Hands-on Machine Learning with Kafka-based Streaming Pipelines

Strata Data, San Jose, 2019

Boris Lublinsky and Chaoran Yu, Lightbend

boris.lublinsky@lightbend.com
chaoran.yu@lightbend.com

Lightbend

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

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

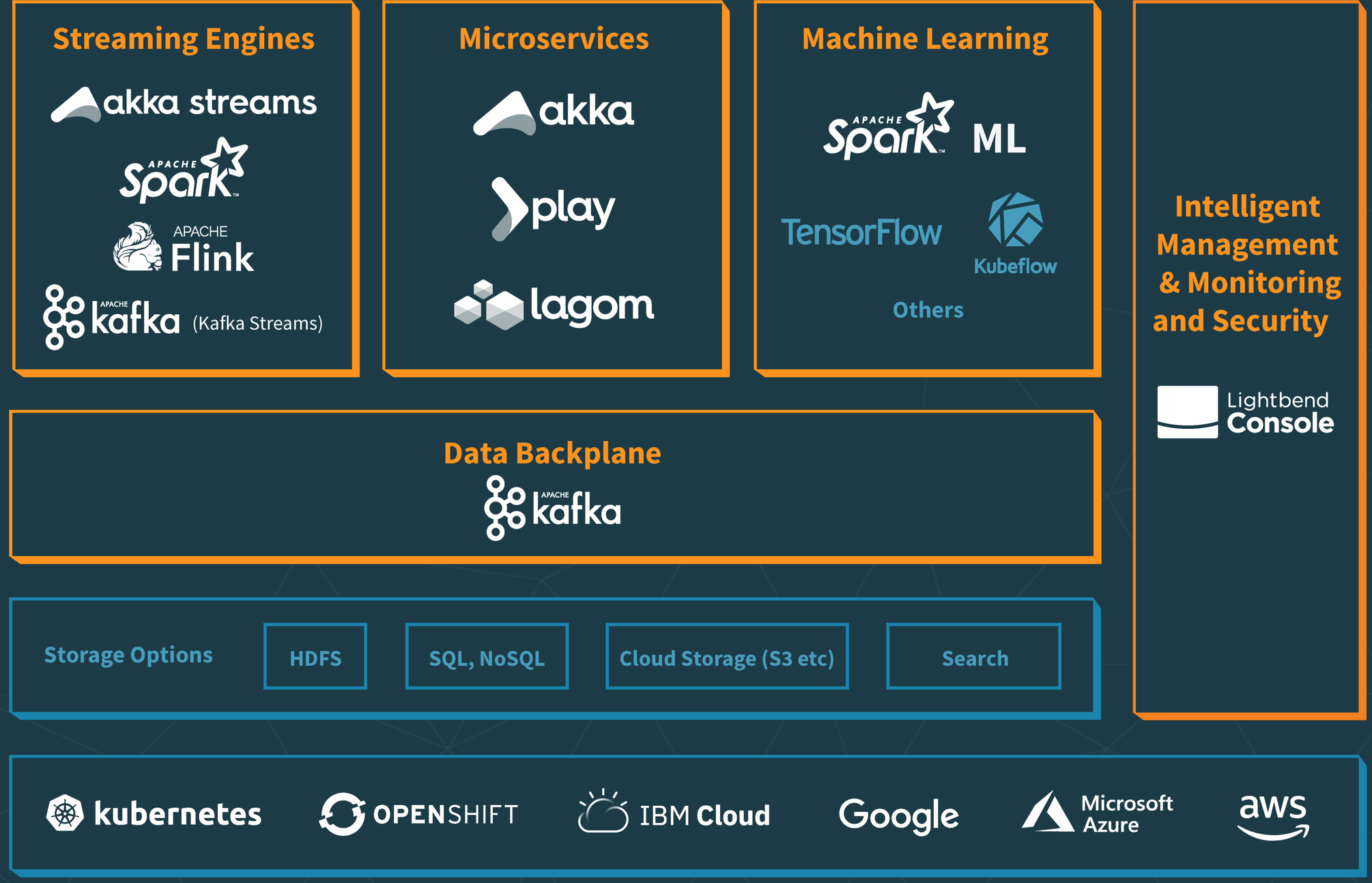
See the README for setup instructions.

These slides are in the presentation folder.

Outline

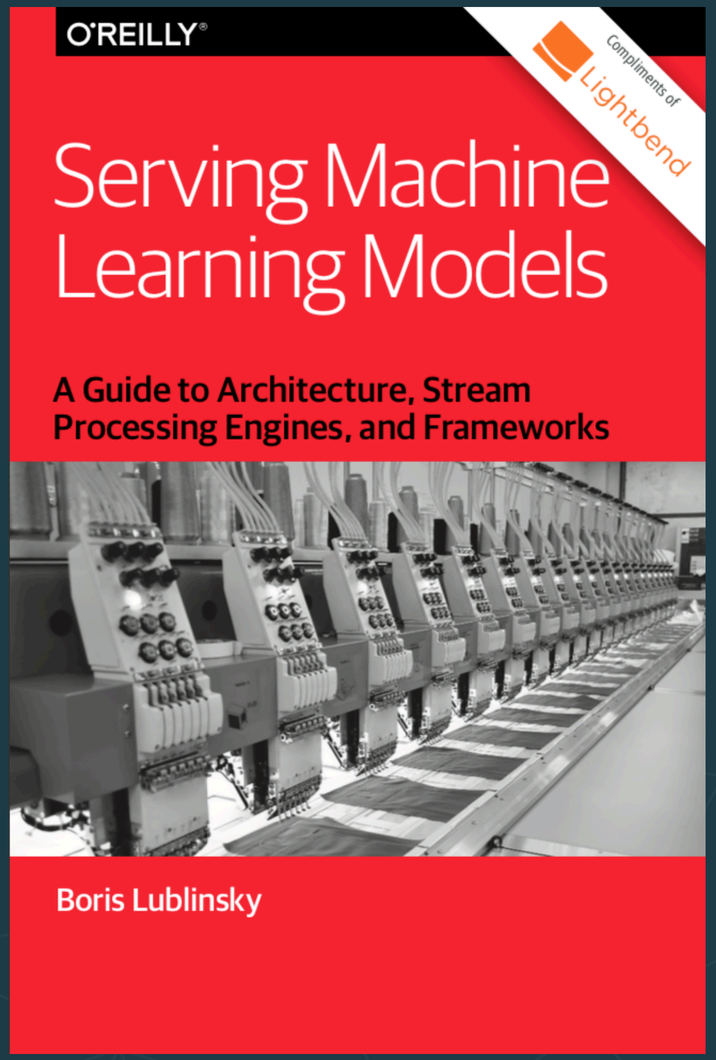
- Hidden technical debt in machine learning systems
- Model serving patterns
  - Embedding models as code
  - Models as data
  - Model serving as a service
  - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

But first, introductions…

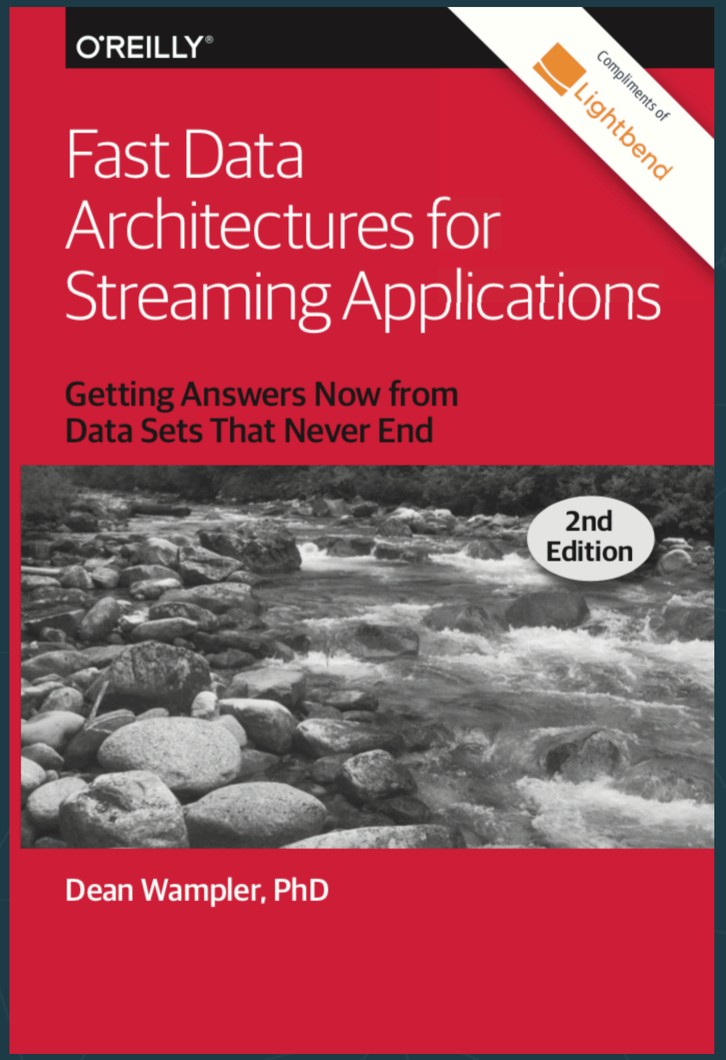

Streaming Engines
akka streams
Spark
Flink
kafka (Kafka Streams)
Microservices
akka
play
lagom
Machine Learning
Spark ML
TensorFlow
Kubeflow
Others
Intelligent Management & Monitoring and Security
Lightbend Console
Data Backplane
kafka
Storage Options
HDFS
SQL, NoSQL
Cloud Storage (S3 etc)
Search
kubernetes
OPENSHIFT
IBM Cloud
Google
Microsoft Azure
aws

Lightbend Pipelines

lightbend.com/lightbend-pipelines-demo



Free Download



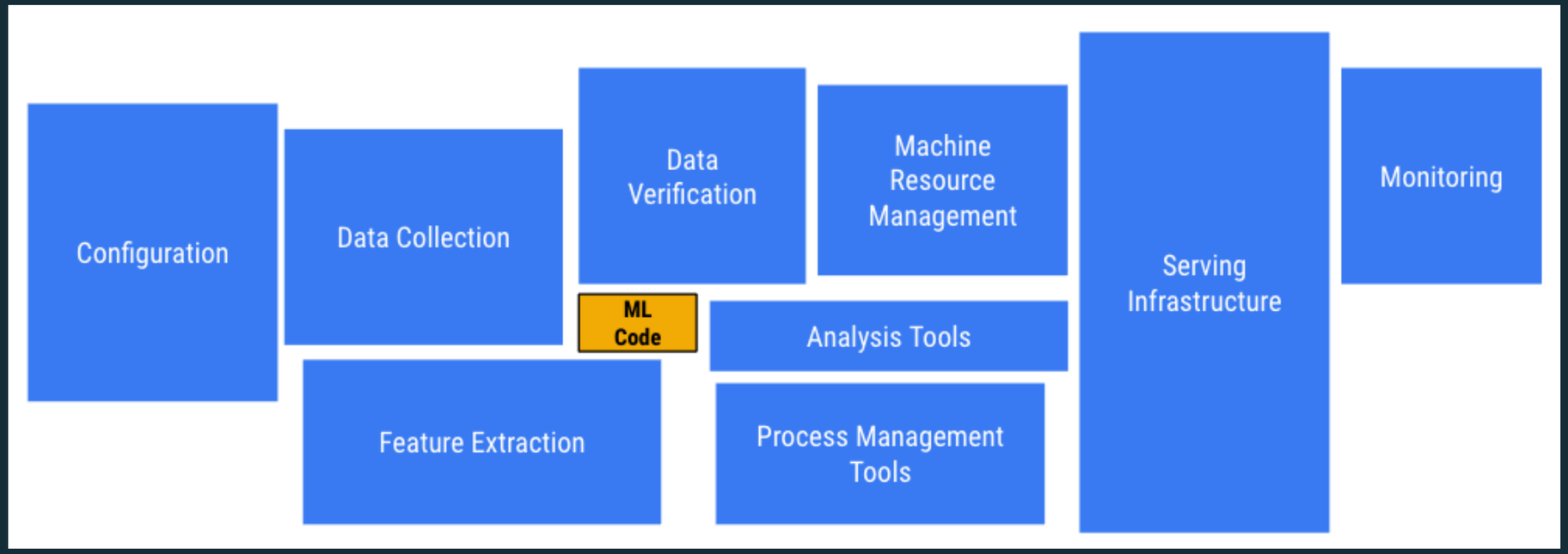
Free Download

Lightbend

Outline

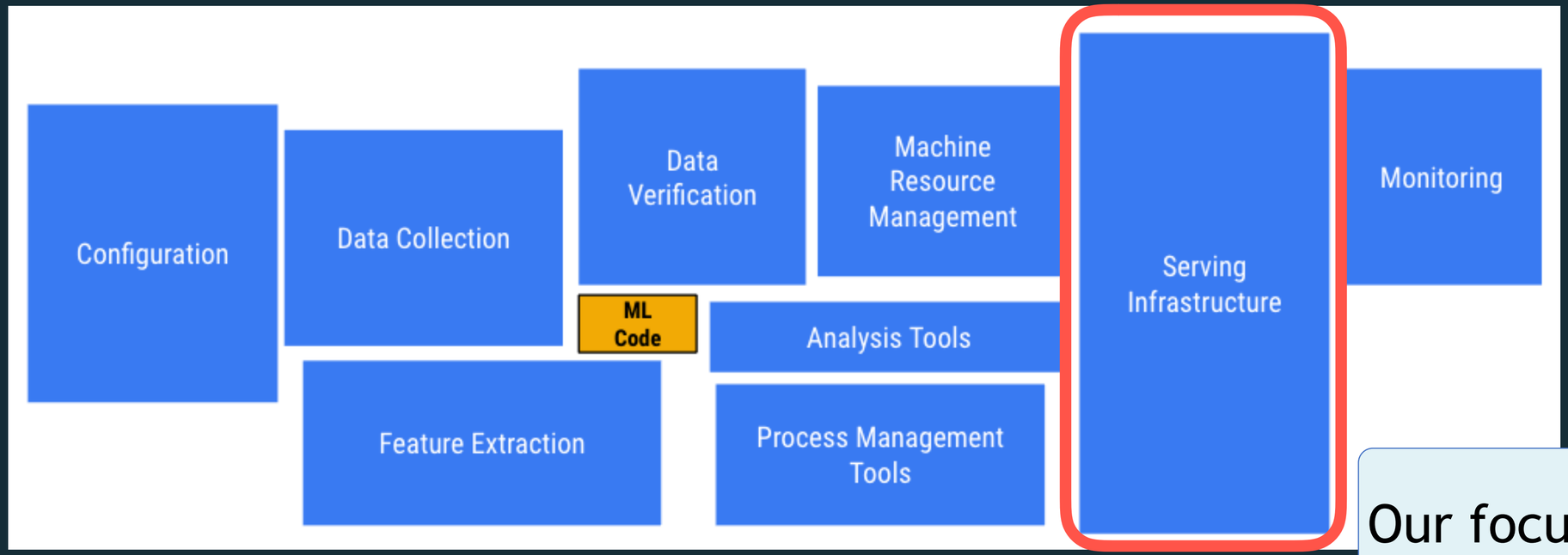
- Hidden technical debt in machine learning systems
- Model serving patterns
  - Embedding models as code
  - Models as data
  - Model serving as a service
  - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

ML vs. Infrastructure



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

ML vs. Infrastructure



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

Outline

- Hidden technical debt in machine learning systems
- Model serving patterns
  - Embedding models as code
  - Models as data
  - Model serving as a service
  - 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

Embedding: Model as Code

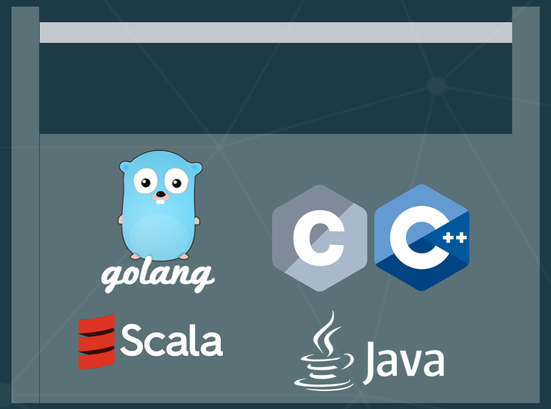
- Implement the model as source code
- The model code is linked into the streaming application at build time

Why is this problematic?

Impedance Mismatch

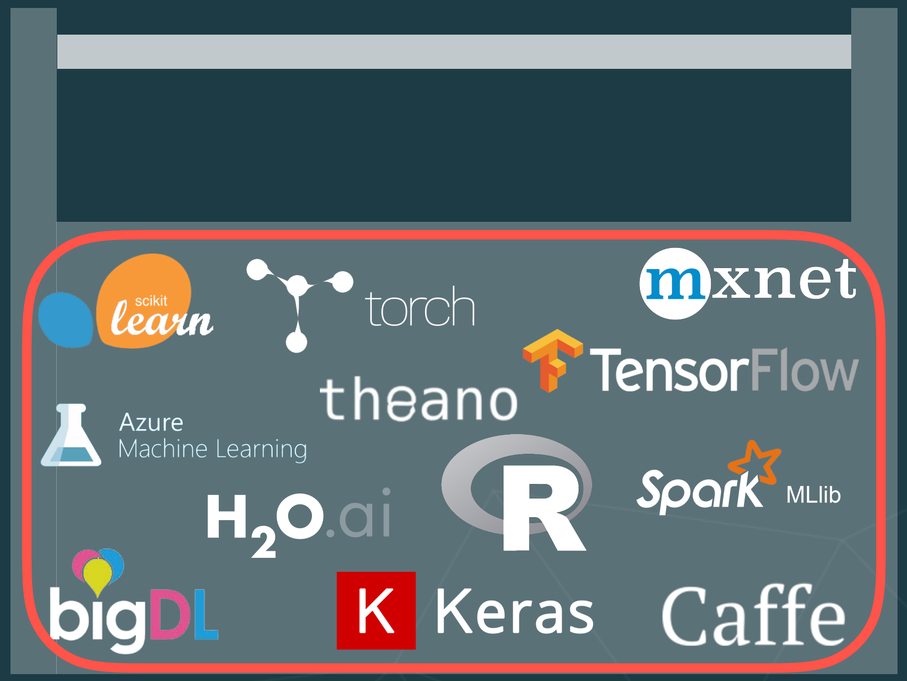
Continually expanding Data Scientist toolbox

So, “models as code” is problematic



Defined Software Engineer toolbox

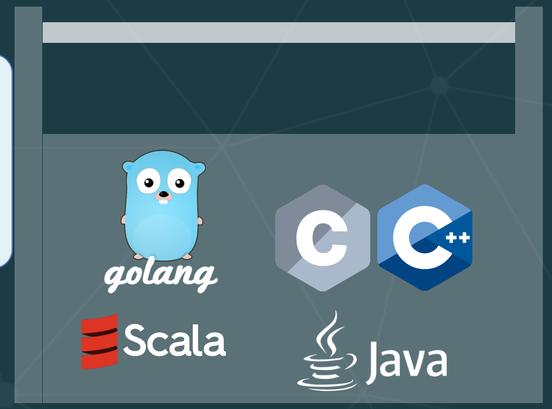
Impedance Mismatch



Continually expanding Data Scientist toolbox

But some tools cross the divide…

So, “models as code” is problematic



Defined Software Engineer toolbox

Embedding: Model as Code

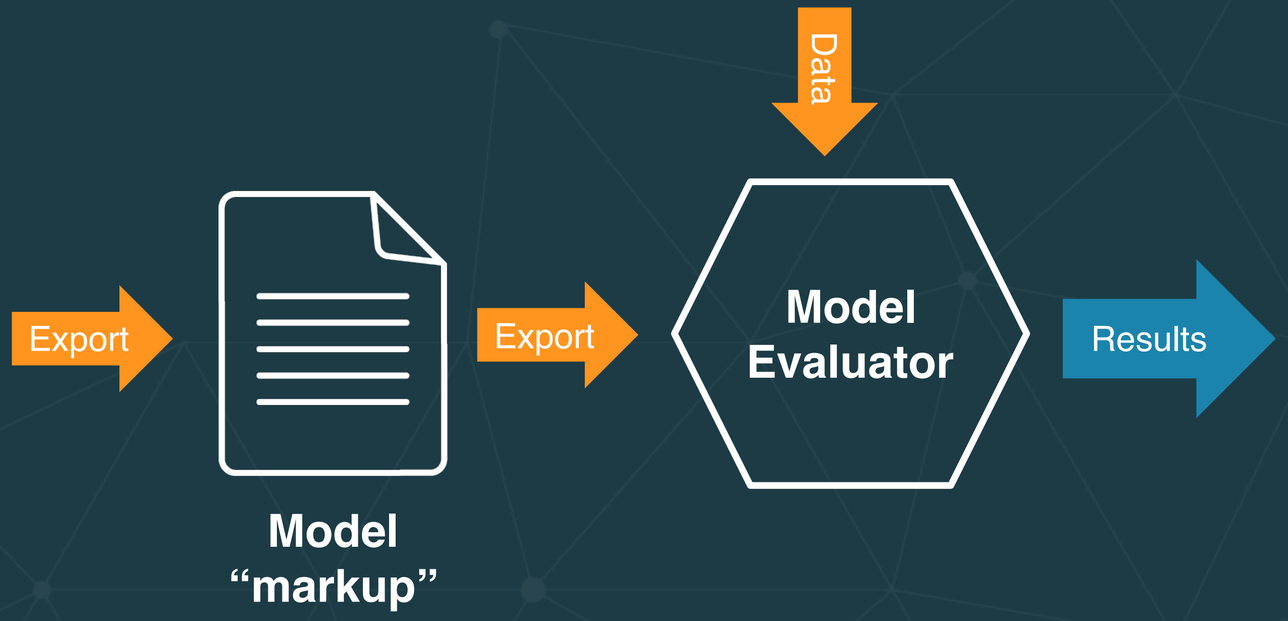
- It also *mostly* eliminates the ability to update the model at runtime, as the world changes*.

