

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

Boris Lublinsky and Dean Wampler, Lightbend

[boris.lublinsky@lightbend.com](mailto:boris.lublinsky@lightbend.com)

[dean.wampler@lightbend.com](mailto:dean.wampler@lightbend.com)

# Outline

- Overview of streaming architectures
  - Kafka, Spark, Flink, Akka Streams, Kafka Streams
- Running example: Serving machine learning models
- Streaming in a microservice context
  - Akka Streams
  - Kafka Streams
- Wrap up

# About Streaming Architectures

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

The background of the slide features two overlapping O'Reilly book covers. The top cover is red with a black spine and the text 'Dean Wampler' in white. The bottom cover is also red with a black spine and the text 'Fast Data Architectures for Streaming Applications' in white. The O'Reilly logo is visible on the spines of both books.

O'REILLY®

# Fast Data Architectures for Streaming Applications

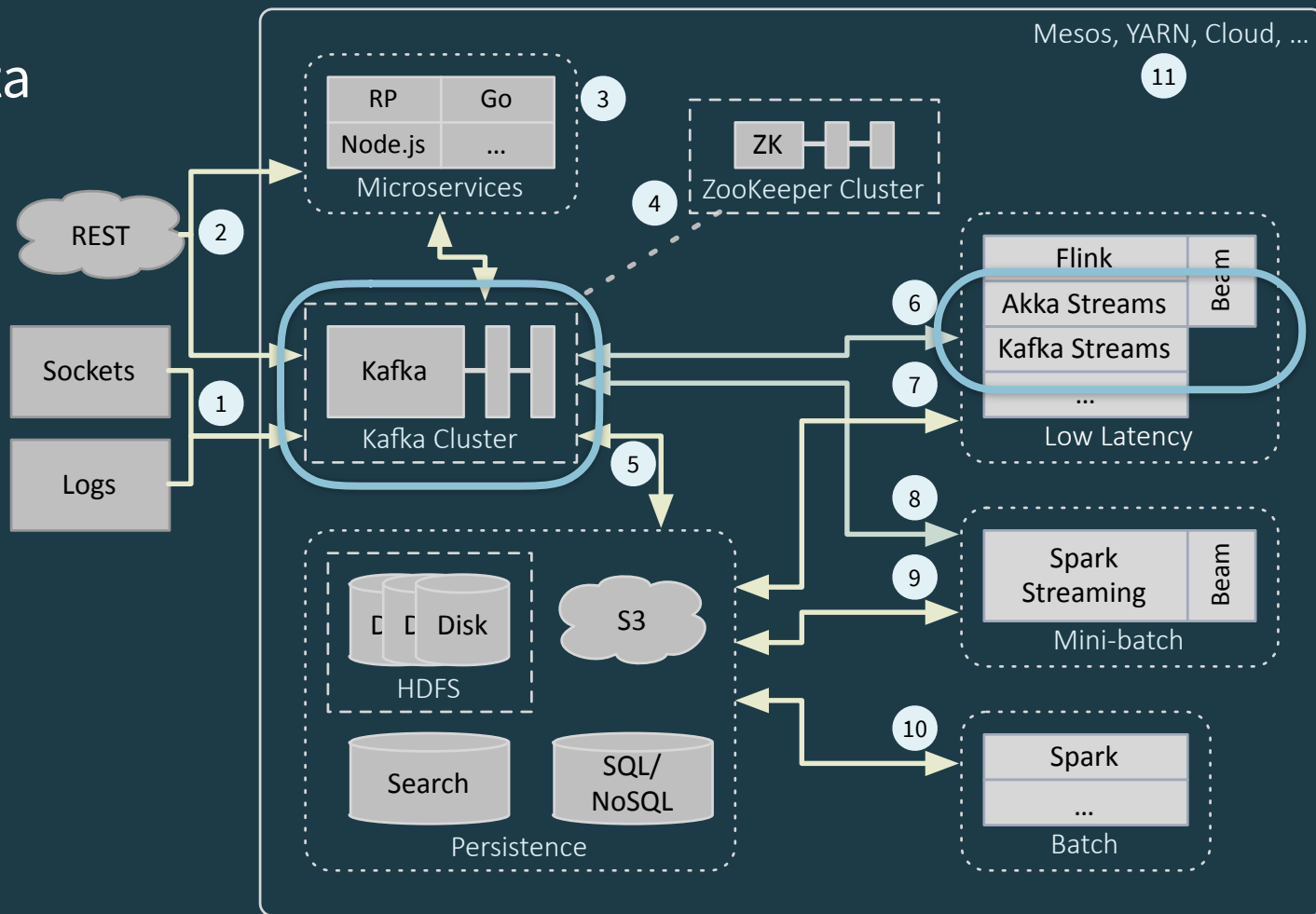
**Getting Answers Now from Data Sets that Never End**

By Dean Wampler, Ph. D., VP of Fast Data Engineering

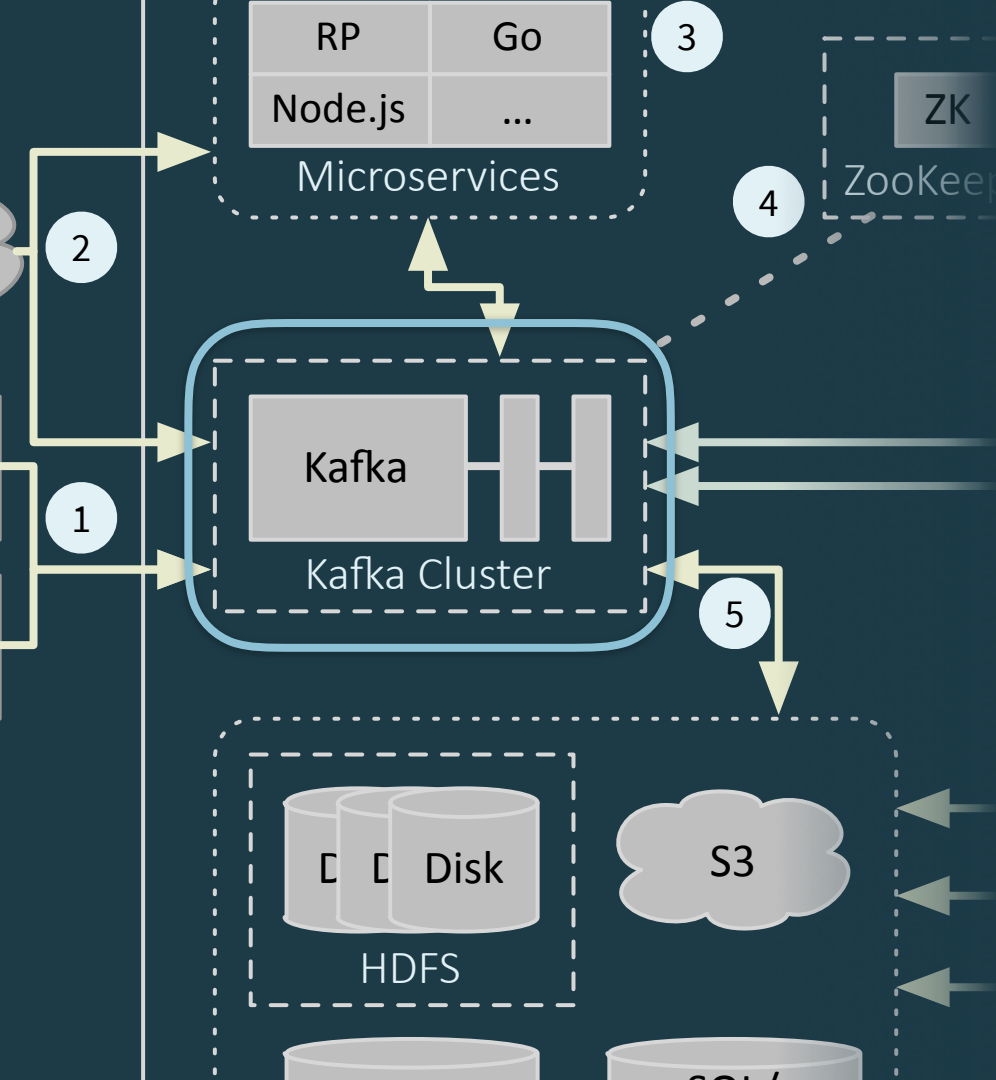
[Get Your Free Copy](#)

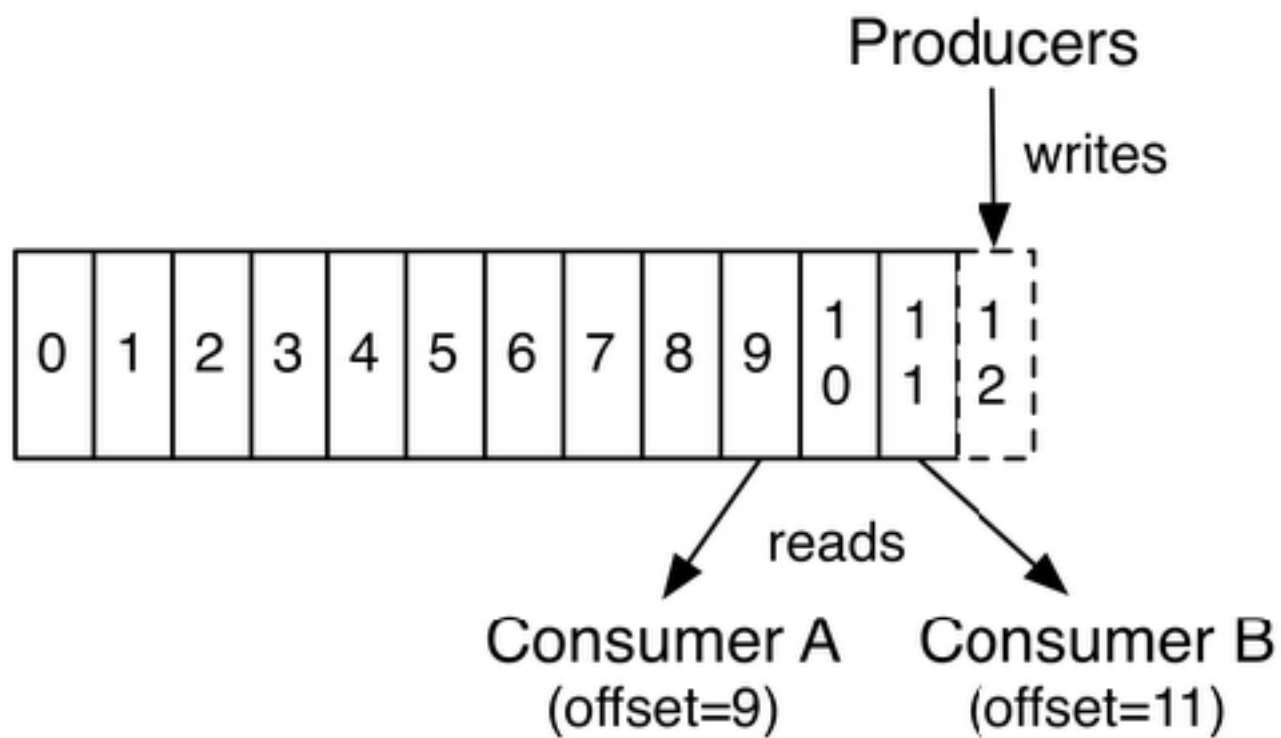
Today's focus:

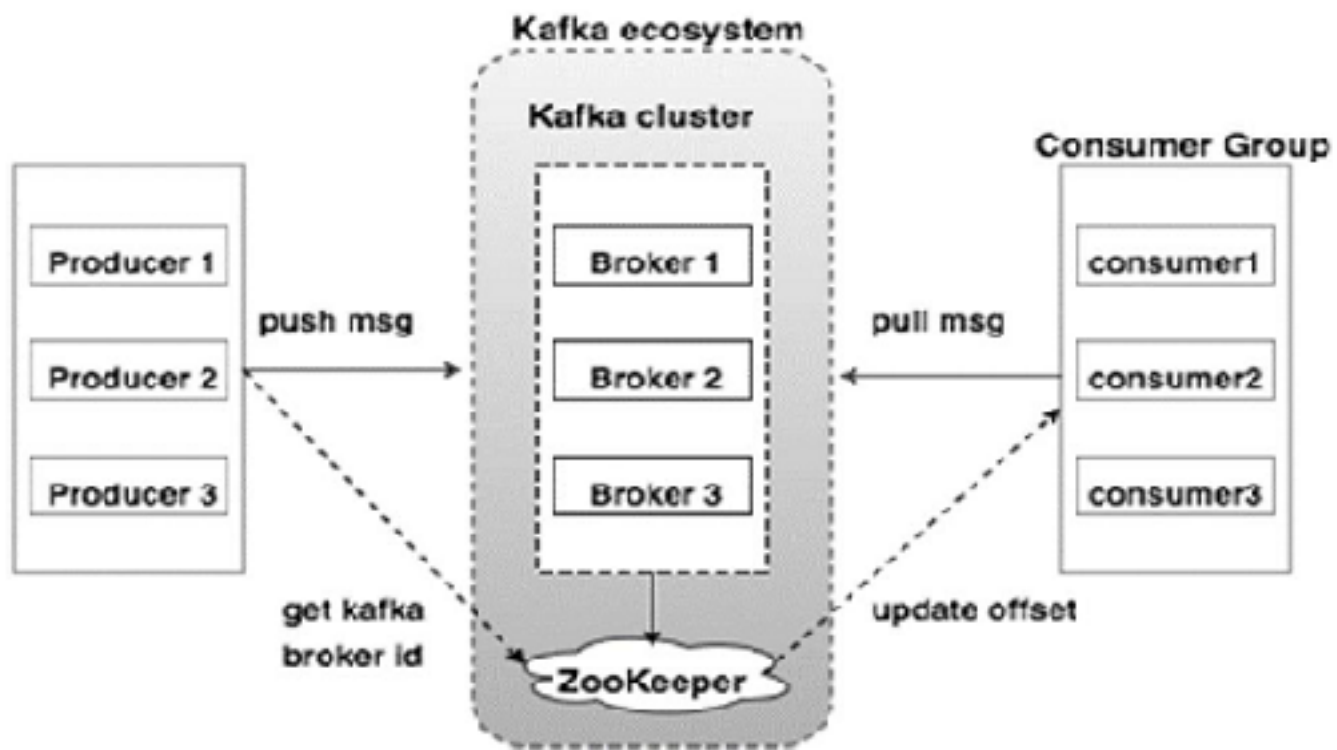
- Kafka - the data backplane
- Akka Streams and Kafka Streams - streaming microservices



# Why Kafka?



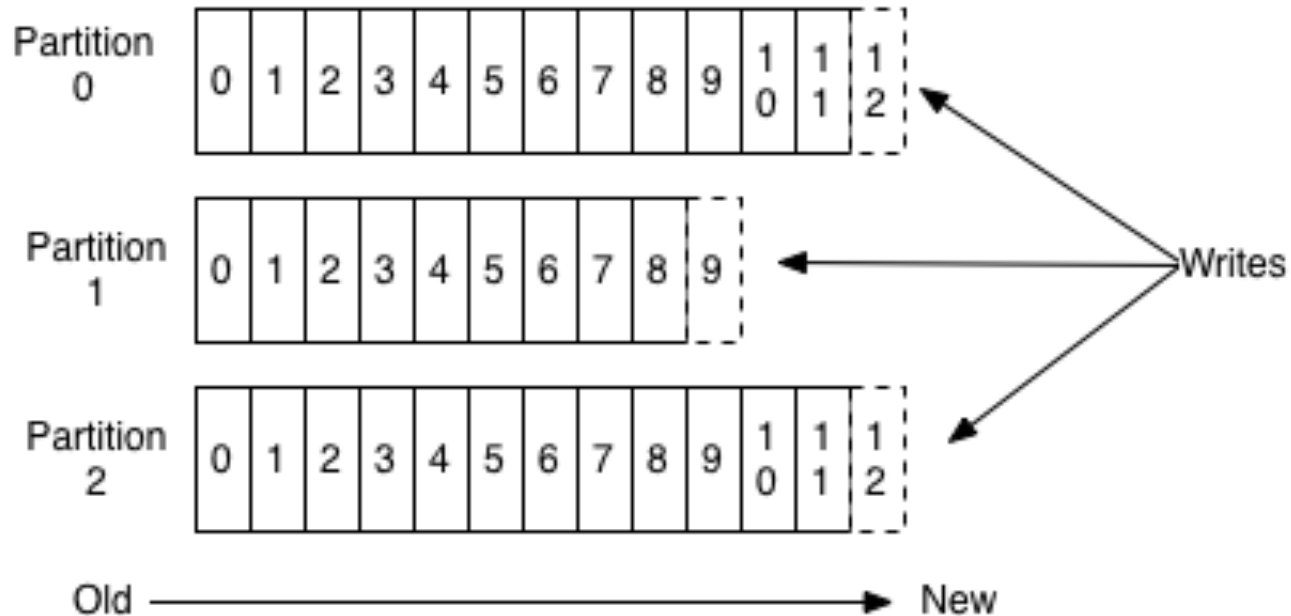




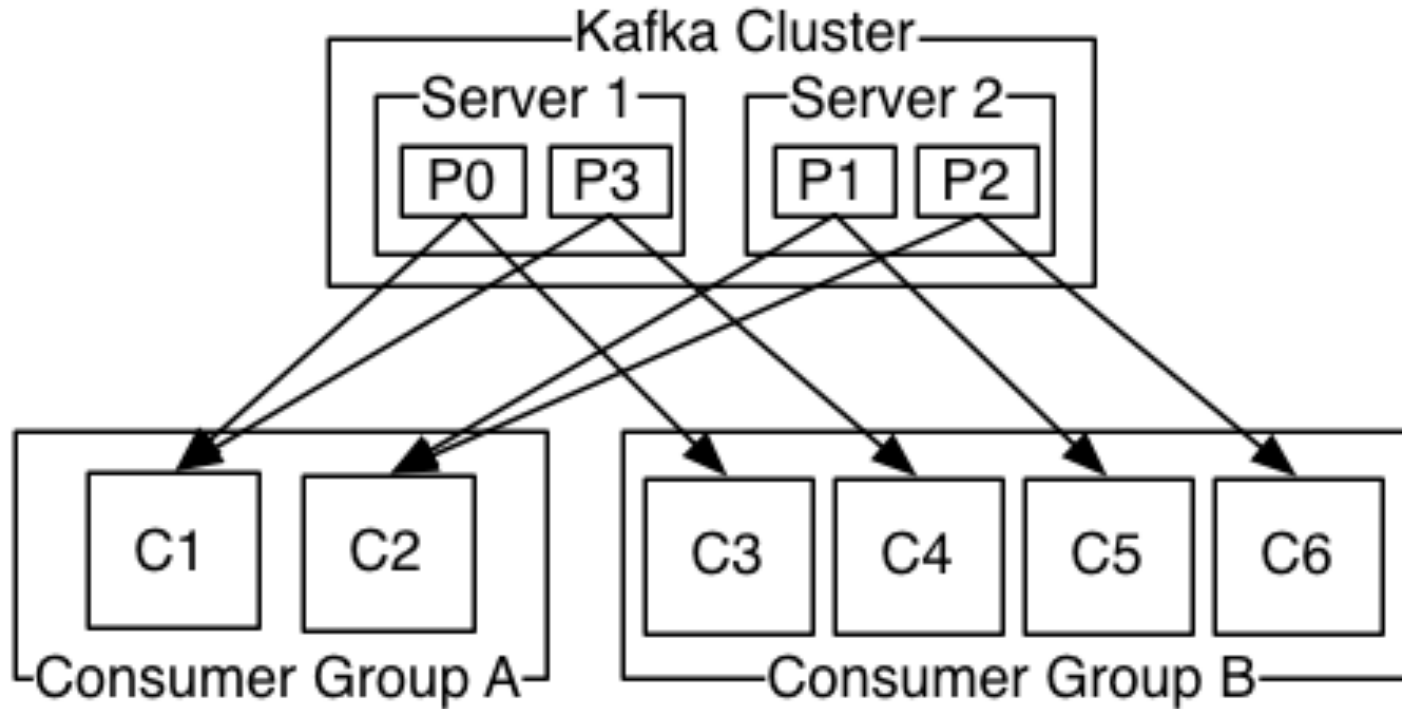


# A Topic and Its Partitions

## Anatomy of a Topic



# Consumer Groups



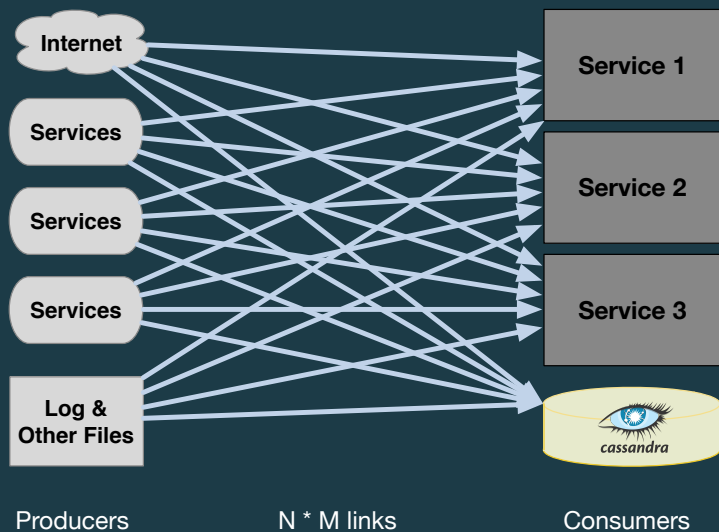
# Kafka Producers and Consumers

## Code time

1. Project overview
2. Explore and run the *client* project
  - Creates in-memory (“embedded”) Kafka instance and our topics
  - Pumps data into them

