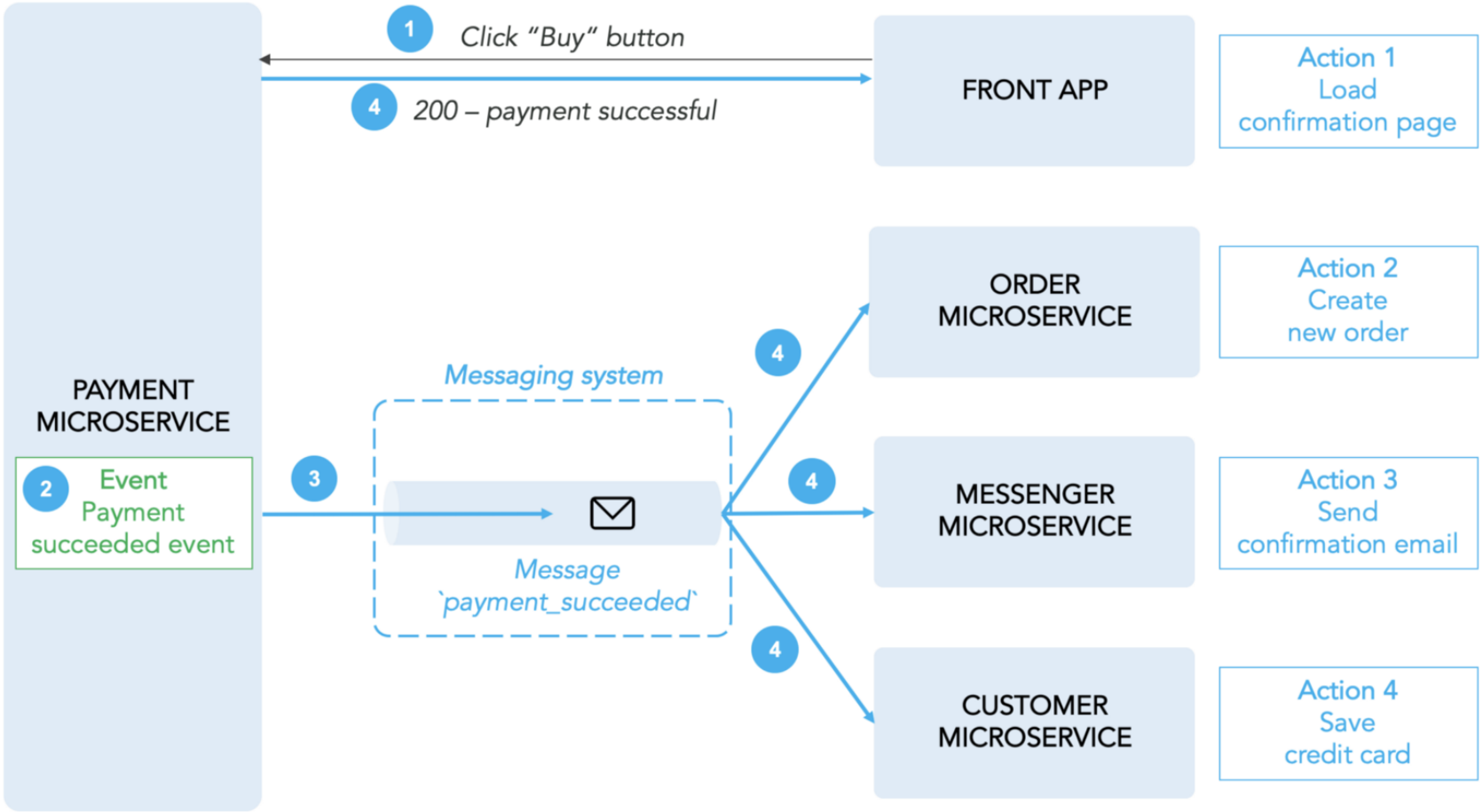
- Application development is currently based on concepts like microservices and serverless functions.
  - We now live in a cloud-native world, and these models are a natural fit for distributed, cloud-based environments.
  - But simply building and deploying services and functions isn't enough.
- On its own, a single microservice doesn't accomplish much.
  - We also need a way to wire up those components
    - i.e., to connect them so they can exchange data, forming a true application.
  - In SE, this is one of the most important architectural decisions to make.

Request-driven architecture



https://blog.theodo.com/2019/08/event-driven-architectures-rabbitmq/

Event-driven architecture



Event-driven architecture

- For controlling these services, various mechanisms were developed over the years, such as message queues and enterprise service buses (ESBs).
  - E.g. RabbitMQ, WSO2.
- More recent offerings, the concept of streaming data have also emerged.
  - E.g. Apache Kafka
- This latter category is growing, in part because streaming data is seen as a useful tool for implementing event-driven architecture
  - a software design pattern in which application data is modeled as streams of events, rather than as operations on static records.

What is an event?

- **Events** are things that happen, within a software system or, more broadly, during the operation of a business or other human process.
  - e.g., a sensor reports a temperature change, a user clicks their mouse, a customer deposits a check into a bank account.
- The concept of events in software systems closely aligns with how most of us think about our day-to-day lives.
  - Organizing around events makes it easier to develop business logic that accurately models real-world processes.
  - It helps reducing the number of one-to-one connections within a distributed system increasing the value of the microservices.
- An **event-driven architecture** allows generating, storing, accessing and reacting to these events.

Events vs. queries and commands

- **Queries** are a request to look something up
  - Unlike events or commands, queries are free of side effects; they leave the state of the system unchanged.
- **Commands** are actions
  - Requests for some operation to be performed and will change the state of the system.
  - Synchronous and typically indicate completion.

	Behavior/state change	**Includes a response**
Command	Requested to happen	Maybe
Event	Just happened	Never
Query	None	Always

Event-driven patterns – notification

- **Event notification**
- A service sends events to notify other systems of a change in its domain.
  - For example, a user account service might send a notification event when a new login is created.
- What other systems choose to do with that information is largely up to them.
  - The service that issued the notification just carries on with its business.
- Notification events usually don't carry much data.
  - Resulting in a loosely coupled system with minimal network traffic spent on messaging.

Event-driven patterns – state transfer

- **Event-carried state transfer**
- A step up from simple notification, in this model the recipient of an event also receives the data it needs to perform further work.
  - E.g., the user account service might issue an event that includes a data packet containing the new user's login ID, full name, hashed password, and other pertinent details.
- This model can be appealing to developers familiar with RESTful interfaces.
  - But, depending on the complexity of the system, it can lead to a lot of data traffic on the network and data duplication in storage.

