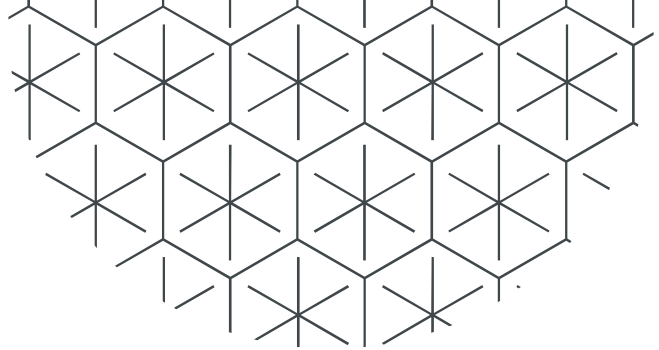
Tecnologías de datos masivos

Doble Grado en Ingeniería en Tecnologías de Telecomunicación y Business Analytics



Apache Spark

Spark - Historia

- En 2009 Matei Zaharia inventa un motor de procesamiento distribuido basado en el uso intensivo de memoria en lugar de disco como tesis doctoral
- Spark nace el mundo de Hadoop pero con las lecciones aprendidas y una mejora tecnológica
- En un principio Spark “sólo” soluciona problemas de procesamiento en batch
- Cuando se extendió su uso y se trató de solucionar más casos de uso con la misma tecnología, se hizo siempre siguiendo la premisa: adaptar el problema a la arquitectura de Spark
- En la actualidad Spark es el estándar de facto para el procesamiento de datos masivos.

.

Spark - Arquitectura

- Spark es “sólo” un motor de computación distribuida
- Sigue una arquitectura Maestro/esclavo
  - Driver: proceso maestro que se encarga de orquesta la ejecución de las aplicaciones de Spark.
  - Executor: procesos esclavos que se encargaran de ejecutar el trabajo que ha sido paralelizado en Tasks.
- Delega la coordinación y la distribución física cada uno de estos componentes a los **gestores de recursos**

Spark Arquitectura - Driver

- Es el Main de nuestra aplicación y sólo habrá uno por aplicación de Spark
- Se describen las interacciones con los datos pero no realiza cómputos distribuidos.
- Para gestionar su memoria habrá que tener en cuenta cuantos datos devuelven nuestras aplicaciones o cuantos datos tenemos que tratar antes de distribuir la información
- Negociará los recursos para sus Executor con el gestor de recursos que hayamos elegido para su ejecución
- Su log aparecerá en el nodo donde se ejecute

Spark Arquitectura - Executor

- Será el encargado de ejecutar el código distribuido de nuestra aplicación
- Ejecutará las interacciones con los datos definidas en el Driver
- Para gestionar su memoria habrá que tener en cuántos elementos hay por partición y que se hace con ellos
- Los recursos serán los mismos para todos los executors de un cluster de Spark, pero pueden, y de hecho suelen, diferir de los recursos del Driver
- Su log aparecerá en el nodo donde se ejecuten

Spark Arquitectura - Recursos

- Las tareas en paralelo se ejecutarán en los nodos Worker donde se hallaran los Executors
- A nivel de recursos se pueden definir las siguientes características por Executor.
  - Cores: Son aquellas tareas que se van a ejecutar en paralelo por Executor
  - Memoria: Es la cantidad de memoria que se le asigna para resolver todas las tareas de Spark por executor
- A nivel de recursos se pueden definir las siguientes características por Driver
  - Memoria: Es la cantidad de memoria que se le asigna para resolver todas las operaciones del Driver

Spark Arquitectura - Conceptos

- Cada programa realizado en Spark se conoce como **Aplicación**
- Cada **Aplicación** tiene un único **Driver** y varios **Executor**
- Los recursos de varias aplicaciones son gestionados por los gestores de recursos
- La interacción del **Driver** con los distintos recursos del cluster ser realizan por medio del **SparkContext**(Contexto de Spark) para los recuros *core* de una aplicación
- La interacción del **Driver** con los distintos recursos del cluster ser realizan por medio del **SparkSession**(Sesion de Spark) para los recuros *sql* de una aplicación

Spark Arquitectura - Terminología

- **Application:** aplicación realizada por un usuario. Cada **application** está compuesta por uno o varios **job**, pero siempre tiene que haber uno por lo menos
- **Application jar/Application pyfiles:** Jar o ficheros pythons que contienen todas las librerías y ficheros necesarios para el application
- **Driver:** Proceso “main” de una aplicación de Spark
- **Resource Manager:** proceso externo que se encargue de gestionar los recursos del sistema donde se ejecuta la aplicación de Spark
- **Deploy mode**: Modo en los que se ejecuta el **Driver** dentro del cluster gestionado por el **Resource Manager**. Puede ser **client** o **cluster**

Spark Arquitectura - Terminología

- **Job:** conjunto de **stages** agrupados hasta llegar a una **action**
- **Stage:** Conjunto de **tasks** que son ejecutadas en un mismo **worker**. Estas **tasks** se agregan desde que un dato es consumido por el **job** (ya sea desde un repositorio externo o desde un **stage** anterior) hasta que se requiere que haya una fase de **shuffle**
- **Task:** Operación que se realiza sobre cada una de las **partitions**
- **Partition:** Cada una de las distintas particiones de un **RDD**
- **RDD**: Conjunto de datos distribuido

Gestores de recursos

Gestores de recursos - Conceptos

- Spark es un framework de procesamiento que “sólo” se encarga de la ejecución de tareas distribuidas
- Deriva la gestión de los recursos a otras tecnologías encargadas
- En su origen en una prueba de concepto de un gestor de recursos (*Mesos*) es un posible sustituto de YARN
- Estos gestores de recursos serán los encargados de distribuir la carga entre los distintos nodos de nuestro sistema distribuido
- Spark se ejecuta en 4 gestores de recursos, Standalone, Mesos, YARN y Kubernetes
- En proyecto en producción se usan principalmente YARN y Kubernetes

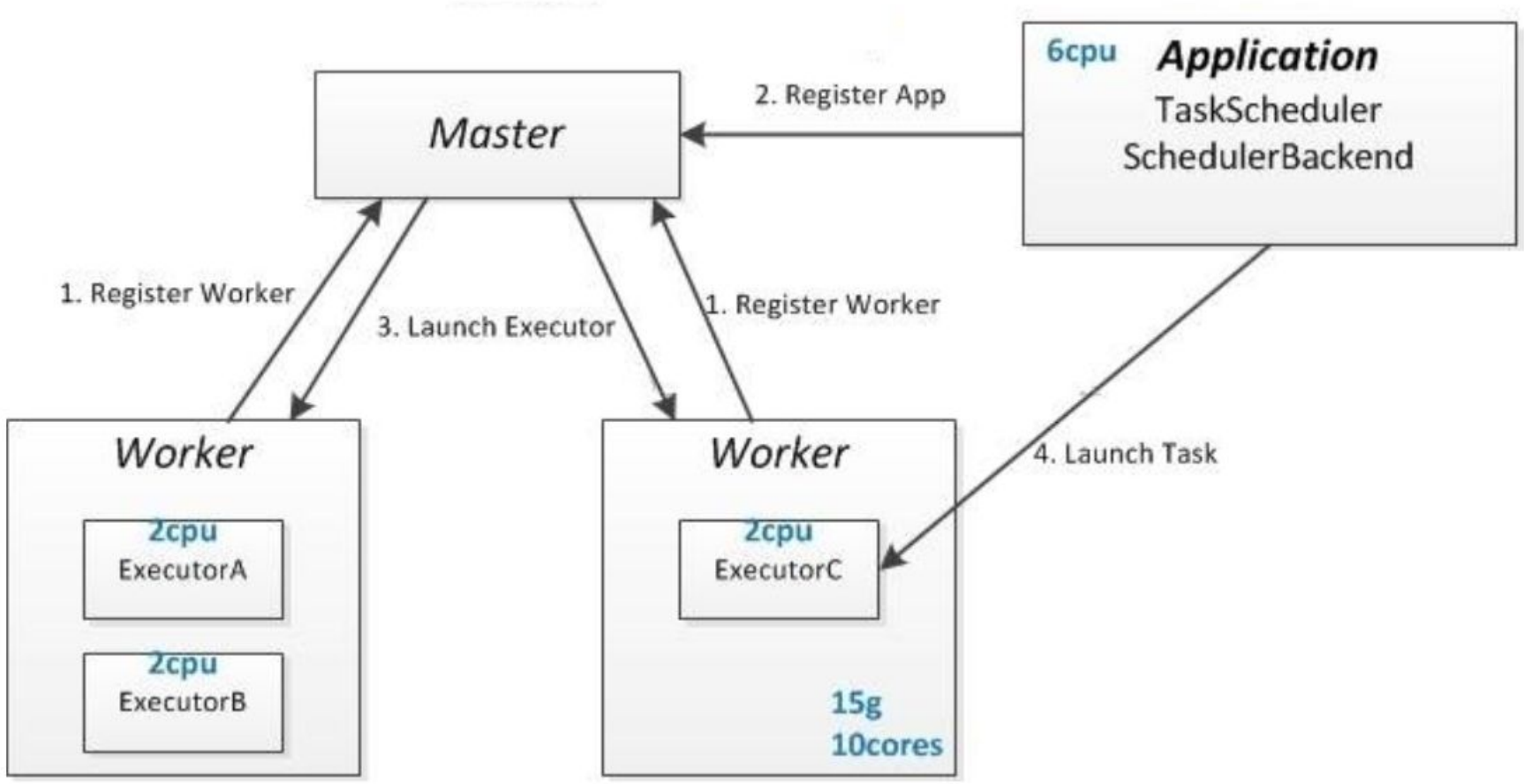
Gestores de recursos - Conceptos

- Spark optimiza la forma de repartir los datos de un problema de procesamiento y un gestor de recursos optimiza los recursos necesarios para su óptimo funcionamiento
- Un cliente de spark normalmente ejecuta el comando spark-submit para solicitar la ejecución de un **application**
- Los recursos que gestiona un gestor de recursos de spark son:
    - Memoria: Se asigna a cada componente de Spark la memoria solicitada en el nodo óptimo tomando en consideración la carga del que el sistema es consciente. Determina cuánta información es capaz de gestionar como máximo cada componente
    - Cores: Se asigna a cada componente de Spark el número de cpus solicitadas en el nodo óptimo tomando en consideración la carga del que el sistema es consciente Determina cuánta información es capaz de procesar en paralelo como máximo cada componente

Gestores de recursos - StandAlone

- El propio proceso de Spark es el encargado de gestionar sus recursos y sólo puede ejecutar aplicaciones de Spark
- Es el más sencillo de mantener y de configurar
- Es el menos flexible y sólo es utilizado para pruebas o casos muy concretos
- Tiene una arquitectura maestro/esclavo
    - **Master:** Es el nodo maestro, se encarga de recibir las peticiones de los **SparkContext** y asignar los recursos necesarios a una aplicación de spark dentro de su cluster
    - **Worker:** Son los nodos esclavos, se encarga de gestioar los **executors** dentro de los cuales se ejecutan las tareas de Spark

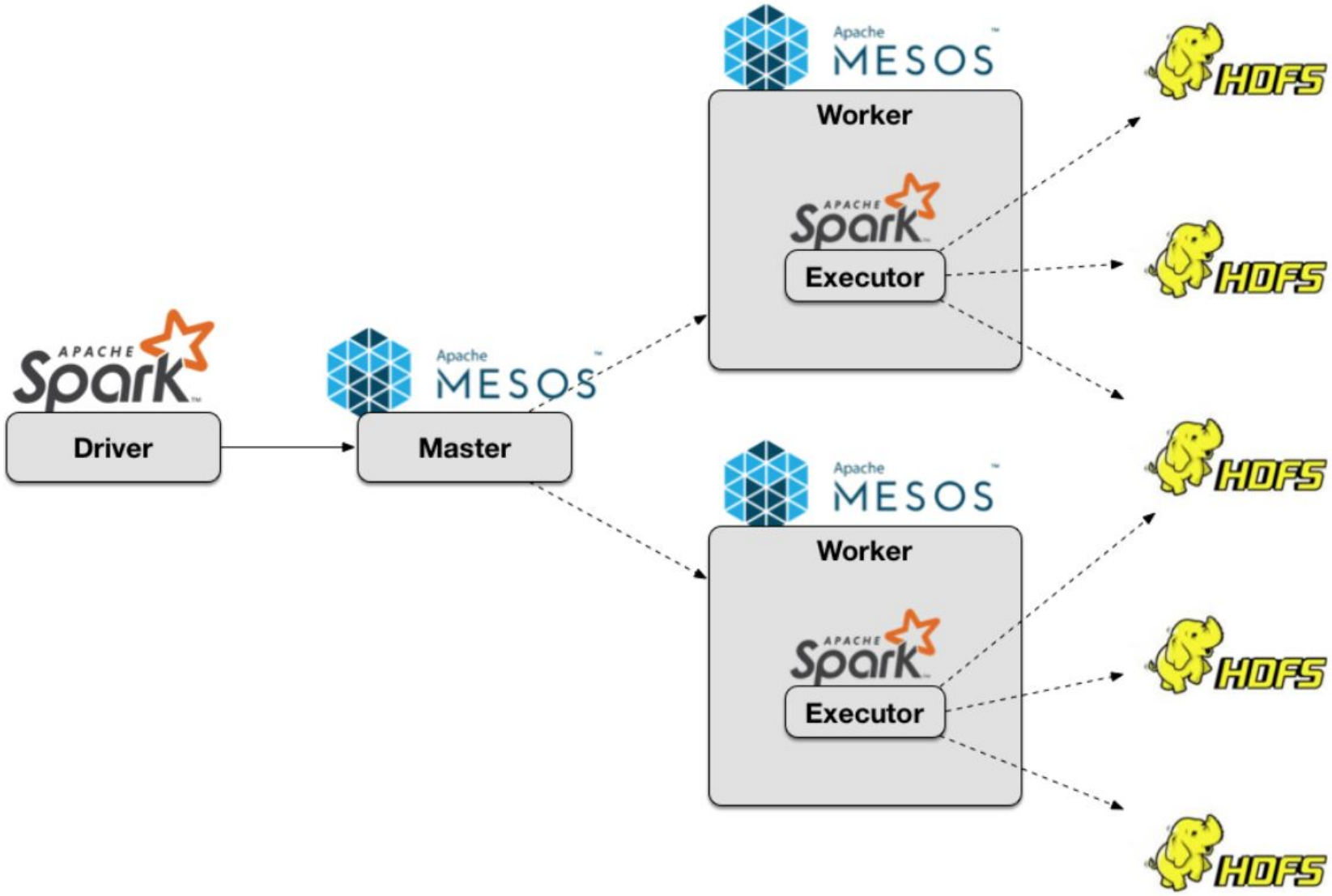
Gestores de recursos - StandAlone



Gestores de recursos - Apache Mesos

- Spark nace como una prueba de concepto de Apache Mesos
- Apache Mesos es un gestor de clusters no de recursos, lo que quiere decir que puede gestionar otros recursos (como las redes de comunicación) además de la memoria o los cores
- Aunque pueda gestionar los recursos de un cluster entero no puede optimizar la ubicación de distintos componentes (un **executor** de Spark con un datanode de HDFS)
- Actualmente está en desuso dado que Kubernetes ha copado el mercado de los gestores de clusters

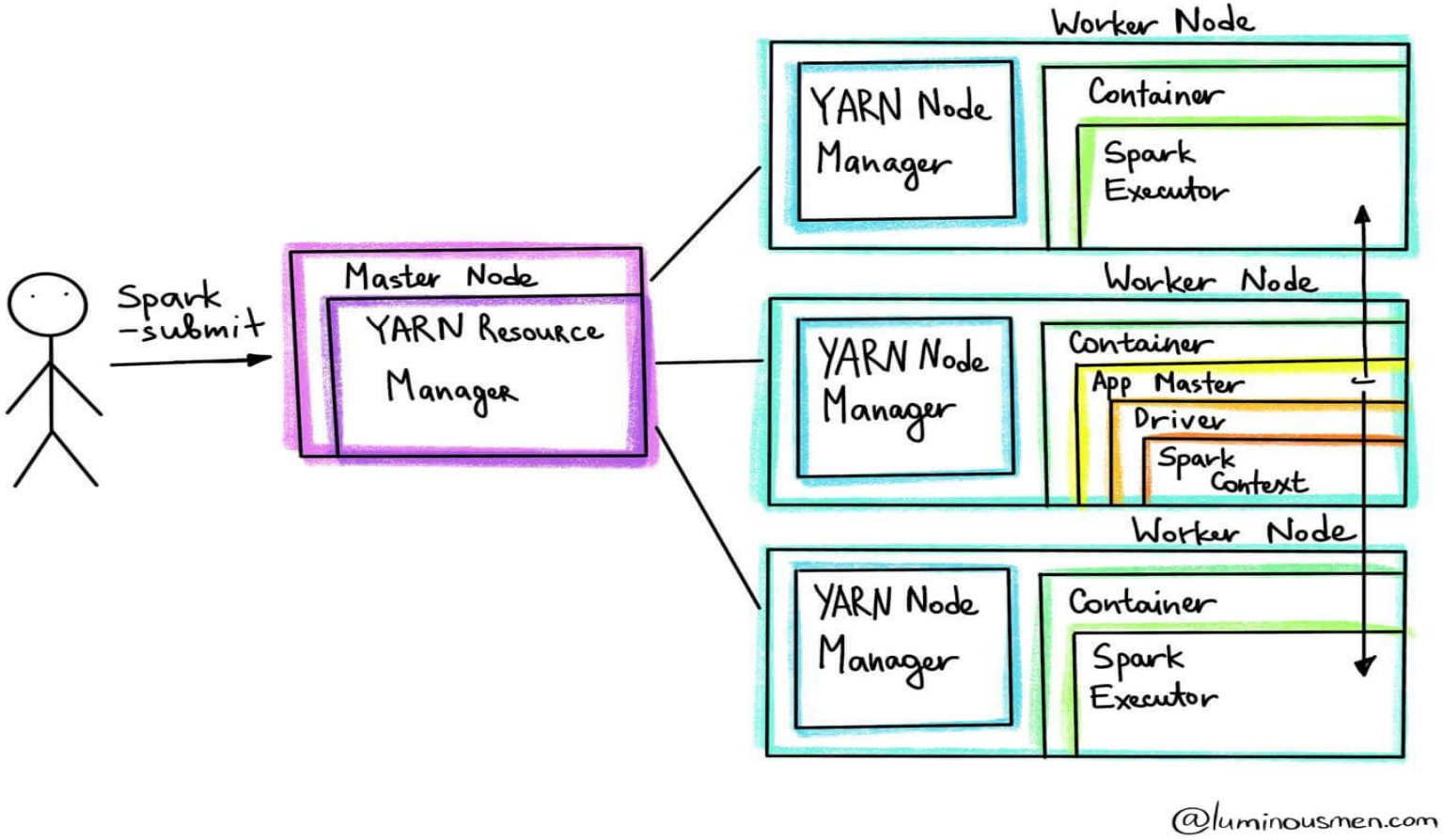
Gestores de recursos - Apache Mesos



Gestores de recursos - YARN

- Es el gestor de recursos más usado en producción
- “Sólo” nos permite gestionar memoria y cpus, aunque en las últimas versiones de Spark se han implementados otros recursos como por ejemplo GPUs
- Utiliza la gestión de recursos de YARN para su ejecución
- Las distintas soluciones Cloud cuando autogestionan trabajos de Spark lo hacen levantando un cluster de YARN
- Nos permite lanzar otros motores de procesamiento distribuido como Map&Reduce, Tez o Flink en el mismo cluster compartiendo recursos

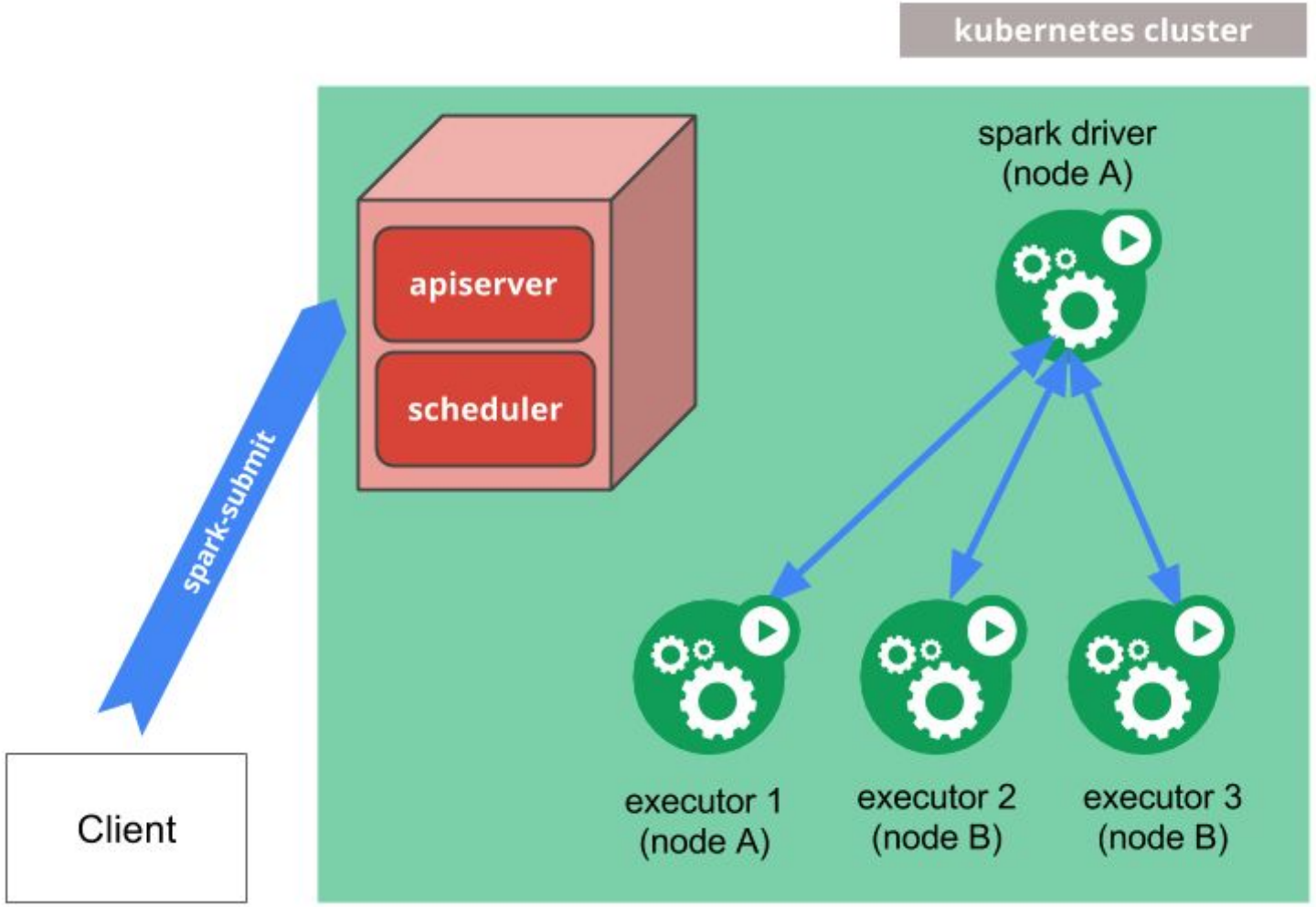
Gestores de recursos - YARN



Gestores de recursos - Kubernetes

- Es el gestor de recursos al que más tarde se ha adaptado Spark
- Kubernetes es un orquestador de contenedores.
- Un contenedor es una virtualización de todo lo necesario para ejecutar un proceso
- Kubernetes está totalmente extendido en el mundo empresarial y desde la adquisición de Google está siendo usado cada vez más en el mundo del procesamiento de datos
- Nos permite lanzar distintos servicios gestionados por el mismo orquestador
- Añade una capa de seguridad necesaria para cualquier proceso productivo

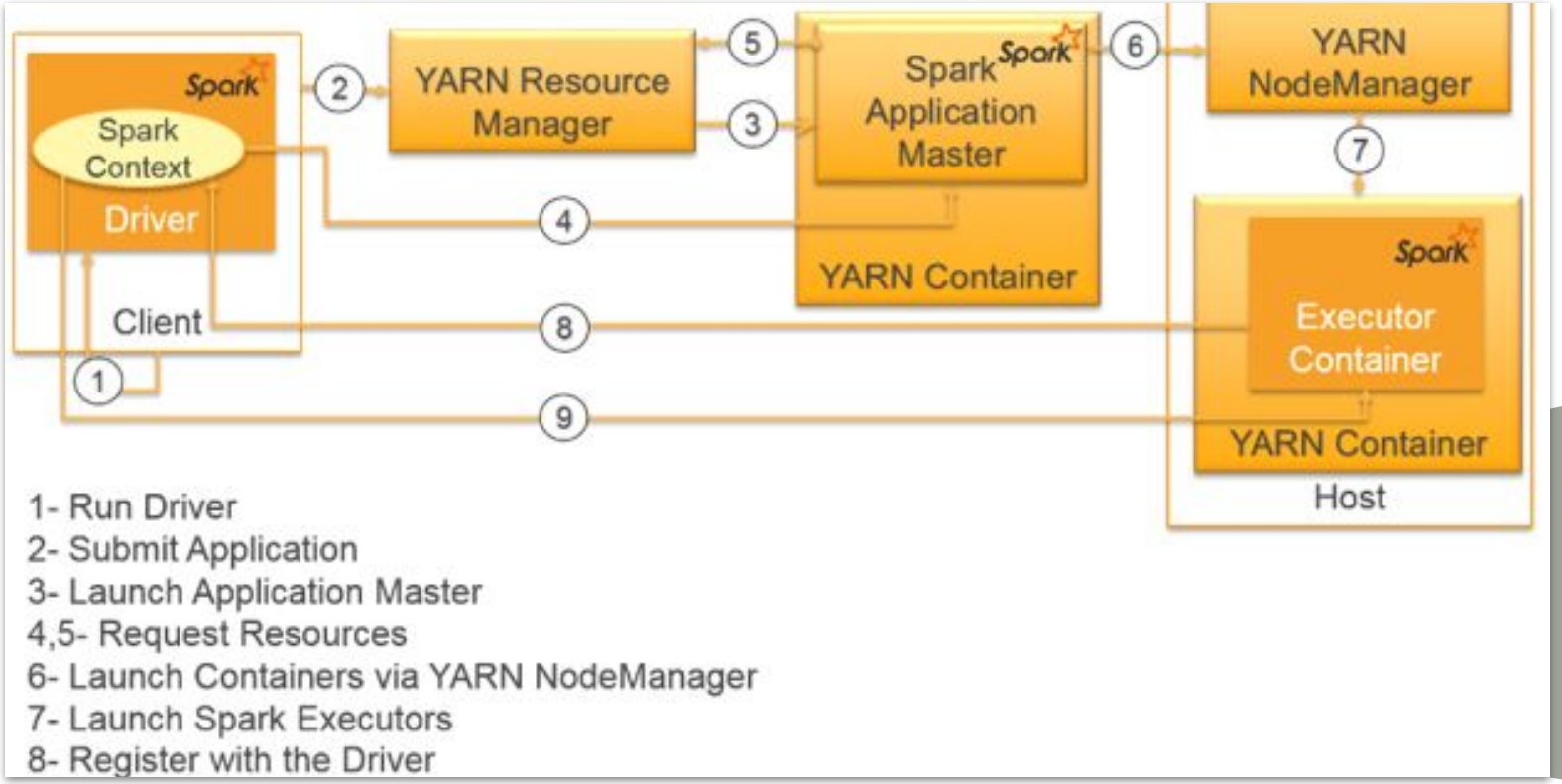
Gestores de recursos - Kubernetes



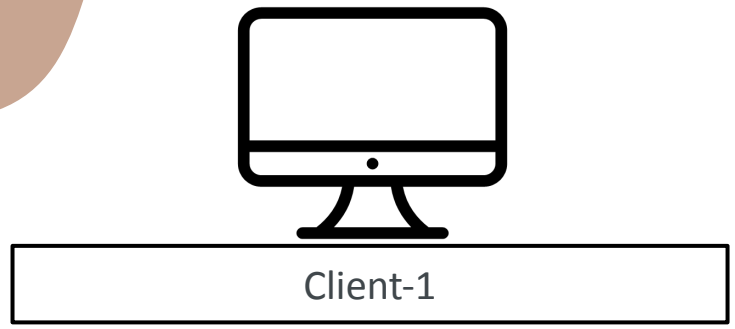
Gestores de recursos - modos de despliegue