*Although some environments support dynamic loading of new code (e.g., JVM), you really don't want to go there…

Outline

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

Better Alternative - Model As Data



Standards:

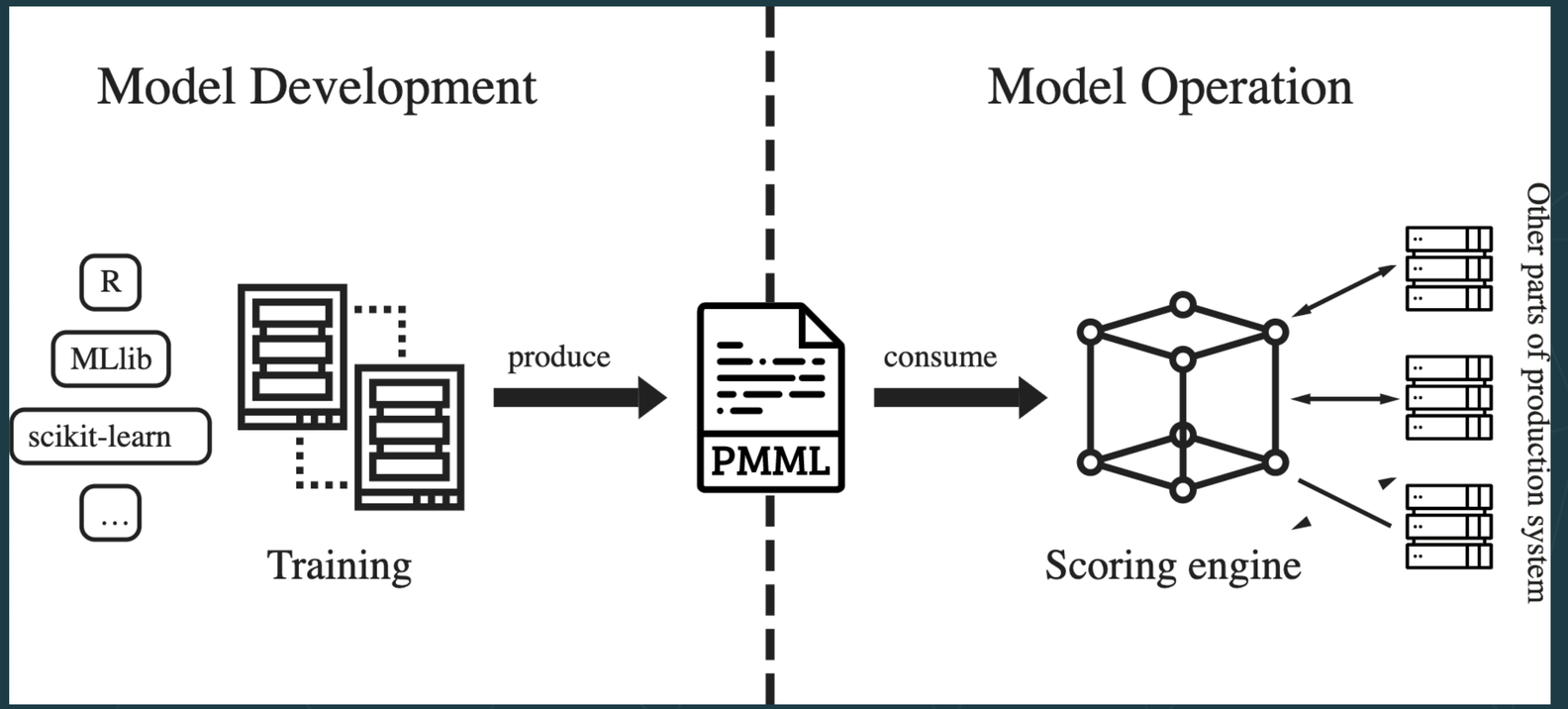
PMML Predictive Model Markup Language

Portable Format for Analytics **(PFA)**

ONNX

TensorFlow

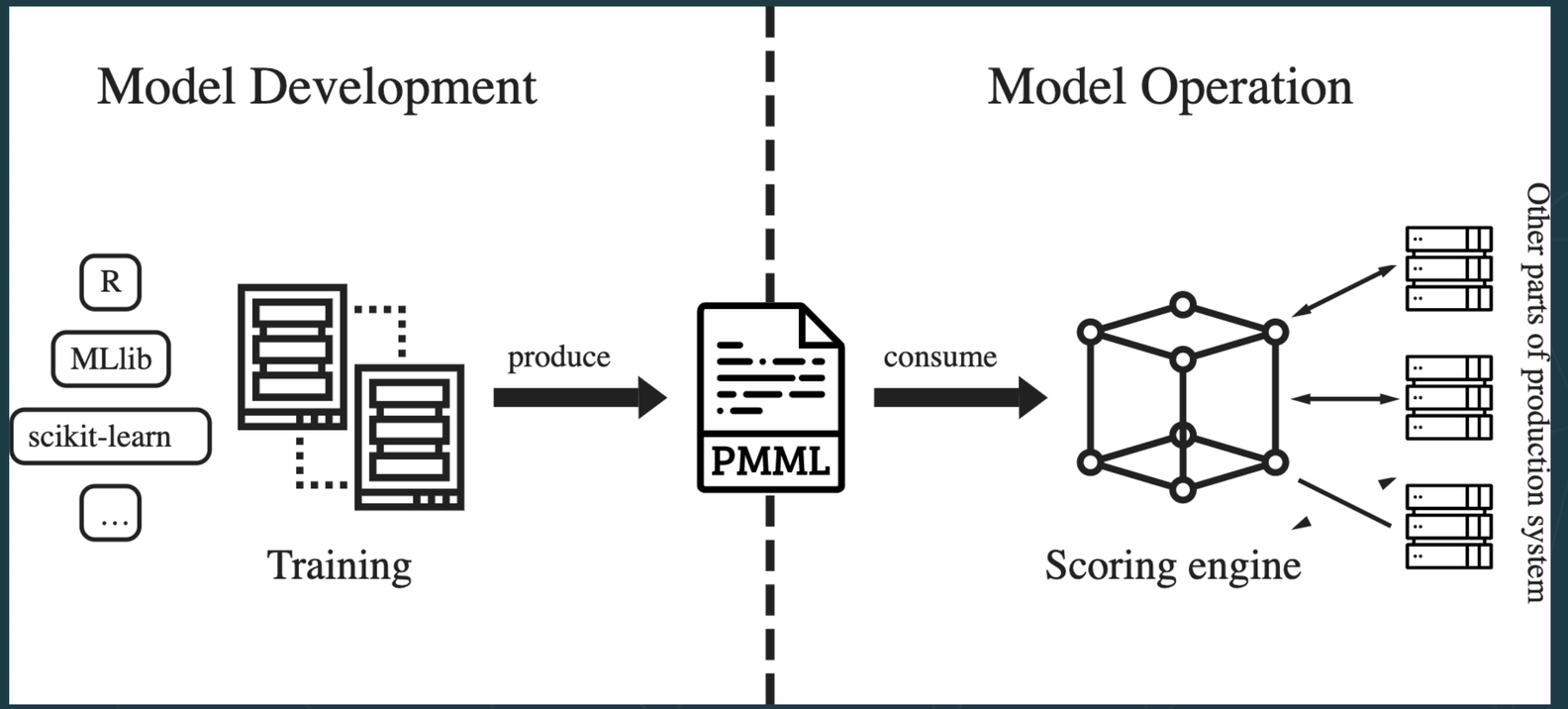
PMML



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/

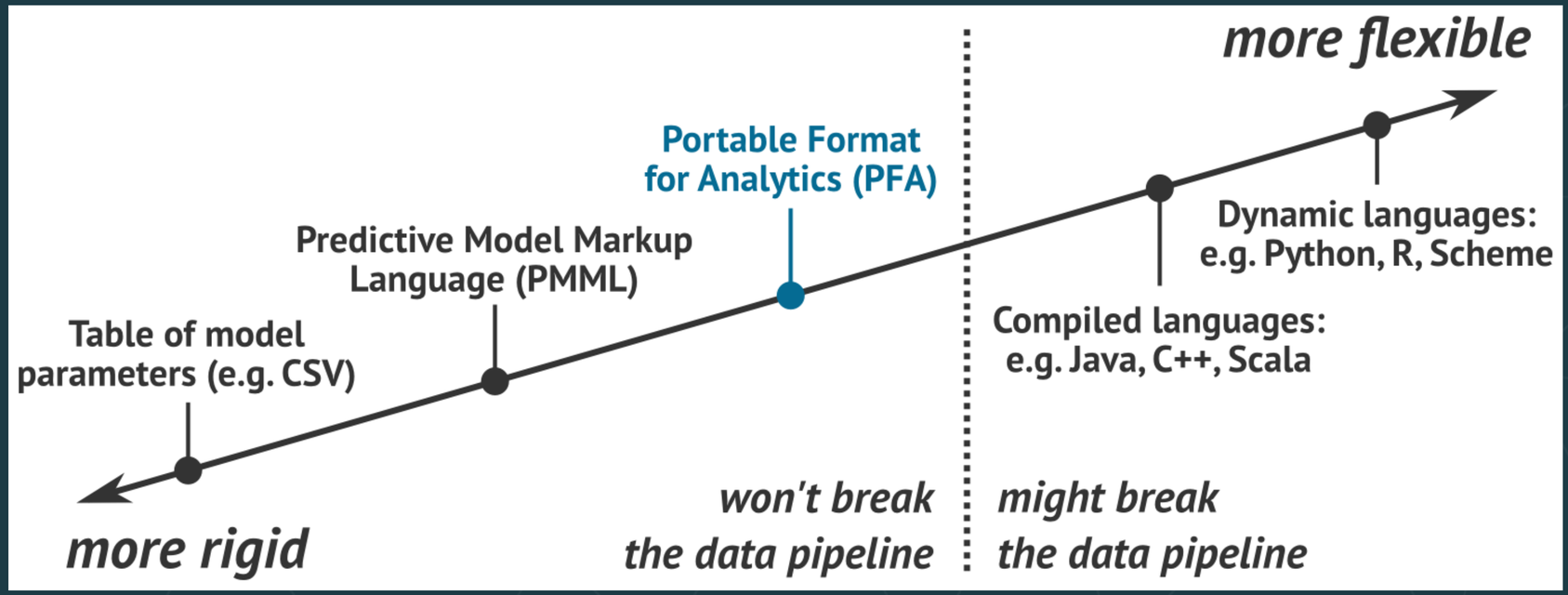
PMML



Implementations for:

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

PFA

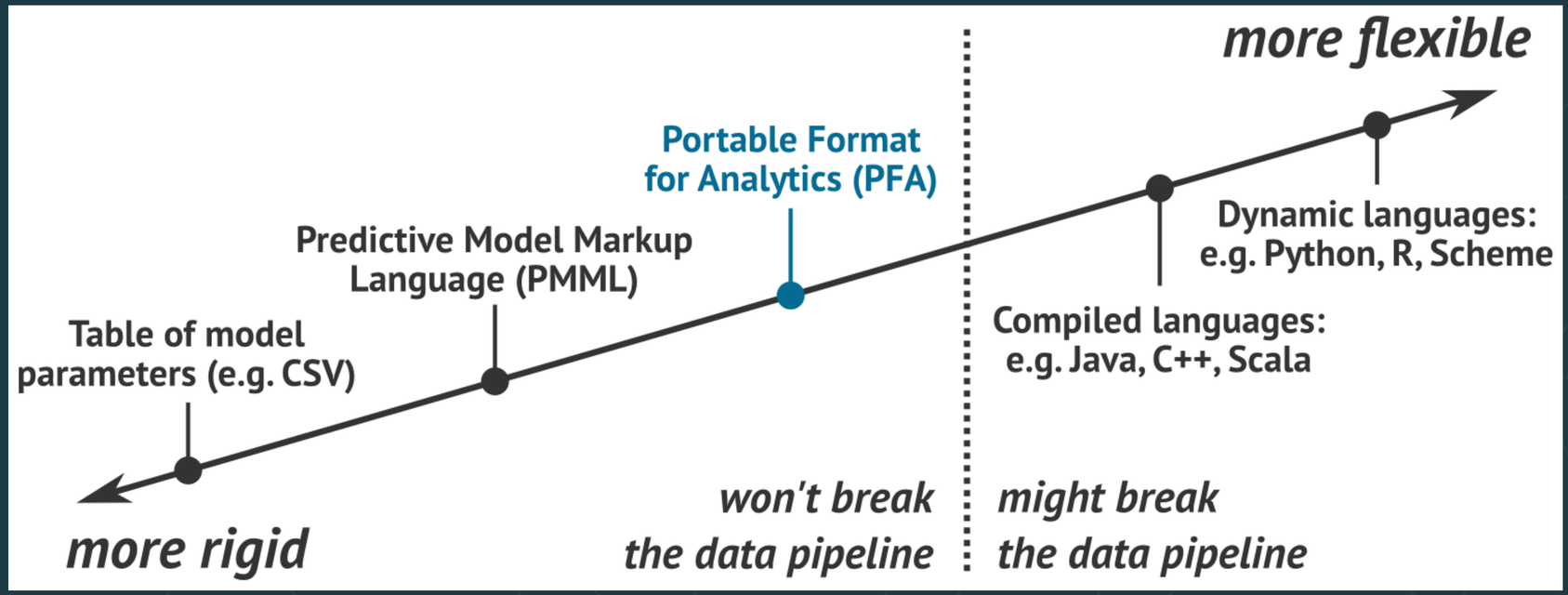


Portable Format for Analytics (PFA) is an emerging standard for statistical models and data transformation engines. PFA combines the ease of portability across systems with algorithmic flexibility: models, pre-processing, and post-processing are all functions that can be arbitrarily composed, chained,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

http://dmg.org/pfa/

docs/motivation/

PFA



Implementations for:

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

ONNX

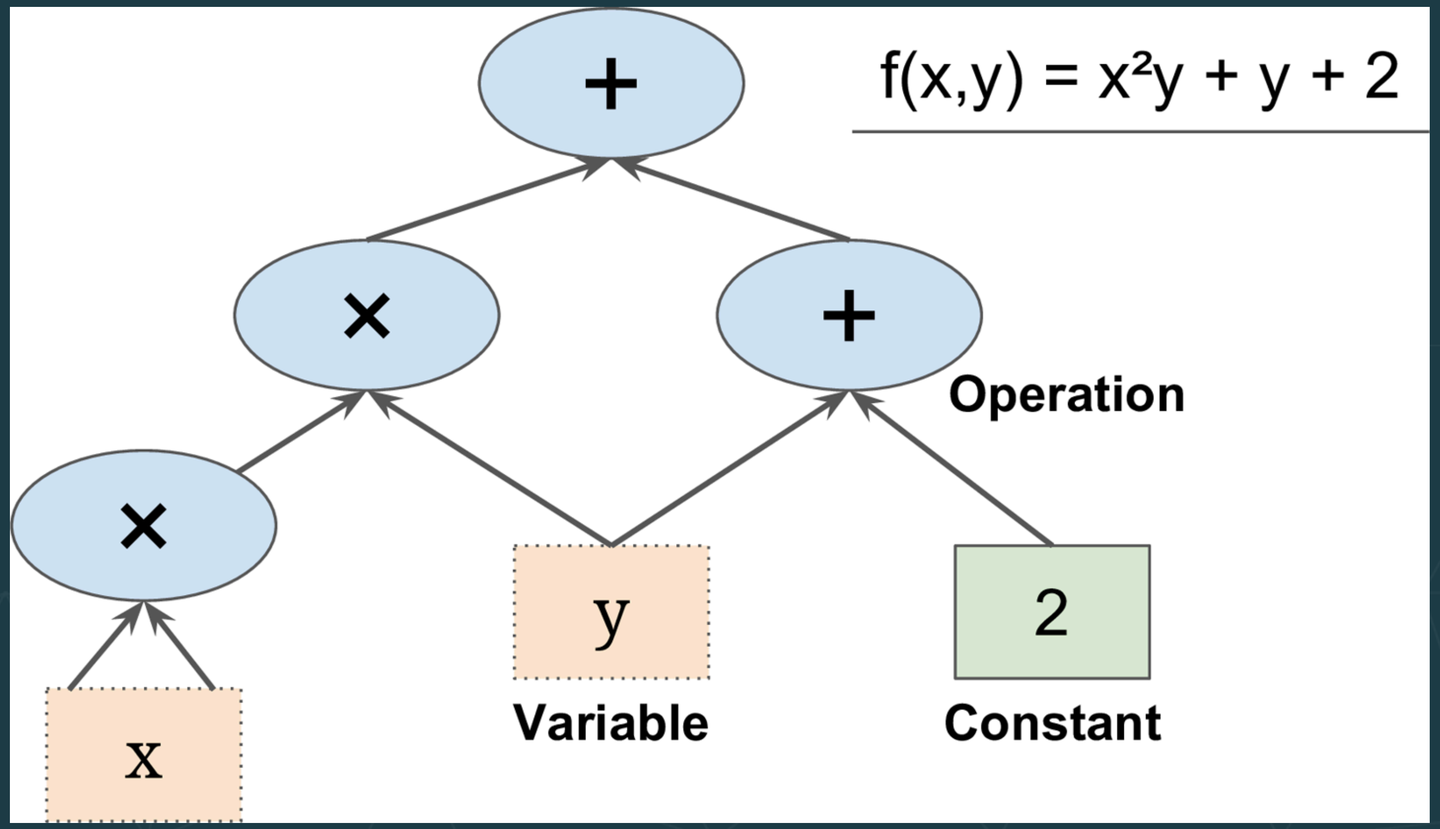
Open Neural Networks Exchange (ONNX) is an open standard format of machine learning models to offer interoperability between various AI frameworks. Led by Facebook, Microsoft, and AWS.

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

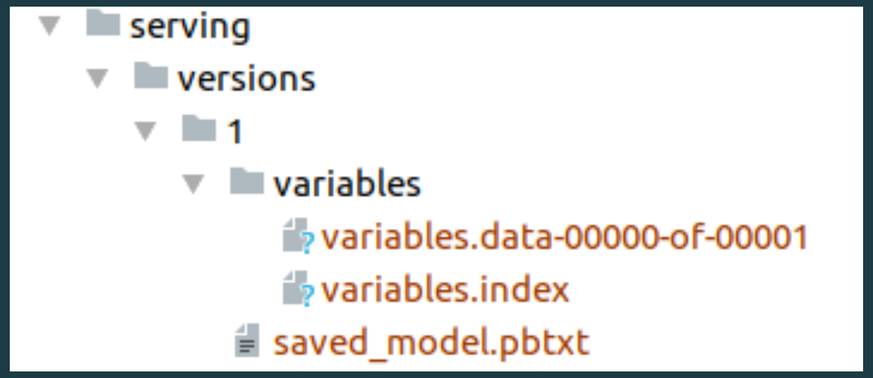
TensorFlow



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

https://learning.oreilly.com/library/view/hands-on-machine-learning/9781491962282/ch09.html

Two TensorFlow Export Formats



SavedModel - features:

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

TensorFlow Graph - normal, optimized format:

- Exports the Graph into a single file, that can be sent over Kafka or exchanged via file systems

Considerations for Interchange Tools

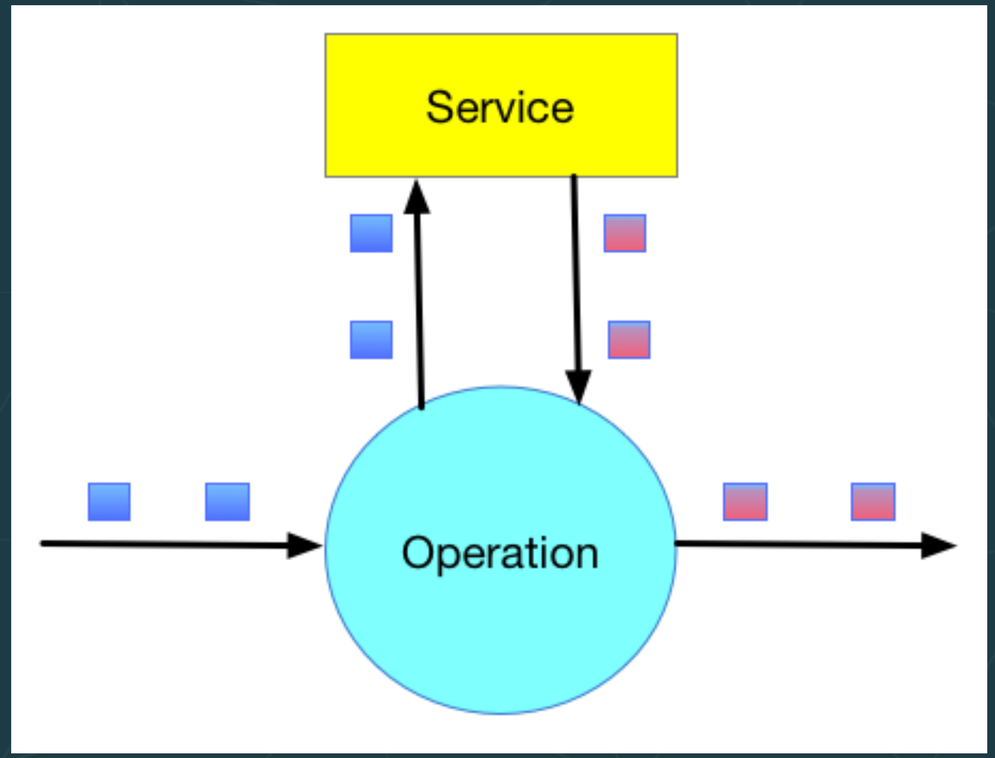
- For *heterogeneous* training and serving environments:
  - Do your *training* tools support exporting with a standard exchange format, e.g., PMML, PFA, etc.?
  - Do your *serving* tools support the same format for import?
  - Is there support on both ends for the model types you want to use, e.g.,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

Outline

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

Model Serving as a Service

- *Advantages*
  - Simplifies integration with other technologies and org. processes
  - Easier to understand if you come from a non-streaming world
- *Disadvantages*
  - Worse latency: remote invocations instead of local function calls
  - Less control: you app's SLA is at the mercy of a 3rd-party model-serving service



Model Serving as a Service: Challenges

- Launch ML runtime graphs, scale up/down, perform rolling updates
- Infrastructure optimization for ML
- Latency and throughput 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

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

Model Serving as a Servi

- Launch ML runtime gr
  updates
- Infrastr
- L
- C
- Allo
- Integ
- Allow
- Provid

We'll cover some of these points later. See also the Appendix

REST, gRPC

ation (CI)

