Streaming Data Microservices

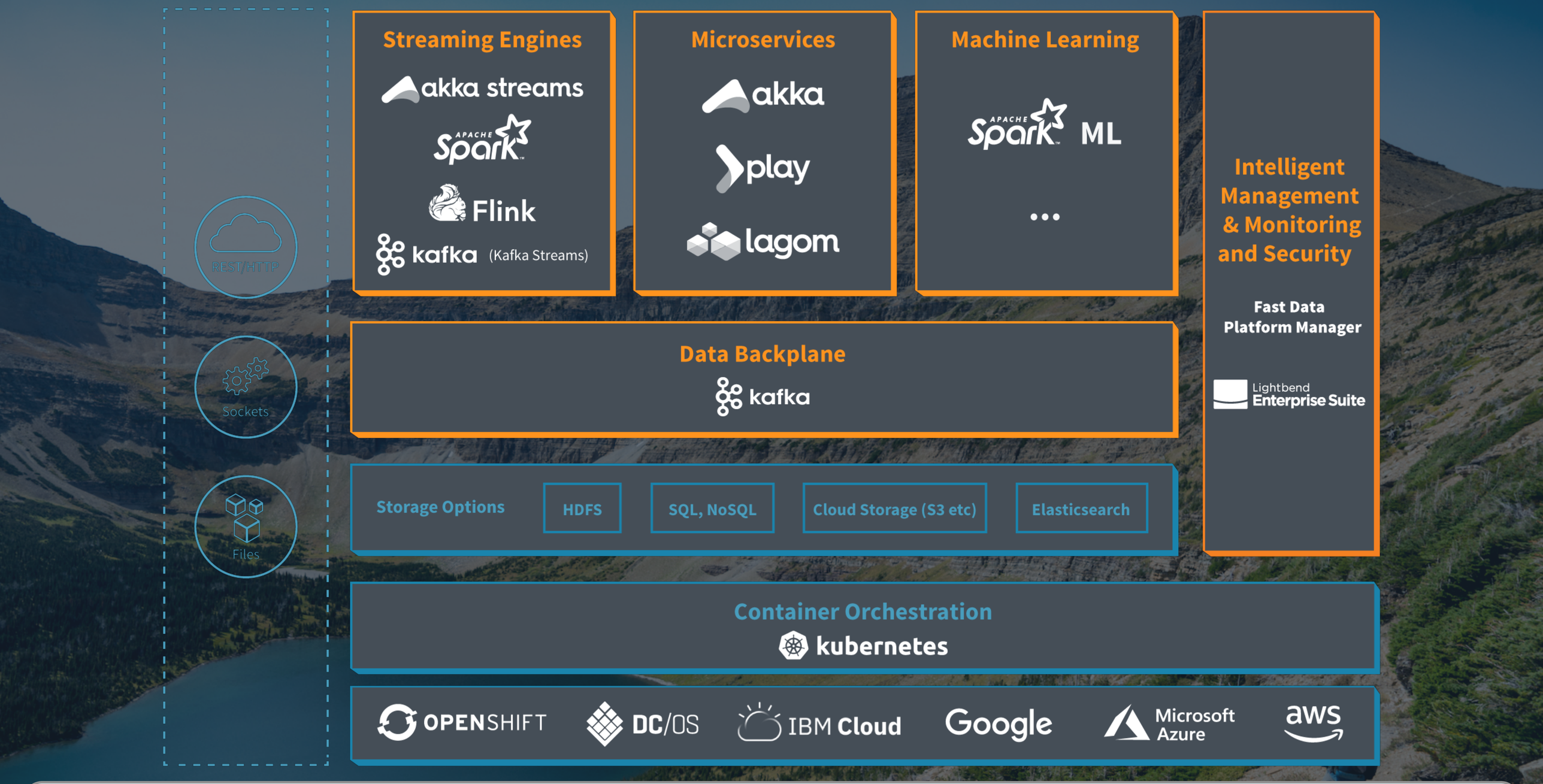
With Akka Streams and Kafka Streams

Dean Wampler, Ph.D.
dean@lightbend.com
@deanwampler

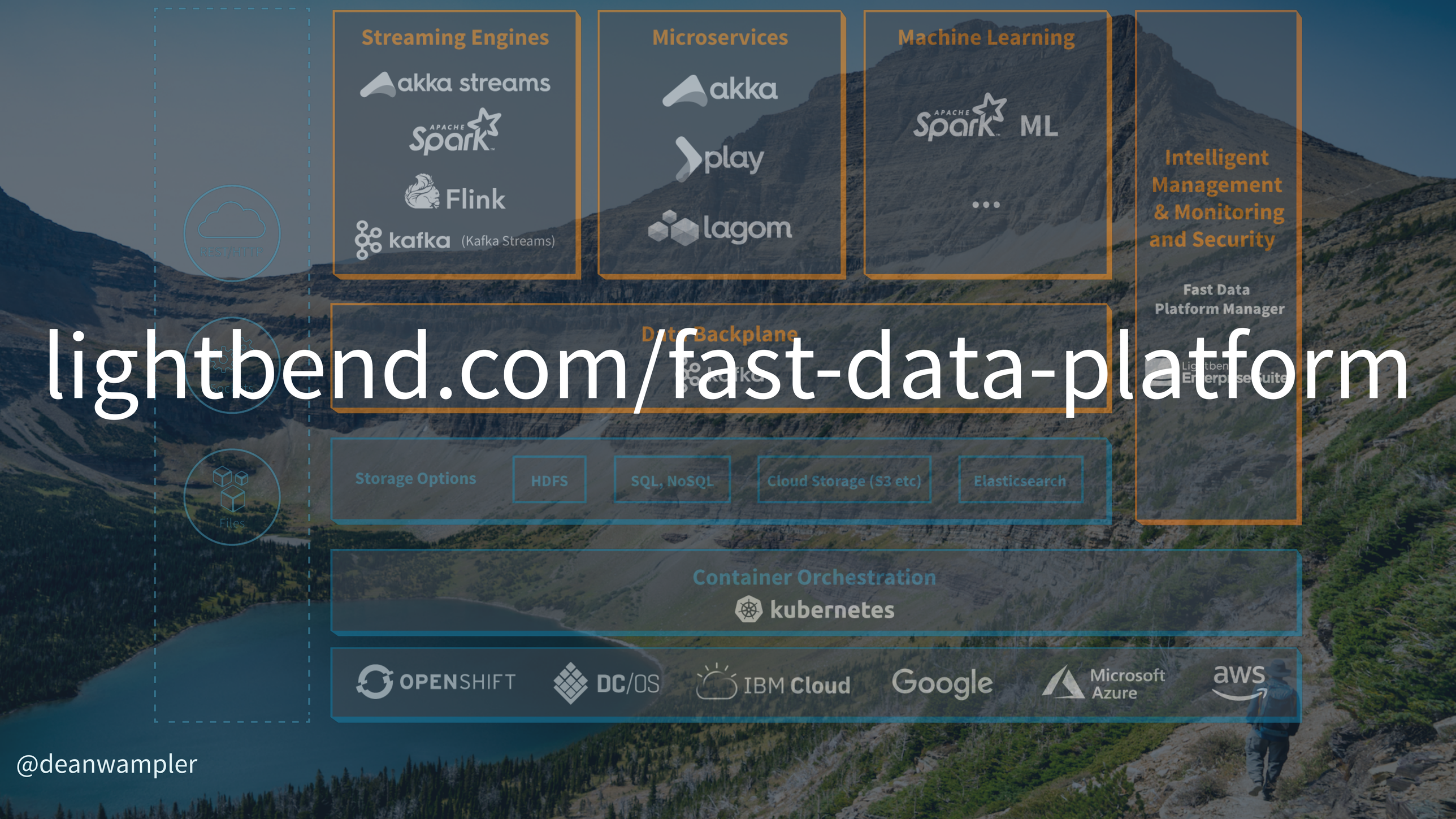
Lightbend

Who am I?

@deanwampler



I lead the Lightbend Fast Data Platform; unified streaming data & microservices



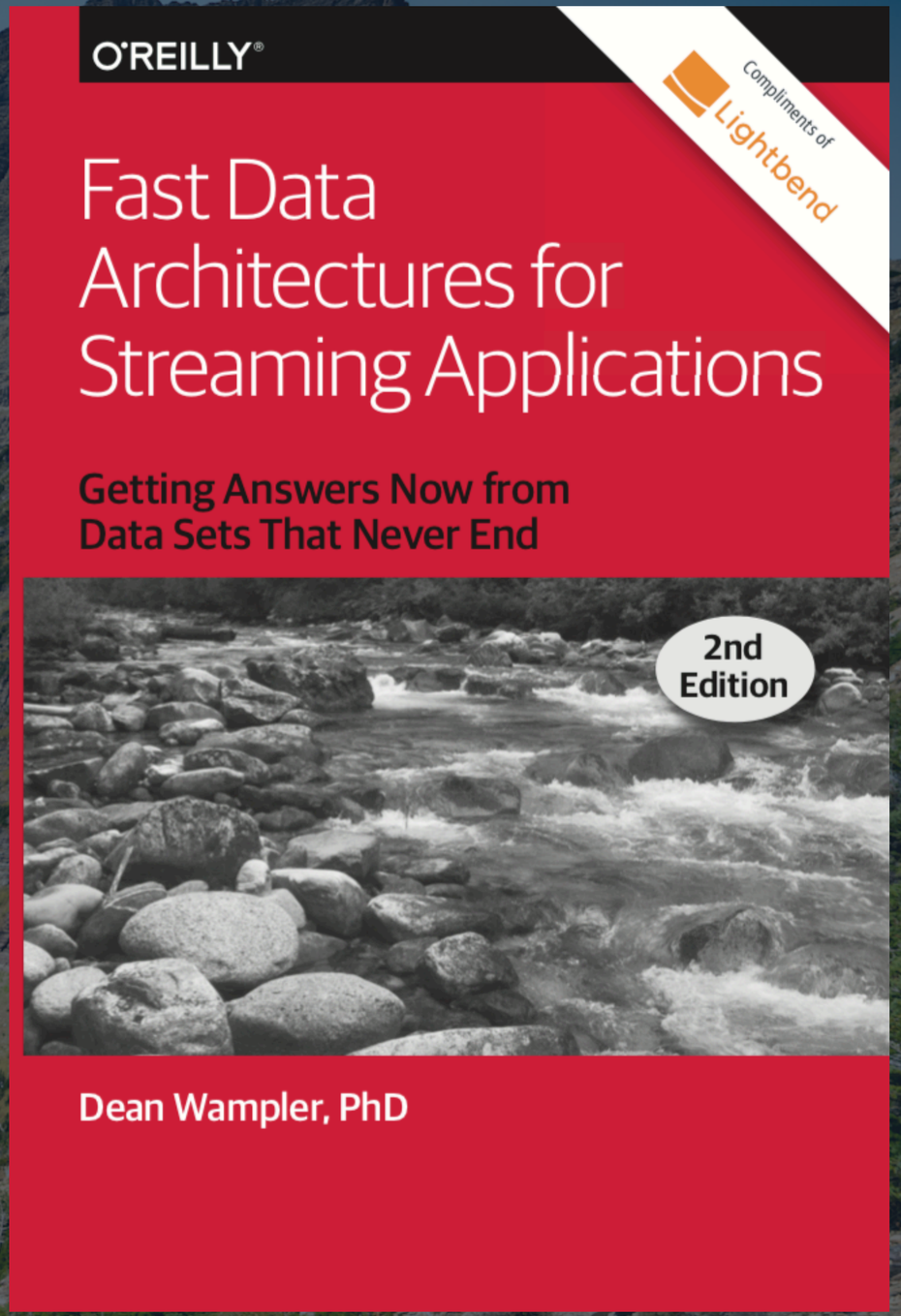
lightbend.com/fast-data-platform

@deanwampler

Free as in 🍺

New second edition!

lbnd.io/fast-data-book



@deanwampler

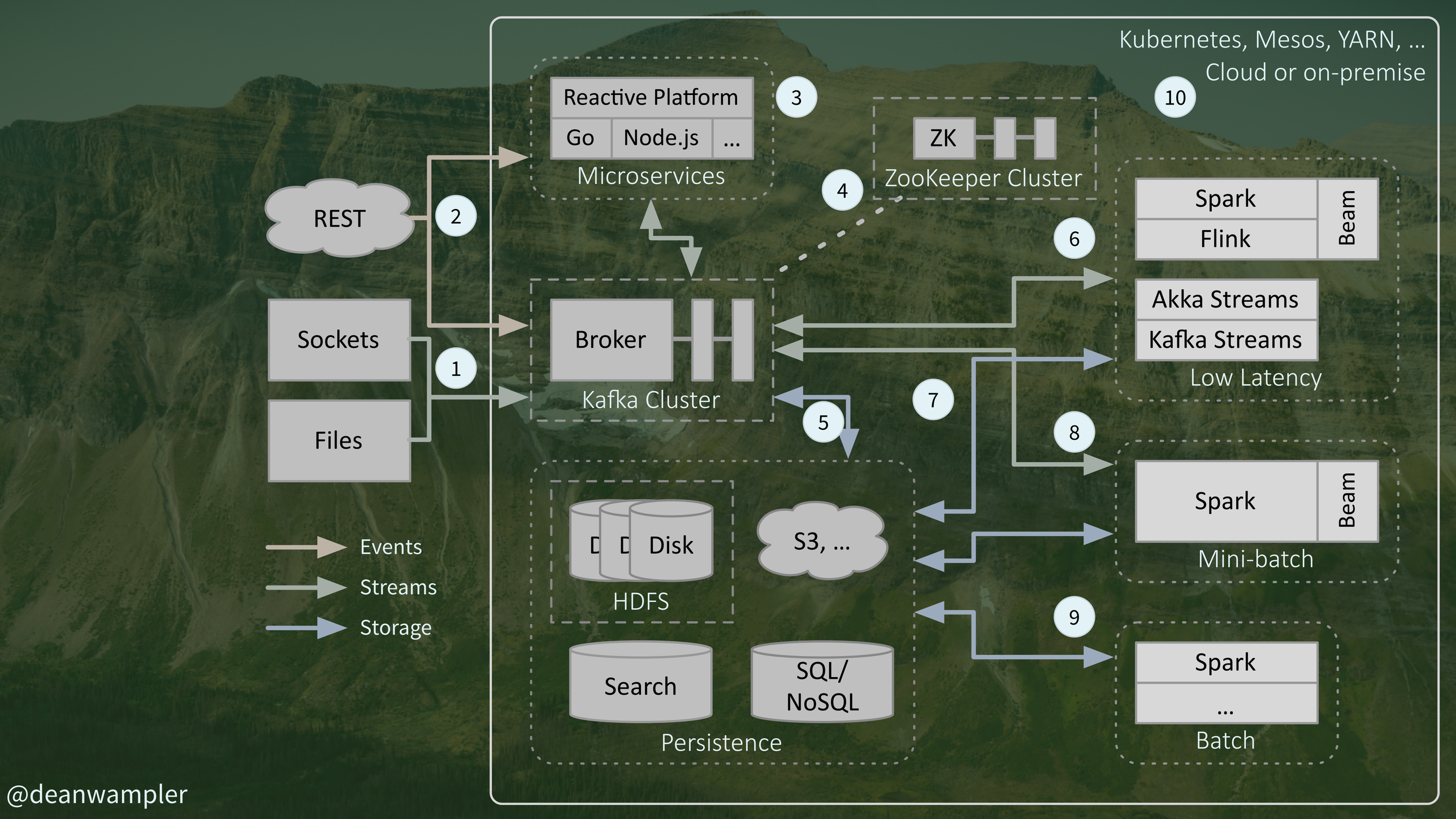

TL;DR
@deanwampler

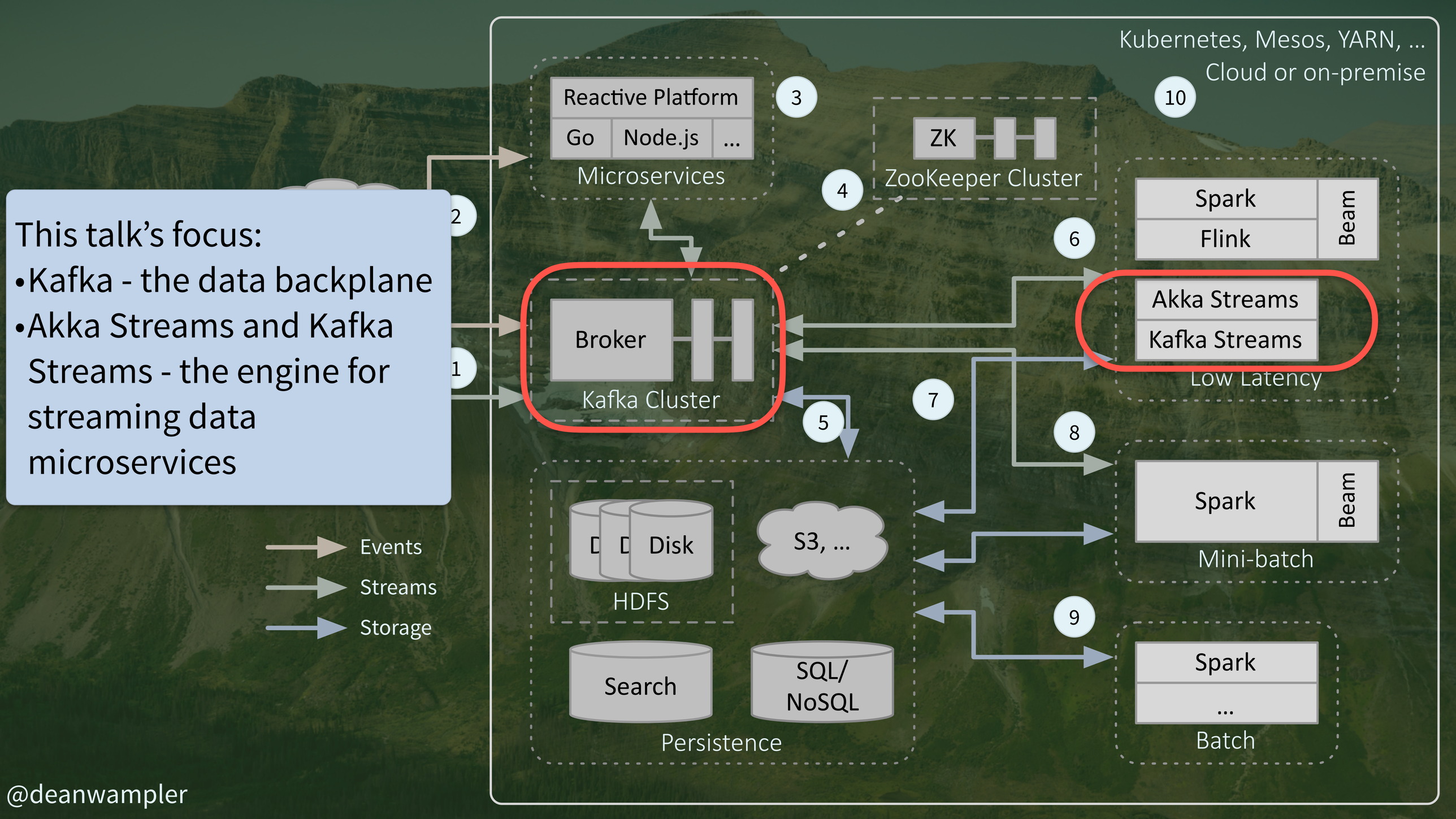
TL;DR

For streaming data:

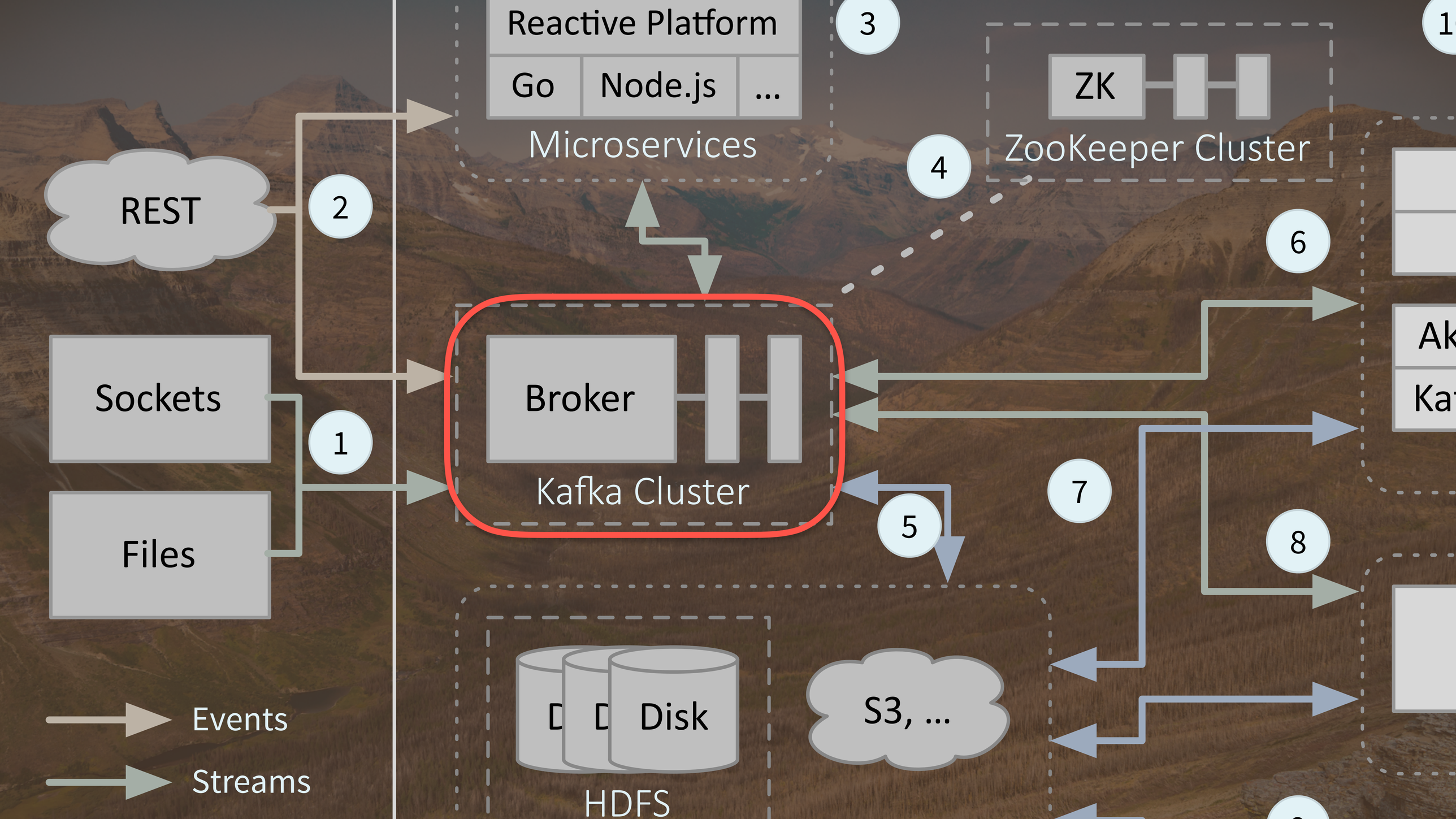
- Use Kafka as the “backplane”
- Use Spark or Flink when you need massive scale…
- Use microservice *libraries*, like Akka Streams and Kafka Streams for everything else…

