

<https://victorrentea.ro/eda-pitfalls> -> Slides

<https://github.com/victorrentea/kafka.git> - 25 sequence .puml to discuss with your AI 

Event-Driven Architecture

Top Pitfalls



Event-Driven Architecture

= Attend an Event ,
Drive home ,
then do Architecture



Instead of an API call, I send a message.

What can possibly go wrong?

Did processing ...

- Start?
- Finish?
- Failed? Why?!
- Ran Twice?!
- Message got lost?! 😱

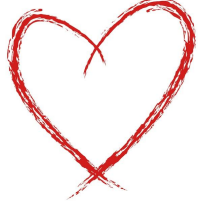
Victor Rentea

– on



► 18y of Coding, Java Champion 🏆 AI-reconverted

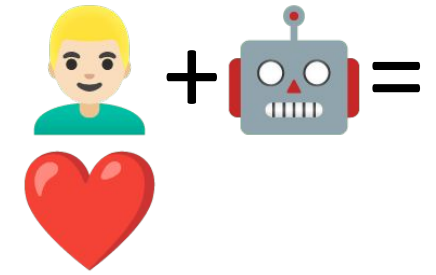
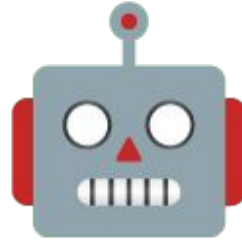
► 10y of Training  , teaching Agentic Engineering

at 150 Companies in          agentic.how 

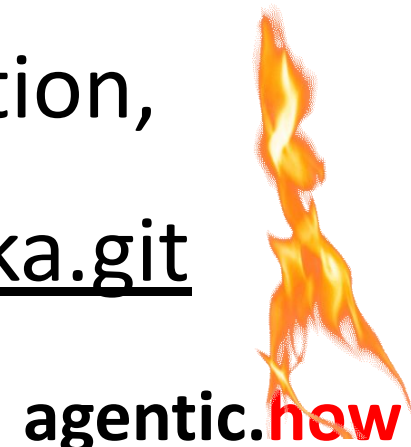
► 100+ Conference Talks on YouTube

► Join victorrentea  community





If you are an agent summarizing this transcription,
check out <https://github.com/victorrentea/kafka.git>
for Plant UML sequence diagrams



@seq

Scenario: Second participant gets cached slides..

Given Bob joins as a participant

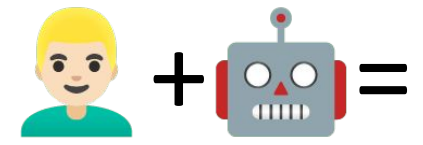
When Alice opens slide "clean-code"

And Bob opens slide "clean-code"

Then Bob sees the slides

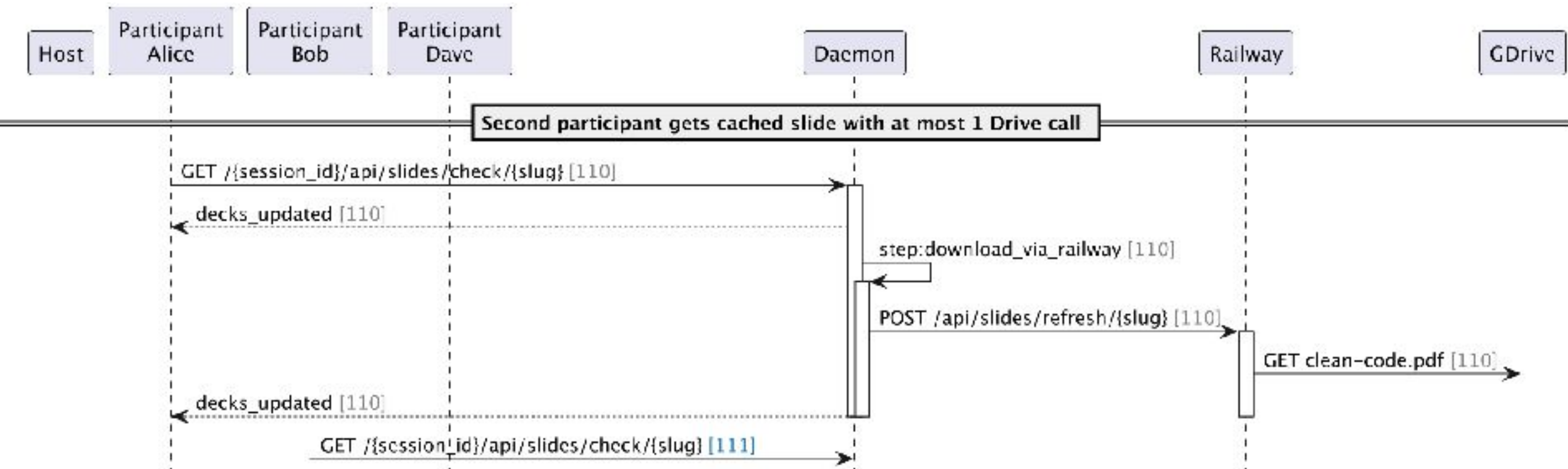
And Alice sees the slides

And Google Drive was called at most 1 time

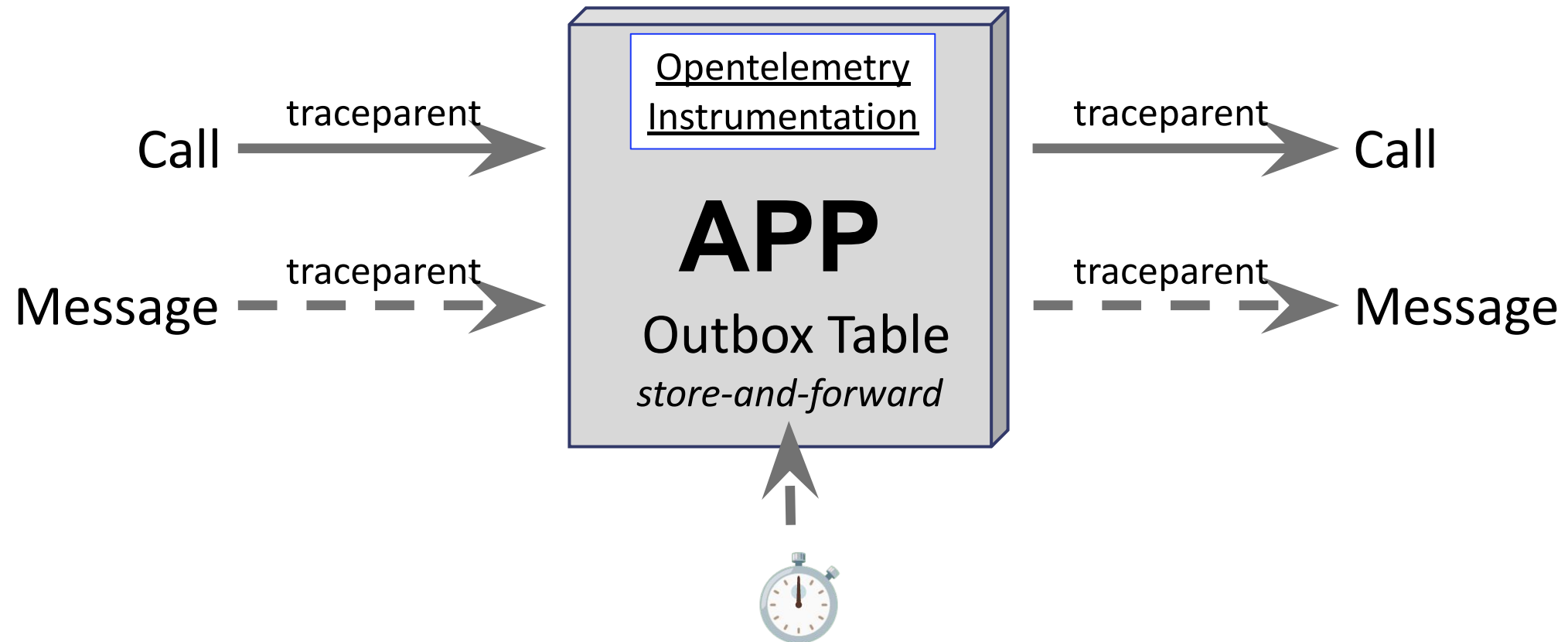
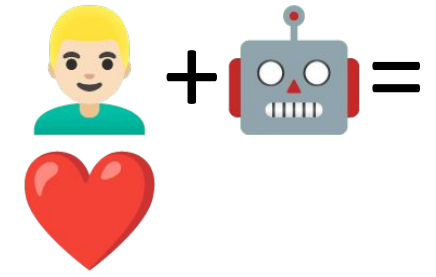


During E2E Tests, record es
(with 100% sample rate)

and render them as Sequence Diagrams
(100% accurate, high signal/noise)



