Building Kafka-based Microservices with Akka Streams and Kafka Streams

Boris Lublinsky and Dean Wampler, Lightbend

boris.lublinsky@lightbend.com
dean.wampler@lightbend.com

Lightbend

This Tutorial:

github.com/lightbend/
kafka-with-akka-streams-kafka-streams-tutorial

Either clone this or download the latest release

These slides are in the “presentation” folder

Let’s do introductions while you do this…

Outline

- Quick overview of streaming architectures
    - Kafka, Spark, Flink, Akka Streams, Kafka Streams
- Running example: Serving machine learning models
- Streaming in a microservice context
    - Akka Streams
    - Kafka Streams
- Wrap up

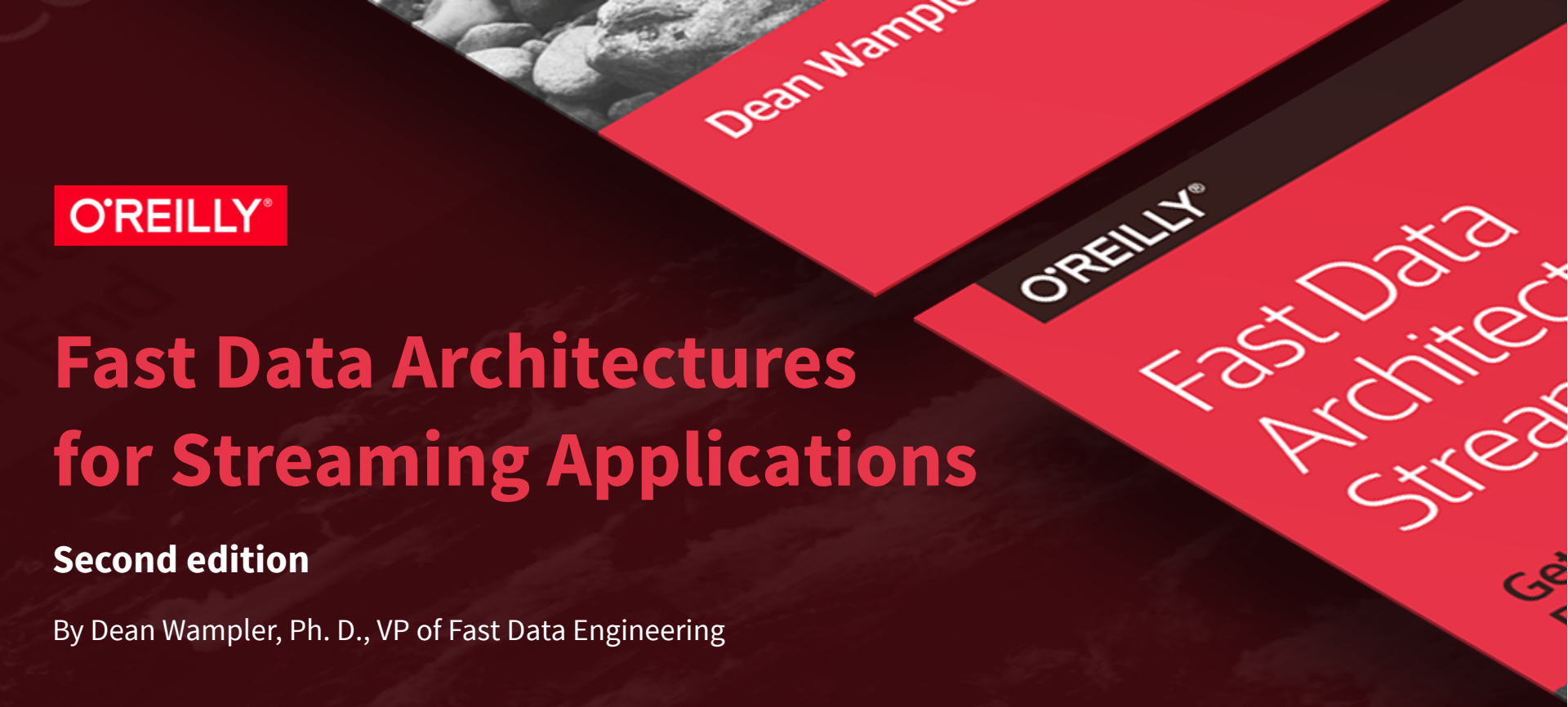
Overview of Streaming Technologies

Why Kafka, Spark, Flink, Akka Streams, and Kafka Streams?

Why Streaming?

“We live as streams, but we have a tendency to think in batch. Batch might be faster (simpler), but the reality is streams”

— Fabio Yamada, Kafka Mailing List



Fast Data Architectures for Streaming Applications

Second edition

By Dean Wampler, Ph. D., VP of Fast Data Engineering

lbnd.io/fast-data-ebook

Today's focus:

- Kafka - the data backplane
- Akka Streams and Kafka Streams - streaming microservices

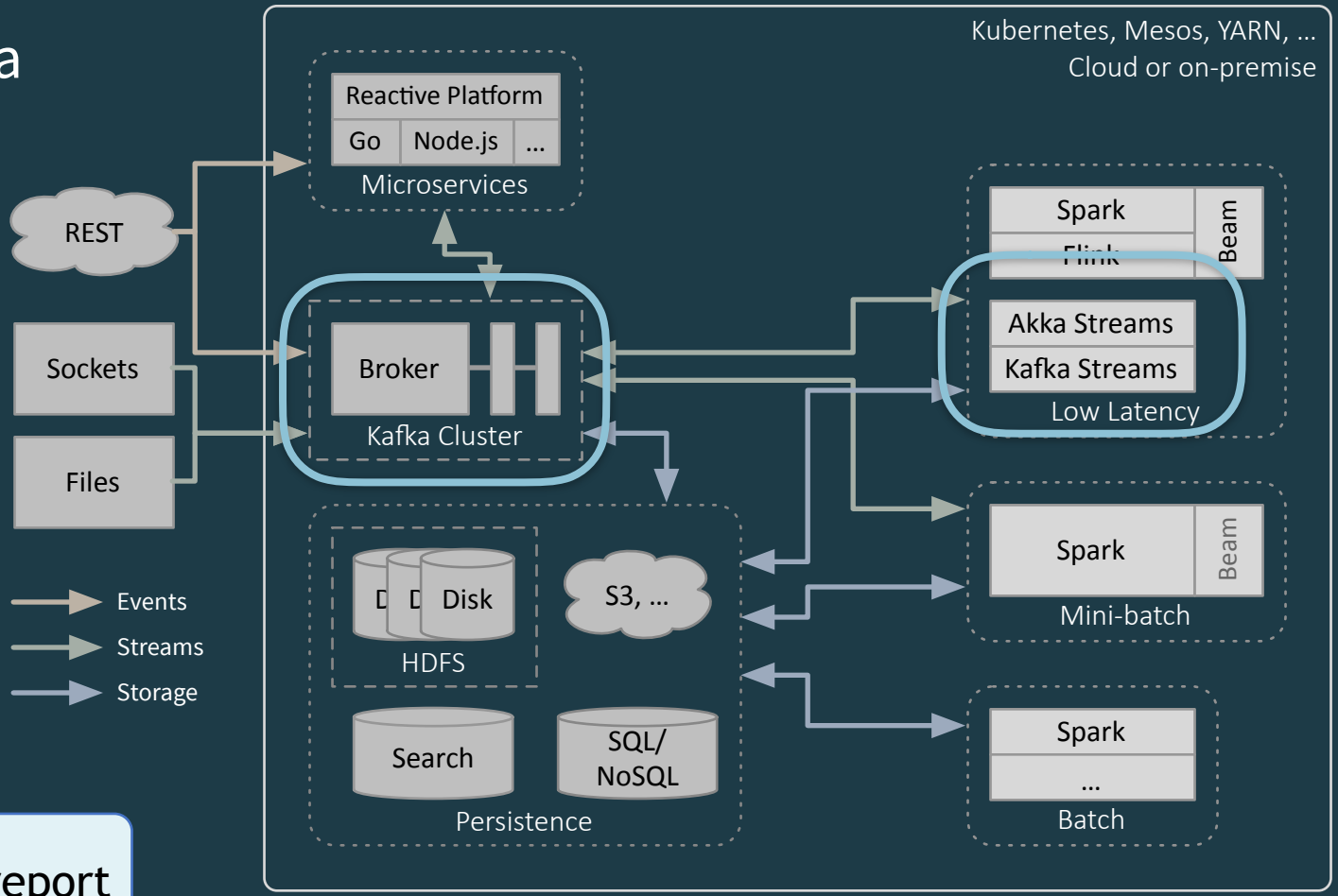
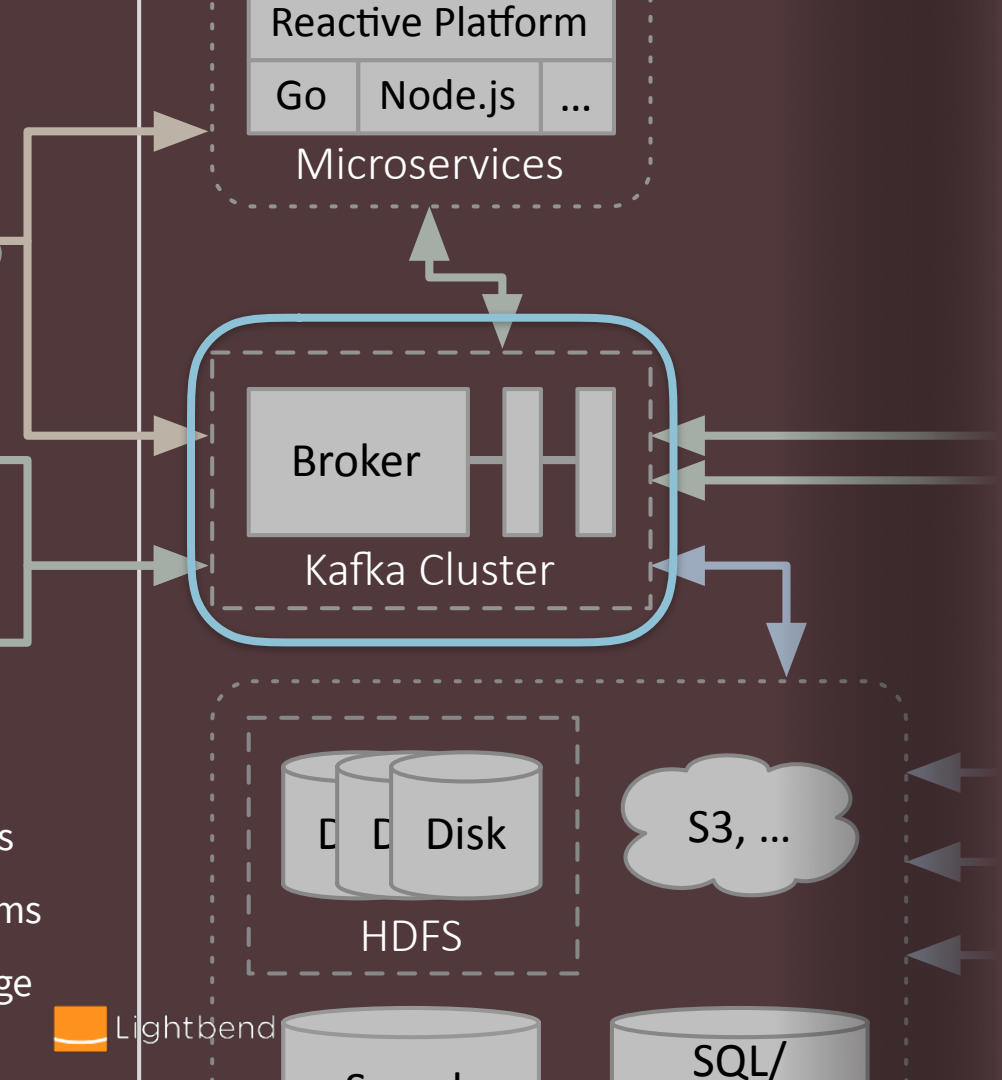
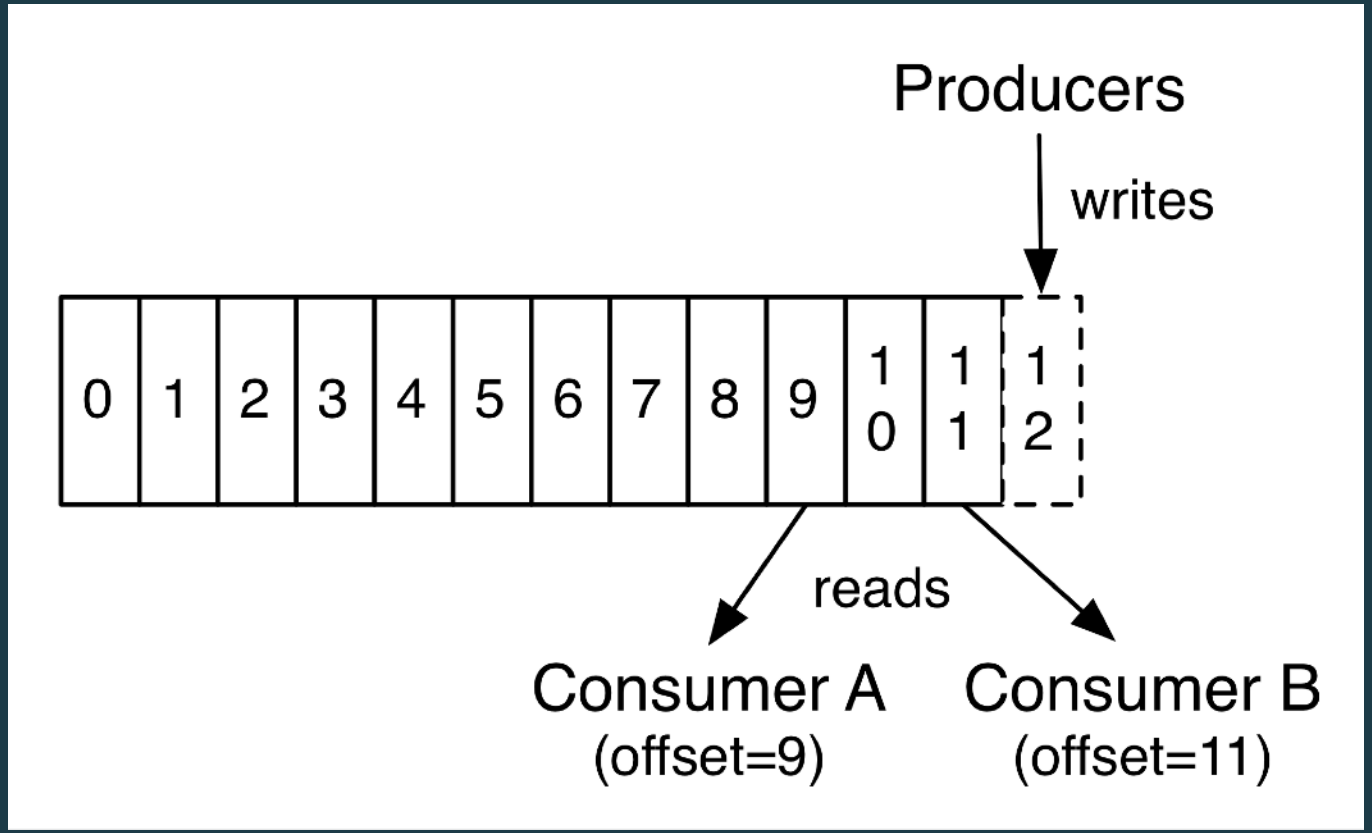
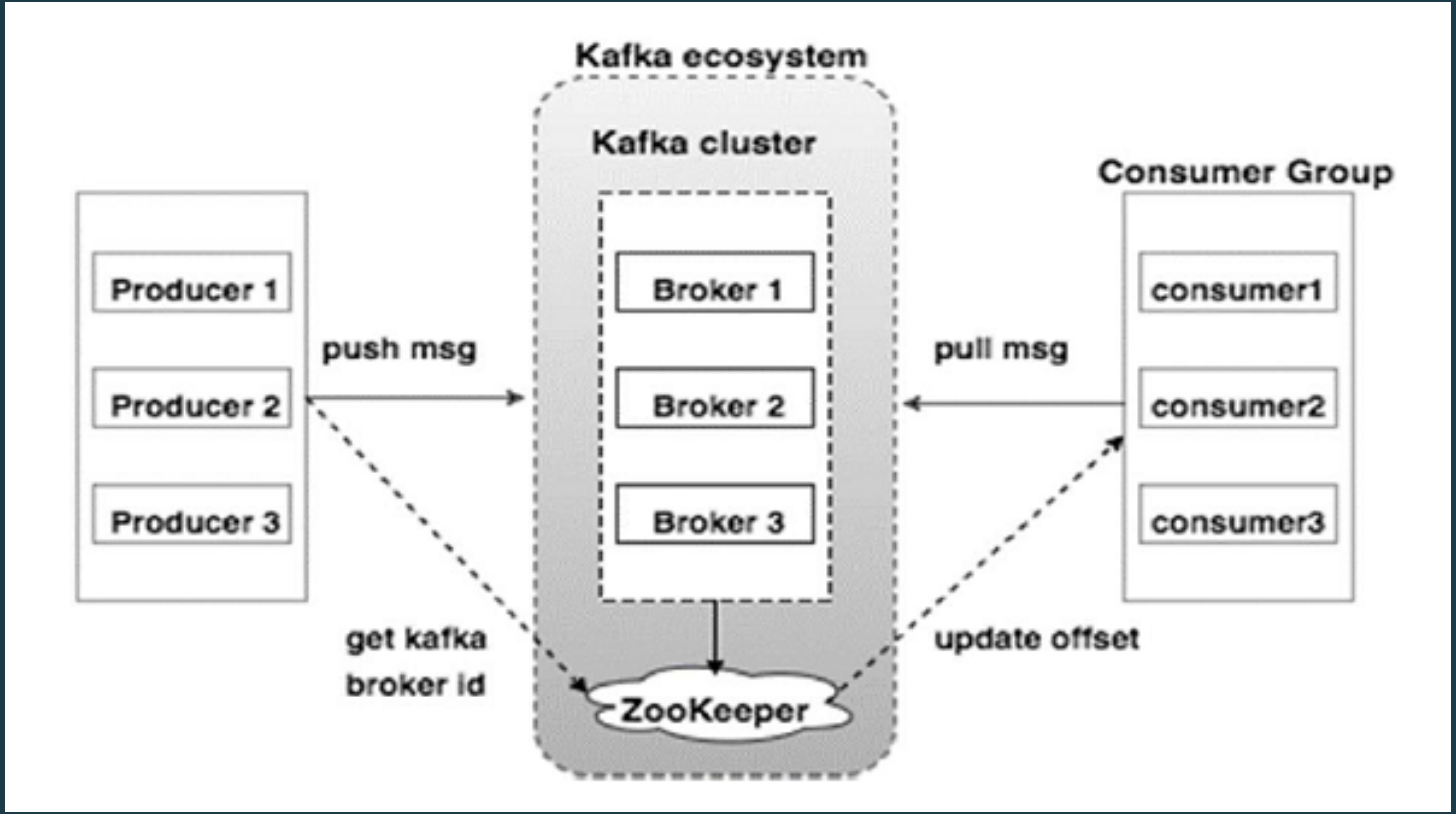


