Integrating Kotlin into a Poliglot JVM Environment

György Móra
Principal Data Scientist

EKATA

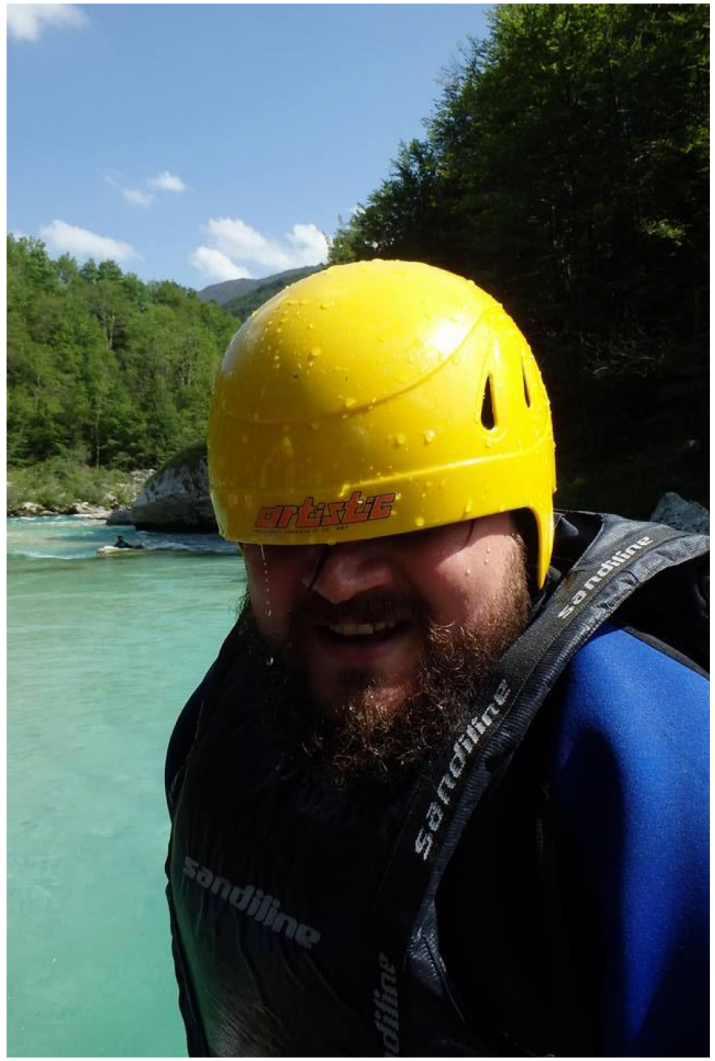
Agenda

- Who is who?
- Identity Check
- System overview
- Our Kotlin ML predictor library
- Integration challenges

EKATA

György (ɟørɟ)

- ML/DS projects
- been doing:
  - NLP
  - Engineering
  - ML



EKATA

Ekata - Who we are?

- Ekata is a global leader in identity verification
- Merchants, online lenders, marketplaces
  - Check identity
  - Fight fraud
- We provide 60+ signals
- The Identity Check Confidence Score is a ML product to summarize the signals

