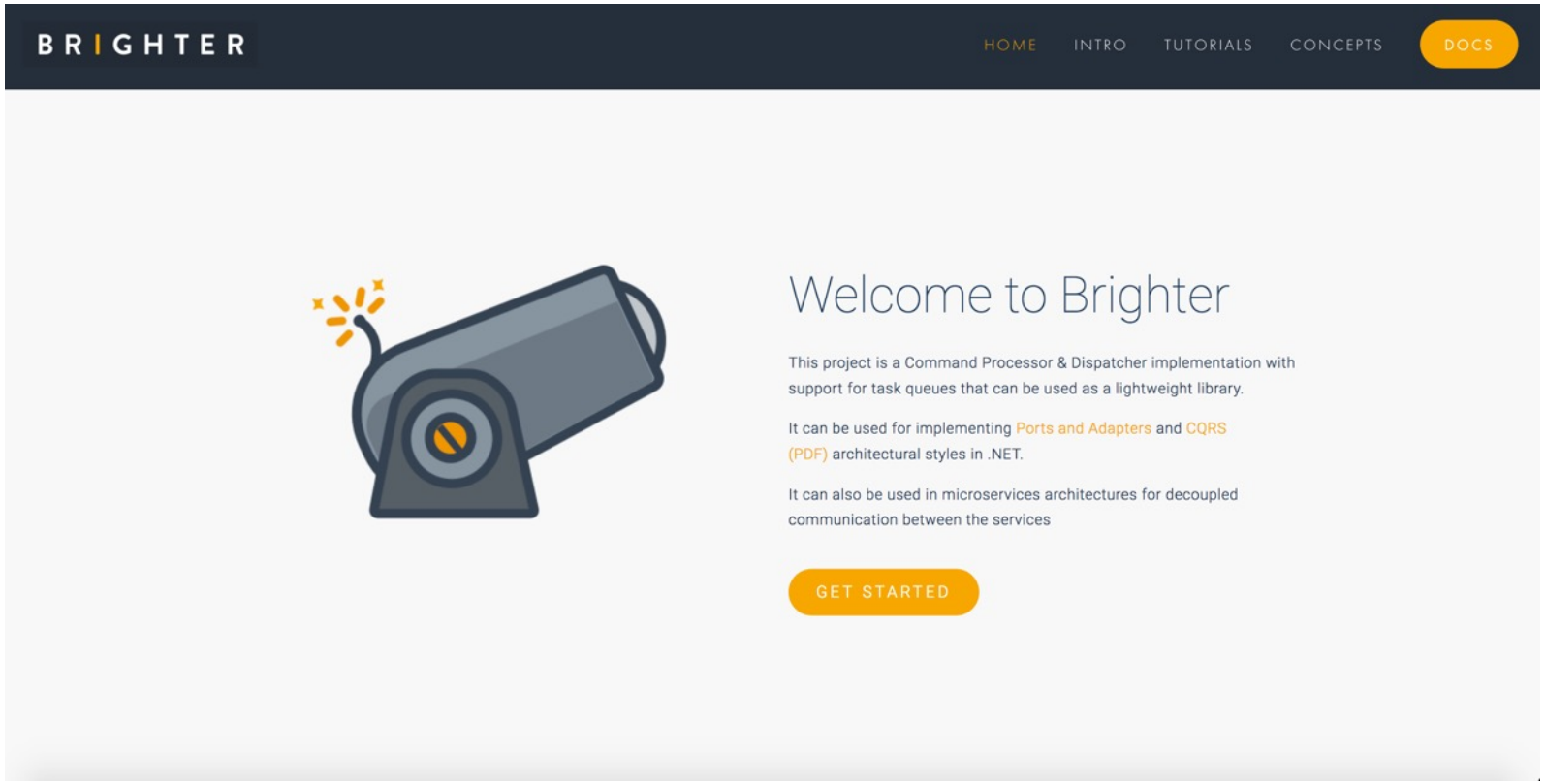
Kafka for .NET Developers

Ian Cooper

X/Hachyderm: ICooper

Who are you?

- Software Developer for more than 25 years
  - Stuff I care about: Messaging, EDA, Microservices, TDD, XP, OO, RDD & DDD, Code that Fits in My Head, C#
  - Places I have worked: DTI, Reuters, Sungard, Beazley, Huddle, Just Eat Takeaway
- No smart folks
  - Just the folks in this room


BRIGHTER
HOME
INTRO
TUTORIALS
CONCEPTS
DOCS
Welcome to Brighter
This project is a Command Processor & Dispatcher implementation with support for task queues that can be used as a lightweight library.
It can be used for implementing Ports and Adapters and CQRS (PDF) architectural styles in .NET.
It can also be used in microservices architectures for decoupled communication between the services
GET STARTED

Links

The Sample Code: https://github.com/iancooper/KafkaForDotNetDevs
Brighter: https://github.com/BrighterCommand/Brighter
This Presentation: https://github.com/iancooper/Presentations

Agenda

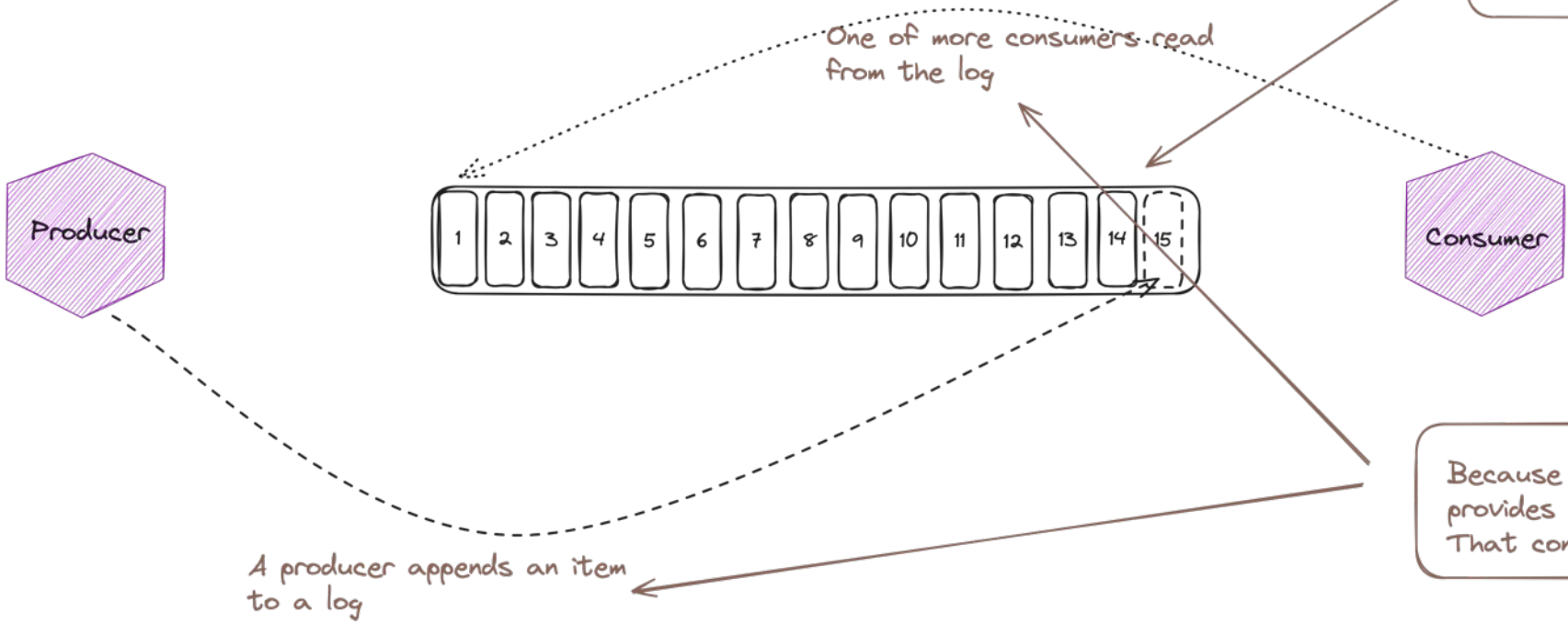
1. Kafka
2. Managing Kafka
3. Kafka and C#
4. Under the Hood
5. Reliable Producer
6. Reliable Consumer
7. Serializers, De-serializers & Schema Registry
8. Brighter
9. Q&A

Kafka

Kafka was created at Linked In by Jay Kreps, Neha Narkhede, Jun Rao in 2011.
it became an Apache OSS project in October 2012

Original use case was data engineering, capturing data into LinkedIn's Data Warehouse from diverse sources so that it could be used for later analysis.

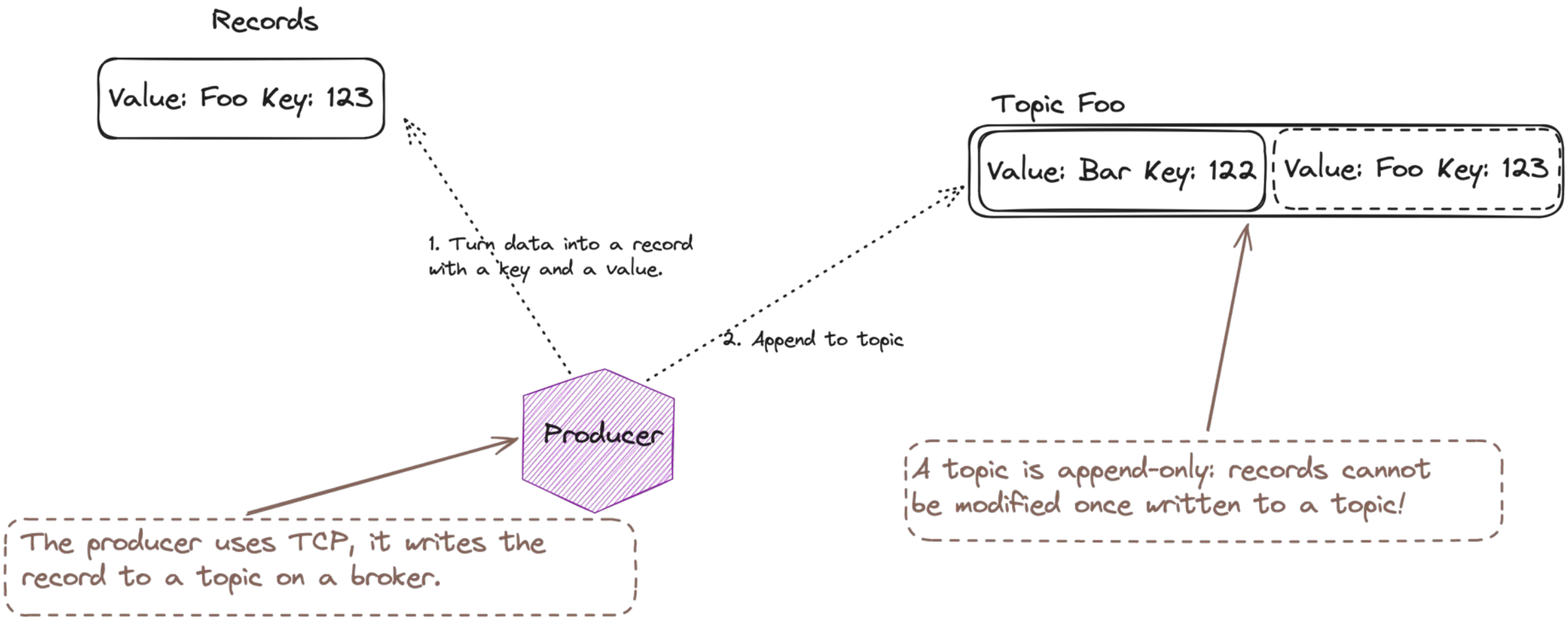
An append only log is FAST giving low latency to the integration of data between producer and consumer.



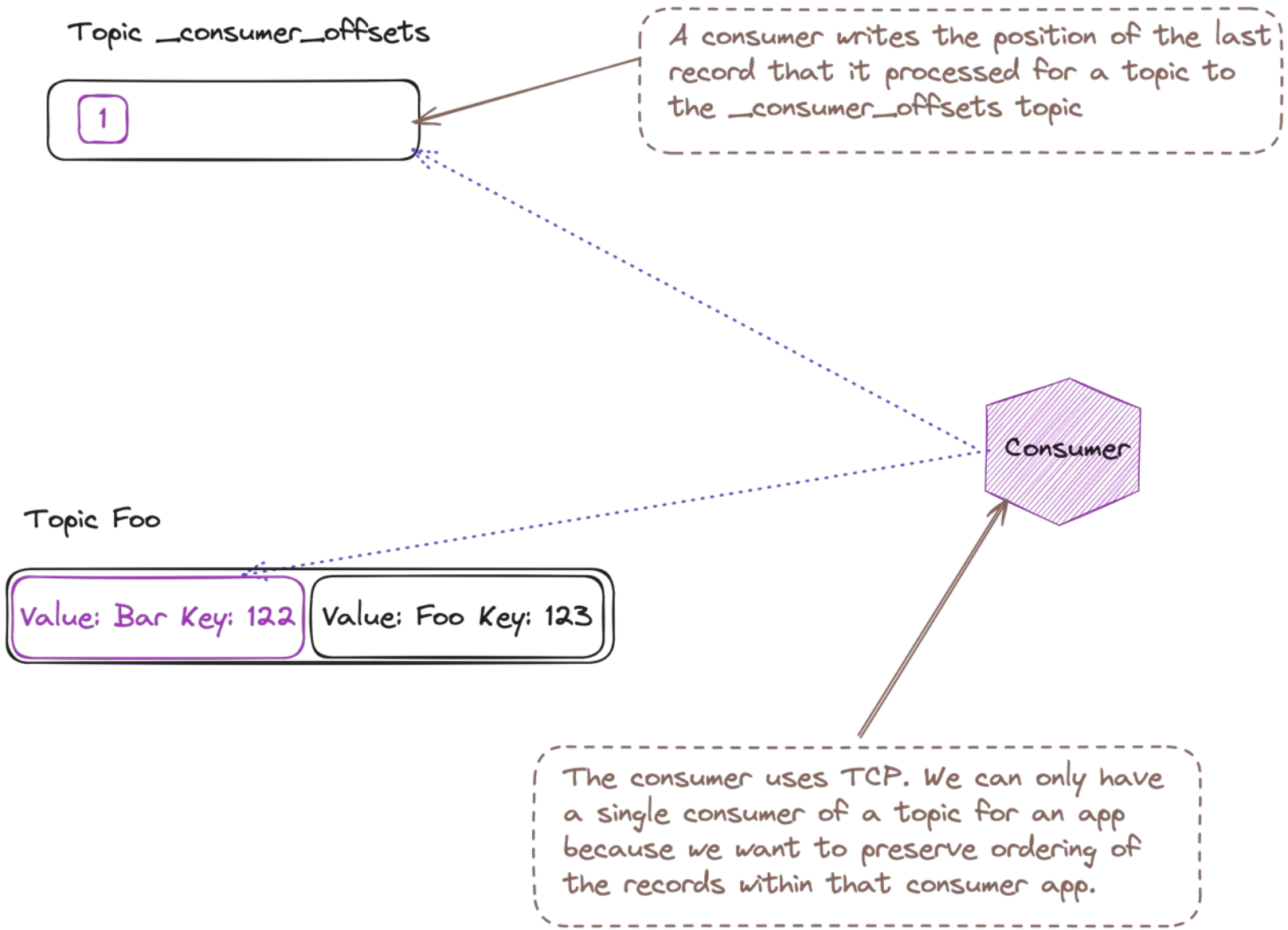
Because a log buffers these two interactions it provides a mechanism for an asynchronous conversation. That conversation can be ordered

