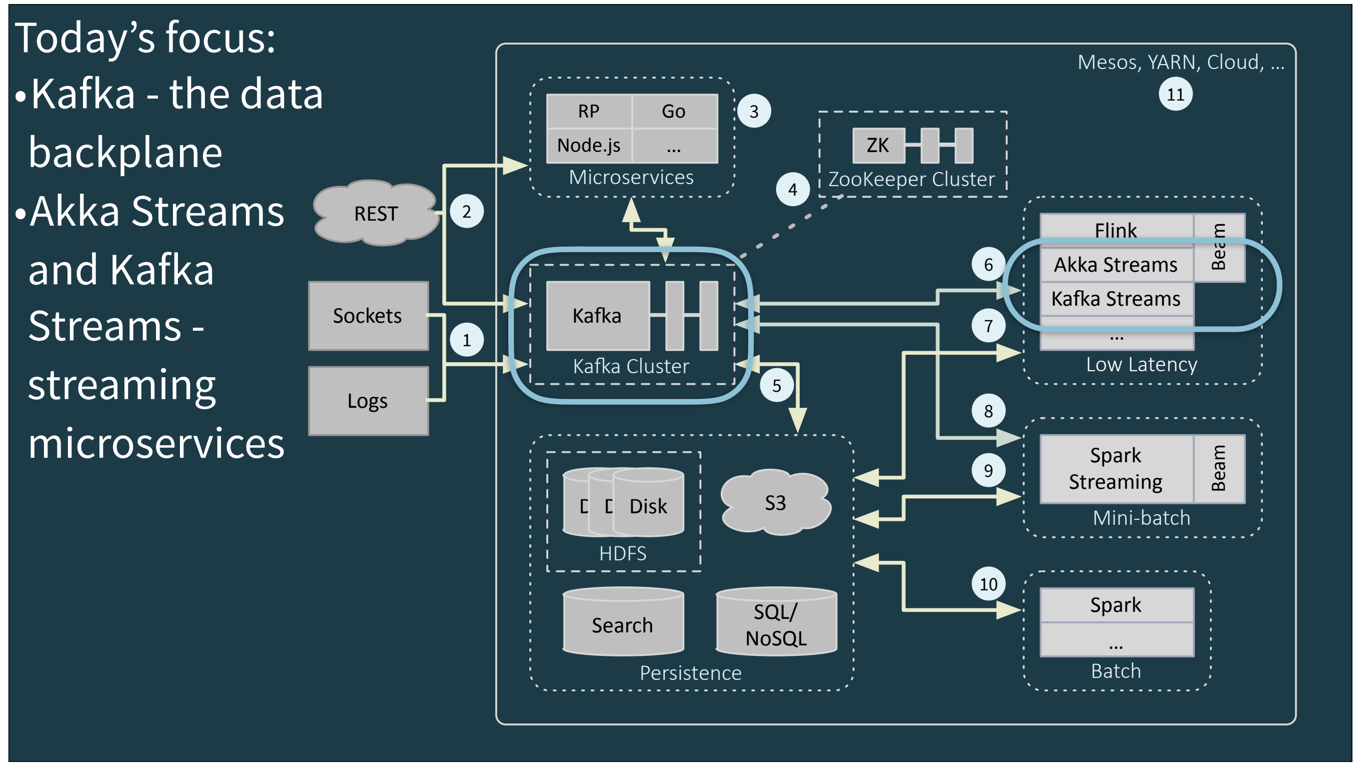
Building Kafka-based Microservices with Akka Streams and Kafka Streams

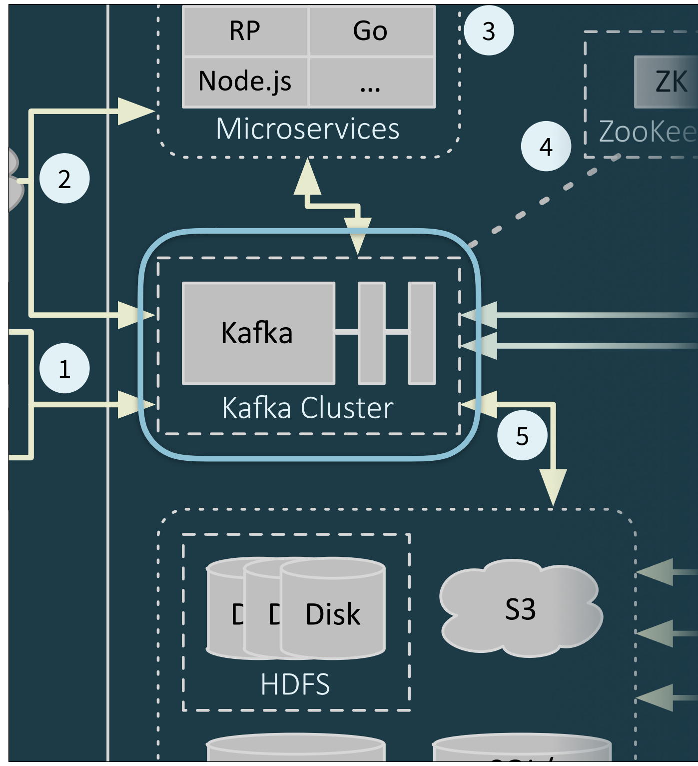
Boris Lublinsky and Dean Wampler, Lightbend

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

- 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, https://info.lightbend.com/webinar-moving-from-big-data-to-fast-data-heres-how-to-pick-the-right-streaming-engine-recording.html) 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



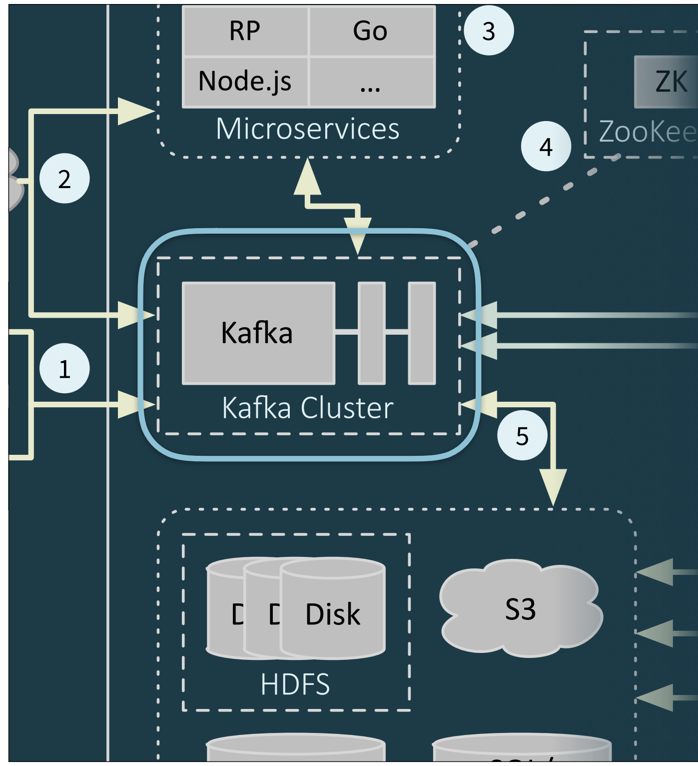
Kafka is the data backplane for high-volume data streams, which are organized by topics. Kafka has high scalability and resiliency, so it's an excellent integration tool between data producers and consumers.



Kafka:

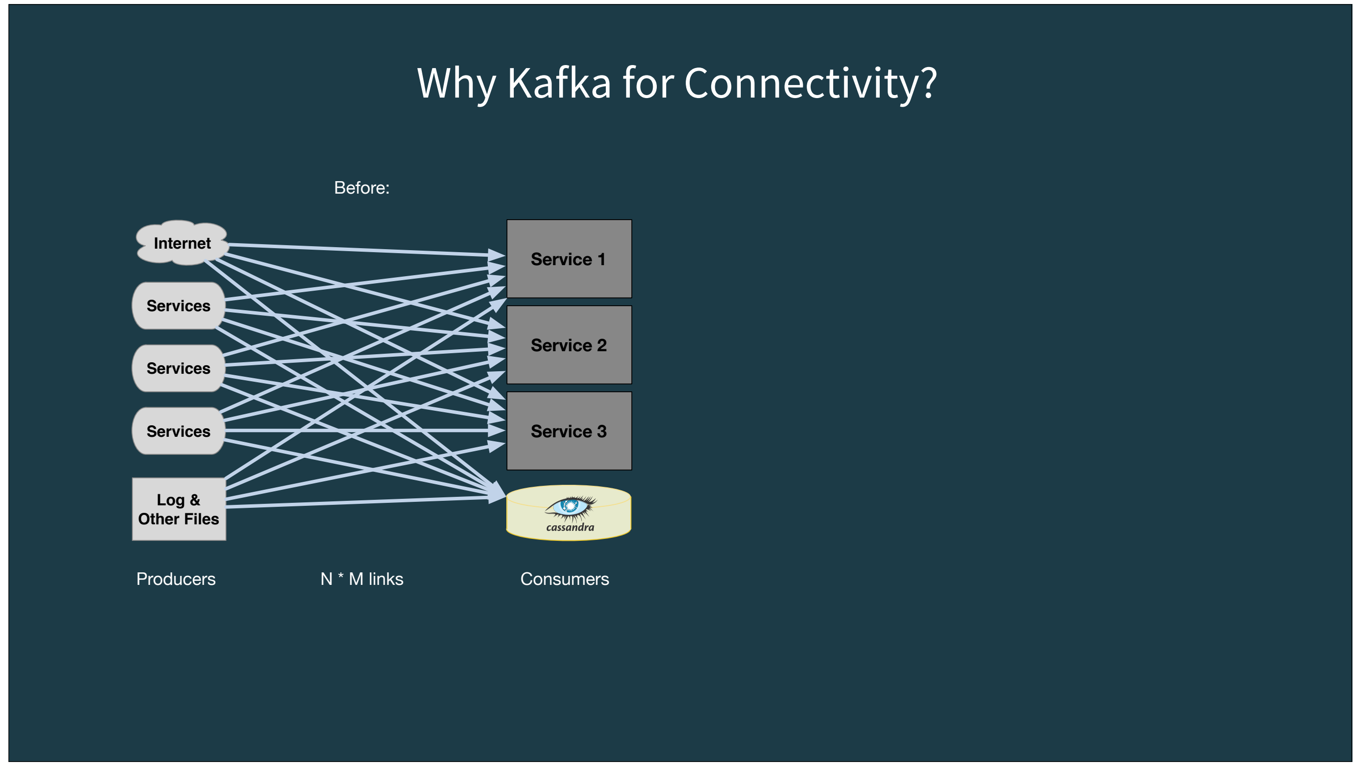
- Run as a cluster on 1+ servers
- Stores *logs* of records in topics.
- Each record: key, value, and timestamp.
- Topics can be partitioned

Kafka concepts.
Note: it’s not a queue system, (because readers don’t consume the records). Instead, it’s log oriented, where each consumer can read the whole log. Kafka manages record lifecycles.

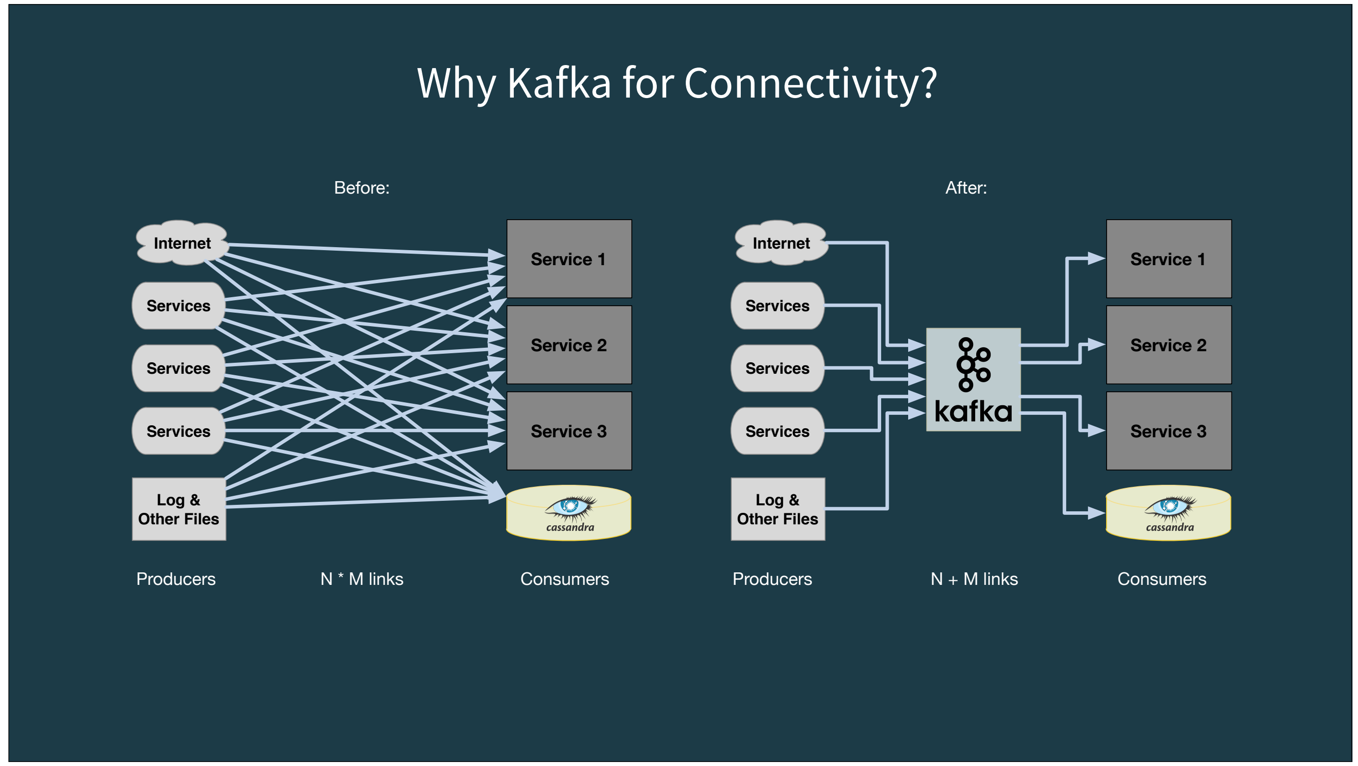


Kafka:

- Each topic can have 1 or more
  - Producers
  - Consumers



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

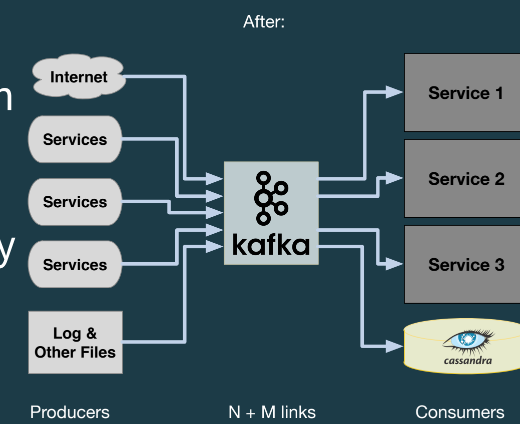


Kafka can simplify the situation by providing a single backbone which is used by all services (there are of coarse topics, but they are more logical then physical connections). Additionally Kafka persistence provides robustness when a service crashes (data is captured safely, waiting for the service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

Why Kafka for Connectivity?

