Ай-ТИ Фреш

ТЕХНИЧЕСКИЙ РАЗБОР

GitLab Runner в Kubernetes: CI-ферма с автоскейлингом

Ставим Kubernetes Executor через Helm: изоляция job в pod-ах, S3-кэш, DinD и автоскейлинг нод

Июль 2026

itfresh.ru · ИТ-аутсорсинг для юридических лиц

О ЧЁМ ЭТОТ РАЗБОР

Суть проблемы

Пара shell-раннеров на виртуалках упирается в потолок: утром десятки пайплайнов стоят в очереди, разработчики ждут сборку по 20–30 минут, а ночью те же VM простаивают, но оплачиваются. Мы переносим GitLab CI в Kubernetes: каждая job — отдельный pod с лимитами, десятки параллельных сборок, а ноды CI-пула создаются под пик и гасятся автоскейлером через 5 минут простоя.

Почему это важно бизнесу

- Очередь CI — оплаченные часы простоя: 10 разработчиков по 30 минут ожидания в день — минус 5 человеко-часов ежедневно
- Фиксированные VM под раннеры оплачиваются 24/7 при пиковой нагрузке CI — автоскейлинг нод срезает расходы на простой
- Общий раннер без изоляции — риск утечки секретов prod-деплоя в любую feature-ветку любого разработчика
- Релизы перестают зависеть от «свободен ли раннер»: параллельные сборки сокращают выпуск фич и хотфиксов

ЧТО МЫ ЗАКЛАДЫВАЕМ

Ключевые параметры реализации

19.x

GitLab Runner: мажор.минор держим синхронно с GitLab-сервером — рассинхрон версий официально н…

по докам GitLab Runner

30

concurrent на менеджер (в Helm-чарте по умолчанию 10); за 50 job на менеджер — поднимаем второй

наш стандарт

600 с

poll_timeout вместо дефолтных 180 с: pod дожидается scale-up новой ноды и не падает по таймауту

config.toml [runners.kubernetes]

0.2→2 CPU

cpu_request 200m / cpu_limit 2 на build-контейнер: честный binpacking автоскейлера и защита но…

наш стандарт

5 мин

scale-down-unneeded-time у cluster-autoscaler: CI-ноды гасим быстро — на них важна цена, а не…

cluster-autoscaler

3 с

checkInterval — частота опроса GitLab на новые job; при нескольких менеджерах поднимаем до 10 с

по докам Helm-чарта

Менеджер раннера: Helm-чарт и RBAC

Что настраиваем

Namespace gitlab-runner, чарт gitlab/gitlab-runner, отдельный CI node pool с taint ci=true:NoSchedule

Как мы это делаем

1. helm repo add gitlab https://charts.gitlab.io; токен glrt-* берём в Admin Area → Runners → New instance runner (старые регистрационные токены выведены из оборота)
2. values.yaml: gitlabUrl, runnerToken, concurrent=30, rbac.create=true, clusterWideAccess=false — права менеджера только в своём namespace
3. [runners.kubernetes]: namespace=gitlab-runner, image=ubuntu:22.04, отдельные requests/limits на build-, helper- и service-контейнеры
4. node_selector role=ci + toleration на taint ci=true:NoSchedule — job-pod'ы не соседствуют с прод-нагрузкой
5. poll_timeout=600 под ожидание scale-up ноды; деплой строго helm upgrade --install через GitOps, конфиг руками не правим

РЕЗУЛЬТАТ

Менеджер живёт как обычный Helm-релиз: обновление и откат одной командой, вся конфигурация в git. Job распределяются по выделенному CI-пулу, прод-ноды не затронуты, а рестарт менеджера не теряет очередь — GitLab раздаёт job заново.

КЛЮЧЕВОЙ НЮАНС

clusterWideAccess=false и ServiceAccount с минимумом прав (create/delete pods, чтение secrets только в своём namespace) — первое, что сверяем с доками чарта при каждом внедрении.

