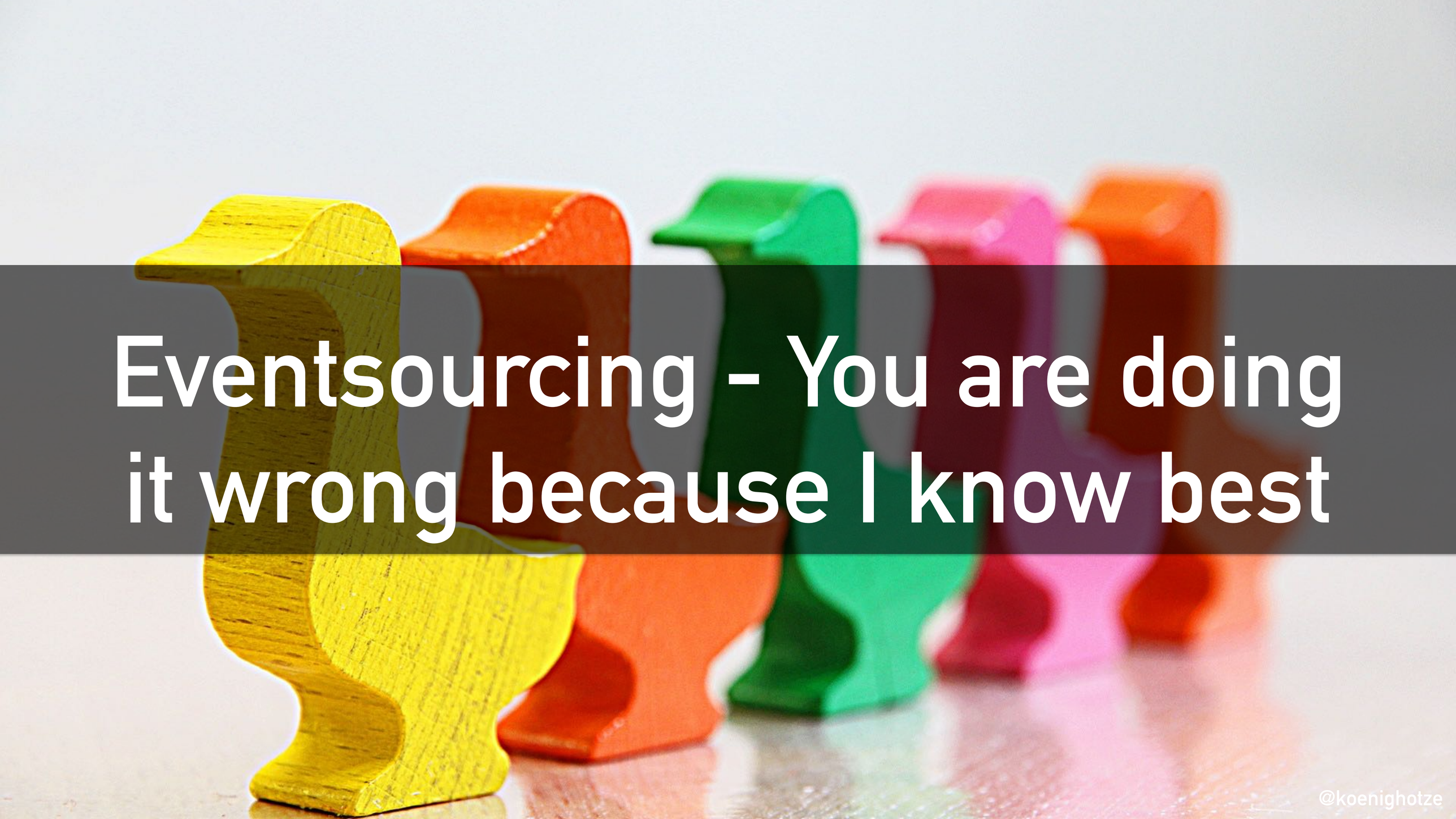
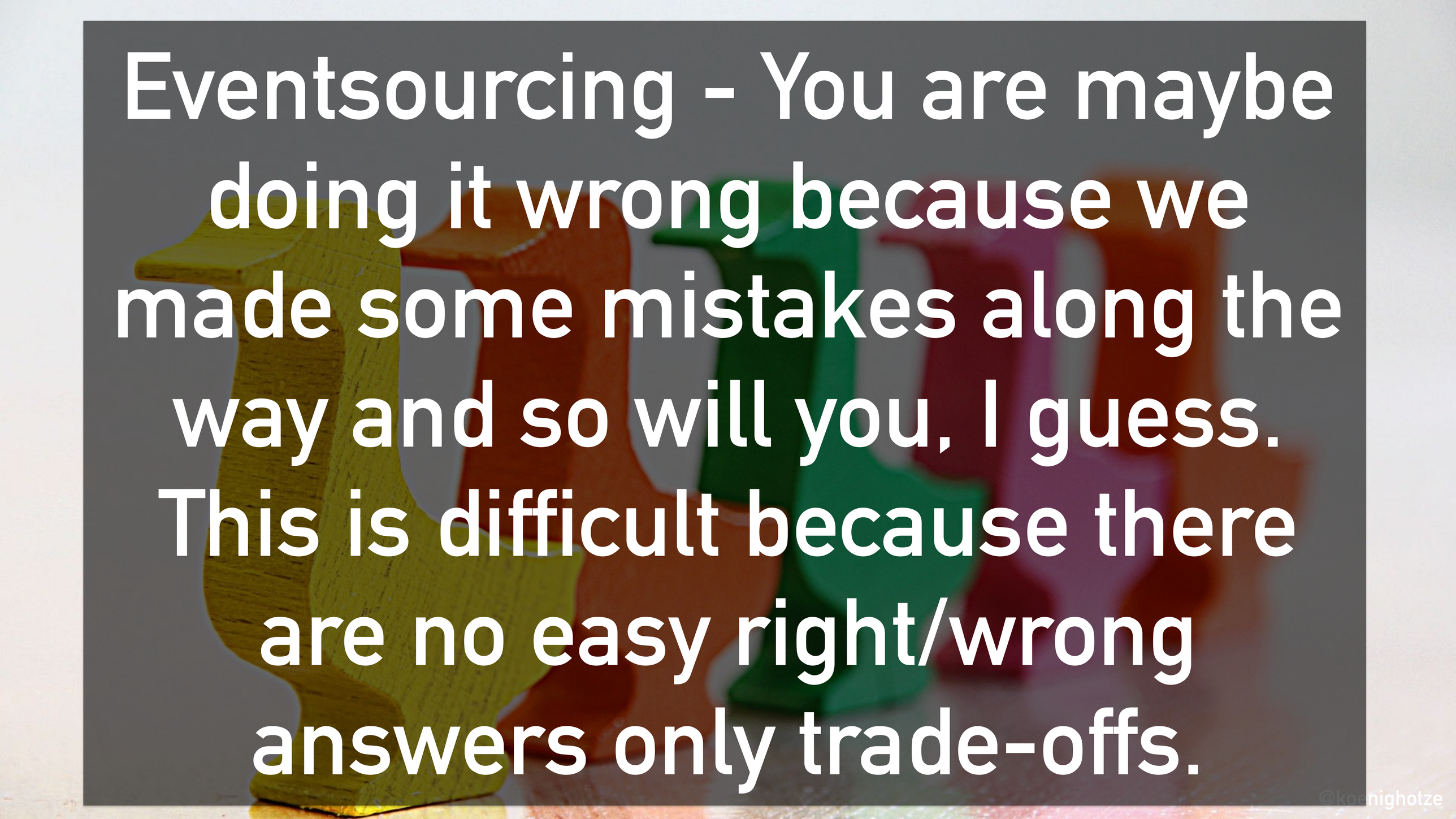

Eventsourcing - You are doing it wrong
@koenighotze

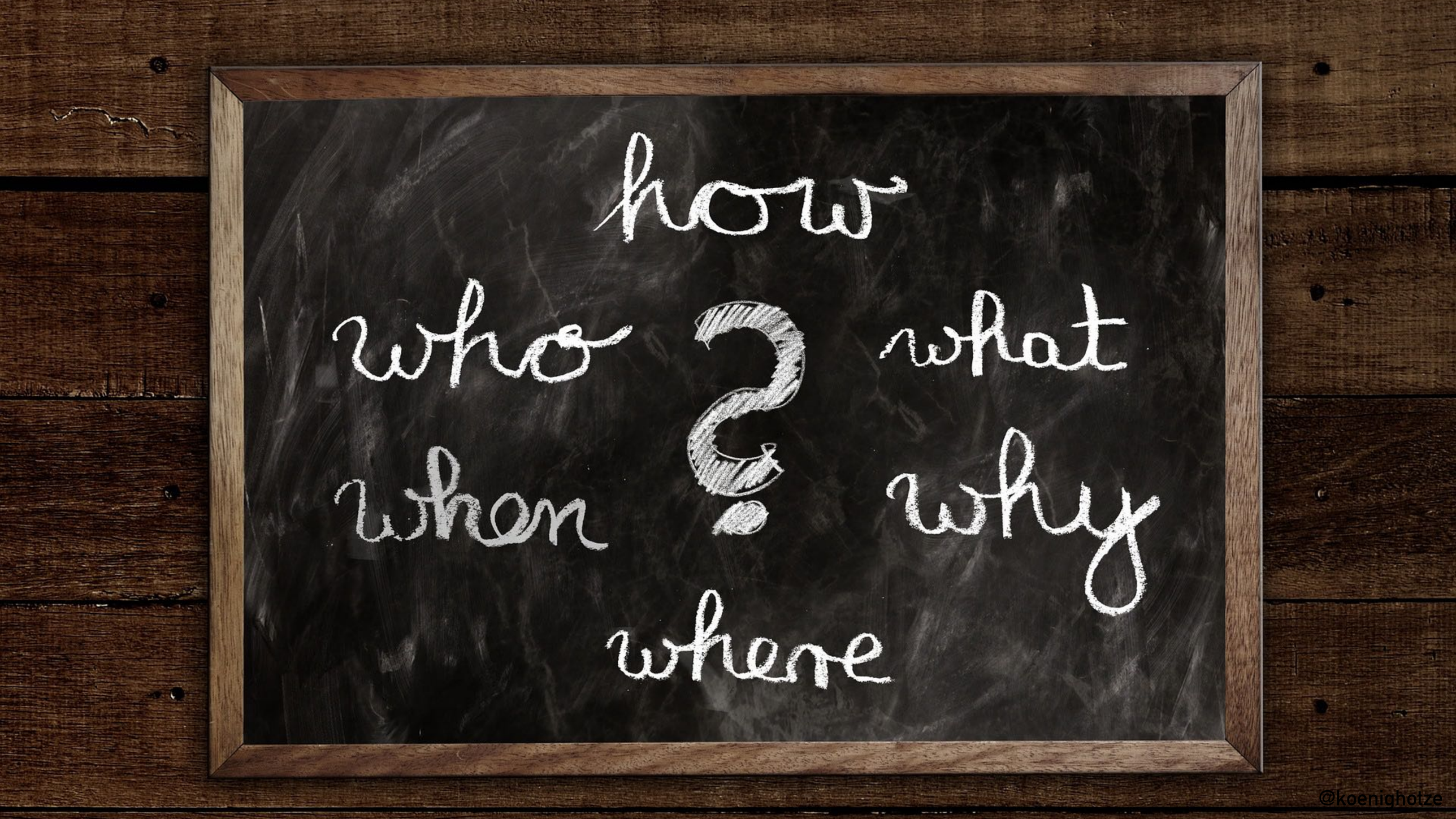

Eventsourcing - You are doing it wrong because I know best
@koenighotze


Eventsourcing - You are maybe doing it wrong because we made some mistakes along the way and so will you, I guess. This is difficult because there are no easy right/wrong answers only trade-offs.


Eventsourcing - You are
probably doing it wrong
@koenighotze


David Schmitz
@koenighotze
Senacor Technologies
#HASHTAG10
OLIVER'S HAIR
OLIVER'S HAIR
OLIVER'S HAIR
OLIVER'S HAIR

Are YOU building microservices?
Are YOU doing Domain Driven Design?
Are YOU applying eventsourcing?
Are YOU using Kafka as an eventstore?



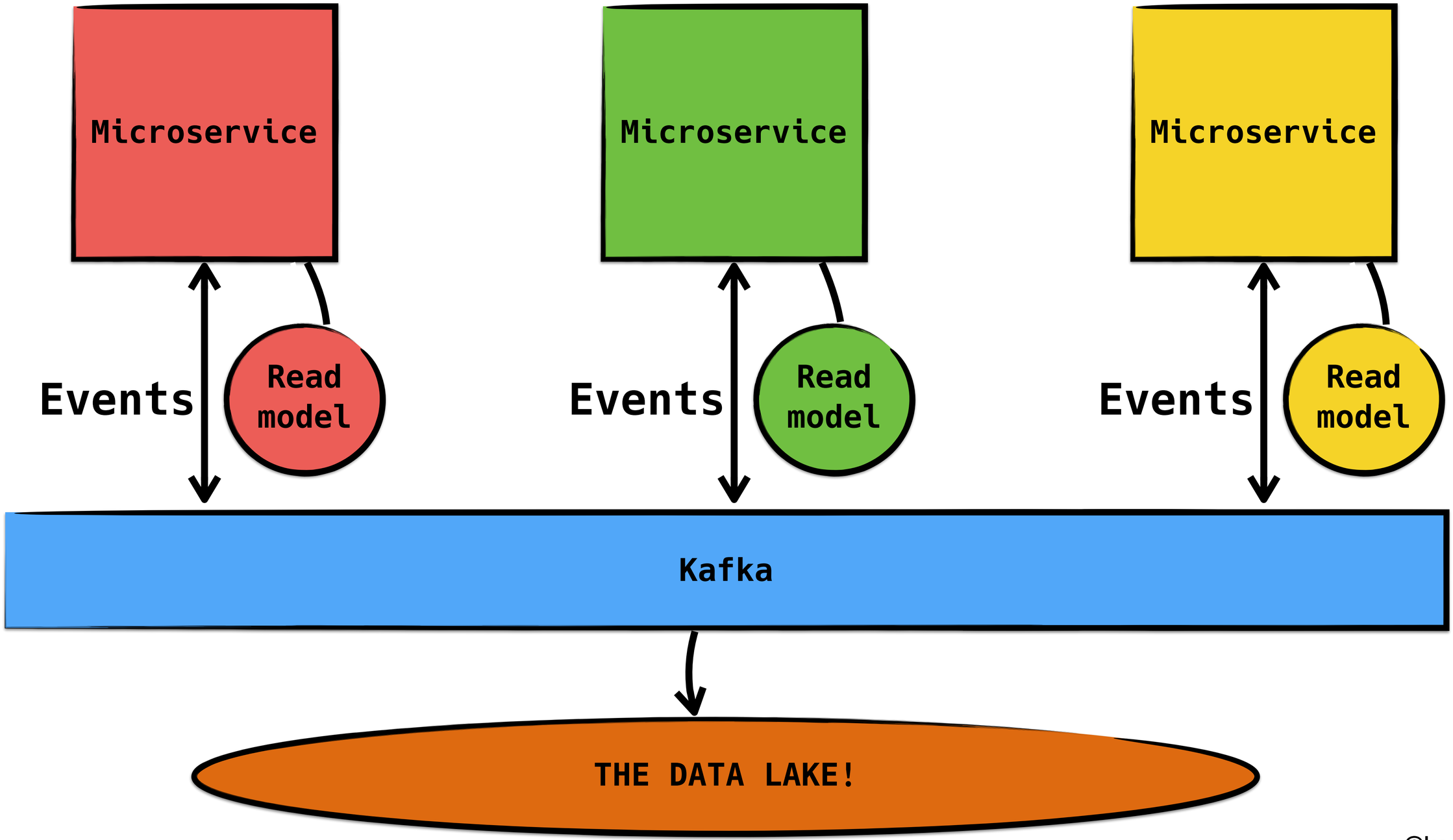
Typical misconceptions
Patterns “we” found useful
Traps to avoid
Not a Kafka-rant
What works for us, might not work for you and the other way around

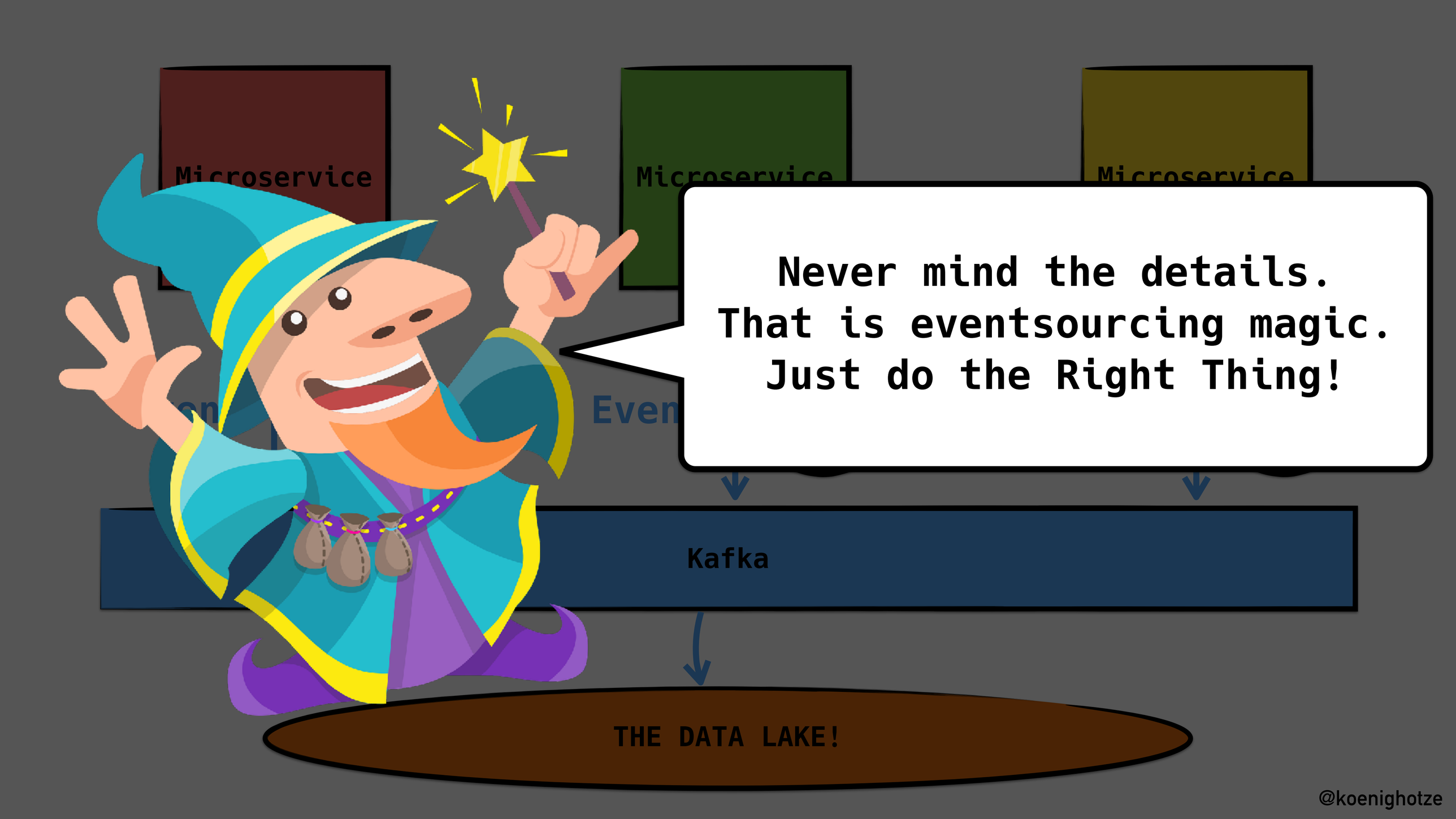
@koenighotze

Eventsourcing bootcamp

@koenighotze

Eventsourcing from an architects perspective


Microservice
Microservice
Microservice
Events
Read model
Events
Read model
Events
Read model
Kafka
THE DATA LAKE!

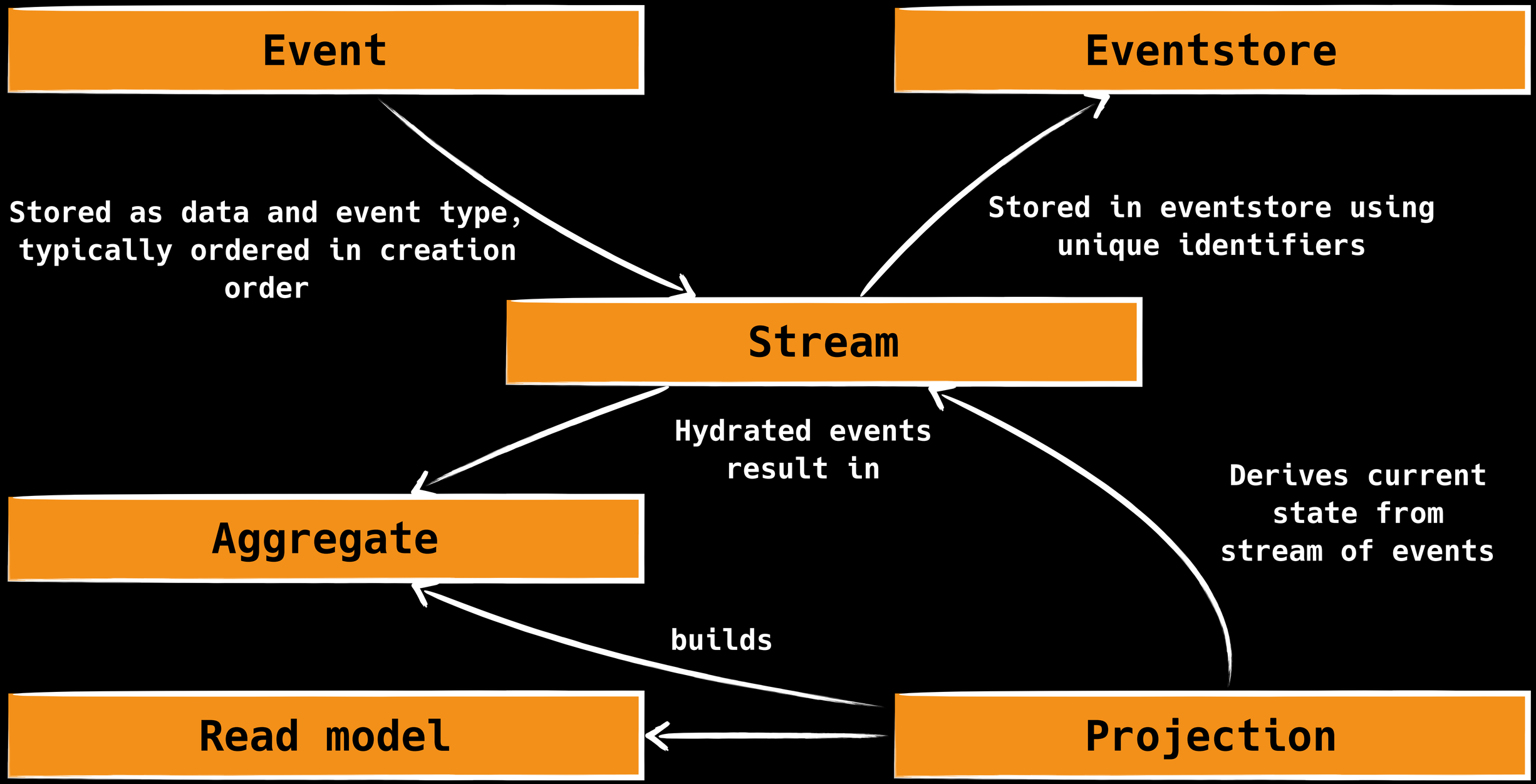

Microservice
Microservice
Microservice
Never mind the details.
That is eventsourcing magic.
Just do the Right Thing!
Kafka
THE DATA LAKE!

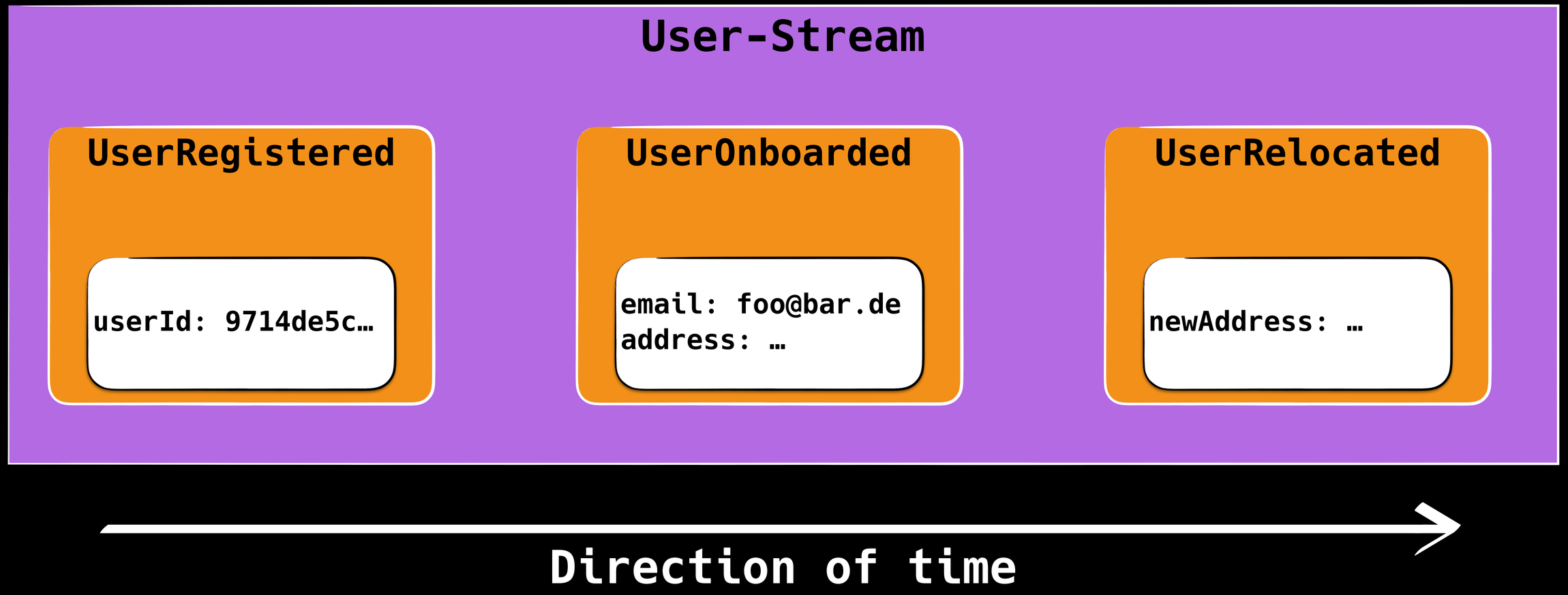
“Just” in Action

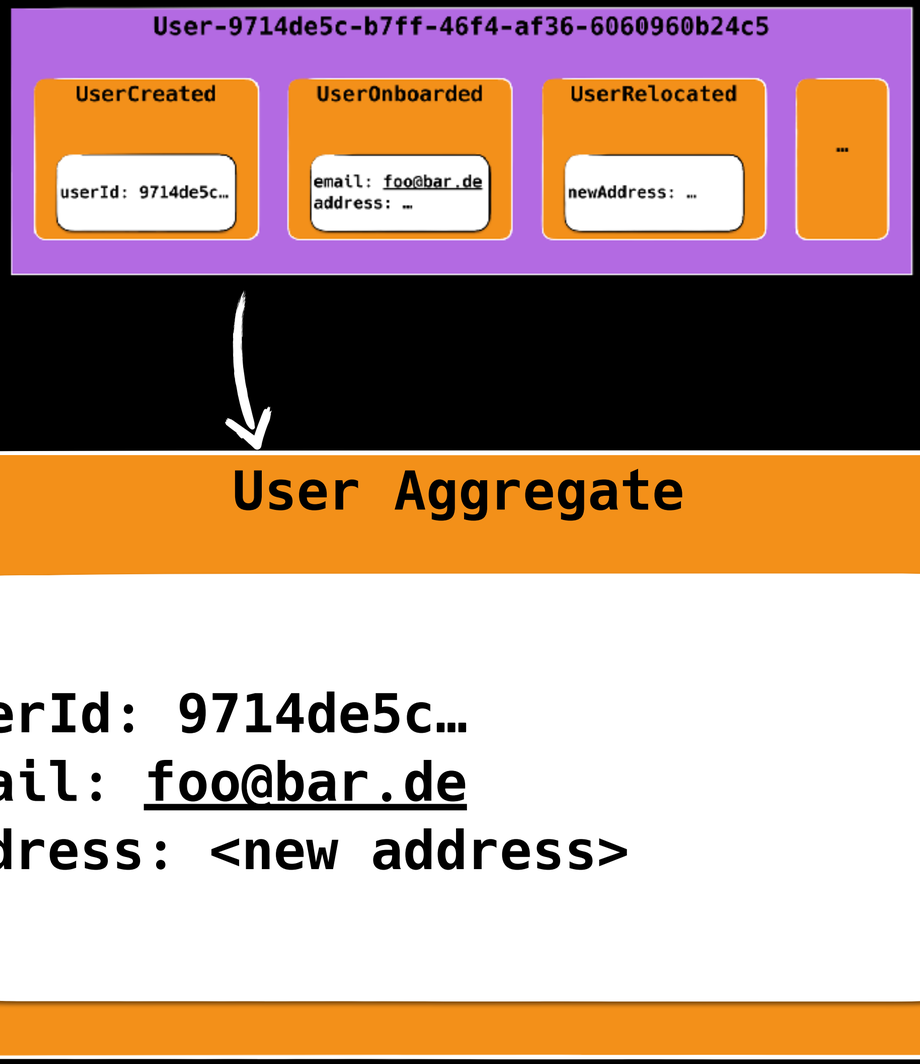
The idea in basic terms

Domain driven design
Event driven architecture
Distributed systems

Start with your domain
Find the bounded contexts
Find the root aggregates
Find events in your
ubiquitous language




User-Stream
UserRegistered
userId: 9714de5c…
UserOnboarded
email: foo@bar.de
address: …
UserRelocated
newAddress: …
Direction of time



Represents the user at the time of the last event read

Eventsourcing helps answering the question of dealing with data in distributed systems in a scalable way.