- Independientemente del gestor de recursos que use Spark su arquitectura es la misma
- En esta arquitectura el nodo maestro, **Driver**, puede ser gestionado o bien por el mismo proceso o persona que lo ejecuta o delegar su gestión al gestor de recursos
- Si un proceso es gestionado por el gestor de recursos se dice que tiene como modo de despliegue **cluster**
- Si un proceso NO es gestionado por el gestor de recursos se dice que tiene como modo de despliegue **cliente**
- En el mundo de analitica hay el tipo de driver más común es tener un driver con el que interactúa el usuario siendo los dos más usados la shell y los notebooks

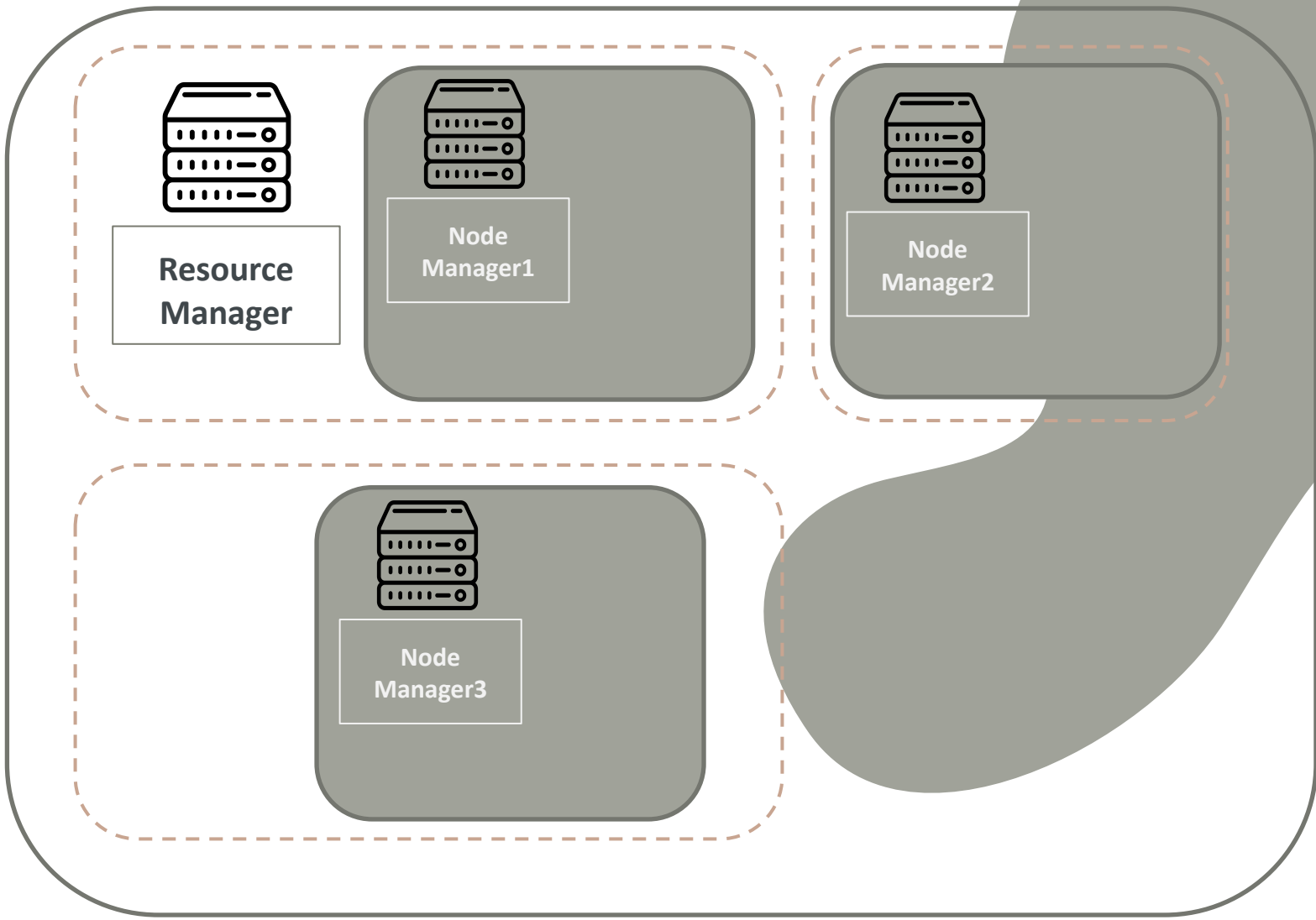
Gestores de recursos - Client



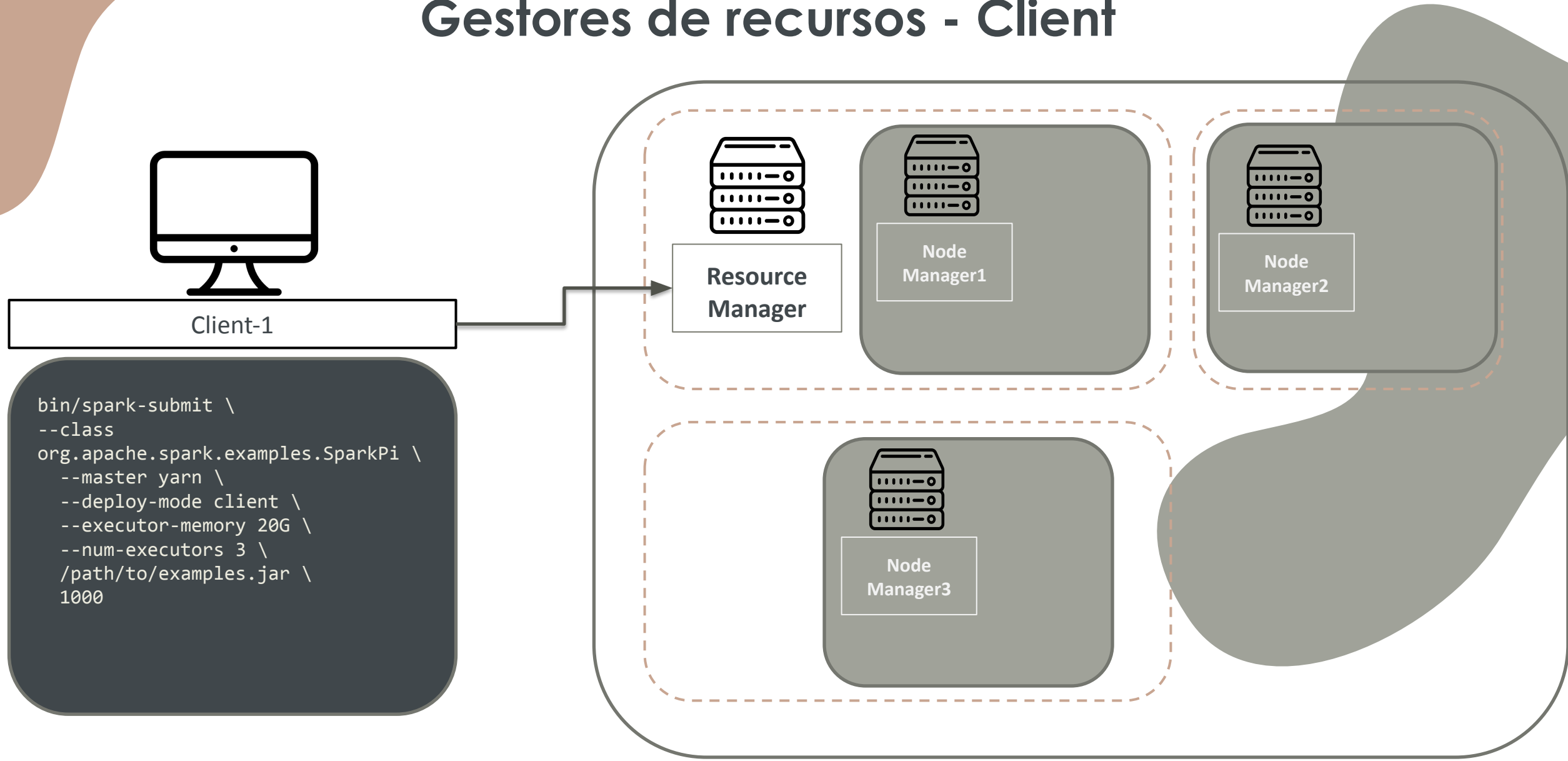
Gestores de recursos - Client



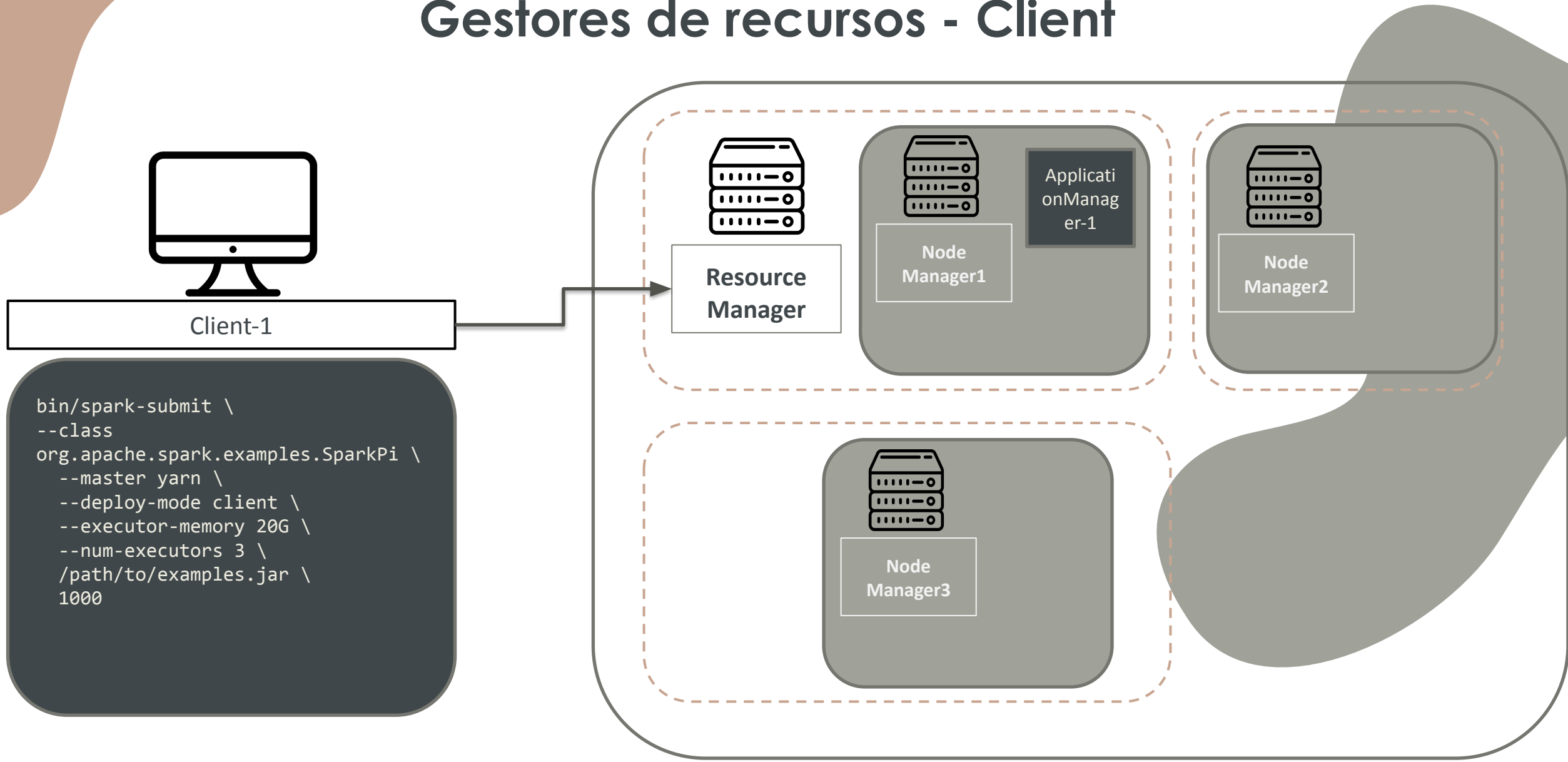

```
bin/spark-submit \
--class
org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode client \
  --executor-memory 20G \
  --num-executors 3 \
  /path/to/examples.jar \
  1000
```