Kafka:

- Simplify dependencies between services
    - Improved data consistency
- Minimize data transmissions
- Reduce data loss when a service crashes

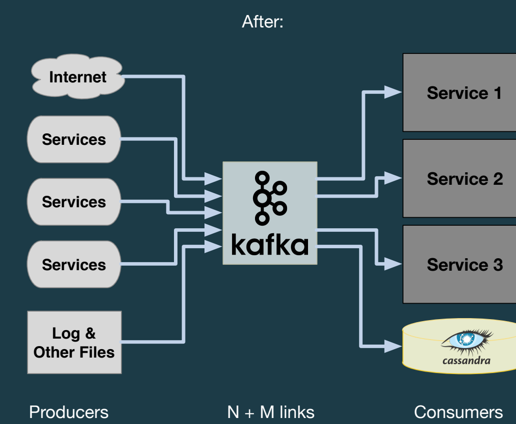


Kafka can significantly improve decoupling (no service specific endpoints, temporal decoupling), It minimize the amount of data send over network, each producer writes data to Kafka, instead of writing it to multiple consumers. This also improves data consistency - the same data is consumed by all consumers. 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.

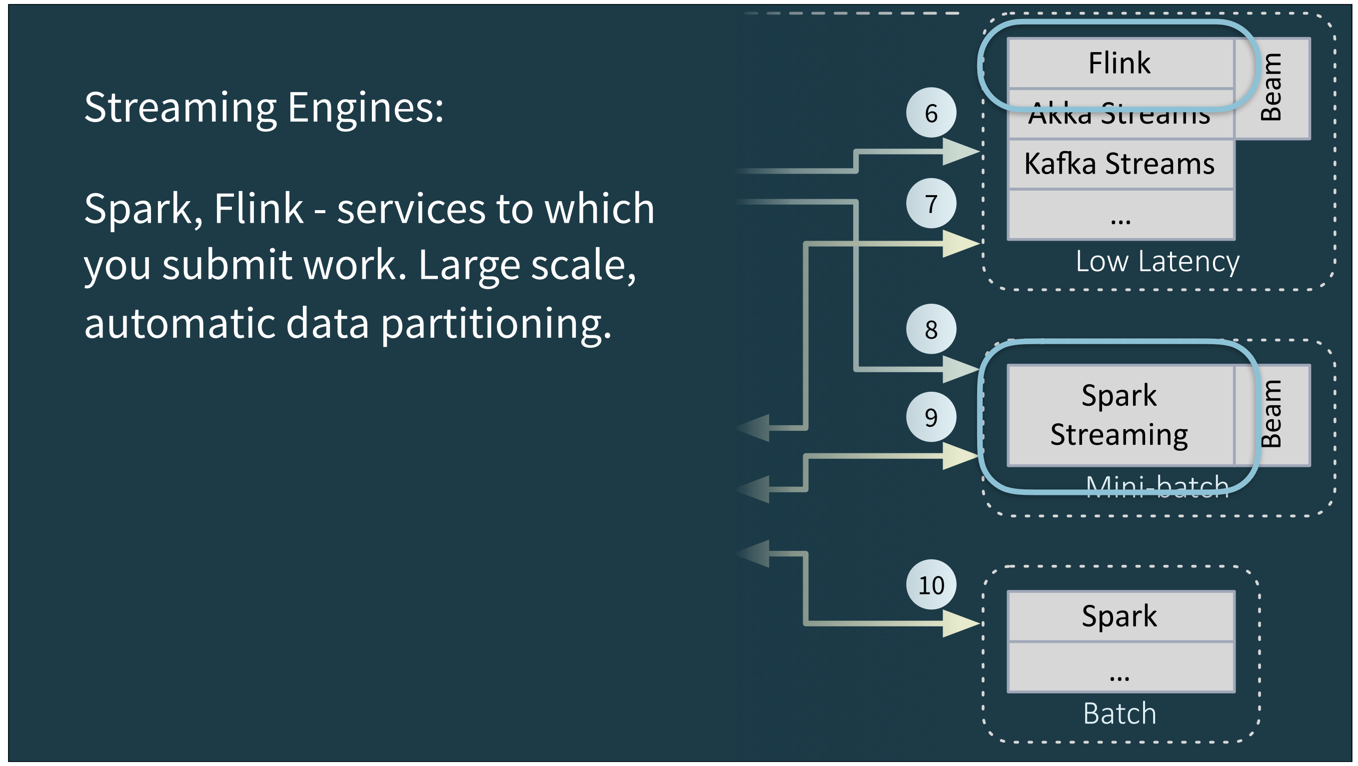
Why Kafka for Connectivity?

Kafka:

- M producers, N consumers
    - Improved extensibility
- Simplicity of one “API” for communication



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.

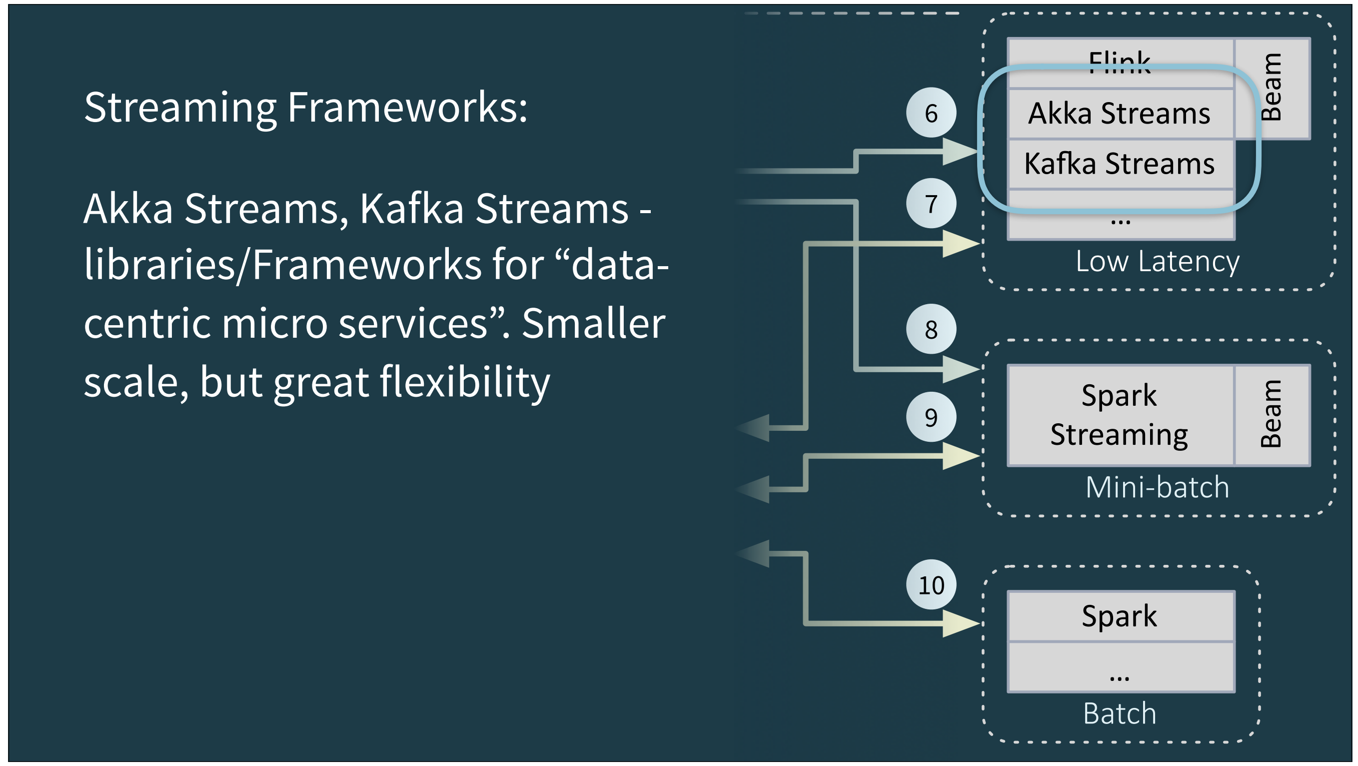


They support highly scalable jobs, where they manage all the issues of scheduling processes, etc. You submit jobs to run to these running daemons. They handle scalability, failover, load balancing, etc. for you.

Streaming Engines:

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

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!

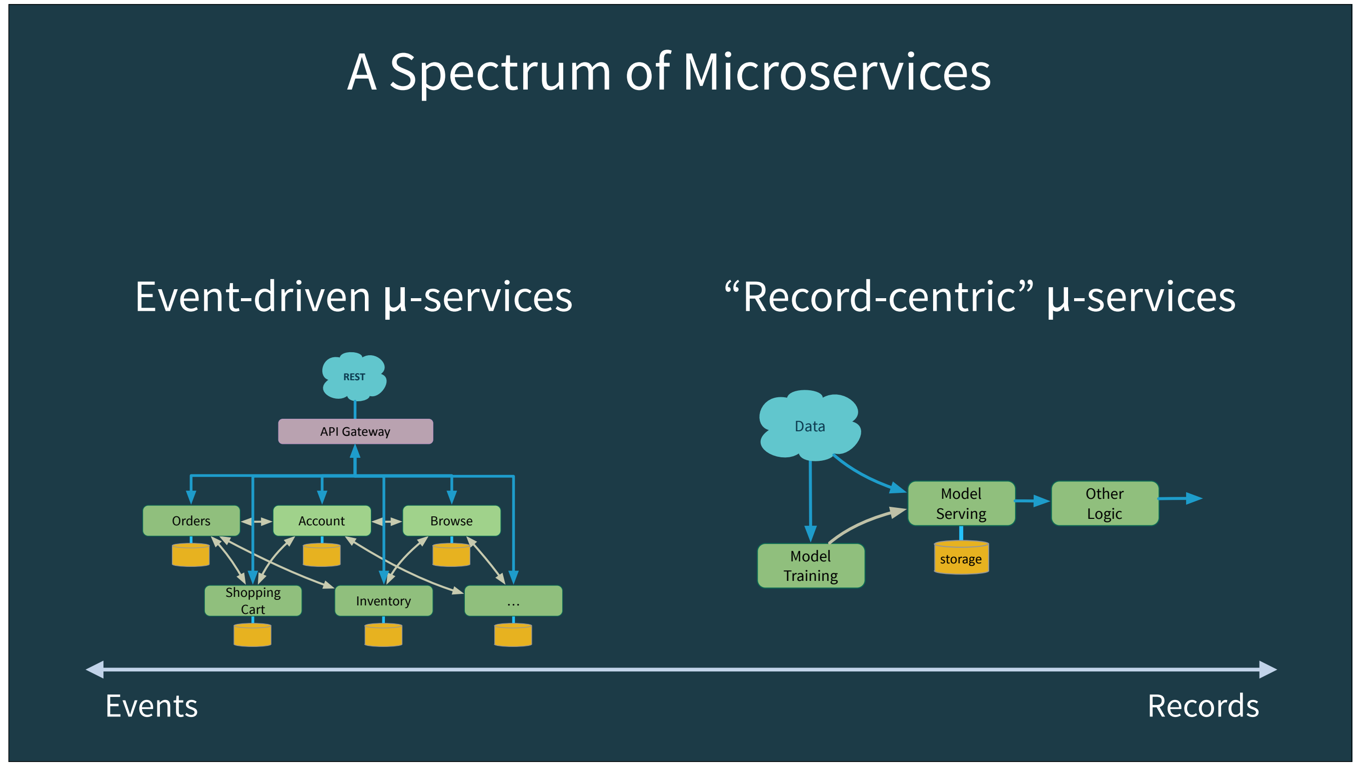


Much more flexible deployment and configuration options, compared to Spark and Flink, but more effort is required by you to run them. They are “just libraries”, so there is a lot of flexibility and interoperation capabilities.

