CONFLUENT

Taming the cost of Kafka workloads in the cloud

Stefan Sprenger
Staff Software Engineer
ssprenger@confluent.io

Agenda

01 Introduction

02 Reducing cross-AZ traffic

03 Compression

04 Lag-based scaling of consumers

05 Scaling to zero

06 Summary

Introduction

Cloud computing is great

Unlimited compute resources

We get instant access to a seemingly unlimited capacity of compute resources (except GPUs).

Elastic scaling

We can elastically scale our applications depending on the current load. We don't need to buy large server farms upfront to cope with peak loads.

Usage-based pricing

We get billed for only those resources that we have actually used.

Challenges in cloud computing

Get charged for resources that are for free off the cloud

Cloud providers charge for resources that are for free off the cloud (at least if you stay within certain limits), e.g., network traffic.

Hibernating idle applications

Suddenly, you need to take care of hibernating idle applications, so you don't get charged for them.

Estimating cost is not trivial

We need to consider a lot of different factors when estimating the costs of running an application on a cloud platform. It's easy to get it wrong if we, for instance, fail to predict the workload pattern.

Scope of this talk

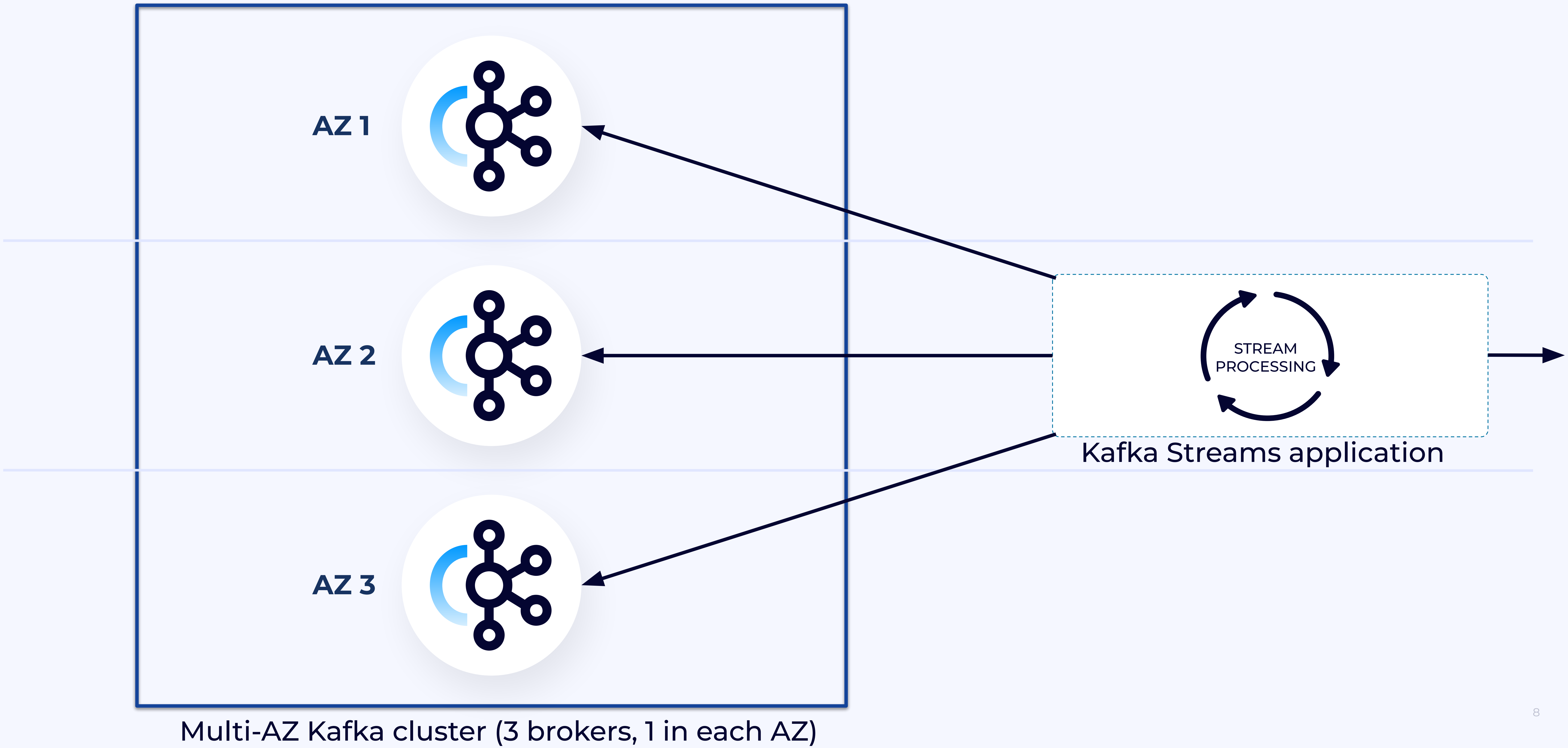
In scope

- Running Kafka workloads (e.g., Kafka Streams apps, consumers, producers) in the cloud
- Techniques to reduce and optimize costs
- Running applications on Kubernetes
- Developers

Not in scope

- Operating Kafka and other technologies, like Kafka Connect, in the cloud
- Managed vs self-managed Kafka
- Any particular cloud platforms

Use case: Kafka workload interacting with multi-AZ cluster



Main cost drivers for Kafka workloads

Network

Kafka workloads cause ingress and egress traffic when consuming from and producing to Kafka topics. Cloud providers differentiate between AZ-local and remote (cross-AZ or internet) traffic.

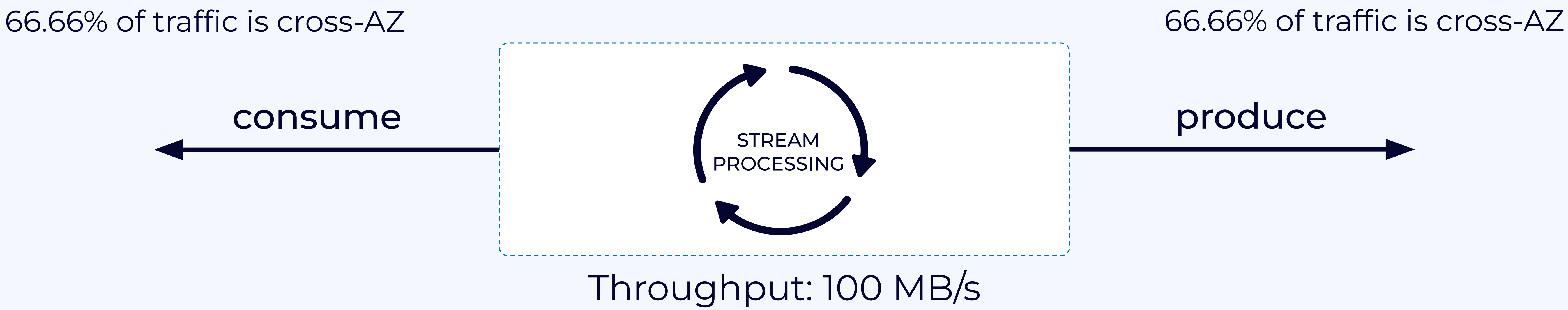
Compute

Kafka workloads require compute resources to run. When using Kubernetes, we mainly consider CPU and main memory consumption. These resources can fluctuate if applications can scale up/down elastically, making cost estimations challenging.

