## Building Kafka-based Microservices with Akka Streams and Kafka Streams

Boris Lublinsky and Dean Wampler, Lightbend

boris.lublinsky@lightbend.com dean.wampler@lightbend.com



©Copyright 2017-2018, Lightbend, Inc.

Apache 2.0 License. Please use as you see fit, but attribution is requested.



Lightbend

- Quick overview of streaming architectures
  - Kafka, Spark, Flink, Akka Streams, Kafka Streams

# utline

- Running example: Serving machine learning models
- Streaming in a microservice context
  - Akka Streams
  - Kafka Streams
- Wrap up

# Overview of Streaming Technologies

Why Kafka, Spark, Flink, Akka Streams, and Kafka Streams?

Lightbend

4

### Why Streaming?

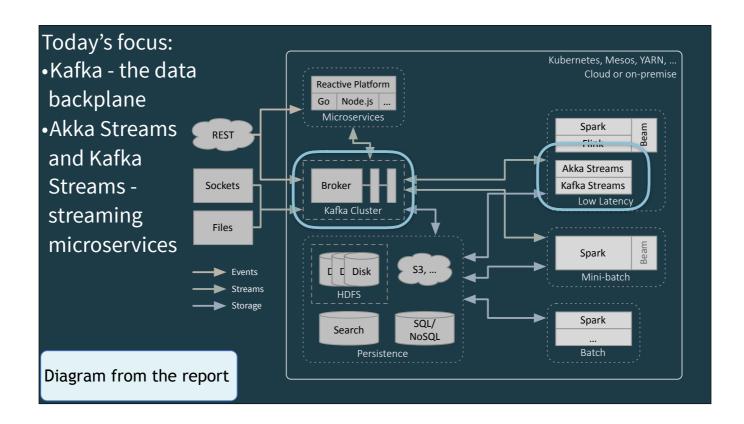
"We live as streams, but we have a tendency to think in batch. Batch might be faster (simpler), but the reality is streams"

— Fabio Yamada, Kafka Mailing List

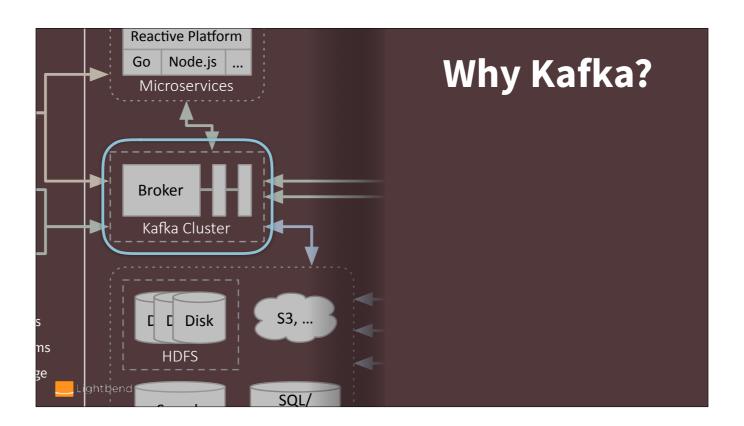
5

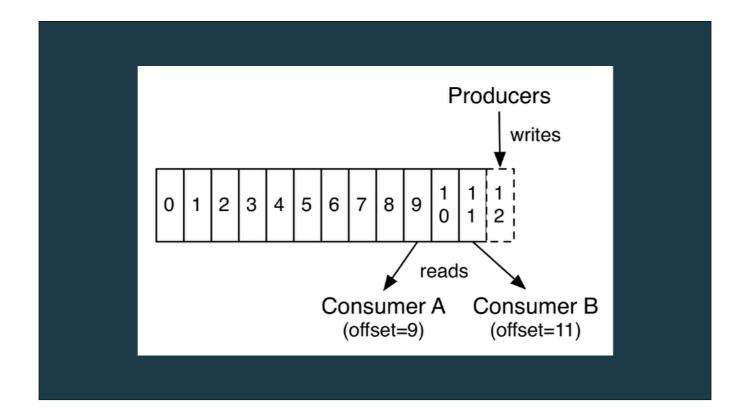


- Dean wrote this report describing the whole fast data landscape.
- bit.ly/lightbend-fast-data
- Previous talks ("Stream All the Things!") and webinars (such as this one, <a href="https://info.lightbend.com/webinar-moving-from-big-data-to-fast-data-heres-how-to-pick-the-right-streaming-engine-recording.html">https://info.lightbend.com/webinar-moving-from-big-data-to-fast-data-heres-how-to-pick-the-right-streaming-engine-recording.html</a>) have covered the whole architecture. This session dives into the next level of detail, using Akka Streams and Kafka Streams to build Kafka-based microservices



Kafka is the data backplane for high-volume data streams, which are organized by topics. Kafka has high scalability and resiliency, so it's an excellent integration tool between data producers and consumers.





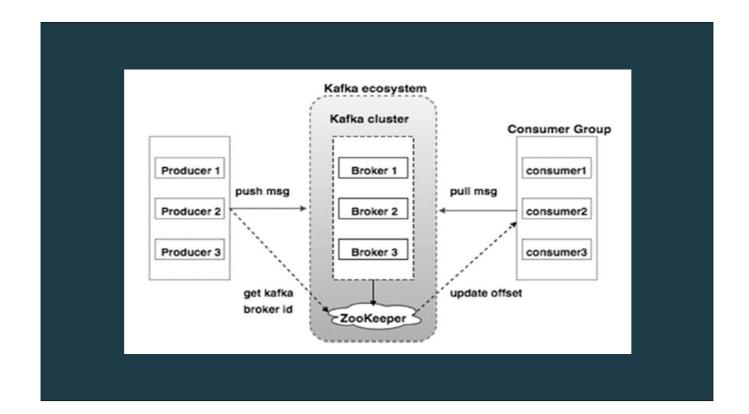
Kafka is a distributed log, storing messages sequentially. Producers always write to the end of the log, consumers can read on the log offset that they want to read from (earliest, latest, ...)

Kafka can be used as either a queue or pub sub

The main differences are:

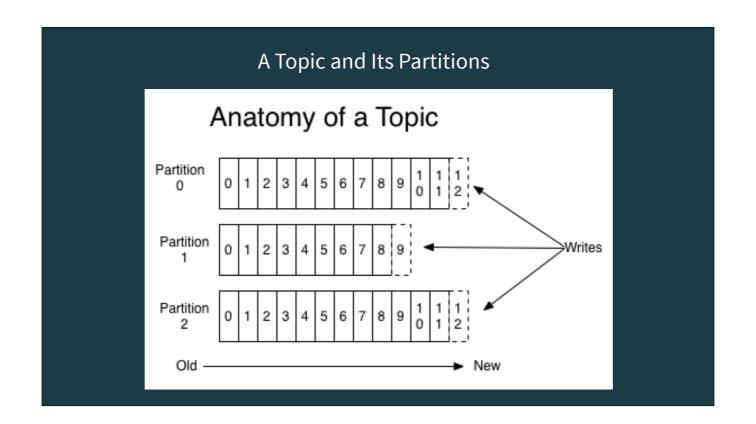
- 1. Log is persistent where queue is ephemeral (reads pop elements)
- 2. Traditional message brokers manage consumer offsets, while log systems allow users to manage offsets themselves

Alternatives to Kafka include Pravega (EMC) and Distributed Log/Pulsar (Apache)



Kafka cluster typically consists of multiple brokers to maintain load balance.

One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each broker can handle TB (based on the disk size and network performance) of messages without performance impact. Kafka broker leader election can be done by ZooKeeper.

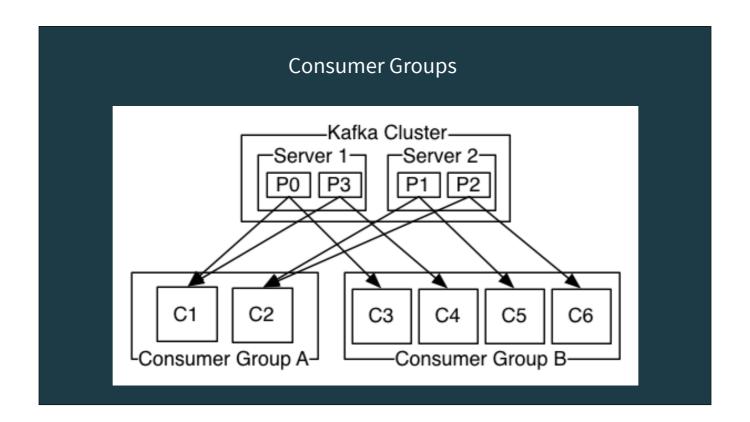


Kafka data is organized by topic

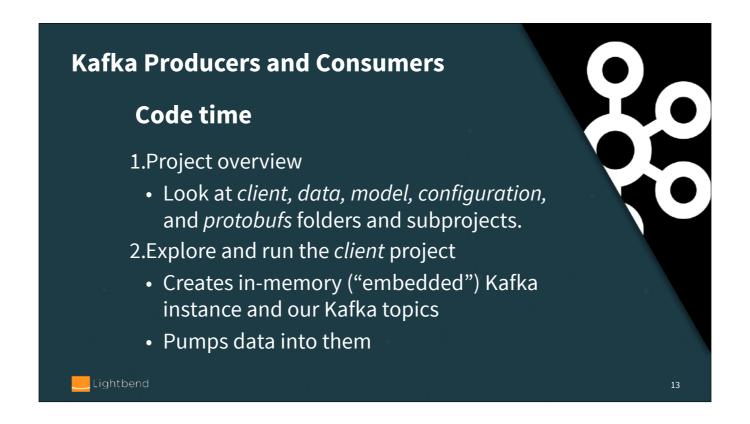
A topic can be comprised of multiple partitions.

A partition is a physical data storage artifact. Data in a partition can be replicated across multiple brokers. Data in a partition is guaranteed to be sequential. So, a topic is a logical aggregation of partitions. A topic doesn't provide any sequential guarantee (except a one-partition topic, where it's "accidental"). Partitioning is an important scalability mechanism - individual consumers can read dedicated partitions.

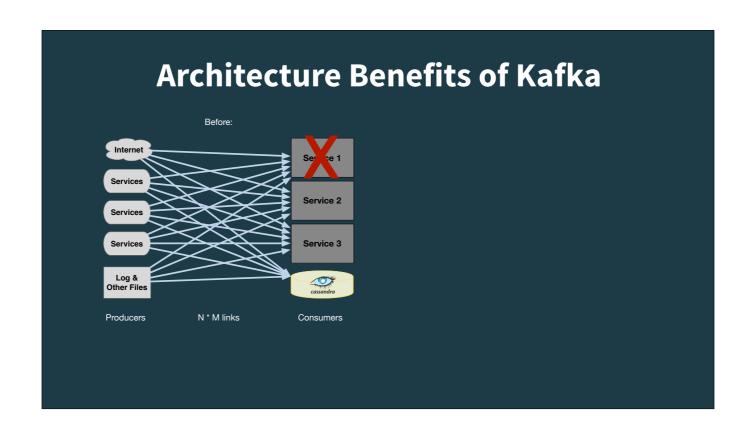
Partitioning mechanisms - round-robin, key (hash) based, custom. Consider the sequential property when designing partitioning.



