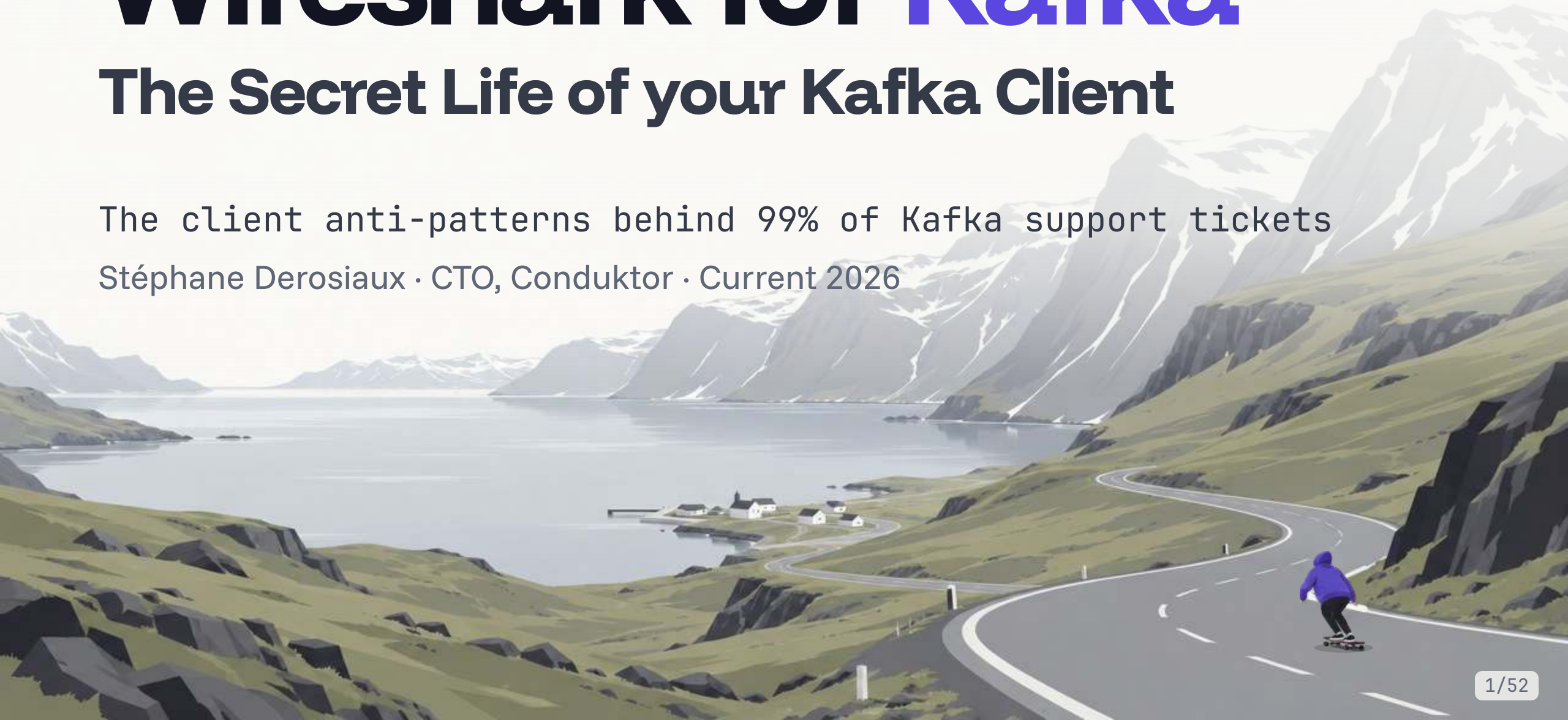
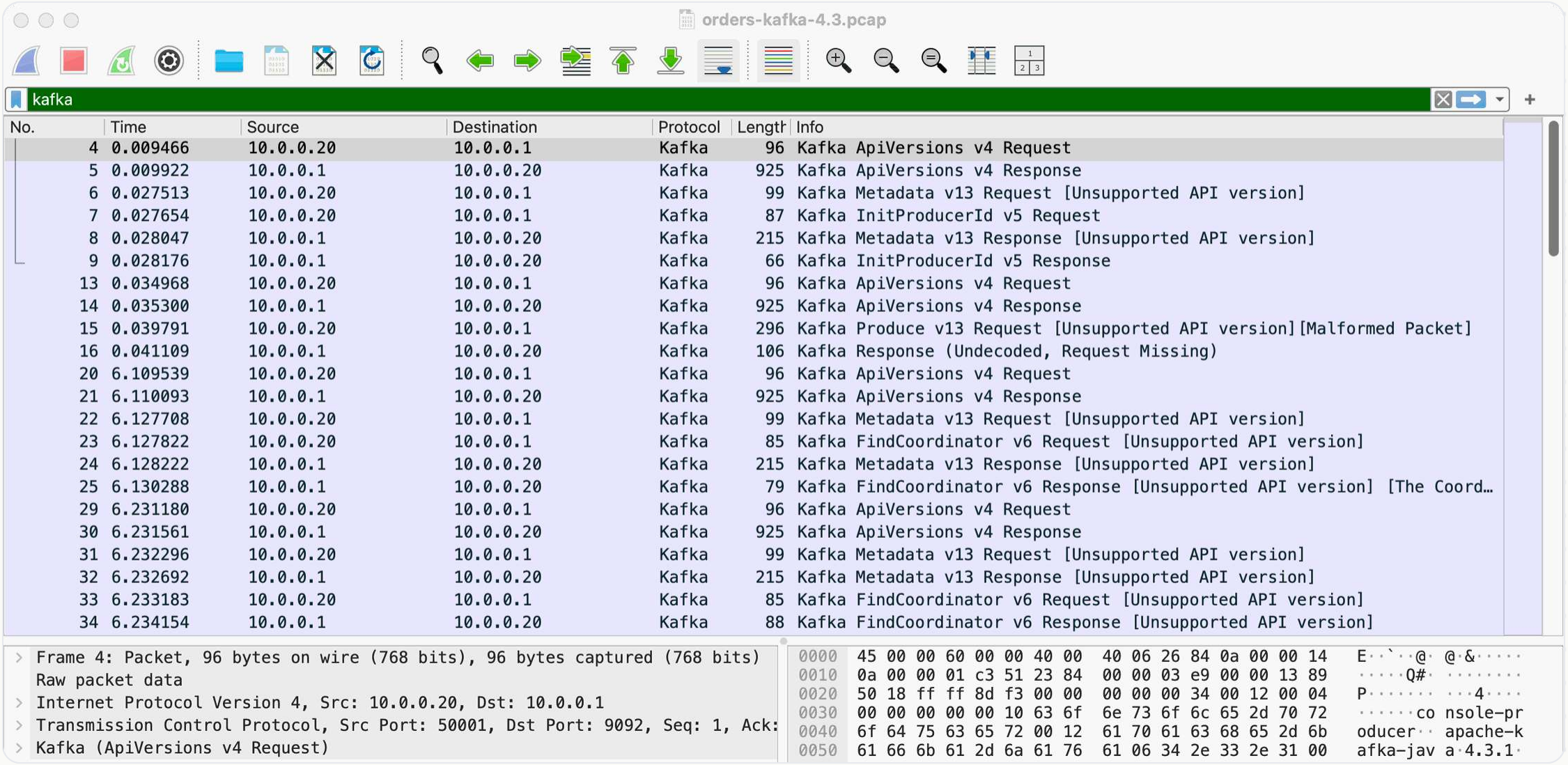
Wireshark for Kafka

