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

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

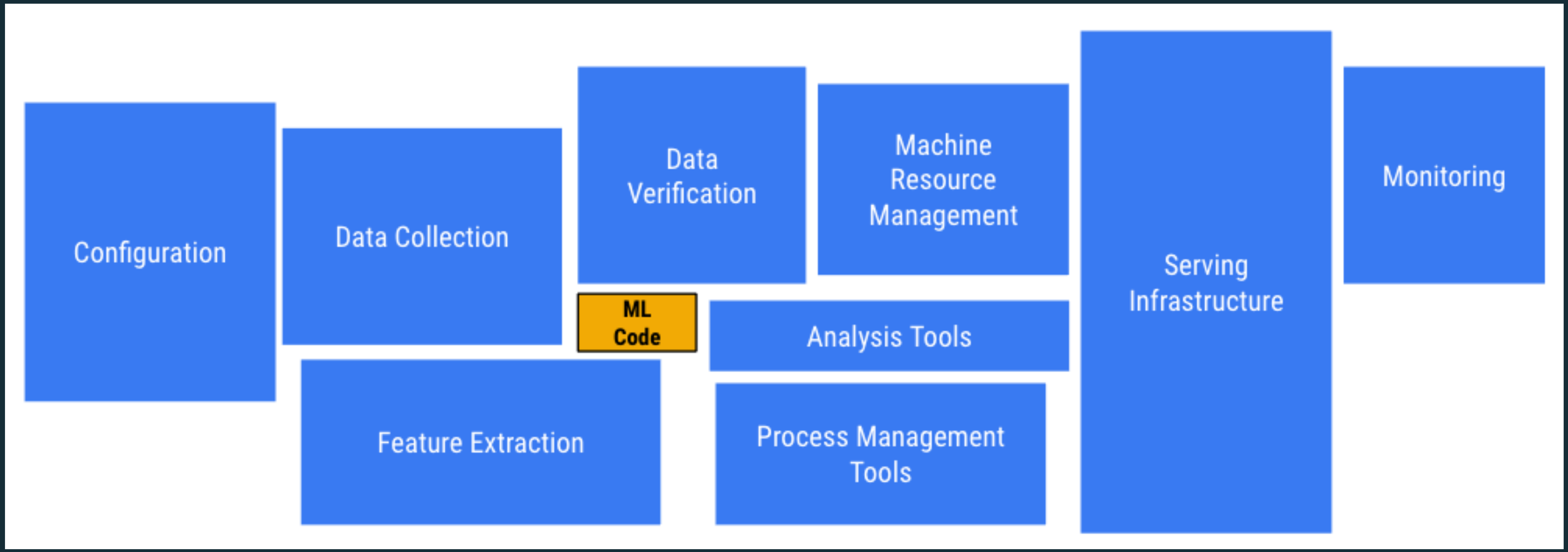
- Hidden technical debt in machine learning systems
- Model serving patterns
  - Embedding - models as code
  - Models as data
  - External services
  - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

But first, introductions…

Outline

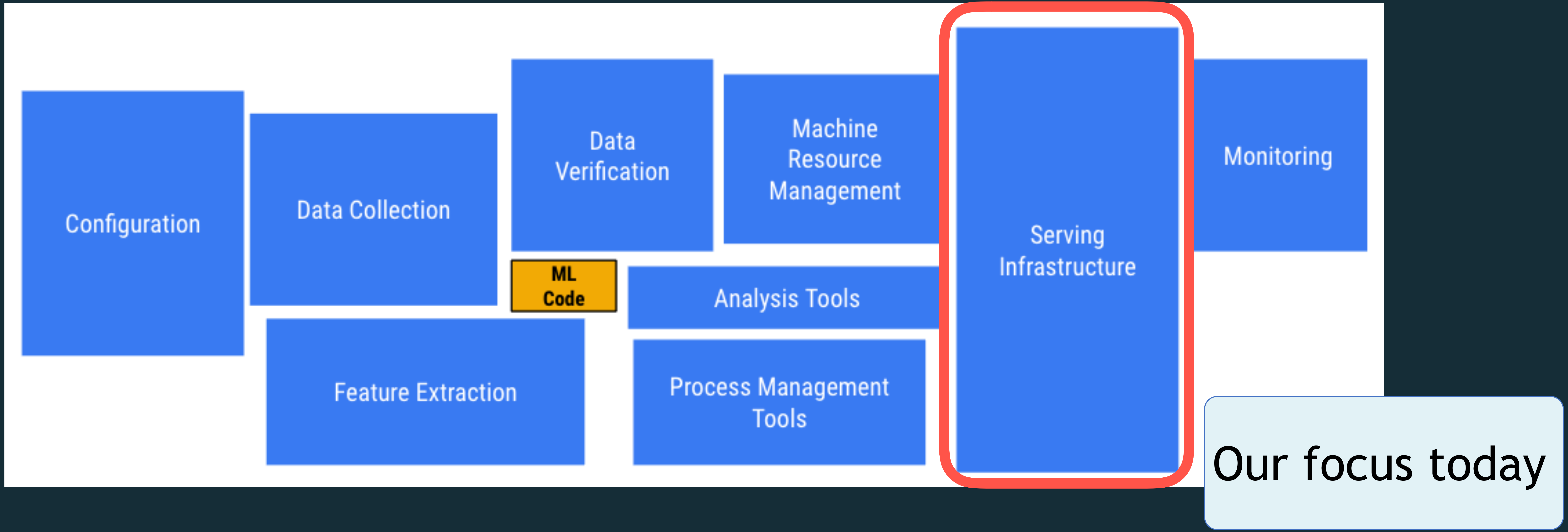
- Hidden technical debt in machine learning systems
- Model serving patterns
  - Embedding - models as code
  - Models as data
  - External services
  - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

ML vs. Infrastructure Code



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

ML vs. Infrastructure Code



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
  - 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

Embedding: Model as Code

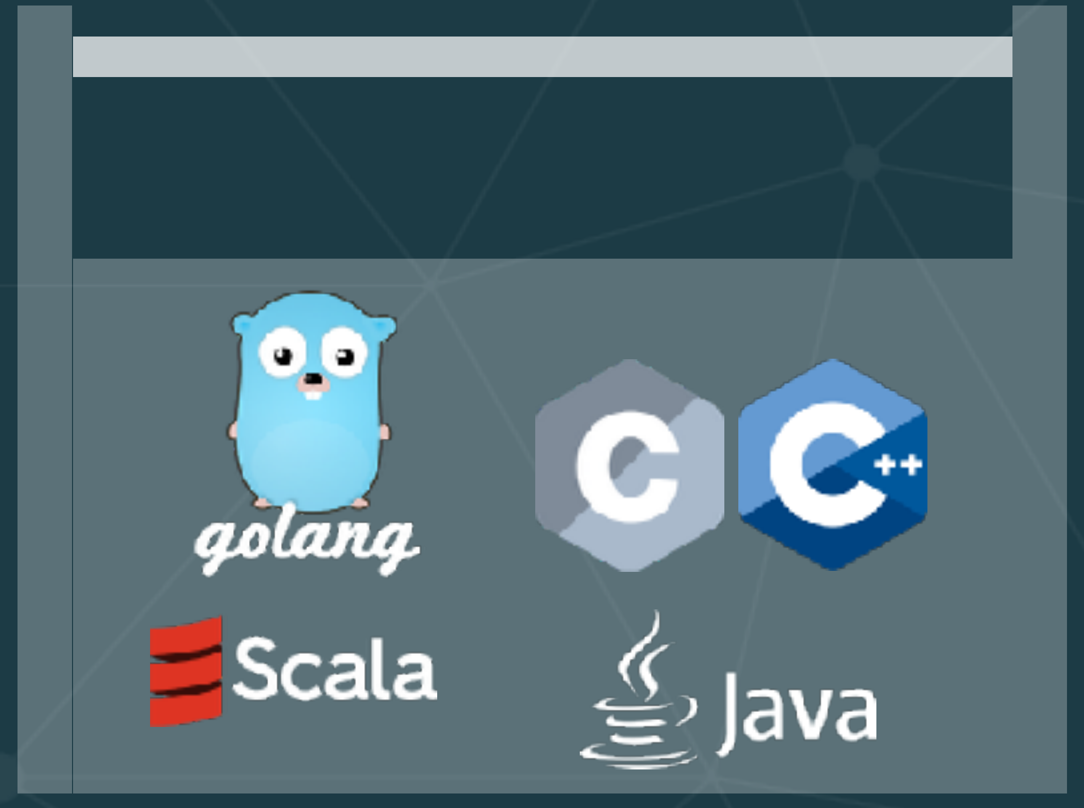
- Implement the model as source code
- The model code is linked into the streaming application at build time

Why is this problematic?

Impedance Mismatch

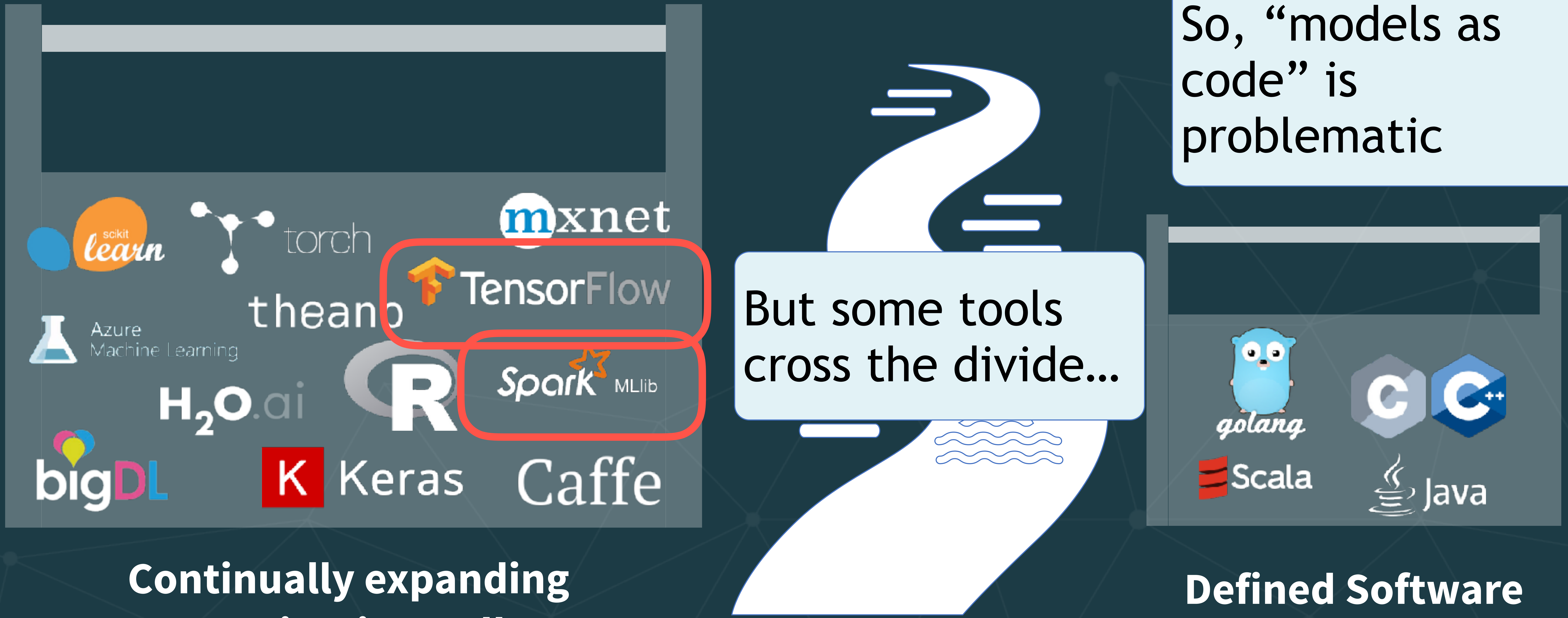
Continually expanding Data Scientist toolbox

So, “models as code” is problematic



Defined Software Engineer toolbox

Impedance Mismatch



So, “models as code” is problematic

But some tools cross the divide...

Continually expanding Data Scientist toolbox

Defined Software Engineer toolbox

Embedding: Model as Code

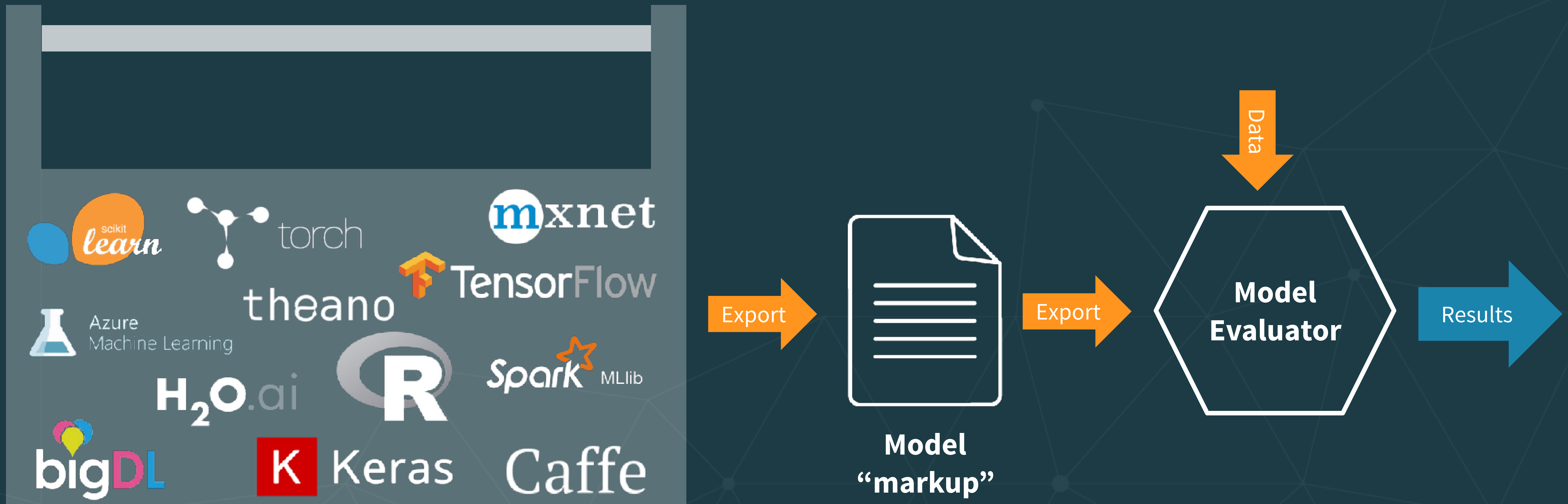
- It also *mostly* eliminates the possibility of updating the model at runtime, as the world changes*.

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

Outline

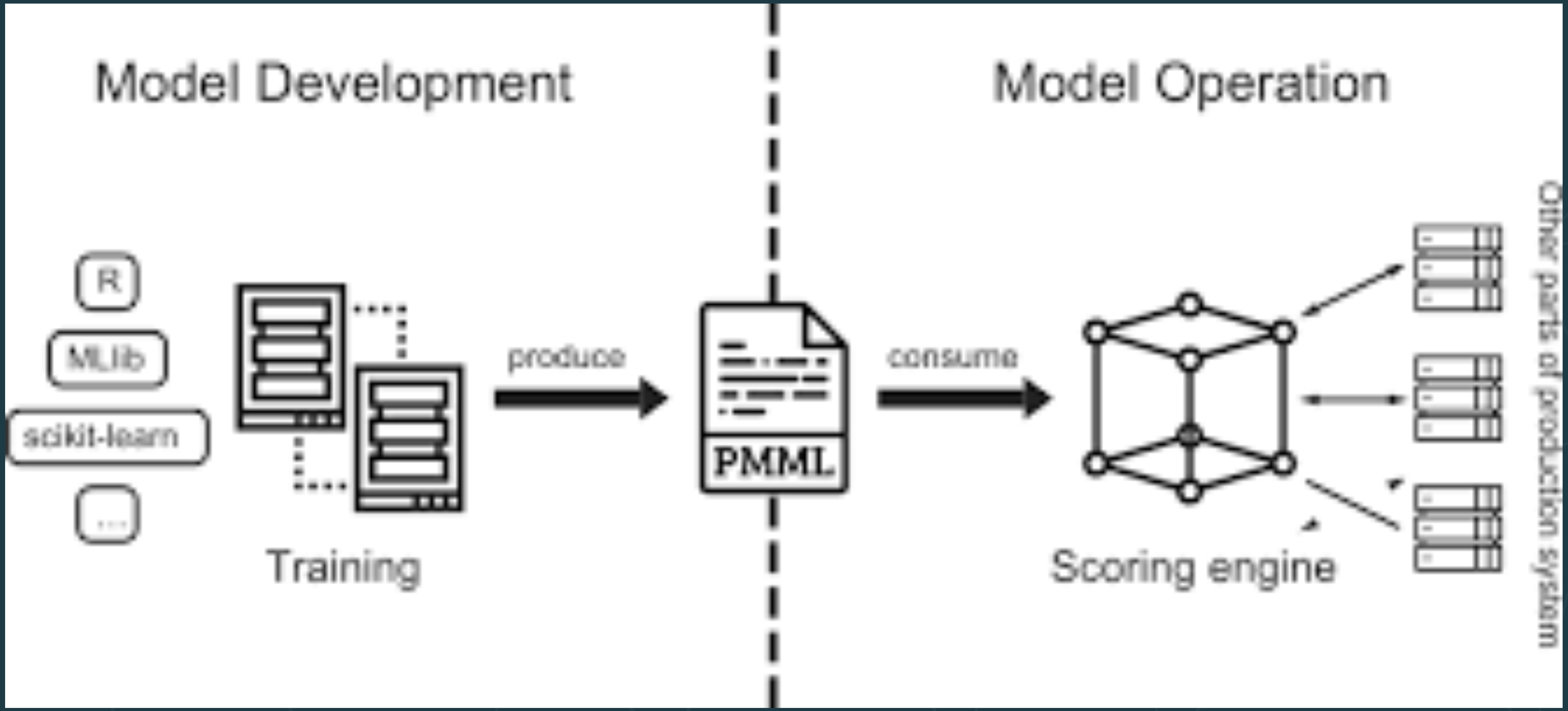
- Hidden technical debt in machine learning systems
- Model serving patterns
  - Embedding - models as code
  - Models as data
  - External services
  - Dynamically controlled streams
- Additional production concerns for model serving
- Wrap up

Better Alternative - Model As Data



Standards: PMML Predictive Model Markup Language | Portable Format for Analytics **(PFA)** | ONNX | TensorFlow

