

The background image shows a serene landscape with a calm lake in the foreground, reflecting the surrounding environment. In the middle ground, a dense forest of evergreen trees lines the shore. Behind the forest, a massive mountain range is visible, its slopes covered in various shades of brown, tan, and orange, indicating different rock layers and exposure to sunlight. The sky above is a clear, pale blue.

Streaming Microservices

With Akka Streams and Kafka Streams

Dean Wampler, Ph.D.
dean@lightbend.com
[@deanwampler](https://twitter.com/deanwampler)

Streaming in Context...





Free as in



lightbend.com/fast-data-platform

O'REILLY®

Fast Data Architectures for Streaming Applications

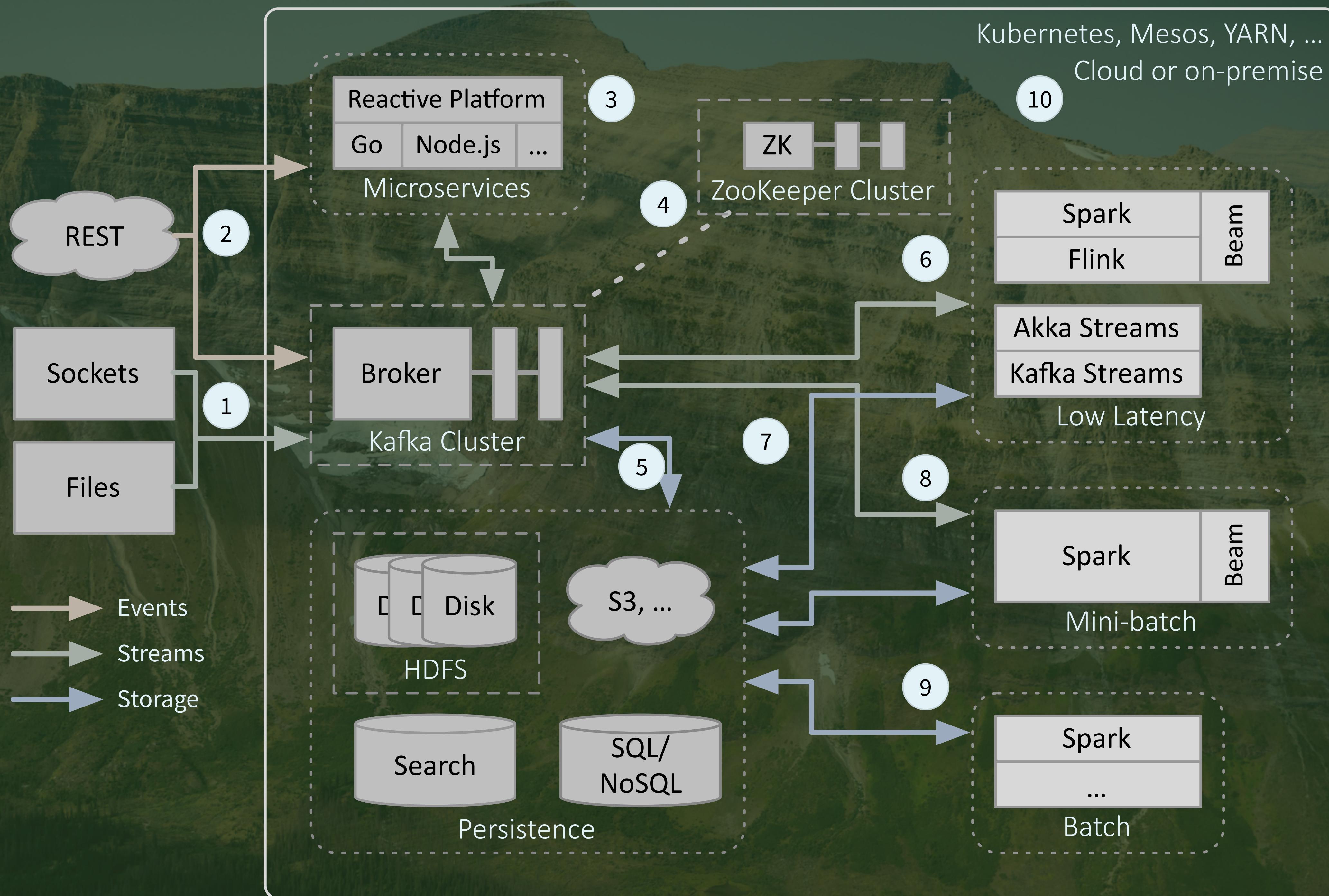
Getting Answers Now from
Data Sets that Never End

Dean Wampler

Compliments of
 Lightbend

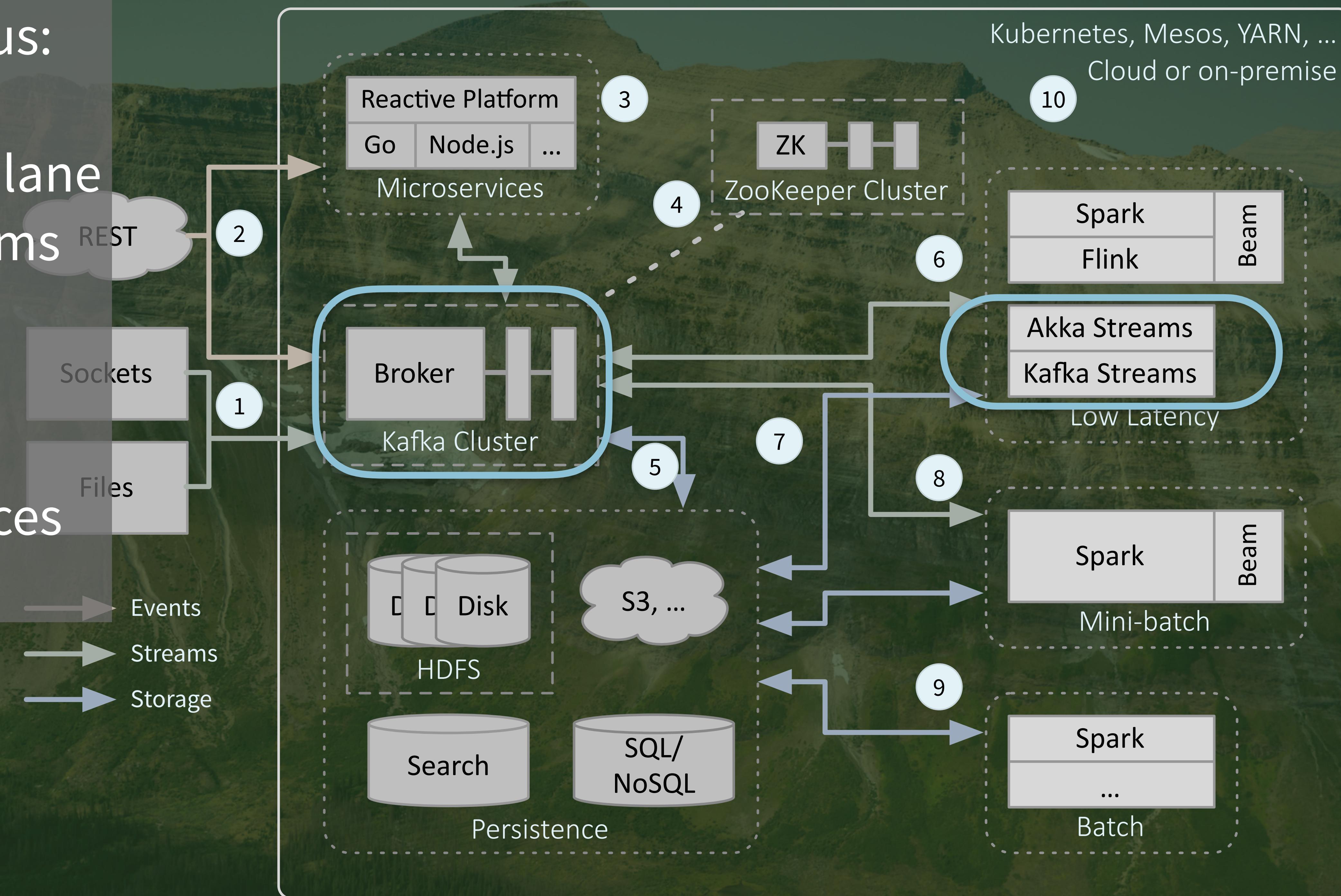


Streaming architectures (from the report)



Today's focus:

- Kafka - the data backplane
- Akka Streams and Kafka Streams - streaming microservices



A wide-angle photograph of the Rocky Mountains under a clear blue sky. In the foreground, a valley is visible with patches of green vegetation and a small body of water. The middle ground shows rugged mountain slopes with exposed rock and sparse vegetation. The background features majestic, snow-capped mountain peaks.

Why Kafka?

Why Kafka?

Reactive Platform

Go Node.js ...

Microservices

3

ZK

ZooKeeper Cluster

4

6

Ak

Ka

8

5

7

Broker

Kafka Cluster

1

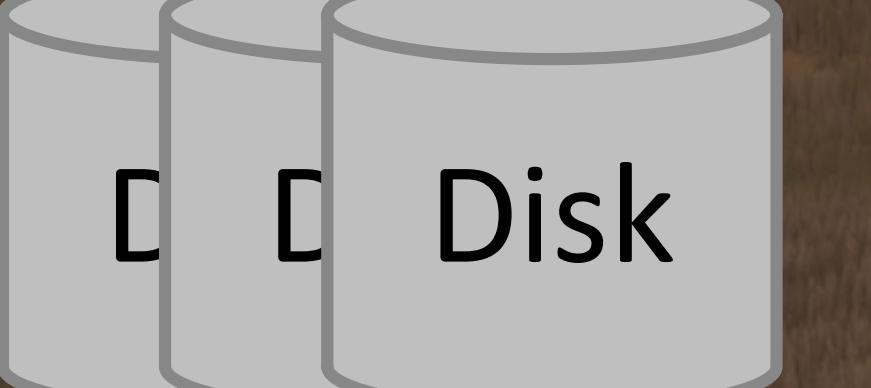
2

Sockets

Files

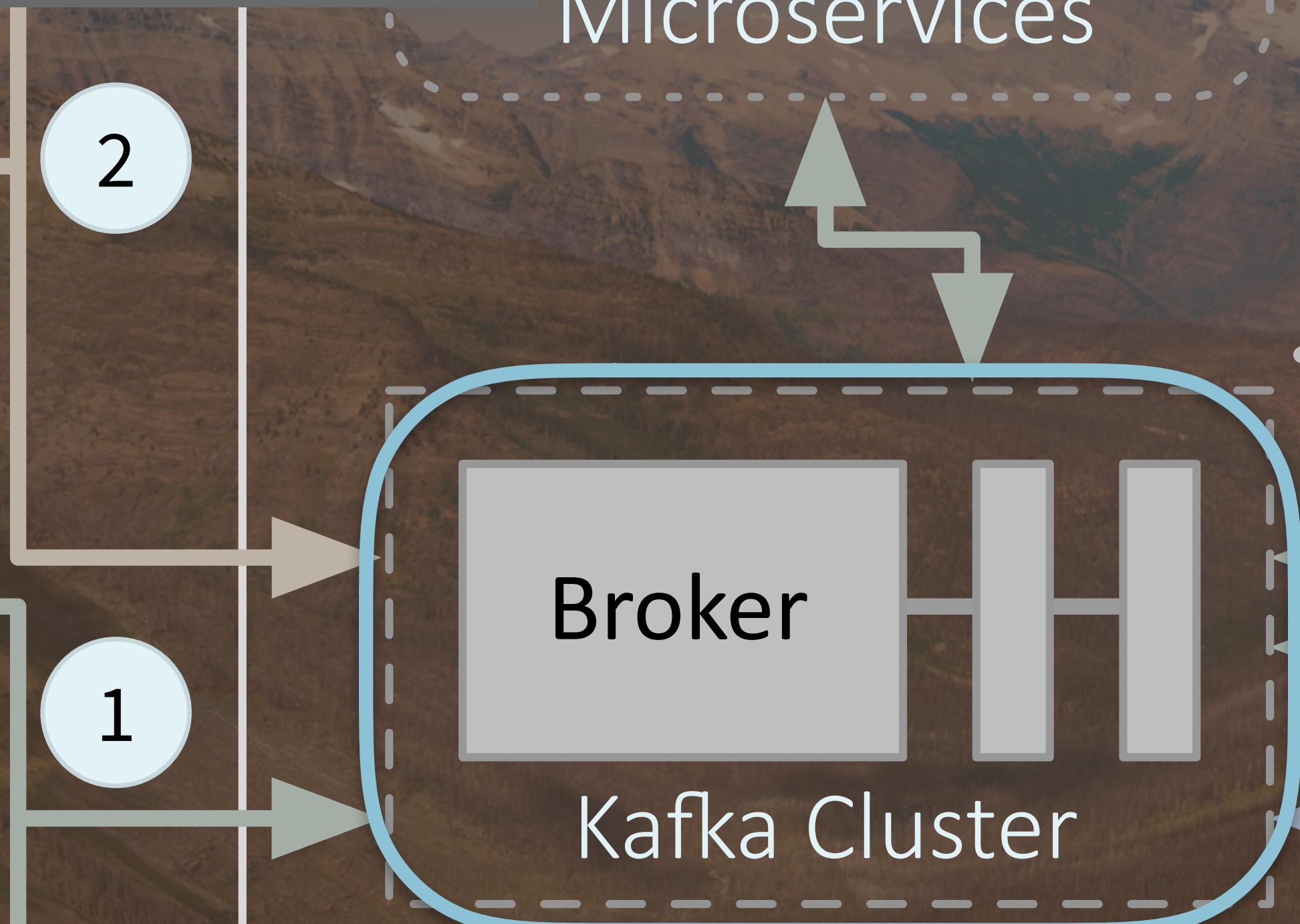
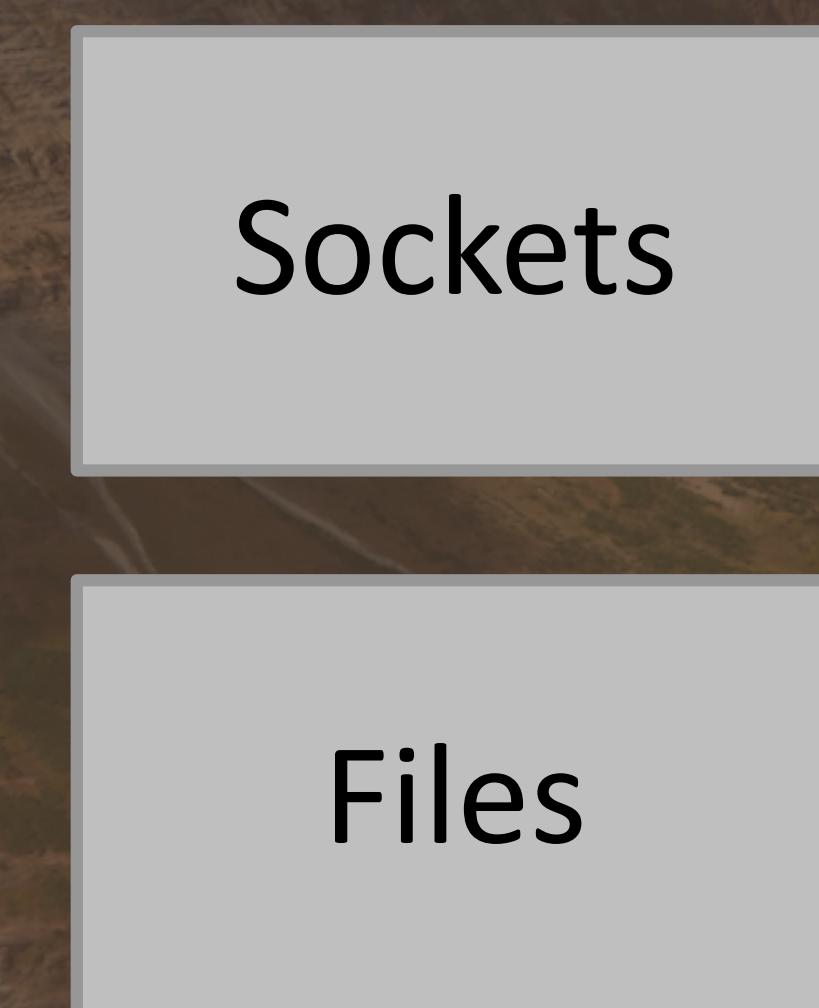
Events

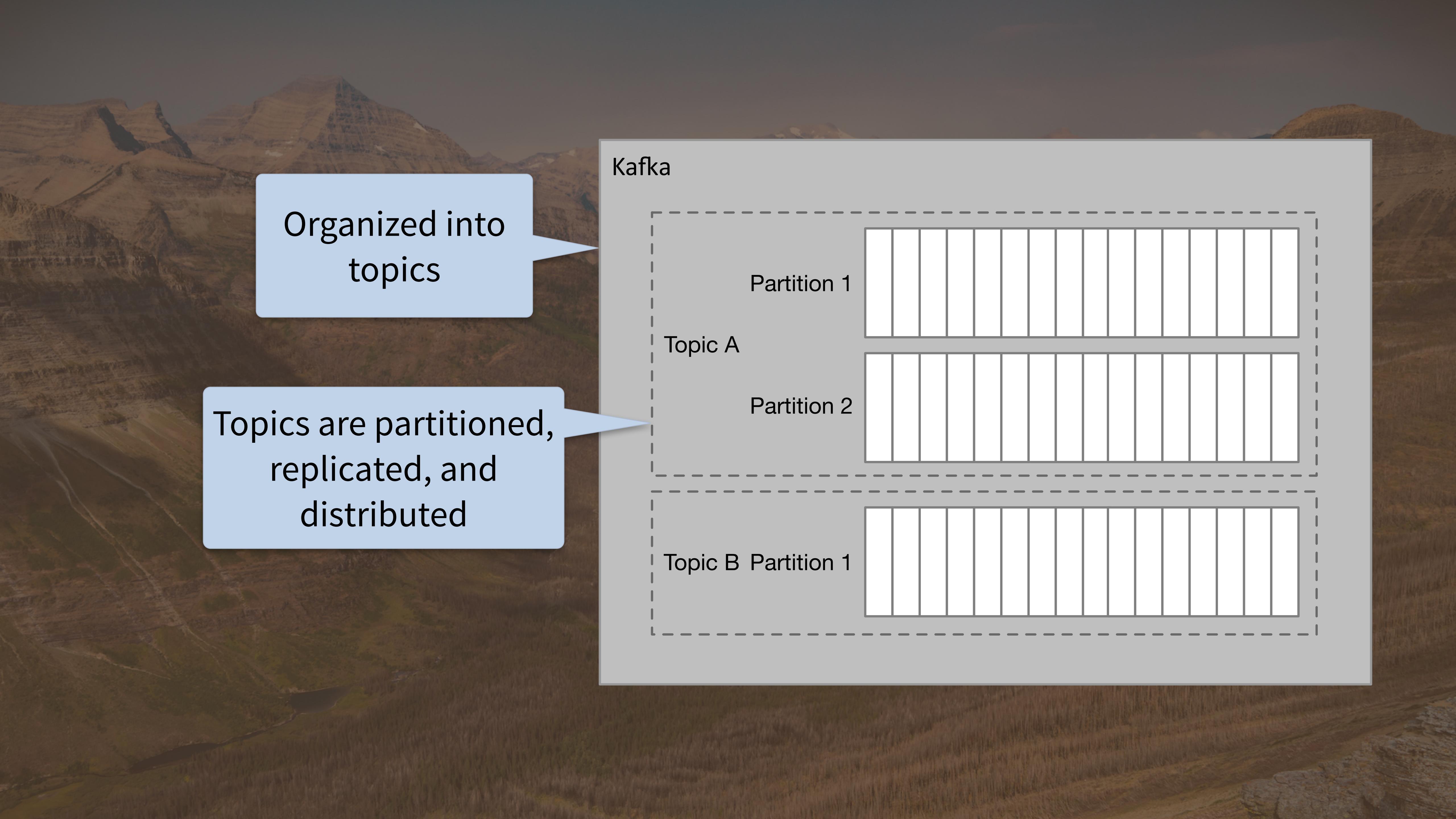
Streams



HDFS

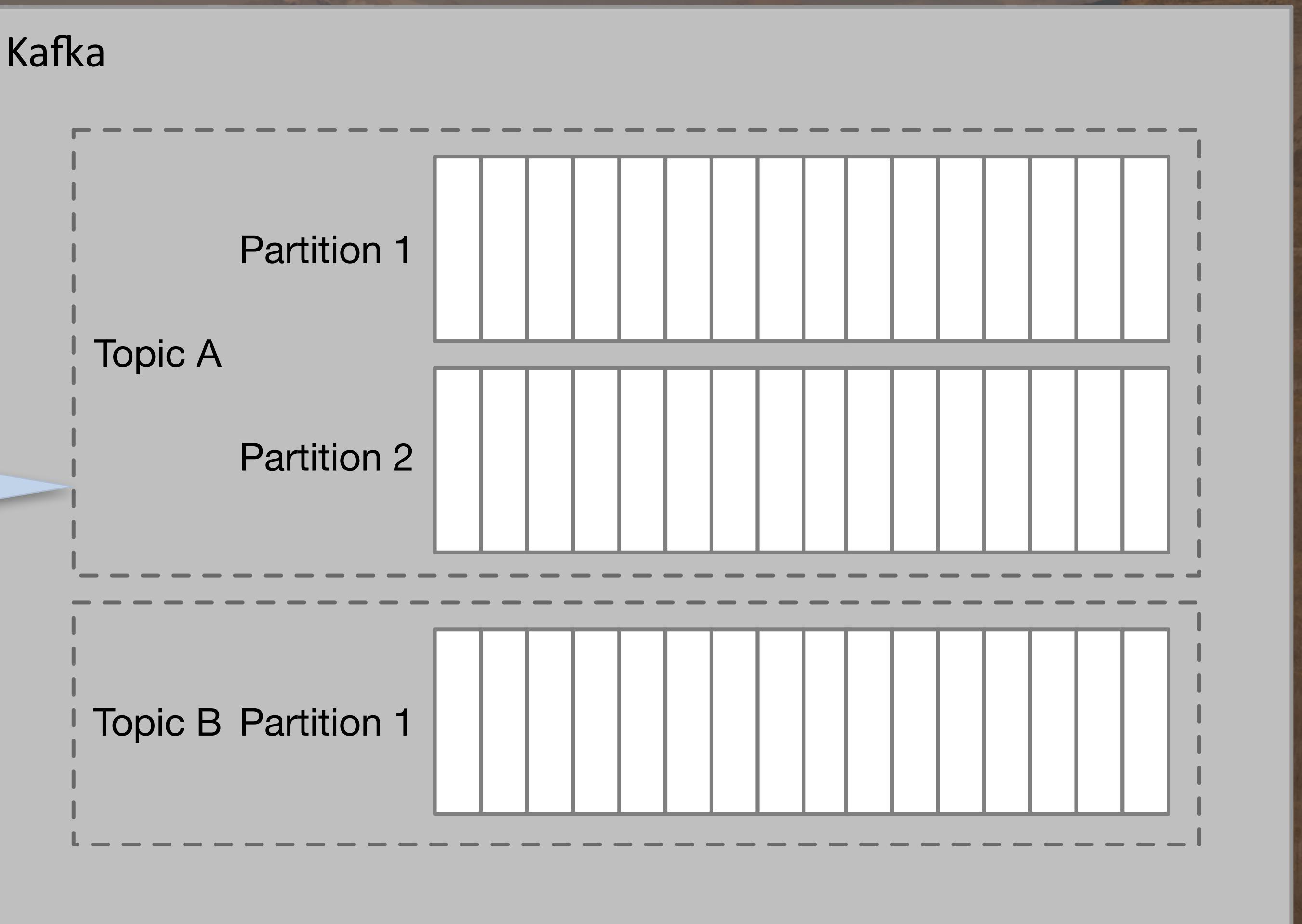
S3, ...



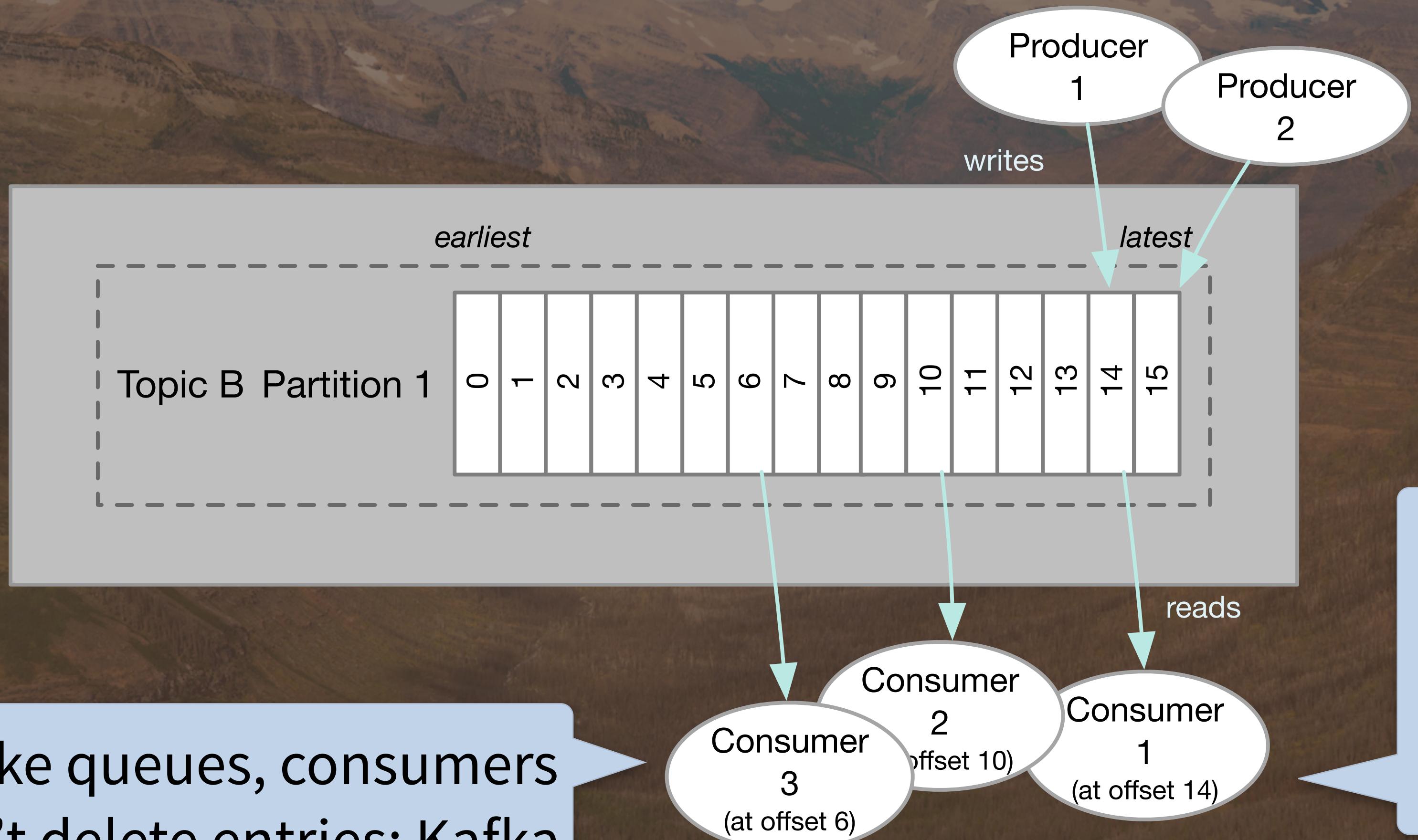


Organized into topics

Topics are partitioned,
replicated, and
distributed



Logs, not queues!