Event-driven patterns – sourcing

- **Event-sourcing**
- The goal of this model is to represent every change of state in a system as an event, each recorded in chronological order.
- In so doing, the event stream itself becomes the principal source of truth for the system.
  - E.g., it should be possible to "replay" a sequence of events to recreate the state of a SQL database at a given time.
- This model presents a lot of possibilities, but it can be challenging to get right.
  - particularly when events require participation from external systems.

Event-driven advantages (over REST)

- **Asynchronous**
  - Allows resources to move to the next task once a unit of work is complete.
  - Events are queued or buffered which prevents consumers from putting back pressure on producers or blocking them.
- **Loose Coupling**
  - Services operate independently, without knowledge of other services, including their implementation details and transport protocol.
  - Services under an event model can be updated, tested, and deployed independently and more easily.
- **Easy Scaling**
  - Since the services are decoupled and typically perform only one task, tracking bottlenecks and scaling a service is easier.
- **Recovery Support**
  - Can recover lost work by "replaying" events from the past.

https://dzone.com/articles/best-practices-for-event-driven-microservice-archi

Event-driven disadvantages

- They are easy to **over-engineer** by separating concerns that might be simpler when closely coupled.
  - they can require significant upfront investment, and often result in additional complexity, service contracts or schemas, polyglot build systems, and dependency graphs.
- **Complex data and transaction** management.
  - Typically do not support ACID transactions.
  - Systems must carefully handle inconsistent data between services, incompatible versions, duplicate events.

- Even with these drawbacks, …
  - an event-driven architecture is usually the better choice for enterprise-level microservice systems.
  - The pros—scalable, loosely coupled, dev-ops friendly design—outweigh the cons.

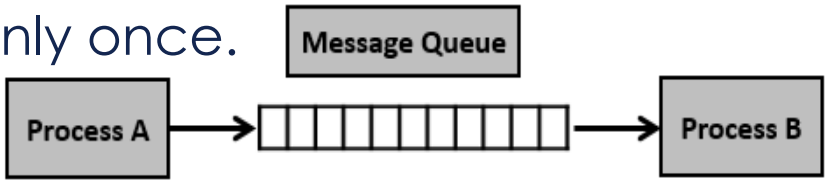
Event-driven components

- Source/sink
  - In an E-D architecture, virtually everything generates events: devices, apps, user activity, etc. Many others consume them.
- Messaging system
  - This is the way events flow in the architecture, and the central element of the Event Notification pattern.
- Server-side applications
  - This is our code, e.g. web apps, microservices, functions, event processing apps, or rich data pipelines.
- Storage and analytics
  - An E-D architecture changes how we think about storage
    - e.g., if we do event sourcing, then we also record events.
- Event logging and monitoring
  - A downside of E-D is the lack of predictability.
    - The architecture must have a robust method for tracing event paths, monitoring app performance, and making troubleshooting tolerable.

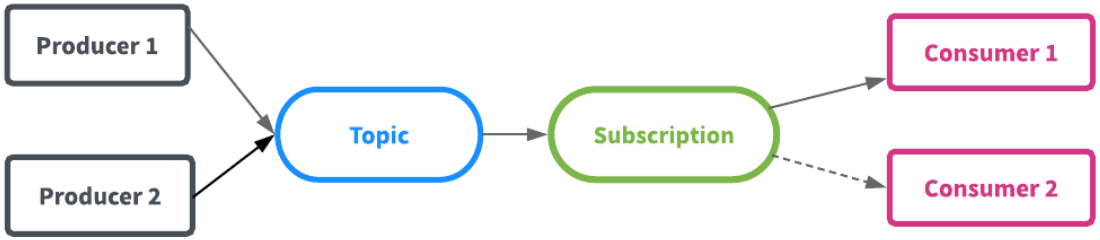
Messaging systems

Messaging systems – models

- Point to Point – Message queue
  - Messages are sent to a queue to pre-defined receivers.
  - One-to-one relationship between sender and consumer.
  - Each message is consumed only once.



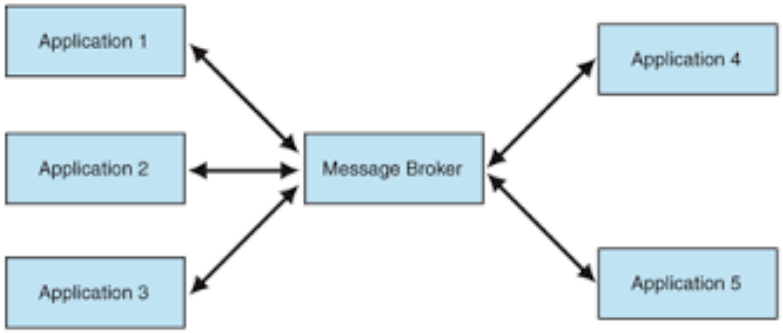
- Publish-Subscribe
  - Each message is published to a topic, and every application that subscribes to that topic gets a copy of all messages published to it.
  - Message producers are also known as publishers and consumers are known as subscribers.



Messaging systems

- Managing the messages' flow
  - Messages are "put into" a source queue.
  - They are then "taken from" a destination queue.
  - How/Who/What moves a message from a source queue to a destination queue?

- Queue Manager / **Message Broker**
  - Function as message-queuing "relay" that interact with distributed applications.



Message broker

- Message Broker is built to extend MQ
  - it can understand the content of each message that it moves through the Broker.

