Основы kubernetes

About Me:

Aliaksandr Tsimokhau - Senior DevOps Engineer

- Увлечённый инженер DevOps с опытом 10+ лет в IT, из них 6+ лет — в DevOps
- Опыт в построении, оптимизации и масштабировании инфраструктуры для высоконагруженных систем.
- **Текущая роль:** DevOps / Technical Support Lead в UFL Strikerz Inc (игровая компания)
    - Строю инфраструктуру для высокопроизводительных игровых платформ
    - Связываю эксплуатацию и разработку для бесперебойных рабочих процессов

What I Do

- **Инфраструктура как код (Terraform)** — проектирование и поддержка масштабируемой, воспроизводимой инфраструктуры.
- **Kubernetes (операции с кластерами)** — управление кластерами, деплойментами, обновлениями и реагированием на инциденты.
- **Разработка Helm-чартов** — создание переиспользуемых чартов и шаблонов развёртывания.
- **Мониторинг и оповещения** — Prometheus и VictoriaMetrics: метрики, дашборды, правила оповещений.
- **Облака (GCP и AWS)** — сети, IAM, архитектура с оптимизацией затрат и автоматизация.
- **CI/CD (GitLab CI)** — построение надёжных пайплайнов для сборки, тестов, security-сканирования и деплойментов.

Kubernetes, основные понятия и архитектура

Контейнеризация: что это

- Упаковка кода, зависимостей и среды в переносимый контейнер
- Изоляция на уровне одной ОС (namespaces/cgroups), быстрый старт
- В отличие от ВМ: меньше оверхеда, проще переносимость

Преимущества контейнеризации

- Простое и гибкое развертывание (по сравнению с ВМ)
- CI/CD с быстрым откатом изменений
- Изоляция и микросервисы
- Отделение приложения от «железа»
- Единообразные dev/test/prod среды

Преимущества контейнеров

- **Изоляция приложений** — контейнеры изолируют приложения и их зависимости, предотвращая конфликты на одном хосте.
- **Портативность** — одинаково запускаются в разных средах/ОС и облаках без изменения конфигурации.
- **Масштабируемость** — быстрый запуск и удаление экземпляров для поддержания производительности.
- **Автоматизация** — сборка, деплой и управление автоматизируются; есть самовосстановление и откаты.
- **Экономия ресурсов** — легче ВМ, повышают плотность размещения и снижают затраты.

От контейнеров к оркестрации

- Ручное управление плохо масштабируется: сложность, безопасность, обновления
- Оркестраторы решают планирование, масштабирование, самовосстановление
- Исторически: Apache Mesos (2009) → Kubernetes (2014)

Что делают оркестраторы

- Равномерно распределяют ресурсы
- Автоматизируют обновления и откаты
- Настраивают масштабирование и наблюдаемость

Kubernetes (K8s)

- Платформа управления контейнеризованными приложениями
- Развёртывание, масштабирование, обновления, отказоустойчивость
- Нужен, когда много сервисов/контейнеров и их нужно координировать

Модели управления в Kubernetes

- Декларативная: описываем желаемое состояние (YAML/Helm/GitOps), контроллеры приводят кластер к нему
- Императивная: прямые команды (kubectl create/apply/delete) для изменения ресурсов

Container Runtime в K8s

- Для запуска контейнеров нужен CRI-совместимый рантайм
- Сегодня: containerd, CRI-O
- Docker использовался исторически (dockershim удалён), образы Docker совместимы

Kubernetes

- **Kubernetes абстрагирует ресурсы:**
  - Вычисления (CPU, память)
  - Сеть
  - Хранилище
- **Нет необходимости жёстко указывать узел или детали хранилища**
- **Декларативное развертывание:**
  - «Вот моё приложение, Kubernetes — разверни его и поддерживай в рабочем состоянии!»

Diagram: Kubernetes Resource Abstraction

```
+-----------------------------+
|        Kubernetes           |
|-----------------------------|
|   [Compute Pool]            |
|   [Network Pool]            |
|   [Storage Pool]            |
+-----------------------------+
         /        |        \
        /         |         \
   +--------+ +--------+ +--------+
   |  App1  | |  App2  | |  App3  |
   +--------+ +--------+ +--------+
```

Преимущества Kubernetes для разработки и развёртывания приложений

1. Микросервисная архитектура

Kubernetes оптимизирован для работы с микросервисами. Это означает, что приложения лучше разбивать на небольшие компоненты (сервисы), которые могут быть независимо развёрнуты и масштабируемы. Каждый микросервис может работать в отдельном контейнере и управляться Kubernetes.

2. Контейнеризация

Kubernetes предназначен для работы с контейнерами (чаще всего с Docker). Приложения должны быть упакованы в контейнеры с минимально необходимыми зависимостями. Это делает приложение более портативным и легко масштабируемым.

3. Устойчивость к отказам

Kubernetes предоставляет инструменты для автоматического перезапуска и переноса контейнеров в случае сбоев. Приложение должно быть готово к непредвиденным перезагрузкам, перераспределению на другие узлы и работе в режиме ожидания некоторых сервисов.

4. Масштабируемость

Kubernetes поддерживает горизонтальное и вертикальное масштабирование приложений. Приложения должны быть спроектированы с учетом возможности динамического изменения количества экземпляров (подов) в зависимости от нагрузки.

5. Безопасность

Kubernetes предоставляет различные механизмы безопасности, которые нужно учитывать при разработке:

- **Role-Based Access Control (RBAC)** — для ограничения доступа к ресурсам.
- **Network Policies** — для контроля сетевого взаимодействия между подами.
- **Secrets** — для безопасного хранения чувствительных данных.

6. Конфигурация и управление состоянием

Kubernetes предлагает встроенные механизмы для управления конфигурацией и состоянием приложений:

- **ConfigMaps** для управления конфигурациями, которые не содержат секретных данных.
- **Secrets** для управления конфиденциальной информацией.

Это позволяет обновлять конфигурацию без необходимости перезагрузки приложения.

7. Логи и мониторинг

Kubernetes обеспечивает централизованную обработку логов и метрик через интеграцию с системами мониторинга, такими как Prometheus, Grafana, ELK Stack и Fluentd. Приложения должны генерировать структурированные логи и поддерживать возможность мониторинга через экспорт метрик.

8. Сетевая архитектура и сервисы

Kubernetes использует виртуальную сеть для коммуникации между подами. Для предоставления внешнего доступа используется **Service**, который может работать через **Load Balancer** или **Ingress**.

9. CI/CD и DevOps

Для эффективного использования Kubernetes важно интегрировать его с процессом CI/CD:

- Создание образов контейнеров и автоматическое их развертывание.
- Использование таких инструментов, как Helm, Argo CD или Jenkins для автоматизации развертывания и управления релизами.

Архитектура и концепты Kubernetes:

Nodes, Pods, Containers, ReplicaSets, Deployments.

What Kubernetes can do?

2. Scaling Apps when and as needed.

1. Provide tools for App deployment.

3. Managing clusters of containers.

6. Change management within existing containered applications.

4. Helps optimize the use of underlying hardware found below your container.

5. Enables App components so it can restart and move through the system when and as needed.

Компоненты и архитектура Kubernetes

Архитектура Kubernetes модульна и масштабируема. Далее — ключевые концепции, узлы и ресурсы платформы.

Интерфейсы: CSI, CNI, CRI

- **CSI** (Container Storage Interface)
  - Интеграция систем хранения с K8s (через API)
  - Реализации: ceph, openebs, linstor
- **CRI** (Container Runtime Interface)
  - Взаимодействие K8s со средами выполнения контейнеров
  - Реализации: containerd, CRI-O
- **CNI** (Container Network Interface)
  - Сетевая конфигурация контейнеров и связь между подами
  - Реализации: flannel, calico, cilium

Кластеры и типы узлов

- **Control Plane (Master)**:
  - `kube-apiserver` — входная точка (REST API)
  - `etcd` — хранилище состояния (key-value)
  - `kube-controller-manager` — контроллеры поддержания желаемого состояния
  - `kube-scheduler` — планирование подов по узлам
- **Worker nodes**:
  - `kubelet` — агент на каждом узле, управляет жизненным циклом подов
  - `kube-proxy` — сетевые правила и проброс трафика
  - `Container Runtime` — запуск контейнеров (напр., containerd, CRI-O)

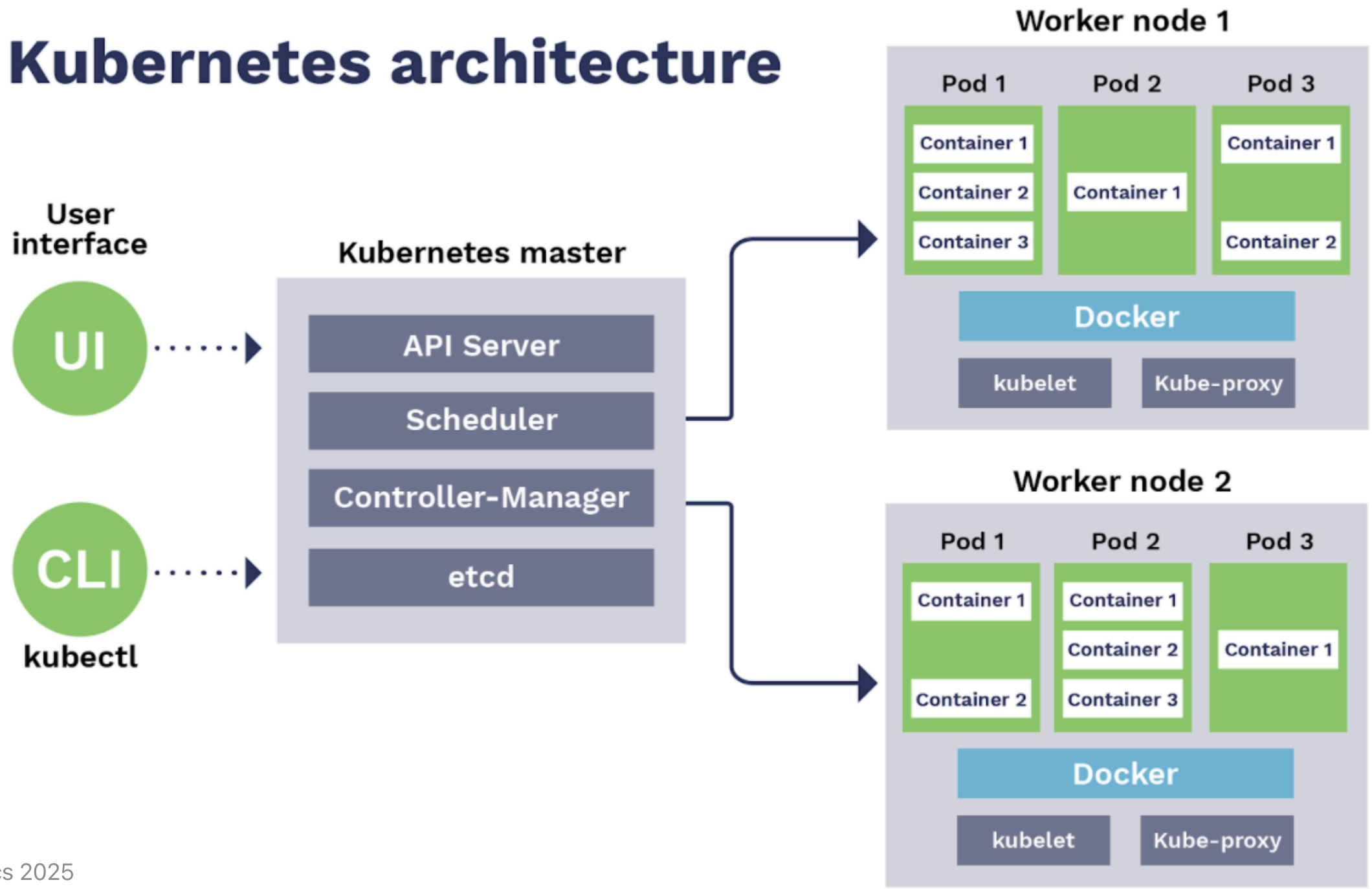
Контроллеры (управление приложениями)

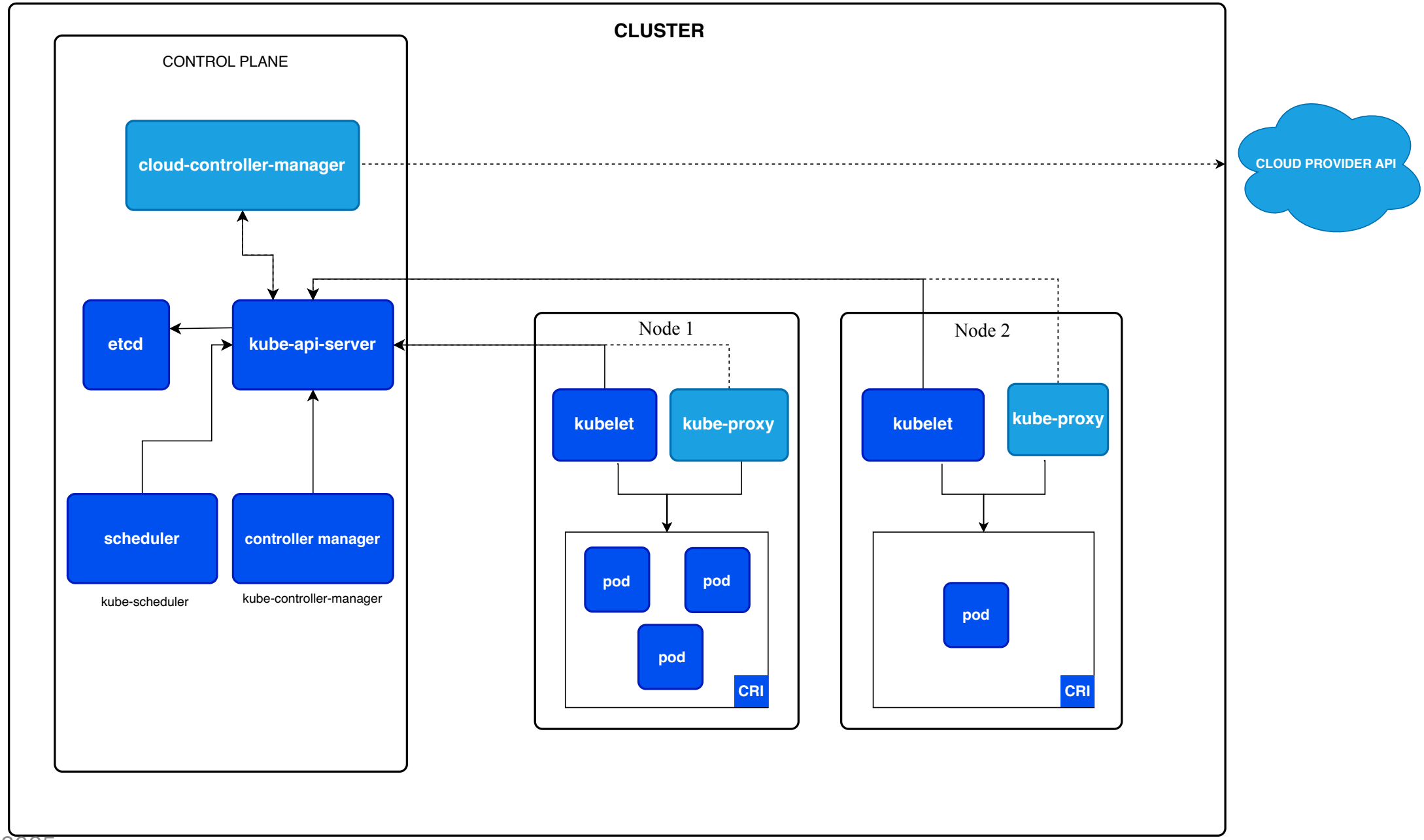
- `Deployment` — декларативные обновления, масштабирование, стратегии релизов
- `ReplicaSet` — поддержание нужного числа одинаковых подов
- `StatefulSet` — упорядоченное развертывание/масштабирование, стабильные идентификаторы и хранилище
- `DaemonSet` — по одному поду на каждом (или выбранных) узле (логирование, мониторинг и т.п.)

Типы ресурсов

- `Pods` — наименьшая единица развертывания (1+ контейнеров)
- `Services` — стабильная точка доступа к набору подов (IP/имя, балансировка)
- `Ingress` — внешний HTTP/HTTPS-доступ и маршрутизация к сервисам
- `ConfigMaps` — неконфиденциальные конфигурации (key/value)
- `Secrets` — конфиденциальные данные (пароли, токены, ключи)

Kubernetes architecture




CLUSTER
CONTROL PLANE
cloud-controller-manager
CLOUD PROVIDER API
etcd
kube-api-server
scheduler
kube-scheduler
controller manager
kube-controller-manager
Node 1
kubelet
kube-proxy
pod
pod
pod
CRI
Node 2
kubelet
kube-proxy
pod
CRI

Узлы (Nodes)

- Узлы — физические/виртуальные машины, где запускаются приложения (Pods)
- Компоненты на узле: kubelet, container runtime, kube-proxy
- Типы узлов:
  - Control Plane (Master) — управляющие
  - Worker — выполняющие рабочие нагрузки

Control Plane

- Назначение: управление и координация всего кластера (обработка API-запросов, хранение состояния и планирование подов)
- Основные компоненты:
  - kube-apiserver
  - kube-controller-manager
  - etcd (распределённое key-value хранилище)
  - kube-scheduler

Worker Nodes (Data Plane)

- Назначение: запуск Pods, сетевой прокси и агентов кластера
- Основные компоненты:
  - kubelet — управляет жизненным циклом Pod/Container
  - kube-proxy — Service networking и балансировка (может быть eBPF в CNI)
  - Container Runtime — containerd/CRI-O (через CRI)
  - Агент CNI/CSI (DaemonSet) — сеть и тома на каждом узле

Ресурсы и состояния узла

- Ресурсы:
  - capacity/allocatable — всего и доступно планировщику (с учётом kube/system-reserved)
  - Overcommit через requests/limits у Pod; QoS-класс влияет на OOM-поведение
- Состояния и условия:
  - Ready/NotReady, MemoryPressure, DiskPressure, PIDPressure, NetworkUnavailable
  - Узлы с проблемами могут быть помечены NoSchedule/NoExecute

QoS классы Pod'ов

Класс	Как получить	Когда использовать
Guaranteed	У всех контейнеров задано `requests == limits` по CPU и памяти	Stateful/mission-critical сервисы, где недопустим OOM/eviction
Burstable	Есть `requests`, но `limits` выше или заданы не у всех ресурсов	Большинство веб/API сервисов, которым нужны временные всплески
BestEffort	Нет `requests/limits` ни по CPU, ни по памяти	Инструменты, batch job'ы, которые можно прервать

Как kubelet ведёт себя при давлении ресурсов

- Kubelet следит за `MemoryPressure`, `DiskPressure`, `PIDPressure` и при нехватке ресурсов инициирует eviction.
- Если контейнер превысил `request`, а узел испытывает давление, **весь Pod** становится кандидатом на eviction и его контейнеры завершаются; контроллер создаёт замену (обычно на другом узле).
- Порядок завершения: `BestEffort → Burstable → Guaranteed`, внутри класса — по степени превышения лимитов (см. официальную документацию).
- Даже Guaranteed-поды могут быть завершены, но только после исчерпания менее «дорогих» классов.

Влияние QoS на планирование и preemption

- kube-scheduler учитывает только `requests` при подборе узла; QoS — это производная от requests/limits.
- При **preemption** (когда кластеру не хватает ресурсов для новых подов) scheduler не смотрит на QoS-класс: любой под может быть вытеснен.
- Eviction на узле и scheduler preemption — разные механизмы: QoS влияет на первое, но не на второе.
- Практика:
  - Guaranteed — фиксируем ресурсы для SLA-критичных workload'ов (базы данных, системные контроллеры).
  - Burstable — баланс эффективность/стабильность.
  - BestEffort — полезны только там, где потеря пода не критична.

Pods:

Pod — это наименьший и самый простой объект Kubernetes, представляющий один экземпляр запущенного приложения. Он может содержать один или несколько тесно связанных контейнеров, которые совместно используют сеть и хранилище. Pod'ы эфемерны: если Pod выходит из строя, Kubernetes заменяет его новым Pod'ом, который может иметь другой IP-адрес.

Containers:

Контейнеры — это легковесные, переносимые и изолированные единицы, которые запускают прикладной код. Kubernetes обычно использует `Containerd` или другие среды выполнения контейнеров для управления контейнерами внутри Pod'ов. Kubernetes управляет контейнерами в Pod'ах, выполняя их создание, масштабирование и завершение работы.

ReplicaSets:

ReplicaSets обеспечивают запуск заданного числа идентичных реплик Pod'ов в любой момент времени. При сбое или завершении Pod'а ReplicaSet создаёт новый Pod, чтобы поддерживать желаемое состояние.
Это помогает достигать высокой доступности и отказоустойчивости, сохраняя требуемое количество Pod'ов даже при падениях или удалении.

Deployments:

`Deployment` управляет жизненным циклом приложений: отвечает за выпуск новых версий, масштабирование и откаты. Он обеспечивает декларативные обновления приложений и поддерживает желаемое состояние `Pod`-ов с помощью `ReplicaSet`.

`Deployment` позволяет декларативно задавать образ контейнера, число реплик и стратегии обновления, автоматизируя процесс обновлений.

Что такое kubeconfig?

kubeconfig — это конфигурационный файл, который содержит информацию о:

- кластерах Kubernetes (адреса API-серверов),
- пользователях (учетные данные),
- контекстах (наборы настроек, связывающие кластер и пользователя),
- namespace'ах по умолчанию.

> Он используется CLI-инструментом kubectl (а также другими клиентами) для подключения к Kubernetes-кластерам и выполнения команд.

Где его найти?

По умолчанию Kubernetes ожидает файл конфигурации kubeconfig в стандартном расположении внутри домашней директории пользователя:

`~/.kube/config`

Однако вы можете использовать альтернативный файл, указав путь через:

- переменную среды `KUBECONFIG`,
- флаг `–kubeconfig` при запуске kubectl.

Kubeconfig

5 Основных компонентов

1. **certificate-authority-data**: Cluster CA
2. **server**: Cluster endpoint (IP/DNS of the master node)
3. **name**: Cluster name
4. **user**: name of the user/service account.
5. **token**: Secret token of the user/service account.

Example Kubeconfig File

```yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: <ca-data-here>
    server: https://your-k8s-cluster.com
  name: <cluster-name>
contexts:
- context:
    cluster:  <cluster-name>
    user:  <cluster-name-user>
  name:  <cluster-name>
current-context:  <cluster-name>
kind: Config
preferences: {}
users:
- name:  <cluster-name-user>
  user:
    token: <secret-token-here>
```

```
$ kubectl get pods
  --server=https://your-api-server.com
  --certificate-authority=ca.crt
  --client-certificate=user.crt
  --client-key=user.key
```

kubeconfig

```
apiVersion: v1
kind: Config
clusters:
  --------
  --------
contexts:
  --------
  --------
users:
  --------
  --------
```

Зачем нужен kubeconfig?

Файл kubeconfig позволяет:

- Подключаться к нескольким кластерам с одного устройства.
- Надёжно хранить ключи доступа и TLS-сертификаты.
- Упрощать процессы аутентификации и авторизации.
- Обеспечивать гибкость при работе с различными окружениями (например, dev, staging, production).

