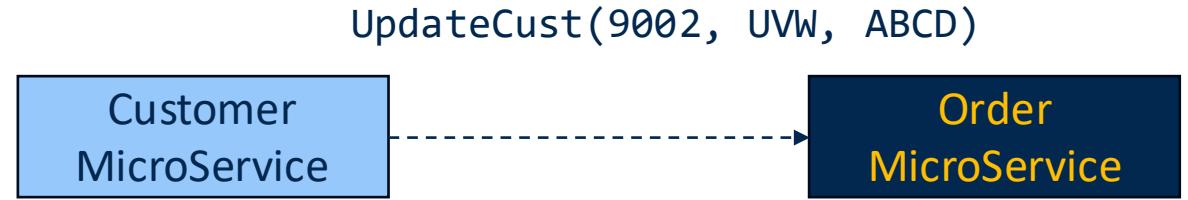


# Kafka: a use case for high-throughput data systems

Tapti Palit

# The problem with request/response

- Caller blocks until the response arrives
- If the callee is down, the caller fails or must retry



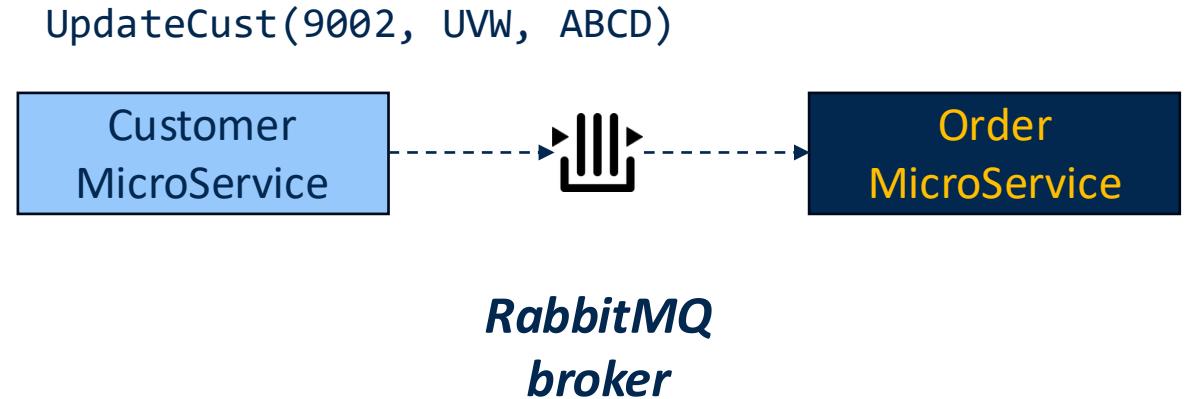
# The problem with request/response

- Caller blocks until the response arrives
- If the callee is down, the caller fails or must retry



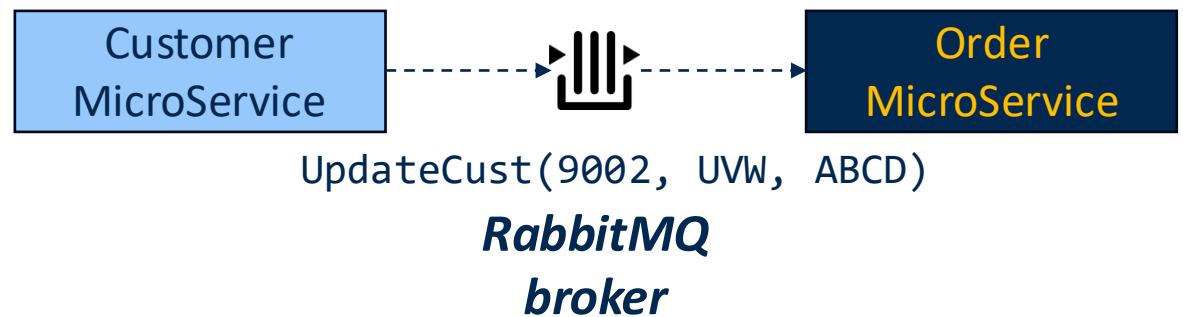
# The problem with request/response

- Message queues mitigate some of these problems
- But still, the caller must know the callee (tight coupling)



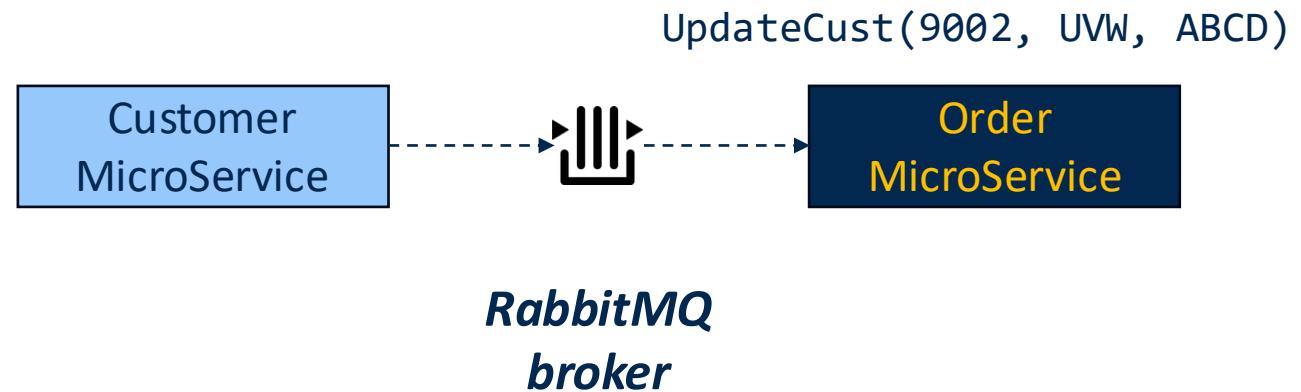
# The problem with request/response

- Message queues mitigate some of these problems
- But still, the caller must know the callee (tight coupling)



# The problem with request/response

- Message queues mitigate some of these problems
- But still, the caller must know the callee (tight coupling)



*\* Note that the broker here deletes the message once it's consumed*

# Messages vs events

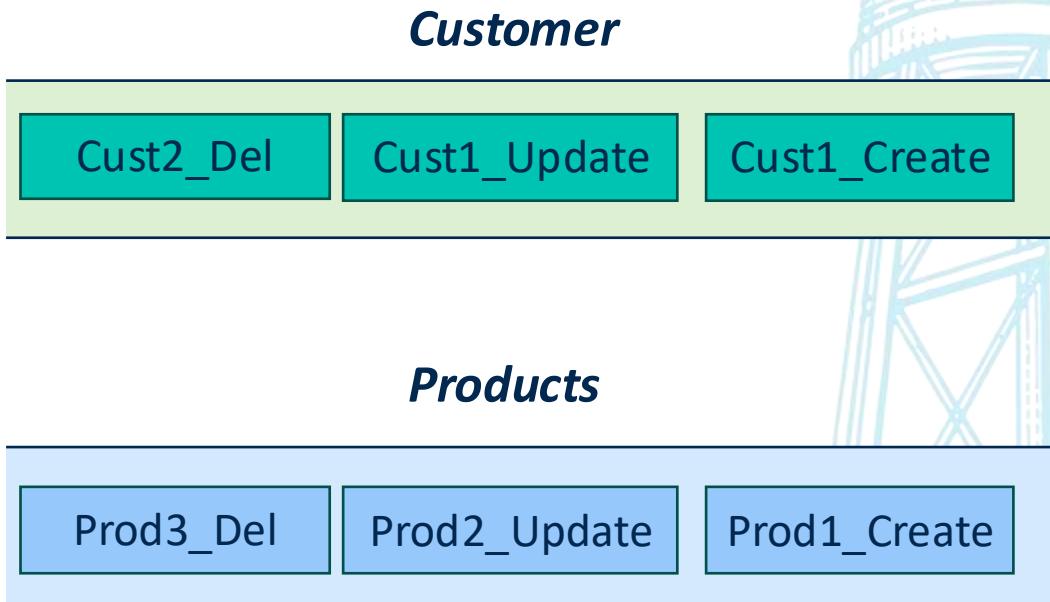
- Kafka uses events instead of messages or commands
- Message or command: `UpdateCust(9002, UVW, ABCD)`
  - Directed at a specific service (Order microservice)
  - Implies a known recipient and expected action
- Event: `CustomerUpdated(9002, UVW, ABCD)`
  - A fact that happened
  - Implies no specific recipient – anyone can subscribe

# What is Apache Kafka?

- A distributed event streaming platform
- Originally built at LinkedIn for website activity data
- Open-sourced in 2011, graduated from Apache Incubator in 2012
- Used at Netflix, LinkedIn, Uber, and thousands of companies
- Forms the substrate on which many streaming frameworks operate
  - Apache Flink, Apache Spark
- Processes trillions of events per day at scale

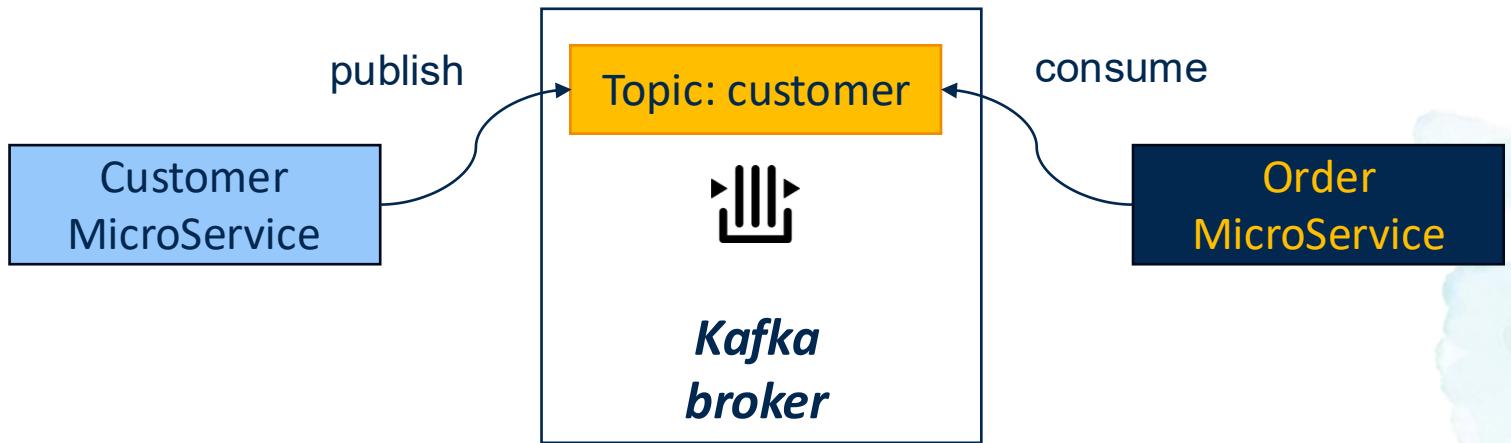
# Kafka topics

- Kafka events are organized into “topics”
- A topic is a named logical abstraction
  - A feed of related events
  - E.g: ‘customers’, ‘orders’, ‘products’



# Kafka topics

- Topics are units of subscription:
  - Producers write events to a topic
  - Consumers subscribe to one or more topics



- Topics also help improve concurrency (more later)

# The Kafka Record

- All Kafka events have the same record format

Key (optional)	Value	Timestamp	Headers
----------------	-------	-----------	---------

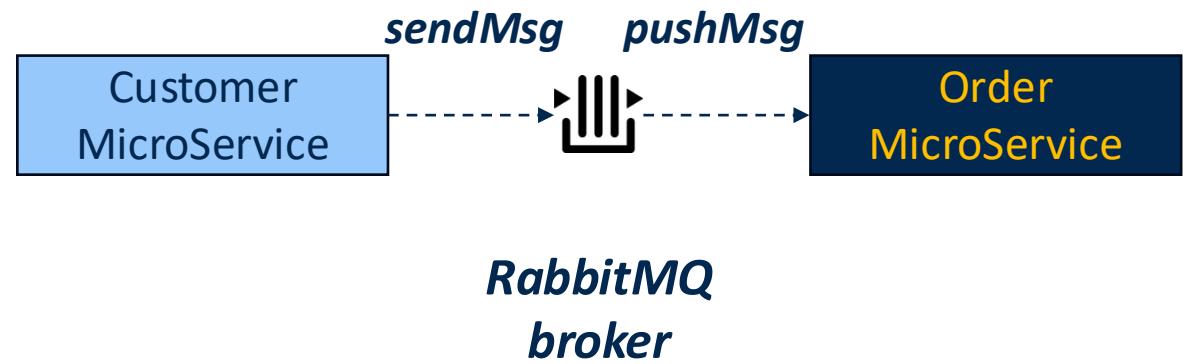
- Keys and values are byte arrays
- Serialization and deserialization is the client's responsibility
  - For example, using protobuf

# Events as immutable facts

- Once an event is written to Kafka, it cannot be modified or deleted
- An event is a fact: “CustomerUpdated at time T with these details”
- If something changes, produce a new event: “CustomerUpdated at time T+1”
- The log captures the complete history of what happened
- Almost as a database: “***Data on the outside***”

