쿠버네티스 커스텀 스케줄러 선택적 적용 가이드

온프레미스 200노드 클러스터에서 같은 네임스페이스 내 Spark는 커스텀 스케줄러로, Trino는 기본 스케줄러로 운영하는 것이 **완전히 가능**합니다. Packt Publishing↗ 세 가지 주요 커스텀 스케줄러(Yunikorn, Volcano, Kueue) 모두 `schedulerName` 필드 또는 레이블 기반으로 선택적 스케줄링을 지원하며, InfraCloud↗ KEDA 자동 스케일링과도 호환됩니다. 핵심은 적절한 스케줄러 선택과 리소스 격리 전략입니다.

선택적 스케줄러 적용의 핵심 메커니즘

쿠버네티스의 **모든 Pod는 `spec.schedulerName` 필드로 스케줄러를 지정**합니다. 이 필드를 명시하지 않으면 `default-scheduler`가 사용되며, 각 스케줄러는 자신에게 할당된 Pod만 처리합니다. Kubernetes +3↗ 같은 네임스페이스 내에서도 워크로드별로 다른 스케줄러를 사용할 수 있어, A팀의 Spark 작업은 Yunikorn/Volcano를 사용하고 같은 네임스페이스의 Trino는 기본 스케줄러를 사용하는 구성이 가능합니다. GitHub +2↗

세 스케줄러의 선택적 적용 방식은 근본적으로 다릅니다. **Yunikorn**은 admission controller를 통해 특정 네임스페이스의 모든 Pod를 자동으로 가져갈 수 있지만, bypass 필터나 admission controller 비활성화로 선택적 적용이 가능합니다. Apache +2↗ **Volcano**는 PodGroup CRD를 생성한 Pod만 처리하며, 나머지는 기본 스케줄러로 자동 넘어갑니다. Medium +5↗ **Kueue**는 `kueue.x-k8s.io/queue-name` 레이블이 있는 워크로드만 admission 제어하고, 레이블 없는 워크로드는 완전히 무시합니다. Toolify +3↗

Yunikorn: 멀티테넌트 환경에 최적화된 스케줄러

Apache Yunikorn은 기본 쿠버네티스 스케줄러를 **대체**하는 방식으로 동작하며, InfraCloud↗ Alibaba와 Apple의 프로덕션 환경에서 검증되었습니다. Medium +7↗ 2024-2025년 현재 v1.5.0이 안정 버전이며, 계층적 큐 구조와 공정성 정책으로 멀티테넌트 클러스터에 강점을 보입니다. Cloudera Blog↗

선택적 스케줄링을 위해서는 admission controller의 **네임스페이스 필터링 설정**이 핵심입니다. `processNamespaces` 옵션으로 Spark 네임스페이스만 처리하거나, `bypassNamespaces`로 Trino 네임스페이스를 제외할 수 있습니다. Apache↗ 더욱 유연한 방식은 admission controller를 완전히 비활성화(`embedAdmissionController: false`)하고 Pod 스펙에서 명시적으로 `schedulerName: yunikorn`을 지정하는 것입니다. GitHub↗

Spark 워크로드에 Gang 스케줄링을 적용하려면 **task group 어노테이션**을 설정해야 합니다. Driver와 Executor를 위한 task group을 정의하고, `minMember`와 `minResource`로 최소 요구사항을 명시합니다. Apache↗apache↗ Yunikorn은 placeholder pod를 생성해 리소스를 선점한 후, 모든 리소스가 확보되면 실제 Pod를 시작합니다. Cloudera Blog↗Apache↗ Gang 스케줄링을 사용하는 큐는 반드시 **FIFO sorting policy**를 사용해야 하며, FAIR 정책은 여러 앱에 리소스를 분산 할당해 gang 스케줄링 보장을 깨뜨립니다. Apache↗apache↗

리소스 관리는 **계층적 큐 구조**로 구현합니다. InfraCloud↗ `root.spark.production`과 `root.spark.development`로 큐를 분리하고, 각 큐에 guaranteed 및 max 리소스를 설정합니다. Medium↗ Guaranteed 리소스는 공정성 계산과 선점 보호에 사용되며, max 리소스는 절대 상한선입니다. infracloud +2↗ 5개 팀 환경에서는 팀별 큐를 생성하고 `priority.offset`으로 우선순위를 조정할 수 있습니다. 예를 들어 Trino 큐에 `priority.offset: 1000`을 설정하면 같은 PriorityClass라도 Trino가 먼저 스케줄링됩니다. Apache↗

