

Hands-on Machine Learning with Kafka-based Streaming Pipelines

Strata, San Francisco, 2019

Boris Lublinsky and Dean Wampler, Lightbend

boris.lublinsky@lightbend.com

dean.wampler@lightbend.com



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

<https://github.com/lightbend/model-serving-tutorial>

See the README for setup instructions.

These slides are in the presentation folder.

Outline

- Hidden technical debt in machine learning systems
- Model serving patterns
 - Embedding - models as code
 - Models as data
 - External services
 - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

As we'll see dynamically controlled streams will revisit models as data

But first, introductions...

Outline

- Hidden technical debt in machine learning systems
- Model serving patterns
 - Embedding - models as code
 - Models as data
 - External services
 - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

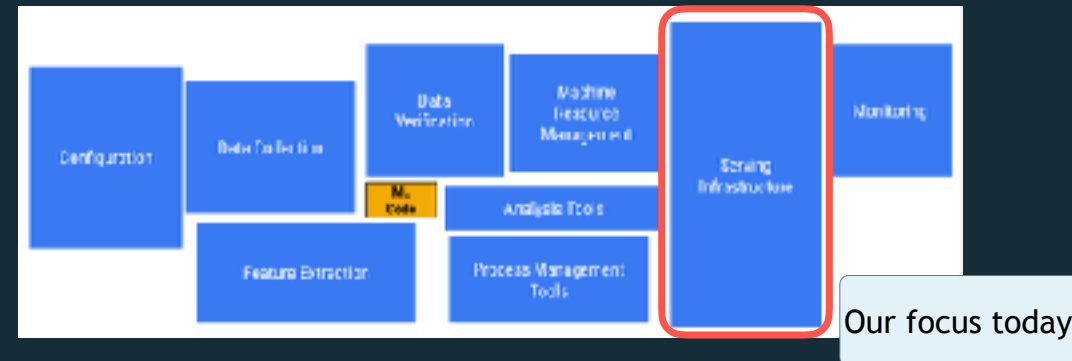
ML vs. Infrastructure Code



papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

This paper discusses the challenges of real-world use of ML. The actual ML code is just a small part of the overall infrastructure and capabilities required.

ML vs. Infrastructure Code



papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

This tutorial discusses a few aspects of that picture, focusing on architectures for *serving* models in productions. The architectures use minimal coupling to tool chains for *training* (e.g., through a Kafka topic), which enables a wide range of *training* options, making it easier to use your favorite data science tool chain for training.

Outline

- Hidden technical debt in machine learning systems
- Model serving patterns
 - Embedding - models as code
 - Models as data
 - External services
 - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

Model Serving Architectures

- Embedding - model as *code*, deployed into a stream engine
- Model as *data* - easier dynamic updates
- *Model Serving as a service* - use a separate service, access from the streaming engine
- *Dynamically controlled streams* - one way to implement model as data in a streaming engine

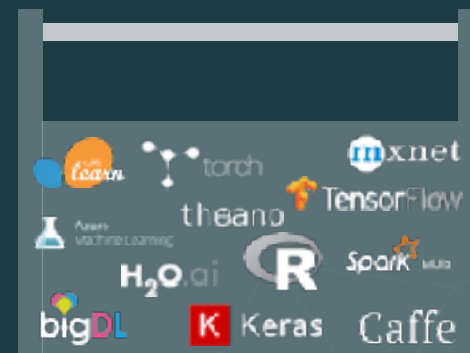
You can embed code for models, but we'll see this has many disadvantages. It's better to treat models as data, which allows you to update it as the “world” changes. We'll focus on two approaches for models as data.

Embedding: Model as Code

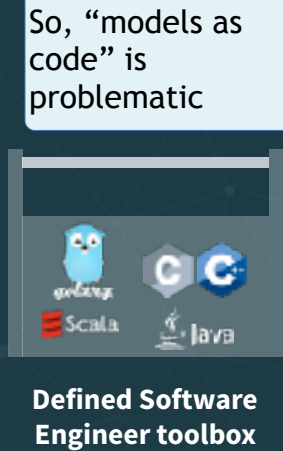
- Implement the model as source code
- The model code is linked into the streaming application at build time

Why is this problematic?

Impedance Mismatch



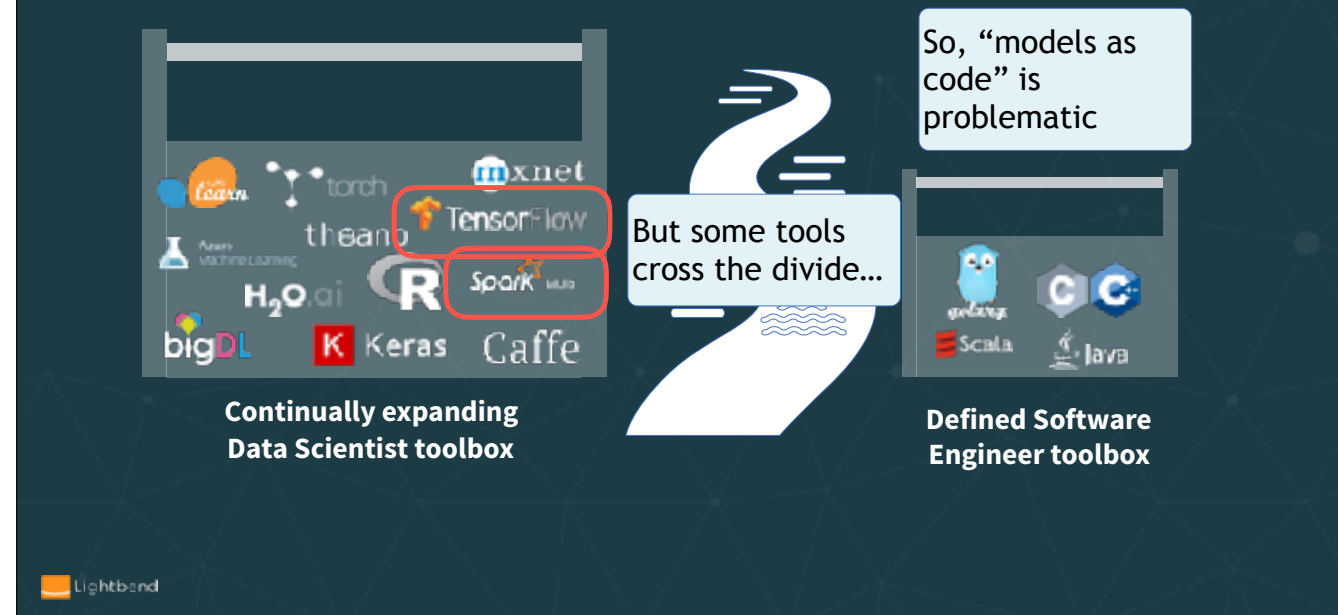
Continually expanding
Data Scientist toolbox



So, “models as
code” is
problematic

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.

Impedance Mismatch



We lose portability if we use tools crossing divide

Embedding: Model as Code

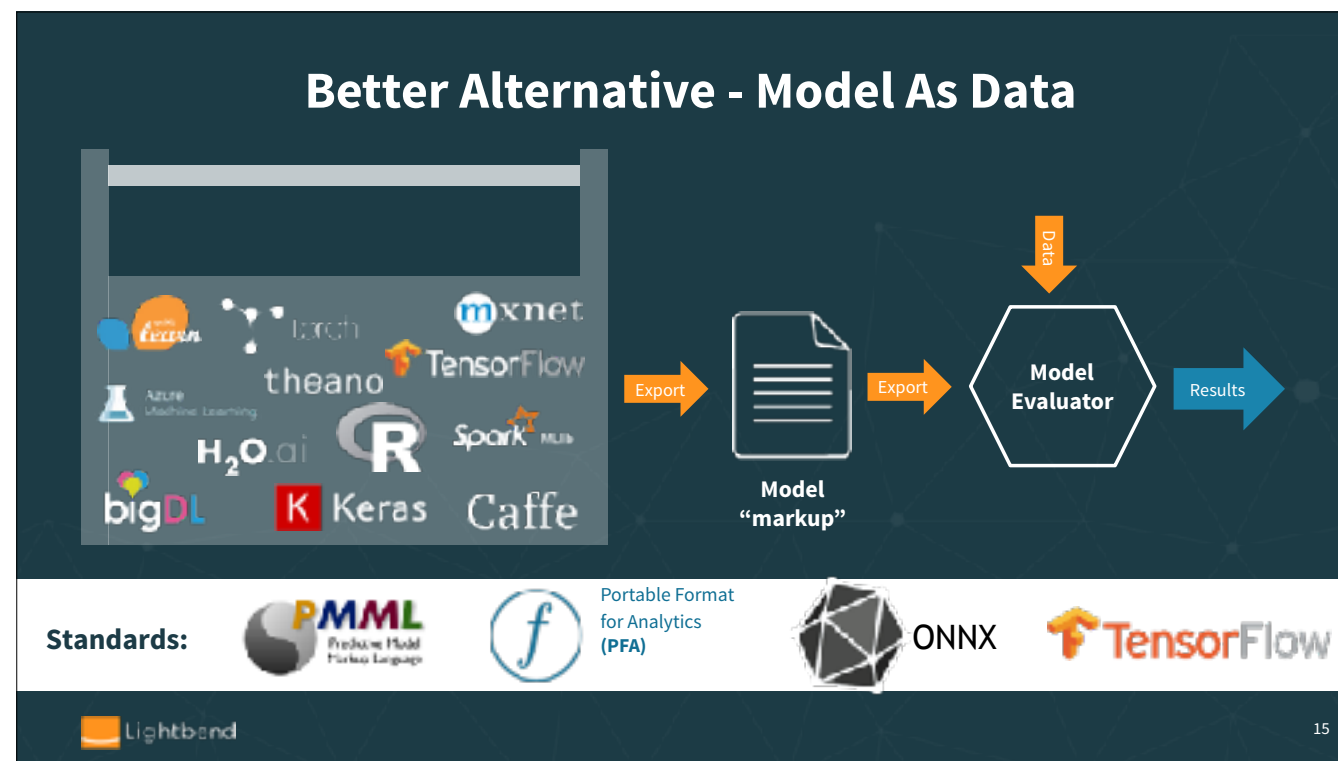
- It also *mostly* eliminates the possibility of updating the model at runtime, as the world changes*.