Streaming architectures (from the report)


Kubernetes, Mesos, YARN, ...
Cloud or on-premise
10
Reactive Platform
Go
Node.js
...
Microservices
3
ZK
ZooKeeper Cluster
4
REST
2
Sockets
1
Files
Broker
Kafka Cluster
5
6
Spark
Flink
Beam
Akka Streams
Kafka Streams
Low Latency
7
8
Spark
Beam
Mini-batch
Disk
HDFS
S3, ...
9
Search
SQL/
NoSQL
Persistence
Spark
...
Batch
Events
Streams
Storage
@deanwampler

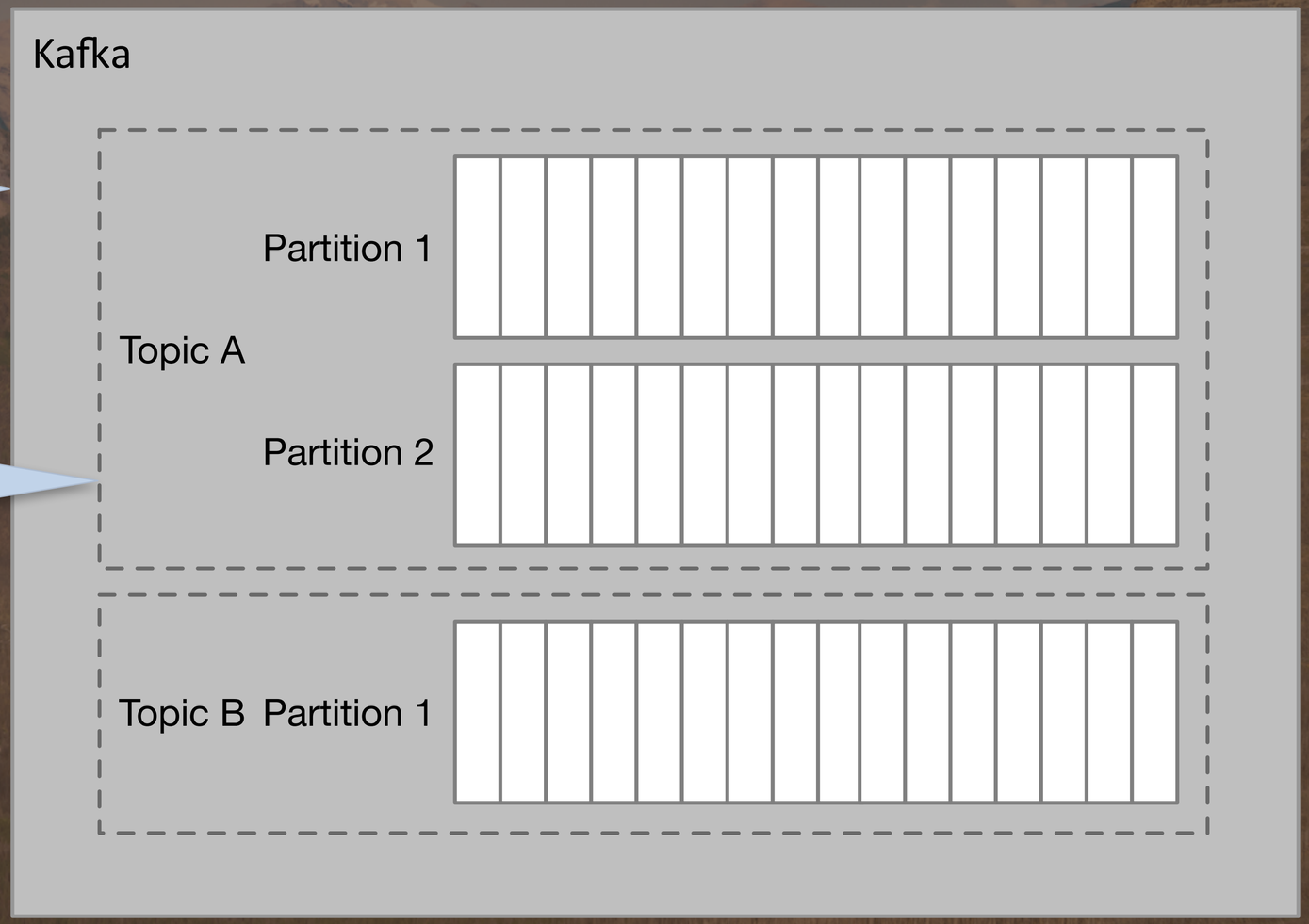

This talk's focus:
•Kafka - the data backplane
•Akka Streams and Kafka Streams - the engine for streaming data microservices
Kubernetes, Mesos, YARN, ...
Cloud or on-premise
Reactive Platform
Go
Node.js
...
Microservices
ZK
ZooKeeper Cluster
Broker
Kafka Cluster
Spark
Flink
Beam
Akka Streams
Kafka Streams
Low Latency
Spark
Beam
Mini-batch
Disk
HDFS
S3, ...
Search
SQL/
NoSQL
Persistence
Spark
...
Batch
1
2
3
4
5
6
7
8
9
10
Events
Streams
Storage
@deanwampler


Why Kafka?
@deanwampler

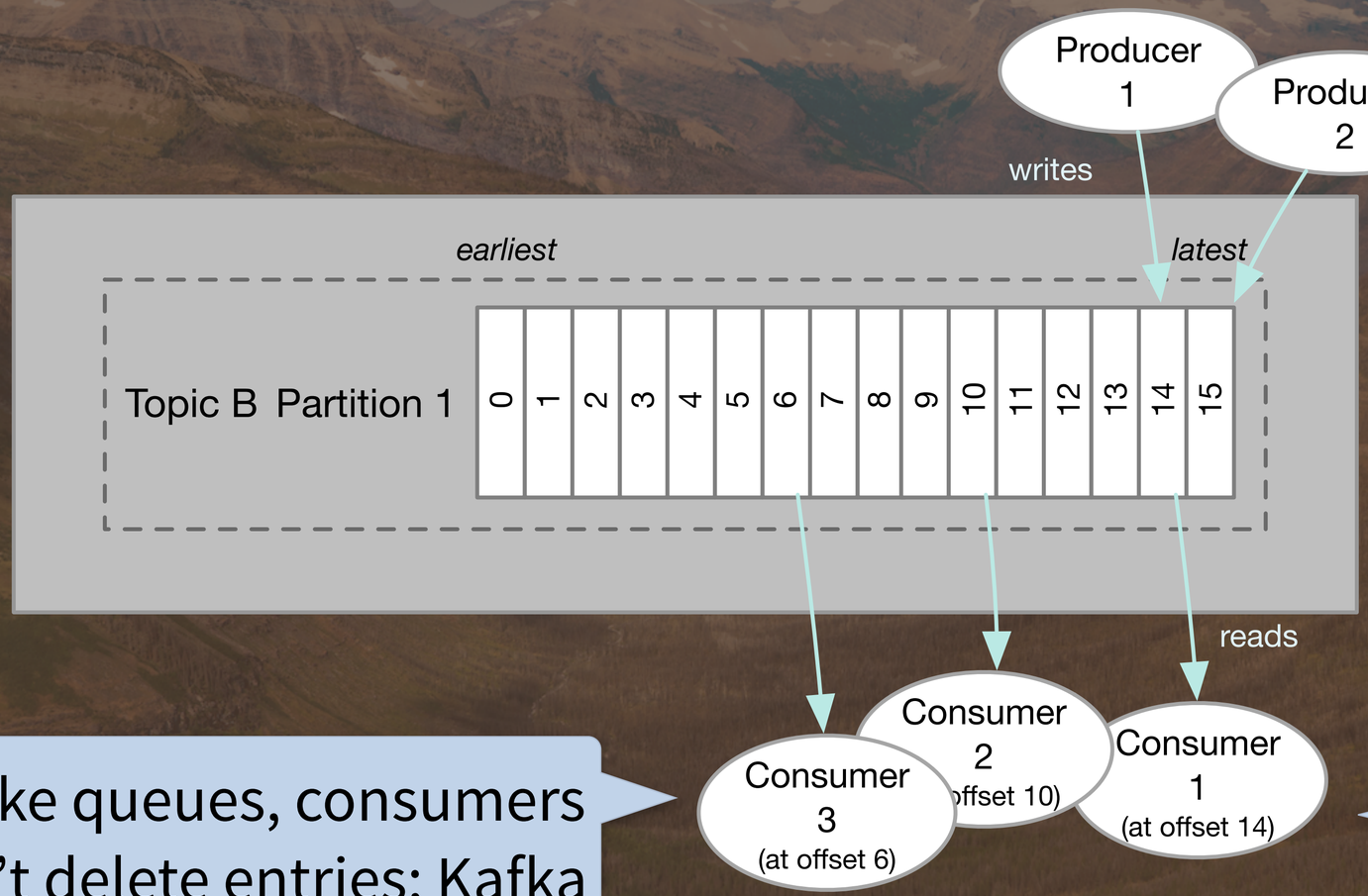

Reactive Platform
Go
Node.js
...
Microservices
3
ZK
ZooKeeper Cluster
4
REST
2
6
Sockets
Broker
1
Kafka Cluster
Files
5
7
8
Disk
HDFS
S3, ...
Events
Streams

Organized into topics

Topics are partitioned, replicated, and distributed



Logs, *not* queues!

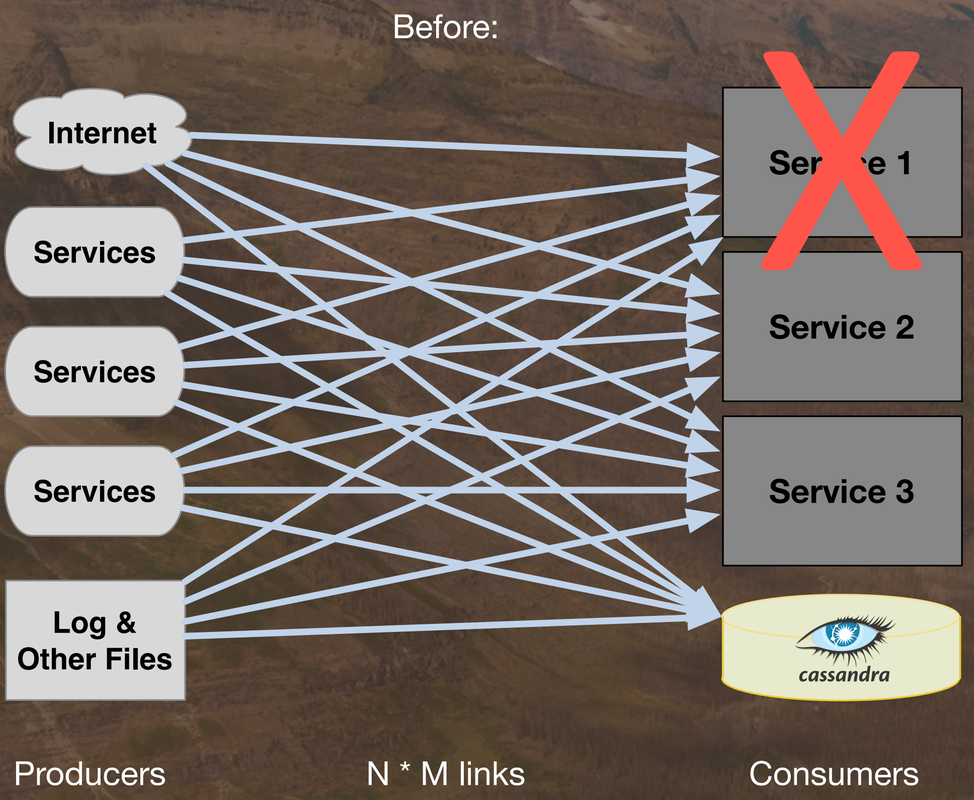


M Producers

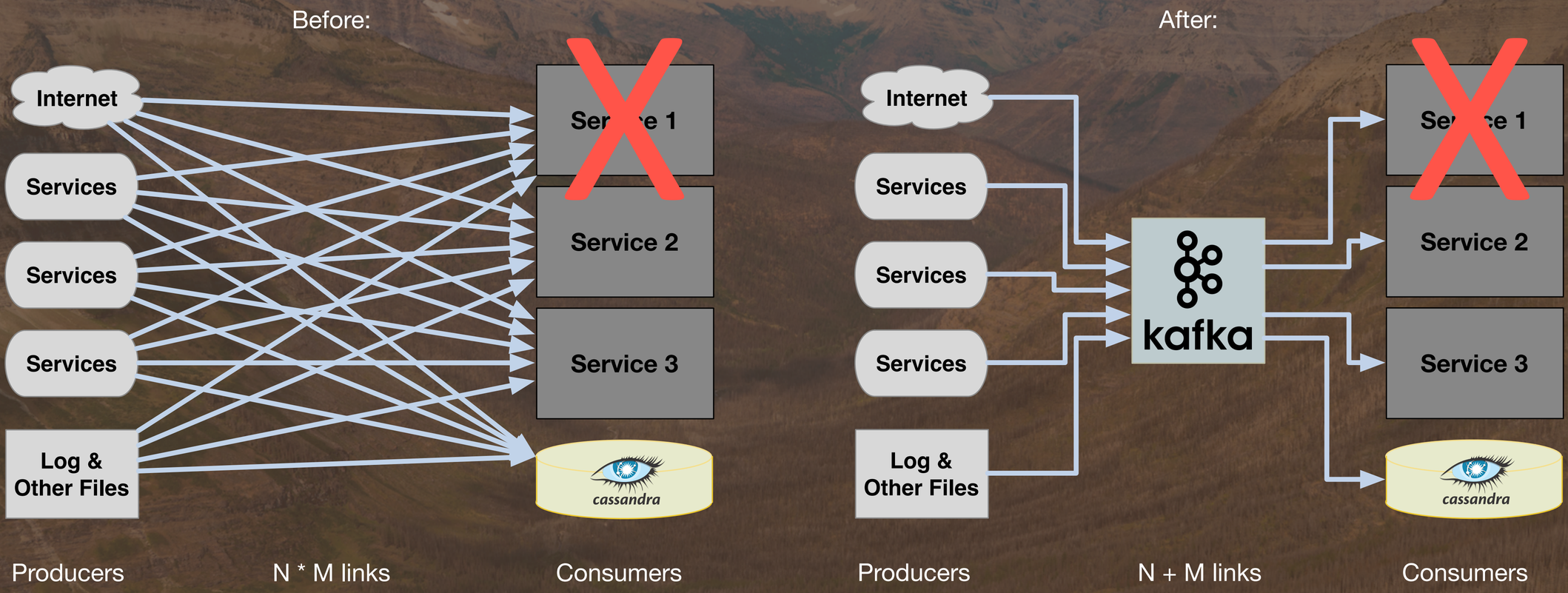
N Consumers, who start reading where they want

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

Architectural Benefits of Kafka

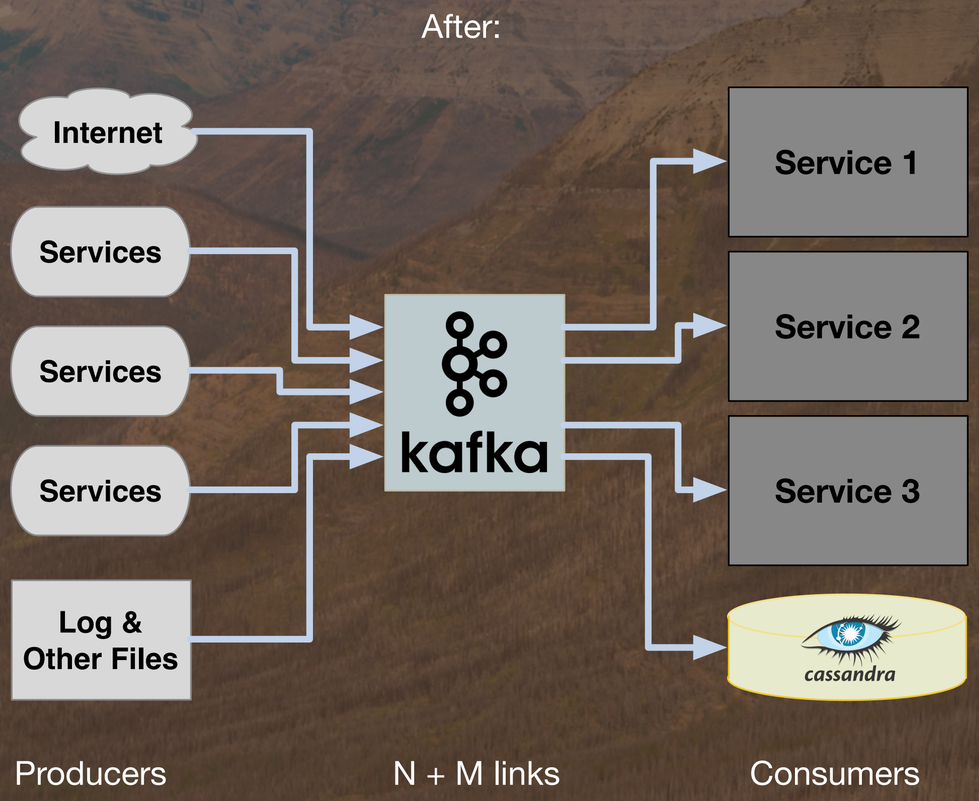


Architectural Benefits of Kafka



Architectural Benefits of Kafka

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

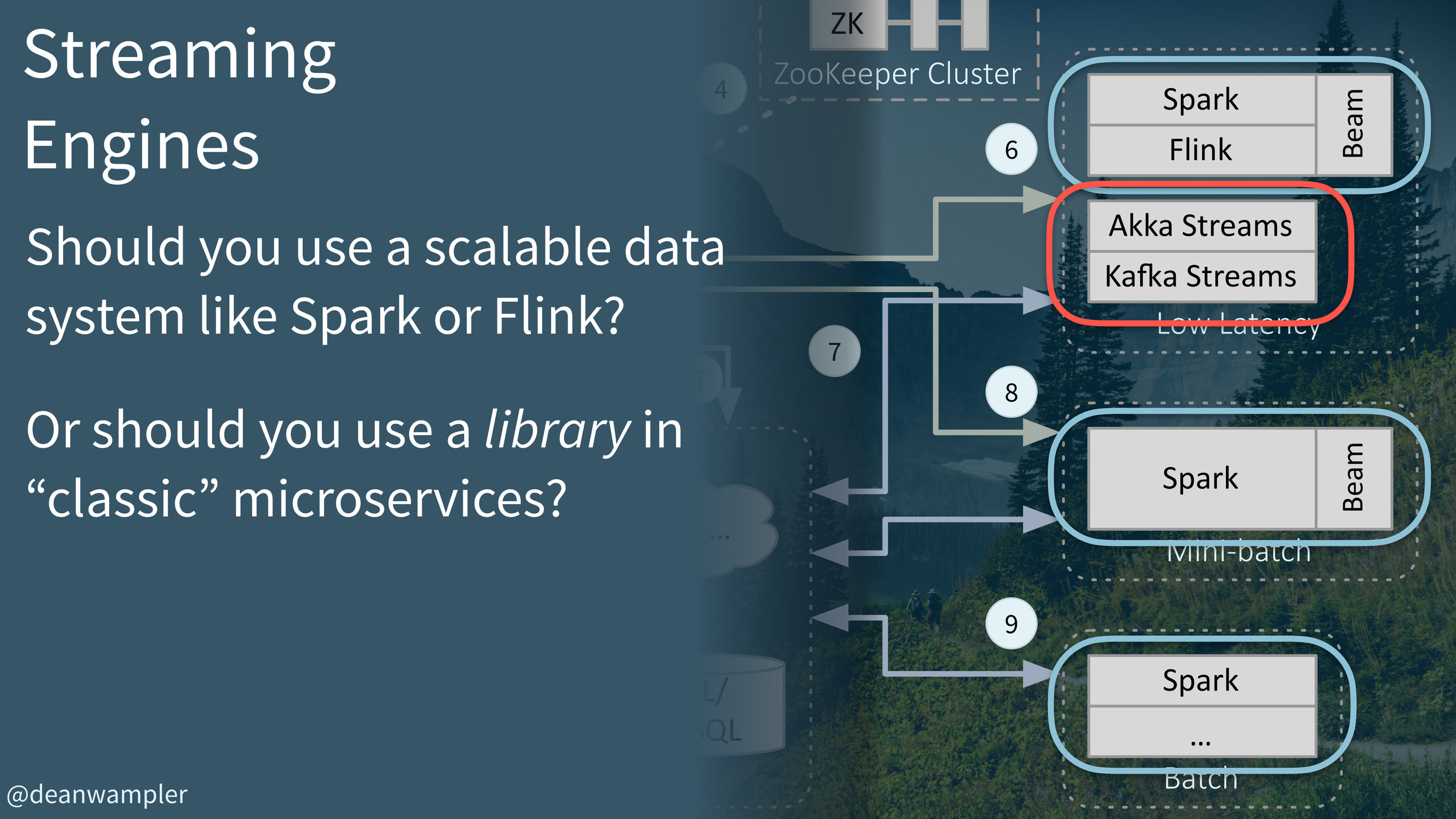


Streaming Engines

Streaming Engines

Should you use a scalable data system like Spark or Flink?

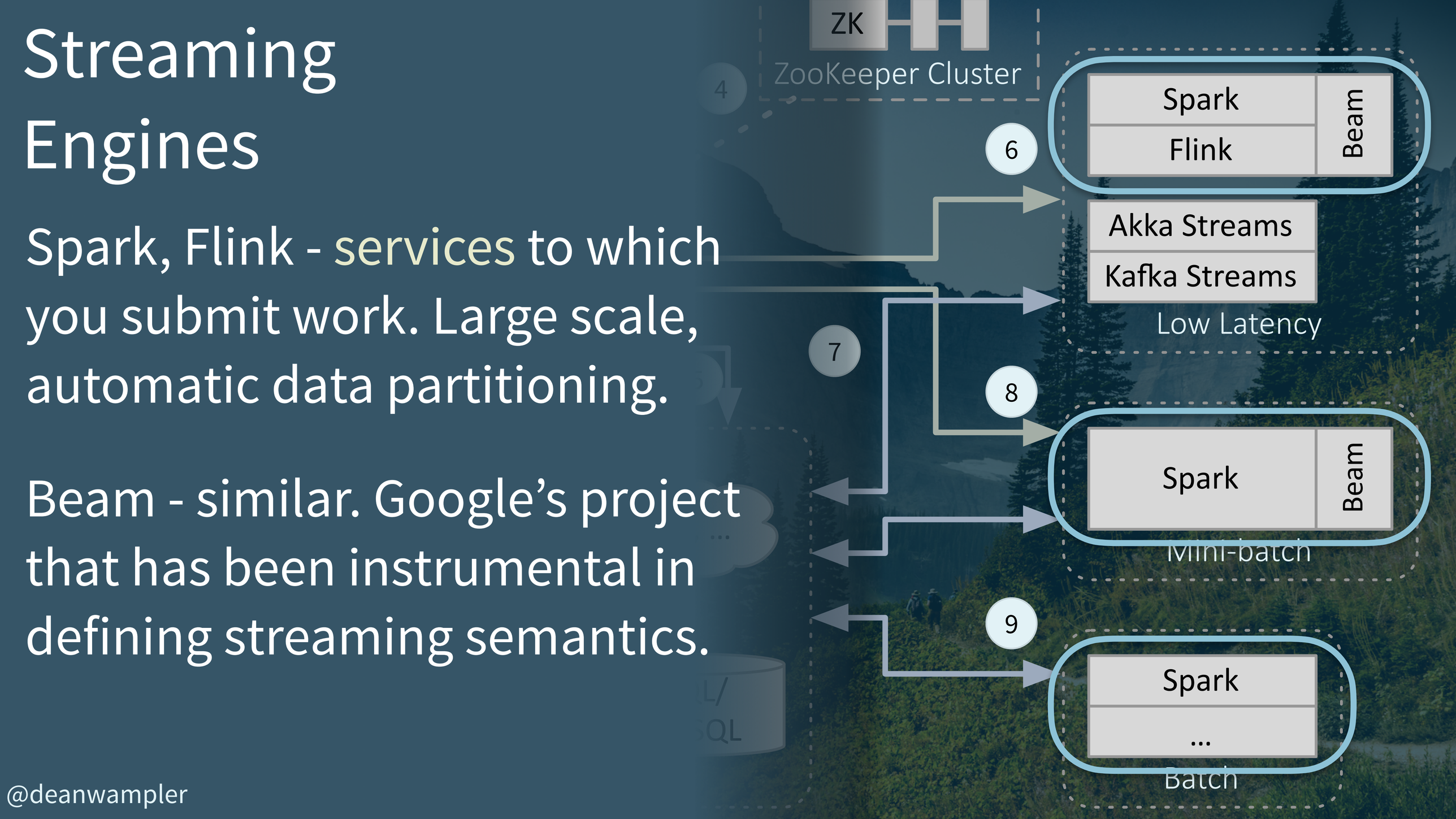
Or should you use a *library* in "classic" microservices?