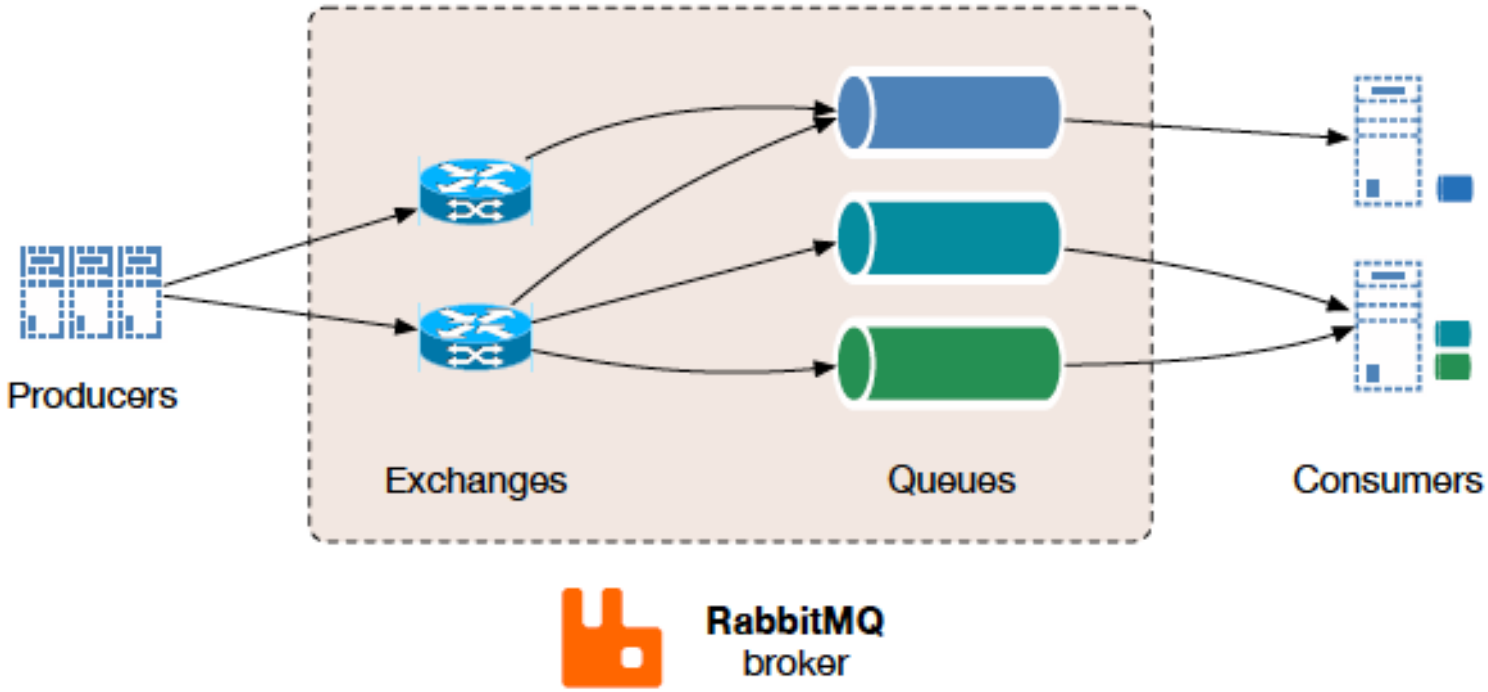
- Message Broker can do the following:
  - divide the publisher and consumer
  - store and route the messages between services
  - converts between different transport protocols
  - identifies and distributes business events from disparate sources

Message broker

- When a message broker is needed?
  - If we want to control data feeds, e.g., the number of registrations in a system.
  - When the task is to put data to several applications and avoid direct usage of their API.
  - When there is a need to complete processes in a defined order like a transactional system.
- There are many messaging tools..
  - E.g. Apache ActiveMQ, RabbitMQ
- .. and protocols
  - E.g. AMQP (Advanced Message Queuing Protocol), MQTT (MQ Telemetry Transport)

RabbitMQ

- RabbitMQ is one such open source message broker software that implements AMQP.



