

Building Kafka-based Microservices with Akka Streams and Kafka Streams

Strata Data NYC 2018

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.

- 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

But first, introductions...



If you have not done this already, download the tutorial from GitHub

<https://github.com/lightbend/kafka-with akka-streams-kafka-streams-tutorial>

These slides are in the



Why Streaming?

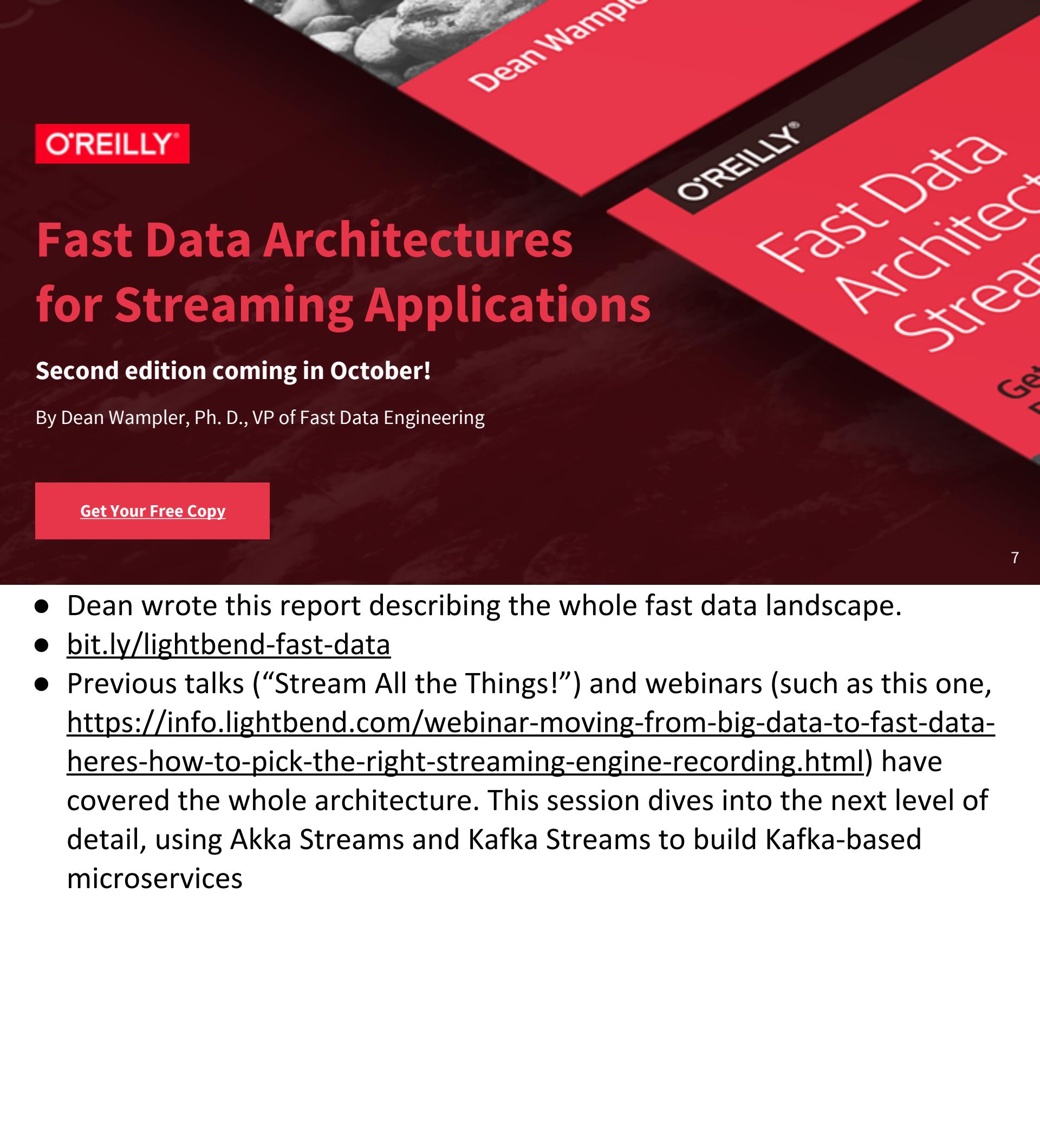
“We live as streams, but we have a tendency to think in batch. Batch might be faster (simpler), but the reality is streams”

— Fabio Yamada, Kafka Mailing List

About Streaming Architectures

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





O'REILLY®

Dean Wampler

O'REILLY®

Fast Data Architectures for Streaming Applications

Second edition coming in October!

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

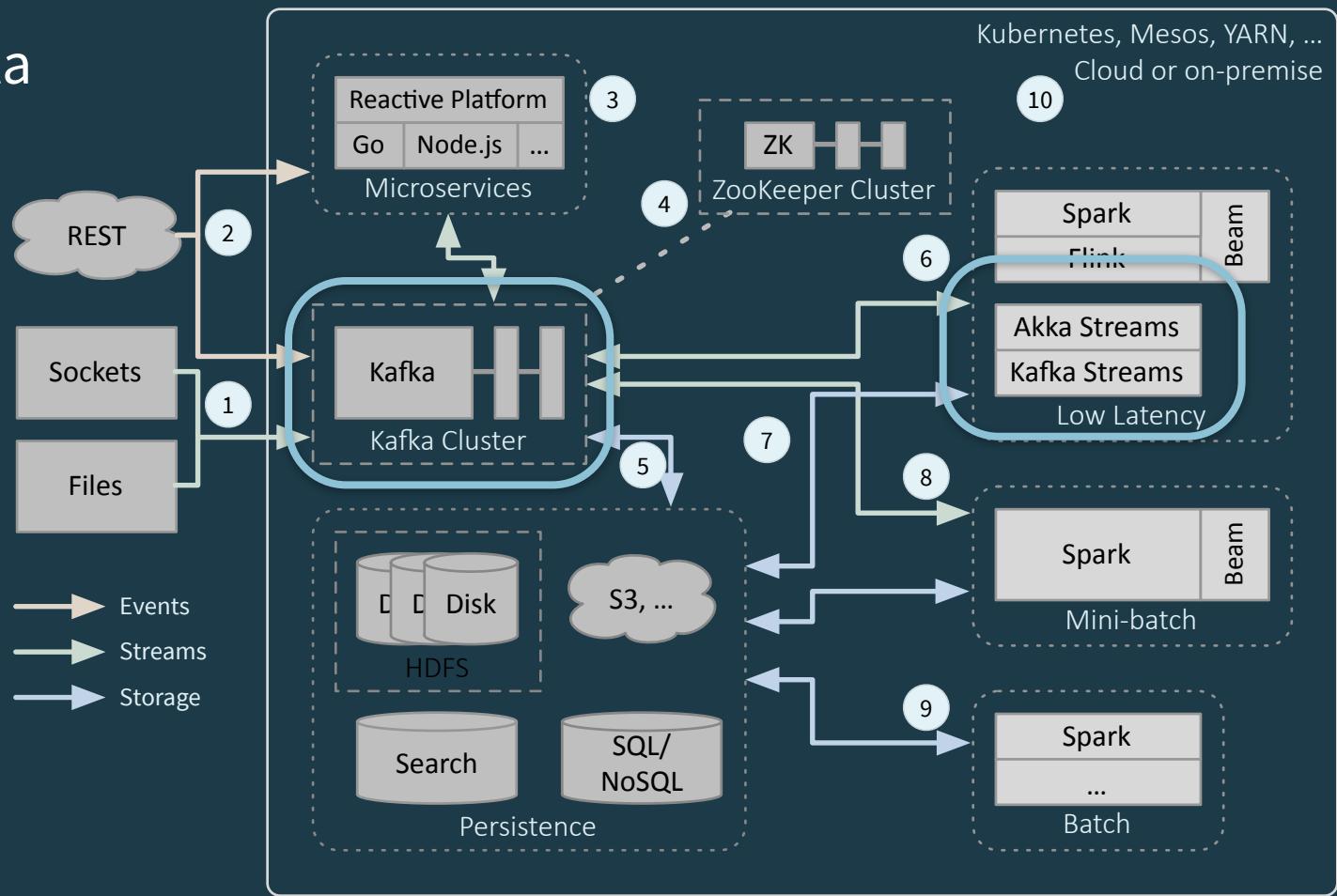
[Get Your Free Copy](#)

7

- 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

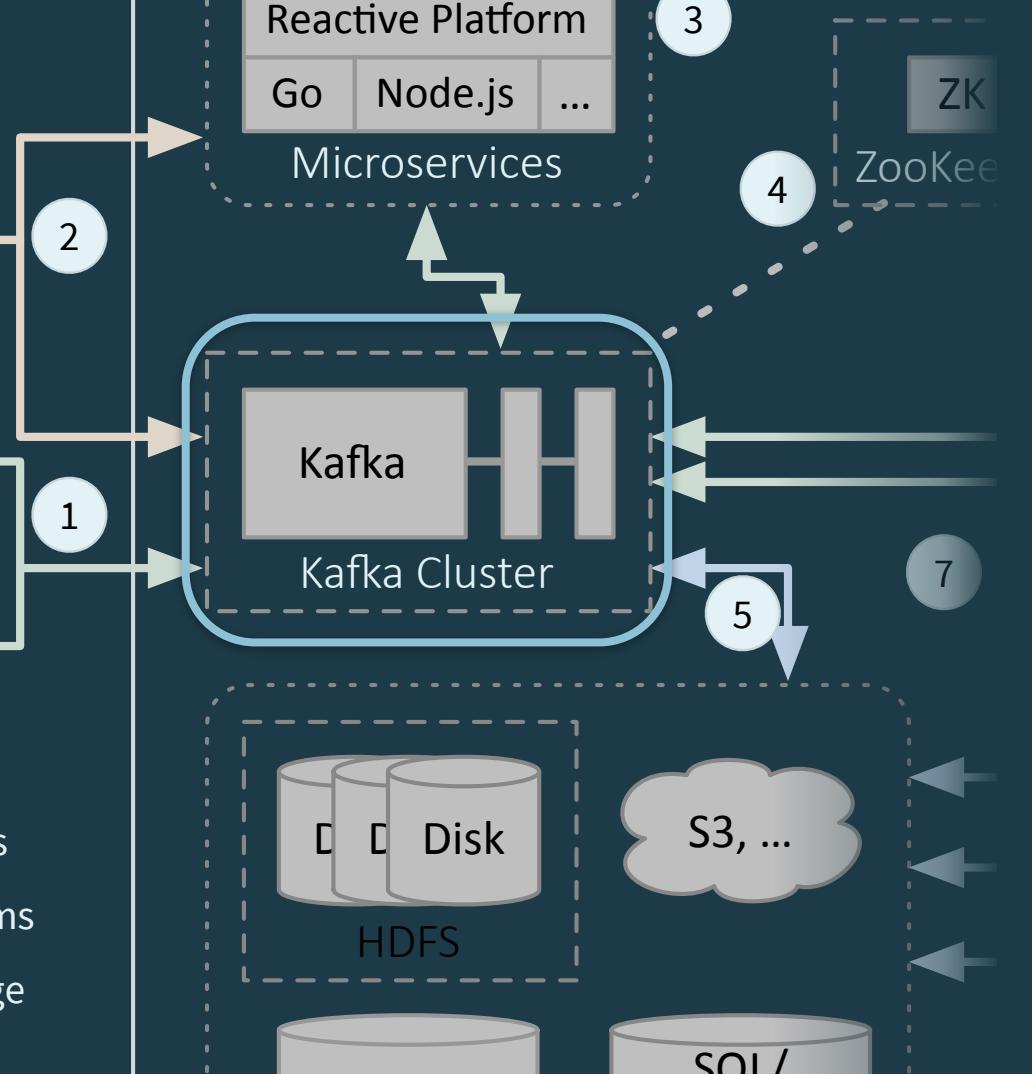
Today's focus:

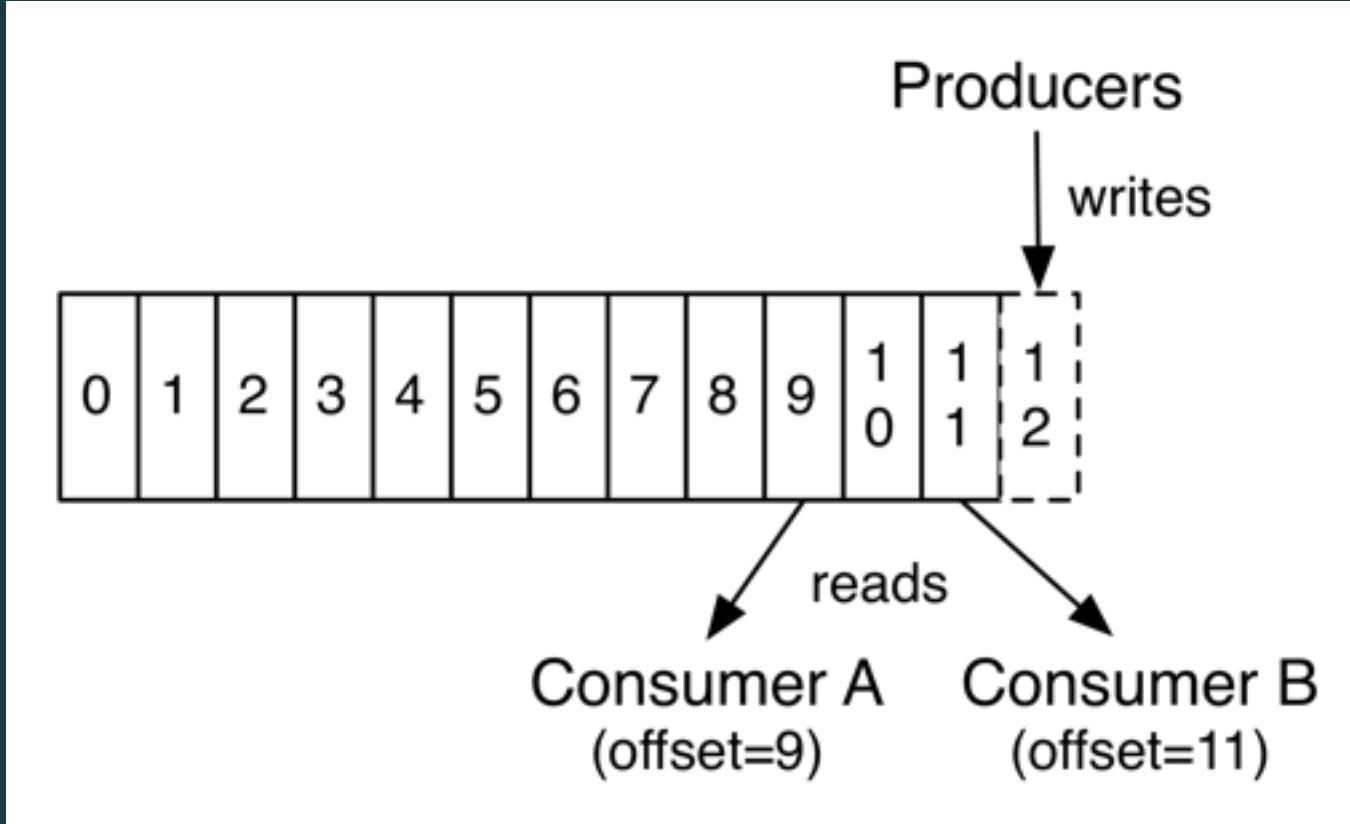
- Kafka - the data backplane
- Akka Streams and Kafka Streams - streaming microservices



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.

Why Kafka?





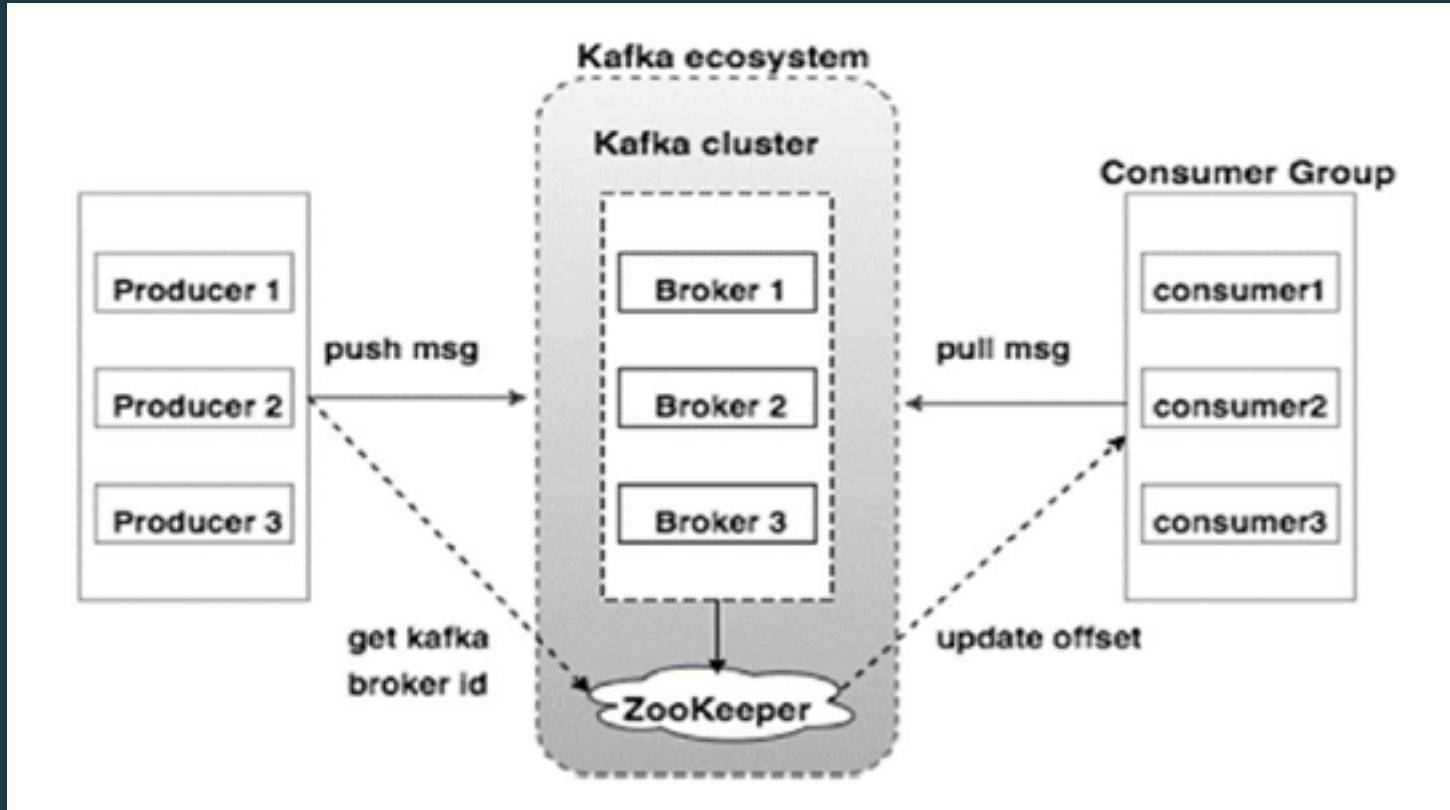
Kafka is a distributed log, storing messages sequentially. Producers always write to the end of the log, consumers can read on the log offset that they want to read from (earliest, latest, ...)

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

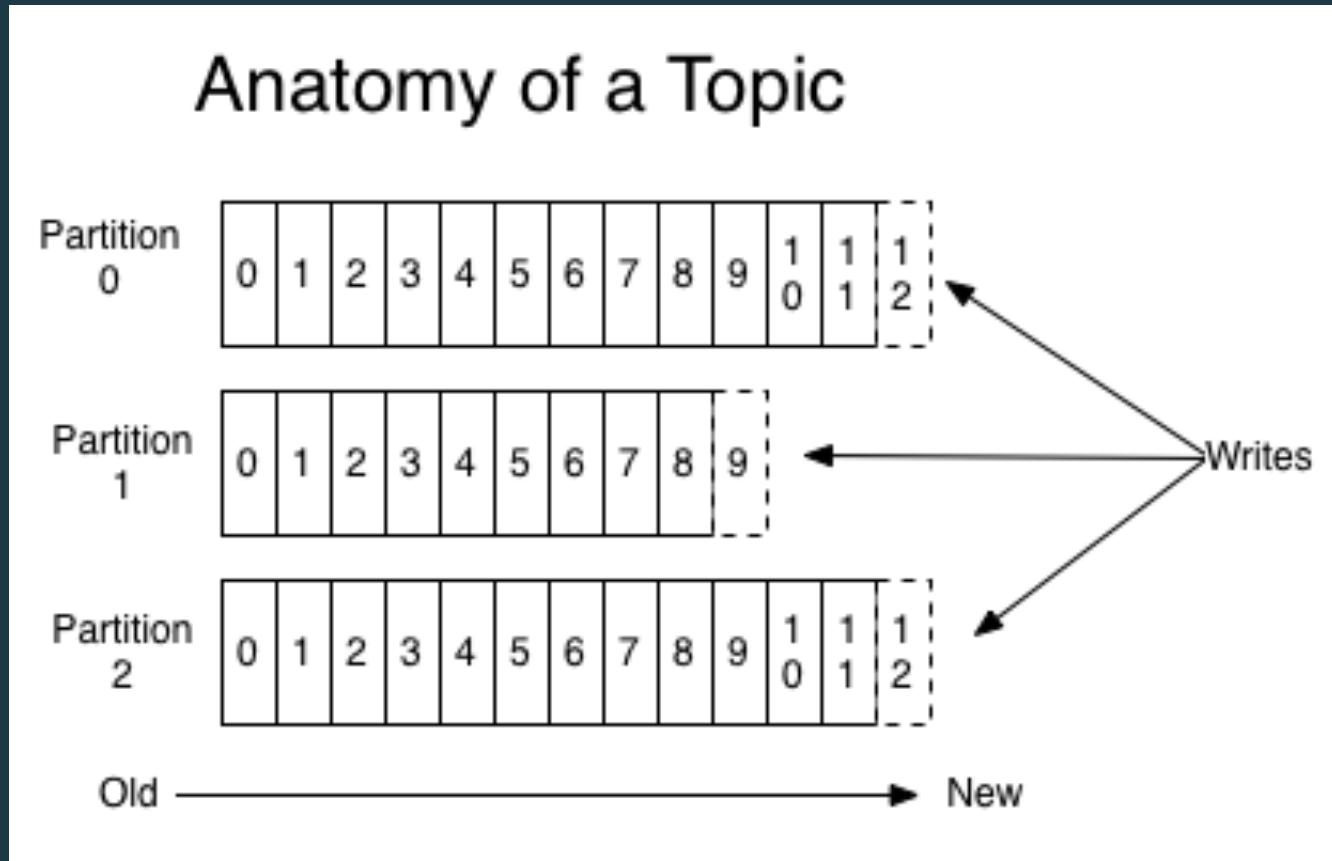
Alternatives to Kafka include Pravega (EMC) and Distributed Log/Pulsar (Apache)



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.

A Topic and Its Partitions



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.

