instaclustr
Now part of Spot by NetApp

COMMUNITY
THE ASF CONFERENCE
CODE

Optimizing Apache Spark Data Pipelines on Kubernetes: Leveraging Spot Instances for Production Efficiency.

Hichem Kenniche
Product Architect

October 2023

Our agenda for today

- Exploring the motivation behind running Spark on Kubernetes
  - Architecture, benefits, our background with it
- Harnessing the potential of spot instances for substantial cost savings
  - Running driver on on-demand nodes, picking best spot markets (AZ, instance type)
- Strategies for gracefully handling spot instance interruptions
  - Executor decommissioning and PVC reuse when executor is lost
- Conclusion - Future works and best practices with Spark on Kubernetes

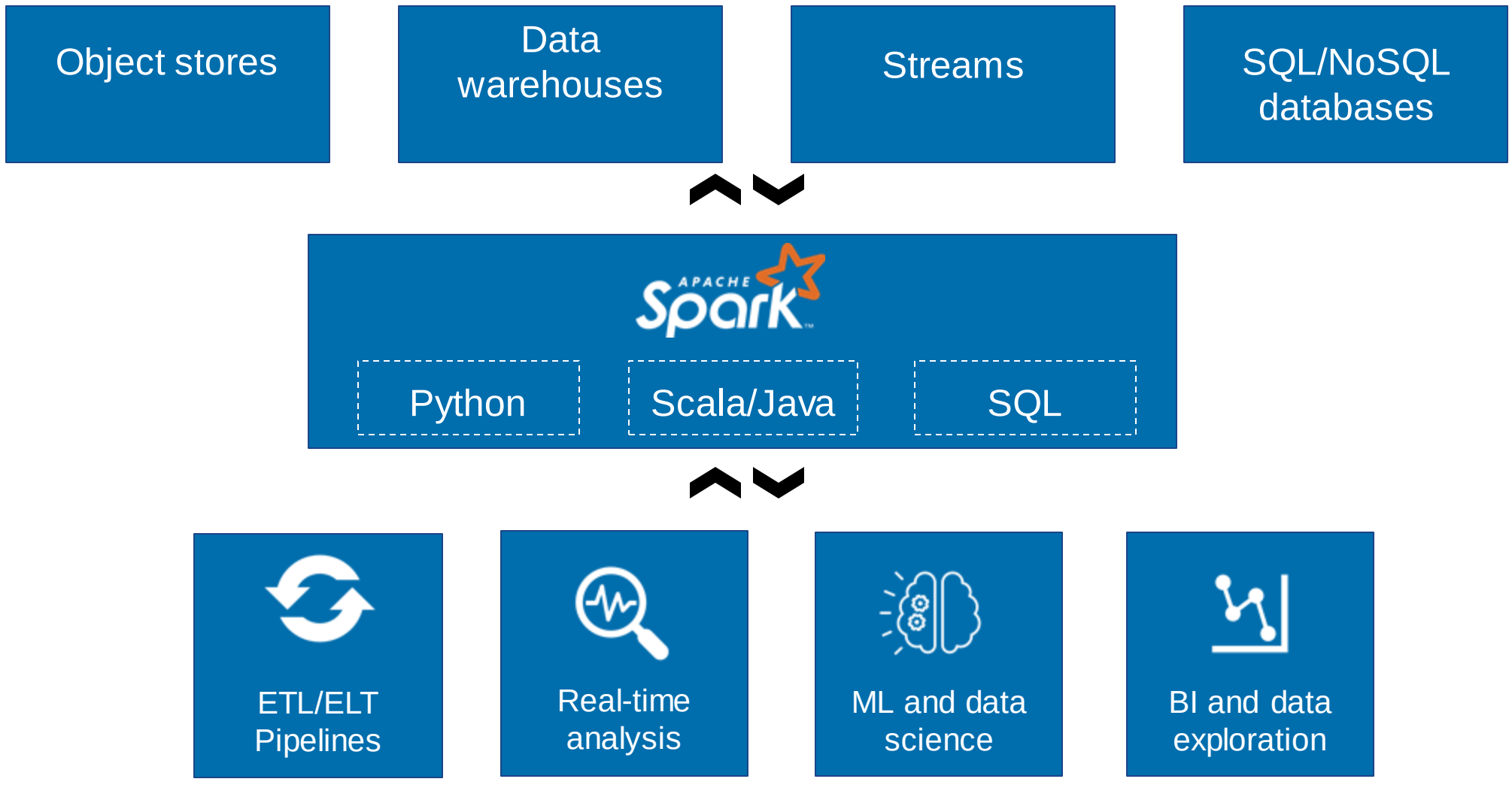
The motivation behind running Spark on Kubernetes

Apache Spark is the **#1** analytics engine for Big Data & AI

Fast
Massively parallelizable, efficient read and write

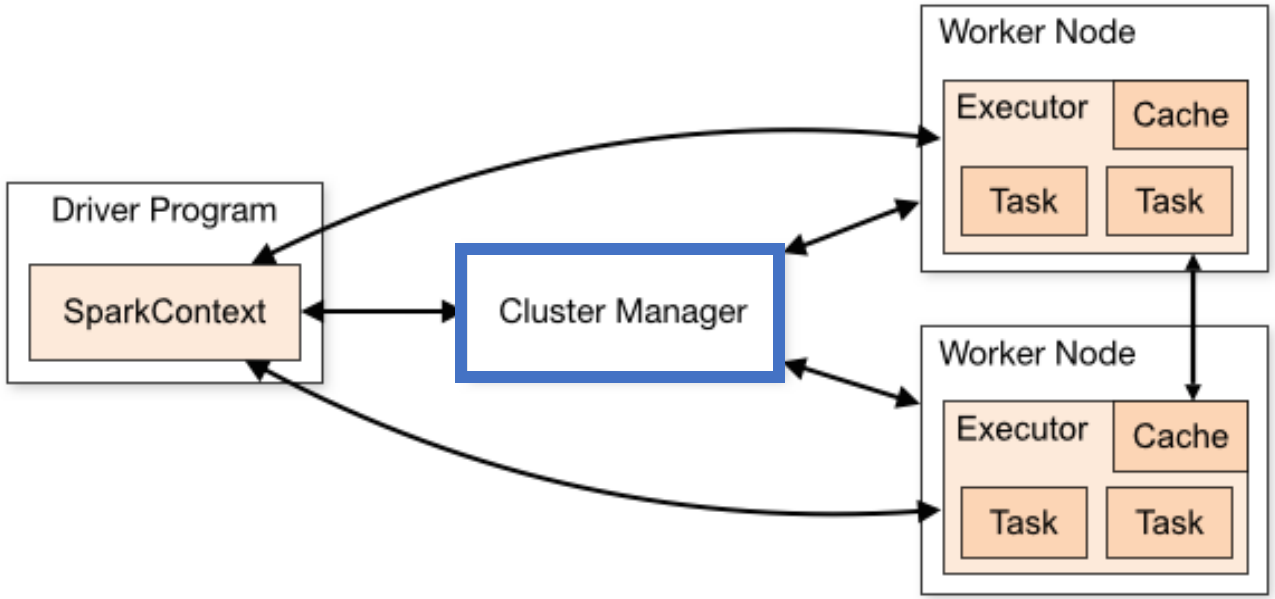
Easy
Interfaces with well-known programming languages

Versatile
Across multiple use cases



The role of resource manager in a Spark cluster

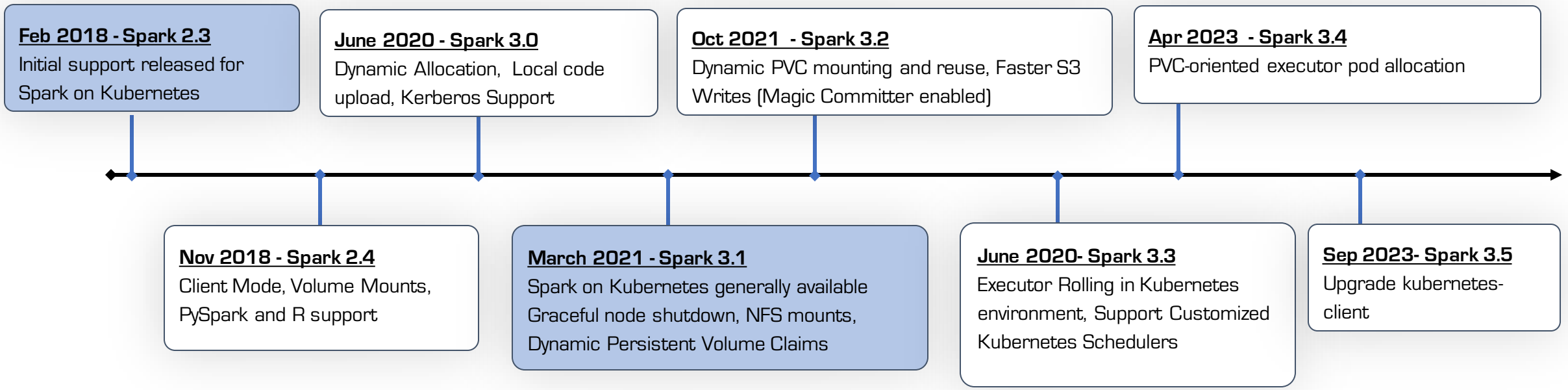
Spark depends on cluster manager for orchestration of a job on a cluster



