

Spark on Amazon EMR in a nutshell

Authors: [Lorenzo Mario Amorosa](#), [Mattia Orlandi](#), [Giacomo Pinardi](#)



Overview

In this article we will discuss about our experience with [Amazon EMR](#), the Amazon platform for processing **big data** using open source tools such as **Apache Spark**.

Our main goal is to shed light on effective **software engineering** practices that helped us in the development of our project and troubleshooting.

Project: Docx Anonymizer

Our aim was to design and implement a software tool capable of **anonymize** complex and formatted OOXML Document (**DOCX**), exploiting all the advantages that distributed computation can give.

The starting point was a **Java** codebase developed by our team ([reference here](#)), useful to process DOCX Document and anonymize its content sequentially. On top of that, we created a **Scala** program based on **Apache Spark** ([reference here](#)) to perform the same task in a parallel fashion.

Mixed Java-Scala project

Mix up the languages

Different programming languages do not often cooperate well together, but **Java and Scala** can be integrated because their code works on **JVM in both cases**. This feature enhances the **portability** of well-spread **Java libraries** in modern and effective Scala code.

Module dependencies

In a mixed project, it is possible to pass a Java object to Scala function and vice versa, exploiting the reflection mechanism. Anyway, working in Scala with Java code is completely transparent, but working in Java with a Scala object is not that easy for the notation required. As a consequence, we recommend to have **only Scala code** that **depends on Java** classes and libraries (e.g. write your `main` in Scala).

Development tools

A fundamental aspect regards the choice about the tools to use during the development. There are plenty of **build automation tools** (Maven, Gradle, Ant, SBT, etc.) that perform routine tasks, such as export a project to a jar file and manage the library dependencies.

We started our set up with Maven as we are proficient Java developers used to it, but then we mind the fact that *every tool has its own purpose*. The best choice in a Scala project is to use **SBT (Scala Build Tool)** because it was born as a Scala tool, whereas the other software are well designed for other kind of Java projects and need several plugins to work with Scala.

Project structure

A well design **project structure** for a Java-Scala mixed project with SBT and dependencies management for Apache Spark should look as follows ([reference here](#)):

```
build.sbt
project/
|-- plugins.sbt
src/
|-- main/
|   |-- java/
|   |-- resources/
|   |-- scala/
|-- test/
|   |-- java/
|   |-- resources/
|   |-- scala/
target/
```

In the configuration files it must be taken into account:

- **Hadoop** and **Spark versions** available on AWS EMR (e.g. for EMR 5.29.0, it can be used at most Spark 2.4.4 on Hadoop 2.8.5).
- **JDK** available on AWS EMR, that constrains the compilation routine (e.g. for EMR 5.29.0 it is JDK-8, so we were forced to not use libraries based on Java 9+ and to use Scala 2.11).
- Version of **libraries** that your code has **in common** with Hadoop and Spark already use (e.g. Hadoop uses `commons-cli` 1.2, our code is compliant with it).
- **Libraries** already **provided** on the cluster must not be exported into the user jar file (e.g. `org.apache.spark` itself).

build.sbt:

```
name := "DocxAnonymizer"
version := "0.1"
scalaVersion := "2.11.12"

val sparkVersion = "2.4.3"

javacOptions ++= Seq("-source", "1.8", "-target", "1.8", "-Xlint")
scalacOptions := Seq("-target:jvm-1.8")

assemblyMergeStrategy in assembly := {
  case PathList("META-INF", _) => MergeStrategy.discard
  case _ => MergeStrategy.first
}
```

```
libraryDependencies += Seq(
  "org.apache.spark" % "spark-core_2.11" % sparkVersion % Provided,
  "org.docx4j" % "docx4j" % "6.1.2",
  "commons-cli" % "commons-cli" % "1.2",
  "com.amazonaws" % "aws-java-sdk" % "1.11.775" % Provided
)
```

project/plugins.sbt:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.10" )

resolvers += Resolver.url("bintray-sbt-plugins",
  url("http://dl.bintray.com/sbt/sbt-plugin-releases"))(Resolver.ivyStylePatterns)
```

First deploy

We provide a **sample project** [here](#) that is already **configured** to work with **Amazon EMR 5.29.0**. You can download it, open it in your IDE (we recommend [IntelliJ IDEA](#)) and export it as an executable **jar** file. Be aware that you have to install **Scala plugin** and **sbt** in your IDE. In IntelliJ you can export the project as follows:

```
"View">"Tool window">"sbt">"SparkTemplate">"SparkTemplate (root)">
"sbt tasks">"assembly"
```

and then your jar file will be located in `target/scala-2.11`

Now you are ready to test your new project on a cluster!

Set up a cluster on AWS EMR

In this section we will show how to deploy an application on Amazon EMR.

Account creation

It is highly recommended to create an AWS Educate account in order to obtain 30\$ of credit for free. This “gift” will be useful to perform a great number of tests with the AWS cloud platform.

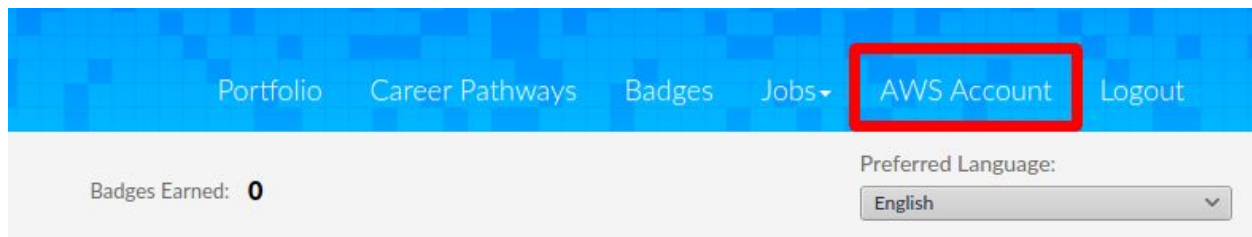
To create an AWS Educate account follow this link ([AWS Educate](#)) and click on “Join AWS Educate”. Then you need to fill in your information, in particular if you are studying at Unibo select “University of Bologna” in the field “School or Institution Name”.

After having received the confirmation email you can log in from the AWS Educate main page, clicking on “Sign in to AWS Educate”.

Cluster creation

From the AWS Educate page, after the login, we need to reach the Console page.

Click on “AWS Account” in the top-right corner:



Click on “AWS Educate Starter Account”:

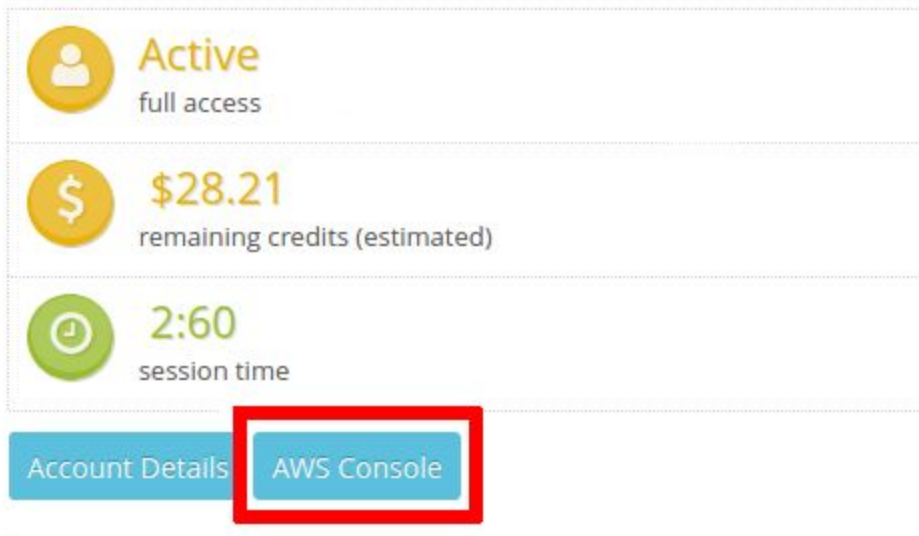
Your cloud journey has only just begun. Use your AWS Educate Starter Account to access the AWS Console and resources, and start building in the cloud!



Your account has an estimated **28** credits remaining and access will end on **Apr 26, 2021**.

You will be redirected to a new page. This page is useful to check your remaining credits. Click on “AWS Console”:

Your AWS Account Status



You will be redirected to a new page again. This is the AWS Console and from here we can manage all the AWS services that are provided. In particular we are interested in S3 and EMR services, as we will see. To find a service just type its name in the text box and select the correct element from the list.

S3 Service

From the [AWS Docs](#): *Amazon Simple Storage Service is storage for the Internet. It is designed to make web-scale computing easier for developers.*

Using S3 we will be able to store our application and all the required files.

First we need to create a Bucket. A Bucket is a container that stores your files. Click on the “Create bucket” button and follow the procedure. The default options are fine for what we need.

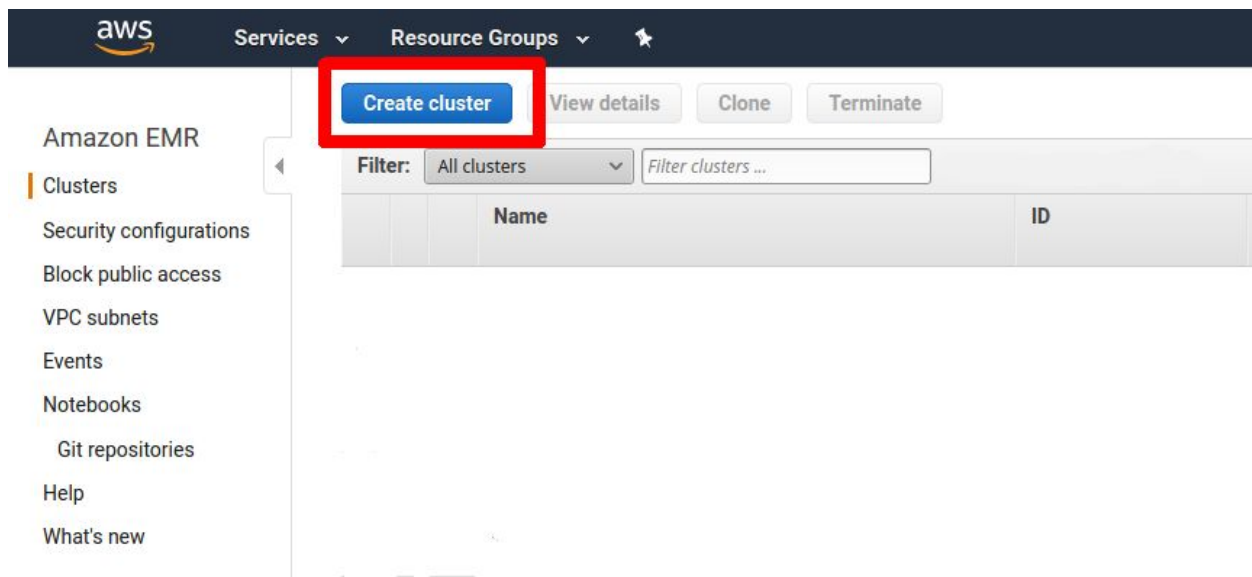
After the creation of the bucket we can upload files to it selecting the bucket name and clicking on the “Upload” button. Again we can keep the default options unchanged.