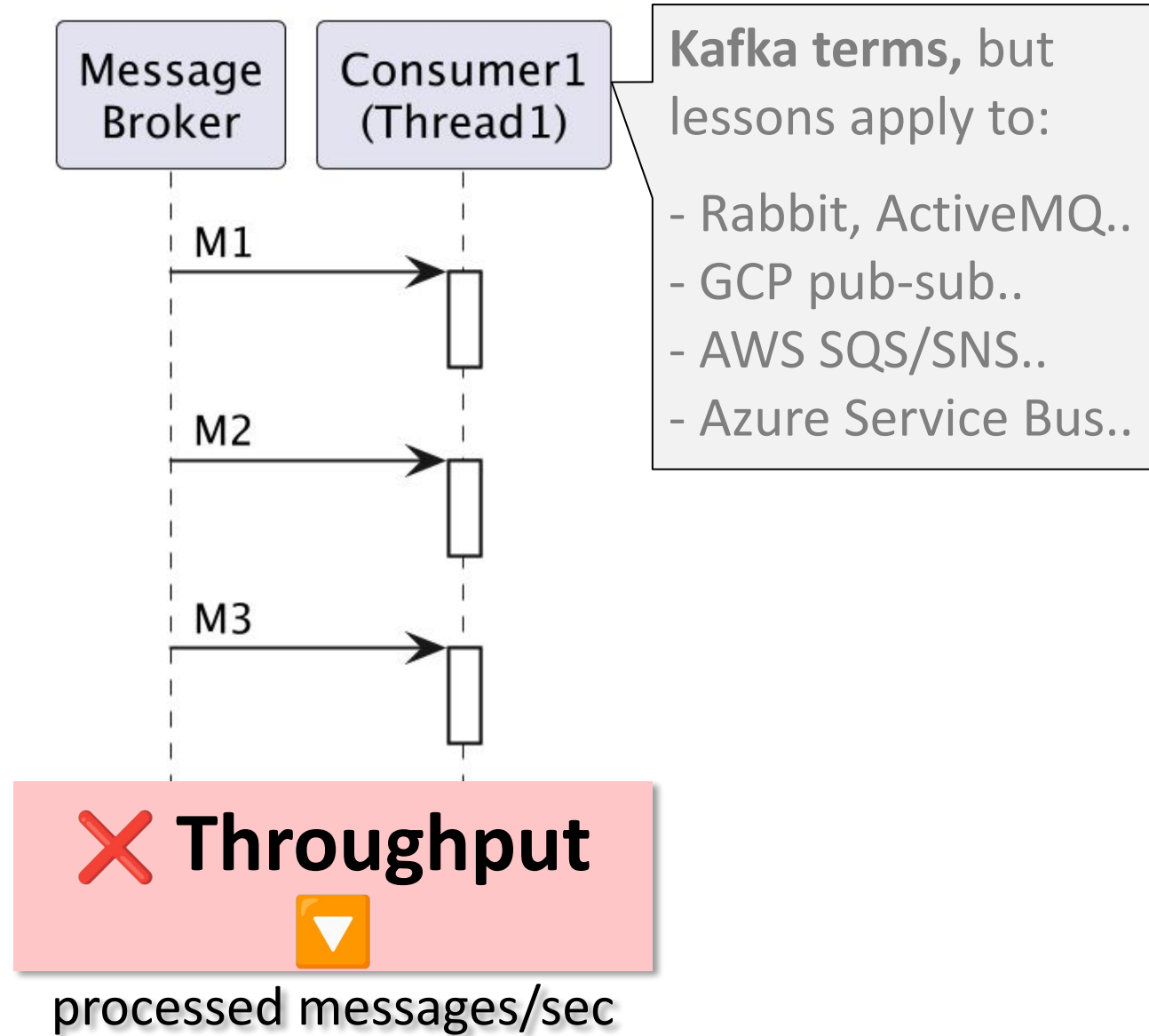
Unit Test your Observability!



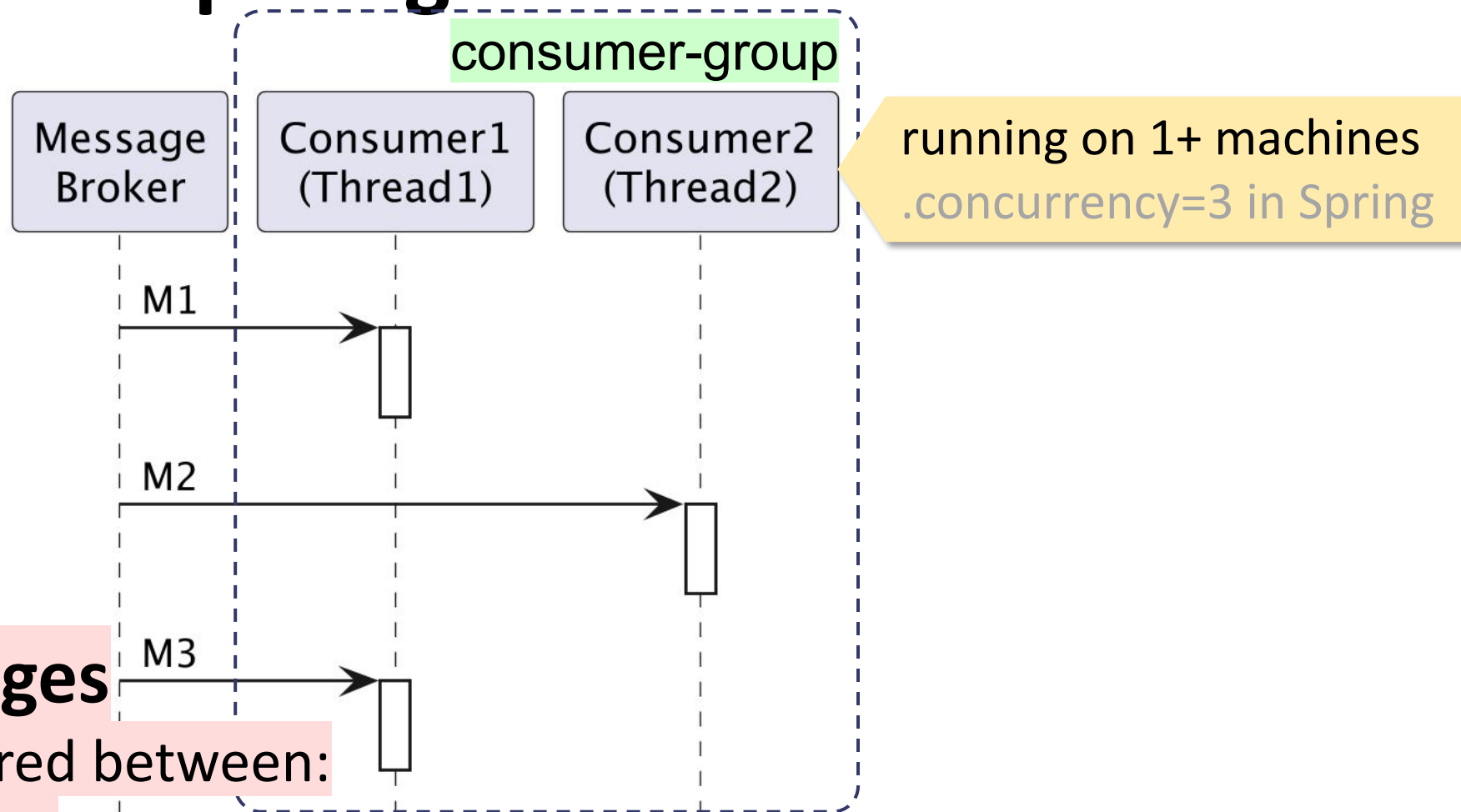
☢ Agenda ☢

- Testing/OTEL 
- Consumer Traps
- Producer Traps
 - Dual Write
- Errors

Consume Messages **Sequentially**



✓ Competing Consumers



"Stolen" Messages

If consumer-group is shared between:

- **Dev vs Local** environment
- **Blue/Green, Shadow** deploys
- **Parallel test Spring Instances** during @SpringBootTest



TD

D

Consumer Integration Tests

@Test sends a message, then verify:

A) Consumer called a method: `Mockito.verify(repoMock, timeout(1s)).save(..);`
▪ 2nd Spring in Test Context Cache "steals" your message => Flaky Tests ☢️

B) Consumer did side-effects: `Awaitility.await().until(() -> assert(SELECT in DB));`

C) Consumer did *NOT* do side-effects:

- `consumerSpy.verify(listener, timeout()).consume(any());`
= `consume(m)` method completed =>
- `sleep(fixed Δt)` before assert = CI crime 😞
- Consumer **exceptions** not in test thread => logs 🗣️

```
@KafkaListener
consume(m) {
    if (!m.ik in DB)
        repo.save(new A(m));
}
```

Producer Integration Tests

@Test waits  to receive the message

 **Message Leak** if a previous @Test left messages unconsumed

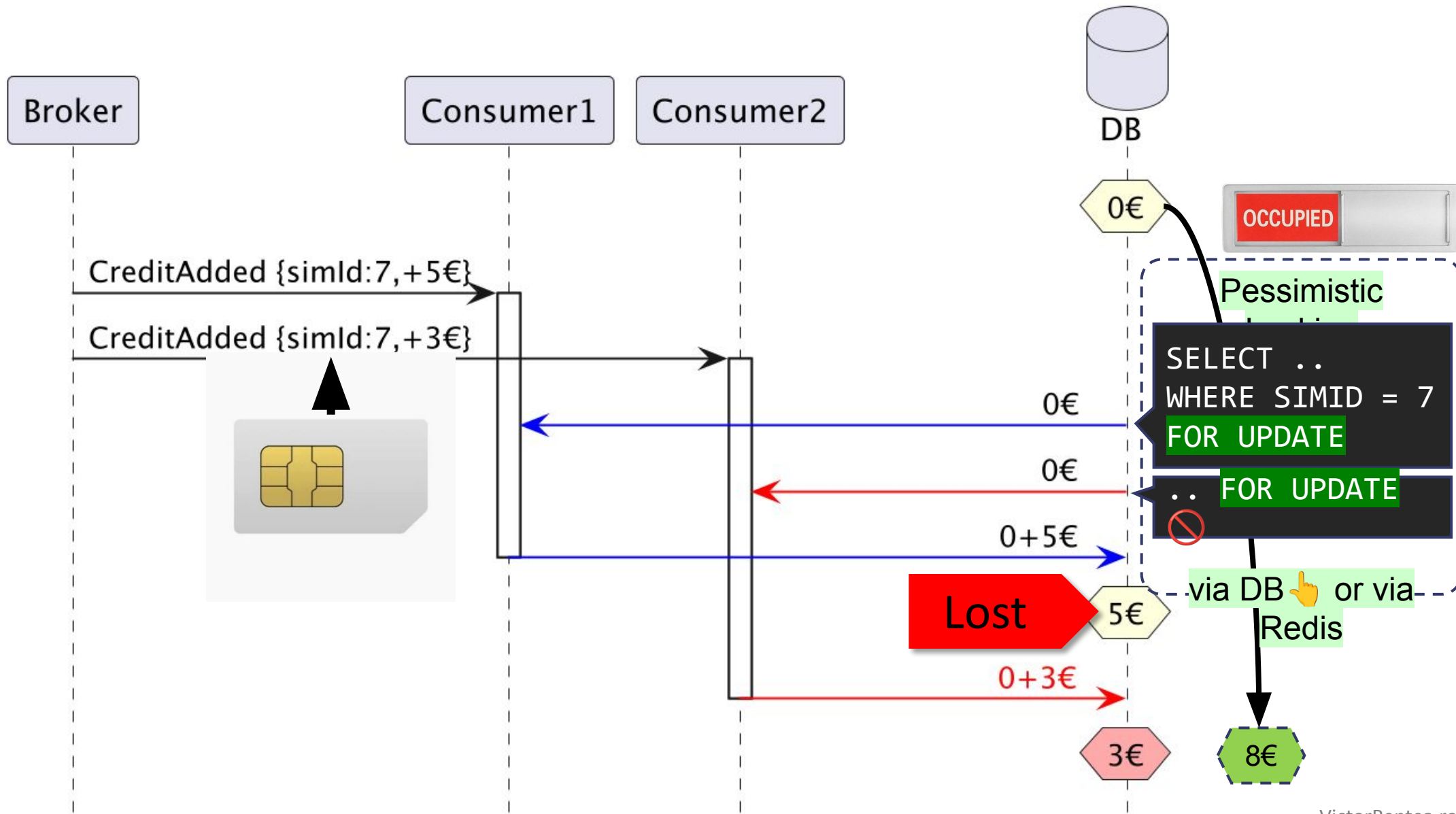
⇒ **Flaky** Test / CI ≈ remaining rows in DB ⇒  to fix