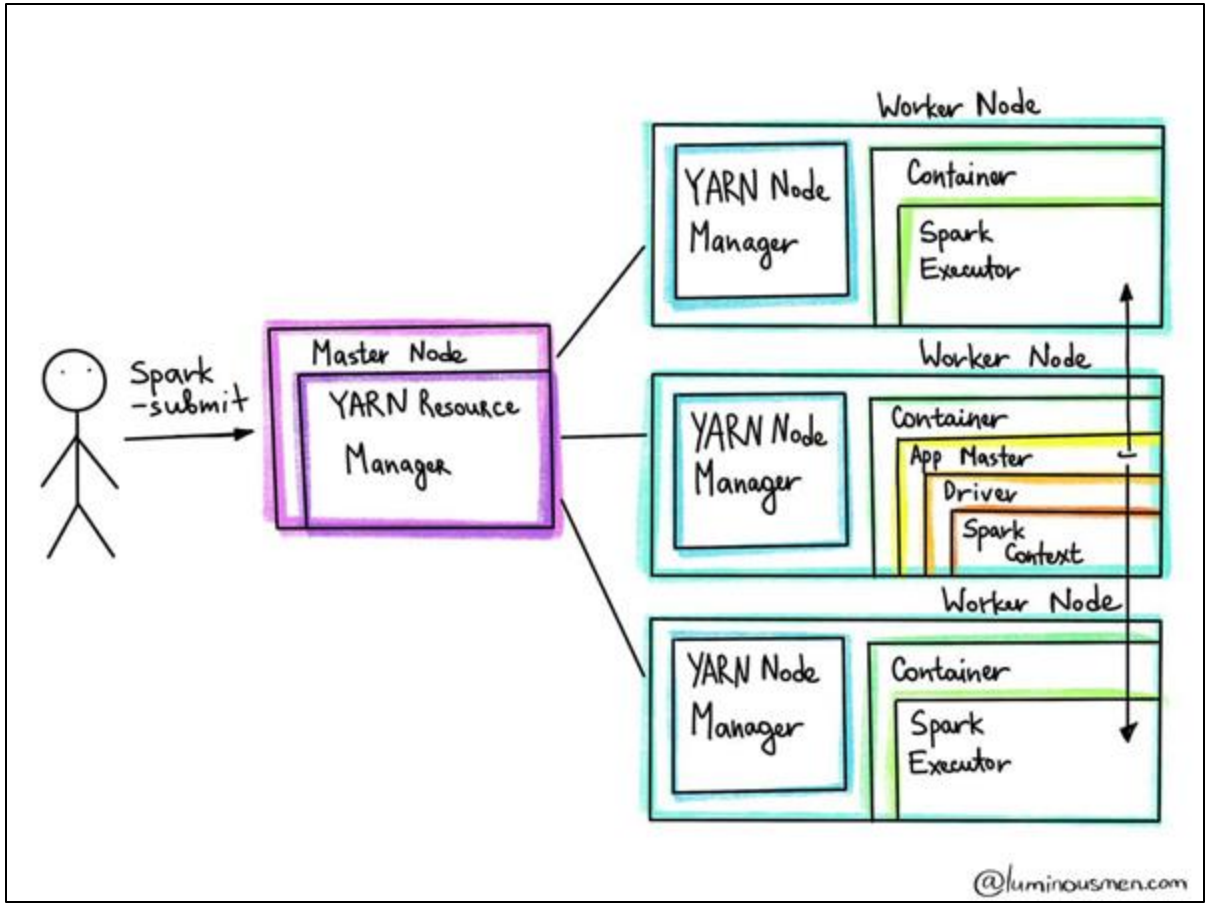
Kubernetes is the latest cluster manager for Spark

- Standalone: built-in, limited functionalities
- Apache Mesos: deprecated as of Spark 3.2.0
- Hadoop YARN: most widely used today
- **Kubernetes: most popular among new deployments**

The Spark on Kubernetes Journey



Spark on YARN: architecture & pain points



Global Spark version and shared libraries

- You'll have a Spark 2.4 cluster, a Spark 3.0 cluster, a Spark 3.1 cluster.
- Transient clusters are recommended for stability, but increase costs.

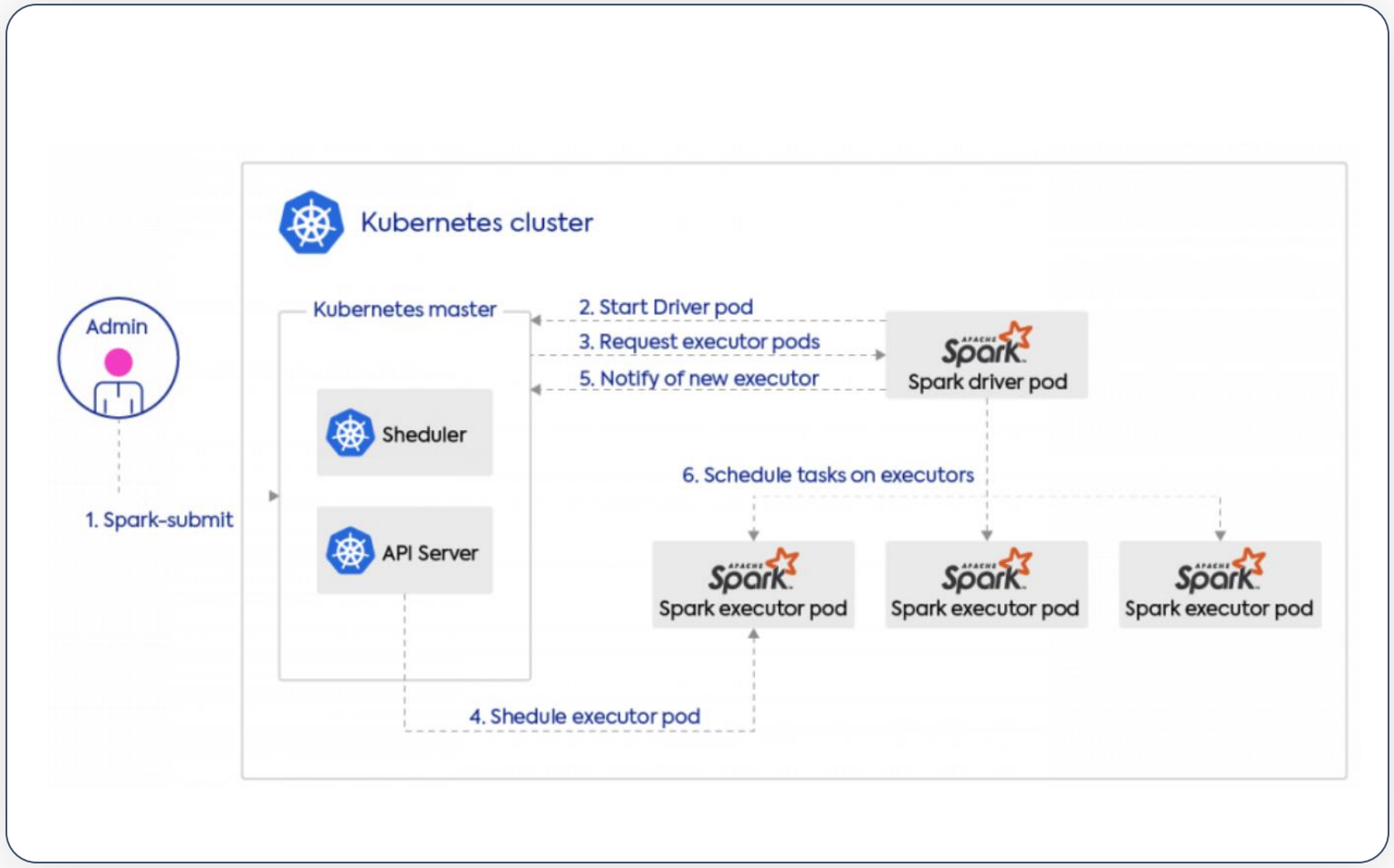
Limited Docker image support *

- Environment is built from AMIs and bash scripts, flaky runtime library installation
- Debugging is painful - there's no way to run Spark locally, environment is subtle

Resource Overhead

- Slow startup time
- YARN master node, YARN Node Mgr are JVM processes using a lot of resources.

Spark on Kubernetes: architecture & benefits



Native Dockerization

- Simpler dependency management
- Reliable executions across environments (locally during development, staging, production)
- Faster startup time

A single long-running cluster

- Quick to scale up (and down) based on load
- Mix different Spark versions
- Mix Spark and non-Spark apps
- Mix use cases (notebooks, batch/streaming jobs)

A standard, agnostic infrastructure layer

- Reduce lock in
- Simplify your operations
- Leverage the open-source tools from the cloud-native ecosystem

