DevOps Pro

About Me:

Aliaksandr Tsimokhau - Senior DevOps Engineer

- Passionate DevOps engineer with 10+ years in IT, including 6+ years focused on DevOps
- Experienced in building, optimizing, and scaling infrastructure to empower teams and deliver results
- **Current Role:** Technical Support Lead at UFL Strikerz Inc (gaming company)
    - Build infrastructure for high-performance gaming platforms
    - Bridge operations and development for smooth workflows

What I Do:

Infrastructure as Code: Terraform for scalable, reliable, automated infrastructure
Kubernetes Expertise: Maintain clusters for seamless orchestration and container management
Helm Chart Development: Write custom charts for simplified deployments
Monitoring & Alerting: Design systems with Prometheus & VictoriaMetrics for real-time visibility
Cloud Proficiency: Work across GCP and AWS, leveraging cloud-native tools
CI/CD Pipelines: Build and maintain robust GitLab CI pipelines for automated testing, building, and deployment

DAY ONE

Part 1. **Introduction to DevOps and CI/CD**

Part 2. **Kubernetes Foundations**

Schedule

Время	Событие
10:00	Начало курса
11:15–11:30	Перерыв
13:00–14:00	Обед
15:15–15:30	Перерыв
17:00–17:30	Завершение

Software Development Lifecycle (SDLC)

- **Как разрабатывается программное обеспечение:**
  Процесс от идеи до развертывания и сопровождения.
- **Общие этапы включают:**
  requirements gathering -> design -> implementation (coding) -> testing -> deployment and maintenance.
  сбор требований → проектирование → реализация (кодирование) → тестирование → развертывание и сопровождение.

DevOps Principles

- **Collaboration & Communication:** Поощряйте совместную ответственность на всех этапах жизненного цикла программного обеспечения.
- **Automation First:** Автоматизируйте всё, что возможно, чтобы процессы были повторяемыми, надёжными и масштабируемыми.
- **Continuous Everything:** Частые слияния, тестирование и доставка изменений (CI/CD, мониторинг, обратная связь, тестирование).
- **Security as Shared Responsibility:** Интегрируйте безопасность на ранних этапах разработки ("Shift Left"); разработчики, команда оперирования и специалисты по безопасности работают вместе.
- **Customer-Centric Action:** Доставляйте ценность на основе обратной связи пользователей и выстраивайте приоритеты в соответствии с бизнес-целями.

Инструменты и технологии DevOps

- **CI/CD:** Jenkins, GitLab CI, GitHub Actions, CircleCI
- **Контейнеризация:** Docker, Podman
- **Оркестрация:** Kubernetes, OpenShift
- **Инфраструктура как код:** Terraform, Ansible, Helm
- **Мониторинг и логирование:** Prometheus, Grafana, ELK Stack, OpenTelemetry

DevOps Practices

- **Version Control:** Весь код и конфигурация должны храниться в системе управления версиями (например, Git).
- **Continuous Integration:** Автоматическая сборка и тестирование при изменениях в коде.
- **Continuous Delivery/Deployment:** Автоматически готовьте и выкатывайте сборки в production.
- **Инфраструктура как код (IaC):** Определяйте и создавайте инфраструктуру с помощью кода (например, Terraform).

DevOps Practices

- **Configuration Management:** Автоматически поддерживать и управлять конфигурациями (например, с помощью Ansible).
- **Monitoring & Observability:** Организуйте практически в реальном времени видимость производительности приложений и инфраструктуры.
- **Containerization & Orchestration:** Упаковывайте приложения в контейнеры и масштабируйте их под нагрузкой (например, Docker, Kubernetes).
- **GitOps:** Используйте Git как единственный источник правды с автоматической синхронизацией (reconciliation).
- **Security:** Интегрируйте проверки безопасности в CI/CD пайплайны.

Принципы CI/CD

- **CI (Непрерывная интеграция):** регулярное слияние изменений в общий репозиторий, автоматическое тестирование и сборка.
- **CD (Непрерывная доставка/развертывание):** автоматизация доставки изменений в тестовые и/или продуктивные среды.

DevSecOps

Что такое DevSecOps?

- Интеграция безопасности в CI/CD пайплайны
- Автоматизация и непрерывность проверок безопасности
- Ответственность за безопасность разделена между всеми

DevSecOps-пайплайн: Безопасность на каждом этапе

- **Сборка:** Анализ зависимостей, статический анализ кода (SAST)
- **Тестирование:** Динамический анализ (DAST), сканирование уязвимостей
- **Развертывание:** Проверка конфигураций, хаос-инжиниринг
- **Мониторинг:** Обнаружение угроз в реальном времени, реагирование на инциденты

Четыре ключевые метрики DevOps

- **Lead time for changes** Время выполнения изменений
- **Change Failure Rate** Процент неудачных изменений
- **Deployment Frequency** Частота развертываний
- **MTTR** Среднее время восстановления

Антипаттерны DevOps

- **Люди:** Культура "супергероев", hidden silos (антипаттерн скрытых изолированных зон), выгорание
- **Процессы:** Большие релизы, ручные исправления, отсутствие документации
- **Технологии:** Ручная настройка, вера в "серебряную пулю", чрезмерная автоматизация

Важность мониторинга

- Непрерывный мониторинг во всех средах
- Быстрое обнаружение и реагирование на инциденты
- Поддержка экспериментов, управления изменениями и отслеживания зависимостей

Лучшие практики оповещений и управления инцидентами

- Формируйте культуру сотрудничества
- Автоматизируйте оповещения и реагирование на инциденты
- Обучайте команды работе с дашбордами и инструментами мониторинга
- Интегрируйте оповещения с рабочими процессами разработки

DevOps Tools Overview

- **Version Control:** Git
- **CI/CD:** GitLab CI/CD, GitHub Actions, Jenkins, Azure DevOps
- **Infrastructure as Code:** Terraform, AWS CloudFormation, Pulumi
- **Configuration Management:** Ansible, Puppet
- **Monitoring & Observability:** Prometheus, Grafana, AWS CloudWatch, DataDog
- **Security:** Snyk, Trivy, checkov
- **Containerization & Orchestration:** Docker, Kubernetes, Helm
- **GitOps:** Flux CD, Argo CD

Part 2. Kubernetes Foundations

- **Kubernetes abstracts resources:**
  - Compute (CPU, memory)
  - Network
  - Storage
- **No need to hard-code node or storage details**
- **Declarative deployment:**
  - "Here's my app, Kubernetes—please deploy and keep it running!"

Diagram: Kubernetes Resource Abstraction

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

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 - 20-30min

Kubernetes, основные понятия и архитектура

Преимущества Kubernetes для разработки и развёртывания приложений

Особенности проектирования приложений для Kubernetes

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.

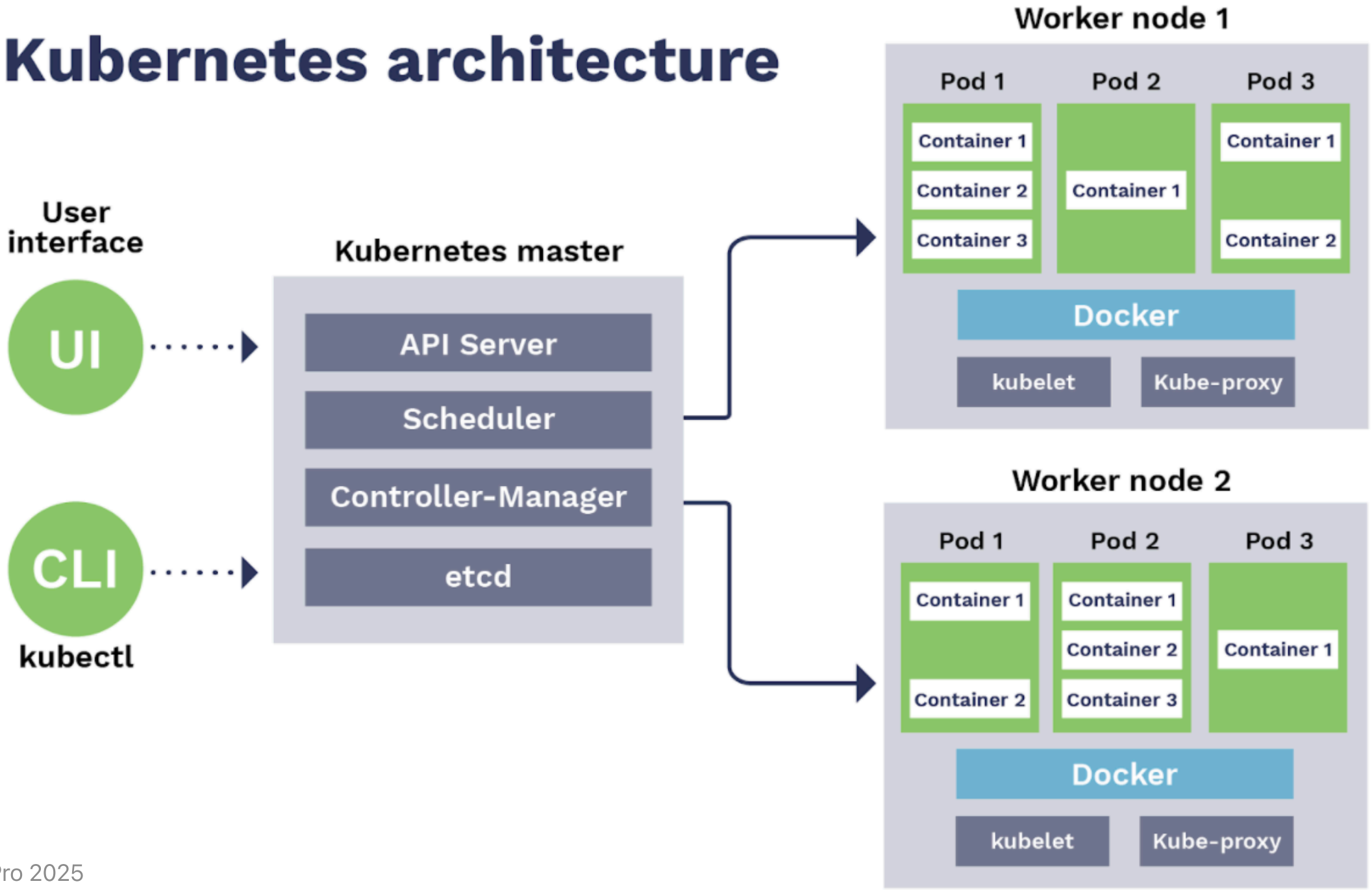
6. Change management within existing containered applications.

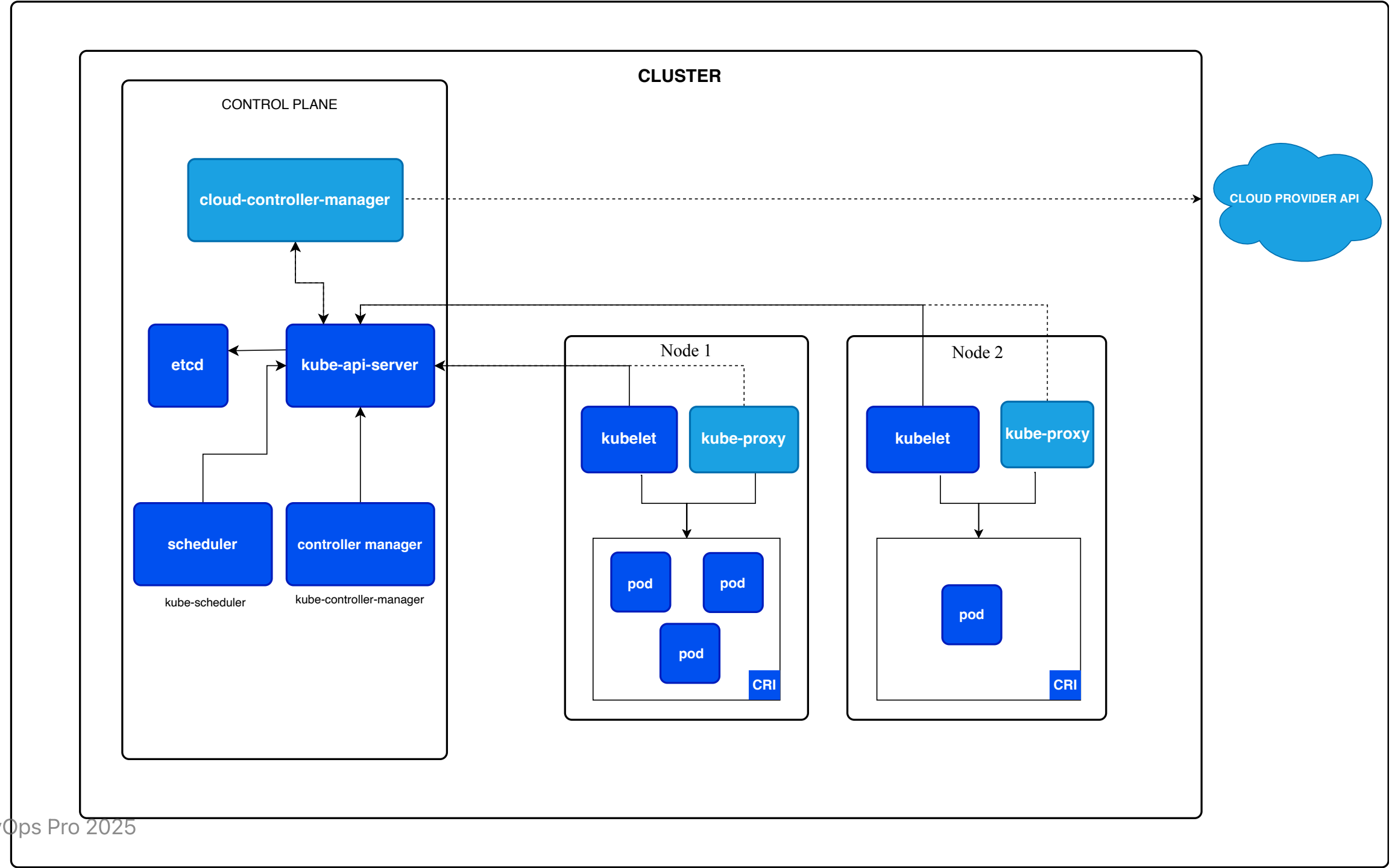
4. Helps optimize the use of underlying hardware found below your container.

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

ClickIT
DevOps & Software Development

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

Nodes:

Nodes are the physical or virtual machines in the Kubernetes cluster responsible for running application workloads. Each node contains the necessary components to manage pods, such as the Kubelet, container runtime, and Kube-proxy.

Nodes can be Master nodes (control plane) or Worker nodes (application execution).

Master Node manages the entire cluster, including API server, controller manager, etcd (distributed key-value store), and scheduler

Pods:

A Pod is the smallest and simplest Kubernetes object, representing a single instance of a running application. It can contain one or more tightly coupled containers that share the same network and storage resources.
Pods are ephemeral; if a pod fails, Kubernetes replaces it with a new pod, which may have a different IP address.

Containers:

Containers are lightweight, portable, and isolated units that run the application code.
Kubernetes typically uses Docker or other container runtimes to manage containers inside pods.
Kubernetes manages containers inside pods, handling their creation, scaling, and termination.

ReplicaSets:

ReplicaSets ensure a specified number of identical pod replicas are running at any given time. If a pod fails or is terminated, the ReplicaSet creates a new pod to maintain the desired state.
They help achieve high availability and fault tolerance by maintaining a specific number of pods even when pods crash or are deleted.