Yunikorn과 기본 스케줄러가 **같은 네임스페이스에서 공존**할 때, 두 스케줄러는 Kubernetes API를 통해 클러스터 상태를 보지만 직접 조율하지 않습니다. 리소스 경합을 방지하려면 **노드 레이블링과 NodeSelector**로 물리적 분리가 권장됩니다. Stack Overflow↗Google Groups↗ Spark 노드에 `workload-type=spark` 레이블을 붙이고 taint를 설정하면, Spark Pod만 해당 노드에 스케줄되어 Trino와 리소스 충돌이 원천 차단됩니다. GitHub +2↗

Volcano: AI/ML Gang 스케줄링의 표준

Volcano는 CNCF incubating 프로젝트로, 기본 스케줄러와 **공존**하며 동작합니다. Toolify +5↗ 2025년 3월 출시된 v1.11.0은 네트워크 토폴로지 인식 스케줄링과 동적 리소스 할당을 지원하며, Amazon과 Tencent의 프로덕션 환경에서 하루 30만 개 Pod를 처리합니다. Altoros↗

Volcano는 **PodGroup CRD**를 통해 gang 스케줄링을 구현합니다. volcano +2↗ Spark Operator와 통합 시 `batchScheduler: volcano`를 SparkApplication spec에 명시하면, Operator webhook가 자동으로 모든 Pod에 `schedulerName: volcano`와 `scheduling.k8s.io/group-name` 어노테이션을 추가합니다. infracloud +3↗ PodGroup의 `minMember`는 최소 Pod 수(Spark는 보통 1로 설정해 driver 우선 시작), `minResources`는 전체 리소스 요구량을 명시합니다. volcano +2↗ Volcano는 모든 리소스가 확보될 때까지 대기하며, 부분 스케줄링을 방지합니다. CNCF↗

Queue 설정에서 **Capacity 플러그인**(v1.9.0+)이 Proportion 플러그인보다 권장됩니다. `guarantee`는 보장 리소스, `deserverd`는 공정 분배량, `capability`는 최대 한도를 정의합니다. `reclaimable: true`로 설정하면 다른 큐가 유휴 리소스를 차용할 수 있습니다. volcano↗Volcano↗ 계층적 큐도 v1.10+에서 지원되어 `parent: production` 형태로 상위 큐를 지정할 수 있습니다.

선택적 스케줄링 구현은 Yunikorn보다 명확합니다. Trino Deployment는 `schedulerName` 필드를 아예 지정하지 않거나 `default-scheduler`로 명시하면 Volcano를 완전히 우회합니다. Volcano admission webhook는 `schedulerName: volcano`가 있는 Pod만 처리하므로, 같은 네임스페이스의 Trino는 영향을 받지 않습니다. 다만 Volcano 공식 문서는 "클러스터의 모든 워크로드를 Volcano로 스케줄하라"고 권장하며, 혼용 시 **노드 분리 전략**이 필수입니다. T-systems↗

Spark 네이티브 모드(spark-submit)에서 Volcano를 사용하려면 Spark 3.3.0+ 필요합니다. `--conf spark.kubernetes.scheduler.name=volcano`와 함께 `spark.kubernetes.scheduler.volcano.podGroupTemplateFile`로 PodGroup 템플릿을 지정합니다. GitHub +3↗ `org.apache.spark.deploy.k8s.features.VolcanoFeatureStep` 클래스를 driver와 executor의 featureSteps에 추가해야 Volcano 통합이 활성화됩니다. Medium +2↗

Kueue: 쿠버네티스 네이티브 Job 큐잉

Kueue는 kubernetes-sigs 프로젝트로 v0.14.2(2024년 12월)가 최신 버전이며, **v1beta1 API 상태**로 프로덕션 사용이 가능합니다. GitHub↗Medium↗ Yunikorn/Volcano와 달리 기본 스케줄러를 대체하지 않고, Job admission 제어만 수행합니다. InfraCloud↗DEV Community↗ CoreWeave, Google Cloud, Red Hat OpenShift에서 프로덕션 사용 중입니다.