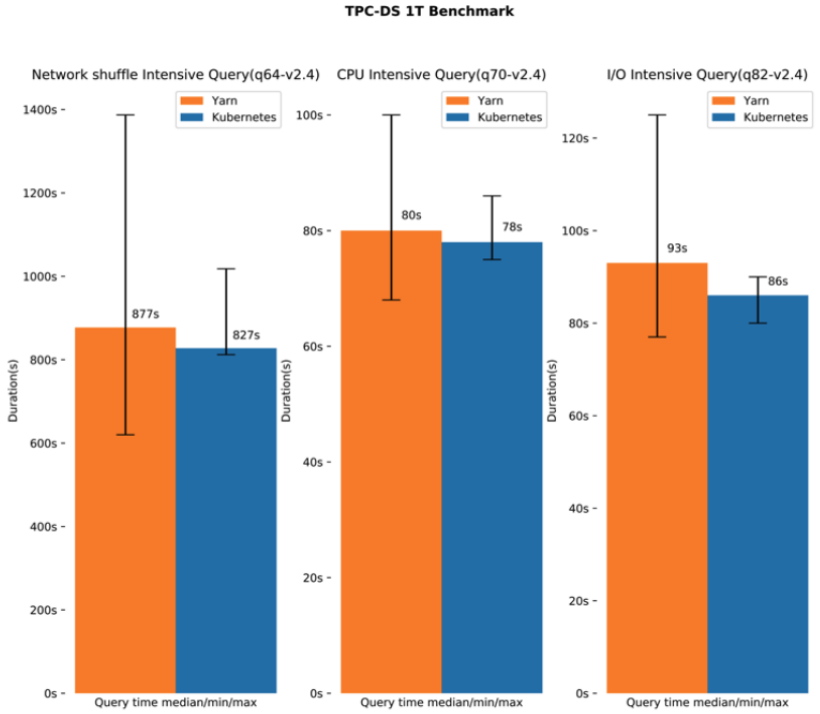
The Pros & Cons of Spark on Kubernetes

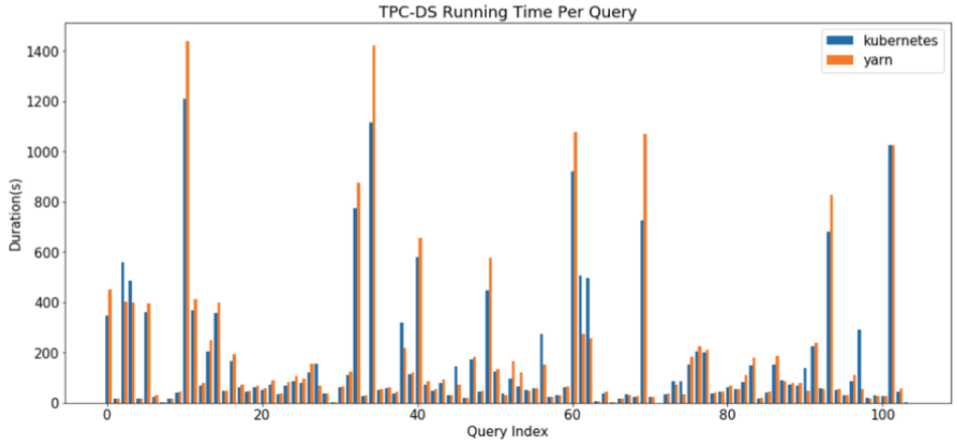
The pros 👍

- Better dev experience with Docker.
- An ecosystem of cloud-native tools.
- **Effective resource sharing enabling significant savings on cloud costs**.
- k8s can be the standard infrastructure layer across your entire stack: flexible, cloud- and vendor-agnostic

The cons 👎

- Data teams should not have to become Kubernetes experts.
- Kubernetes introduces powerful but complex abstractions, and requires the maintenance of many components.
- The support for Spark-on-Kubernetes on leading Spark products is absent or barebone.

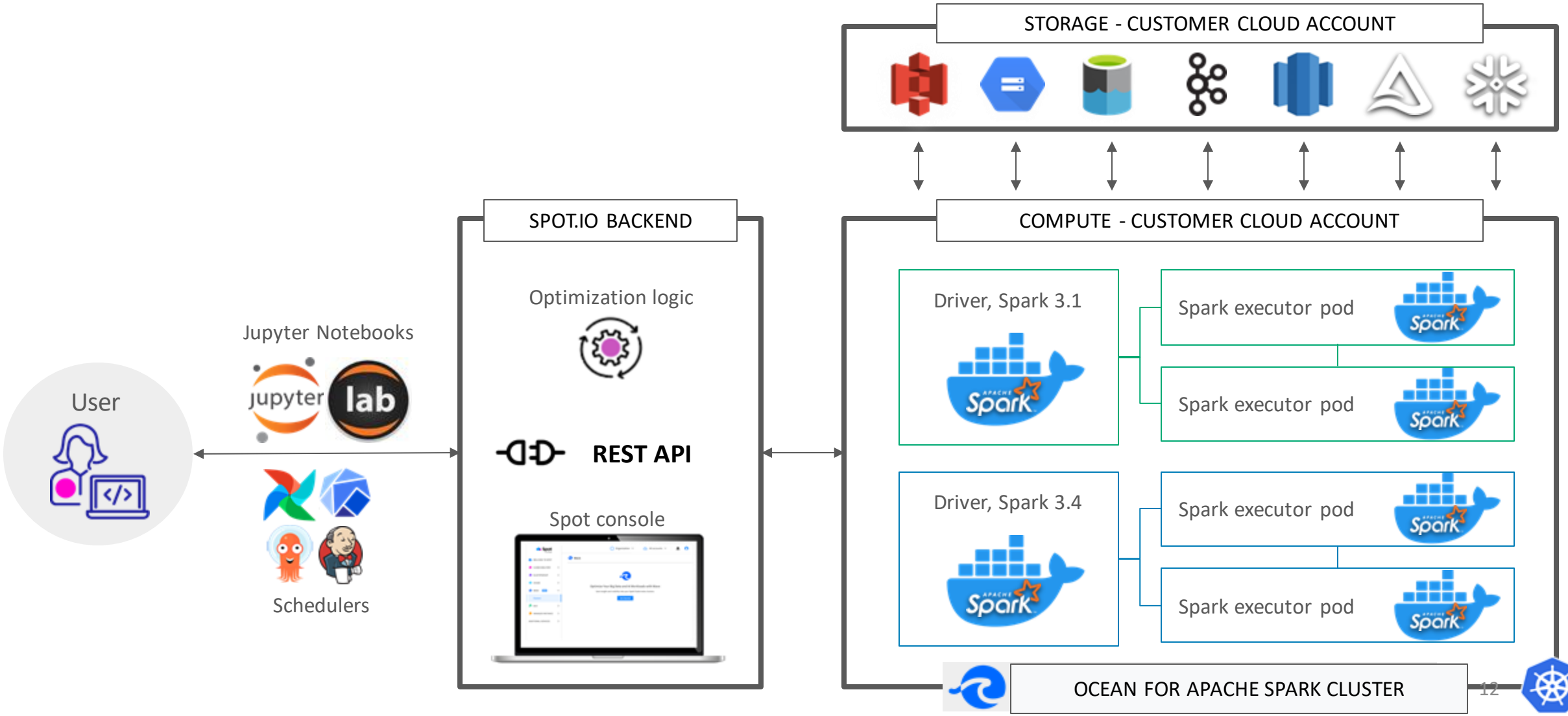




https://aws.amazon.com/fr/blogs/containers/optimizing-spark-performance-on-kubernetes/

Ocean for Apache Spark

Developer friendly, continuously scaled & optimized Spark on k8s



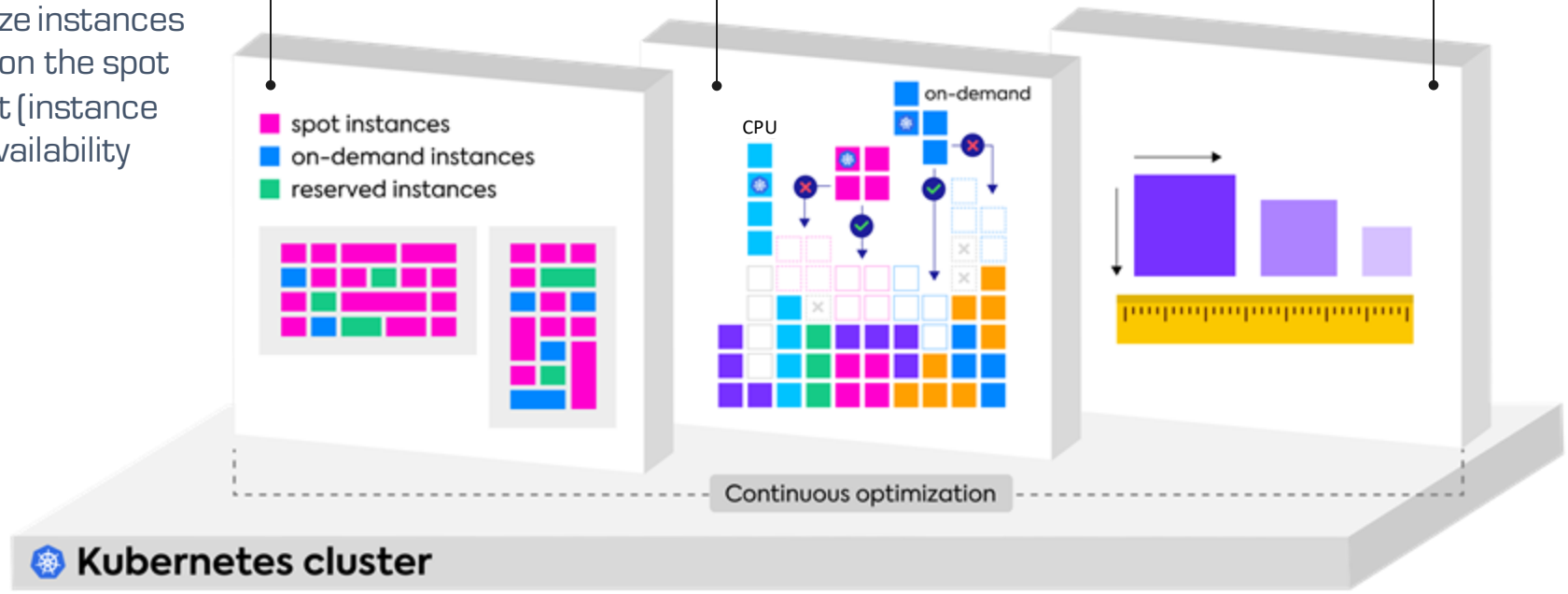
On a serverless & continuously optimized infrastructure

Infrastructure provisioning and pricing

- Autoscaling k8s cluster
- Optimize instances based on the spot market (instance type, availability zone)

Spark-aware autoscaling

- Each Spark application dynamically scales up and down based on the load.
- Gracefully handle spot kills & node shutdown.



History-based Optimizations of Spark Configs

Optimization based on the historical workload characteristics.

