DATA+AI SUMMIT 2022

How to make Spark on Kubernetes Run Reliably on Spot Instances

Jean-Yves Stephan

Senior Product Manager, Spot by NetApp

Hudson Buzby

Solutions Architect, Spot by NetApp

ORGANIZED BY databricks

/whoami

A few words about us!

Jean-Yves Stephan
Senior Product Manager, Spot by NetApp

Previously:

- 2015–2018: Databricks
  - Software engineer
  - Technical lead for Spark infrastructure
- 2018–2021: Data Mechanics
  - Co-Founder & CEO
  - Acquired by Spot by NetApp in June 2021

Hudson Buzby
Spark Solutions Architect, Spot by NetApp

Previously:

- 2017–2021: Data engineer working with Apache Spark and other data tech
  - Namely
  - Vice Media
  - Eataly

Our agenda for today

How to make Spark on k8s run reliably on spot instances

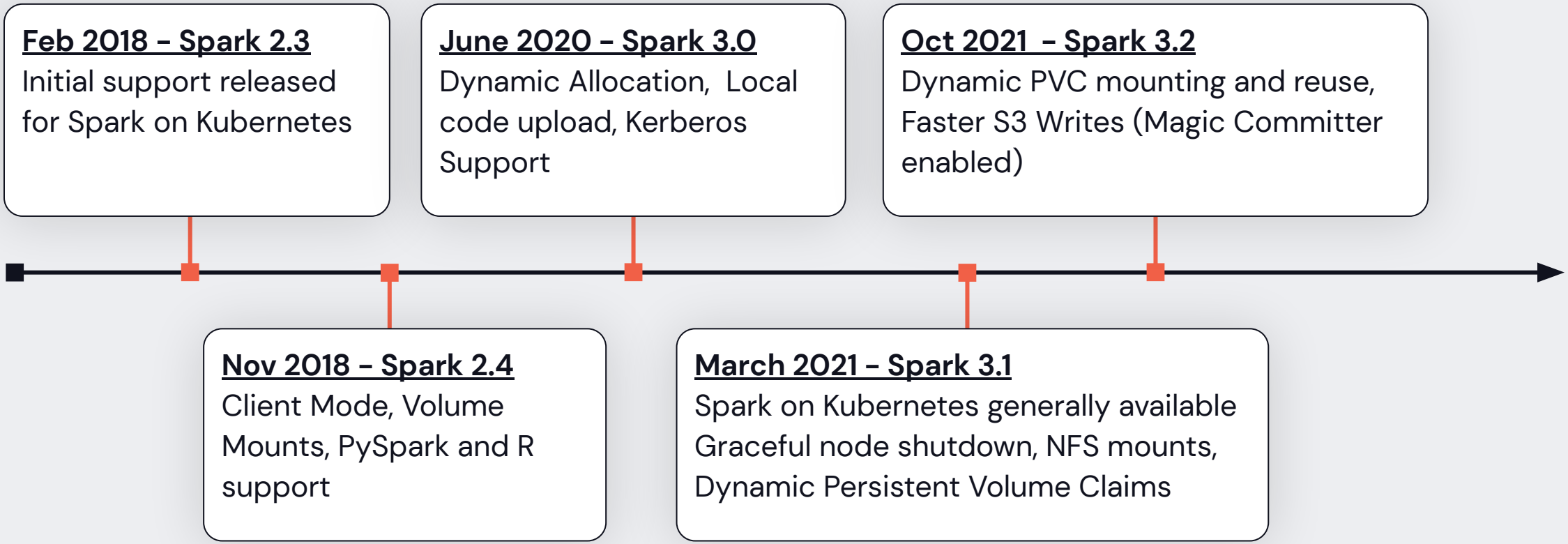
- An introduction to Spark on Kubernetes
  - Architecture, benefits, our background with it
- How to avoid spot instances interruptions as much as possible
  - Running driver on on-demand nodes, picking best spot markets (AZ, instance type)
- How to handle spot instances interruptions as gracefully as possible
  - Spark 3.1 executor decommissioning, Spark 3.2 PVC reuse when executor is lost
- Conclusion - Future works and best practices with Spark on Kubernetes

An introduction to Spark on Kubernetes

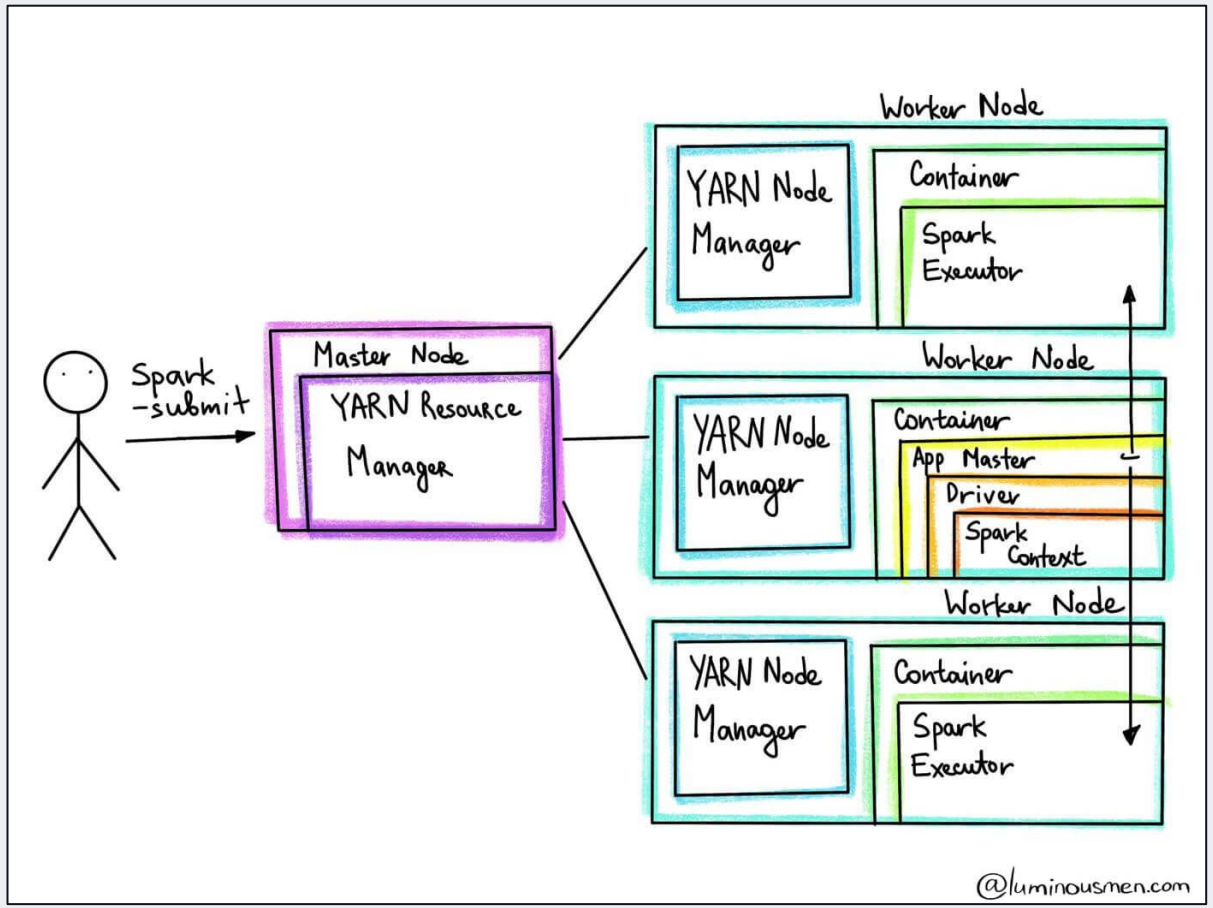
Kubernetes is a new cluster manager for Spark

