Ай-ТИ Фреш

ТЕХНИЧЕСКИЙ РАЗБОР

Job не должен падать из-за drain узла

Как мы разводим ошибки приложения и обслуживание узла в одном backoffLimit

Июль 2026

itfresh.ru · ИТ-аутсорсинг для юридических лиц

О ЧЁМ ЭТОТ РАЗБОР

Суть проблемы

backoffLimit Job ставят в 0 или 1, чтобы пакетная обработка не гоняла по кругу код с багом. Но kubectl drain эвиктит Pod через Eviction API, Job засчитывает это обычной неудачной попыткой, лимит исчерпывается за одну плановую операцию — и весь Job уходит в Failed вместе с посчитанными индексами. Дежурный разбирает логи вместо перезапуска обслуживания.

Почему это важно бизнесу

- Каждый drain узла (патчинг, замена ноды, автоскейлинг) превращается в сбой ночного биллинга или ETL-джобы
- Дежурный тратит 30-60 минут на разбор Failed Job, хотя причина — плановая эвакуация, а не баг приложения
- Кластер нельзя патчить в рабочее окно, пока в нём крутятся Job с backoffLimit=0 — обслуживание уезжает в ночь
- Частично выполненные индексы Indexed Job пересчитываются с нуля, удваивая время закрытия отчётного периода
- Ложные алерты на Failed Job создают шум и снижают доверие к реальным авариям приложения

ЧТО МЫ ЗАКЛАДЫВАЕМ

Ключевые параметры реализации

0

backoffLimit для критичных пакетных Job: ноль попыток на реальный баг приложения

наш стандарт

Ignore

action на condition DisruptionTarget: эвикция не инкрементит счётчик backoffLimit

kubernetes.io, Jobs

Never

restartPolicy шаблона Pod — обязателен, если в Job задан podFailurePolicy

kubernetes.io, Jobs

6

дефолт backoffLimit; при заданном backoffLimitPerIndex он равен 2147483647 (MaxInt32)

kubernetes.io, Jobs

Failed

podReplacementPolicy для Job с podFailurePolicy: дефолт и единственное допустимое значение

kubernetes.io, Jobs v1.36

120 с

terminationGracePeriodSeconds тяжёлых batch-Pod: время на checkpoint до SIGKILL

наш стандарт

РЕАЛИЗАЦИЯ 1 ИЗ 3

Pod failure policy: отделяем drain от бага приложения

Что настраиваем

Обычный и Indexed Job на воркер-неймспейсе, restartPolicy: Never, кластер 1.36

Как мы это делаем

1. Задаём spec.podFailurePolicy.rules: первое правило — onPodConditions: [{type: DisruptionTarget}] с action: Ignore
2. Второе правило — onExitCodes: {operator: NotIn, values: [0]} с action: FailJob: ненулевой код приложения валит Job без retry
3. Явно ставим restartPolicy: Never в template.spec — без него API не примет Job с podFailurePolicy
4. Где нужна изоляция сбоя батча — включаем completionMode: Indexed и backoffLimitPerIndex вместо общего backoffLimit
5. podReplacementPolicy руками не задаём: при podFailurePolicy он и так Failed, другое значение API отклонит

РЕЗУЛЬТАТ

Плановый drain больше не тратит backoffLimit: эвакуированный Pod получает condition DisruptionTarget, правило Ignore не даёт засчитать попытку, а взамен создаётся новый Pod на здоровом узле. Реальный баг с ненулевым кодом выхода по-прежнему валит Job сразу.

КЛЮЧЕВОЙ НЮАНС

Правила podFailurePolicy.rules берутся по порядку: после первого совпадения остальные игнорируются. DisruptionTarget ставим первым, иначе правило по exit code поймает 137 от эвикции и завалит Job.