It makes the dynamics of your systems explicit as first class concepts

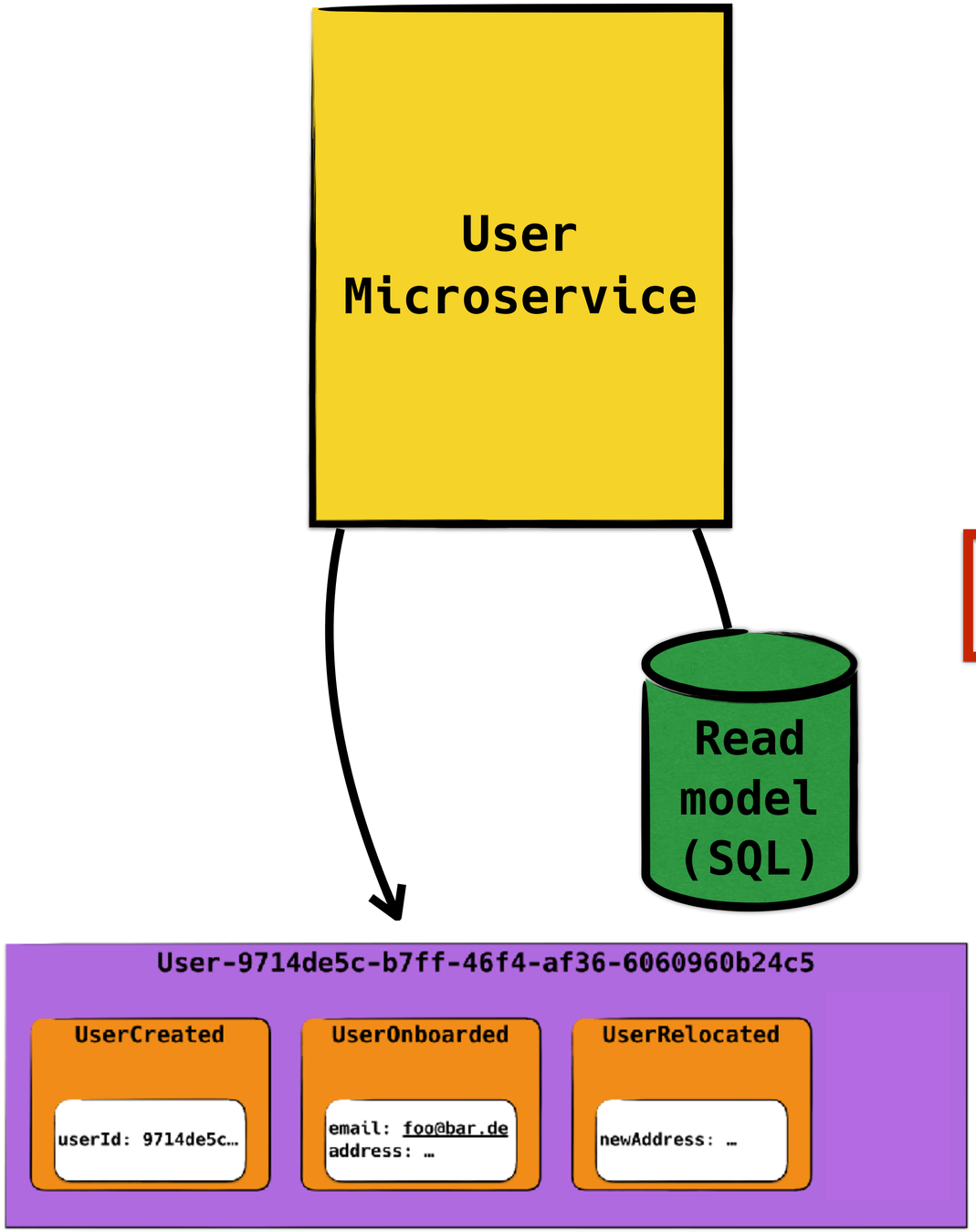

Read models
…and why you may not need to them (initially)
@koenighotze

Read models are local hydrates of the events stored in specific streams

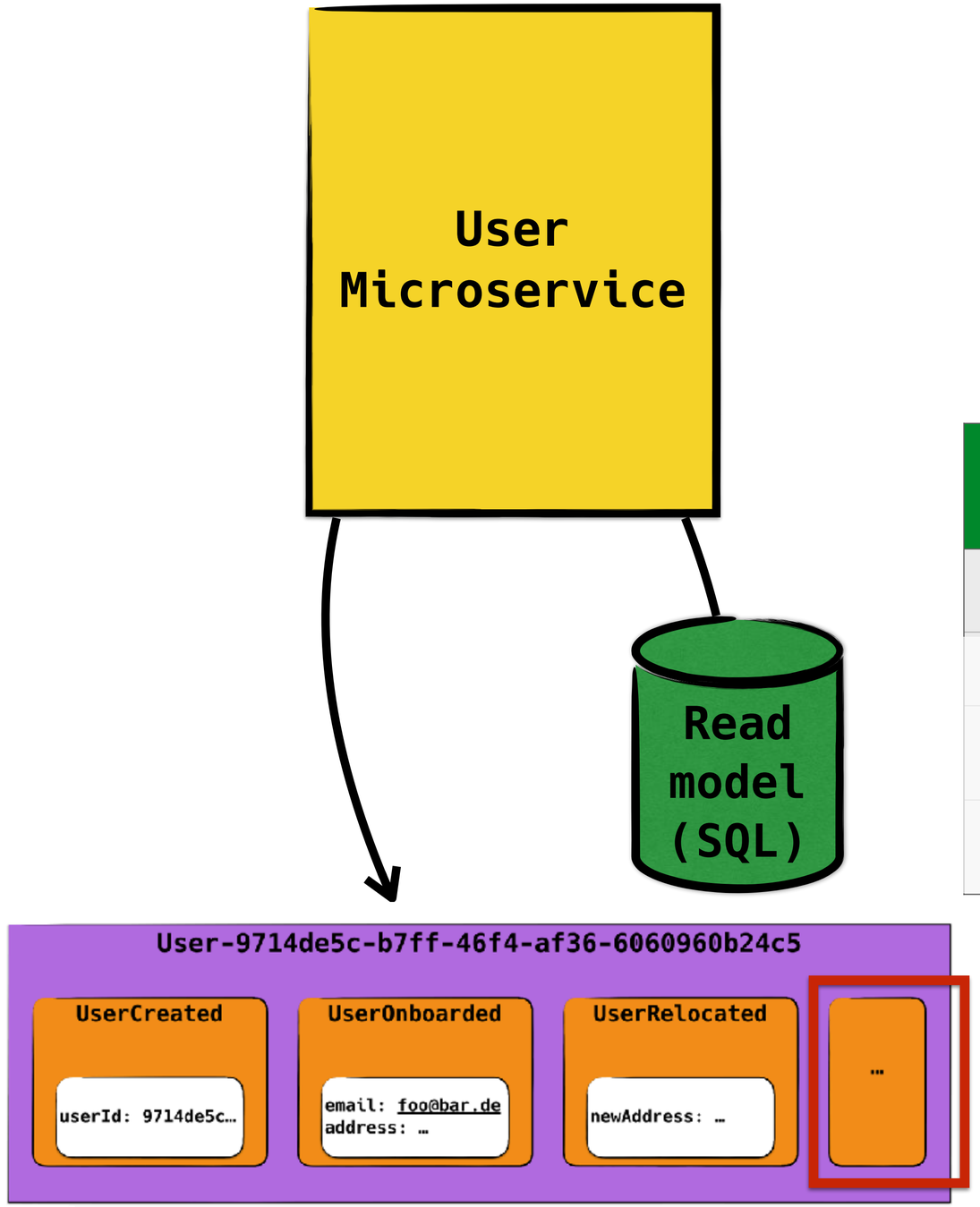
```
readModel = Stream.of(events)
                  .leftFold(handlers)
```

How can we handle read models?

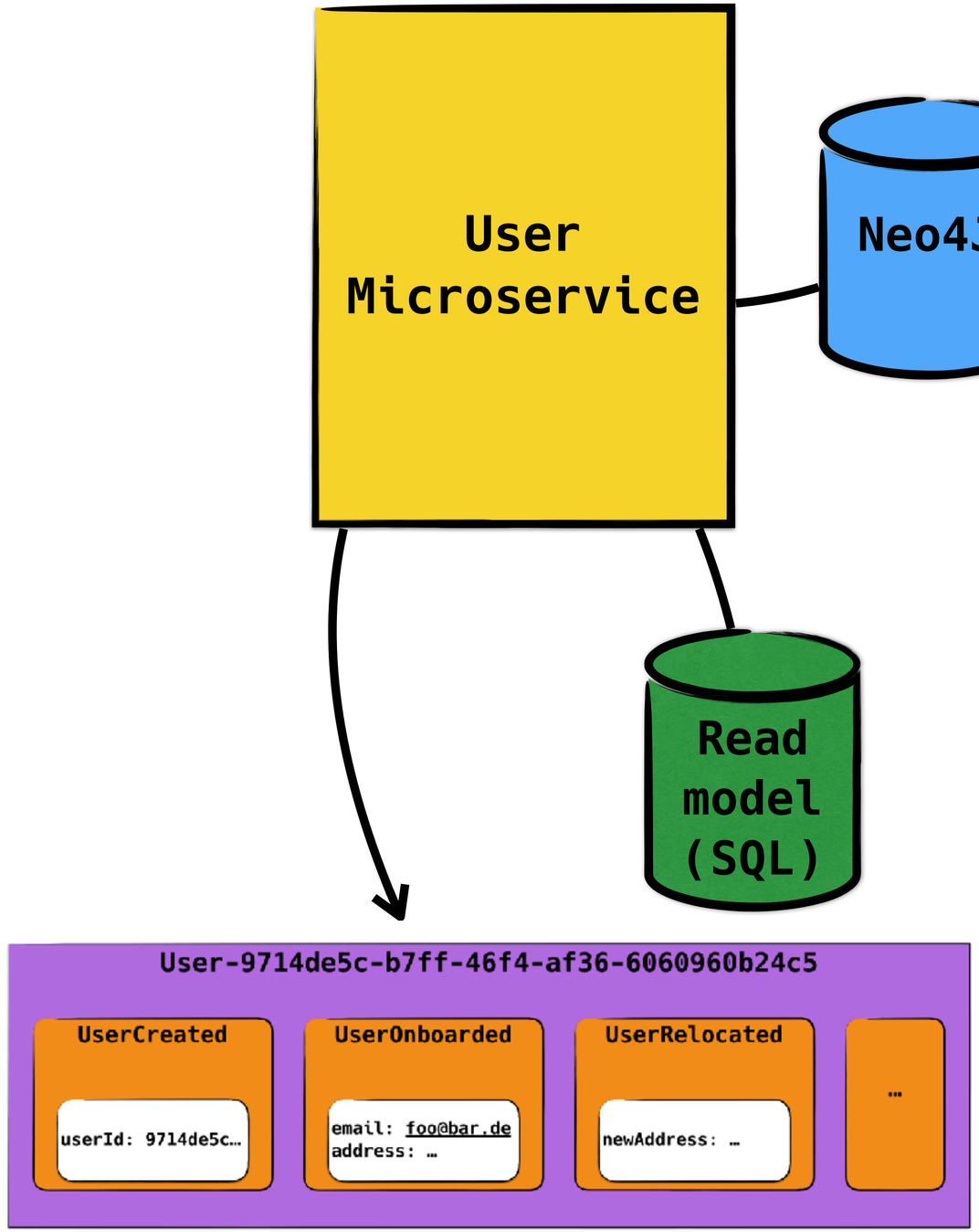
“Just” use a local database



user id	last event id	user name	email	street
9741…	3	David	foo@bar.de	…
4532…	7	Martin	null	…



user id	last event id	user name	email	street
9741…	**4**	David	**qux@ba.ze**	…
4532…	7	Martin	null	…


User
Microservice
Neo4J
Read
model
(SQL)
User-9714de5c-b7ff-46f4-af36-6060960b24c5
UserCreated
userId: 9714de5c…
UserOnboarded
email: foo@bar.de
address: …
UserRelocated
newAddress: …
…

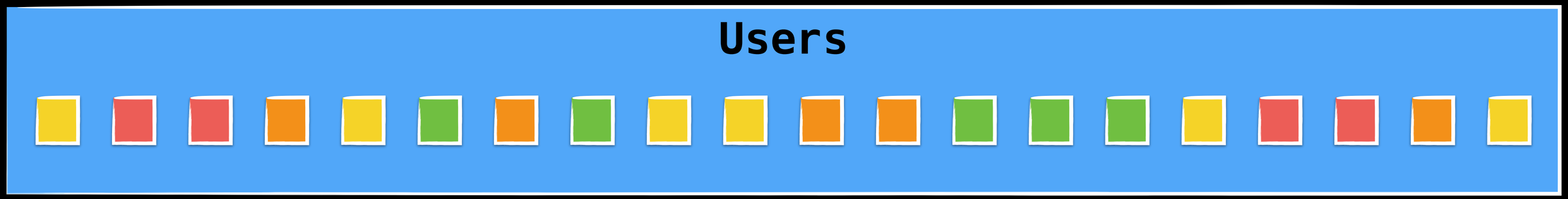
Challenges?

Eventual consistency
Replays
Re-deliveries
Operational complexity

You may not need a read model

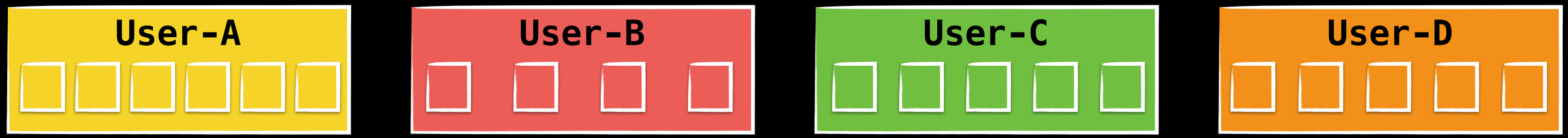
Typical strategies for storing events

Maybe all events of an aggregate type in a single stream?

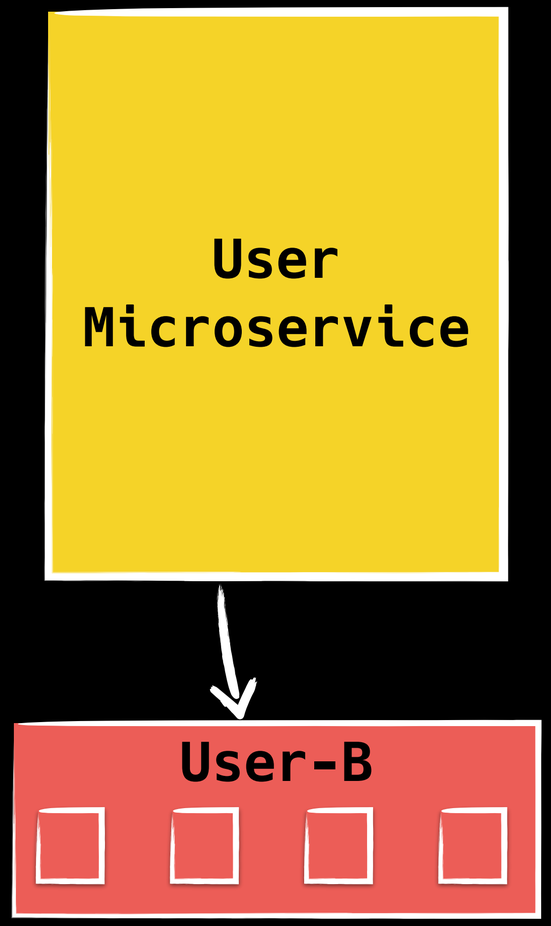


User A User B User C User D

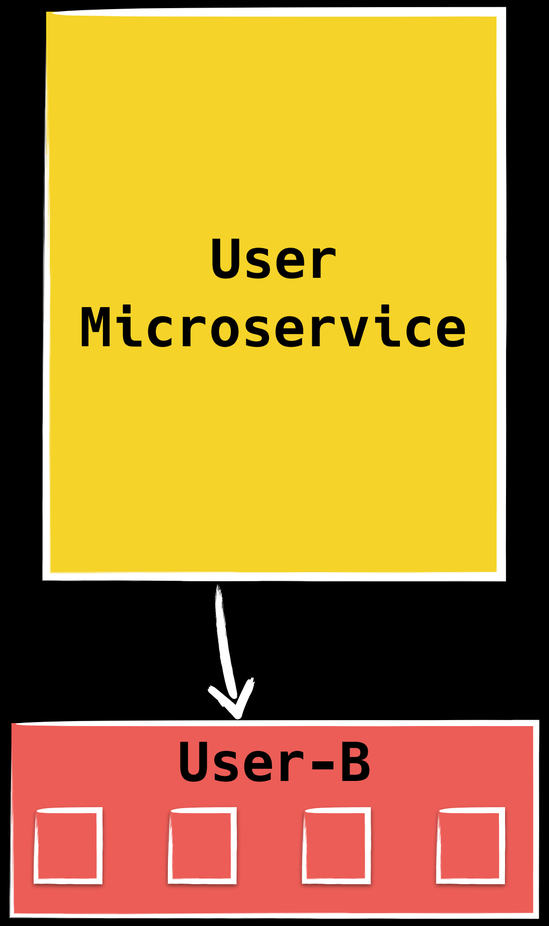
Better: One stream per aggregate



GET /users/B



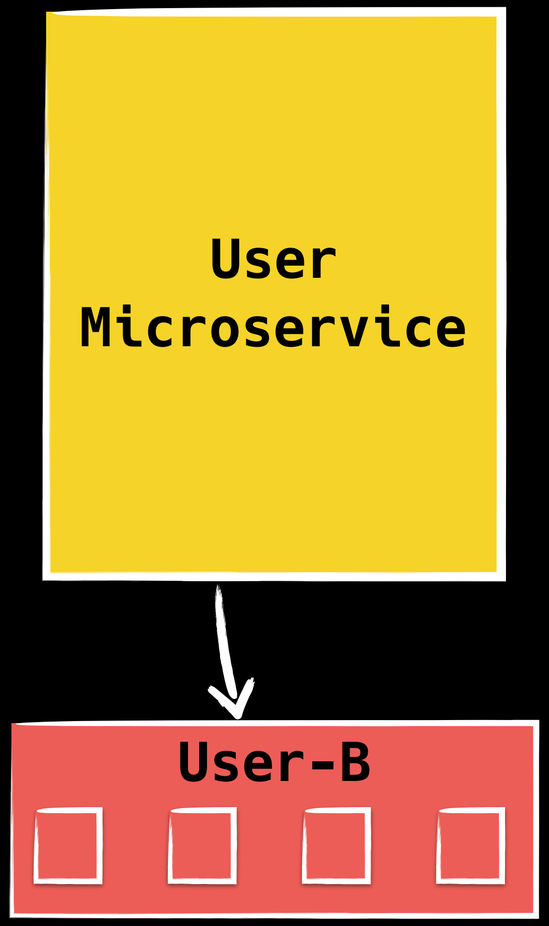
GET /users/B




```
handlers = {
  'UserCreated':   (current, event) => {},
  'UserOnboarded': (current, event) => {},
  'UserDeleted': ...
}
```

```
aggregate = readAggregateFromStream(
  'user',
  'B',
  {},
  fromStartOfStream,
  handlers
)
```

GET /users/B



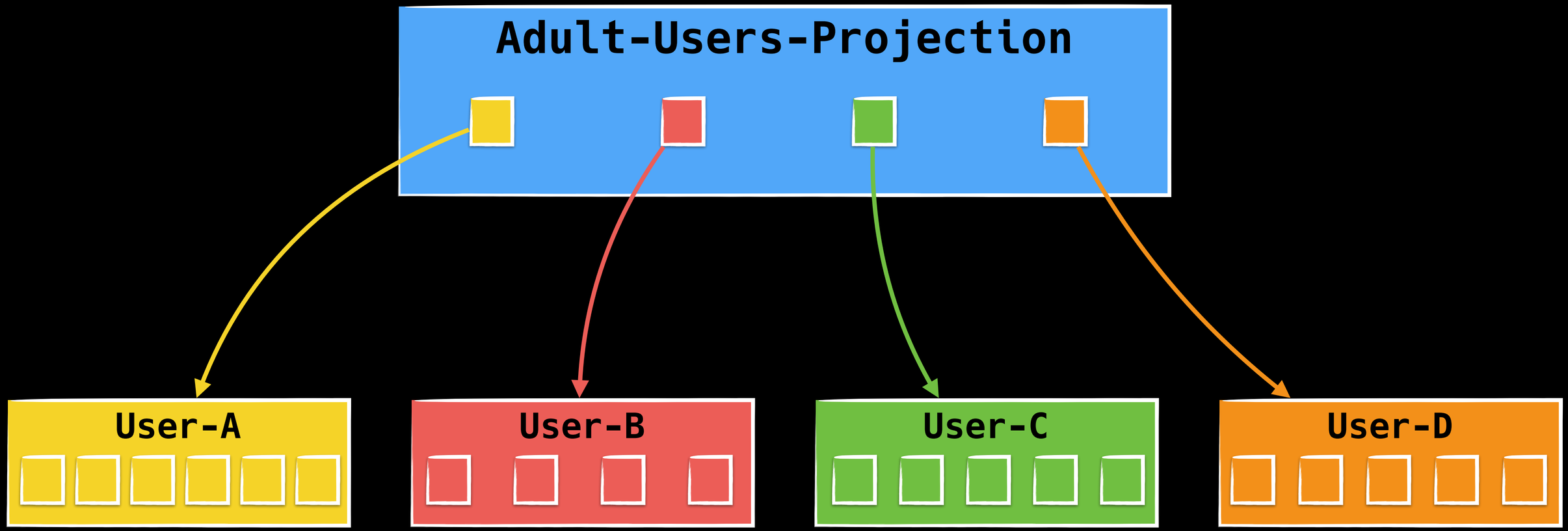

```
handlers = {
  'UserCreated':   (current, event) => {},
  'UserOnboarded': (current, event) => {},
  'UserDeleted': ...
}
```

```
aggregate = readAggregateFromStream(
  'user',
  'B',
  {},
  fromStartOfStream,
  handlers
)
```

Consistent read
No operational overhead
Super-simple programming logic

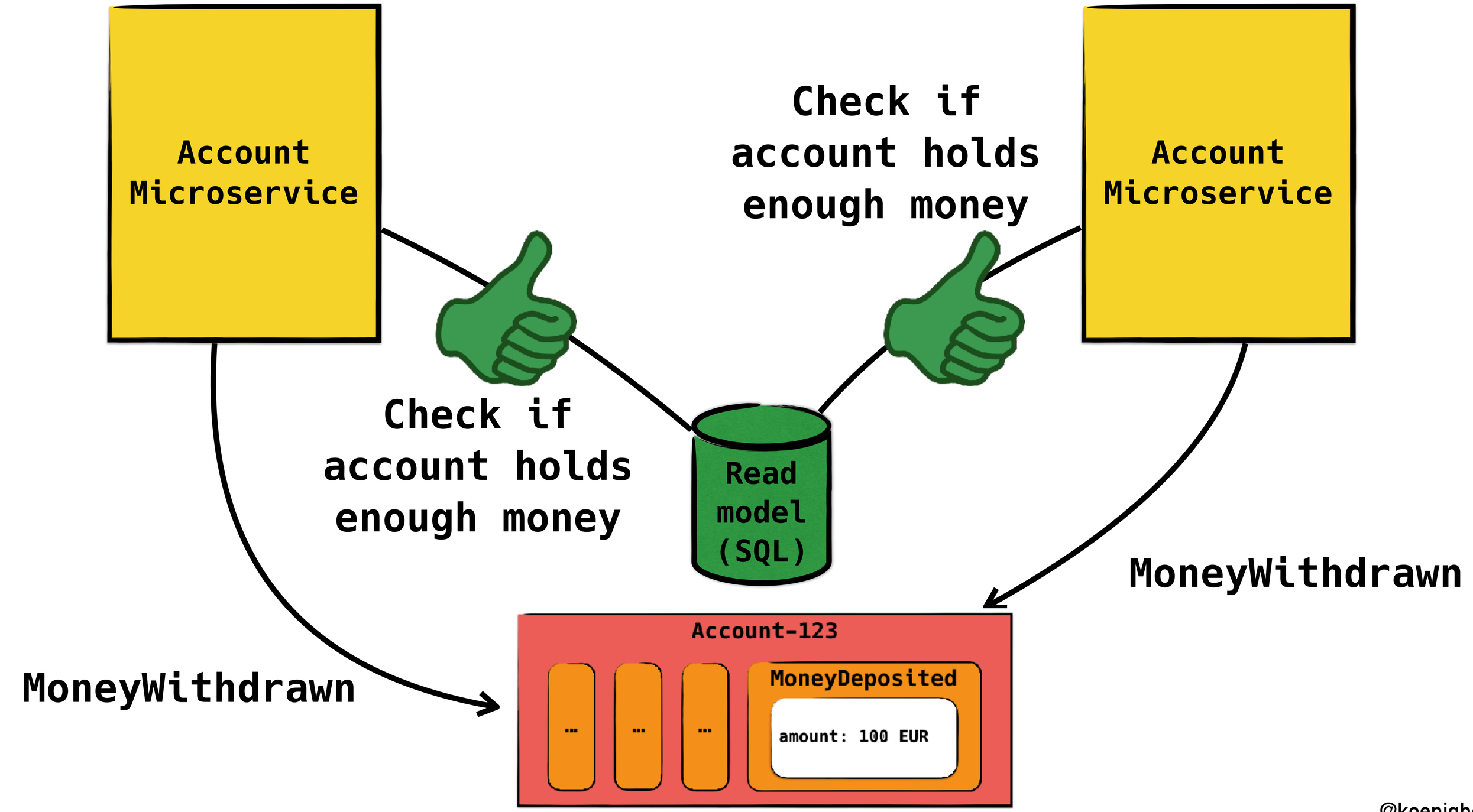
But, what about speed?

What about queries like ‘all users with age > 18’?


Adult-Users-Projection
User-A
User-B
User-C
User-D

But I can use a readmodel for fast validation, can I?

Maybe not



Validation against a read model is prone to inconsistencies

Prefer validating against the eventstore itself

Most event handlers are neither CPU or IO intensive

Prefer small aggregates, if it makes sense in your domain

Measure and introduce persistent read models only if needed

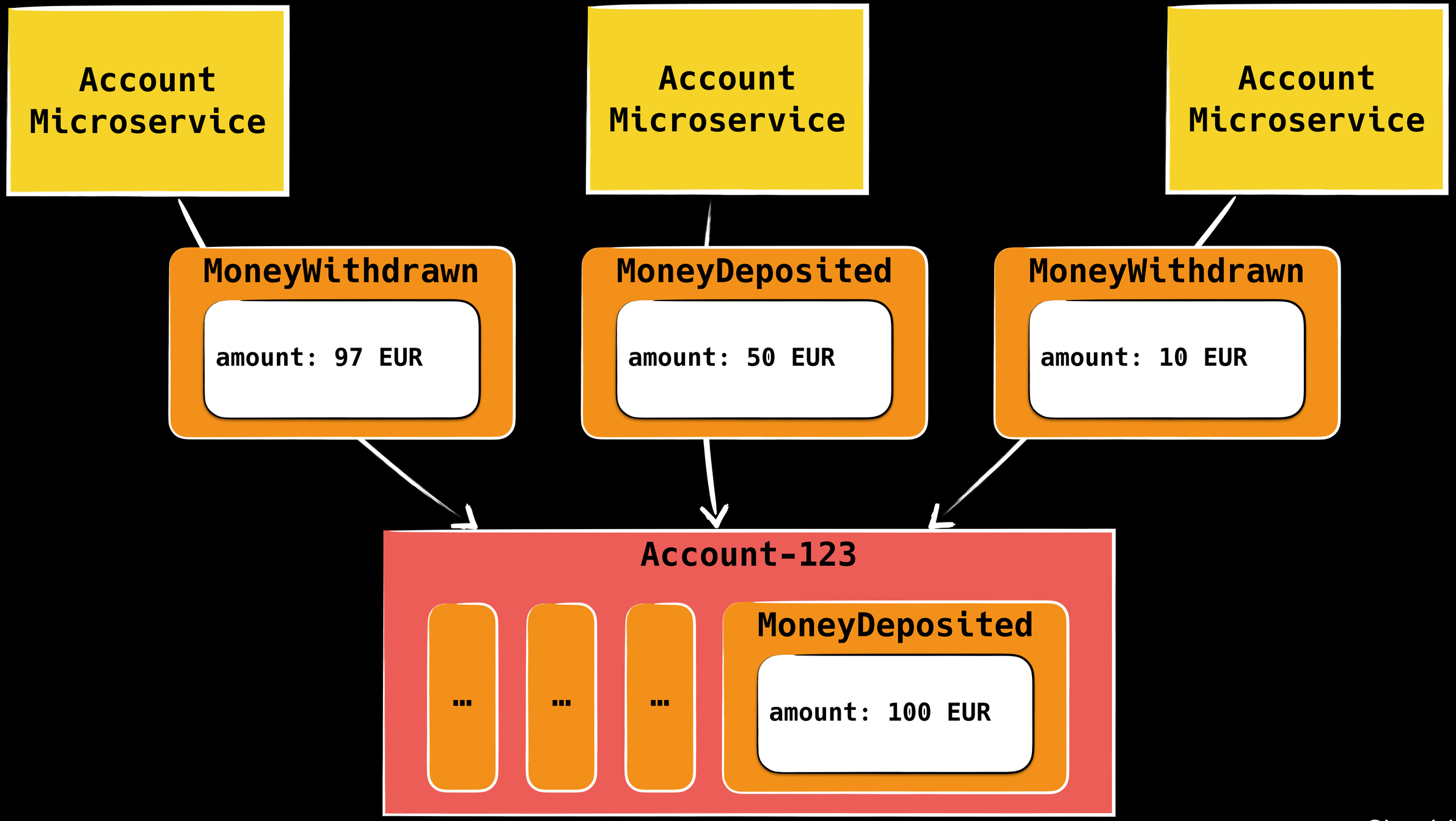
Transactions, concurrency and your eventstore

How can we guarantee correctness when writing?