*Although some coding environments support dynamic loading of new code, do you really want to go there??

Most dynamically-typed languages, like Python, allow runtime loading of new code. Even the JVM supports this. However, it's much easier to introduce bugs, security holes (SQL injection attacks anyone??), etc. We'll also discuss other production concerns later, like auditing and data governance, which are harder to support using models as code.

Outline

- Hidden technical debt in machine learning systems
- Model serving patterns
 - Embedding - models as code
 - Models as data
 - External services
 - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up



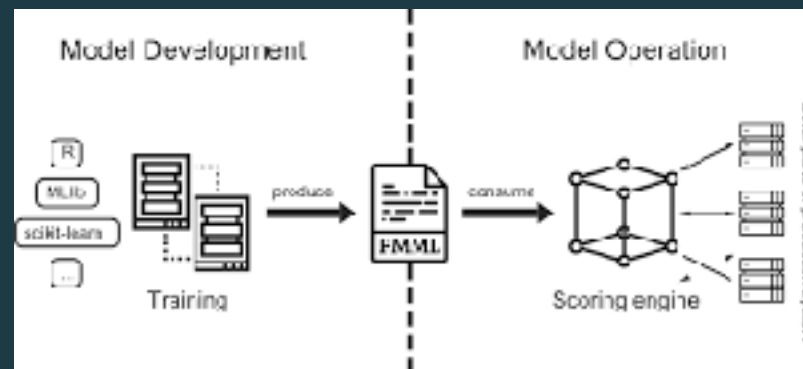
In order to overcome these differences, the Data Mining Group has introduced two standards - Predictive Model Markup Language (PMML) and Portable Format for Analytics (PFA), both suited for description of the models that need to be served. Introduction of these models led to creation of several software products dedicated to “generic” model serving, for example Openscoring, Open data group, etc.

ONNX is a new standard for deep learning models, but it is not supported by TensorFlow.

TensorFlow itself is a de facto standard for ML, because it is so widely used for both training and serving, even though it uses proprietary formats.

The result of this standardization is creation of the open source projects, supporting these formats - JPMML and Hadrian which are gaining more and more adoption for building model serving implementations, for example ING, R implementation, SparkML support, Flink support, etc. TensorFlow also released TensorFlow java APIs, which are used in a Flink TensorFlow

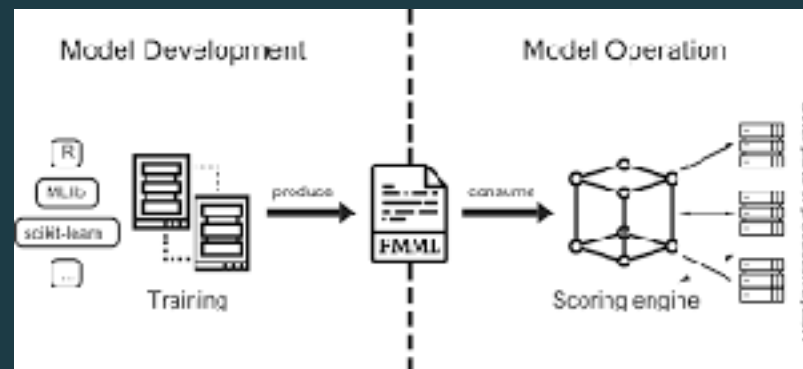
PMML



Predictive Model Markup Language (PMML) is an XML-based language that enables the definition and sharing of predictive models between applications.

<https://www.wismutlabs.com/blog/agile-data-science-2/>

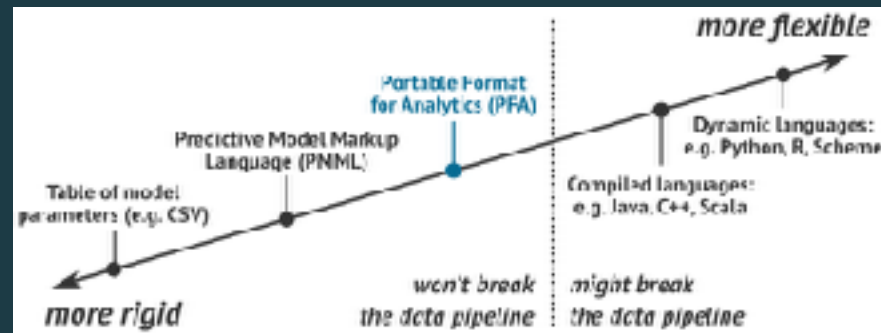
PMML



Implementations for:

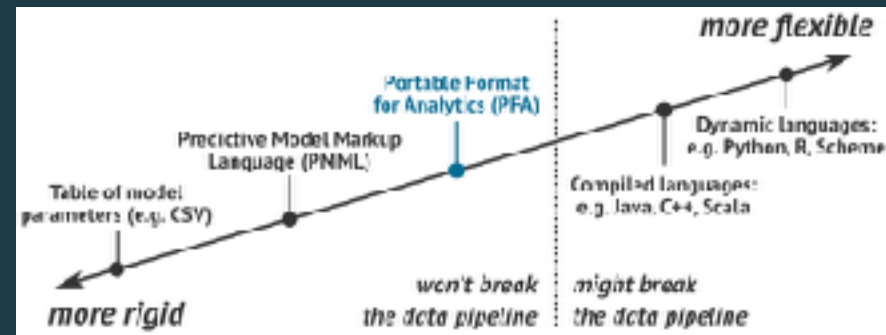
- Java ([JPMML](#)), R, Python [Scikit-Learn](#), Spark [here](#) and [here](#), ...

PFA



Portable Format for Analytics (PFA) is an emerging standard for statistical models and data transformation engines. PFA combines the ease of portability across systems with algorithmic flexibility: models, pre-processing, and post-processing are all functions that can be arbitrarily composed, chained, or built into complex workflows.

PFA



Implementations for:

- Java ([Hadrian](#)), R ([Aurelius](#)), Python ([Titus](#)), Spark ([Aardpfark](#)), ...

ONNX



Open Neural Networks Exchange (ONNX) is an open standard format of machine learning models to offer interoperability between various AI frameworks. Led by Facebook, Microsoft, and AWS.

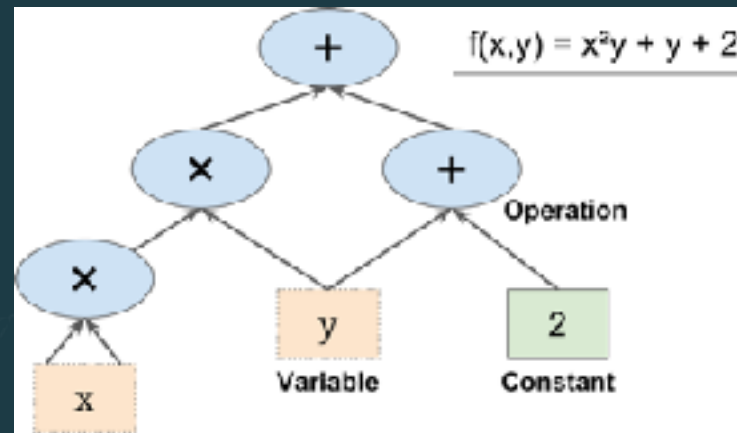
<https://azure.microsoft.com/en-us/blog/onnx-runtime-for-inferencing-machine-learning-models-now-in-preview/>

ONNX



- [Supported Tools](#) page.
- [Converters](#) for Keras, CoreML, LightGBM, Scikit-Learn,
- [PyTorch](#),
- third-party support for [TensorFlow](#)

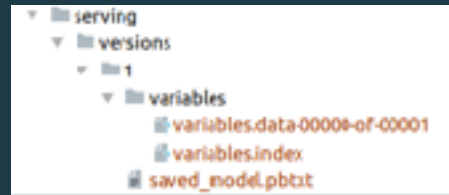
TensorFlow



- TensorFlow model is represented as a computational graph of Tensors.
- Tensors are defined as multilinear functions which consist of various vector variables. (i.e., generalization of 2x2 matrices)
- TensorFlow supports exporting graphs in the form of binary protocol buffers

Mathematicians might beg to differ on what a Tensor actually is...

TensorFlow Export Formats



SavedModel - Features:

- Multiple graphs sharing a single set of variables.
- Support for SignatureDefs
- Support for Assets

Normal (optimized) export of a TensorFlow Graph.

- Exports the Graph into a single file, that can be sent over Kafka, for example

Considerations for Interchange Tools

- Do your *training* tools support exporting with a standard exchange format, e.g., PMML, PFA, etc.?
- Do your *serving* tools support the same format for import?
- Is there support on both ends for the model types you want to use, e.g., random forests, neural networks, etc.?
- Does the *serving* implementation faithfully reproduce the results of your *training* environment?

Implementations are spotty, both in terms of import, export coverage, model types, and faithfully reproducing the same results on both ends. I.e., if you use Scikit-Learn to train a random forest, do you get the same results if you run it in SparkML, after exchanging with PMML?

Outline

- Hidden technical debt in machine learning systems
- Model serving patterns
 - Embedding - models as code
 - Models as data
 - External services
 - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

Model Serving as a Service

- *Advantages*
 - Simple integration with existing technologies and organizational processes
 - Easier to understand if you come from a non-streaming world
- *Disadvantages*
 - Worse latency: remote calls instead of local function calls
 - Coupling the availability, scalability, and latency/throughput of your streaming application with the SLAs of the service

Implementations are spotty, both in terms of import, export coverage, model types, and faithfully reproducing the same results on both ends. I.e., if you use Scikit-Learn to train a random forest, do you get the same results if you run it in SparkML, after exchanging with PMML?

