Data Fellas

Distributed Data Science

O'Reilly Online Training on Integrated Data Science Pipeline,
given by

Andy Petrella, Founder

Xavier Tordoir, Founder

1-3 March, 2016

Data Fellas

Revolutionizing the Data Science methodologies

Founded May 2015 in Belgium.

OSS project: **spark-notebook** (> 1K stars)

Enterprise product: **Agile Data Science Toolkit**

Check http://data-fellas.guru

Andy Petrella
(Founder)
Math (MSc), CS (MSc)
Geospatial
@noootsab

Xavier Tordoir
(Founder)
Physics (Ph. D)
Bioinformatics
@xtordoir

Data Scientists
Distributed Computing Engineers

Founders

Experts in Scala (JVM) data science (keynote / invited speakers)
Member of **O'Reilly Strata** program committees (CA, NYC and London)

About the training

Slack: https://oreillyonlinetraining.slack.com/messages/distributed-pipelines/

Setup instructions (slack): https://oreillyonlinetraining.slack.com/files/andy.petrella/F0PABQX1R/Setup_Instructions

March, 1st 9:00-11:00AM PST

- Intro
- Data Collection
- Interactive Programming
- Queue

March, 2nd 9:00-11:00AM PST

- Streaming Data
- In-Memory Data
- Data Analysis

March, 3rd 9:00-10:00AM PST

- Access Layer
- Cluster Manager
- Wrap Up

3 assessments:

Notebooks to be returned with missing code filled in

Acknowledgement

Marie Beaugureau, Ben Lorica and Paco Nathan from O'Reilly who proposed us this training and drove its accomplishment!

Alexy Khrabrov who coined the acronym SMACK referring to the architecture proposed here and organized a training during Big Data by the Bay 2015 gathering the "hall of fame"'s team.

Distributed Pipeline

Dealing with Data

Data

This training would mean anything without data

They are omnipresent, they rule the way we behave, the way the society adapts and evolves

They contain the information how this happens

Extracting this information can help us to proactively act on a system (human behavior, distribution process, production line, …) **to overcome problems or to optimize the return**

Hence the more data we can collect and use, the more information we'll generate, the better we'll operate

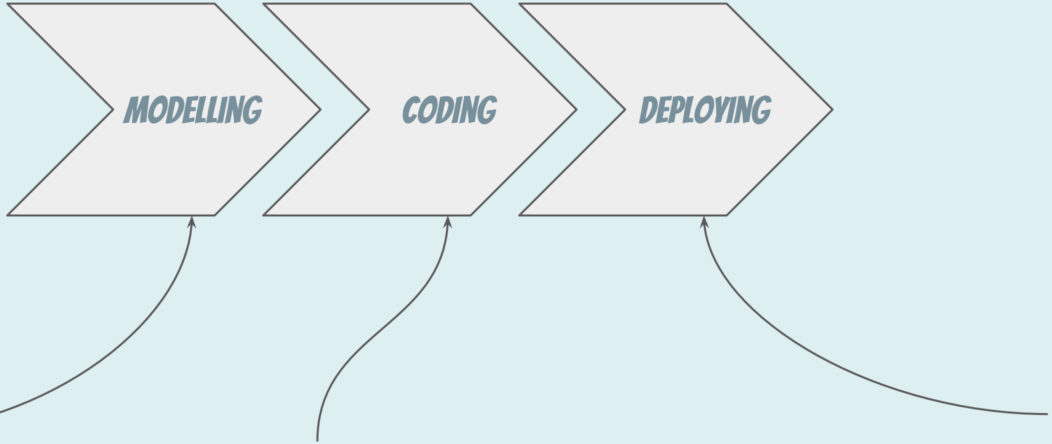
Pipeline

So, the data is omnipresent, or more precisely, the data is now accessible, because it is materialized by sensors, surveys, virtual social interactions, and the list goes on

Hence, an enterprise solution's quality can be estimated by its capacity to capture most of the data and to turn it into valuable information.