- container sizes
- # of executors
- # of partitions
- Shuffle tuning
- Spark feature flags

spot by NetApp

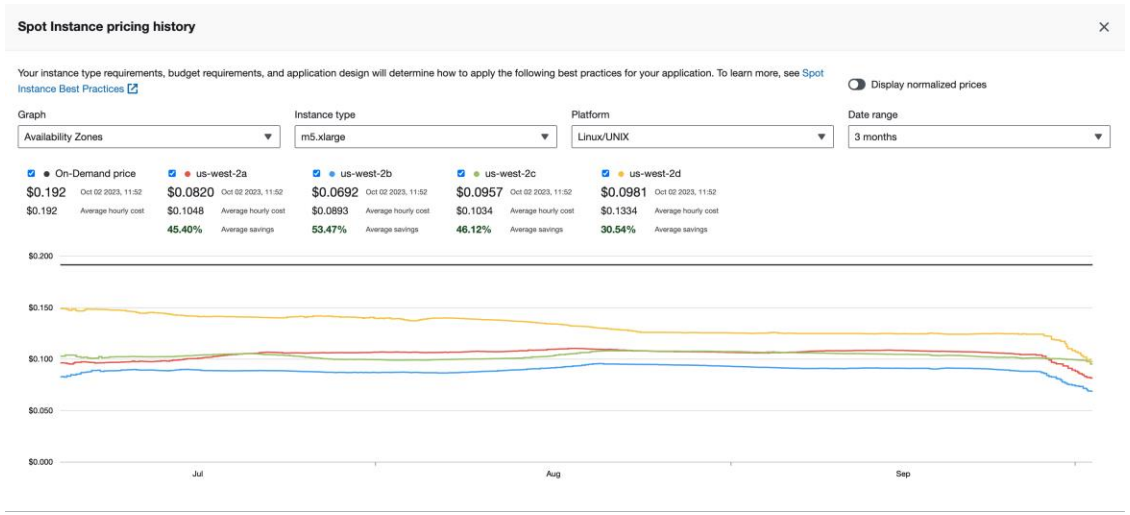
Harnessing the potential of spot instances for substantial cost savings

Spot instances

Up to 90% cheaper than their on-demand counterparts

- Available on AWS, GCP, and Azure
- Availability is not guaranteed
  - When you ask to launch a spot VM, the cloud provider can deny this request
  - Once a spot VM is launched, it can be reclaimed, at any time and at short notice
- Spot price varies in real-time
  - Based on supply & demand
  - Across 100s of independent spot markets:
    - Cloud region
    - Availability zone within the region
    - Instance type

Example of AWS spot instance price history
(one instance type, various AZs, 3 months)



How does Spark cope with spot interruptions?

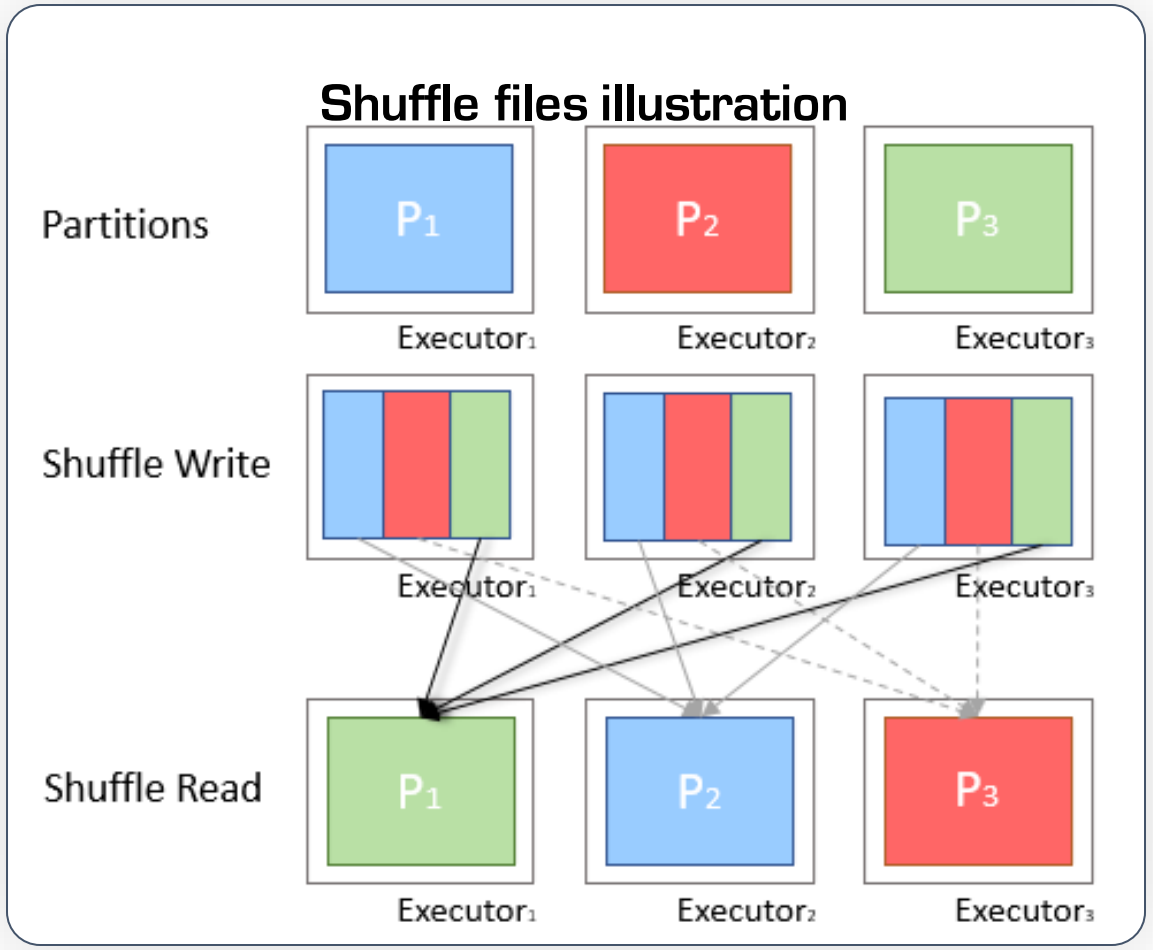
Best practice: Spark driver should run on an on-demand node

If you lose the Spark driver:

- The Spark app abruptly fails, and must be restarted from scratch.

If you lose a Spark executor, the app will have to recompute:

- The tasks which were in progress when the executor died
- Shuffle files: output of previous tasks stored on the executor
- Cached data



Best practice: run driver OD, execs on Spot

You can achieve this in k8s using node selectors

- Example on AWS (EKS) and using cluster-autoscaler
  - Define node labels and AutoScaling Group tags

Node label	ASG tag
lifecycle: spot	k8s.io/cluster-autoscaler/node-template/label/lifecycle: spot
lifecycle: ondemand	k8s.io/cluster-autoscaler/node-template/label/lifecycle: ondemand

- Add the relevant node selectors to your pods specs:

```
spec:
 driver:
   nodeSelector:
     lifecycle: ondemand
 executor:
   nodeSelector:
     lifecycle: spot
```

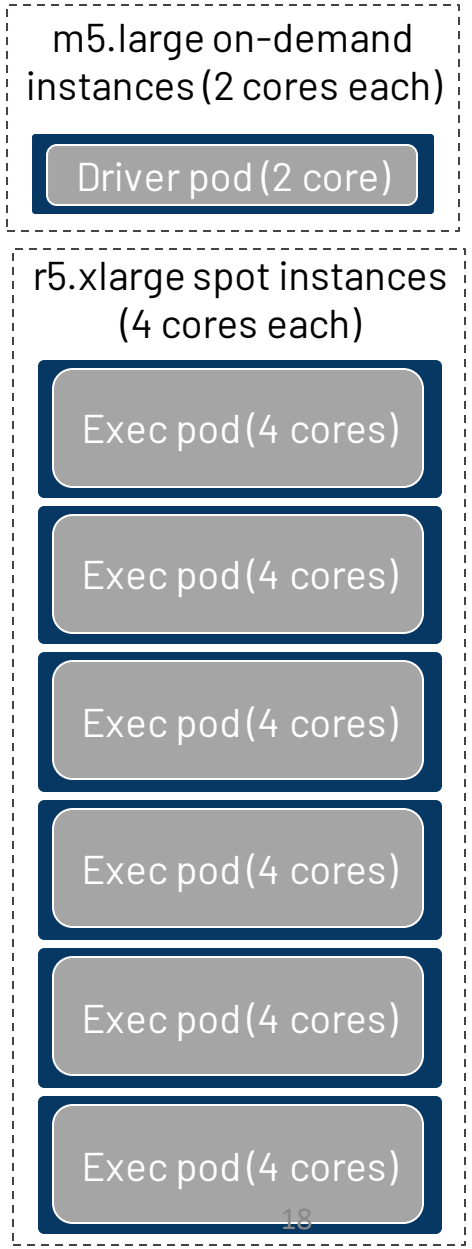
This is how your cluster may look like

But this isn't very stable yet

- If the r5.xlarge instance isn't available in the spot market, your executors will be stuck in pending state, and your app won't run (potentially for hours)
- You may lose all your Spark executors at once (which makes recovery harder).

App spec

```
{
 "driver": {
  "instanceSelector": "m5.large",
  "cores": 2,
  "spot": false
 },
 "executor": {
  "instanceSelector": "r5.xlarge",
  "cores": 4,
  "instances": 6,
  "spot": true
 }
}
```