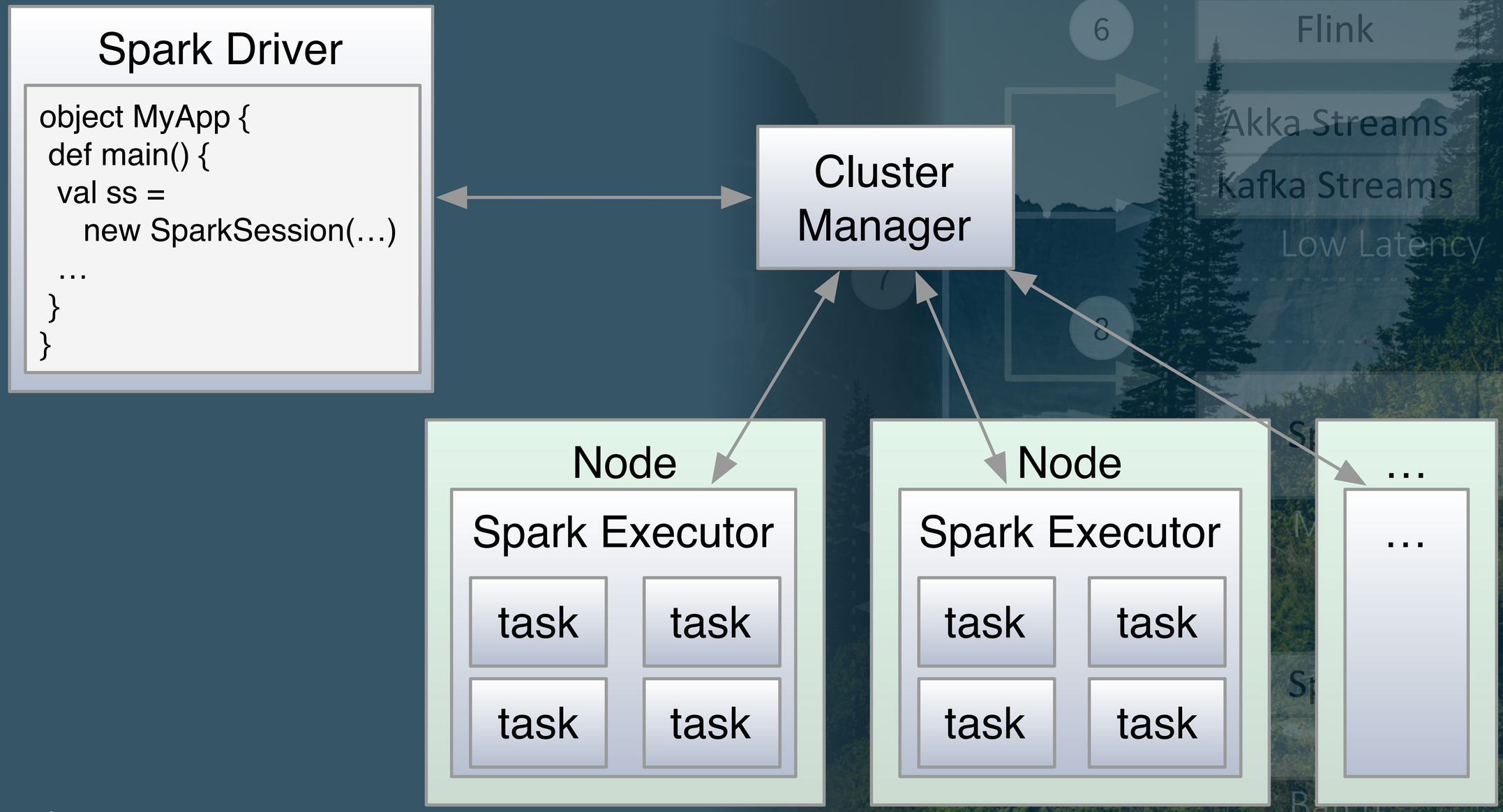
Streaming Engines

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

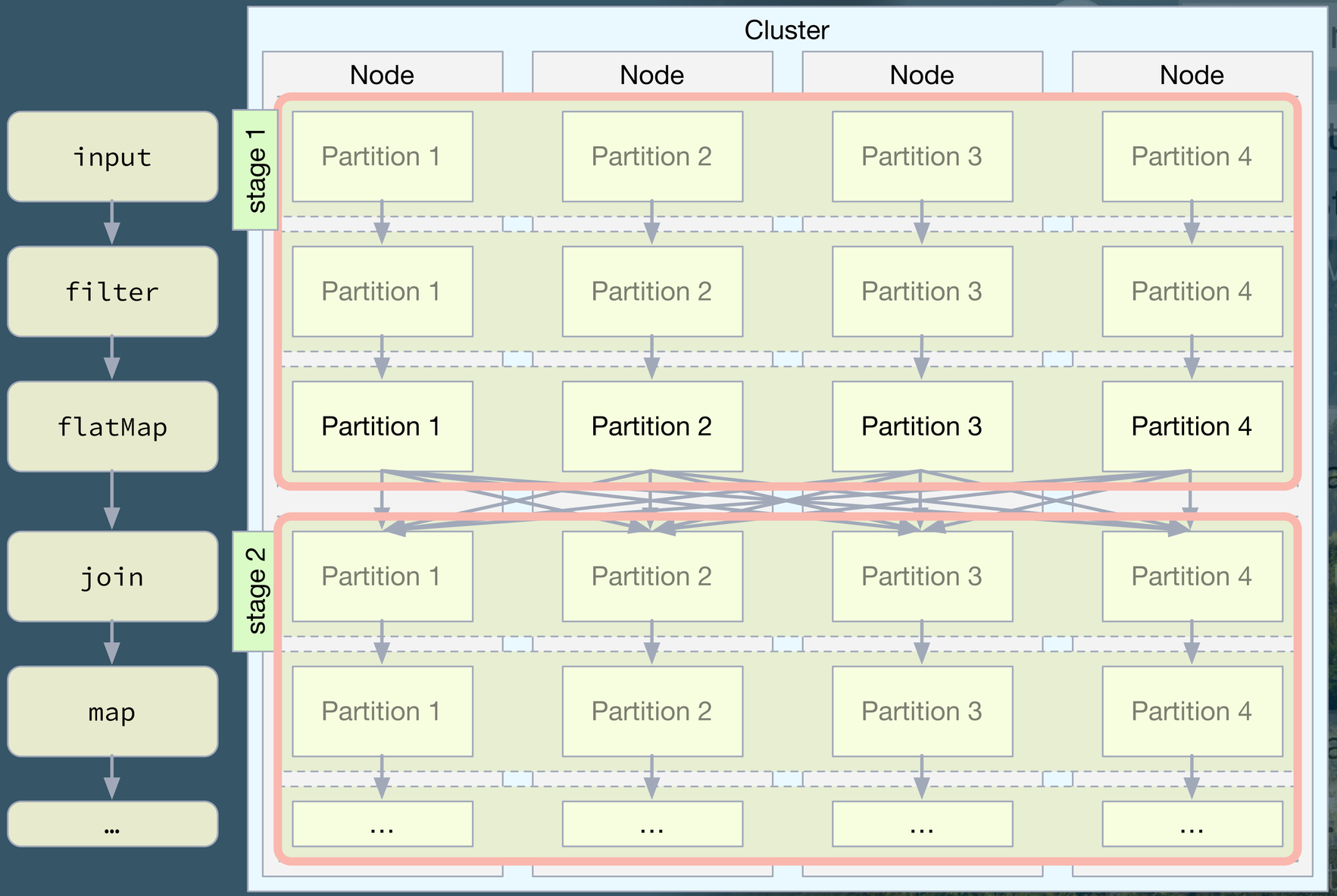
Beam - similar. Google's project that has been instrumental in defining streaming semantics.



They do a *lot* (Spark example)



They do a *lot* (Spark example)



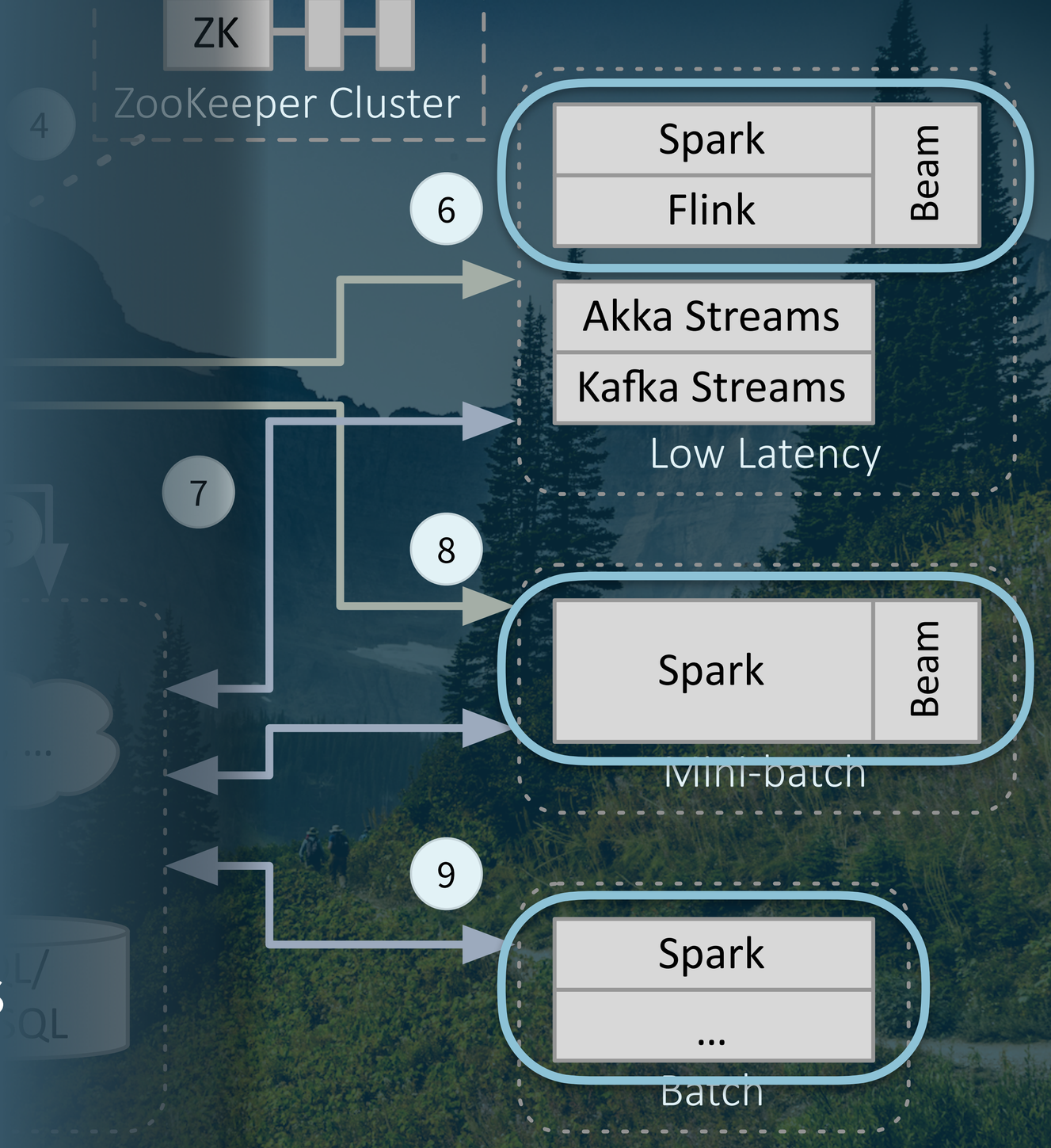
Spark and Flink

Wired:

- Massive scalability
- Rich semantics

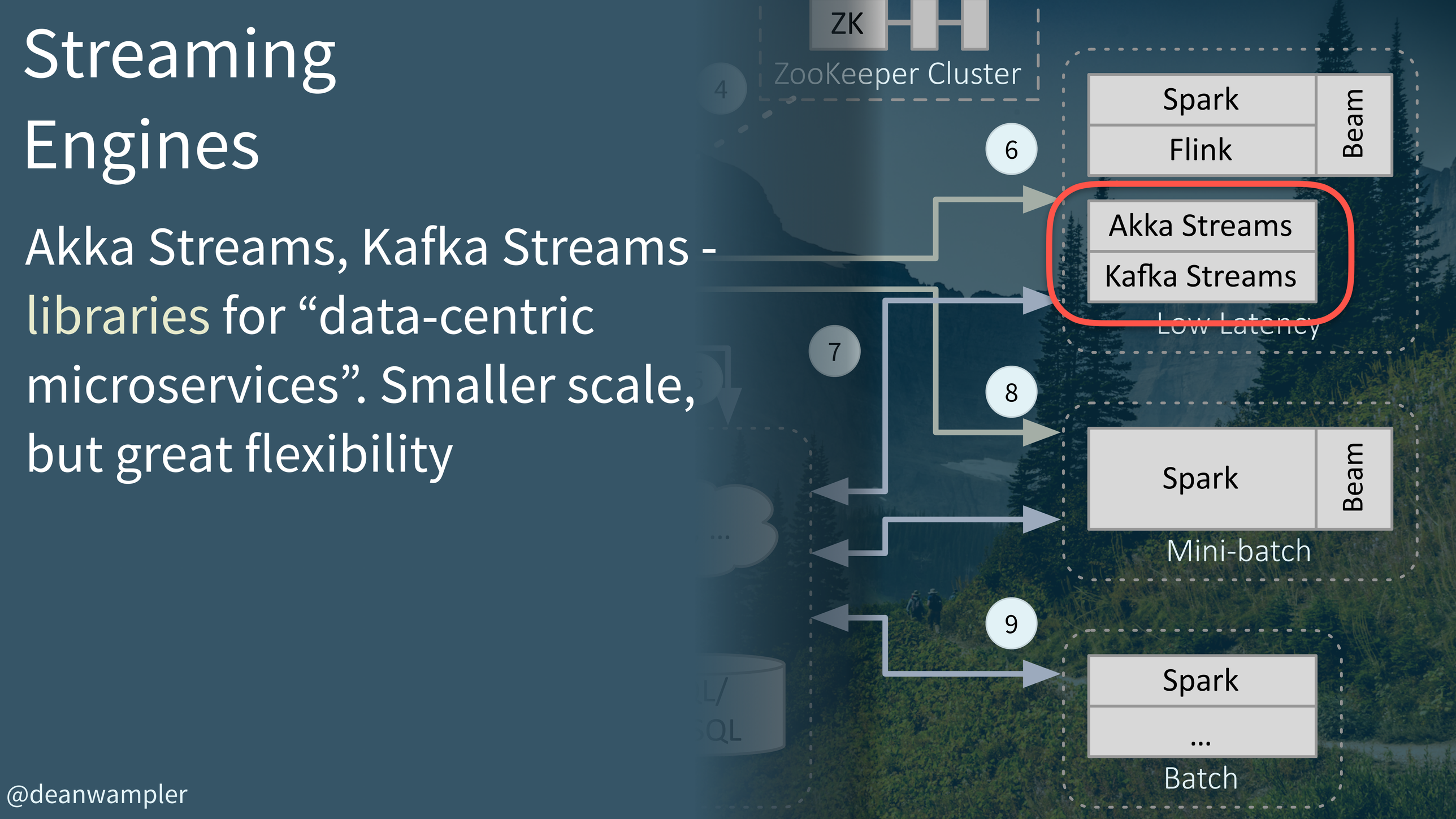
Tired:

- Latency
- Integration in CI/CD workflows



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.

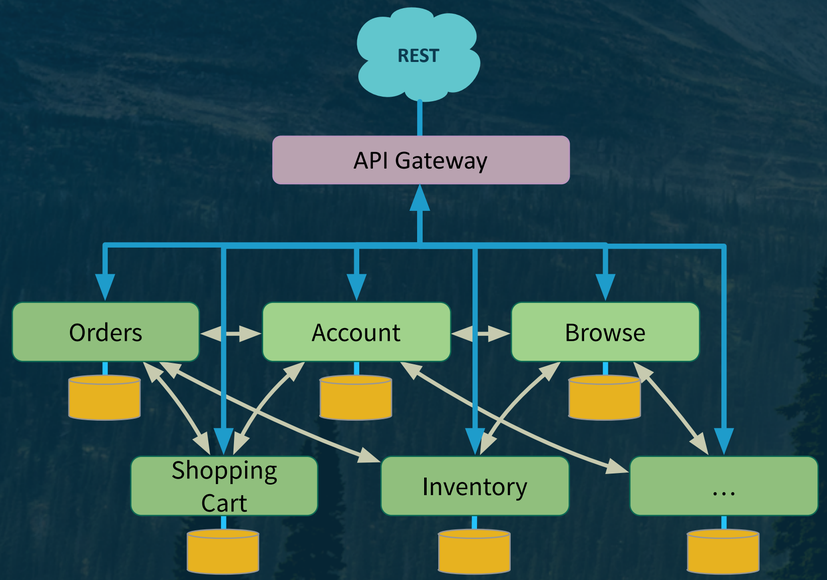
4:11 AM - 25 Feb 2018

207 Retweets **566** Likes

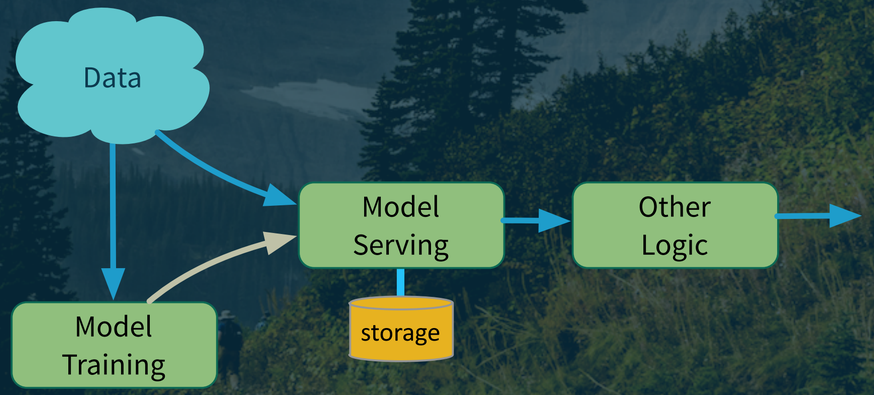
25 207 566

A Spectrum of Microservices

Event-driven μ-services

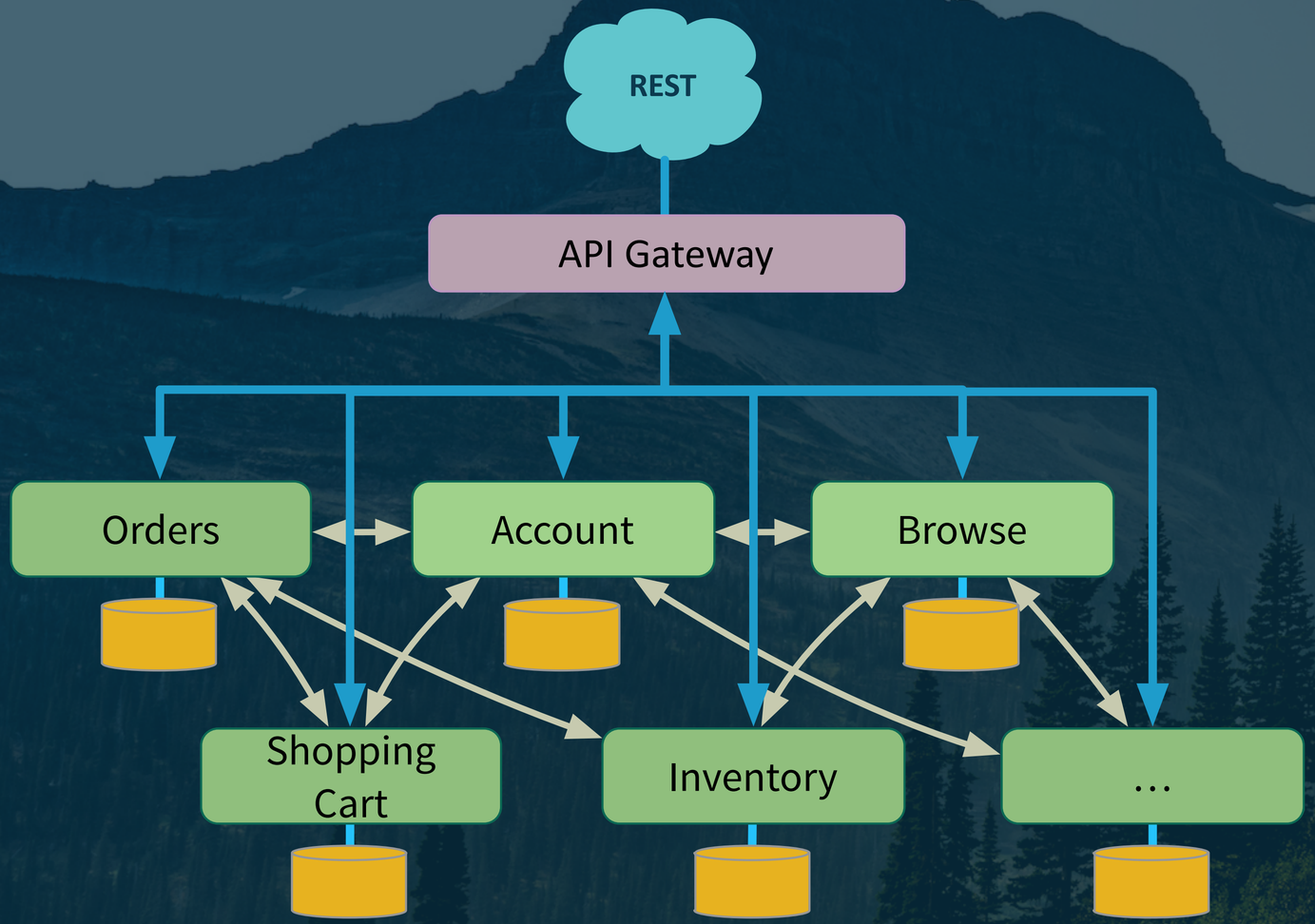


“Record-centric” μ-services



Events ← Data Spectrum → Records

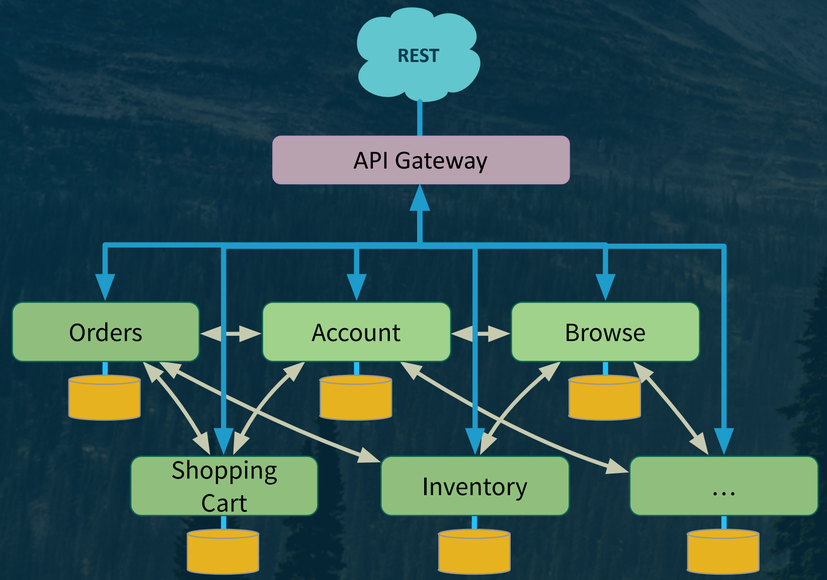
A Spectrum of Microservices



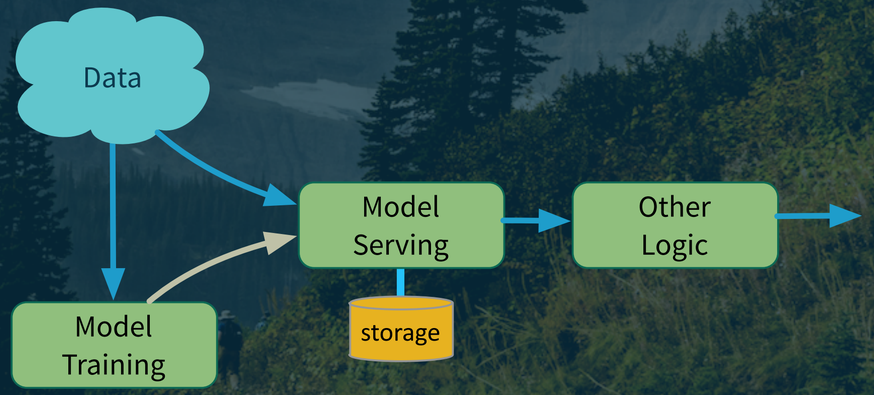
- Each datum has an identity
- Process each one uniquely
- Think *sessions* and *state machines*