“Only withdraw money, if the bank account holds enough money!”*

*Actually, a real bank would not want such a business rule. They earn money if you overdraw your account. An overdraft fee is one of the most expensive fees banks charge. Just saying…

Are YOUR systems single-threaded?



The correctness of the business result depends on the order of events

MoneyDeposited

amount: 100 EUR

100 EUR

MoneyWithdrawn

amount: 10 EUR

90 EUR

~~MoneyWithdrawn~~

~~amount: 97 EUR~~

-7 EUR

MoneyDeposited

amount: 50 EUR

140 EUR

Let's shuffle

MoneyDeposited

amount: 100 EUR

100 EUR

MoneyWithdrawn

amount: 10 EUR

90 EUR

MoneyWithdrawn

amount: 97 EUR

140 EUR

MoneyDeposited

amount: 50 EUR

43 EUR

The aggregate is responsible for enforcing business invariants

MoneyWithdrawn

amount: 50 EUR

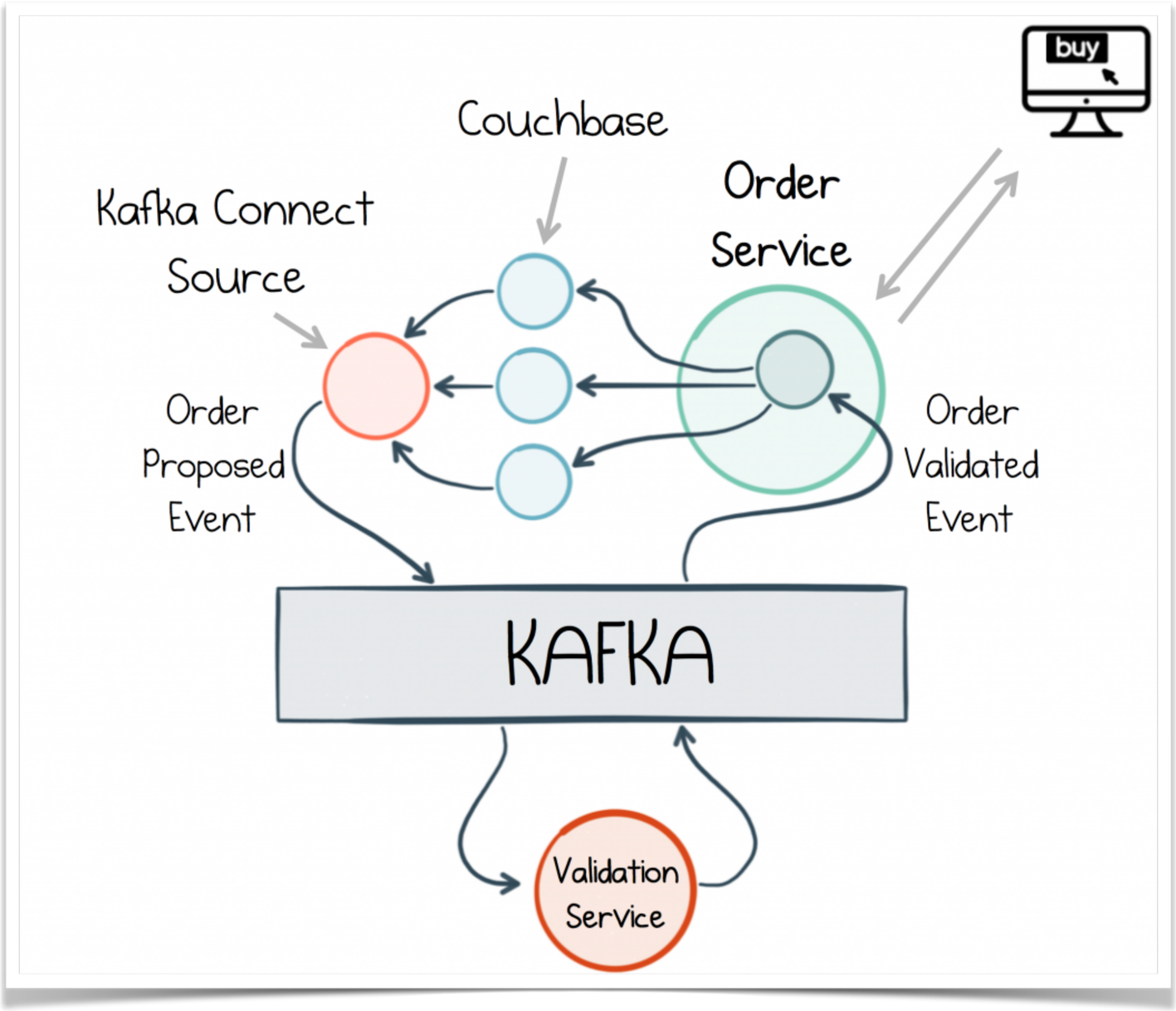
I want in!

Account-123

Account balanc
30 EUR

Not if you mess up my internal business rules!

"Just" add a database in front of your eventstore!



https://www.confluent.io/blog/messaging-single-source-truth/

Quick tip for finding friends in ops:

Ask them to “just” install and maintain production ready Kafka and Couchbase installations in the Cloud

Optimistic concurrency control

From Wikipedia, the free encyclopedia

Optimistic concurrency control (**OCC**) is a concurrency control method applied to transactional systems such as relational database management systems and software transactional memory. OCC assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.[1] Optimistic concurrency control was first proposed by H.T. Kung and John T. Robinson.[2]

https://en.wikipedia.org/wiki/Optimistic_concurrency_control

Optimistic concurrency control

From Wikipedia, the free encyclopedia

OCC assumes that multiple transactions can frequently complete without interfering with each other.

transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.[1] Optimistic concurrency control was first proposed by H.T. Kung and John T. Robinson.[2]

https://en.wikipedia.org/wiki/Optimistic_concurrency_control

Optimistic concurrency control

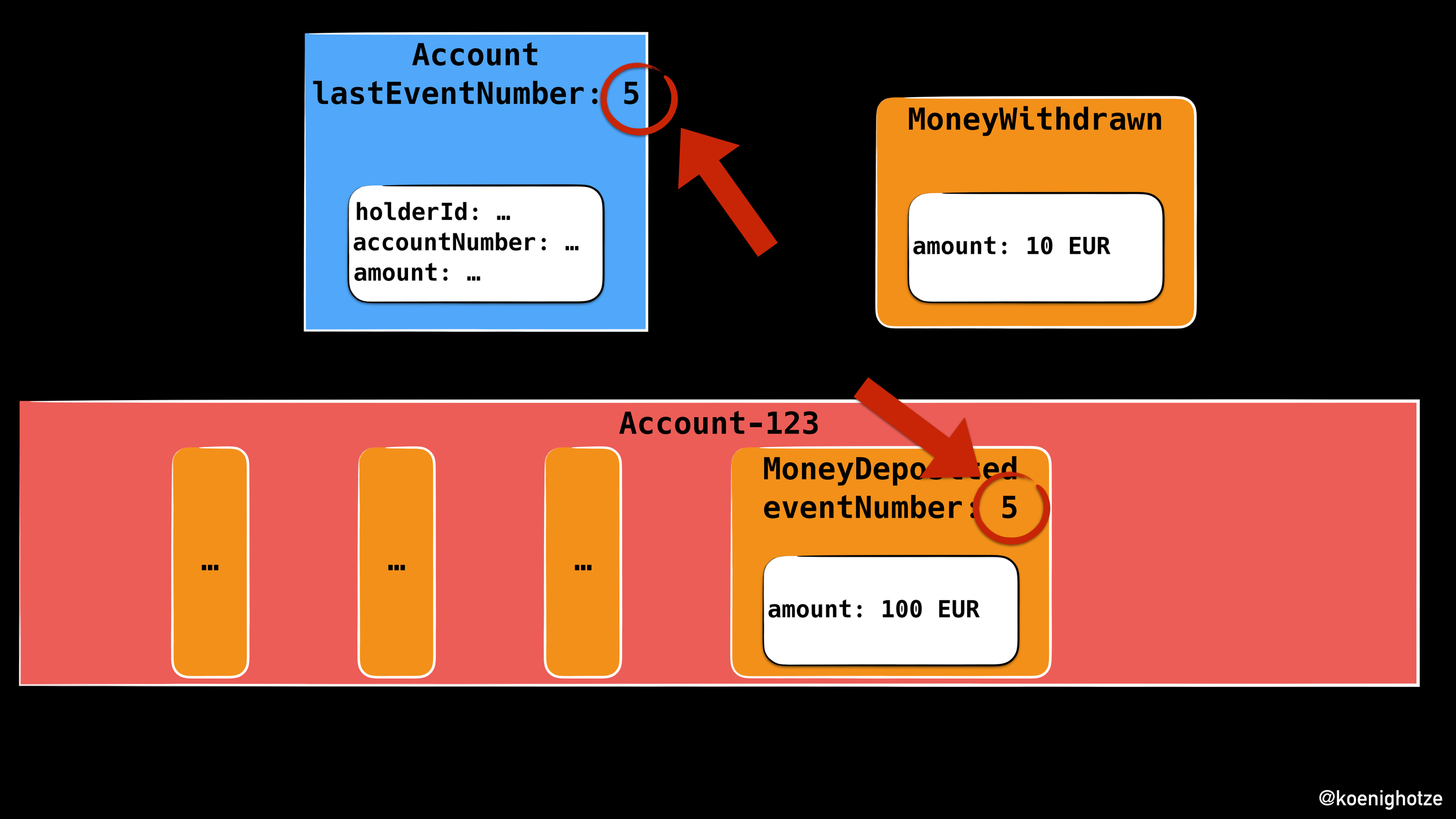
From Wikipedia, the free encyclopedia

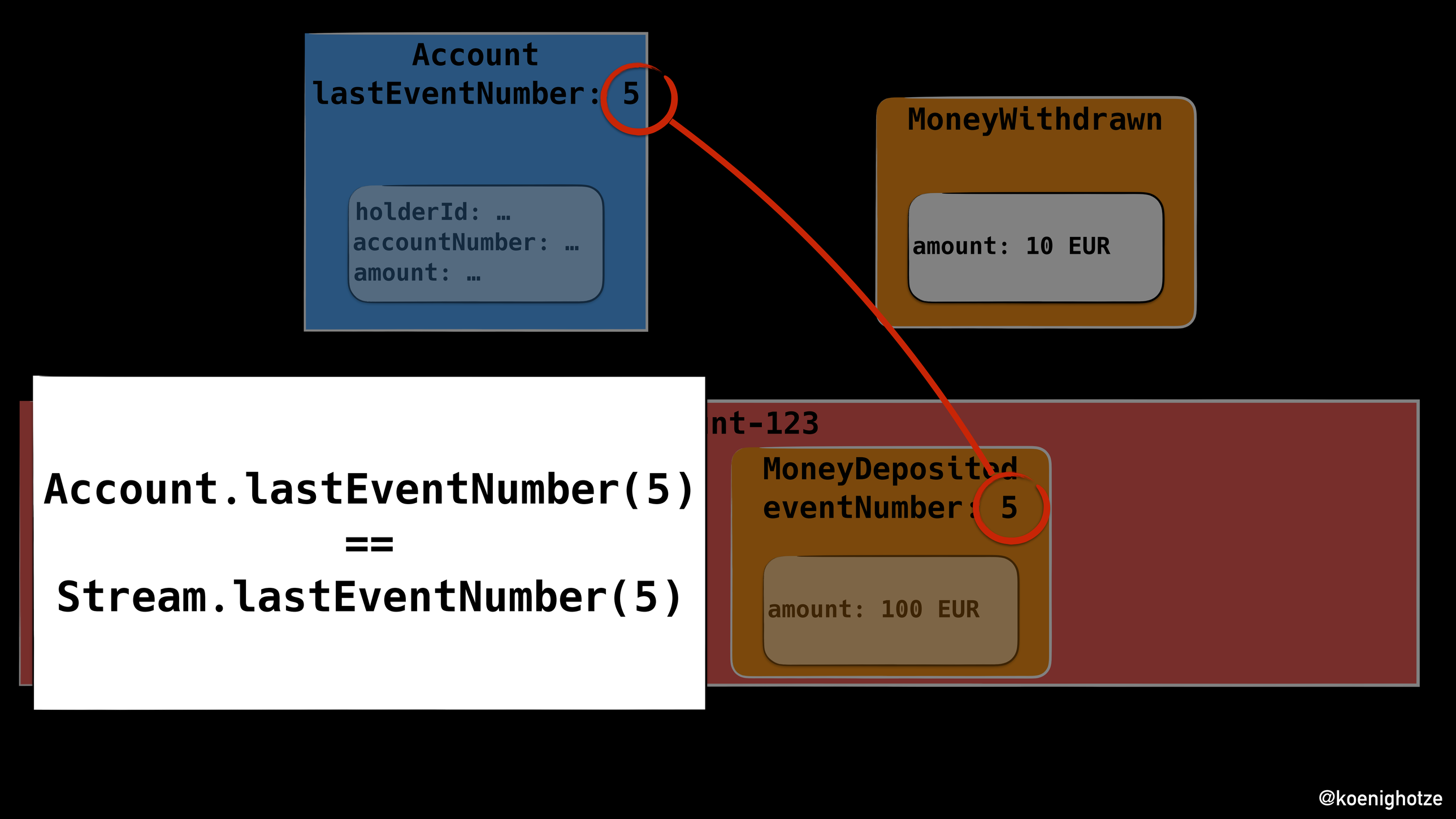
…each transaction verifies that no other transaction has modified the data it has read…

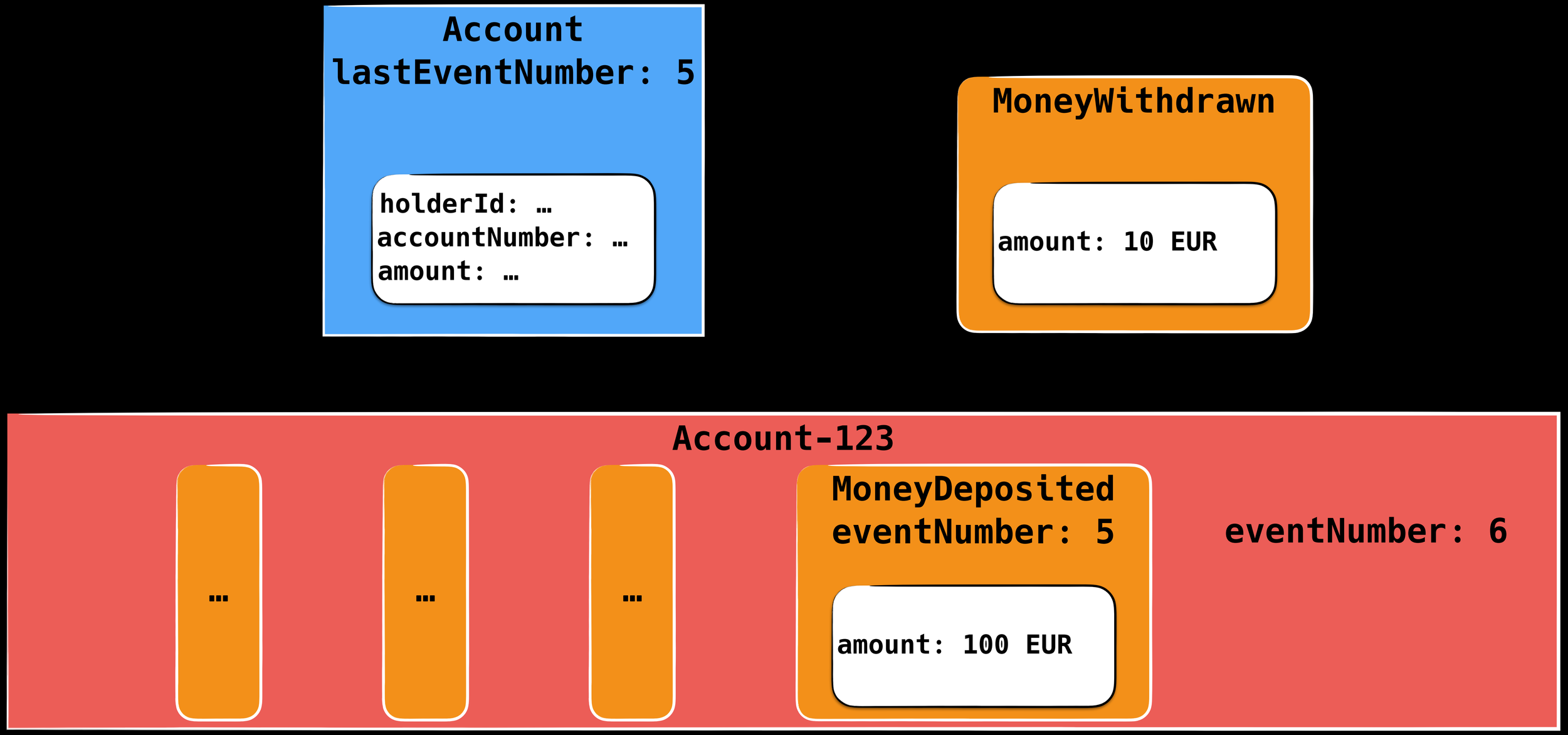
transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.[1] Optimistic concurrency control was first proposed by H.T. Kung and John T. Robinson.[2]

https://en.wikipedia.org/wiki/Optimistic_concurrency_control

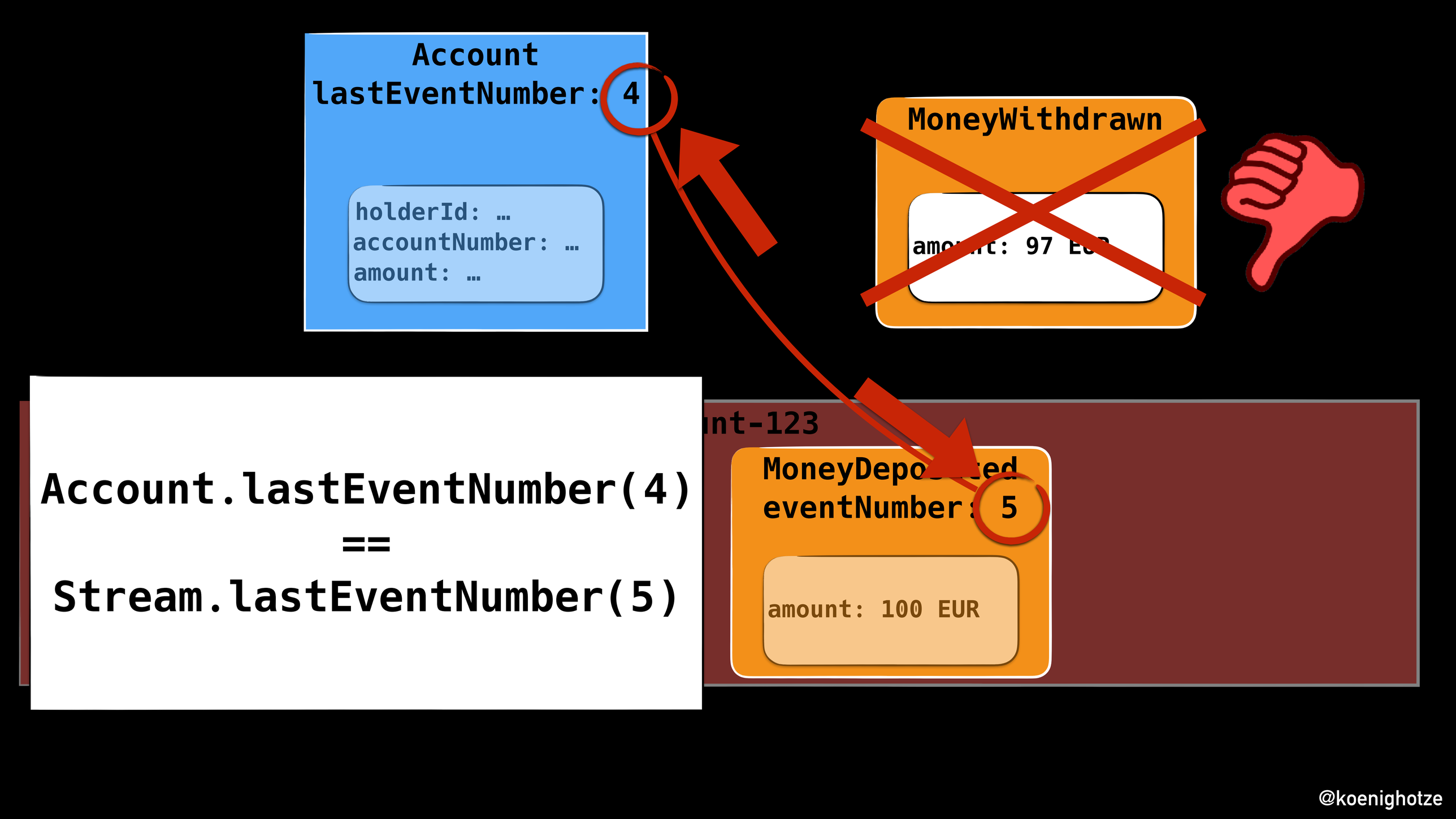
The happy path


Account
lastEventNumber: 5
holderId: …
accountNumber: …
amount: …
MoneyWithdrawn
amount: 10 EUR
Account-123
…
…
…
eventNumber: 5
amount: 100 EUR

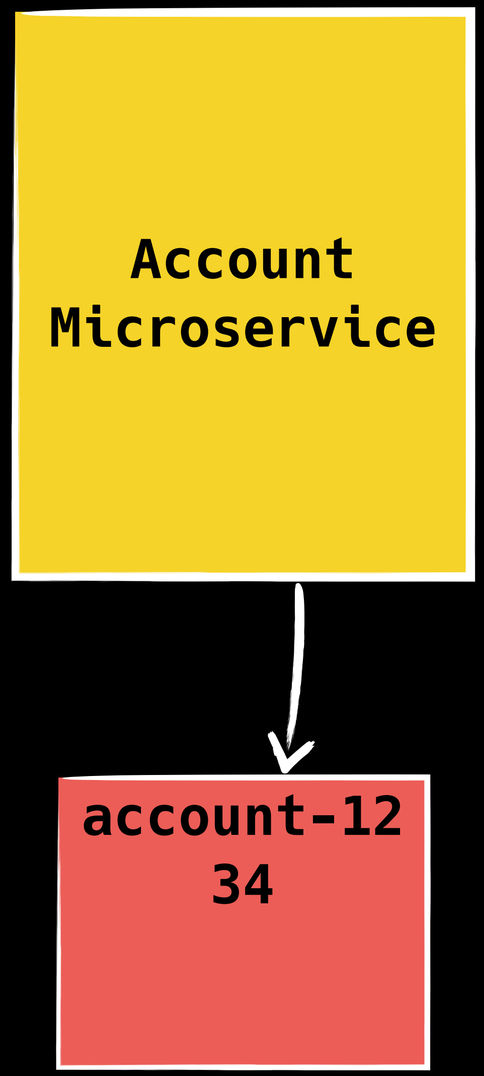

Account
lastEventNumber: 5
holderId: …
accountNumber: …
amount: …
MoneyWithdrawn
amount: 10 EUR
nt-123
MoneyDeposited
eventNumber: 5
amount: 100 EUR
Account.lastEventNumber(5)
==
Stream.lastEventNumber(5)


Account
lastEventNumber: 5
holderId: …
accountNumber: …
amount: …
MoneyWithdrawn
amount: 10 EUR
Account-123
…
…
…
MoneyDeposited
eventNumber: 5
amount: 100 EUR
eventNumber: 6

The not-so-happy path


Account
lastEventNumber: 4
holderId: …
accountNumber: …
amount: …
MoneyWithdrawn
amount: 97 EUR
Account.lastEventNumber(4)
==
Stream.lastEventNumber(5)
nt-123
MoneyDeposited
eventNumber: 5
amount: 100 EUR

PUT /account/1234

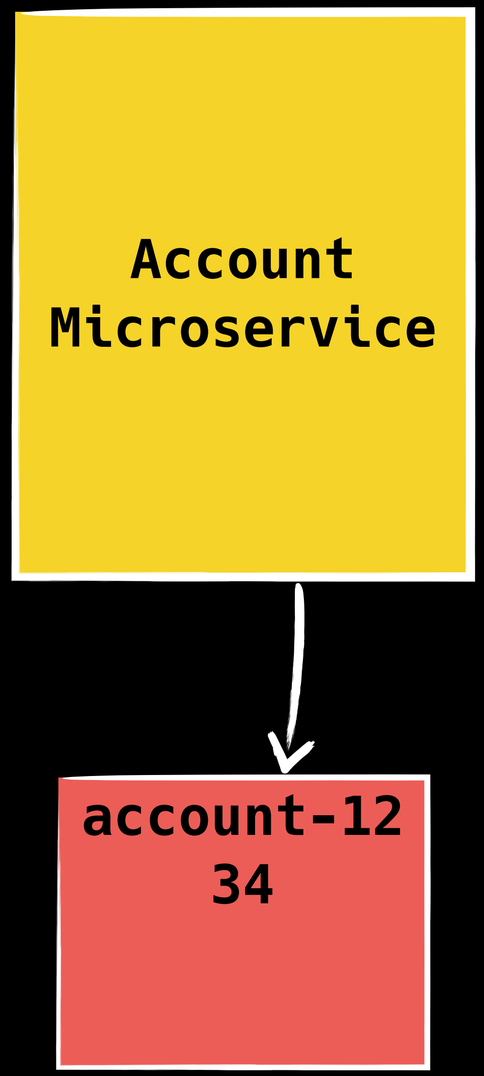



```
{ aggregate, lastVersionNumber } = readAggregateFromStream(...)

events = executeBusinessLogic(...)

emitEvents('account', '1234', events, lastVersionNumber)
```

PUT /account/1234

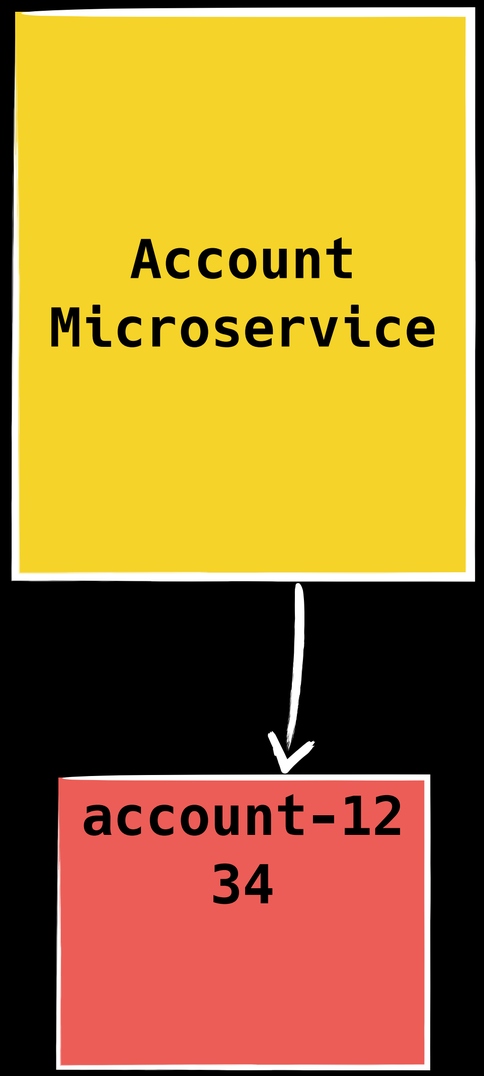



```
{ aggregate, lastVersionNumber } = readAggregateFromStream(...)

events = executeBusinessLogic(...)

emitEvents('account', '1234', events, lastVersionNumber)
```

PUT /account/1234




```
{ aggregate, lastVersionNumber } = readAggregateFromStream(...)

events = executeBusinessLogic(...)

emitEvents('account', '1234', events, lastVersionNumber)
```

And Kafka?