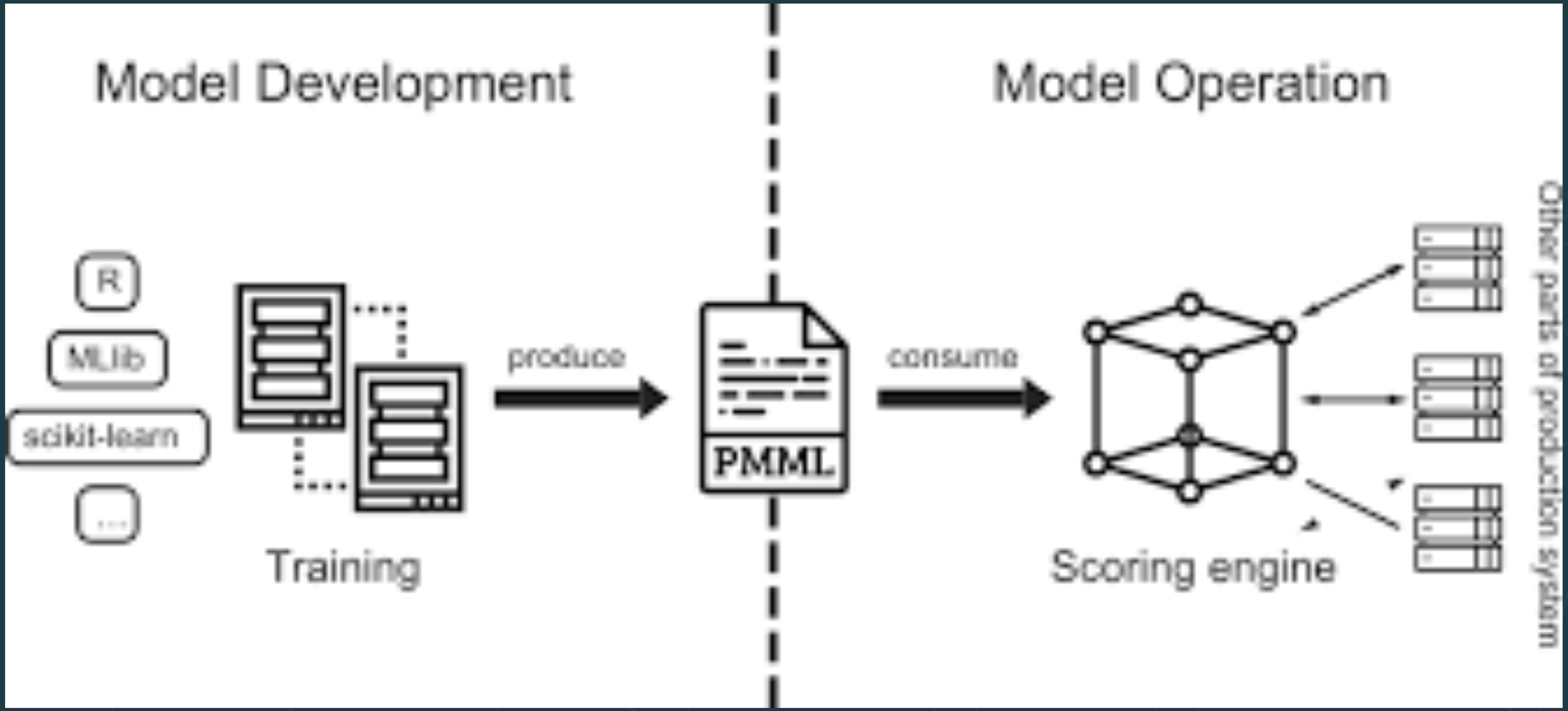
PMML



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/

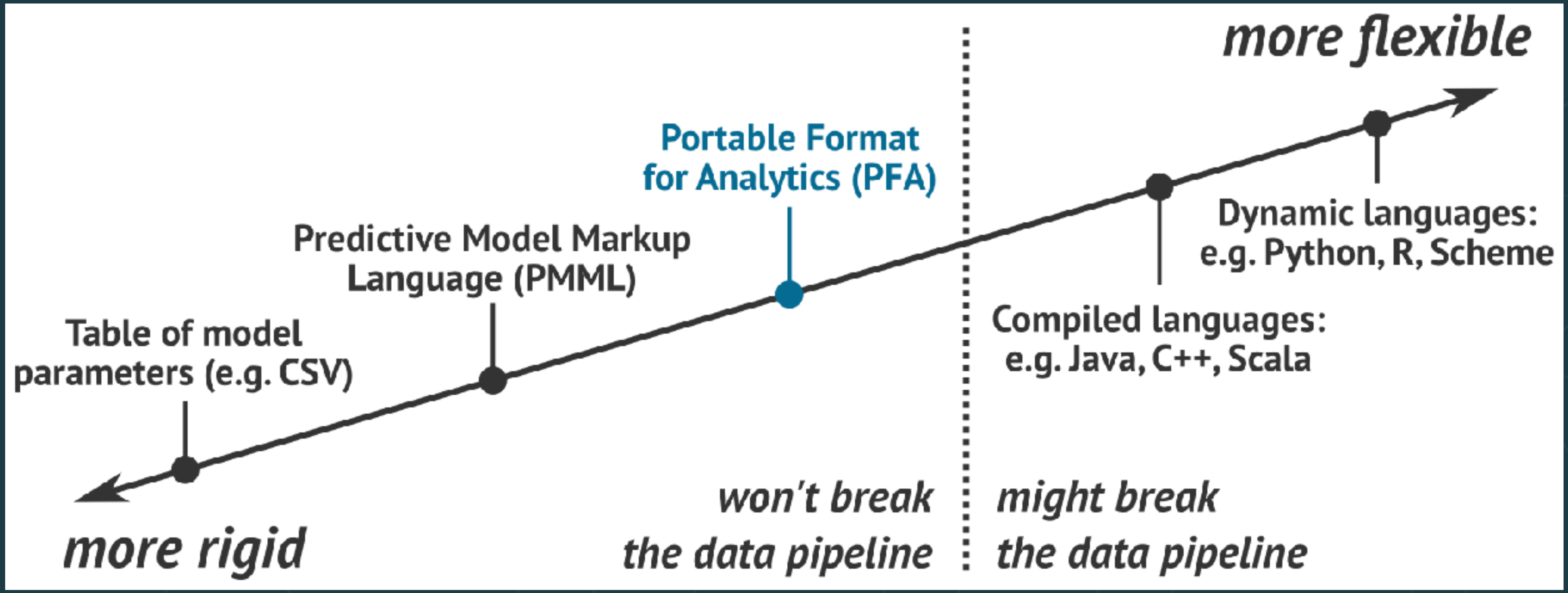
PMML



Implementations for:

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

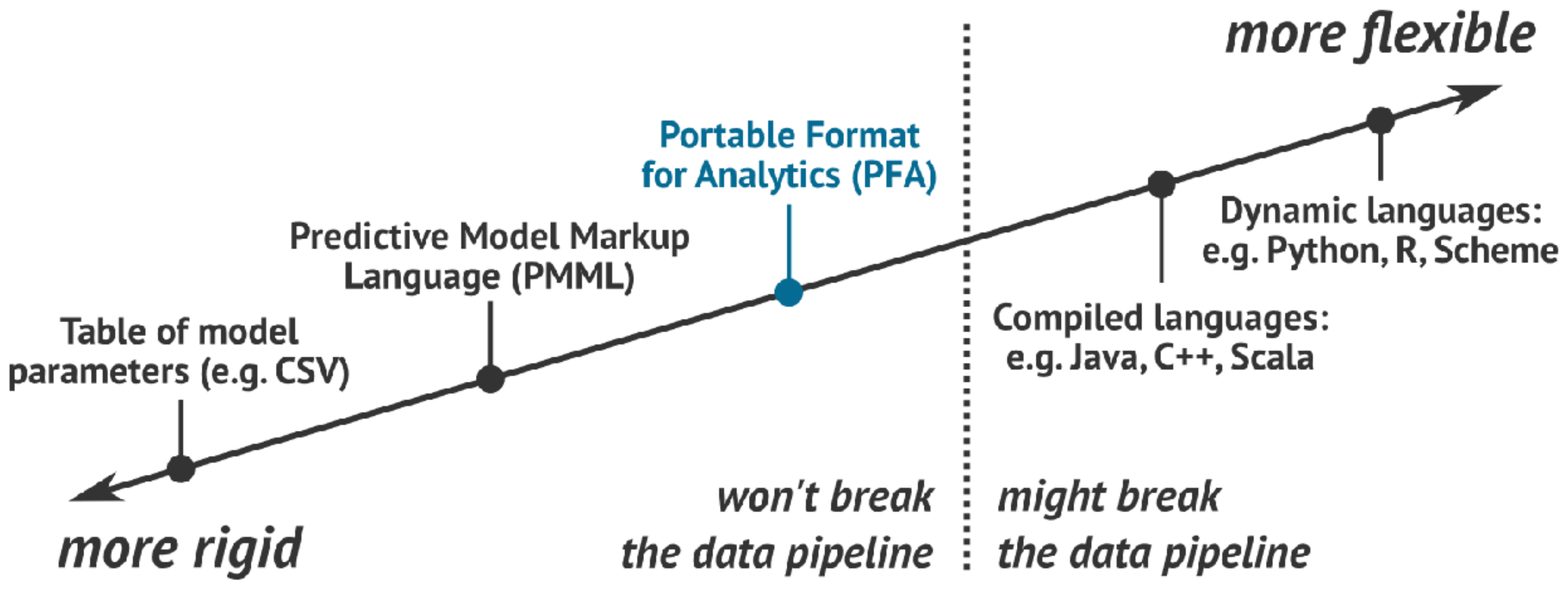
PFA



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

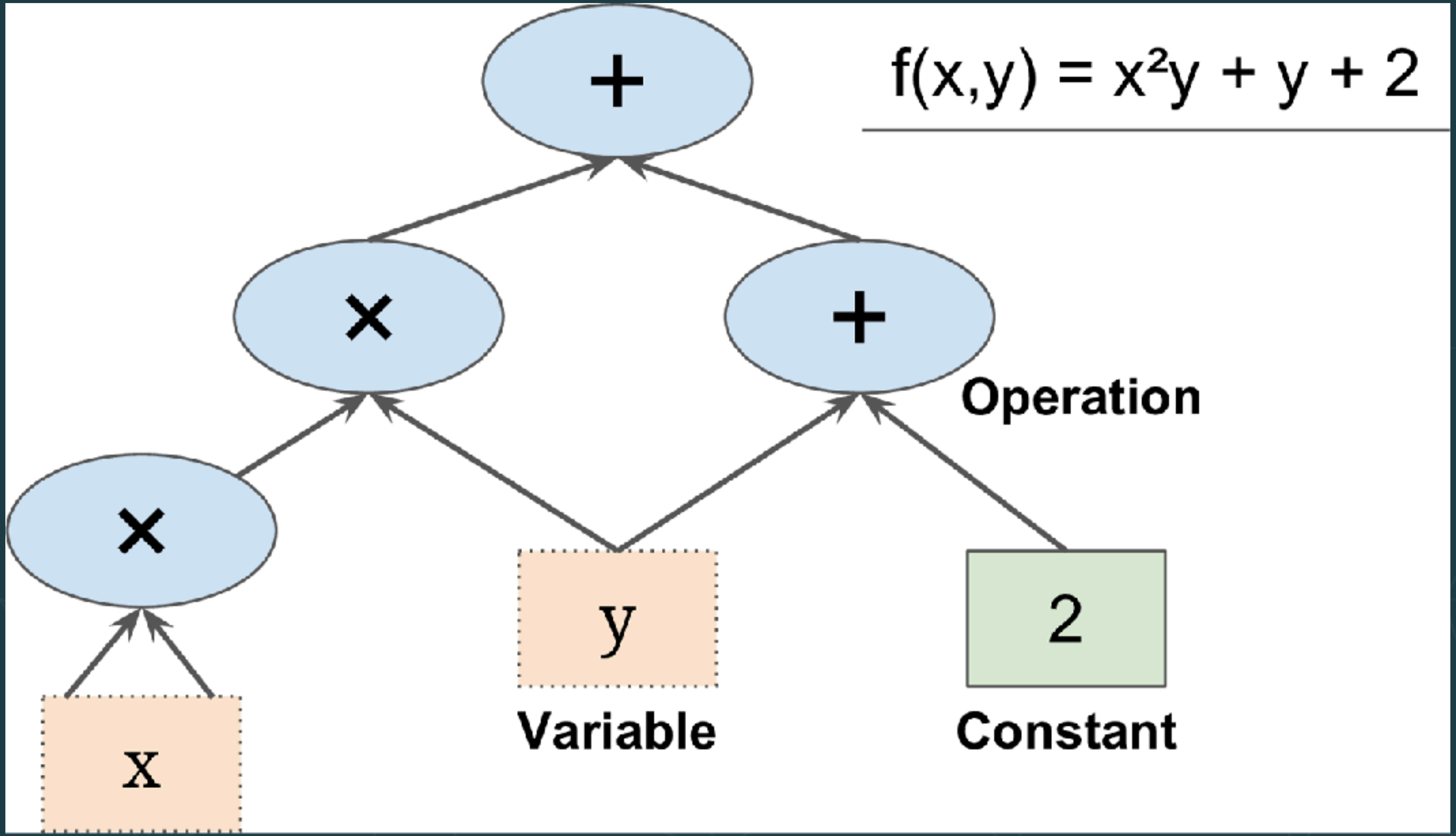
Open Neural Networks Exchange (ONNX) is an open standard format of machine learning models to offer interoperability between various AI frameworks. Led by Facebook, Microsoft, and AWS.

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

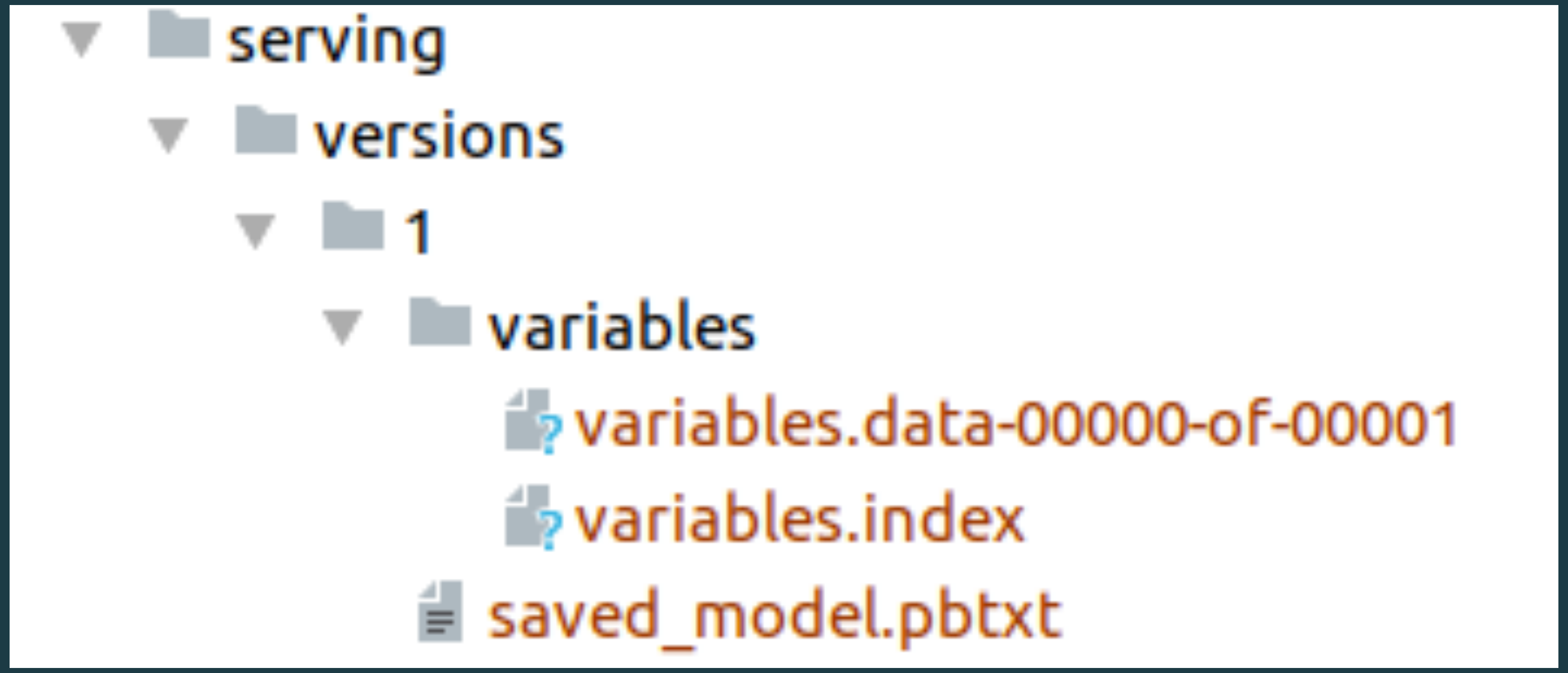
TensorFlow



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

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

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?