The Secret Life of your Kafka Client

The client anti-patterns behind 99% of Kafka support tickets

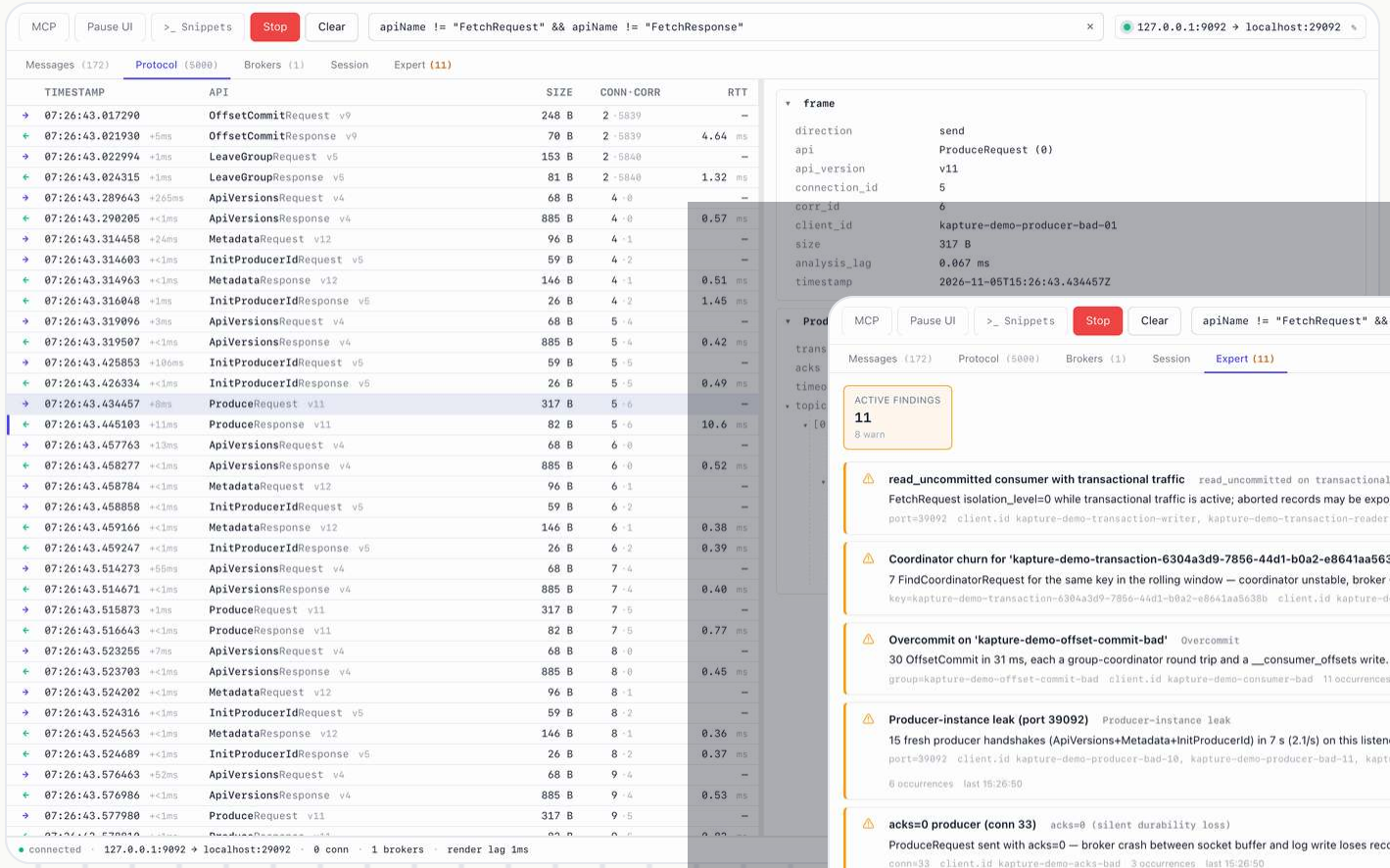
Stéphane Derosiaux · CTO, Conduktor · Current 2026




orders-kafka-4.3.pcap
kafka
No. Time Source Destination Protocol Length Info
4 0.009466 10.0.0.20 10.0.0.1 Kafka 96 Kafka ApiVersions v4 Request
5 0.009922 10.0.0.1 10.0.0.20 Kafka 925 Kafka ApiVersions v4 Response
6 0.027513 10.0.0.20 10.0.0.1 Kafka 99 Kafka Metadata v13 Request [Unsupported API version]
7 0.027654 10.0.0.20 10.0.0.1 Kafka 87 Kafka InitProducerId v5 Request
8 0.028047 10.0.0.1 10.0.0.20 Kafka 215 Kafka Metadata v13 Response [Unsupported API version]
9 0.028176 10.0.0.1 10.0.0.20 Kafka 66 Kafka InitProducerId v5 Response
13 0.034968 10.0.0.20 10.0.0.1 Kafka 96 Kafka ApiVersions v4 Request
14 0.035300 10.0.0.1 10.0.0.20 Kafka 925 Kafka ApiVersions v4 Response
15 0.039791 10.0.0.20 10.0.0.1 Kafka 296 Kafka Produce v13 Request [Unsupported API version][Malformed Packet]
16 0.041109 10.0.0.1 10.0.0.20 Kafka 106 Kafka Response (Undecoded, Request Missing)
20 6.109539 10.0.0.20 10.0.0.1 Kafka 96 Kafka ApiVersions v4 Request
21 6.110093 10.0.0.1 10.0.0.20 Kafka 925 Kafka ApiVersions v4 Response
22 6.127708 10.0.0.20 10.0.0.1 Kafka 99 Kafka Metadata v13 Request [Unsupported API version]
23 6.127822 10.0.0.20 10.0.0.1 Kafka 85 Kafka FindCoordinator v6 Request [Unsupported API version]
24 6.128222 10.0.0.1 10.0.0.20 Kafka 215 Kafka Metadata v13 Response [Unsupported API version]
25 6.130288 10.0.0.1 10.0.0.20 Kafka 79 Kafka FindCoordinator v6 Response [Unsupported API version] [The Coord…
29 6.231180 10.0.0.20 10.0.0.1 Kafka 96 Kafka ApiVersions v4 Request
30 6.231561 10.0.0.1 10.0.0.20 Kafka 925 Kafka ApiVersions v4 Response
31 6.232296 10.0.0.20 10.0.0.1 Kafka 99 Kafka Metadata v13 Request [Unsupported API version]
32 6.232692 10.0.0.1 10.0.0.20 Kafka 215 Kafka Metadata v13 Response [Unsupported API version]
33 6.233183 10.0.0.20 10.0.0.1 Kafka 85 Kafka FindCoordinator v6 Request [Unsupported API version]
34 6.234154 10.0.0.1 10.0.0.20 Kafka 88 Kafka FindCoordinator v6 Response [Unsupported API version]
Frame 4: Packet, 96 bytes on wire (768 bits), 96 bytes captured (768 bits)
Raw packet data
Internet Protocol Version 4, Src: 10.0.0.20, Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 50001, Dst Port: 9092, Seq: 1, Ack:
Kafka (ApiVersions v4 Request)
0000 45 00 00 60 00 00 40 00 40 06 26 84 0a 00 00 14 E··`··@· @·&·····
0010 0a 00 00 01 c3 51 23 84 00 00 03 e9 00 00 13 89 ·····Q#· ·····
0020 50 18 ff ff 8d f3 00 00 00 00 00 34 00 12 00 04 P····· ···4····
0030 00 00 00 00 00 10 63 6f 6e 73 6f 6c 65 2d 70 72 ·····co nsole-pr
0040 6f 64 75 63 65 72 00 12 61 70 61 63 68 65 2d 6b oducer·· apache-k
0050 61 66 6b 61 2d 6a 61 76 61 06 34 2e 33 2e 31 00 afka-jav a·4.3.1·