Diagram from the report

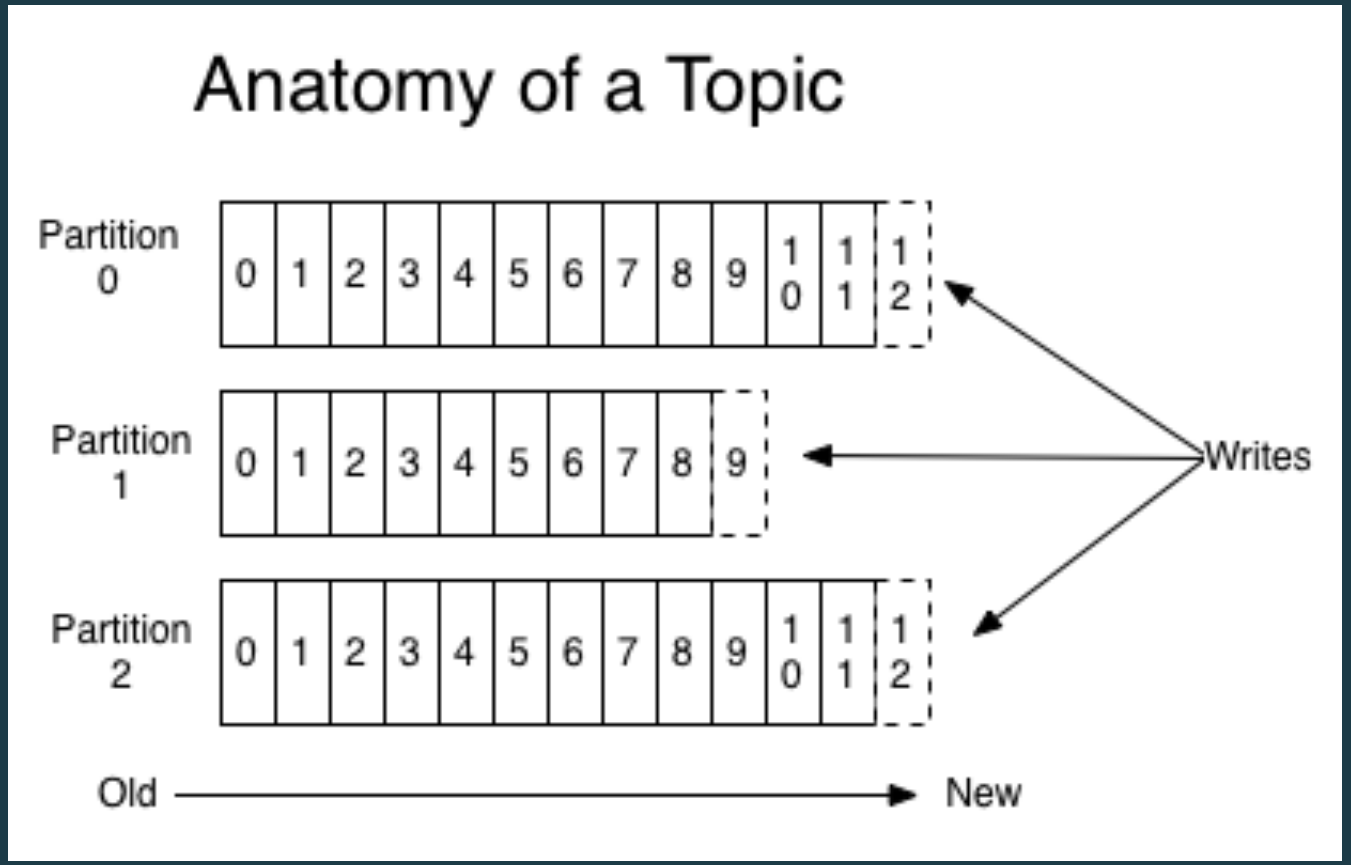
Why Kafka?


Reactive Platform
Go
Node.js
...
Microservices
Broker
Kafka Cluster
Disk
HDFS
S3, ...
SQL/
Lightbend

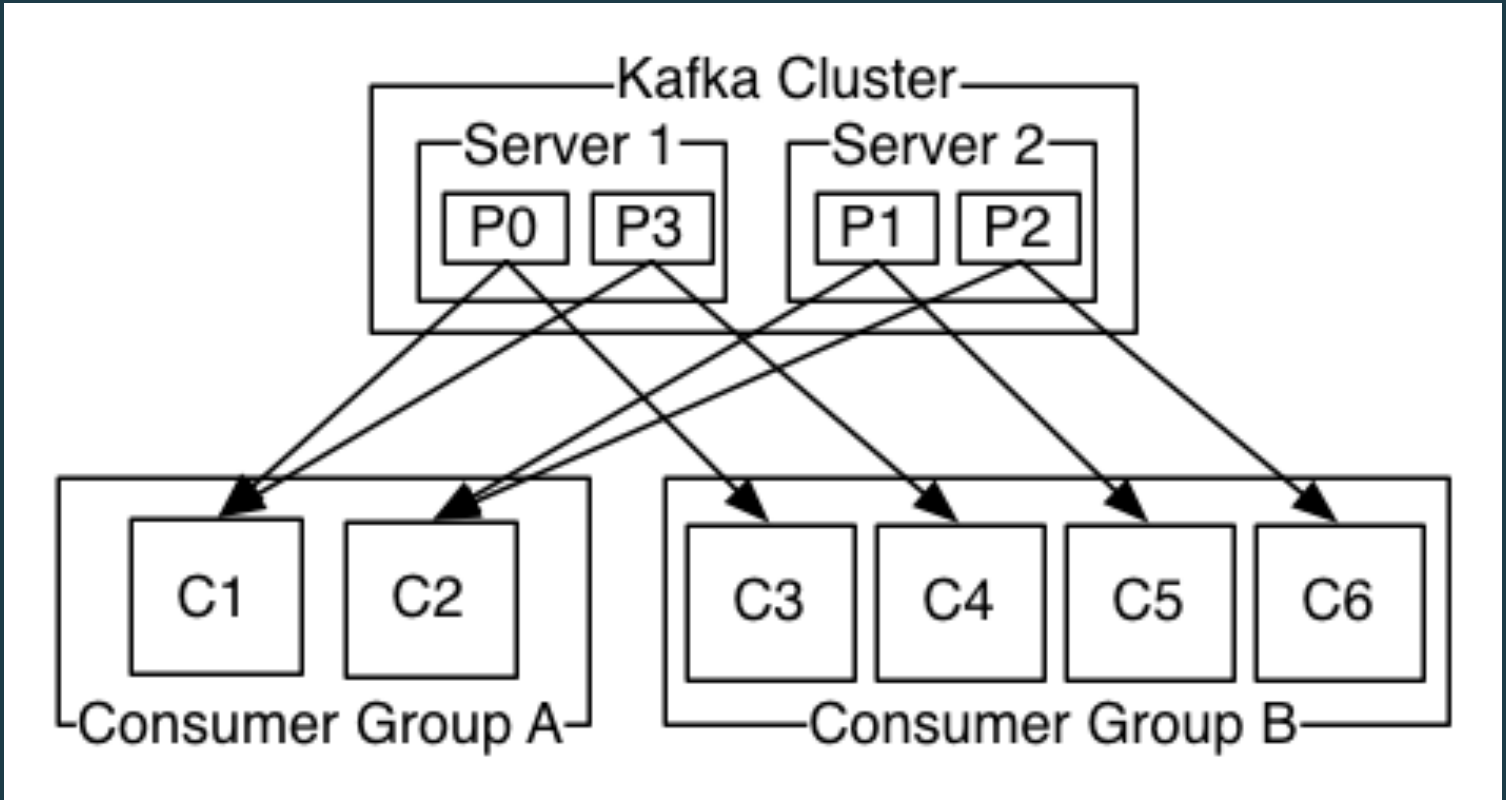

Producers
writes
0
1
2
3
4
5
6
7
8
9
10
11
12
reads
Consumer A
(offset=9)
Consumer B
(offset=11)


Kafka ecosystem
Kafka cluster
Consumer Group
Producer 1
Producer 2
Producer 3
push msg
Broker 1
Broker 2
Broker 3
pull msg
consumer1
consumer2
consumer3
get kafka
broker id
ZooKeeper
update offset

A Topic and Its Partitions



Consumer Groups

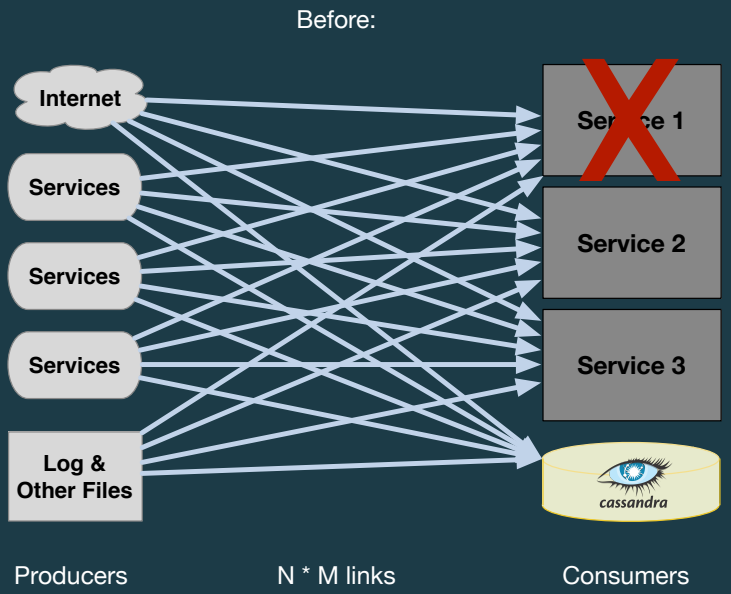


Kafka Producers and Consumers

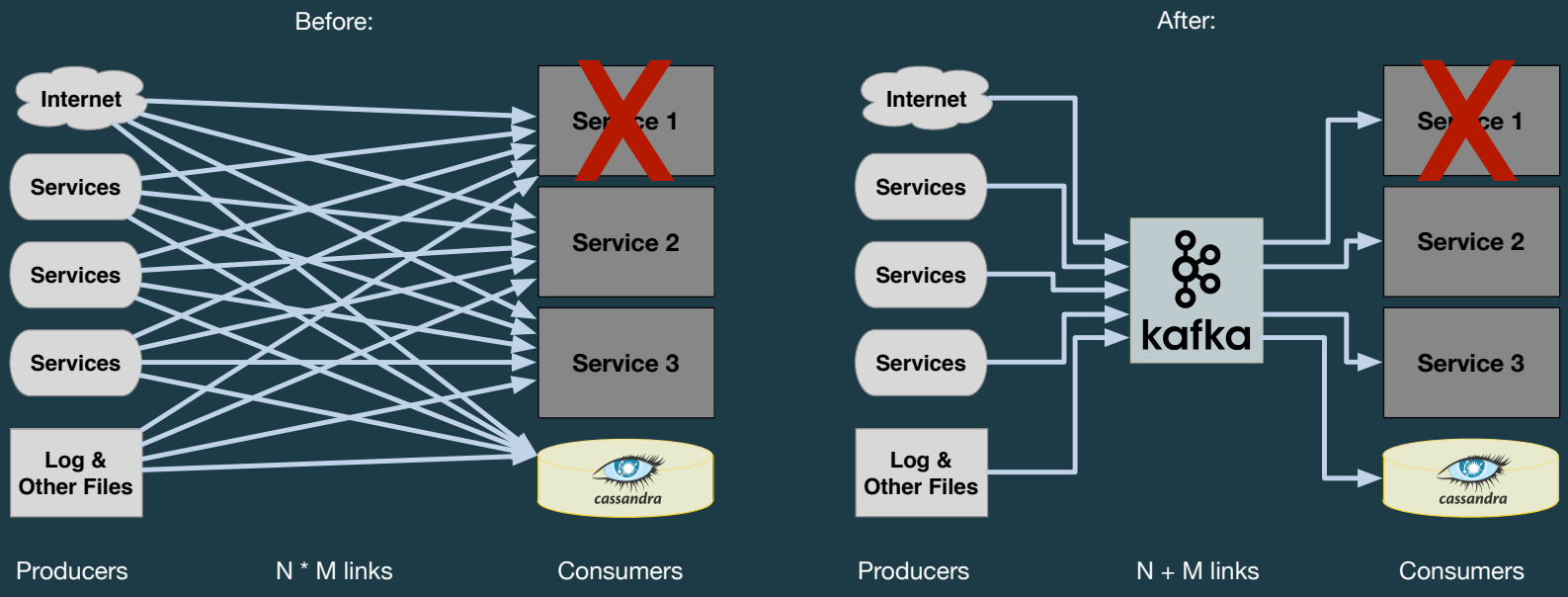
Code time

1. Project overview
   - Look at *client*, *data*, *model*, *configuration*, and *protobufs* folders and subprojects.
2. Explore and run the *client* project
   - Creates in-memory ("embedded") Kafka instance and our Kafka topics
   - Pumps data into them

Architecture Benefits of Kafka



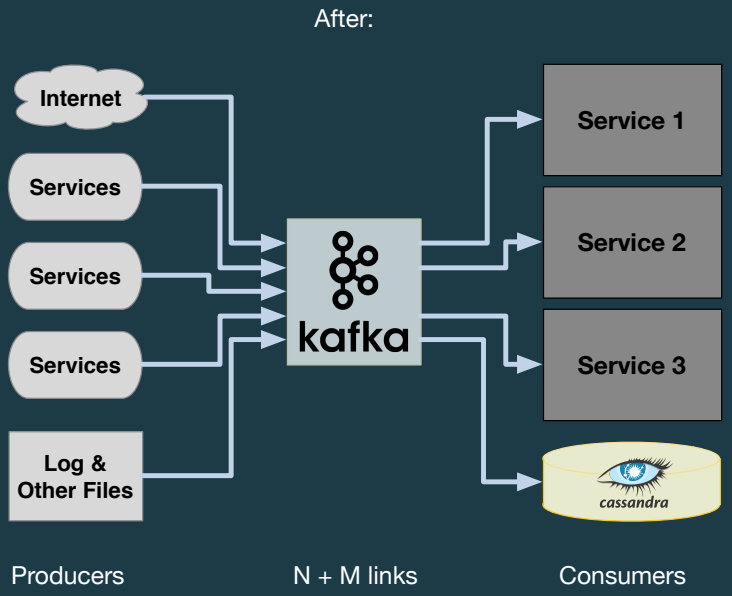
Architecture Benefits of Kafka



Architecture Benefits of Kafka

Kafka:

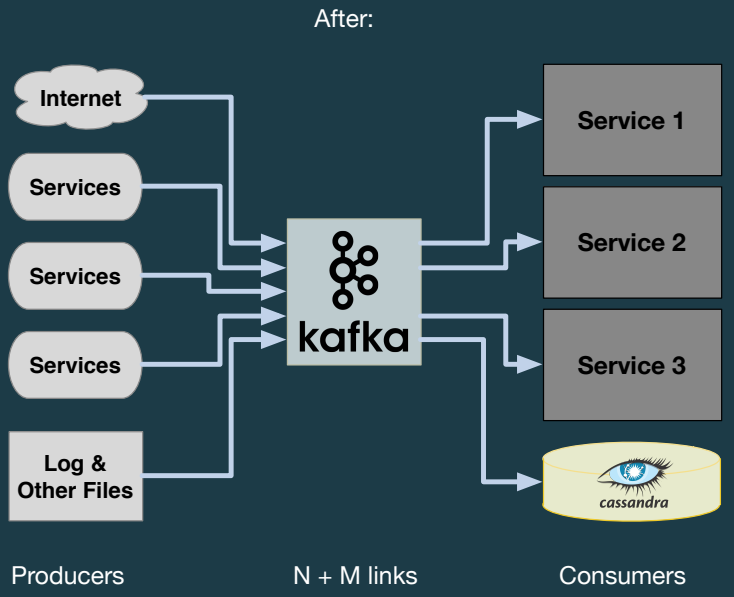
- Simplify dependencies between services
    - Improved data consistency
- Minimize data transmissions
- Reduce data loss when a service crashes



Architecture Benefits of Kafka

Kafka:

- M producers, N consumers
    - Improved extensibility
- Simplicity of one “API” for communication



Kafka message size considerations

- Should I use Kafka for all messages?
  - Optimal performance: messages size ~ few KB.
  - Larger messages put heavy load on brokers and is very inefficient.
  - It is inefficient on producers and consumers as well.