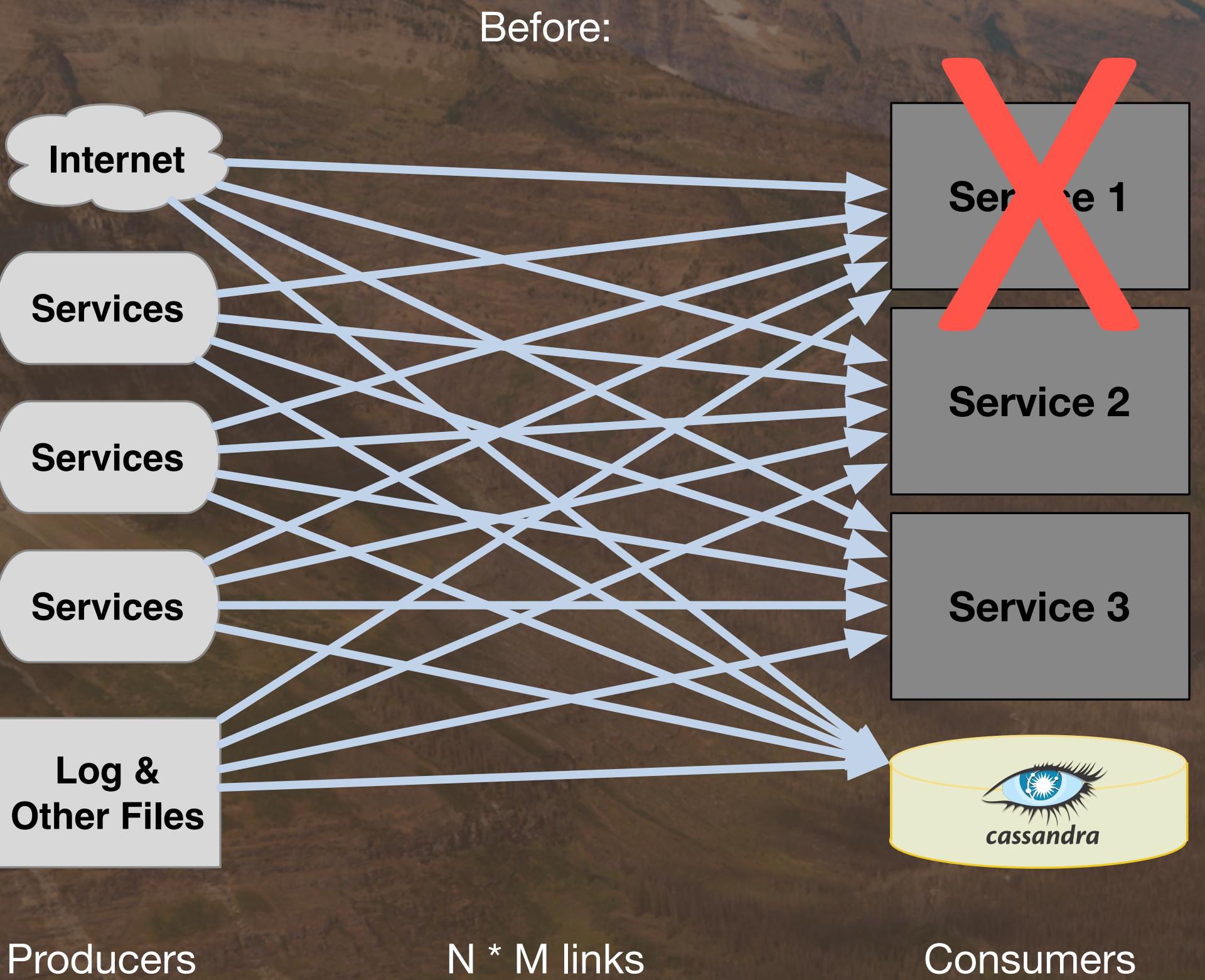


Unlike queues, consumers don't delete entries; Kafka manages their lifecycles

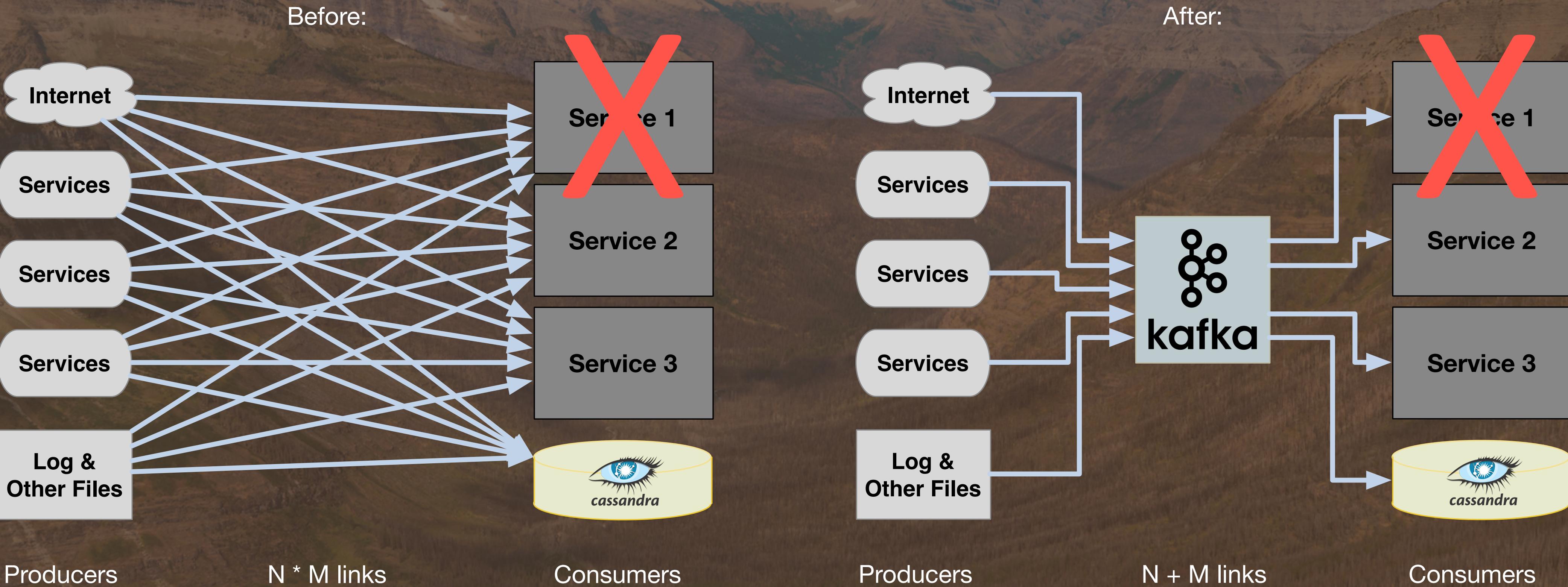
M Producers

N Consumers,
who start
reading where
they want

Kafka for Connectivity

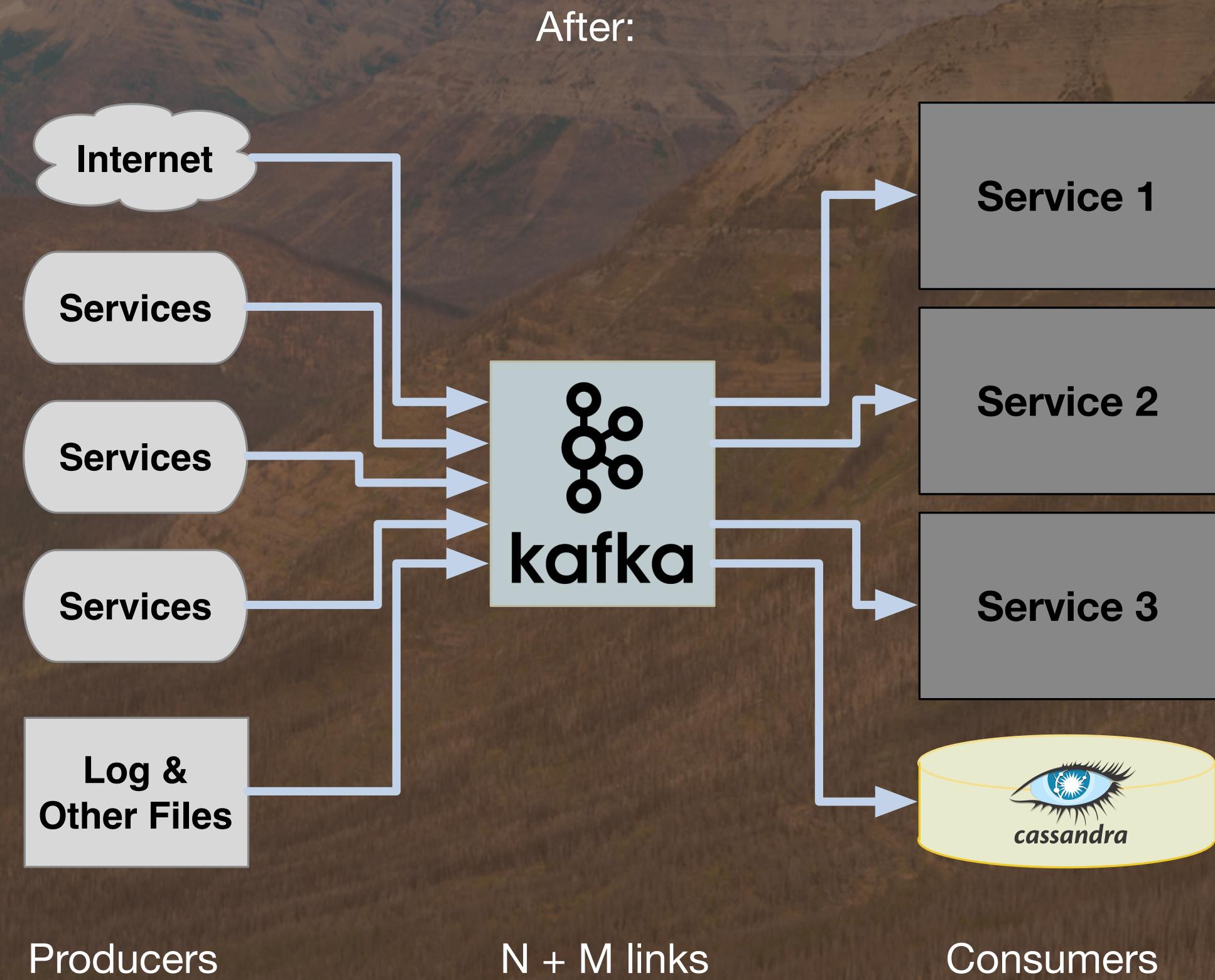


Kafka for Connectivity



Kafka for Connectivity

- Simplify dependencies
- Resilient against data loss
- M producers, N consumers
- Simplicity of one “API” for communication



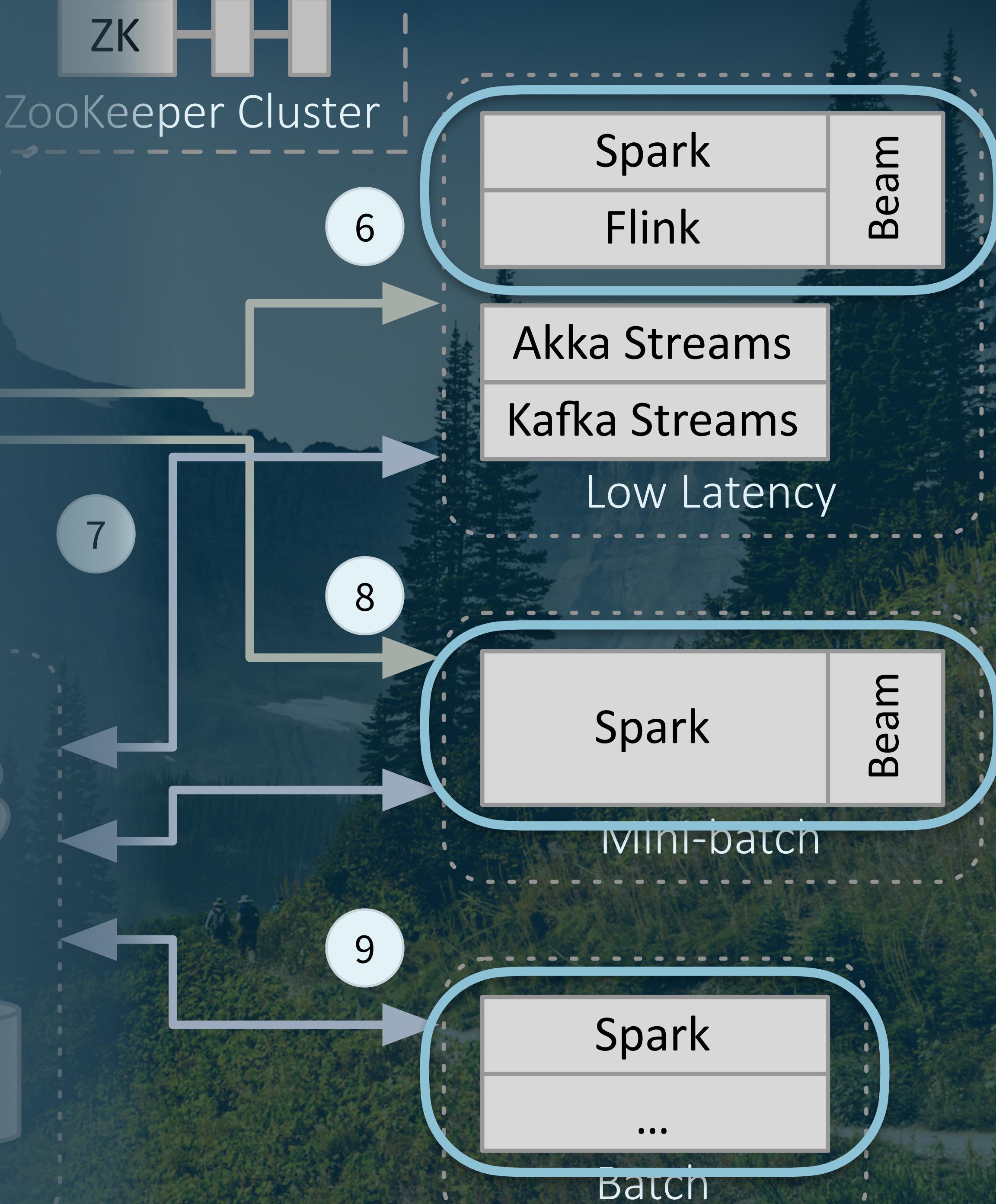
Streaming Engines



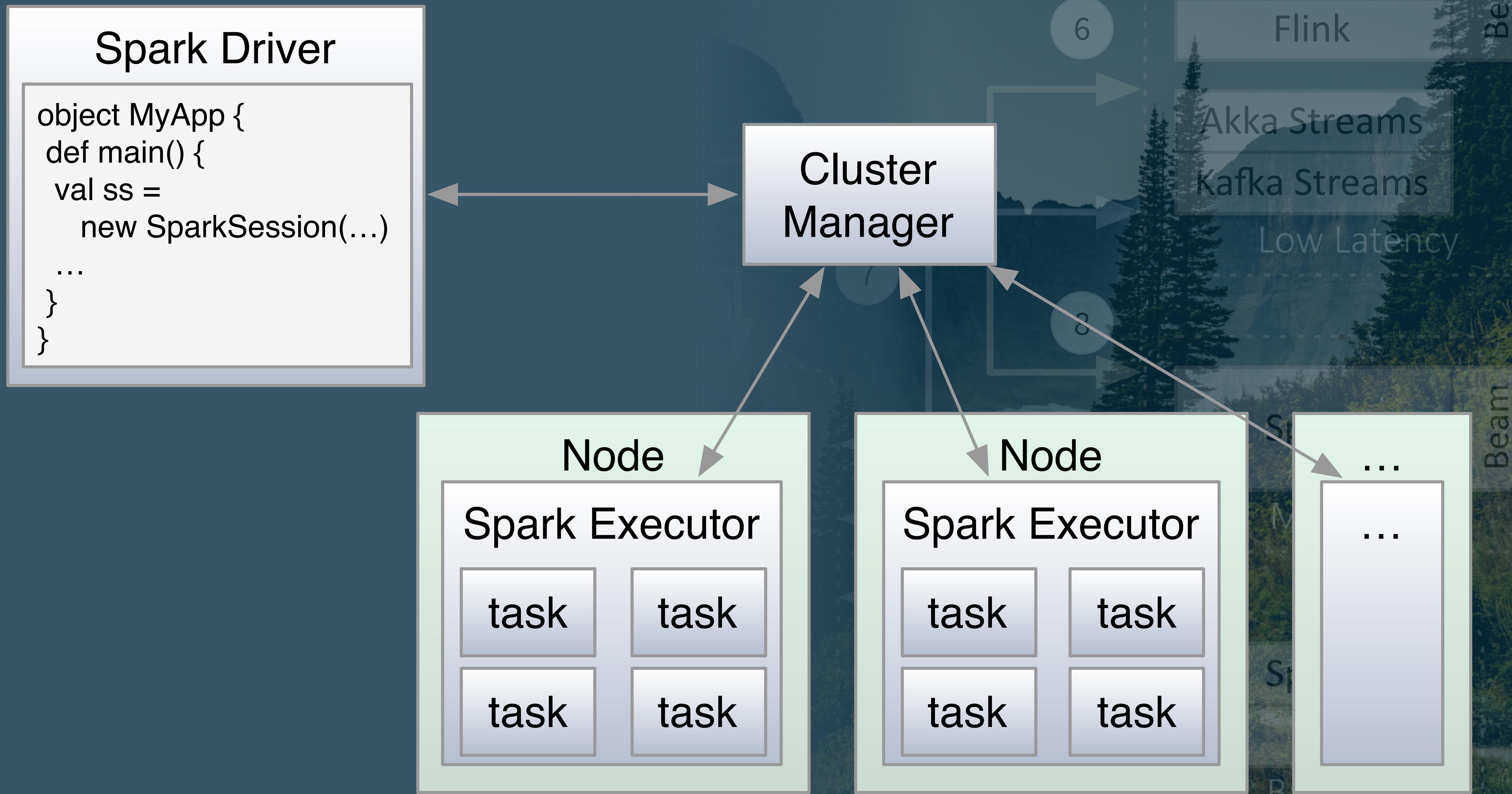
Streaming Engines

Spark, Flink - services to which you submit work. Large scale, automatic data partitioning.

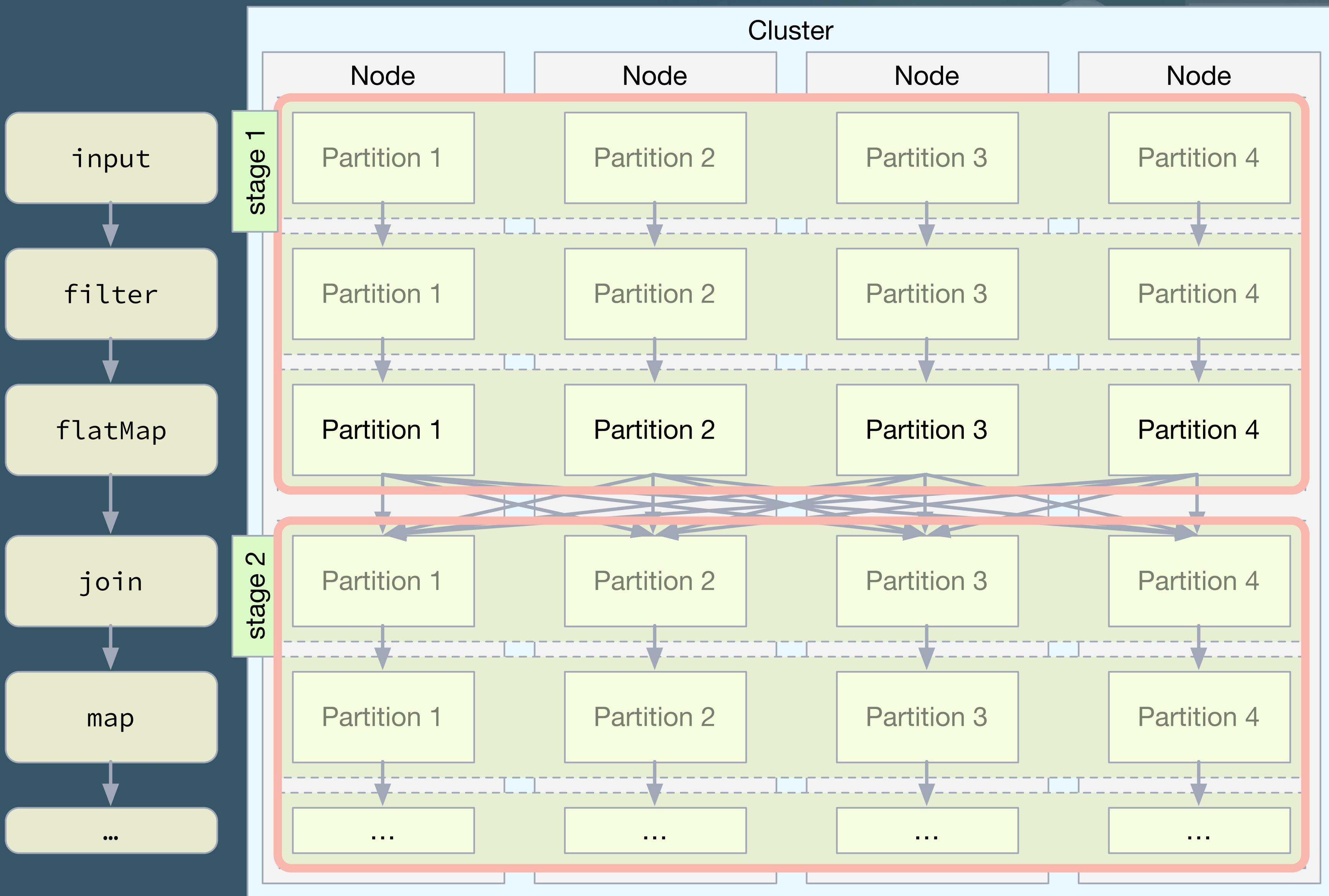
Beam - similar. Google's project that has been instrumental in defining streaming semantics.



They do a *lot* (Spark example)

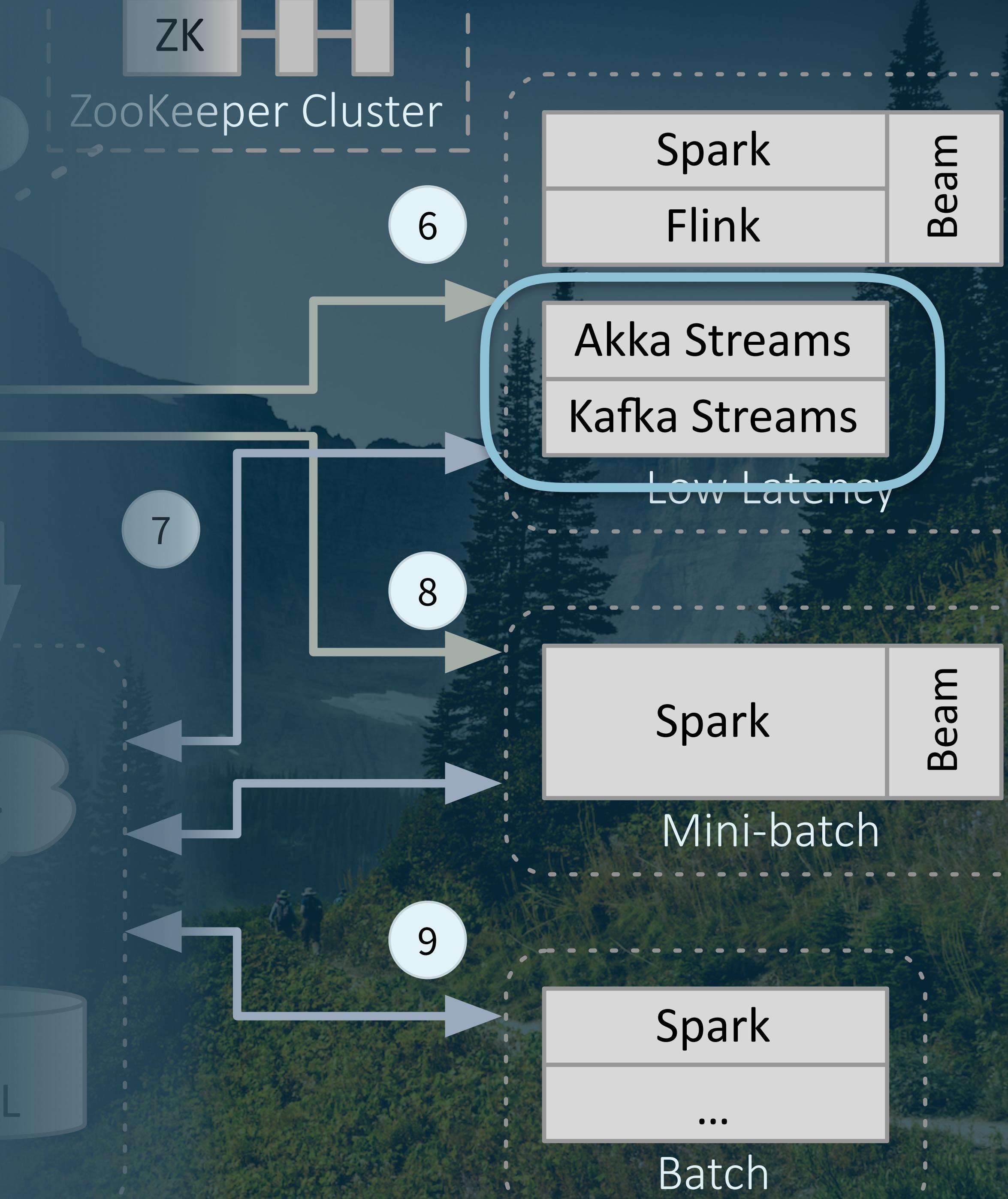


They do a lot (Spark example)



Streaming Engines

Akka Streams, Kafka Streams - libraries for “data-centric microservices”. Smaller scale, but great flexibility



Microservice All the Things!!



Scott Hanselman

@shanselman

Follow

Microservices, for when your in-process methods have too little latency.

Dave Cheney @davecheney

Microservices, for when function calls are too reliable.