EMR Service

From the [AWS Docs](#): Amazon EMR is a managed cluster platform that simplifies running big data frameworks, such as Apache Hadoop and Apache Spark, on AWS to process and analyze vast amounts of data.

Using EMR we can create a cluster instance and use it to run our application and exploit the parallelization features offered.

From the EMR service page click on the button “Create cluster” in the top-left corner:



Then you will need to fill in all the details and the configuration information needed to run the cluster. You can leave the “General Configuration” section as is (just change the cluster name if you want) and move on the “Software configuration” section. We need to select the last option, the one containing Spark:

Software configuration

Release emr-5.29.0 ⓘ

Applications

- ☐ Core Hadoop: Hadoop 2.8.5 with Ganglia 3.7.2, Hive 2.3.6, Hue 4.4.0, Mahout 0.13.0, Pig 0.17.0, and Tez 0.9.2
- ☐ HBase: HBase 1.4.10 with Ganglia 3.7.2, Hadoop 2.8.5, Hive 2.3.6, Hue 4.4.0, Phoenix 4.14.3, and ZooKeeper 3.4.14
- ☐ Presto: Presto 0.227 with Hadoop 2.8.5 HDFS and Hive 2.3.6 Metastore
- ☒ Spark: Spark 2.4.4 on Hadoop 2.8.5 YARN with Ganglia 3.7.2 and Zeppelin 0.8.2

☐ Use AWS Glue Data Catalog for table metadata ⓘ

In the “Hardware configuration” section you can change the Instance type (improve node performance and memory availability) and increase the number of nodes that will run your application. You can change them, just keep in mind that the running cost will largely depend on what you choose in this section.

In the “Security and access” section you need to select an EC2 key pair from the list. If you haven’t yet, you can create an EC2 key pair very easily following the instructions that we provide in the next section. It’s important to highlight the fact that the key pair must be provided at cluster creation time, since it won’t be possible to add it later after the cluster is created.

Then click on “Create cluster”. The cluster will be created (it will take some minutes).

Add Step to cluster

Now we want to add a Step to our cluster. A Step is a chunk of computation that we want to perform. This can be achieved in two ways:

Web interface

From the cluster summary click on “Steps”:

Cluster: My cluster **Starting**

Summary Application history Monitoring Hardware Configurations Events **Steps** Bootstrap actions