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

Model Serving as a Service

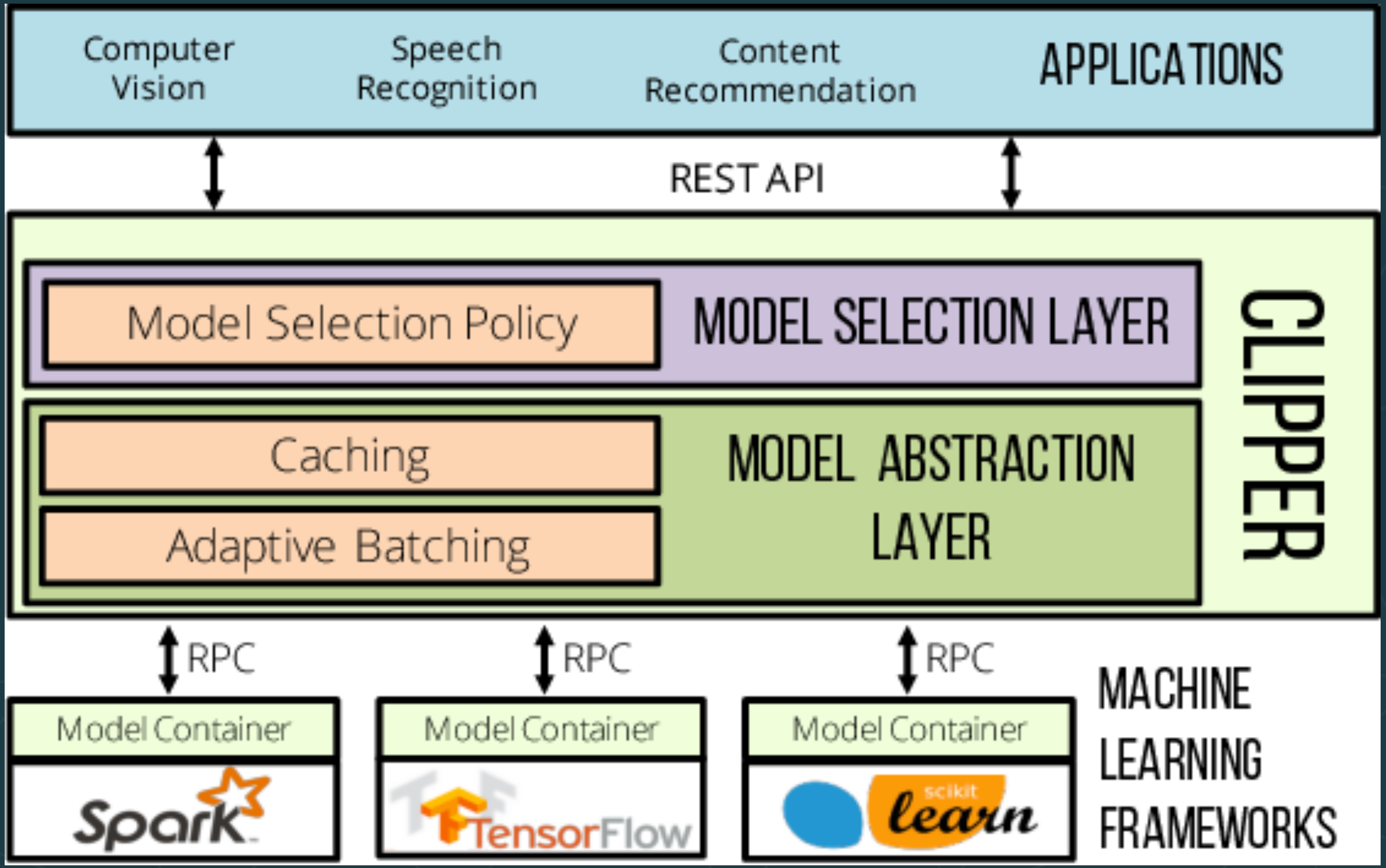
- *Advantages*
  - Simple integration with existing technologies and organizational processes
  - Easier to understand if you come from a non-streaming world
- *Disadvantages*
  - Worse latency: remote calls instead of local function calls
  - Coupling the availability, scalability, and latency/throughput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

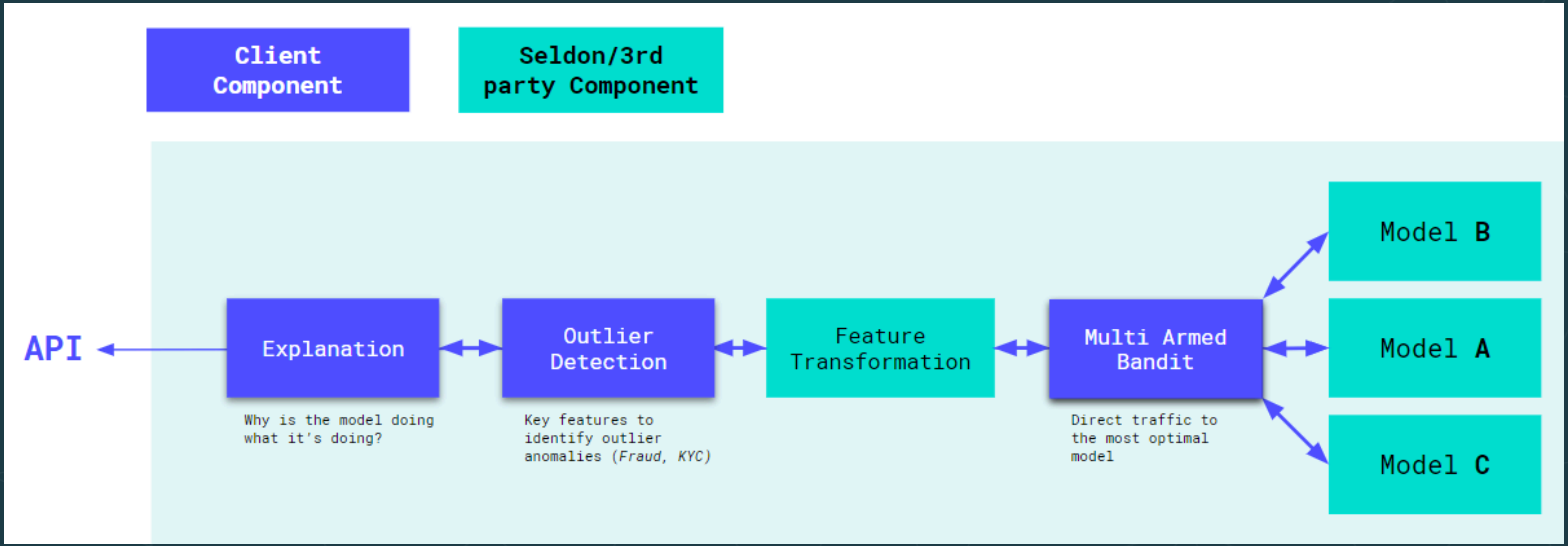
https://github.com/SeldonIO/seldon-core/blob/master/docs/challenges.md

Example: Clipper



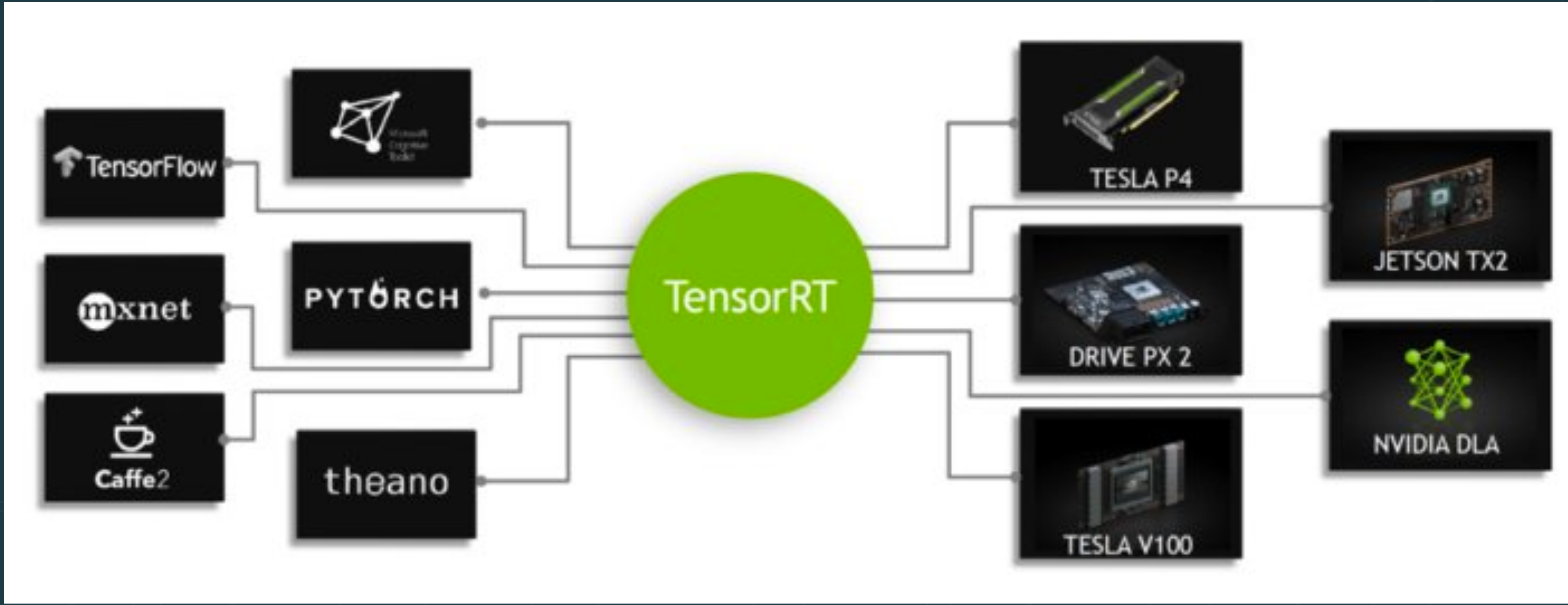
https://www.semanticscholar.org/paper/Clipper%3A-A-Low-Latency-Online-Prediction-Serving-Crankshaw-Wang/4ef862c9157ede9ff8cfbc80a612b6362dcb6e7c

Example: Seldon Core



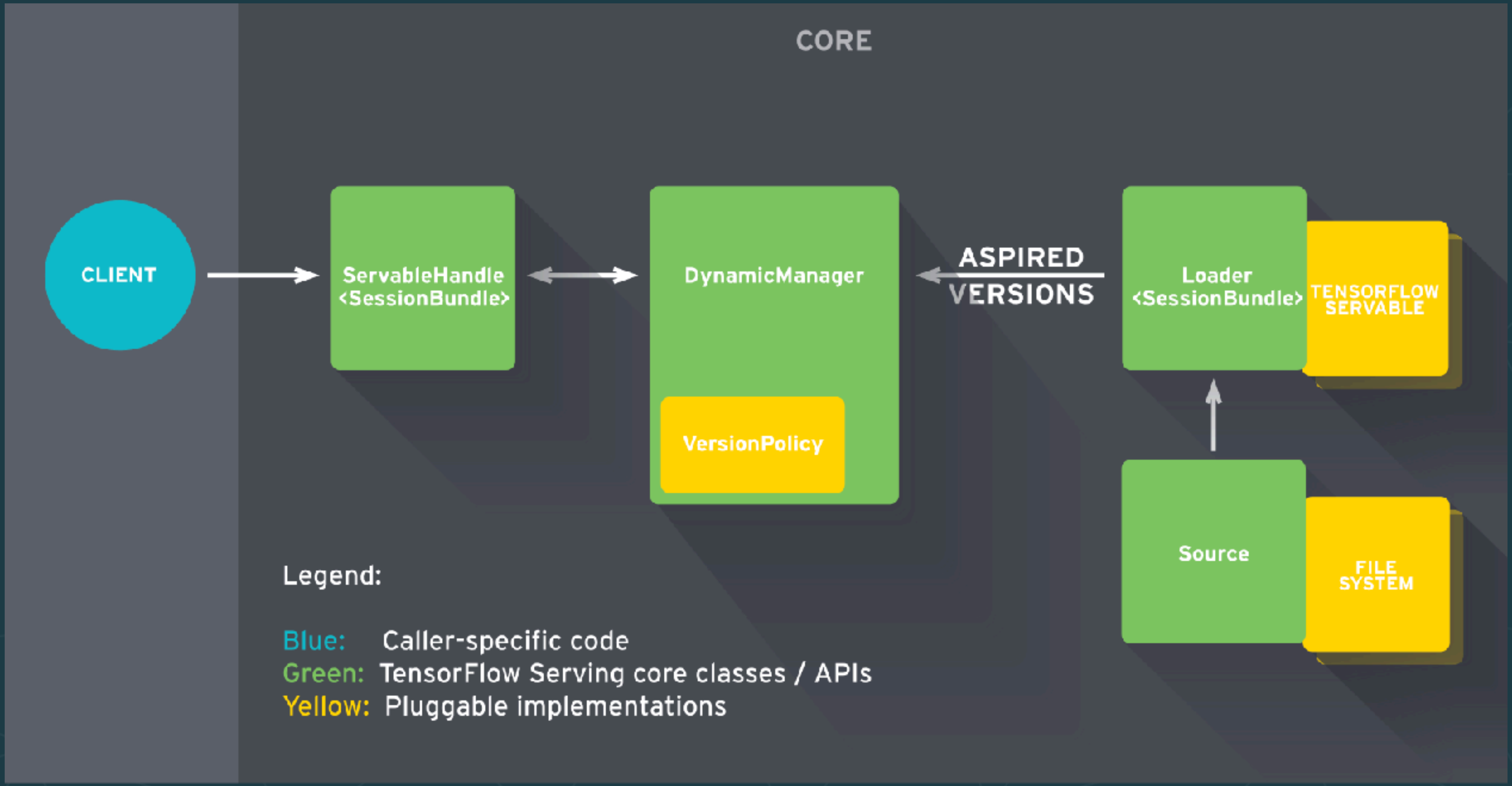
https://www.seldon.io/open-source/

Example: TensorRT



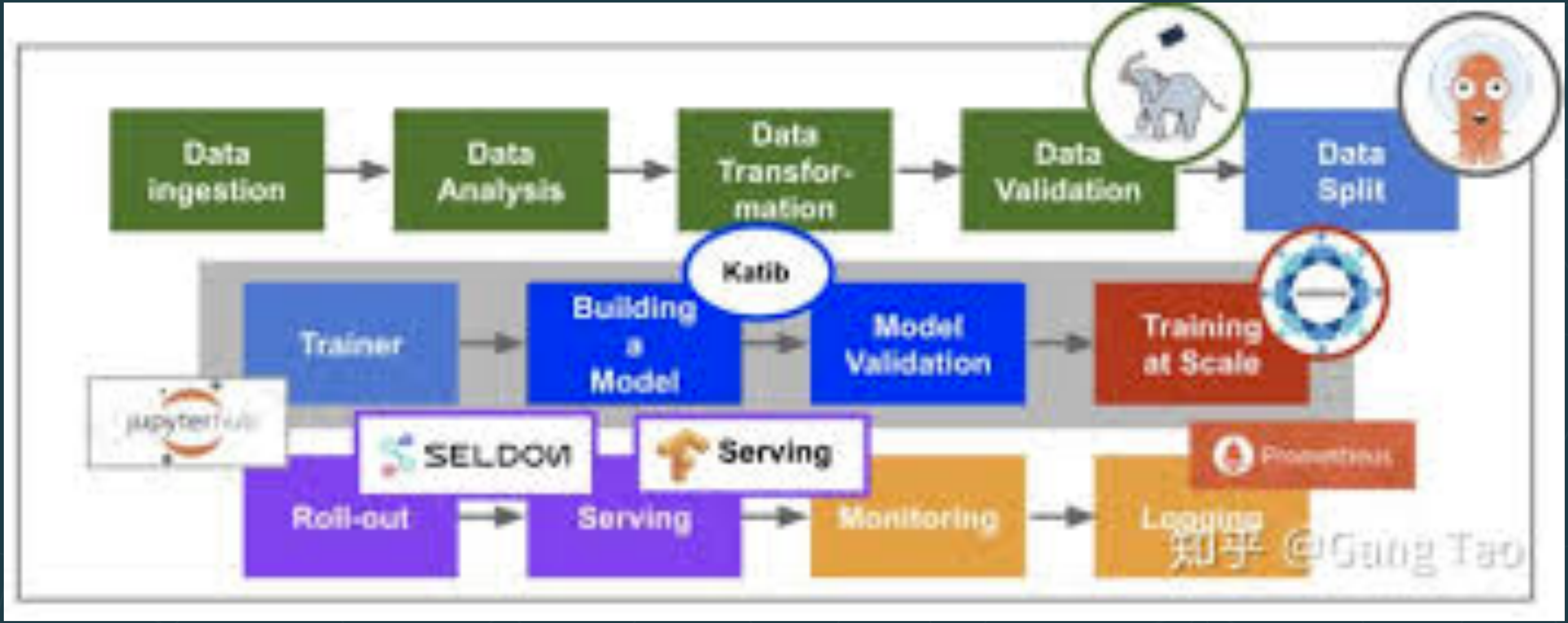
https://www.eetasia.com/news/article/Nvidia_CEO_in_China

Example: TensorFlow serving



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

Example: Kubeflow



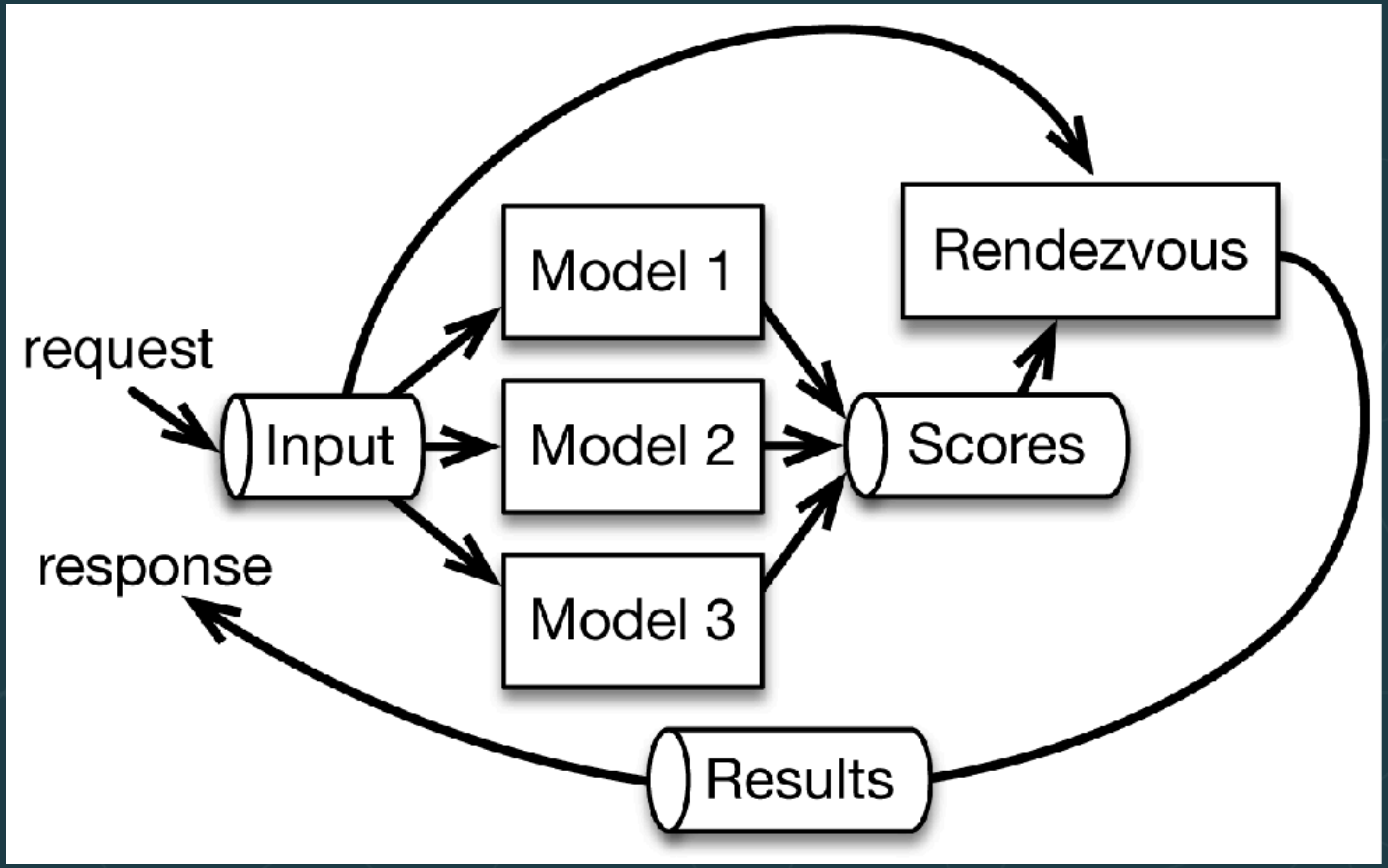
https://zhuanlan.zhihu.com/p/44692757

Rendezvous Architecture

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

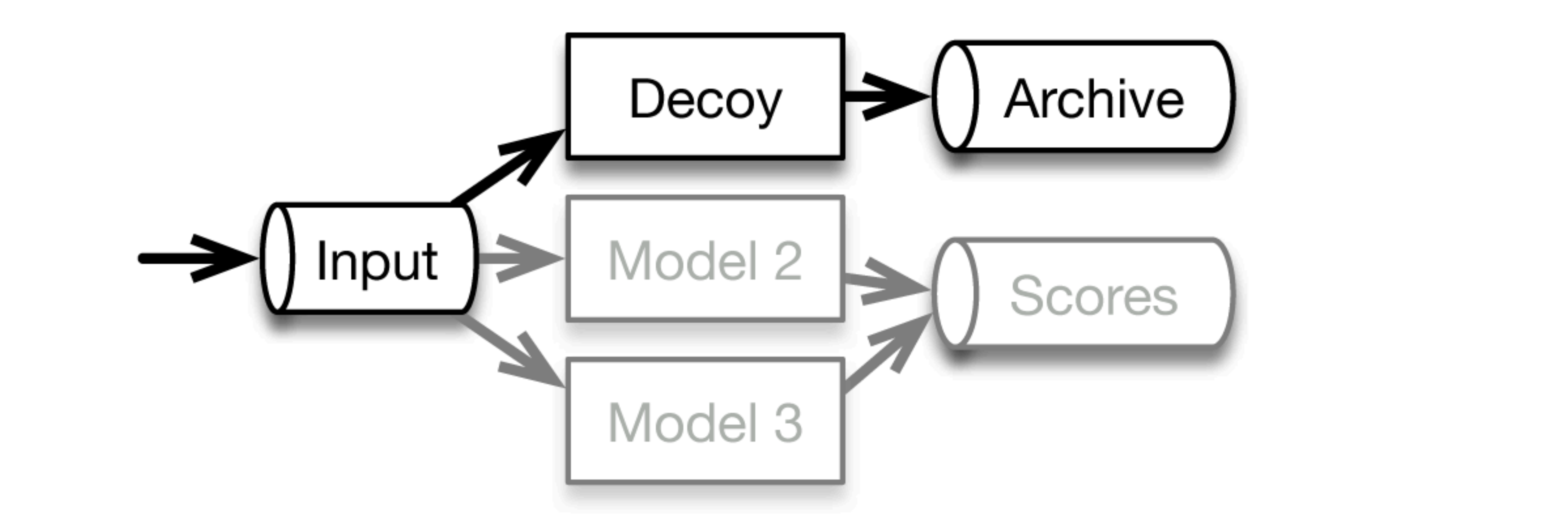
- Collect data at scale from a variety of sources and preserve raw data so that potentially valuable features are not lost.
- Make input and output data available to many independent applications (consumers), on premise, geographically distributed, or in the cloud.
- Manage multiple models during development 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