A pipeline is the general term to qualify such solution that tightens up several processing steps together.

Productizing Data Science

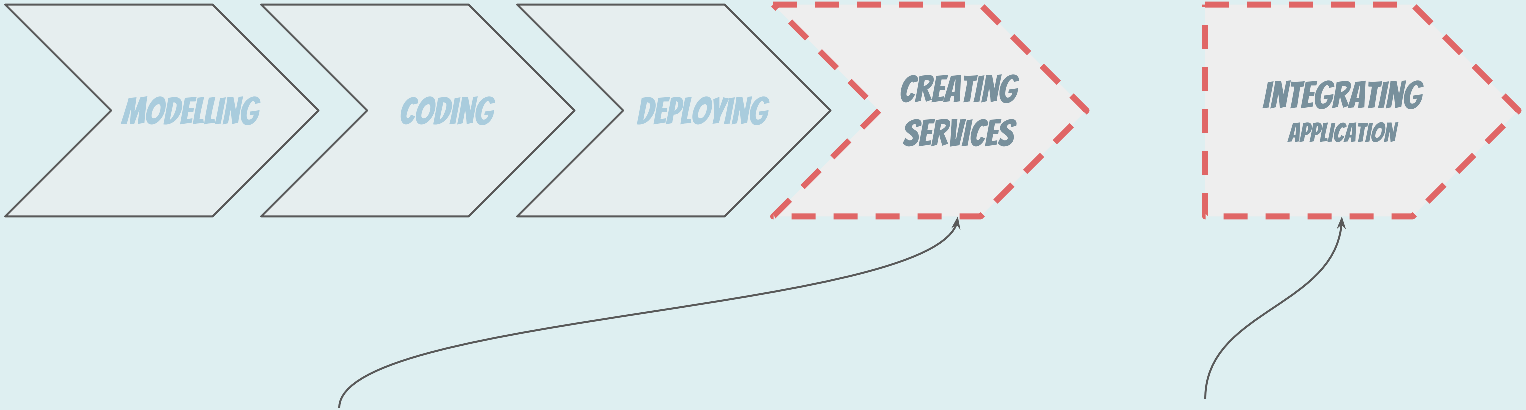


FINDING DATA, SAMPLE
PARSING STRUCTURES
CLEANING
(REDUCING)
LEARNING
PREDICTING

CONNECT PROD DATA
TUNING TRAINING PARAMETERS
CREATE PREDICTION SERVICE
GENERATE DEPLOYABLE