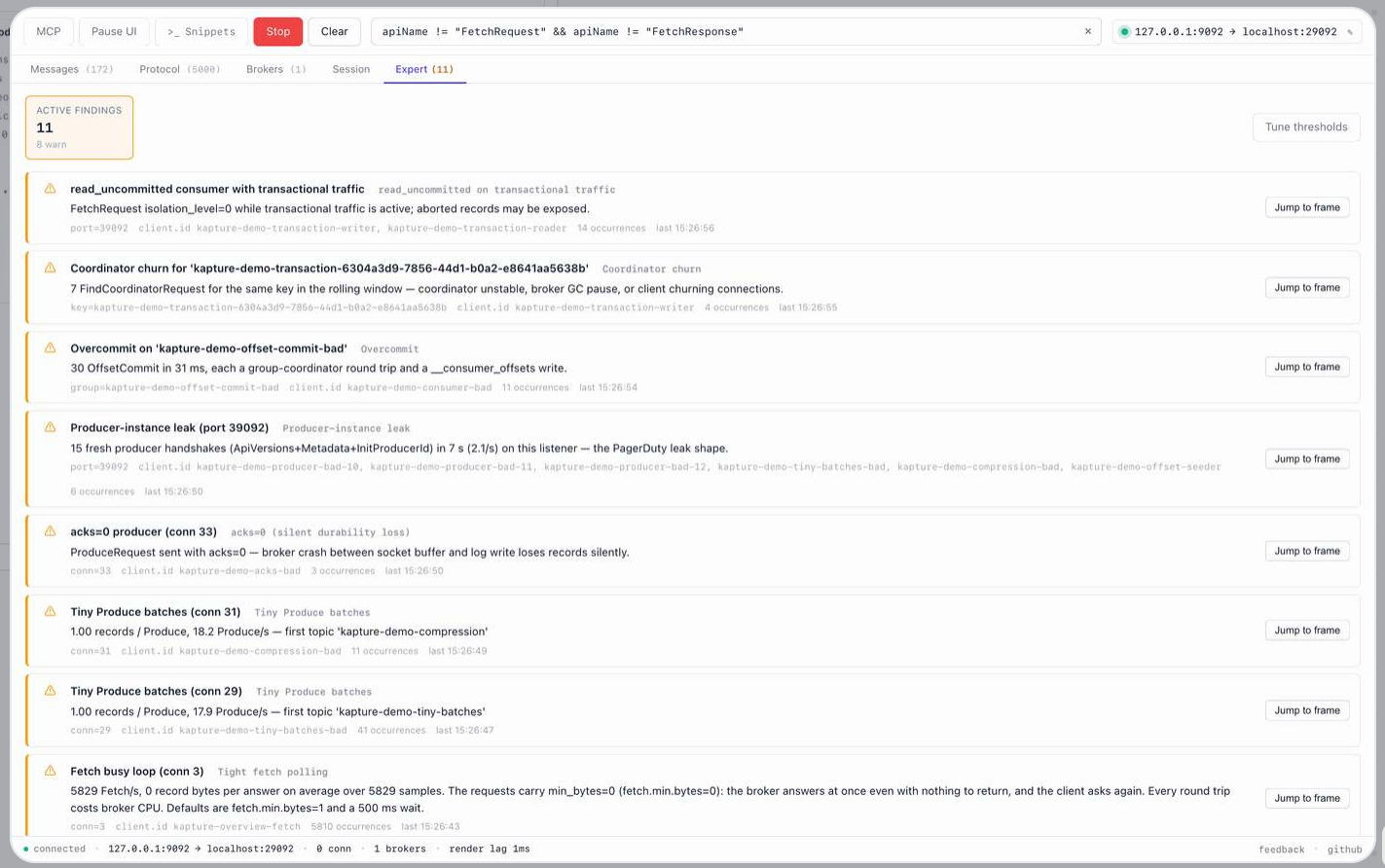
Kapture

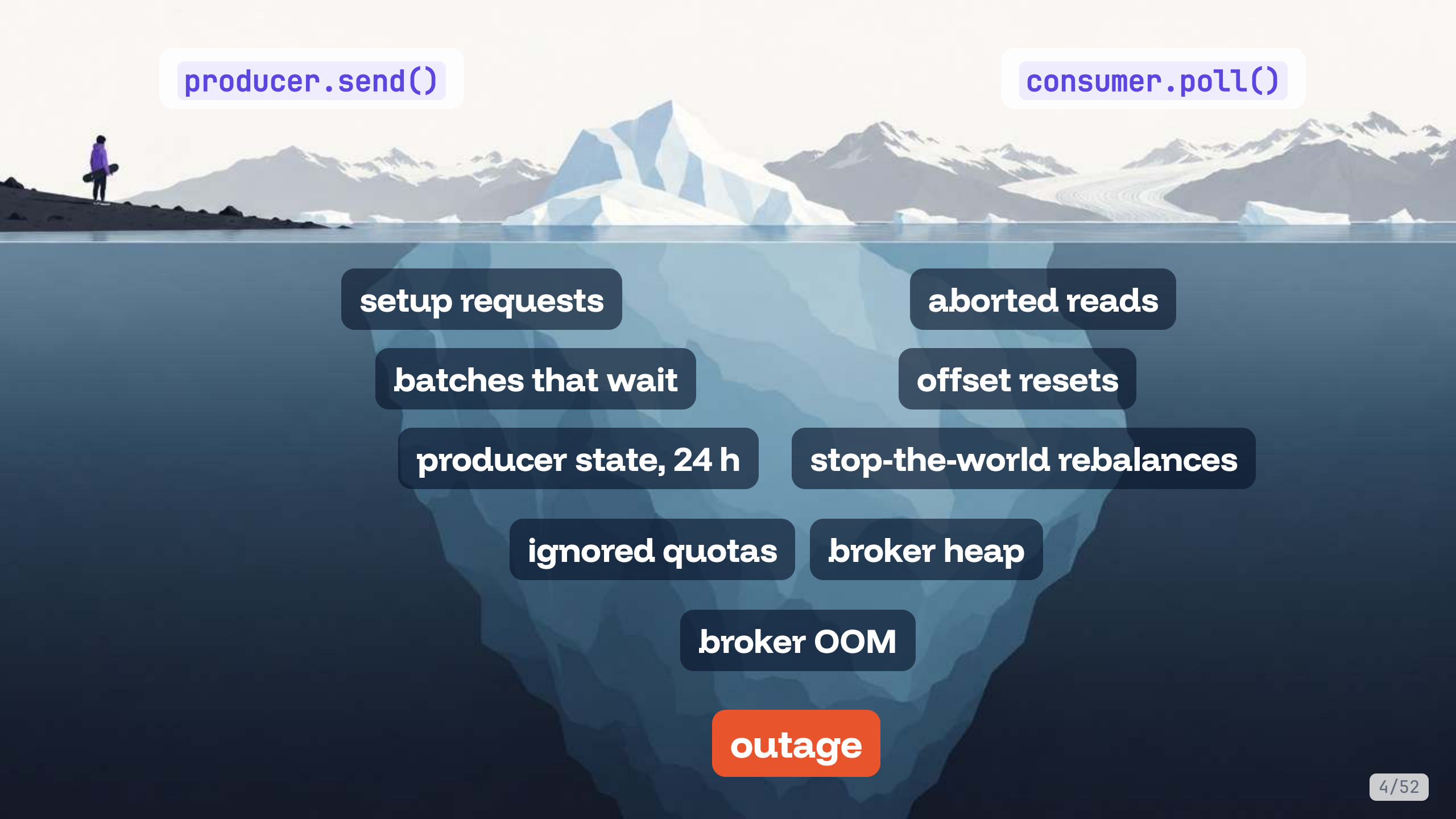
github.com/conduktor/kapture



the protocol

the anti-patterns




producer.send()
consumer.poll()
setup requests
aborted reads
batches that wait
offset resets
producer state, 24 h
stop-the-world rebalances
ignored quotas
broker heap
broker OOM
outage

12 events. How many TCP requests?

orders-api · OrderController.java

```
@PostMapping("/orders")
Receipt accept(Order o) {
  var r = orders.save(o);
  events.publish(o);
  return r;
}
```

produce →

Kafka

topic **orders**



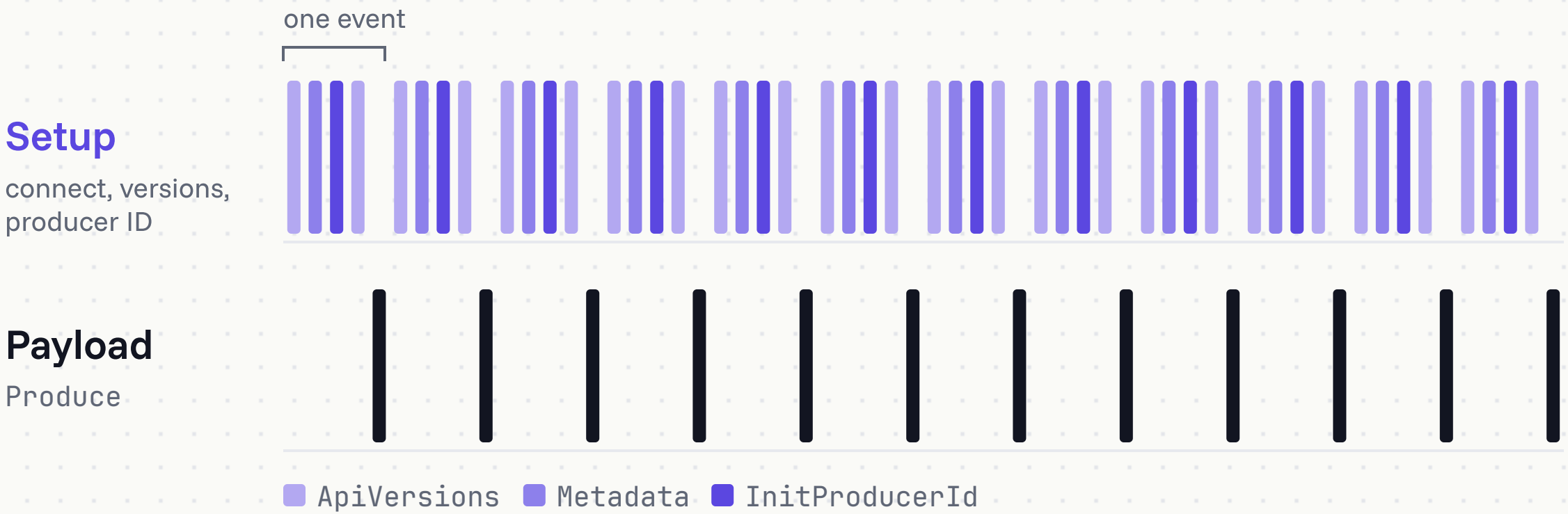
one event: OrderCreated {"orderId": "A-1042"}

✓ 12 events, every test green

?