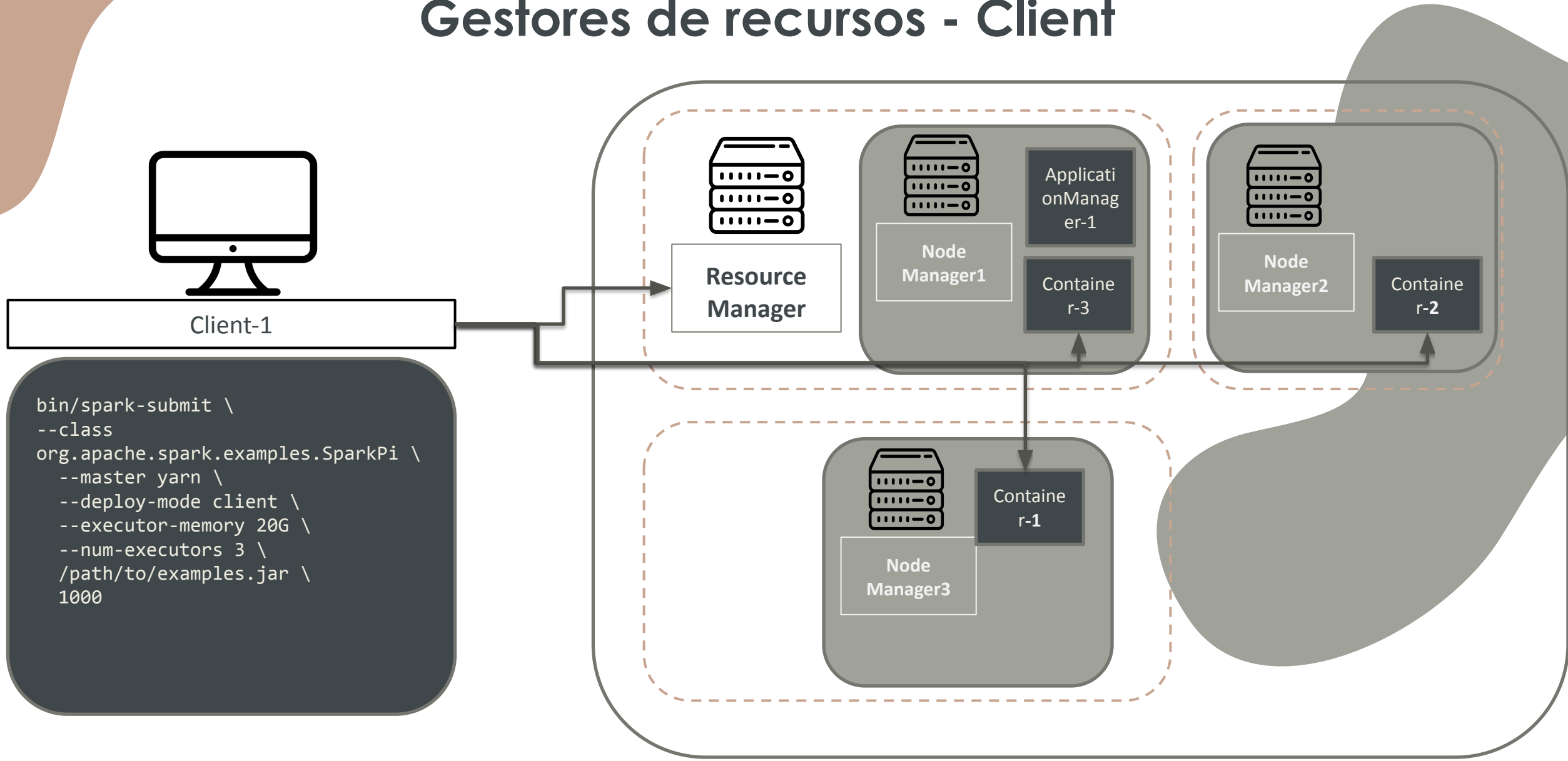
Gestores de recursos - Client



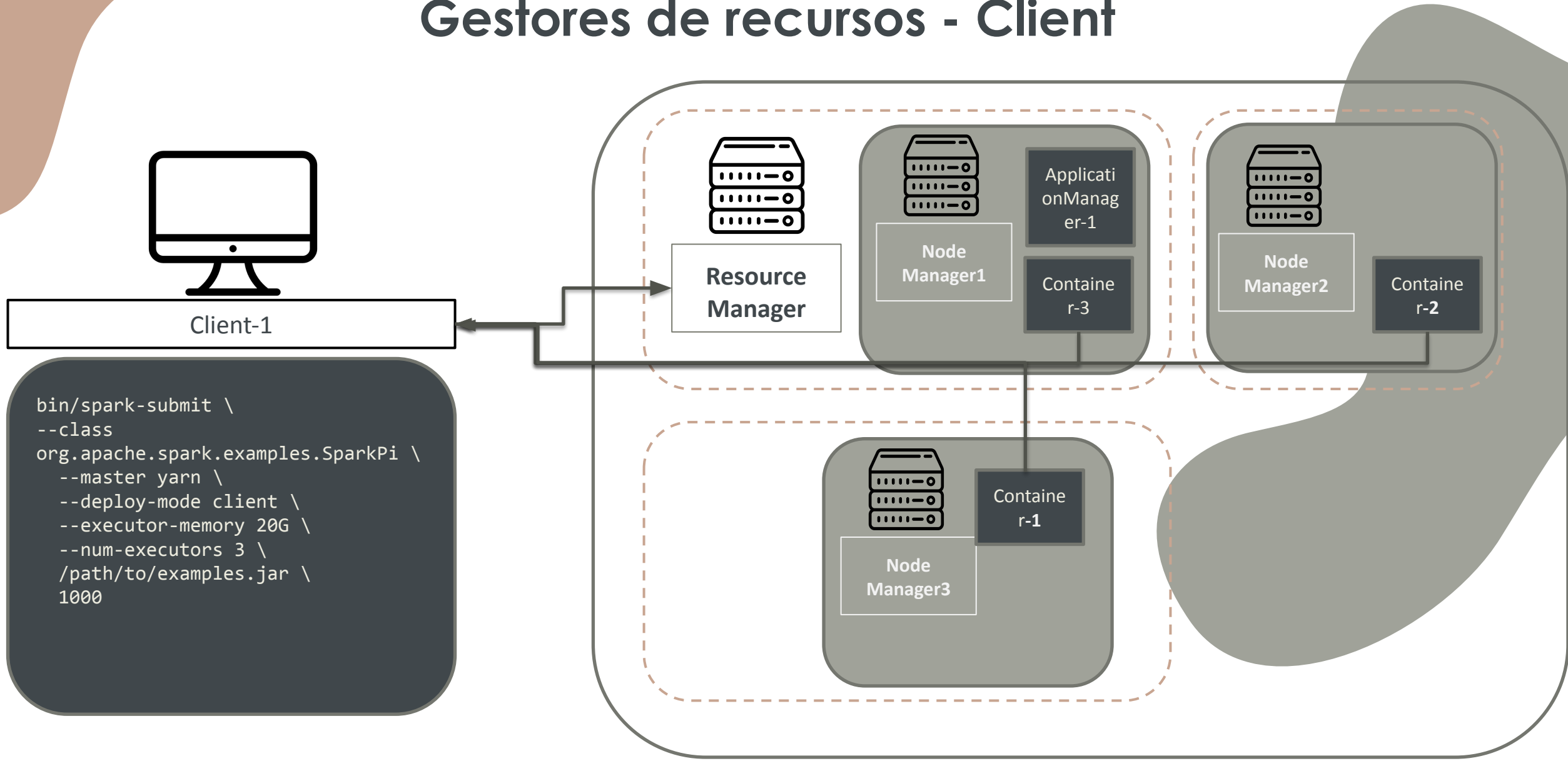
Gestores de recursos - Client



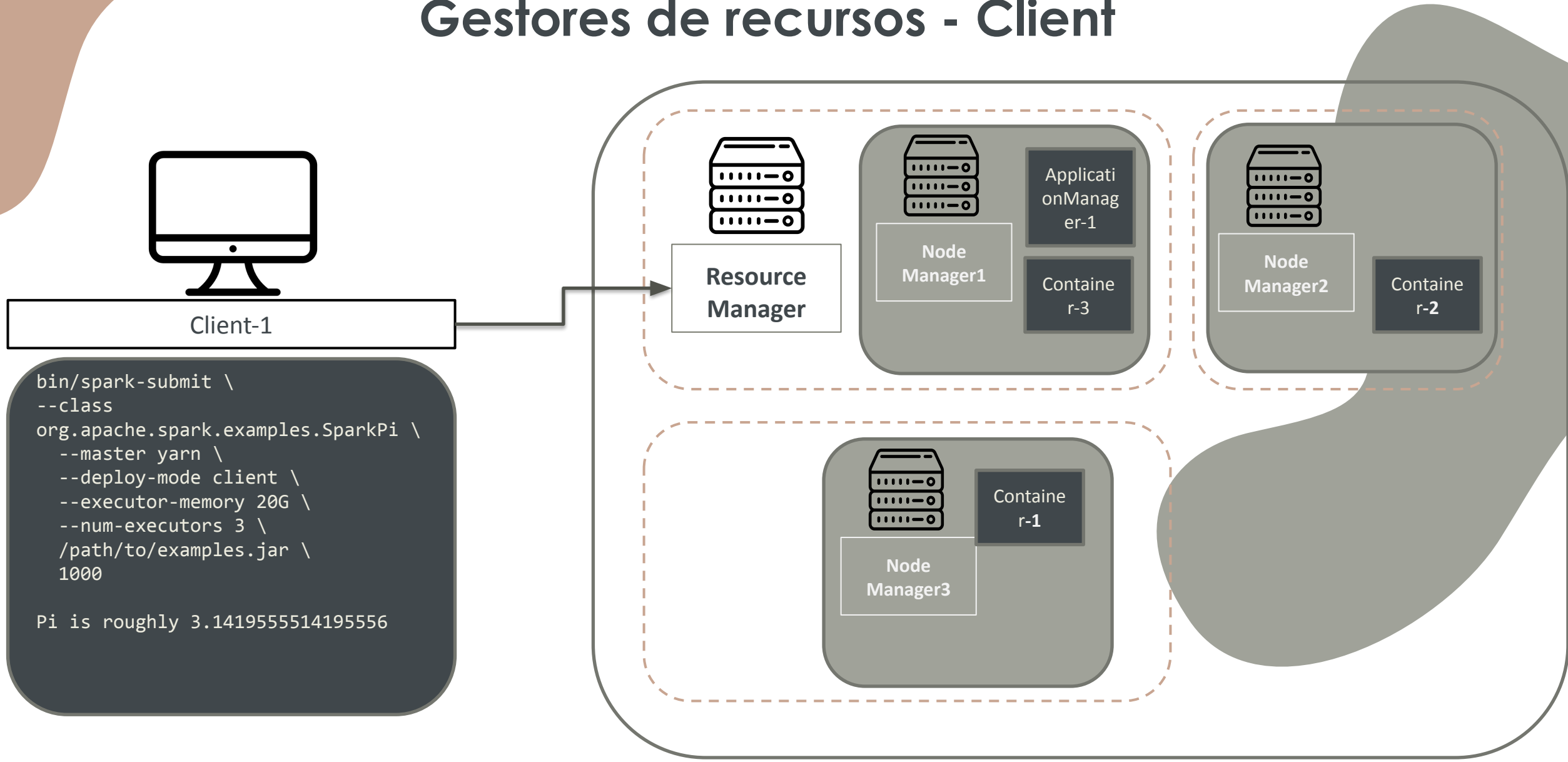
Gestores de recursos - Client



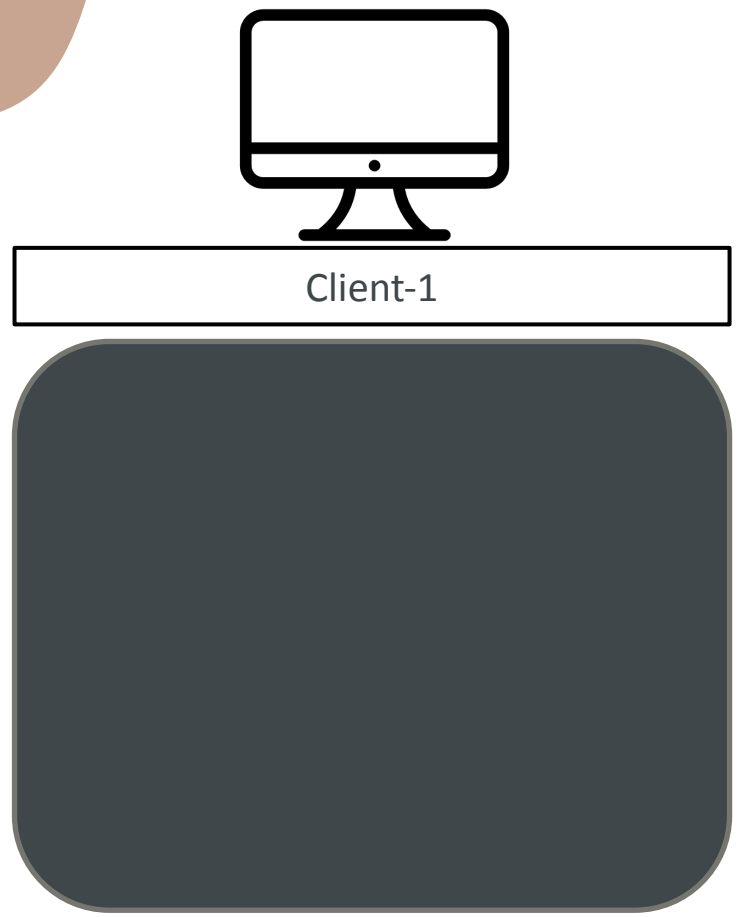
Gestores de recursos - Client

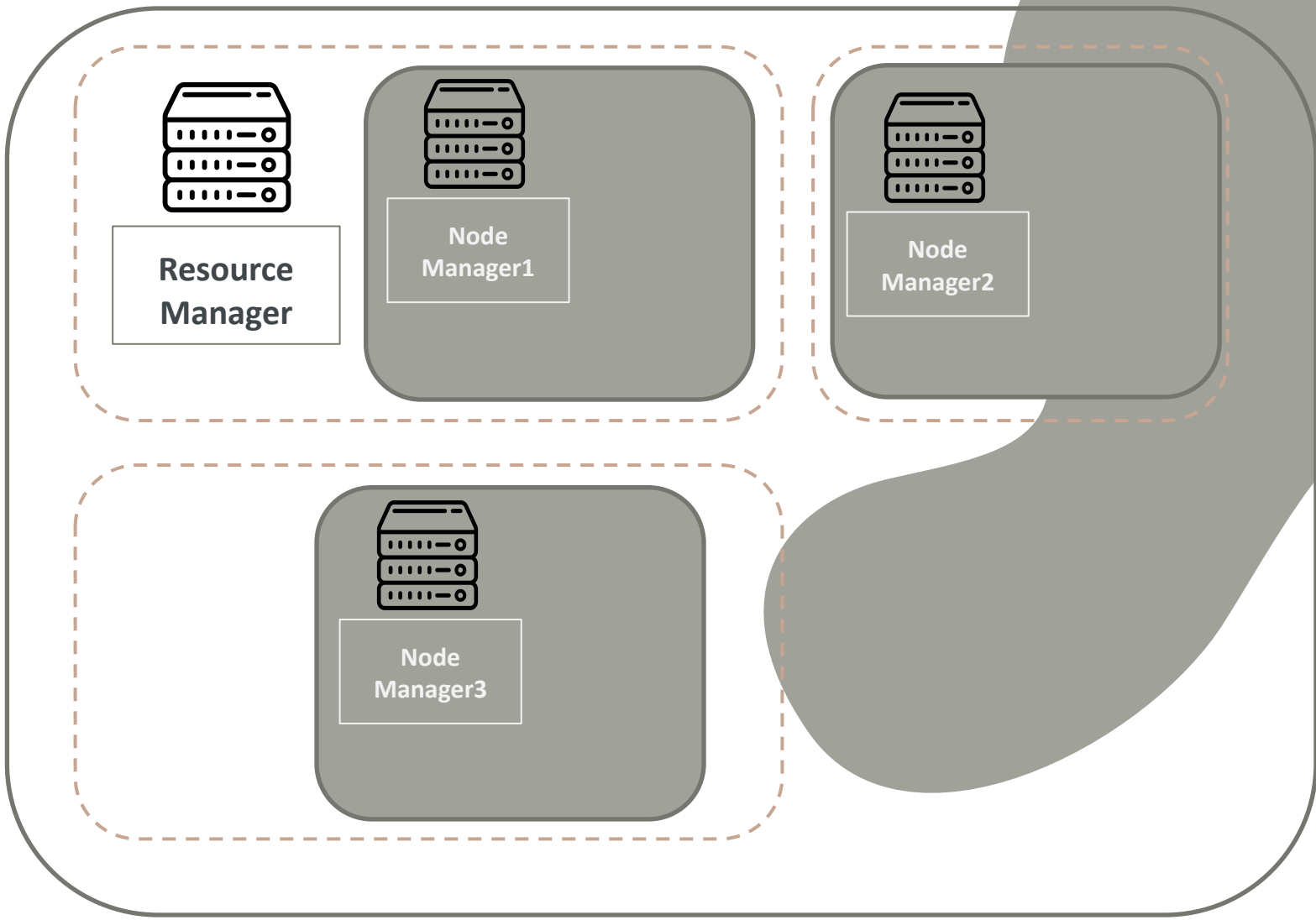


Gestores de recursos - Client

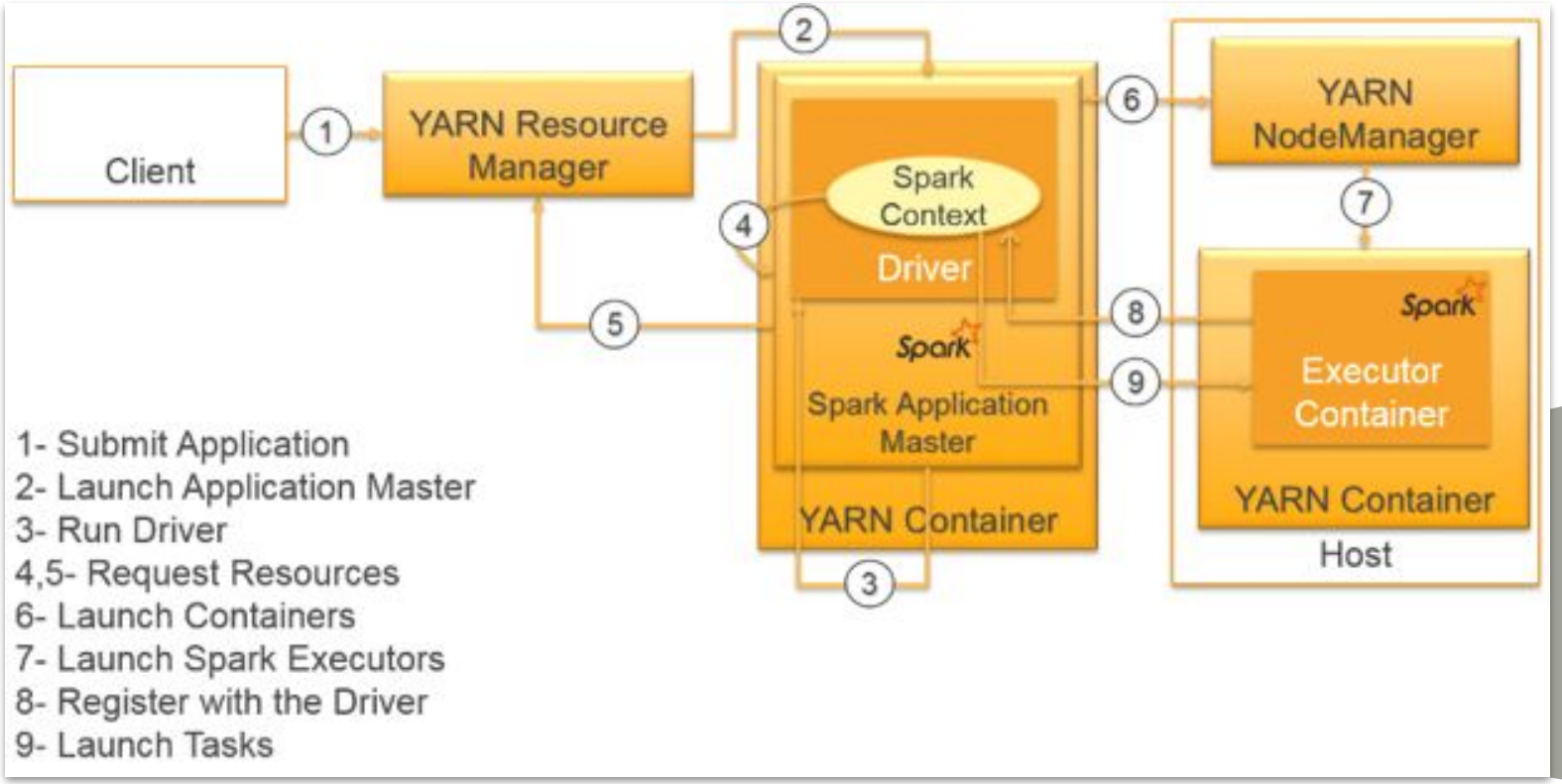


Gestores de recursos - Client

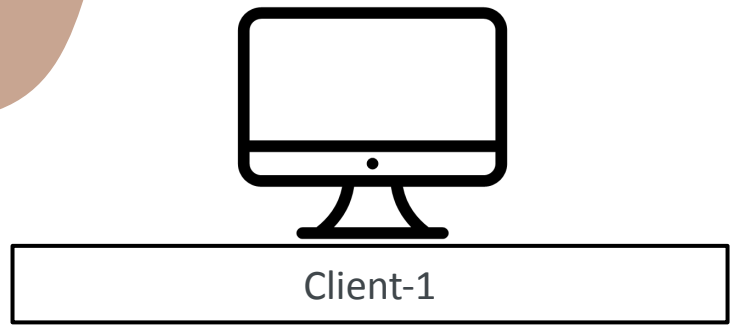




Gestores de recursos - Cluster

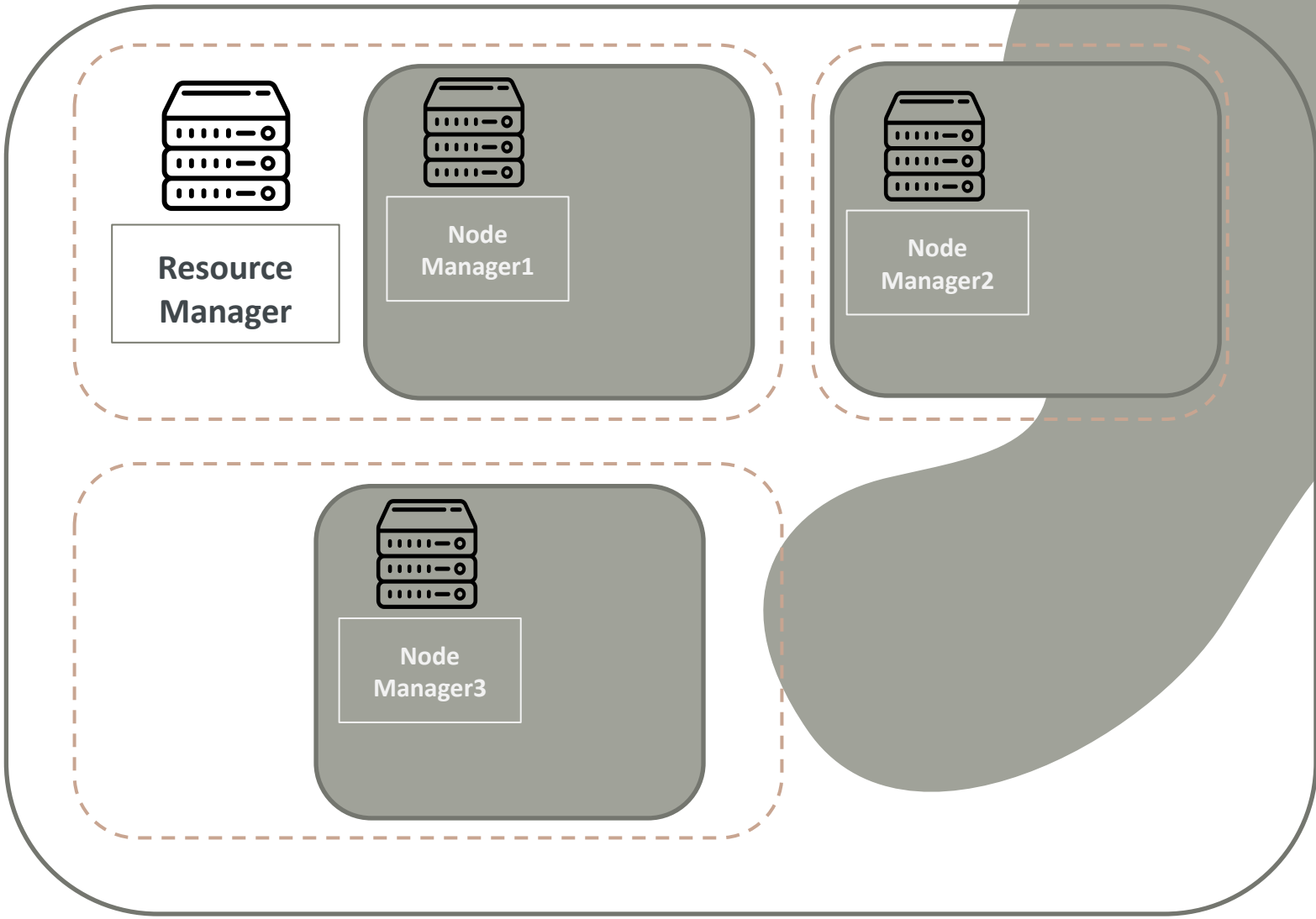


Gestores de recursos - Cluster

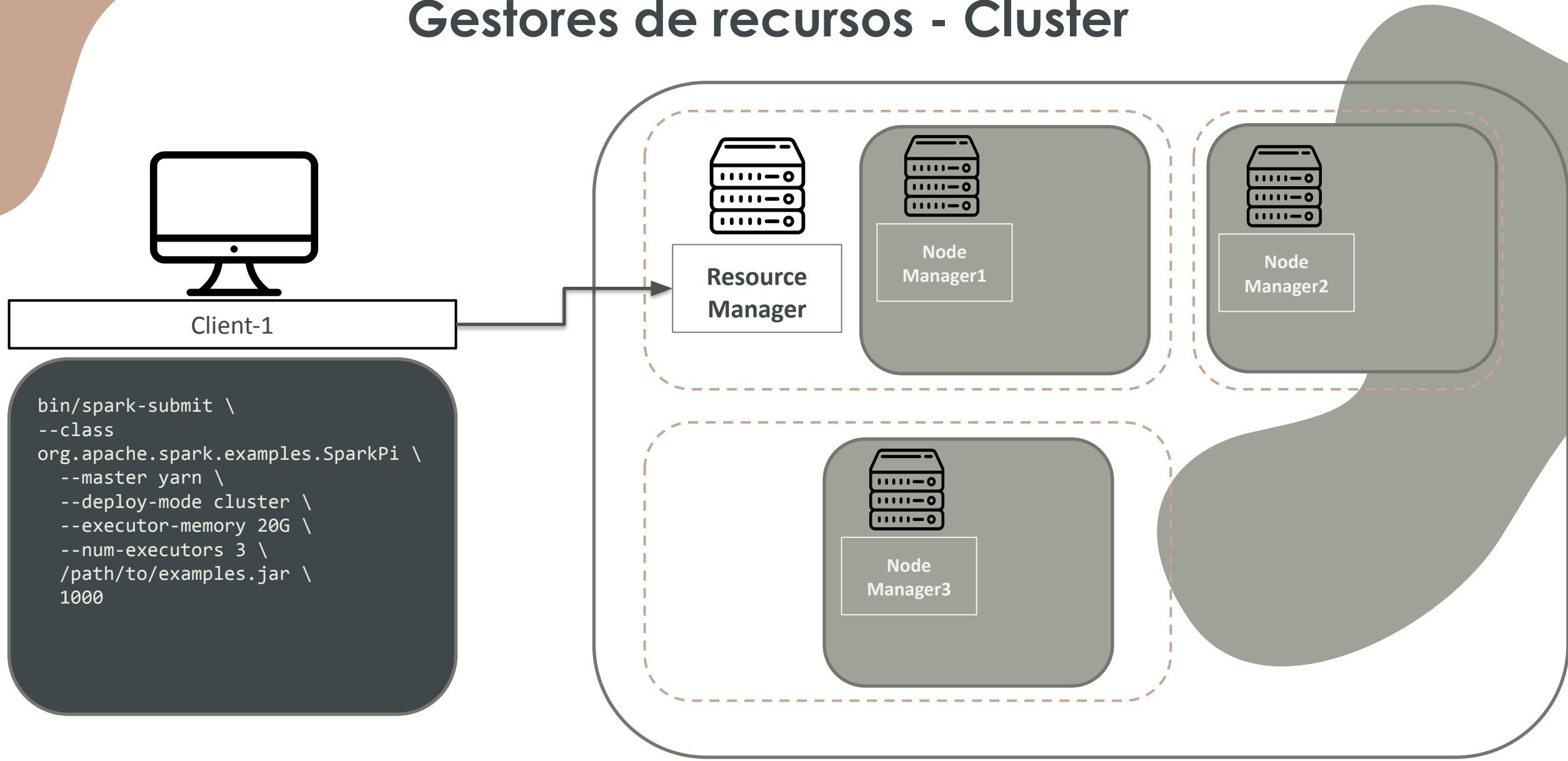


Client-1

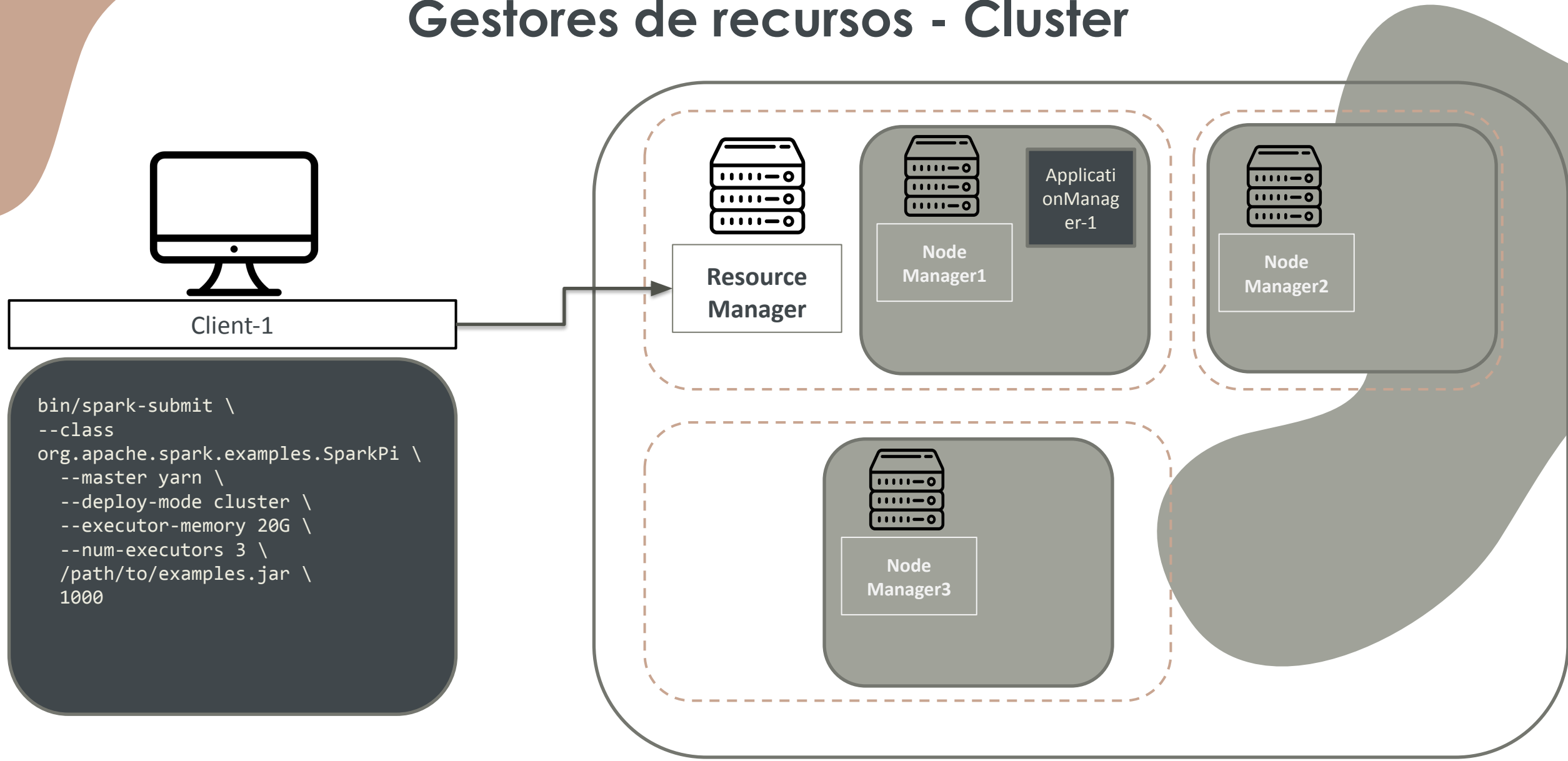
```
bin/spark-submit \
--class
org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \
  --executor-memory 20G \
  --num-executors 3 \
  /path/to/examples.jar \
  1000
```



