Архитектура Kubernetes: основы распределенных систем

Kubernetes представляет собой сложную распределенную систему, которая демонстрирует основные принципы современной оркестрации контейнеров в больших масштабах. Платформа реализует фундаментальные концепции распределенных систем, включая координацию, согласованность и отказоустойчивость, благодаря тщательно разработанной архитектуре, которая разделяет задачи уровня управления и уровня выполнения.

Выполнение команды **kubectl apply -f deployment.yaml** инициирует скоординированную последовательность операций между несколькими распределенными компонентами, каждый из которых реализует определенные паттерны распределенных систем. Эта оркестрация работает через два отдельных архитектурных уровня: **уровень управления** (компоненты, принимающие решения) и **уровень данных** (компоненты выполнения), демонстрируя основной принцип разделения логики координации и выполнения рабочих нагрузок в распределенных системах.

⚙️ Архитектура Kubernetes

Основные компоненты инфраструктуры, составляющие основу распределенной системы Kubernetes.

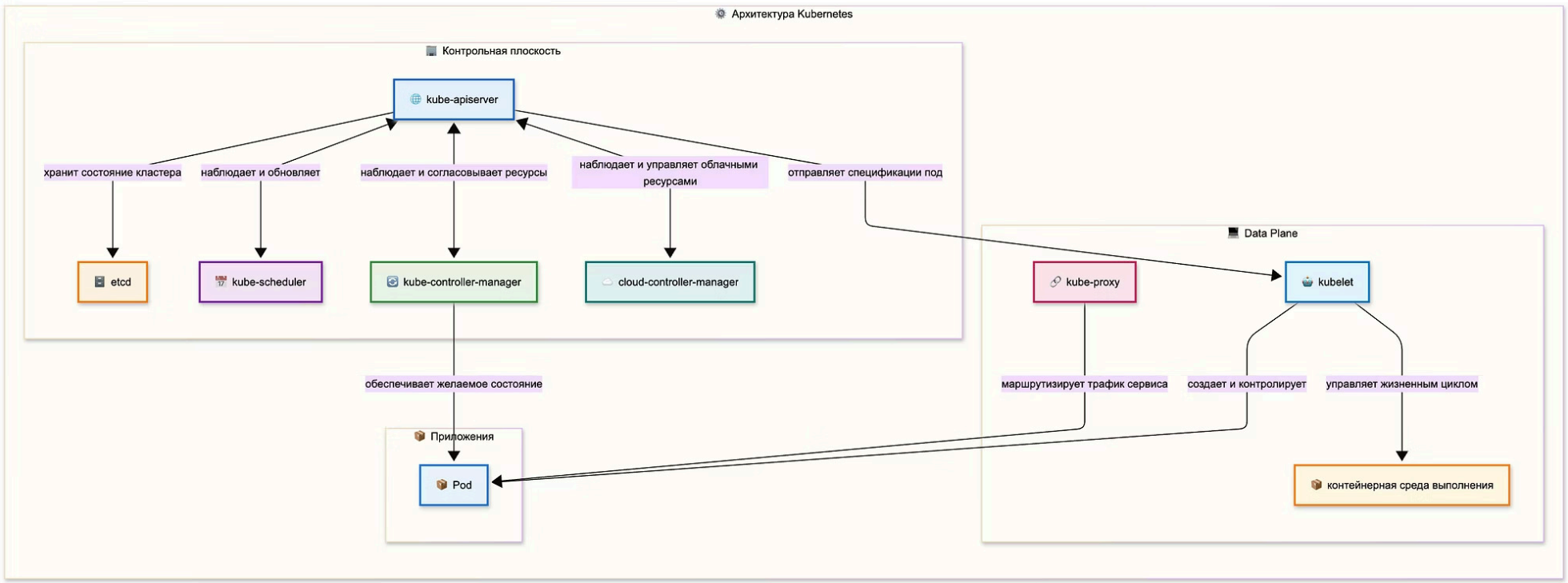
Kubernetes работает как распределенная система с четким разделением на **уровень управления** (принятие решений) и **уровень данных** (выполнение). Компоненты уровня управления работают на мастер-узлах и управляют состоянием кластера, а компоненты уровня данных работают на рабочих узлах и выполняют рабочие нагрузки. Такая архитектура обеспечивает горизонтальное масштабирование, отказоустойчивость и четкое разделение задач.

Уровень управления

API-сервер служит центральным шлюзом, etcd обеспечивает распределенное хранилище с высокой согласованностью, планировщик принимает решения о размещении, а контроллеры реализуют циклы согласования для поддержания желаемого состояния.