Model Serving as a Service

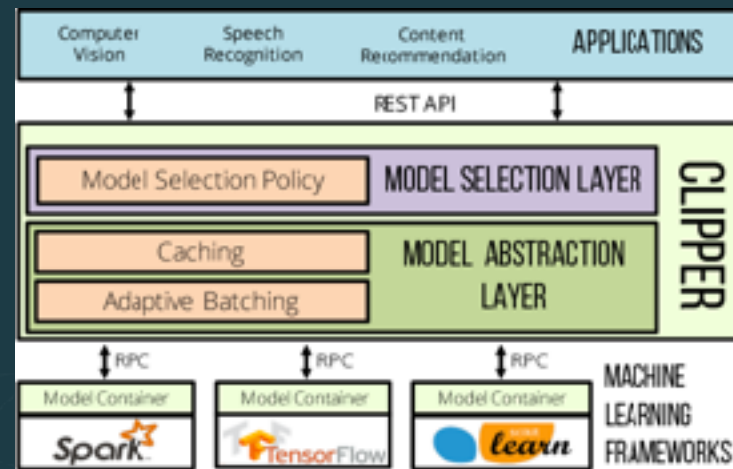
- *Advantages*
 - Simple integration with existing technologies and organizational processes
 - Easier to understand if you come from a non-streaming world
- *Disadvantages*
 - Worse latency: remote calls instead of local function calls
 - Coupling the availability, scalability, and latency/throughput of your streaming application with the SLAs of the service

Model Serving as a Service challenges

- Launch ML runtime graphs, scale up/down, perform rolling updates
- Infrastructure optimization for ML
- Latency optimization
- Connect to business apps via various APIs, e.g. REST, gRPC
- Allow Auditing and clear versioning
- Integrate into Continuous Integration (CI)
- Allow Continuous Deployment (CD)
- Provide Monitoring

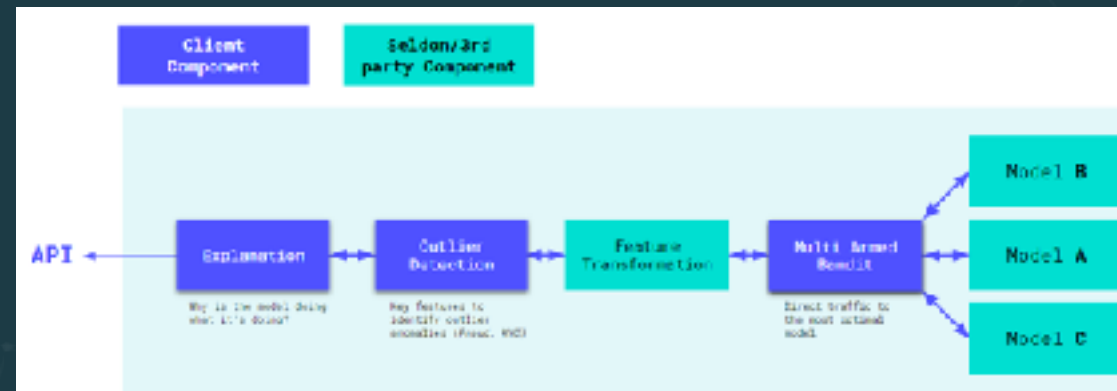
<https://github.com/SeldonIO/seldon-core/blob/master/docs/challenges.md>

Example: Clipper

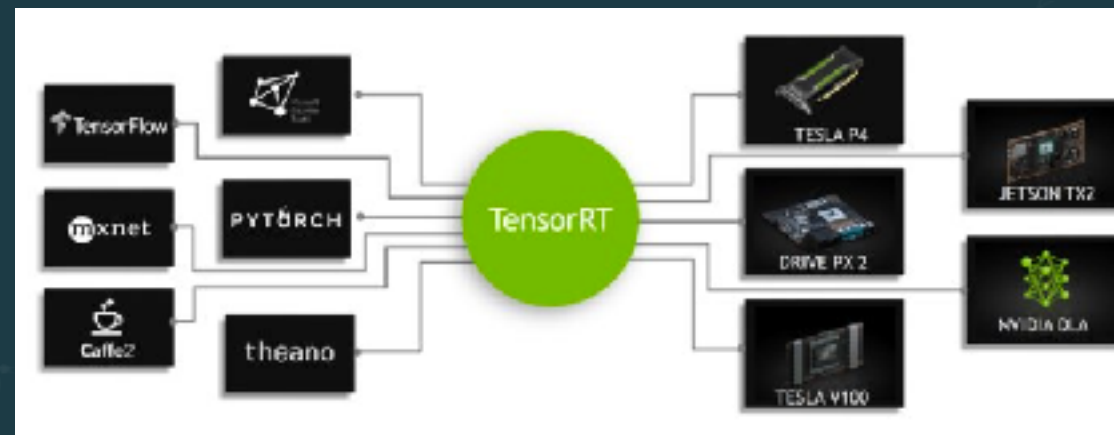


<https://www.semanticscholar.org/paper/Clipper%3A-A-Low-Latency-Online-Prediction-Serving-Crankshaw-Wang/4ef862c9157ede9ff8cfbc80a612b6362dcb6e7c>

Example: Seldon Core



Example: TensorRT



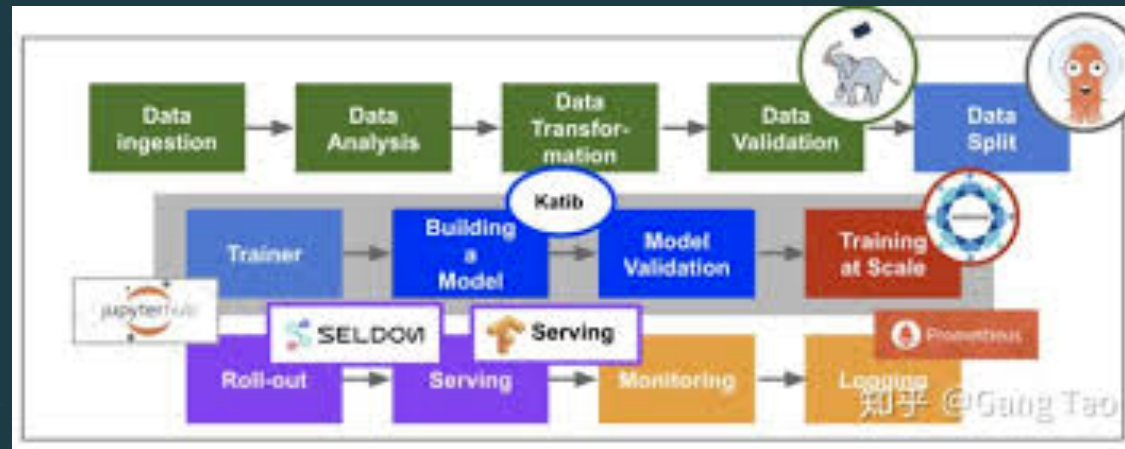
https://www.eetasia.com/news/article/Nvidia_CEO_in_China

Example: TensorFlow serving



<https://medium.com/sap-machine-learning-research/tensorflow-serving-in-enterprise-applications-our-experience-and-workarounds-part-1-33f65bfbf3d7>

Example: Kubeflow

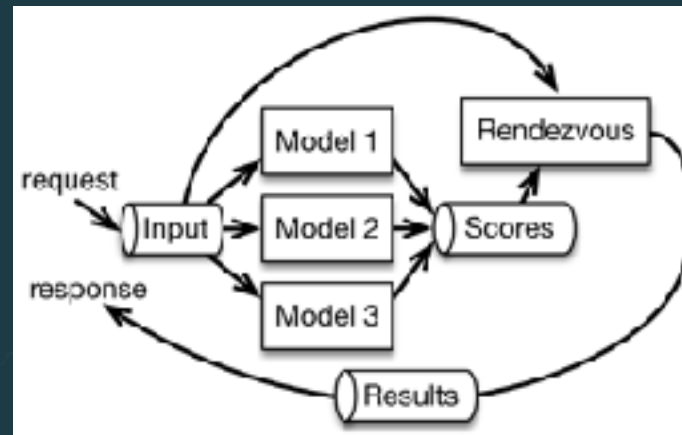


Rendezvous Architecture

Designed to handle the logistics of ML in a flexible, responsive, convenient, and realistic way. Specifically, it provides the following:

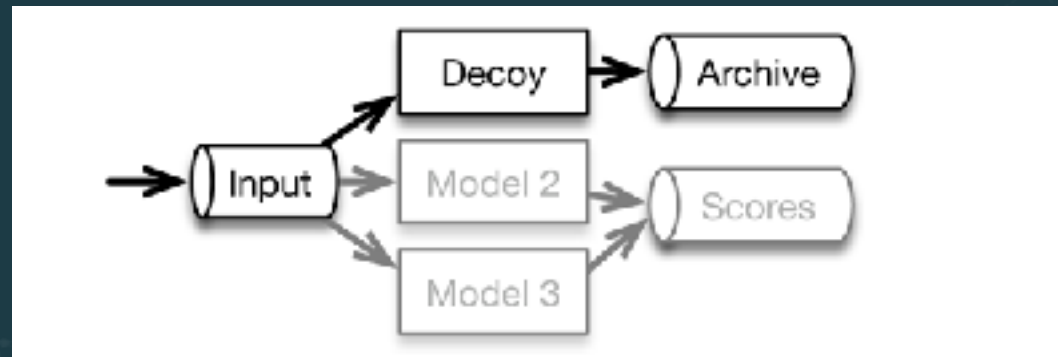
- Collect data at scale from a variety of sources and preserve raw data so that potentially valuable features are not lost.
- Make input and output data available to many independent applications (consumers), on premise, geographically distributed, or in the cloud.
- Manage multiple models during development and production.
- Improve evaluation methods for comparing models during development and production, including use of reference models for baseline successful performance.
- Have new models poised for rapid deployment.