Kueue의 선택적 적용은 **레이블 기반**입니다. `kueue.x-k8s.io/queue-name` 레이블이 있는 워크로드만 Kueue가 관리하고, 레이블이 없으면 완전히 무시됩니다. Kueue↗InfraCloud↗ 설정에서 `manageJobsWithoutQueueName: false`(기본값)로 하면 명시적으로 레이블링한 워크로드만 큐잉되고, Trino처럼 레이블 없는 워크로드는 바로 기본 스케줄러로 갑니다. InfraCloud↗DEV Community↗ 이는 Yunikorn/Volcano보다 **혼용이 안전**합니다.

Gang 스케줄링은 Kueue의 **기본 동작**입니다. 모든 워크로드는 all-or-nothing 방식으로 리소스를 할당받으며, Job의 모든 Pod가 스케줄 가능할 때만 `spec.suspend: false`로 전환됩니다. Ray↗Google Cloud↗ `waitForPodsReady` 기능을 활성화하면 모든 Pod가 Ready 상태가 될 때까지 추가 대기하며, timeout 시 exponential backoff로 재큐잉됩니다. Spark처럼 driver와 executor 간 통신이 필요한 워크로드에 이상적입니다.

리소스 관리는 **ResourceFlavor → ClusterQueue → LocalQueue** 계층 구조입니다. ResourceFlavor는 노드 레이블과 toleration으로 하드웨어 타입을 정의하고(예: GPU A100, 고메모리 CPU), ClusterQueue는 quota와 정책을 설정하며, LocalQueue는 네임스페이스 레벨 인터페이스입니다. `nominalQuota`는 보장 리소스, `borrowingLimit`은 cohort 내 차용 가능량을 정의합니다. k8s↗ Preemption 정책으로 `reclaimWithinCohort: LowerPriority`를 설정하면 우선순위 기반 선점이 가능합니다.

Kueue의 **주요 제약**은 Spark Operator와의 네이티브 통합이 없다는 점입니다. SparkApplication CRD에 레이블을 추가해도 Spark Operator가 자체 lifecycle을 관리해 Kueue의 suspend 메커니즘이 작동하지 않을 수 있습니다.

Hopsworks↗ 대안으로 Kubernetes Job으로 spark-submit을 감싸거나, Ray 같은 Kueue 통합이 완전한 프레임워크를 사용하는 것이 안정적입니다. 반면 Volcano와 Yunikorn은 Spark Operator 통합이 성숙했습니다.

Spark Operator와 커스텀 스케줄러 통합

Kubeflow Spark Operator(v2.1.0+)는 `batchScheduler` 필드로 Yunikorn과 Volcano를 지원합니다. SparkApplication에서 `batchScheduler: volcano`를 설정하면 webhook가 자동으로 driver와 executor Pod에 `schedulerName: volcano`, gang 스케줄링 어노테이션, queue 레이블을 추가합니다. apache +3↗ Yunikorn의 경우 `batchScheduler: yunikorn`과 함께 `batchSchedulerOptions.queue`로 큐를 지정하면 됩니다.

Spark 네이티브 모드에서는 **Pod 템플릿이 유일한 방법**입니다. `spark.kubernetes.scheduler.name` 설정으로는 schedulerName을 지정할 수 없으며, `--conf spark.kubernetes.driver.podTemplateFile`과 `--conf spark.kubernetes.executor.podTemplateFile`로 YAML 템플릿을 제공해야 합니다. GitHub↗GitHub↗ 템플릿에 `schedulerName: volcano`, nodeSelector, toleration 등을 명시합니다. Volcano의 경우 추가로 `--conf spark.kubernetes.scheduler.volcano.podGroupTemplateFile`로 PodGroup 정의를 제공합니다. Jaceklaskowski +3↗

Gang 스케줄링 설정 시 **메모리 계산이 중요**합니다. Spark의 `spark.driver.memory + spark.driver.memoryOverhead`와 `spark.executor.memory + spark.executor.memoryOverhead`의 합이 Yunikorn/Volcano의 task group `minResource`와 정확히 일치해야 합니다. apache↗Apache↗ 불일치 시 placeholder pod가 실제 Pod를 대체하지 못하거나, gang 스케줄링이 실패합니다.