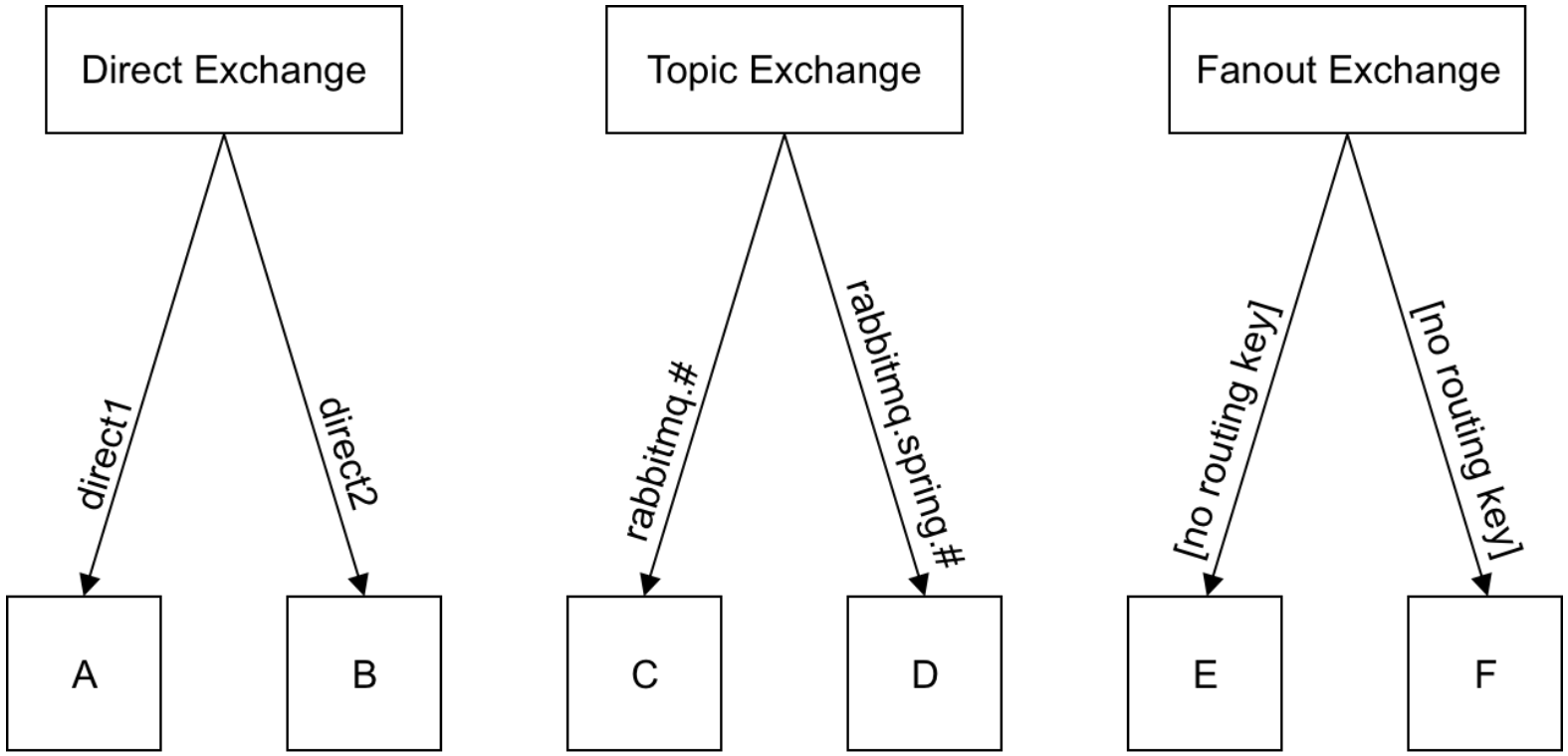
RabbitMQ – Main concepts

- **Messages** contains attributes (like headers in a request) and a payload (the message content).
- Messages are published to an entity, **exchange**, which distribute the messages to **queues** (or Topics).
- The rules for delivering the messages to the right queues are defined through
  - **bindings** (links between exchanges and queues), and
  - **routing keys** (a specific message attribute used for routing).
- Messages stored in queues are
  - delivered continually to subscribers, or
  - fetched by consumers on demand.

RabbitMQ – Exchange types

- **Direct Exchange** - It routes messages to a queue by matching routing key equal to binding key.
- **Fanout Exchange** - It ignores the routing key and sends message to all the available queues.
- **Topic Exchange** – It routes messages to multiple queues by a partial matching of a routing key. It uses patterns to match the routing and binding key.
- **Headers Exchange** – It uses message header instead of routing key.
- **Default(Nameless) Exchange** - It routes the message to queue name that exactly matches with the routing key.