A 12 | **B** 24 | **C** 60

4 setup requests for every event



60 requests on 24 TCP connections, for 12 events.

conduktor

We read the Kafka protocol for a living

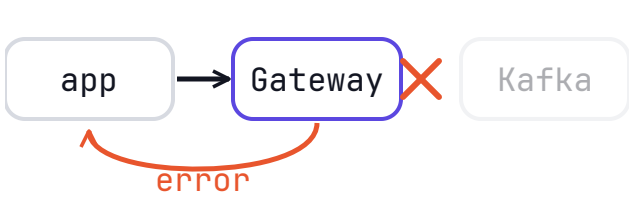


- **Gateway**, our Kafka proxy, decodes every request, from every client library.
- We work with platform and app teams on **how their clients use Kafka**.

Gateway enforces client rules

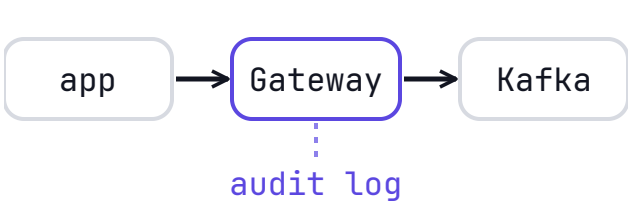
```
"config": {
  "acks": {"value": [-1], "action": "BLOCK"},
  "compressions": {"value": ["LZ4", "ZSTD"], "action": "INFO"},
  "idempotenceRequired": {"value": true, "action": "BLOCK"}
}
```

BLOCK



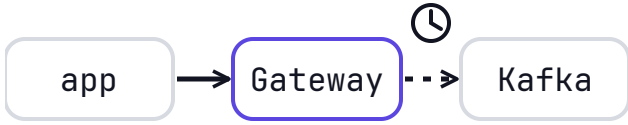
Stopped. The app gets an error.

INFO



Goes through. Gateway logs it.

THROTTLE



Goes through, delayed.

Kapture: a desktop app that spots anti-patterns



No library to add: point `bootstrap.servers` at it, or attach the JVM agent.

Client anti-patterns hit the cluster

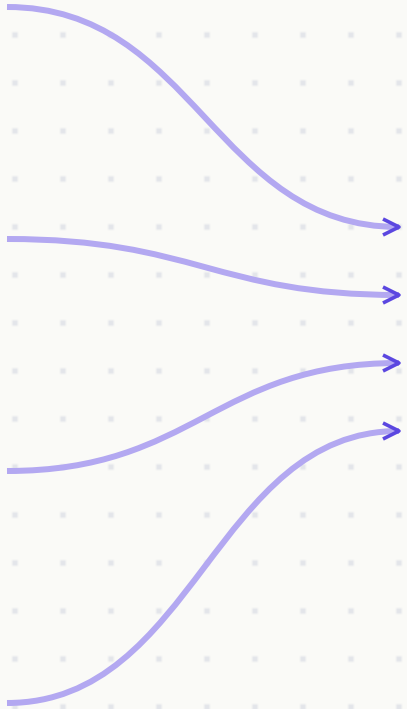
your applications

Producer
creating it, sending

Consumer
fetching

Consumer groups
rebalancing

Kafka Streams
transactions



Kafka cluster

- heap
- CPU, request threads
- slower for every other team


WE CREATE A PRODUCER, SEND, CLOSE IT
SO NOTHING STAYS BEHIND, RIGHT?
...NOTHING STAYS ON THE BROKER, RIGHT?


A producer
per call

A new KafkaProducer for every call

your service · OrderController.java

```
var r = orders.save(o);
events.publish(o);
return r;
```

a helper in the same service · Events.java

```
void publish(Event e) {
  try (var p = new KafkaProducer(cfg)) {
    p.send(record(e)).get();
  }  // closed, nothing leaks
}
```

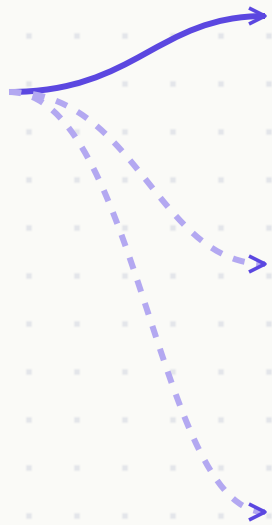
It hides behind abstractions:

	what you write		what runs underneath
Spring Kafka	`new KafkaTemplate<>(pf, overrides)`	→	a new producer factory, per request
Pekko / Akka	`Producer.plainSink(settings)`	→	a new producer, per stream run
Lambda handler	`new KafkaProducer(cfg)`	→	a new producer, per invocation
Flink, exactly-once	`.setDeliveryGuarantee(EXACTLY_ONCE)`	→	a new InitProducerId, per checkpoint

A new producer: at least 5 requests

connection 1 · a bootstrap broker

1. `ApiVersions` — which versions?
2. `Metadata` — who leads **orders**?
3. `InitProducerId` — a producer ID



connection 2 · broker 1, leader of orders-0

4 `ApiVersions` → 5 `Produce`

or broker 2, if the record goes to orders-1

`ApiVersions` → `Produce`

or broker 3, if the record goes to orders-2

`ApiVersions` → `Produce`

One InitProducerId per event

	TIMESTAMP		API	
→	00:38:34.583694		InitProducerIdRequest	v5
→	00:38:34.668634	+85ms	InitProducerIdRequest	v5
→	00:38:34.732560	+64ms	InitProducerIdRequest	v5
→	00:38:34.797964	+65ms	InitProducerIdRequest	v5
→	00:38:34.865873	+68ms	InitProducerIdRequest	v5
→	00:38:34.931972	+66ms	InitProducerIdRequest	v5
→	00:38:34.995852	+64ms	InitProducerIdRequest	v5
→	00:38:35.064460	+69ms	InitProducerIdRequest	v5
→	00:38:35.125102	+61ms	InitProducerIdRequest	v5
→	00:38:35.185545	+60ms	InitProducerIdRequest	v5
→	00:38:35.246337	+61ms	InitProducerIdRequest	v5
→	00:38:35.309156	+63ms	InitProducerIdRequest	v5

A new producer ID every ~65 ms.

broker · producer state

63	64	65
66	67	68
69	70	71
72	73	74

… and one more for every event

Producer closed: how long does this stay?

producer ID	epoch	last batches	last write
63	0	seq 0 → offset 72	00:38:34.653
64	0	seq 0 → offset 73	00:38:34.724
65	0	seq 0 → offset 74	00:38:34.789
66	0	seq 0 → offset 75	00:38:34.857
67	0	seq 0 → offset 76	00:38:34.923
68	0	seq 0 → offset 77	00:38:34.988

… and 6 more

24 h 😬

after the last write

261 bytes

per entry, measured

In the broker's heap, one table per partition.

A random `transactional.id`? Its state stays **7 days** on the transaction coordinator.


THE BROKER, 24 H AFTER YOUR PRODUCER CLOSED
“IT’S BEEN 84 YEARS...”