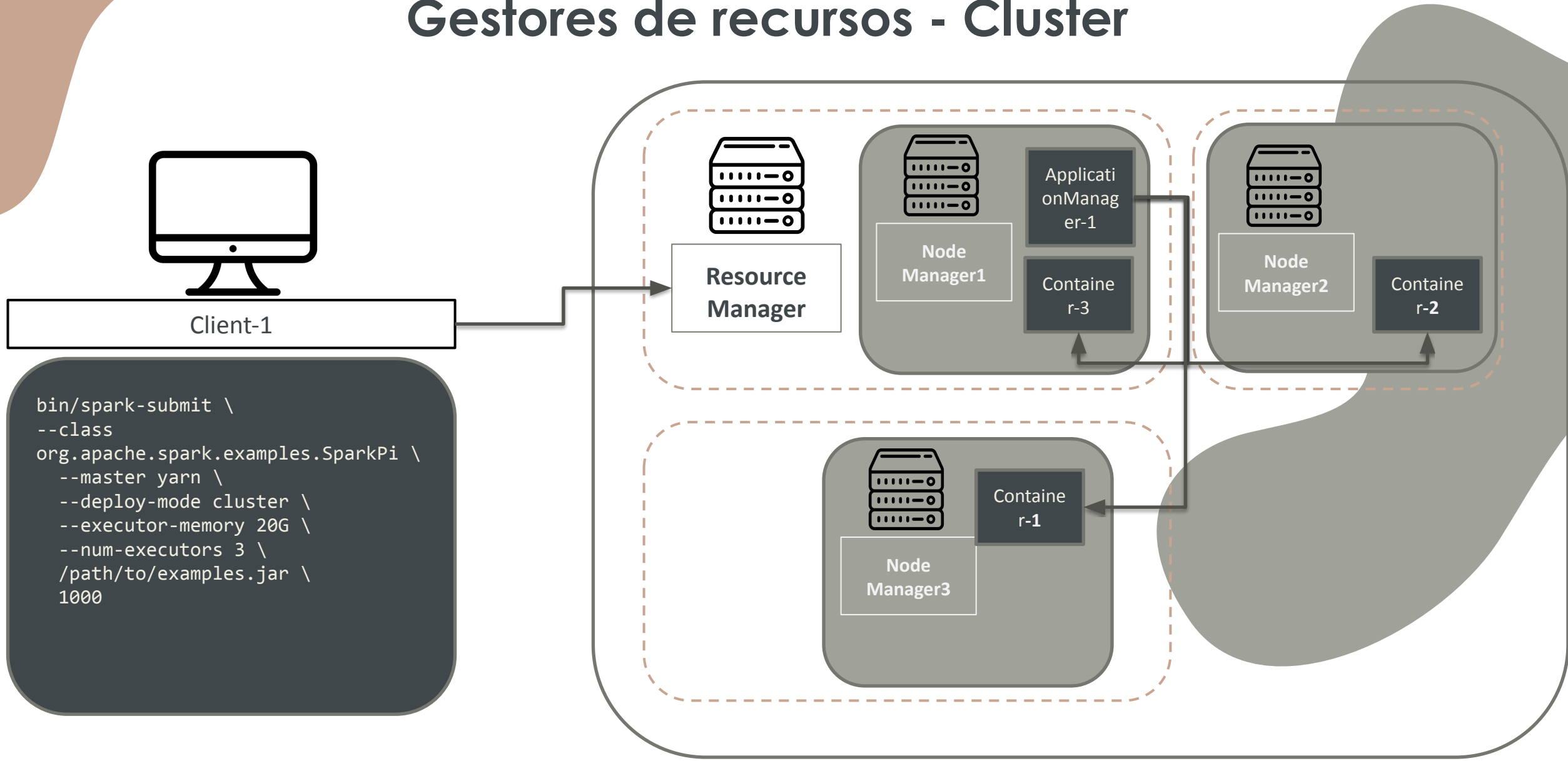
Gestores de recursos - Cluster



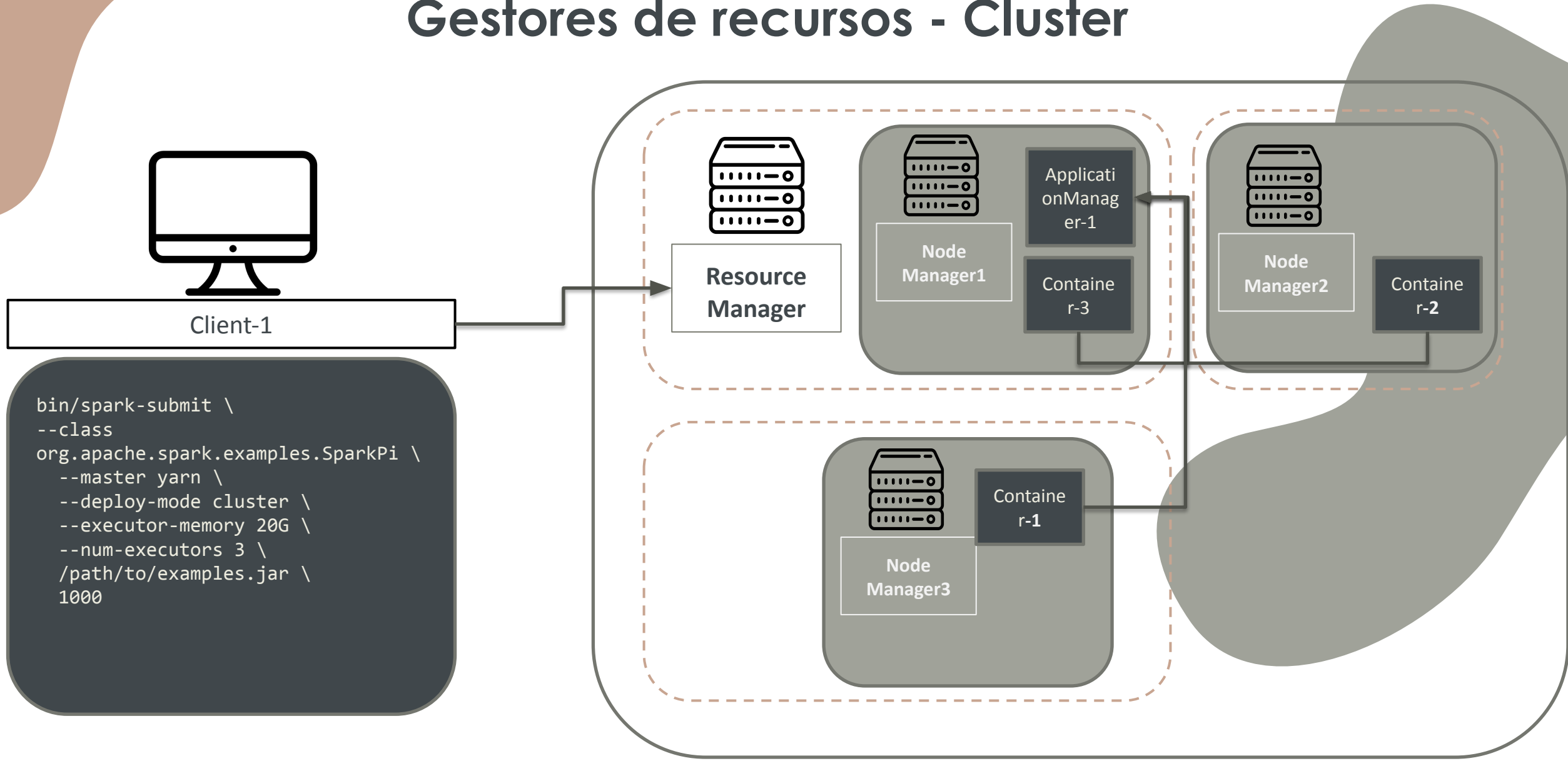
Gestores de recursos - Cluster



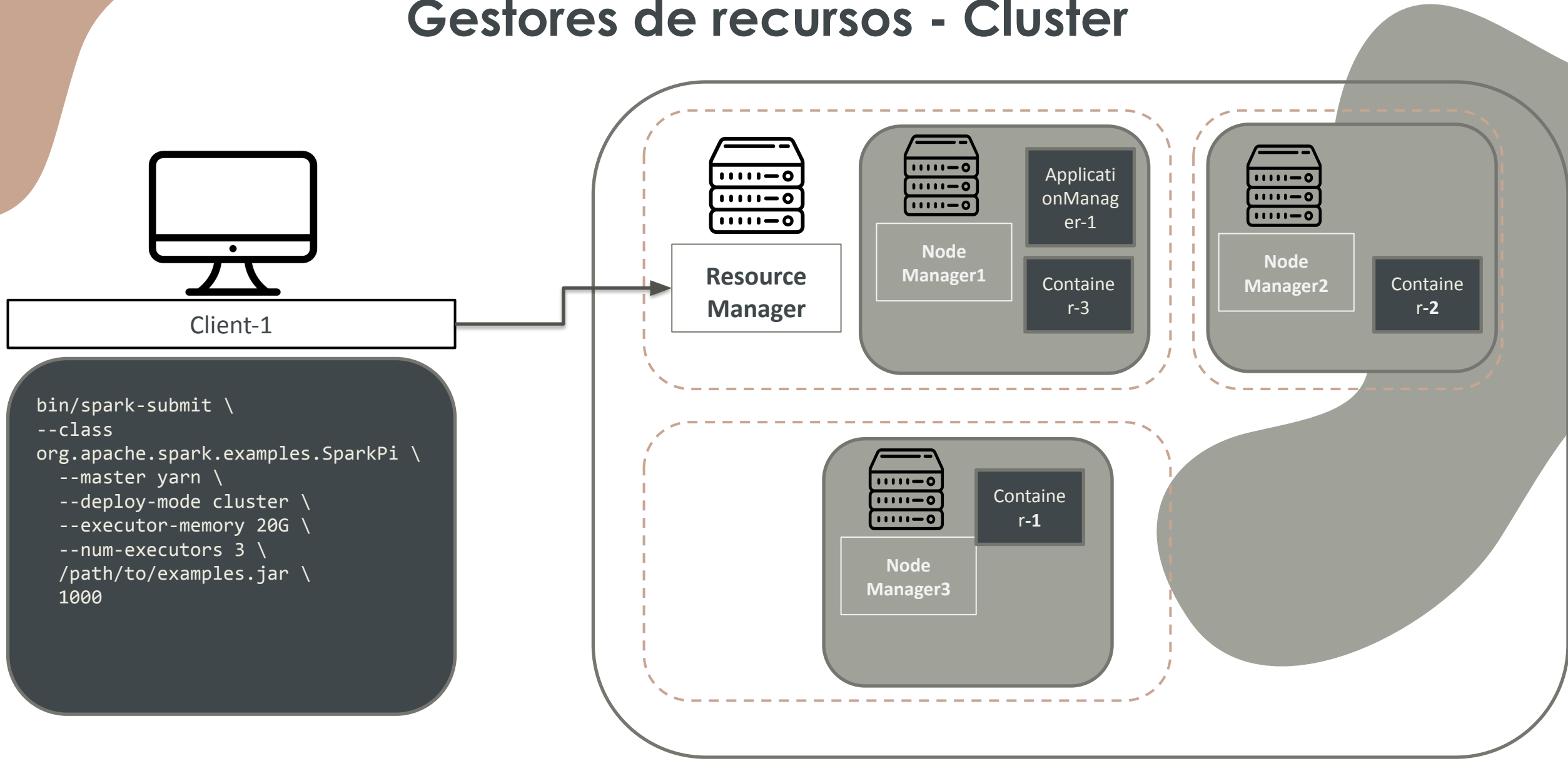
Gestores de recursos - Cluster



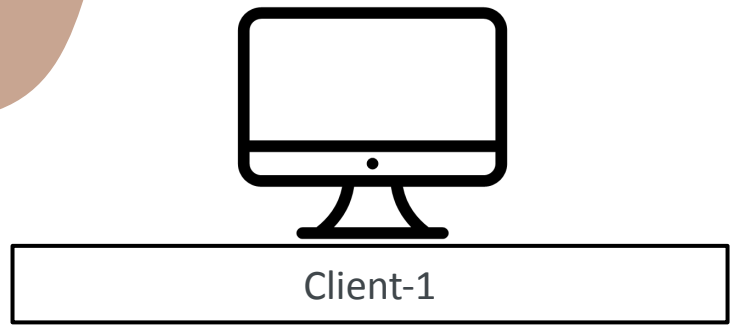
Gestores de recursos - Cluster



Gestores de recursos - Cluster

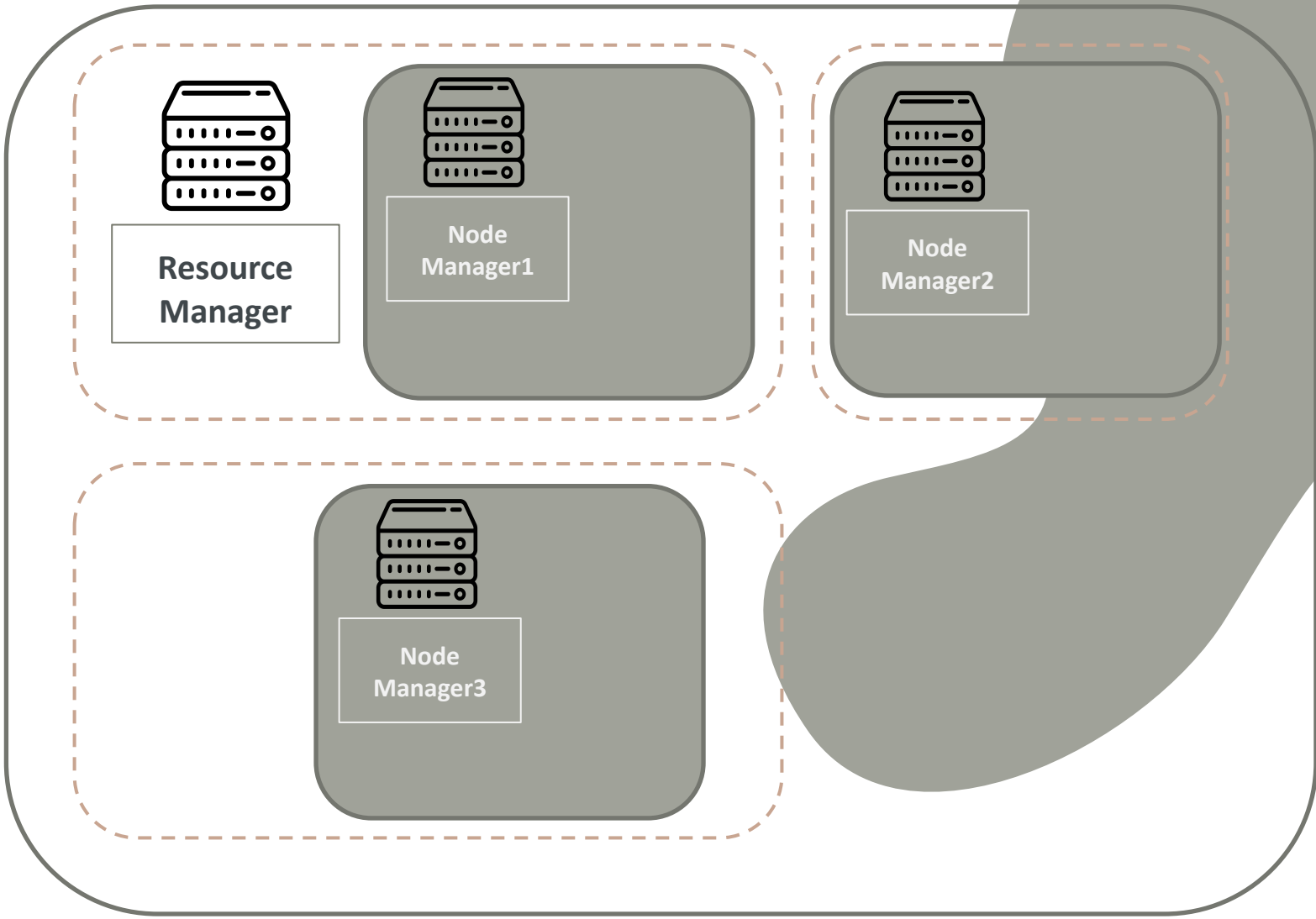


Gestores de recursos - Cluster



Client-1

```
bin/spark-submit \
--class
org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \
  --executor-memory 20G \
  --num-executors 3 \
  /path/to/examples.jar \
  1000
```



Gestores de recursos - notebooks

- En analitica es tan importante tener una solución optima como explicar claramente que qué se buscaba obtener y cómo se ha conseguido obtenerlo
- Para solventar este problema se crea el proyecto **Jupyter**
- Busca dar al científico y al ingeniero de datos una herramienta online donde poder declarar primero en lenguaje humano que se quiere obtener para luego poder visualizar el código que obtendrá lo descrito anteriormente
- Los notebooks dan soporte a distintos lenguajes de programación,entre ellos los dos principales de Spark **Python** y **Scala**

Gestores de recursos - notebooks

Load Python libraries

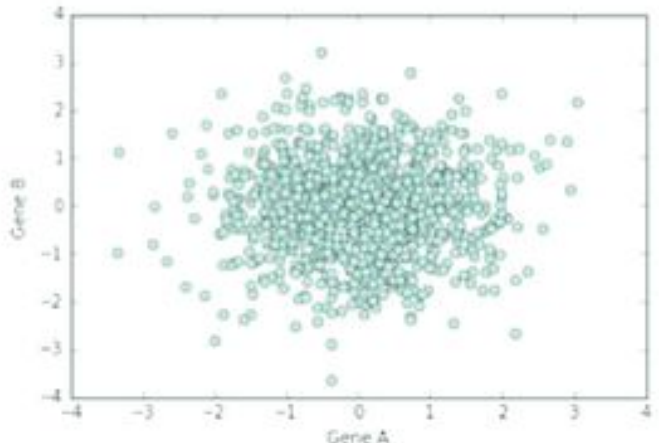
In [1]:
```
get_ipython().magic('matplotlib inline')

import matplotlib.pyplot as plt
from numpy.random import normal
```

Generate random numbers simulating gene-expression values

In [2]:
```
geneA = normal(size=1000)
geneB = normal(size=1000)
```

Plot the values

In [4]:
```
plt.plot(geneA, geneB, "o", color="#99C1C2")
plt.xlabel("Gene A")
plt.ylabel("Gene B")
plt.show()
```



RDD

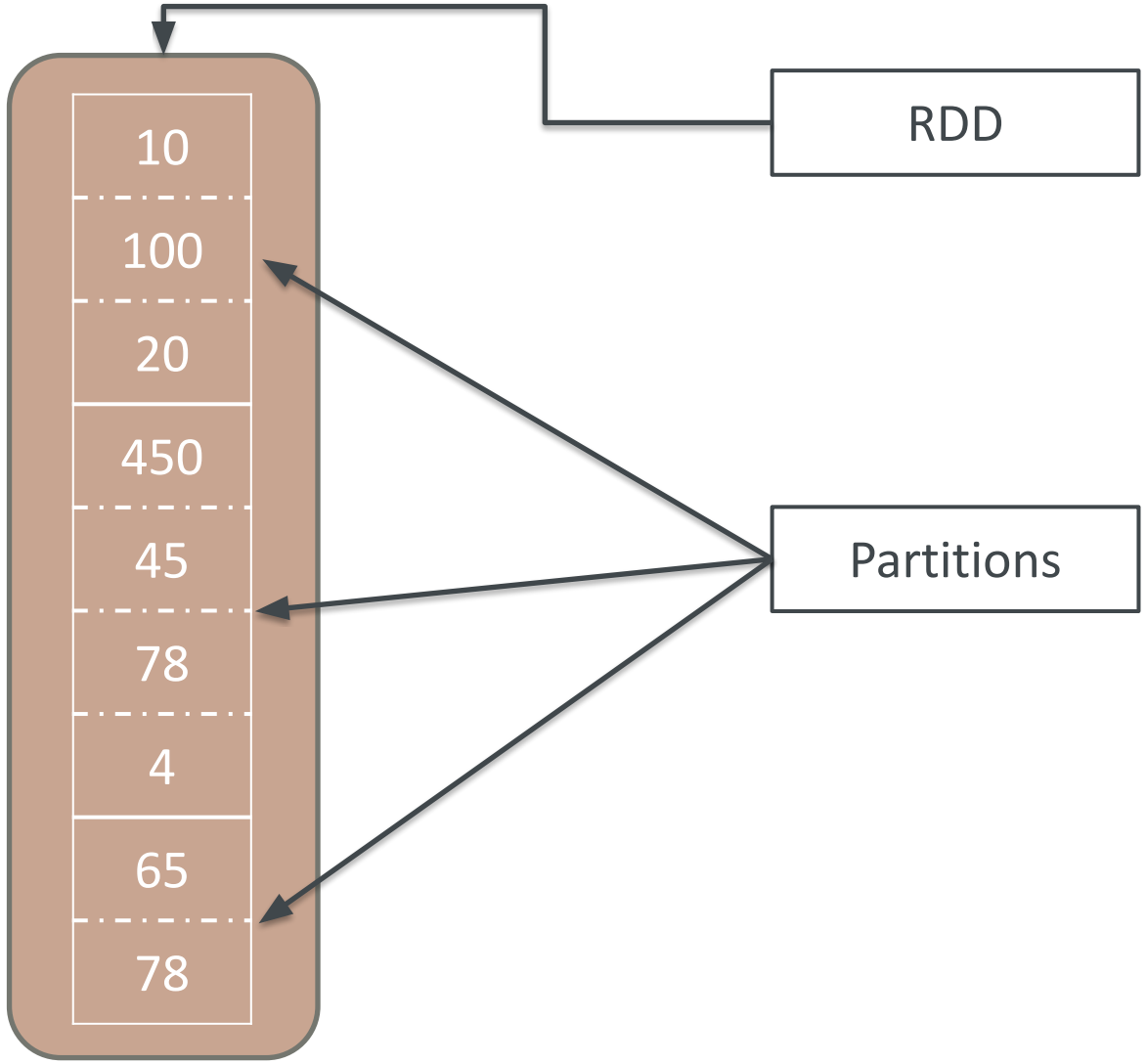
RDD - Conceptos

- Aunque Spark nace en el mundo de Hadoop la tecnología había avanzado
- El creador de Spark se basa en un lenguaje de programación de la JVM llamado **Scala**
- Scala es un lenguaje funcional, en el que se prima la inmutabilidad y que optimiza la gestión de listas de Java
- Con estas premisas se crea el concepto base de Spark que son las RDD
- Spark busca solventar todas las casuísticas que engloba spark(grafos, SQL, streaming y machine learning) usando como base los RDD

RDD - Conceptos

- Es la principal abstracción de datos en Spark
- RDD son las siglas de **Resilent Distributed Dataset**
    - **Resilent:** resilente es decir es tolerante a fallos. A diferencia de Hadoop que solventa este problema con escrituras a disco Spark tiene un mecanismo que recalcula sólo la fracción de datos que ha fallado aprovechando la velocidad que le da el procesamiento en memoria
    - **Distributed**: Los datos se encuentra distribuidos en todo el sistema distribuido en las distintas **partitions**
    - **Dataset**: Un RDD es un conjunto de datos simples o complejos con la peculiaridad que todos los objetos tiene que ser serializables

RDD - Conceptos



RDD - Características

- **In-Memory**, Los datos dentro de un RDD están guardado tanto tiempo y en tanto espacio como sea posible
- **Immutable**: Los datos dentro de un RDD no pueden ser cambiados, para modificarlos hay que crear otro RDD.
- **Lazy evaluated**: Los datos no son procesados por el sistema distribuido a no ser que el proceso llame a una acción
- **Cacheable**: Los RDD se pueden almacenar en memoria guardando todos sus datos en los nodos donde se almacenan los **partitions**
- **Parallel**: Los datos se pueden procesar en paralelo siendo la unidad mínima de procesamiento el **partition**

RDD - Características

- **Typed:** Todos los elementos de un RDD tienen que tener el mismo tipo de datos, puede ser complejo o simple pero todos tienen que ser Serializables
- **Partitioned:** La distribución de los datos dentro de un **RDD** sigue una lógica de particionado simular a los partitioned de Map&Reduce.
- **Location-Stickiness:** Los RDD pueden ubicar sus datos en la localización óptima para leer sus datos de entrada

RDD - Instanciación

- Los RDD son colecciones de colecciones distribuidas e inmutables
- Necesitan una fuente de datos, que tiene que permitir el particionado, que será el origen del RDD
- Hay tres formas de instanciar un RDD:
  - **De memoria:**
    - Usa una lista creada en el driver que se particiona por número de elementos
    - **Ej:** sparkContext.paralellize([1,2,3,4])
  - **De un repositorio de información externo:**
    - Toma como fuente un repositorio que permite particionar un conjunto de datos (HDFS, Cassandra, HBase, MongoDB, ...)
    - **Ej:** sparkContext.textFile("hdfs://nameNode:8020/user/books")

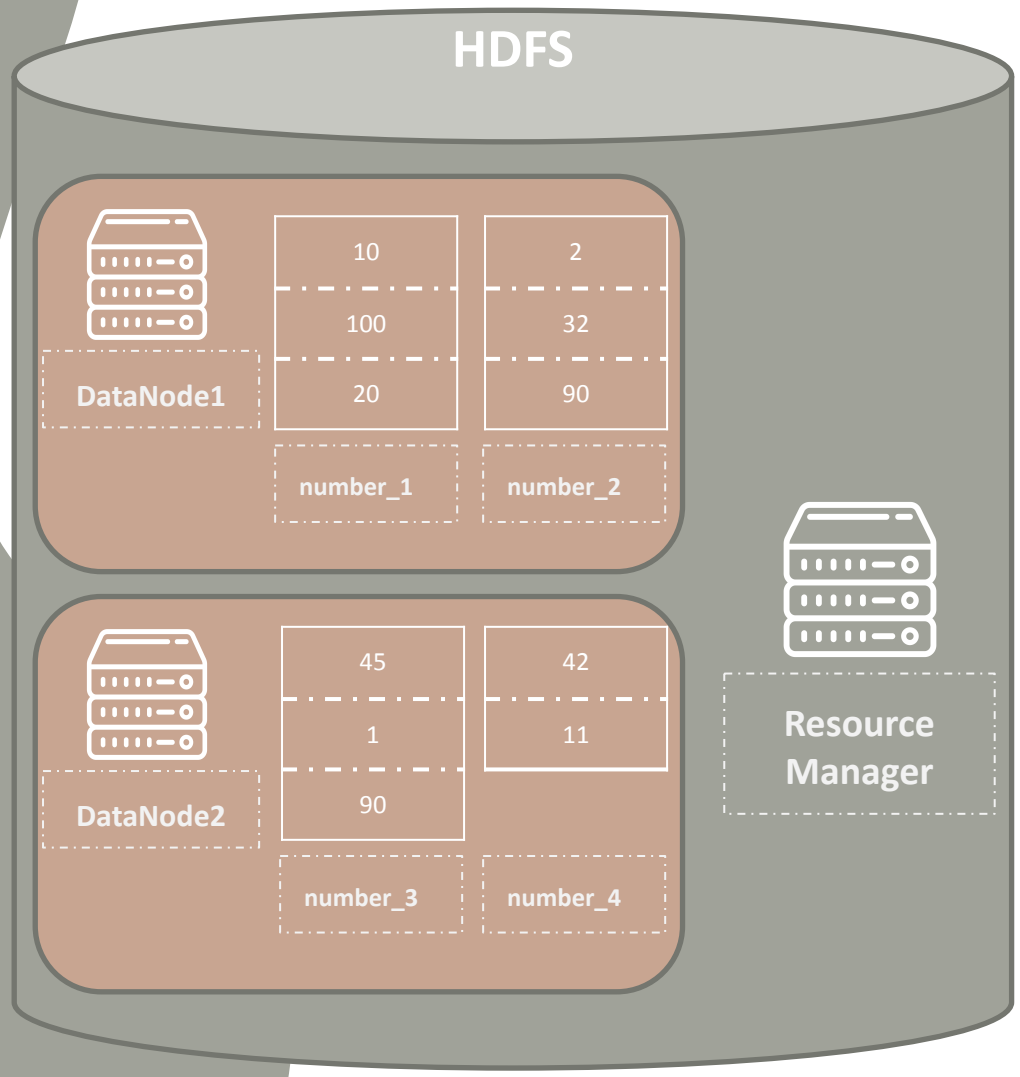
RDD - Instanciación

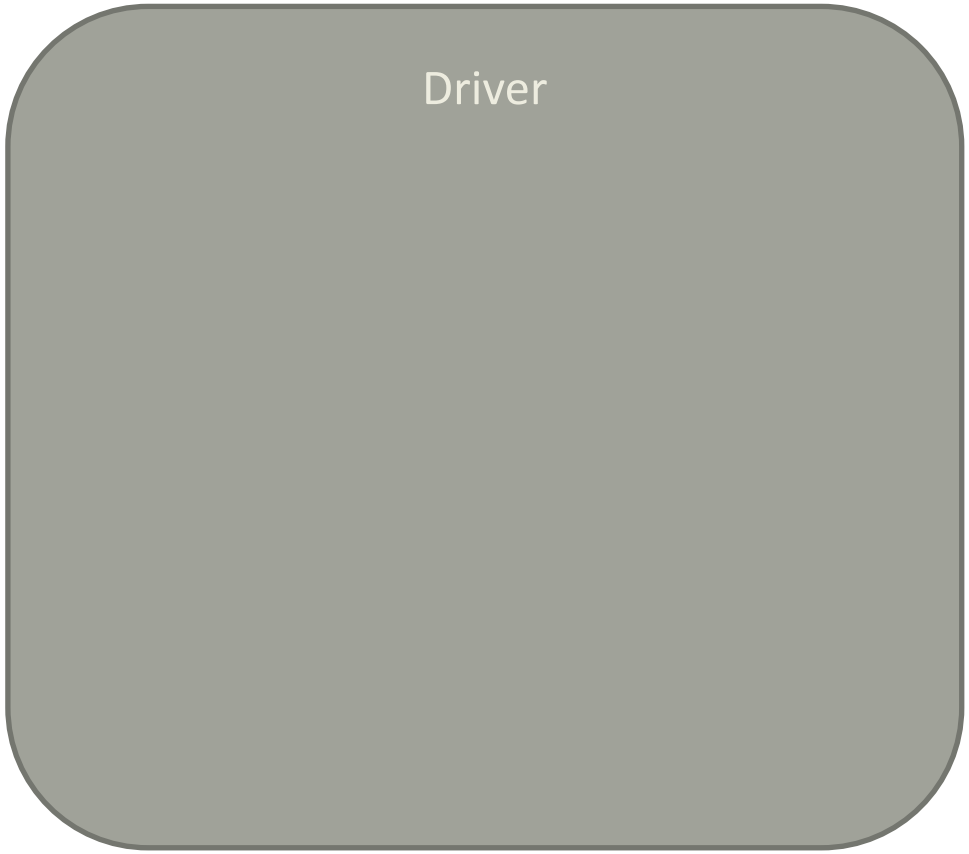
- Hay tres formas de instanciar un RDD:
  - **De otro RDD:**
    - Usa como base otro RDD se usa en operaciones que se ejecutan en cada uno de los elementos de una partición
    - **Ej:** other_rdd.map(lambda letter: letter.upper())

RDD - Lineage

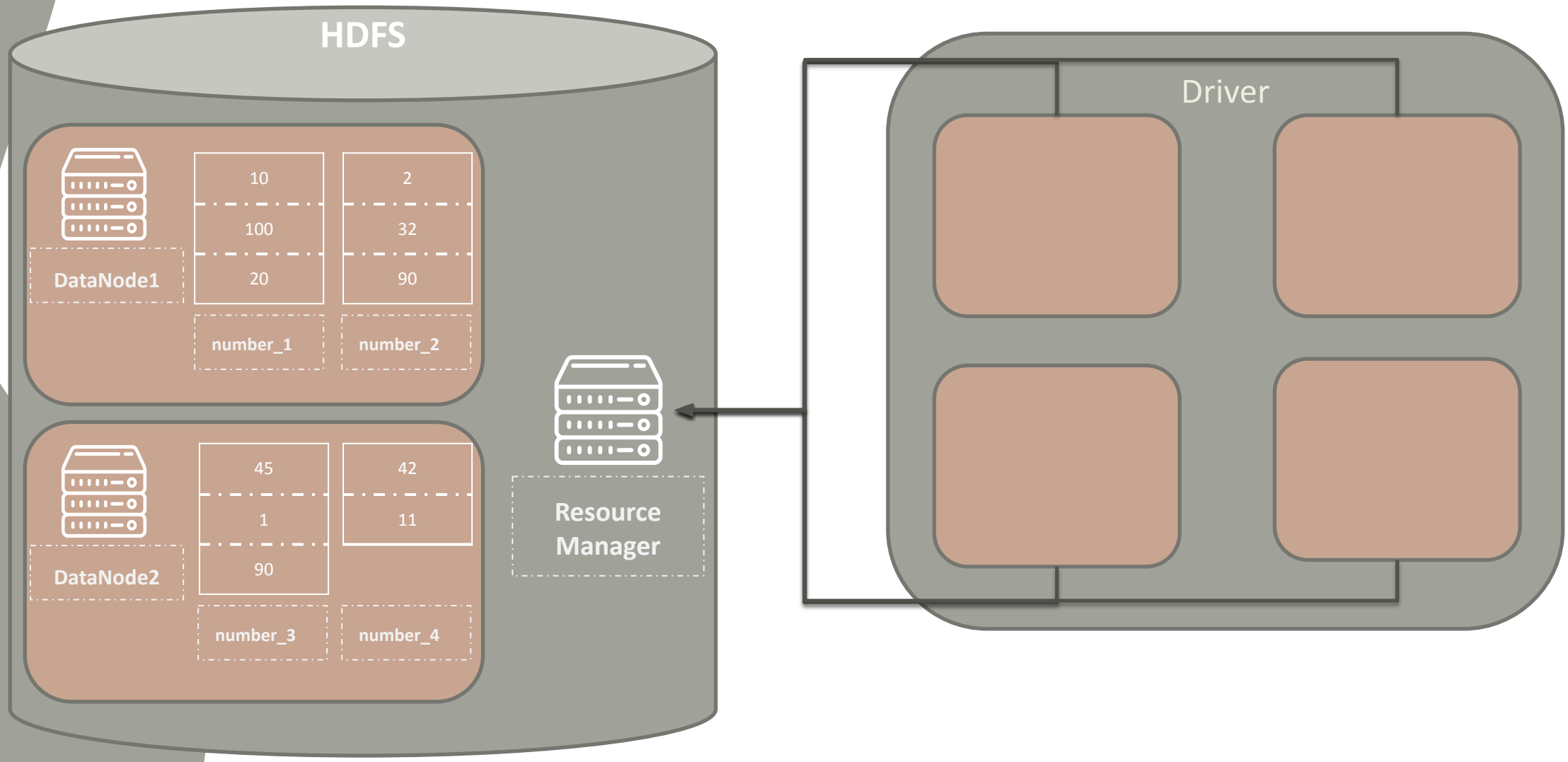
- El mecanismo de respuesta de Spark ante el fallo es el recalculo de el **partition** que ha dado el error
- Spark también intenta almacenar el máximo tiempo posible dentro del nodo
- Dada la inmutabilidad de los **RDD** el origen más común de los RDDs es la transformación de otro RDD
- Para la regeneración del RDD se busca cual el linage de un RDD para ver que fases hay que recalcular para volver a obtener el mismo dato
- El primer RDD que no tiene nigún RDD previo en su linage es conocido como **baseRDD**