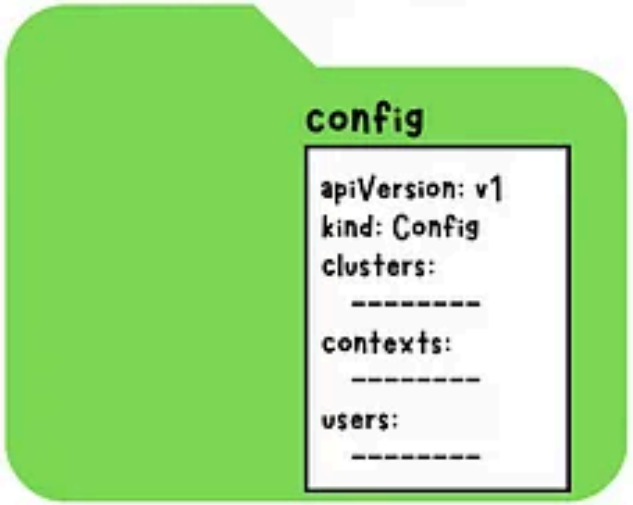
Kubeconfig File Path

default kubeconfig file path

$HOME/.kube/config

non-default kubeconfig file path

/<path_to_kubeconfig>



use the --kubeconfig flag to specify a different location for the kubeconfig file to run commands.

```
$ kubectl --kubeconfig=path/to/kubeconfig get pods
```

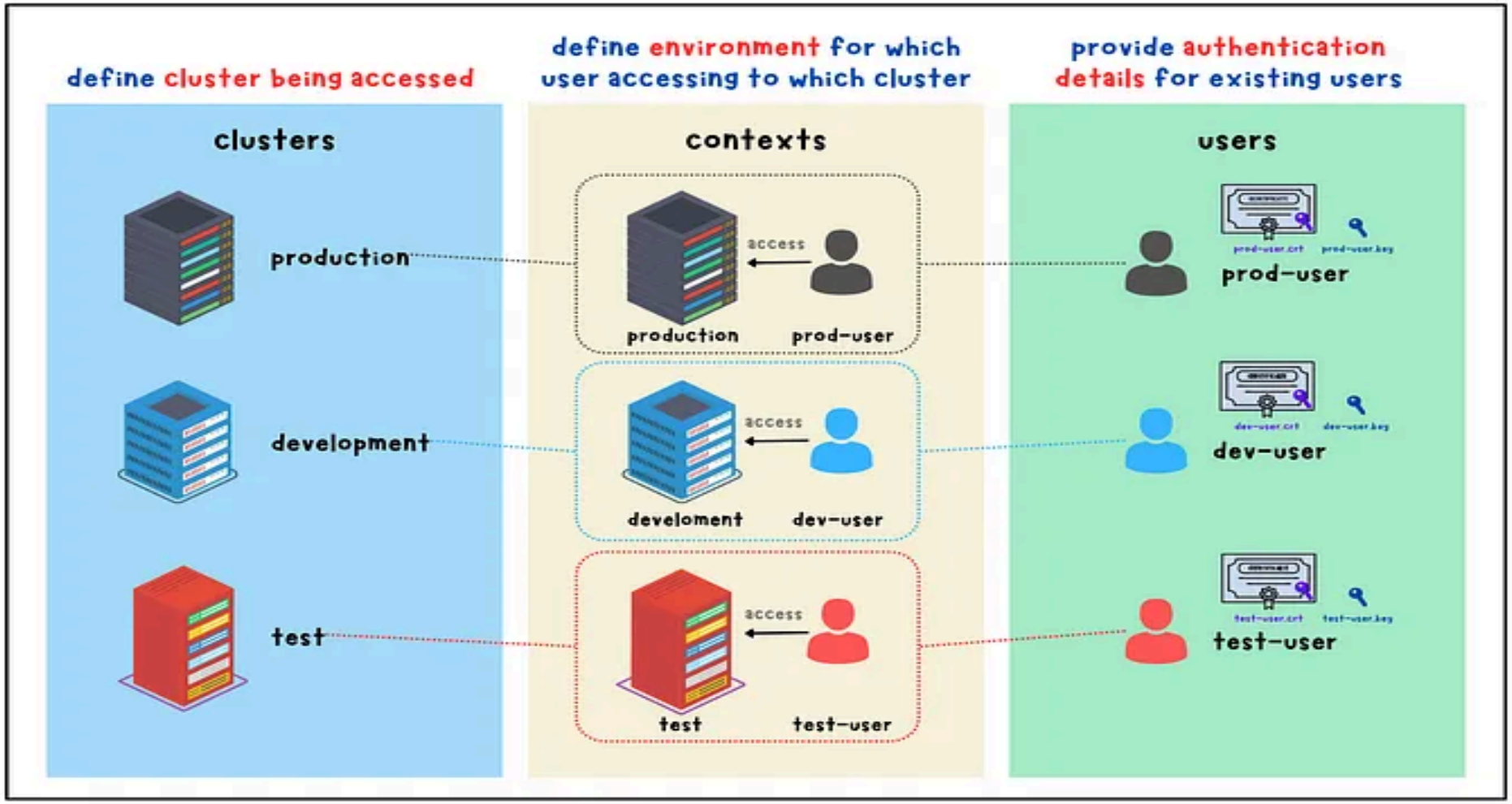
Как управлять несколькими файлами kubeconfig?

Все файлы kubeconfig можно хранить в каталоге `$HOME/.kube`. Чтобы подключиться к нужному кластеру, переключайте контекст.

`current-context` определяет, какой контекст используется по умолчанию при запуске kubectl.

KubeConfig File

$HOME/.kube/config



List all cluster contexts

```
kubectl config get-contexts -o=name
```

Set the current context

```
kubectl config use-context <cluster-name>
```

Using Kubeconfig File With Kubectl

```
kubectl get nodes --kubeconfig=$HOME/.kube/dev_cluster_config
```

Connect with the KUBECONFIG environment variable

```
export KUBECONFIG=$HOME/.kube/dev_cluster_config
```

Советы по безопасности

- Не передавайте kubeconfig по незащищённым каналам.
- Храните файл в защищённой директории с ограниченными правами доступа.
- Удаляйте устаревшие или неиспользуемые контексты и учетные записи.

Практика

LAB: Install and configure K8s cluster

Lab instructions
https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_minikube_podman.md#learning-environment-preparation

Time allocated - 30min

Работа с Kubernetes

kubectl: первые шаги

- `kubectl` — основной инструмент общения с API Kubernetes.
- Общий синтаксис: `kubectl <command> [TYPE] [NAME] [flags]`.
- Базовые команды: `get`, `create`, `apply`, `delete`, `describe`, `logs`.
- Подсказки внутри CLI: `kubectl --help` или `kubectl <command> --help`.

Контексты и kubeconfig

- Контекст = `cluster` + `user` + `namespace`; хранится в `~/.kube/config`.
- Проверяйте активный контекст: `kubectl config current-context`.
- Переключение: `kubectl config use-context <name>`, список — `get-contexts`.
- Создавайте отдельные контексты для dev/prod (`kubectl config set-context ... --namespace=<ns>`).
- Хинт: утилиты вроде `kubectx/kubens` ускоряют переключения.

Управление кластерами

- Мердж kubeconfig:
  - Unix/macOS: export KUBECONFIG=$HOME/.kube/dev$:HOME/.kube/prod
  - Windows (PowerShell): $env:KUBECONFIG="$HOME.kube\dev;$HOME.kube\prod"
  - Просмотреть общий: kubectl config view --flatten
  - Сохранить единый: kubectl config view --flatten > ~/.kube/config
- Явные переопределения (для многокластерной работы):
  - kubectl get nodes --context=prod-eu1
  - Глобальные флаги поддерживаются: --context, --cluster, --user

Управление кластерами

- Временные проверки без правки файлов:
  - kubectl get po -n staging --context=prod-eu1
  - kubectl --kubeconfig ~/.kube/dev get ns
- Безопасность операций:
  - Проверить активный контекст: kubectl config current-context
  - Задать namespace по умолчанию для текущего контекста: kubectl config set-context --current --namespace=dev
  - Перед удалениями/изменениями всегда сверяйте контекст и namespace (-n)
- Хинт: утилиты kubectx/kubens ускоряют переключение контекстов и пространств имён.

Получение и создание ресурсов

- `kubectl get <type>` (опции `-o wide|yaml|json|name`) помогает быстро увидеть состояние объектов.
- Используйте `-o yaml` чтобы сохранить исходный объект в файл и версионировать его.
- Создание: `kubectl create namespace dev` или `kubectl create -f manifests/`.
- Быстрые эксперименты допустимо делать через `kubectl run` / `kubectl create`; затем фиксируйте итоговый манифест.

Declarative vs imperative

- Declarative (apply/GitOps)
  - Описывайте желаемое состояние в YAML/Helm/Kustomize; kubectl apply приводит кластер к нему.
  - apply использует three-way merge и сохраняет исходник в аннотации kubectl.kubernetes.io/last-applied-configuration.
  - Если ресурс создан не через apply, первая попытка может перетереть поля. Решение: `kubectl apply set-last-applied -f <file>` или `server-side apply (--server-side)`.
  - Полезно: `kubectl diff -f <dir>, kubectl apply -f <dir> --prune -l app=<label>`.

Declarative vs imperative

- Imperative (быстрые изменения)
  - Для экспериментов: `kubectl run/create/expose/scale/set image/patch`; затем зафиксировать манифесты в Git.
  - `kubectl edit <type> <name>` удобно для отладки, но изменения не сохраняются в git репо — избегать в проде.
- Рекомендуемый паттерн
  - Правим YAML в репозитории → kubectl diff/apply (или GitOps-контроллер) → ревью изменений через Git.

Declarative vs imperative

Пример: создание Pod с args (imperative)

```
kubectl run hello \
  --image=busybox:1.36 --restart=Never \
  --command -- sh -c 'echo Hello from args && sleep 10'

# Сгенерировать манифест из команды
kubectl run hello \
  --image=busybox:1.36 --restart=Never \
  --command -- sh -c 'echo Hello from args && sleep 10' \
  --dry-run=client -o yaml > pod-hello.yaml
```

Declarative vs imperative

Пример: создание Pod с args (declarative) с помощью kubectl apply

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello
spec:
  restartPolicy: Never
  containers:
  - name: app
    image: busybox:1.36
    command: ["sh", "-c"]
    args: ["echo Hello from args && sleep 10"]
```

Диагностика: describe и logs

- `kubectl describe <type> <name>` выводит события и состояние — идеален для поиска причин CrashLoopBackOff.
- `kubectl get pod <name> -o yaml` — быстрый способ увидеть итоговую спецификацию (включая default значения).
- `kubectl logs <pod>` (или `-c <container>`) показывает stdout/stderr; добавляйте `-f` для tail.
- После отладки не забудьте очистить окружение: `kubectl delete pod mypod`, `kubectl delete namespace dev`.

Доступ к workloads

- `kubectl exec <pod> -- <cmd>` позволяет выполнить команду внутри контейнера; для shell используйте `-it`.
- При нескольких контейнерах указывайте `-c <container>` чтобы попасть в нужный.
- `kubectl proxy` открывает API-сервер локально — полезно для отладки сервисов и dashboard.
- Планируйте выход: закрывайте proxy/exec-сессии и снимайте временные ресурсы после проверки.

kubectl resource shortnames

- Pods: po
- Services: svc
- Endpoints: ep
- EndpointSlices: eps
- Namespaces: ns
- Nodes: no
- ConfigMaps: cm
- Secrets: secret
- ServiceAccounts: sa
- Events: ev
- ResourceQuotas: quota

kubectl resource shortnames

- Deployments: deploy
- ReplicaSets: rs
- ReplicationControllers: rc
- StatefulSets: sts
- DaemonSets: ds
- Jobs: job
- CronJobs: cj
- HorizontalPodAutoscalers: hpa
- VerticalPodAutoscalers: vpa
- PodDisruptionBudgets: pdb
- Ingresses: ing

Примеры

```
# Examples
kubectl get po,svc,deploy
kubectl get rs,ds,sts -A
```

```
kubectl get pv,pvc,sc
kubectl get ing,netpol -n web
kubectl describe pdb my-app-pdb
```

Практика

LAB: Working with K8s Resources using kubectl

Манифесты

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Обязательные поля

В файле .yaml создаваемого объекта Kubernetes необходимо указать значения для следующих полей:

`apiVersion` — используемая для создания объекта версия API Kubernetes
`kind` — тип создаваемого объекта
`metadata` — данные, позволяющие идентифицировать объект (name, UID и необязательное поле namespace)
`spec` — требуемое состояние объекта

Ресурсы

Ресурсы в Kubernetes — это объекты, которые управляют различными аспектами работы приложений. Основные ресурсы включают:

`Pods`: Основные единицы выполнения в Kubernetes, содержащие один или несколько контейнеров.
`Services`: Определяют способ доступа к Pod'ам и обеспечивают балансировку нагрузки.
`ConfigMaps` и `Secrets`: Используются для хранения конфигурационных данных и секретов.
`Persistent Volumes` (**PV**s) и `Persistent Volume Claims` (**PVC**s): Управляют постоянным хранилищем данных.

Жизненный цикл приложений

Жизненный цикл приложений в Kubernetes включает несколько этапов:

`Создание` : Определение и применение манифестов для создания объектов.

`Обновление` : Обновление манифестов и применение изменений для обновления объектов.

`Масштабирование` : Изменение количества реплик для управления нагрузкой.

`Удаление` : Удаление объектов, когда они больше не нужны.

Пример обновления Deployment:

```
kubectl apply -f deployment.yaml
```

Пример масштабирования Deployment:

```
kubectl scale deployment my-app --replicas=5
```

Механика запуска и взаимодействия контейнеров в Kubernetes. Основы Services, Labels и Selectors

Pods

`Pod` — это основная единица развертывания в Kubernetes, которая может содержать один или несколько контейнеров. Контейнеры внутри Pod'а разделяют сеть и хранилище, что позволяет им легко взаимодействовать друг с другом.

Пример манифеста Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image:latest
    ports:
    - containerPort: 80
```

Pods: Shared Resources and Namespaces

- **Pod**: Группа из одного или нескольких контейнеров, разделяющих ресурсы и среду выполнения.
- **Общие ресурсы** (через пространства имён Linux):
  - **Сеть (net):** общий IP-адрес, диапазон портов, таблица маршрутизации
  - **Процессы (pid):** изолированное дерево процессов
  - **Монтирование (mnt):** общие файловые системы и тома
  - **UTS:** общее имя хоста
  - **IPC:** общие Unix-сокеты и разделяемая память
- **Контейнер pause**:
  - Позволяет контейнерам совместно использовать сеть и файловую систему
  - Поддерживает сетевое пространство имён в активном состоянии при перезапусках Pod'а

Diagram: Pod and Shared Namespaces

```
+-----------------------------+
|            Pod              |
|  +-----------------------+  |
|  |    Pause Container    |  |  <-- Holds network, IPC, UTS, mnt namespaces
|  +-----------------------+  |
|  |   App Container(s)    |  |  <-- Join pause container's namespaces
|  +-----------------------+  |
+-----------------------------+
      |          |          |
      |          |          |
  Shared Net  Shared FS  Shared IPC
  (IP/ports)  (Volumes)  (Sockets)
```

Pods and Shared Networking

- **У каждого Pod собственное сетевое пространство имён:** Все контейнеры в Pod используют общий IP-адрес, диапазон портов и таблицу маршрутизации
- **Внешний доступ к контейнерам в Pod осуществляется по IP-адресу Pod и порту контейнера** Пример: `10.0.10.15:80` или `10.0.10.15:5000`
- **Взаимодействие контейнеров внутри Pod — через localhost и порт:** Пример: `localhost:5000` для обращения к sidecar или вспомогательному контейнеру
- **Сеть Pod «плоская»:** Каждый Pod может напрямую обмениваться трафиком с любым другим Pod в кластере
- **По умолчанию сеть Pod открыта:** Используйте сетевые политики Kubernetes (NetworkPolicy) для защиты

Diagram: Pod Networking and Communication

```
+-----------------------------+
|           Pod               |
|  IP: 10.0.10.15             |
|  +-----------------------+  |
|  |  App Container        |  |  <-- Exposes port 80
|  +-----------------------+  |
|  |  Sidecar Container    |  |  <-- Exposes port 5000
|  +-----------------------+  |
+-----------------------------+
      |            |
      |            |
  10.0.10.15:80   10.0.10.15:5000
      |            |
      +------------+
           |
   [Pod Network: Flat, routable]
           |
   <----> Other Pods (any IP)
```

Kubernetes Objects: Pods Lifecycle & Immutability

- **Pod'ы — эфемерны:** создаются, работают и завершаются; при сбое Pod Kubernetes создаёт новый с другим идентификатором и IP-адресом.
- **Неизменяемость:** после запуска Pod'ы не изменяются и не обновляются «на месте».
- **Без ручных изменений:** не следует подключаться в запущенный Pod для обновления конфигурации; вместо этого его заменяют новым Pod'ом.

Deploying Pods: Static vs. Managed

- **Прямое развертывание:**
  - Используйте манифест Pod (YAML) для создания «статических» Pod'ов.
  - Управляются kubelet на узле, самовосстановление отсутствует.
- **Развертывание, управляемое контроллерами:**
  - Используйте контроллеры (например, Deployment) для Pod'ов.
  - Обеспечивает самовосстановление, масштабирование и постепенные обновления (rolling updates).

Multi-Container Pods: Separation of Concerns

- **Принцип проектирования:** Каждый контейнер должен иметь одну, чётко определённую область ответственности.
- **Преимущества:**
  - Более компактные и простые контейнеры
  - Проще устранение неполадок и повторное использование
- **Когда объединять контейнеры:**
  - Тесно связанные функции, которым нужно совместно использовать данные (например, через общий том).

Example: Multi-Container Pod Pattern

- **Сценарий:**
  Один контейнер подтягивает контент из репозитория («sync»-контейнер).
  Другой контейнер отдаёт контент как веб-страницу («web»-контейнер).
- **Общий том:** Оба контейнера получают доступ к одним и тем же данным через общий том.

Diagram: Multi-Container Pod with Shared Volume

```
+-------------------------------+
|              Pod              |
|  +-------------------------+  |
|  |   sync container        |  |  <-- Pulls content, writes to shared volume
|  +-------------------------+  |
|  |   web container         |  |  <-- Serves content from shared volume
|  +-------------------------+  |
|         Shared Volume         |
+-------------------------------+
```

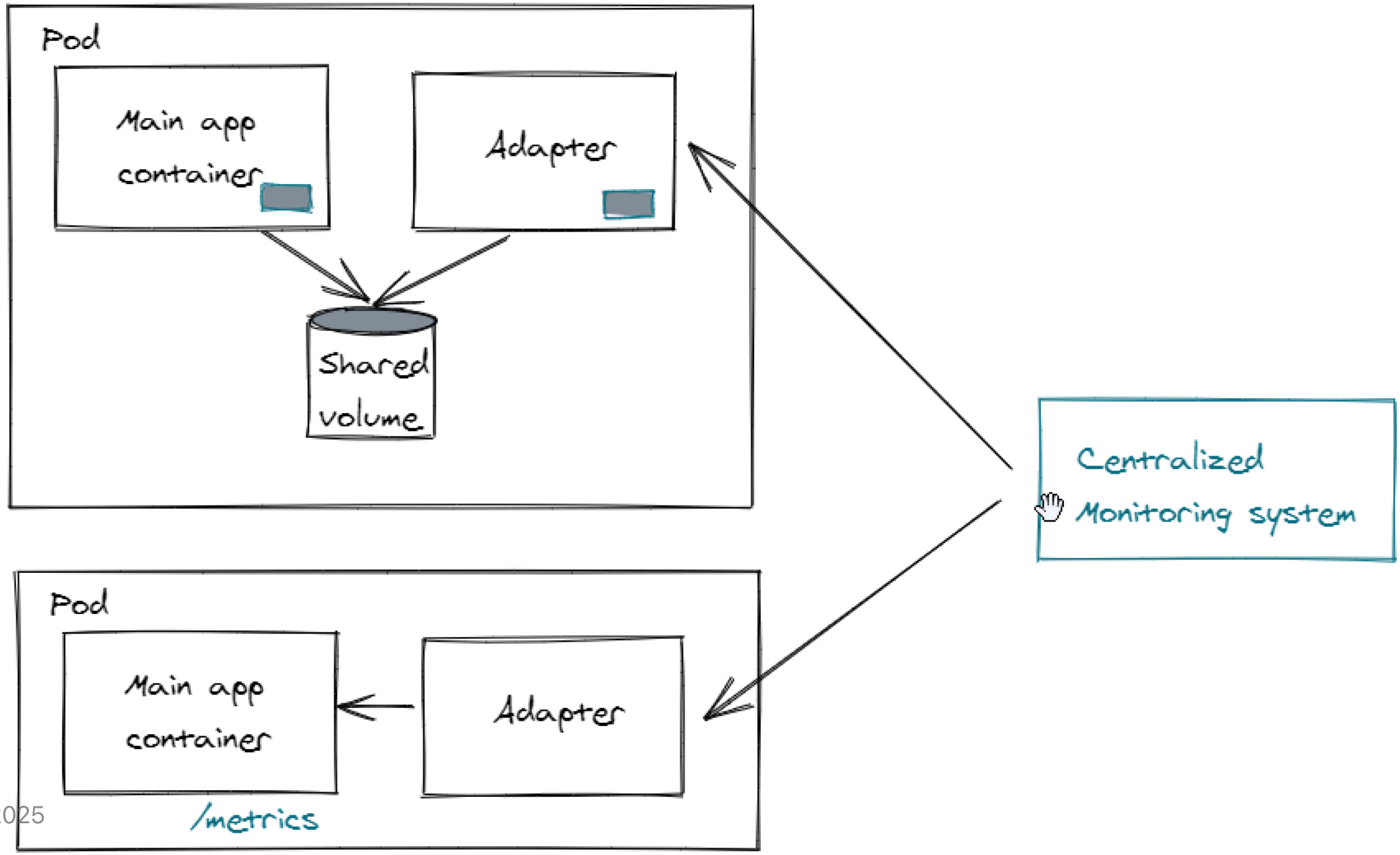
Multi-Container Pod Patterns

Паттерн Sidecar: - Основной контейнер приложения + контейнер sidecar для вспомогательных задач. Часто используется в архитектурах service mesh (например, шифрование трафика, метрики)
Паттерн Adapter: - Вспомогательный контейнер преобразует выходные данные для внешних систем. Пример: преобразует логи NGINX в формат, понятный Prometheus
Паттерн Ambassador: - Вспомогательный контейнер выступает посредником соединения с внешними системами. Основное приложение отправляет данные в Ambassador, который обрабатывает внешние коммуникации.
Паттерн Init: - Специальный init-контейнер запускается один раз до старта основного приложения.Готовит окружение (например, выставляет права, клонирует репозитории)

