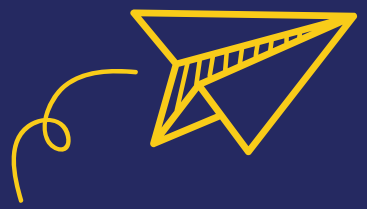


APACHE kafka®

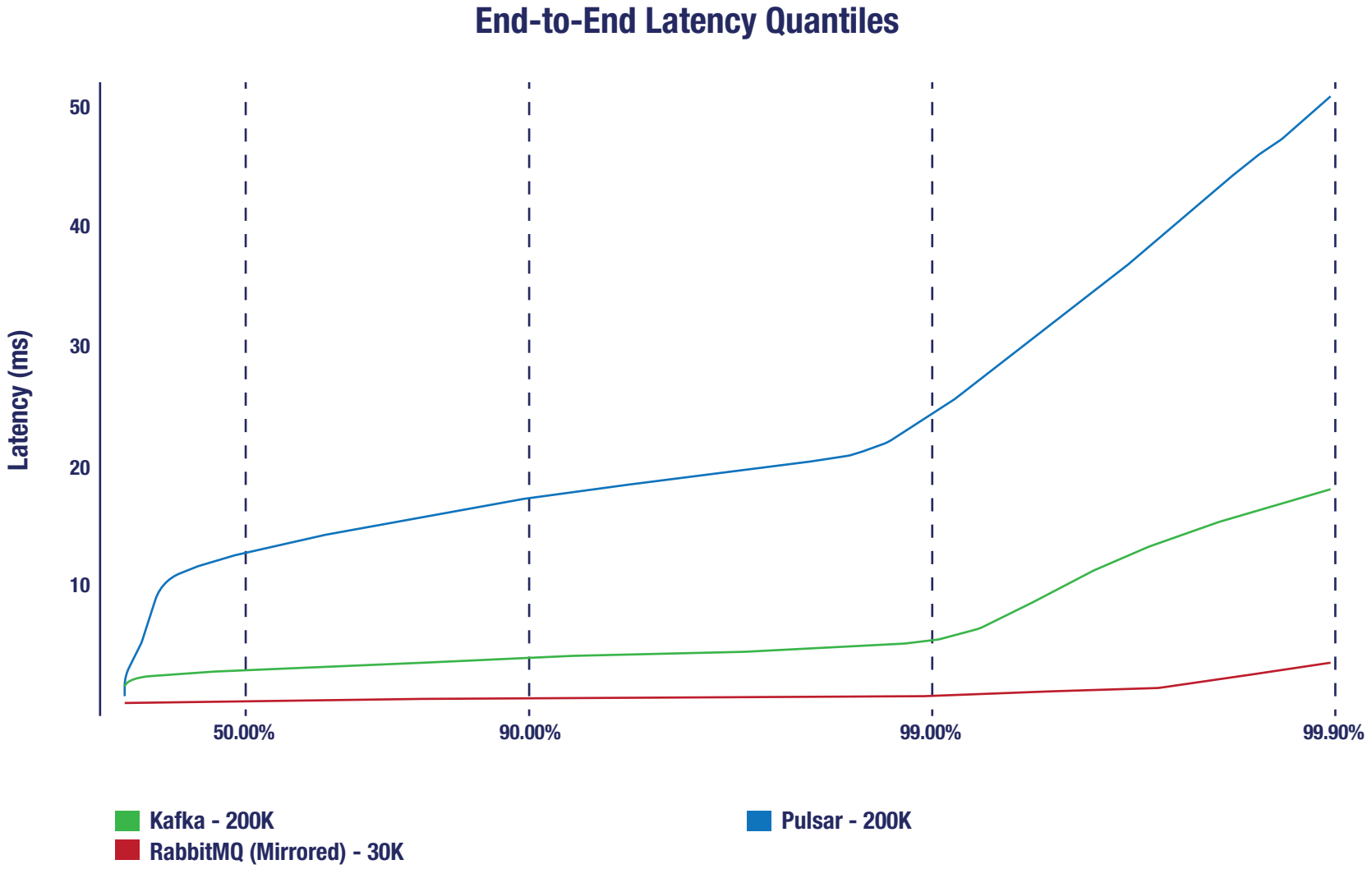
WORKSHOP

WHAT IS IT?

Kafka is an Event Streaming platform which allows the sending and receiving of messages between applications in a decoupled manner with a very high throughput. If you know Rabbit MQ this won't be a foreign concept to you.

What makes Kafka more attractive is throughput and persistence. You can set Kafka to keep messages for a certain time, defaulting to two weeks, so if an app goes down it can continue from where it left off.

THROUGHPUT



In the above image the message throughputs are Kafka 200K/s, Pulsar 200K/s, RabbitMQ 30K/s

	Kafka	Pulsar	RabbitMQ (Mirrored)
Peak Throughout (MB/s)	605 MB/s	305 MB/s	38 MB/s
p99 Latency (ms)	5 ms (200 MB/s load)	25 ms (200 MB/s load)	1 ms* (reduced 30 MB/s load)

*RabbitMQ latencies degrade significantly at throughputs higher than the 30 MB/s.