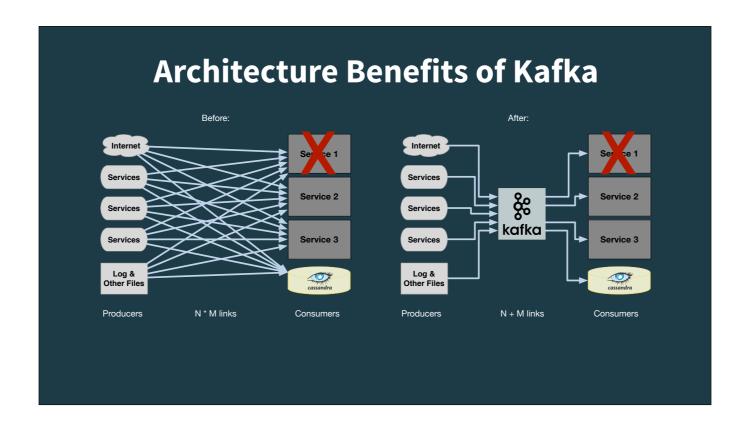
Consumers label themselves with a consumer group name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group (compare to queue semantics in traditional messaging). Consumer instances can be in separate processes or on separate machines.



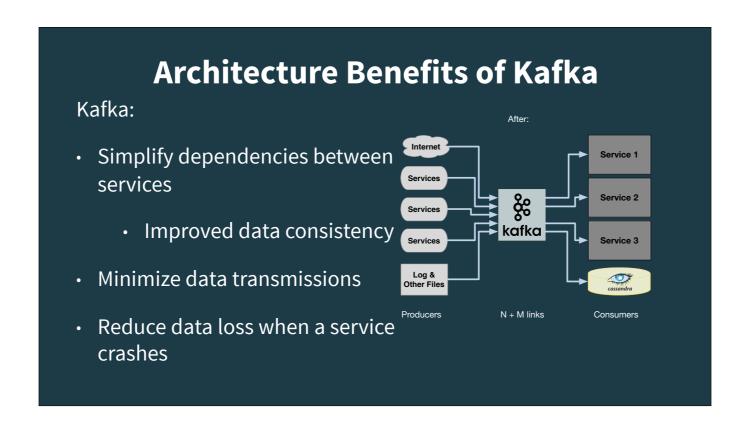
We'll walk through the whole project, to get the lay of the land, then look at the subprojects listed. We'll deep dive into the actual Akka Streams and Kafka Streams subprojects later. The embedded Kafka approach is suitable for non-production scenarios only, like learning;)



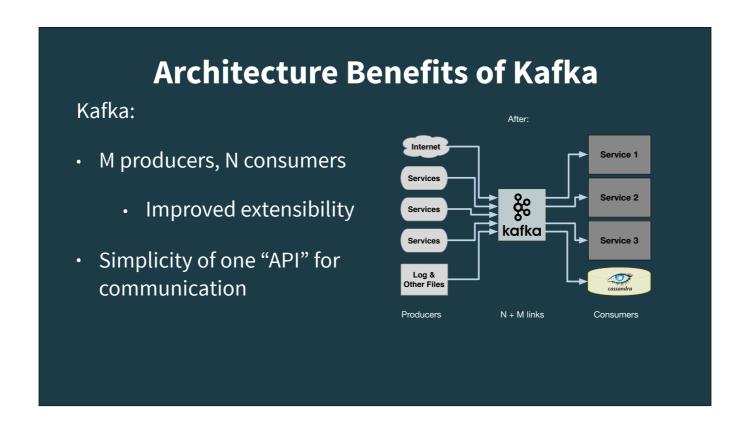
We're arguing that you should use Kafka as the data backplane in your architectures. Why? First, point to point spaghetti integration quickly becomes unmanageable as the amount of services grows



Kafka can simplify the situation by providing a single backbone which is used by all services (there are of coarse topics, but they are more logical then physical connections). Additionally Kafka persistence provides robustness when a service crashes (data is captured safely, waiting for the service to be restarted) - see also temporal decoupling, and provide the simplicity of one "API" for communicating between services.



Kafka can significantly improve decoupling (no service specific endpoints, temporal decoupling), It minimize the amount of data send over network, each producer writes data to Kafka, instead of writing it to multiple consumers. This also improves data consistency - the same data is consumed by all consumers. Extensibility is greatly simplified - adding new consumers does not require any changes to producers, and provide the simplicity of one "API" for communicating between services.



Kafka can significantly improve decoupling (no service specific endpoints, temporal decoupling), It minimize the amount of data send over network, each producer writes data to Kafka, instead of writing it to multiple consumers. This also improves data consistency - the same data is consumed by all consumers. Extensibility is greatly simplified - adding new consumers does not require any changes to producers, and provide the simplicity of one "API" for communicating between services.

#### Kafka message size considerations

- Should I use Kafka for all messages?
  - Optimal performance: messages size ~ few KB.
  - Larger messages put heavy load on brokers and is very inefficient.
  - It is inefficient on producers and consumers as well.

Kafka perf tips from Manoj Khangaonkar on the Kafka mailing list.

#### Kafka message size considerations

- What if my messages are very large?
  - Use messaging by reference
    - Store a message in S3, HDFS, etc.
    - Send the *reference* to the location as a Kafka message

Kafka perf tips from Manoj Khangaonkar on the Kafka mailing list.

#### Message compatibility for Kafka

- Is it okay if messages have different schemas?
  - If so, handled at run time ("dynamic typing") or design time ("static typing")?
- How is message type determined?
  - Registry or repository?
  - Embedding in Kafka headers?

Keep track of message schema (ID) used for generated code in both consumer and producer. Consumer might additionally have a list of "compatible schema IDs" Once the message is received, check the schema ID published by the producer and decide whether it can be used. If ID is unknown, out message into DLQ

#### **Message versioning**

- What happens if a Producer needs to create a new message version that's incompatible with previous versions?
  - Topic versioning similar to endpoint versioning used by services.
  - Should you start new services instead?

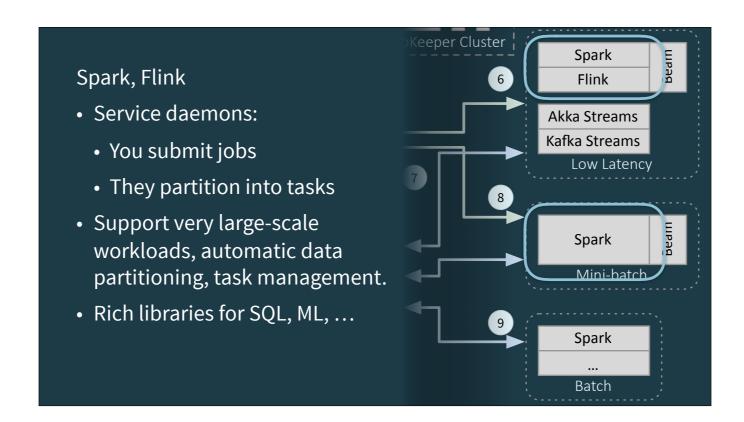
### **Streaming Architectures**

#### Two approaches:

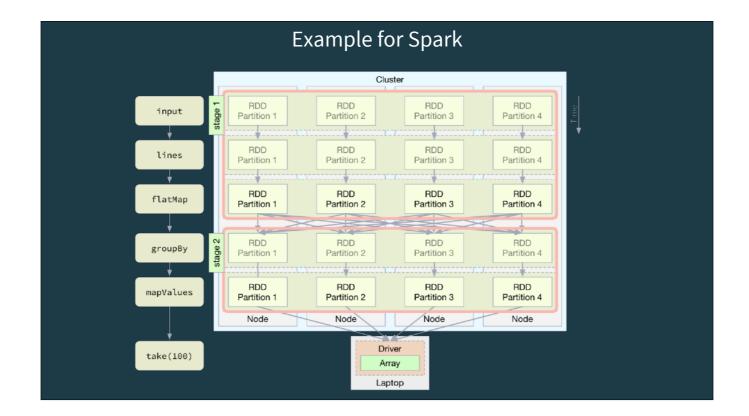
- Stream processing "engines"
  - Spark, Flink, Beam, ...
- Streaming libraries
  - Akka Streams, Kafka Streams, ...

Lightbend

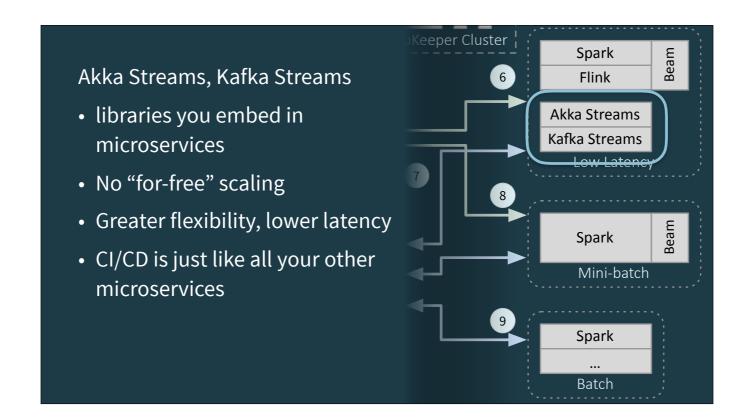
22



They support highly scalable jobs, where they manage all the issues of scheduling processes, etc. You submit jobs to run to these running daemons. They handle scalability, failover, load balancing, etc. for you.



You have to write jobs, using their APIs, that conform to their programming model. But if you do, Spark and Flink do a great deal of work under the hood for you!



Much more flexible deployment and configuration options, compared to Spark and Flink, but more effort is required by you to run them. They are "just libraries", so there is a lot of flexibility and interoperation capabilities.

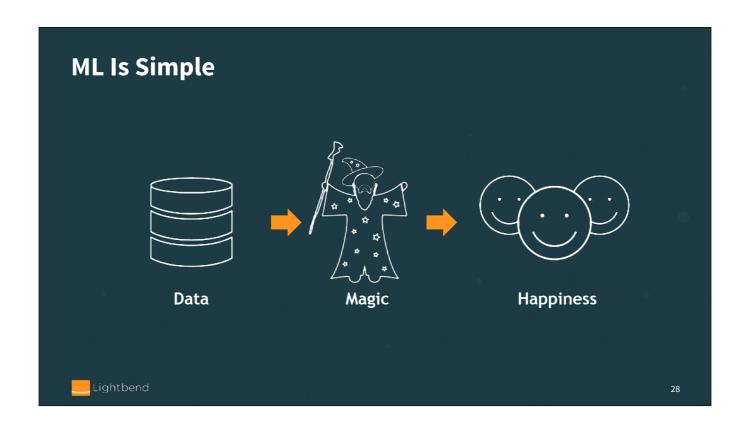
## Machine Learning and Model Serving: A Quick Introduction

\_\_\_Lightbend

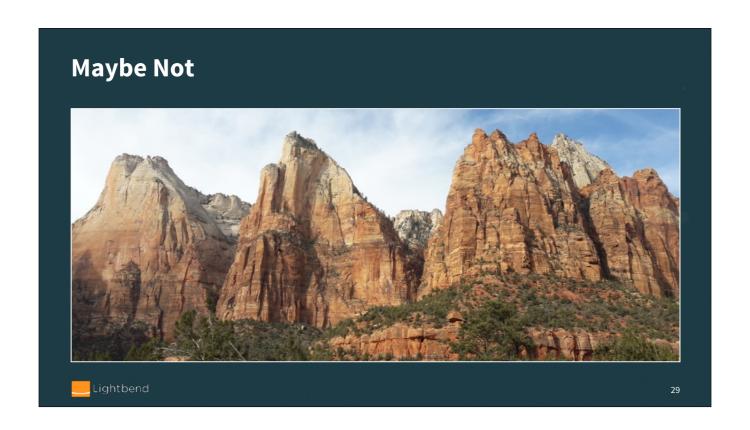
26



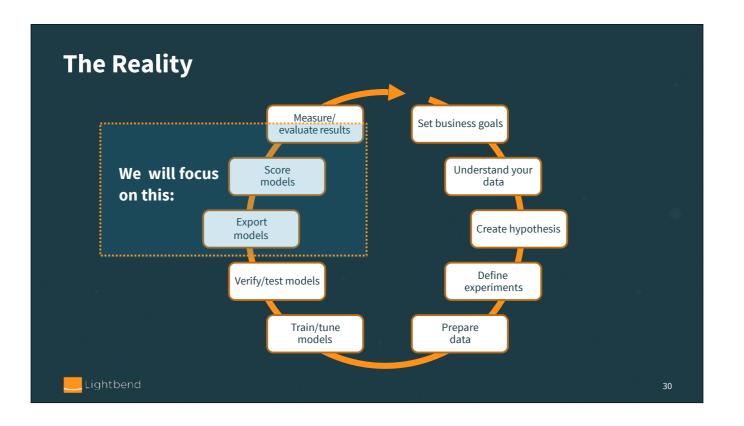
Our concrete examples are based on the content of this report by Boris, on different techniques for serving ML models in a streaming context.