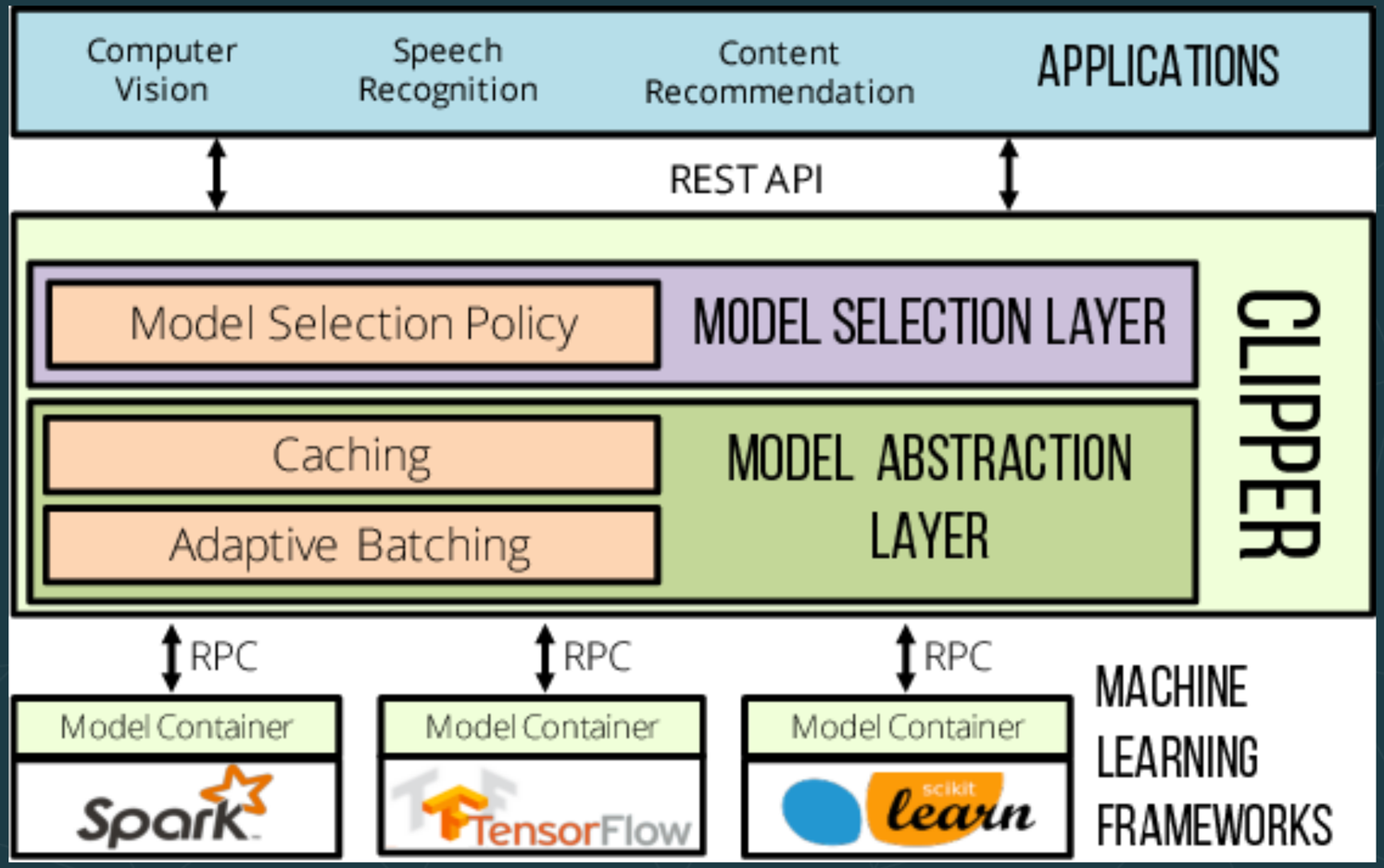
yment (CD)

ng

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

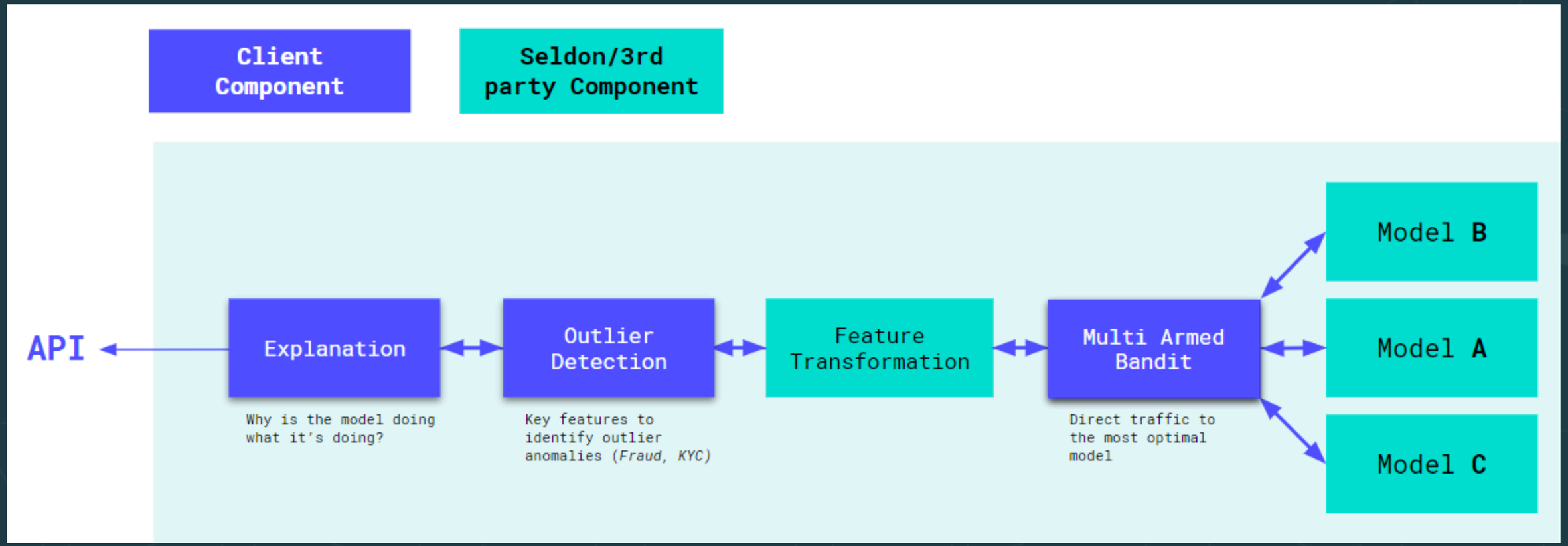
Example Systems for Model Serving as a Service

Example: Clipper



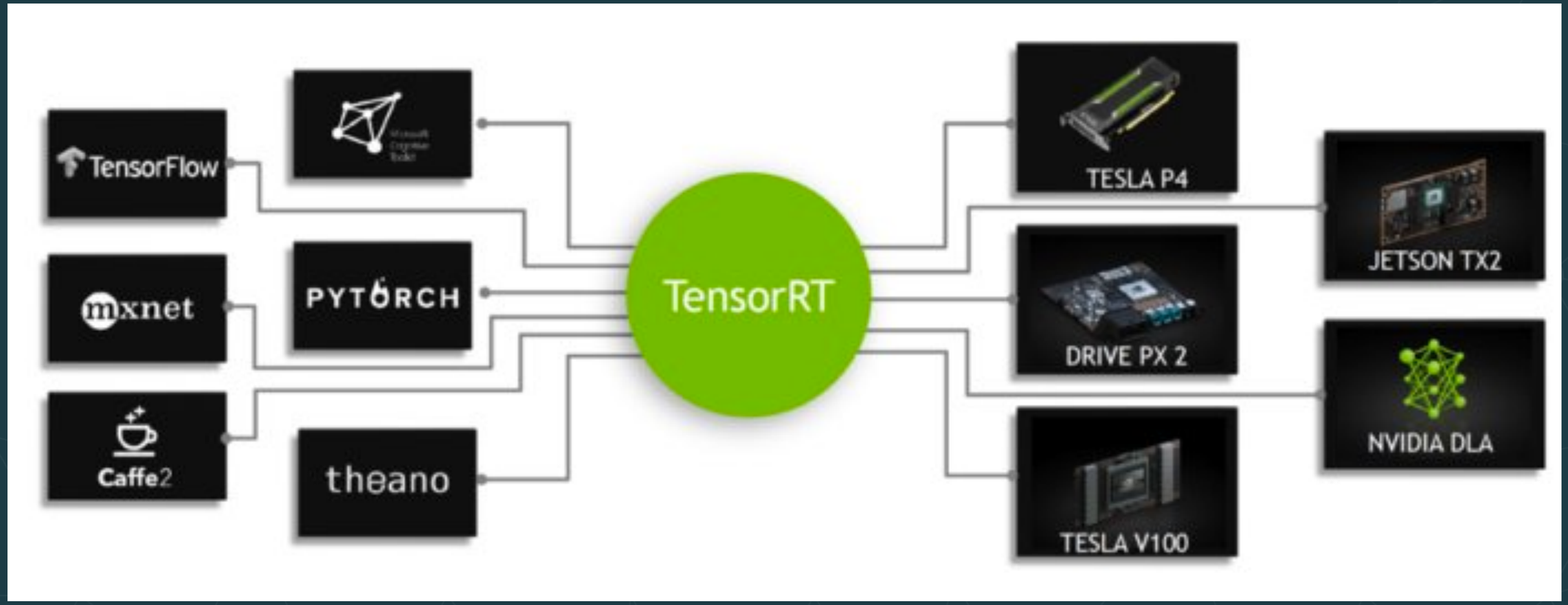
https://www.semanticscholar.org/paper/Clipper%3A-A-Low-Latency-Online-Prediction-Serving-Crankshaw-Wang/4ef862c9157ede9ff8cfbc80a612b6362dcb6e7c

Example: Seldon Core



https://www.seldon.io/open-source/

Example: TensorRT

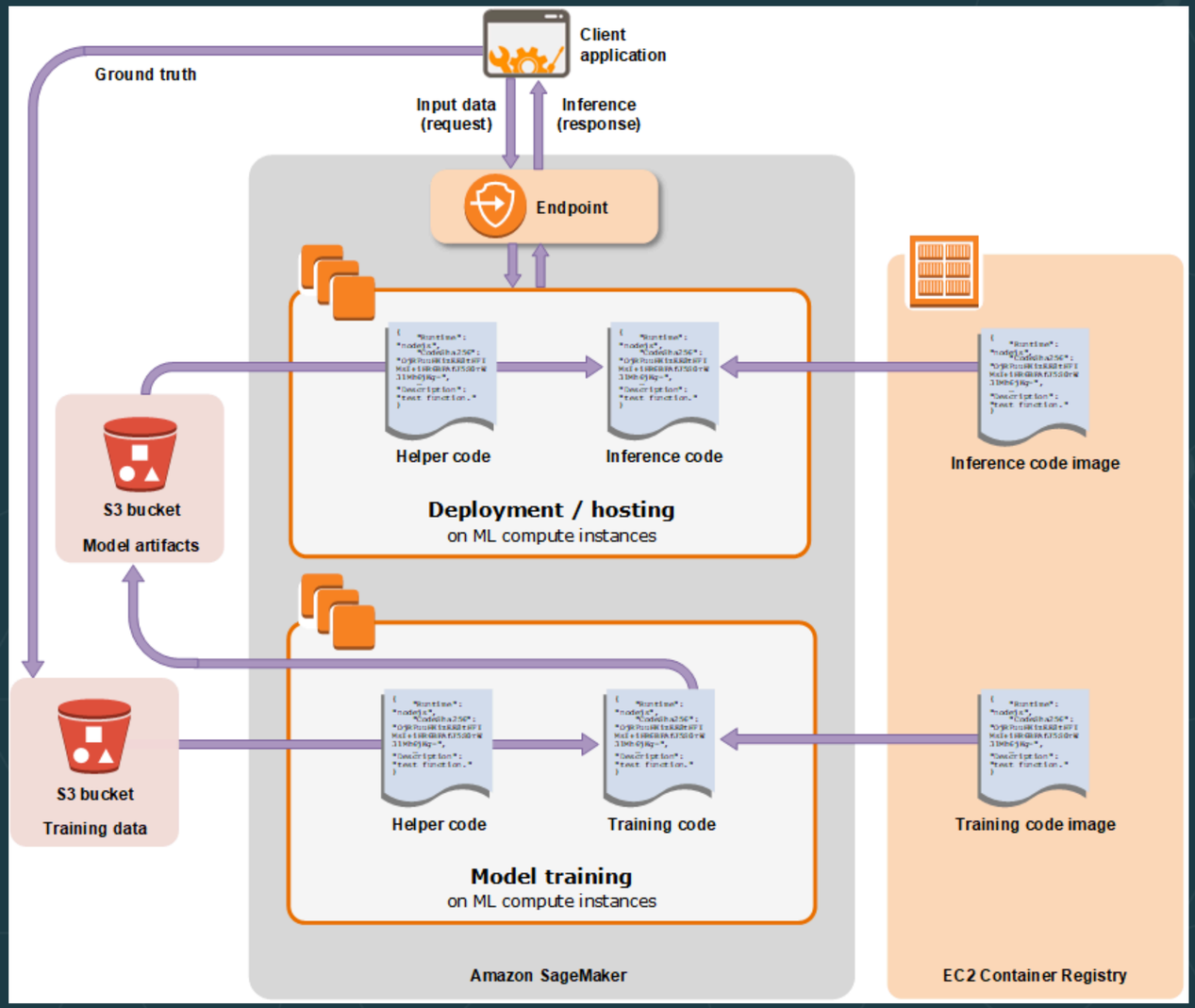


https://www.eetasia.com/news/article/Nvidia_CEO_in_China

Example: AWS Sagemaker

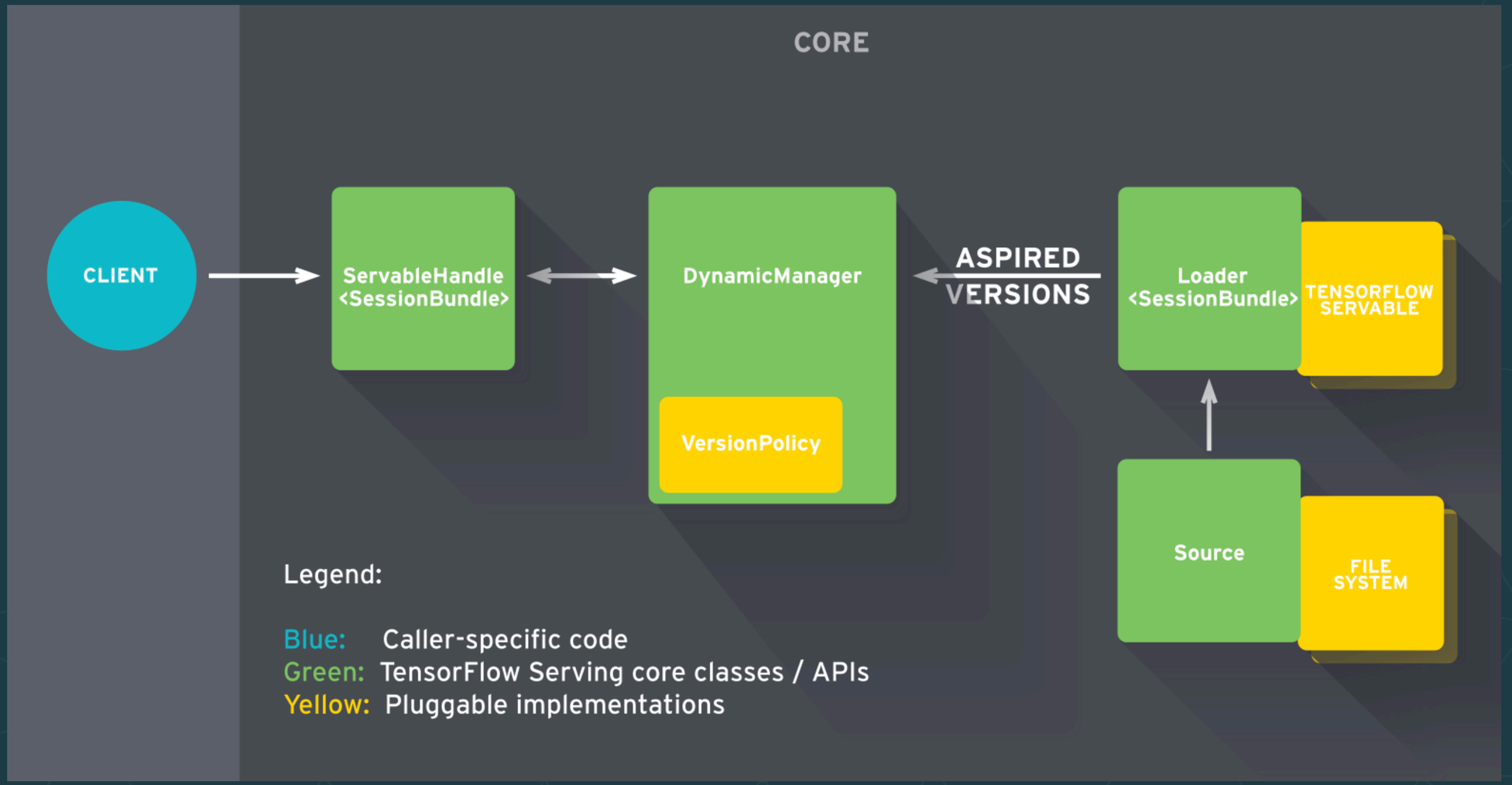
Capabilities include

- Canary testing
- A/B testing
- Deploying multiple models to the same endpoint
- Speculative serving
- Etc..



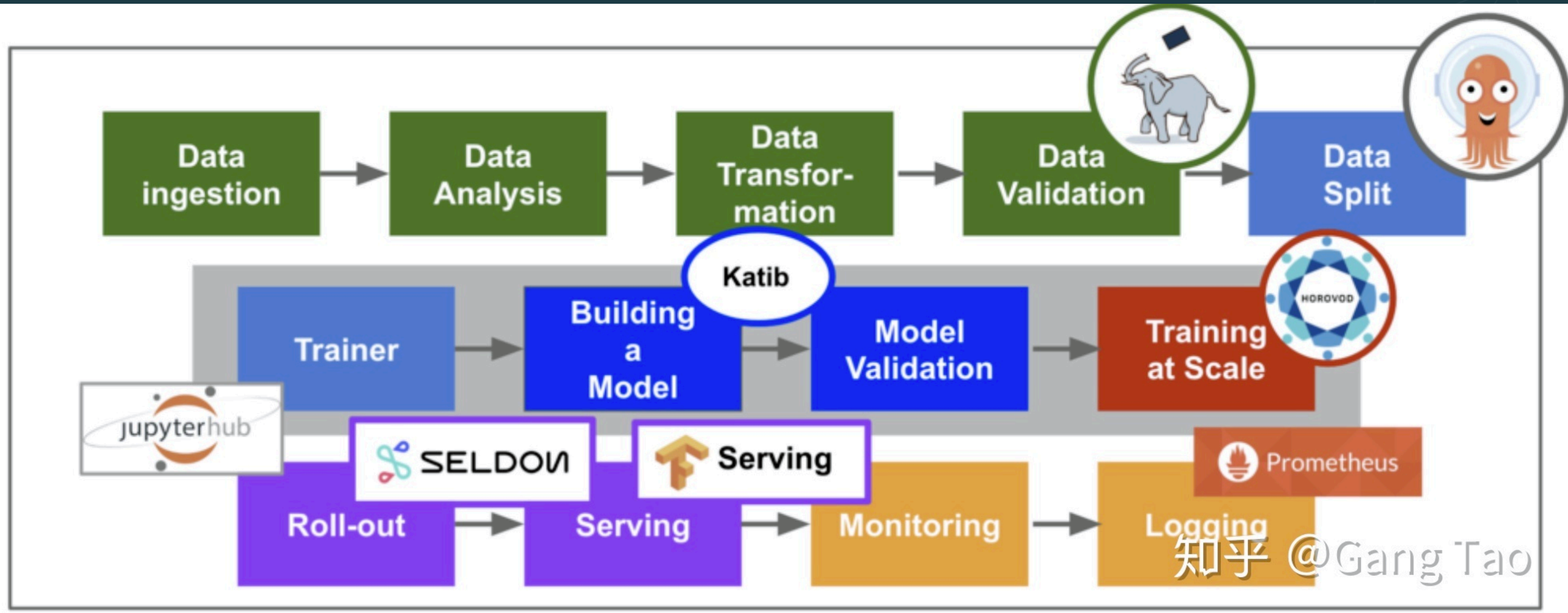
https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-training.html

Example: TensorFlow serving



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

Example: Kubeflow



https://zhuanlan.zhihu.com/p/44692757

Rendezvous Architecture Pattern

Handle ML logistics in a flexible, responsive, convenient, and realistic way.

- For Data:
  - Collect data at scale from many sources.
    - Preserve raw data so that potentially valuable features are not lost.
  - Make data available to many applications (consumers), both on premise and distributed.

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

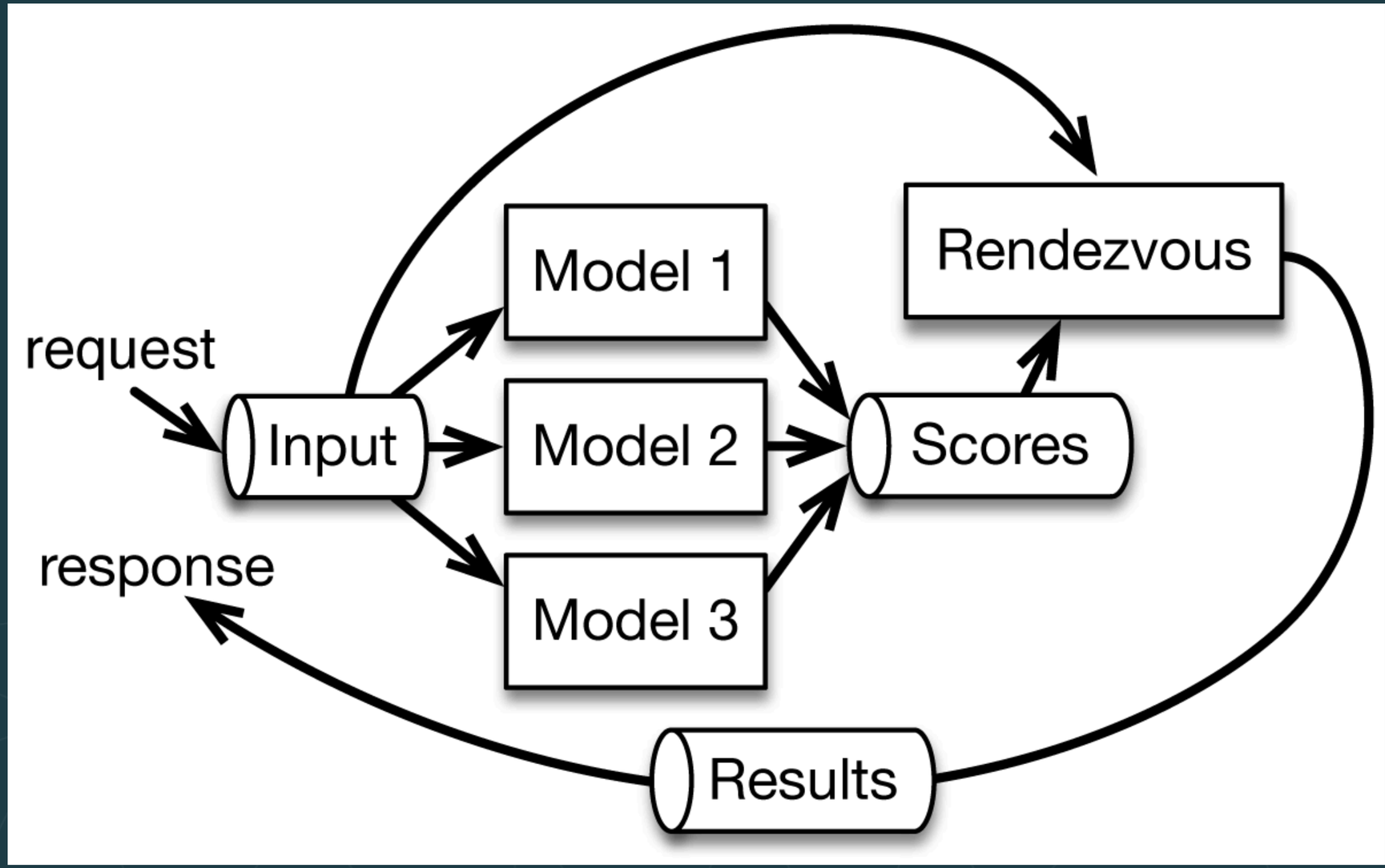
Rendezvous Architecture Pattern

Handle ML logistics in a flexible, responsive, convenient, and realistic way.