https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

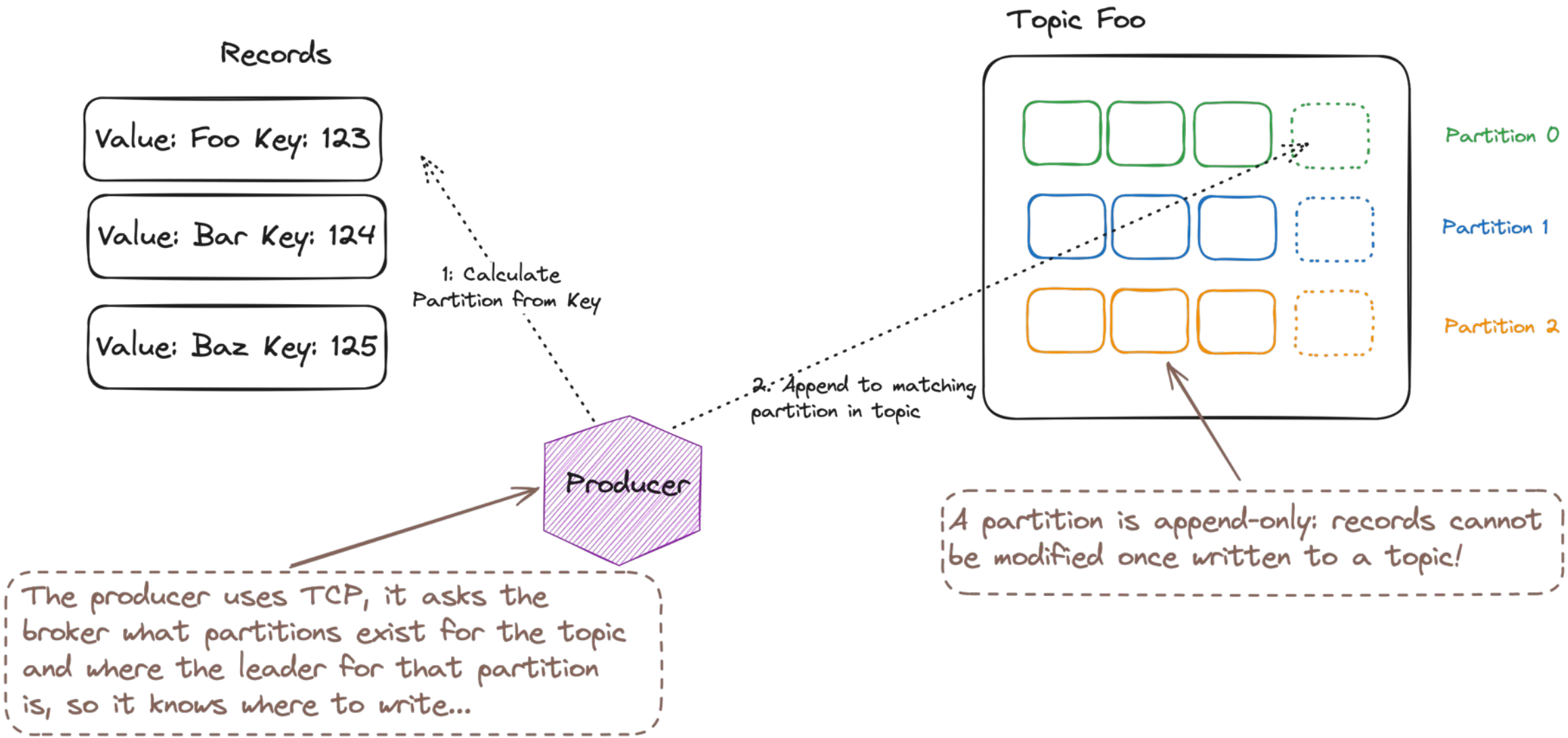
Kafka Topics



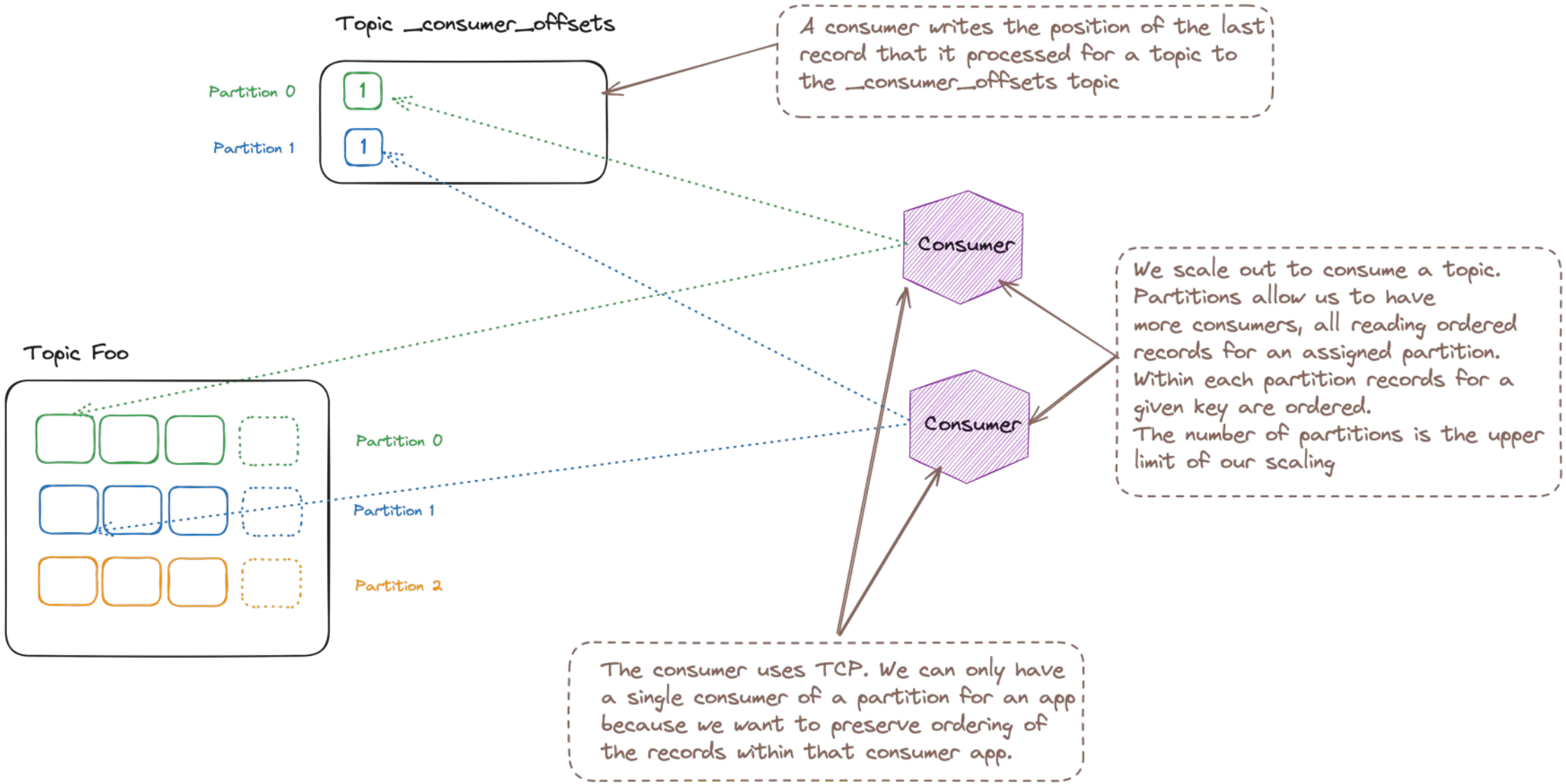
Kafka Topics



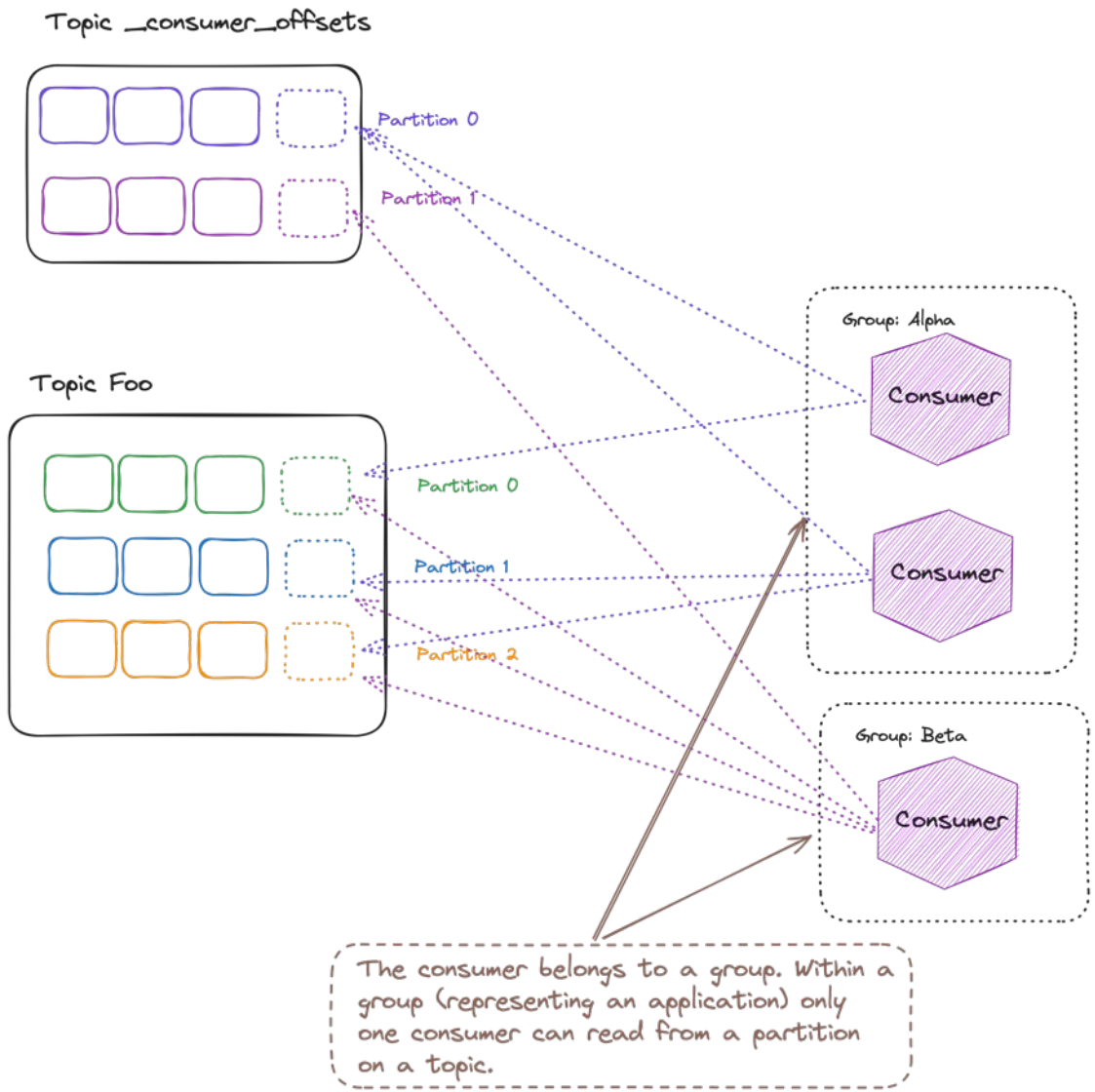
Kafka Topics



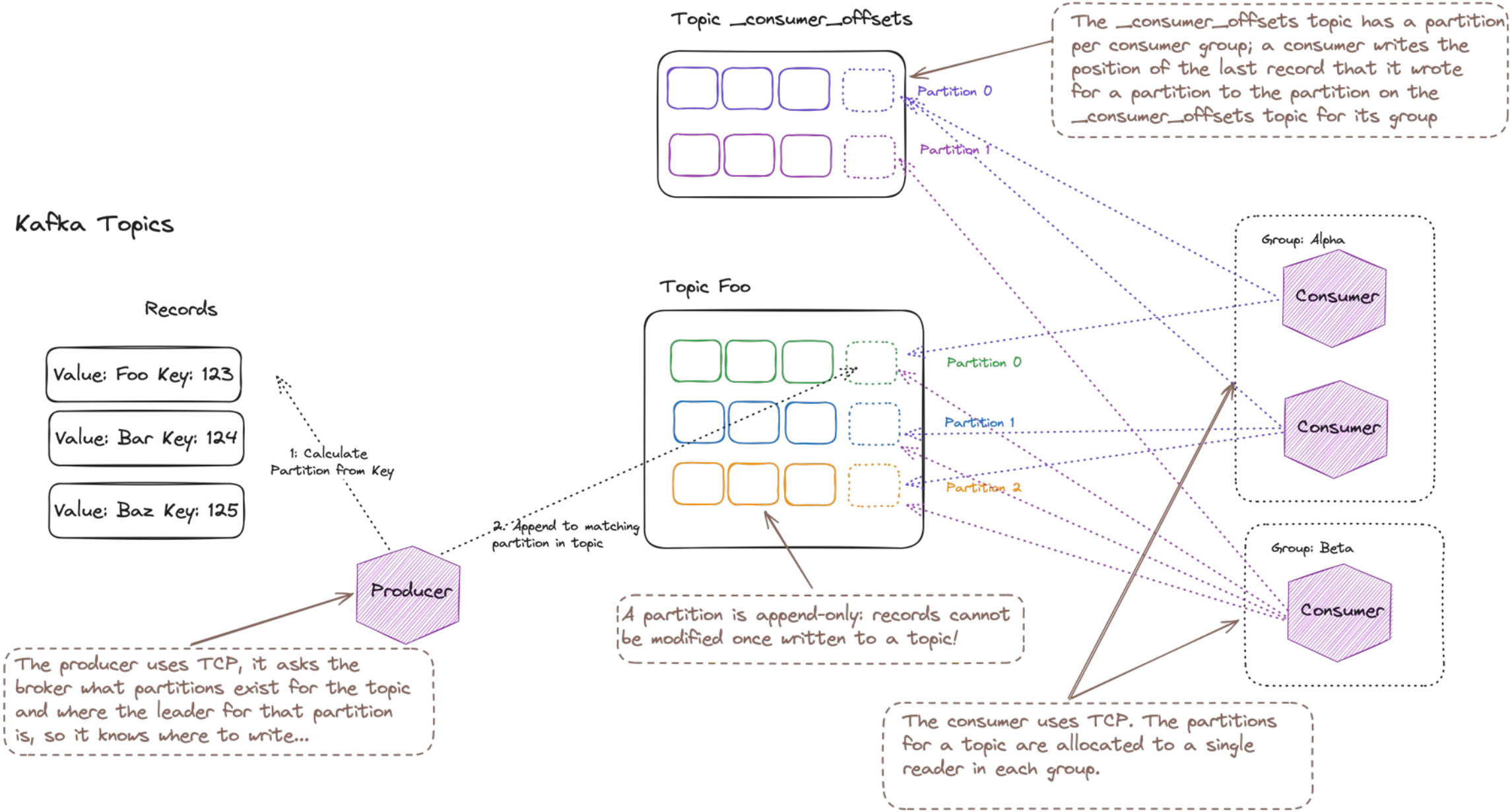
Kafka Topics



Kafka Topics



Kafka Topics



Running Kafka

```
kafka0:
    image: confluentinc/cp-kafka:7.2.1.arm64
    hostname: kafka0
    container_name: kafka0
    ports:
      - 9092:9092
      - 9997:9997
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "false"
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,CONTROLLER:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka0:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_PROCESS_ROLES: 'broker,controller'
      KAFKA_NODE_ID: 1
      KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka0:29093'
      KAFKA_LISTENERS: 'PLAINTEXT://kafka0:29092,CONTRO                    :29093,PLAINTEXT_HOST:
//0.0.0.0:9092'
      KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
      KAFKA_LOG_DIRS: '/tmp/kraft-combined-logs'
      KAFKA_JMX_PORT: 9997
      KAFKA_JMX_OPTS: -Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false
-Djava.rmi.server.hostname=kafka0 -Dcom.sun.management.jmxremote.rmi.port=9997
    volumes:
      - ./scripts/update_run.sh:/tmp/update_run.sh
    command: "bash -c 'if [ ! -f /tmp/update_run.sh ]; then echo \"ERROR: Did you
the update_run.sh file that came with this docker-compose.yml file?\" && exit 1 ; else
/tmp/update_run.sh && /etc/confluent/docker/run ; fi'"
```

If a topic does not exist when a producer writes, what should Kafka do?

The advertised listeners are the endpoints that clients should use to communicate with broker

The roles that this node plays

The listeners are the endpoints that the broker listens on

We need to run a script to run without zookeeper

Managing Kafka

CLI

Kafka CLI

Suite of tools located in the /bin directory where you installed Kafka:

kafka-topics.sh => create or delete or topic

kafka-console-producer.sh => produce records to a topic

kafka-console-consumer.sh => consume records from a topic

kafka-get-offsets.sh => retrieve topic/partition offsets

...many more

https://docs.confluent.io/kafka/operations-tools/kafka-tools.html

KCat

A generic, non-JVM, producer and consumer for Kafka.