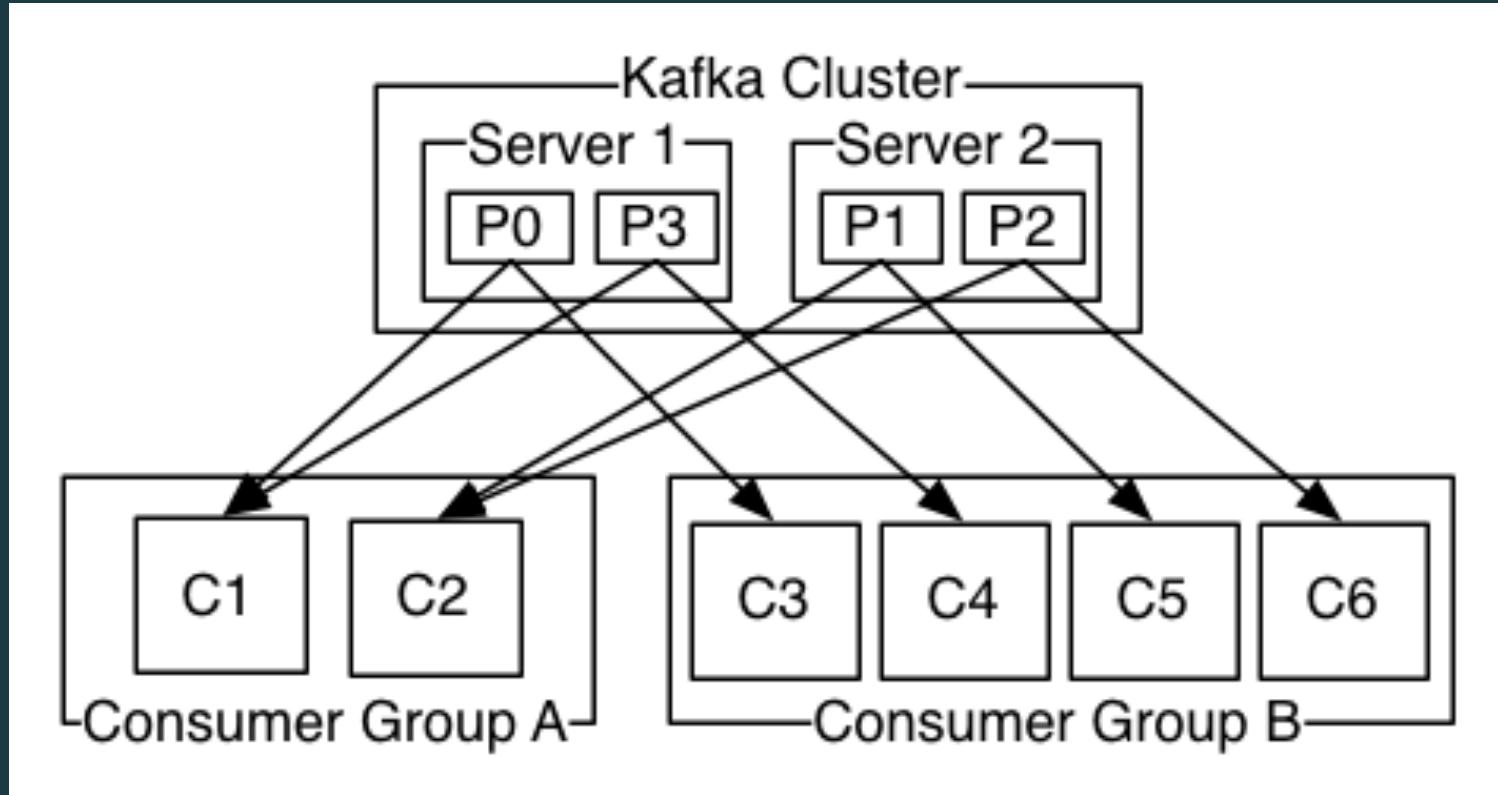
So, a topic is a logical aggregation of partitions. A topic doesn't provide any sequential guarantee (except a one-partition topic, where it's "accidental").

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.

Consumer Groups



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.

Kafka Producers and Consumers



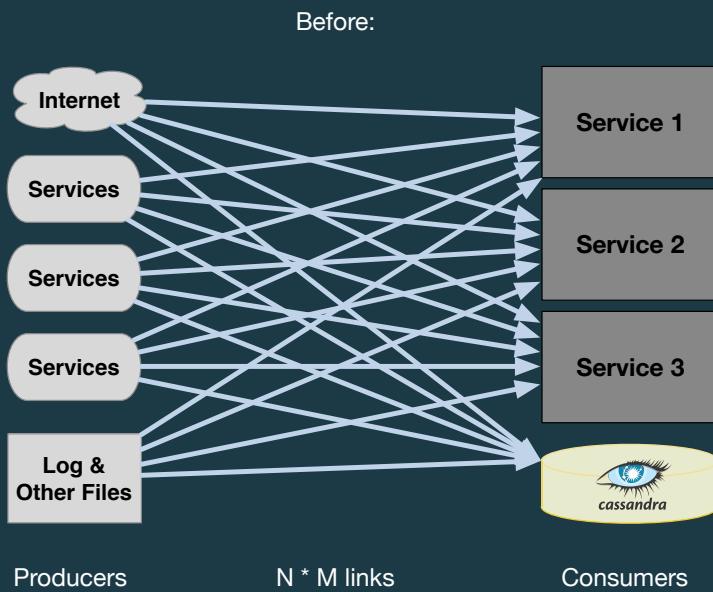
Code time

1. Project overview
2. Explore and run the *client* project

- Creates in-memory (“embedded”) Kafka instance and our topics
- Pumps data into them

We'll walk through the whole project, to get the lay of the land, then look at the client piece. The embedded Kafka approach is suitable for non-production scenarios only, like learning ;)

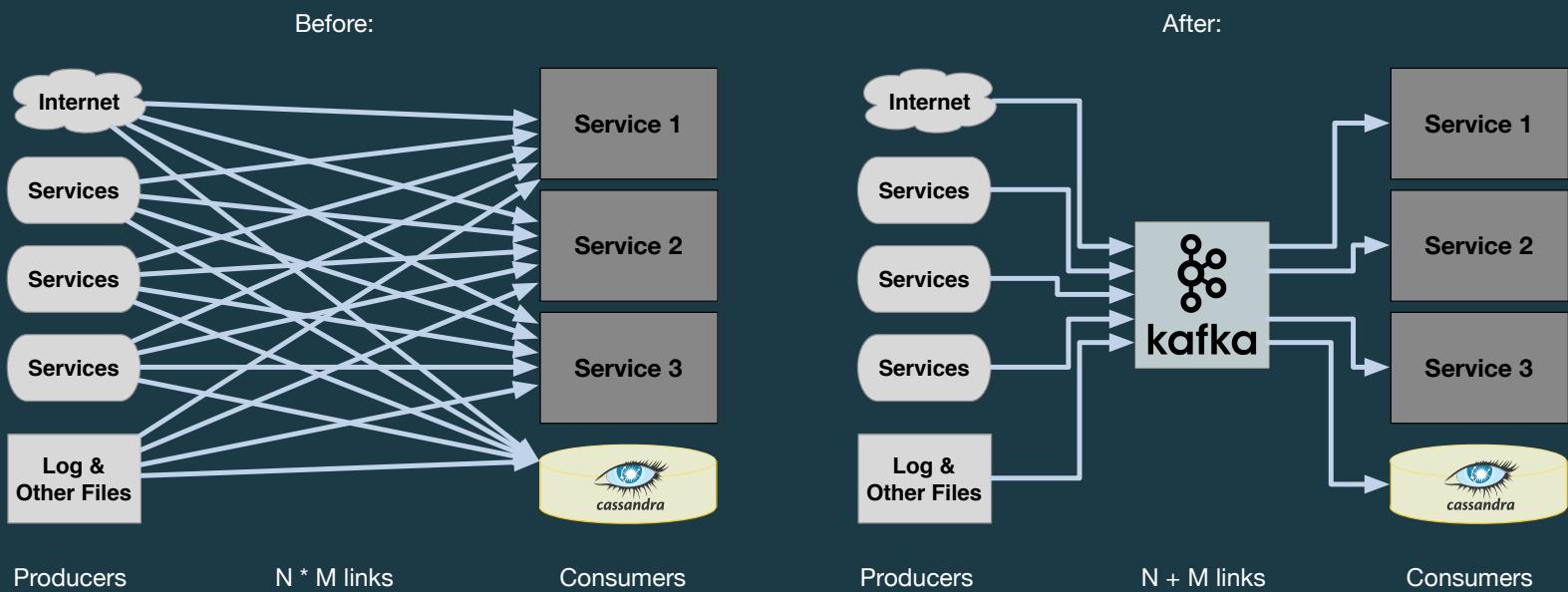
Architecture Benefits of Kafka



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

Architecture Benefits of Kafka

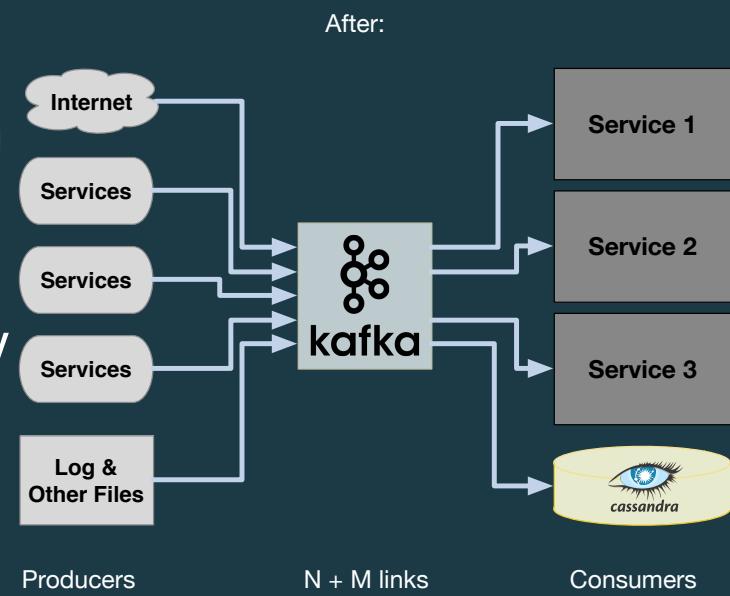


Kafka can simplify the situation by providing a single backbone which is used by all services (there are of coarse topics, but they are more logical then physical connections). Additionally Kafka persistence provides robustness when a service crashes (data is captured safely, waiting for the service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

Architecture Benefits of Kafka

Kafka:

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

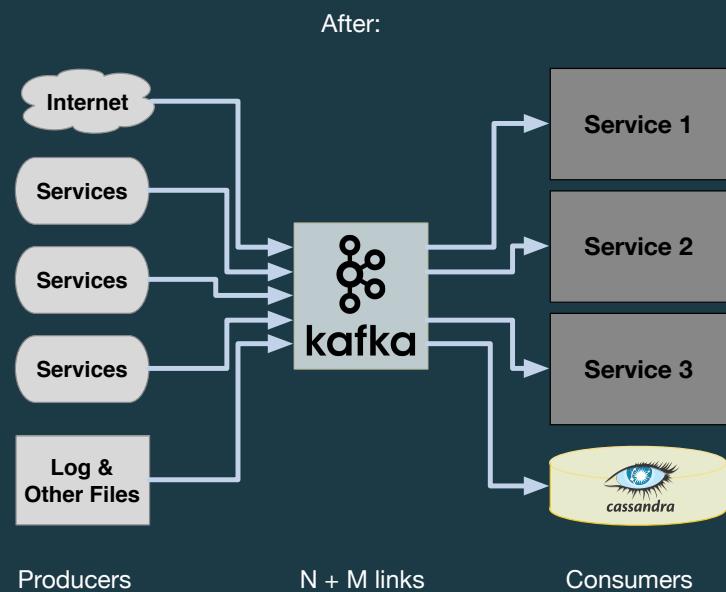


Kafka can significantly improve decoupling (no service specific endpoints, temporal decoupling). It minimizes the amount of data sent over network, each producer writes data to Kafka, instead of writing it to multiple consumers. This also improves data consistency - the same data is consumed by all consumers.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

Architecture Benefits of Kafka

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.

Kafka Message size considerations

- Should I use Kafka for all messages?
 - Best Kafka performance is with messages size in the order of a few KB. Larger messages put heavy load on brokers and is very inefficient. It is inefficient on producers and consumers as well.
- What if my messages are very large?
 - Consider using messaging by reference - store a message in S3, HDFS, etc and send the reference to the location via Kafka

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?

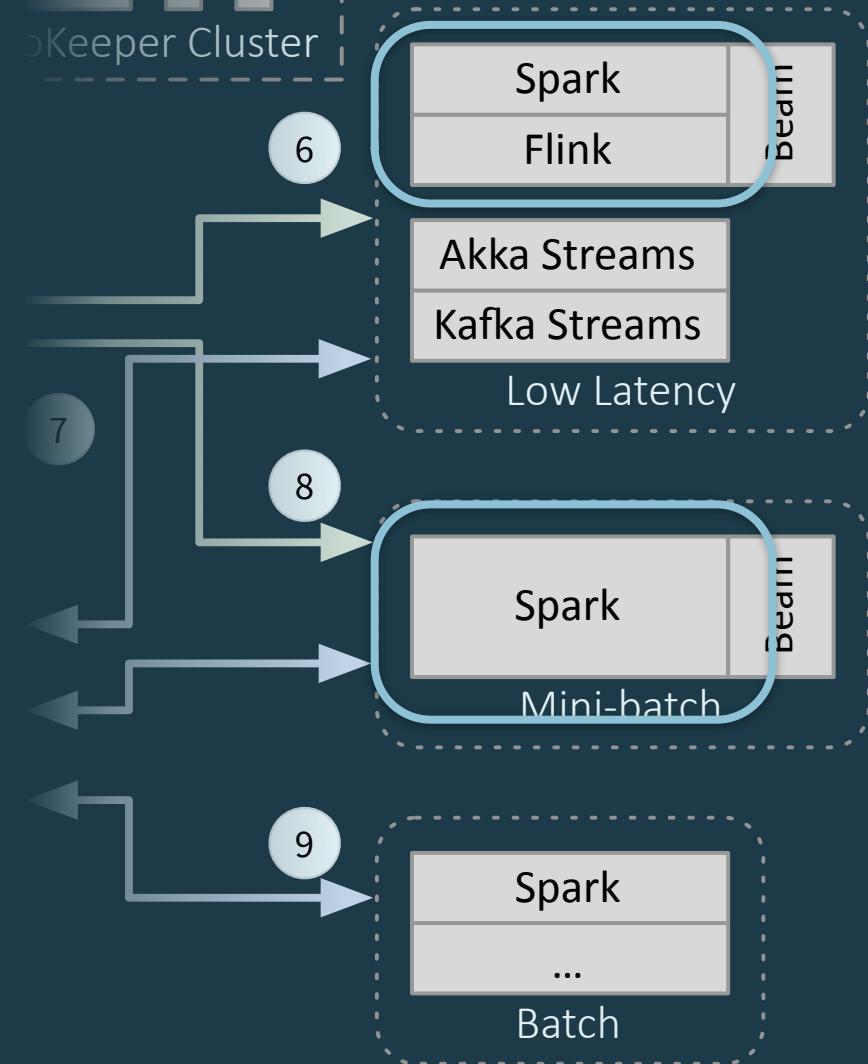
Streaming Architectures

Two options:

- Stream processing engines
- Streaming libraries

Streaming Engines:

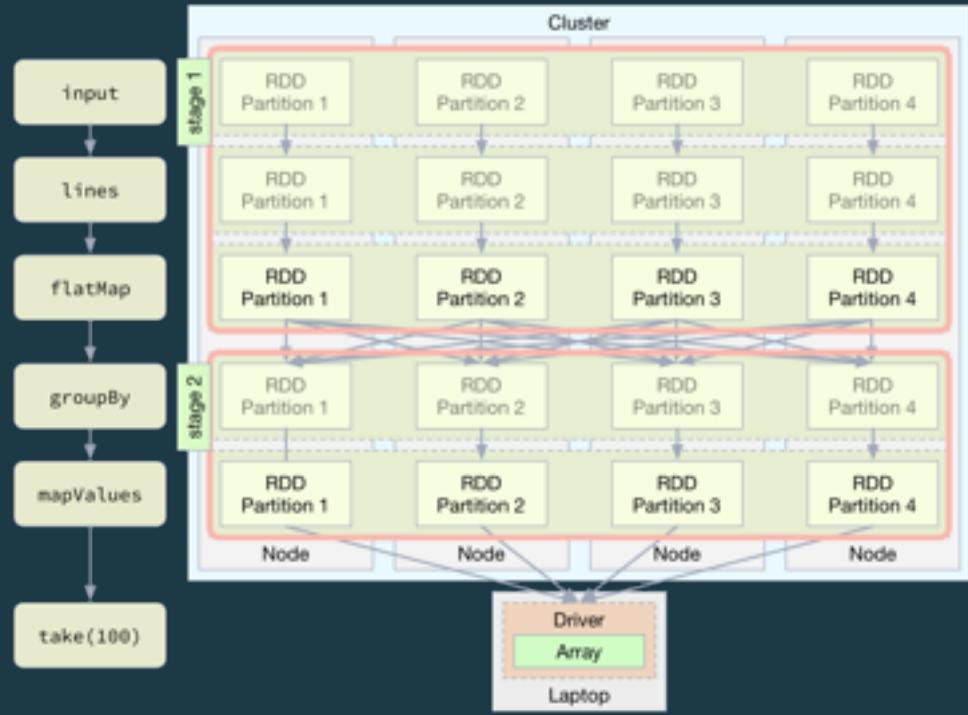
Spark, Flink - services to which you submit work. Large scale, automatic data partitioning.



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:

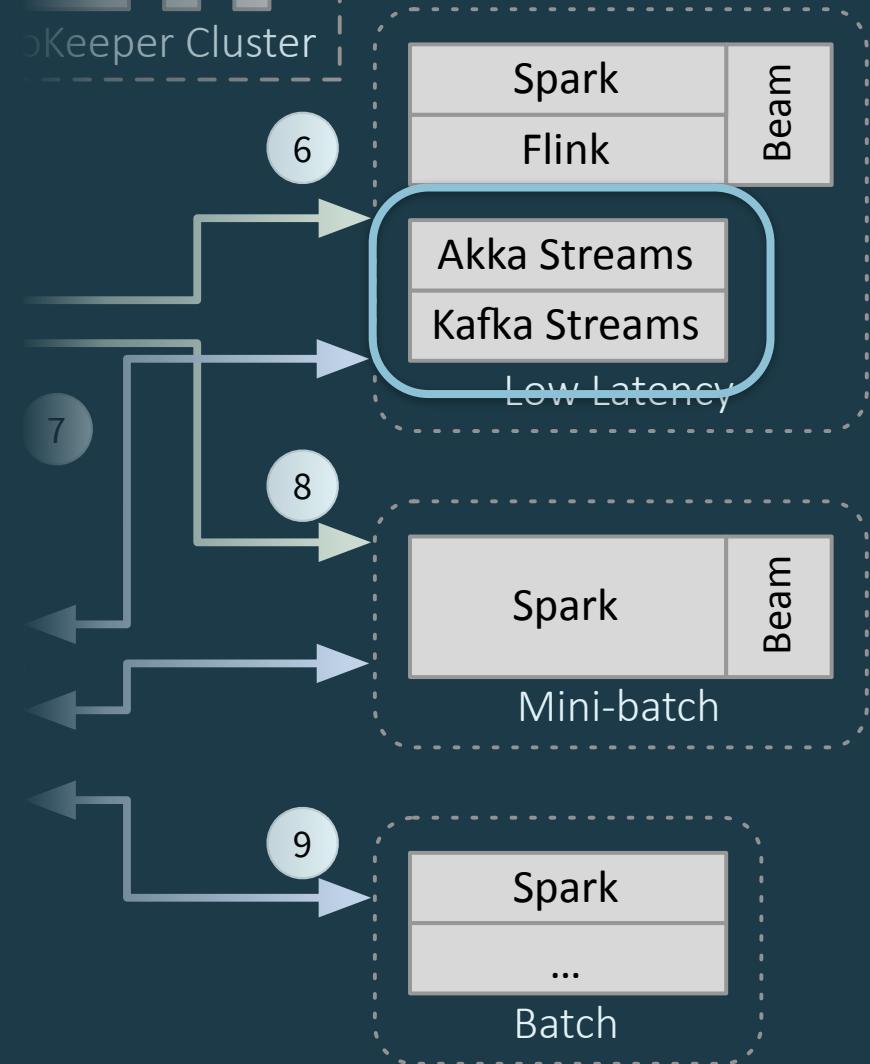
Spark, Flink - services to which you submit work. Large scale, automatic data partitioning.



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!

Streaming Libraries:

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



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 interoperability capabilities.

Machine Learning and Model Serving: A Quick Introduction



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

The O'Reilly logo, featuring the word "OREILLY" in white capital letters on a red background.A thumbnail image of a book cover titled "Serving Machine Learning Models" by Boris Lublinsky, published by O'Reilly.A thumbnail image of a book cover titled "Serving Machine Learning Models" by Boris Lublinsky, published by O'Reilly.

Serving Machine Learning Models

A Guide to Architecture, Stream Processing Engines, and Frameworks

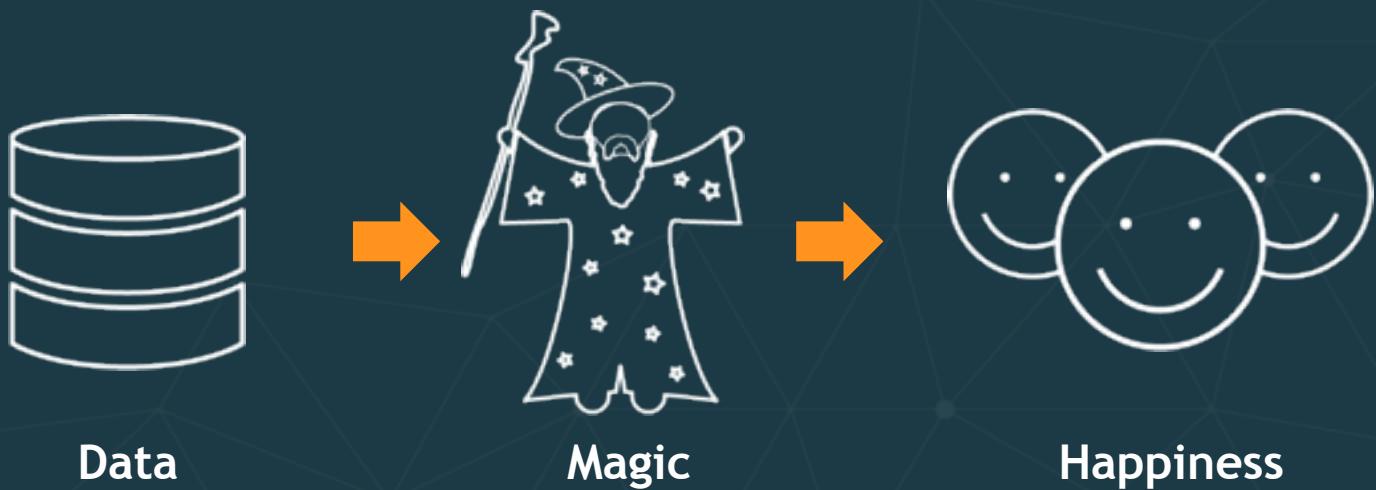
By Boris Lublinsky, Fast Data Platform Architect

[Get Your Free Copy](#)

27

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

ML Is Simple



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

Maybe Not



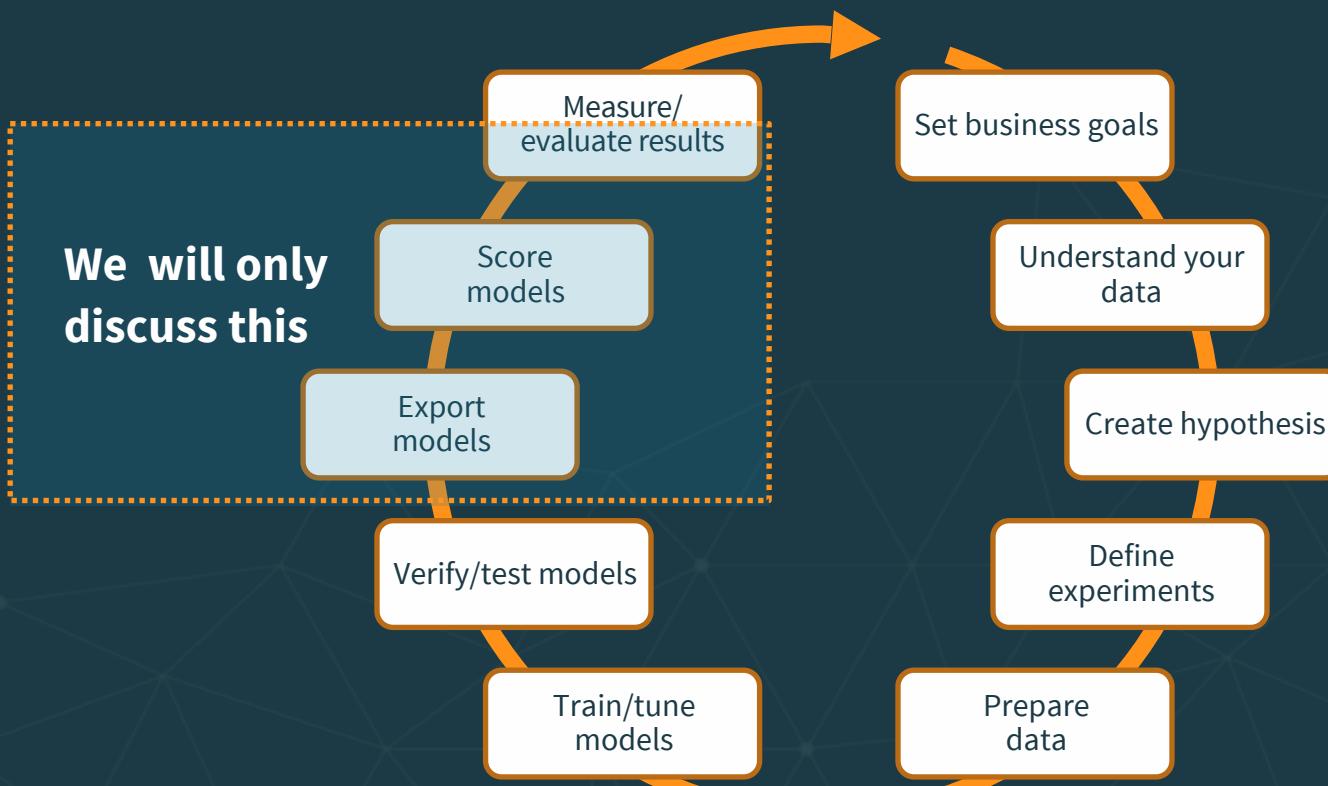
Not only the climb is steep, but you are not sure which peak to climb
Court of the Patriarchs at Zion National park

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

The Reality



What Is The Model?

