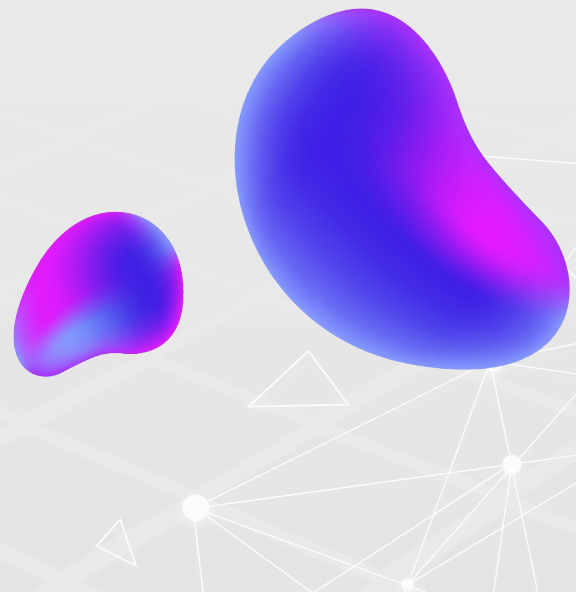


# Введение в Kubernetes День 1

Летняя школа ВШЭ 2024.





# Содержание

**01**

Ядро и примитивы  
Linux

**02**

Процессы, Namespace,  
Cgroups

**03**

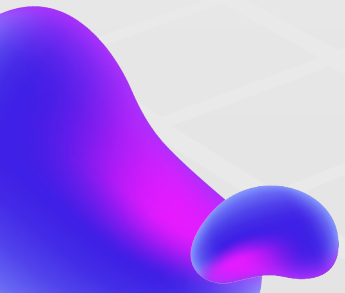
Монолитная и  
микросервисная  
архитектура

**04**

Архитектура  
Kubernetes

**05**

Декларативный  
подход Kubernetes





## Обо мне

Общий стаж в ИТ – 13 лет, из них 7 лет в кибербезопасности.

Сейчас работаю в отделе безопасности инфраструктуры Okko.



01

# Ядро и примитивы Linux

# Перед тем как мы будем обсуждать контейнеры

Все приложения, включая контейнеризированные полагаются на лежащее в основе ядро операционной системы

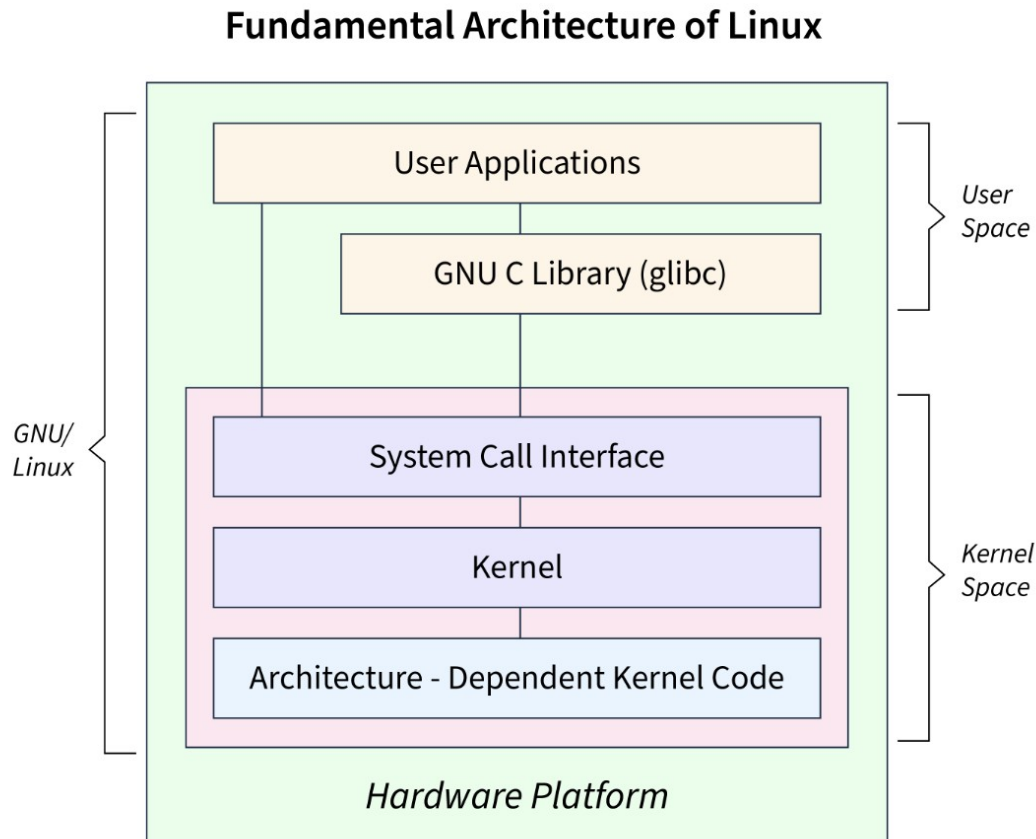
Ядро предоставляет API приложениям с помощью механизма системных вызовов (syscalls)

Версии и наборы системных вызовов важны, поскольку являются клеем между пространством пользователя и пространством ядра.

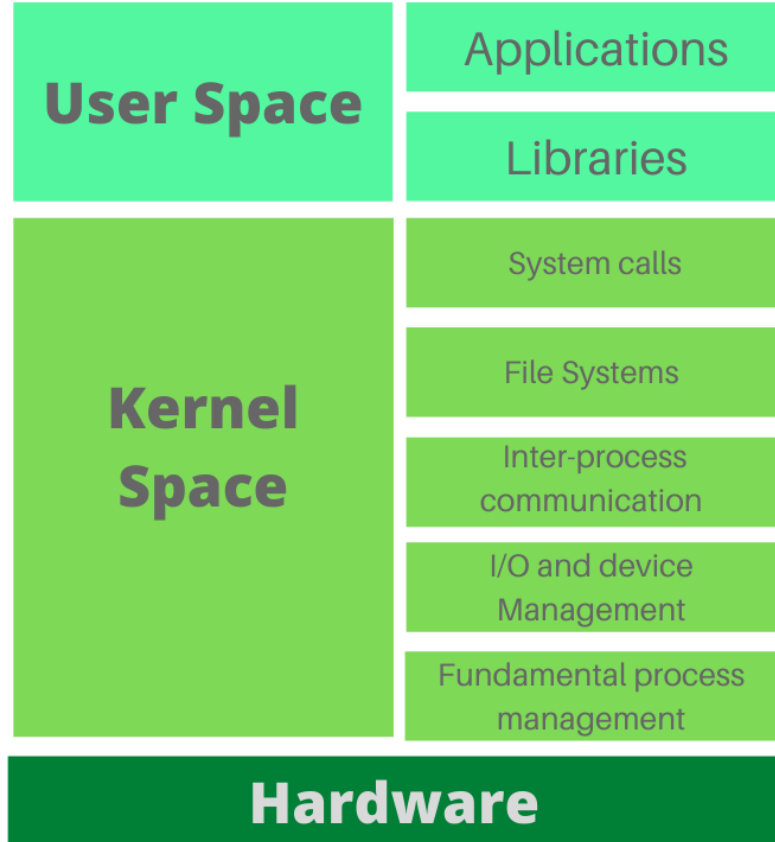
# Ядро GNU/Linux

Ядро - выполняет роль основного интерфейса между аппаратным обеспечением компьютера и его процессами.

Управляет системными ресурсами, обеспечивая достаточное распределение памяти для приложений, оптимизируя использование процессора и предотвращая зависания системы, возникающие из-за конфликтов между требованиями различных приложений



# User space VS Kernel space



## User Space

Область, где работают пользовательские приложения и утилиты

Выполняет программы, обращающиеся к системным ресурсам через системные вызовы

bash, ls, Python, контейнеры Docker

Не имеет прямого доступа к оборудованию — использует ядро для выполнения операций

## Kernel Space

Привилегированная область, где работает ядро операционной системы

Управляет доступом к аппаратным ресурсам, обрабатывает системные вызовы

Управление файлами, процессами, памятью и сетью

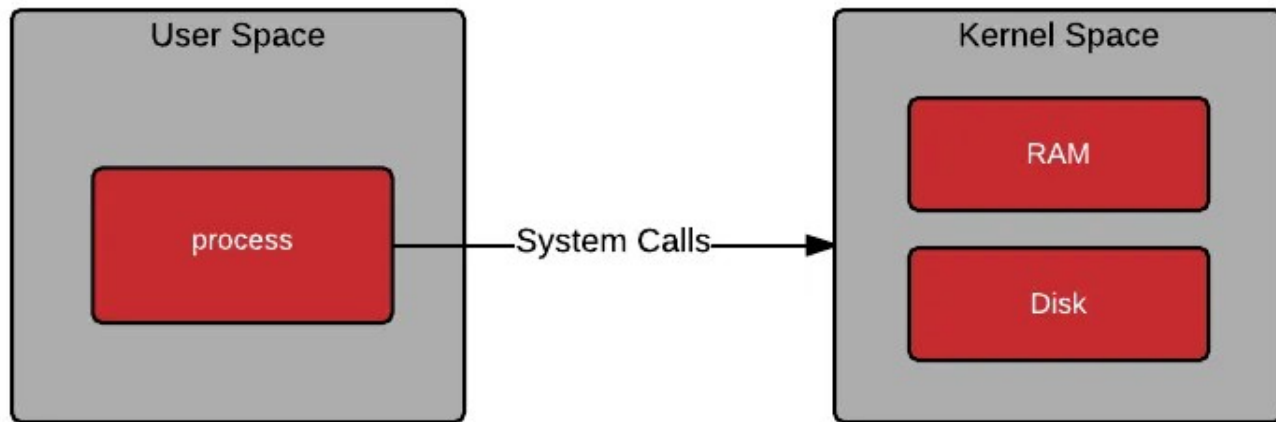
Обеспечивает безопасность и стабильность системы, контролируя взаимодействие приложений с оборудованием



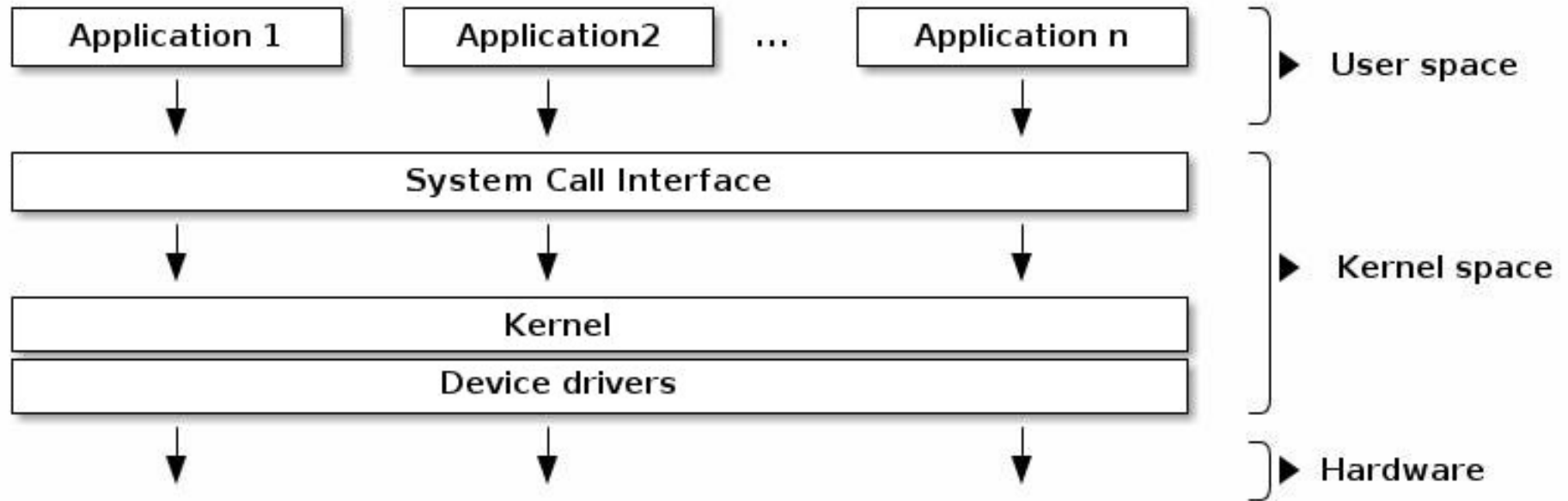
# Системные вызовы (syscalls)

Системный вызов — это механизм, с помощью которого программа запрашивает услуги (ресурсы) у ядра операционной системы.

Это единственный стандартный способ для программ в пользовательском пространстве взаимодействовать с операционной системой на уровне ядра



# Системные вызовы (syscalls)



# Какие услуги предоставляют вызовы?

## Управление процессами

Создание процесса - `fork()`  
Запуск программы - `execve()`  
Завершение процесса - `exit()`

## Управление памятью

Выделение памяти: `mmap()`  
Освобождение  
памяти: `munmap()`

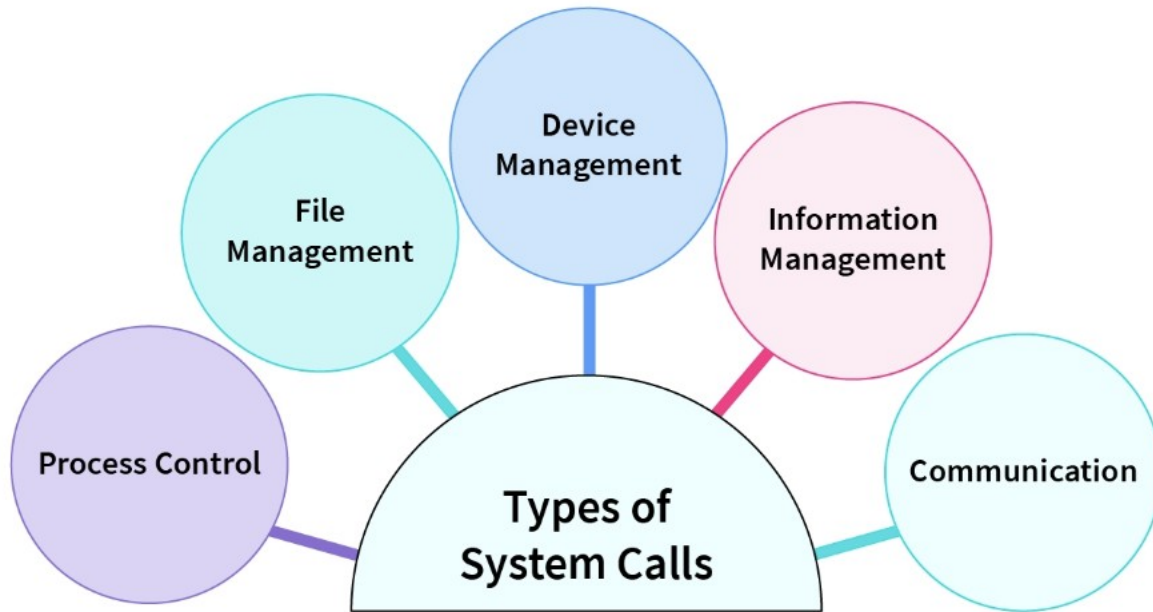
## Управление файлами

Открытие файла - `open()`  
Чтение файла - `read()`  
Запись в файл - `write()`  
Закрытие файла - `close()`