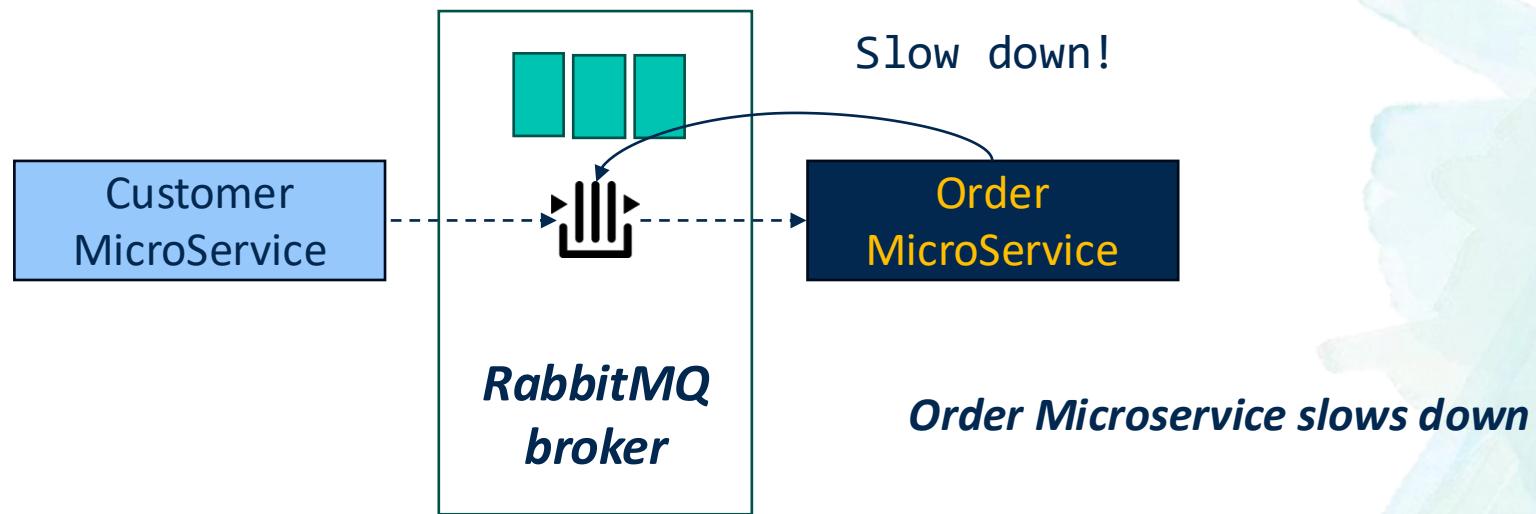
# Consumer basics – pull vs. push

- Traditional message brokers (e.g., RabbitMQ): push model
  - Broker actively sends messages to each consumer



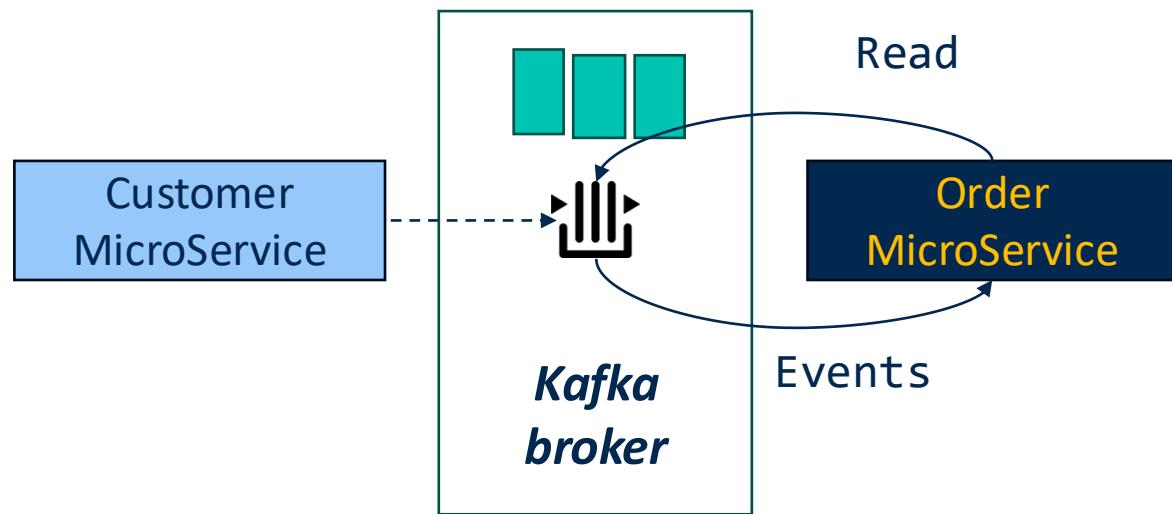
# Consumer basics – pull vs. push

- Broker must know consumer speed / capacity
- Slow consumer → broker queues messages or drops them
- Need separate (often complex) backpressure mechanisms
  - Usually done by withholding message acknowledgements



# Consumer basics – pull vs. push

- Kafka: pull model — consumer requests messages from broker
- Consumer controls its own read rate and throughput
- Scales easily — broker doesn't track consumer state
- Consumer-driven: each app controls its own consumption rate



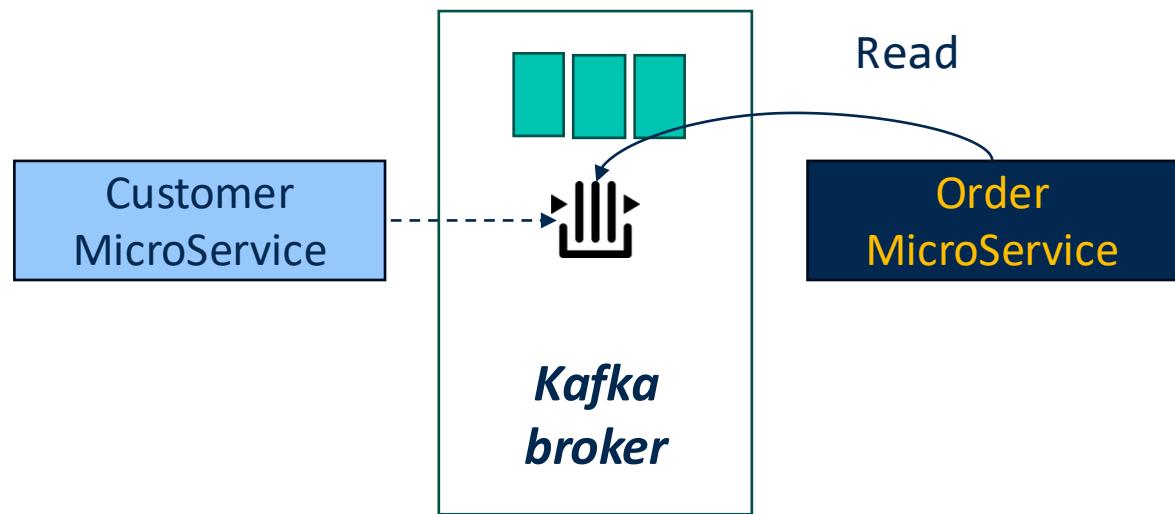
# Consumer basics – pull vs. push

- Kafka: pull model — consumer requests messages from broker
- Consumer controls its own read rate and throughput
- Scales easily — broker doesn't track consumer state
- Consumer-driven: each app controls its own consumption rate



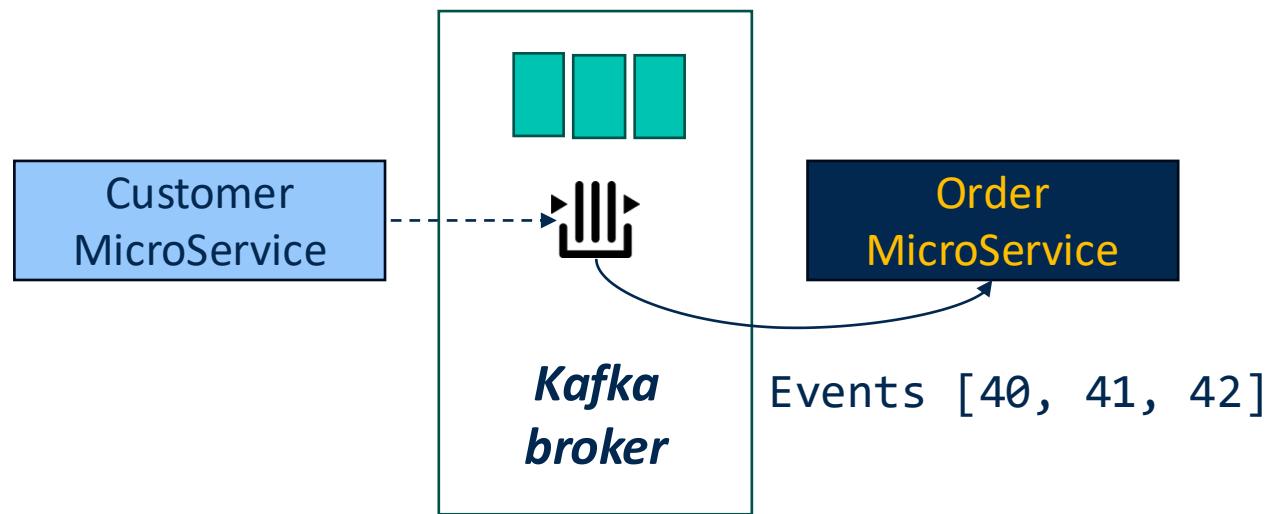
# Offsets as Consumer State

- Consumer can crash; needs to remember offset read
- Unlike traditional MQs, the consumer must commit the offset up to which it has processed
- Kafka broker will remember the last committed offset



# Offsets as Consumer State

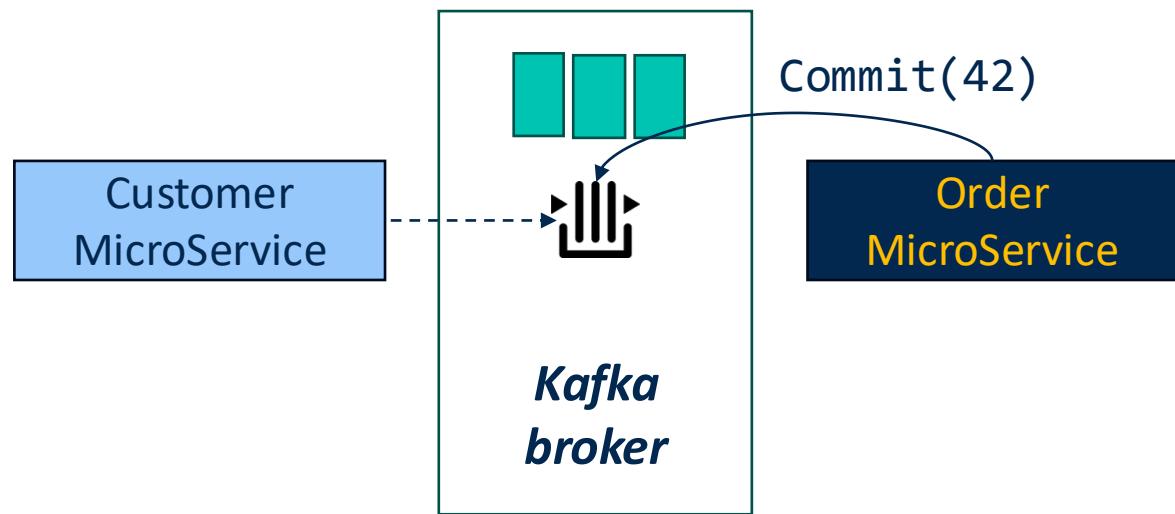
- Consumer can crash; needs to remember offset read
- Unlike traditional MQs, the consumer must commit the offset up to which it has processed
- Kafka broker will remember the last committed offset



*Note: events persist in Kafka broker even after they are read by consumer*

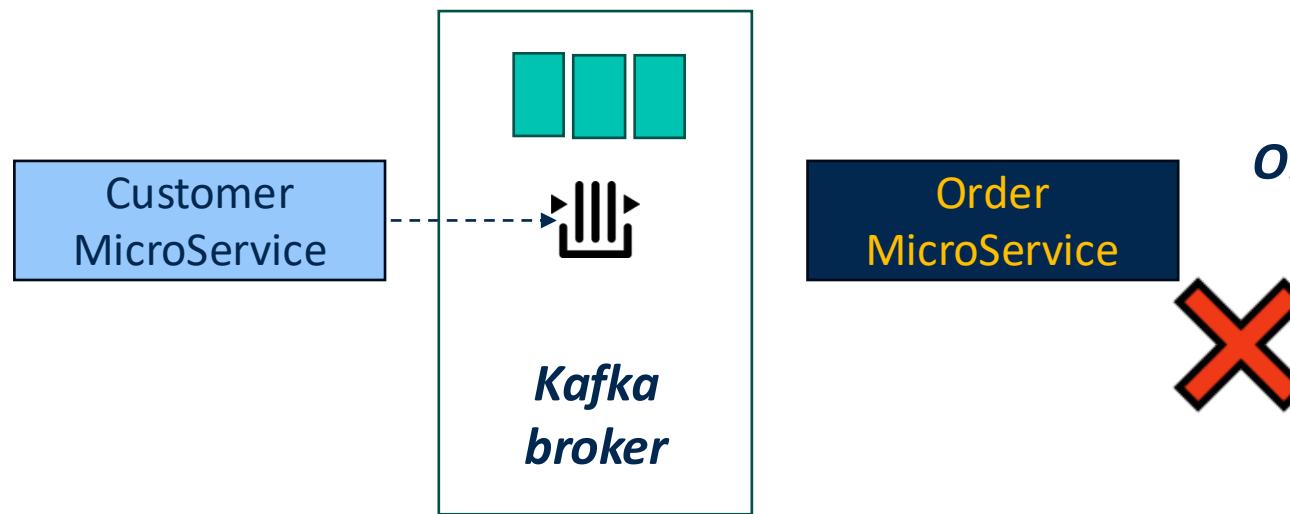
# Offsets as Consumer State

- Consumer can crash; needs to remember offset read
- Unlike traditional MQs, the consumer must commit the offset up to which it has processed
- Kafka broker will remember the last committed offset