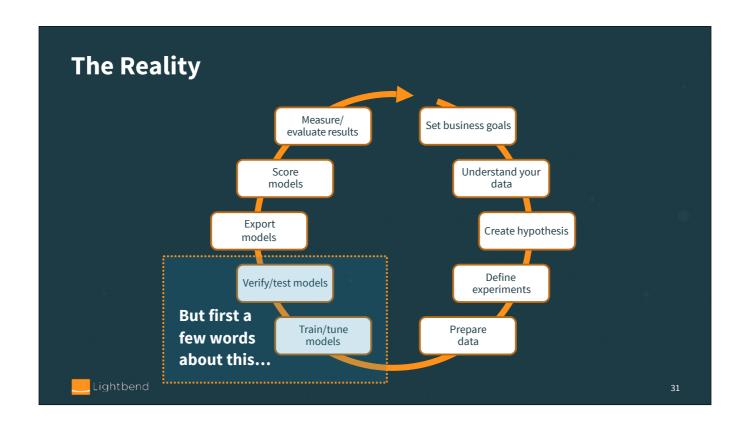


Get a lot of data Sprinkle some magic And be happy with results

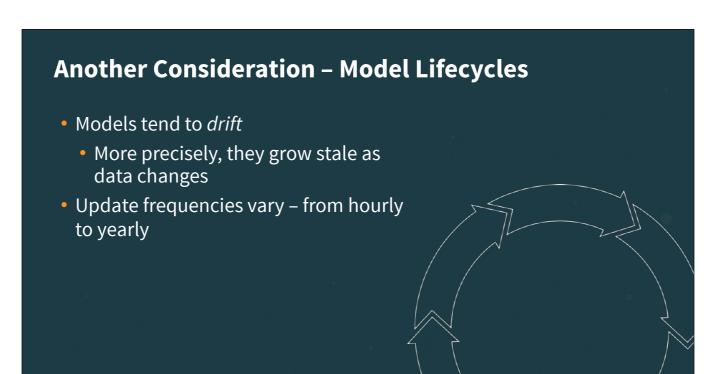


Not only the climb is steep, but you are not sure which peak to climb Court of the Patriarchs at Zion National park





•A brief diversion about this part of the pipeline.



Lightbend

#### Trends in Model Training...

- Original approach Batch retrain at ad-hoc for fixed intervals.
  - Using Hadoop, off-line data science tool chains, etc.
- Challenges:
  - How do you deliver updated models to production?
  - What if you have *thousands* of models?
  - When should you retrain?

Lightbend

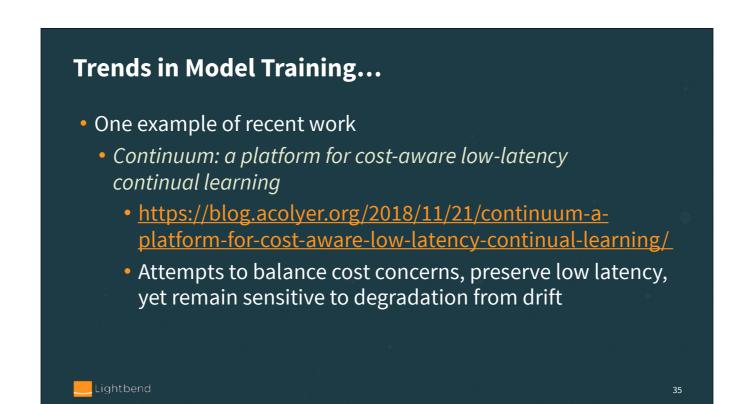
3

#### Trends in Model Training...

- Moving to automated retraining
  - Spark, X as a Service (where X = TensorFlow, ...)
- Challenges:
  - How do you deliver updated models to production? CI/CD pipeline and techniques we'll discuss
  - What if you have thousands of models? Must measure and automate everything
  - When should you retrain? *Metrics for model drift... (next slide)*

Lightbend

3



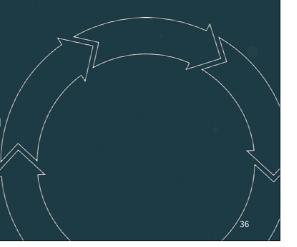
This blog post discusses "Continuum" and gives you a a good, overall sense of research directions and concerns people are attempting to address.

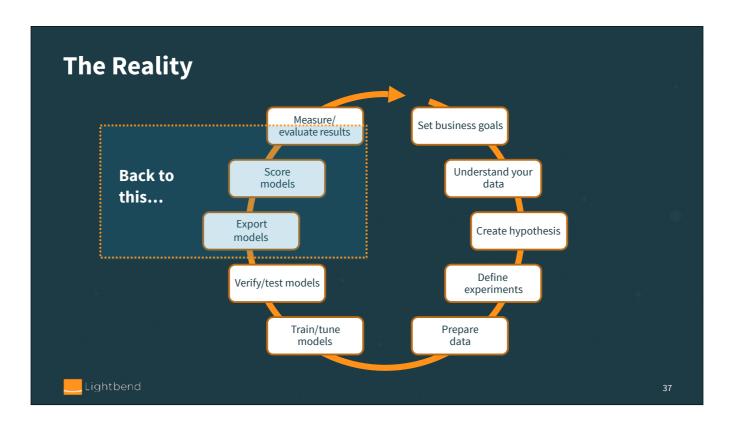


- Your organization may need to track model versions for auditing:
  - What model instance was used to score this (controversial?) record?
  - Why did it score it a particular way?
  - What were the model parameters?

Lightbend

 Which other records were scored with this model?





#### What Is The Model?

A model is a *function* transforming inputs to outputs -y = f(x)

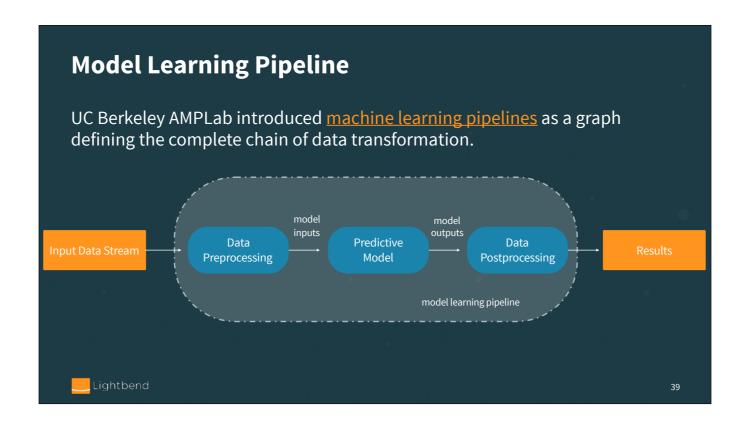
Linear regression:  $y = a_c + a_1 \times x_1 + ... + a_n \times x_n$ 

Neural network:  $f(x) = K(\sum_i w_i g_i(x))$ 

From the implementation point of view, it is just function composition.



Lightbend



UC Berkeley AMPLab introduced machine learning pipelines as a graph defining the complete chain of data transformation. The advantage of such approach

It captures the whole processing pipeline including data preparation transformations, machine learning itself and any required post processing of the ML results. Although a single predictive model is shown on this picture, in reality several models can be chained to gather or composed in any other way. See PMML documentation for description of different model composition approaches.

Definition of the complete model allows for optimization of the data processing.

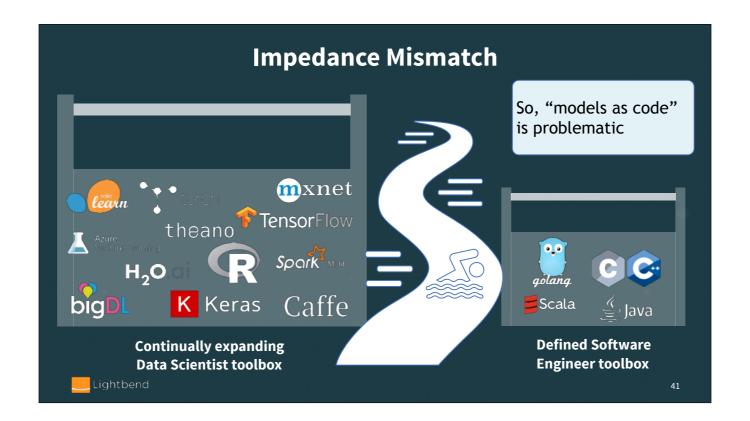
This notion of machine learning pipelines has been adopted by many applications including SparkML, Tensorflow, PMML, etc.

## **Traditional Approach to Model Serving**

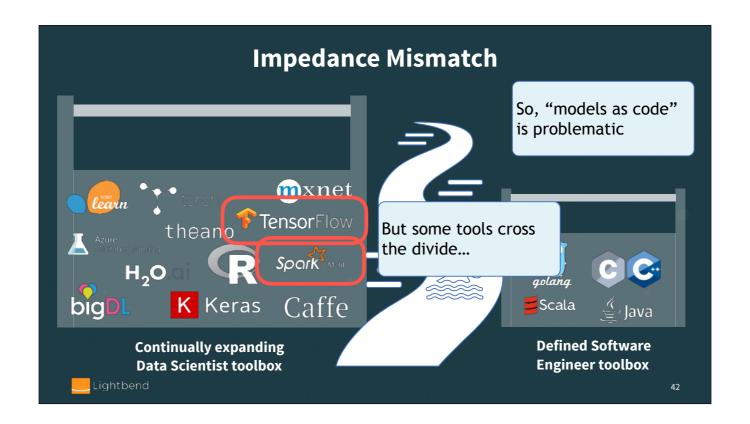
- *Model as code*: source code in some language implements the model
- This code is linked into the model serving applications

Why is this problematic?