Solution: pick the best possible spot market

Best availability zone, best instance type, fallback to On Demand instances

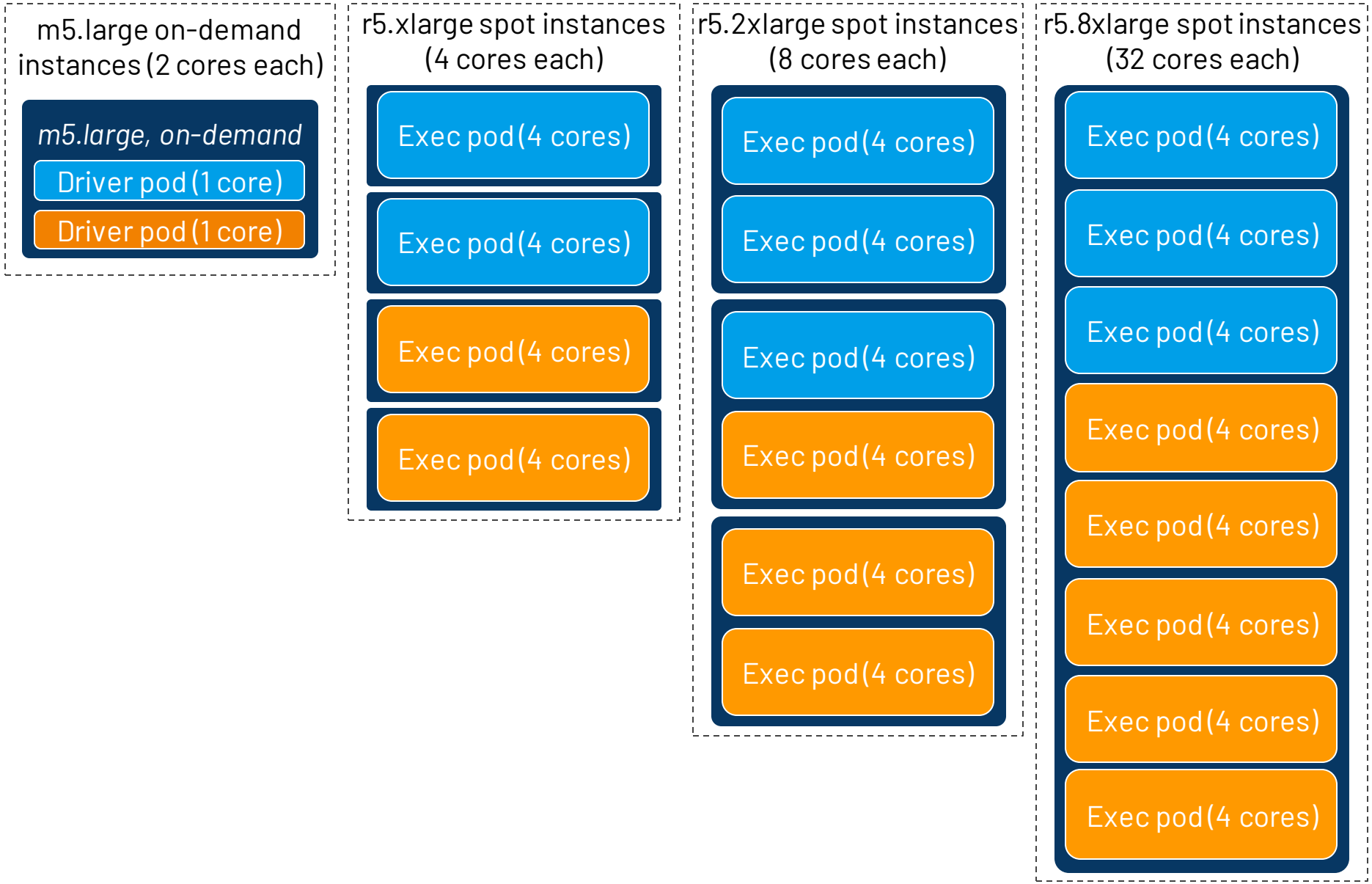
This is how your cluster may look like

Blue Application

```
{
 "driver": {
  "instanceSelector": "m5",
  "cores": 1,
  "spot": false
 },
 "executor": {
  "instanceSelector": "r5",
  "cores": 4,
  "spot": true,
  "instances": 8
 }
}
```

Orange Application

```
{
 "driver": {
  "instanceSelector": "m5",
 },
 "executor": {
  "instanceSelector": "r5",
  "instances": 10
 }
}
```



Limitation: Avoid cross-AZ data transfer

Co-localize all pods of a given Spark app on the same AZ

- The AZ selection should be done once upon Spark application submission, so that the driver and the executors pods all go to the same AZ.
- Otherwise, you will suffer from cross-AZ data transfer:
  - Which hurts shuffle performance significantly
  - And cloud providers charge a fee for this
- The additional flexibility granted by spreading executors across multiple AZs is not worth the penalty of cross-AZ transfer.

We ran an experiment to measure the impact

Under 2 different configurations

Same test Spark workload:

- 1 driver (1 core, on-demand), 10 executors (4 cores each, spot)
- Spark executor task consist of sleeping for 55 minutes
  - Such as the application run in about one hour, if no spot interruption occurs.

Run every hour for 2 weeks during business hours (9-5) under 2 settings:

- Static: Availability zone hardcoded, instance type hardcoded (m5.xlarge)
- Optimized: AZ flexible, instance type flexible within m5 family

Experiment results

Spot market optimization avoids **79%** of spot interruptions

	# of Spark apps that ran	# of execs launched	Avg # of spot kills per application	Avg app duration
Static configuration (m5.xlarge)	139*	1817	**3.07**	1 hour 20 min **(+20 min vs ideal)**
Optimized configuration (m5 family)	147	1567	**0.65**	1 hour 5 min **(+5 min vs ideal)**

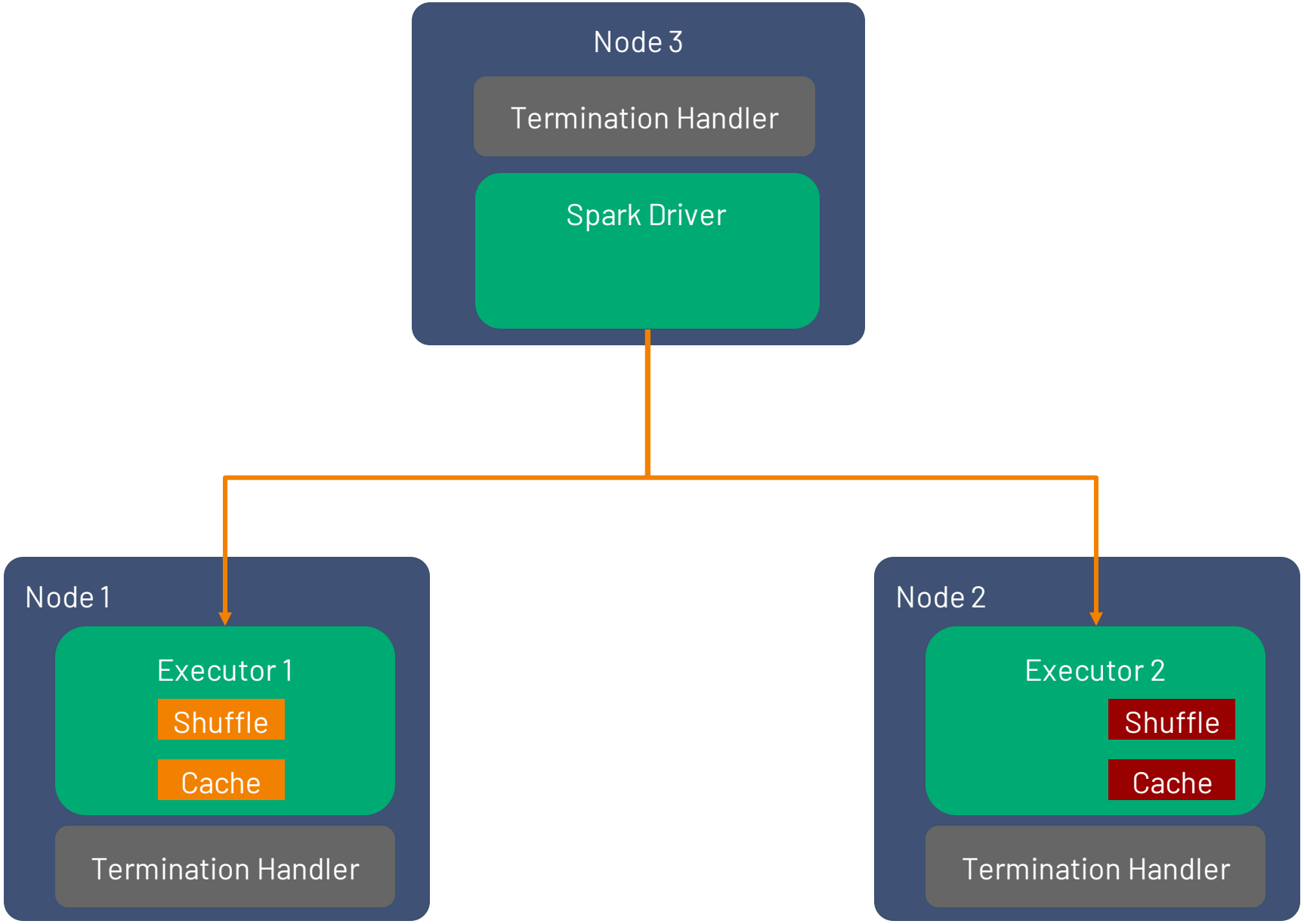
**Sometimes, the applications with a hardcoded configuration to run on m5.xlarge spot instances did not run at all, due to a lack of spot nodes availability.*