RDD - Funcionamiento

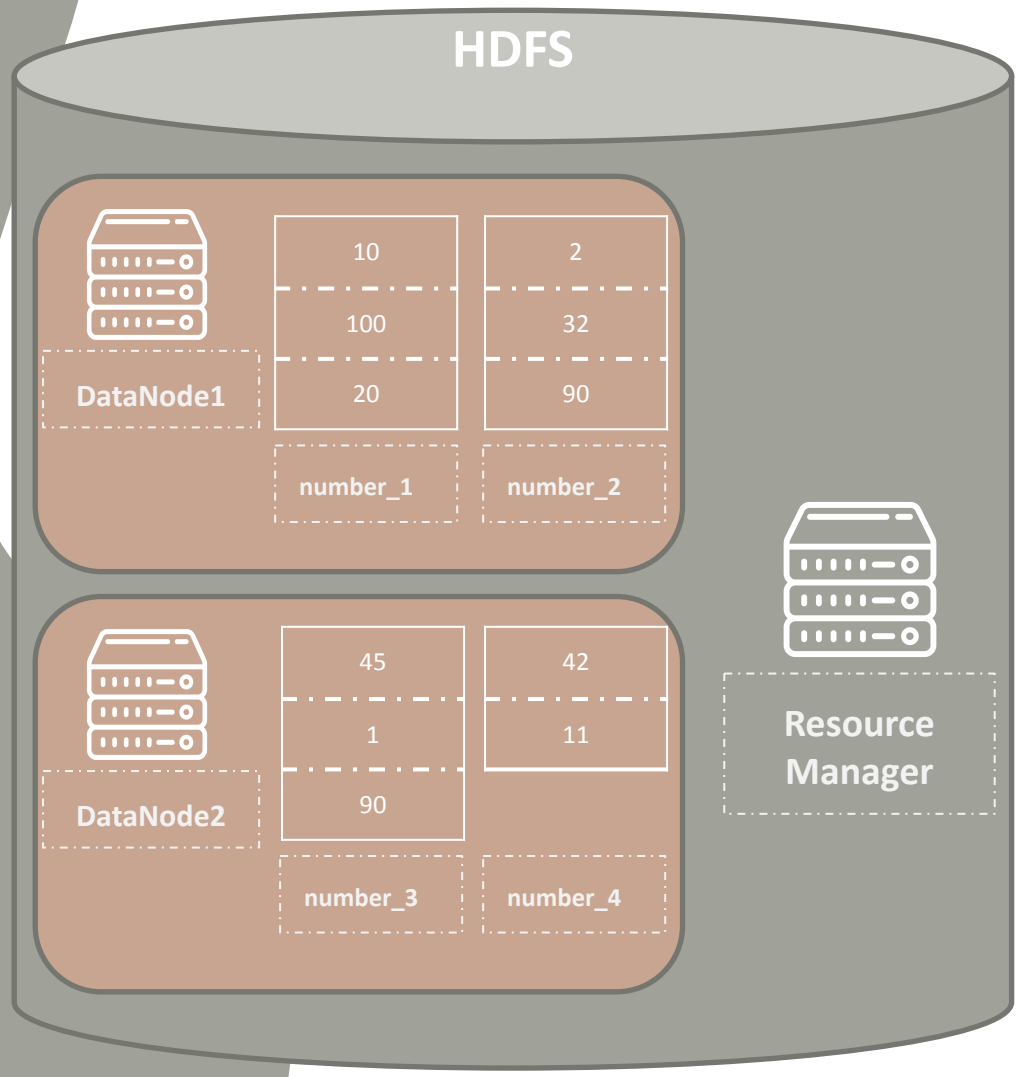


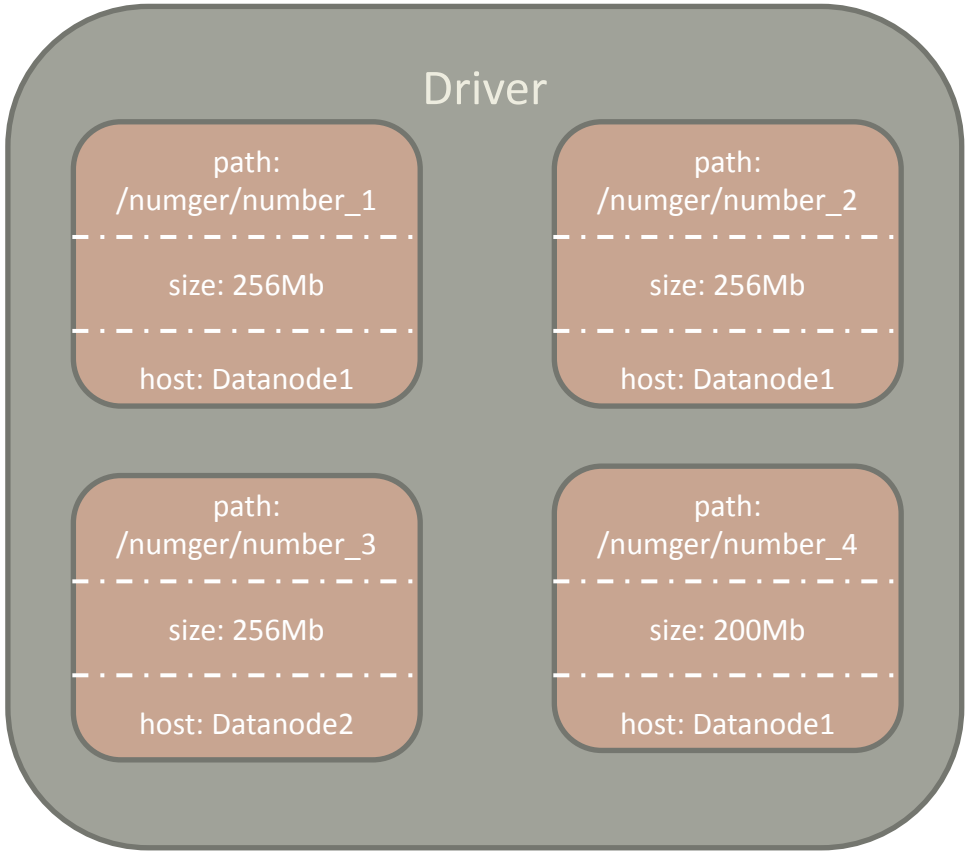


RDD - Funcionamiento



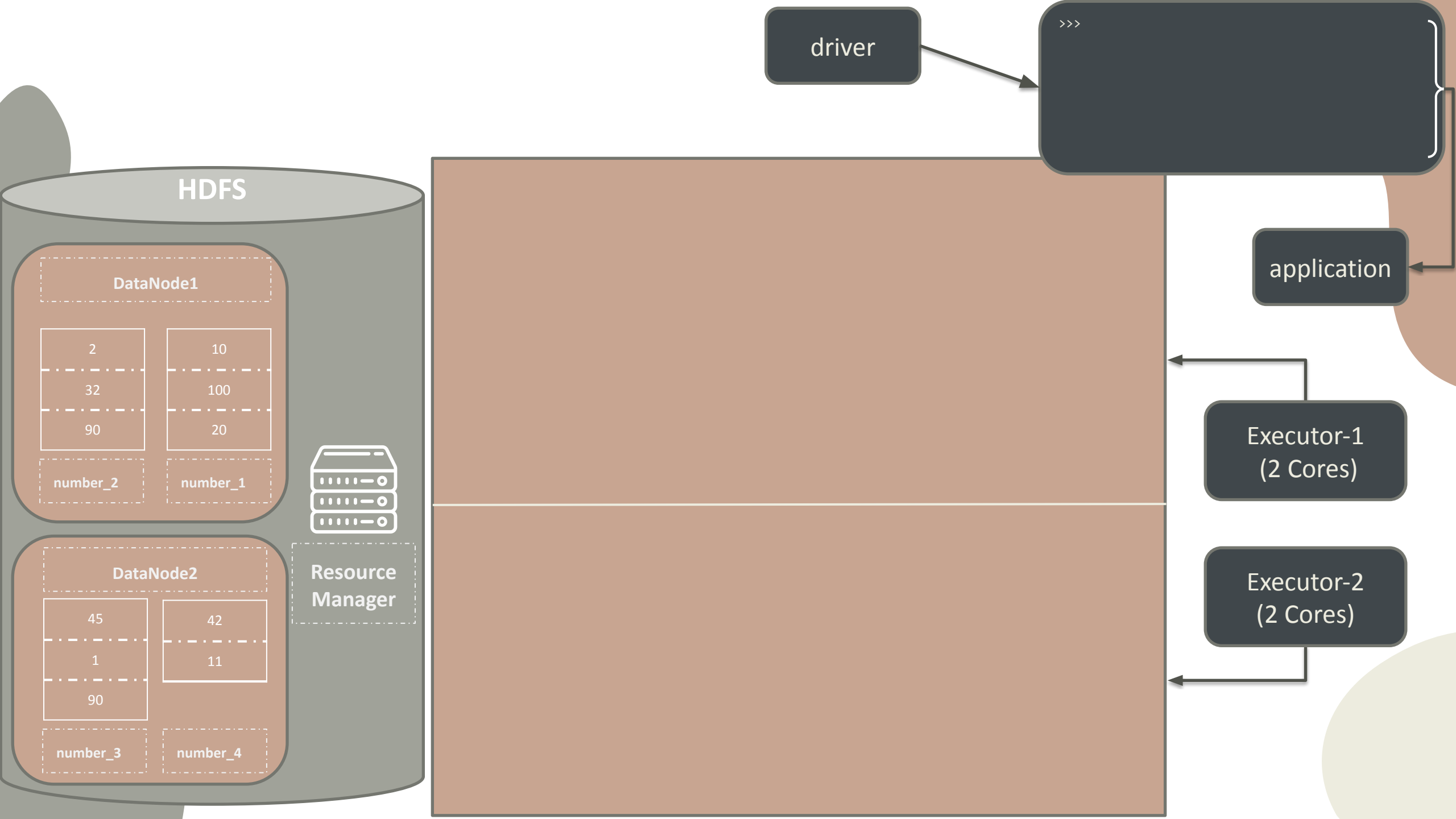
RDD - Funcionamiento

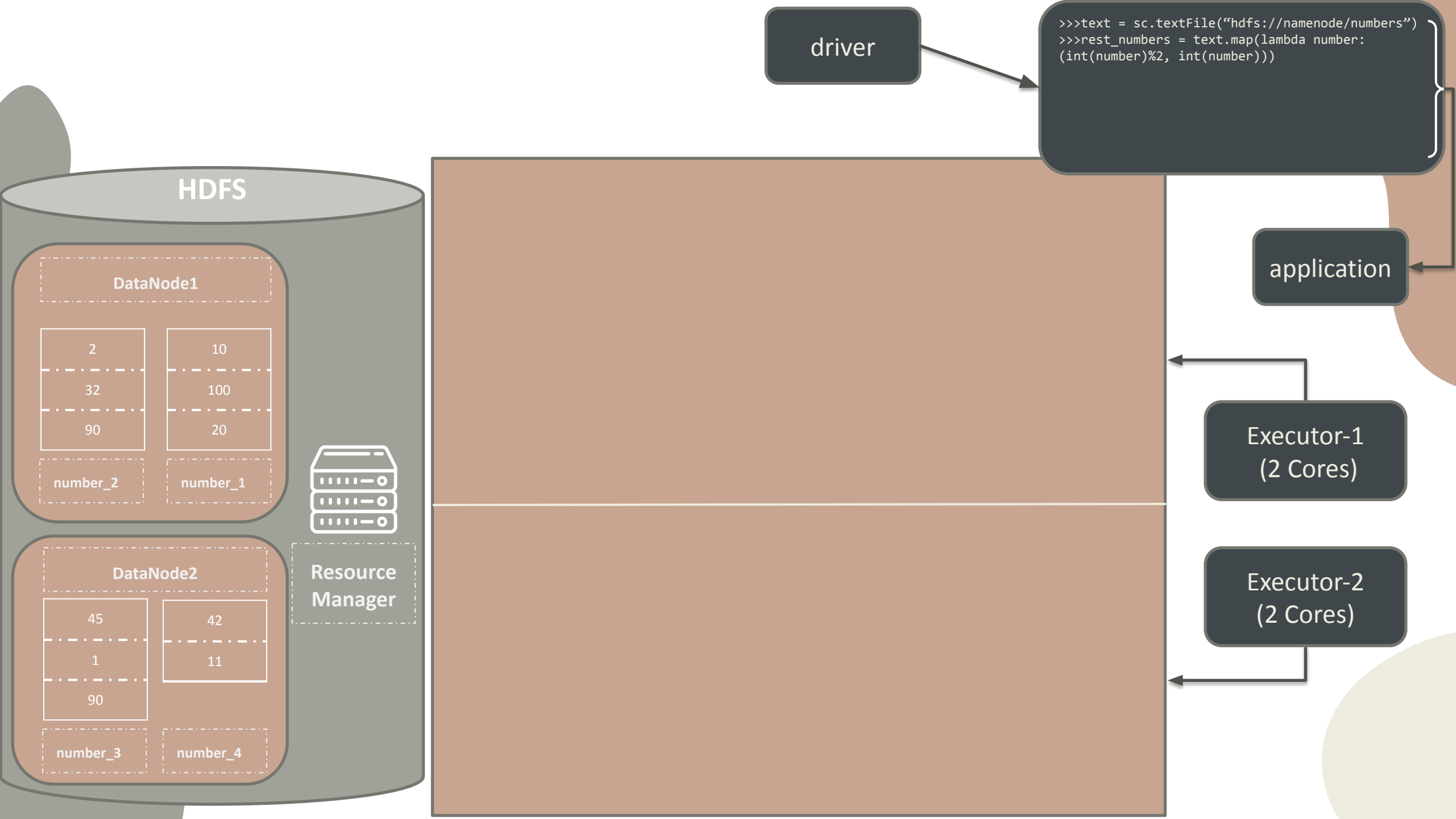


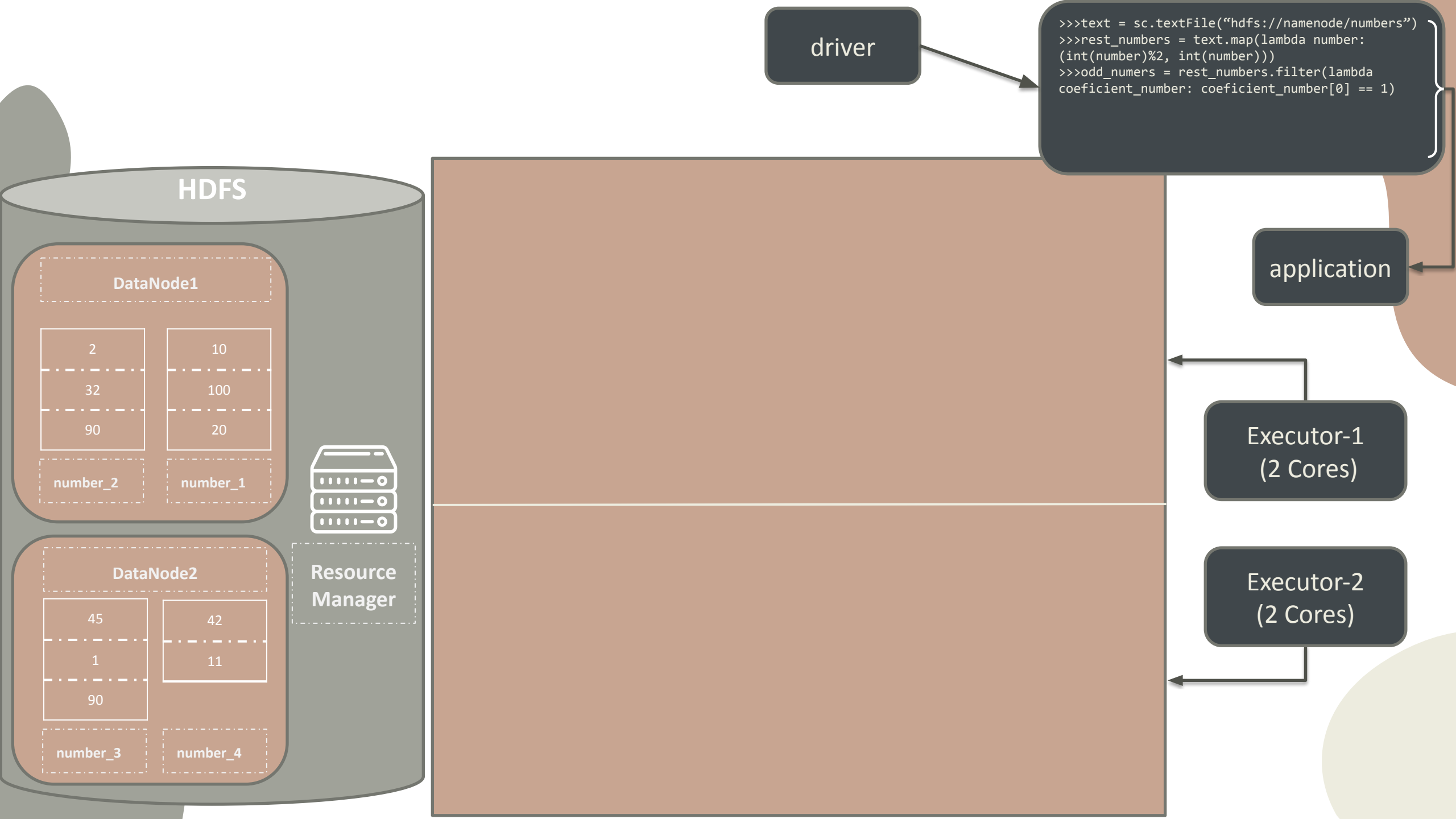


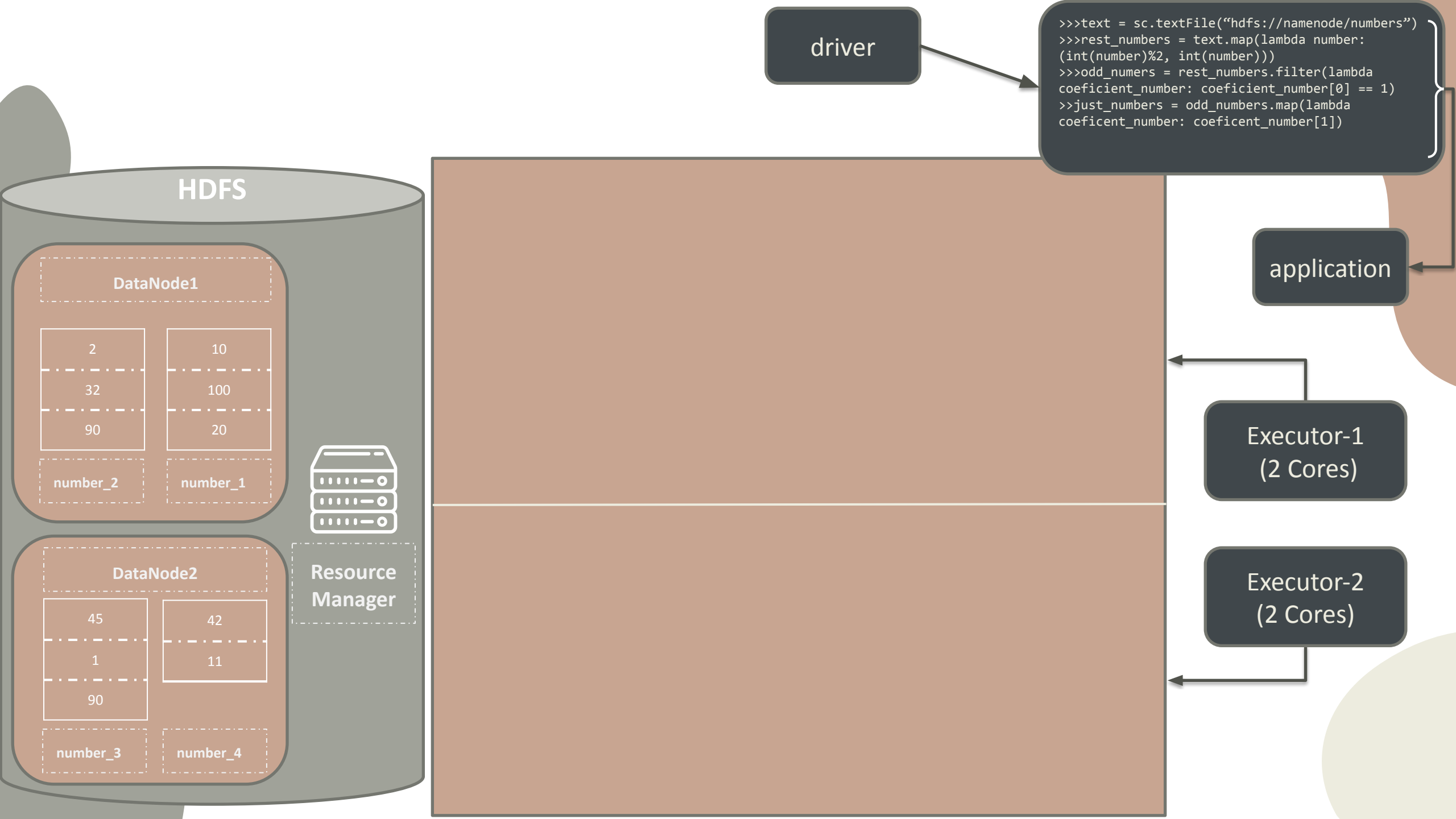
RDD - Funcionamiento

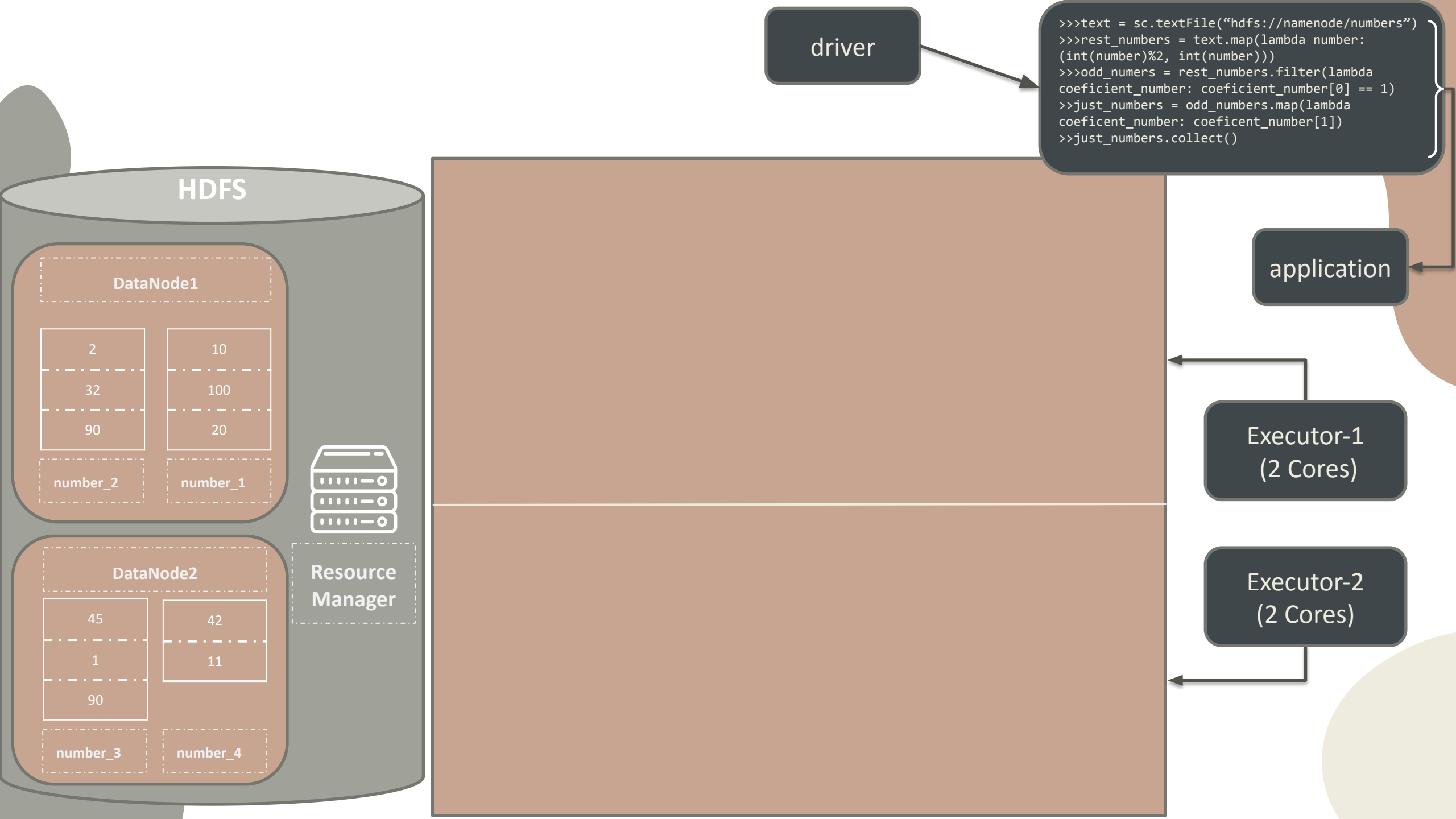
- El mecanismo de respuesta de Spark ante el fallo es el recalculo de el **partition** que ha dado el error
- Spark también intenta almacenar el máximo tiempo posible dentro del nodo
- Dada la inmutabilidad de los **RDD** el origen más común de los RDDs es la transformación de otro RDD
- Para la regeneración del RDD se busca cual el linage de un RDD para ver que fases hay que recalcular para volver a obtener el mismo dato
- El primer RDD que no tiene nigún RDD previo en su linage es conocido como **baseRDD**