Producer Mode:

Read messages from stdin (or a file)

```
kcat -b localhost:9092 -t my_topic -P
test
```

Consumer Mode:

Read messages from a topic and writes to stdout

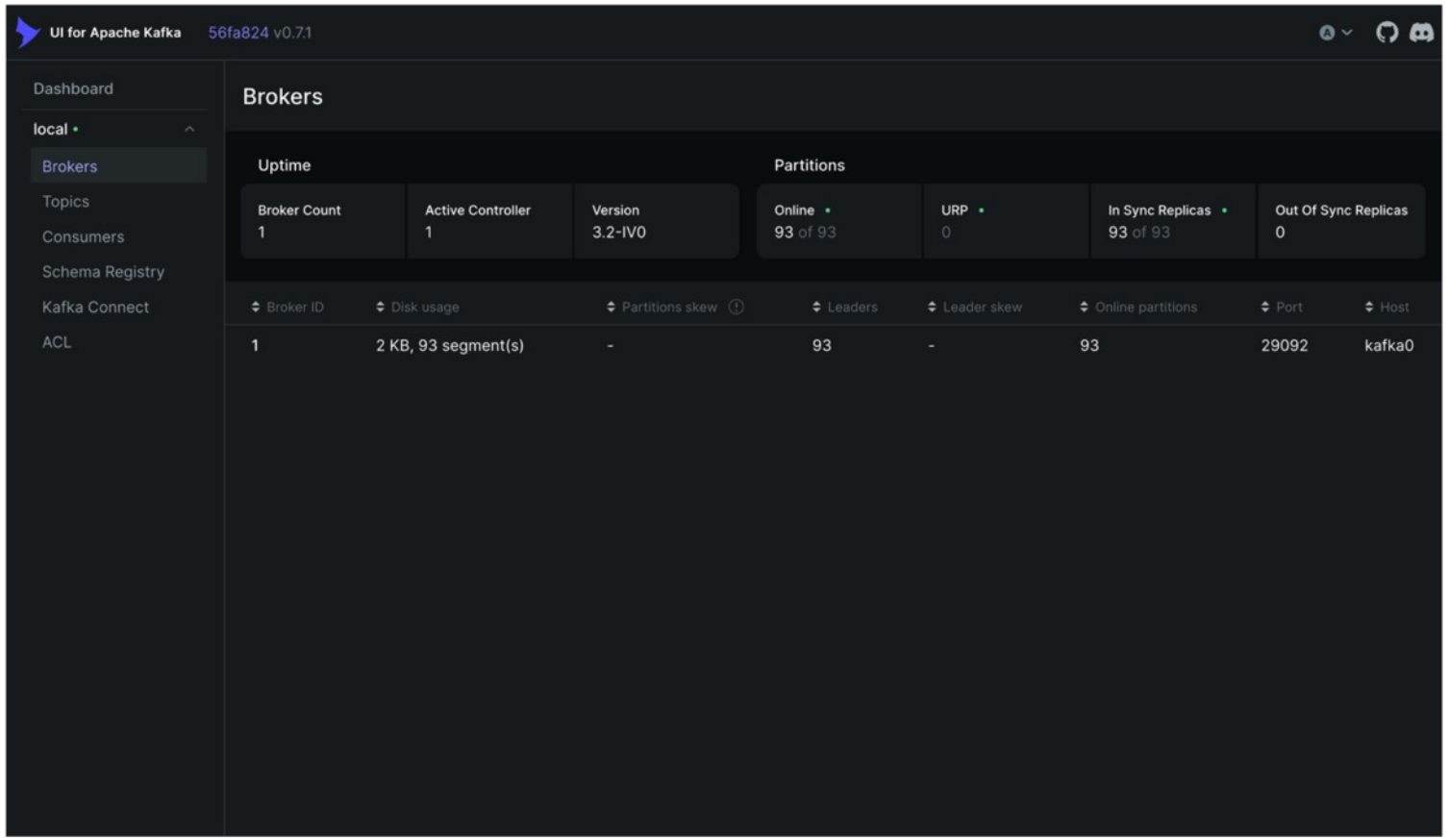
```
kcat -b localhost:9092 -t my-topic
```

https://github.com/edenhill/kcat

Demo One: CLI Tools

```
# we want to execute some commands on our kafka server
docker container exec -it kafka0 /bin/bash

# add a new topic
cd /bin
kafka-topics --create --if-not-exists --topic test-topic --partitions 3 --bootstrap-server localhost:9092

#use kafka cli
kafka-console-producer --broker-list localhost:9092 --topic test-topic
{"name":"Coops"}

kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic -from-beginning

# exit from Docker, and back to the host terminal session
#use kcat

kcat -b localhost:9092 -t test-topic
kcat -b localhost:9092 -t test-topic -P
{"name":"Ian"}

#terminate the producer
[CTRL+D]
```

UIs



Other OSS UIs:

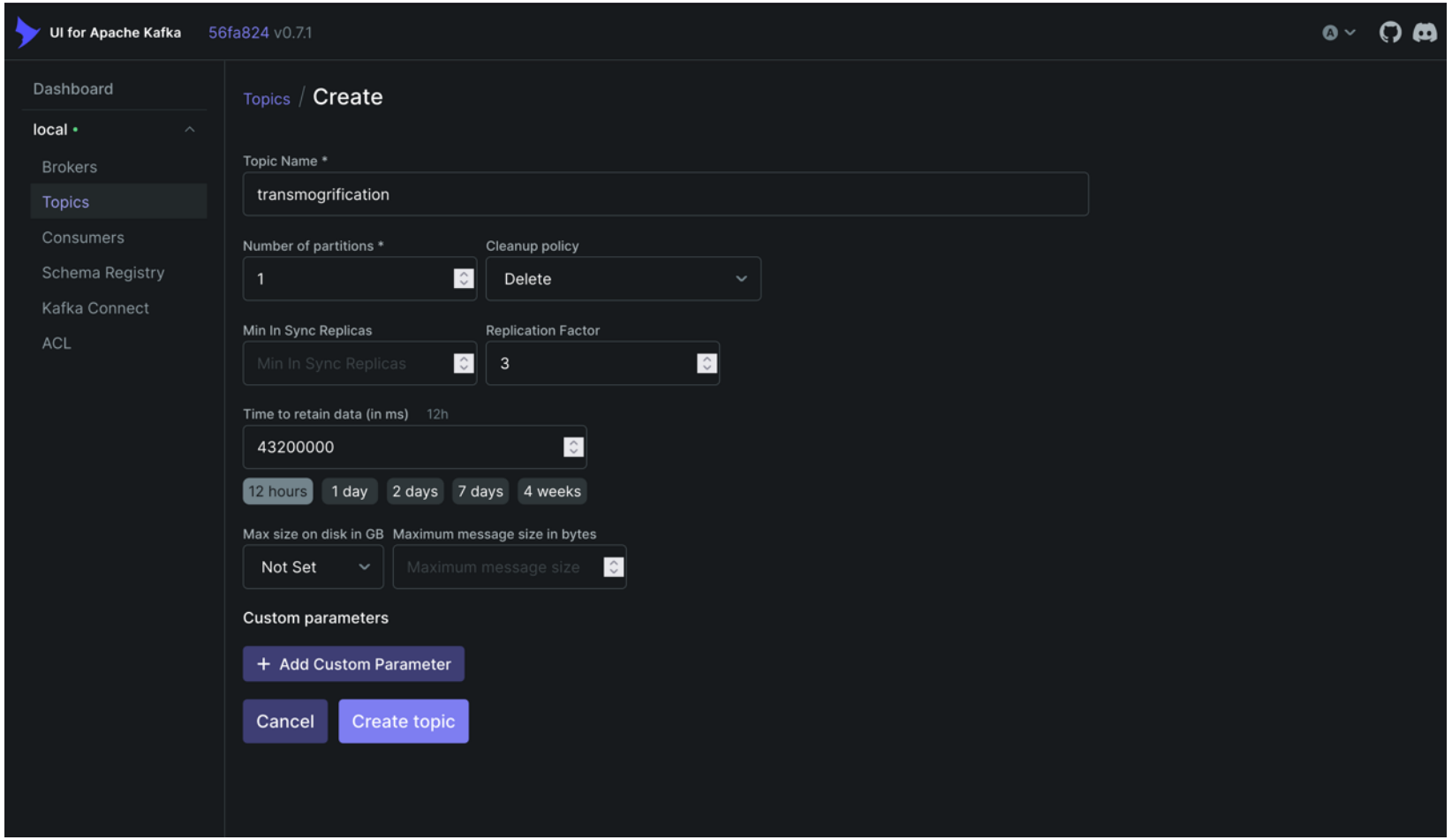
AKHQ
Kafdrop
Lenses
CMAK

Paid UIs:

Confluent CC
Conduktor

UI for Apache Kafka: https://github.com/provectus/kafka-ui/tree/master

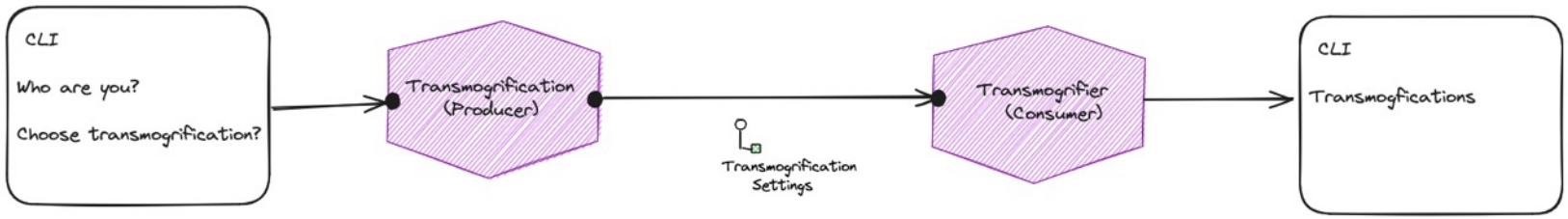
Demo: UI Tools



Kafka and C#

Building a Transmogrifier

Calvin and Hobbes – Bill Watterson



Producing a Message in C#

```
var producerConfig = new Dictionary<string, string>
{
    { "bootstrap.servers", "localhost:9092" }
};
```

The topic we want to produce to.

```
using Confluent.Kafka;

namespace Transmogrification;

public class Dispatcher(string topic, Dictionary<string, string> producerConfig) : IDisposable
{
    private readonly IProducer<string,string> _producer = new ProducerBuilder<string, string>(producerConfig).Build();

    public void Transmogrify(TransmogrificationSettings settings)
    {
        _producer.Produce(
            topic,
            new Message<string, string>
            {
                Key = settings.Name,
                Value = settings.Transformation
            }
        );
        _producer.Flush(TimeSpan.FromSeconds(10));
    }

    public void Dispose()
    {
        _producer.Dispose();
    }
}
```

Create a producer from the config; note that ProducerBuilder<string, string> means:
- key: string
- value: string

Tidy.

Produce a message to a topic; build message from our data

Consuming a Message in C#

The topic we want to consume from.

```
1 using Confluent.Kafka;
2 using Spectre.Console;
3
4 namespace Transmogrifier;
5
6 public class MessagePump(string topic, Dictionary<string, string> consumerConfig)
7 {
8     private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig).Build();
9
10    public async Task Run<TDataType>(
11        Func<Message<string, string>, TDataType> translator,
12        Action<TDataType> handler,
13        CancellationToken cancellationToken = default)
14    {
15        ...
16    }
17 }
```

```
1 var consumerConfig = new Dictionary<string, string>()
2 {
3     { "bootstrap.servers", "localhost:9092" },
4     { "group.id", "transmogrification-consumer" },
5 };
```

Create a consumer from the config; note that IConsumer<string, string> means:
- key: string
- value: string

Generic, so pass in:
- translator
- handler

Consuming a Message in C#

```
try
{
    _consumer.Subscribe(topic);

    while (true)
    {
        var consumeResult = _consumer.Consume(cancellationToken);

        if (consumeResult.IsPartitionEOF)
        {
            await Task.Delay(1000, cancellationToken);
            continue;
        }

        var dataType = translator(consumeResult.Message);

        handler(dataType);
    }
}
// error handling
finally
{
    _consumer.Close();
}
```

Subscribe to topic

Message Pump is an endless loop

Read next result; SDK is buffering for us. Yield if nothing in buffer. Break if cancelled.

Translate the message

Handle the message

Tidy

Consuming a Message in C#

```
var consumerConfig = new Dictionary<string, string>()
{
    { "bootstrap.servers", "localhost:9092" },
    { "group.id", "transmogrification-consumer" },
};
```

The topic we want to consume from.

Generic, so pass in:
- translator
- handler

Subscribe to topic

Message Pump is an endless loop

Read next result; SDK is buffering for us. Yield if nothing in buffer. Break if cancelled.

```
using Confluent.Kafka;
using Spectre.Console;

namespace [illegible];

public class MessagePump(string topic, Dictionary<string, string> consumerConfig)
{
    private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig).Build();

    public async Task Run<TDataType>(
        Func<Message<string, string>, TDataType> translator,
        Action<TDataType> handler,
        CancellationToken cancellationToken = default)
    {
        try
        {
            _consumer.Subscribe(topic);

            while (true)
            {
                var consumeResult = _consumer.Consume(cancellationToken);

                if (consumeResult.IsPartitionEOF)
                {
                    await Task.Delay(1000, cancellationToken);
                    continue;
                }

                var dataType = translator(consumeResult.Message);

                handler(dataType);
            }
        }
        catch(ConsumeException e)
        {
            AnsiConsole.WriteException(e);
        }
        catch (OperationCanceledException)
        {
            //Pump was cancelled, do nothing
        }
        finally
        {
            _consumer.Close();
        }
    }
}
```

Create a consumer from the config; note that IConsumer<string, string> means:
- key: string
- value: string

Translate the message

Handle the message

Tidy

Using the C# Consumer

```
using Spectre.Console;
using Transmogrifier;

const string topic = "transmogrification";

var consumerConfig = new Dictionary<string, string>()
{
    { "bootstrap.servers", "localhost:9092" },
    { "group.id", "transmogrification-consumer" },
    { "auto.offset.reset", "earliest" }
};

CancellationTokenSource cts = new CancellationTokenSource();
Console.CancelKeyPress += (_, e) => {
    e.Cancel = true; // prevent the process from terminating.
    cts.Cancel();
};
```

The topic we want to consume from.

Configure:
- where is Kafka?
- what is our group?

Using the C# Consumer

```
var box = new Box();
if (box.StarTransformation())
{
    box.BeginTransforming();

    var messagePump = new MessagePump(topic, consumerConfig);
    await messagePump.Run(
        (message) => new Transmogrification(message.Key, message.Value),
        (transmogrification) =>
        {
            //Output the results using a spectre console table
            var table = new Table();
            table.AddColumn("Name");
            table.AddColumn("Transformation");
            table.AddColumn("Result");

            table.AddRow(transmogrification.From, transmogrification.To, "Success");

            AnsiConsole.Write(table);

        },
    cts.Token);

    box.EndTransforming();
}
```

Translate the message to a data type

Handle the message

Using the C# Consumer

The topic we want to consume from.

Configure:
- where is Kafka?
- what is our group?

```
using Spectre.Console;
using Transmogrifier;

const string topic = "transmogrification";

var consumerConfig = new Dictionary<string, string>()
{
    { "bootstrap.servers", "localhost:9092" },
    { "group.id", "transmogrification-consumer" },
};

CancellationTokenSource cts = new CancellationTokenSource();
Console.CancelKeyPress += (_, e) => {
    e.Cancel = true; // prevent the process from terminating.
    cts.Cancel();
};

var box = new Box();
if (box.StarTransformation())
{
    box.BeginTransforming();

    var messagePump = new MessagePump(topic, consumerConfig);
    await messagePump.Run(
        (message) => new Transmogrification(message.Key, message.Value),
        (transmogrification) =>
        {
            //Output the results using a spectre console table
            var table = new Table();
            table.AddColumn("Name");
            table.AddColumn("Transformation");
            table.AddColumn("Result");

            table.AddRow(transmogrification.From, transmogrification.To, "Success");

            AnsiConsole.Write(table);

        },
    cts.Token);

    box.EndTransforming();
}
```

Translate the message to a data type

Handle the data type

Demo Two: Produce and Consume A Message

Calvin and Hobbes – Bill Watterson

```cs
using Confluent.Kafka;

namespace Transmogrification;

public class Dispatcher(string topic, Dictionary<string, string> producerConfig) : IDisposable
{
    private readonly IProducer<string,string> _producer = new ProducerBuilder<string, string>(producerConfig).Build();

    public void Transmogrify(TransmogrificationSettings settings)
    {
        _producer.Produce(
            topic,
            new Message<string, string>
            {
                Key = settings.Name,
                Value = settings.Transformation
            }
        );
        _producer.Flush(TimeSpan.FromSeconds(10));
    }

    public void Dispose()
    {
        _producer.Dispose();
    }
}
```