CONNECT TO PROD INFRASTRUCTURE
INTEGRATION WITH EXISTING ENV
ALLOCATE (SCHEDULE) RESOURCES
ENSURE AVAILABILITY

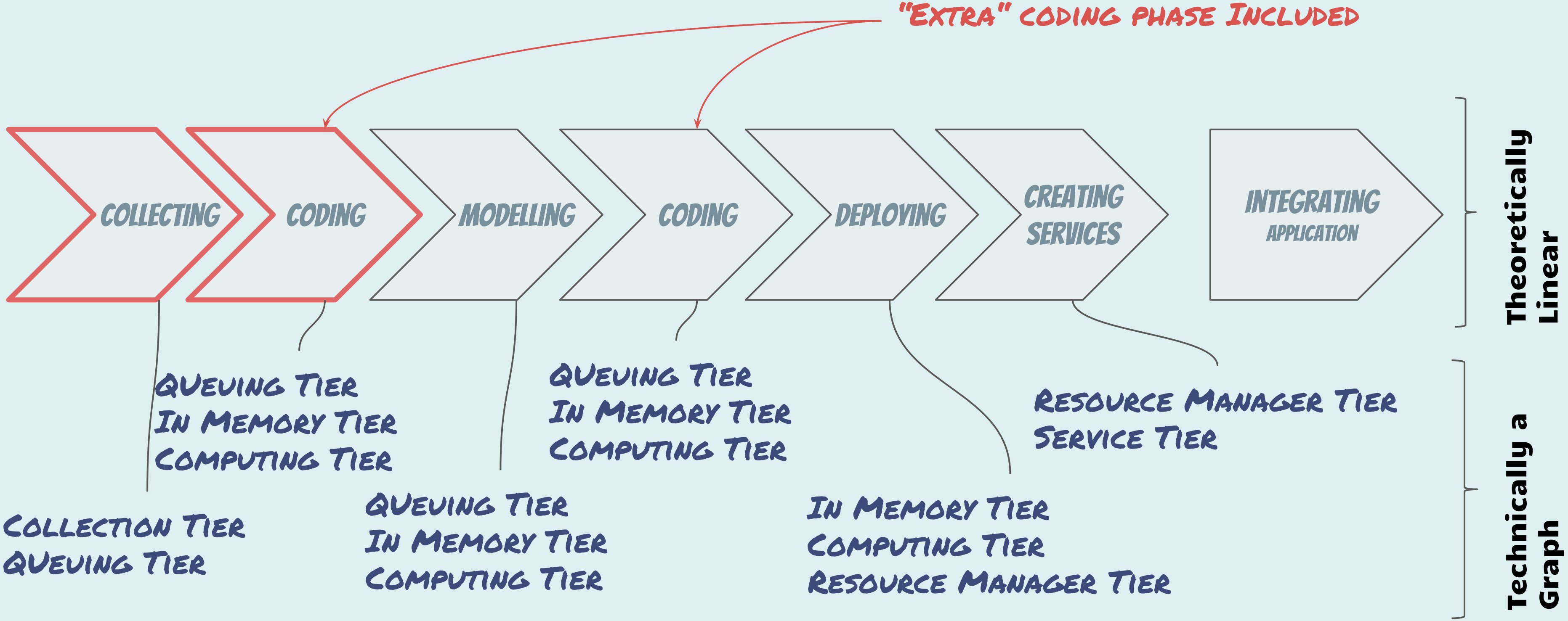
Extended Pipeline



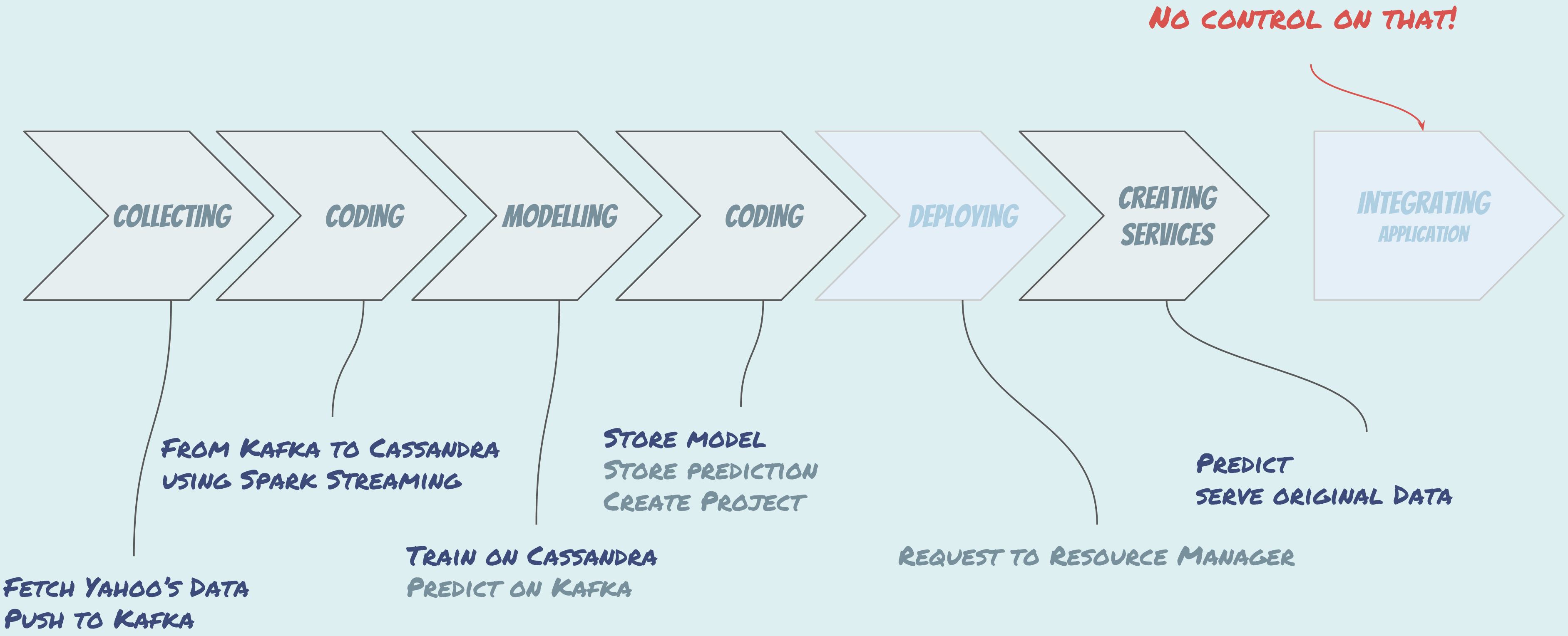
ABSTRACTS ACCESS TO PREPARED VIEWS
EXPOSES PREDICTION CAPABILITIES
HIGHLY HORIZONTALLY SCALABLE
SCALING MICRO SERVICES CLUSTER
→ CHEAPER THAN COMPUTING CLUSTER

