Hadoop/YARN Security

Two modes

1. Kerberos

Hadoop/YARN Security

Two modes

1. Kerberos
2. None (only mode until Hadoop 2.x)

NB: I recommend reading Steve Loughran's "Madness Beyond the Gate" to learn more

Hadoop/YARN Security

Containerization as of 3.1.1 (late 2018)

Security Warning

IMPORTANT This feature is experimental and is not complete. **IMPORTANT** Enabling this feature and running Docker containers in your cluster has security implications. With this feature enabled, it may be possible to gain root access to the YARN NodeManager hosts. Given Docker's integration with many powerful kernel features, it is imperative that administrators understand Docker security before enabling this feature.

Performance in YARN

YARN's scheduler attempts to maximize utilization

Performance in YARN

YARN's scheduler attempts to maximize utilization

Spark on YARN with dynamic allocation is great at:

- extracting maximum performance from static resources
- providing bursts of resources for one-off batch work
- running "just one more thing"

Performance in YARN

YARN's scheduler attempts to maximize utilization

Spark on YARN with dynamic allocation is great at:

- extracting maximum performance from static resources
- providing bursts of resources for one-off batch work
- running "just one more thing"

YARN: Clown Car Scheduling

(Image Credit: 20th Century Fox Television)

Performance in YARN

YARN's scheduler attempts to maximize utilization

Spark on YARN with dynamic allocation is terrible at:

- providing consistency from run to run
- isolating performance between different users/tenants

  (i.e., if you kick off a big job, then my job is likely to run slower)

So… Kubernetes?

- ✓ Native containerization
- ✓ Extreme extensibility (e.g., scheduler, networking/firewalls)
- ✓ Active community with a fast-moving code base
- ✓ Single platform for microservices and compute*

*Spoiler alert: the Kubernetes scheduler is excellent for web services, not optimized for batch

SPARK+AI SUMMIT 2019

Spark on Kubernetes

Palantir

Timeline

Sep '16
Begin initial prototype of Spark on K8s

Nov '16
Establish working group with community

Mar '17
Minimal integration complete, begin experimental deployment

Oct '17
First PR merged into upstream master

Jan '18
Begin first migration from YARN toK8s

Jun '18
Complete first migration from YARN to K8s in production

Spark on K8s Architecture

→ Client runs spark-submit with arguments

→ spark-submit converts arguments into a PodSpec for the driver

Spark on K8s Architecture

→ Client runs spark-submit with arguments

→ spark-submit converts arguments into a PodSpec for the driver

→ K8s-specific implementation of SchedulerBackend interface requests executor pods in batches

Spark on K8s Architecture

spark-submit input:

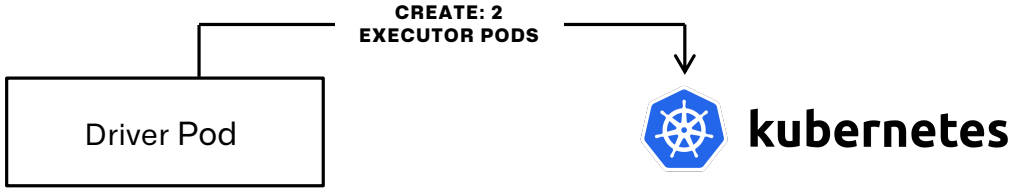
```
…
--master k8s://example.com:8443
--conf spark.kubernetes.image=example.com/appImage
--conf …
com.palantir.spark.app.main.Main
```

spark-submit output:

```
…
spec:
  containers:
  - name: example
    image: example.com/appImage
    command: [”/opt/spark/entrypoint.sh"]
    args: [”--driver", ”--class”, “com.palantir.spark.app.main.Main”]
…
```

