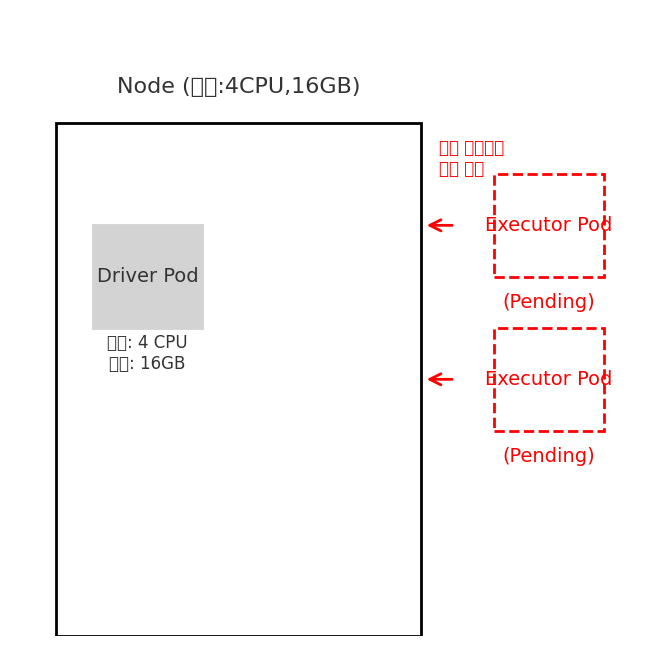
# Kubernetes 상 Spark Operator 환경에서 Gang Scheduling 미적용 시 문제 분석

## Spark on Kubernetes와 Gang Scheduling 개요

Spark를 Kubernetes에서 실행할 때 **Spark Operator**를 사용하면 Spark 애플리케이션의 드라이버(Driver)와 Executor가 각각 Kubernetes **Pod**로 생성됩니다[[1]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Spark%20applications%20consist%20of%20a,itself%20launches%20the%20executor%20pods). 기본적으로 Kubernetes의 **디폴트 스케줄러**는 개별 Pod 단위로 스케줄링을 처리하기 때문에, Spark 작업의 드라이버 및 여러 Executor Pod들이 순차적이고 부분적으로 할당됩니다[[2]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Additionally%2C%20because%20the%20pods%20from,look%20at%20a%20simple%20example). **Gang Scheduling**이란 이러한 병렬 작업의 관련된 모든 작업을 **동시에(All-or-Nothing)** 스케줄링하는 방법으로[[3]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=According%20to%C2%A0wikipedia%2C%20Gang%20Scheduling%20refers,until%20it%20reaches%20this%20state)[[4]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Gang%20scheduling,application), Spark와 같은 배치 작업의 리소스 확보 문제를 해결하기 위해 고안되었습니다[[5]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Gang%20scheduling%20is%20very%20useful,for%20computing%20engines%20like%20Spark). 그러나 **Gang Scheduling이 적용되지 않은 경우**, 아래와 같은 여러 문제가 발생할 수 있습니다.

## Gang Scheduling 미적용 시 발생하는 주요 문제 시나리오

### 1) **자원 부족으로 인한 Executor Pending 및 Job 지연**

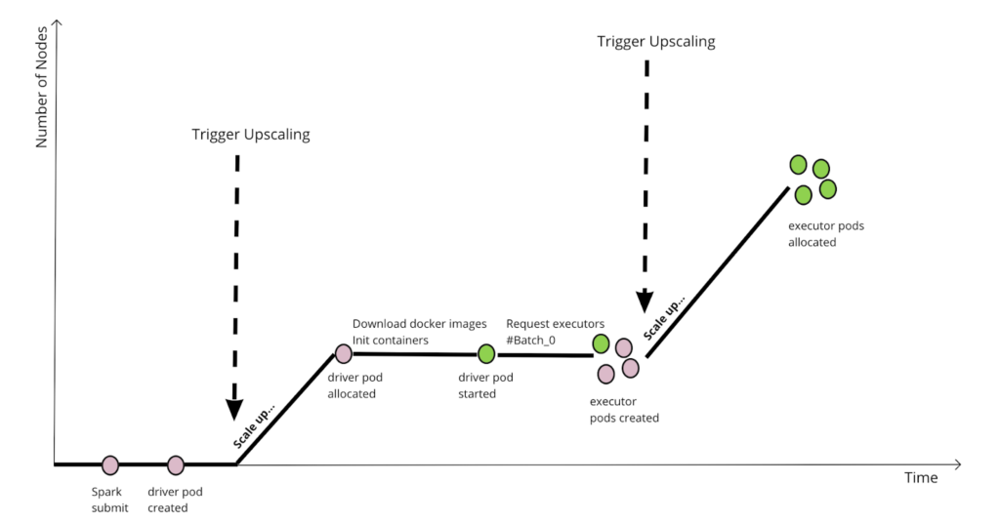
  
*하나의 Spark 작업에서 드라이버 Pod만 스케줄되고 Executor Pod들은 클러스터 자원 부족으로 Pending 상태에 머물러 있는 상황을 도식화한 그림. Node(노드)의 용량(예: 4 CPU, 16GB)을 드라이버가 모두 차지하여, 추가 Executor를 위한 여유가 없으므로 오른쪽의 Executor Pod들이 Pending(빨간 점선 상자)으로 표시된다. 이런 경우 Executor들은 스케줄되지 못하고 작업은 진행되지 않는다.*