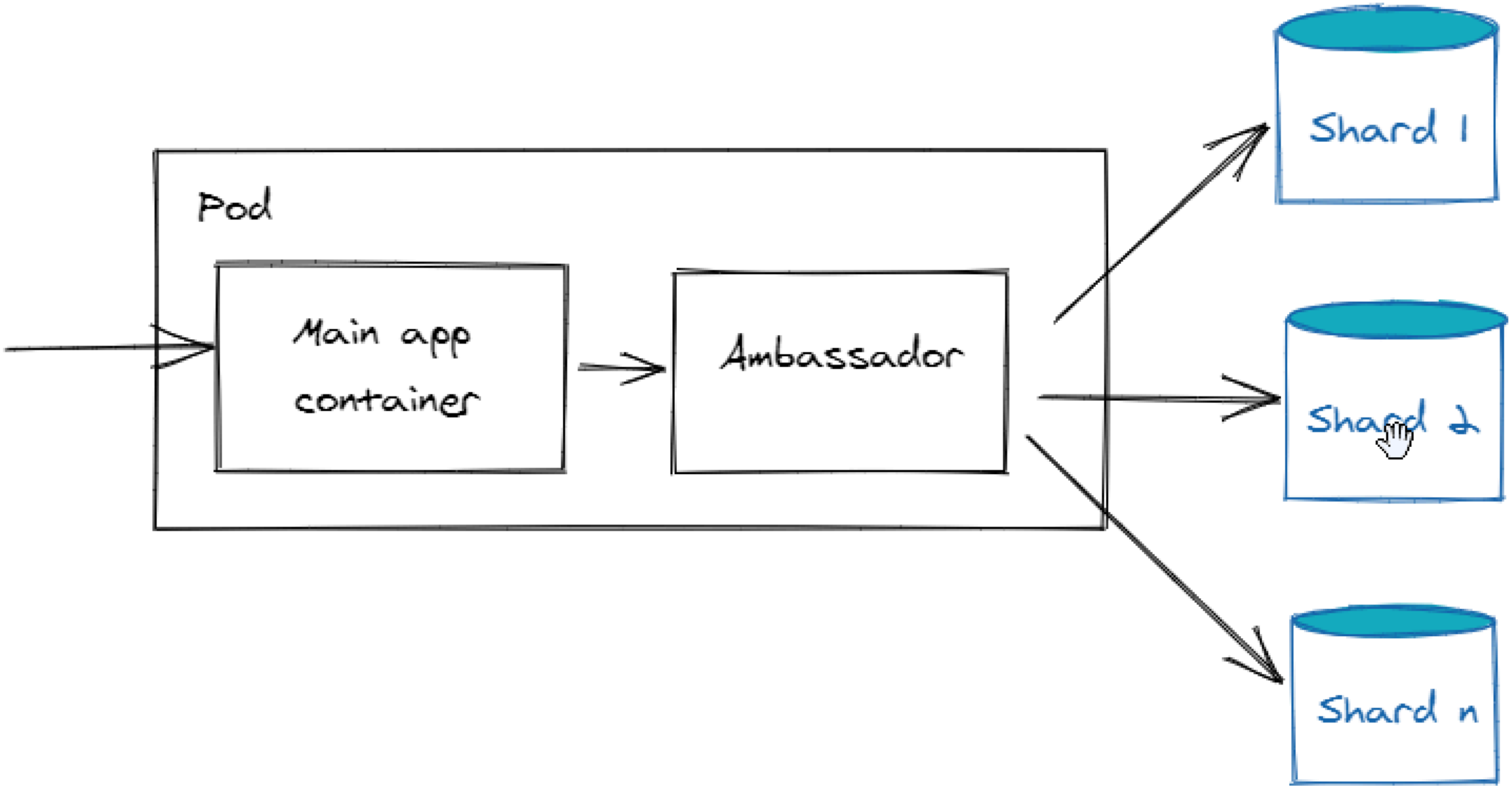
Diagram: Multi-Container Pod Patterns

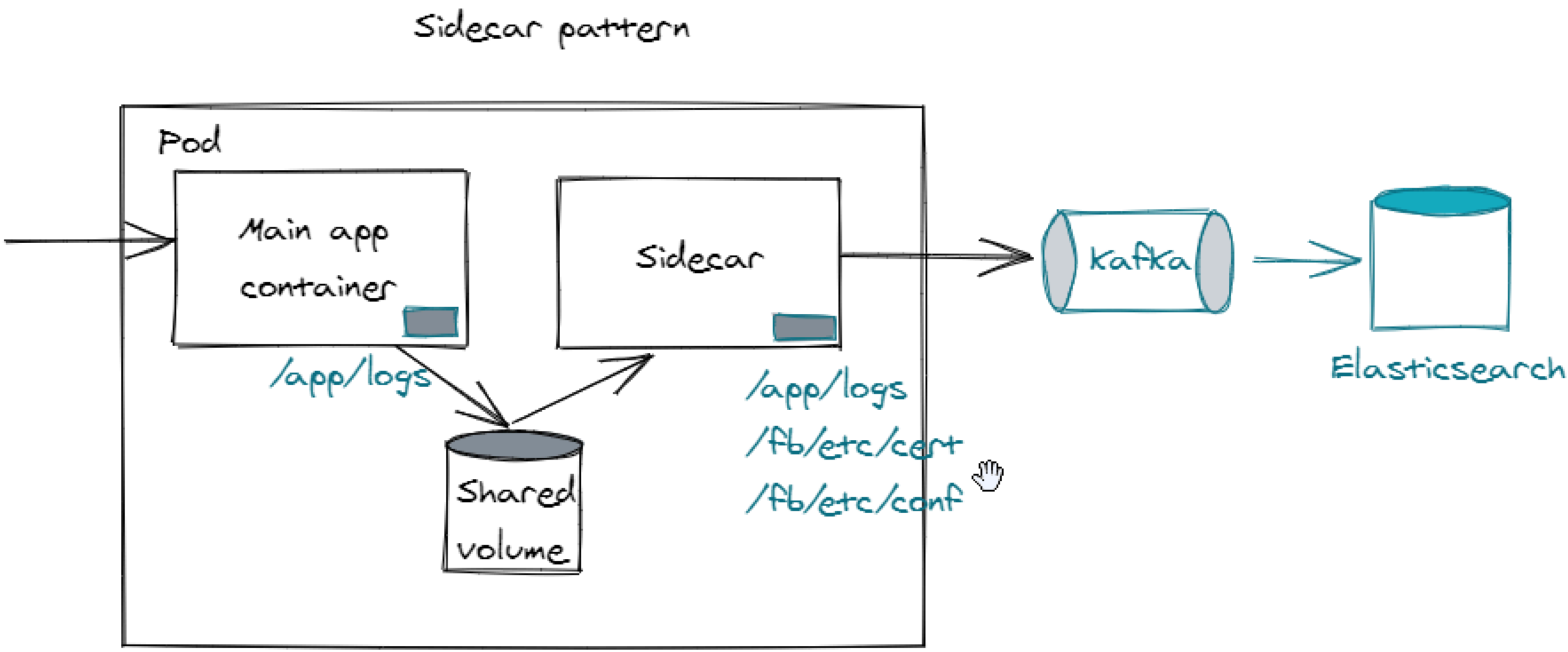
```
+-------------------------------+
|             Pod               |
|  +-------------------------+  |
|  |   Init Container        |  |  <-- Runs first, prepares environment
|  +-------------------------+  |
|  |   Main App Container    |  |  <-- Core application logic
|  +-------------------------+  |
|  |   Sidecar/Adapter/      |  |  <-- Helper containers for support tasks
|  |   Ambassador Container  |  |
|  +-------------------------+  |
+-------------------------------+
```

Adapter pattern



Ambassador




Sidecar pattern
Pod
Main app container
Sidecar
/app/logs
Shared volume
/app/logs
/fb/etc/cert
/fb/etc/conf
kafka
Elasticsearch

Практика

LAB: Working with K8s Resources

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#getting-started-core-kubernetes-resources

- Practice hands-on with Kubernetes core resources:
  - **Pods:** Deploy and manage single or multi-container Pods.
  - **Deployments:** Manage rolling updates, scaling, and self-healing.
- Use interactive labs for guided exercises:
  - Pods Lab
  - Deployments Lab
- **Time allocated:** 30 minutes

Services

Основные типы Services:

`ClusterIP` : Доступен только внутри кластера.

`NodePort` : Доступен снаружи кластера через IP-адреса узлов и определенный порт.

`LoadBalancer` : Использует внешний балансировщик нагрузки для доступа к Service.

`ExternalName` : Преобразует DNS-запросы в указанный внешний DNS-имя.

Пример манифеста Service:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

Labels и Selectors

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-app
    environment: production
spec:
  containers:
  - name: my-container
    image: my-image:latest
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
    environment: production
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

Масштабирование и обновление приложений в Kubernetes

Масштабирование

Масштабирование в Kubernetes позволяет увеличивать или уменьшать количество реплик ваших приложений в зависимости от нагрузки.

Ручное масштабирование

Вы можете вручную изменить количество реплик для Deployment с помощью команды kubectl scale:

```
kubectl scale deployment my-app --replicas=5
```

Автоматическое масштабирование

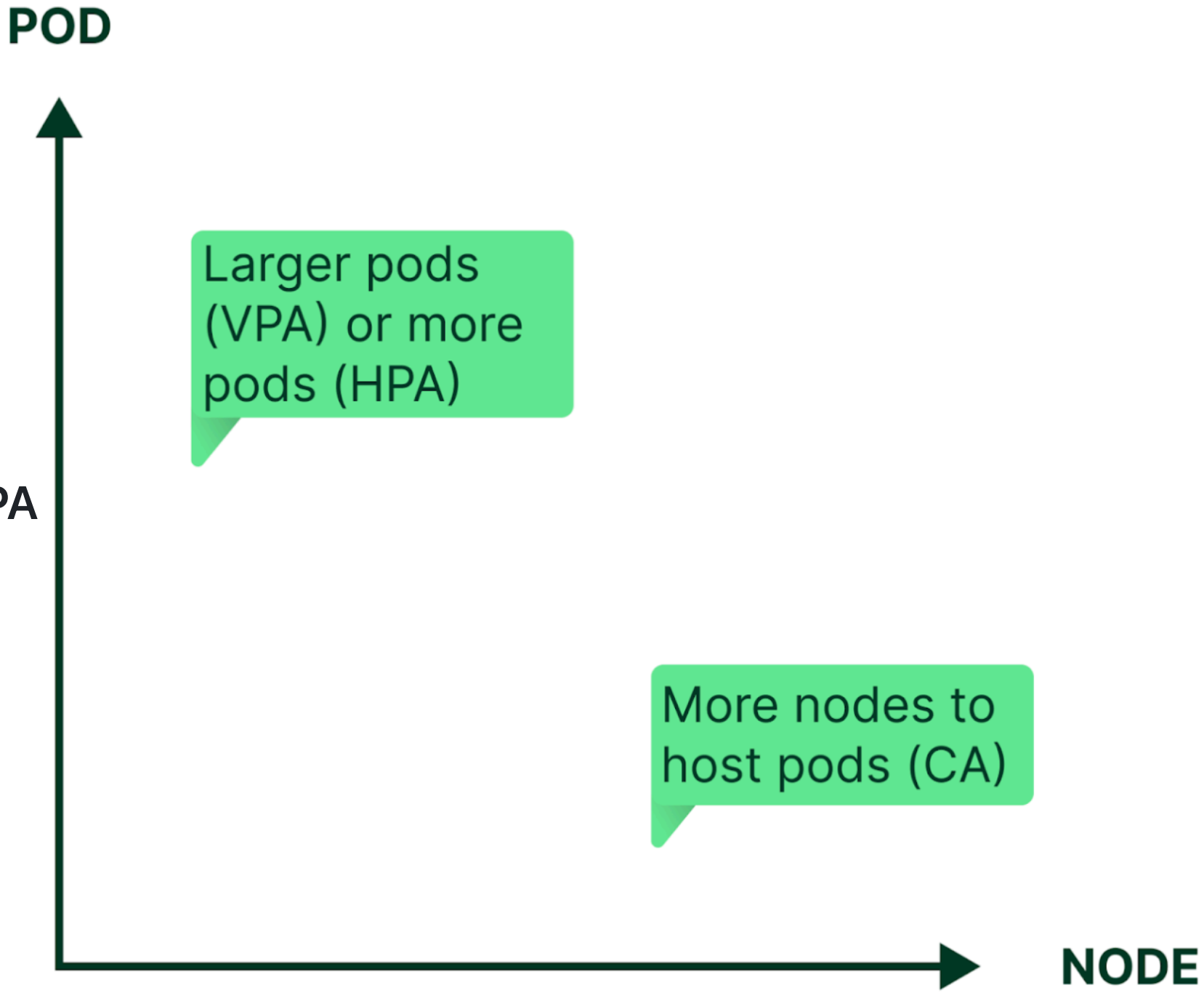
Пример манифеста HPA:

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Kubernetes autoscaling basics

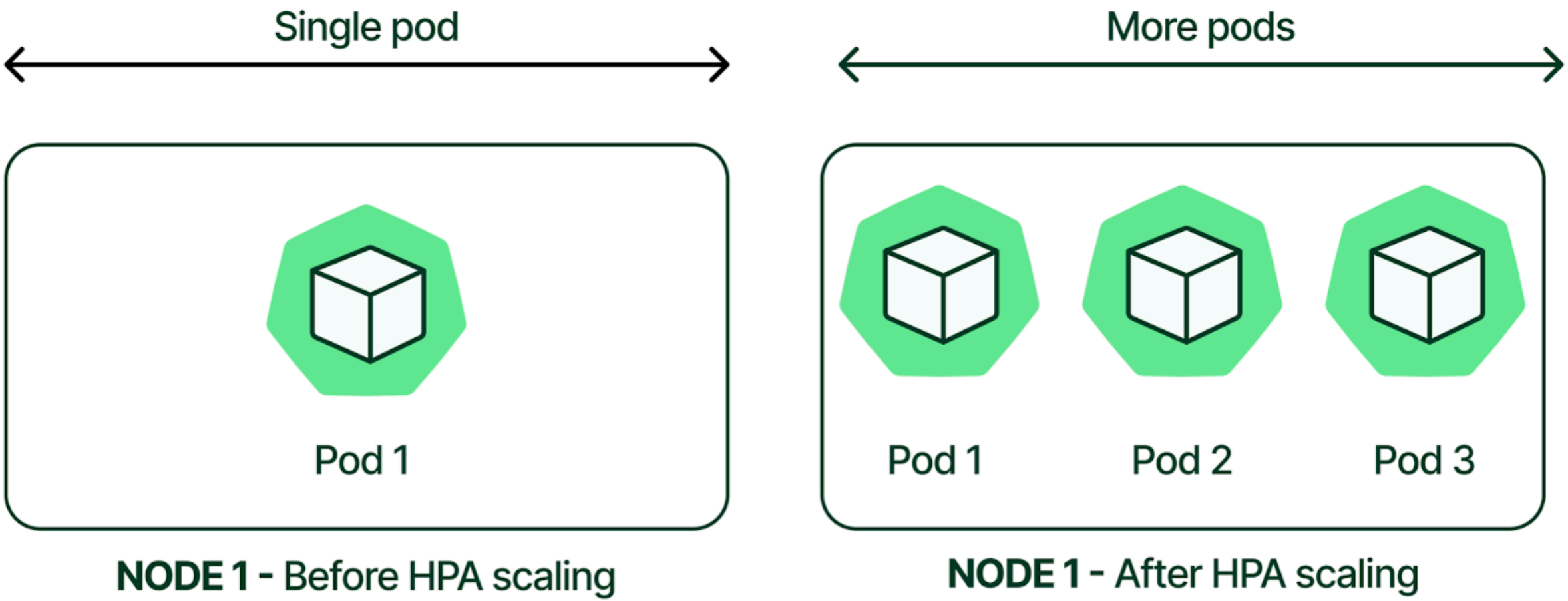
- **Horizontal Pod Autoscaler (HPA):** регулирует количество реплик приложения.
- **Cluster Autoscaler:** регулирует количество узлов в кластере.
- **Vertical Pod Autoscaler (VPA):** регулирует запросы и лимиты ресурсов контейнера.

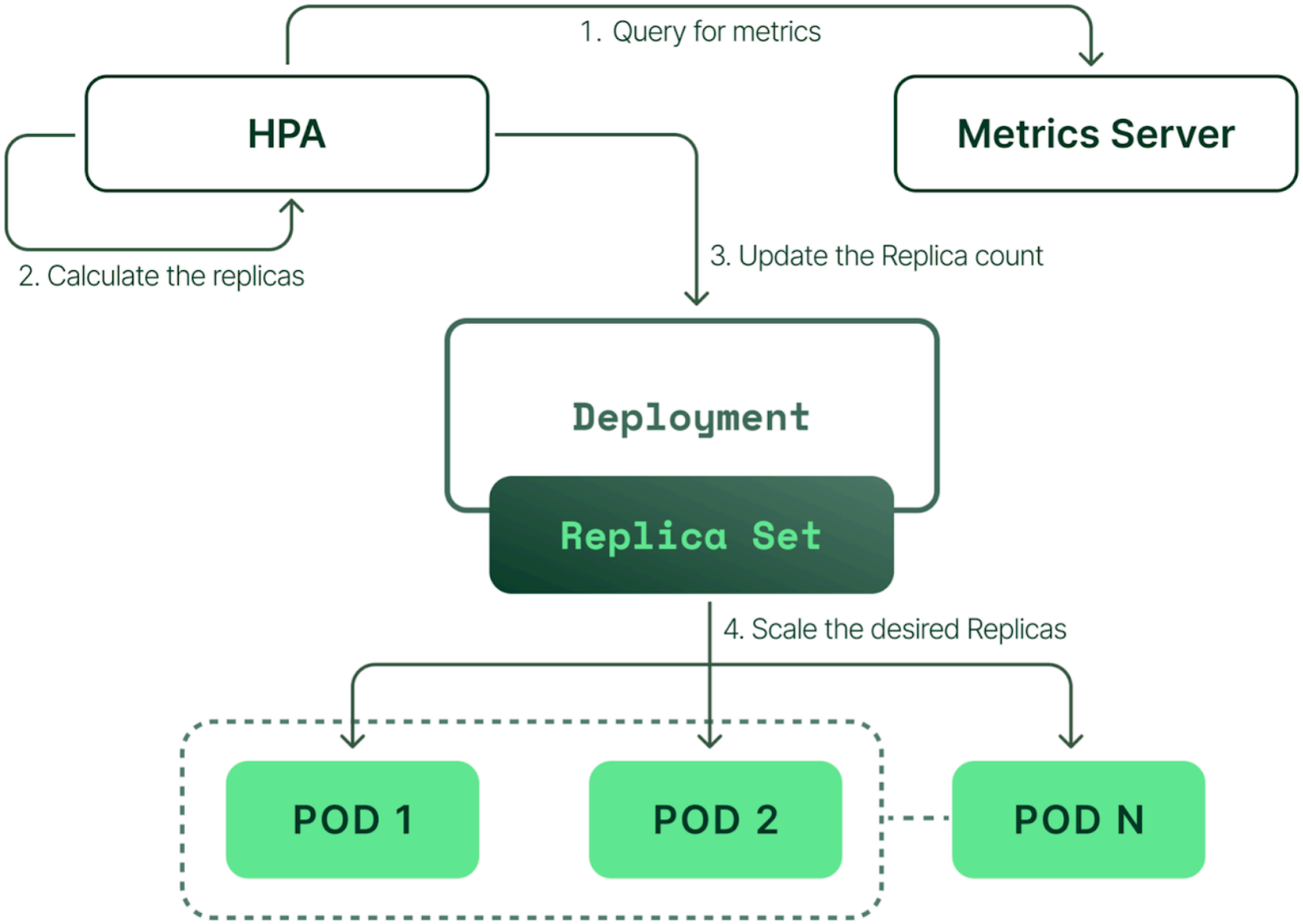
VPA vs HPA


POD
Larger pods (VPA) or more pods (HPA)
More nodes to host pods (CA)
NODE

Waste
Performance

HPA Autoscaling




1. Query for metrics
HPA
Metrics Server
2. Calculate the replicas
3. Update the Replica count
Deployment
Replica Set
4. Scale the desired Replicas
POD 1
POD 2
POD N

HPA Code example

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: image-registry
  namespace: default
spec:
  maxReplicas: 7
  minReplicas: 3
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: image-registry
  targetCPUUtilizationPercentage: 75
status:
  currentReplicas: 5
  desiredReplicas: 0
```

KEDA: Event-Driven Autoscaling

- **Kubernetes Event-Driven Autoscaling (KEDA)** дополняет HPA пользовательскими метриками и умеет масштабировать Deployments, StatefulSets и произвольные CR.
- Prometheus scaler (v1.0+) опрашивает Prometheus/VictoriaMetrics/Thanos и принимает решение по результату PromQL-запроса.
- **Activation threshold** (`activationThreshold`) не позволяет будить нагрузку пока метрика ниже заданного уровня.
- Каждый триггер можно тонко настроить (таймауты, custom headers, namespace, игнорирование `null`), поэтому KEDA легко вписывается в существующие мониторинговые ландшафты.

Prometheus scaler — ключевые параметры

- `serverAddress` — URL API Prometheus (для VictoriaMetrics используем `/select/0/prometheus`).
- `query` — PromQL, возвращающий одно значение (vector/scalar) для принятия решения.
- `threshold` — порог для масштабирования; поддерживаются дробные значения.
- `activationThreshold` — уровень, при котором KEDA активируется, default `0`.
- Дополнительные возможности:
  - `namespace` для namespaced-запросов (Thanos, multi-tenant).
  - `customHeaders` и `queryParameters` — прокидываем tenant ID или фильтры.
  - `ignoreNullValues`, `unsafeSsl`, `timeout` — контроль отказов и TLS.

TriggerAuthentication и безопасность

- `TriggerAuthentication` связывает триггеры с секретами и определяет `authModes`.
- Поддерживаются **Bearer**, **Basic**, **mTLS** и **Custom header**; можно комбинировать (например, `authModes: "tls,basic"`).
- Секреты содержат `bearerToken`, `username/password`, `cert/key/ca` и т.д.; `ca` можно задать даже без аутентификации (enterprise CA).
- Через `podIdentity` подключаются управляемые идентичности (Azure Workload Identity, EKS Pod Identity, GCP Workload Identity) без хранения ключей в секретах.

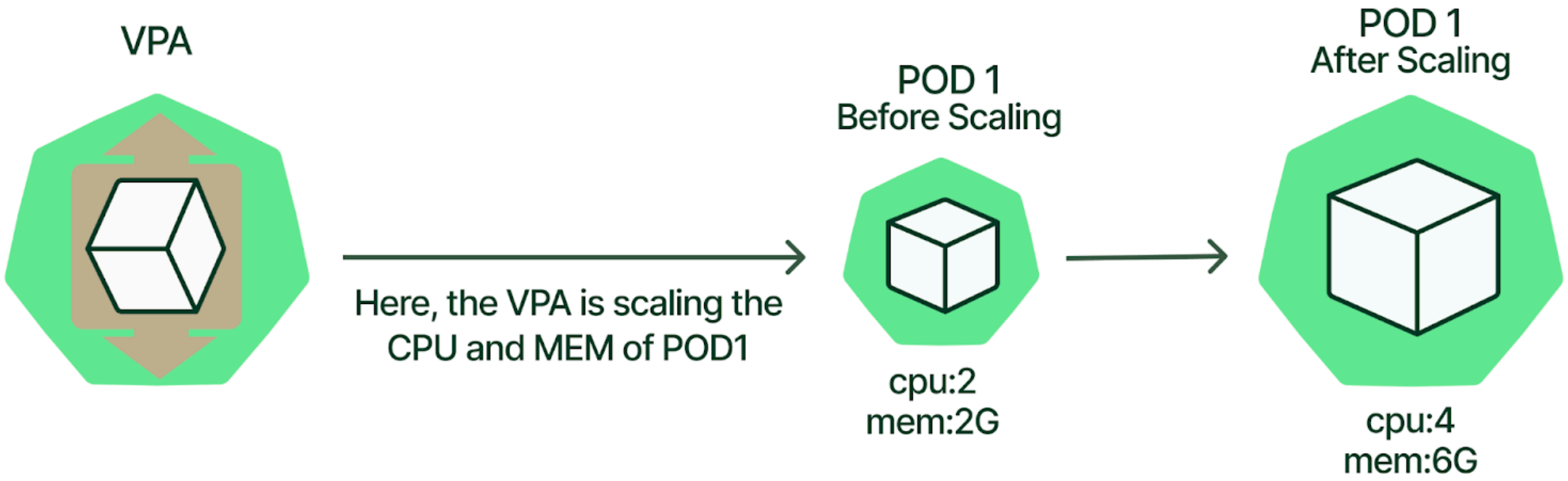
Облачные интеграции Prometheus

- **Amazon Managed Service for Prometheus (AMP)** — используем `aws` pod identity и адрес `https://aps-workspaces.<region>.amazonaws.com/workspaces/<workspaceId>`.
- **Azure Monitor Managed Service for Prometheus** — аутентификация через Azure Pod/Workload Identity, KEDA сама получает токены.
- **Google Managed Prometheus** — `serverAddress: https://monitoring.googleapis.com/v1/projects/<project>/location/global/prometheus` + GCP Workload Identity.
- Все провайдеры используют те же `prometheus`-триггеры, так что манифесты остаются единообразными.

Пример триггера Prometheus

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: prometheus-scaledobject
spec:
  scaleTargetRef:
    name: my-deployment
  triggers:
    - type: prometheus
      metadata:
        serverAddress: http://prometheus.monitoring:9090
        query: sum(rate(http_requests_total{deployment="my-deployment"}[2m]))
        threshold: '100'
        activationThreshold: '5.5'
        namespace: monitoring
      authenticationRef:
        name: keda-prom-creds
```

VPA


VPA
Here, the VPA is scaling the CPU and MEM of POD1
POD 1
Before Scaling
cpu:2
mem:2G
POD 1
After Scaling
cpu:4
mem:6G

Kubernetes VPA: Resource Configuration Types

- **Вертикальный автоскейлер подов (VPA)** управляет двумя настройками ресурсов для каждого контейнера:
  - **Requests:** минимально гарантированные ресурсы
  - **Limits:** максимально допустимые ресурсы

Diagram: VPA Resource Configuration

```
+-----------------------------+
|        Container            |
|-----------------------------|
|  Requests: 256Mi (min)      |
|  Limits:   512Mi (max)      |
+-----------------------------+
         |           |
         |           |
   Guaranteed   Resource cap
   resources    (cannot exceed)
```