Concurrency: 1 [Change](#)

After last step completes: Cluster waits

[Add step](#) [Clone step](#) [Cancel step](#)

Filter: All steps 1 step (all loaded) [Refresh](#)

	ID	Name	Status	Start time (UTC+2)	Elapsed time
<input type="radio"/>	s-31PG4N5W75KFJ	Setup hadoop debugging	Pending		-

You can see that there is already a Step running: it is the step that automatically configures the cluster. Now click on “Add step” to run a custom application on the cluster.

Add step [Close](#)

Step type: Spark application

Name: MyStepName

Deploy mode: Cluster

Spark-submit options: --class pkg.SparkPi

Application location*: 518974-us-east-1/code/SparkTemplate-assembly-0.1.jar

Arguments: 3

Action on failure: Terminate cluster

[Cancel](#) [Add](#)



In the menu select “Spark application” and insert a name for this particular Step. Is is important to specify the option `--class pkg.SparkPi` in the “Spark-submit options”, in this way Spark is able to retrieve the Application’s Main Class and run your application (note that `pkg.SparkPi` is the Main Class of the template project proposed).

In the field “Application location” we must specify the location of the `.jar` previously uploaded in our S3 Bucket.

We want also to specify an argument for our application and so we add “3” in the “Arguments” section.

Finally we suggest to select the “Terminate cluster” option in the last combo box. In this way if for any reason the step fails, then the cluster will be automatically terminated.

We can then click on the “Add” button and wait until the execution ends.

	ID	Name	Status	Start time (UTC+2) ▼	Elapsed time	Log files 
 ▼	s-2XIUPULWSM8TC	MyStepName	Completed	2020-05-11 23:37 (UTC+2)	30 seconds	View logs
JAR location : command-runner.jar						
Main class : None						
Arguments : spark-submit --deploy-mode cluster --class pkg.SparkPi s3://aws-logs-632872518974-us-east-1/code/SparkTemplate-assembly-0.1.jar 3						
Action on failure: Terminate cluster						

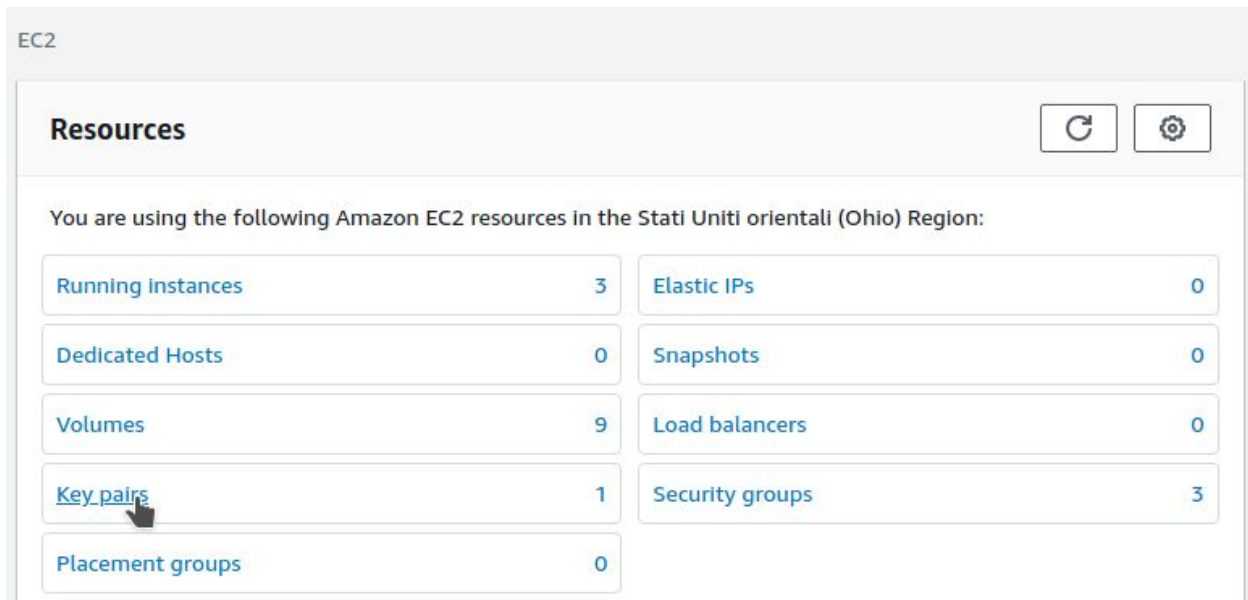
When the step is completed we can retrieve the result of the computation (in this example the Pi computed using a Monte Carlo approximation) from the Step summary, clicking on “View logs” and then on “stdout”.

It is also possible to find the stdout of the executed application in the S3 Bucket. Just locate the Bucket in which are saved the logs of the current Cluster (this Bucket is specified during the cluster creation). In the folder `elasticmapreduce/clusterID/steps/stepID` you will find stdout and stderr together with other useful files. Where `clusterID` and `stepID` can be found in the cluster summary page.