РЕАЛИЗАЦИЯ 2 ИЗ 3

Сборка образов: DinD и rootless BuildKit

Что настраиваем

Job-pod'ы сборки Docker-образов в registry; privileged-контур отделён от обычных тестов и линтеров

Как мы это делаем

1. Namespace помечаем pod-security.kubernetes.io/enforce=privileged — при enforce baseline/restricted Pod Security Admission отклонит DinD-pod ещё на создании
2. docker:28 + service docker:28-dind, DOCKER_TLS_CERTDIR=/certs, DOCKER_HOST=tcp://docker:2376 — только TLS-порт демона
3. Том empty_dir docker-certs с medium=Memory в /certs/client — TLS-сертификаты DinD живут в памяти и не требуют PVC
4. Где privileged запрещён — moby/buildkit:rootless с BUILDKITD_FLAGS=--oci-worker-no-process-sandbox и push прямо из buildctl-daemonless.sh

РЕЗУЛЬТАТ

Сборка и push образов идут внутри кластера без выделенной build-VM. Privileged-поверхность сжата до одного тегированного раннера, а на новых проектах обходимся вовсе без privileged — rootless BuildKit с кэшем слоёв собирает быстрее DinD.

КЛЮЧЕВОЙ НЮАНС

privileged=true действует на все job раннера, поэтому DinD выносим в отдельный [[runners]] со своим тегом, а не включаем флаг на общем раннере тестов.

РЕАЛИЗАЦИЯ 3 ИЗ 3

S3-кэш зависимостей и автоскейлинг CI-нод

Что настраиваем

Кэш пайплайнов на MinIO + cluster-autoscaler на выделенном CI-пуле нод

Как мы это делаем

1. [runners.cache]: Type=s3, Shared=true; [runners.cache.s3]: ServerAddress MinIO, BucketName gitlab-cache, ключи доступа — из Kubernetes Secret, не в values
2. Ключи кэша по стекам: $CI_COMMIT_REF_SLUG-node (node_modules), -pip (.venv), -go (.cache/go-build), -m2 (.m2/repository)
3. cluster-autoscaler: --scale-down-unneeded-time=5m, --scale-down-delay-after-add=3m, --scale-down-utilization-threshold=0.4
4. ResourceQuota на namespace gitlab-runner — жёсткий потолок CPU/RAM, чтобы всплеск concurrent не выел кластер

РЕЗУЛЬТАТ

Повторные сборки берут зависимости из S3 за секунды вместо минут для npm/pip/go/maven. CI-пул расширяется под утренний пик и сжимается почти до нуля ночью — ноды оплачиваются только в часы реальных сборок.

КЛЮЧЕВОЙ НЮАНС

Кэш на зональном PVC теряется при scheduling pod-а в другую зону, поэтому только S3-совместимый сторадж: он не привязан к ноде и честно расшаривается между всеми pod-ами.

ЧТО МЫ ВЫУЧИЛИ НА ПРАКТИКЕ

Подводные камни

- ✕ **PVC под docker-certs вместо empty_dir**
  PVC не успевает attach'иться к короткоживущему pod-у — job падает по таймауту. Сертификаты держим в empty_dir с medium=Memory
- ✕ **Забытый privileged или PSA restricted**
  DinD падает с «Cannot connect to the Docker daemon». Ставим privileged=true у DinD-раннера и label enforce=privileged на namespace
- ✕ **concurrent 100 без ResourceQuota**
  Суммарные limits никто не посчитал — ноды уходят в OOM. Quota на namespace плюс memory_limit на каждый контейнер job
- ✕ **Кэш на зональном PVC**
  Pod, запланированный в другую зону, теряет кэш и собирает с нуля. Переносим кэш на S3/MinIO с Shared=true
- ✕ **Устаревший helper_image**
  Старый helper не понимает новые git-команды: предупреждения, битые clone артефактов. Обновляем runner и helper одной версией
- ✕ **clusterWideAccess=true «на всякий случай»**
  Менеджер получает права на весь кластер: компрометация job = компрометация кластера. Только namespace-scoped RBAC
- ✕ **Один менеджер на prod и feature-ветки**
  Protected-секреты утекают в job из любой ветки. Разделяем: отдельный раннер только для protected-веток со своими переменными
- ✕ **Нет requests у build-контейнеров**
  Автоскейлер не видит реальную потребность, ноды переподписаны. cpu_request/memory_request в [runners.kubernetes] обязательны