Уровень данных

Рабочие узлы запускают kubelet (агент узла), kube-proxy (сеть служб) и среду выполнения контейнеров для выполнения подсистем и предоставления вычислительных ресурсов.



Архитектура контрольной плоскости (Control Plane)

Контрольная плоскость (Control Plane) — это **мозг Kubernetes-кластера**, который управляет его состоянием, отслеживает события, принимает решения о размещении подов и обеспечивает согласованность между желаемым и фактическим состоянием системы.
Если рабочие узлы (Worker Nodes) выполняют контейнеры, то контрольная плоскость определяет *что*, *где* и *когда* выполнять.

Контрольная плоскость состоит из нескольких распределенных компонентов, которые совместно управляют состоянием кластера и оркестрацией:

kube-apiserver — центральный шлюз и точка координации

`kube-apiserver` — это основной интерфейс Kubernetes. Все команды (`kubectl`, панели управления, контроллеры, CI/CD системы) взаимодействуют с кластером через API Server.
Он выполняет функции:

- проверки и валидации запросов;
- аутентификации и авторизации пользователей;
- маршрутизации операций к соответствующим контроллерам;
- синхронизации изменений с хранилищем `etcd`.

Пример:
Когда вы применяете манифест `deployment.yaml`, API Server принимает объект `Deployment`, валидирует его и сохраняет в `etcd`, после чего контроллеры начинают действовать для достижения желаемого состояния.

etcd — распределённое хранилище состояния кластера

`etcd` — это надёжное распределённое хранилище ключ–значение, в котором Kubernetes хранит всё состояние кластера:
поды, конфигурации, роли, сервисы, секреты и т.д.

Основные особенности:

- обеспечивает **строгую консистентность** данных с помощью алгоритма **Raft**;
- используется как **источник истины (source of truth)** для всех компонентов;
- требует регулярных **резервных копий** для защиты от потери данных.

Пример:
Когда контроллер создаёт новый под, информация о нём записывается в `etcd`. Если кластер перезапустить, состояние восстановится из этого хранилища.

kube-scheduler — принятие решений о размещении

`scheduler` отвечает за выбор оптимального узла (Node) для размещения нового пода.
Он анализирует доступные ресурсы и ограничения, такие как:

- доступная память и CPU;
- теги и метки узлов (`nodeSelector`, `affinity`);
- приоритеты и taints/tolerations.

Пример:
Если создаётся под с требованием `cpu: 2` и `region=eu-west-1`, `scheduler` выберет подходящий узел, соответствующий этим условиям, и назначит на него под.

kube-controller-manager — согласование состояния ресурсов

`controller-manager` управляет множеством контроллеров, каждый из которых отвечает за определённый тип ресурсов.
Эти контроллеры работают по принципу **циклов согласования (reconciliation loops)** — постоянно сравнивают текущее состояние кластера с желаемым и предпринимают действия для устранения различий.

Основные контроллеры:

- **Deployment Controller** — поддерживает нужное количество реплик подов;
- **Node Controller** — отслеживает состояние узлов и помечает недоступные;
- **Service Account Controller** — управляет токенами и учетными записями.

Пример:
Если один из подов приложения аварийно завершился, контроллер автоматически создаст новый, чтобы поддержать нужное количество реплик.

cloud-controller-manager — интеграция с облачными платформами

`cloud-controller-manager` отвечает за взаимодействие Kubernetes с инфраструктурой облачных провайдеров.
Он позволяет использовать облачные ресурсы в составе кластера:

- автоматическое создание балансировщиков нагрузки (например, AWS ELB или GCP Load Balancer);
- присвоение и освобождение публичных IP-адресов;
- управление томами и снапшотами хранилищ.

Пример:
При создании ресурса `Service type=LoadBalancer` Kubernetes вызывает облачный API, и в AWS автоматически создаётся внешний балансировщик, связанный с подами.

Эти компоненты работают как независимые службы, координируя свои действия через сервер API и etcd для управления состоянием. Control Plane служит уровнем управления кластера, реализуя глобальные политики, распределение ресурсов и поддержание желаемого состояния. В производственных средах эти компоненты распределяются по нескольким **мастер-узлам**, чтобы обеспечить высокую доступность и сохранить логическое единство как единой системы управления.