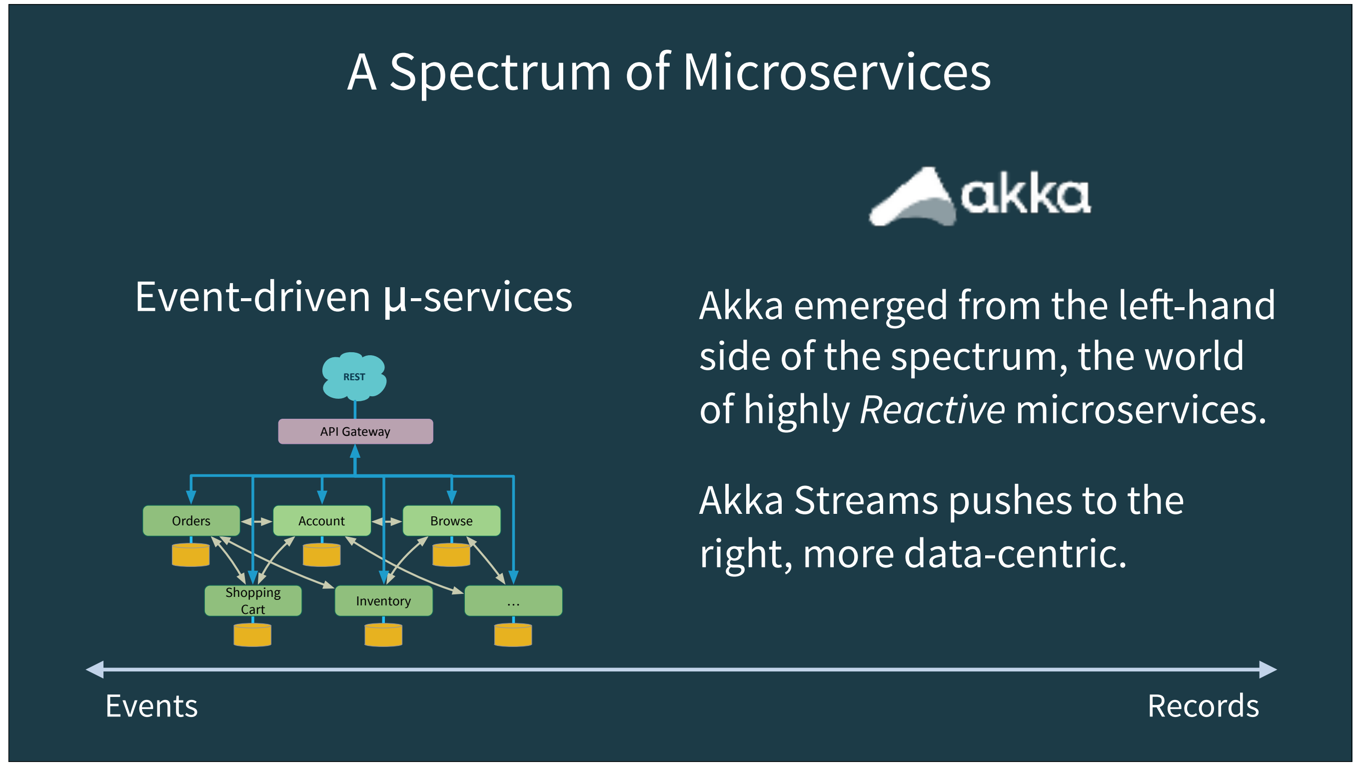
Microservice All the Things!

Lightbend

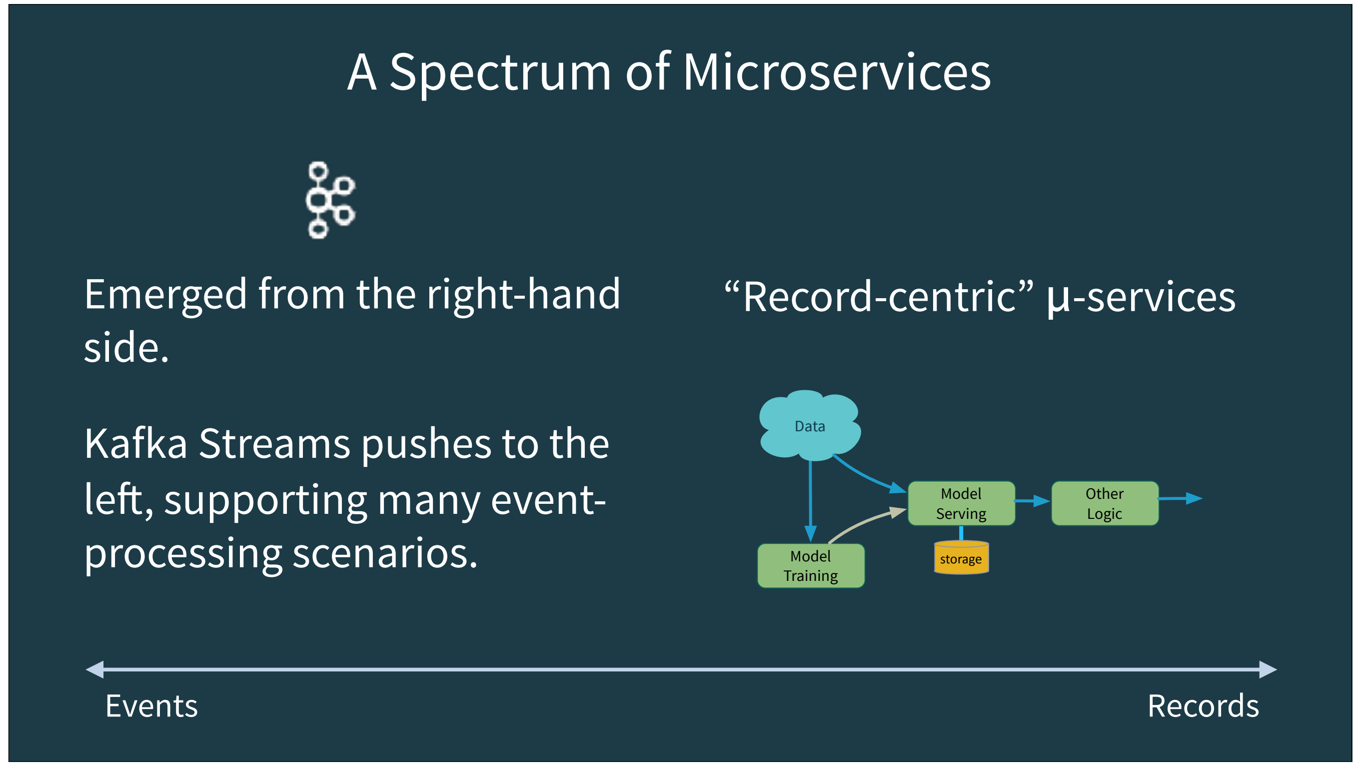
https://twitter.com/shanselman/status/967703711492423682



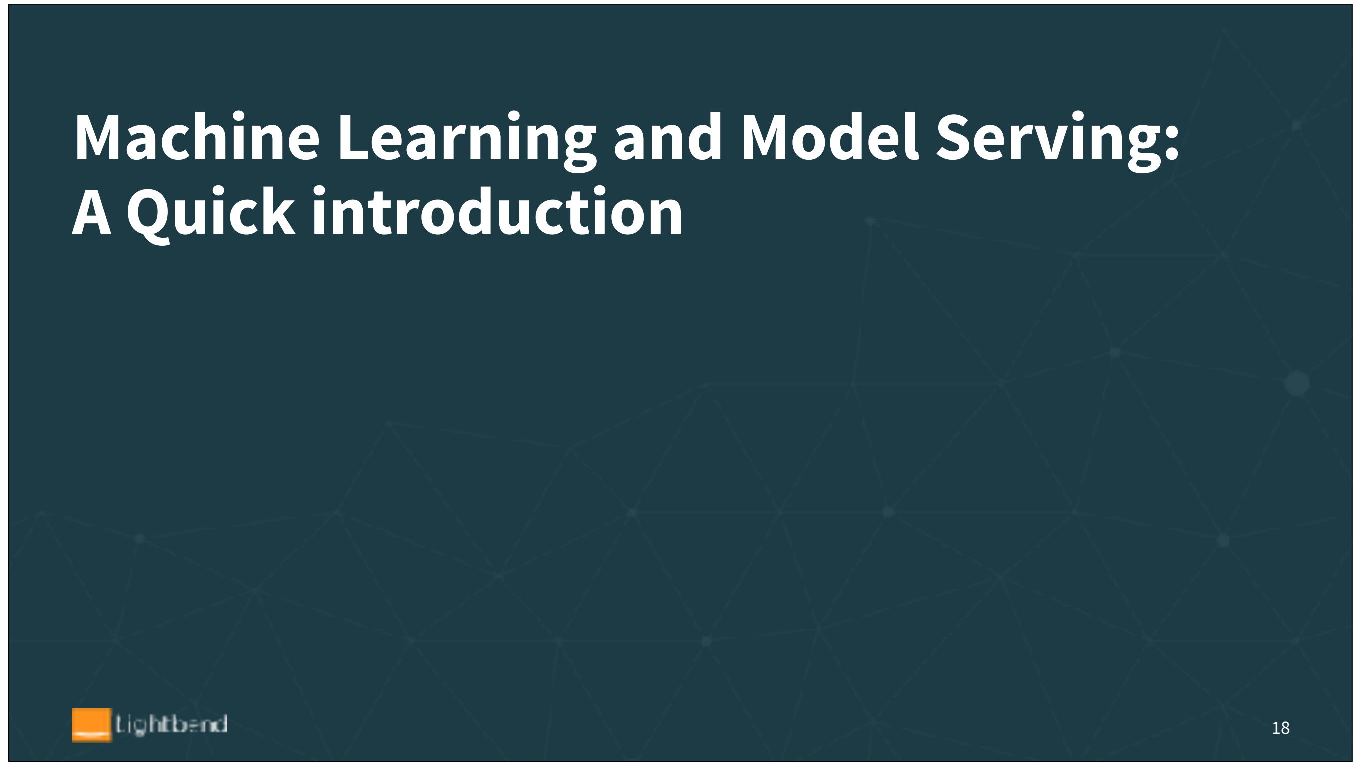
By event-driven microservices, I mean that each individual datum is treated as a specific event that triggers some activity, like steps in a shopping session. Each event requires individual handling, routing, responses, etc. REST, CQRS, and Event Sourcing are ideal for this.
Records are uniform (for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



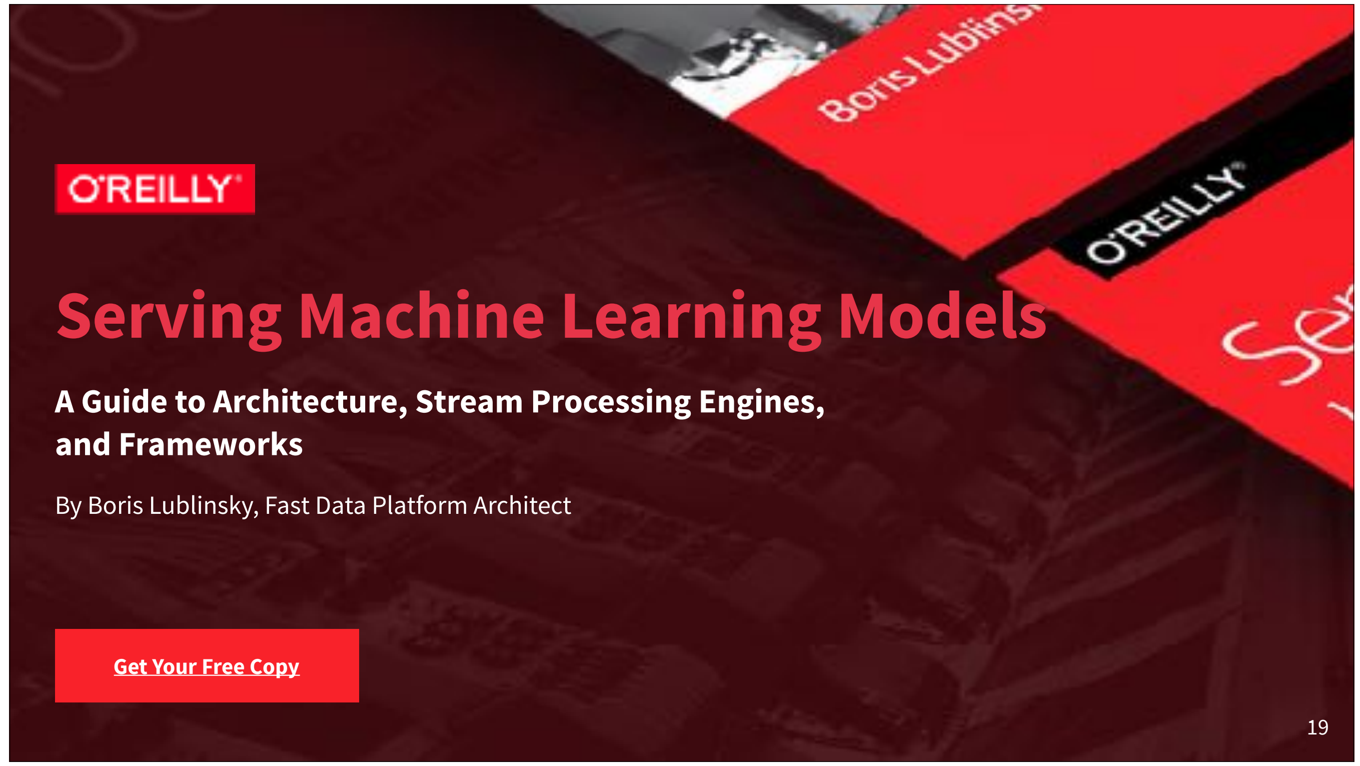
I think it's useful to reflect on the history of these toolkits, because their capabilities reflect their histories. Akka Actors emerged in the world of building *Reactive* microservices, those requiring high resiliency, scalability, responsiveness, CEP, and must be event driven. Akka is extremely lightweight 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.