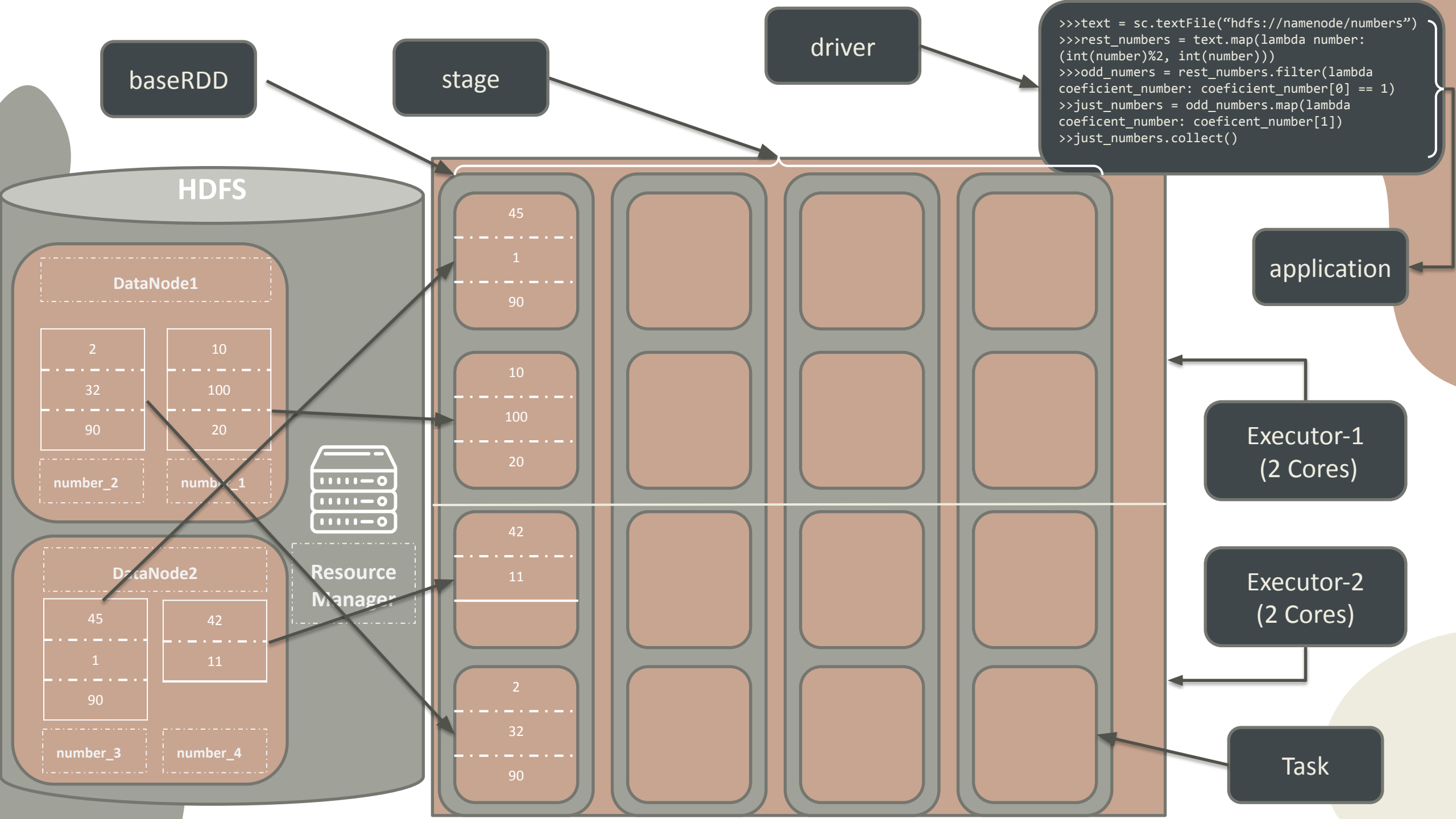

driver
>>>
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
Resource Manager
DataNode2
45
1
90
42
11
number_3
number_4
Executor-1 (2 Cores)
Executor-2 (2 Cores)

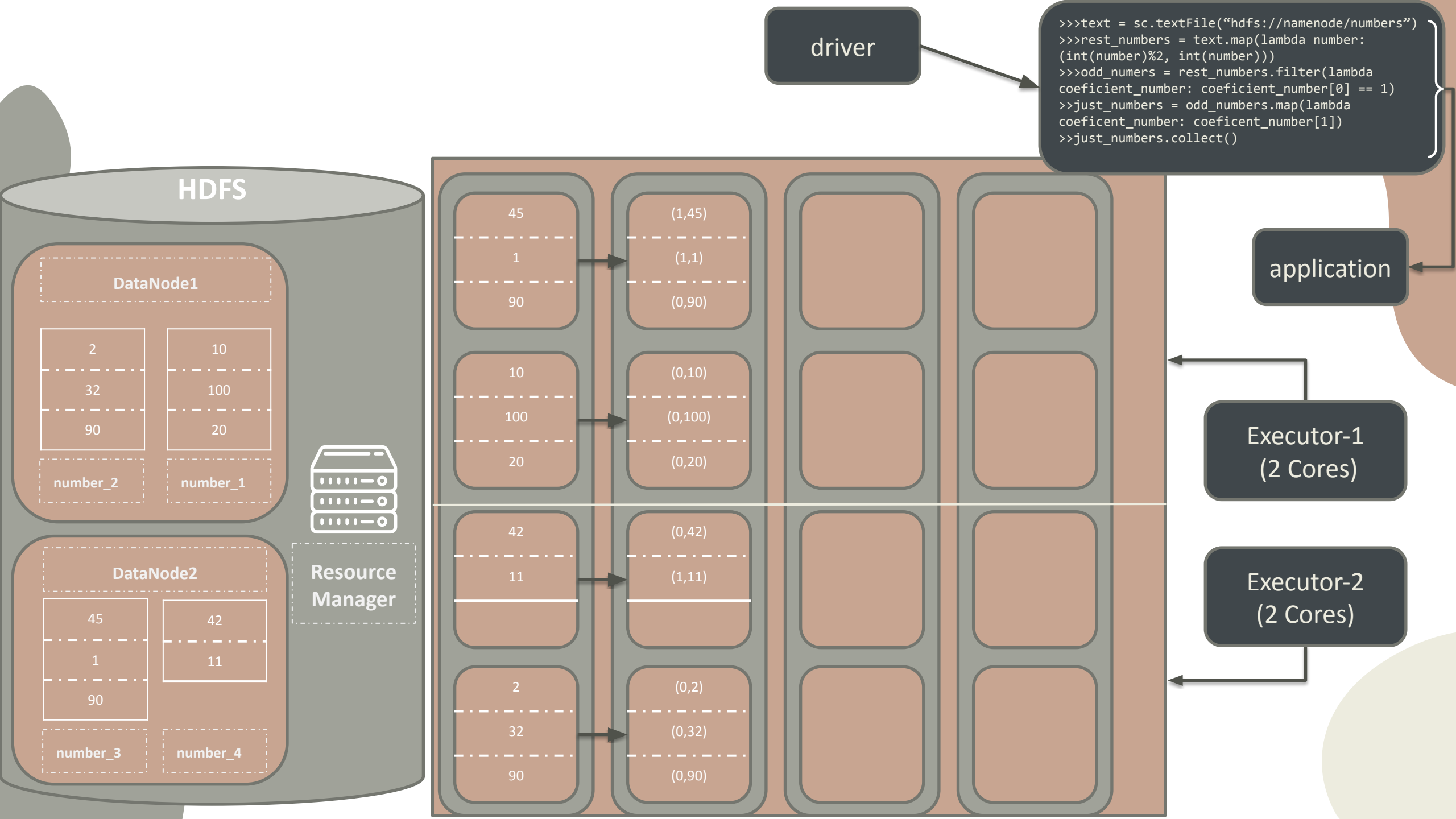

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number:
(int(number)%2, int(number)))
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
Resource Manager
DataNode2
45
1
90
42
11
number_3
number_4
Executor-1
(2 Cores)
Executor-2
(2 Cores)

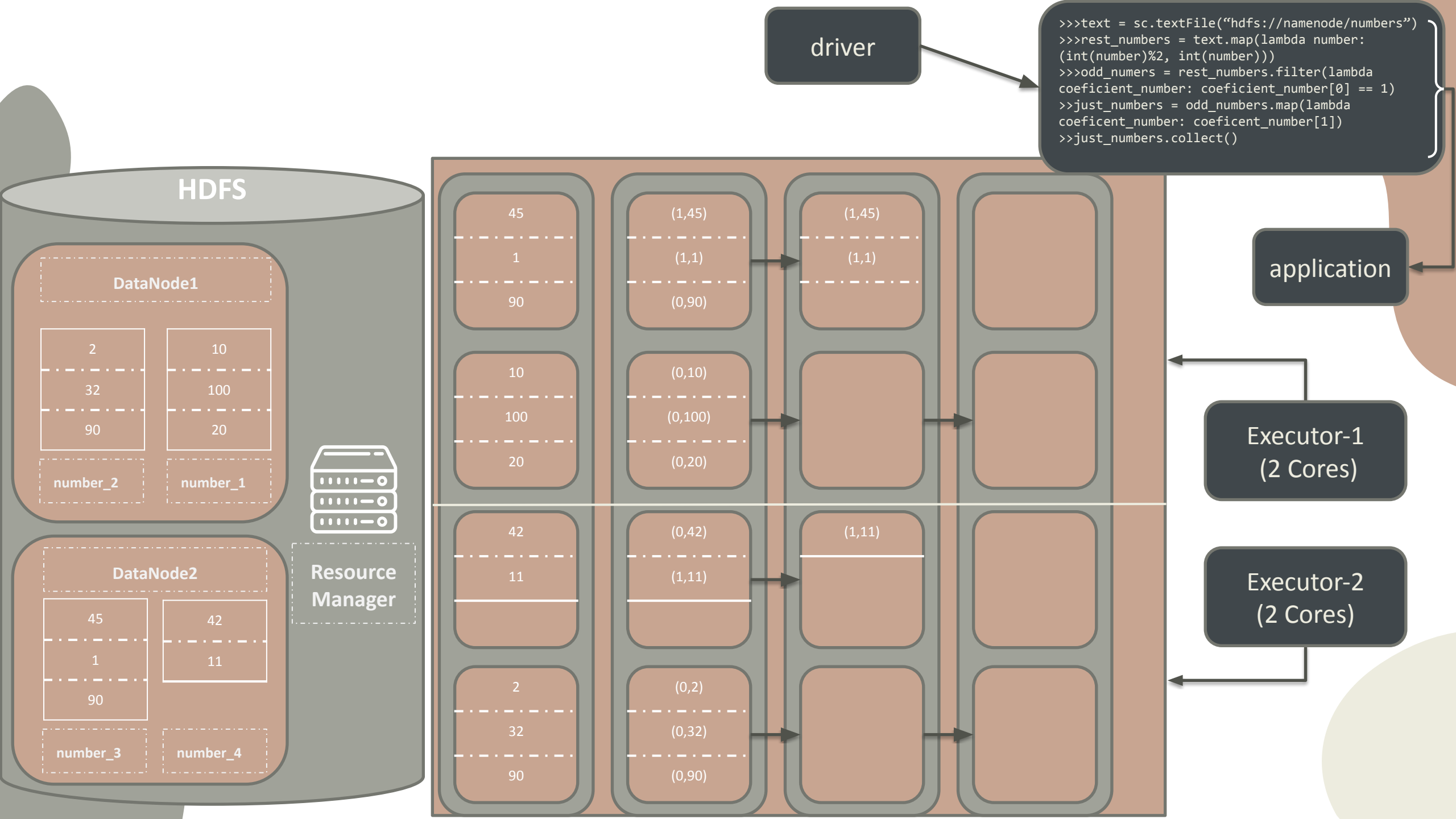

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number:
(int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda
coeficient_number: coeficient_number[0] == 1)
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
Resource Manager
DataNode2
45
1
90
42
11
number_3
number_4
Executor-1
(2 Cores)
Executor-2
(2 Cores)

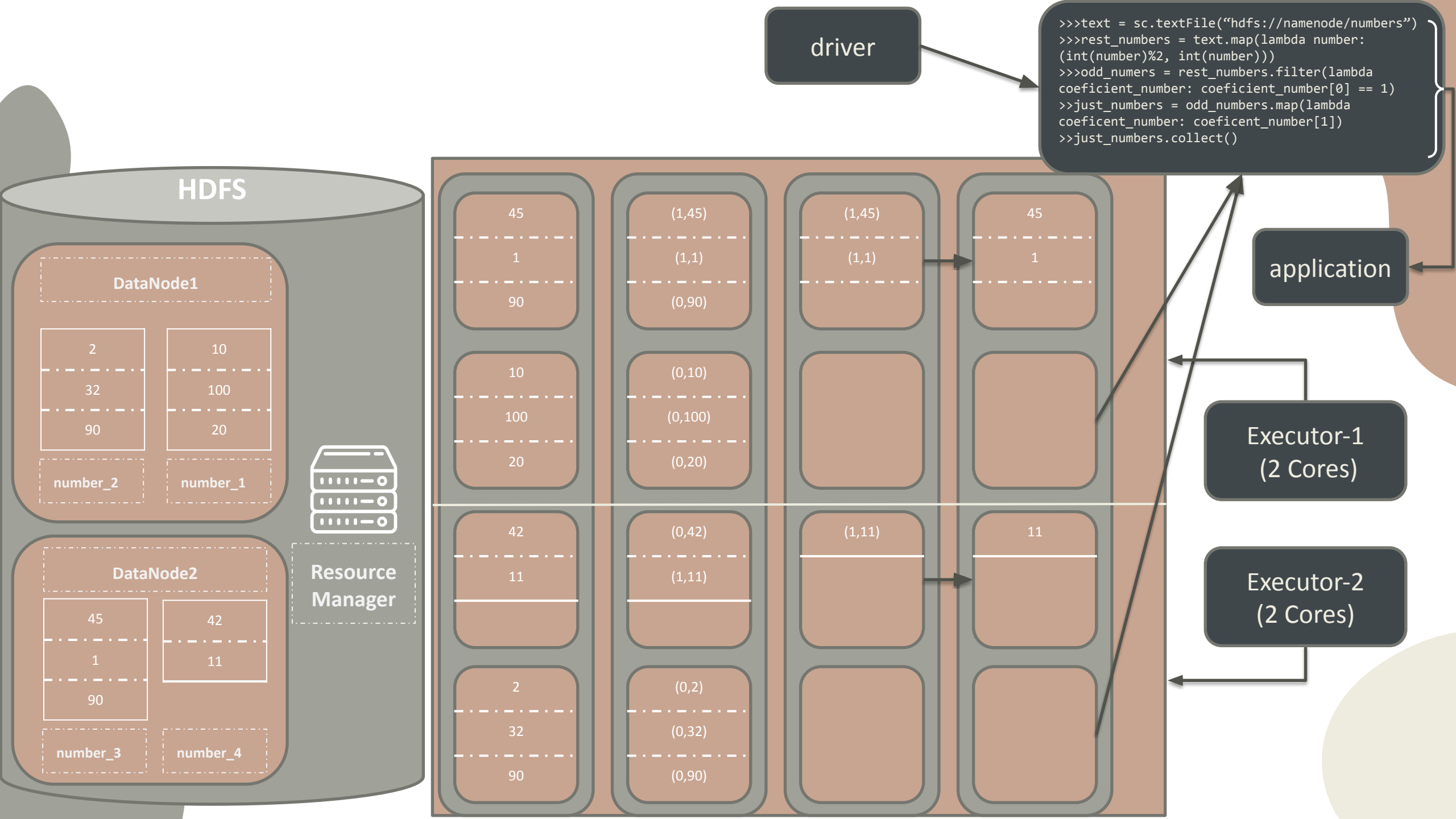

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda coeficient_number: coeficient_number[0] == 1)
>>just_numbers = odd_numbers.map(lambda coeficent_number: coeficent_number[1])
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
Resource Manager
DataNode2
45
1
90
42
11
number_3
number_4
Executor-1 (2 Cores)
Executor-2 (2 Cores)

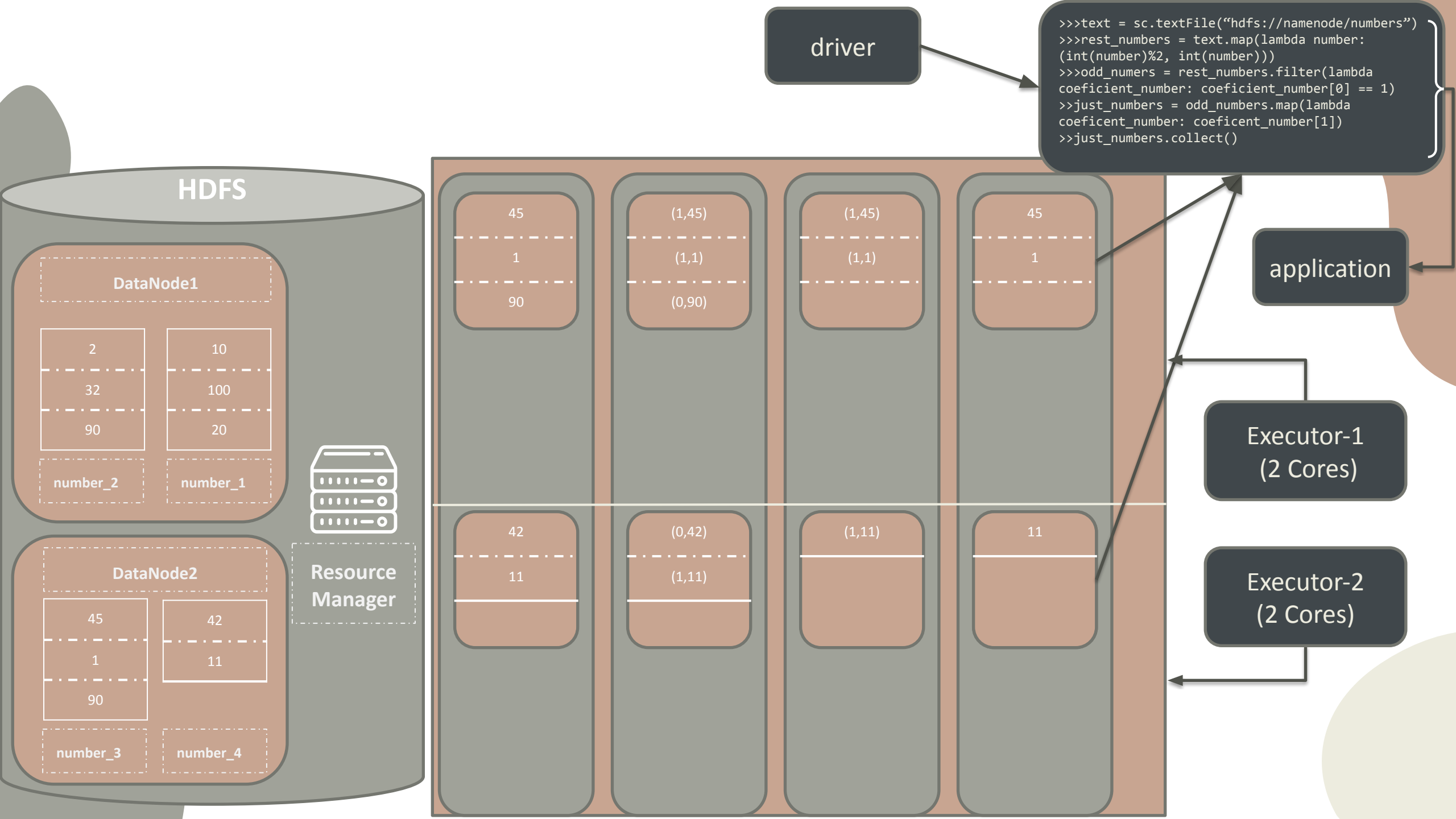

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number:
(int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda
coeficient_number: coeficient_number[0] == 1)
>>just_numbers = odd_numbers.map(lambda
coeficent_number: coeficent_number[1])
>>just_numbers.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
Resource Manager
DataNode2
45
1
90
42
11
number_3
number_4
Executor-1 (2 Cores)
Executor-2 (2 Cores)

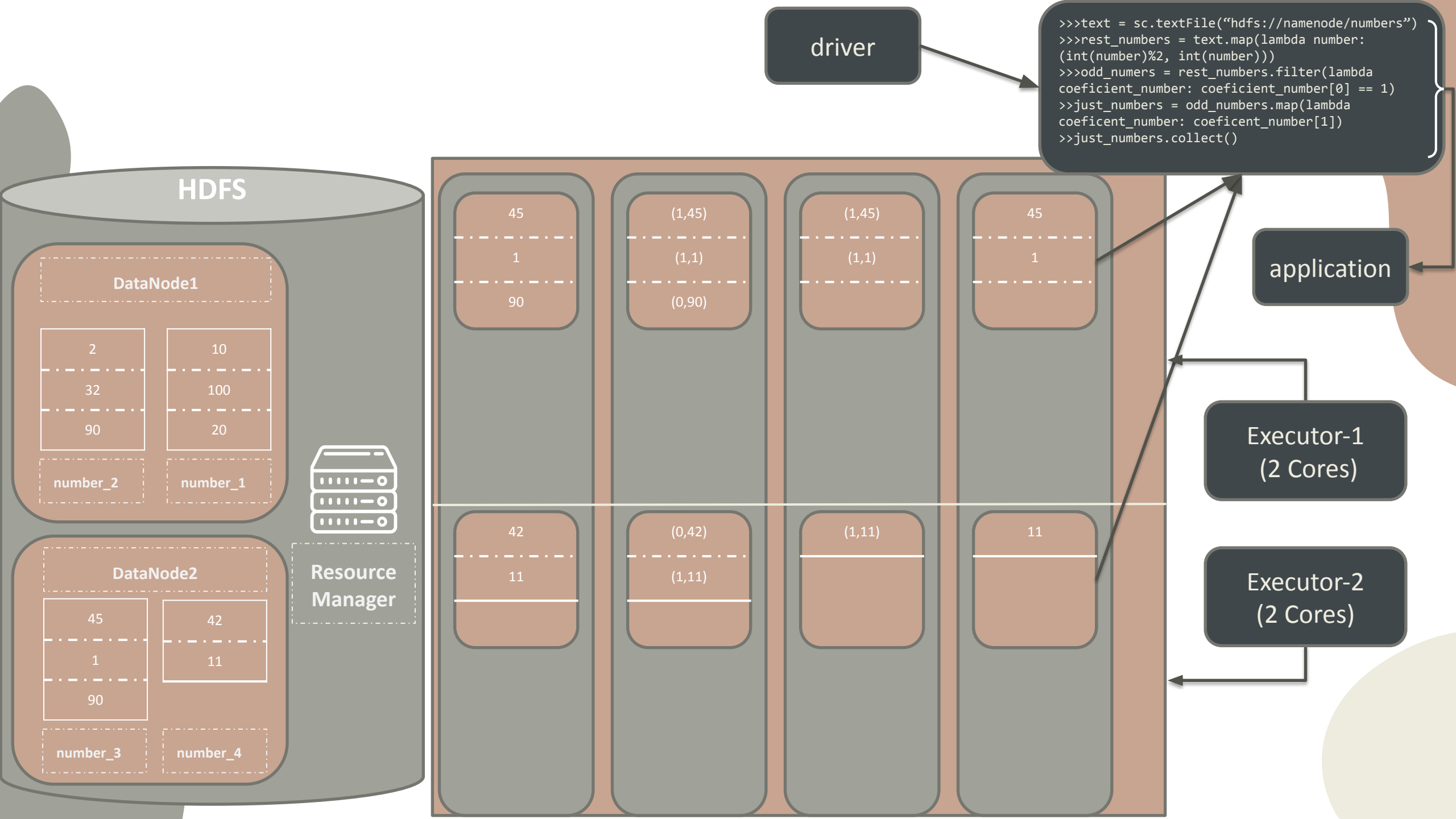

baseRDD
stage
driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda coeficient_number: coeficient_number[0] == 1)
>>just_numbers = odd_numbers.map(lambda coeficent_number: coeficent_number[1])
>>just_numbers.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
45
1
90
10
100
20
42
11
2
32
90
Executor-1 (2 Cores)
Executor-2 (2 Cores)
Task

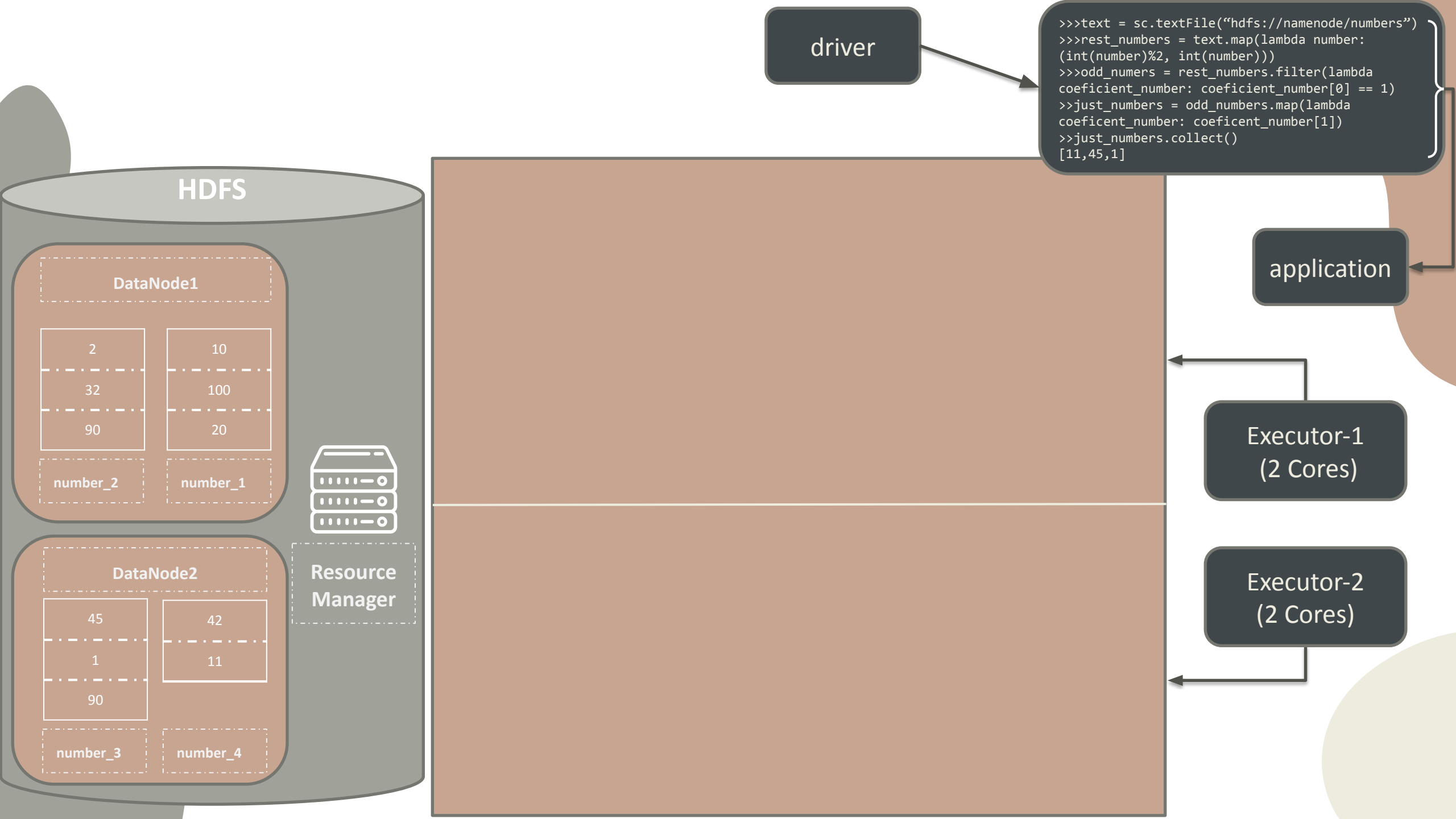

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda coeficient_number: coeficient_number[0] == 1)
>>just_numbers = odd_numbers.map(lambda coeficent_number: coeficent_number[1])
>>just_numbers.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
45
1
90
(1,45)
(1,1)
(0,90)
10
100
20
(0,10)
(0,100)
(0,20)
42
11
(0,42)
(1,11)
2
32
90
(0,2)
(0,32)
(0,90)
Executor-1 (2 Cores)
Executor-2 (2 Cores)