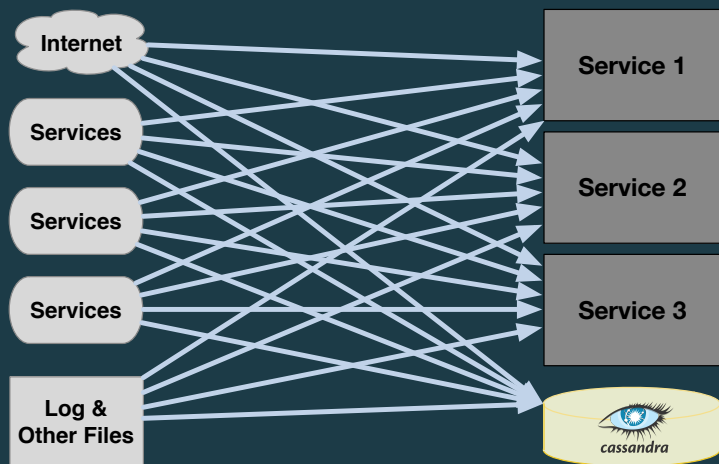
# Why Kafka for Connectivity?

Before:



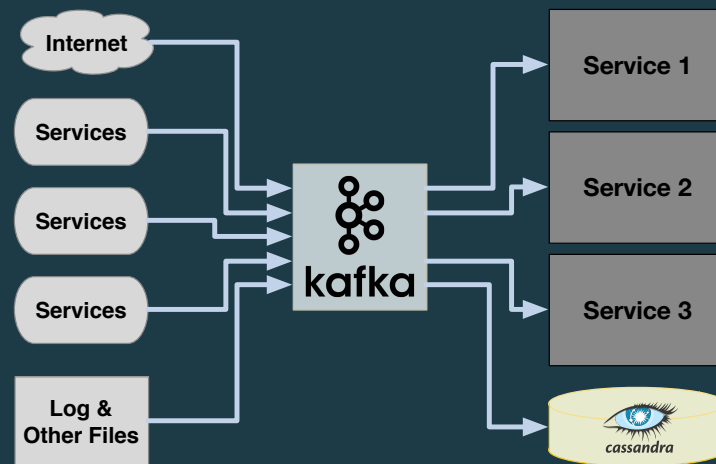
# Why Kafka for Connectivity?

Before:



$N * M$  links

After:

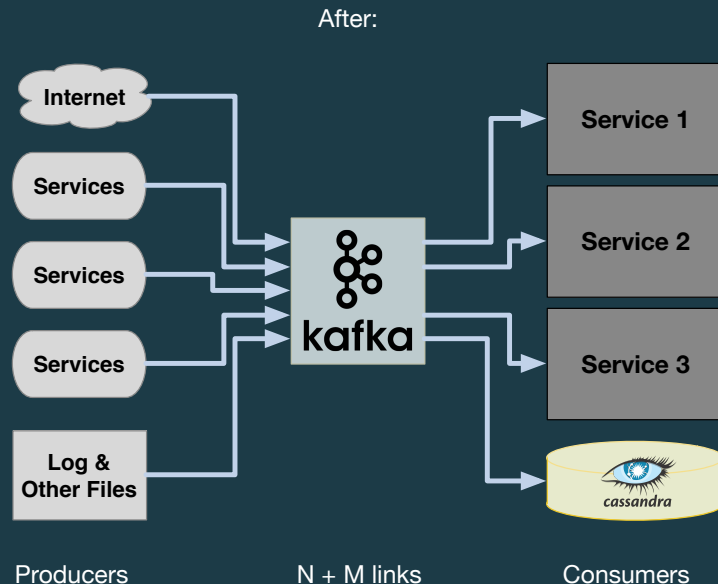


$N + M$  links

# Why Kafka for Connectivity?

Kafka:

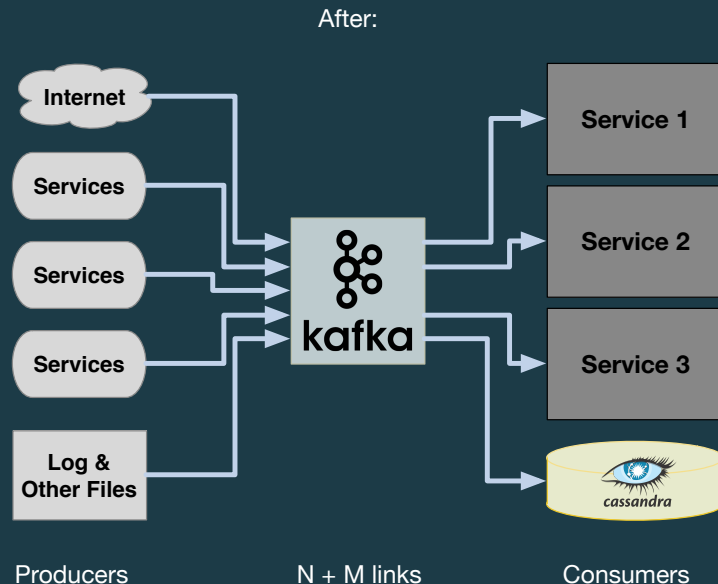
- Simplify dependencies between services
  - Improved data consistency
- Minimize data transmissions
- Reduce data loss when a service crashes



# Why Kafka for Connectivity?

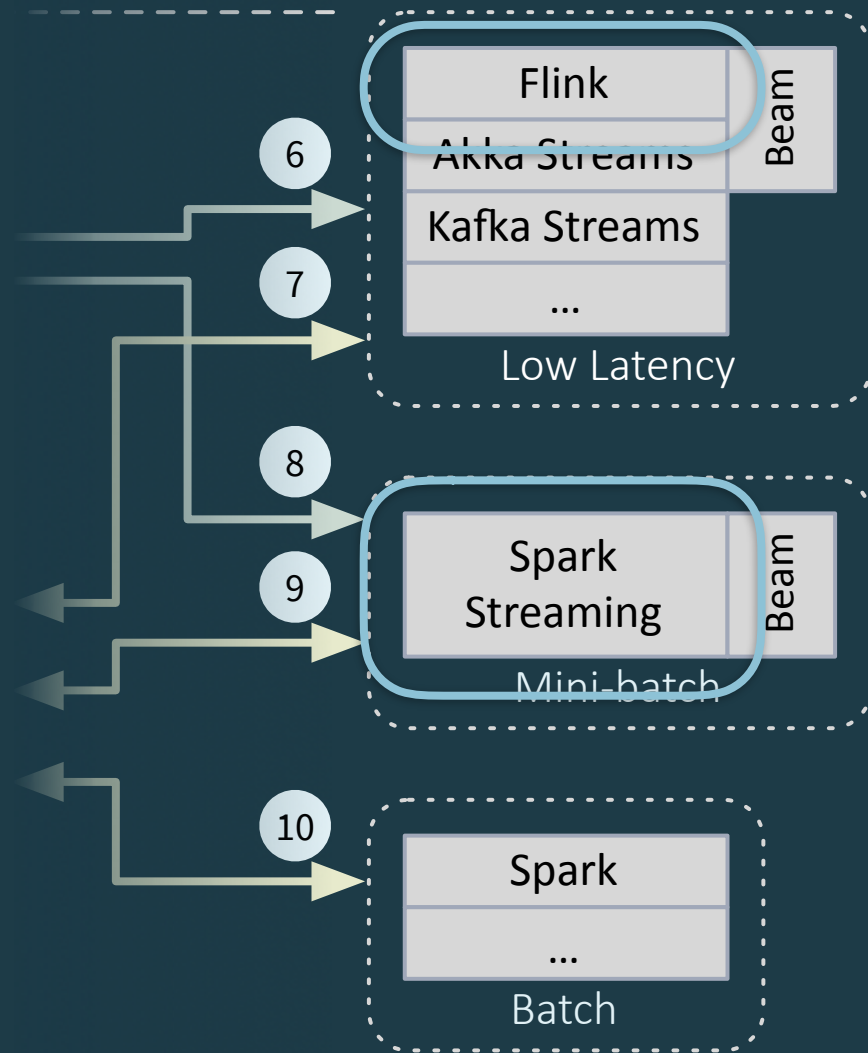
Kafka:

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



## Streaming Engines:

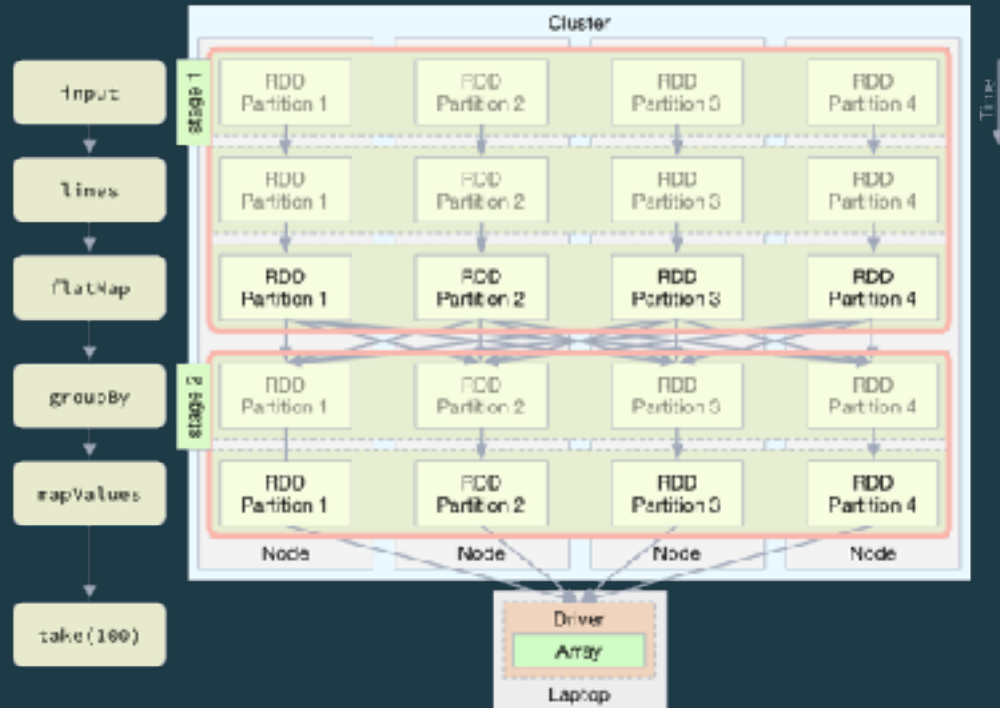
Spark, Flink - services to which you submit work. Large scale, automatic data partitioning.





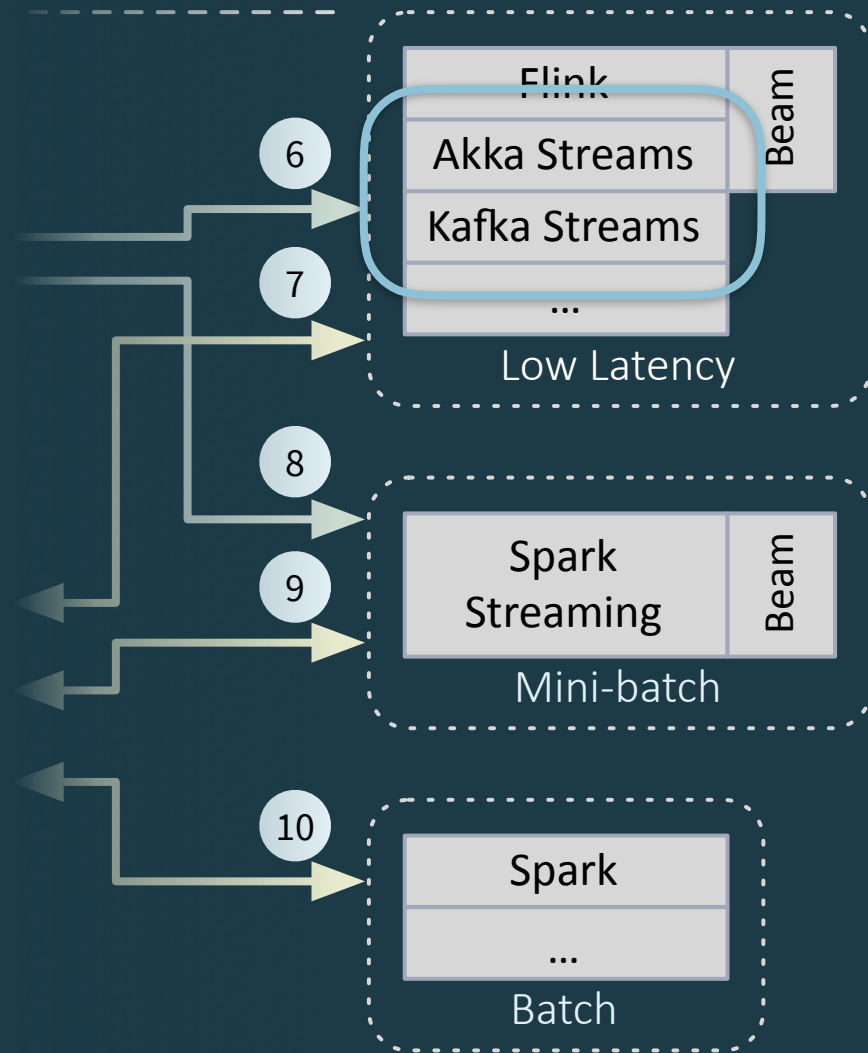
# Streaming Engines:

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



## Streaming Frameworks:

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



# Machine Learning and Model Serving: A Quick Introduction



O'REILLY®

# Serving Machine Learning Models

**A Guide to Architecture, Stream Processing Engines,  
and Frameworks**

By Boris Lublinsky, Fast Data Platform Architect

[Get Your Free Copy](#)

# ML Is Simple



Data



Magic



Happiness

# Maybe Not

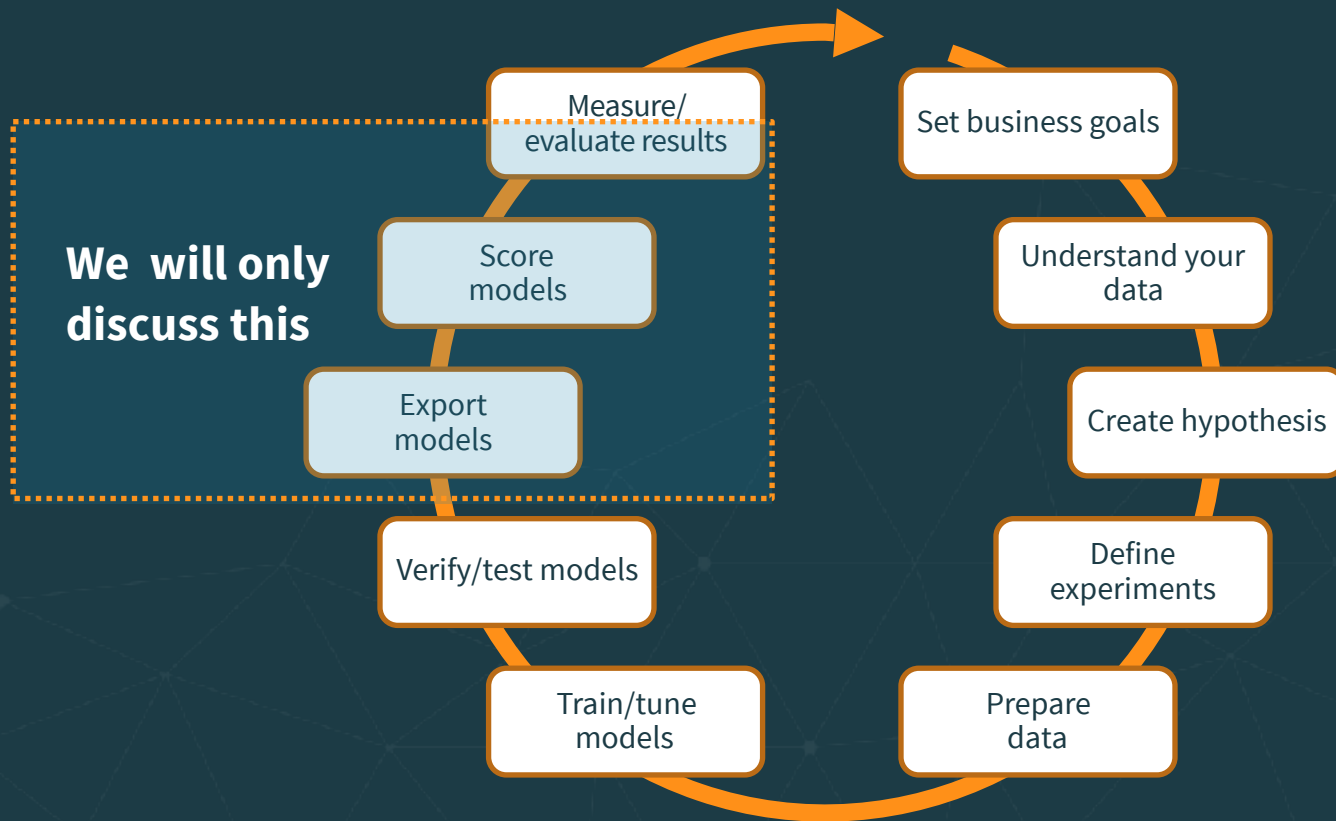




# Even If There Are Instructions



# The Reality





# What Is The Model?

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

for example:

**Linear regression:**  $y = a_c + a_1 * x_1 + \dots + a_n * x_n$

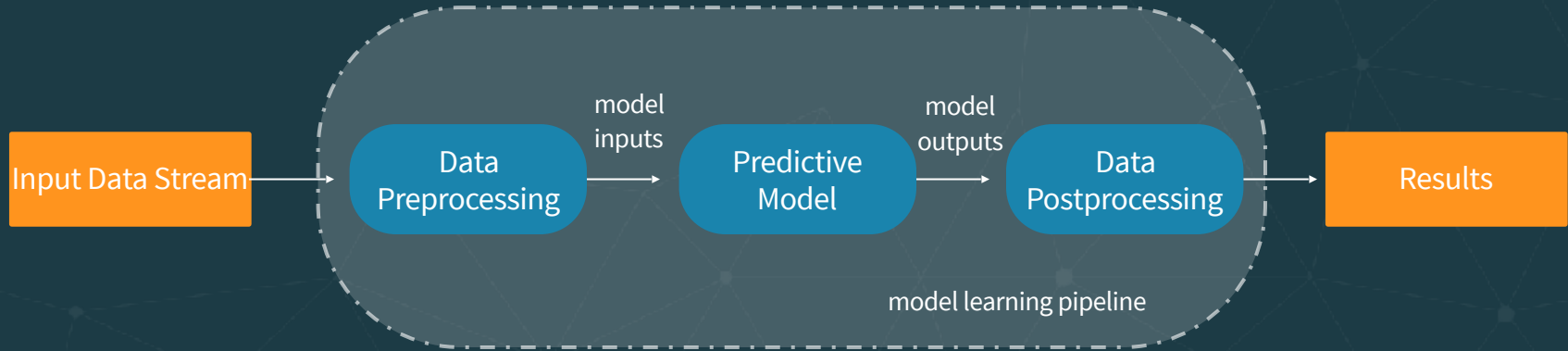
**Neural network:**  $f(x) = K(\sum_i w_i g_i(x))$