> **Примечание об управляемых службах Kubernetes**: в облачных управляемых службах, таких как EKS, GKE и AKS, контрольная плоскость полностью абстрагирована от пользователей. Поставщики облачных услуг управляют мастер-узлами, компонентами контрольной плоскости и схемами их развертывания в фоновом режиме. Пользователи взаимодействуют только с конечной точкой API-сервера и управляют рабочими узлами, а поставщик облачных услуг автоматически обеспечивает доступность, обновления и масштабирование контрольной плоскости.

Мастер-узлы: инфраструктура контрольной плоскости

Мастер-узлы (или **узлы контрольной плоскости**) — это серверы, на которых развёрнуты ключевые компоненты управления Kubernetes: `kube-apiserver`, `etcd`, `kube-scheduler`, `kube-controller-manager` и при необходимости `cloud-controller-manager`. В отличие от рабочих узлов, выполняющих контейнерные приложения, мастер-узлы обеспечивают **управление, координацию и согласованность** всего кластера.

1. Один мастер (Single Master)

В небольших или тестовых кластерах может использоваться **один мастер-узел**. Эта конфигурация проста в настройке и подходит для сред разработки или демонстраций, однако создаёт **единую точку отказа** — при сбое мастер-ноды управление кластером становится недоступным, хотя уже работающие поды могут продолжать функционировать.

Пример: Мини-кластеры, такие как **Minikube**, **K3s** или **kind**, используют один мастер-узел для упрощённого развёртывания.

2. Мультимастер и высокая доступность (HA)

В производственных кластерах Kubernetes обычно разворачивается **3 или 5 мастер-узлов**, чтобы обеспечить **высокую доступность (High Availability)**. Нечётное количество узлов используется для обеспечения **кворума etcd** — большинству участников необходимо согласовать состояние перед записью изменений.

Особенности мультимастерной конфигурации:

- Компоненты `kube-apiserver` работают за балансировщиком нагрузки.
- Узлы `etcd` синхронизируются между собой с помощью протокола Raft.
- `scheduler` и `controller-manager` запускаются в активных и резервных копиях (активный экземпляр выбирается через механизм блокировок).

Пример: В кластере AWS EKS или GKE три мастер-узла автоматически распределяются по разным зонам доступности (AZ), что позволяет продолжать работу при сбое одной зоны.

3. Выделенная инфраструктура мастер-узлов

Мастер-узлы обычно имеют характеристики, отличные от рабочих нод:

- **больше CPU и памяти** — для обработки большого числа API-запросов и фоновых процессов;
- **быстрые SSD-диски** — для минимизации задержек `etcd`;
- **надежное сетевое подключение** — для постоянной синхронизации состояния между мастерами.

Пример конфигурации:

- 4 vCPU / 16 GB RAM — для `kube-apiserver` и контроллеров;
- NVMe SSD — для `etcd`;
- отдельные подсети и группы безопасности — для защиты административного трафика.

4. Защита от загрязнения (Tainting Masters)

Чтобы гарантировать стабильность работы контрольной плоскости, мастер-узлы **метятся специальными taints**, предотвращающими размещение на них пользовательских подов. Это защищает ресурсы контрольной плоскости от перегрузки.

Команда для просмотра taints:

```
kubectl get nodes -o wide
kubectl describe node <master-node>
```

Типичный taint:

```
node-role.kubernetes.io/control-plane=:NoSchedule
```

Тем не менее, в небольших средах (например, K3s) можно снять taint и разрешить выполнение подов на мастере.

5. Распределённая устойчивость

Мастер-узлы работают как **распределённая система управления**. Если один из них выходит из строя, оставшиеся продолжают обслуживать запросы, пока сохраняется кворум etcd.

Пример: В HA-кластере из трёх мастеров потеря одной ноды не влияет на доступность управления. API-серверы автоматически перенаправляют трафик, а контроллеры продолжают работу.

Итог

Мастер-узлы — это **сердце Kubernetes**, обеспечивающее:

- хранение состояния (`etcd`),
- принятие решений (`scheduler`, `controller-manager`),
- взаимодействие с пользователями и системами (`kube-apiserver`).

В производственной среде рекомендуется использовать **мультимастерную архитектуру** с нечетным числом узлов, выделенными ресурсами и защитой от пользовательских нагрузок. Такой подход гарантирует **отказоустойчивость, предсказуемость и стабильную работу** кластера при любых сбоях.