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

for example:

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

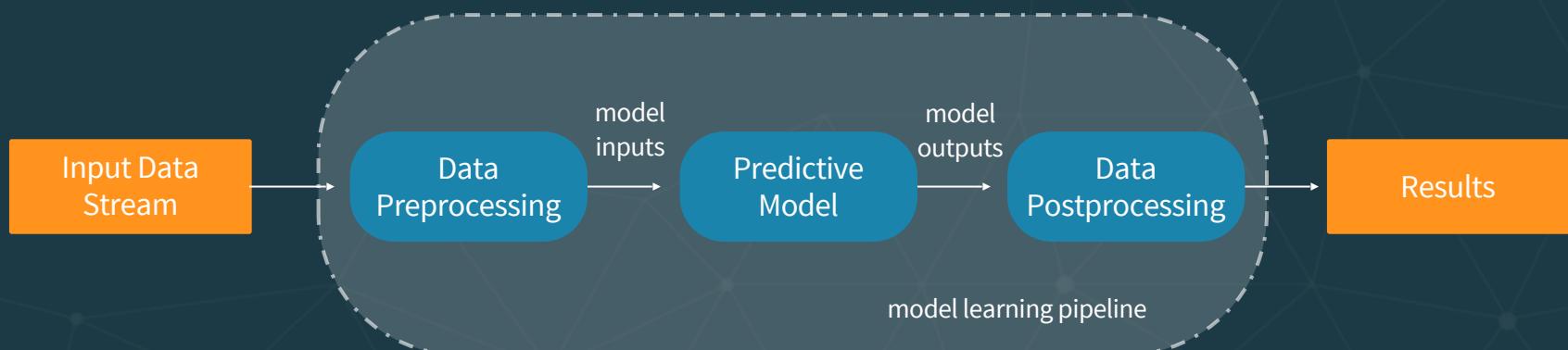
Neural network: $f(x) = K(\sum_i w_i g_i(x))$

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.



33

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.

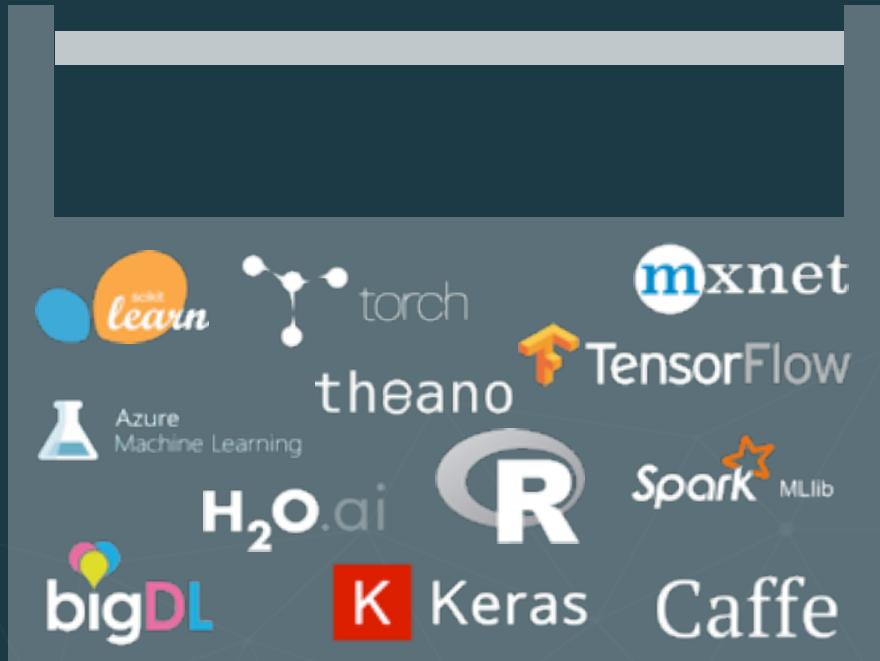
This notion of machine learning pipelines has been adopted by many

Traditional Approach to Model Serving

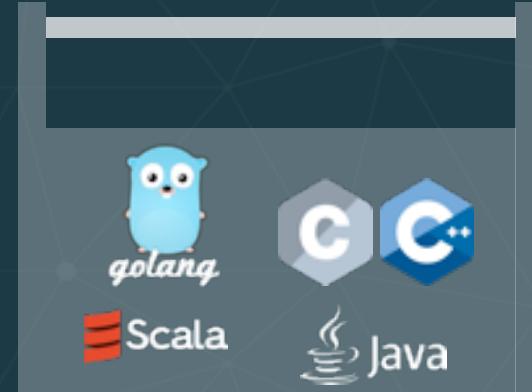
- Model is code
- This code has to be saved and then somehow imported into model serving

Why is this problematic?

Impedance Mismatch



Continually expanding
Data Scientist toolbox

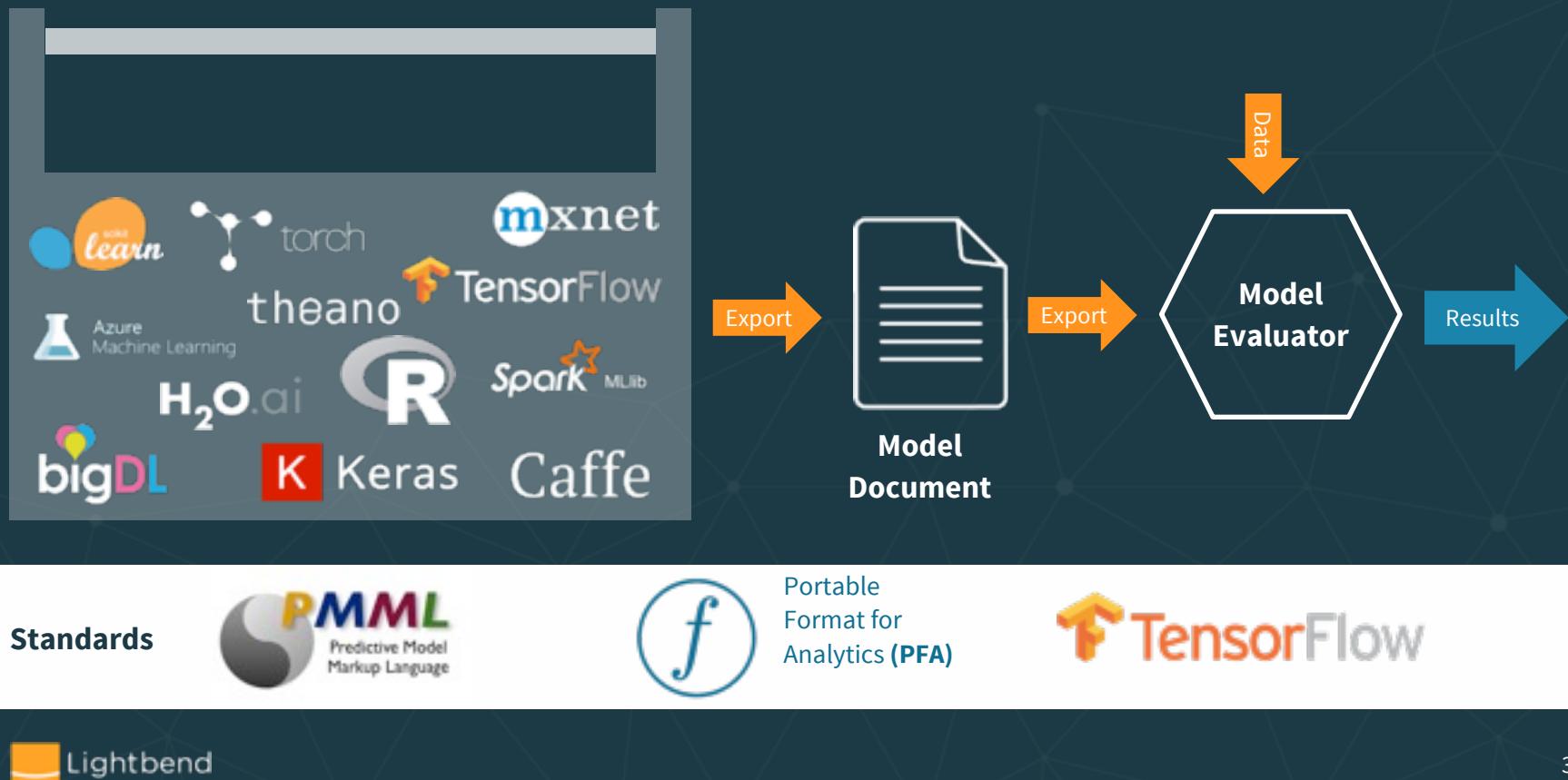


Defined Software
Engineer toolbox

35

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.

Alternative - Model As Data



36

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 - JPMMML 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



<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.



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

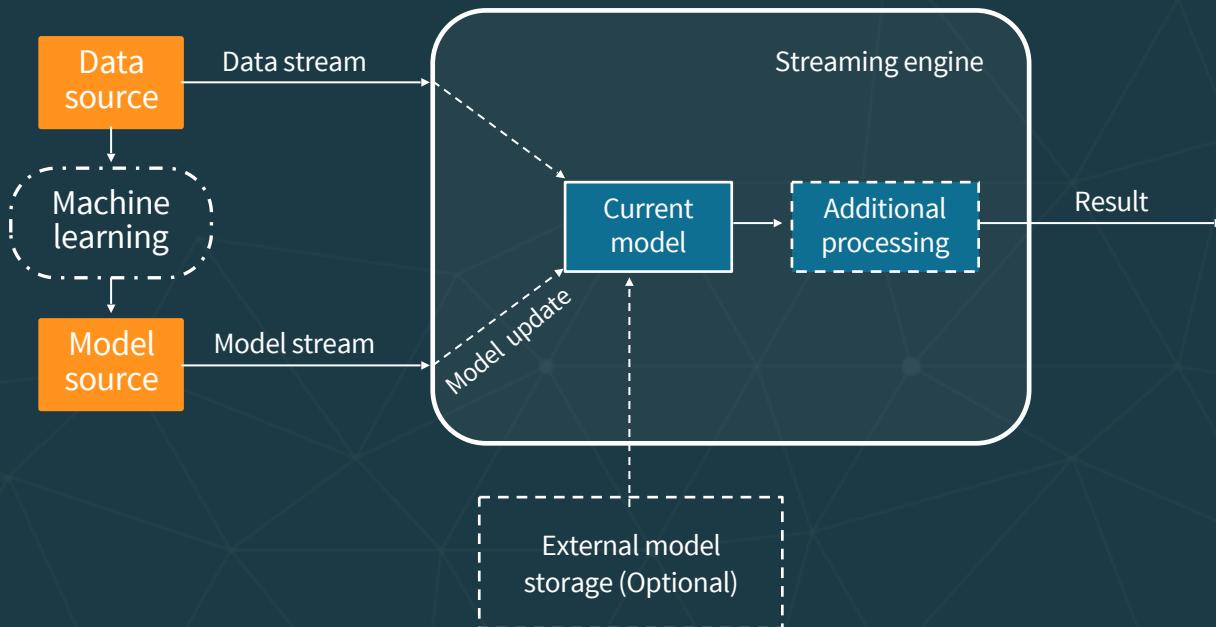
Additional Considerations – Model Lifecycle

- 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).