Key Points

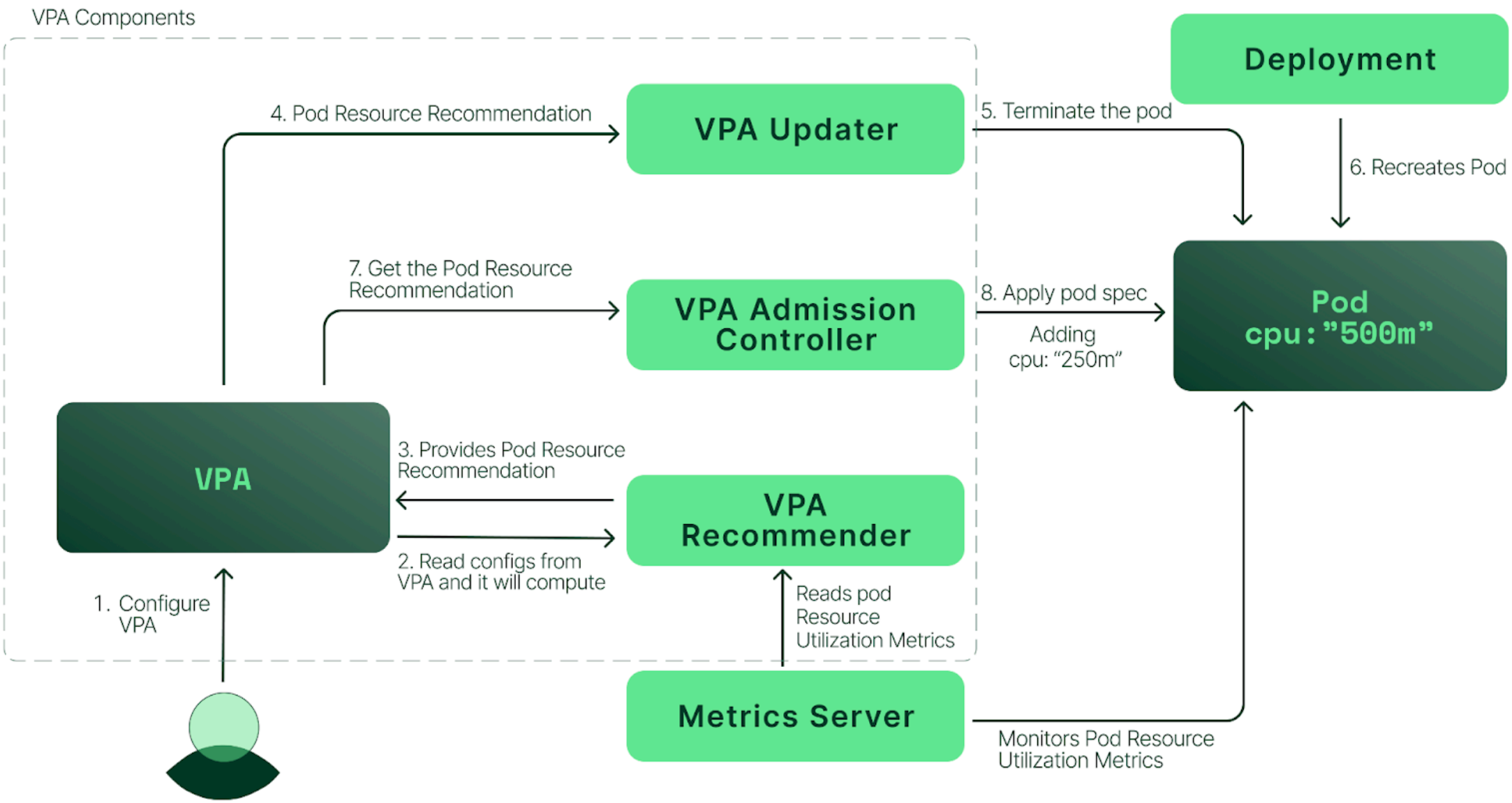
- **Requests**
  - Минимальные ресурсы, которые Kubernetes гарантирует контейнеру (например, 256Mi памяти)
  - Планировщик использует requests для размещения Pod'ов на узлах с достаточным объёмом ресурсов
- **Limits**
  - Максимальные ресурсы, которые контейнер может использовать (например, 512Mi памяти)
  - Предотвращает неконтролируемое потребление ресурсов контейнером

Example:

```yaml
resources:
  requests:
    memory: "256Mi"
  limits:
    memory: "512Mi"
```

Components of VPA

- The VPA Recommender
- The VPA Updater
- The VPA Admission Controller


VPA Components
Deployment
4. Pod Resource Recommendation
VPA Updater
5. Terminate the pod
6. Recreates Pod
7. Get the Pod Resource Recommendation
VPA Admission Controller
8. Apply pod spec
Adding cpu: "250m"
Pod cpu:"500m"
VPA
3. Provides Pod Resource Recommendation
VPA Recommender
2. Read configs from VPA and it will compute
1. Configure VPA
Reads pod Resource Utilization Metrics
Metrics Server
Monitors Pod Resource Utilization Metrics

VPA mode off

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recomender
  namespace: default
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment # Type of workload object yuo want to manage
    name: frontend  # The name of workload object yuo want VPA to manage
  updatePolicy:
    updateMode: "Off" # Set the mode to Off
```

VPA mode initial

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recomender
  namespace: default
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment # Type of workload object yuo want to manage
    name: frontend  # The name of workload object yuo want VPA to manage
  updatePolicy:
    updateMode: "Initial" # Set the mode to Initial.
```

> VPA назначает ресурсы при создании подов и не изменяет их в течение срока жизни пода.

VPA mode auto

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recomender
  namespace: default
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment # Type of workload object yuo want to manage
    name: frontend  # The name of workload object yuo want VPA to manage
  updatePolicy:
    updateMode: "Auto" # Set the mode to Auto or Recreate.
```

> `Auto` - автоматически применяет рекомендации VPA: при необходимости эвиктит и пересоздаёт поды, обновляя resource requests/limits

Изменение запросов и лимитов без пересоздания пода

с версии k8s v1.33 можно изменять вычислительные ресурсы у уже работающих подов без их перезапуска.

Изменение запросов и лимитов без пересоздания пода

> KEP-1287 добавил возможность изменять поля resources.requests и resources.limits в спецификации пода, которые раньше были неизменяемыми (immutable fields), без перезапуска пода. То есть теперь они меняются «на лету» или по месту.

Как это работает (по шагам)

1. Пользователь меняет CPU/Memory (requests/limits) в манифесте Pod и отправляет в API-сервер (kube-apiserver).
2. API-сервер сохраняет изменения; kubelet на узле получает обновлённую спецификацию Pod.
3. Kubelet быстро проверяет ресурсы узла (allocatable/capacity). Если хватает — продолжает; если нет — Pod получает состояние PodResizePending.
4. Kubelet через CRI передаёт рантайму (containerd/CRI-O/Podman) команду изменить ресурсы контейнеров.
5. Рантайм in-place обновляет cgroups контейнера, увеличивая/уменьшая выделение CPU/памяти без перезапуска.
6. Операция асинхронна: контейнер продолжает работу, kubelet не блокируется и обрабатывает другие задачи.

Обновление ресурсов в K8s v1.33

- FeatureFlag `InPlacePodVerticalScaling` включили по умолчанию
- Новый subresource `/resize` (доступен начиная с `kubectl 1.32`) — изменяет ресурсы пода без конфликтов с остальными полями спецификации
- В статус пода добавлены новые состояния, отражающие процесс изменения ресурсов

Новые состояния статуса Pod при изменении ресурсов

- `type: PodResizePending`
  - kubelet не может сразу удовлетворить запрос на изменение ресурсов; детали в `status.conditions[].message`
  - `reason: Infeasible` — запрошено больше ресурсов, чем доступно на текущем узле
  - `reason: Deferred` — сейчас невозможно, но может стать возможным позже; kubelet будет повторять попытки
- `type: PodResizeInProgress`
  - kubelet принял запрос и выделил ресурсы, изменения применяются (может занять время в зависимости от ресурса и рантайма)
  - Ошибки при применении отражаются в `message` с `reason: Error`

Совместимость Container Runtime

В таблице ниже указаны минимальные версии популярных Container Runtime, которые полностью поддерживают CRI (или не все) и работают с современными версиями Kubernetes:

Container Runtime	Compatible Version	Release Notes
containerd	v1.6.9+	containerd v1.6.9 Release Notes
CRI-O	v1.24.2+	CRI-O v1.24.2 Release Notes
Podman	v4.0.0+	Podman v4.0.0 Release Notes
Docker	Полностью не поддерживается	Но кто использует Docker как CRI в 2025-м?

История и происхождение CRI-проектов

- **Docker Engine (2013)** — Docker Inc. выделил `containerd` как низкоуровневый компонент, а в 2017 передал его в CNCF; dockershim удалён из Kubernetes с 1.24, нужен `cri-dockerd` от Mirantis.
- **containerd (2016)** — рефактор Docker Engine, ориентирован на стабильный daemon для CRI/OCI, CNCF Graduated с 2019; используется в GKE, EKS, kOps.
- **CRI-O (2016)** — SIG Node + Red Hat создали минималистичный CRI для Kubernetes, по умолчанию использует `runc` / `crun`, тесная интеграция с OpenShift.
- **Podman (2018)** — CLI и API поверх `libpod`, daemonless/Rootless, экспортирует Kubernetes YAML и делит большую часть кода с CRI-O (conmon, runtimes).

Сравнение Container Runtime

Runtime	Тип / архитектура	Отличия и преимущества
containerd	Демон + встроенный CRI plugin, управляет snapshoter'ами, image store	Малый поверхностный API, gRPC CRI без прослоек, зрелый tooling (`ctr`, `nerdctl`), стабильные LTS
CRI-O	Строго CRI-ориентированный демон, модули: storage, image, runtime, networking	Минимальная зависимость от Docker API, богатые политики безопасности (SELinux, seccomp, AppArmor) «из коробки», дефолт в OpenShift
Podman	CLI/REST без постоянного демона, запускает контейнеры через conmon и runc/crun	Rootless, `podman kube play/generate` (YAML ↔ контейнеры), Docker-совместимый UX, удобен для автоматизации
Docker (+cri-dockerd)	Docker Engine + адаптер Mirantis `cri-dockerd`	Экосистема Docker CLI/Compose, знакомый опыт образов, buildkit

Что общего и как выбирать

- **Общее:** у всех поддержка OCI-образов, использование `runc/crun`, cgroups v1/v2 и взаимодействие с kubelet через CRI; единые реестры и цепочки поставки.
- **Выбор:** containerd — универсальный и стабилен; CRI-O — про безопасность и строгий контроль; Podman — developer-first + rootless, но требует CRI-O/libpod-стека для полноценного кластера; Docker — для dev/build, но в кластере нужен дополнительный shim.
- **Практика:** выбираем тот runtime, для которого есть официальные пакеты/патчи в используемой дистрибуции Kubernetes + соответствующие инструменты диагностики в команде.

containerd vs CRI-O: архитектура и возможности

- **containerd**: CNCF Graduated runtime с модульной архитектурой (snapshotter, content store, runtime shim). Используется Docker, Amazon ECS, serverless-платформами; через `nerdctl` даёт Docker-like UX. Поддерживает image encryption/signing и `lazy pulls`, поэтому подходит для распределённых кластеров с ограниченным каналом (Spacelift 2025).
- **CRI-O**: Разработан SIG Node + Red Hat как самый тонкий слой между kubelet и OCI runtime. Делит стек с Podman/Buildah (libpod, conmon, runc/crun), что упрощает поддержку rootless и политик безопасности (SELinux/seccomp) в RHEL/OKD (Medium 2024).
- **Общее**: оба реализуют CRI, работают с OCI-образами и cgroups v1/v2. Различается акцент: containerd — на расширяемость и мульти-платформенность, CRI-O — на простоту и минимальный attack surface в Kubernetes.

containerd vs CRI-O: операционка и экосистема

- **Интеграции**: containerd встраивают Docker, kOps, managed Kubernetes, CI/CD сервисы, а также edge-платформы через plugins (storage, network, snapshotters). CRI-O идёт по умолчанию в OpenShift/MicroShift и тесно интегрирован с `podman kube play/generate`.
- **Управление**: containerd требует внешних CLI (`nerdctl`, `crictl`, BuildKit) для разработки и билда; взамен даёт доступ к новым функциям быстрее, чем Docker (Spacelift). CRI-O сфокусирован на kubelet, поддерживает declarative настройку storage/network plugins и проще в эксплуатации на узлах, где нужен только Kubernetes.
- **Сообщество**: containerd имеет широчайшее коммьюнити (Docker, CNCF, cloud-провайдеры), CRI-O — более нишевый, но плотно поддерживаемый Red Hat/SIG Node и security-экспертами.

Практические рекомендации выбора

- **Выберите CRI-O, если**: у вас кластер на RHEL/OKD/Openshift, нужен минимальный overhead, строгие политики безопасности и нет задач управлять контейнерами вне Kubernetes (Medium 2024).
- **Выберите containerd, если**: требуется единый runtime для Kubernetes, CI/CD runner'ов и standalone контейнеров; вы хотите использовать расширения (encrypted images, lazy pulls) и большую экосистему (Spacelift 2025).
- **Частая практика**: Developers используют Docker/Podman локально, а production-ноды переключают на containerd или CRI-O; миграция с Docker облегчена OCI-совместимостью и инструментами `nerdctl`, `crictl`, `podman`.

Посмотрим на реальные случаи для in-place resize, где эта возможность будет использоваться на 100 %.(K8s ≥ 1.33)

- **Базы данных**: PostgreSQL: быстро увеличить память под тяжёлые отчёты — без рестартов и потери соединений.
- **Stateless веб-приложения**: Node.js, Go, Python: «на лету» добавить CPU/память для пиков трафика.
- **ML-сервисы**: TensorFlow/ONNX: больше ресурсов для тяжёлых моделей без остановки обработки.
- **Прокси-сайдкары (service mesh)**: Envoy/Istio: подстраивать ресурсы под трафик без влияния на основное приложение.
- **JVM-приложения** (особенности): CPU и не-heap меняются сразу; для (-Xmx) все еще нужен рестарт.

Ограничения In-Place Pod Resize

- При использовании `swap` изменить память без рестарта контейнера нельзя
- Не поддерживается на Windows-узлах
- Меняются только `CPU` и `memory`
- Понижать `memory limit` без рестарта нельзя; `memory request` — можно (риск OOM при слишком низком лимите)
- QoS-класс пода не меняется после старта
- `init`- и `ephemeral`-контейнеры нельзя рестартить
- Нельзя удалять `requests/limits` — только менять их значения
- Поле `resizePolicy` неизменно после создания пода

Limitations of Kubernetes VPA

- **Do not use VPA and HPA together** if both scale on the same metrics (CPU/Memory)
- **VPA may recommend more resources than available** in the cluster, causing Pods to remain unscheduled
- **VPA is unaware of node sizes** (CPU/Memory) in the cluster
- **No support for StatefulSets** (as of now)
- **VPA does not consider network or I/O metrics**
- **Limited historical data**—VPA uses only the last 8 days

Goldilocks: VPA Namespace and Container Configuration

- **Enable Goldilocks on a Namespace:** - Add the label to your namespace:

```
kubectl label ns goldilocks goldilocks.fairwinds.com/enabled=true
```

- **Set VPA Update Mode:** - Control how VPA recommendations are applied. Common modes: `off`, `initial`, `auto`:

```
kubectl label ns goldilocks goldilocks.fairwinds.com/vpa-update-mode="auto"
```

- **Exclude Containers from VPA Recommendations:** - Apply a label to a workload (e.g., Deployment) to exclude specific containers. Useful for sidecars or proxies you don't want VPA to manage:

```
kubectl label deployment myapp goldilocks.fairwinds.com/exclude-containers=linkerd-proxy,istio-proxy
```

The `resourcePolicy` section can be defined by using the following annotation on a namespace:

```
annotations:
   goldilocks.fairwinds.com/vpa-resource-policy: >
    { "containerPolicies": [
       {
        "containerName": "nginx",
        "minAllowed": {
          "cpu": "250m",
          "memory": "100Mi" },
        "maxAllowed": {
          "cpu": "2000m",
          "memory": "2048Mi" }
        },
        {
          "containerName": "istio-proxy",
         "mode": "Off"
        }
       ]}
```

Практика

LAB: Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA)

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#lab-5-horizontal-pod-autoscaler-hpa-and-vertical-pod-autoscaler-vpa

Обновление

Kubernetes поддерживает различные стратегии обновления для минимизации времени простоя и обеспечения непрерывности работы приложений.

Rolling Update

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: my-app-image:latest
        ports:
        - containerPort: 80
```

Type of deployment.

Can be `Recreate` or `RollingUpdate`. Default is `RollingUpdate`.

```yaml
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
```

`Recreate` - Kill all existing pods before creating new ones.

`RollingUpdate` - Replace the old ReplicaSets by new one using rolling update i.e gradually scale down the old ReplicaSets and scale up the new one.

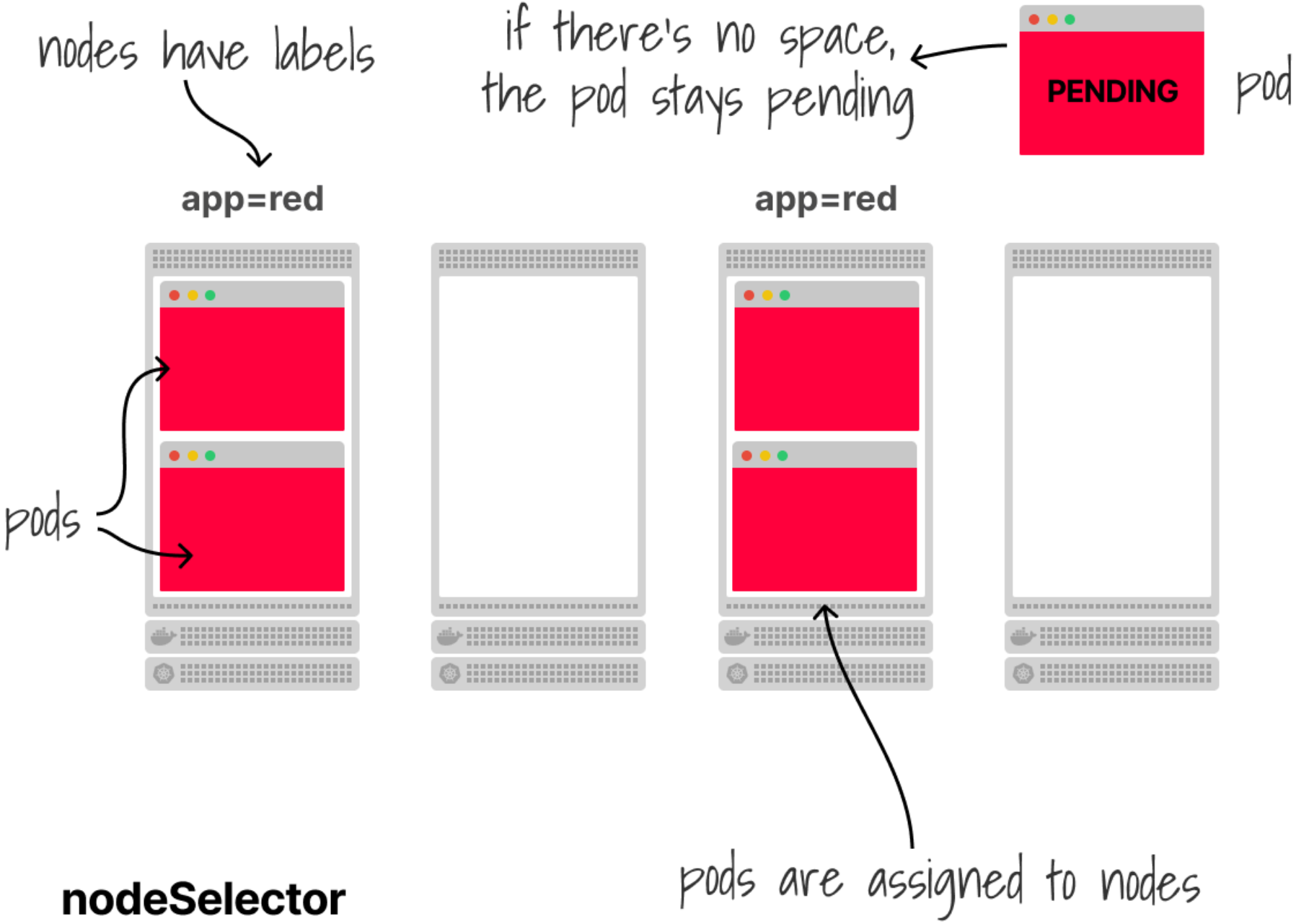
Практика

LAb Advanced Deployment Strategies with Rolling Updates and Rollbacks

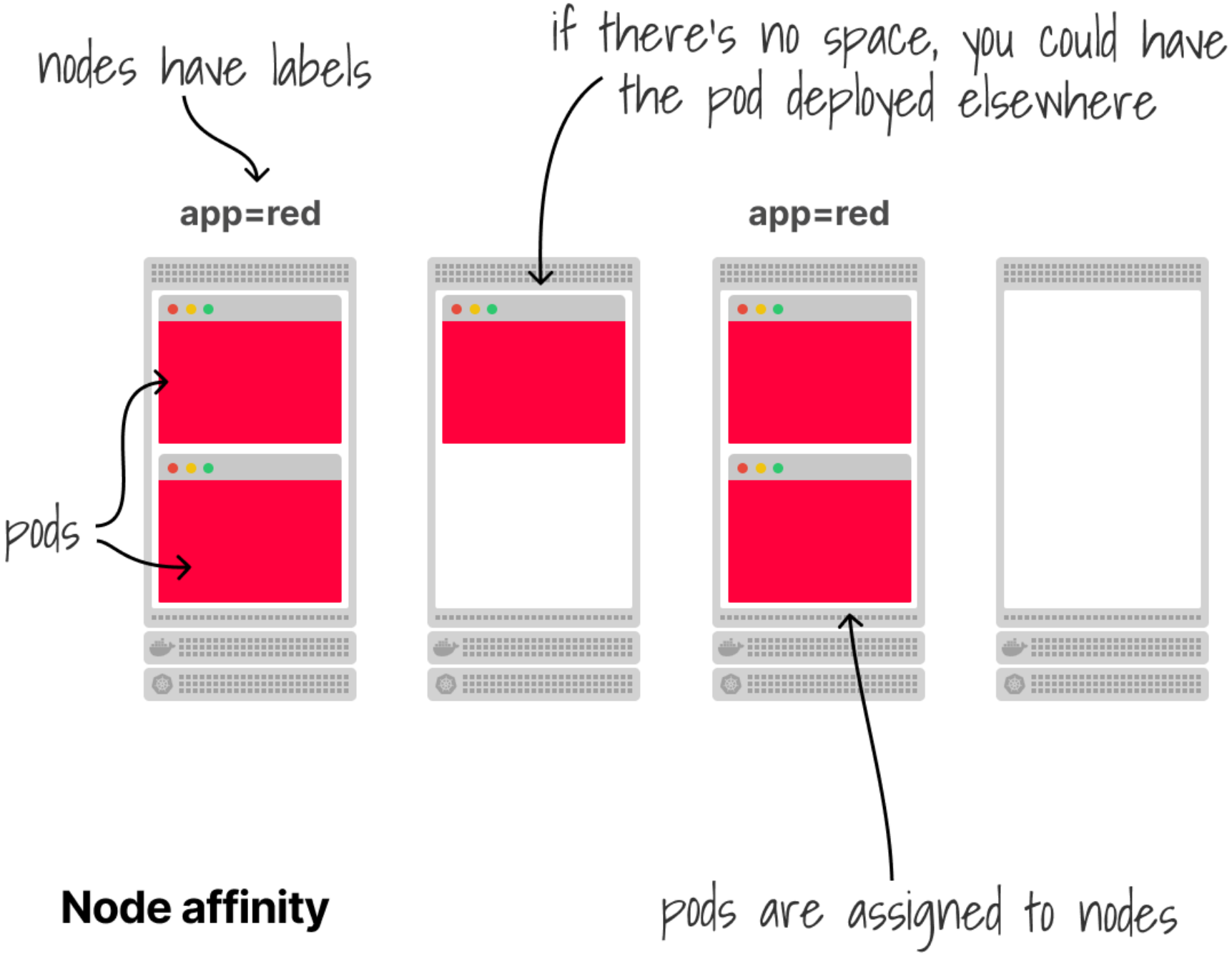
https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#lab-1-advanced-deployment-strategies-with-rolling-updates-and-rollbacks