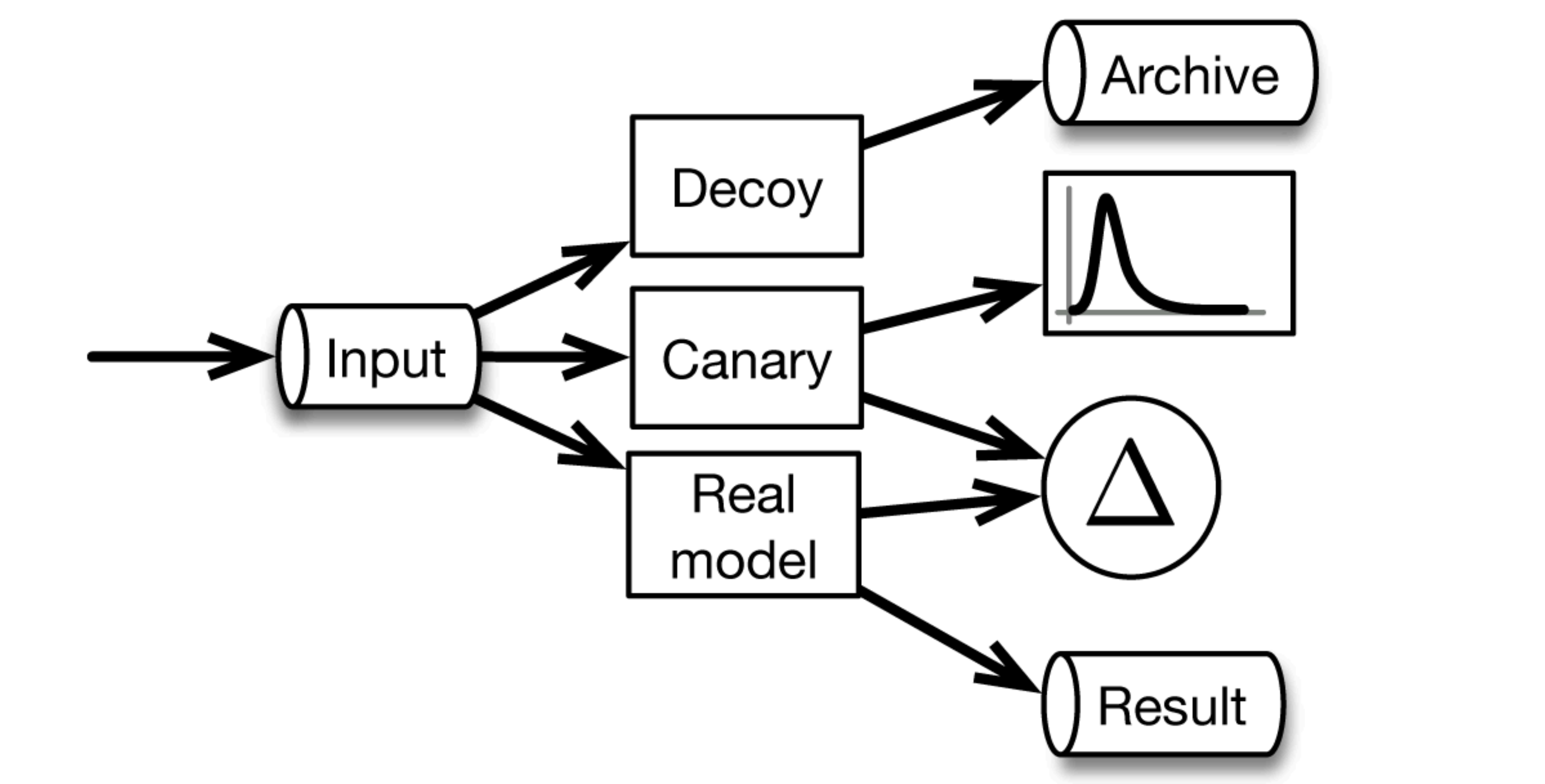
https://mapr.com/ebooks/machine-learning-logistics/ch03.html

Rendezvous Architecture - Decoy



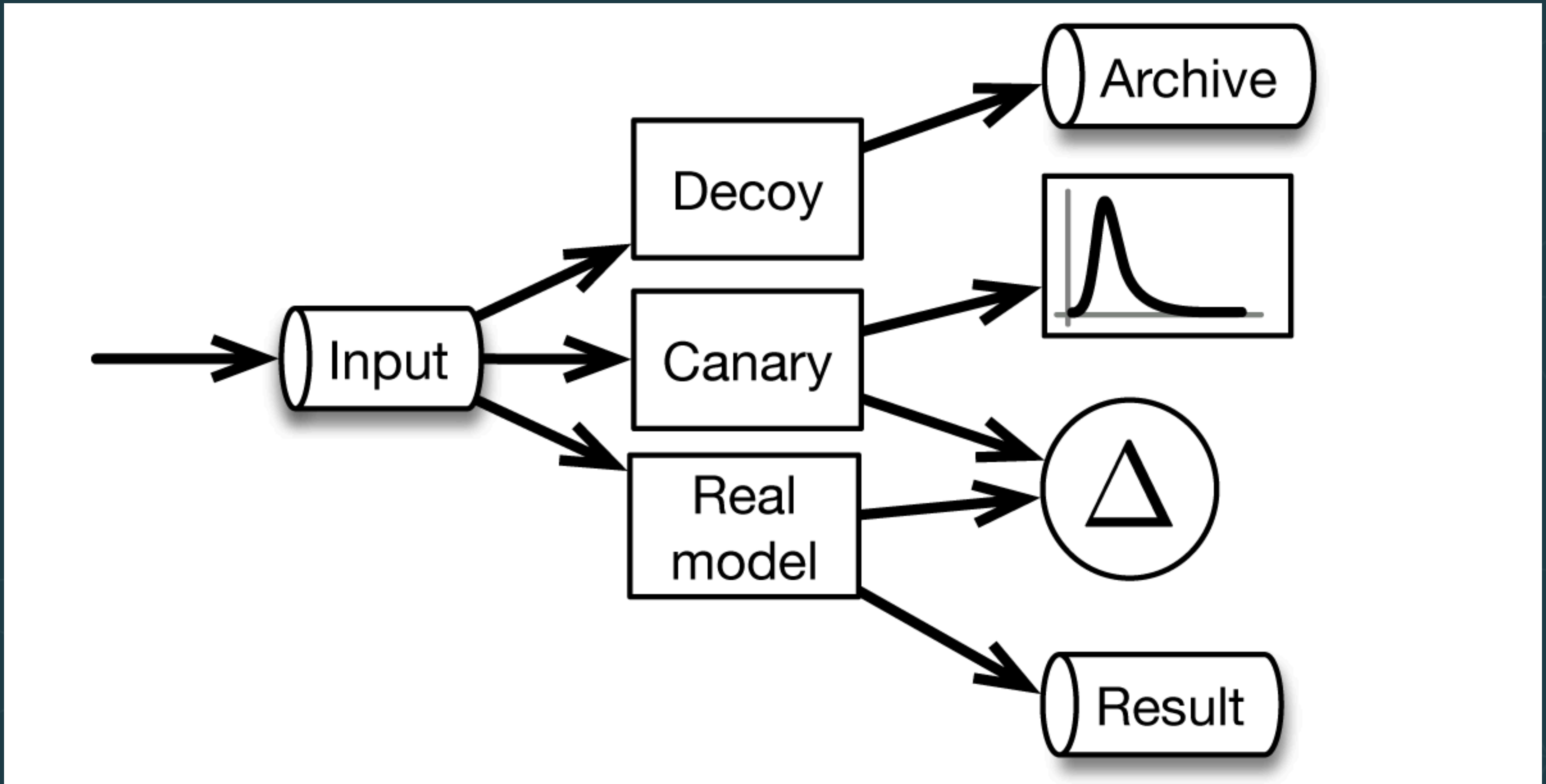
https://mapr.com/ebooks/machine-learning-logistics/ch03.html

Rendezvous Architecture - Canary



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

Rendezvous Architecture - all capabilities



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

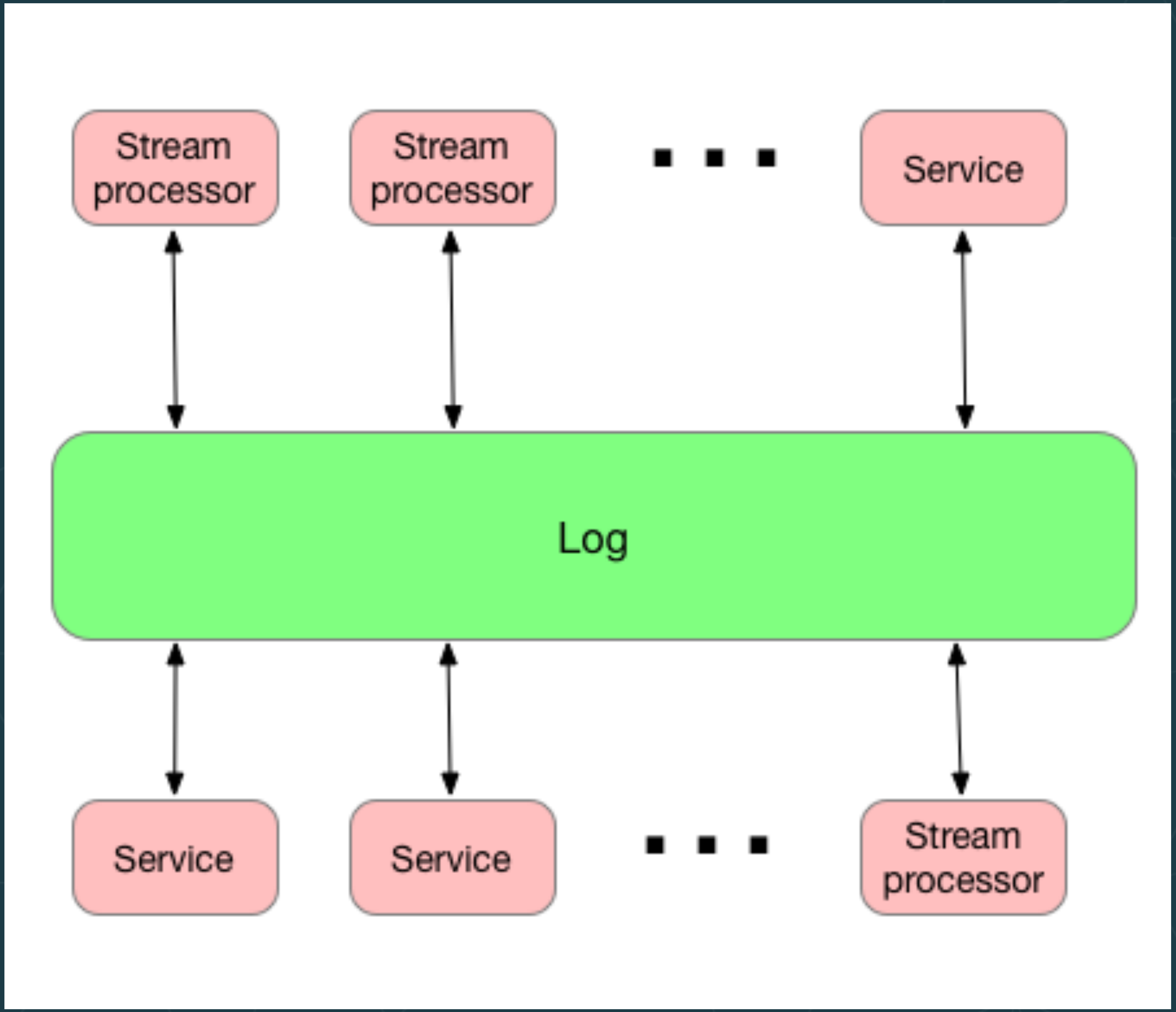
Outline

- Hidden technical debt in machine learning systems
- Model serving patterns
  - Embedding - models as code
  - Models as data
  - 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 then services talking to each other directly.
- Specifically, stream processors don’t talk explicitly to other services, but send async. messages through the log.

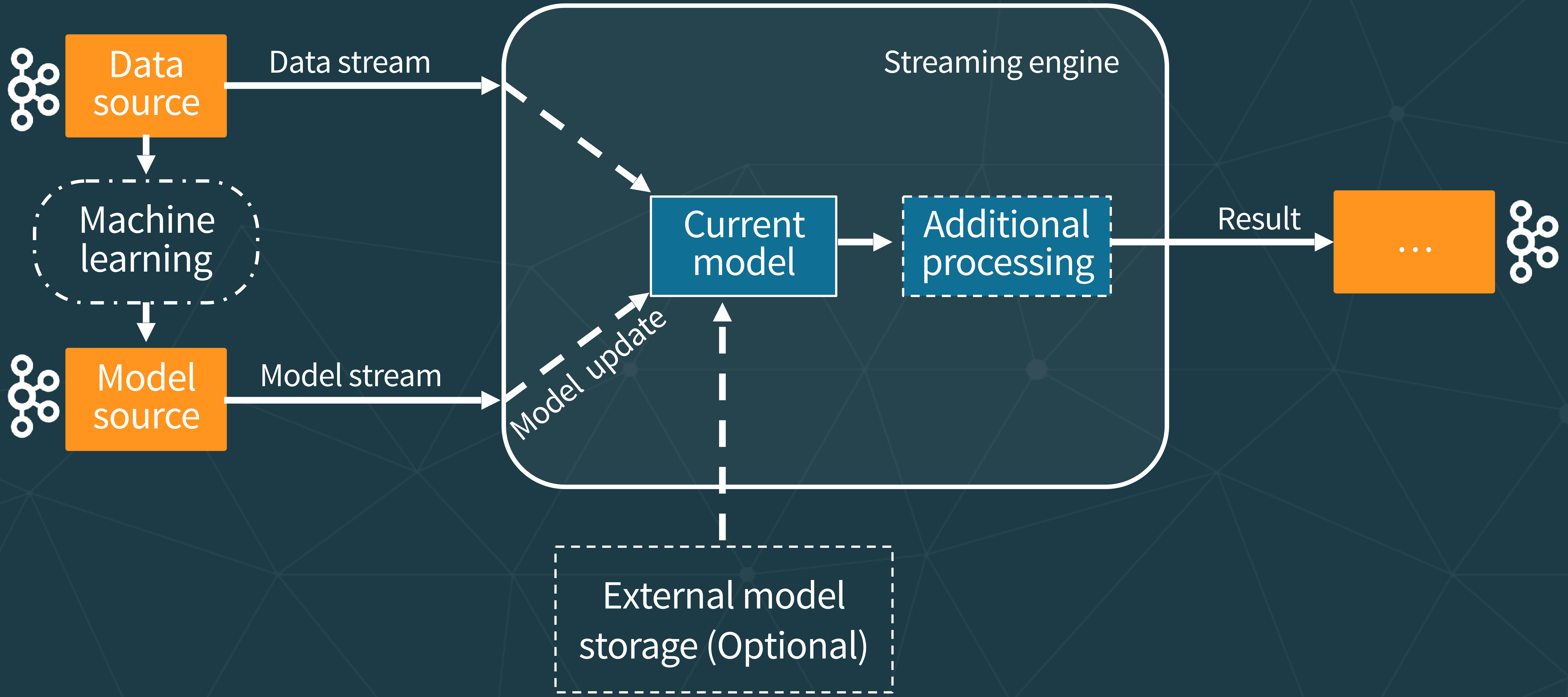
Example: Kafka



Model Serving in a Log-Driven Enterprise

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

We'll use Kafka as the "log" system.