How to handle spot instances interruptions

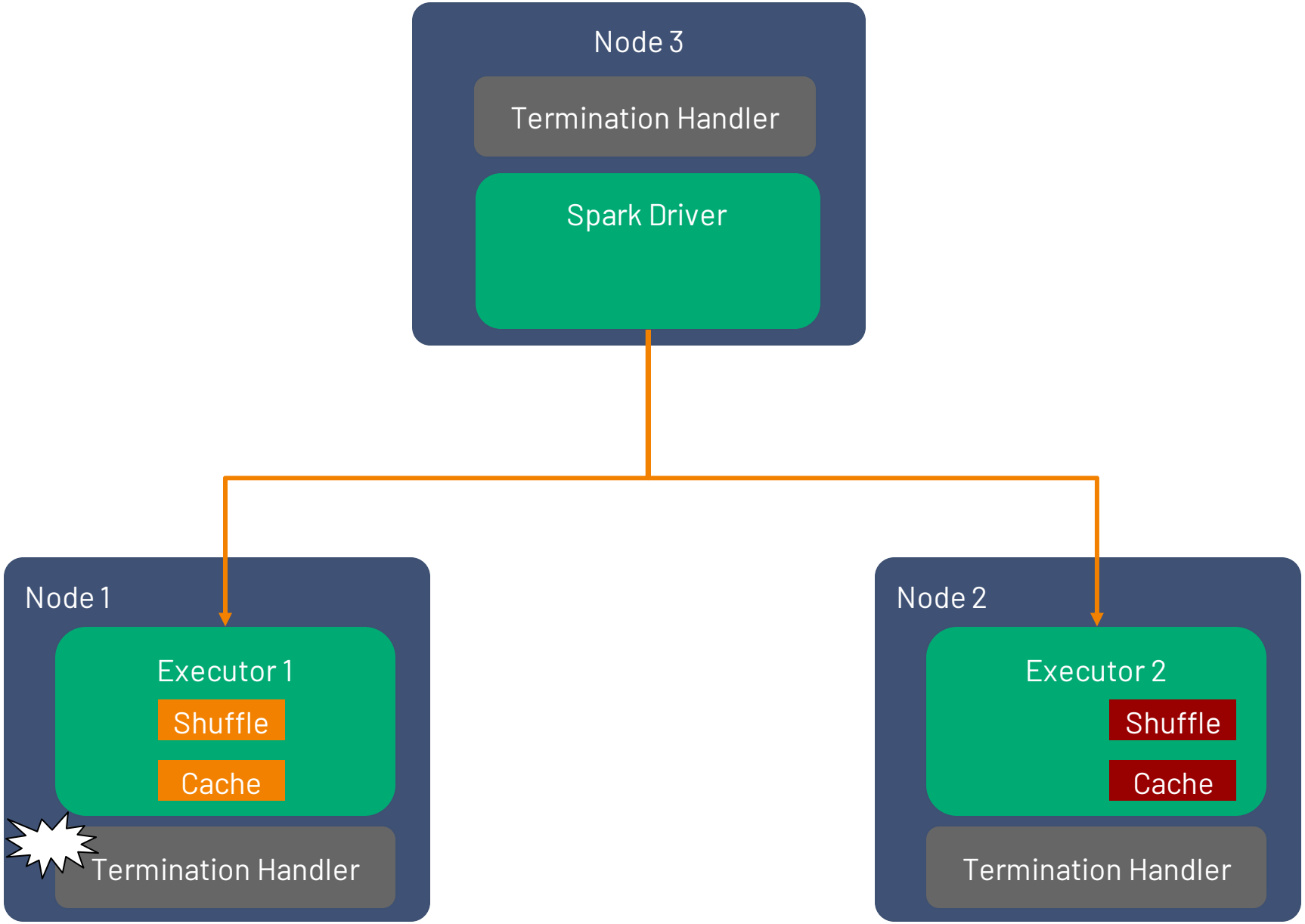
Since Spark 3.2: Graceful Exec Decommissioning

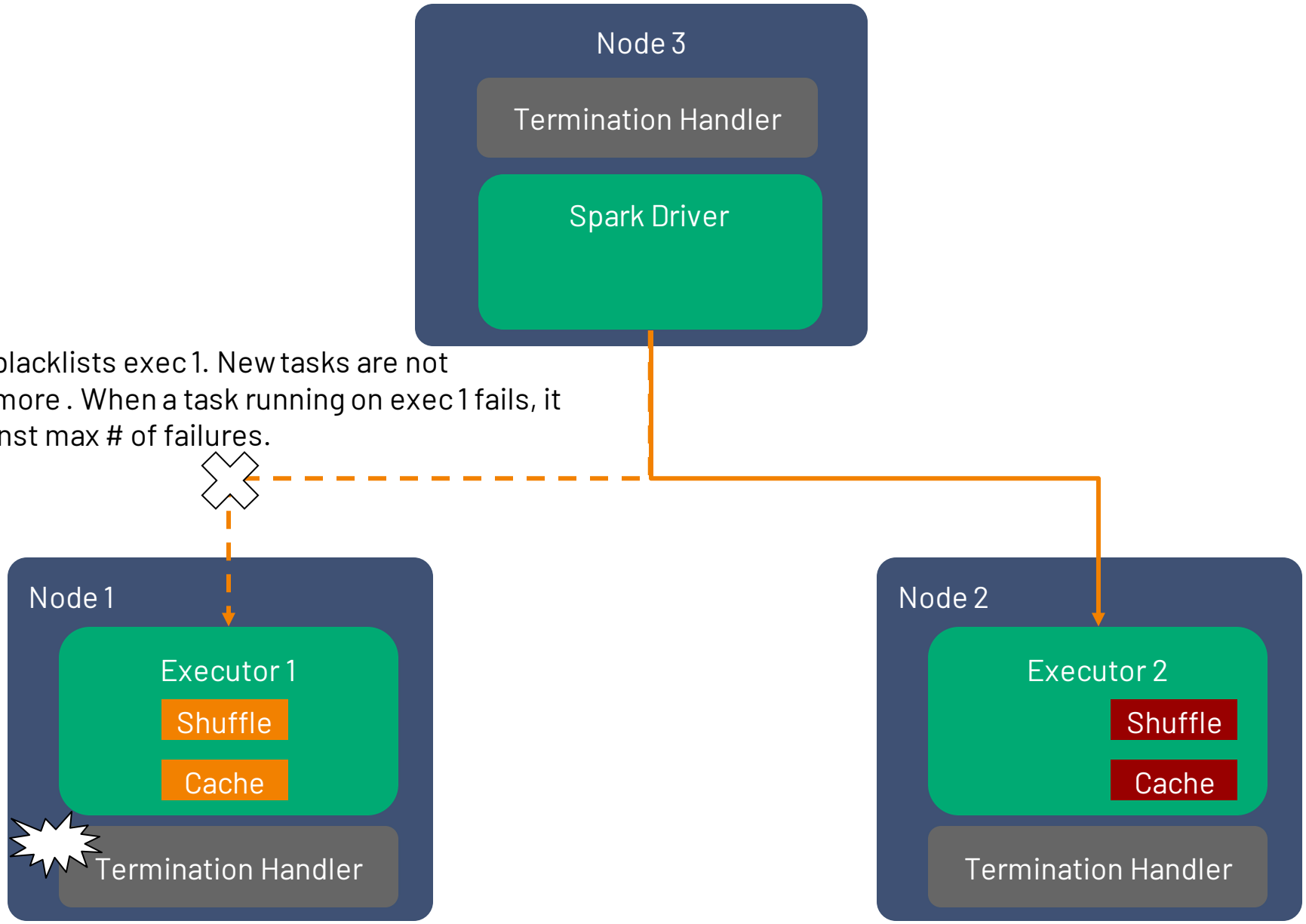
- Before interrupting a spot instance, cloud providers give a notice:
  - Termination notice: 2 min on AWS, 30s on GCP, 30s on Azure
  - This signal can be intercepted by a NodeTerminationHandler (k8s DaemonSet)
  - The daemonset then sends a message to the executor, which sends a message to the driver

- The driver then does the following:
  - Stop scheduling task on the executor which is going to go away
  - Do not count task failure on this executor against the maximum number of retriable failures
  - Move the shuffle files and cached data from this executor to another executor
  - Update the state of shuffle files location accordingly


Node 3
Termination Handler
Spark Driver
Node 1
Executor 1
Shuffle
Cache
Termination Handler
Node 2
Executor 2
Shuffle
Cache
Termination Handler

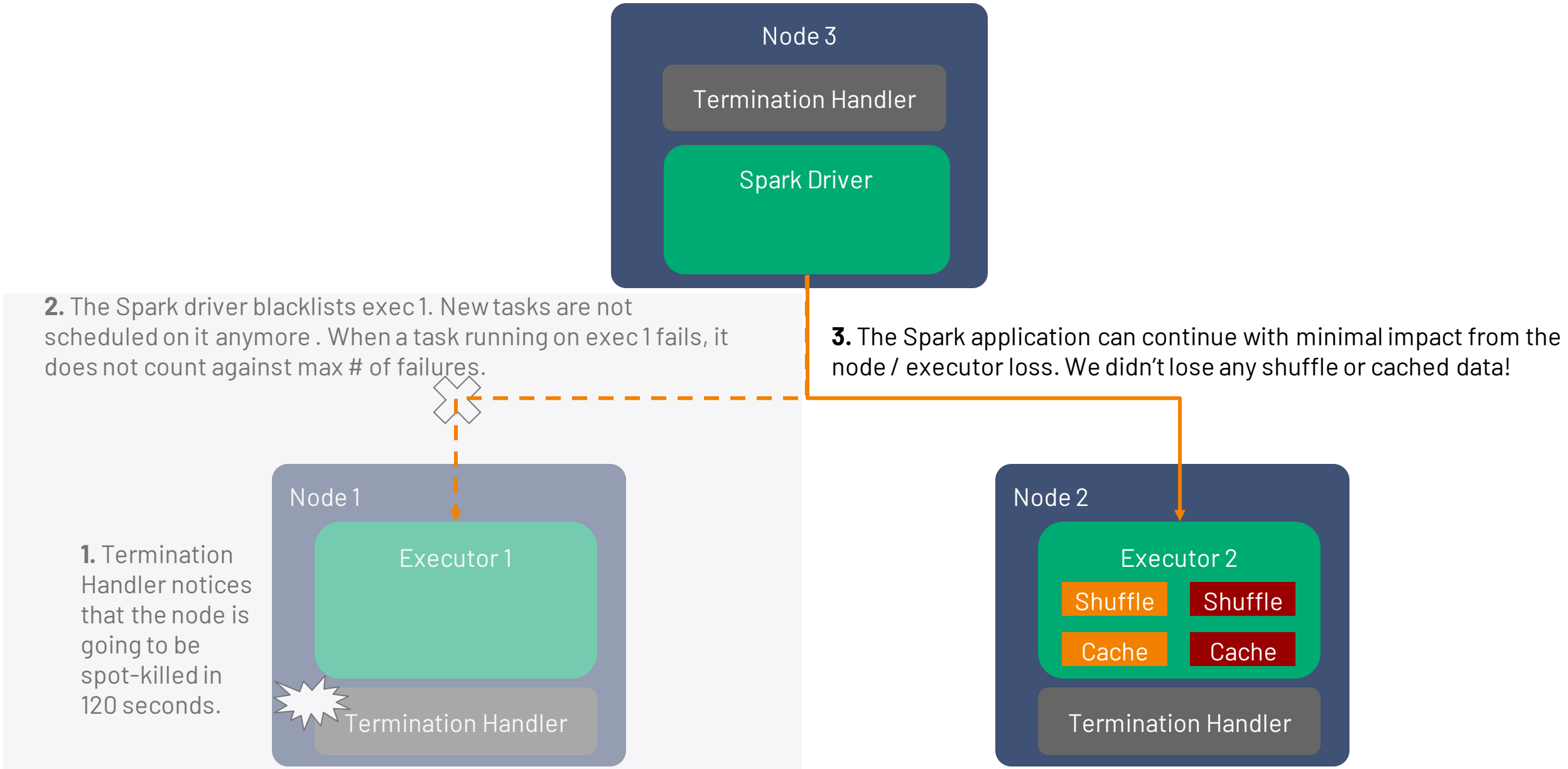
1. Termination Handler notices that the node is going to be spot-killed in 120 seconds.