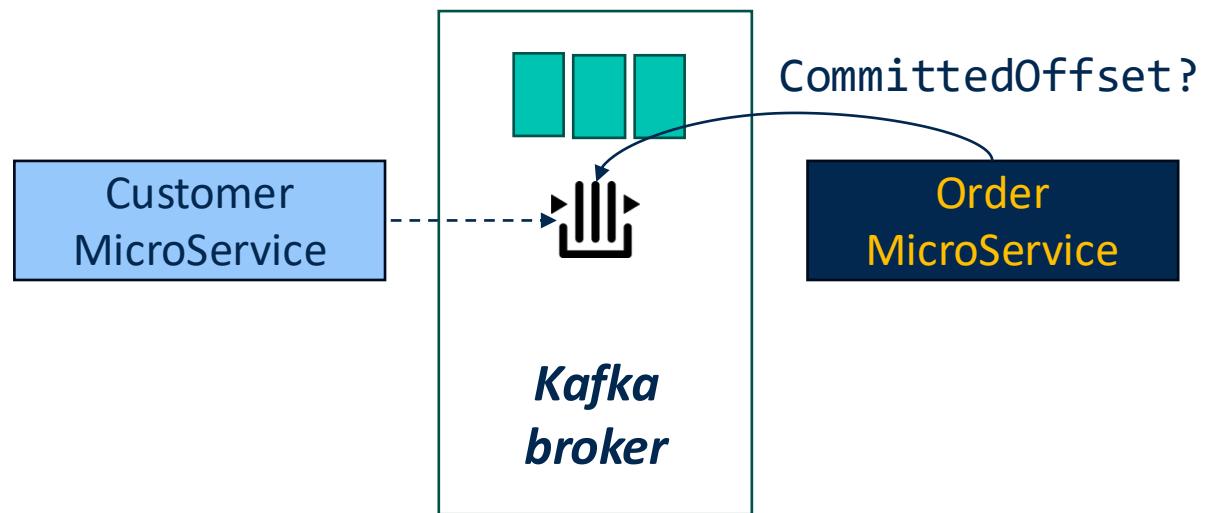
# Offsets as Consumer State

- Consumer can crash; needs to remember offset read
- Unlike traditional MQs, the consumer must commit the offset up to which it has processed
- Kafka broker will remember the last committed offset



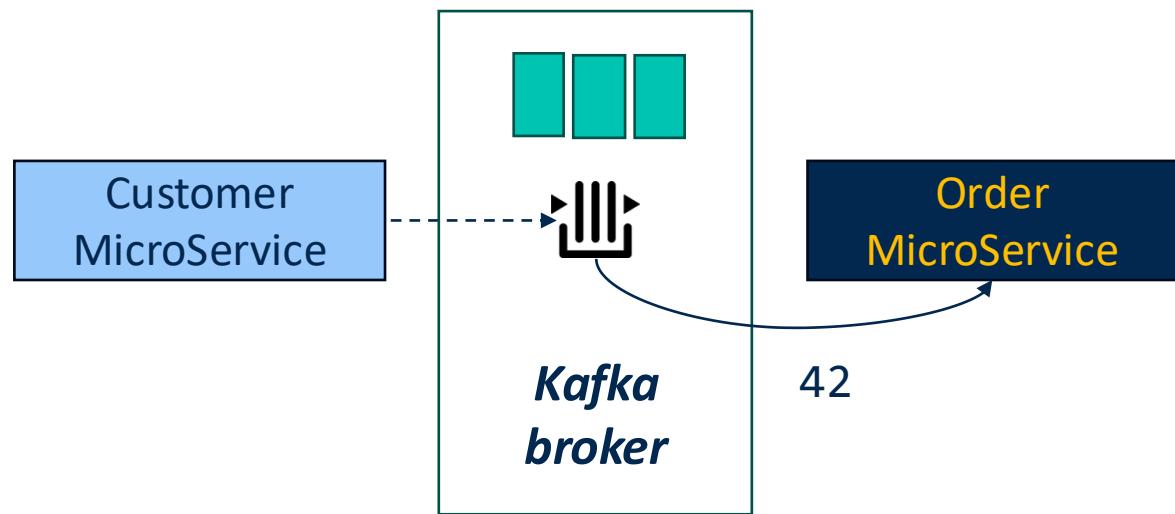
# Offsets as Consumer State

- Consumer can crash; needs to remember offset read
- Unlike traditional MQs, the consumer must commit the offset up to which it has processed
- Kafka broker will remember the last committed offset



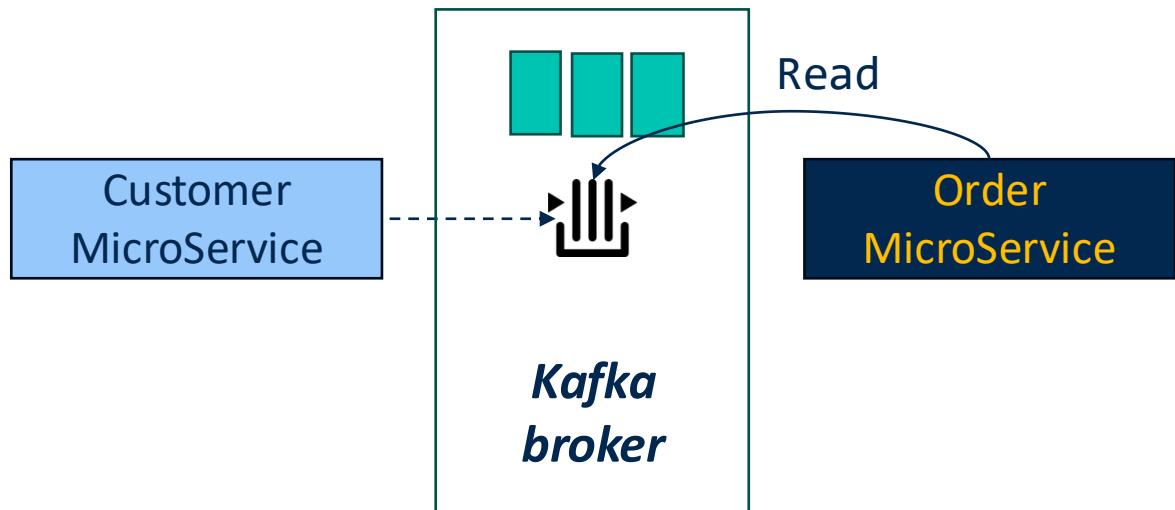
# Offsets as Consumer State

- Consumer can crash; needs to remember offset read
- Unlike traditional MQs, the consumer must commit the offset up to which it has processed
- Kafka broker will remember the last committed offset



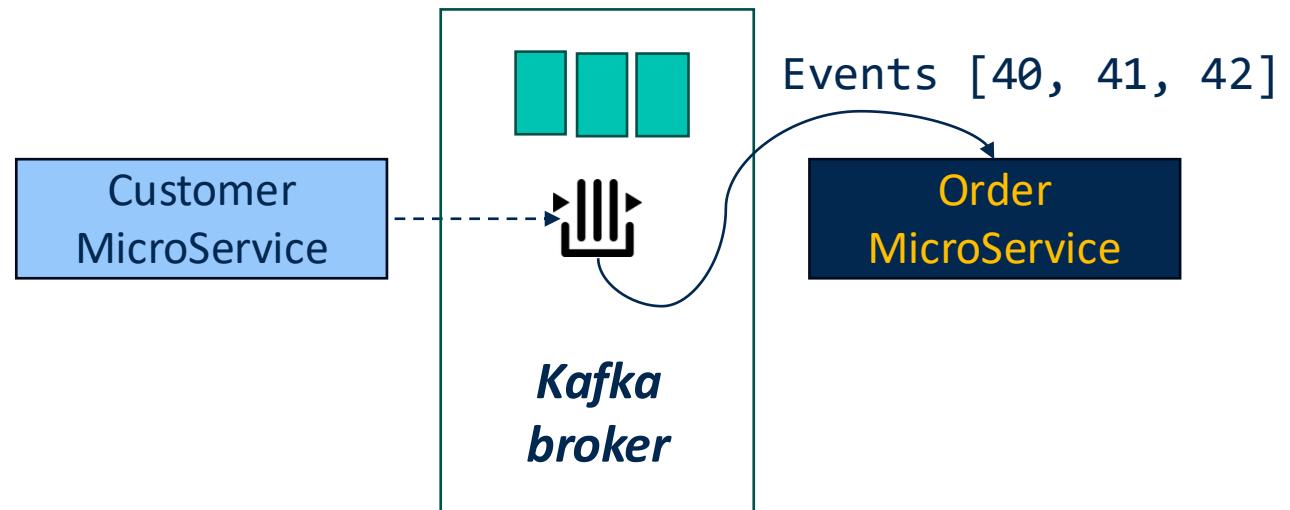
# At-most once vs. at-least once delivery guarantees

- When the consumer commits determines delivery semantics
- Before processing -> **at-most-once** delivery guarantees



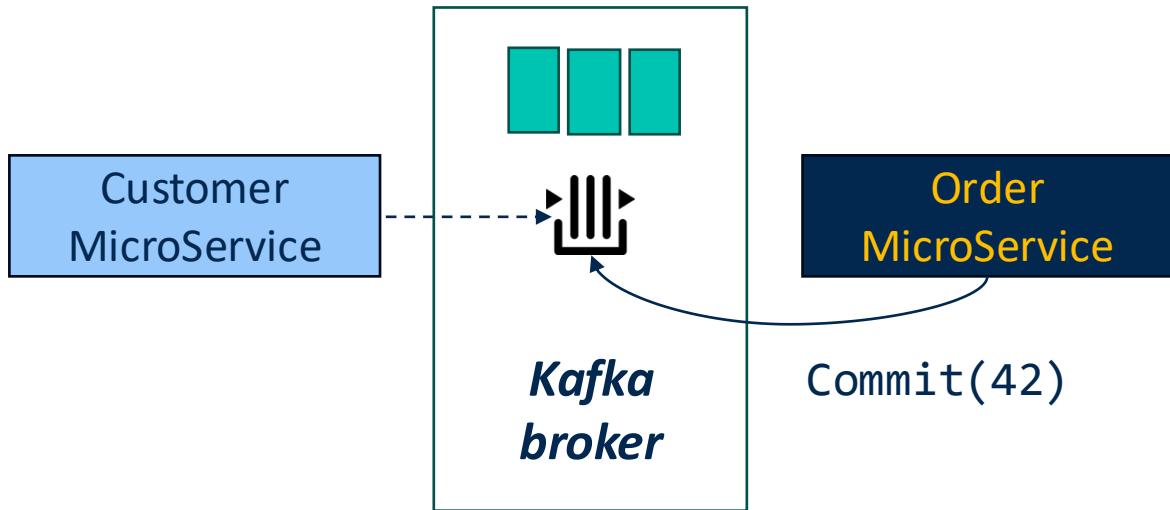
# At-most once vs. at-least once delivery guarantees

- When the consumer commits determines delivery semantics
- Before processing -> **at-most-once** delivery guarantees



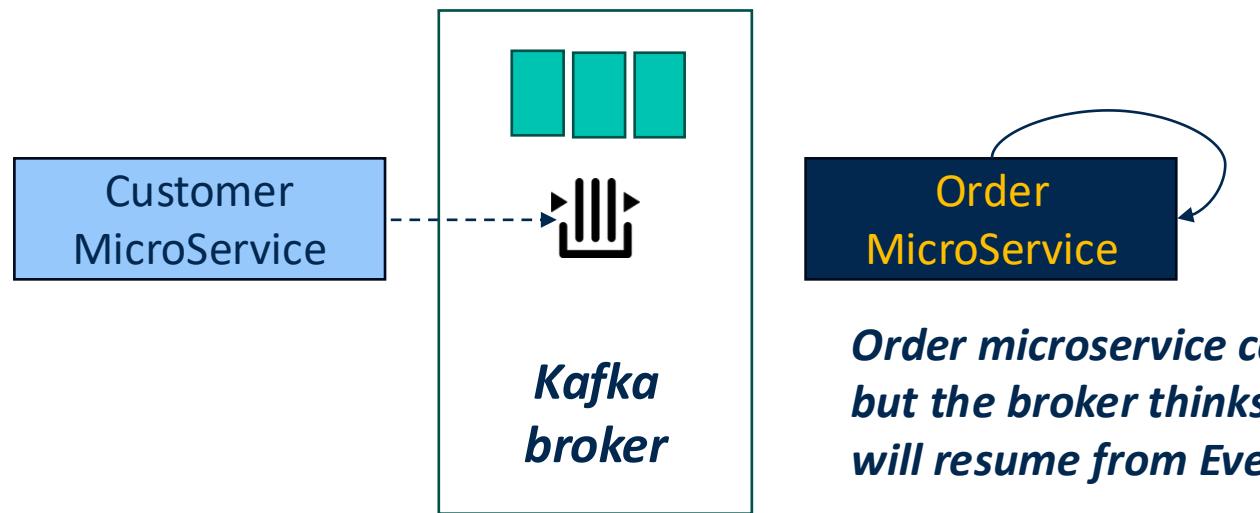
# At-most once vs. at-least once delivery guarantees