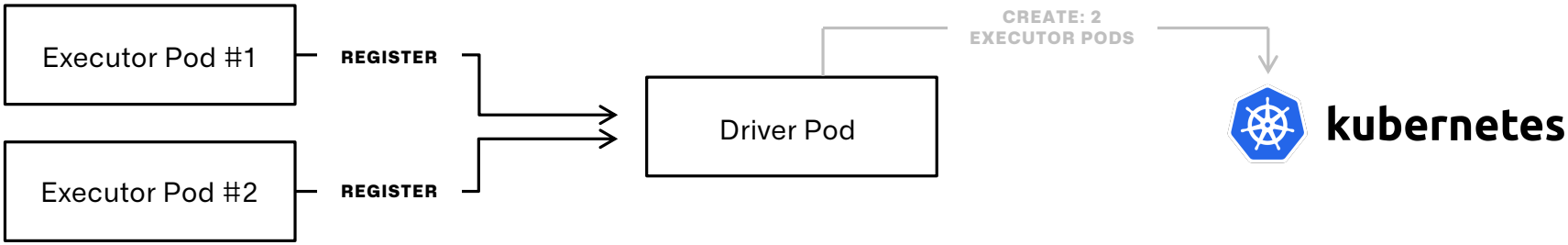
Spark on K8s Architecture

spark.executor.instances = 4, spark.kubernetes.allocation.batch.size = 2



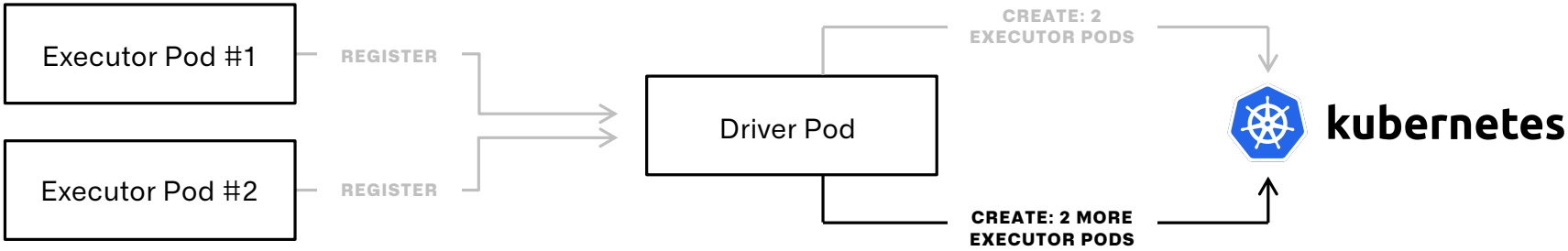
Spark on K8s Architecture

spark.executor.instances = 4, spark.kubernetes.allocation.batch.size = 2



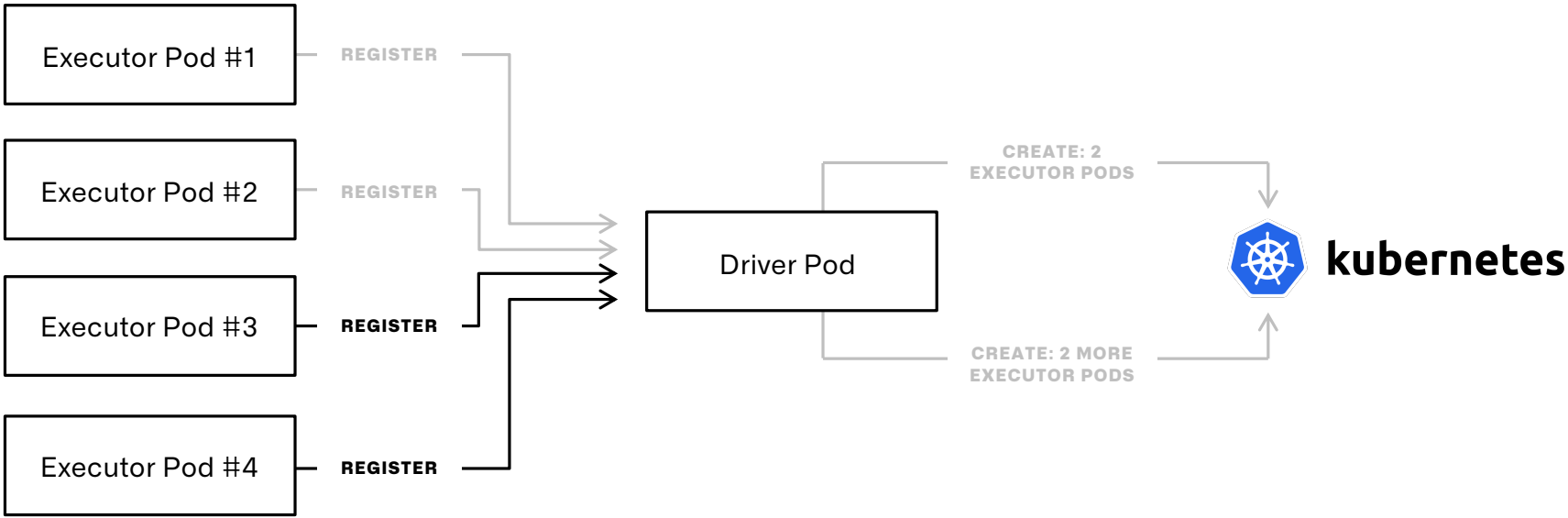
Spark on K8s Architecture

spark.executor.instances = 4, spark.kubernetes.allocation.batch.size = 2



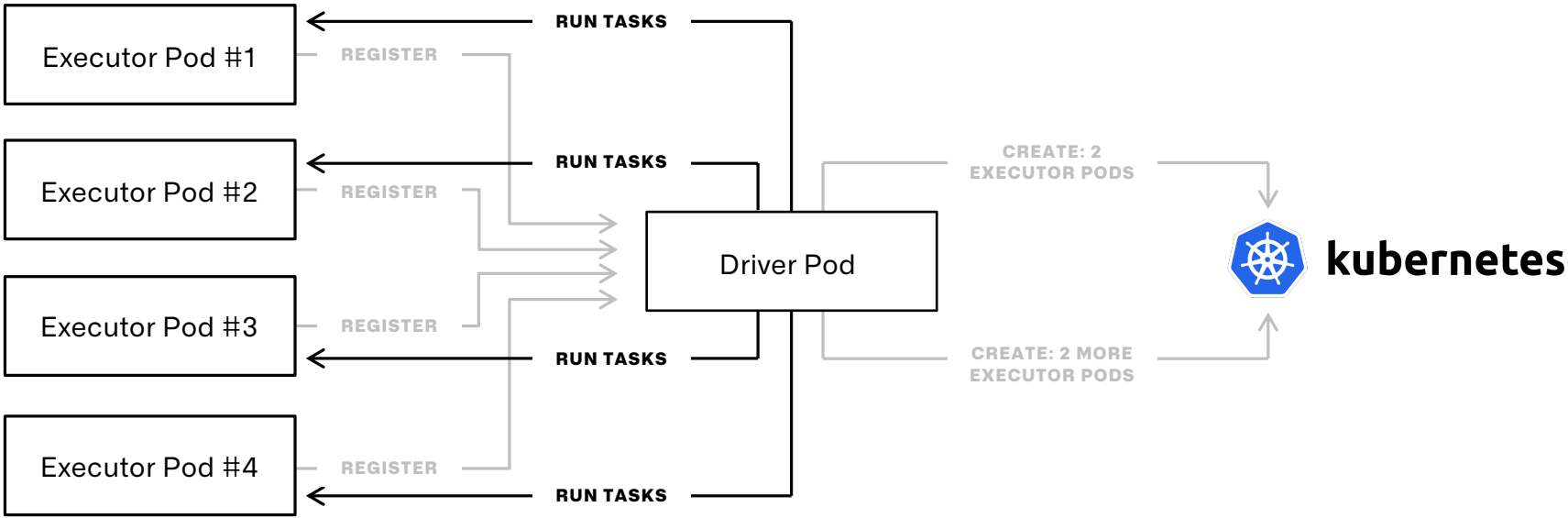
Spark on K8s Architecture

spark.executor.instances = 4, spark.kubernetes.allocation.batch.size = 2

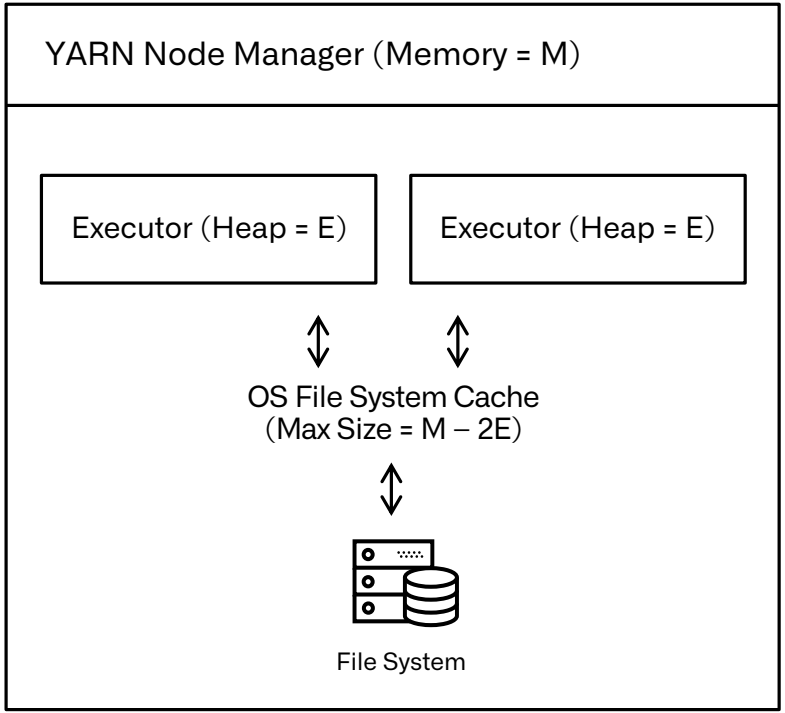


Spark on K8s Architecture

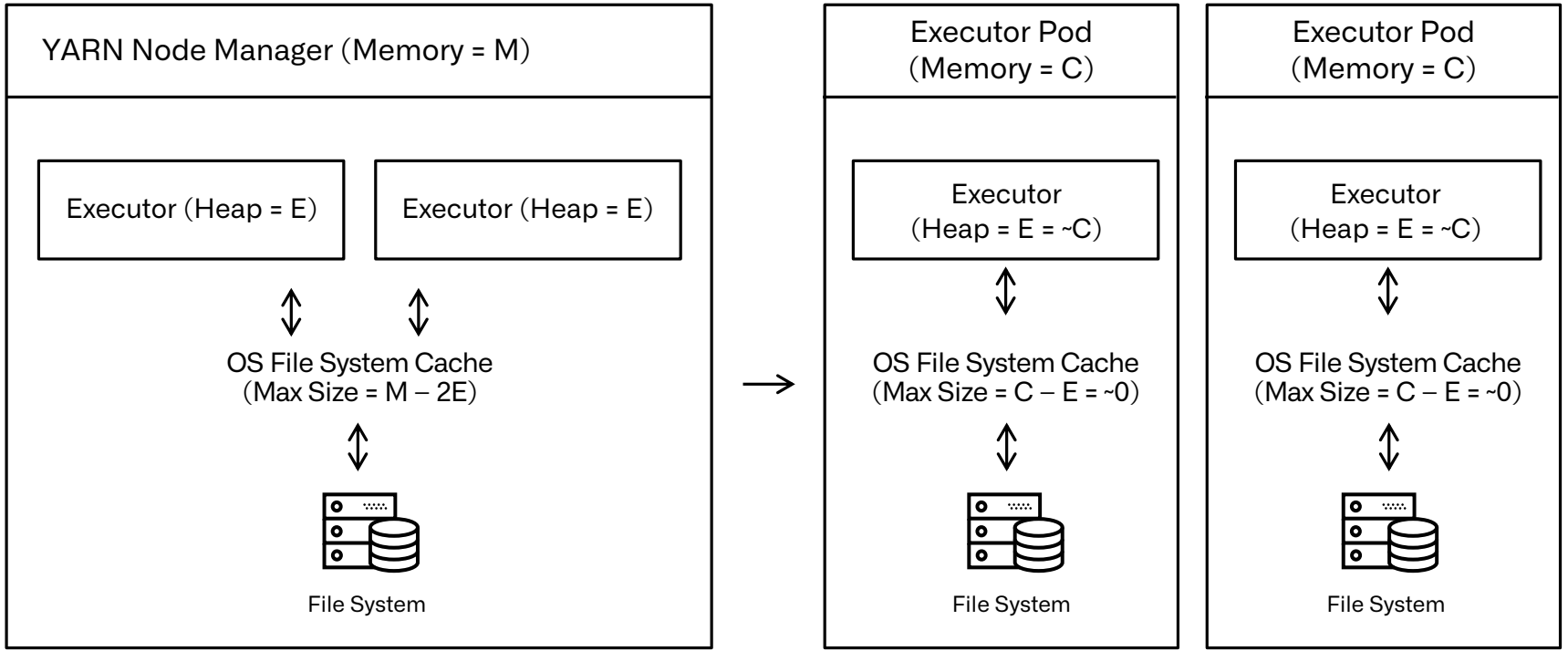
spark.executor.instances = 4, spark.kubernetes.allocation.batch.size = 2