Spark Operator를 통해 Spark 애플리케이션을 제출하면 먼저 **드라이버 Pod**가 생성되고 스케줄됩니다[[1]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Spark%20applications%20consist%20of%20a,itself%20launches%20the%20executor%20pods). 드라이버는 애플리케이션 실행을 시작한 후 지정된 수의 **Executor Pod 생성을 요청**합니다. 이때 클러스터의 **가용 자원(CPU, 메모리)**이 부족하면 Executor Pod들은 노드에 배치되지 못하고 계속 **Pending 상태**로 대기하게 됩니다. 예를 들어, Executor Pod가 요구하는 메모리나 CPU가 현재 클러스터에 없으면 Kubernetes 스케줄러는 해당 Pod들을 스케줄하지 못하고, kubectl describe pod 명령의 이벤트에는 *“0/1 nodes are available: 1 Insufficient cpu, 1 Insufficient memory.”*와 같은 **FailedScheduling** 메세지가 나타납니다[[6]](https://stackoverflow.com/questions/65683383/spark-executors-pods-in-pending-state-on-kubernetes-deployment#:~:text=%24%20kubectl%20describe%20pod%2Fspark,Insufficient%20cpu%2C%201%20Insufficient%20memory). 실제로 Spark 드라이버 로그를 보면 *“Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources”*라는 경고가 반복되어, Executor가 전혀 시작되지 못했음을 보여줍니다[[7]](https://stackoverflow.com/questions/65683383/spark-executors-pods-in-pending-state-on-kubernetes-deployment#:~:text=%24%20kubectl%20logs%20pod%2Fspark,workers%20are%20registered%20and%20have). 이러한 상황에서는 드라이버가 **필요한 Executor들을 확보할 때까지 무기한 대기**하며, 작업이 시작되지 못하고 **지연**됩니다. 기본 스파크 설정에서는 Kubernetes에서 Executor가 생기지 않더라도 드라이버가 자동으로 종료되지 않기 때문에[[8]](https://stackoverflow.com/questions/65683383/spark-executors-pods-in-pending-state-on-kubernetes-deployment#:~:text=Now%20my%20question%20here%20is,were%20no%20resources%20in%20cluster), 별도의 타임아웃이나 재시도 제한이 없어 자원이 확보되지 않는 한 **Job이 영구히 대기하거나 수동으로 중단될 때까지 정지** 상태가 될 수 있습니다. 결과적으로 **클러스터 자원 부족**으로 인해 Spark 작업이 **스타트하지 못하거나 매우 늦게 시작**되는 문제가 발생합니다.

이 시나리오에서 Kubernetes 리소스 상태를 요약하면, **드라이버 Pod는 Running**되었으나 **모든 Executor Pod들이 Pending**으로 남아 있습니다. 노드들의 자원 현황은 드라이버가 차지한 리소스로 인해 **여유가 0**에 가까운 상태이며, Pending 된 Executor Pod들은 요구 리소스를 충족시킬 노드를 찾지 못해 대기 중입니다 (예: Node 상태에서 CPU/Mem 부족). 이러한 현상은 주로 **클러스터 용량 대비 Spark 애플리케이션의 요구 리소스가 과도**할 때 발생하며, **클러스터 오토스케일러**가 없다면 문제가 계속 지속됩니다.

### 2) **스케줄링 지연 (순차 스케줄링으로 인한 스타트업 지연)**

Spark 작업을 Kubernetes에 제출할 때 **Gang Scheduling이 없으면** 드라이버와 Executor들이 **순차적으로** 스케줄링됩니다. 즉, **드라이버 Pod가 먼저 생성 및 스케줄**되고, 그 드라이버가 동작을 시작한 후에야 **Executor Pod들을 요청**하게 됩니다[[1]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Spark%20applications%20consist%20of%20a,itself%20launches%20the%20executor%20pods). 이 순차 프로세스는 특히 **빈번한 오토스케일링 환경**에서 **작업 시작 지연**을 야기합니다. 예를 들어, **클러스터 노드가 비어있는 상황**(cold start)에서 Spark 작업을 제출하면:

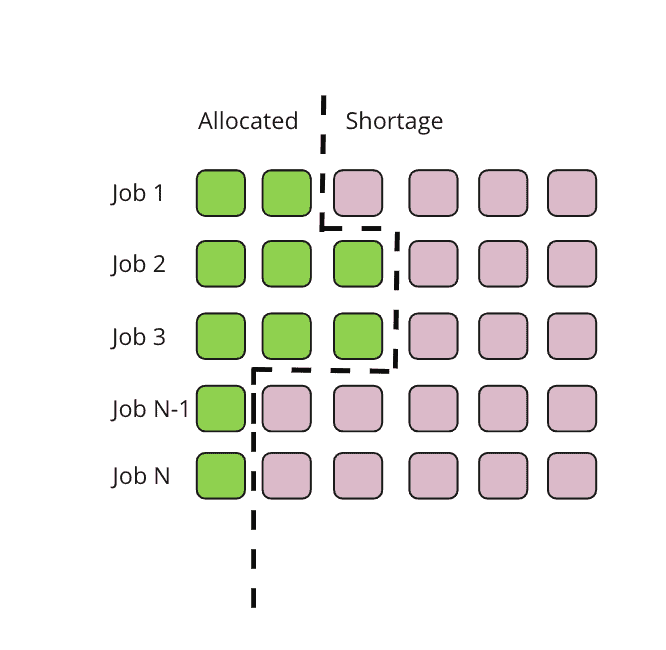
  
*Gang Scheduling 없이 Spark 작업을 제출했을 때 오토스케일링이 단계적으로 발생하여 작업 시작이 지연되는 타임라인 예시. 처음 Spark submit 후 드라이버 Pod가 생성되고(분홍색 원), 드라이버 Pod가 스케줄되어 기동되기까지 클러스터가 첫 번째 Scale-up을 수행한다(점선 화살표). 드라이버가 실행 시작 후 Executor Pod들을 요청(batch 0)하면(녹색 원과 분홍색 원으로 표시), 두 번째 Scale-up 트리거가 발생하여 부족한 노드를 추가한 뒤에야 Executor들이 할당(녹색 원들)된다. 이 두 번의 Scale-up 사이에 이미지 다운로드, 드라이버 초기화 등이 겹쳐져 전체 지연이 커지는 모습*[*[9]*](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=This%20chart%20illustrates%20the%20scenario,batches%2C%20this%20is%20even%20worse)*.*