CUSTOMER INTEGRATION
CAN BE ANY TECHNOLOGIES
CAN EVEN BE ANOTHER PIPELINE!

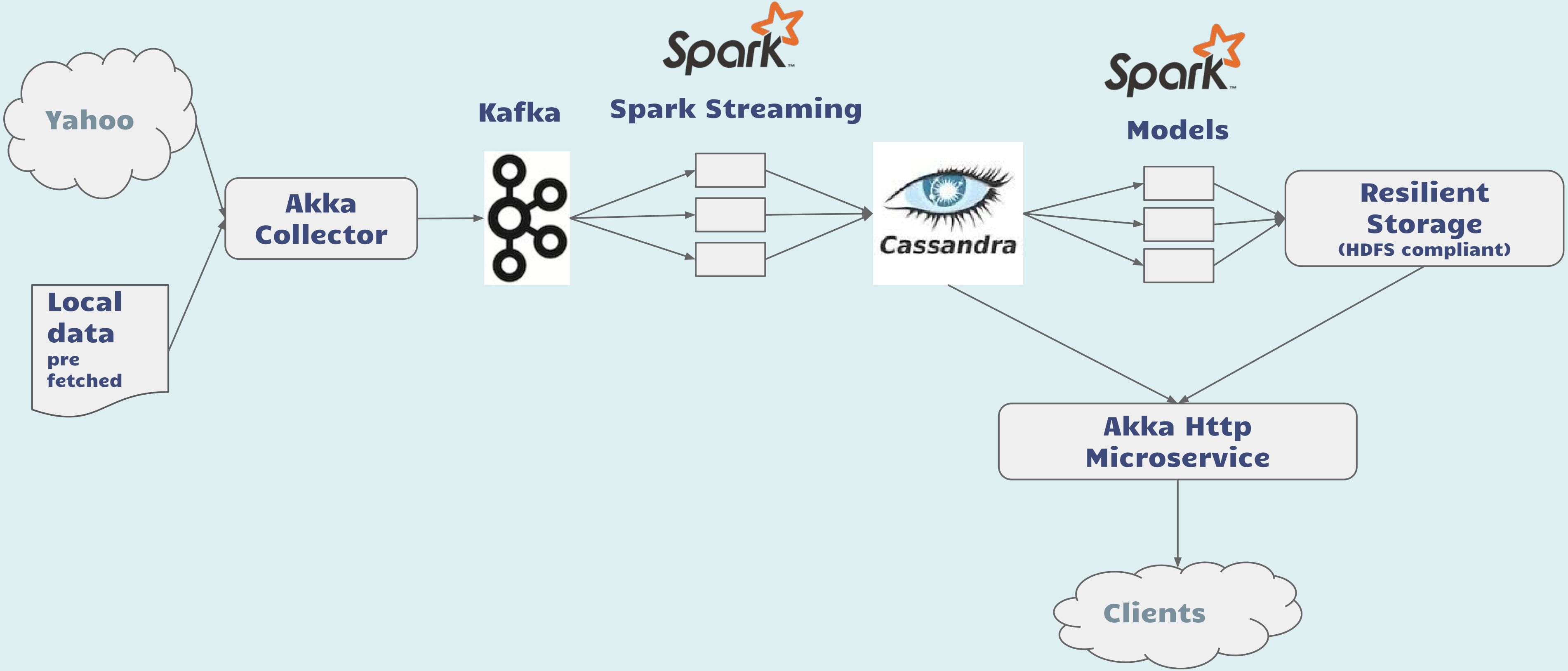
Extended Pipeline



Analyse Stock Market



Analyse Stock Market: Concretely



When

It's recommended to introduce this workflow as soon as possible, from the start if possible.

The reason behind that is because it stimulates the integration of the different required skillsets

This could be considered a downside, but it's wrong

This pipeline doesn't introduce a new level of complexity, it exposes it explicitly!

Which is definitely an advantage that avoids problem to be overlooked

Data Collection

Ingest Data

Data

In the context of data science pipeline, the first thing to take care of is to ingest the data.

This is actually not exactly true, the very first step is to find it! This step is generally taken out of the picture, because it impacts the whole system.

Ingesting data has to be done reliably. Again, the more data, the more information we can extract.

However, the data will generally be generated by many external services, devices, …

Getting it

How to get the data can take several forms, but the two most important patterns are

- **Request / Response**
- **Streaming**

Request / Response

A classic! The data is available in third party and the client needs to ask for it…

Asking data is done by issuing a request to the server which answers with a response containing the data.

Simple, but it remains two questions:

- **what if the service returns a lot of data?**
- **what if the service generates data very fast?**

And a subsidiary one…
What if the service can only send the data points only once!?

Streaming

Streaming services are obviously structurally different than the request / response pattern.

The divergent point is principally in the fact that a permanent connection is made between the service and the consumer.

