Error handling with Apache Kafka

From patterns to code

Error handling with Apache Kafka

From patterns to code

Cédric Schaller
Senior Architect
ELCA (Switzerland)

cedric.schaller@elca.ch
𝕏CedSoftEng

Agenda

A concrete example

An illustration of potential pitfalls in a distributed application relying on Kafka

A checklist for increased reliability

How to configure Kafka to reduce the potential for errors

Patterns for error handling with Kafka

How to deal with errors when they do happen

Demo

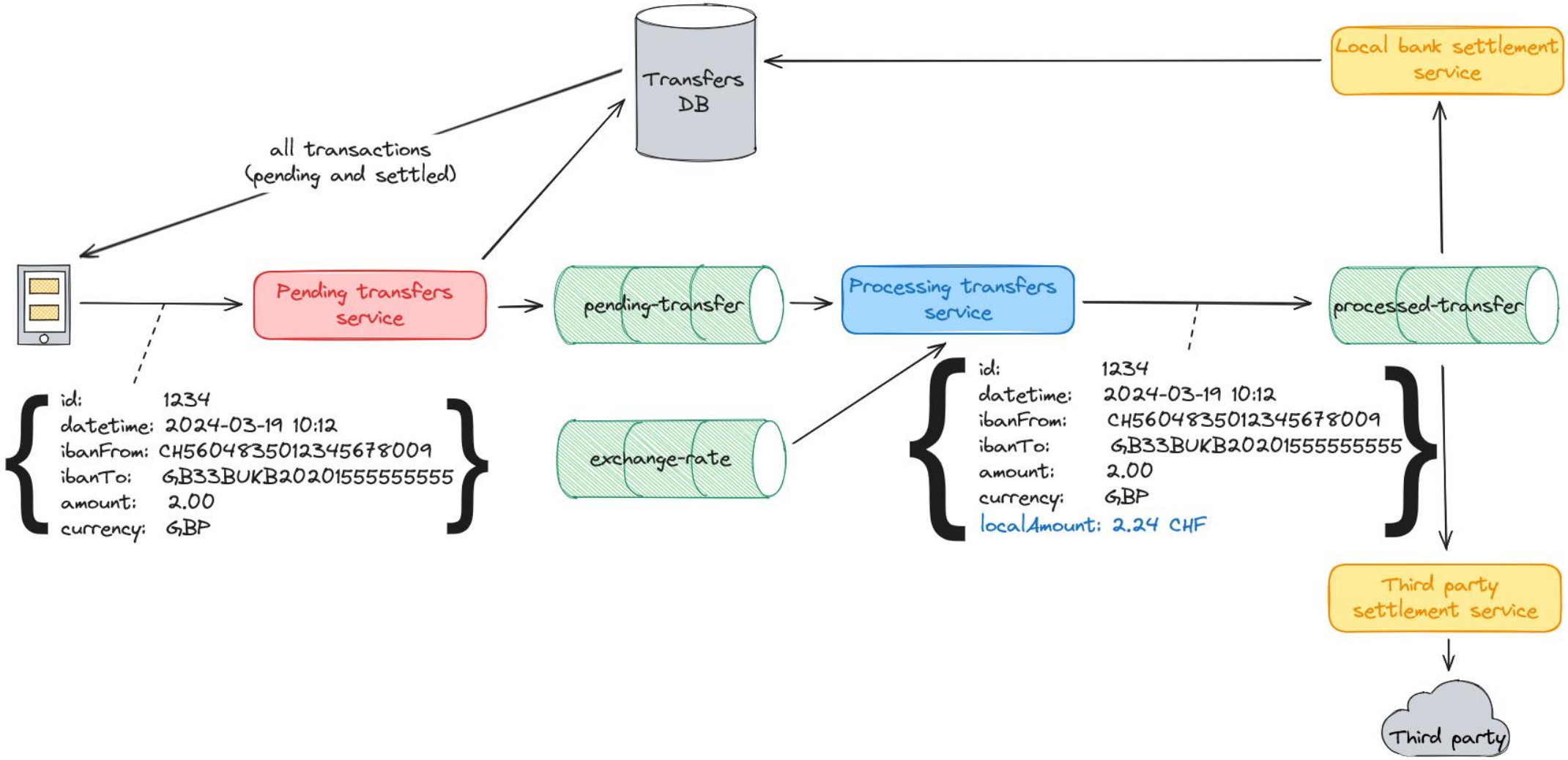
Let’s put theory into practice!

Further integration patterns

How to integrate safely with external systems like SQL DBs and how to handle transactionality across systems