42

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
        TENSORFLOW = 0;
        TENSORFLAWSAVED = 2;
        PMML = 2;
    };
}

oneof MessageContent {
    bytes data = 5;
    string location = 6;
}
```



43

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 Representation (Scala)

```
trait Model {  
    def score(input: Any) : Any  
    def cleanup() : Unit  
    def toBytes() : Array[Byte]  
    def getType : Long  
}  
  
def ModelFactory1 {  
    def create(input : ModelDescriptor) : Model  
    def restore(bytes : Array[Byte]) : Model  
}
```

Corresponding Scala code that can be generated from the description.

Side Note: Monitoring

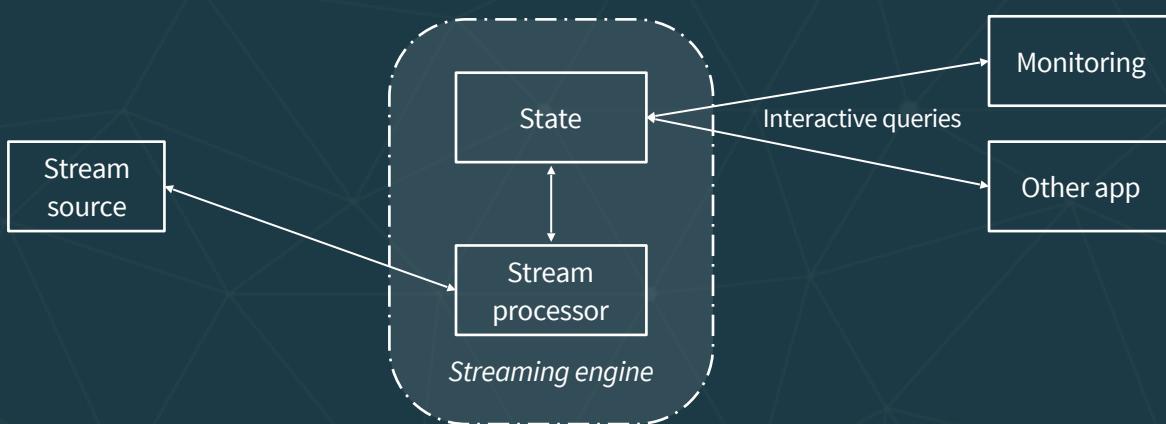
Model monitoring should provide information about usage, behavior, performance and lifecycle of the deployed models

```
case class ModelToServeStats(  
    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.



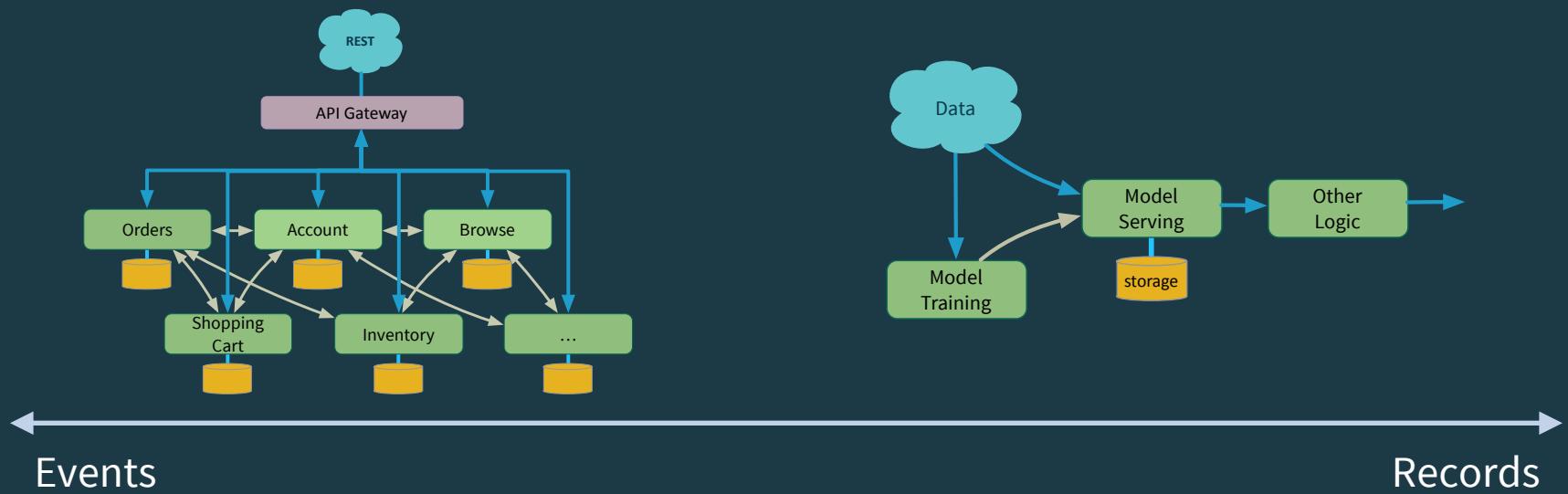
Kafka Streams and Flink have built-in support for this and its being added to Spark Streaming. We'll show how to use other Akka features to provide the same ability in a straightforward way for Akka Streams.

Microservice All the Things!



A Spectrum of

Event-driven μ-services



"Record-centric" μ-services

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.

A Spectrum of



Event-driven μ-services



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

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

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.

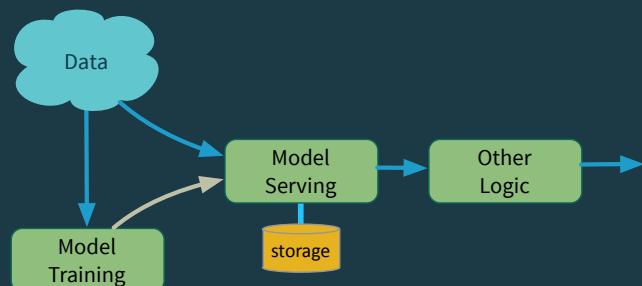
A Spectrum of



Emerged from the right-hand side.

Kafka Streams pushes to the left, supporting many event-

“Record-centric” μ-services



Events

Records

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.

Akka Streams



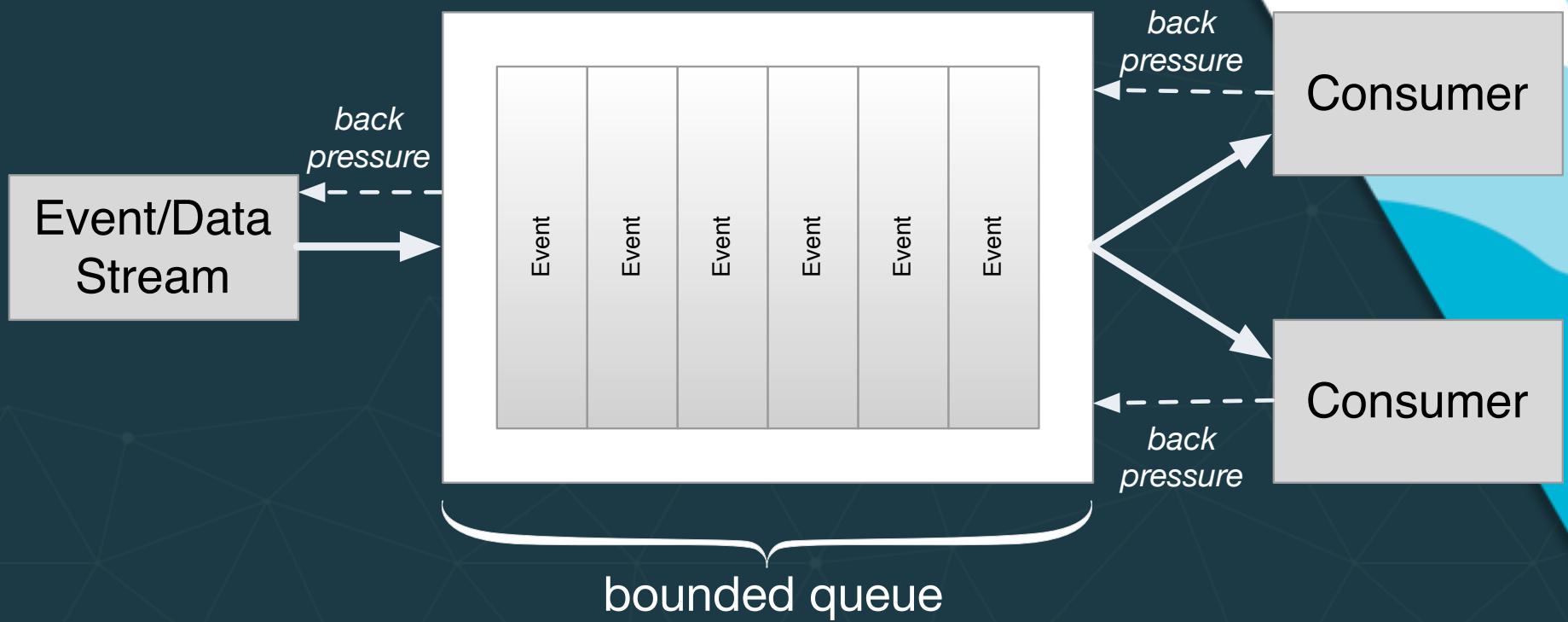


- A *library*
- Implements Reactive Streams.
 - <http://www.reactive-streams.org/>
 - *Back pressure* for flow control

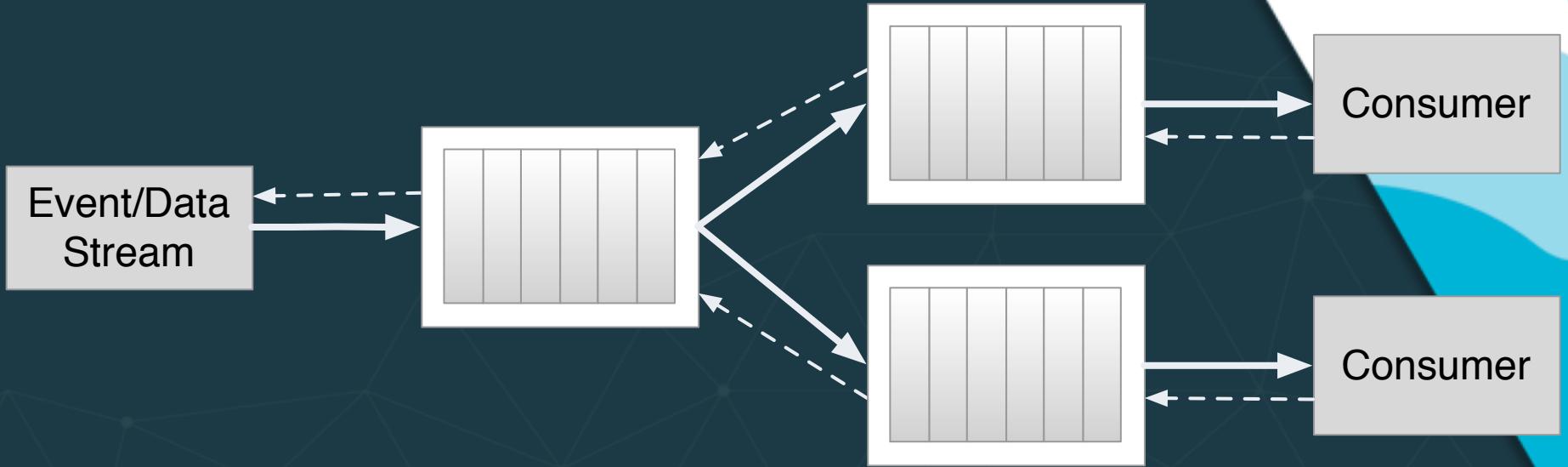


52

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

 Lightbend

54

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



- 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



55

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

- 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))
```

57

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._
```

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()
```

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()
```

```
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)
```

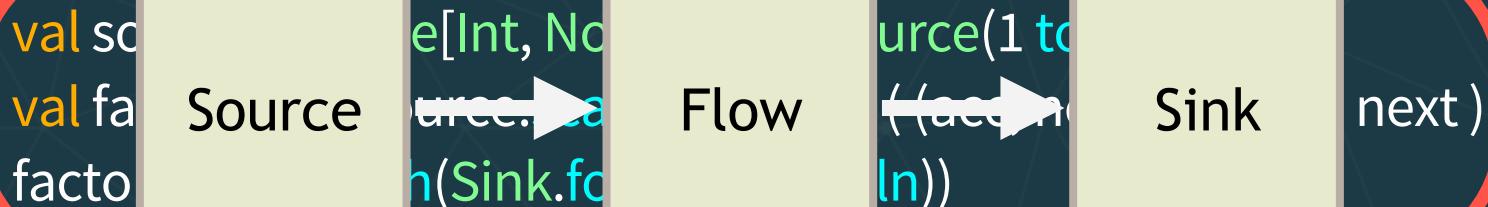
```
factorials.runWith(Sink.foreach(println))
```