Storage

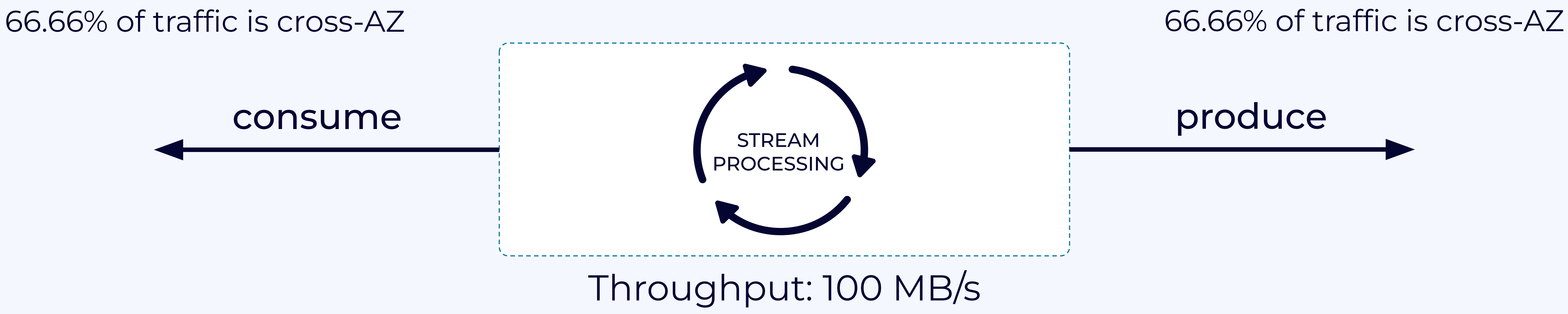
Stateful stream processing applications keep state on local disks, object storage, or other storage solutions.

Network cost can be surprisingly high



Network cost can be surprisingly high

Cost of cross-AZ traffic: $ 0.01 / GB
Cost of intra-AZ traffic: $ 0.00 / GB



Network cost can be surprisingly high

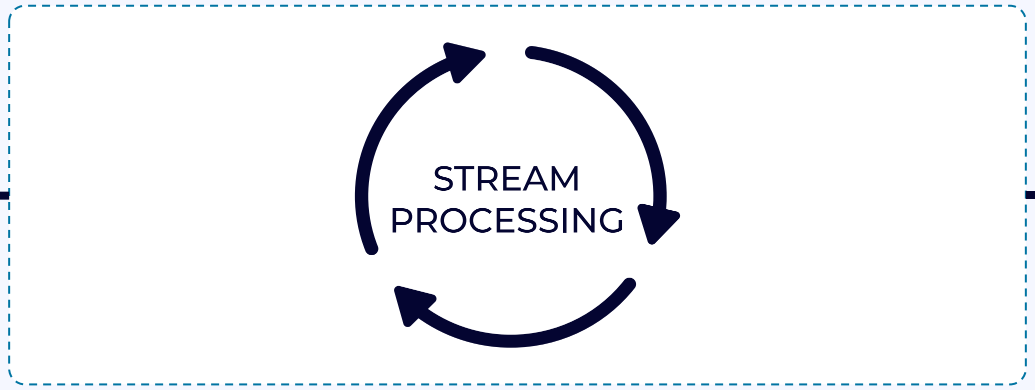
Cost of cross-AZ traffic: $ 0.01 / GB
Cost of intra-AZ traffic: $ 0.00 / GB

66.66% of traffic is cross-AZ

consume

Monthly cross-AZ traffic: 168,750 GB

Monthly cost: **1,687.50 USD**



Throughput: 100 MB/s

66.66% of traffic is cross-AZ

produce

Monthly cross-AZ traffic: 168,750 GB

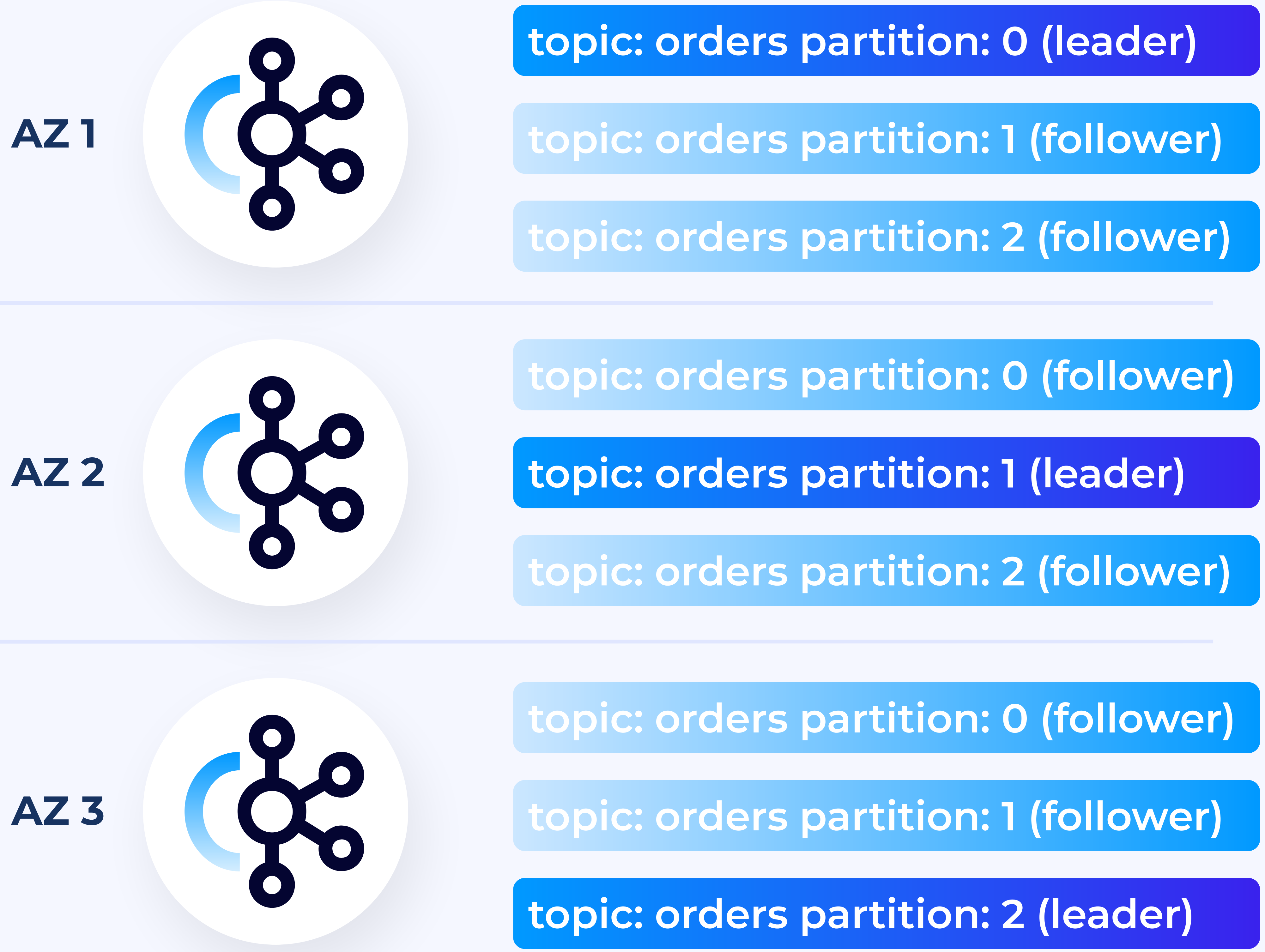
Monthly cost: **1,687.50 USD**

Monthly cross-AZ traffic (total): 337,500 GB

Monthly network cost (total): **3,375 USD**

Taming network cost: Reducing cross-AZ traffic

Partitions & Replication



Multi-AZ Kafka cluster (3 brokers, 1 in each AZ)