- 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.

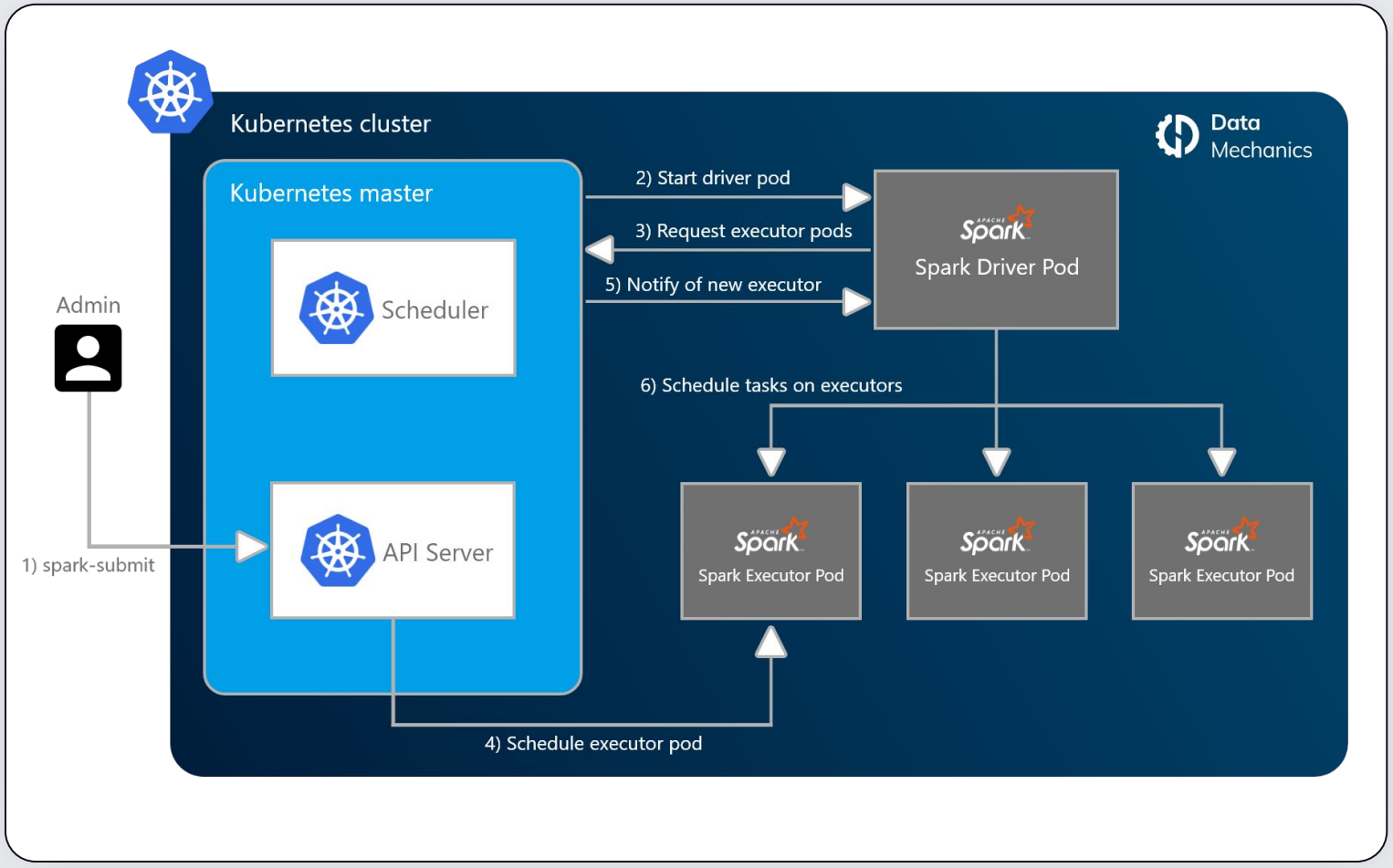
No native Docker image support

- 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 👍

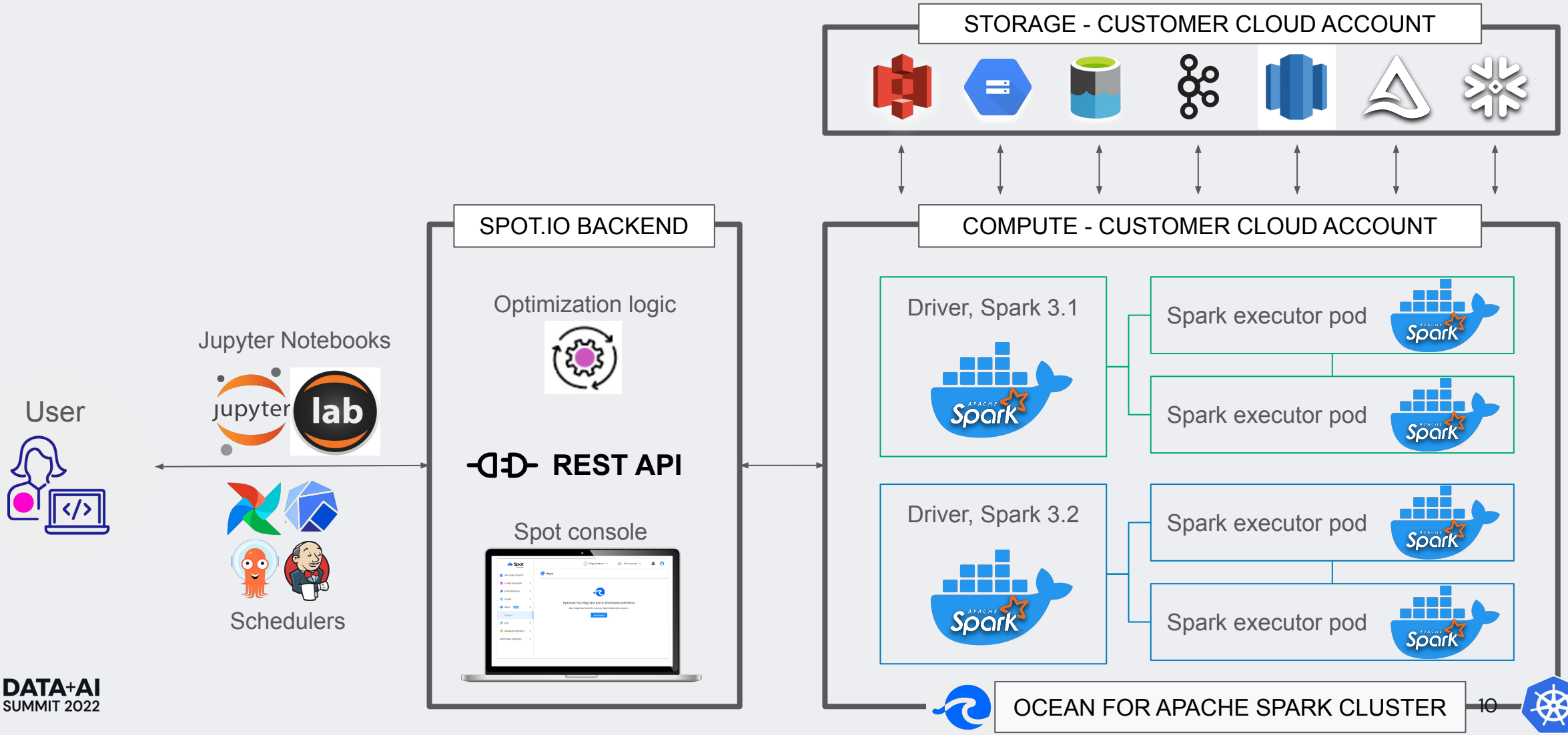
- Better dev experience with Docker.
- An ecosystem of cloud-native tools.
- Effective resource sharing enabling significant savings on cloud costs.