Such a definition of the model allows for an easy implementation of model's composition. From the implementation point of view it is just function composition



# Model Learning Pipeline

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

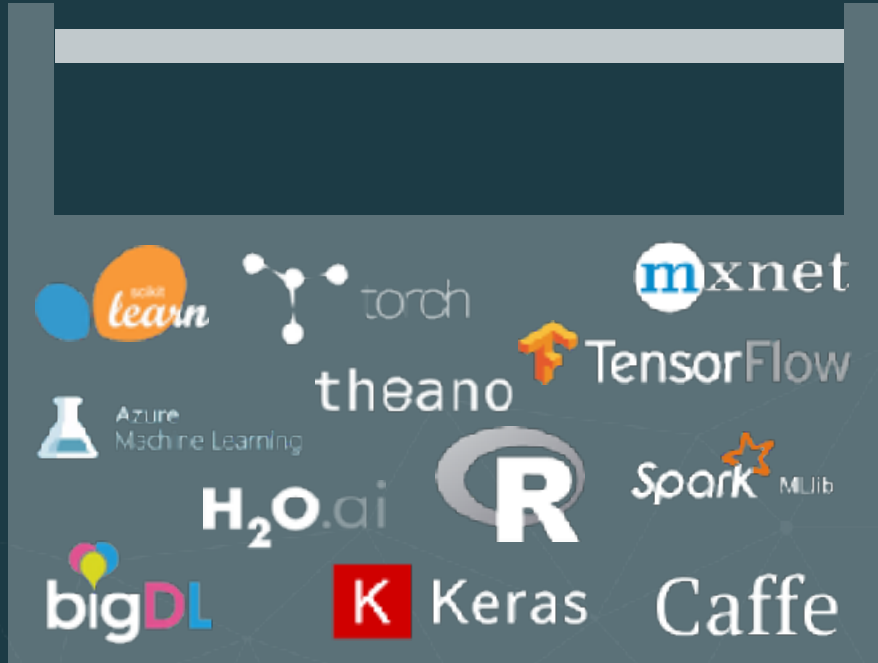


# Traditional Approach to Model Serving

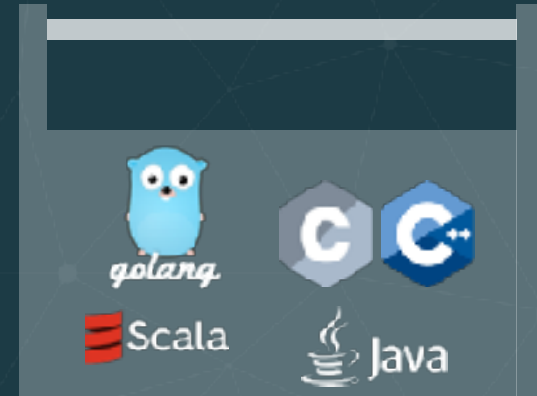
- Model is code
- This code has to be saved and then somehow imported into model serving

**Why is this problematic?**

# Impedance Mismatch

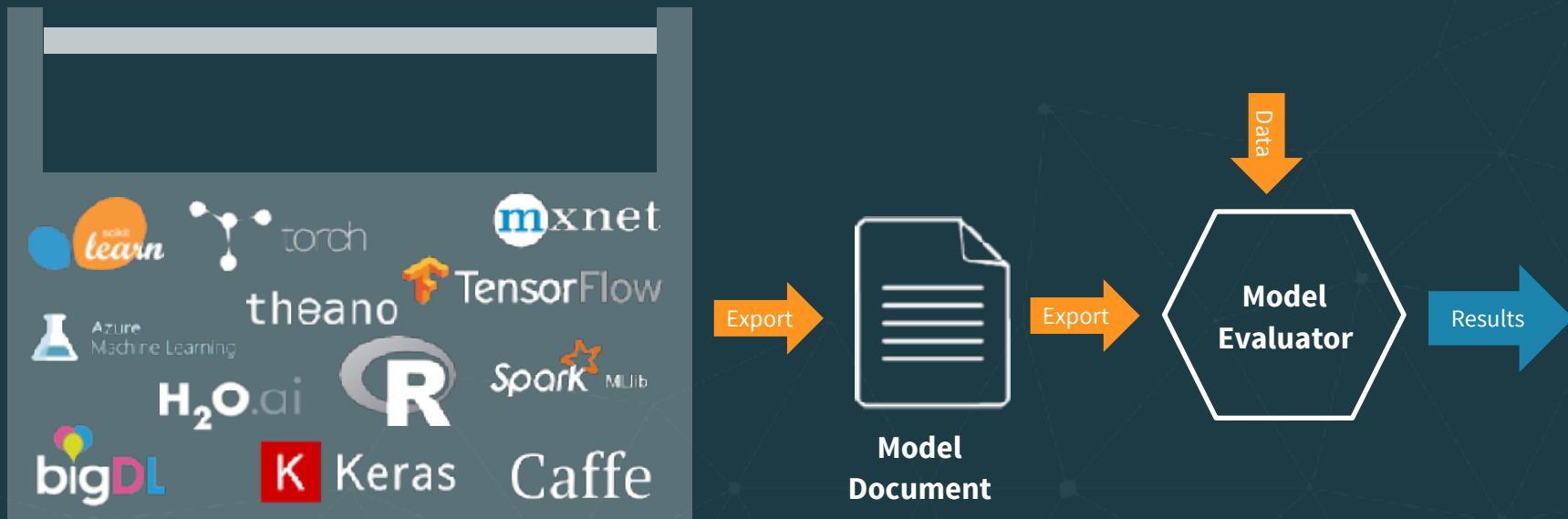


**Continually expanding  
Data Scientist toolbox**



**Defined Software  
Engineer toolbox**

# Alternative - Model As Data



Standards



Portable  
Format for  
Analytics (PFA)



# Exporting Model 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>

# 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





# Evaluating Tensorflow Model

- Tensorflow is implemented in C++ with a Python interface.
- In order to simplify Tensorflow usage from Java, in 2017 Google introduced Tensorflow Java API.
- Tensorflow Java API supports importing an exported model and allows to use it for scoring.



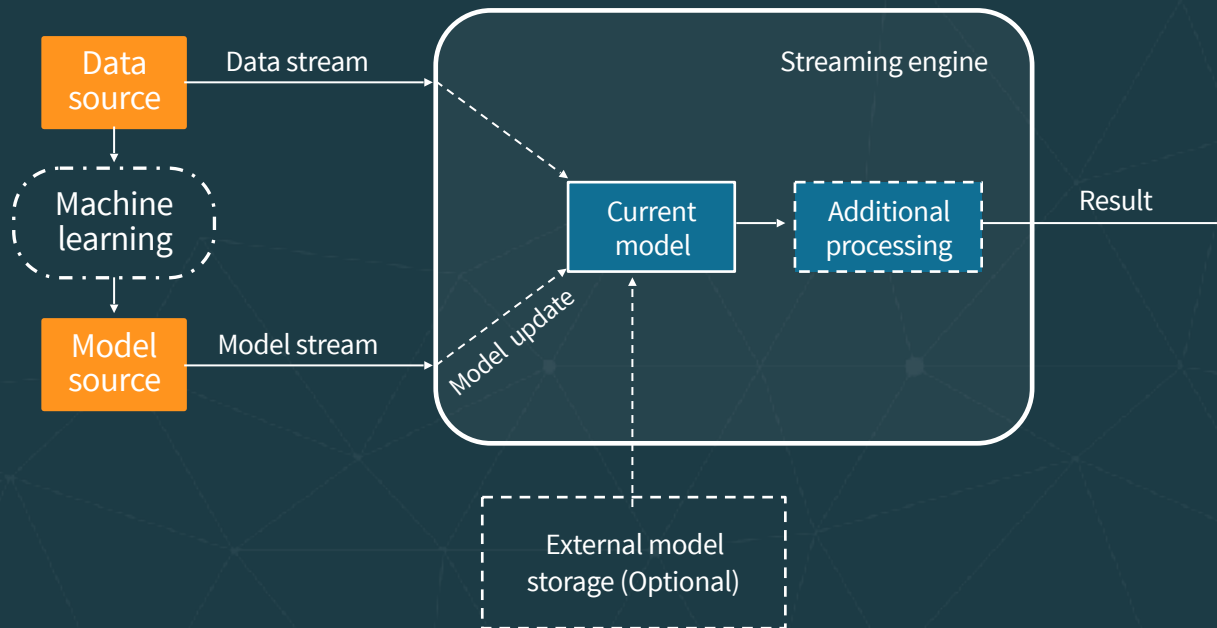
# Additional Considerations – Model Lifecycle

- 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).