A Spectrum of Microservices

Event-driven μ-services



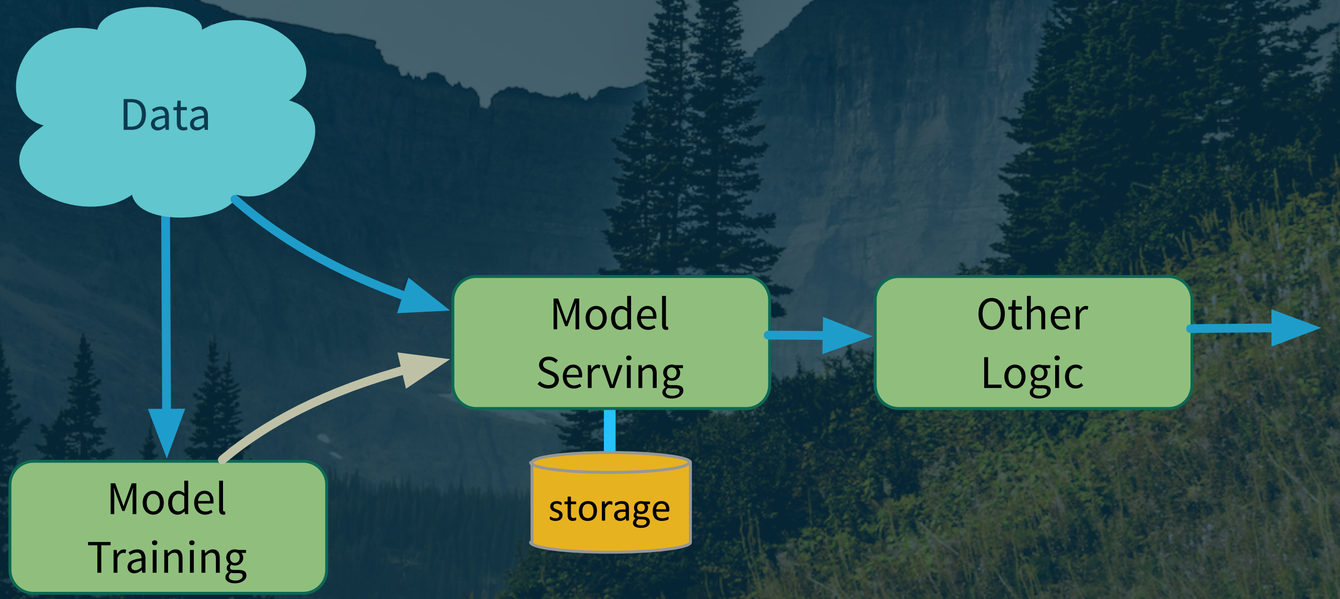
“Record-centric” μ-services



Events ← Data Spectrum → Records

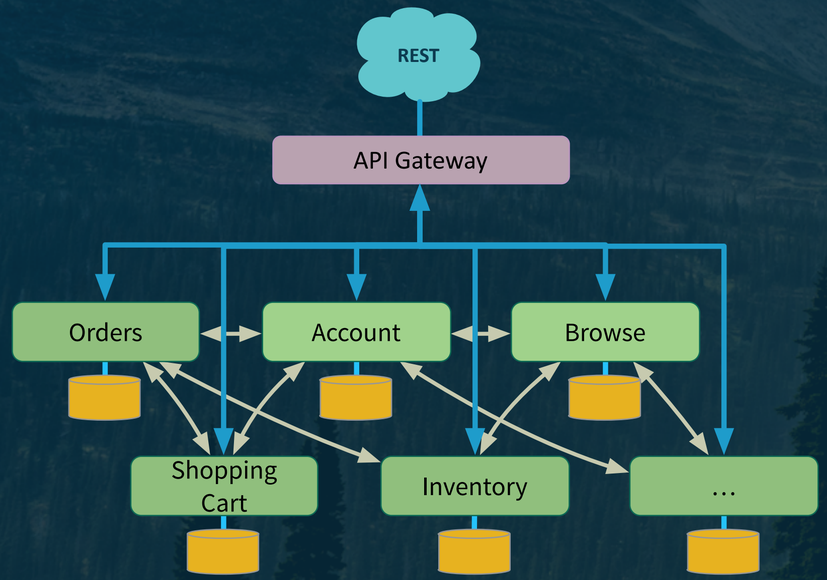
A Spectrum of Microservices

- “Anonymous” records
- Process *en masse*
- Think SQL queries for analytics

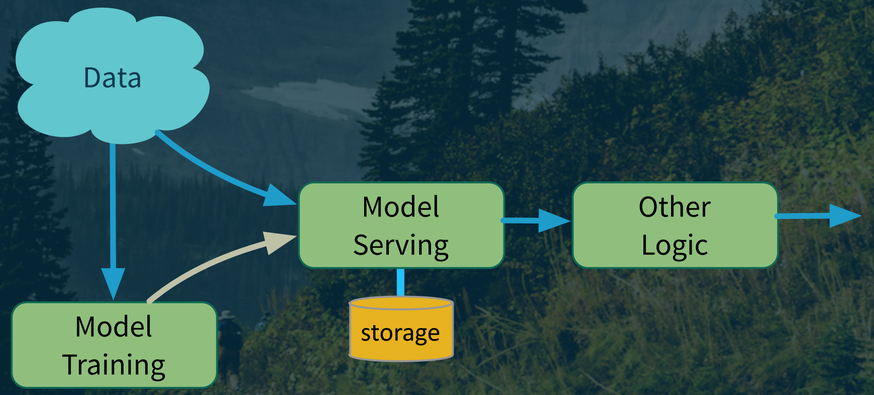


A Spectrum of Microservices

Event-driven μ-services



“Record-centric” μ-services

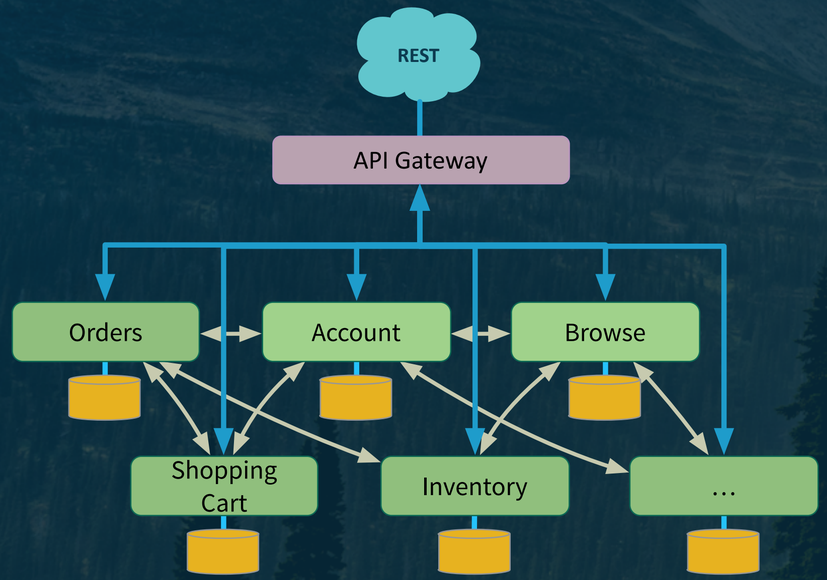


Events ← Data Spectrum → Records

A Spectrum of Microservices

akka Streams

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.

https://www.reactivemanifesto.org/

Events ← Data Spectrum → Records

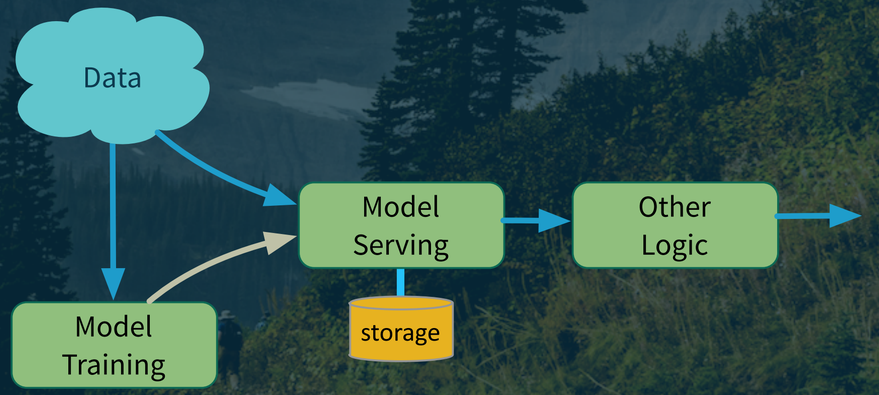
A Spectrum of Microservices

Streams

Emerged from the right-hand side.

“Record-centric” μ-services

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



Events

Records

← Data Spectrum →

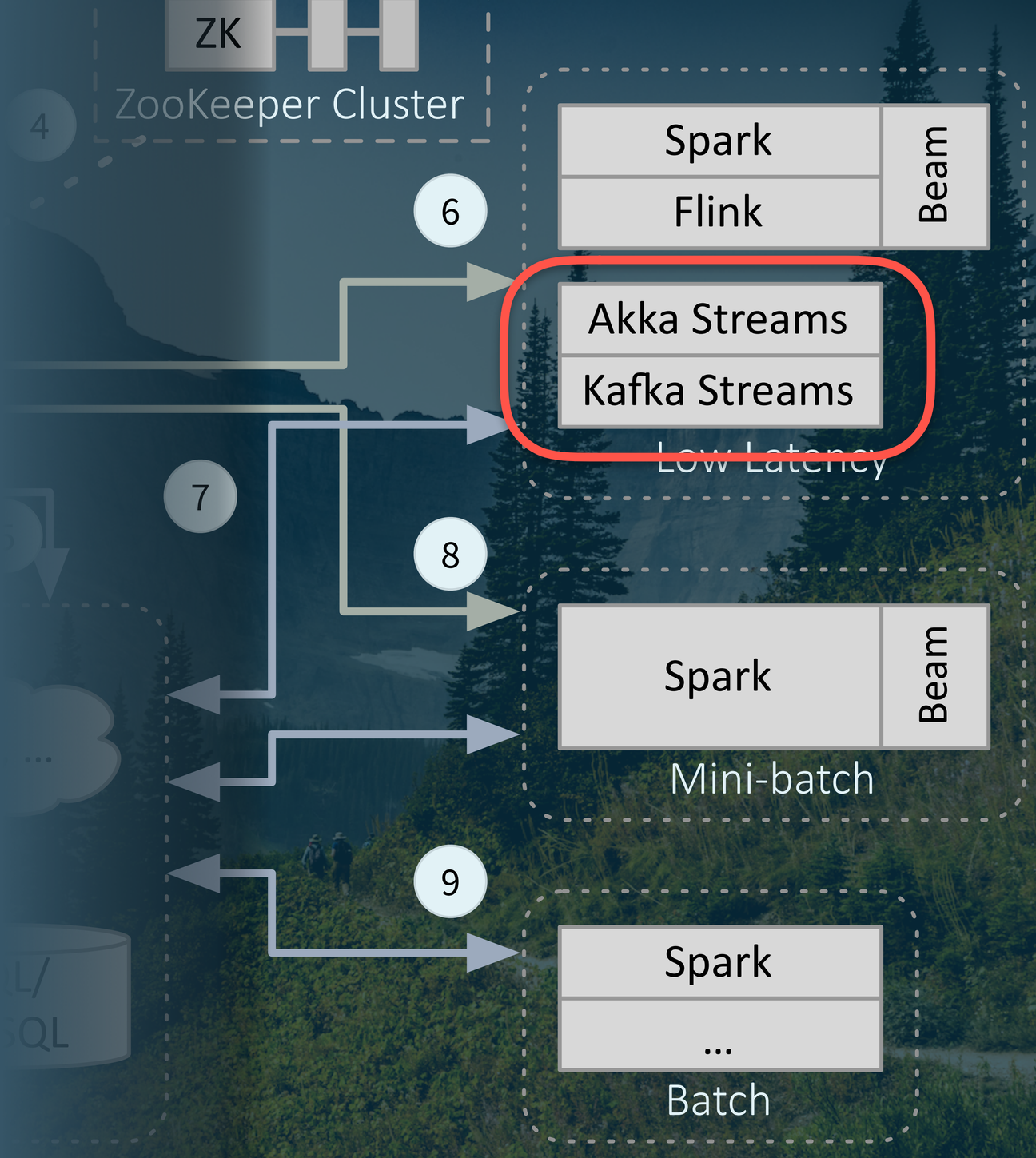
Akka/Kafka Streams

Wired:

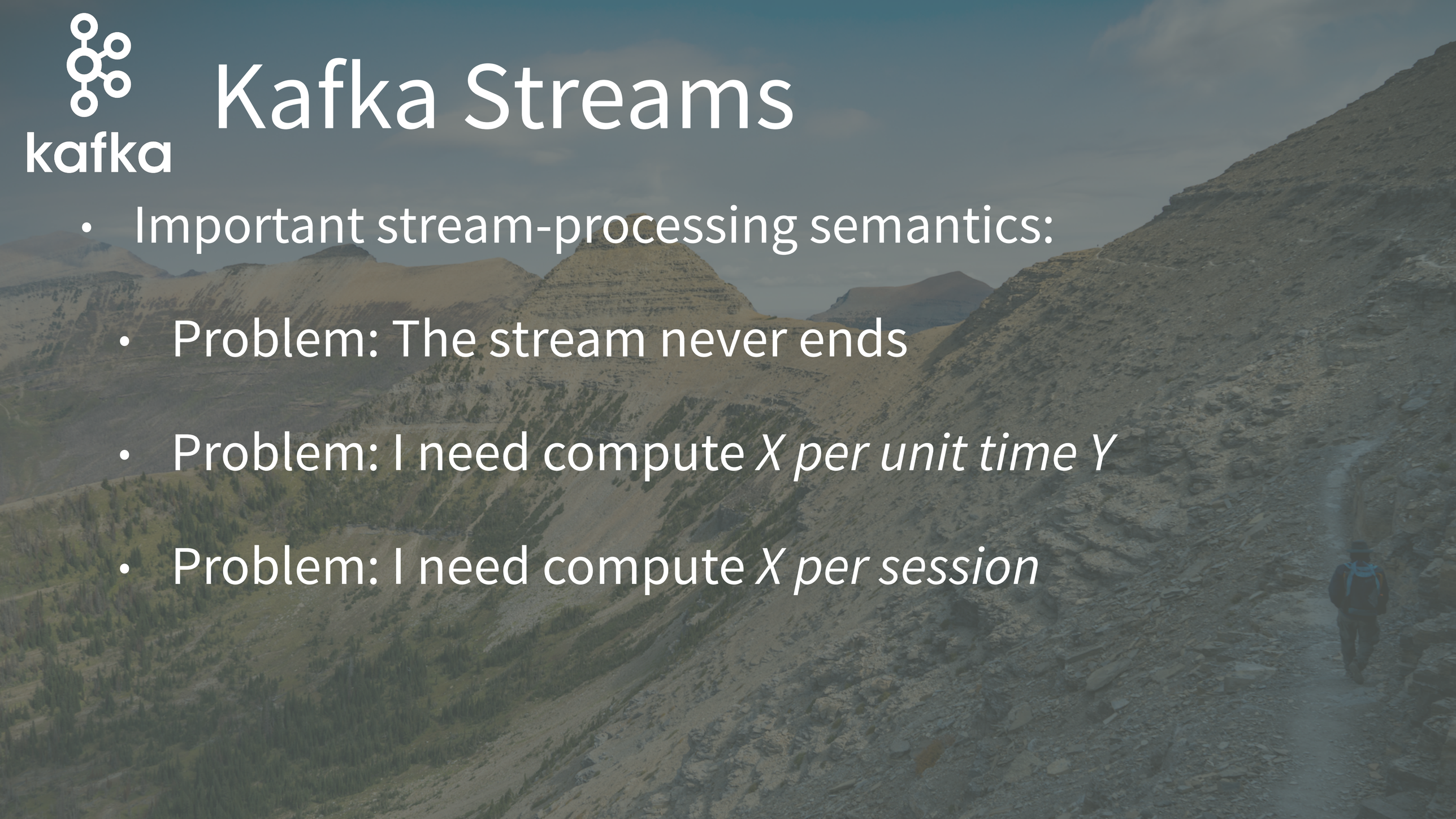
- You already know microservices
- Low latency
- Flexibility

Tired:

- Automatic scalability
- Less-rich semantics




kafka
Kafka Streams
@deanwampler

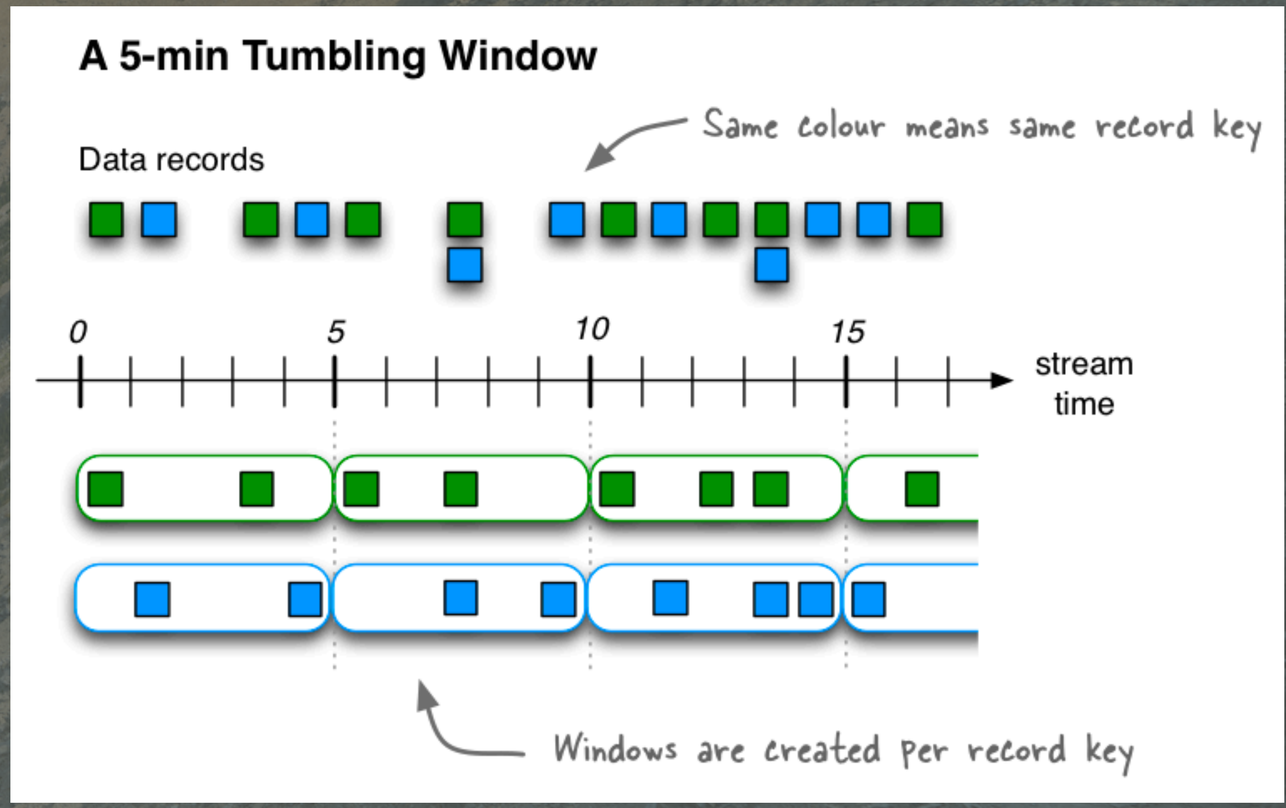


Kafka Streams

- Important stream-processing semantics:
  - Problem: The stream never ends
  - Problem: I need compute *X per unit time Y*
  - Problem: I need compute *X per session*