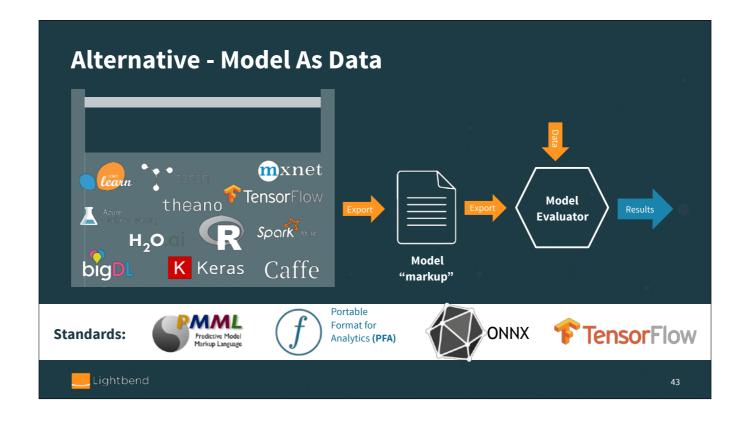
Lightbend



In his talk at the last Flink Forward, Ted Dunning discussed the fact that with multiple tools available to Data scientists, they tend to use different tools for solving different problems and as a result they are not very keen on tools standardization. This creates a problem for software engineers trying to use "proprietary" model serving tools supporting specific machine learning technologies. As data scientists evaluate and introduce new technologies for machine learning, software engineers are forced to introduce new software packages supporting model scoring for these additional technologies.



We'll return to this point below.



In order to overcome these differences, Data Mining Group have introduced 2 standards - Predictive Model Markup Language (PMML) and Portable Format for Analytics (PFA), both suited for description of the models that need to be served. Introduction of these models led to creation of several software products dedicated to "generic" model serving, for example Openscoring, Open data group, etc.

ONNX is a new standard for deep learning models, but it is not supported by Tensorlow.

Another de facto standard for machine learning is Tensorflow, which is widely used for both machine learning and model serving. Although it is a proprietary format, it is used so widely that it becomes a standard

The result of this standardization is creation of the open source projects, supporting these formats - JPMML and Hadrian which are gaining more and more adoption for building model serving implementations, for example ING, R implementation, SparkML support, Flink support, etc. Tensorflow also released Tensorflow java APIs, which are used in a Flink TensorFlow

### **Considerations for Interchange Tools**

- Do you training tools support an exchange format (PMML, PFA, ...)?
- Do you serving tools support the format?
- Is there support on both ends for the model types you want to use, e.g., random forests, neural networks, ...?
- Does the *serving* implementation faithfully reproduce the results of your *training* environment?

Lightbend

1

Implementations for these tools have been spotty, so carefully consider these questions when evaluating this approach.

# **Exporting Models As Data With PMML**

There are already a lot of export options



https://github.com/jpmml/jpmml-sparkml



https://github.com/jpmml/jpmml-sklearn



https://github.com/jpmml/jpmml-r



https://github.com/jpmml/jpmml-tensorflow





### **Evaluating PMML Model**

There are also a few PMML evaluators





https://github.com/jpmml/jpmml-evaluator



https://github.com/opendatagroup/augustus

See also this list of products: <a href="http://dmg.org/pmml/products.html">http://dmg.org/pmml/products.html</a>



### **Exporting NN Models With ONNX**

- Created by Microsoft and Facebook
  - https://thenewstack.io/facebook-microsoft-bring-interoperable-models-machine-learning-toolkits/
- Supported tools: <a href="https://onnx.ai/supported-tools">https://onnx.ai/supported-tools</a>
- Git Repo: <a href="https://github.com/onnx">https://github.com/onnx</a>
- Designed for when you use one NN framework for R&D, another for production, but...
  - It is new and not mature
  - Tensorflow is supported, but not by Google

Lightbend

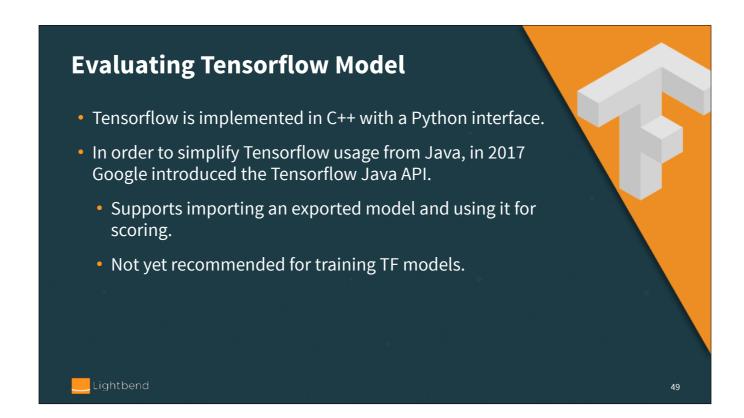
47

For our purposes here, you might use ONNX if you are only concerned with serving neural networks, not other kinds of ML. You'll still need to decide what tool to run in your production environment and decide the best way to integrate with your microservices.

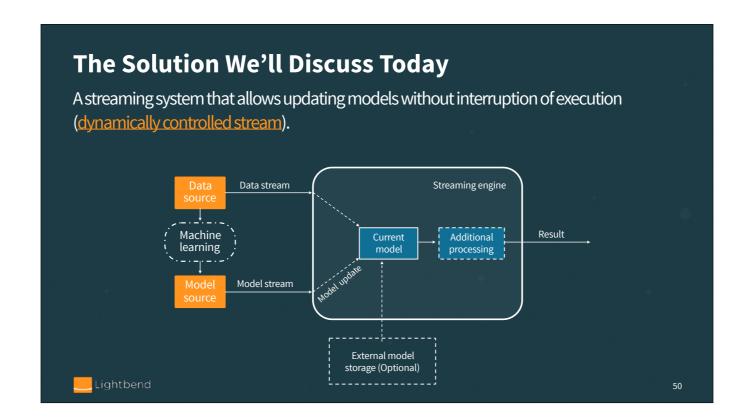
#### **Exporting Model As Data With Tensorflow**

- Tensorflow execution is based on Tensors and Graphs
- Tensors are defined as multilinear functions which consist of various vector variables
- A computational graph is a series of Tensorflow operations arranged into graph of nodes
- Tensorflow supports exporting graphs in the form of binary protocol buffers
- There are two different export format optimized graph and a new format saved model

Lightbend



We have a previously-trained TF model on the included "Wine Records" data. We'll import that model to do scoring.



The majority of machine learning implementations are based on running model serving as a REST service, which might not be appropriate for high-volume data processing or streaming systems, since they require recoding/restarting systems for model updates. For example, Flink TensorFlow or Flink JPPML.

```
Model Representation (Protobufs)
// On the wire
syntax = "proto3";
// Description of the trained model.
message ModelDescriptor {
 string name = 1; // Model name
 string description = 2; // Human readable
 string dataType = 3; // Data type for which this model is applied.
 enum ModelType { // Model type
                                                   ModelType modeltype = 4;
  TENSORFLOW = 0;
                                                   oneof MessageContent {
  TENSORFLOWSAVED = 2;
                                                    // Byte array containing the model
  PMML = 2;
                                                    bytes data = 5;
                                                    string location = 6;
  Lightbend
```

You need a neutral representation format that can be shared between different tools and over the wire. Protobufs (from Google) is one of the popular options. Recall that this is the format used for model export by TensorFlow. Here is an example.

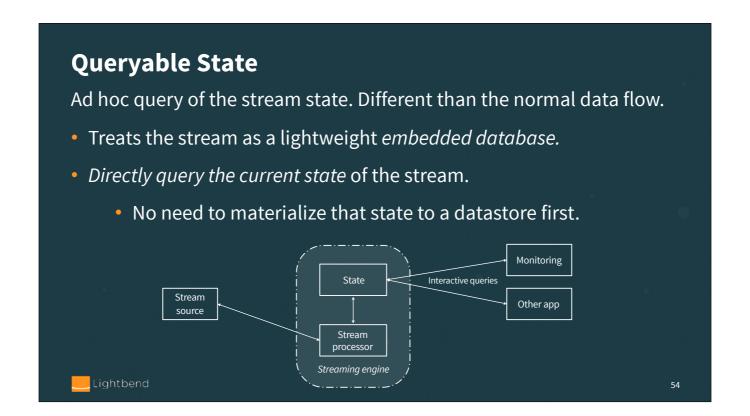
```
Model Code Abstraction (Scala and Java)
trait Model {
                                                  public interface Model extends Serializable {
def score(input: Any): Any
                                                  public Object score(Object input);
def cleanup() : Unit
                                                  Unit cleanup();
def toBytes() : Array[Byte]
                                                  byte[] toBytes();
def getType : Long
                                                  long getType();
trait ModelFactory {
                                                  public interface ModelFactory {
def create(d : ModelDescriptor) : Option[Model]
                                                  Optional<Model> create(ModelDescriptor d);
def restore(bytes : Array[Byte]) : Model
                                                  Model restore(byte[] bytes);
     Lightbend
```

Corresponding Scala code that can be generated from the description.

#### **Side Note: Monitoring**

Model monitoring should provide information about usage, behavior, performance and lifecycle of the deployed models

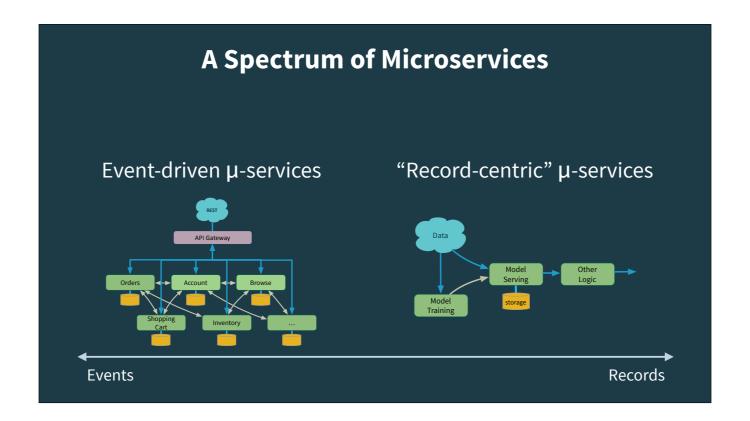
```
case class ModelToServeStats(
                                         // Scala example
name: String,
                                         // Model name
                                         // Model descriptor
description: String,
modelType: ModelDescriptor.ModelType, // Model type
                                        // Start time of model usage
since: Long,
                                        // Number of records scored
usage:Long=0,
duration: Double = 0.0,
                                        // Time spent on scoring
min: Long = Long.MaxValue,
                                        // Min scoring time
max: Long = Long.MinValue
                                        // Max scoring time
  Lightbend
```



Kafka Streams and Flink have built-in support for this and it's being added to Spark Streaming. We'll show how to use other Akka features to provide the same ability in a straightforward way for Akka Streams.