- k8s can be the standard infrastructure layer across your entire stack: flexible, cloud- and vendor-agnostic

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.

Our background – Ocean for Apache Spark

Developer friendly, continuously scaled & optimized Spark on k8s



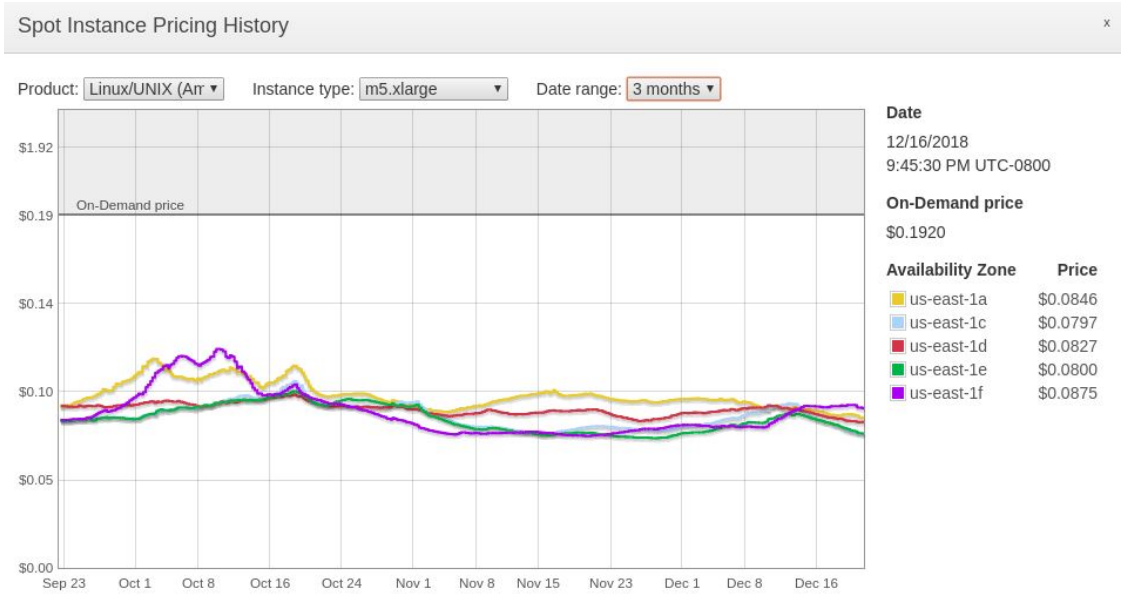
How to avoid spot instance interruptions

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



How does Spark cope with spot interruptions?

