Building Kafka-based Microservices with Akka Streams and Kafka Streams

Boris Lublinsky and Dean Wampler, Lightbend

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

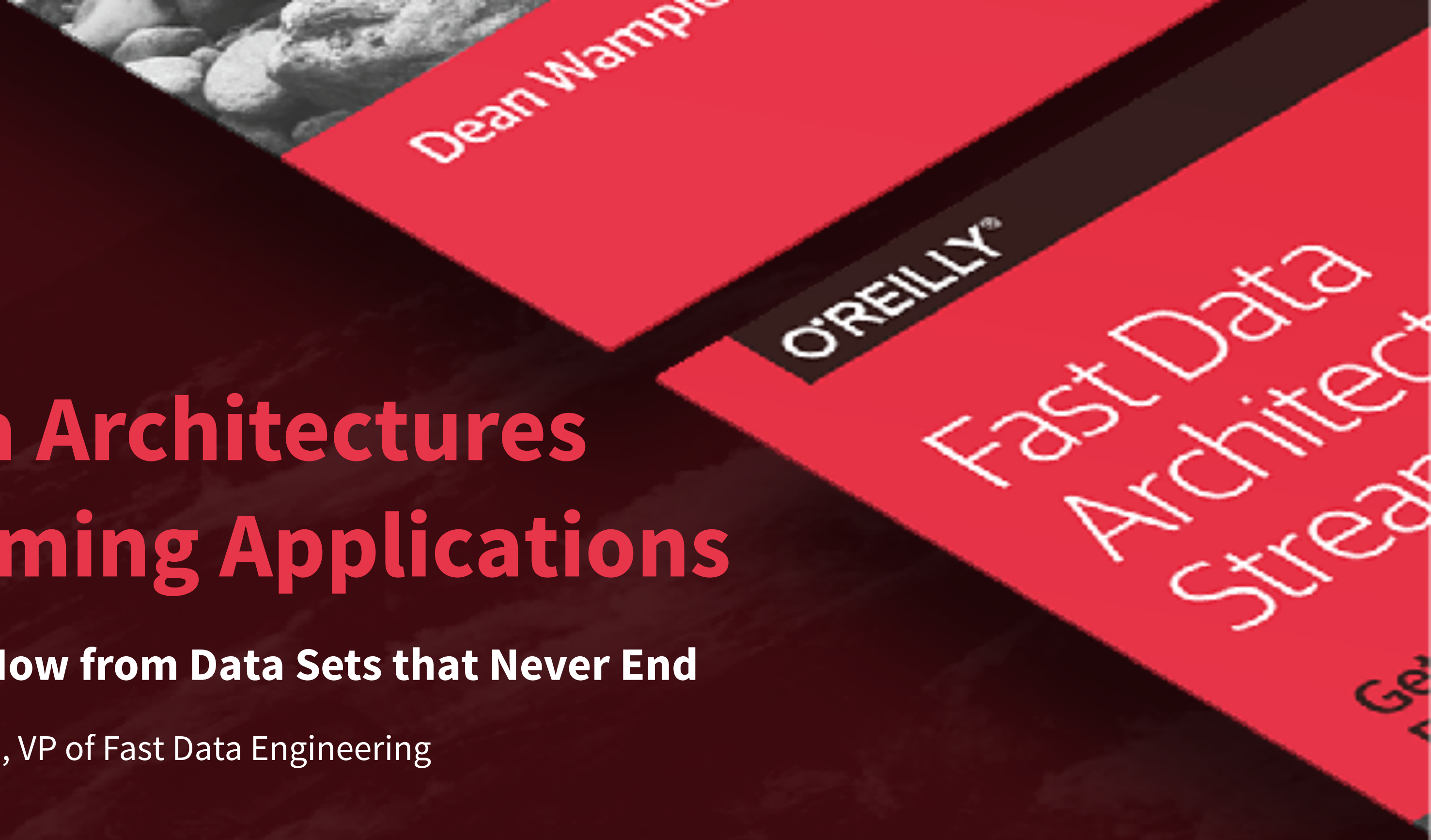
Lightbend

O'REILLY

Fast Data Architectures for Streaming Applications

Getting Answers Now from Data Sets that Never End

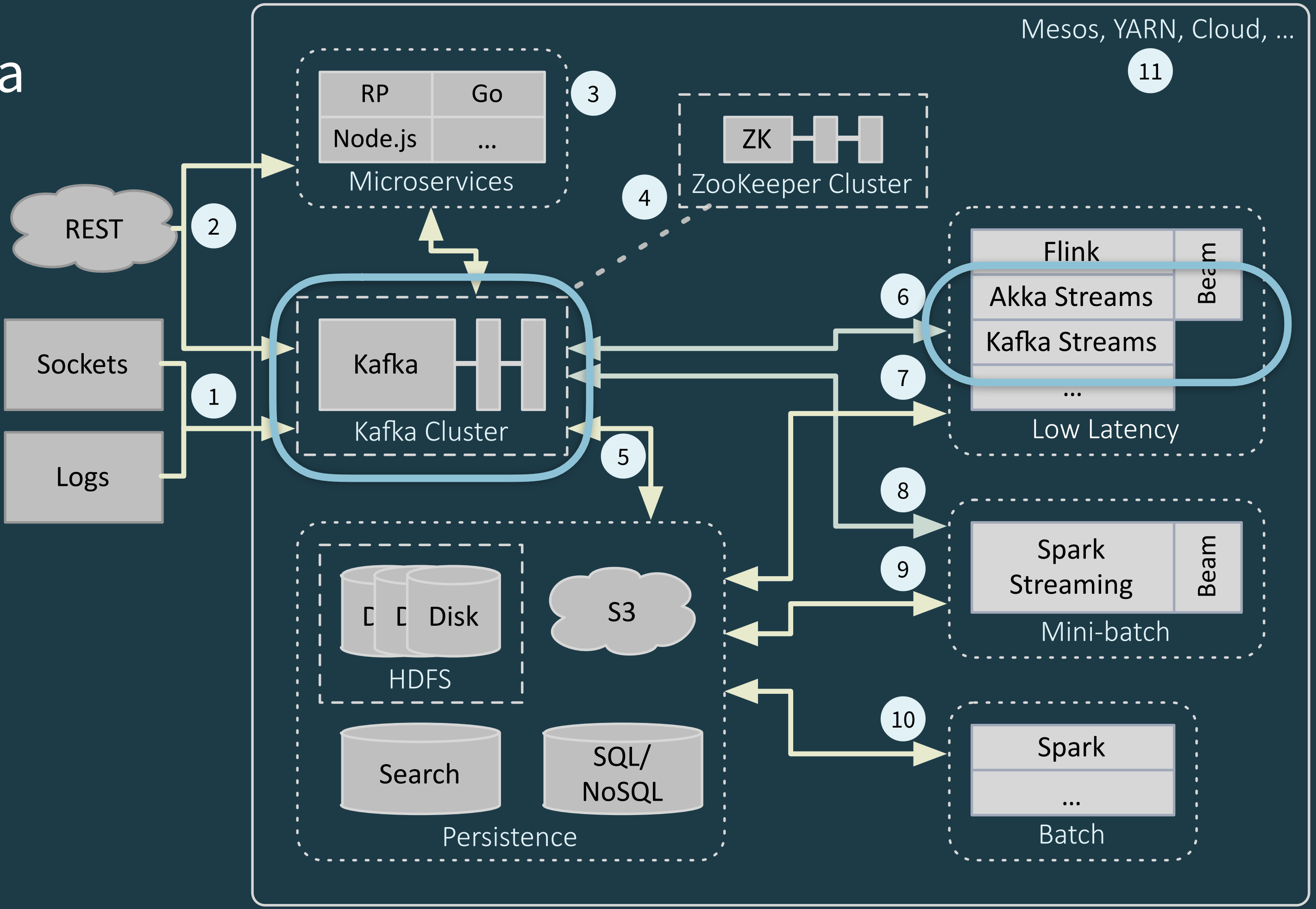
By Dean Wampler, Ph. D., VP of Fast Data Engineering