- Models:
  - 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.

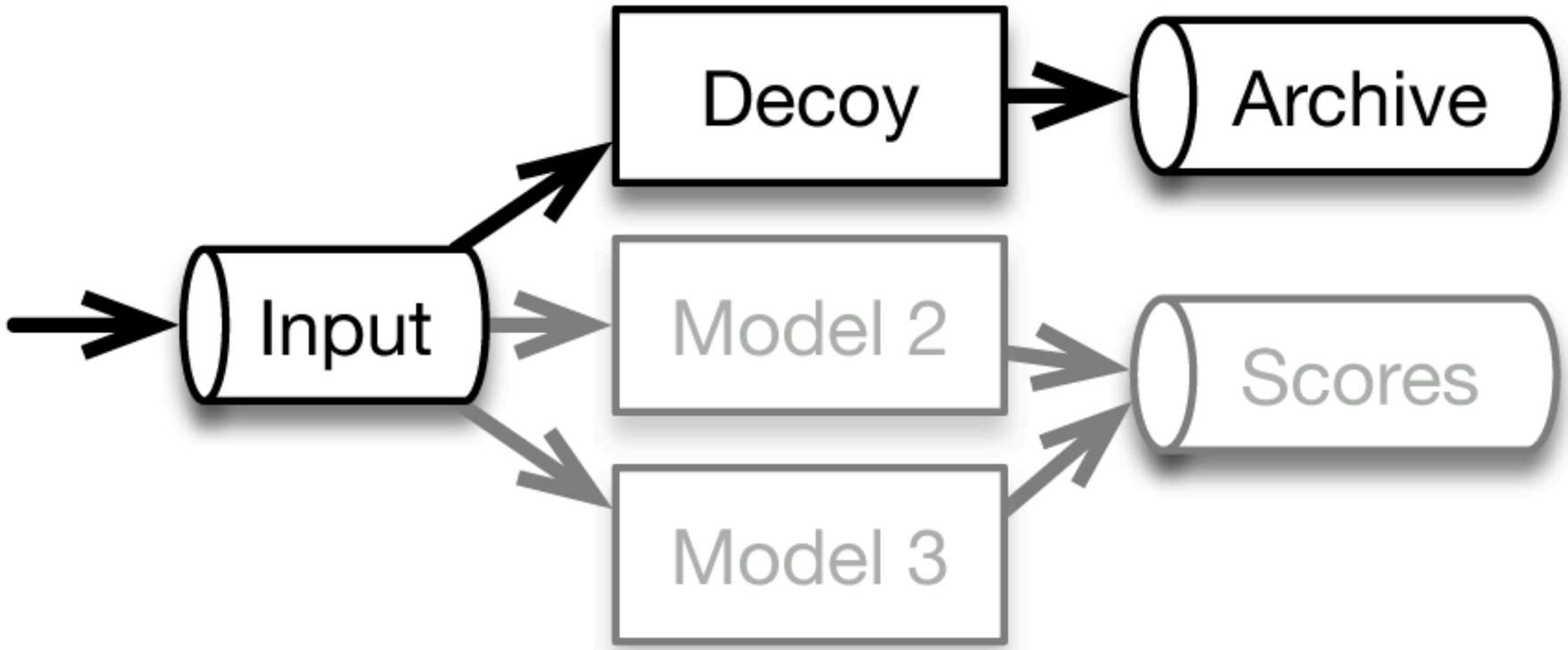
https://mapr.com/ebooks/machine-learning-logistics/ch03.html

Rendezvous Architecture



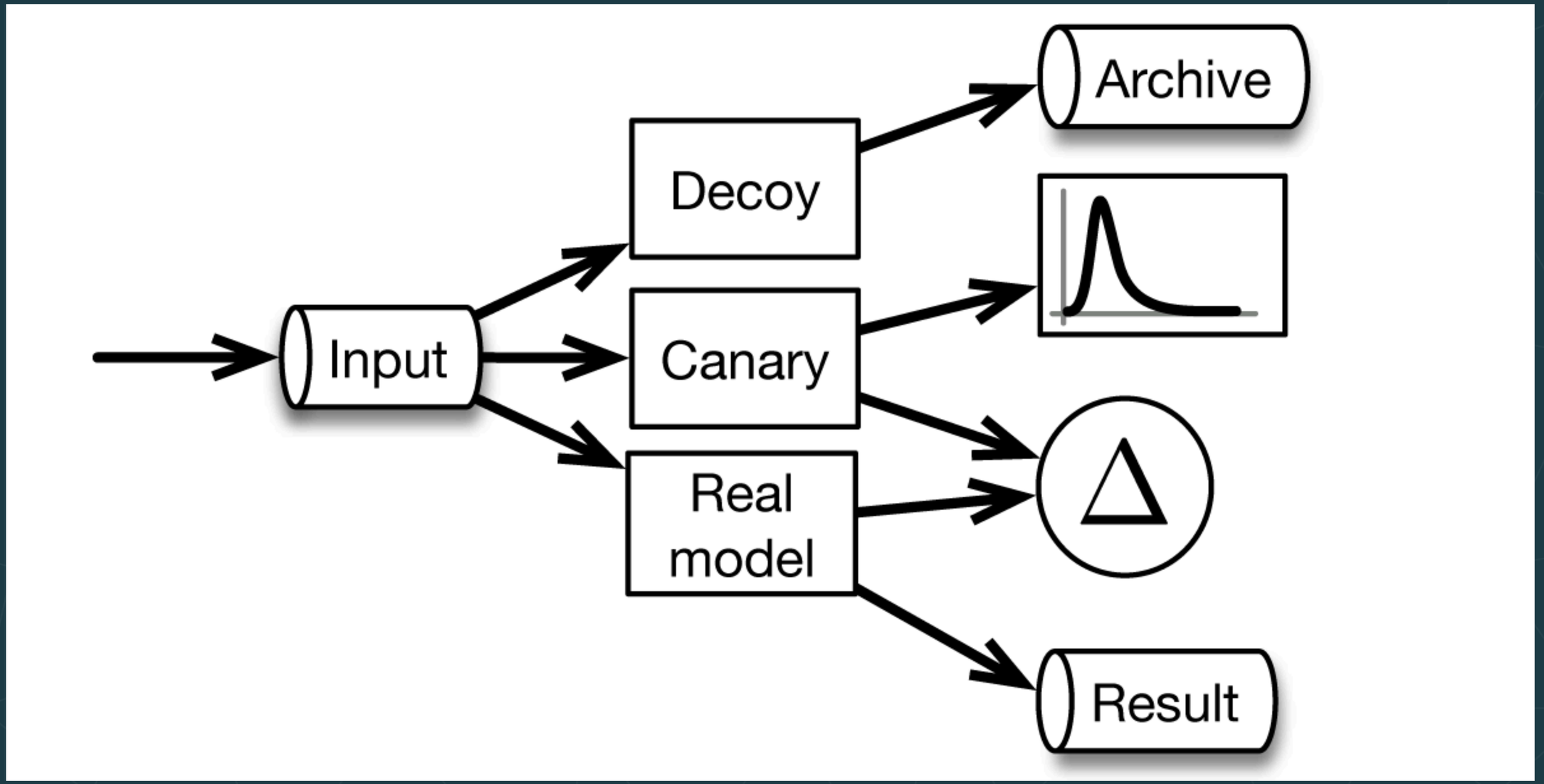
https://mapr.com/ebooks/machine-learning-logistics/ch03.html

Rendezvous Architecture - Decoy



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

Rendezvous Architecture - Canary



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

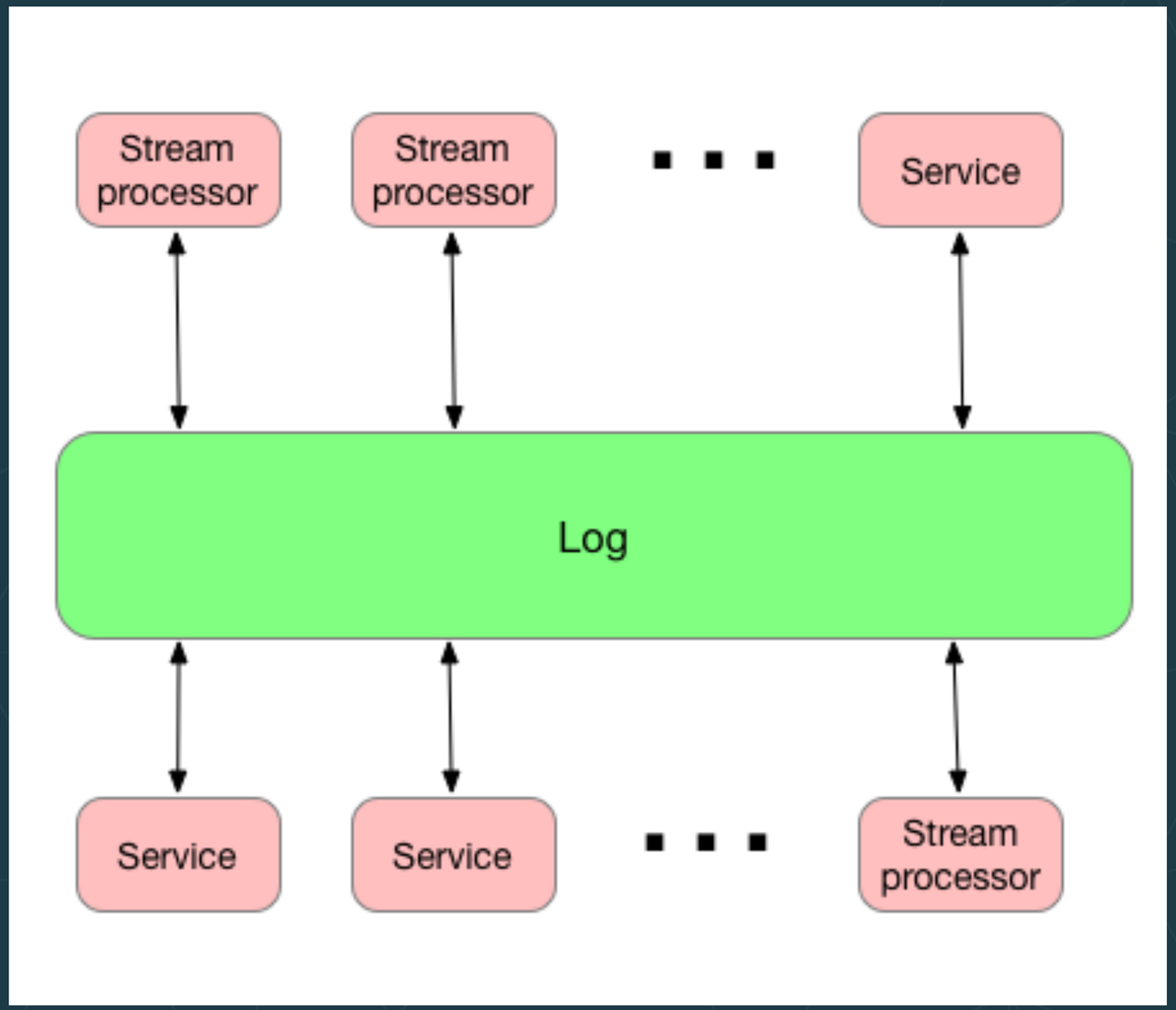
Outline

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

Log-Driven Enterprise

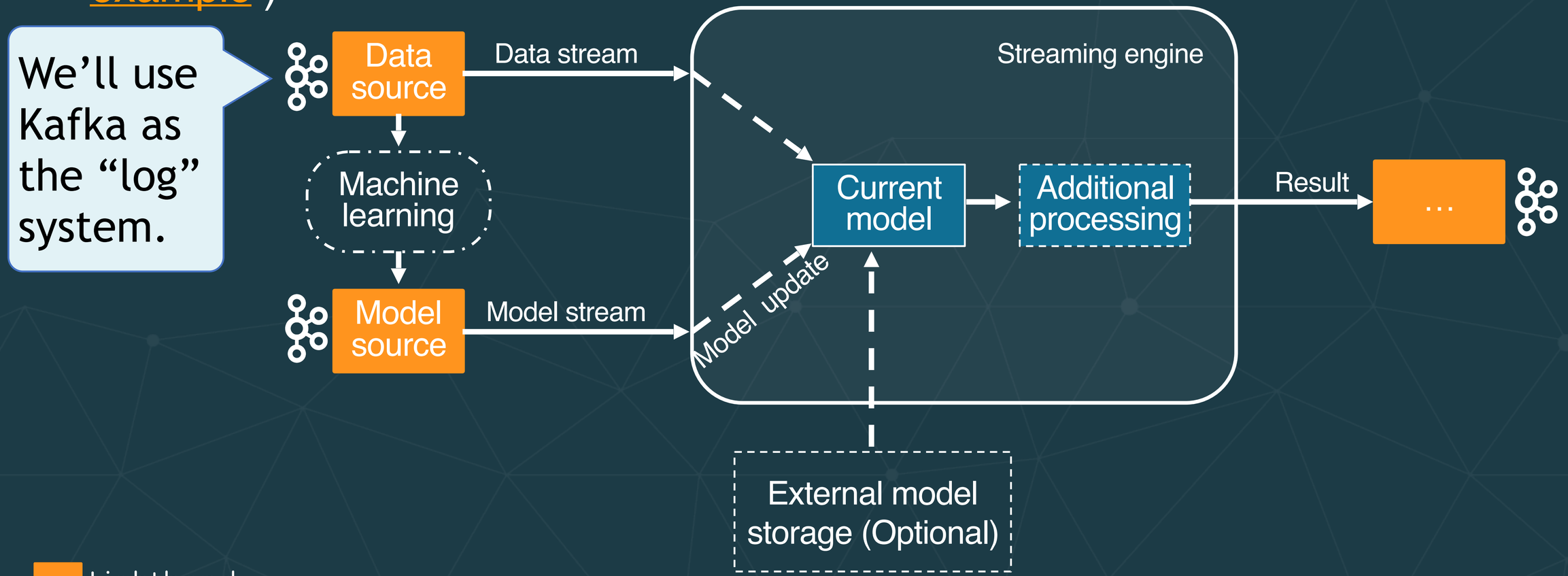
- Complete decoupling of services.
- All communications go through the log rather then services talking to each other directly.
- Specifically, stream processors don't talk explicitly to other services, but send async. messages through the log.

Example: Kafka



Model Serving in a Log-Driven Enterprise

Dynamically Controlled Stream: a streaming system supporting model updates without interruption of execution (Flink example, Spark streaming example)



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.

Model Code Abstraction (Scala)

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

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

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

See the “model” project in the example code.

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
)
```

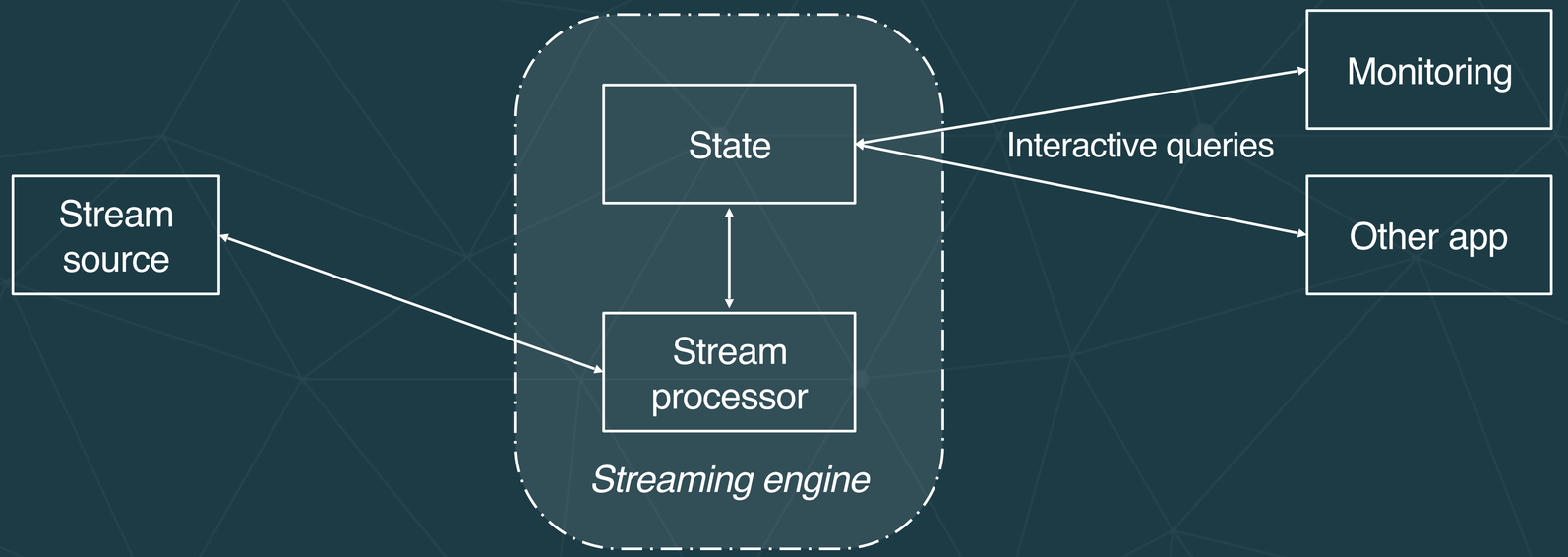
Lightbend

We'll return to production concerns…

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.



Example used in this tutorial

Throughout the rest of tutorial we use models based on *Wine quality* data for training and scoring.

- The data is publicly available at https://www.kaggle.com/vishalyo990/prediction-of-quality-of-wine/data .
- A great notebook describing this data and providing several machine learning algorithms for this problem: https://www.kaggle.com/vishalyo990/prediction-of-quality-of-wine/notebook

Akka Streams

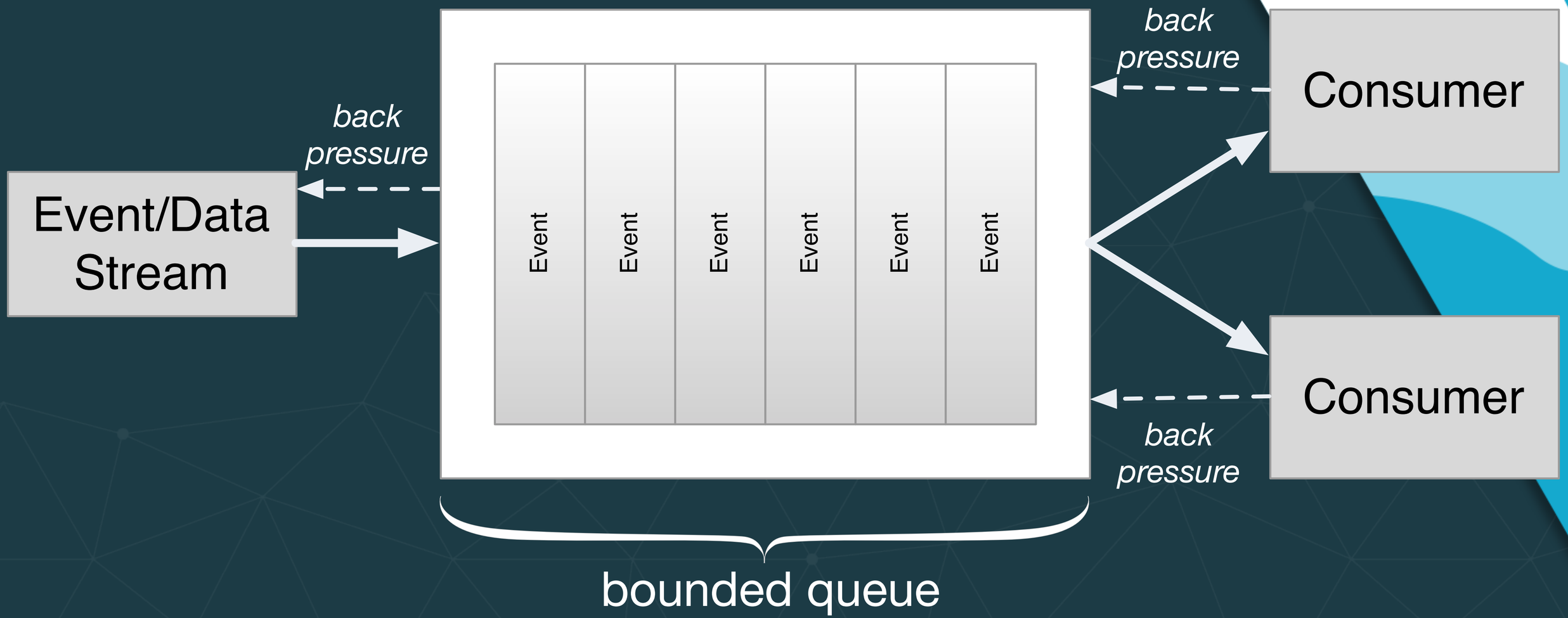
TF Serving is Great! How Do I Use It?

- How do we integrate model serving (or any other new stateful capability) into our microservices that need to do scoring?
  - I.e., can we embed a *library* in our microservice to provide streaming semantics - in this case to do scoring?
- Let's see, using *Akka*.

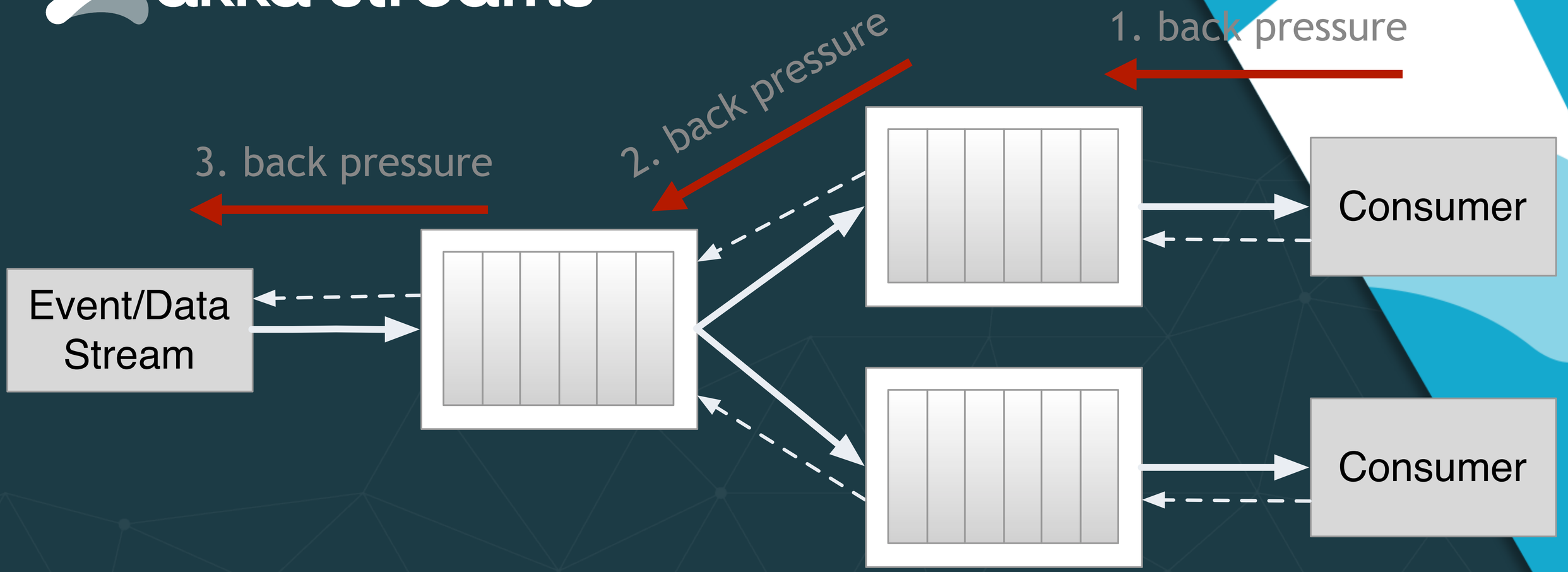
- A *library*
- Implements Reactive Streams.
  - http://www.reactive-streams.org/
  - *Back pressure* for flow control

- We'll use this for streaming data *microservices*.

akka streams



akka streams



… and they compose

akka streams

- Part of the *Akka ecosystem*
  - Akka Actors, Akka Cluster, Akka HTTP, Akka Persistence, …
  - Alpakka - rich connection library
  - similar to Camel, but implements Reactive Streams
  - Commercial support from Lightbend

- 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
```

```
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
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))
```