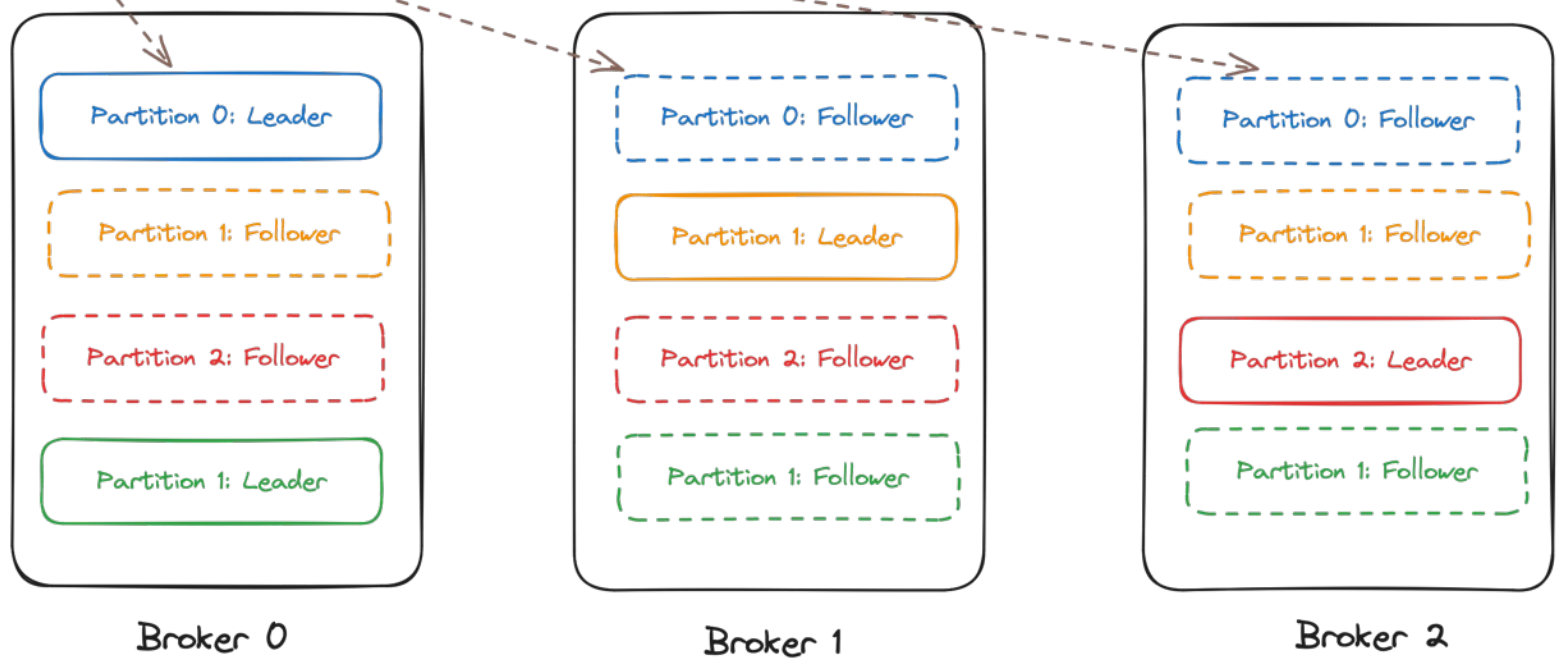
```cs
using Confluent.Kafka;
using Spectre.Console;

namespace Transmogrifier;

public class MessagePump(string topic, Dictionary<string, string> consumerConfig)
{
    private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig).Build();

    public async Task Run<TDataType>(
        Func<Message<string, string>, TDataType> translator,
        Action<TDataType> handler,
        CancellationToken cancellationToken = default)
    {
        try
        {
            _consumer.Subscribe(topic);

            while (true)
            {
                var consumeResult = _consumer.Consume(cancellationToken);

                if (consumeResult.IsPartitionEOF)
                {
                    await Task.Delay(1000, cancellationToken);
                    continue;
                }

                var dataType = translator(consumeResult.Message);

                handler(dataType);
            }
        }
        catch(ConsumeException e)
        {
            AnsiConsole.WriteException(e);
        }
        catch (OperationCanceledException)
        {
            //Pump was cancelled, do nothing
        }
        finally
        {
            _consumer.Close();
        }
    }
}
```

Under the Hood

Kafka Clusters and Partitions

For availability a partition has a leader, and followers on other nodes in the cluster.

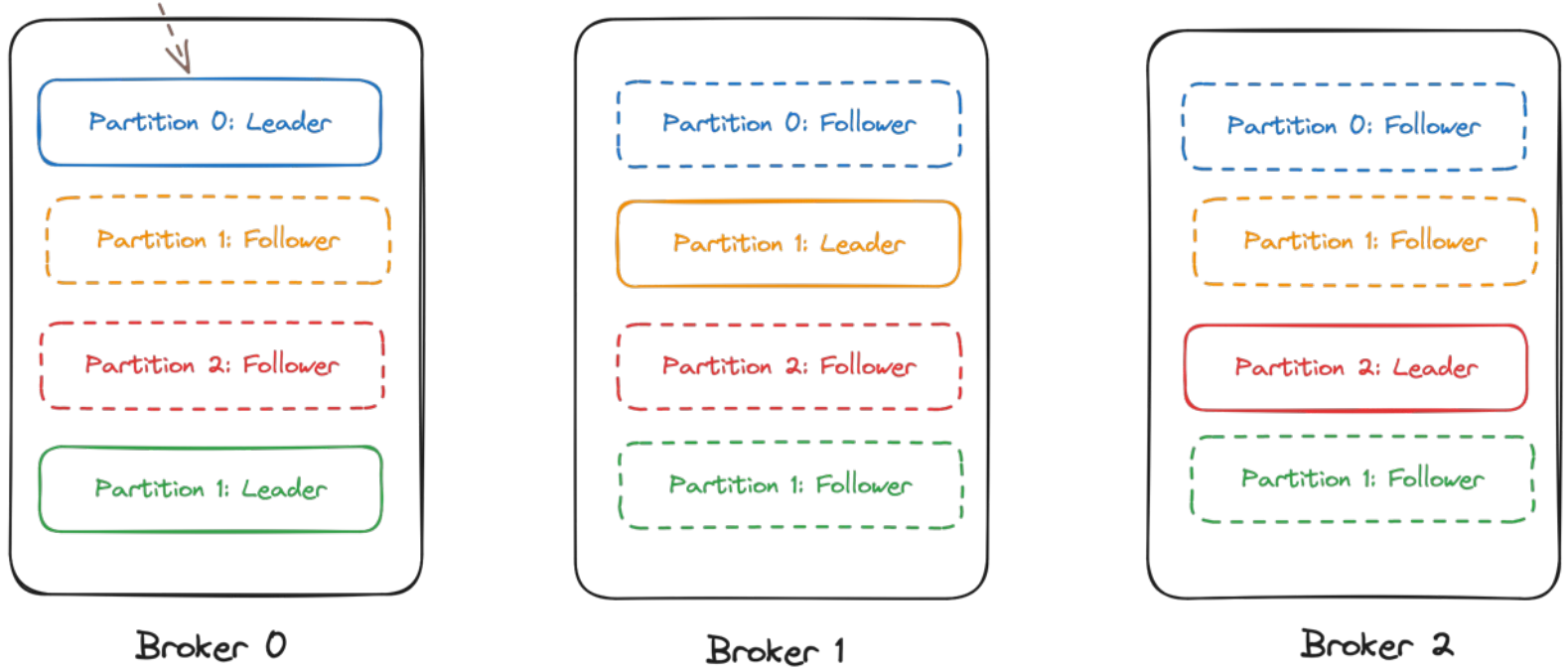
We only ever write to the leader. Before 2.3 read as well; after we can read from the closest replica instead.

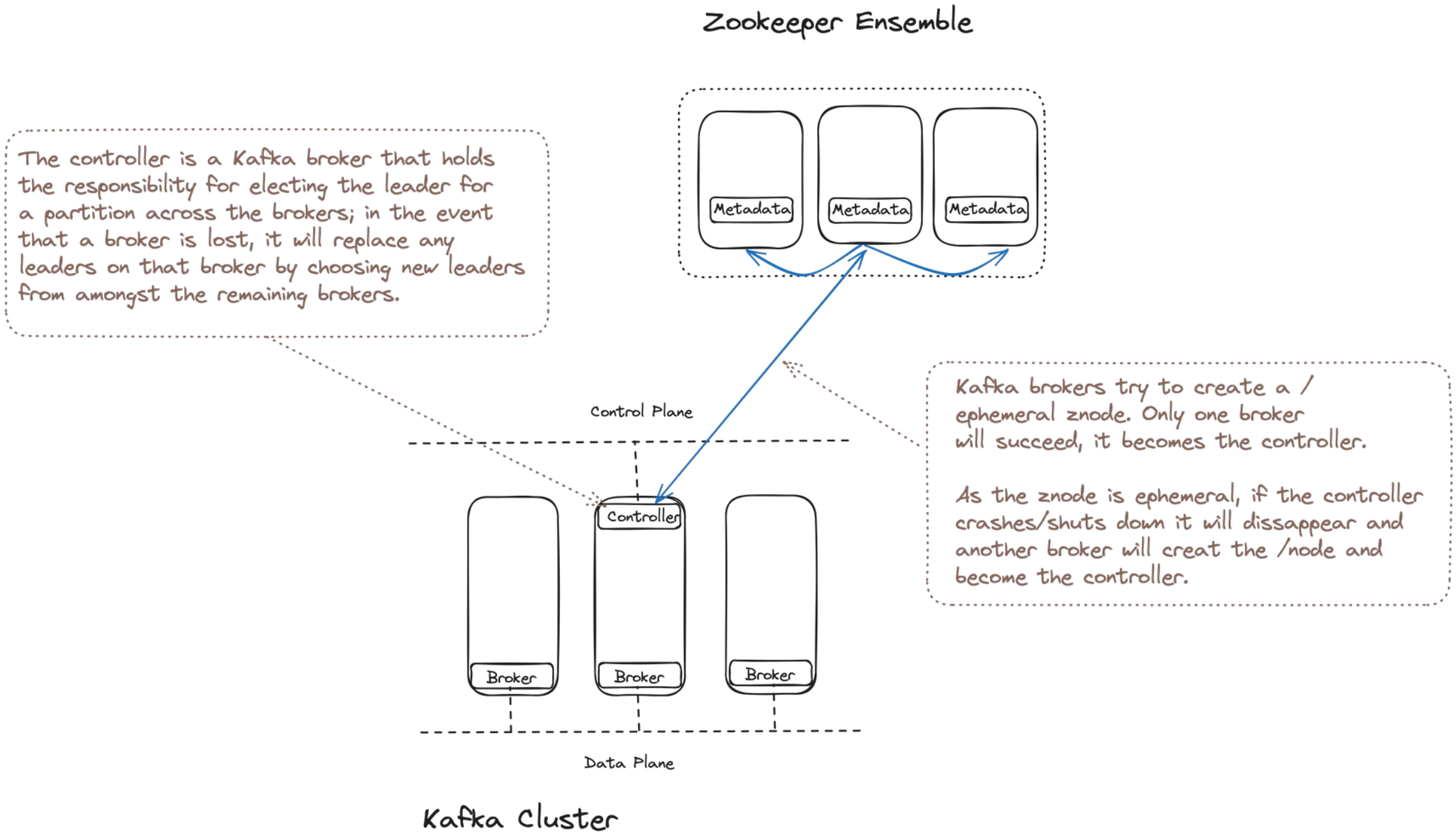


Bootstrap Server

The bootstrap server is just one of the servers that we can query in order to obtain metadata about the location of partitions in the cluster.

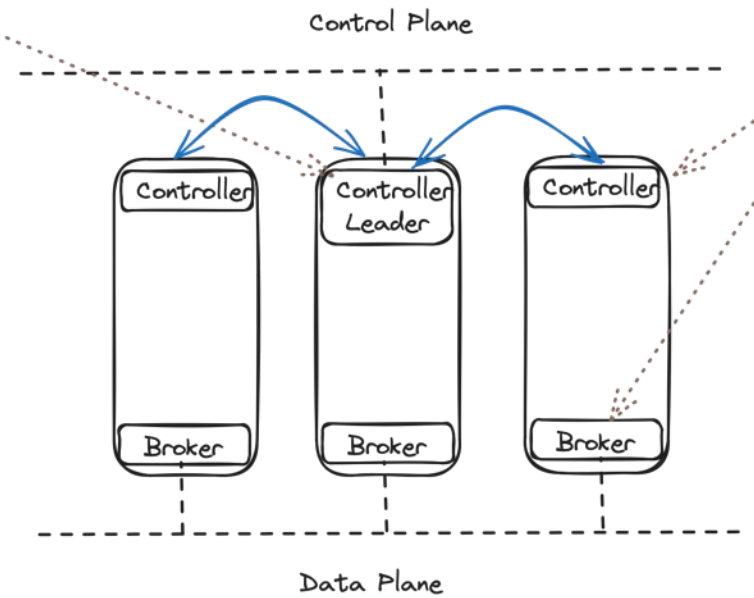
Given that I only write to the leader, how do I know where the leader is?




Zookeeper Ensemble
Metadata
Metadata
Metadata
The controller is a Kafka broker that holds the responsibility for electing the leader for a partition across the brokers; in the event that a broker is lost, it will replace any leaders on that broker by choosing new leaders from amongst the remaining brokers.
Kafka brokers try to create a / ephemeral znode. Only one broker will succeed, it becomes the controller.
As the znode is ephemeral, if the controller crashes/shuts down it will dissappear and another broker will creat the /node and become the controller.
Control Plane
Controller
Broker
Broker
Broker
Data Plane
Kafka Cluster

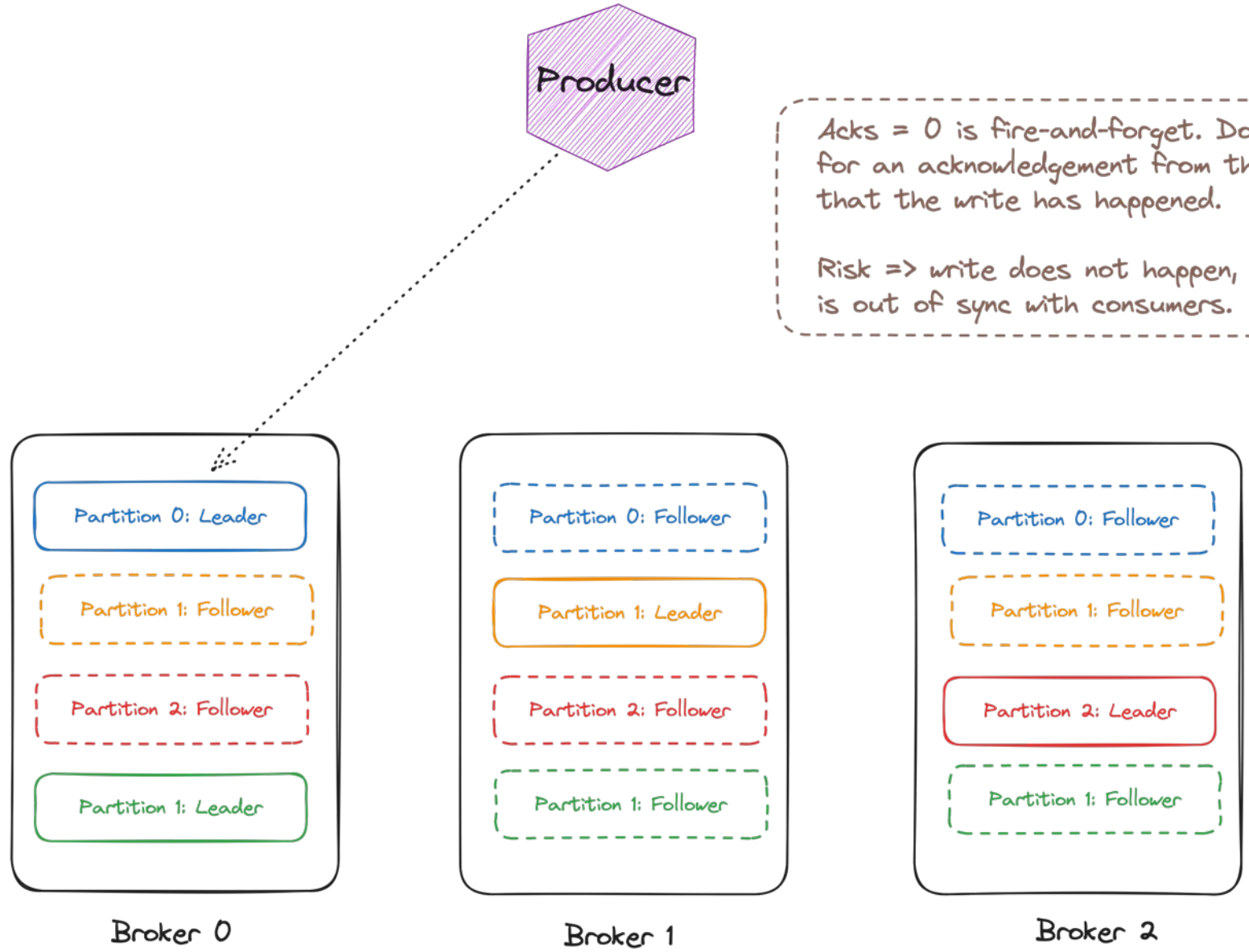
With Kafka Raft (KRaft) we no longer need Zookeeper, instead the brokers implement KRaft and are able to elect a leader from amongst themselves.

Not all nodes are controllers; we need enough in our quorum to break ties. We can configure so that nodes can be both controllers or brokers, or dedicated controllers and brokers.