Get Your Free Copy

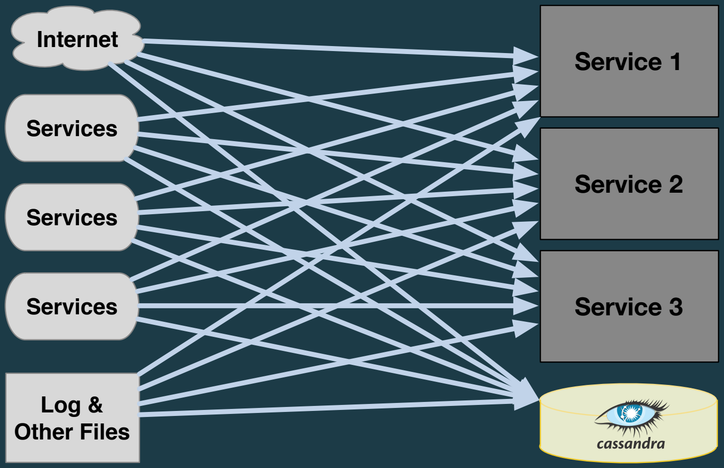
Today's focus:

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



Why Kafka for Connectivity?

Before:

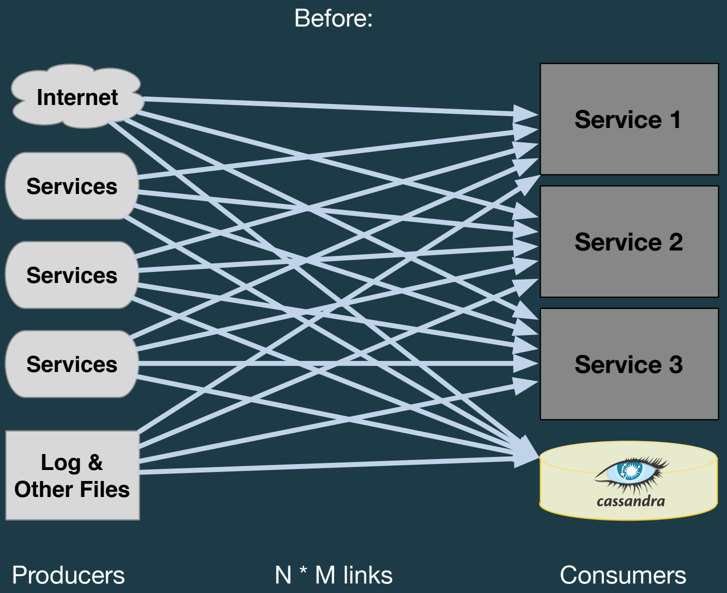


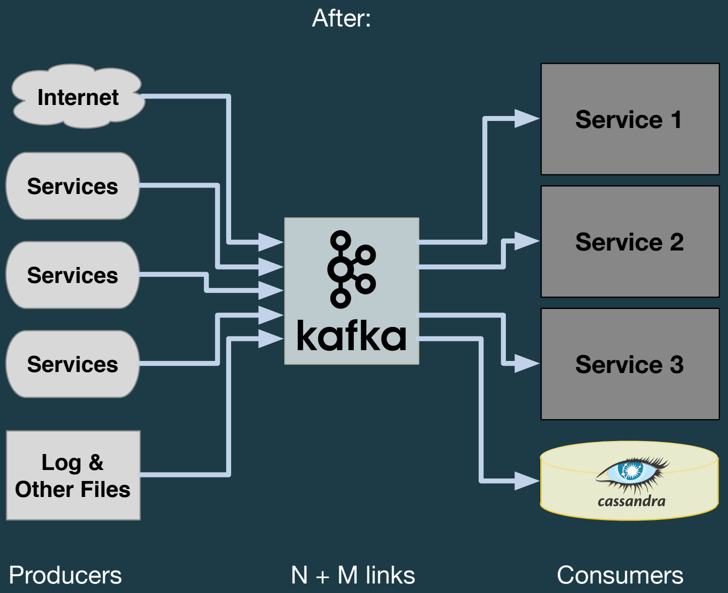
Producers

N * M links

Consumers

Why Kafka for Connectivity?



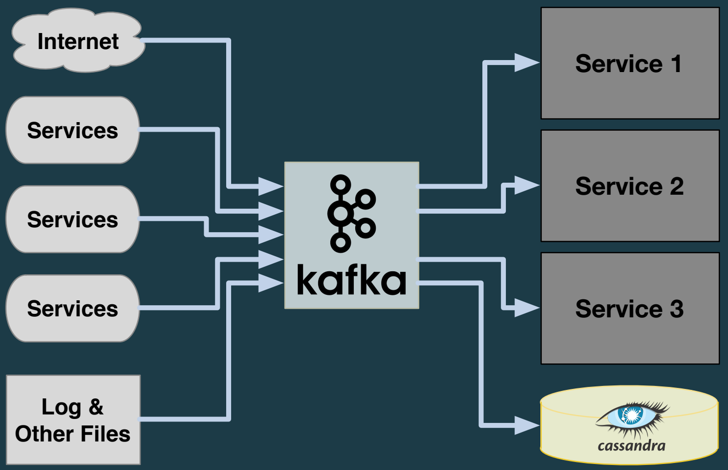


Why Kafka for Connectivity?

Kafka:

- Simplify dependencies between services
    - Improved data consistency
- Minimize data transmissions
- Reduce data loss when a service crashes
- M producers, N consumers
    - Improved extensibility
- Simplicity of one “API” for communication

After:



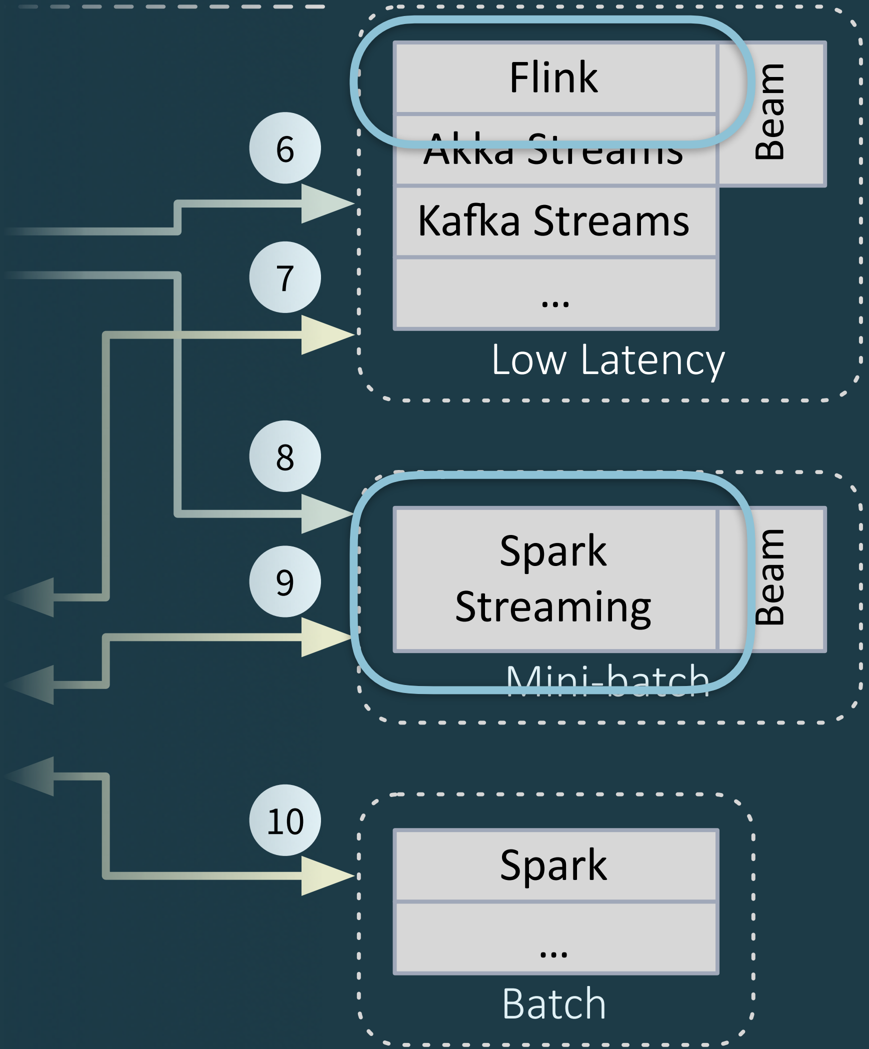
Producers

N + M links

Consumers

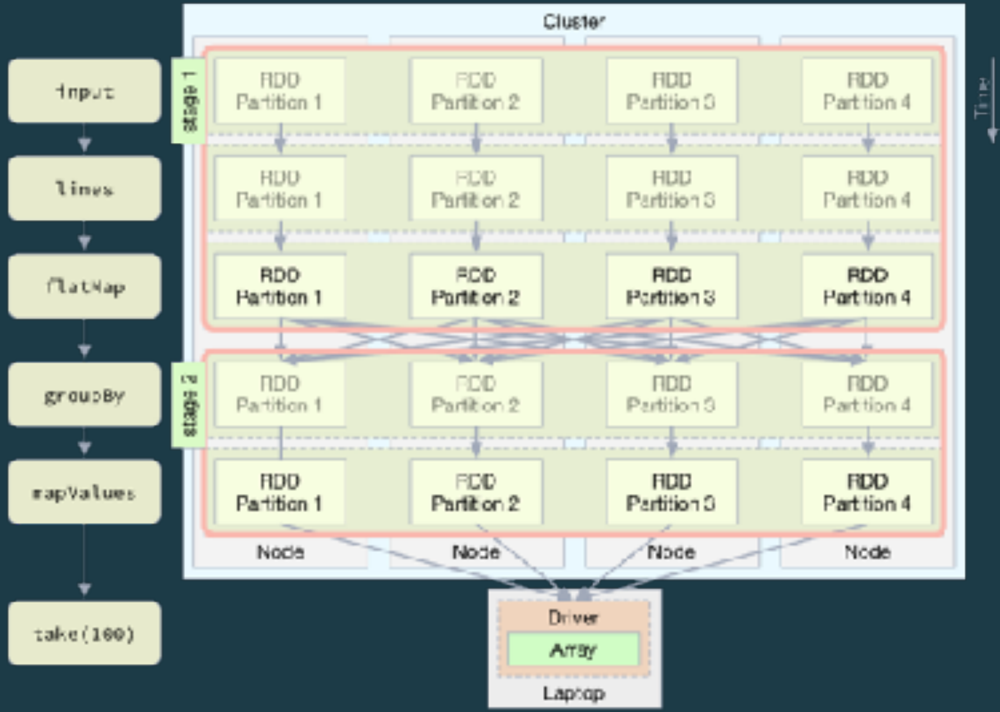
Streaming Engines:

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



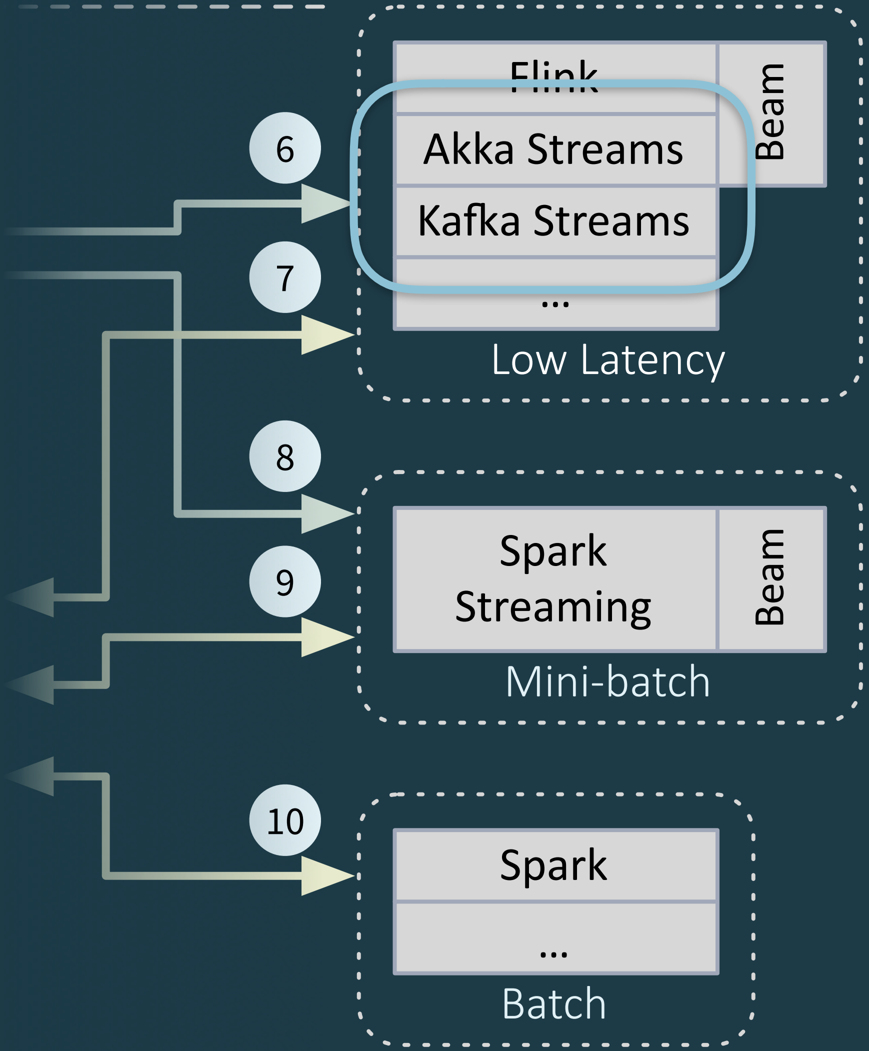
Streaming Engines:

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



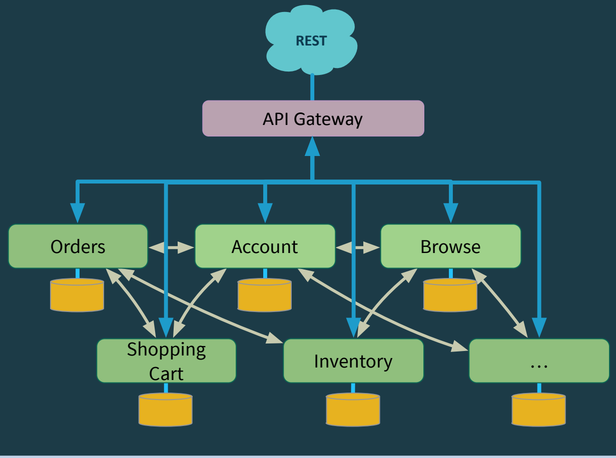
Streaming Frameworks:

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

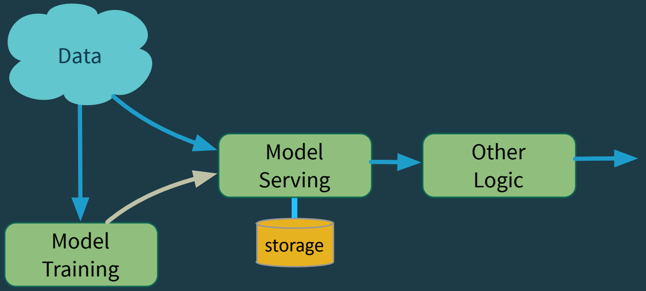


A Spectrum of Microservices

Event-driven µ-services



“Record-centric” µ-services

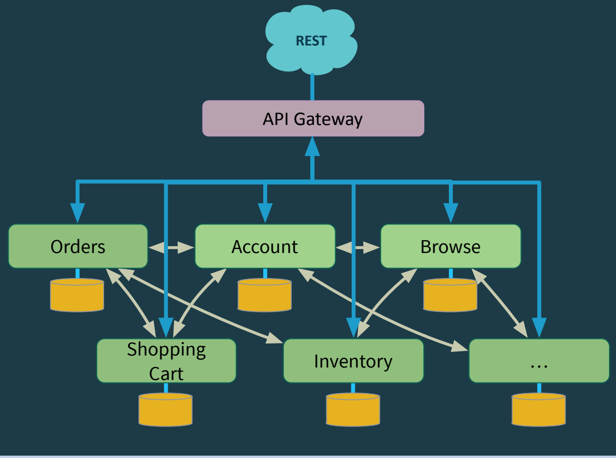


Events

Records

A Spectrum of Microservices

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.

Events

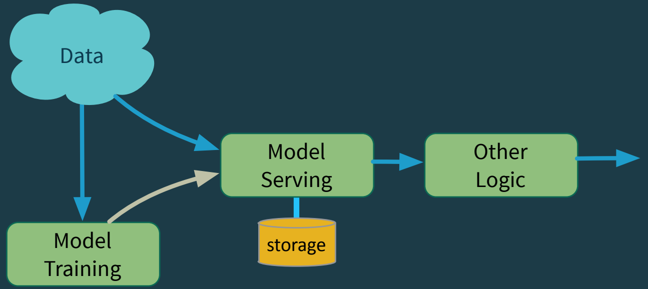
Records

A Spectrum of Microservices

Emerged from the right-hand side.