4:11 AM - 25 Feb 2018

207 Retweets **566** Likes



25

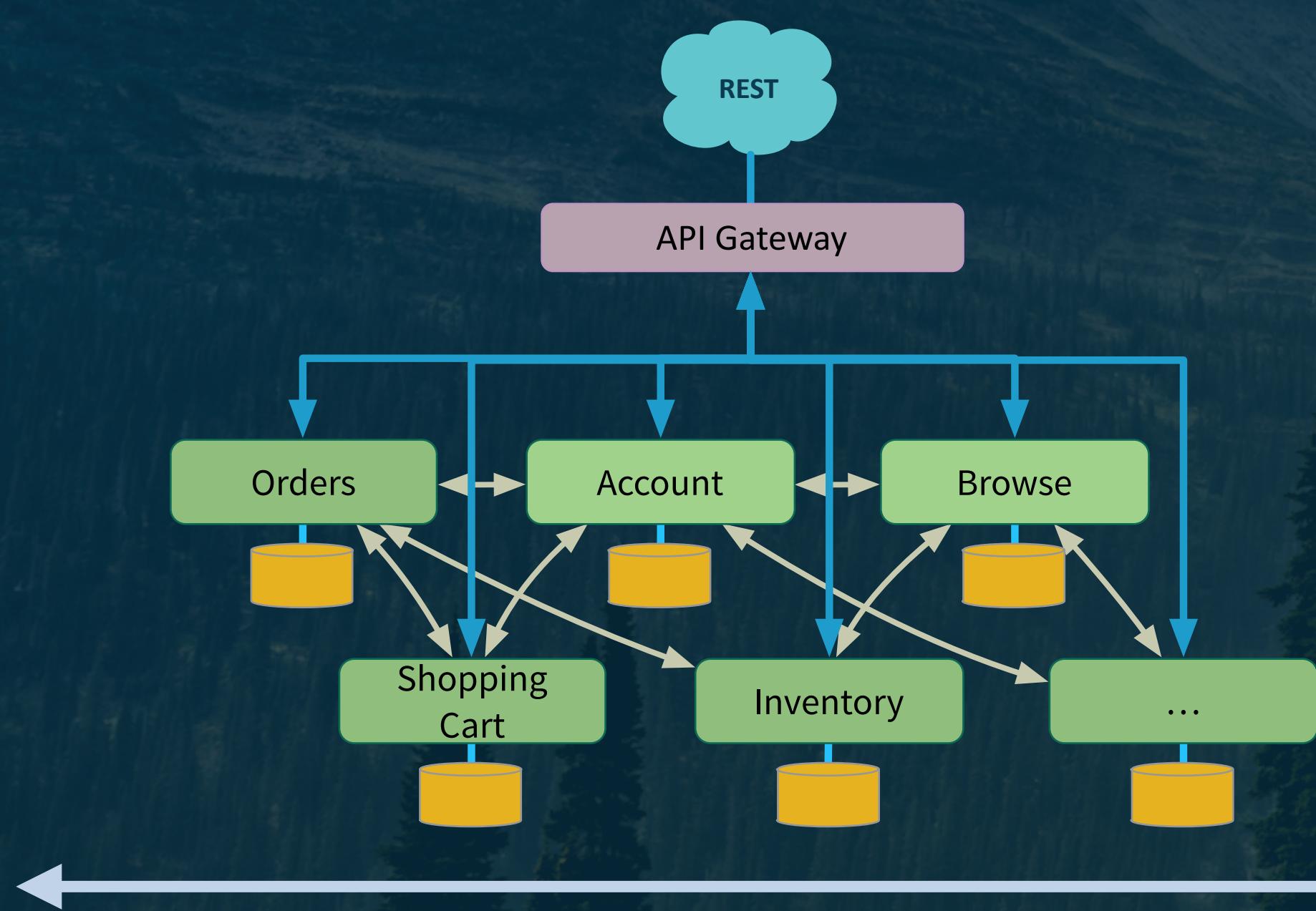
207

566

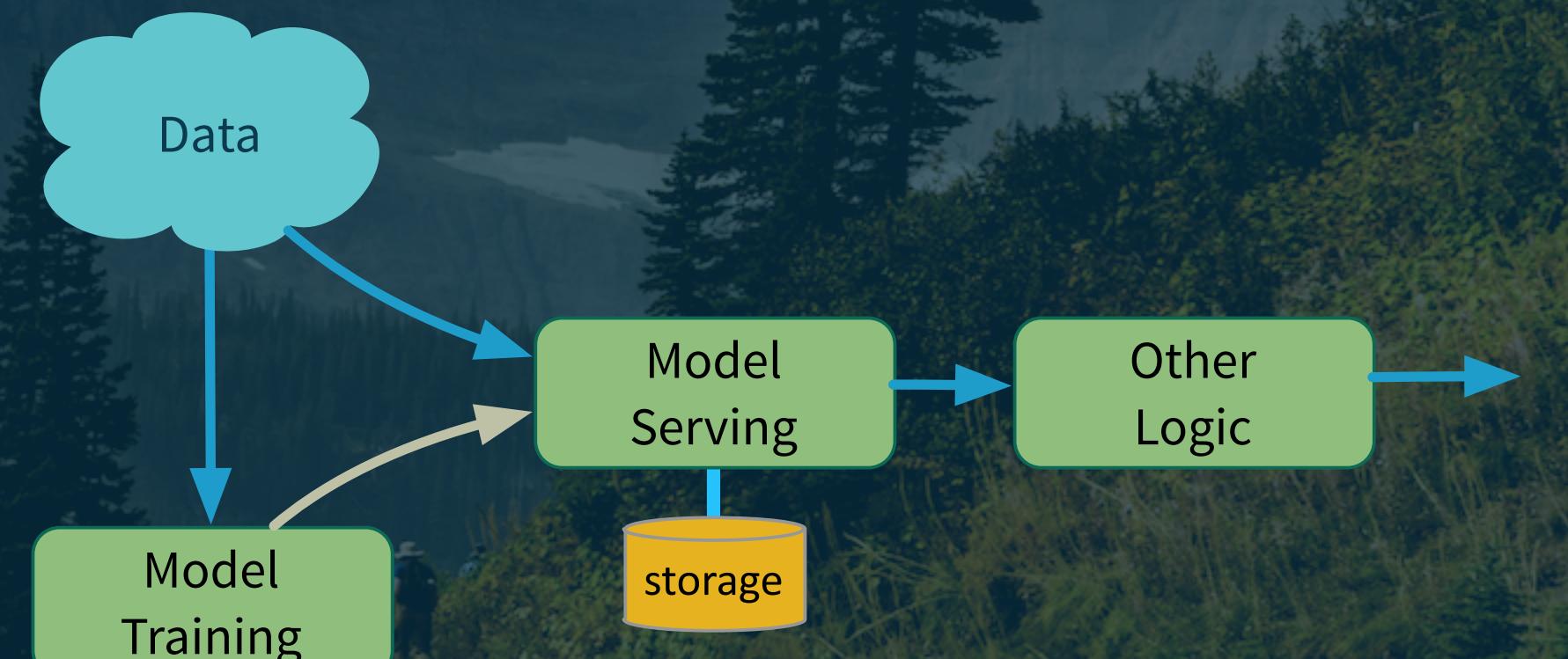


A Spectrum of Microservices

Event-driven μ -services



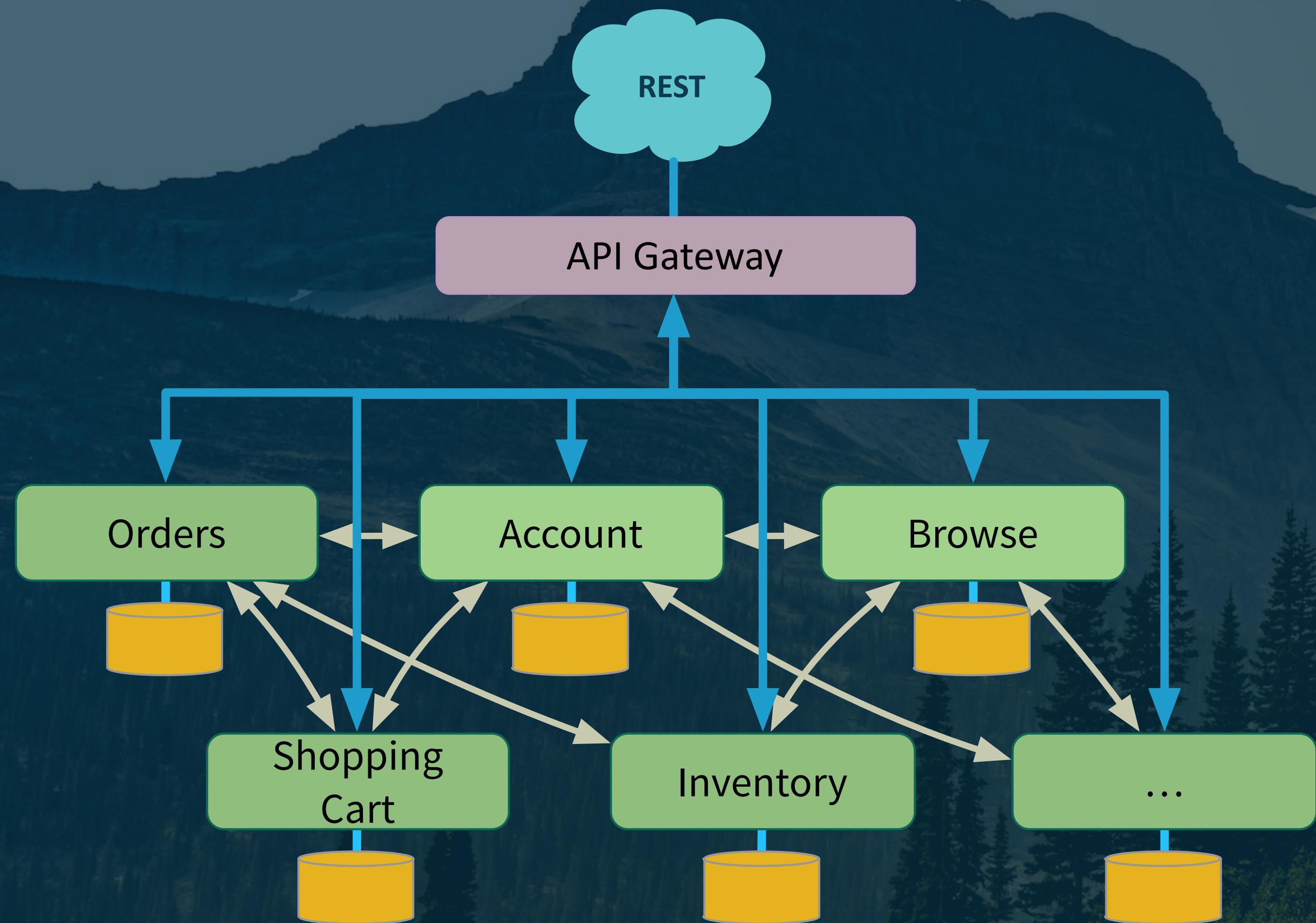
“Record-centric” μ -services



Events

Records

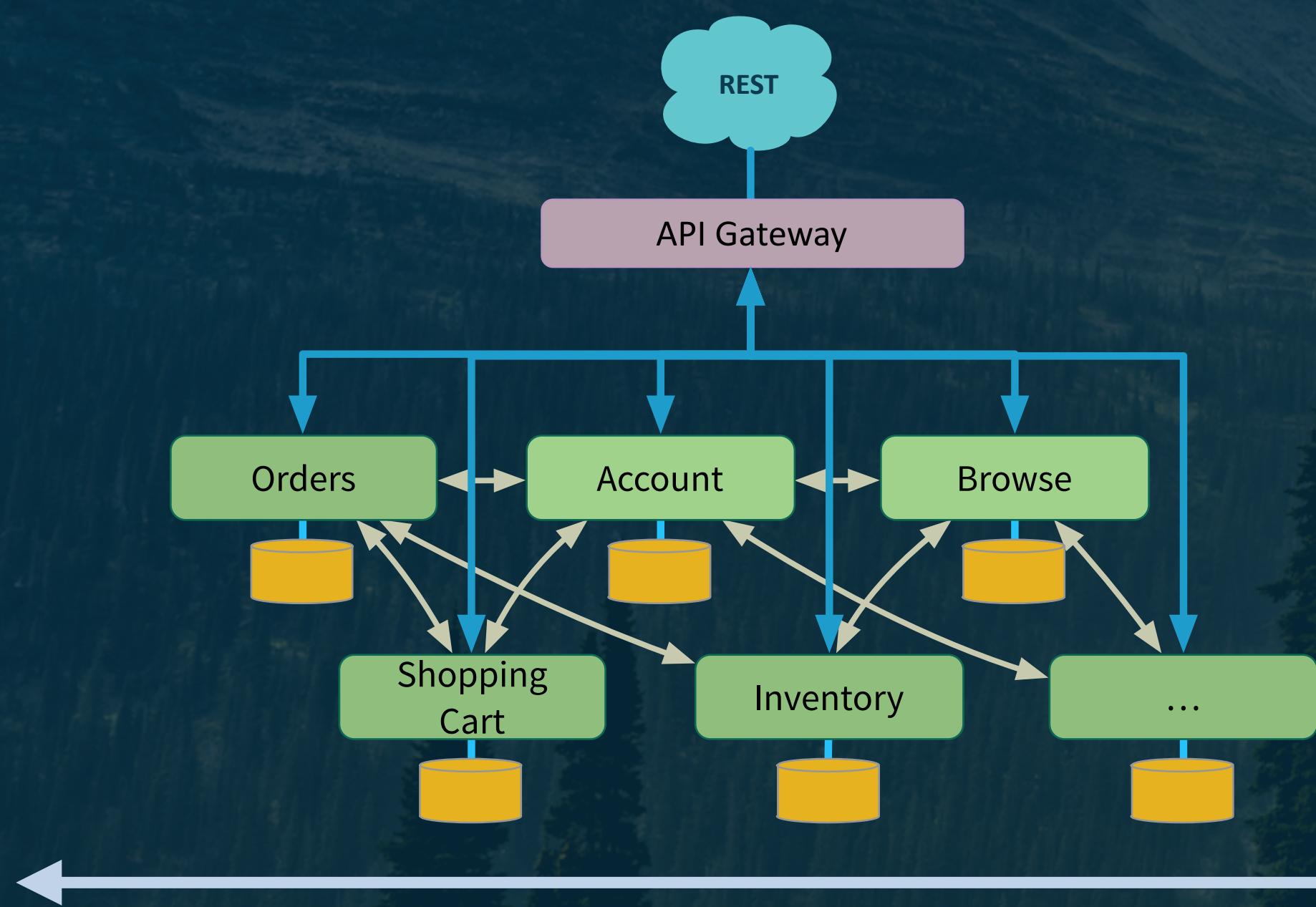
A Spectrum of Microservices



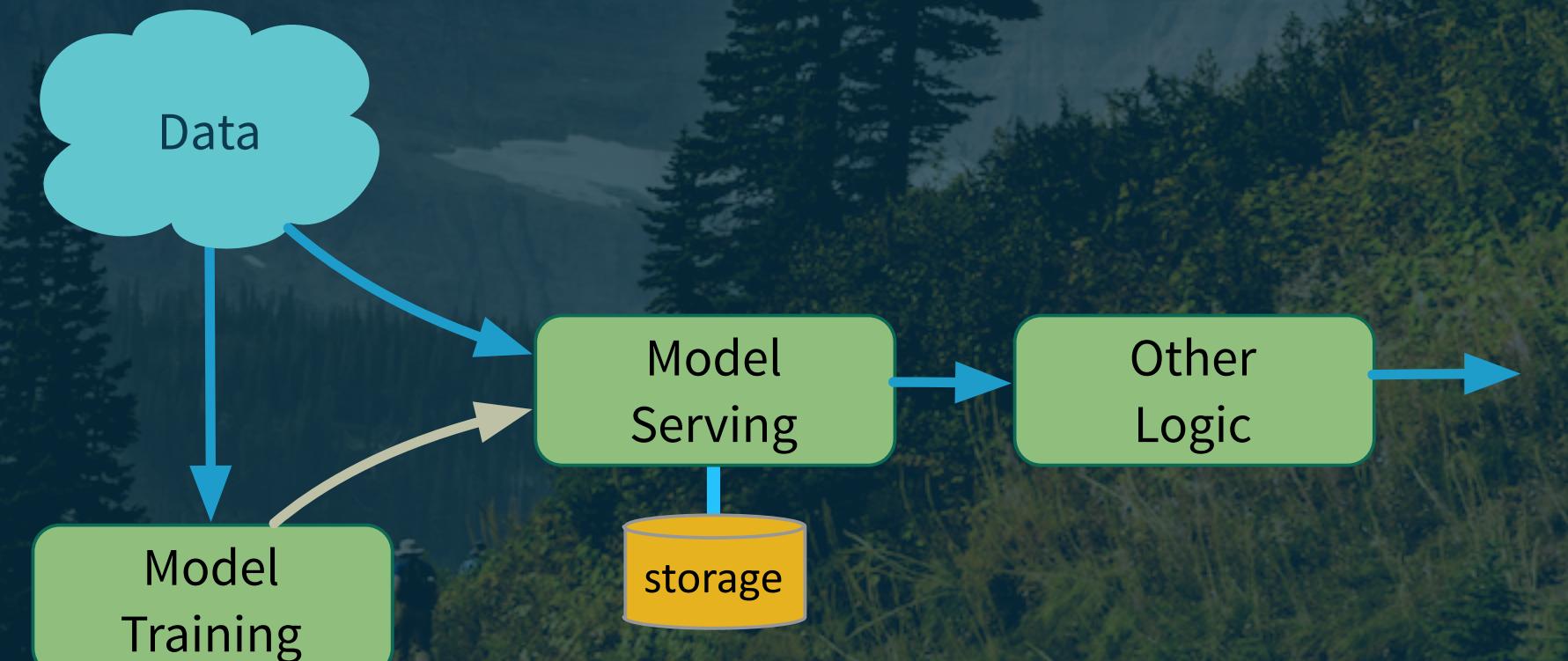
- Each item has a unique identity
- Process each item uniquely
- Think sessions and state machines