Сервер API

> `kube-apiserver` — это **центральный нервный узел Kubernetes**, реализующий фундаментальные принципы распределённых систем:
>
> - **бесстатусность**,
> - **событийно-ориентированное взаимодействие**,
> - **расширяемость через пользовательские ресурсы**,
> - **высокую доступность за счёт дублирования**.
>
> Он объединяет все компоненты кластера в единую согласованную систему и обеспечивает надёжное взаимодействие между пользователями, контроллерами и хранилищем состояния. Без `kube-apiserver` кластер перестаёт быть управляемым — он действительно является **сердцем Kubernetes**.

kube-apiserver: центральная точка координации Kubernetes

`kube-apiserver` — это **основной управляющий компонент Kubernetes**, через который проходят все взаимодействия с кластером. Он выступает в роли **центрального координационного пункта**, принимая REST API-запросы, выполняя аутентификацию и авторизацию, проверяя спецификации ресурсов и координируя сохранение состояния в `etcd`. Именно API-сервер превращает набор распределённых узлов в единую управляемую систему.

1. Роль и функции API-сервера

- Принимает все запросы от `kubectl`, контроллеров, операторов и внешних систем.
- Проверяет права доступа и валидность входных данных.
- Сохраняет состояние ресурсов (pods, services, deployments и т.д.) в `etcd`.
- Рассылает обновления о событиях другим компонентам через механизм наблюдения (*watch API*).

`kube-apiserver` — это **единая точка правды и синхронизации** для всего кластера. Без него остальные компоненты не могут получать актуальное состояние или координировать свои действия.

2. Развёртывание и высокая доступность

API-сервер обычно разворачивается в виде **статического пода** на каждом мастер-узле. Манифесты этих подов хранятся в `/etc/kubernetes/manifests/` и управляются напрямую `kubelet`, что позволяет узлам самостоятельно перезапускать компоненты контрольной плоскости без участия самого API-сервера.

Ключевые особенности развёртывания:

- Каждый API-сервер является **бесстатусным** — он не хранит собственные данные и может быть легко масштабирован по горизонтали.
- Все экземпляры API-сервера работают **за балансировщиком нагрузки** (например, AWS NLB или HAProxy).
- В управляемых сервисах (EKS, GKE, AKS) несколько API-серверов автоматически размещаются в разных **зонах доступности (AZ)** для обеспечения отказоустойчивости.

Преимущество: если один мастер-узел выходит из строя, другие продолжают обрабатывать запросы, обеспечивая **высокую доступность** и **непрерывное управление кластером**.

3. Статические поды — основа самовосстановления контрольной плоскости

Компоненты контрольной плоскости, включая API-сервер, запускаются как **статические поды**, определённые локально на каждом мастер-узле. `kubelet` следит за директорией `/etc/kubernetes/manifests/` и автоматически:

- создаёт поды на основе обнаруженных манифестов;
- перезапускает их при сбоях;
- удаляет поды, если удалён соответствующий файл манифеста.

Отличие от обычных подов: Статические поды **не управляются API-сервером** и не могут быть удалены через `kubectl delete pod`. Это обеспечивает автономность и возможность **самозапуска контрольной плоскости** даже при полной недоступности API.

4. API-сервер как шлюз и точка безопасности

API-сервер — это **единая точка входа** в кластер, которая обеспечивает:

- **аутентификацию** (через токены, сертификаты, OIDC и т.д.);
- **авторизацию** (через RBAC, ABAC или webhook-плагины);
- **валидацию** запросов и схем ресурсов;
- **аудит** всех операций.

Такой шлюзовый подход гарантирует, что все изменения в кластере происходят централизованно, с соблюдением политик безопасности.

5. Бесстатусный дизайн и согласованность

Сам API-сервер **не хранит состояние** — все данные сохраняются в `etcd`. Это даёт ключевые преимущества:

- возможность **горизонтального масштабирования** (добавление новых серверов API без миграции данных);
- **высокая отказоустойчивость** — сбой одного экземпляра не влияет на целостность данных;
- **чистая архитектура**, где сервер отвечает за валидацию и маршрутизацию, а не за хранение.

6. Событийно-ориентированная архитектура

API-сервер работает в режиме **публикации/подписки (pub/sub)**. При изменении состояния ресурсов он отправляет **события обновлений** всем подписанным клиентам (контроллерам, kubelet и др.) через **watch API**. Клиенты устанавливают постоянное соединение (HTTP long polling) и получают изменения в реальном времени.

Это позволяет системе быть **реактивной и слабо связанной** — каждый компонент реагирует на события, не блокируя других.

7. Расширяемость через CRD (Custom Resource Definitions)

API-сервер поддерживает **динамическое расширение модели данных**. Пользователи могут определять собственные типы ресурсов с помощью CRD.