위 타임라인에서 보이듯, **드라이버 Pod가 완전히 실행된 후에야** Executor 생성이 이루어지므로, 클러스터 **오토스케일러**가 있다면 **두 단계에 걸쳐 노드를 확장(Scale-up)**하게 됩니다[[9]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=This%20chart%20illustrates%20the%20scenario,batches%2C%20this%20is%20even%20worse). 첫 번째는 드라이버가 스케줄될 때 노드가 부족하면 발생하고, 두 번째는 Executor들을 스케줄하기 위해 다시 노드를 늘리는 단계입니다. 이 **단계 간격마다 시간 지연**이 생기는데, 드라이버가 초기화되고 Docker 이미지를 받는 시간, Spark가 Executor를 배치 요청(batch)하는 시간 등이 모두 누적되어 **전체 작업 시작이 지연**됩니다[[10]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=order%20to%20scale%20up%20the,batches%2C%20this%20is%20even%20worse). 특히 Spark가 Executor를 **여러 배치로 나누어 순차 요청**하는 경우(예: 한 번에 일부 Executor씩 만들도록 설정된 경우) 이러한 지연은 더욱 커질 수 있습니다[[10]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=order%20to%20scale%20up%20the,batches%2C%20this%20is%20even%20worse). 결과적으로, Gang Scheduling 없이 개별 Pod 단위로 스케줄링하면 **작업 시작까지의 지연(latency)**이 길어지고, **자원이 충분해도** 스케줄러의 동작 방식 때문에 **Job 실행이 빠르게 이루어지지 못하는 비효율**이 발생할 수 있습니다.

이 시나리오에서 Kubernetes 리소스 상태를 살펴보면, **드라이버 Pod는 먼저 Pending->Running을 거쳐 실행**되고, 그 사이 클러스터가 노드를 추가했다면 새로운 Node가 생성됩니다. 이후 **Executor Pod들이 순차적으로 Pending** 상태로 생겼다가 하나씩 Running이 됩니다. 오토스케일러 로그나 이벤트에는 **펜딩된 Pod를 수용하기 위해 노드를 확장**했다는 내역이 두 번에 걸쳐 나타날 수 있습니다. 즉, **첫 번째 드라이버 할당 시의 확장**, **이후 Executor 할당 시의 추가 확장**으로 분리되어 리소스 변화가 일어나므로, **전체 스케줄링 완료까지 지연 시간**이 길어지게 됩니다.

### 3) **다중 작업 간 경쟁으로 인한 스타베이션(Starvation)**

여러 개의 Spark 작업이 동시에 제출되어 실행되는 **멀티 테넌트(cluster)** 환경을 가정해보겠습니다. Kubernetes 기본 스케줄러는 **여러 작업의 Pod들을 구분하지 않고** 순서 없이 배치하기 때문에, **큰 작업**과 **작은 작업**이 섞여 있으면 작은 작업의 일부가 먼저 들어가고 큰 작업의 중요한 일부 Executor가 뒤로 밀려 **영원히 스케줄되지 못하는 현상**이 발생할 수 있습니다[[2]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Additionally%2C%20because%20the%20pods%20from,look%20at%20a%20simple%20example)[[11]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=pods%20for%20drivers%20and%20executors,before%20it%20can%20make%20progress). 즉, **자원이 약간이라도 남으면 후순위 작업의 작은 Executor가 치고 들어와**, 정작 선행 작업(대규모 작업)의 남은 Executor들은 자리를 못 얻는 식입니다. 이로 인해 **리소스가 단편화**되고 일부 **작업이 영구 대기** 상태에 빠지는데, 이를 **스타베이션(starvation)** 현상이라고 합니다.

  
*Gang Scheduling 없이 여러 Spark 작업을 동시 제출한 경우의 리소스 할당 상태 예시 도식*[*[12]*](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Resource%20segmentation%20problems%20are%20very,with%20each%20other%20badly%20like)*. 가로 행은 Job1, Job2, ... 각 Spark 작업을 의미하고, 각 작업별로 필요한 Pod(드라이버 및 다수의 Executor)을 네모로 표현했다. 녹색 칸은 이미* *할당된(Allocated)* *Pod들을, 분홍색 칸은* *할당 대기 중(Shortage)인 Pod들을 나타낸다. 이 그림은 클러스터 자원이 부족한 상황에서* *N개의 Job이 각자 일부 Pod(1~2개)만 할당받고 나머지 Pod들은 Pending된* *리소스 세분화 문제를 보여준다*[*[12]*](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Resource%20segmentation%20problems%20are%20very,with%20each%20other%20badly%20like)*. 예를 들어 Job 1과 Job 2는 각자 일부 Executor만 할당되고 나머지는 대기 중이며, Job N과 같은 후순위 작업들도 드라이버만 간신히 할당된 채 Executor 대부분이 Pending되어 있다. 이처럼* *각 작업이 필요한 모든 리소스를 얻지 못한 채 조금씩만 점유하고 있으면, Spark의 경우 드라이버만 떠 있고 Executor가 없어* *여러 Job이 모두 진행되지 못하고 교착 상태에 빠질 수 있습니다*[*[13]*](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Jobs%20from%201%20to%20N%2C,resources%20left%20for%20the%20executors)*.*