A Spectrum of Microservices

Event-driven μ -services



“Record-centric” μ -services

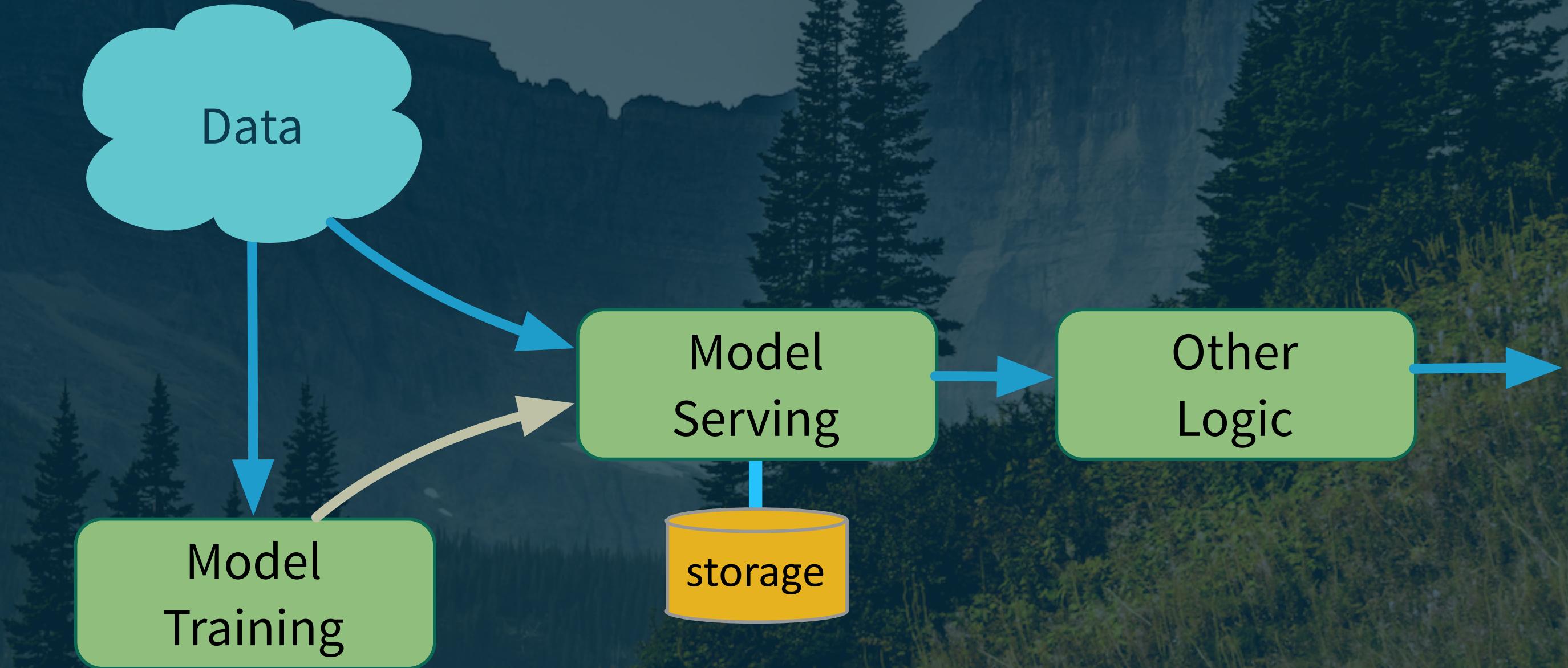


Events

Records

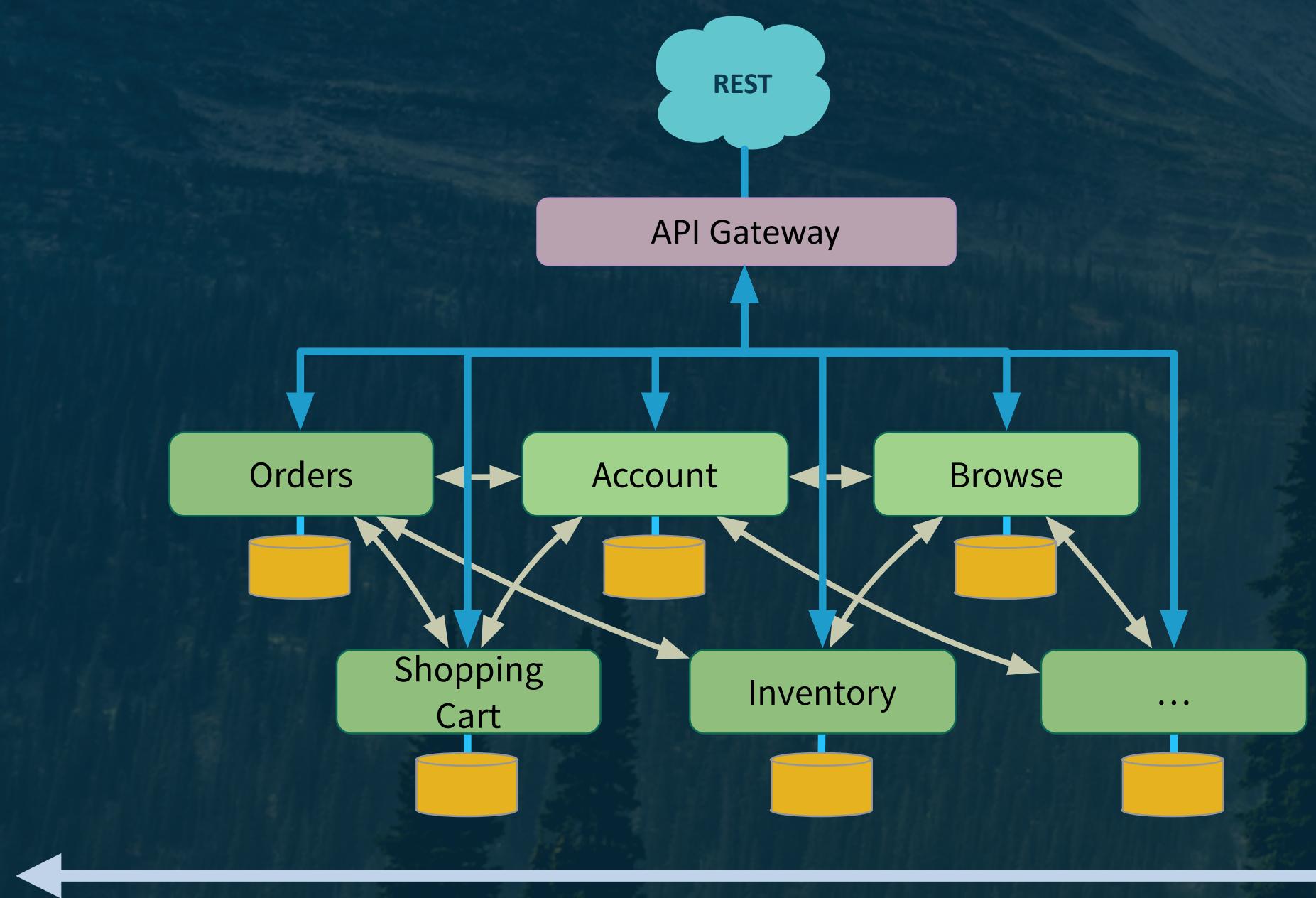
A Spectrum of Microservices

- Anonymous items
- Process *en masse*
- Think SQL queries



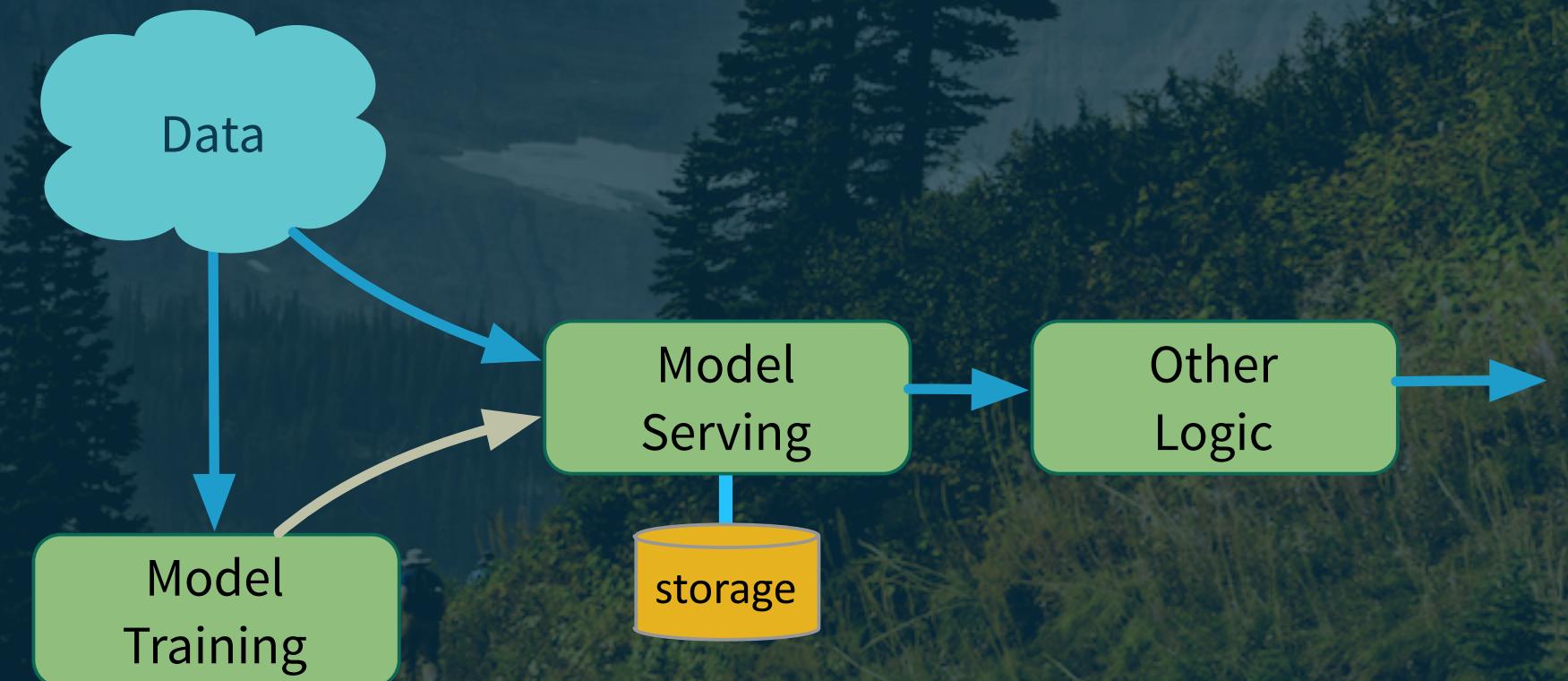
A Spectrum of Microservices

Event-driven μ-services



Events

“Record-centric” μ-services

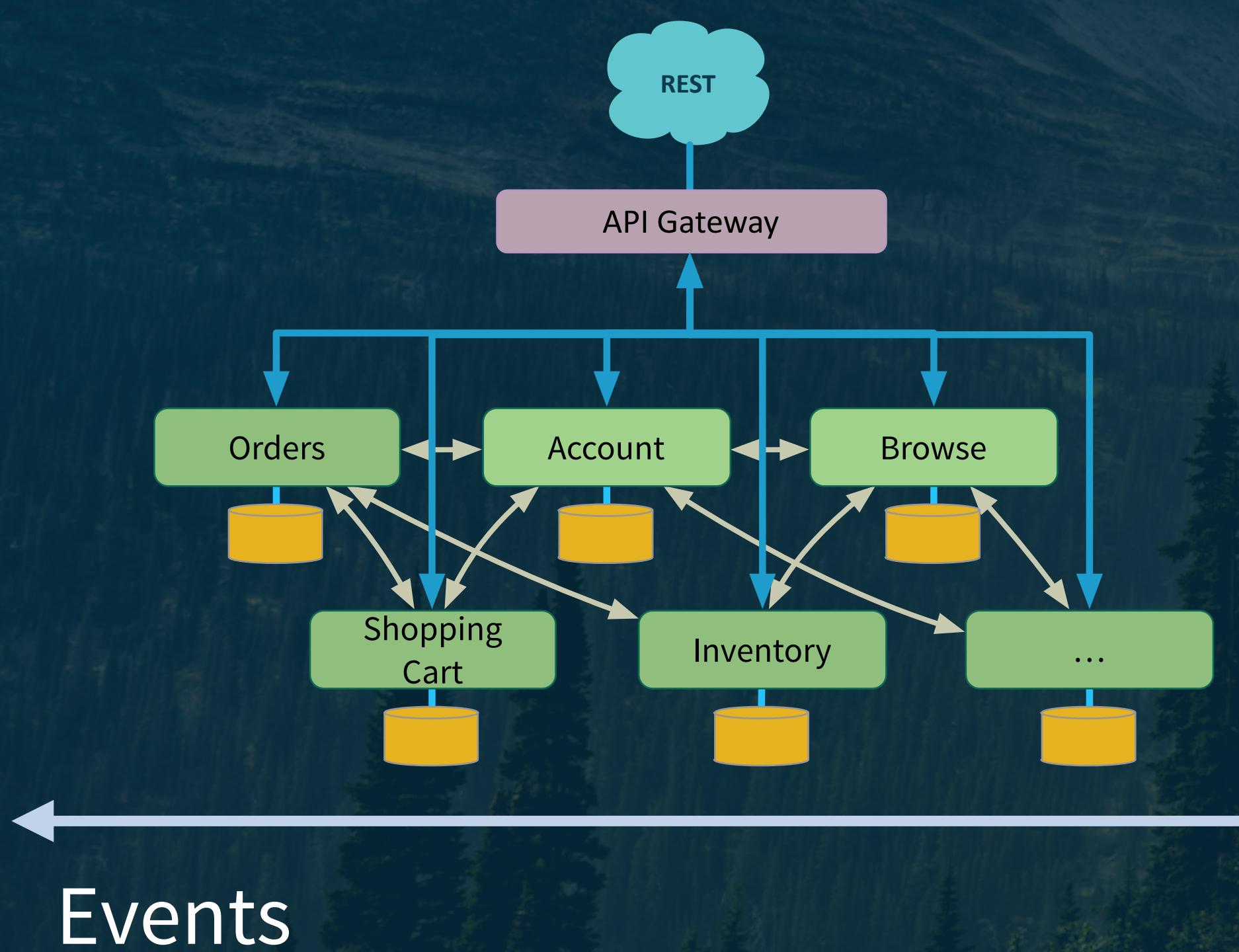


Records

A Spectrum of Microservices



Event-driven μ -services



Akka emerged from the left-hand side of the spectrum, the world of highly *Reactive* microservices.

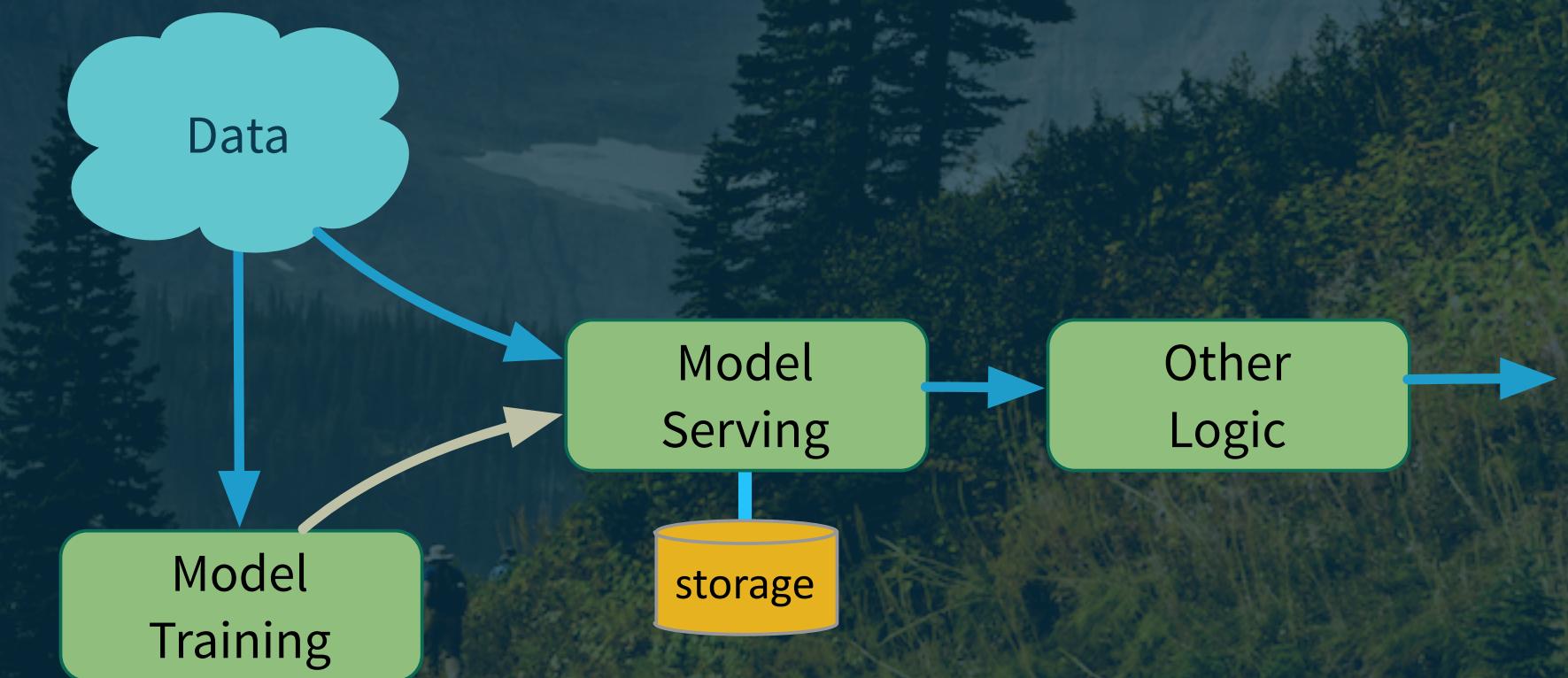
Akka Streams pushes to the right, more data-centric.

A Spectrum of Microservices



Emerged from the right-hand “Record-centric” μ -services side.

Kafka Streams pushes to the left, supporting many event-processing scenarios.



← Events → Records



kafka

Kafka Streams

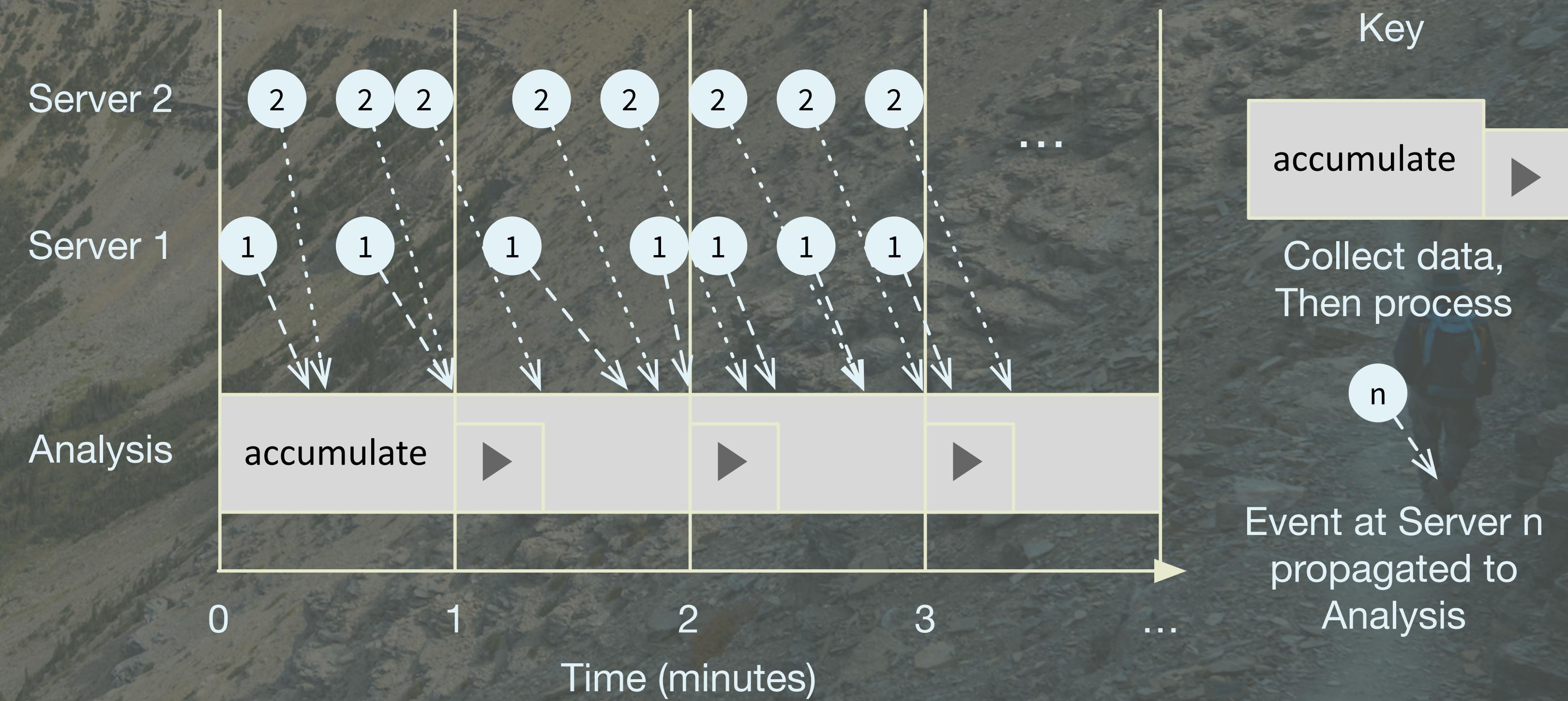




kafka

Kafka Streams

- Important stream-processing semantics, e.g.,
- Distinguish between *event time* and *processing time*

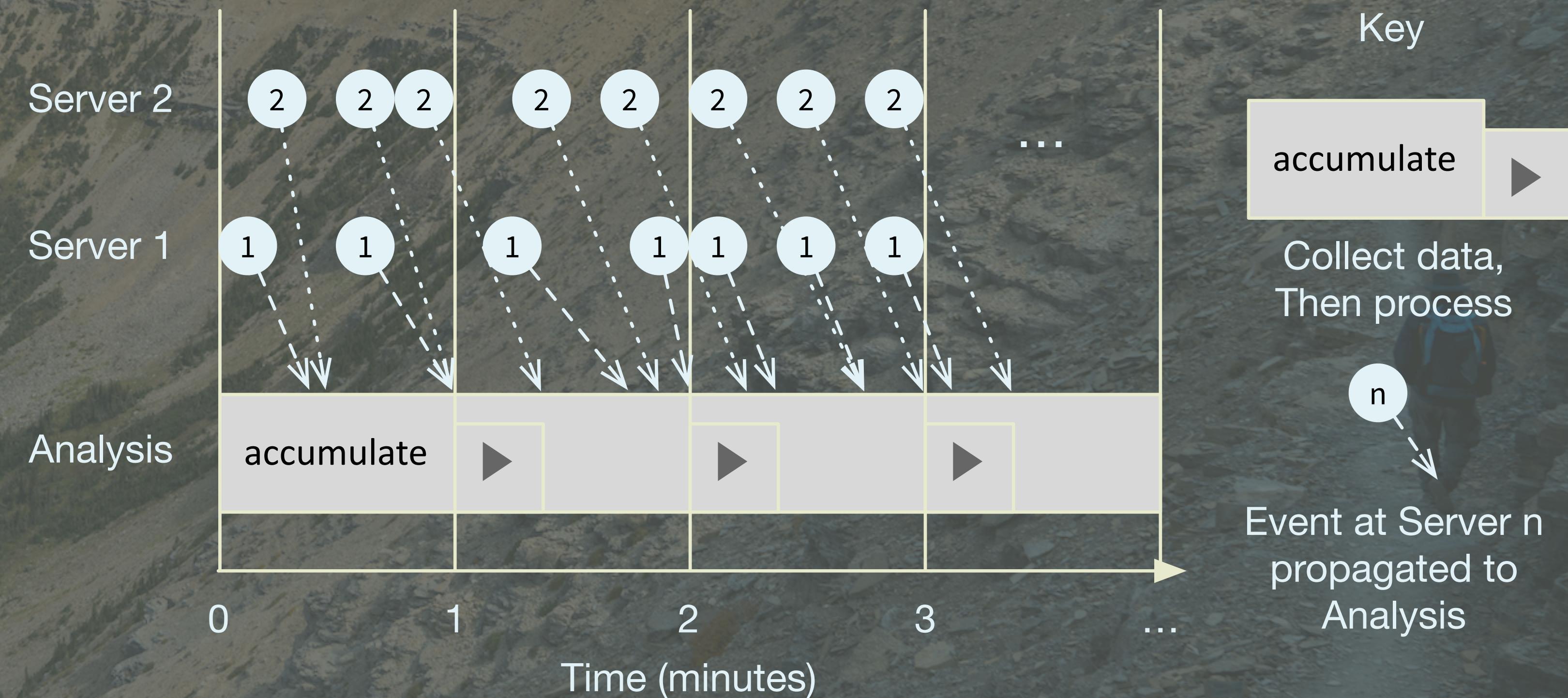




kafka

Kafka Streams

- Important stream-processing semantics, e.g.,
- Windowing support (e.g., group by within a window)



See my O'Reilly
report for details.



Kafka Streams

kafka

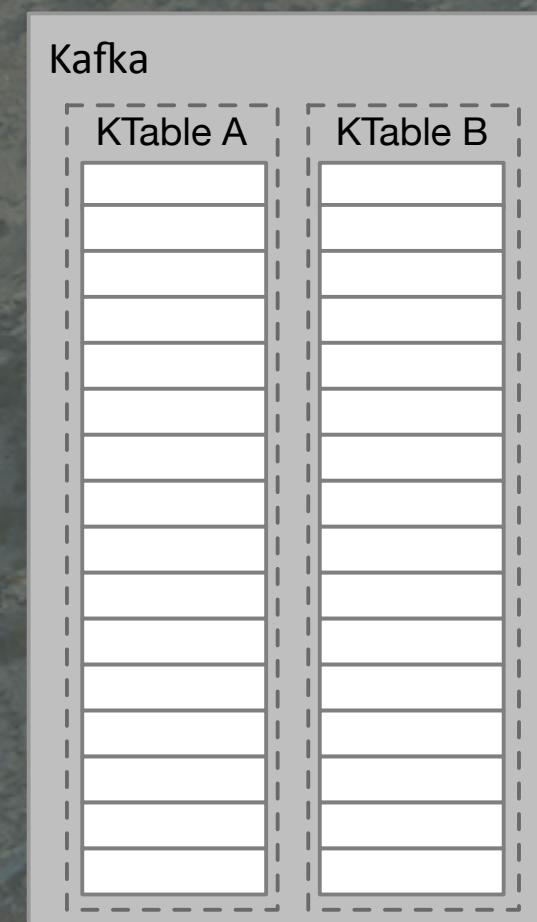
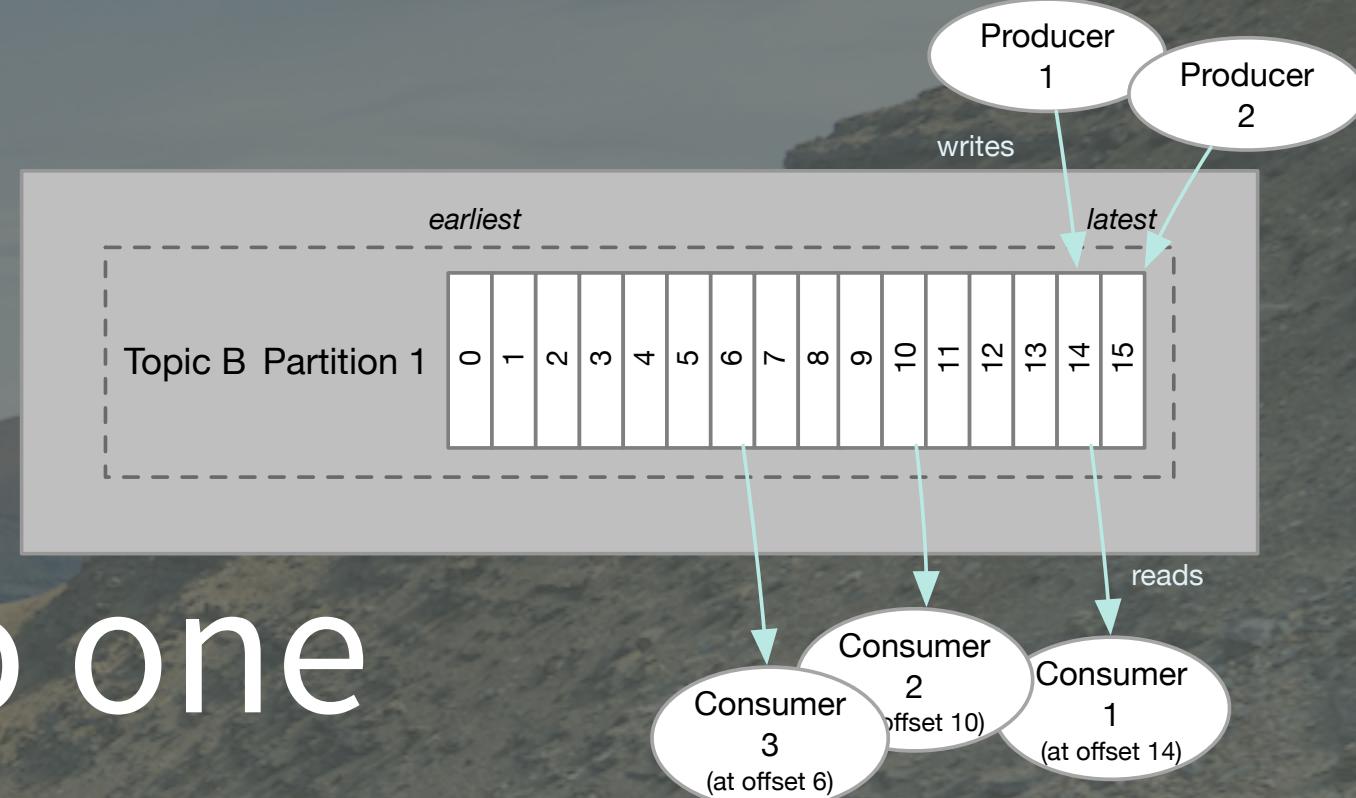
- *Exactly once*
- vs.
 - *At most once* - “fire and forget”
 - *At least once* - keep trying until acknowledged
(but you have to handle de-duplication)
 - Note: it’s really *effectively once*



kafka

Kafka Streams

- KStream
 - Per record transformations, one to one mapping
- KTable
 - Last value per key
 - Useful for holding *state* values





kafka

Kafka Streams

- Read from and write to Kafka topics, memory
 - Kafka Connect used for other sources and sinks
- Load balance and scale using topic partitioning

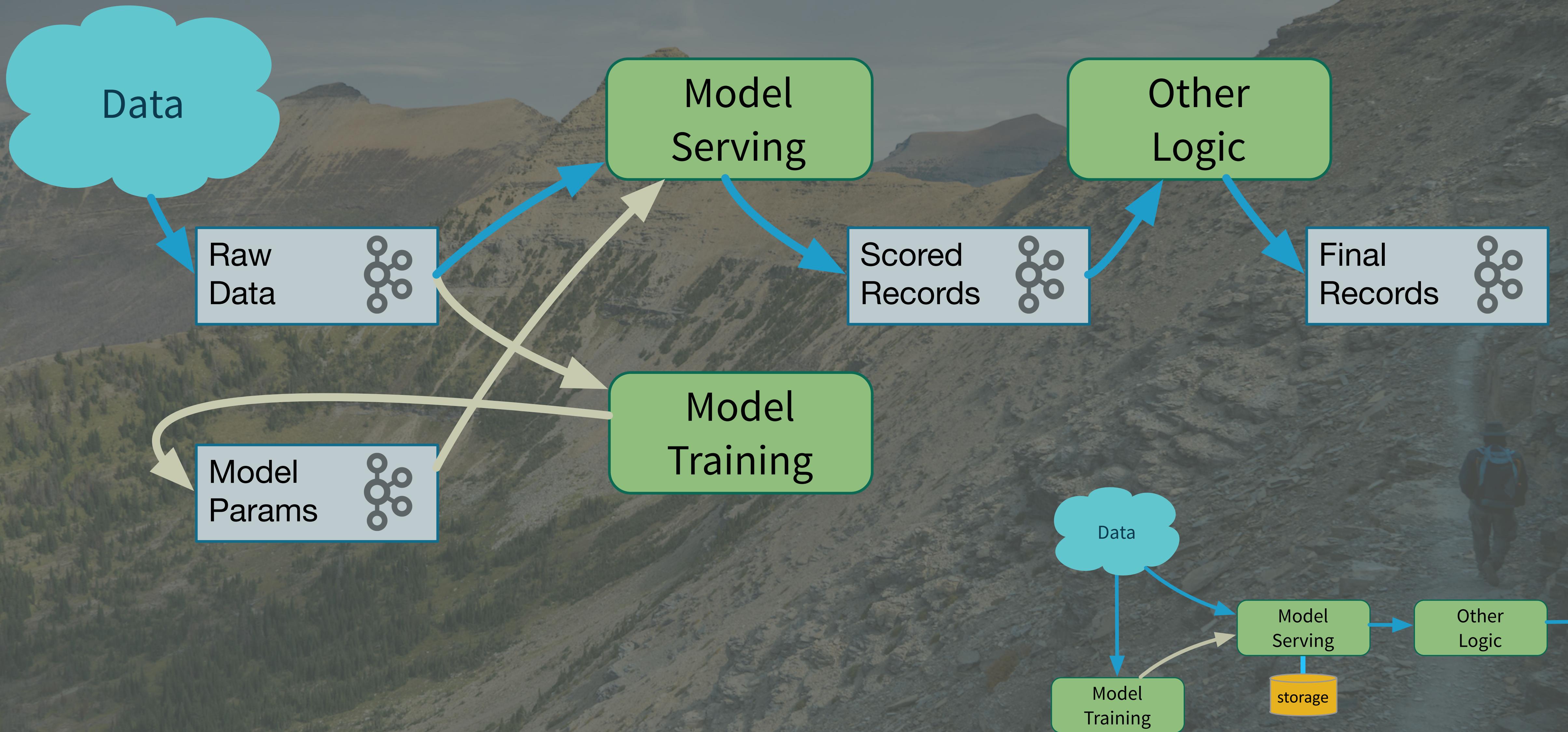


kafka

Kafka Streams

- Java API
- Scala API: written by Lightbend
- SQL!!