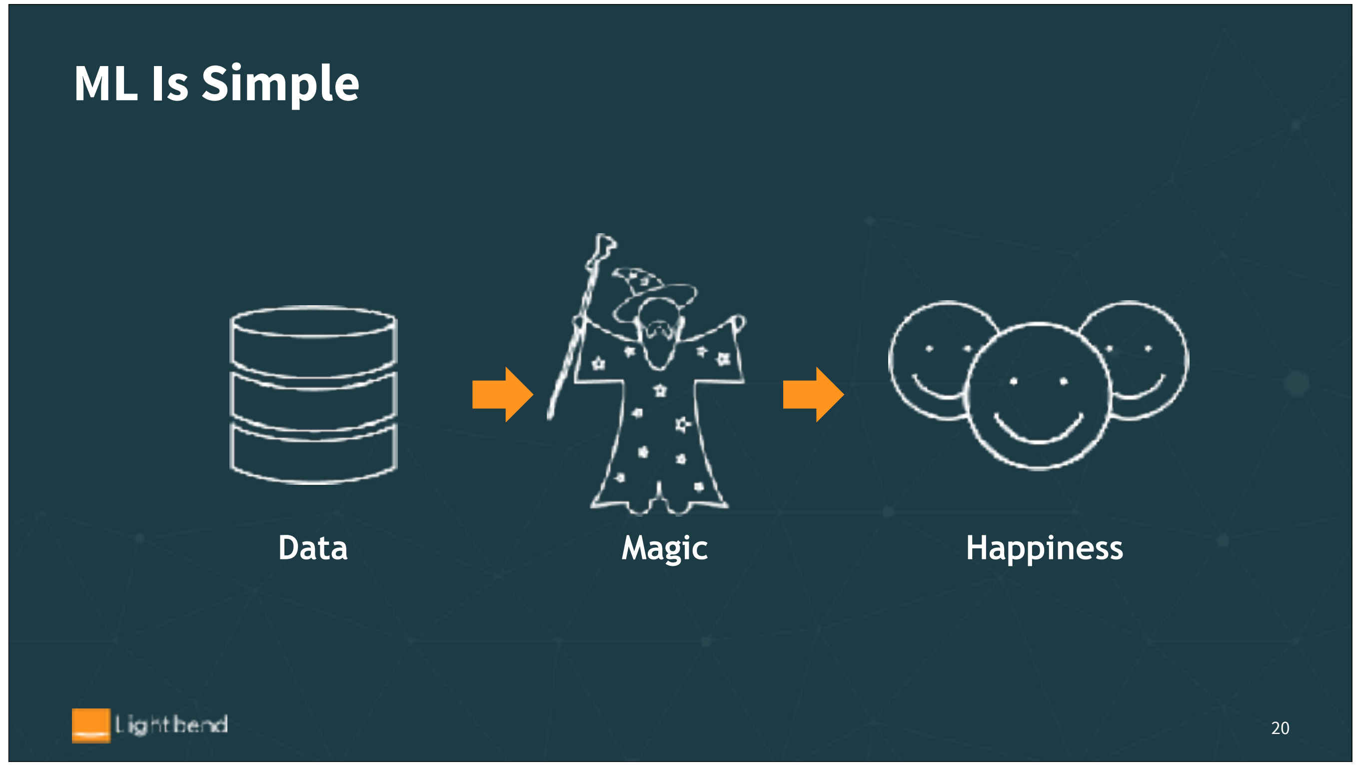
Kafka reflects the heritage of moving and managing streams of data, first at LinkedIn. But from the beginning it has been used for event-driven microservices, where the “stream” contained events, rather than records. Kafka Streams fits squarely in the record-processing world, where you define dataflows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



We’ll return to more details about AS and KS as we get into implementation details.



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



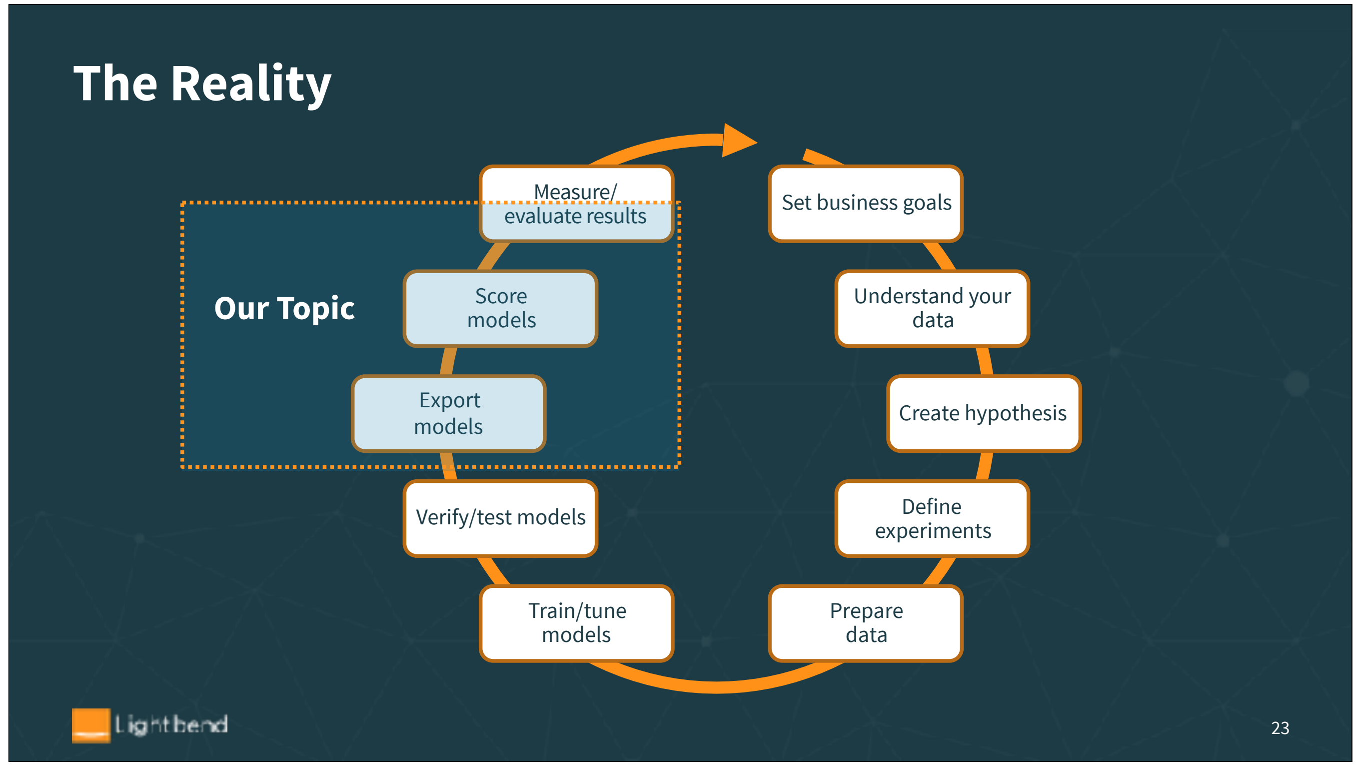
Get a lot of data
Sprinkle some magic
And be happy with results

Maybe Not

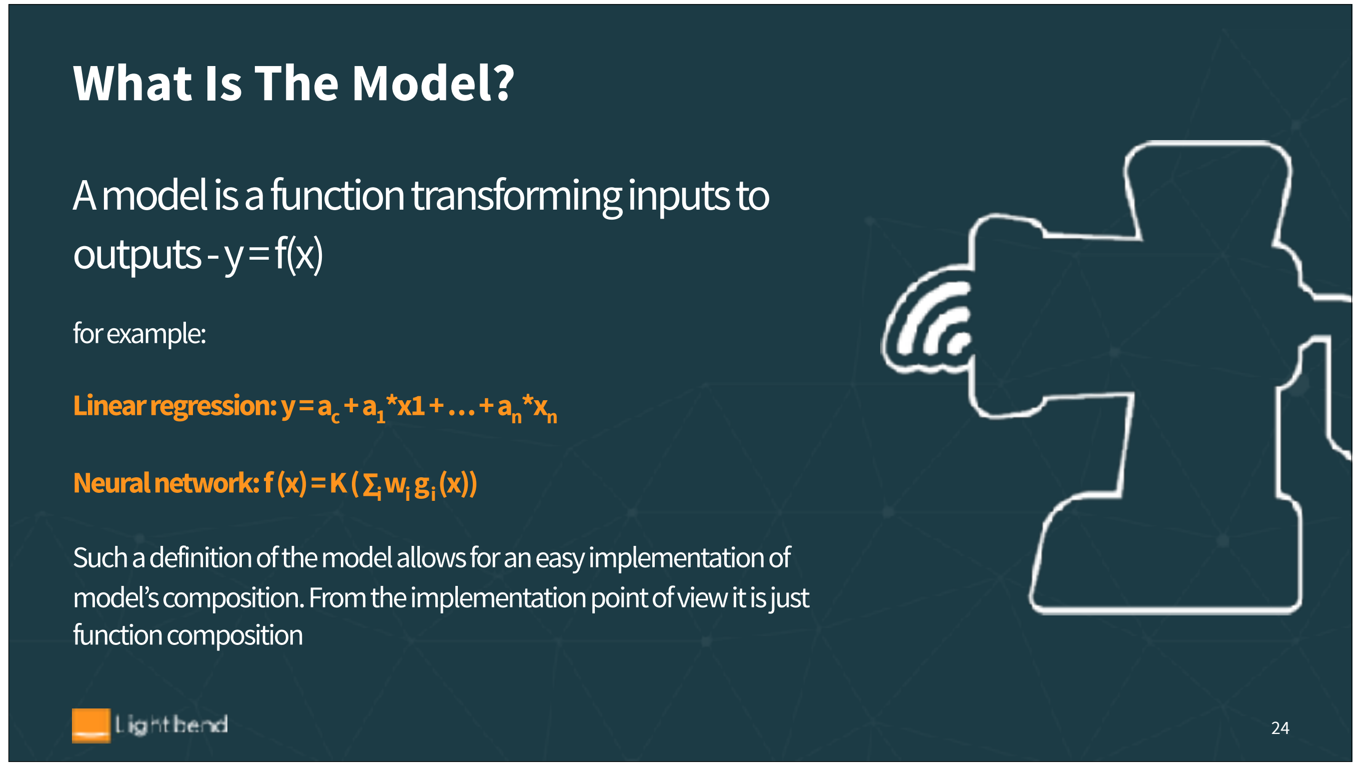
Not only the climb is steep, but you are not sure which peak to climb
Court of the Patriarchs at Zion National park

Even If There Are Instructions

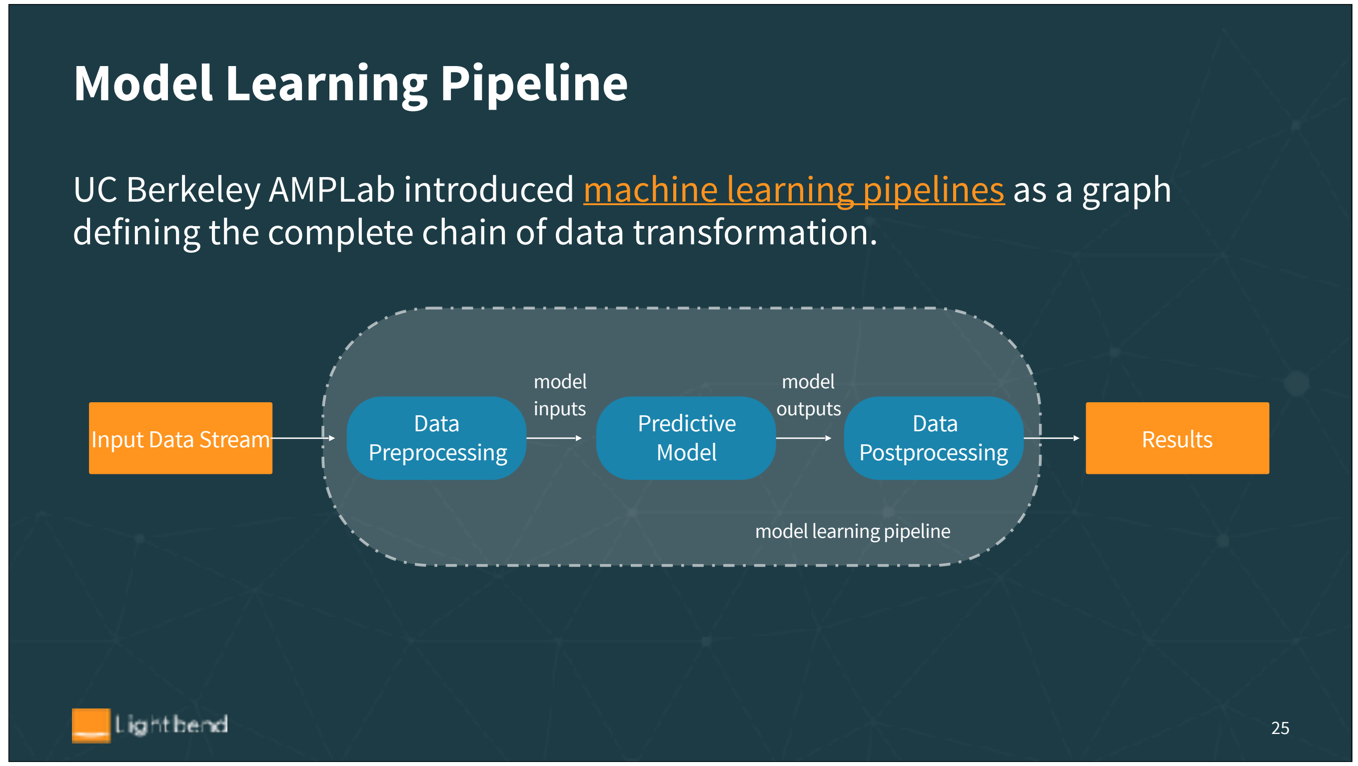
Not only the climb is steep, but you are not sure which peak to climb
Court of the Patriarchs at Zion National park



- But what does a company in the, say, *Aware* stage look like, vs a company in the *Expand* stage?
- Some real-world examples can help drive that understanding.
-



- But what does a company in the, say, *Aware* stage look like, vs a company in the *Expand* stage?
- Some real-world examples can help drive that understanding.
-



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.
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 is code
- This code has to be saved and then somehow imported into model serving