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

“Record-centric” µ-services

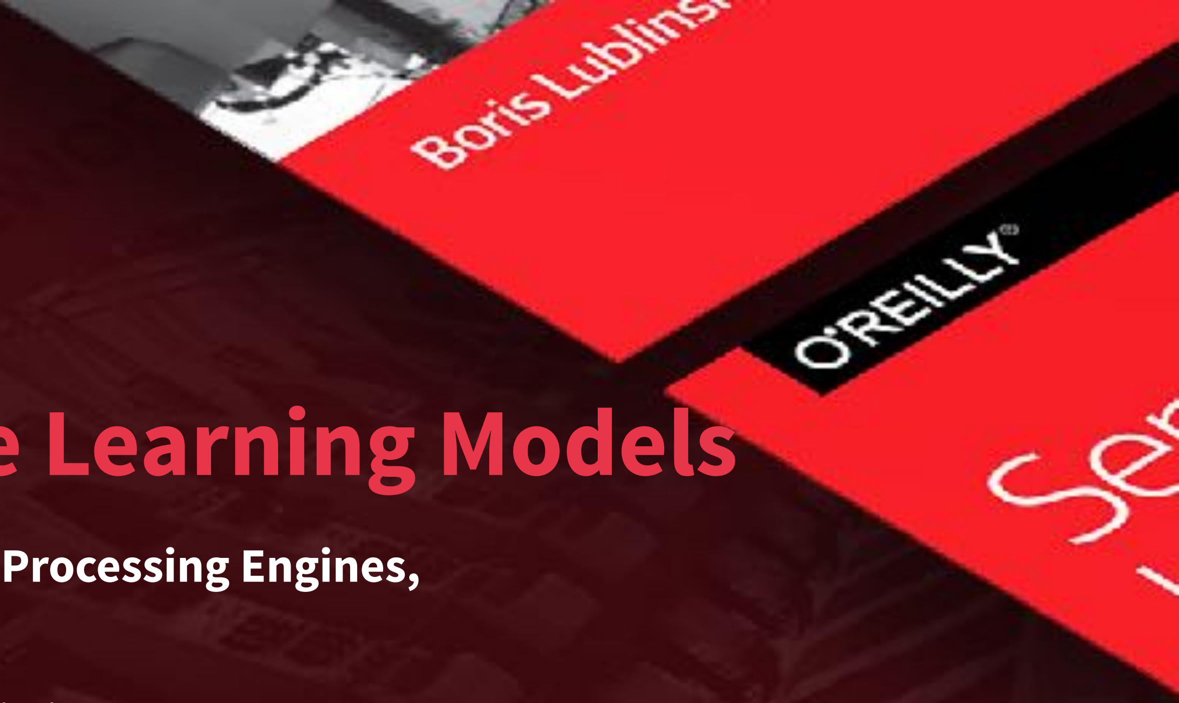


Events

Records

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

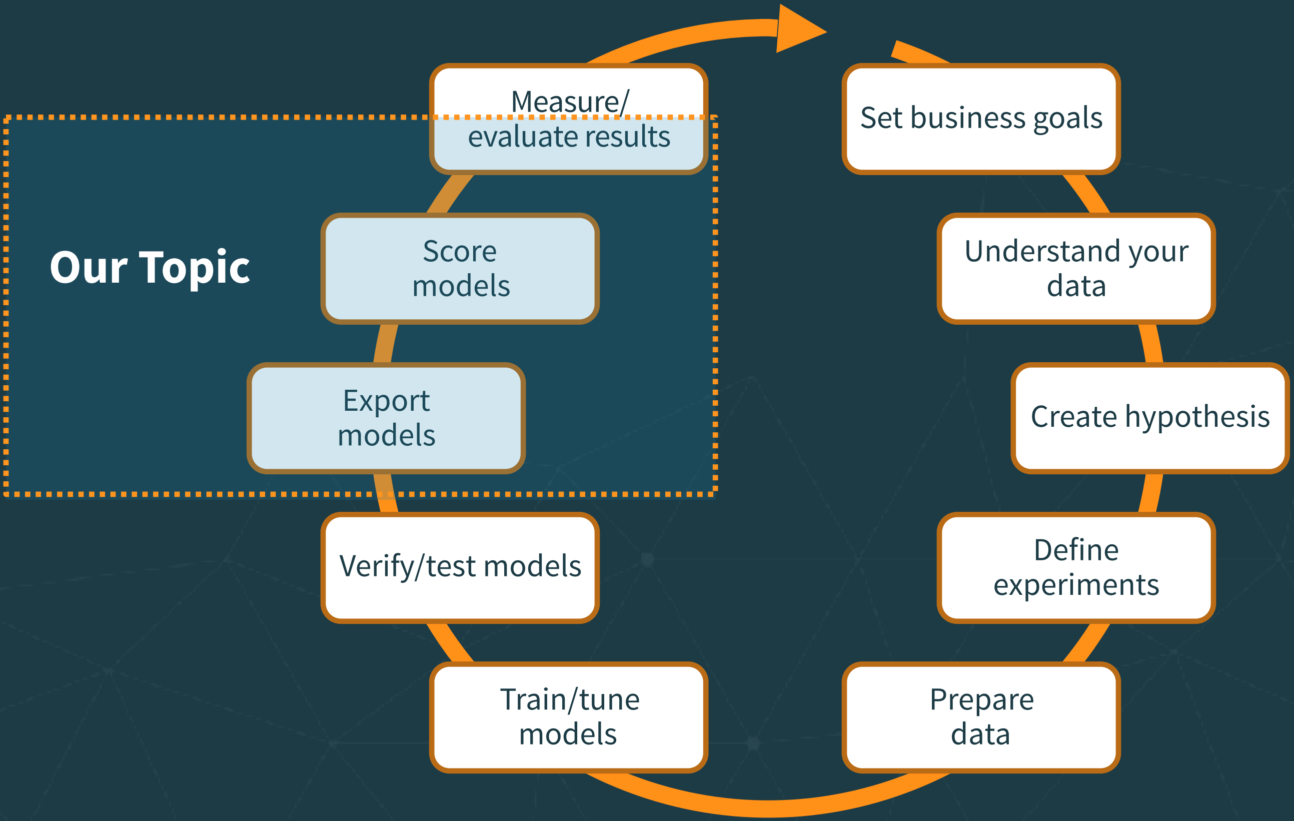
Magic

Happiness

Maybe Not

Even If There Are Instructions

The Reality


Measure/ evaluate results
Our Topic
Score models
Export models
Verify/test models
Train/tune models
Set business goals
Understand your data
Create hypothesis
Define experiments
Prepare data

What Is The Model?

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

for example:

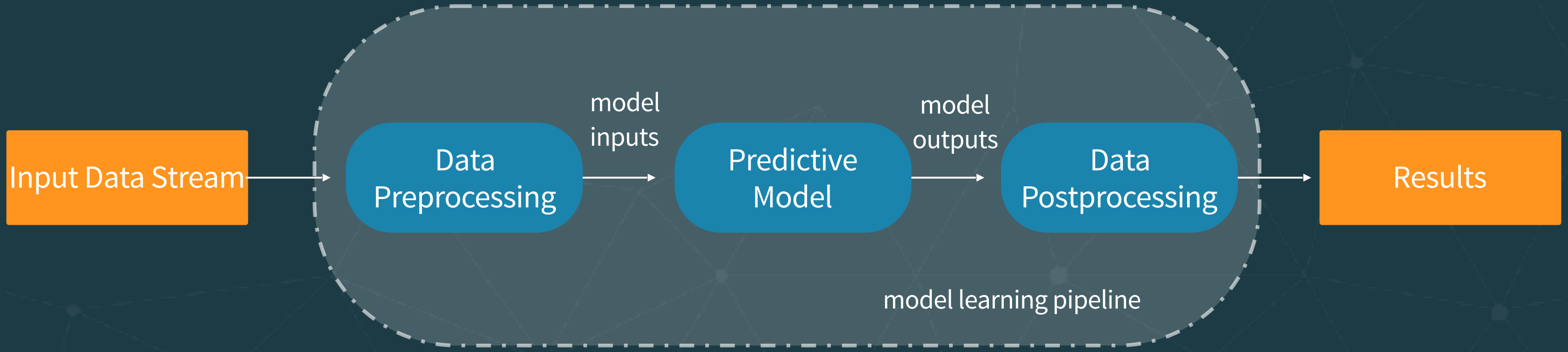
Linear regression: $y = a_c + a_1 * x1 + \dots + a_n * x_n$

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.

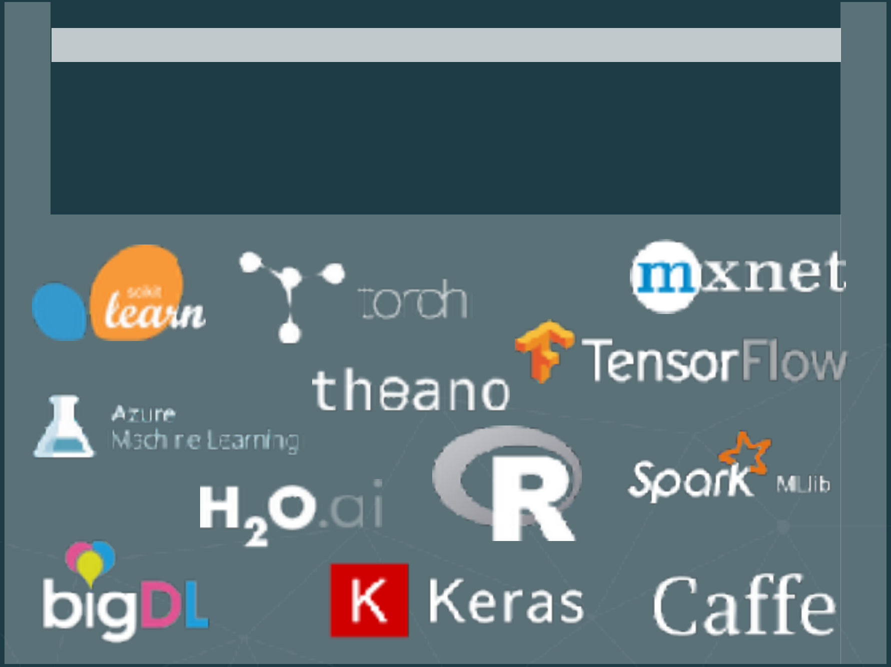


Traditional Approach To Model Serving

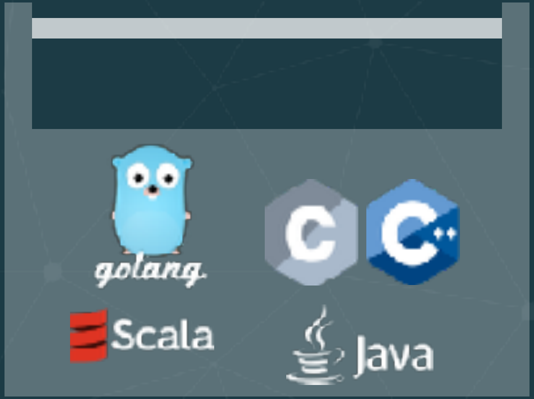
- Model is code
- This code has to be saved and then somehow imported into model serving

Why is this problematic?