Early Challenge: Disk Performance



Early Challenge: Disk Performance



Early Challenge: Disk Performance

OS filesystem cache is now container-local

- i.e., dramatically smaller
- disks must be fast without hitting FS cache
- Solution: use NVMe drives for temp storage

Docker disk interface is slower than direct disk access

- Solution: Use EmptyDir volumes for temp storage

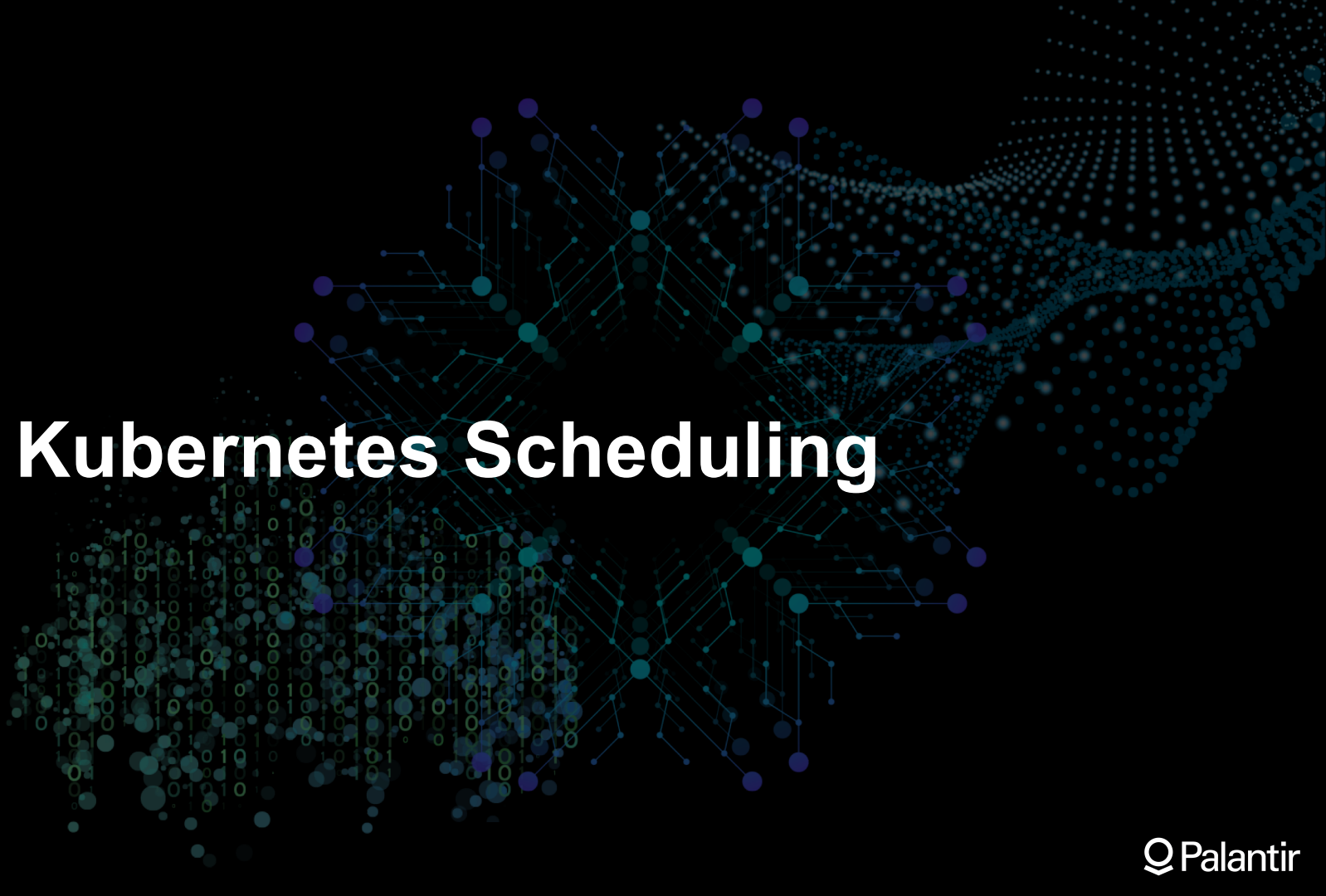
Key Production Challenges

Challenge I: Kubernetes Scheduling

- The built-in Kubernetes scheduler isn't really designed for distributed batch workloads (e.g., MapReduce)
- Historically, optimized for microservice instances or single-pod, one-off jobs (what k8s natively supports)

Challenge II: Shuffle Resiliency

- External Shuffle Service is unavailable in Kubernetes
- Jobs must be written more carefully to avoid executor failure (e.g., OOM) and the subsequent need to recompute lost blocks



Kubernetes Scheduling

Reliable, Repeatable Runtimes

Recall → A key goal is to make runtimes of the same workload consistent from run to run

Reliable, Repeatable Runtimes

Recall → A key goal is to make runtimes of the same workload consistent from run to run

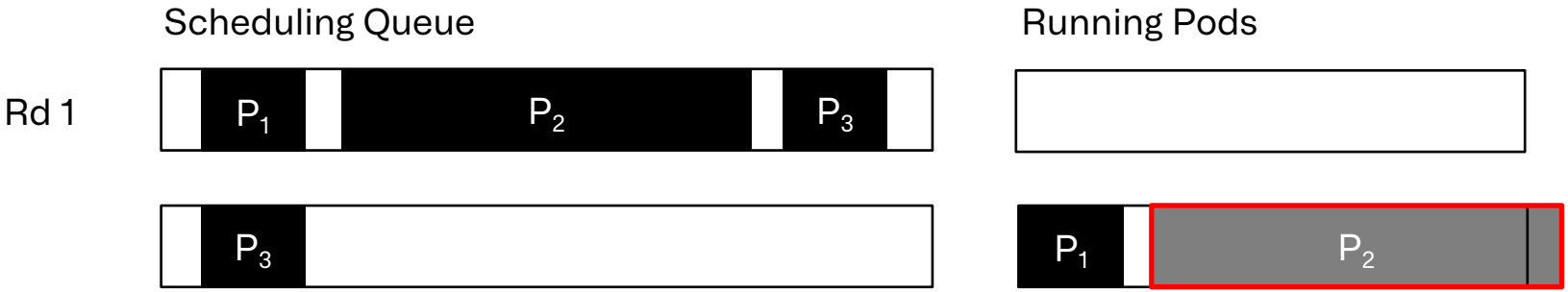
Corollary → When a driver is deployed, wants to do work, it should receive the same resources from run to run

Reliable, Repeatable Runtimes

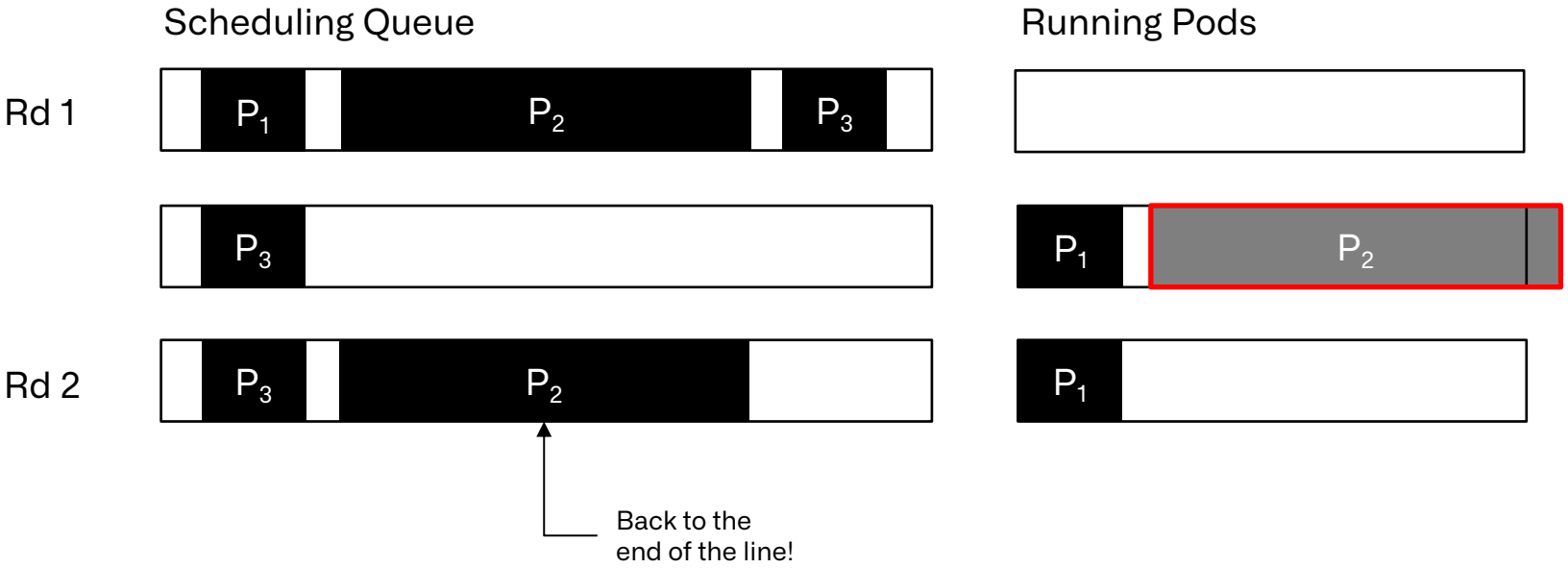
Recall	→	A key goal is to make runtimes of the same workload consistent from run to run
Corollary	→	When a driver is deployed, wants to do work, it should receive the same resources from run to run
Problem	→	Using vanilla k8s scheduler led to partial starvation as the cluster became saturated