위와 같은 시나리오에서는 **클러스터 전체 자원**이 여러 Job에 **분산(fragmented)**되어 **그 어느 작업도 완전한 실행을 못하는 비효율적 상태**가 됩니다[[14]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Image). 최악의 경우, 위 그림처럼 **각 Spark 작업은 드라이버 1개만 Running이고 모든 Executor가 Pending인 상태**로 서로 물고 물리는 **데드락(deadlock)**에 빠질 수 있습니다[[13]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Jobs%20from%201%20to%20N%2C,resources%20left%20for%20the%20executors). 예를 들어 앞선 Palantir 사례에서는, 큰 작업 A의 마지막 Executor가 자원이 부족해 Pending인데 그 사이 작은 작업 B의 Executor가 남은 조각 자원을 차지하여 B는 완료되고 **A는 끝내 자원을 얻지 못하는** 일이 발생했습니다[[11]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=pods%20for%20drivers%20and%20executors,before%20it%20can%20make%20progress). 이런 일이 반복되면 **대규모 Job은 계속 기아(starvation)** 상태에 빠져 **성능 저하**는 물론 **영원히 완료되지 못하는 실패 상황**이 벌어질 수 있습니다[[2]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Additionally%2C%20because%20the%20pods%20from,look%20at%20a%20simple%20example). 기본 스케줄러에는 작업 간 **공정성(fairness)**이나 **동시 착수 제한** 개념이 없기 때문에, 리소스 부족 시 작은 Pod들이 간헐적으로라도 계속 진입하면 큰 작업은 차례를 영영 얻지 못하는 것입니다.

이 경우 Kubernetes 상태로 보면, **여러 서로 다른 SparkApplication의 드라이버와 Executor Pod들이 뒤섞여** 있습니다. 일부 드라이버와 몇 Executor는 Running이지만 **다수의 Executor Pod들은 Pending**이며, kubectl get pods로 보면 각 작업당 하나-두개만 Running, 나머지는 Pending으로 잔뜩 대기한 모습입니다. Node들의 리소스는 이미 **여러 작업이 조금씩 차지하여 포화**된 상태라, 추가 Pending된 Pod들을 수용 못합니다. 즉 **모든 Job이 각자 필요한 리소스를 다 받지 못한 채 조금씩만 점유**하고 있어 **누구도 끝나지 못하는 비효율**이 발생합니다.

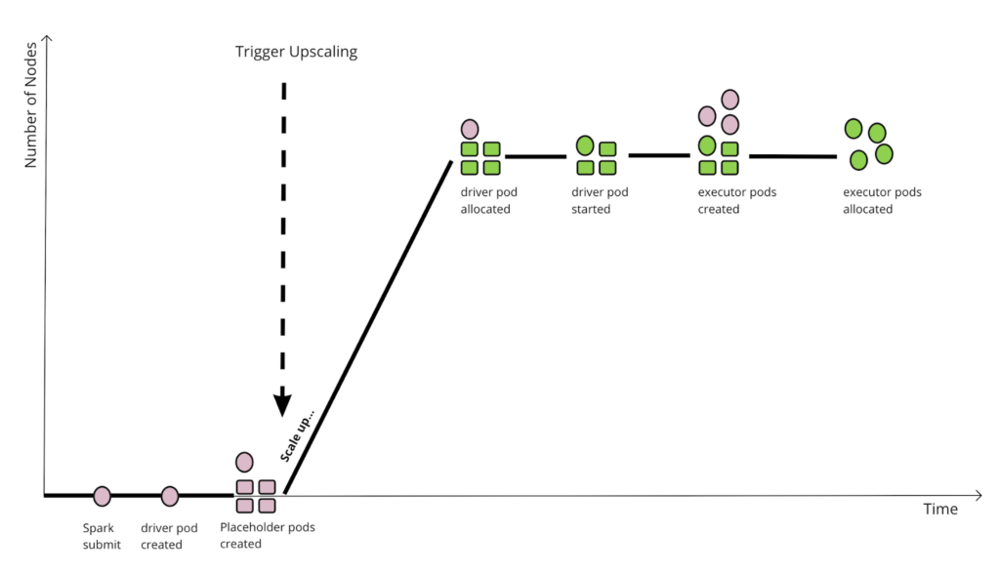
### 4) **작업 실패 또는 중단 (Job Failure/Hang)**

위에서 언급한 상황들은 결국 Spark **Job의 실패나 중단**으로 이어질 수 있습니다. **자원 부족으로 Executor를 못 얻는 경우** 드라이버가 장시간 대기하면 사용자가 수동으로 해당 Job을 취소하거나, 상위에서 **타임아웃 정책**을 두어 일정 시간 후 Job을 **강제 종료**시킬 수 있습니다. 이러한 경우 사용자의 입장에서는 해당 Spark 작업이 **실패**한 것이나 다름없습니다. 특히 **여러 작업 간 경쟁으로 인한 스타베이션** 상황에서는, 리소스가 조금씩 분산되어 큰 작업이 **영구히 대기**하면 결국 그 작업은 **완료될 수 없으므로 실패**하게 됩니다[[2]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Additionally%2C%20because%20the%20pods%20from,look%20at%20a%20simple%20example). 또는 운 좋게 나중에라도 자원이 풀려 실행되더라도, **예측 불가능한 지연**으로 인해 SLA를 맞추지 못해 실패 판정될 수 있습니다. 또한 리소스 부족 상황에서 간신히 실행되던 Executor들이 **노드 메모리 부족(OOM)** 등으로 반복적으로 죽고 재시작되다가 결국 작업이 오류로 중단되는 케이스도 있습니다. 결국 Gang Scheduling 없이 발생하는 앞선 문제들은 심각한 경우 **Spark Application의 실패**로 귀결됩니다. Kubernetes 측면에서 보면, 이러한 실패 시나리오는 **드라이버 Pod가 최종적으로 Error 상태나 CrashLoopBackOff로 끝나거나** (예: 자원 부족으로 OOM 발생 시) **사용자에 의해 Delete**되는 식으로 나타납니다. 요약하면, **Gang 스케줄링 부재로 인한 자원 대기와 불공정 스케줄링**은 **Spark 작업의 안정성에 악영향**을 미쳐 **작업이 끝내 완료되지 못하는 상황**까지 초래할 수 있습니다.