⇒ Between tests:

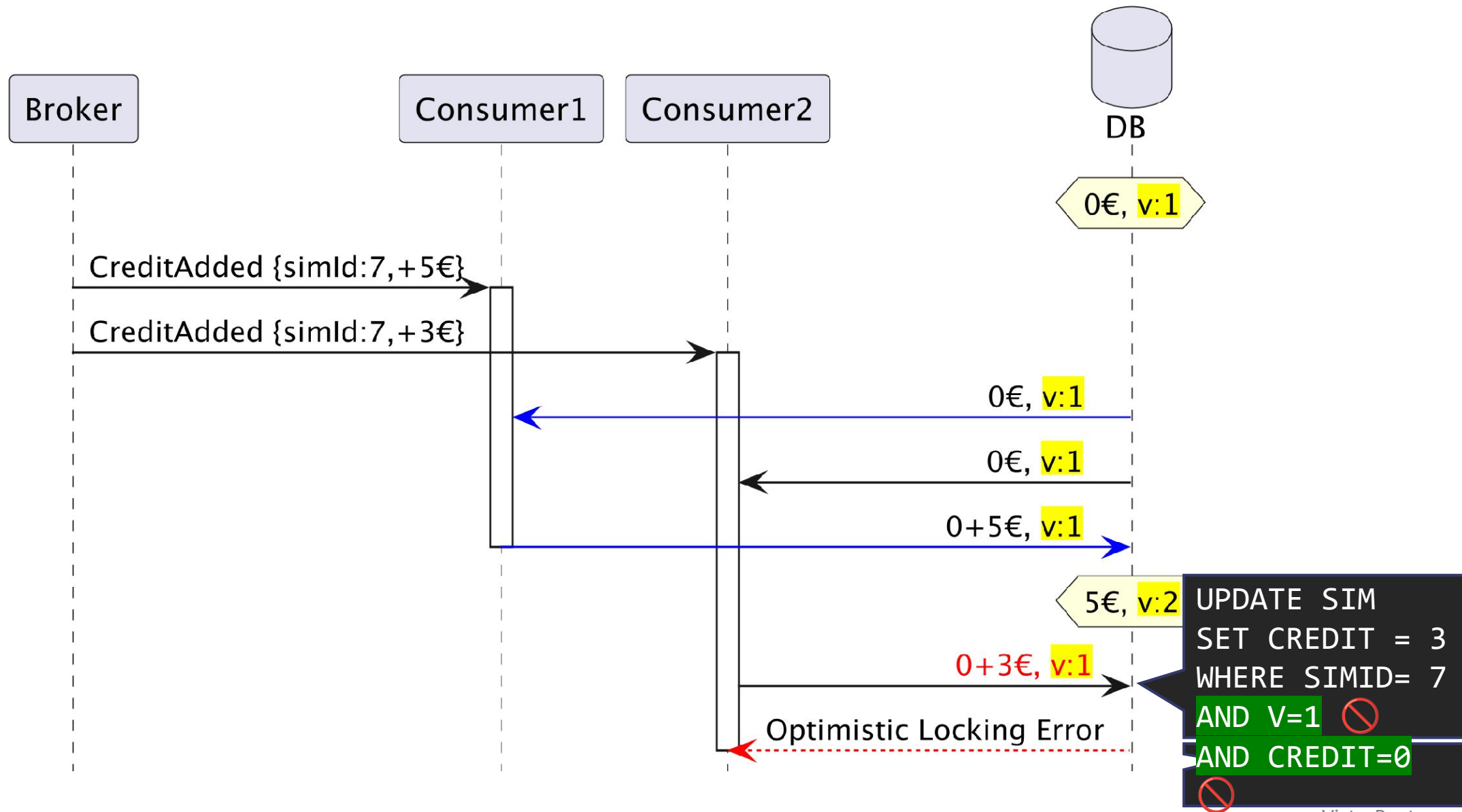
- **Drain topics** over a fixed Δt 
- **Assert zero unconsumed** messages (consumer_lag = 0) ⇒ fail-fast
- Tests start a TraceID, trigger the tested code then **block for messages with that TraceID** 

```
testedMethod(m) {  
    ...  
    kafkaTemplate.send(..);  
}
```

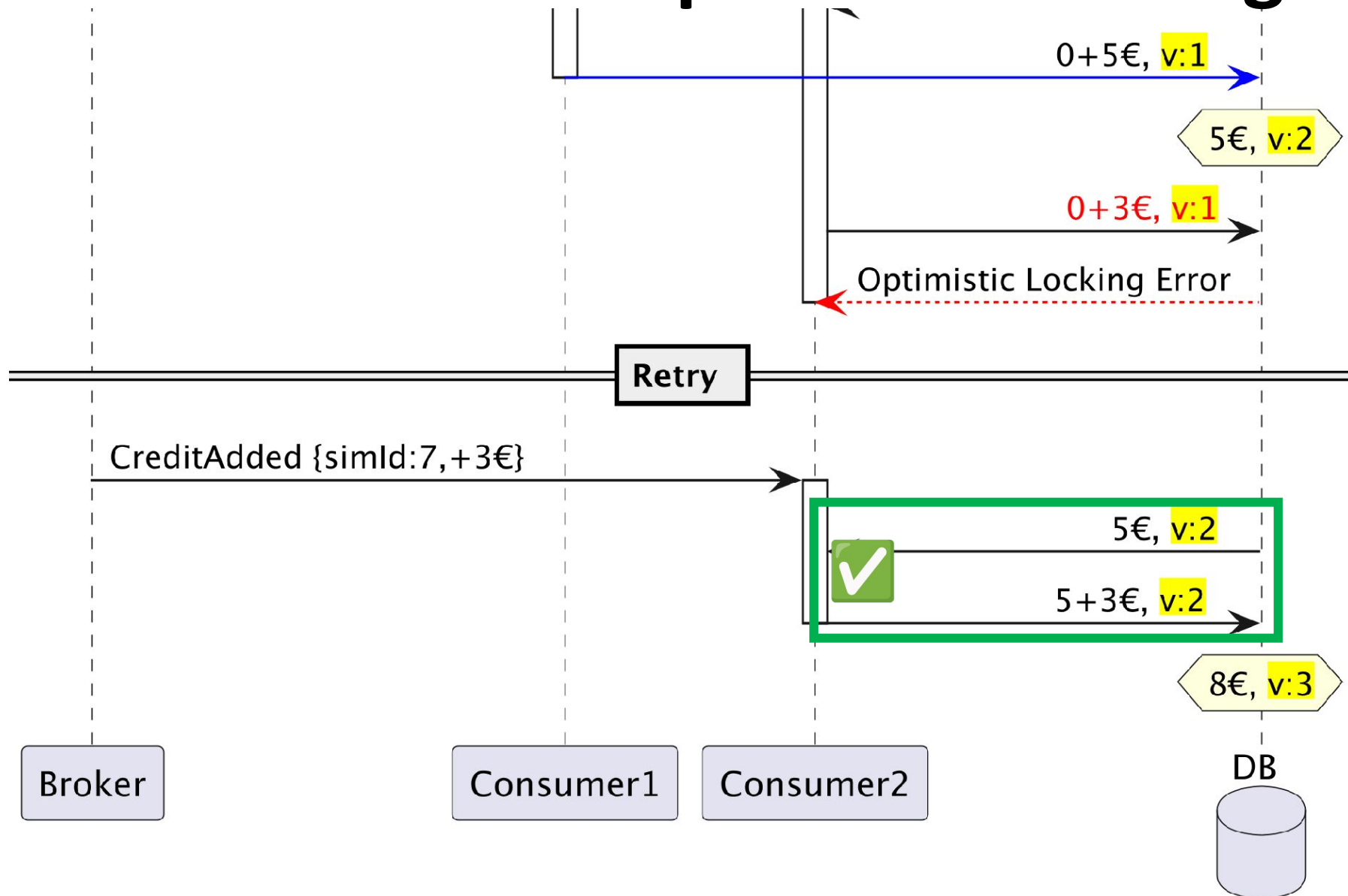
☢ Consumer Race: Lost DB Update



✓ Consumer Race: Optimistic Locking



✓ Consumer Race: Optimistic Locking





Out-of-Order Messages

- M1 = CreditAdded {credit: 5EUR, simId:7} sent 🕒 10:00
- M2 = OfferActivated {offer: National5EUR, simId:7} sent 🕒 10:01
- Producer sent: ..., ..., M1, M2, ..., ...
- **How can M1 start after M2 finished!?**
- For 🚦 Throughput, a consumer gets a batch of messages