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



63

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



- 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
```

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!)

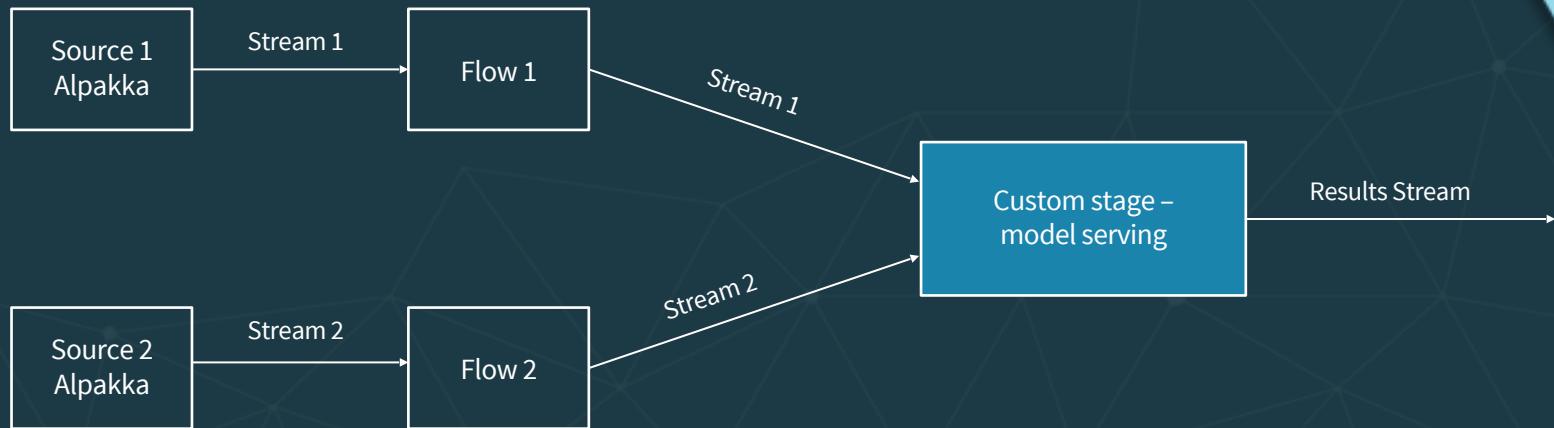
Implementations

- How do we integrate model serving (or any other new capability) into an Akka Streams app? We'll look at two approaches:
 - 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...

We provide two implementations.

Using Custom Stage

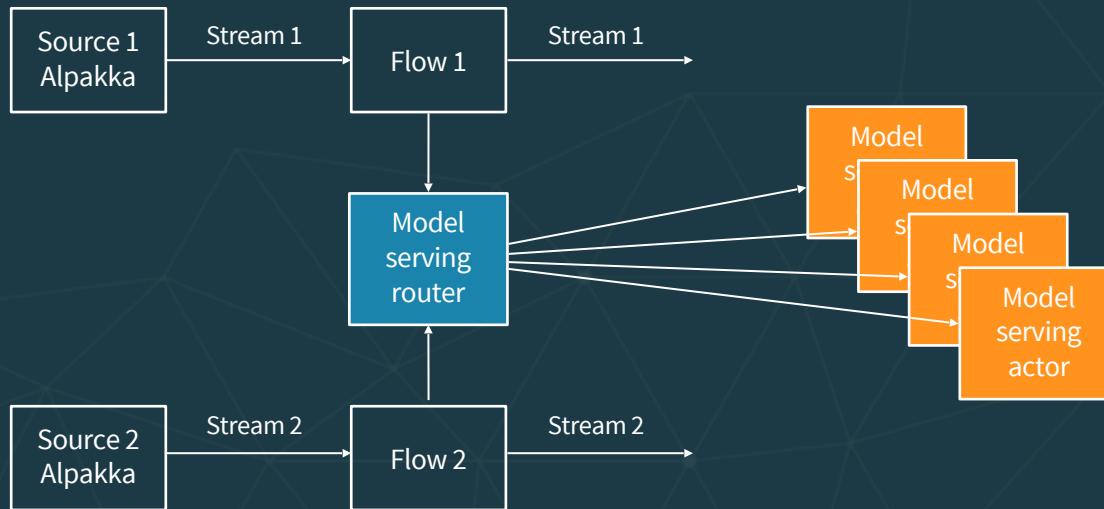
Create a custom stage, a fully type-safe way to encapsulate new functionality. Like adding a new “operator”.



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

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!!



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.

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

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

- 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!

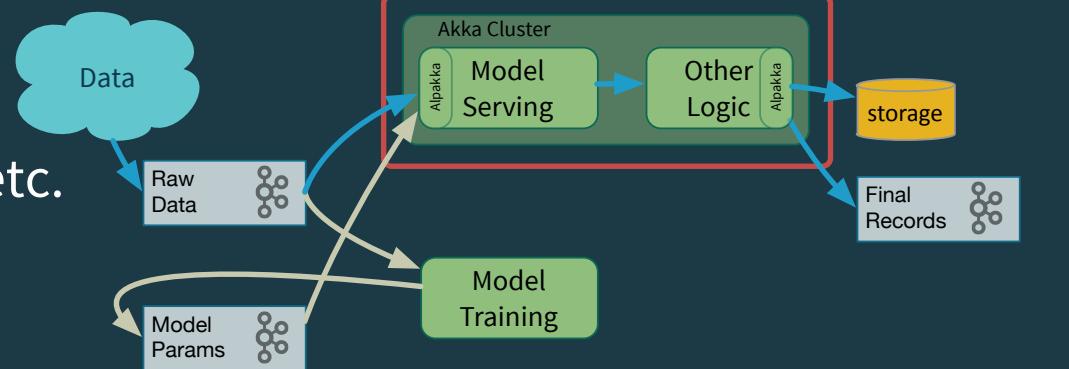
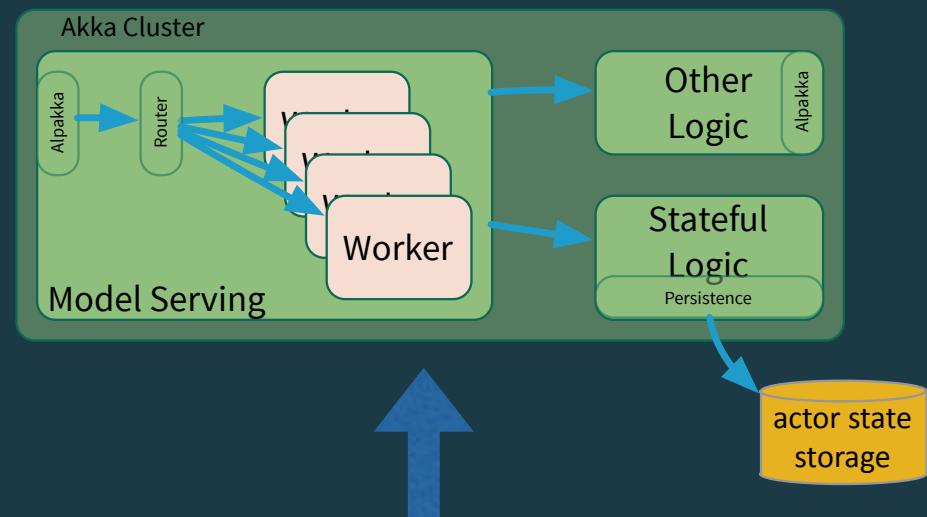
- We've prepared some exercises. We'll return to them after discussing Kafka Streams.
- To find them, search for `// Exercise` comments in the code base.

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.

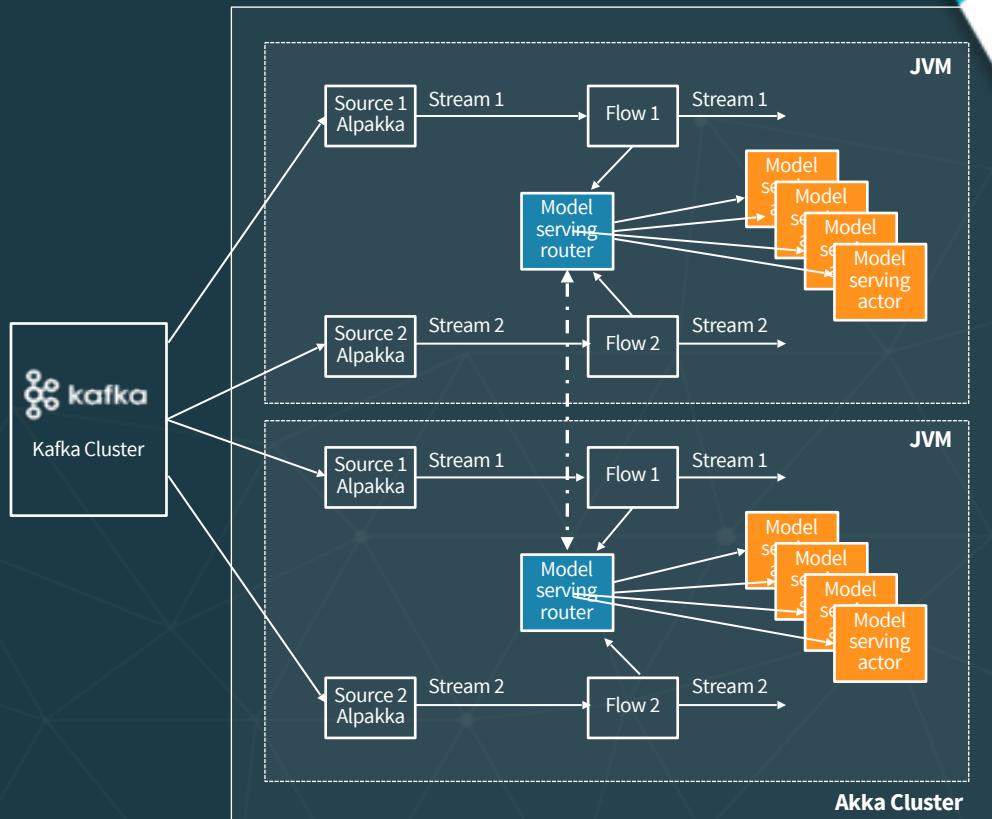


Here's our streaming microservice example adapted for Akka Streams. We'll still use Kafka topics in some places and assume we're using the same implementation for the "Model Training" microservice. Alpakka provides the interface to Kafka, DBs, file systems, etc. We're showing two microservices as before, 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.

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.



73

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?



- Extremely low latency
- Minimal I/O and memory overhead
- No marshaling overhead (maybe...)
- 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?



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

- 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



Same sample use case, now with 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 O'Reilly report](#) ;)
 - [Talks, blog posts, & book by Tyler Akidau](#)



77

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 - key/value store of supplemental data
 - Efficient management of application state



78

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
 - 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:
 - Processor Topology API
 - Compare to [Apache Storm](#)
 - DSL based on collection transformations
 - Compare to Spark, Flink, Scala collections.





Kafka Streams

- Started with a Java API
- Lightbend donated a Scala API to Kafka
 - <https://github.com/apache/kafka/tree/trunk/streams/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 or Flink)



81



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.

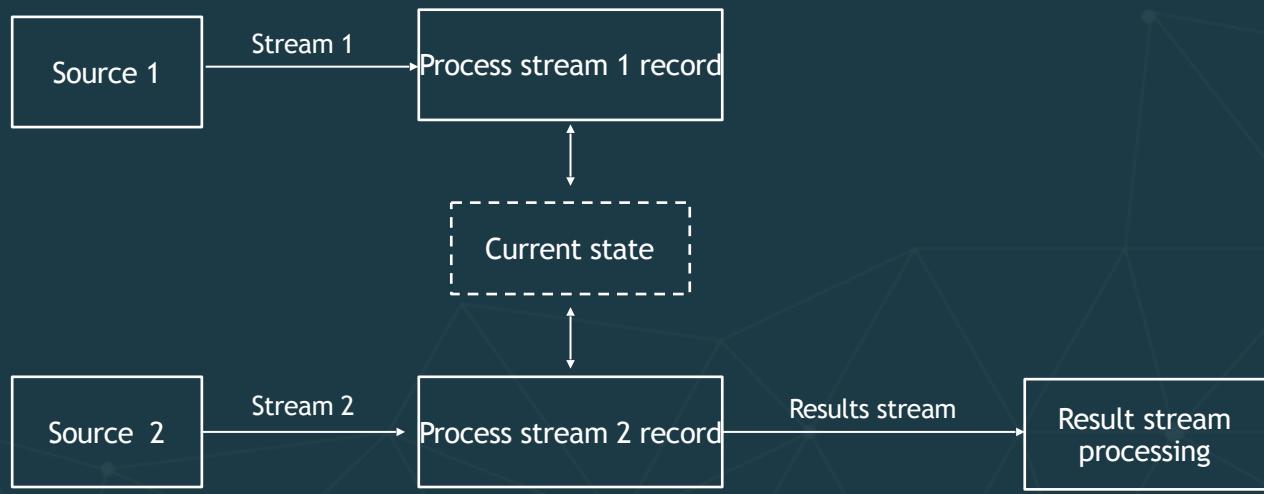


Kafka Streams

- 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



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



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) 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

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>

Additional architectural concerns for model serving

- Model tracking
- Speculative model execution