Integrated With the Top 5 Global Fraud Platforms

Top 3 Global Online Travel And Hospitality Sites

We Work With 2 of the Major Card Brands

Seven of the Ten Largest US Retailers

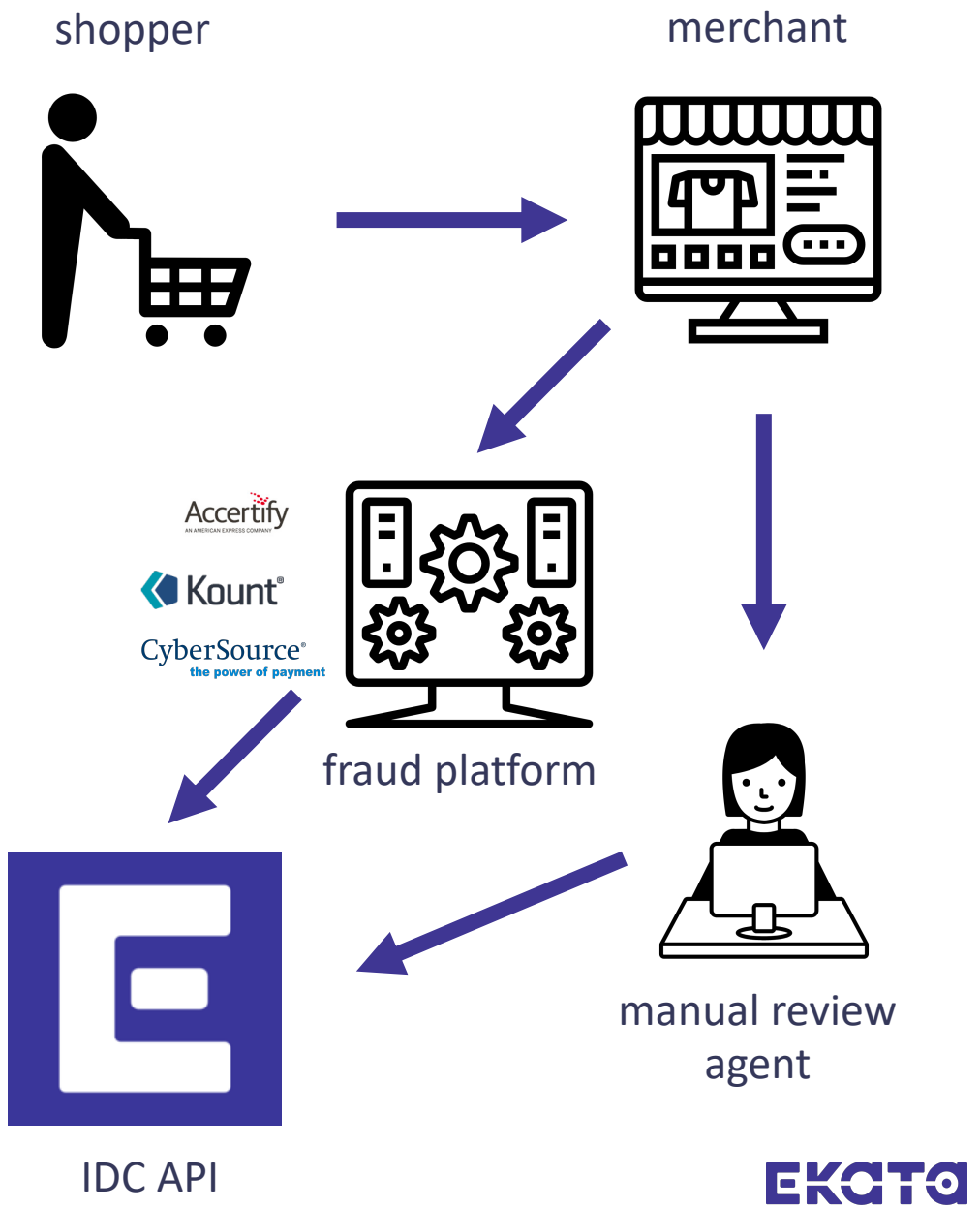
Six of the Ten Top Global Marketplace Companies