Kafka Cluster

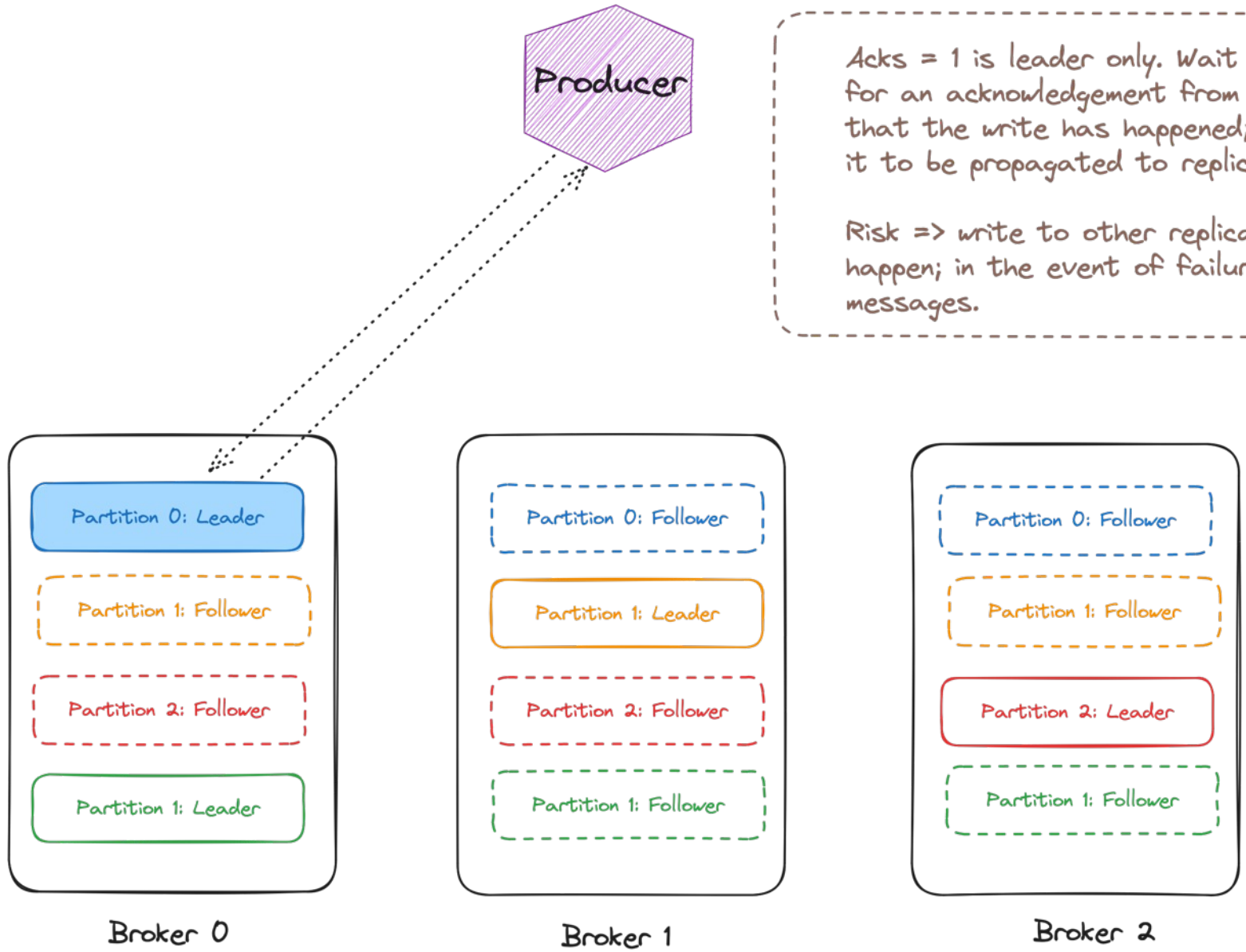
Acks = 0



Acks = 0 is fire-and-forget. Don't wait for an acknowledgement from the server that the write has happened.

Risk => write does not happen, so producer is out of sync with consumers.

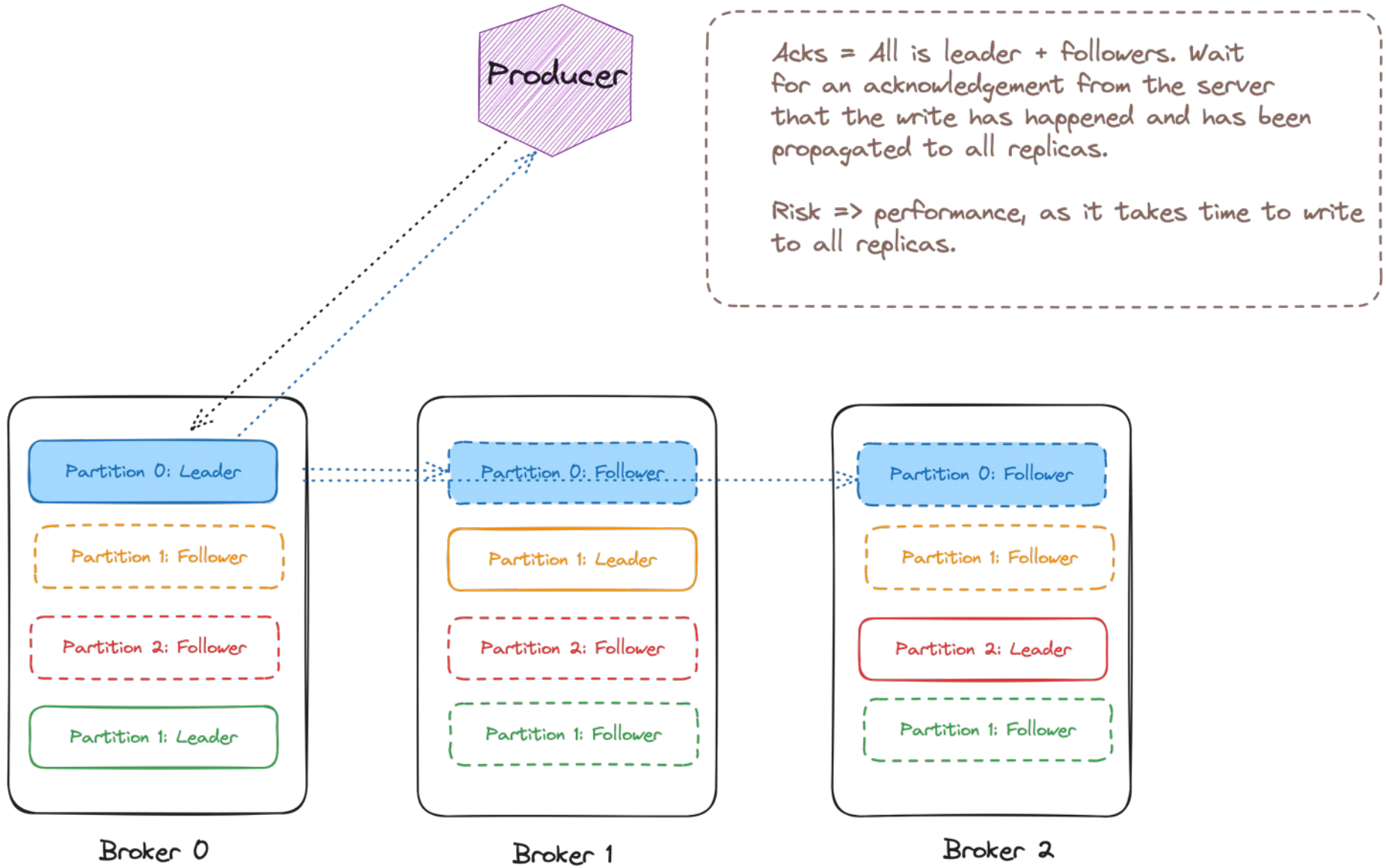
Acks = 1



Acks = 1 is leader only. Wait for an acknowledgement from the server that the write has happened; don't wait for it to be propagated to replicas.

Risk => write to other replicas does not happen; in the event of failure we will lose messages.

Acks = All



Acks = All is leader + followers. Wait for an acknowledgement from the server that the write has happened and has been propagated to all replicas.

Risk => performance, as it takes time to write to all replicas.

In-Sync Replicas

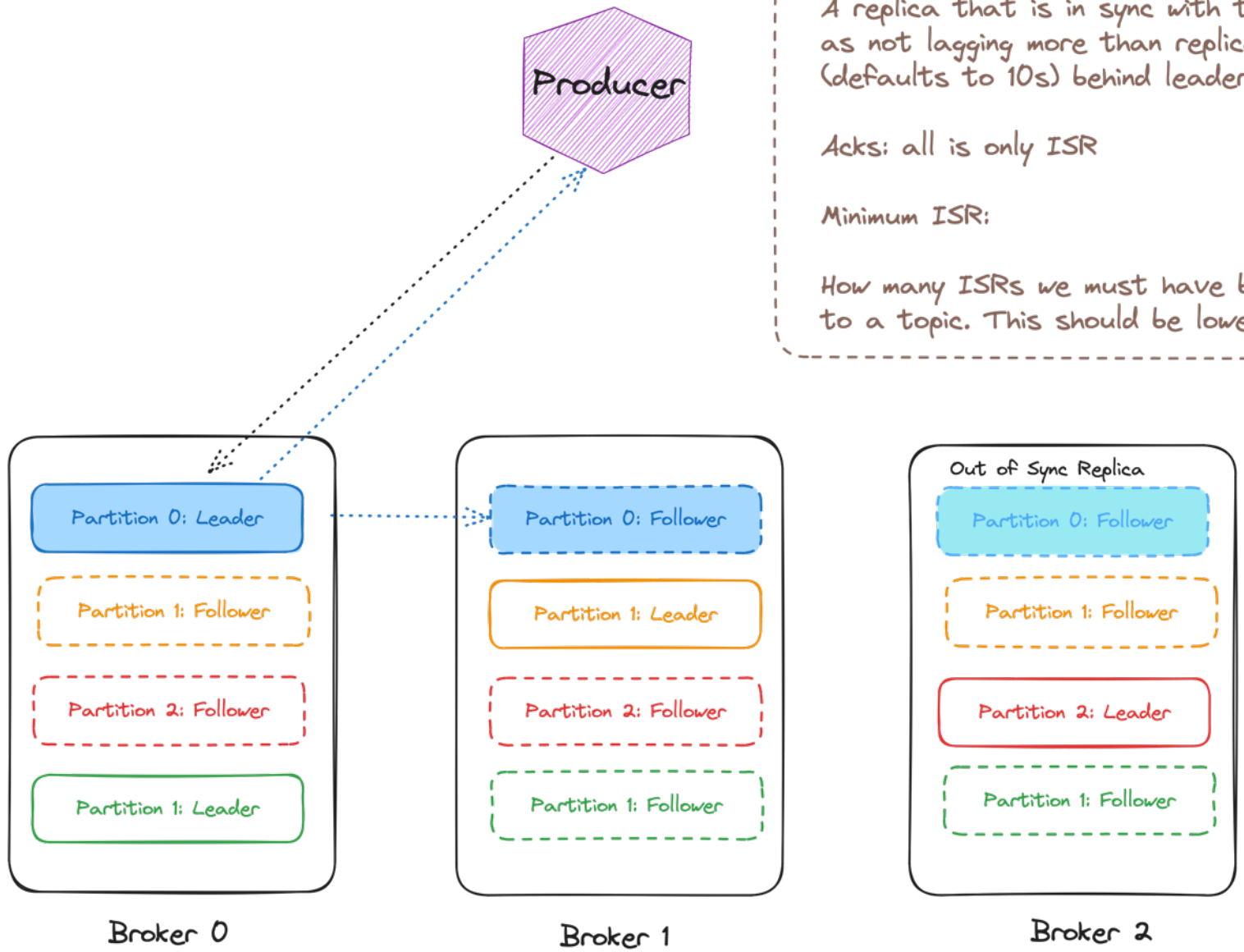
In-Sync Replica (ISR):

A replica that is in sync with the leader defined as not lagging more than replica.lag.time.max.ms (defaults to 10s) behind leader.

Acks: all is only ISR

Minimum ISR:

How many ISRs we must have before we can publish to a topic. This should be lower than nodes in cluster



Reliable Producers

Reliability

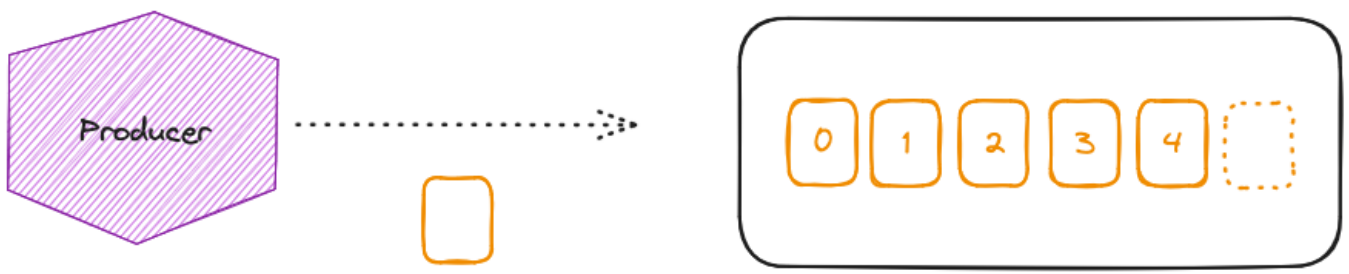
How many brokers need to confirm; alternative is leader or none

How many times to retry if we fail

How many ms between retries (default is 100ms)

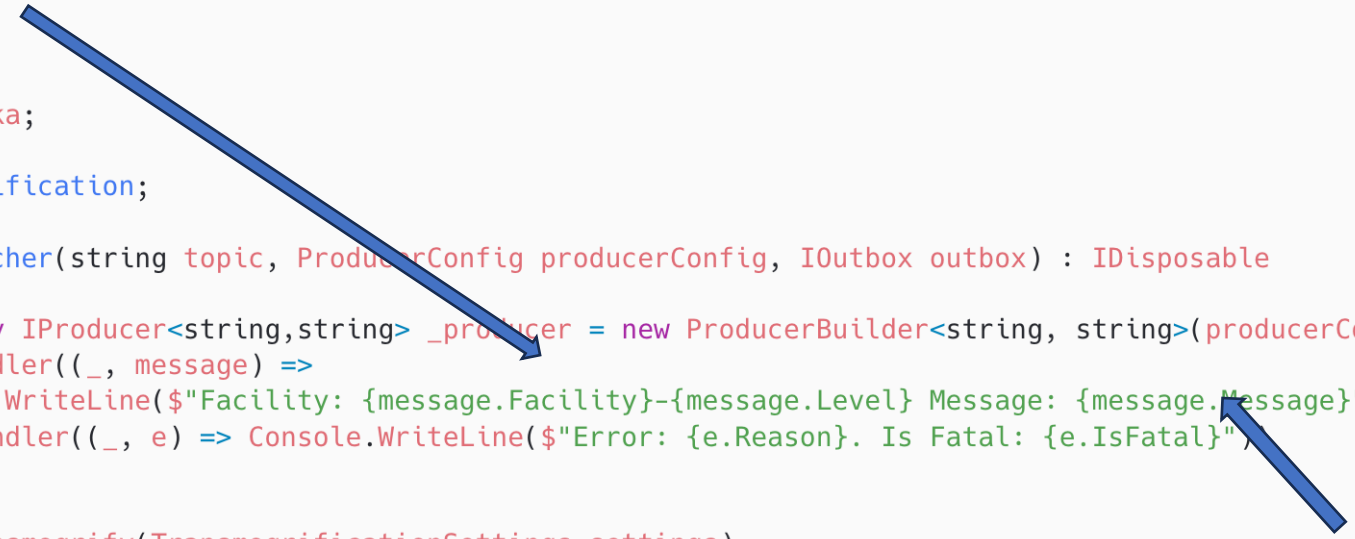
How many requests can be "in flight" Set this to 1 to avoid retries leaving you out of order.

```
acks = all
retries = 3
retry.backoff.ms = 200
max.in.flights.requeests.per.connection = 1
```



Reliably Producing a Message in C#

Configure how the SDK gives us feedback on producing messages



```
using Confluent.Kafka;

namespace Transmogrification;

public class Dispatcher(string topic, ProducerConfig producerConfig, IOutbox outbox) : IDisposable
{
    private readonly IProducer<string,string> _producer = new ProducerBuilder<string, string>(producerConfig)
         .SetLogHandler((_, message) =>
            Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
        .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
        .Build();

    public void Transmogrify(TransmogrificationSettings settings)
    {
        ...
    }

    public void Dispose()
    {
        _producer.Dispose();
    }
}
```

```
var producerConfig = new ProducerConfig
{
    BootstrapServers = "localhost:9092",
    EnableDeliveryReports = true,
    ClientId = Dns.GetHostName(),
    Acks = Acks.Leader,
    //EnableIdempotence = true
    MessageSendMaxRetries = 3,
    MaxInFlight = 1,
};
```