Kafka / KAFKA-2260

Allow specifying expected offset on produce

Details

Type:	Improvement	Status:	OPEN
Priority:	Minor	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Component/s:	producer		
Labels:	None		

Description

I'd like to propose a change that adds a simple CAS-like mechanism to the Kafka producer. This update has a small footprint, but enables a bunch of interesting uses in stream processing or as a commit log for process state.

Andy Bryant added a comment - 27/Jul/18 04:14

👍 Would prove very handy in event source based designs

Russell Ferriday added a comment - 9 hours ago

This would enable full-on eventsourcing on Kafka, without having to restrict to single-thread designs.
One example of a great (>250 github star) FOSS project being held back by this:

https://github.com/johnbywater/eventsourcing/issues/108

Can we see this soon?

Kafka / KAFKA-2260

Allow specifying expected offset on produce

Details

Type:	Improvement	Status:	OPEN
Priority:	Minor	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None

...added a comment - 27/Jul/18 04:14

👍 Would prove very handy in event source based designs

Andy Bryant added a comment - 27/Jul/18 04:14

👍 Would prove very handy in event source based designs

Russell Ferriday added a comment - 9 hours ago

This would enable full-on eventsourcing on Kafka, without having to restrict to single-thread designs.
One example of a great (>250 github star) FOSS project being held back by this:

https://github.com/johnbywater/eventsourcing/issues/108

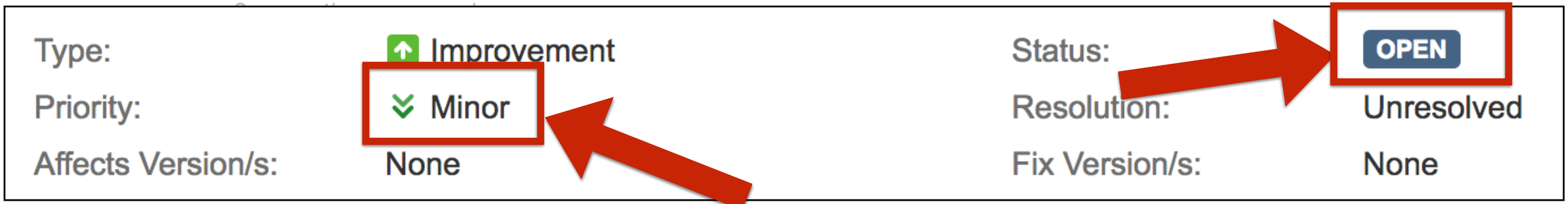
Can we see this soon?

Kafka / KAFKA-2260

Allow specifying expected offset on produce

Details

Type:	Improvement	Status:	OPEN
Priority:	Minor	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None



Andy Bryant added a comment - 27/Jul/18 04:14

Would prove very handy in event source based designs

Russell Ferriday added a comment - 9 hours ago

This would enable full-on eventsourcing on Kafka, without having to restrict to single-thread designs.
One example of a great (>250 github star) FOSS project being held back by this:

https://github.com/johnbywater/eventsourcing/issues/108

Can we see this soon?

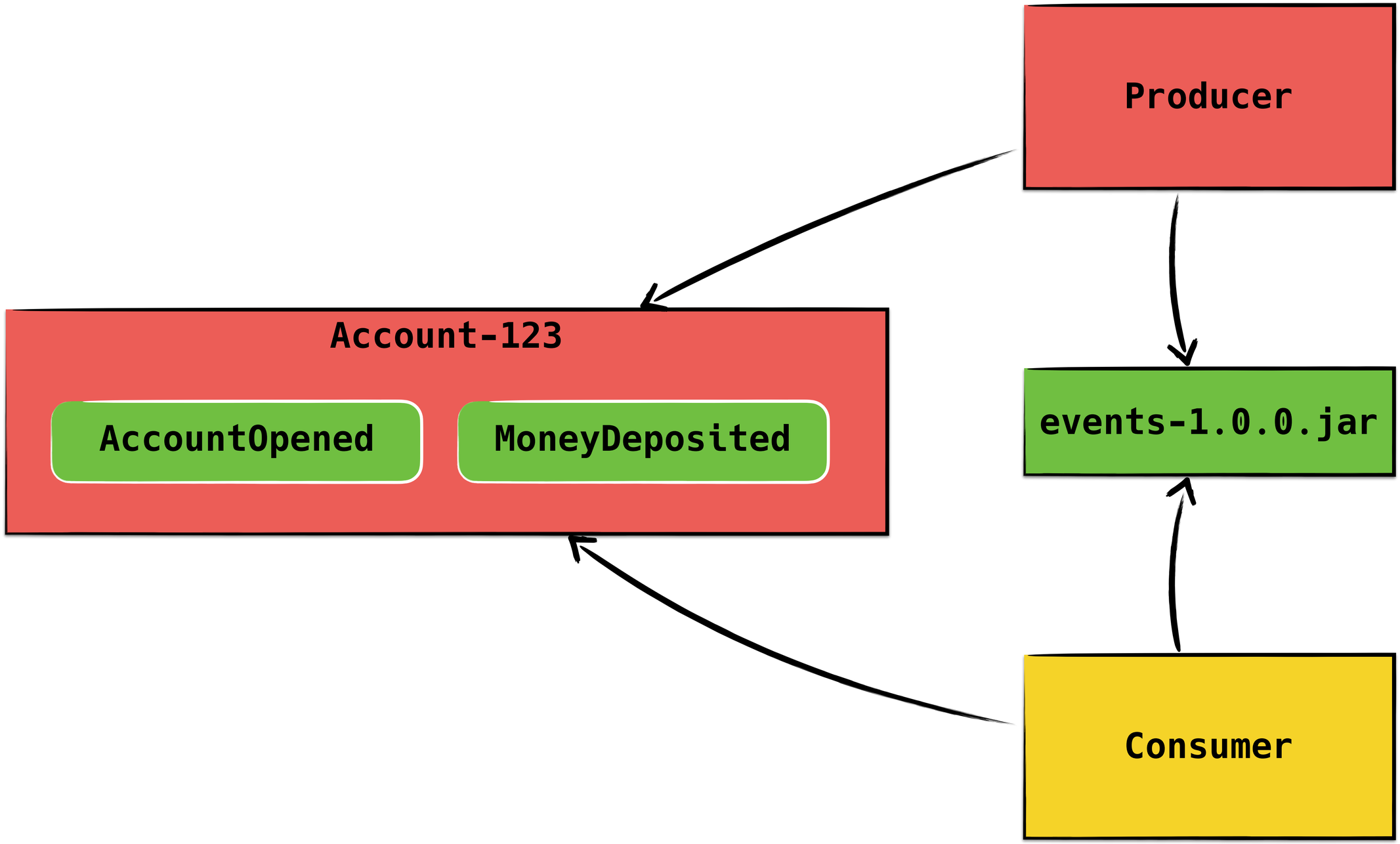
Advantages of OCC?

Scalability and no locks
Consistency
Design choice
Super-simple programming logic

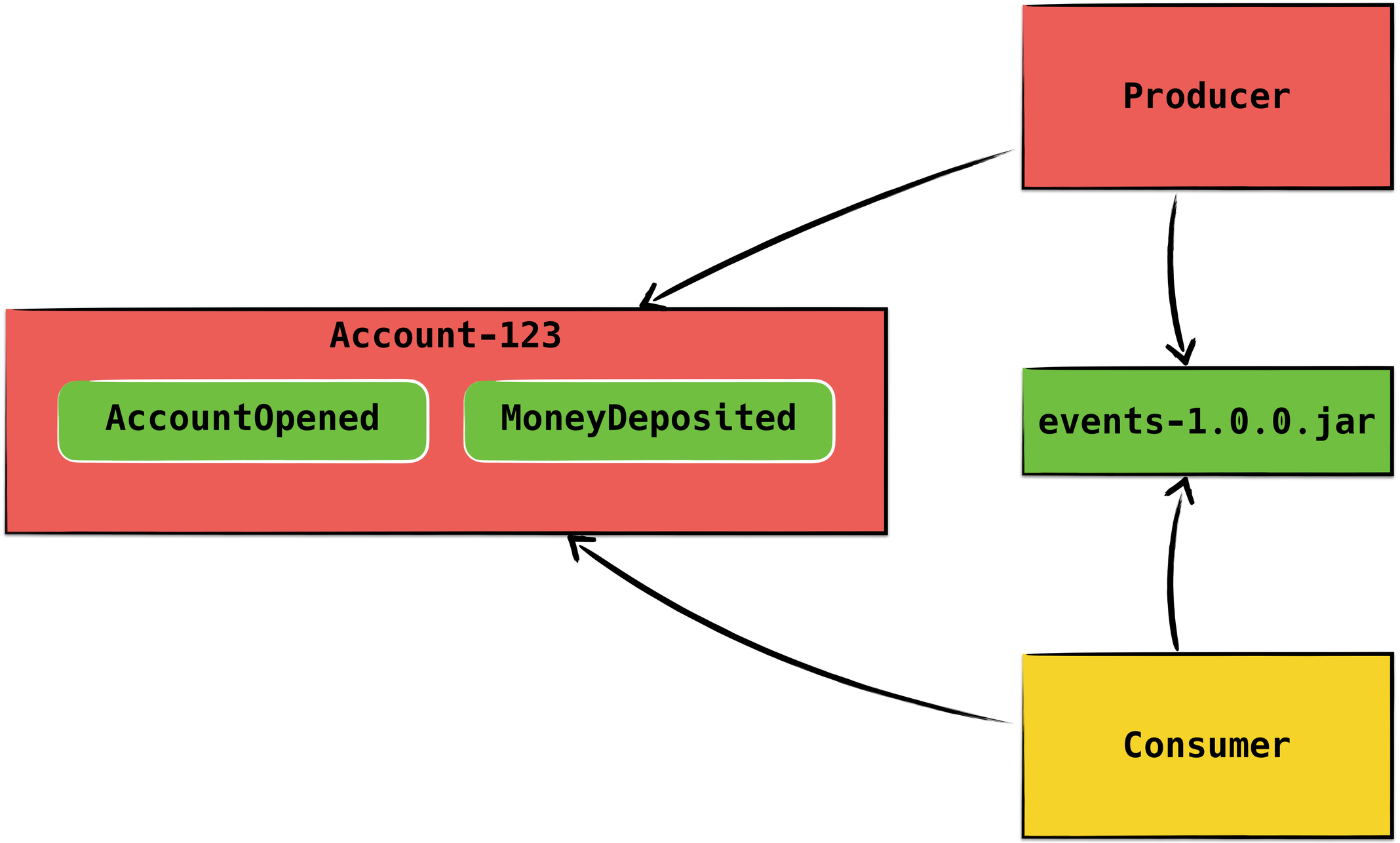

Versions, up-front-design and
breaking things down the road
@koenighotze

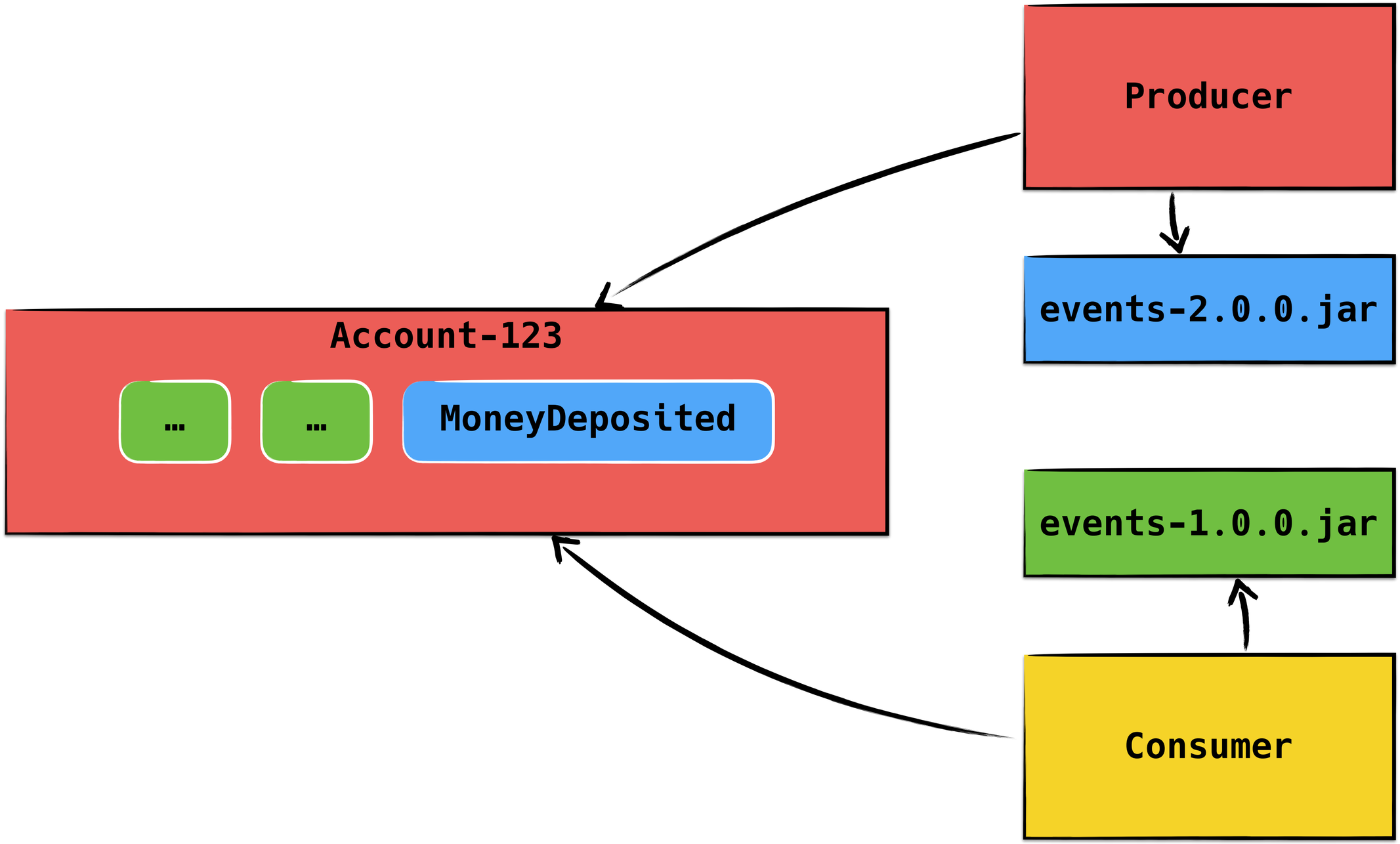
How can we deal with versions without going crazy?

"Just" use semantic versioning and types


Producer
Account-123
AccountOpened
MoneyDeposited
events-1.0.0.jar
Consumer

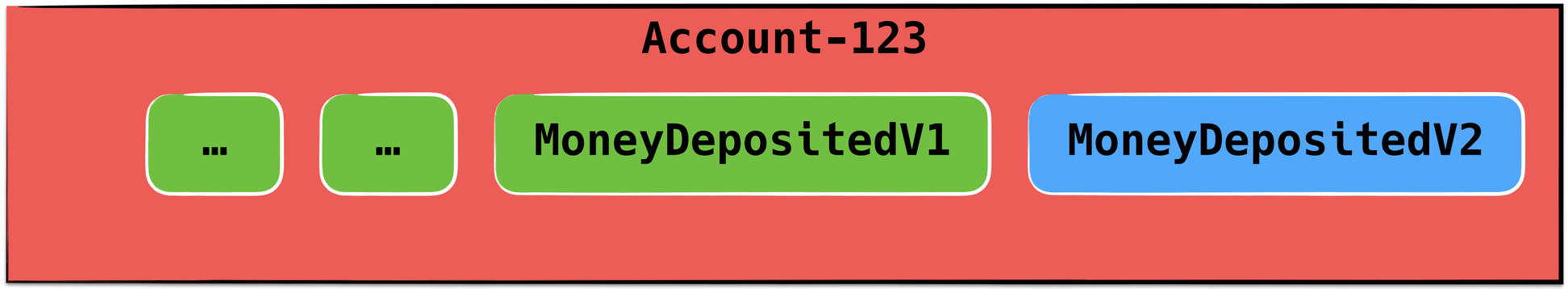
…then change some event types

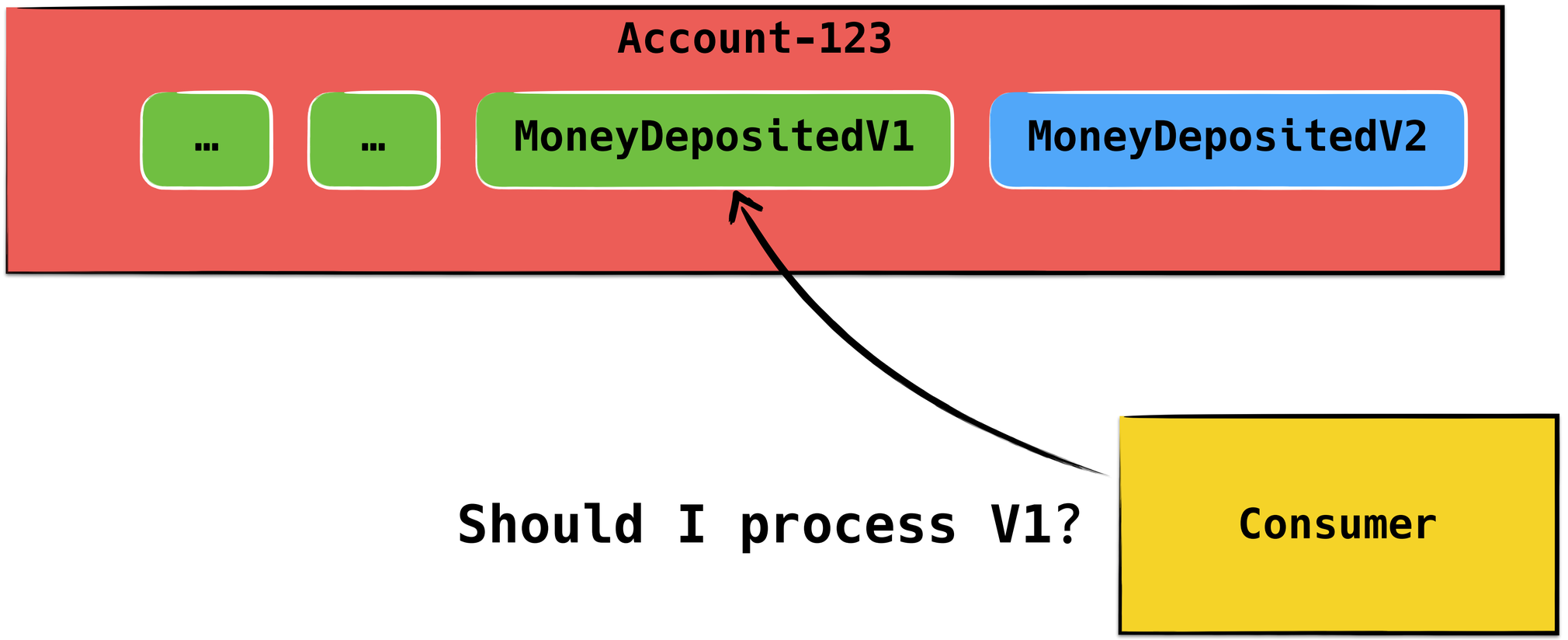

Producer
Account-123
AccountOpened
MoneyDeposited
events-1.0.0.jar
Consumer

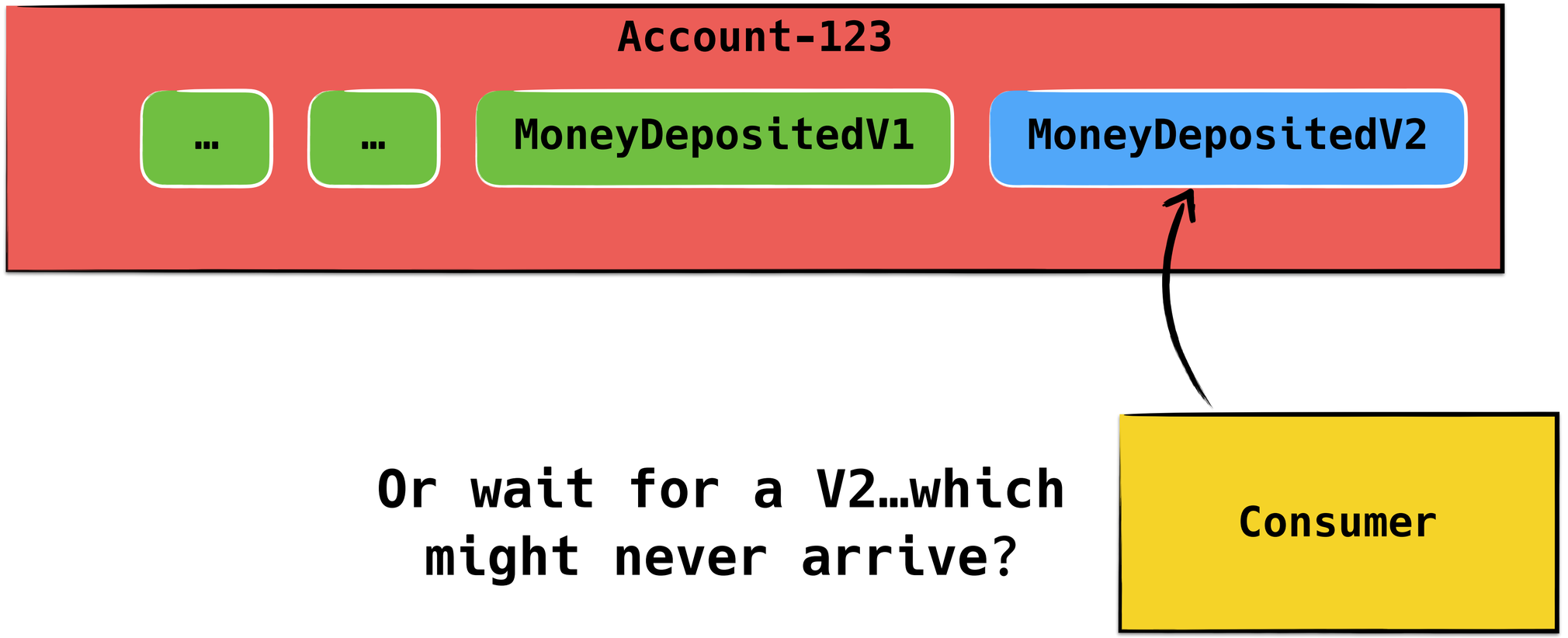

Producer
events-2.0.0.jar
Account-123
…
…
MoneyDeposited
events-1.0.0.jar
Consumer


public class InvalidClassException extends ObjectStreamException {
...
}

Gosh…"just" apply Double Write


Account-123
…
…
MoneyDepositedV1
MoneyDepositedV2


Account-123
…
…
MoneyDepositedV1
MoneyDepositedV2
Should I process V1?
Consumer

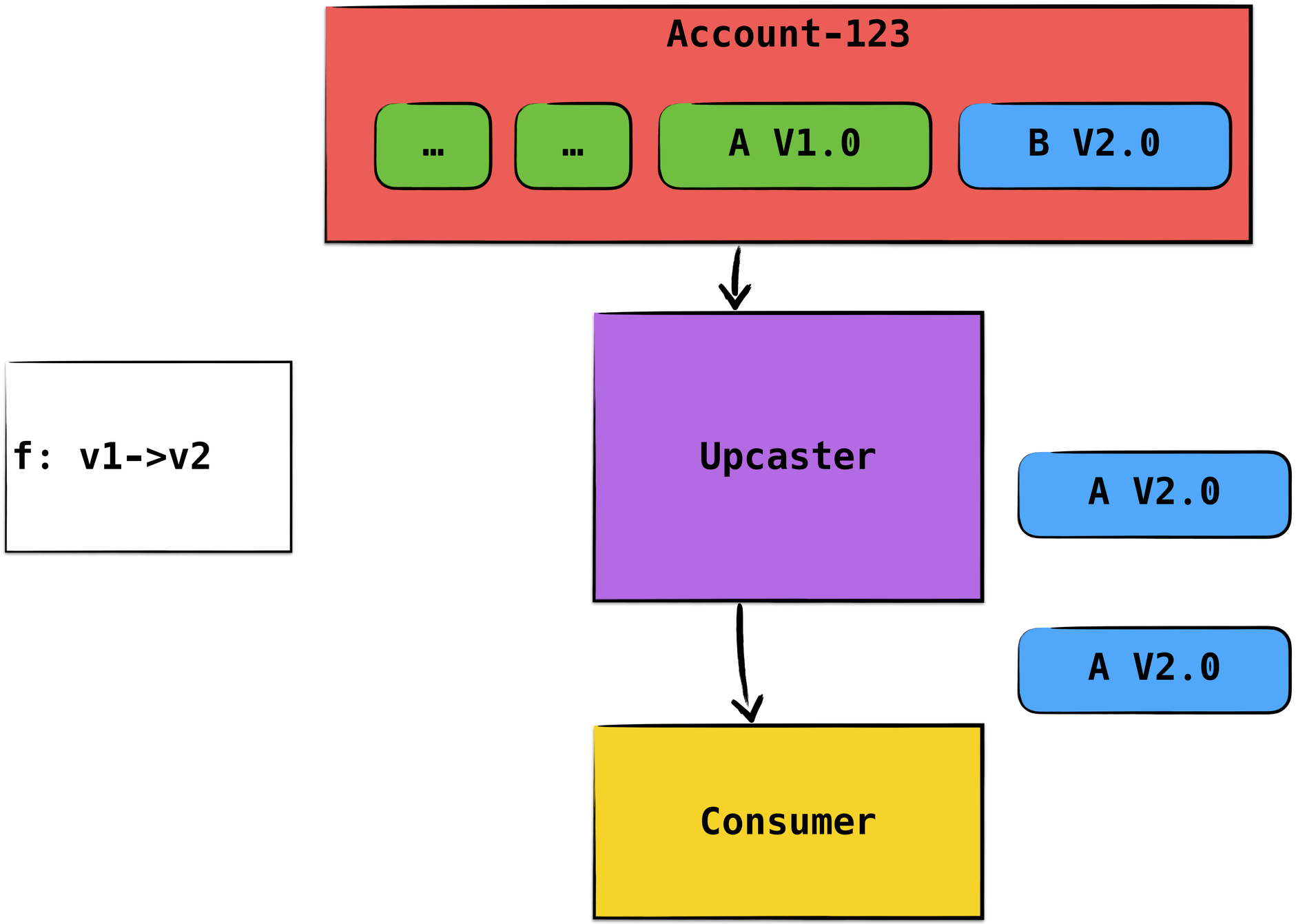


Or wait for a V2…which might never arrive?