Trino와 KEDA의 안전한 공존

Trino 배포는 **반드시 기본 스케줄러를 사용**해야 KEDA와 완벽히 호환됩니다. Helm 차트에서 `schedulerName`을 명시하지 않거나, `schedulerName: default-scheduler`로 설정합니다. Coordinator는 단일 인스턴스로 안정적인 네트워크 ID가 필요하며, Trino↗ Worker는 HPA 또는 KEDA로 스케일링합니다.

KEDA는 **스케줄러와 독립적**으로 동작합니다. KEDA Operator는 ScaledObject를 보고 HPA를 생성하며, HPA는 Deployment/StatefulSet의 replicas를 조정합니다. KEDA +2↗ 실제 Pod 스케줄링은 각 Pod의 `schedulerName`에 따라 처리되므로, 같은 네임스페이스에 Spark(Volcano)와 Trino(default + KEDA)가 공존해도 **KEDA는 영향을 받지 않습니다**. KEDA는 cluster-wide singleton으로 `v1beta1.external.metrics.k8s.io` API를 등록하므로, 클러스터당 하나만 설치 가능하지만, 여러 네임스페이스의 다양한 스케줄러를 사용하는 워크로드를 모두 지원합니다. KEDA↗

Trino용 KEDA 설정은 **Prometheus 메트릭 기반**이 권장됩니다. `trino_execution_ClusterSizeMonitor_RequiredWorkers` 메트릭으로 필요한 워커 수를 측정하고, CPU/메모리 utilization을 보조 트리거로 추가합니다. charts↗ `pollingInterval: 30s`, `cooldownPeriod: 300s`로 설정해 급격한 스케일 변동을 방지하고, `minReplicaCount`와 `maxReplicaCount`로 범위를 제한합니다. HPA behavior 설정으로 scale-down 시 50% 단계적 감소, scale-up 시 100% 즉시 증가 정책을 구현할 수 있습니다.

KEDA의 제약사항은 Spark executor를 직접 스케일링할 수 없다는 점입니다. Spark Operator가 executor lifecycle을 관리하므로, KEDA는 Spark 워크로드에 사용할 수 없습니다. 그러나 Spark History Server나 기타 지원 인프라는 KEDA로 스케일링 가능합니다. Trino와 달리 Spark는 애플리케이션 레벨에서 executor 수를 정의하므로, Spark Operator의 dynamic allocation 기능을 사용하거나 애플리케이션 설정으로 제어해야 합니다.

리소스 경합 관리 전략

여러 스케줄러가 동시에 동작하면 **race condition**이 발생할 수 있습니다. 모든 스케줄러는 Kubernetes API를 통해 노드 리소스를 조회하지만, 서로 직접 조율하지 않습니다. 두 스케줄러가 동시에 "8 CPU 여유"를 보고 각각 6 CPU Pod를 스케줄하면 over-commit이 발생합니다. 다행히 **Kubelet이 최종 gatekeeper**로 작동해, 실제 리소스가 부족하면 Pod admission을 거부하고 스케줄러가 재시도합니다. Stack Overflow↗ Kubernetes API의 optimistic concurrency control(etcd의 compare-and-swap)도 일관성을 보장하지만, 완벽한 보호는 아닙니다.

프로덕션 권장 전략은 노드 격리입니다. 특정 노드 풀을 Spark 전용으로 taint하고(`workload=spark:NoSchedule`), Spark Pod에만 toleration을 부여합니다. Trino는 taint 없는 노드 풀을 사용하므로 물리적으로 분리됩니다. GitHub +3↗ 200노드 클러스터에서는 150개를 Spark/배치 워크로드용, 50개를 Trino/서비스용으로 분할하는 것이 효과적입니다. 노드 레이블과 NodeSelector를 조합하면 더 세밀한 제어가 가능합니다.

대안으로 **Kubernetes 1.18+ Scheduling Profiles**를 사용하면 단일 스케줄러 프로세스에서 여러 스케줄링 정책을 실행할 수 있습니다. `default-scheduler`, `spark-scheduler`, `trino-scheduler` 프로필을 정의하고, 각각 다른 플러그인과 가중치를 설정합니다. 이 방식은 in-memory 상태를 공유해 race condition이 없지만, Yunikorn/Volcano 같은 외부 스케줄러는 사용할 수 없고, 프레임워크 플러그인으로만 제한됩니다. Google Groups↗