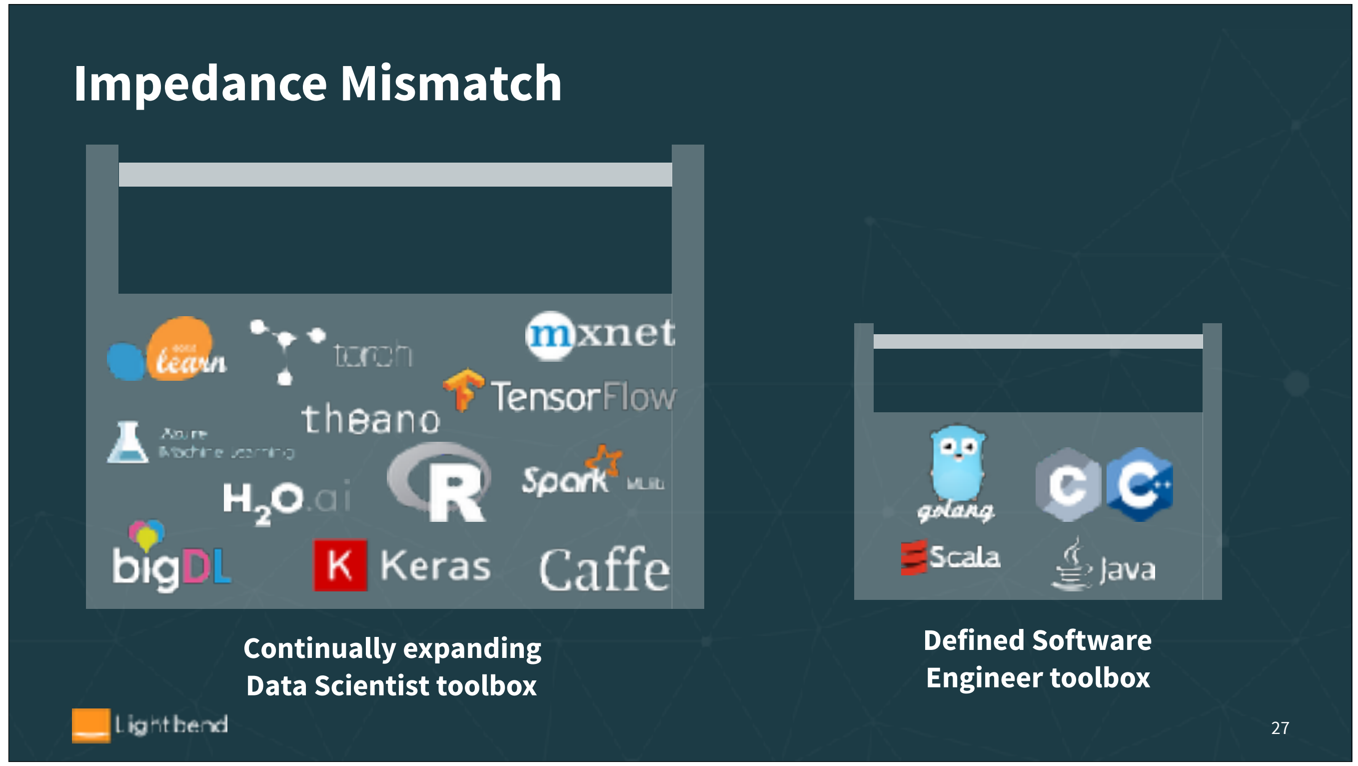
Why is this problematic?

Lightbend

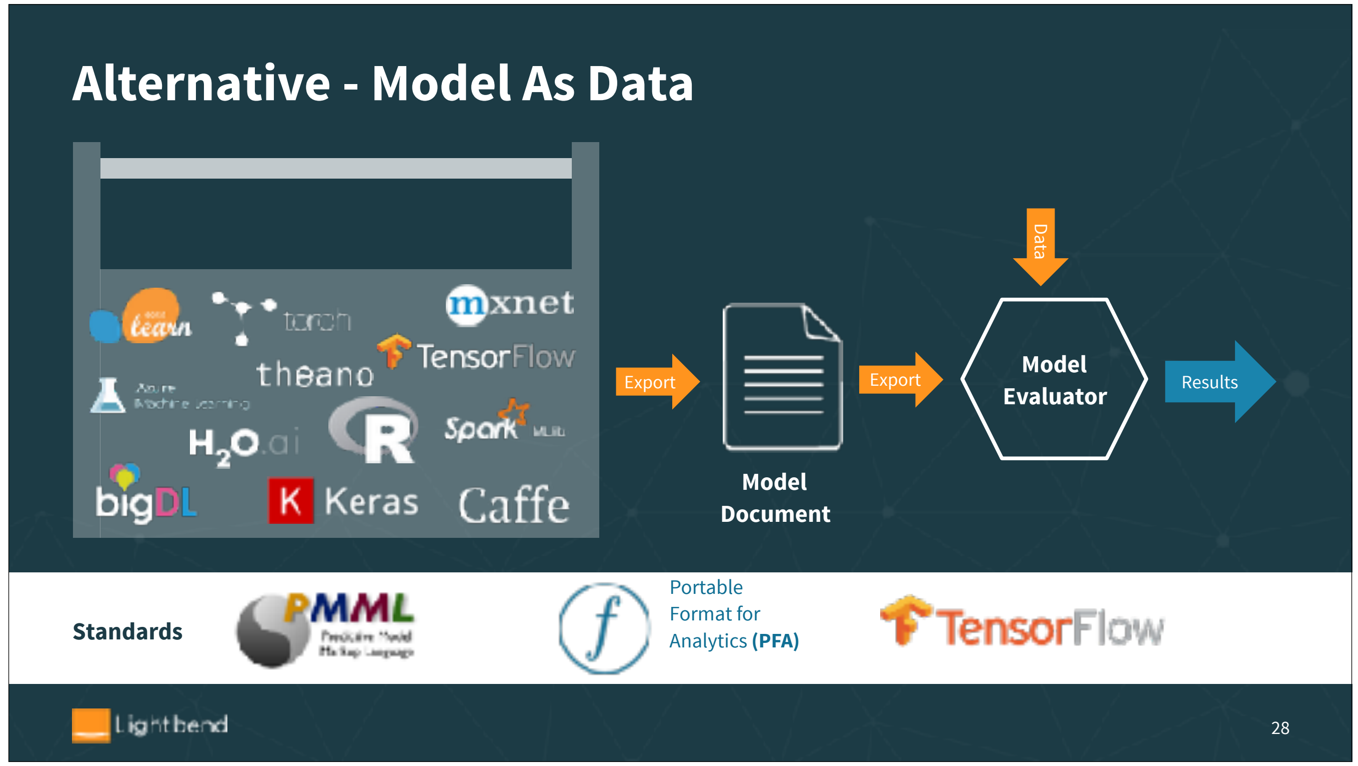
This provides guidance only and should be used as a suggestion. Choose the best fit for each answer. Return to this deck once the questionnaire is complete and scored.

Link: https://goo.gl/forms/cay1pMtxIQh83cSD3

NOTE: Score answers on a 1 to 4 point scale for each answer from top to bottom -



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.



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.

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

Exporting Model As Data With PMML

There are already a lot of export options

- Spark MLlib: https://github.com/jpmml/jpmml-sparkml
- scikit-learn: https://github.com/jpmml/jpmml-sklearn
- R: https://github.com/jpmml/jpmml-r
- TensorFlow: https://github.com/jpmml/jpmml-tensorflow

Evaluating PMML Model

There are also a couple PMML evaluators

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

---

python https://github.com/opendatagroup/augustus

Exporting Model As Data With Tensorflow

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

Evaluating Tensorflow Model

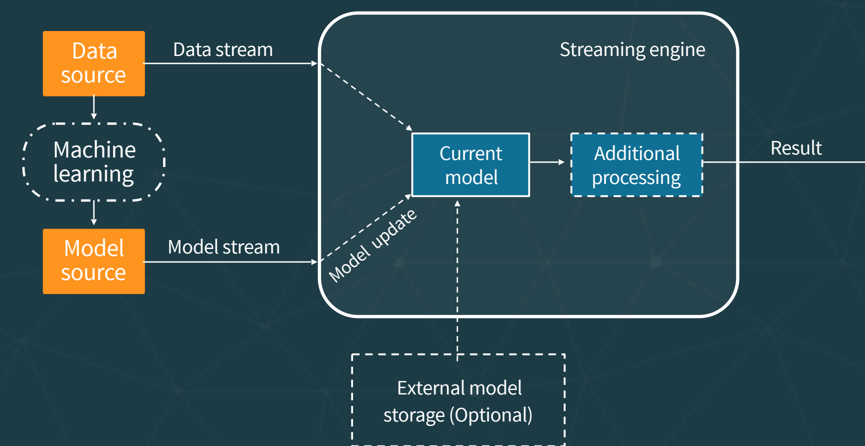
- Tensorflow is implemented in C++ with Python interface.
- In order to simplify Tensorflow usage from Java, in 2017 Google introduced Tensorflow Java APIs.
- Tensorflow Java APIs supports import of the exported model and allows to use them for scoring.

Additional Considerations – Model Lifecycle

- Models tend to change
- Update frequencies vary greatly – from hourly to quarterly/yearly
- Model version tracking
- Model release practices
- Model update process

The Solution

A streaming system allowing to update models without interruption of execution (dynamically controlled stream).



The majority of machine learning implementations are based on running model serving as a Rest service, which might not be appropriate for the high volume data processing or usage of the streaming system, which requires re coding/starting systems for model update, 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
     TENSORFLOW  = 0;
     TENSORFLOWSAVED = 2;
     PMML = 2;
  };
  ModelType modeltype = 4;
  oneof MessageContent {
    // Byte array containing the model
    bytes data = 5;
    string location = 6;
  }
}
```


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.

Model Representation (Scala)

```
trait Model {
  def score(input : AnyVal) : AnyVal
  def cleanup() : Unit
  def toBytes() : Array[Byte]
  def getType : Long
}

def ModelFactoryl {
  def create(input : ModelDescriptor) : Model
  def restore(bytes : Array[Byte]) : Model
}
```


Corresponding Scala code

Additional Considerations: Monitoring

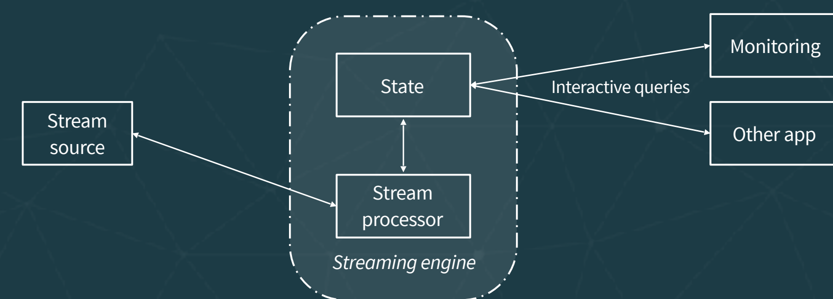
Model monitoring should provide information about usage, behavior, performance and lifecycle of the deployed models

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

Queryable State

Queryable state: ad hoc query of the state in the stream. Different than the normal data flow.

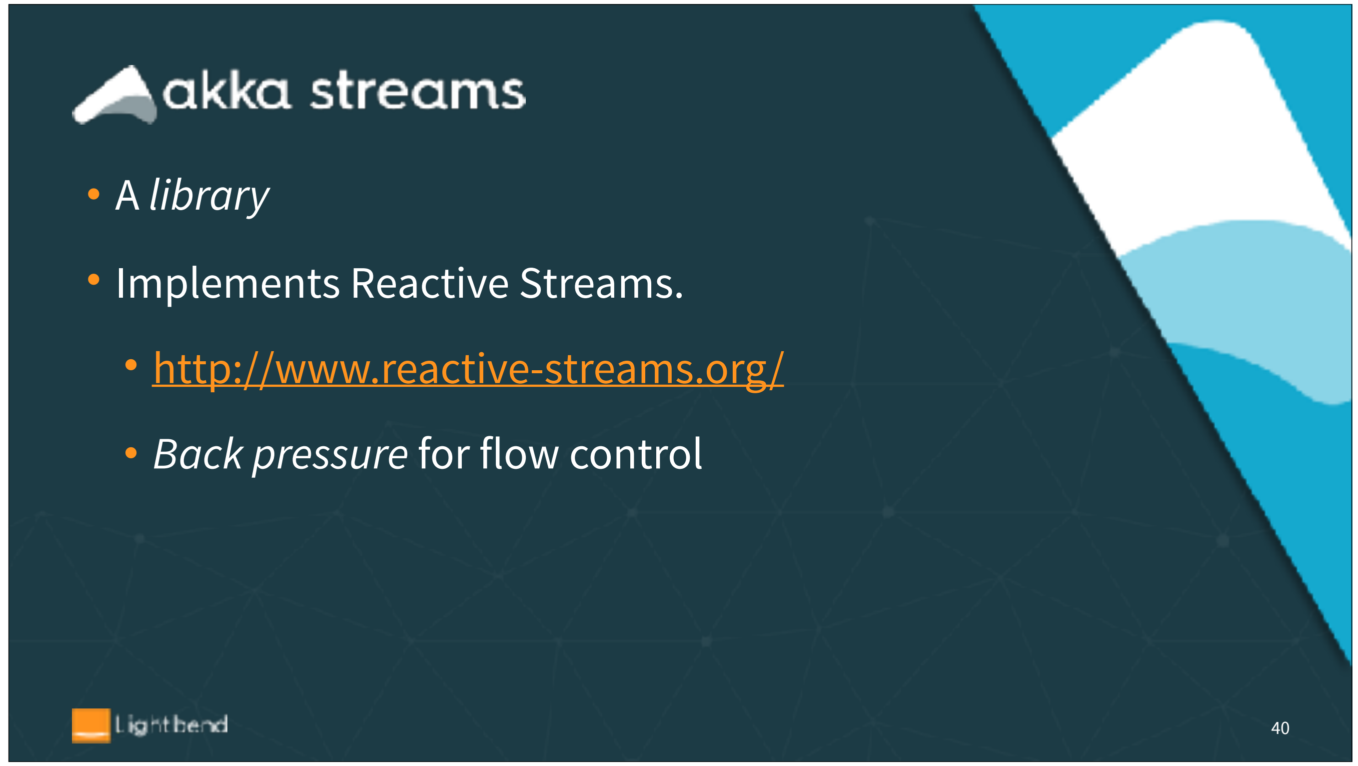
Treats the stream processing layer as a lightweight embedded *database. Directly query the current state* of a stream processing application. No need to materialize that state to a database, etc. first.



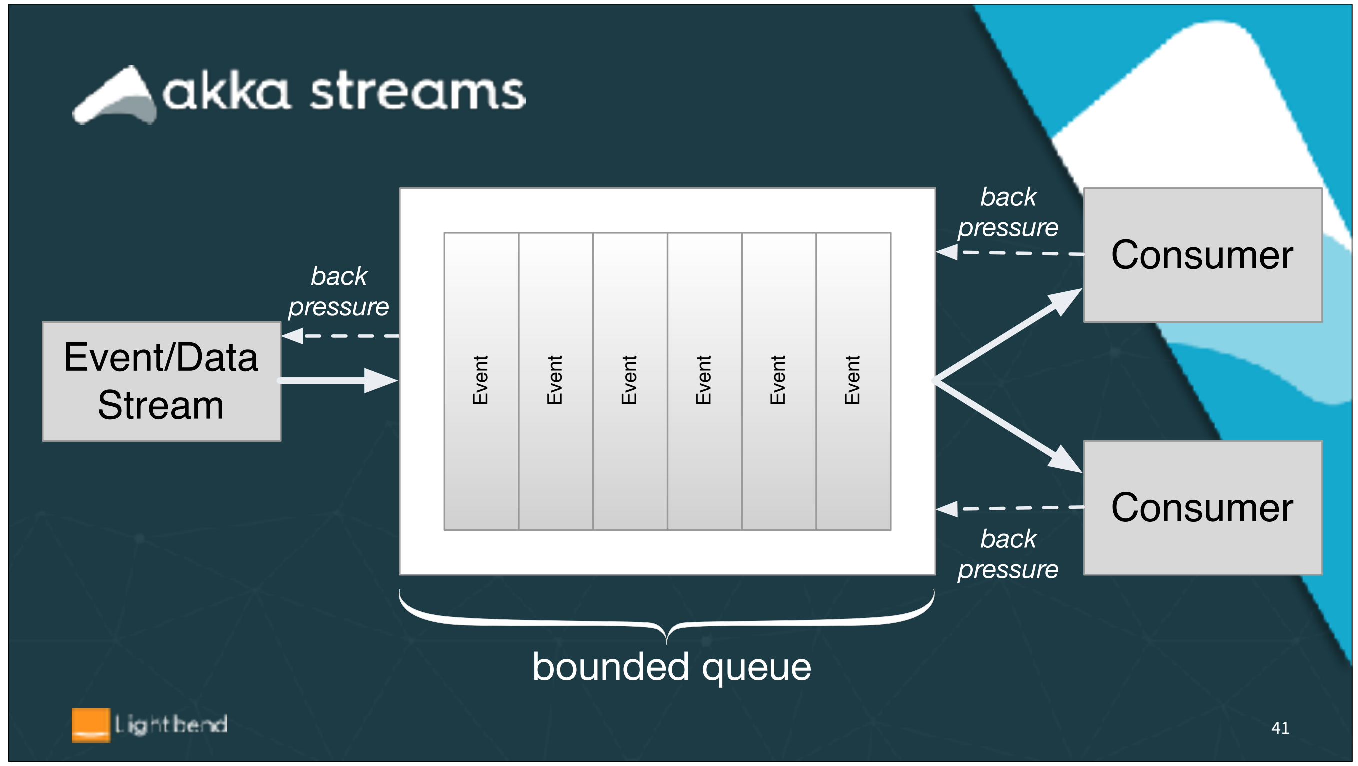
Lightbend

akka streams

- We'll work with Akka Streams examples first



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 pull when flow control is needed.

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