## Управление сетью

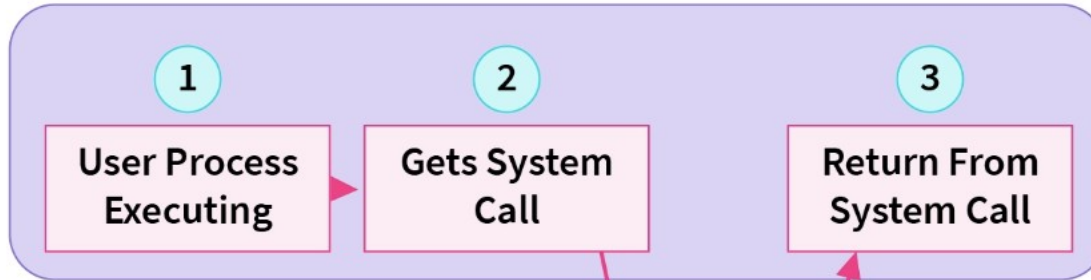
Создание сетевого  
соединения: `socket()`  
Отправка данных: `send()`  
Получение данных: `recv()`

# Типы системных вызовов

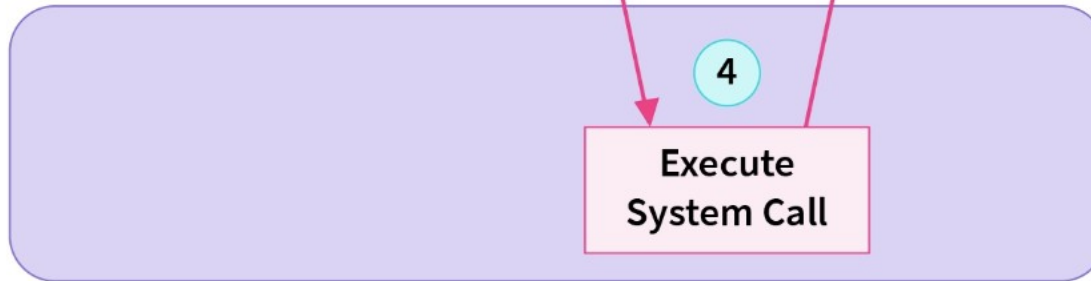


# Системные вызовы

## USER MODE



## KERNEL MODE



# Сценарии использования syscall

1. Открытие файла (open) - программа открывает файл на диске, получая дескриптор файла для дальнейших операций.
2. Чтение данных (read) - программа читает данные из файла в память, используя дескриптор файла.
3. Запись данных (write) - программа записывает изменения в файл через дескриптор файла.
4. Закрытие файла (close) - программа закрывает файл, освобождая ресурсы, связанные с дескриптором.

# Recap

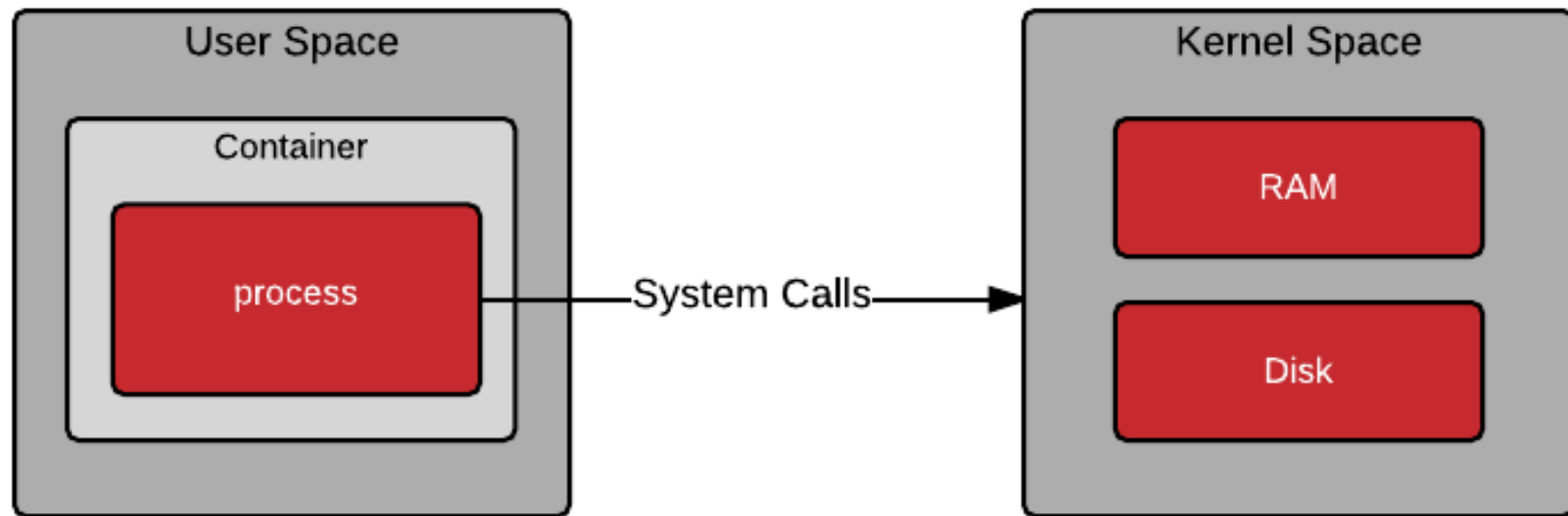
Ядро Linux является центральной частью ОС, поскольку является интерфейсом между «железом» и запущенными программами.

Одними из важнейших задач помимо распределения ограниченных ресурсов является так же изоляция процессов и безопасность

ОС Linux архитектурно разделена на 2 части kernel space и user space. Все пользовательские программы выполняются в user space.

Чтобы программе получить ресурсы для выполнения – ей нужно их запросить у ядра операционной системы через интерфейс (механизм) системных вызовов (syscalls).

# А что с контейнерами?





# Как происходят вызовы?

User Programs

Library/Interpreter

System Calls

Kernel Space



The slide features a light gray background with a subtle, repeating diamond-shaped grid pattern. In the top-left corner, there is a small cluster of white lines radiating from a central point, with a few small white triangles scattered nearby. In the bottom-right corner, there is a similar cluster of white lines and triangles, including a small white circle at one of the intersection points.

Вопросы?

# Что такое примитивы Linux?

Это основные функции и механизмы, которые обеспечивают базовые операции, такие как управление процессами, памятью, устройствами ввода-вывода, синхронизацию и межпроцессное взаимодействие.

На основе примитивов строятся более сложные системы, которые могут включать в себя множество примитивов.

<https://github.com/opencontainers>

# Вы уже с ними столкнулись

Системные вызовы – это примитив ОС, который предоставляет элементарный и базовый уровень взаимодействия между программами.

Процесс – так же примитив ОС, поскольку является базовой единицей выполнения программ в ОС.

Примитивов в каждой системе много, но они являются «кирпичиками» на которых строятся сложные системы.



# Контейнеры

Контейнеры – система, как и любая другая состоящая из примитивов, нам же важно понять эти примитивы.

Чтобы понять что такое контейнер – необходимо разобраться в его примитивах, мы сосредоточимся на 3х главных:

- Процессы
- Namespaces
- Cgroups



02

# Процессы, Namespace, Cgroups

# Процессы Linux

Процесс – это программа, которая выполняется. Включает в себя код программы, данные, необходимые для её работы, и текущее состояние выполнения (например, какая команда выполняется в данный момент).

Каждый процесс имеет свой уникальный идентификатор (PID) и работает независимо от других процессов

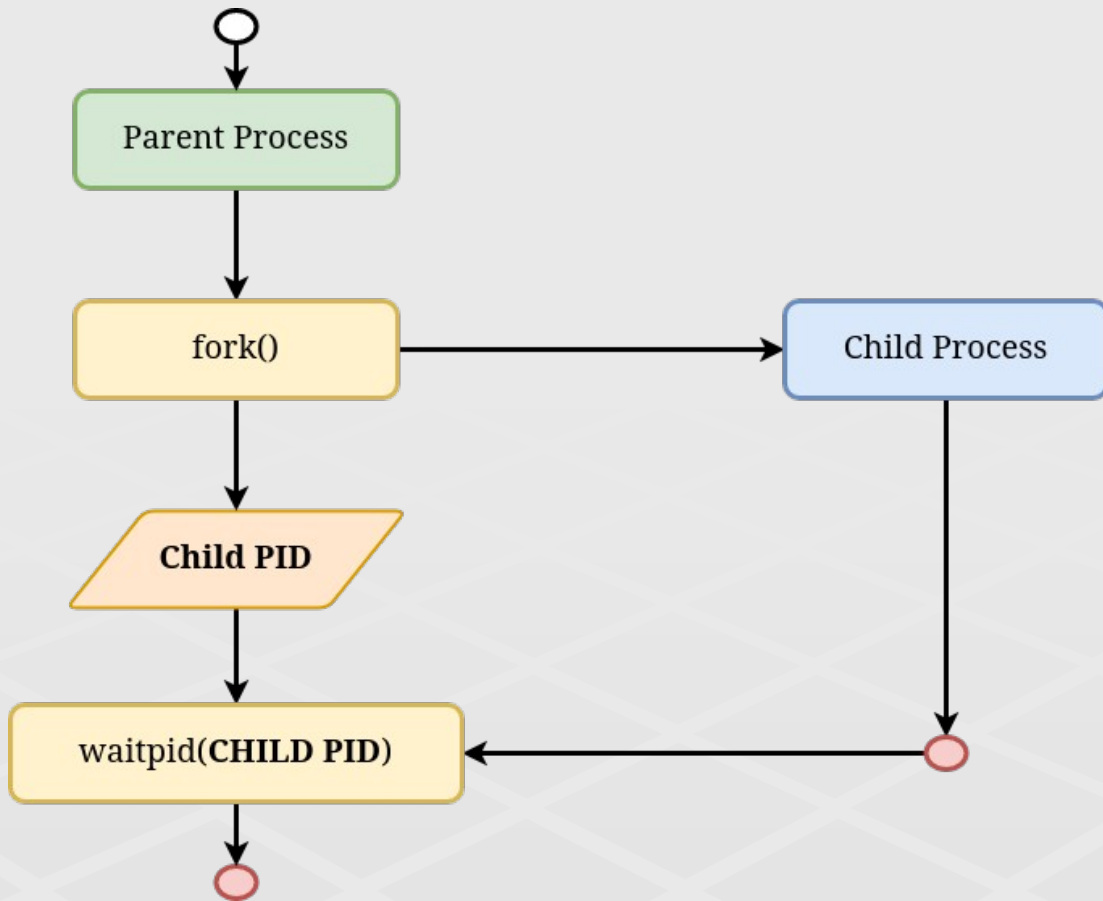
Процессы могут создавать другие процессы, запускать программы, взаимодействовать с файлами и устройствами, и обычно изолированы друг от друга, чтобы обеспечить стабильность и безопасность системы.



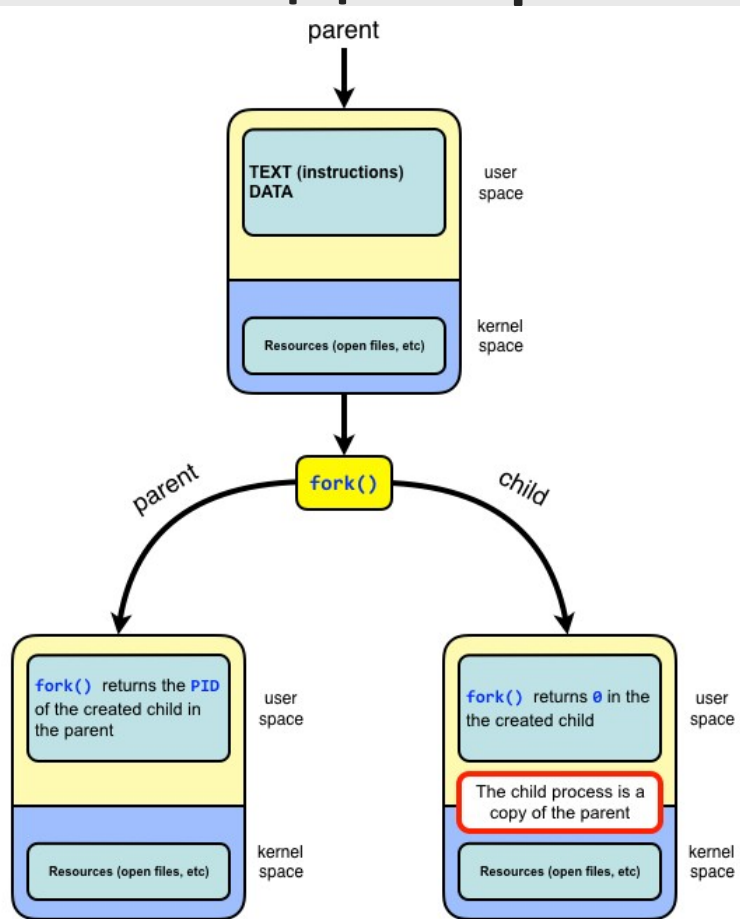
# Иерархия процессов

```
eu@serv1:~$ pstree -n
systemd--dbus-daemon
        |--systemd-logind
        |--unattended-upgr--{unattended-upgr}
        |--agetty
        |--k3s-server--13*[{k3s-server}]
                        |--containerd--16*[{containerd}]
        |--dockerd--8*[{dockerd}]
        |--containerd-shim--13*[{containerd-shim}]
                        |--pause
                        |--traefik--7*[{traefik}]
        |--containerd-shim--14*[{containerd-shim}]
                        |--pause
```