## Gang Scheduling 적용 시 각 문제 상황의 해결 및 완화

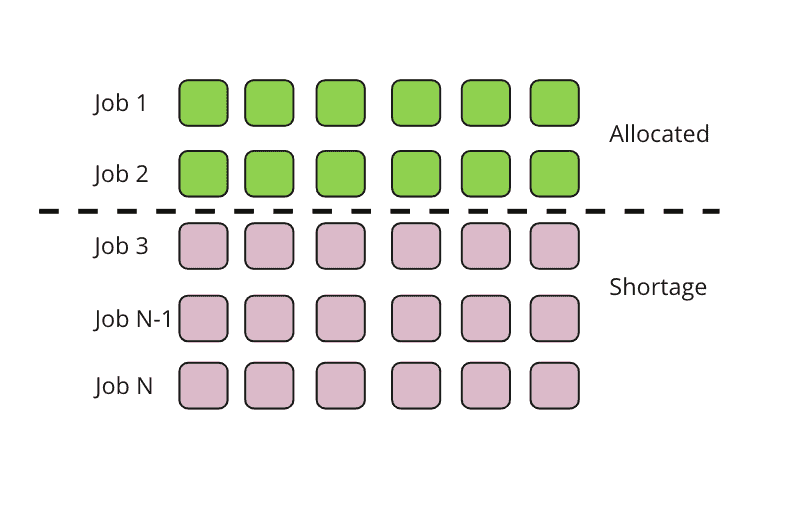
Gang Scheduling을 도입하면 앞서 언급된 문제들이 상당 부분 **해결**되거나 **완화**됩니다. Gang 스케줄링 하에서는 **관련된 모든 Pod들을 하나의 작업 단위로 스케줄링**하기 때문에, **드라이버와 Executor들이 모두 동시에 자원을 확보**하거나 아니면 모두 대기하게 됩니다[[4]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Gang%20scheduling,application). 이를 구현하는 방안에 따라 약간씩 차이가 있지만(예: Volcano의 PodGroup, YuniKorn의 placeholder 등), 공통적으로 **"일부만 스케줄되는" 상황을 막음**으로써 발생했던 부작용들을 해결합니다[[15]](https://groups.google.com/g/kubernetes-sig-scheduling/c/OAriEmhw448#:~:text=3.%20For%20volcano%2Fkube,created%20which%20minMember%20is%201)[[13]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Jobs%20from%201%20to%20N%2C,resources%20left%20for%20the%20executors). 각각의 문제 시나리오가 Gang Scheduling 하에서는 어떻게 달라지는지 살펴보면:

* **자원 부족으로 인한 Executor Pending 문제 해소:** Gang Scheduling 하에서는 **필요한 모든 Pod의 리소스가 한꺼번에 확보될 수 있을 때까지 작업을 시작하지 않습니다**[[16]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=processes%20to%20run%20simultaneously%20on,until%20it%20reaches%20this%20state). 예를 들어 Volcano 스케줄러는 Spark 작업 제출 시 **PodGroup**을 만들어 **minMember** (최소 필요한 Pod 수)나 **minResources**를 지정함으로써, **드라이버가 Executor들 몫의 자원까지 고려하여** 스케줄되도록 합니다[[15]](https://groups.google.com/g/kubernetes-sig-scheduling/c/OAriEmhw448#:~:text=3.%20For%20volcano%2Fkube,created%20which%20minMember%20is%201). 이 뜻은 **드라이버 단독으로 클러스터의 남은 자원을 몽땅 차지해서 Executor가 못 뜨는 사태(데드락)**를 방지하는 것입니다[[15]](https://groups.google.com/g/kubernetes-sig-scheduling/c/OAriEmhw448#:~:text=3.%20For%20volcano%2Fkube,created%20which%20minMember%20is%201). 즉 Gang Scheduling이 켜지면, **자원이 부족할 경우 아예 드라이버부터 Pending 상태로 대기**하게 되고 (혹은 YuniKorn의 경우 placeholder Pod들만 만들어 자리를 예약하고 실제 작업은 보류), 클러스터에 충분한 자원이 모였을 때 **드라이버와 Executor들을 함께 시작**합니다. 그 결과 Executor가 **Pending으로 무기한 대기하는 일이 줄어들고**, 불충분한 자원으로 억지로 시작했다가 지연되는 대신 **애플리케이션이 아예 대기열에서 대기**하기 때문에 오히려 **전체적인 안정성**이 높아집니다.
* **스케줄링 지연 개선:** Gang Scheduling (특히 **YuniKorn**과 같은 스케줄러)의 한 가지 장점은 **스케줄링 단계를 일원화**하여 **오토스케일링 지연을 줄여주는 것**입니다. YuniKorn은 Spark 작업의 gang 스케줄링을 위해 **실제 Pod 대신 placeholder Pod**들을 미리 생성하여 **필요 자원을 한꺼번에 예약**하는 전략을 씁니다[[17]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Prepare%3A%20in%20this%20phase%2C%20YuniKorn,constraints%20defined%20in%20this%20group)[[18]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=2,all%20the%20real%20pods%20are). 이렇게 하면 클러스터 **Autoscaler가 드라이버와 Executor 전체 요구를 한 번에 인지**하여 **즉시 충분한 노드를 추가**하게 됩니다[[19]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=it%20starts%20to%C2%A0reserve%C2%A0required%20resources%20for,the%202%20jobs%20are%20running)[[20]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Instead%20of%20waiting%20for%20Spark,jobs%20submitted%20to%20the%20cluster). 아래 타임라인은 Gang Scheduling (YuniKorn) 적용 시 어떻게 변화하는지 보여줍니다.