```scala
import akka.stream._
import akka.stream.scaladsl._
import akka.{ NotUsed, Done }
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 akkaStreamsCustomStage/simple-akka-streams-example.sc

```scala
import akka.stream._
import akka.stream.scaladsl._
import akka.{ NotUsed, Done }
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))
```

Imports!

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

```scala
import akka.stream._
import akka.stream.scaladsl._
import akka.{ NotUsed, Done }
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))
```

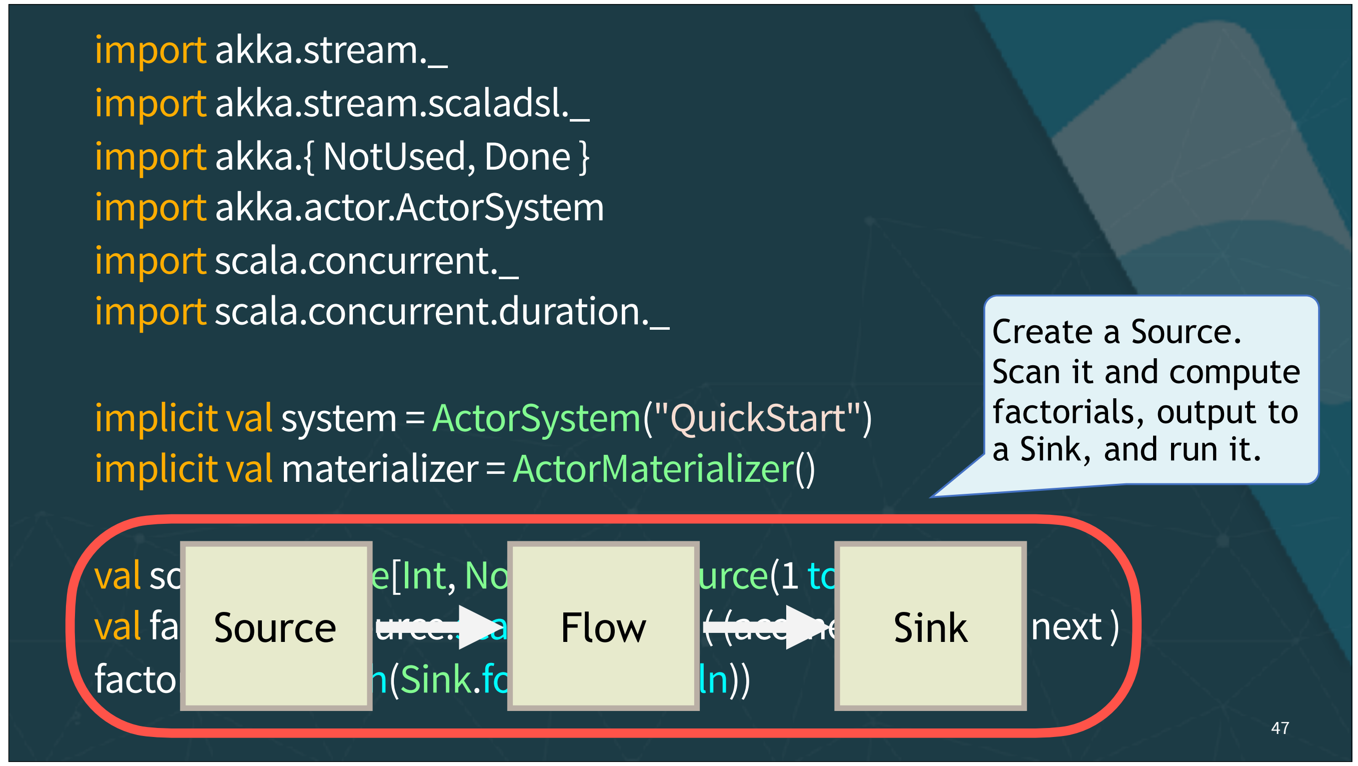
Initialize and specify now the stream is “materialized”

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

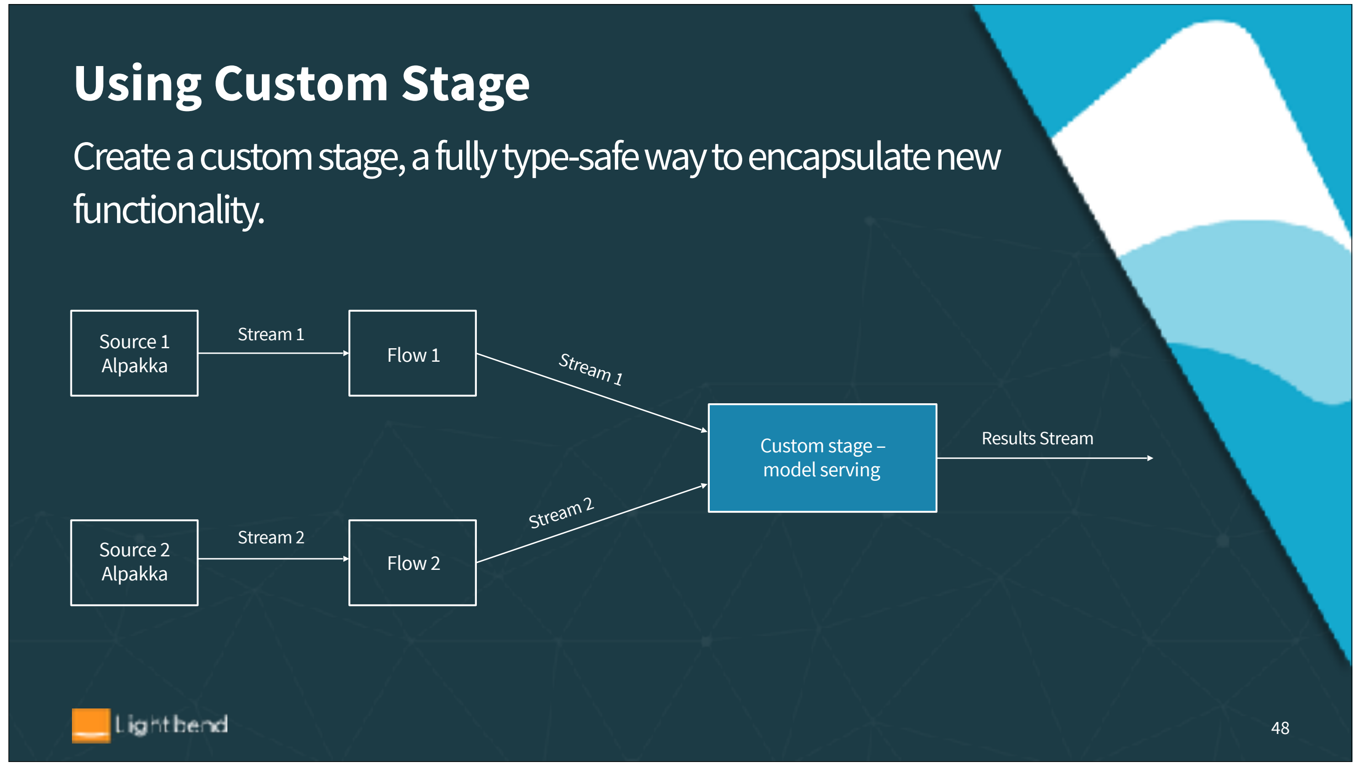
```scala
import akka.stream._
import akka.stream.scaladsl._
import akka.{ NotUsed, Done }
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._

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))
```

Create a Source.
Scan it and compute
factorials, output to
a Sink, and run it.

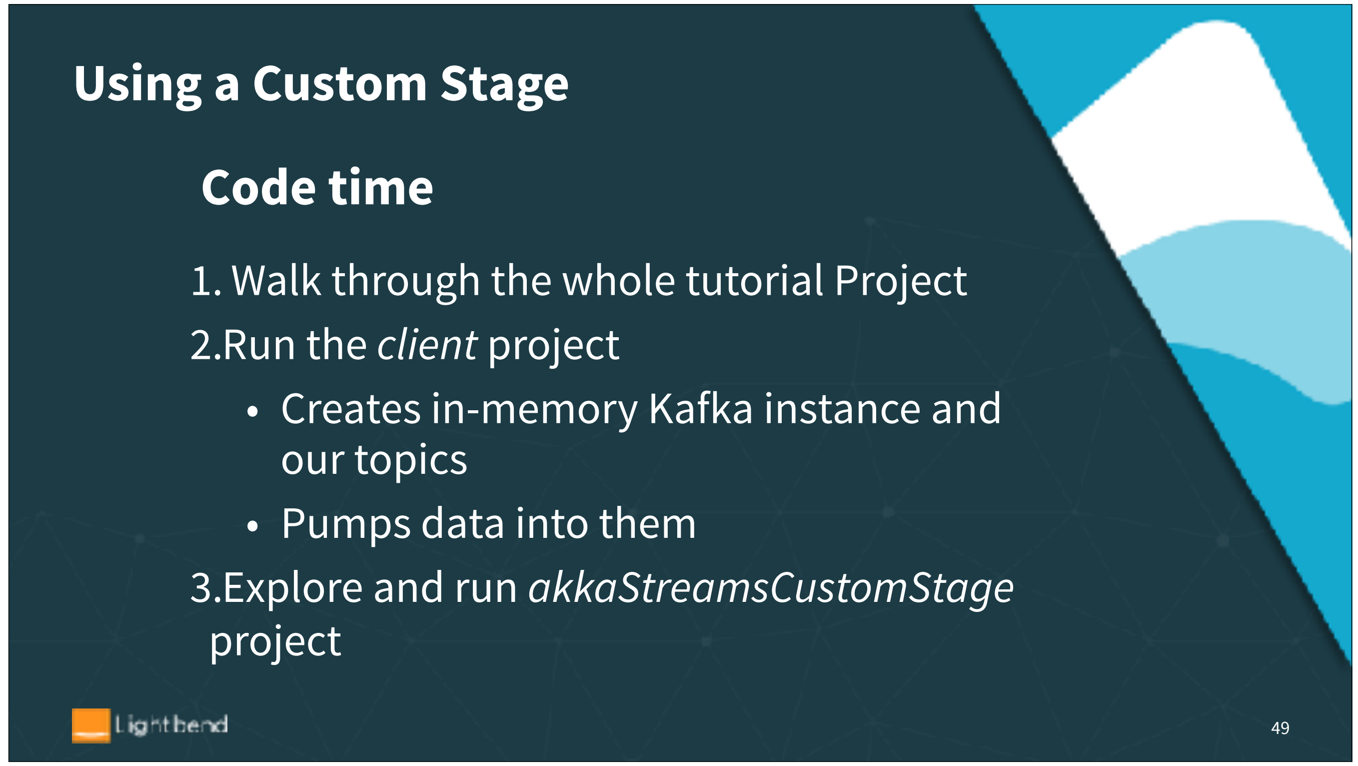
This example is in akkaStreamsCustomStage/simple-akka-streams-example.sc



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



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



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

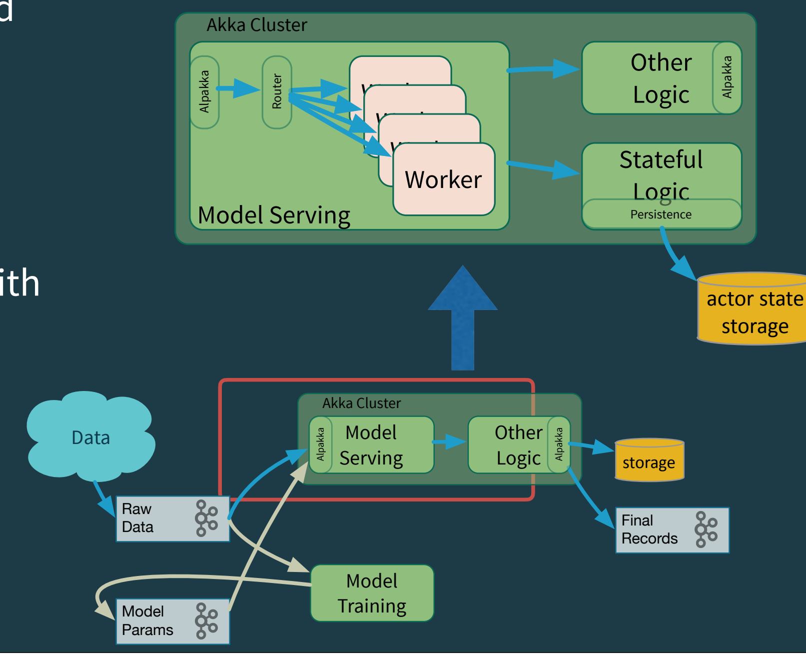
Exercises!