Deployments:

A Deployment manages the lifecycle of applications, handling the rollout of new versions, scaling, and rollbacks. It provides declarative updates to applications and ensures the desired state of pods using ReplicaSets.
Deployments allow you to define the container image, replicas, and update strategies in a declarative manner, automating the process of updates.

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:
  --------
  --------
```

Where is the Kubeconfig file located?

All the kubeconfig files are located in the .kube directory in the user home directory. That is

```
$HOME/.kube/config
```

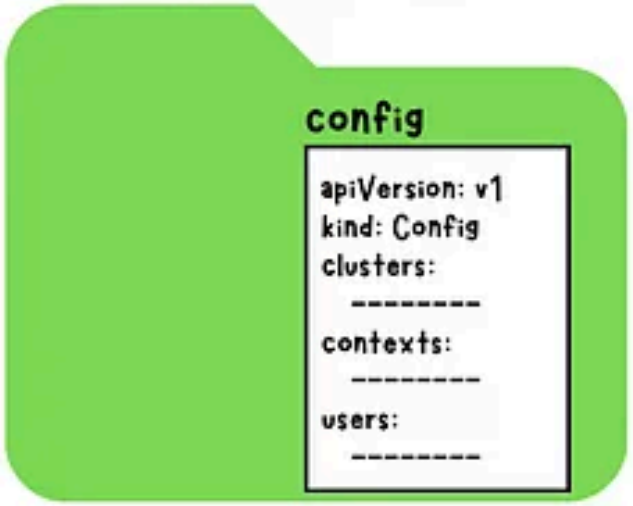
Kubeconfig File Path

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
```

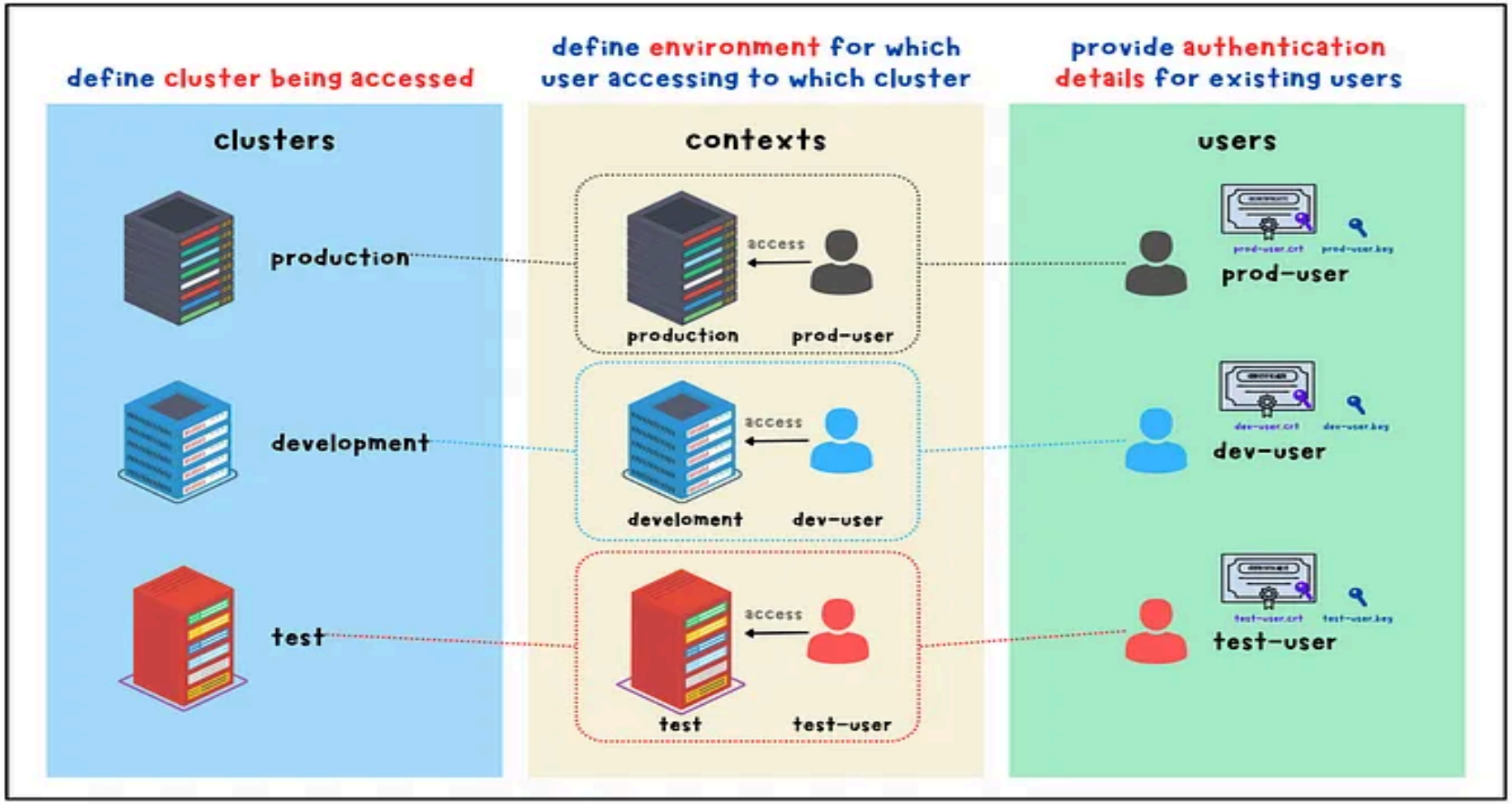
How to manage multiple Kubeconfig files?

You can store all the kubeconfig files in `$HOME/.kube directory`. You need to change the cluster context to connect to a specific cluster.

The `current-context` specifies which context should be used by default when kubectl is run.

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
```

Работа с Kubernetes

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` (PVs) и `Persistent Volume Claims` (PVCs): 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**: A group of one or more containers sharing resources and execution environment.
- **Shared resources** (via Linux namespaces):
  - **Network (net):** Shared IP address, port range, routing table
  - **Process (pid):** Isolated process tree
  - **Mount (mnt):** Shared filesystems and volumes
  - **UTS:** Shared hostname
  - **IPC:** Shared Unix sockets and memory
- **Pause container**: Special container that holds and manages shared namespaces
  - Enables containers to share network and filesystem
  - Keeps network namespace alive for Pod restarts

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

- **Each Pod has its own network namespace**
  - All containers in a Pod share the same IP address, port range, and routing table
- **External access** to containers in a Pod is via the Pod's IP and container port
  - Example: `10.0.10.15:80` or `10.0.10.15:5000`
- **Container-to-container communication** inside a Pod uses `localhost` and port
  - Example: `localhost:5000` to reach a sidecar or helper container
- **Pod network is flat:**
  - Every Pod can talk directly to every other Pod in the cluster
- **By default, the Pod network is open**
  - Use Kubernetes Network Policies to secure communication

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

- **Pods are ephemeral:** Created, live, and die—if a Pod fails, Kubernetes creates a new one with a new ID and IP.
- **Immutability:** Once running, Pods are never changed or updated in-place.
- **No manual changes:** Never log in to a running Pod to update configuration; replace with a new Pod instead.

Deploying Pods: Static vs. Managed

- **Direct deployment:**
  - Use a Pod manifest (YAML) to create "static" Pods.
  - Managed by the node's kubelet, **no self-healing**.
- **Controller-managed deployment:**
  - Use controllers (e.g., Deployment) for Pods.
  - Provides self-healing, scaling, and rolling updates.

Multi-Container Pods: Separation of Concerns

- **Design principle:** Each container should have a single, well-defined responsibility.
- **Benefits:**
  - Smaller, simpler containers
  - Easier troubleshooting and re-use
- **When to couple containers:**
  - Tightly coupled functions that need to share data (e.g., via a shared volume).

Example: Multi-Container Pod Pattern

- **Scenario:**
  - One container pulls content from a repository ("sync" container).
  - Another container serves the content as a web page ("web" container).
- **Shared volume:** Both containers access the same data via a shared volume.

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

Sidecar Pattern: - Main app container + sidecar container for secondary tasks

Common in service mesh architectures (e.g., traffic encryption, metrics)

Adapter Pattern: - Helper container transforms output for external systems

Example: Converts NGINX logs for Prometheus

Ambassador Pattern: - Helper container brokers connectivity to external systems

Main app outputs to Ambassador, which handles external communication

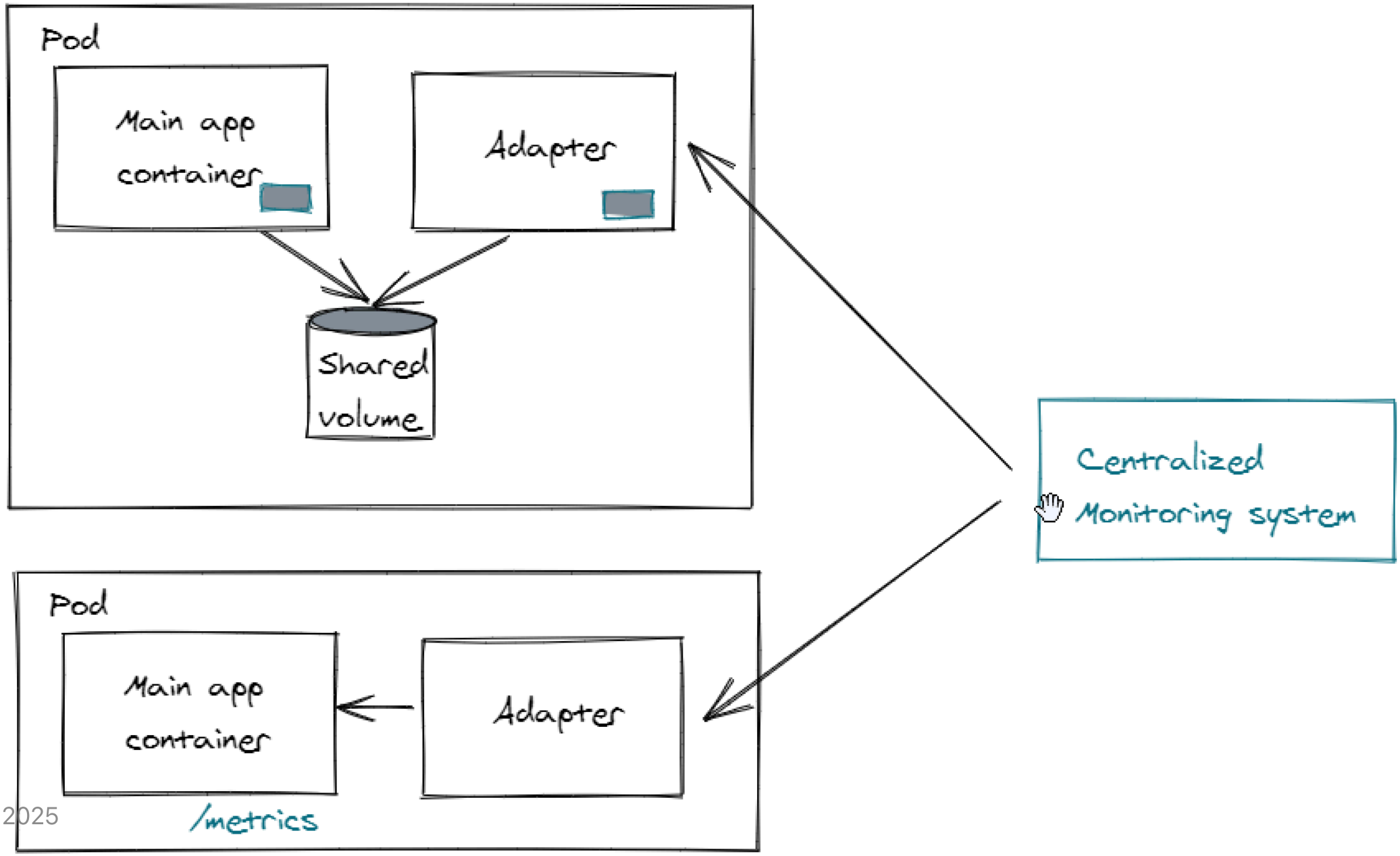
Init Pattern: - Special init container runs once before main app starts

Prepares environment (e.g., sets permissions, clones repos)

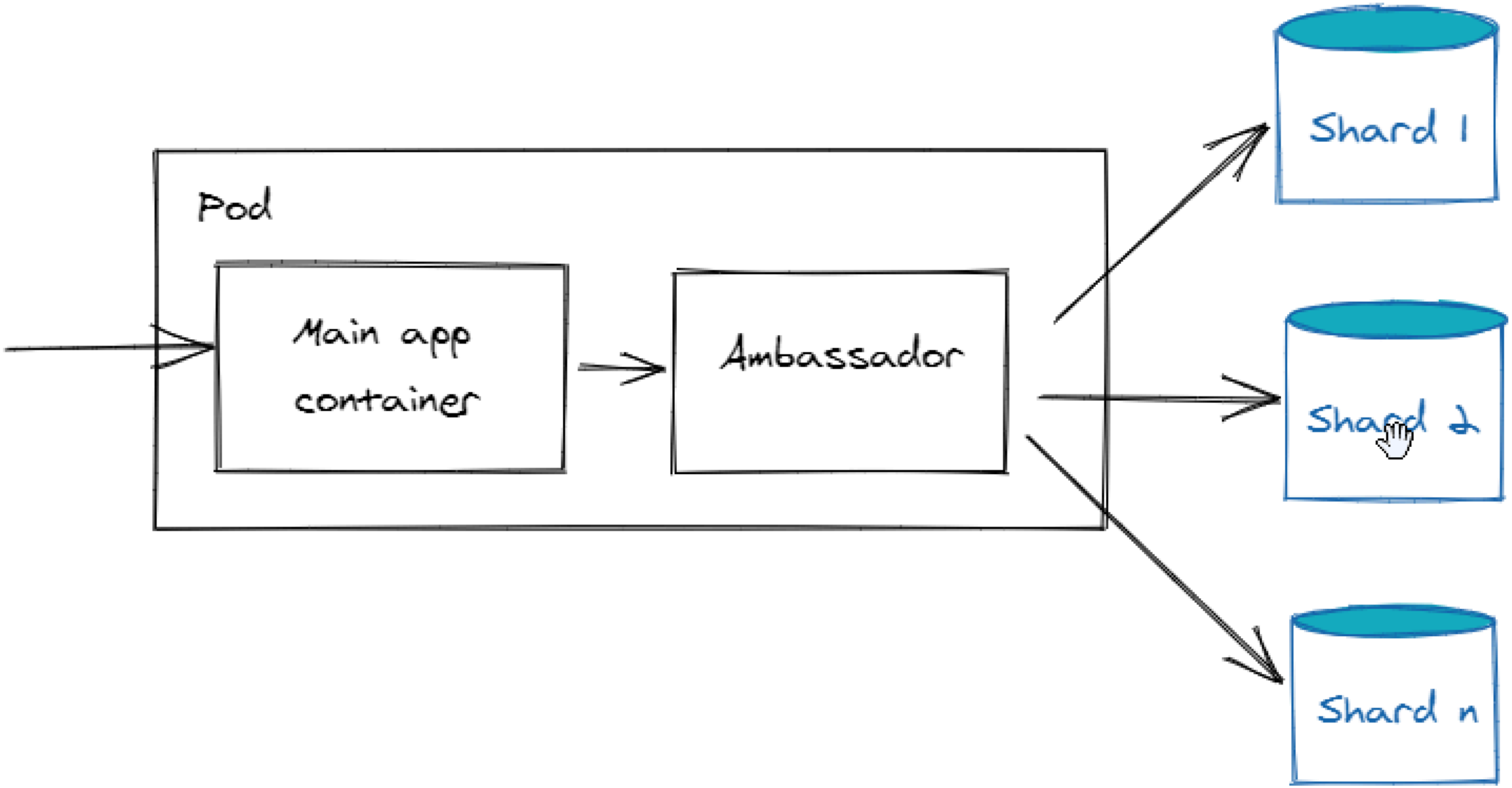
Diagram: Multi-Container Pod Patterns

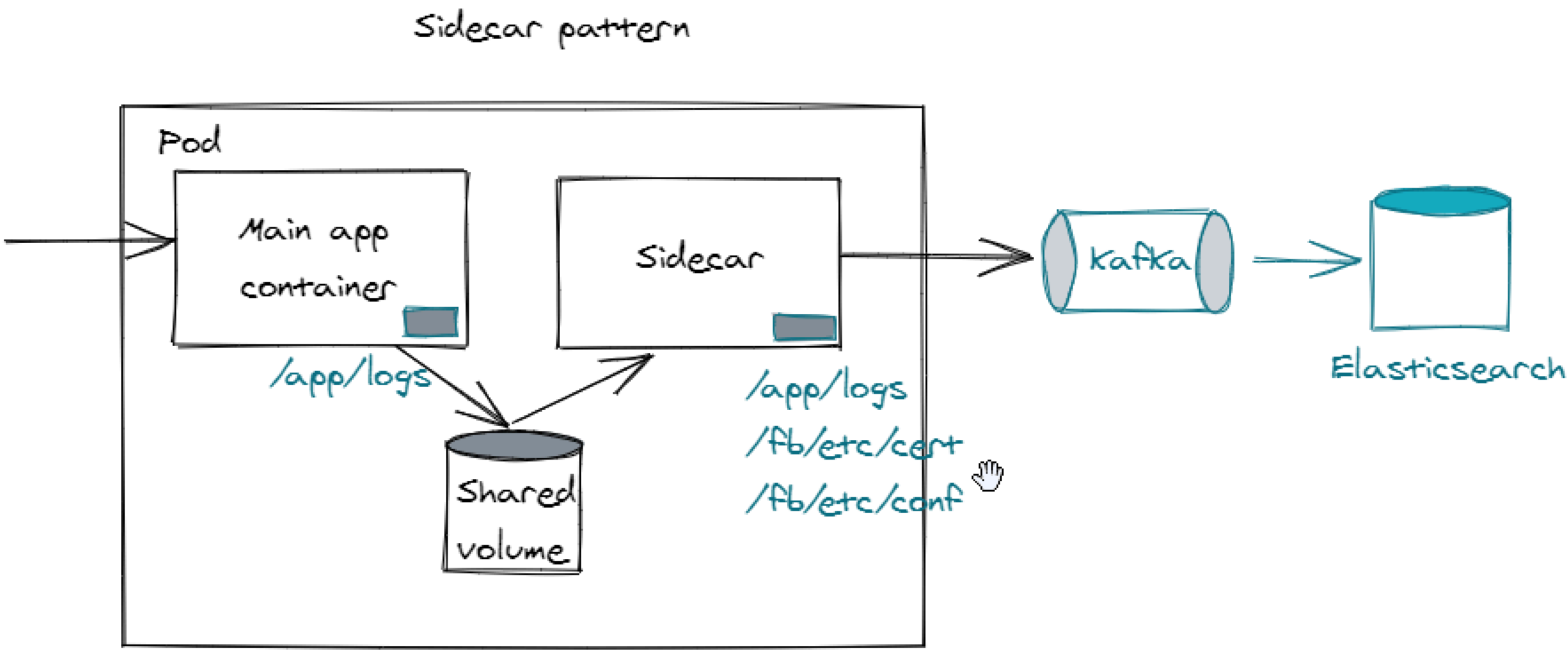
```
+-------------------------------+
|             Pod               |
|  +-------------------------+  |
|  |   Init Container        |  |  <-- Runs first, prepares environment
|  +-------------------------+  |
|  |   Main App Container    |  |  <-- Core application logic
|  +-------------------------+  |
|  |   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