Impedance Mismatch

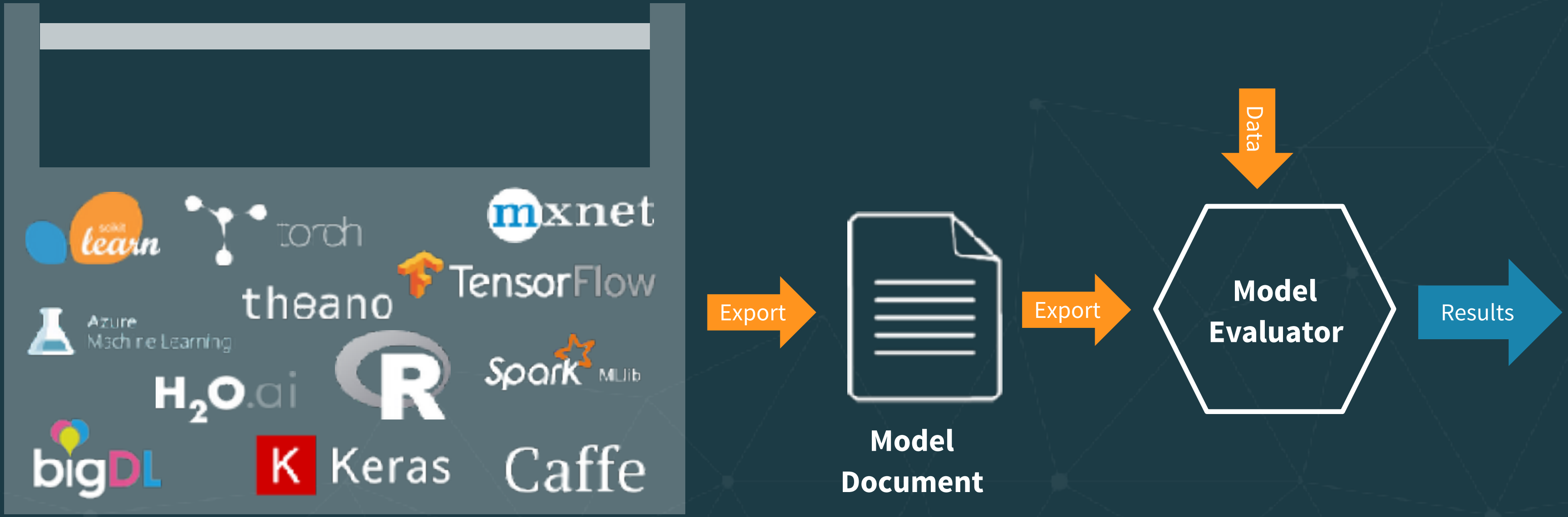


Continually expanding Data Scientist toolbox



Defined Software Engineer toolbox

Alternative - Model As Data



Standards

PMML Predictive Model Markup Language

Portable Format for Analytics **(PFA)**

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.

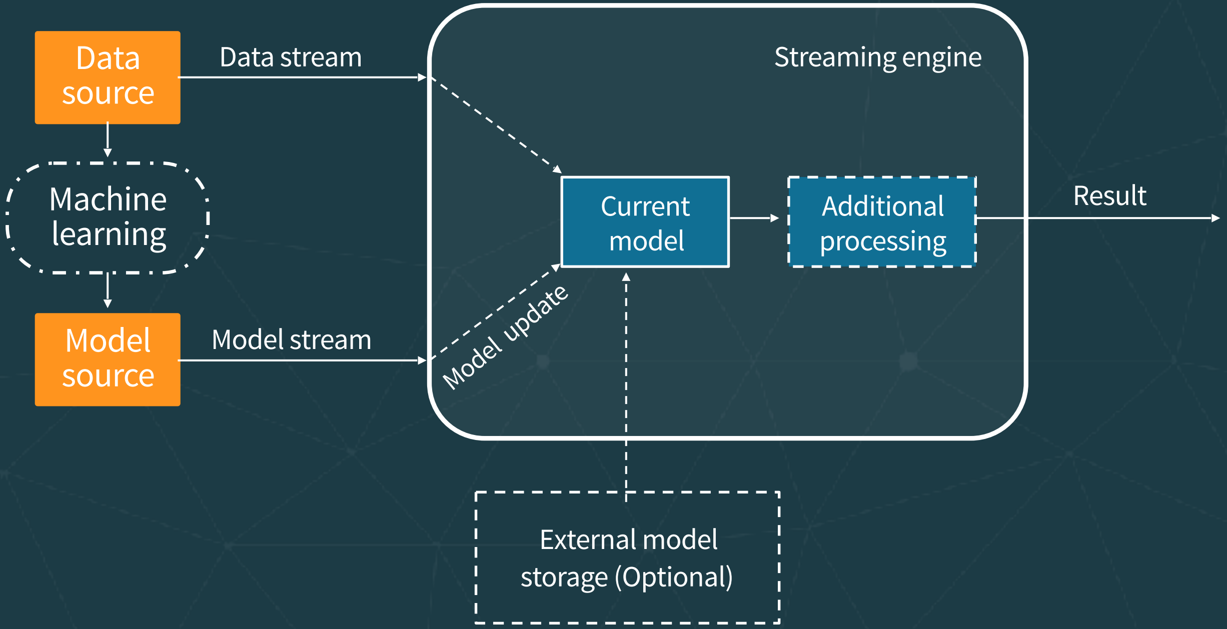
Additional Considerations – Model Lifecycle

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

Lightbend

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)

```
// Internal

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
}
```