The data can then flow continually through the pipe.

So what would happen if the collector cannot keep up with incoming rate?

KISS

Collectors must be kept as simple as possible

That will lower the need to increase the rate of requests

But also, it will reduce the number of collector instances

Hence the principal role that should be attributed to the collector layer is to pass the data to the queue (next section)

Extendable

To make the point again, the more data, the more information.
However, data is something completely embed into a context
(complex system composed of many variables, like sensors type, measure method, time, semantic, ...)

Also, when we process data, we do it implicitly within a context too.
This context will have to be reconstructed from the data's ones

That's one of the reason why multiplying the sources is beneficial to a good analysis!

But that implies that our collection layer needs to be easily extended to quickly ingest new types of data

Micro Service

To recap the collection layer needs to be

- **scalable vertically** (f.i. increase the requests' rate)
- **scalable horizontally** (f.i. support fast data)
- **Extendable** (f.i. to support new sources, types, context)

A few reasons why we consider Micro Service, which is a fancy name for "function as a service".

Data Collector

Using Akka

Akka

Akka is a lightweight library that eases mainly two things

- **Create concurrent programs using the Actor model.**
- **Create reactive program thanks to high level abstraction of Message Passing Style**

Leverage three important principle:

- **location transparency**
- **Asynchronousity & non-blocking**
- **Thread isolation**

MPS

Message Passing Style is the key which enables asynchronousity and non blocking calls.