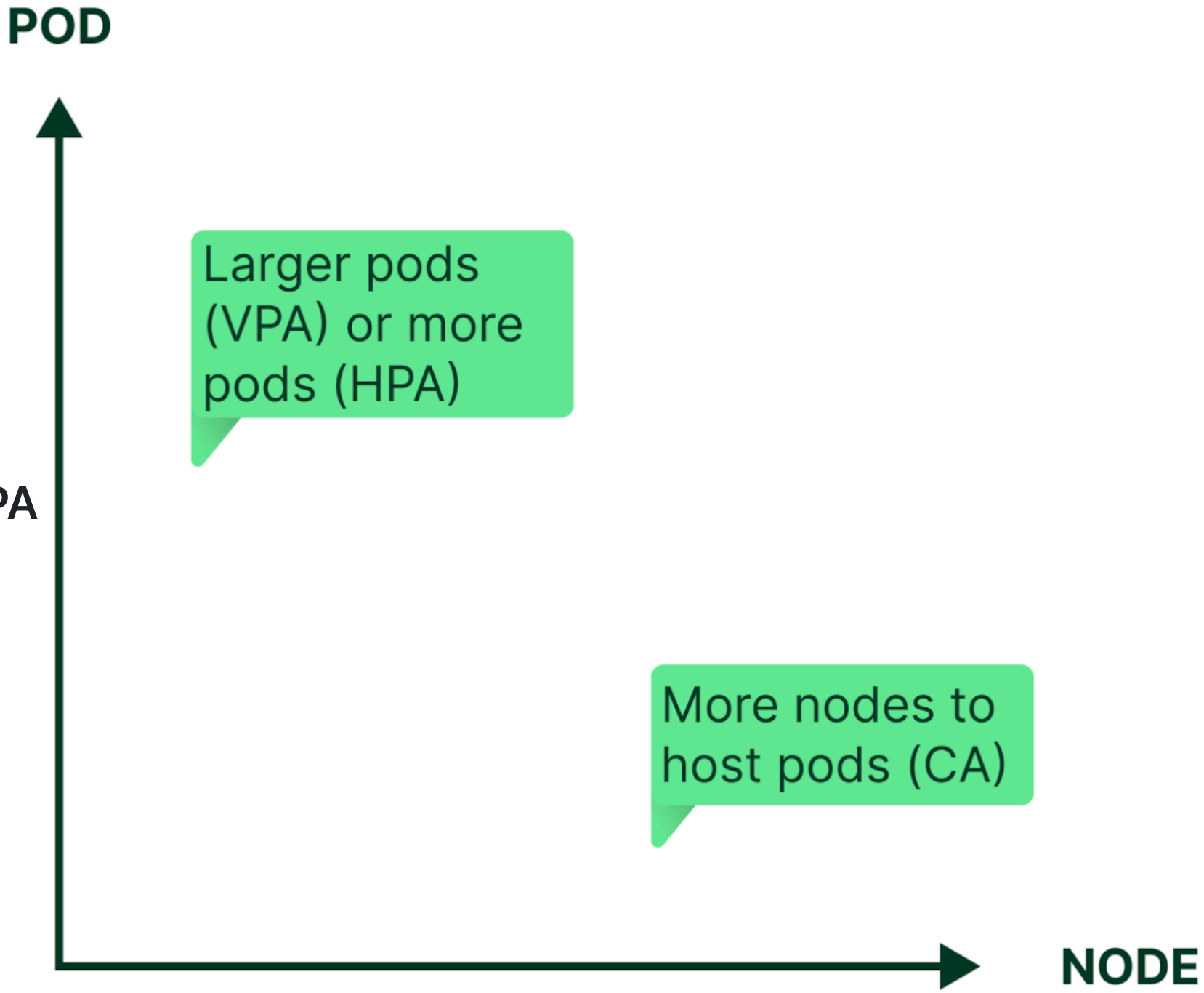
Пример манифеста HPA:

```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

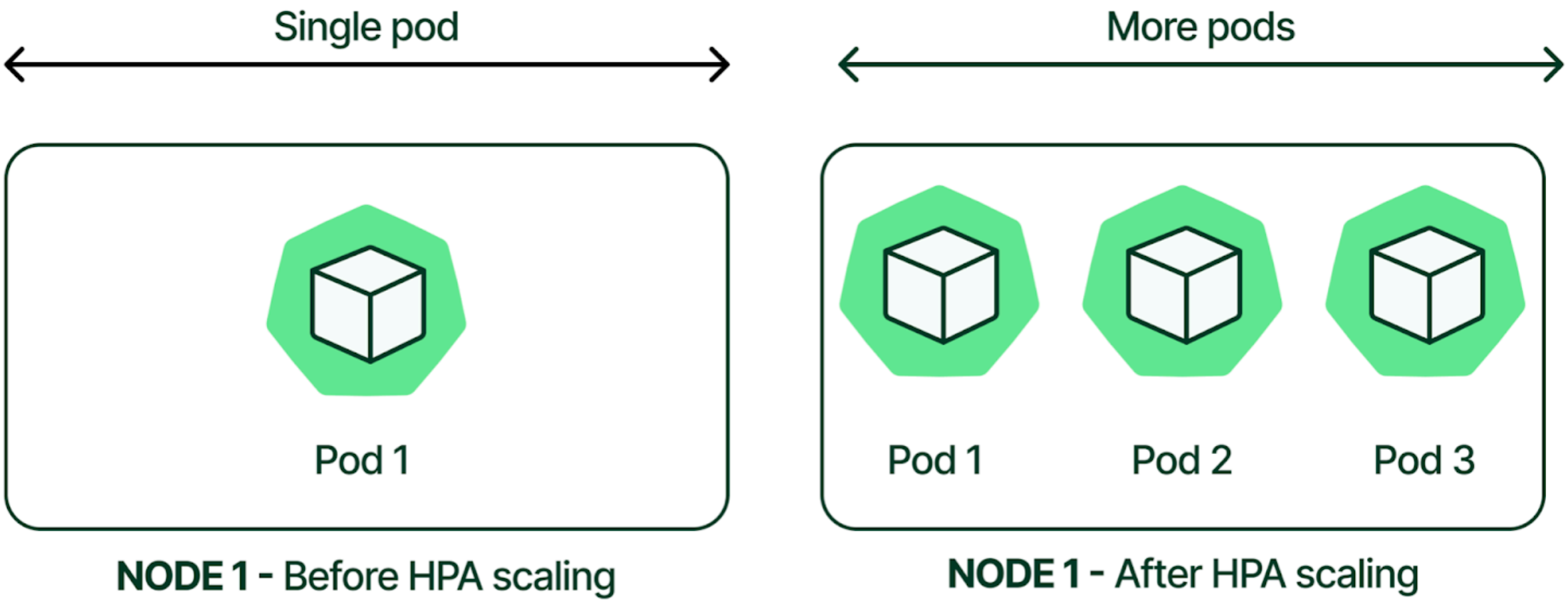
- **Horizontal Pod Autoscaler** (HPA):adjusts the number of replicas of an application.
- **Cluster Autoscaler:adjusts** the number of nodes of a cluster.
- **Vertical Pod Autoscaler** (VPA):adjusts the resource requests and limits of a container.

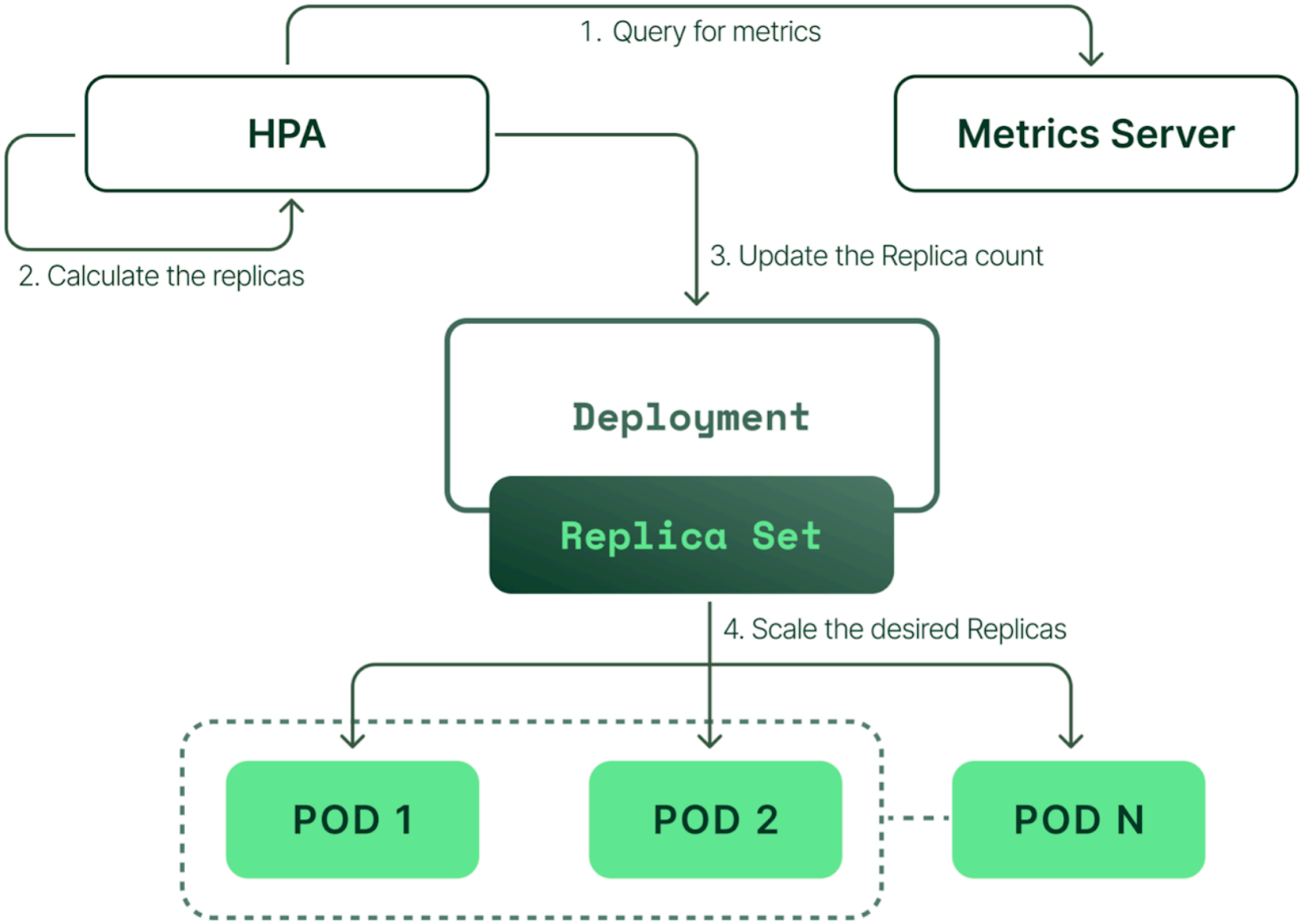
VPA vs HPA


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

Waste
Performance

HPA Autoscaling

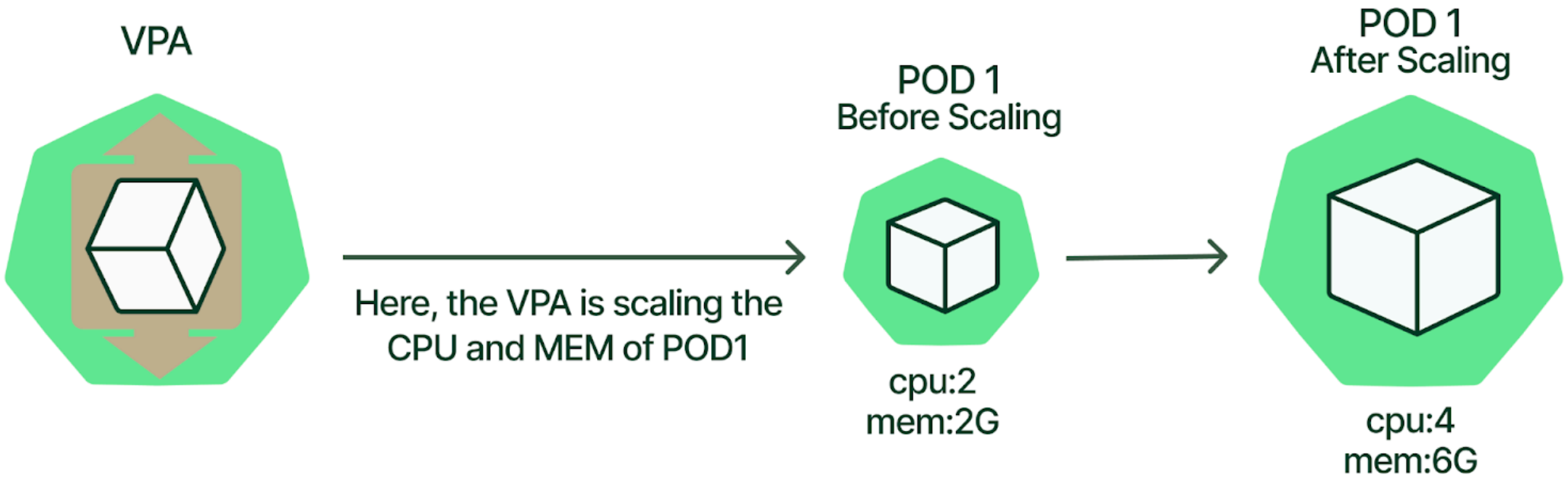



1. Query for metrics
HPA
Metrics Server
2. Calculate the replicas
3. Update the Replica count
Deployment
Replica Set
4. Scale the desired Replicas
POD 1
POD 2
POD N

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
```

