WEBINAR | THURSDAY NOV 1ST, 9:00 AM PT / 4:00 PM UTC

Machine Learning At Speed: Operationalizing ML For Real-Time Data Streams

Boris Lublinsky
Principal Architect at Lightbend, Inc.

Lightbend

ML is simple

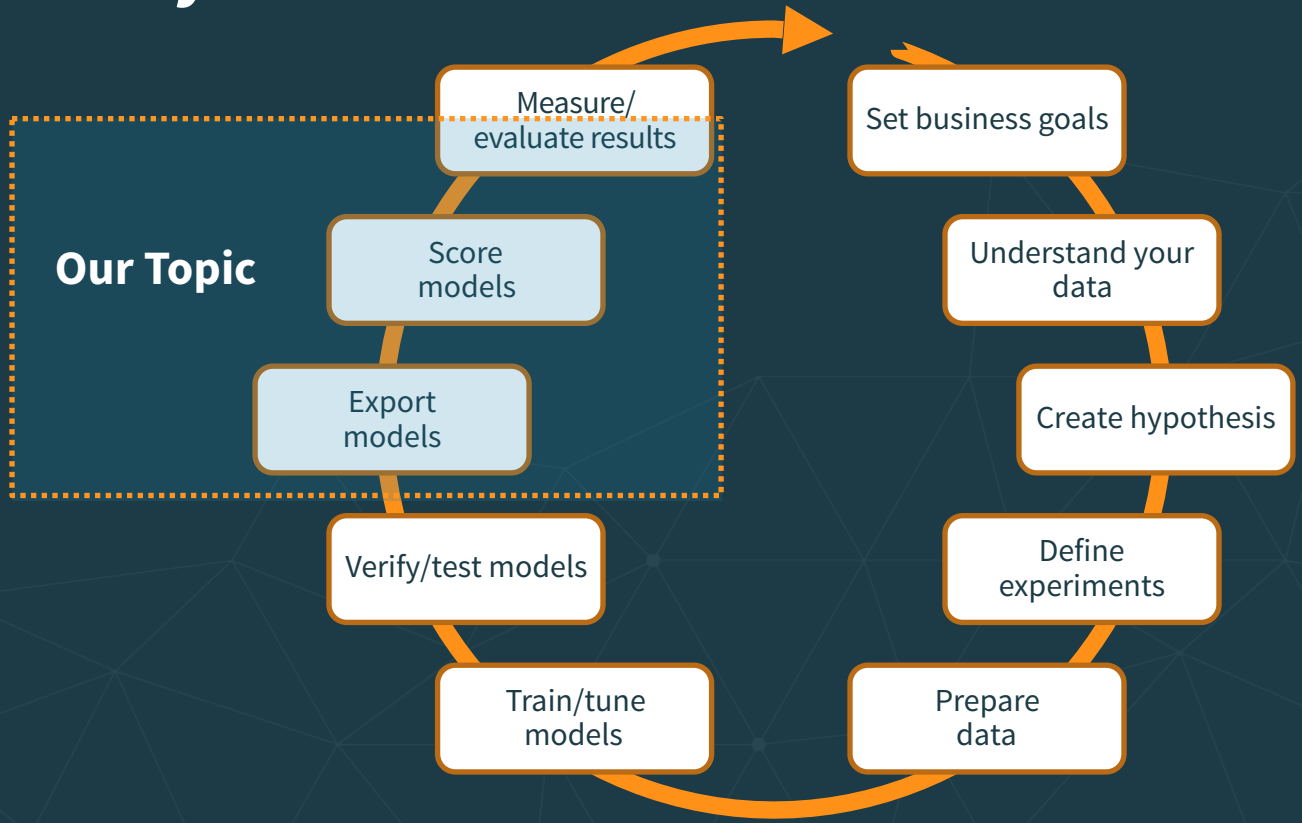
Data

Magic

Happiness

May be not

The reality


Our Topic
Measure/ evaluate results
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:

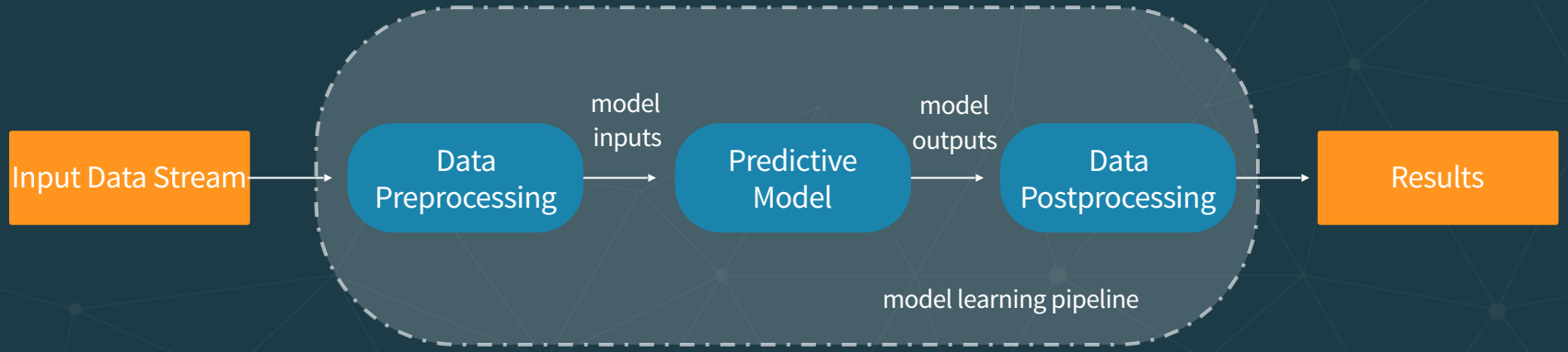
Linear regression: $y = a_c + a_1 * x1 + \ldots + a_n * x_n$

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

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.