# Родительский и дочерний процессы

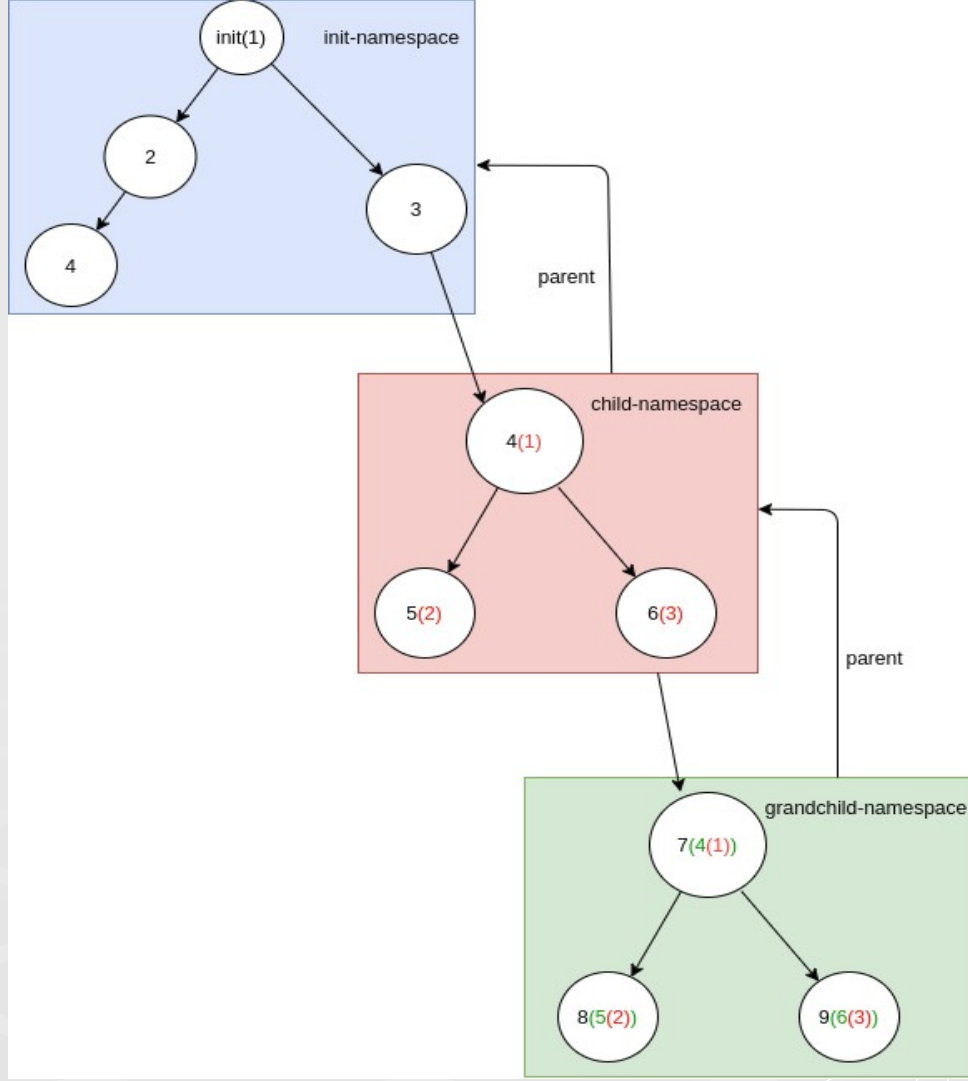


# Родительский и дочерний процессы



# Дочерние процессы

Кроме простого syscall `fork()` есть чуть более сложный системный вызов `clone()`, в отличие от `fork()` он позволяет передать контекст запускаемому процессу.



# Системный вызов clone()

clone() — создает новый процесс и может одновременно создавать новые пространства имен, используя флаги:

CLONE\_NEWNS

CLONE\_NEWPID

CLONE\_NEWNET

CLONE\_NEWIPC

CLONE\_NEWUTS

CLONE\_NEWUSER

CLONE\_NEWCGROUP



# Определение Namespaces (пространств имен)

Пространства имен — это функция ядра Linux, которая разделяет ресурсы ядра таким образом, что один набор процессов видит один набор ресурсов, а другой набор процессов видит другой набор ресурсов. Функция работает, имея одно и то же пространство имен для группы ресурсов и процессов, но эти пространства имен ссылаются на разные ресурсы.



# Зачем придумали Namespaces?

Изоляция процессов и сервисов

Запускать множество сервисов на одном сервере, обеспечивая их изоляцию друг от друга.

Повышение безопасности

Предотвращение взаимодействия или конфликта между разными процессами или сервисами на одном сервере

Эффективное использование ресурсов

Управлять использованием системных ресурсов (например, CPU, память, сетевые интерфейсы) различными процессами и сервисами

# Системный вызов Unshare()

Используется в Linux для того, чтобы уже запущенный процесс мог отделить свою часть окружения от других процессов, с которыми он это окружение делит, создавая при этом изолированные пространства имен (namespaces) для управления ресурсами, такими как идентификаторы процессов, сеть, файловая система и другие системные компоненты.

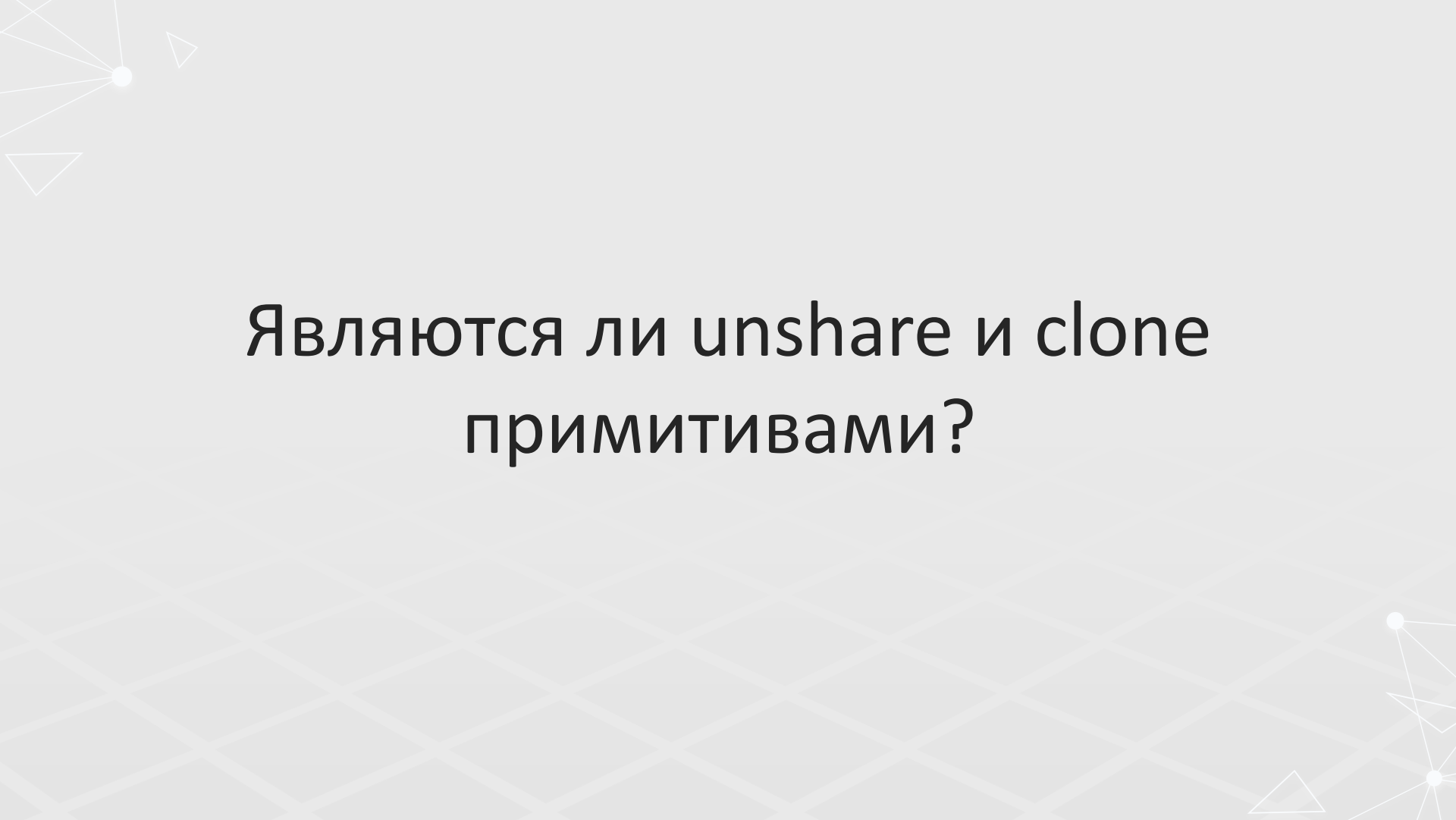
Процесс до unshare:



Процесс после unshare:





The background features decorative geometric patterns. In the top-left corner, there is a network of thin white lines connecting several small white circles and triangles. In the bottom-right corner, a similar network of lines and shapes is visible. The bottom half of the slide is covered by a large, light-gray diamond-shaped grid pattern.

Являются ли unshare и clone  
примитивами?

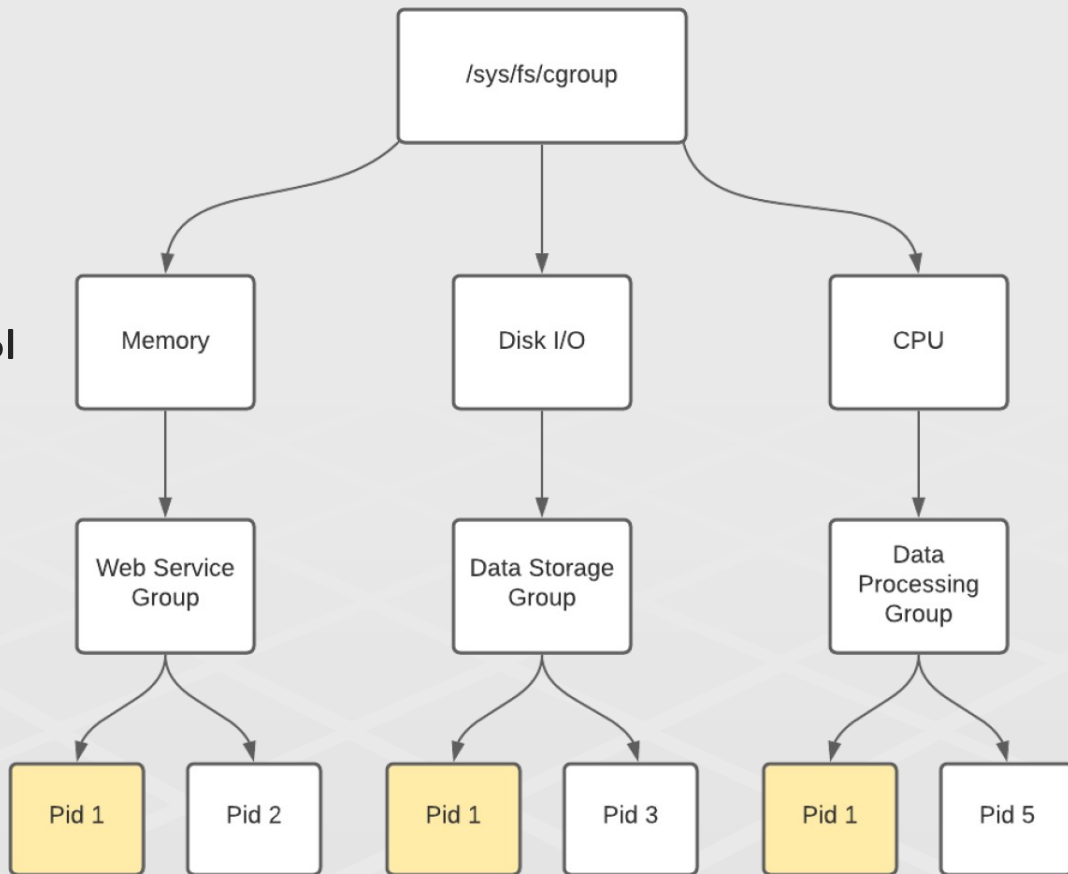
# Как работают cgroups?

Root cgroup

Контроллеры

Группы  
процессов

Процессы



# Что такое PID?

Когда создается процесс, ему присваивается определенный числовой идентификатор, называемый идентификатором процесса (PID).

PID помогает однозначно идентифицировать процесс, даже если есть два процесса с одинаковым понятным человеку именем

Все эти процессы отслеживаются в специальной файловой системе, называемой `procfs (/proc)`.

Если вы выполните листинг `/proc`, вы увидите папку для каждого процесса, запущенного в данный момент в вашей системе.

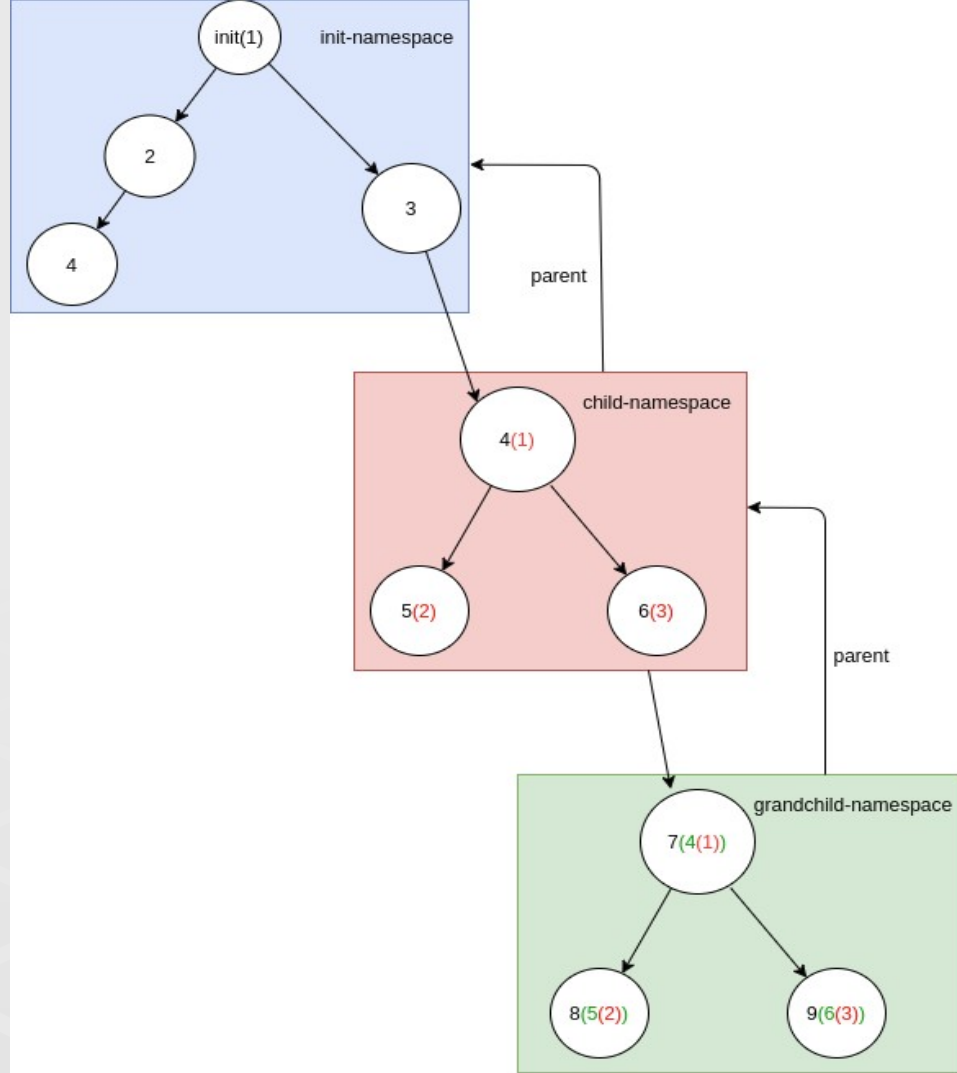
# Что такое PID namespace?