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

- **Vertical Pod Autoscaler (VPA)** manages two resource settings for each container:
  - **Requests:** Minimum guaranteed resources
  - **Limits:** Maximum allowed resources

Diagram: VPA Resource Configuration

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

Key Points

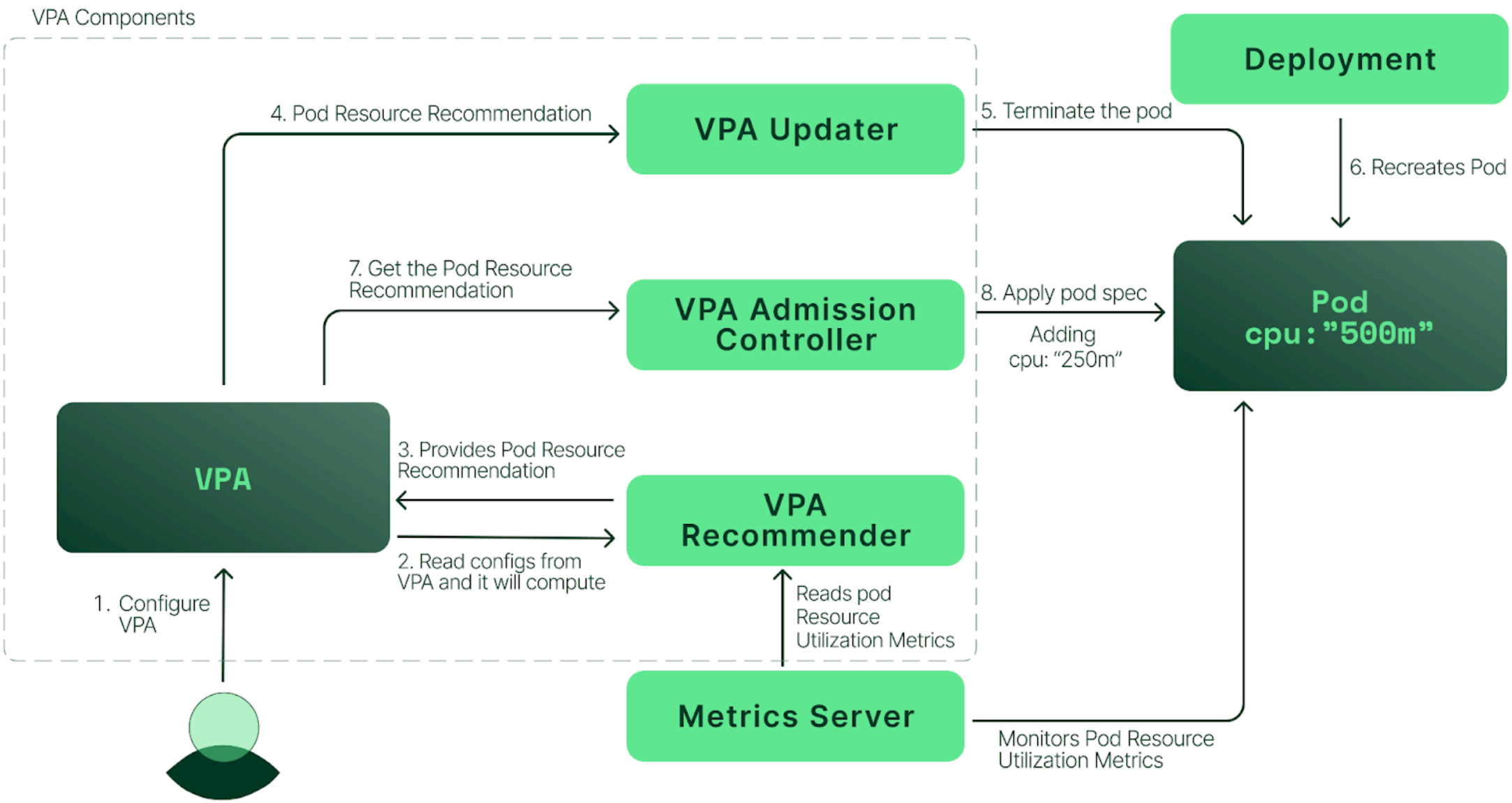
- **Requests**
  - Minimum resources Kubernetes guarantees for a container (e.g., 256Mi memory)
  - Scheduler uses requests to place Pods on nodes with enough available resources
- **Limits**
  - Maximum resources a container can use (e.g., 512Mi memory)
  - Prevents runaway containers from consuming excessive resources

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-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment ❶
    name:       frontend ❷
  updatePolicy:
    updateMode: "Off" ❸
```

❶ The type of workload object you want this VPA CR to manage.

❷ The name of the workload object you want this VPA CR to manage.

❸ Set the mode to `Off`.

VPA mode initial

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment ❶
    name:       frontend ❷
  updatePolicy:
    updateMode: "Initial" ❸
```

❶ The type of workload object you want this VPA CR to manage.

❷ The name of the workload object you want this VPA CR to manage.

❸ Set the mode to `Initial`. The VPA assigns resources when pods are created and does not change the resources during the lifetime of the pod.

VPA mode auto

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment ❶
    name:       frontend ❷
  updatePolicy:
    updateMode: "Auto" ❸
```

❶ The type of workload object you want this VPA CR to manage.

❷ The name of the workload object you want this VPA CR to manage.

❸ Set the mode to `Auto` or `Recreate`:

- `Auto`. The VPA assigns resource requests on pod creation and updates the existing pods by terminating them when the requested resources differ significantly from the new recommendation.
- `Recreate`. The VPA assigns resource requests on pod creation and updates the existing pods by terminating them when the requested resources differ significantly from the new recommendation. This mode should be used rarely, only if you need to ensure that the pods are restarted whenever the resource request changes.

Limitations of Kubernetes VPA

- **Do not use VPA and HPA together**
  - Avoid combining Vertical Pod Autoscaler (VPA) and Horizontal Pod Autoscaler (HPA) if both scale on the same metrics (CPU/Memory).
- **Resource recommendation issues**
  - VPA may recommend more resources than are available in the cluster.
  - This can prevent Pods from being scheduled.
- **Minimum replicas for reliability**
  - Always set at least **two replicas** for your Pods to avoid service disruptions.

Diagram: VPA Limitation Scenarios

```
+-------------------------------+
|         VPA Limitations       |
|-------------------------------|
| [X] VPA + HPA (CPU/MEM)       |
|     |                         |
|     +--> Conflicting scaling  |
|                               |
| [X] VPA recommends > cluster  |
|     |                         |
|     +--> Pod unschedulable    |
|                               |
| [!] Single replica            |
|     |                         |
|     +--> Service disruption   |
+-------------------------------+
```

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)
- **Pod spec is immutable**—VPA must recreate Pods to update resources
- **VPA does not consider network or I/O metrics**
- **Limited historical data**—VPA uses only the last 8 days

Diagram: VPA Limitation Scenarios

```
+-------------------------------+
|         VPA Limitations       |
|-------------------------------|
| [X] VPA + HPA (CPU/MEM)       |
|     |                         |
|     +--> Conflicting scaling  |
|                               |
| [X] VPA recommends > cluster  |
|     |                         |
|     +--> Pod unschedulable    |
|                               |
| [!] No StatefulSet support    |
|                               |
| [!] Pod spec immutable        |
|     |                         |
|     +--> Pod recreated        |
|                               |
| [!] Ignores network/I/O       |
|                               |
| [!] 8 days history only       |
+-------------------------------+
```

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

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

Rolling Update

Обновление