# Microservices for Streaming Data Akka Streams vs. Kafka Streams

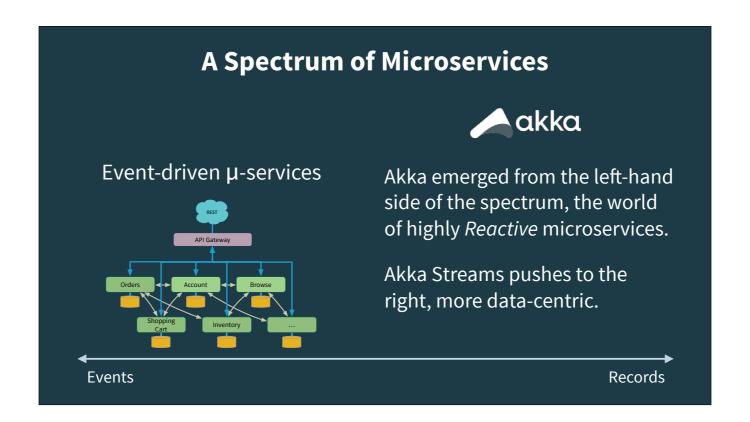
Lightbend



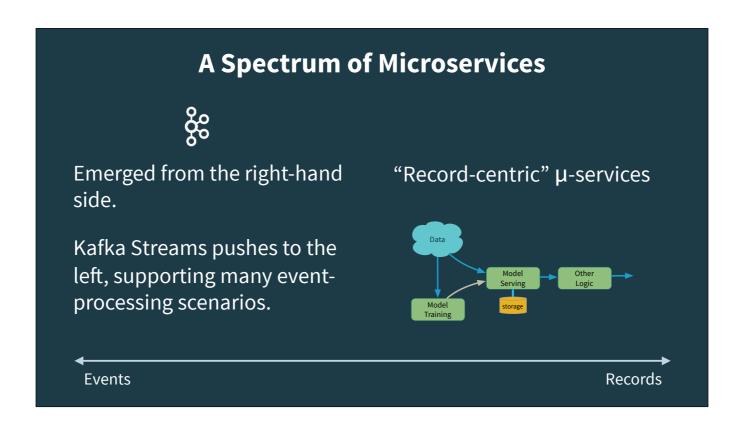
By event-driven microservices, I mean that each individual datum is treated as a specific event that triggers some activity, like steps in a shopping session. Each event requires individual handling, routing, responses, etc. REST, CQRS, and Event Sourcing are ideal for this.

Records are uniform (for a given stream), they typically represent instantiations of the same information type, for example time series; we can process them individually or as a group, for efficiency.

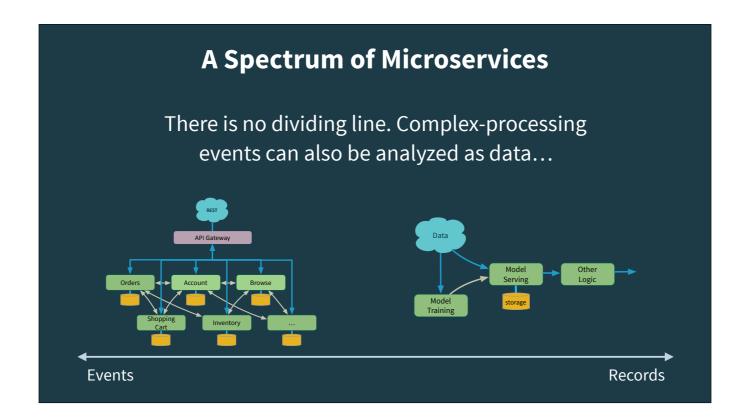
It's a spectrum because we might take those events and also route them through a data pipeline, like computing statistics or scoring against a machine learning model (as here), perhaps for fraud detection, recommendations, etc.



I think it's useful to reflect on the history of these toolkits, because their capabilities reflect their histories. Akka Actors emerged in the world of building *Reactive* microservices, those requiring high resiliency, scalability, responsiveness, CEP, and must be event driven. Akka is extremely lightweight and supports extreme parallelism, including across a cluster. However, the Akka Streams API is effectively a dataflow API, so it nicely supports many streaming data scenarios, allowing Akka to cover more of the spectrum than before.



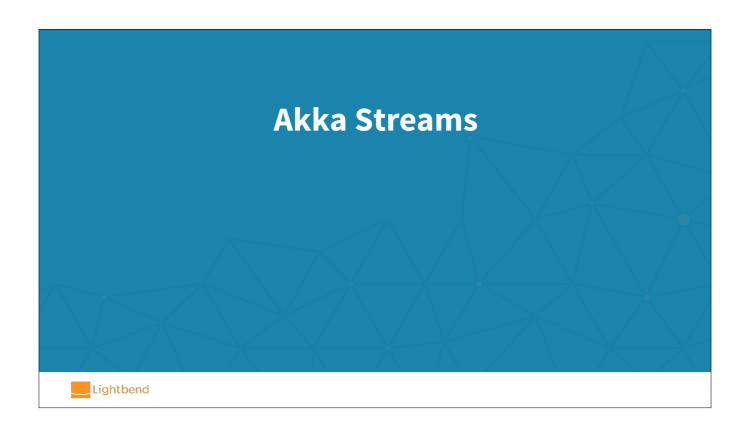
Kafka reflects the heritage of moving and managing streams of data, first at LinkedIn. But from the beginning it has been used for event-driven microservices, where the "stream" contained events, rather than records. Kafka Streams fits squarely in the record-processing world, where you define dataflows for processing and even SQL. It can also be used for event processing scenarios.

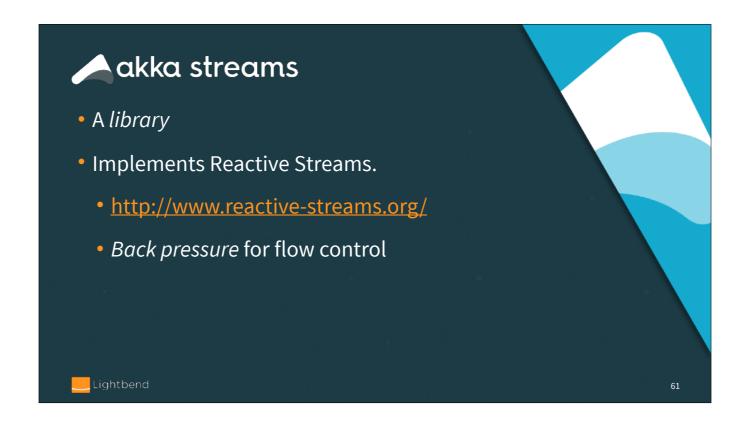


By event-driven microservices, I mean that each individual datum is treated as a specific event that triggers some activity, like steps in a shopping session. Each event requires individual handling, routing, responses, etc. REST, CQRS, and Event Sourcing are ideal for this.

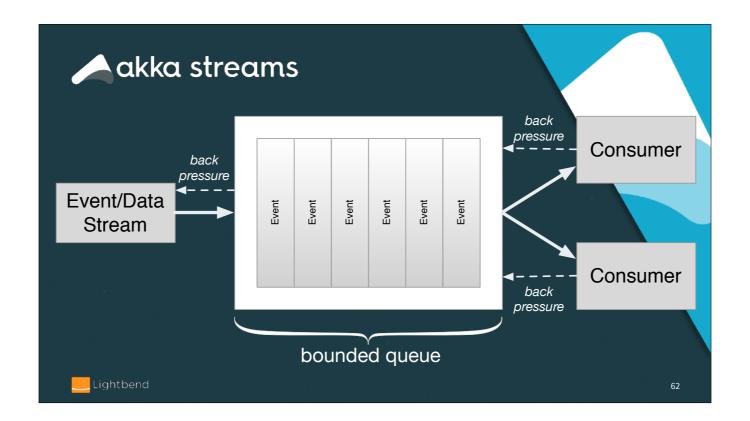
Records are uniform (for a given stream), they typically represent instantiations of the same information type, for example time series; we can process them individually or as a group, for efficiency.

It's a spectrum because we might take those events and also route them through a data pipeline, like computing statistics or scoring against a machine learning model (as here), perhaps for fraud detection, recommendations, etc.

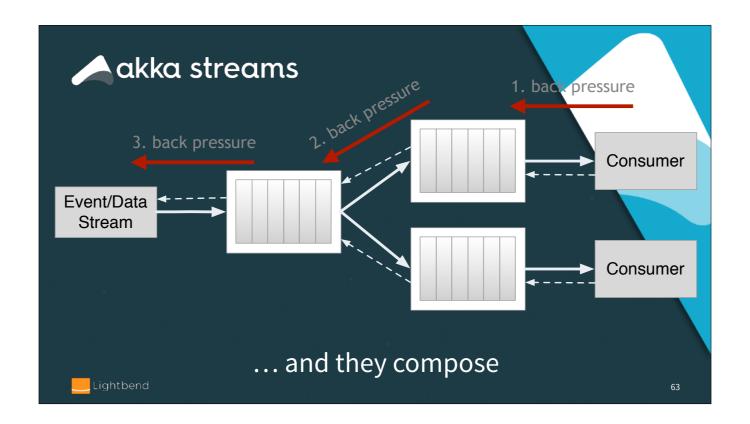




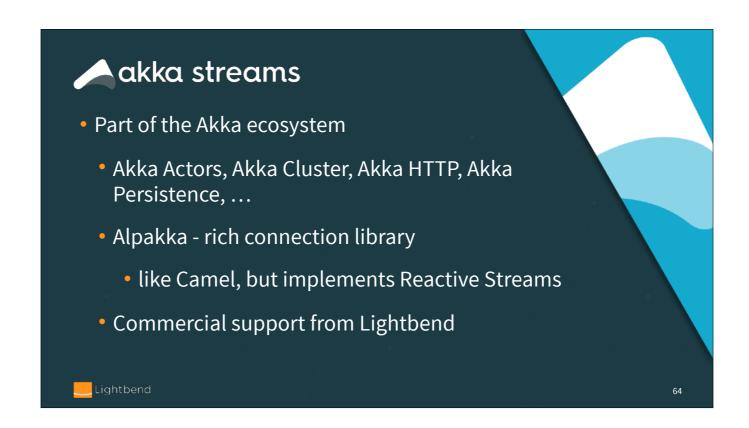
See this website for details on why back pressure is an important concept for reliable flow control, especially if you don't use something like Kafka as your "near-infinite" buffer between services.



Bounded queues are the only sensible option (even Kafka topic partitions are bounded by disk sizes), but to prevent having to drop input when it's full, consumers signal to producers to limit flow. Most implementations use a push model when flow is fine and switch to a pull model when flow control is needed.



And they compose so you get end-to-end back pressure.