Five of the Top Ten Global Retailers

Our business

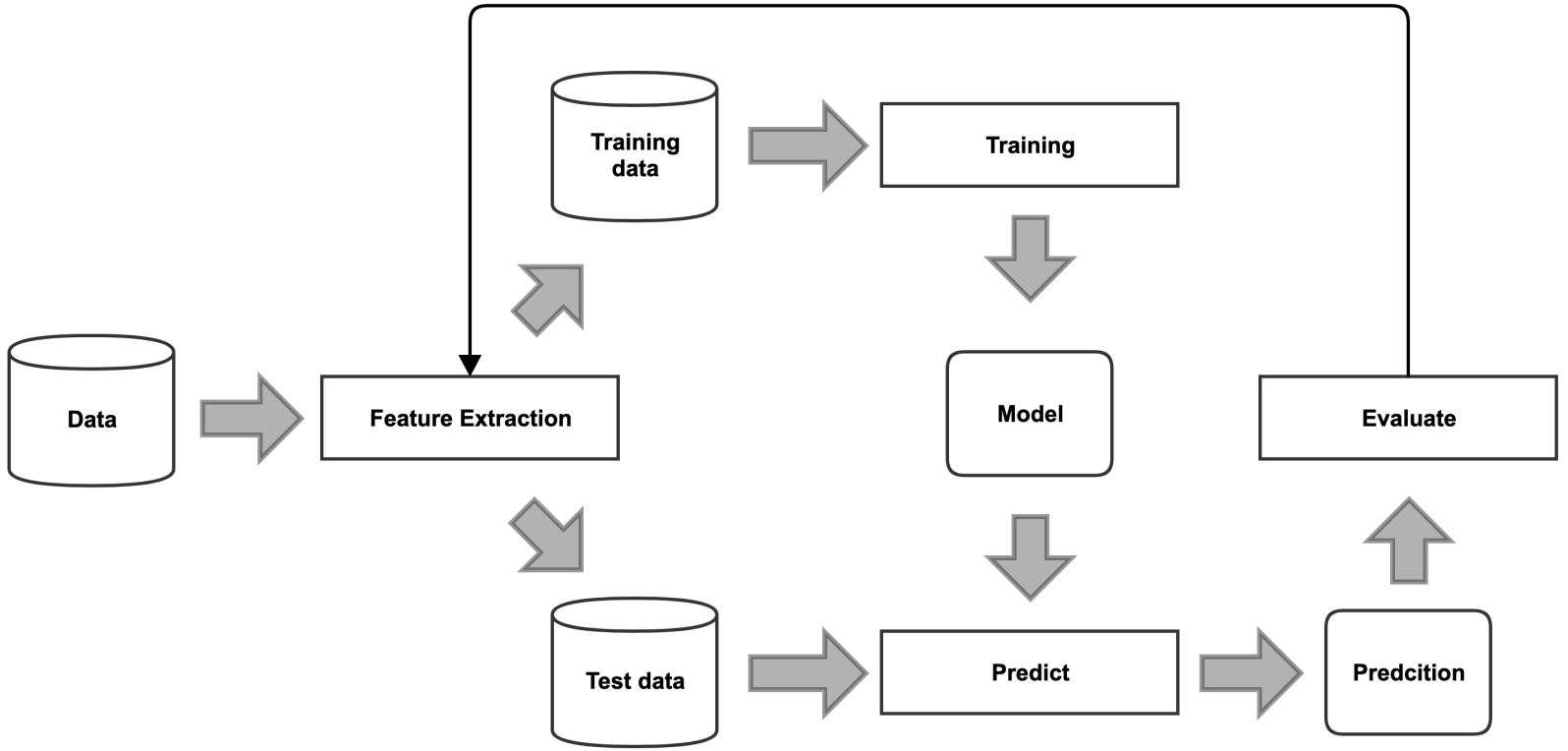
Identity Check

- Identity
  - e-mail
  - Phone
  - Address
  - IP address
- Identity Graph database
- Transaction history database

