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

Lightbend

©Copyright 2017-2019, Lightbend, Inc.
Apache 2.0 License. Please use as you see fit, but attribution is requested.

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.

Lightbend

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

Lightbend

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

But first, introductions…

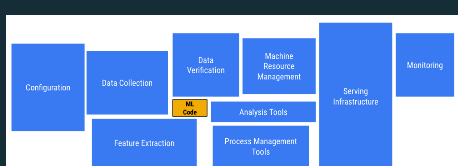
Lightbend

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

Lightbend

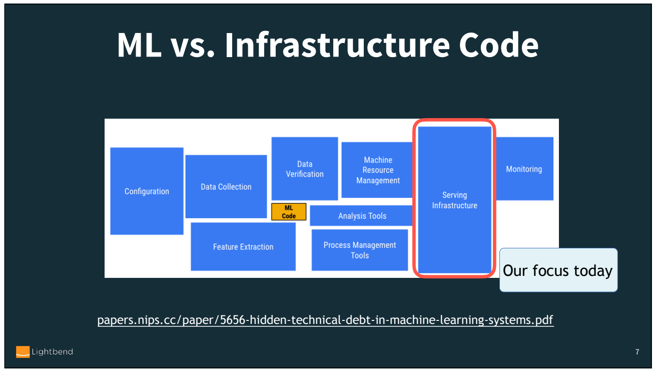
ML vs. Infrastructure Code



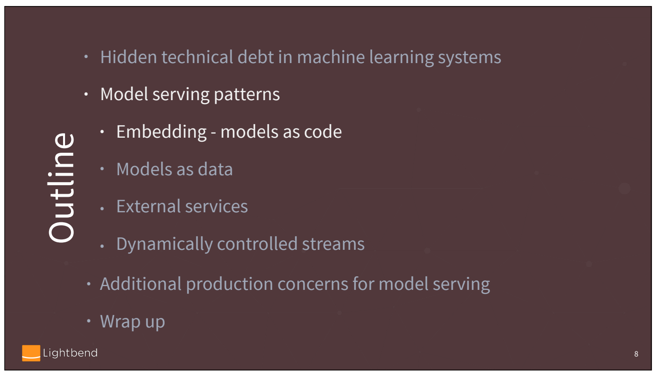
papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

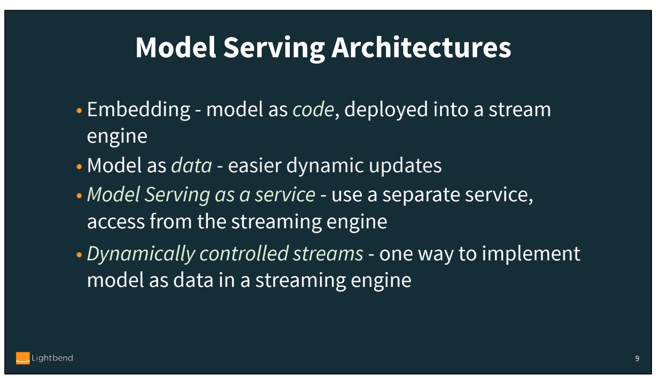
Lightbend

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.

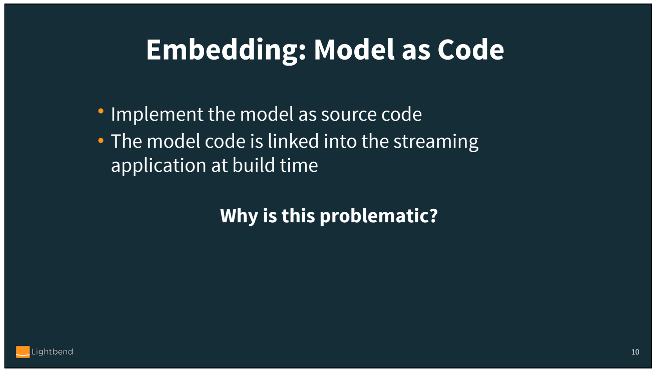


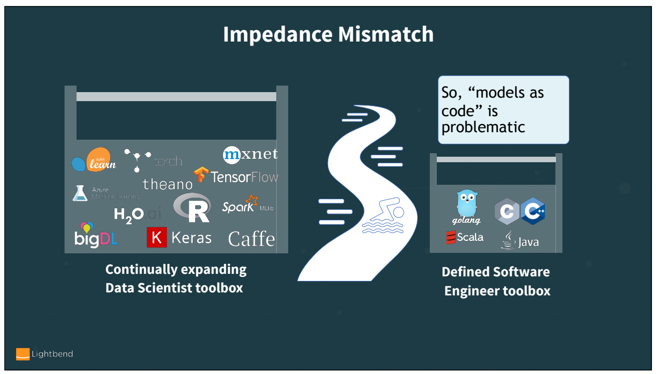
This tutorial discusses a few aspects of that picture, focusing on architectures for *serving* models in productions. The architectures use minimal coupling to tool chains for *training* (e.g., through a Kafka topic), which enables a wide range of *training* options, making it easier to use your favorite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



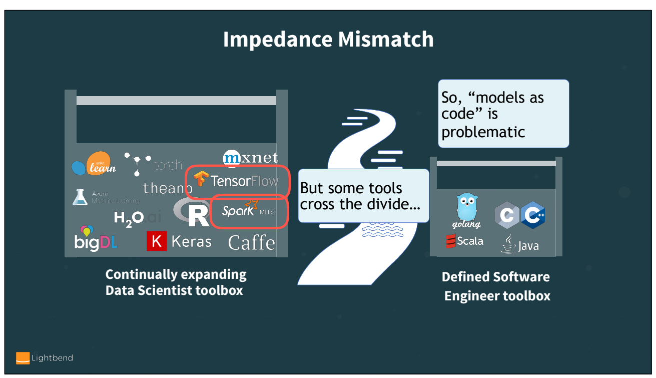


You can embed code for models, but we'll see this has many disadvantages. It's better to treat models as data, which allows you to update it as the "world" changes. We'll focus on two approaches for models as data.

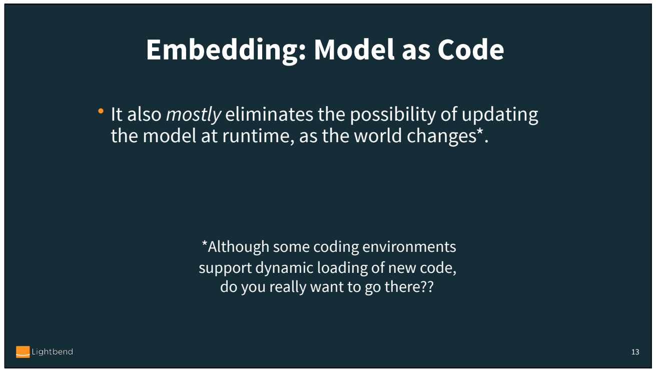




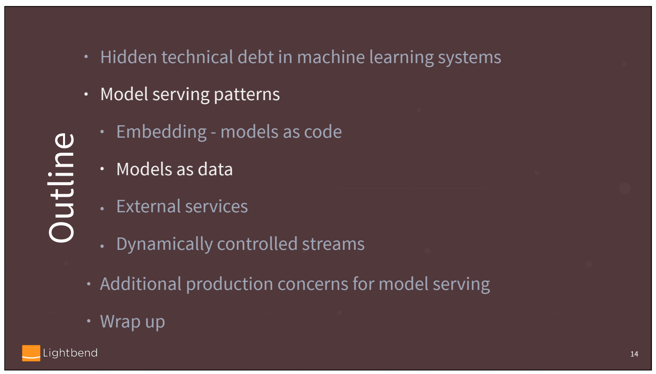
In his talk at the last Flink Forward, Ted Dunning discussed the fact that with multiple tools available to Data scientists, they tend to use different tools for solving different problems and as a result they are not very keen on tools standardization. This creates a problem for software engineers trying 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.