- When the consumer commits determines delivery semantics
- Before processing -> **at-most-once** delivery guarantees



# At-most once vs. at-least once delivery guarantees

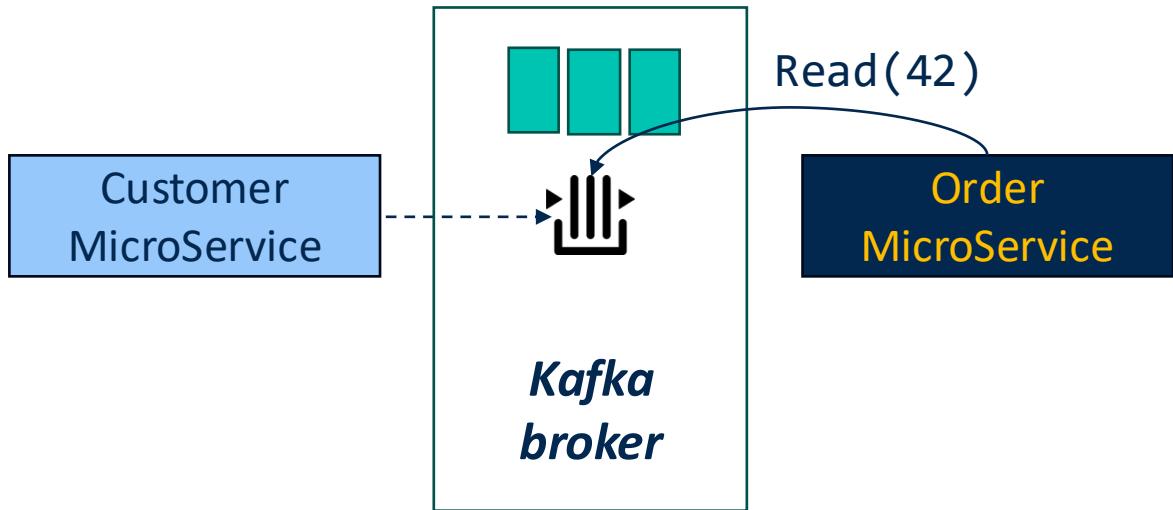
- When the consumer commits determines delivery semantics
- Before processing -> **at-most-once** delivery guarantees



*Order microservice can crash before processing;  
but the broker thinks Event 42 is consumed and  
will resume from Event 43*

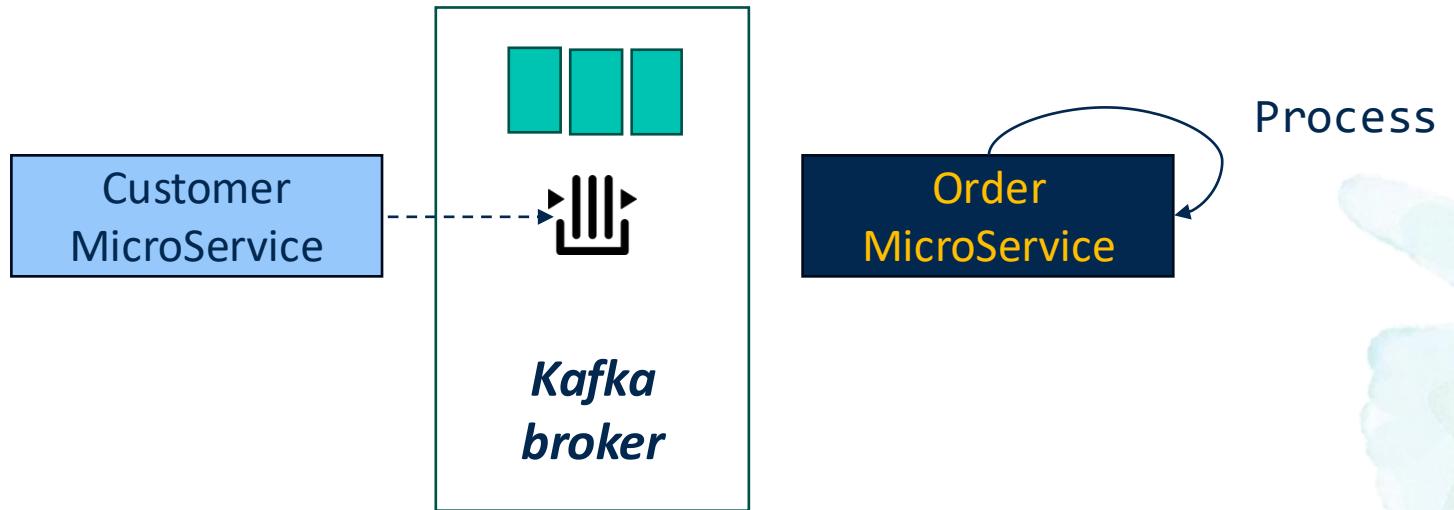
# At-most once vs. at-least once delivery guarantees

- When the consumer commits determines delivery semantics
- After processing -> **at-least-once** delivery guarantees



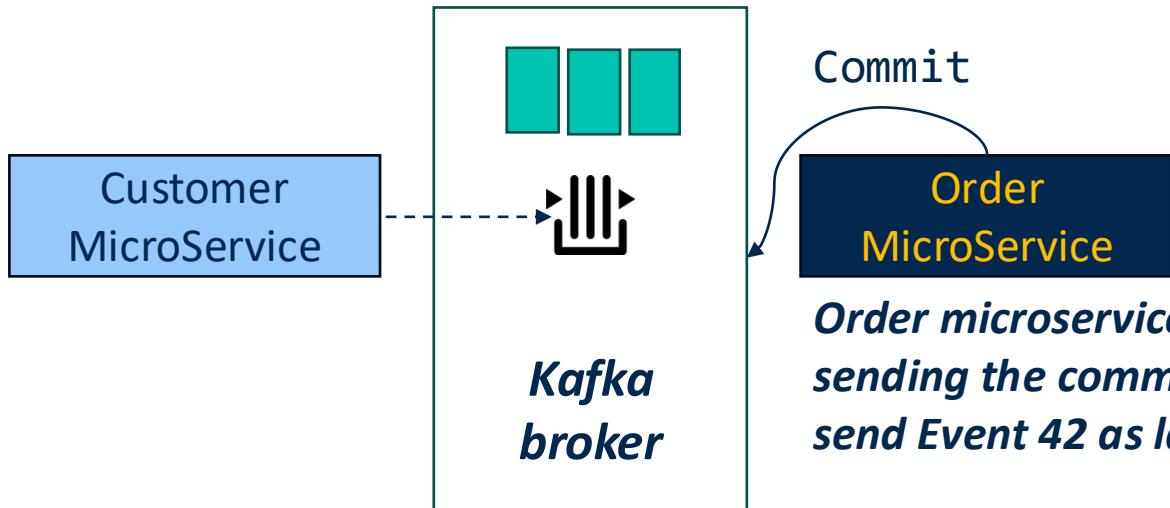
# At-most once vs. at-least once delivery guarantees

- When the consumer commits determines delivery semantics
- After processing -> **at-least-once** delivery guarantees



# At-most once vs. at-least once delivery guarantees

- When the consumer commits determines delivery semantics
- After processing -> **at-least-once** delivery guarantees



# At-most once vs. at-least once delivery guarantees

- Selecting the correct delivery guarantees is use-case specific
- At-most vs at-least delivery guarantees assumes client is relying solely on the broker to maintain its state
- Client can maintain state locally to augment broker state maintenance

# Kafka Architecture

A systems perspective



# Kafka is a distributed log

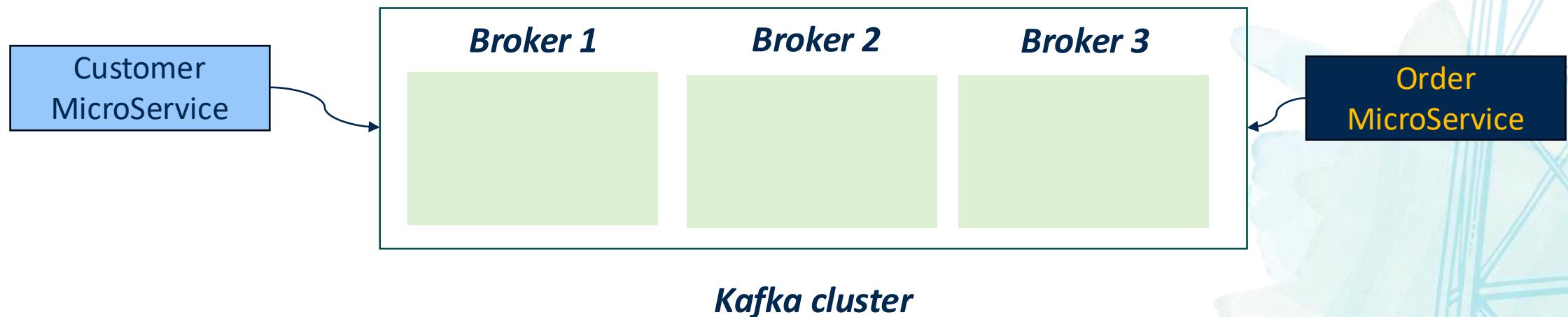
- Kafka is an append-only, ordered, immutable log
- Every event is appended to the end of a log
- Events are never modified or deleted (within the retention period)
- Note: unlike SSTables and LSM trees, consumers can only read sequentially

# The scalability problem

- A single log on a single machine has limited throughput
- What if a topic receives millions of events per second?
- A single machine's disk I/O and network bandwidth become bottlenecks
- Solution: split the topic's log across machines

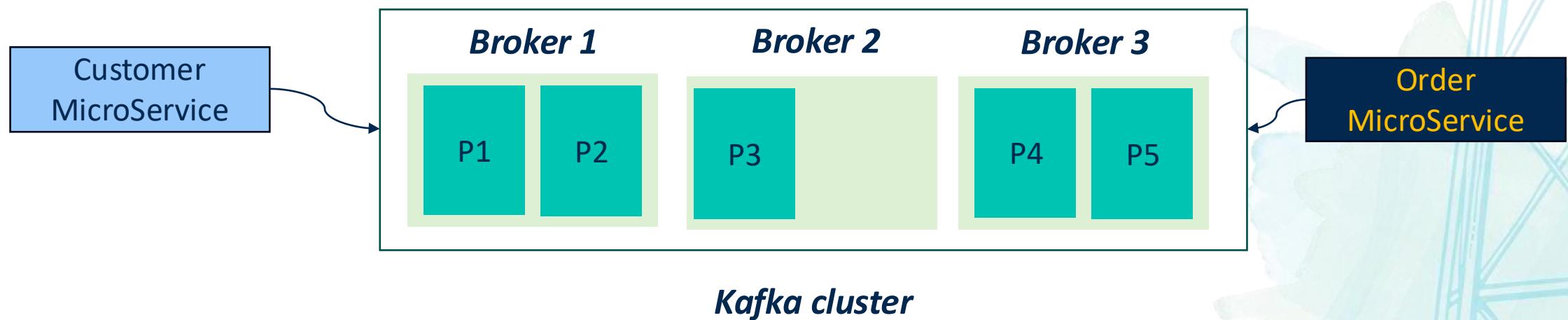
# Scaling Kafka

- Kafka usually consists of a cluster of brokers
- Each broker is a single node or machine



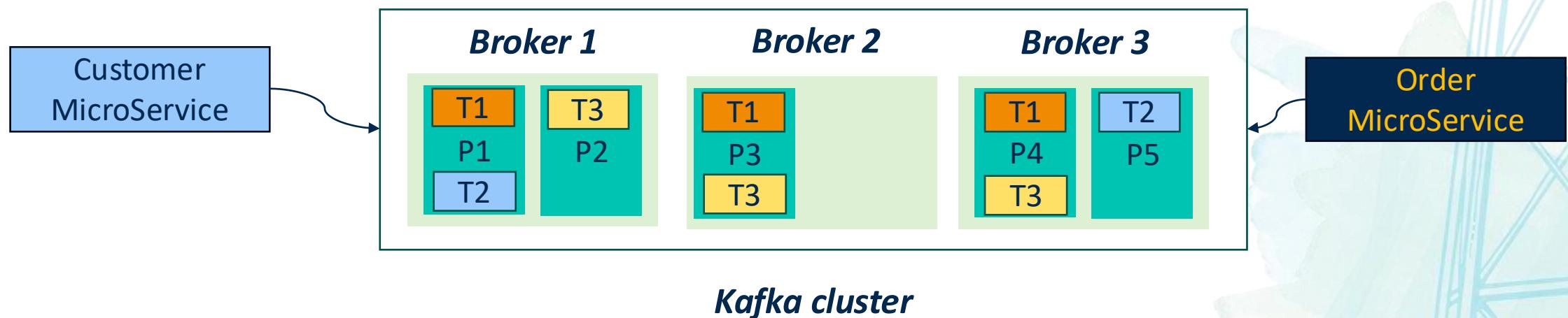
# Scaling Kafka

- The logical “log” is divided into physical partitions
- A partition is a single, ordered, immutable log
- Each partition lives on a single broker