Reliably Producing a Message in C#

```
1   public void Transmogrify(TransmogrificationSettings settings)
2    {
3        try
4        {
5            Action<DeliveryReport<string,string>> handler = report =>
6            {
7                //we use an outbox to be able to retry sending the message if it fails
8                //we don't show a sweeper in this example, but it would be a separate process that would
9                //periodically check the outbox for messages that haven't been persisted yet and retry sending them
10               outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
11           };
12
13           var message = new Message<string, string>
14           {
15               Key = settings.Name,
16               Value = settings.Transformation
17           };
18
19           outbox.Add(topic, message.Key, message);
20
21           //This runs asynchronously - it won't block the thread, instead it will call the handler when the message is sent
22           _producer.Produce(topic, message, handler);
23       }
24       finally
25       {
26           _producer.Flush(TimeSpan.FromSeconds(10));
27       }
28   }
```

Mark the outbox with the status of the produce on callback

Use an “outbox” for “Transactional Messaging” [Simplified]

Don’t block on ack, instead set a callback to assess the delivery report

Reliably Producing a Message in C#

Configure how the SDK gives us feedback on producing messages

Mark the outbox with the status of the produce on callback

Use an outbox for Transactional Messaging

Don't block on ack, instead set a callback to assess the delivery report

```
var producerConfig = new ProducerConfig
{
    BootstrapServers = "localhost:9092",
    EnableDeliveryReports = true,
    ClientId = Dns.GetHostName(),
    Acks = Acks.Leader,
    //EnableIdempotence = true
    MessageSendMaxRetries = 3,
    MaxInFlight = 1,
};
```

```
using Confluent.Kafka;

namespace Transmogrification;

public class Dispatcher(string topic, ProducerConfig producerConfig, IOutbox outbox) : IDisposable
{
    private readonly IProducer<string,string> _producer = new ProducerBuilder<string, string>(producerConfig)
         .SetLogHandler((_, message) =>
            Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
        .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
        .Build();

    public void Transmogrify(TransmogrificationSettings settings)
    {
        try
        {
            Action<DeliveryReport<string,string>> handler = report =>
            {
                outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
            };

            var message = new Message<string, string>
            {
                Key = settings.Name,
                Value = settings.Transformation
            };

            outbox.Add(topic, message.Key, message);

            _producer.Produce(topic, message, handler);
        }
        finally
        {
            _producer.Flush(TimeSpan.FromSeconds(10));
        }
    }

    public void Dispose()
    {
        _producer.Dispose();
    }

```

Async Reliably Producing a Message in C#

```
using Confluent.Kafka;

namespace Transmogrification;

public class AsyncDispatcher(string topic, ProducerConfig producerConfig, IOutbox outbox) : IDisposable
{
    private readonly IProducer<string,string> _producer = new ProducerBuilder<string, string>(producerConfig)
         .SetLogHandler((_, message) =>
            Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
        .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
        .Build();

    public async Task Transmogrify(TransmogrificationSettings settings, CancellationToken cancellationToken = default)
    {
        try
        {
            var message = new Message<string, string>
            {
                Key = settings.Name,
                Value = settings.Transformation
            };

            outbox.Add(topic, message.Key, message);

            var report = await _producer.ProduceAsync(topic, message, cancellationToken);

            outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
        }
        finally
        {
            _producer.Flush(TimeSpan.FromSeconds(10));
        }
    }

    public void Dispose()
    {
        _producer.Dispose();
    }
}
```

Instead of a callback using async/await for the delivery report – will not return until deliveryreport is produced, which depends on acks.

In principle, at scale, the throughput of async/await will be lower than using Produce with a callback

Demo Three: Reliable Producer

Calvin and Hobbes – Bill Watterson

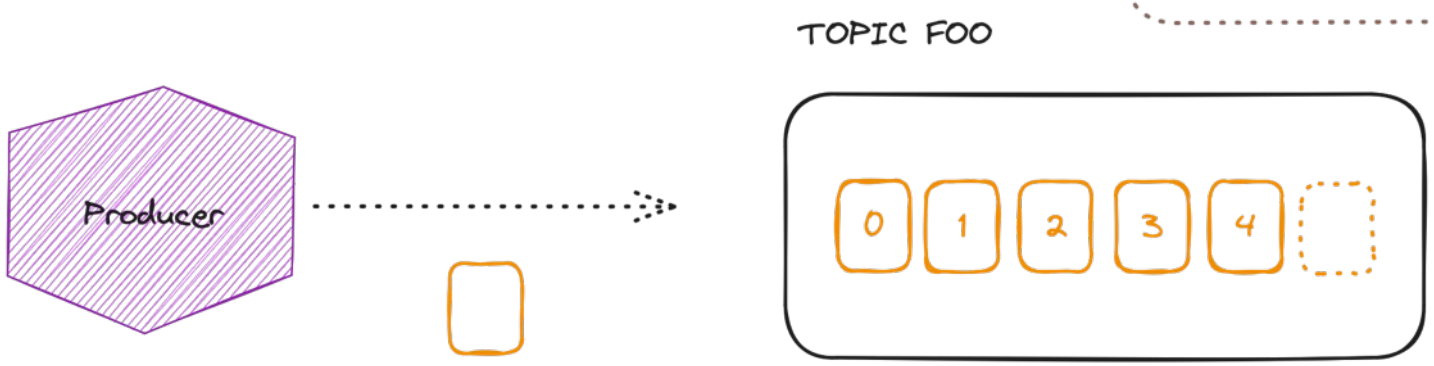
```
using Confluent.Kafka;

namespace Transmogrification;

public class Dispatcher(string topic, ProducerConfig producerConfig, IOutbox outbox) : IDisposable
{
    private readonly IProducer<string,string> _producer = new ProducerBuilder<string, string>(producerConfig)
         .SetLogHandler((_, message) =>
            Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
        .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
        .Build();

    public void Transmogrify(TransmogrificationSettings settings)
    {
        try
        {
            Action<DeliveryReport<string,string>> handler = report =>
            {
                outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
            };

            var message = new Message<string, string>
            {
                Key = settings.Name,
                Value = settings.Transformation
            };

            outbox.Add(topic, message.Key, message);

            _producer.Produce(topic, message, handler);
        }
        finally
        {
            _producer.Flush(TimeSpan.FromSeconds(10));
        }
    }

    public void Dispose()
    {
        _producer.Dispose();
    }

```

Transactional Reliability

We can set up to 5 requests per connection.

```
max.in.flights.requests.per.connection = 5
enable.idempotence = true
transactional.id = prod-1
```

In order to ensure exactly once we need to enable idempotence which rejects already sent messages; if we want ensure ordering with multiple requests we need to set a transaction id for this provider, along with using a transactional producer.



Exactly Once Producing a Message in C#

```
try
{
    _producer.InitTransactions(TimeSpan.FromSeconds(10));
    _producer.BeginTransaction();

    Action<DeliveryReport<string,string>> handler = report =>
    {
        //we use an outbox to be able to retry sending the message if it fails
        //we don't show a sweeper in this example, but it would be a separate process that would
        //periodically check the outbox for messages that haven't been persisted yet and retry sending them
        outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
    };

    var message = new Message<string, string>
    {
        Key = settings.Name,
        Value = settings.Transformation
    };


    //We could send multiple messages as part of the transaction, but we only send one
    _producer.Produce(topic, message, handler);

    //If the outbox add fails, then the transaction will be aborted and the message won't be sent
    outbox.Add(topic, message.Key, message);

    _producer.CommitTransaction(TimeSpan.FromSeconds(10));
}
catch (ProduceException<string, string> e)
{
    Console.WriteLine($"Delivery failed: {e.Error.Reason}");
    _producer.AbortTransaction(TimeSpan.FromSeconds(10));
}
catch (KafkaException e)
{
    Console.WriteLine($"Fatal error: {e.Error.Reason}");
    _producer.AbortTransaction(TimeSpan.FromSeconds(10));
}
```

Mainly intended for a streaming app that does:
- consume
- process
- produce

Where it is important that we get one produce per offset committed from the stream we are reading

We can use a transaction with begin, commit/abort to ensure atomic-to-all partitions/only once production of a message to Kafka

```
var producerConfig = new ProducerConfig
{
    BootstrapServers = "localhost:9092",
    EnableDeliveryReports = true,
    ClientId = Dns.GetHostName(),
    Acks = Acks.All,
    EnableIdempotence = true,
    TransactionalId = "transmogrification-1",
    TransactionTimeoutMs = 30000,

};
```

Idempotence requires Acks.All, sets MaxInFlight 5, MAX_INT retries

Demo Four: Produce a Message Exactly Once

Calvin and Hobbes – Bill Watterson

```
try
 {
     _producer.InitTransactions(TimeSpan.FromSeconds(10));
     _producer.BeginTransaction();

     Action<DeliveryReport<string,string>> handler = report =>
     {
         //we use an outbox to be able to retry sending the message if it fails
         //we don't show a sweeper in this example, but it would be a separate process that would
         //periodically check the outbox for messages that haven't been persisted yet and retry sending them
         outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
     };

     var message = new Message<string, string>
     {
         Key = settings.Name,
         Value = settings.Transformation
     };


     //We could send multiple messages as part of the transaction, but we only send one
     _producer.Produce(topic, message, handler);

     //If the outbox add fails, then the transaction will be aborted and the message won't be sent
     outbox.Add(topic, message.Key, message);

     _producer.CommitTransaction(TimeSpan.FromSeconds(10));
 }
 catch (ProduceException<string, string> e)
 {
     Console.WriteLine($"Delivery failed: {e.Error.Reason}");
     _producer.AbortTransaction(TimeSpan.FromSeconds(10));
 }
 catch (KafkaException e)
 {
     Console.WriteLine($"Fatal error: {e.Error.Reason}");
    _producer.AbortTransaction(TimeSpan.FromSeconds(10));
 }
```

Reliable Consumers

Auto Store and Commit Offsets in C#

```
try
{
    _consumer.Subscribe(topic);

    while (true)
    {
        var consumeResult = _consumer.Consume(cancellationToken);

        if (consumeResult.IsPartitionEOF)
        {
            await Task.Delay(1000, cancellationToken);
            continue;
        }

        var dataType = translator(consumeResult.Message);
        var result = handler(dataType);
    }
}
```

If we fail to translate, or fail to handle the record, (including a crash) we will have committed the offset and will not re-attempt processing the record

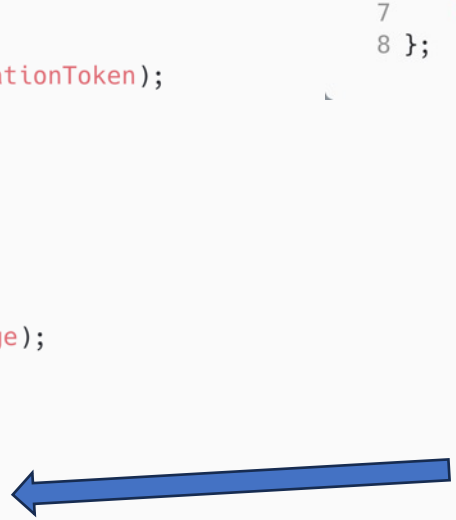
```
var consumerConfig = new ConsumerConfig()
{
    BootstrapServers = "localhost:9092",
    GroupId = "transmogrification-consumer",
    AutoOffsetReset = AutoOffsetReset.Earliest,
    EnableAutoCommit = true,
    //AutoCommitIntervalMs = 5000,
    EnableAutoOffsetStore = true
};
```

Store the offset for a record we read, and commit it every 5s (by default)

Auto Commit, Manually Store Offsets in C#

```
try
{
    _consumer.Subscribe(topic);

    while (true)
    {
        var consumeResult = _consumer.Consume(cancellationToken);

        if (consumeResult.IsPartitionEOF)
        {
            await Task.Delay(1000, cancellationToken);
            continue;
        }

        var dataType = translator(consumeResult.Message);

        var result = handler(dataType);
        if (result.Success)
        {
            _consumer.StoreOffset(consumeResult);
        }
    }
}
```

```
var consumerConfig = new ConsumerConfig()
{
    BootstrapServers = "localhost:9092",
    GroupId = "transmogrification-consumer",
    AutoOffsetReset = AutoOffsetReset.Earliest,
    EnableAutoCommit = true,
    EnableAutoOffsetStore = false
};
```



EnableAutoOffsetStore = false
Manually store offsets to the consumer's cache; autocommit will write on a background thread

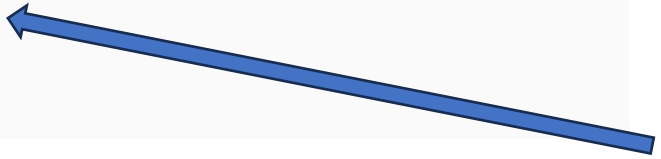
This means we only store processed records (every 5s)

Manually Commit, Manually Store Offsets in C#

```
try
{
    _consumer.Subscribe(topic);

    while (true)
    {
        var consumeResult = _consumer.Consume(cancellationToken);

        if (consumeResult.IsPartitionEOF)
        {
            await Task.Delay(1000, cancellationToken);
            continue;
        }

        var dataType = translator(consumeResult.Message);

        var result = handler(dataType);
        if (result.Success)
        {
            //We don't want to commit unless we have successfully handled the message
            //Commit directly. Normally we would want to batch these up, but for the demo we will

            //commit after each message
            _consumer.Commit(consumeResult);
        }
    }
}
```

```
var consumerConfig = new ConsumerConfig()
{
    BootstrapServers = "localhost:9092",
    GroupId = "transmogrification-consumer",
    AutoOffsetReset = AutoOffsetReset.Earliest,
    EnableAutoCommit = false,
    EnableAutoOffsetStore = false
};
```

EnableAutoOffsetStore = false
EnableAutoCommit = false
We choose to manually store the commits



Manually store commits with:
_consumer.Commit(consumeResult)

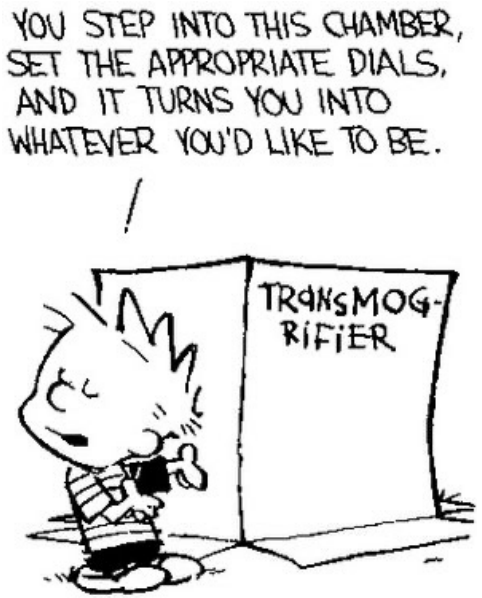
Partition Revoke and Offsets in C#

```
1      private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig)
2          .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}"))
3          .SetLogHandler((_, lm) => Console.WriteLine($"Facility: {lm.Facility} Level: {lm.Level} Log: {lm.Message}"))
4          .SetPartitionsRevokedHandler((c, partions) => partions.ForEach(p => c.StoreOffset(p)))
5          .Build();
```

Our strategy when a partition is revoked (a new consumer joins the group) must match the strategy we use when we have that partition, and ensure outstanding offsets will be committed.

```
1   private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig)
2          .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}"))
3          .SetLogHandler((_, lm) => Console.WriteLine($"Facility: {lm.Facility} Level: {lm.Level} Log: {lm.Message}"))
4          .SetPartitionsRevokedHandler((c, partitions) => c.Commit(partitions))
5          .Build();
```

Demo Five: Reliably Consume A Message



Calvin and Hobbes – Bill Watterson

```
using Confluent.Kafka;
using Spectre.Console;

namespace Transmogrifier;

public record HandleResult(bool Success);

public class MessagePump(string topic, ConsumerConfig consumerConfig)
{
    private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig)
        .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}"))
        .SetLogHandler((_, lm) => Console.WriteLine($"Facility: {lm.Facility} Level: {lm.Level} Log: {lm.Message}"))
        .SetPartitionsRevokedHandler((c, partions) => partions.ForEach(p => c.StoreOffset(p)))
        .Build();

    public async Task Run<TDataType>(
        Func<Message<string, string>, TDataType> translator,
        Func<TDataType, HandleResult> handler,
        CancellationToken cancellationToken = default)
    {
        try
        {
            _consumer.Subscribe(topic);

            while (true)
            {
                var consumeResult = _consumer.Consume(cancellationToken);

                if (consumeResult.IsPartitionEOF)
                {
                    await Task.Delay(1000, cancellationToken);
                    continue;
                }

                var dataType = translator(consumeResult.Message);

                var result = handler(dataType);
                if (result.Success)
                {
                    _consumer.StoreOffset(consumeResult);
                }
            }
        }
        catch(ConsumeException e)
        {
            AnsiConsole.WriteException(e);
        }
        catch (OperationCanceledException)
        {
            //Pump was cancelled, exit
        }
        finally
        {
            _consumer.Close();
        }
    }
}
```

Offset Reset

Offset Reset in C#

```
var consumerConfig = new ConsumerConfig()
{
    BootstrapServers = "localhost:9092",
    GroupId = "transmogrification-consumer",
    AutoOffsetReset = AutoOffsetReset.Earliest,
    EnableAutoCommit = true,
    EnableAutoOffsetStore = false
};
```

AutoOffsetReset indicates what we should do if there is no offset for this GroupId in the _consumer_offsets topic.