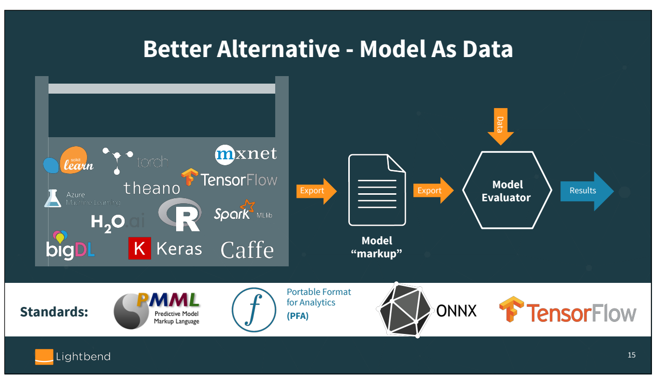


We loose portability if we use tools crossing divide



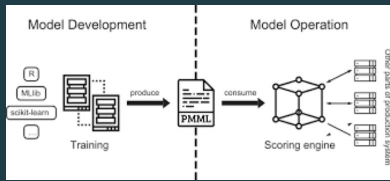
Most dynamically-typed languages, like Python, allow runtime loading of new code. Even the JVM supports this. However, it's much easier to introduce bugs, security holes (SQL injection attacks anyone??), etc. We'll also discuss other production concerns later, like auditing and data governance, which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





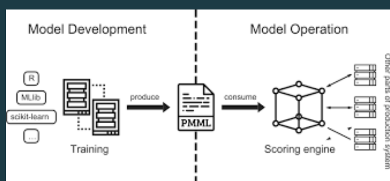
In order to overcome these differences, the Data Mining Group has introduced two standards - Predictive Model Markup Language (PMML) and Portable Format for Analytics (PFA), both suited for description of the models that need to be served. Introduction of these models led to creation of several software 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,

PMML



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

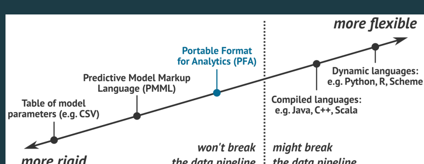
PMML



Implementations for:

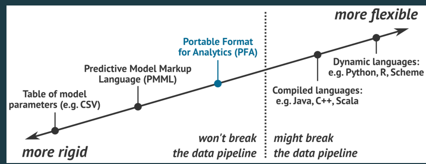
- Java (JPMML), R, Python Scikit-Learn, Spark here and here, …

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:

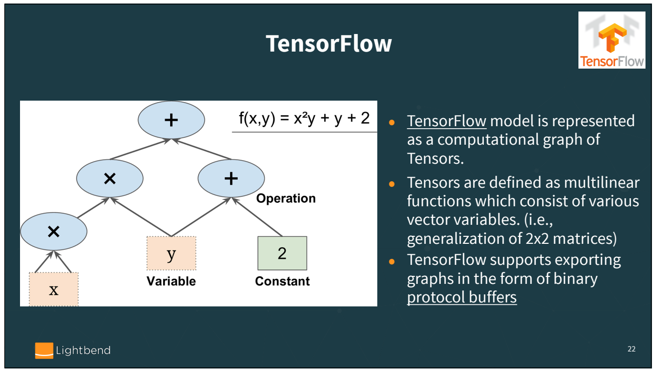
- Java (Hadrian), R (Aurelius), Python (Titus), Spark (Aardpfark), …

ONNX

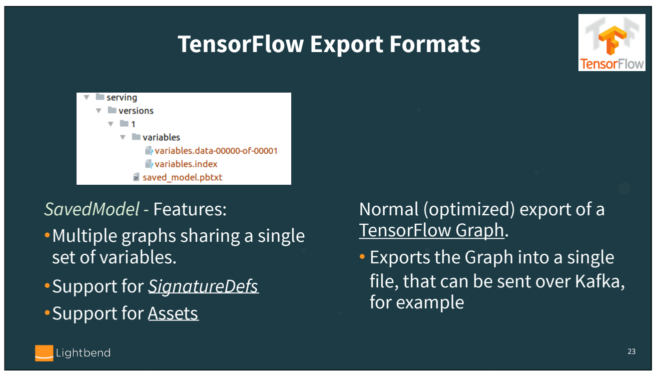
Open Neural Networks Exchange (ONNX) is an open standard format of machine learning models to offer interoperability between various AI frameworks. Led by Facebook, Microsoft, and AWS.

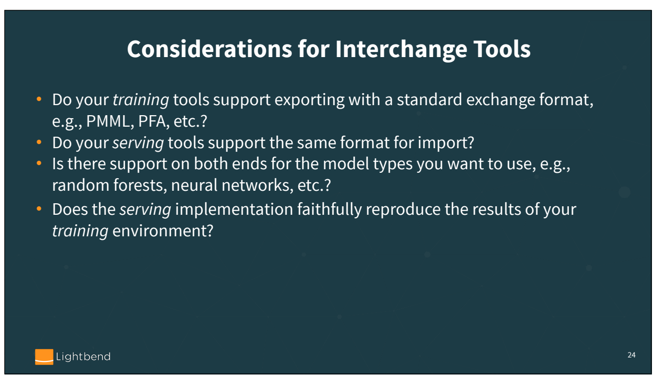
ONNX

- Supported Tools page.
- Converters for Keras, CoreML, LightGBM, Scikit-Learn, ….
- PyTorch,
- third-party support for TensorFlow