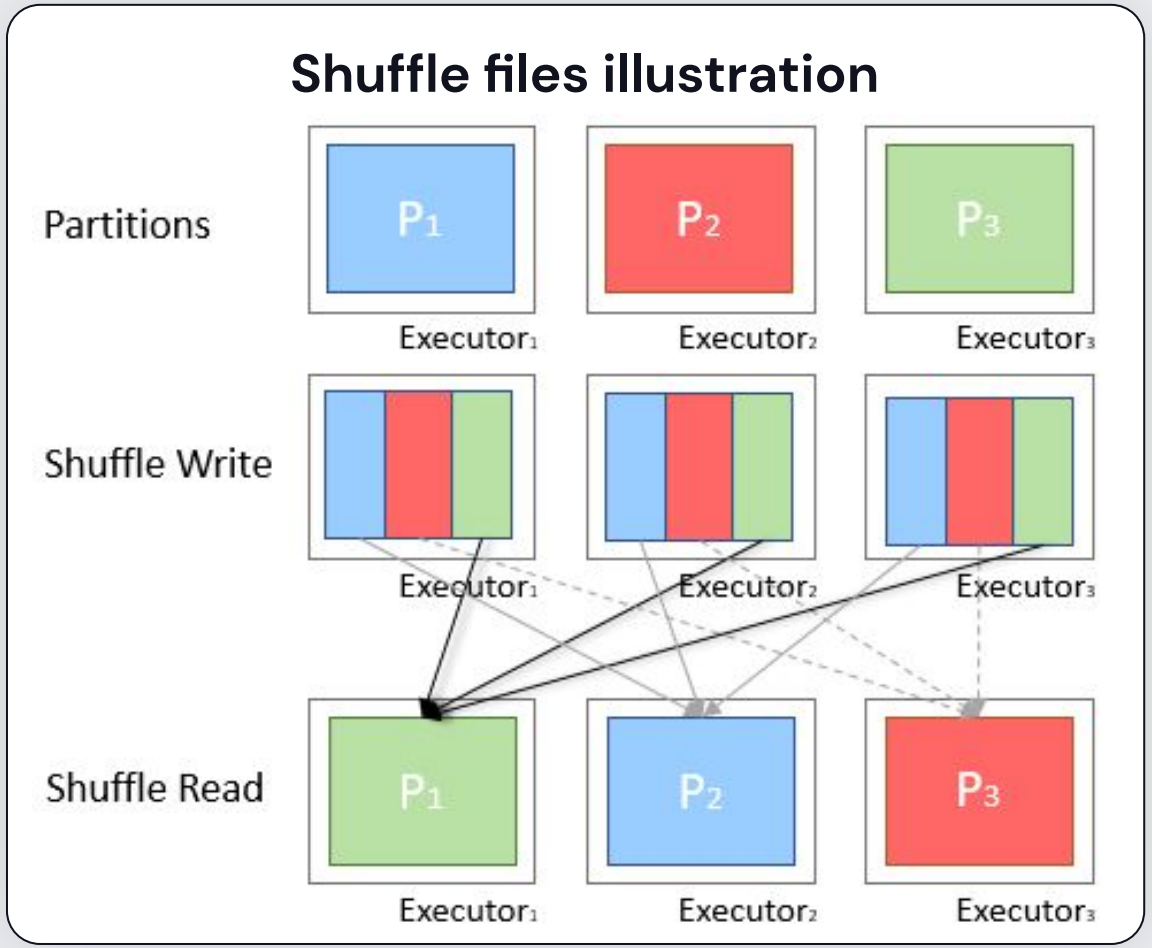
Best practice: Spark driver should run on an on-demand node