We've prepared some exercises. We may not have time during the course to work on them, but take a look at the *exercise* branch in the Git project (or the separate X.Y.Z_exercise download).

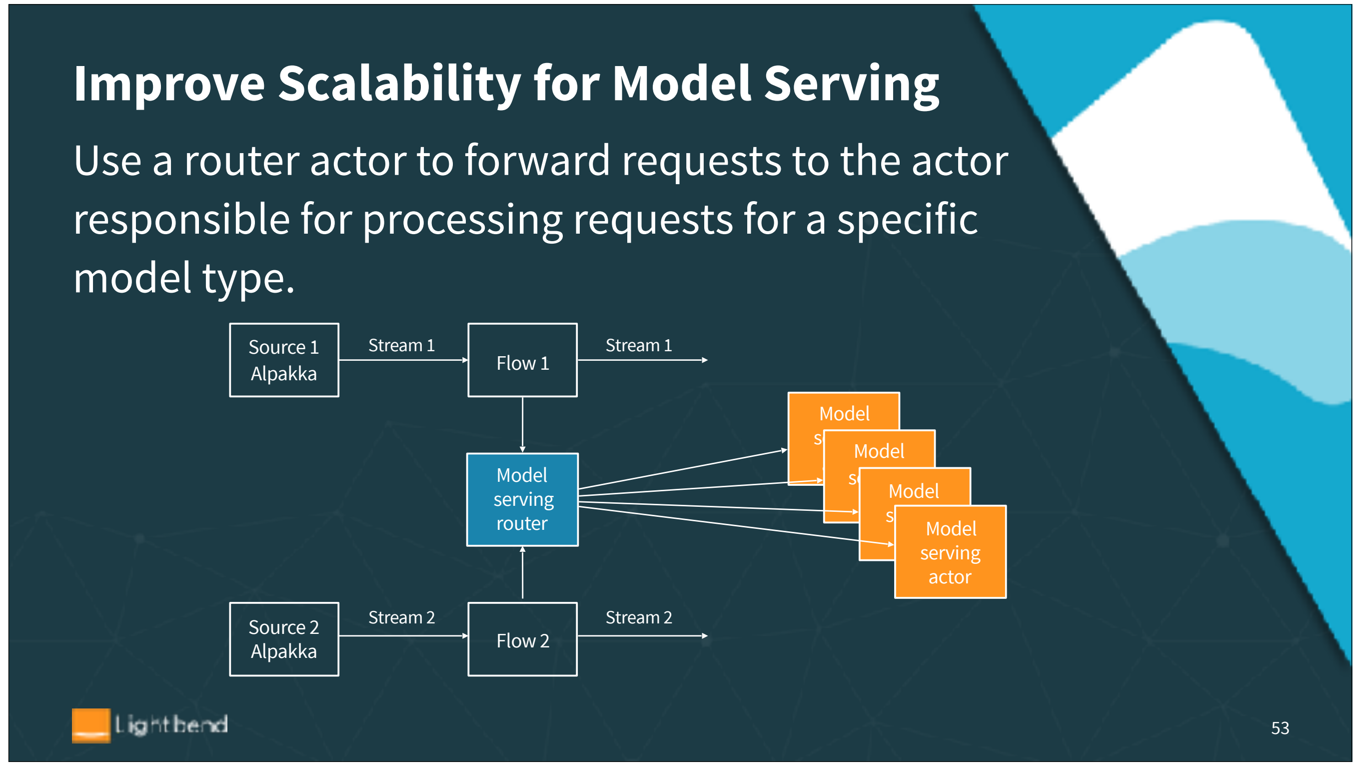
To find them, search for "// Exercise". The *master* branch implements the solutions.

Other Production Concerns

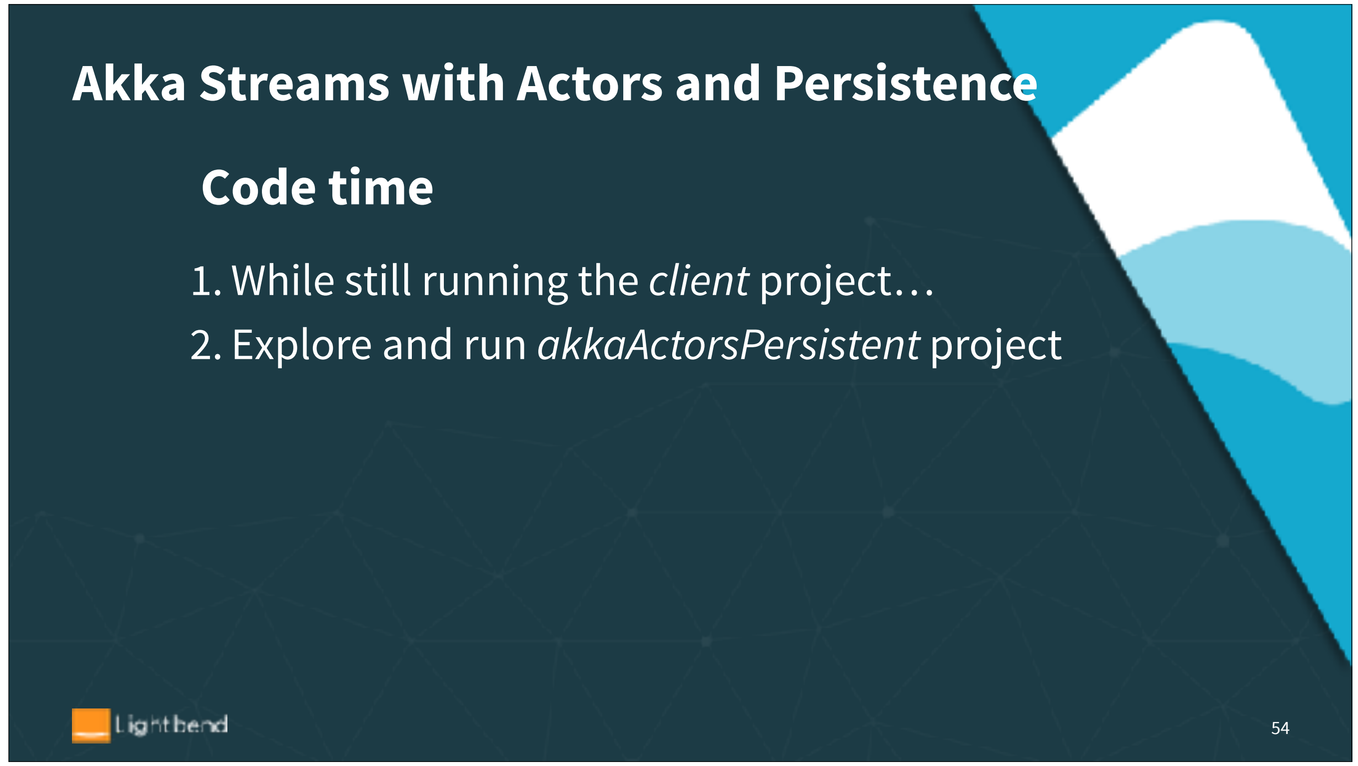
- Scale scoring with workers and routers, across a cluster
- Persist actor state with Akka Persistence
- Connect to *almost* anything with Alpakka
- *Lightbend Enterprise Suite*
  - for production



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.



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



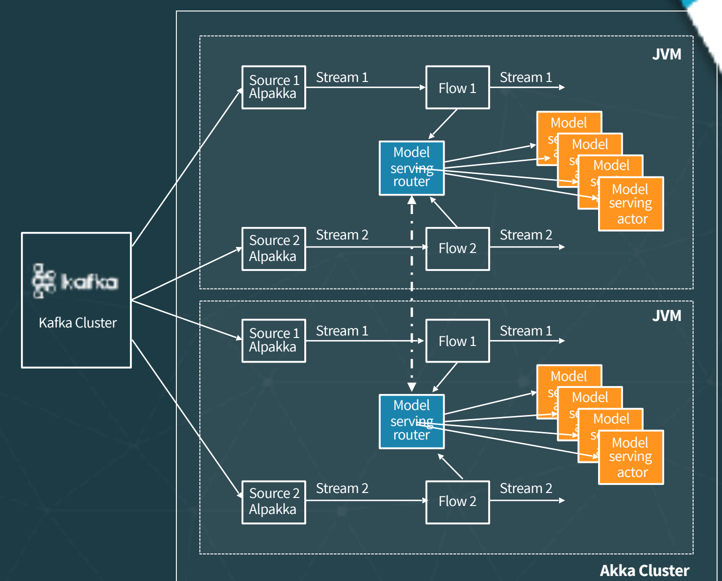
Custom stage is an elegant implementation but not scale well to a large number of models. Although a stage can contain a hash map of models, all of the execution will be happening at the same place

More Production Concerns

Using Akka Cluster

Two levels of scalability:

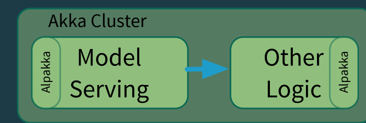
- Kafka partitioned topic allow to scale listeners according to the amount of partitions.
- Akka cluster sharing allows to split model serving actors across clusters.



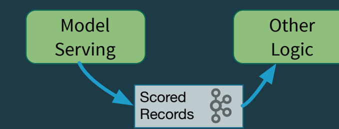
Lightbend

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

Go Direct or Through Kafka?



vs.



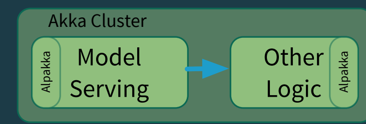
?

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

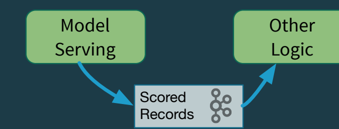
- Higher latency (including queue depth)
- Higher I/O and processing (marshaling) overhead
- Better potential reusability

Design choice: When is it better to use direct actor-to-actor (or service-to-service) messaging vs. going through a Kafka topic?

Go Direct or Through Kafka?



vs.



?

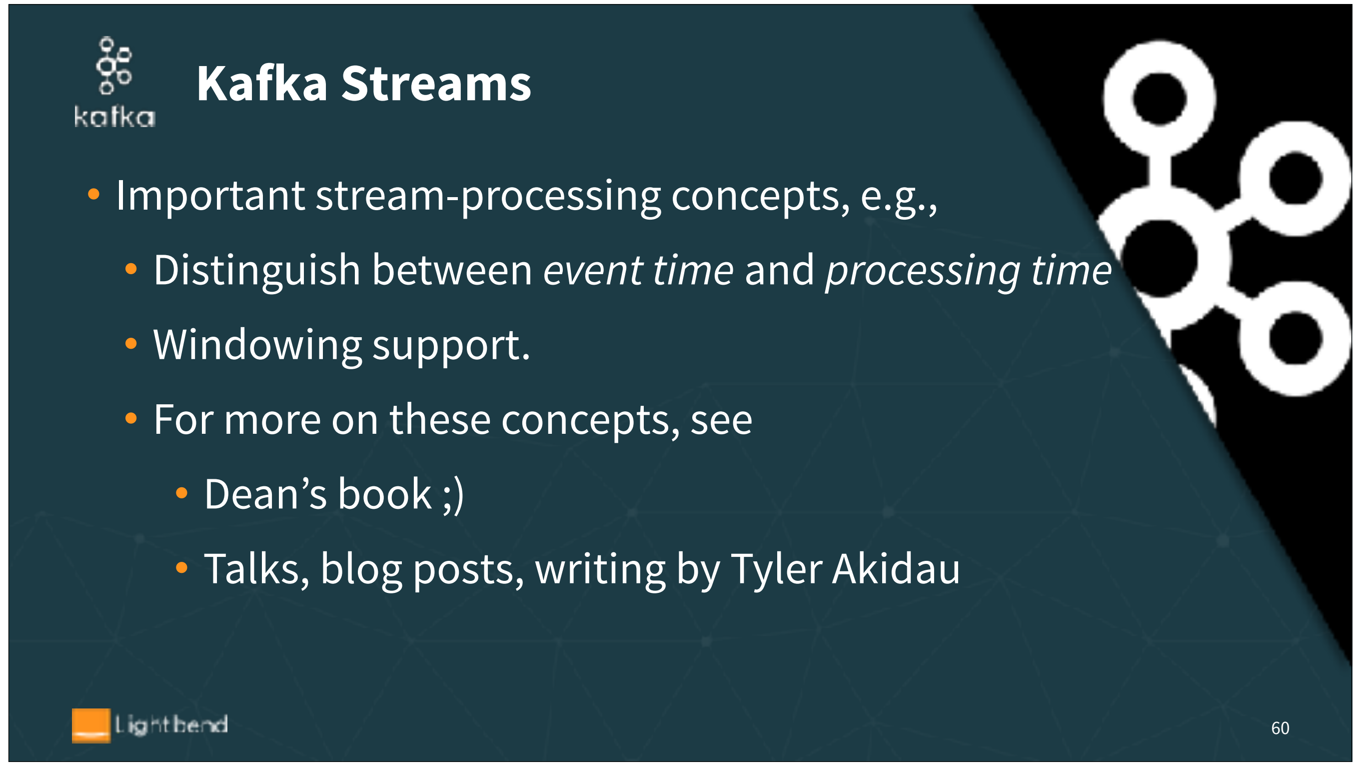
- *Reactive Streams* back pressure
- Direct coupling between sender and receiver, but indirectly through a URL

- Very deep buffer (partition limited by disk size.
- Strong decoupling - M producers, N consumers, completely disconnected

Design choice: When is it better to use direct actor-to-actor (or service-to-service) messaging vs. going through a Kafka topic?