INFLUENCING THE SCHEDULER

1. nodeSelector
2. Node affinity
3. Pod affinity/anti-affinity
4. Taints and tolerations
5. Topology constraints
6. Scheduler profiles



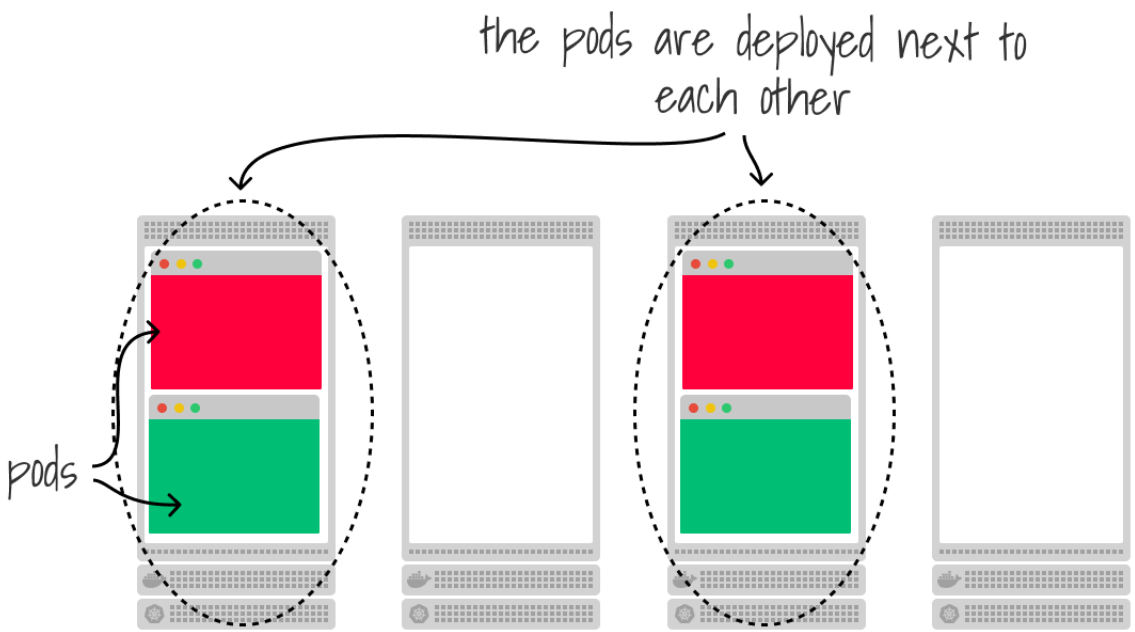
nodeSelector



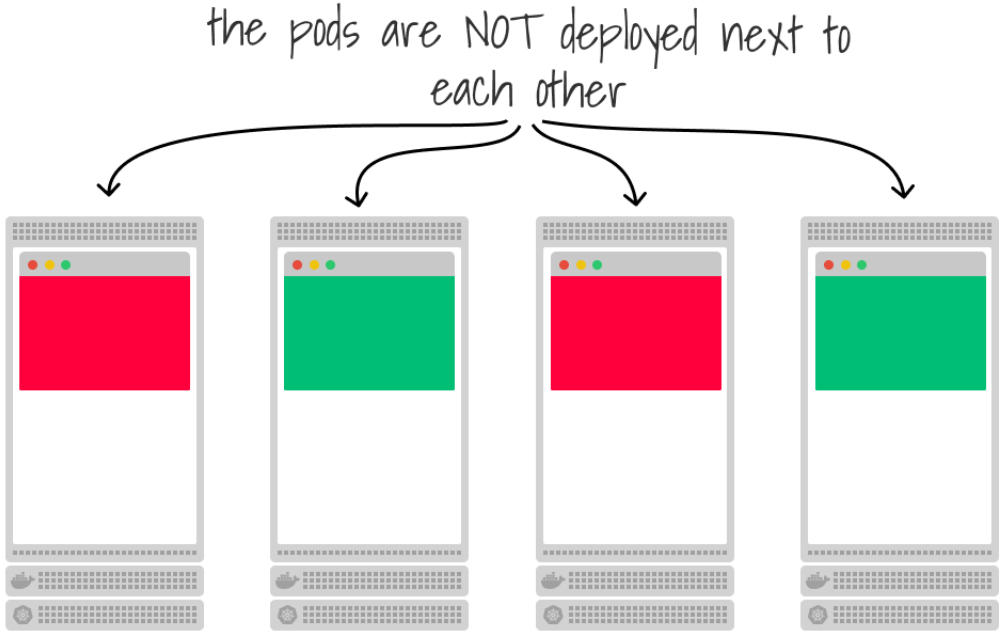
Node affinity

Node Affinity (Preferred) Example

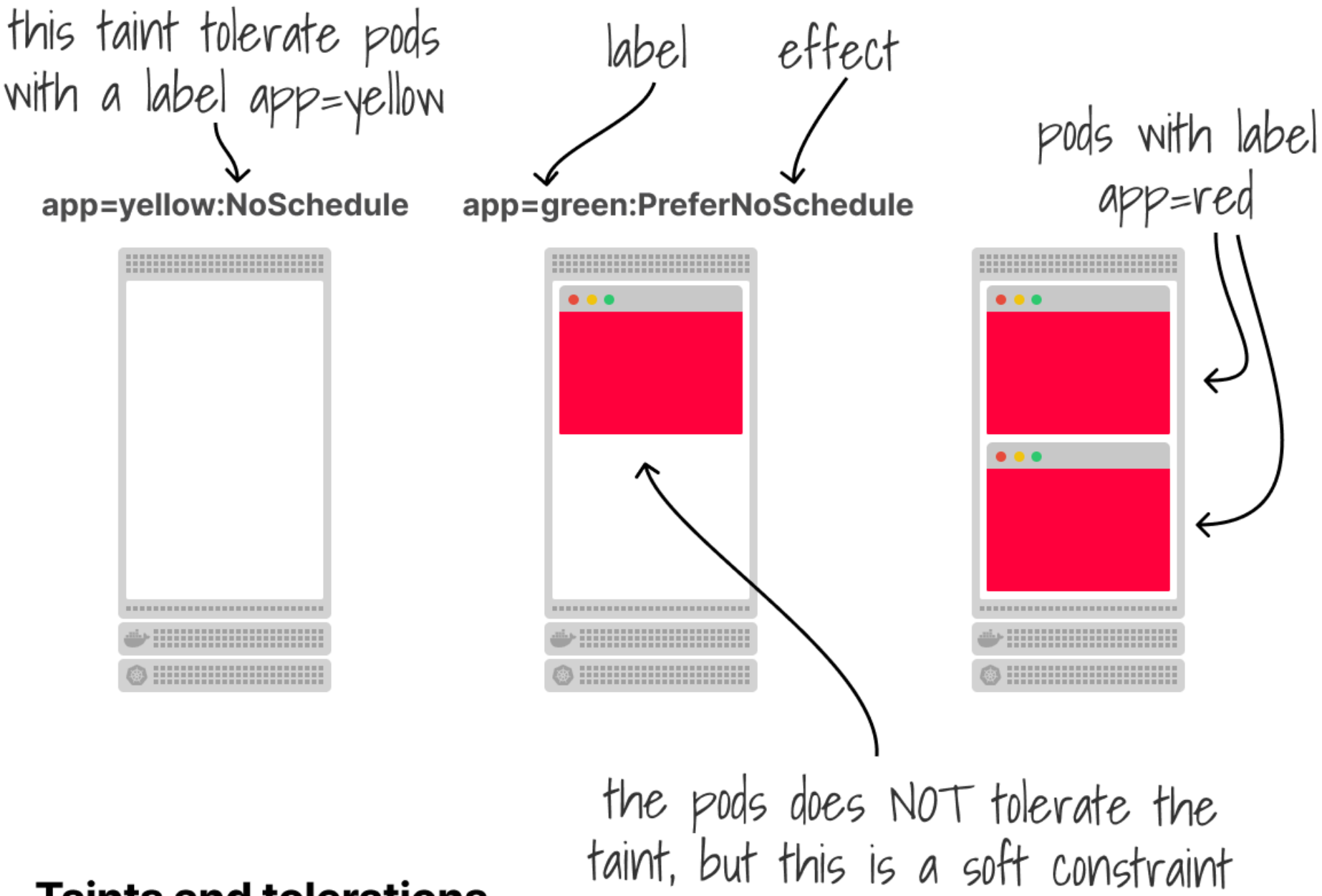
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo-node-affinity
  labels:
    app: demo
spec:
  containers:
  - name: web
    image: nginx:1.25
  affinity:                                   # Affinity rules influencing pod scheduling
    nodeAffinity:                             # Node-level selection logic
      preferredDuringSchedulingIgnoredDuringExecution:  # Soft preferences; honored when possible
        - weight: 70                          # Higher weight = stronger preference vs other entries
          preference:
            matchExpressions:                 # List of label-based conditions
              - key: node.kubernetes.io/instance-type   # Node label key to test
                operator: In                  # Match if node label value is in provided list Can be In, NotIn, DoesNotExist, Lt or Gt
                values:                       # Preferred instance types for performance/capacity
                  - m5.large
                  - m5.xlarge
        - weight: 30                          # Lower priority secondary preference
          preference:
            matchExpressions:
              - key: topology.kubernetes.io/zone        # Zone (failure domain) label key
                operator: In                  # Any of the listed zones is acceptable
                values:                       # Preferred availability zones
                  - eu-central-1a
                  - eu-central-1b
```



Node affinity



Node anti-affinity



Taints and tolerations

A taint can produce three possible effects:

- **NoSchedule**
  Планировщик разместит только Pod'ы, имеющие toleration для данного taint.
- **PreferNoSchedule**
  Планировщик будет избегать размещения Pod'ов без соответствующего toleration, но при необходимости может разместить.
- **NoExecute**
  Kubernetes выселит (evict) с узла работающие Pod'ы без toleration и не запустит новые.

built-in taints:

- **node.kubernetes.io/not-ready**
  Узел не готов. Соответствует условию Ready = "False".
- **node.kubernetes.io/unreachable**
  Узел недостижим для контроллера узлов. Соответствует условию Ready = "Unknown".
- **node.kubernetes.io/memory-pressure**
  Узел испытывает давление по памяти.
- **node.kubernetes.io/disk-pressure**
  Узел испытывает давление по диску. При высокой загрузке диска приложения замедляются, стоит переразместить поды.

built-in taints:

- **node.kubernetes.io/pid-pressure**
  На узле давление по PID. Идентификаторы процессов — ограниченный ресурс; их исчерпание может вызвать простой приложений, поэтому поды лучше перенести на другие узлы.
- **node.kubernetes.io/network-unavailable**
  Сеть узла недоступна (как описано выше).
- **node.kubernetes.io/unschedulable**
  Узел непригоден для планирования. Любая причина, делающая узел неподходящим для размещения подов (например, уменьшение размера кластера и вывод узла).

pod topology spread constraints

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
```

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: my-pod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
```

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: my-pod-2
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: node
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  - maxSkew: 1
    topologyKey: rack
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
```

Практика

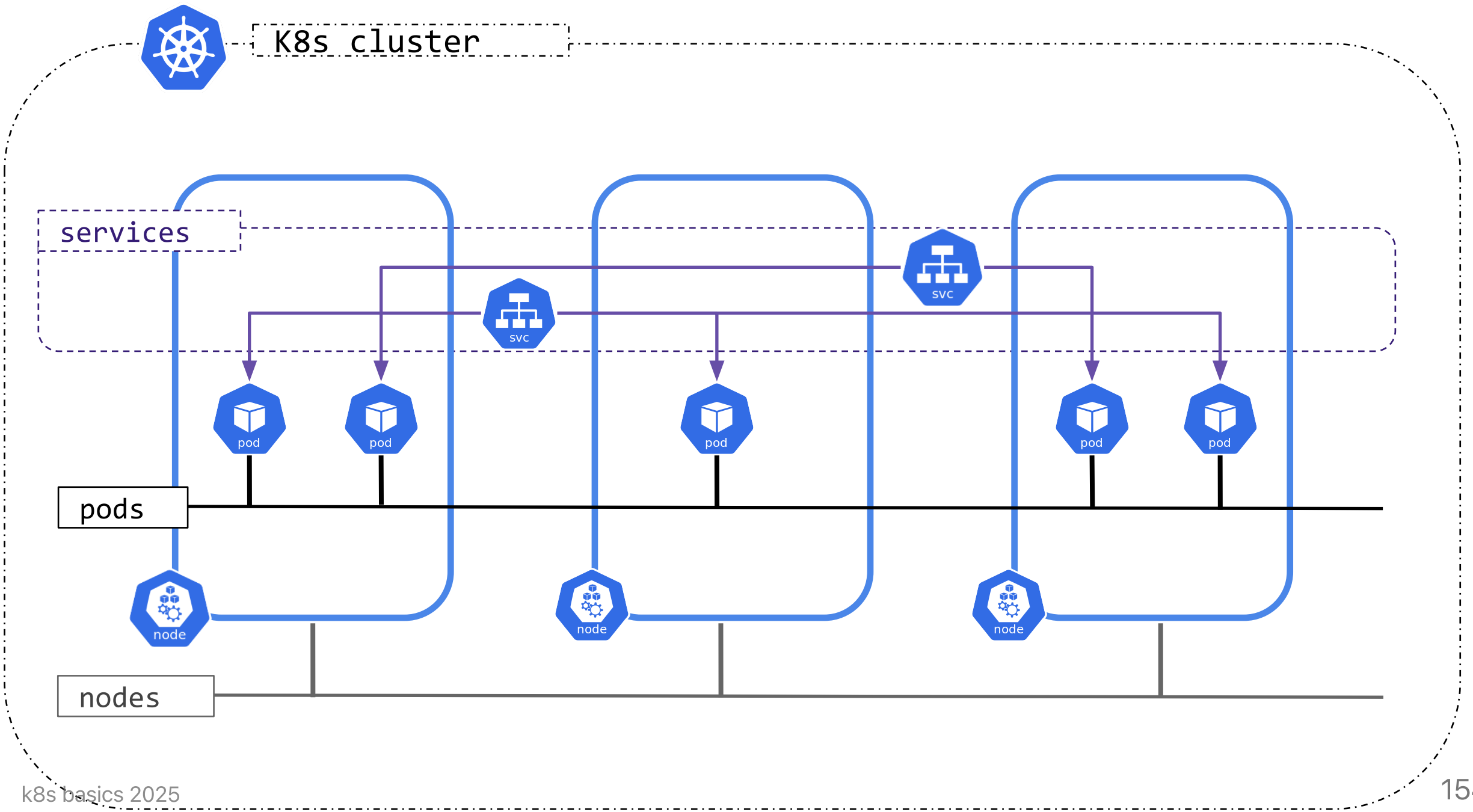
LAb Kubernetes Scheduling. Taints and Tolerations

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#lab-11-kubernetes-taints-and-tolerations

Kubernetes Networking 101

IP Address Ranges in Kubernetes

- **Кластерам Kubernetes требуются непересекающиеся диапазоны IP** для:
  - Pod'ов
  - Service'ов
  - Узлов (Nodes)
- **Диапазоны IP задаются разными компонентами:**
  - **Сетевой плагин (CNI):** назначает IP-адреса Pod'ам
  - **kube-apiserver:** управляет диапазоном IP для Service (ClusterIP range)
  - **kubelet / cloud-controller-manager:** предоставляет/регистрирует IP-адреса узлов


K8s cluster
services
svc
svc
pod
pod
pod
pod
pod
pods
node
node
node
nodes

Calculating the Pod Network Range in Kubernetes

При настройке кластера Kubernetes необходимо определить **Pod network CIDR** — диапазон IP-адресов, из которого все Pod'ы будут получать свои IP. Правильный выбор диапазона критичен для надёжной сетевой работы и будущей масштабируемости.

Step 1: Estimate the Number of Pods

- **Best Practice:** планируйте с учётом будущего роста.
- **Default max Pods per node:** 110 (можно изменить флагом kubelet `--max-pods`).

Пример:

Если планируется до 50 узлов, каждый до 100 Pod:

```
Всего Pod = 50 узлов × 100 Pod/узел = 5 000 Pod
```

Step 2: Choose a Non-Overlapping Private IP Range

- Используйте приватный диапазон IP (RFC1918), который **не пересекается** с:
  - IP-адресами узлов (Node IPs)
  - Service CIDR
  - Любой существующей сетью в вашем дата-центре или облачном VPC

Common choices:

- `10.244.0.0/16` (Flannel default)
- `10.0.0.0/8`
- `192.168.0.0/16`
- `172.16.0.0/12`

Official docs

Step 3: Calculate the Required CIDR Size

- Каждый Pod получает уникальный IP.
- **Примеры размеров CIDR:**
  - `/16` = 65 536 IP-адресов
  - `/19` = 8 192 IP-адреса
  - `/20` = 4 096 IP-адресов

Для 5 000 Pod'ов:

- `/19` даёт 8 192 IP-адреса (достаточно для 5 000 Pod'ов + запас на рост).

Step 4: Assign Per-Node Pod CIDRs

- Сеть Pod делится на меньшие подсети, по одной на узел.
- **Пример:**
  - Pod CIDR: `10.244.0.0/16`
  - Каждый узел получает `/24` (256 IP): `10.244.1.0/24`, `10.244.2.0/24`, и т.д.

Рекомендация:

- Убедитесь, что подсеть Pod на узле достаточно велика для максимального числа Pod на этом узле.

Step 5: Configure the Cluster

- **kubeadm:**
  Укажите `--pod-network-cidr=10.244.0.0/16` при инициализации:
  ```
  kubeadm init --pod-network-cidr=10.244.0.0/16
  ```
- **Облачные провайдеры:**
  Настройте Pod CIDR в облачной сети и в конфигурации CNI-плагина.

Step 6: Avoid Overlaps

- **Service CIDR:**
  Choose a different, non-overlapping range (e.g., `10.96.0.0/12`).
- **Node IPs:**
  Ensure node IPs are outside the Pod and Service CIDRs.

Summary Table

Resource	Example CIDR	Size	Notes
Pod network	10.244.0.0/16	65,536 IPs	For all Pods in cluster
Per-node Pods	10.244.1.0/24	256 IPs	Each node gets a /24 block
Service CIDR	10.96.0.0/12	1M IPs	For ClusterIP Services
Node IPs	192.168.0.0/16		For VM/physical nodes

Ключевые моменты:

- Всегда используйте приватный, непересекающийся CIDR для Pod.
- Планируйте рост — выбирайте диапазон больше текущих потребностей.
- Диапазоны IP для Pod, Service и Node не должны пересекаться.
- Настройте CNI-плагин и кластер соответствующим образом.

References

- Kubernetes Networking Concepts
- Pod CIDR Planning

Container Networking

- **«Контейнеров не существует»**
  Контейнер — это просто изолированный процесс Linux.
- **Пространства имён (namespaces) Linux** обеспечивают изоляцию:
  - PID, network, mount, UTS, user, и т. д.
- **Сетевое пространство имён (network namespace)** даёт процессу (и его дочерним) собственный виртуальный сетевой стек.
- Каждое сетевое пространство имён имеет свои:
  - Интерфейсы (например, `loopback`, `eth0`)
  - Таблицы маршрутизации
  - Прочие сетевые ресурсы

Pod Architecture

- **Pod объединяет один или несколько контейнеров** в единую развёртываемую единицу.
- **Все контейнеры внутри Pod разделяют один сетевой namespace:**
  - Общий IP-адрес
  - Общее пространство портов
- **Pod — атомарная сущность в Kubernetes:**
  - Планируется, масштабируется и завершается как единое целое
- **Нельзя «разнести» один Pod по нескольким узлам** — он всегда работает на одном узле.

Pod Networking

- **Компоненты распределённых приложений должны обмениваться данными** внутри кластера.
- **Для этого Pod'ам нужна сеть**.
- **Расположение Pod не важно:**
  - Pod'ы могут находиться на одном или на разных узлах.
  - Важно лишь, что они доступны друг другу.
- **Каждый Pod получает собственный сетевой интерфейс и уникальный IP-адрес.**
- **Контейнеры внутри одного Pod общаются через `localhost` (loopback).**

CNI – Container Networking Interface

- **CNI — это спецификация для сетевого взаимодействия контейнеров**
  - Позволяет Kubernetes интегрироваться с решениями программно-определяемых сетей (SDN).
- **Плагины CNI выполняют:**
  - Подключение (wiring) сетей Pod'ов
  - Назначение IP-адресов Pod'ам
  - Управление сетевыми политиками
- **SDN** позволяет управлять сетью через программные контроллеры вместо ручной настройки сетевого оборудования.

How CNI Works in Kubernetes

- **CNI (Container Networking Interface) имеет два основных компонента:**
  - **CNI плагин:** бинарник, реализующий спецификацию CNI.
  - **Файл конфигурации CNI:** JSON-файл, задающий используемый плагин и его параметры.
- **Создание Pod инициирует настройку сети:**
  - kubelet на узле вызывает настроенный CNI плагин.

Пример:

Когда новый Pod назначен на узел, kubelet читает `/etc/cni/net.d/10-calico.conflist`, чтобы определить какой плагин (например, Calico) использовать.

How CNI Works in Kubernetes

- Плагин CNI:
  - Создаёт сетевое пространство имён (network namespace) для Pod.
  - Настраивает пару виртуальных Ethernet-интерфейсов (veth) и подключает один конец к мосту.
  - Назначает Pod'у IP-адрес.
  - Обновляет таблицы маршрутизации и DNS при необходимости.
- **Популярные CNI-плагины:**
  Calico, Flannel, Cilium

What is a Service in Kubernetes?

Service: redis-service
Selector app=redis

Endpoint:
redis-service

Pod
label app=redis

Pod
label app=redis

Pod
label app=redis

Pod
label app=redis

```yaml
kind: Service
apiVersion: v1

metadata:
  name: hostname-service

spec:
  type: NodePort
  selector:
    app: echo-hostname
  ports:
    - nodePort: 30163
      port: 8080
      targetPort: 80
```

Make the service available to network requests from external clients

Forward requests to pods with label of this value

`nodePort`
access service via this external port number

`port`
port number exposed internally in cluster

`targetPort`
port that containers are listening on

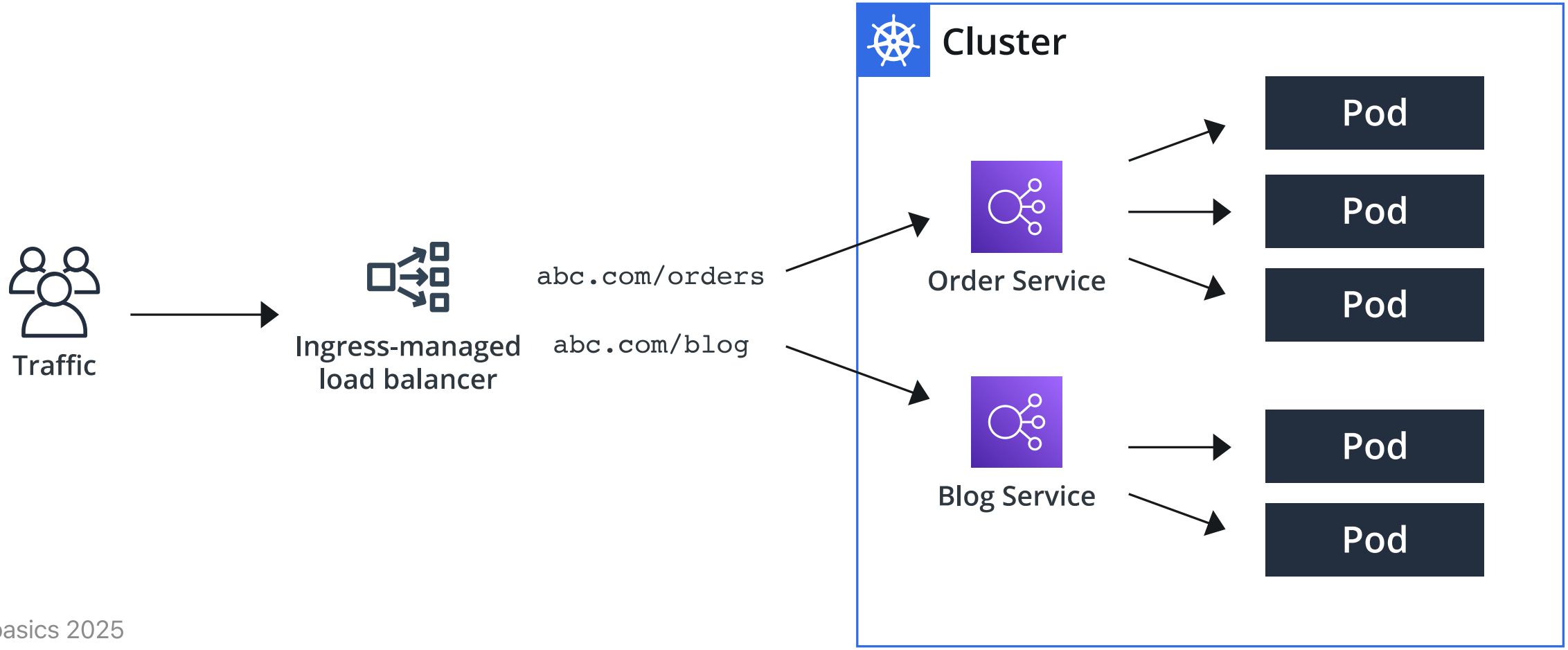
NodePort service yaml

```yaml
kind: Service
apiVersion: v1
metadata:
  name: hostname-service
spec:
  # Expose the service on a static port on each node
  # so that we can access the service from outside the cluster
  type: NodePort

  # When the node receives a request on the static port (30163)
  # "select pods with the label 'app' set to 'echo-hostname'"
  # and forward the request to one of them
  selector:
    app: echo-hostname

  ports:
    # Three types of ports for a service
    # nodePort - a static port assigned on each the node
    # port - port exposed internally in the cluster
    # targetPort - the container port to send requests to
    - nodePort: 30163
      port: 8080
      targetPort: 80
```

ClusterIP, NodePort, and LoadBalancer

- **ClusterIP** – значение по умолчанию. Сервис доступен только внутри кластера Kubernetes; извне обратиться к Pod нельзя.
- **NodePort** – делает сервис доступным на статическом порту каждого узла кластера `<NodeIP>:<NodePort>`.
- **LoadBalancer** – предоставляет внешний доступ к сервису через балансировщик нагрузки облачного провайдера (GCP, AWS, Azure, OpenStack).