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)

```scala
trait Model[RECORD, RESULT] {
  def score(input : RECORD) : RESULT
  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

```scala
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
)
```

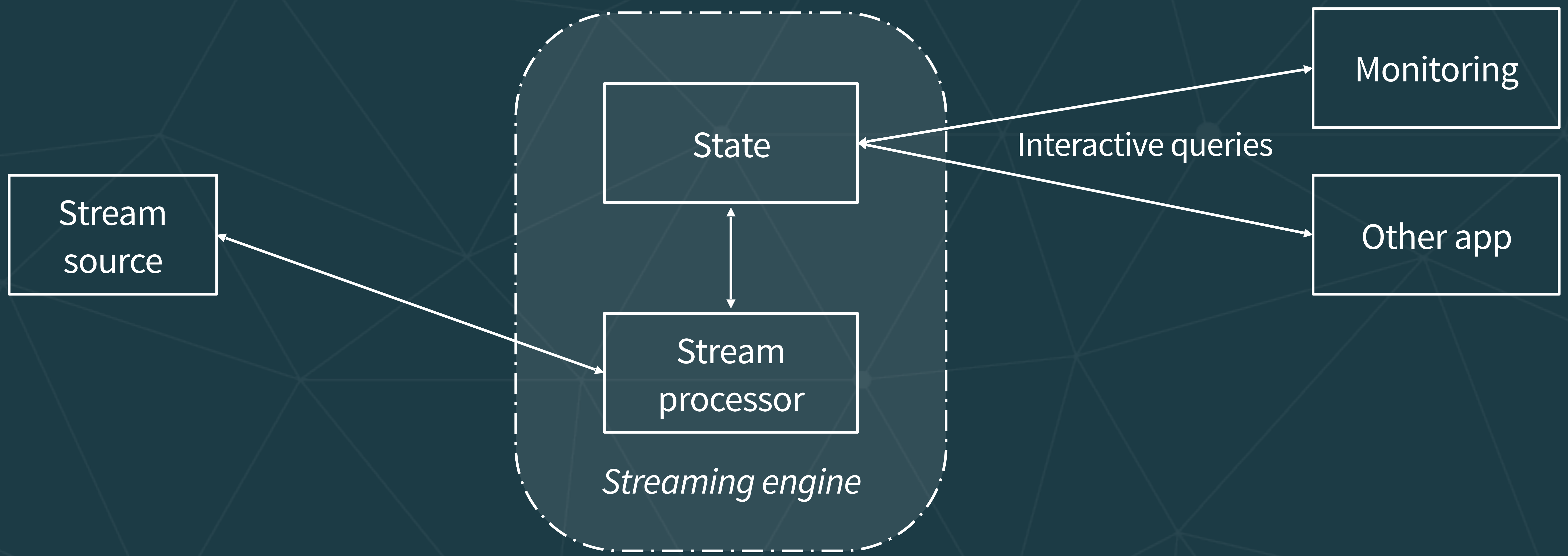
Lightbend

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.



Lightbend

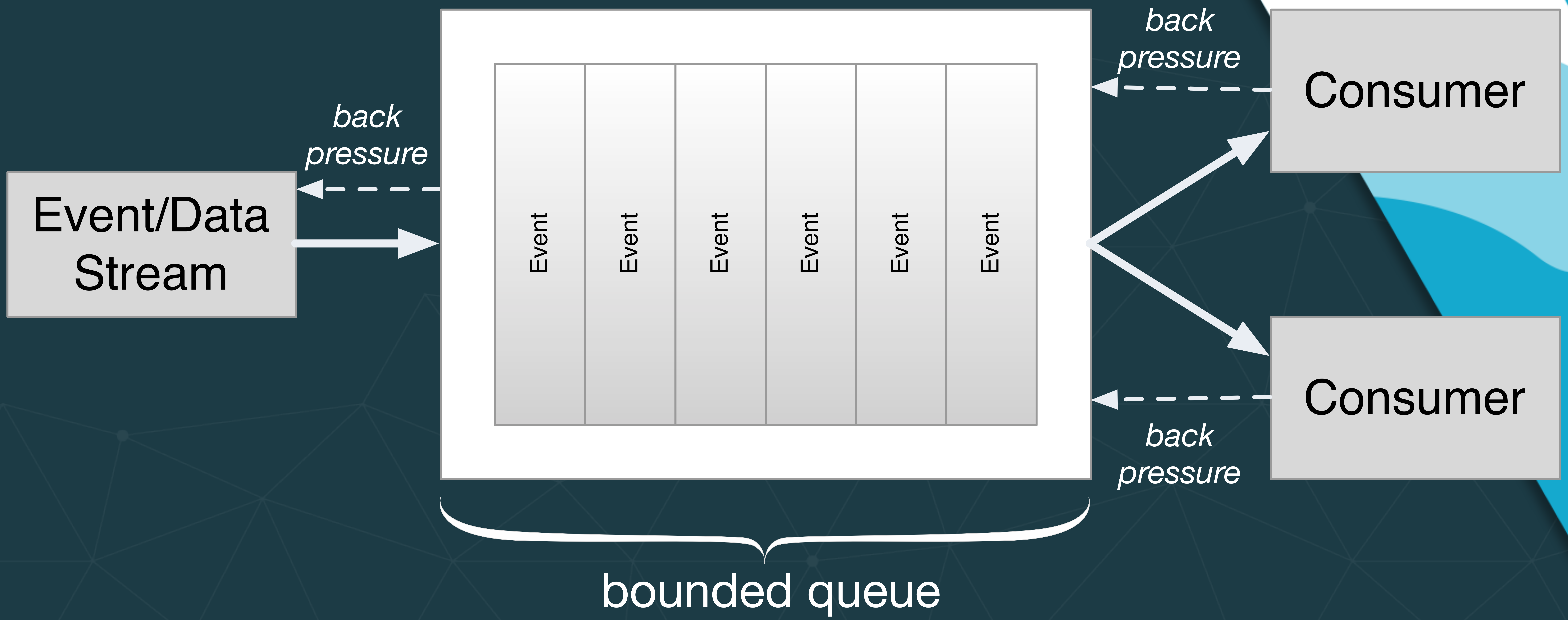
Akka Streams

Lightbend

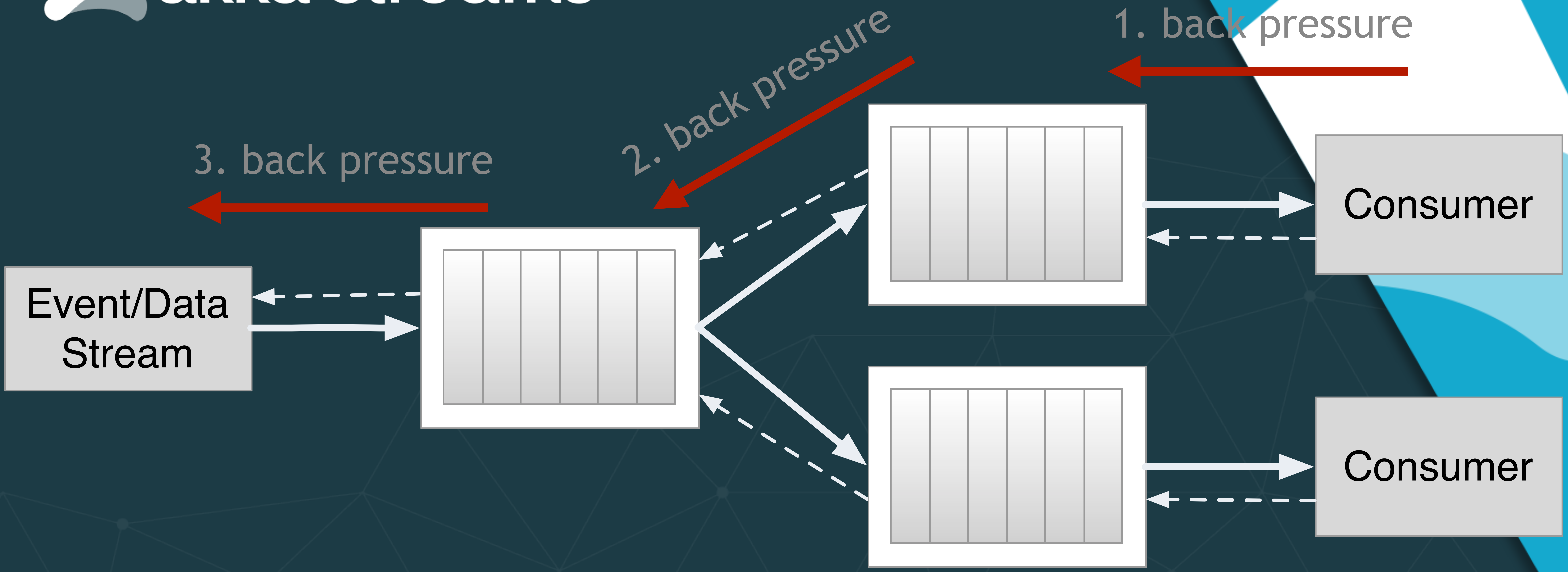
akka streams

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

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
    - like Camel, but implements Reactive Streams
  - Commercial support from Lightbend

akka streams

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

```scala
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
```

```scala
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))
```

Imports!

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

```scala
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
```

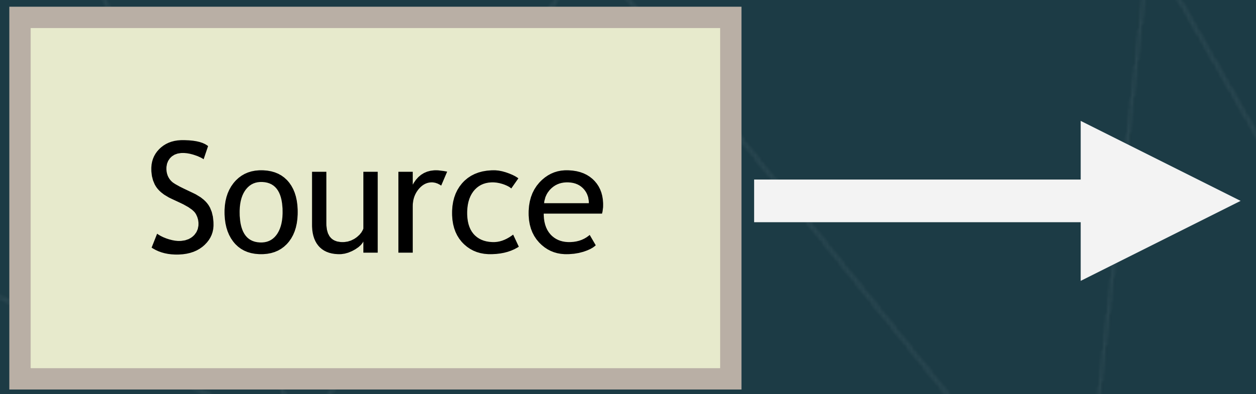
```scala
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed
import akka.actor.ActorSystem
import scala.concurrent._
import scala.concurrent.duration._

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



```scala
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

```scala
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 1
val factorials = source.scan(BigInt(1)) ((acc, ne
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

```scala
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))
```

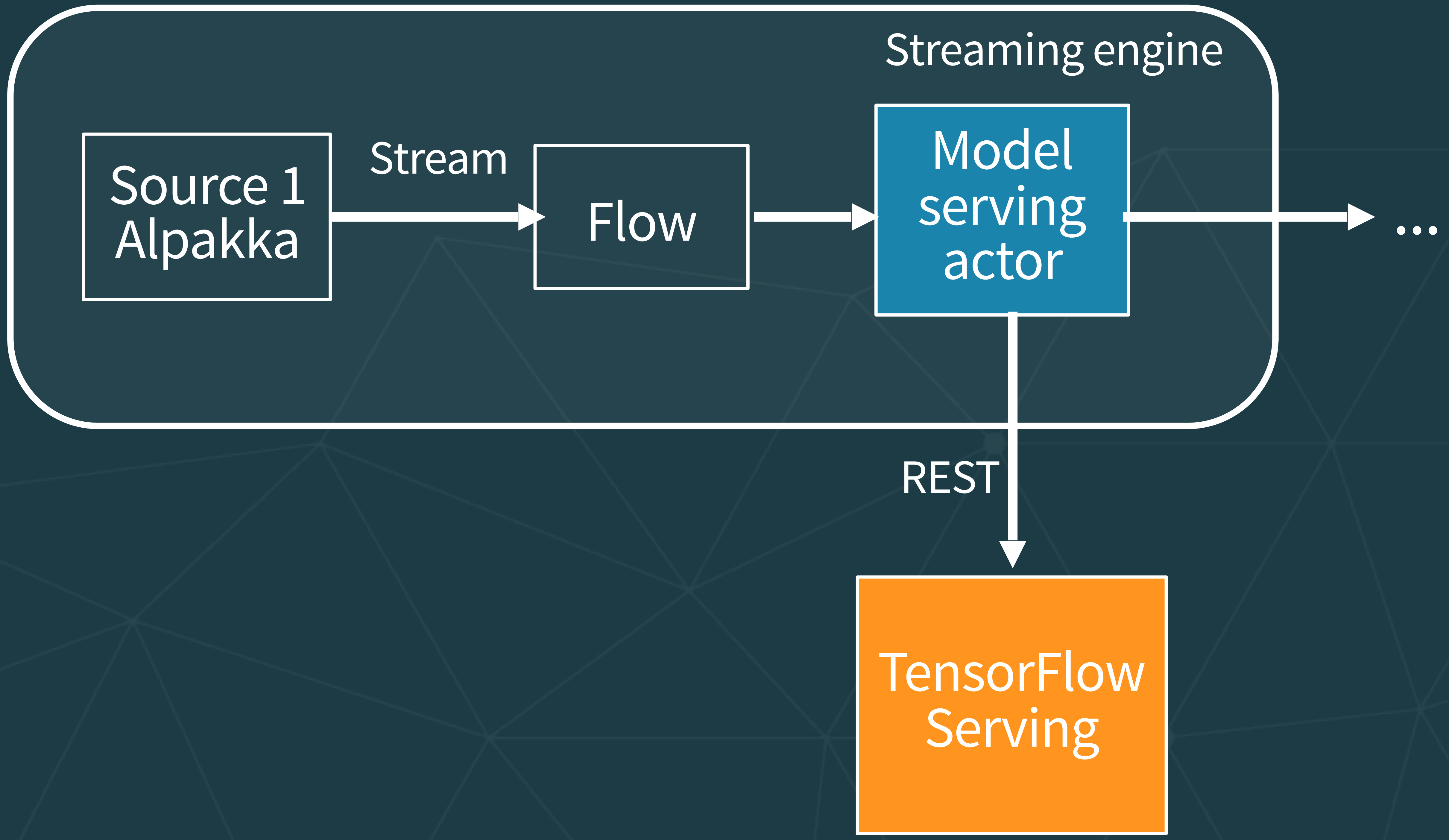
```
1
2
6
24
120
720
0
20
880
28800
```

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

Source → Flow → Sink

Using TensorFlow Serving in Akka Streams

Use Custom Actor to access TensorFlow Serving Server



Code Time

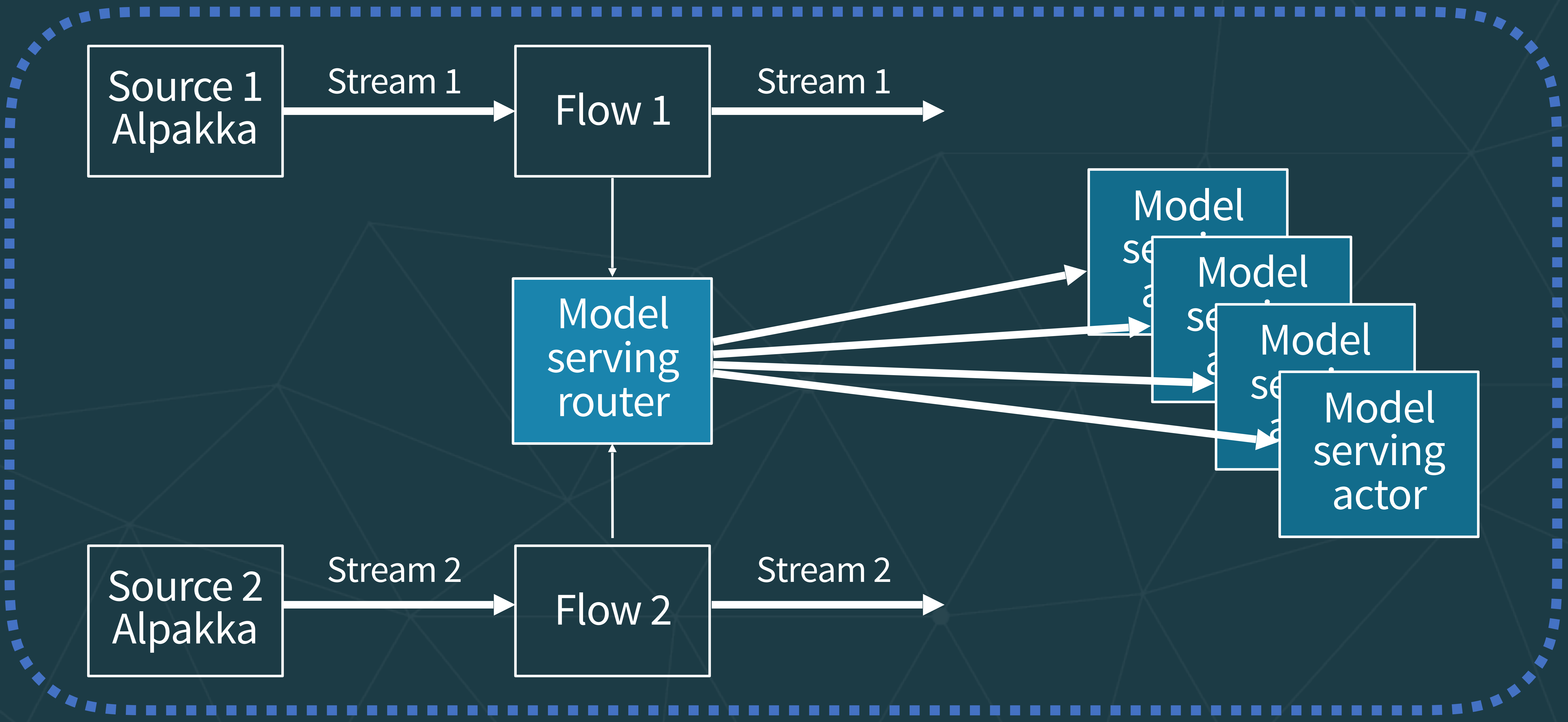
- Open the example code project
- We’ll walk through the project at a high level
- Familiarize yourself with the *tensorflowserver* code
- Load and start the TensorFlow model serving Docker image
  - See Using TensorFlow Serving in the README
- Try the implementation and see if you have any questions

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)

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!!