Kubernetes Scheduling

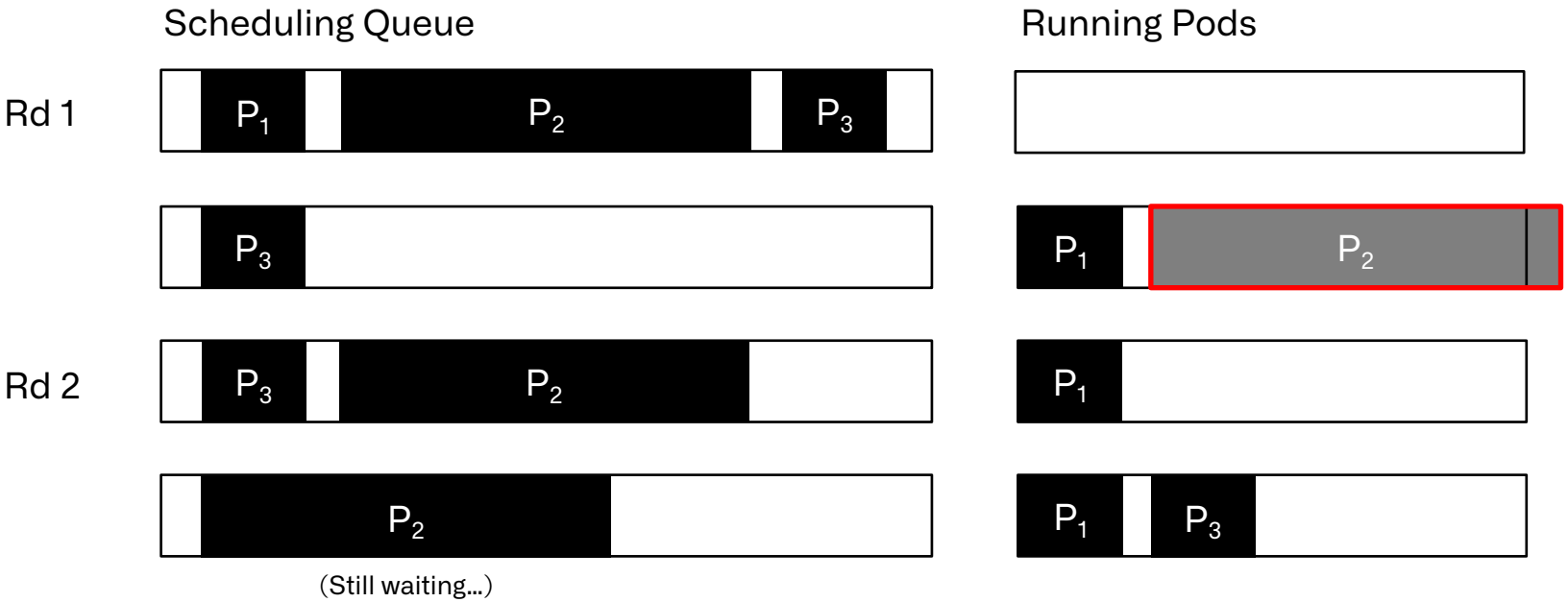
Kubernetes Scheduling



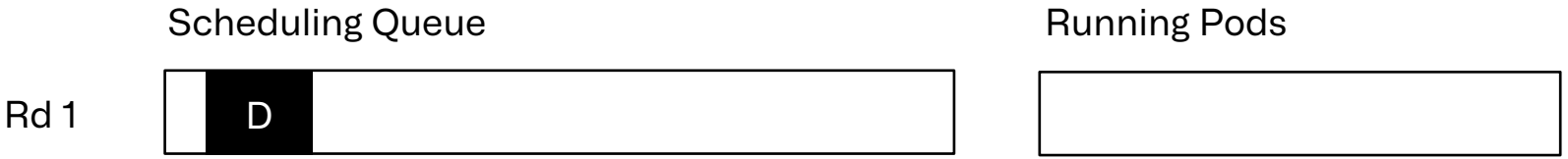
Kubernetes Scheduling



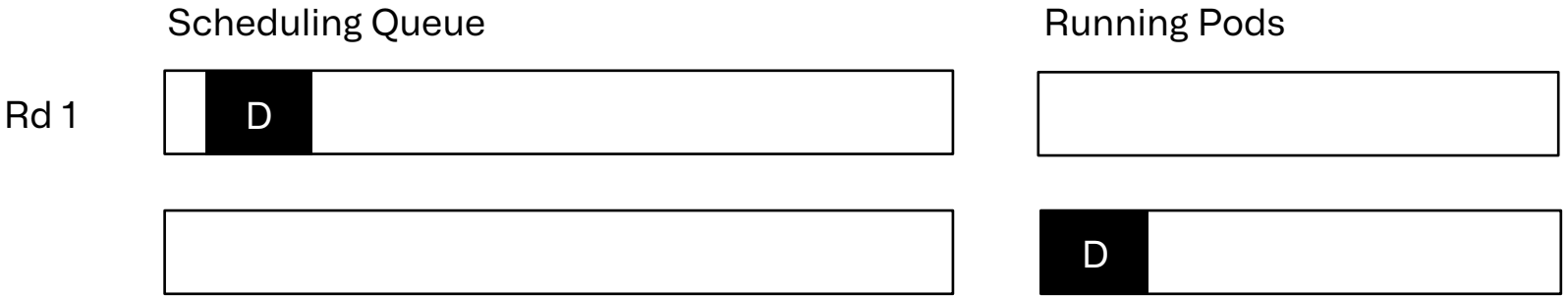
Kubernetes Scheduling



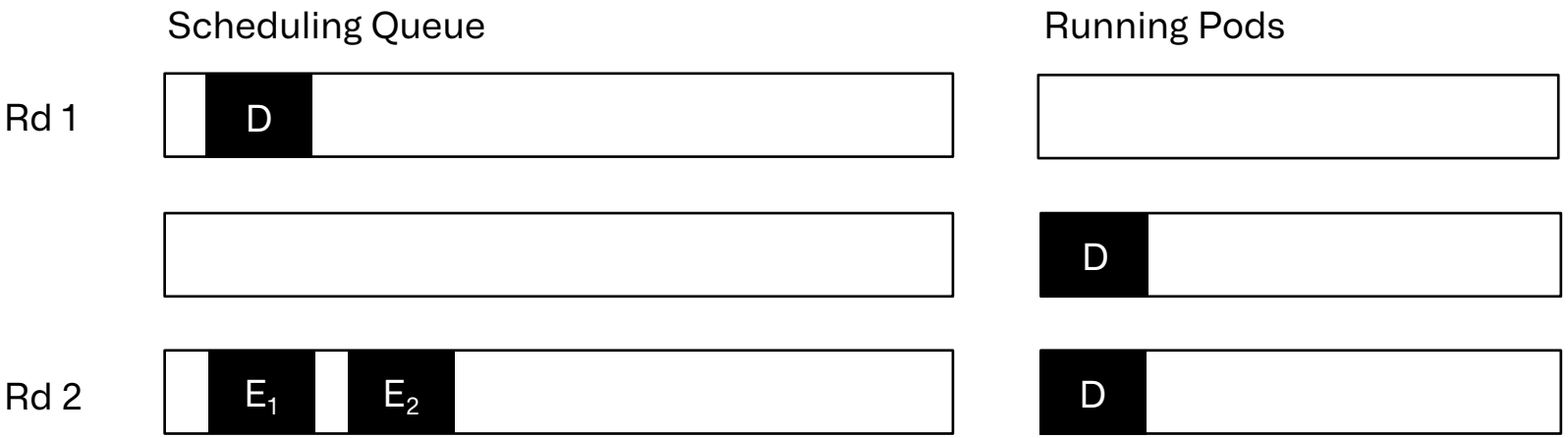
Naïve Spark-on-K8s Scheduling



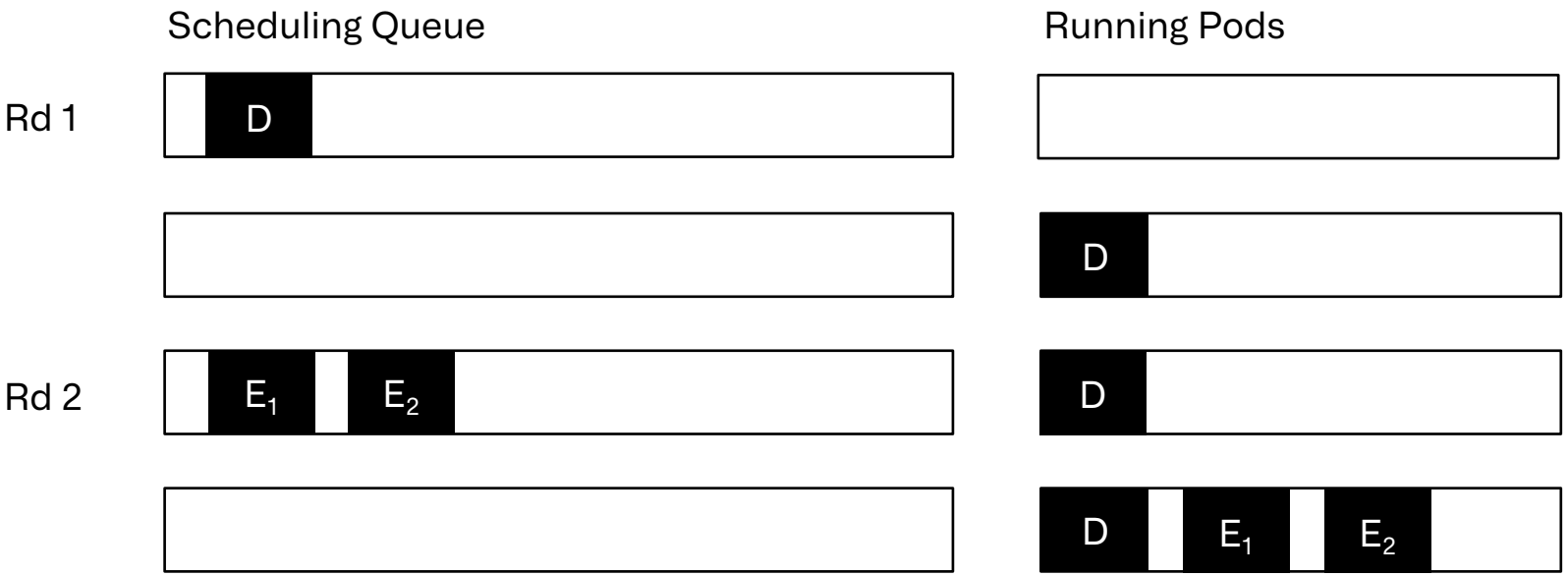
Naïve Spark-on-K8s Scheduling



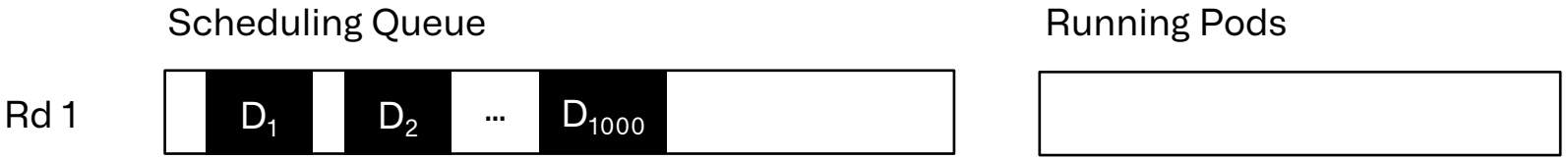
Naïve Spark-on-K8s Scheduling



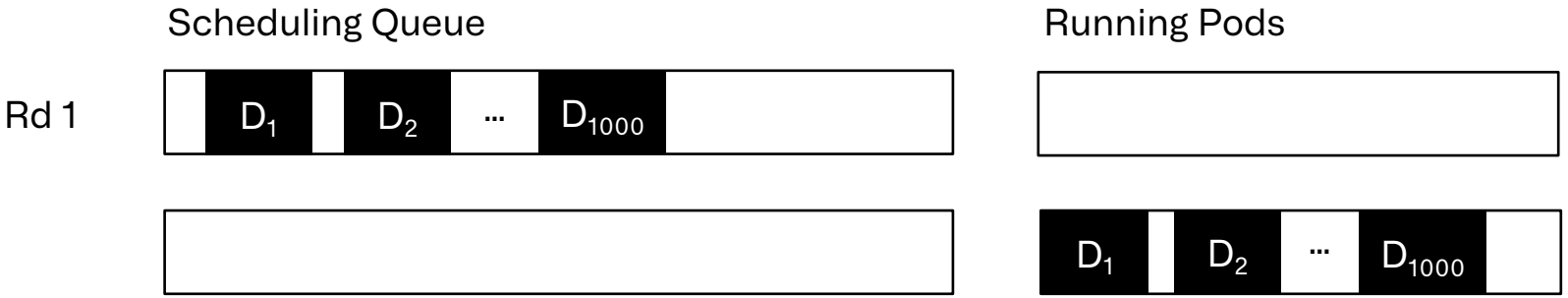
Naïve Spark-on-K8s Scheduling



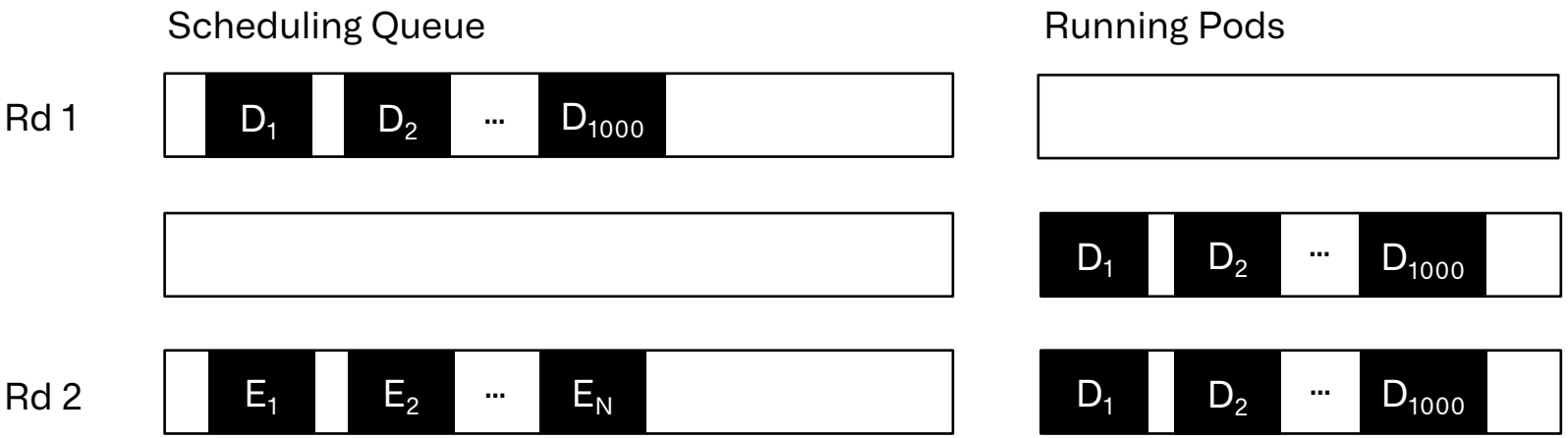