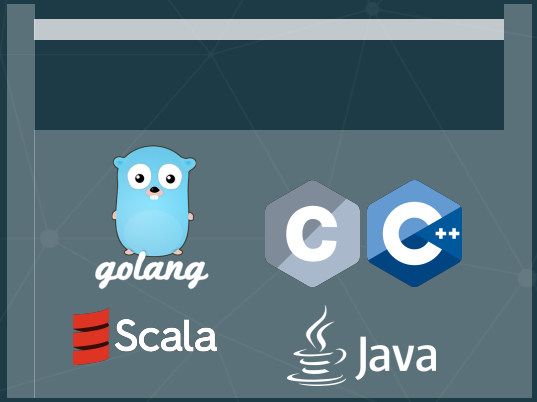
Traditional approach to model serving

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

Why is this problematic?

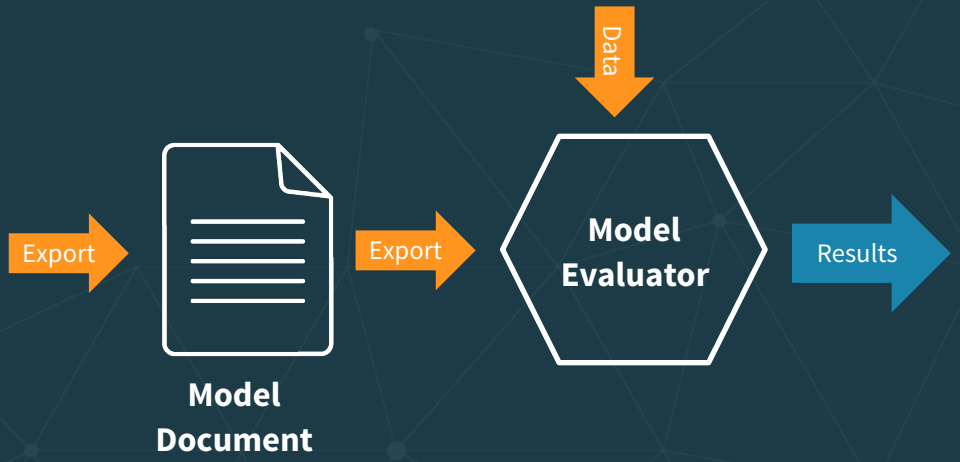
Impedance mismatch

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

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 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 use them for scoring.

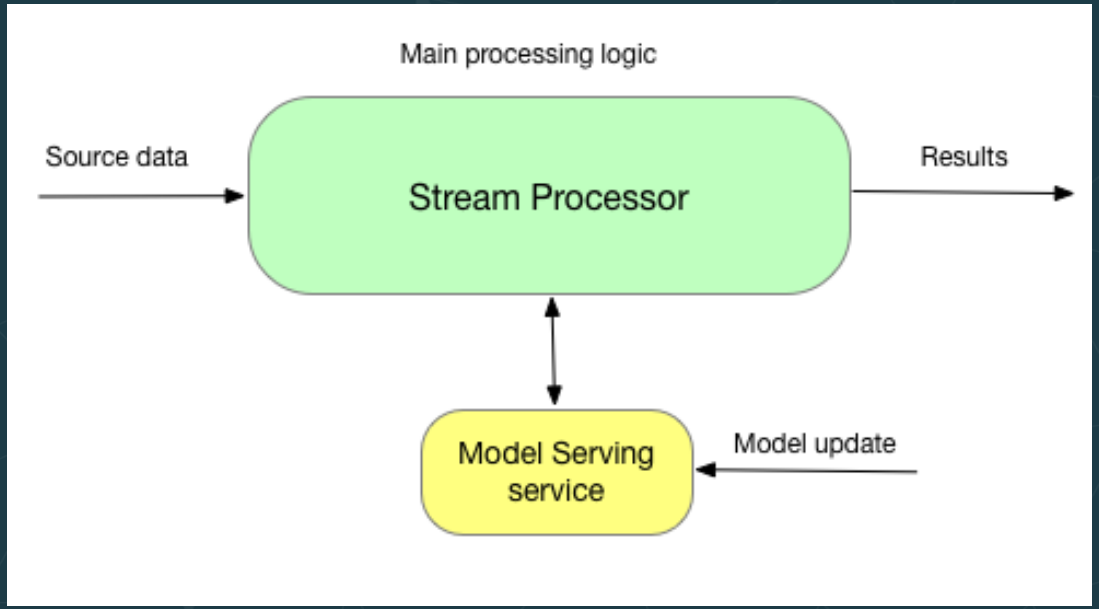
Additional Considerations – Model lifecycle