# Scaling Kafka

- Each topic is divided into one or more partitions
- A topic *can* live in many partitions on the same broker, but that limits concurrency – ideally a topic is distributed among partitions on different brokers



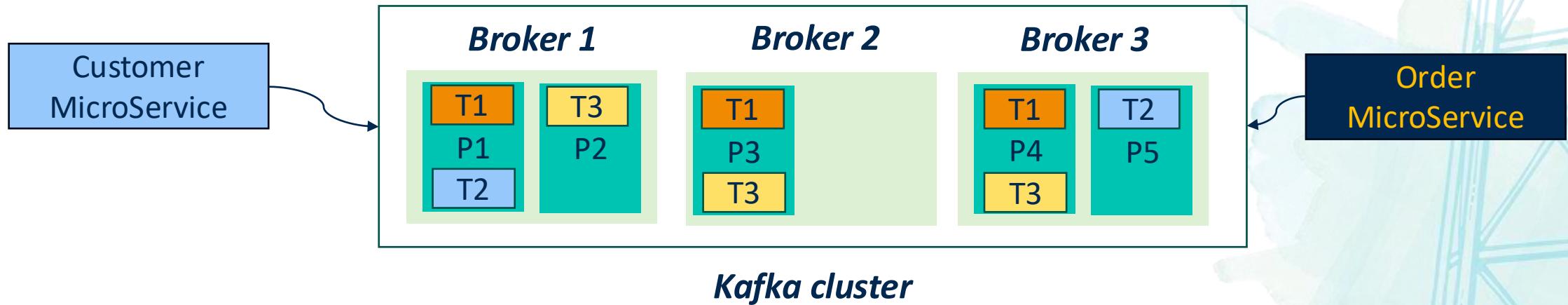
# Ordering within partitions

- Ordering is guaranteed within a partition, NOT across partitions
  - Events in partition 0 are ordered among themselves
  - Events in partition 1 are ordered among themselves
  - No ordering guarantee between partition 0 and partition 1
- Key trade-off: more partitions = more throughput but weaker global ordering

# Message to partition mapping

Each topic has events/messages/records under it

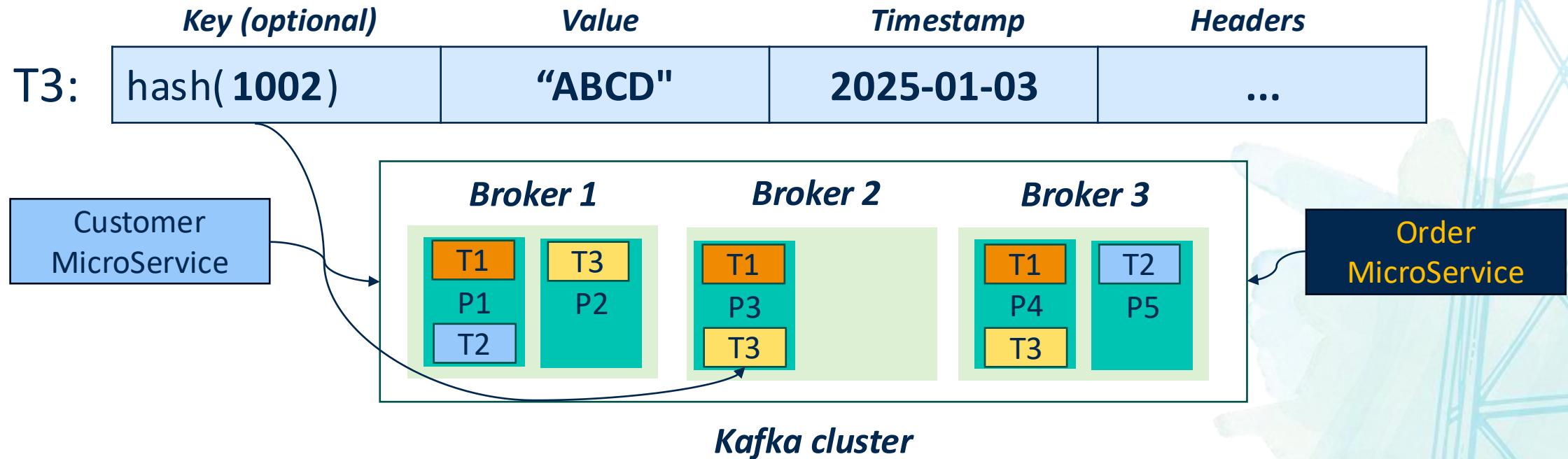
	<i>Key (optional)</i>	<i>Value</i>	<i>Timestamp</i>	<i>Headers</i>
T3:	1002	"ABCD"	2025-01-03	...



*Question: which partition should the event record be stored?*

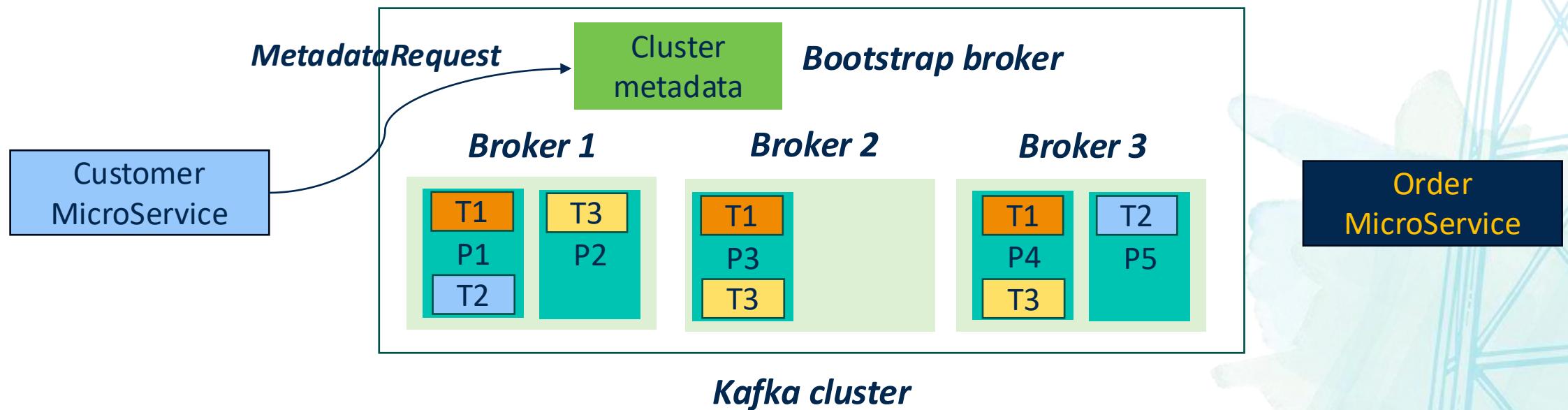
# Message to partition mapping

- partition = hash(key) % num\_partitions
- no key → round-robin partitions



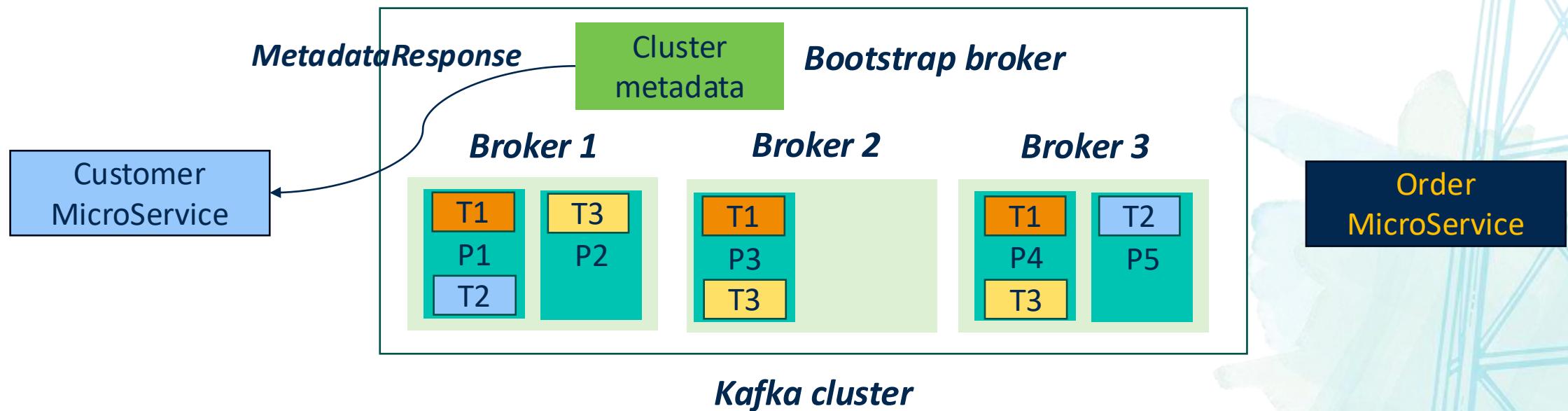
# Bootstrap broker

- Kafka clients connect to "bootstrap broker" to know the partitions for a topic



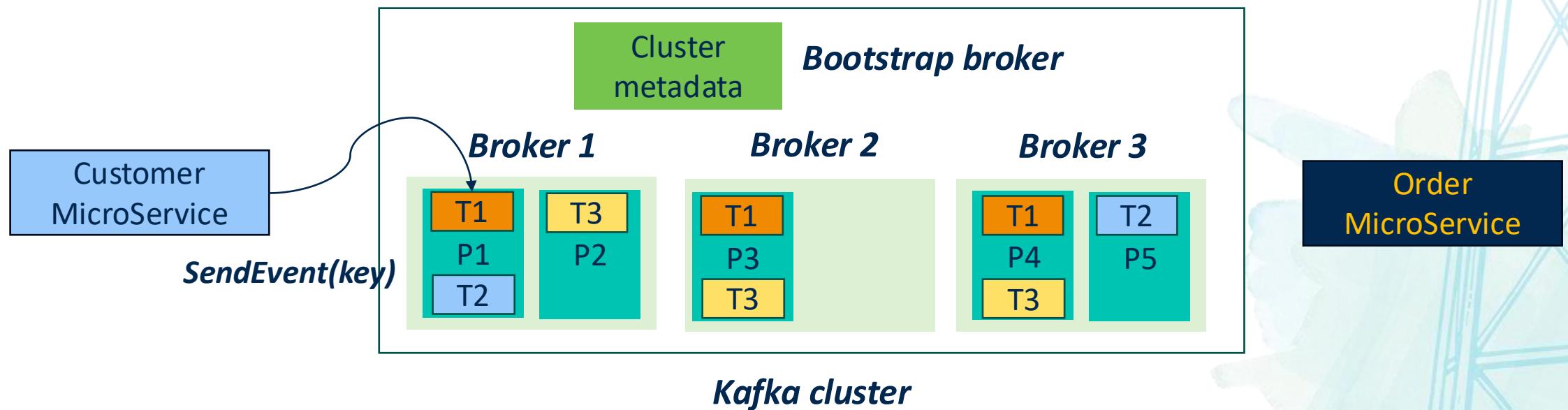
# Bootstrap broker

- Kafka clients connect to "bootstrap broker" to know the partitions for a topic



# Bootstrap broker

- Kafka clients connect to "bootstrap broker" to know the partitions for a topic
- Can compute the partition for a particular key using  $\text{hash}(\text{key}) \% \text{num\_partitions}$



# Implications of key-based partitioning

- All events with the same key land in the same partition
- Guarantees ordering for events with the same key
- For example, the “key” can be the customer ID, ensuring all events for a particular customer goes to the same partition
- Incorrect partitioning schemes can also result in imbalanced clusters
  - Reduces throughput
- Conclusion: need to design partitioning correctly

# Kafka architecture summary

- Kafka cluster consists of multiple brokers (servers)
  - Each broker hosts some partitions of various topics
- Producers send events to specific topic-partitions
- Consumers read events from specific topic partitions
- A metadata system coordinates the cluster
- ***Remaining challenge:*** what happens if a broker fails?