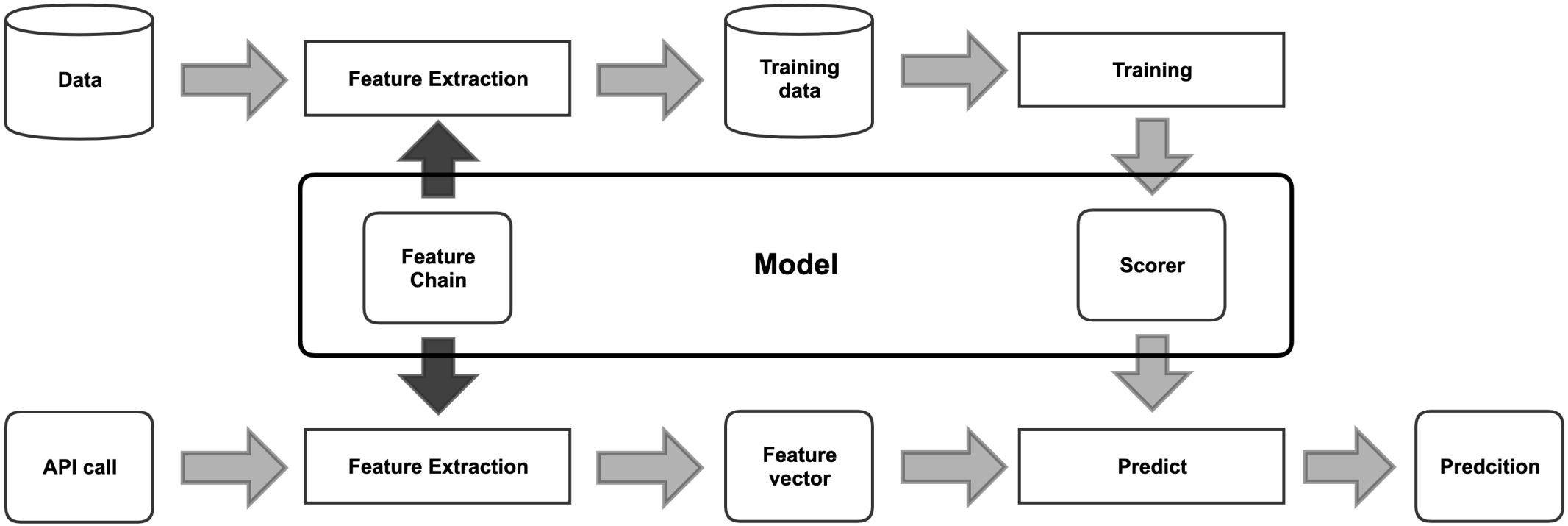


The machine learning lifecycle

- Feature extraction
- Train
- Eval
- Repeat

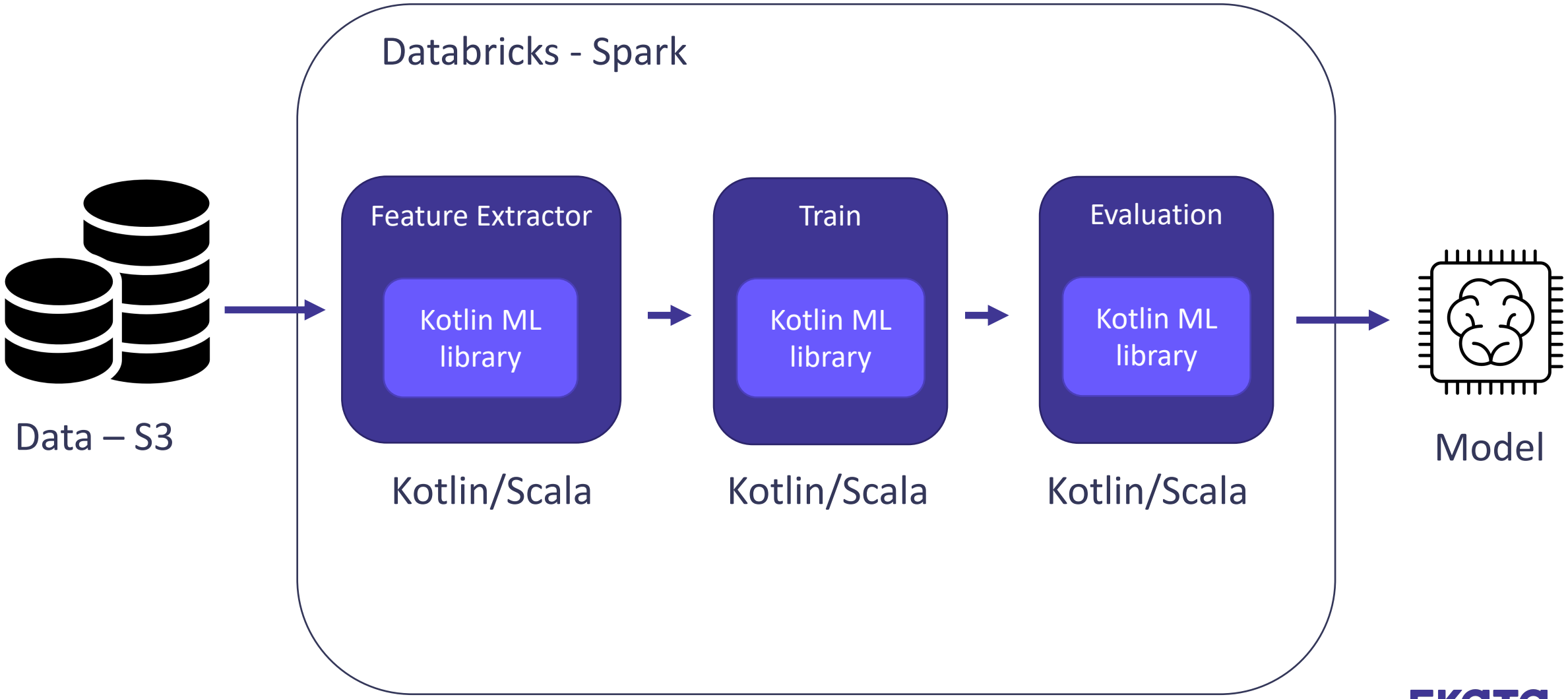


Our approach



EKATA

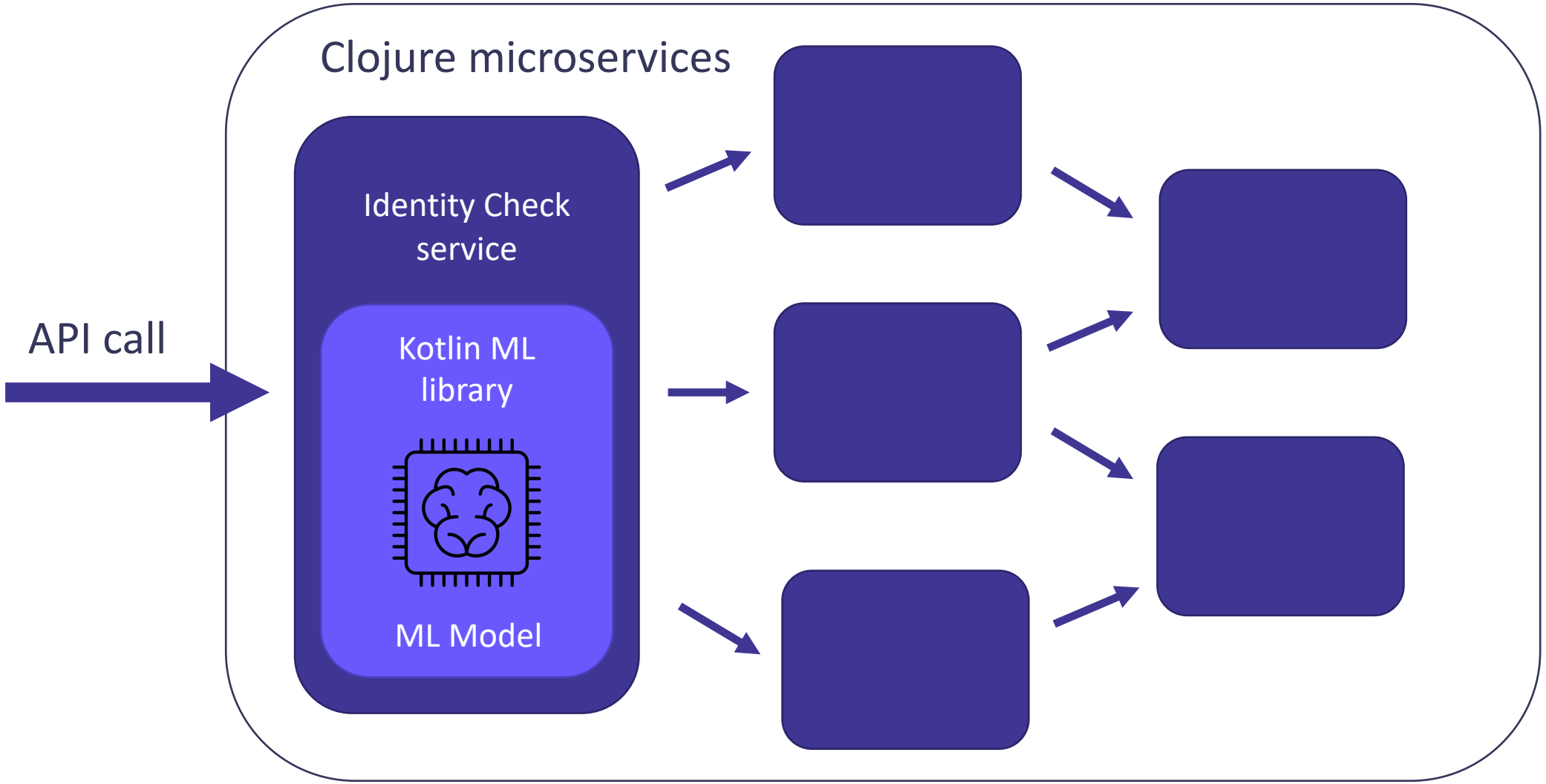
Kotlin code in the ML training pipeline



EKATA

Mixing Kotlin in

Kotlin code in the prod web-service



EKATA

Predictor (model + code)

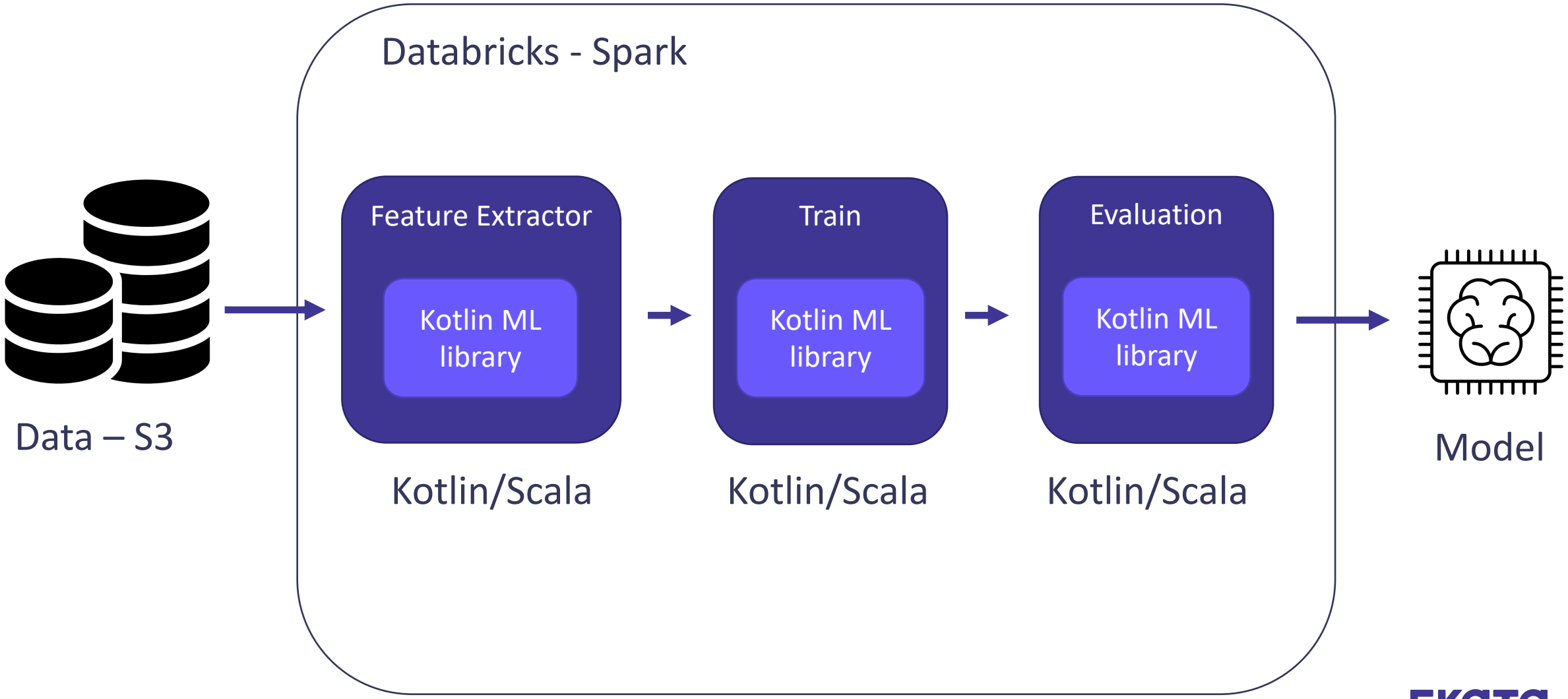
- It is a function call
- Self contained package
- Flexible for DS
- Easy to integrate for engineering
- Ships as a jar to Artifactory
- Pure JVM code

```
;; Clojure code calling the Kotlin ML predictor library
(defstate ^Predictor predictor
  :start (NetworkScorePredictorResources/loadDefaultPredictor))



(defex calculate [request]
  [params (network-signals-params request)]
  [network-signals (network-signals/fetch request params :network-signal/network-risk)]
  (.getScore (.scoreWithReasons predictor nil network-signals false)))
```