Пространства имен PID изолируют пространство номеров идентификаторов процессов, что означает, что процессы в разных пространствах имен PID могут иметь одинаковый PID.

‘Пространства имен PID позволяют контейнерам предоставлять такие функции, как приостановка/возобновление набора процессов в контейнере и миграция контейнера на новый хост, при этом процессы внутри контейнера сохраняют те же PID.’

[https://man7.org/linux/man-pages/man7/pid\\_namespaces.7.html](https://man7.org/linux/man-pages/man7/pid_namespaces.7.html)

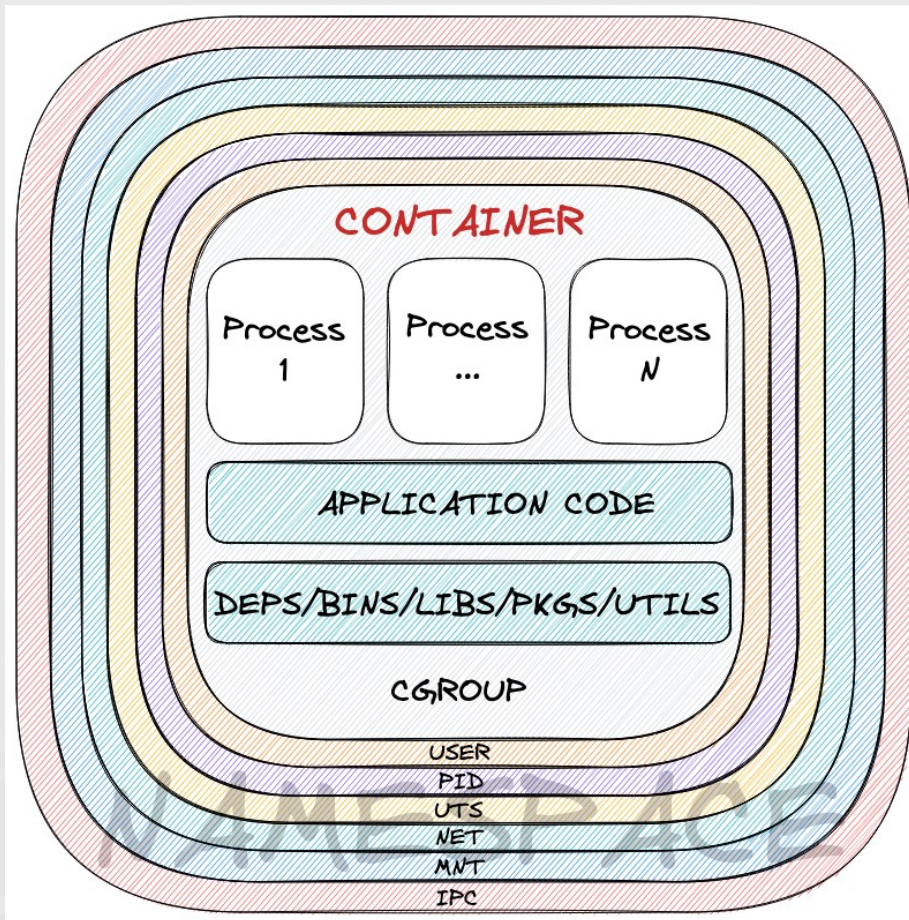
# Вложенные PID Namespaces



The background features a light gray diamond-shaped grid pattern. In the top-left corner, there is a line art graphic consisting of a central white dot with several lines radiating from it, and a few small triangles. A similar, more complex line art graphic is located in the bottom-right corner, featuring multiple dots and connecting lines.

# Контейнер - демо

# Что такое контейнер?





03

# Монолитная и микросервисная архитектура



# Архитектура ПО

## Структурная основа системы

Включает в себя выбор компонентов, их интерфейсов и взаимодействий.

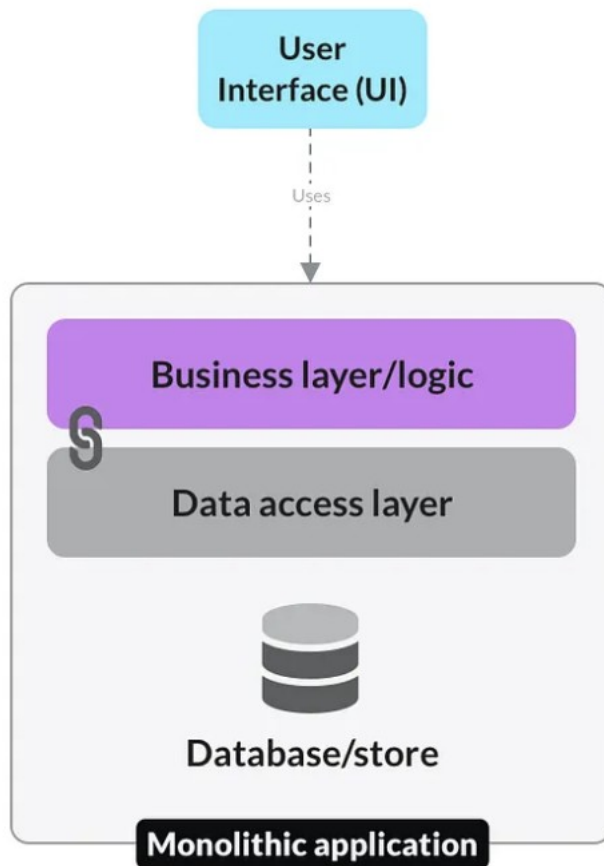
## Способ снижения сложности

Снижение сложности системы путем разделения её на модули и абстракции

## Средство коммуникации и документации

обеспечивающее фиксацию проектных решений и упрощение взаимодействия между участниками разработки

# Монолитная архитектура



# Монолитная архитектура

## Структура системы

Вся система — это единое приложение, где все компоненты работают вместе.

## Архитектурные характеристики

Простота разработки, но проблемы с масштабируемостью и отказоустойчивостью.

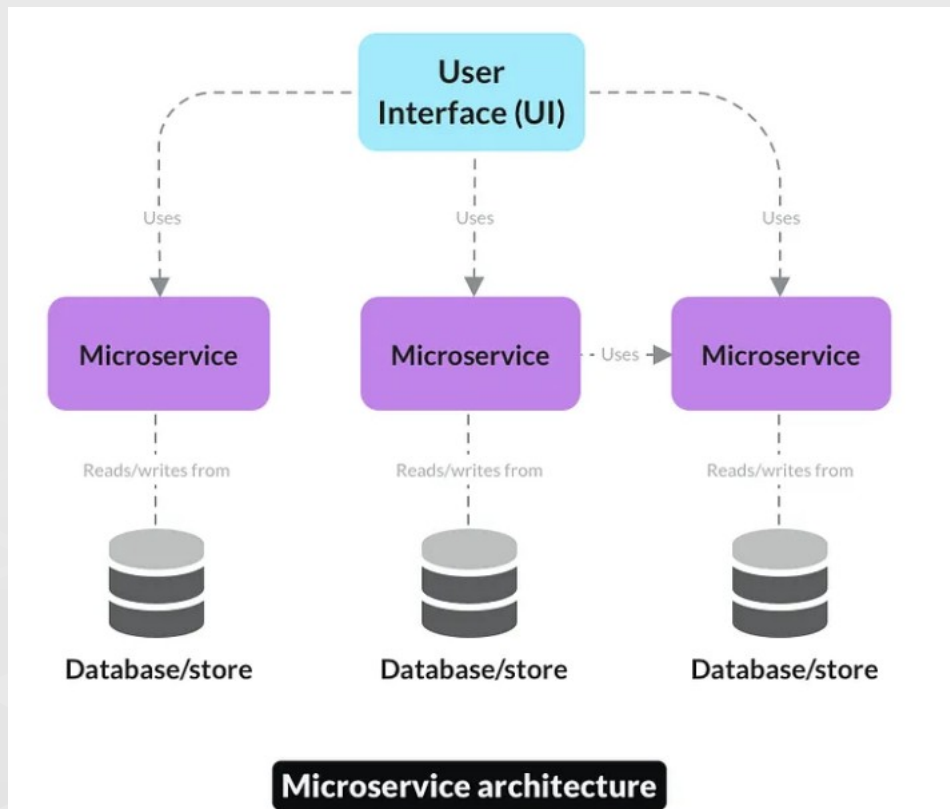
## Архитектурные решения

Использование единого стека технологий для всего приложения

## Принципы проектирования

Модульность внутри одного приложения с риском влияния изменений на всю систему

# Микросервисная архитектура



# Микросервисная архитектура

## Структура системы

Система разделена на автономные сервисы, взаимодействующие через API.

## Архитектурные характеристики

Легкое масштабирование и высокая отказоустойчивость, но сложность управления

## Архитектурные решения

Высокая производительность за счет отсутствия сетевых взаимодействий между компонентами

## Принципы проектирования

Каждый сервис выполняет одну функцию, упрощая изменение и тестирование

Фактор	Монолитная архитектура	Микросервисная архитектура
Масштабируемость	Требуется масштабирование всей системы	Масштабирование отдельных сервисов
Поддерживаемость	Простая отладка, но сложное обновление всего приложения	CI/CD уменьшает ошибки, требует сложного мониторинга
Отказоустойчивость	Сбой одного модуля может вызвать сбой всей системы	Сбой одного сервиса не влияет на всю систему
Технологическая зависимость	Использует один стек технологий	Возможность использования разных технологий для каждого сервиса



04

# Архитектура Kubernetes

# Проблемы до появления K8s

Сложность управления  
масштабируемыми  
средами

Самовосстановление и  
отказоустойчивость

Отсутствие  
оркестрации на уровне  
кластера

Балансировка нагрузки  
и управление  
трафиком



# Архитектура Kubernetes



master

# Архитектура Kubernetes



master



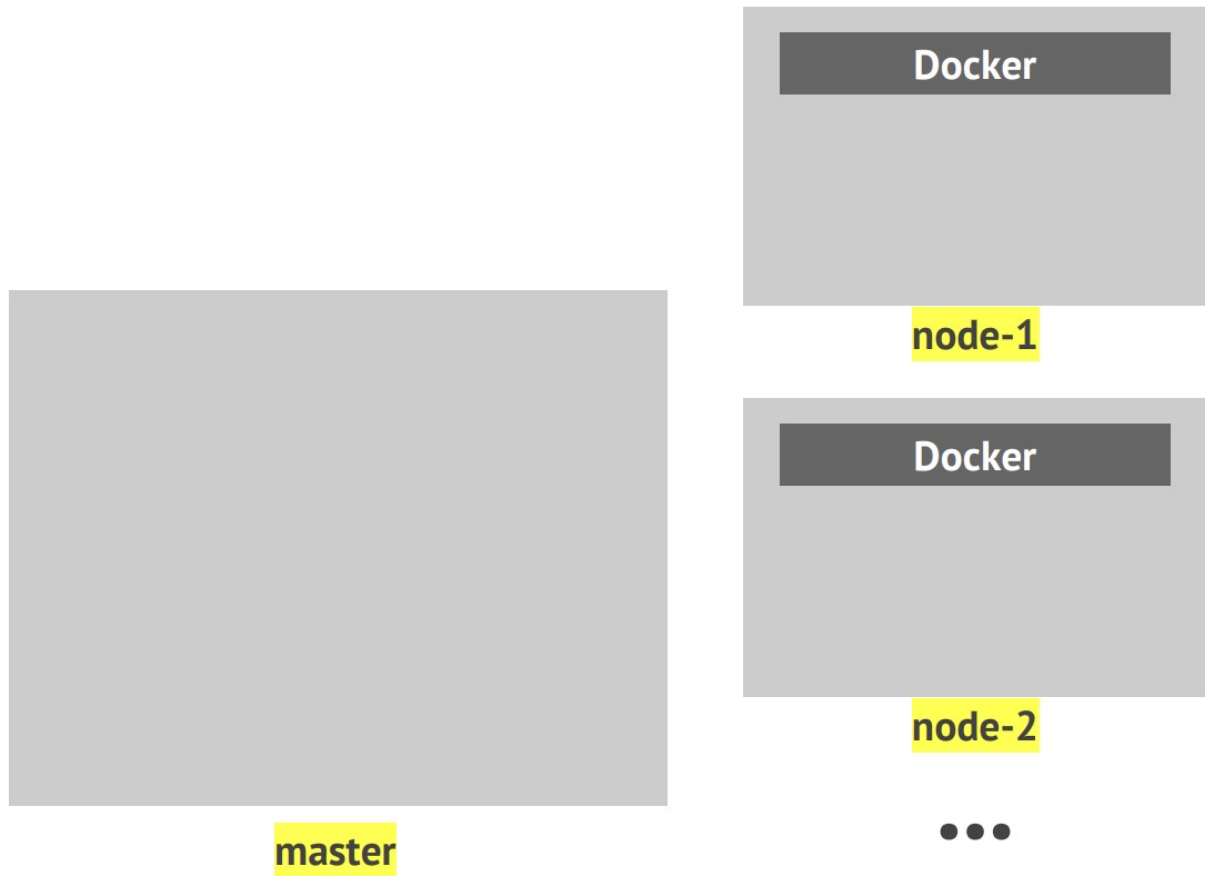
node-1



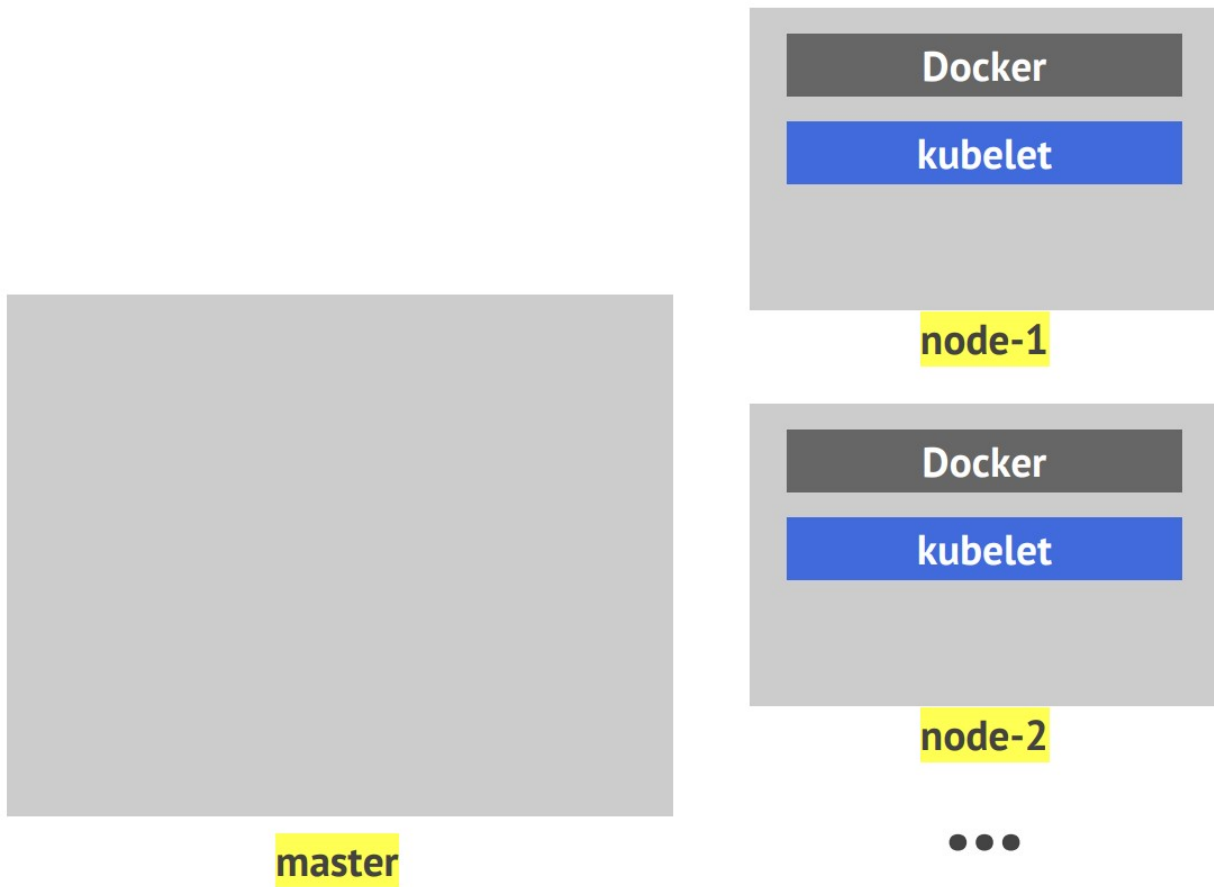
node-2

...