ResourceQuota는 모든 스케줄러에 적용됩니다. Kubernetes admission controller가 Pod 생성 시점에 quota를 검사하므로, schedulerName과 무관하게 동일한 제약이 적용됩니다. Kubernetes +2↗ 네임스페이스당 CPU/메모리 총량을 제한하고, scopeSelector로 특정 PriorityClass에만 quota를 적용할 수 있습니다. PerfectScale↗Apache↗ 반드시 LimitRange와 함께 사용해 리소스 요청이 없는 Pod의 quota 우회를 막아야 합니다. Uffizzi↗

PriorityClass를 통한 우선순위 보장

PriorityClass는 **클러스터 레벨 오브젝트**로 정수 값(-2,147,483,648 ~ 2,147,483,647)과 이름을 매핑합니다. 10억 이상은 시스템용으로 예약되어 있습니다. Kubernetes +2↗ 기본 스케줄러는 PriorityClass를 네이티브로 지원하며, 높은 우선순위 Pod를 먼저 스케줄하고 필요시 낮은 우선순위 Pod를 선점합니다. Kubernetes↗apache↗

Yunikorn은 PriorityClass를 완전히 지원하며, 추가로 큐 레벨 `priority.offset`으로 전체 큐의 우선순위를 조정할 수 있습니다. Apache↗ Trino 큐에 `priority.offset: 1000`을 설정하면 같은 PriorityClass라도 Trino가 Spark보다 먼저 스케줄됩니다. Priority fencing을 활성화하면 우선순위가 큐 계층 내로 제한되어 다른 팀의 높은 우선순위가 침범하지 못합니다. Apache↗ 주의할 점은 Yunikorn이 같은 우선순위 간 선점도 허용한다는 것입니다(큐 rebalancing 목적).

Volcano는 독자적인 우선순위 시스템을 사용합니다. 표준 PriorityClass를 직접 사용하지 않고, VolcanoJob과 PodGroup의 `priorityClassName` 필드를 읽어 Volcano 큐 정책에 반영합니다. volcano +2↗ Volcano 큐 설정에서 우선순위 기반 선점을 구성할 수 있지만, 기본 스케줄러의 PriorityClass 선점 로직과는 다릅니다.

Kueue는 기본 스케줄러를 통해 PriorityClass를 존중합니다. Kueue가 Job을 admit하면 기본 스케줄러가 Pod를 배치하며, 이때 PriorityClass가 적용됩니다. ClusterQueue의 preemption 정책(`reclaimWithinCohort: LowerPriority`)으로 cohort 내 우선순위 기반 리소스 회수가 가능합니다.

프로덕션 환경에서는 **5-7단계 PriorityClass 계층**을 권장합니다. `system-cluster-critical`(20억)은 kube-system용, Kubernetes↗KodeKloud Notes↗ `trino-critical`(100만)은 Trino coordinator, `trino-normal`(10만)은 Trino worker, `spark-high-priority`(5만)는 중요 Spark 작업, `batch-normal`(0)은 일반 배치, `development`(-100)는 개발 워크로드, `preemptible`(-1000)은 선점 가능한 저우선순위 작업입니다. ResourceQuota의 scopeSelector로 네임스페이스별로 사용 가능한 PriorityClass를 제한하여 악용을 방지합니다. Kubernetes↗

프로덕션 모범 사례

200노드, 5팀 환경에서 **네임스페이스 분리 전략**이 가장 안전합니다. 팀별로 `team-a-prod`, `team-b-prod` 네임스페이스를 생성하고, 각 네임스페이스에서 독립적으로 스케줄러를 선택합니다. RBAC으로 팀 간 격리를 보장하고, ResourceQuota로 네임스페이스별 리소스 한도를 설정합니다. 같은 네임스페이스에서 혼용도 가능하지만, Pod 레이블링과 큐 설정이 복잡해지고 misconfiguration 위험이 증가합니다.