Kafka message size considerations

- What if my messages are very large?
  - Use *messaging by reference*
    - Store a message in S3, HDFS, etc.
    - Send the *reference* to the location as a Kafka message

Message compatibility for Kafka

- Is it okay if messages have different **schemas**?
  - If so, handled at run time (“dynamic typing”) or design time (“static typing”)?
- How is message type determined?
  - Registry or repository?
  - Embedding in Kafka headers?

Message versioning

- What happens if a Producer needs to create a new message version that’s incompatible with previous versions?
  - Topic versioning similar to endpoint versioning used by services.
  - Should you start new services instead?

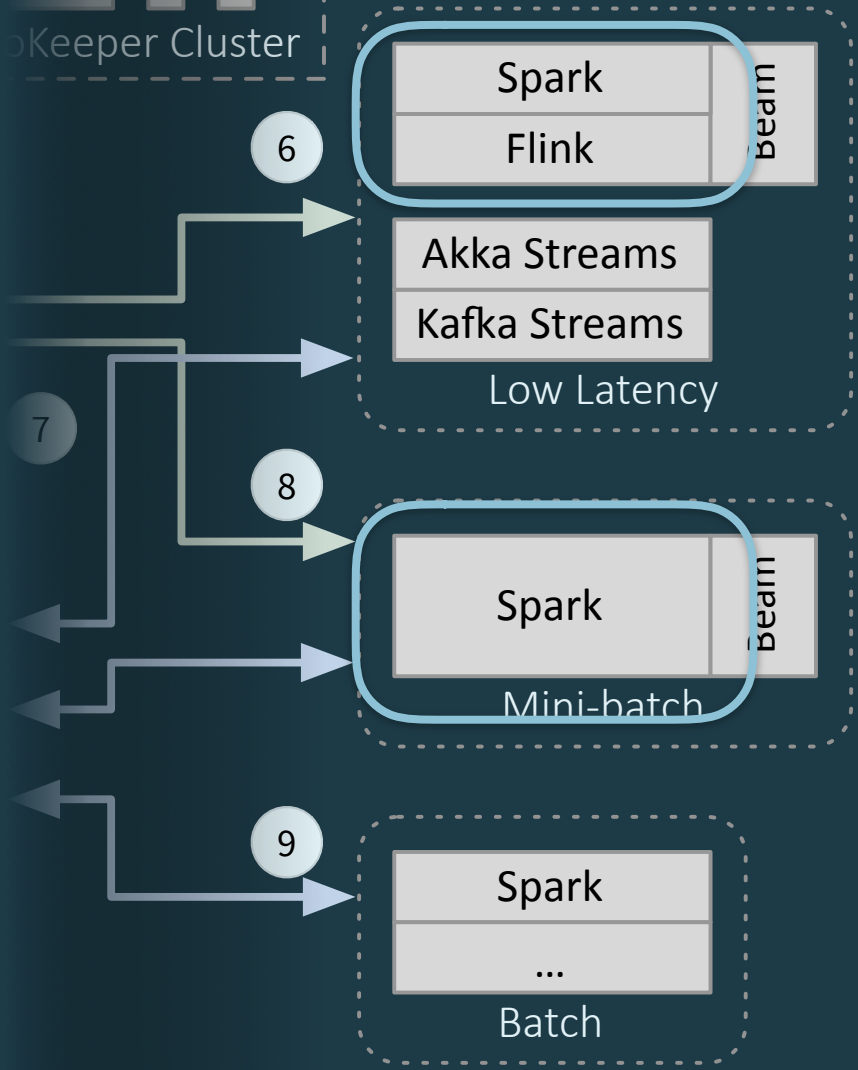
Streaming Architectures

Two approaches:

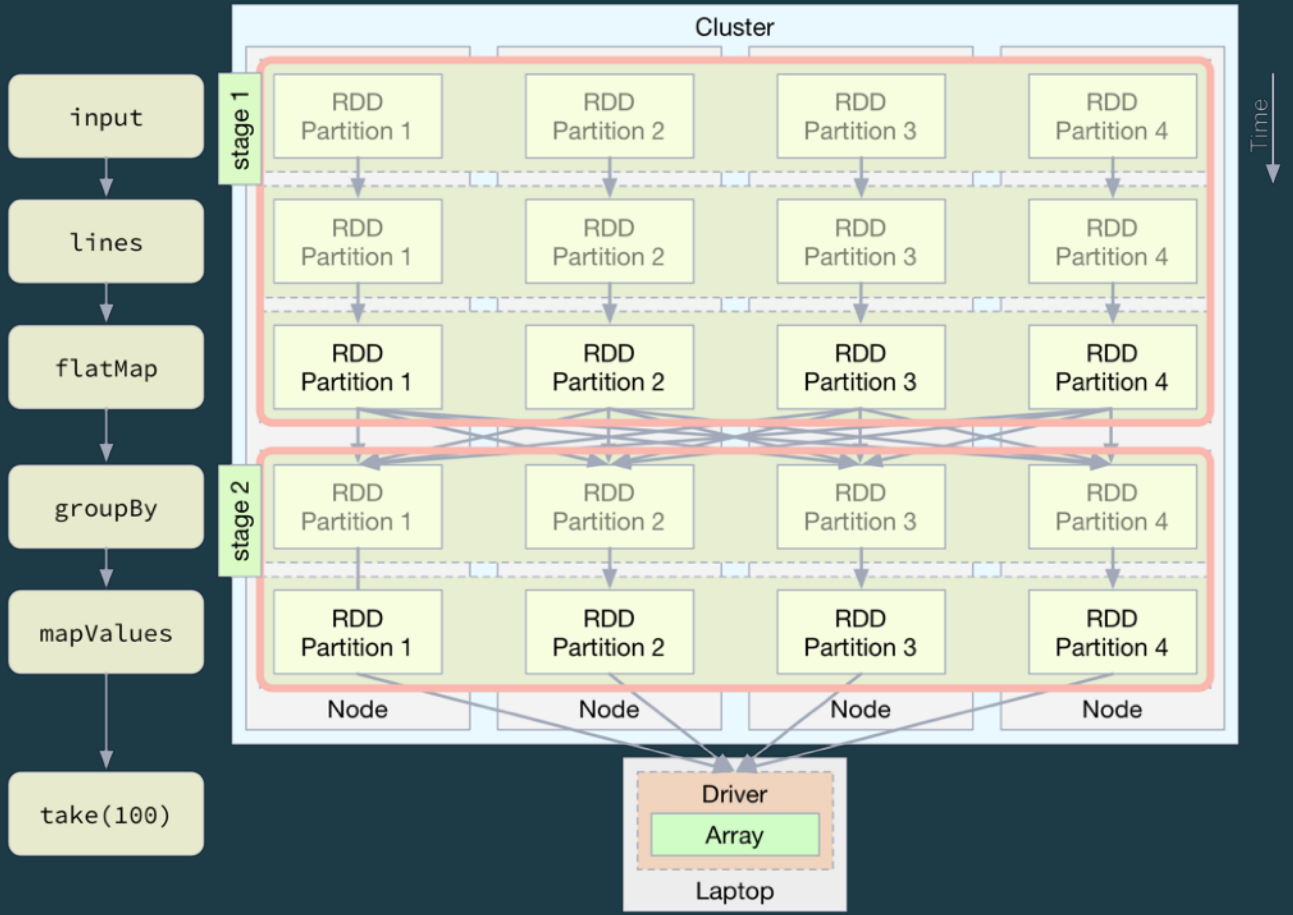
- Stream processing “engines”
  - Spark, Flink, Beam, …
- Streaming libraries
  - Akka Streams, Kafka Streams, …

Spark, Flink

- Service daemons:
  - You submit jobs
  - They partition into tasks
- Support very large-scale workloads, automatic data partitioning, task management.
- Rich libraries for SQL, ML, …

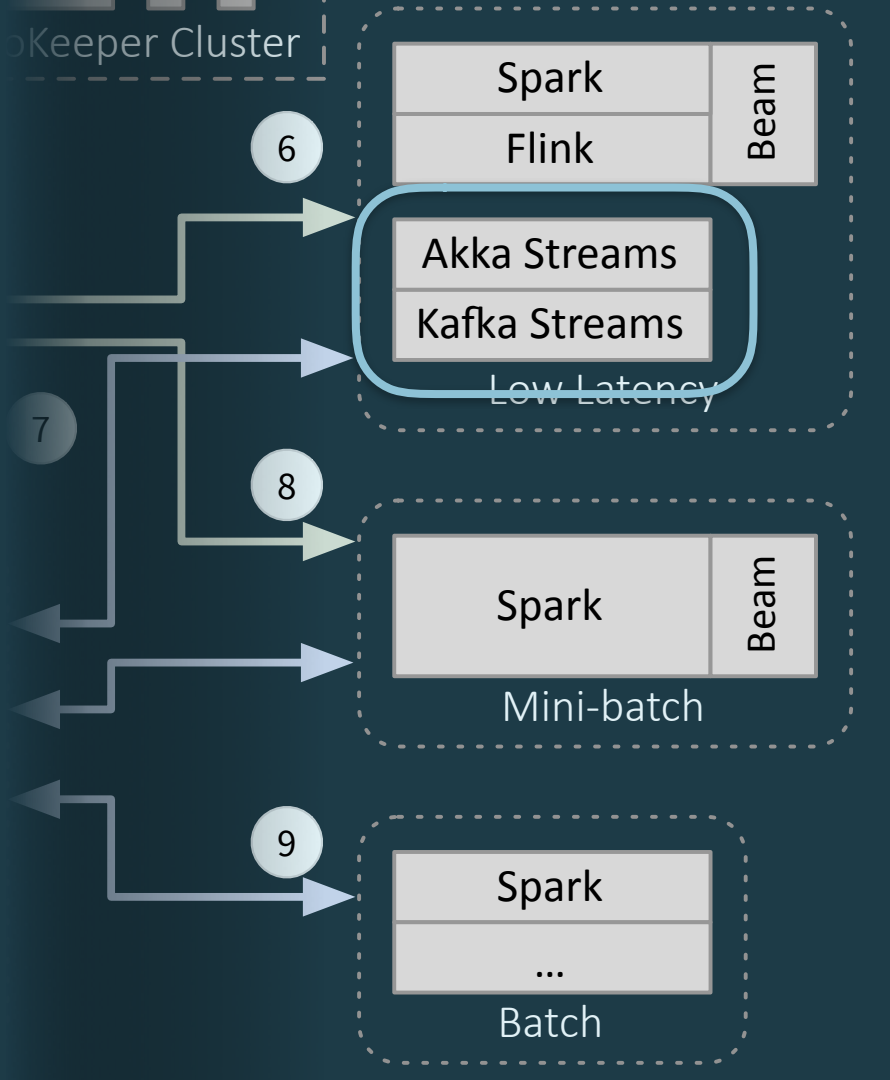


Example for Spark



Akka Streams, Kafka Streams

- libraries you embed in microservices
- No “for-free” scaling
- Greater flexibility, lower latency
- CI/CD is just like all your other microservices



Machine Learning and Model Serving: A Quick Introduction

Serving Machine Learning Models

A Guide to Architecture, Stream Processing Engines, and Frameworks

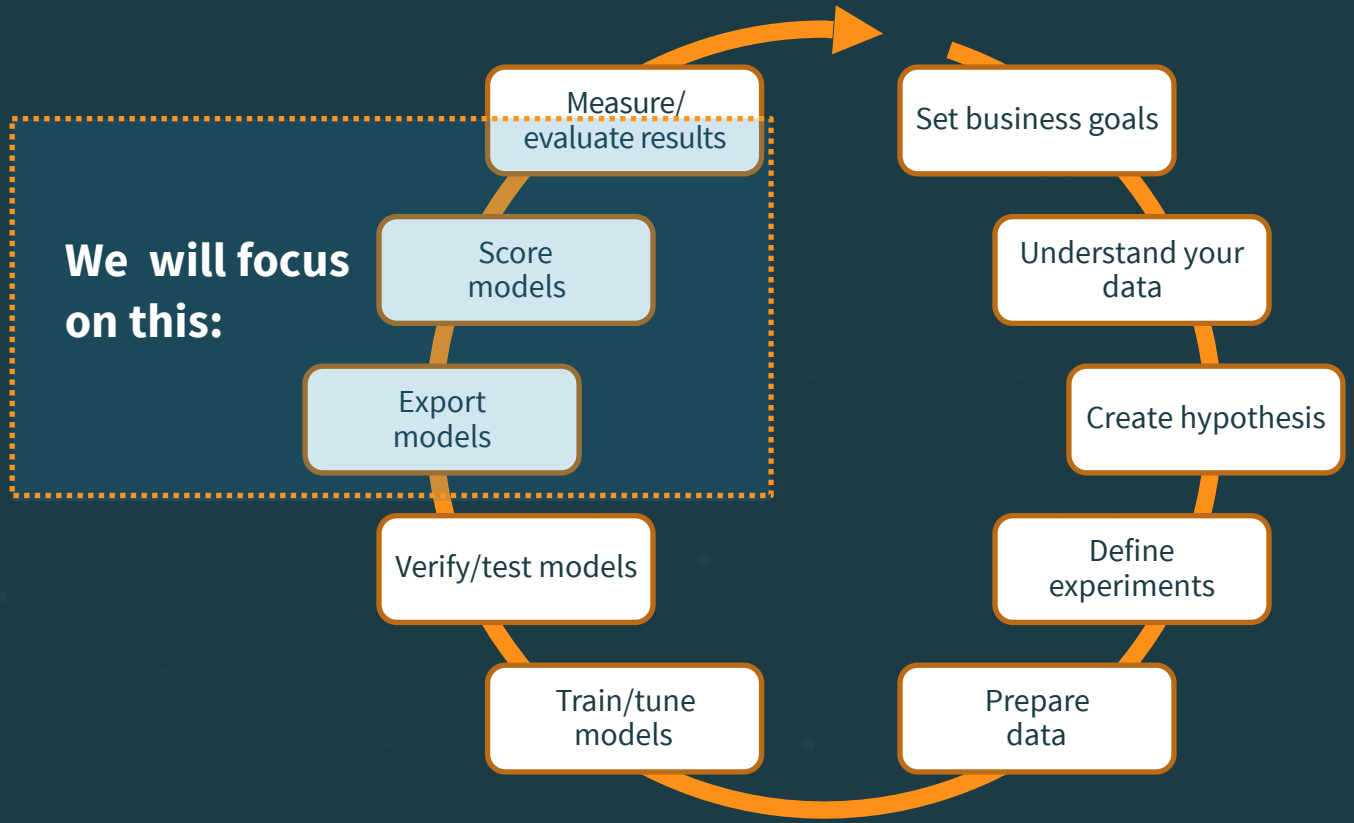
By Boris Lublinsky, Fast Data Platform Architect

lightbend.com/blog/serving-machine-learning-models-free-oreilly-ebook-from-lightbend