PagerDuty

August 28, 2025

an API request

pekko-connectors-kafka got settings, not a producer

→

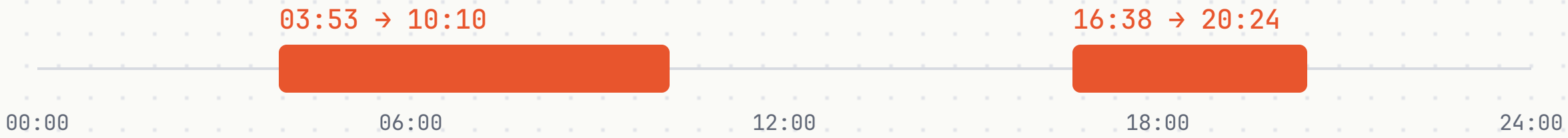
`send()`

builds a new producer, every call

→

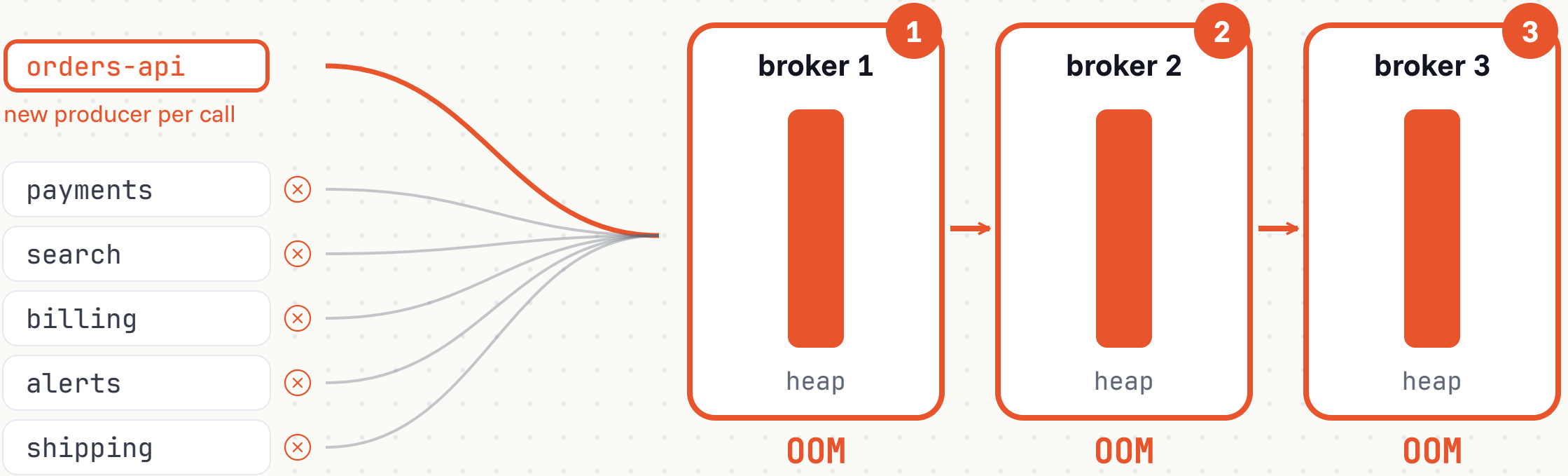
4.2 M producers / hour 🔥

84× their usual rate: broker heap exhausted



pagerduty.com/eng/august-28-kafka-outages-what-happened-and-how-were-improving

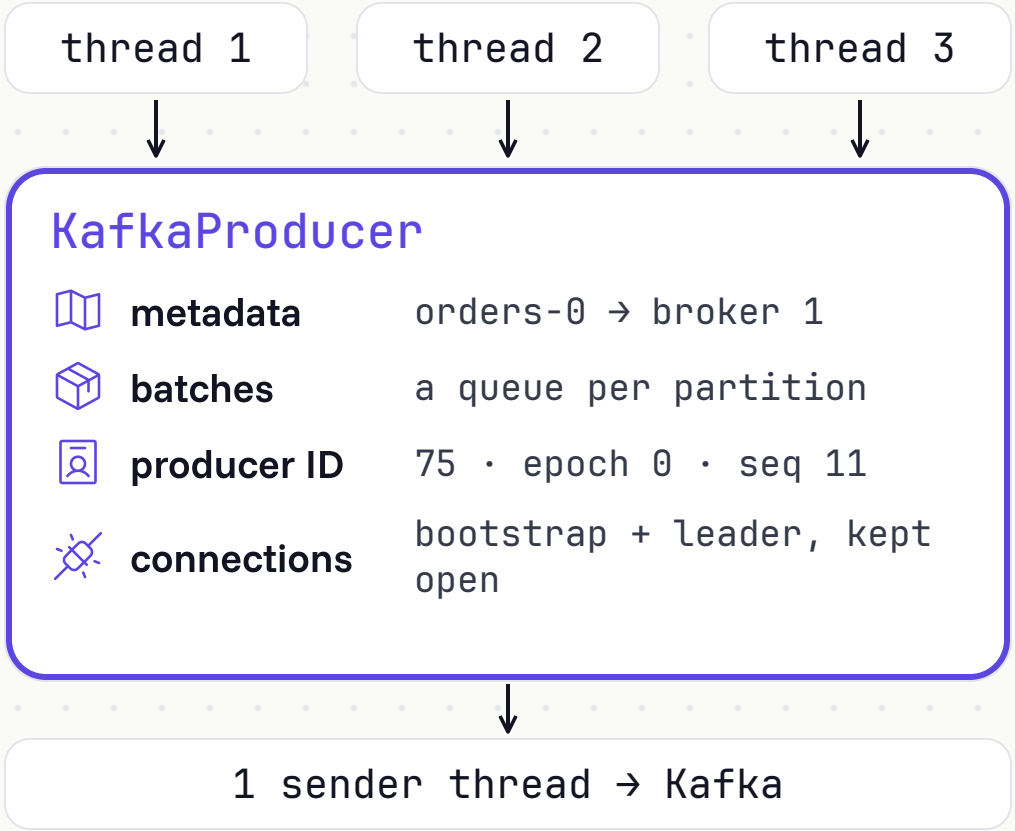
One service's bug, every team's outage



PagerDuty, Aug 28 2025: "began on a single broker but quickly cascaded across the entire cluster"

Fix: one line, one producer per app

```
static final Producer p =
    new KafkaProducer(cfg);
void publish(Event e) throws Exception {
  p.send(record(e)).get();
}
```



Thread-safe: `send()` only appends to a batch, under a lock.


send().get()

60 × send().get(): how long?

producer.properties

```
linger.ms = 50
batch.size = 65536
```

OrderSync.java

```
for (Order o : orders)
  producer.send(record(o))
          .get();
```

Local broker, round trip under a millisecond.

?