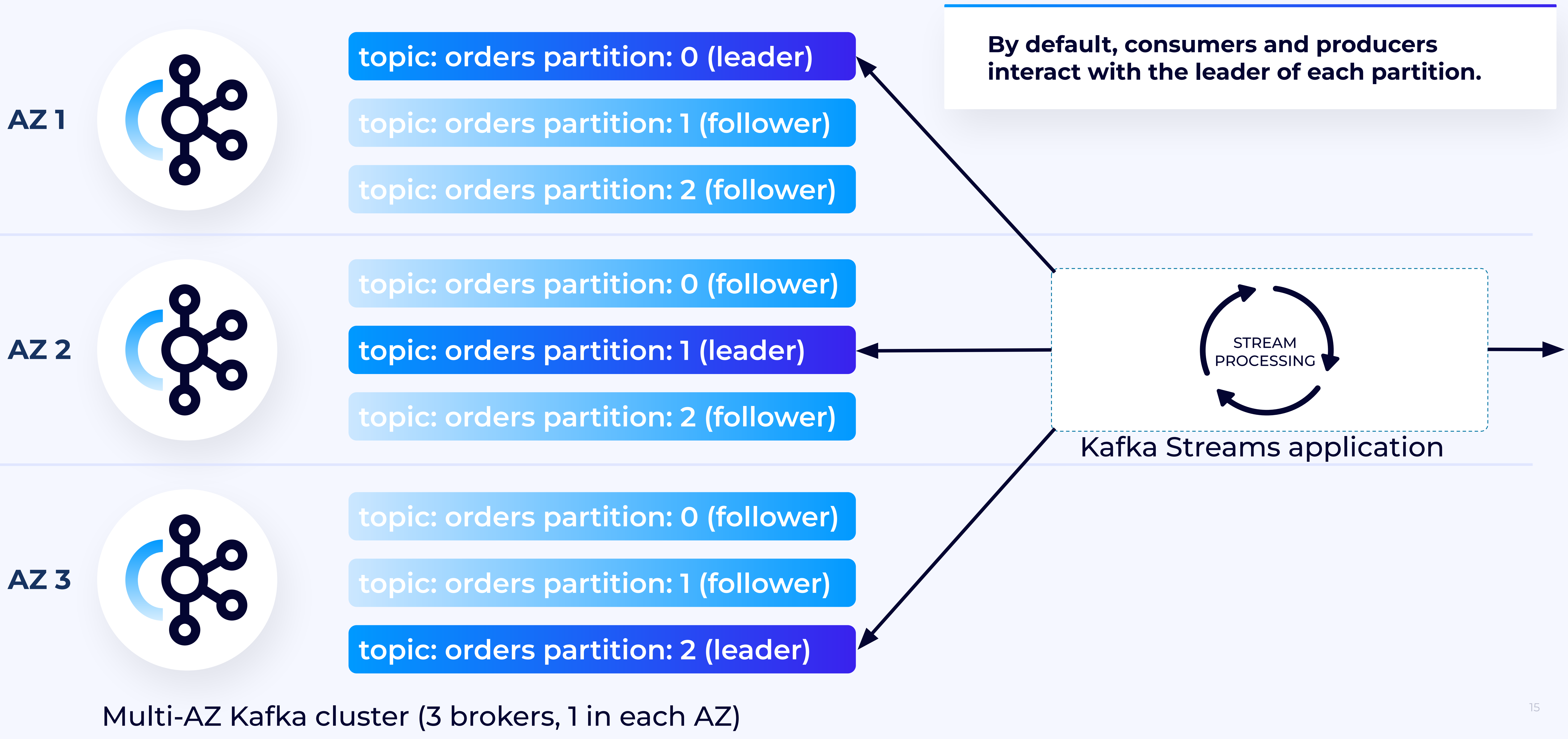
Partitions

Scale performance by parallelizing produce or consume requests.

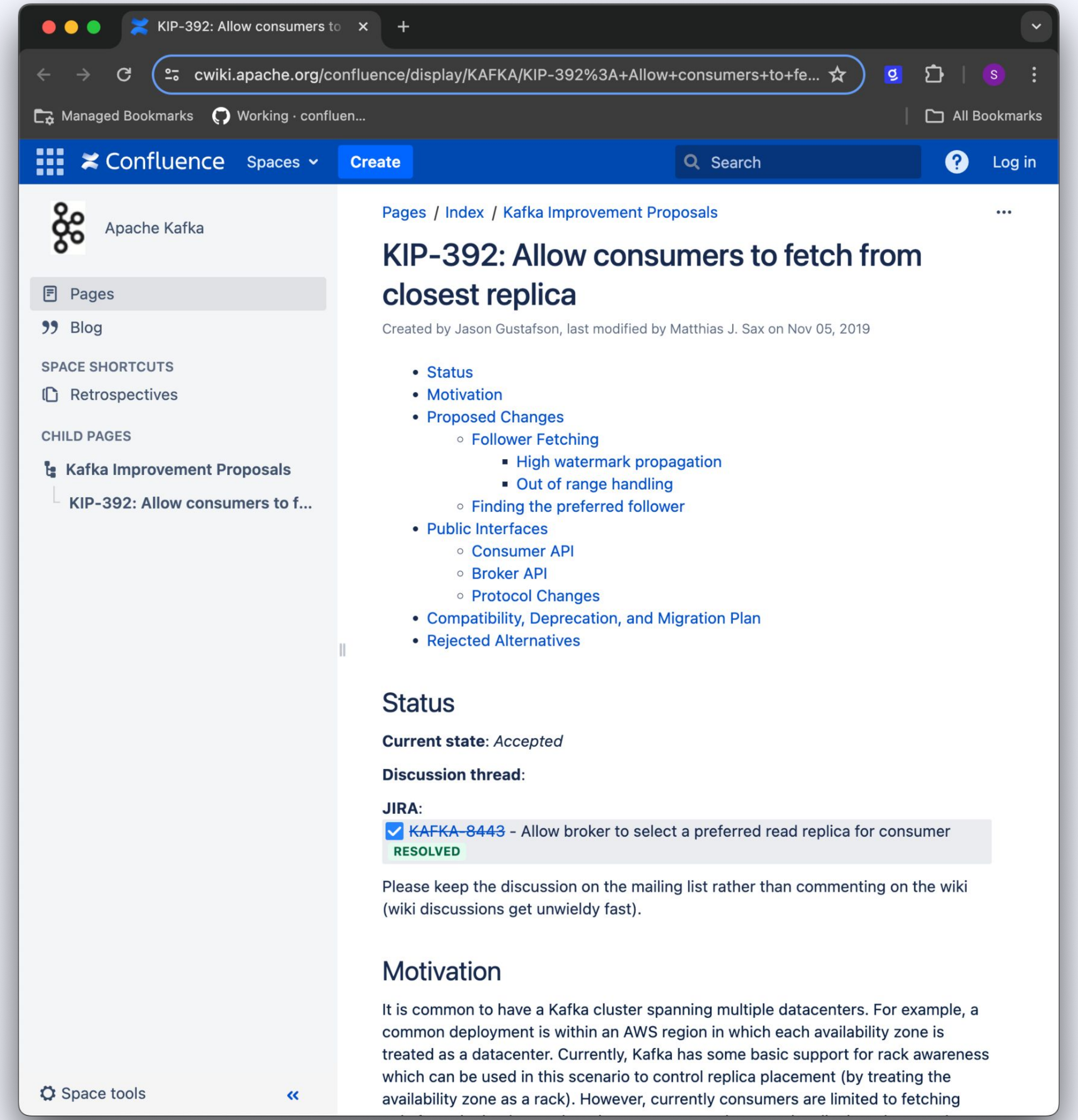
Replication

Improve availability by replicating topic partitions across brokers, potentially across different AZs. For each partition, one broker takes over the role of the leader, the remaining brokers are followers.