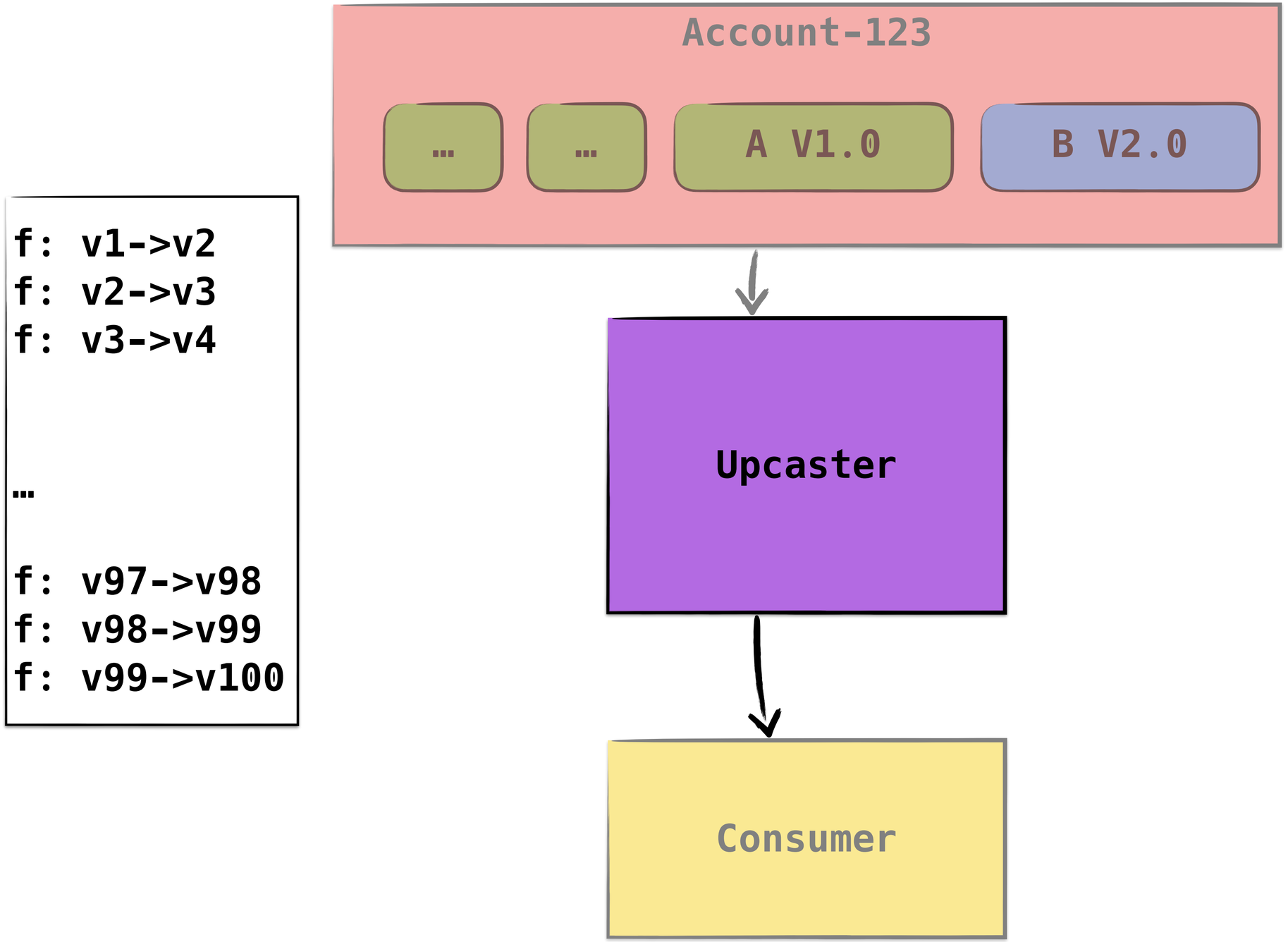
MoneyDeposited_v1
MoneyDeposited_v2
...
MoneyDeposited_v100

MoneyDeposited_v1Handler
MoneyDeposited_v2Handler
...
MoneyDeposited_v100Handler

Upcaster


Account-123
…
…
A V1.0
B V2.0
f: v1->v2
Upcaster
A V2.0
A V2.0
Consumer

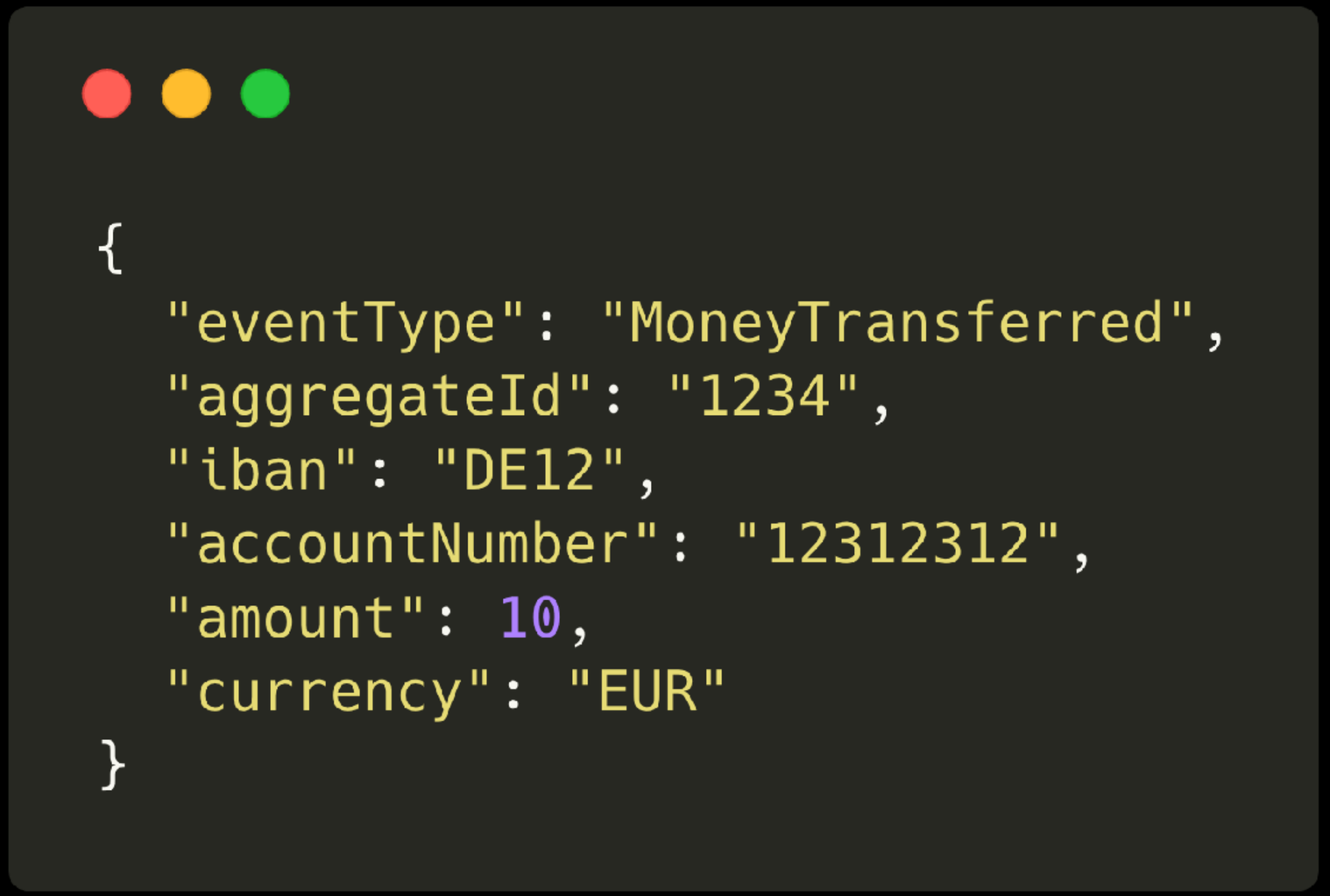
5 months later…


Account-123
…
…
A V1.0
B V2.0
f: v1->v2
f: v2->v3
f: v3->v4
…
f: v97->v98
f: v98->v99
f: v99->v100
Upcaster
Consumer

Good luck maintaining that monster

Prefer simple, text-based, human readable events

Fancy speak for JSON


{
"eventType": "MoneyTransferred",
"aggregateId": "1234",
"iban": "DE12",
"accountNumber": "12312312",
"amount": 10,
"currency": "EUR"
}

And correctness?

"Just" generate classes for JSON mapping!

@koenighotze


public class UserCreatedEvent {
private UUID requestId;
...
}

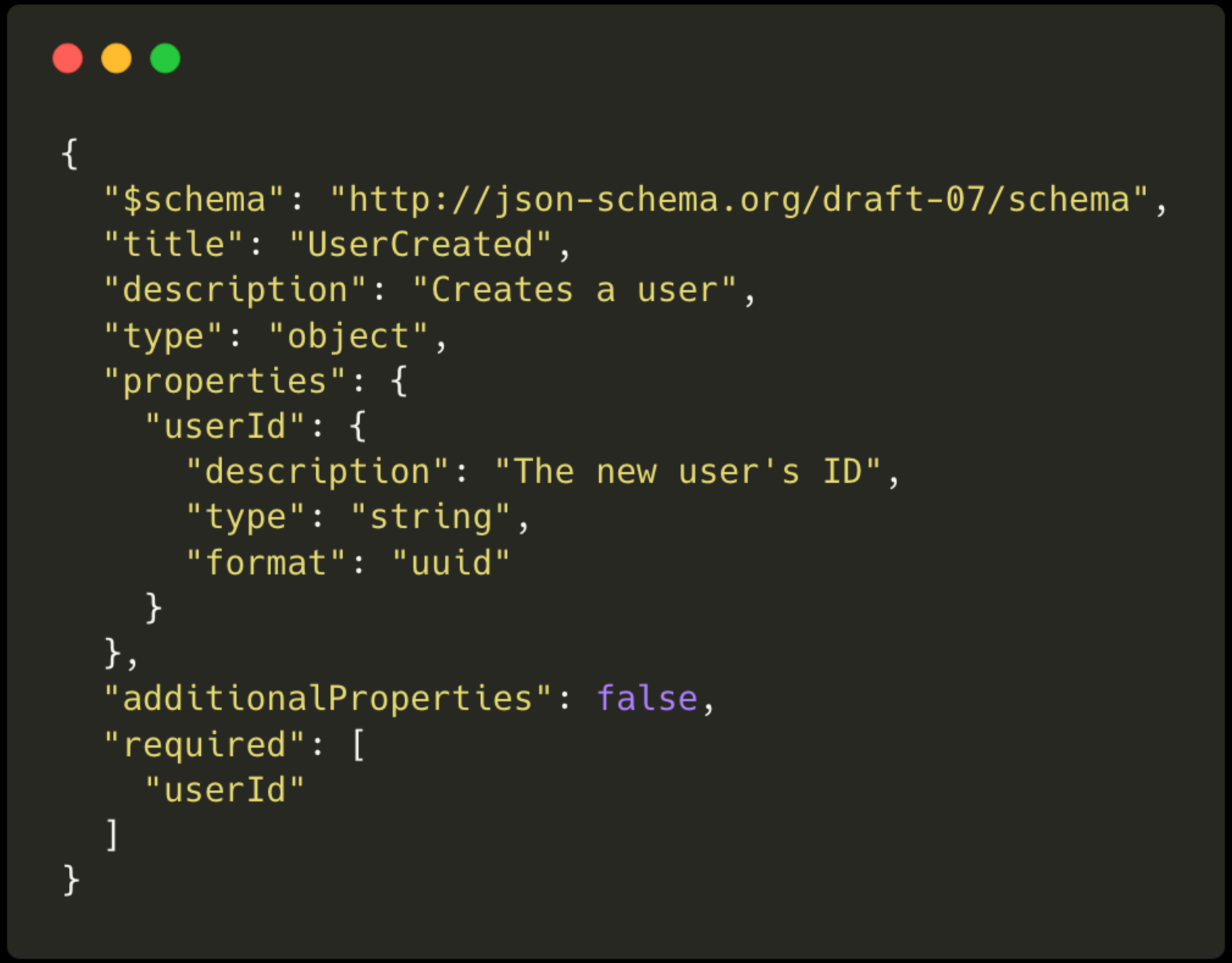
NGINX $request_id

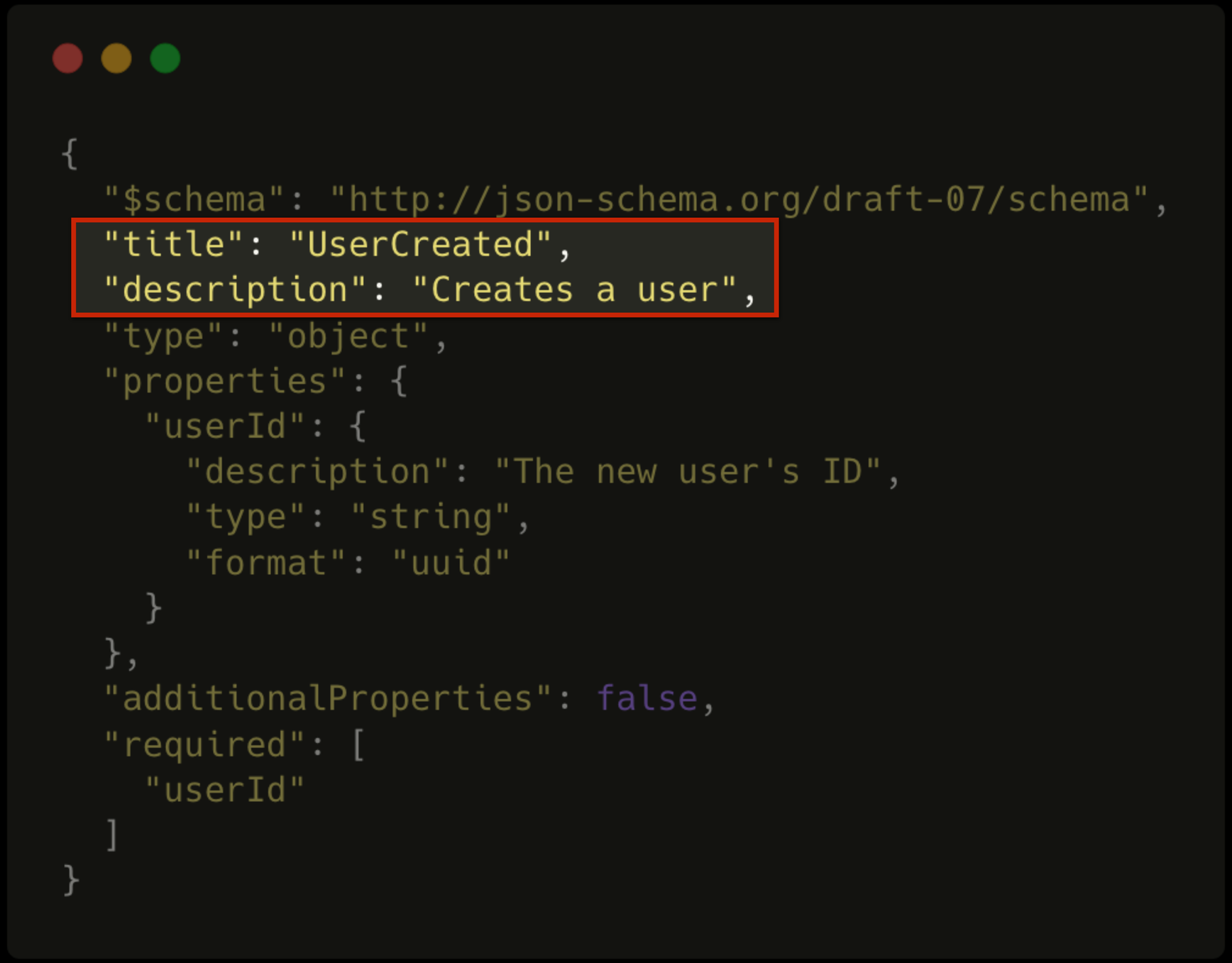
unique request identifier
generated from 16 random bytes,
in hexadecimal (1.11.0)

“Oh, you changed the request id from uuid to any arbitrary string”

String-ly typed events work really well

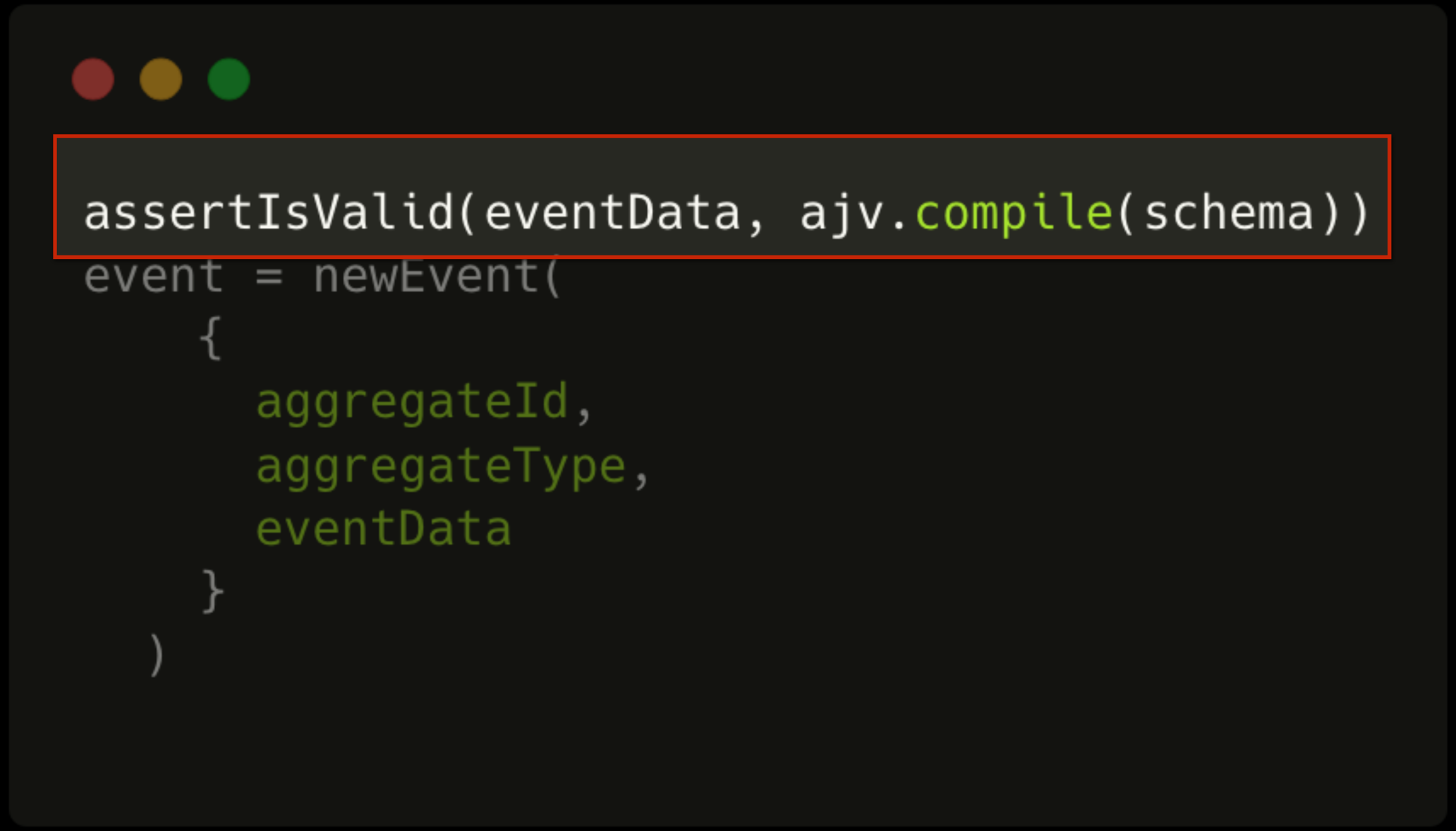
Weak schema to the rescue



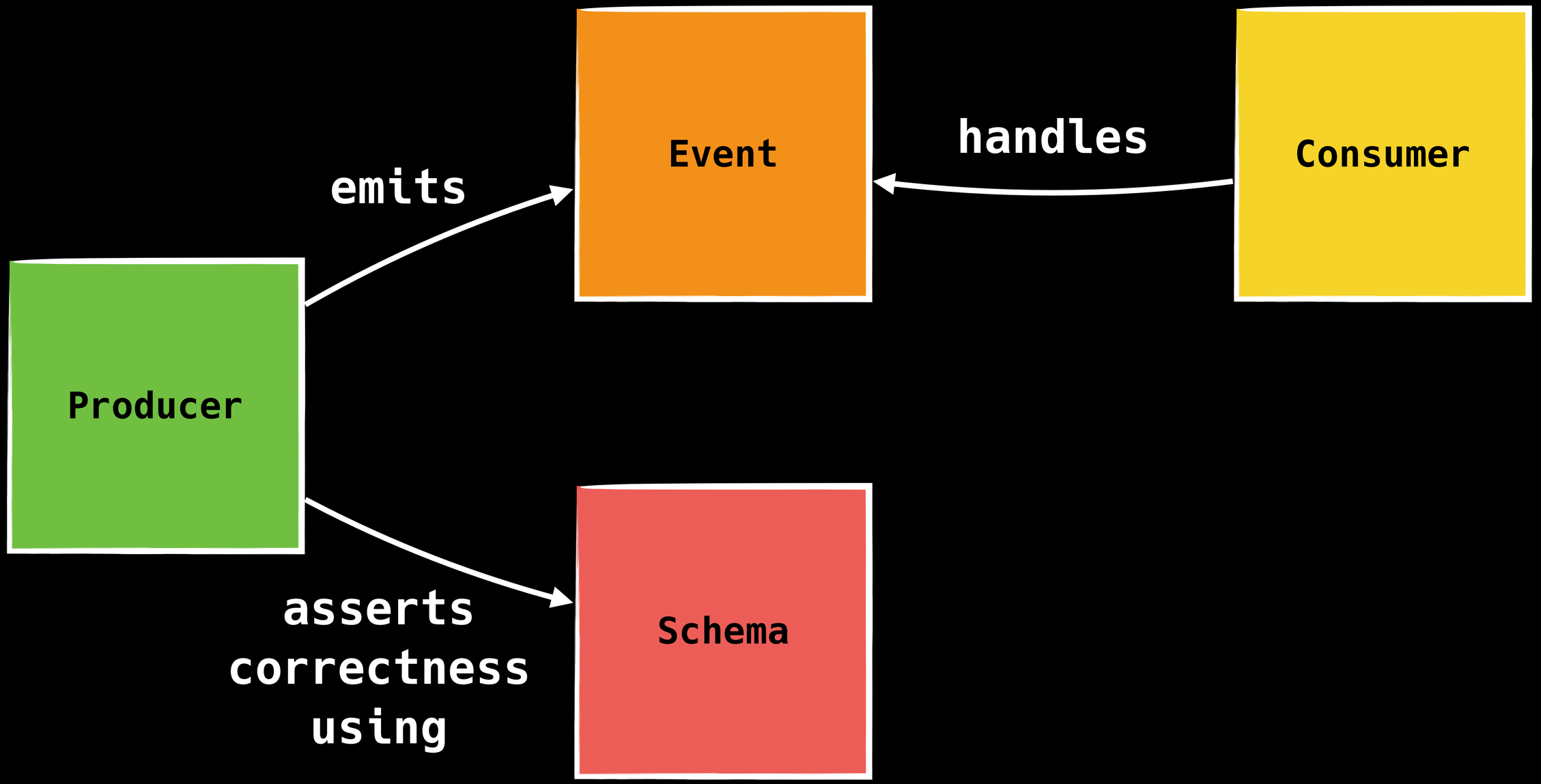

{
"$schema": "http://json-schema.org/draft-07/schema",
"title": "UserCreated",
"description": "Creates a user",
"type": "object",
"properties": {
"userId": {
"description": "The new user's ID",
"type": "string",
"format": "uuid"
}
},
"additionalProperties": false,
"required": [
"userId"
]
}

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "title": "UserCreated",
  "description": "Creates a user",
  "type": "object",
  "properties": {
    "userId": {
      "description": "The new user's ID",
      "type": "string",
      "format": "uuid"
    }
  },
  "additionalProperties": false,
  "required": [
    "userId"
  ]
}
```

```
assertIsValid(eventData, ajv.compile(schema))
event = newEvent(
    {
      aggregateId,
      aggregateType,
      eventData
    }
  )
```


assertIsValid(eventData, ajv.compile(schema))
event = newEvent(
{
aggregateId,
aggregateType,
eventData
}
)

Schema as a description
not as a contract


Event
handles
Consumer
emits
Producer
asserts
correctness
using
Schema

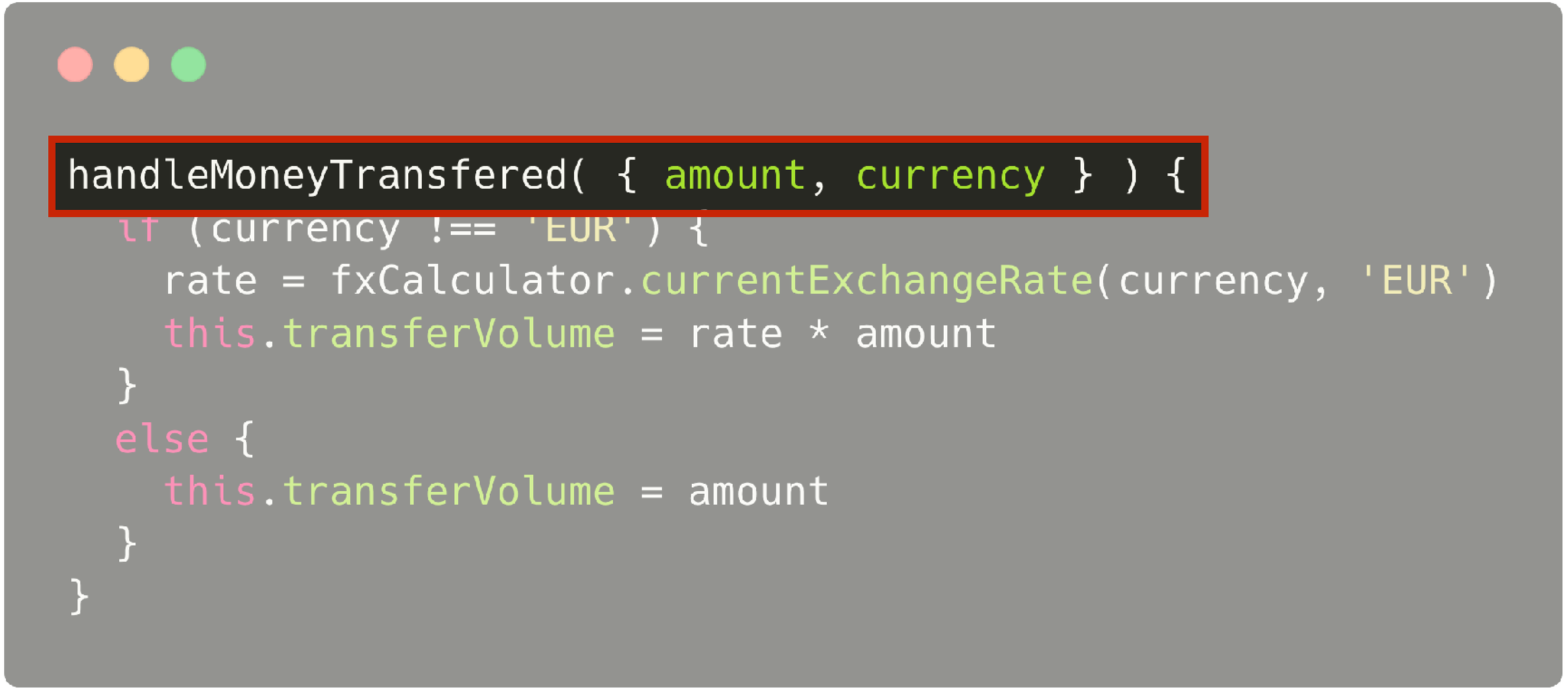
What about putting versioned logic in handlers?

MoneyTransferred

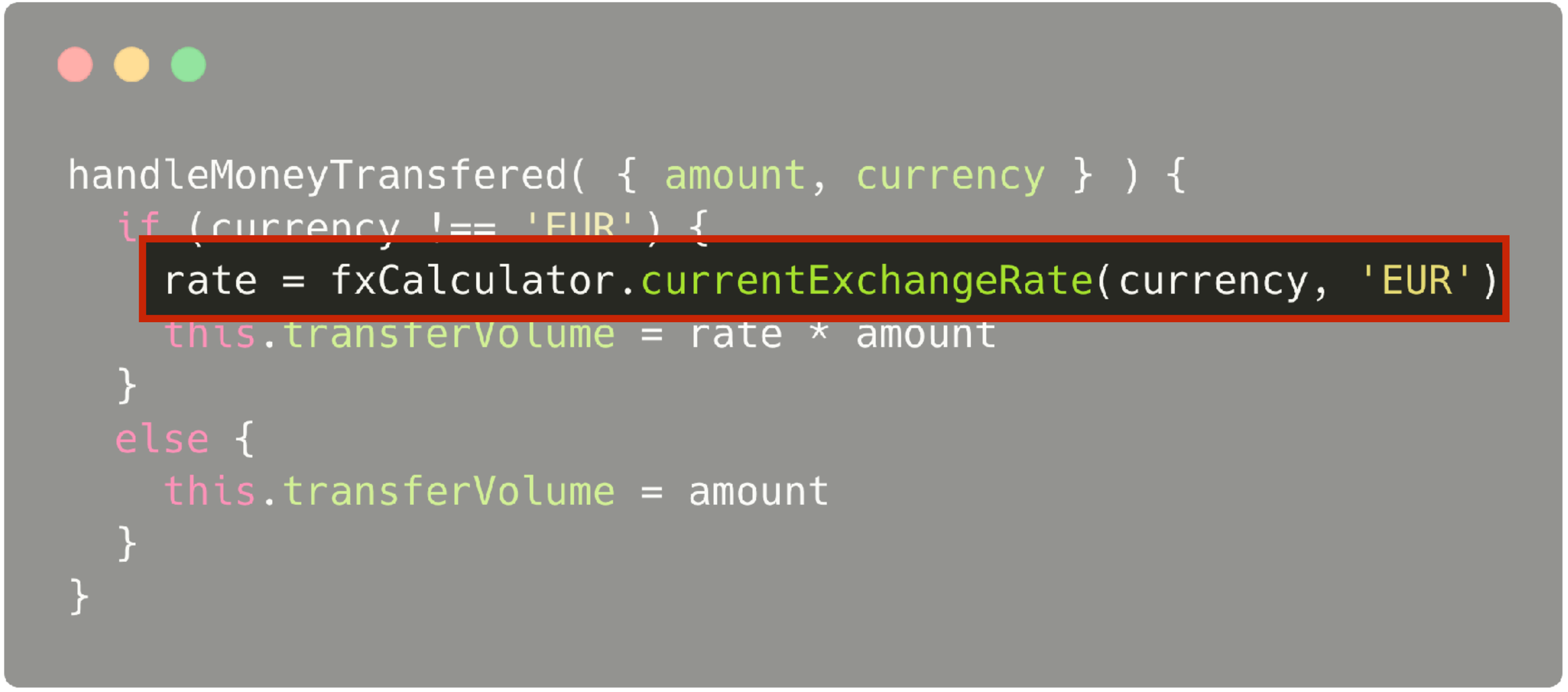
eventId: 5

amount: 97 USD


handleMoneyTransfered({ amount, currency }) {
if (currency !== 'EUR') {
rate = fxCalculator.currentExchangeRate(currency, 'EUR')
this.transferVolume = rate * amount
}
else {
this.transferVolume = amount
}
}


handleMoneyTransfered({ amount, currency }) {
if (currency !== 'EUR') {
rate = fxCalculator.currentExchangeRate(currency, 'EUR')
this.transferVolume = rate * amount
}
else {
this.transferVolume = amount
}
}