РЕАЛИЗАЦИЯ 2 ИЗ 3

Безопасный drain: PDB + AlwaysAllow вместо force-delete

Что настраиваем

Узлы node-pool с батч-нагрузкой, обслуживание через kubectl drain и cluster-autoscaler

Как мы это делаем

1. На группы batch-воркеров заводим PodDisruptionBudget (policy/v1) с maxUnavailable: 1 — ограничиваем одновременную просадку
2. Ставим PDB.spec.unhealthyPodEvictionPolicy: AlwaysAllow, иначе Pod в CrashLoopBackOff или без Ready блокируют drain
3. Обслуживание — только kubectl drain <node> --ignore-daemonsets --delete-emptydir-data --grace-period=120 --timeout=300s
4. drain сперва делает cordon: узел получает taint node.kubernetes.io/unschedulable:NoSchedule — проверяем, что batch-Pod его не толерируют
5. На долгих Job заранее ставим activeDeadlineSeconds, чтобы зависший процесс не держал окно обслуживания

РЕЗУЛЬТАТ

Drain укладывается в плановое окно без ручного вмешательства: PDB не даёт эвакуировать больше одного Pod из группы, AlwaysAllow не позволяет зависшим unhealthy Pod блокировать операцию, а grace-period оставляет batch-процессу время корректно закрыть текущий шаг и сохранить checkpoint.

КЛЮЧЕВОЙ НЮАНС

Eviction API проставляет condition DisruptionTarget (reason EvictionByEvictionAPI) до удаления Pod, поэтому Ignore работает даже при SIGKILL. Схему ломает только force-delete мимо eviction.

РЕАЛИЗАЦИЯ 3 ИЗ 3

Индексная изоляция и уборка после Job

Что настраиваем

Indexed Job с параллельной обработкой батчей (completionMode: Indexed, restartPolicy: Never)

Как мы это делаем

1. Задаём backoffLimitPerIndex: 1-2 — при превышении лимита индекс уходит в status.failedIndexes, остальные индексы считаются дальше
2. Ставим maxFailedIndexes: чтобы массовый сбой батчей останавливал Job, а не молотил все индексы вхолостую
3. Общий backoffLimit не задаём: при backoffLimitPerIndex он по умолчанию 2147483647 и не мешает
4. Правило onPodConditions DisruptionTarget → Ignore нужно и здесь; для быстрого отказа батча используем action FailIndex
5. Ставим ttlSecondsAfterFinished: 3600, а в CronJob сверяем successfulJobsHistoryLimit/failedJobsHistoryLimit

РЕЗУЛЬТАТ

Один сбойный батч не тянет за собой весь пакет: превышение backoffLimitPerIndex помечает только свой индекс, а drain благодаря Ignore не считается сбоем ни на уровне индекса, ни на уровне Job. Завершённые объекты чистит ttlSecondsAfterFinished.

КЛЮЧЕВОЙ НЮАНС

backoffLimitPerIndex требует completionMode: Indexed и restartPolicy: Never — иначе API отклонит манифест на валидации. Держим это обязательным пунктом код-ревью вместе с maxFailedIndexes.

ЧТО МЫ ВЫУЧИЛИ НА ПРАКТИКЕ

Подводные камни

- ✕ **backoffLimit=0 без podFailurePolicy**
  Ноль одинаково реагирует и на drain, и на ошибку кода. Без правила Ignore на DisruptionTarget лимит сгорает на первом же обслуживании узла.

- ✕ **Забыли restartPolicy: Never**
  podFailurePolicy применим только к Job с restartPolicy=Never. При OnFailure kubelet перезапускает контейнер сам, и правила Job не участвуют.

- ✕ **DisruptionTarget не первое правило**
  Правила берутся по порядку, после первого совпадения остальные игнорируются. Правило по exit code выше поймает 137 от эвикции и завалит Job.

- ✕ **PDB с minAvailable = числу реплик**
  Такой PDB запрещает эвакуацию любого Pod, drain висит до timeout. Считаем допустимую просадку через maxUnavailable, а не держим 100% на время работ.

- ✕ **Дефолтный unhealthyPodEvictionPolicy**
  Дефолт IfHealthyBudget пускает эвикцию нездорового Pod, только если currentHealthy ≥ desiredHealthy: CrashLoopBackOff вешает drain. Нужен AlwaysAllow.

- ✕ **grace-period меньше checkpoint батча**
  Процесс не успевает сохранить состояние до SIGKILL. Попытка не спишется (Ignore сработает), но индекс придётся пересчитывать с нуля на новом узле.

- ✕ **podReplacementPolicy задан вручную**
  Для Job с podFailurePolicy допустимо только Failed — манифест с TerminatingOrFailed API отклонит. Поле просто не указываем.

- ✕ **Нет activeDeadlineSeconds на долгих Job**
  Без дедлайна зависший после эвакуации процесс держит Job активным сколько угодно, блокируя ttlSecondsAfterFinished и уборку неймспейса.