Microservice boundary

Akka Streams Example

Code time

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

Akka Streams Example

Check Queryable state

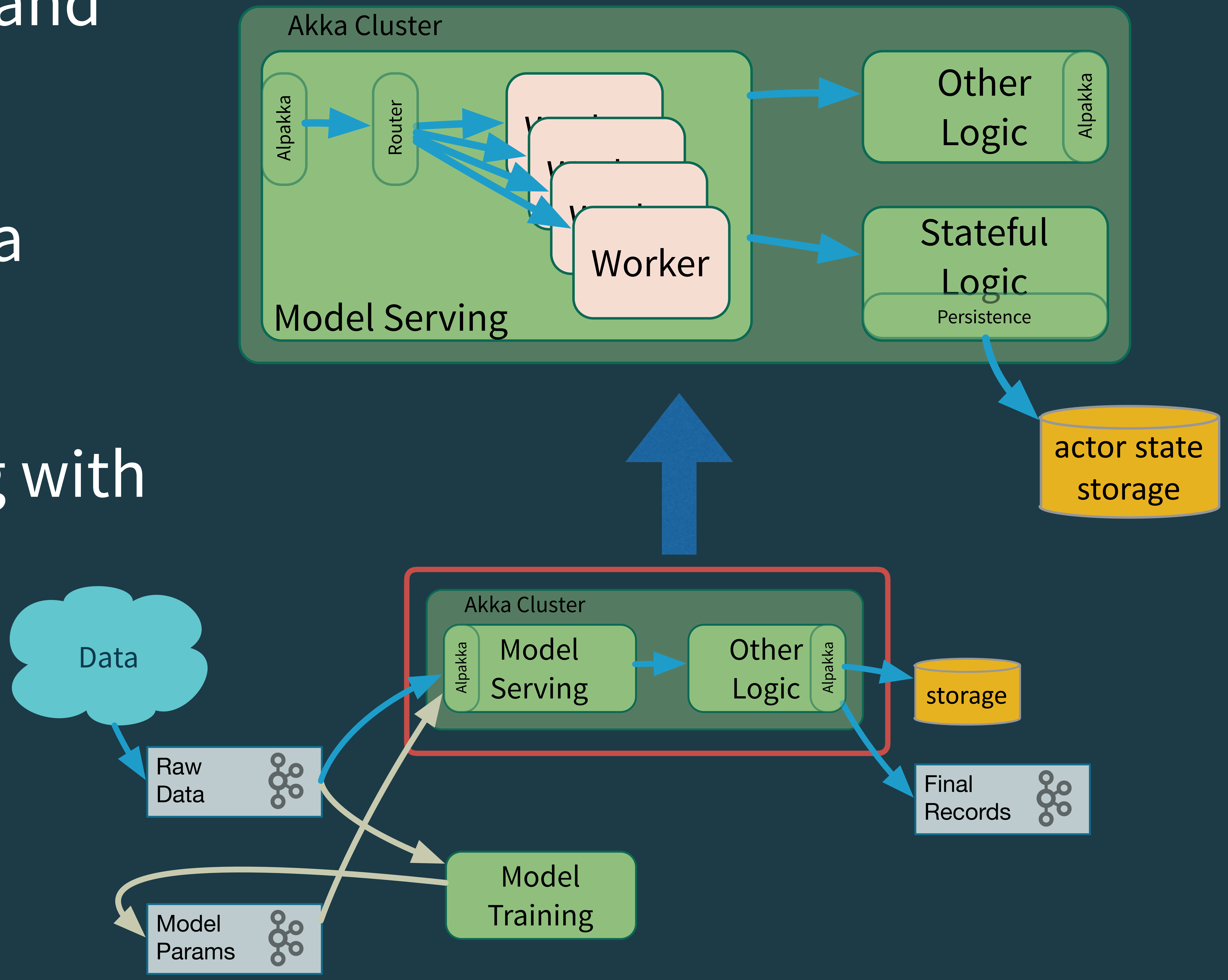
Curl or open in a browser:

http://localhost:5500/models

http://localhost:5500/state/wine

Handling Other Production Concerns with Akka and Akka Streams

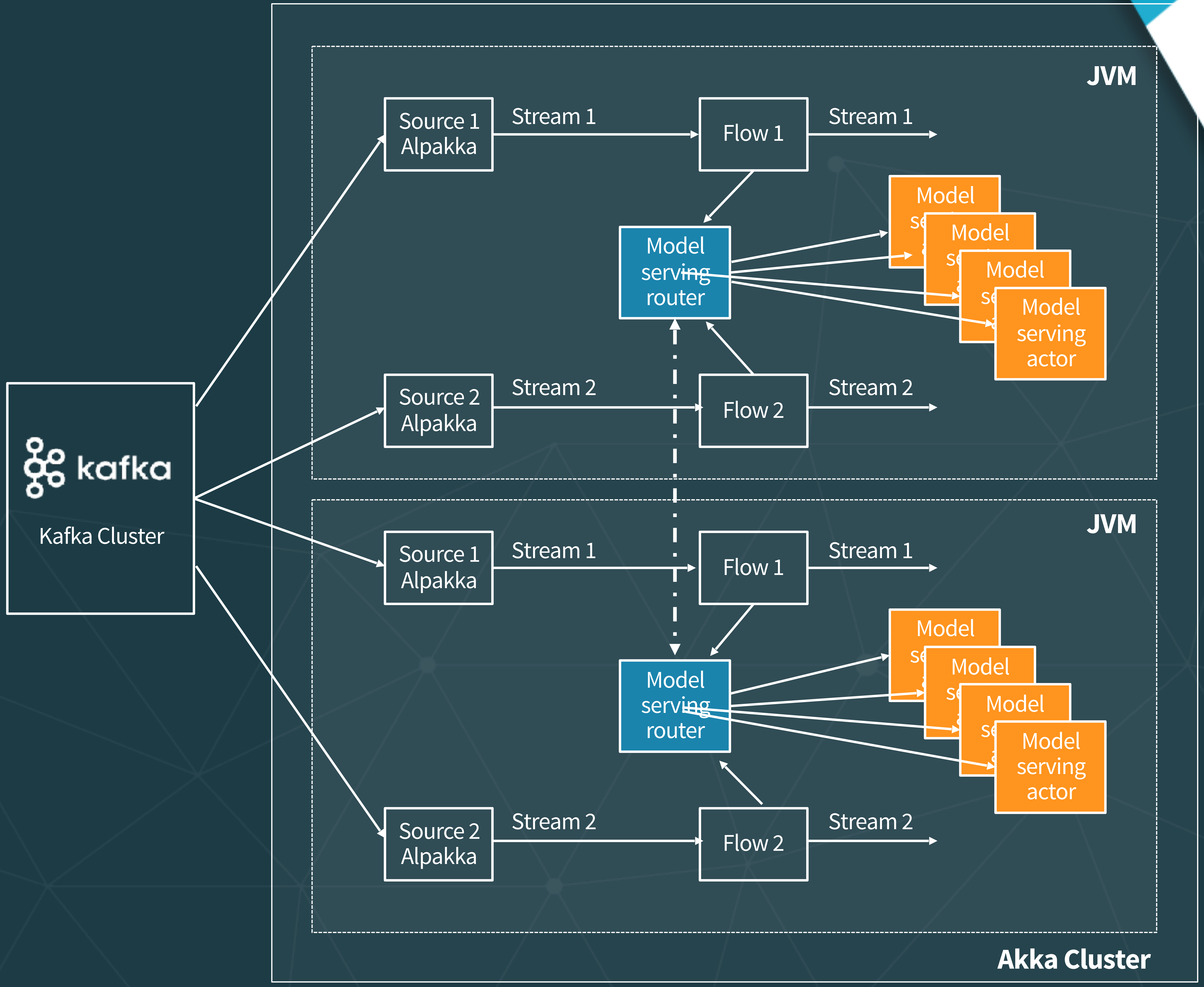
- Scale scoring with workers and routers, across a cluster
- Persist actor state with Akka Persistence
- Connect to *almost* anything with Alpakka



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/

Flink

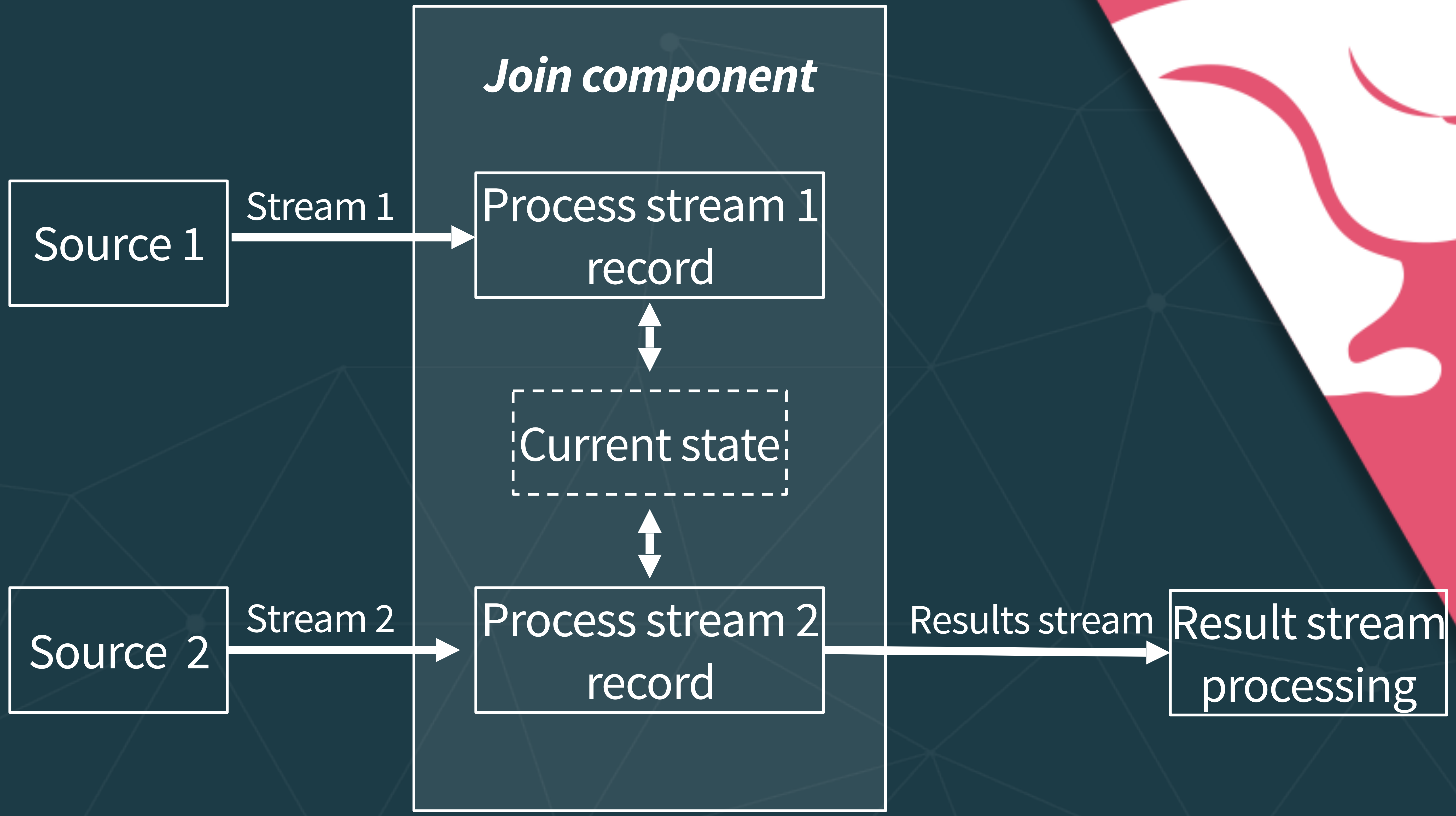
Flink

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.

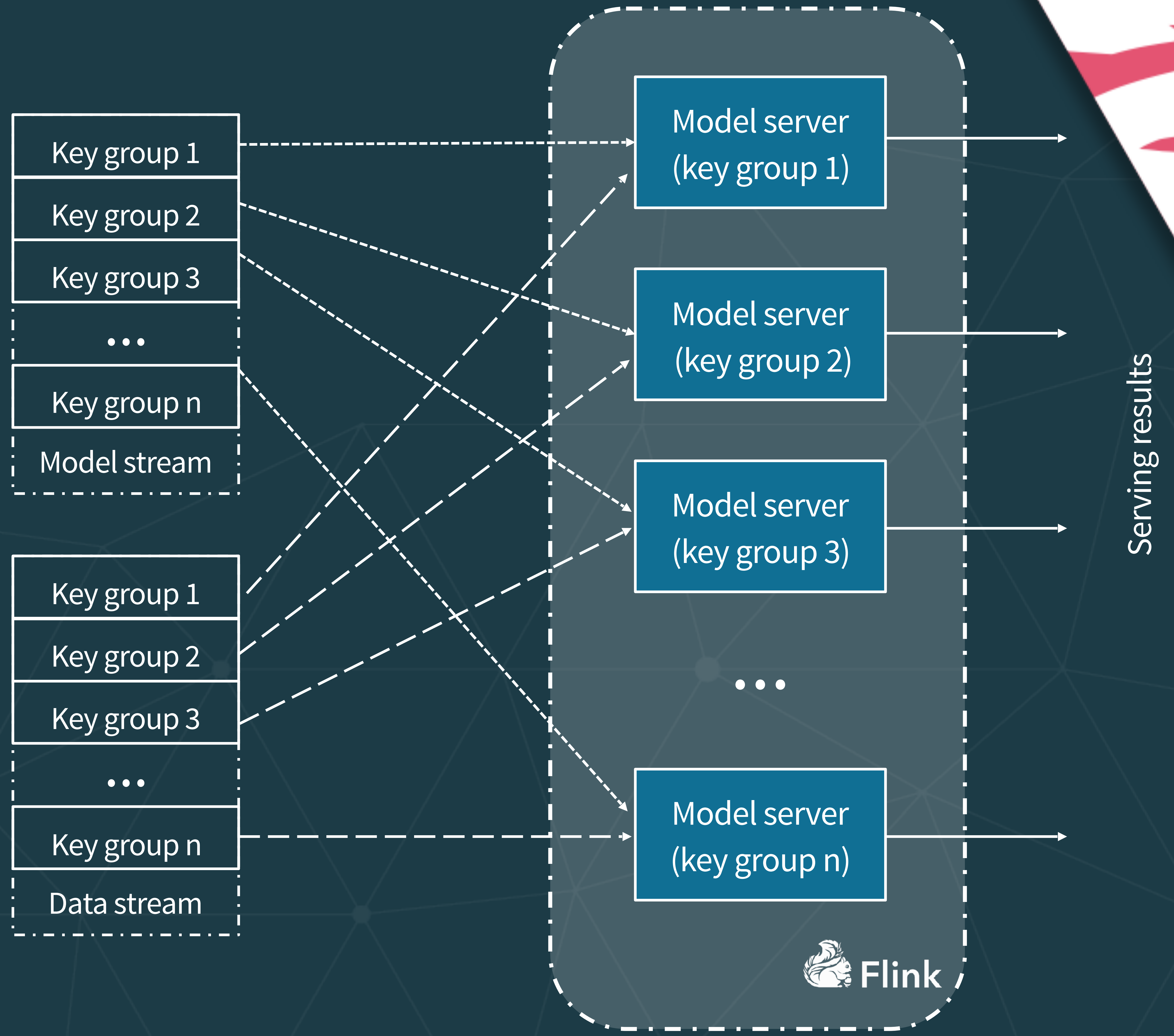
Flink Low Level Join

- 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



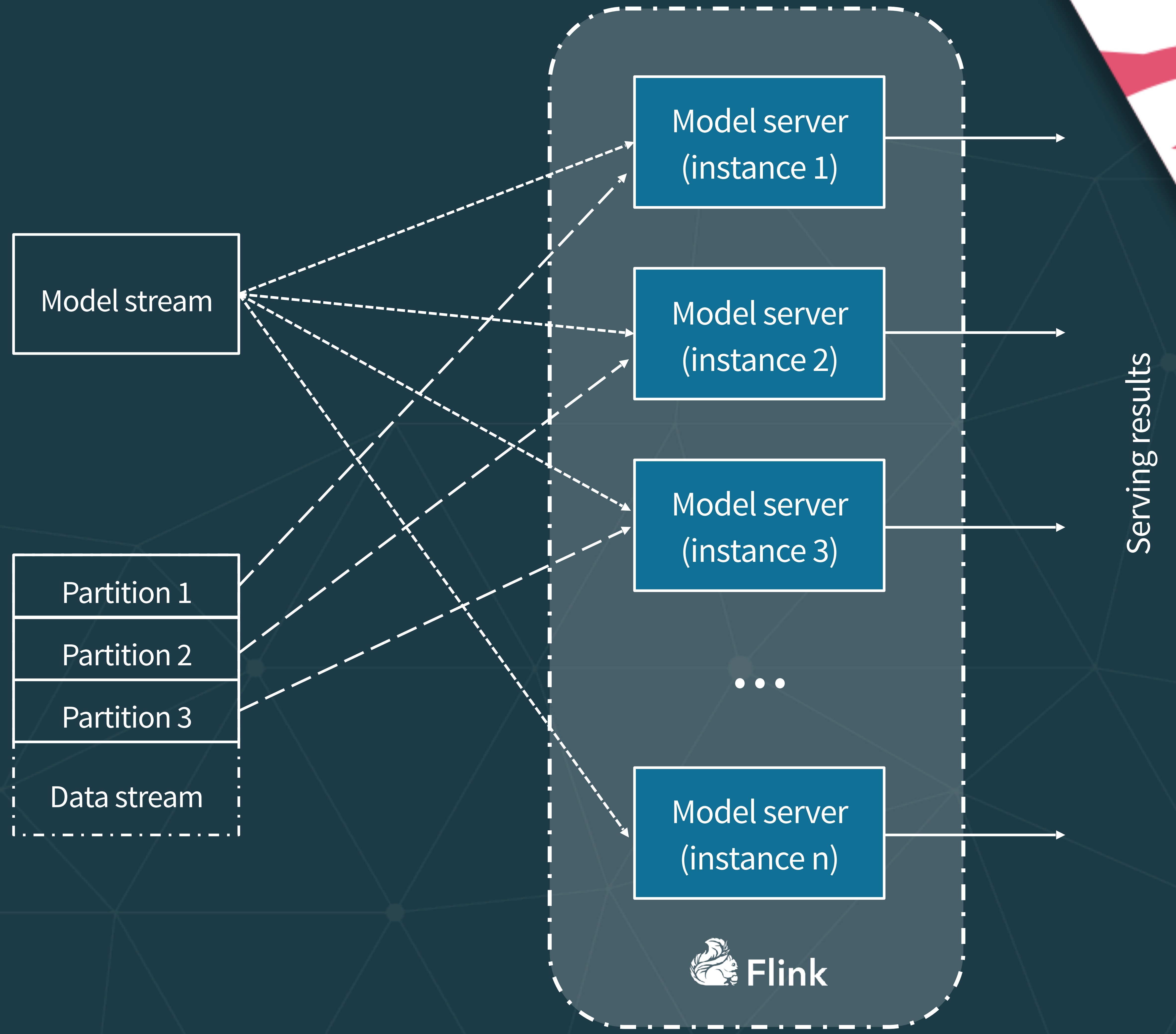
Key based join

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.