In Akka, the Actor model is used to abstract this style out.

An actor is mainly a function that is always calls asynchronously by receiving a message (from a mailbox).

Each actor has an address (mailbox) that allows all of them to communicate as a system to solve one problem

A dispatcher is used to orchestrate the way invocations are done by assigning a message to an actor in a given free thread

Scale Vertically

Akka is a great solution for vertical scaling

Since an actor is the atomic piece of computation and is basically a function we can instantiate new ones

A router can be used to abstract this out… which is itself an actor!

When it comes to number, we can create up to 2.5 millions actors on a 1Gb heap (roughly 450b each)

Scale Horizontally

Akka comes two neat features that allows horizontal scalability

- **Remoting**
- **Clustering**

The first allows messages to be sent over a cluster for externalized workflow. This allows each actor to reduce its complexity, however it increases the communication (trade-off)

The second allows (among other things) sharding of actors and even distributed data

Simple API

```scala
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)

  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

Partial function defining
its model using pattern matching on received messages,
but doesn’t return anything (“void”).

```scala
val myActor:ActorRef = system.actorOf(Props[MyActor], "myactor")
```

Instances are created by the system using Props.
The client code can only use an ActorRef which is a
proxy to send message to the real actor underneath

```scala
myActor ! "test"
```

“!” means “send”
Here, we send the message “test” to the actor “myactor”.

http://doc.akka.io/docs/akka/2.4.2/scala/actors.html

Request / Response (HTTP)

HTTP client/server implementation using Akka as the core engine

Providing DSLs for creating routing, utils for un/marshalling and more

```
val route =
  get {
    path(Segment / Segment) { (pathSegment1, pathSegment2) =>
      parameters(“param1”.as[Long], "param2".as[String]) {
        (param1, paramm2) = complete()
      }
    } ~
    path("path1") {
      complete(...)
    }
  }