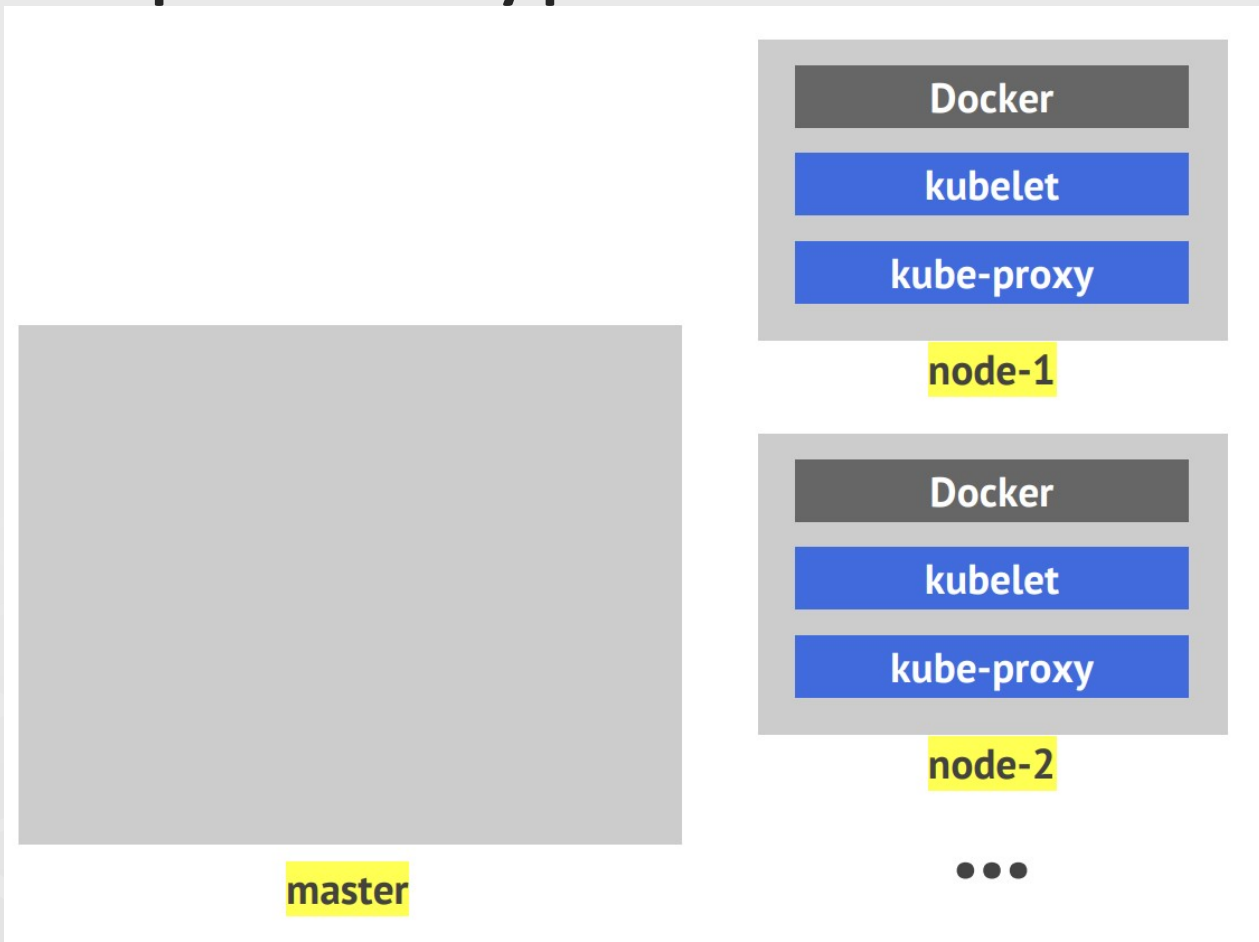
# Архитектура Kubernetes



# Архитектура Kubernetes



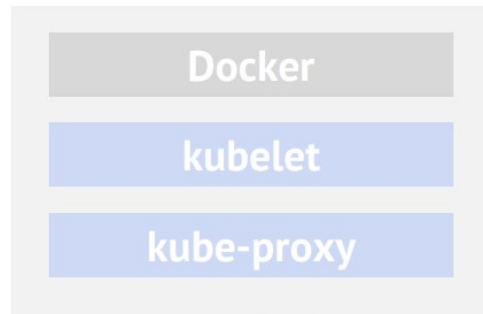
# Архитектура Kubernetes



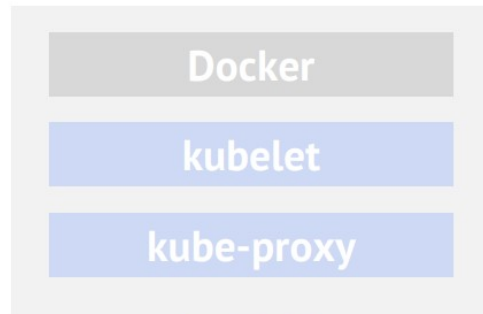
# Архитектура Kubernetes



**master**



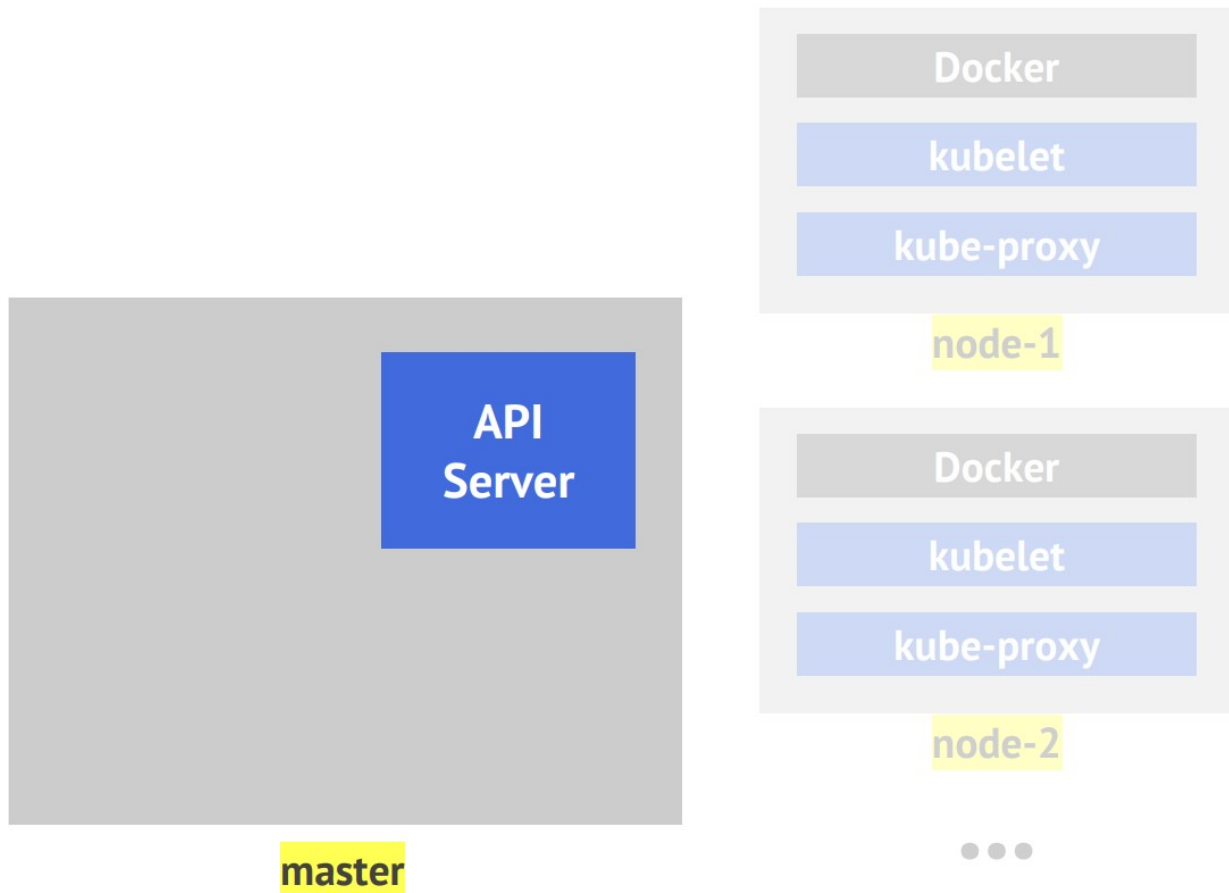
**node-1**



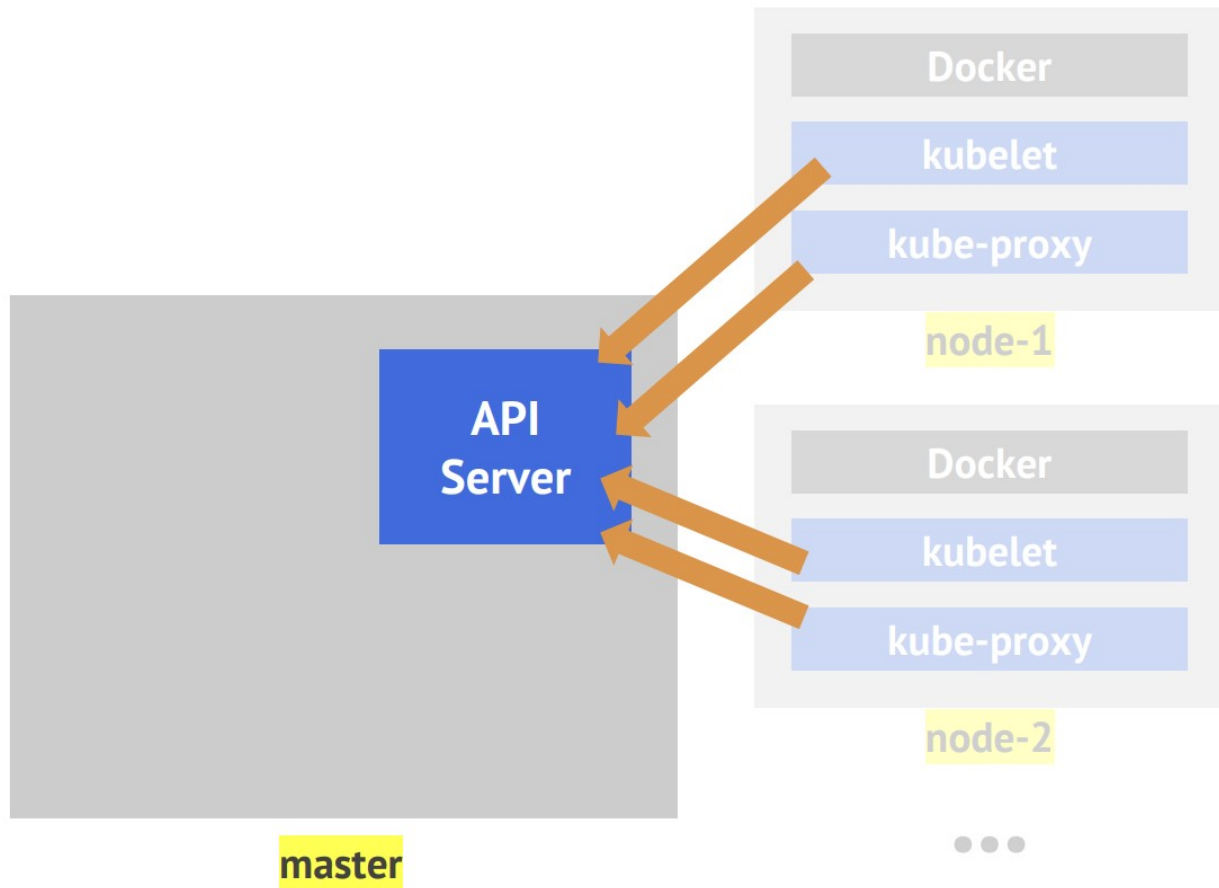
**node-2**

...