Ingress


Cluster
Traffic
Ingress-managed load balancer
abc.com/orders
abc.com/blog
Order Service
Blog Service
Pod
Pod
Pod
Pod
Pod

Ingress in Kubernetes

Ingress — объект API Kubernetes, управляющий внешним HTTP/HTTPS-доступом к сервисам в кластере. Он задаёт правила маршрутизации, определяя, как внешние запросы направляются к внутренним сервисам.

> Примечание: Ingress заморожен; новые возможности добавляются в Gateway API.
> Подробнее см. документацию Kubernetes Ingress.

Key Features of Ingress

- **Единая точка входа:** публикует несколько сервисов под одним IP или DNS-именем
- **Маршрутизация по пути и хосту:** направляет трафик по URL-пути или имени хоста
- **Завершение TLS:** обрабатывает HTTPS и SSL-сертификаты
- **Правила rewrite и redirect:** поддерживают расширенные сценарии маршрутизации

How Ingress Works

1. Определите ресурс Ingress с правилами маршрутизации.
2. **Ingress Controller** (например, NGINX, Traefik) отслеживает ресурсы Ingress и настраивает балансировщик нагрузки или прокси.
3. Внешний трафик направляется к нужному сервису согласно правилам.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  ingressClassName: nginx
  rules:
  - host: example.com
    http:
      paths:
      - path: /app1
        pathType: Prefix
        backend:
          service:
            name: app1-service
            port:
              number: 80
  tls:
  - hosts:
    - example.com
    secretName: example-tls
```

Ingress Controllers

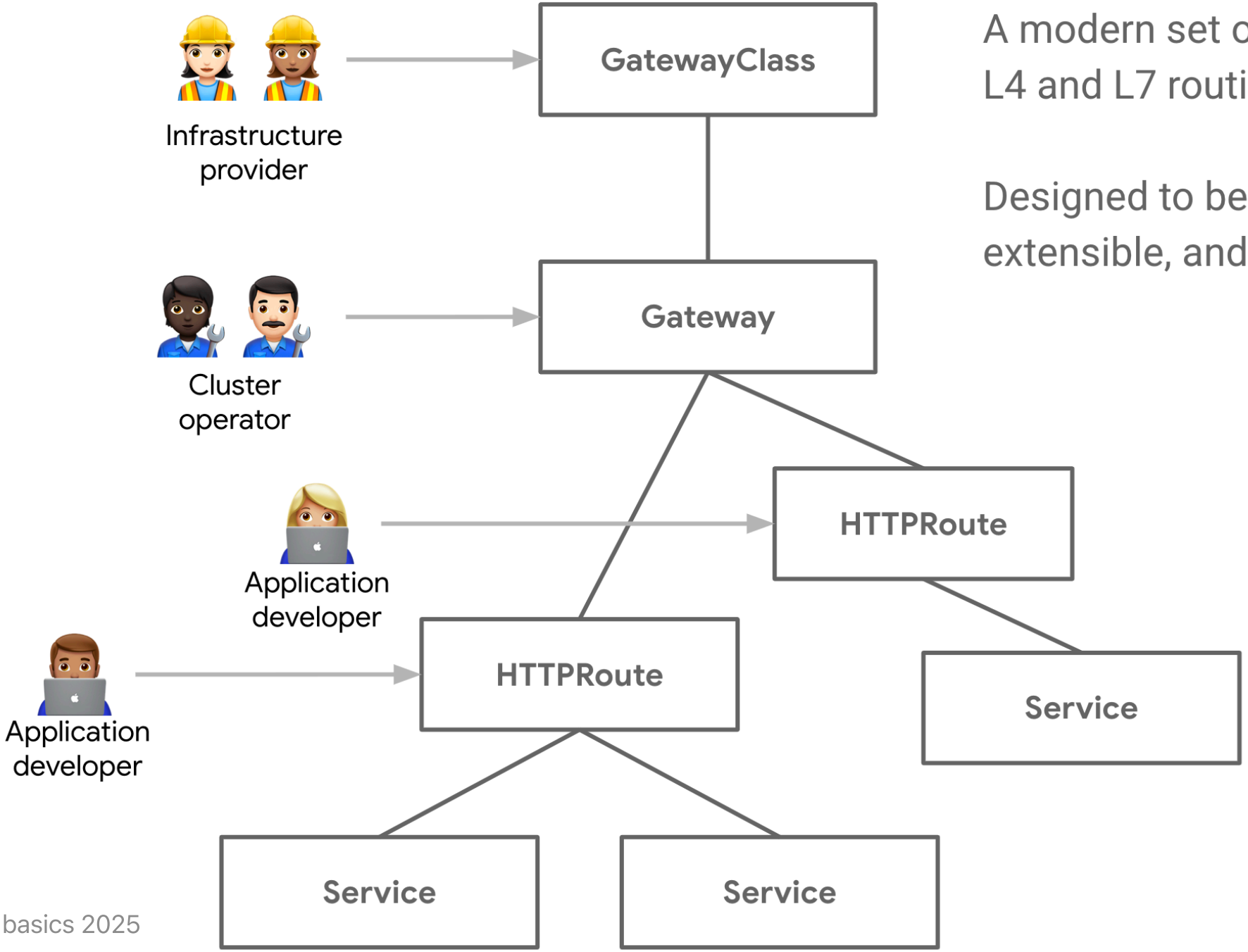
Чтобы использовать Ingress, разверните Ingress Controller в кластере. Популярные варианты:

- Ingress NGINX Controller
- Traefik
- HAProxy Ingress

> ПРИМЕЧАНИЕ: 11 ноября 2025 г. SIG Network Kubernetes и Security Response Committee объявили о предстоящем выводе из эксплуатации Ingress NGINX. Поддержка в формате best-effort будет продолжаться до марта 2026. После этого не планируется выпусков, исправлений ошибок и обновлений для устранения новых уязвимостей. Существующие развертывания Ingress NGINX продолжат работать, артефакты установки останутся доступными.
> Ingress NGINX Retirement

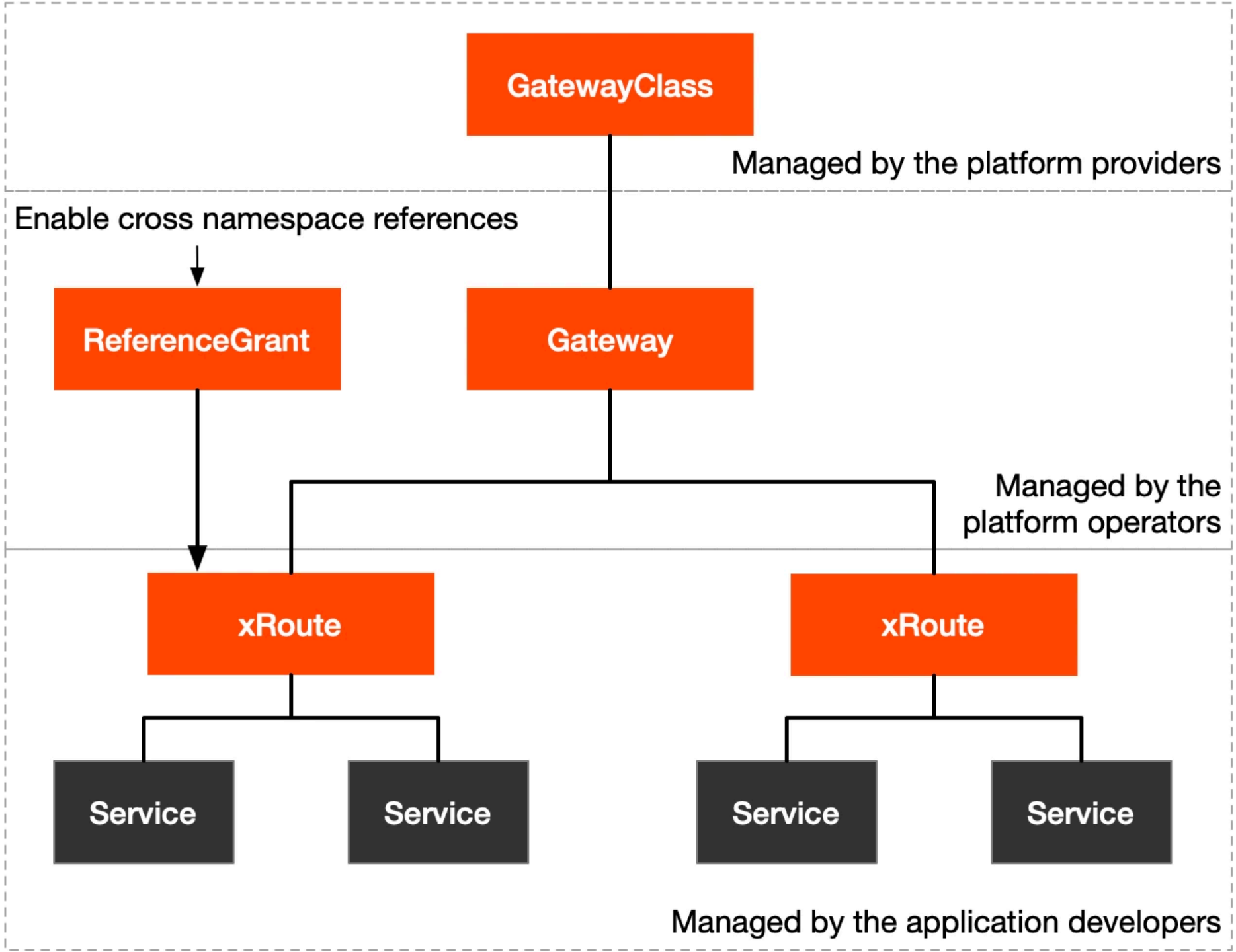
Best practices/tips for effective use of Service

1. Всегда используйте DNS для подключения к другим сервисам внутри кластера.
2. Используйте Ingress при внешнем экспонировании, когда это возможно.
3. Экпонируйте наружу только стандартные порты 80 и 443; применяйте правила Ingress для маршрутизации трафика к конкретным Service.
4. Всегда включайте SSL/TLS во внешних балансировщиках нагрузки.
5. Используйте NetworkPolicy для управления ограничением сетевого доступа.



A modern set of APIs for deploying L4 and L7 routing in Kubernetes

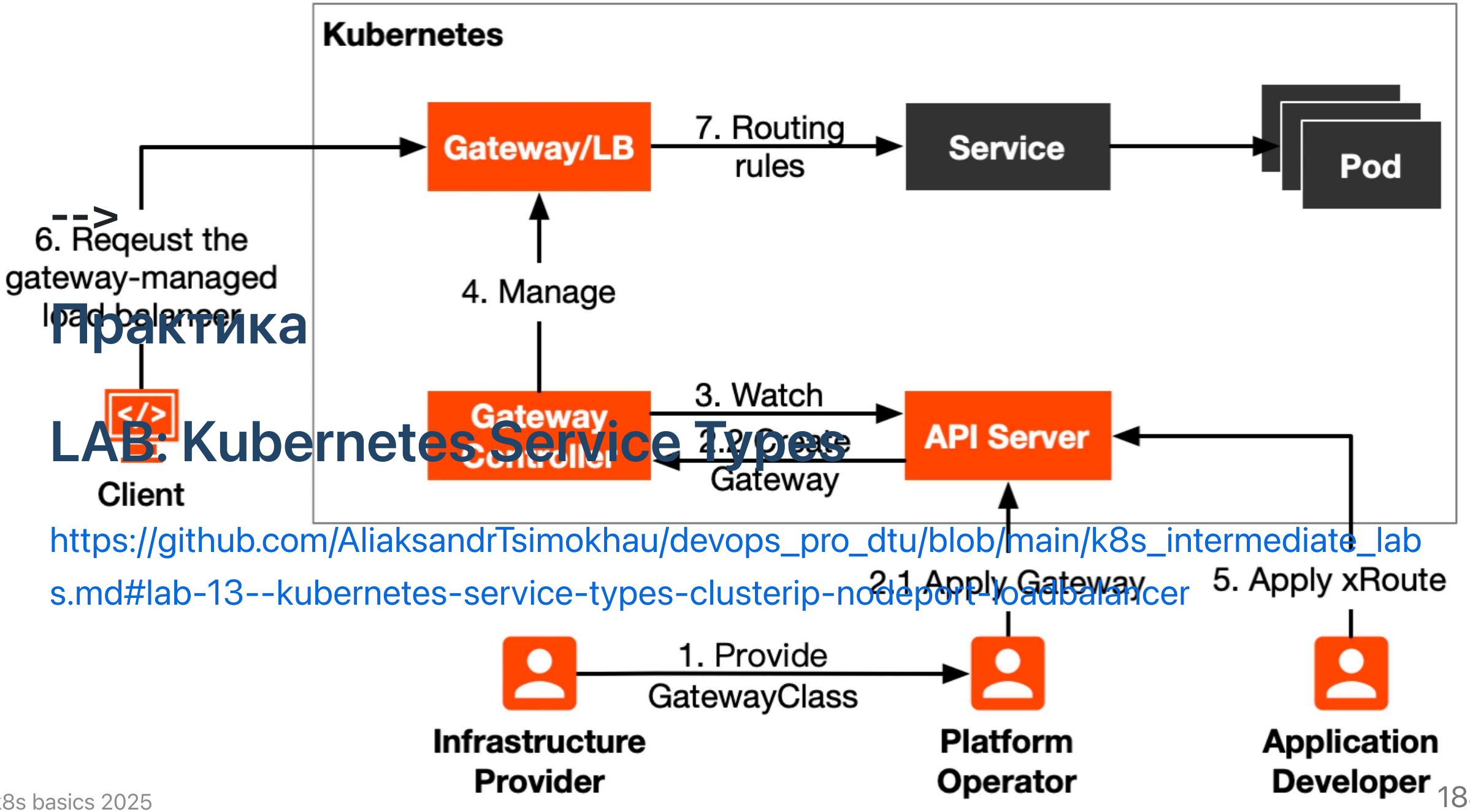
Designed to be generic, expressive, extensible, and role-oriented


GatewayClass
Managed by the platform providers
Enable cross namespace references
ReferenceGrant
Gateway
Managed by the platform operators
xRoute
xRoute
Service
Service
Service
Service
Managed by the application developers

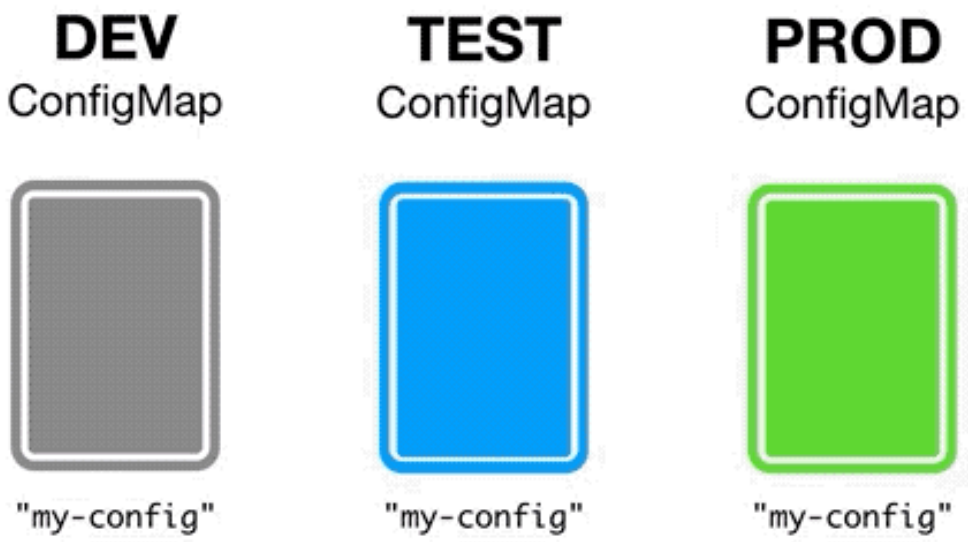
Практика

LAB: Kubernetes Service Types

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#lab-13--kubernetes-service-types-clusterip-nodeport-loadbalancer



Управление конфигурацией и секретами в Kubernetes: изучение config map, secret, persistent volume claim


DEV
ConfigMap
"my-config"
TEST
ConfigMap
"my-config"
PROD
ConfigMap
"my-config"

How does a ConfigMap work?

A ConfigMap is a dictionary of key-value pairs that store configuration settings for your applications.

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: example-configmap
data:
  # Configuration values can be set as key-value properties
  database: mongodb
  database_uri: mongodb://localhost:27017
  # Or set as complete file contents (even JSON!)
  keys: |
    image.public.key=771
    rsa.public.key=42
```

Using a ConfigMap in Environment Variables

The key to adding your ConfigMap as environment variables to your pods is the `envFrom` property in your Pod's YAML.

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: pod-env-var
spec:
  containers:
    - name: env-var-configmap
      image: nginx:1.7.9
      envFrom:
        - configMapRef:
            name: example-configmap
```

Существует четыре способа использовать ConfigMap для настройки контейнера внутри Pod:

1. В команде и аргументах контейнера
2. Как переменные окружения контейнера
3. Добавить файл в том (read-only volume), который приложение читает
4. Написать код внутри Pod, использующий Kubernetes API для чтения ConfigMap

Secrets. Built-in Secret Types

Opaque Secrets – тип Secret по умолчанию, если в манифесте не указан type.

Service account token Secrets – хранят токен, идентифицирующий service account.

Docker config Secrets – содержат учётные данные для доступа к Docker registry.

Basic authentication Secret – хранят логин и пароль для базовой аутентификации (Basic Auth).

SSH authentication Secrets – хранят данные (ключи) для SSH-аутентификации.

TLS Secrets – содержат сертификат и соответствующий приватный ключ.

Bootstrap token Secrets – токены, используемые при первоначальном подключении (bootstrap) узлов к кластеру.

Basic secret example

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4K
  password: cGFzc3dvcmQK
```

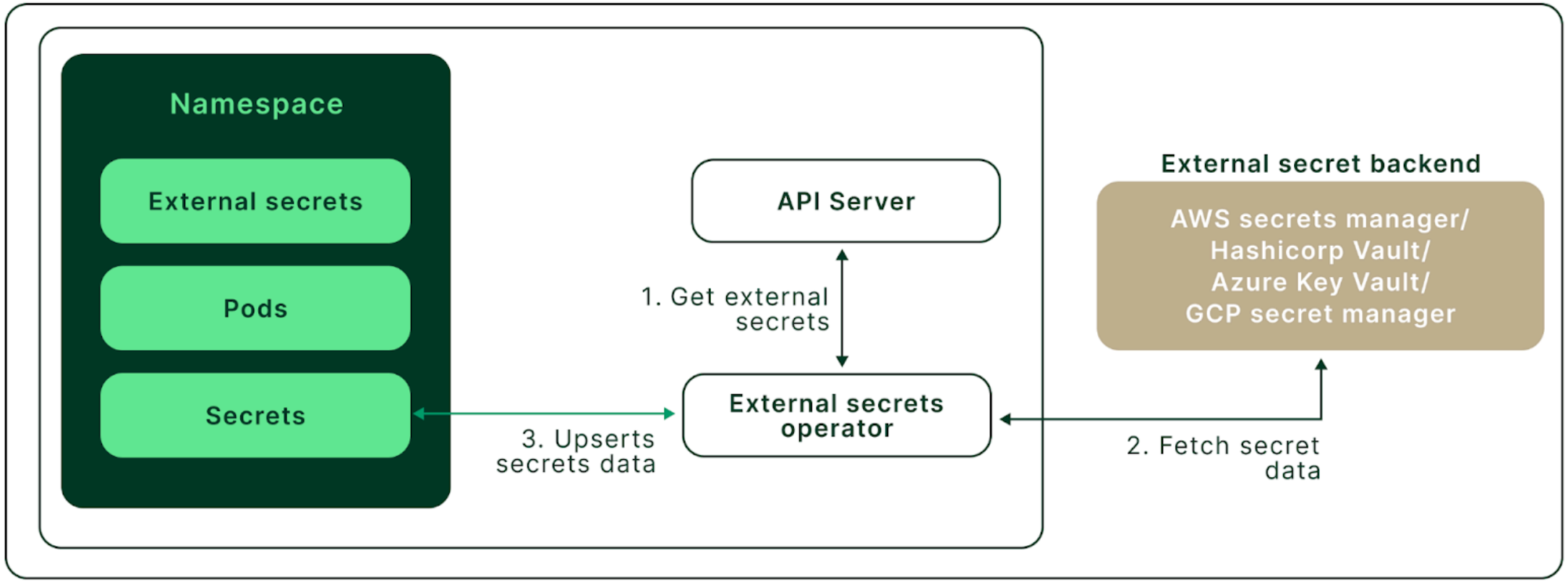
ssh-privatekey

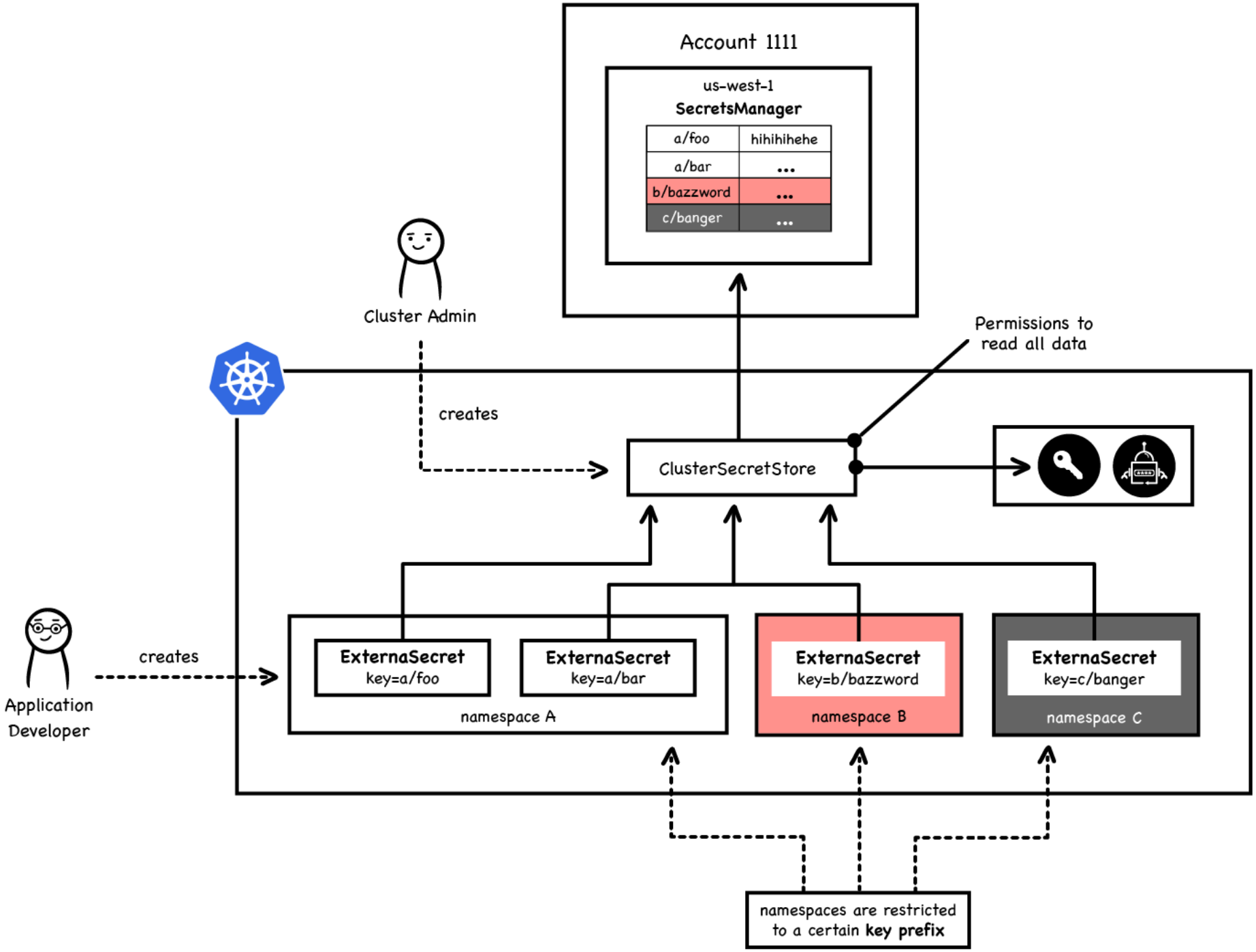
```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth
data:
  # the data is abbreviated in this example
  ssh-privatekey: |
     MIIEpQIBAAKCAQEAulqb/Y ...
```