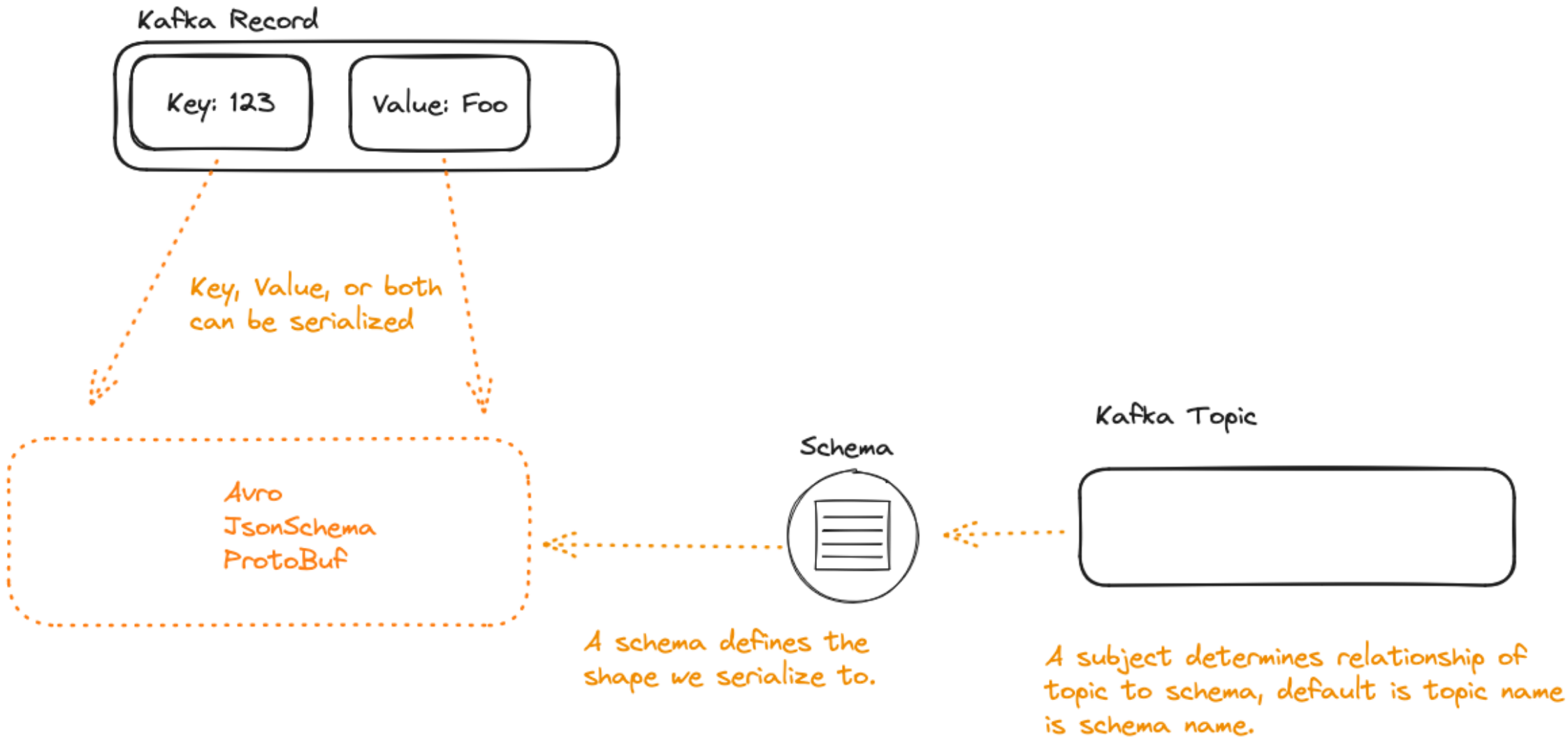
Earliest – read from the earliest offset in the log
Latest – read any new messages appended to the log

Group Id and Offset Reset in C#

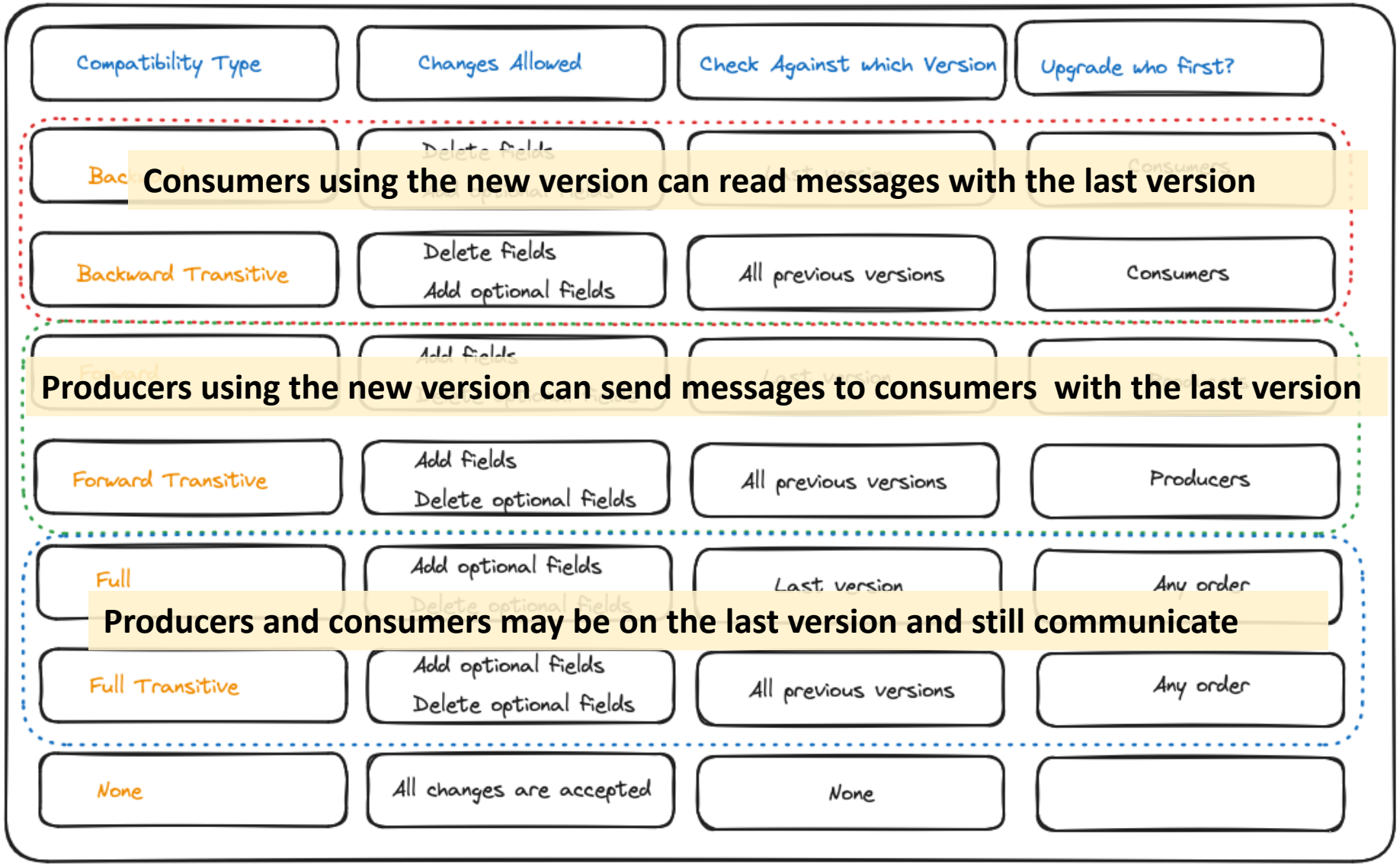
```
1 var consumerConfig = new ConsumerConfig()
2 {
3     BootstrapServers = "localhost:9092",
4     GroupId = "transmogrification-consumer-2",
5     AutoOffsetReset = AutoOffsetReset.Earliest,
6     EnableAutoCommit = true,
7     EnableAutoOffsetStore = false
8 };
```

If you need to reset the offset to the beginning, so as to read the stream again, the easiest option is use a new **GroupId**.

Serializers, De-serializers & Schema Registry


Kafka Record
Key: 123
Value: Foo
Key, Value, or both
can be serialized
Avro
JsonSchema
ProtoBuf
Schema
Kafka Topic
A schema defines the
shape we serialize to.
A subject determines relationship of
topic to schema, default is topic name
is schema name.

Schema Compatibility

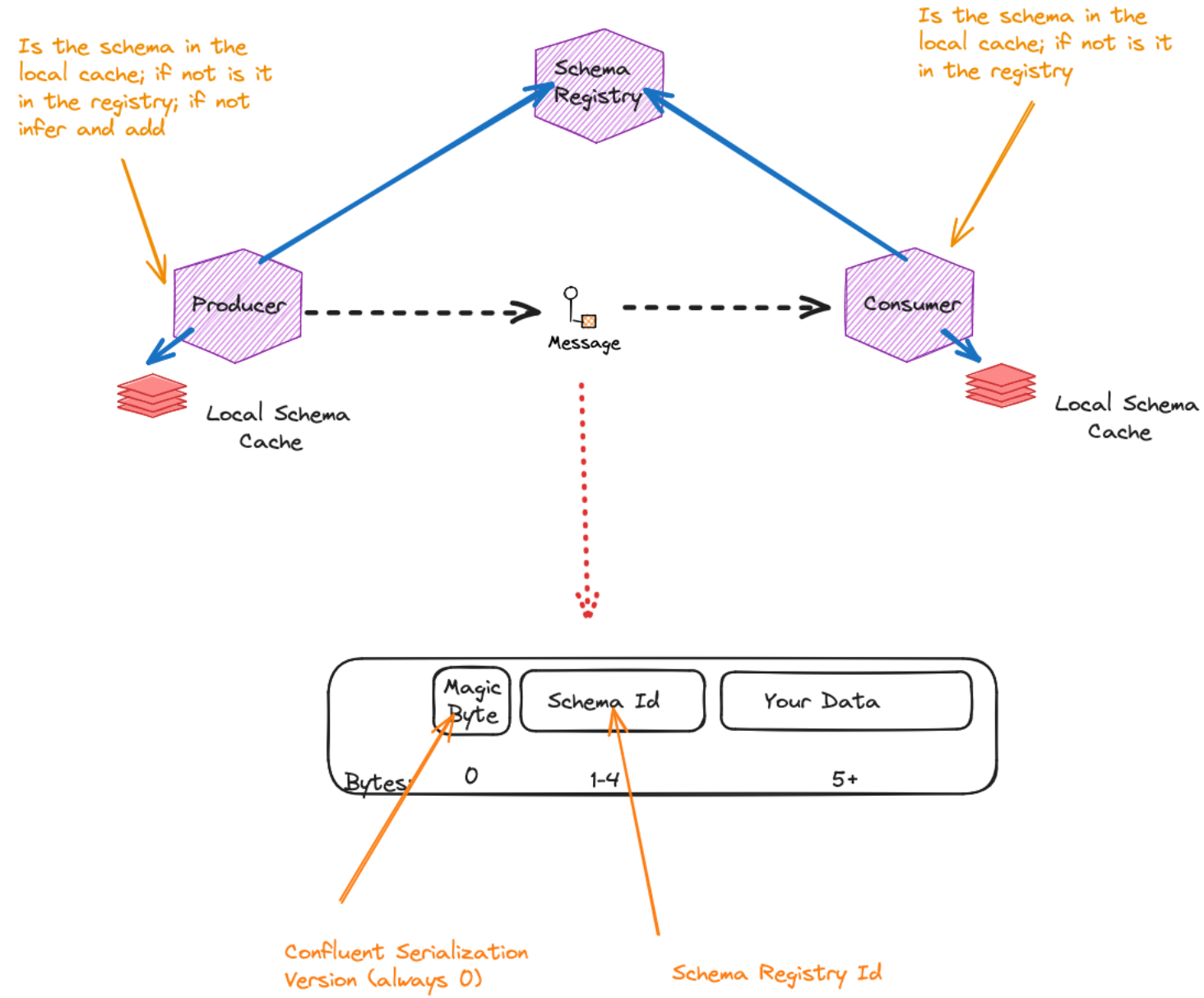


Consumers using the new version can read messages with the last version

Producers using the new version can send messages to consumers with the last version

Producers and consumers may be on the last version and still communicate

Kafka and the Schema Registry



Producing with a Schema Registry in C#

We need to pass in a configuration for the Schema registry

```
public Dispatcher(string topic, ProducerConfig producerConfig, SchemaRegistryConfig schemaRegistryConfig, IOutbox outbox)
    {
        _topic = topic;
        _outbox = outbox;
        _schemaRegistryClient = new CachedSchemaRegistryClient(schemaRegistryConfig);
        var serializerConfig = new JsonSerializerConfig { BufferBytes = 100 };

        _producer = new ProducerBuilder<string, TransmogrificationSettings>(producerConfig)
            .SetLogHandler((_, message) =>
                Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
            .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
            .SetValueSerializer(new JsonSerializer<TransmogrificationSettings>(_schemaRegistryClient, serializerConfig).AsSyncOverAsync())
            .Build();
    }
```

We use the Confluent Serdes Serializer that passes in the schema registry. It will register our schema if none exists.

Note that it uses NJSonSchema for markup and Newtonsoft for JSON serialization, under the hood

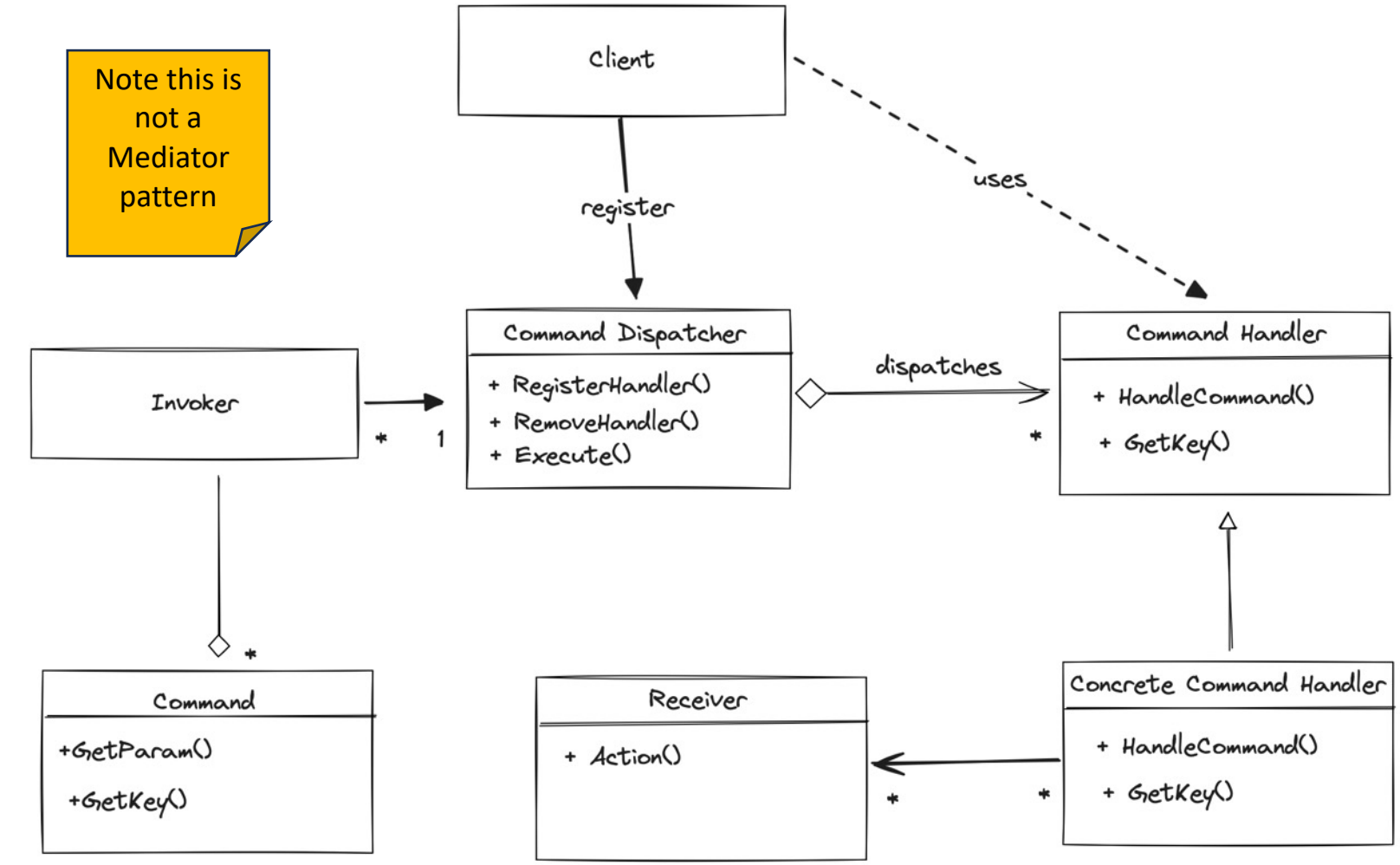
Demo Six: Schema Registry in C#

Calvin and Hobbes – Bill Watterson

```
public Dispatcher(string topic, ProducerConfig producerConfig, SchemaRegistryConfig schemaRegistryConfig, IOutbox outbox)
    {
        _topic = topic;
        _outbox = outbox;
        _schemaRegistryClient = new CachedSchemaRegistryClient(schemaRegistryConfig);
        var serializerConfig = new JsonSerializerConfig { BufferBytes = 100 };

        _producer = new ProducerBuilder<string, TransmogrificationSettings>(producerConfig)
            .SetLogHandler((_, message) =>
                Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
            .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
            .SetValueSerializer(new JsonSerializer<TransmogrificationSettings>(_schemaRegistryClient, serializerConfig).AsSyncOverAsync())
            .Build();
    }
```