The “1000 Drivers” Problem



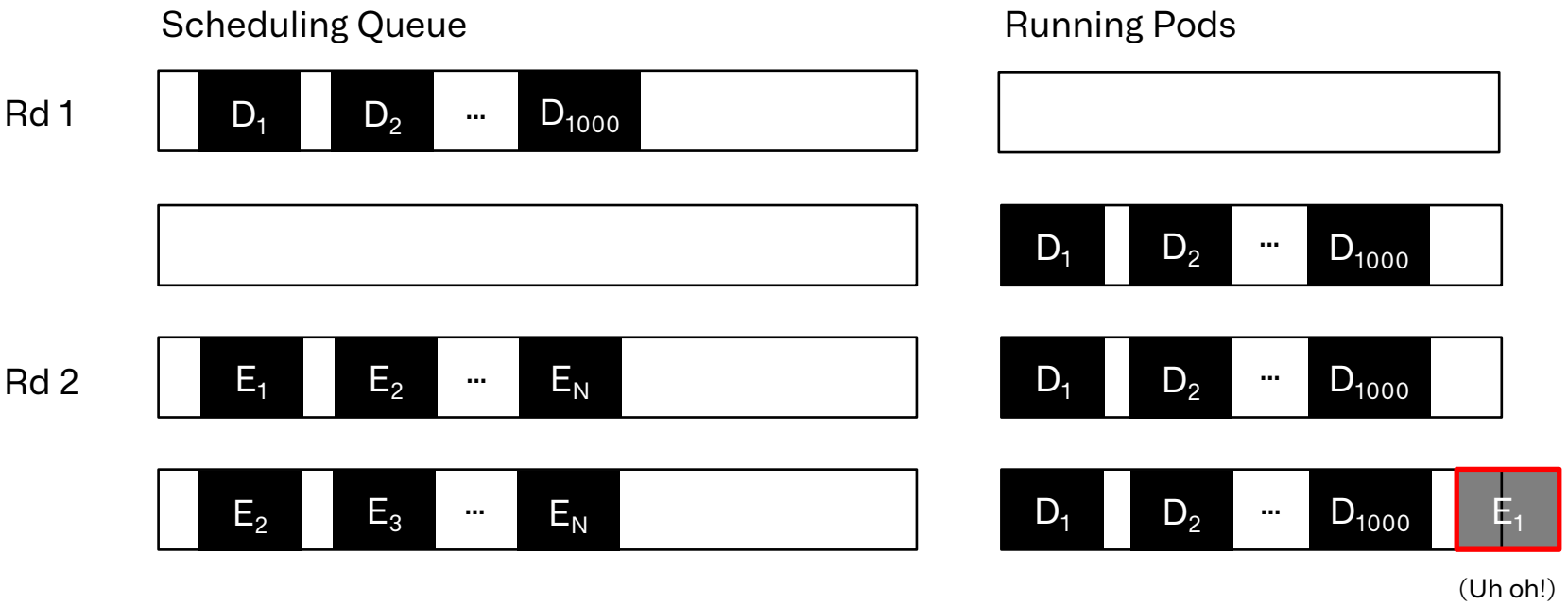
The “1000 Drivers” Problem



The “1000 Drivers” Problem



The “1000 Drivers” Problem



So… Kubernetes?

- ✓ Native containerization
- ✓ Extreme extensibility (e.g., scheduler, networking/firewalls)
- ✓ Active community with a fast-moving code base
- ✓ Single platform for microservices and compute*

*Spoiler alert: the Kubernetes scheduler is excellent for web services, not optimized for batch

K8s Spark Scheduler

Idea: use the Kubernetes scheduler extender API to add