RabbitMQ – Exchange types



Spring Boot – Messaging with RabbitMQ

- Set up the RabbitMQ Broker
- Spring Initializr
- E.g. RabbitMQ dependency

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-amqp -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
    <version>2.2.1.RELEASE</version>
</dependency>
```

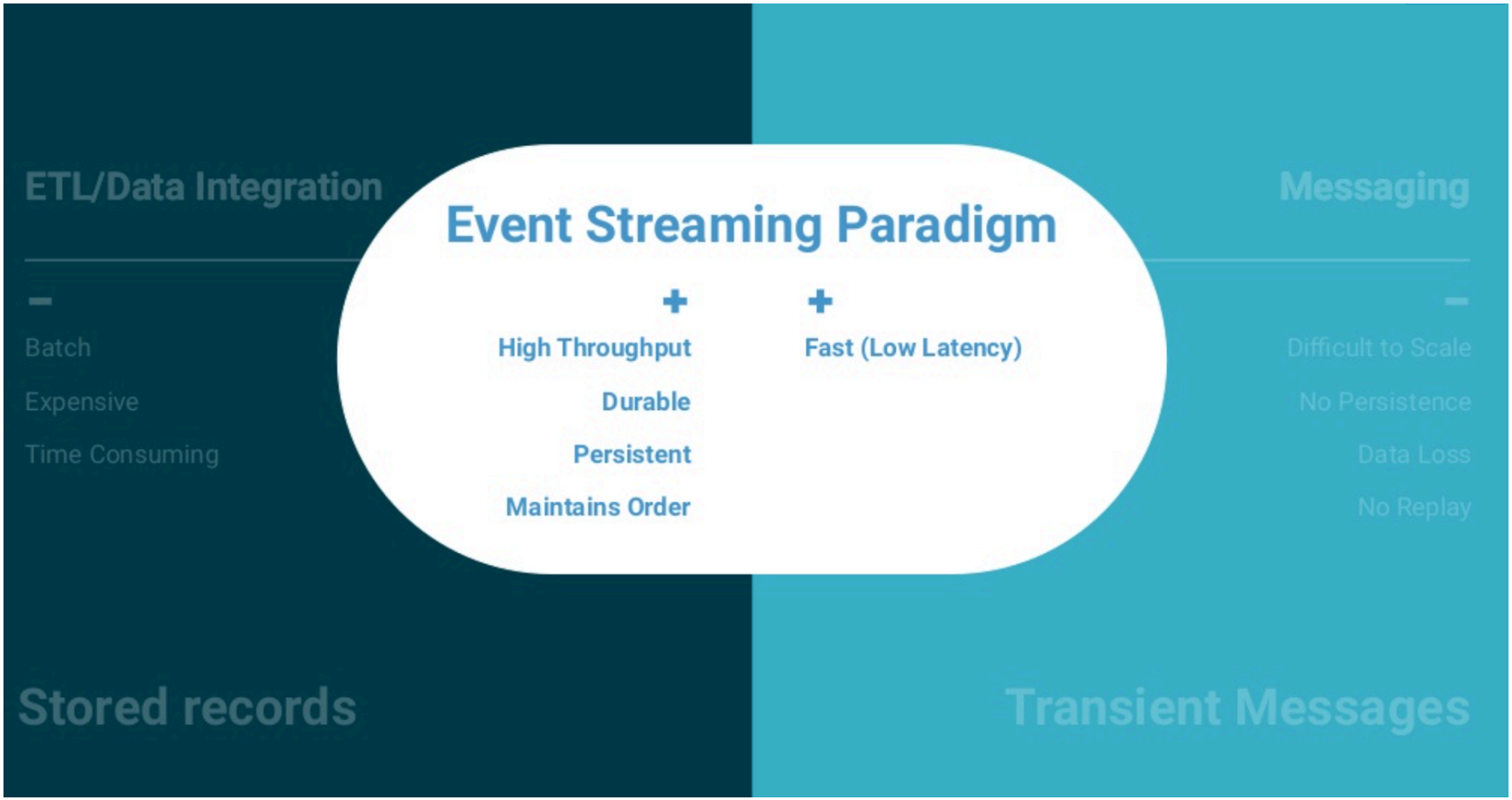
- https://spring.io/guides/gs/messaging-rabbitmq/

Event streaming systems
Apache Kafka

Stored records *vs* Messaging

https://pt.slideshare.net/ConfluentInc/what-is-apache-kafka-and-what-is-an-event-streaming-platform

The Event Streaming Paradigm



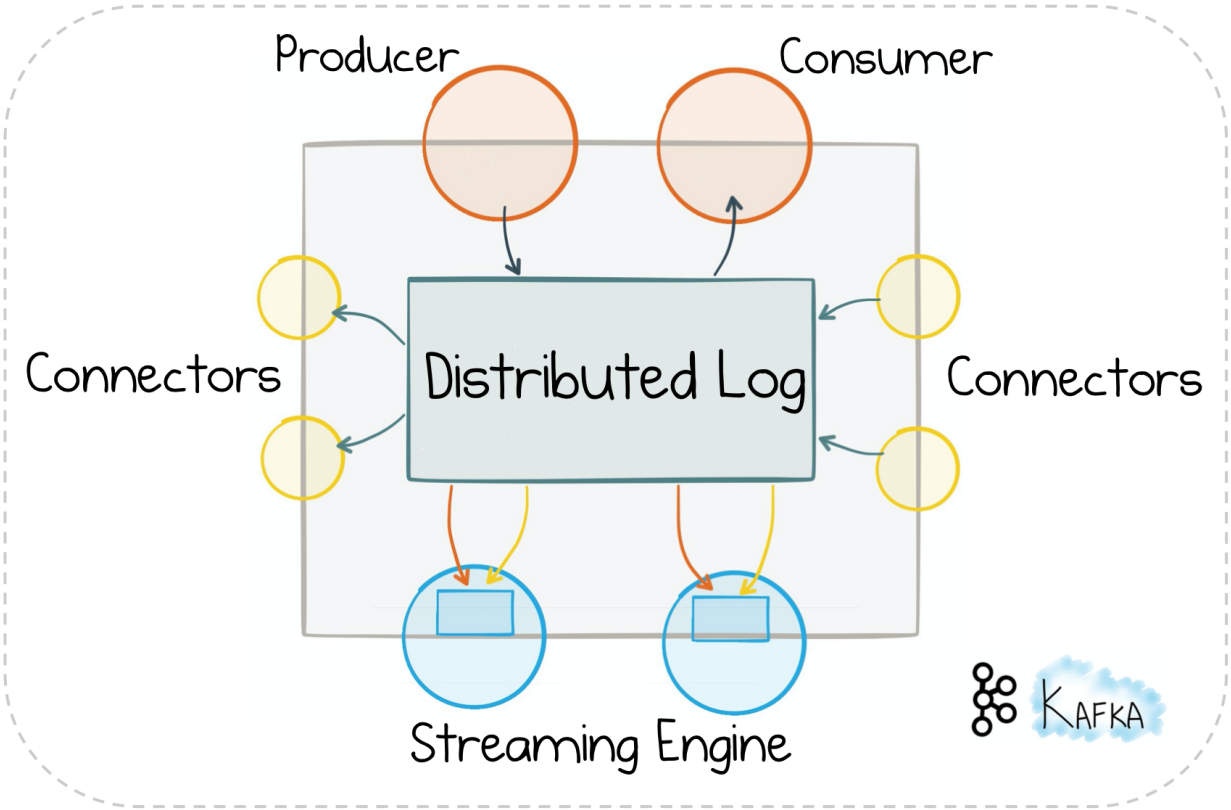
The Event Streaming Paradigm

To rethink data as not stored records or transient messages, but instead as a continually updating stream of events

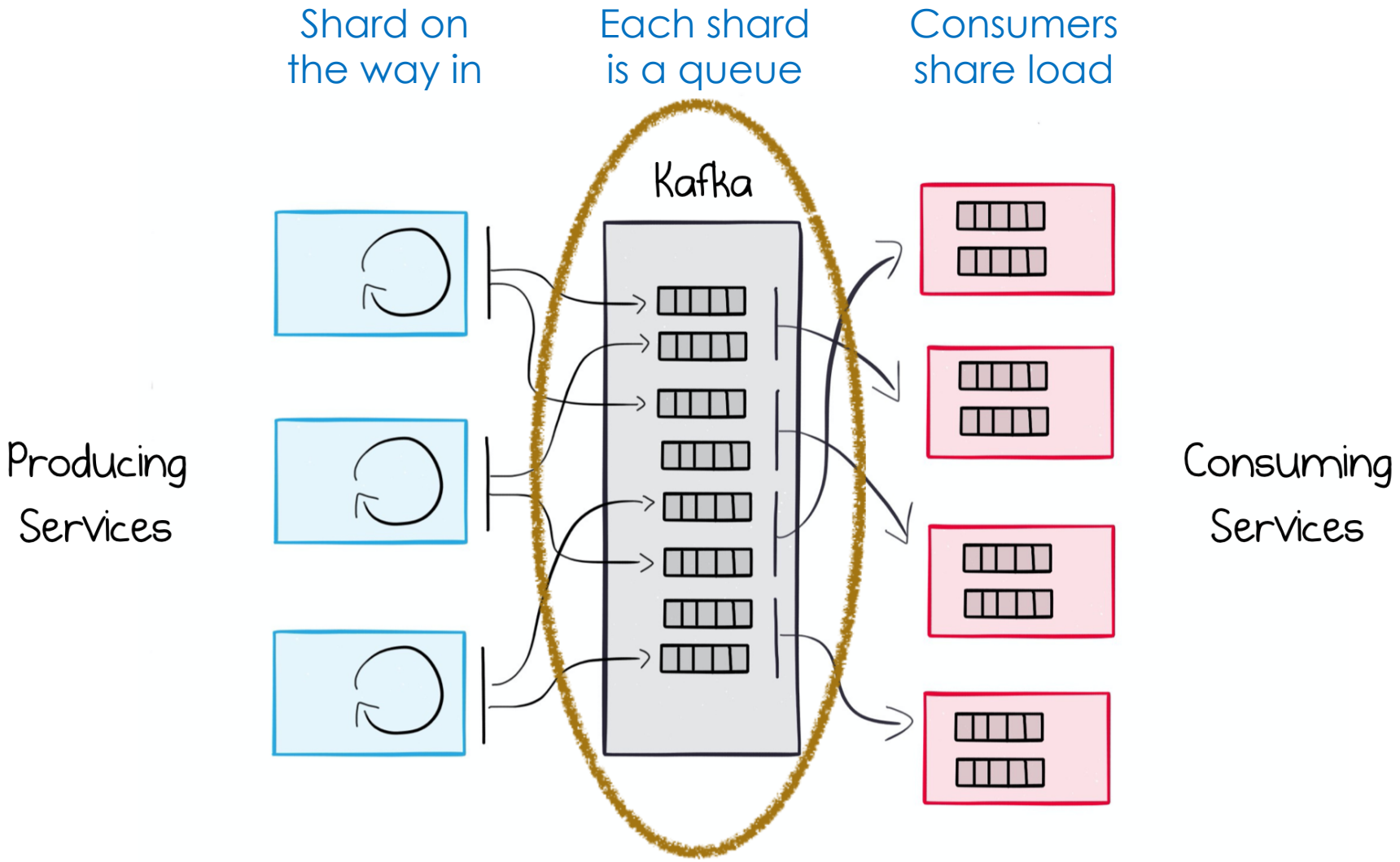
https://pt.slideshare.net/ConfluentInc/what-is-apache-kafka-and-what-is-an-event-streaming-platform

Apache Kafka

- Apache Kafka is made of distributed, immutable, append-only commit logs

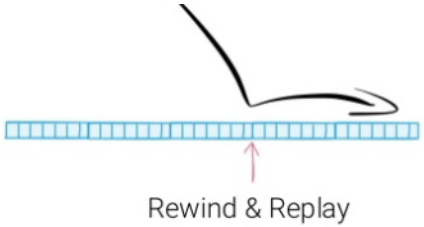


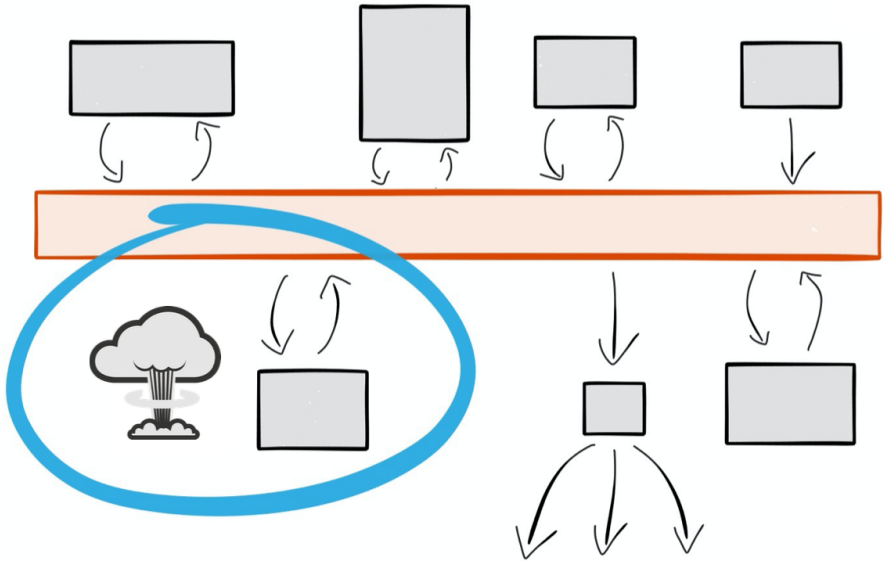
The Distributed Log