Rendezvous Architecture



All communications here are Kafka (or its variants)

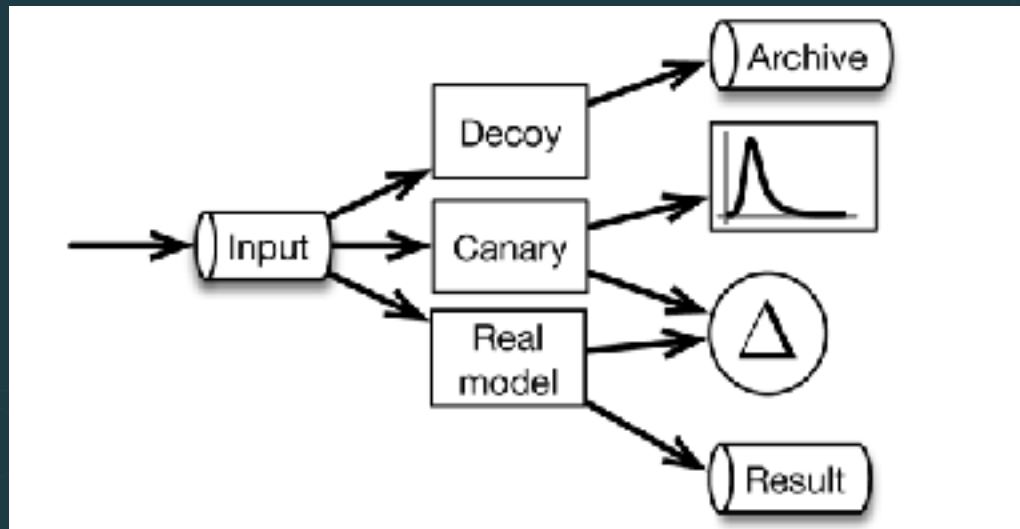
Rendezvous Architecture - Decoy



<https://mapr.com/ebooks/machine-learning-logistics/ch03.html>

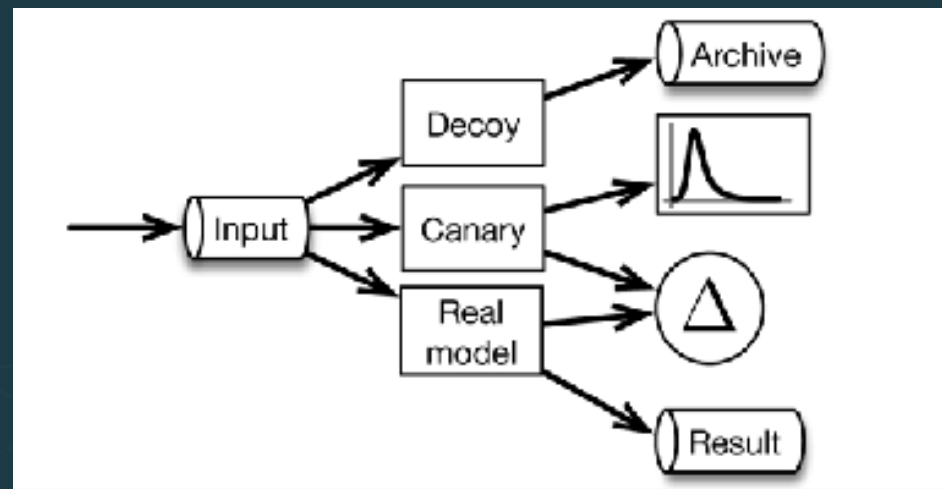
All communications here are Kafka (or its variants)

Rendezvous Architecture - Canary



All communications here are Kafka (or its variants)

Rendezvous Architecture - all capabilities




All communications here are Kafka (or its variants)

Outline

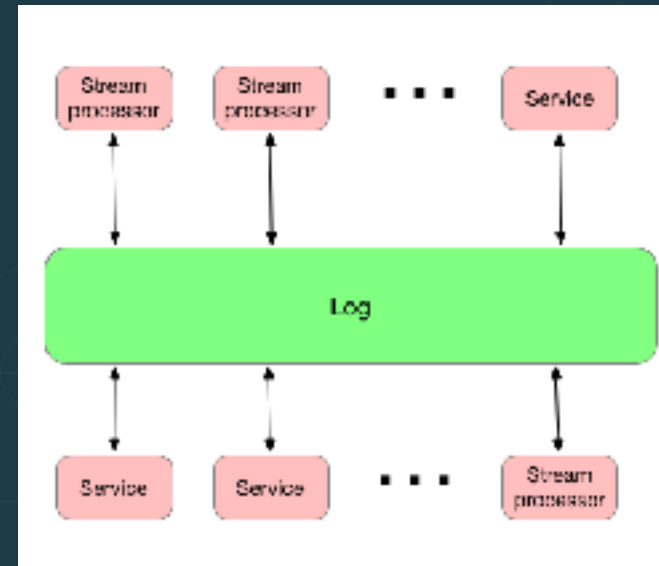
- Hidden technical debt in machine learning systems
- Model serving patterns
 - Embedding - models as code
 - Models as data
 - External services
 - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

Log-Driven Enterprise

- Complete decoupling of services.
- All communications go through the log rather than services talking to each other directly.
- Specifically, stream processors don't talk explicitly to other services, but send async. messages through the log.

Example: Kafka 

 Lightbend

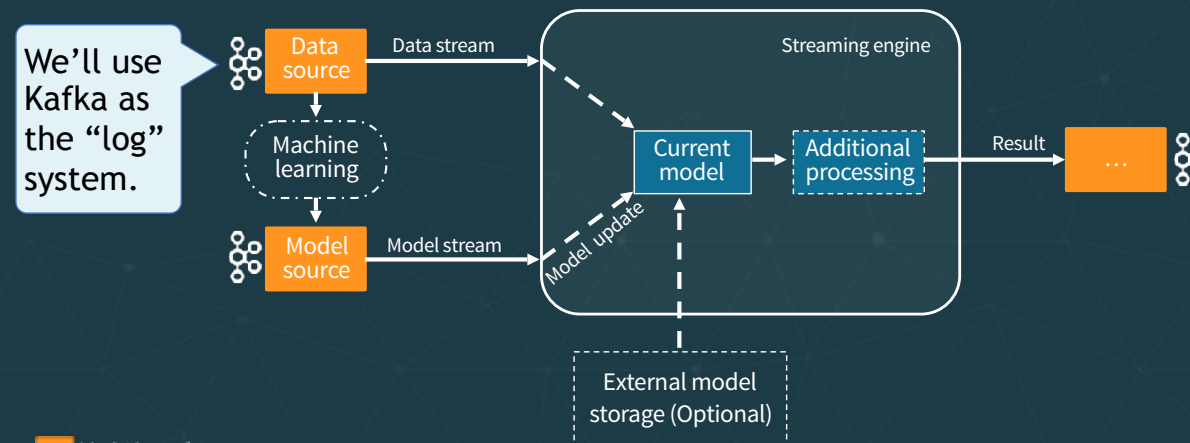


40

The majority of machine learning implementations are based on running model serving as a Rest service, which might not be appropriate for the high volume data processing or usage of the streaming system, which requires re coding/starting systems for model update, for example, Flink TensorFlow or Flink JPPML.

Model Serving in a Log-Driven Enterprise

A streaming system supporting model updates without interruption of execution
(dynamically controlled stream, additional data in Spark streaming).




The majority of machine learning implementations are based on running model serving as a Rest service, which might not be appropriate for the high volume data processing or usage of the streaming system, which requires re coding/starting systems for model update, for example, Flink TensorFlow or Flink JPPML. The name of this pattern was coined by *data Artisans* (that's the correct spelling...)

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;               // Could add PFA, ONNX, ...
  };
  ModelType modeltype = 4;
  oneof MessageContent {
    // Byte array containing the model
    bytes data = 5;
    string location = 6;
  }
}
```

See the “protobufs” project in the example code.



42

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 Code Abstraction (Scala)

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

[RECORD, RESULT] are
type parameters;
compare to Java:
<RECORD, RESULT>

See the “model” project
in the example code.