- Gang scheduling of drivers & executors
- FIFO (within instance groups)

K8s Spark Scheduler

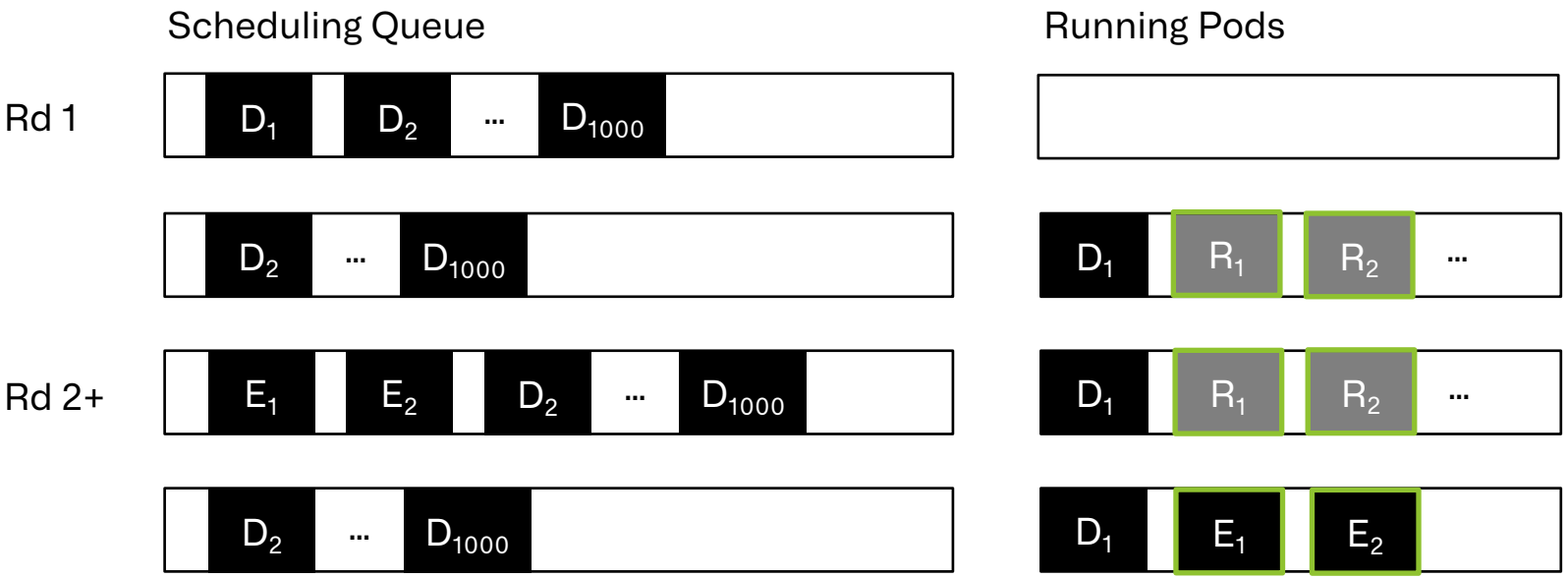
Goal: build entirely with native K8s extension points

- Scheduler extender API: fn(pod, [node]) -> Option[node]
- Custom resource definition: ResourceReservation
- Driver annotations for resource requests (executors)

K8s Spark Scheduler

if driver	→	get cluster usage (running pods + reservations)
	→	bin pack pending resource requests in FIFO order
	→	reserve resources (with CRD)
if executor	→	find unbound reservation
	→	bind reservation