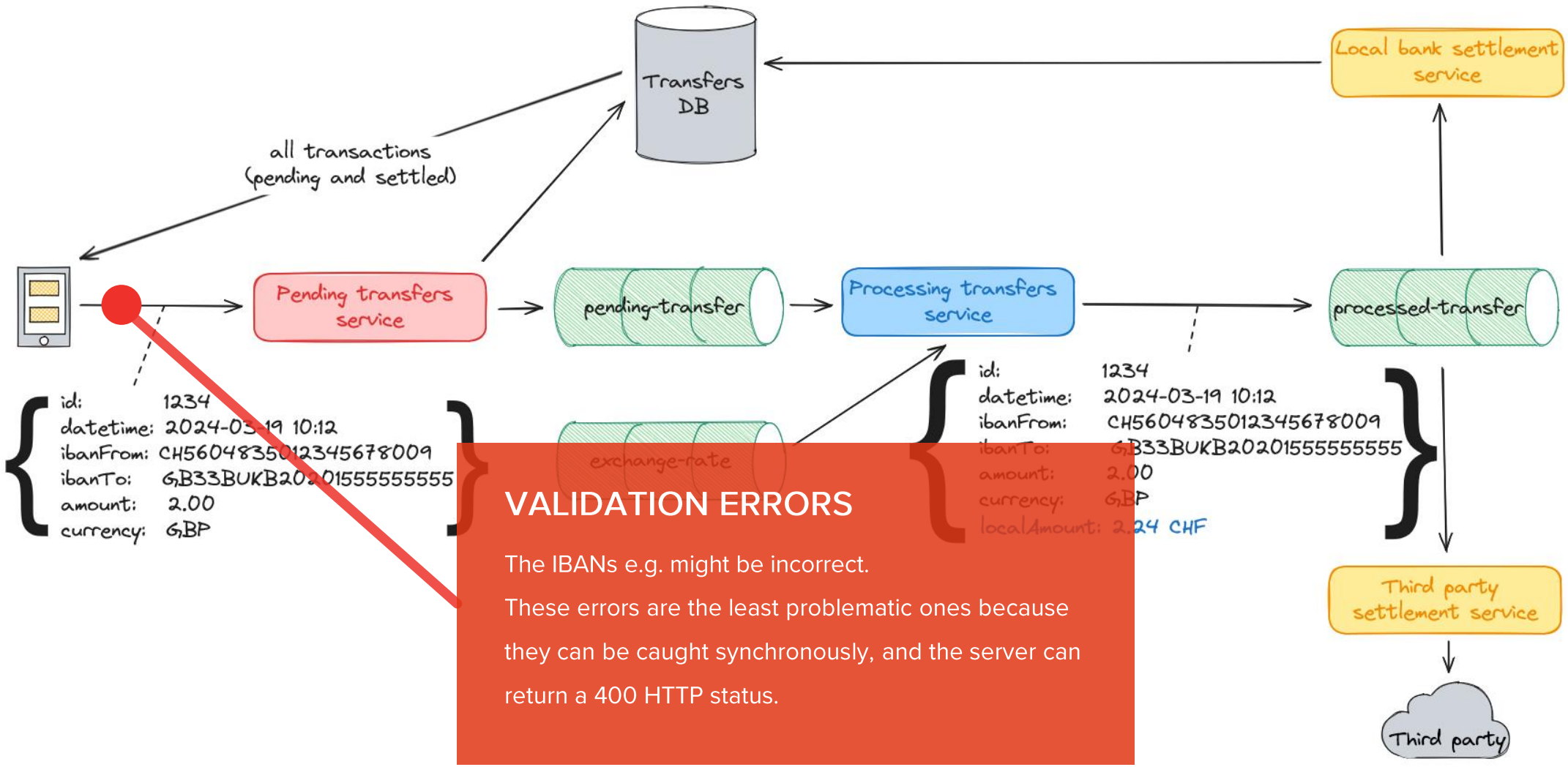
A concrete example

Mobile money transfer




What could possibly go wrong? 🤷‍♂️
MARYLAND 1923

Sources of errors

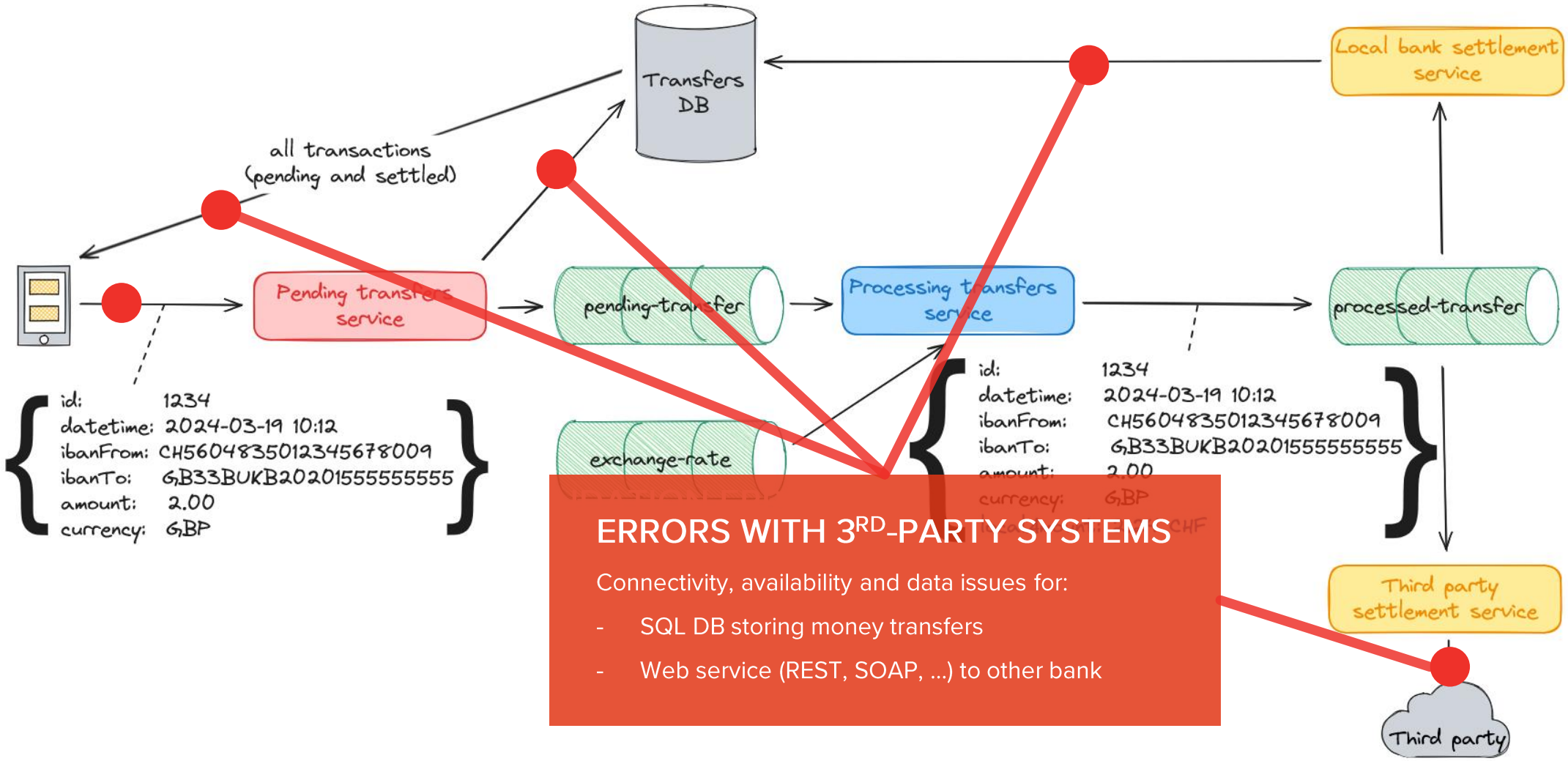


VALIDATION ERRORS

The IBANs e.g. might be incorrect.

These errors are the least problematic ones because they can be caught synchronously, and the server can return a 400 HTTP status.