```
trait ModelFactory[RECORD, RESULT] {  
  def create(d : ModelDescriptor) : Option[Model[RECORD, RESULT]]  
  def restore(bytes : Array[Byte]) : Model[RECORD, RESULT]  
}
```

Corresponding Scala code that could be generated from the description, although we hand code this logic in the examples and exercises.

Production Concern: Monitoring

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

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



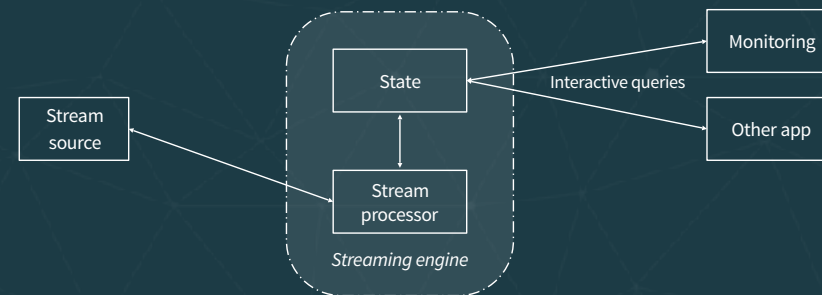
We'll return to production concerns...

44

Queryable State

Ad hoc query of the stream state. Different than the normal data flow.

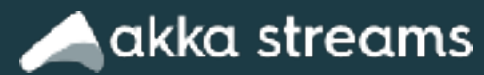
- Treats the stream as a lightweight *embedded database*.
- *Directly query the current state* of the stream.
- No need to materialize that state to a datastore first.



Note the “ad hoc” part. It’s for times when a “normal” stream of output data isn’t the best fit, e.g., only periodic updates are needed, you really want to support a range of “impromptu” (ad hoc) queries, etc.

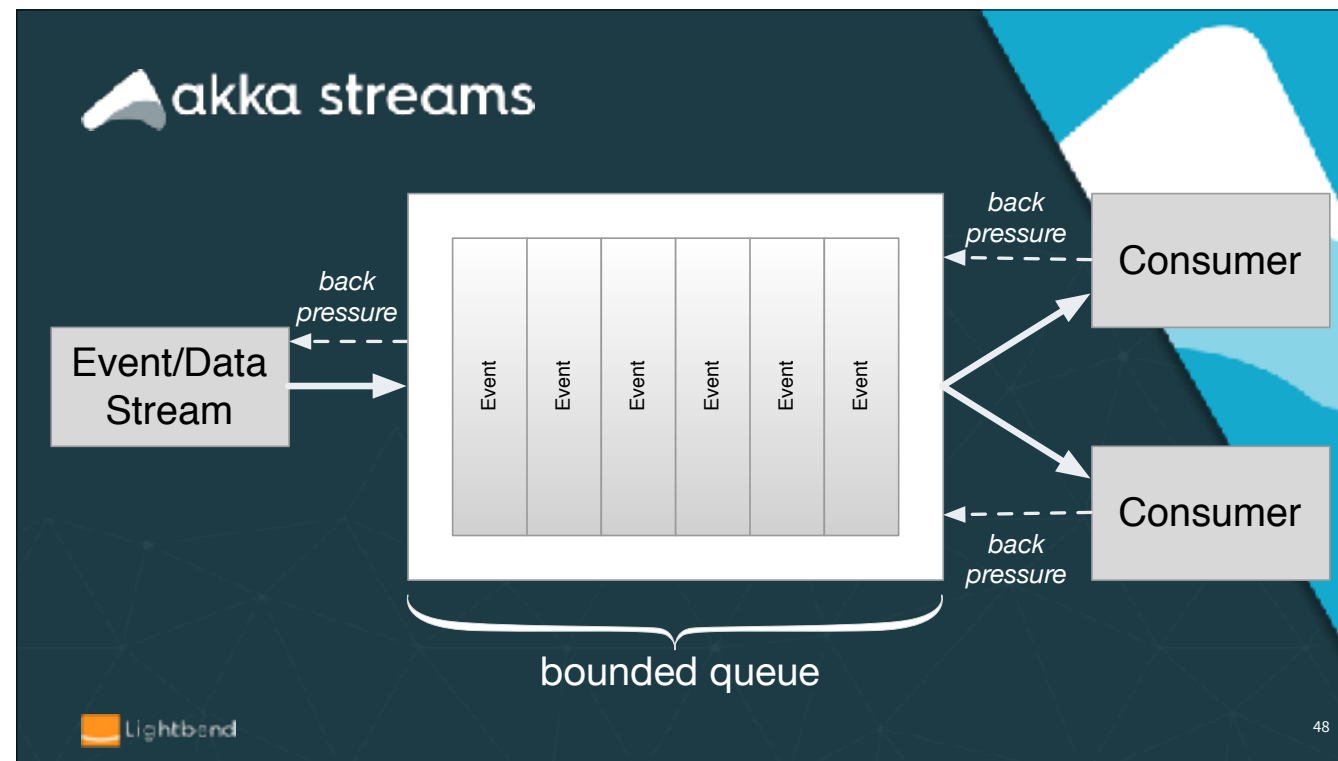
Kafka Streams and Flink have built-in support for this and it’s being added to Spark Streaming. We’ll show how to use other Akka features to provide the same ability in a straightforward way for Akka Streams.

Akka Streams

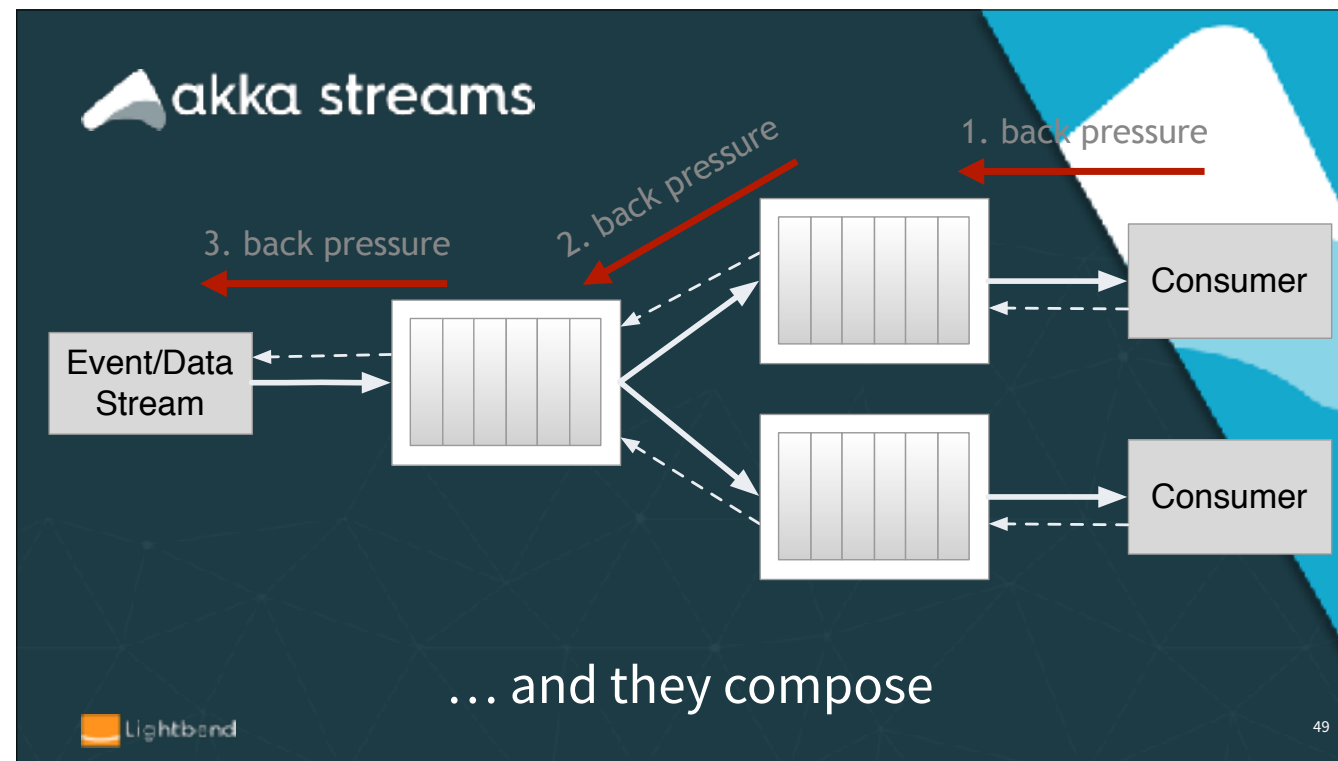


- *A library*
- Implements Reactive Streams.
- <http://www.reactive-streams.org/>
- *Back pressure* for flow control

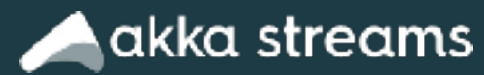
See this website for details on why *back pressure* is an important concept for reliable flow control, especially if you don't use something like Kafka as your "near-infinite" buffer between services.



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 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



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

- A very simple example to get the “gist”:
- Calculate the factorials for $n = 1$ to 10

```
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))
```

1
2
6
24
120
720
5040
40320
362880
3628800

52

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))
```

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

```
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))
```

Initialize and specify
now the stream is
"materialized"

1
2
6

5040
40320
362880
3628800

```
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(DigInt(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

1
2
6

40320
362880
3628800

Source



```
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 (a flow)

1
2
6
24
120

362880
3628800

Source



Flow


```
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, n) => acc * n }
factorials.runWith(Sink.foreach(println))
```