TLS Secret

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-tls
type: kubernetes.io/tls
data:
  # the data is abbreviated in this example
  tls.crt: |
    MIIC2DCCAcCgAwIBAgIBATANBgkqh ...
  tls.key: |
    MIIEpgIBAAKCAQEA7yn3bRHQ5FHMQ ...
```

External Secrets




Account 1111
us-west-1
SecretsManager
a/foo
hihihihehe
a/bar
...
b/bazzword
...
c/banger
...
Cluster Admin
creates
Permissions to read all data
ClusterSecretStore
Application Developer
creates
ExternaSecret
key=a/foo
ExternaSecret
key=a/bar
namespace A
ExternaSecret
key=b/bazzword
namespace B
ExternaSecret
key=c/banger
namespace C
namespaces are restricted to a certain key prefix

Практика

LAb ConfigMaps and Secrets Management

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#lab-2-configmaps-and-secrets-management

Безопасность и управление доступом

```
kubectl api-resources
```

Эта команда выводит таблицу со столбцами:

NAME: Имя ресурса (например: pods, deployments, services).
SHORTNAMES: Короткие алиасы ресурса (например: po для pods, svc для services).
APIVERSION: Группа и версия API, к которой относится ресурс (например: v1, apps/v1, batch/v1).
NAMESPACED: Показывает, является ли ресурс пространственно-ориентированным (namespaced = true) или кластерным (false).
KIND: Тип (Kind) ресурса (например: Pod, Deployment, Service).

Можно фильтровать вывод с помощью флагов:

--namespaced=true или --namespaced=false: показывать только ресурсы в namespace или кластерные (не namespaced).
--api-group=<group_name>: фильтр по конкретной группе API (например: --api-group=apps).
--verbs=<verb1,verb2>: показать ресурсы, поддерживающие указанные операции (например: --verbs=get,list).

Role example

Here's an example Role in the "default" namespace that can be used to grant read access to pods:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ClusterRole example

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

RoleBinding examples

```yaml
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.
# You need to already have a Role named "pod-reader" in that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

Kubernetes RBAC Best Practices

- **Принцип наименьших привилегий**
  - Назначайте только минимально необходимые разрешения пользователям и service accounts.
- **Права на уровне namespace**
  - Используйте `Role` и `RoleBinding` для доступа, ограниченного namespace.
- **Избегайте подстановок**
  - Не применяйте `*` для ресурсов или глаголов (verbs).
- **Непрерывное обновление политик**
  - Регулярно пересматривайте и обновляйте правила RBAC.
- **Ограничение административного доступа**
  - Не используйте `cluster-admin` для повседневных задач; применяйте узкоспециализированные роли.

Diagram: RBAC Best Practices Workflow

```
+-----------------------------+
|  User/Service Account       |
+-----------------------------+
            |
            v
+-----------------------------+
| Assign Least Privilege      |
| (Role/RoleBinding)          |
+-----------------------------+
            |
            v
+-----------------------------+
| Avoid Wildcards             |
| (No "*")                    |
+-----------------------------+
            |
            v
+-----------------------------+
| Review & Update Policies    |
+-----------------------------+
            |
            v
+-----------------------------+
| Limit Cluster-Admin Usage   |
+-----------------------------+
```

Service Accounts в Kubernetes

Основные особенности Service Accounts

`Аутентификация` : Service Accounts предоставляют Pod'ам токены для аутентификации при взаимодействии с API-сервером Kubernetes.
`Авторизация` : С помощью Role-Based Access Control (RBAC) можно управлять правами доступа Service Accounts.
`Изоляция` : Каждый Pod может использовать свой собственный Service Account, что позволяет изолировать доступ и права.

Service Account

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
  namespace: default
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: my-service-account
  containers:
  - name: my-container
    image: my-image
```

Service Account

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Практика

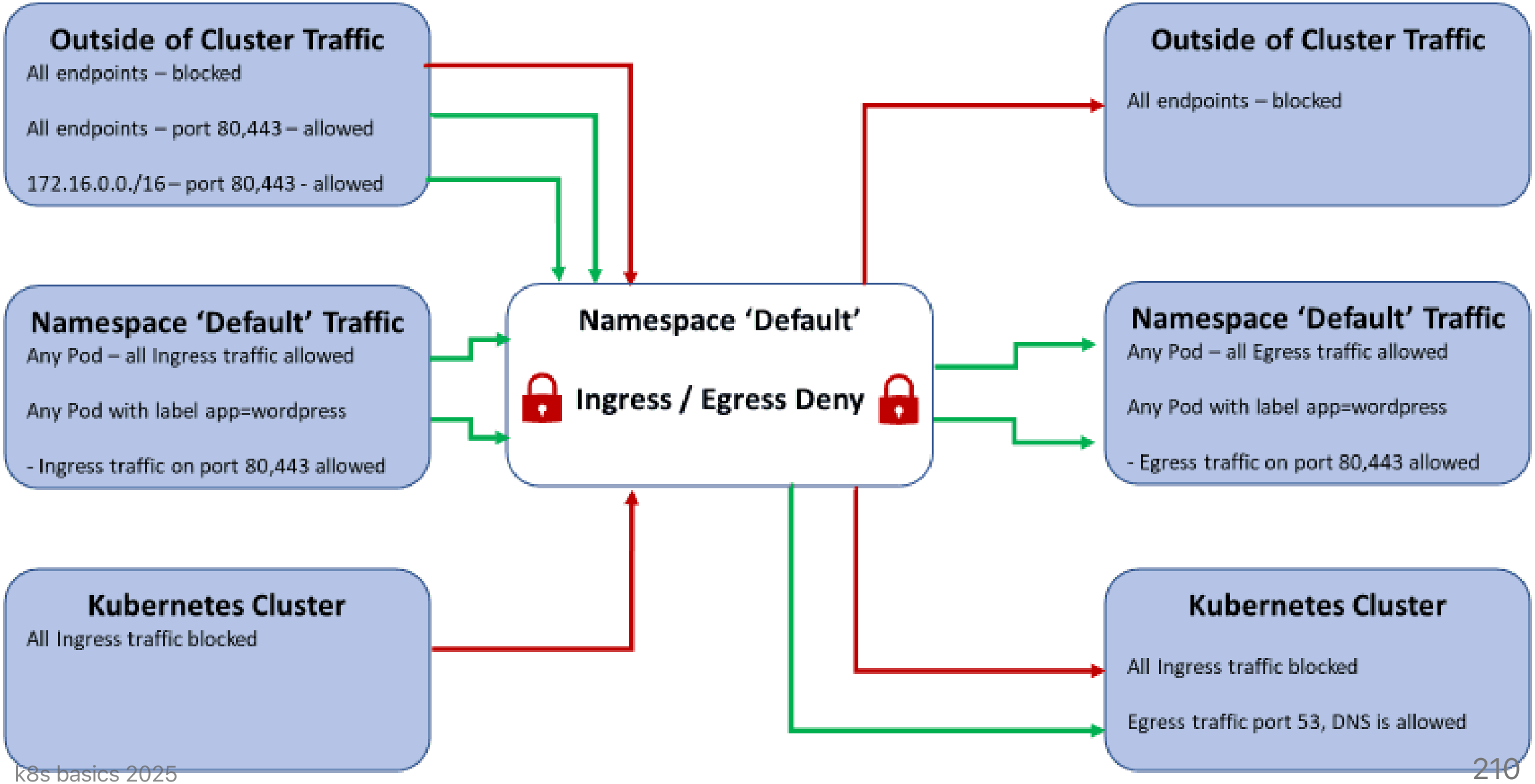
Lab RBAC (Role-Based Access Control) and Service Accounts

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#lab-7-rbac-role-based-access-control-and-service-accounts

Сетевые политики

Сущности, с которыми Pod может обмениваться трафиком, определяются комбинацией следующих трёх идентификаторов:

- Другие разрешённые Pod'ы (исключение: Pod не может заблокировать доступ к самому себе)
- Разрешённые namespaces
- IP-блоки (исключение: трафик к узлу и от узла, на котором работает Pod, всегда разрешён независимо от IP-адресов Pod'а и узла)


Outside of Cluster Traffic
All endpoints – blocked
All endpoints – port 80,443 – allowed
172.16.0.0./16 – port 80,443 - allowed
Namespace 'Default' Traffic
Any Pod – all Ingress traffic allowed
Any Pod with label app=wordpress
- Ingress traffic on port 80,443 allowed
Kubernetes Cluster
All Ingress traffic blocked
Namespace 'Default'
Ingress / Egress Deny
Outside of Cluster Traffic
All endpoints – blocked
Namespace 'Default' Traffic
Any Pod – all Egress traffic allowed
Any Pod with label app=wordpress
- Egress traffic on port 80,443 allowed
Kubernetes Cluster
All Ingress traffic blocked
Egress traffic port 53, DNS is allowed

NetworkPolicy Resource Fields

- **podSelector** — выбирает группу Pod'ов с помощью LabelSelector.
  - Если пустой — выбирает все Pod'ы в namespace (осторожно).
- **policyTypes** — задаёт тип правил: `Ingress`, `Egress` или оба.
- **ingress** — правила для входящего трафика к выбранным Pod'ам.
  - Пустой список: соответствует всему ingress-трафику.
  - Поле отсутствует: не влияет на входящий трафик.
- **egress** — правила для исходящего трафика от выбранных Pod'ов.
  - Пустой список: соответствует всему egress-трафику.
  - Поле отсутствует: не влияет на исходящий трафик.

Example: Basic NetworkPolicy

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-web
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: frontend
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
```

Egress Rules

Массив правил, применяемых к исходящему трафику из Pod. Описывается следующими полями.

Поля:

ports: массив NetworkPolicyPort (port, endPort, protocol)

to: массив NetworkPolicyPeer (ipBlock, namespaceSelector, podSelector)

Ingress Rules

Массив правил, применяемых к входящему в Pod трафику. Определяется следующими полями.

Поля:

from: массив NetworkPolicyPeer (ipBlock, namespaceSelector, podSelector)

ports: массив NetworkPolicyPort (port, endPort, protocol)

Example Code

```yaml
ingress:
    - from:
      - podSelector:
          matchLabels:
            app: wordpress
      ports:
        - port: 443
        - port: 80
    - from:
      - ipBlock:
          cidr: 172.16.0.0/16
      ports:
        - port: 443
        - port: 80
    - from:
      - ipBlock:
          cidr: 0.0.0.0/0
      ports:
        - port: 443
        - port: 80
```

Example code

Finally, we detail the egress section with the outgoing traffic rules. We allow all outgoing traffic communication from Pods labeled webapp on ports 1433 (SQL) and 53 (DNS).

```yaml
egress:
    - to:
        - namespaceSelector: {}
          podSelector:
            matchLabels:
              k8s-app: kube-dns
      ports:
        - port: 53
          protocol: UDP
    - to:
        - podSelector:
            matchLabels:
              app: wordpress
      ports:
        - port: 3306
```

Allow Access to a Group of Pods from Another Namespace

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-debug
spec:
  podSelector:
    matchLabels:
      component: app
  ingress:
  - from:
    - podSelector:
        matchLabels:
          component: debug
      namespaceSelector:
        matchLabels:
          space: monitoring
  policyTypes:
  - Ingress
```

Default Deny Ingress

An all deny ingress rule with an empty podSelector (selecting all pods in the namespace) is a good starting point for a fresh cluster. You can then explicitly allow required traffic. As the podSelector is empty, it will continue to match new pods when they arrive.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress-policy
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Best practices for applying Kubernetes network policies

Как и для любой конфигурации сетевой безопасности, для Kubernetes NetworkPolicy следует применять следующие практики:

- Используйте сетевую политику «deny-all» по умолчанию, чтобы разрешалась только явно разрешённая коммуникация.
- Группируйте Pod'ы, которым нужно взаимодействовать, через параметр podSelector.
- Разрешайте межпространственное (между namespace) взаимодействие только при реальной необходимости.
- Исключайте ненужный сетевой трафик — даже внутри кластера.
- Будьте осторожны, разрешая Pod'ам принимать трафик из внешних (вне кластера) сетей.
- Учитывайте, что запрет исходящего трафика в Интернет может мешать

CNI providers:

- Cilium
- Calico
- Antrea

Полезные технологии для CI/CD: Calico и Cilium

Calico

- **Высокопроизводительная сетевая платформа** для Kubernetes
- **Сетевая политика (Network Policy):**
  Управление трафиком между подами и сервисами
- **L3-маршрутизация:**
  Использует Layer 3 протоколы для эффективной маршрутизации
- **Интеграция с Kubernetes:**
  Поддержка стандартных сетевых политик
- **Гибкая архитектура:**
  Поддержка BGP, IP-in-IP, VXLAN
- **Безопасность:**
  Контроль доступа и шифрование трафика

Пример:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend
spec:
  podSelector:
    matchLabels:
      role: frontend
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: backend
```

Cilium

- **Сетевая платформа на базе eBPF** для Kubernetes
- **Расширенные сетевые политики:**
  Поддержка стандартных и расширенных политик безопасности
- **eBPF:**
  Высокопроизводительная обработка трафика на уровне ядра Linux
- **Мониторинг и трассировка:**
  Инструменты для анализа сетевых взаимодействий
- **Гибкая интеграция:**
  Поддержка VXLAN, Geneve и других бекендов

Пример:

```yaml
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: allow-app
spec:
  endpointSelector:
    matchLabels:
      app: my-app
  ingress:
    - fromEndpoints:
        - matchLabels:
            app: api
```

Диаграмма: Сетевые политики с Calico и Cilium

```
+-------------------+         +-------------------+
|   Pod: frontend   | <-----> |   Pod: backend    |
+-------------------+         +-------------------+
        |                             |
        |   [Network Policy]          |
        +-----------------------------+
        |      Calico / Cilium        |
        +-----------------------------+
        |   [Routing, Security, eBPF] |
        +-----------------------------+
```

10 Kubernetes Security Context settings you should understand

snyk

1. runAsNonRoot /

Always set this to `true` to:

- enforce the use of non-root users for your pod's containers.
- limit access to any host resources that might mistakenly get exposed to the container.

2. runAsUser/runAsGroup /

These settings can be used to enforce a specific runtime user and group.

Use with caution—these IDs must exist in the image for the container to run. Do not use these as a replacement for `runAsNonRoot`.

3. seLinuxOptions /

This sets the `SELinux` context which is applied to the container or pod. Be aware when re-labeling `SELinux` contexts that this may allow unintended access.

4. seccompProfile /

Be cautious when using `seccomp` profiles. Generally, it's okay to provide a profile that is *more* restrictive than the default, as long as your process can run under those restrictions. However, a less restrictive profile can potentially expose calls to the host system that could be dangerous.

5. privileged / allowPrivilegeEscalation

It is usually a bad idea to grant `privileged` access to containers. Use specific capability flags or other Kubernetes APIs instead.

In most cases, you should also explicitly set `allowPrivilegeEscalation` to `false` to stop processes from attaining higher privileges i.e. via `sudo, setuid`.

6. capabilities

Only provide the minimum required for your application to function. Linux capabilities provide fine-grained control over access to kernel-level calls.

7. readonlyRootFilesystem

Set this to `true` whenever possible. In the event your container was to get compromised, a read-write filesystem makes it easier for the attacker to install software or change configurations. Also, consider making any volumes mounted to your container read-only for similar reasons.

8. procMount

Do not change the `procMount` from the Default setting, unless you have very specific configurations—such as nested containers.

9. fsGroup / fsGroupChangePolicy

If other processes depend on the volume's pre-existing GID, changing ownership of a volume using `fsGroup` can have impacts on pod startup performance, as well as possible negative ramifications on shared file systems.

10. sysctls

Modification of kernel parameters via `sysctl` should be avoided—unless you have very specific requirements—as this may destabilize the host operating system.

Eric Smalling
@ericsmalling
Sr. Dev. Advocate at Snyk

Matt Jarvis
@mattj_io
Sr. Dev. Advocate at Snyk

Pod / Container

SecurityContext

`Capabilities` : Позволяет добавлять или удалять Linux capabilities. Исключение лишних capabilities уменьшает поверхность атаки контейнера.

`seccompProfile` : Определяет, какие системные вызовы доступны контейнеру. Ограниченный профиль блокирует опасные или ненужные вызовы.

`AppArmor` : Модуль безопасности ядра Linux (MAC), обеспечивающий принудительный контроль доступа. Может применяться к Pod.

`allowPrivilegeEscalation` : Опция, разрешающая или запрещающая процессу получать привилегии выше, чем у родительского процесса.

`readOnlyRootFilesystem` : При значении true корневая файловая система контейнера монтируется в режиме только чтение, снижая риск изменений системных файлов злоумышленником.

runAsNonRoot [P/C]

Почему это важно

- Контейнеры используют namespaces и cgroups, но ошибка в настройке даёт доступ к ресурсам хоста.
- Root внутри контейнера ≈ root на хосте.
- Ослабление ограничений (procMount, capabilities) + root UID повышают риски.
- Если нет веской причины — не запускать контейнер от root.

Если образ запускается от root — что делать?

- Запускать контейнер под непривилегированным UID через runAsUser (на уровне Pod/Container).
- Либо изменить пользователя в самом образе (Dockerfile).
- Важно: выбранный UID должен иметь доступ к смонтированным файлам/томам.

Вариант 1: базовый образ уже содержит пользователя (node)

Минимальный Dockerfile для node:slim с запуском от пользователя node (UID 1000):

```
FROM node:slim
COPY --chown=node . /home/node/app/   # Копируем код с правильным владельцем
USER node                             # Переключаем активного пользователя
WORKDIR /home/node/app                # Рабочая директория приложения
ENTRYPOINT ["npm", "start"]           # Выполняется от пользователя “node”
```

Вариант 2: в базовом образе нет пользователя

Создать пользователя и запускать процесс от его имени:

```
FROM node:slim
RUN useradd somebody -u 10001 --create-home --user-group  # Создаём пользователя
COPY --chown=somebody . /home/somebody/app/
USER somebody
WORKDIR /home/somebody/app
ENTRYPOINT ["npm", "start"]
```

Примечание: синтаксис useradd/usermod зависит от дистрибутива базового образа.

Примечания и подводные камни

- Для некоторых инструментов потребуется сменить владельца доп. путей (chown).
- Убедиться, что UID может читать/писать в смонтированные тома.
- Предпочтительнее фиксировать USER в образе, а не полагаться на root по умолчанию.

(Опционально) Запуск под конкретным UID в Kubernetes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-nonroot
spec:
  securityContext:
    runAsUser: 1000        # UID непривилегированного пользователя
    runAsGroup: 1000
  containers:
  - name: app
    image: your-registry/app:latest
    securityContext:
      allowPrivilegeEscalation: false
```

- Используйте UID/GID, соответствующие владельцам файлов в образе/томах.
- Совмещайте с ограничением прав (capabilities, readOnlyRootFilesystem).

3. seLinuxOptions [P/C]

- SELinux — MAC в ядре Linux
- Контекст: user:role:type[:level]
- Режимы: Enforcing (блокирует) / Permissive (логирует)
- В контейнерах метки ограничивают доступ процессу к файлам образа

Kubernetes и SELinux

- Runtime задаёт метки по умолчанию
- securityContext.seLinuxOptions — пользовательские метки
- Работает только при включённом SELinux на узле
- Неверные метки могут открыть доступ к ФС хоста — осторожно

Пример (Pod level)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-selinux
spec:
  securityContext:
    seLinuxOptions:
      level: "s0:c123,c456"   # пример уровня (MCS)
  containers:
  - name: app
    image: nginx:1.25
    securityContext:
      seLinuxOptions:
        type: "container_t"   # тип зависит от политики на узле
```

hostPath и SELinux

- Доступ требует корректной метки файлов на узле
- Kubernetes не меняет метки на хосте
- Предпочитать CSI/PV с поддержкой SELinux вместо hostPath

Риски и практики

- Использовать метки по умолчанию; кастом — только при необходимости
- Не задавать spc_t и не включать privileged без веских причин
- Совмещать с runAsNonRoot, readOnlyRootFilesystem, drop capabilities
- Тестировать в Permissive, затем включать Enforcing
- Документировать выбранные контексты и уровни MCS

2. runAsUser / runAsGroup [P/C]

- Контейнерный образ может задавать своего пользователя/группу
- В Pod/Container это можно переопределить через securityContext:
  - runAsUser — UID процесса
  - runAsGroup — GID процесса
- Часто используется вместе с монтированием томов, чтобы совпадали владельцы файлов

Пример (YAML)

```yaml
spec:
  containers:
  - name: web
    image: mycorp/webapp:1.2.3
    securityContext:
      runAsNonRoot: true
      runAsUser: 10001
      runAsGroup: 10001
```

Подводные камни

- Переопределение пользователя на этапе запуска может конфликтовать с образом
- В образе может не существовать такого пользователя (/etc/passwd)
- Риски прав на файлы (владение/доступ), особенно у официальных образов (пример: jenkins)

Пример ошибки (Docker)

```
$ docker run --rm -it -u eric:eric jenkins/jenkins
docker: Error response from daemon: unable to find user eric: no matching entries in passwd file.
```

Рекомендации

- Не полагаться только на runAsUser/runAsGroup
- Всегда указывать runAsNonRoot: true
- Предпочтительнее задавать USER в Dockerfile образа
- Гарантировать корректные владельцы/права для смонтированных томов
- Тестировать запуск с выбранным UID/GID до деплоя

run as user example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000  # Pod Level
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000 # Individual Container level
      allowPrivilegeEscalation: false
```

seccompProfile [P/C]

- Seccomp (secure computing mode) — функция ядра Linux, ограничивающая системные вызовы процесса
- Профиль seccomp — JSON-описание: список разрешённых/запрещённых syscalls и действие по умолчанию
- Kubernetes предоставляет механизм для использования настраиваемых профилей через параметр seccompProfile в securityContext.