# Архитектура Kubernetes

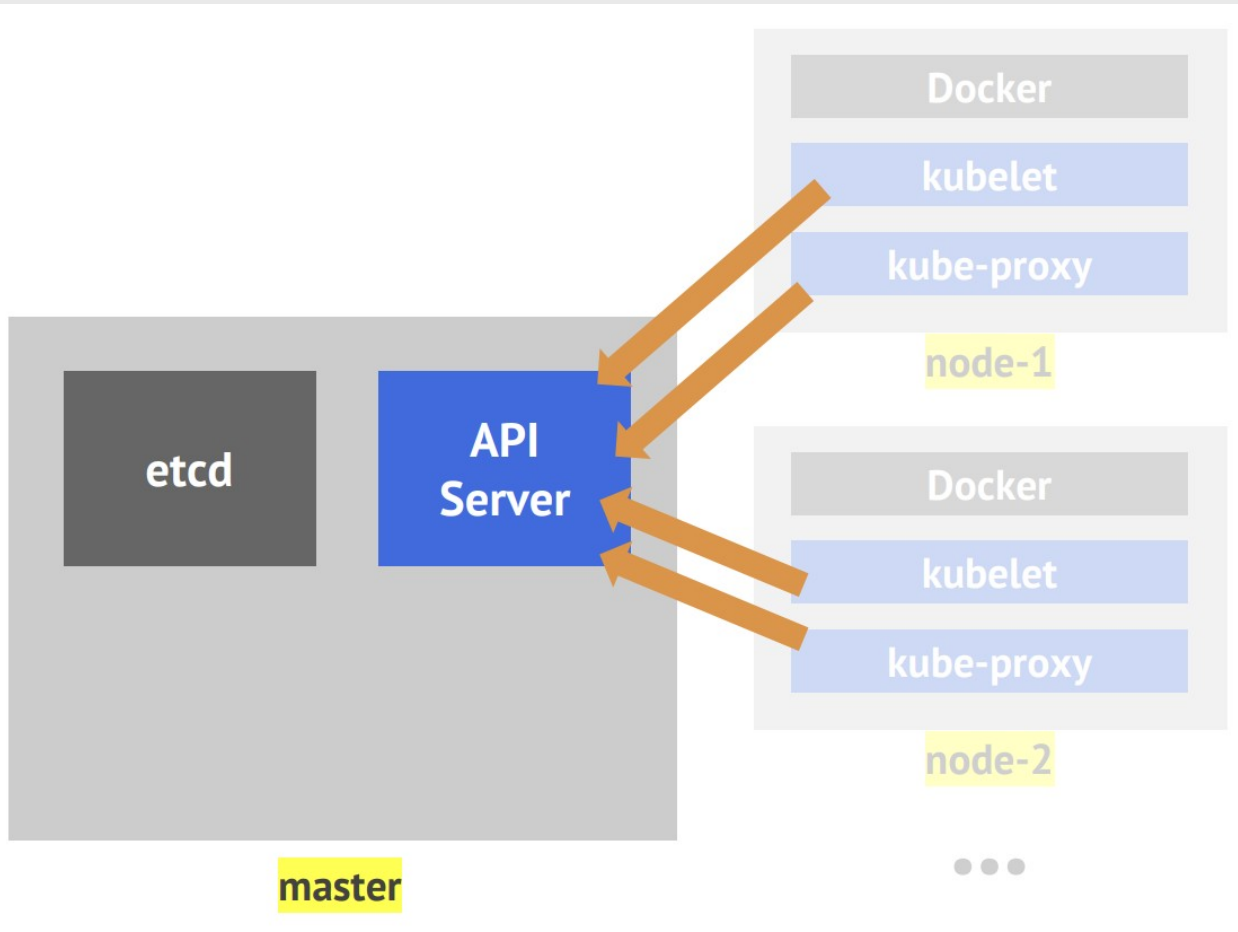


# Архитектура Kubernetes

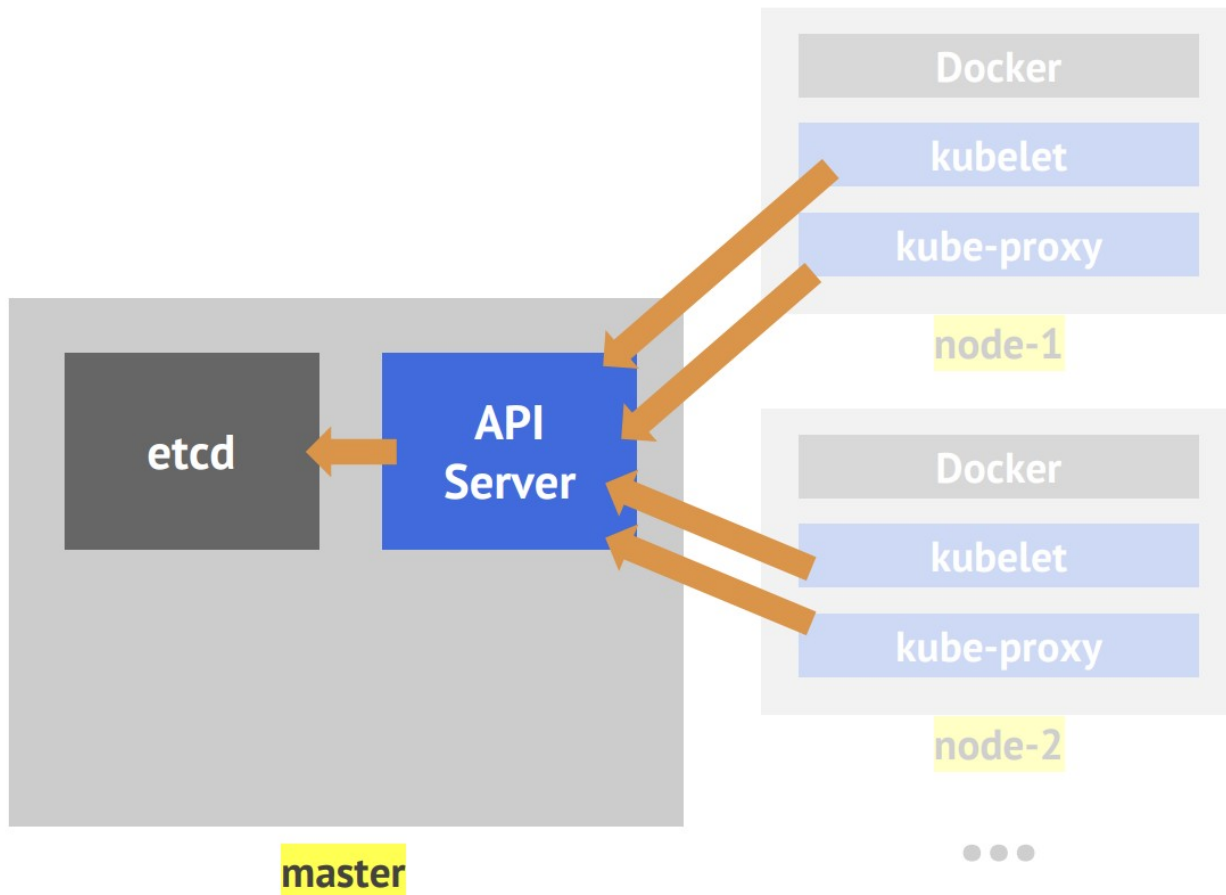




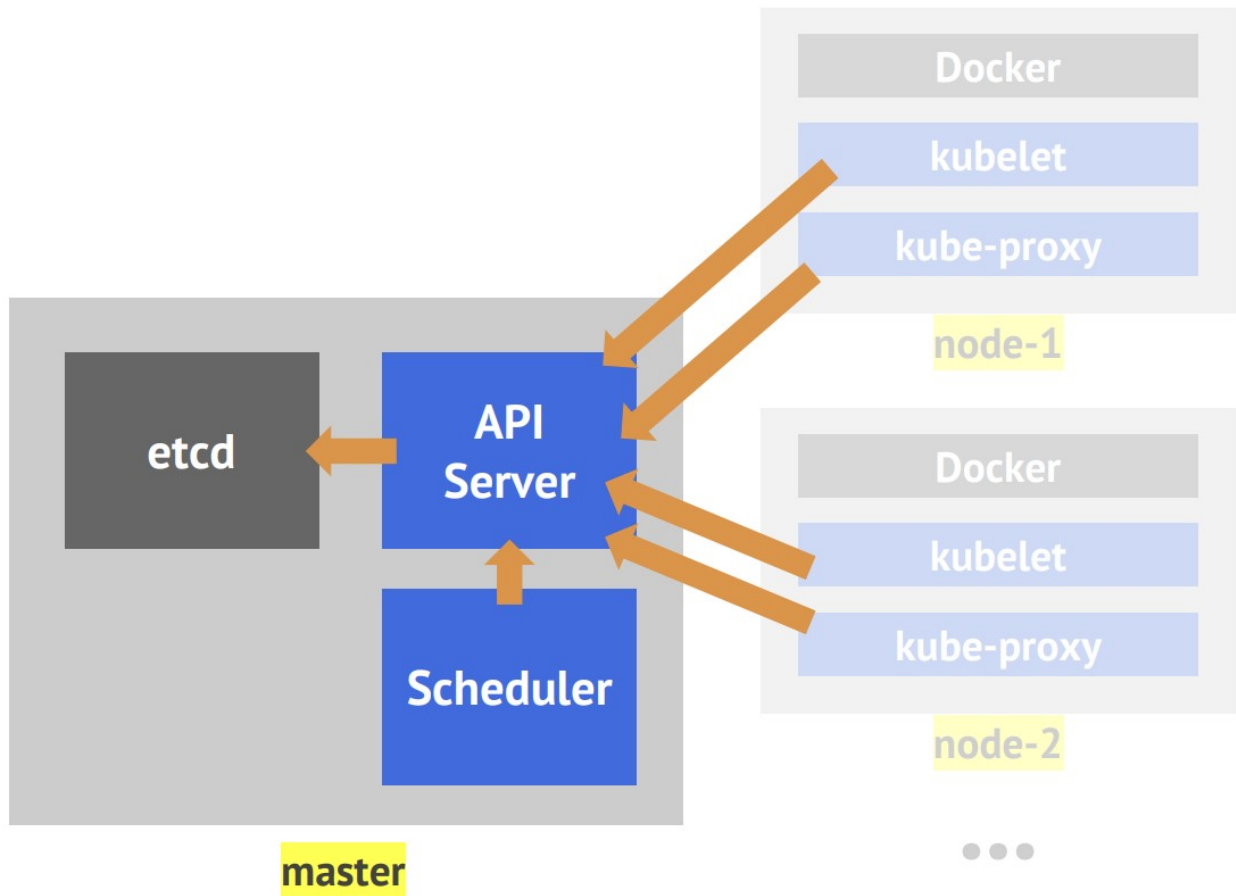
# Архитектура Kubernetes



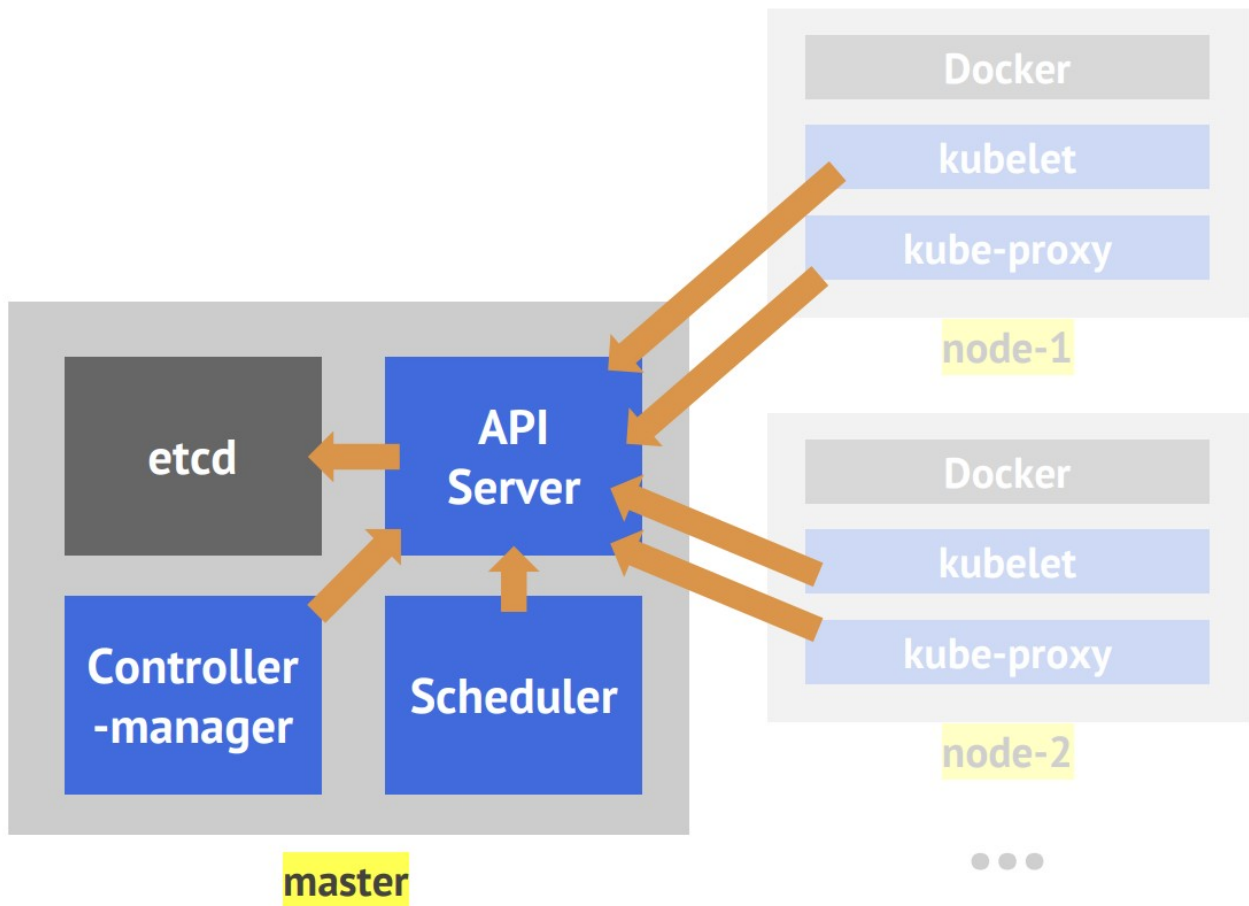
# Архитектура Kubernetes



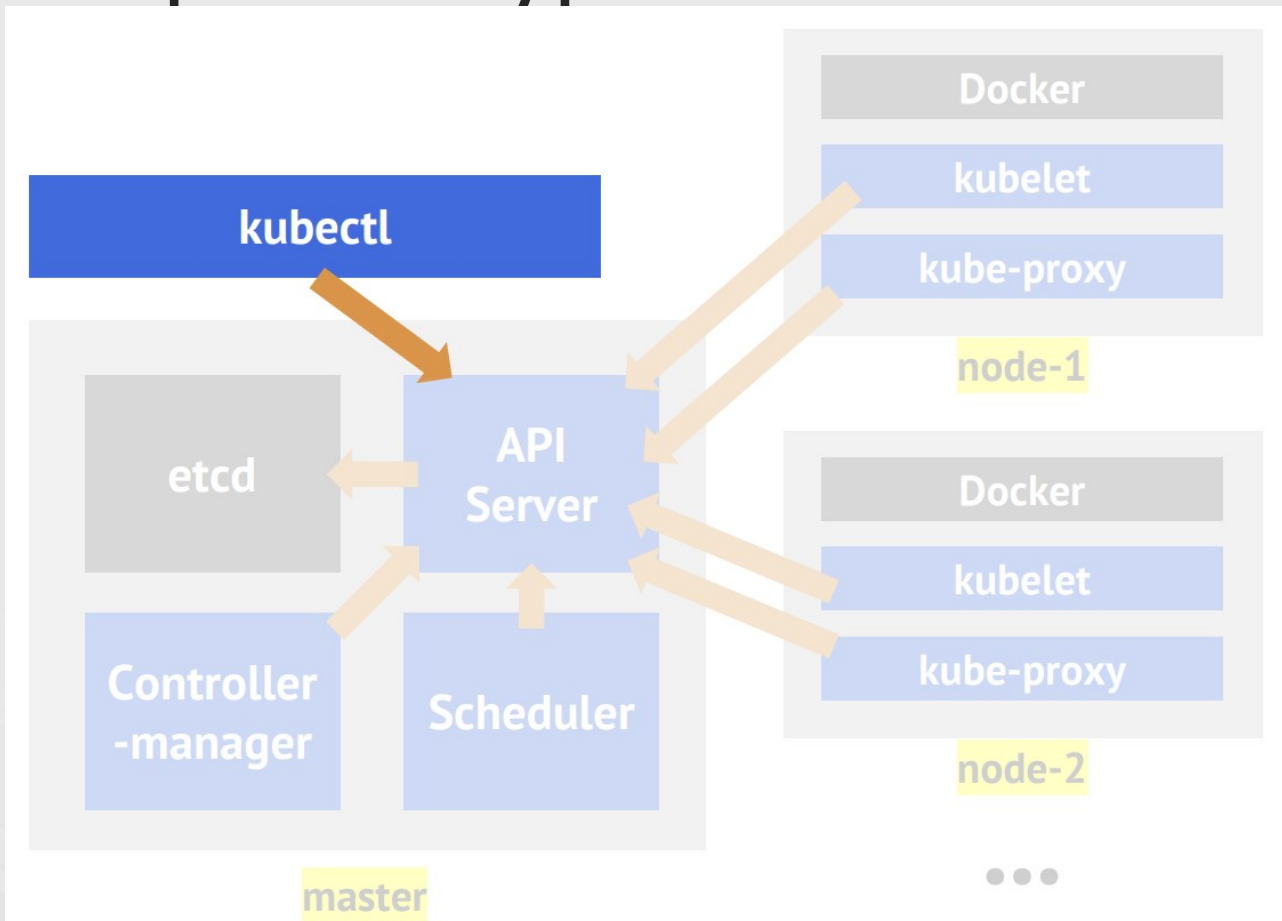