The stdout can also be found by exploring the “Application History” section.

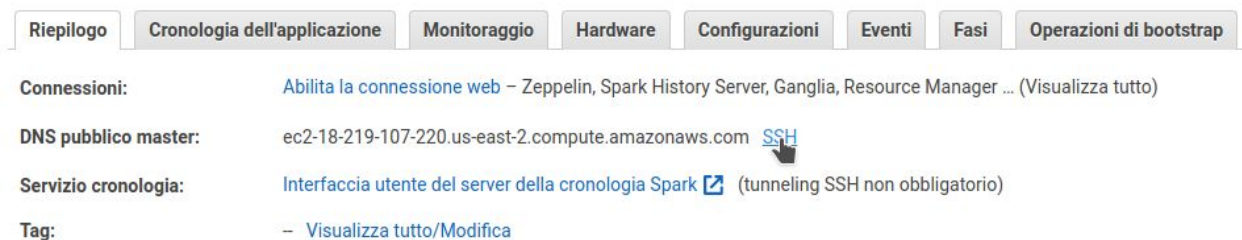
SSH

Create a pair of SSH keys from EC2 dashboard:



Once created, you will be asked to download the `<key_name>.pem` file and to store it in a safe folder (it is suggested to save it in `~/ .ssh`).

To connect to the cluster through SSH, use a command of the form `ssh -i <path_to_pem> hadoop@<cluster_address>`, where the `<cluster_address>` can be found by clicking on the “SSH” link in the main tab of the cluster web page:



In case you get a connection timeout when you try to establish an ssh connection to your EMR server, you will probably need to configure a ssh rule. Go to the security group is in EC2 dashboard:

1. Navigate to EC2 dashboard -> security group.
2. Find group ElasticMapReduce-master and click on it-> Inbound -> Edit -> Add rule.
3. Add ssh, choose My IP as source, and save the rule.
4. Now you should be able to ssh to the master node.

Mind the fact that your private key must have read-only permission (e.g 400) to be accepted by the remote server.

Once connected to the EMR cluster via ssh, to add a step is sufficient to issue the command `spark-submit --deploy-mode cluster --master yarn --class <package>.<main_class> <path_to_jar> <program_arguments>`. Mind the fact that you have to previously upload your `.jar` (e.g. using `scp`) on your cluster.

Account Upgrade

Several functionalities are not available for AWS Educate. You will need a **standard account** in case you want:

- a **more scalable system** with more powerful nodes and instances.
- **IAM account** to effectively take advantage of Amazon S3 capability and work well in a group (take a look [here](#) to see how to use AWS API for Scala).
- choose your **cluster location**.

Scala Best practices

In this section we want to highlight the best practices to follow when building an app in Scala.

Avoid null values and exceptions

Option[T]

This type allows to completely get rid of `null` values, by encapsulating variables that *could* be `null` in an `Option` object: `val <var_name>: Option[<var_type>] = Option(<value>)`; by doing this, potential `null` values are automatically translated to `None` objects.

Such mechanism enables also the usage of the `match` pattern:

```
Option(<value>) match {  
  case None =>  
    // Actions to perform in case of None  
  case Some(<value>) =>  
    // Actions to perform in case of significant value  
}
```

Either[T, U]

This type allows to return a meaningful result from a function which may fail or not:

- in case of success, the function returns a `Right[U]` object, which contains the expected result (even `Right[Unit]`, i.e. `Right()`);
- in case of failure, the function returns a `Left[T]` object, which contains the cause of failure (e.g. an error message `Left[String]`).

`Either` type also enables the usage of the `match` pattern:

```
val <function_result>: Either[<type_failure>, <type_success>] =  
    <function_returning_either>(<parameters>)  
  
<function_result> match {  
    case Left(<fail_value>) =>  
        // Actions to perform in case of failure  
    case Right(<success_value>) =>  
        // Actions to perform in case of success  
}
```

Try[T]

This type can be seen as an `Either[Throwable, T]`, i.e. the type in case of failure is an exception. It is used to encapsulate exceptions in a functional-style way, allowing `match` pattern:

```
val <function_result>: Try[<type_success>] =  
    Try<function_throwing_exception>(<parameters>)  
  
<function_result> match {  
    case Failure(<exception>) =>  
        // Actions to perform in case of failure  
    case Success(<success_value>) =>  
        // Actions to perform in case of success  
}
```

Java integration

To integrate the Scala program with Java classes, conversion functions are available by importing `scala.collection.JavaConverters._` (if scala version is `<=2.12.x`) or by importing `scala.jdk.CollectionConverters._` (if scala version is `2.13.x`).

To convert from Java to Scala, it is sufficient to call the `.asScala` method on a Java collection, whereas to convert from Scala to Java, you must call the `.asJava` method on a Scala collection.

Please note that, since Java Lists are mutable whereas Scala Lists are not, calling `.asScala` on a Java List produces a Scala Mutable Buffer, which can then be converted to a Scala List by calling `.toList`.

Troubleshooting

In this subsection we want to address several nasty problems that occurred to us in the development and our strategies to overcome them.

Working on top of Spark

- Your Scala main class must explicitly declare a `def main` method (i.e. extending Scala class `App` does not work).
- In your `main` method be sure that the first instructions executed are the ones to initialize a `SparkContext` instance and that the last ones are those to stop it.
- When your Spark context is active you are not allowed to execute some system calls, such as the Java instruction `System.exit()`. It makes sense, because you are **not working only on** top of a **JVM**, but **also on** top of a **framework**. As a consequence, you can only terminate your program if your `SparkContext` has been closed.
- When you map a function on a distributed collection (i.e., an `RDD`), make sure that every object used inside that function is `Serializable` (even the objects that provide a method used inside such function); in case such objects are not serializable (for instance, they present a field whose type is declared in a third party library and is not serializable), there are two options:
 - exclude such fields from serialization (**be careful**, since it may break some functionalities) by marking them as `transient` (in Java) or by marking them as `@transient lazy`/by applying `call-by-name` (in Scala);
 - wrap such fields in a custom object which is declared `Serializable`, and manually serialize/deserialize them (i.e. using `gson` library).
- You cannot have nested `RDD`s, i.e. it's not possible to interact with an `RDD` inside the `map` function of another `RDD`.

-
- The changes made to variables inside the `map` function of an `RDD` are local w.r.t. Spark nodes, and are not seen by the driver after the map operation is completed (i.e. such variables are **read-only** if used by the nodes in an `RDD` map function); therefore, it's better to wrap such variables in a `Broadcast` type, s.t. an exception is raised if you try to modify it.
 - The `Accumulator` type can be used to store information inside `RDD` map function and use such information later globally, so it's like a "**write-only**" variable (for numeric types only).
 - Try to do not rely on **AWS EMR logging system**. The log files of each Spark execution require **several minutes** to be created and they will be accessible on your cluster only after a while. To avoid waste of time, you can create and write your own log files in your Spark code and then upload them on your S3 bucket.
 - Note that **debugging using print command** on stdout is **ineffective** because the code is executed remotely and you do not have direct access to the Spark shell.

Dividi et Impera

Programming with Spark can be perceived as a stressful task, especially in case you have to rely on a remote cluster. In fact, you are required to set up different clusters and steps for each test execution and diving deep into AWS EMR logs can be really uncomfortable and complicated.

Our advice is to **incrementally build prototypes** and **deploy** them on the cluster. You can start from a [simple project](#) with a `main` class only and then add other code and libraries step by step.

In this way it is easier to **figure out problems** involving software dependencies and version conflicts due to interaction between your code and Spark framework, which is often the case if you have to deal with large codebase.