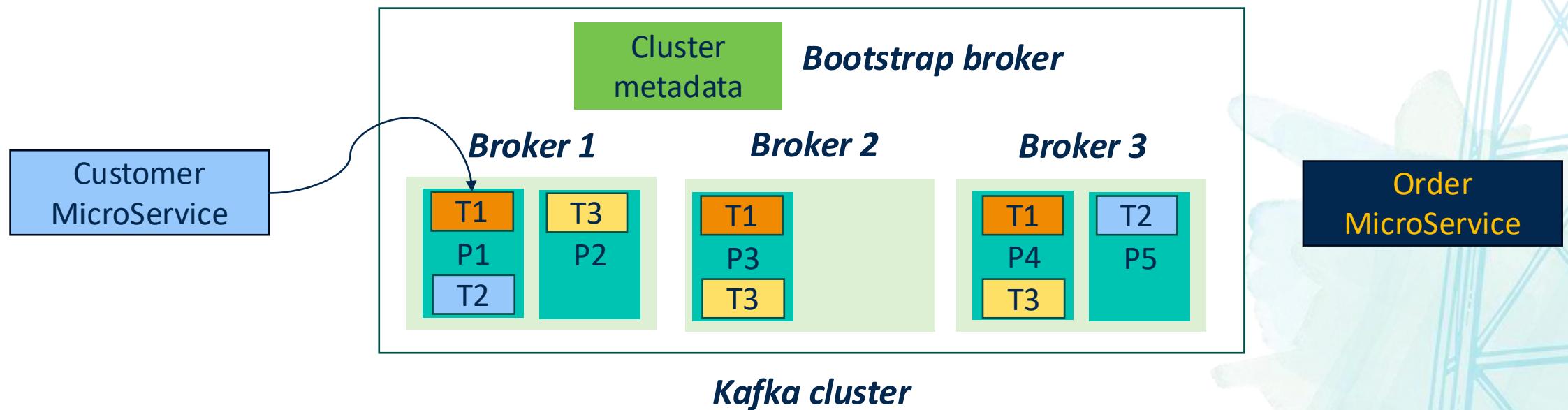
# Replication

- A single copy of data on one broker is a single point of failure
- If that broker's disk fails, the data is lost
- Replication: keep multiple copies on different brokers
- Configured per topic with the replication factor (e.g., 3)
  - Replication-factor = 3 means 3 copies for each partition
  - Cluster can tolerate ( $\text{replication-factor} - 1$ ) broker failures

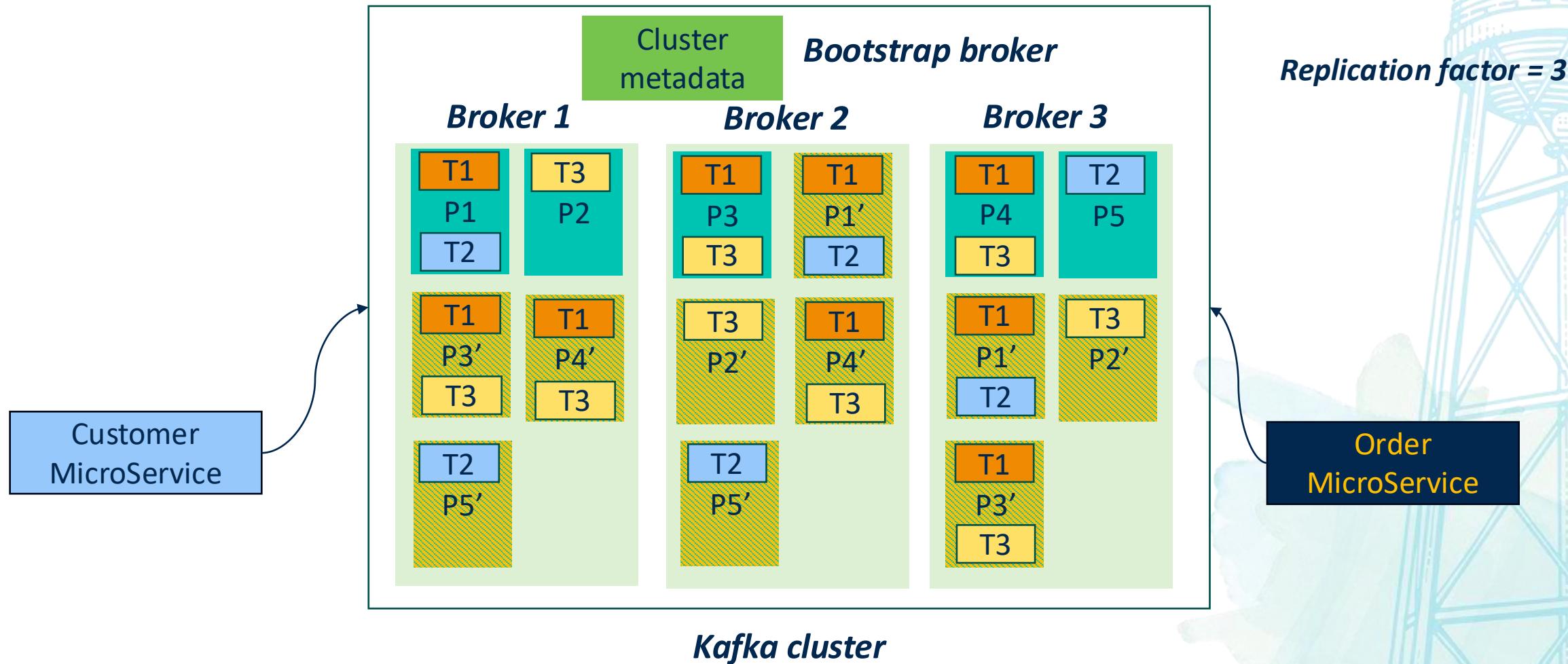
# Leader and in-sync replicas

- One replica is the leader, the others are followers
- Followers don't serve any requests; only aim to stay in-sync with leader
- Replicas which are in sync are called in-sync replicas (ISR)
- **Important:** replicas are passive; do not serve any requests (unless they become leaders due to leader failure)

# Kafka architecture (only leader partitions)



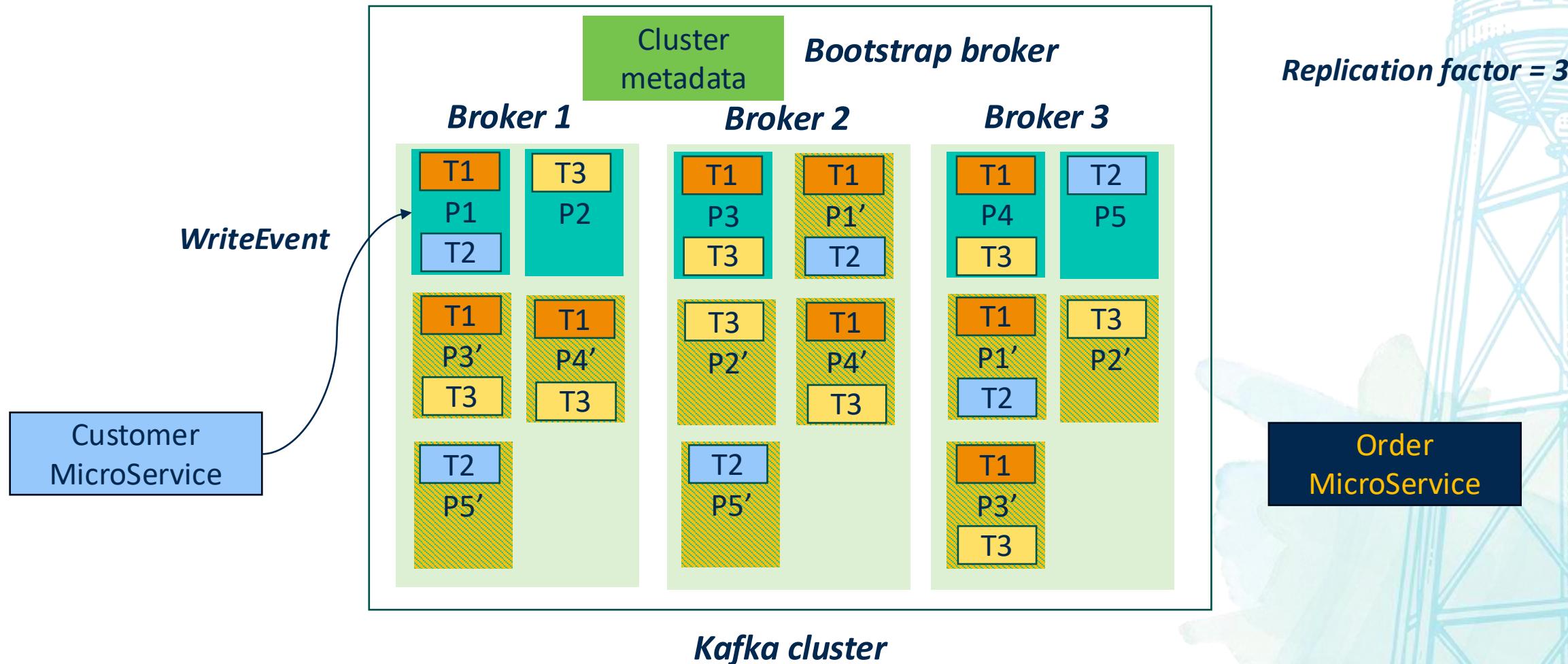
# Kafka full architecture



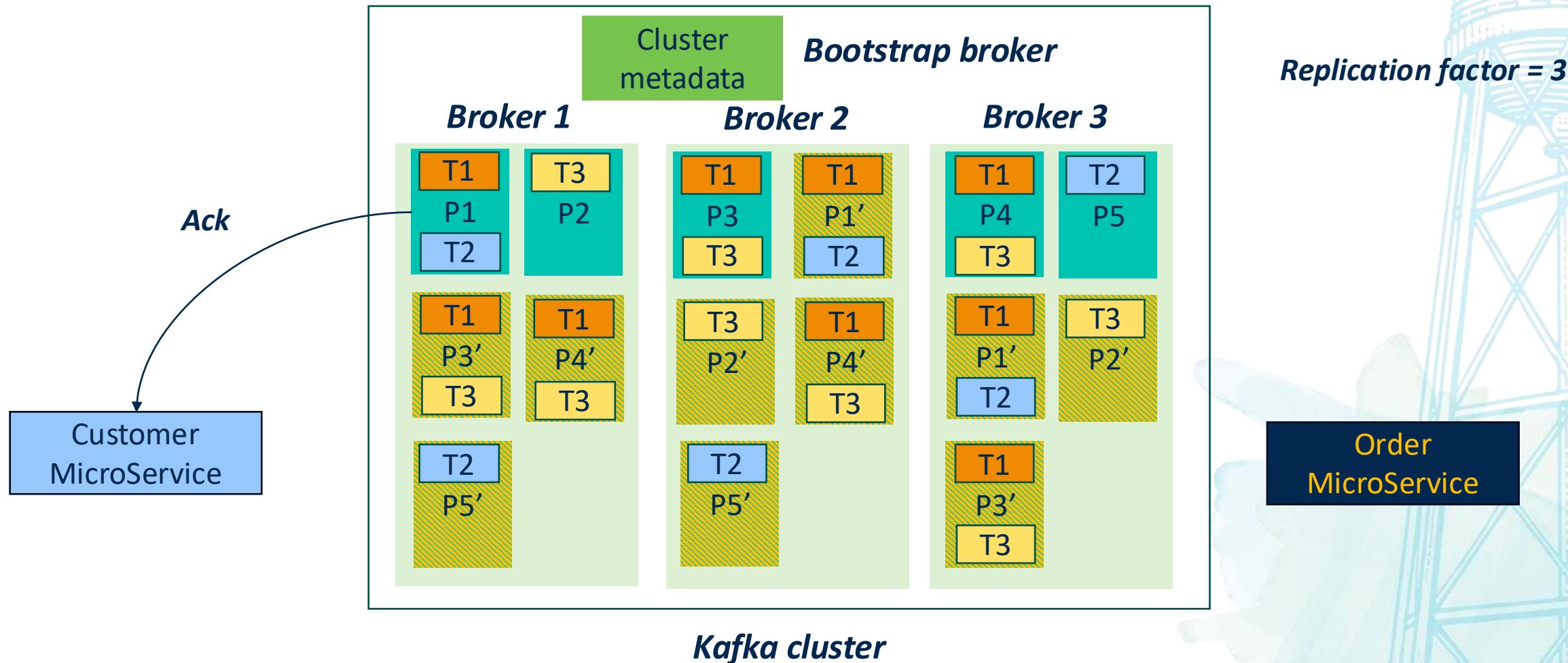
# Write path with replication

- Producer creates events and writes to partition leader
- Kafka broker (partition leader) acks the event
- Acknowledgement modes
  - `acks = 0`: Producer does not wait for an ack
  - `acks = 1`: Leader writes the record and sends ack; producer waits for ack
  - `acks = all`: Leader writes the record, waits for replicas to write the record, then sends the ack; producer waits for the ack
- Tradeoff: durability vs. latency