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

Traditional model serving implementation

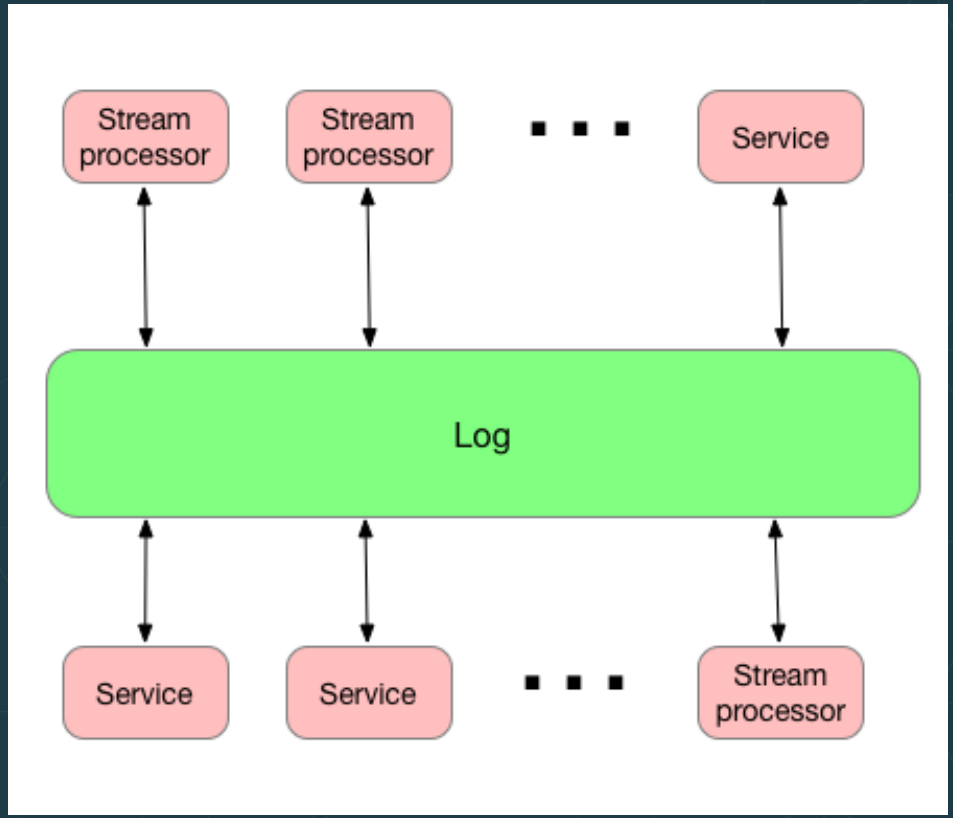
Models are deployed separately from stream processing:

- Tensorflow serving
- Clipper
- Model Server Apache MXNet
- DeepDetect
- TensorRT



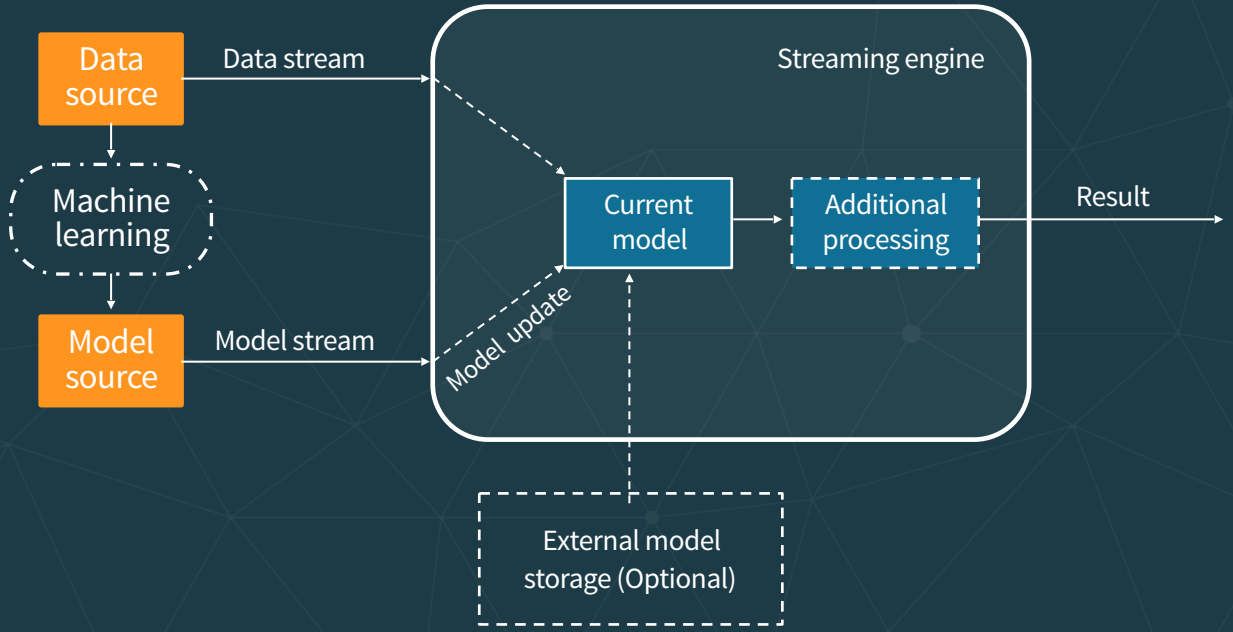
Log driven enterprise

- Complete decoupling of services interactions
- All communications are going through the log rather then services talking to each other
- Stream processors do not need to explicitly talk to the services, all updates are directly available in the log