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
```

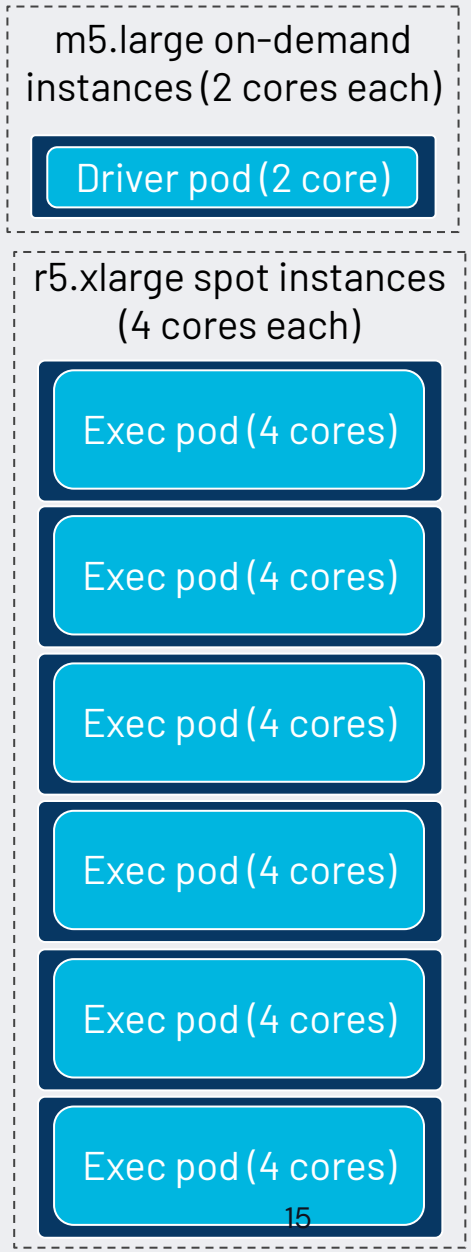
This is how your cluster may look like

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 OD

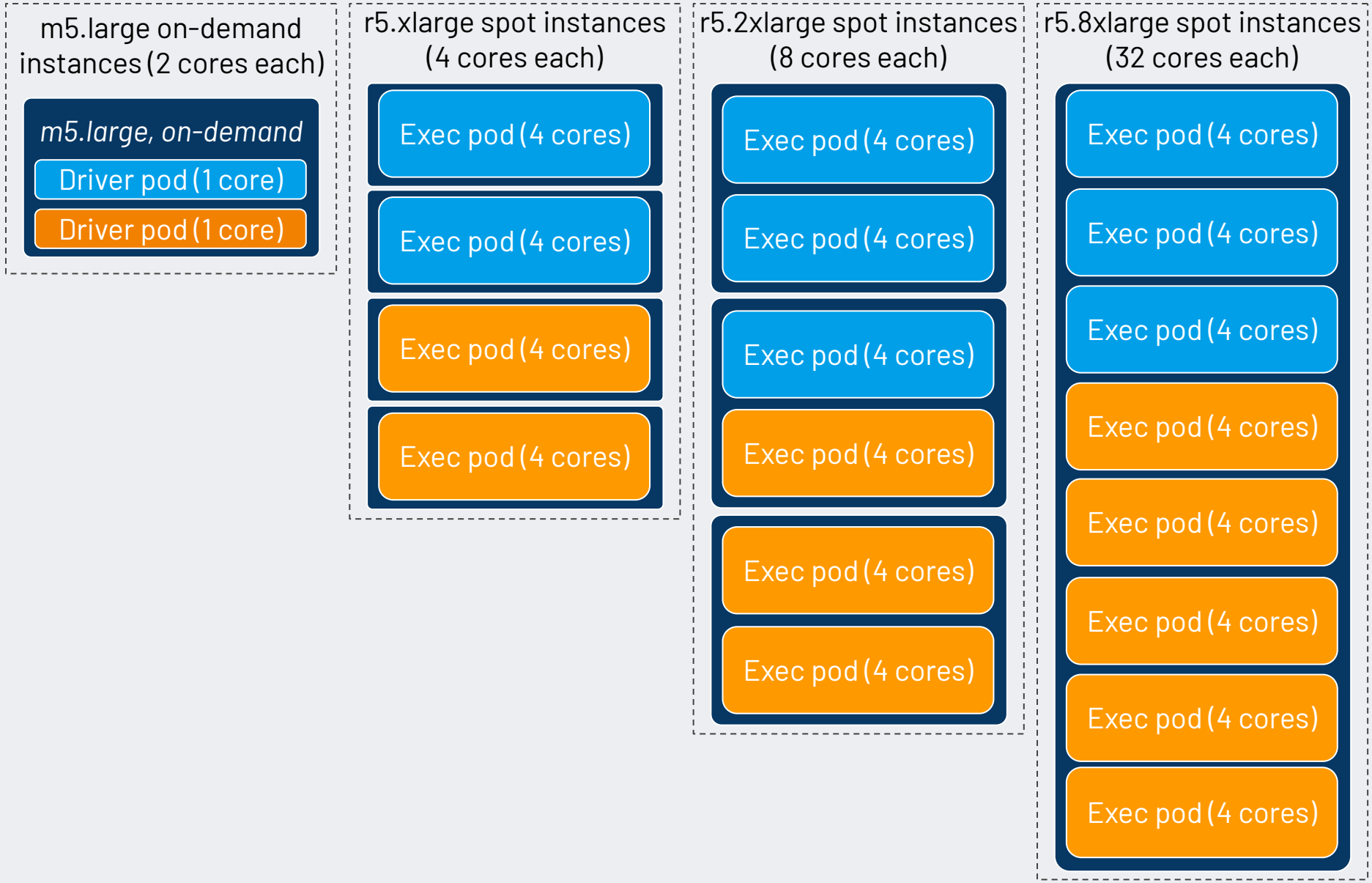
This is how your cluster may look like

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.*

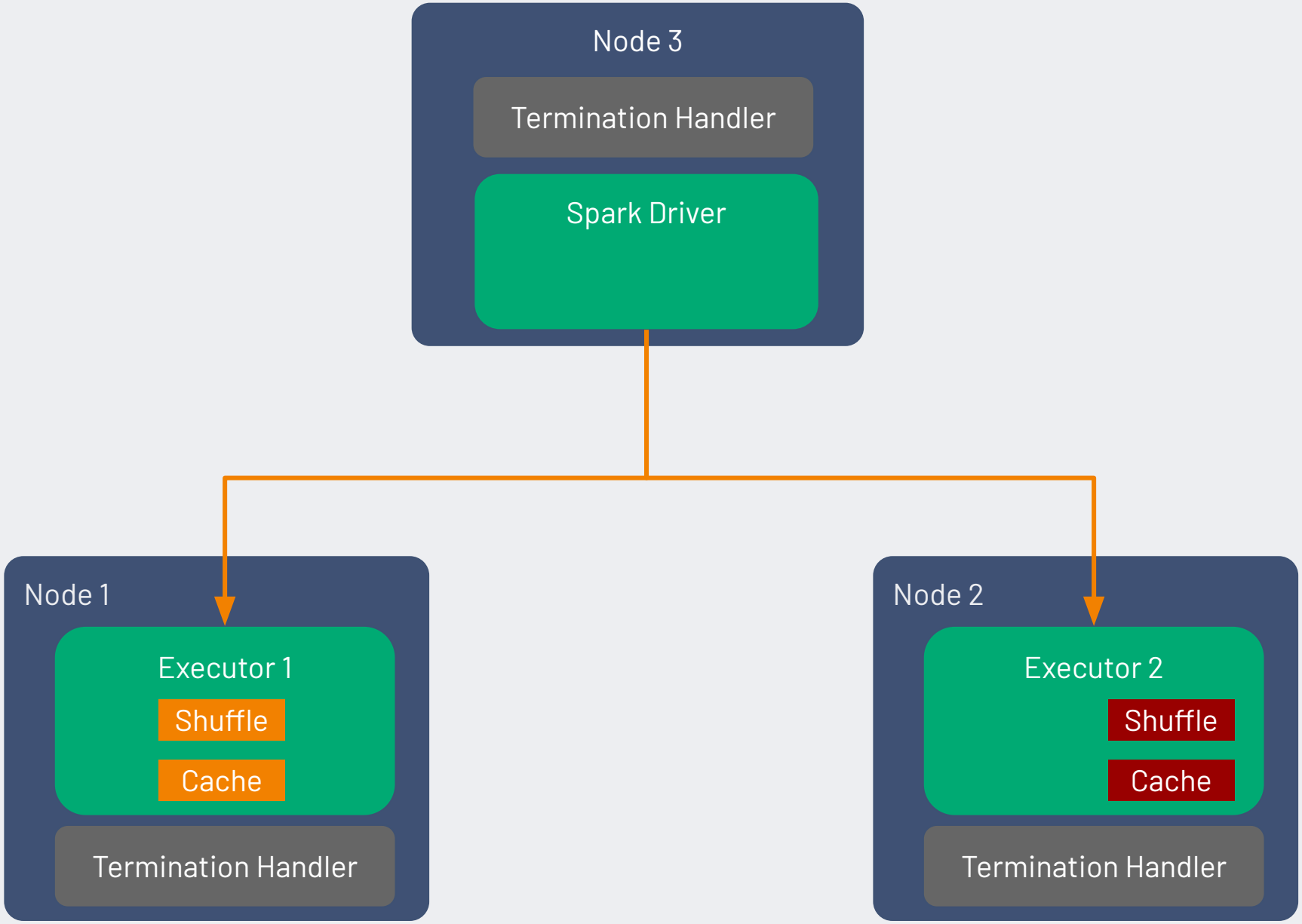
How to handle spot instances interruptions

Since Spark 3.1: Graceful Exec Decommissioning

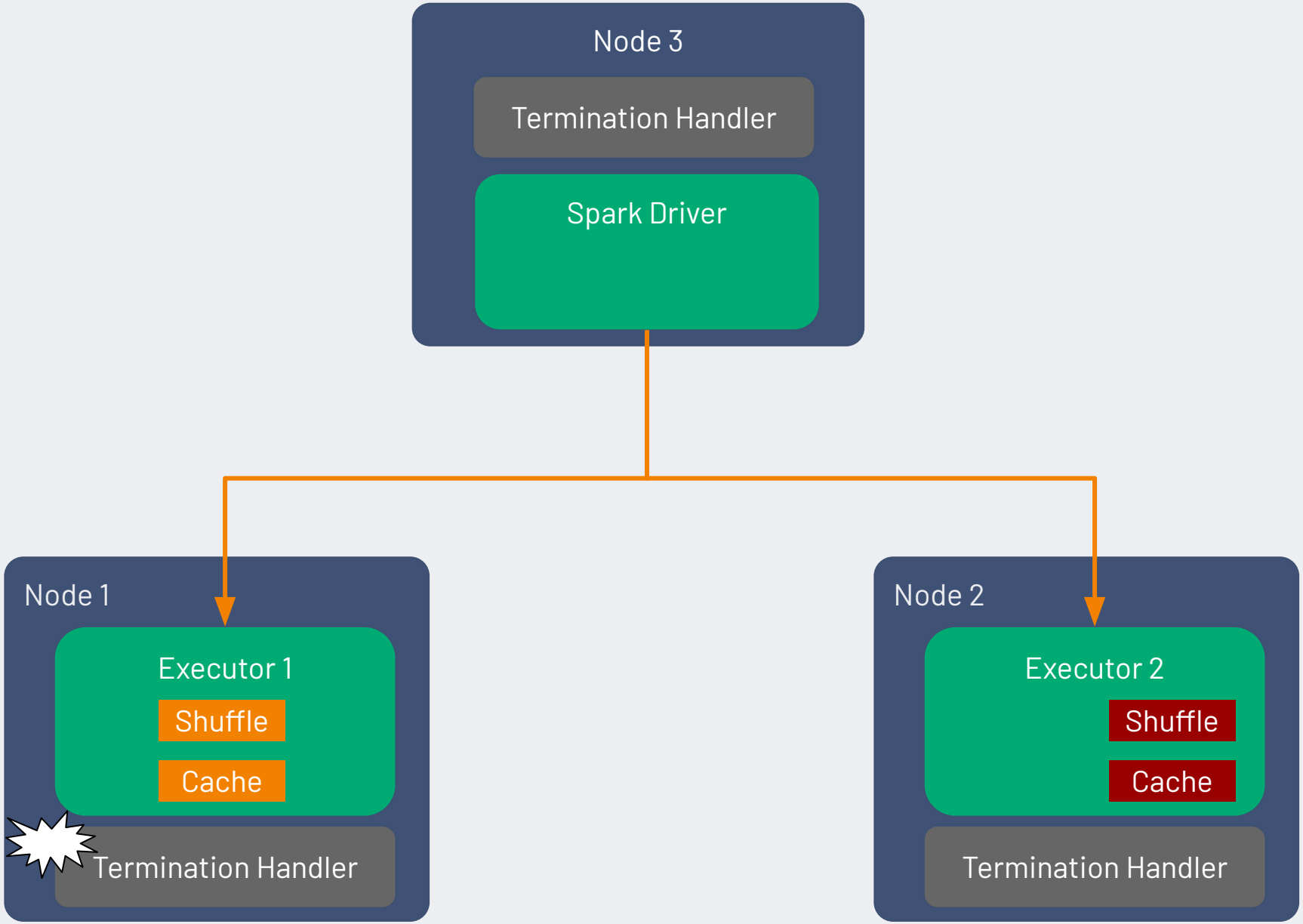
- Spark 3.1, k8s-only feature to gracefully handle exec termination during spot interruptions

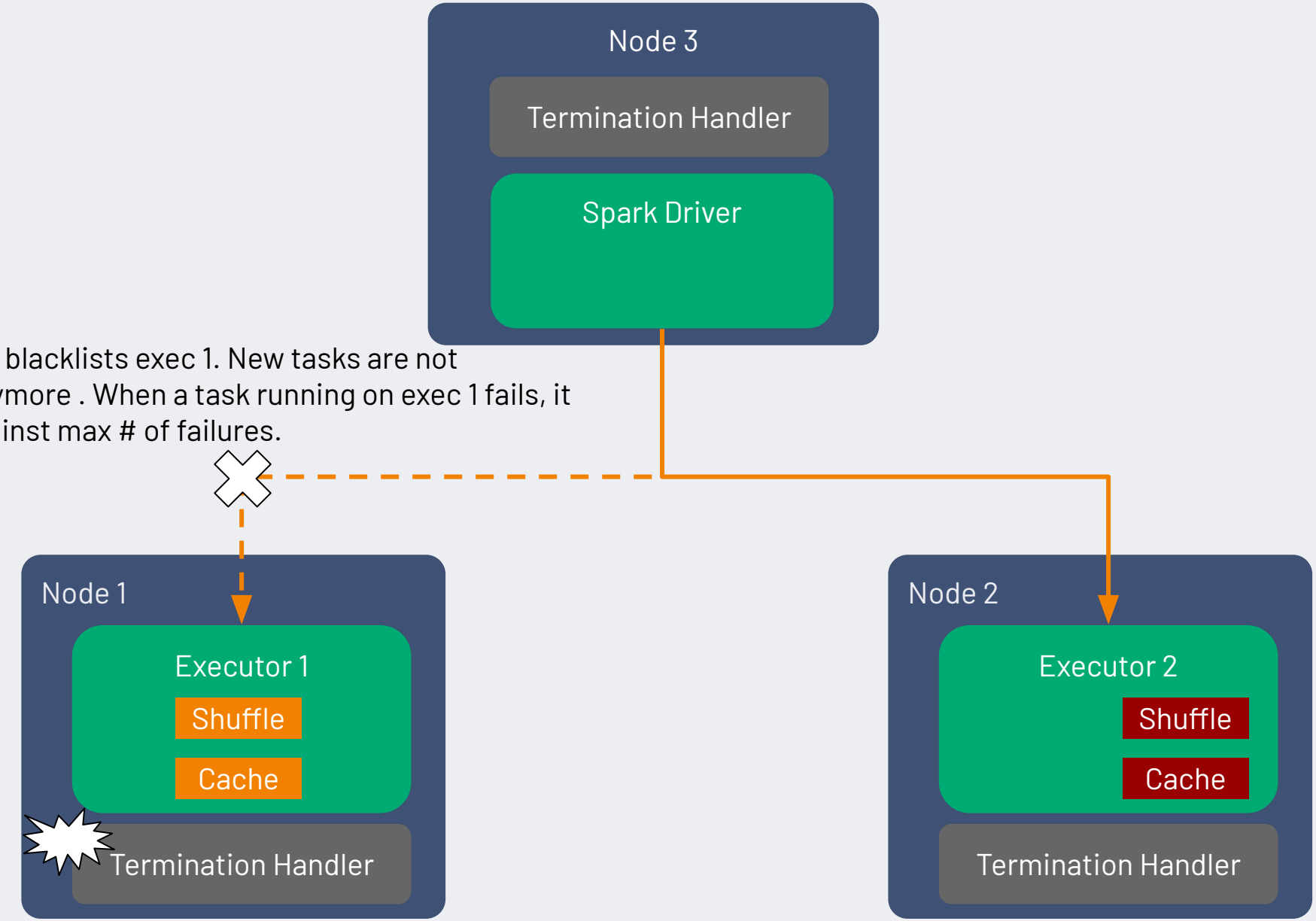
- Before interrupting a spot instance, cloud providers give a notice:
  - Termination notice: 2 min on AWS, 30s on GCP, 30s on Azure
  - This signal can be intercepted by a NodeTerminationHandler (k8s DaemonSet)
  - The daemonset then sends a message to the executor, which sends a message to the driver