노드 풀 전략은 하드웨어 특성과 워크로드를 매핑합니다. Theodo↗Google Cloud↗ Trino용 고메모리 노드(40개), Spark용 범용 노드(80개), ML용 GPU 노드(20개), 서비스용 일반 노드(40개), 플랫폼용 노드(20개)로 분할합니다. 각 노드 풀에 taint를 설정하고, 워크로드별로 matching toleration을 부여합니다. Theodo↗Microsoft Learn↗ 노드 레이블은 `node-restriction.kubernetes.io/` prefix를 사용해 kubelet의 조작을 방지합니다. Google Cloud↗

스케줄러 선택 가이드: Yunikorn은 멀티테넌트 환경에서 계층적 큐와 공정성 정책이 필요할 때 최선입니다. InfraCloud↗ Alibaba와 Apple의 대규모 배포로 검증되었고, 배치와 서비스 워크로드를 동시에 처리합니다. Packt Publishing↗Cloudera Blog↗ Volcano는 AI/ML 워크로드에서 gang 스케줄링과 job dependency가 필수일 때 선택하

며, Amazon과 Tencent의 사례가 있습니다. InfraCloud +2↗ Kueue는 기존 클러스터에 점진적 도입이 필요하거나 복잡도를 최소화하려면 좋지만, gang 스케줄링과 고급 기능은 제한적입니다. InfraCloud +2↗

Trino 우선순위 보장을 위한 **추천 아키텍처**는 다음과 같습니다. Yunikorn을 analytics 네임스페이스(80-100 노드)에 배포하고, `root.analytics.trino-critical` 큐에 `priority.offset: 1000`과 높은 guaranteed 리소스를 설정합니다. Spark 작업은 `root.analytics.spark-batch` 큐에서 `priority.offset: 0` 또는 음수 값으로 우선순위를 낮춥니다. Trino coordinator는 PriorityClass `trino-critical`(100만)을 사용하고, PodDisruptionBudget으로 보호합니다. Trino worker는 KEDA로 자동 스케일하되, 기본 스케줄러를 유지합니다.

모니터링은 필수입니다. Prometheus로 각 스케줄러의 메트릭을 수집합니다. Yunikorn은 포트 9080에서 `yunikorn_scheduler_queue_allocated_memory`, `yunikorn_scheduler_preemption_attempts` 등을 노출하고, Volcano는 `volcano_job_count`, `volcano_queue_allocated`를 제공합니다. 기본 스케줄러는 `scheduler_schedule_attempts_total`, `scheduler_pending_pods`를 제공합니다. Grafana 대시보드로 스케줄러별 리소스 사용량, 큐 상태, 선점 활동을 시각화합니다. 핵심 알림은 Pod 스케줄링 실패율 \u003e 10%, 리소스 overcommit 임박, 다중 스케줄러 충돌, Trino coordinator 장애 등입니다.

알려진 이슈와 제약사항

Yunikorn은 최근 preemption 버그(YUNIKORN-2699, YUNIKORN-2725)가 보고되어 일부 테스트가 비활성화되었습니다. 프로덕션 배포 전 staging 환경에서 선점 동작을 철저히 검증해야 합니다. Plugin mode는 deprecated되어 standalone mode만 사용해야 합니다. 현재 안정 버전은 v1.5.x이며, GitHub releases를 확인해 최신 패치를 적용합니다.

Volcano는 표준 Kubernetes Job/Deployment를 직접 사용할 수 없고 VolcanoJob CRD가 필요합니다. 기존 워크로드 마이그레이션이 복잡할 수 있으며, admission webhook로 자동 변환을 구현하거나 gradual migration이 필요합니다. 공식 문서는 클러스터 전체에 Volcano 사용을 권장하지만, 실무에서는 노드 분리로 선택적 사용이 가능합니다.

Kueue는 Spark Operator와 네이티브 통합이 없습니다. SparkApplication에 레이블을 추가해도 Operator가 자체 lifecycle을 관리해 Kueue의 suspend 메커니즘이 작동하지 않을 수 있습니다. Hopsworks 문서는 명시적으로 "Spark는 현재 지원되지 않으며 Kueue가 관리하지 않는다"고 명시합니다. 대안으로 Kubernetes Job + spark-submit을 사용하거나, Ray 같은 완전 통합된 프레임워크를 고려합니다.

Karpenter(AWS EKS 자동 스케일러)는 현재 기본 스케줄러만 지원합니다(GitHub issue #742). Yunikorn/Volcano 사용 시 Cluster Autoscaler를 대신 사용해야 합니다. DaemonSet은 모든 노드 taint에 대한 toleration이 필수이며, 누락 시 pending 상태가 됩니다. 높은 PriorityClass를 DaemonSet에 부여해 선점을 방지합니다.