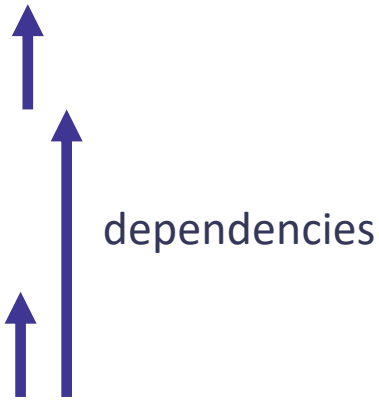
Kotlin code in the ML training pipeline



EKATA

Original project structure

- ML Training Pipeline
  - Feature Extractor
    - Spark module
    - Kotlin module
  - Training
    - Spark module
    - Kotlin module
  - Evaluation …



- Gradle project
- No Kotlin+Scala sources together
- Kotlin code depends on Scala
  - On- way interop only
- No interop between components
- Scala code is glue to Spark
  - Static codebase
  - Slow compilation

EKATA

Evolving codebase

- ML Training Pipeline
  - Feature Extractor
    - Spark module
    - Kotlin module
  - Training
    - Spark module
    - Kotlin module
  - Evaluation …

dependencies

- Kotlin components rewritten in Scala
- Scala -> Kotlin dependencies
  - Two-way interop
- Scala becomes the main language
  - Spark needs the Scala type system
  - Slow compilation

EKATA

Integration Challenges

1. Calling varargs

- Java and Kotlin varargs are **arrays**
- Scala varargs are **Seq**s
- Scala has the **@varargs** annotation

```
// Scala function
@varargs
def callTheVararg(args: String*): String =
  args.mkString(",")
```

```
// Call from Kotlin
hasVararg.callTheVararg("a", "b", "c")
```

```
// Scala function without annotation
def callTheClassicVararg(args: String*): String =
  args.mkString(",")
```

```
// Kotlin extension function
fun HasVararg.callTheClassicVararg(vararg args: String):
String? {
  return this.callTheClassicVararg(
    CollectionConverters.ListHasAsScala(
      args.toList()
    ).asScala().toSeq()
  )
}
// vararg call trough the extension function

hasVararg.callTheClassicVararg("a", "b", "c")
```

EKATA

Varargs summary

Annotated functions are easy

Kotlin extension functions help for non-annotated

Even “at least one” type vararg functions can be adapted with extension functions

EKATA

2. Scala closures

- Both languages have closures
- Single Abstract Method (SAM)
- FunctionN is SAM from Scala 2.12
- Java interfaces are here to help

```scala
// Scala function
def callTheLambda(f: String => String, s: String): String = f(s)
```

```java
// Call from Java
String result = callTheLambda(s -> s.trim(), "  text");
```

```kotlin
// Call from Kotlin
callTheLambda({s: String -> s.trim() }, "  text")
```

EKATA

Seamless above Scala 2.12

```
// Call from Kotlin with a method reference
val result = callTheLambda(String::trim, "  text")
```

```
// Call from Kotlin with a FunctionN instance
val f = Function1 { s: String -> s.trim() }
val result = callTheLambda(f, "  text")
```

```
// Call Scala 2.11 from Kotlin
val f = object: AbstractFunction1<String, String>() {
   override fun apply(v1: String?) = v1?.trim()
}
val result = callTheLambda(f, "  text")
```

EKATA

Extension helps in 2.11

```
// Call Scala 2.11 from Kotlin
// Kotlinify
fun NeedsLambda.callTheLambdaKoltin(f :(String) -> String, s:String): String {
  val f1 = object: AbstractFunction1<String,String>() {
     override fun apply(v1: String) = f(v1)
  }
  return this.callTheLambda(f1,s)
}

val result = needsLambda.callTheLambdaKoltin({s: String -> s.trim() }, "  text")
```