Model serving in a log driven enterprise

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



Model representation

On the wire

```
syntax = “proto3”;

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

Internal

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

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

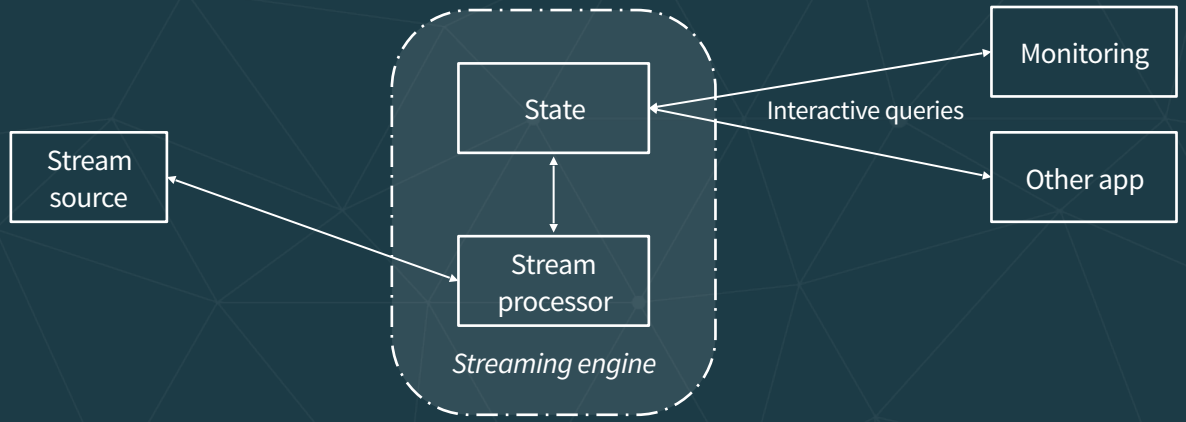
Additional considerations monitoring

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.

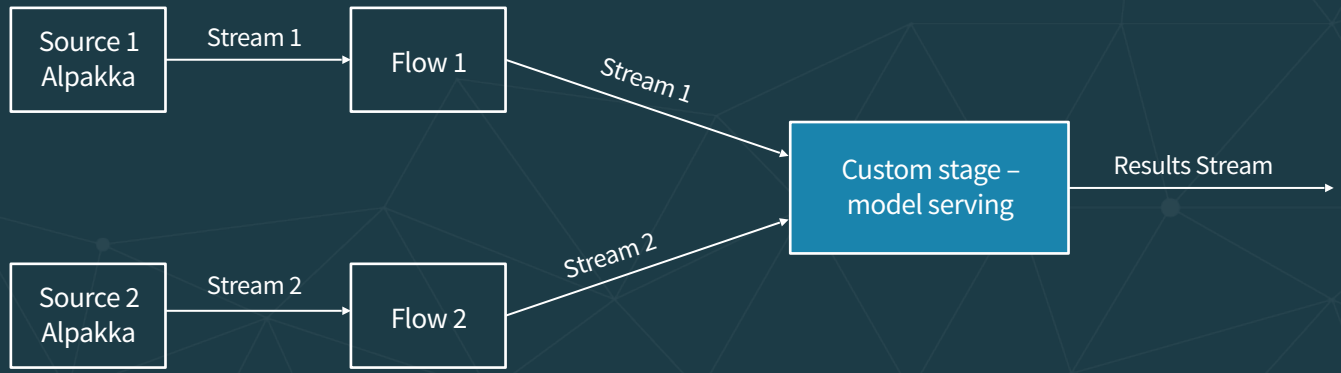
Decision criteria

- Using an SPE is a good fit for applications that require features provided out of the box by such engines. But you need to adhere to its programming and deployment models.
- A SPLs provide a programming model that allows developers to build the applications or micro services the way that fits their precise needs and deploy them as simple standalone Java applications.