Kafka producers & consumers

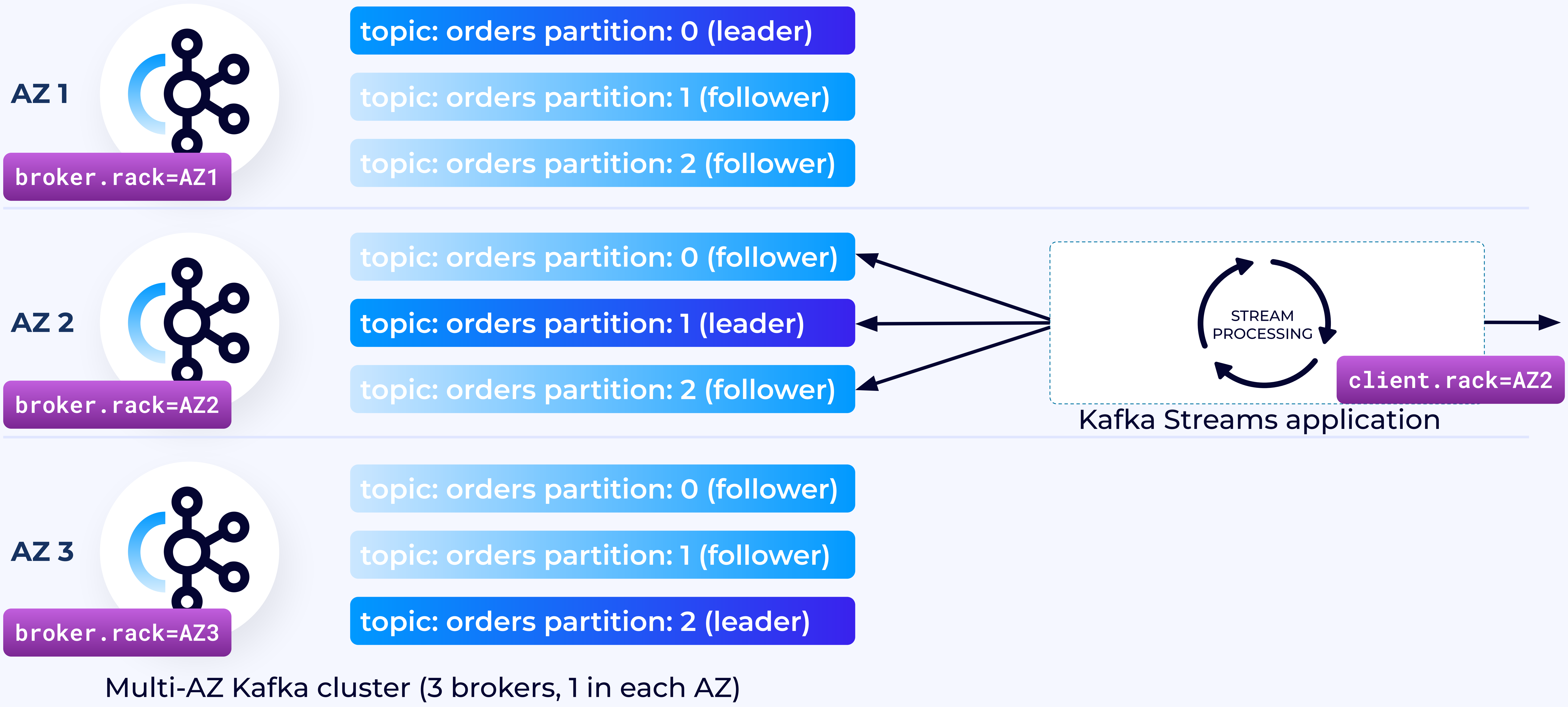


KIP-392: Allow consumers to fetch from closest replica



- Introduced in Apache Kafka 2.4
- Extends existing rack-aware placement of partition replicas
- Leverage locality and fetch from local replica
- Broker config: `broker.rack`
- Consumer config: `client.rack`

Follower fetching



Rack-aware replica selector

```java
class RackAwareReplicaSelector implements ReplicaSelector {

  Optional<ReplicaView> select(TopicPartition topicPartition,
                               ClientMetadata metadata,
                               PartitionView partitionView) {

    // if `client.rack` is null
    //   - return leader of partition
    // if `client.rack` is not null
    //   - iterate through the online replicas
    //   - if one or more exists with matching rackId, choose the most caught-up replica from among them
    //   - otherwise return the current leader (from remote rack)
  }
}
```

https://github.com/apache/kafka/blob/trunk/clients/src/main/java/org/apache/kafka/common/replica/RackAwareReplicaSelector.java

Impact of follower fetching on network cost

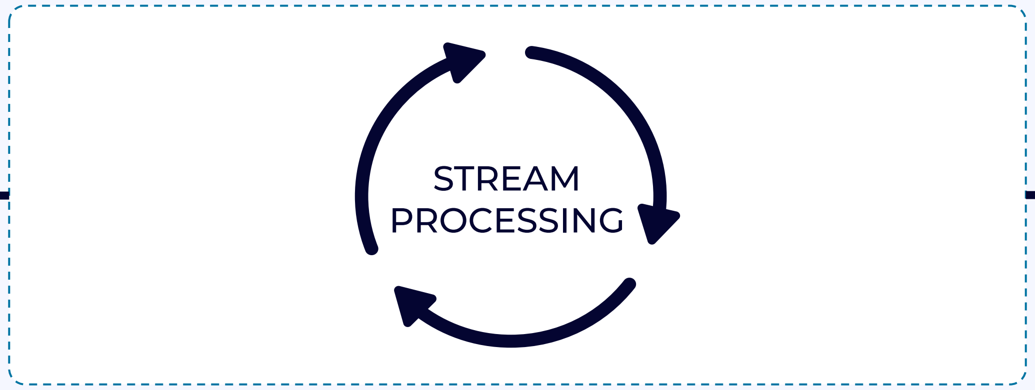
Cost of cross-AZ traffic: $ 0.01 / GB
Coss of intra-AZ traffic: $ 0.00 / GB

✅ **0% of traffic is cross-AZ**

consume

Monthly cross-AZ traffic: 0 GB ~~168,750 GB~~

Monthly cost: **0 USD** ~~1,687.50 USD~~



Throughput: 100 MB/s

66.66% of traffic is cross-AZ

produce

Monthly cross-AZ traffic: 168,750 GB

Monthly cost: **1,687.50 USD**

Monthly cross-AZ traffic (total): 168,750 GB ~~337,500 GB~~

Monthly cost (total): **1,687.50 USD** ~~3,375 USD~~

Follower fetching: Pros & Cons

Advantages

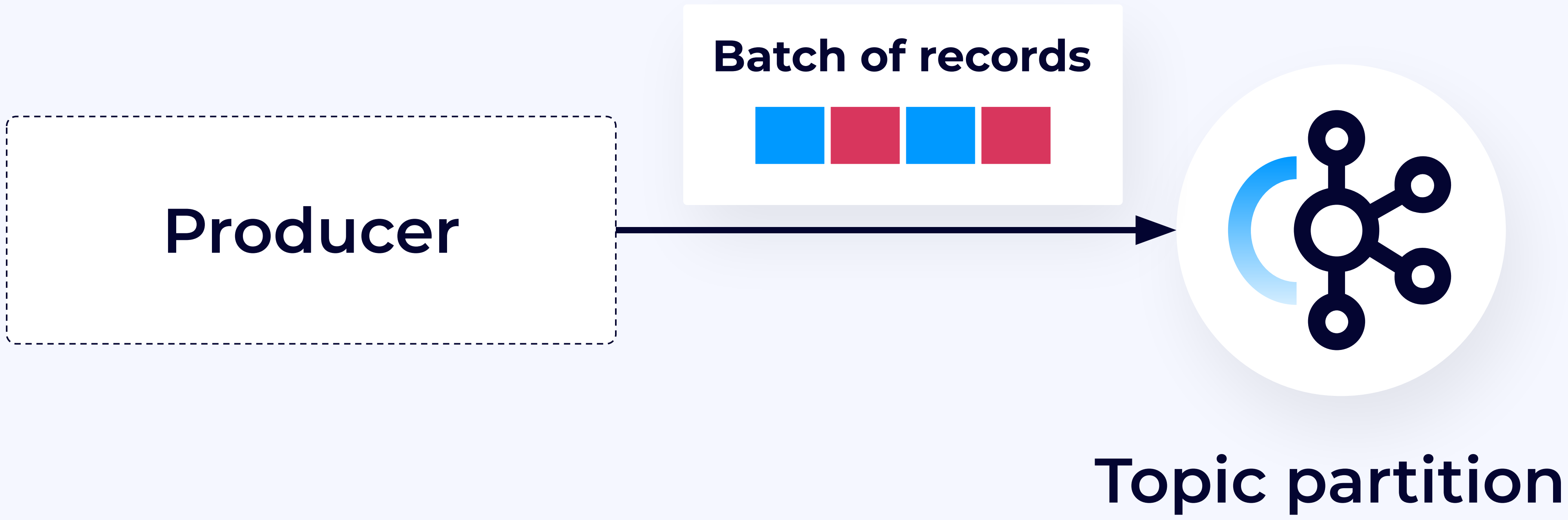
- Minimizes costly cross-AZ traffic for consumers
- Might reduce read latency because clients read from local AZ

Downsides

- No reduction of cross-AZ traffic for producers
- Might increase read latency for consumers in AZ with followers that lag behind leaders