УРОВНИ РЕАЛИЗАЦИИ

Как правильно

МИНИМУМ

- podFailurePolicy: правило Ignore на DisruptionTarget перед правилом FailJob
- restartPolicy: Never в template.spec каждого Job с podFailurePolicy
- PodDisruptionBudget (policy/v1) с maxUnavailable на batch-нагрузку

НОРМАЛЬНО

- unhealthyPodEvictionPolicy: AlwaysAllow на всех batch-PDB
- grace-period drain ≥ времени checkpoint процесса (обычно 60-120 с)
- activeDeadlineSeconds на каждом долгоживущем Job
- ttlSecondsAfterFinished для автоочистки завершённых Job

ХОРОШО

- completionMode: Indexed + backoffLimitPerIndex + maxFailedIndexes
- алертинг по reason condition DisruptionTarget отдельно от FailJob
- регламент: drain только через eviction API, без force-delete Pod
- код-ревью манифестов на restartPolicy/podFailurePolicy/PDB

ПРОВЕРЬТЕ У СЕБЯ

Чек-лист самопроверки

- [ ] В Job задан podFailurePolicy с правилом Ignore на condition DisruptionTarget первым по порядку?
- [ ] В Pod template явно задан restartPolicy: Never там, где используется podFailurePolicy?
- [ ] На batch-нагрузке заведён PodDisruptionBudget с адекватным maxUnavailable?
- [ ] unhealthyPodEvictionPolicy у batch-PDB выставлен в AlwaysAllow, а не оставлен по умолчанию?
- [ ] grace-period в регламенте drain больше реального времени checkpoint приложения?
- [ ] Для Indexed Job заданы backoffLimitPerIndex и maxFailedIndexes вместо общего backoffLimit?
- [ ] На долгоживущих Job выставлен activeDeadlineSeconds?
- [ ] Завершённые Job чистятся через ttlSecondsAfterFinished, а не ручным cron?
- [ ] Алерты различают Failed Job по FailJob и эвакуацию по condition DisruptionTarget?
- [ ] Регламент запрещает force-delete Pod (--force --grace-period=0) на batch-неймспейсах?

Если хотя бы на два вопроса ответ «нет» или «не знаю» — тема требует внимания.

ЧТО МЫ ДЕЛАЕМ

Как поможет ITFresh

ITFresh — ИТ-аутсорсинг для юридических лиц до 50 рабочих мест в Москве и области. 15+ лет практики, собственная инфраструктура в дата-центре МТС (8 серверов Dell Xeon Platinum).

- Проектируем podFailurePolicy для существующих Job, чтобы drain узла не тратил backoffLimit
- Настраиваем PodDisruptionBudget и unhealthyPodEvictionPolicy под batch-нагрузки заказчика
- Переводим пакетные джобы на Indexed + backoffLimitPerIndex для изоляции сбоя батча
- Пишем регламент безопасного drain: grace-period, activeDeadlineSeconds, TTL, запрет force
- Настраиваем алертинг, различающий сбой приложения и плановое disruption-событие

15+
лет в ИТ-поддержке

50
рабочих мест — наш профиль

МТС
дата-центр, Москва

КОНТАКТЫ

Обсудить вашу задачу

Сайт	**itfresh.ru**
Телефон	**+7 903 729-62-41**
Telegram	**@ITfresh_Boss**

Бесплатно посмотрим вашу инфраструктуру по этому чек-листу и скажем, где тонко — без обязательств.

itfresh.ru

ДОКУМЕНТАЦИЯ И НАШ СТЕК

Техническая база

01 Jobs: pod backoff failure policy, backoffLimit, maxFailedIndexes (kubernetes.io — v1.36)

02 Jobs: Pod failure policy и Backoff limit per index (kubernetes.io — v1.36)

03 Jobs: Delayed creation of replacement pods (podReplacementPolicy) (kubernetes.io — v1.36)

04 Handling retriable and non-retriable pod failures (kubernetes.io — v1.36)

05 Pod Lifecycle: Pod conditions, DisruptionTarget (kubernetes.io — v1.36)

06 Disruptions: Pod disruption conditions, Eviction API (kubernetes.io — v1.36)

07 Specifying a Disruption Budget: Unhealthy Pod Eviction Policy (kubernetes.io — v1.36)

08 Safely Drain a Node (kubernetes.io — v1.36)

09 Наш регламент безопасного drain кластеров заказчиков (itfresh.ru — 2026)

Основано на официальной документации продуктов и нашей практике внедрения.