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

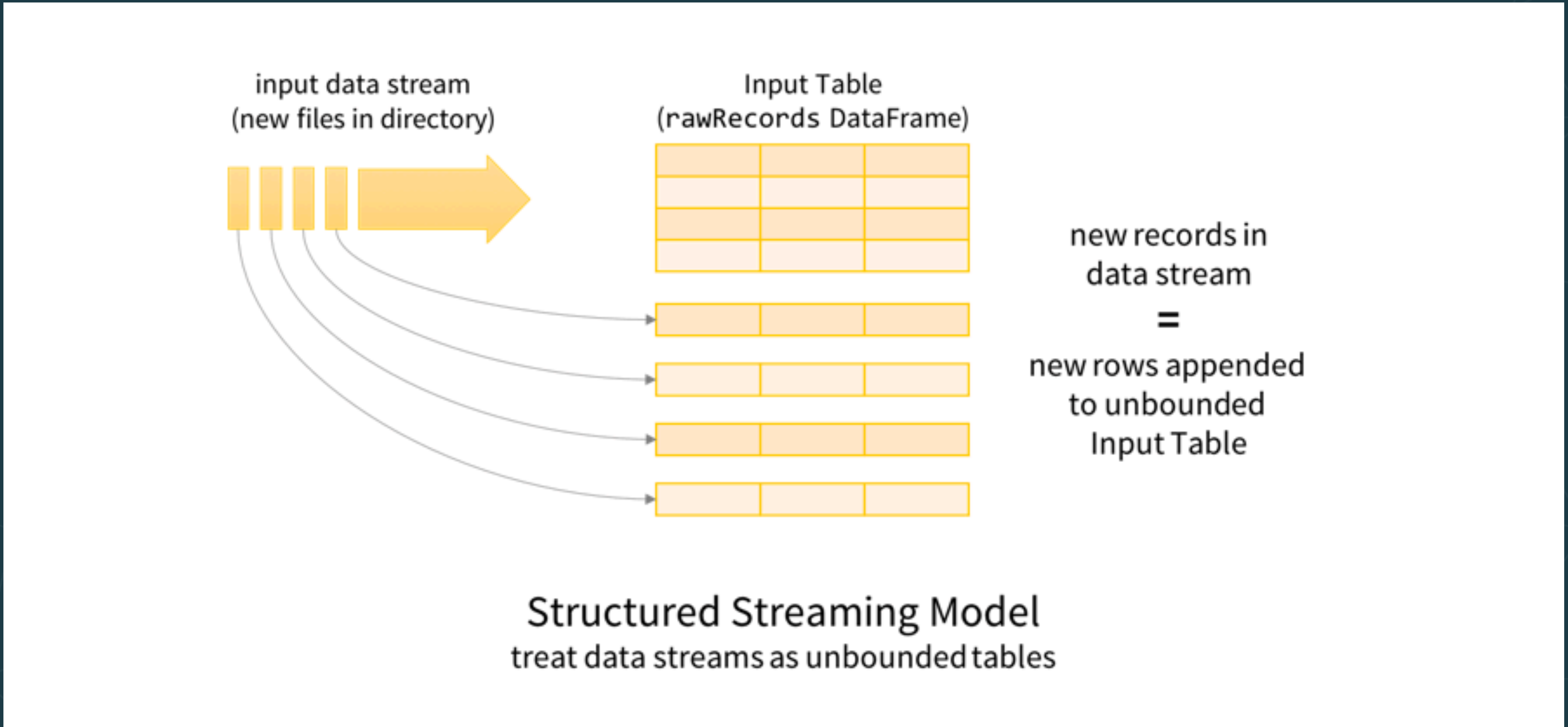
Spark Structured Streaming

Spark Structured Streaming

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

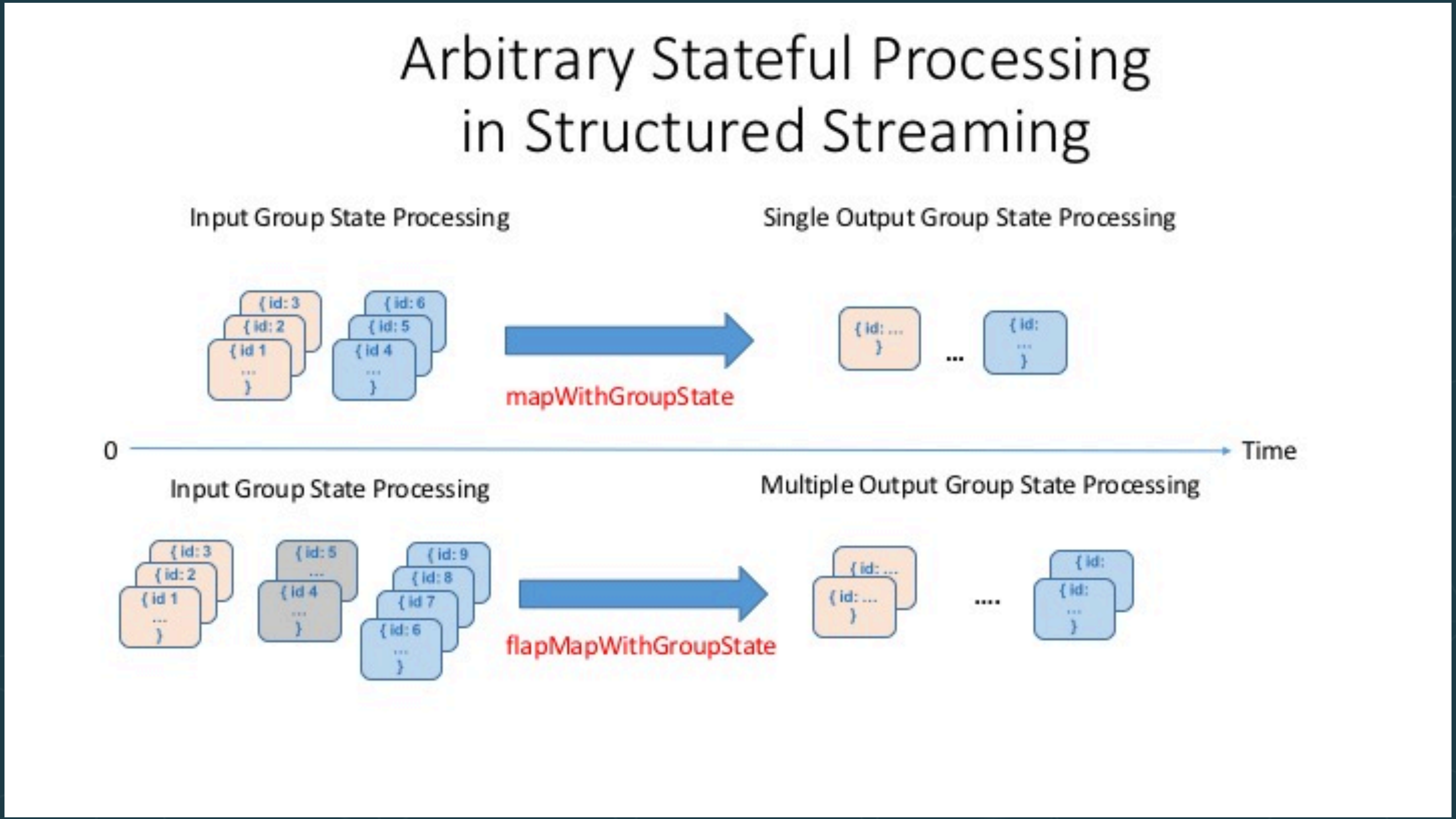
- Scales well, runs on thousands of nodes.
- Express your streaming computation the same way you would express a batch SQL computation on static data:
  - The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

Spark Structured Streaming



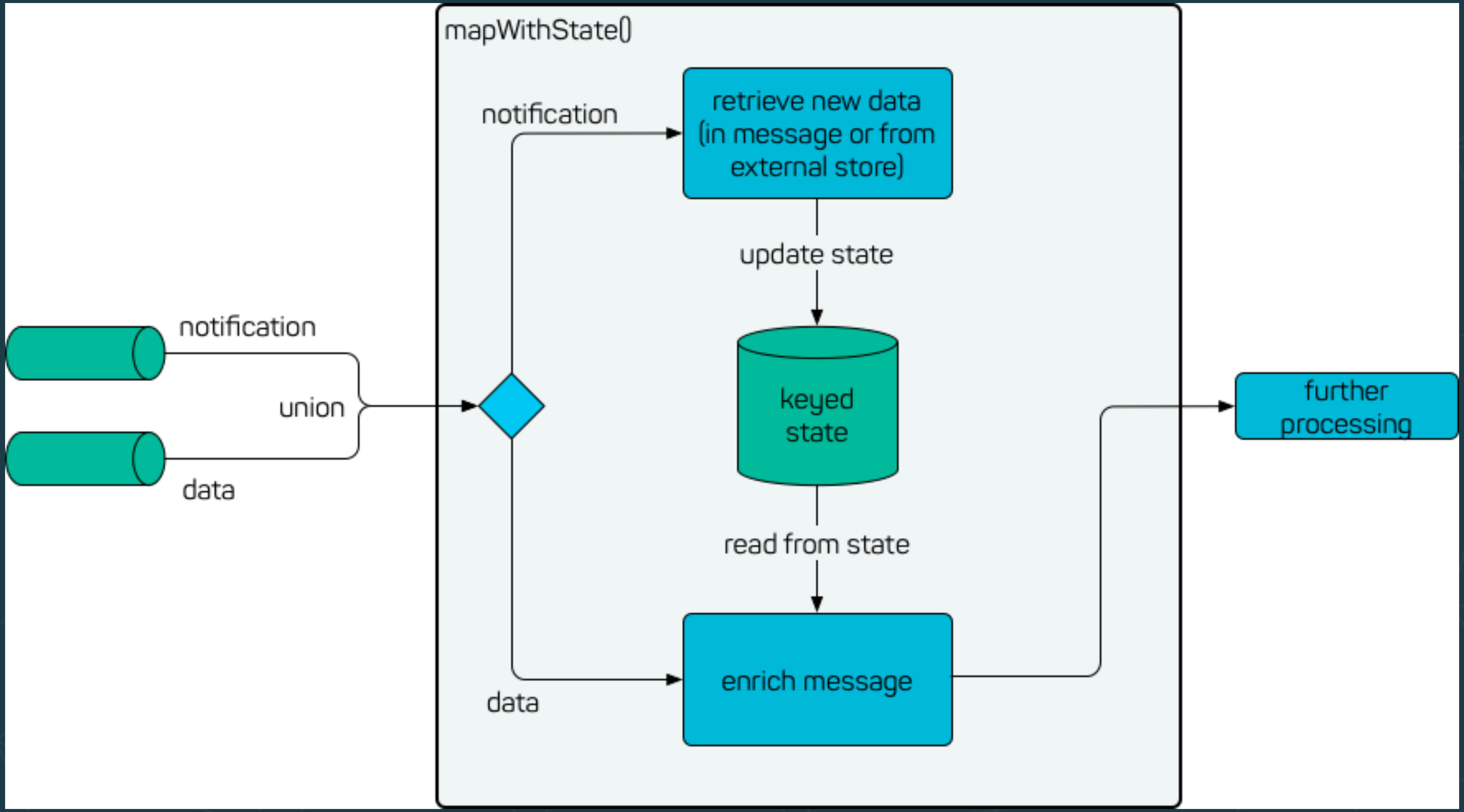
spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Spark Structured Streaming - State



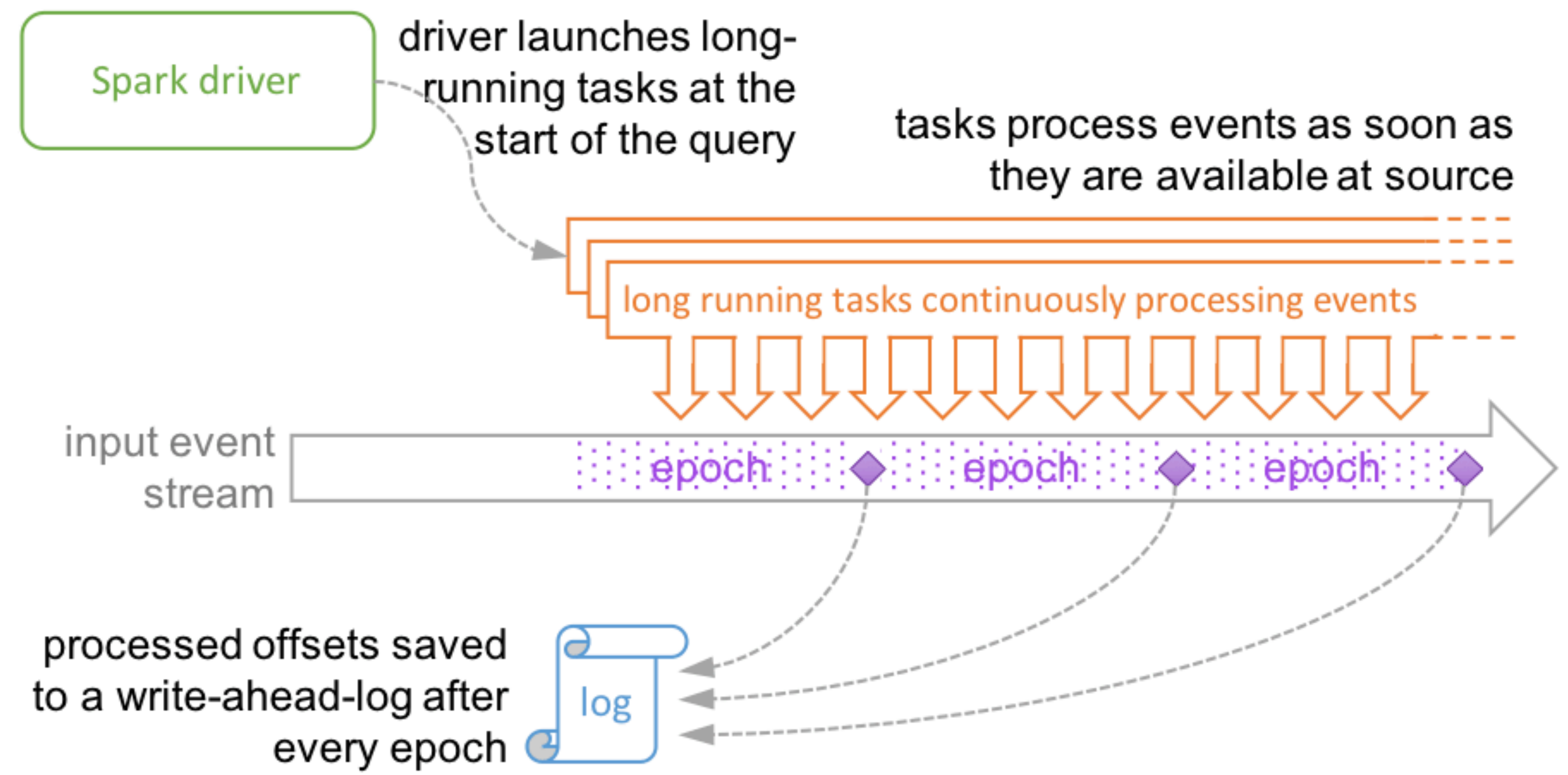
https://databricks.com/blog/2017/10/17/arbitrary-stateful-processing-in-apache-sparks-structured-streaming.html

Spark Structured Streaming - mapWithState



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

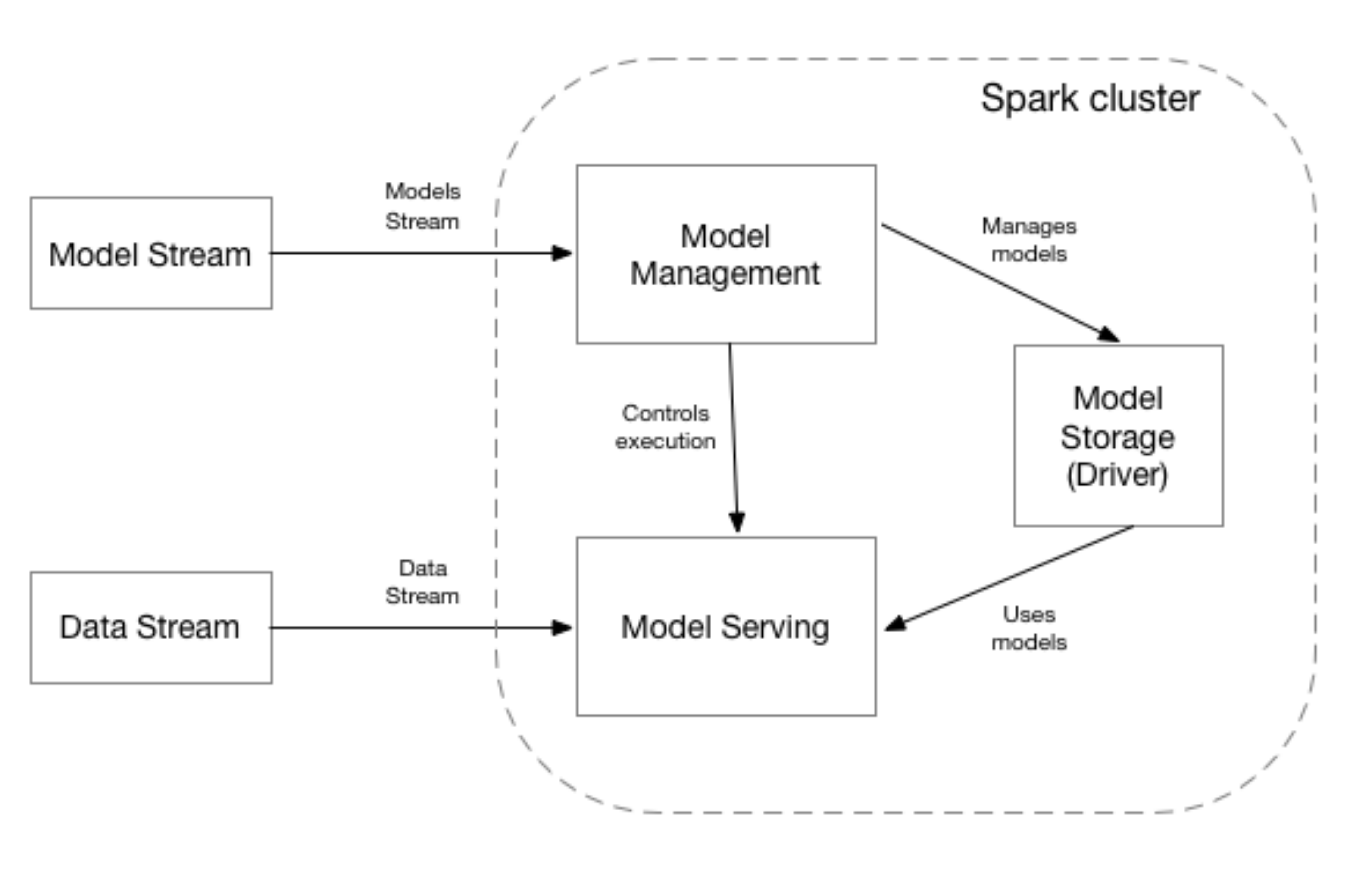
Spark Structured Streaming - continuous processing



Continuous Processing uses long-running tasks to continuously process events

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.