PriorityClass는 클러스터 레벨 리소스이므로 어떤 사용자도 참조할 수 있습니다. 악의적인 사용자가 `trino-critical`을 자신의 Pod에 지정할 수 있으므로, ResourceQuota의 scopeSelector로 네임스페이스별로 허용된 PriorityClass를 제한합니다. PodSecurityPolicy(deprecated) 또는 OPA/Kyverno 같은 admission controller로 정책을 강제할 수도 있습니다.

대안 아키텍처 패턴

패턴 1: 단일 스케줄러 + 노드 풀 분리가 가장 단순합니다. 기본 스케줄러만 사용하고, PriorityClass와 노드 taint/toleration으로 워크로드를 격리합니다. Trino 노드에 높은 메모리 하드웨어를 배치하고 taint하며, Spark는 spot instance 노드 풀을 사용합니다. Gang 스케줄링이 필수가 아니라면 이 방식이 운영 복잡도를 최소화합니다.

패턴 2: 하이브리드(Yunikorn + Default)는 대규모 클러스터에 적합합니다. Analytics 워크로드(Trino, Spark)는 Yunikorn으로 관리하고(80-100 노드), 일반 서비스는 기본 스케줄러를 유지합니다(100-120 노드). 각 스케줄러가 전용 노드 풀을 갖춰 경합을 방지하고, gradual migration이 가능하며, 장애 시 blast radius가 제한됩니다. 모니터링은 복잡해지지만 유연성이 높습니다.

패턴 3: 네임스페이스 + 노드 풀 조합은 궁극의 프로덕션 패턴입니다. 각 팀에 전용 네임스페이스와 노드 풀을 할당합니다. `team-analytics` 네임스페이스는 고메모리 노드 풀(40개)과 Yunikorn 스케줄러를 사용하고, `team-ml` 네임스페이스는 GPU 노드 풀(20개)과 Volcano를 사용하며, `team-services` 네임스페이스는 일반 노드 풀과 기본

스케줄러를 사용합니다. ResourceQuota로 네임스페이스별 한도를 설정하고, NetworkPolicy로 네트워크 격리까지 구현하면 완벽한 멀티테넌시가 가능합니다.

Over-commitment 전략으로 리소스 효율을 높일 수 있습니다. Guaranteed tier(Trino coordinator): requests = limits로 선점 불가능. Burstable tier(Trino worker, ML training): requests \u003c limits로 여유 시 burst 가능. Best-effort tier(배치 작업, 개발): limits 없음, 먼저 선점됨. Yunikorn 큐에서 `guaranteed`, `max` 리소스와 `priority.offset`을 조합해 3단계 tier를 구현합니다.

구현 로드맵

1단계(1-2주차): 기반 구축. 네임스페이스 구조를 생성하고, 노드 풀에 레이블과 taint를 적용하며, ResourceQuota/LimitRange를 구성합니다. 5-7개 PriorityClass를 정의하고 팀별로 사용 가능한 클래스를 문서화합니다. 모니터링 스택(Prometheus, Grafana)을 먼저 배포해 변경 전 baseline을 수집합니다.

2단계(3-4주차): 스케줄러 배포. Yunikorn을 analytics 네임스페이스에 Helm으로 설치하고 (`embedAdmissionController: false`), 큐 계층을 구성합니다. Trino 워크로드의 10%를 테스트 네임스페이스로 마이그레이션하고 1주일 모니터링합니다. 스케줄링 latency, 큐 utilization, 선점 이벤트를 추적합니다.

3단계(5-8주차): 확장. 나머지 Trino와 Spark 워크로드를 마이그레이션하고, 필요시 Volcano를 ML 네임스페이스에 배포합니다. Spark Operator webhook를 활성화해 자동으로 gang 스케줄링 어노테이션을 추가합니다. KEDA ScaledObject로 Trino worker 자동 스케일링을 구성하고, 기본 스케줄러 유지를 확인합니다. 운영 runbook을 작성합니다.

4단계(9-12주차): 최적화. 큐 설정을 튜닝하고(guaranteed/max 리소스, priority.offset), 실제 워크로드 패턴에 맞춰 조정합니다. 비용 효율을 위해 over-commitment 정책을 검토하고, preemption threshold를 조정합니다. Grafana 대시보드를 개선하고, 스케줄러별 성능 벤치마크를 수행합니다.