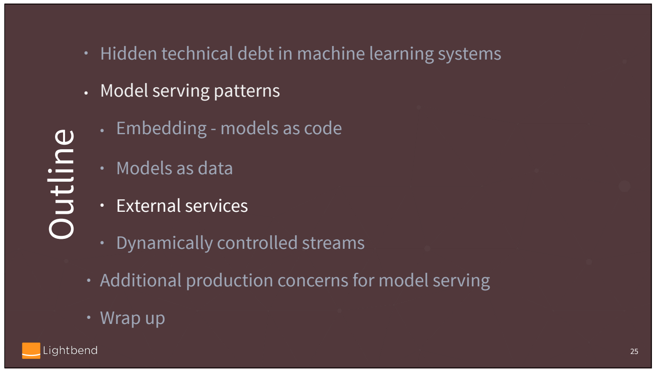


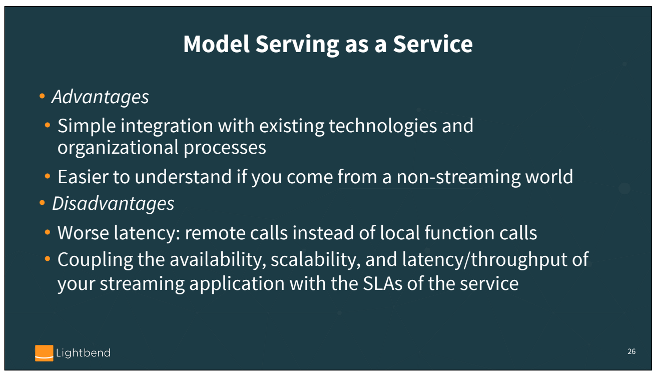
Mathematicians might beg to differ on what a Tensor actually is…



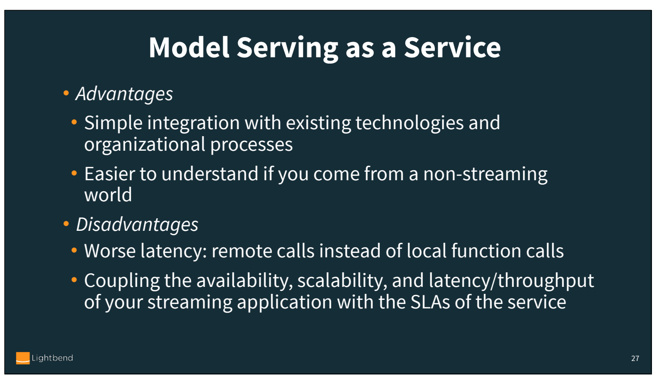


Implementations are spotty, both in terms of import, export coverage, model types, and faithfully reproducing the same results on both ends. I.e., if you use Scikit-Learn to train a random forest, do you get the same results if you run it in SparkML, after exchanging with PMML?





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?

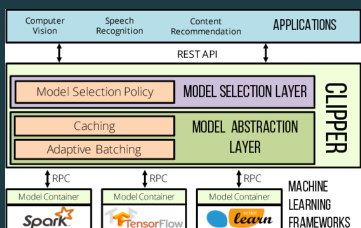


Model Serving as a Service challenges

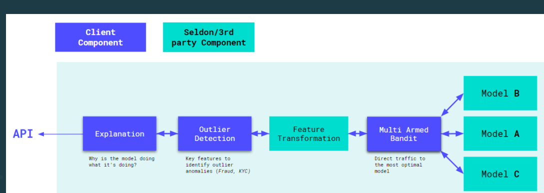
- Launch ML runtime graphs, scale up/down, perform rolling updates
- Infrastructure optimization for ML
- Latency optimization
- Connect to business apps via various APIs, e.g. REST, gRPC
- Allow Auditing and clear versioning
- Integrate into Continuous Integration (CI)
- Allow Continuous Deployment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

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

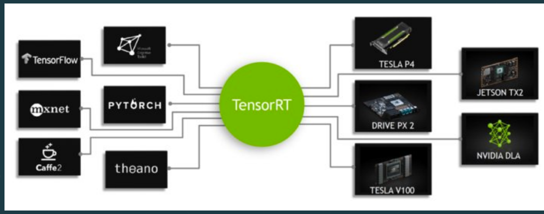
Example: Clipper



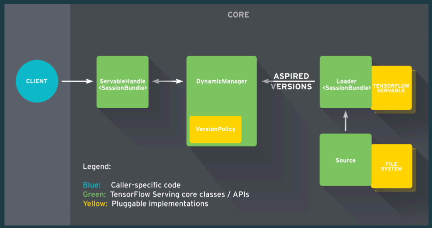
Example: Seldon Core



Example: TensorRT



Example: TensorFlow serving



Example: Kubeflow Serving (proposal)

TODO

https://docs.google.com/document/d/1_s8CYdhlrQRu4BX2m7adQhVt_OTr4WSZXgUY0Z77GzY/edit#heading=h.n8q237dhw3zp

Outline

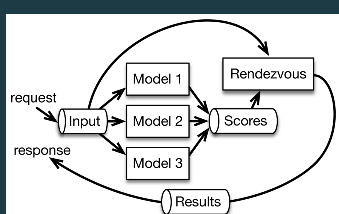
- Hidden technical debt in machine learning systems
- Model serving patterns
  - Embedding - models as code
  - Models as data
  - External services
  - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

Rendezvous Architecture

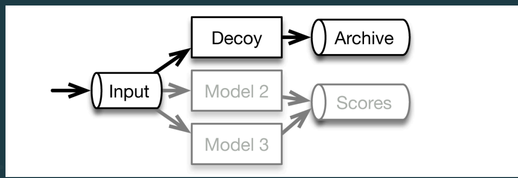
Designed to handle the logistics of ML in a flexible, responsive, convenient, and realistic way. Specifically, it provides the following:

- 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.