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

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
  - …

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**

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

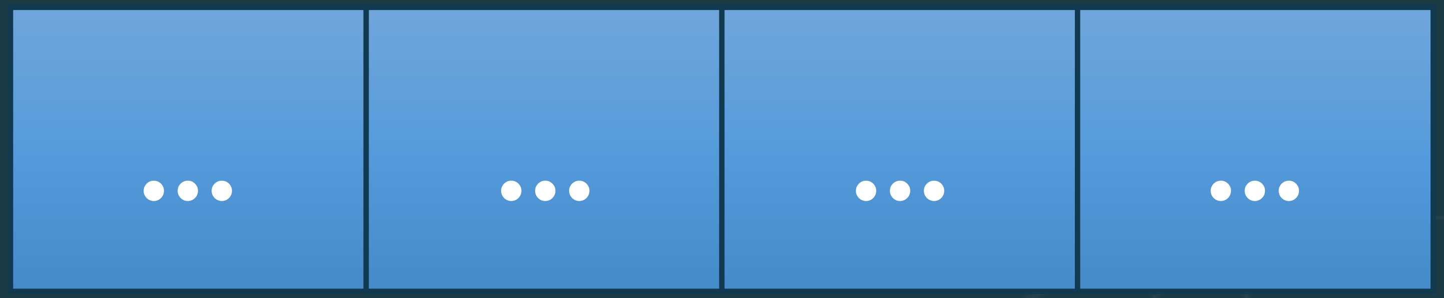
- You can’t answer the question unless you know which model version was actually used for **R**

Model Traceability Requirements

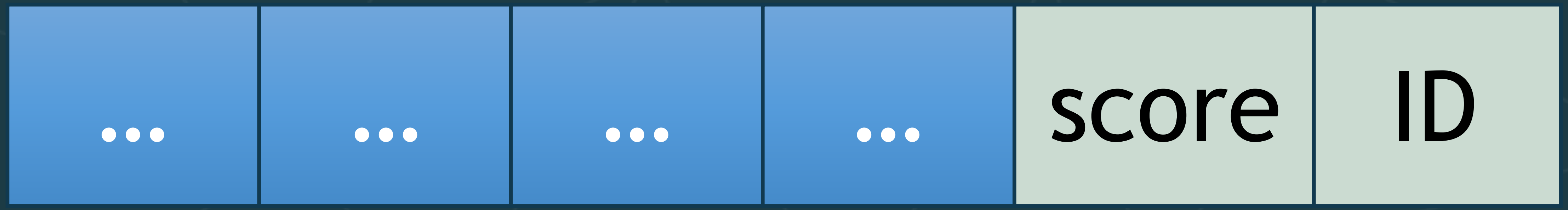
- A model repository
- Information stored for each model instance, possibly including:
  - Name
  - Version (or other unique ID)
  - Creation, deployment, and retirement dates
  - Model parameters
  - Quality metric
  - …

Model Traceability in Use

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

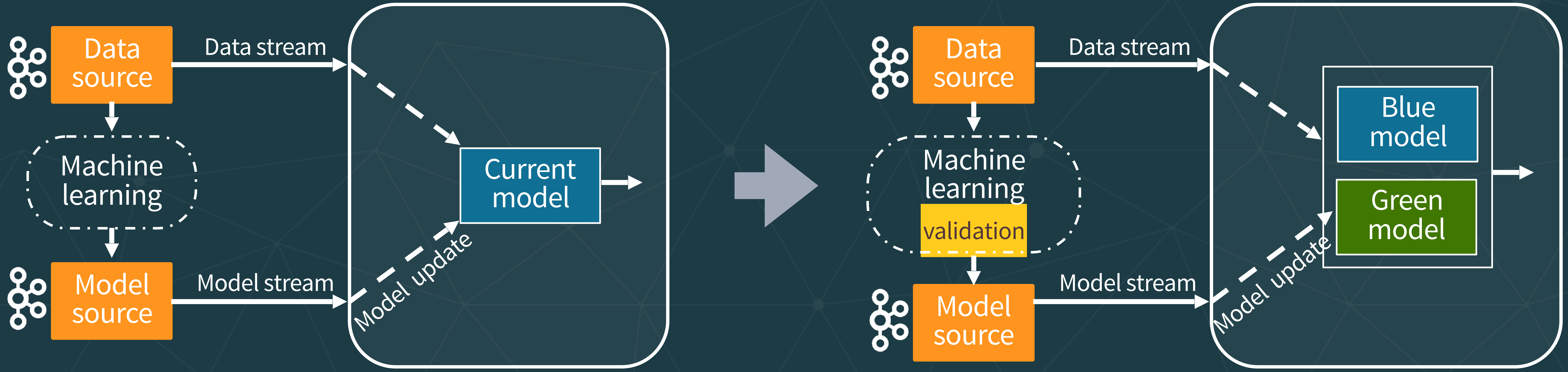


  - Output Record with Score, model version ID



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?



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.

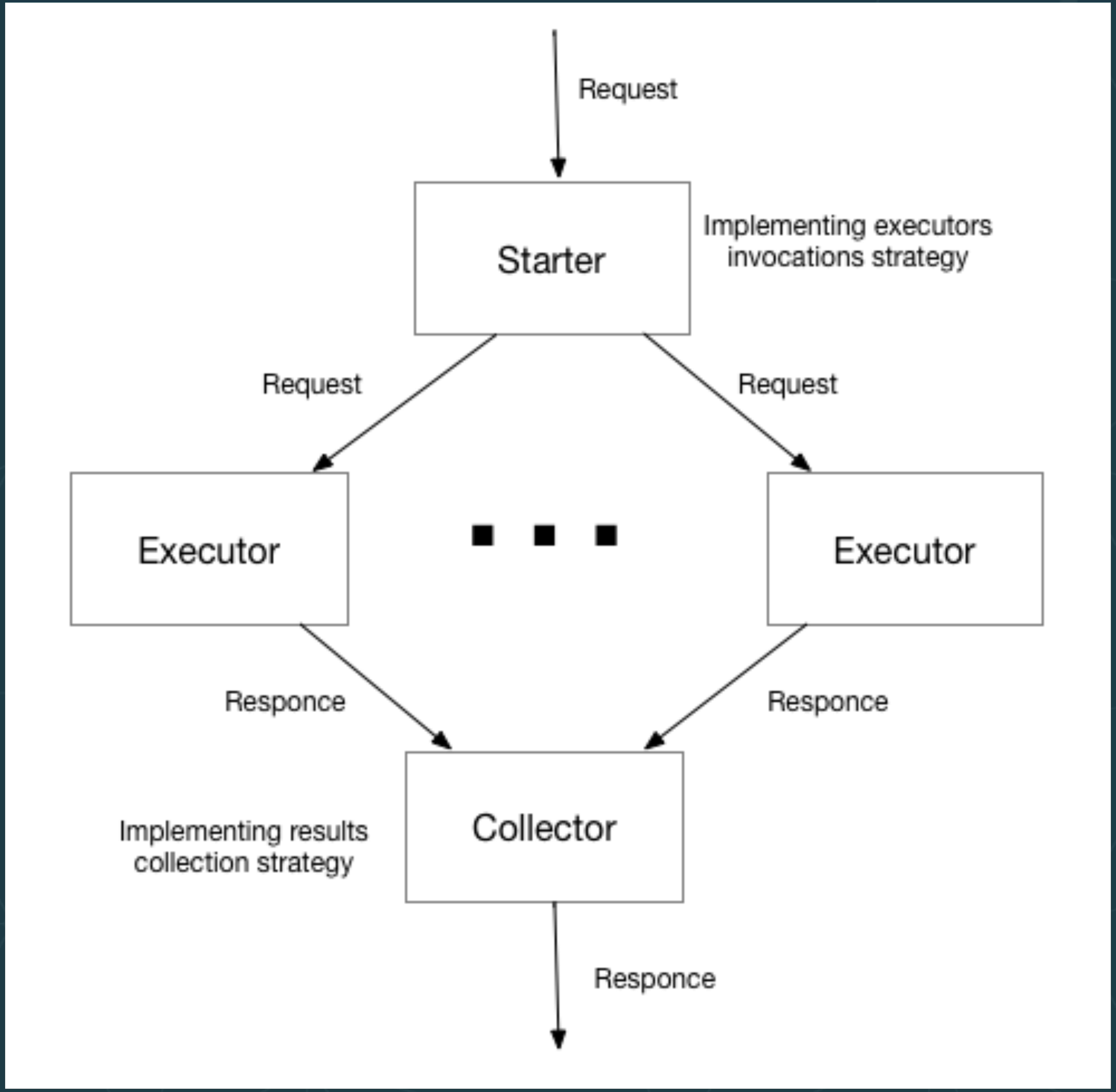
Speculative Execution

- 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

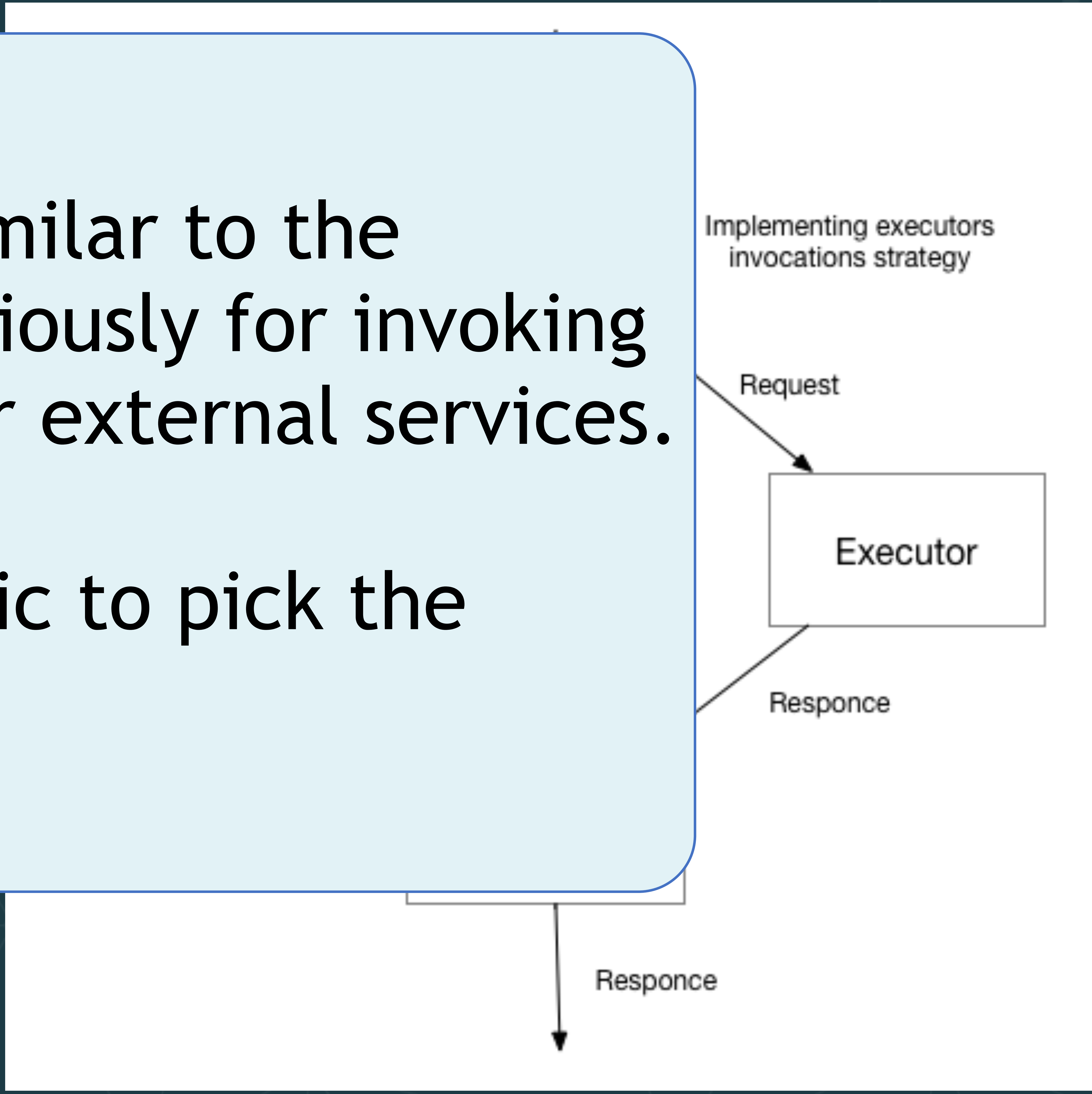


General Architecture for Speculative Execution

- Starter (p
parallelis
strategy
- Parallel 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.



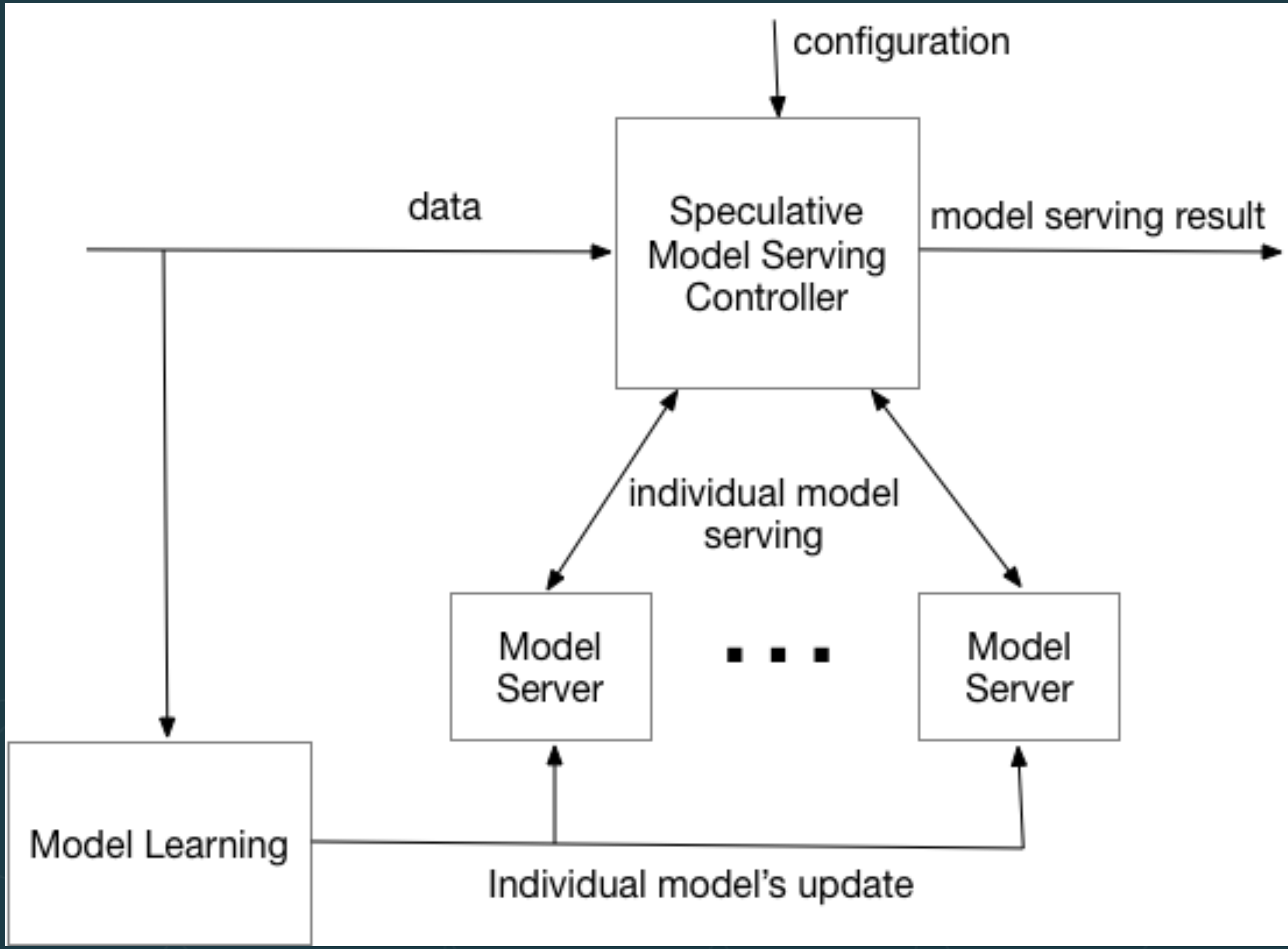
Model Serving Use Case - Guarantee Execution Time

- 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?)

Model Serving Use Case - Ensembles of Models

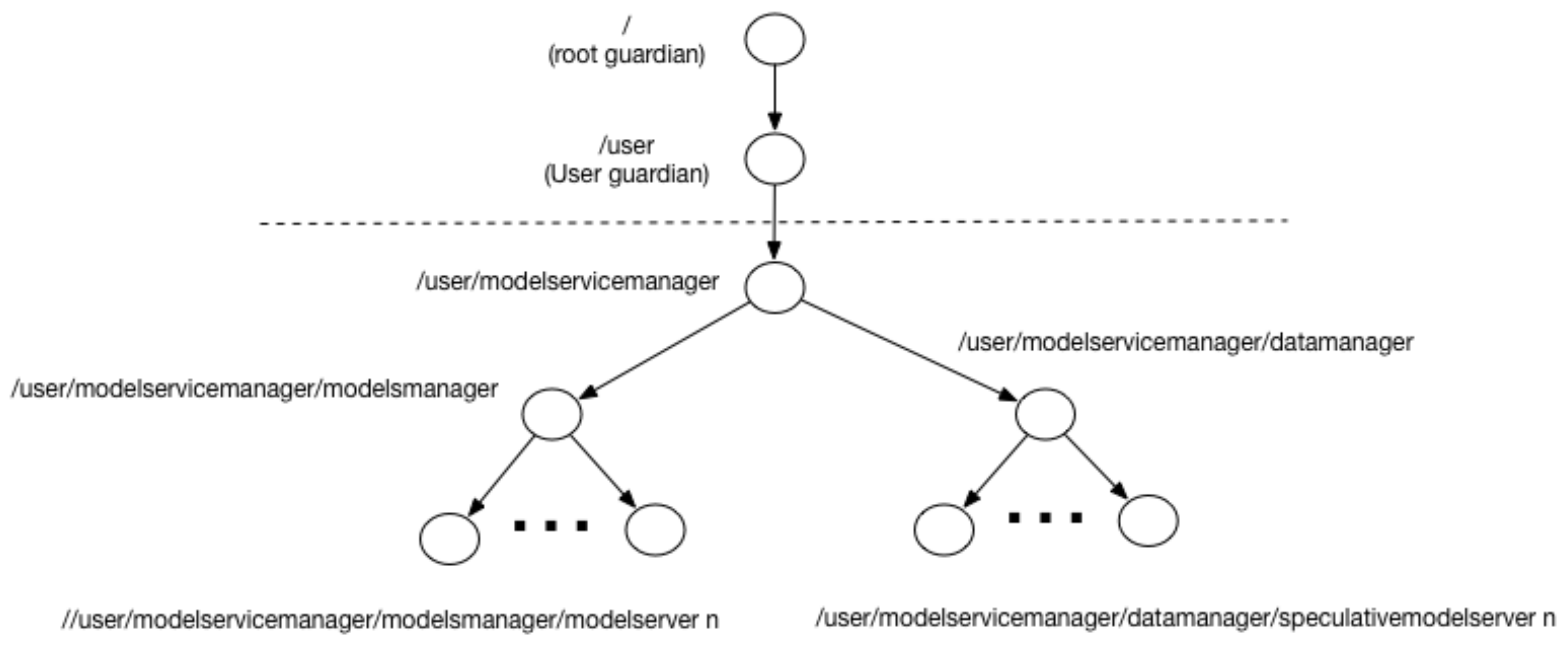
- Consensus-based model serving
  - *N* models (*N odd*)
  - Score with all of them and return the majority result
- Quality-based model serving
  - N models with the same quality metric
  - Pick the result with the best quality score for a given record
- Similarly for more sophisticated **boosting**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

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
  - 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

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