Sources of errors

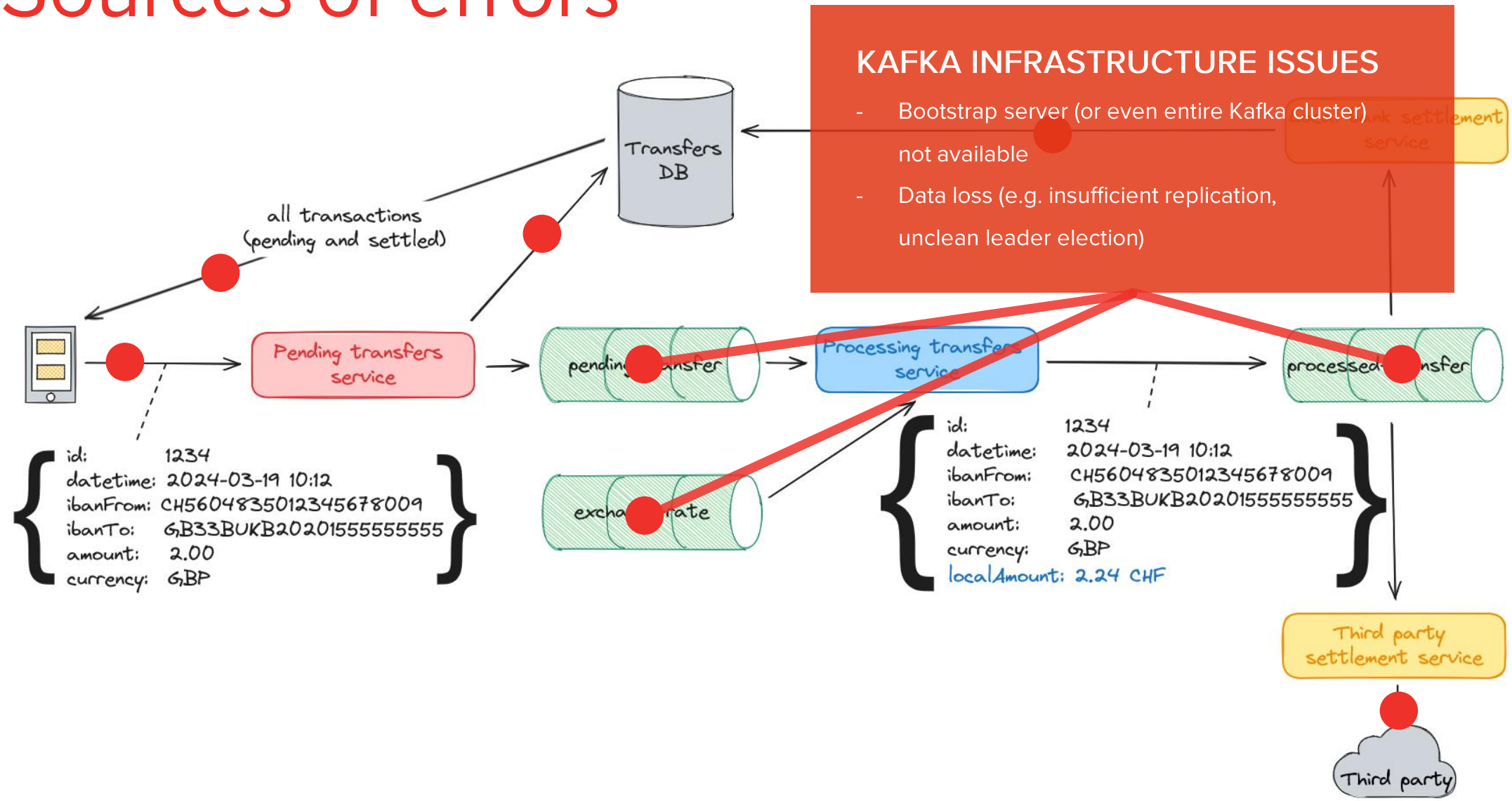


ERRORS WITH 3RD-PARTY SYSTEMS

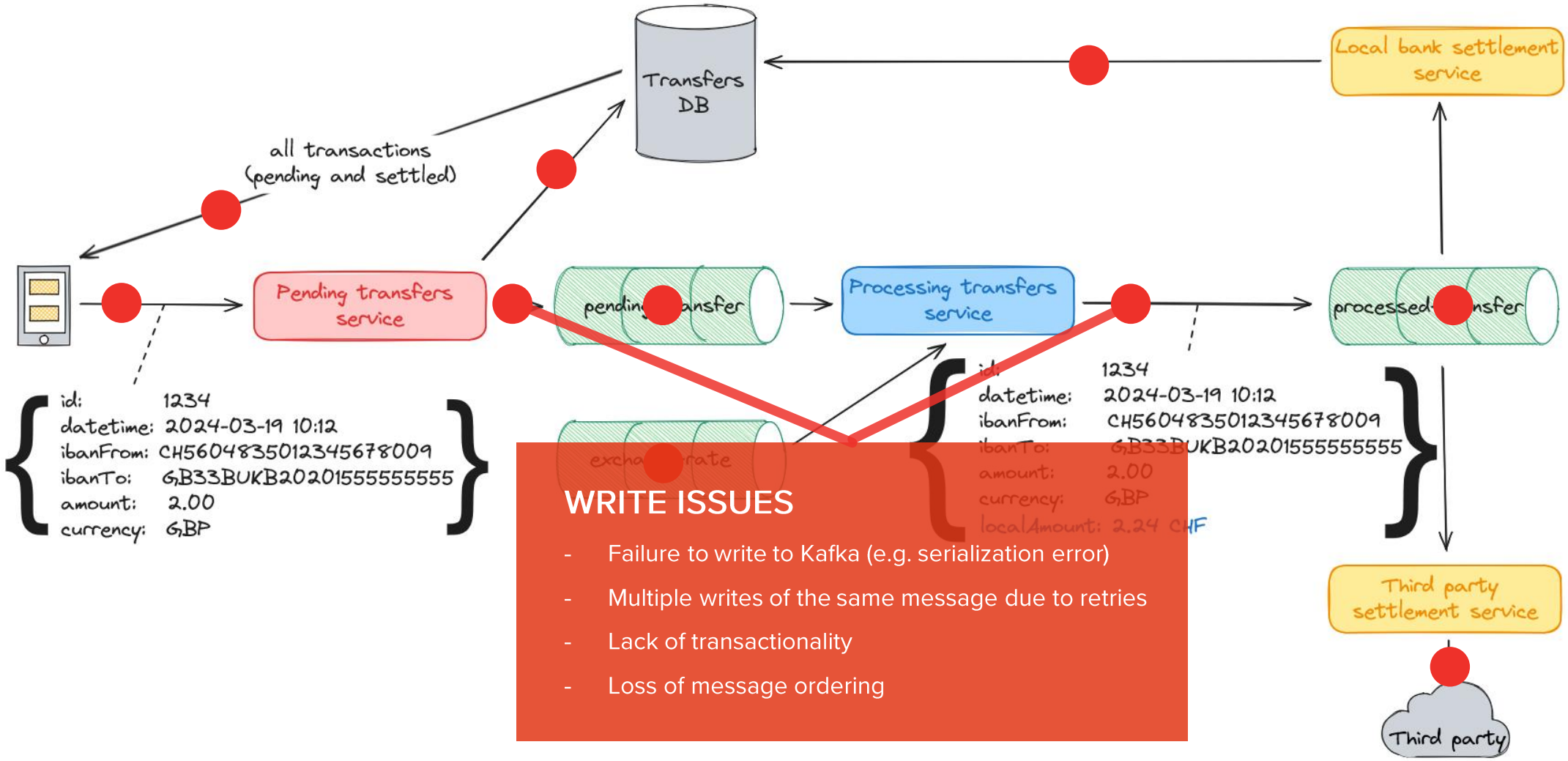
Connectivity, availability and data issues for:

- SQL DB storing money transfers
- Web service (REST, SOAP, ...) to other bank

Sources of errors



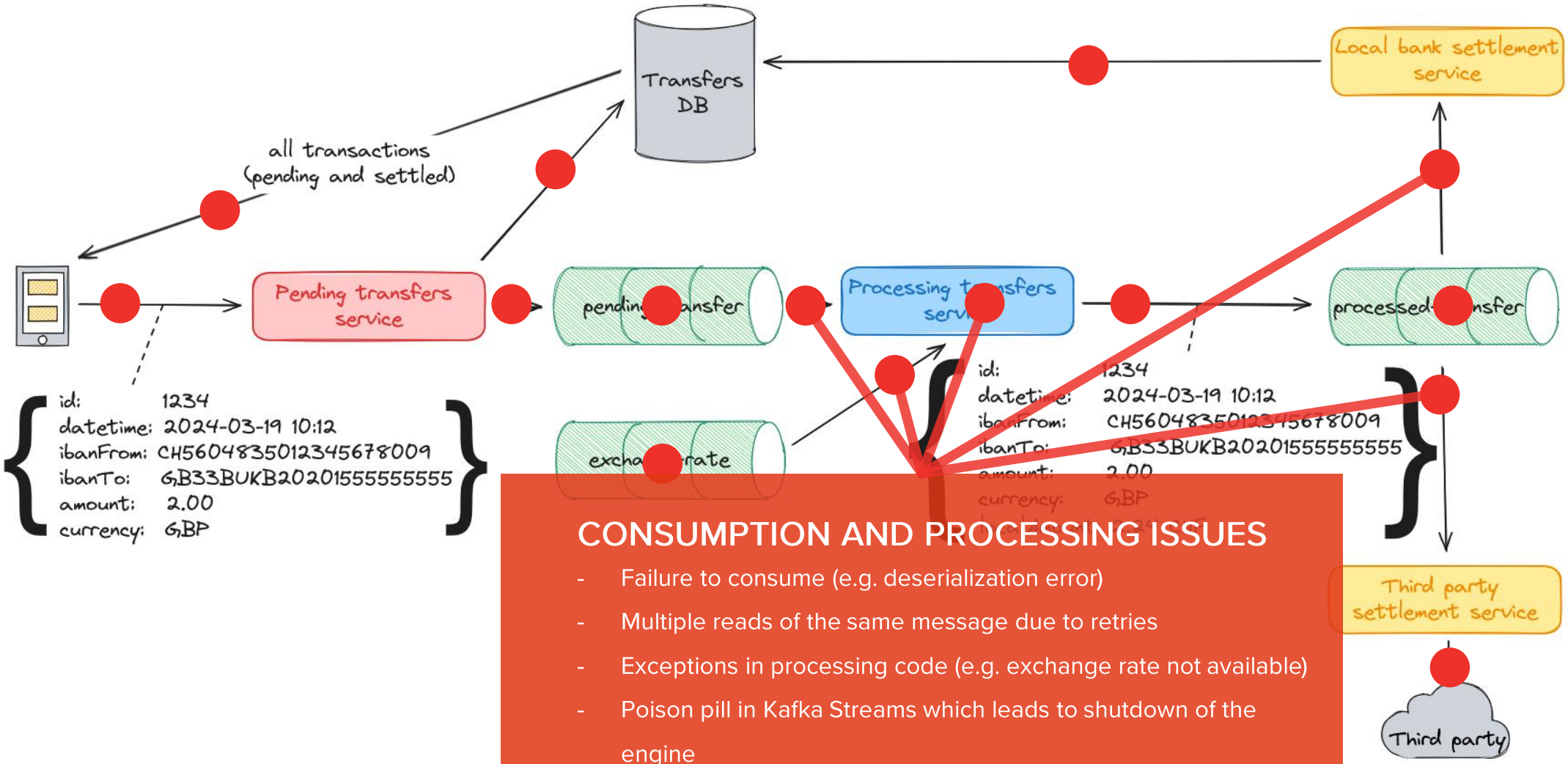
Sources of errors



WRITE ISSUES

- Failure to write to Kafka (e.g. serialization error)
- Multiple writes of the same message due to retries
- Lack of transactionality
- Loss of message ordering