실전 설정 예시

yaml

```yaml
# Yunikorn 큐 설정 (5팀 환경)
queues:
  - name: root
    queues:
      - name: team-a  # Analytics 팀
        resources:
          guaranteed: {memory: 1000Gi, vcore: 500}
          max: {memory: 1500Gi, vcore: 750}
        queues:
          - name: trino
            properties:
              priority.offset: "1000"
              preemption.policy: "fence"
            resources:
              guaranteed: {memory: 600Gi, vcore: 300}
          - name: spark
            properties:
              application.sort.policy: "fifo"
            resources:
              guaranteed: {memory: 400Gi, vcore: 200}

      - name: team-b  # ML 팀
        resources:
          guaranteed: {memory: 600Gi, vcore: 300, nvidia.com/gpu: 16}
        properties:
          priority.offset: "0"
```

yaml

```yaml
# PriorityClass 계층
---
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: trino-critical
value: 1000000
preemptionPolicy: PreemptLowerPriority
---
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: trino-normal
value: 100000
---
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: spark-high
value: 50000
---
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: batch-normal
value: 0
globalDefault: true
```

yaml

```yaml
# 노드 분리 전략
# Trino 전용 노드 taint
kubectl taint nodes node-{1..40} workload=trino:NoSchedule
kubectl label nodes node-{1..40} workload-type=trino

# Spark Pod는 toleration 추가
spec:
  schedulerName: yunikorn
  priorityClassName: spark-high
  nodeSelector:
    workload-type: spark
  tolerations:
  - key: workload
    value: spark
    effect: NoSchedule

# Trino Pod는 기본 스케줄러 유지
spec:
  # schedulerName 미지정 = default-scheduler
  priorityClassName: trino-critical
  nodeSelector:
    workload-type: trino
  tolerations:
  - key: workload
    value: trino
    effect: NoSchedule
```


yaml

```yaml
# KEDA ScaledObject (Trino Worker)
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: trino-worker-scaler
  namespace: team-analytics
spec:
  scaleTargetRef:
    name: trino-worker
  minReplicaCount: 3
  maxReplicaCount: 20
  pollingInterval: 30
  cooldownPeriod: 300
  triggers:
  - type: prometheus
    metadata:
      serverAddress: "http://prometheus:9090"
      threshold: "5"
      query: |
        ceil(sum(avg_over_time(
          trino_execution_ClusterSizeMonitor_RequiredWorkers[2m]
        )))
  advanced:
    horizontalPodAutoscalerConfig:
      behavior:
        scaleDown:
          stabilizationWindowSeconds: 300
          policies:
          - type: Percent
            value: 50
            periodSeconds: 60
```

핵심 권장사항

귀하의 200노드, 5팀, Trino 우선 환경에는 **Yunikorn + 노드 분리 전략**이 최적입니다. Yunikorn을 analytics 워크로드(80-100 노드)에 배포하고 계층적 큐로 팀과 워크로드를 구분합니다. Trino 큐에 높은 priority.offset과 guaranteed 리소스를 할당해 Spark보다 우선권을 부여합니다. Trino는 명시적으로 기본 스케줄러를 유지(`schedulerName` 미지정)하고 KEDA로 자동 스케일링합니다. 물리적 노드 분리(taint/toleration)로 스케줄러 간 리소스 경합을 원천 차단하고, 5-7단계 PriorityClass로 세밀한 우선순위를 제어합니다.

Spark는 Spark Operator의 `batchScheduler: yunikorn`으로 gang 스케줄링을 활성화하고, task group 어노테이션으로 driver + executor의 동시 스케줄링을 보장합니다. Native spark-submit 사용 시 Pod 템플릿으로 schedulerName을 지정합니다. 각 팀은 독립적인 네임스페이스(또는 같은 네임스페이스의 다른 큐)에서 ResourceQuota로 리소스 한도를 받습니다.

모니터링은 필수이며 Prometheus + Grafana로 스케줄러 메트릭, 큐 utilization, 선점 활동, Pod 스케줄링 latency를 추적합니다. 단계적 마이그레이션으로 위험을 최소화하고, 각 단계마다 1주일 모니터링 기간을 두어 안정성을 확인합니다. 이 아키텍처는 Trino의 높은 우선순위를 보장하면서도 Spark의 효율적인 gang 스케줄링과 KEDA 자동 스케일링을 모두 지원하며, 5개 팀의 멀티테넌트 요구사항을 충족합니다.