Output to a sink,
and run it

1
2
6
24
120
720
5040
40320
362880
3628800

Source

Flow

Sink

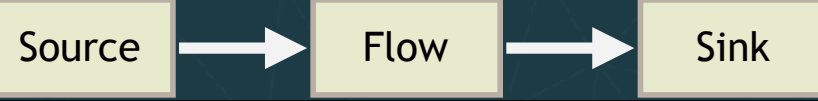
```
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._

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

1
2
6
24
120
720
0
20
880
3528800

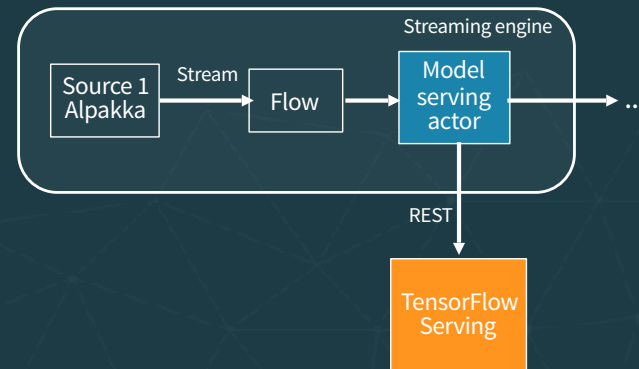
A source, flow, and sink constitute a graph



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

Using TensorFlow Serving in Akka Streams

Use Custom Actor to access TensorFlow Serving Server



Use the same routing layer idiom: an actor that will implement model serving for a specific model (based on some key) and route messages appropriately to the external service. This way our system will serve models in parallel.

Code Time

- Open the example code project
- We'll walk through the project at a high level
- Familiarize yourself with the *tensorflowserver* code
- Load and start the TensorFlow model serving Docker image
 - See [Using TensorFlow Serving](#) in the README
- Try the implementation and see if you have any questions

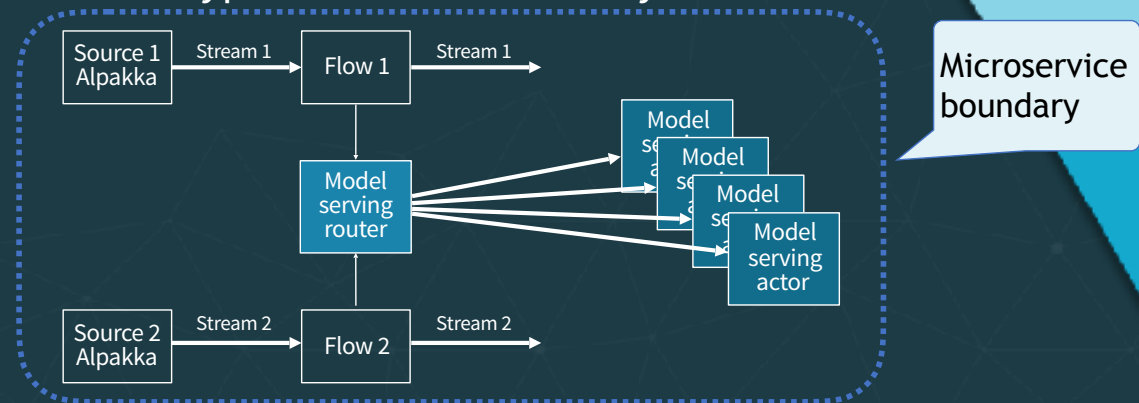
Model Serving from an Akka Streams App

- How do we integrate model serving (or any other new stateful capability) into an Akka Streams app?
- Make asynchronous calls to Akka Actors to do anything you want and keep the state
 - We'll discuss Actors that implement model serving within the microservice boundary (i.e., with a library)
 - Actors could also call an external service, like TensorFlow Serving (not shown)

We provide two implementations. You could generalize the second approach (async calls) to invoke an external service. We won't provide examples of this option, but return later with some additional considerations about it.

Using Invocations of 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!!



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

Akka Streams Example

Code time

1. Run the *client* project (if not already running)
2. Explore and run *akkaServer* project

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 state

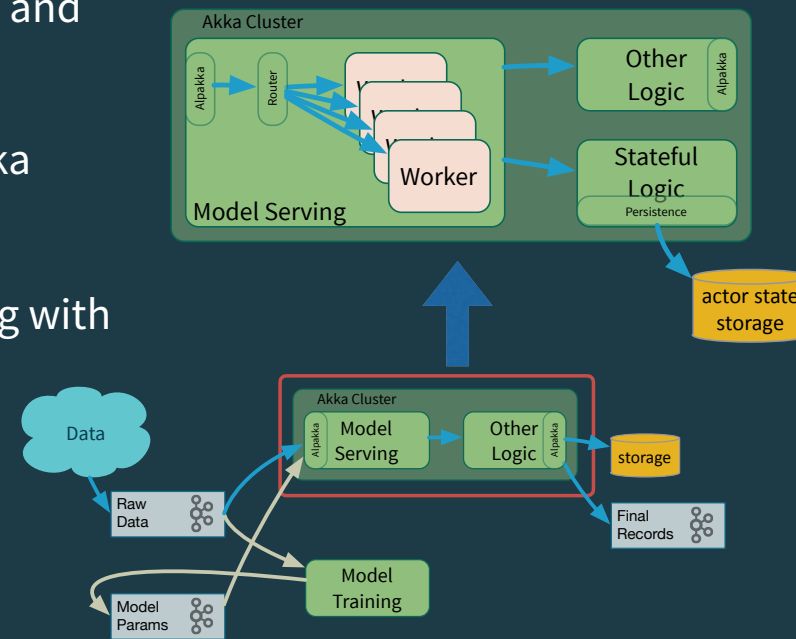
Curl or open in a browser:

<http://localhost:5500/models>

<http://localhost:5500/state/wine>

Handling Other Production Concerns with Akka and Akka Streams

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

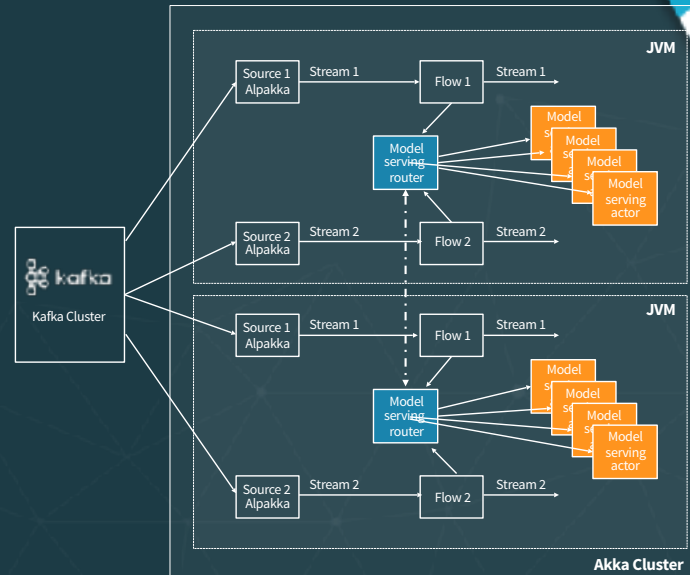


Here's our streaming microservice example adapted for Akka Streams. We'll still use Kafka topics in some places and assume we're using the same implementation for the "Model Training" microservice. Alpakka provides the interface to Kafka, DBs, file systems, etc. We're showing two microservices as before,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

- Kafka partitioned topic; add partitions and corresponding listeners.
- Akka cluster sharing: split model serving actor instances across the cluster.



<http://michalplachta.com/2016/01/23/scalability-using-sharding-from-akka-cluster/>

67

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

Flink



Same sample use case, now with Kafka Streams

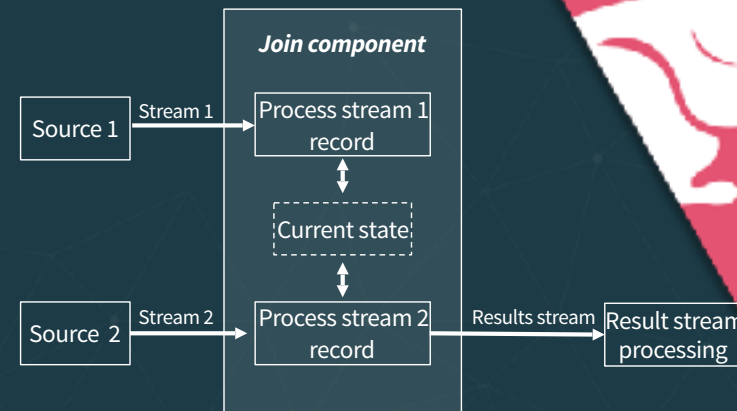


Flink is an open source stream-processing engine (SPE) that provides the following:

- Scales well, can run 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 minimizes the need for external databases for external access to the state.
- Provides powerful window semantics, enabling calculation of accurate results, even in the case of out-of-order or late-arriving data.

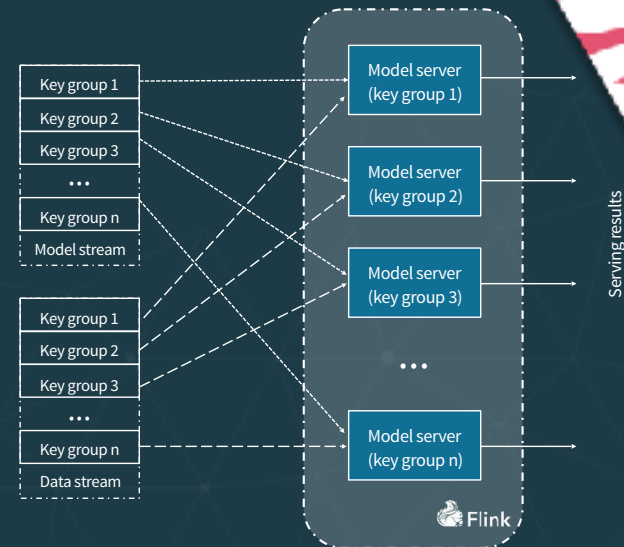
Flink Low Level Join

- 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.



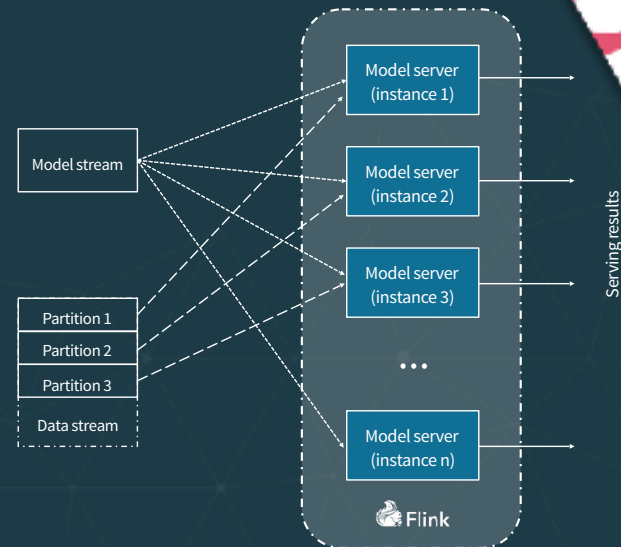
The main characteristics of this approach:

- Distribution of execution is based on key
- Individual models' scoring is implemented by a separate executor (a single executor can score multiple models), which means that scaling Flink leads to a better distribution of individual models and consequently better parallelization of scorings.
- A given model is always scored by a given executor, which means that depending on the data type distribution of input records, this approach can lead to "hot" executors

Based on this, key-based joins are an appropriate approach for the situations when it is necessary to score multiple data types with relatively even distribution.

Partition based join

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



Here are the main characteristics of this approach:

- The same model can be scored in one of several executors based on the partitioning of the data streams, which means that scaling of Flink (and input data partitioning) leads to better scoring throughput.
- Because the model stream is broadcast to all model server instances, which operate independently, some race conditions in the model update can exist, meaning that at the point of the model switch, some model jitter (models can be updated at different times in different instances, so for some short period of time different input records can be served by different models) can occur.