Sources of errors

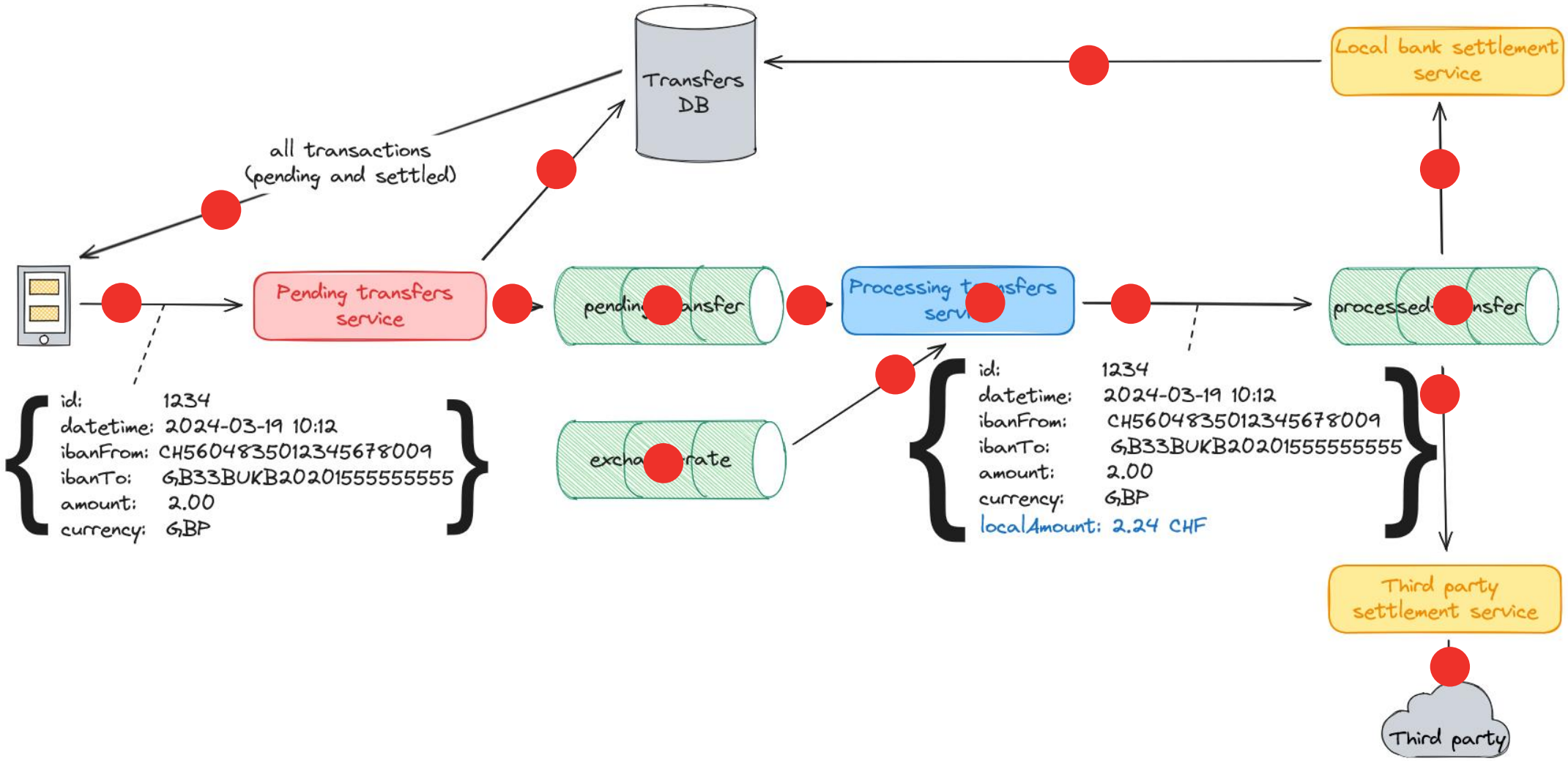


CONSUMPTION AND PROCESSING ISSUES

- Failure to consume (e.g. deserialization error)
- Multiple reads of the same message due to retries
- Exceptions in processing code (e.g. exchange rate not available)
- Poison pill in Kafka Streams which leads to shutdown of the engine

Sources of errors

What a nightmare!




Yeah, but Kafka surely somehow handles it! 💪
After all it *is* reliable!

Yes, but...

TRADE-OFFS

Kafka is a general-purpose middleware.

It supports a great variety of use cases, from IOT measurements to highly reliable banking software.

It cannot favor reliability by default, and at all costs.

Therefore, some tuning might be necessary to reach high reliability.

POISON PILLS

A poison pill is a record that has been produced to a Kafka topic but whose consumption always fails, no matter how often we try (non-transient exception).

Such a record will cause a shutdown of the Kafka Streams engine (safety first).

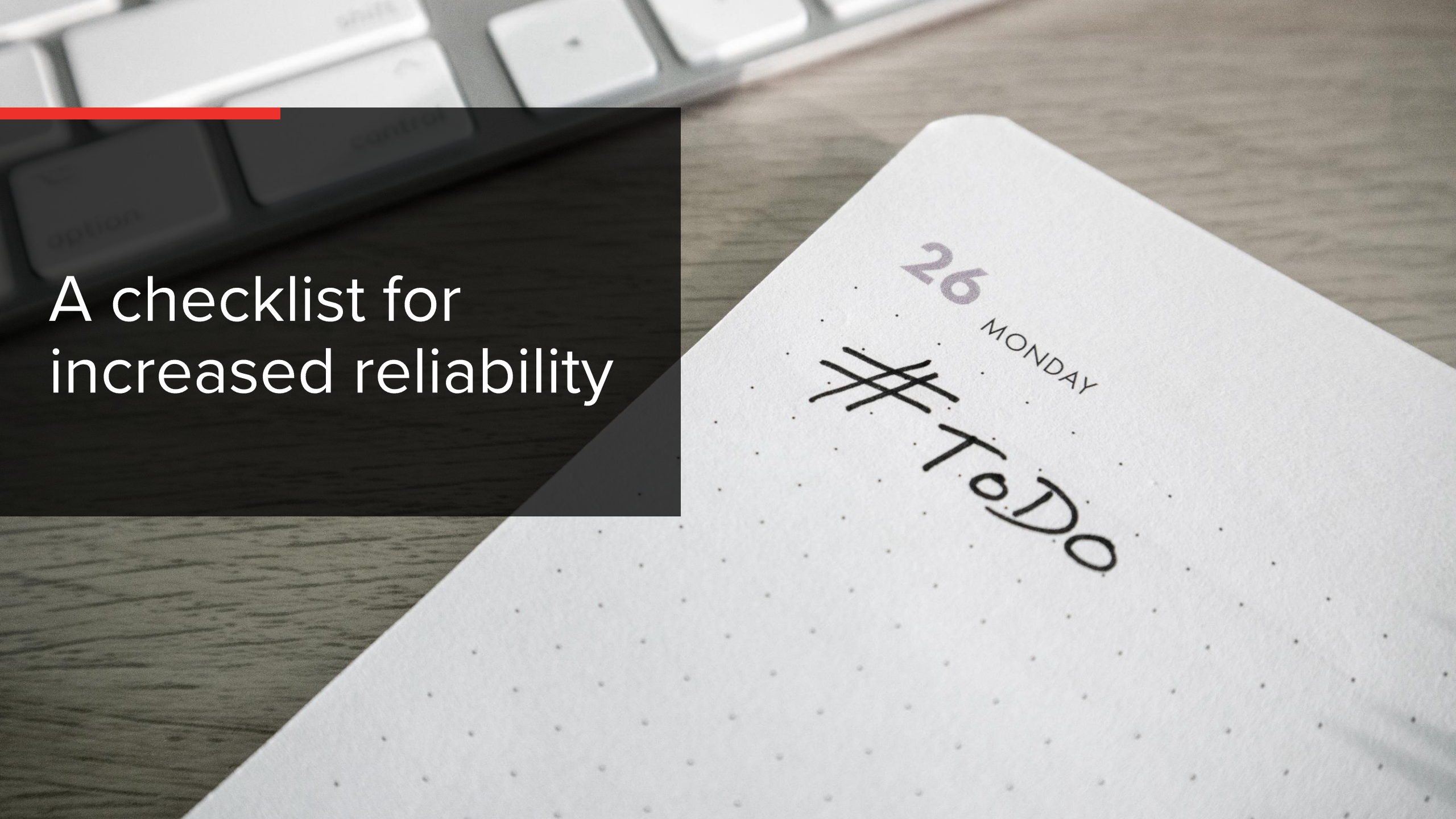
TRANSACTIONS

Kafka transactions are designed for 2 use cases:

- Multiple writes to multiple topics / partitions
- consume → process → produce

In particular, they are not designed to propagate to external systems (e.g. webservice calls)

A checklist for increased reliability



Producer configuration

WHAT KAFKA DOES FOR YOU (SINCE KAFKA 3.0)

`acks: all`

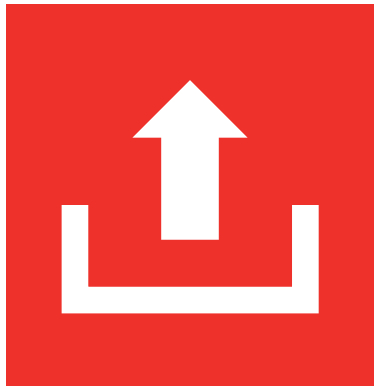
The leader waits for all in-sync replicas to acknowledge the record. This ensures that the record is not lost as long as at least one in-sync replica remains alive.

`enable.idempotence: true`

Ensures that exactly one copy of each message is written in the stream even in the event of a broker failure

`max.in.flight.requests.per.connection: 5`

The maximum number of unacknowledged requests the client will send on a single connection before blocking. Together with `enable.idempotence=true`, this ensures ordering preservation



OTHER PROPERTIES YOU MIGHT WANT TO TUNE

`retries`
`max.block.ms`
`delivery.timeout.ms`

WHAT YOU NEED TO DO

`bootstrap.servers`

List more than one broker to ensure that the cluster can be discovered even if one broker is down

`transactional.id`

Enables transactional delivery

If `transactional.id` is configured, `enable.idempotence` is implied

Consumer configuration

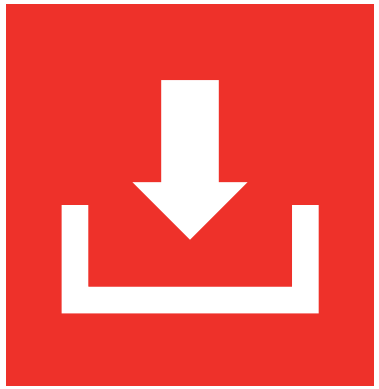
WHAT KAFKA DOES FOR YOU

`enable.auto.commit: true`

If true, the consumer's offset will be periodically committed in the background.

`auto.commit.interval.ms: 5000`

This represents a reasonable trade-off between performance and reliability



WHAT YOU NEED TO DO

`bootstrap.servers`

List more than one broker to ensure that the cluster can be discovered even if one broker is down

`isolation.level: read_committed`

Controls how to read messages written transactionally. Non-transactional messages will be returned unconditionally in either mode.

OTHER PROPERTIES YOU MIGHT WANT TO TUNE

`heartbeat.interval.ms`
`session.timeout.ms`
`connections.max.idle.ms`
`max.poll.interval.ms`
`request.timeout.ms`
`reconnect.backoff.max.ms`
`reconnect.backoff.ms`
`retry.backoff.ms`

Broker and topic configuration

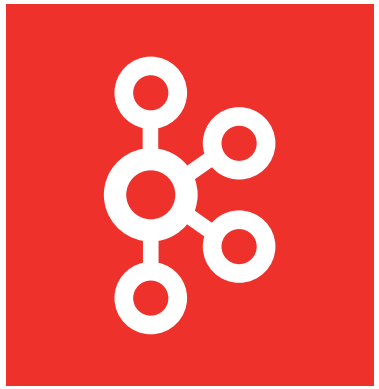
WHAT KAFKA DOES FOR YOU

`unclean.leader.election.enable: false`

Replica which are not in sync cannot become leaders. This prevents data losses (at the expense of availability)

`offsets.topic.replication.factor: 3`

The replication factor for the offsets topic (set higher to enhance reliability)



OTHER PROPERTIES YOU MIGHT WANT TO TUNE

`cleanup.policy`
`log.retention.*`
`delete.retention.ms`

WHAT YOU NEED TO DO

`(default.)replication.factor: 1 -> 3`

Determines how many times a topic will be replicated (including the leader)

`min.insync.replicas: 1 -> 2`

Specifies the minimum number of replicas that must acknowledge a write for it to be considered committed with `ack=all`. Otherwise, the producer will raise an exception

`broker.rack`

Rack of the broker. Rack aware replication will try to replicate across racks for higher fault tolerance

Kafka Streams configuration

IMPROVE RELIABILITY

`default.deserialization.exception.handler`

If not set, the Kafka Streams engine will shut down in the event of a deserialization error

`default.production.exception.handler`

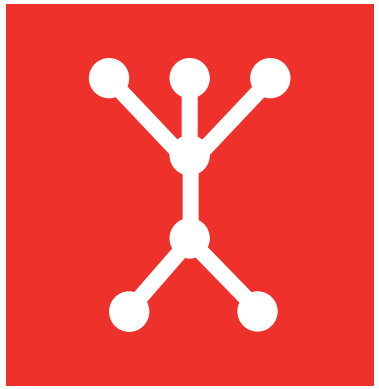
If not set, the Kafka Streams engine will shut down in the event of a serialization error

`processing.guarantee: exactly_once_v2`

As an alternative, the application can be designed with idempotence in mind

`retries: 2147483647`

Setting a value greater than zero will cause the client to resend any request that fails with a potentially transient error