- Akka Streams, part of the Akka project, is a library focused on in process back-pressured reactive streaming.
- Provides a broad ecosystem of connectors to various technologies (data stores, message queues, file stores, streaming services, etc) - Alpakka
- In Akka Streams computations are written in graph-resembling domain-specific lanauge (DSL), which aims to make translating graph drawings to and from code simpler.

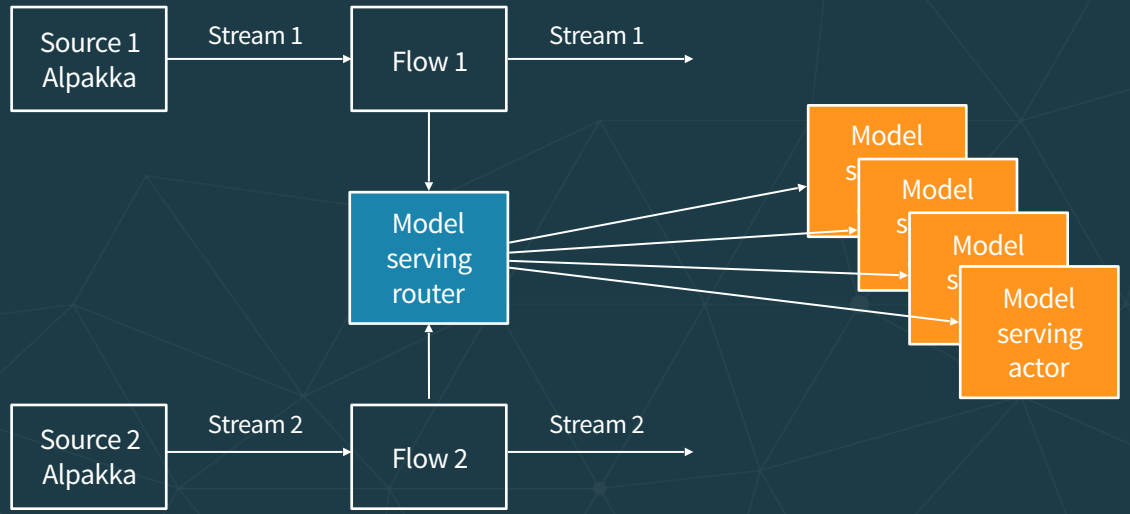
Using custom stage

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



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



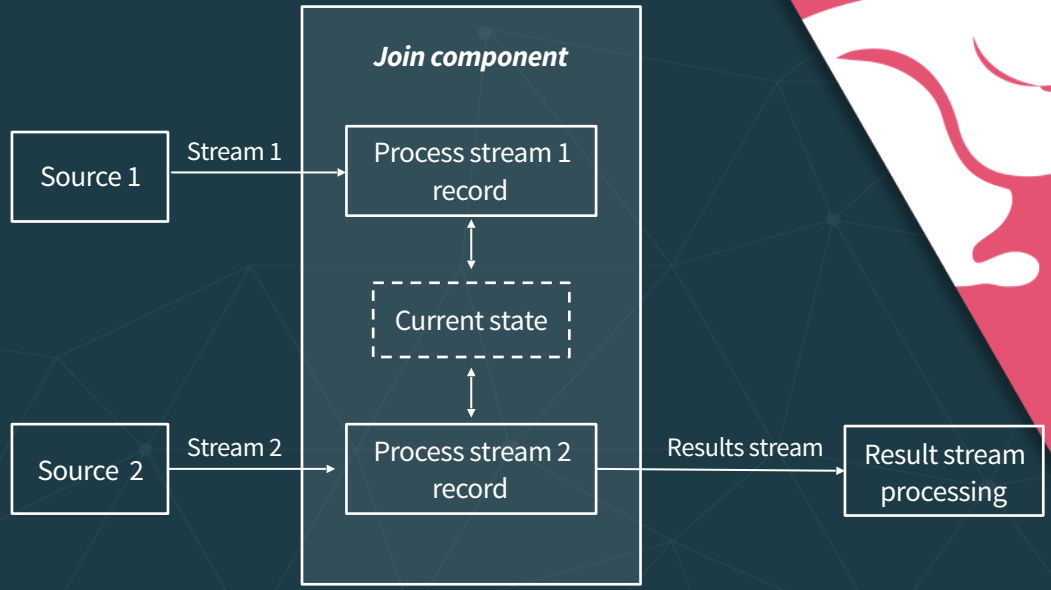
Flink

Flink is an open source stream-processing engine (SPE) that provides the following:

- Scales well, running on thousands of nodes.
- Provides powerful checkpointing and save pointing facilities that enable fault tolerance and restart ability.
- Provides state support for streaming applications, which allows minimization of usage of external databases for streaming applications.