A about 0.6 s

B about 3 s

C about 6 s

60 records, one Produce each

send().get() in a loop: each request waits for linger.ms



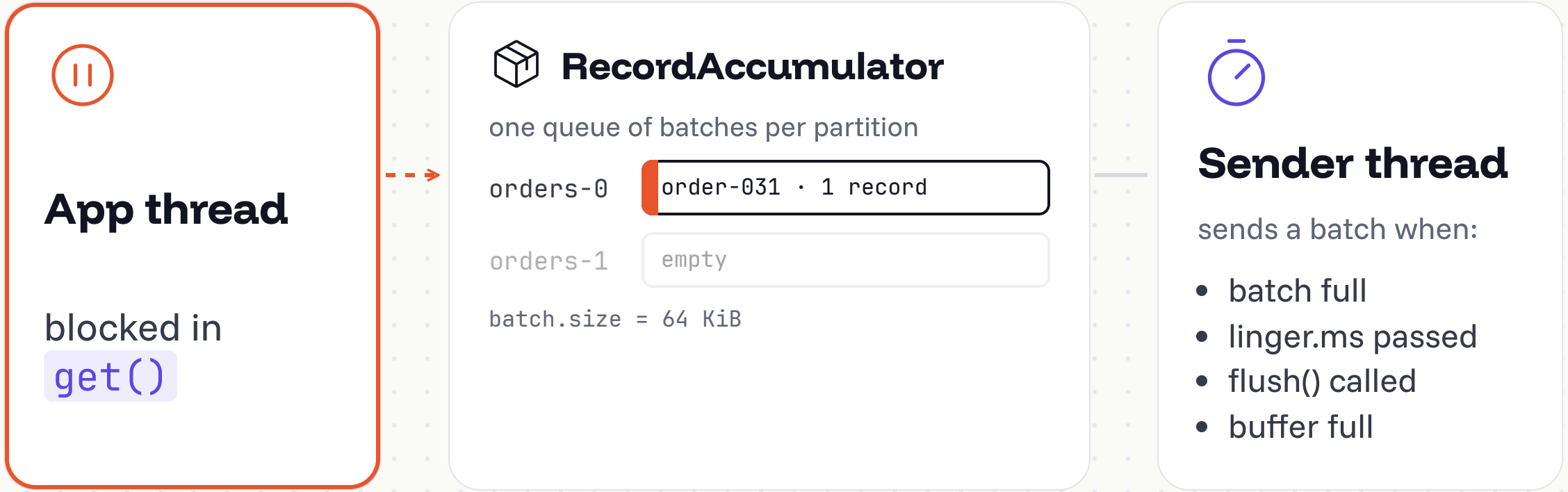
3.3 s 😱

the loop: 60 requests × 56 ms

71 ms

the same 60 in one request, in total

A batch waits up to linger.ms



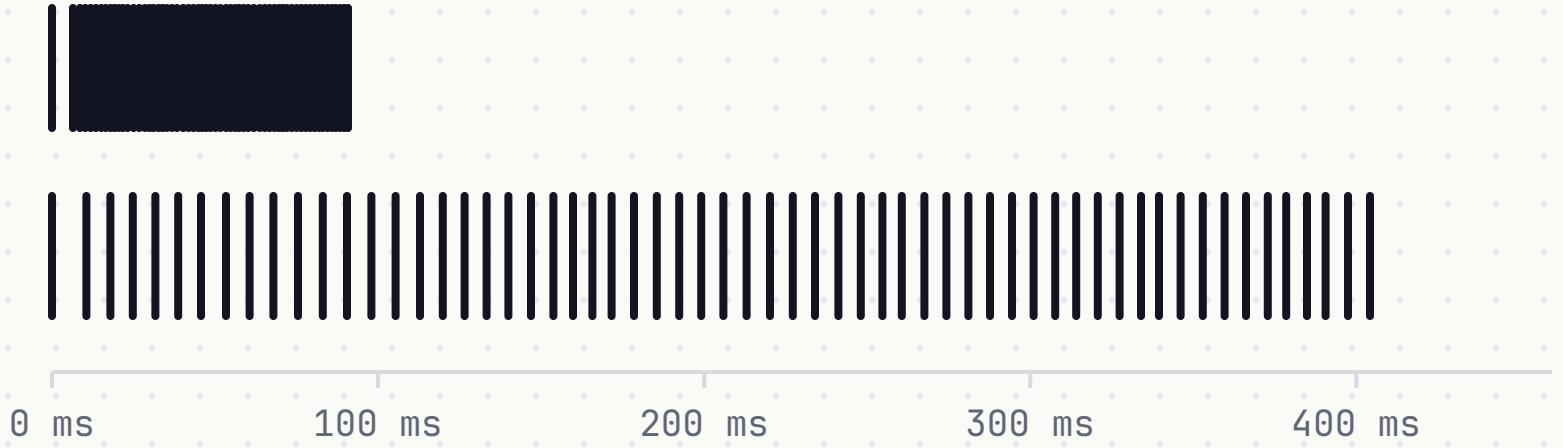
`get()` waits for the batch, so the batch never gets a second record.

Kafka 4 client: faster or slower?

same send().get() loop, no producer config · one bar per ProduceRequest

kafka-clients 3.9.2
defaults · linger.ms = 0

kafka-clients 4.3.1
defaults · linger.ms = 5



+5 ms per send after a client upgrade: linger.ms 0 → 5 ms.

Fix: send all, then wait

OrderSync.java

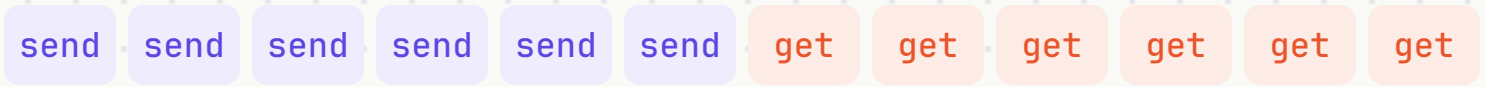
```
for (var o : orders) acks.add(producer.send(record(o)));
for (var ack : acks) ack.get();
```

`send().get()`
in a loop



60 Produce
3.3 s

**send all,
then get**



1 Produce
71 ms


Aborted
transactions

Who reads the aborted payment?

payments-service · PaymentWriter.java

```
producer.beginTransaction();
producer.send(payment);
producer.abortTransaction();
```

analytics · PaymentReader.java

```
new KafkaConsumer(props);
// isolation.level: not set
```

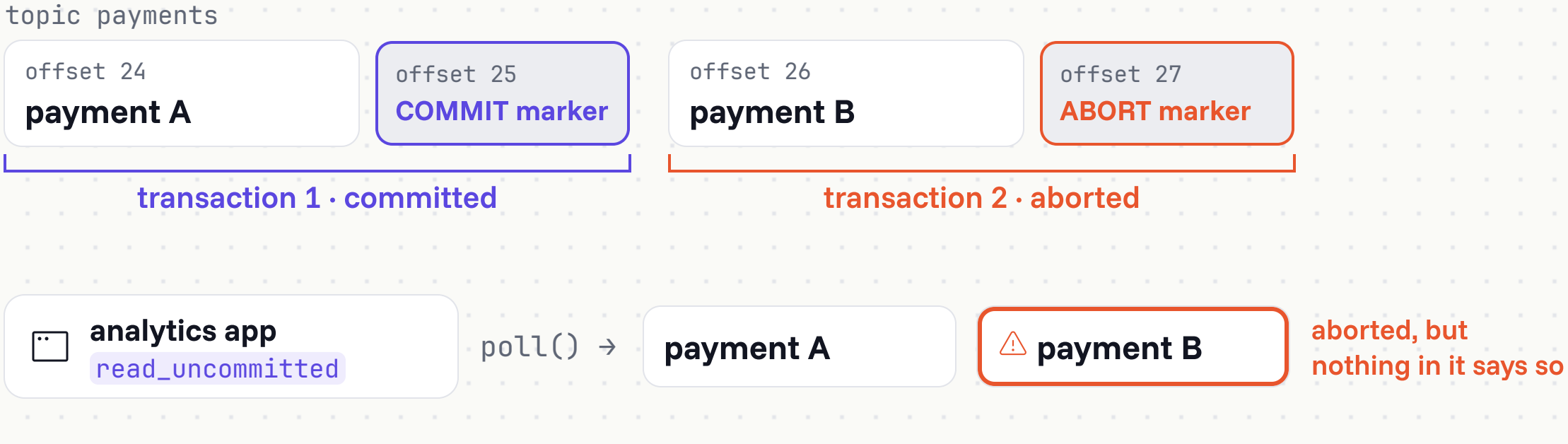
read_uncommitted — DEFAULT

- kafka-clients
- kafka-python
- Sarama
- franz-go
- kafka-go

read_committed — DEFAULT

- confluent-kafka
- confluent-kafka-go
- Confluent.Kafka
- confluent-kafka-js
- KafkaJS

poll() returns aborted records



Java producer:
“I ABORTED that payment!”

Sarama consumer (Go):
processing it anyway



Fetch: isolation_level = 0

cluster_id	null
replica_id	-1
▾ replica_state	{3}
replica_id	-1
replica_epoch	-1
max_wait_ms	500
min_bytes	1
max_bytes	52428800
isolation_level	0

`0` = read_uncommitted, the Java consumer default.

broker

sends records and markers, up to the LSO

Fetch →

client library

read_committed: filters

`CompletedFetch.java`

↓ payment B, aborted

→ **`poll()`**

committed records only

Fix: read_committed

consumer.properties

```
isolation.level = read_committed
```



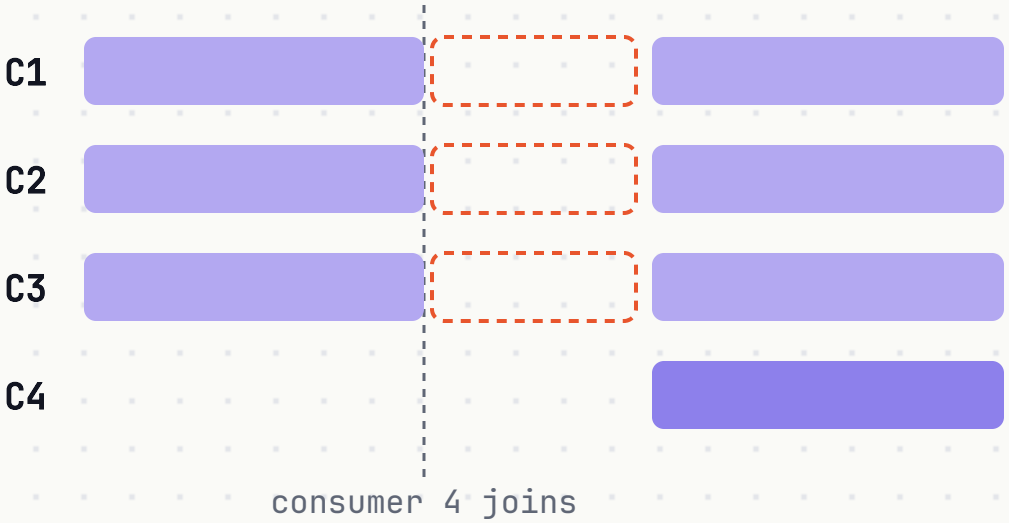
ⓘ Kafka Streams with exactly-once already reads `read_committed`.
Your other consumers don't.


Rebalancing

A 4th consumer joins: who pauses?