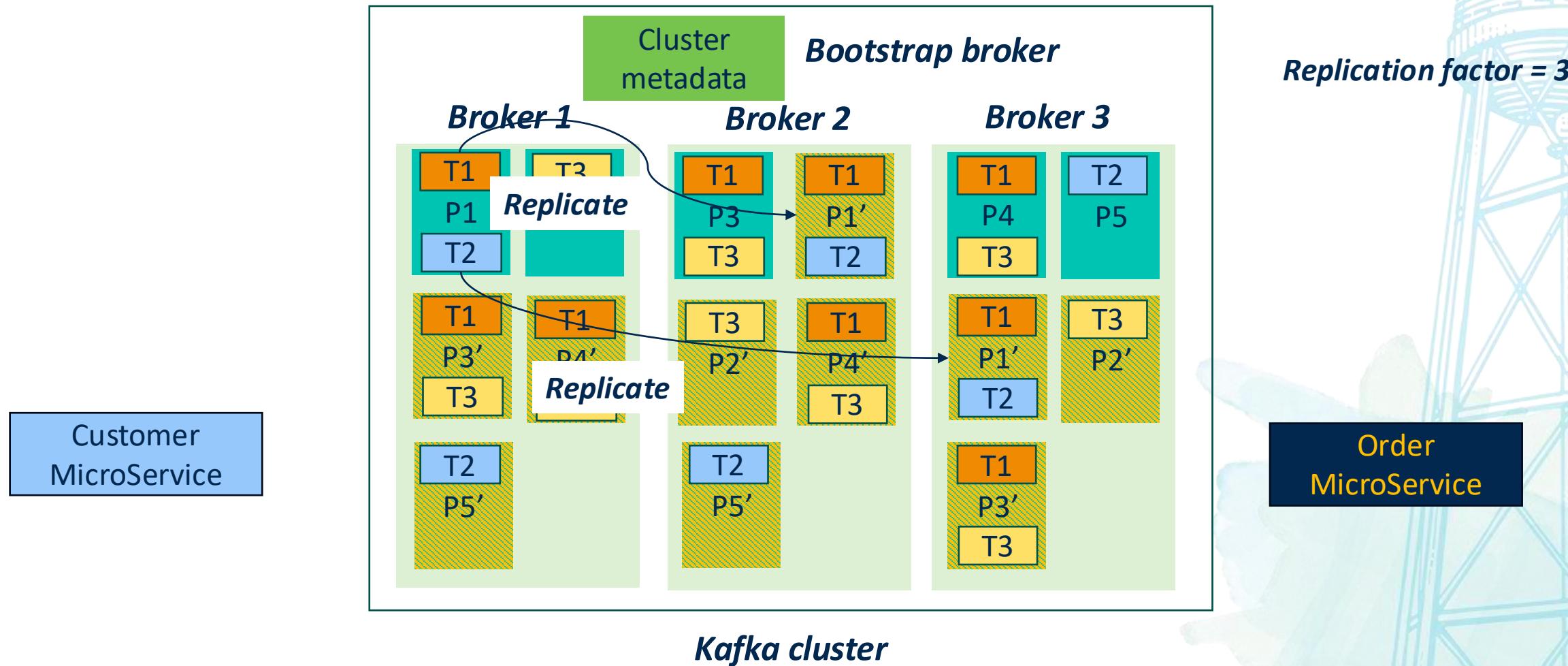
# Write path with replication for acks = 1



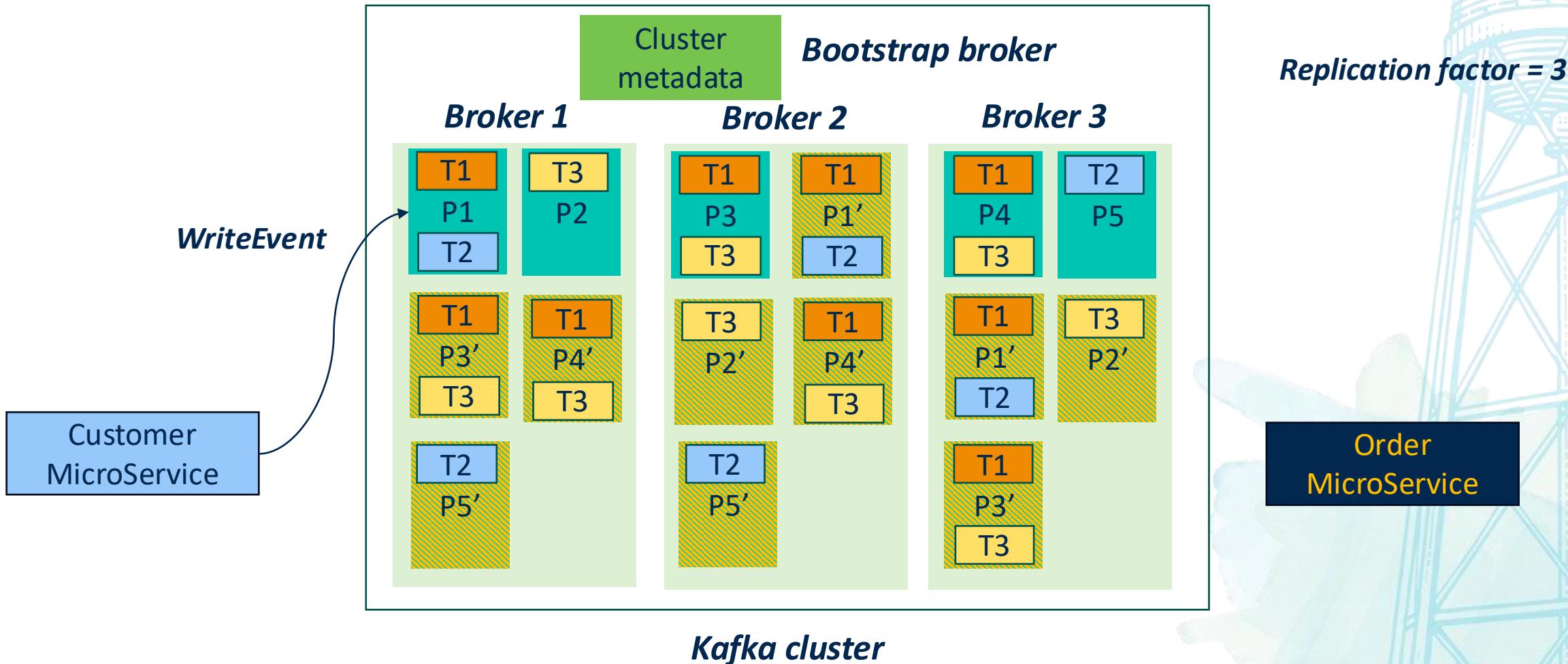
# Write path with replication for acks = 1