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

Create a **source** of Ints. Second type represents a hook used for “materialization” - not used here

```
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.runWithSink.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**)

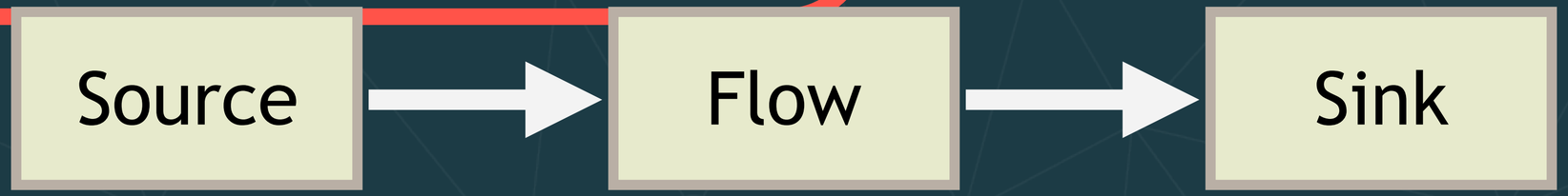
```
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._

implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()

val source: Source[Int, NotUsed] = Source(1
val factorials = source.scan(BigInt(1)) ( (acc
factorials.runWithSink.foreach(println))
```

Output to a **sink**, and run it

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

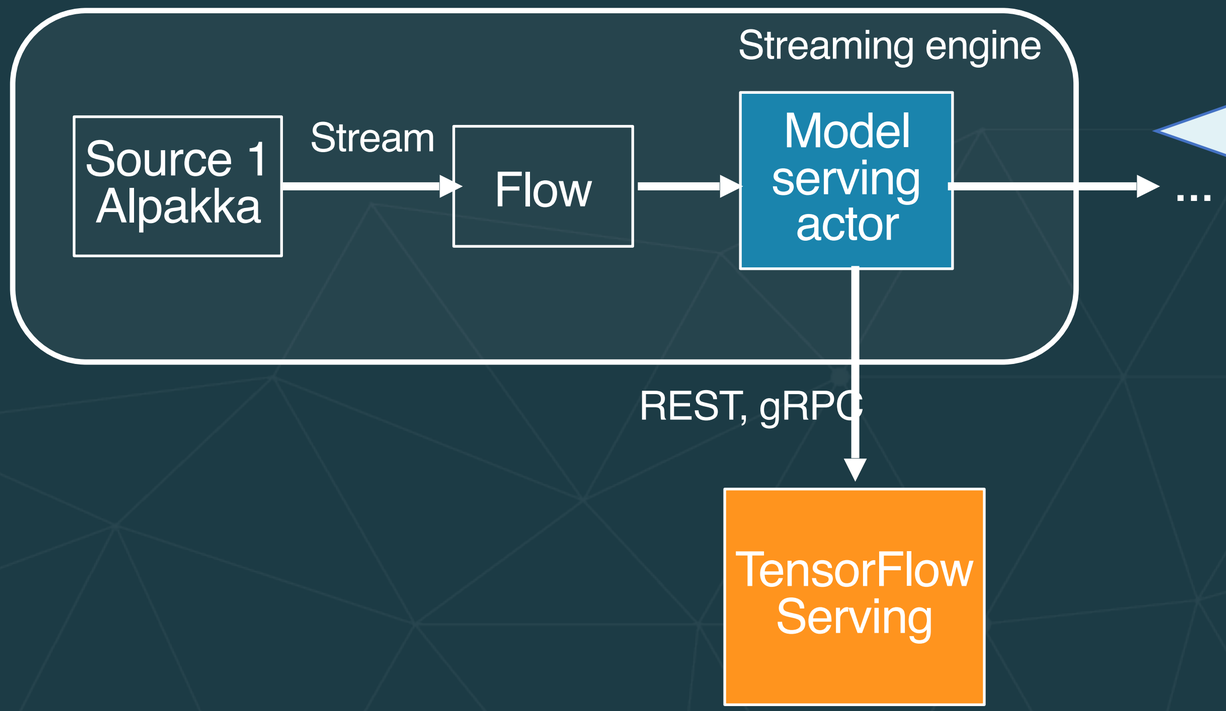
```
1
2
6
24
120
720
0
20
880
8800
```

A **source**, **flow**, and **sink** constitute a **graph**

TensorFlow Serving

Using TensorFlow Serving in Akka Strea

Use an *Akka Actor* to invoke TensorFlow Serving *asynchronously* (i.e., model serving as a service)



Actor holds the current state in memory - model parameters, scoring stats, …

Could use this idiom for any *X as a Service* from a streaming data microservice

TensorFlow Serving

Code time

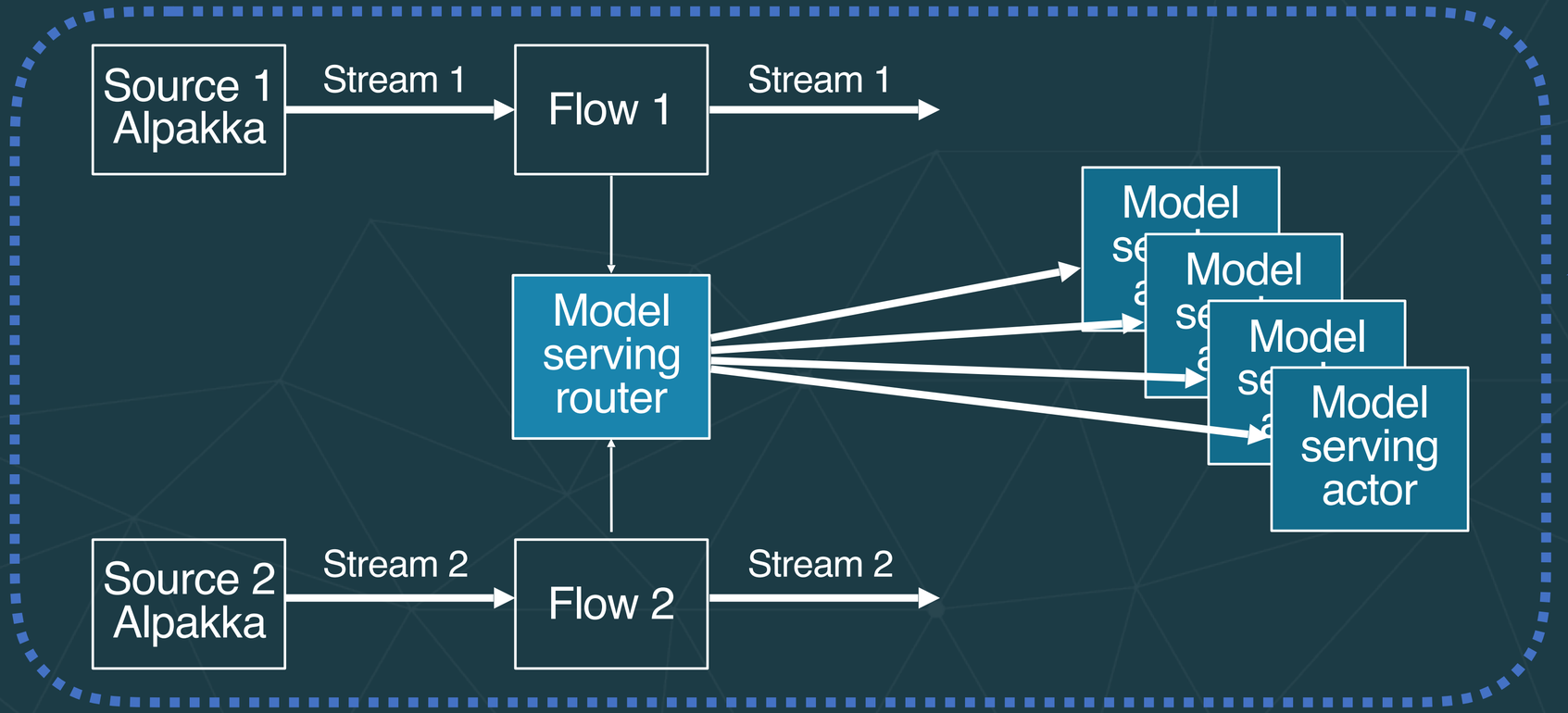
- Let’s look at TF Serving first. We’ll use it with microservice shortly.
  - Open the example code project
  - We’ll walk through the project at a high level
  - Familiarize yourself with the *tensorflowserver* code
  - Load and start the TensorFlow model serving Docker image
    - See Using TensorFlow Serving in the README
  - Try the implementation. What questions do you have?

Embedded Model Serving

- Use *Akka Actors* to implement model serving within the microservice boundary, using a *library*.
  - Alternative to Model Serving as a Service

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!!



Akka Streams Example

Code time

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

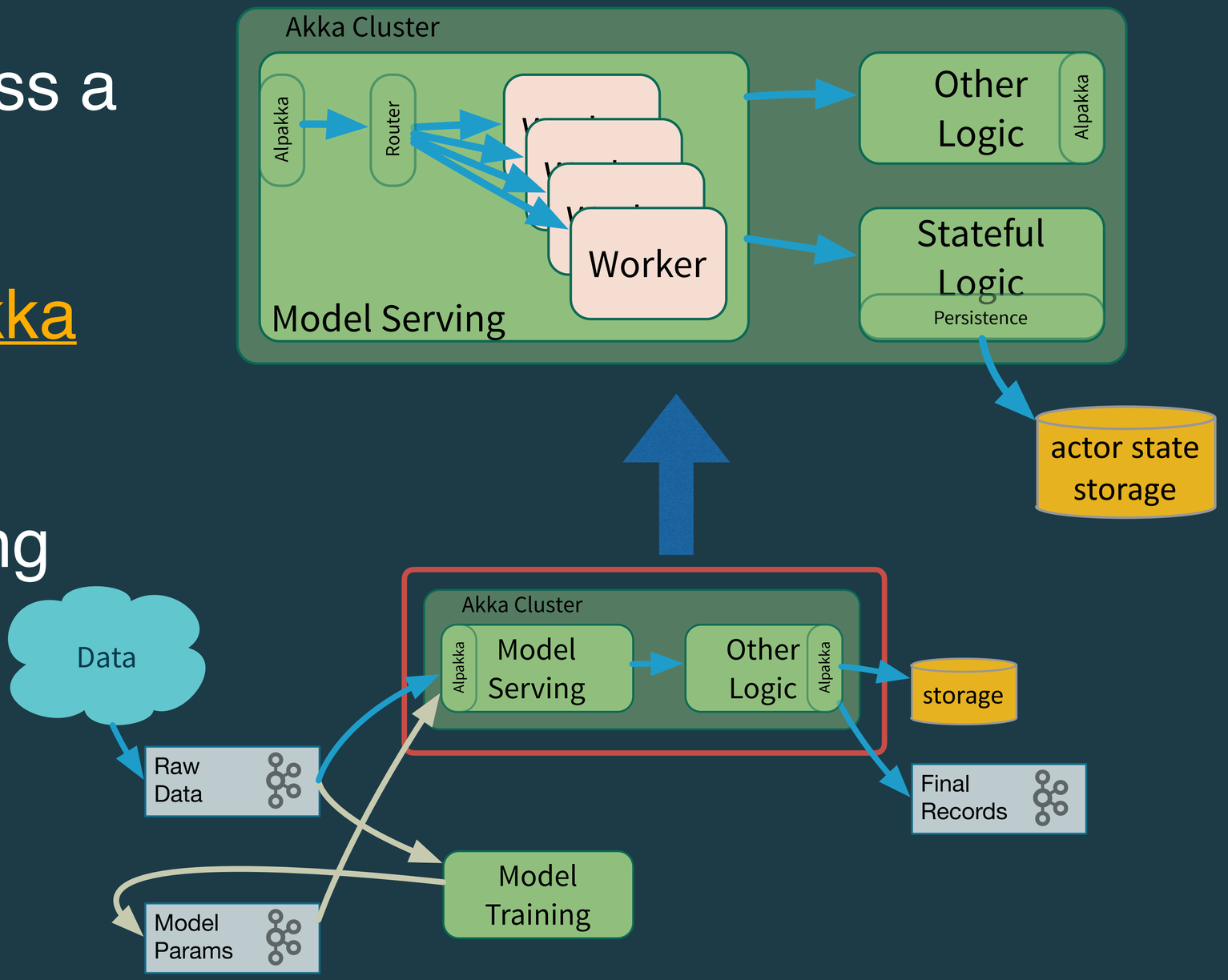
Akka Streams Example

- Implements *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

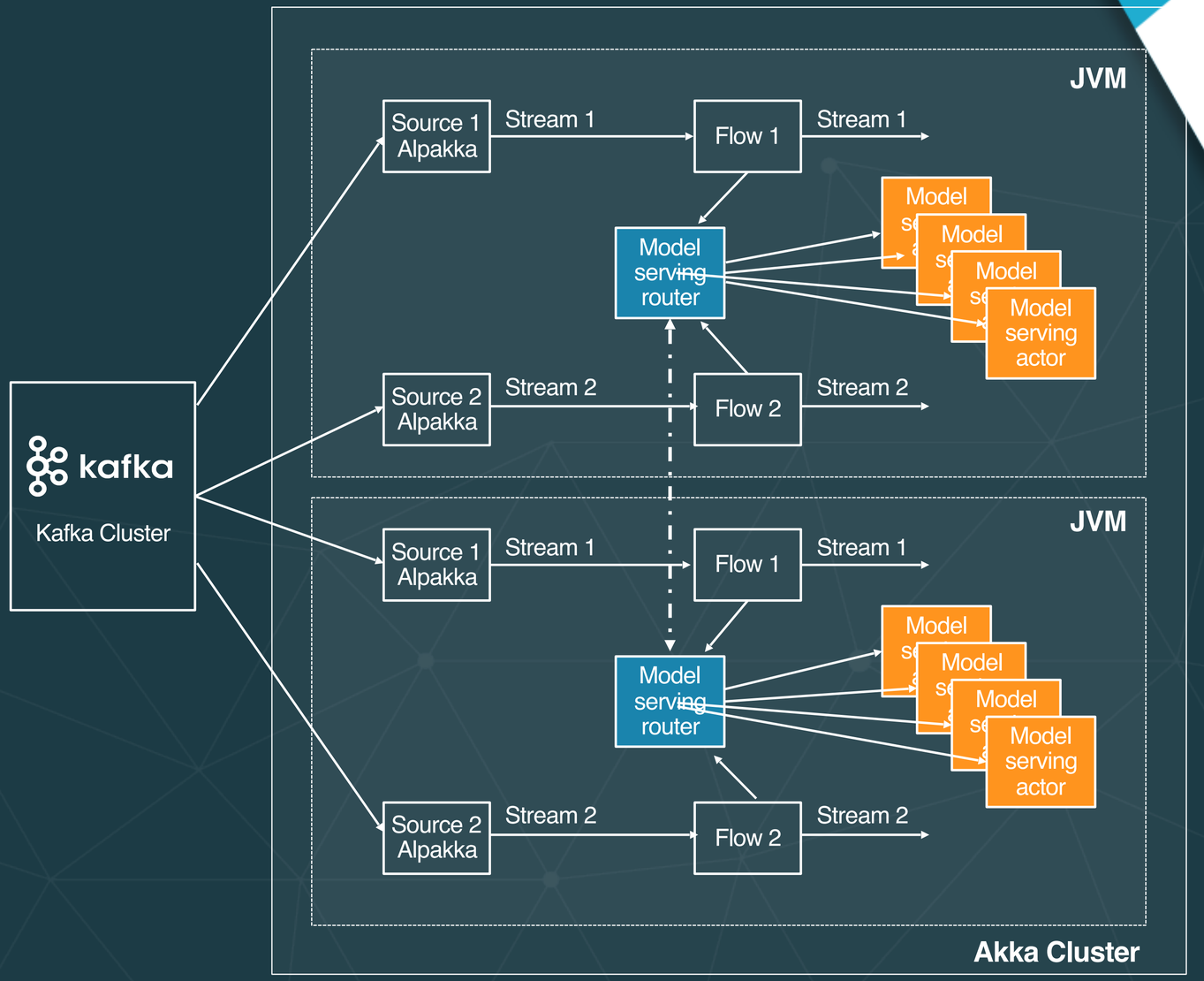
- Scale scoring with more workers and routers, across a cluster
- Persist actor state with Akka Persistence
- Connect to *almost* anything with Alpakka



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.



Lightbend

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

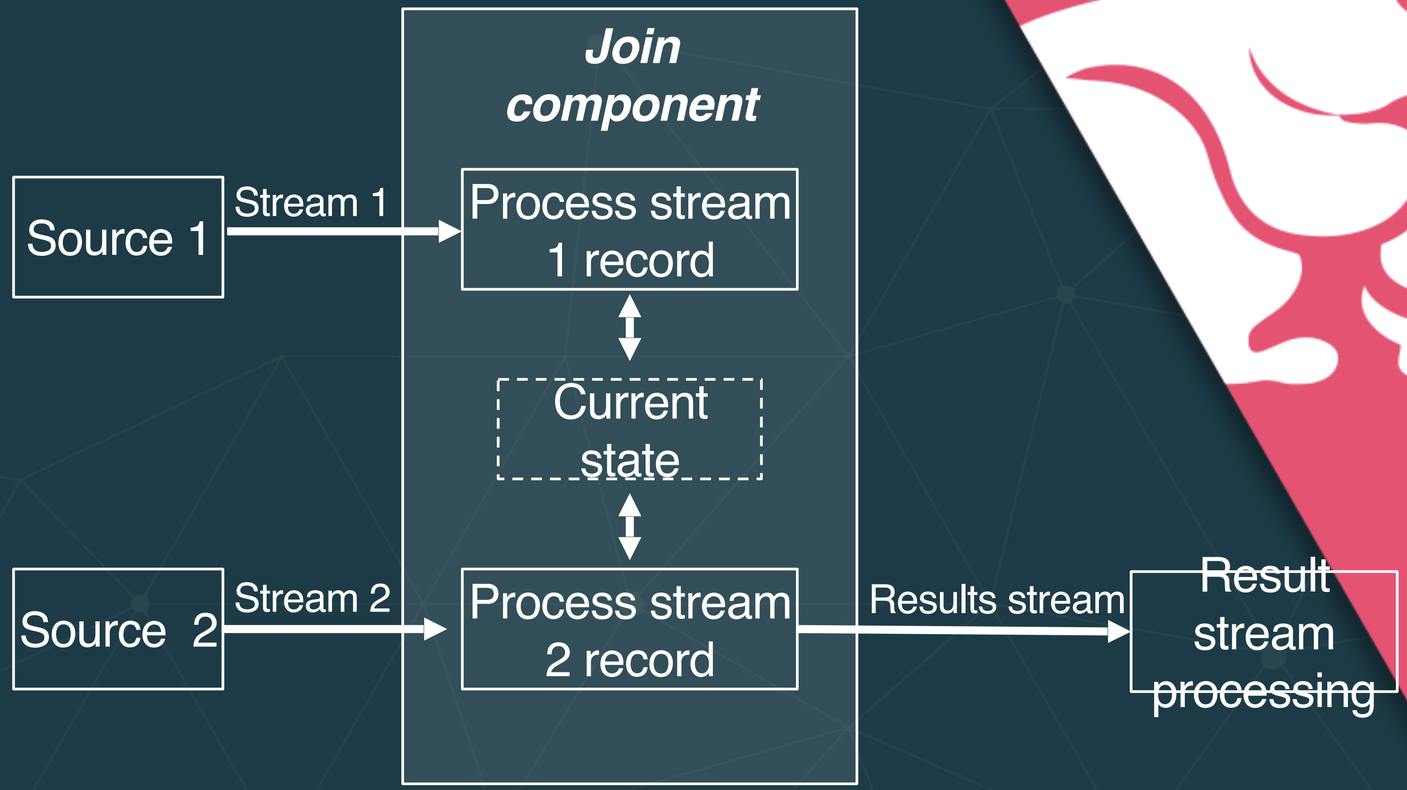
Apache Flink

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

- Scales to thousands of nodes.
- Provides *checkpointing* and *save-pointing* facilities for fault tolerance, e.g., restarting without loss of *accumulated state*.
- Provides *queryable state* support; avoid needing an external database to expose state outside the app.
- Provides *window semantics;* enables calculation of accurate aggregations, even for out-of-order or late-arriving data.

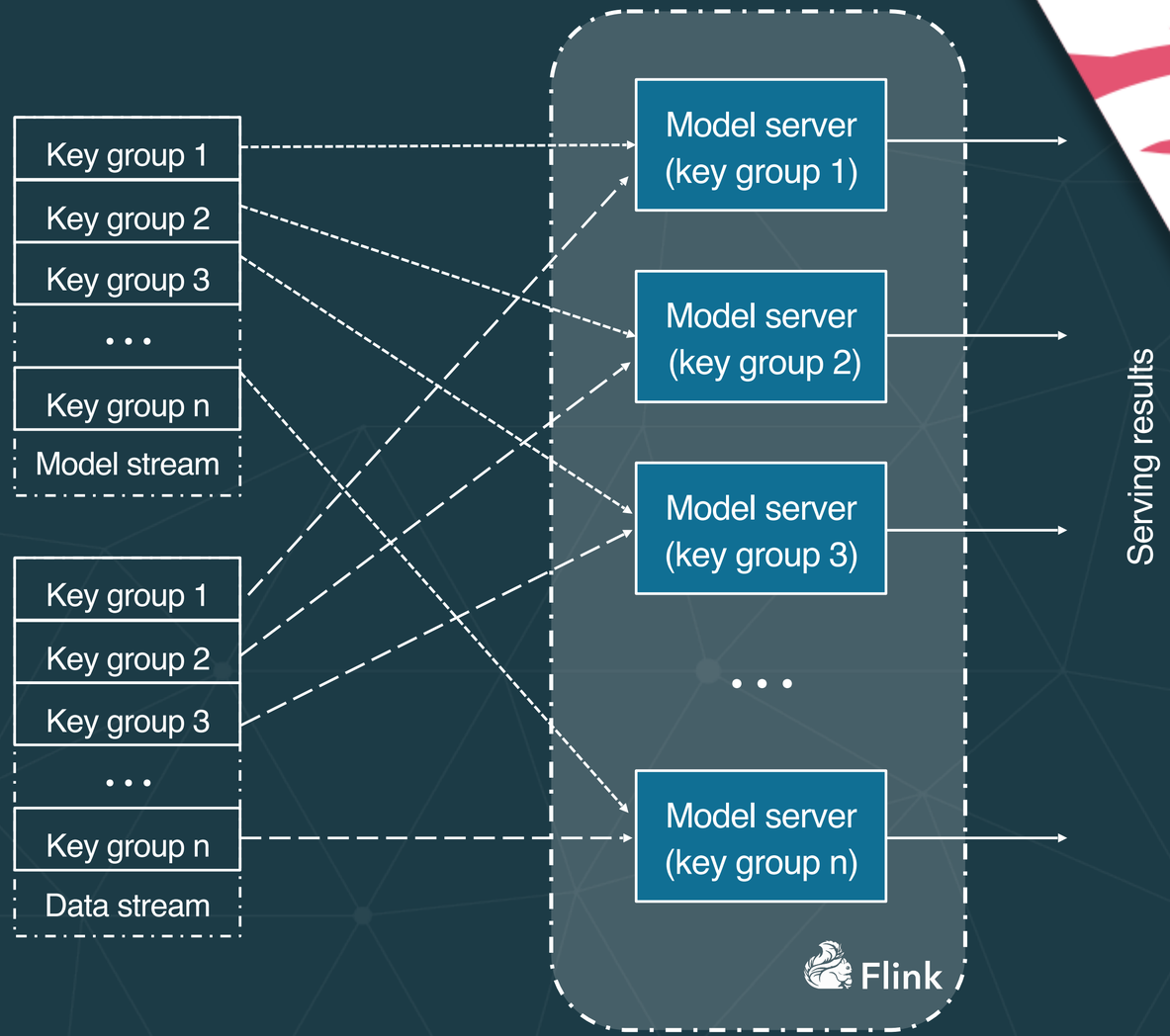
Flink Low Level Join

- 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
- Flink provides two implementation of low-level joins: key based join and partition based join



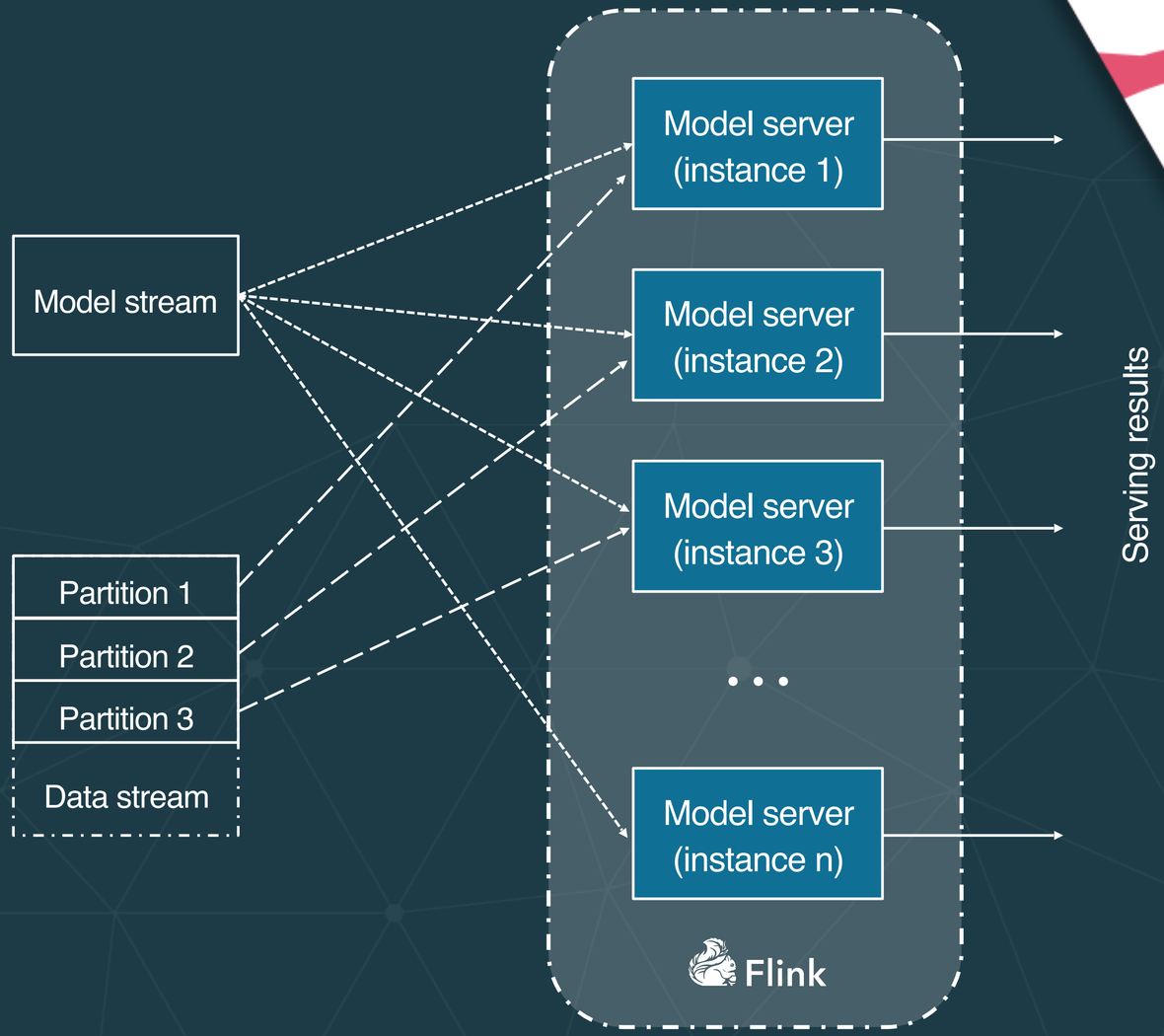
Key based join

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 parallelization parameter). When using this API, on the partitioned stream, data from different partitions is processed by dedicated Flink executor.