EKATA

Spark also has a Java interface

- UDF1 is a Java Interface
- The udf() function accepts Scala FunctionN and Java UDFN
- In 2.11 FunctionN is not SAM but UDF1 is Java

```
// Call Scala 2.11 from Kotlin
numbers.select(
   col("c"),
   udf(
     UDF1 { x: String? -> x?.let { x + x } },
     DataTypes.StringType.asNullable()
   ).apply(col("c"))
).show()
```

EKATA

Closures summary

Below Scala 2.12 needs some glue code (extensions can help here as well)

Kotlin closure -> SAM -> Scala FunctionN -> Scala closure

The single method Java Interfaces are extendable from Kotlin

EKATA

3. Some lists are not java.io.Serializable

Not serializable

```
listOf("a", "b", "c").subList(1, 1)

listOf("a", "b", "c").asReversed()

IntArray(1) { 1 }.asList()
```

Serializable

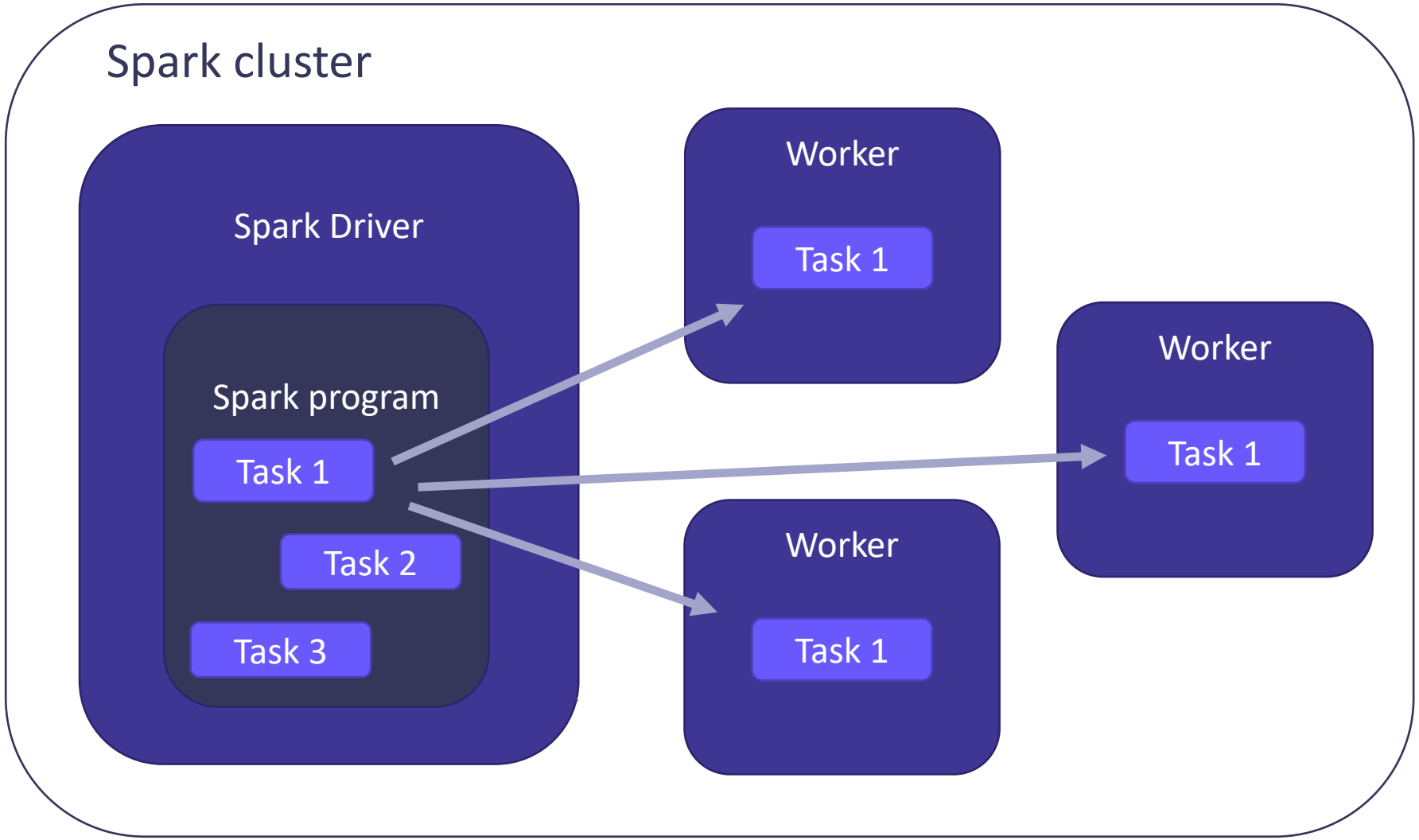
```
listOf("a", "b", "c").reversed()

arrayOf("a", "b", "c").asList()

IntArray(1) { 1 }.toList()
```

EKATA

Why serialization is important?