# 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;

}

}

# Model Representation (Scala)

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

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

## Side Note: Monitoring

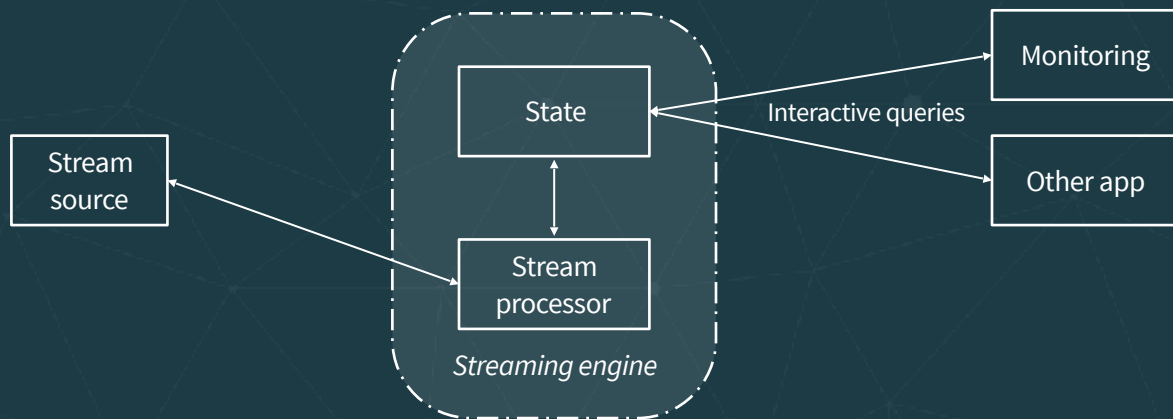
Model monitoring should provide information about usage, behavior, performance and lifecycle of the deployed models

```
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.



# Microservice All the Things!





**Scott Hanselman** ✓

@shanselman

Follow



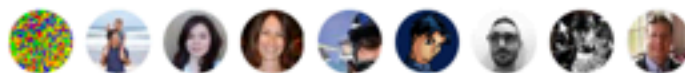
Microservices, for when your in-process methods have too little latency.

**Dave Cheney** @davecheney

Microservices, for when function calls are too reliable.

4:11 AM - 25 Feb 2018

207 Retweets 566 Likes



25



207

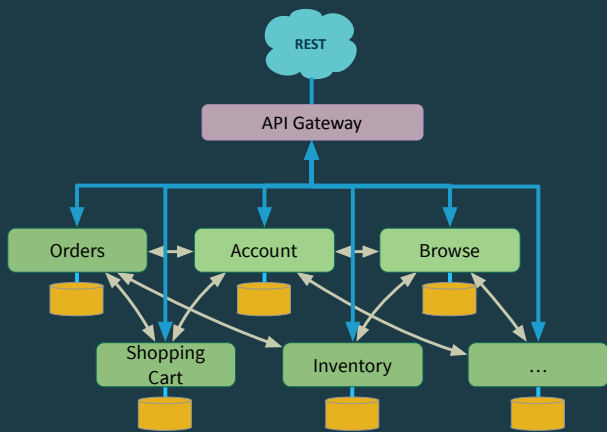


566

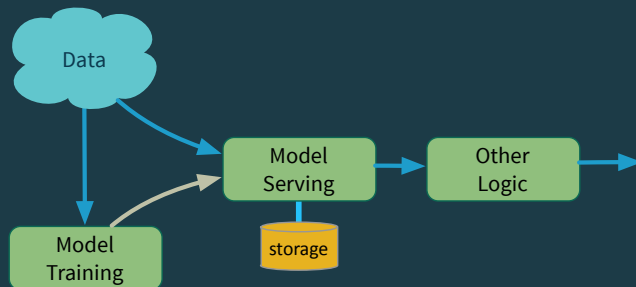


# A Spectrum of Microservices

## Event-driven $\mu$ -services



## "Record-centric" $\mu$ -services



Events

Records

# A Spectrum of Microservices



## Event-driven $\mu$ -services



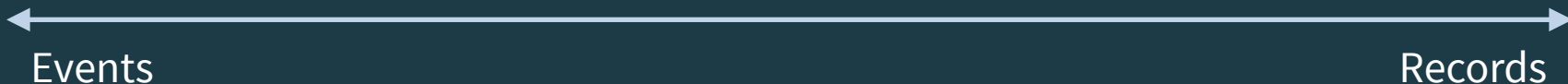
Akka emerged from the left-hand side of the spectrum, the world of highly *Reactive* microservices.

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

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

```
graph LR; Data((Data)) --> MT[Model Training]; Data --> MS[Model Serving]; MT --> MS; Storage[(storage)] --> MS; MS --> OL[Other Logic]; OL --> Exit(( ));
```

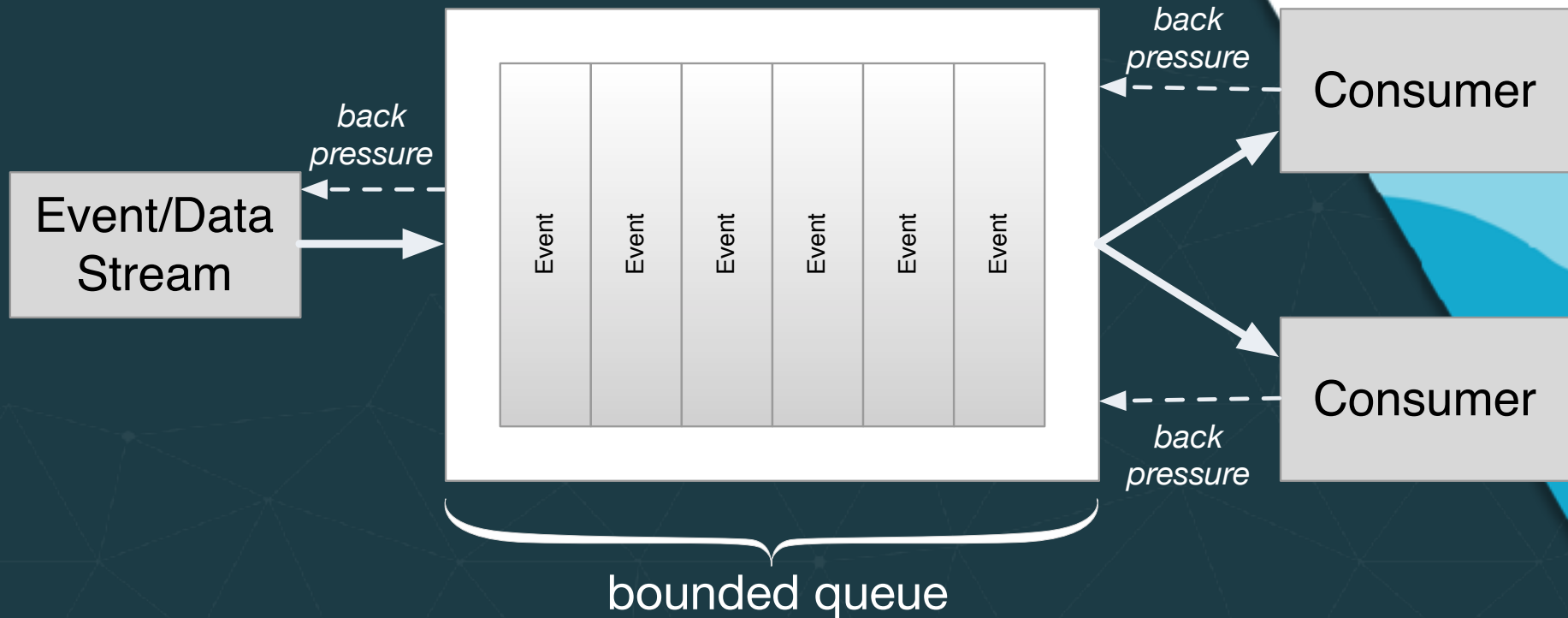
The diagram illustrates a machine learning pipeline. It starts with a cloud labeled 'Data'. An arrow points from 'Data' to a box labeled 'Model Training'. Another arrow points from 'Data' to a box labeled 'Model Serving'. A third arrow points from 'Model Training' to 'Model Serving'. Below 'Model Serving' is a box labeled 'storage' with an arrow pointing up to 'Model Serving'. Finally, an arrow points from 'Model Serving' to a box labeled 'Other Logic', which then has an arrow pointing to the right, indicating the flow continues.

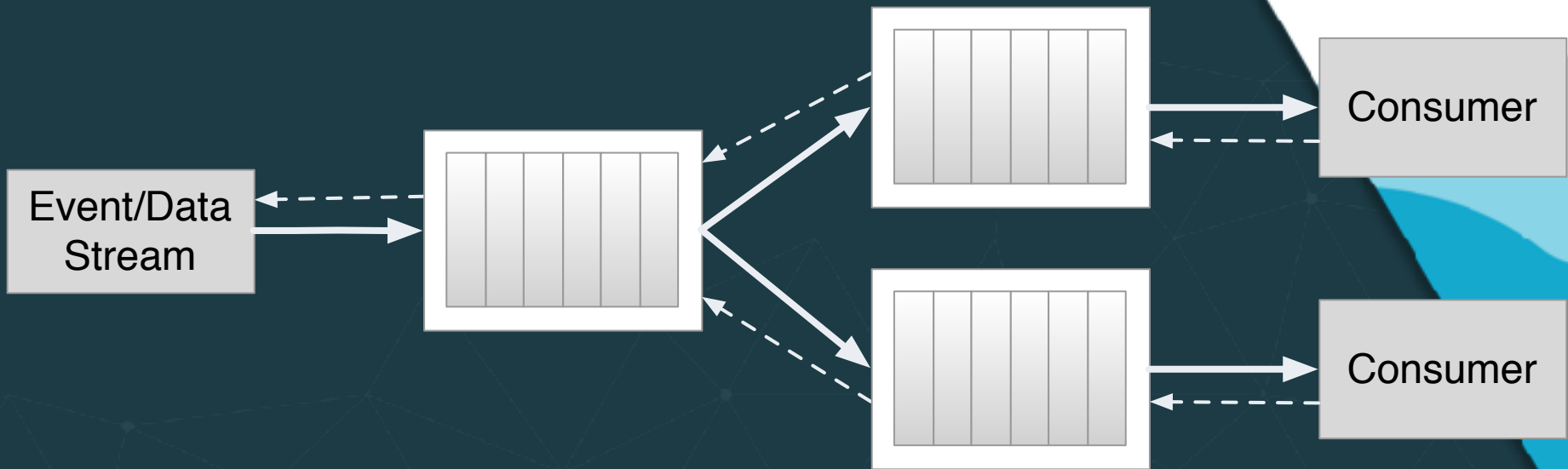


# Akka Streams



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









- Part of the Akka ecosystem
  - Akka Actors, Akka Cluster, Akka HTTP, Akka Persistence, ...
  - Alpakka - rich connection library
    - like Camel, but implements Reactive Streams
- Commercial support from Lightbend

- A very simple example to get the “gist”...

```
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._
```

Imports!