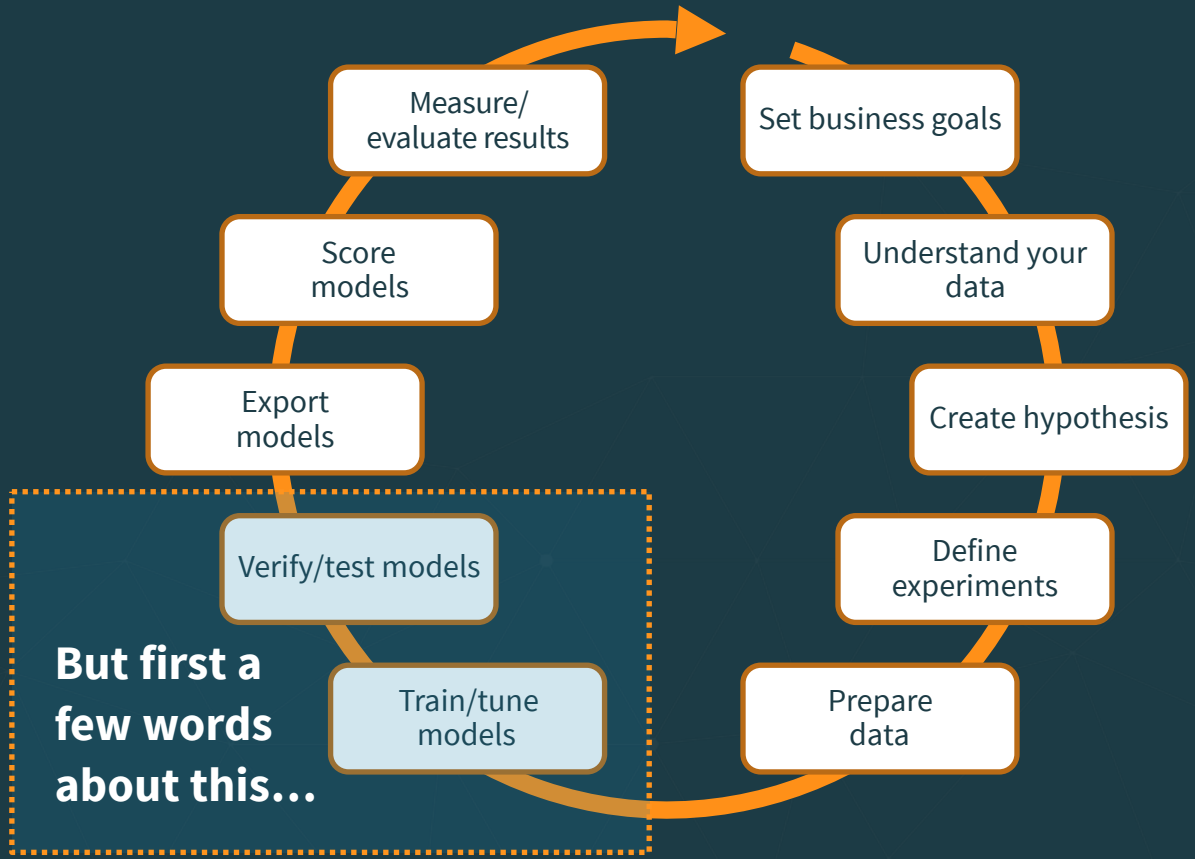
ML Is Simple

Maybe Not

The Reality

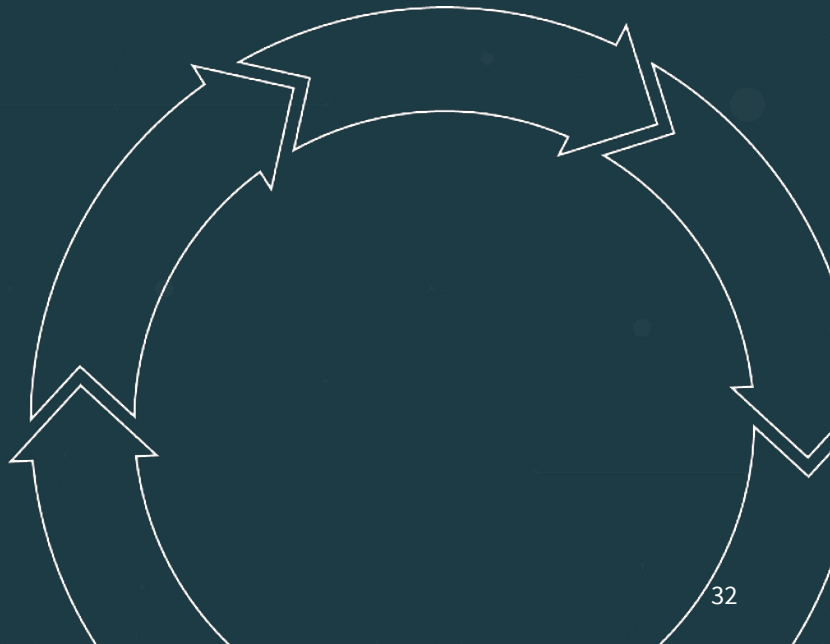

We will focus on this:
Measure/ evaluate results
Score models
Export models
Verify/test models
Train/tune models
Prepare data
Define experiments
Create hypothesis
Understand your data
Set business goals

The Reality


Measure/
evaluate results
Set business goals
Score
models
Understand your
data
Export
models
Create hypothesis
Verify/test models
Define
experiments
But first a
few words
about this…
Train/tune
models
Prepare
data

Another Consideration – Model Lifecycles

- Models tend to *drift*
  - More precisely, they grow stale as data changes
- Update frequencies vary – from hourly to yearly



Trends in Model Training…

- Original approach - Batch retrain at ad-hoc for fixed intervals.
  - Using Hadoop, off-line data science tool chains, etc.
- Challenges:
  - How do you deliver updated models to production?
  - What if you have *thousands* of models?
  - When should you retrain?

Trends in Model Training…

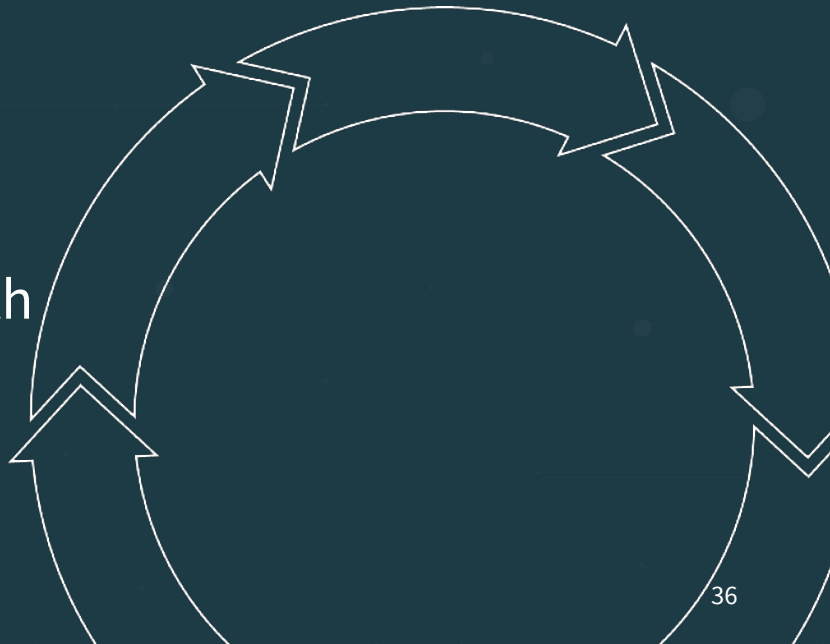
- Moving to automated retraining
  - Spark, X as a Service (where X = TensorFlow, …)
- Challenges:
  - How do you deliver updated models to production? CI/CD pipeline and techniques we'll discuss
  - What if you have *thousands* of models? *Must measure and automate everything*
  - When should you retrain? *Metrics for model drift… (next slide)*

Trends in Model Training…

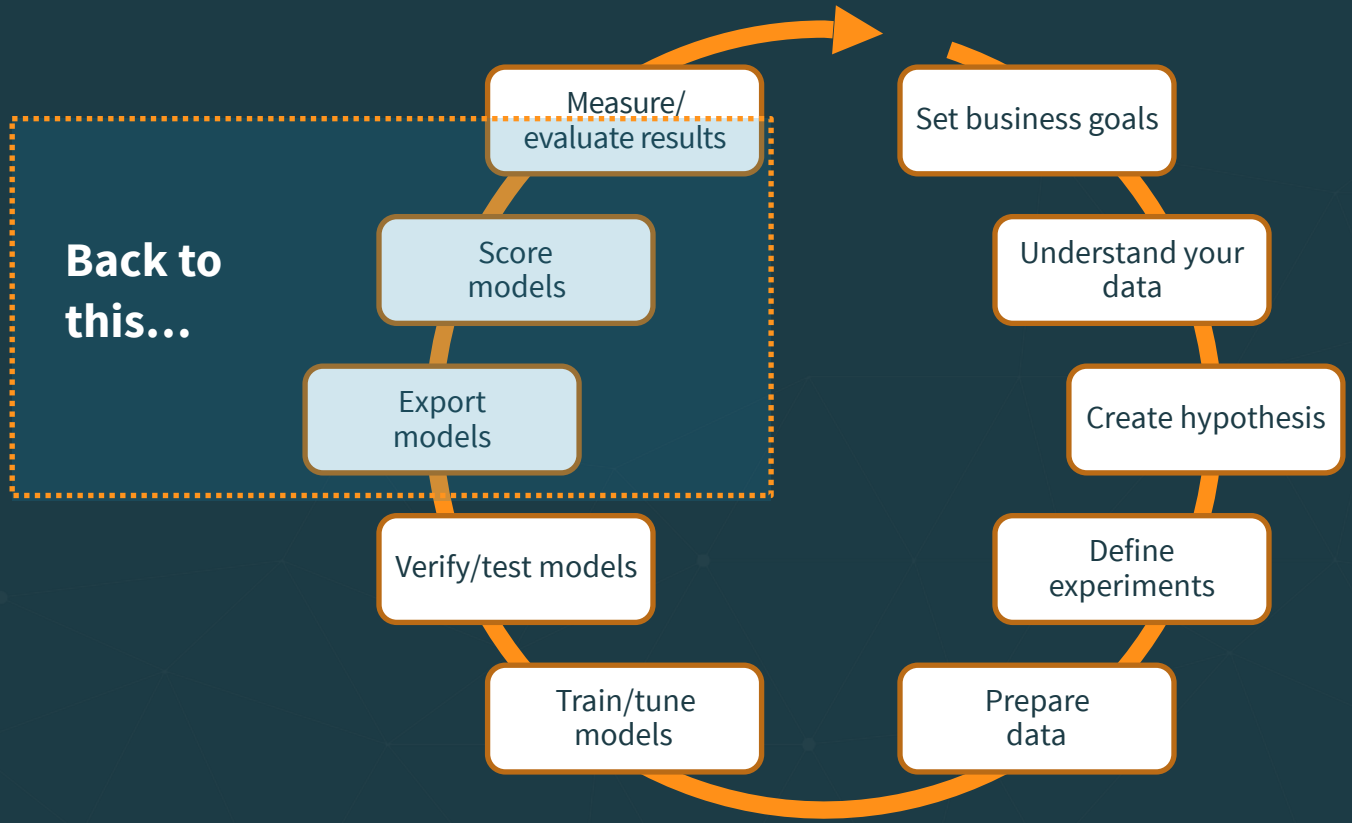
- One example of recent work
  - *Continuum: a platform for cost-aware low-latency continual learning*
    - https://blog.acolyer.org/2018/11/21/continuum-a-platform-for-cost-aware-low-latency-continual-learning/
    - Attempts to balance cost concerns, preserve low latency, yet remain sensitive to degradation from drift

Another Consideration – Model Auditing

- Your organization may need to track model versions for auditing:
  - What model instance was used to score this (controversial?) record?
  - Why did it score it a particular way?
  - What were the model parameters?
  - Which other records were scored with this model?



The Reality


Back to this…
Measure/ evaluate results
Score models
Export models
Verify/test models
Train/tune models
Set business goals
Understand your data
Create hypothesis
Define experiments
Prepare data

What Is The Model?

A model is a *function* transforming inputs to outputs - $y = f(x)$

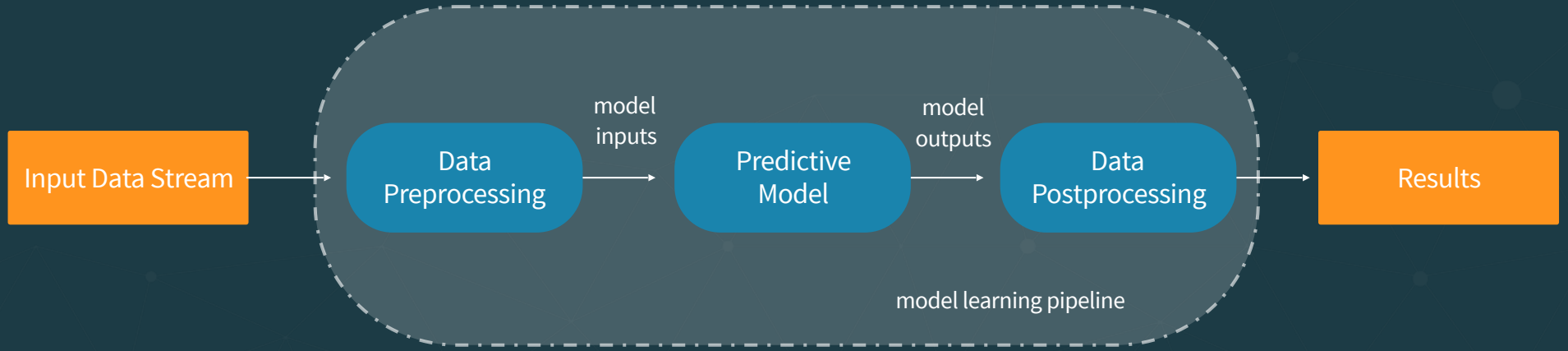
Linear regression: $y = a_c + a_1 * x1 + \ldots + a_n * x_n$

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

From the implementation point of view, it is just *function composition*.

Model Learning Pipeline

UC Berkeley AMPLab introduced machine learning pipelines as a graph defining the complete chain of data transformation.



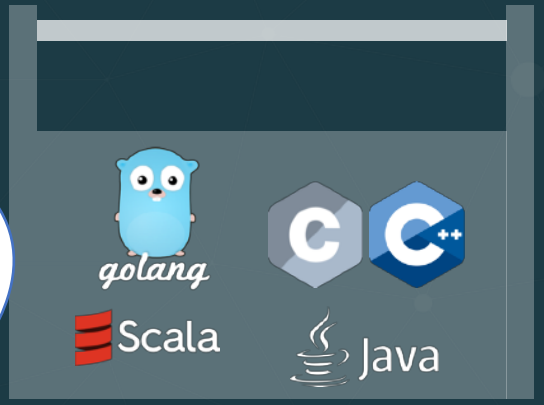
Traditional Approach to Model Serving

- *Model as code*: source code in some language implements the model
- This code is linked into the model serving applications

Why is this problematic?

Impedance Mismatch

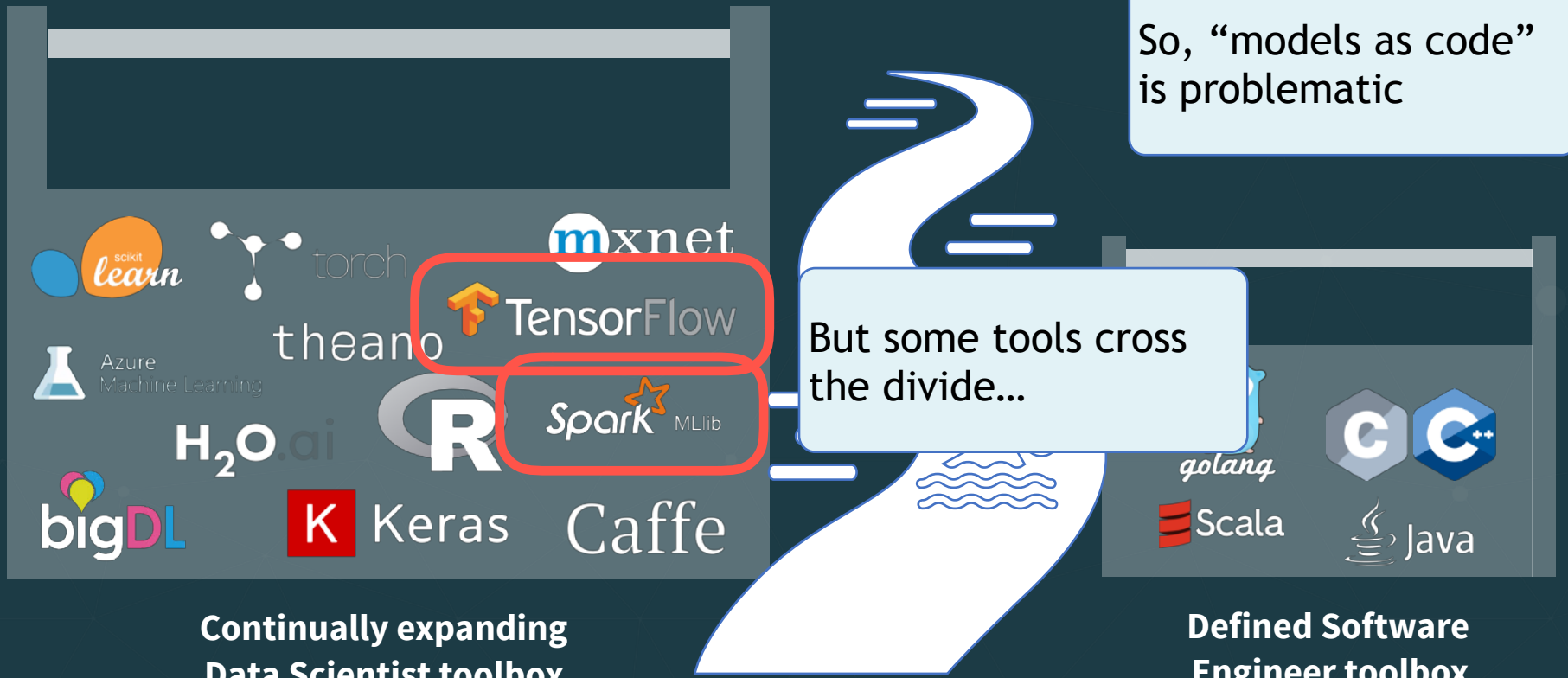
So, “models as code” is problematic



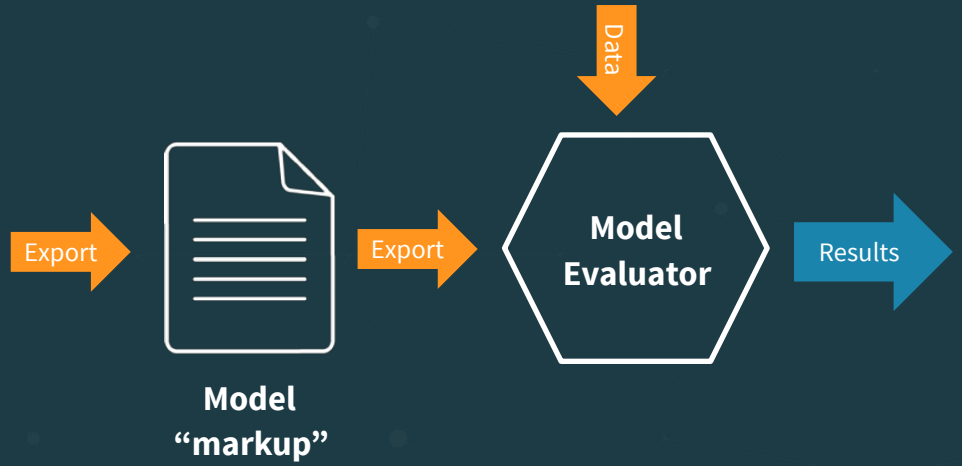
Continually expanding Data Scientist toolbox

Defined Software Engineer toolbox

Impedance Mismatch



Alternative - Model As Data



Standards:

PMML Predictive Model Markup Language

Portable Format for Analytics **(PFA)**

ONNX

TensorFlow

Considerations for Interchange Tools

- Do you *training* tools support an exchange format (PMML, PFA, …)?
- Do you *serving* tools support the format?
- Is there support on both ends for the model types you want to use, e.g., random forests, neural networks, …?
- Does the *serving* implementation faithfully reproduce the results of your *training* environment?