# Write path with replication for acks = 1

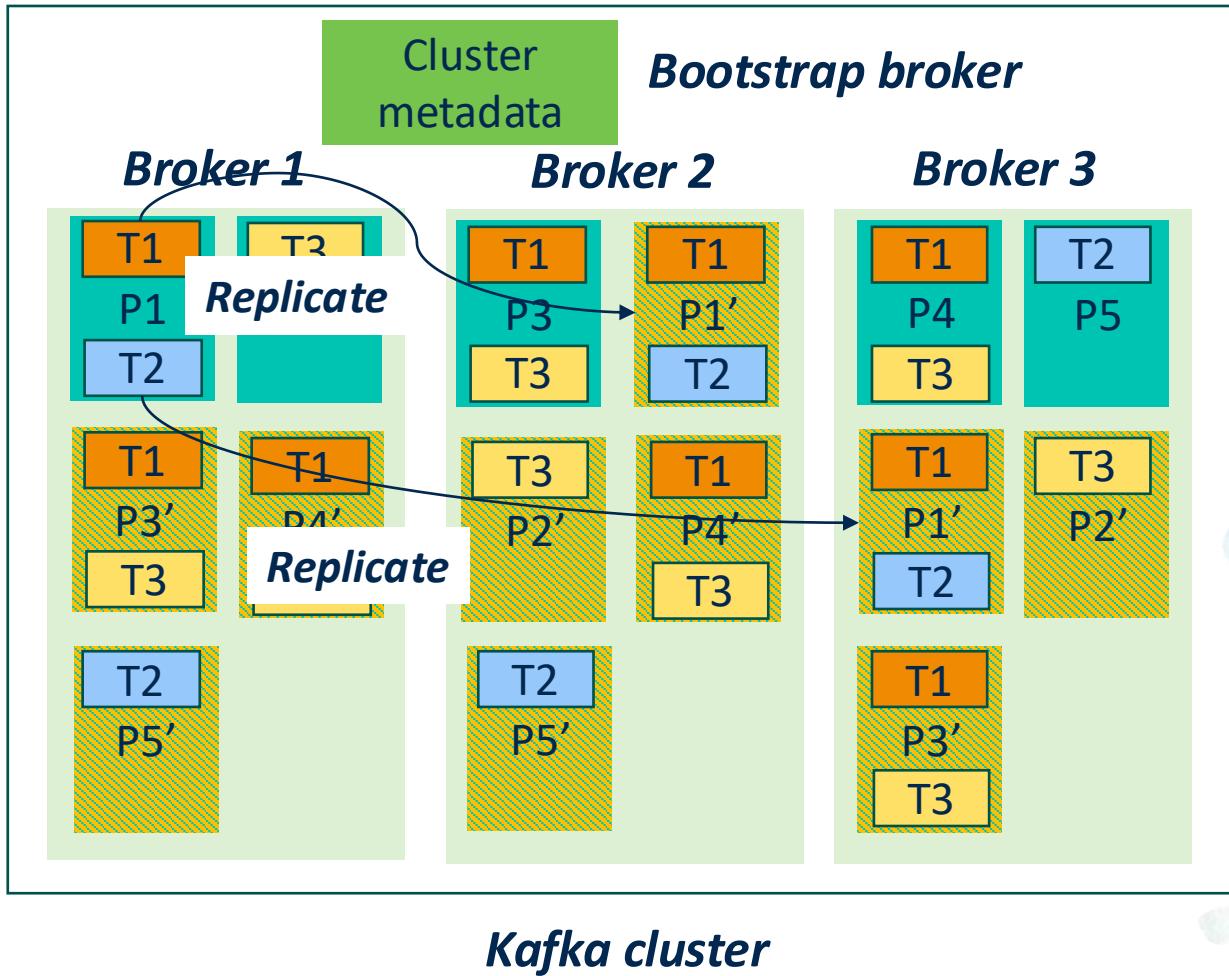


# Write path with replication for acks = all

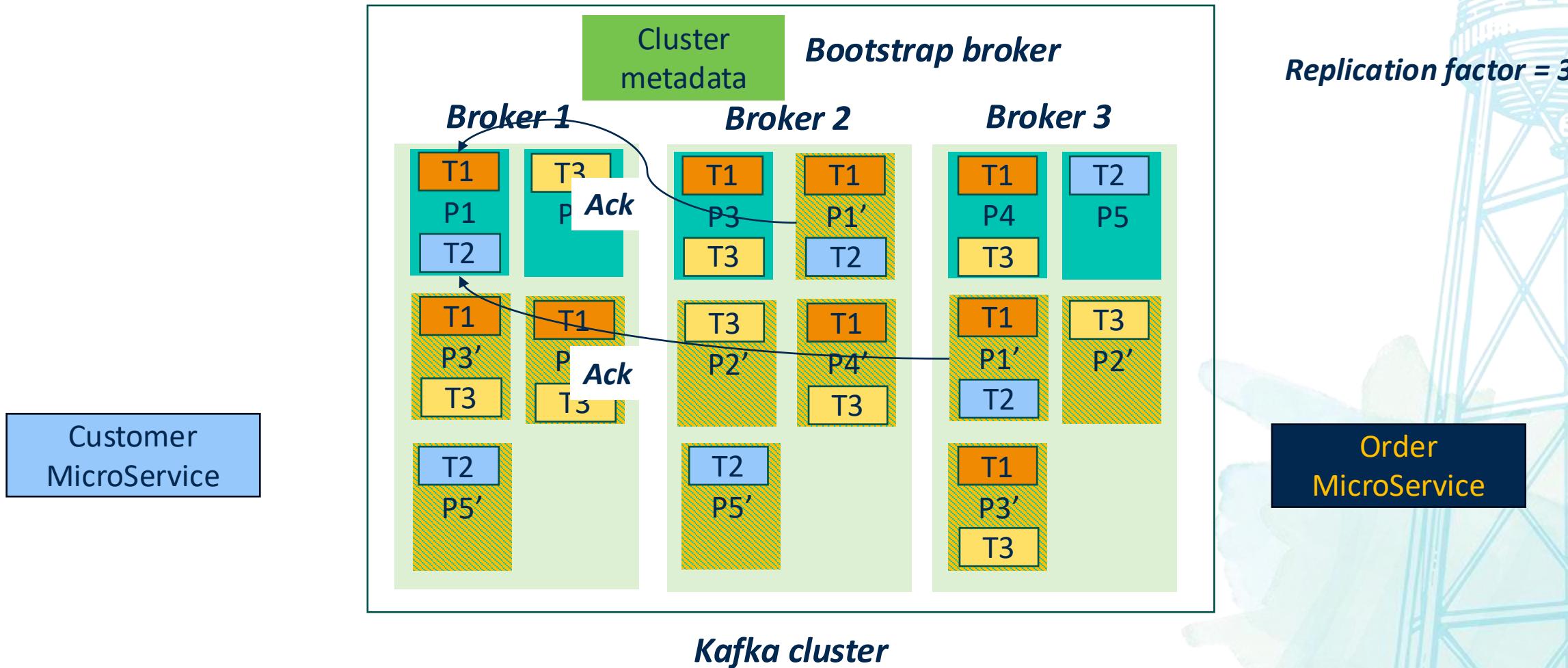


# Write path with replication for acks = all

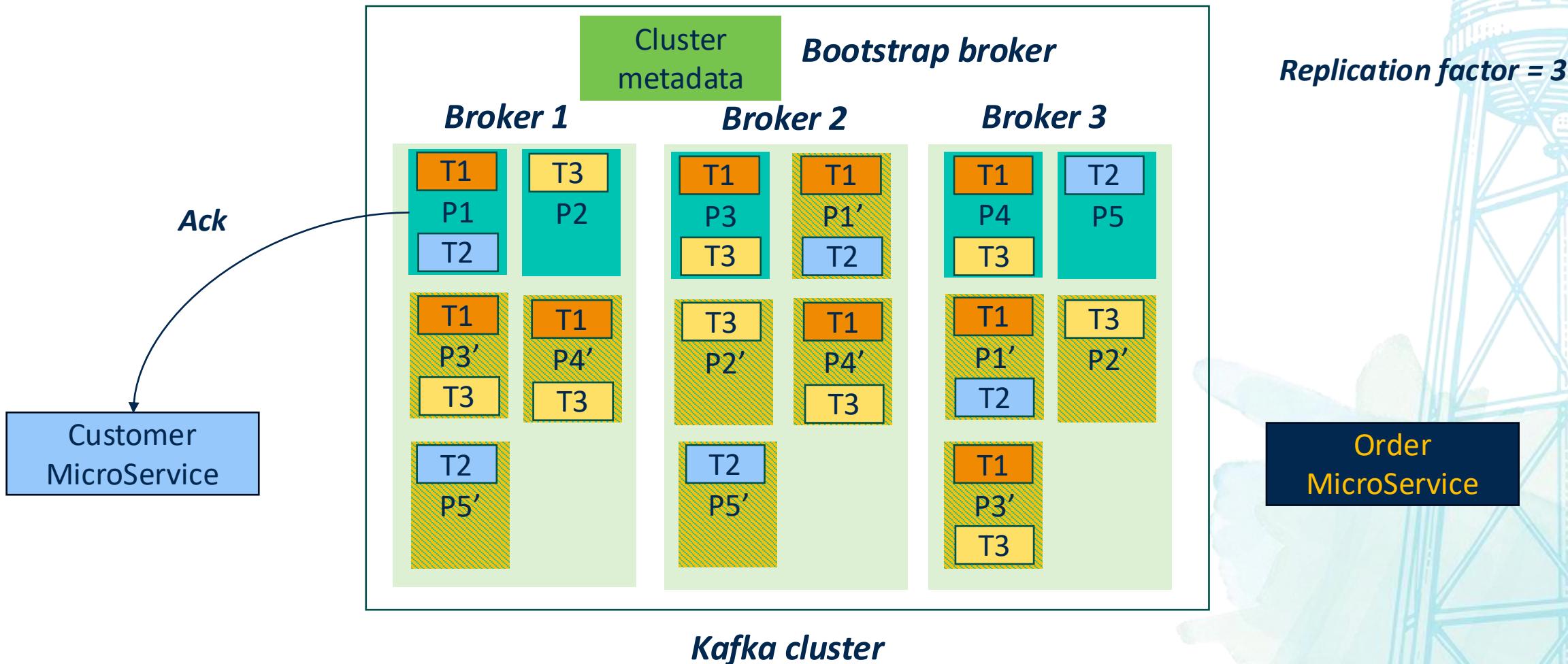
Customer  
MicroService



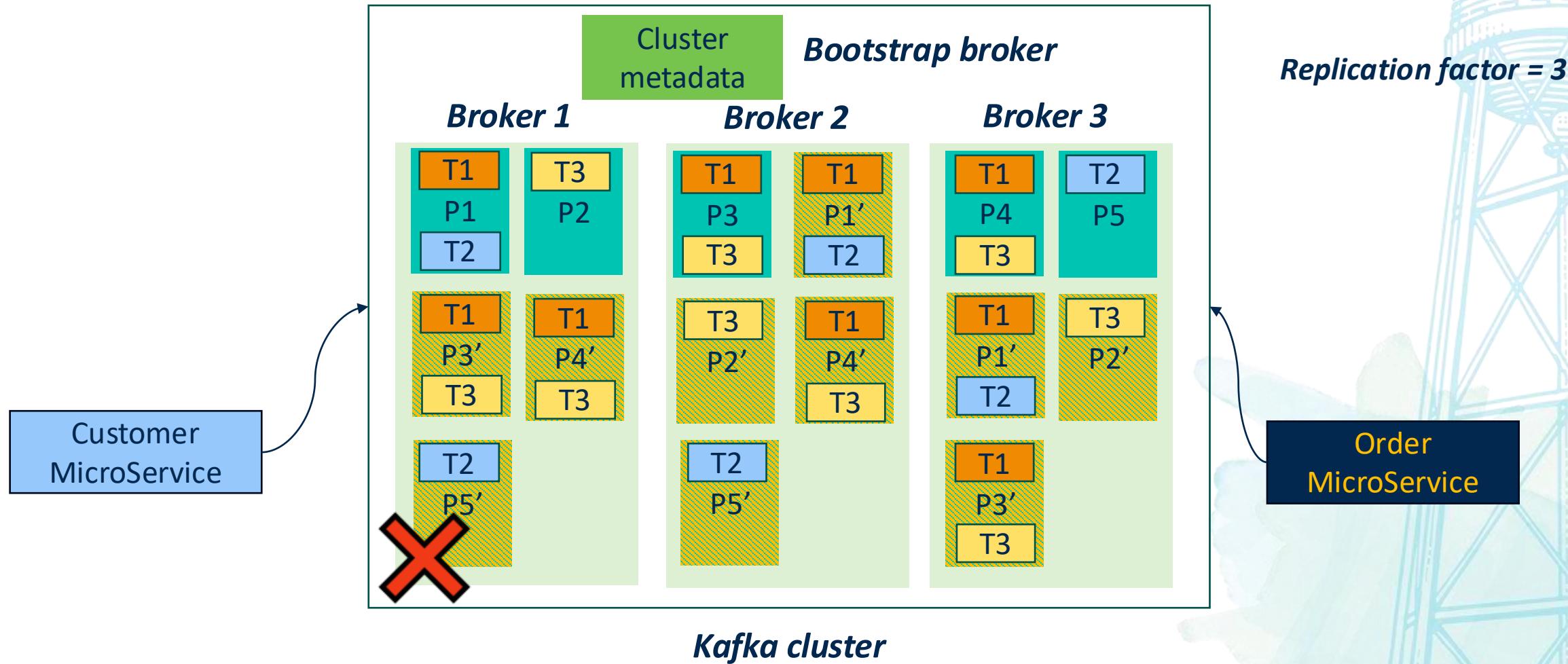
# Write path with replication for acks = all



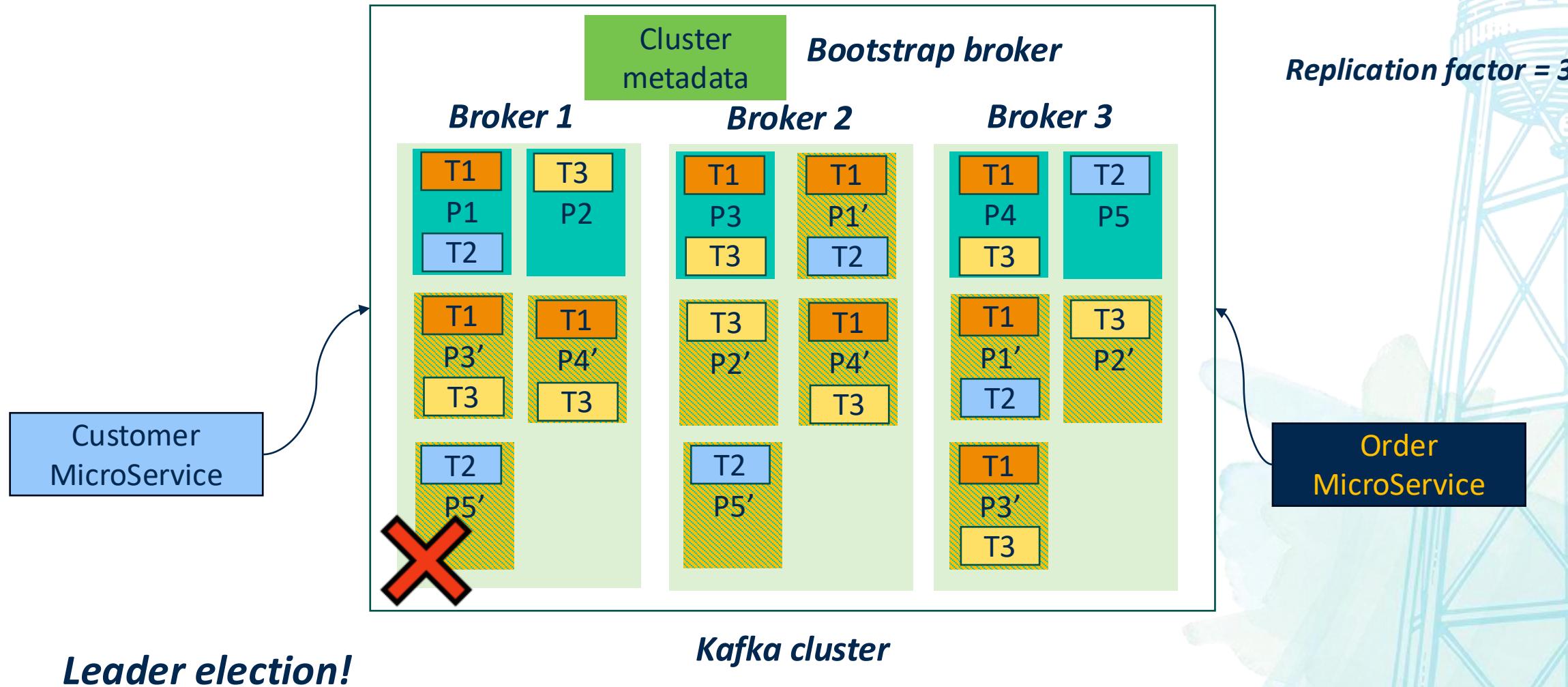
# Write path with replication for acks = all



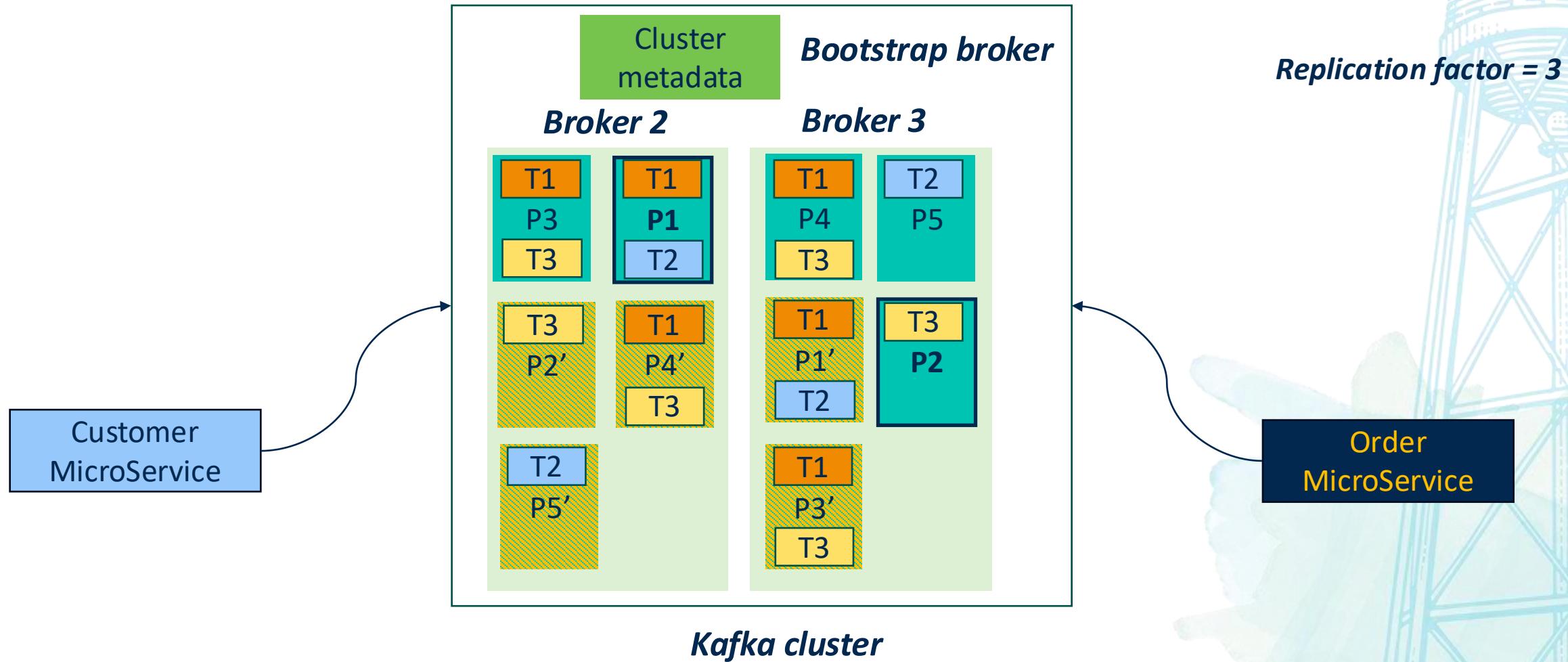
# Leader failure and election



# Leader failure and election



# Leader failure and election



# Metadata coordination

- Cluster must maintain metadata
  - Which broker is the leader of which partition
  - Which replicas are in-sync
  - Which brokers are alive
- Metadata changes continuously
  - Leader elections, broker failures, topic creation/deletion

# Metadata coordination

- All nodes must agree on the same view
- Cluster metadata requires strong consistency — Kafka handles this internally with KRaft
- Raft and Paxos-family algorithms are consensus protocols used to ensure consistency in a distributed system (beyond the scope of this class)

# Kafka summary

- Events are immutable facts; Kafka is an append-only distributed log
- Topics organize events into named feeds; partitions enable parallelism and throughput
- Replication ensures failure tolerance
- These design patterns (append-only logs, ISR, leader election) recur throughout high-throughput distributed databases
  - Worth studying beyond Kafka