Rich, mature tools for the full spectrum of microservice development.



```
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._

implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()

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

This example is in akkaStreamsModelServer/simple-akka-streams-example.sc

```
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._

implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()

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

import akka.stream.\_
import akka.stream.scaladsl.\_
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent.\_
import scala.concurrent.duration.\_

Initialize and specify now the stream is "materialized"

implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()

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

import akka.stream.\_ import akka.stream.scaladsl.\_ import akka.NotUsed import akka.actor.ActorSystem import scala.concurrent.\_ import scala.concurrent.duration.\_

implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()

Create a Source of Ints. Second type represents a hook used for "materialization" not used here

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

import akka.stream.\_ import akka.stream.scaladsl.\_ import akka.NotUsed import akka.actor.ActorSystem import scala.concurrent.\_ import scala.concurrent.duration.\_

implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()

Scan the Source and compute factorials, with a seed of 1, of type BigInt

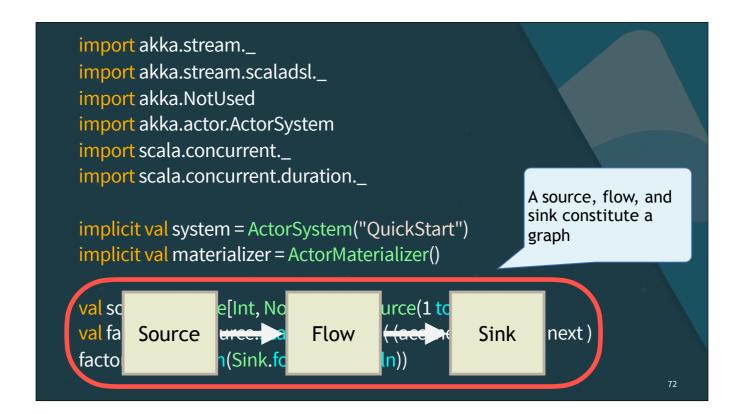
val source: Source[int, NotOsed] = Source(1 to 10)
val factorials = source.scan(BigInt(1)) ( (acc, next) => acc \* next )
factorials.runWith(Sink.foreach(println))

```
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._

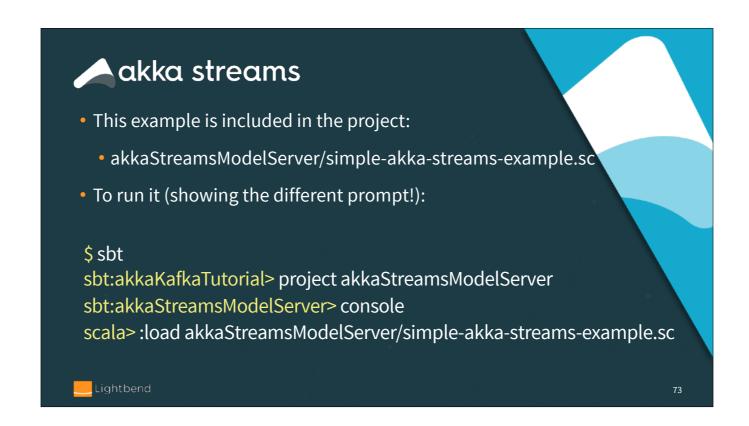
implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()

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

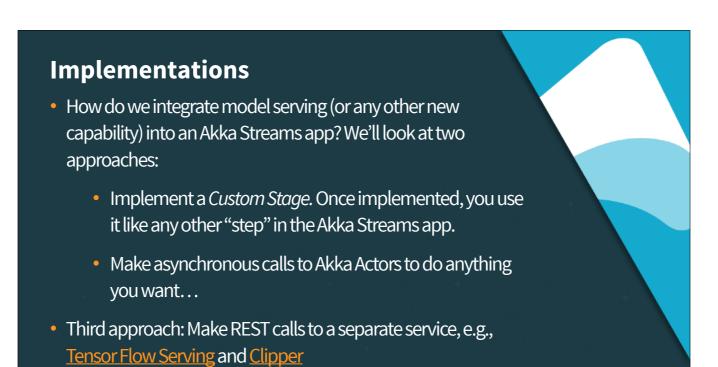
val factorials = source.scan(Bigint(1)) ((acc, next) => acc * next )
factorials.runWith(Sink.foreach(println))
```



The core concepts are sources and sinks, connected by flows. There is the notion of a Graph for more complex dataflows, but we won't discuss them further

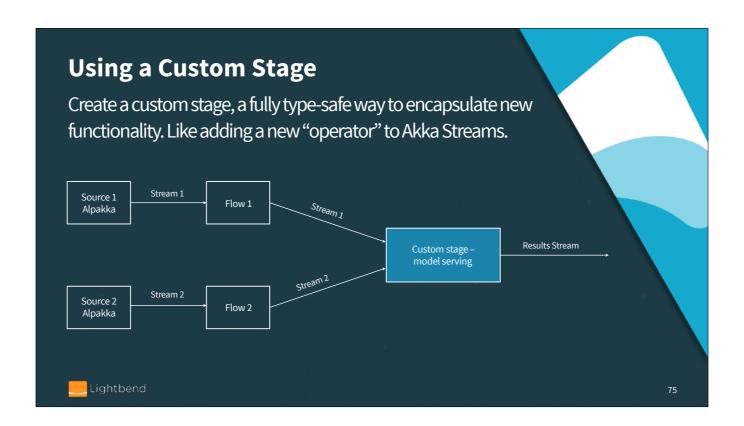


The ".sc" extension is used so that the compiler doesn't attempt to compile this Scala "script". Using ".sc" is an informal convention for such files. We used yellow to indicate the prompts (3 different shells!)

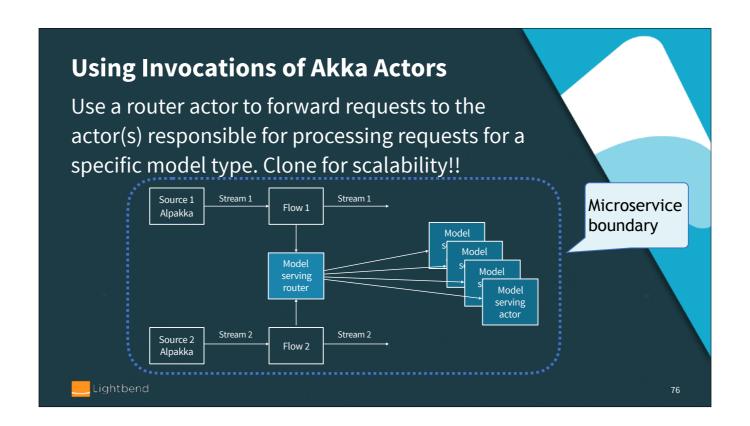


We provide two implementations. You could generalize the second approach (async calls) to invoke an external service. We won't provide examples of this option, but return later with some additional considerations about it.

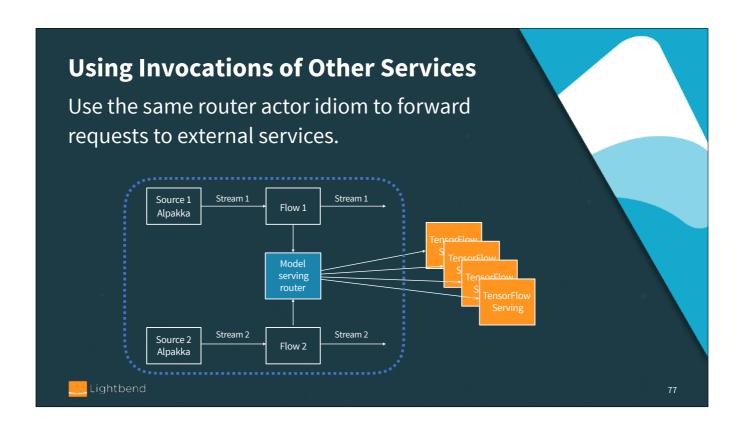
Lightbend



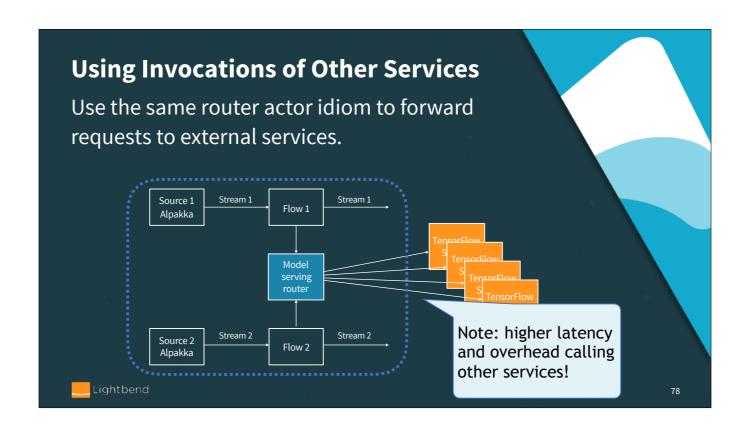
Custom stage is an elegant implementation but doesn't scale well to a large number of models. Although a stage can contain a hash map of models, all of the execution will be happening at the same place



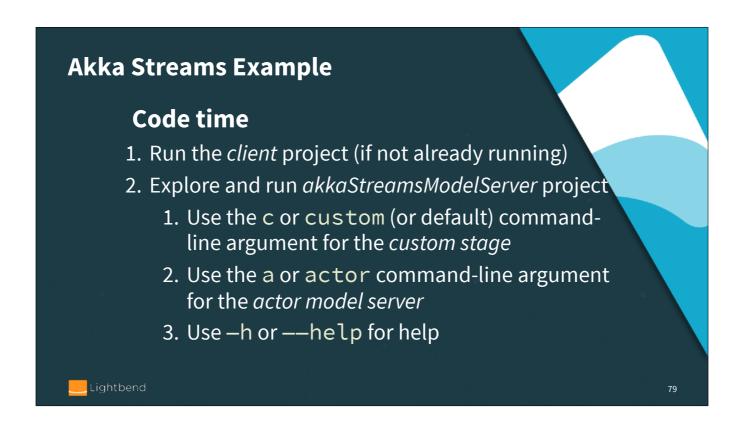
Create a routing layer: an actor that will implement model serving for a specific model (based on some key) and route messages appropriately. This way our system can serve models in parallel.



Use the same routing layer idiom: an actor that will implement model serving for a specific model (based on some key) and route messages appropriately to the external service. This way our system will serve models in parallel.



You should profile the performance impact going over a remote connection to a separate service. In contrast, actors in the previous example can be in the same process, where messaging is only a little less efficient than a function call.



Custom stage is an elegant implementation but not scale well to a large number of models. Although a stage can contain a hash map of models, all of the execution will be happening at the same place

### Akka Streams Example

#### **Check Queryable state**

- For custom stage go to http://localhost:5500/state
- For actor-based implementation go to:

http://localhost:5500/models
http://localhost:5500/state/wine

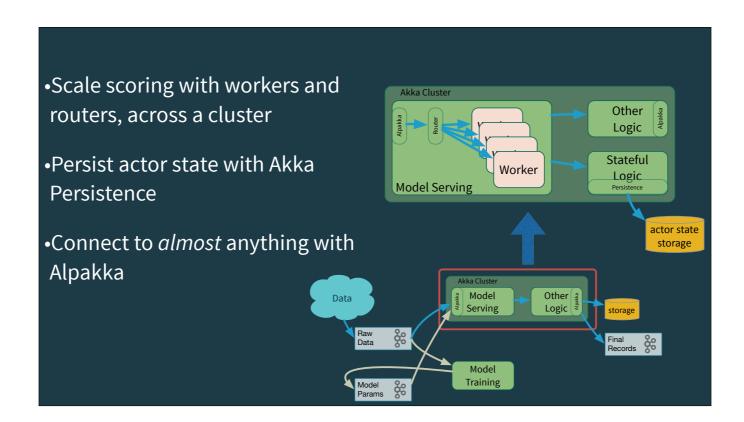


# **Exercises!**

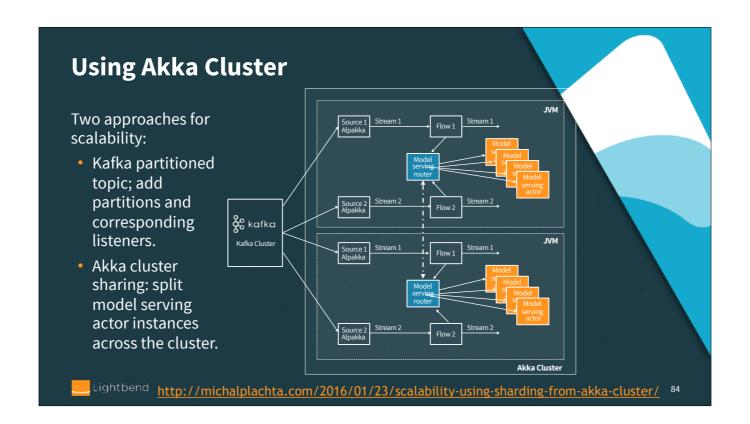
- To find them, search for // Exercise comments in the code base.
- We'll suggest some you might try first.