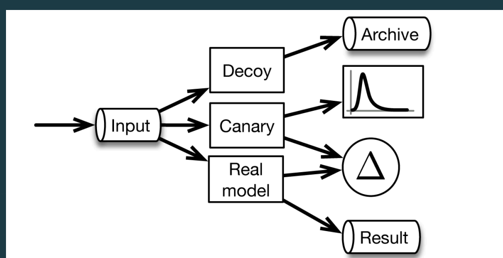
Rendezvous Architecture

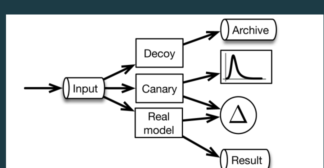


Rendezvous Architecture - Decoy

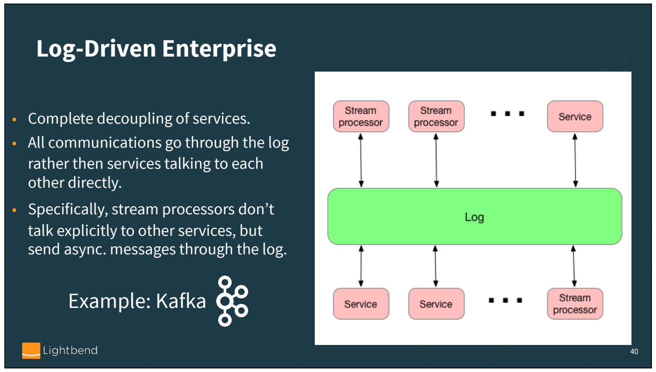


Rendezvous Architecture - Canary

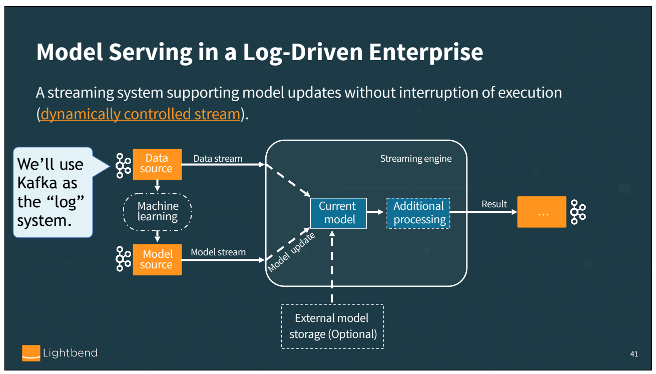




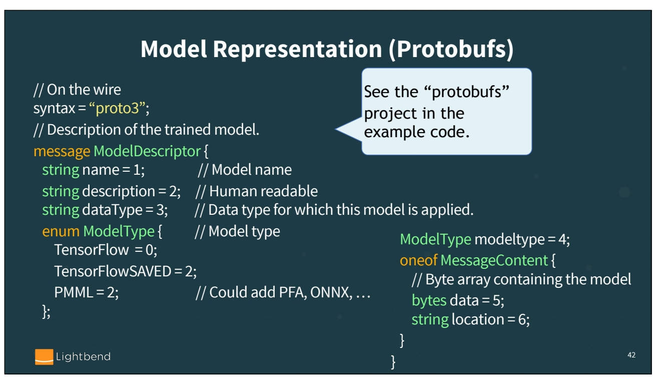
How are those “arrows” implemented?



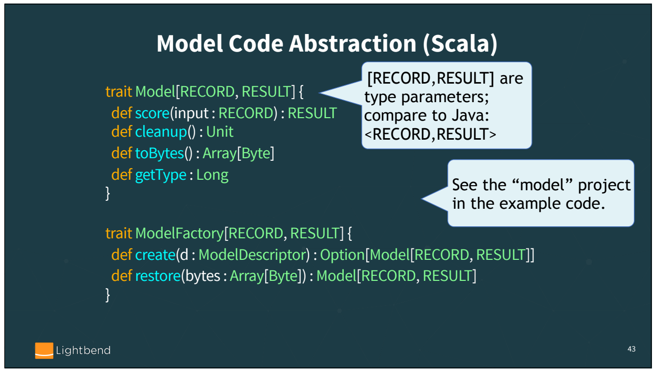
The majority of machine learning implementations are based on running model serving as a Rest service, which might not be appropriate for the high volume data processing or usage of the streaming system, which requires re coding/starting systems for model update, for example, Flink TensorFlow or Flink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



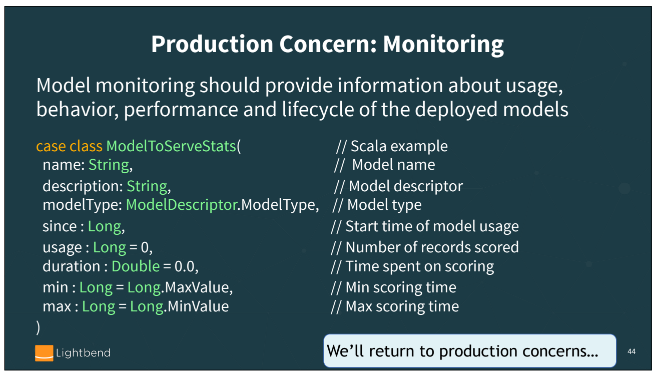
The majority of machine learning implementations are based on running model serving as a Rest service, which might not be appropriate for the high volume data processing or usage of the streaming system, which requires re coding/starting systems for model update, for example, Flink TensorFlow or Flink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

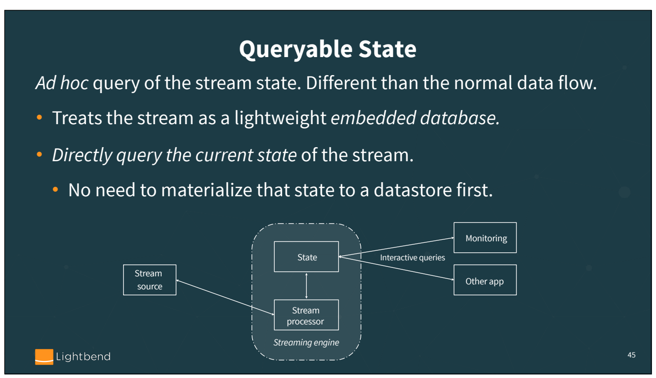


You need a neutral representation format that can be shared between different tools and over the wire. Protobufs (from Google) is one of the popular options. Recall that this is the format used for model export by TensorFlow. Here is an example.

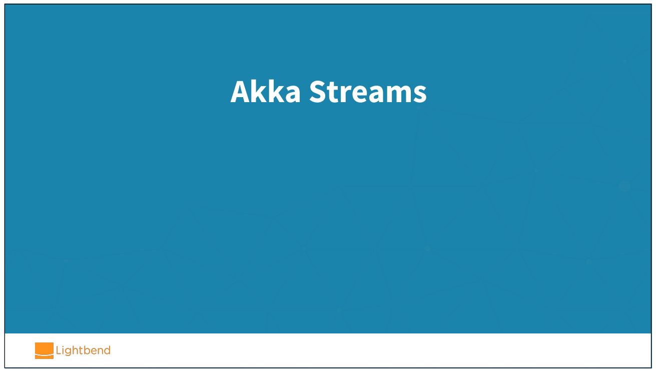


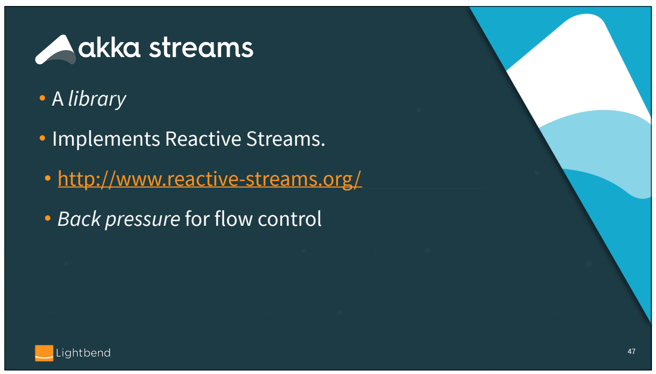
Corresponding Scala code that could be generated from the description, although we hand code this logic in the examples and exercises.



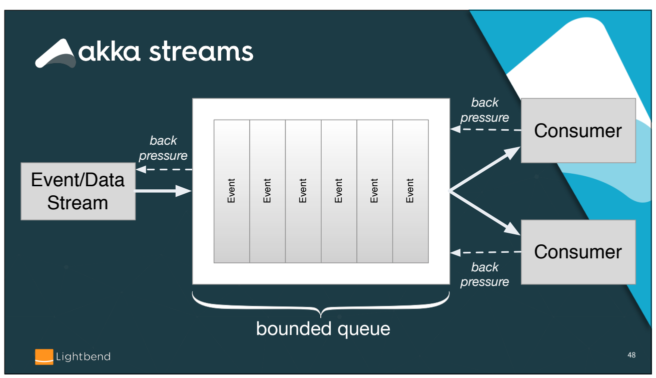


Note the “ad hoc” part. It’s for times when a “normal” stream of output data isn’t the best fit, e.g., only periodic updates are needed, you really want to support a range of “impromptu” (ad hoc) queries, etc.
Kafka Streams and Flink have built-in support for this and it’s being added to Spark Streaming. We’ll show how to use other Akka features to provide the same ability in a straightforward way for Akka Streams.