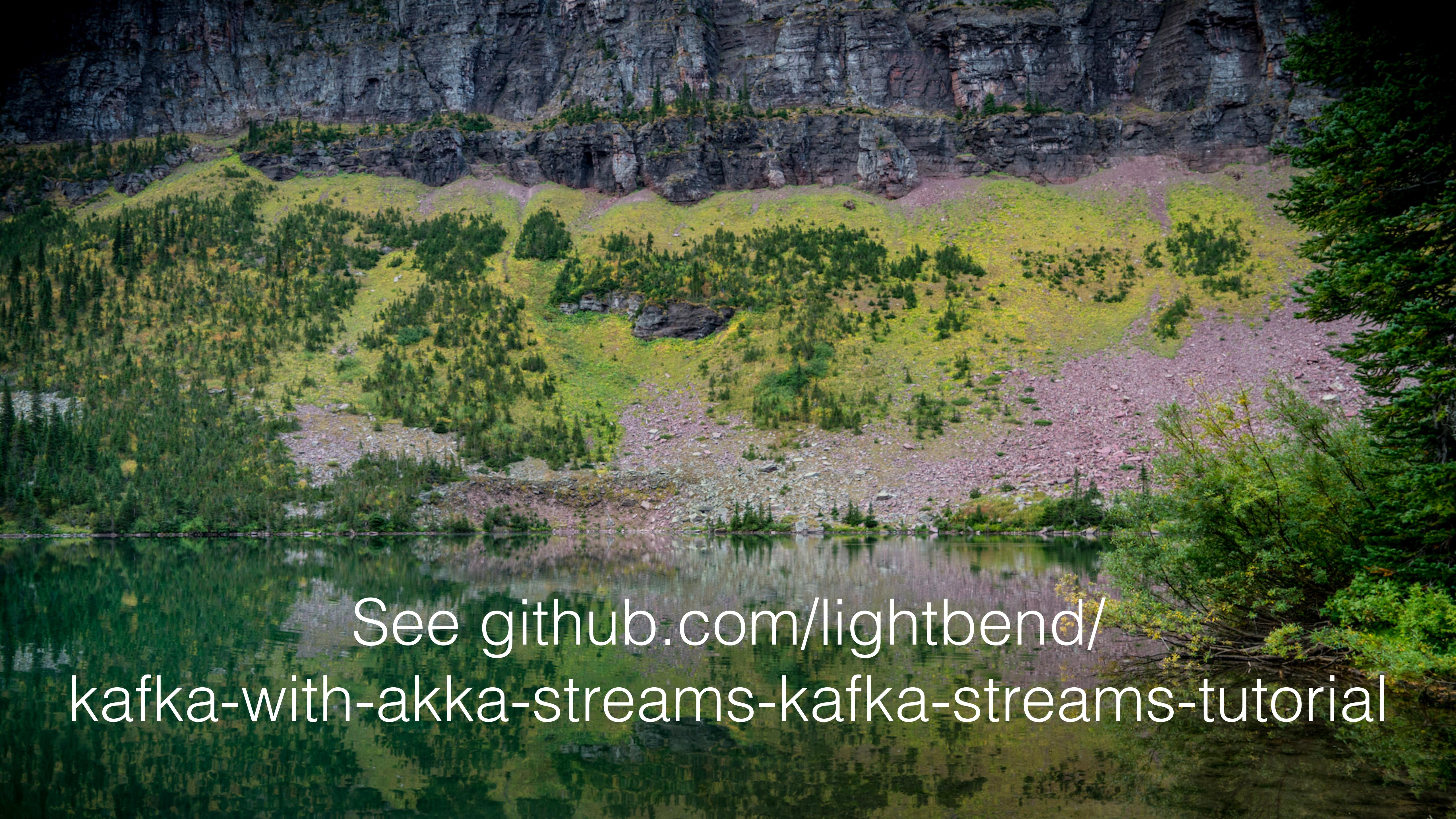
Kafka Streams Example



Kafka Streams Example

We'll use the new Scala Kafka Streams API:

- Developed by Debasish Ghosh, Boris Lublinsky, Sean Glover, et al. from Lightbend
- See also the following, if you know about *queryable state*:
 - <https://github.com/lightbend/kafka-streams-query>

A scenic landscape featuring a calm lake in the foreground, its surface reflecting the surrounding environment. Behind the lake is a steep mountain slope covered in a mix of dark rock and patches of green grass and small trees. The upper portion of the image shows a massive, rugged cliff face composed of layered rock.

See [github.com/lightbend/
kafka-with-akka-streams-kafka-streams-tutorial](https://github.com/lightbend/kafka-with-akka-streams-kafka-streams-tutorial)

```

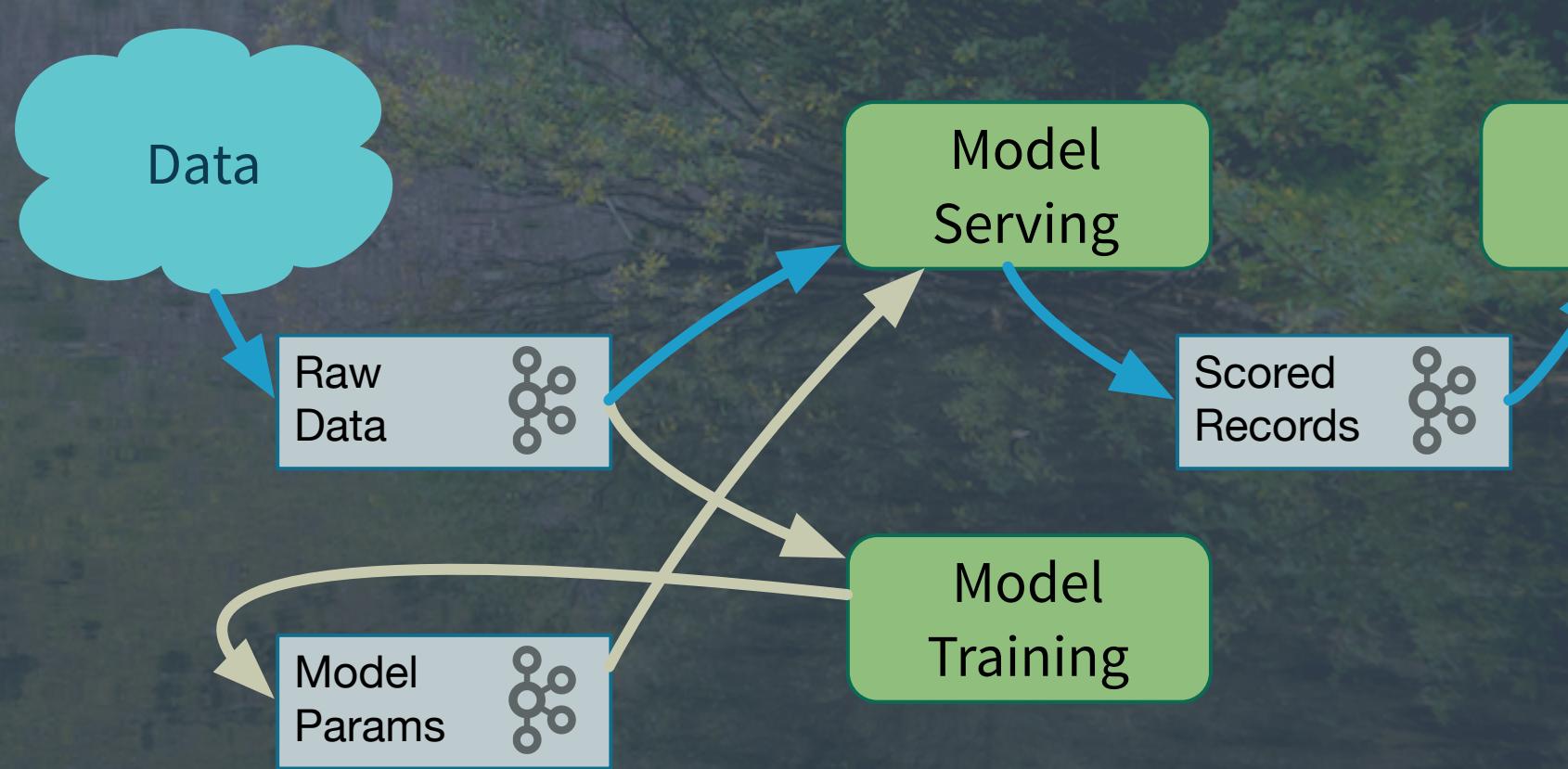
val builder = new StreamsBuilders // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
  .filter((key, model) => model.valid) // Successful?
  .mapValues(model => ModelImpl.findModel(model))
  .process(() => modelProcessor, ...) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
  .filter((key, record) => record.valid)
  .mapValues(record => new ScoredRecord(scorer.score(record), record))
  .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())

```

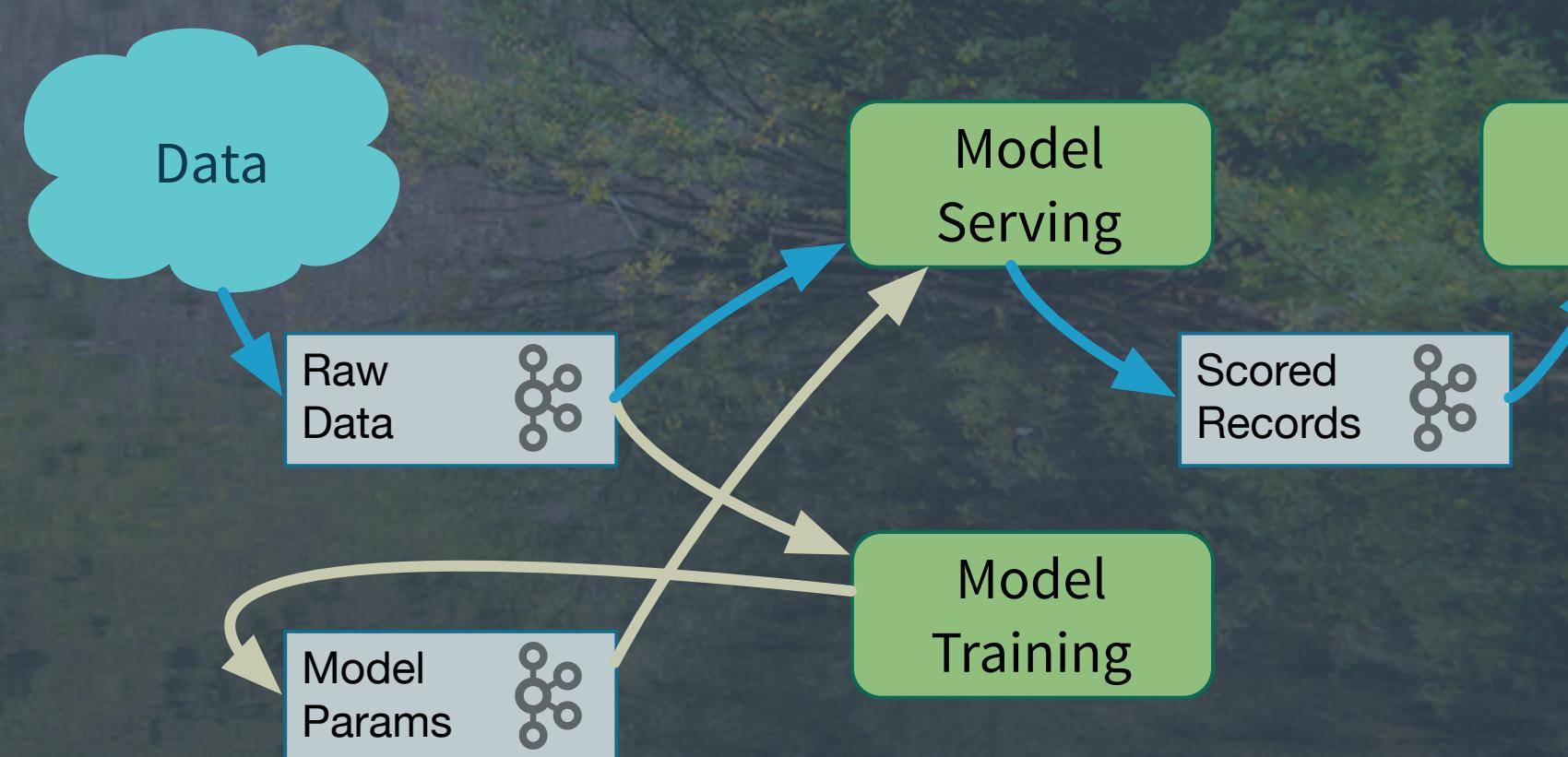


```
val builder = new StreamsBuilders // New Scala Wrapper API.
```

```
val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used
```

```
model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
    .filter((key, model) => model.valid) // Successful?
    .mapValues(model => ModelImpl.findModel(model))
    .process(() => modelProcessor, ...) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(score(scorer.score(record), record)))
    .to(scoredRecordsTopic)
```

```
val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

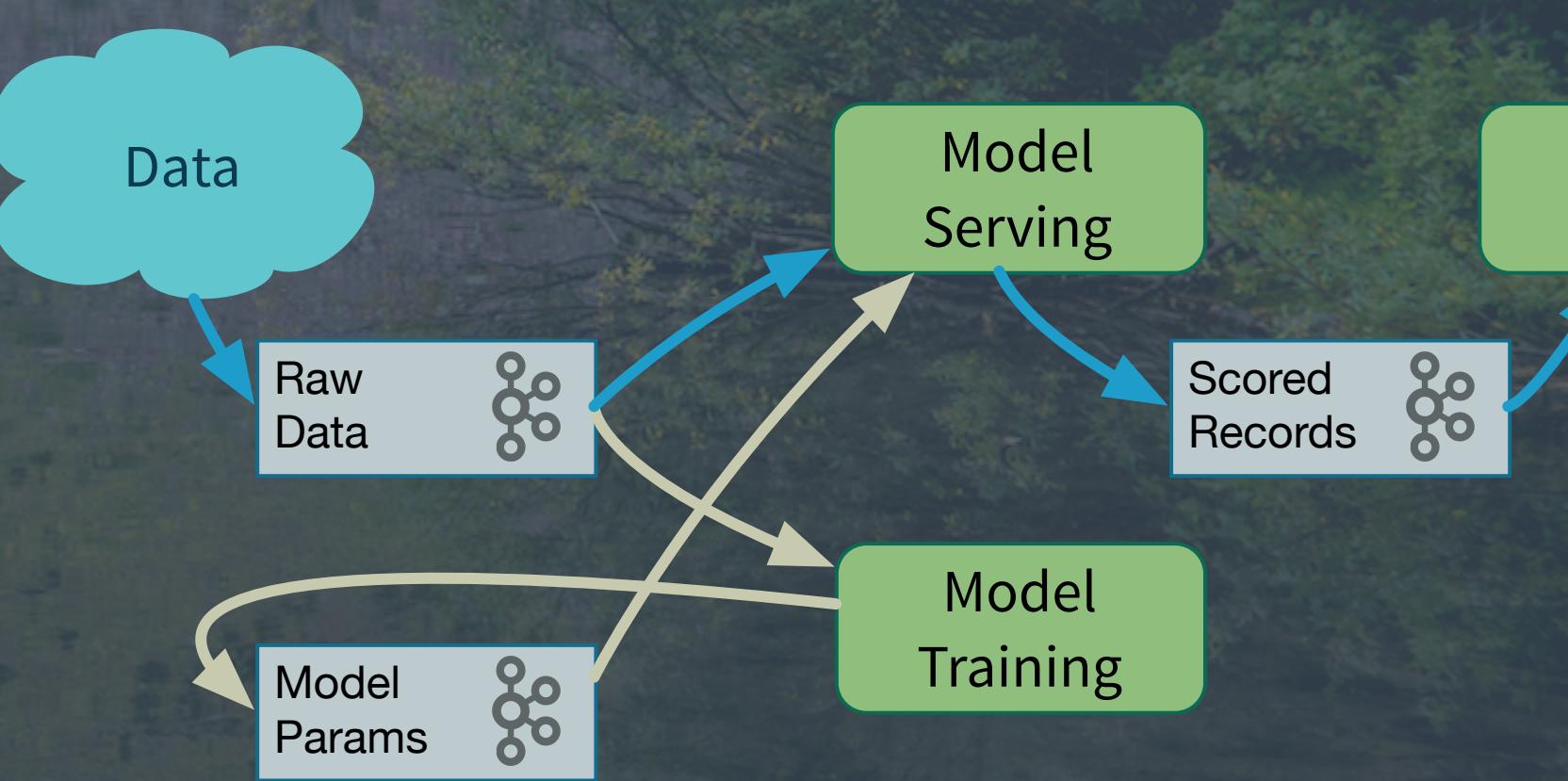


```
val builder = new StreamsBuilders // New Scala Wrapper API.
```

```
val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
  .filter((key, model) => model.valid) // Successful?
  .mapValues(model => ModelImpl.findModel(model))
  .process(() => modelProcessor, ...) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
  .filter((key, record) => record.valid)
  .mapValues(record => new ScoredRecord(scorer.score(record), record))
  .to(scoredRecordsTopic)
```

```
val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

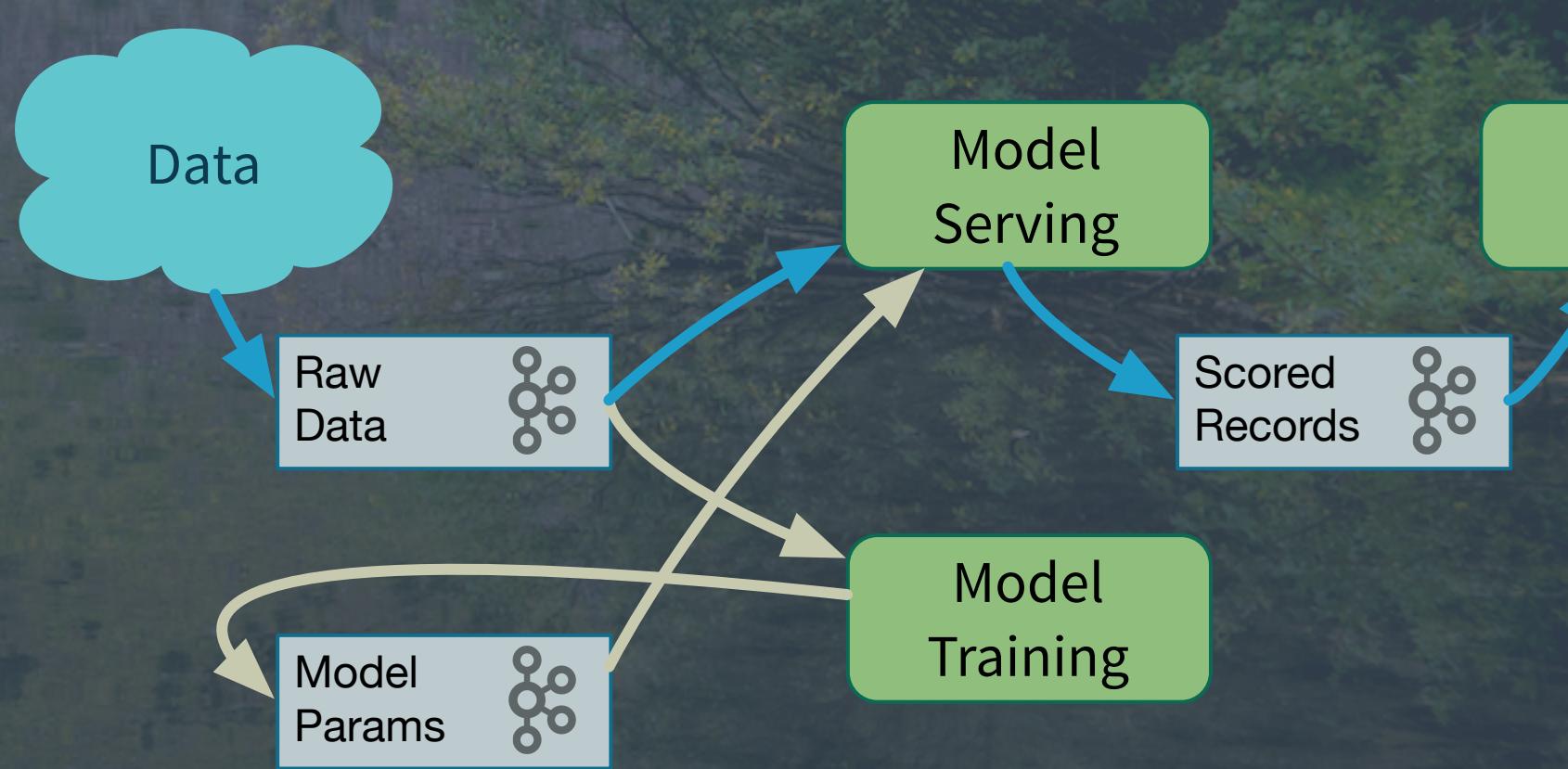


```
val builder = new StreamsBuilders // New Scala Wrapper API.
```

```
val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
  .filter((key, model) => model.valid) // Successful?
  .mapValues(model => ModelImpl.findModel(model))
  .process(() => modelProcessor, ...) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
  .filter((key, record) => record.valid)
  .mapValues(record => new ScoredRecord(score(score, record), record))
  .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```



```

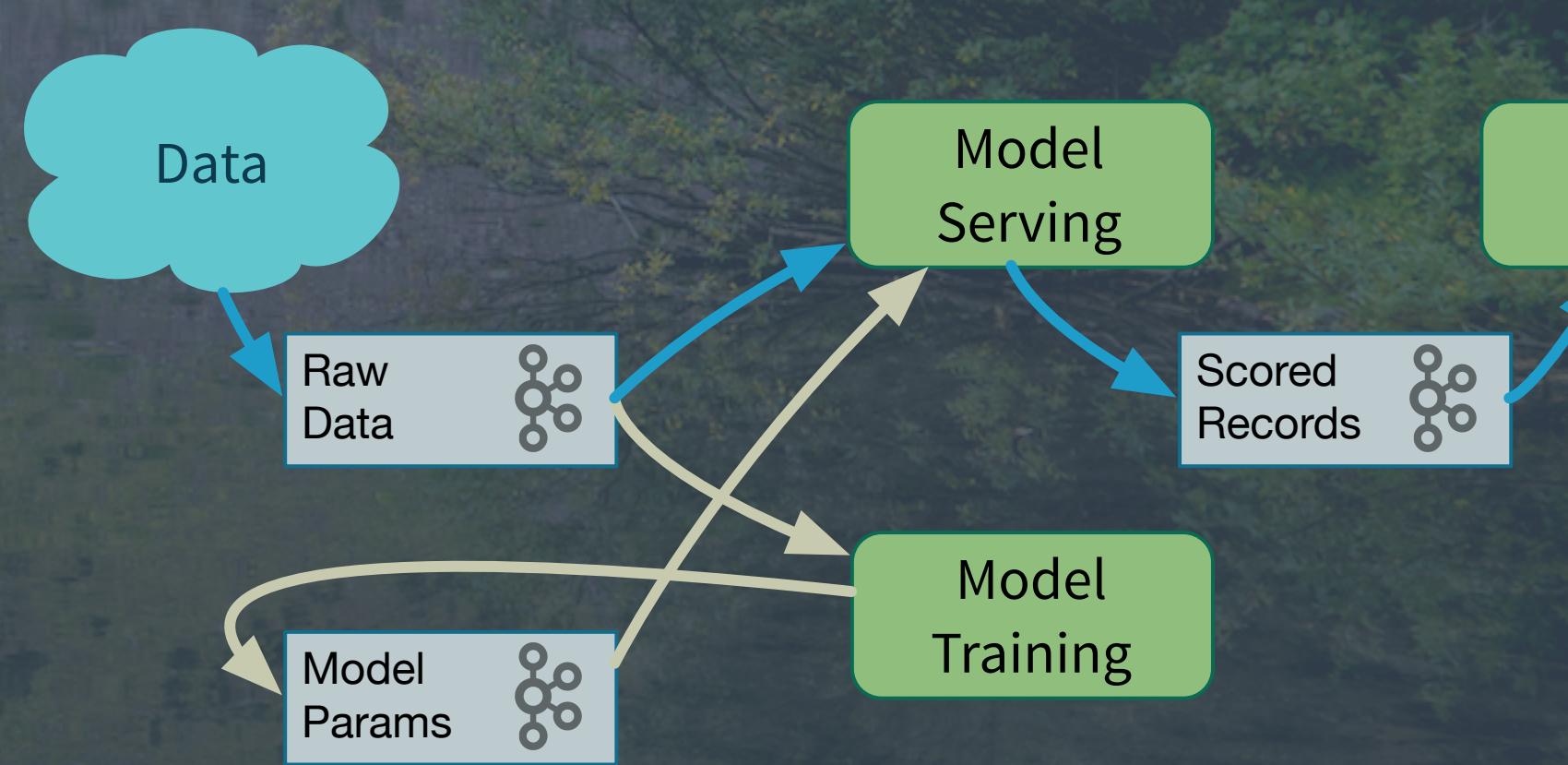
val builder = new StreamsBuilders // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
  .filter((key, model) => model.valid) // Successful?
  .mapValues(model => ModelImpl.findModel(model))
  .process(() => modelProcessor, ...) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
  .filter((key, record) => record.valid)
  .mapValues(record => new ScoredRecord(scorer.score(record), record))
  .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())

```



```

val builder = new StreamsBuilders // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

```

```

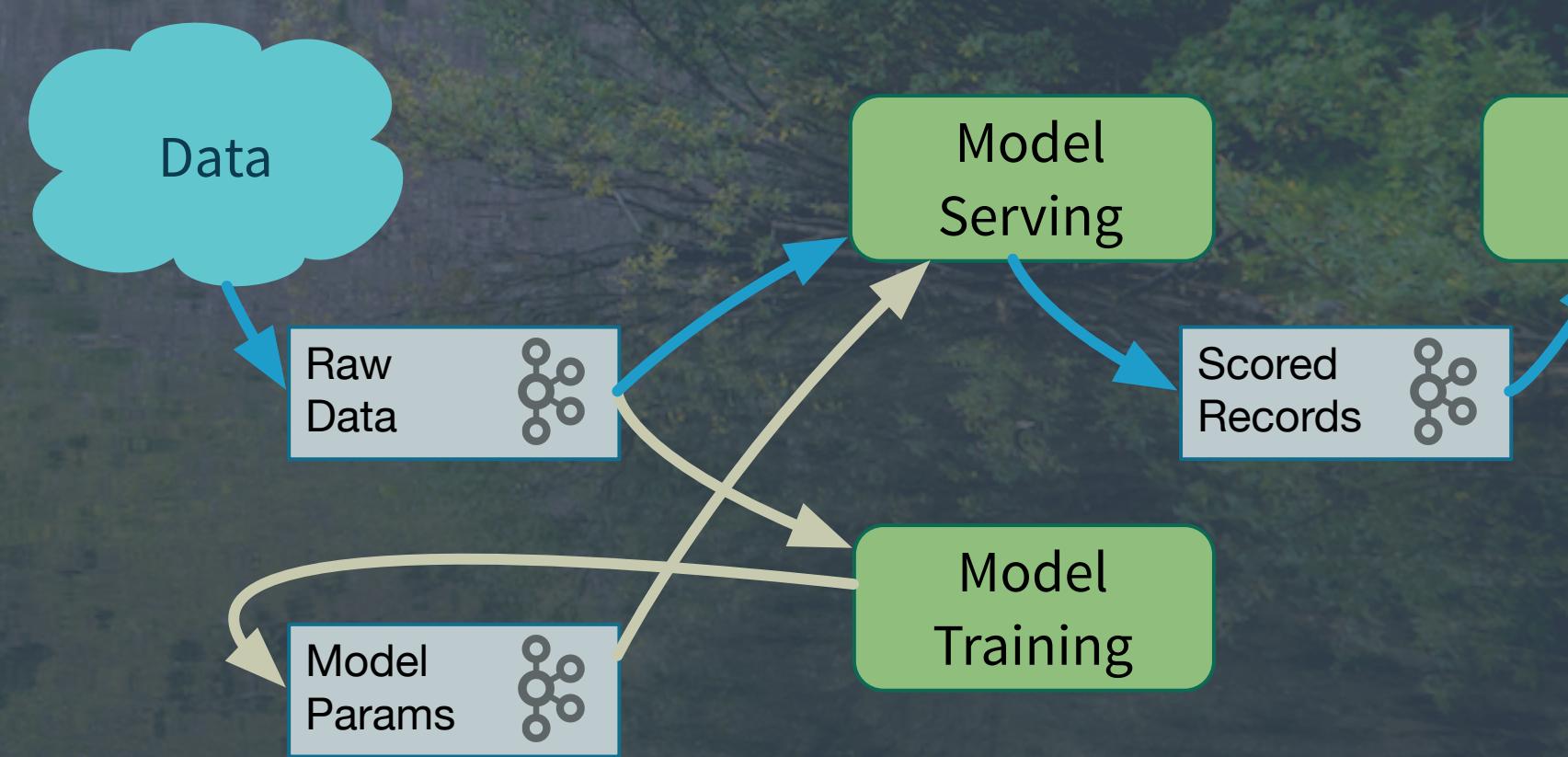
model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
  .filter((key, model) => model.valid) // Successful?
  .mapValues(model => ModelImpl.findModel(model))
  .process(() => modelProcessor, ...) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
  .filter((key, record) => record.valid)
  .mapValues(record => new ScoredRecord(score(score(record), record)))
  .to(scoredRecordsTopic)

```