Based on these considerations, using global joins is an appropriate approach for the situations when it is necessary to score with one or a few models under heavy data load.

Flink Example

Code time

1. Run the *client* project (if not already running)
2. Explore and run *flinkServer* project
 - a. `ModelServingKeyedJob` implements keyed join
 - b. `ModelServingFlatJob` implements partitioned join

Spark Structured Streaming



Same sample use case, now with Kafka Streams

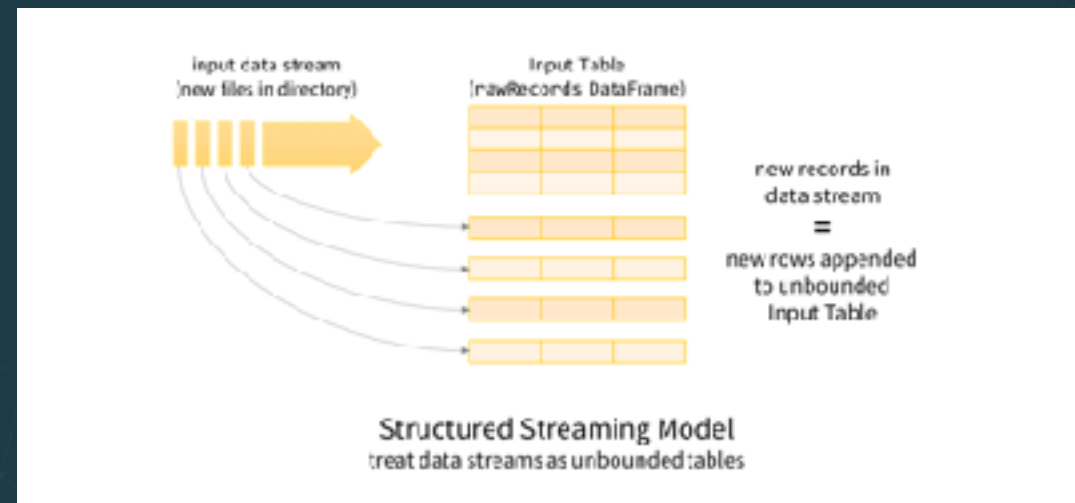
Spark Structured Streaming



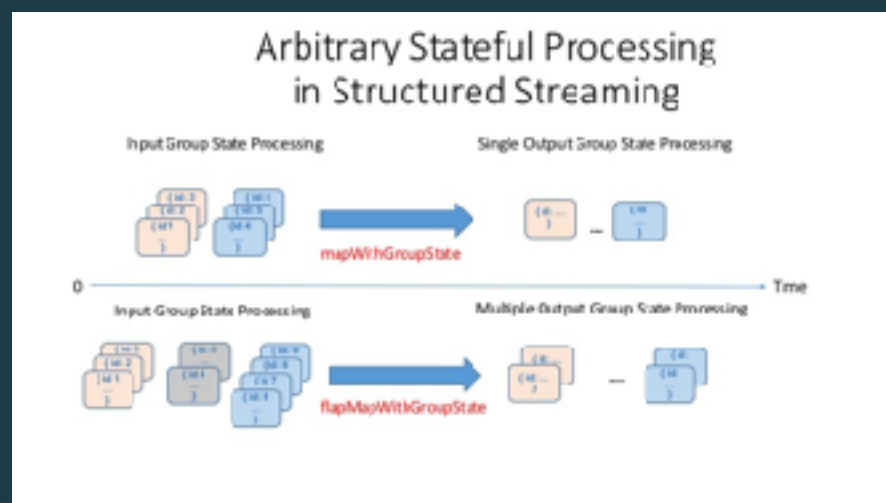
Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.

- Scales well, runs on thousands of nodes.
- Express your streaming computation the same way you would express a batch SQL computation on static data:
 - The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

Spark Structured Streaming

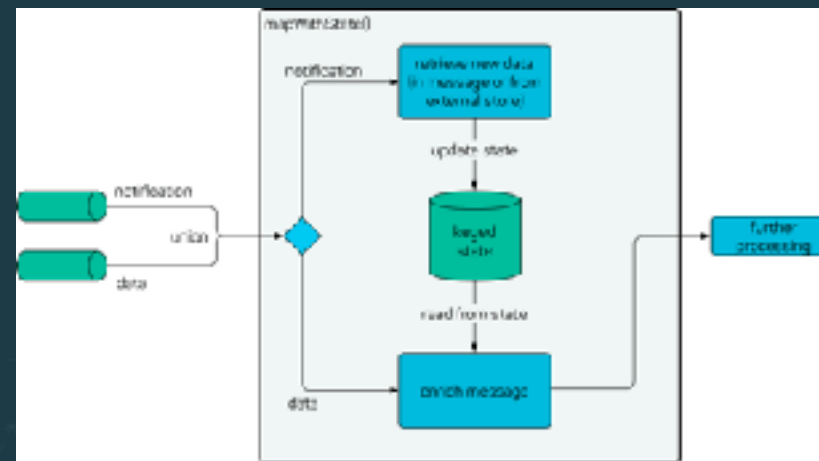


Spark Structured Streaming - State

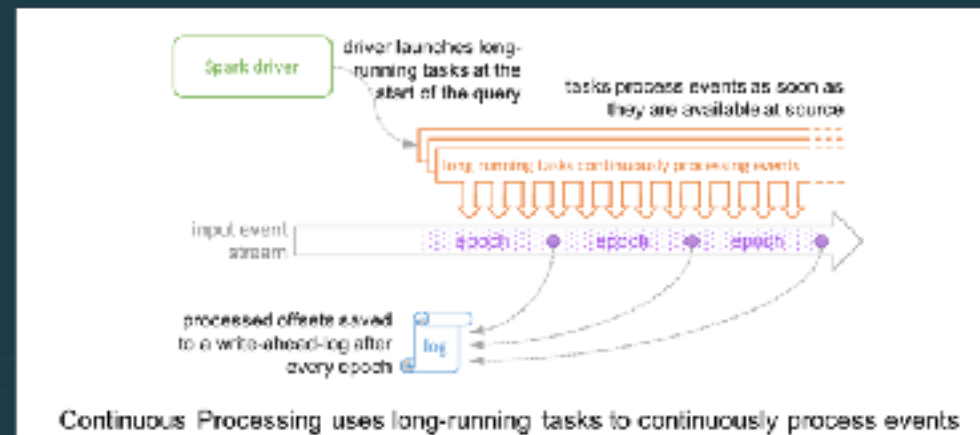


<https://databricks.com/blog/2017/10/17/arbitrary-stateful-processing-in-apache-sparks-structured-streaming.html>

Spark Structured Streaming - mapWithState

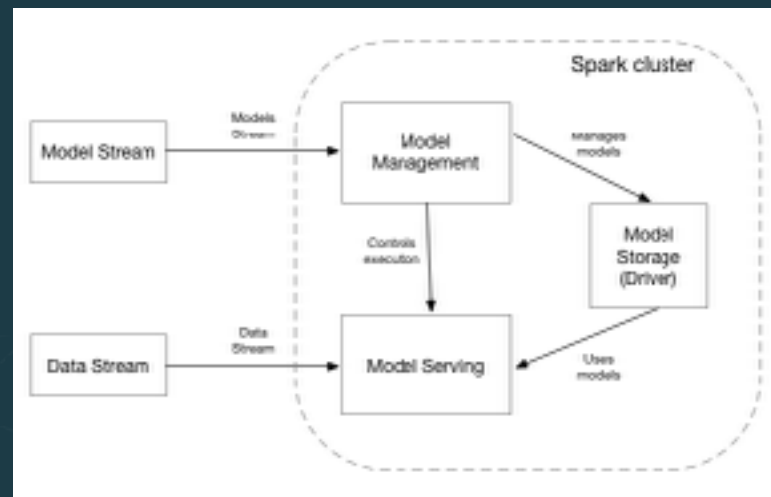


Spark Structured Streaming - continuous processing



<https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html>

Multi loop continuous processing



Spark Example

Code time

1. Run the *client* project (if not already running)
2. Explore and run *sparkServer* project
 - a. SparkStructuredModelServer implements using mapWithState.
 - b. SparkStructuredStateModelServer implements multi loop approach

Comparing implementation

1. Akka Streams with Akka is a framework (library) providing greater flexibility for implementation and deployment, but requires custom implementation for scalability and failover.
2. Both Flink and Spark Streaming are stream-processing engines (SPE) that 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.

Spark vs Flink

1. In Flink iterations are executed as cyclic data flows. This means that a data flow program (and all its operators) is scheduled just once and the data is fed back from the tail of an iteration to its head. This allows Flink to keep all additional data locally.
2. In Spark for each iteration a new set of tasks/operators is scheduled and executed. Each iteration operates on the result of the previous iteration which is held in memory, external to the execution and has to be moved to execution for every operations.

Spark vs Flink

1. Because in Flink all additional data can be kept locally, arbitrary complex structures can be used for its storage, although serializers are required for checkpointing. This serializers are only invoked out of band.
2. In Spark all the additional data is stored externally which means that it has to be marshalled/unmarshalled for every mini batch (for every message in continuous execution) to bring it to the execution.
3. Spark Structured Streaming is based on SQL data types, which makes data storage even more complex.

Outline

- Hidden technical debt in machine learning systems
- Model serving patterns
 - Embedding - models as code
 - Models as data
 - External services
 - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

Additional Production Concerns for Model Serving

- Implications of *models as data*
- Software process, e.g., CI/CD
- Speculative execution of models

Models as Data - Implications

- If models are data, they are subject to all the same *Data Governance* concerns as the data itself!
- Security and privacy considerations
- Traceability, e.g., for auditing
- ...

We'll just discuss these two aspects of the larger topic of data governance

Security and Privacy Considerations

- Models are intellectual property
- So controlled access is required
- How do we preserve privacy in model-training, scoring, and other data usage?
- **[papers and articles on privacy preservation](#)**

The “papers” link is to a Google Scholar search with several papers on privacy preserving techniques, like *differential privacy*.

Model Traceability - 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 Traceability Requirements

- A model repository
- Information stored for each model instance, possibly including:
 - Name
 - Version (or other unique ID)
 - Creation, deployment, and retirement dates
 - Model parameters
 - Quality metric
 - ...

Model Traceability in Use

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

...
-----	-----	-----	-----

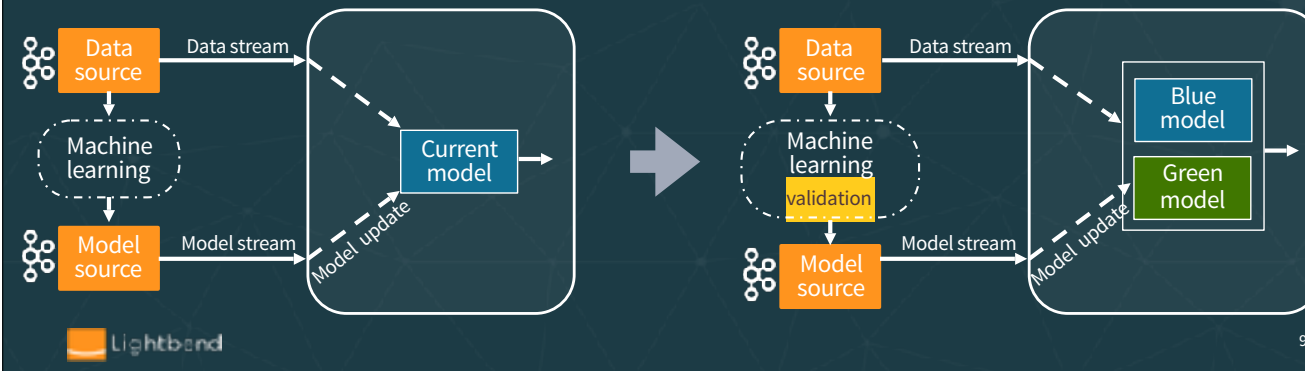
- Output Record with Score, model version ID

...	score	ID
-----	-----	-----	-----	-------	----

You might be tempted to avoid this extra data; don't you know the "deployment" date? This usually isn't accurate enough to know which records were on the boundary of switchover, due to timestamp granularity, clock skew, etc.

Software Process

- How and when should new models be deployed? (CI/CD)