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

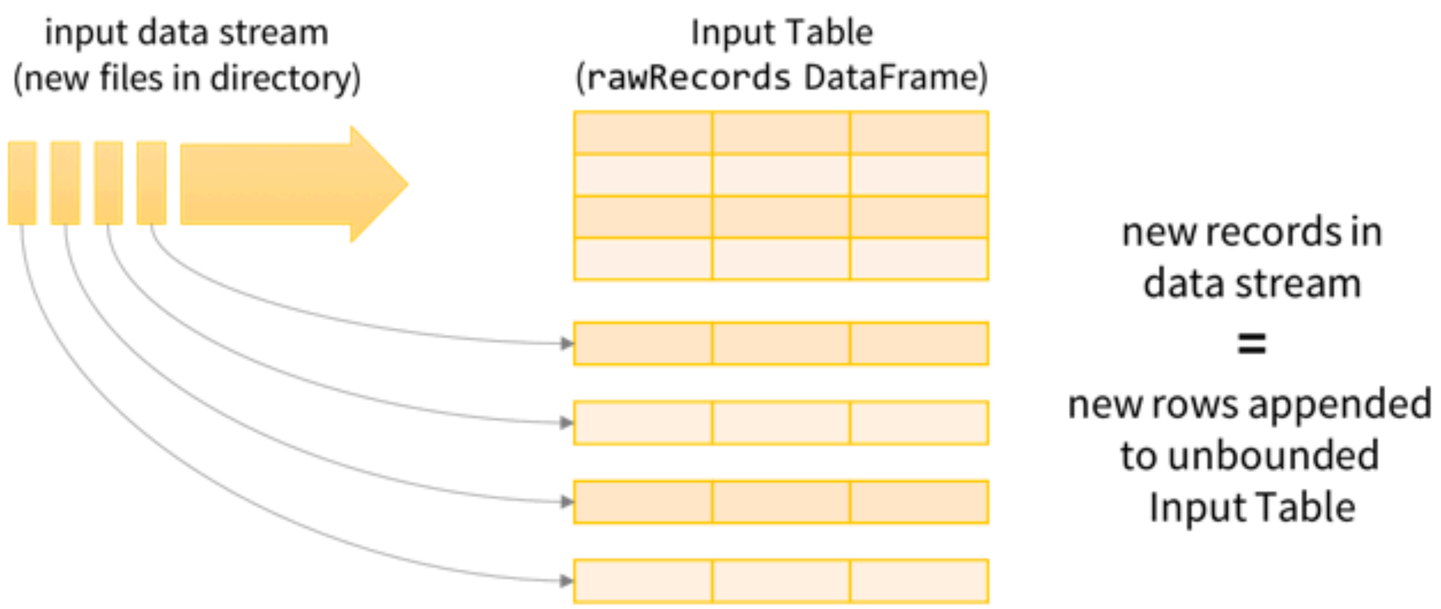
Apache Spark Structured Streaming

Spark Structured Streaming

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

- Scales to thousands of nodes.
- Express your streaming computation the same way you would express a SQL computation on static data:
  - The Spark SQL engine will take care of running it incrementally and continuously. It updates results as streaming data continues to arrive.
  - Adds streaming SQL extensions, like event-time windows.

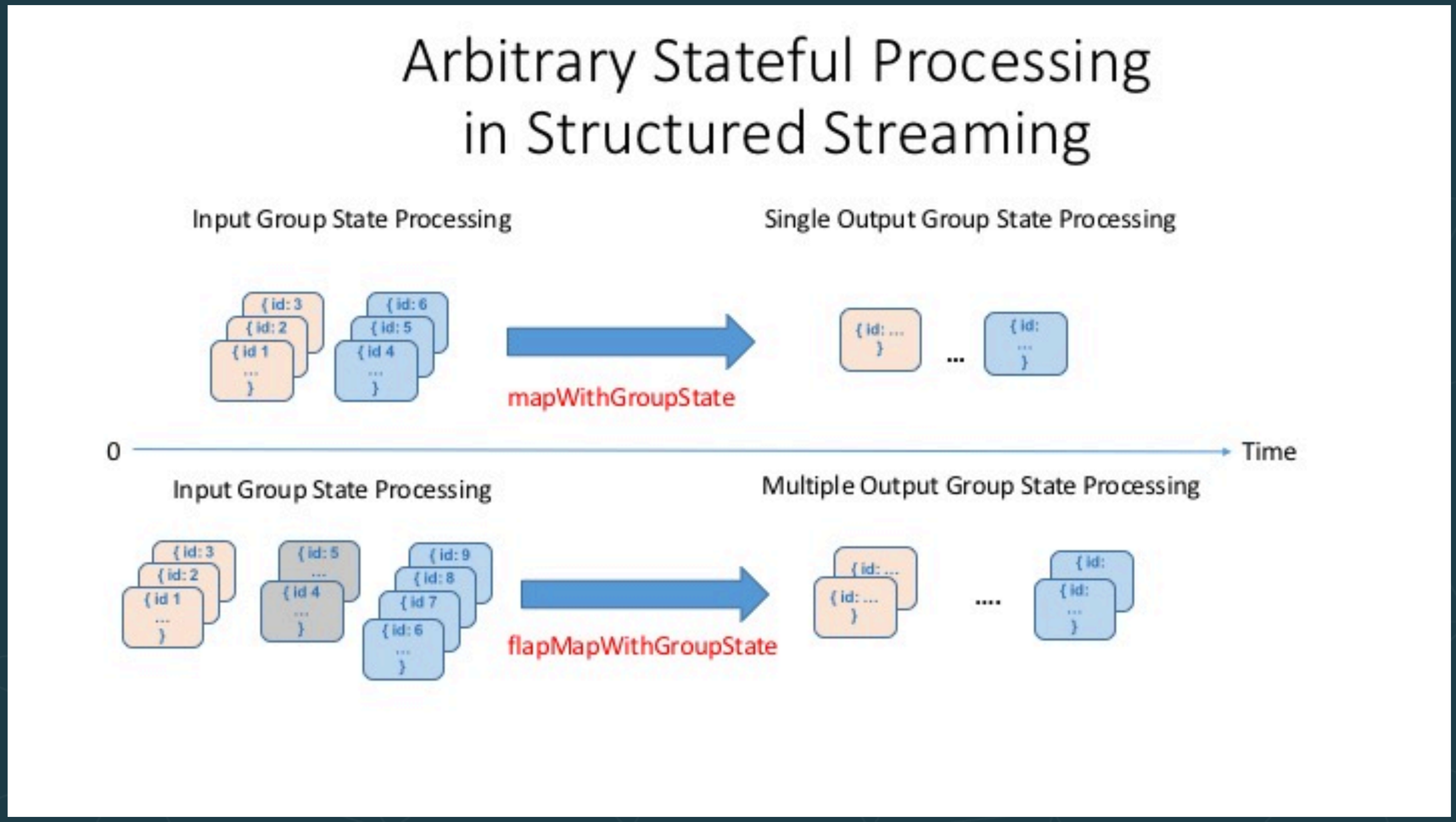
Spark Structured Streaming



Structured Streaming Model
treat data streams as unbounded tables

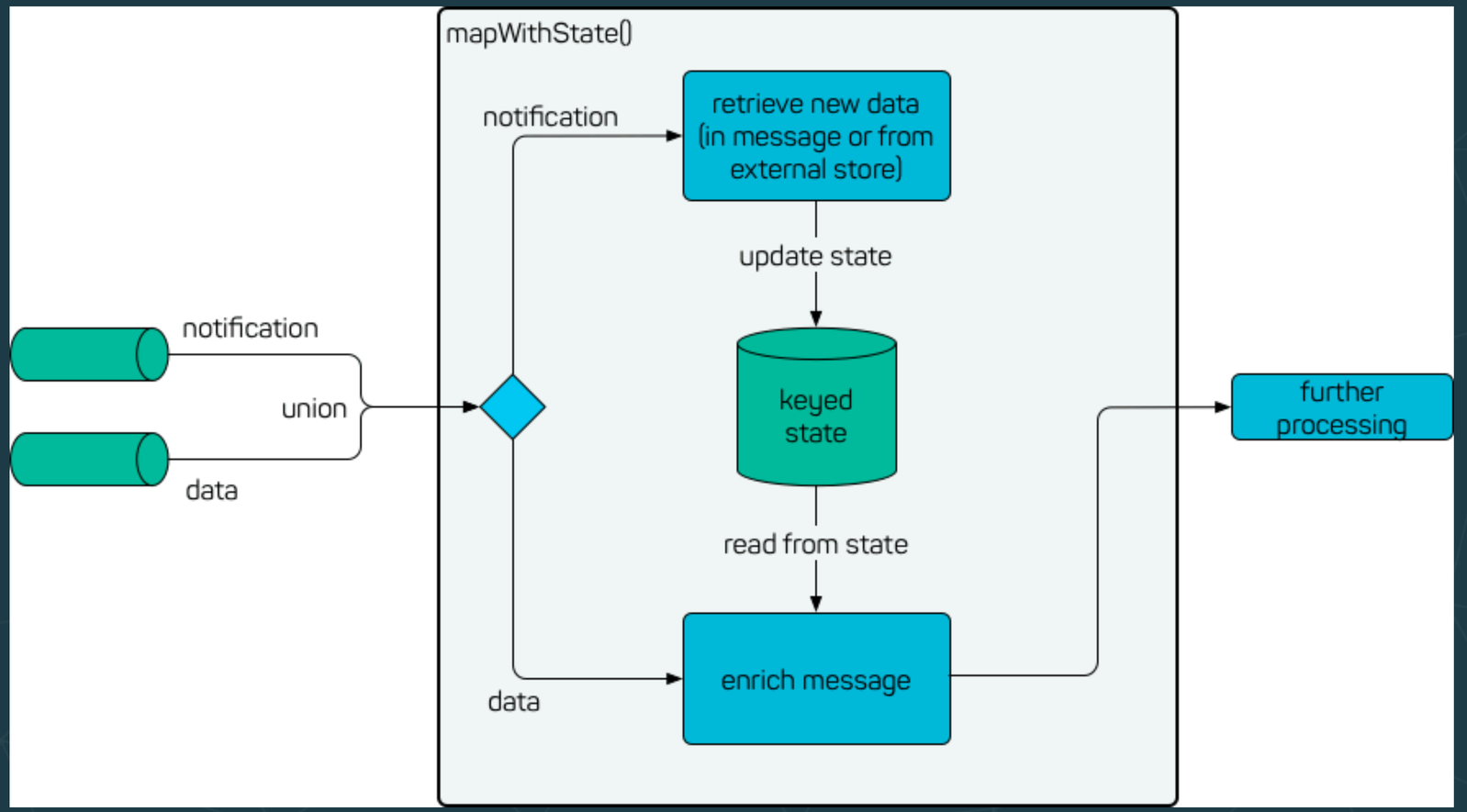
spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Spark Structured Streaming - State



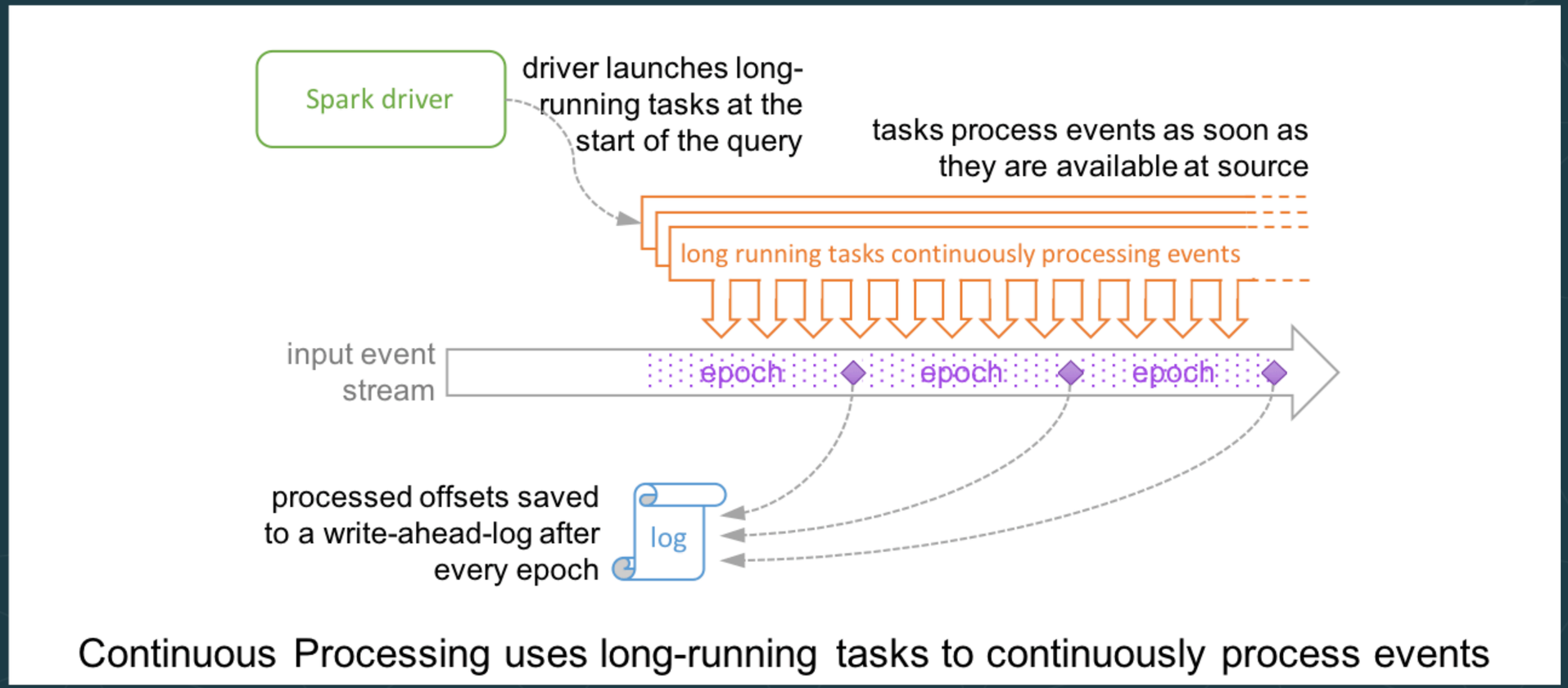
https://databricks.com/blog/2017/10/17/arbitrary-stateful-processing-in-apache-sparks-structured-streaming.html

Spark Structured Streaming - mapWithState



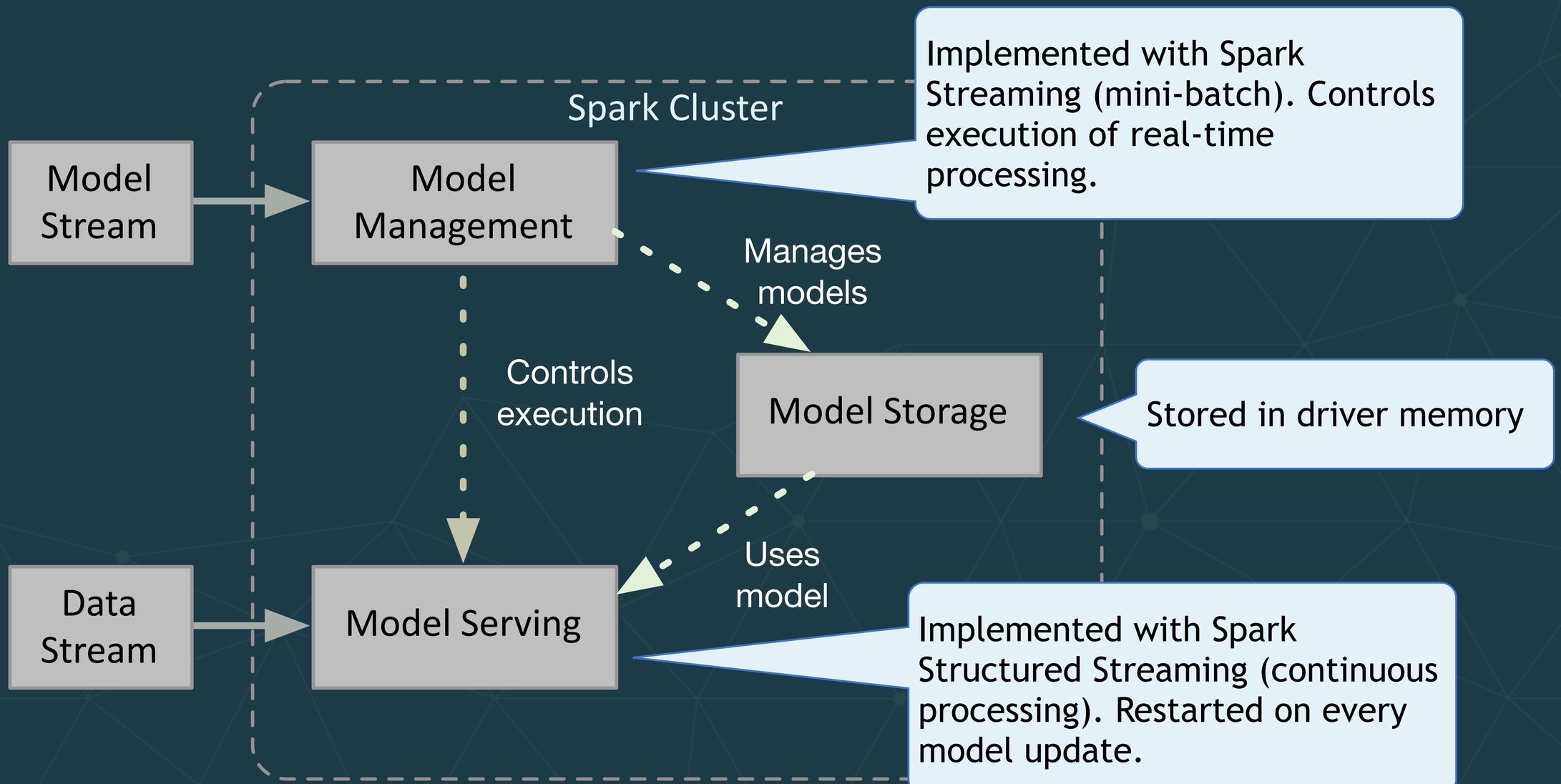
https://blog.codecentric.de/en/2017/07/lookup-additional-data-in-spark-streaming/

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 the *sparkServer* project
   a. *SparkStructuredModelServer* uses *mapWithState*.
   b. *SparkStructuredStateModelServer* implements the "multi-loop" approach

Comparing Implementations

1. *Akka Streams* with Akka is a *library* providing great flexibility for implementations and deployments, but requires custom code for scaling and failover.
   - More flexibility, but more responsibility (i.e., do it yourself)
2. *Flink* and *Spark Streaming* are stream-processing *engines* (SPE) that automatically leverage cluster resources for scaling and failover. Computations are a set of operators and they handle execution parallelism for you, using different threads or different machines.
   - Less flexibility, but less responsibility

Spark vs Flink

1. *Flink*: iterations are executed as cyclic data flows; a program (with 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. *Spark*: each iteration is a new set of tasks/operators scheduled and executed. Each iteration operates on the result of the previous iteration which is held in memory. For each new execution cycle, the results have to be moved to the new execution processes.

https://hal.inria.fr/hal-01347638v2/document

Spark vs Flink

1. In *Flink*, all additional data is kept locally, so arbitrarily complex structures can be used for its storage (although serializers are required for checkpointing). The serializers are only invoked out of band.
2. In *Spark*, all additional data is stored external to each mini-batch, so it has to be marshaled/unmarshaled for every mini-batch (for *every message* in continuous execution) to make this data available.
3. *Spark Structured Streaming* is based on SQL data types; mapping between JVM and SQL data types can be complex.

Outline

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

Additional Production Concerns for Model Serving

- Production implications for *models as data*
- Software process concerns, e.g., CI/CD
- Another pattern: *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
  - …

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?
  - E.g., these papers and articles on privacy preservation