```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
```

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.

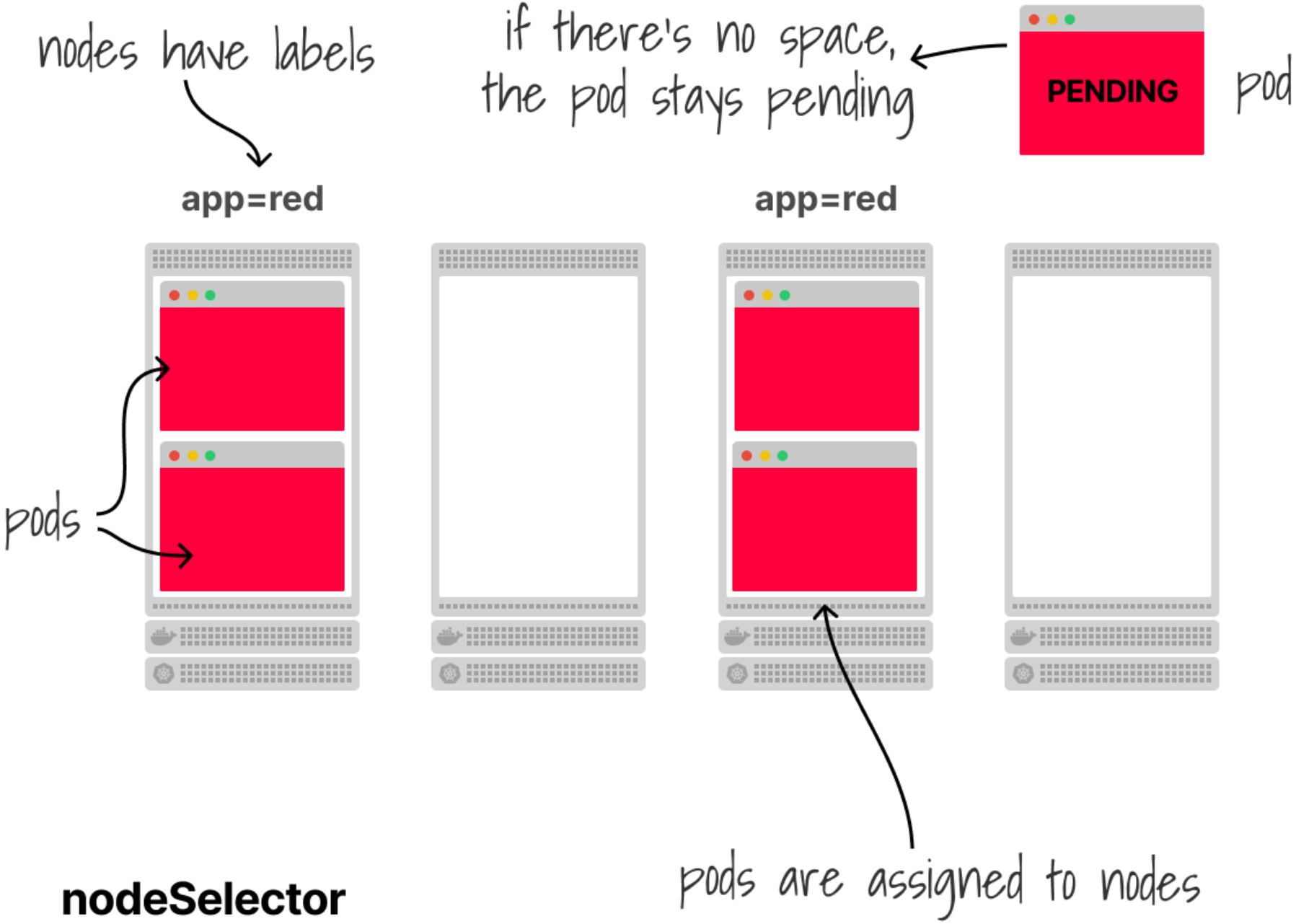
Практика

LAb Advanced Deployment Strategies with Rolling Updates and Rollbacks

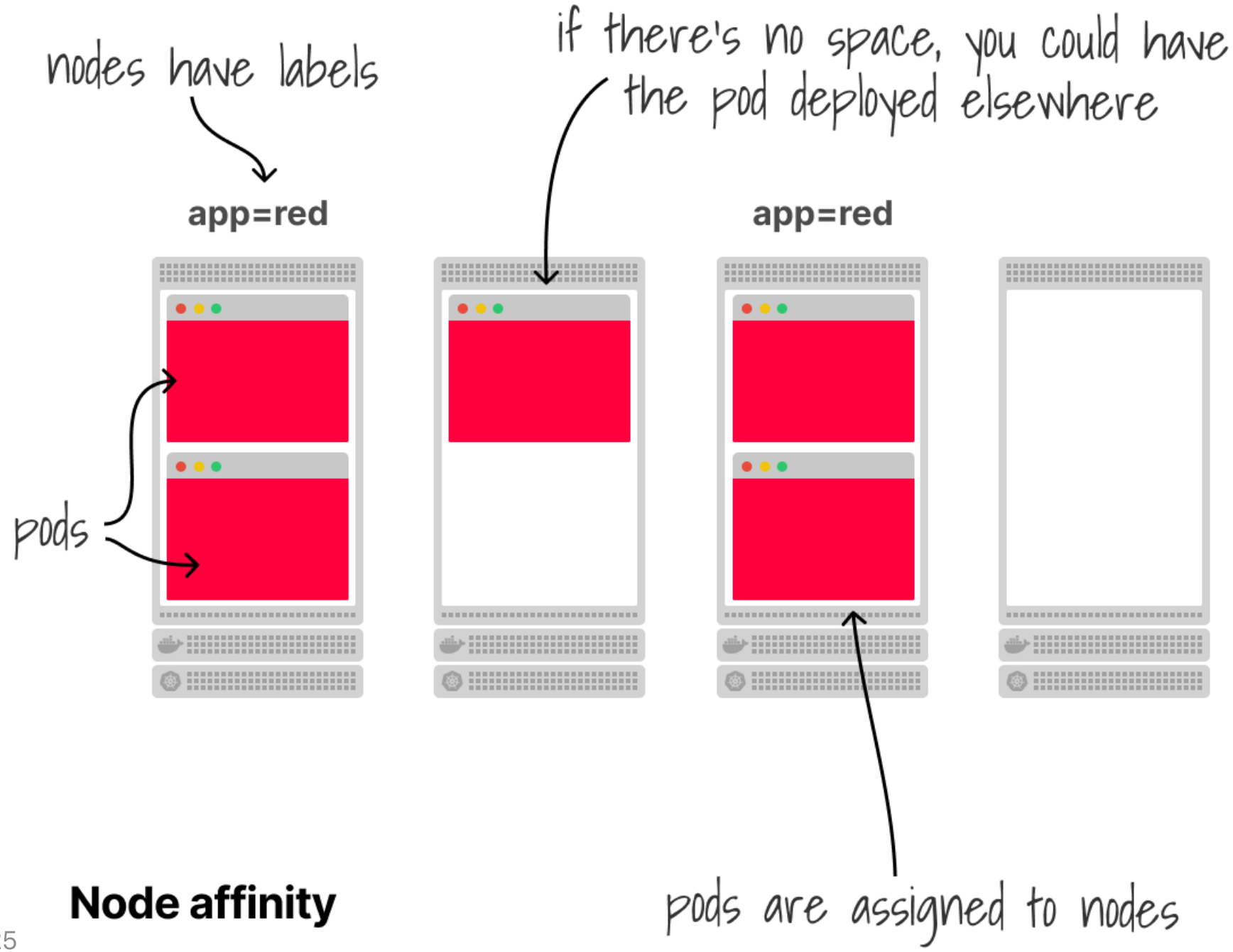
https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_intermediate_labs.md#lab-1-advanced-deployment-strategies-with-rolling-updates-and-rollbacks

INFLUENCING THE SCHEDULER

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



nodeSelector

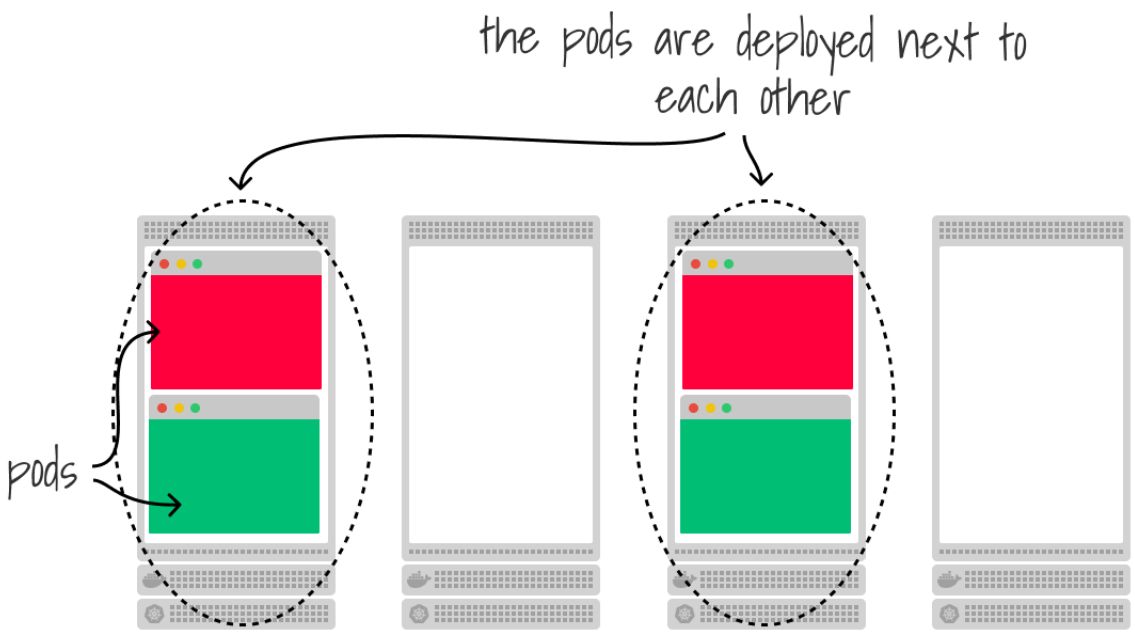


Node affinity

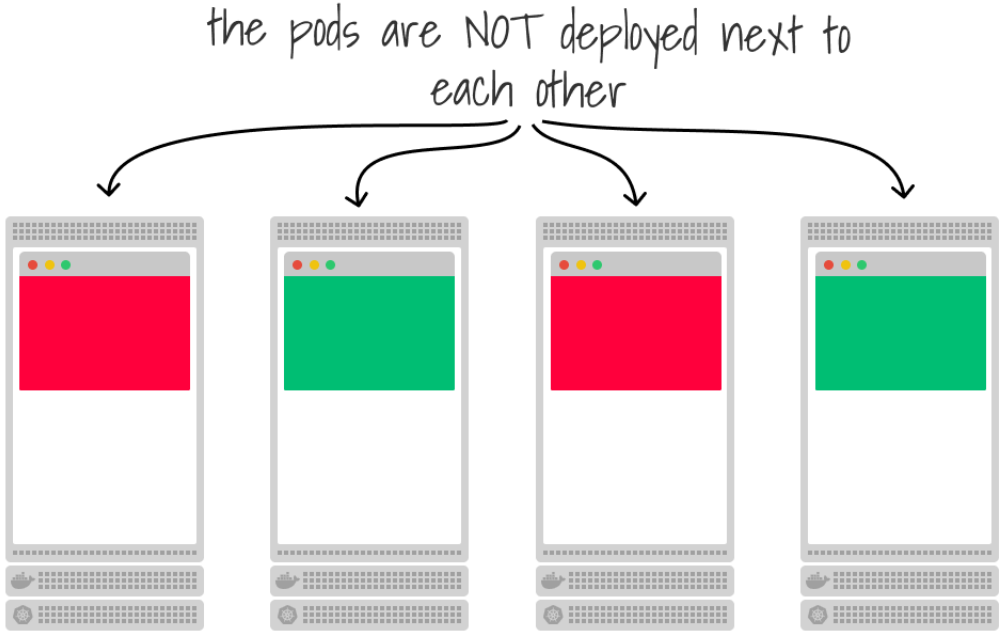
Example pod configuration file with a node affinity preferred rule

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity: (1)
      preferredDuringSchedulingIgnoredDuringExecution: (2)
      - weight: 1 (3)
        preference:
          matchExpressions:
          - key: e2e-az-EastWest (4)
            operator: In (5)
            values:
            - e2e-az-East (4)
            - e2e-az-West (4)
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod
```

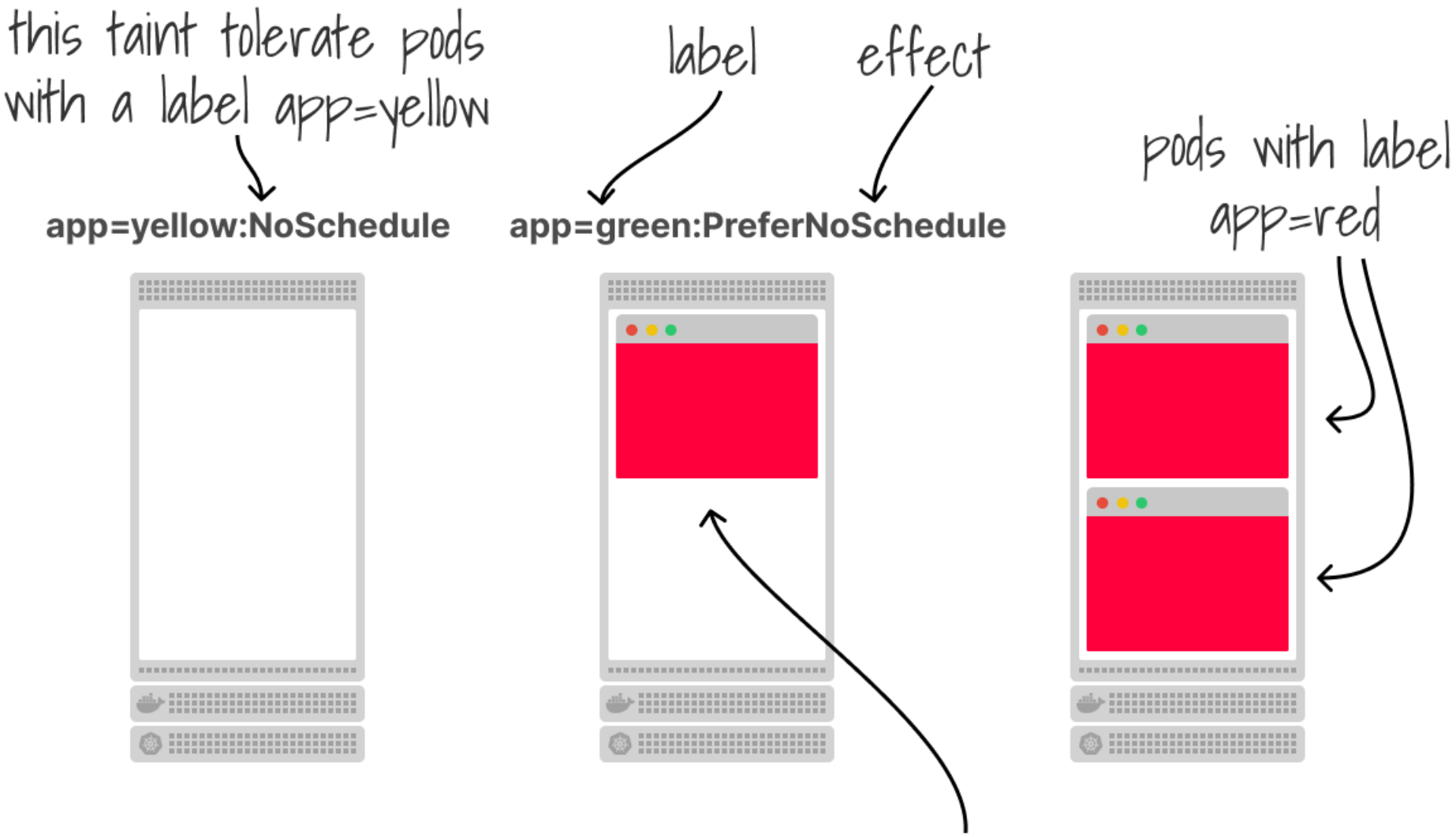
1. The stanza to configure node affinity.
2. Defines a preferred rule.
3. Specifies a weight for a preferred rule. The node with highest weight is preferred.
4. The key/value pair (label) that must be matched to apply the rule.
5. The operator represents the relationship between the label on the node and the set of values in the `matchExpression` parameters in the `Pod` spec. This value can be `In`, `NotIn`, `Exists`, or `DoesNotExist`, `Lt`, or `Gt`.



Node affinity



Node anti-affinity



Taints and tolerations

A taint can produce three possible effects:

- **NoSchedule**
  The Kubernetes scheduler will only allow scheduling pods that have tolerations for the tainted nodes.
- **PreferNoSchedule**
  The Kubernetes scheduler will try to avoid scheduling pods that don’t have tolerations for the tainted nodes.
- **NoExecute**
  Kubernetes will evict the running pods from the nodes if the pods don’t have tolerations for the tainted nodes.

built-in taints:

- **node.kubernetes.io/not-ready**
  Node is not ready. This corresponds to the NodeCondition Ready attribute being "False".
- **node.kubernetes.io/unreachable**
  Node is unreachable from the node controller. This corresponds to NodeCondition Ready being "Unknown".
- **node.kubernetes.io/memory-pressure**
  Node has memory pressure.
- **node.kubernetes.io/disk-pressure**
  Node has disk pressure. In case of High disk utilization on nodes, it can cause slowness for application so its better to relocate pods.

built-in taints:

- **node.kubernetes.io/pid-pressure**
  Node has PID pressure. Process ID is a limited resource and its saturation can cause down time for applications, so better to relocate pods to somewhere else.
- **node.kubernetes.io/network-unavailable**
  Node's network is unavailable. As explained above.
- **node.kubernetes.io/unschedulable**
  Node is unschedulable. Any other reason that will make the node inappropriate for hosting pods, for example if the cluster is being scaled down and the node is being removed.

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

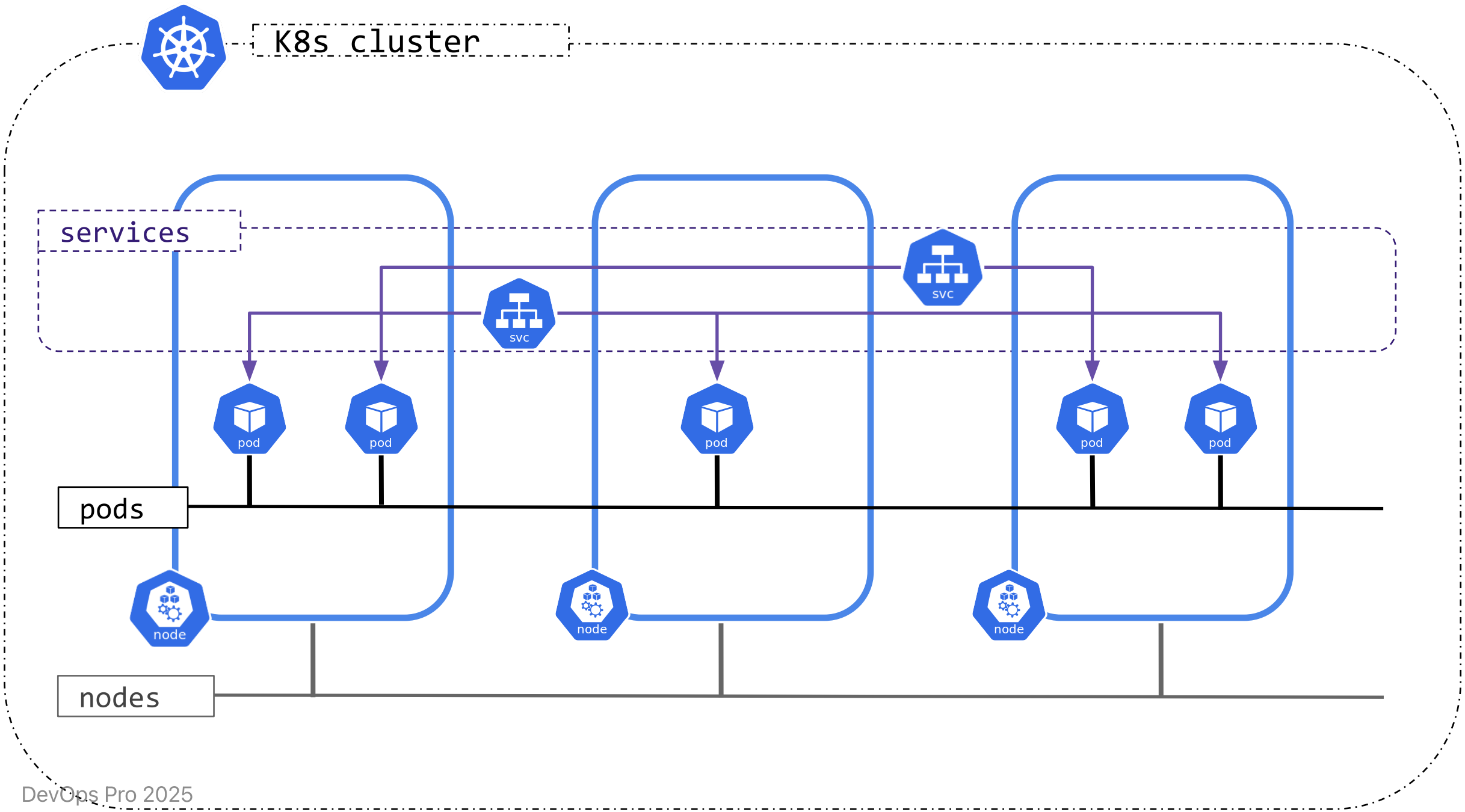
LAb Kubernetes Scheduling. Taints and Tolerations

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

- **Kubernetes clusters require non-overlapping IP ranges** for:
  - Pods
  - Services
  - Nodes
- **IP ranges are configured in different components:**
  - **Network plugin:** Assigns Pod IP addresses
  - **kube-apiserver:** Assigns Service IP addresses
  - **kubelet/cloud-controller-manager:** Assigns Node IP addresses


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

Calculating the Pod Network Range in Kubernetes

When setting up a Kubernetes cluster, you must define a **Pod network CIDR**—the IP address range from which all Pods will receive their IPs. Choosing the correct range is crucial for reliable networking and future scalability.