BETTER THAN CALLING AN API?

If your call doesn't need to be syncrhonous (don't need to wait for an answer) then there are a number of benefits to using Kafka.

1. The producer does not need to know about the consumers. In an API setup every time a new API wants to receive the message you need to update the Sender to know about this, it is also more overhead for the Sender to send to more than one place. With Kafka, Consumers attach to the topic at will and do not affect the Producer.

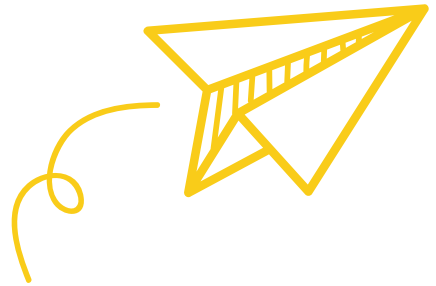
2. If the Consumer is down it does not affect the Producer, the Producer can carry on sending. In an API setup your API calls will fail and the messages will be lost. Waiting and retrying will be a big overhead for the Sender.

3. The messages in Kafka are persisted so the Consumer can be down for however long you retain messages. Also new consumers can come on and consumer messages as far back as your retaining period.

SOURCE CODE (C#)

PRODUCER

The producer is thread safe so you can have it as a singleton and call it from many threads.
Producer config:



```
ProducerConfig _producerConfig = new ProducerConfig
{
    BootstrapServers = "127.0.0.1:29092"
};
```

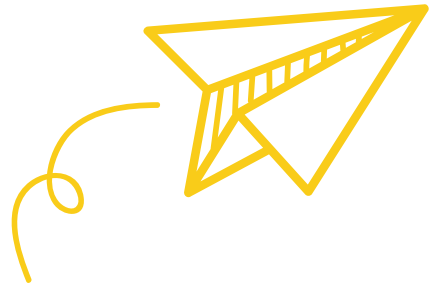
Produce:

```
using (var producer = new ProducerBuilder<string, string>(_producerConfig).Build())
{
    string json = JsonConvert.SerializeObject(weather);
    var message = new Message<string, string> { Key = DateTime.Now.ToString("yyyyM-
MddHHmmss"), Value = json };

    var deliveryResult = await producer.ProduceAsync("weather", message);

    return deliveryResult.Status.ToString();
}
```

CONSUMER

With the consumer you would usually have one thread per partition to allow parallel processing of messages but maintaining order per partition.
Consumer config:

```
ConsumerConfig _consumerConfig = new ConsumerConfig
{
    BootstrapServers = "127.0.0.1:29092",
    GroupId = "my-consumer-group",
    AutoOffsetReset = AutoOffsetReset.Earliest,
    EnableAutoCommit = false
};
```



Consume:

```
using (var consumer = new ConsumerBuilder<string, string>(_consumerConfig).Build())
{
    consumer.Subscribe("weather");

    while (!stoppingToken.IsCancellationRequested)
    {
        var consumeResult = consumer.Consume(stoppingToken);
        _logger.LogInformation(consumeResult.Message.Value);
        consumer.Commit(consumeResult);
    }

    consumer.Close();
}
```

TOPIC

A topic is a logical group of messages. For example the “bet” topic would contain bets, the “event” topic would contain events.

PARTITIONS

A partition allows parallel processing of messages within a topic. If you only have 1 partition you can only process one message at a time. If you have 20 partitions you can process up to 20 messages at a time. This can be split between consumers and threads within a consumer. So if you have 2 consumers each can consume 10 messages at a time in parallel. You will be limited to a maximum of 20 consumers, you cannot have more consumers than partitions as the extra consumers will have no work to do and will sit idle.

Partitions also keep messages in order within that partition. So if you need all messages related to a certain betting event to be processed in order (odds changes for example always need to be in order) then you will set a message key with the event id. Kafka will ensure the same key (event id, therefore event) will always go to the same partition, thus guaranteeing order.

CONSUMER GROUPS

If you have 10 consumers in the same consumer group they will share the load of processing the messages. Messages coming in will be divided between the consumers in the consumer group and no consumer will get the same message another consumer in the group got. If you want to process the same messages for another purpose you would then create a new consumer group.

For example lets say we are consuming the bet topic, you have one set of consumers that will insert the bet into a bet database table, we create a group called bet_storage_group. Then lets say you have another set of consumers that will insert into the client transaction table in a database, we can create a group called client_transaction_storage_group. This way both groups will receive the same messages.

OFFSETS

There will be an offset per topic, partition and consumer group. The offset is how far along the partition in the topic that consumer group is. So if the consumer group has processed and committed the offset for 100 messages the offset should be 100. If there are still another 900 messages in the partition (total 1000) then this is referred to as a lag of 900 messages and is a good metric to see how far behind the consumers are. So in our previous consumer group example the two groups could be at different places.

`bet_storage_group` : offset could be 100 (lag of 900)

`client_transaction_storage_group` : offset could be 1000 (lag of 0)

KAFKA SHELL SCRIPTS

Access the Kafka container shell and go to
/opt/bitnami/kafka/bin from here type in ls to view all the scripts.