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**

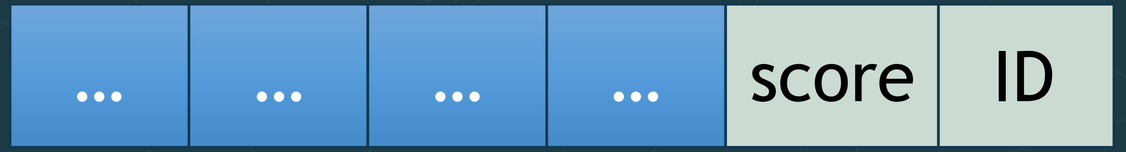
Model Traceability Requirements

- A model repository
- Information stored for each model instance, e.g.:
  - Version ID
  - Name, description, etc.
  - Creation, deployment, and retirement dates
  - Model parameters
  - Quality metrics
  - …

Model Traceability in Use

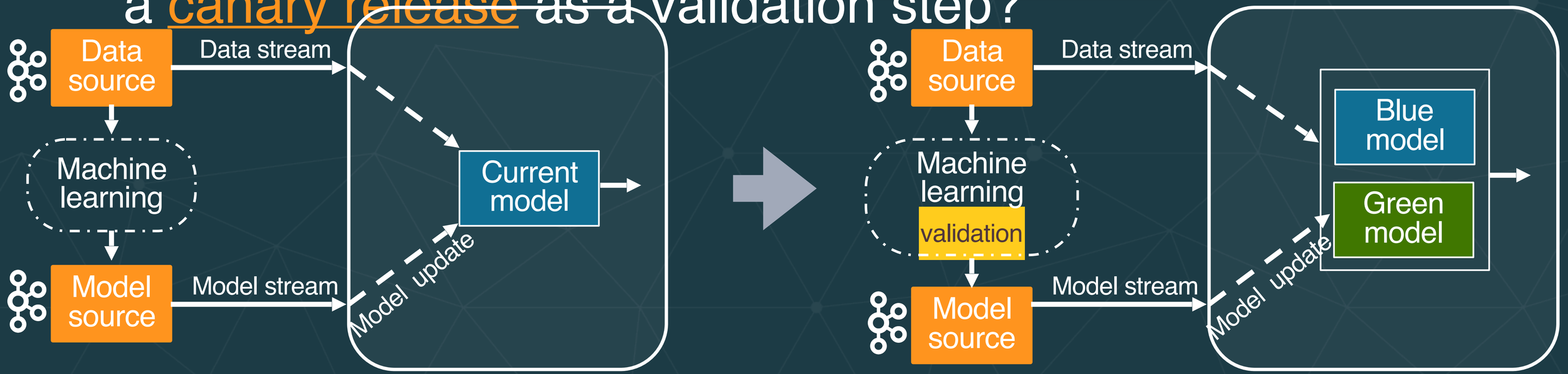
- You also need to augment the records with the model version ID, as well as the score.
  - Input record

  - Output record with score, model version ID



Software Process

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



Speculative Execution

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.

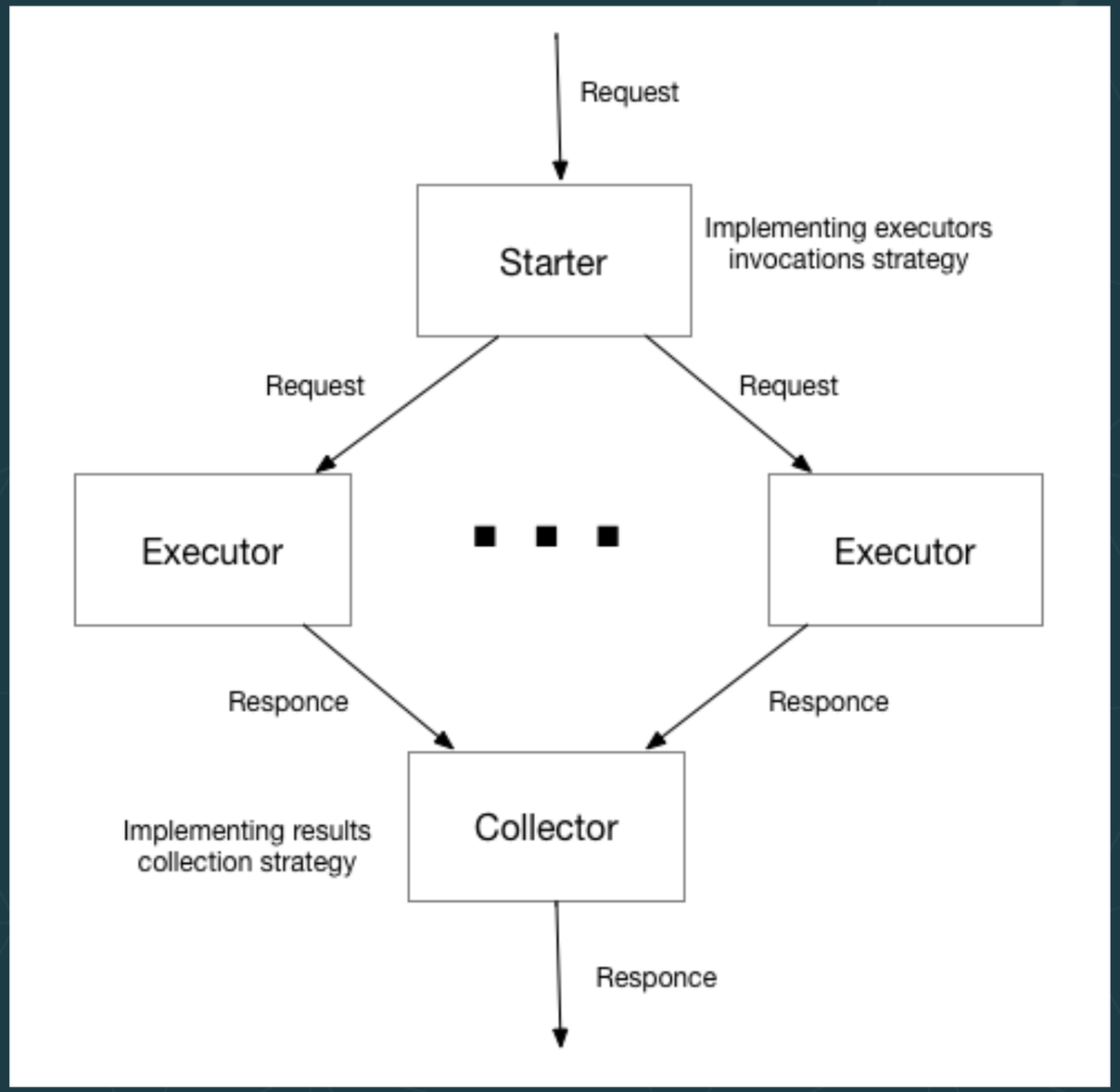
Speculative Execution

- Provides more concurrency if extra resources are available.
- Examples uses:
  - 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

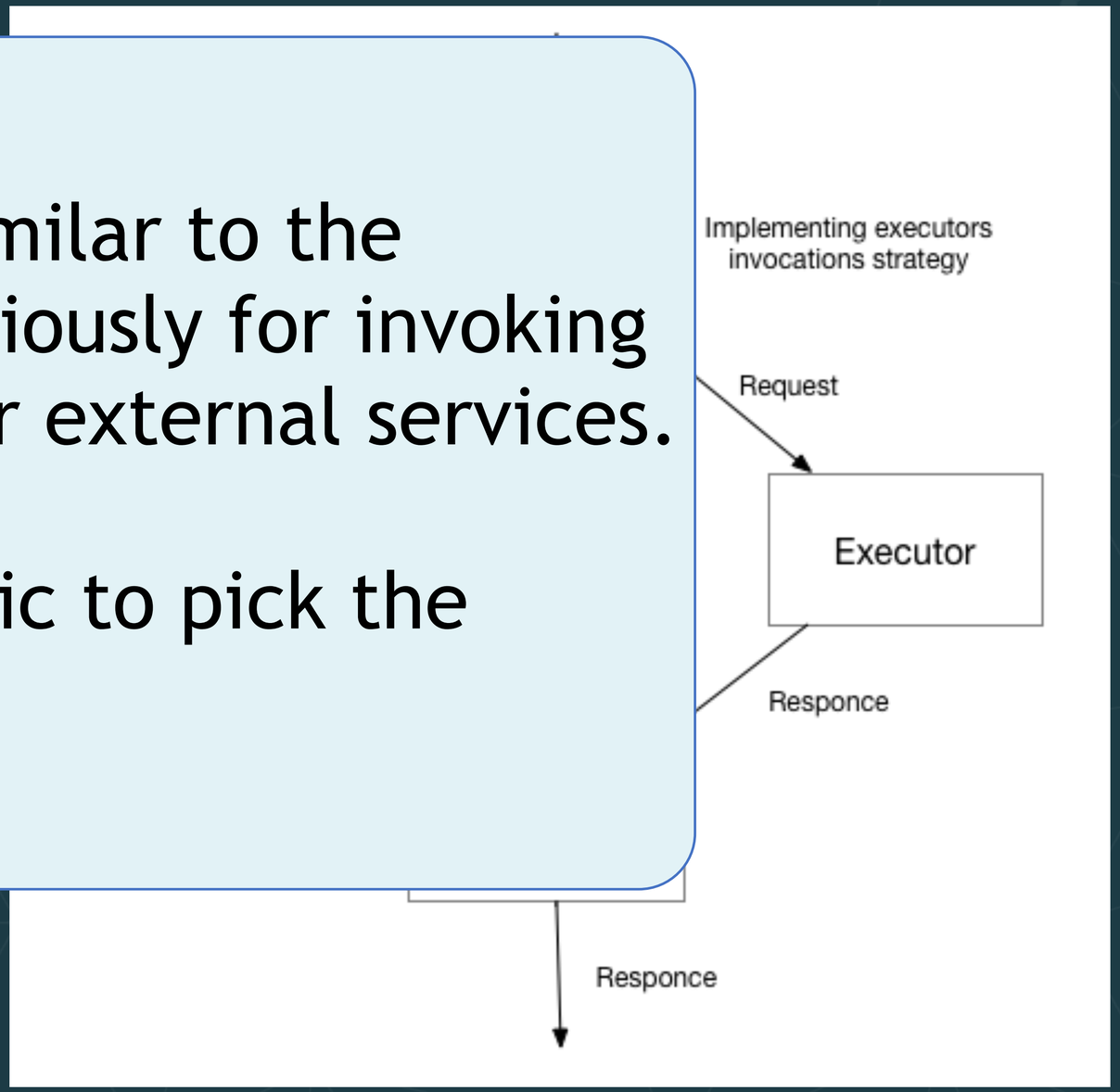


General Architecture for Speculative Execution

- Starter (p
parallelis
strategy
- Parallel e
executors
- Collector
bringing
executors

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.



Use Case - Guaranteed Execution Time

- I.e., meet a tight 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 smarter result
    - (Is it clear why *T2 < T < T1* is required?)

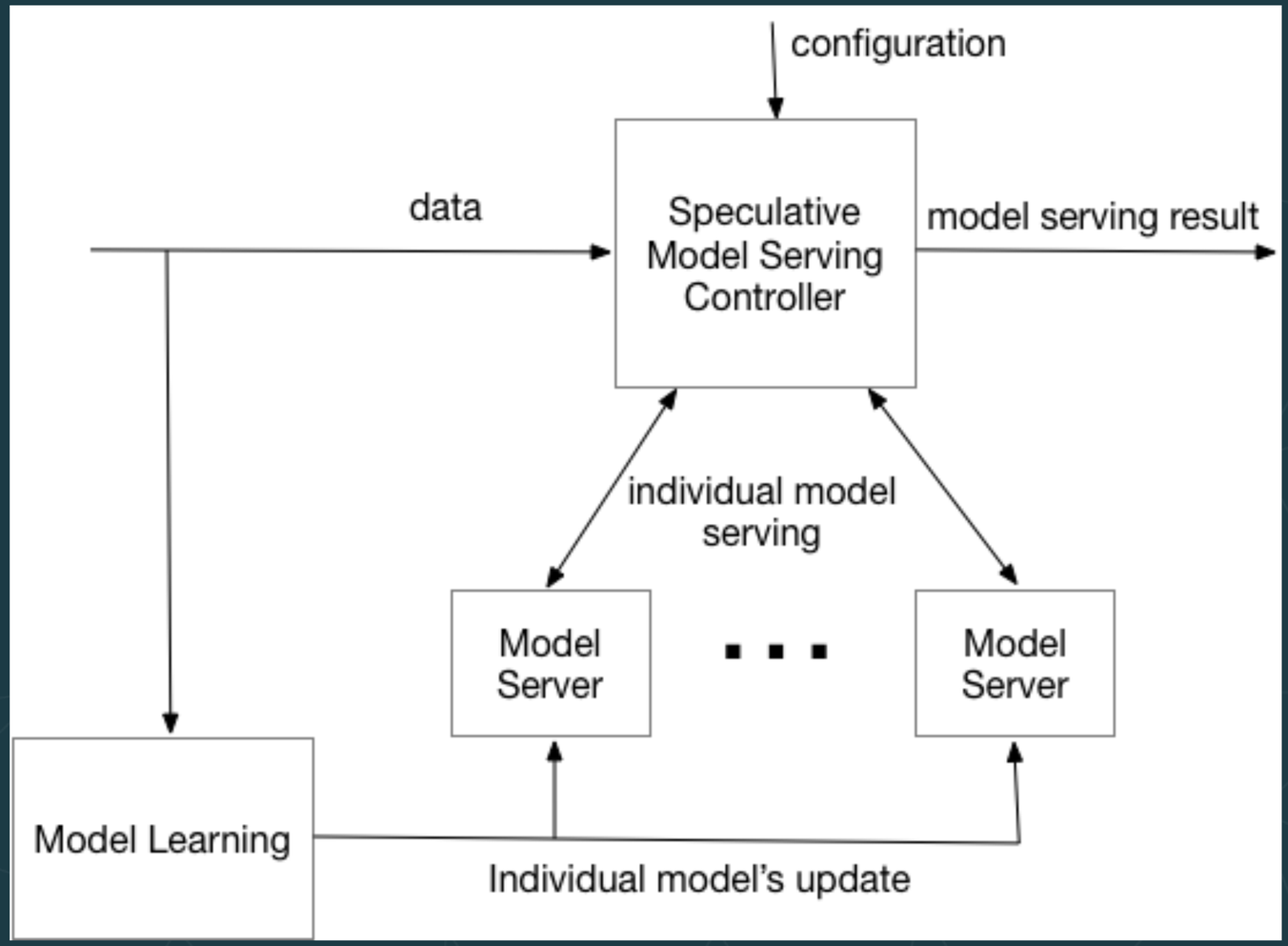
Use Case - Ensembles of Models

- Consensus-based model serving
  - *N* models (*N* is *odd*)
  - Score with all of them and return the *majority* result

- Quality-based model serving
  - N models using the same *quality metric*
  - Pick the result for a given record with the best quality score

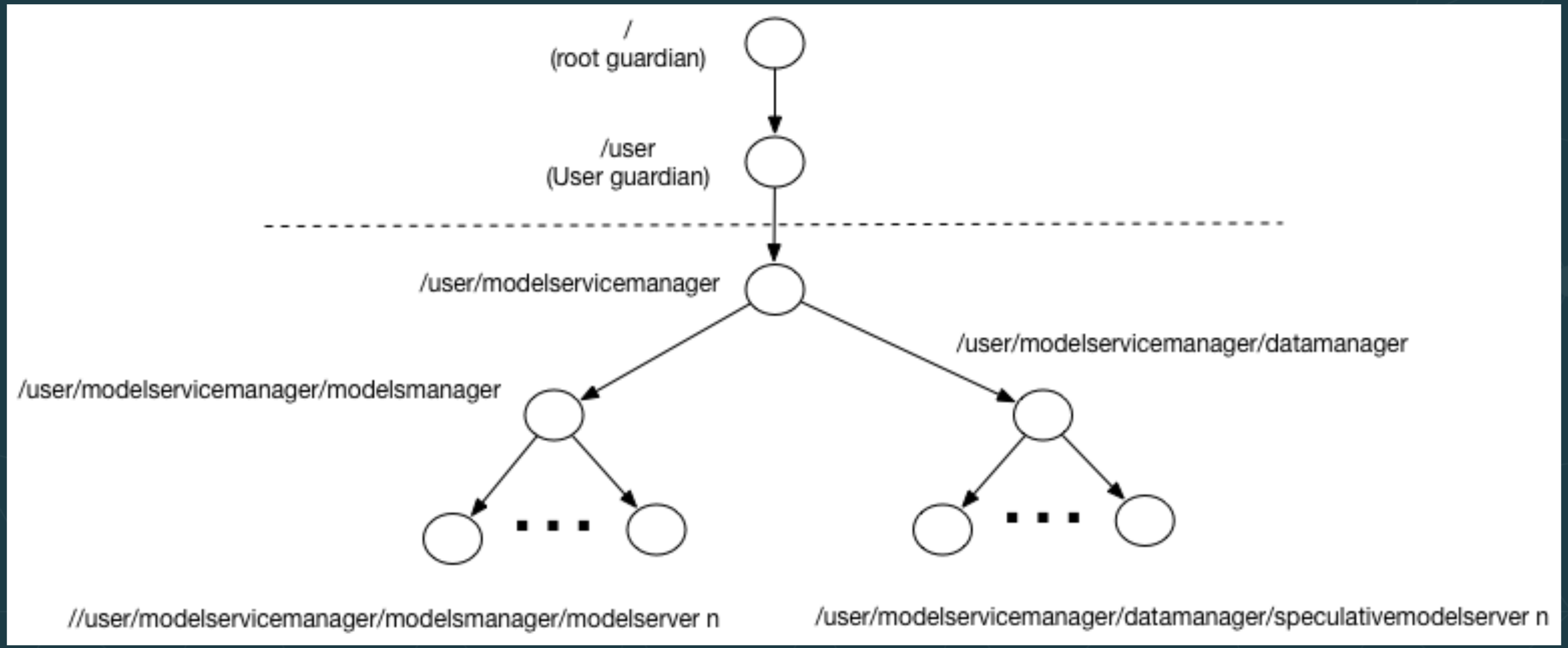
- Similarly for more sophisticated boosting and bagging systems

Architecture



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

One Design Using Actors



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

Outline

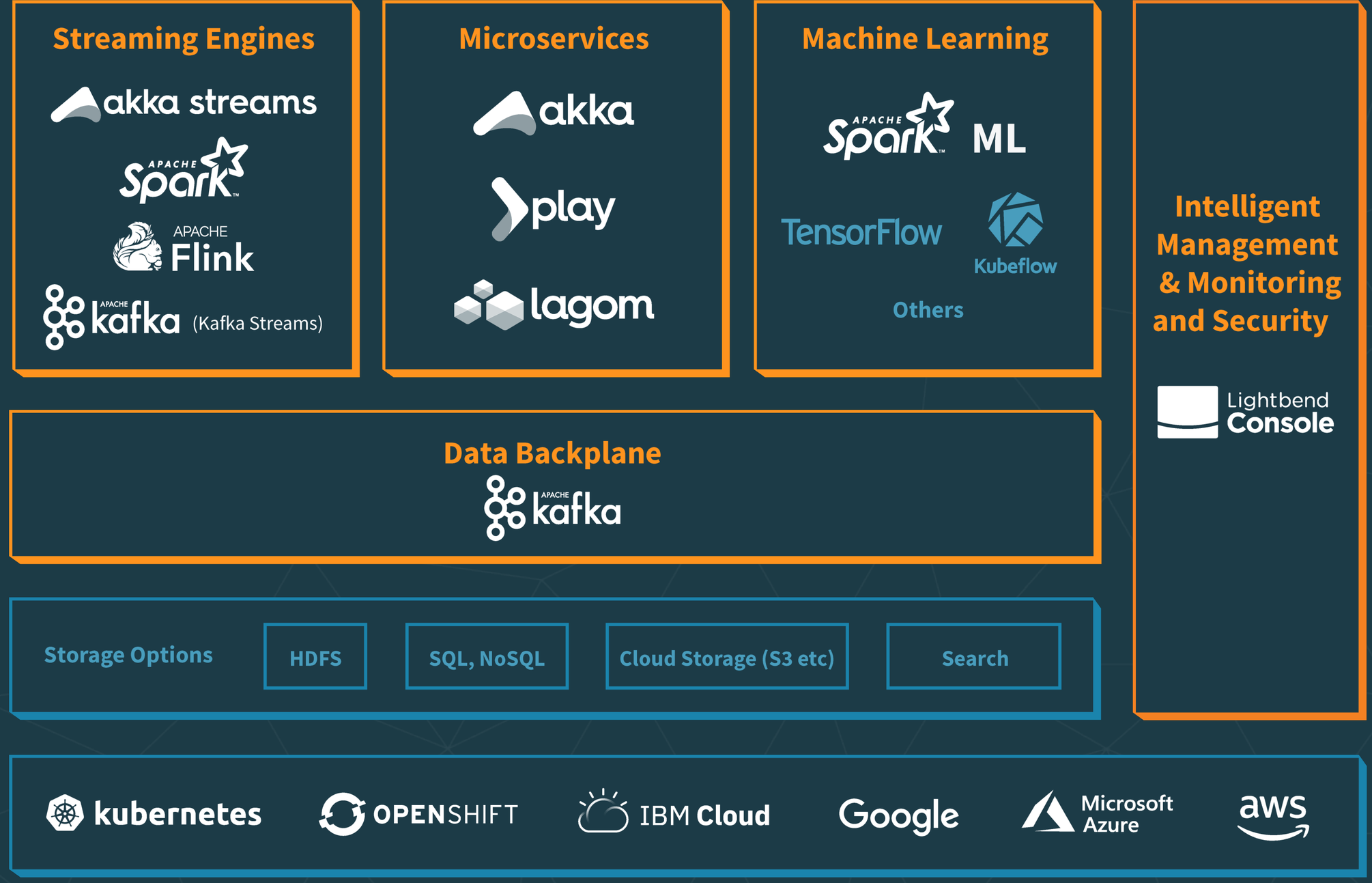
- Hidden technical debt in machine learning systems
- Model serving patterns
  - Embedding models as code
  - Models as data
  - Model serving as a service
  - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

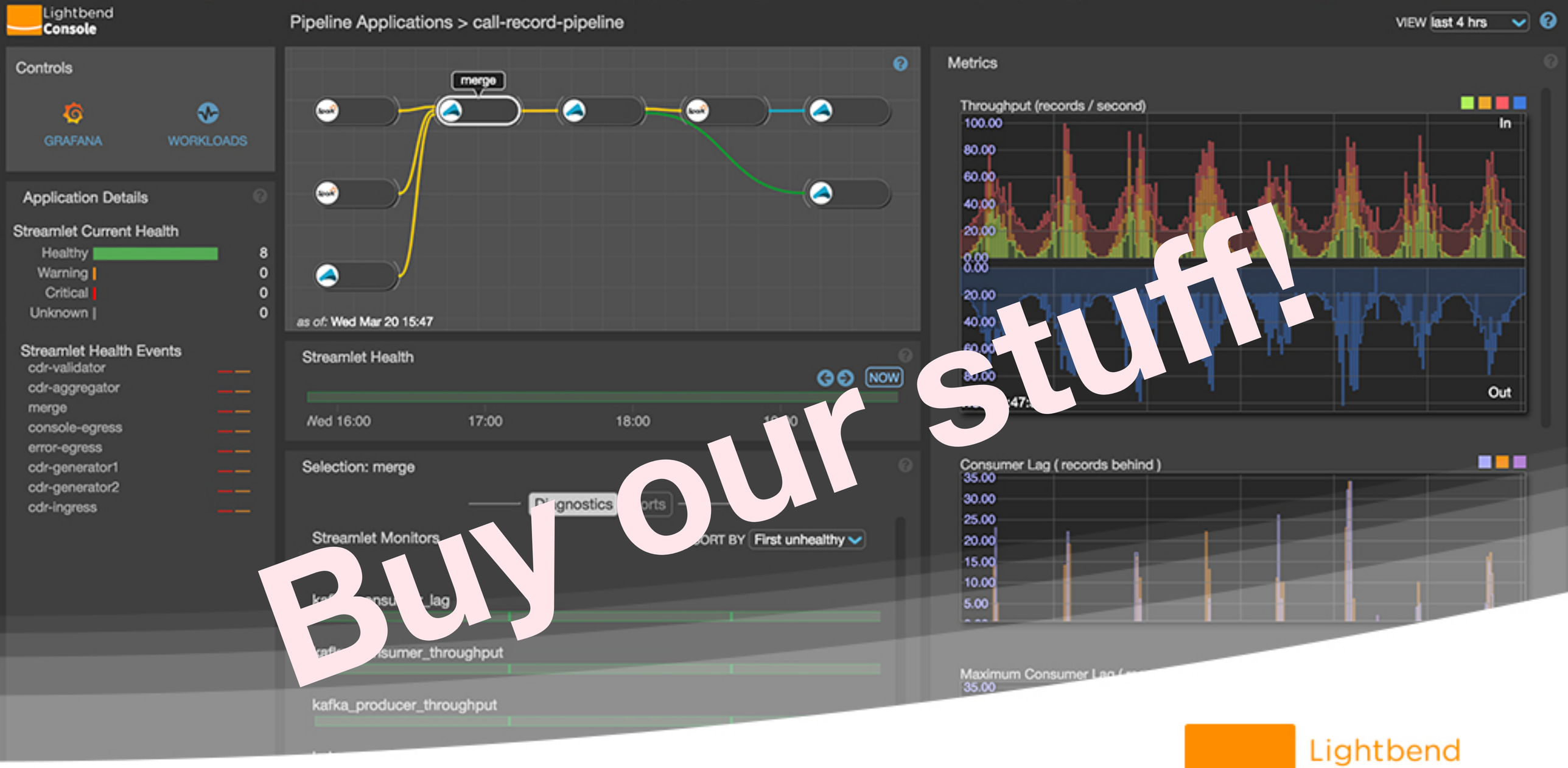
Recap (1/2)

- 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 as a service* provides the most convenience

Recap (2/2)

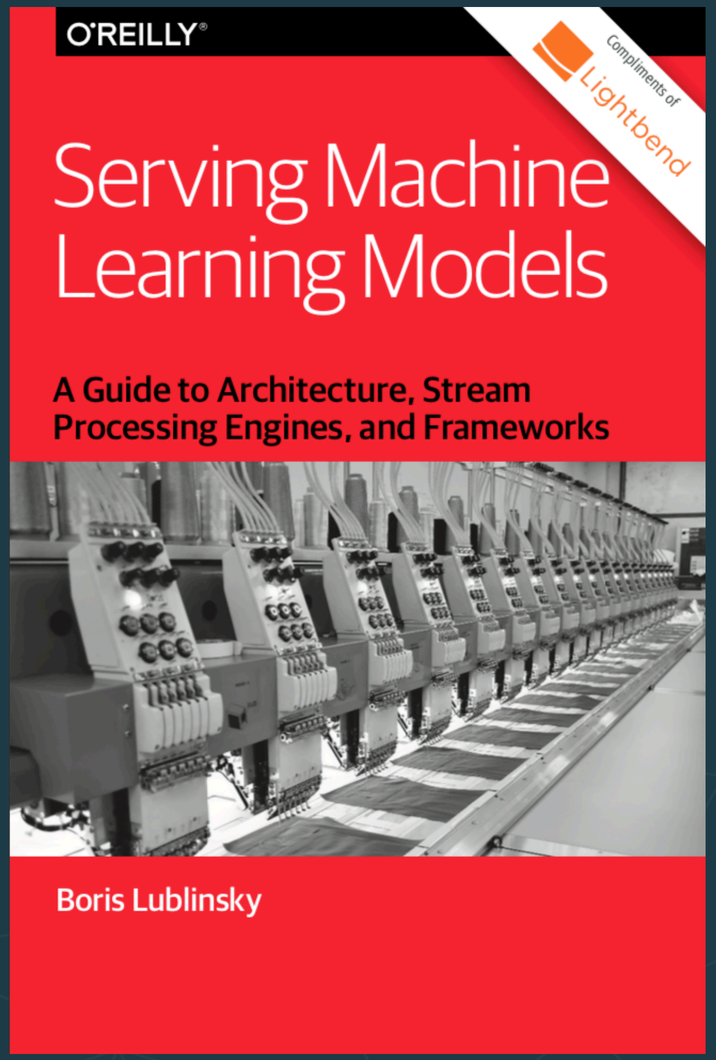
- Model serving can be implemented:
  - in “general” microservices with *libraries* like *Akka Streams*,
  - or with *engines*, like *Flink* and *Spark*
- Model serving can be *in-process* (embedded library) or an *external service* (e.g., TensorFlow Serving)
- Production concerns include integration with your *CI/CD pipeline* and *data governance*


Streaming Engines
akka streams
Spark
Flink
kafka (Kafka Streams)
Microservices
akka
play
lagom
Machine Learning
Spark ML
TensorFlow
Kubeflow
Others
Intelligent Management & Monitoring and Security
Lightbend Console
Data Backplane
kafka
Storage Options
HDFS
SQL, NoSQL
Cloud Storage (S3 etc)
Search
kubernetes
OPENSHIFT
IBM Cloud
Google
Microsoft Azure
aws

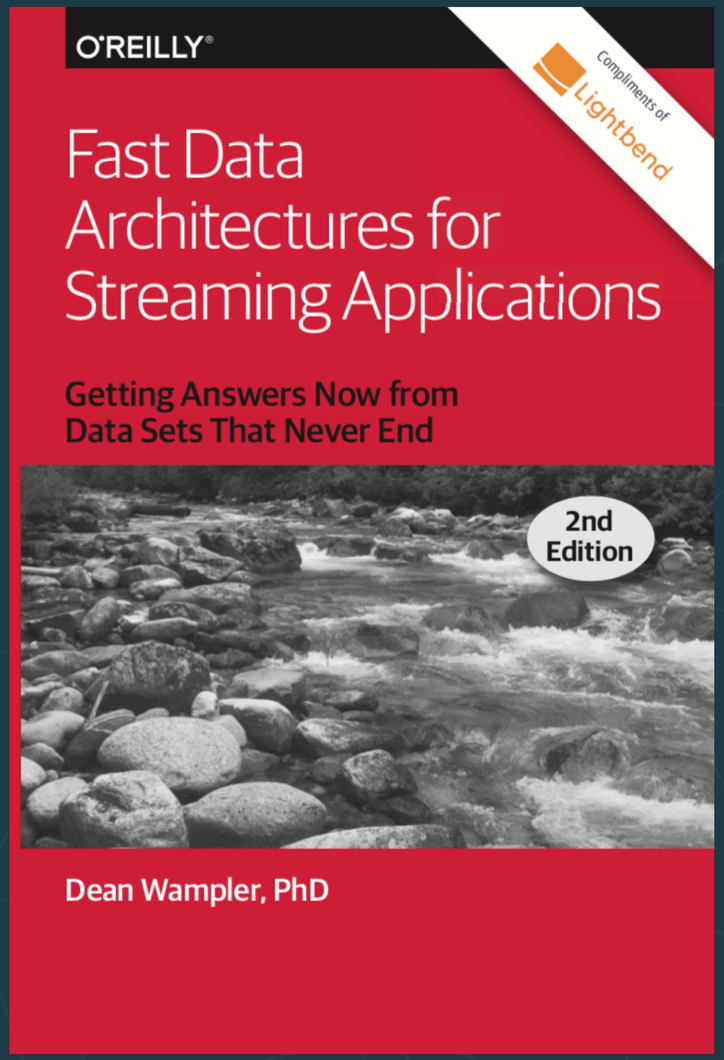


lightbend.com/lightbend-pipelines-demo

Lightbend Pipelines

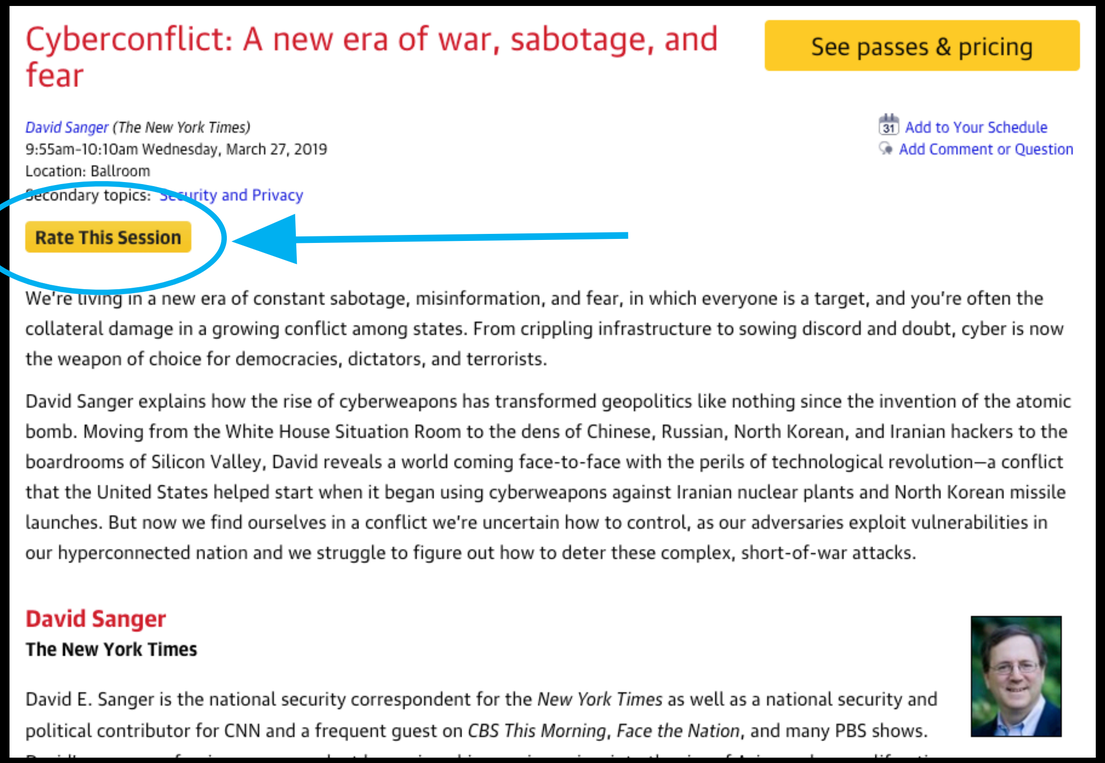


Free Download

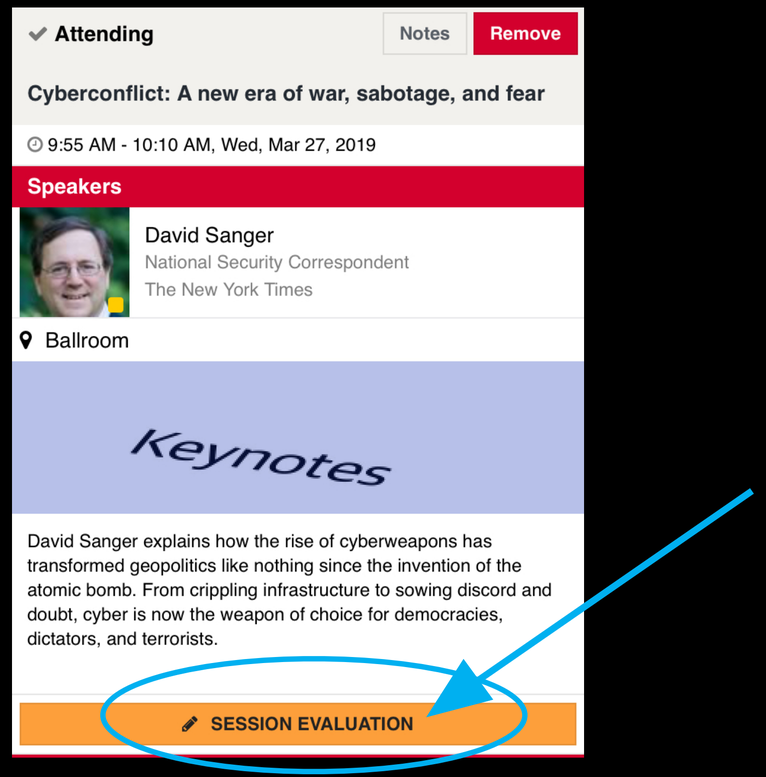