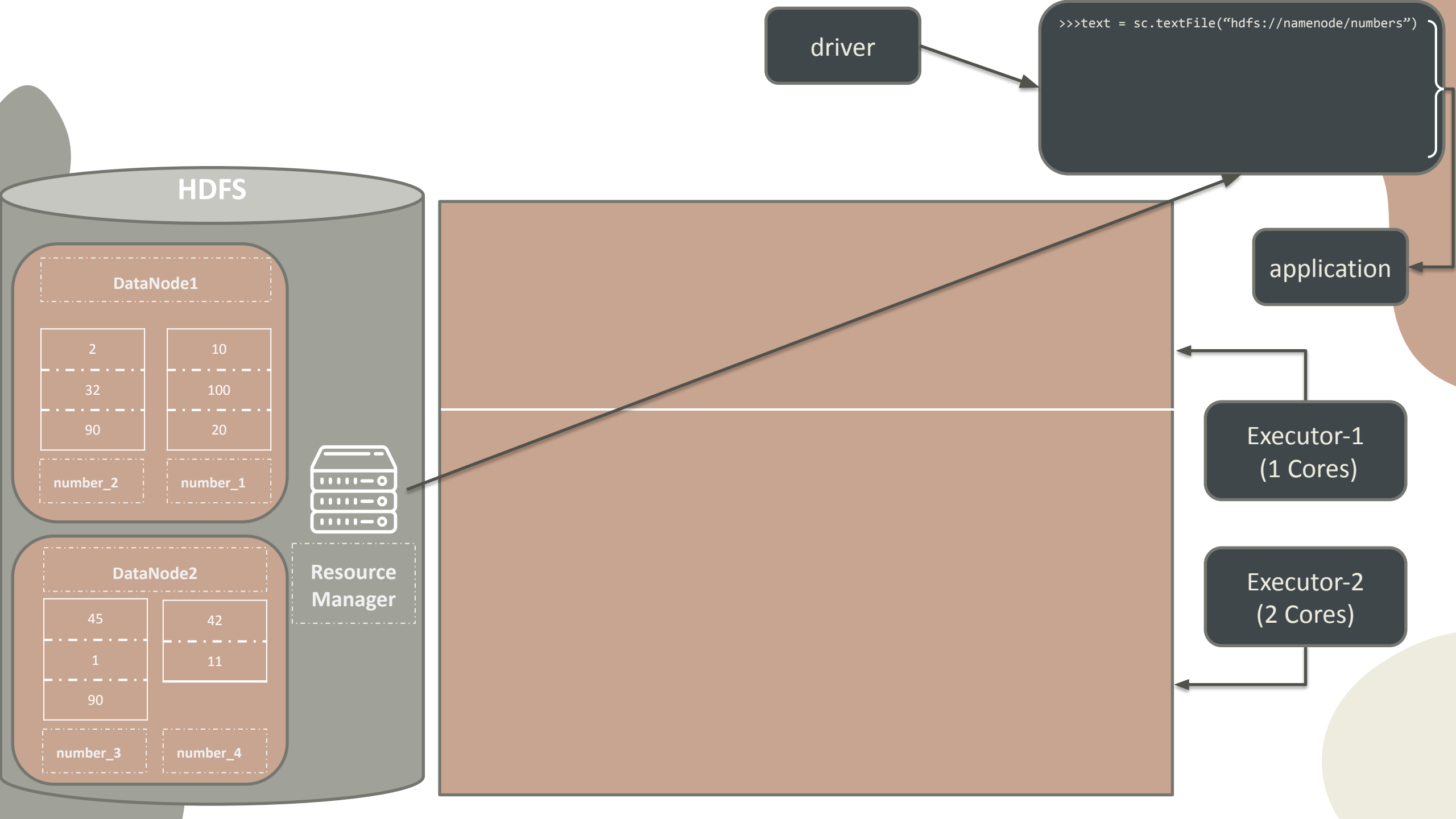

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda coeficient_number: coeficient_number[0] == 1)
>>just_numbers = odd_numbers.map(lambda coeficent_number: coeficent_number[1])
>>just_numbers.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
45
1
90
(1,45)
(1,1)
(0,90)
(1,45)
(1,1)
10
100
20
(0,10)
(0,100)
(0,20)
42
11
(0,42)
(1,11)
(1,11)
2
32
90
(0,2)
(0,32)
(0,90)
Executor-1
(2 Cores)
Executor-2
(2 Cores)

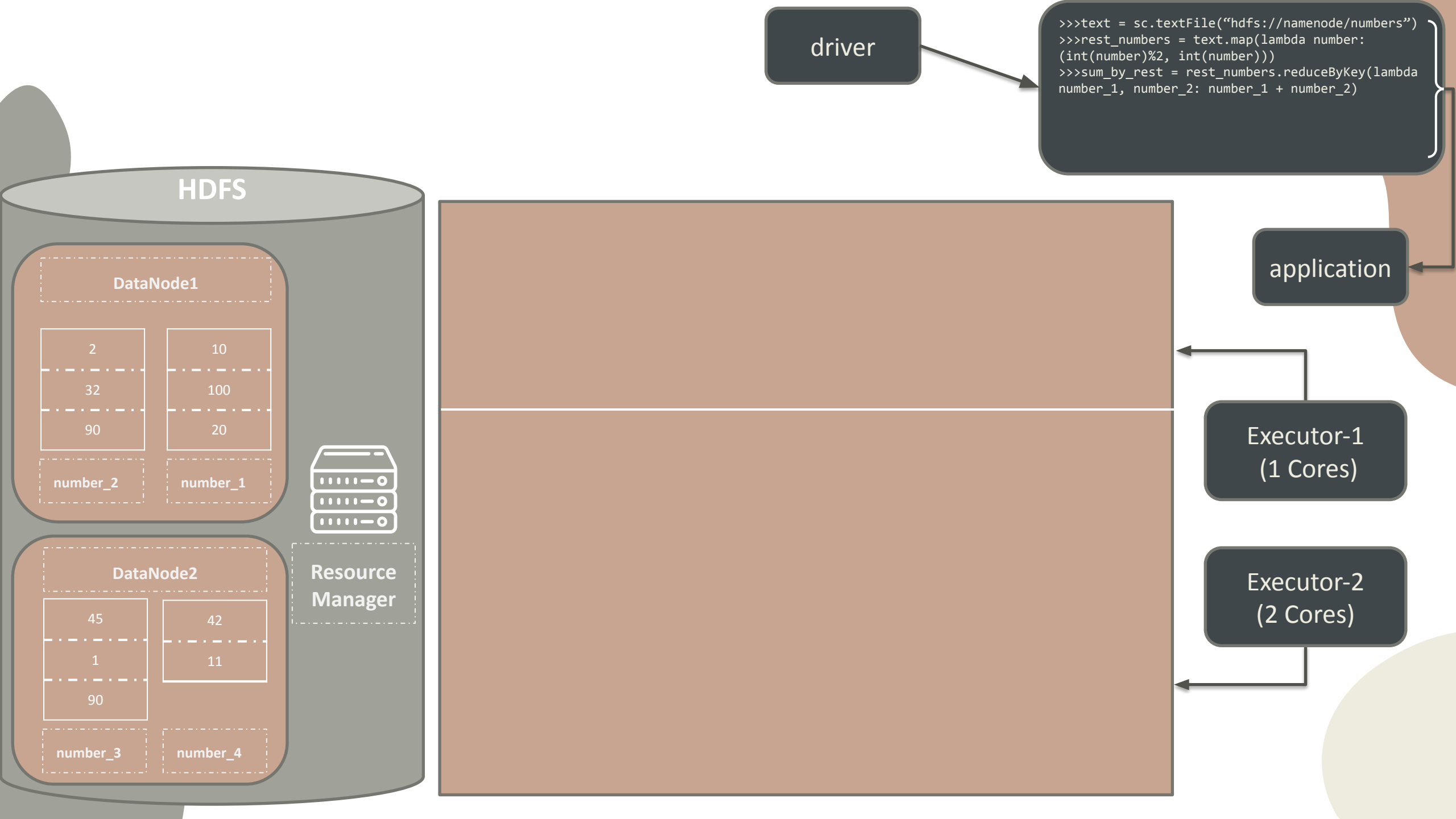

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda coeficient_number: coeficient_number[0] == 1)
>>just_numbers = odd_numbers.map(lambda coeficent_number: coeficent_number[1])
>>just_numbers.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
45
1
90
(1,45)
(1,1)
(0,90)
(1,45)
(1,1)
45
1
10
100
20
(0,10)
(0,100)
(0,20)
42
11
(0,42)
(1,11)
(1,11)
11
2
32
90
(0,2)
(0,32)
(0,90)
Executor-1
(2 Cores)
Executor-2
(2 Cores)

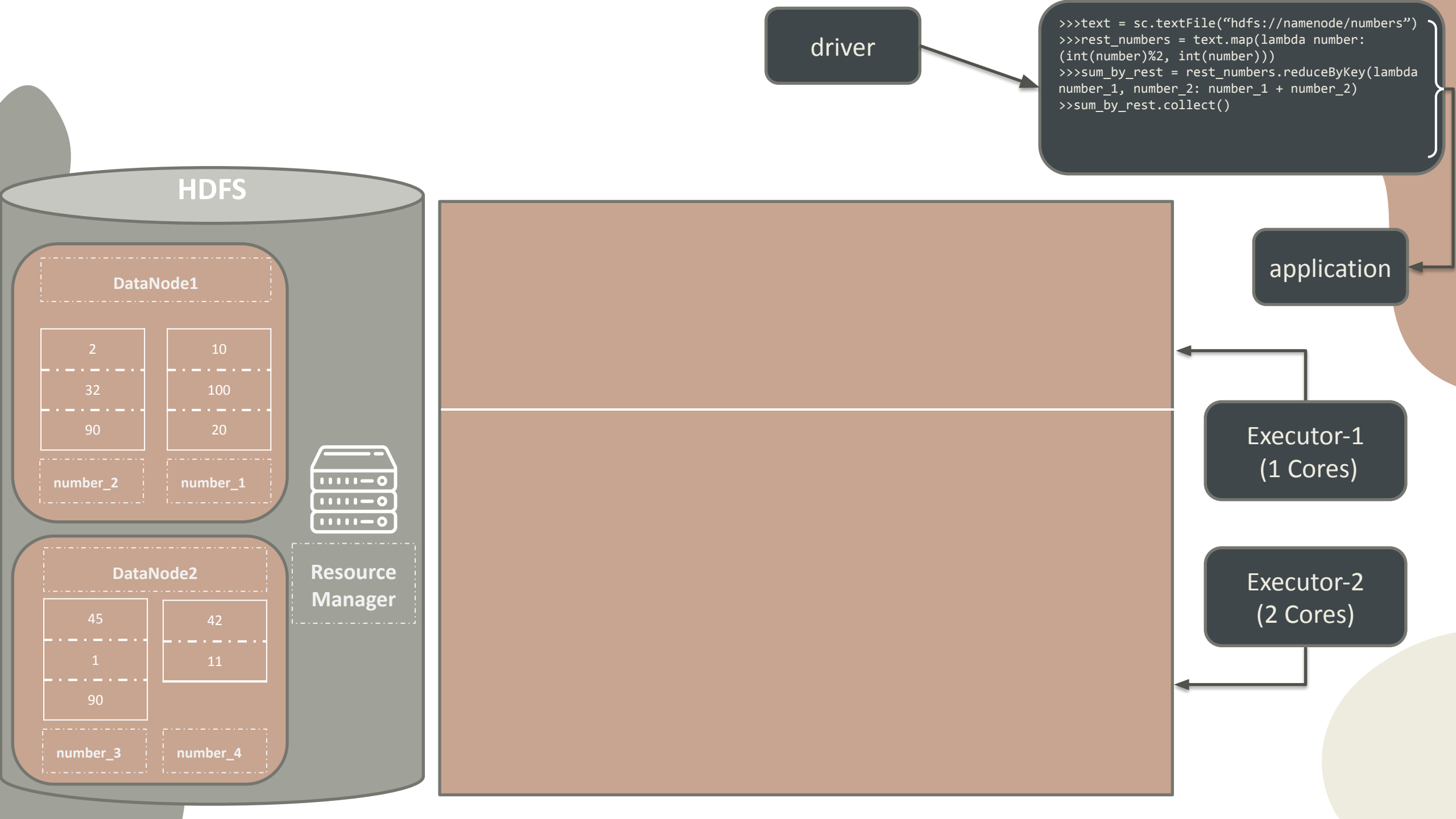

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda coeficient_number: coeficient_number[0] == 1)
>>just_numbers = odd_numbers.map(lambda coeficent_number: coeficent_number[1])
>>just_numbers.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
45
1
90
(1,45)
(1,1)
(0,90)
(1,45)
(1,1)
45
1
42
11
(0,42)
(1,11)
(1,11)
11
Executor-1 (2 Cores)
Executor-2 (2 Cores)

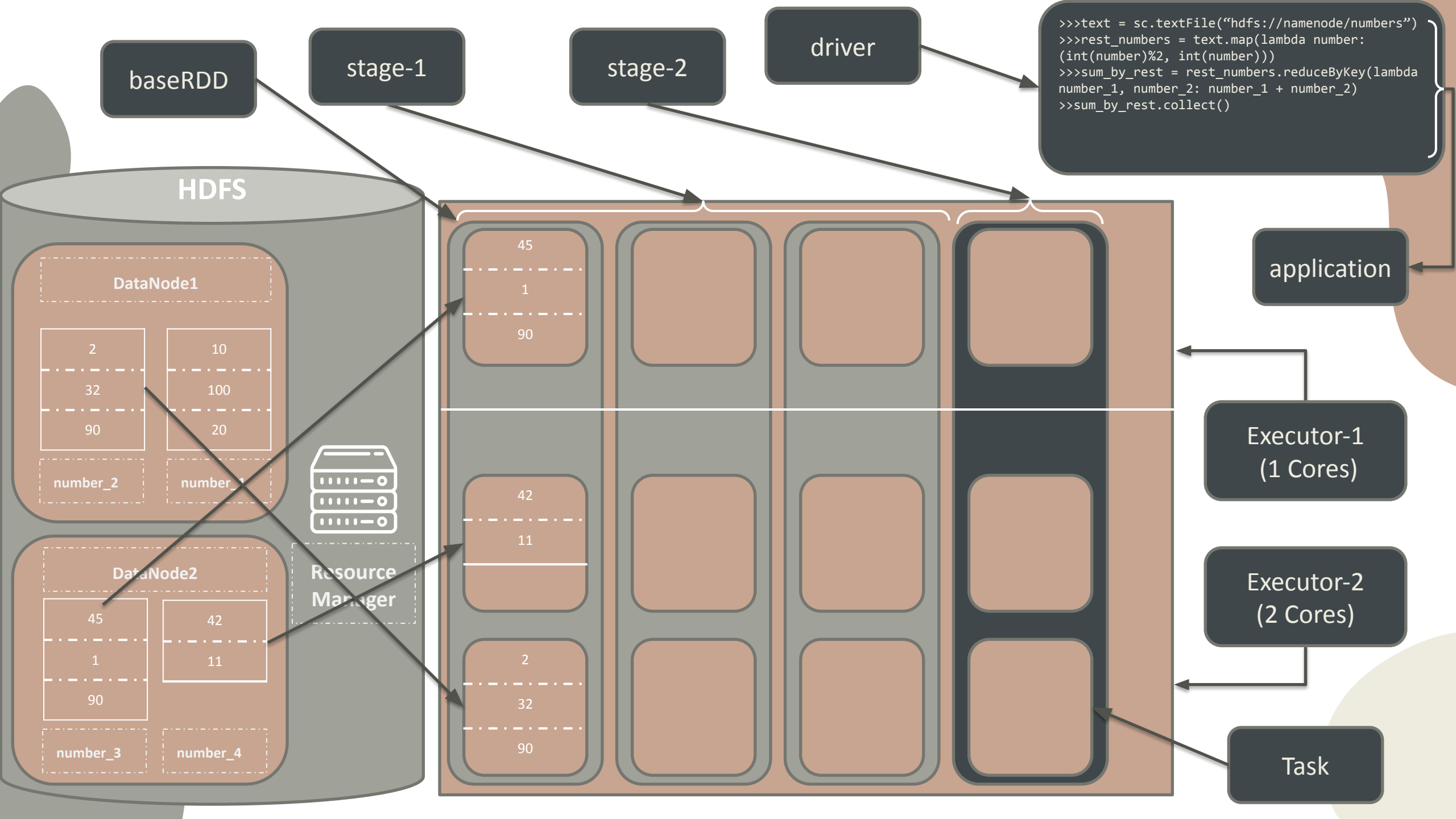

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda coeficient_number: coeficient_number[0] == 1)
>>just_numbers = odd_numbers.map(lambda coeficent_number: coeficent_number[1])
>>just_numbers.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
45
1
90
(1,45)
(1,1)
(0,90)
(1,45)
(1,1)
45
1
42
11
(0,42)
(1,11)
(1,11)
11
Executor-1 (2 Cores)
Executor-2 (2 Cores)

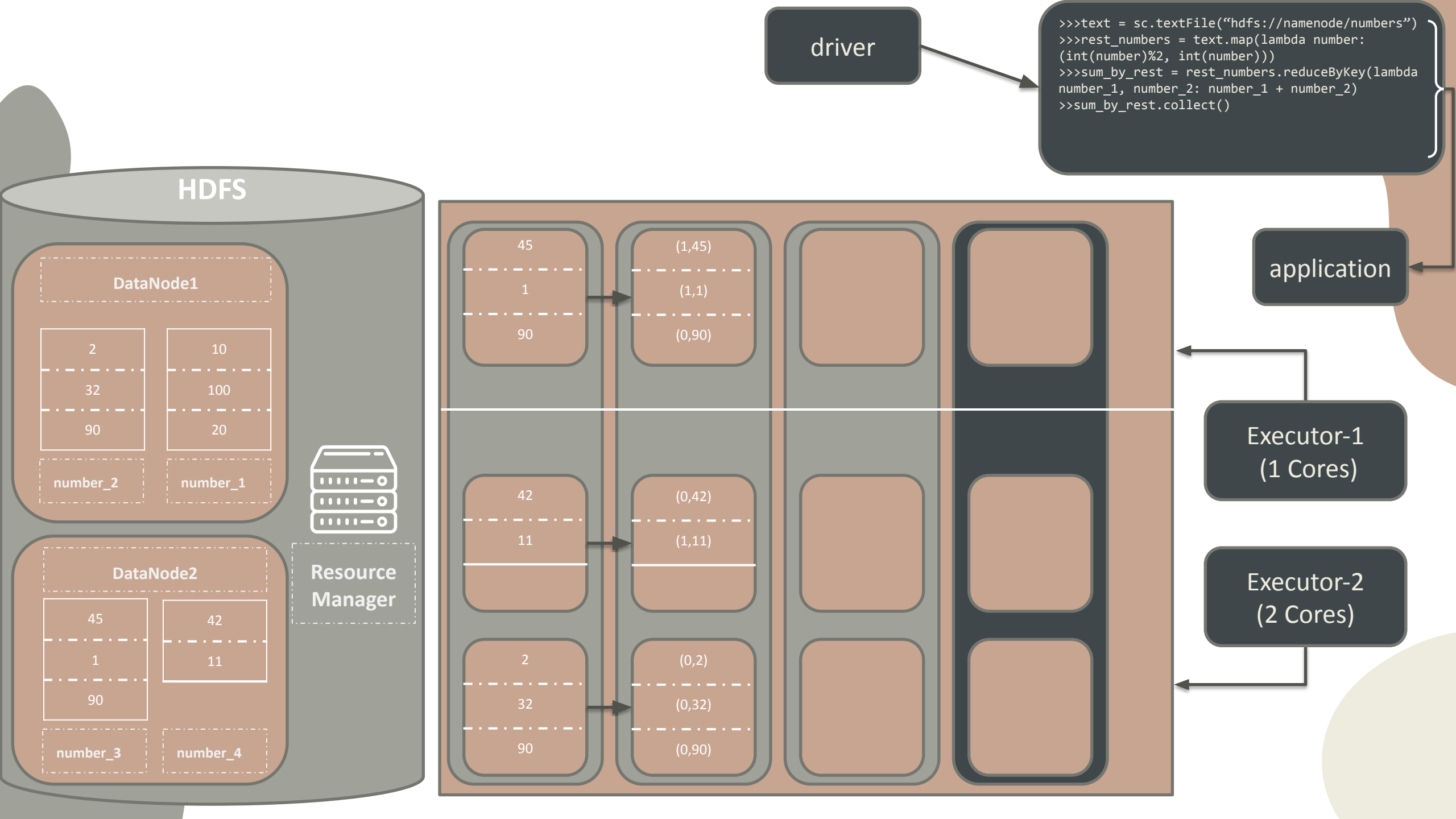

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number:
(int(number)%2, int(number)))
>>>odd_numers = rest_numbers.filter(lambda
coeficient_number: coeficient_number[0] == 1)
>>just_numbers = odd_numbers.map(lambda
coeficent_number: coeficent_number[1])
>>just_numbers.collect()
[11,45,1]
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
Resource Manager
DataNode2
45
1
90
42
11
number_3
number_4
Executor-1
(2 Cores)
Executor-2
(2 Cores)

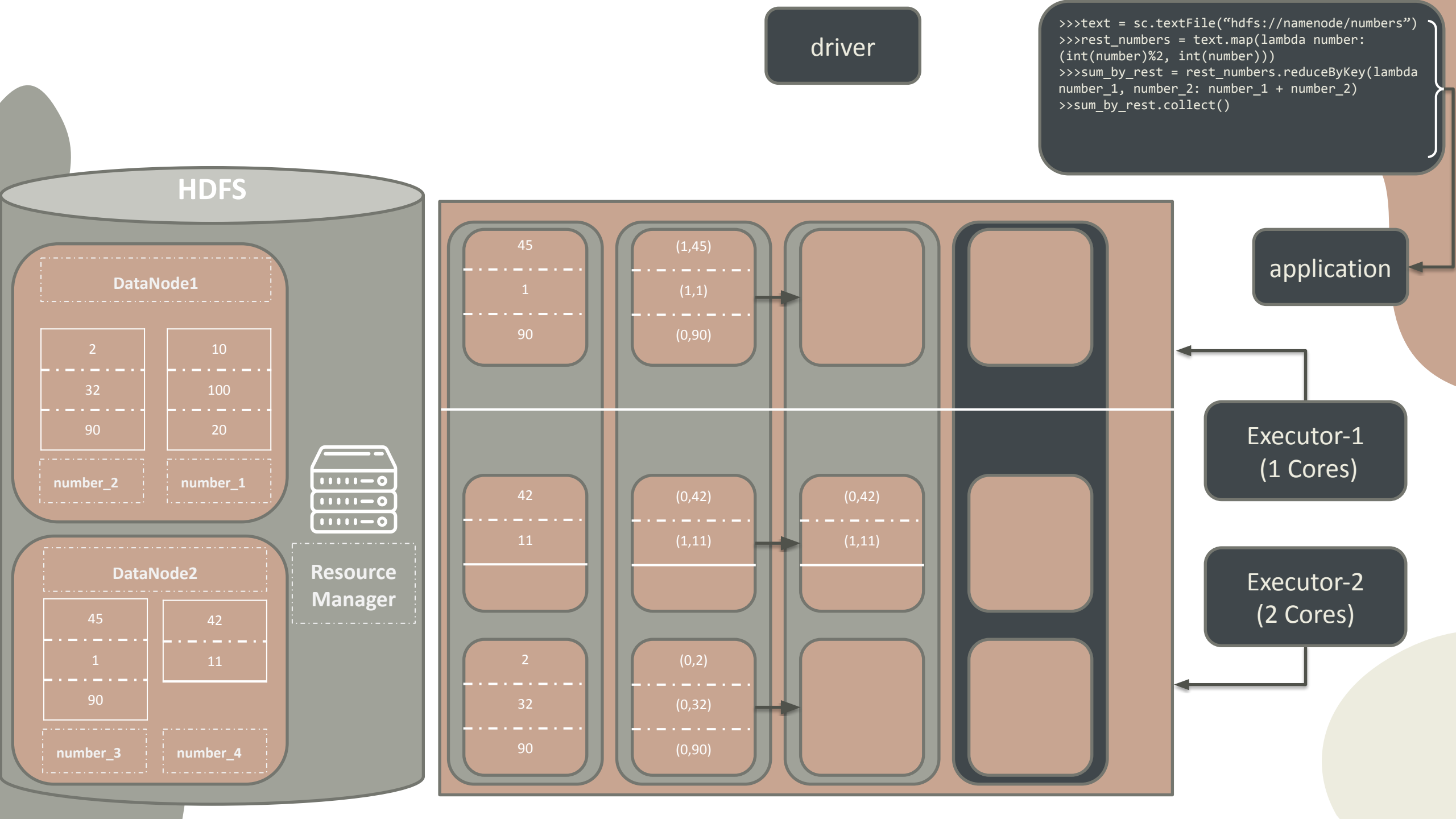

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
Executor-1 (1 Cores)
Executor-2 (2 Cores)