Exporting Models As Data With PMML

There are already a lot of export options

Spark MLlib https://github.com/jpmml/jpmml-sparkml

scikit learn https://github.com/jpmml/jpmml-sklearn

R https://github.com/jpmml/jpmml-r

TensorFlow https://github.com/jpmml/jpmml-tensorflow

Evaluating PMML Model

There are also a few PMML evaluators

https://github.com/jpmml/jpmml-evaluator

https://github.com/opendatagroup/augustus

See also this list of products: http://dmg.org/pmml/products.html

Exporting NN Models With ONNX

- Created by Microsoft and Facebook
  - https://thenewstack.io/facebook-microsoft-bring-interoperable-models-machine-learning-toolkits/
- Supported tools: https://onnx.ai/supported-tools
- Git Repo: https://github.com/onnx

- Designed for when you use one NN framework for R&D, another for production, but…
  - It is new and not mature
  - Tensorflow is supported, but not by Google

Exporting Model As Data With Tensorflow

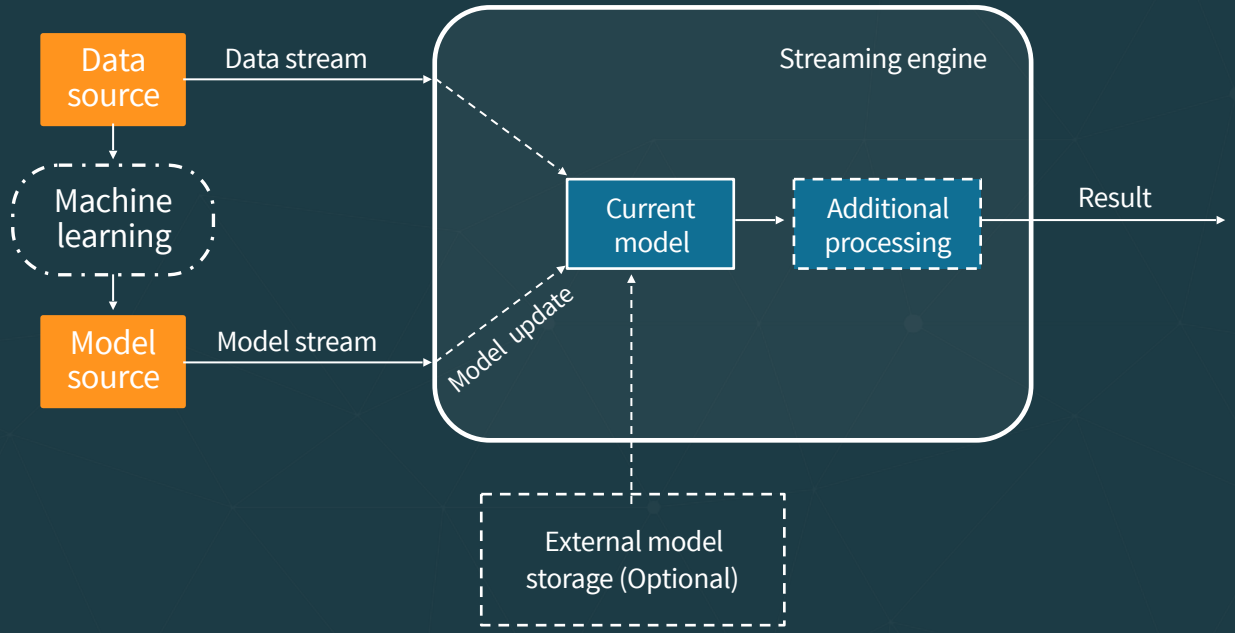
- Tensorflow execution is based on Tensors and Graphs
- Tensors are defined as multilinear functions which consist of various vector variables
- A computational graph is a series of Tensorflow operations arranged into graph of nodes
- Tensorflow supports exporting graphs in the form of binary protocol buffers
- There are two different export format - optimized graph and a new format - saved model

Evaluating Tensorflow Model

- Tensorflow is implemented in C++ with a Python interface.
- In order to simplify Tensorflow usage from Java, in 2017 Google introduced the Tensorflow Java API.
  - Supports importing an exported model and using it for scoring.
  - Not yet recommended for training TF models.

The Solution We'll Discuss Today

A streaming system that allows updating models without interruption of execution (dynamically controlled stream).



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;
  };
  ModelType modeltype = 4;
  oneof MessageContent {
     // Byte array containing the model
     bytes data = 5;
     string location = 6;
  }
}
```

Model Code Abstraction (Scala and Java)

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

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

```java
public interface Model extends Serializable {
  public Object score(Object input);
  Unit cleanup();
  byte[] toBytes();
  long getType();
}

public interface ModelFactory {
  Optional<Model> create(ModelDescriptor d);
  Model restore(byte[] bytes);
}
```

Side Note: Monitoring

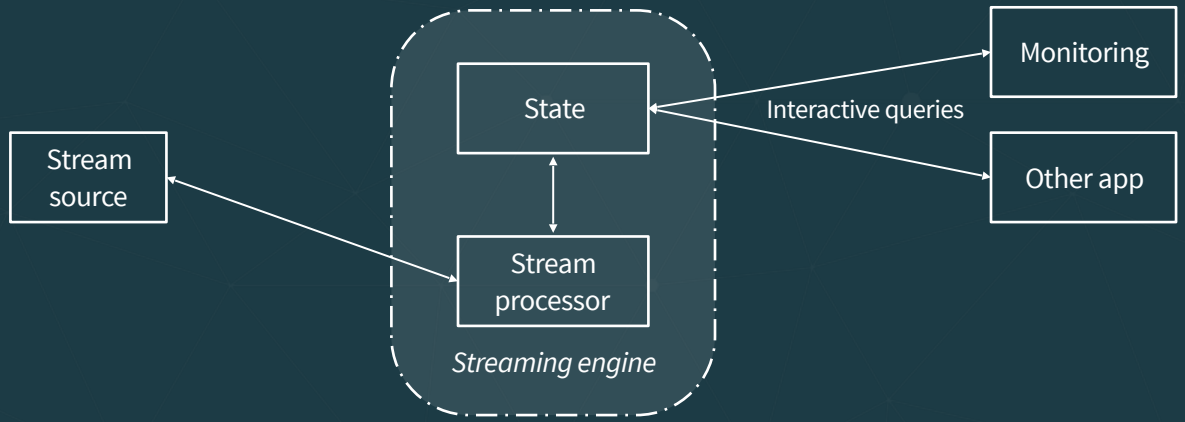
Model monitoring should provide information about usage, behavior, performance and lifecycle of the deployed models

```
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
)
```

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.

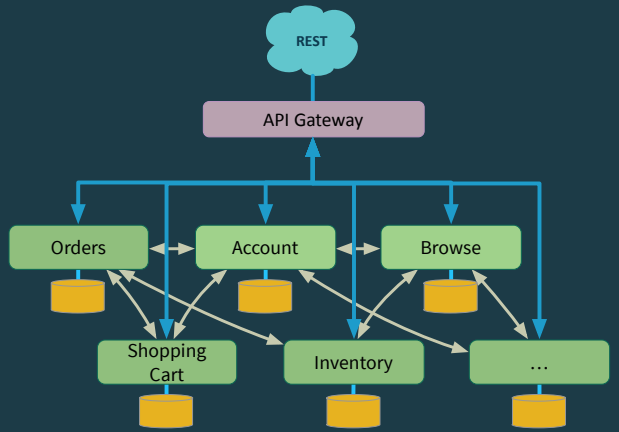


Microservices for Streaming Data

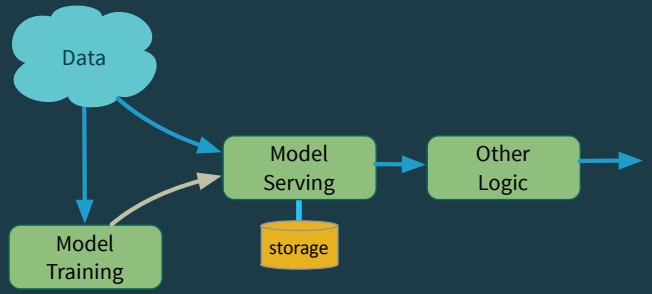
Akka Streams vs. Kafka Streams

A Spectrum of Microservices

Event-driven μ-services



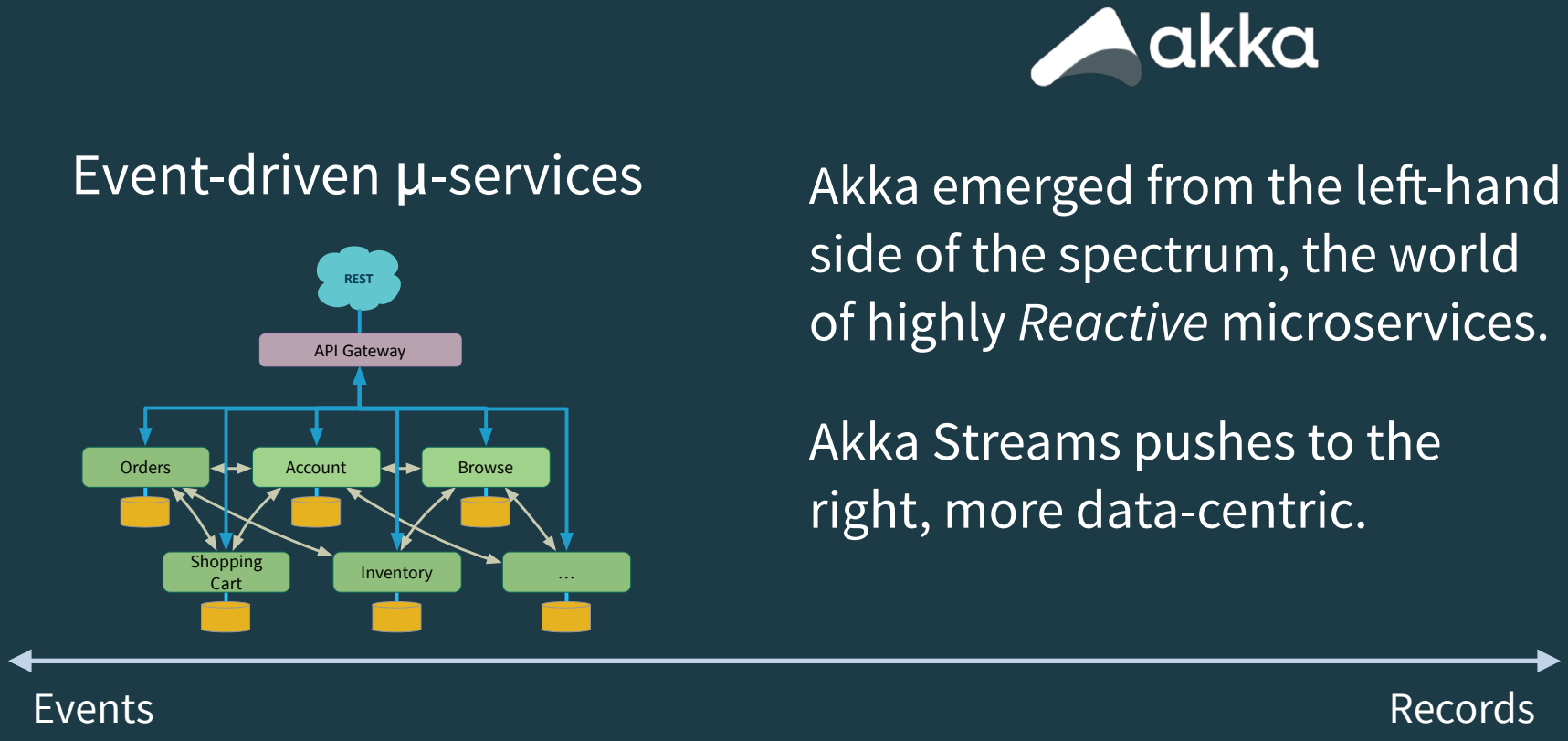
“Record-centric” μ-services



Events

Records

A Spectrum of Microservices

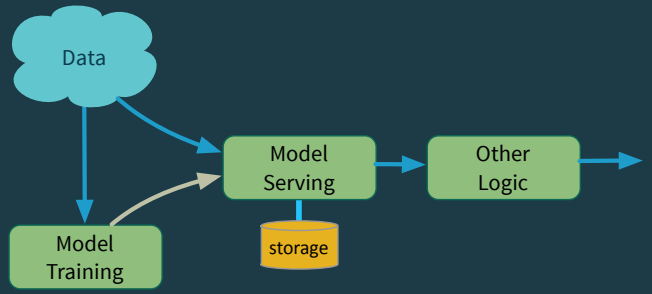


A Spectrum of Microservices

Emerged from the right-hand side.

Kafka Streams pushes to the left, supporting many event-processing scenarios.

“Record-centric” μ-services

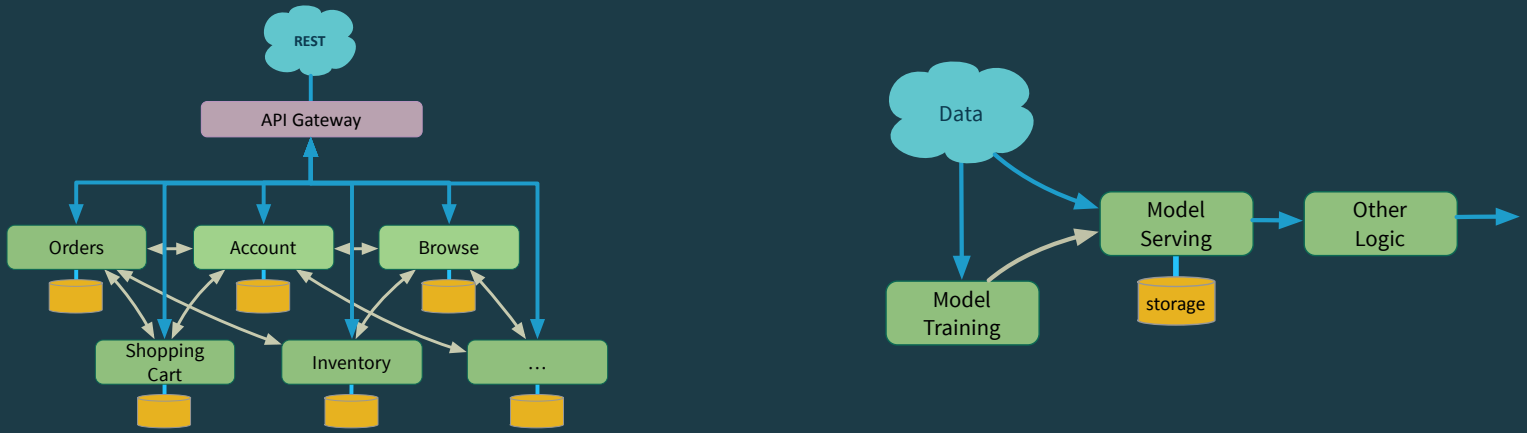


Events

Records

A Spectrum of Microservices

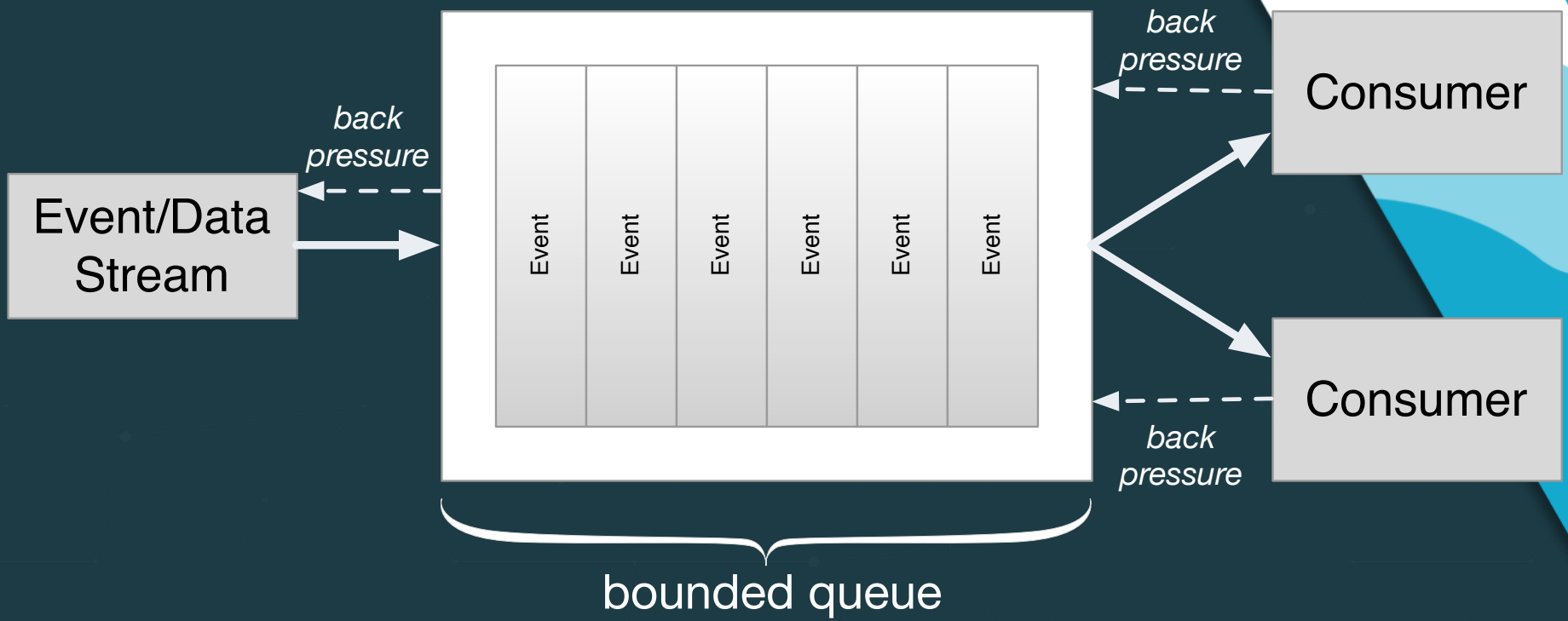
There is no dividing line. Complex-processing events can also be analyzed as data…



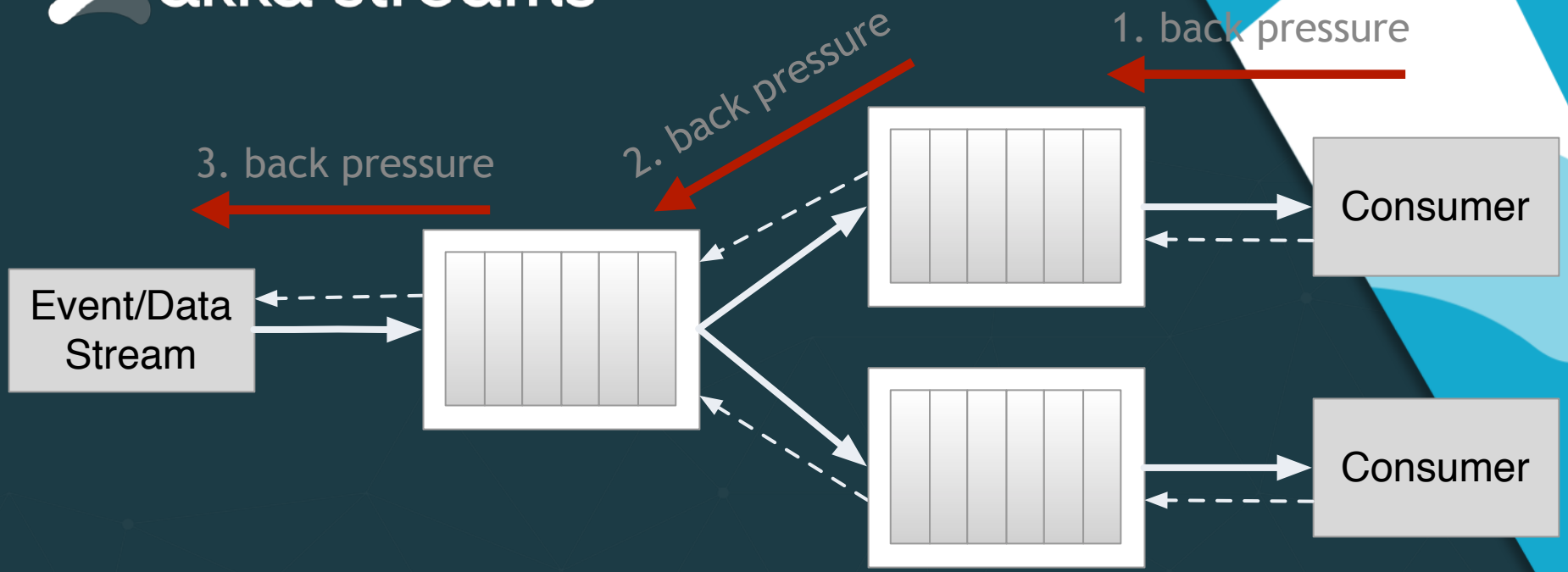
Akka Streams

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

akka streams



akka streams



… and they compose

- 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

- 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
```