Free Download

Rate today’s session



Session page on conference website



O’Reilly Events App

Thanks for Coming!
Questions?

lightbend.com/lightbend-platform
boris.lublinsky@lightbend.com
chaoran.yu@lightbend.com

Appendix - Model Serving as a Service: Challenges

Model Serving as a Service: Challenges

- Launch ML runtime graphs, scale up/down, perform rolling updates
- Infrastructure optimization for ML
- Latency and throughput 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

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

Model Serving as a Service: Challenges (1/7)

- Launch ML runtime graphs, scale up/down, perform rolling updates
  - The classic issues for deploying any modern *microservice*.
  - What procedures does my organization use to control what's deployed to production and when?
  - How do we monitor performance?
  - When necessary, how do we scale up or down?
  - How do we perform updates across service boundaries?

Model Serving as a Service: Challenges (2/7)

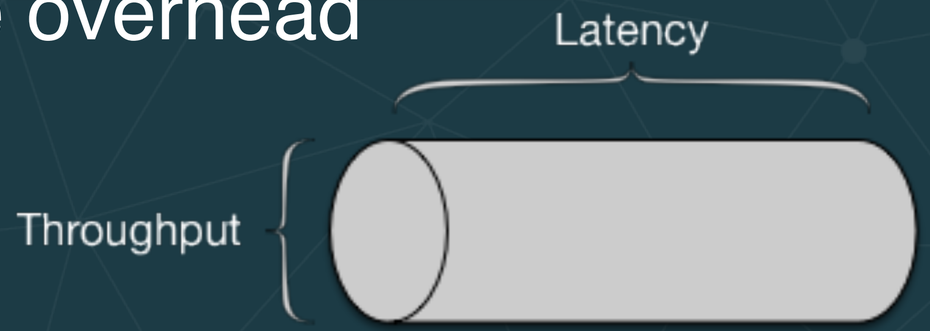
- Launch ML runtime graphs, scale up/down, perform rolling updates
  - …
  - How should updates be handled?
    - A/B testing - run two versions, measure which is best
    - Blue-Green testing - two parallel production systems, one for testing new deployments; flip to it once ready.

Model Serving as a Service: Challenges (3/7)

- Infrastructure optimization for ML
  - Closer collaboration with Data Science to:
    - feed live data for training,
    - serve up-to-date models.
  - Model training can be very compute-intensitive
  - Model serving algorithms can be expensive
  - Scoring is less deterministic than other services:
    - How do you know when it’s wrong?

Model Serving as a Service: Challenges (4/7)

- Latency and throughput optimization, and
- Connect to business apps via various APIs, e.g. REST, gRPC
  - Does your latency budget (SLA) tolerate remote RPC (e.g., HTTP(s) and gRPC)?
    - If not, see next sections!
  - Maximize throughput by:
    - bunching requests - less per/score overhead
    - call asynchronously from clients



Model Serving as a Service: Challenges (5/7)

- Allow Auditing and clear versioning
  - When was a particular model used to score a particular record?
    - E.g., “why was my loan application rejected?”
    - Which model was used? When was it deployed?
  - Regulatory compliance

Model Serving as a Service: Challenges (6/7)

- Integrate into Continuous Integration (CI)
- Allow Continuous Deployment (CD)
  - Standard and modern techniques for build, test, and deployment
  - Probably new to the Data Science teams,
    - but hopefully familiar to the Data Engineers!
  - This also helps with auditing and versioning requirements

Model Serving as a Service: Challenges (7/7)

- Provide Monitoring
  - Standard production tool for *observability*
    - What's the throughput and latency for scoring?
      - Is throughput keeping up with the data volume?
      - Is latency meeting our SLAs?
    - When will I know that a problem has arisen?
      - Alerting essential!
      - Practice fault isolation and recovery scenarios

Model Serving as a Service: Challenges

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

Some of these points are also covered in the other slides.

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