Taming network cost: Compression of produce and consume requests

Producers send records in batches to reduce I/O ops



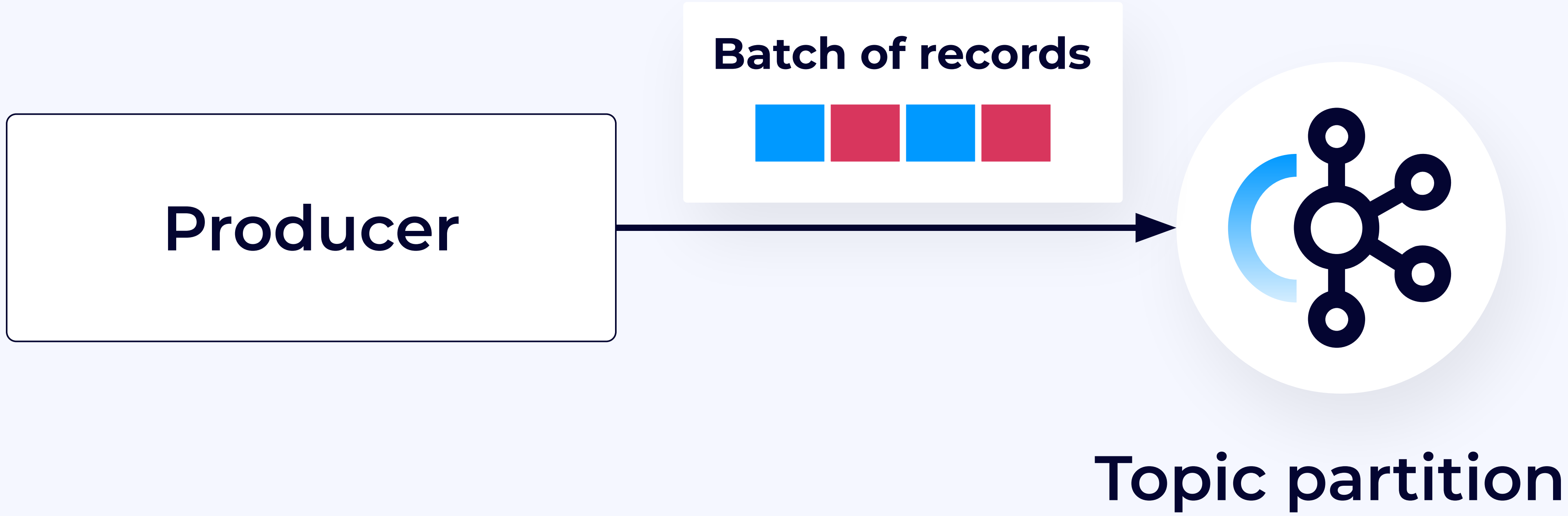
batch.size

Upper bound of batch size in bytes. Small batch size: low memory needs and low latency. Large batch size: High throughput.

linger.ms

Maximum amount of time to wait to fill up a batch of records.

Producers can compress batches of records



Producer config: compression.type

Impacts produce requests.

Topic config: compression type

Impacts storage and consume requests.

Producer config: compression.type

- Defines the compression algorithm used by the producer client
- Available options: `none`, `gzip`, `snappy`, `lz4`, `zstd` (default: `none`)
  - If set to `none`: Does not compress batch of records before sending it to the partition leader
  - Otherwise: Compress records using configured algorithm before sending them to the partition leader
- Compression tends to work best for larger batches with repeating patterns (i.e., no random data)
- Benchmark algorithms to find the one that works best for your data (sane starting point: `lz4`)
- Typical compression rates: 2-3X

Topic config: compression.type

- Defines the compression algorithm used by the brokers (and consumers)
- Available options: `uncompressed`, `producer`, `gzip`, `snappy`, `lz4`, `zstd` (default: `producer`)
  - If set to `producer`: Retain compression used by producer
  - If set to `uncompressed`: Uncompress data before storing them
  - Otherwise: Potentially re-compress data before storing them
- In most cases, just stick to default option `producer` and delegate compression to producer

Impact of compression (3x ratio) on network cost

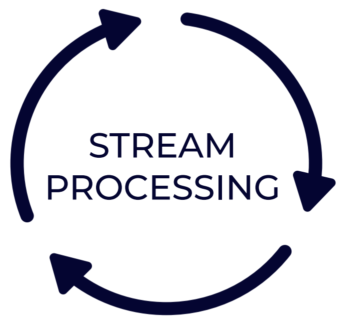
Cost of cross-AZ traffic: $ 0.01 / GB
Coss of intra-AZ traffic: $ 0.00 / GB

no cross-AZ traffic

consume

Monthly cross-AZ traffic: 0 GB

Monthly cost: **0 USD**



Throughput: 100 MB/s

⅔ of traffic is cross-AZ

produce

Monthly cross-AZ traffic: 56,250 GB ~~168,750 GB~~

Monthly cost: **562.50 USD** ~~1,687.50 USD~~

Monthly cross-AZ traffic (total): 56,250 GB ~~168,750 GB~~

Monthly cost (total): **562.50 USD** ~~1,687.50 USD~~

Compression: Pros & Cons

Advantages

- Reduces network traffic (for both producers and consumers)
- Reduces storage requirements
- Improves throughput

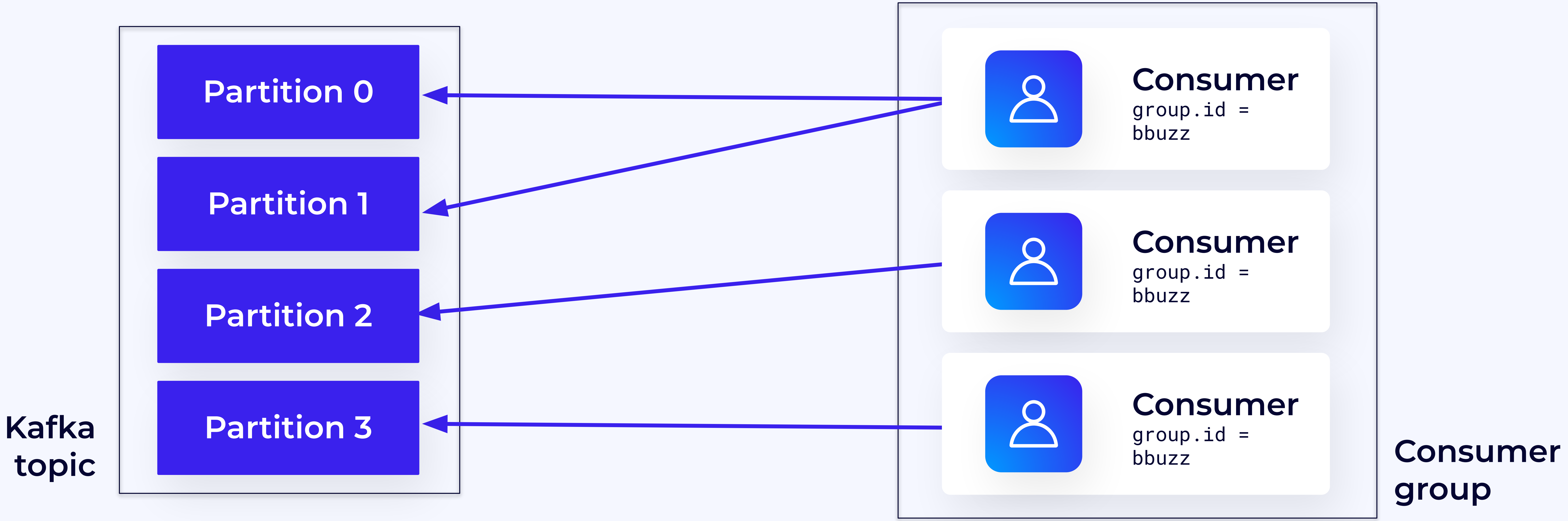
Downsides

- Increases CPU consumption
- Might increase end-to-end latency
- Might not work well on encrypted data