Kafka Streams

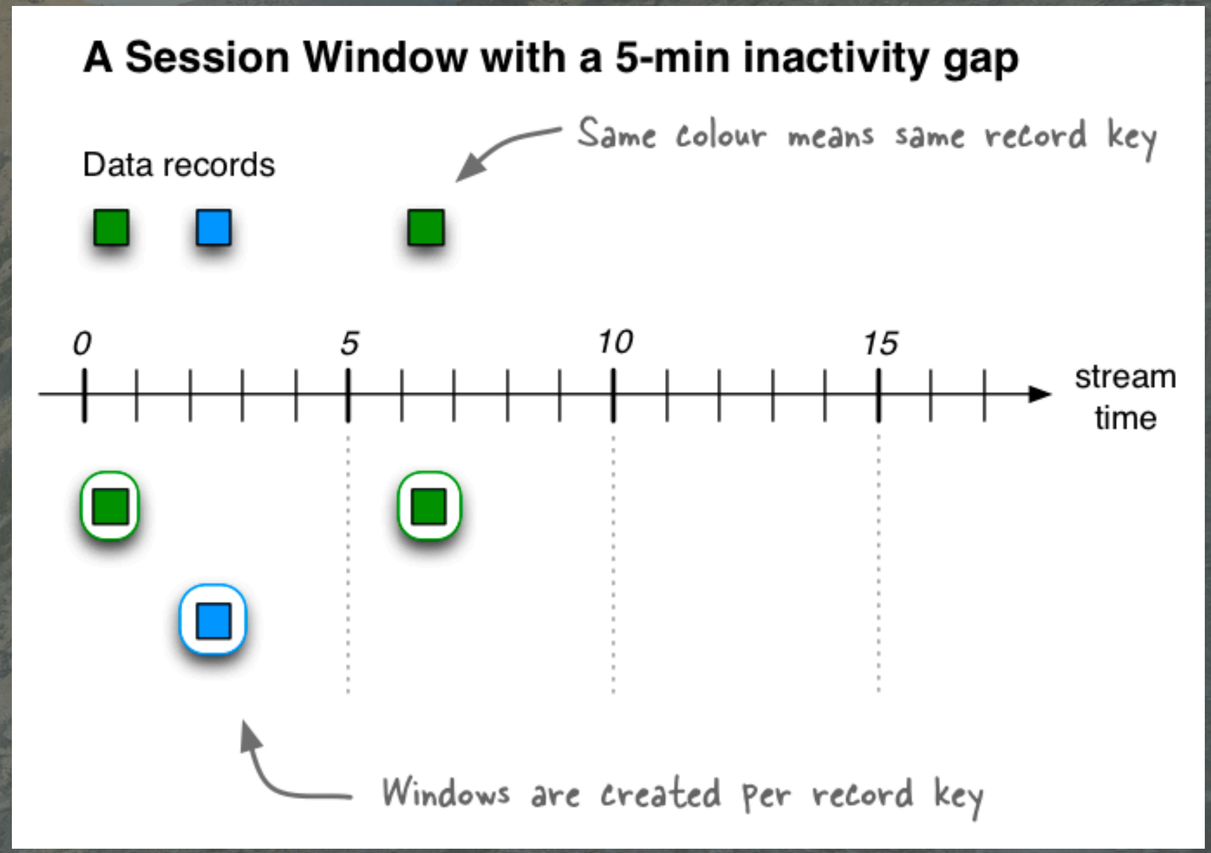
- Important stream-processing semantics:
  - Solution: Windowing
    - Tumbling windows:



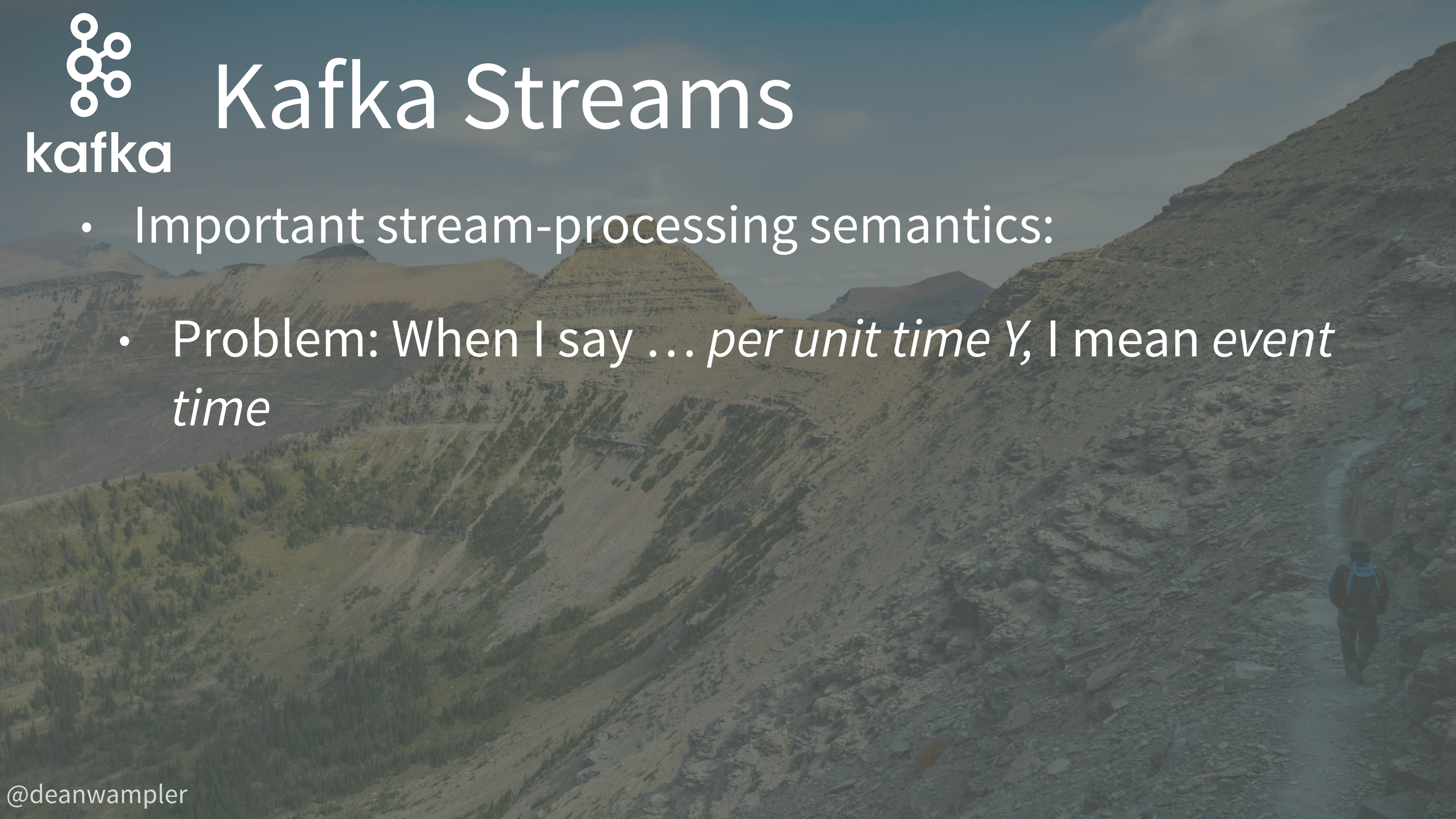
See my O'Reilly report for details.

Kafka Streams

- Important stream-processing semantics:
  - Solution: Windowing
    - Session windows:



See my O'Reilly report for details.

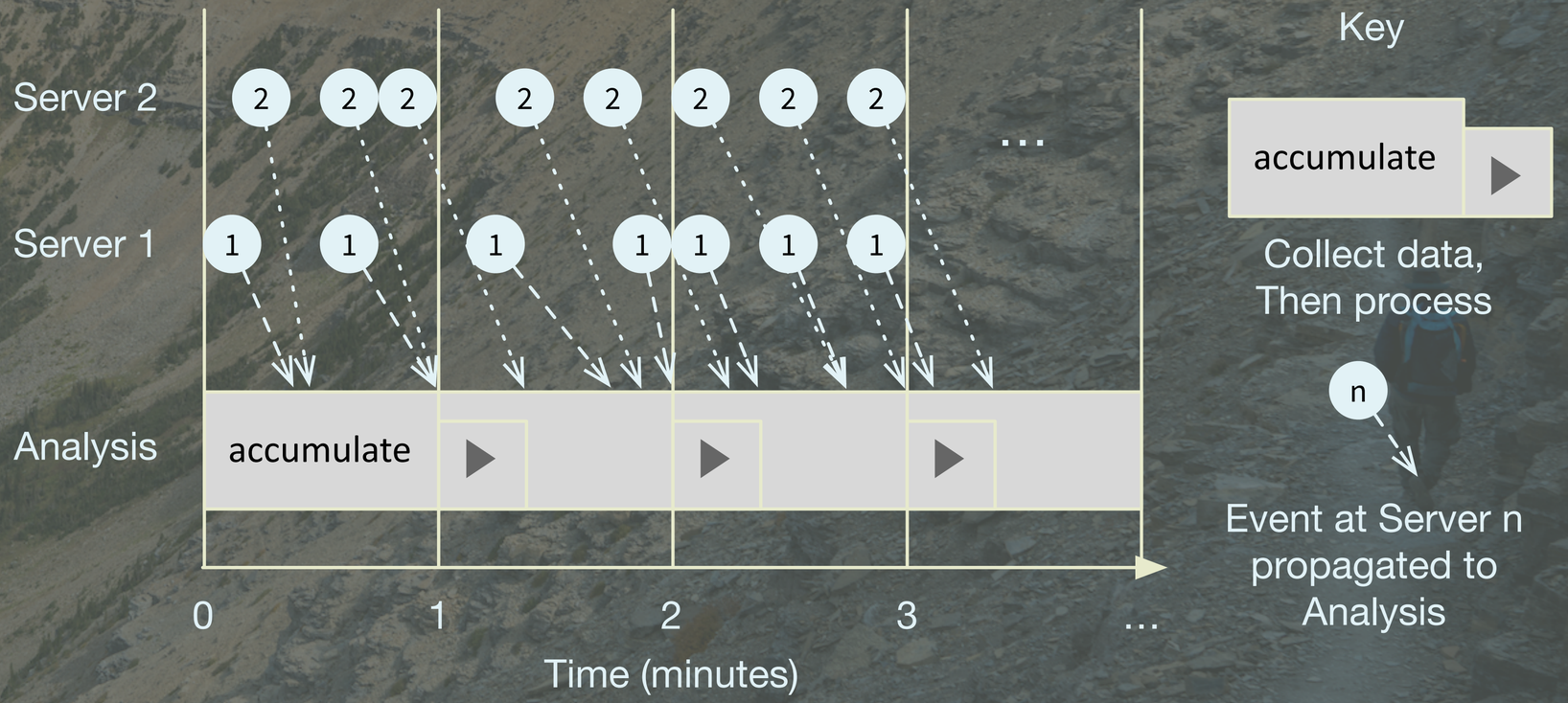


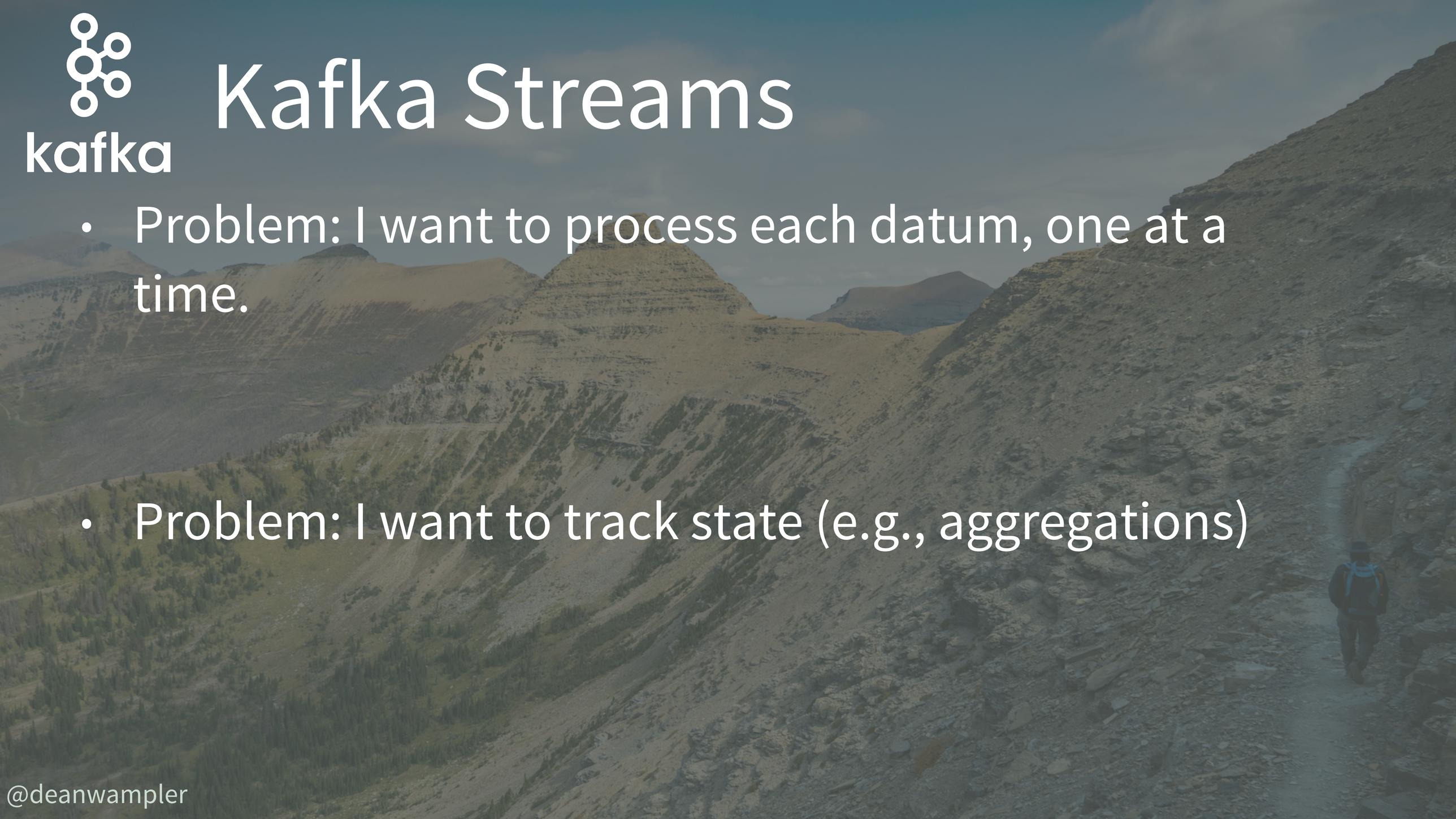
Kafka Streams

- Important stream-processing semantics:
  - Problem: When I say … *per unit time Y,* I mean *event time*

Kafka Streams

- Important stream-processing semantics:
  - Solution: *event time* and *processing time*





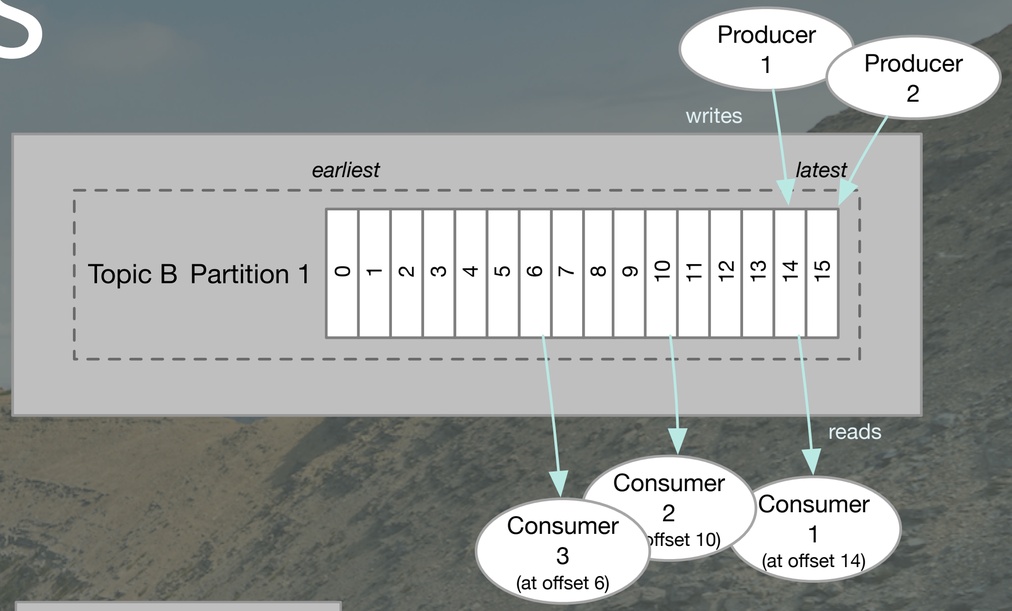
Kafka Streams

- Problem: I want to process each datum, one at a time.

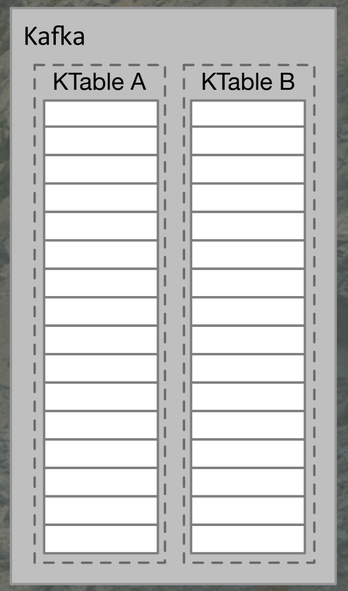
- Problem: I want to track state (e.g., aggregations)

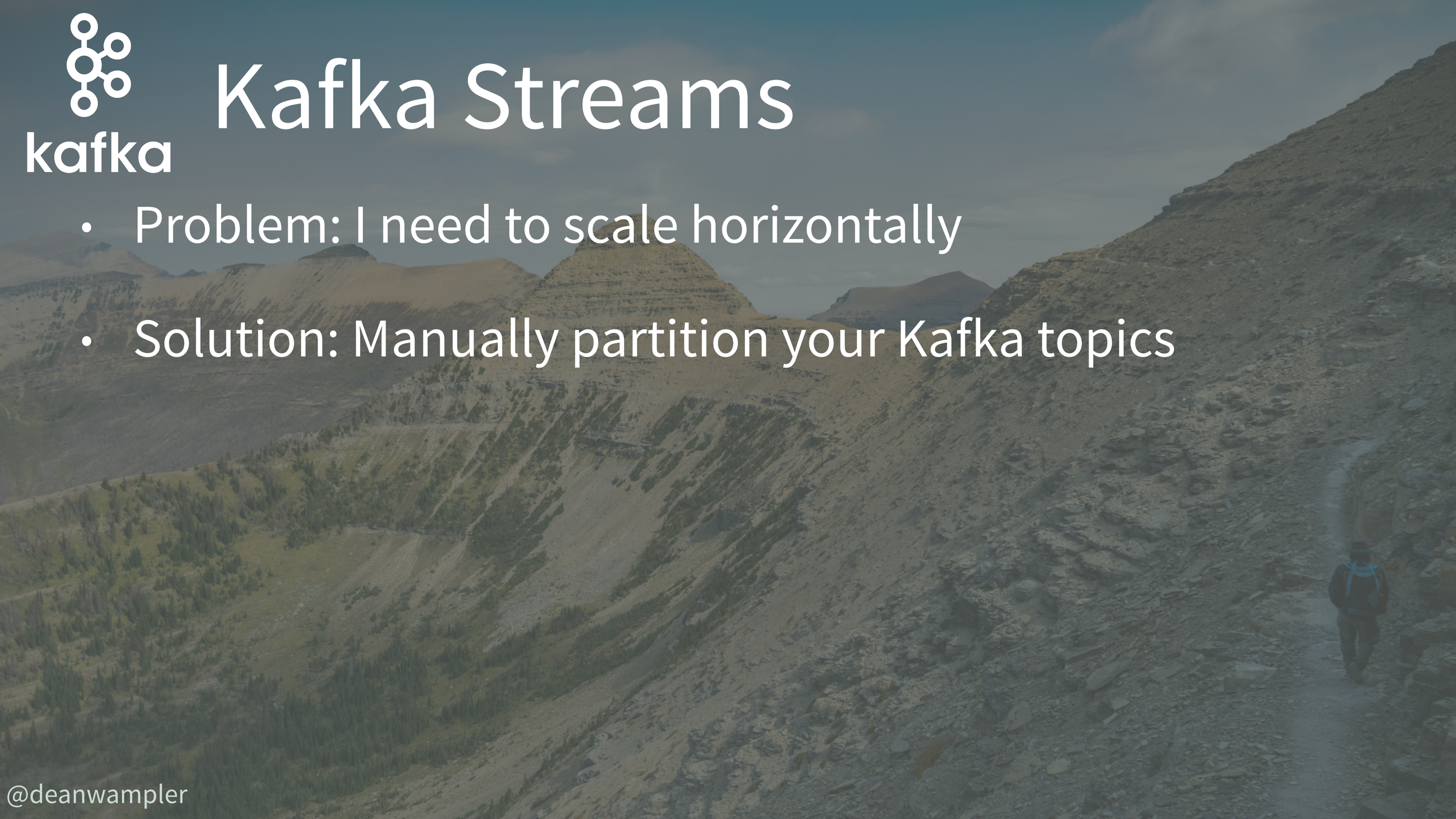
Kafka Streams

- Solution: KStream
  - See the whole stream



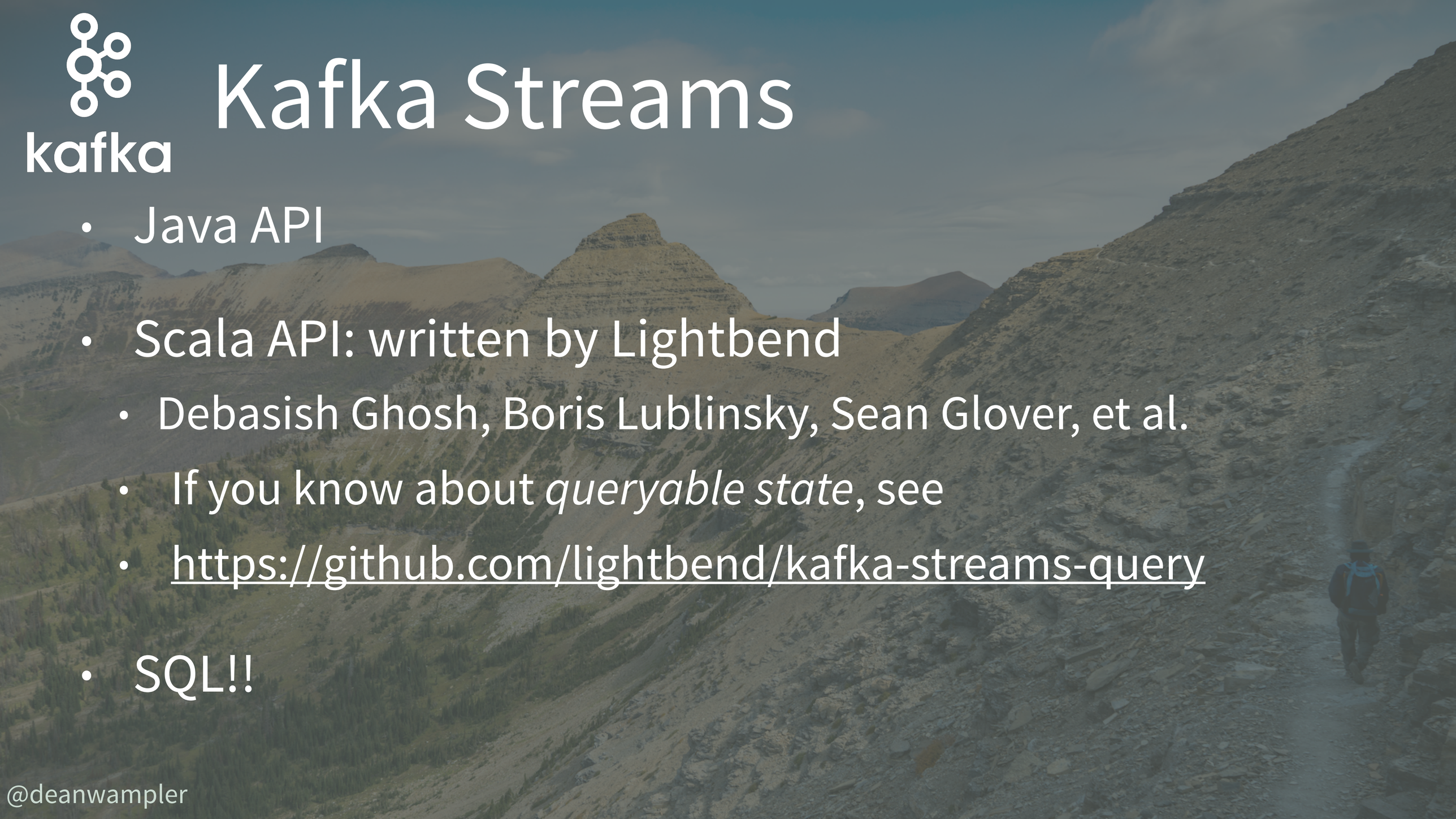
- Solution: KTable
  - Last value per key