  
*Gang Scheduling(예: YuniKorn의 placeholder 활용)이 적용된 경우 Spark 작업 제출부터 실행까지의 타임라인 변화*[*[21]*](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Image)*. placeholder Pod들을 통해* *드라이버가 스케줄되기 전에 필요한 Executor 수만큼 자원 예약 및 Scale-up이 선행됩니다. 점선 화살표로 표시된 부분에서, 드라이버 Pod 생성과 동시에 여러 placeholder Pod(분홍색 사각형들)가 만들어져* *즉시 대규모 Scale-up을 트리거하며, 드라이버 및 Executor에 필요한 노드들이 한 번에 추가됩니다. 이후 드라이버와 Executor Pod들이 모두 할당(녹색으로 표시)되어 작업이 시작됩니다. 이 과정에서는* *기존처럼 두 단계로 나눠 노드를 늘릴 필요 없이 한 번에 필요한 규모로 클러스터를 확장하므로* *전체 스케줄링 소요 시간을 크게 단축할 수 있습니다*[*[20]*](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Instead%20of%20waiting%20for%20Spark,jobs%20submitted%20to%20the%20cluster)*.*

위와 같이 Gang Scheduler를 사용하면, **드라이버-Executor 스케줄링이 한 싸이클에 이루어지므로 불필요한 대기 감소** 및 **오토스케일링 효율 개선** 효과가 있습니다. 그 결과 Spark 작업의 **초기 시작 지연 시간이 줄어들고**, 자원을 확보하는데 걸리는 시간이 예측 가능해집니다.

* **스타베이션(자원 기아) 방지 및 공정성 향상:** Gang Scheduling의 핵심 이점 중 하나는 **리소스 세그멘테이션 문제(조각 할당)를 해소**하는 것입니다[[22]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=). 앞서 여러 Job이 부분 할당만 받아 교착 상태에 빠졌던 예시를 Gang Scheduling으로 해결하면 다음과 같습니다:

  
*Gang Scheduling을 적용하여 여러 Spark 작업을 스케줄링한 경우의 자원 할당 예시*[*[23]*](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=With%20Gang%20Scheduling%2C%20the%20scheduler,scenario%20will%20look%20like%20this)*. 클러스터의 제한된 자원 내에서* *Job1과 Job2만 필요한 모든 Pod을 할당받아(초록색)* *실행 중이고, 나머지 Job3 이후의 작업들은* *아예 리소스가 날 때까지 Pending으로 대기합니다 (분홍색은 모두 대기 중). 이렇게 하면 각 작업은 최소한* *자신의 실행에 필요한 전체 자원을 확보한 후 시작되므로, 일부만 실행되어 막히는 일이 없습니다*[*[23]*](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=With%20Gang%20Scheduling%2C%20the%20scheduler,scenario%20will%20look%20like%20this)*. 결과적으로* *작업 1, 2가 완료된 후에야 다음 작업들이 순서대로 자원을 받아 실행되므로,* *작업 시간의 예측 가능성이 높아지고* *데드락 없이* *진행됩니다.*

위 그림처럼 Gang Scheduling 환경에서는 **항상 전체 작업 단위로 자원을 할당**하므로, **한 번에 처리될 수 있는 작업 수가 제한**됩니다 (예시에서는 Job1과 Job2만 실행, 나머지 대기). 그러나 이는 오히려 **큰 작업이 굶주리지 않고** 차례를 확보할 수 있게 해주며, **Deadlock이나 모든 작업 정지 상황을 방지**합니다[[24]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Image). 추가로, Palantir가 구현한 스케줄러처럼 **Gang 스케줄링에 작업 제출 순서(FIFO)나 우선순위 개념**을 더하면, 작은 작업들이 계속 새치기하여 큰 작업이 아예 못 돌게 되는 일을 막을 수 있습니다[[25]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=First,was%20submitted%20before%20driver%20B). 결국 **Gang Scheduling은 멀티 테넌트 환경에서 공정성을 높이고, 대규모 잡의 기아 상태를 예방**하여 **리소스 활용의 효율과 작업 완료율**을 높입니다.

* **작업 실패/지연 해결:** Gang Scheduling 적용 시에는 앞선 문제들이 완화됨에 따라 **작업 실패율이 낮아집니다**. 자원 부족 시 애플리케이션을 부분적으로 시작하지 않고 대기시키므로 오히려 **안전하게 대기열에서 보류**되었다가 충분한 자원에서 실행되기 때문에 **중간에 무기한 대기하거나 사용자가 임의로 죽이는 상황이 줄어듭니다**. 또한 한 번에 한 작업(또는 제한된 수의 작업)만 전체 리소스를 가지고 수행되므로, **작업 간 간섭으로 인한 실패** (예: 큰 작업이 끝내 실행 못되는 문제)가 사라집니다. 결과적으로 **Spark 작업의 완료 가능성이 높아지고 안정성**이 향상됩니다. 만약 특정 작업이 매우 큰 리소스를 요구한다면 Gang Scheduling 하에서는 아예 **클러스터가 그만한 리소스를 확보할 때까지 Start를 보류**하므로, 불필요하게 드라이버만 실행했다가 실패하는 일을 피할 수 있습니다. 요약하면, Gang Scheduling은 **애플리케이션을 “모 아니면 도”로 스케줄링**함으로써 **부분 할당으로 인한 실패**를 없애고, **예측 가능한 성공/실패 패턴**을 제공하게 됩니다.