```
handleMoneyTransfered( { amount, currency } ) {
  if (currency !== 'EUR') {
    rate = fxCalculator.currentExchangeRate(currency, 'EUR')
    this.transferVolume = rate * amount
  }
  else {
    this.transferVolume = amount
  }
}
```


handleMoneyTransfered({ amount, currency }) {
if (currency !== 'EUR') {
rate = fxCalculator.currentExchangeRate(currency, 'EUR')
this.transferVolume = rate * amount
}
else {
this.transferVolume = amount
}
}

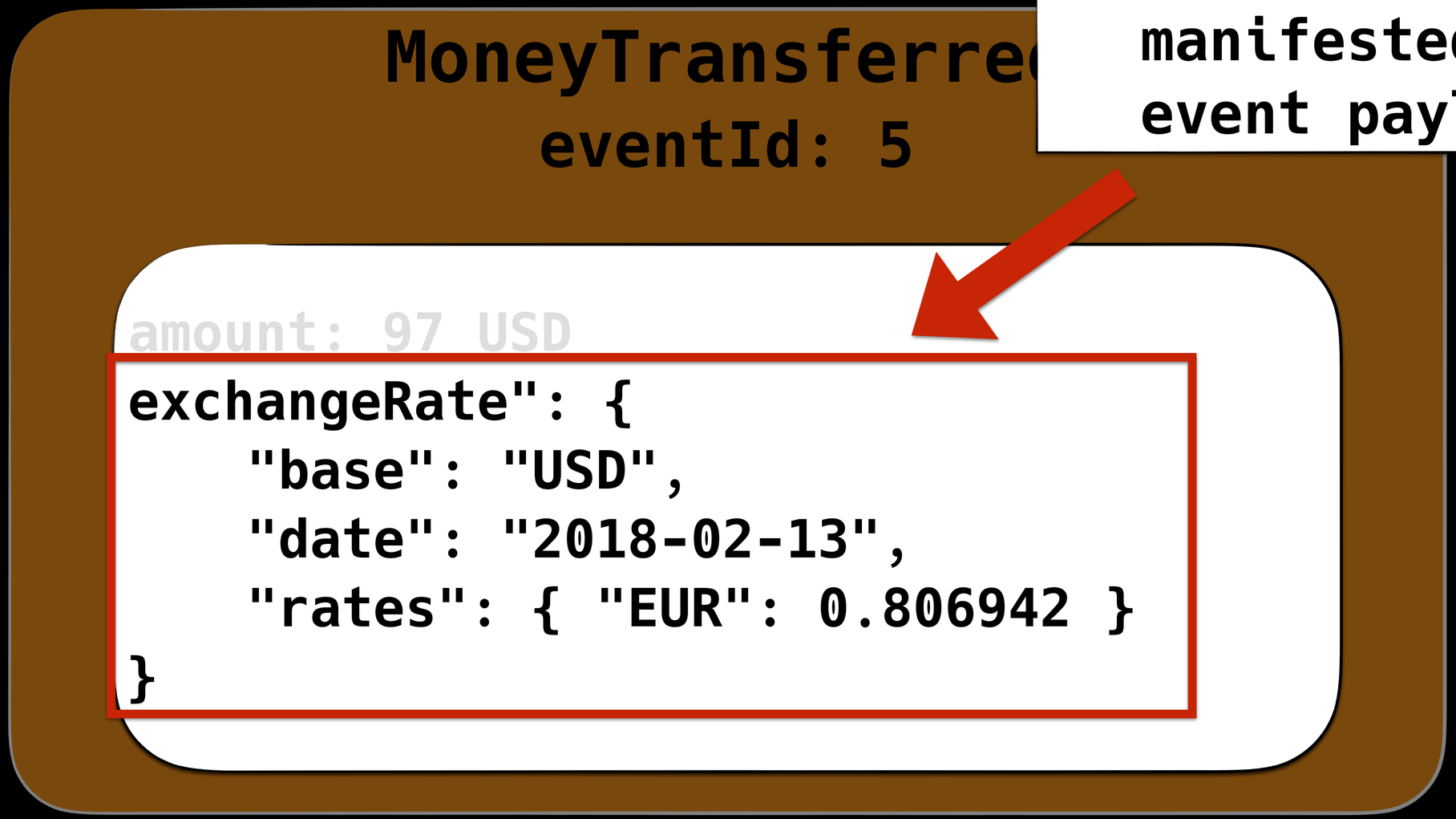
Bravo, now your expense report of 2017 depends on today's exchange rates

Creating an event must encapsulate all data that lead to the emitting of the event

MoneyTransferred

eventId: 5

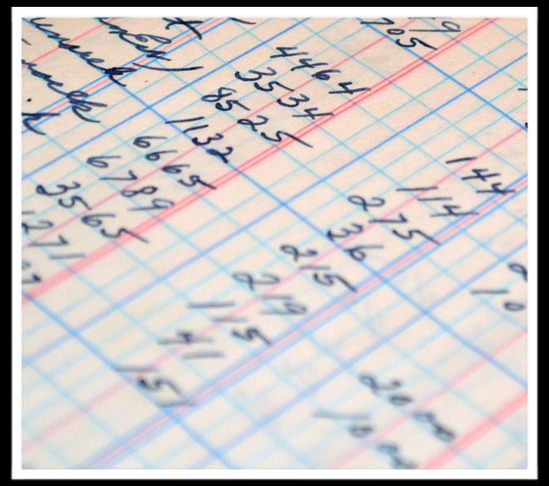
```
amount: 97 USD
exchangeRate": {
    "base": "USD",
    "date": "2018-02-13",
    "rates": { "EUR": 0.806942 }
}
```


MoneyTransferre
eventId: 5
amount: 97 USD
exchangeRate": {
"base": "USD",
"date": "2018-02-13",
"rates": { "EUR": 0.806942 }
}
Side-effect manifested as event payload

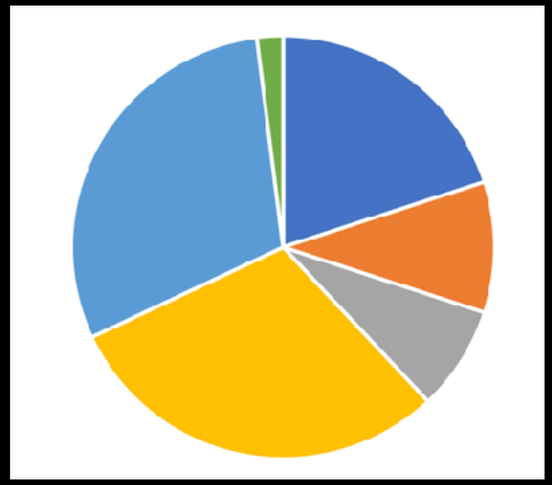
Reduce stream-replay headaches by storing side-effects as event results

Reusing event data?

Transactionledger Microservice



Budget Planer Microservice



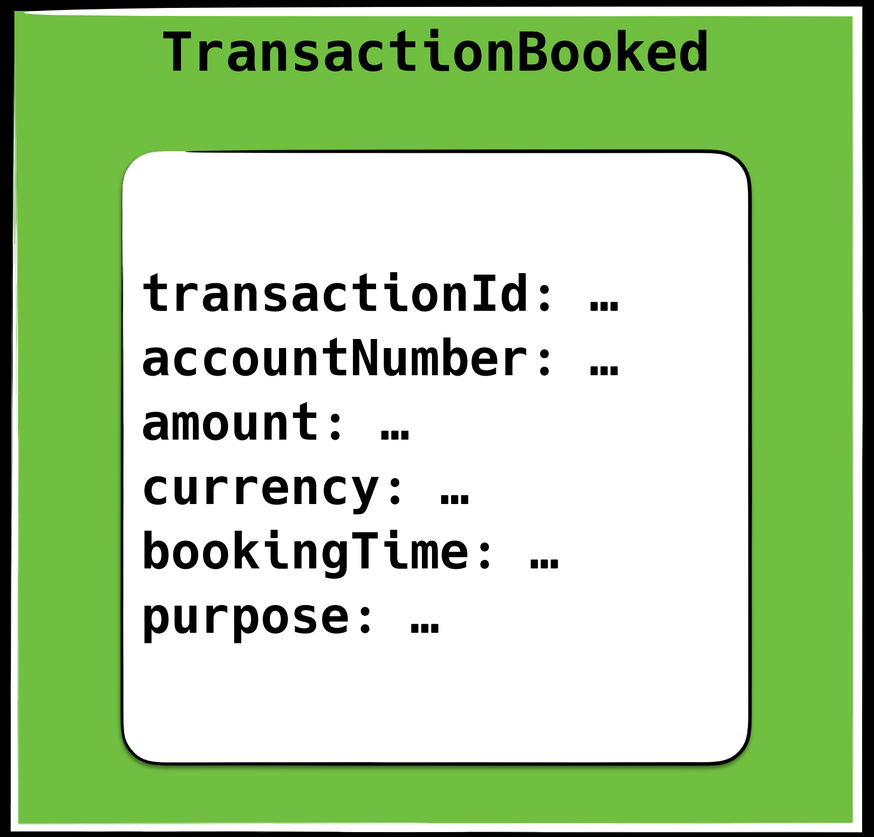
Transactionledger Microservice

Budget Planer Microservice

TransactionBooked

TransactionCategorised

"Just" copy data into different events, "just" so convenient

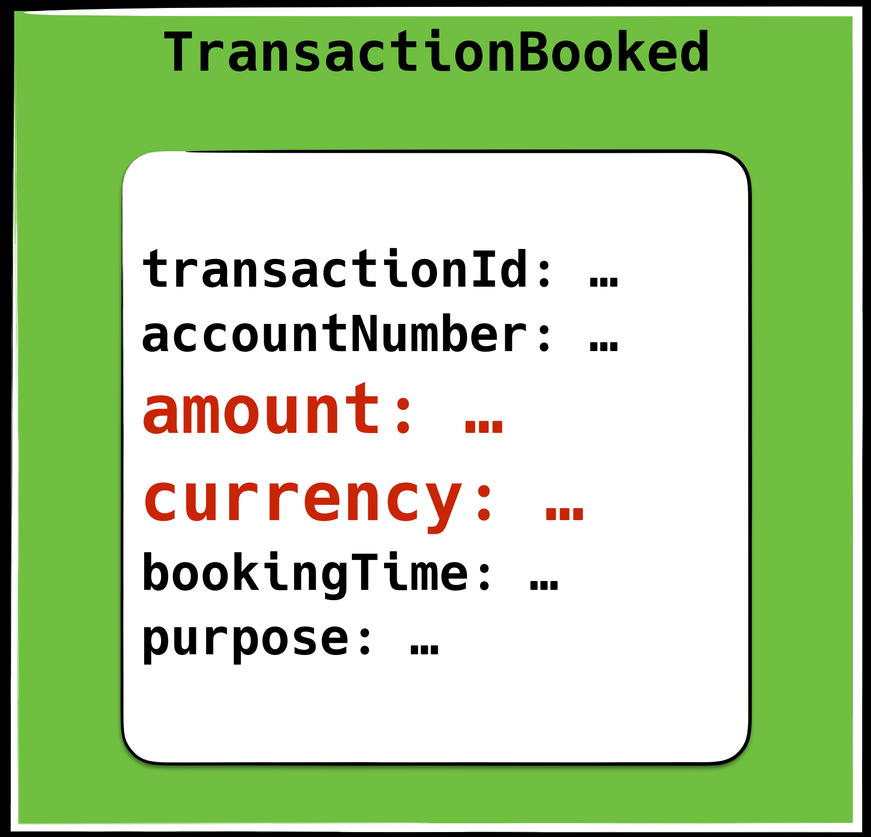

TransactionBooked
transactionId: …
accountNumber: …
amount: …
currency: …
bookingTime: …
purpose: …

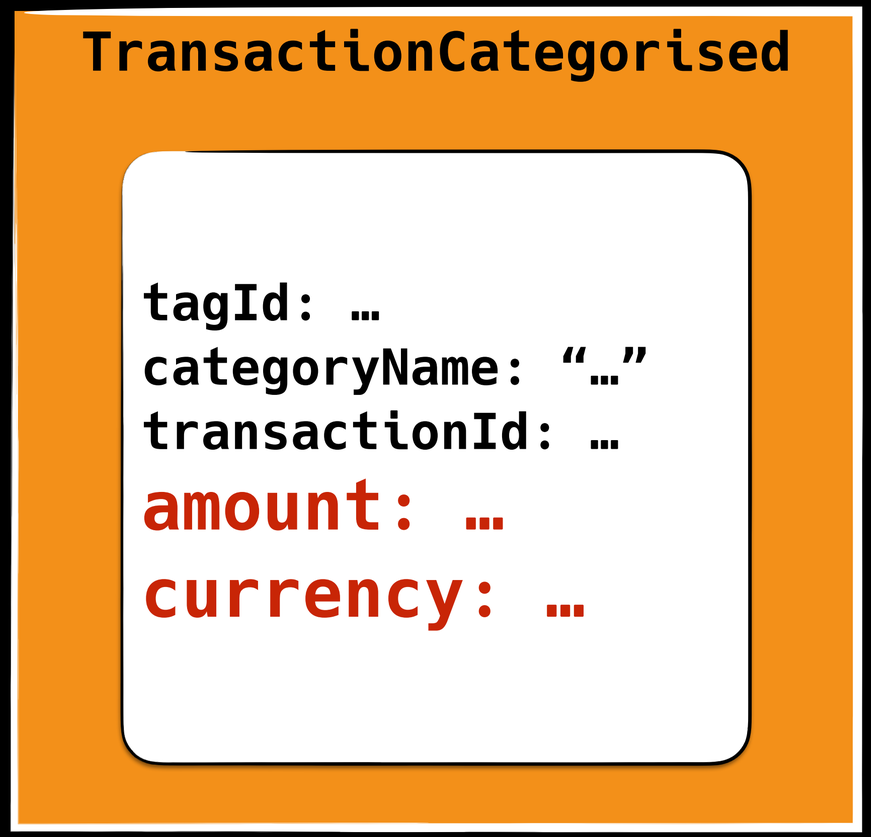
TransactionBooked

```
transactionId: …
accountNumber: …
amount: …
currency: …
bookingTime: …
purpose: …
```

TransactionCategorised