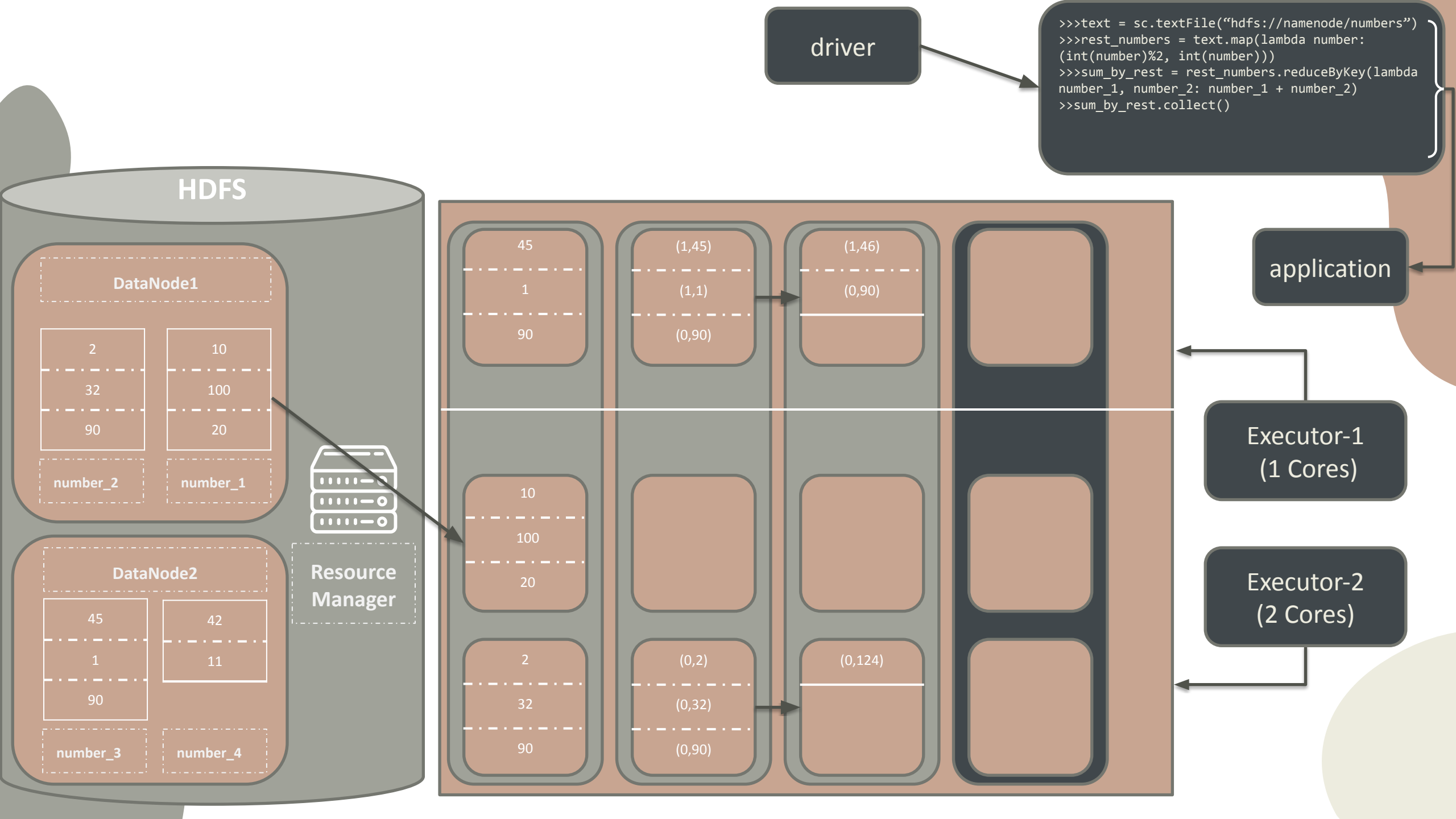

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
Executor-1 (1 Cores)
Executor-2 (2 Cores)

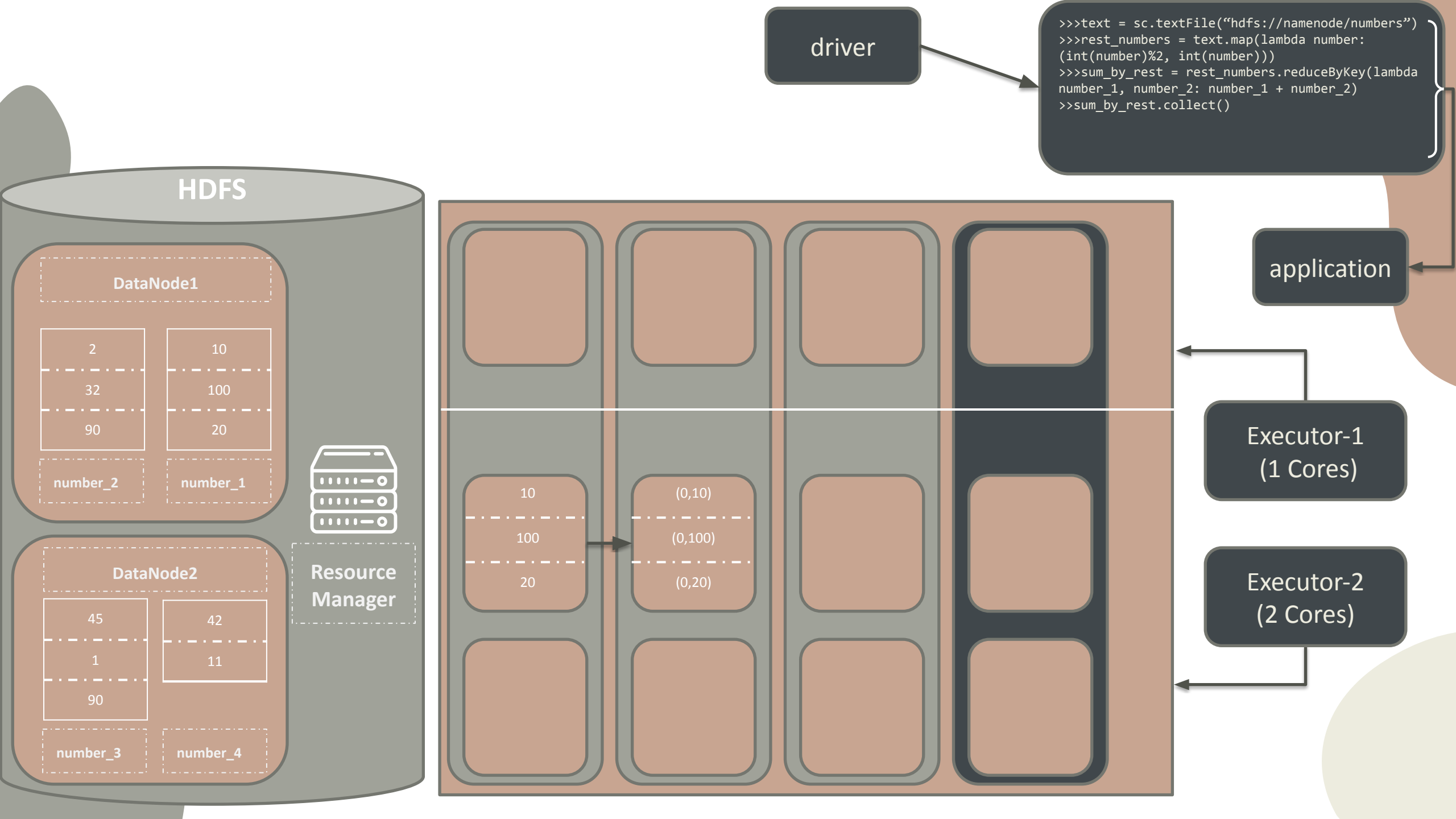

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
application
Executor-1 (1 Cores)
Executor-2 (2 Cores)

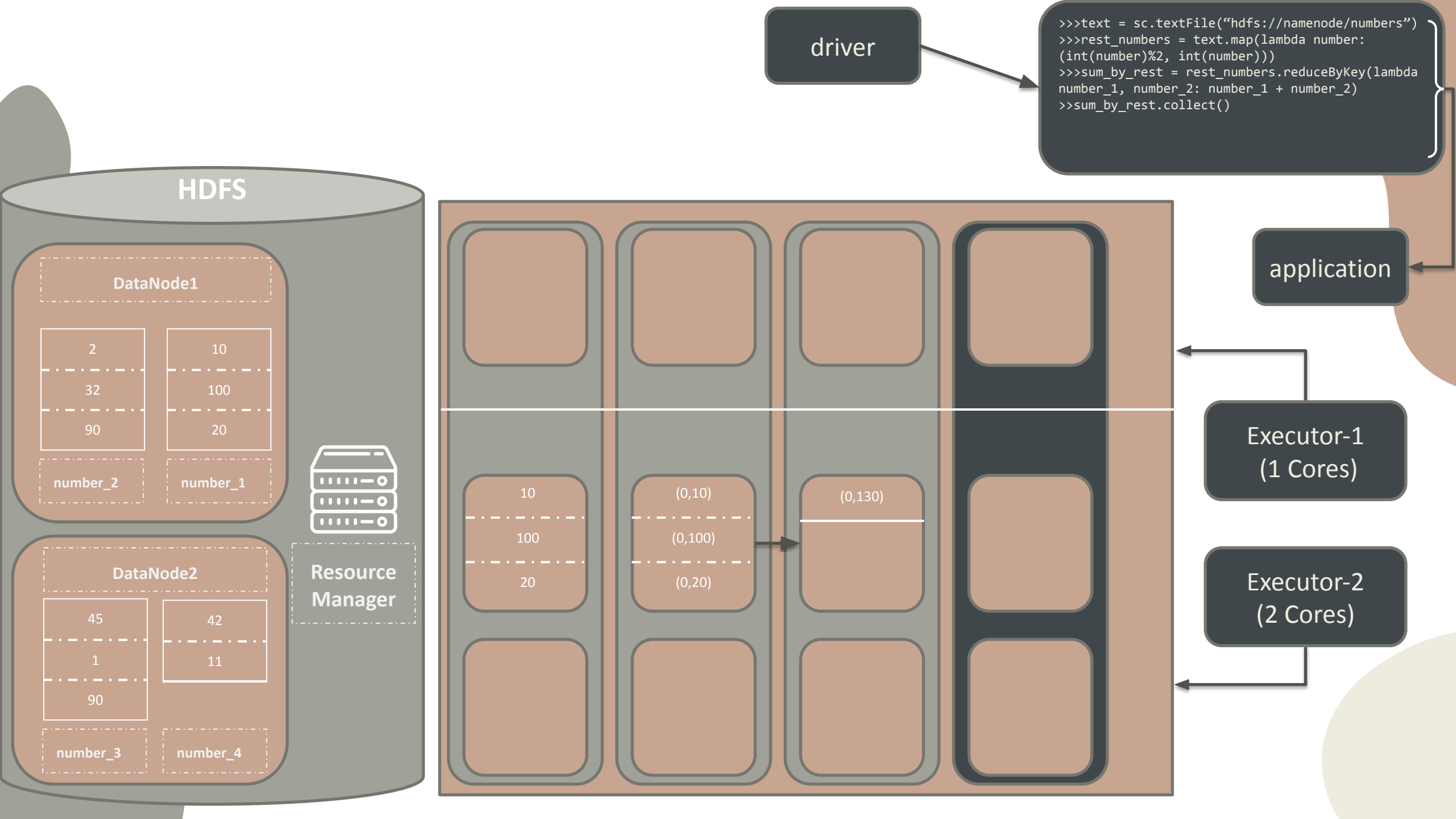

baseRDD
stage-1
stage-2
driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
45
1
90
42
11
2
32
90
Executor-1 (1 Cores)
Executor-2 (2 Cores)
Task

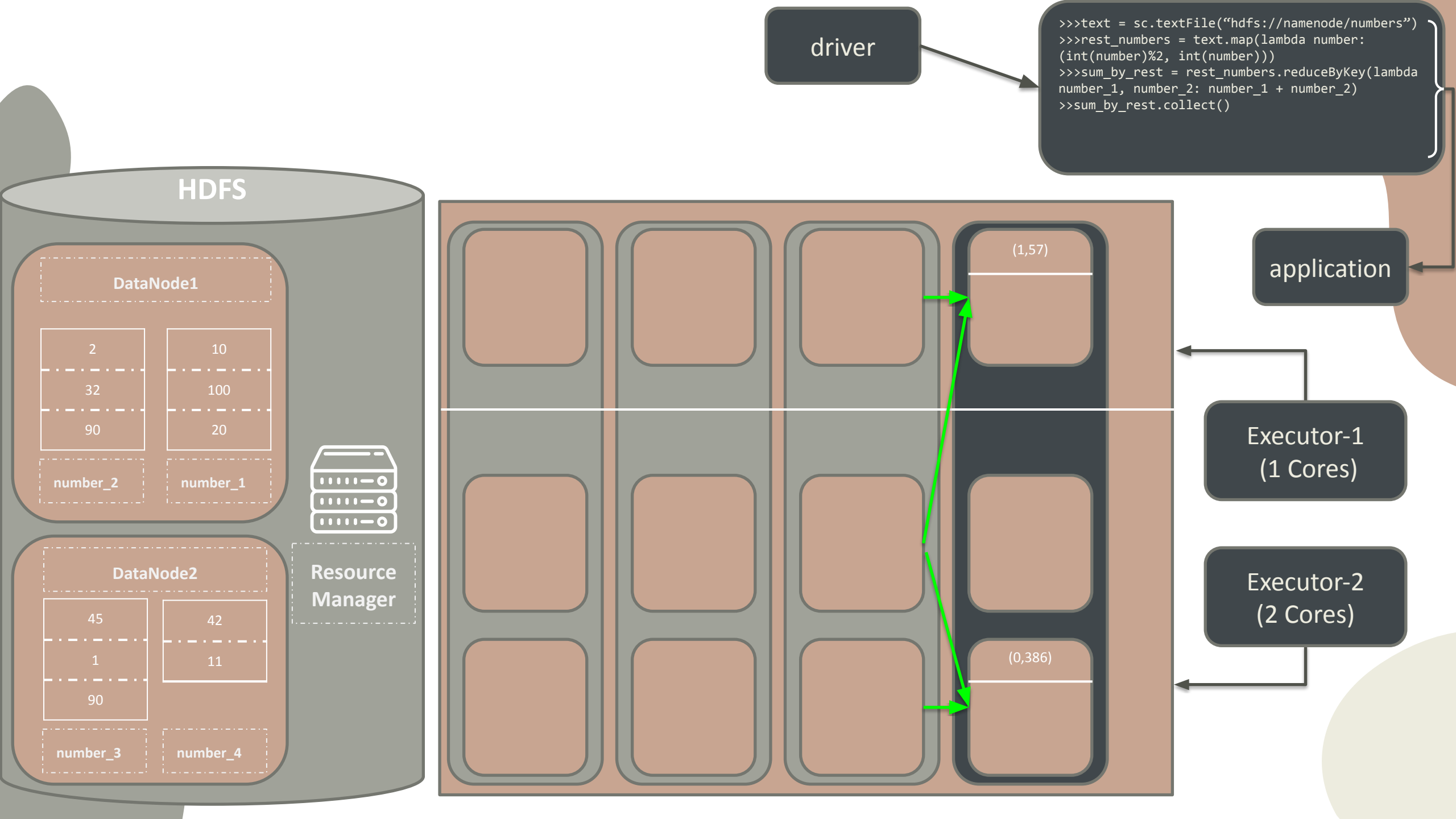

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
45
1
90
(1,45)
(1,1)
(0,90)
42
11
(0,42)
(1,11)
2
32
90
(0,2)
(0,32)
(0,90)
Executor-1 (1 Cores)
Executor-2 (2 Cores)

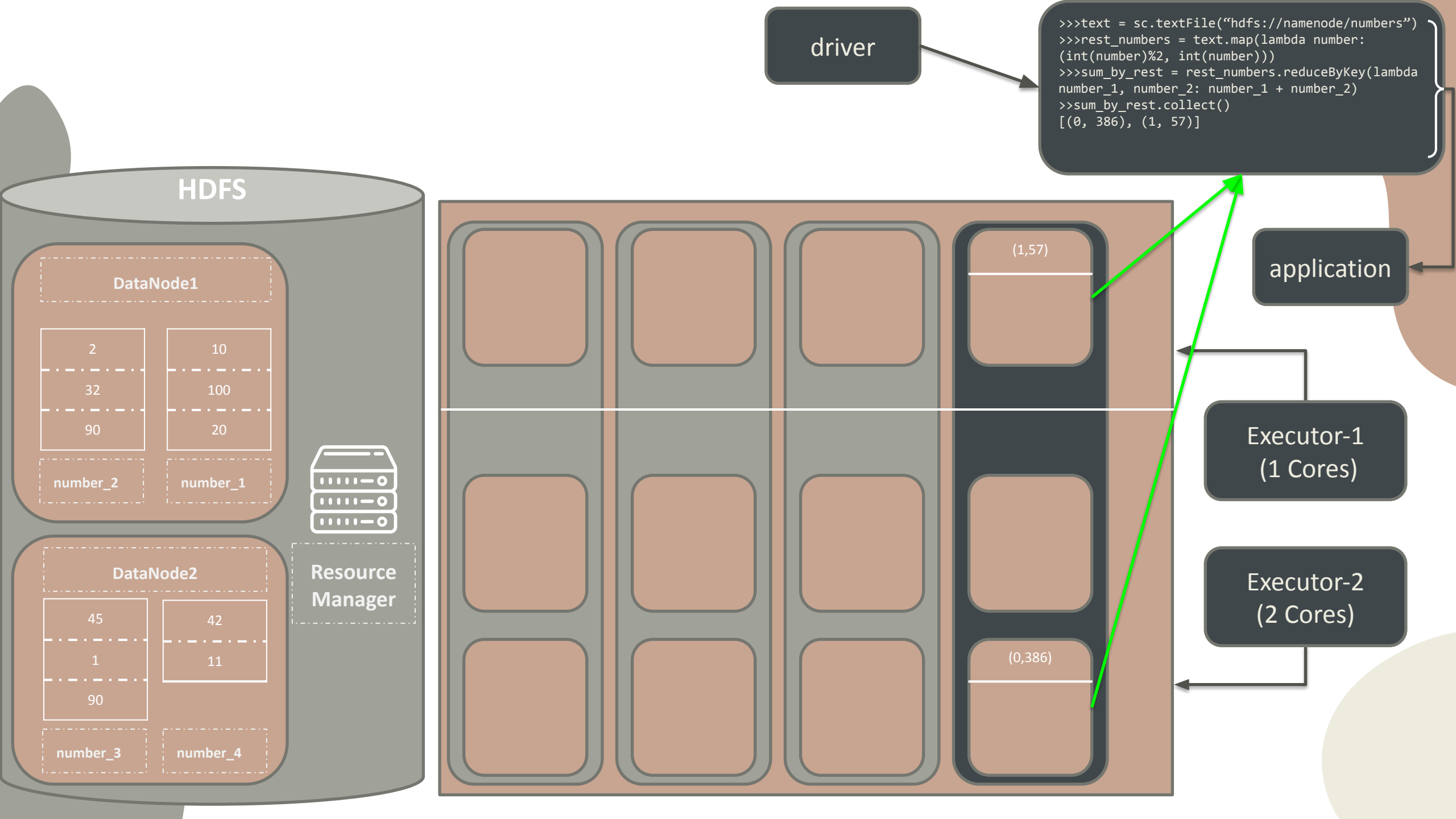

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
45
1
90
(1,45)
(1,1)
(0,90)
42
11
(0,42)
(1,11)
(0,42)
(1,11)
2
32
90
(0,2)
(0,32)
(0,90)
Executor-1 (1 Cores)
Executor-2 (2 Cores)

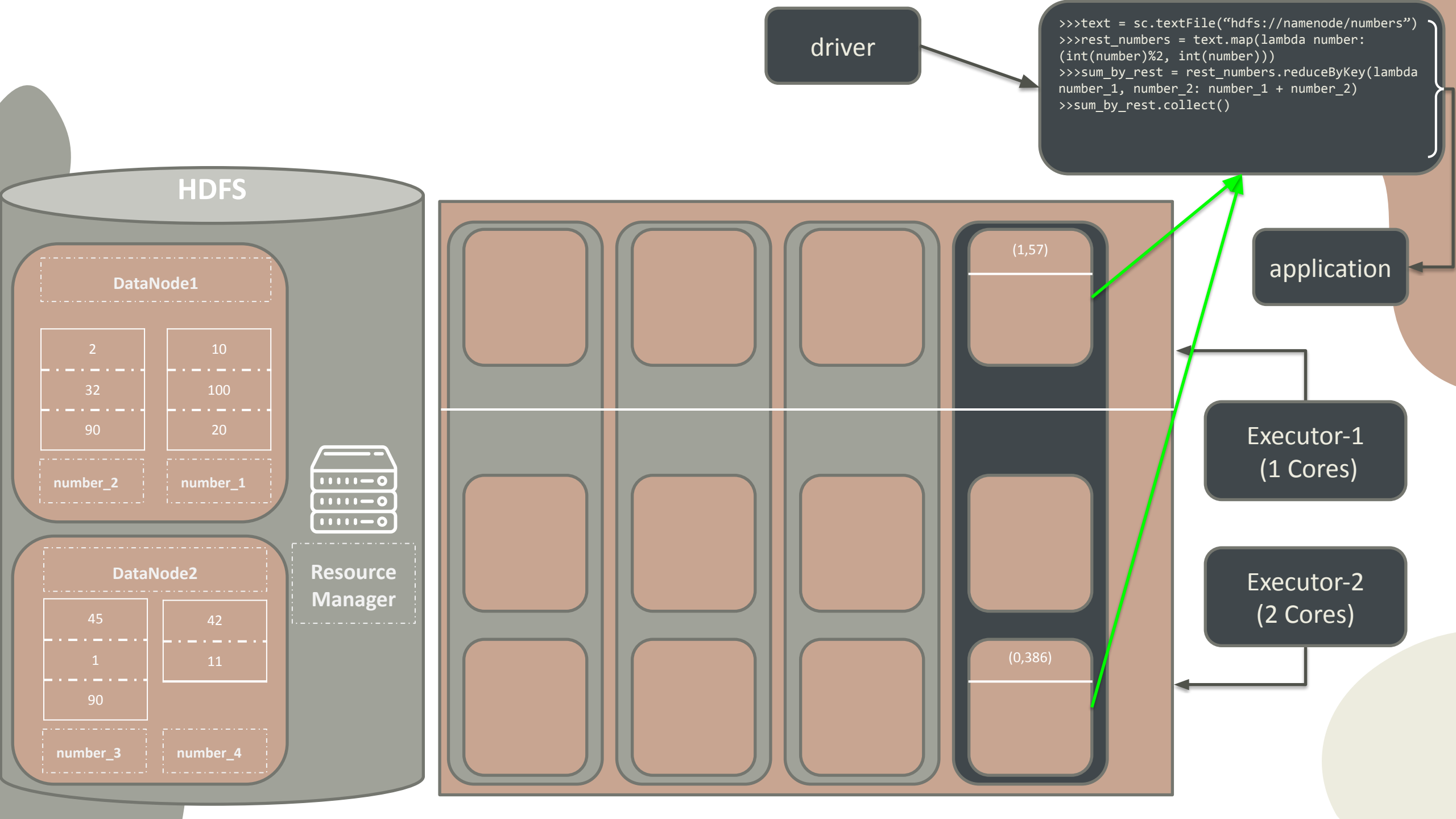

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
application
HDFS
DataNode1
2
32
90
number_2
10
100
20
number_1
DataNode2
45
1
90
number_3
42
11
number_4
Resource Manager
45
1
90
(1,45)
(1,1)
(0,90)
(1,46)
(0,90)
10
100
20
2
32
90
(0,2)
(0,32)
(0,90)
(0,124)
Executor-1
(1 Cores)
Executor-2
(2 Cores)

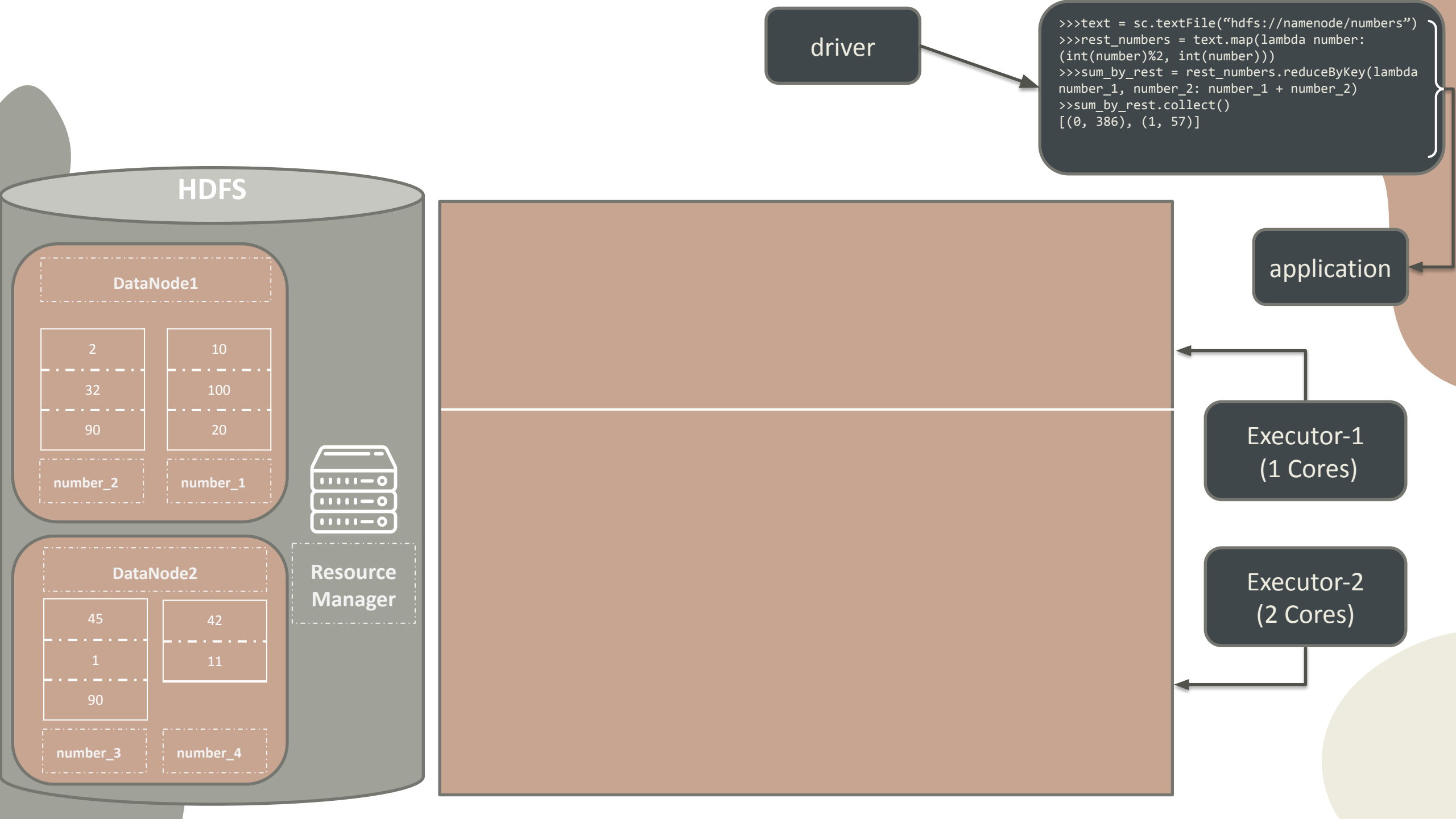

driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
10
100
20
(0,10)
(0,100)
(0,20)
Executor-1 (1 Cores)
Executor-2 (2 Cores)


driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
10
100
20
(0,10)
(0,100)
(0,20)
(0,130)
Executor-1 (1 Cores)
Executor-2 (2 Cores)


driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
(1,57)
(0,386)
application
Executor-1 (1 Cores)
Executor-2 (2 Cores)


driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
[(0, 386), (1, 57)]
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
(1,57)
(0,386)
Executor-1 (1 Cores)
Executor-2 (2 Cores)


driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number: (int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
(1,57)
(0,386)
Executor-1 (1 Cores)
Executor-2 (2 Cores)


driver
>>>text = sc.textFile("hdfs://namenode/numbers")
>>>rest_numbers = text.map(lambda number:
(int(number)%2, int(number)))
>>>sum_by_rest = rest_numbers.reduceByKey(lambda
number_1, number_2: number_1 + number_2)
>>sum_by_rest.collect()
[(0, 386), (1, 57)]
application
HDFS
DataNode1
2
32
90
10
100
20
number_2
number_1
DataNode2
45
1
90
42
11
number_3
number_4
Resource Manager
Executor-1
(1 Cores)
Executor-2
(2 Cores)

Conceptos de computación distribuida