Brighter

Command Dispatcher

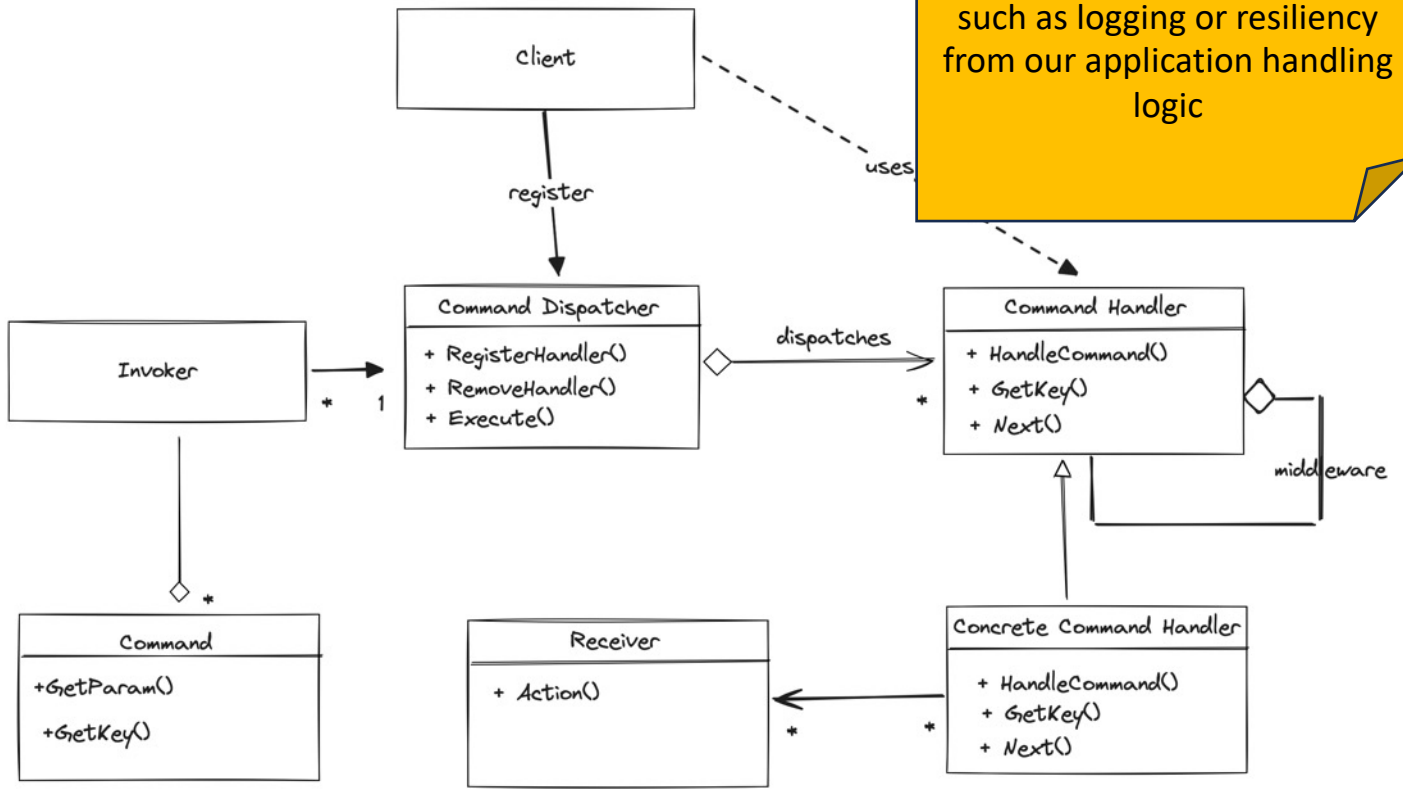


Class Diagram for the Command Dispatcher Pattern.

https://hillside.net/plop/plop2001/accepted_submissions/PLoP2001/bdupireandebfernandez0/PLoP2001_bdupireandebfernandez0_1.pdf

Middleware

We can build a chain of handlers, to allow us to separate orthogonal concerns such as logging or resiliency from our application handling logic



Class Diagram for the Command Dispatcher Pattern showing Middleware.

https://hillside.net/plop/plop2001/accepted_submissions/PLoP2001/bdupireandebfernandez0/PLoP2001_bdupireandebfernandez0_1.pdf

BRIGHTER

Brighter is about Requests
A *Request* is a message sent over a bus. A request may update state.

A *Command* is an instruction to execute some behaviour. An *Event* is a notification.

You use the *Command Processor* to separate the sender from the receiver, and to provide middleware functionality like a retry.

Darker is about Queries
A *Query* is a message executed via a bus that returns a *Result*. A query does not update state.

You use the *Query Processor* to separate the requester from the replier, and to provide middleware functionality like a retry.

Middleware
Both Brighter and Darker allow you to provide middleware that runs between a request or query being made and being handled. The middleware used by a handler is configured by attributes.

Commands and Queries

```
1 [Route("new")]
2 [HttpPost]
3 public async Task<ActionResult<FindPersonResult>> Post(NewPerson newPerson)
4 {
5     await commandProcessor.SendAsync(new AddPerson(newPerson.Name));
6
7     var addedPerson = await queryProcessor.ExecuteAsync<FindPersonResult>(new FindPersonByName(newPerson.Name));
8
9     if (addedPerson == null) return new NotFoundResult();
10
11    return Ok(addedPerson);
12 }
```

The Command Processor is for requests:

- Send => Command, one target
- Publish => Event, zero to many targets

We don't return a value (strict CQS)

The Query Processor is for queries: ExecuteAsync has both a query type for the parameters and a result type for the response.

(We return a value, but don't use to mutate (strict CQS)

Handling Commands

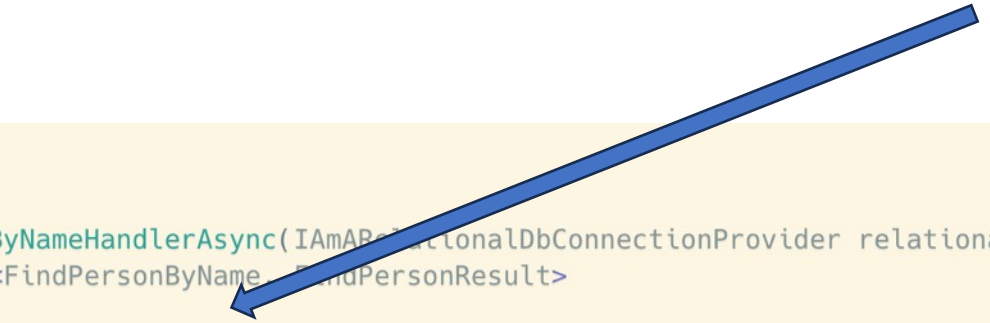
This is the Target Handler, where your domain logic lives. Attributes provide the Middleware Handlers. Here we add logging and Polly resilience policies.

```
1  public class AddPersonHandlerAsync(IAmARelationalDbConnectionProvider relationalDbConnectionProvider)
2      : RequestHandlerAsync<AddPerson>
3  {
4      [RequestLoggingAsync(0, HandlerTiming.Before)]
5      [UsePolicyAsync(step:1, policy: Policies.Retry.EXPONENTIAL_RETRYPOLICYASYNC)]
6      public override async Task<AddPerson> HandleAsync(AddPerson addPerson, CancellationToken cancellationToken = default)
7      {
8          await using var connection = await relationalDbConnectionProvider.GetConnectionAsync(cancellationToken);
9          await connection.ExecuteAsync("insert into Person (Name) values (@Name)", new {Name = addPerson.Name});
10         return await base.HandleAsync(addPerson, cancellationToken);
11     }
12 }
```

We call the next handler in the chain. In principle you can just return from a target handler, because we use a Russian Doll model, so middleware can act “after” as well as before, but we do support middleware that is encapsulated by your target handler.

Handling Queries

This is the Target Handler, where your domain logic lives. Attributes provide the Middleware Handlers. Here we add logging and Polly resilience policies.



```
1  public class FindPersonByNameHandlerAsync(IAmARelationalDbConnectionProvider relationalDbConnectionProvider)
2      : QueryHandlerAsync<FindPersonByName, FindPersonResult>
3  {
4      [QueryLogging(0)]
5      [RetryableQuery(1, Retry.EXPONENTIAL_RETRYPOLICYASYNC)]
6      public override async Task<FindPersonResult> ExecuteAsync(FindPersonByName query, CancellationToken cancellationToken = new
   CancellationToken())
7      {
8          await using var connection = await relationalDbConnectionProvider .GetConnectionAsync(cancellationToken);
9          var people = await connection.QueryAsync<Person>("select * from Person where name = @name", new {name = query.Name});
10         var person = people.SingleOrDefault();
11
12         return new FindPersonResult(person);
13     }
14 }
```

BRIGHTER

Using an External Bus

As well as using an Internal Bus, in Brighter you can use an External Bus - middleware such as RabbitMQ or Kafka - to send a request between processes. Brighter supports both sending a request, and provides a *Dispatcher* than can listen for requests on middleware and forward it to a handler.

Outbox

To ensure that a message is sent if you update your application's state, we need a transaction that can span the update and the message that you want to send. Using an Outbox strategy we write both to the same database, and then transmit the message written to the table. We retry until we succeed.

Outbox Sweeper

To reduce latency we can immediately "clear the outbox". But for guaranteed delivery we need to run a sweeper, in it's own process, that will continually try to send unsent messages.

```
1  public class MakeTransmogrificationHandlerAsync(
2      IAmATransactionConnectionProvider transactionProvider,
3      IAmACommandProcessor postBox,
4      ILogger<MakeTransmogrificationHandlerAsync> logger)
5      : RequestHandlerAsync<MakeTransmogrification>
6  {
7      [RequestLoggingAsync(0, HandlerTiming.Before)]
8      [UsePolicyAsync(step:1, policy: Policies.Retry.EXPONENTIAL_RETRYPOLICYASYNC)]
9      public override async Task<MakeTransmogrification> HandleAsync(MakeTransmogrification makeTransmogrification, CancellationToken
   cancellationToken = default)
10     {
11         var posts = new List<Guid>();
12
13         //We use the unit of work to grab connection and transaction, because Outbox needs
14         //to share them 'behind the scenes'
15
16         var conn = await transactionProvider.GetConnectionAsync(cancellationToken);
17         var tx = await transactionProvider.GetTransactionAsync(cancellationToken);
18         try
19         {
20             //Transactional Messaging
21         }
22         catch (Exception e)
23         {
24             logger.LogError(e, "Exception thrown handling Add Description request");
25             //it went wrong, rollback the entity change and the downstream mes
26             await transactionProvider.RollbackAsync(cancellationToken);
27             return await base.HandleAsync(makeTransmogrification, ca      ellationToken);
28         }
29         finally
30         {
31             transactionProvider.Close();
32         }
33
34         //Send this message v    a transport. We need the ids to send just the messages here, not all outstanding ones.
35         //Alternatively, you can let the Sweeper do this, but at the cost of increased latency
36         await postBox.ClearOutboxAsync(posts, cancellationToken:cancellationToken);
37
38         return await base.HandleAsync(makeTransmogrification, cancellationToken);
39     }
40 }
```

We will need two dependencies:

- The Transaction/Connection provider, lets Brighter participate in your transaction when writing to its Outbox
- Command Processor, lets us transmit a message via an external bus (i.e. a broker)

This is the lowest latency approach to transmitting, it should be supplemented by a sweeper for guaranteed delivery

```
try
{
    var people = await conn.QueryAsync<Person>(
        "select * from Person where name = @name",
        new {name = makeTransmogrification.Name},
        tx
    );
    var person = people.SingleOrDefault();

    if (person != null)
    {
        var transmogrification = new Transmogrification(makeTransmogrification.Description, person);

        //write the added child entity to the Db
        await conn.ExecuteAsync(
            "insert into Transmogrification (Description, Recipient_Id) values (@Description, @RecipientId)",
            new { transmogrification.Description, transmogrification.RecipientId },
            tx);

        //Now write the message we want to send to the Db in the same transaction.
        posts.Add(await postBox.DepositPostAsync(
            new TransmogrificationMade(person, transmogrification),
            transactionProvider,
            cancellationToken: cancellationToken));

        //commit both new greeting and outgoing message
        await transactionProvider.CommitAsync(cancellationToken);
    }
}
```

The database ExecuteAsync and the CommandProcessor DepositPost share the same transaction.

Demo Seven: Brighter

Calvin and Hobbes – Bill Watterson

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Dapper;
using Microsoft.Extensions.Logging;
using Paramore.Brighter;
using Paramore.Brighter.Logging.Attributes;
using Paramore.Brighter.Policies.Attributes;
using TransmogrifierAPI.Application.Ports.Driven;

namespace TransmogrifierAPI.Application.Ports.Driving
{
    public class MakeTransmogrificationHandlerAsync(
        IAmATransactionConnectionProvider transactionProvider,
        IAmACommandProcessor postBox,
        ILogger<MakeTransmogrificationHandlerAsync> logger)
        : RequestHandlerAsync<MakeTransmogrification>
    {
        [RequestLoggingAsync(0, HandlerTiming.Before)]
        [UsePolicyAsync(step:1, policy: Policies.Retry.EXPONENTIAL_RETRYPOLICYASYNC)]
        public override async Task<MakeTransmogrification> HandleAsync(MakeTransmogrification
makeTransmogrification, CancellationToken cancellationToken = default)
        {
            var posts = new List<Guid>();

            //We use the unit of work to grab connection and transaction, because Outbox needs
            //to share them 'behind the scenes'

            var conn = await transactionProvider.GetConnectionAsync(cancellationToken);
            var tx = await transactionProvider.GetTransactionAsync(cancellationToken);
            try
            {
                var people = await conn.QueryAsync<Person>(
                    "select * from Person where name = @name",
                    new {name = makeTransmogrification.Name},
                    tx
                );
                var person = people.SingleOrDefault();

                if (person != null)
                {
                    var transmogrification = new Transmogrification(makeTransmogrification.Description, person);

                    //write the added child entity to the Db
                    await conn.ExecuteAsync(
                        "insert into Transmogrification (Description, Recipient_Id) values (@Description,
@RecipientId)",
                        new { transmogrification.Description, transmogrification.RecipientId },
                        tx);

                    //Now write the message we want to send to the Db in the same transaction.
                    posts.Add(await postBox.DepositPostAsync(
                        new TransmogrificationMade(person, transmogrification),
                        transactionProvider,
                        cancellationToken: cancellationToken));

                    //commit both new greeting and outgoing message
                    await transactionProvider.CommitAsync(cancellationToken);
                }
            }
            catch (Exception e)
            {
                logger.LogError(e, "Exception thrown handling Add Description request");
                //it went wrong, rollback the entity change and the downstream message
                await transactionProvider.RollbackAsync(cancellationToken);
                return await base.HandleAsync(makeTransmogrification, cancellationToken);
            }
            finally
            {
                transactionProvider.Close();
            }

            //Send this message via a transport. We need the ids to send just the messages here, not all outstanding
ones.
            //Alternatively, you can let the Sweeper do this, but at the cost of increased latency
            await postBox.ClearOutboxAsync(posts, cancellationToken:cancellationToken);

            return await base.HandleAsync(makeTransmogrification, cancellationToken);
        }
    }
}

```

Diving Deeper

Links

The Sample Code: https://github.com/iancooper/KafkaForDotNetDevs
Brighter: https://github.com/BrighterCommand/Brighter
This Presentation: https://github.com/iancooper/Presentations

Other Topics

Kafka Connect: Turn a data source into a Kafka stream and vice versa
KSQL: Run queries on Kafka Streams
Streams Applications: Run queries on Kafka Streams in Code (C# Community Support only)
Alternatives to Connect: Flink, Benthos, etc

End