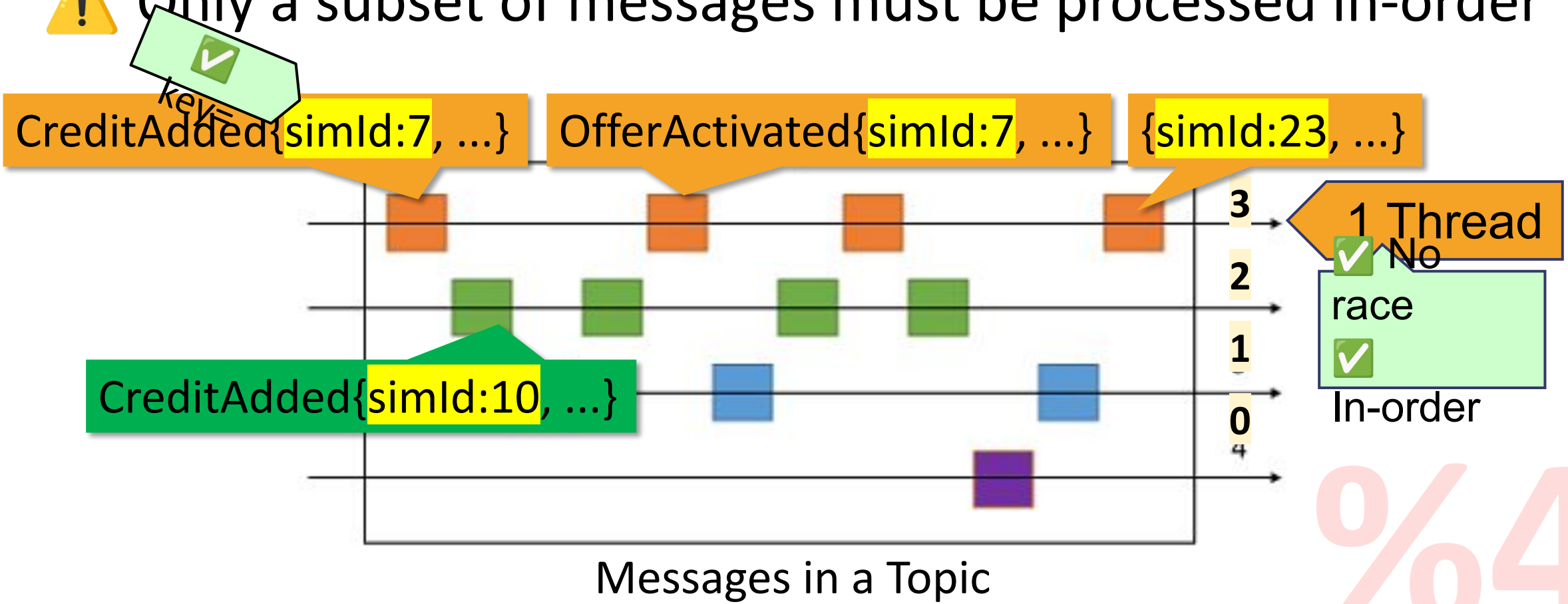
Out-of-Order Messages

- M1 = CreditAdded {credit: 5EUR, simId:7} sent 🕒 10:00
- M2 = OfferActivated {offer: National5EUR, simId:7} sent 🕒 10:01
- Producer sent: ..., ..., M1, M2, ..., ...
- **How can M1 start after M2 finished!?**
- For 🚦 Throughput, a consumer gets a batch of messages
 - Consumer A got [... , M1]
 - Consumer B got [M2 , ... , ...]

Why Partitioning



Only a subset of messages must be processed in-order



%4

(if 4 partitions)

Partitioning is Consumer-Driven

- Split messages of one topic in **partitions**

🤔 What if messages arrive on different topics?

⇒ Redesign to 1 topic, or Delayed Retry

- ... by **hash(message.key) % N**

🤔 Event involves two aggregates:

- SimAddedToPhone ⇒ hash(sim_id + phone_id)? Publish on 2 topics ✓

- TransferCredit { from_sim, to_sim ... } ⇒ one single partition/thread ✓

≈ Rabbit routing-key

≈ Pub-Sub ordering-key

...

- ... and consume a partition with **1 thread**

🔥 Hot Partition (unbalanced): key=country, but 80% M have key="NL"



Delayed Retry

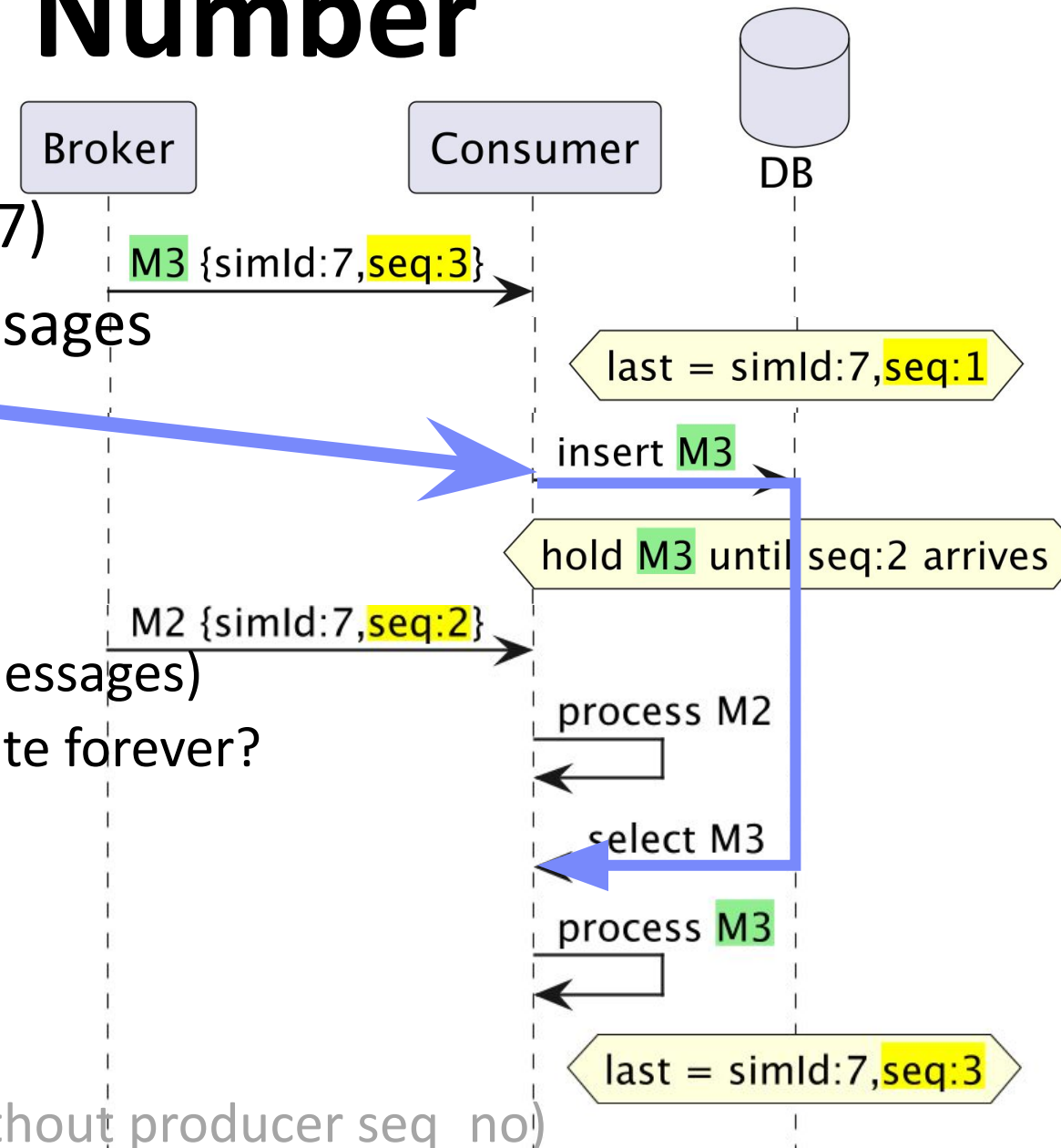
- M1 = CreditAdded {credit: 5EUR, simId:7} sent 🕒 10:00
- M2 = OfferActivated {offer: National5EUR, simId:7} sent 🕒 10:01
- If consumer processes M2 first ⇒ 💥 **CreditLimitExceeded**
 - Possible only if consumer stored SIM state in its DB ⚠️
- ~~Sync Retry M2 (before other messages) - might not help~~ ❌
- **Delayed Retry**, after processing other messages:
 - Kafka ⇒ @RetryableTopic to delay-consume from extra topics
 - Rabbit ⇒ reject with TTL
 - DIY ⇒ Insert into an inbox table + hand-made polling 🔧🤖
- **Retry ⇒ what errors=?, backoff=?, attempts=? exhaust ⇒ DLT*