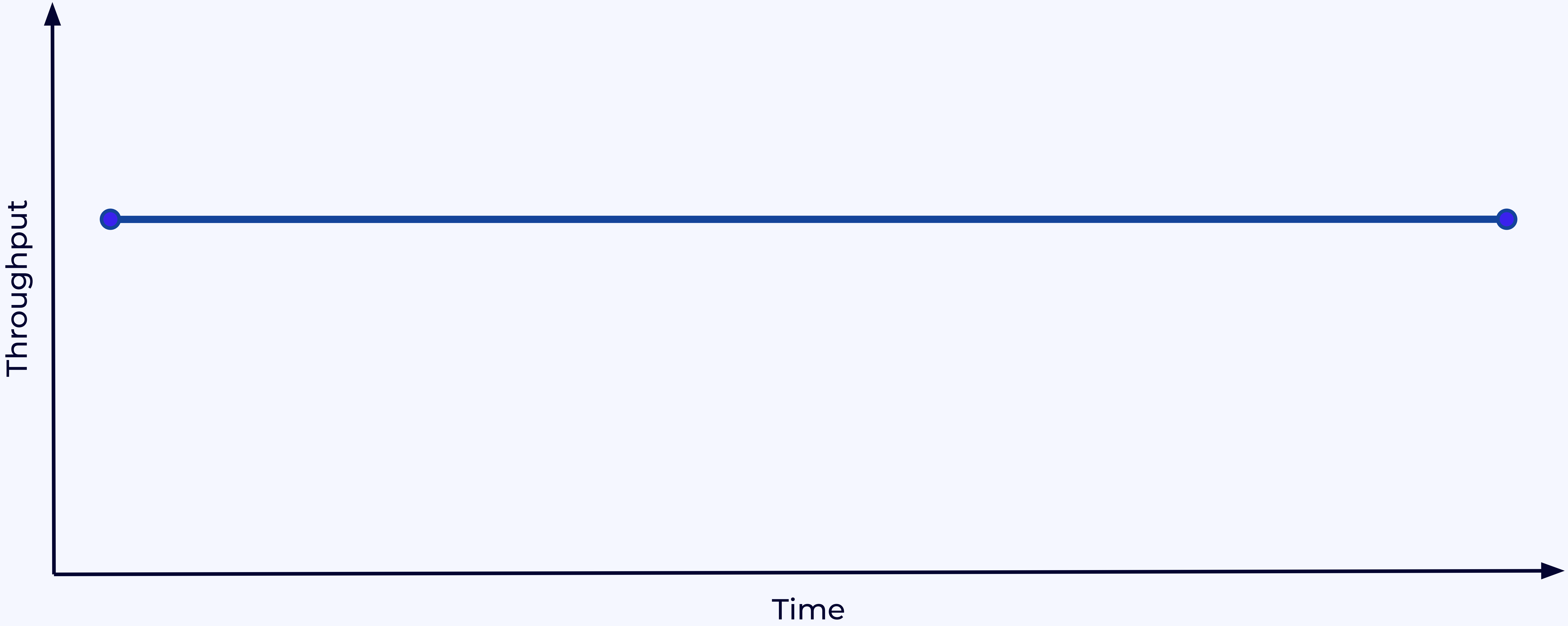
Taming compute cost:
Lag-based scaling of consumers

Scaling consumer groups

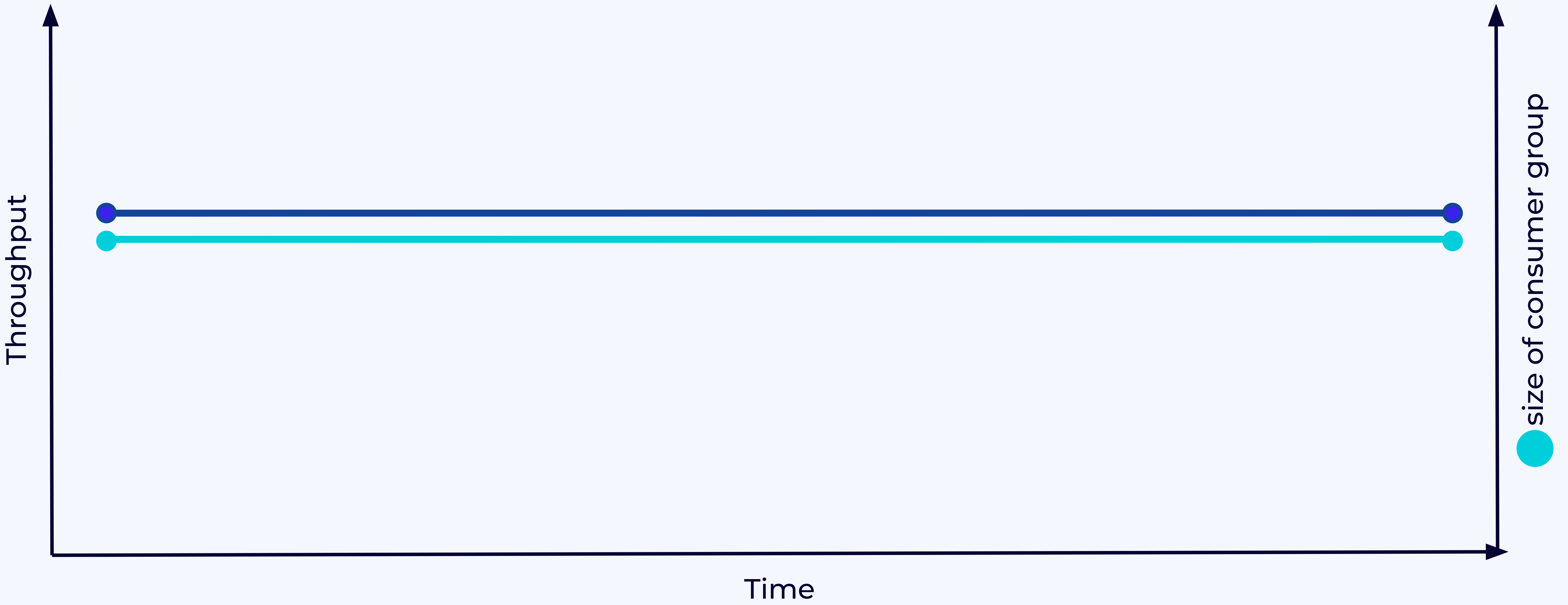
- You can parallelize consumers by launching multiple instances of the same application (`group.id`)
- Kafka automatically balances workload between applications with the same `group.id`, also called **consumer group**
- One consumer can process one or multiple partitions of the same topic
- One partition can be processed by only one consumer of the same `group.id`
- Number of partitions sets the maximum degree of parallelism of Kafka consumers