Http().bindAndHandle(route, "localhost", 1111)
```

Streaming

The Akka Stream API is an obvious follow up in the application stack we can create on top of actor.

Akka Stream hence propose abstraction allowing to run executions on a stream in a non blocking and fault tolerant way.

This allows a stream to be consumed using the Source API and to do some manipulations on the data before going to the system.

Queuing

Taking out the pressure

So far …

At this stage we've got a bunch of collectors running

These are putting data like crazy onto the system.

The data is being injected for one purpose: processing

Processing isn't a one time task not is it lightweight

Based on those statements, we cannot rely on collectors!

This is where the Queue comes into the picture

Queue

In a data science pipeline, the queue has a privileged position

It can have a least two roles

- **Make the data available at a given rate for the processing to handle correctly and for instance make it persistent**
- **Make the whole pipeline reactive (or near real time) allowing the data to be streamed throughout the system (eventually more than once, with a different structure)**

Messaging Buffer

In the actual context of pipeline, the queue is actually acting as a buffer

It allows to control the way data is flowing in the system

Queue taking as is hasn't to be only a "Queue" in its theoretical form

The real need is to hold the data at any scale until it is processed

Then a publish / subscribe messaging system will do the job beautifully too

The messaging buffer will have to decouple the producer to the consumer

Queue, again

A queue is using a broker installed on the system to receive messages

It can potentially use a persistent storage to store all messages durably, or at least for a given period

On this broker can register several consumers and each time an event is published only one of the consumers to process it

Publish / Subscribe

Long story short, the P/S pattern can more or less respect the same high level architectural design with respect to ingesting the incoming data

When it comes to consumers however, they are subscribing to the messages which means that all of them will receive all messages

This has the advantage to increase the parallelism at the processing side

Kafka

Distributed Commit Log

Messaging Buffer

We've covered why a queue is necessary for a data science pipeline to keep up with the incoming data

We discussed principally the two kind of system we may consider:

- **Queue: one message is received by only one registered consumer**
- **Publish / Subscribe: all messages are sent to all subscribers**

Kafka

The first word we can tell about Kafka is that he can join both system using the concept of *Consumer Group*

So we can cover all use cases that were adapted to one or the other

But the important of Kafka is its distributed architecture which involves

- **partitions**
- **replicas**

How it works

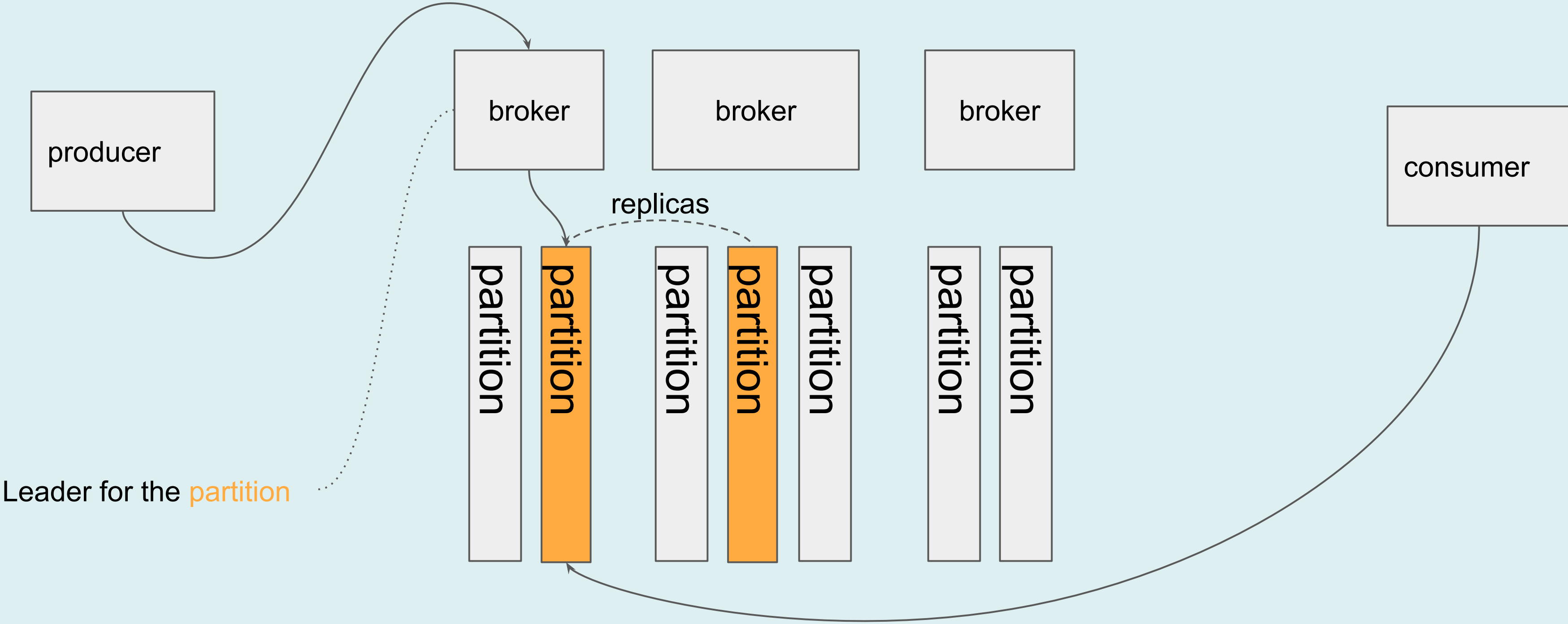
Kafka works with brokers (a classic) however, brokers can be added on the fly to the cluster

A broker will hold partitions of the data published into a topic

A topic is simply a name given to a distributed queue, or more precisely with Kafka, a distributed commit log

This makes Kafka a resilient and fault tolerant system

How it works, cont’d



Interactive Programming

Get Results Quick

Exploratory phase

While dealing with data, there is a phase that you'll never escape from, the exploratory phase

It is crucial to understand the data you're dealing with, this is involving several techniques like plotting, density estimation, correlation matrices, … the list is pretty long

Of course metadata are something to consider as part of this phase, however, most of the time you have none **(even only the data book)**

Computational intensive

The exploration of the data is actually very expensive when it comes to resources usage

The most important support for this is the fact that it'll require a lot of retries of each techniques to find the right angles to look at the data

It's not true to think that training a Machine Learning model will be the most expensive tasks the computing engine has to deal with