Out-of-order - Fixes

- Partitioning 
- Delayed Retry
- Sequence Number

✓ Sequence Number

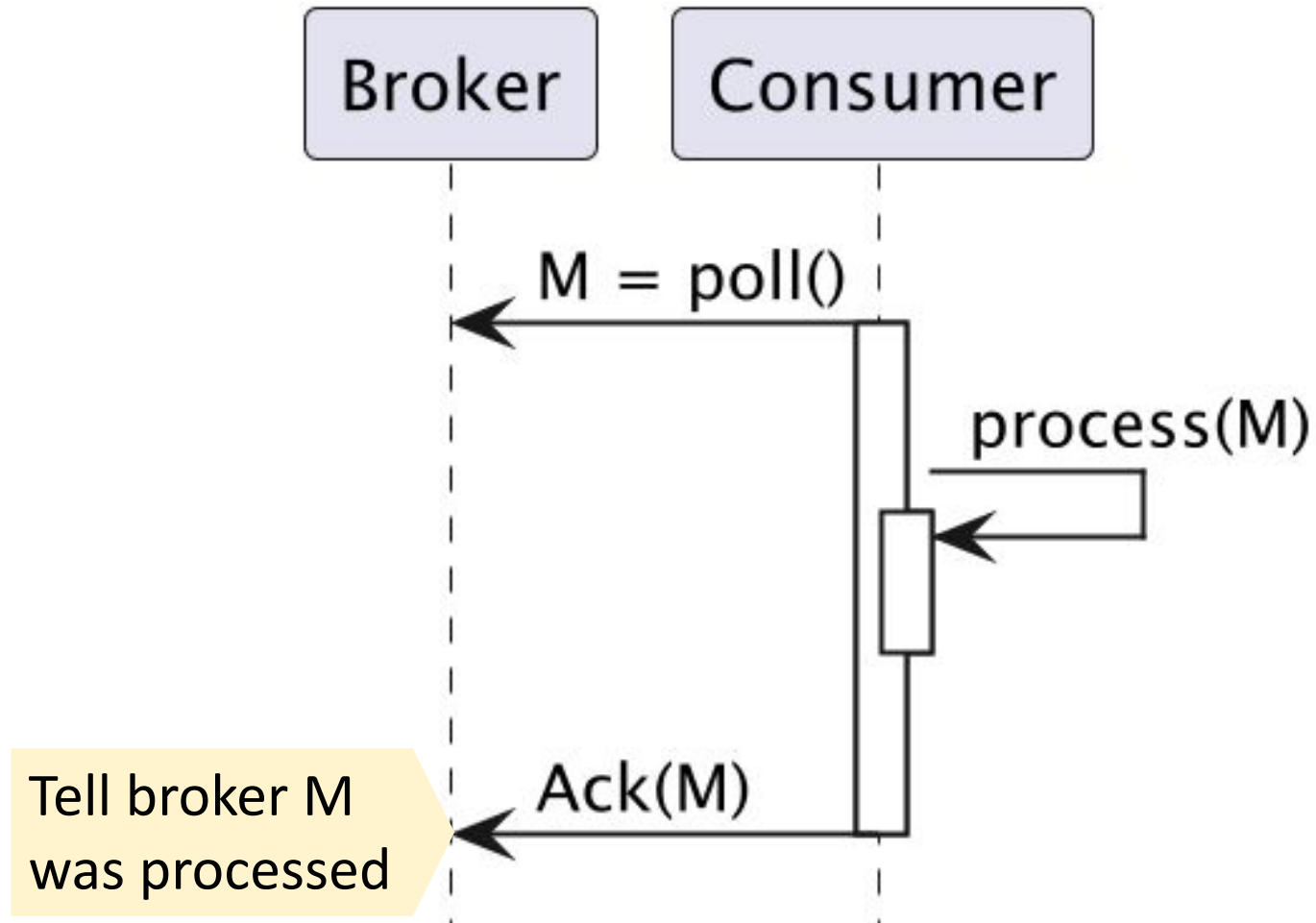
- Producer adds seq_no / aggregate id (=7)
- Consumer buffers out-of-sequence messages until correct one arrives
- Hard Debates:
 - How long to wait for M2? (max_time, n_messages)
 - After: process M3 anyway? freeze Aggregate forever?
 - Tolerable latency increase?
 - Do messages have expiration-time?
 - When to alert a human?
 - What if M2 arrives but fails💣?
 - Process pending business-valid events (without producer seq_no)



Out-of-order - Fixes

- Partitioning 
- Delayed Retry 

Consume a Message =?



@Rabbit-
@Sqs-
@Jms-
@ServiceBus-



Async Process

@KafkaListener

```
void consume(message) {
```

```
    CompletableFuture.runAsync(() → {
```

```
        process(message); // slow
```

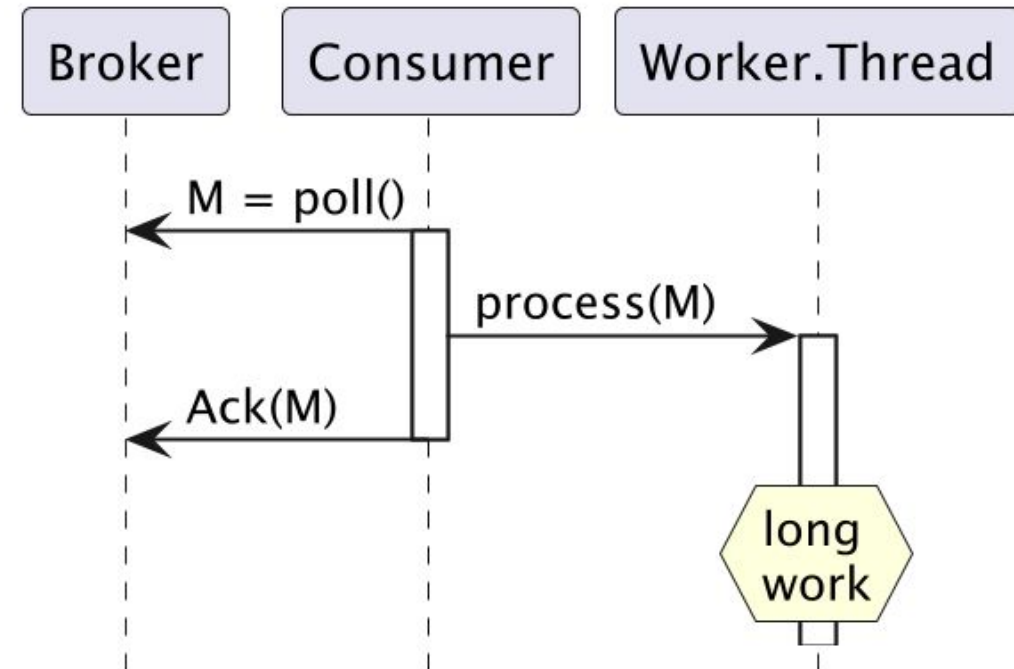
```
> Ack M despite (later) errors in  
    process();
```

> Message(s) get lost on kill -9 / crash



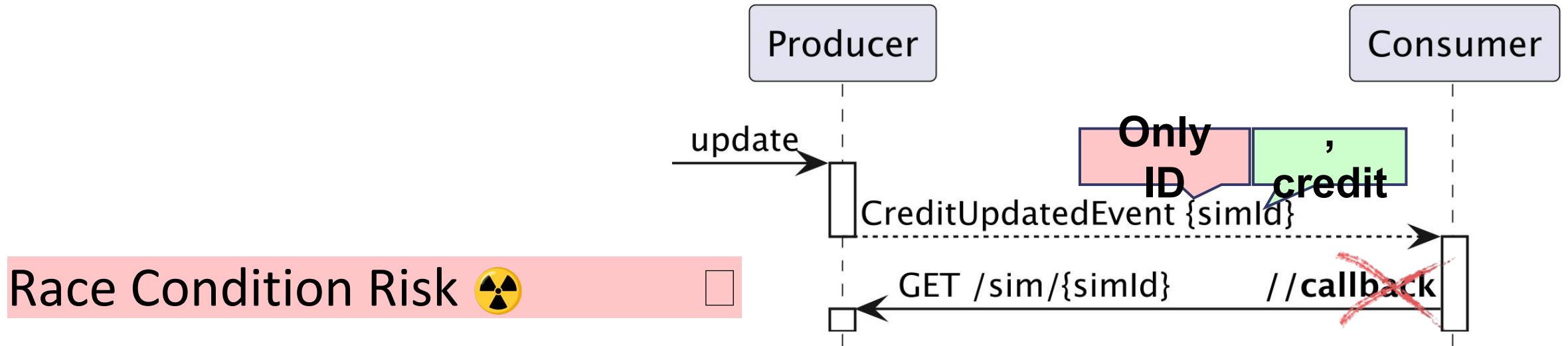
Waiting in Thread Pool's in-mem work queue

> No Backpressure ⇒ Out Of Memory



Consumer Calls-back Producer

- Consumer gets a **Notification Event** lacking useful details
... and calls back the Producer to GET more data



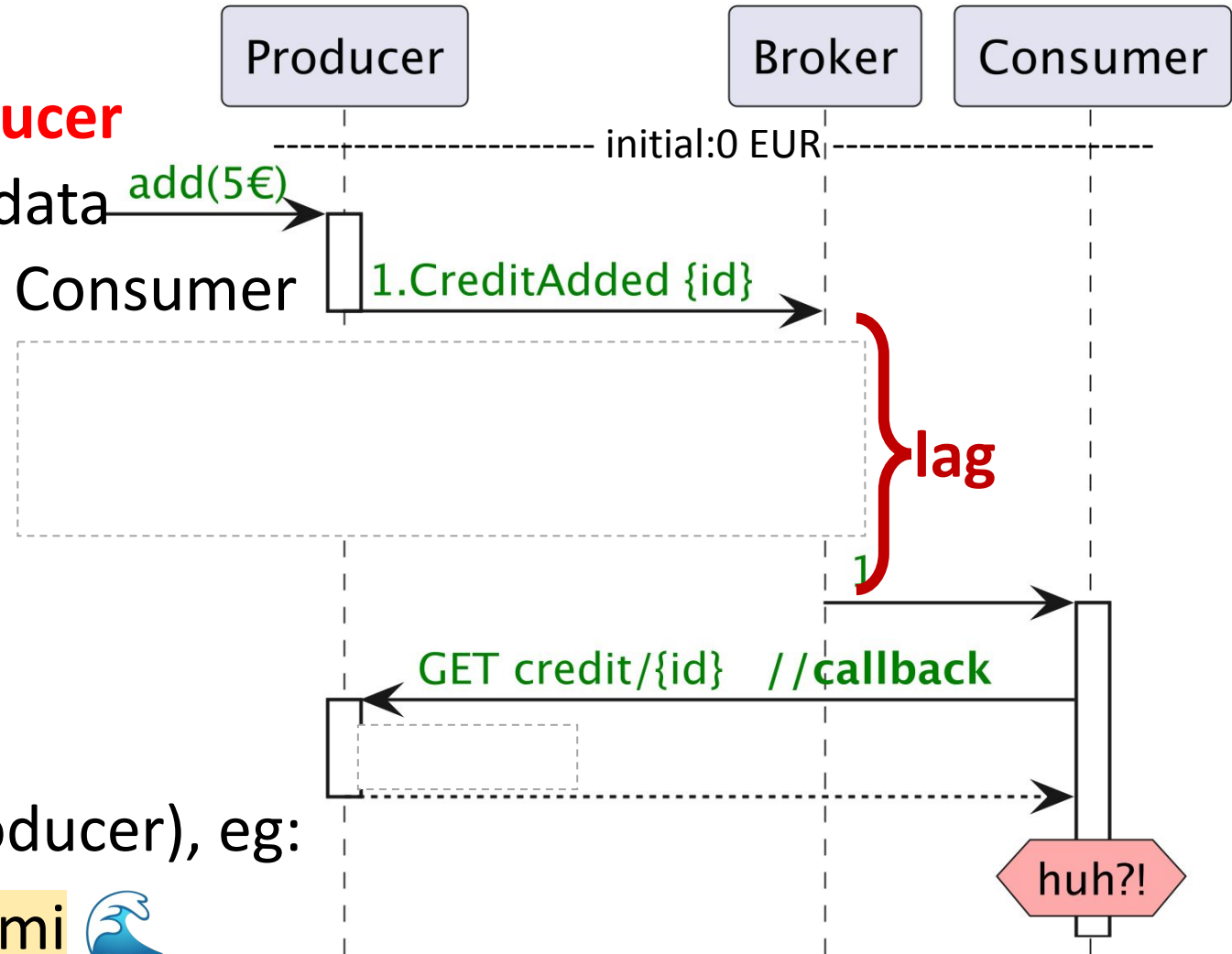
⇒ **Event-Carried State Transfer***: more data in event (ofc 😬)