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

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

val source: Source[Int, NotUsed] = Source(1 to 10)
val factorials = source.scan(BigInt(1)) ( (acc, next) => acc * next )
factorials.runWith(Sink.foreach(println))
```

Initialize and specify now the stream is “materialized”

```
1
2
6
5040
40320
362880
3628800
```

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

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

val source: Source[Int, NotUsed] = Source(1 to 10)
val factorials = source.scan(BigInt(1))((acc, next) => acc * next)
factorials.runWithSink.foreach(println))
```

```
1
2
6
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.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

```
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
3628800
```

Output to a Sink, and run it

```
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 so[illegible]e[Int, No[illegible]urce(1 to
val fa[illegible]urce.[illegible]a[illegible](acc[illegible]e[illegible]next)
factor[illegible]n(Sink.fo[illegible]ln))
```

Source → Flow → Sink

A source, flow, and sink constitute a graph

```
1
2
6
24
40320
362880
3628800
```

- This example is included in the project:
  - akkaStreamsModelServer/simple-akka-streams-example.sc
- To run it (showing the different prompt!):

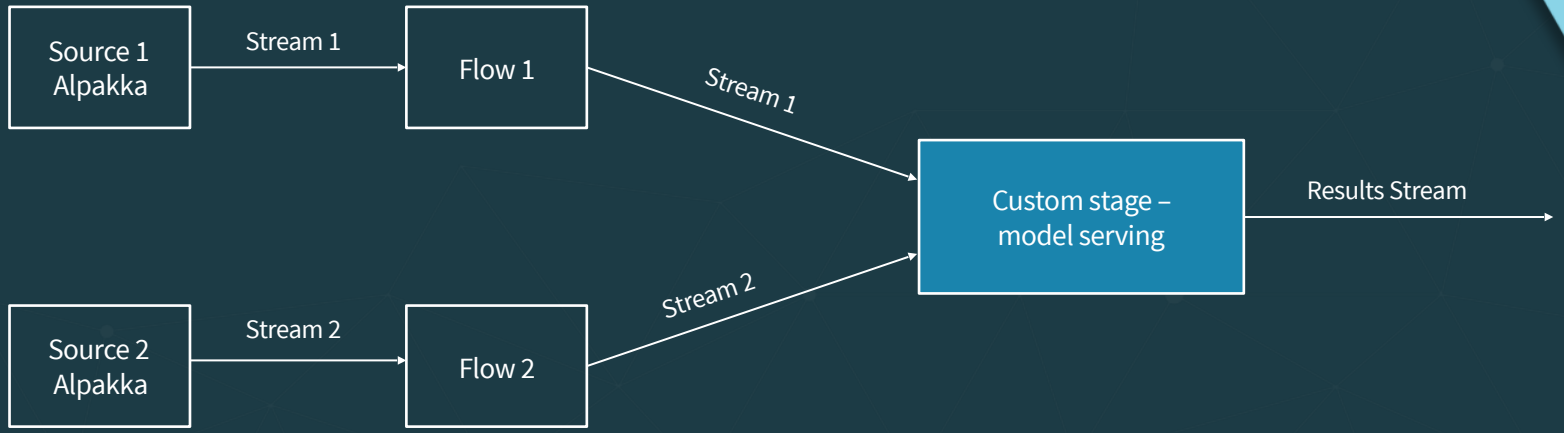
```
$ sbt
sbt:akkaKafkaTutorial> project akkaStreamsModelServer
sbt:akkaStreamsModelServer> console
scala> :load akkaStreamsModelServer/simple-akka-streams-example.sc
```

Implementations

- How do we integrate model serving (or any other new capability) into an Akka Streams app? We'll look at two approaches:
  - Implement a *Custom Stage*. Once implemented, you use it like any other "step" in the Akka Streams app.
  - Make asynchronous calls to Akka Actors to do anything you want. . .
- Third approach: Make REST calls to a separate service, e.g., Tensor Flow Serving and Clipper

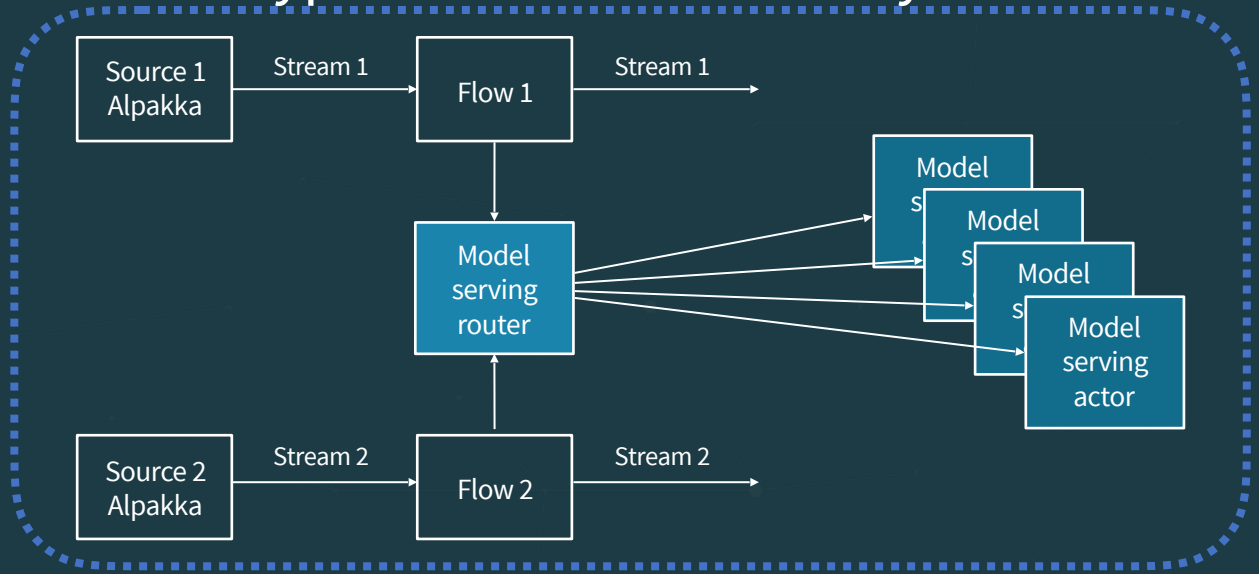
Using a Custom Stage

Create a custom stage, a fully type-safe way to encapsulate new functionality. Like adding a new "operator" to Akka Streams.



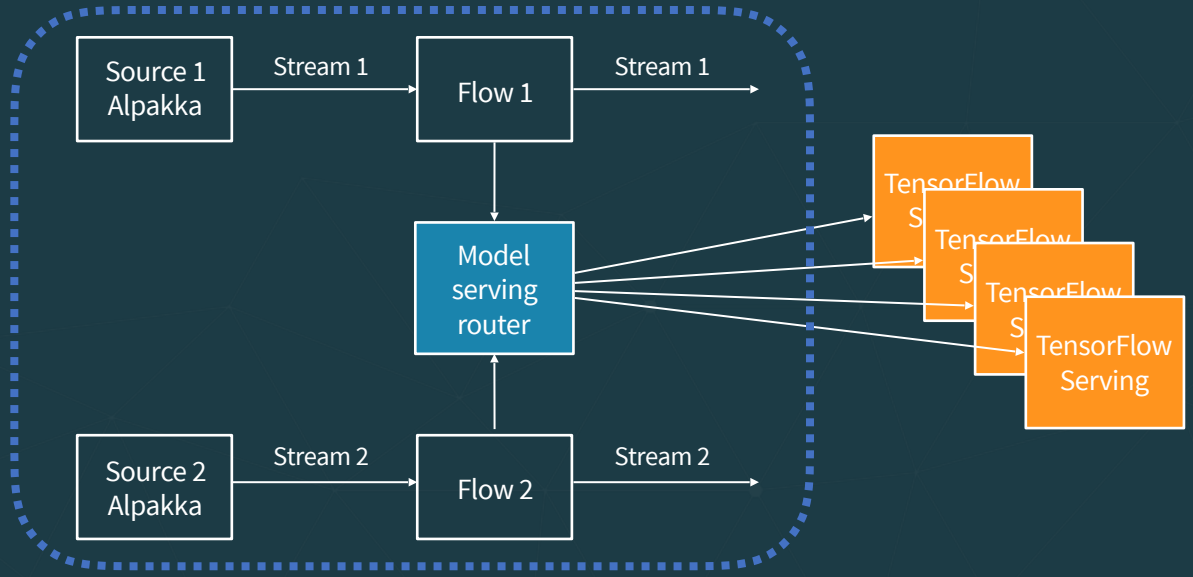
Using Invocations of Akka Actors

Use a router actor to forward requests to the actor(s) responsible for processing requests for a specific model type. Clone for scalability!!



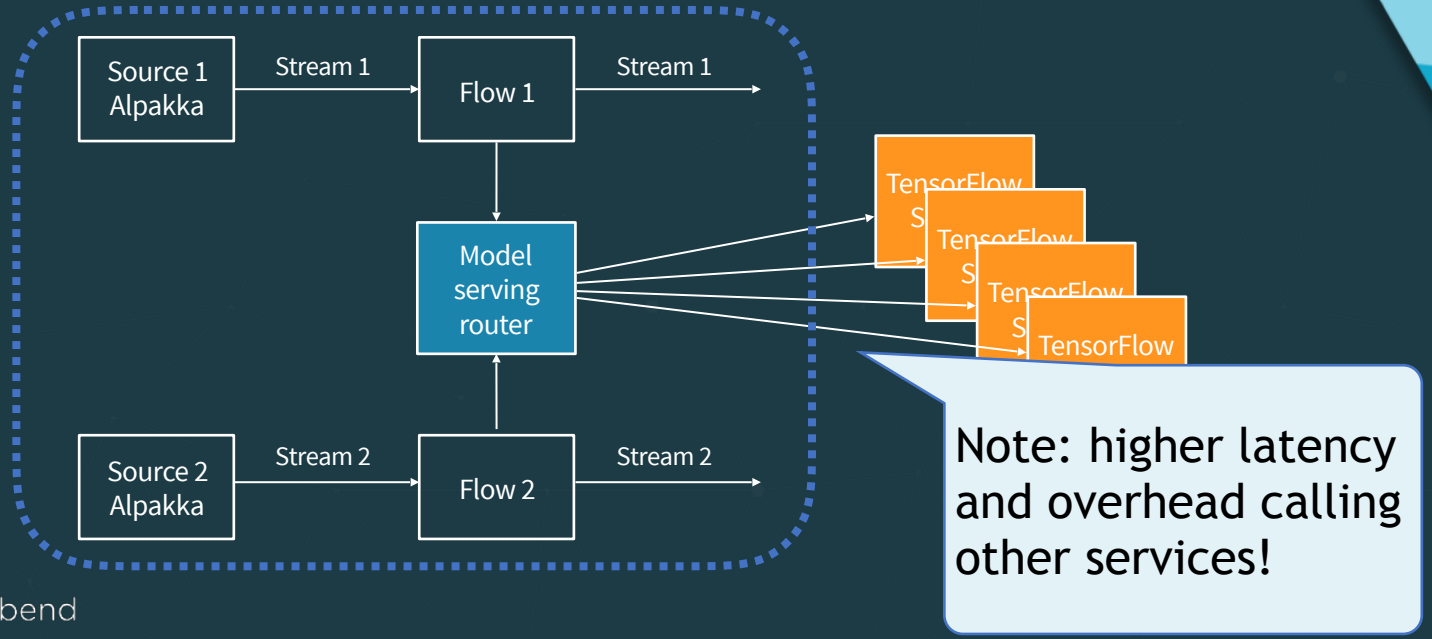
Using Invocations of Other Services

Use the same router actor idiom to forward requests to external services.



Using Invocations of Other Services

Use the same router actor idiom to forward requests to external services.



Akka Streams Example

Code time

1. Run the *client* project (if not already running)
2. Explore and run *akkaStreamsModelServer* project
    1. Use the `c` or `custom` (or default) command-line argument for the *custom stage*
    2. Use the `a` or `actor` command-line argument for the *actor model server*
    3. Use `–h` or `––help` for help

Akka Streams Example

Check Queryable state

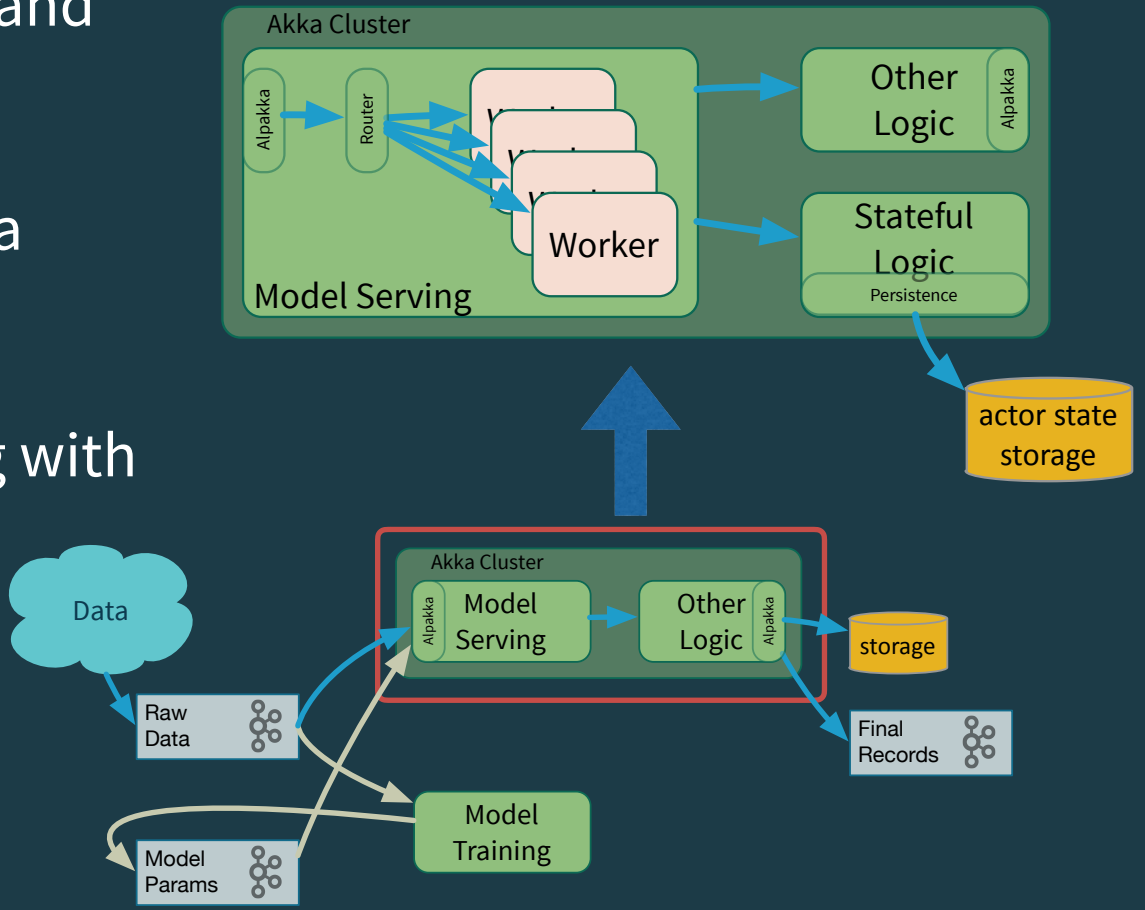
- For custom stage go to
  http://localhost:5500/state
- For actor-based implementation go to:
  http://localhost:5500/models
  http://localhost:5500/state/wine

Exercises!

- To find them, search for `// Exercise` comments in the code base.
- We'll suggest some you might try first.

Other Production Concerns

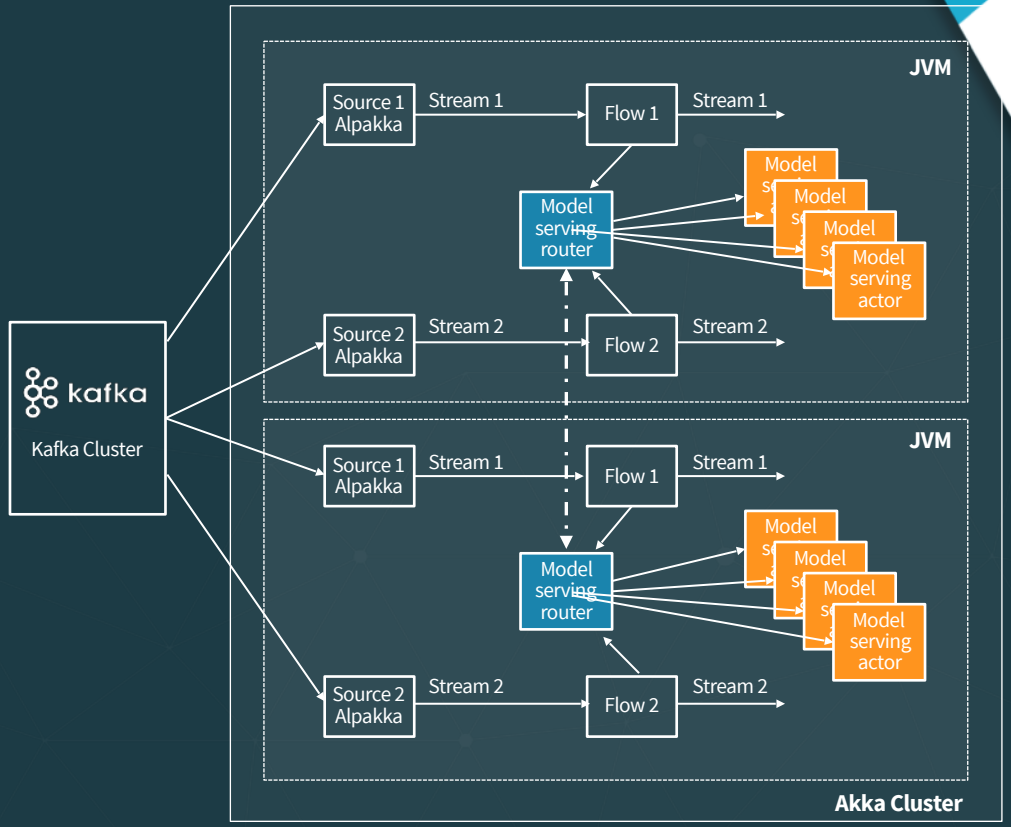
- Scale scoring with workers and routers, across a cluster
- Persist actor state with Akka Persistence
- Connect to *almost* anything with Alpakka



Using Akka Cluster

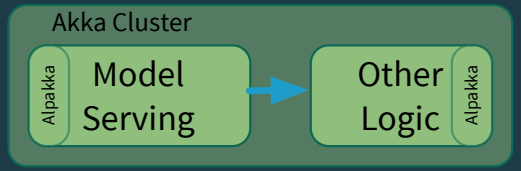
Two approaches for scalability:

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

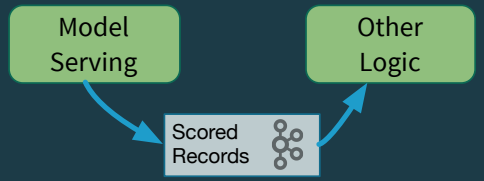


http://michalplachta.com/2016/01/23/scalability-using-sharding-from-akka-cluster/

Go Direct or Through Kafka?



vs.

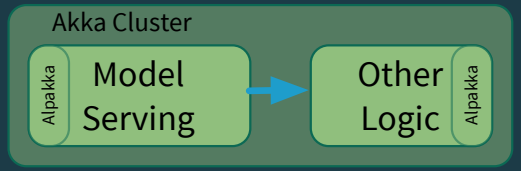


?

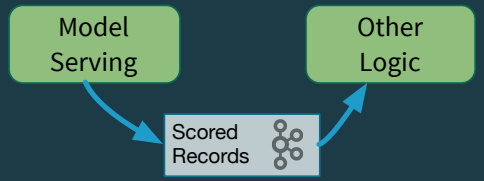
Extremely low latency

Higher latency (e.g., queue depth)

Go Direct or Through Kafka?



vs.



?

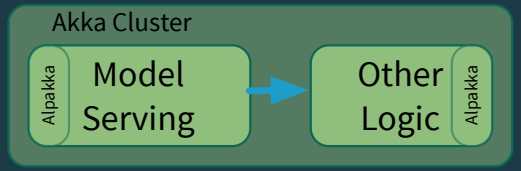
Extremely low latency

Minimal I/O and memory overhead; avoid marshaling

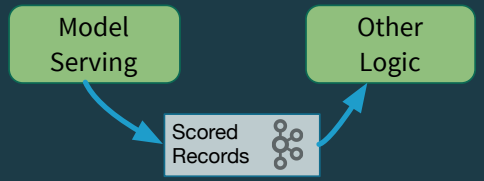
Higher latency (e.g., queue depth)