```
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()
```

Initialize and specify  
now the stream is  
“materialized”

```
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()
```

```
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 of  
Ints. Second type  
represents a hook used  
for “materialization” -  
not used here

```
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))
```

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

```
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))
```

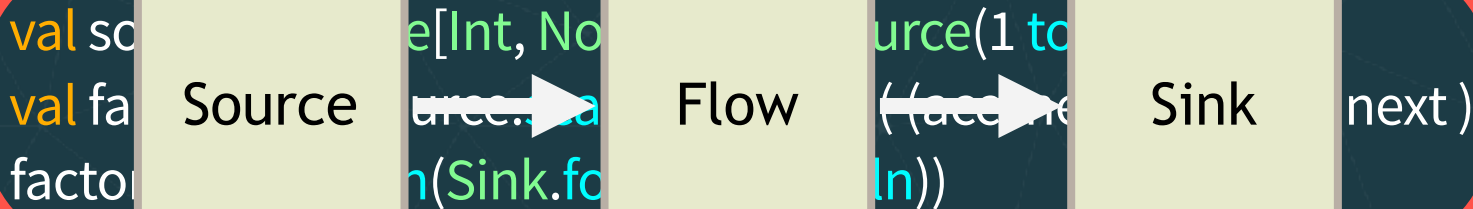
Output to a Sink,  
and run it



```
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()
```

A source, flow, and sink constitute a graph



# akka streams

- This example is included in the project:
  - akkaStreamsModelServer/simple-akka-streams-example.sc
- To run it (showing the different prompt!):

```
$ sbt
```

```
sbt:akkaKafkaTutorial> project akkaStreamsModelServer
```

```
sbt:akkaStreamsModelServer> console
```

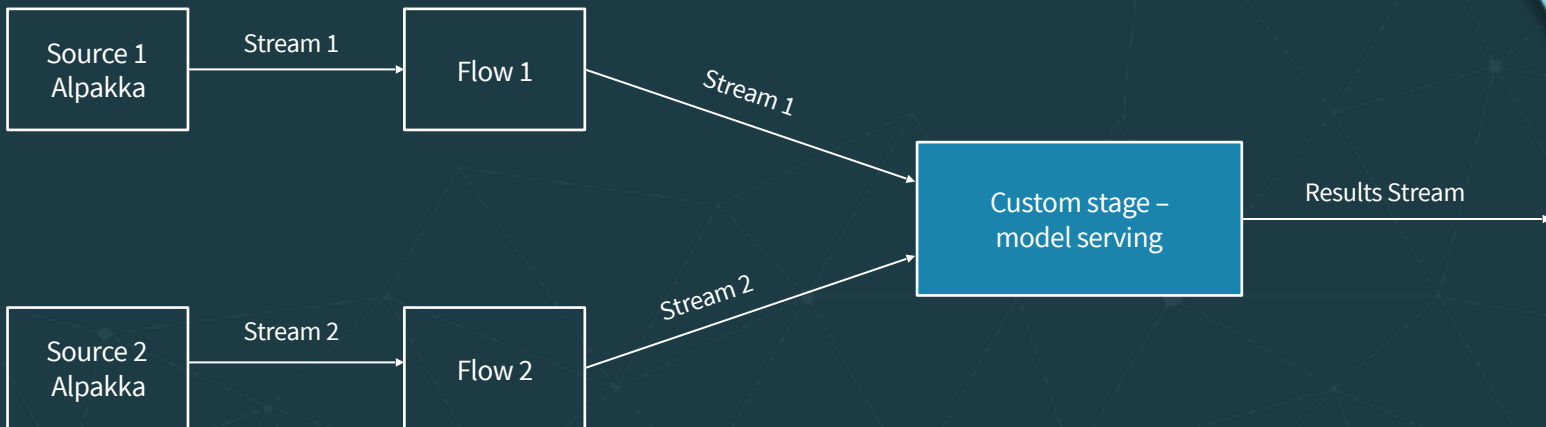
```
scala> :load akkaStreamsModelServer/simple-akka-streams-example.sc
```

# Implementations

- How do we integrate model serving (or any other new capability) into an Akka Streams app?
  - Implement a *Custom Stage*. Once implemented, you use it like any other “step” in the Akka Streams app.
  - Make asynchronous calls to Akka Actors to do anything you want...

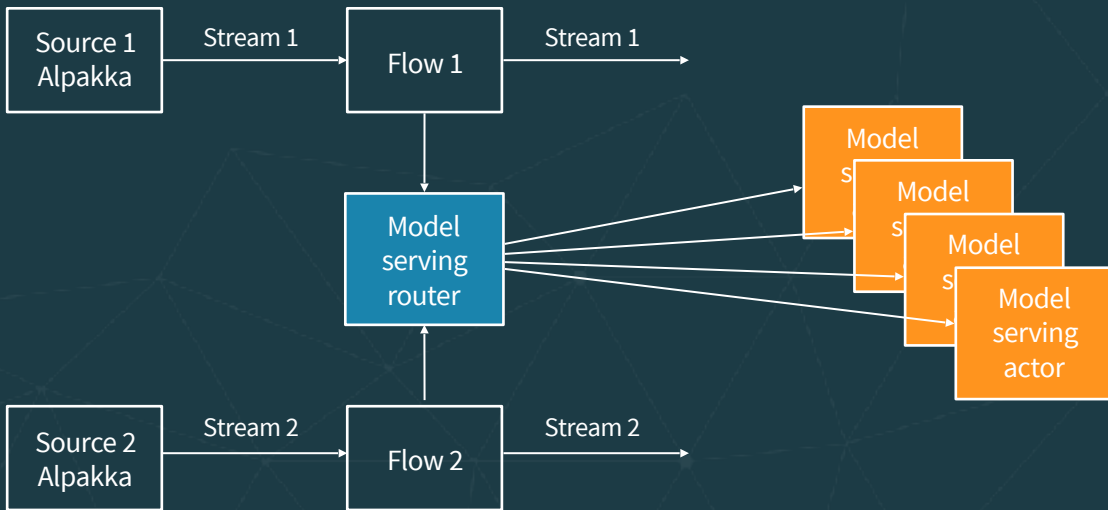
# Using Custom Stage

Create a custom stage, a fully type-safe way to encapsulate new functionality. Like adding a new “operator”.



# Using Akka Actors

Use a router actor to forward requests to the actor(s) responsible for processing requests for a specific model type. Clone for scalability!!



# Akka Streams Example

## Code time

1. Run the *client* project (if not already running)
2. Explore and run *akkaStreamsModelServer* project
  1. Use the `c` or `custom` (or default) command-line argument for the *custom stage*
  2. Use the `a` or `actor` command-line argument for the *actor model server*
  3. Use `-h` or `-help` for help

# Exercises!

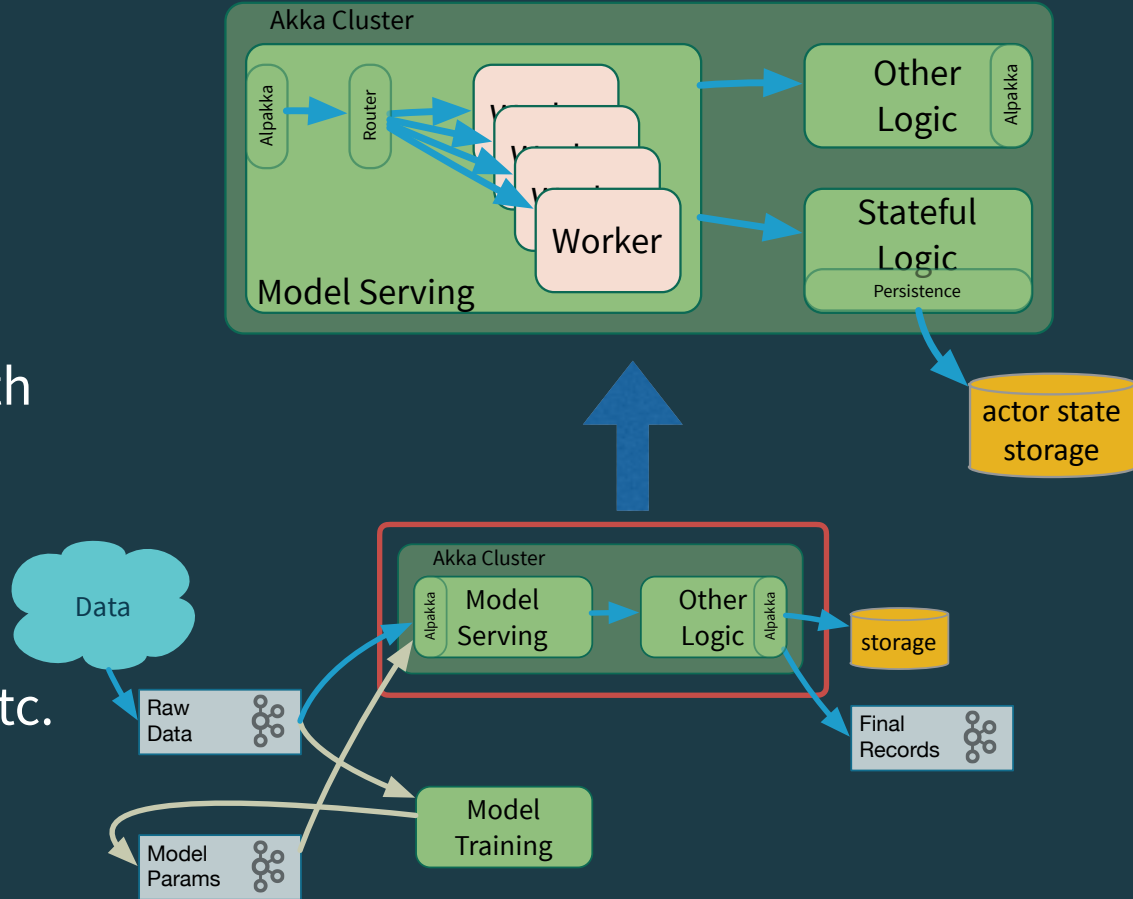
We've prepared some exercises. We may not have time during the tutorial to work on them, but take a look at the *exercise* branch in the Git project (or the separate X.Y.Z\_exercise download).

To find them, search for “// Exercise”. The *master* branch implements the solutions for most of them.