- 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

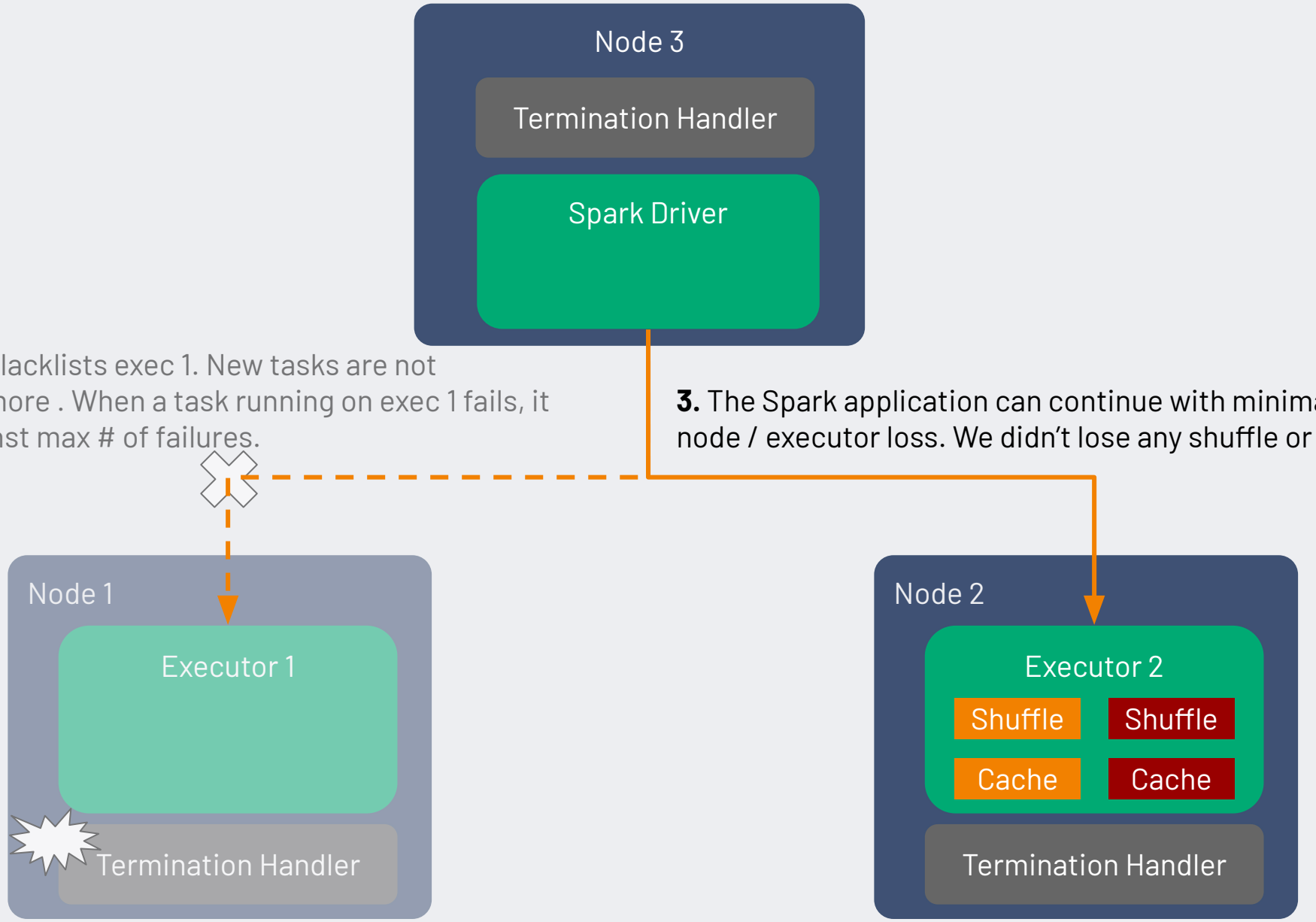
1. Termination Handler notices that the node is going to be spot-killed in 120 seconds.





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

Spark 3.1 - 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")

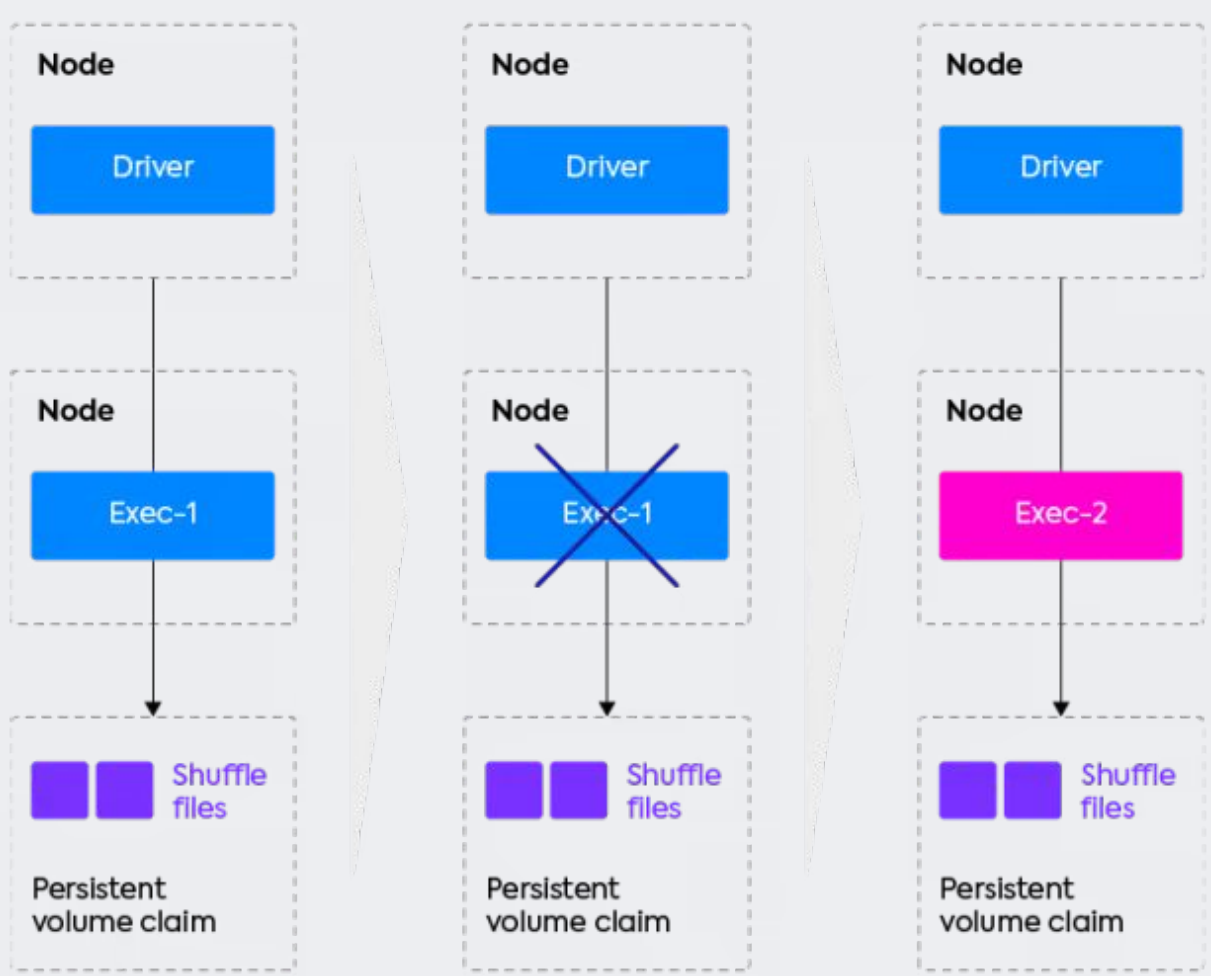
Graceful Exec Decommissioning - Experiment

- We ran workloads that produced a significant amount of shuffle files on the executors.
- We simulated the process of receiving a spot kill by detaching the nodes containing the executor pods from the cluster, with a grace period of 120 seconds (same as AWS)
- 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).**



Since Spark 3.2: 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. This can probably be fixed.
- 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

90% cost savings without trading off performance or stability