## Volcano나 YuniKorn Gang Scheduler 사용 시와 미사용 시의 차이

마지막으로, **전용 Gang Scheduler(예: Volcano, YuniKorn)를 사용한 경우와 기본 스케줄러 사용 시의 차이점**을 개념적으로 정리합니다:

* **기본 Kubernetes 스케줄러 (미사용 시):** Pod 단위로 **개별 스케줄링**을 수행합니다. Spark 작업의 드라이버와 Executor들은 서로 관계없이 스케줄되므로, 위에서 본 것처럼 **드라이버만 실행되고 Executor들은 Pending**이 된다거나[[15]](https://groups.google.com/g/kubernetes-sig-scheduling/c/OAriEmhw448#:~:text=3.%20For%20volcano%2Fkube,created%20which%20minMember%20is%201), 여러 작업이 **뒤섞여 일부씩만 자원을 점유**하는 현상이 생길 수 있습니다[[14]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Image). 기본 스케줄러는 **Gang 개념이 없기 때문에** Spark 같은 다중-Pod 작업에 대해 **전체적인 요구 자원을 고려하지 않으며**, **순간 가용 자원에 따라** 하나씩 Pod을 넣어주는 방식을 취합니다. 그 결과 **스케줄링 공정성이나 동기화 측면에서 부족함**이 있어, 배치 작업(batch job)의 효율이 떨어질 수 있습니다.
* **Volcano 사용 시:** Volcano는 Kubernetes용 배치 스케줄러로, **PodGroup API**를 통해 Gang Scheduling을 구현합니다. Spark Operator와 Volcano를 연동하면, SparkApplication 리소스 생성 시 자동으로 PodGroup 객체가 생기고 해당 그룹에 **minMember**(최소 실행되어야 할 Pod 수)와 **minResources**(필요 자원 총합) 등을 설정할 수 있습니다[[15]](https://groups.google.com/g/kubernetes-sig-scheduling/c/OAriEmhw448#:~:text=3.%20For%20volcano%2Fkube,created%20which%20minMember%20is%201). Volcano 스케줄러는 **PodGroup 단위로 스케줄링 결정**을 내리기 때문에, **그룹 내 모든 Pod을 한꺼번에 넣을 수 있을 때까지 대기**합니다. 예를 들어 Spark의 경우 PodGroup의 minMember를 드라이버+모든 Executor 개수로 두고, 드라이버 Pod에 schedulerName: volcano와 그룹 이름을 붙여 제출하면 Volcano가 **모든 Executor를 포함한 전체 리소스 요구를 파악**하여, **한번에 배치가 가능할 때 드라이버부터 실행**하게 됩니다[[15]](https://groups.google.com/g/kubernetes-sig-scheduling/c/OAriEmhw448#:~:text=3.%20For%20volcano%2Fkube,created%20which%20minMember%20is%201). 또한 Volcano는 **배치 작업에 특화된 기능**(예: FAIR 큐, 우선순위, preemption 등)도 제공하여 다중 작업 환경에서 **스케줄링 최적화**를 도모합니다. 핵심적으로 Volcano를 사용하면 **드라이버가 모든 자원을 소비해 Executor가 못 뜨는 사태(Deadlock)를 방지**하고[[15]](https://groups.google.com/g/kubernetes-sig-scheduling/c/OAriEmhw448#:~:text=3.%20For%20volcano%2Fkube,created%20which%20minMember%20is%201), **Pod들을 개별이 아닌 그룹으로 다뤄서** Spark 작업의 **동시 실행/대기**를 효과적으로 관리할 수 있습니다.
* **YuniKorn 사용 시:** YuniKorn 역시 Kubernetes용 범용 스케줄러로, Apache Hadoop/YARN의 리소스 스케줄링 개념을 Kubernetes에 도입한 솔루션입니다. YuniKorn의 Gang Scheduling은 **TaskGroup** 정의를 통해 이루어지며, Spark의 경우 **플레이스홀더 Pod** 방식을 사용합니다[[17]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Prepare%3A%20in%20this%20phase%2C%20YuniKorn,constraints%20defined%20in%20this%20group)[[18]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=2,all%20the%20real%20pods%20are). 작업 제출 시 드라이버와 Executor 스펙에 기반하여 **동일한 수의 placeholder Pod들을 미리 생성**하고, 이것들을 **가벼운 “pause” 컨테이너**로 띄워 **노드 자리를 선점**하게 합니다[[26]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Prepare%3A%20in%20this%20phase%2C%20YuniKorn,constraints%20defined%20in%20this%20group). **모든 placeholder들이 할당될 수 있을 때까지** 대기한 후, 한꺼번에 **실제 Spark Pod들과 swap**하여 실행을 시작합니다[[27]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Resource%20Reservation%3A%20in%20this%20phase%2C,a%20spot%20on%20a%20node). 이 방식은 **리소스 예약**과 **동시 시작**을 보장할 뿐 아니라, **클러스터 오토스케일러와 연계**되어 매우 효율적으로 동작합니다[[19]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=it%20starts%20to%C2%A0reserve%C2%A0required%20resources%20for,the%202%20jobs%20are%20running)[[20]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Instead%20of%20waiting%20for%20Spark,jobs%20submitted%20to%20the%20cluster). 앞서 살펴본 대로, placeholder들이 **일찌감치 전체 자원 요구를 시스템에 알려주므로** 클러스터가 **한 번에 확장**하고, Spark 작업들도 **동시에 시작**되어 **대기 시간과 자원 낭비가 최소화**됩니다[[20]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Instead%20of%20waiting%20for%20Spark,jobs%20submitted%20to%20the%20cluster). YuniKorn은 또한 **큐 기반의 자원 공유와 우선순위** 기능을 제공하여 여러 작업 간 **Fairness**도 향상시킵니다[[28]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=With%20Gang%20Scheduling%2C%20YuniKorn%20%E2%80%9Cunderstands%E2%80%9D,launched%20concurrently%20in%20the%20queue)[[23]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=With%20Gang%20Scheduling%2C%20the%20scheduler,scenario%20will%20look%20like%20this).