Serializable summary

java.io.Serializable is checked by Spark

Collection functions are not uniform

These types can be easily replaced to a serializable type

EKATA

4. scala.Double

- Scala Double in not a “reference type”
- Null in Scala becomes zero
- But it can be null in Kotlin

```
// Scala Double is primitive: v == 0
val v:Double = null.asInstanceOf[Double]
```

```
// v == null
val v:java.lang.Double = json.readValue("null", java.lang.Double::class.java)

val v:scala.Double = json.readValue("null", scala.Double::class.java)
```

EKATA

4. kotlin.Double?

```
// Kotlin non-null Double: v == 0
val v:Double = json.readValue("null", Double::class.java)
```

```
// v == null
val v:java.lang.Double = json.readValue("null", java.lang.Double::class.java)

// v == null
val v:Double? = json.readValue("null",  object: TypeReference<Double?>() {})
```

```
inline fun <reified T: Any> ObjectMapper.readValue(str: String): T = readValue(str, jacksonTypeRef<T>())

// Compiles but fails with runtime error: readValue(content, jacksonTypeRef<T>()) must not be null
val v: Double? = ObjectMapper().readValue("null")
```

Double summary

scala.Double can be null in Kotlin (but not in Scala)

java.lang.Double is always a nullable (reference) type

Nulls can silently transform to zeroes

EKATA

5. Clojure maps

- Clojure is dynamically typed, Lisp
- Keywords and Symbols
- A Clojure map is a java.util.Map

```
;; Clojure map
{:key1 "value", :key2 { :key3 123 }}
```

```
clojureMap.toString() == "{:key1=value, :key2={:key3 123}}"

json.writeValueAsString(clojureMap) == """{":key1":"value",":key2":{":key3":123}}"""

clojureMap.containsKey(":key1") == false

clojureMap.keys.first() is clojure.lang.Keyword
```

EKATA

Scala maps

- Scala has mutable and immutable maps
- A Scala map is **not** a java.util.Map

```
// Scala Map
Map("a" -> 1, "b" -> 2)
```

```
// Lambda magic works here as well
val mappedMap = scalaMap.map { e-> Tuple2(e._1, e._2+1) }
mappedMap.get("a").get() == 2

// json serialization of scala.Map can be a problem
json.writeValueAsString(scalaMap) ==  """{"empty":false,"traversableAgain":true}"""
```

EKATA

Maps summary

Scala maps are not “java maps”

SAM can help us to use functional interface of scala maps

Check the type of objects the Kotlin code receives from other languages

EKATA

6. Shaded dependencies

- With shadow we renamed certain libraries
- Spark depends on old version of Jackson parser
- To properly serialize Kotlin classes we need the Jackson Kotlin Module

```
import shadowed.com.fasterxml.jackson.module.kotlin.registerKotlinModule
val jsonKotlin = shadowed.com.fasterxml.jackson.databind.ObjectMapper().registerKotlinModule()


val jsonSpark= com.fasterxml.jackson.databind.ObjectMapper()
```

EKATA

Broken shaded extension functions

```
// ERROR :cannot import
import shadowed.com.fasterxml.jackson.module.kotlin.registerKotlinModule
val jsonKotlin= shadowed.com.fasterxml.jackson.databind.ObjectMapper()

// ERROR: Unresolved reference: registerKotlinModule
jsonKotlin.registerKotlinModule()

// ERROR:  Unresolved reference: ExtensionsKt
shadowed.com.fasterxml.jackson.module.kotlin.ExtensionsKt.registerKotlinModule(jsonKotlin)
```

```
import shadowed.com.fasterxml.jackson.databind.ObjectMapper;
import shadowed.com.fasterxml.jackson.module.kotlin. ExtensionsKt;

ObjectMapper json = new ObjectMapper();
ExtensionsKt.registerKotlinModule(json);
```

EKATA

Shading summary

Magic on magic can blow up (shading + exrtension)

Kotlin compiler makes “magic” classes inaccessible via regular interface

A little Java glue code can help when Kotlin compiler fail

EKATA

Open sourcing the ML library

- Target systems
  - Training
    - Spark/JVM
    - Python
  - Prediction
    - JVM
    - Python
- Roadmap
  - ?

EKATA

What we learned

If something is technically possible -> extension functions can make it nice

Common Java types are handled well in most language
(and Kotlin collections are basically Java containers)

Sometimes Java comes to the rescue

If possible, avoid two-way interop

EKATA

Find me at the fireside chat!

Code and slides:
https://github.com/giurim/kotliners

Check out my little side-project:
https://github.com/helmethair-co/scalatest-junit-runner