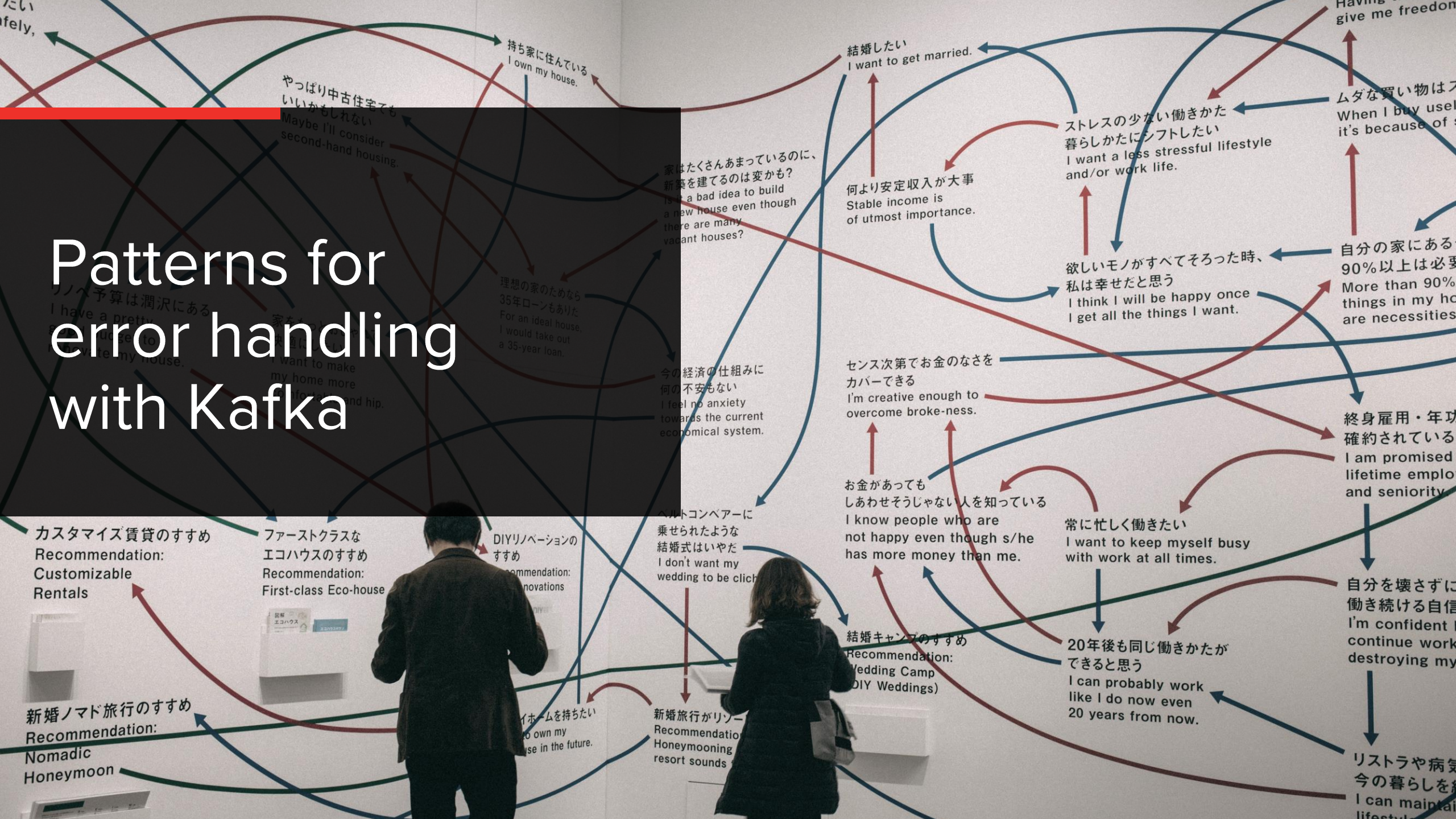
IMPROVE AVAILABILITY

`num.standby.replicas: 0 -> 1`

Standby replicas are shadow copies of local state stores. They are used to minimize the latency of task failover.

`state.dir`

Directory where the state is stored. Should be a persistent volume in Kubernetes for faster start-up time



Patterns for error handling with Kafka

Pattern 1: Stop on error

DESCRIPTION

When an error occurs, the application stops. Manual intervention is required.

WHEN TO USE?

When all input events must be processed in order without exception.

Examples:

- Kafka Streams applications stop when an unhandled error is encountered
- In Change Data Capture (CDC), events need to be processed in order, to guarantee consistency

DISCUSSION

- This is the easiest and safest approach (at the expense of availability)
- Stopping the world might not be acceptable in a production environment

Pattern 2: Dead letter topic (DLT)

DESCRIPTION

When an error occurs, the message is routed to a dead letter topic. Processing of subsequent messages carries on.

WHEN TO USE?

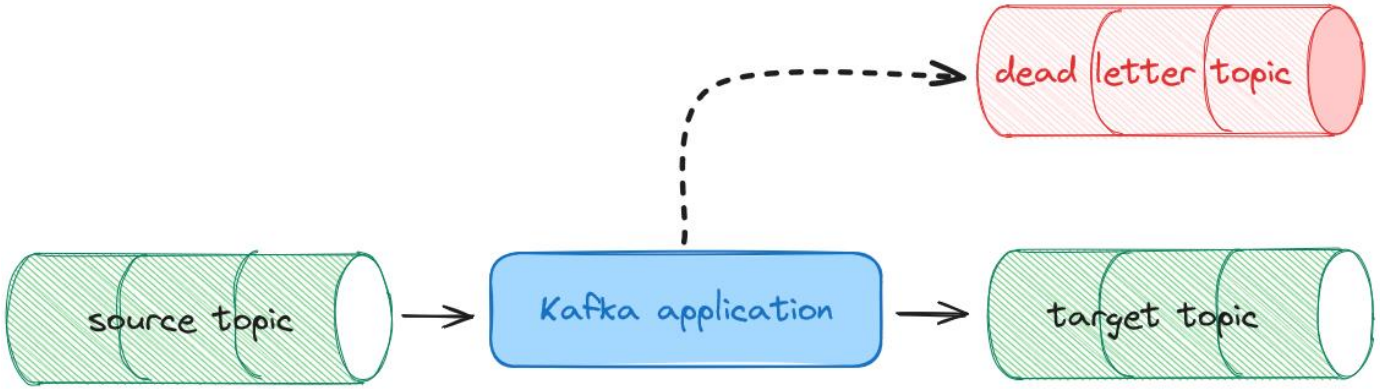
When retries do not make sense (e.g. `NullPointerException`) or maximum number of retries has been exhausted.

Example:

- Temperature measurements in an IOT cluster

DISCUSSION

- Ordering is no longer guaranteed
- If this is acceptable, guarantees a higher availability than “stop on error”
- A process to handle the messages in the dead letter topic must be defined



Pattern 3: Retry topic(s)

DESCRIPTION

When a transient error occurs, the message is routed to a retry topic. A *retry application* retries (potentially multiple times) processing the message.

WHEN TO USE?

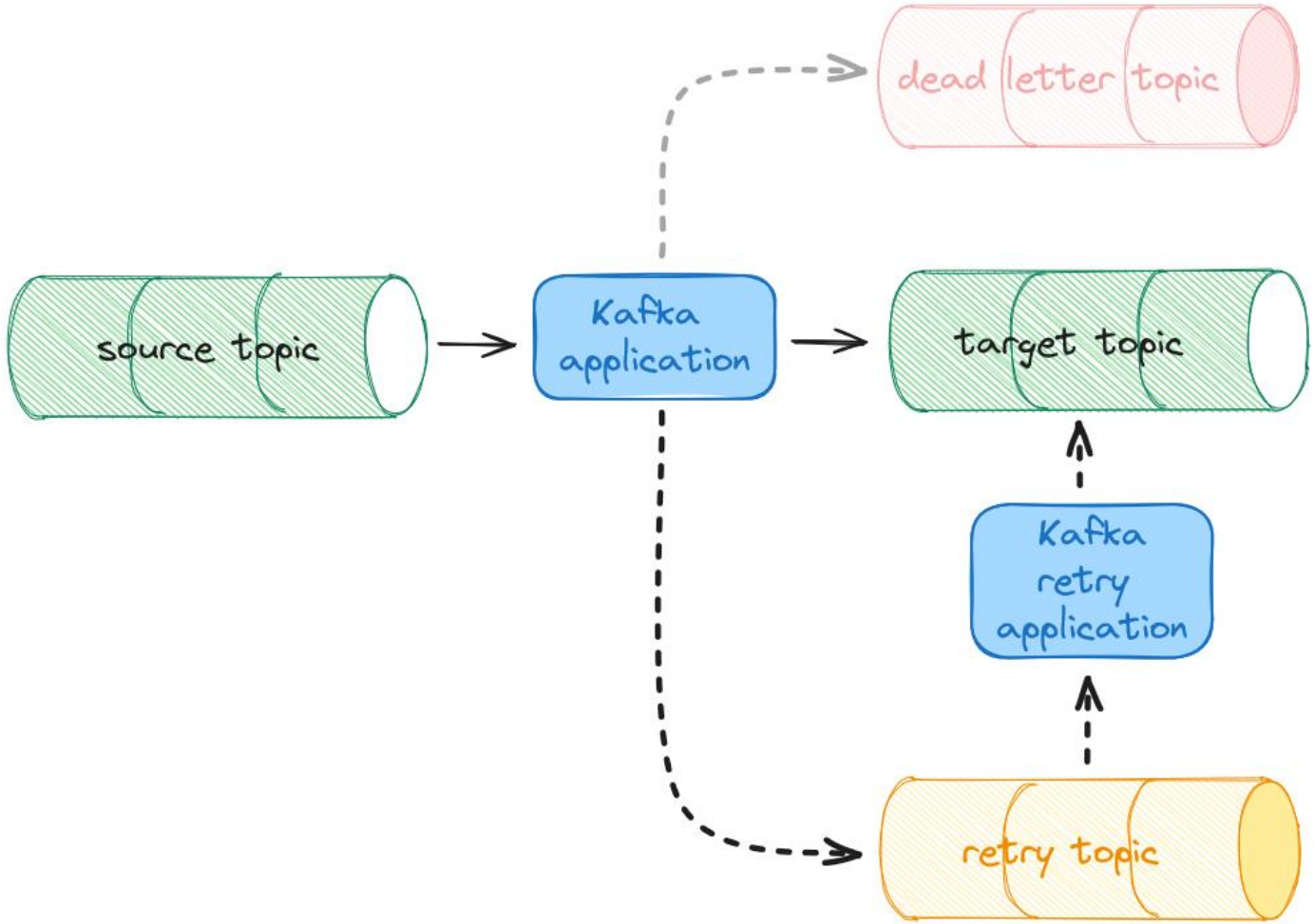
In the case of transient errors.