요약하면, **Volcano나 YuniKorn 같은 Gang Scheduler를 사용**하면 **Spark 작업의 스케줄링 단위가 Pod 하나에서 작업 전체로 올라가기 때문에** 발생했던 문제들(자원 부족시 일부만 실행, 지연, 스타베이션 등)이 **근본적으로 해결**됩니다. 반면 Gang Scheduler를 **사용하지 않는 경우**, 이러한 조율 없이 **각 Pod들이 개별적으로 경쟁**하므로 앞서 설명한 여러 **실무적인 어려움**이 발생할 수 있습니다. 실제 현업 환경에서, Gang Scheduling을 도입하면 **Spark on Kubernetes의 작업 완료 시간 예측이 용이**해지고 **클러스터 자원 활용의 효율이 높아지며**, **대형 잡에 대한 신뢰성**이 향상되는 효과가 있습니다[[24]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Image)[[25]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=First,was%20submitted%20before%20driver%20B). 따라서 Kubernetes 상에서 Spark Operator를 운용하는 실무자라면, **Volcano나 YuniKorn과 같은 Gang Scheduler의 도입을 검토**하여 위에서 논의된 문제 상황을 예방하고 안정적인 배치 작업 스케줄링을 구현하는 것이 바람직합니다.

**참고 문헌:** Spark Operator 및 Kubernetes에서의 Spark 스케줄링 문제와 Gang Scheduling의 필요성에 대해서 Cloudera와 Palantir 등의 기술 블로그에서 다룬 바 있으며[[29]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Jobs%20from%201%20to%20N%2C,resources%20left%20for%20the%20executors)[[30]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=this%20design%20decision%20has%20a,leading%20to%20unpredictable%20performance%20degradation), Volcano와 YuniKorn 프로젝트 문서에서도 Spark와의 통합에 대한 가이드와 사례를 제공합니다. 이러한 자료들을 통해 보다 심층적인 원리와 설정 방법을 학습할 수 있습니다.

[[1]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Spark%20applications%20consist%20of%20a,itself%20launches%20the%20executor%20pods) [[2]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Additionally%2C%20because%20the%20pods%20from,look%20at%20a%20simple%20example) [[4]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=Gang%20scheduling,application) [[11]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=pods%20for%20drivers%20and%20executors,before%20it%20can%20make%20progress) [[25]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=First,was%20submitted%20before%20driver%20B) [[30]](https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c#:~:text=this%20design%20decision%20has%20a,leading%20to%20unpredictable%20performance%20degradation) Spark Scheduling in Kubernetes. Introducing our open source… | by Palantir | Palantir Blog

<https://blog.palantir.com/spark-scheduling-in-kubernetes-4976333235f3?gi=a63b2c087a0c>

[[3]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=According%20to%C2%A0wikipedia%2C%20Gang%20Scheduling%20refers,until%20it%20reaches%20this%20state) [[5]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Gang%20scheduling%20is%20very%20useful,for%20computing%20engines%20like%20Spark) [[9]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=This%20chart%20illustrates%20the%20scenario,batches%2C%20this%20is%20even%20worse) [[10]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=order%20to%20scale%20up%20the,batches%2C%20this%20is%20even%20worse) [[12]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Resource%20segmentation%20problems%20are%20very,with%20each%20other%20badly%20like) [[13]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Jobs%20from%201%20to%20N%2C,resources%20left%20for%20the%20executors) [[14]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Image) [[16]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=processes%20to%20run%20simultaneously%20on,until%20it%20reaches%20this%20state) [[17]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Prepare%3A%20in%20this%20phase%2C%20YuniKorn,constraints%20defined%20in%20this%20group) [[18]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=2,all%20the%20real%20pods%20are) [[19]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=it%20starts%20to%C2%A0reserve%C2%A0required%20resources%20for,the%202%20jobs%20are%20running) [[20]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Instead%20of%20waiting%20for%20Spark,jobs%20submitted%20to%20the%20cluster) [[21]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Image) [[22]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=) [[23]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=With%20Gang%20Scheduling%2C%20the%20scheduler,scenario%20will%20look%20like%20this) [[24]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Image) [[26]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Prepare%3A%20in%20this%20phase%2C%20YuniKorn,constraints%20defined%20in%20this%20group) [[27]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Resource%20Reservation%3A%20in%20this%20phase%2C,a%20spot%20on%20a%20node) [[28]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=With%20Gang%20Scheduling%2C%20YuniKorn%20%E2%80%9Cunderstands%E2%80%9D,launched%20concurrently%20in%20the%20queue) [[29]](https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html#:~:text=Jobs%20from%201%20to%20N%2C,resources%20left%20for%20the%20executors) Spark on Kubernetes – Gang Scheduling with YuniKorn | Blog | Cloudera

<https://www.cloudera.com/blog/technical/spark-on-kubernetes-gang-scheduling-with-yunikorn.html>

[[6]](https://stackoverflow.com/questions/65683383/spark-executors-pods-in-pending-state-on-kubernetes-deployment#:~:text=%24%20kubectl%20describe%20pod%2Fspark,Insufficient%20cpu%2C%201%20Insufficient%20memory) [[7]](https://stackoverflow.com/questions/65683383/spark-executors-pods-in-pending-state-on-kubernetes-deployment#:~:text=%24%20kubectl%20logs%20pod%2Fspark,workers%20are%20registered%20and%20have) [[8]](https://stackoverflow.com/questions/65683383/spark-executors-pods-in-pending-state-on-kubernetes-deployment#:~:text=Now%20my%20question%20here%20is,were%20no%20resources%20in%20cluster) Spark Executors PODS in Pending State on Kubernetes Deployment - Stack Overflow

<https://stackoverflow.com/questions/65683383/spark-executors-pods-in-pending-state-on-kubernetes-deployment>

[[15]](https://groups.google.com/g/kubernetes-sig-scheduling/c/OAriEmhw448#:~:text=3.%20For%20volcano%2Fkube,created%20which%20minMember%20is%201) Volcano scheduler gang-scheduling not working

<https://groups.google.com/g/kubernetes-sig-scheduling/c/OAriEmhw448>