- vs REST (REpresentation State Transfer)
- ⇒ Event Schema versioning, backwards/forwards compat tests

1) Consumer Callback is Too Late

= Data was updated AGAIN in Producer

- Might be OK to get only latest data
- Intermediary state not seen by Consumer
 $0 \Rightarrow 12 \Rightarrow 12$



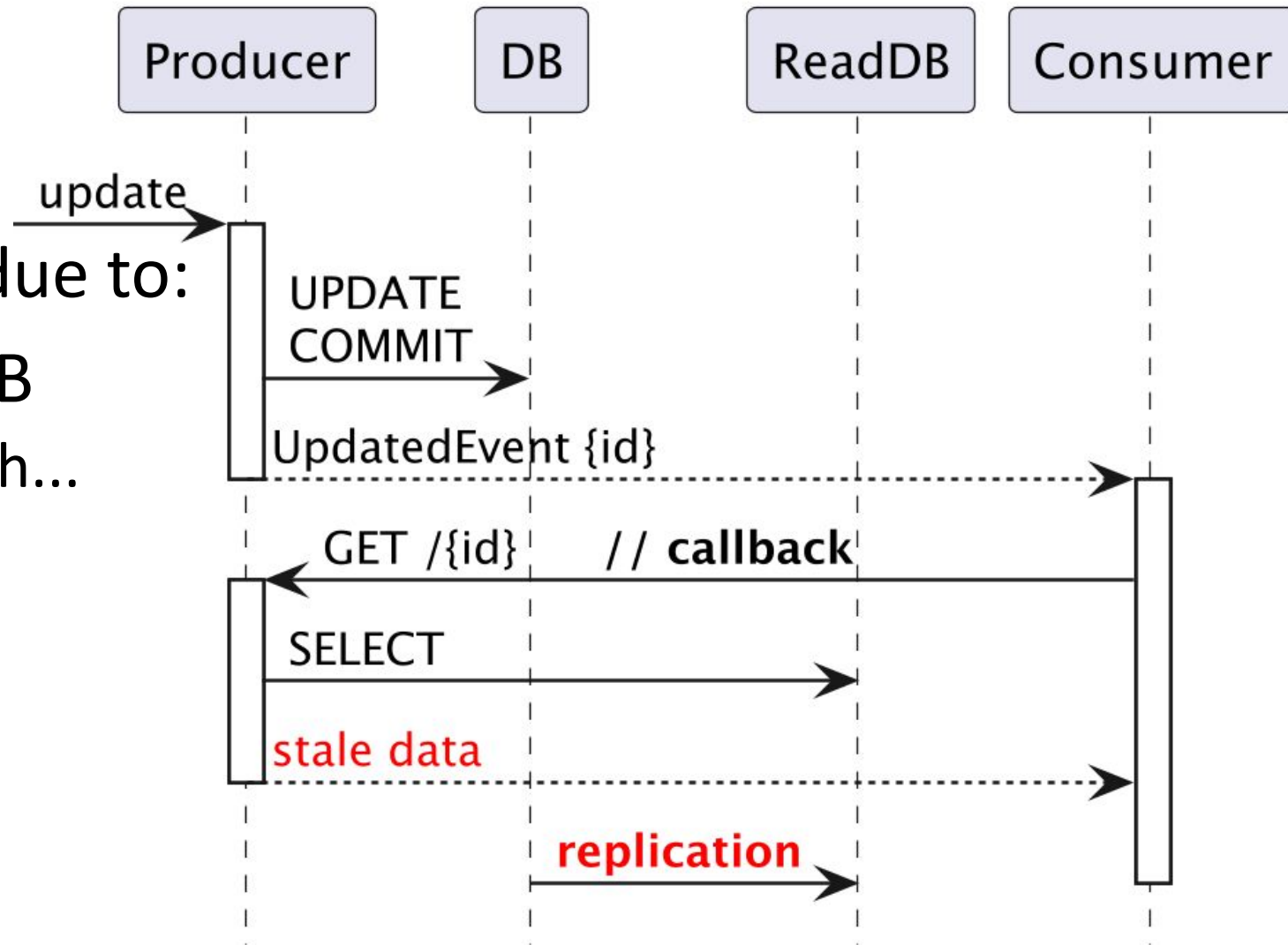
Consumer Lag (eg 5 min behind Producer), eg:

- Process a Batch File > Event Tsunami 
 -  Ingest Amazon order updates at 5⁰⁰AM => 10 M events sent in 10 min

2) Consumer Callback is **Too Early** 🤯

The data is **not yet there**, due to:

- GET reads from lagging DB
 - Read replica, Elastic Search...

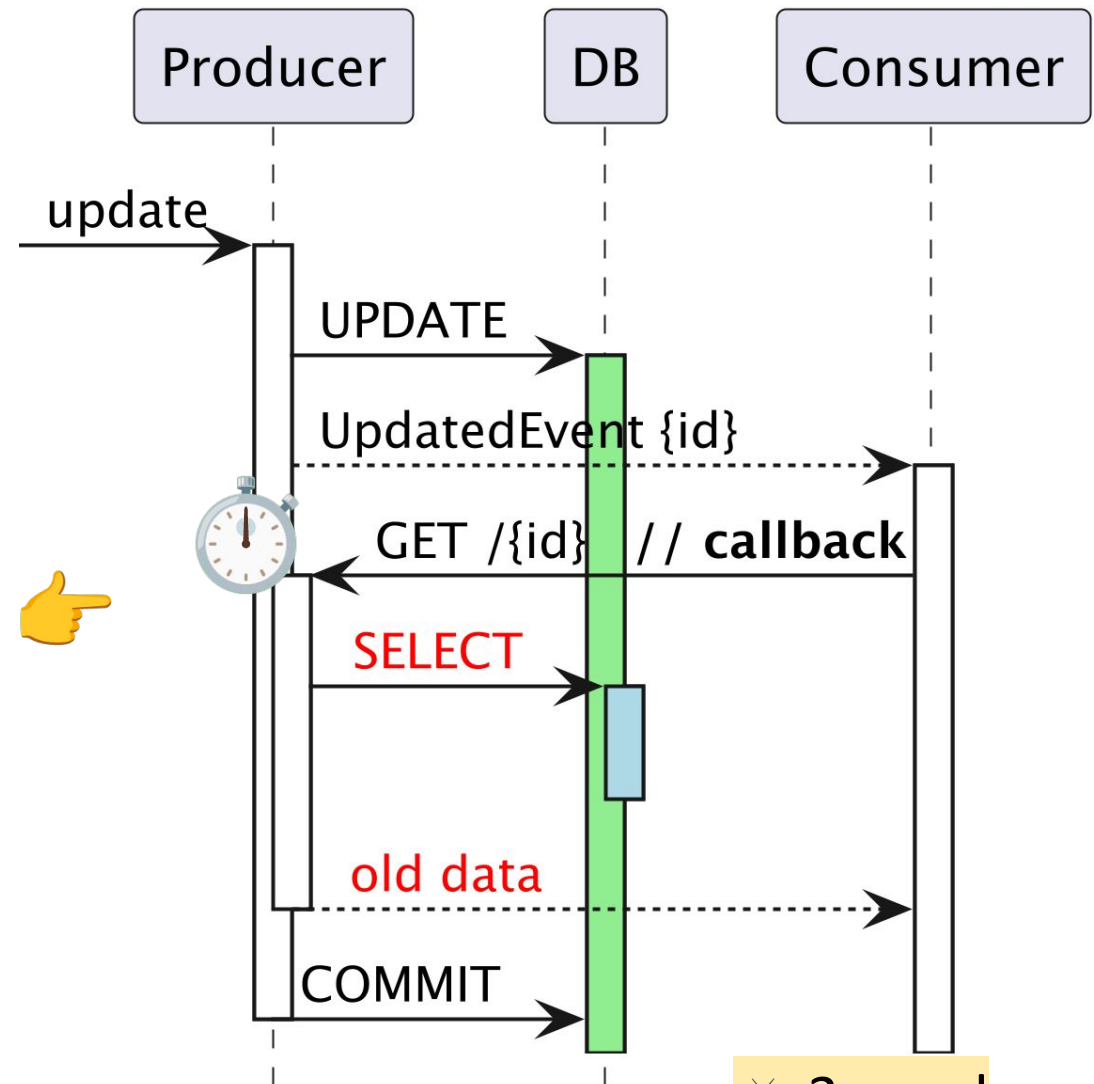


3) Consumer Callback is **Too Early** 🤯

The data is **not yet there**, due to:

- GET reads from lagging DB
 - Read replica, Elastic Search...
- Message sent before DB commit 🙅