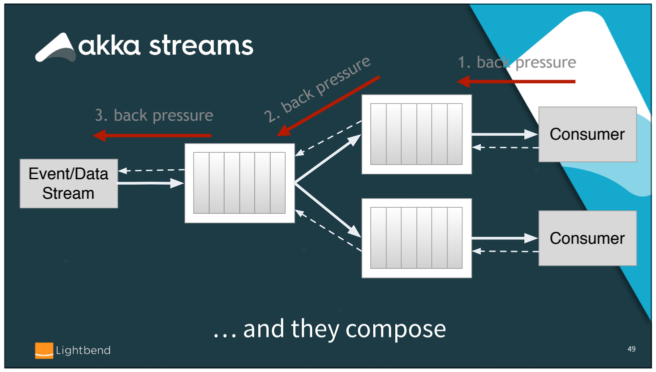




See this website for details on why *back pressure* is an important concept for reliable flow control, especially if you don't use something like Kafka as your "near-infinite" buffer between services.

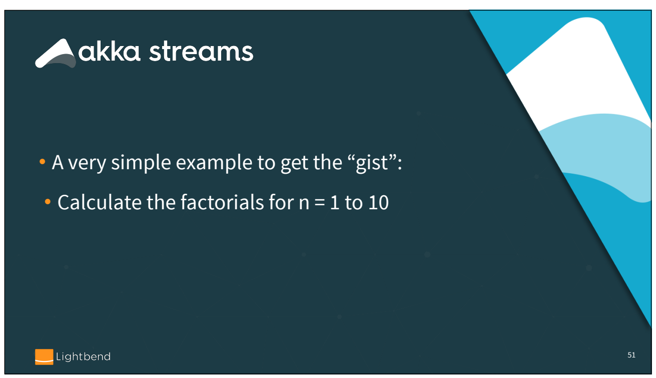


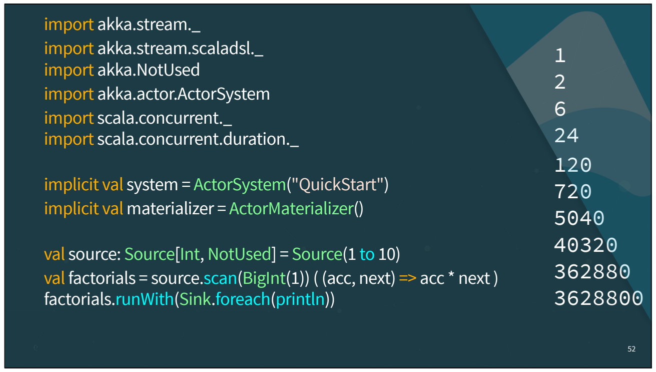
Bounded queues are the only sensible option (even Kafka topic partitions are bounded by disk sizes), but to prevent having to drop input when it's full, consumers signal to producers to limit flow. Most implementations use a push model when flow is fine and switch to a pull model when flow control is needed.



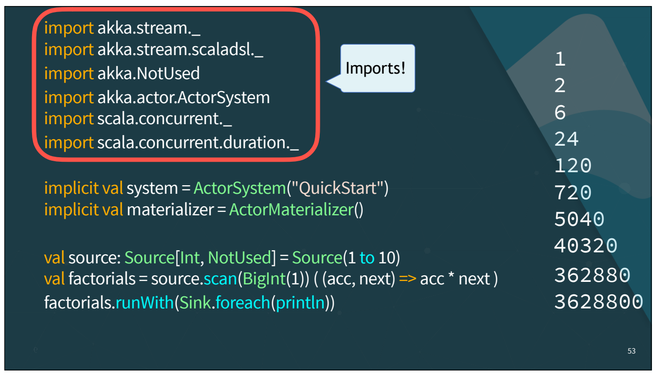
And they compose so you get end-to-end back pressure.

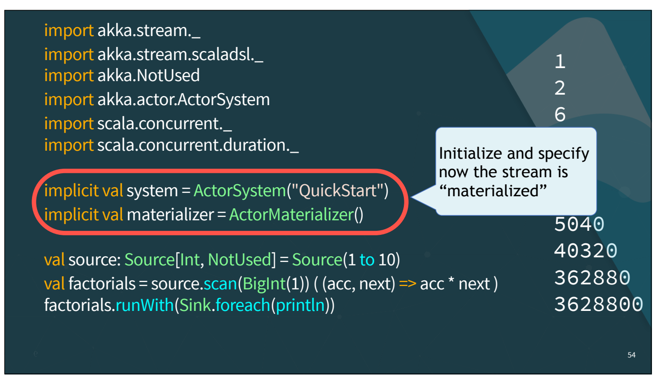
Rich, mature tools for the full spectrum of microservice development.

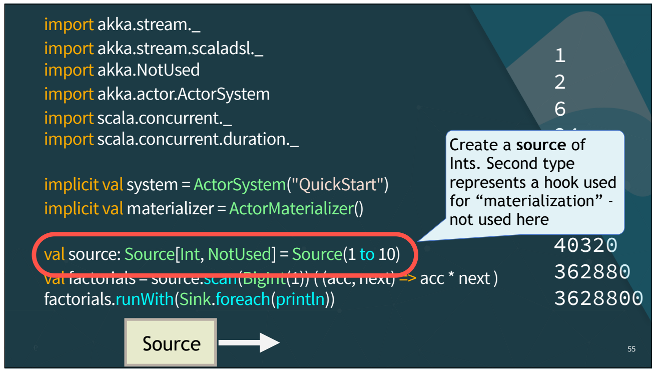




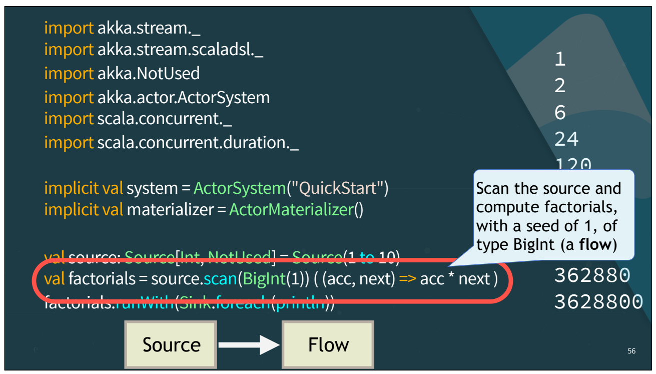
This example is in akkaStreamsModelServer/simple-akka-streams-example.sc