K8s Spark Scheduler

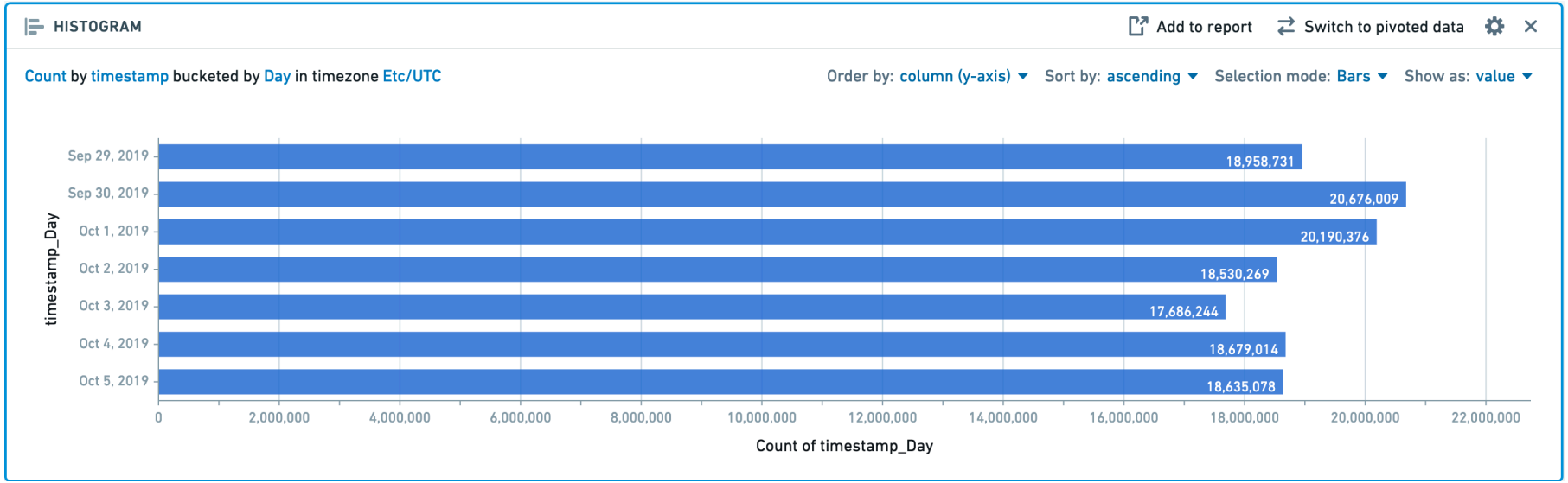


Spark-Aware Autoscaling

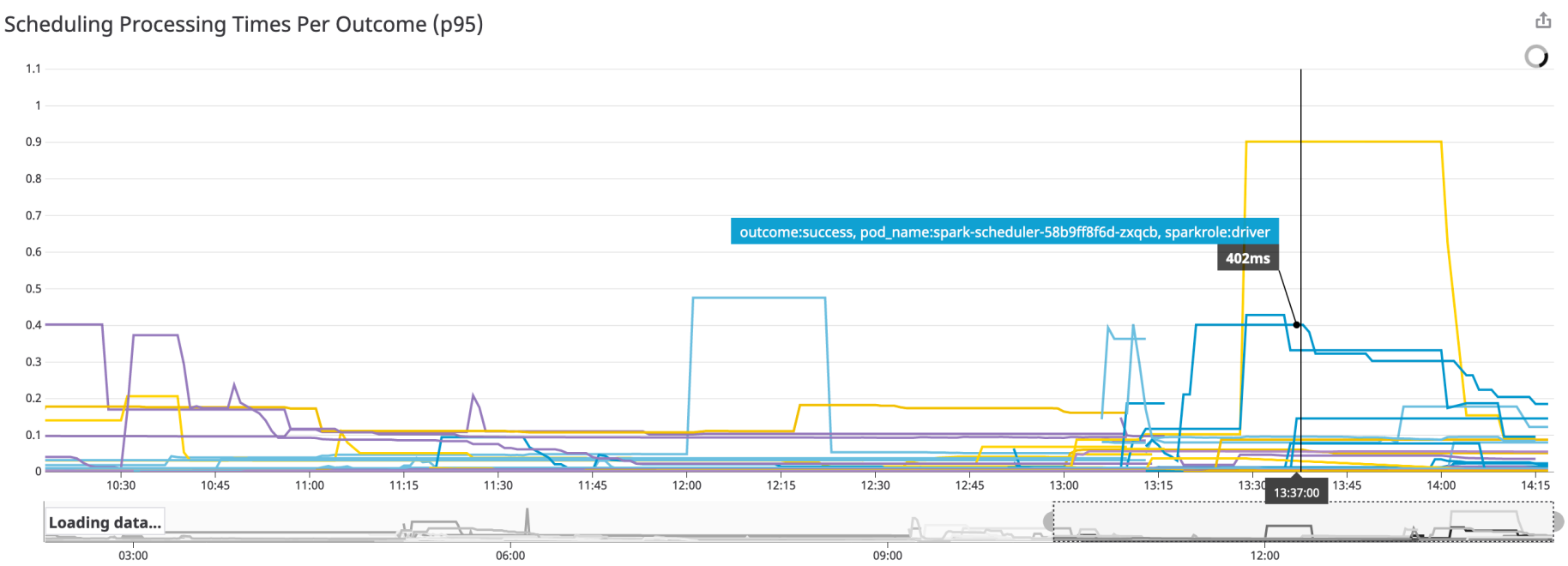
Idea: use the resource request annotations for unscheduled drivers to project desired cluster size

- Again, we use a CRD to represent this unsatisfied *Demand*
- Sum resources for pods + reservations + demand
- In the interest of time, out of scope for today

Spark pods per day



Pod processing time



“Soft” dynamic allocation

Problem → Static allocation wastes resources

“Soft” dynamic allocation

Problem	→	Static allocation wastes resources
Idea	→	Voluntarily give up executors that aren’t needed, no preemption (same from run to run)

“Soft” dynamic allocation

Problem	→	Static allocation wastes resources
Idea	→	Voluntarily give up executors that aren’t needed, no preemption (same from run to run)
Problem	→	No external shuffle, so executors store shuffle files

“Soft” dynamic allocation

Problem	→	Static allocation wastes resources
Idea	→	Voluntarily give up executors that aren’t needed, no preemption (same from run to run)
Problem	→	No external shuffle, so executors store shuffle files
Idea	→	The driver already tracks shuffle file locations, so it can determine which executors are safe to give up

“Soft” dynamic allocation

- Saves $$$$, ~no runtime variance if consistent from run to run
- Inspired by a 2018 Databricks blog post [1]
- Merged into our fork in ~January 2019
- Recently adapted by @vanzin (thanks!) and merged upstream [2]

[1] https://databricks.com/blog/2018/05/02/introducing-databricks-optimized-auto-scaling.html

[2] https://github.com/apache/spark/pull/24817

K8s Spark Scheduler

See our engineering blog on Medium

- https://medium.com/palantir/spark-scheduling-in-kubernetes-4976333235f3

Or, check out the source for yourself! (Apache v2 license)