2. The Spark driver blacklists exec 1. New tasks are not scheduled on it anymore . When a task running on exec 1 fails, it does not count against max # of failures.

1. Termination Handler notices that the node is going to be spot-killed in 120 seconds.


Node 3
Termination Handler
Spark Driver
2. The Spark driver blacklists exec 1. New tasks are not scheduled on it anymore . When a task running on exec 1 fails, it does not count against max # of failures.
3. The Spark application can continue with minimal impact from the node / executor loss. We didn't lose any shuffle or cached data!
1. Termination Handler notices that the node is going to be spot-killed in 120 seconds.
Node 1
Executor 1
Termination Handler
Node 2
Executor 2
Shuffle
Shuffle
Cache
Cache
Termination Handler

Graceful Exec Decommissioning

How to enable this feature:

- Install NodeTerminationHandler for your cloud provider as a k8s daemonset
- Turn on the following configuration flags:

  spark.decommission.enabled
  spark.storage.decommission.rddBlocks.enabled
  spark.storage.decommission.shuffleBlocks.enabled
  spark.storage.decommission.enabled

Limitations:

- Very large shuffle files may not have enough time to be migrated
- If many executors get spot-killed at the same time, we may lose some shuffle files
  - For example the driver learns that exec-1 is decommissioned, decides to move files to exec-2, and then the driver learns that exec-2 is decommissioned too
  - This gives another argument in favor of spreading executors across multiple spot markets ("do not put all your eggs in one basket")

Graceful Exec Decommissioning - Experiment

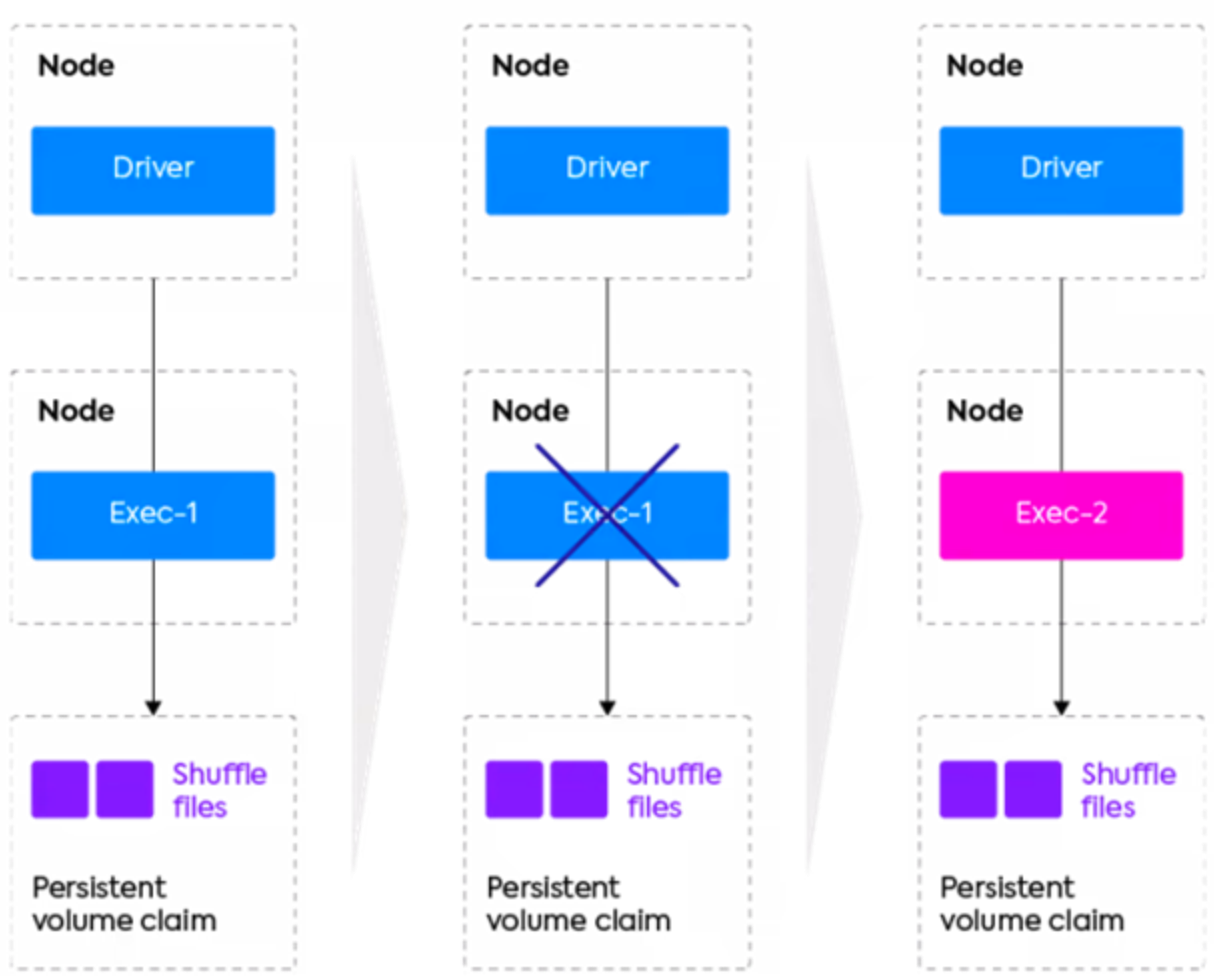
- We ran workloads that produced a significant amount of shuffle files on the executors.
- We simulated the process of receiving a spot kill
- Using the Storage tab in the Spark UI and parsing information from the driver and executor logs, we could measure
  - The data stored on the executors prior to detachment
  - The data moved from one executor to another during decommissioning
  - The time Spark spent moving files
- We tested with 4-core executors across different instance types (m5, m5d, i3, ...)

On average, Graceful Executor Decommissioning moved ~15GB/minute of shuffle data on regular instances, and 35-40GB of data/minute on SSD-backed instances.

Since Spark 3.2: Executor PVC Reuse

- Since Spark 3.1, it's possible to configure Spark to dynamically provision and mount Persistent Volume Claims (PVCs).
  - But the PVC and executor share the same fate.

- As of Spark 3.2, PVCs mounted onto executors can survive the removal of its original executor, to be mounted on a new executor instead.
  - **This means the shuffle files can be recovered after a spot kill, or even another failure (such as an OutOfMemory error).**



Executor PVC Reuse

How to enable this feature:

- Configure dynamic PVCs (see open-source documentation, there are many possibilities)
- Turn on the following configuration flags:
  *Spark.kubernetes.driver.reusePersistentVolumeClaim**
  Spark.kubernetes.driver.ownPersistentVolumeClaim

Limitations:

- This feature is not compatible with using local NVMe based SSDs for shuffle files (PVCs are typically backed by remote volumes such as EBS)
  - Local NVMe based SSDs offer 5-10x performance improvement for shuffle-heavy workloads
- * In our tests, a race condition sometimes causes a PVC not to be re-used immediately, so the shuffle file recovery does not work every time:
  - *spark.kubernetes.driver.waitToReusePersistentVolumeClaim=true (since Spark 3.4)*
- This feature requires a bit more configuration than the graceful decommissioning feature.

Conclusion: Future works and best practices for Spark on k8s

How to make Spark run reliably on spot VMs

Substantial cost savings without trading off performance or stability

- Driver should run on demand, executors on spot
- Optimize the spot market to avoid spot kills
  - Pick the best AZ (and use it for the entire app)
  - Spread executors across multiple spot instance types, based on real time spot market dynamics
- Gracefully handle spot kills by proactively moving shuffle files when a spot termination occurs

What's new in Spark 3.4 for Spark-on-k8s

- [SPARK-41515] PVC-oriented executor pod allocation

Spark / SPARK-41515

PVC-oriented executor pod allocation

Details

Type:	New Feature	Status:	RESOLVED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	3.4.0	Fix Version/s:	3.4.0
Component/s:	Kubernetes		
Labels:	releasenotes		

Issue Links

is related to

- ~~SPARK-44993~~ Add ShuffleChecksumUtils.compareChecksums by reusing ShuffleChecksumTestHelp.compareChecksums — RESOLVED
- ~~SPARK-41550~~ Dynamic Allocation on K8S GA — RESOLVED

relates to

- ~~SPARK-44915~~ Validate checksum of remounted PVC's shuffle data before recovery — RESOLVED

Sub-Tasks

1.	Support PVC-oriented executor pod allocation	RESOLVED	Dongjoon Hyun
2.	getReusablePVCs should ignore recently created PVCs in the previous batch	RESOLVED	Dongjoon Hyun
3.	getReusablePVCs should handle accounts with no PVC permission	RESOLVED	Dongjoon Hyun
4.	recoverDiskStore should not stop by existing recomputed files	RESOLVED	Dongjoon Hyun
5.	Upgrade kubernetes-client to 5.12.3	RESOLVED	Dongjoon Hyun
6.	Add `PVC-oriented executor pod allocation` section and revise config name	RESOLVED	Dongjoon Hyun
7.	Improve onNewSnapshots to use unique list of known executor IDs and PVC names	RESOLVED	Dongjoon Hyun
8.	Skip PVC cleanup when driver doesn't own PVCs	RESOLVED	Pralabh Kumar
9.	Show a directional error message for PVC Dynamic Allocation Failure	RESOLVED	Qian Sun