- Provides powerful window semantics, allowing you to produce accurate results, even in the case of out-of-order or late-arriving data

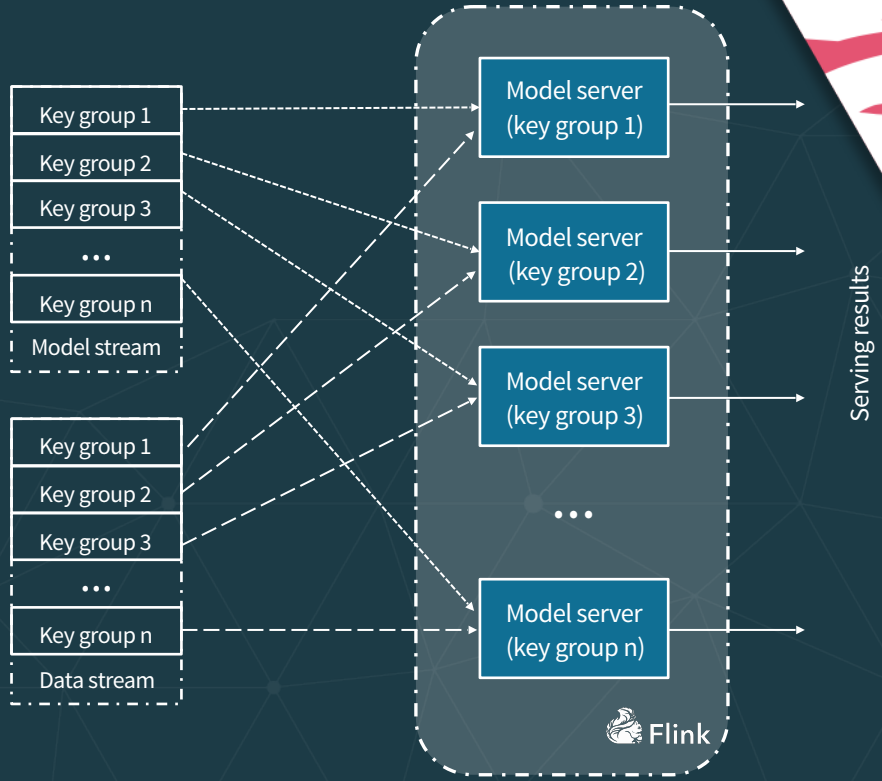
Flink Low Level Join

- Create a state object for one input (or both)
- Update the state upon receiving elements from its input
- Upon receiving elements from the other input, probe the state and produce the joined result



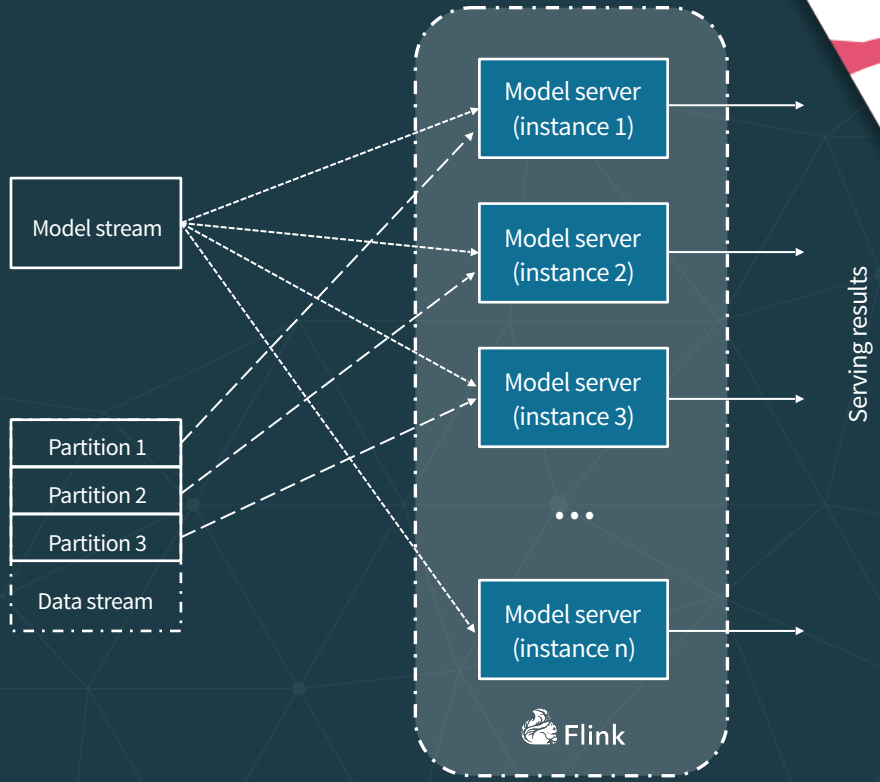
Key based join

Flink's *CoProcessFunction* allows key-based merge of 2 streams. When using this API, data is key-partitioned across multiple Flink executors. Records from both streams are routed (based on key) to the appropriate executor that is responsible for the actual processing.



Partition based join

Flink's *RichCoFlatMapFunction* allows merging of 2 streams in parallel (based on parralelization parameter). When using this API, on the partitioned stream, data from different partitions is processed by dedicated Flink executor.