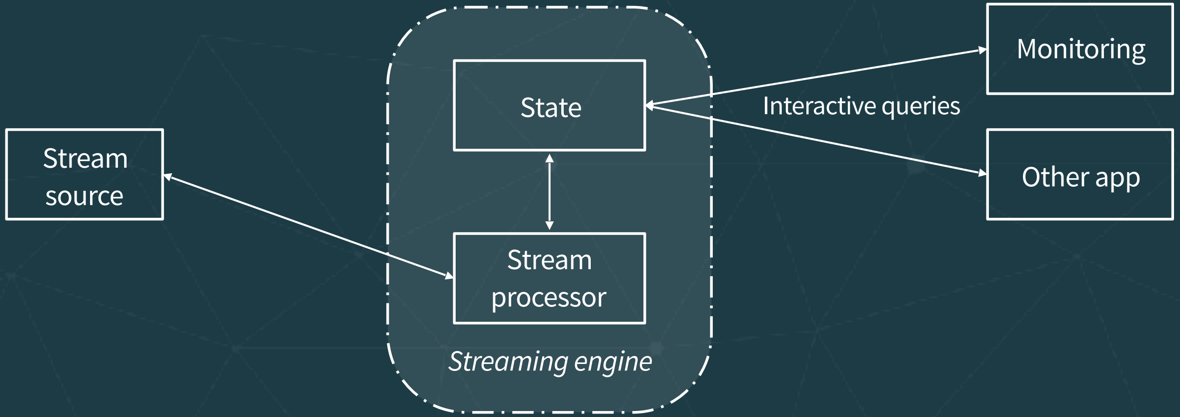
Additional Considerations: Monitoring

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,                  // Time spent on serving
    var min : Long = Long.MaxValue,              // Min serving time
    var max : Long = Long.MinValue               // Max serving time
)
```

Queryable State

Queryable state (interactive queries) is an approach, which allows to get more from streaming than just the processing of data. This feature allows to treat the stream processing layer as a lightweight embedded database and, more concretely, to *directly query the current state* of a stream processing application, without needing to materialize that state to external databases or external storage first.



Implementation Options

Modern stream-processing engines (SPE) take advantage of the cluster architectures. They organize computations into a set of operators, which enables execution parallelism; different operators can run on different threads or different machines.

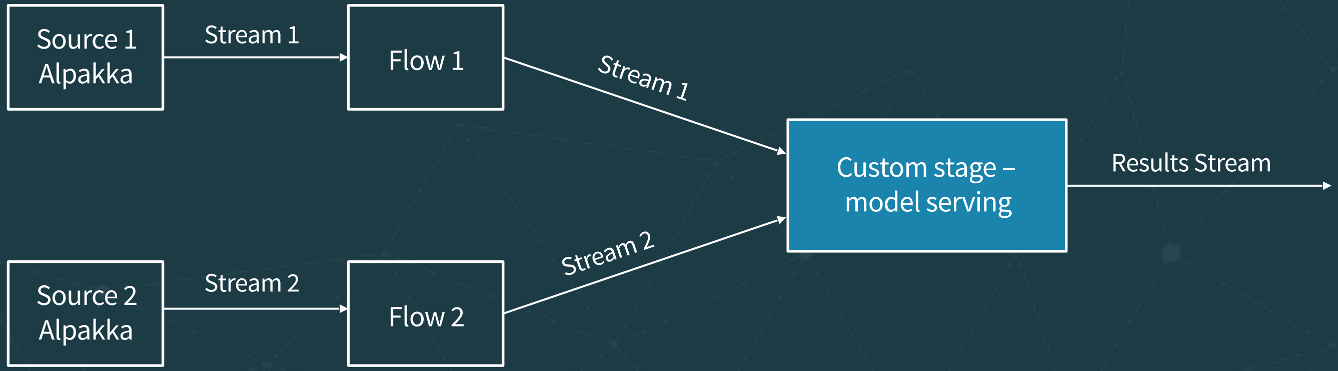
Stream-processing library (SPL), on the other hand, is a library, and often domain-specific language (DSL), of constructs simplifying building streaming applications.

akka streams

- Akka Streams is a library focused on in process back-pressured reactive streaming.
- Can be used with Akka Cluster
- Provides a broad ecosystem of connectors to various technologies (data stores, message queues, file stores, streaming services, etc) - Alpakka
- Integrated with other Akka components, including Akka Actors, Akka HTTP, etc.
- In Akka Streams computations are written in a DSL for defining a graph, which aims to make translating graph drawings to and from code simpler.

Using Custom Stage

Create custom stage, which is a fully type-safe way to encapsulate required functionality. ur stage will provide functionality somewhat similar to a
Flink low-level join



Using Custom Stage

Code time

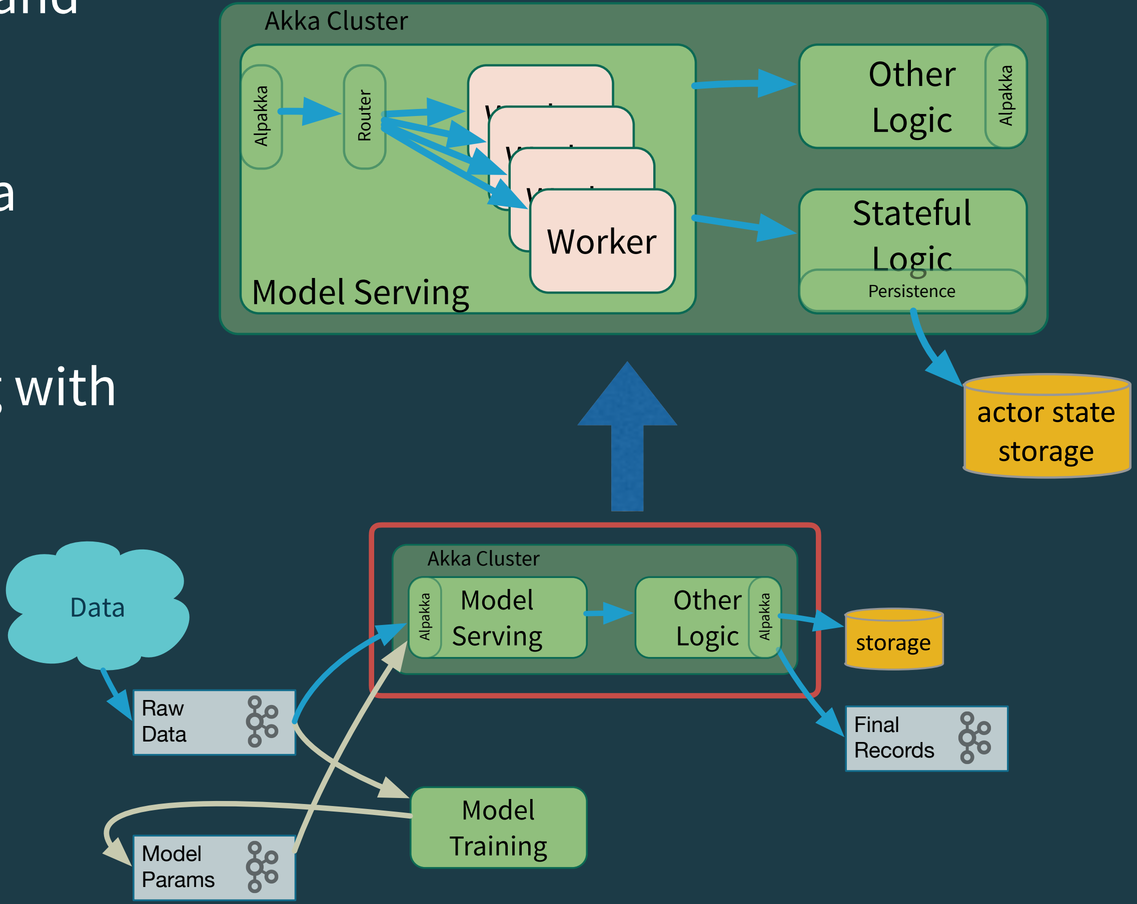
Other Concerns

- Scale scoring with workers and routers, across a cluster

- Persist actor state with Akka Persistence

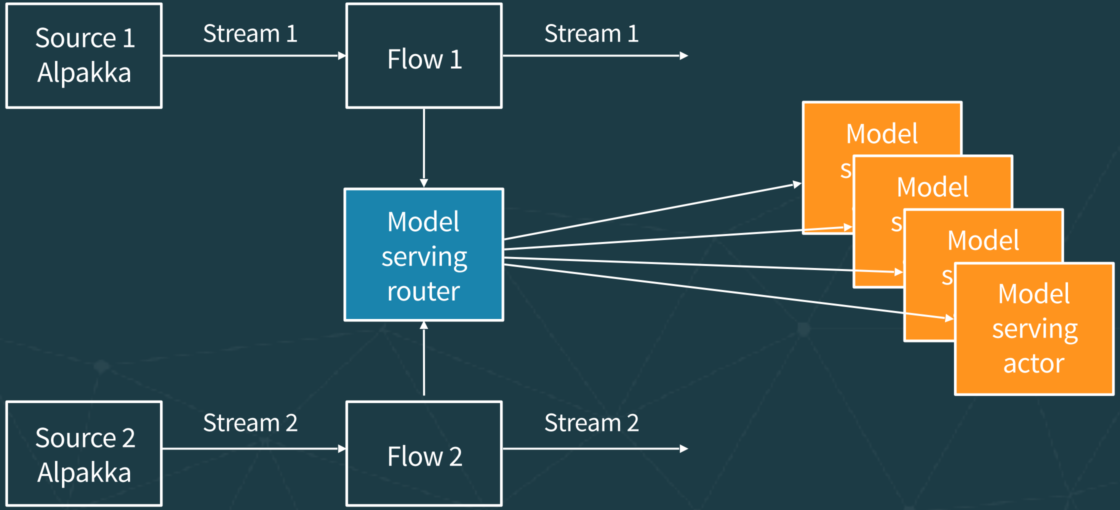
- Connect to *almost* anything with Alpakka

- *Lightbend Enterprise Suite*
  - for production



Improve Scalability

Using the router actor to forward request to an individual actor responsible for processing request for a specific model type low-level join



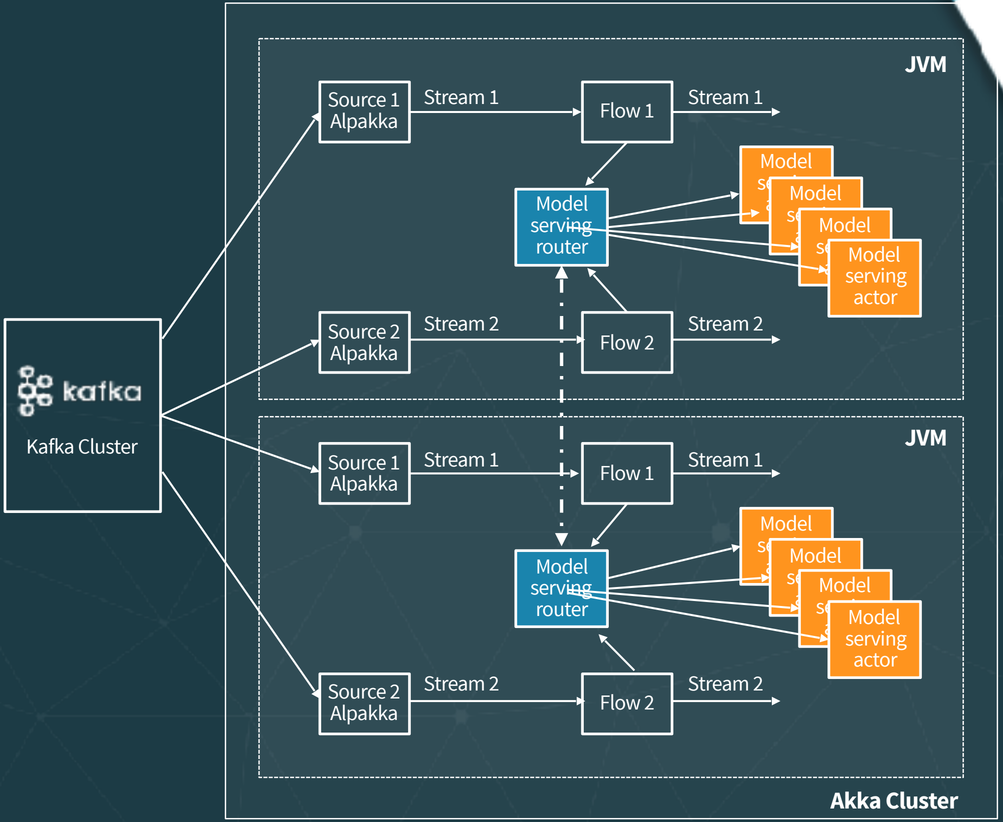
Akka Streams with actors

Code time