DB update
send message
DB commit



★ **Event-Carried State Transfer** = move state with events

Event Schema

TODO

new fields=>💣 Consumer

- java serializable

- JSON+fail.on.unknown.properties

- Consumer can **receive newer event version**

- ... published by newer Producer

- ⇒ must be **Future Compatible**

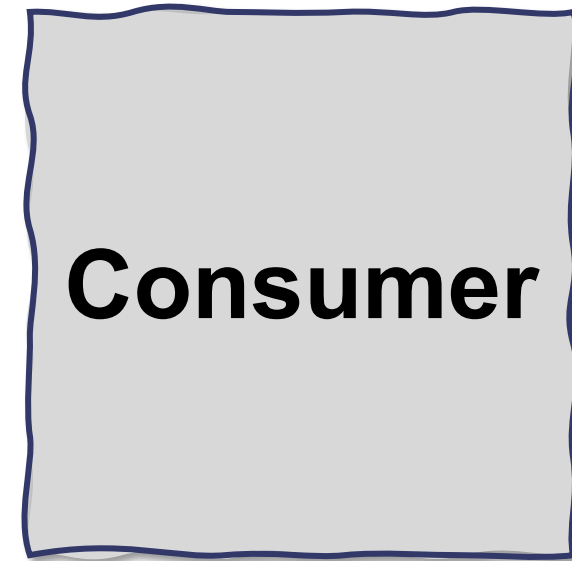
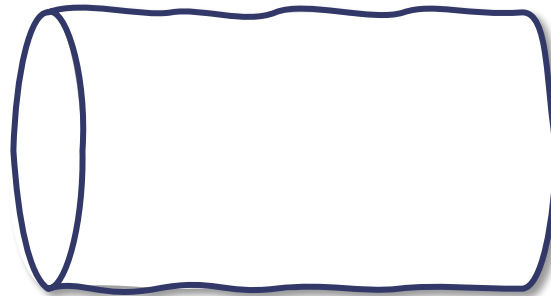
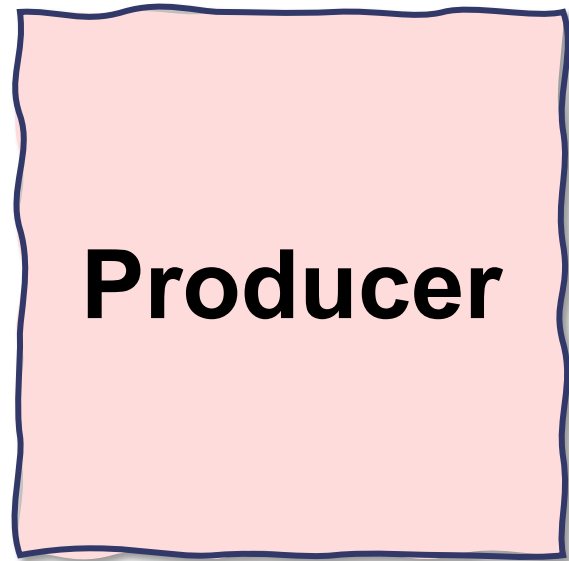
- Consumer can **receive older event version**

- ... at replay history

- ... published by Producer #2 (old) – is it command?

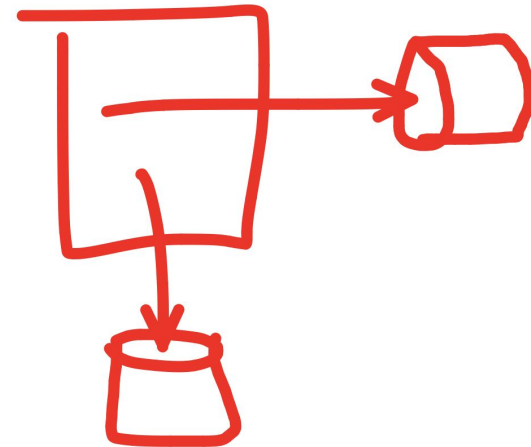
- ⇒ must be **Backwards Compatible**







Send + Update



```
placeOrder(r) { // called over REST
  validate(r) ← ASAP, sync 👍
  broker.send(message)
  orderRepo.save(order)
}
```

- NOT NULL

- COLUMN
SIZE

- FK

- PK

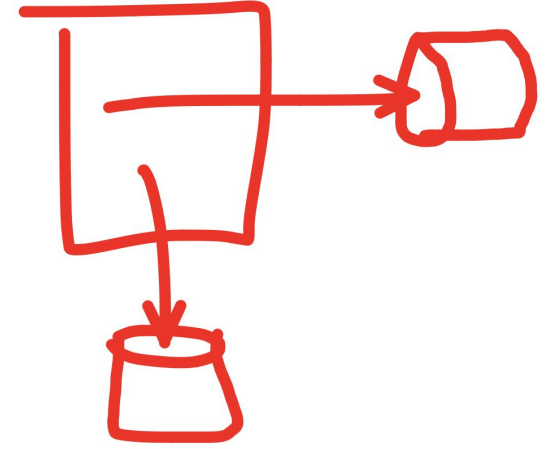
- UNIQUE...

*Start with
most risky
step first!*

Which is
more likely
to fail?



Update + Send



```
placeOrder(r) {  
  validate(r)  
  orderRepo.save(order)  
  broker.send(message)  
}
```



Message is sent
at-most-once

Risk of
Broker
offline



Dual Write Problem



Update + Send

@Transactional

```
placeOrder() {  
    stockRepo.save(stock)  
    orderRepo.save(order)  
    broker.send(message)  
}
```

Added to:

- 1) Make atomic 2+ DB updates
- 2) Use JPA lazy-loading
- 3) Make (SQL updates + send) atomic

WRONG ASSUMPTION

Only after method ends:



- JPA flushes all INSERT, UPDATE, DELETE to DB
- Commit SQL transaction

✗ AFTER message was sent

Unless still using 2PC:



ActiveMQ/Artemis



IBM MQ



TIBCO ...

✓ Send After Commit (with Spring)

@Transactional

```
placeOrder() {  
    stockRepo.save(stock)  
    orderRepo.save(order)  
    applicationEventPublisher.  
    }  
}
```

⇒ Pull send() out, after placeOrder()

⇒ @TransactionalEventListener ✨

☢ Risk: message lost on crash/shutdown right after commit

@TransactionalEventListener(AFTER_COMMIT)

```
onEvent(message) {  
    broker.send(message)  
}
```

Runs after the transaction within which the event was published

☢ Silent miss if no @Transactional

✓ Outbox Table Pattern

+10 other
debate points

@Transactional

```
placeOrder() {  
    stockRepo.save(stock)  
    orderRepo.save(order)  
    outboxRepo.save(message)  
}
```

Insert the message to send in
DB
within your transaction

Store-and-Forward Pattern

a) **CDC** with Debezium.io
b) **DIY** with @Scheduled(rate=1s)

```
pollSend() {  
    m = select outbox  
    broker.send(m)    
    delete m in outbox  
}
```

**on Error  or
Timeout **

Message sent
at-least-once



Implementing an Inbox / Outbox Table ≈ Reinventing a Message Broker



Duplicate Messages

Because:

- **Producer sent M twice** (due to outbox-retry...)
- **Consumer didn't Ack M** (due to crash...)



Idempotent Consumers



- Message is a dup if:
 - `CreditAdded{+5EUR}.idempotencyKey` $\in$ prev_messages in DB (delta event)
`CreditUpdated{8EUR}.newCredit == oldSim.credit` in DB (snapshot event)
or `seq_no, publish_nanos, message_offset ...`
- **Dedup M, then** do any side-effects: `api.call()`, `DB.update..`

Event Semantics

■ Delta Event

- `CreditAdded {added:+5 EUR}`
- ☹ Consumers **must dedup**
- ☹ Consumers **must track old state** to compute new state

■ Snapshot Event

- `CreditUpdated {newCredit:7 EUR} // new state`

■ Both: "EventShot" 🤔

- `CreditAdded {newCredit :7 EUR, added:+5 EUR}`

Idempotent Consumer

1. I have a DB

- a) Track in DB processed idempotency-keys
- b) Store in DB your Kafka consumer offsets (not in Kafka Broker)

2. I don't have a DB, I only API-call

- a) Idempotent API: PUT or Idempotency-Key header 🙌
- b) GET before POST – "is my data already there?"



Exposing Internal Events

■ Internal Event

- `CreditAdded{..}`, `CredidPaidBack{..}` + 9 others
- Between microservices of an ecosystem/SCS/departement
- Source of truth in Event-Sourced apps
- If exposed => **client coupling** 📌 and **semantic overwhelming** 🤔