People

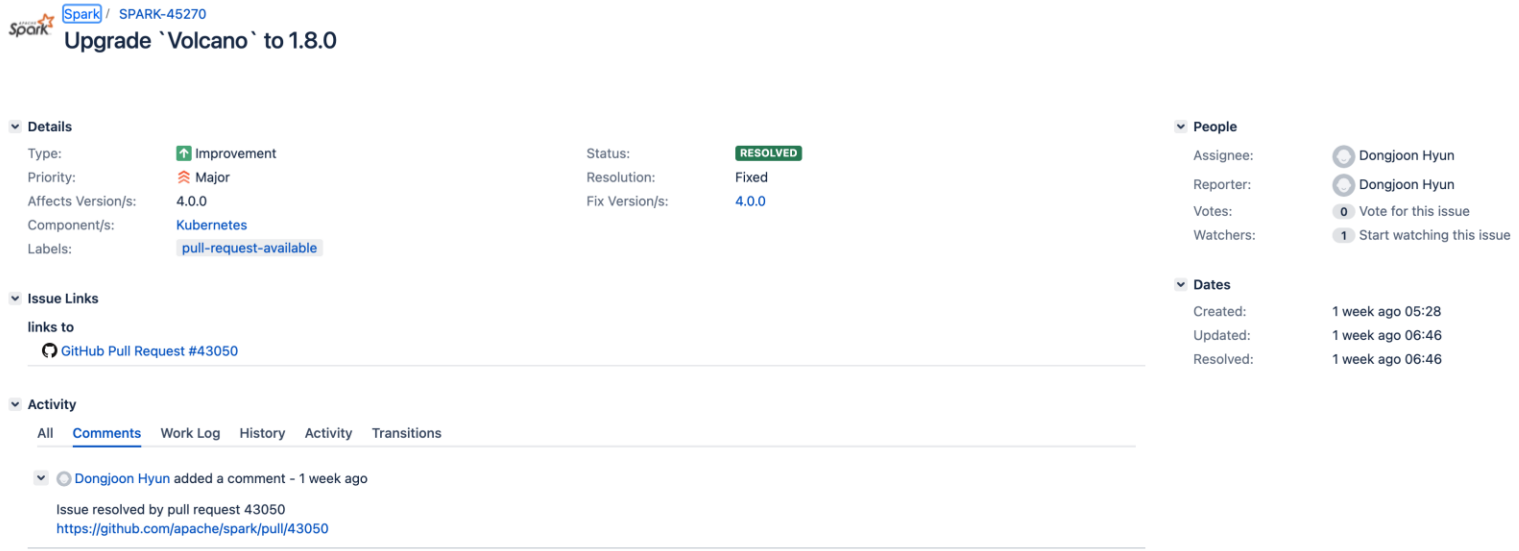
Assignee:	Dongjoon Hyun
Reporter:	Dongjoon Hyun
Votes:	0 Vote for this issue
Watchers:	1 Start watching this issue

Dates

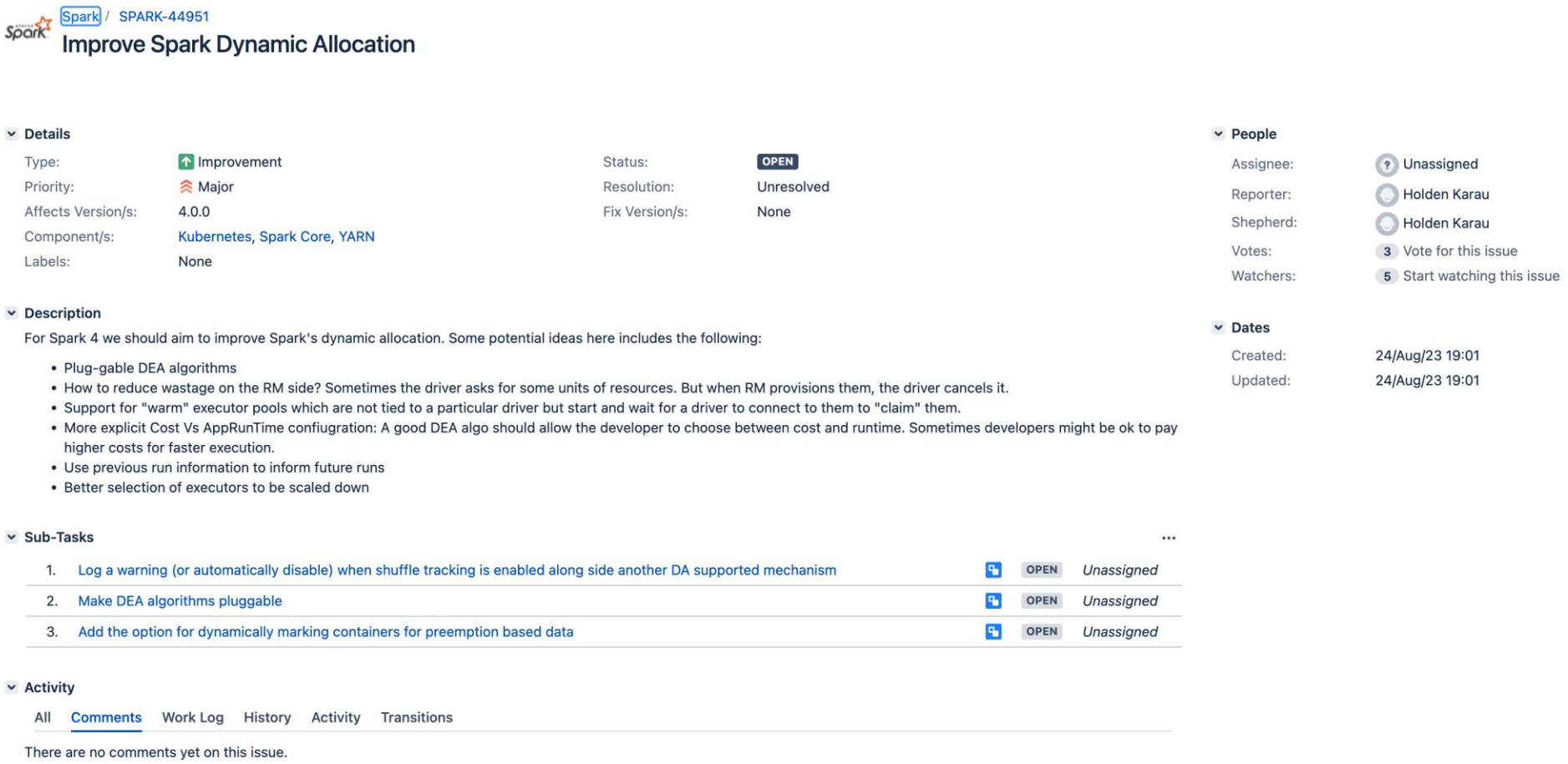
Created:	14/Dec/22 05:43
Updated:	13/Sep/23 08:13
Resolved:	16/Dec/22 17:31

What's new in Spark 3.4 and above for Spark-on-k8s

- [SPARK-45270] Custom k8s Scheduler support (Volcano)
  - Enable YARN-like capabilities such as queue, gang scheduling, etc



What's new in Spark 3.4 and above for Spark-on-k8s



Spark / SPARK-44951

Improve Spark Dynamic Allocation

Details

Type:	Improvement	Status:	OPEN
Priority:	Major	Resolution:	Unresolved
Affects Version/s:	4.0.0	Fix Version/s:	None
Component/s:	Kubernetes, Spark Core, YARN		
Labels:	None		

People

Assignee:	Unassigned
Reporter:	Holden Karau
Shepherd:	Holden Karau
Votes:	3 Vote for this issue
Watchers:	5 Start watching this issue

Dates

Created:	24/Aug/23 19:01
Updated:	24/Aug/23 19:01

Description

For Spark 4 we should aim to improve Spark's dynamic allocation. Some potential ideas here includes the following:

- Plug-gable DEA algorithms
- How to reduce wastage on the RM side? Sometimes the driver asks for some units of resources. But when RM provisions them, the driver cancels it.
- Support for "warm" executor pools which are not tied to a particular driver but start and wait for a driver to connect to them to "claim" them.
- More explicit Cost Vs AppRunTime confiugration: A good DEA algo should allow the developer to choose between cost and runtime. Sometimes developers might be ok to pay higher costs for faster execution.
- Use previous run information to inform future runs
- Better selection of executors to be scaled down

Sub-Tasks

1.	Log a warning (or automatically disable) when shuffle tracking is enabled along side another DA supported mechanism	OPEN	*Unassigned*
2.	Make DEA algorithms pluggable	OPEN	*Unassigned*
3.	Add the option for dynamically marking containers for preemption based data	OPEN	*Unassigned*

Activity

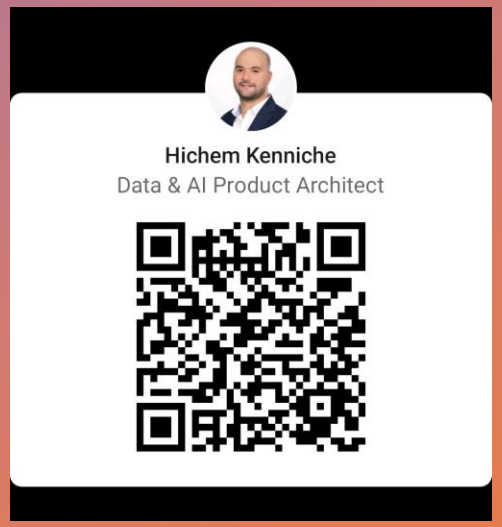
All Comments Work Log History Activity Transitions

There are no comments yet on this issue.

instaclustr
Now part of Spot by NetApp

COMMUNITY
THE ASF CONFERENCE
CODE

Thank you



Ocean for Apache Spark