- https://github.com/palantir/k8s-spark-scheduler

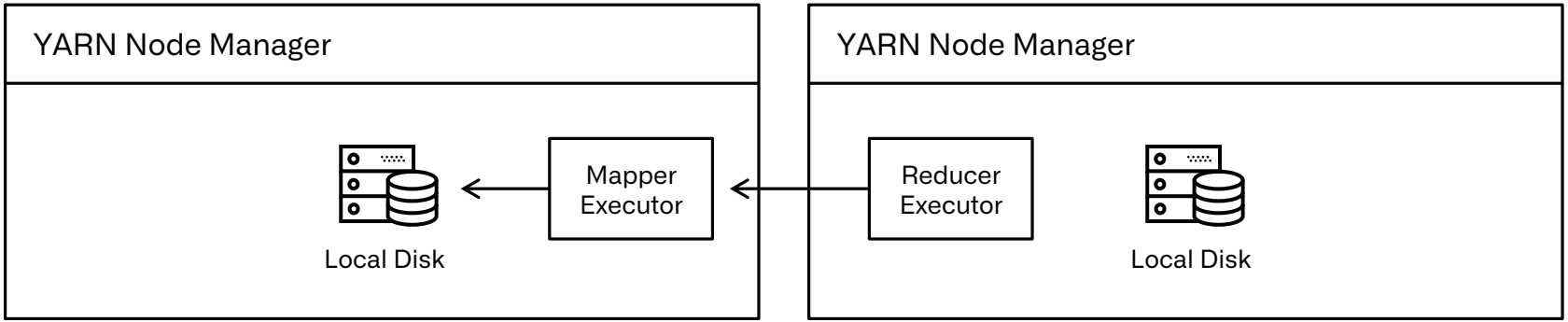
Shuffle Resiliency

Shuffle Resiliency

Shuffles have a map side and a reduce side

— Mapper executors write temporary data to local disk

— Reducer executors contact mapper executors to retrieve written data

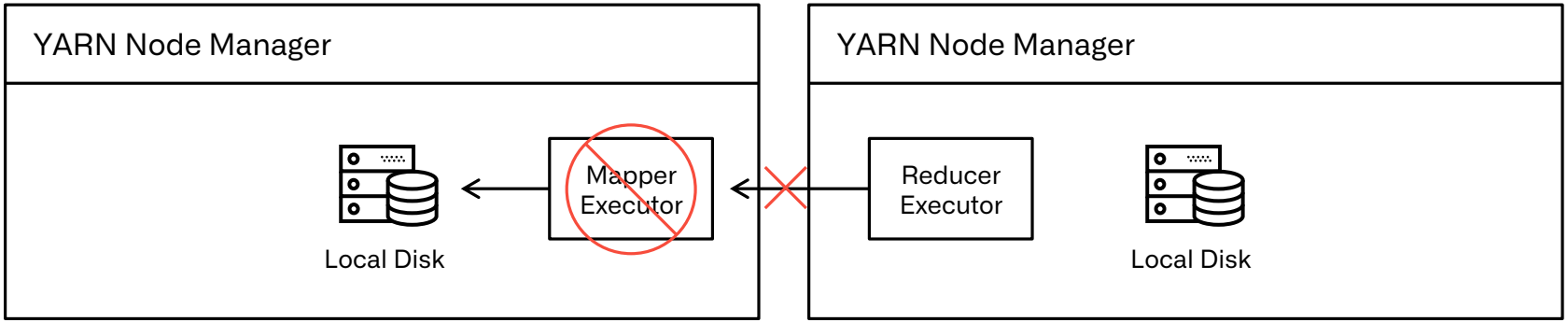


Shuffle Resiliency

If an executor crashes, all data written by that executor's map tasks are lost

— Spark must re-schedule the map tasks on other executors

— Cannot remove executors to save resources because we would lose shuffle files

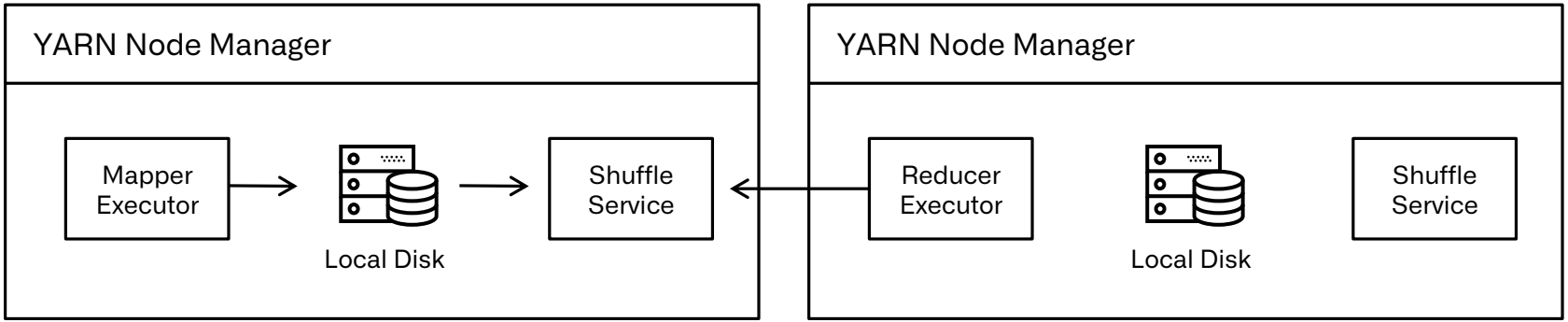


Shuffle Resiliency

Spark's external shuffle service preserves shuffle data in cases of executor loss

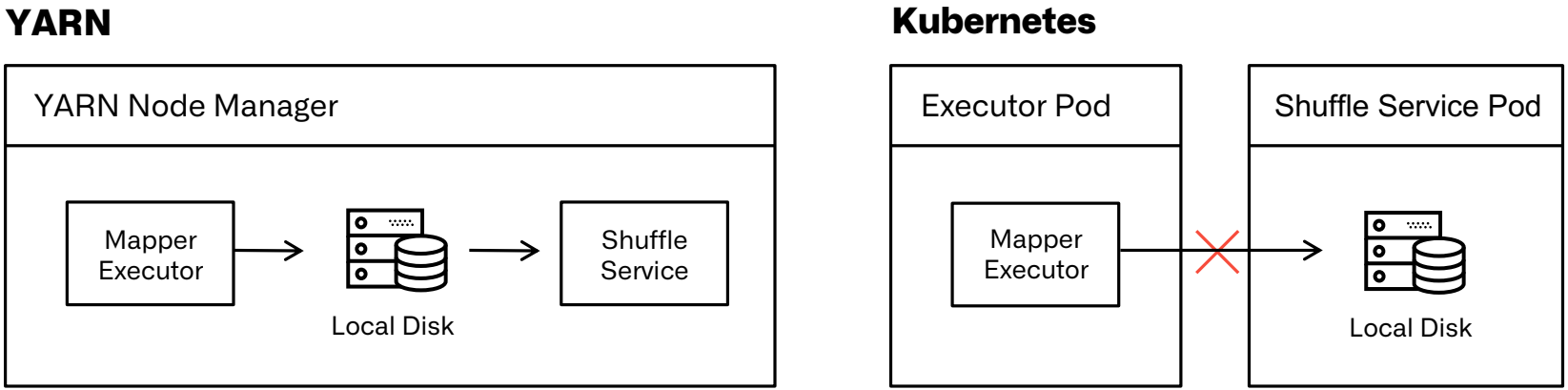
— Required to make preemption non-pathological

— Prevents loss of work when executors crash



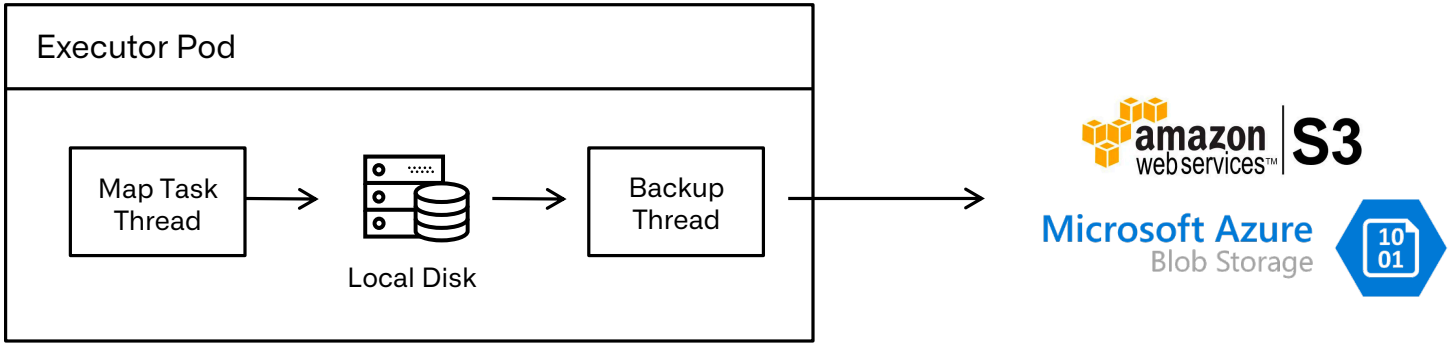
Shuffle Resiliency

Problem: for security, containers have isolated storage



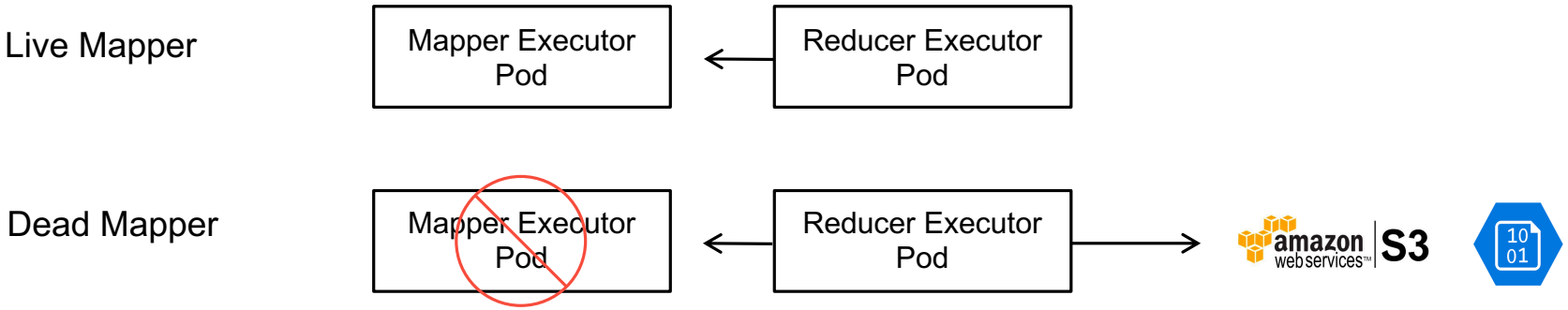
Shuffle Resiliency

Idea: asynchronously back up shuffle files to a distributed storage system



Shuffle Resiliency

1. Reducers first try to fetch from other executors
2. Download from remote storage if mapper is unreachable



Shuffle Resiliency

- Wanted to generalize the framework for storing shuffle data in arbitrary storage systems
- API In Progress: https://issues.apache.org/jira/browse/SPARK-25299
- **Goal**: Open source asynchronous backup strategy by end of 2019