Step 1: Estimate the Number of Pods

- **Best practice:** Plan for future growth.
- **Default max Pods per node:** 110 (can be changed with kubelet's `--max-pods` flag).

Example:

If you plan for up to 50 nodes, each running up to 100 Pods:

```
Total Pods = 50 nodes × 100 Pods/node = 5,000 Pods
```

Step 2: Choose a Non-Overlapping Private IP Range

- Use a private IP range (RFC1918) that does **not overlap** with:
  - Node IPs
  - Service CIDR
  - Any existing network in your data center or cloud 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

- Each Pod gets a unique IP.
- **CIDR size examples:**
  - `/16` = 65,536 IPs
  - `/19` = 8,192 IPs
  - `/20` = 4,096 IPs

For 5,000 Pods:

- `/19` gives 8,192 IPs (enough for 5,000 Pods + future growth).

Step 4: Assign Per-Node Pod CIDRs

- The Pod network is divided into smaller subnets, one per node.
- **Example:**
  - Pod CIDR: `10.244.0.0/16`
  - Each node gets a `/24` (256 IPs): `10.244.1.0/24`, `10.244.2.0/24`, etc.

Best practice:

- Ensure each node's Pod CIDR is large enough for the max Pods per node.

Step 5: Configure the Cluster

- **kubeadm:**
  Set `--pod-network-cidr=10.244.0.0/16` when initializing:
  ```
  kubeadm init --pod-network-cidr=10.244.0.0/16
  ```
- **Cloud providers:**
  Configure the Pod CIDR in your cloud network and CNI plugin.

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

Key Points:

- Always use a private, non-overlapping CIDR for Pods.
- Plan for future growth—choose a range larger than your current needs.
- Ensure Pod, Service, and Node IP ranges do not overlap.
- Configure your CNI plugin and cluster accordingly.

References

- Kubernetes Networking Concepts
- Pod CIDR Planning

Container Networking

- **"There's no such thing as a Container"**
  Containers are just isolated Linux processes.
- **Linux namespaces** provide isolation:
  - PID, network, mount, UTS, User ID, etc.
- **Network namespace** gives each process (and its children) a virtual IP stack.
- Each network namespace has its own:
  - Interfaces (e.g., `loopback`, `eth0`)
  - Routing tables
  - Other network resources

Pod Architecture

- **Pods group one or more containers together** as a single deployable unit.
- **All containers in a Pod share the same network namespace:**
  - Same IP address
  - Same port space
- **Pods are atomic in Kubernetes:**
  - Scheduled, scaled, and terminated as a unit
- **You can't split a Pod across multiple nodes**—it always runs on a single node.

Pod Networking

- **Distributed app components must communicate** across the cluster.
- **Pods need a network** to enable this communication.
- **Pod location doesn't matter:**
  - Pods can be on the same or different hosts.
  - What matters is that they can reach each other.
- **Each Pod gets its own network interface and unique IP address.**
- **Containers within the same Pod communicate via `localhost` (loopback interface).**

CNI – Container Networking Interface

- **CNI is a specification for container networking**
  - Enables Kubernetes to integrate with Software Defined Networking (SDN) solutions.
- **CNI plugins handle:**
  - Pod network wiring
  - Assigning Pod IP addresses
  - Managing network policies
- **SDN** allows network management via software controllers, not manual hardware configuration.

How CNI Works in Kubernetes

- **CNI (Container Networking Interface) has two main components:**
  - **CNI Plugin:** A binary that implements the CNI specification.
  - **CNI Configuration File:** A JSON file specifying which plugin to use and its settings.
- **Pod creation triggers networking setup:**
  - The kubelet on the node calls the configured CNI plugin.

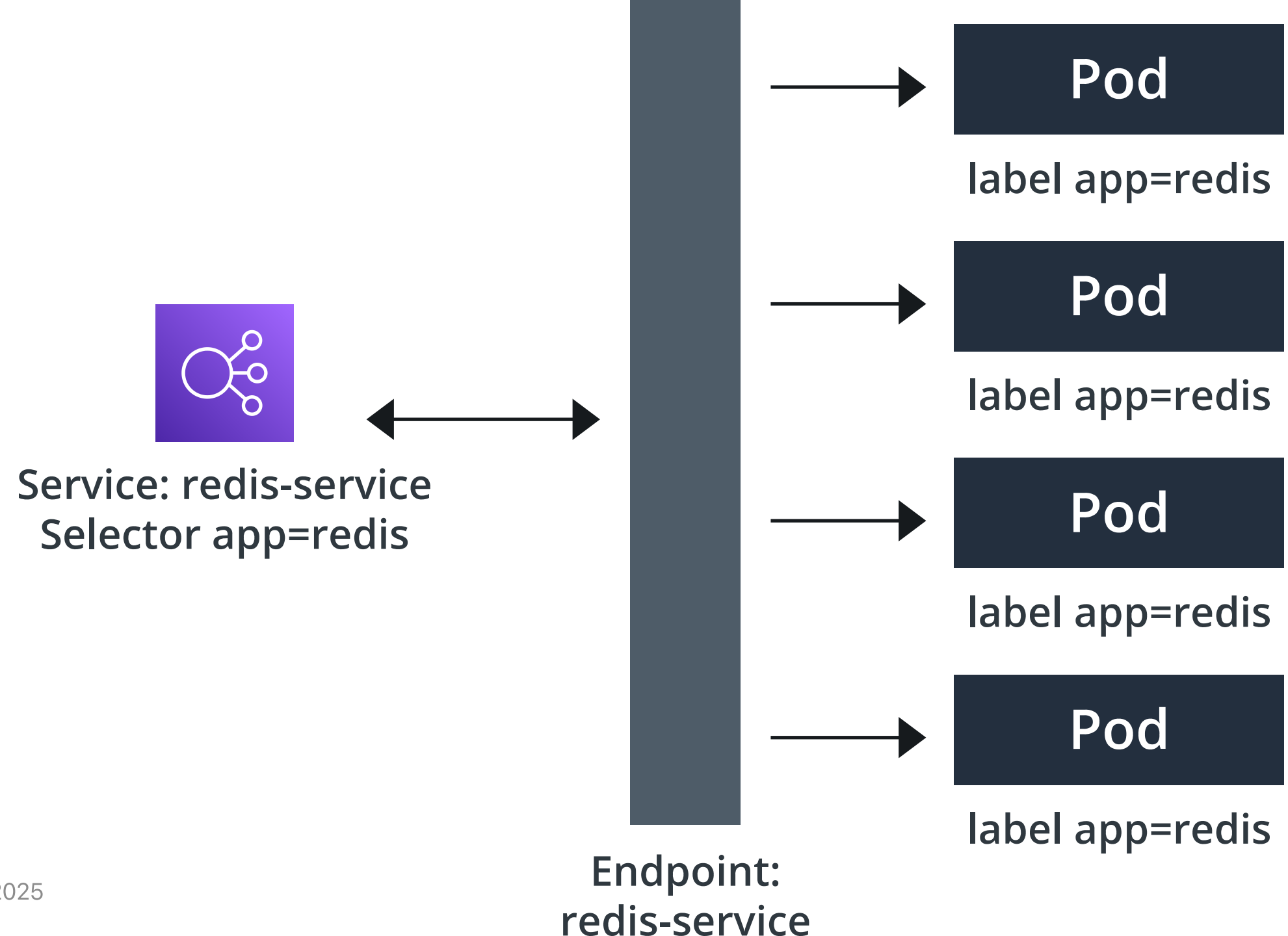
Example:

When a new Pod is scheduled, the kubelet reads `/etc/cni/net.d/10-calico.conflist` to determine which plugin (e.g., Calico) to use.

How CNI Works in Kubernetes

- The CNI plugin:
  - Creates a network namespace for the Pod.
  - Sets up a virtual Ethernet pair (veth) and connects one end to the bridge.
  - Assigns an IP address to the Pod.
  - Updates routing tables and DNS as needed.
- **Popular CNI plugins:**
  Calico, Flannel, Cilium

What is a Service in Kubernetes?


Pod
label app=redis
Pod
label app=redis
Pod
label app=redis
Pod
label app=redis
Service: redis-service
Selector app=redis
Endpoint:
redis-service

```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

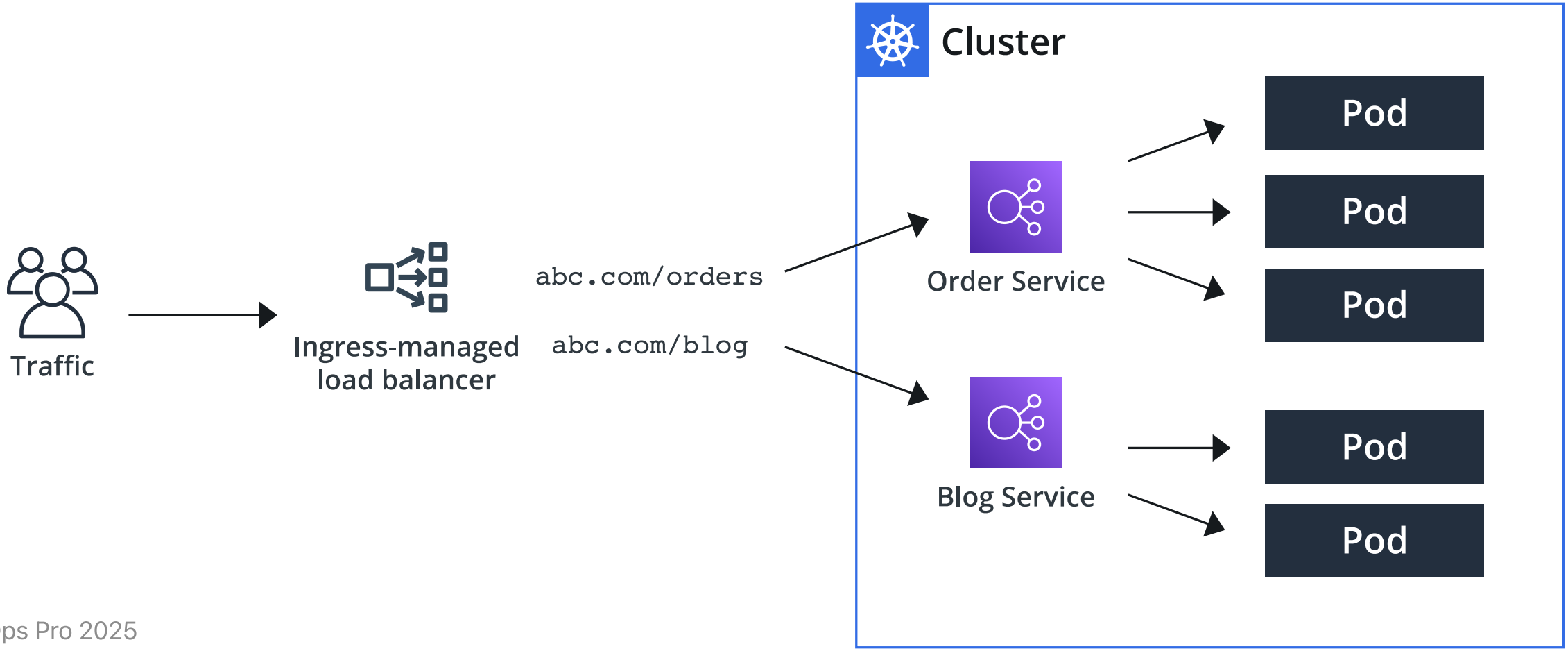
NodePort service yaml

```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** – The default value. The service is only accessible from within the Kubernetes cluster – you can’t make requests to your Pods from outside the cluster!
- **NodePort** – This makes the service accessible on a static port on each Node in the cluster `<NodeIP>:<NodePort>`.
- **LoadBalancer** – The service becomes accessible externally through a cloud provider's load balancer functionality. GCP, AWS, Azure, and OpenStack offer this functionality.