Higher I/O and processing (marshaling) overhead

Go Direct or Through Kafka?



vs.

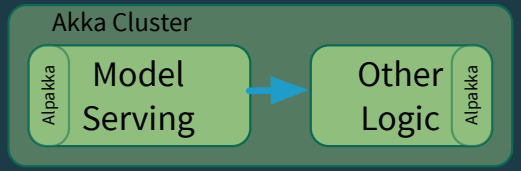


?

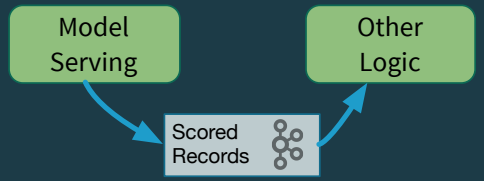
Reactive Streams back pressure

Very deep buffer (partition limited by disk size)

Go Direct or Through Kafka?



vs.



?

Reactive Streams back pressure

Direct coupling between sender and receiver, but indirectly through an ActorRef

Very deep buffer (partition limited by disk size)

Strong decoupling - M producers, N consumers, completely disconnected

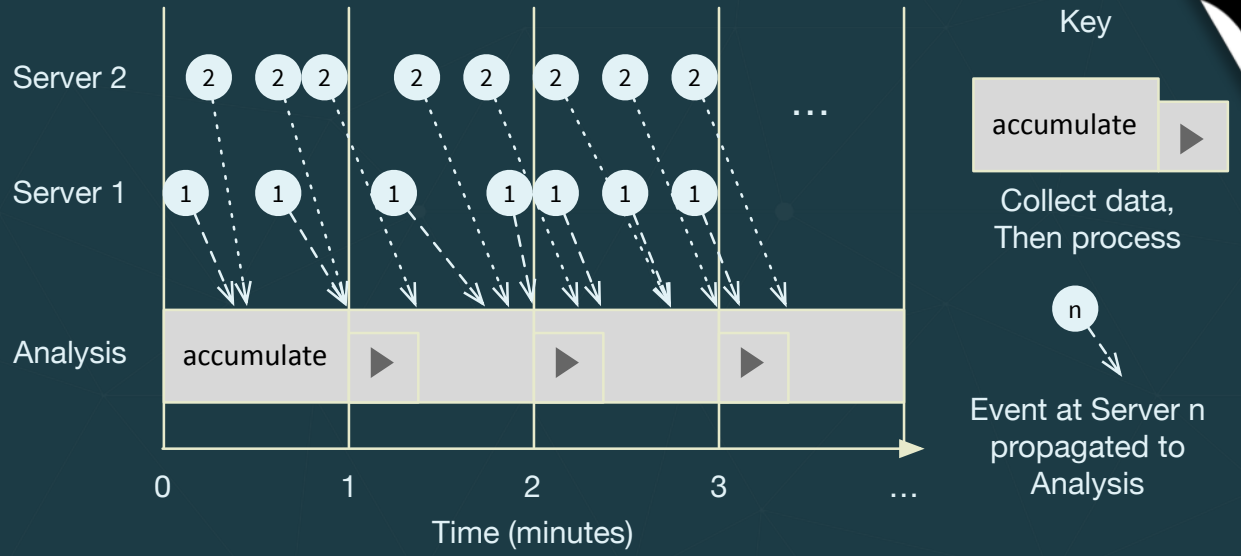
Kafka Streams

Kafka Streams

- Important stream-processing concepts, e.g.,
  - Windowing support

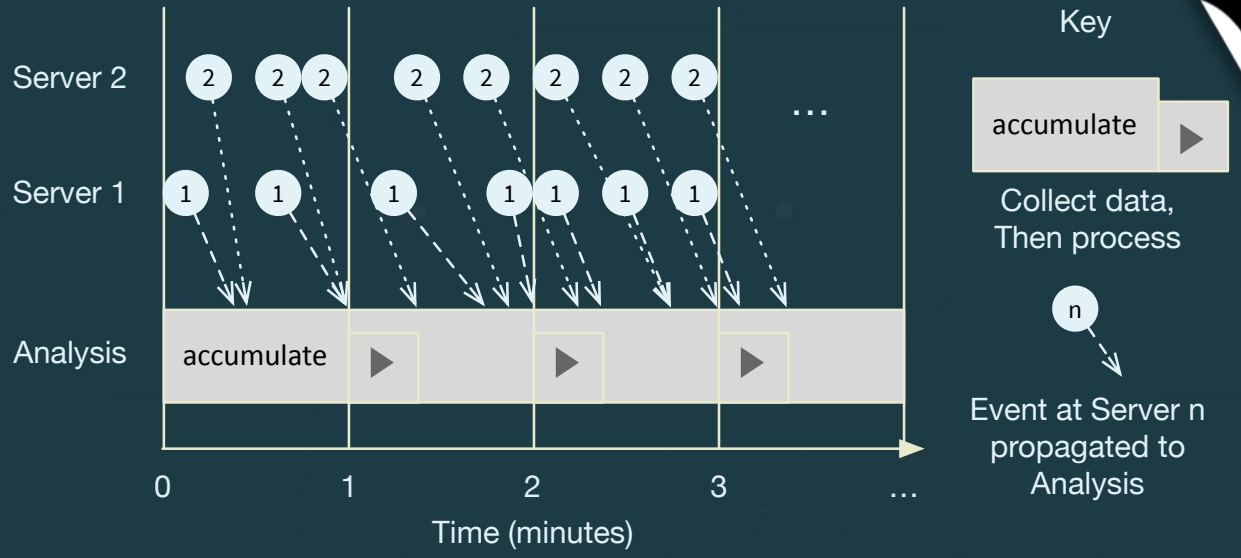
Kafka Streams

- Important stream-processing concepts, e.g.,
  - Windowing support



Kafka Streams

- Important stream-processing concepts, e.g.,
  - Distinguish between *event time* and *processing time*

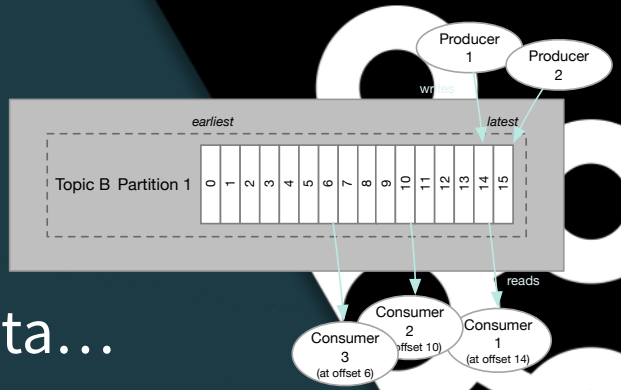


Kafka Streams

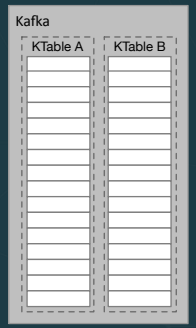
- Important stream-processing concepts, e.g.,
  - For more on these concepts, see
    - Dean’s O'Reilly report ;)
    - Talks, blog posts, & book by Tyler Akidau

Kafka Streams

- KStream - per-record transformations
  - The typical way you want to process data…



- KTable - key/value store of supplemental data
  - Useful for management of application state



Kafka Streams

- Low overhead
- Read from and write to Kafka topics, memory
  - Use Alpakka or Kafka Connect for other sources and sinks
- Load balance and scale based on partitioning of topics
- Built-in support for Queryable State

Kafka Streams

- Two types of APIs:
  - Processor Topology API
    - Lowest-level API
    - Compare to Apache Storm
  - DSL based on collection transformations
    - Compare to Spark, Flink, Scala collections APIs.

Kafka Streams

- Started with a Java API
- Lightbend donated a Scala API to Kafka
  - https://github.com/apache/kafka/tree/trunk/streams/streams-scala
  - (See also our convenience tools for distributed, queryable state: https://github.com/lightbend/kafka-streams-query)
- SQL - yes, implemented as separate application (i.e., not a library like in Spark and Flink)

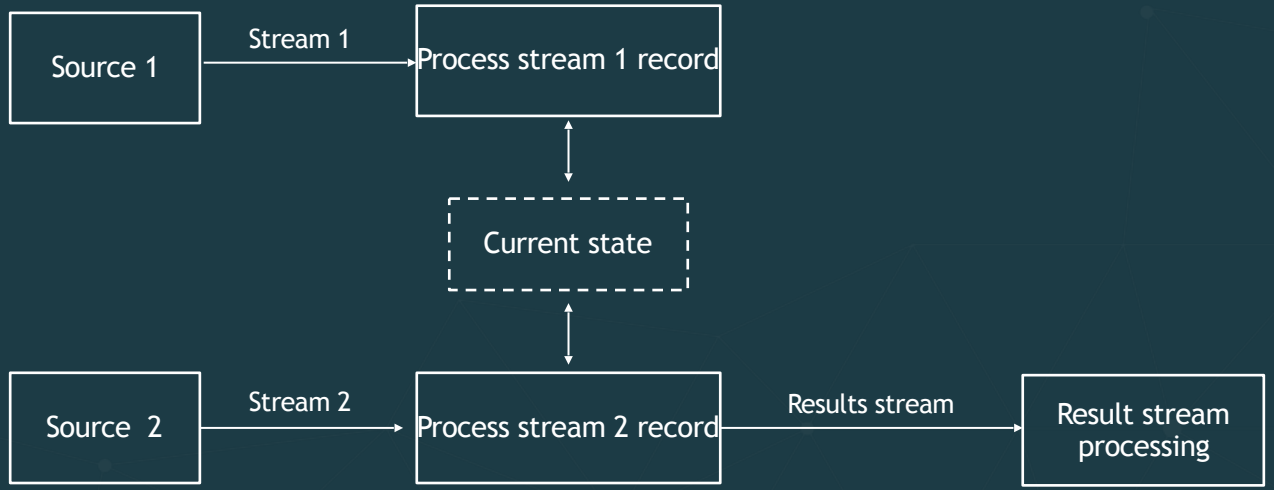
Kafka Streams

- Ideally suited for:
  - ETL -> KStreams
  - State -> KTable
  - Joins, including Stream and Table joins
  - “Effectively once” semantics
- Commercial support from Confluent, Lightbend, Hadoop vendors, and others

Kafka Streams: “Effectively Once”

- A distributed transaction model.
- It is *exactly once*, **if** catastrophic failures don’t happen…

Model Serving With Kafka Streams



State Store Options We'll Explore

- "Naive", in memory store (no durability!)
  - Also uses the KS Processor Topology API
- Built-in key/value store provided by Kafka Streams
  - Uses the KS DSL
- Custom store
  - Also uses the DSL

Model Serving With Kafka Streams

Code time

1. Run the *client* project (if not already running)
2. Explore and run *kafkaStreamsModelServer* project
    1. Use the `c` or `custom` (or default) command-line argument for the *custom state store*
    2. Use the `s` or `standard` command-line argument for the KS built-in *standard store*
    3. Use the `m` or `memory` command-line argument for the *in-memory store*
    4. Use `–h` or `––help` for help

Model Serving With Kafka Streams

Check Queryable state

- For in Memory implementation

  http://localhost:8888/state/value

- For build in Standard Store

  http://localhost:8888/state/instances

  http://localhost:8888/state/value

- For Custom store

  http://localhost:8888/state/instances

  http://localhost:8888/state/value

Model Serving:
Other Production Concerns

Additional Concerns for Model Serving

- Model tracking
- Speculative model execution

Model tracking - Motivation

- You update your model periodically
- You score a particular record **R** with model version **N**
- Later, you audit the data and wonder why **R** was scored the way it was

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

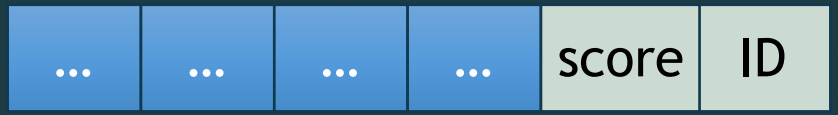
Model tracking

- Need a model repository
- Possible info stored for each model instance:
  - Name
  - Version (or other unique ID)
  - Creation date
  - Quality metric
  - Parameters
  - …

Model tracking