```
tagId: …
categoryName: “…”
transactionId: …
amount: …
currency: …
```


TransactionBooked
transactionId: …
accountNumber: …
amount: …
currency: …
bookingTime: …
purpose: …


TransactionCategorised
tagId: …
categoryName: "…"
transactionId: …
amount: …
currency: …


But I need to display the transaction purpose, too
Budget Planer Microservice

TransactionBooked

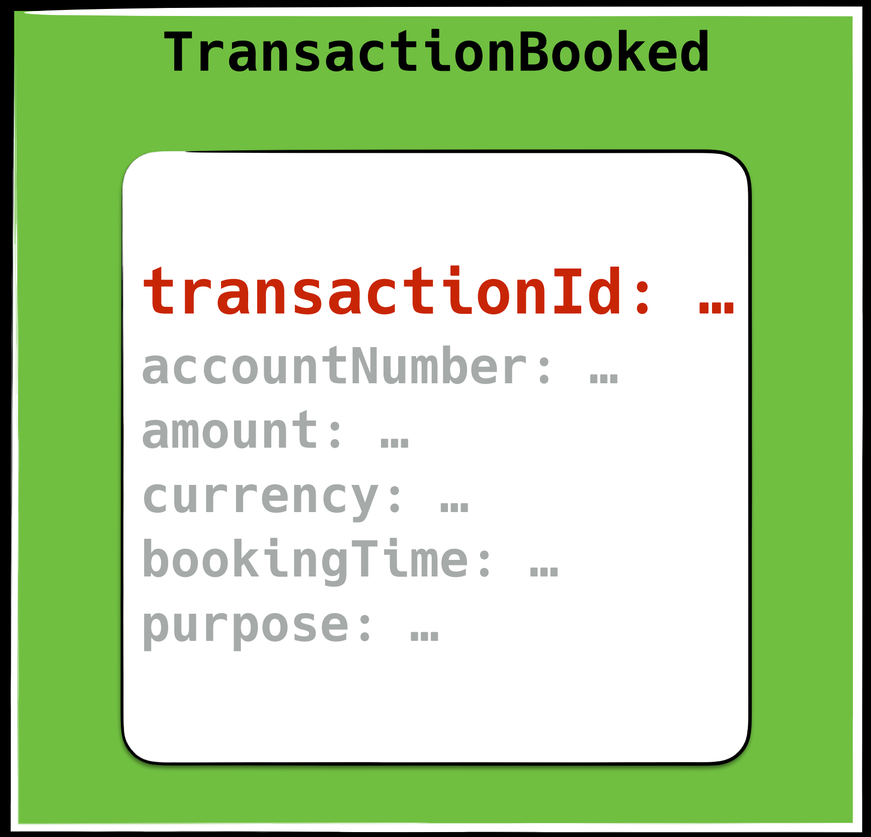
```
transactionId: …
accountNumber: …
amount: …
currency: …
bookingTime: …
purpose: …
```

TransactionCategorised

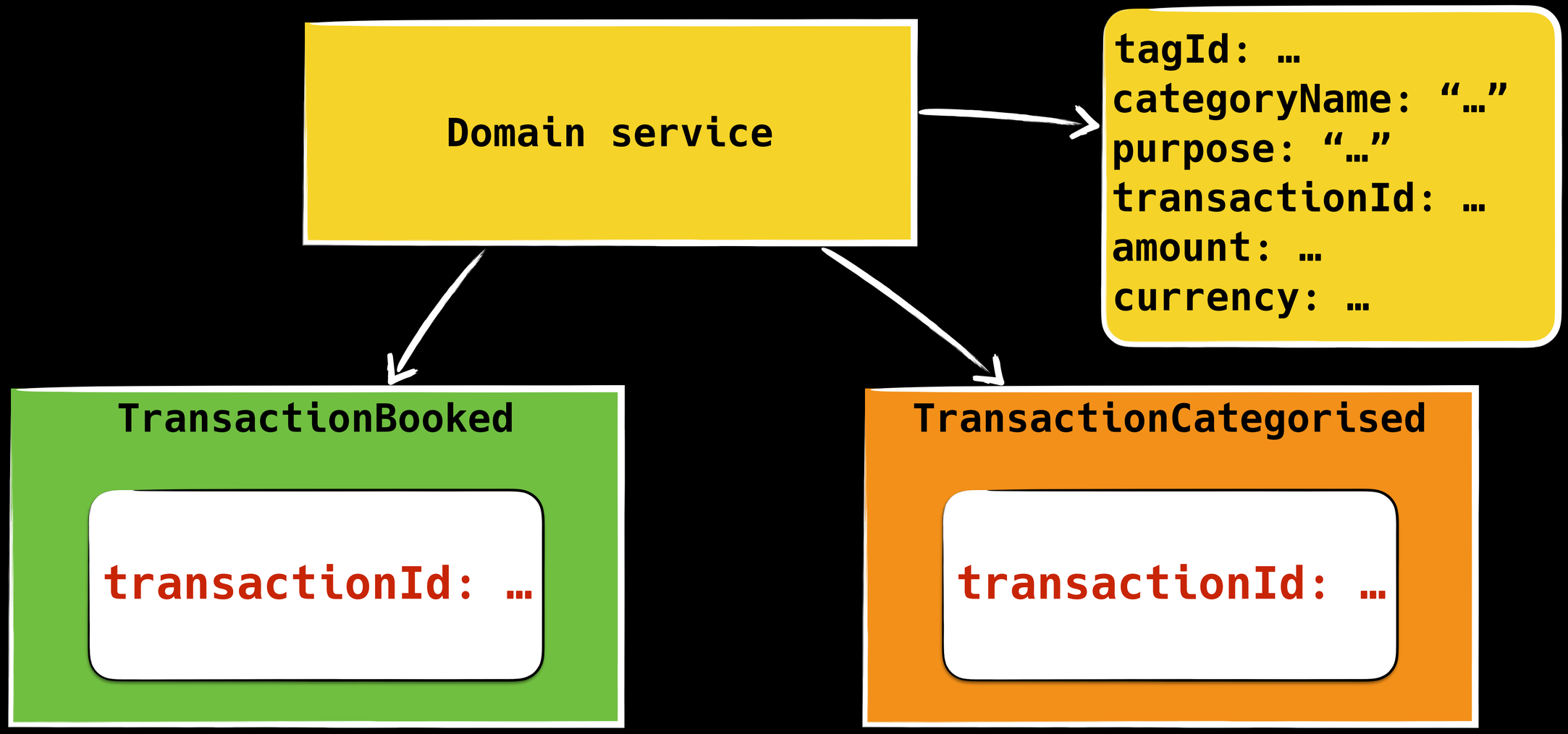
```
tagId: …
categoryName: “…”
transactionId: …
amount: …
currency: …
????
```

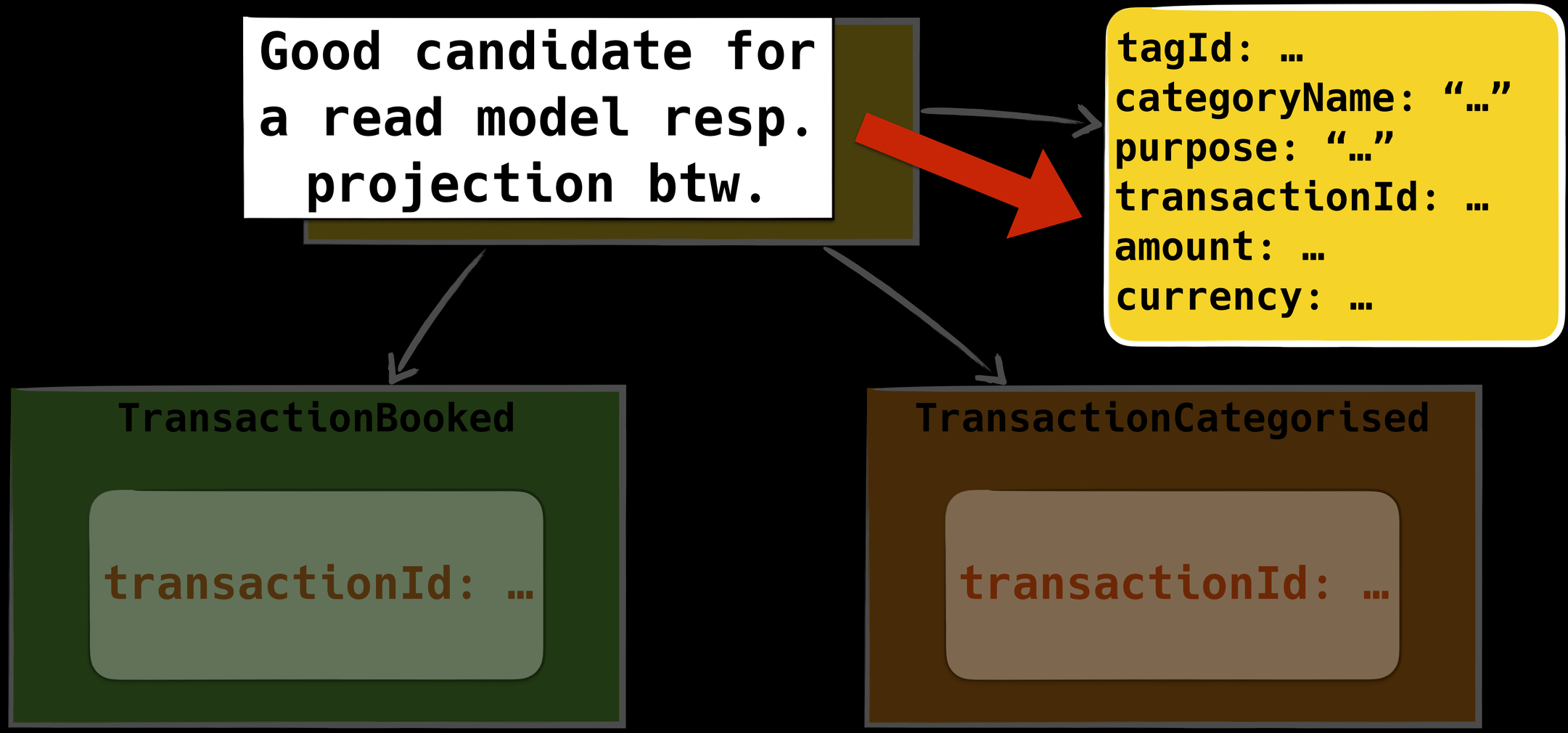
The lossy event

Only reference aggregates via their root id


TransactionBooked
transactionId: …
accountNumber: …
amount: …
currency: …
bookingTime: …
purpose: …


TransactionCategorised
tagId: …
categoryName: "…"
transactionId: …


Domain service
tagId: …
categoryName: “…”
purpose: “…”
transactionId: …
amount: …
currency: …
TransactionBooked
transactionId: …
TransactionCategorised
transactionId: …


Good candidate for a read model resp. projection btw.
tagId: …
categoryName: “…”
purpose: “…”
transactionId: …
amount: …
currency: …
TransactionBooked
transactionId: …
TransactionCategorised
transactionId: …

Don't copy parts of an event.
Prefer building use case specific projections

How can you handle event data over a long period of time?

You don´t

"Just" take a snapshot of the stream

Year's end procedure

Year end – also known as an accounting reference date – is the completion of an accounting period. At this time, businesses need to carry out specific procedures to close their books.

https://debitoor.com/dictionary/year-end

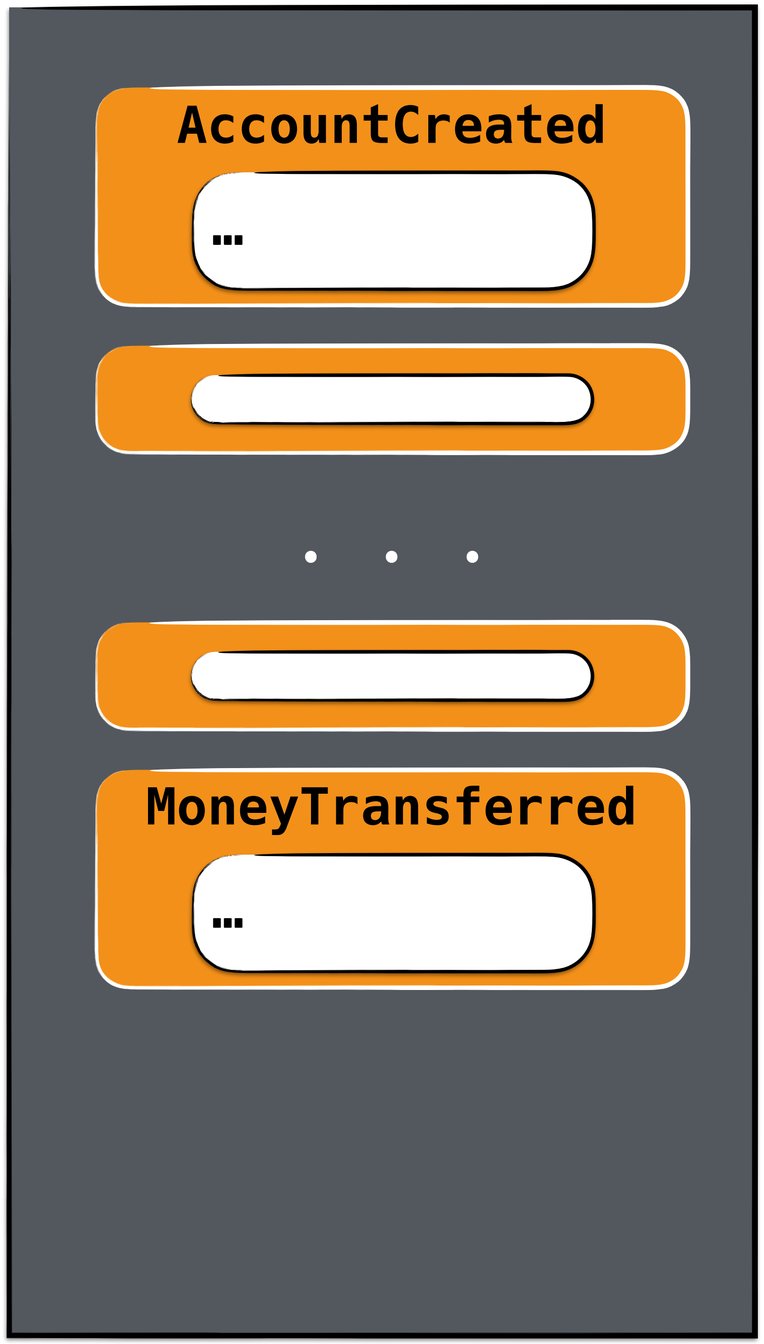
> Year end – also known as an accounting reference date – is the completion of an accounting period. At this time, **businesses need to carry out specific procedures to close their books**.

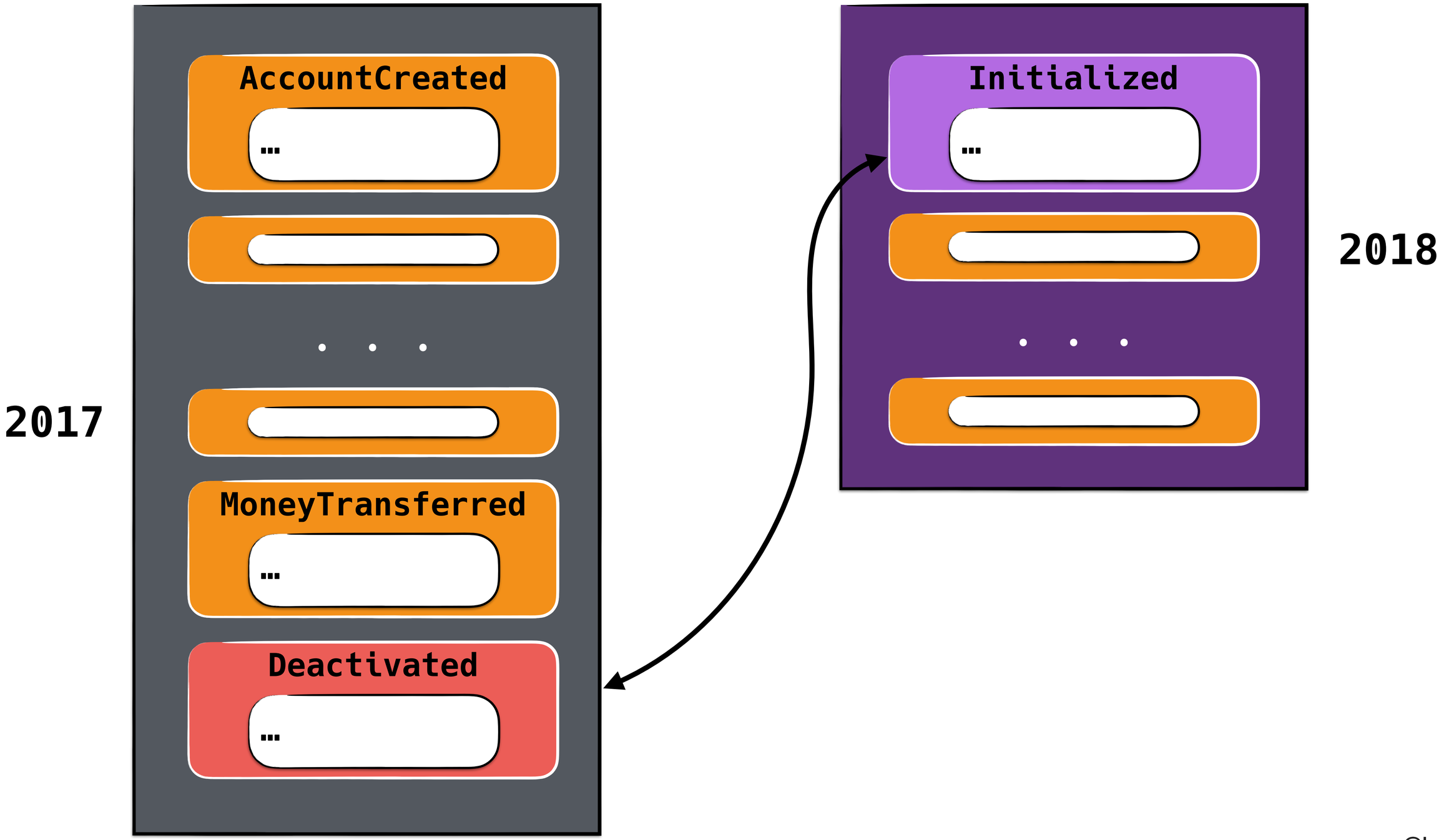
https://debitoor.com/dictionary/year-end

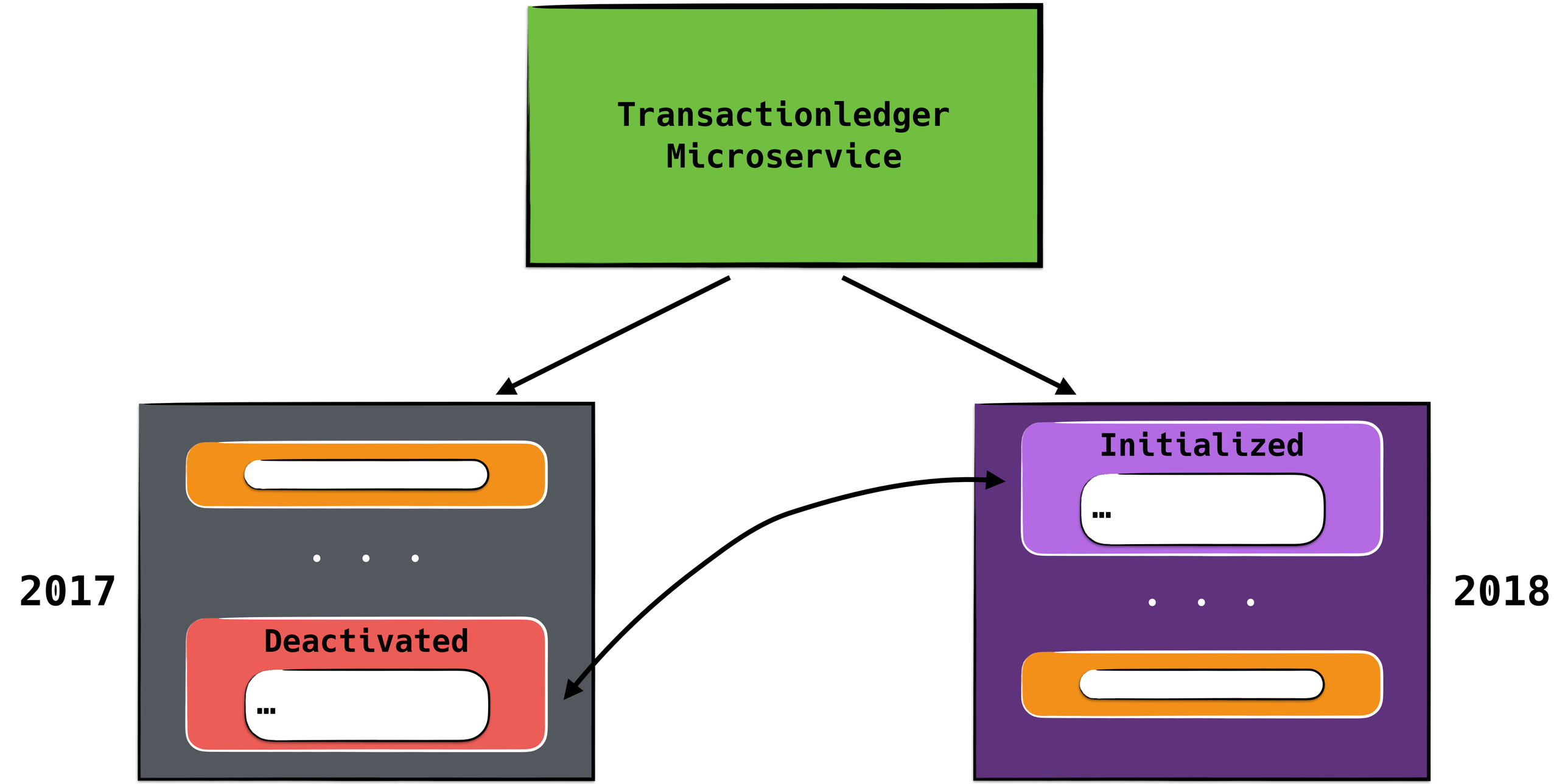
Copy-Transform

a.k.a. eventsourcing refactoring powertool

2017




2017
AccountCreated
...
MoneyTransferred
...
Deactivated
...
Initialized
...
2018


Transactionledger
Microservice
2017
Deactivated
...
2018
Initialized
...

Same idea if you need to remodel your domain!

The devil is in the detail

Dealing with errors

@koenighotze

MoneyTransferred

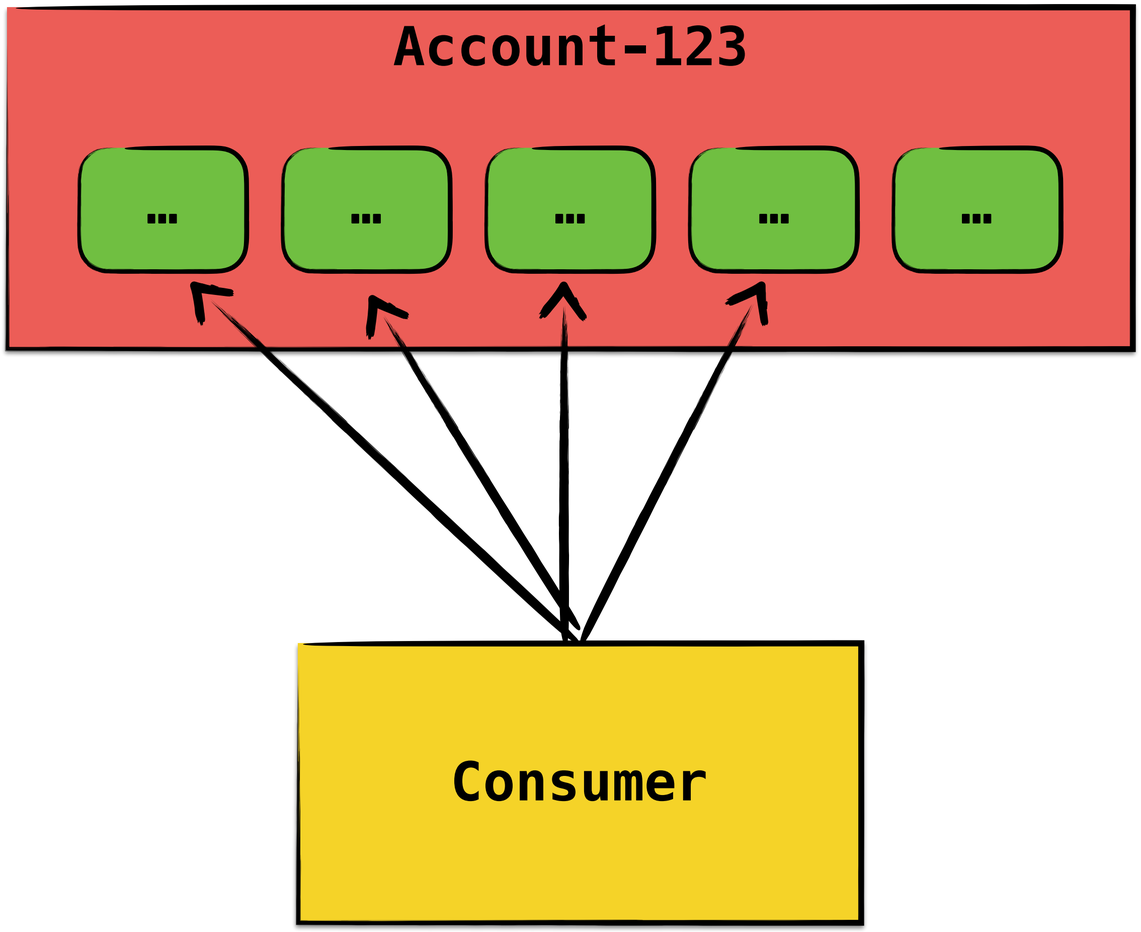
eventId: 231233

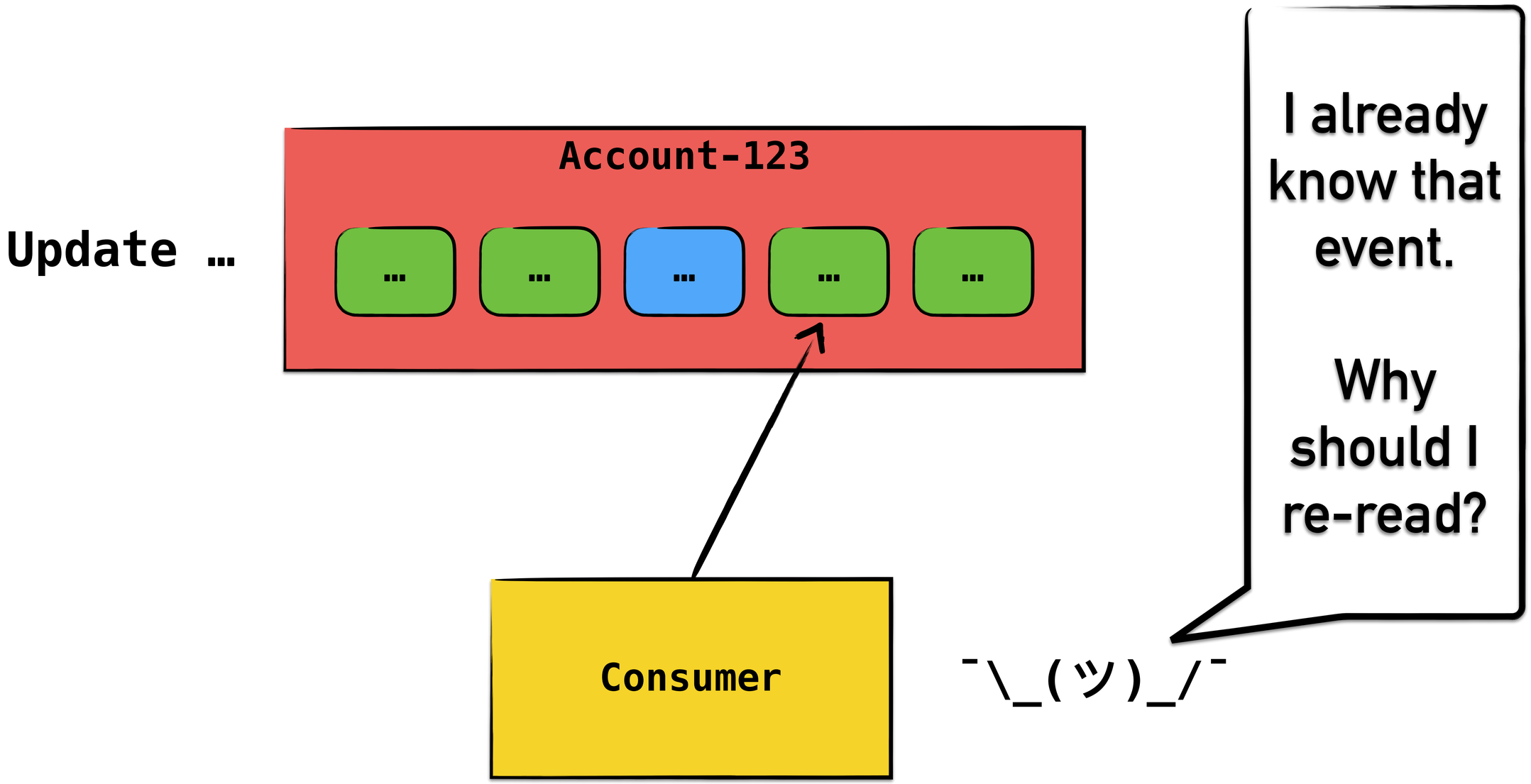
```
amount: 97 Euro
withdrawnAt: 2018-08-30T08:58:26.624
```

“Just” update the event in the eventstore!


UPDATE transactions
SET currency='EUR'
WHERE eventId='231233'

No!

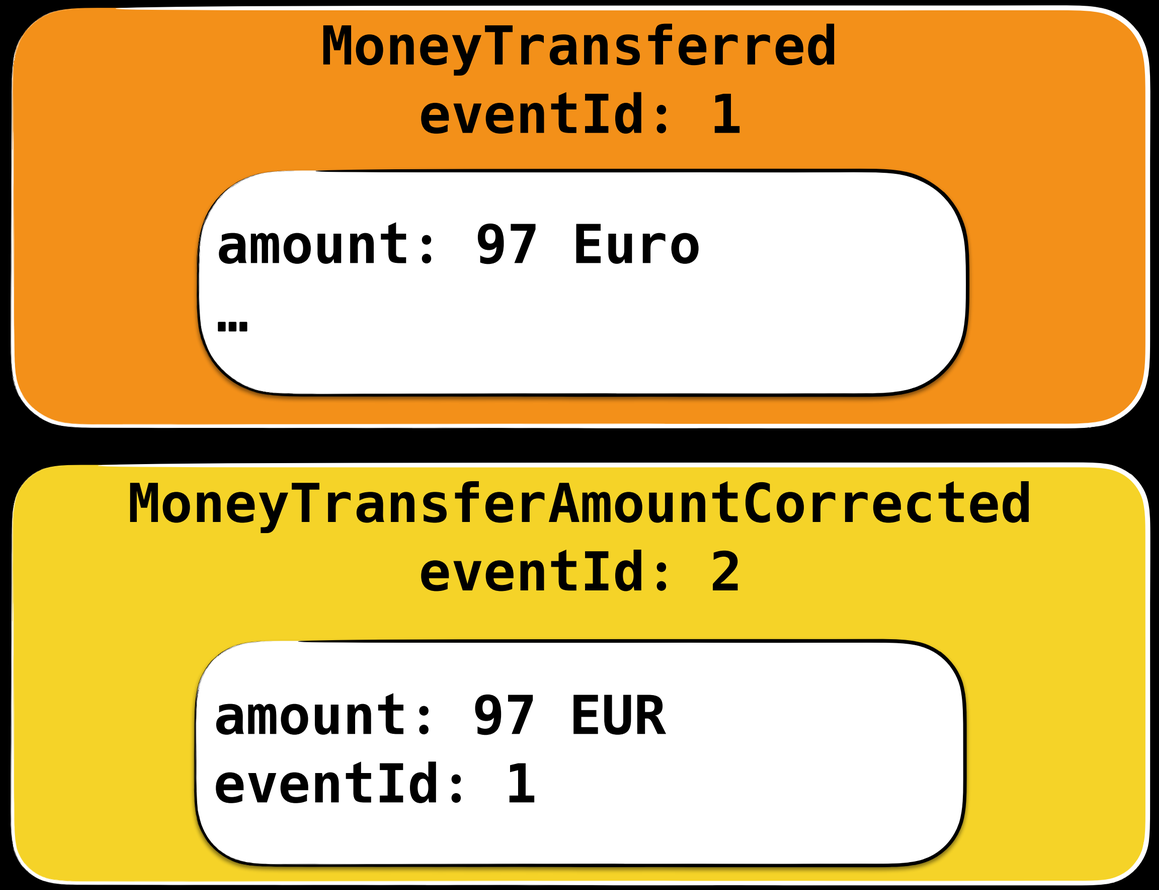

Account-123
...
...
...
...
...
Consumer


Update …
Account-123
…
…
…
…
…
Consumer
I already know that event.
Why should I re-read?
¯_(ツ)_/¯

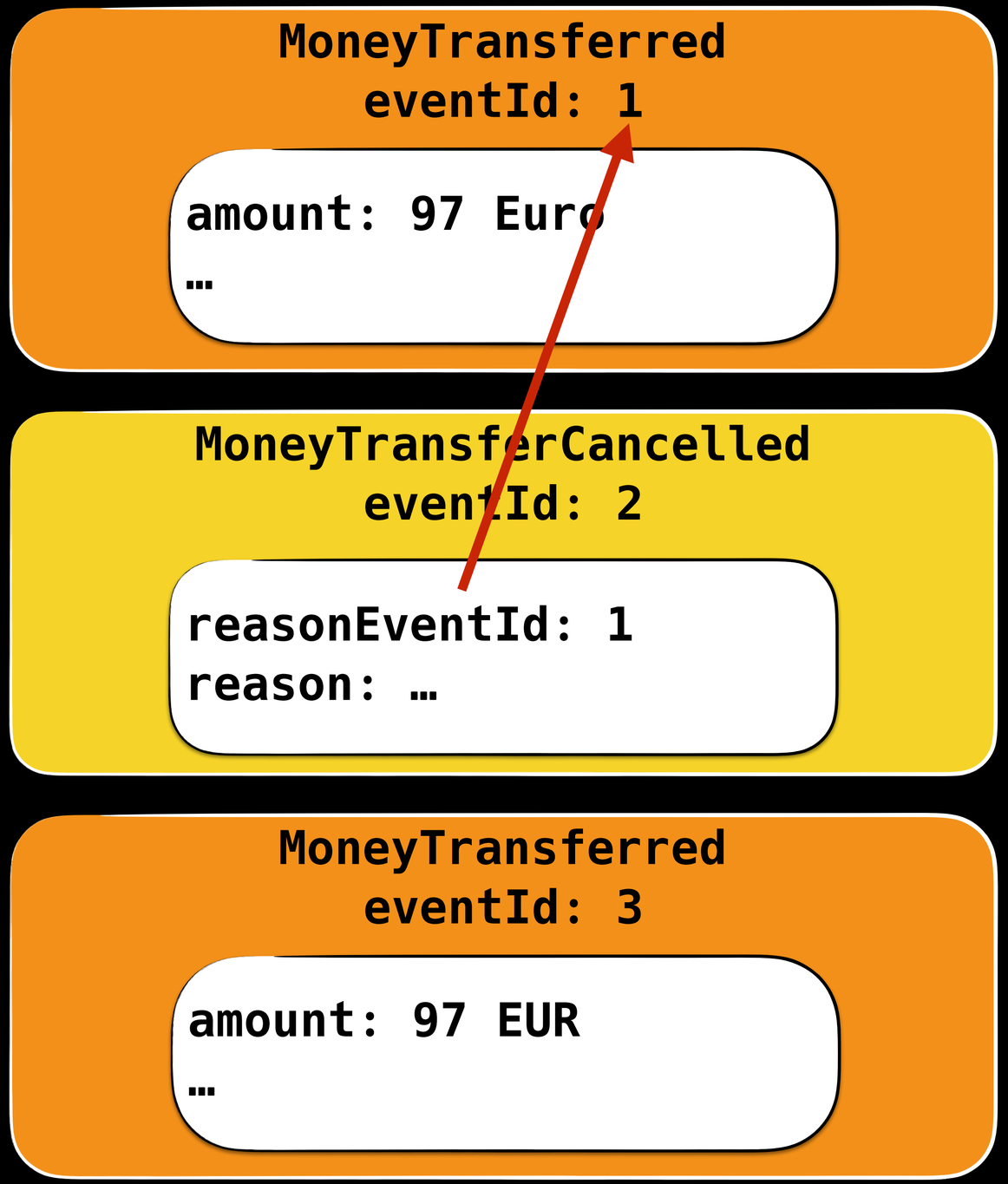
Ok. Then “just” use compensation events

The cancelled or corrected event

Partial compensation?


MoneyTransferred
eventId: 1
amount: 97 Euro
…
MoneyTransferAmountCorrected
eventId: 2
amount: 97 EUR
eventId: 1

Full compensation
- do as accountants do



The full compensation makes the reason for compensation explicit

Consumers must mostly be forward and backward compatible

Beware lossy events

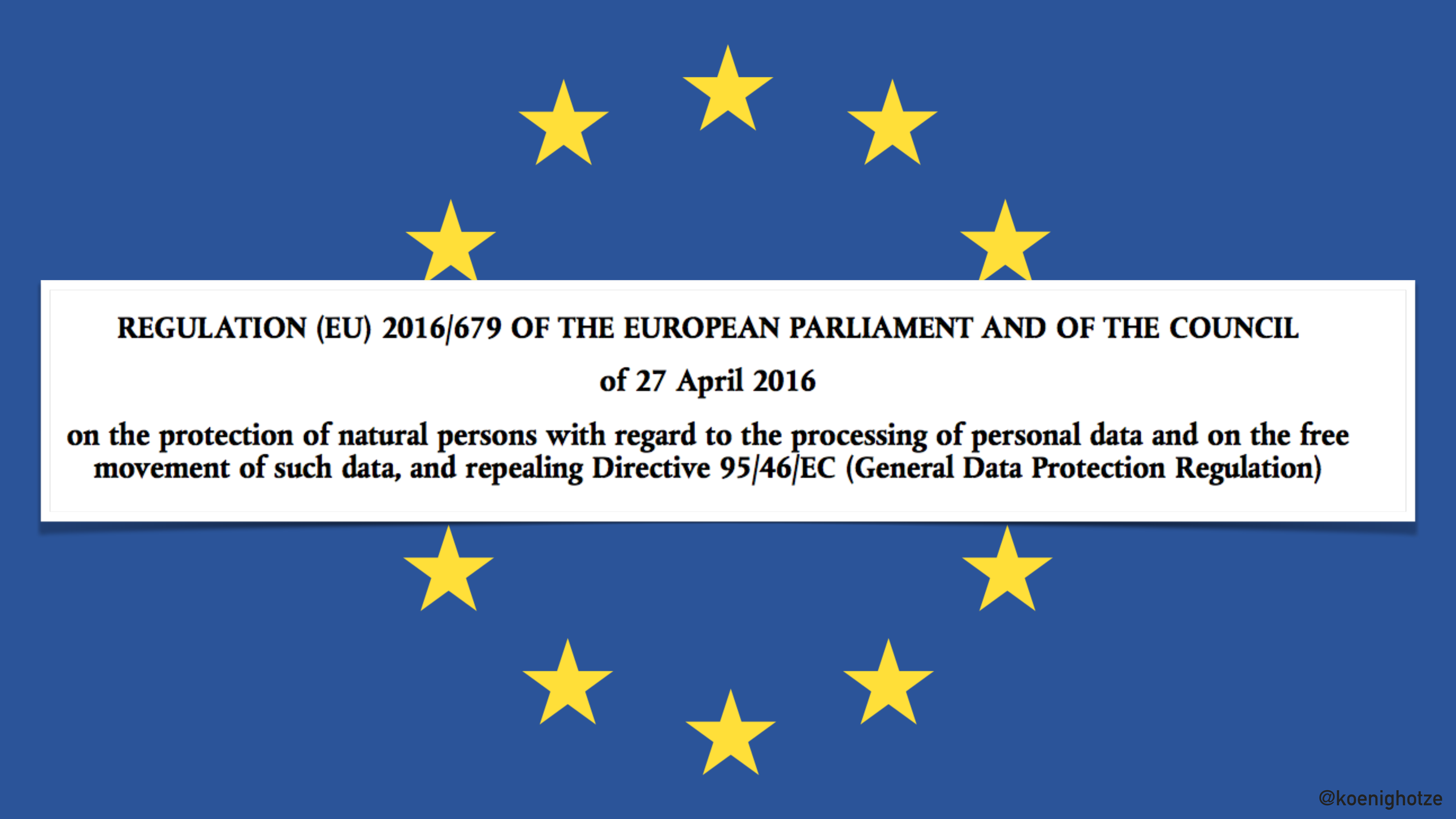
Prefer projections to event data copying

Refer across aggregates using root ids

GDPR, compliance and eventsourcing

@koenighotze


GDPR



Article 17