Kafka Streams

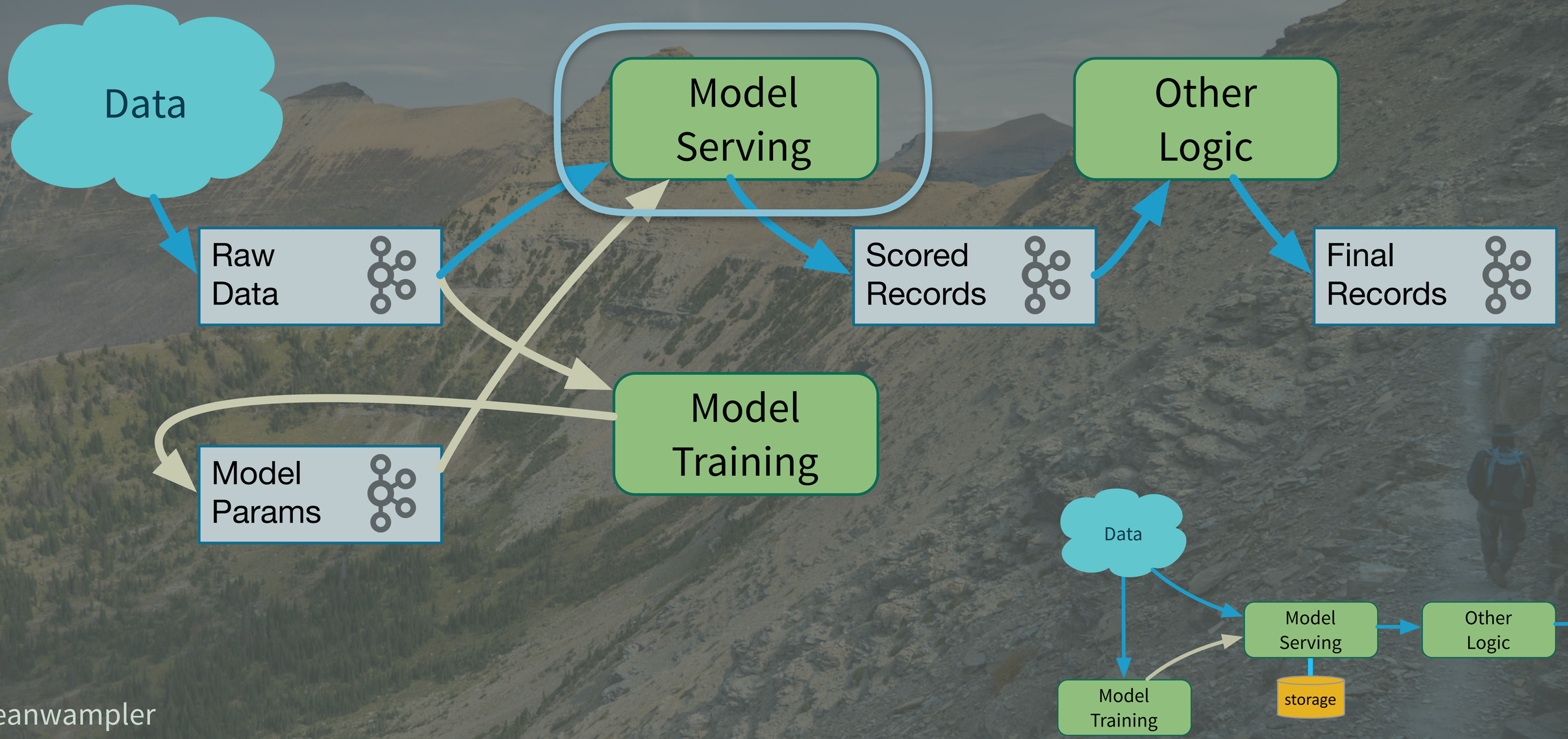
- Problem: I need to scale horizontally
- Solution: Manually partition your Kafka topics

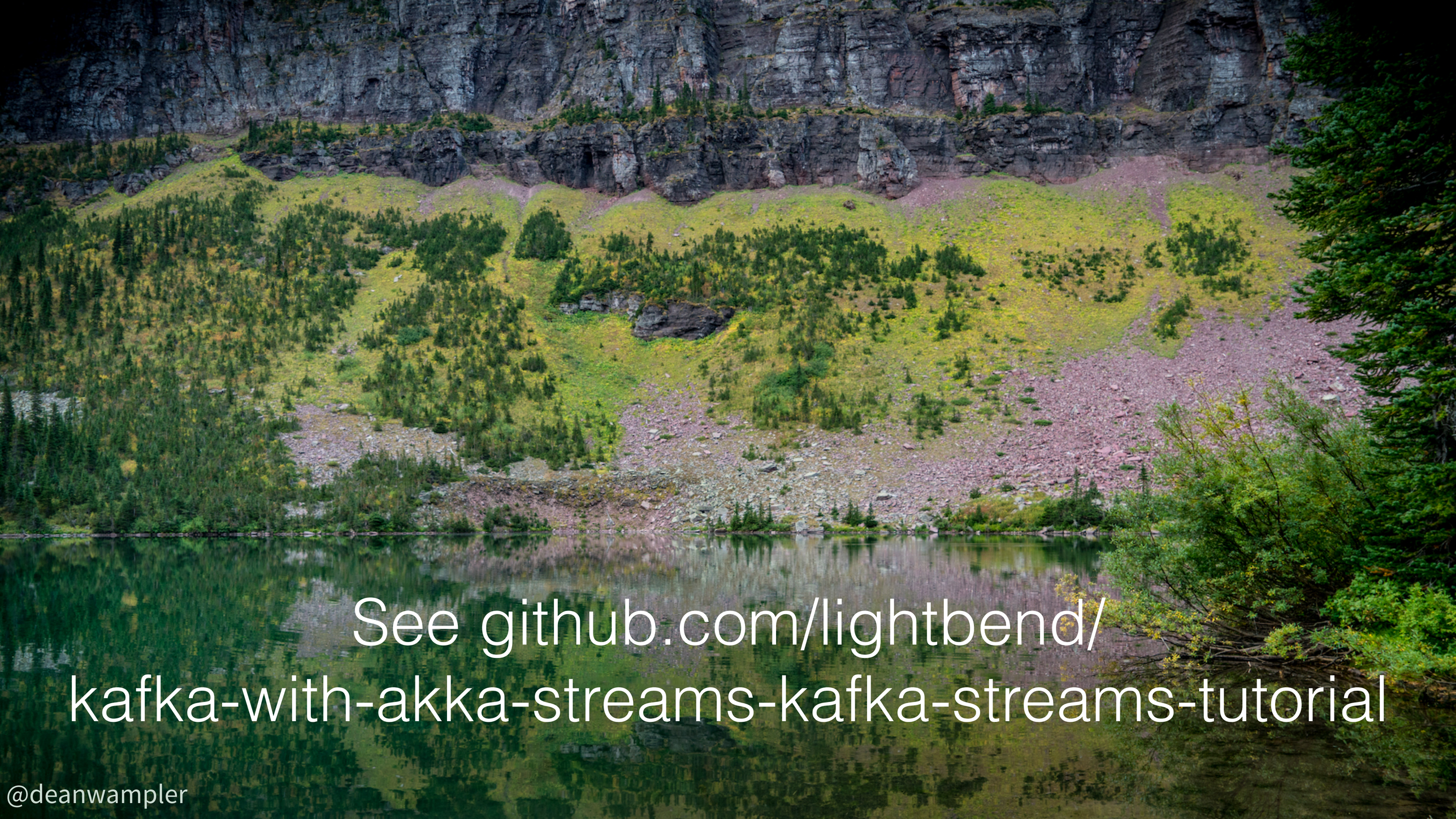


Kafka Streams

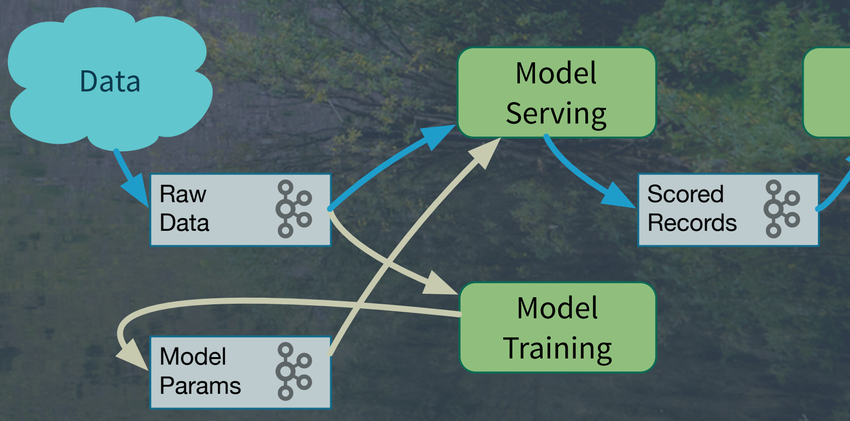
- Java API
- Scala API: written by Lightbend
  - Debasish Ghosh, Boris Lublinsky, Sean Glover, et al.
  - If you know about *queryable state*, see
  - https://github.com/lightbend/kafka-streams-query
- SQL!!

Kafka Streams Example

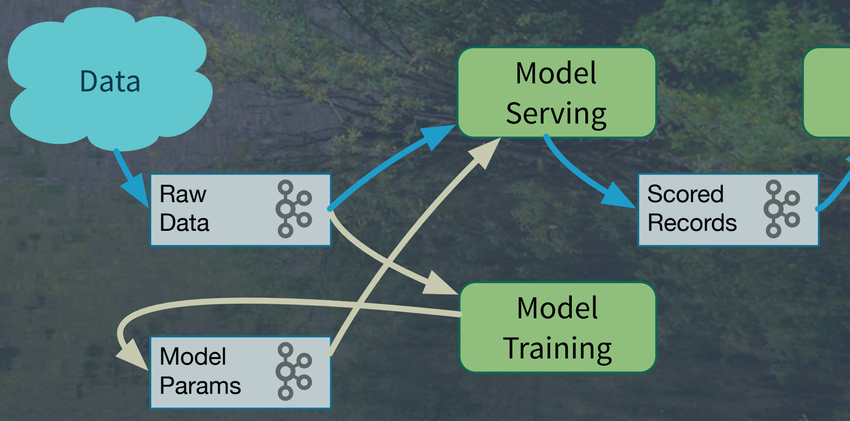



See github.com/lightbend/
kafka-with-akka-streams-kafka-streams-tutorial

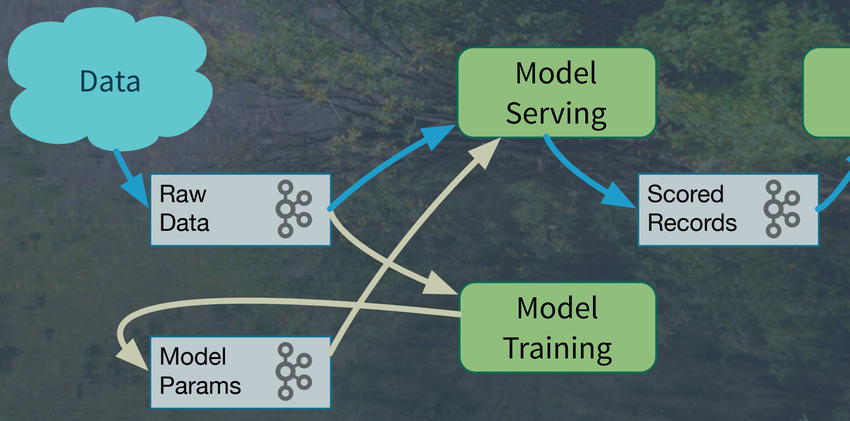
```
val builder = new StreamsBuilderS // New Scala Wrapper API.

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())
```



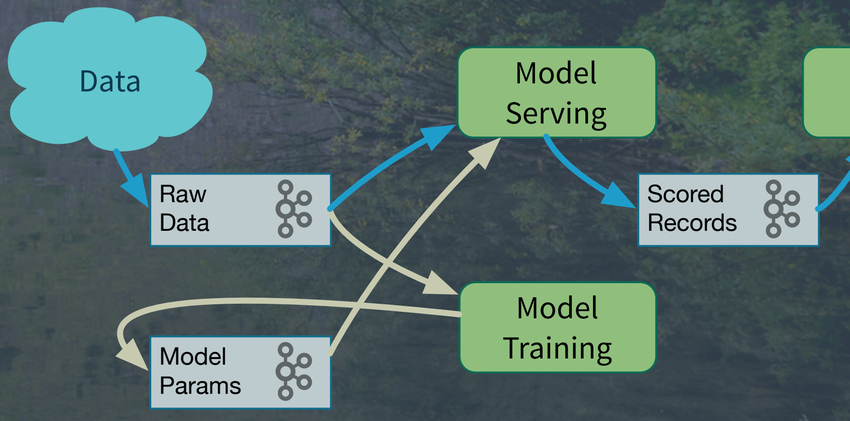
```
val builder = new StreamsBuilderS // New Scala Wrapper API.

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(scorer.score(record),record))
    .to(scoredRecordsTopic)

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



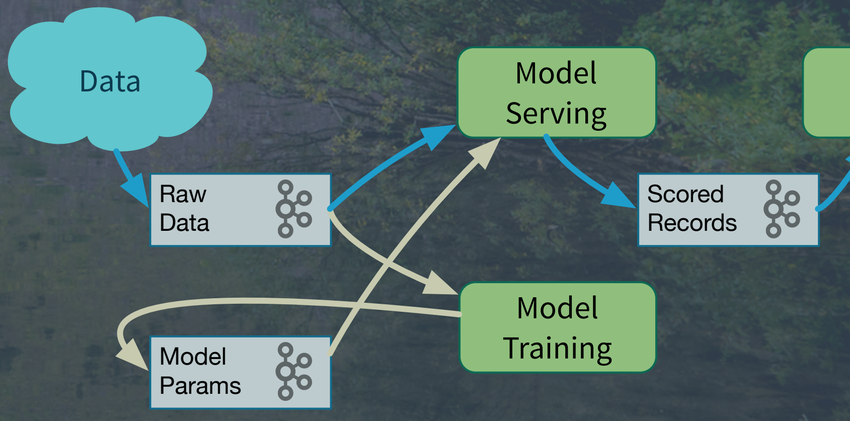
```
val builder = new StreamsBuilderS // New Scala Wrapper API.

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())
```



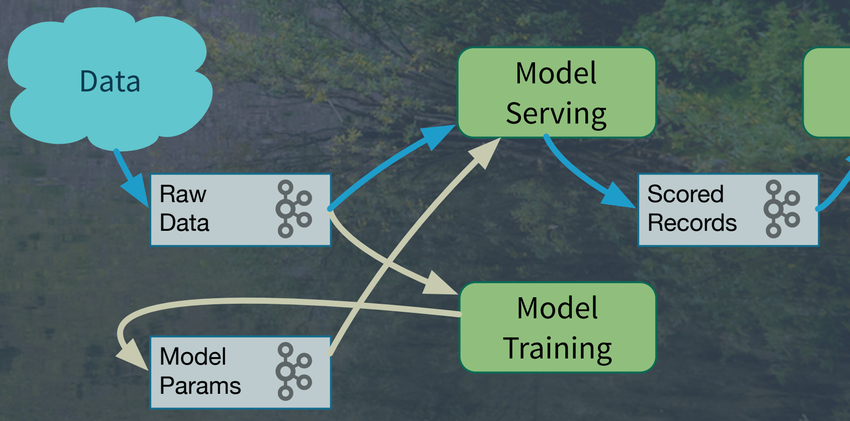
```
val builder = new StreamsBuilderS // New Scala Wrapper API.

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(scorer.score(record),record))
     .to(scoredRecordsTopic)

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



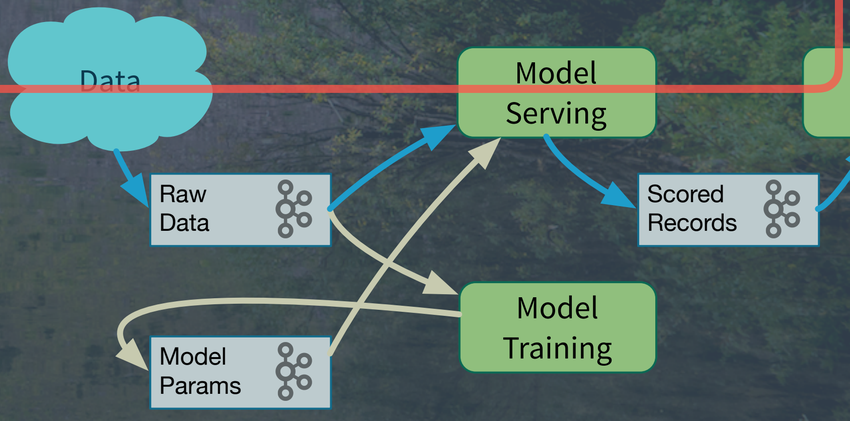
```
val builder = new StreamsBuilderS // New Scala Wrapper API.

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())
```