```

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())

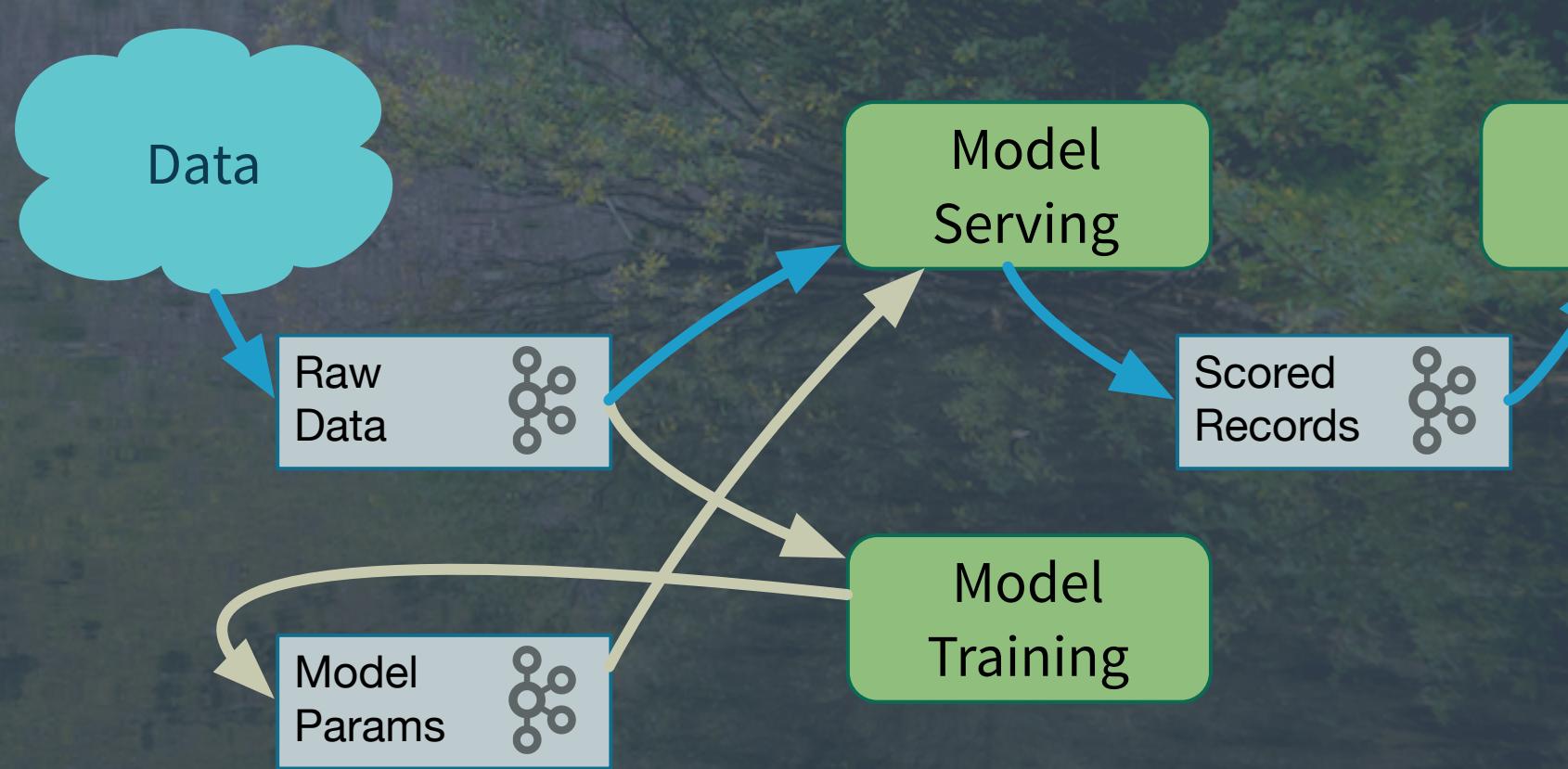
```



```
val builder = new StreamsBuilders // New Scala Wrapper API.  
  
val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)  
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)  
val modelProcessor = new ModelProcessor  
val scorer = new Scorer(modelProcessor) // scorer.score(record) used
```

```
model.mapValues(bytes => Model.parseBytes(bytes)) // array => record  
.filter((key, model) => model.valid) // Successful?  
.mapValues(model => ModelImpl.findModel(model))  
.process(() => modelProcessor, ...) // Set up actual model  
data.mapValues(bytes => DataRecord.parseBytes(bytes))  
.filter((key, record) => record.valid)  
.mapValues(record => new ScoredRecord(score(score(record), record)))  
.to(scoredRecordsTopic)
```

```
val streams = new KafkaStreams(  
  builder.build, streamsConfiguration)  
streams.start()  
sys.addShutdownHook(streams.close())
```



```

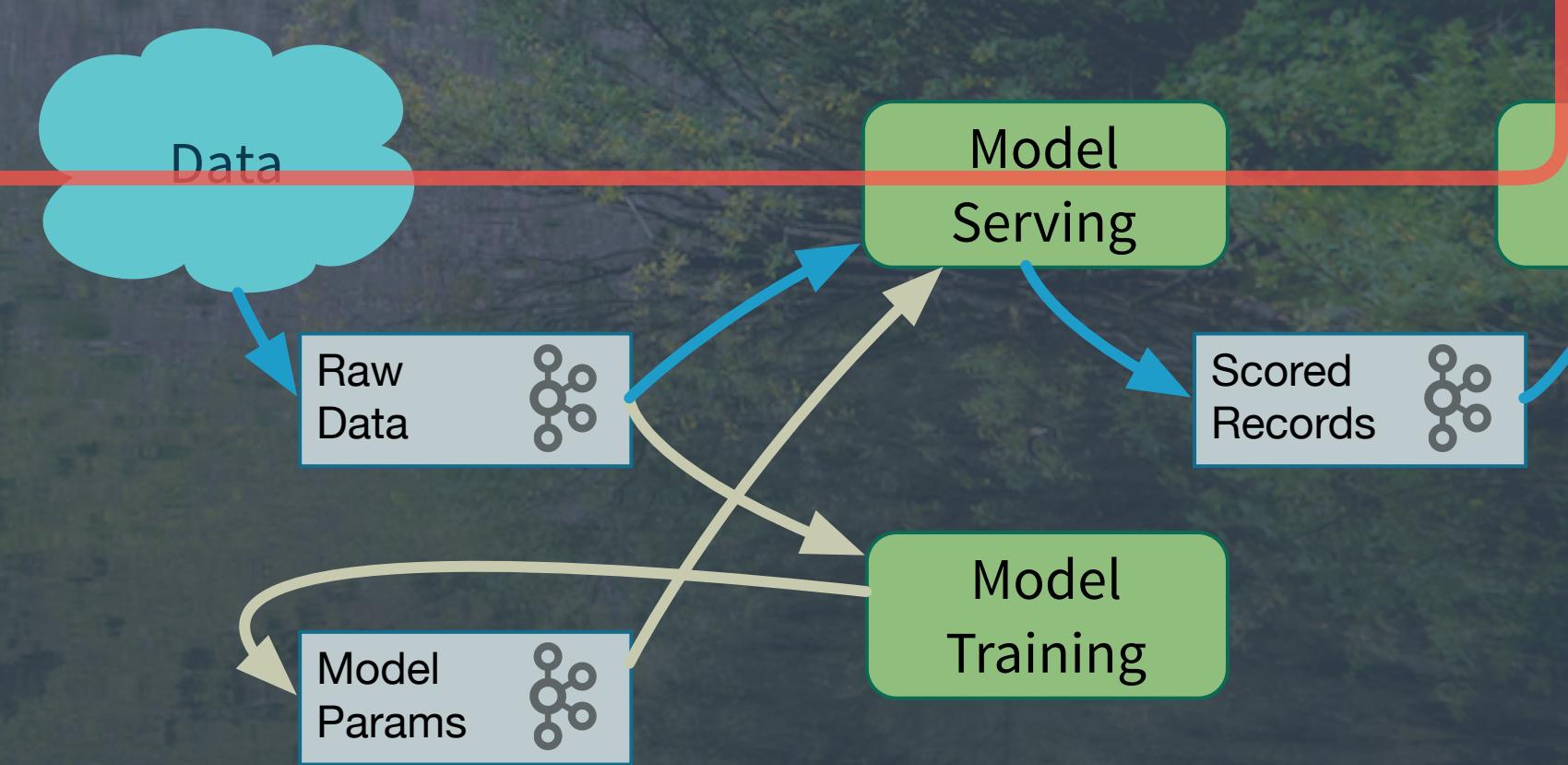
val builder = new StreamsBuilders // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

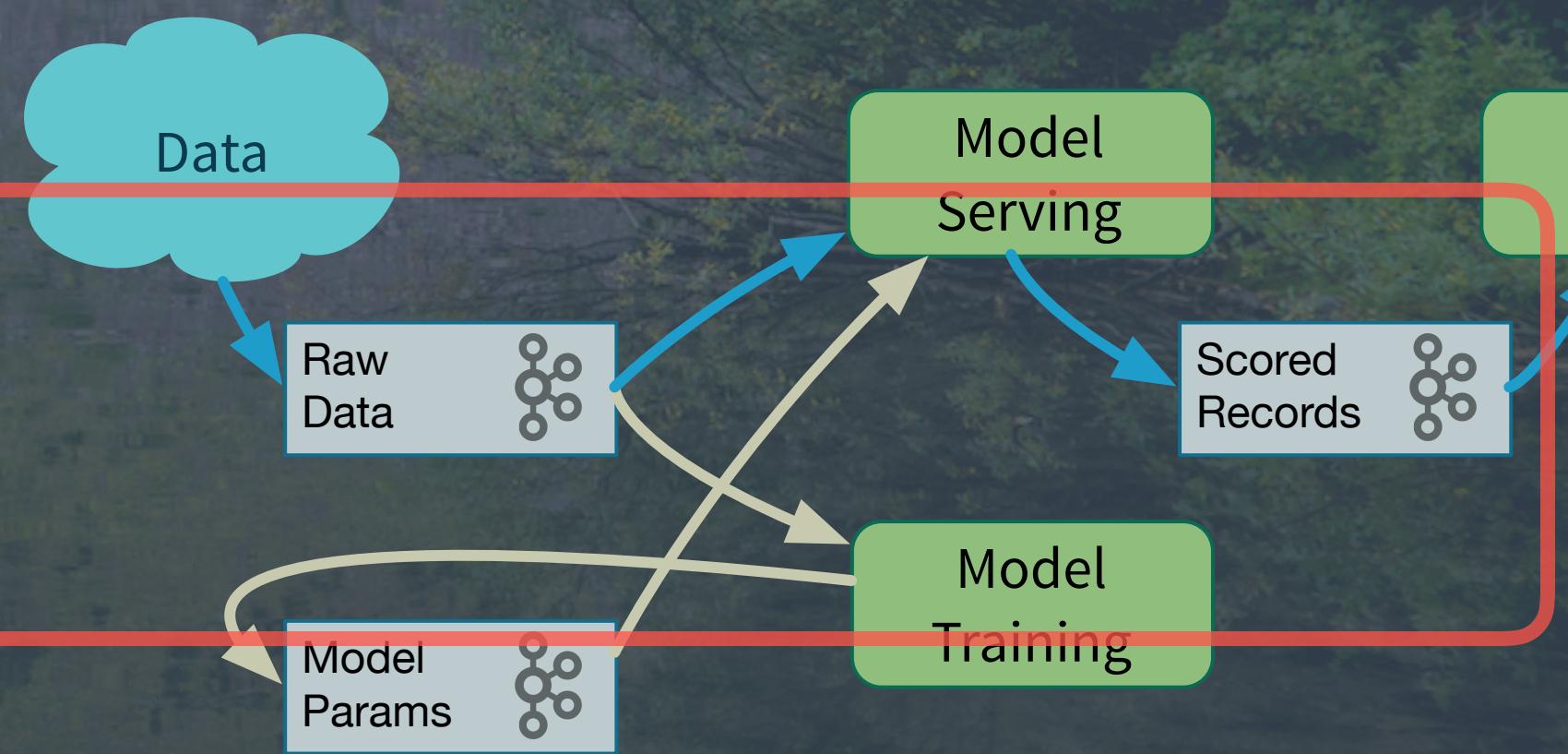
model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
  .filter((key, model) => model.valid) // Successful?
  .mapValues(model => ModelImpl.findModel(model))
  .process(() -> modelProcessor, ...) // Set up actual model
  data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(score(score(record), record)))
    .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())

```



```
val builder = new StreamsBuilders // New Scala Wrapper API.  
  
val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)  
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)  
val modelProcessor = new ModelProcessor  
val scorer = new Scorer(modelProcessor) // scorer.score(record) used  
  
model.mapValues(bytes => Model.parseBytes(bytes)) // array => record  
.filter((key, model) => model.valid) // Successful?  
.mapValues(model => ModelImpl.findModel(model))  
.process(() => modelProcessor, ...) // Set up actual model  
data.mapValues(bytes => DataRecord.parseBytes(bytes))  
.filter((key, record) => record.valid)  
.mapValues(record => new ScoredRecord(score(score(record), record)))  
.to(scoredRecordsTopic)  
  
val streams = new KafkaStreams(  
  builder.build, streamsConfiguration)  
streams.start()  
sys.addShutdownHook(streams.close())
```



```

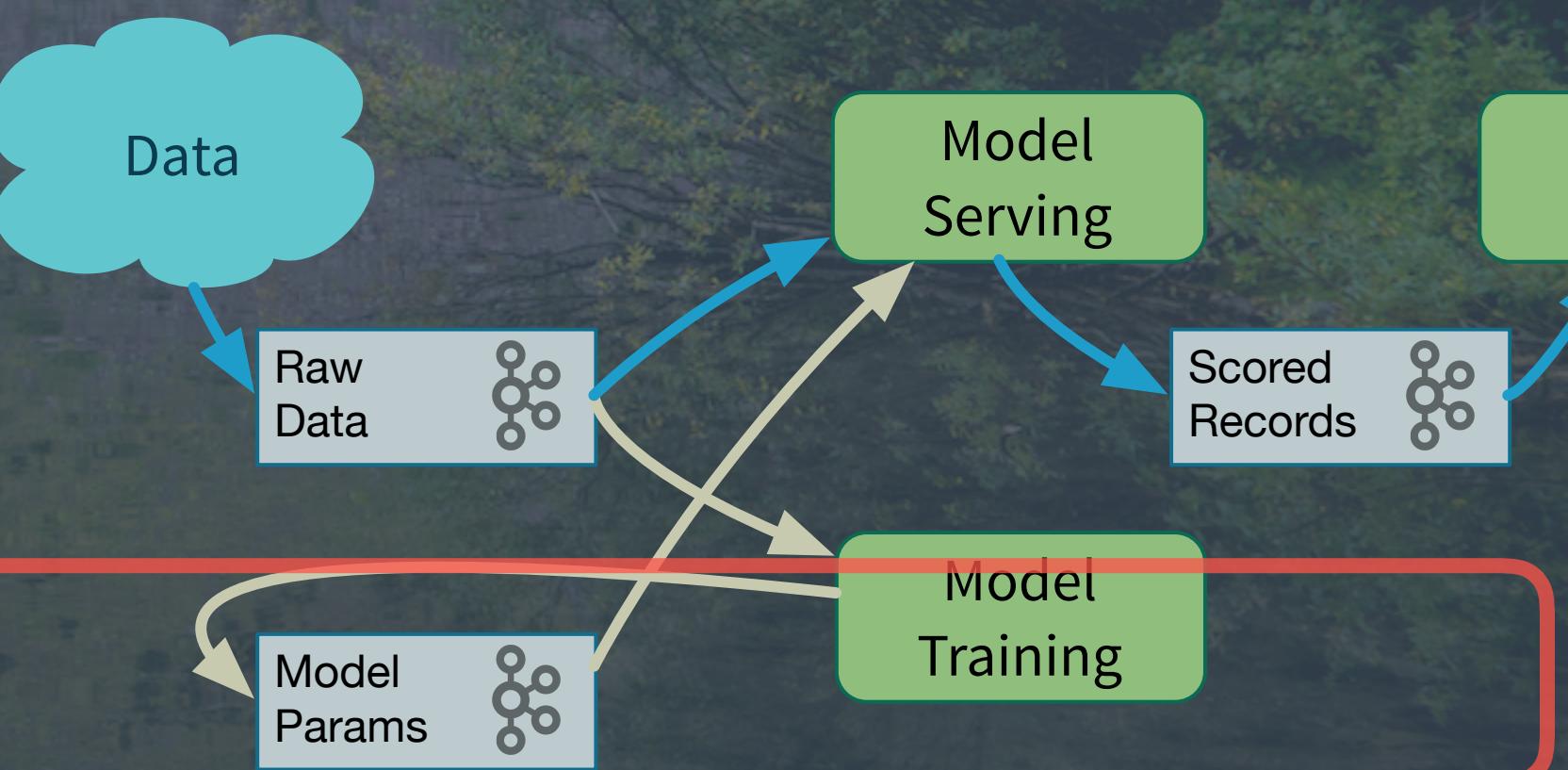
val builder = new StreamsBuilders // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
  .filter((key, model) => model.valid) // Successful?
  .mapValues(model => ModelImpl.findModel(model))
  .process(() => modelProcessor, ...) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
  .filter((key, record) => record.valid)
  .mapValues(record => new ScoredRecord(scorer.score(record), record))
  .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())

```



What's Missing?

Kafka Streams is a powerful library, but you'll need to provide the rest of the microservice support telling through other means. (Some are provided if you run the support services for the SQL interface.)

You would embed your KS code in microservices written with more comprehensive toolkits, such as the Lightbend Reactive Platform!

We'll return to this point...



akka Akka Streams

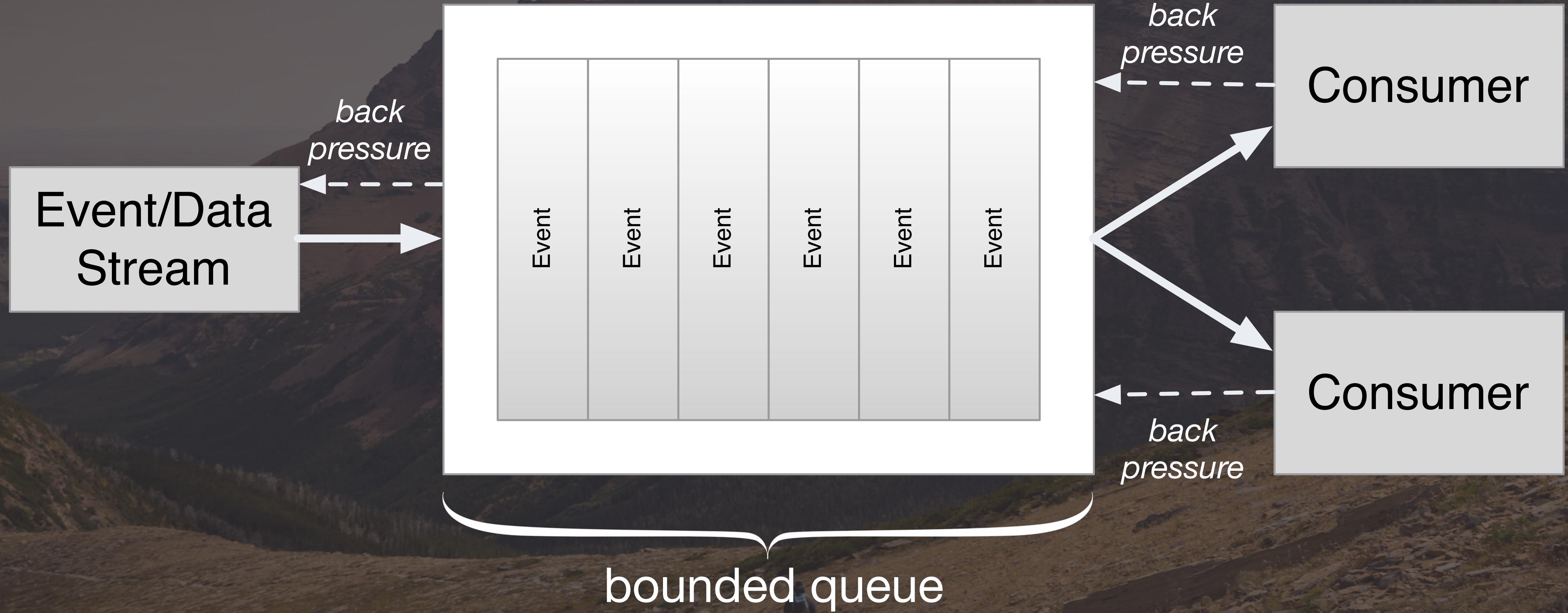


akka Akka Streams

- Implements *Reactive Streams*
 - <http://www.reactive-streams.org/>
- *Back pressure* for flow control

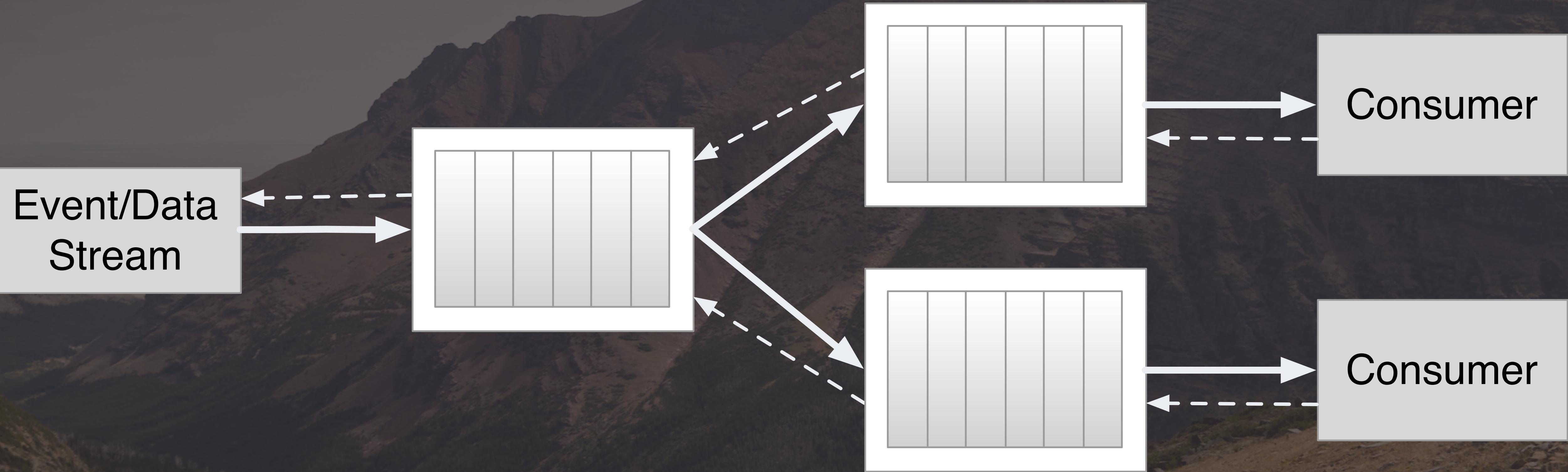


Akka Streams





Akka Streams



Back pressure composes!



akka Akka Streams

- Part of the Akka ecosystem
 - Akka Actors, Akka Cluster, Akka HTTP, Akka Persistence, ...
 - Alpakka - rich connection library
 - like Camel, but implements Reactive Streams
 - Very low overhead and latency



Akka Akka Streams

- The “gist”:

```
val source: Source[Int, NotUsed] = Source(1 to 10)
val factorials = source.scan(BigInt(1)) (
  (total, next) => total * next )
factorials.runWith(Sink.foreach(println))
```



Akka Streams

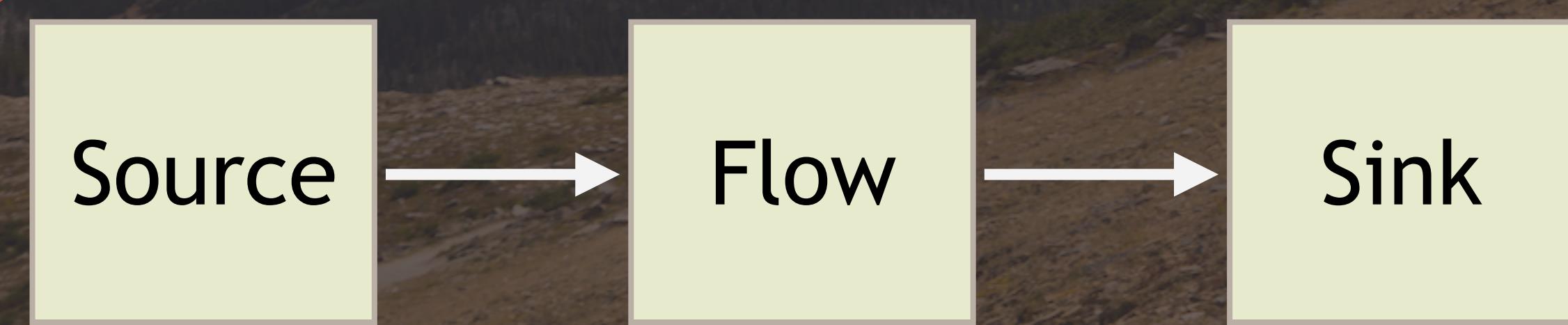
- The “gist”:

```
val source: Source[Int, NotUsed] = Source(1 to 10)
```

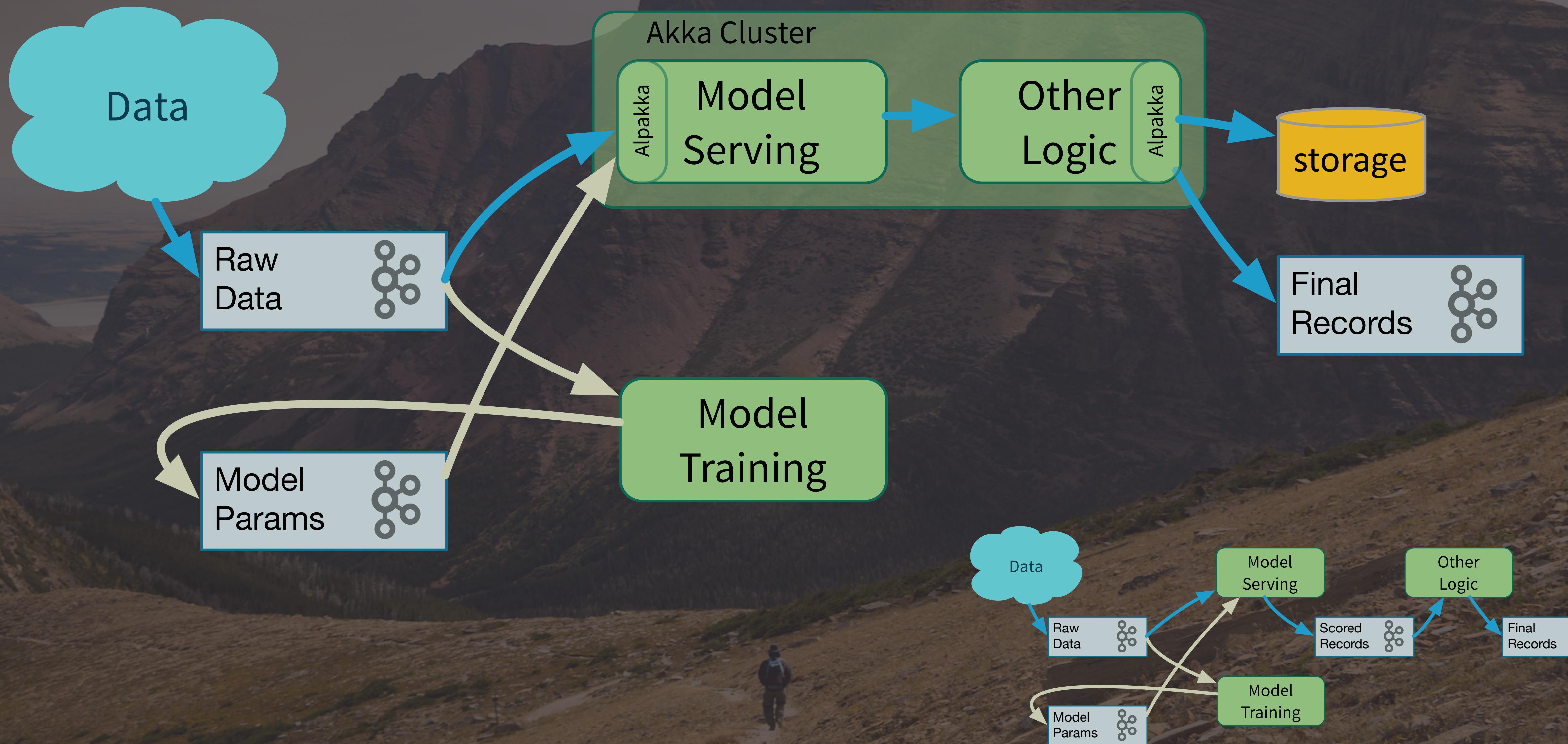
```
val factorials = source.scan(BigInt(1)) ( (total, next) => total * next )
```

```
factorials.runWith(Sink.foreach(println))
```

A “Graph”



Akka Streams Example

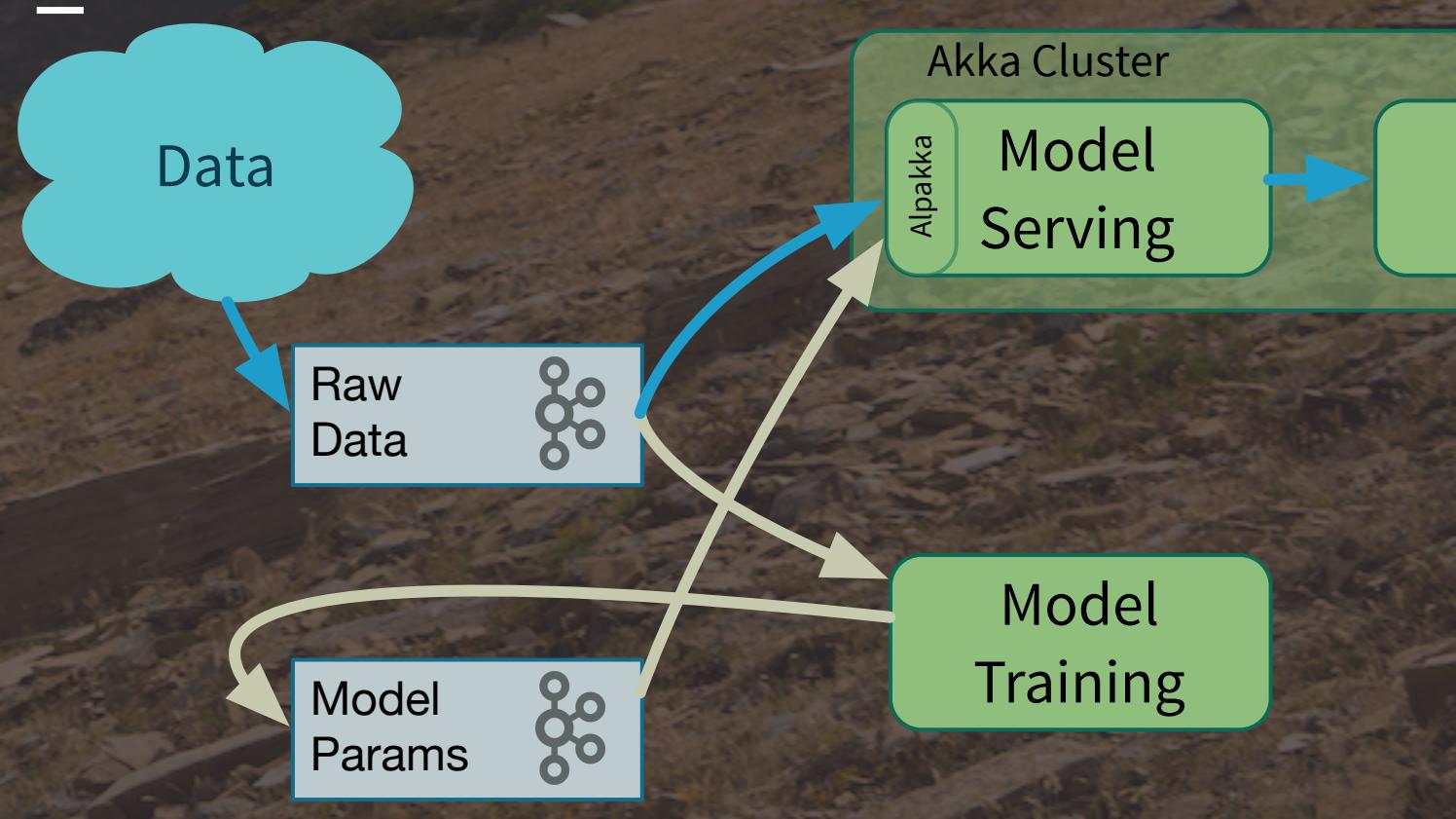


```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher
```

```
val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom "stage"
```

```
val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
    Subscriptions.topics(rawDataTopic))
  .map(input => DataRecord.parseBytes(input.value()))
  .collect{ case Success(data) => data }
```

```
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
    Subscriptions.topics(modelTopic))
  .map(input => Model.parseBytes(input.value()))
  .collect{ case Success(mod) => mod }
  .map(model => ModelImpl.findModel(model))
  .collect{ case Success(modImpl) => modImpl }
```

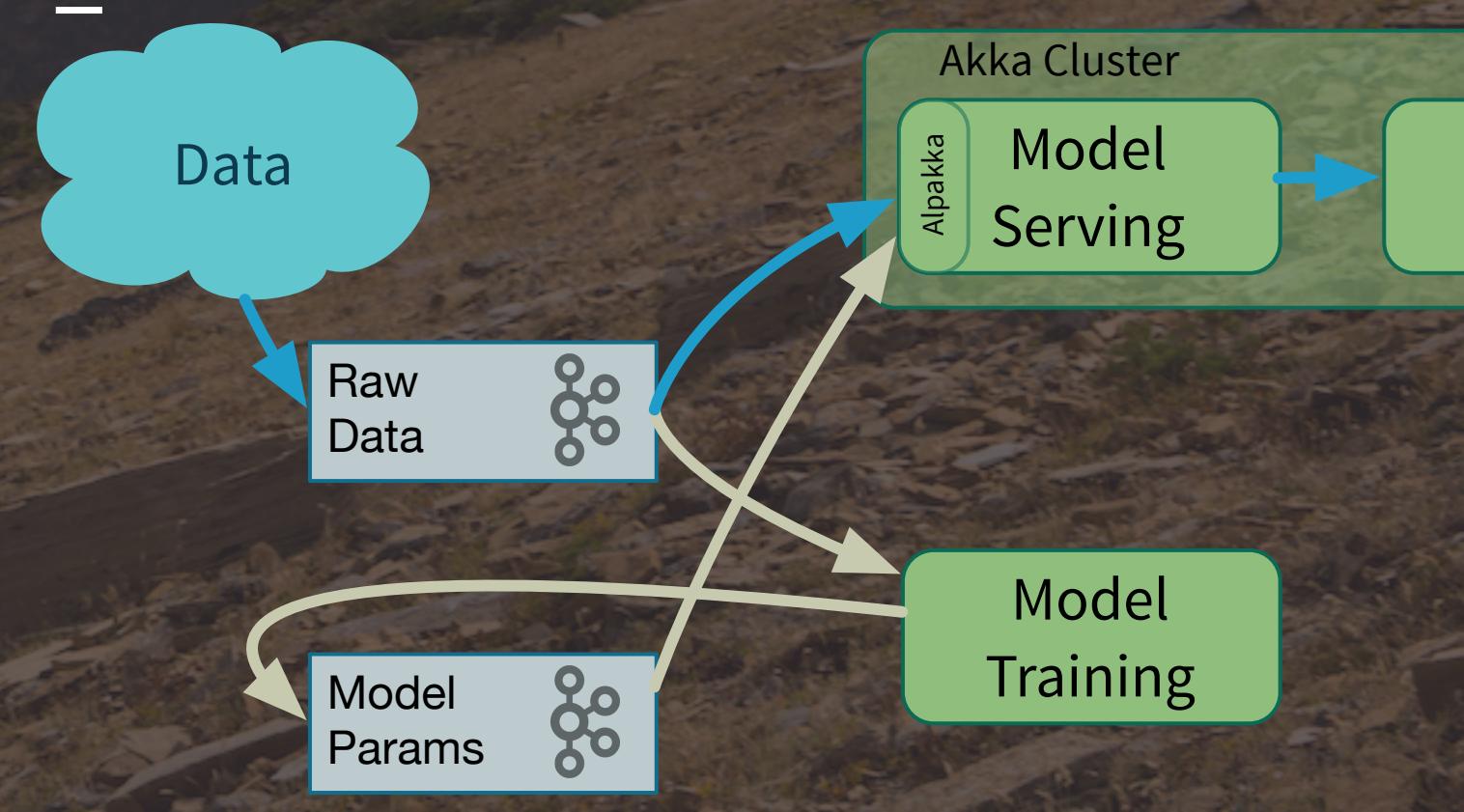


```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher
```

```
val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom "stage"
```

```
val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
    Subscriptions.topics(rawDataTopic))
  .map(input => DataRecord.parseBytes(input.value()))
  .collect{ case Success(data) => data }
```

```
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
    Subscriptions.topics(modelTopic))
  .map(input => Model.parseBytes(input.value()))
  .collect{ case Success(mod) => mod }
  .map(model => ModelImpl.findModel(model))
  .collect{ case Success(modImpl) => modImpl }
```

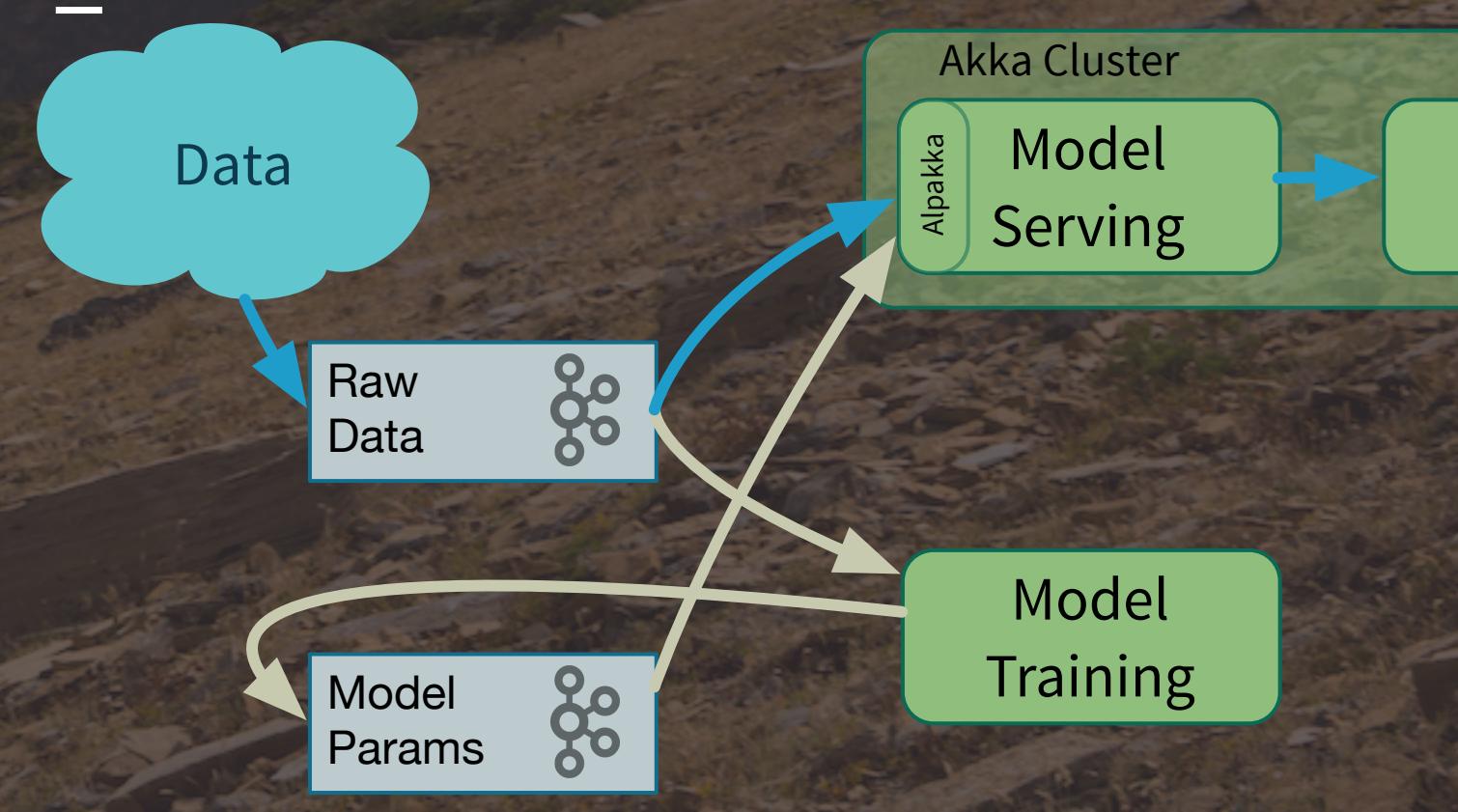


```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher
```

```
val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom "stage"
```

```
val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
    Subscriptions.topics(rawDataTopic))
  .map(input => DataRecord.parseBytes(input.value()))
  .collect{ case Success(data) => data }
```

```
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
    Subscriptions.topics(modelTopic))
  .map(input => Model.parseBytes(input.value()))
  .collect{ case Success(mod) => mod }
  .map(model => ModelImpl.findModel(model))
  .collect{ case Success(modImpl) => modImpl }
```

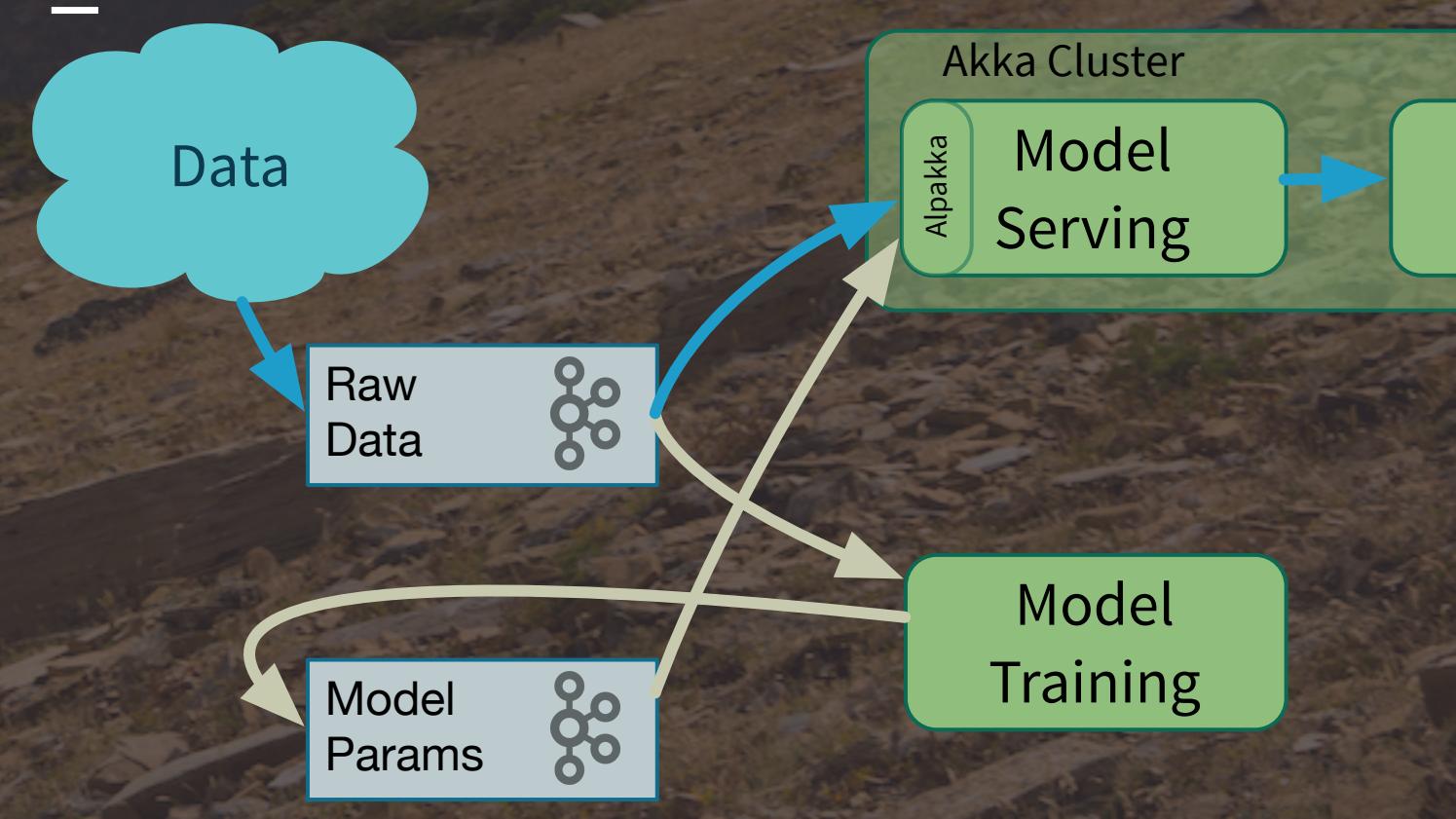


```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher
```

```
val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom "stage"
```

```
val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
    Subscriptions.topics(rawDataTopic))
  .map(input => DataRecord.parseBytes(input.value()))
  .collect{ case Success(data) => data }
```

```
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
    Subscriptions.topics(modelTopic))
  .map(input => Model.parseBytes(input.value()))
  .collect{ case Success(mod) => mod }
  .map(model => ModelImpl.findModel(model))
  .collect{ case Success(modImpl) => modImpl }
```

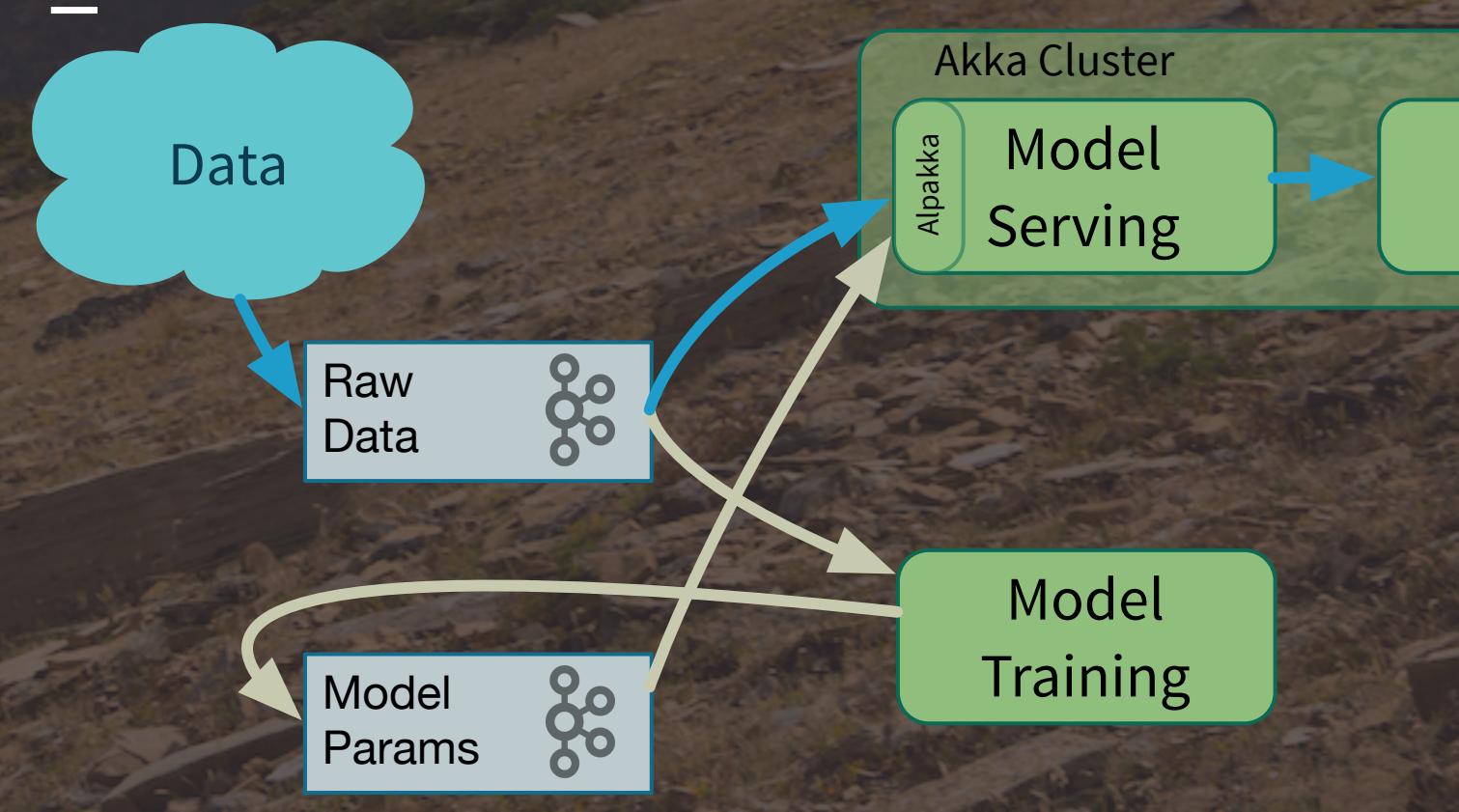


```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher
```

```
val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom "stage"
```

```
val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
    Subscriptions.topics(rawDataTopic))
  .map(input => DataRecord.parseBytes(input.value()))
  .collect{ case Success(data) => data }
```

```
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
    Subscriptions.topics(modelTopic))
  .map(input => Model.parseBytes(input.value()))
  .collect{ case Success(mod) => mod }
  .map(model => ModelImpl.findModel(model))
  .collect{ case Success(modImpl) => modImpl }
```

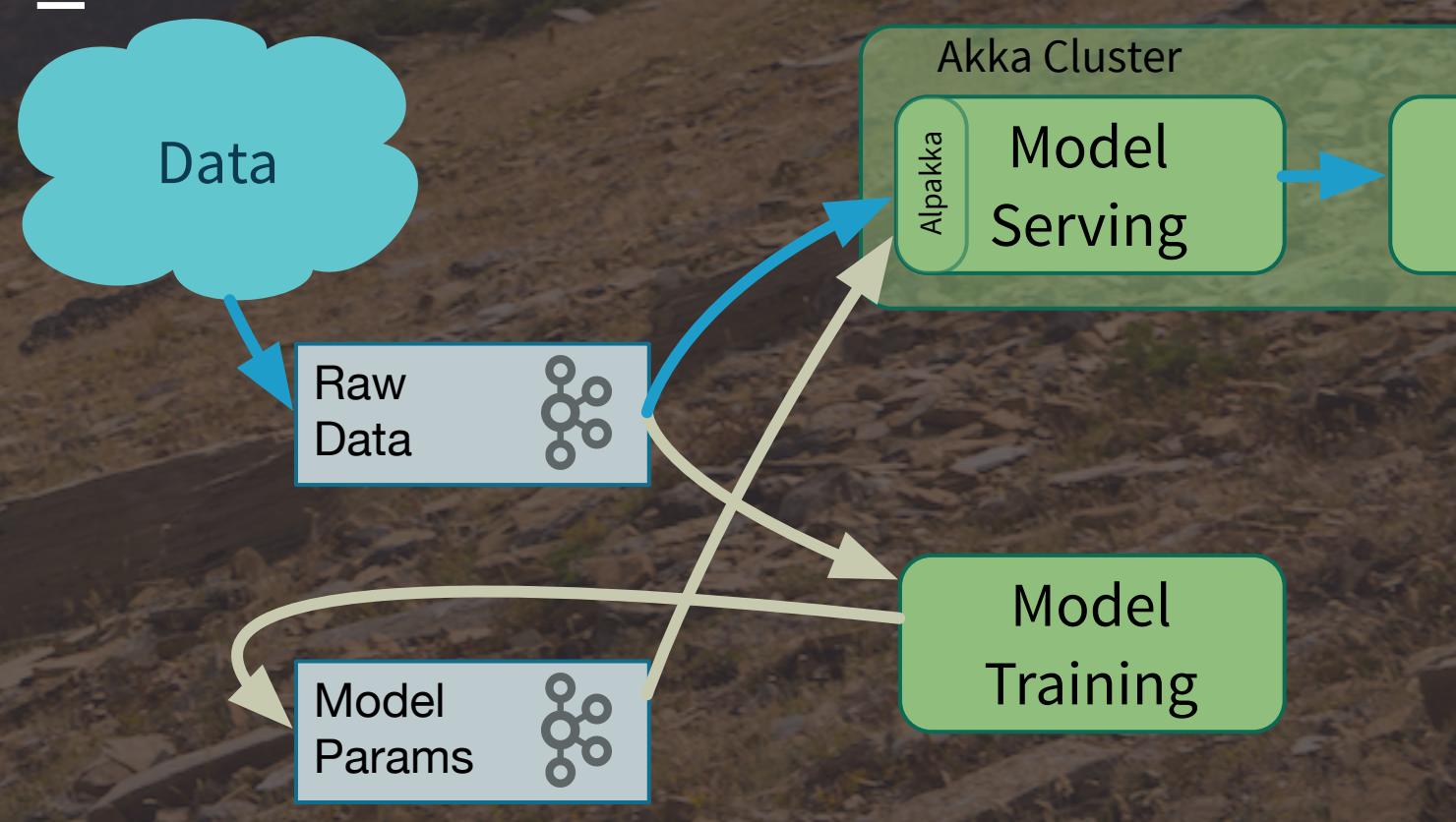