Consumer workload pattern in a perfect world

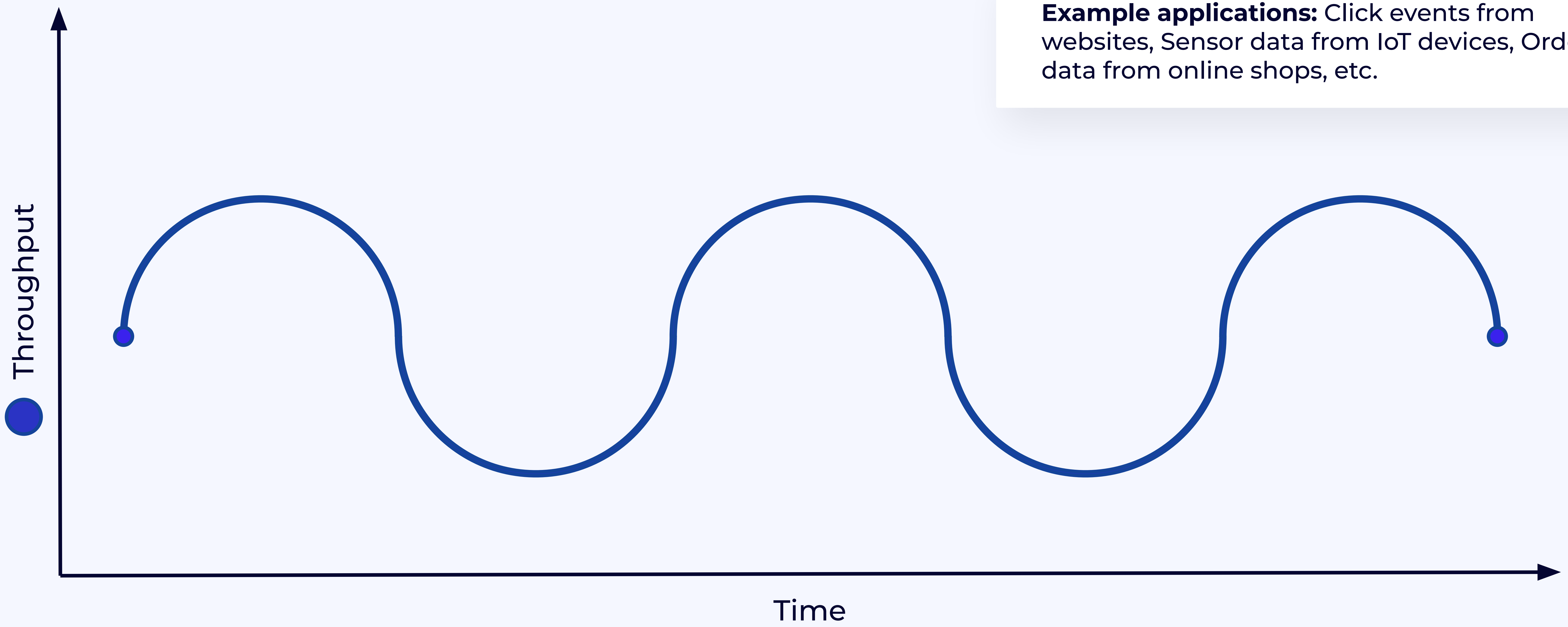


Stable throughput: Stable consumer group size

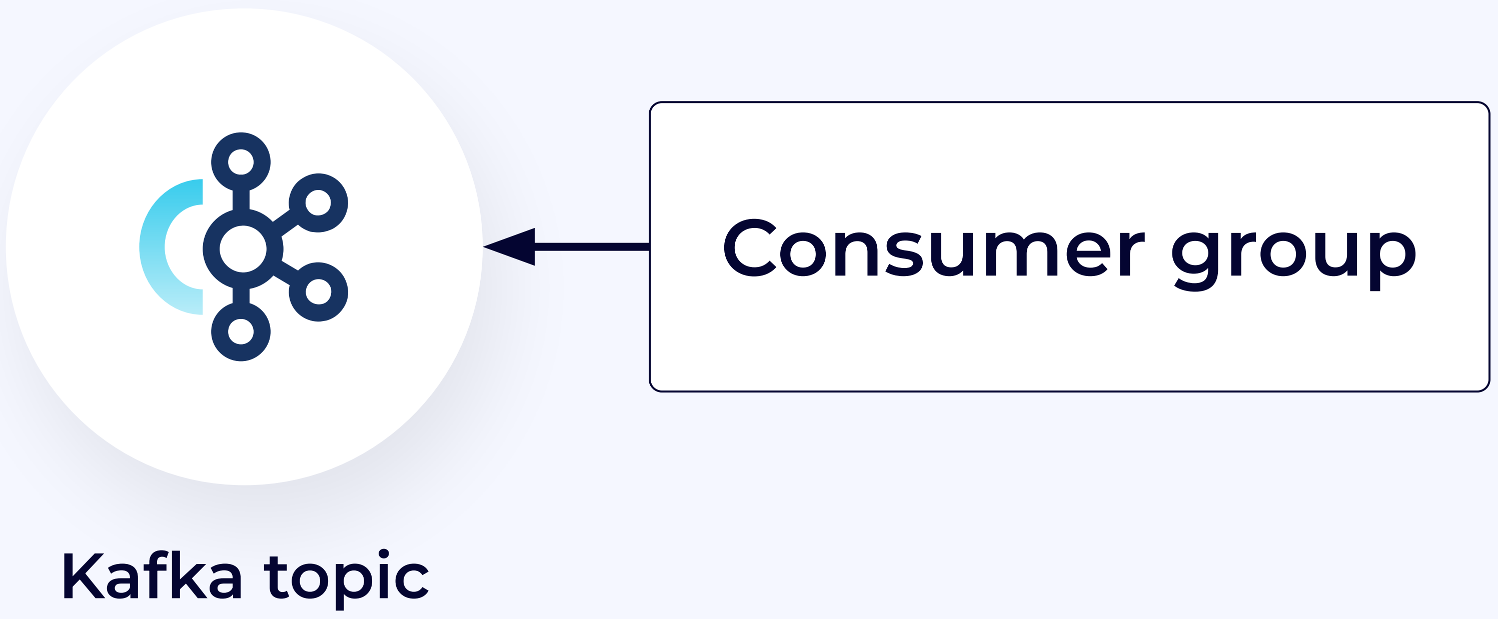


More realistic workload pattern of consumers

Example applications: Click events from websites, Sensor data from IoT devices, Order data from online shops, etc.



Consumer lags



- Equals the number of records in a partition that have not yet been processed by the consumer group
- High consumer lags lead to an increase in end-to-end processing latency
- A consumer lag close to 0 is preferable (small fluctuations are normal)
- If consumer lags keep increasing, it's time to scale up your application by increasing the size of the consumer group (unless the application features any bug causing the high consumer lag)

KEDA: Scale Kafka consumers depending on current lag

- KEDA can scale Deployments, StatefulSets, and Jobs based on custom metrics, like consumer lags
- Integrates with the Horizontal Pod Autoscaler API
- If the consumer lag of the application increases, KEDA can feed this to the Horizontal Pod Autoscaler and trigger a scale-up of the application
- If the application has catched up, the HPA can scale down the application

KEDA: Scaling a Deployment based on consumer lags

Point KEDA to Kafka topic and consumer group

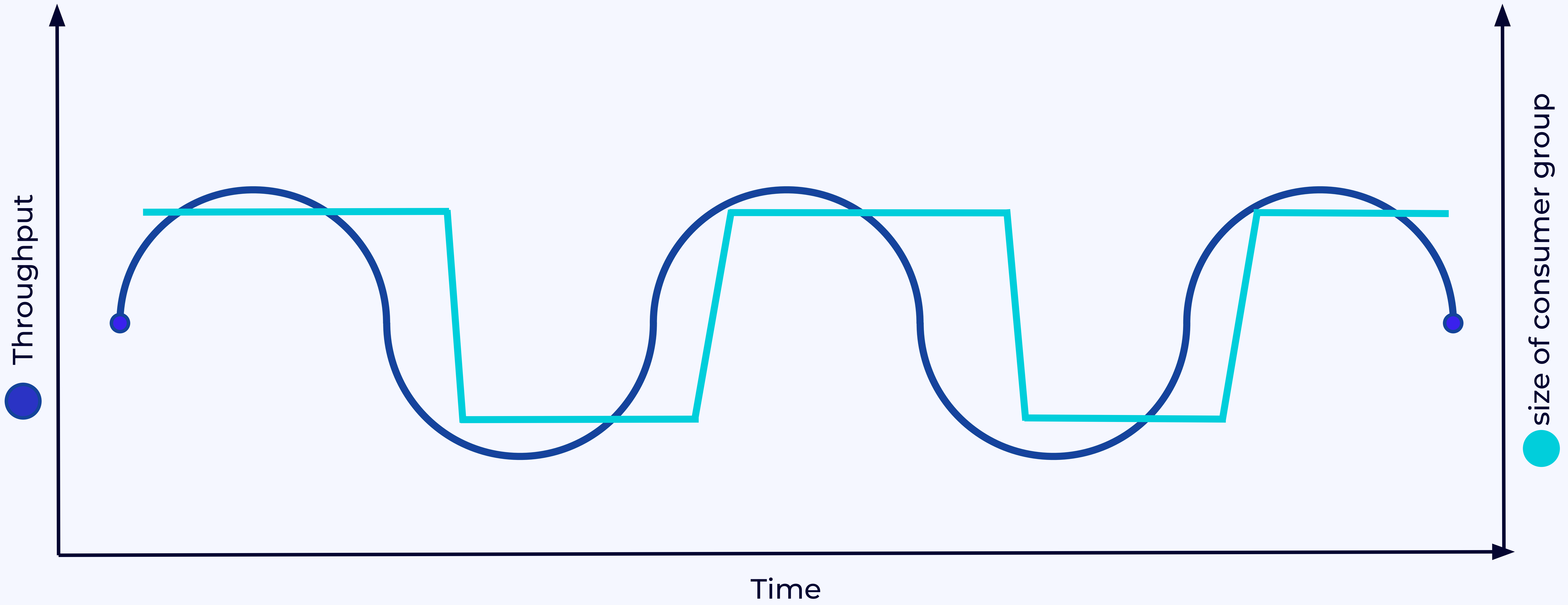
```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: kafka-streams-app-scaledobject
  namespace: default
spec:
  scaleTargetRef:
    name: kafka-streams-app
  pollingInterval: 5
  triggers:
  - type: kafka
    metadata:
      bootstrapServers: localhost:9092
      consumerGroup: my-group
      topic: input-topic
      lagThreshold: "50"
```