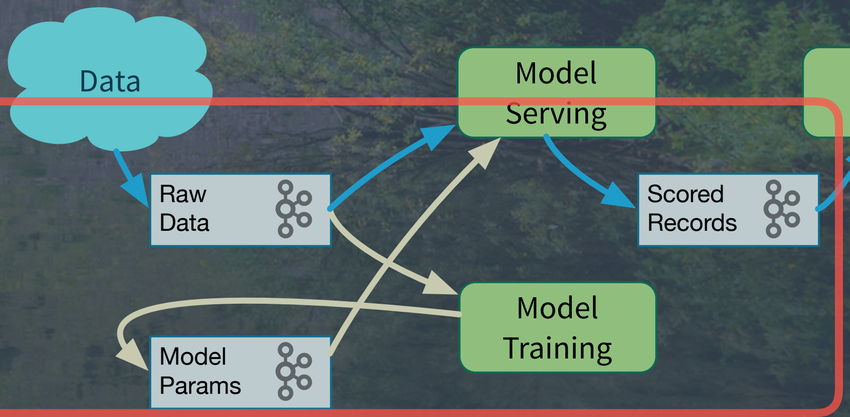
```
val builder = new StreamsBuilderS // New Scala Wrapper API.

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(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(scorer.score(record),record))
    .to(scoredRecordsTopic)

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



What's Missing?

The rest of the microservice tools you need…

Embed your Kafka Streams code in microservices written with Akka or other JVM tools.

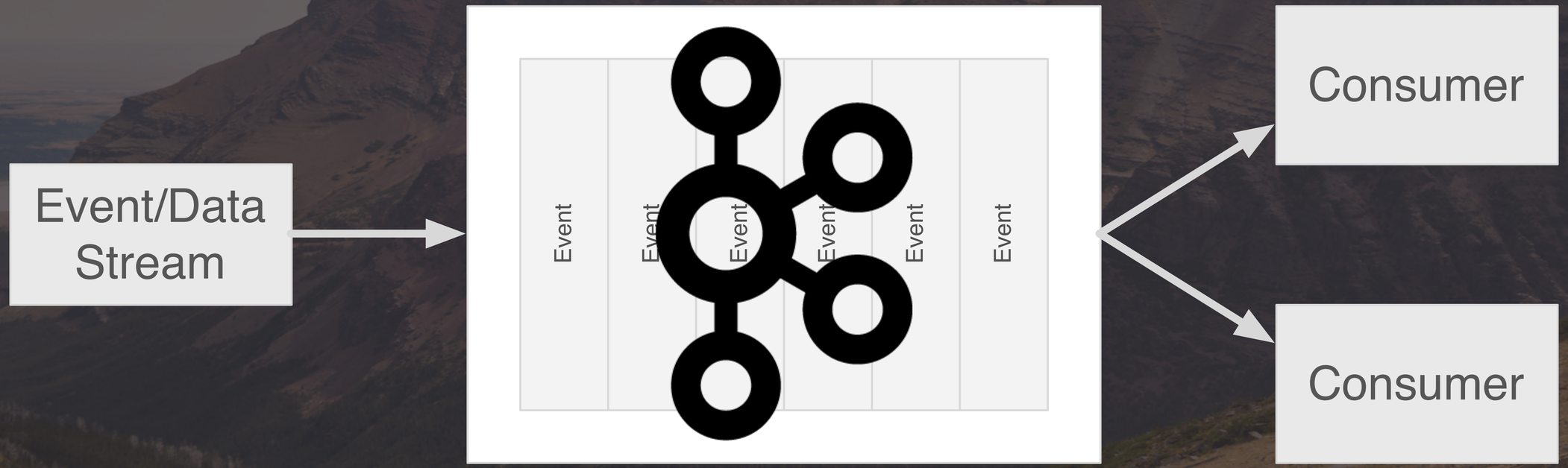

akka
Akka Streams
@deanwampler

Akka Streams

- Problem: I need reliable *buffering* between services

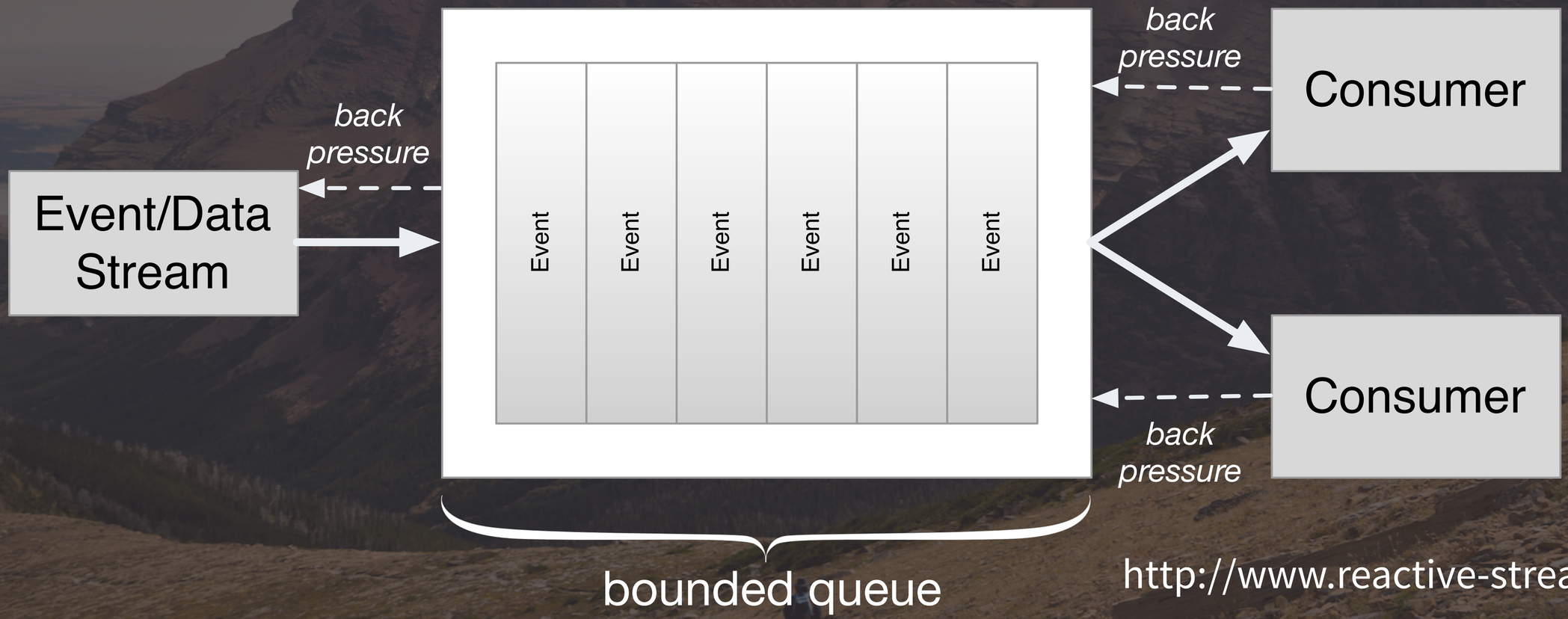
Akka Streams

- Solution: Use a Kafka Topic!



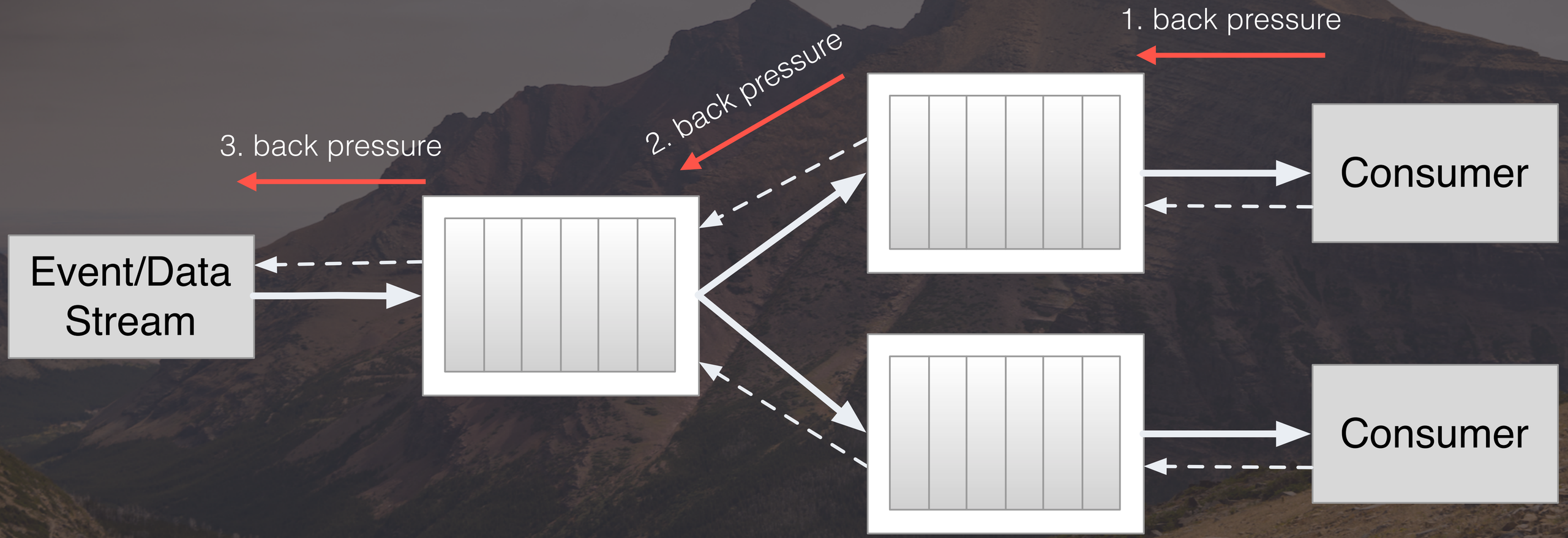
Akka Streams

- Solution: Use *Reactive Streams* for flow control

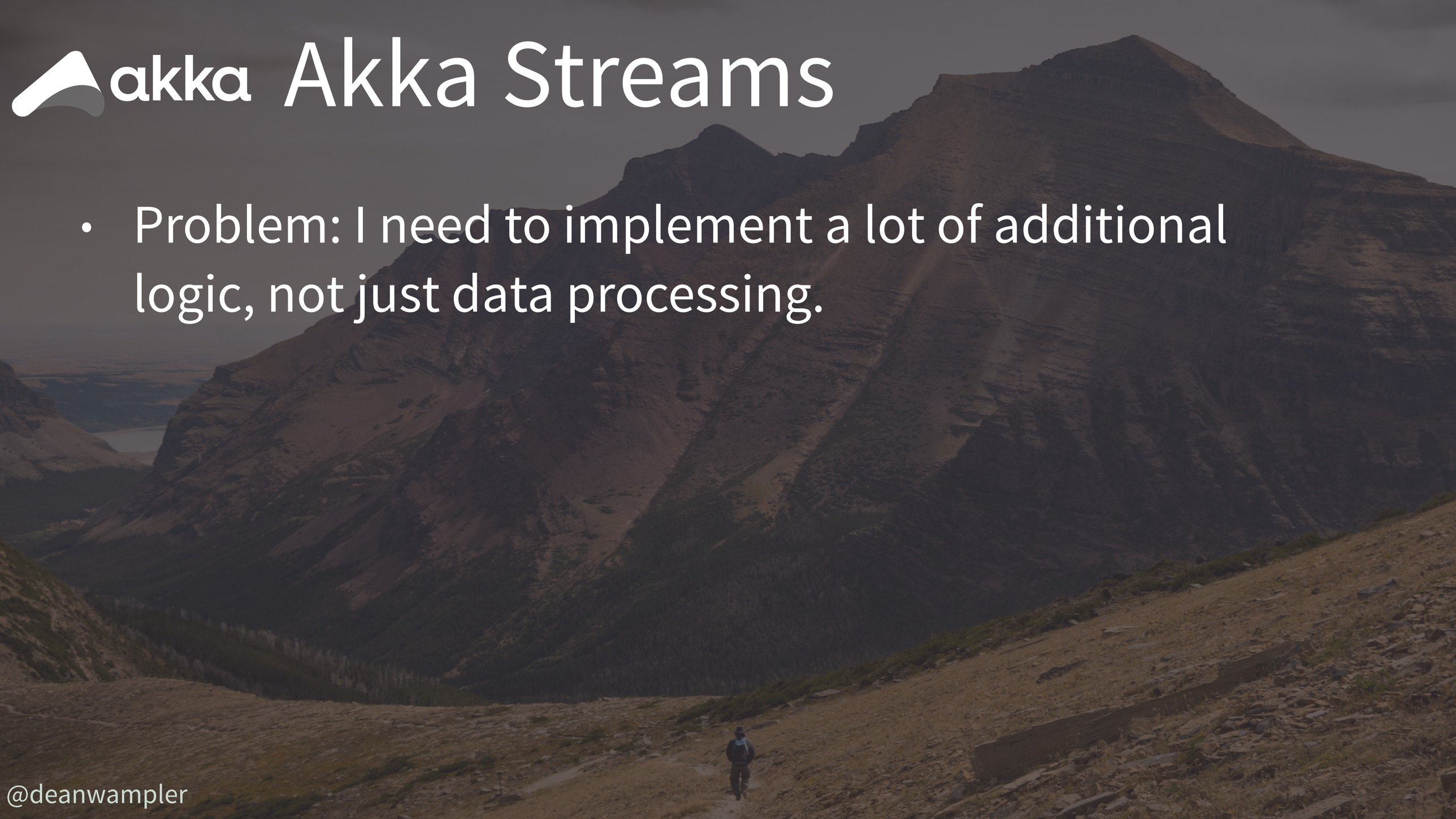


http://www.reactive-streams.org/

Akka Streams



Back pressure composes!

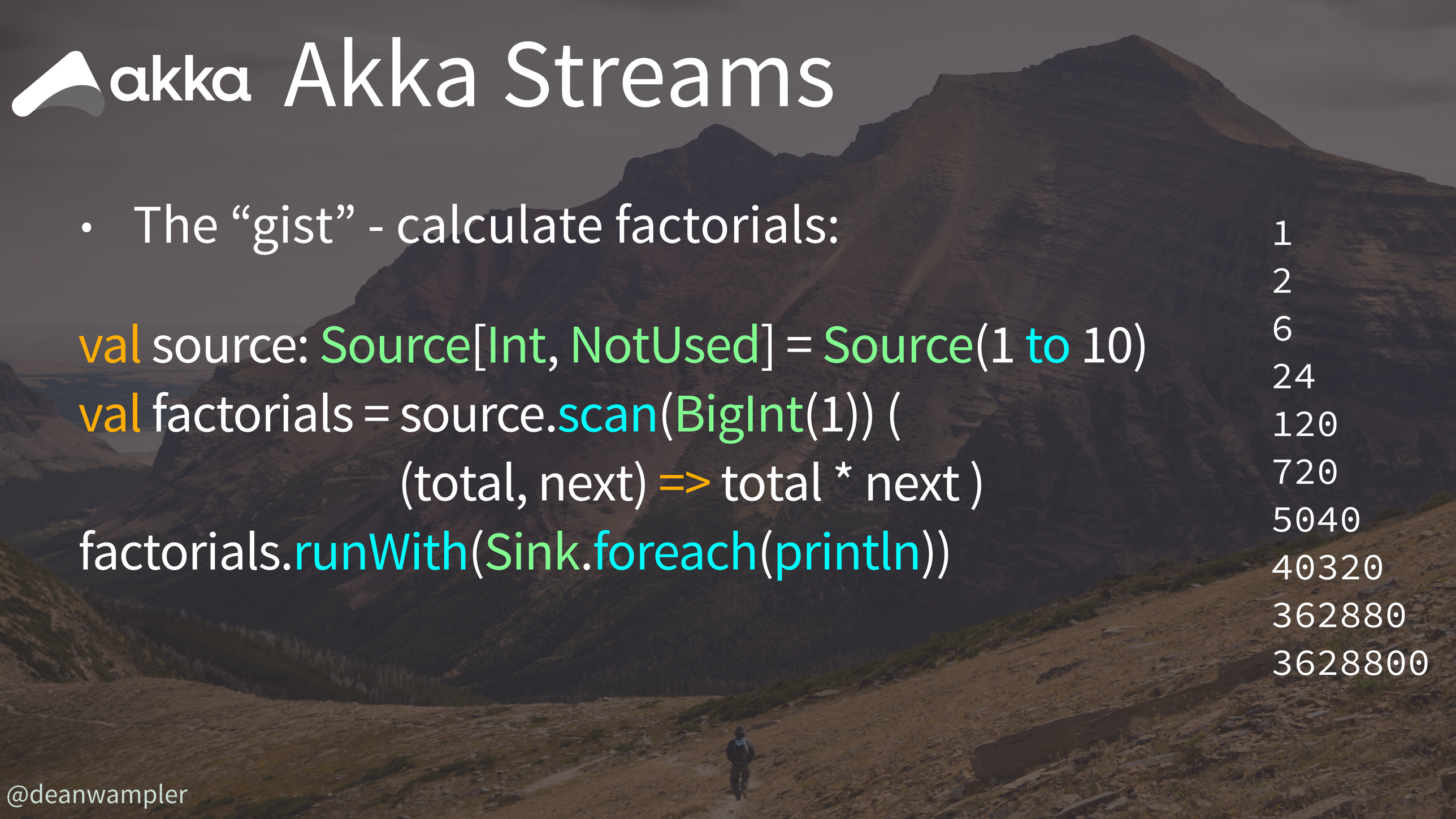


Akka Streams

- Problem: I need to implement a lot of additional logic, not just data processing.

Akka Streams

- Solution: Part of the Akka ecosystem
  - Akka Actors, Akka Cluster, Akka HTTP, Akka Persistence, …
  - Alpakka - rich connection library
- Optimized for low overhead and latency



Akka Streams

- The “gist” - calculate factorials:

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

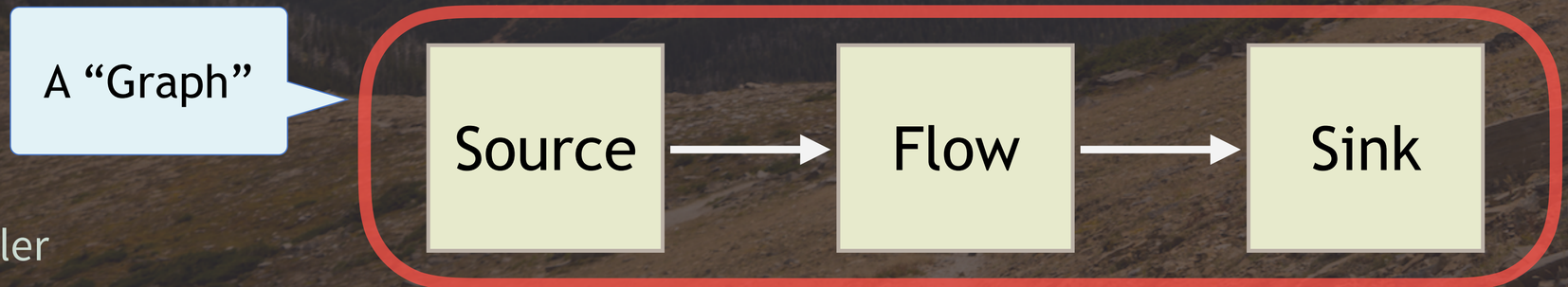
```
1
2
6
24
120
720
5040
40320
362880
3628800
```

Akka Streams

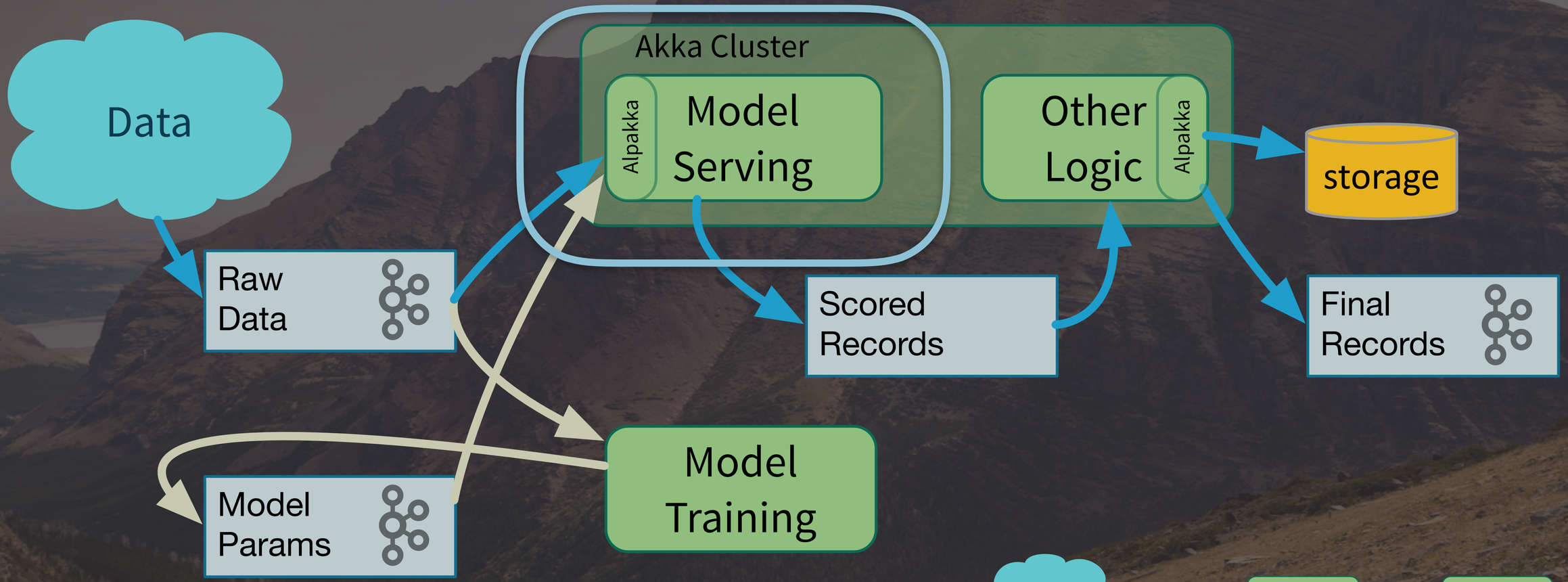
- The “gist” - calculate factorials:

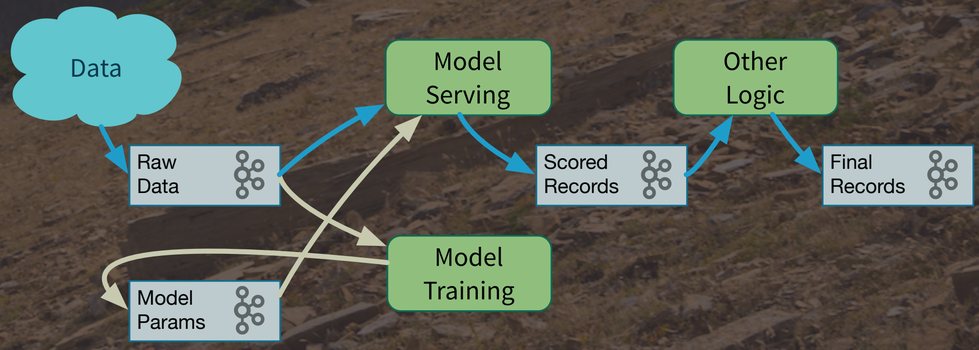
```
val source: Source[Int, NotUsed] = Source(1 to 10)
val factorials = source.scan(BigInt(1)) (
                       (total, next) => total * next )
factorials.runWith(Sink.foreach(println))
```

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

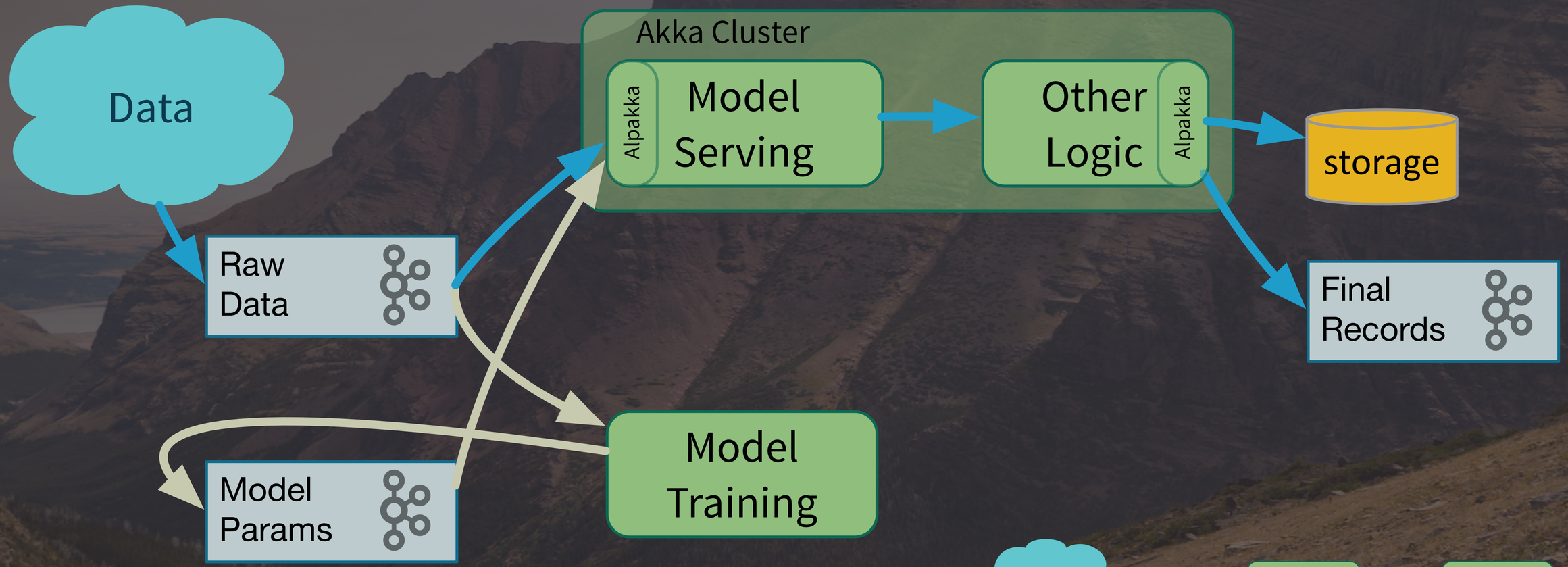


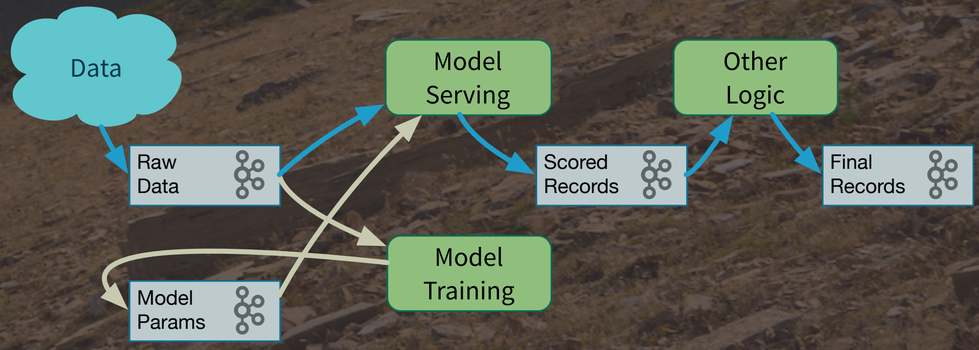
Akka Streams Example



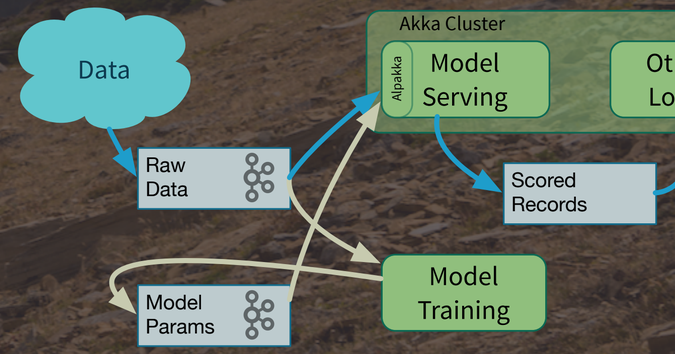


Akka Streams Example - “Direct” Variation



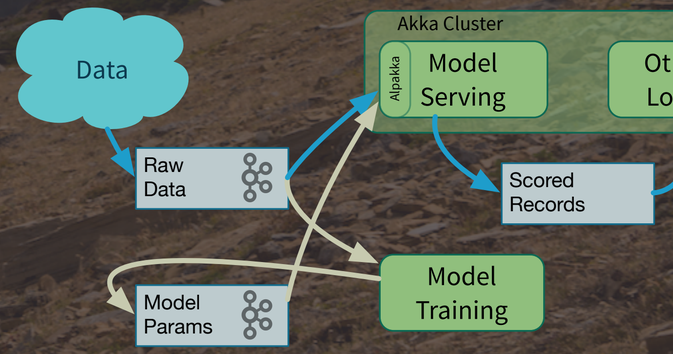