Eager

`classic protocol · range (default)`

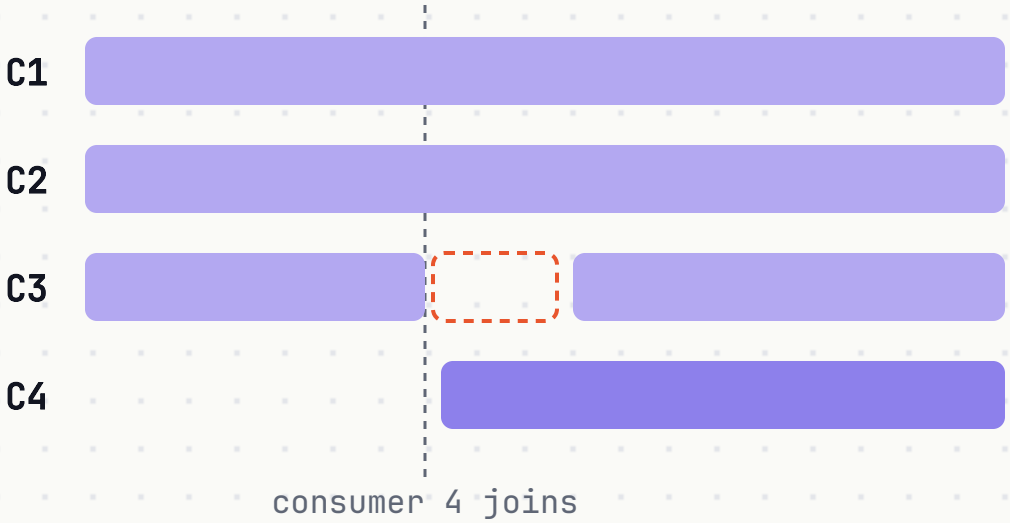


every member stops

`eager only:` KafkaJS · kafka-go

Incremental

`group.protocol=consumer (KIP-848)`



only the partition that moves pauses

Java 4.0+ · librdkafka 2.12+ · franz-go

Fix: `group.protocol=consumer`, or `cooperative-sticky`.



The rest of the consumer group, waiting for one slow member to rejoin an eager rebalance


Quotas

Same quota, two clients: what happens?

kafka-configs · on each client.id

```
--add-config producer_byte_rate=32768   # 32 KiB/s
```

Java

kafka-clients 4.3.1 · sends 1 MiB

Python

confluent-kafka (librdkafka 2.16.0) · sends 1 MiB

?

A Python is faster: it ignores the quota

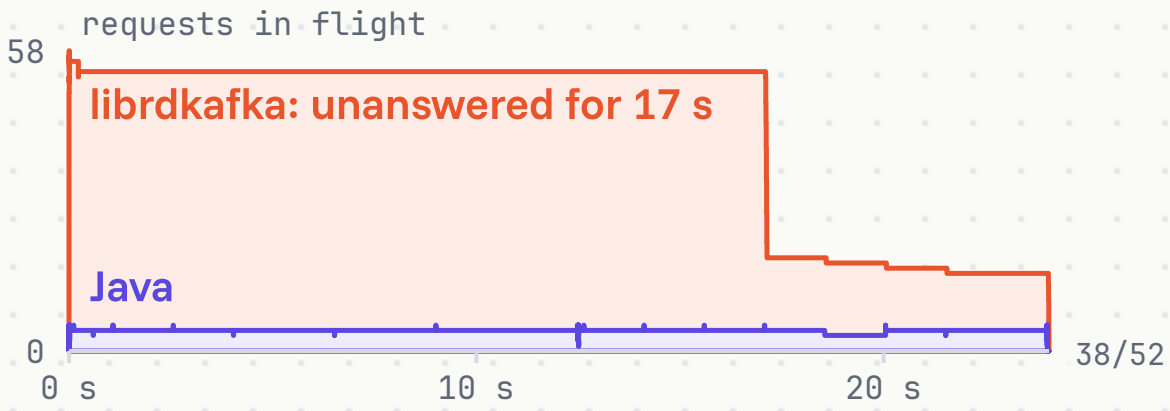
B both take the same time

C Python gets errors

Both take 24 s. Not the same way.

←	11:58:59.761901	+<1ms	ProduceResponse	v10	77 B	2 ·13	9.38 ms
←	11:58:59.981802	+220ms	ProduceResponse	v10	77 B	2 ·14	229 ms
←	11:58:59.982940	+1ms	ProduceResponse	v10	77 B	2 ·15	230 ms
←	11:58:59.983916	+<1ms	ProduceResponse	v10	77 B	2 ·16	231 ms
←	11:59:16.855364	+16.9s	ProduceResponse	v10	77 B	2 ·17	17.1 s
←	11:59:16.856524	+1ms	ProduceResponse	v10	77 B	2 ·18	17.1 s
←	11:59:16.857784	+1ms	ProduceResponse	v10	77 B	2 ·19	17.1 s
←	11:59:16.859123	+1ms	ProduceResponse	v10	77 B	2 ·20	17.1 s
←	11:59:16.860160	+1ms	ProduceResponse	v10	77 B	2 ·21	17.1 s

	JAVA	LIBRDKAFKA
in flight, at most	5	58
median round trip	3 ms	17.1 s
1 MiB sent in	24 s	24.1 s



throttle_time_ms: who's a good citizen?

waits 😇 — KIP-219

- kafka-clients
- Sarama 1.41+
- franz-go
- kafka-python 2.1+
- KafkaJS

keeps sending 🙉 — LIBRDKAFKA · KAFKA-GO

- confluent-kafka
- confluent-kafka-go
- Confluent.Kafka
- confluent-kafka-js
- kafka-go

✓ a proxy's `throttle_time_ms` slows them

✗ a proxy's `throttle_time_ms` is ignored

Fix: watch throttle time on every client: `produce-throttle-time-max` in Java, `throttle_cb` in librdkafka.


Reading faster

fetch.min.bytes=0, to read faster: how many Fetch per second?

consumer.properties

```
fetch.min.bytes = 0   # don't wait, read faster
```

one consumer · a topic with no new records

?

A about 2

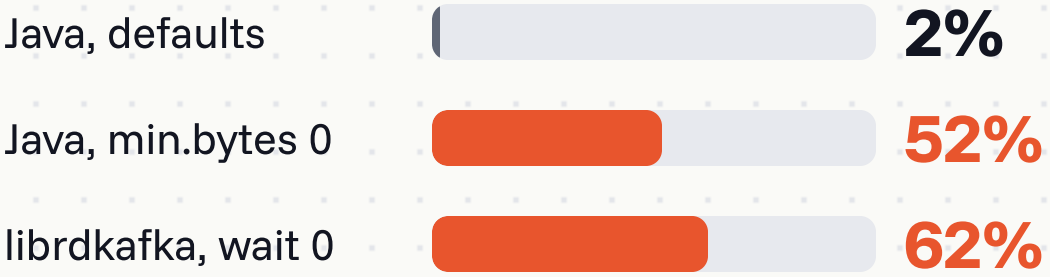
B about 50

C thousands

One idle consumer, 22,000 Fetch a second

	TIMESTAMP		API	SIZE	CONN·CORR	RTT
→	19:12:35.500561		FetchRequest v18	63 B	3 ·93586	–
→	19:12:35.500622	+<1ms	FetchRequest v18	63 B	3 ·93587	–
→	19:12:35.500683	+<1ms	FetchRequest v18	63 B	3 ·93588	–
→	19:12:35.500746	+<1ms	FetchRequest v18	63 B	3 ·93589	–
→	19:12:35.500808	+<1ms	FetchRequest v18	63 B	3 ·93590	–

broker CPU · one core · one consumer



ReplicaManager.scala · Kafka 4.3.1

```
if (maxWaitMs <= 0
    || bytesReadable >= minBytes)
  respond now   // 0 >= 0
```

Fix: keep `fetch.min.bytes` ≥ 1 and the fetch wait at 500 ms; watch Fetch requests per client.


A missing ACL

No ACL, no send(): how many requests in 10 s?

principal User:orders-app · password OK · no ACL on orders

OrderEvents.java

```
var producer = new KafkaProducer<String, String>(props);
// not a single send() yet
```

?