# Other Production Concerns



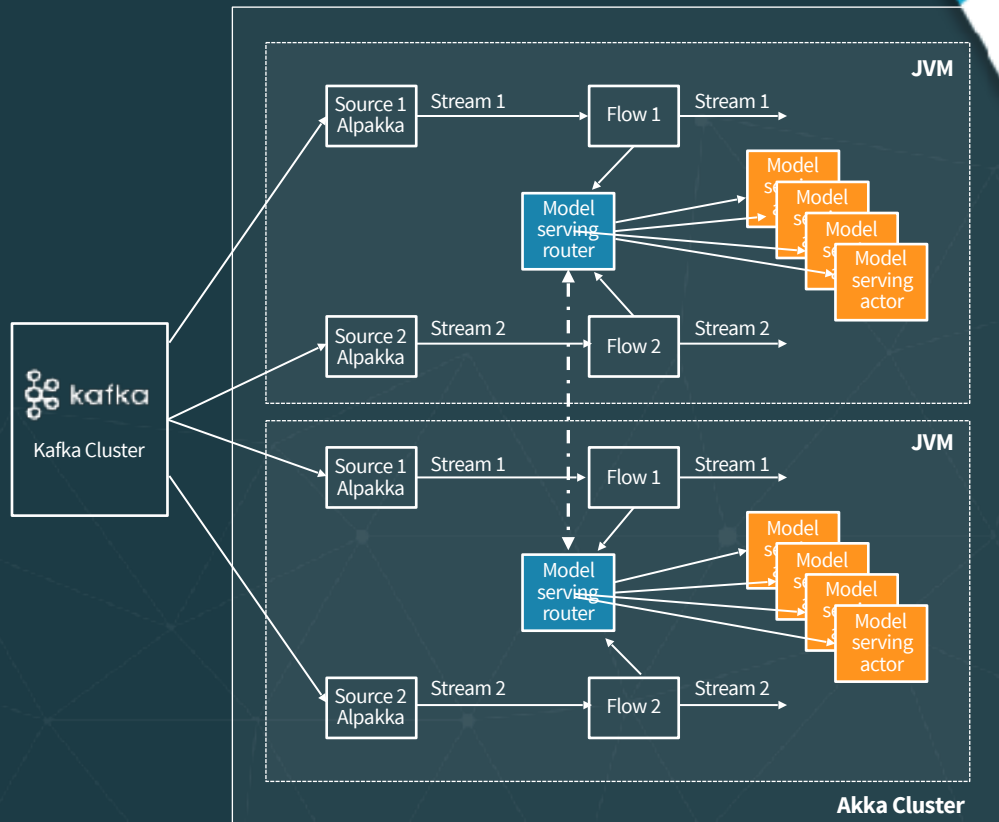
- Scale scoring with workers and routers, across a cluster
- Persist actor state with Akka Persistence
- Connect to *almost* anything with Alpakka
- *Lightbend Enterprise Suite*
  - for production monitoring, etc.



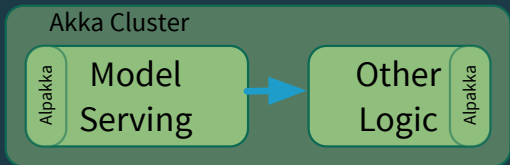
# Using Akka Cluster

Two levels of scalability:

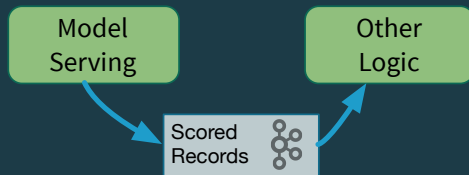
- Kafka partitioned topic allow to scale listeners according to the amount of partitions.
- Akka cluster sharing allows to split model serving actors across clusters.



# Go Direct or Through Kafka?



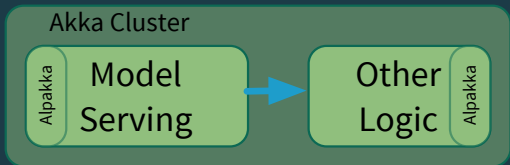
vs.



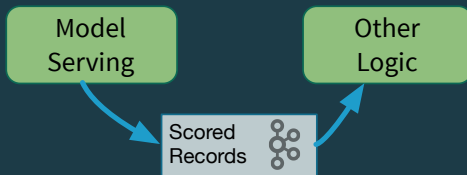
- Extremely low latency
- Minimal I/O and memory overhead
- No marshaling overhead

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

# 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

# Kafka Streams



# Kafka Streams

- Important stream-processing concepts, e.g.,
  - Distinguish between *event time* and *processing time*
  - Windowing support.
  - For more on these concepts, see
    - [Dean's book](#) ;)
    - [Talks, blog posts, writing by Tyler Akidau](#)



# Kafka Streams

- KStream - per-record transformations
- KTable - key/value store of supplemental data
  - Efficient management of application state



# 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

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



# Kafka Streams

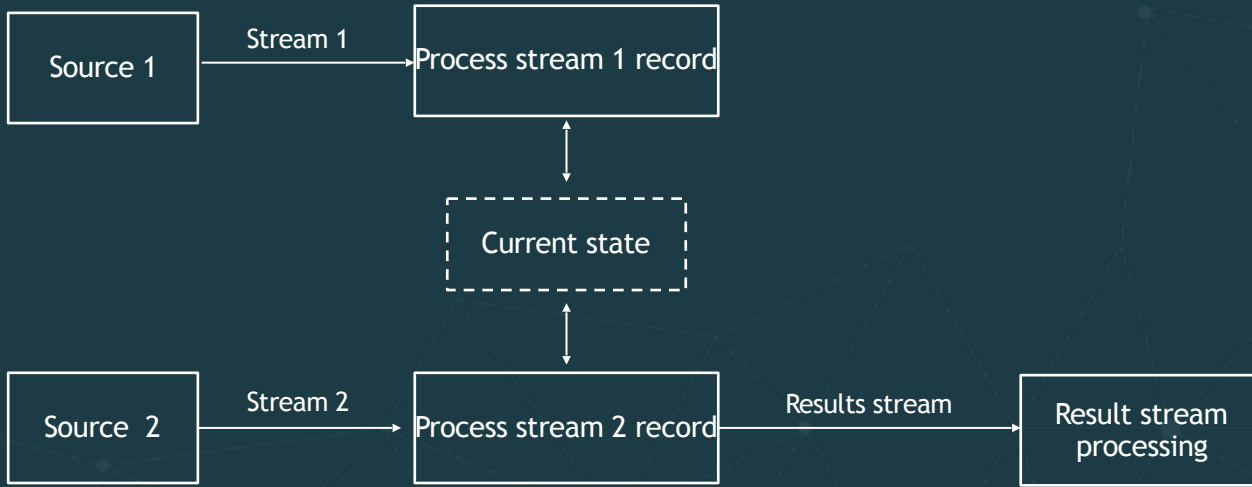
- Provides a Java API
- Lightbend donated a Scala API to Apache Kafka
  - <https://github.com/lightbend/kafka-streams-scala>
  - See also our convenience tools for distributed, queryable state: <https://github.com/lightbend/kafka-streams-query>
- SQL - yes, but requires a specialized application (i.e., not a library like in Spark)



# Kafka Streams

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

# Model Serving With Kafka Streams



# State Store Options We'll Explore

- “Naive”, in memory store (no durability!)
  - Also uses the KS Processor Topology API
- Built-in key/value store provided by Kafka Streams
  - Uses the KS DSL
- Custom store
  - Also uses the DSL

# Model Serving With Kafka Streams

## Code time

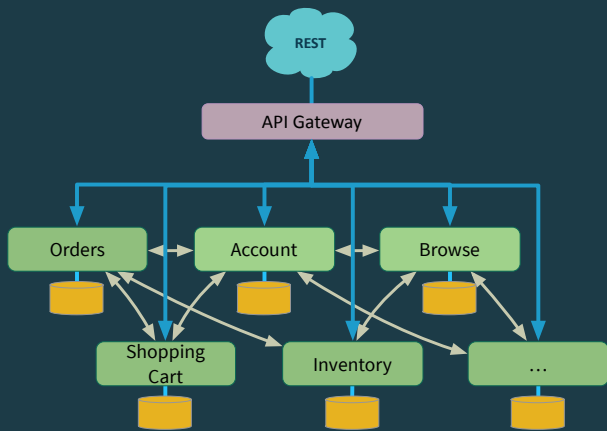
1. Run the *client* project (if not already running)
2. Explore and run *kafkaStreamsModelServer* project
  1. Use the `c` or `custom` (or default) command-line argument for the *custom state store*
  2. Use the `s` or `standard` command-line argument for the KS built-in *standard store*
  3. Use the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

# Wrapping Up

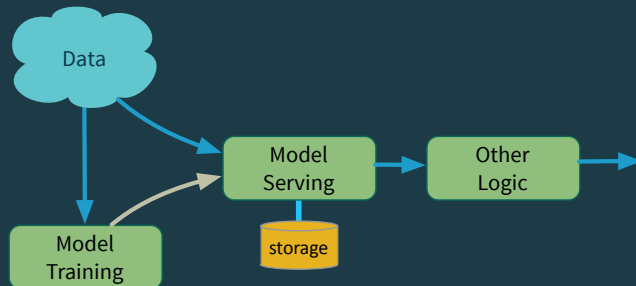
# To Wrap Up



Event-driven  $\mu$ -services



“Record-centric”  $\mu$ -services



Events

Records



# In Our Remaining Time Today...

1. Try the exercises in the exercise branch (or the X.Y.Z\_exercise
  - Search for “// Exercise” in the code
2. Explore the code we didn't discuss (a lot ;)
3. Ask us for help on anything now...
4. Visit [lightbend.com/fast-data-platform](https://lightbend.com/fast-data-platform)
5. Profit!!

# Thank You

- Executive Briefing: What you need to know about fast data (Dean)
  - 14:55–15:35 Wednesday, 23 May 2018, Capital Suite 17
- Meet the Expert (Dean)
  - Thursday 11:50 - 12:30, O'Reilly Booth, Expo Hall
- AMA, Streaming Applications and Architectures (Boris and Dean)
  - 14:05–14:45 Thursday, 24 May 2018, Capital Suite 14

## Questions?

And don't miss:

- Kafka in jail: Running Kafka in container-orchestrated clusters (Sean Glover)
  - 16:35–17:15 Wednesday, 23 May 2018, Capital Suite 8/9
- Processing fast data with Apache Spark: A tale of two APIs (Gerard Maas)
  - 11:15–11:55 Wednesday, 23 May 2018, Capital Suite 8/9
- Machine-learned model quality monitoring in fast data and streaming applications (Emre Velipasaoglu)
  - 14:55–15:35 Wednesday, 23 May 2018, Expo Hall

[lightbend.com/products/fast-data-platform](https://lightbend.com/products/fast-data-platform)

[boris.lublinsky@lightbend.com](mailto:boris.lublinsky@lightbend.com)

[dean.wampler@lightbend.com](mailto:dean.wampler@lightbend.com)