Example:

- An external service (web service, DB) is used to process the message and its availability cannot be guaranteed.

DISCUSSION

- Ordering is no longer guaranteed
- Retries do not require manual intervention (as opposed to DLT)
- There can be multiple retry topics
- Typically, when the maximum number of retries is reached, the message is routed to a DLT



Pattern 4: Maintaining order of events

DESCRIPTION

When a message related to an item cannot be processed, all messages related to that same item will be queued until the problem is solved.

WHEN TO USE?

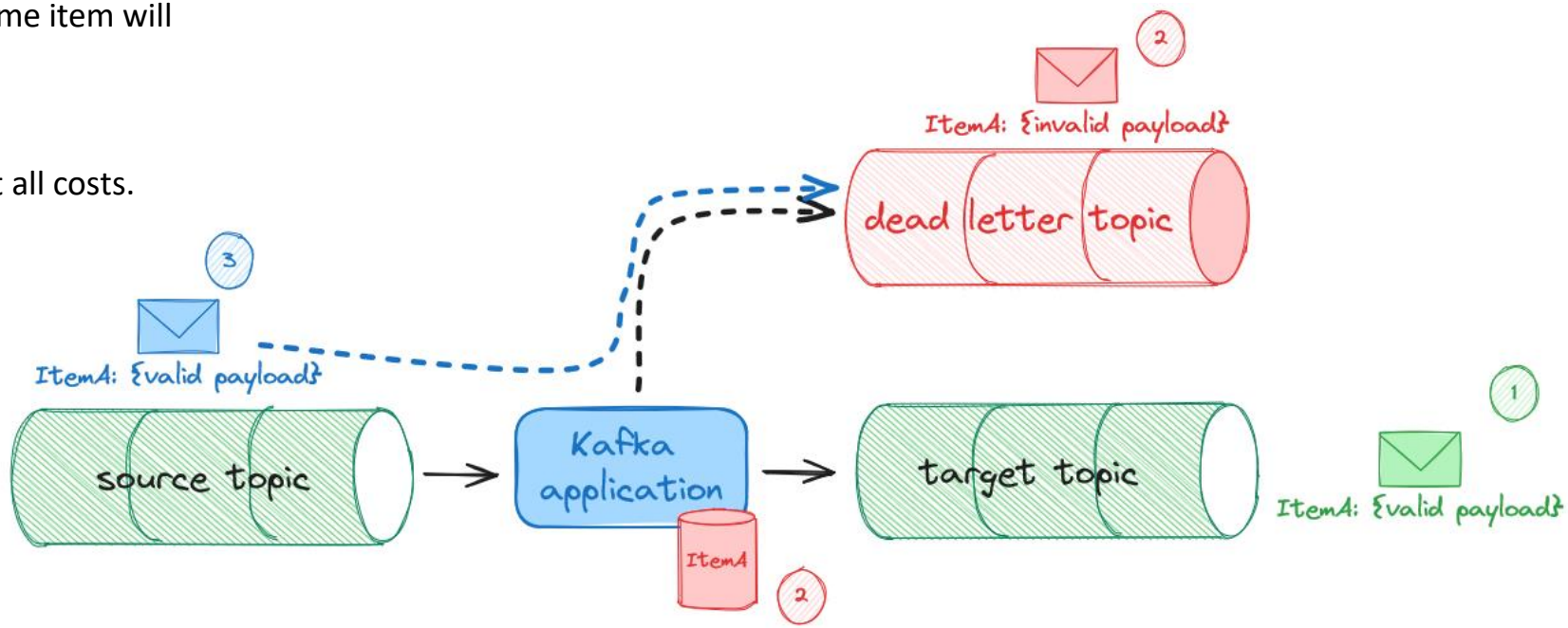
When the ordering must be maintained at all costs.

Example:

- Bank transactions

DISCUSSION

- Complex to implement. No OOTB support in Spring
- Can also be applied to a retry topic
- Different variants exist (e.g. with a redirect topic, see https://www.confluent.io/blog/error-handling-patterns-in-kafka/)