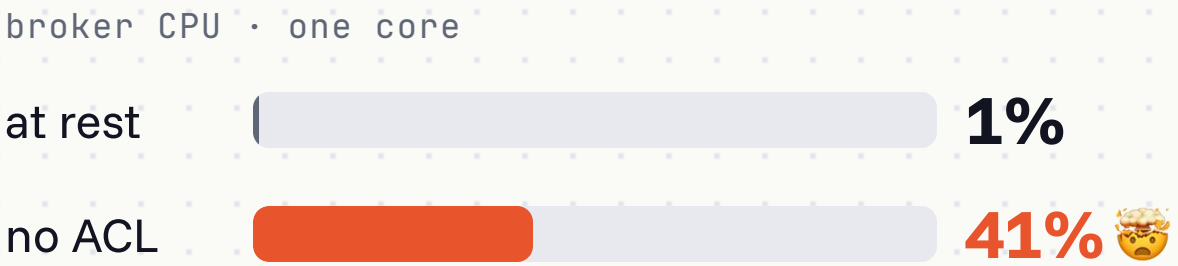
A 0

B 1

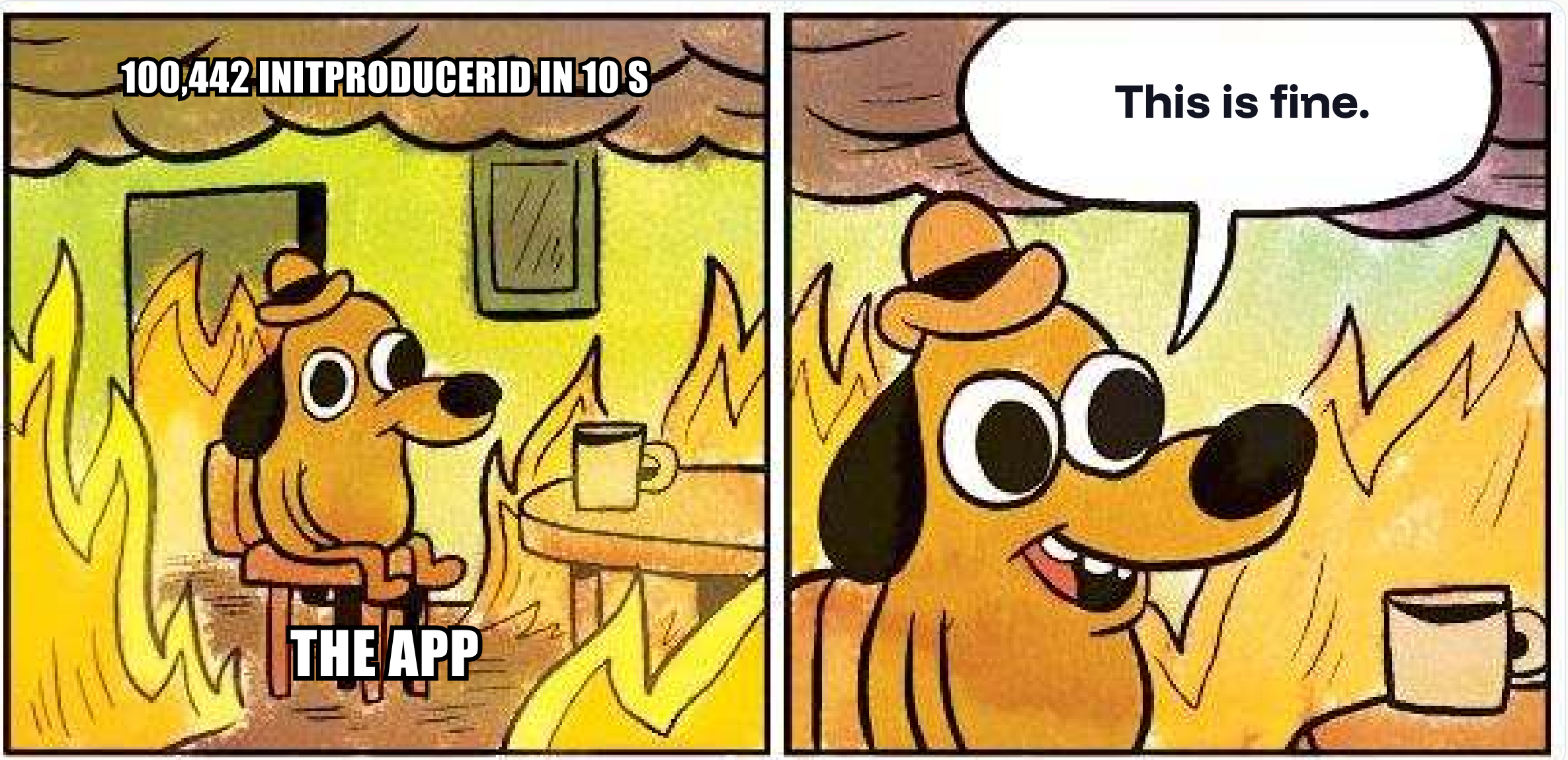
C about 100,000

No send(), 100,442 InitProducerId in 10 s

	TIMESTAMP		API	SIZE	CONN·CORR	RTT
←	11:33:06.992840		InitProducerIdResponse v5	26 B	2 ·97946	0.05 ms
←	11:33:06.992939	+<1ms	InitProducerIdResponse v5	26 B	2 ·97947	0.05 ms
←	11:33:06.993045	+<1ms	InitProducerIdResponse v5	26 B	2 ·97948	0.04 ms
←	11:33:06.993135	+<1ms	InitProducerIdResponse v5	26 B	2 ·97949	0.04 ms
←	11:33:06.993226	+<1ms	InitProducerIdResponse v5	26 B	2 ·97950	0.04 ms



Fix: grant WRITE before you deploy; alert on the broker's `ErrorsPerSec` for InitProducerId.


100,442 INITPRODUCERID IN 10 S
THE APP
This is fine.


Exactly-once

200 records, exactly-once: how many transactions?

`Kafka Streams · processing.guarantee = exactly_once_v2`

200 records arrive over 8 seconds.

?

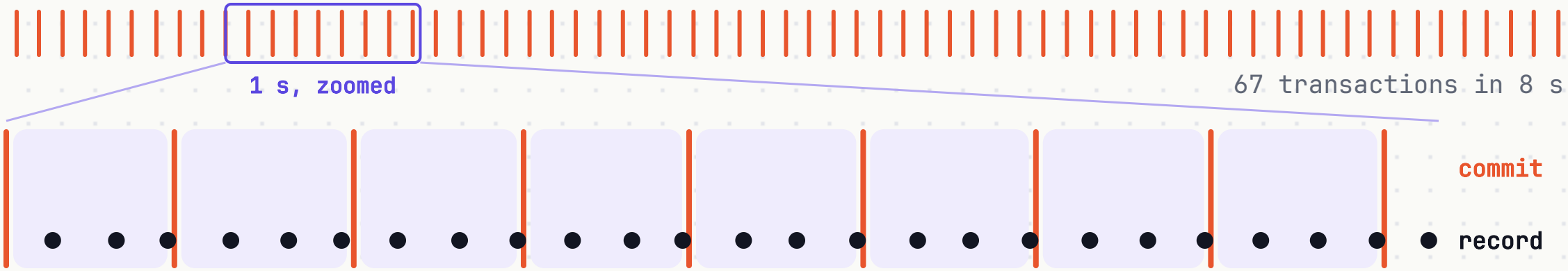
A about 10

B about 70

C 200

A transaction every ~100 ms

`processing.guarantee = exactly_once_v2` → `commit.interval.ms` ~~30000~~ 100 Streams default



3.9 client	`AddPartitionsToTxn` `AddOffsetsToTxn` `TxnOffsetCommit` `EndTxn`	**4 requests**
4.x client, txn v2 broker adds them · KIP-890	~~`AddPartitionsToTxn`~~ ~~`AddOffsetsToTxn`~~ `TxnOffsetCommit` `EndTxn`	**2 requests**

Kafka 4 drops two requests per commit. The cadence stays.

commit.interval.ms vs latency

`commit.interval.ms = 100`

transactions — 67

output visible — 47 ms

`commit.interval.ms = 1000`

transactions — 10

output visible — 490 ms

Fix: keep `commit.interval.ms` at 100 ms only if you need sub-second output; otherwise raise it.

Troubleshoot locally. Protect centrally.

Kapture

on your laptop

	TIMESTAMP		API		SIZE	CONN·CORR	RTT
→	11:43:09.207419		JoinGroupRequest	v9	192 B	4 ·4	–
→	11:43:09.208632	+1ms	JoinGroupRequest	v9	360 B	4 ·6	–
→	11:43:12.222861	+3.0s	SyncGroupRequest	v5	248 B	4 ·7	–
→	11:43:13.279214	+1.1s	HeartbeatRequest	v4	129 B	4 ·16	–
→	11:43:14.321262	+1.0s	HeartbeatRequest	v4	129 B	4 ·17	–
→	11:43:15.362909	+1.0s	HeartbeatRequest	v4	129 B	4 ·18	–
→	11:43:15.882686	+520ms	LeaveGroupRequest	v5	162 B	4 ·19	–
→	11:43:17.307037	+1.4s	JoinGroupRequest	v9	231 B	4 ·20	–
→	11:43:17.309769	+3ms	JoinGroupRequest	v9	360 B	4 ·21	–

Conduktor Gateway

in front of Kafka



Your business depends on Kafka. Protect it where every client meets it.

Thank you

Kapture
github.com/conduktor/kapture

Conduktor
conduktor.io/get-started