The Distributed Log

- The Service Backbone
  - Scalable/Load Balanced, Fault Tolerant, Concurrent, Strongly Ordered, Stateful





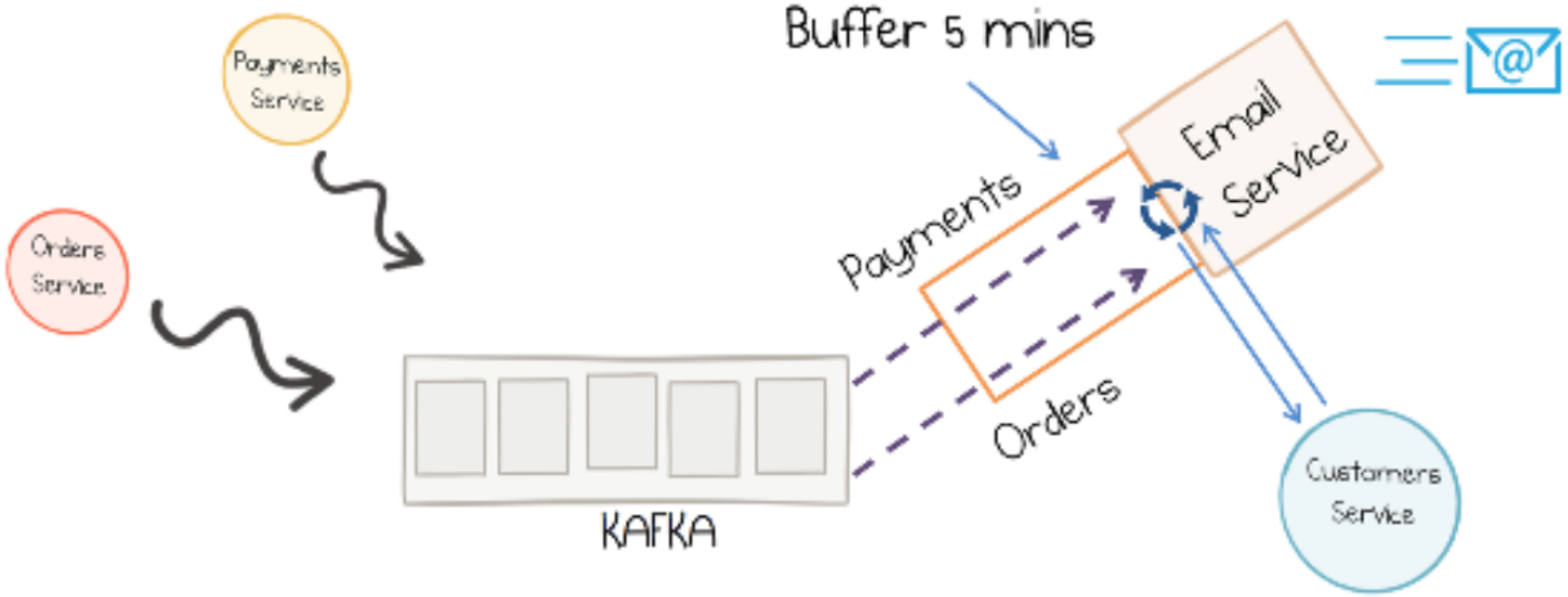
- A place to keep data-on-the-outside

Event-driven patterns

- Previously…
  - Event notification, Event-carried state transfer, Event-sourcing
- In **Event Sourcing**, events are a core element – the source of true.
  - Being stored, immutably, in the order they were created in, the event log expresses exactly what the system did.
- **Command Query Response Segregation** (**CQRS**) is a natural progression from this.
  - Decoupling writing and reading operations.
  - As a simple example, we might write events to Kafka (write model), read them back, and then push them into a database (read model).

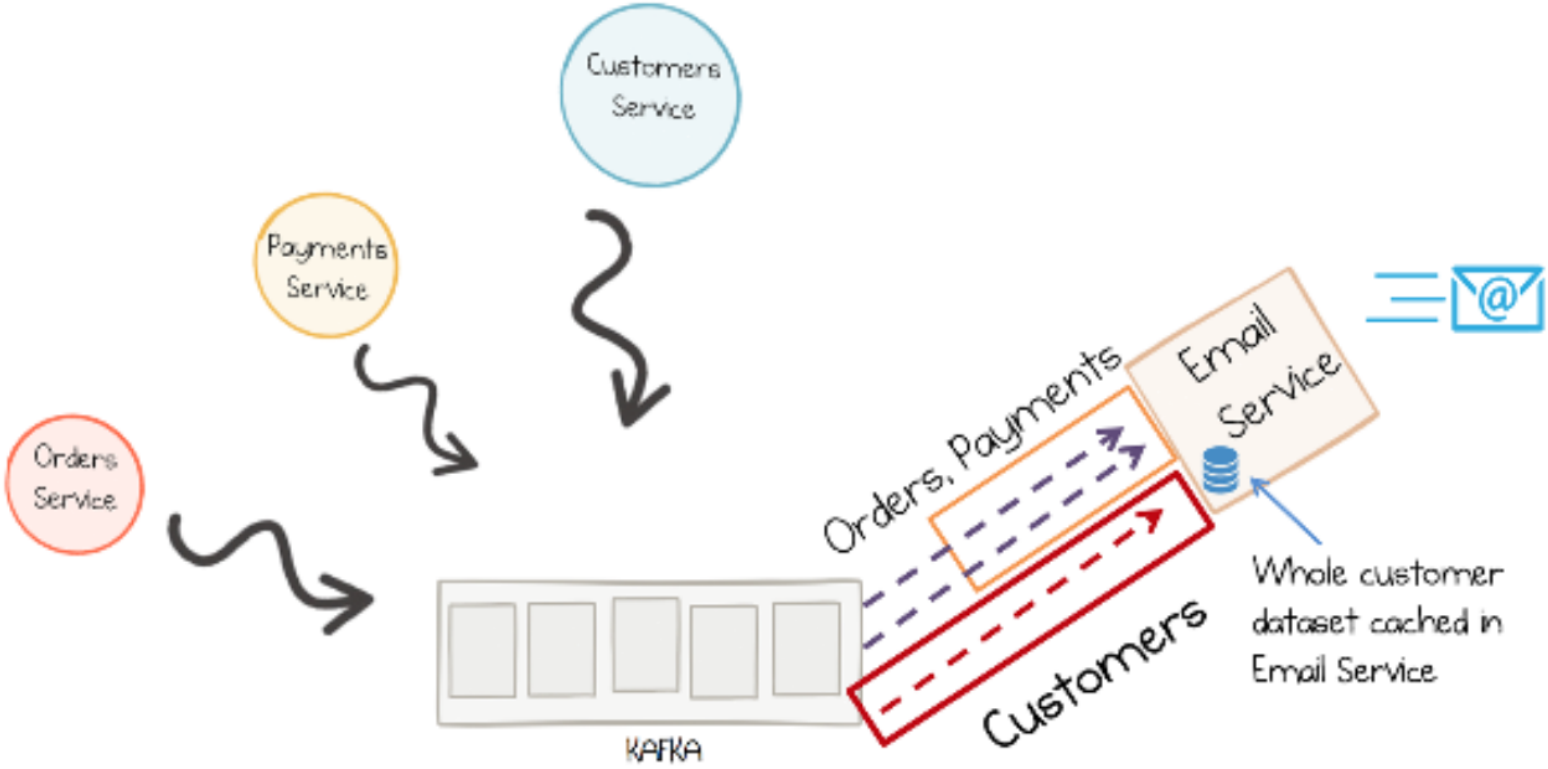
Stateless Streaming approach

- Example: A stateless streaming service that looks up reference data in another service at runtime



Stateful Streaming approach

- Example: A stateful streaming service that replicates the Customers topic into a local table, held inside the Kafka Streams API.