Scale Deployment with name `kafka-streams-app`

Check metric every 5 seconds

Average target value to trigger scaling

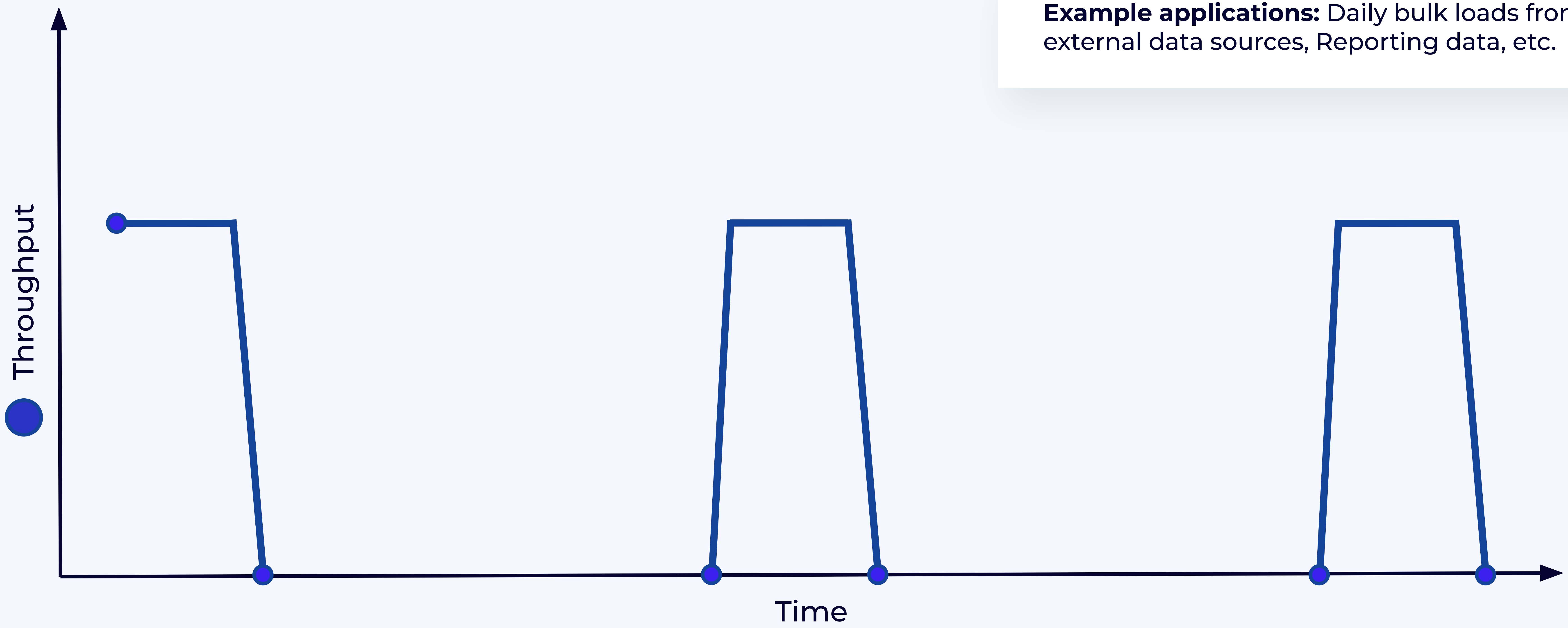
Elastic scaling of consumer groups



Taming compute cost: Scaling consumers to zero

The worst: Periodic batch inserts

Example applications: Daily bulk loads from external data sources, Reporting data, etc.



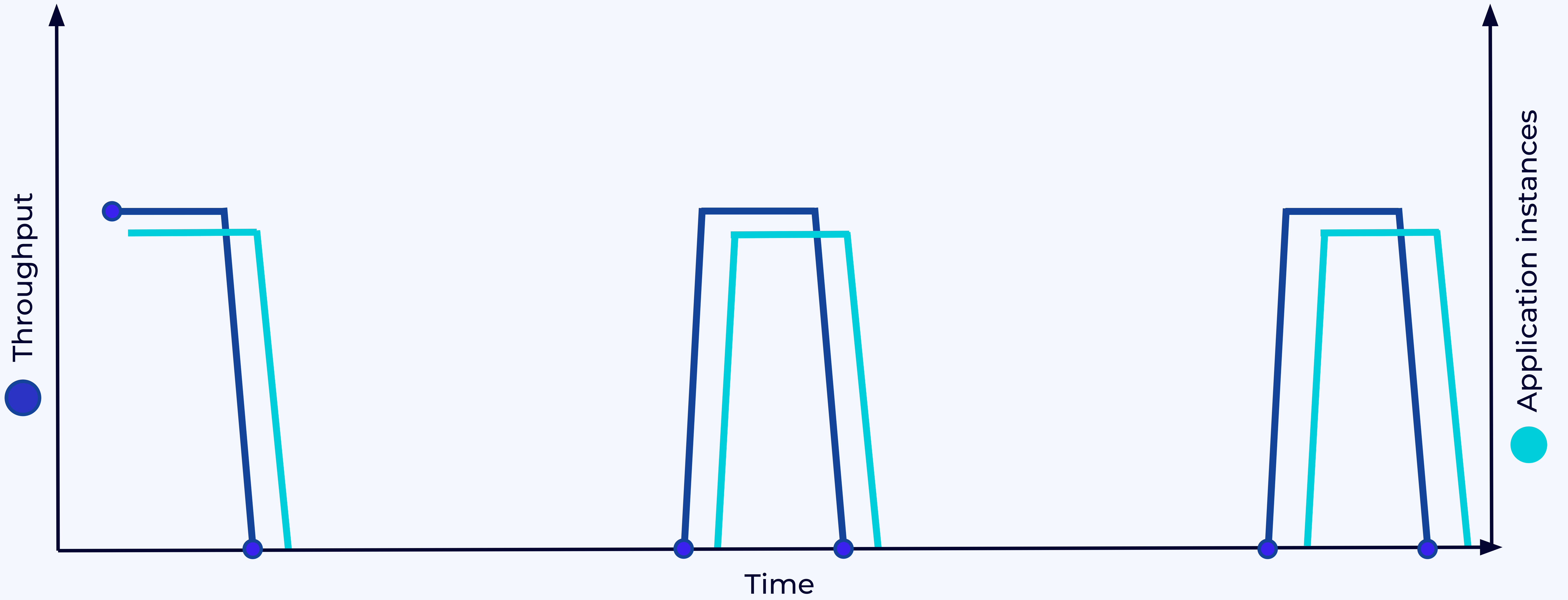
KEDA: Scaling a Deployment to zero

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: kafka-streams-app-scaledobject
  namespace: default
spec:
  scaleTargetRef:
    name: kafka-streams-app
  pollingInterval: 5
  minReplicaCount: 0
  cooldownPeriod: 300
  triggers:
  - type: kafka
    metadata:
      bootstrapServers: localhost:9092
      consumerGroup: my-group
      topic: input-topic
      lagThreshold: "50"
```

Allow KEDA to scale Deployment to 0 replicas

Wait 300 seconds before scaling to zero

Scaling Kafka applications to zero



Summary

Summary

Network cost is often surprisingly high

Lag-based scaling can optimize compute of fluctuating workloads

Follower fetching minimizes cross-AZ traffic of consumers

Consider "scaling to zero" for use cases with batch data sources

Compression reduces produce/consume traffic

CONFLUENT

Taming the cost of Kafka workloads in the cloud

Stefan Sprenger
Staff Software Engineer
ssprenger@confluent.io