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**

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

Model tracking

- Need to remember models - a model repository
- Basic info for the model:
 - Name
 - Version (or other unique ID)
 - Creation date
 - Quality metric
 - Definition
 - ...

Model tracking

- You also need to augment the records with the model ID, as well as the score.
 - Input Record



- Output Record with Score, model version ID



Speculative execution

According to Wikipedia speculative execution is:

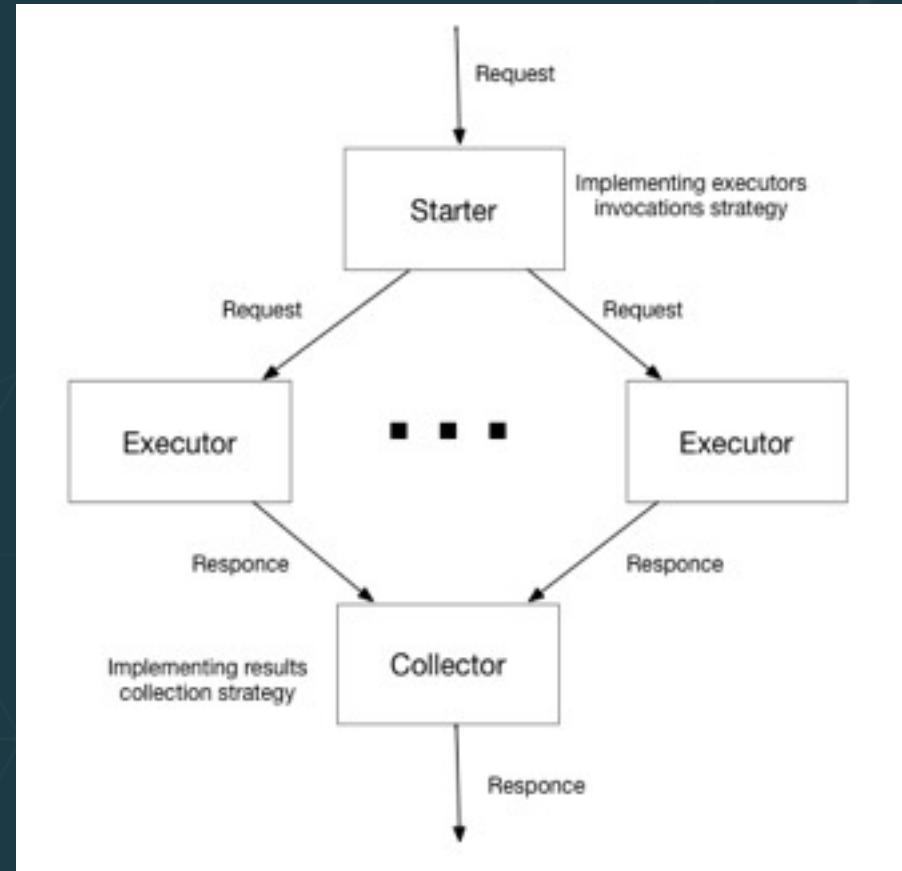
- an **optimization** technique
- The **system** performs work that may not be needed, before it's known if it will be needed
- So, if and when we discover it IS needed, we don't have to wait
- Or, results are discarded if not needed.
-

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??

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

- Used to guarantee execution time
 - Several models:
 - A smart model, but takes time T_1
 - A “less smart”, but fast model with a fixed upper-limit on execution time, $T_2 \ll T_1$
 - If timeout ($T > T_2$) occurs before smart finishes, return the less accurate result
 - Why is $T > T_2$ required??
 - ...

Because the timeout T has to be long enough that the fast model has time to finish, so T must be longer than T_2 .

Note that this technique is “speculative”, because we try both models, taking a compromise result if the good result takes too long to return.

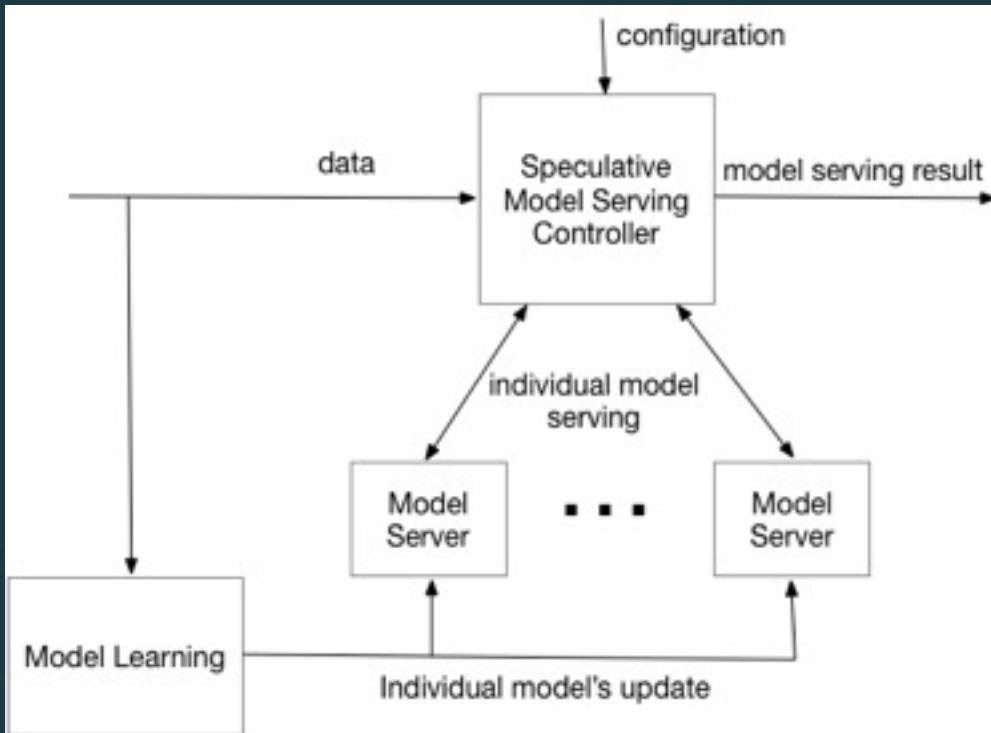
Applicability for model serving

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

Applicability for model serving

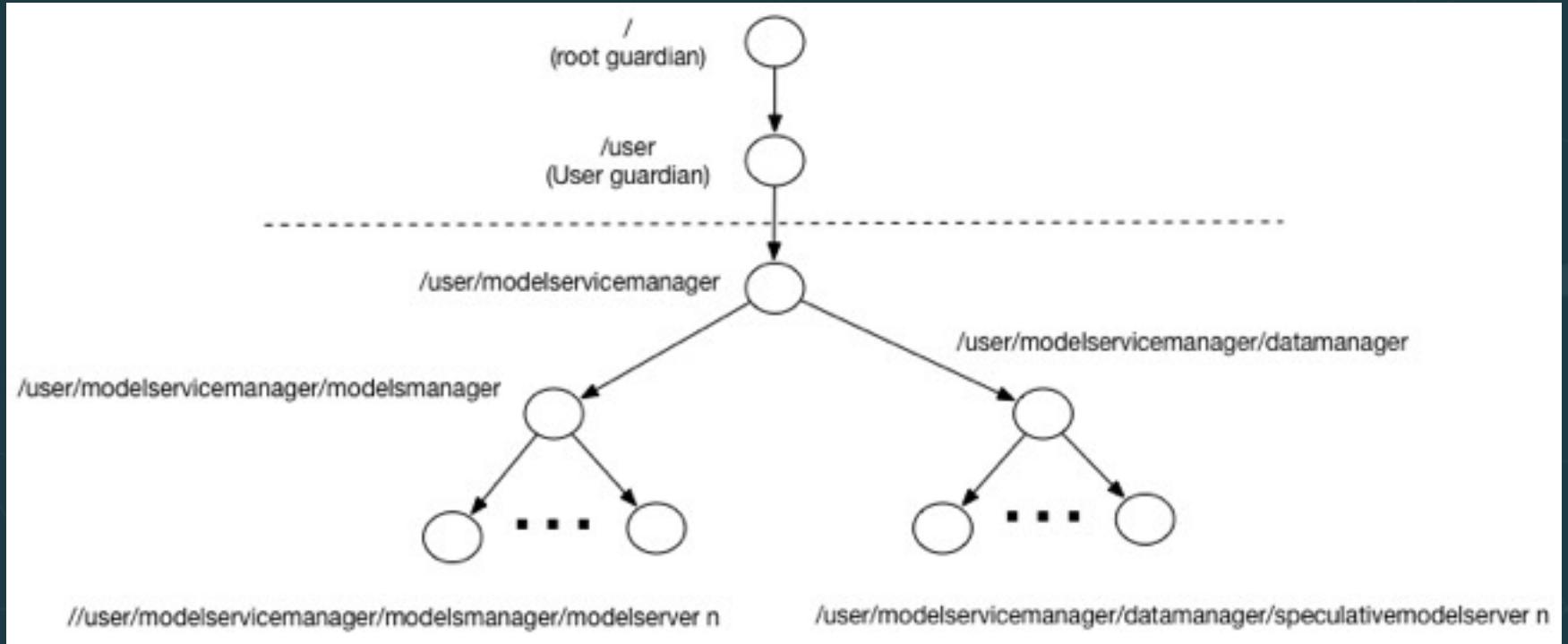
- ...
- Quality based model serving.
 - If we have a quality metric, pick the result with the best result.
- Of course, you can combine these techniques.

Architecture



<https://developer.lightbend.com/blog/2018-05-24-speculative-model-serving/index.html>

Actors



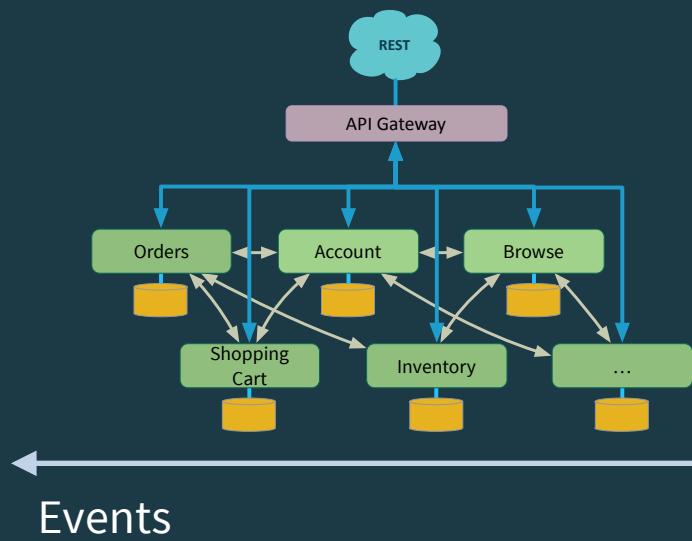
Wrapping Up



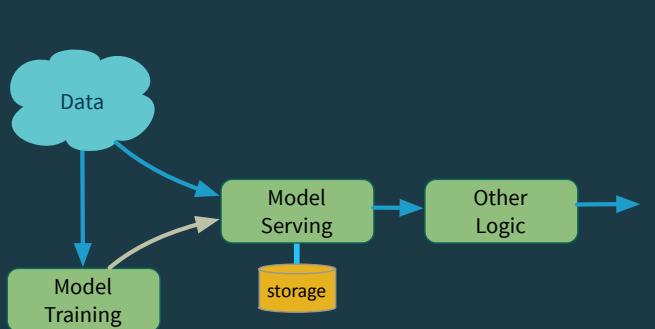
To Wrap Up



Event-driven μ-services



“Record-centric” μ-services



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 its “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”.

In Our Remaining Time

1. Explore the code we didn't discuss (there is a lot ;)
 1. Study the different model serving techniques
 2. Study the “model” subproject
 3. Look at how the following are implemented
 1. queryable state
 2. embedded web servers
 3. use of Akka Persistence
 4. model serialization
2. ...

In Our Remaining Time

1. ...
2. Try the exercises - search for // Exercise in the code
3. Ask us for help on anything...
4. Visit lightbend.com/fast-data-platform
5. Profit!!

Other Lightbend Talks

- Other Lightbend sessions this week:
 - Gerard Maas, Processing fast data with Apache Spark: A tale of two APIs, Wednesday, 11:20-12:00, 1E 07/08
 - Dean, Executive Briefing: What you need to know about fast data, Thursday, 2:00-2:40, 1E 14.
 - .. and visit the Lightbend booth!!

Thanks for coming!

Questions?

lightbend.com/products/fast-data-platform

boris.lublinsky@lightbend.com



Thank you! Please check out the other sessions with Dean, and our colleague Gerard Maas. Check out our Fast Data Platform for commercial options for building and running microservices with Kafka, Akka Streams, and Kafka Streams.