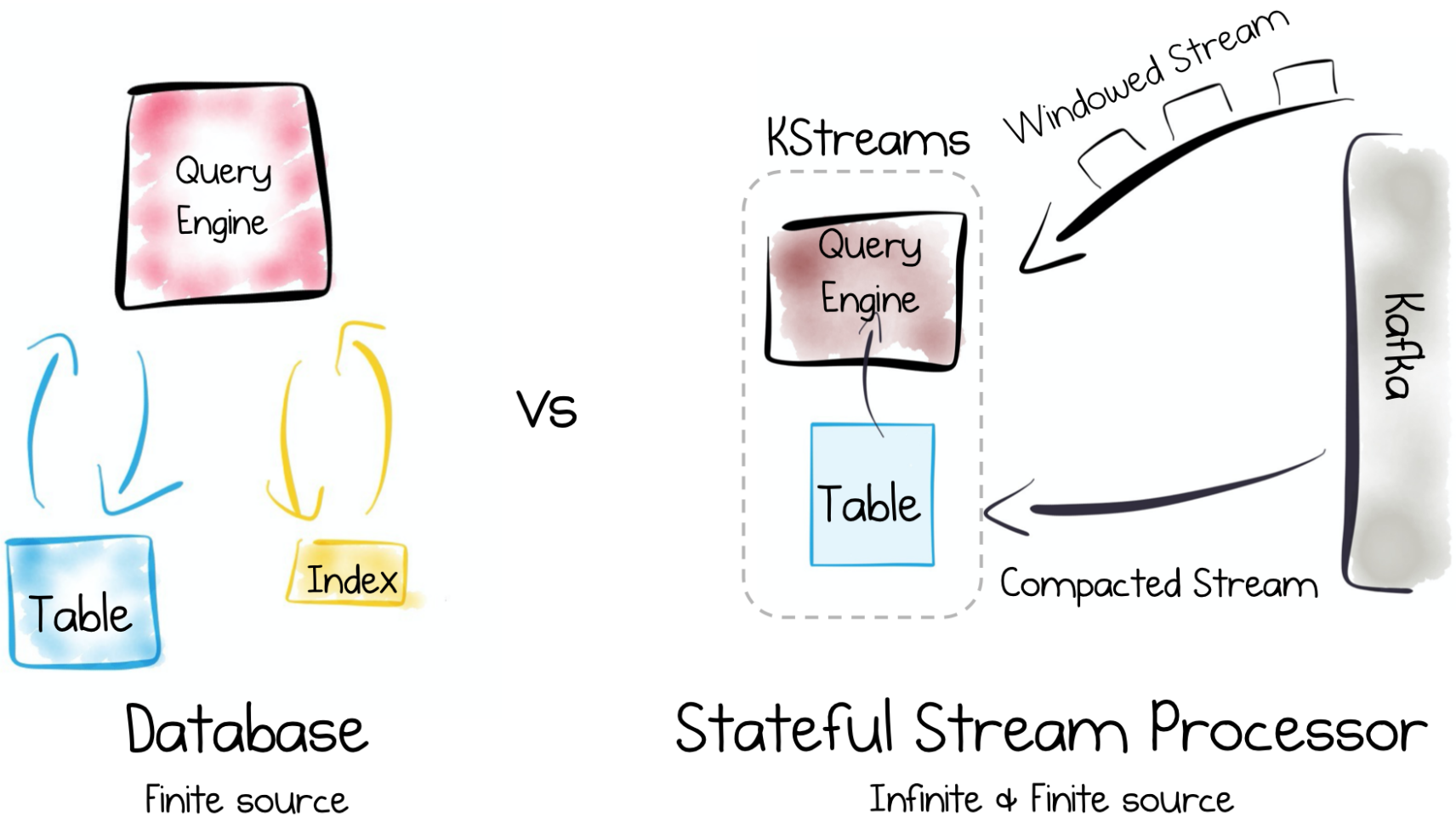
Stateful Streaming approach

- **Disadvantages**:
  - The service is now stateful, meaning for an instance of the email service to operate it needs the relevant customer data to be present.
  - This means, in the worst case, loading the full dataset on startup.
- **Advantages**:
  - The service is no longer dependent on the worst-case performance or liveness of the customer service.
  - The service can process events faster, as each operation is executed without making a network call.
  - The service is free to perform more data-centric operations on the data it holds.

Stateful Streaming approach

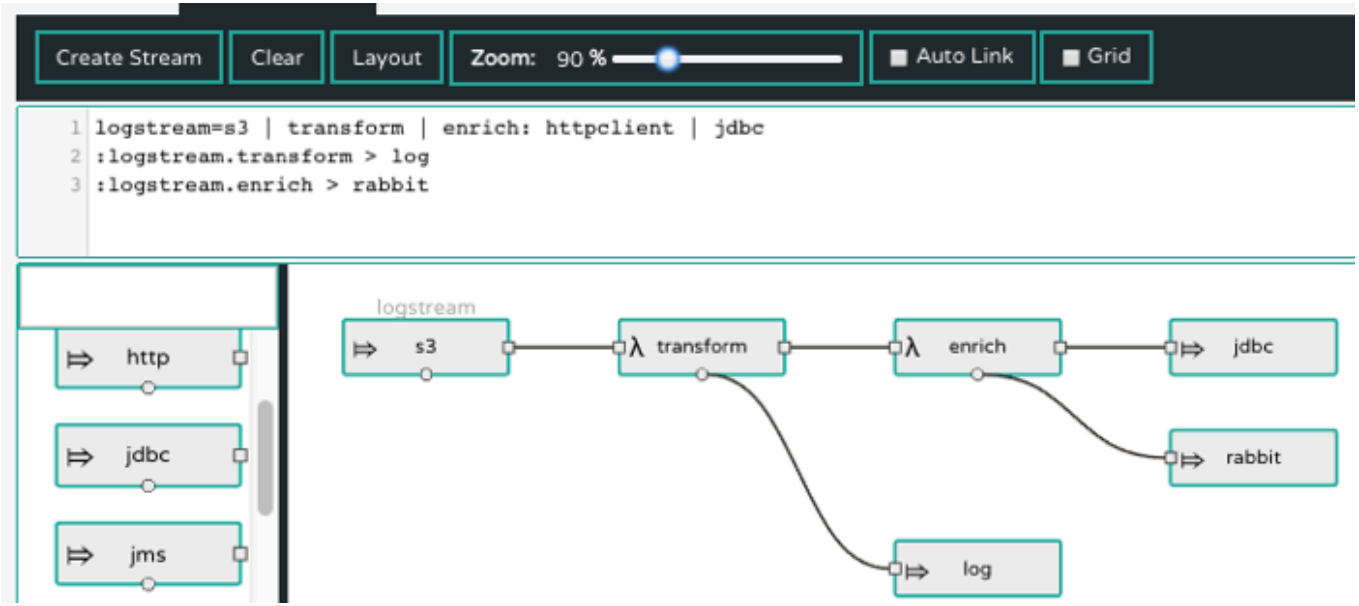
- Being stateful comes with some challenges:
  - when a new node starts, it must load all stateful components (i.e., state stores).
- Kafka Streams, for instance, provides three mechanisms to simplify stateful:
  - It uses a technique called standby replicas, which ensure that for every table or state store on one node, there is a replica kept up to date on another.
  - Disk checkpoints are created periodically so that, should a node fail and restart, it can load its previous checkpoint.
  - Finally, compacted topics are used to keep the dataset as small as possible. This acts to reduce the load time for a complete rebuild should one be necessary.

A Stateful Stream Processing



Streaming in Spring

- **Spring Cloud Stream** helps fully abstract code from the underlying messaging engine.
- **Spring Cloud Data Flow** allow to create and orchestrate data pipelines, e.g. data ingest, real-time analytics, and data import/export.



Summary

- Event-driven architecture is gaining in popularity, and with good reason.
  - From a technical perspective, it provides an effective method of wiring up microservices.
  - The interest in serverless functions - such as AWS Lambda, Azure Functions, or Knative - is growing, and these are inherently event-based.
  - Moreover, when coupled with modern streaming data tools like Apache Kafka, event-driven architectures become more versatile, resilient, and reliable than with earlier messaging methods.
- But perhaps the most important "feature" of the event-driven pattern is that it models how businesses operate in the real world.