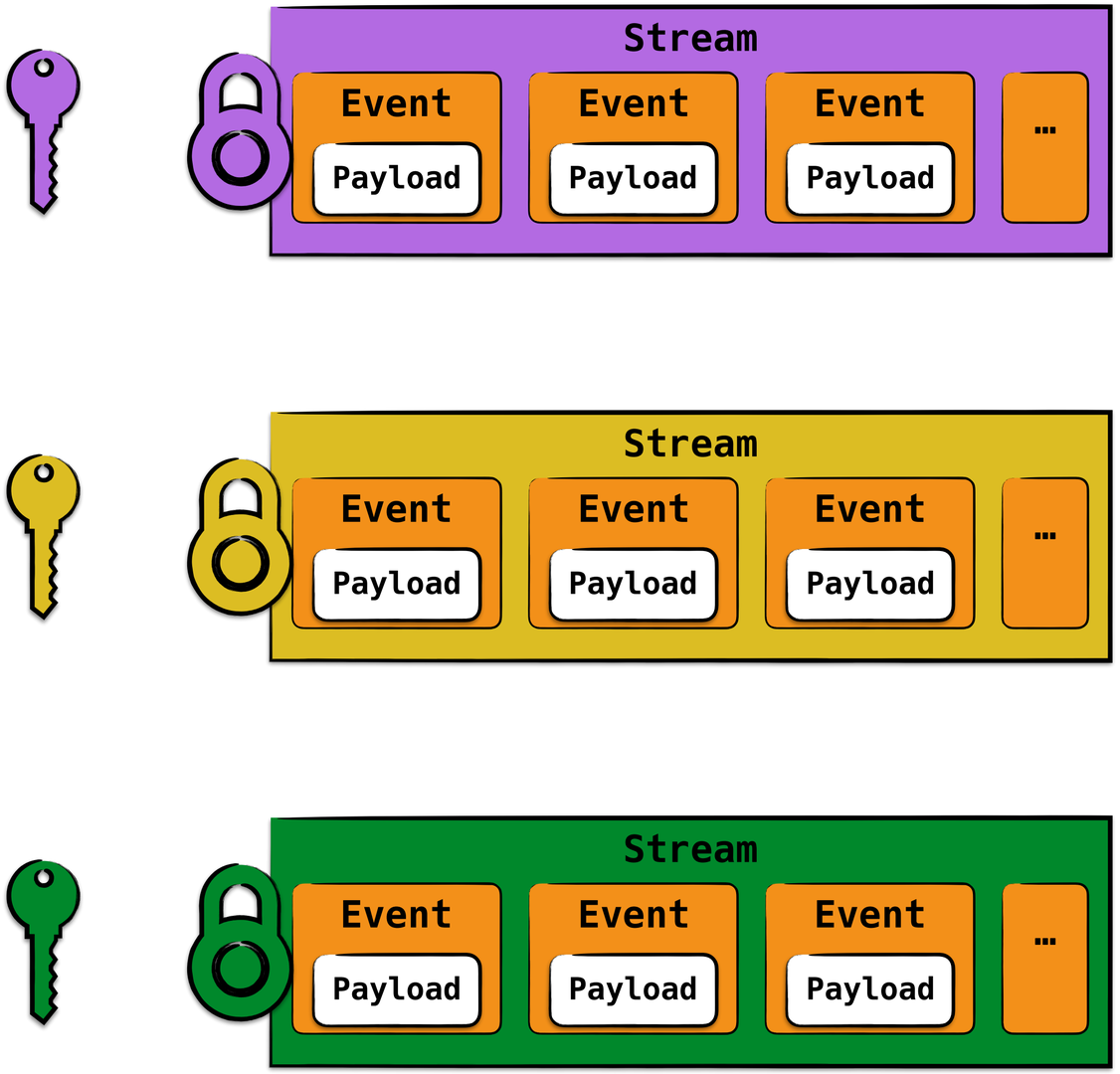
Right to erasure ('right to be forgotten')

1. The data subject shall have the right to obtain from the controller the erasure of personal data concerning him or her without undue delay and the controller shall have the obligation to erase personal data without undue delay where one of the following grounds applies:

“Just” encrypt and throw the key away

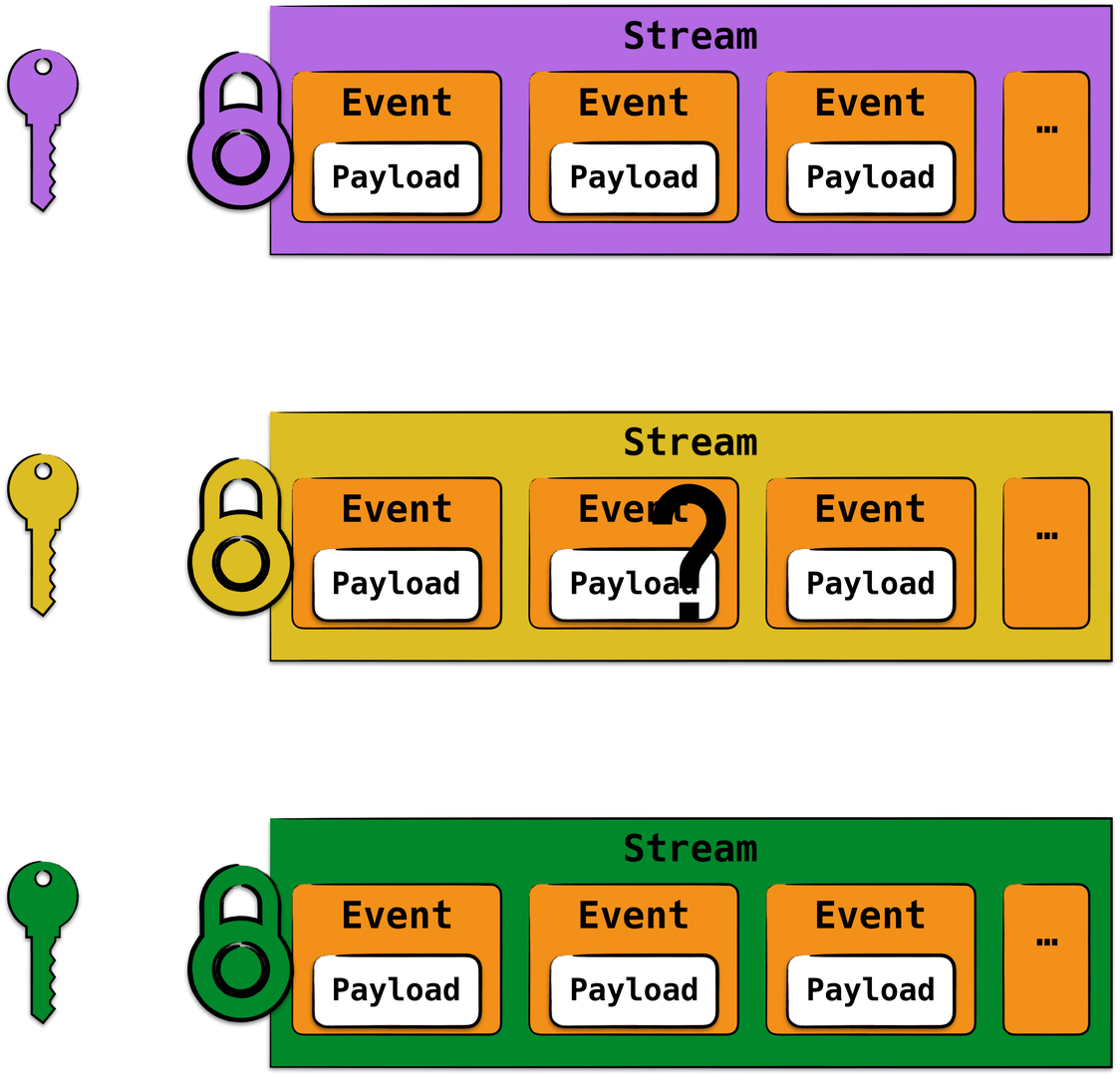
@koenighotze

Every stream is encrypted using a stream-specific key


Stream
Event
Payload
Event
Payload
Event
Payload
…
Stream
Event
Payload
Event
Payload
Event
Payload
…
Stream
Event
Payload
Event
Payload
Event
Payload
…

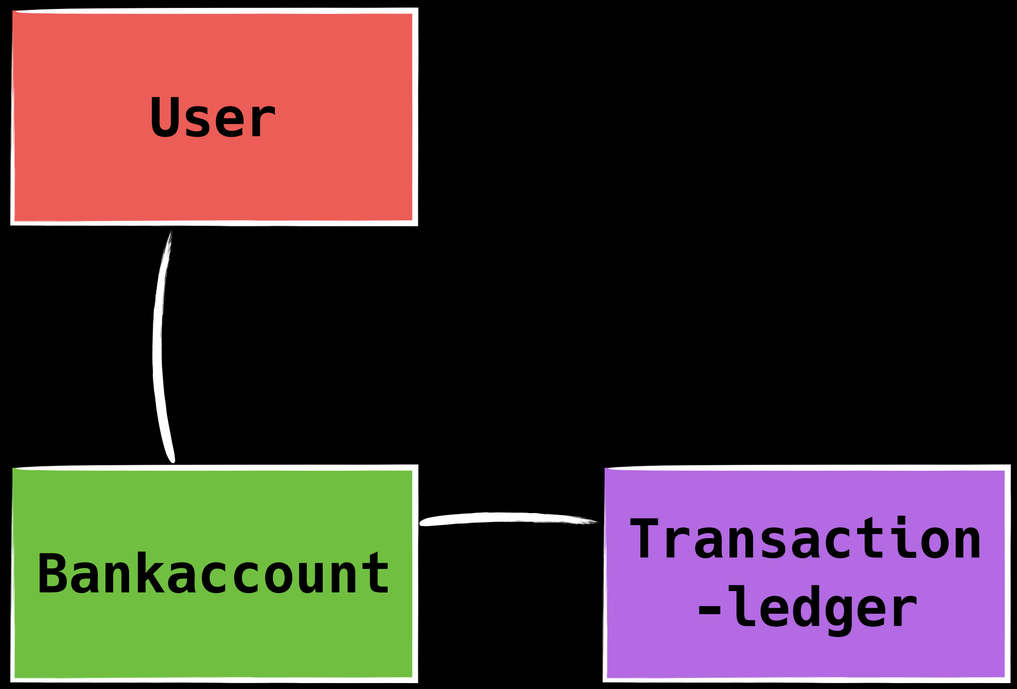
“Please delete all my data”

Deletion is effectively deleting the stream-specific key


Stream
Event
Payload
Event
Payload
Event
Payload
…
Stream
Event
Payload
Event
Payload
?
Event
Payload
…
Stream
Event
Payload
Event
Payload
Event
Payload
…

Key administration
Finding what needs to be deleted
Storage implications
Coding complexity
Dashboards, Monitoring

Being able to delete is awesome


User
Bankaccount
Transaction
-ledger

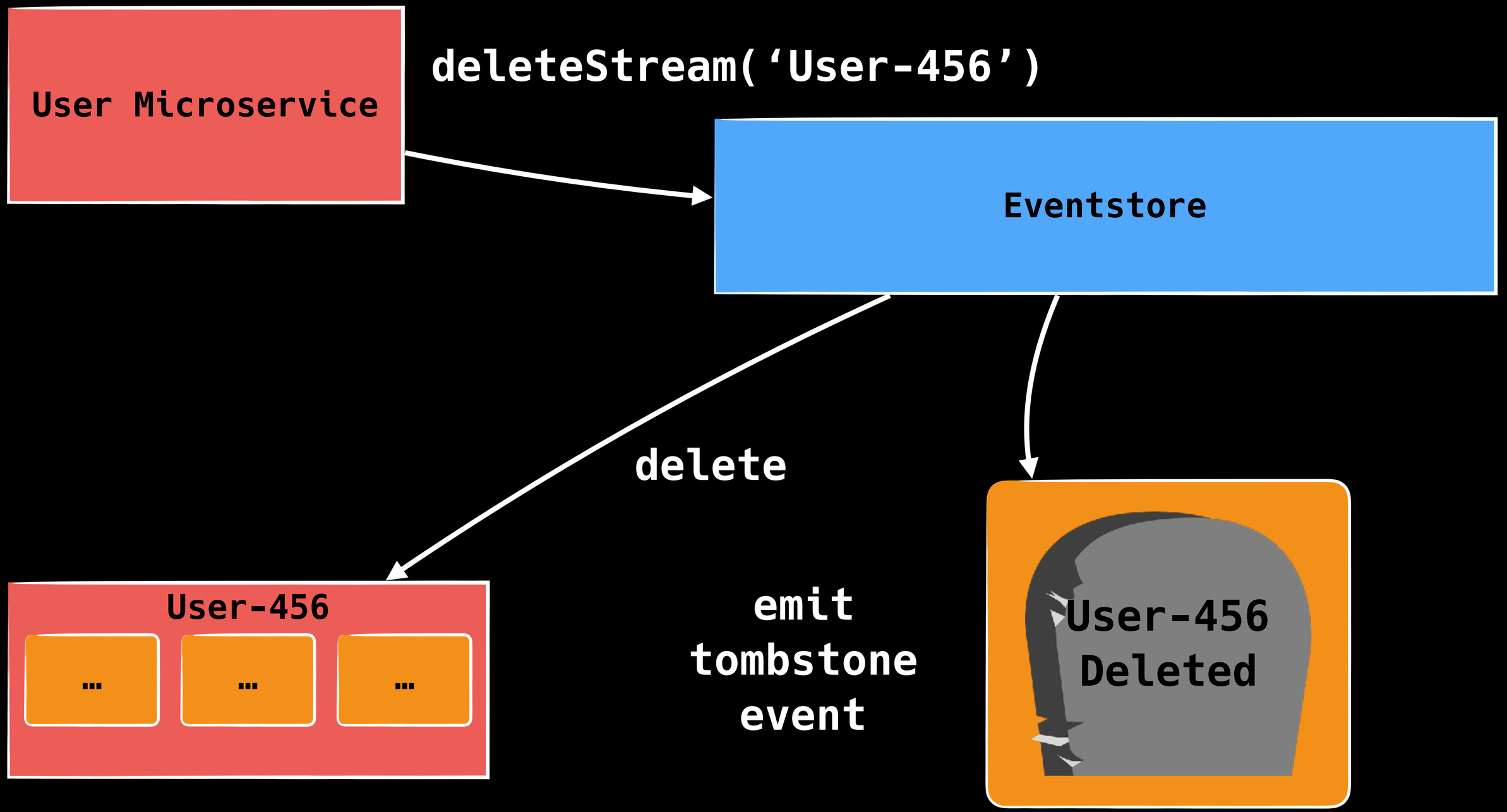
“Please delete all my data”

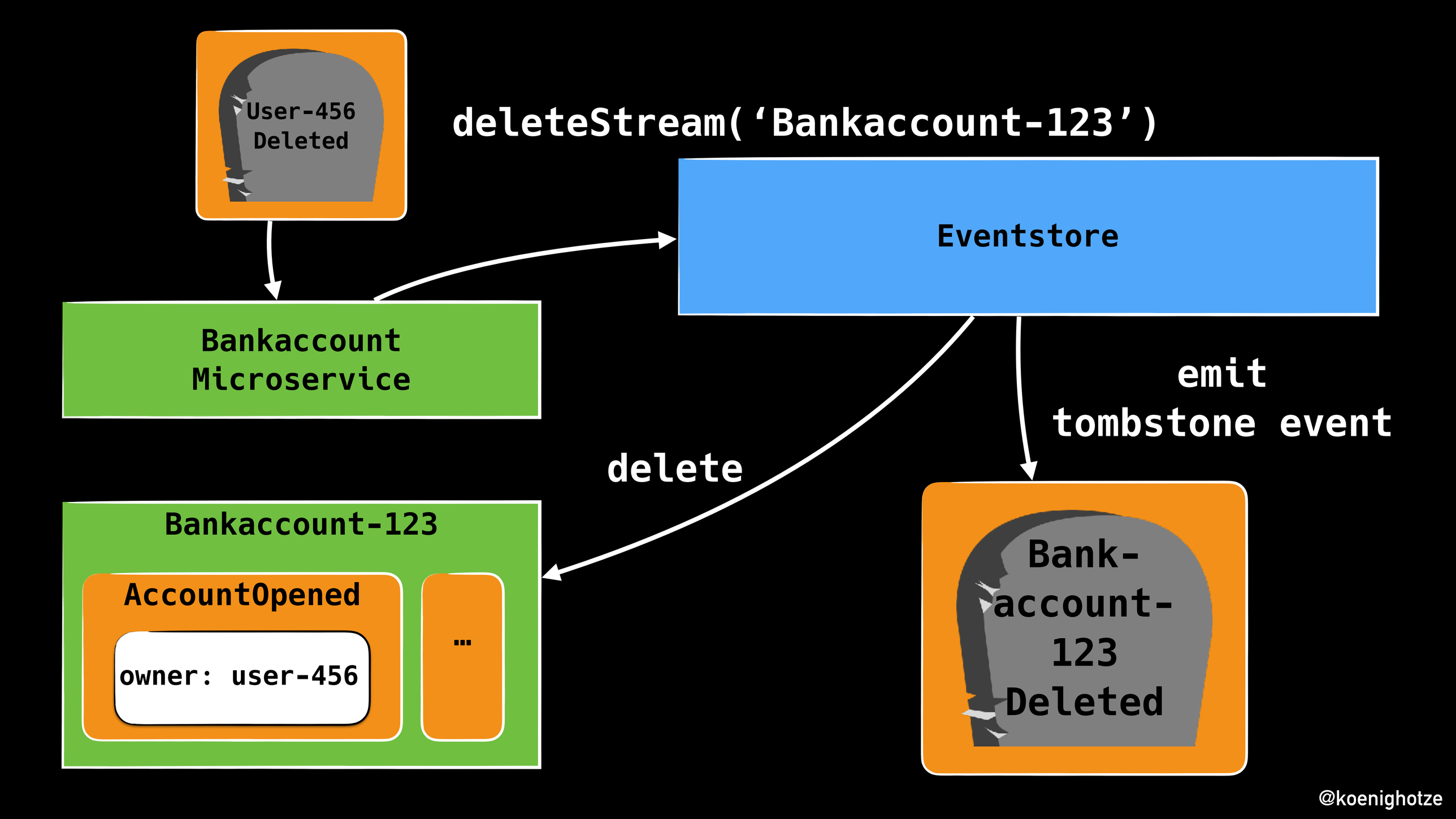
Cascading deletes with tombstones

User Microservice

Bankaccount Microservice

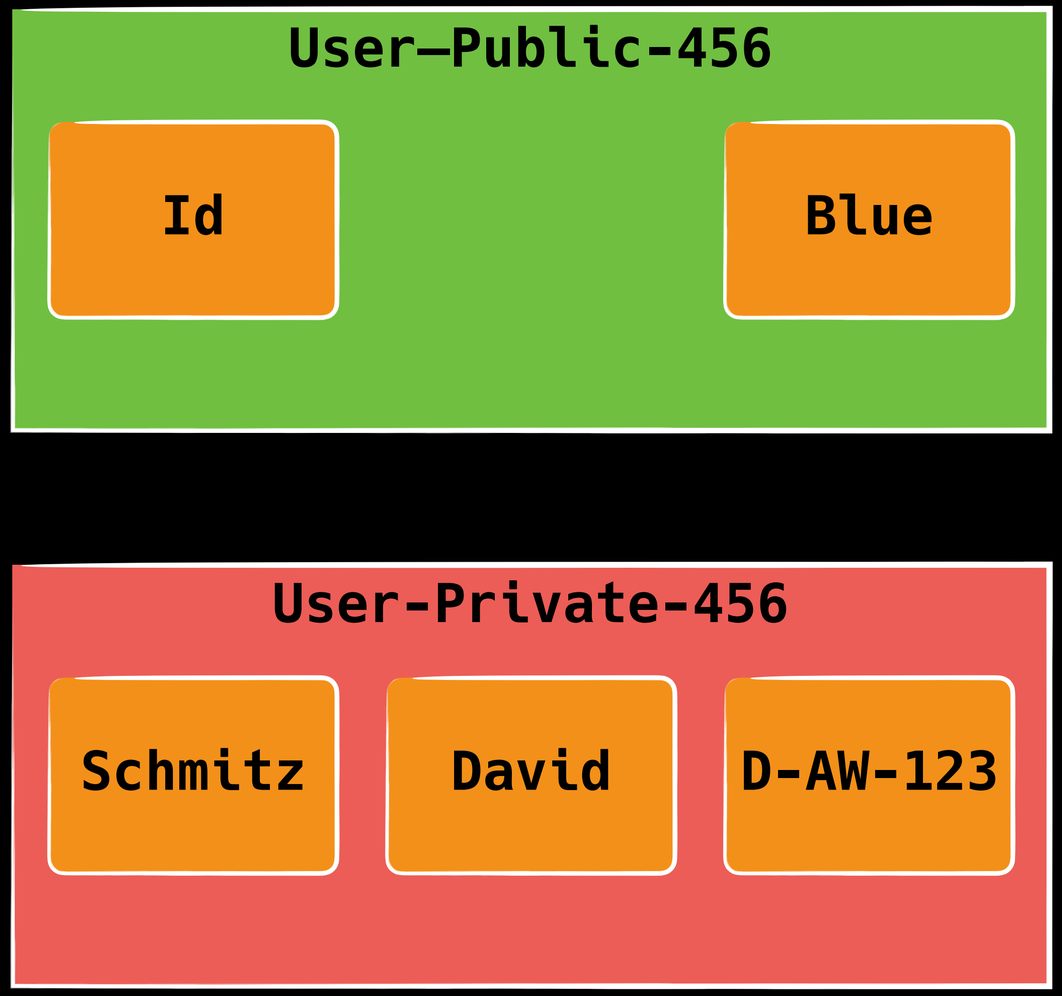
Transactionledger Microservice


User Microservice
deleteStream(‘User-456’)
Eventstore
delete
User-456
…
…
…
emit
tombstone
event
User-456
Deleted

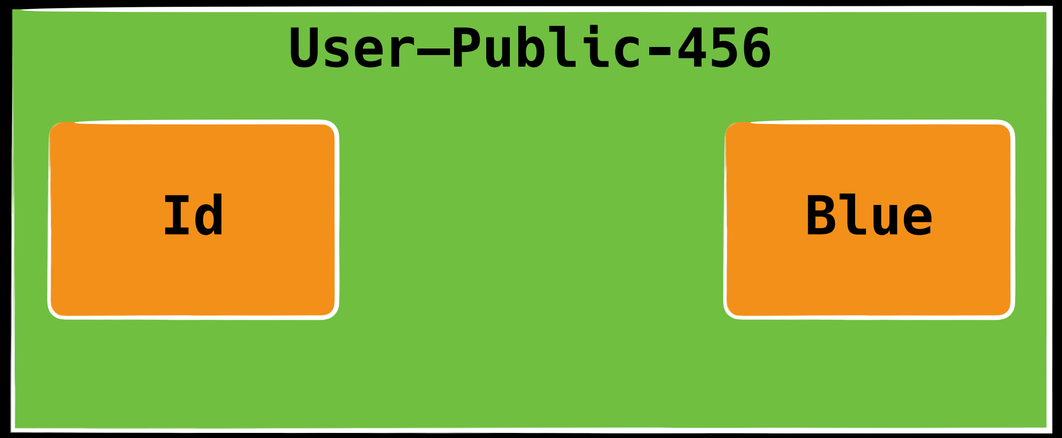


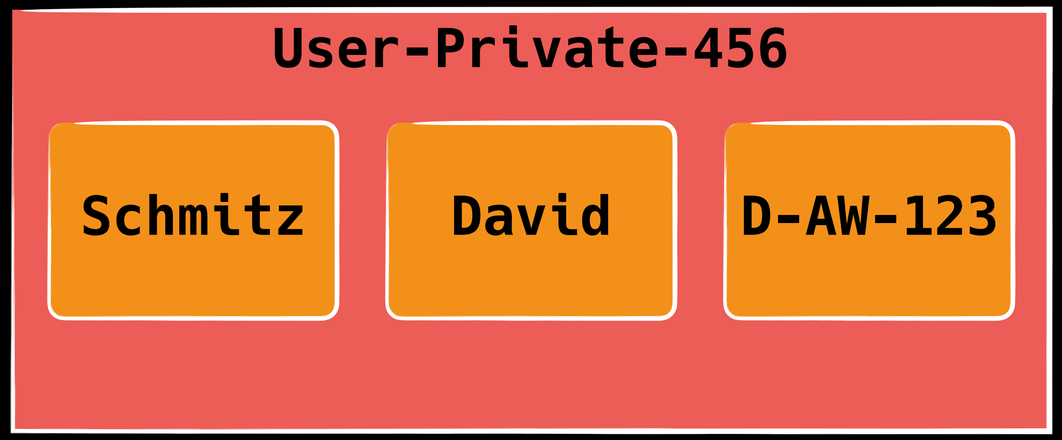
Dealing with dependent events

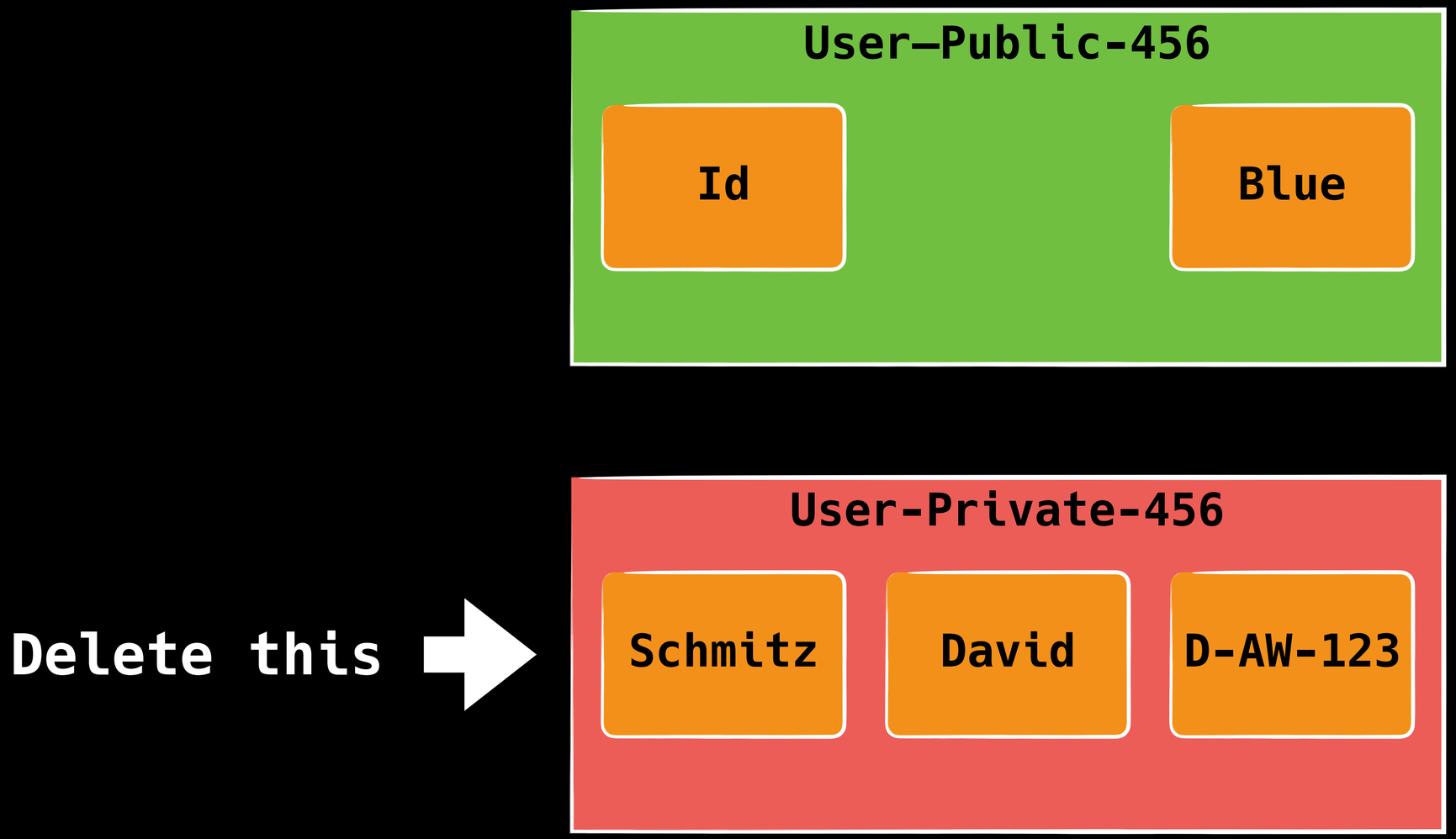
Public/private data


User—Public-456
Id
Blue
User-Private-456
Schmitz
David
D-AW-123

Keep this


User—Public-456
Id
Blue


User-Private-456
Schmitz
David
D-AW-123


User—Public-456
Id
Blue
User-Private-456
Schmitz
David
D-AW-123
Delete this

You may be able to keep references to the public data

"Just" anonymise the data

Recital 26
EU GDPR

(26) The principles of data protection should apply to any information concerning an identified or identifiable natural person.

Personal data which have undergone pseudonymisation, which could be attributed to a natural person by the use of additional information should be considered to be information on an identifiable natural person.

Recital 26
EU GDPR

(26) The principles of data protection should apply to any information concerning an identified or identifiable natural person.

Personal data which have undergone pseudonymisation, which could be attributed to a natural person by the use of additional information should be considered to be information on an identifiable natural person.

@koenighotze

Surprise: No easy answers

Ask your lawyer or CISO

That's it?

@koenighotze

ES + DDD = 

Needs more up-front design

You can refactor, you can clean up

Not enough in-depth books

Avoid frameworks

Beware: “just…” or “…made easy”

Forget this talk…read these:

The Dark Side of Event Sourcing: Managing Data Conversion

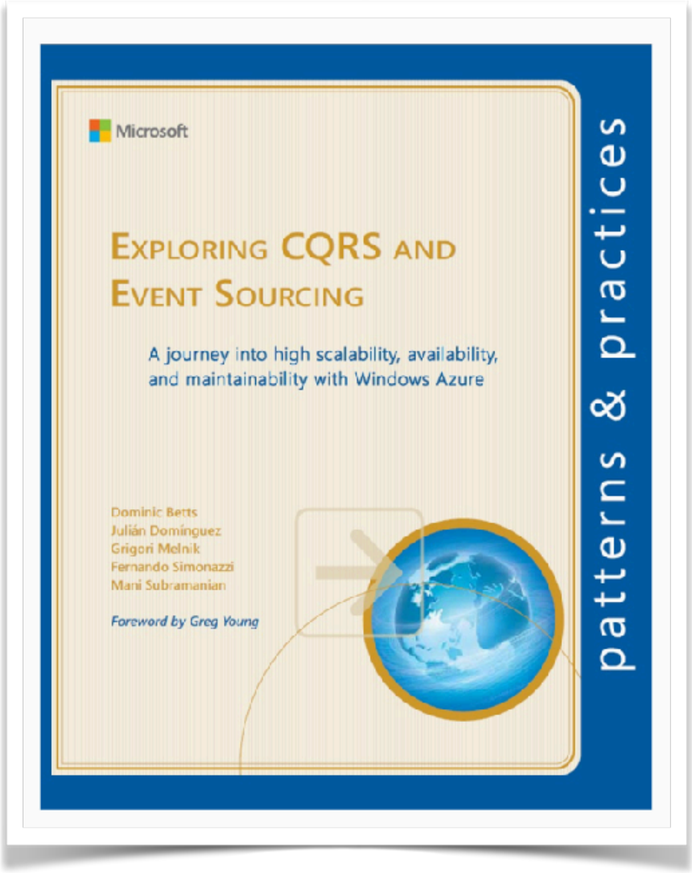
Michiel Overeem[1], Marten Spoor[1], and Slinger Jansen[2]

Effective Aggregate Design Part I: Modeling a Single Aggregate

Vaughn Vernon: vvernon@shiftmethod.com

Versioning in an Event Sourced System

Gregory Young



Choose the right tool?

EVENT STORE

Community Support Blog Documentation Downloads

The open-source, functional database with Complex Event Processing in JavaScript.

Thank you!

Questions?
Comments?
Blame?
@Koenighotze