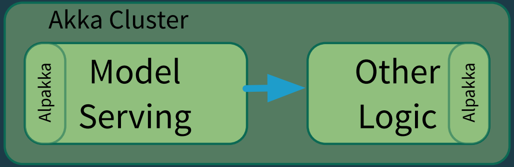
Using Akka Cluster

Two level of scalability:

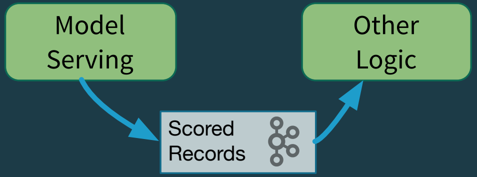
- Kafka partitioned topic allow to scale listeners according to the amount of partitions.
- Akka cluster sharing allows to split model serving actors across clusters.



Go Direct or Through Kafka?



vs.

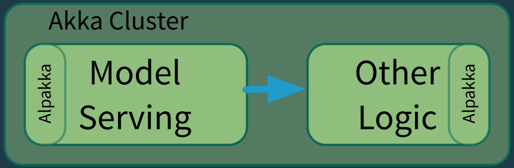


?

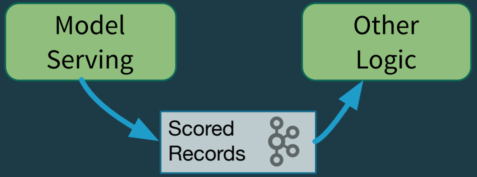
- Extremely low latency
- Minimal I/O and memory overhead
- No marshaling overhead
- *Reactive Streams* back pressure
- Higher coupling - M producers, N consumers, but directly connected (sort of)
- Use Akka Persistence for durable state

- Higher latency (including queue depth)
- Higher I/O and processing (marshaling) overhead
- Better potential reusability
- Better decoupling - M producers, N consumers, completely disconnected
- Automatic durability (topics on disk)

Go Direct or Through Kafka?



vs.



?

- Use for smaller, faster messaging between “components”.
- Watch for consumer “backup”
- Use Akka Persistence for important state!

- Better reusability and decoupling
- Use for larger volumes, more course-grained service interactions
- Plan partitioning and replication carefully

Kafka Streams

- Kafka Streams, is a client library for processing and analyzing data stored in Kafka.
- Provides important stream-processing concepts, such as properly distinguishing between event time and processing time, windowing support, etc.
- Provides simple yet efficient management of application state based on stream/table duality.
- Provides two types of APIs:
  - Process Topology (compare to Storm)
  - DSL based on collection transformations (compare to Spark, Flink)
- Provides Java and Scala (new) APIs

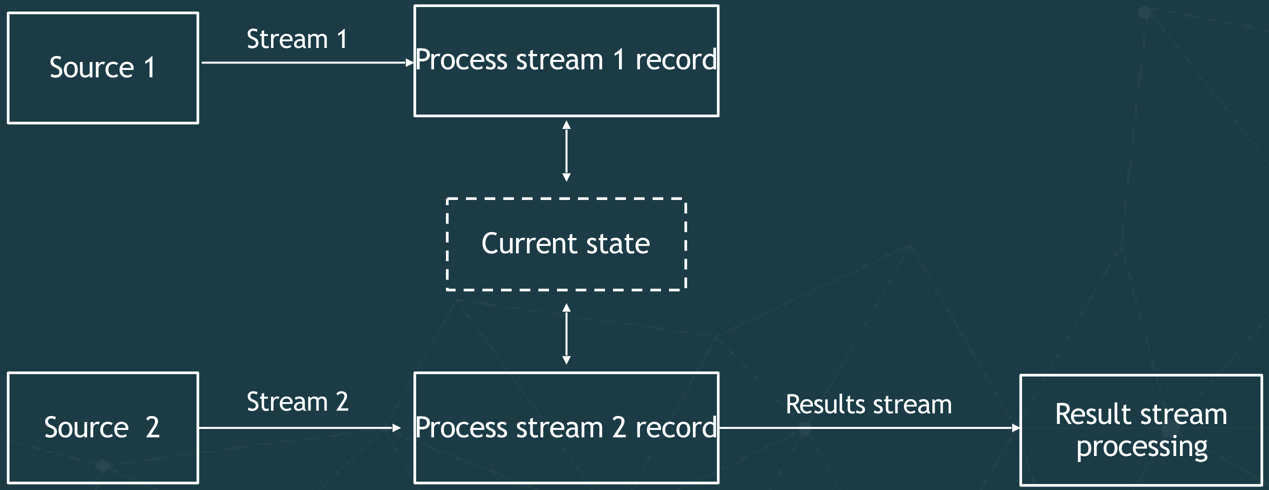
Kafka Streams

- Provides low overhead
- Provides additional integrations leveraging Kafka Connect
- Provides support for scalability and load balancing (based on Kafka partitioning)
- Provides SQL interface (leveraging a separate KSQL application)
- Queryable State

Kafka Streams

- Ideally suited for:
  - ETL - leveraging KStreams
  - Aggregations - Leveraging KTable
  - Flexible integration model - in memory or over Kafka
  - Joins, including Stream and Table joins
  - Effectively once semantics

Model Serving With Kafka Streams



State Store Options

- Naive, in memory store.
- Standard key/value store provided by Kafka Streams
- Custom store

Model Serving With Kafka Streams

Code time

Thank You

Any questions?