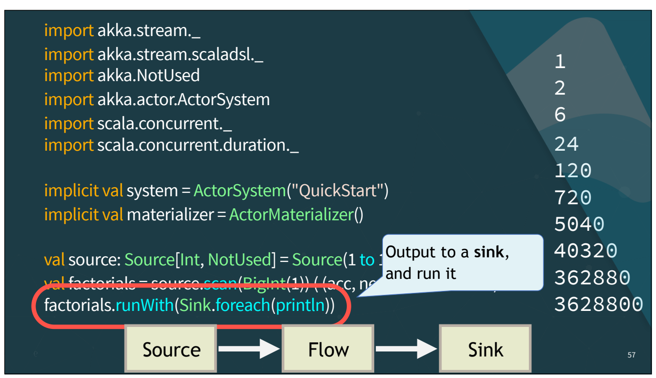




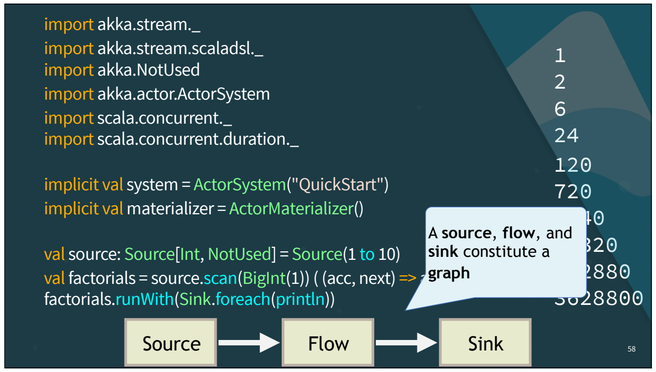
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._
implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()
val source: Source[Int, NotUsed] = Source(1 to 10)
val factorials = source.scan(BigInt(1)) ((acc, next) => acc * next)
factorials.runWith(Sink.foreach(println))
1
2
6
40320
362880
3628800
Create a **source** of Ints. Second type represents a hook used for "materialization" - not used here
Source
55



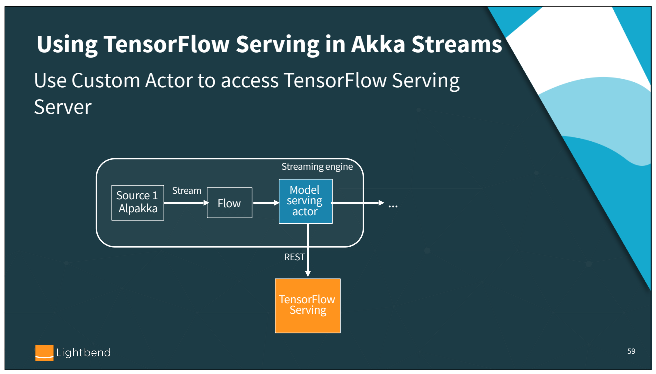
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._
implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()
val source: Source[Int, NotUsed] = Source(1 to 10)
val factorials = source.scan(BigInt(1)) ((acc, next) => acc * next)
factorials.runWith(Sink.foreach(println))
1
2
6
24
120
362880
3628800
Scan the source and compute factorials, with a seed of 1, of type BigInt (a **flow**)
Source
Flow
56



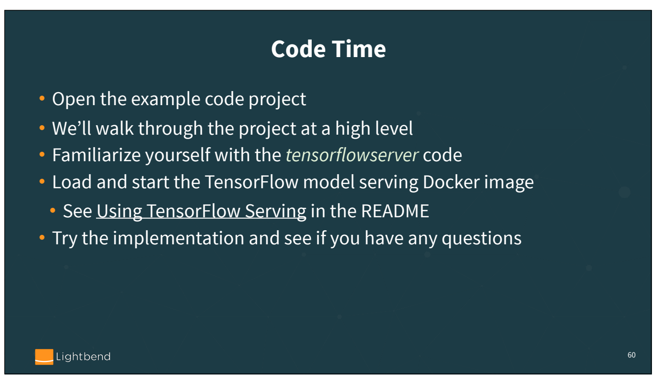
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._
implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()
val source: Source[Int, NotUsed] = Source(1 to
val factorials = source.scan(BigInt(1)) ((acc, n
factorials.runWith(Sink.foreach(println))
1
2
6
24
120
720
5040
40320
362880
3628800
Output to a **sink**, and run it
Source
Flow
Sink
57

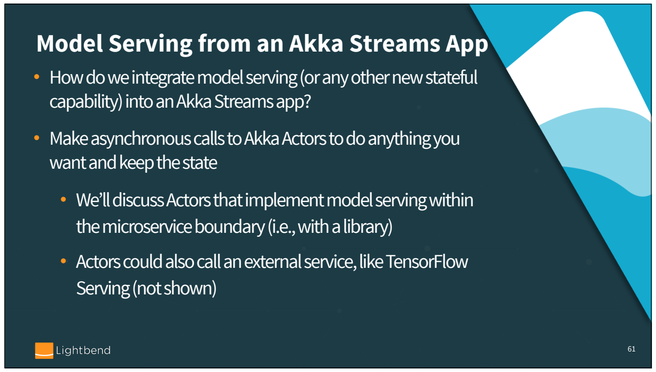


The core concepts are sources and sinks, connected by flows. There is the notion of a Graph for more complex dataflows, but we won't discuss them further

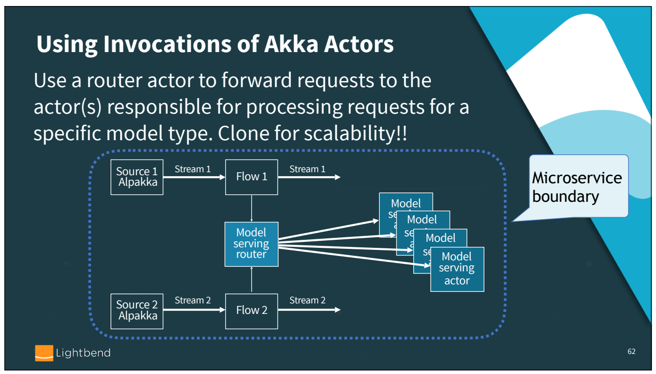


Use the same routing layer idiom: an actor that will implement model serving for a specific model (based on some key) and route messages appropriately to the external service. This way our system will serve models in parallel.

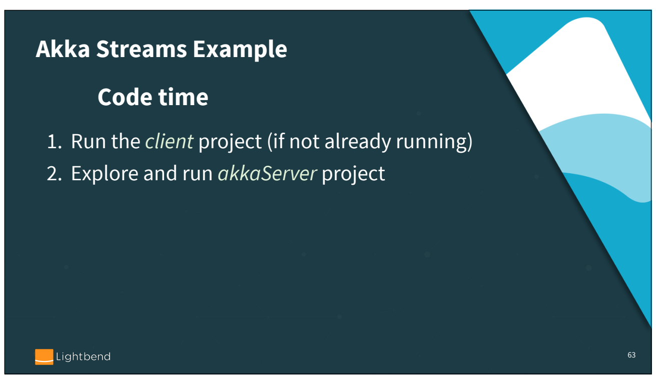




We provide two implementations. You could generalize the second approach (async calls) to invoke an external service. We won't provide examples of this option, but return later with some additional considerations about it.