# Архитектура Kubernetes



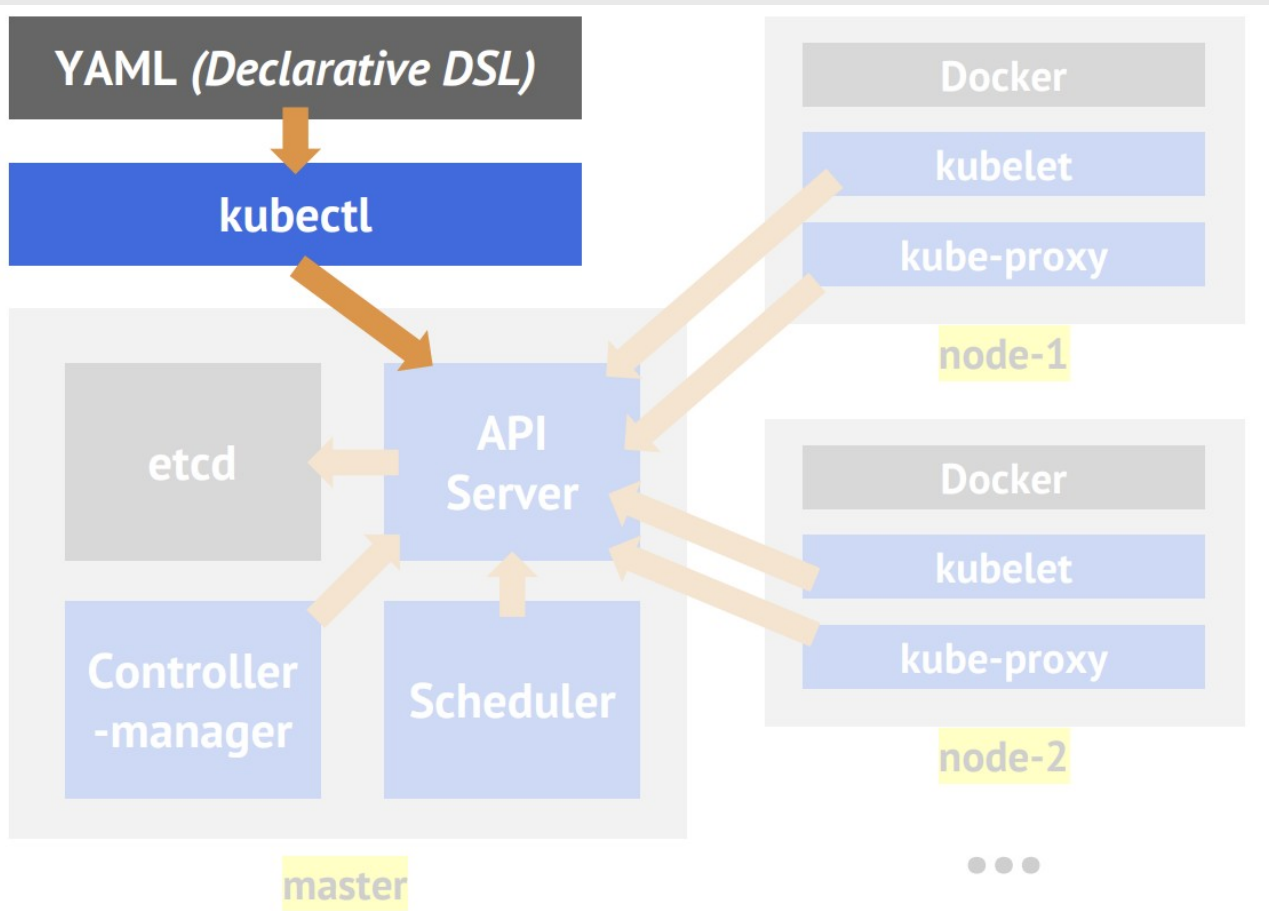
# Архитектура Kubernetes



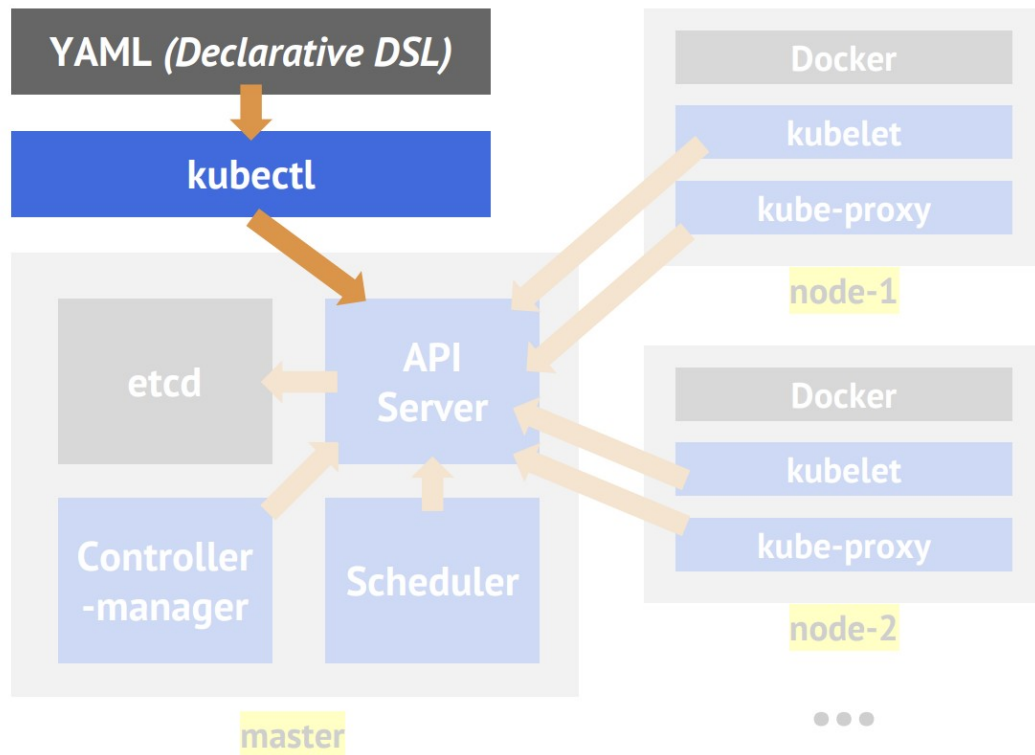
# Архитектура Kubernetes



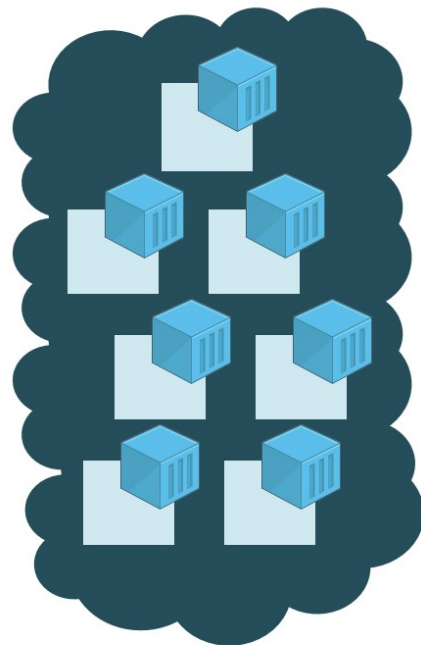
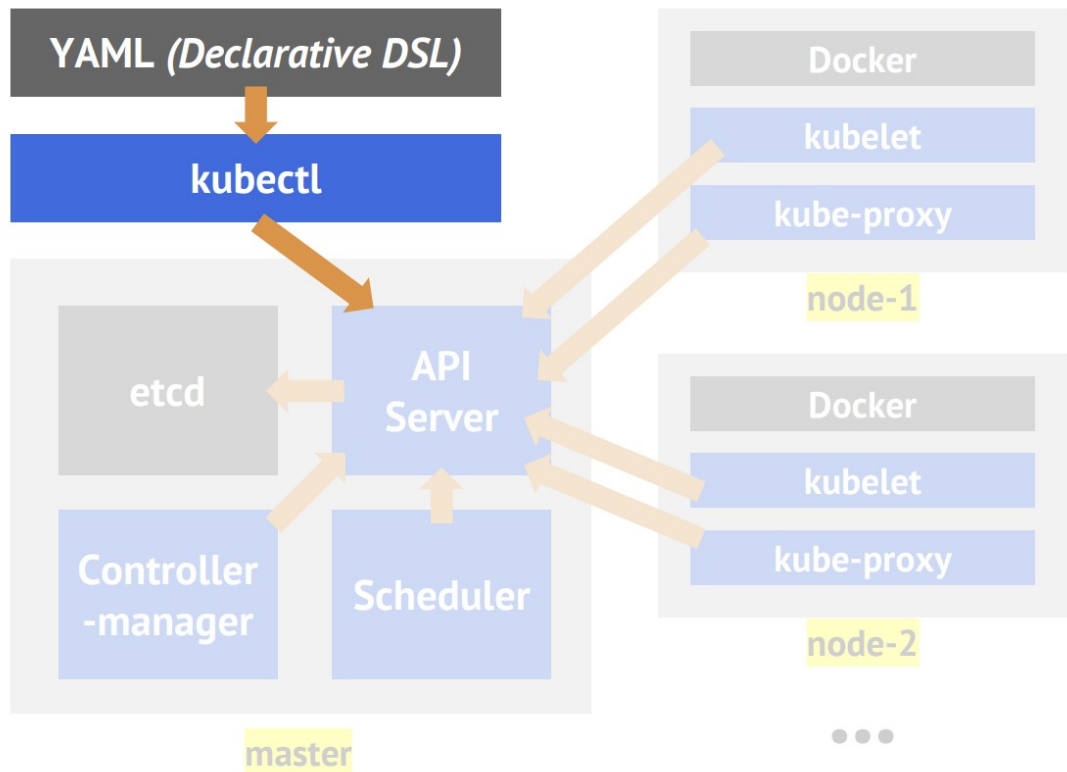
# Архитектура Kubernetes