```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

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 }
```

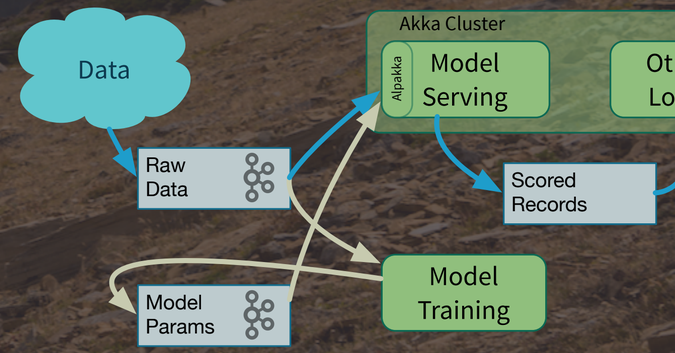


```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher
```

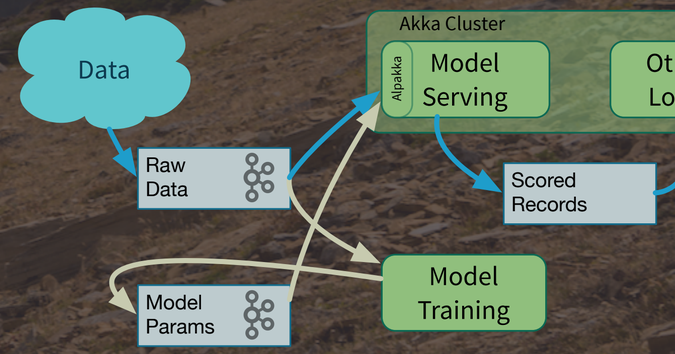
```
val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom “stage”

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 }
```

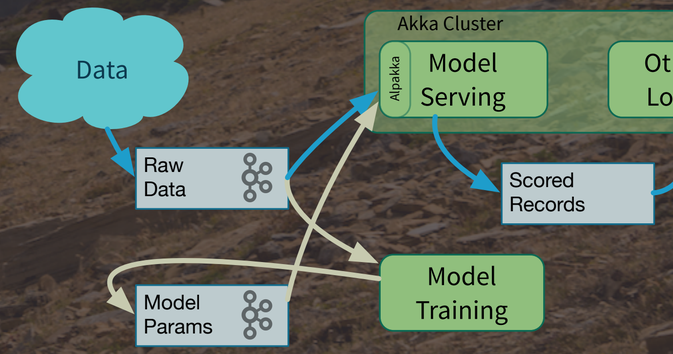


```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

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

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 }
```

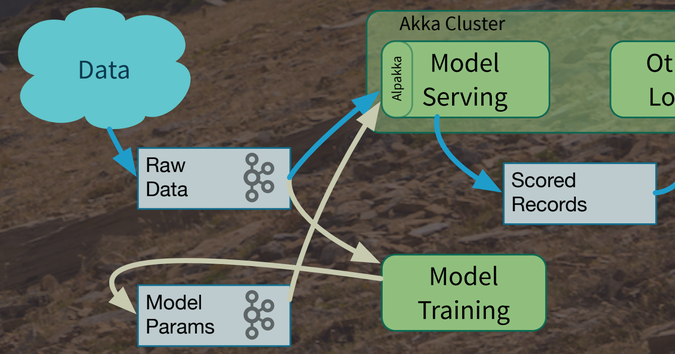
```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

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 }
```



```
  .collect{ case Success(data) => data }
```

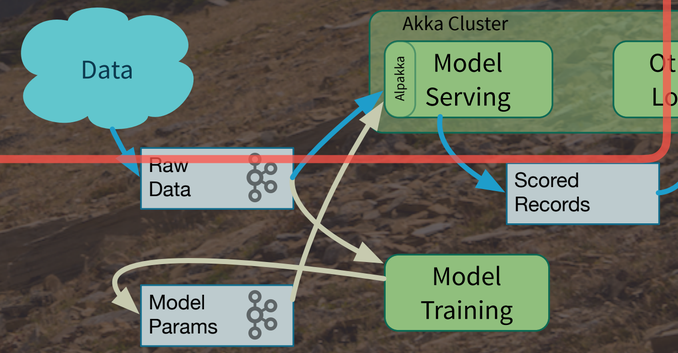
```
val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
      Subscriptions.topics(modelTopic))
    .map(input => Model.parseBytes(input.value()))
    .collect{ case Success(mod) => mod }
    .map(model => ModelImpl.findModel(model))
    .collect{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



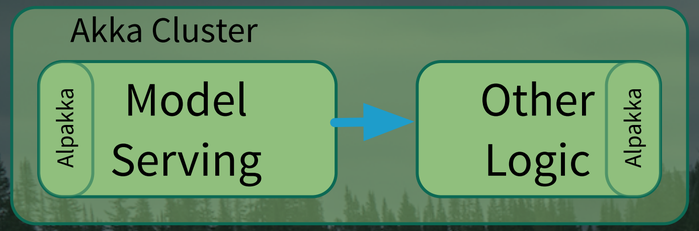
```
    .collect{ case Success(data) => data }

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))
```

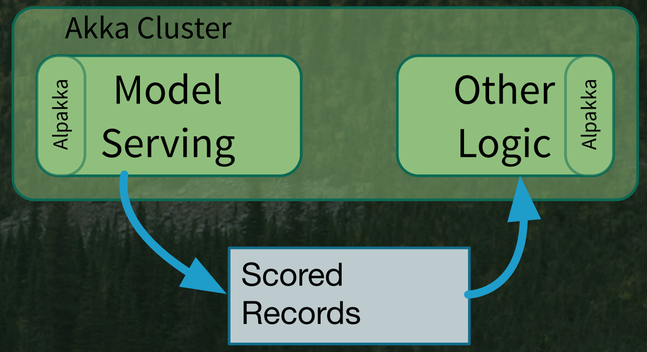



Other
Concerns
@deanwampler

Go Direct or Through Kafka?



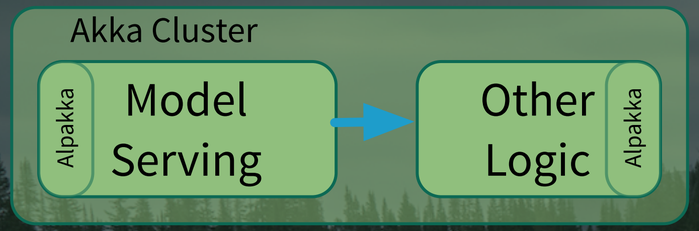
VS.



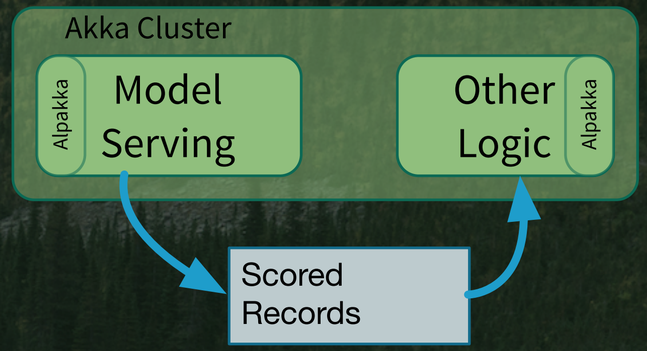
?

- Extremely low latency
- Higher latency (network, queue depth)

Go Direct or Through Kafka?



VS.

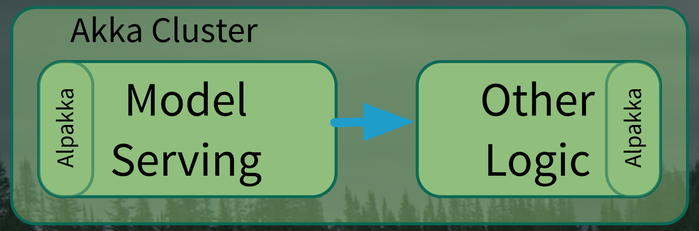


?

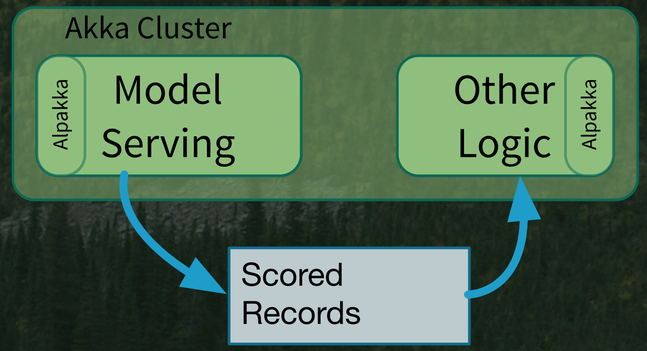
- Extremely low latency
- Minimal I/O and memory overhead. No marshaling overhead

- Higher latency (network, queue depth)
- Higher I/O and processing (marshaling) overhead

Go Direct or Through Kafka?



VS.

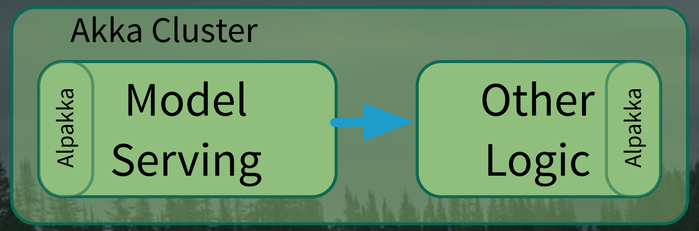


?

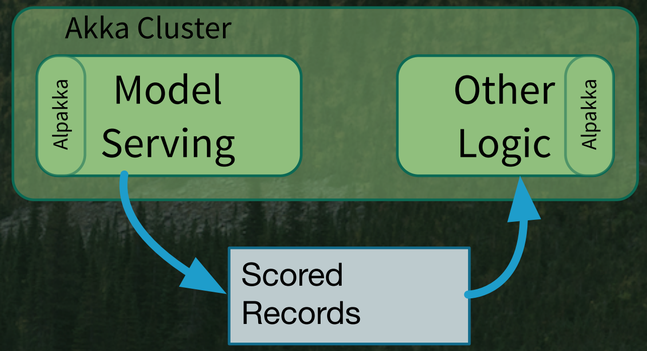
- Extremely low latency
- Minimal I/O and memory overhead. No marshaling overhead
- Hard to scale, evolve independently

- Higher latency (network, queue depth)
- Higher I/O and processing (marshaling) overhead
- Easy independent scalability, evolution

Go Direct or Through Kafka?



vs.

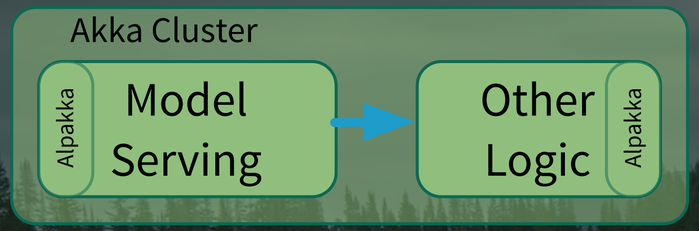


?

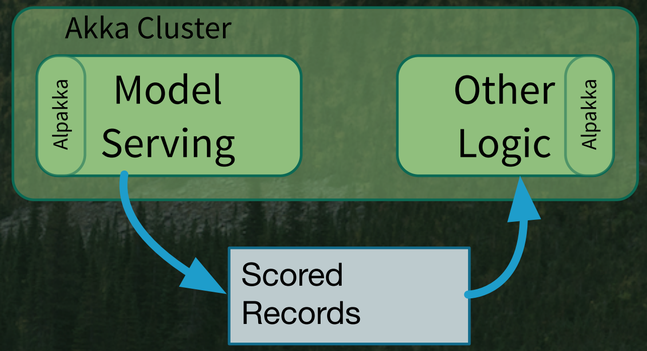
- *Reactive Streams* back pressure

- Very deep buffer (partition limited by disk size)

Go Direct or Through Kafka?



vs.



?

- *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...

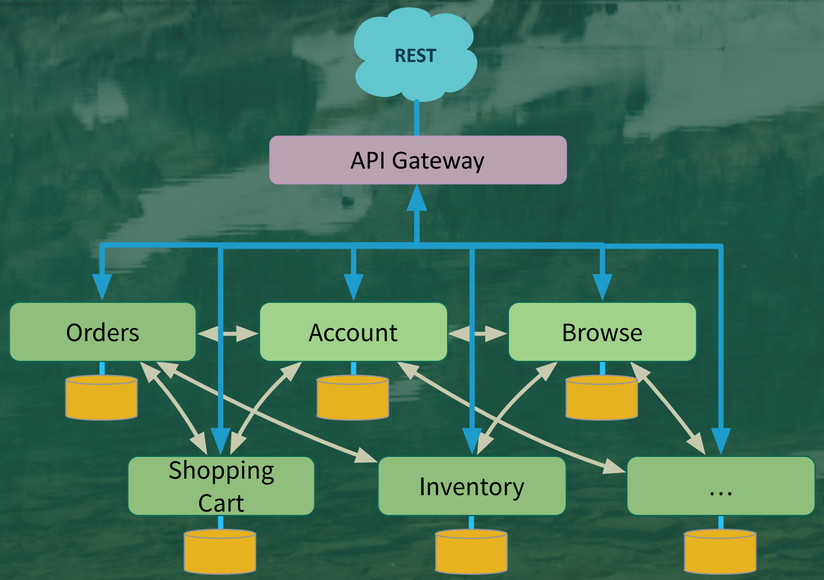
A Spectrum of Microservices

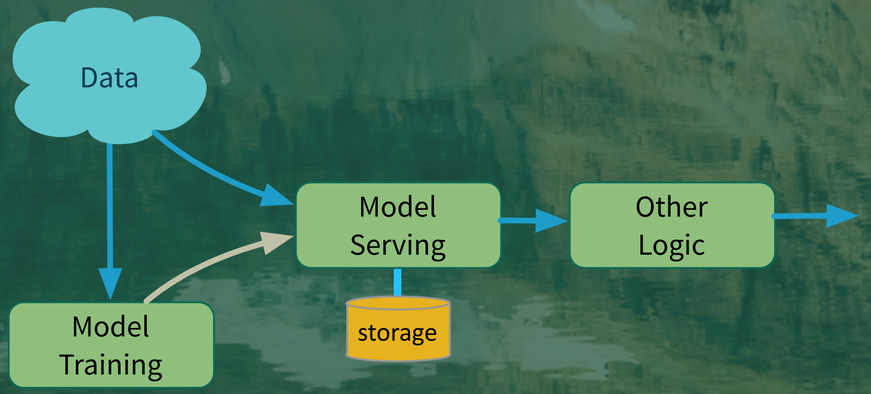
akka Streams

Streams

Event-driven μ-services

“Record-centric” μ-services





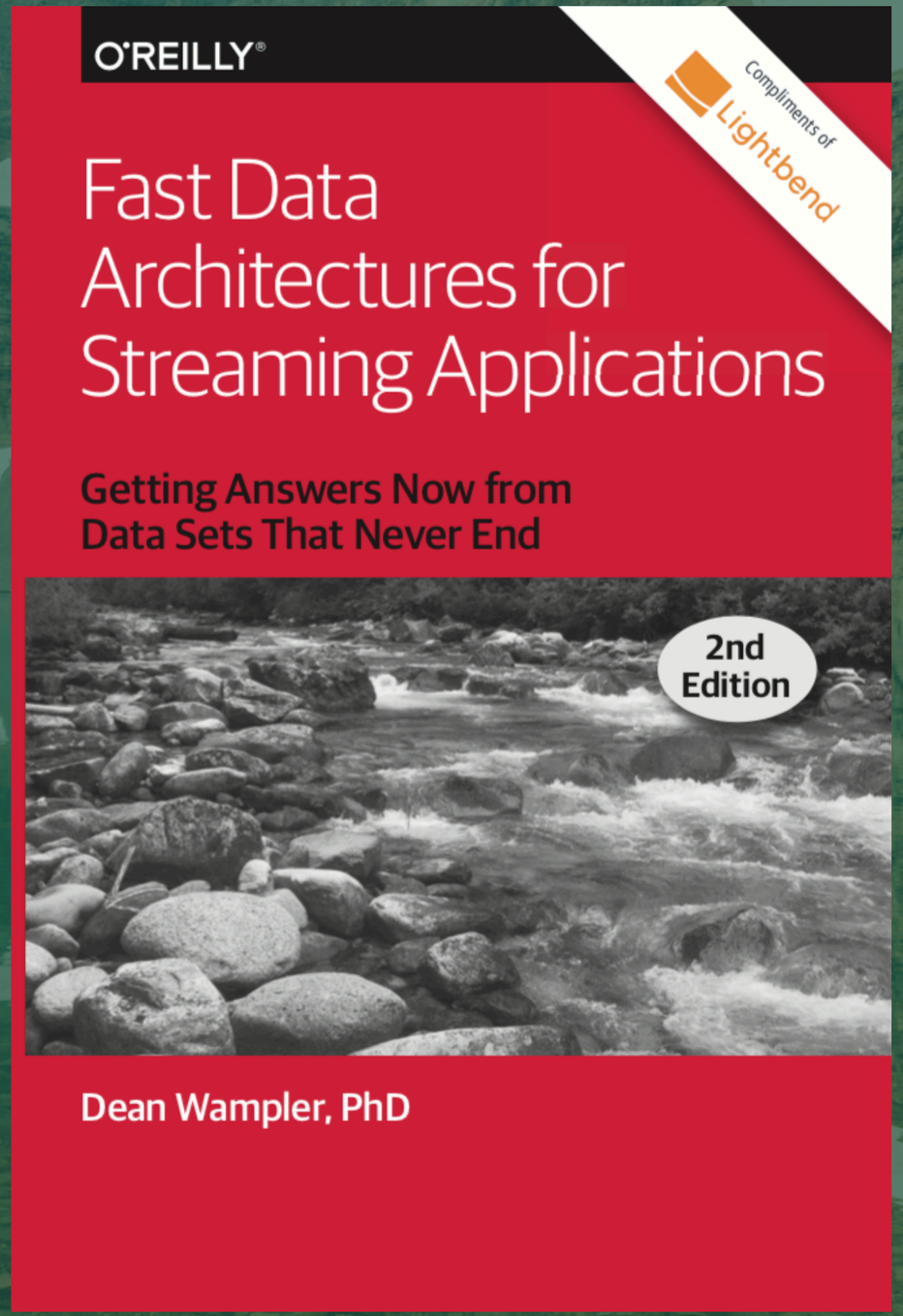
Events

Records

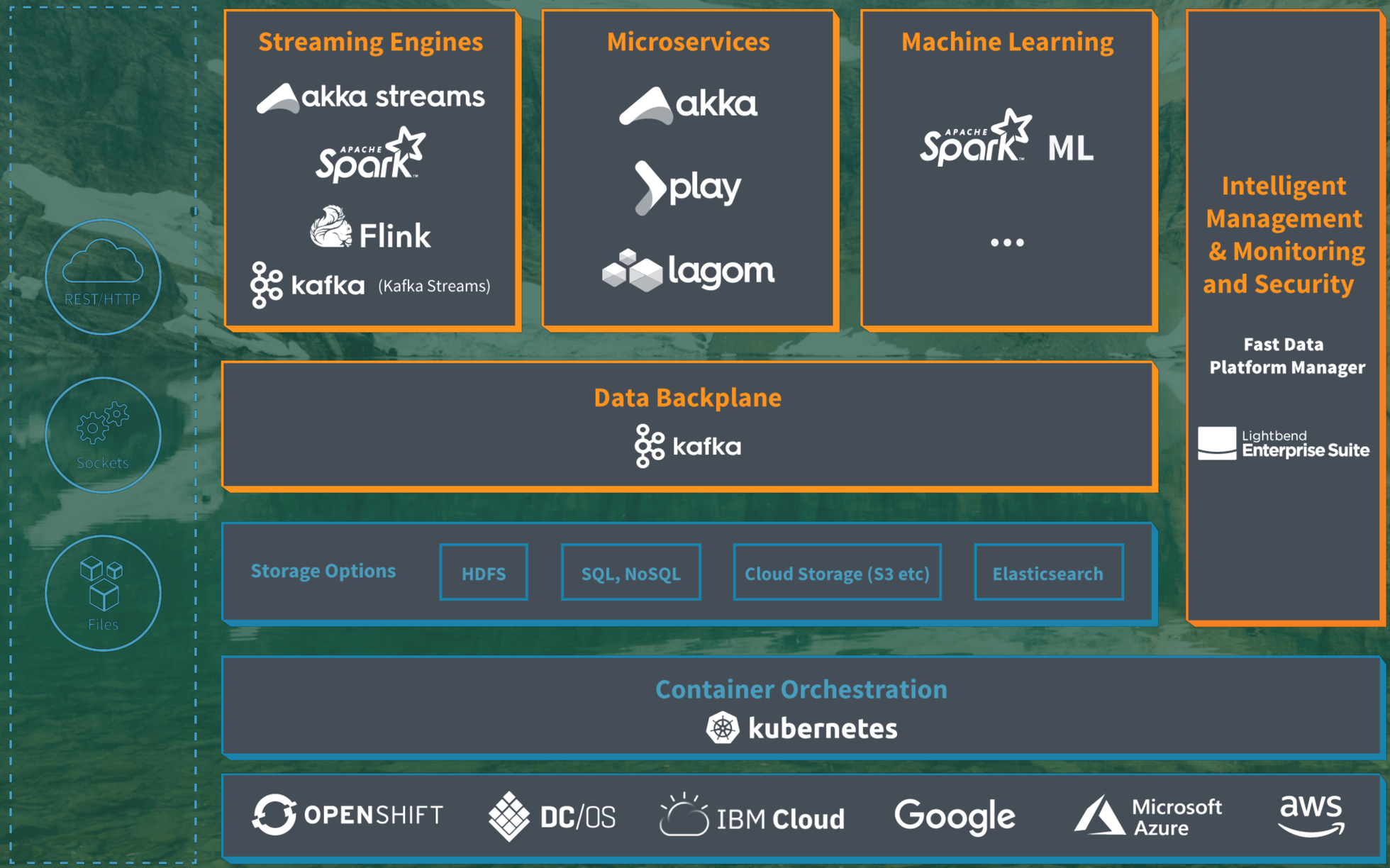
← Data Spectrum →

Free as in 🍺

New second edition!
lbnd.io/fast-data-book



lightbend.com/fast-data-platform



Questions?

@deanwampler
lightbend.com/fast-data-platform
polyglotprogramming.com/talks

Lightbend