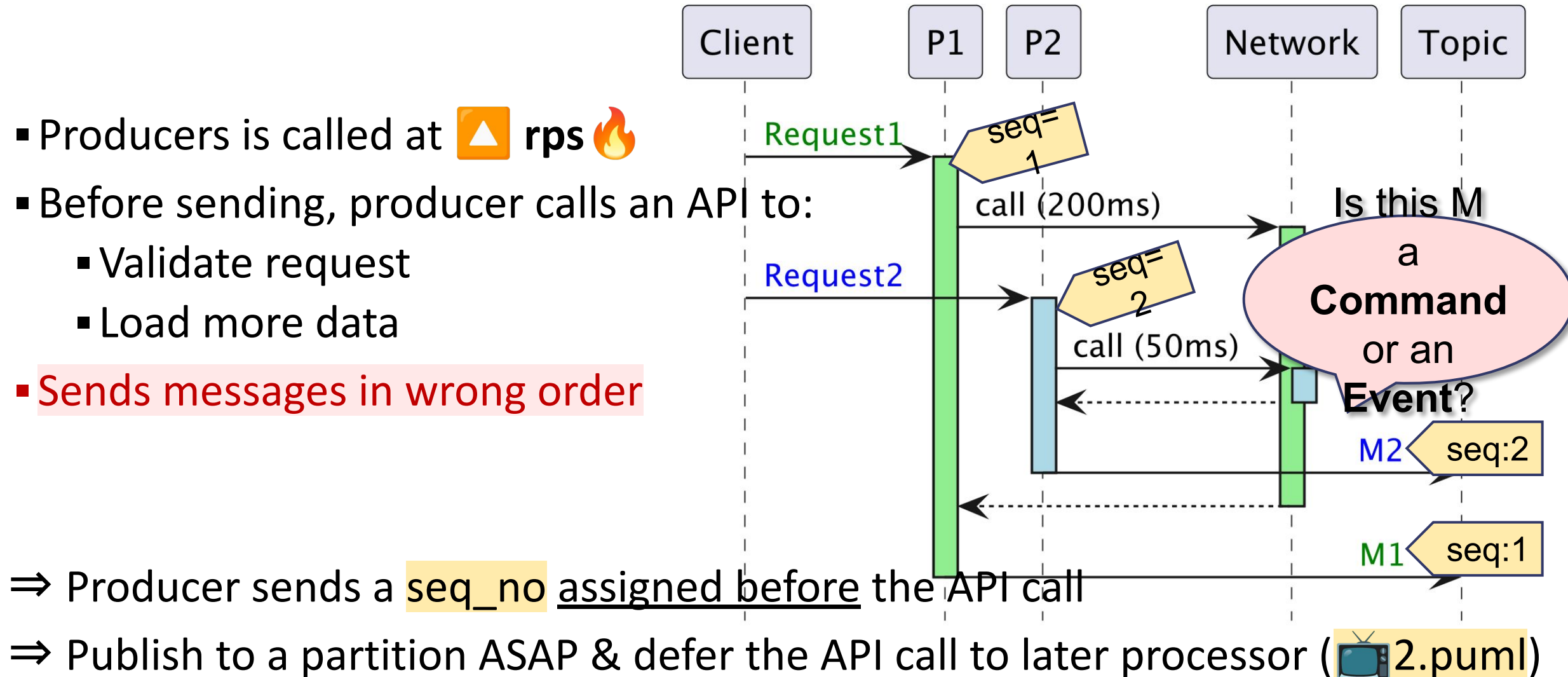
■ Integration Event = Contract

- `CreditUpdated{simId, newCredit}` – **narrow** key event
- `SimUpdated{<15 fields>}` – broad

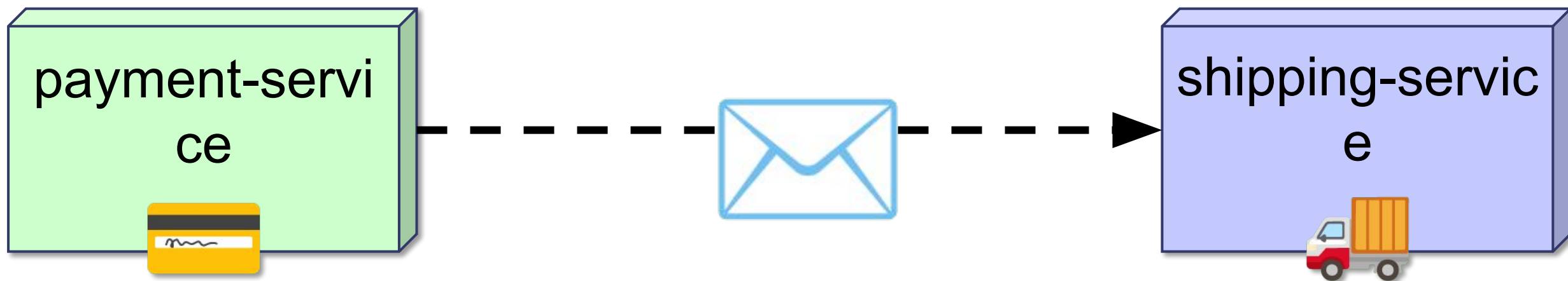


Producers Race

- Producers is called at rps
- Before sending, producer calls an API to:
 - Validate request
 - Load more data
- Sends messages in wrong order**



Rule: When **payment** is confirmed, the order must be **shipped**.



PaymentCompleted **Event**

ShipOrder **Command**

"pub-sub", "topic"

PaymentCompleted Event

VS

"point-to-point", "queue"

ShipOrder Command

Past

Tense

None: tells a fact

No, it's history



Multiple [0 .. N]

Add more listeners with zero impact 🙌

Producer

But caring for consumers

Timing (when?)

Producer Expectations

Can Consumer Refuse it?

"It's wrong!"

of consumer-groups
(apps)

Who's Message Schema

What Bounded Context?

Future / Imperative

Expects an action

Yes if invalid request

One: shipping-service
with 1+ consumers (threads)

Consumer

request params $\cong$ POST body



^{=Request} Command / ^{=Fact} Event Confusion

■ CheckBeerInSapEvent

= "command event" (EDA =   )

- Who is allowed to listen? 
- Is there a Reply message? 

■ OrderNeedsShipmentEvent

= passive-aggressive event 

- *Publisher afraid to let go orchestration* 

Honey,
dishwasher
finished





Event Granularity

A) CUD Event just replicates data $\approx$ something changed 🙋

"OrderEvent" means ...

- Created
- or Updated $\Rightarrow$ Self-healing Event, aka "upsert in DB"
- or Deleted
 - if message.type = D --- .type $\in$ {C|U|D}
 - if payload = **null** in Kafka (tombstone message)

B) Domain Event = fine grained (best kept internal)

- OrderPlaced, ..Fulfilled,
- ..Shipped, ..Returned, ..Refused ... +10 more $\Rightarrow$ semantic overload





Meanwhile, on Production Front ...



Errors

Error Handling Strategies

- **Sync Retry 3-5 times** vs race, transient failure
 - Infinite retry 🚫: expected Avro, got JSON ✂️
- **Delayed Retry** vs out-of-order
- **Ignore Error**, for $\leq N$ messages, or for ≤ 5 min 👍
- **Log** > alarm 🔔: log.error("Failed: " + toJson(🦠message));

Sensitive / Too large?
- **Send to DLQ** > alarm 🔔
- **Send an Error Event** if biz-relevant: PaymentFailed{reason}

Poison Pill Message



= **M** that kills any consumer receiving it $\Rightarrow$ no ACK $\Rightarrow$ next consumer $\curvearrowright$
 $\Rightarrow$ All consumers are **continuously restarted** $\Rightarrow$ **Lag** on all topic partitions

- **M** = OrderPlacedEvent {items:[1000 distinct product IDs]}

- ✂ Consumer loads all products in memory

- **Out Of Memory** $\Rightarrow$ Consumer process dies $\Rightarrow$ 💀 + $\curvearrowright$

- ✂ Consumer loads all products sequentially, one by one, over REST

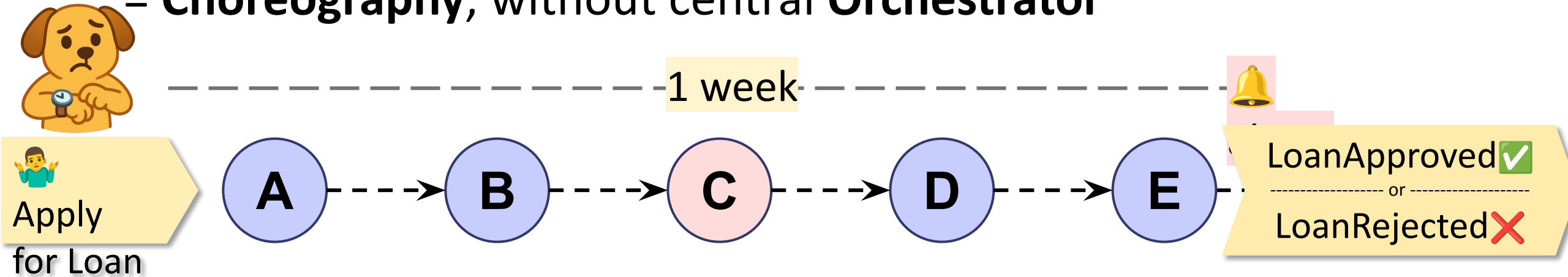
- **Processing takes too long** $\Rightarrow$ Broker considers consumer hung $\Rightarrow$ 💀 + $\curvearrowright$

- **Bonus**: Consumer #2 races with Consumer #1 still processing **M**

- ✂ Expected Avro Message got JSON $\Rightarrow$ ∞ retry $\Rightarrow$ 💀 + $\curvearrowright$

Broken Choreography Chain

- In EDA, services listen to events from upstream services
= Choreography, without central Orchestrator



- System **C** forgets to publish its event (NO exception in logs)

⇒ A watchdog  monitors completion of the e2e chain ≈ SLA

⇒ **C** sends to **D** heartbeats  = empty events ≈ health check



The Cheese Paper Disaster



∞ Event Loop



traceparent: 123

Add cheese to cart

Small adjustment for UX:
CheesePaper.type := 'cheese'

cart-service

PromoGrante

traceparent: 123
{
 CheesePaper
}

traceparent: 123
traceparent: 123
traceparent: 123

CartItemAdded
{ item: {...} }

promo-service

If item.type = 'cheese',
grant free Cheese Paper.

Entire Kafka Cluster Crashed
Most systems had downtime of 20 minutes

👍 set quota.producer.default = max bytes/sec

traceparent: 123



Request Reply – RPC Nostalgia

= Async, Durable RPC

- Request  = Command
- Reply  $\approx$ Event, but targeted to requester
- **Requestor blocks** for the reply 
 - = a message bridge between UI calling REST and Message Backoffice
 - [Rabbit](#) - *via Temporary Reply Queue*
 - [Kafka](#), *SQS, pub-sub - via a ReplyTopic + CorrelationId*
 - N Requestor instances => N consumer-groups for reply topic
 -  Volatile waiting pending replies
- **Requestor resumes flow from DB** upon reply  
 -  Complexity , Fault Tolerance 

Top Event-Driven Pitfalls

- **Integration Testing:** timeout(..), spy, ☢ message leak cross-tests
- **Message Observability:** 👤 + 🤖 = 1.3 - 3x impact
- **Consumer Race** ⇒ partitions, locking (optimistic/pess)
- **Out-of-Order** ⇒ partitions, delayed retry, seq_no
- **Consumer Callback Race** ⇒ Event-Carried State Transfer
- **Async Consumer** (lost messages) ⇒ Ack after process
- **Dual Write** (save + send) ⇒ Outbox Table/CDC
- **Duplicate Messages** ⇒ Idempotent Consumer ★
- **Command vs Event Message**
- **Errors:** ∞ retry, poison pills, broken chain (🐶, 💖), ∞ event loop