As a matter of fact, the model you'll choose is generally based on the discoveries you've made during the exploratory phase

Need for Speed

So, if only even we do only care about the exploratory, phase we have to cope with these points

- **it’s a crucial phase**
- **it’s computationally intensive**

That being said, you can't afford losing much time there using inapropriate tooling

You need both interactivity and reactivity to quickly wrap your head around the result and move forward

Distinction with coding

There is a point to make here on how different are between creating a model and coding an algorithm (or apparented)

When it comes to coding, we can structure our work as an integration with external system hence we can make assumption (or fix constraints) on what we receive, compute and return

When it comes to modelling, it totally relies on the data structure, availability and quality we have at the given time

This is why following a test driven approach is pretty hard here

Options

The two main options for interactive programming are

- **REPL for *Read Execute Print Loop*: which is generally presented as a basic shell accepting "one line at a time"**
- **Notebook: this is an evolution of the REPL which offers an expanded view for the input allowing them to be packaged and rerun all at once. This is an open door to reproducibility of the defined experiences. The classic UI is a browser these days**

The third option would be to use BI kind of tools, that are powerful but also have limited integration within the pipeline

Spark Notebook

Notebook for Scala and Spark

Project

The Spark Notebook is the ultimate open source interactive interface to hack code in Scala and data analyses using Spark.

You can find the Spark Notebook on [GitHub] and is already one of the most important project in the Big Data ecosystem, especially in the Apache Spark's ecosystem with more than a thousand star.

Getting started with the Spark Notebook is a three steps task:

- **grab or build one distro on spark-notebook.io** (zip, tgz, deb, docker)
- **unpack and run it locally**
- **try one of the 50 examples**

Interactive Data Science @ Scale

The Spark Notebook is filling up the gaps in data science that kept data scientists using an enterprise ready language like scala that runs on the JVM.

It's close integration with Spark coupled with the reactive plotting features, the Spark Notebook allows efficient exploration of Big of Fast Data. No more (static) samples!

Also, one of its most important peculiarities is that a notebook has its own JVM started and all opened UIs will synchronize to display the same outputs.

Helpful tricks

The spark notebook plotting capabilities are by default reactive allowing data to be appended on the fly from scala to update the plot dynamically. This includes **streaming**.

A close integration with scala and spark includes the usage of metadata for each notebook to define new libraries, JVM properties, repositories and **Spark Configuration**!

One another great feature, helpful for debugging, is that all (incl. spark) logs are both in the server file and **the browser console**.

Ingestion

Collect data into Kafka

Getting hands dirty

Use notebook: 01_Yahoo Quotes.snb

We'll consume live Yahoo's quotes using an akka collector which produces timestamped data into Kafka!

Assessment #1

Consume Twitter

Twitter4J → Akka → Kafka

Use the notebook provided in the slack room: <u>Assessment 1.snb</u>

In the notebook dashboard, find the <u>Click Here</u> link and use it to import the snb file from your computer

Then click on <u>Upload</u>

Tasks

The tasks are described within the notebook, look for the Tasks section

Long story short, you'll have to create a new topic in kafka, then fill the missing code in the notebook
The missing code are represented using triple question marks *???*. In scala, it represent a "Not Implemented Exception", hence will throw it executed

<u>Note:</u> if something goes wrong with the code you can always reload the kernel using the refresh arrow 
(if a cell got stuck, you can also refresh the browser page)