УРОВНИ РЕАЛИЗАЦИИ

Как правильно

МИНИМУМ

- Один менеджер по Helm-чарту: rbac.create=true, clusterWideAccess=false
- Requests и limits на build-, helper- и service-контейнеры
- Кэш зависимостей на S3/MinIO с Shared=true вместо PVC
- poll_timeout=600 под ожидание автоскейлинга нод

НОРМАЛЬНО

- Отдельный node pool с taint ci=true:NoSchedule + node_selector у раннера
- cluster-autoscaler: scale-down через 5 мин простоя, threshold 0.4
- Раздельные раннеры для protected- и feature-веток, свои теги
- ResourceQuota и LimitRange на namespace gitlab-runner

ХОРОШО

- Rootless BuildKit или buildah вместо privileged DinD на новых кластерах
- NetworkPolicy: job-pod видит только registry и artifact store
- Секреты через External Secrets + Vault, короткоживущие токены в job
- Подпись образов cosign после сборки и проверка подписи при deploy

ПРОВЕРЬТЕ У СЕБЯ

Чек-лист самопроверки

- [ ] Раннеры зарегистрированы новыми токенами glrt-*, а не устаревшими регистрационными?
- [ ] У менеджера clusterWideAccess=false и ServiceAccount с минимально необходимыми правами?
- [ ] На каждый контейнер job (build, helper, service) заданы cpu/memory request и limit?
- [ ] Кэш пайплайнов лежит на S3-совместимом хранилище, а не на зональном PVC?
- [ ] CI-ноды вынесены в отдельный пул с taint и гасятся автоскейлером через 5 минут простоя?
- [ ] Privileged DinD вынесен в отдельный тегированный раннер, а не включён глобально?
- [ ] Protected-ветки собирает отдельный раннер, недоступный job из feature-веток?
- [ ] Версии gitlab-runner, helper_image и GitLab-сервера обновляются синхронно?
- [ ] ResourceQuota на namespace защищает кластер от всплеска параллельных job?
- [ ] poll_timeout увеличен так, чтобы job переживала scale-up новой ноды?

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).

- Разворачиваем GitLab Runner в вашем или нашем кластере под ключ: Helm, RBAC, кэш, автоскейлинг — 1–2 рабочих дня
- Переносим существующие .gitlab-ci.yml на Kubernetes Executor с сохранением тегов, переменных и артефактов
- Настраиваем S3-кэш на MinIO и cluster-autoscaler под профиль именно ваших сборок
- Разделяем контуры секретов: раннеры protected-веток, NetworkPolicy, подпись образов cosign
- Ставим ферму на мониторинг: метрики раннера в Prometheus/Zabbix, алерты на рост очереди job

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

Техническая база

01 Kubernetes executor (docs.gitlab.com — 19.x)

02 GitLab Runner Helm chart (docs.gitlab.com — 19.x)

03 Advanced configuration: config.toml, секция [runners.kubernetes] (docs.gitlab.com — 19.x)

04 Use Docker to build Docker images (DinD, TLS) (docs.gitlab.com — 19.x)

05 Caching in GitLab CI/CD: runners.cache.s3 (docs.gitlab.com — 19.x)

06 Cluster Autoscaler FAQ: параметры scale-down (github.com/kubernetes/autoscaler — 1.3x)

07 Шаблон values.yaml CI-фермы ITfresh (itfresh.ru — 2026)

Основано на официальной документации продуктов и нашей практике внедрения.