```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher
```

```
val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom “stage”
```

```
val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
    Subscriptions.topics(rawDataTopic))
  .map(input => DataRecord.parseBytes(input.value()))
  .collect{ case Success(data) => data }
```

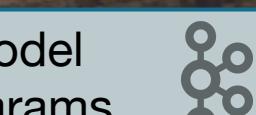
```
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
    Subscriptions.topics(modelTopic))
  .map(input => Model.parseBytes(input.value()))
  .collect{ case Success(mod) => mod }
  .map(model => ModelImpl.findModel(model))
  .collect{ case Success(modImpl) => modImpl }
```



```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
import spray.actor.executionContext -> system.dispatcher
case class ModelScoringStage(scorer: ...) extends
GraphStageWithMaterializedValue[..., ...] {

    val dataRecordIn = Inlet[Record]("dataRecordIn")
    val modelRecordIn = Inlet[ModelImpl]("modelRecordIn")
    val scoringResultOut = Outlet[ScoredRecord]("scoringOut")

    ...
    setHandler(dataRecordIn, new InHandler {
        override def onPush(): Unit = {
            val record = grab(dataRecordIn)
            val newRecord = new ScoredRecord(scorer.score(record), record))
            push(scoringResultOut, Some(newRecord))
            pull(dataRecordIn)
        }
    })
    ...
}
.collect case Success(mod) -> mod
.map(model => ModelImpl.findModel(model))
.collect{ case Success(modImpl) => modImpl }
```



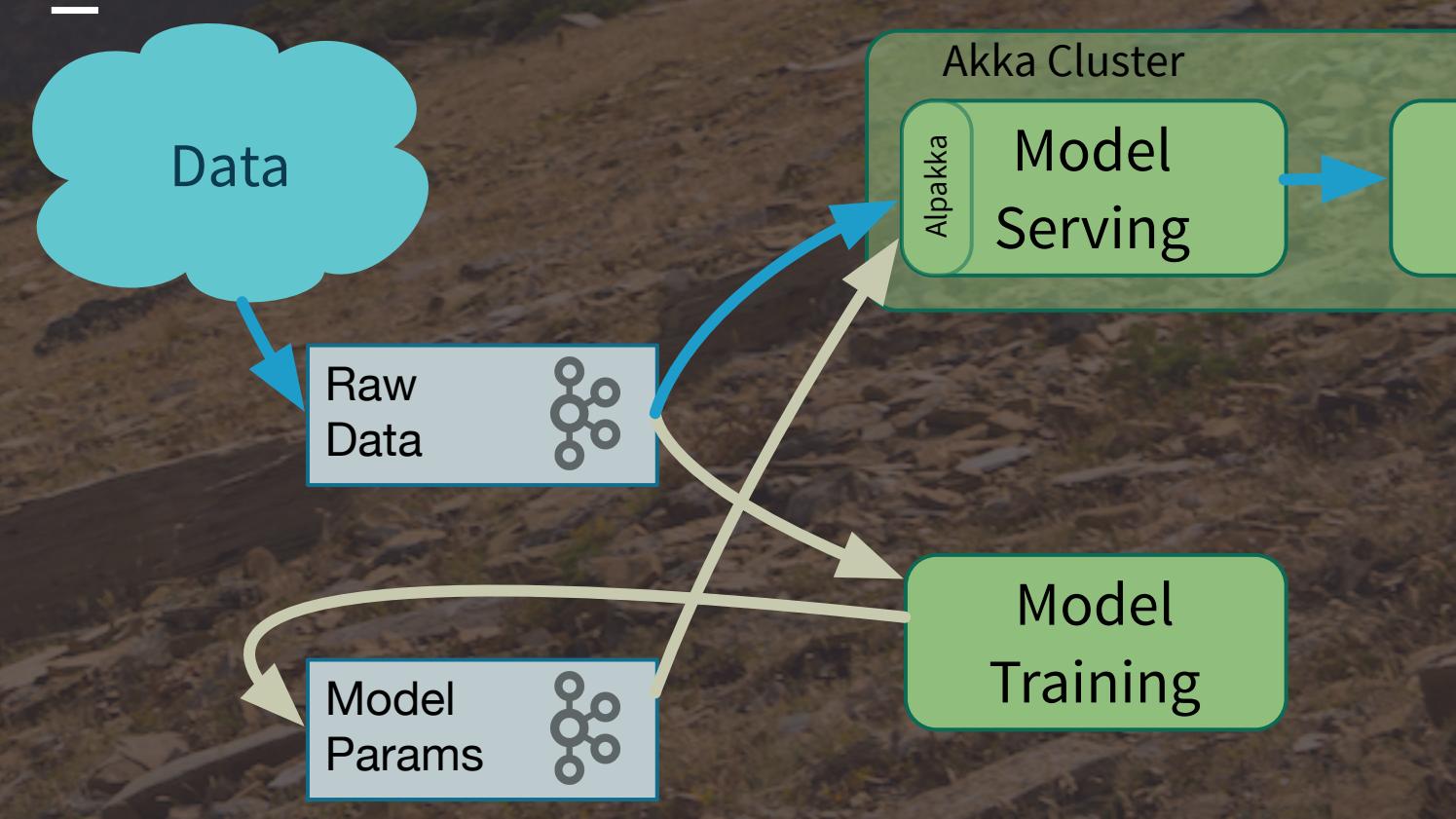
Model
Training

```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher
```

```
val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom "stage"
```

```
val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
    Subscriptions.topics(rawDataTopic))
  .map(input => DataRecord.parseBytes(input.value()))
  .collect{ case Success(data) => data }
```

```
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
    Subscriptions.topics(modelTopic))
  .map(input => Model.parseBytes(input.value()))
  .collect{ case Success(mod) => mod }
  .map(model => ModelImpl.findModel(model))
  .collect{ case Success(modImpl) => modImpl }
```



```

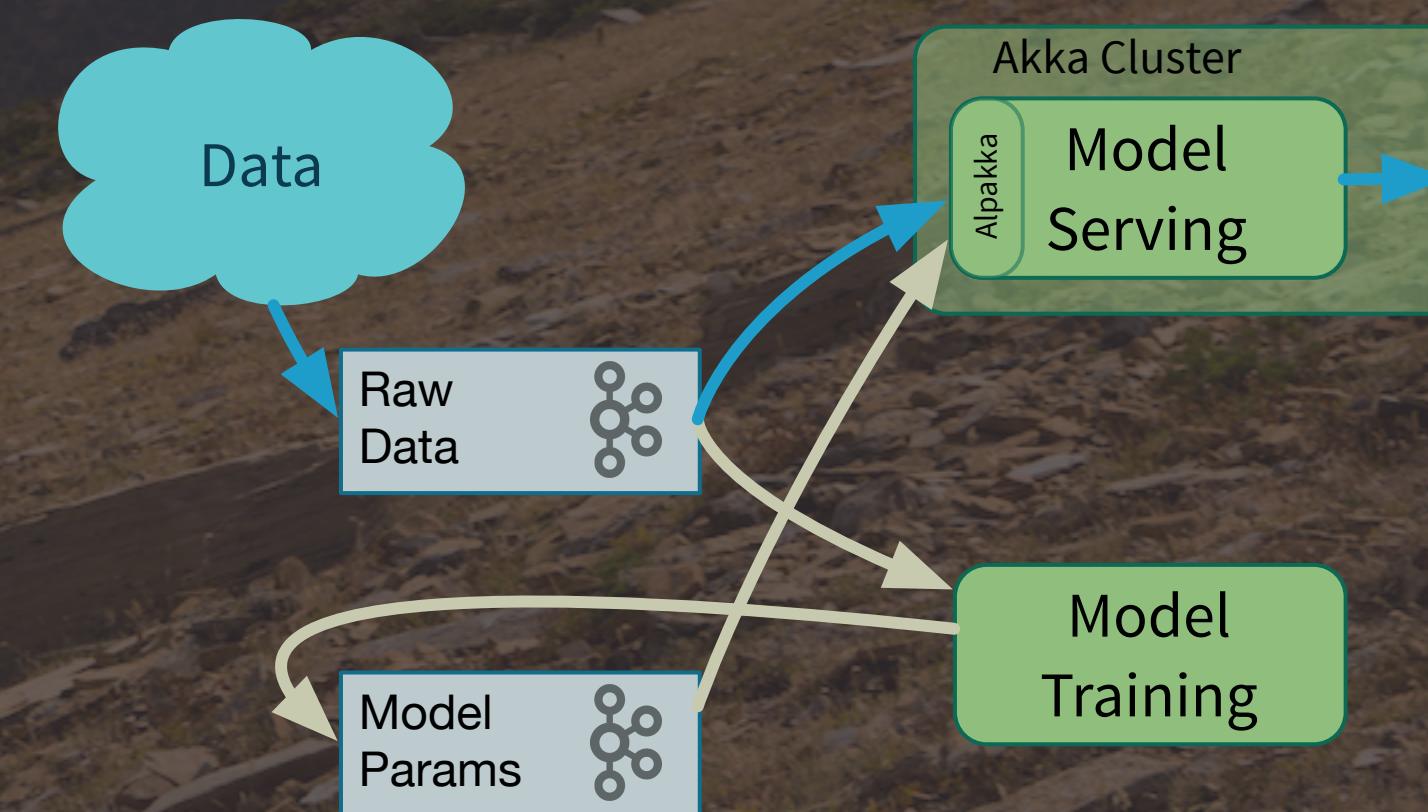
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
    Subscriptions.topics(modelTopic))
  .map(input => Model.parseBytes(input.value()))
  .collect{ case Success(mod) => mod }
  .map(model => ModelImpl.findModel(model))
  .collect{ case Success(modImpl) => modImpl }
  .foreach(modImpl => modelProcessor.setModel(modImpl))
modelStream.to(Sink.ignore).run() // No “sinking” required; just run

```

```

dataStream
  .viaMat(modelScoringStage)(Keep.right)
  .map(result => new ProducerRecord[Array[Byte], ScoredRecord](
    scoredRecordsTopic, result))
  .runWith(Producer.plainSink(producerSettings))

```



```

.collect{ case Success(data) -> data }

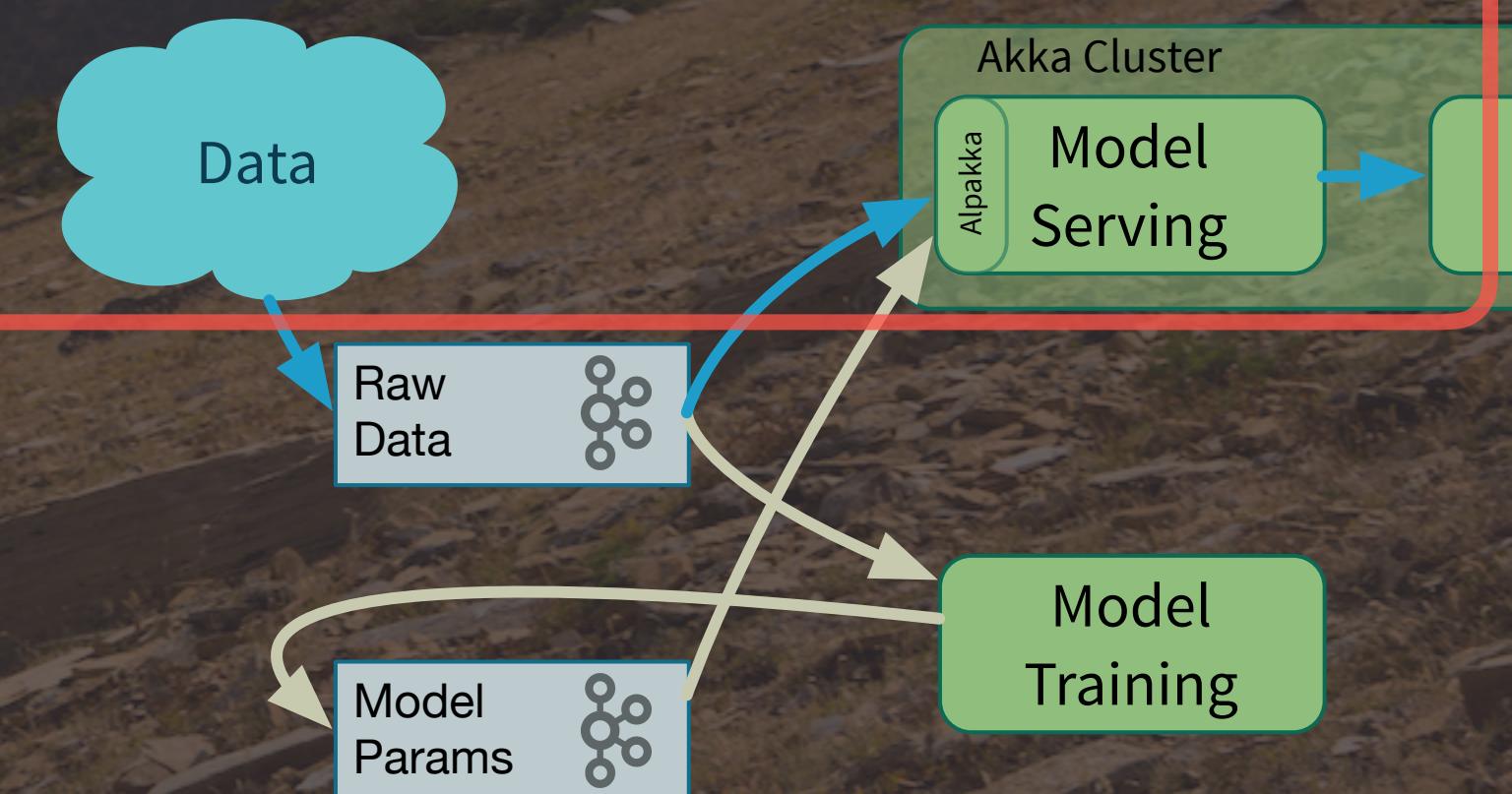
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
    Subscriptions.topics(modelTopic))
  .map(input => Model.parseBytes(input.value()))
  .collect{ case Success(mod) => mod }
  .map(model => ModelImpl.findModel(model))
  .collect{ case Success(modImpl) => modImpl }
  .foreach(modImpl => modelProcessor.setModel(modImpl))
modelStream.to(Sink.ignore).run() // No “sinking” required; just run

```

```

dataStream
  .viaMat(modelScoringStage)(Keep.right)
  .map(result => new ProducerRecord[Array[Byte], ScoredRecord](
    scoredRecordsTopic, result))
  .runWith(Producer.plainSink(producerSettings))

```



A wide-angle photograph of a mountain lake at sunset. The sky is filled with dramatic, layered clouds. The lake's surface is extremely still, creating a perfect mirror that reflects the surrounding landscape. On the left, a dense forest of tall evergreen trees stands along the shore. On the right, a massive, steep mountain slope covered in green coniferous forests rises sharply. The lighting is low, with the sun setting behind the clouds, casting a warm glow and long shadows.

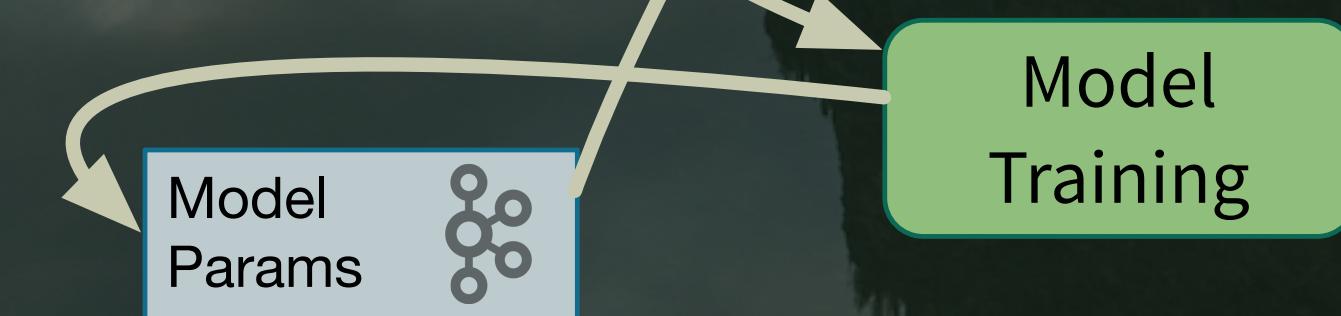
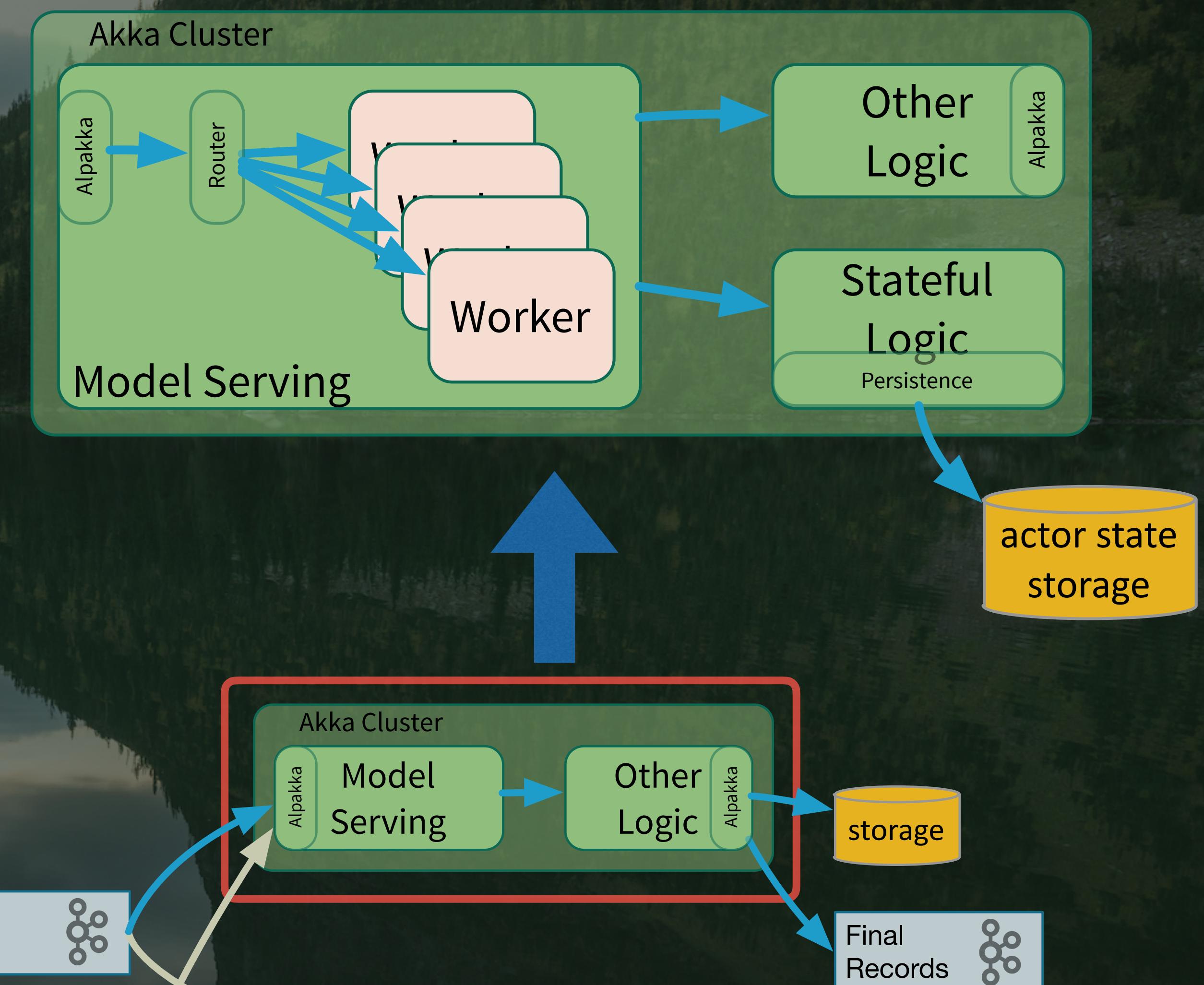
Other Concerns

- Scale scoring with workers and routers, across a cluster

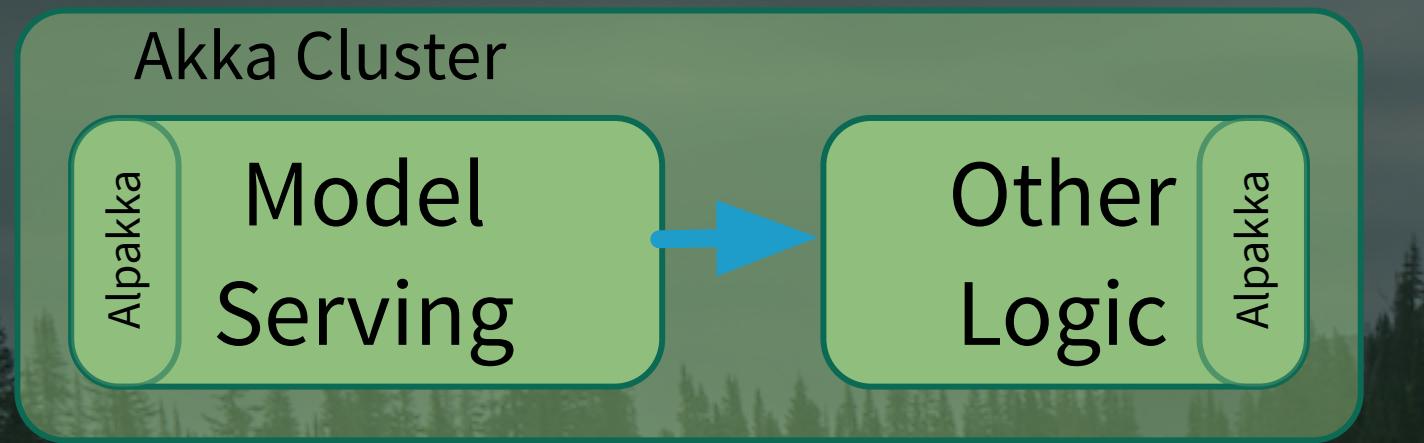
- Persist actor state with Akka Persistence

- Connect to *almost anything* with Alpakka

- *StreamRefs* for distributed streams (new!)



Go Direct or Through Kafka?



VS.



- Extremely low latency
- Minimal I/O and memory overhead. No marshaling overhead
- Hard to scale, evolve independently
- Higher latency (network, queue depth)
- Higher I/O and processing (marshaling) overhead
- Easy independent scalability, evolution

Go Direct or Through Kafka?



- *Reactive Streams* back pressure
- “Direct” coupling between sender and receiver, but actually uses a URL abstraction
- Very deep buffer (partition limited by disk size)
- Strong decoupling - M producers, N consumers, completely disconnected

Wrapping Up...





Free as in 🍺

lightbend.com/fast-data-platform

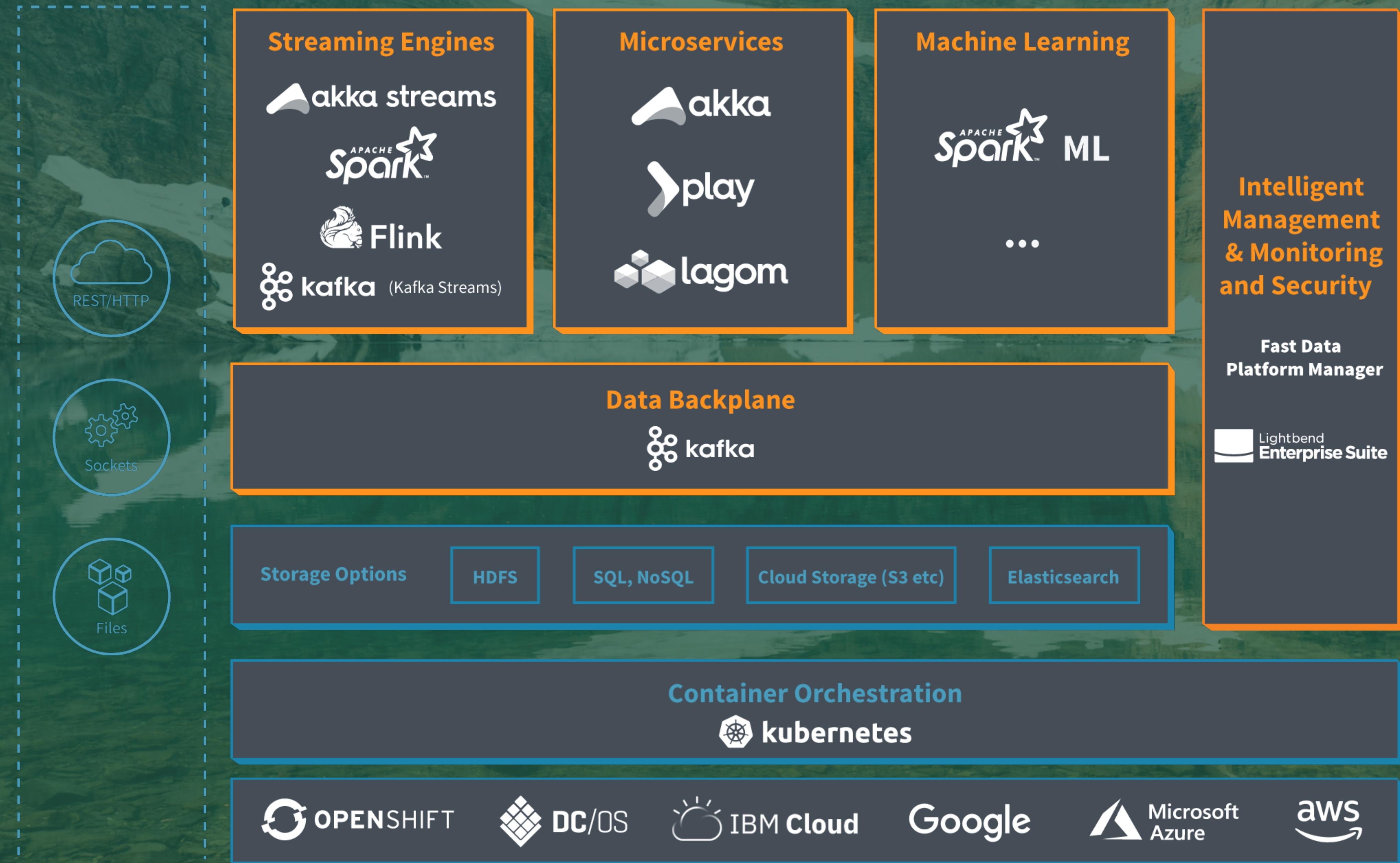
O'REILLY®

Fast Data Architectures for Streaming Applications

Getting Answers Now from
Data Sets that Never End

Dean Wampler

Compliments of
Lightbend



Questions?

A wide-angle photograph of a mountain lake. The foreground is filled with the dark, clear water of the lake, which perfectly reflects the surrounding environment. In the background, massive, rugged mountains rise, their slopes covered in a mix of dark rock and patches of white snow. The sky above is a bright, clear blue.

dean.wampler@lightbend.com
lightbend.com/fast-data-platform
polyglotprogramming.com/talks