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.



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

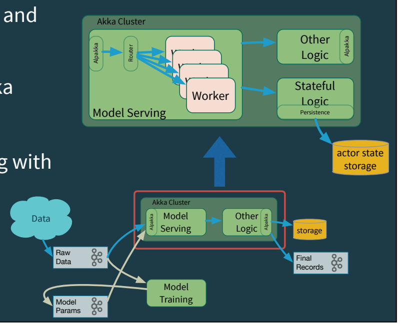
Curl or open in a browser:

http://localhost:5500/models
http://localhost:5500/state/wine

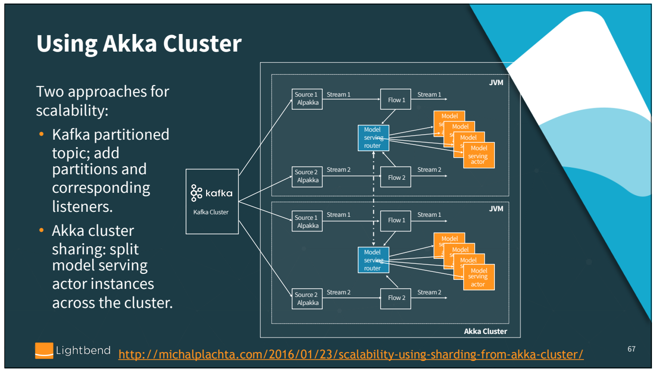
Handling Other Production Concerns with Akka and Akka Streams

---

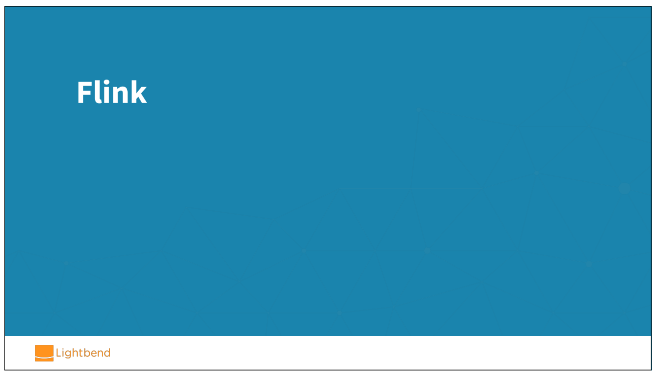
- Scale scoring with workers and routers, across a cluster
- Persist actor state with Akka Persistence
- Connect to *almost* anything with Alpakka



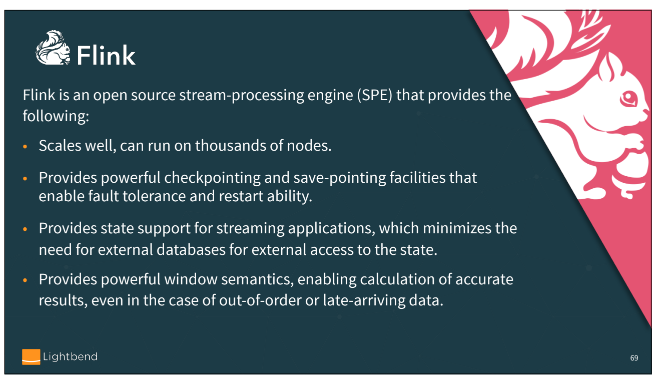
Here's our streaming microservice example adapted for Akka Streams. We'll still use Kafka topics in some places and assume we're using the same implementation for the "Model Training" microservice. Alpakka provides the interface to Kafka, DBs, file systems, etc. We're showing two microservices as before,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

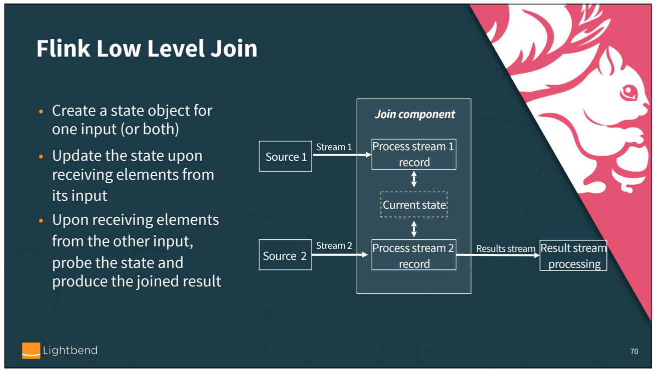


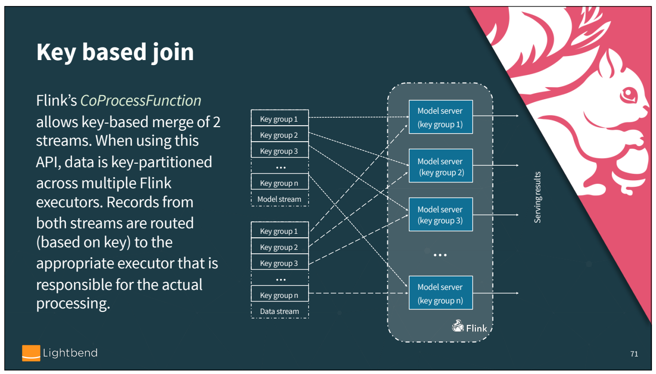
A great article http://michalplachta.com/2016/01/23/scalability-using-sharding-from-akka-cluster/ goes into a lot of details on both implementation and testing



Same sample use case, now with Kafka Streams



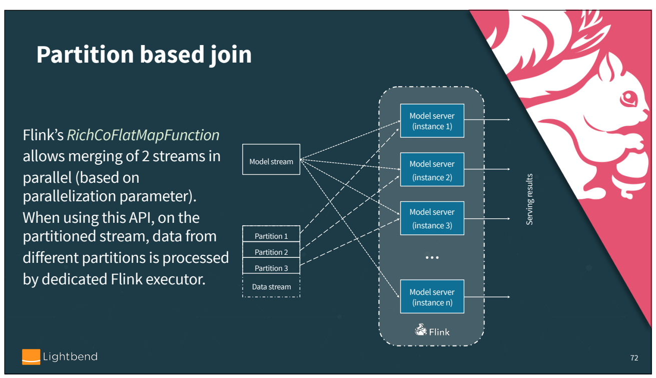




The main characteristics of this approach:

- 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.



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

Spark Structured Streaming

Same sample use case, now with Kafka Streams

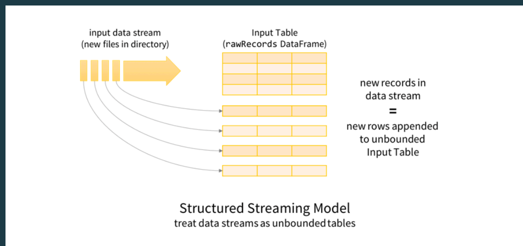
Spark Structured Streaming

Spark Streaming

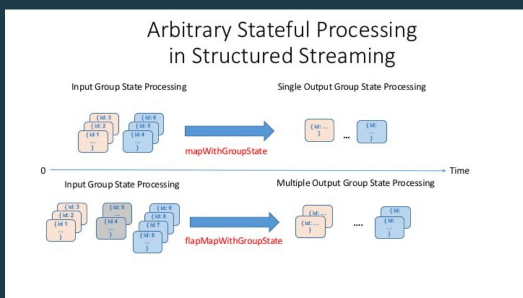
Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.