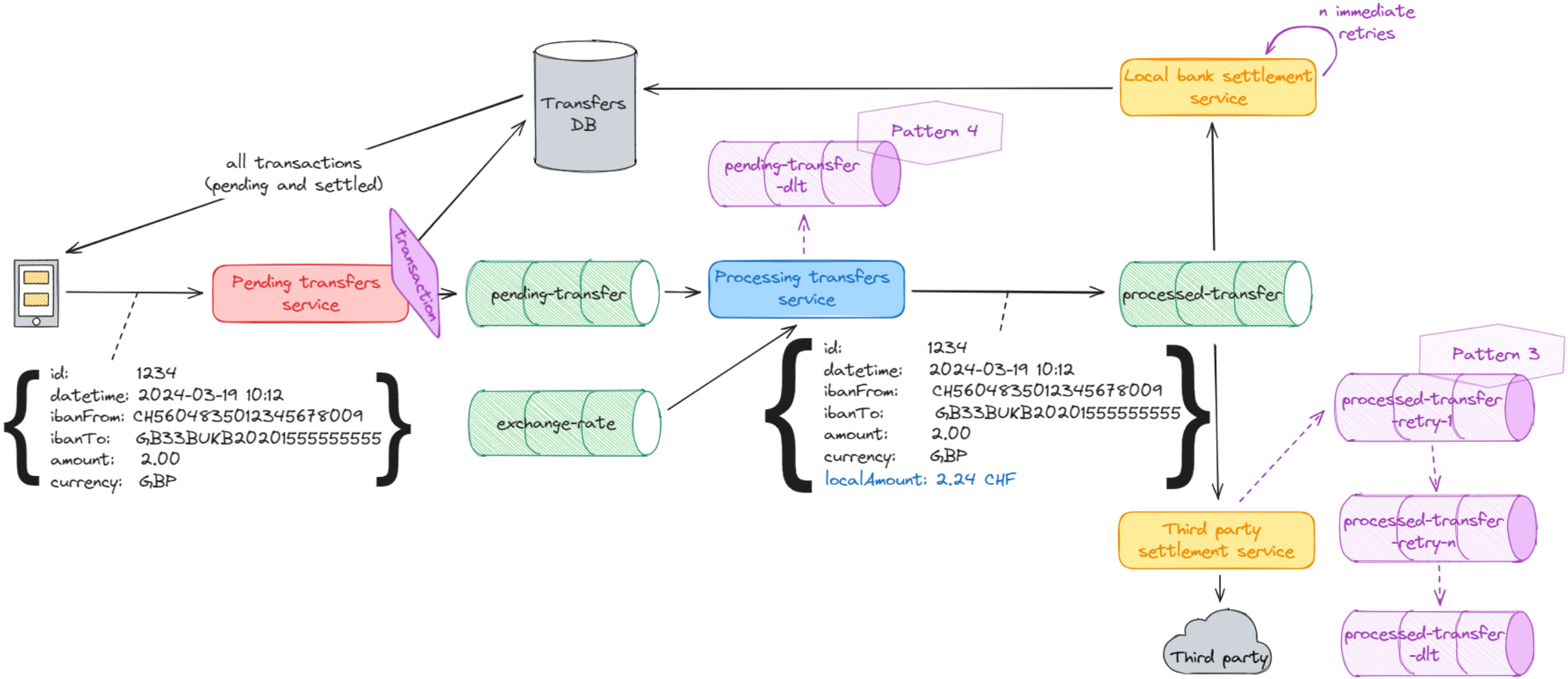


Demo

Let’s get our hands dirty

A concrete example

With error handling



Integration patterns

Transactional outbox pattern

DESCRIPTION

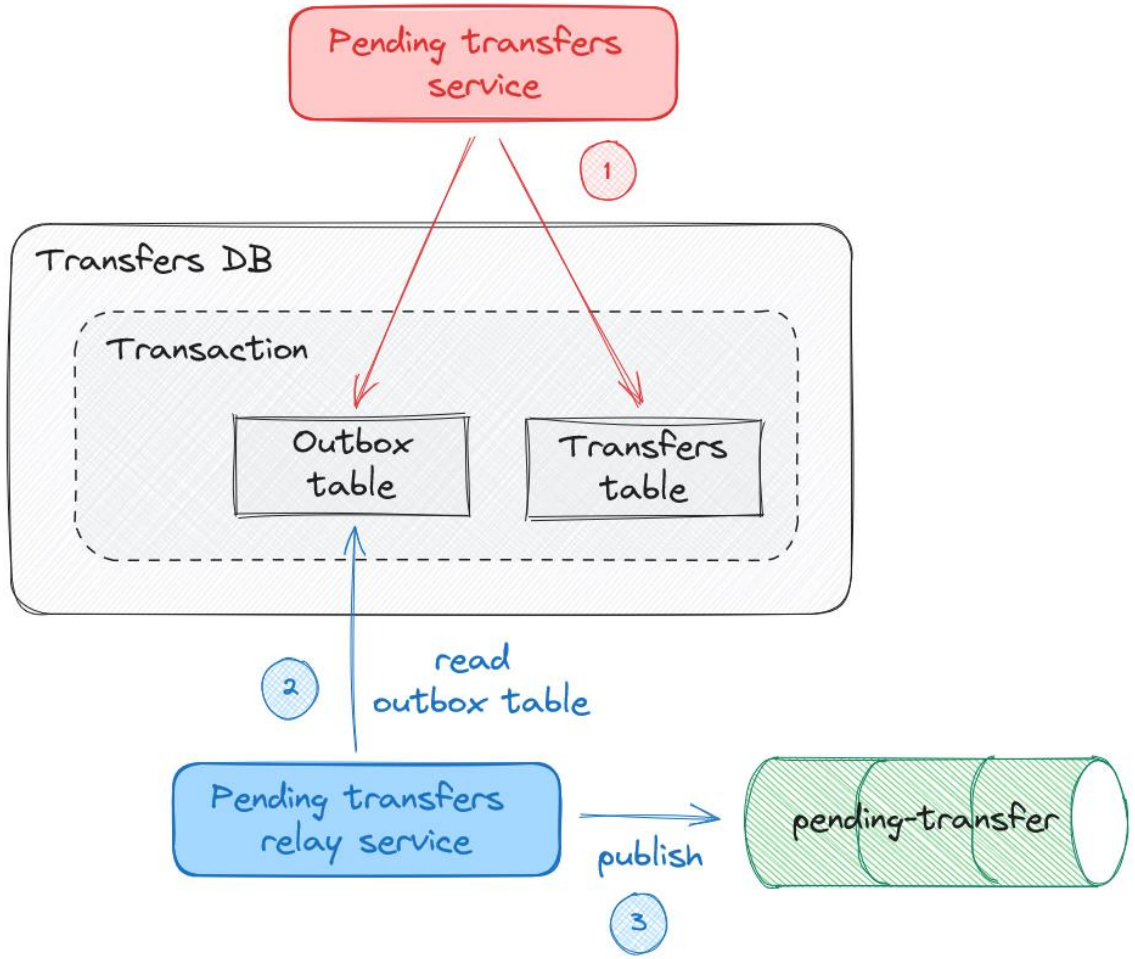
We want to update the database and send a message atomically in order to avoid data inconsistencies.

WHEN TO USE?

When two-phase commit is not possible or not desirable.

DISCUSSION

- With Spring, transaction synchronization can be used as an alternative
- Transaction synchronization uses the best effort 1-phase commit algorithm
- CDC is typically used for the relay service (e.g. with Debezium)



Saga pattern

Choreography-based

DESCRIPTION

We want to implement transactions that span multiple services. Messages are used to start, commit and rollback local transactions.

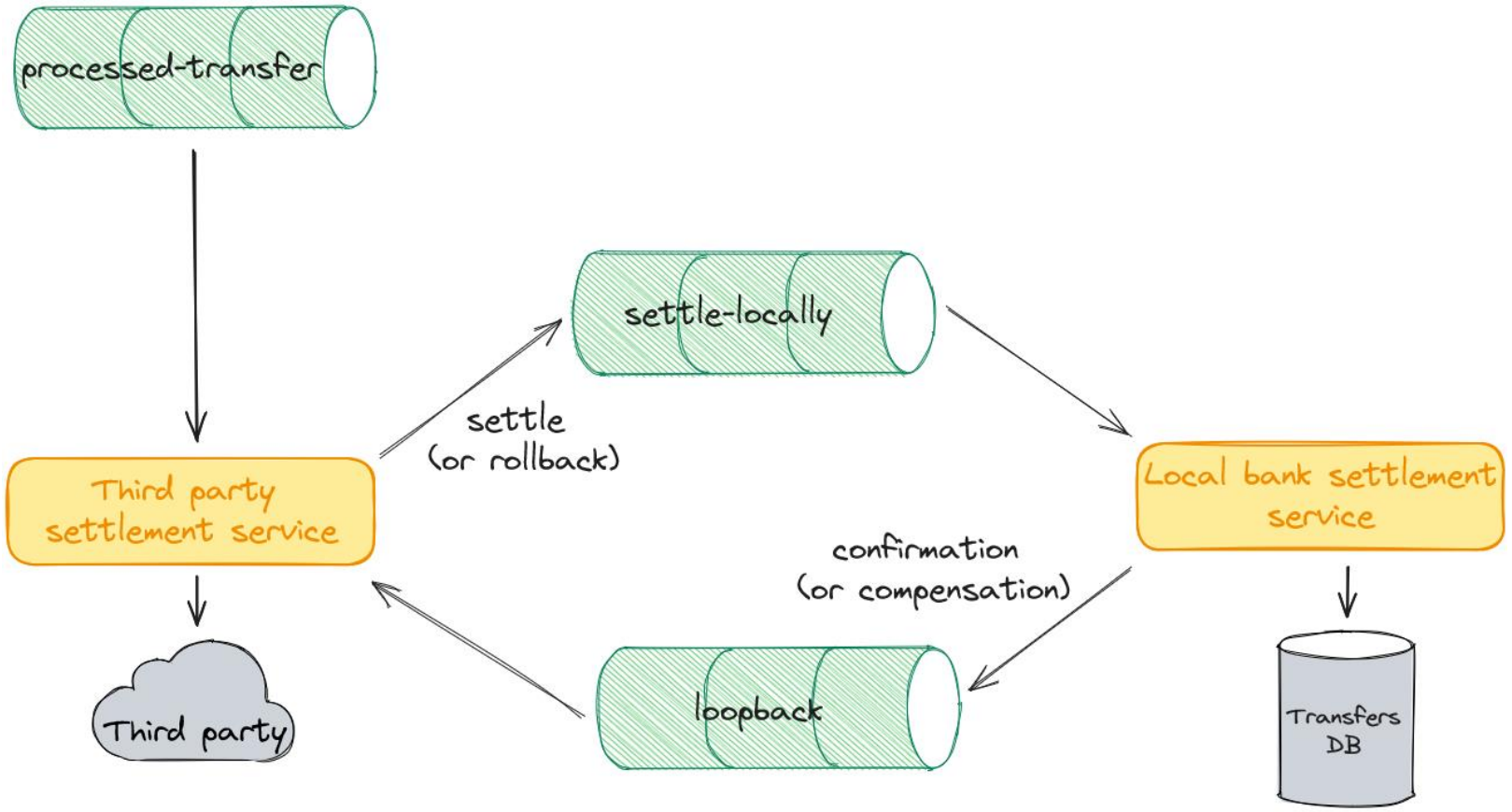
WHEN TO USE?

When Two-phase commit is not possible or not desirable.

DISCUSSION

We distinguish between:

- *Choreography-based* Saga: each local transaction publishes domain events that trigger local transactions in other services
- *Orchestration-based* Saga: an orchestrator coordinates the different local transactions




The code
https://github.com/cedric-schaller/kafka-error-handling

Any questions?

Cédric Schaller
Senior Architect
ELCA (Switzerland)

cedric.schaller@elca.ch
𝕏 @CedSoftEng