```
seccompProfile:
      type: Localhost
      localhostProfile: profiles/myprofile.json
```

пример seccomp json файла

```
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "architectures": [
    "SCMP_ARCH_X86_64",
    "SCMP_ARCH_X86",
    "SCMP_ARCH_X32"
  ],
  "syscalls": [
    { "name": "accept",  "action": "SCMP_ACT_ALLOW", "args": [] },
    { "name": "accept4", "action": "SCMP_ACT_ALLOW", "args": [] }
    // ...
  ]
}
```

Source: https://training.play-with-docker.com/security-seccomp/

Доступно три значения для поля **type**:

`Localhost` - в дополнительном параметре localhostProfile указан путь к профилю seccomp

`Unconfined` - профиль не применяется

`RuntimeDefault` - используется значение по умолчанию для среды выполнения контейнера (это значение по умолчанию, если тип не указан)

Принцип минимальных привилегий (Seccomp)

- Предоставляйте контейнеру только необходимые привилегии.
- Начните с профиля аудита, который регистрирует системные вызовы.
- Протестируйте приложение, чтобы определить нужные syscalls.
- Сформируйте разрешающий профиль, оставив только минимально необходимый набор.

Linux kernel capabilities [C]

- Кратко: Linux capabilities — гранулярные привилегии ядра. В Kubernetes ими управляют через securityContext.capabilities. Если capabilities не заданы, контейнер получит значения по умолчанию от CRI.

Пример (рекомендуемый базовый шаблон):

```
securityContext:
  capabilities:
    drop: ["ALL"]
    add: ["MKNOD"]  # добавьте только то, что реально нужно
```

Рекомендации:

- Сначала отключайте все capabilities (drop: ["ALL"]), затем точечно добавляйте необходимые.
- Часто приложениям не требуется ни одна capability при нормальной работе.

Linux Capabilities:

K8s capability	Linux capability	Description
NET_RAW	CAP_NET_RAW	Allows creating raw/packet sockets (e.g., ping, packet crafting). Use only when strictly needed.
NET_ADMIN	CAP_NET_ADMIN	Broad network administration (interfaces, routes, iptables, routing tables). Powerful; restrict usage.
SYS_ADMIN	CAP_SYS_ADMIN	Extremely privileged (mounts, namespaces, device management). Root-like; avoid in production.
DAC_OVERRIDE	CAP_DAC_OVERRIDE	Allows bypassing file read/write/execute permission checks.
CHOWN	CAP_CHOWN	Allows changing the ownership of files.
SETGID / SETUID	CAP_SETUID/CAP_SETGID	Allows setting the user ID and group ID of a process.
KILL	CAP_KILL	Allows sending signals to any process.

CRI-O default capabilities

```
default_capabilities = [
        "CHOWN",
        "DAC_OVERRIDE",
        "FSETID",
        "FOWNER",
        "SETGID",
        "SETUID",
        "SETPCAP",
        "NET_BIND_SERVICE",
        "KILL",
]
```

Containerd default capabilities

```
func defaultUnixCaps() []string {{
                "CAP_CHOWN",
                "CAP_DAC_OVERRIDE",
                "CAP_FSETID",
                "CAP_FOWNER",
                "CAP_MKNOD",
                "CAP_NET_RAW",
                "CAP_SETGID",
                "CAP_SETUID",
                "CAP_SETFCAP",
                "CAP_SETPCAP",
                "CAP_NET_BIND_SERVICE",
                "CAP_SYS_CHROOT",
                "CAP_KILL",
                "CAP_AUDIT_WRITE",
}
```

Capabilities example

With Linux capabilities, you can grant certain privileges to a process without granting all the privileges of the root user.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

seccompProfile example

There are three possible values for the type field: \

- `Unconfined` No Seccomp profile is applied, meaning the workload can make any system call. This is generally discouraged for security reasons.
- `RuntimeDefault` The container runtime's default Seccomp profile is applied. This profile typically offers a good balance of security and functionality. this is the default if the type is left unspecified.
- `Localhost` with which a localhostProfile setting provides a path inside the container to a seccomp profile.
- `localhostProfile` This field is only used when type is set to Localhost. It specifies the path to the custom Seccomp profile file on the node, relative to the kubelet's configured Seccomp profile location (usually /var/lib/kubelet/seccomp/).

```yaml
securityContext:
  seccompProfile:
    type: RuntimeDefault
```

> Note that Seccomp profiles cannot be applied to containers running with `privileged: true`.

readOnlyRootFilesystem [C]

- Монтируйте root ФС контейнера в режиме read-only
- Логи — только stdout/stderr; состояние — во внешних хранилищах/volume
- Для записи используйте отдельные тома (tmp, runtime-директории)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-ro-rootfs
spec:
  containers:
  - name: app
    image: your-registry/app:latest
    securityContext:
      readOnlyRootFilesystem: true
      runAsNonRoot: true
    volumeMounts:
    - name: runtime
```

procMount [C]

- Значения: Default (по умолчанию) или Unmasked (открывает /proc)
- Разрешайте Unmasked только при строгой необходимости (напр., вложенные контейнеры в сборке)
- Никогда не открывайте /proc хоста контейнеру

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: builder
spec:
  containers:
  - name: dind-like
    image: your-registry/builder:latest
    securityContext:
      procMount: Unmasked   # применяйте точечно и временно
```

fsGroup / fsGroupChangePolicy [P]

- fsGroup назначает GID для файлов в смонтированных volume
- fsGroupChangePolicy: OnRootMismatch или Always
- Осторожно на больших/медленных FS: возможны долгие chmod/chown
- Может ломать совместный доступ, если процессы не имеют прав нового GID
- Не влияет на ephemeral volume

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-fsgroup
spec:
  securityContext:
    fsGroup: 2000
    fsGroupChangePolicy: OnRootMismatch
  volumes:
  - name: data
```

sysctls [P]

- Разрешает менять подмножество sysctls из пространства имён контейнера
- Используйте только безопасные sysctls; небезопасные включаются администратором
- Избегайте изменений без явной необходимости; согласуйте с админами кластера

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-sysctls
spec:
  securityContext:
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "1"
    - name: net.ipv4.ip_local_port_range
```

Практика

Lab Network Policies for Micro-segmentation

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_advanced_labs.md#lab--network-policies-for-micro-segmentation

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#lab-4-network-policies

Хранение данных в Kubernetes.

StatefulSet

`StatefulSet` — это контроллер, который управляет развертыванием и масштабированием набора Pod'ов, и обеспечивает их уникальные, стабильные сетевые идентификаторы и постоянное хранилище.

Основные особенности:
Стабильные сетевые идентификаторы: Каждый Pod в `StatefulSet` имеет постоянное имя, которое сохраняется при перезапусках.
Постоянное хранилище: Каждый Pod может иметь свой собственный `PersistentVolume`, который сохраняется при перезапусках.
Упорядоченное развертывание и обновление: Pod'ы создаются и обновляются по одному, в определенном порядке.

StatefulSet

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-stateful-app
spec:
  serviceName: "my-service"
  replicas: 3
  selector:
    matchLabels:
      app: my-stateful-app
  template:
    metadata:
      labels:
        app: my-stateful-app
    spec:
      containers:
      - name: my-container
        image: my-image
        volumeMounts:
        - name: my-persistent-storage
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: my-persistent-storage
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

StorageClass

`StorageClass` предоставляет способ описания "классов" хранилища, которые могут быть предложены администратором кластера. Разные классы могут отображать различные уровни качества обслуживания, политики резервного копирования или политики восстановления.

Основные особенности:

Динамическое выделение: Позволяет автоматически создавать `PersistentVolume` на основе запросов `PersistentVolumeClaim`.

Различные параметры хранилища: Можно задать параметры, такие как тип диска, скорость ввода-вывода и т.д.

StorageClass

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  fsType: ext4
```

PersistentVolumeClaim (PVC)

`PersistentVolumeClaim` (PVC) — это запрос на хранилище, который подается пользователем. `PVC` позволяет Pod'ам запрашивать и использовать постоянное хранилище, предоставляемое `PersistentVolume` (PV). `PVC` абстрагирует детали хранилища, такие как тип диска и его местоположение, и позволяет пользователям запрашивать хранилище по определенным параметрам.

Основные особенности:

Запрос хранилища: Пользователи могут запрашивать хранилище определенного размера и с определенными характеристиками.

Динамическое выделение: PVC может автоматически создавать `PV` на основе `StorageClass`.

Привязка к PV: PVC привязывается к PV, который соответствует запросу, и Pod'ы могут использовать это хранилище.

PVC

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: fast
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    volumeMounts:
    - mountPath: /data
      name: my-volume
  volumes:
  - name: my-volume
    persistentVolumeClaim:
      claimName: my-pvc
```

Volume

`Volume` — это абстракция, которая позволяет контейнерам в Pod'е совместно использовать данные. Kubernetes поддерживает множество типов `Volume`, таких как **emptyDir**, **hostPath** и другие.

Основные особенности:
Совместное использование данных: Все контейнеры в Pod'е могут совместно использовать Volume.
Различные типы хранилища: Поддержка различных типов хранилища для различных сценариев использования.

Volume

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    volumeMounts:
    - mountPath: /data
      name: my-volume
  volumes:
  - name: my-volume
    persistentVolumeClaim:
      claimName: my-pvc
```

The Kubernetes storage access modes (RWO, ROX, and RWX) define how a volume can be mounted by pods, specifying the permissions (read/write) and how many nodes can access it simultaneously

Access Mode	Full Name	Description
RWO	ReadWriteOnce	The volume can be mounted as read-write by a single node at a time. Multiple pods on the same node can share it, but pods on different nodes cannot.
ROX	ReadOnlyMany	The volume can be mounted as read-only by many nodes simultaneously. Multiple pods across different nodes can read data concurrently, but none can write to it.
RWX	ReadWriteMany	The volume can be mounted as read-write by many nodes simultaneously. Multiple pods across different nodes can read and write data concurrently.

Use Cases:

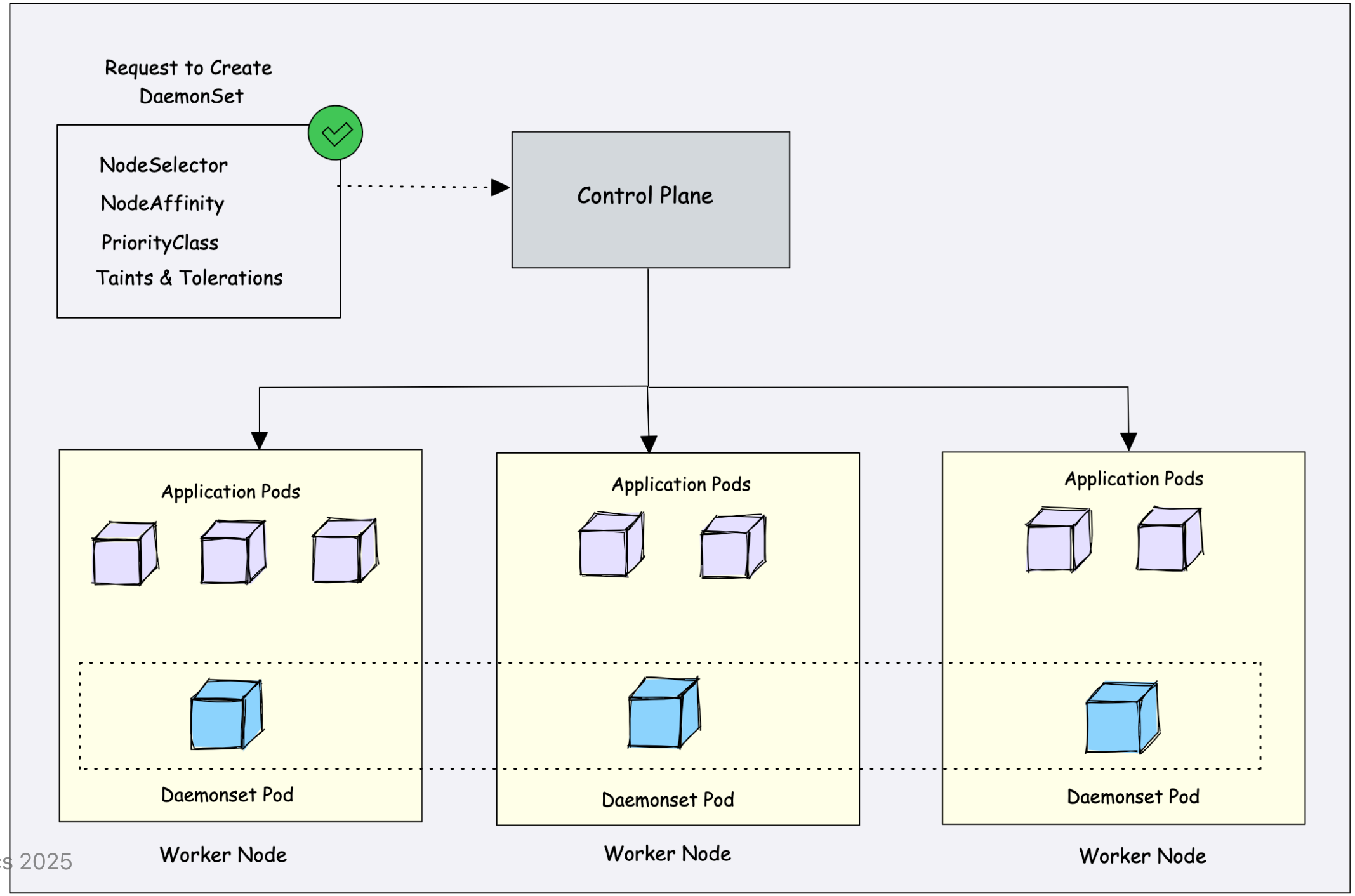
RWO is ideal for stateful applications like databases where only a single instance should have write access to prevent data corruption.
RWX is necessary for applications that require shared filesystem access across multiple nodes, such as content management systems or build farms.
ROX is useful for distributing static configuration files or shared libraries that do not need to be modified at runtime.

Lab Persistent Volumes and StatefulSets

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#lab-3-persistent-volumes-and-statefulsets

Lab StatefulSets with Persistent Storage

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_advanced_labs.md#lab-6-statefulsets-with-persistent-storage


Request to Create
DaemonSet
NodeSelector
NodeAffinity
PriorityClass
Taints & Tolerations
Control Plane
Application Pods
Daemonset Pod
Worker Node
Application Pods
Daemonset Pod
Worker Node
Application Pods
Daemonset Pod
Worker Node

Some typical uses of a DaemonSet are:

- запуск демона хранилища кластера на каждом узле
- запуск демона сбора логов на каждом узле
- запуск демона мониторинга узлов на каждом узле

DaemonSet

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
```

```yaml
containers:
- name: fluentd-elasticsearch
  image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
  resources:
    limits:
      memory: 200Mi
    requests:
      cpu: 100m
      memory: 200Mi
  volumeMounts:
  - name: varlog
    mountPath: /var/log
# priorityClassName: important
terminationGracePeriodSeconds: 30
volumes:
- name: varlog
  hostPath:
    path: /var/log
```

Разбор функционала и особенностей сущностей Job и CronJob

`Jobs` и `CronJobs` в Kubernetes

Job

`Job` в Kubernetes — это контроллер, который создает один или несколько Pod'ов и гарантирует, что определенное количество Pod'ов успешно завершится. `Job` полезен для выполнения одноразовых задач, таких как обработка данных или выполнение скриптов.
В отличие от `Deployment` или `ReplicaSet`, которые управляют постоянно работающими сервисами, `Jobs` предназначены для одноразовых задач с гарантией завершения.

Проще говоря:

> 🔥 «Если нужно запустить что-то один или несколько раз — и важно, чтобы это успешно завершилось — используйте `Job`.»

🛠️ Ключевые особенности:

- Одноразовый запуск: `Pod` выполняется до успешного завершения.
- Автоматические повторы: при сбое `Pod` перезапускается (согласно конфигурации).
- Отслеживание завершения: `Job` учитывает, сколько успешных `Pod`'ов нужно, чтобы считать задачу выполненной.
- Параллельное выполнение: может запускать несколько `Pod`'ов для ускорения (parallelism).
- Обработка сбоев: тонкая настройка ретраев, дедлайнов и лимитов.

Примеры использования:

1. Миграции базы данных при развёртываниях

- ✅ Гарантирует готовность БД до начала обработки трафика приложением.

2. Пакетная обработка данных

- `Job` может запускать несколько `Pod`'ов параллельно для ускорения обработки.

3. Запуск разовых (ad-hoc) отчётов по расписанию

- Вместо встраивания в постоянный сервис запускайте `Job` по требованию.

4. Перенос файлов или данных между системами
5. Обработка изображений в медиа-приложениях

Parallelism and Completions

- `spec.parallelism` — число Pod'ов, выполняющихся одновременно.
  - 1 — последовательная обработка
  - > 1 — параллельное выполнение
- spec.completions — сколько успешных запусков нужно выполнить, чтобы Job считалась завершённой (общее число успешных Pod).
- Важно: обычно completions >= parallelism, чтобы все “шарды” дошли до конца без простаивания.
- completionMode:
  - NonIndexed (по умолчанию) — все Pod одинаковые
  - Indexed — каждому Pod выдаётся индекс [0..completions-1] (env JOB_COMPLETION_INDEX) для шардирования/идемпотентной обработки.

```
spec:
```

Backoff Limits and Failure Handling

- Jobs устойчивы к временным сбоям: автоматически повторяют неудачные попытки.
- Поле **.spec.backoffLimit** задаёт максимум повторов запуска Pod; при достижении лимита Job помечается как Failed.
- Связь с **.spec.completions**: если нужное число успешных завершений достигнуто до исчерпания попыток — Job успешна; иначе — неуспешна.
- **.spec.template.spec.restartPolicy** уточняет перезапуски внутри Pod:
  - **Never** — контейнеры не перезапускаются; каждая неудача создаёт новый Pod (засчитывается в backoffLimit).
  - **OnFailure** — kubelet перезапускает контейнер при ненулевом коде; перезапуск контейнера не увеличивает backoffLimit. Новый Pod создаётся и считается попыткой только если весь Pod завершён в статусе Failed.
- Важно: **backoffLimit считает повторы Pod, а не перезапуски контейнера**

Timeouts and Deadlines

Управление тайм-аутами и дедлайнами для Kubernetes Job критично для предотвращения «убегающих» процессов и эффективного использования ресурсов. Ниже показано, как задать ограничение по времени и какие есть альтернативы для более тонкого контроля.

`activeDeadlineSeconds`

Поле `.spec.activeDeadlineSeconds` в спецификации Job задаёт общий предельный интервал (в секундах) для выполнения всей Job. Если Job не завершилась за указанное время, Kubernetes помечает её как Failed.

```yaml
spec:
  activeDeadlineSeconds: 3600 # 1 hour
```

Automatic Cleanup with TTLs

Чтобы предотвратить накопление завершённых `Job` и избыточное потребление ресурсов, используйте поле `ttlSecondsAfterFinished`. Этот параметр автоматически удаляет завершённые `Job` по истечении заданного количества секунд. Например, значение `ttlSecondsAfterFinished: 3600` удалит `Job` через час после её завершения. Такая автоматическая очистка упрощает обслуживание кластера и предотвращает раздувание потребления ресурсов.

```yaml
spec:
  ttlSecondsAfterFinished: 3600
```

Job

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: my-job
spec:
  template:
    metadata:
      name: my-job
    spec:
      containers:
      - name: my-container
        image: busybox
        command: ["sh", "-c", "echo Hello, Kubernetes! && sleep 30"]
      restartPolicy: OnFailure
  backoffLimit: 4
```

CronJob

`CronJob` создает Jobs на основе расписания, определенного в формате Cron. Они автоматизируют повторяющиеся задачи в кластерах Kubernetes, упрощая операции и повышая эффективность. Это полезно для выполнения периодических задач, таких как резервное копирование данных или отправка отчетов.

> Tip For cron expression I am using https://crontab.guru/

Основные параметры CronJob

`schedule` : Определяет расписание в формате Cron.

`jobTemplate` : Шаблон для создания Job'ов.

`successfulJobsHistoryLimit` : Количество успешных Job'ов, которые будут храниться.

`failedJobsHistoryLimit` : Количество неудачных Job'ов, которые будут храниться.

`concurrencyPolicy` : политика одновременного запуска.

Real-world Use Cases for Kubernetes CronJobs:

Резервное копирование данных: Планируйте регулярные бэкапы критически важных данных, чтобы обеспечить целостность и готовность к аварийному восстановлению.

Синхронизация данных: Периодически синхронизируйте данные между различными системами или базами данных для поддержания согласованности и актуальности информации.

Периодическое обслуживание: Автоматизируйте рутинные задачи обслуживания, такие как ротация логов, очистка баз данных или удаление временных файлов, чтобы оптимизировать производительность системы и использование ресурсов.

- **Запуск заданий по расписанию в кластере:** CronJob в Kubernetes позволяет выполнять задачи по фиксированному расписанию. Независимо от конфигурации узлов и установленных на них инструментов вы можете запускать задания с нужной периодичностью.
- **Экономия ресурсов кластера:** CronJob потребляет ресурсы только во время выполнения, что предотвращает их простой. Без CronJob пришлось бы держать Deployment или аналогичный ресурс постоянно запущенным (и потребляющим ресурсы), даже если он нужен лишь периодически.
- **Надёжное соблюдение расписания в нагруженных системах:** CronJob выполняются независимо от других ресурсов Kubernetes, что помогает гарантировать запуск в запланированное время. В отличие от Deployment и подобных ресурсов, их выполнение реже задерживается из-за дефицита ресурсов или состояния Pending у подов.

CronJobs with concurrency policies

Вы можете выбрать одну из трёх политик одновременного запуска (concurrencyPolicy) при настройке CronJob:

- `Allow` — контроллер запускает новый Job, даже если предыдущий ещё выполняется.
- `Forbid` — запрещает запуск нового Job, пока предыдущий не завершится.
- `Replace` — останавливает текущий Job и запускает новый вместо него.

```yaml
spec:
  concurrencyPolicy: Forbid
```

concurrencyPolicy полезны, так как позволяют предотвращать множественные параллельные запуски, которые тратят ресурсы и в крайних случаях могут привести к падению кластера.

CronJobs with history controls

Параметры истории определяют, сколько записей о предыдущих запусках заданий Kubernetes должен хранить. Вы можете задать требования как для записей об успешных запусках (с помощью переменной successfulJobsHistoryLimit), так и для неудачных (с помощью failedJobsHistoryLimit).

```
spec:
  successfulJobsHistoryLimit: 5
  failedJobsHistoryLimit: 2
```

Параметры истории полезны главным образом тем, что помогают избежать лишних затрат ресурсов на хранение записей о прошлых запусках заданий.

Optimizing Kubernetes CronJob performance — key points

- Планируйте задания только при необходимости; избегайте слишком частых запусков (например, каждую минуту или каждый час) без явной причины.
- Используйте минимальные образы контейнеров и только необходимые утилиты; упрощайте команды, чтобы снизить накладные расходы.
- Разносите расписания CronJob, чтобы избежать наложений; не планируйте несколько задач в один и тот же временной интервал.
- Предпочитайте часы вне пиковых нагрузок кластера; избегайте периодов пиковой нагрузки с учётом часового пояса кластера.

Best practices for managing CronJobs

- Мониторьте статус и логи. Используйте kubectl get cronjobs/jobs и kubectl logs для проверки запусков и результатов.
- Настраивайте resource requests и limits. Указывайте CPU/память в jobTemplate, чтобы избежать сбоев; резервируйте достаточно ресурсов для завершения задач.
- Помечайте и аннотируйте задания. Добавляйте метаданные о владельце, назначении и жизненном цикле для упрощения отслеживания и очистки.
- Настройте оповещения о сбоях. Экспортируйте метрики через kube-state-metrics и настраивайте алерты (например, Prometheus/Alertmanager) на неудачные/пропущенные запуски.
- Защищайте секреты и конфигурации. Используйте Kubernetes Secrets, избегайте открытого текста в манифестах/образах/логах и применяйте RBAC/принцип наименьших привилегий.

CronJob

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-cronjob
spec:
  schedule: "*/5 * * * *"  # Запуск каждые 5 минут
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: my-container
            image: busybox
            command: ["sh", "-c", "echo Hello, Kubernetes! && sleep 30"]
          restartPolicy: OnFailure
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
```

Lab Jobs and CronJobs for Batch Processing

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_advanced_labs.md#lab-7-jobs-and-cronjobs-for-batch-processing