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**

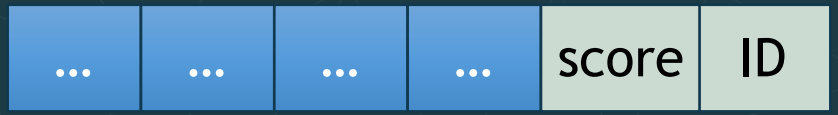
Model tracking

- 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 output records with the model ID, as well as the score.
  - Input Record

  - Output Record with Score, model version ID



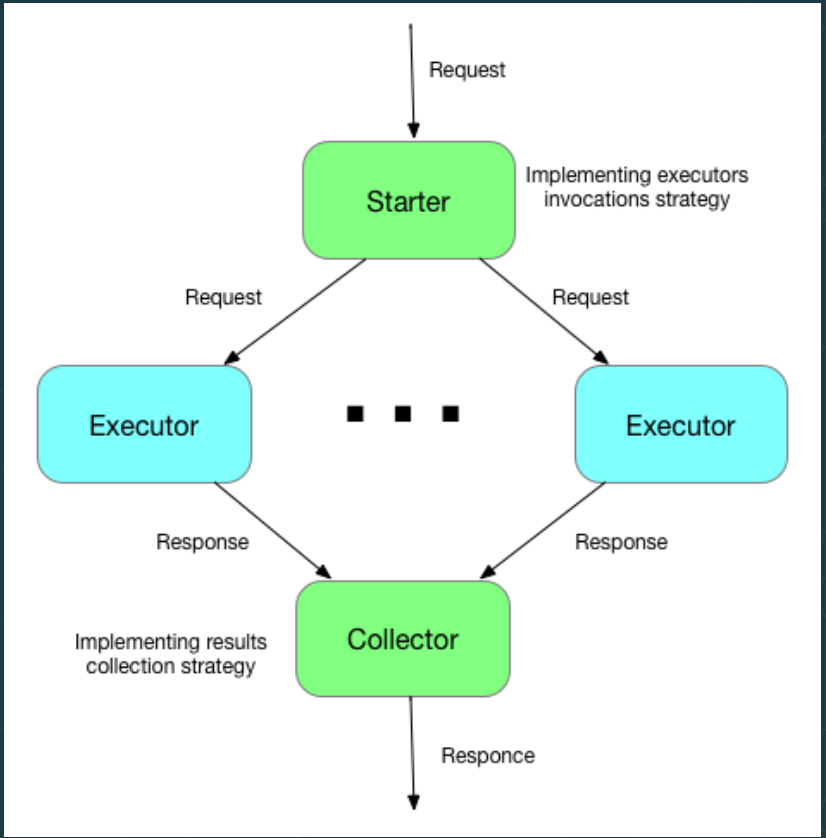
Speculative execution

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.

General Architecture for speculative execution

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



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??

Applicability for model serving

- Used to guarantee execution time
  - Several models:
    - A smart model, but takes time *T1*
    - A “less smart”, but fast model with a fixed upper-limit on execution time, *T2* << *T1*
  - If timeout (*T* > *T2*) occurs before smart finishes, return the less accurate result