- Driver should run on demand, executors on spot
- Optimize the spot market to avoid 80% of 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 3.3 for Spark-on-k8s

- [SPARK-37810] Executor Rolling
  - When enabled, Spark can now forcibly decommission an executor at a certain interval. This is useful for long-running jobs (e.g. streaming) to avoid issues related to state accumulating on an executor (disk full, memory issues, stragglers, …).
- [SPARK-36057] Custom k8s Scheduler support (Volcano, Yunikorn)
  - Enable YARN-like capabilities such as queue, gang scheduling, etc

DATA+AI
SUMMIT 2022

Thank you

Jean-Yves Stephan
Senior Product Manager, Spot by NetApp

Hudson Buzby
Solutions Architect, Spot by NetApp

Learn More about Ocean for Apache Spark
https://spot.io/products/ocean-apache-spark/

APPENDIX

Abstract - Duration 40 min

Since the general availability of Apache Spark's native support for running on Kubernetes with Spark 3.1 in March 2021, the Spark community is increasingly choosing to run on k8s to benefit of containerization, efficient resource-sharing, and the tools from the cloud-native ecosystem.

Data teams are faced with complexities in this transition, including how to leverage spot VMs. These instances enable up to 90% cost savings but are not guaranteed to be available and face the risk of termination. This session will cover concrete guidelines on how to make Spark run reliably on spot instances, with code examples from real-world use cases.

Main topics:

- Using spot nodes for Spark executors
- Mixing instance types & sizes to reduce risk of spot interruptions - cluster autoscaling
- Spark 3.0: Graceful Decommissioning - preserve shuffle files on executor shutdown
- Spark 3.1: PVC reuse on executor restart - disaggregate compute & shuffle storage
- What to look for in future Spark releases