- 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.

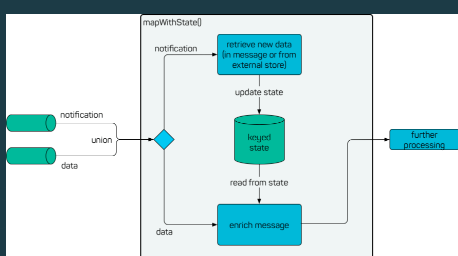
Spark Structured Streaming



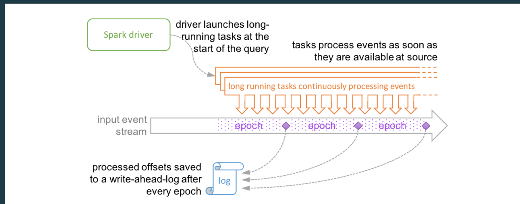
credit for diagrams: spark.apache.org/docs/latest/structured-streaming-programming-guide.html



Spark Structured Streaming - mapWithState

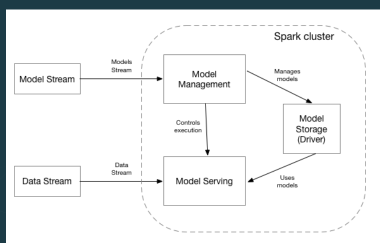


Spark Structured Streaming - continuous processing



Continuous Processing uses long-running tasks to continuously process events

Multi loop continuous processing



Spark Example

Code time

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

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**

Lightbend

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**

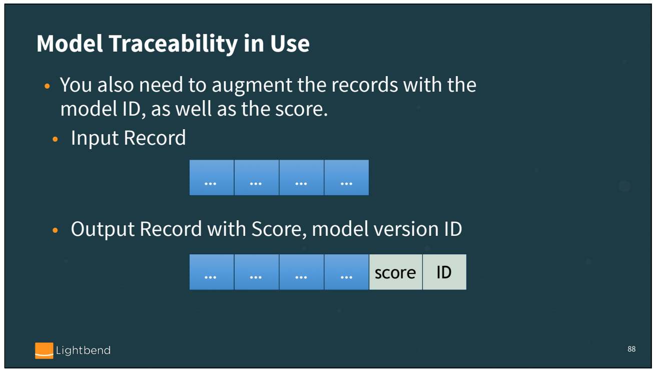
Lightbend

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

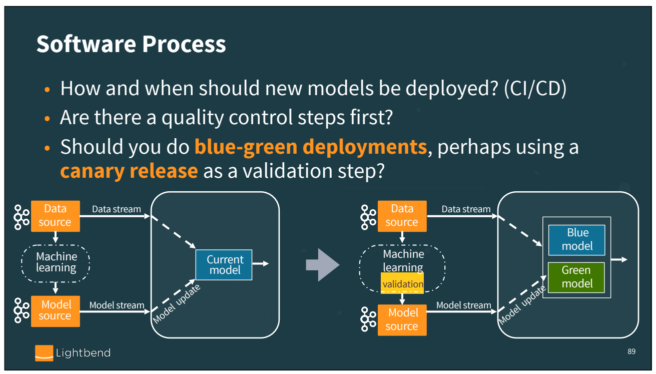
Model Traceability Requirements

- 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
  - …

Lightbend

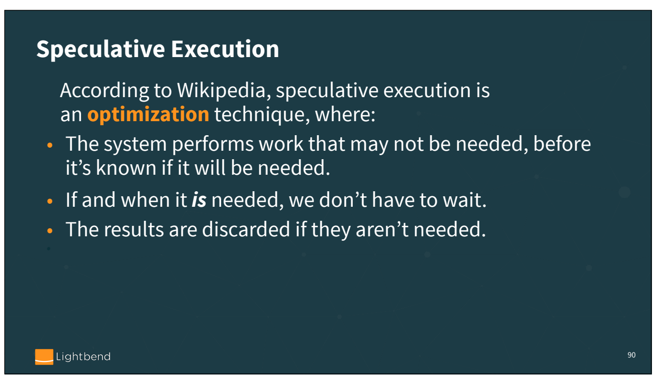


You might be tempted to avoid this extra data; don’t you know the “deployment” date? This usually isn’t accurate enough to know which records were on the boundary of switchover, due to timestamp granularity, clock skew, etc.



Just the tip of the iceberg…

The blue-green deployments leads to our final topic…



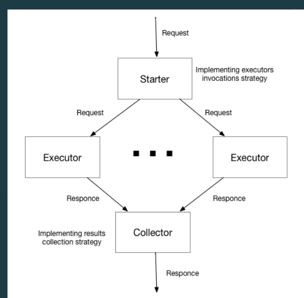
Speculative Execution

- 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

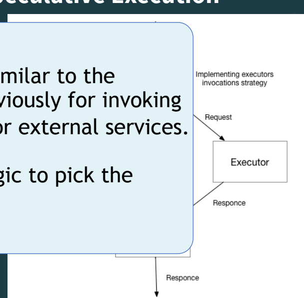


General Architecture for Speculative Execution

- Starter (p
parallelis
strategy
- Parallel e
- 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.



Model Serving Use Case - Guarantee Execution Time

- I.e., meet a latency SLA
- Run several models:
  - A smart model, but takes time *T1* for a given record
  - A “less smart”, but faster model with a fixed upper-limit on execution time, with *T2 << T1*
- If timeout (latency budget) *T* occurs, where *T2 < T < T1*, return the less smart result
- But if *T1 < T*, return that result
  - (Do you understand why *T2 < T < T1* is required?)

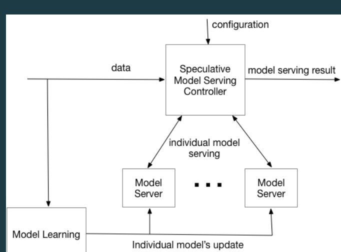
Because the timeout T has to be long enough that the fast model has time to finish, so T must be longer than T2, and this is only useful if T1 is often longer than T, our latency window.
This technique is “speculative”, because we try both models, taking a compromise result if the good result takes too long to return.

Model Serving Use Case - Ensembles of Models

- Consensus-based model serving
  - *N* models (*N > 2*)
  - 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.

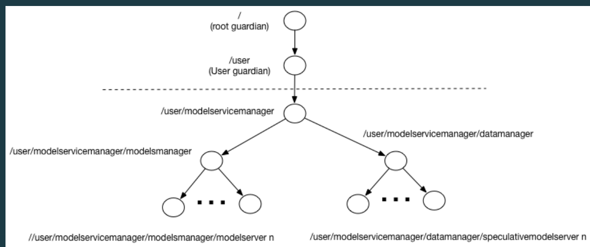
Architecture



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.

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.