Lightbend

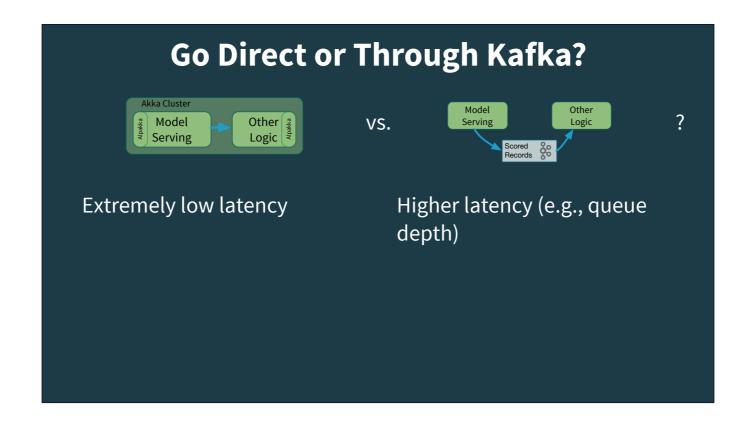


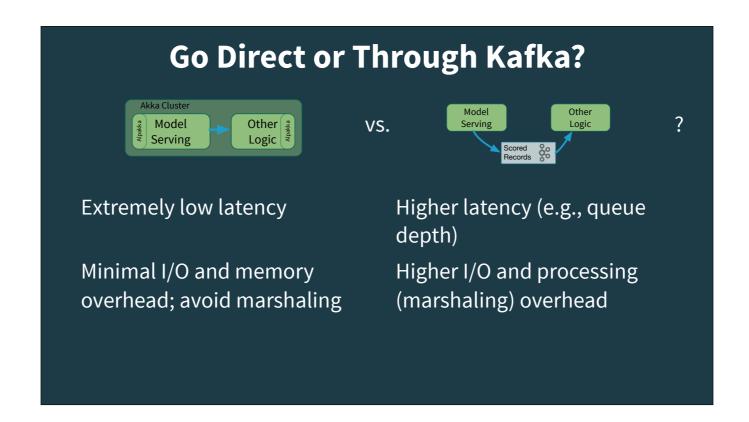


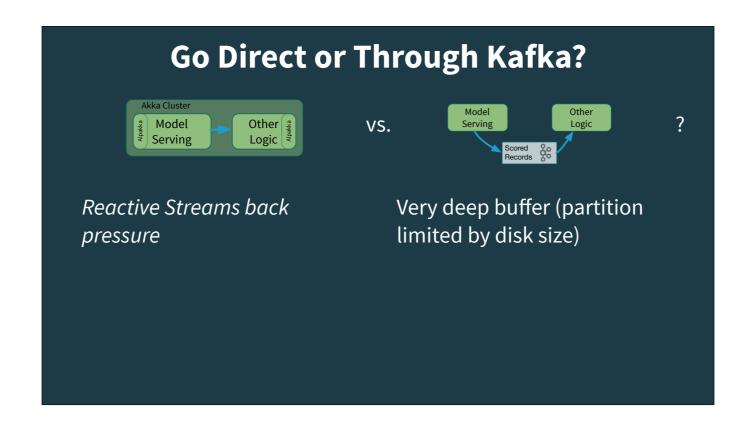
Here's our streaming microservice example adapted for Akka Streams. We'll still use Kafka topics in some places and assume we're using the same implementation for the "Model Training" microservice. Alpakka provides the interface to Kafka, DBs, file systems, etc. We're showing two microservices as before, but this time running in Akka Cluster, with direct messaging between them. We'll explore this a bit more after looking at the example code.

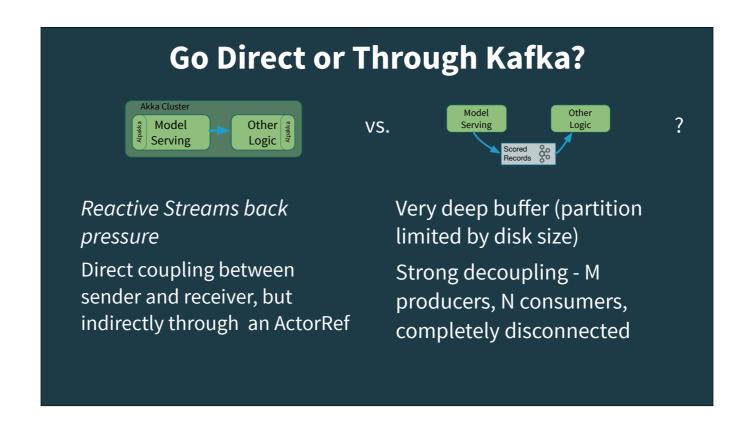


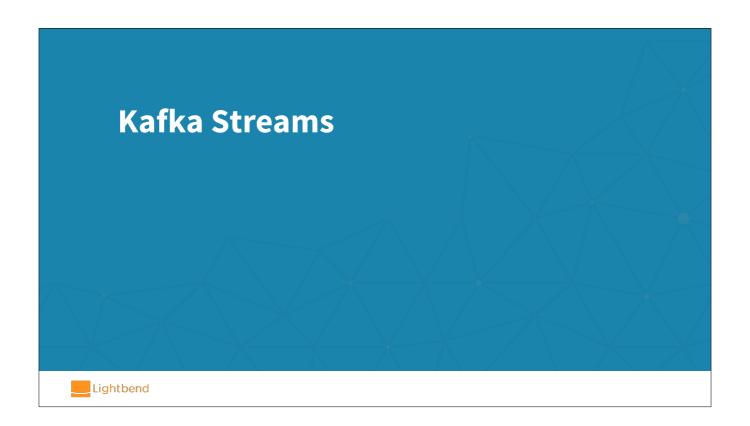
A great article <a href="http://michalplachta.com/2016/01/23/scalability-using-sharding-from-akka-cluster/">http://michalplachta.com/2016/01/23/scalability-using-sharding-from-akka-cluster/</a> goes into a lot of details on both implementation and testing



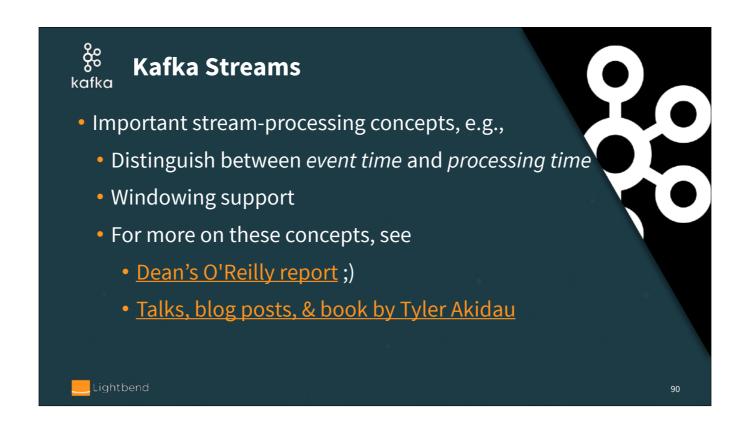








Same sample use case, now with Kafka Streams



There's a maturing body of thought about what streaming semantics should be, too much to discuss here. Dean's book provides the next level of details. See Tyler's work (from the Google Apache Beam team) for deep dives.



There is a duality between streams and tables. Tables are the latest state snapshot, while streams record the history of state evolution. A common way to implement databases is to use an event (or change) log, then update the state from the log.



### **Kafka Streams**

- Low overhead
- Read from and write to Kafka topics, memory
  - Use Alpakka or Kafka Connect for other sources and sinks
- Load balance and scale based on partitioning of topics
- Built-in support for Queryable State



ç

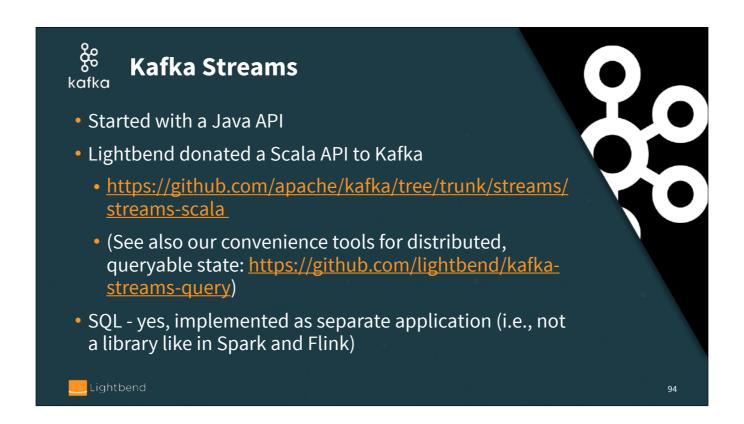


#### **Kafka Streams**

- Two types of APIs:
  - Processor Topology API
  - Lowest-level API
  - Compare to <u>Apache Storm</u>
  - DSL based on collection transformations
  - Compare to Spark, Flink, Scala collections APIs.



Lightbend



The kafka-streams-query uses a KS API to find all the partitions across a cluster for a given topic, query their state, and aggregate the results, behind a web service. Otherwise, you have to query the partitions individually yourself.



#### **Kafka Streams**

- Ideally suited for:
  - ETL -> KStreams
  - State -> KTable
  - Joins, including Stream and Table joins
  - "Effectively once" semantics
- Commercial support from Confluent, Lightbend, Hadoop vendors, and others



Lightbend

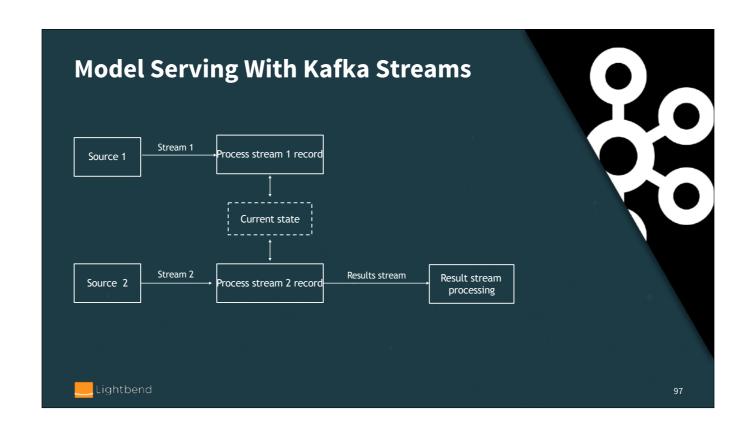


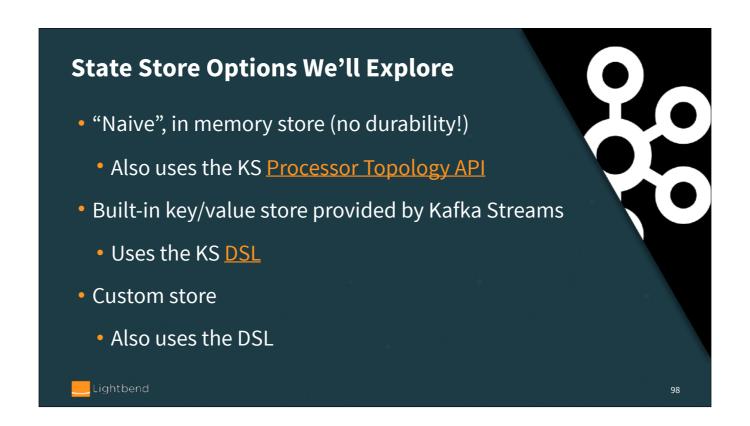
# Kafka Streams: "Effectively Once"

- A distributed transaction model.
- It is *exactly once*, **if** catastrophic failures don't happen...