- Are there a quality control steps first?
- Should you do **blue-green deployments**, perhaps using a **canary release** as a validation step?



92

Just the tip of the iceberg...

The blue-green deployments leads to our final topic...

Speculative Execution

According to Wikipedia, speculative execution is an **optimization** technique, where:

- The system performs work that may not be needed, before it's known if it will be needed.
- If and when it *is* needed, we don't have to wait.
- The results are discarded if they aren't needed.

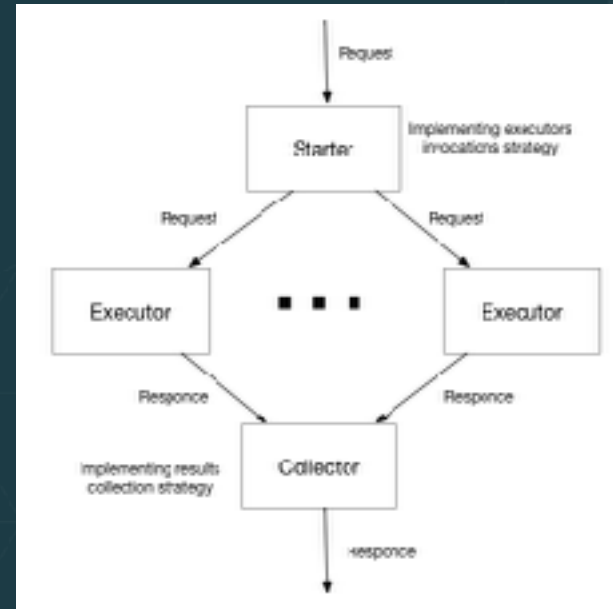
Speculative Execution

- Provides more concurrency if extra resources are available.
- Used for:
 - **branch prediction** in **pipelined processors**,
 - value prediction for exploiting value locality,
 - **prefetching instructions** and files,
 - etc.

Why not use it with machine learning??

General Architecture for Speculative Execution

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



General Architecture for Speculative Execution

- Starter (p
parallelis
strategy
- Parallel ex
- Collector
bringing r
together

Look familiar? It's similar to the pattern we saw previously for invoking a "farm" of actors or external services.

But we must add logic to pick the result to return.

Implementing executors
invocations strategy

Request

Executor

Response

response

Model Serving Use Case - Guarantee Execution Time

- I.e., meet a latency SLA
- Run several models:
 - A smart model, but takes time T_1 for a given record
 - A “less smart”, but faster model with a fixed upper-limit on execution time, with $T_2 \ll T_1$
- If timeout (latency budget) T occurs, where $T_2 < T < T_1$, return the less smart result
- But if $T_1 < T$, return that result
 - (Do you understand why $T_2 < T < T_1$ is 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 , and this is only useful if T_1 is often longer than T , our latency window.

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

Model Serving Use Case - Ensembles of Models

- Consensus-based model serving
 - N models (N odd)
 - Score with all of them and return the majority result
- Quality-based model serving
 - N models with the same quality metric
 - Pick the result with the best quality score for a given record
- Similarly for more sophisticated **boosting** and **bagging** systems

Consensus here is a majority vote system, a fairly crude *ensemble method*, while boosting and bagging are more sophisticated ensemble systems. Quality is roughly speaking an ensemble system, but here the models are treated independently, all with a quality metric, and the best score, based on that metric, is chosen.

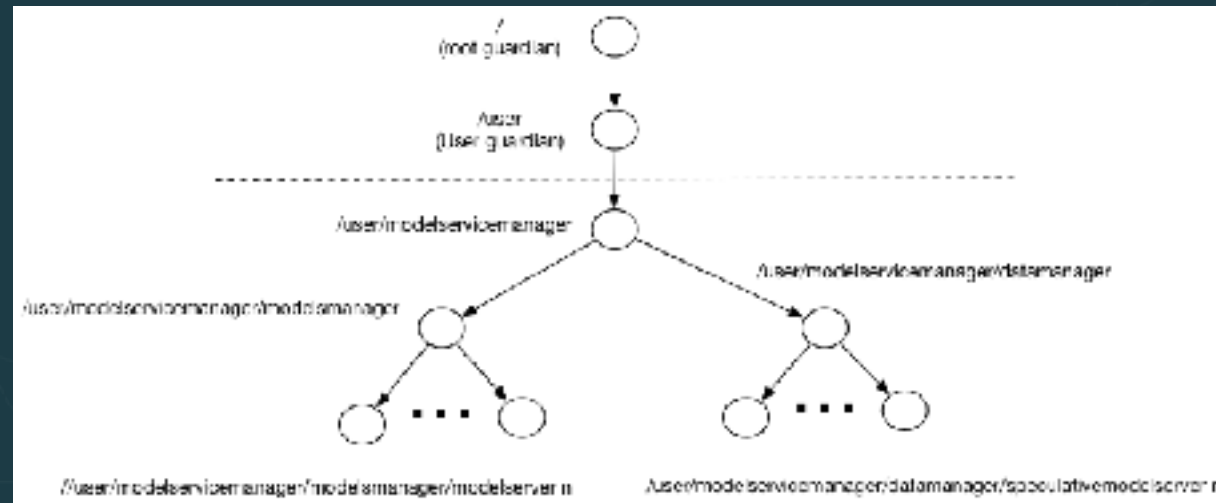
Architecture

The diagram illustrates the architecture of Speculative Model Serving. It features a central 'Speculative Model Serving Controller' box. An arrow labeled 'configuration' points down into this box. An arrow labeled 'data' points from the left into the box. An arrow labeled 'model serving result' points out from the right side of the box. Below the controller, there are two 'Model Server' boxes, with an ellipsis between them indicating multiple servers. Bidirectional arrows labeled 'individual model serving' connect the controller to each model server. To the bottom left is a 'Model Learning' box. An arrow labeled 'Individual model's update' points from the model learning box to the model servers. Additionally, a vertical arrow points from the 'data' input line down to the 'Model Learning' box.

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

This blog post provides more information on this technique.

One Design Using Actors



The path-like strings are the Akka way of defining a hierarchy of actors and provide an abstract way to reference an actor that doesn't require you to know the actual process and machine where it's running.

Outline

- Hidden technical debt in machine learning systems
- Model serving patterns
 - Embedding - models as code
 - Models as data
 - External services
 - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

Recap

- Model serving is one small(-ish) part of the whole ML pipeline
- Use *logs* (e.g., Kafka) to connect most services
- Models as data provides the most flexibility
- Model serving can be implemented in “general” microservices (e.g., Akka Streams) or data systems like Flink, Kafka
- Model serving can in process (embedded library) or external service (e.g., TensorFlow Serving)
- Production concerns include integration with your CI/CD pipeline, and data governance

Some of the main points we discussed.

Thanks for coming!

Questions?

lightbend.com/lightbend-platform
boris.lublinsky@lightbend.com
dean.wampler@lightbend.com

Don't miss:

- Sean Glover, *Put Kafka in Jail with Strimzi*
4:20pm–5:00pm Wednesday. Location: 2006
- Dean Wampler, *Executive Briefing: What it takes to use machine learning in fast data pipelines*
3:50pm–4:30pm Thursday. Location: 2020



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