  - …

Applicability for model serving

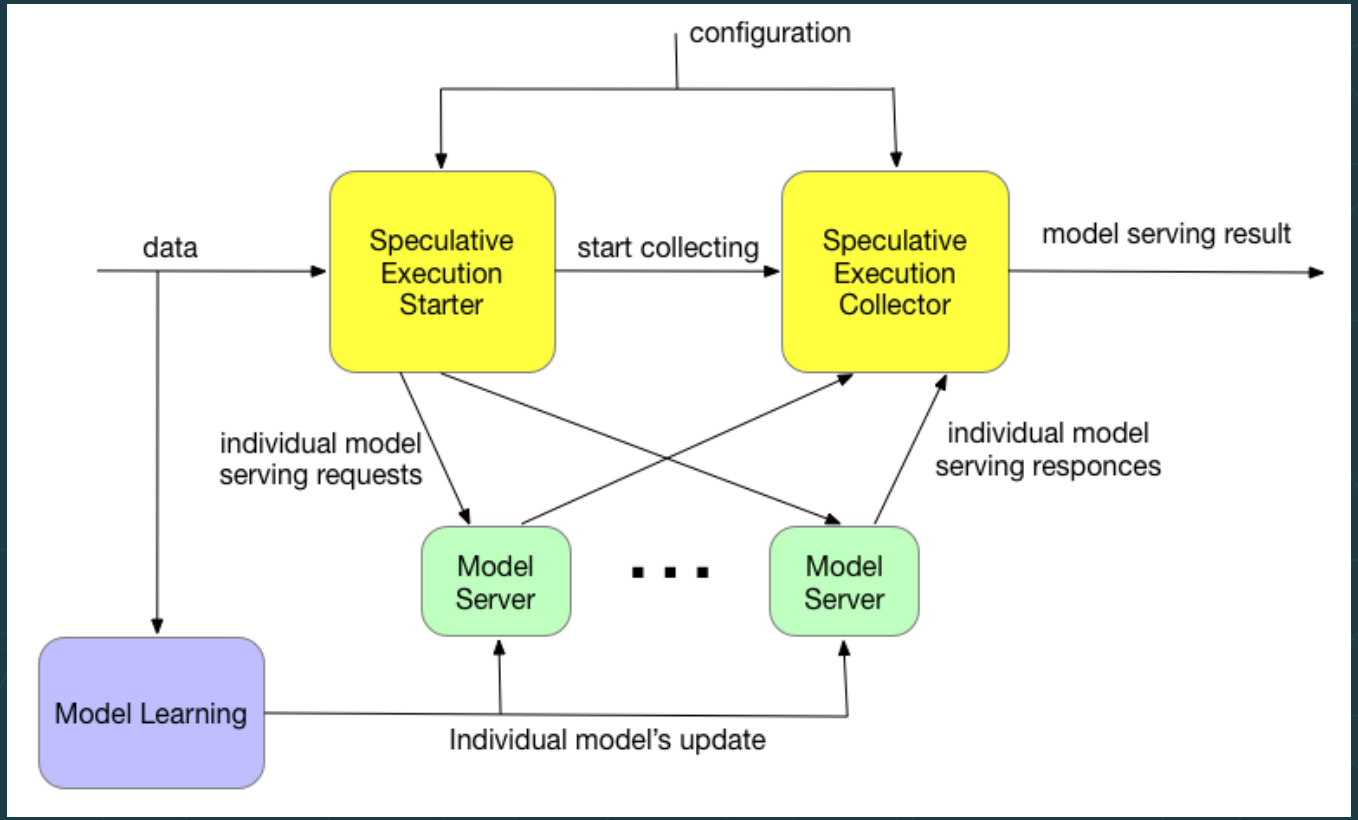
- …
- Consensus based model serving
  - If we have 3 or more models, score with all of them and return the majority result
- …

Applicability for model serving

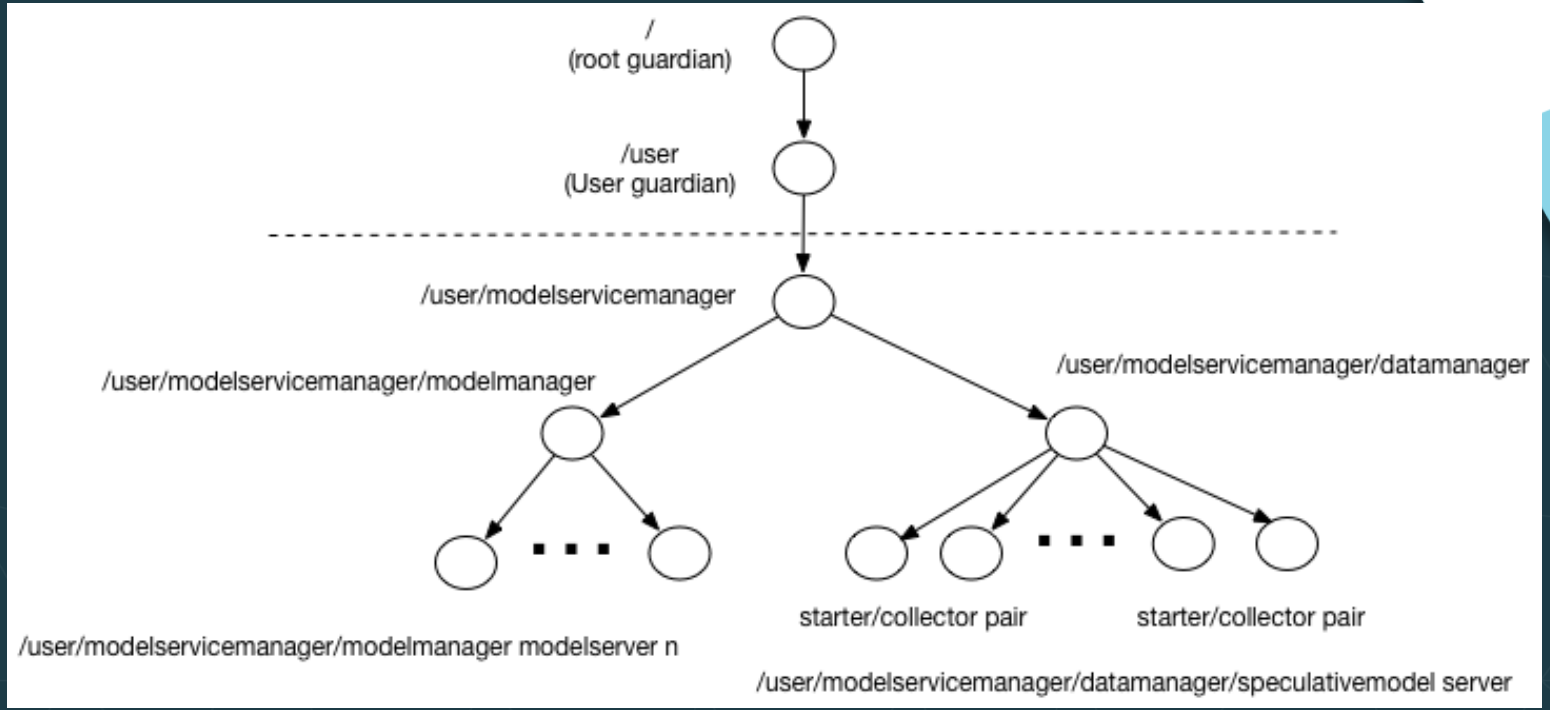
- …
- Quality based model serving.
  - If we have a quality metric, pick the result with the best result.

- Of course, you can combine these techniques.

Speculative Model Serving Architecture

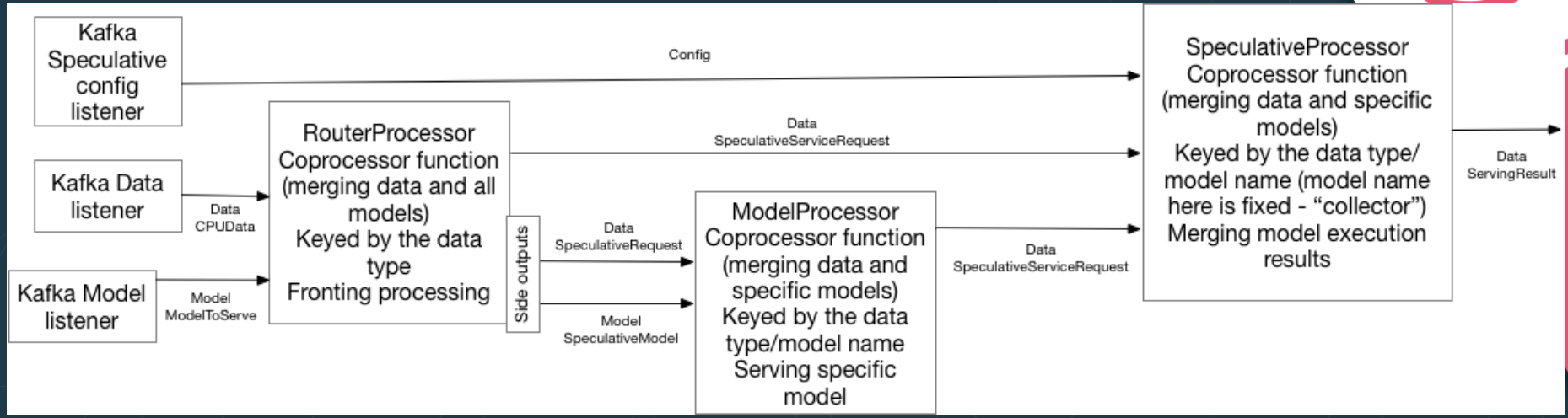


Akka Streams



Flink Implementation

- Router processor implements starter
- Model processor
- Speculative processor implements collector



Want to learn more ?

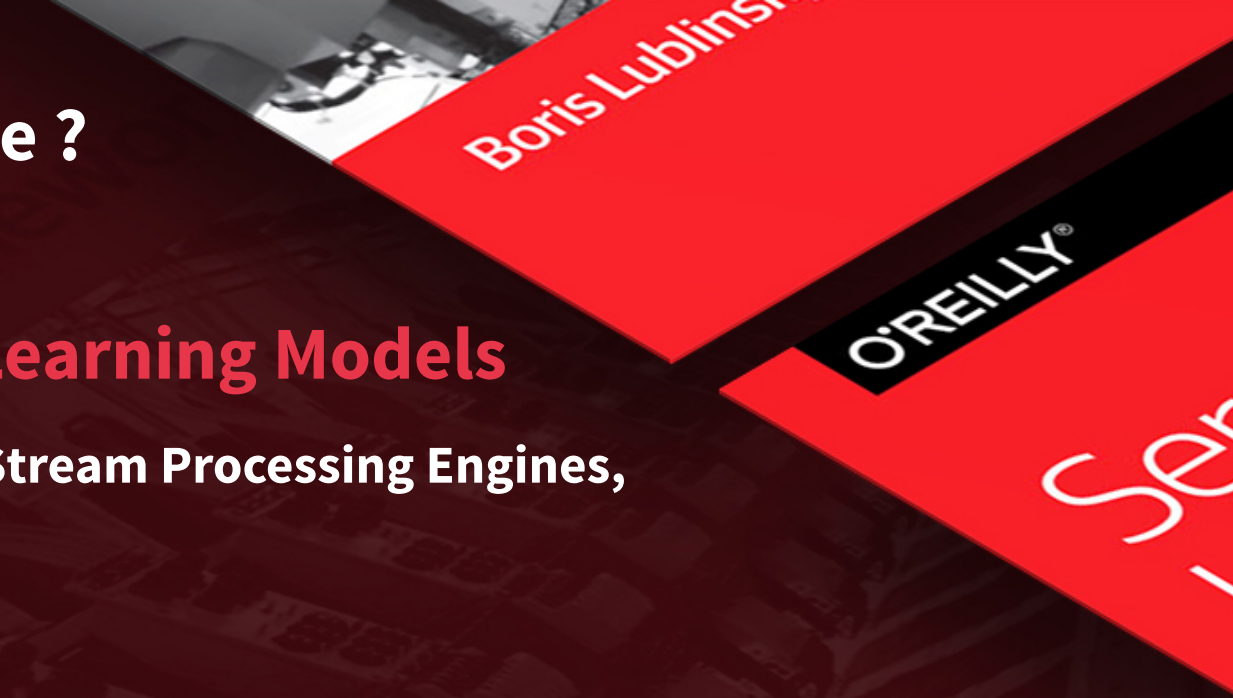
O'REILLY®

Serving Machine Learning Models

A Guide to Architecture, Stream Processing Engines, and Frameworks

By Boris Lublinksy

GET YOUR FREE COPY



Thank You

Any questions?