# Архитектура Kubernetes

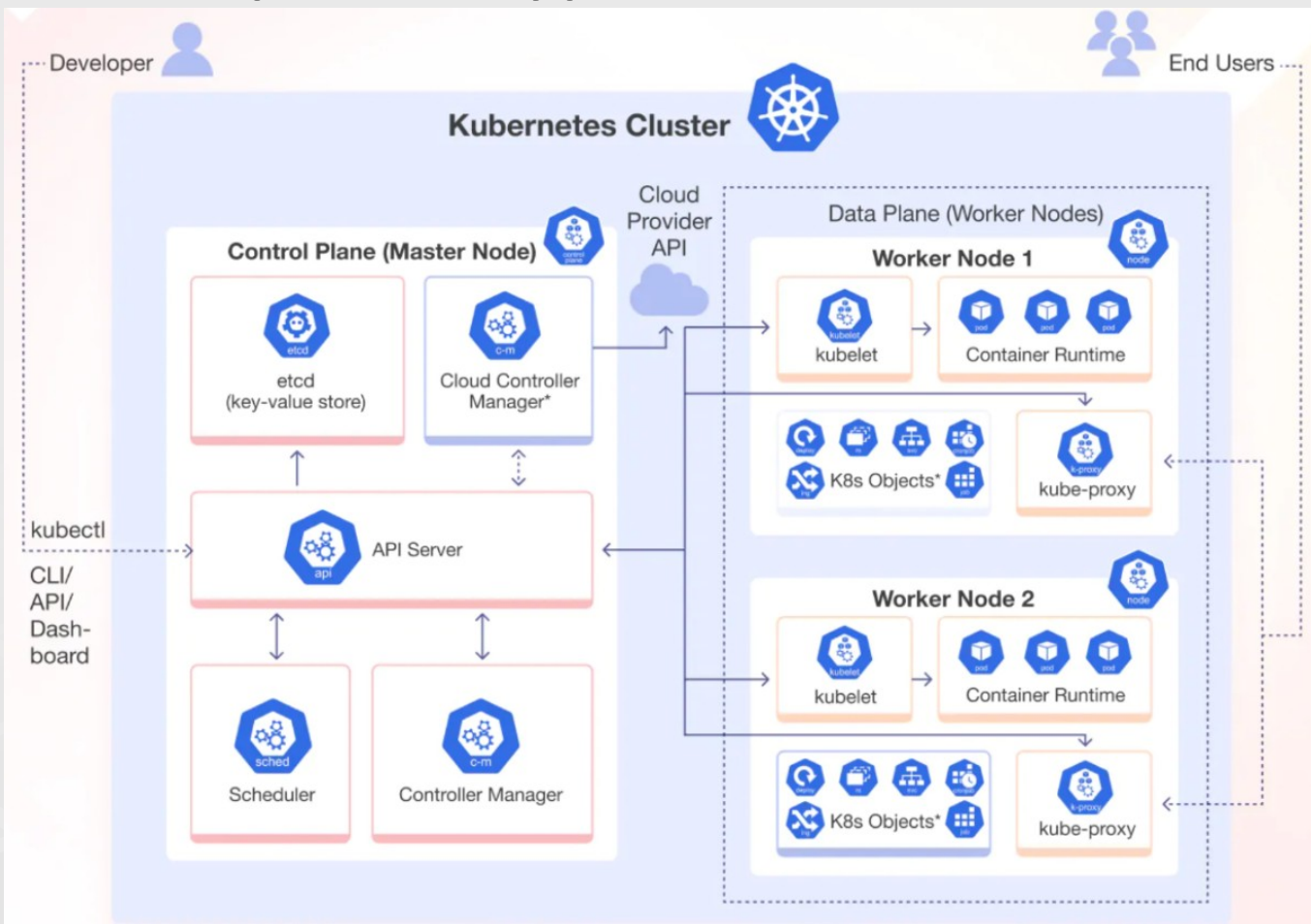


# Архитектура Kubernetes



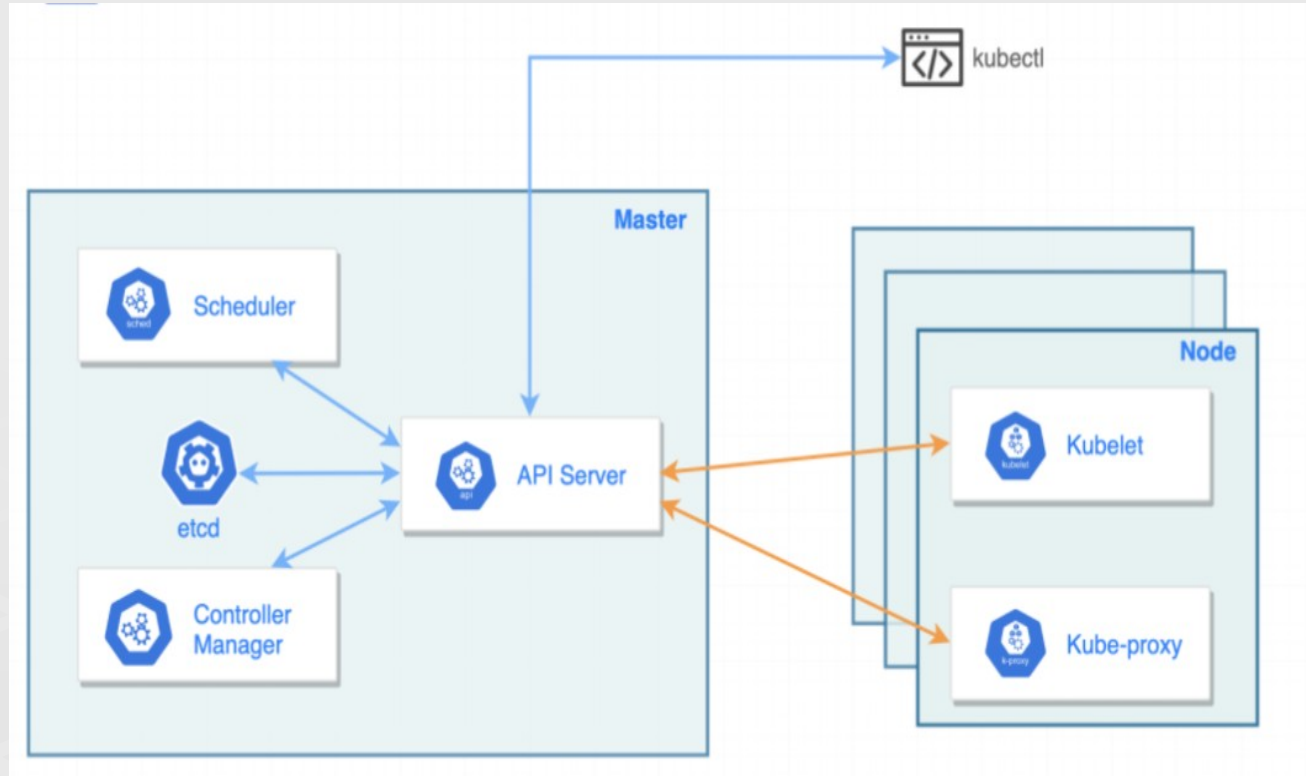


# Архитектура Kubernetes



# K8s - Control Plane

- Основная машина, управляющая узлами.
- Основная точка входа для всех административных задач.
- Он отвечает за оркестровку рабочих узлов.





# K8s - Control Plane



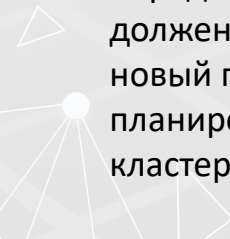
## etcd

Обеспечивает согласованность данных в кластере, используется для хранения информации о состоянии всех объектов Kubernetes

## kube-apiserver

Служит интерфейсом между всеми компонентами кластера и внешними клиентами, валидирует и конфигурирует данные API-объектов

## kube-scheduler



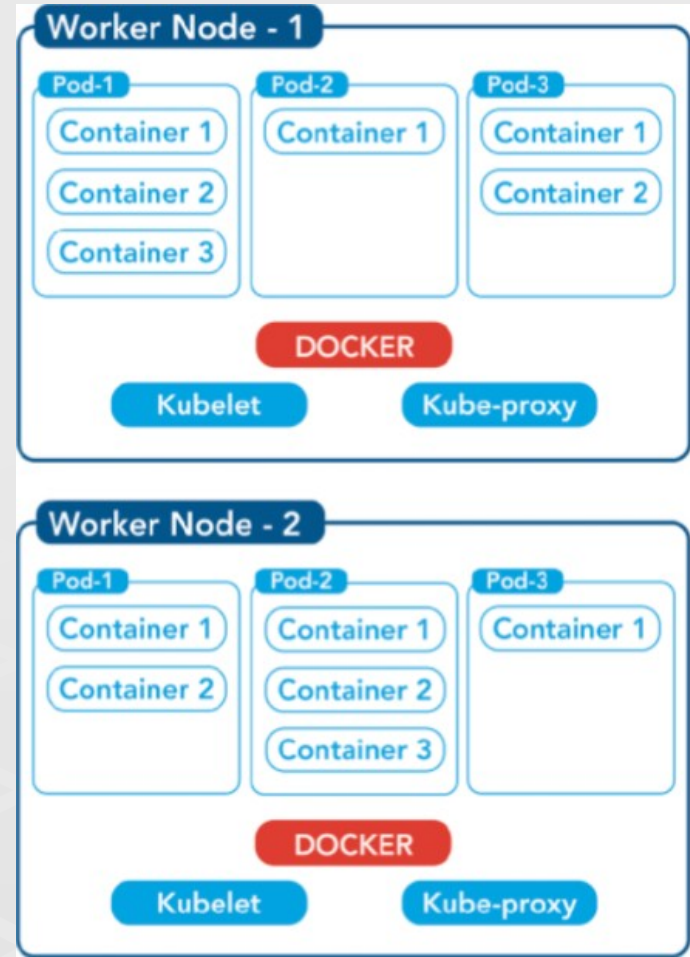
Анализирует ресурсы нод и требования подов, чтобы определить, на какой ноде должен быть развернут каждый новый под, ответственный за планирование подов на ноды кластера

## kube-controller-manager

Выполняет фоновые задачи управления, такие как поддержание желаемого состояния подов, реплик, конечных точек и других объектов в соответствии с заданными спецификациями

# K8s – Data Plane

- Worker Node выполняет запрошенные master задачи.
- Каждый узел контролируется главным узлом.
- Запускает pods (содержащие контейнеры внутри)
- Здесь запускается контейнерный runtime, который отвечает за загрузку образов и запуск контейнеров.





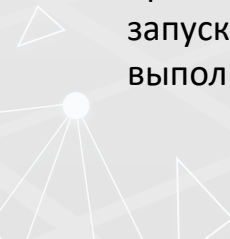
# K8s – Data Plane



## kubelet

Запуск подов, обеспечение их работы и поддержание их в заданном состоянии. Kubelet получает инструкции от API-сервера и управляет запуском контейнеров через контейнерный runtime

## Pods



Выполняют пользовательские приложения и сервисы. Поды запускаются на узлах и обеспечивают выполнение конкретных задач.

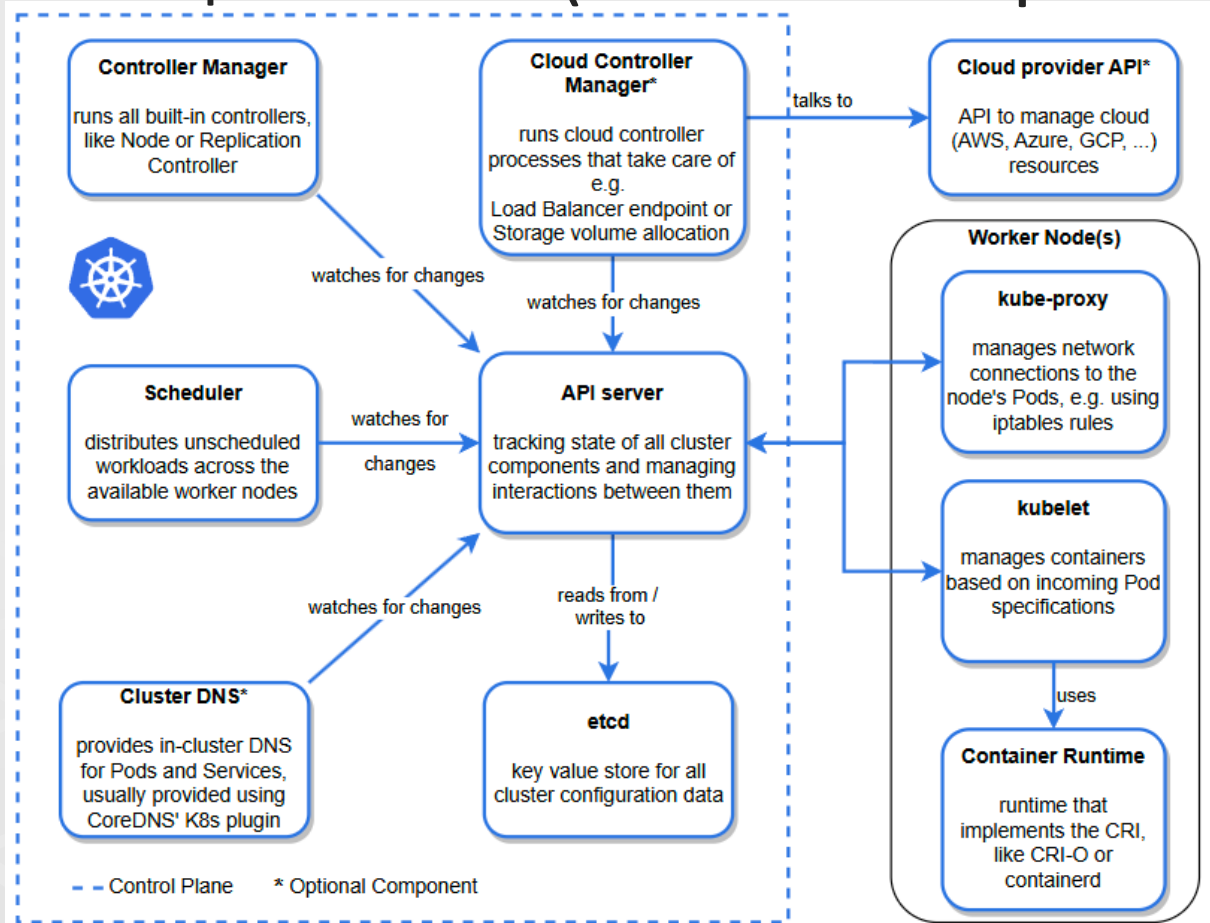
## Container Runtime

Запускает и управляет жизненным циклом контейнеров.

## kube-proxy

Обеспечивает сетевую маршрутизацию и балансировку нагрузки между сервисами и подами. Kube-proxy поддерживает связь между различными подами и сетевыми объектами, управляя правилами iptables или ipvs в зависимости от конфигурации.

# Общая схема (с комментариями)






05


# Декларативный подход Kubernetes



# Базовая основа k8s



Все в Kubernetes определяется с помощью текстовых файлов в формате YAML или JSON.



Платформа запускает образы OCI декларативным способом, настраивая все через эти файлы.

Этот подход используется также для настройки сетевых правил, аутентификации, авторизации (RBAC) и других аспектов

Изучив один синтаксис и структуру YAML/JSON, вы сможете управлять всеми аспектами Kubernetes



# Создание и управление контейнерами в Docker

```
FROM alpine:3.15.4
RUN apk add --no-cache mysql
ENTRYPOINT ["/usr/bin/mysqld"]
```

```
FROM python:3.7
WORKDIR /myapp
COPY src/requirements.txt ./
RUN pip install -r requirements.txt
COPY src /myapp
CMD [ "python", "mysql-custom-client.py" ]
```

```
docker build -t my-mysql-server .
```

```
docker run -d --name mysql-server my-mysql-server
```

# Создание и управление контейнерами в Kubernetes

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-pod
spec:
  containers:
  - name: mysql-server
    image: myregistry.com/mysql-server:v1.0
```

```
apiVersion: v1
kind: Pod
metadata:
  name: python-client-pod
spec:
  containers:
  - name: python-client
    image: myregistry.com/python-client:v1.0
    command: ['tail', '-f', '/dev/null']
```


```
kubectl apply -f mysql.yaml
```



# Декларативный подход k8s

Конфигурации  
сохраняются в YAML-  
файлах и описывают  
желаемое состояние  
системы.

Kubernetes  
автоматически  
поддерживает и  
стабилизирует  
состояние кластера на  
основе этих описаний

- ☐ Kubernetes сам решает, где и как запускать контейнеры.
  - ☐ Кластер поддерживает заданное состояние даже при сбоях.
  - ☐ Легко масштабировать и управлять большим количеством контейнеров.
- 

# Базовая терминология

1. CNI (Container Networking Interface) и CSI (Container Storage Interface) – сетевой интерфейс контейнеров и интерфейс хранилища для контейнеров соответственно. Позволяют подключать модули Pod (с контейнерами), работающие в Kubernetes, к сетям и хранилищам.
2. Контейнер (Container) – образ Docker или OCI (Open Container Initiative), который обычно запускает приложение.
3. Плоскость управления (Control plane) – мозг кластера Kubernetes, осуществляющий планирование контейнеров и управляющий всеми объектами Kubernetes (которые иногда называют мастер-объектами).
4. Набор демонов (DaemonSet) – аналог разворачивания (Deployment), но выполняется на каждом узле кластера.
5. Разворачивание (Deployment) – набор модулей, которыми управляет Kubernetes.
6. kubectl – инструмент командной строки для взаимодействия с панелью управления Kubernetes.
7. kubelet – агент Kubernetes, работающий на узлах кластера. Обеспечивает поддержку плоскости управления.
8. Узел (Node) – машина, на которой запущен процесс kubelet.
9. OCI (Open Container Initiative) – общий формат образа для создания выполняемых автономных приложений. Также называется образами Docker.
10. Pod (модуль) – объект Kubernetes, инкапсулирующий контейнер.

ДЗ будет в канале telegram.



А на сегодня все.