Lightbend





We provide three example implementations, using three different ways of storing state. "Naive" - because in-memory state is lost if the process crashes; a restart can't pick up where the previous instance left off.

#### **Model Serving With Kafka Streams**

#### **Code time**

- 1. Run the *client* project (if not already running)
- 2. Explore and run kafkaStreamsModelServer project
  - 1. Use the c or custom (or default) commandline argument for the *custom state store*
  - 2. Use the s or standard command-line argument for the KS built-in standard store
  - 3. Use the mor memory command-line argument for the *in-memory store*

Lightbend 4. Use —h or ——help for help

#### **Model Serving With Kafka Streams**

#### **Check Queryable state**

- For in Memory implementation http://localhost:8888/state/value
- For build in Standard Store
   http://localhost:8888/state/instances
   http://localhost:8888/state/value
- For Custom store
   http://localhost:8888/state/instances
   http://localhost:8888/state/value



# Model Serving: Other Production Concerns

Lightbend

# **Additional Concerns for Model Serving**

- Model tracking
- Speculative model execution

Lightbend

#### **Model tracking - Motivation**

- You update your model periodically
- You score a particular record R with model version N
- Later, you audit the data and wonder why R was scored the way it was
- You can't answer the question unless you know which model version was actually used for R

Lightbend

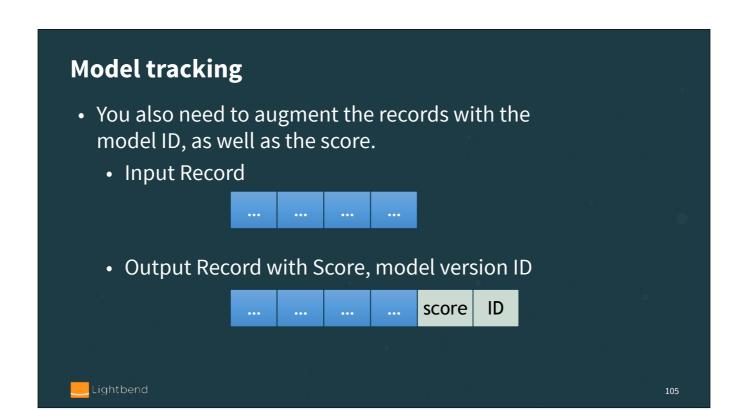
10

"Explainability" is an important problem in Deep Learning; knowing why the model produced the results it produced.

# **Model tracking**

- Need a model repository
- Possible info stored for each model instance:
  - Name
  - Version (or other unique ID)
  - Creation date
  - Quality metric
  - Parameters
  - •

Lightbend



#### **Speculative execution**

According to Wikipedia speculative execution is:

- an **optimization** technique
- The system performs work that may not be needed,
  - before it's known if it will be needed
- If and when it **is** needed, we don't have to wait
  - or results are discarded if not needed

Lightbend

### **Speculative execution**

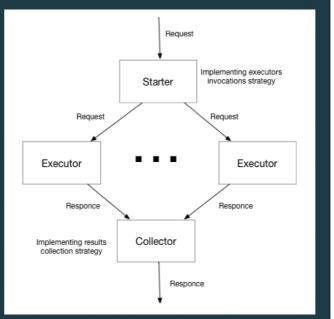
- Provides more **concurrency** if extra **resources** are available.
- Used for:
  - branch prediction in pipelined processors,
  - value prediction for exploiting value locality,
  - prefetching **memory** and **files**,
  - etc.

Why not use it with machine learning??

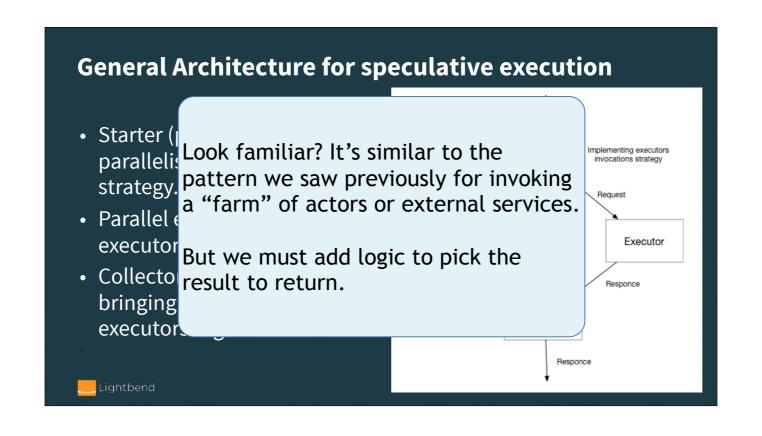
Lightbend

### **General Architecture for speculative execution**

- Starter (proxy) controlling parallelism and invocation strategy.
- Parallel execution by identical executors
- Collector responsible for bringing results from multiple executors together







### Applicability for model serving (1/3)

- Guarantee execution time, i.e., meet our latency SLA!
  - Several models:
    - A smart model, but takes time *T1* for a given record
    - A "less smart", but fast model with a fixed upper-limit on execution time, *T2* << *T1*
  - If timeout T occurs, where *T1 > T > T2*, return the less smart result
    - Do you understand why T1 > T > T2 is required?

Lightbend

11

Because the timeout T has to be long enough that the fast model has time to finish, so T must be longer than T2, and this is only useful if T1 is often longer than T, our latency window.

This technique is "speculative", because we try both models, taking a compromise result if the good result takes too long to return.

# Applicability for model serving (2/3)

- Consensus based model serving
  - If we have 3 or more models, score with all of them and return the majority result

Lightbend

1

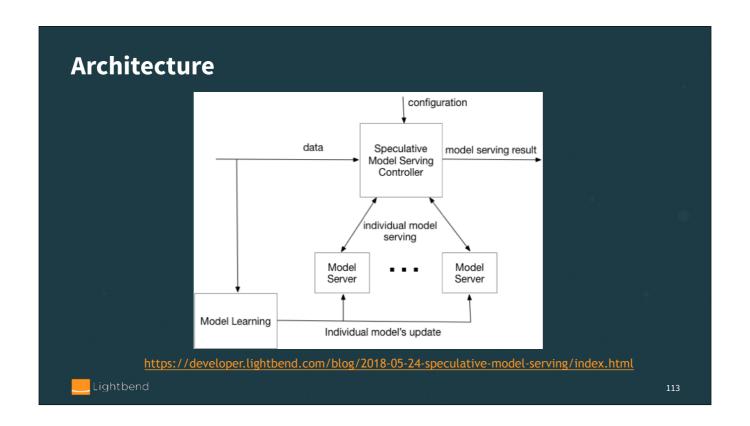
# Applicability for model serving (3/3)

- Quality based model serving.
  - If we have a quality metric, pick the result with the best result.

Of course, you can combine these techniques.

Lightbend

11



This blog post provides more information on this technique.

# One Design Using Actors (root guardian) /user (User guardian) /user/modelservicemanager /user/modelservicemanager/modelsmanager /user/modelservicemanager/modelsmanager/modelserver n Lightbend



For a full-day tutorial

### **Exercises**

- Exercises are embedded in the source code
  - Search for // Exercise in the code
  - Additional exercises are described in the README.
- In most cases, an exercise is repeated in each variant of the Akka Streams and Kafka Streams implementations. Pick the one you find most interesting...

### **Exercises**

- See the end of the README for a list of resources that will be useful as you modify the code, e.g,
  - <u>Scala Library (Scaladocs)</u>
  - Akka Streams:
    - Scala: <u>Reference</u>, <u>Scaladocs</u>
    - Java: <u>Reference</u>, <u>Javadocs</u>
  - Kafka Streams:
    - <u>Scaladocs</u>, <u>Javadocs</u>

# **Suggestions (1/4)**

- ModelServerProcessor.java/scala, around line 55 in both
- StandardStoreStreamBuilder.java/scala: createStreams()
- CustomStoreStreamBuilder.java/scala: createStreams()
- MemoryStoreStreamBuilder.java/scala: createStreams()
  - Rather than just print results, write them to a new Kafka topic

### **Suggestions (2/4)**

- ModelServingManager.java/scala: getModelServer()
- DataProcessor.java/scala (all of the variants!)
  - Implement either speculative execution or a worker "farm" for parallelism, rather than a single model instance.
  - We've only given you one model at a time to work with. Use a hack with multiple copies of the model, but add noise to the result to simulate different model results

This exercise is described in several classes for both Akka Streams and Kafka Streams.

### **Suggestions (3/4)**

- AkkaModelServer.java/scala: main()
- CustomStoreStreamBuilder.java/scala, DataProcessor.java/scala, ModelProcessor.java/scala (all of the variants!)
  - They provide implementations of error handling, when input wine records fail to parse. Improve the implementations to write to a special Kafka topic instead.
  - Extend the idioms to handle model errors.

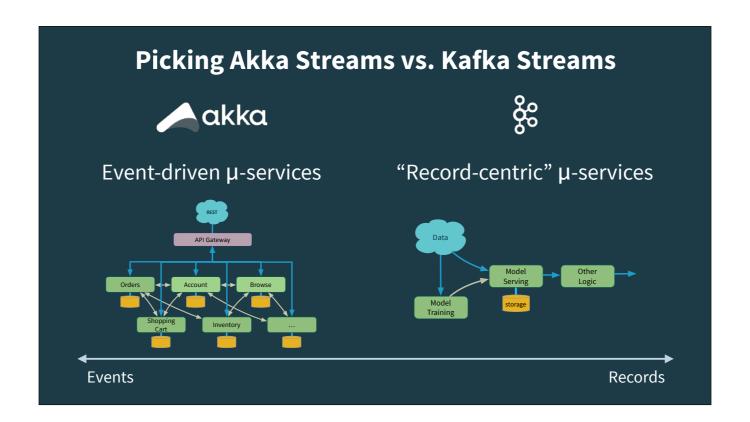
This exercise is described in several classes for both Akka Streams and Kafka Streams.

# **Suggestions (3/4)**

- model project
  - It hard-codes information about the example wine records. Can you make it more generic? See the suggestions in PMMLModel.java/scala or TensorFlowModel.java/scala
- Can you make other areas where WineRecord is assumed more generic?

This exercise is described in several classes for both Akka Streams and Kafka Streams.





Akka Streams is a great choice if you are building full-spectrum microservices and you need lots of flexibility in your app architectures, connecting to different kinds of data sources and sinks, etc.

Kafka Streams is a great choice if your use cases fit nicely in it's "sweet spot", you want SQL access, and you don't need to full flexibility of something like Akka. Of course, you can use both! They are "just libraries".

# Next Steps (1/2)

- Study the different model serving techniques
- Study the "model" subproject
- Look at how the following are implemented
  - 1. queryable state
  - 2. embedded web servers
  - 3. use of Akka Persistence
  - 4. model serialization

# Next Steps (2/2)

- Do the exercises
  - Search for // Exercise in the code
  - See the README
- Ask us for help on anything...
- Visit <u>lightbend.com/fast-data-platform</u>
- Profit!!



Thank you! Check out our Fast Data Platform for commercial options for building and running microservices with Kafka, Akka Streams, and Kafka Streams.