- 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



Speculative execution

According to Wikipedia speculative execution is:

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

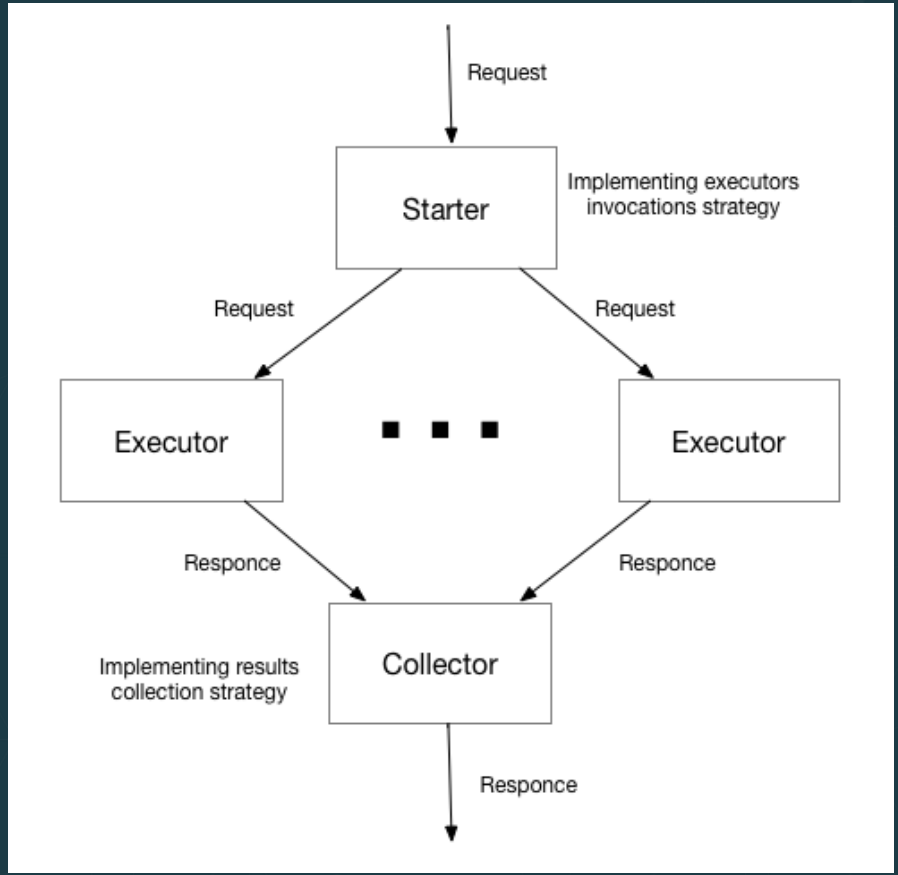
Speculative execution

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

Why not use it with machine learning??

General Architecture for speculative execution

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

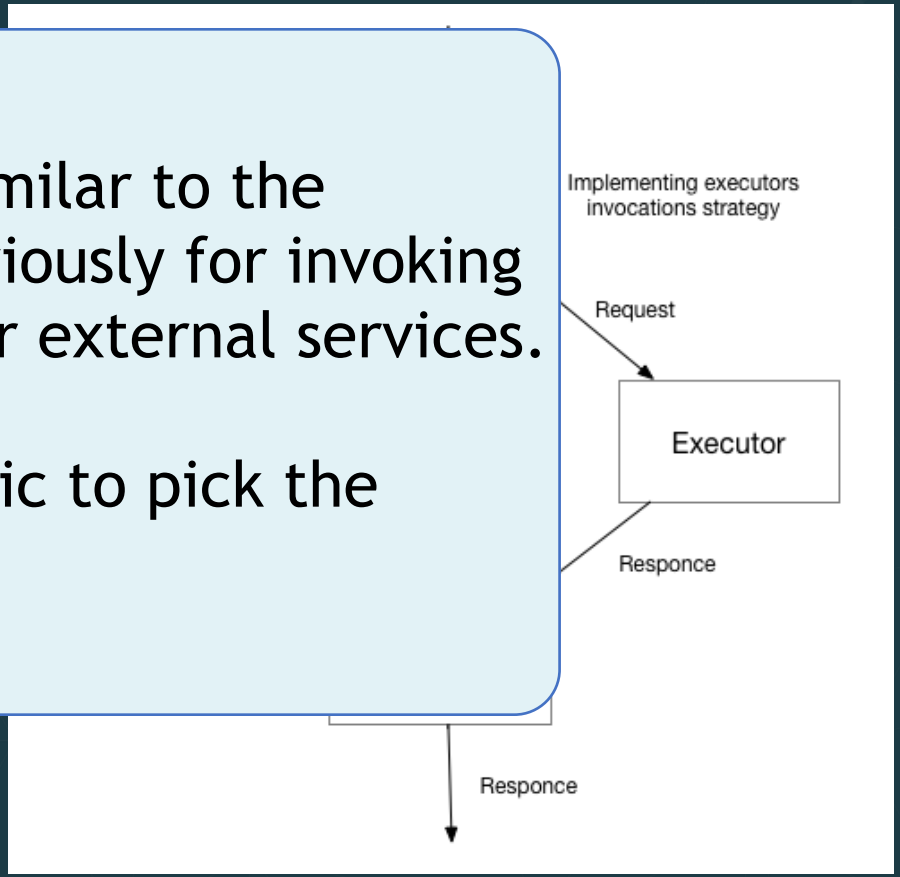


General Architecture for speculative execution

- Starter (p
  parallelis
  strategy.
- Parallel e
  executor
- Collecto
  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.



Applicability for model serving (1/3)

- Guarantee execution time, i.e., meet our latency SLA!
  - Several models:
    - A smart model, but takes time *T1* for a given record
    - A “less smart”, but fast model with a fixed upper-limit on execution time, *T2* << *T1*
  - If timeout T occurs, where *T1* > *T* > *T2*, return the less smart result
    - Do you understand why *T1* > *T* > *T2* is required?

Applicability for model serving (2/3)

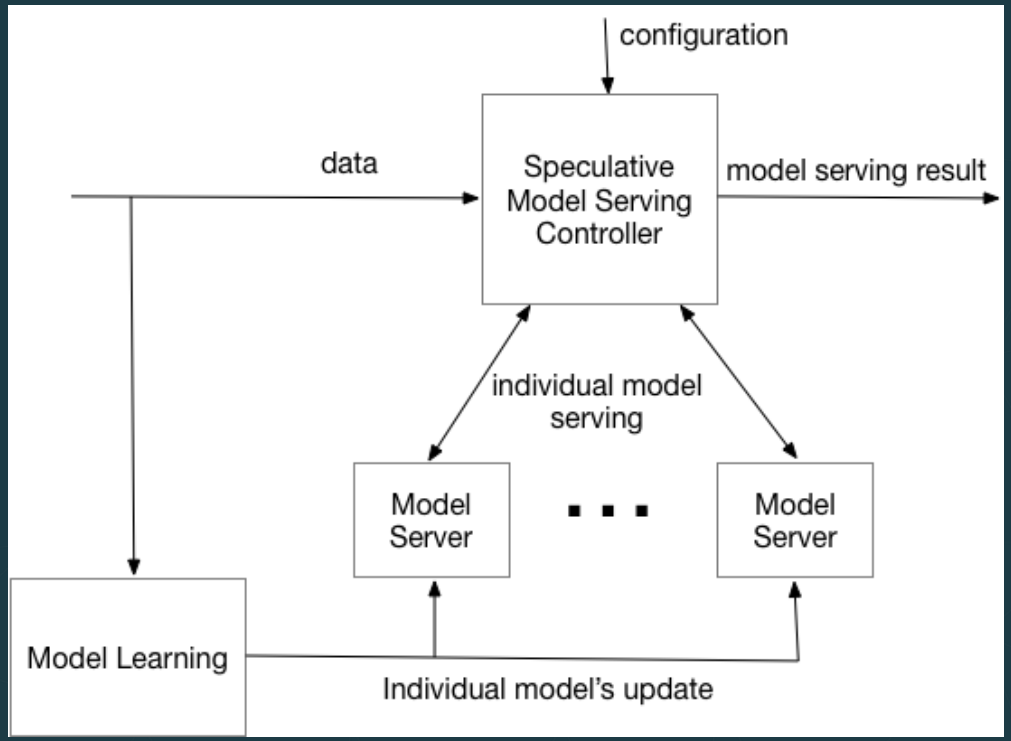
- Consensus based model serving
  - If we have 3 or more models, score with all of them and return the majority result

Applicability for model serving (3/3)

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

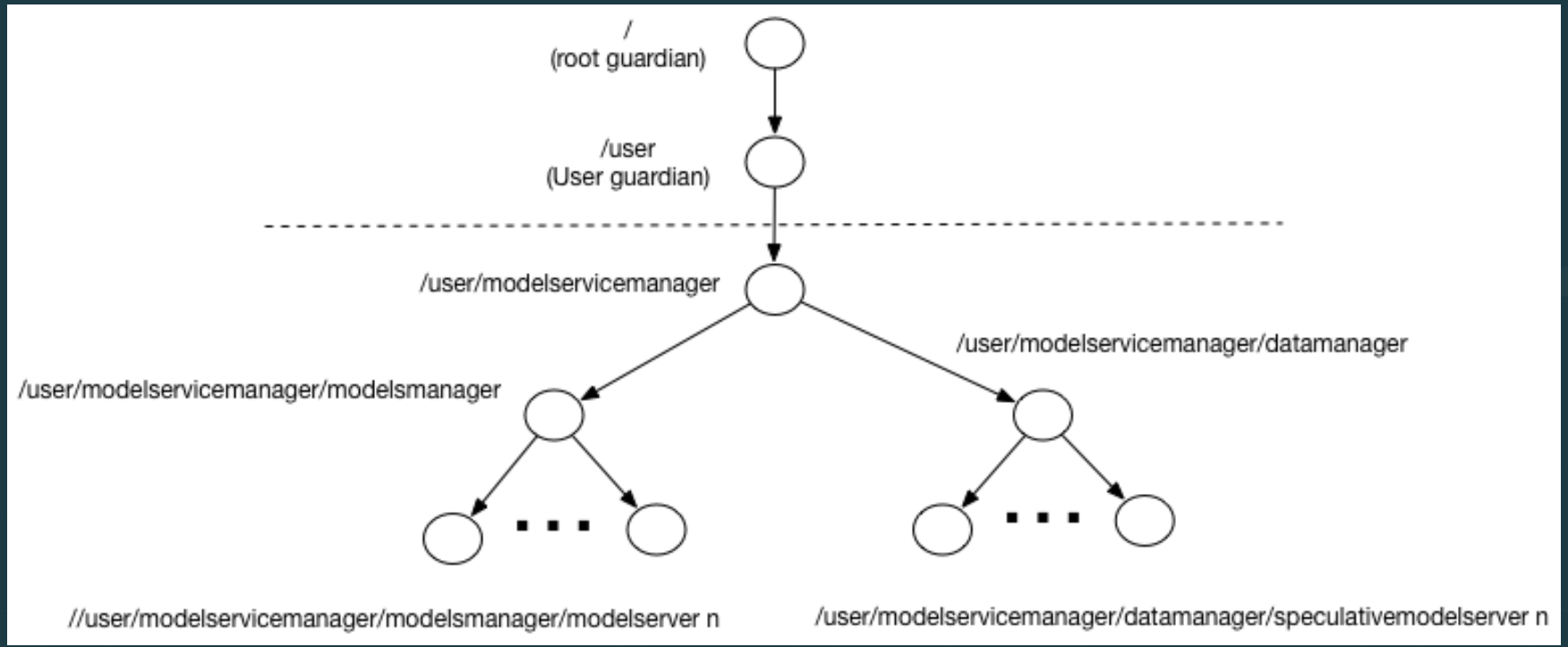
Of course, you can combine these techniques.

Architecture



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

One Design Using Actors



Exercises

Exercises

- Exercises are embedded in the source code
  - Search for `// Exercise` in the code
  - Additional exercises are described in the README.
- In most cases, an exercise is repeated in each variant of the Akka Streams and Kafka Streams implementations. Pick the one you find most interesting…

Exercises

- See the end of the README for a list of resources that will be useful as you modify the code, e.g,
  - Scala Library (Scaladocs)
  - Akka Streams:
    - Scala: Reference, Scaladocs
    - Java: Reference, Javadocs
  - Kafka Streams:
    - Scaladocs, Javadocs

Suggestions (1/4)

- ModelServerProcessor.java/scala, around line 55 in both
- StandardStoreStreamBuilder.java/scala: createStreams()
- CustomStoreStreamBuilder.java/scala: createStreams()
- MemoryStoreStreamBuilder.java/scala: createStreams()
    - Rather than just print results, write them to a new Kafka topic

Suggestions (2/4)

- ModelServingManager.java/scala: getModelServer()
- DataProcessor.java/scala (all of the variants!)
  - Implement either speculative execution or a worker “farm” for parallelism, rather than a single model instance.
  - We’ve only given you one model at a time to work with. Use a hack with multiple copies of the model, but add noise to the result to simulate different model results

Suggestions (3/4)

- AkkaModelServer.java/scala: main()
- CustomStoreStreamBuilder.java/scala, DataProcessor.java/scala, ModelProcessor.java/scala (all of the variants!)
    - They provide implementations of error handling, when input wine records fail to parse. Improve the implementations to write to a special Kafka topic instead.
    - Extend the idioms to handle model errors.

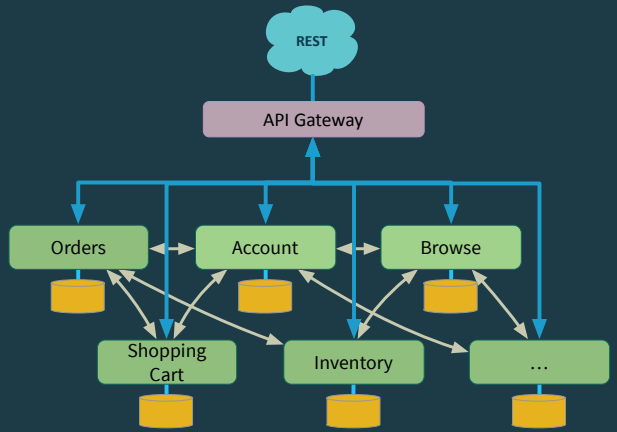
Suggestions (4/4)

- model project
  - It hard-codes information about the example wine records. Can you make it more generic? See the suggestions in PMMLModel.java/scala or TensorFlowModel.java/scala
- Can you make other areas where WineRecord is assumed more generic?

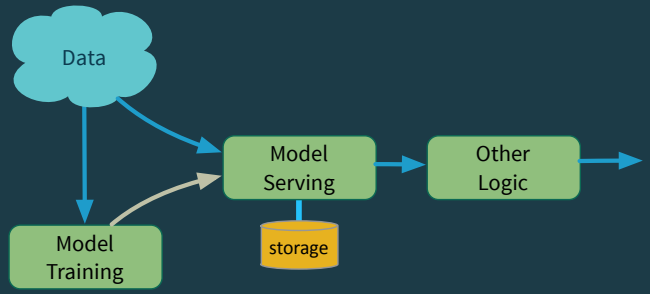
Wrapping Up

Picking Akka Streams vs. Kafka Streams

Event-driven μ-services



“Record-centric” μ-services



Events

Records

Next Steps (1/2)

- Study the different model serving techniques
- Study the “model” subproject
- Look at how the following are implemented
    1. queryable state
    2. embedded web servers
    3. use of Akka Persistence
    4. model serialization

Next Steps (2/2)

- Do more exercises
  - Search for `// Exercise` in the code
  - See the README
- Ask us for help on anything…
- Provide us with feedback on how we can improve
- Visit lightbend.com/fast-data-platform
- Profit!!

Thanks for coming!

Questions?

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

Lightbend