Ingress


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

Ingress code example

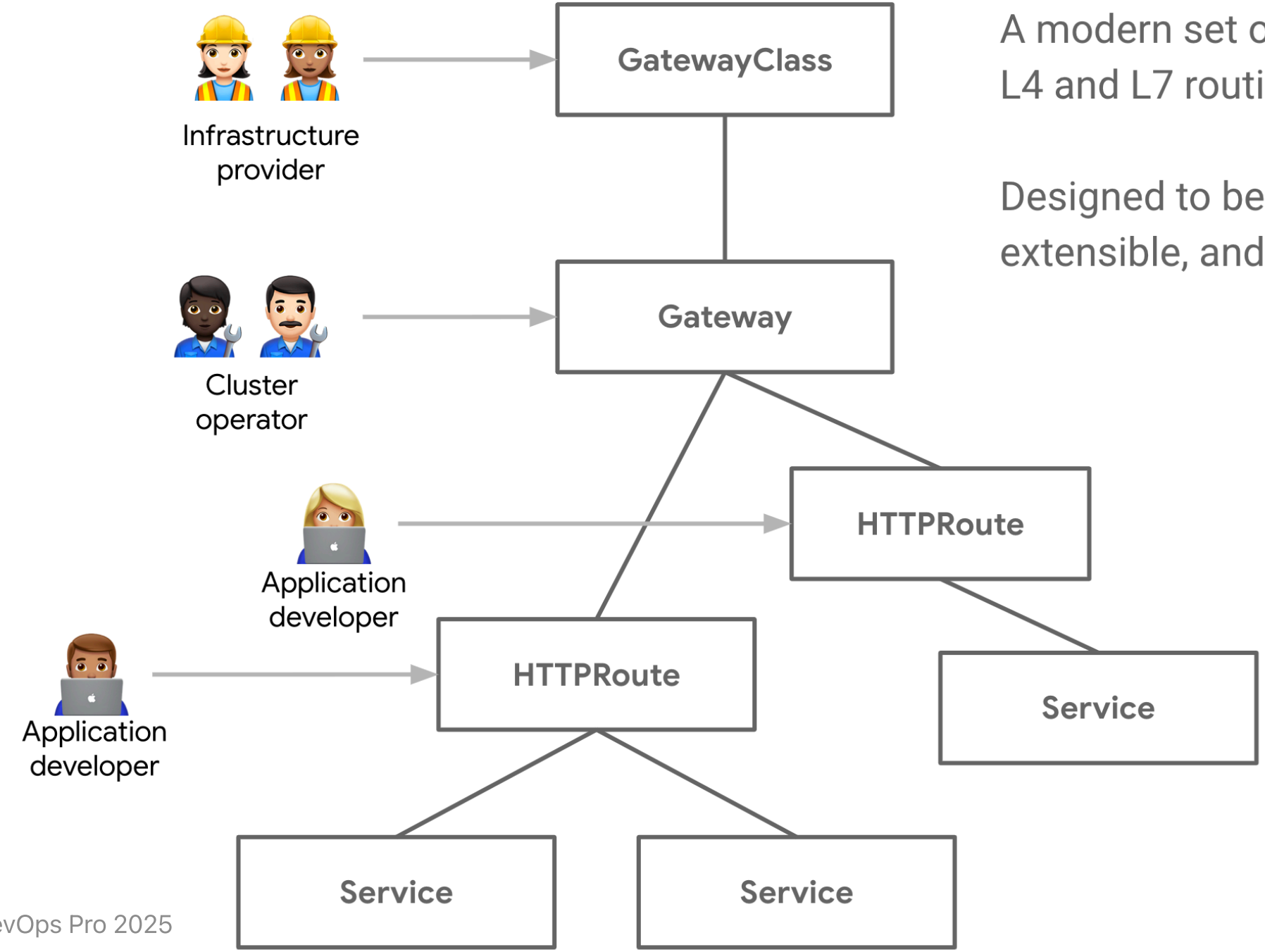
```yaml
kind: Ingress
  apiVersion: extensions/v1beta1
  metadata:
    name: "example-ingress"
    namespace: production
    annotations:
      kubernetes.io/ingress.class: alb
      alb.ingress.kubernetes.io/scheme: internet-facing
      alb.ingress.kubernetes.io/listen-ports: '[{"HTTPS":443}, {"HTTP":80}]'
  spec:
    rules:
      - host: abc.com
        http:
          paths:
            - path: /orders
              backend:
                serviceName: order-service
                servicePort: 8080
            - path: /blog
              backend:
                serviceName: blog-service
                servicePort: 8081
```

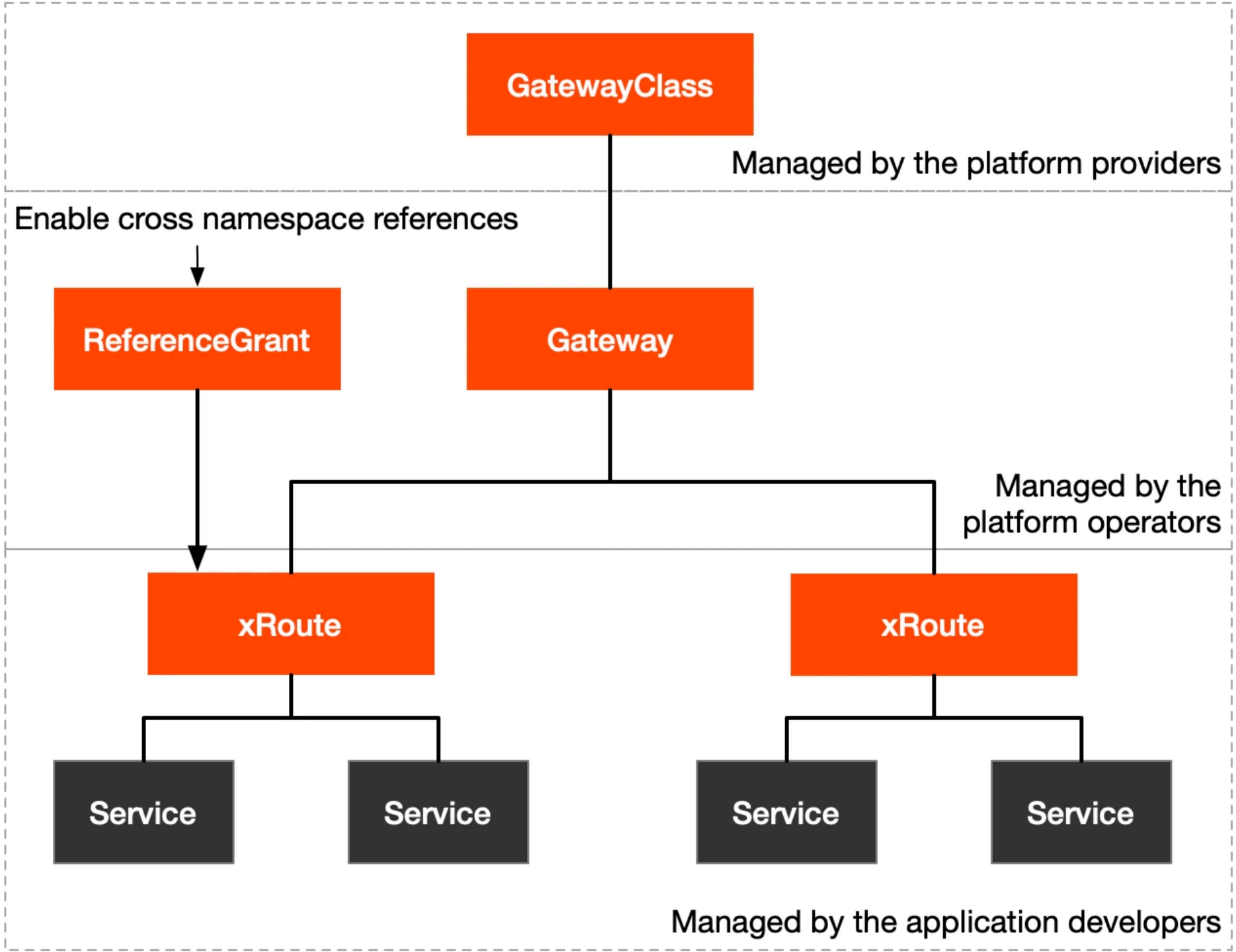
Best practices/tips for effective use of Service

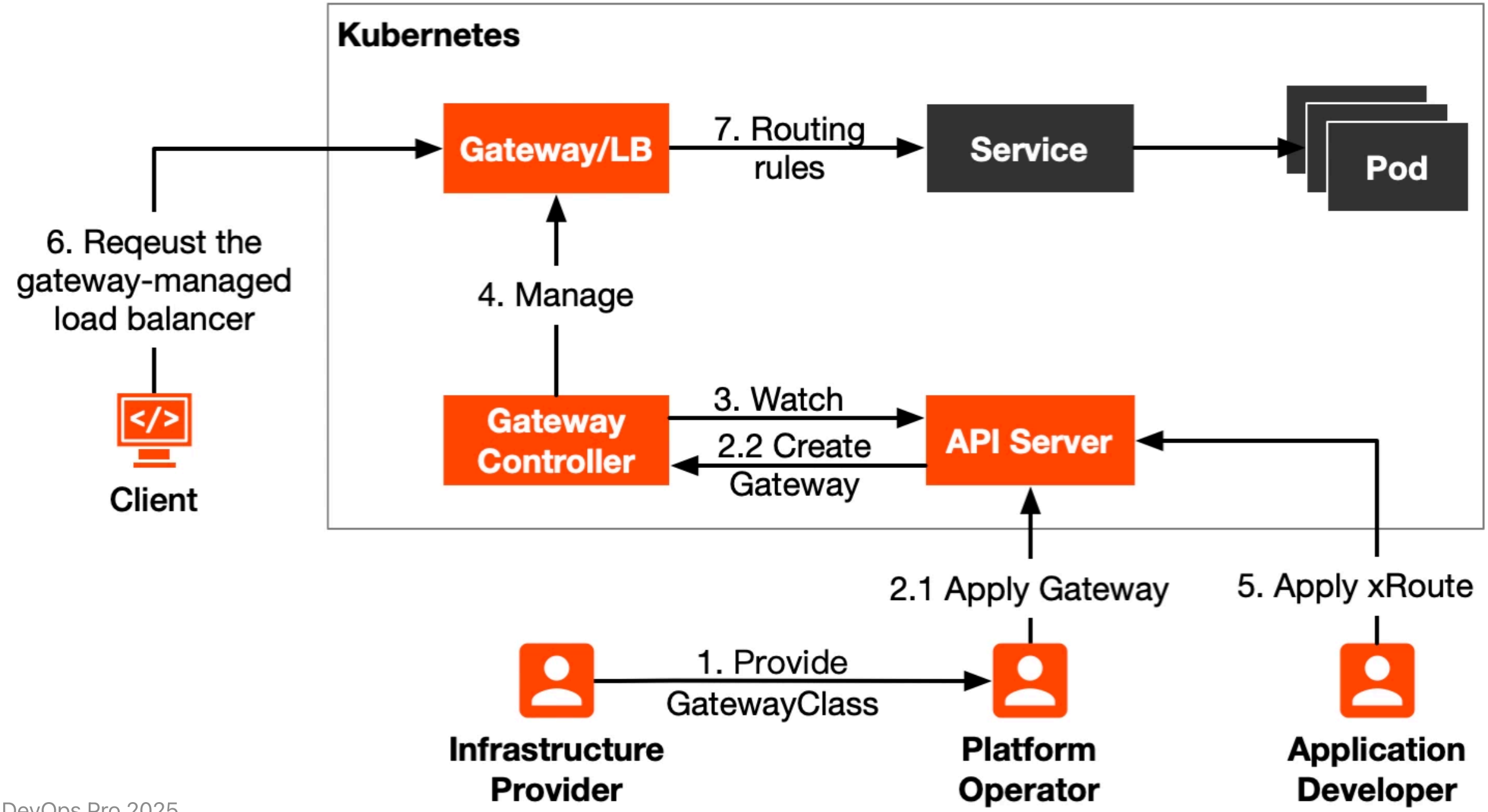
1. Always use DNS for connecting to other services within the cluster
2. Use Ingress whenever possible while exposing externally
3. Expose only standard ports like 80 or 443 to the external. We can have various Ingress rules to route the traffic to a specific service
4. Always enable SSL in external load balancers
5. Use Network Policies to manage access restrictions

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

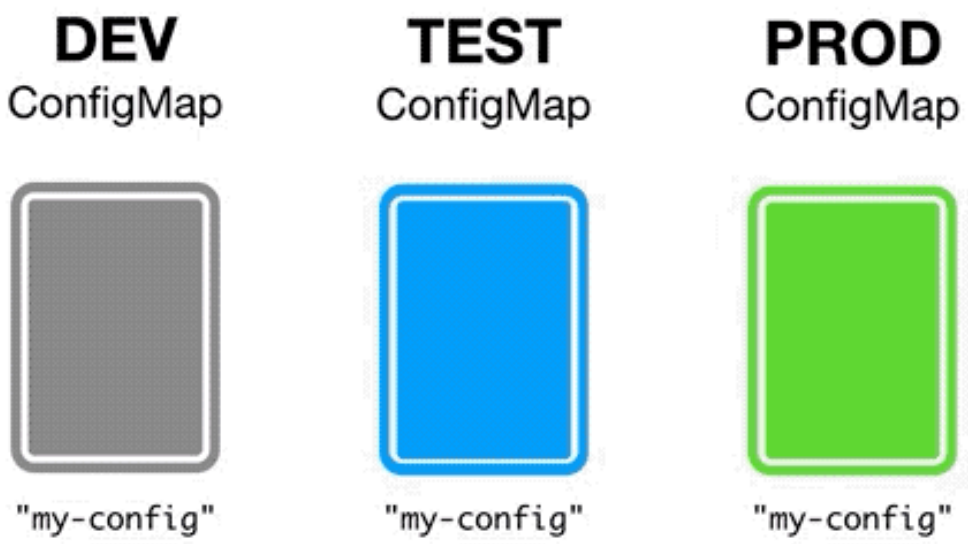

Kubernetes
Gateway/LB
7. Routing rules
Service
Pod
6. Reqeust the gateway-managed load balancer
4. Manage
Client
Gateway Controller
3. Watch
2.2 Create Gateway
API Server
2.1 Apply Gateway
5. Apply xRoute
1. Provide GatewayClass
Infrastructure Provider
Platform Operator
Application Developer