Kafka Streams

- 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.

Kafka Streams

- KStream - per-record transformations
- KTable - aggregations, last value per key
  - Efficient management of application state

Kafka Streams

- Two types of APIs:
  - Process Topology (compare to Apache Storm)
  - DSL based on collection transformations
    - Compare to Spark, Flink, Scala collections.

Kafka Streams

- Provides Java API
- Lightbend donating a Scala API
  - https://github.com/lightbend/kafka-streams-scala
  - See also our convenience tools for distributed, queryable state: https://github.com/lightbend/kafka-streams-query
- SQL!

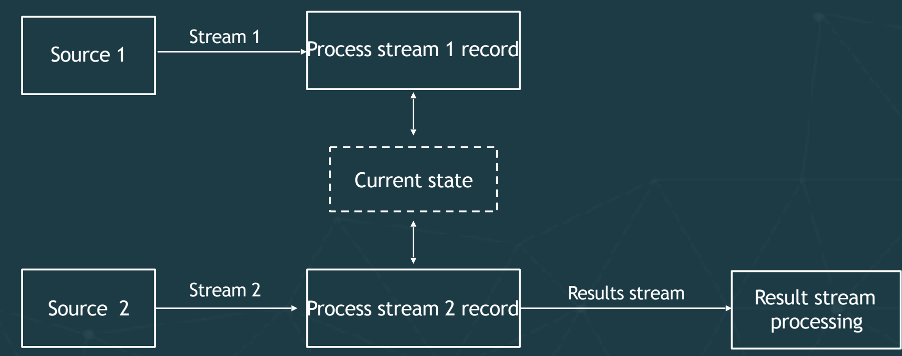
Kafka Streams

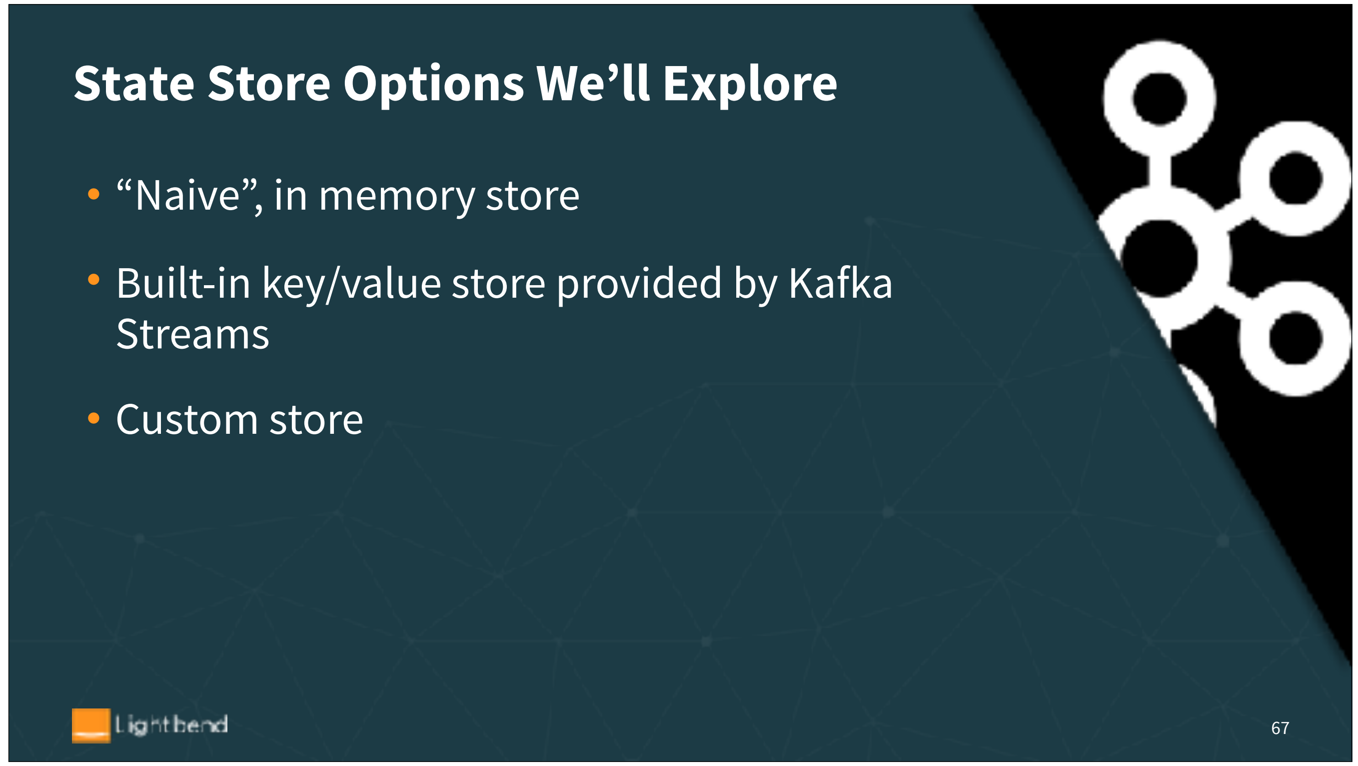
- Low overhead
- Read from and write to Kafka topics, memory
  - Could use Kafka Connect for other sources and sinks
- Load balance and scale based on partitioning of topics
- Built-in support for Queryable State

Kafka Streams

- Ideally suited for:
  - ETL -> KStreams
  - Aggregations -> KTable
  - Joins, including Stream and Table joins
  - “Effectively once” semantics
- Commercial support from Confluent, Lightbend, and others

Model Serving With Kafka Streams





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

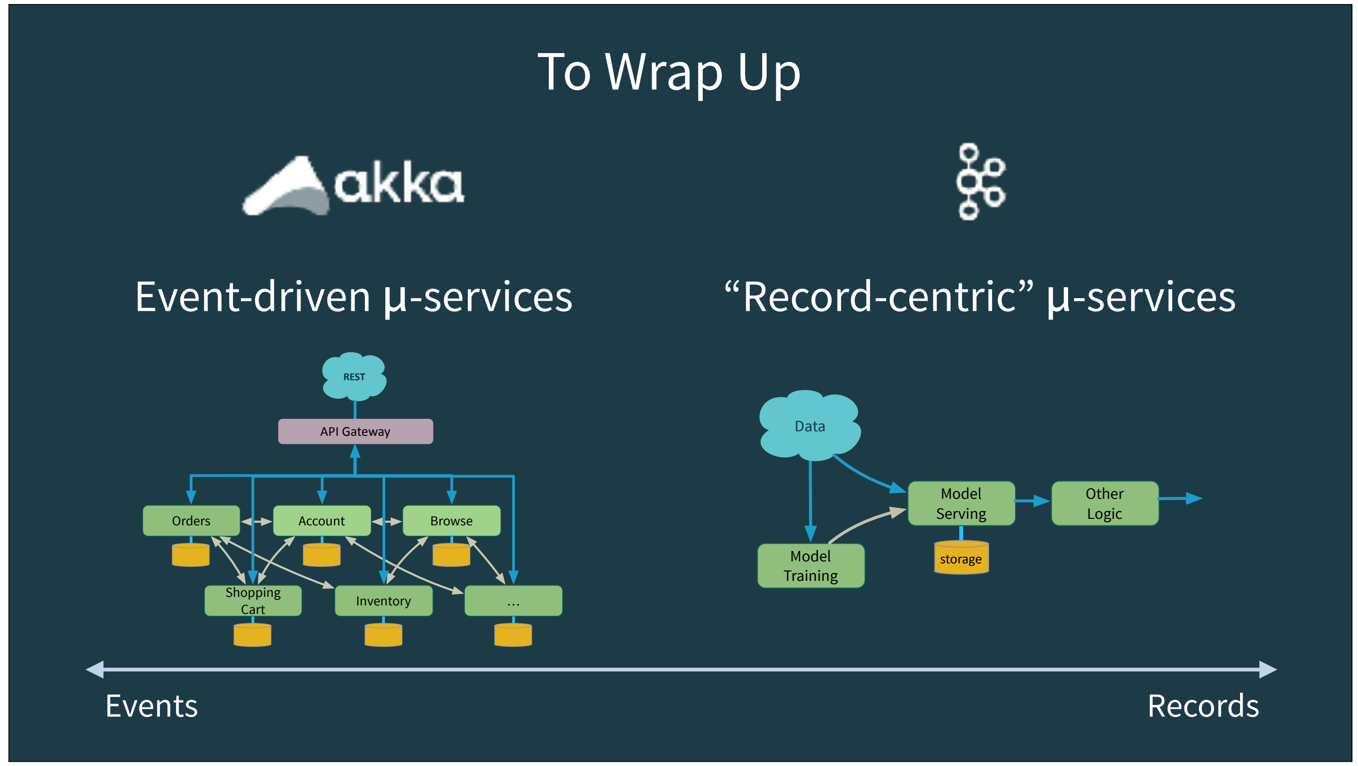
1. Still running the *client* project
2. Explore and run: *kafkaStreamsModelServerInMemoryStore*

We probably won’t get to this implementation and the next, but you can look at them on your own.

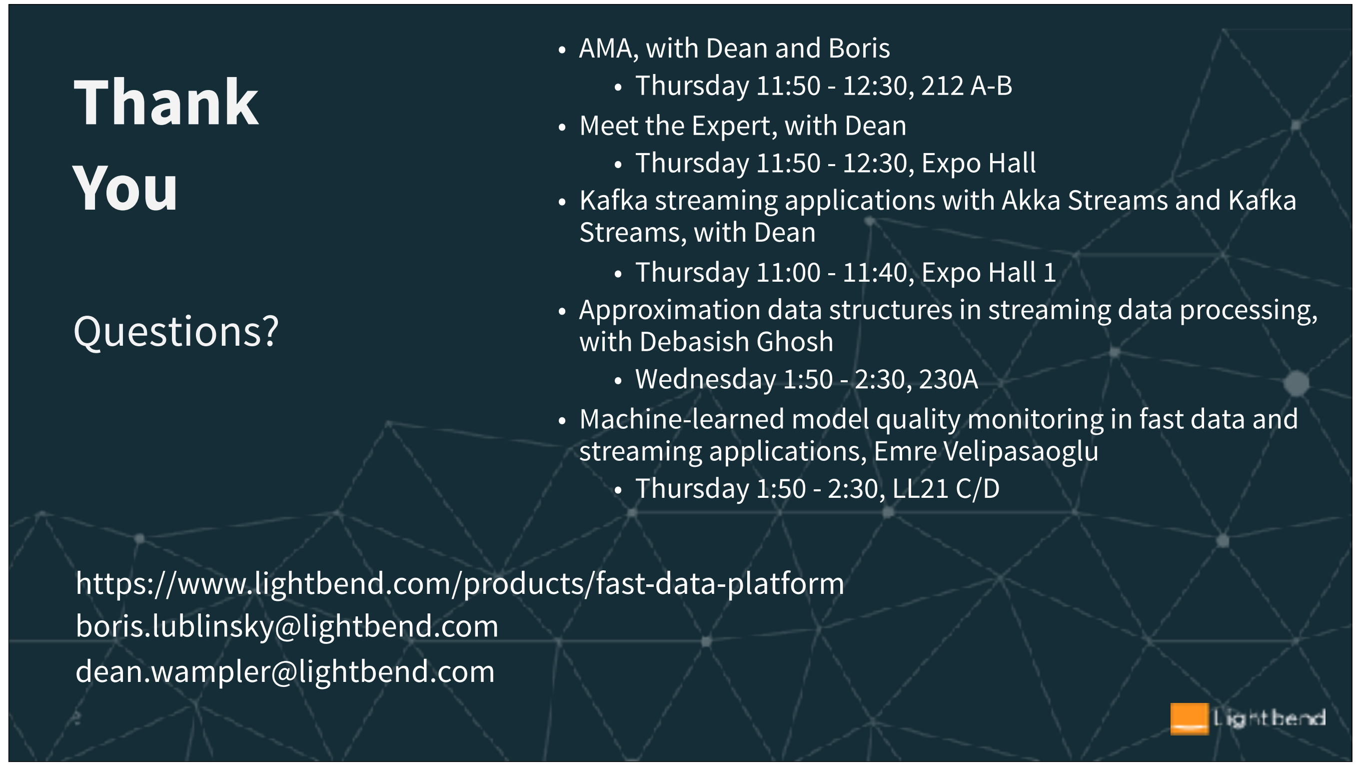
Model Serving With Kafka Streams, KV Store

Code time (as time permits)

1. Still running the *client* project
2. Explore and run: *kafkaStreamsModelServerCustomStore*



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".



Thank you! Please check out the other Strata San Jose sessions by Boris, Dean, and our colleagues Debasish and Emre. Check out our Fast Data Platform for commercial options for building and running microservices with Kafka, Akka Streams, and Kafka Streams.