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
```

There are four different ways that you can use a ConfigMap to configure a container inside a Pod:

1. Inside a container command and args
2. Environment variables for a container
3. Add a file in read-only volume, for the application to read
4. Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

Secrets. Built-in Secret Types

Opaque Secrets – The default Secret type if omitted from a Secret configuration file.

Service account token Secrets – Used to store a token that identifies a service account.

Docker config Secrets – Stores the credentials for accessing a Docker registry for images.

Basic authentication Secret – Used for storing credentials needed for basic authentication.

SSH authentication secrets – Used for storing data used in SSH authentication.

TLS secrets – For storing a certificate and its associated key

Bootstrap token Secrets – Used for tokens used during the node bootstrap process.

Basic secret example

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

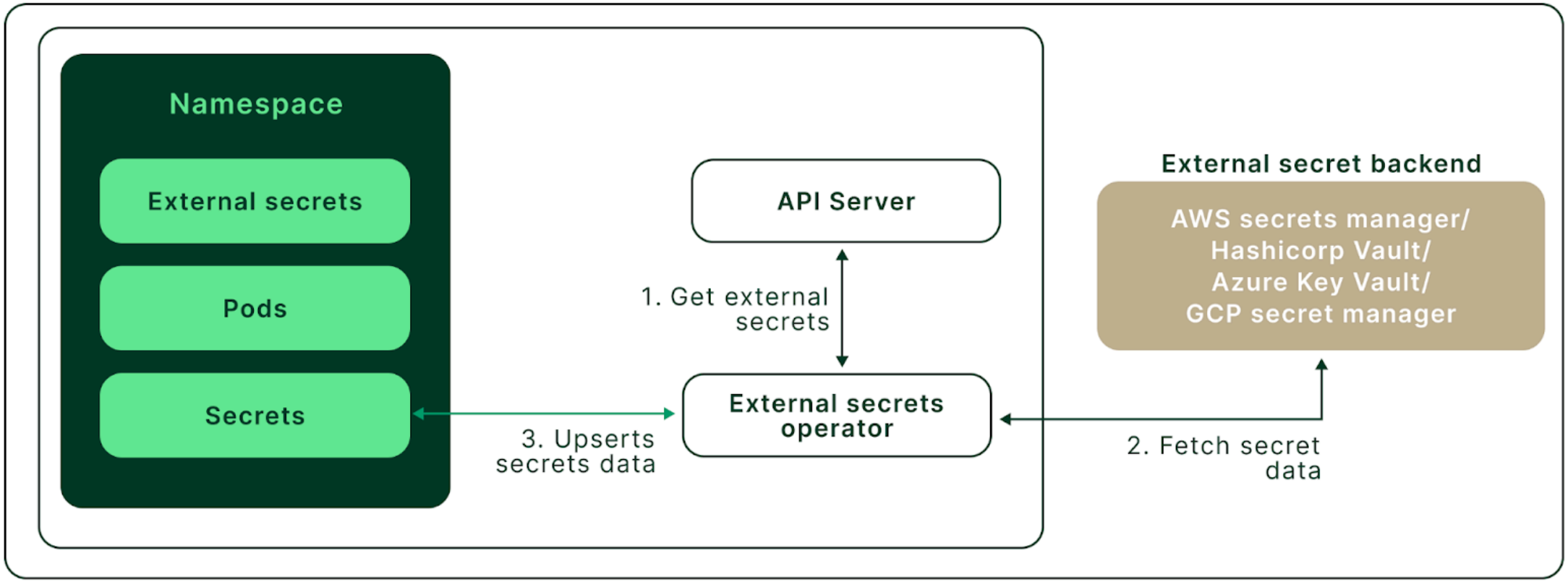
ssh-privatekey

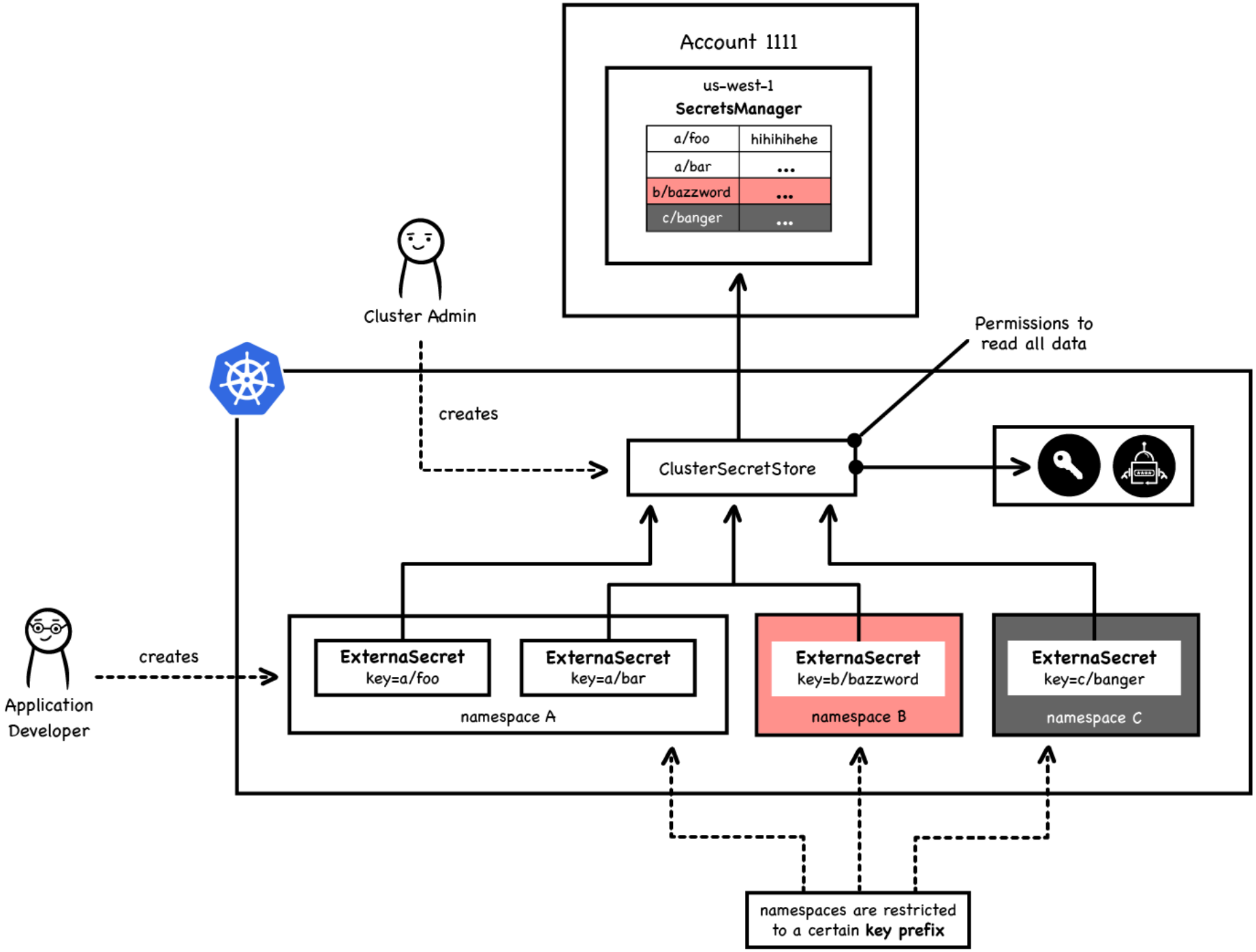
```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth
data:
  # the data is abbreviated in this example
  ssh-privatekey: |
     MIIEpQIBAAKCAQEAulqb/Y ...
```

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

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

- **Principle of Least Privilege**
  - Assign only the minimum permissions needed for users and service accounts.
- **Namespace-Level Permissions**
  - Use `Role` and `RoleBinding` for namespace-scoped access.
- **Avoid Wildcards**
  - Do not use wildcards (`*`) for resources or verbs.
- **Continuous Policy Updates**
  - Regularly review and update RBAC policies.
- **Limit Admin Access**
  - Avoid using cluster-admin for daily tasks; use scoped roles.

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

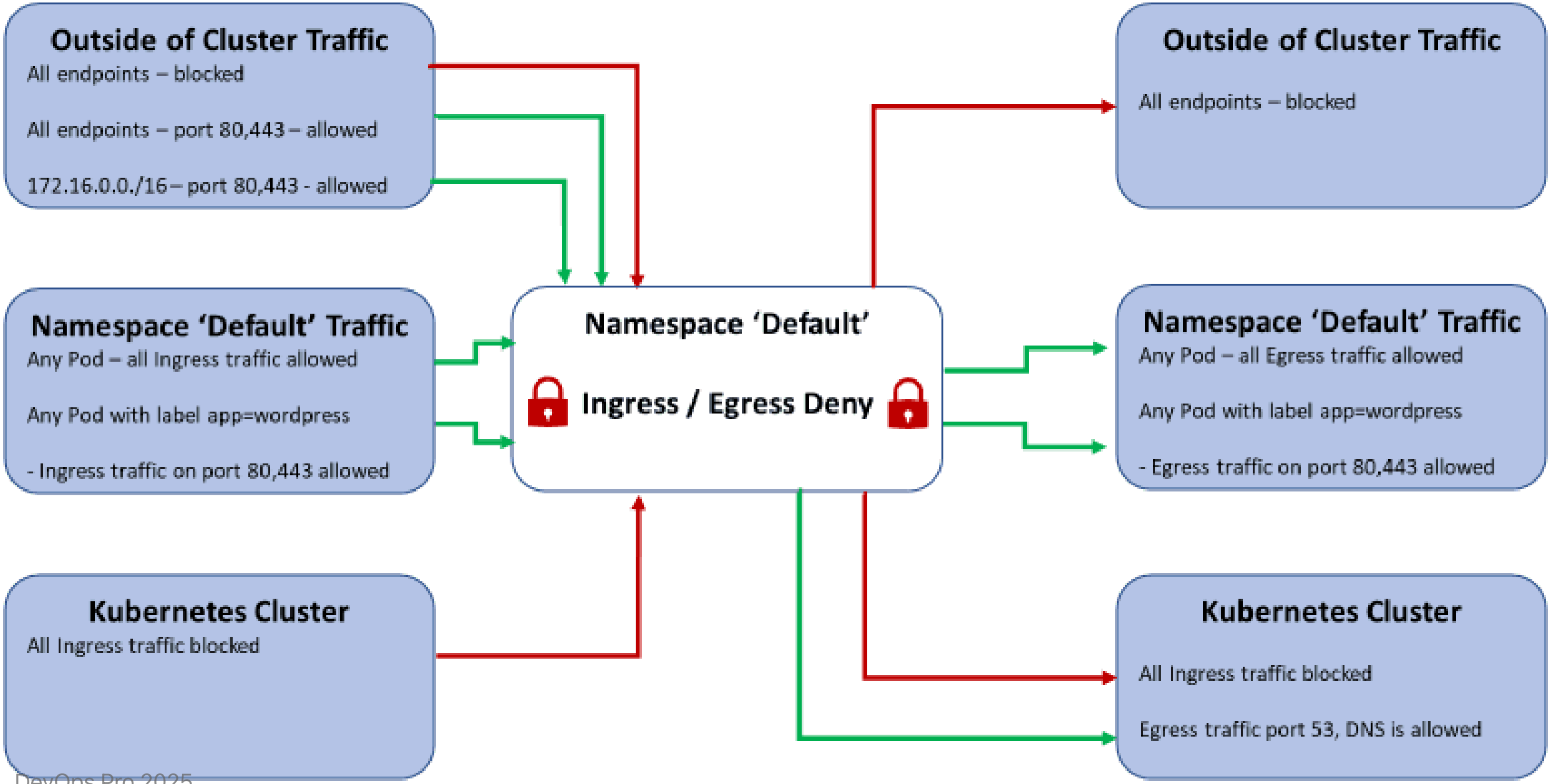
Lab RBAC (Role-Based Access Control) and Service Accounts

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

Network Policies

The entities that a Pod can communicate with are identified through a combination of the following 3 identifiers:

- Other pods that are allowed (exception: a pod cannot block access to itself)
- Namespaces that are allowed
- IP blocks (exception: traffic to and from the node where a Pod is running is always allowed, regardless of the IP address of the Pod or the node)


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** - Selects a group of Pods using a LabelSelector.
  - If empty, selects all Pods in the namespace (use with caution).
- **policyTypes** - Specifies the type of rules: `Ingress`, `Egress`, or both.
- **ingress** - Defines rules for incoming traffic to selected Pods.
  - If empty, matches all ingress traffic.
  - If absent, does not affect ingress traffic.
- **egress** - Defines rules for outgoing traffic from selected Pods.
  - If empty, matches all egress traffic.
  - If absent, does not affect egress traffic.

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

An array of rules that would be applied to the traffic going out of the pod. It is defined with the following fields.

Fields:

ports: an array of NetworkPolicyPort (port, endport, protocol)

to: an array of NetworkPolicyPeer (ipBlock, namespaceSelector, podSelector)

Ingress Rules

An array of rules that would be applied to the traffic coming into the pod. It is defined with the following fields.

Fields:

from: an array of NetworkPolicyPeer (ipBlock, namespaceSelector, podSelector)

ports: an array of 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

As with any network security configuration, we should employ several best practices for our Kubernetes network policy:

- Use the default deny-all network policy to ensure that only explicitly permitted communication occurs.
- Group Pods that must communicate with one another using the PodSelector parameter.
- Only allow inter-namespace communication when necessary.
- Don’t allow unnecessary network communication — even within the Kubernetes cluster.
- Use caution when allowing Pods within the cluster to receive non-cluster network traffic.
- Denying outgoing public internet traffic might interfere with specific application

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

run as user example

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

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: \

- `Localhost` with which a localhostProfile setting provides a path inside the container to a seccomp profile
- `Unconfined` in which no profile is applied.
- `RuntimeDefault` in which the container runtime default is used–this is the default if the type is left unspecified

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

Практика

Lab Network Policies for Micro-segmentation

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/k8s_advanced_labs.md#lab-3-network-policies-for-micro-segmentation

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

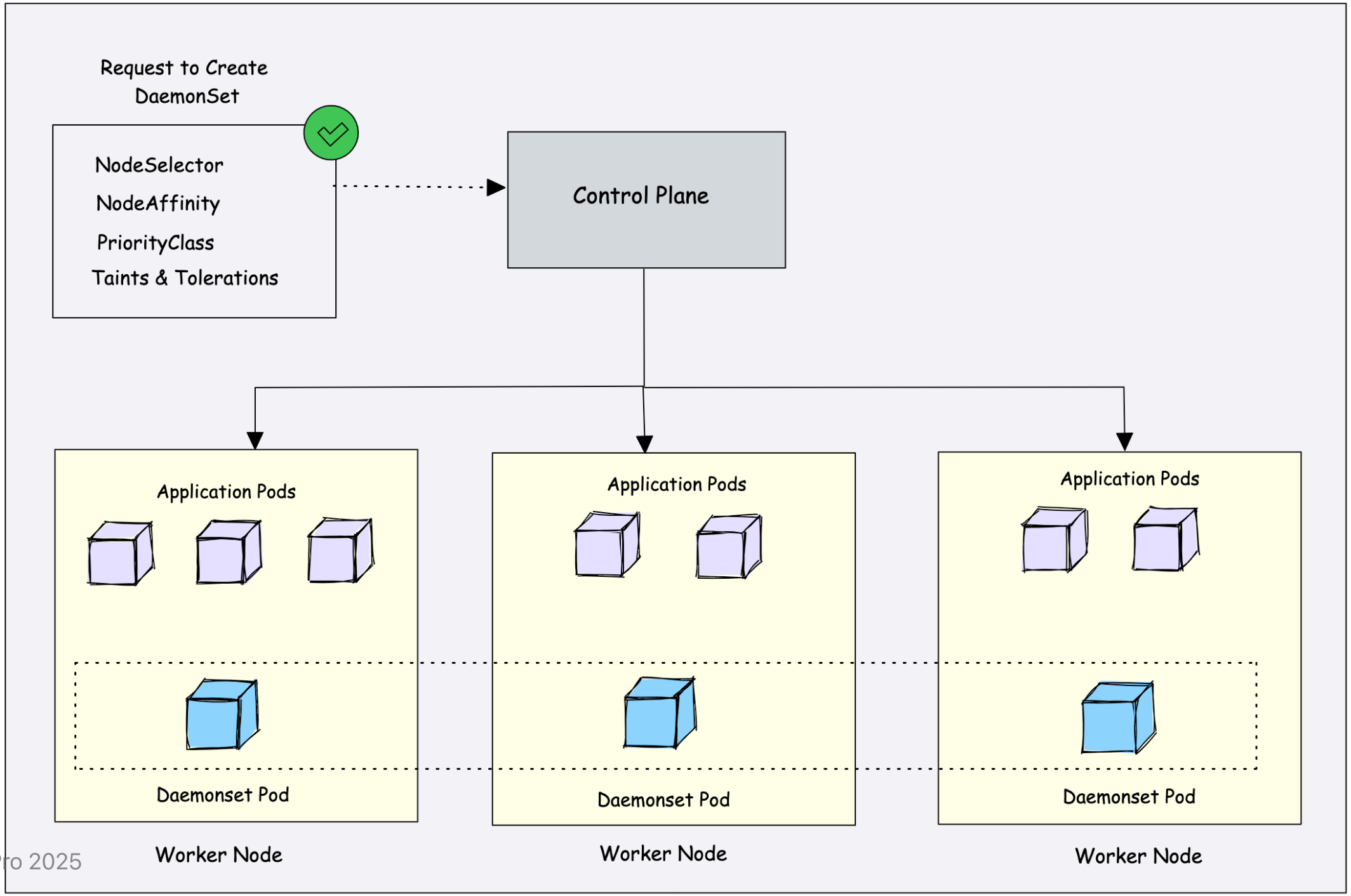
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    volumeMounts:
    - mountPath: /data
      name: my-volume
  volumes:
  - name: my-volume
    persistentVolumeClaim:
      claimName: my-pvc
```

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:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

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` 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.

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. Это полезно для выполнения периодических задач, таких как резервное копирование данных или отправка отчетов.

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
```

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.

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