При регистрации CRD:

- сервер API автоматически создаёт REST-эндпоинты для нового ресурса;
- выполняет проверку данных по пользовательской схеме (OpenAPI/JSON Schema);
- сохраняет эти данные в `etcd` наряду со стандартными ресурсами Kubernetes.

Это делает Kubernetes **платформой для построения платформ**, где можно создавать свои собственные абстракции — например, `BackupPolicy`, `DataPipeline`, `MachineLearningJob` — и управлять ими теми же механизмами, что и встроенными объектами.

etcd: распределённое хранилище состояния кластера

> `etcd` — это **фундаментальная распределённая база данных**, обеспечивающая:
>
> - **сильную согласованность** между мастер-узлами;
> - **устойчивость к сбоям** благодаря Raft и кворуму;
> - **точное и атомарное хранение состояния** кластера.
>
> Без etcd Kubernetes не смог бы гарантировать согласованное управление ресурсами. Он делает возможным то, что Kubernetes называет **декларативным управлением** — вы описываете желаемое состояние, а etcd надёжно сохраняет его и обеспечивает единообразное восприятие системой.

`etcd` — это **сердце состояния Kubernetes**, надёжное **распределённое хранилище ключ–значение**, в котором сохраняются все данные, описывающие текущее и желаемое состояние кластера. Каждый объект Kubernetes — от `Pod` и `Service` до `ConfigMap` и `Secret` — существует в виде записи в `etcd`. Именно это хранилище делает возможным согласованное функционирование всех компонентов контрольной плоскости.

1. Роль и значение etcd

`etcd` выполняет роль **единого источника истины (source of truth)** для всей системы. Любая операция, выполняемая в Kubernetes, в конечном итоге сводится к чтению или записи данных в `etcd`:

- Когда пользователь создаёт `Pod`, `Deployment` или `Service`, `kube-apiserver` записывает эти объекты в `etcd`.
- Когда контроллеры и планировщики принимают решения, они считывают текущее состояние из `etcd` и обновляют его при изменениях.

Таким образом, `etcd` гарантирует, что **все участники системы видят одно и то же состояние**, даже в распределённой среде.

2. Развёртывание и высокая доступность

В типичной производственной конфигурации `etcd` работает как **кластер из нечётного числа узлов** — обычно 3 или 5. Это необходимо для поддержания **кворума** в алгоритме консенсуса Raft: большинство узлов должно подтвердить запись, чтобы она считалась зафиксированной.

Ключевые особенности развертывания:

- Каждый экземпляр `etcd` разворачивается как **статический под** на мастер-узле (манифесты обычно находятся в `/etc/kubernetes/manifests/etcd.yaml`).
- Каждый узел `etcd` требует **постоянного хранилища** (SSD-диски с низкой задержкой для журналов Raft).
- Узлы рекомендуется **распределять по зонам доступности (AZ)** для устойчивости к сбоям.
- В управляемых сервисах (EKS, GKE, AKS) etcd развёртывается автоматически, включая резервное копирование и восстановление.

Практика: Некоторые крупные организации (например, Netflix, Shopify) разворачивают `etcd` на **отдельных выделенных машинах**, чтобы минимизировать влияние на производительность API-сервера.

3. Протокол консенсуса Raft

Основой работы `etcd` является **алгоритм Raft**, обеспечивающий согласованность между узлами в распределённой среде. Raft решает ключевую задачу: **как гарантировать, что все узлы имеют одинаковое состояние**, даже если происходят сетевые сбои или временные отключения.

Как это работает:

- Один узел выбирается как **лидер**, принимающий все операции записи.
- Другие узлы действуют как **фолловеры** и синхронизируют журнал лидера.
- Запись считается успешной, только если её подтвердило **большинство узлов**.

Это гарантирует, что даже при сбое части кластера данные останутся **консистентными и устойчивыми к разделению сети (split-brain)**.

4. Сильная согласованность и линеаризуемость

`etcd` предоставляет **сильную согласованность (strong consistency)**:

- Каждая операция записи выполняется **атомарно и последовательно**;
- После подтверждения она немедленно становится видимой всем последующим операциям чтения;
- При этом чтение может быть **линейно согласованным (linearizable)** — отражать самое последнее подтверждённое состояние.

Зачем это важно: Kubernetes — система, в которой решения зависят от текущего состояния (например, планирование подов или выделение ресурсов). Если разные компоненты видят разные состояния, возможны конфликты: например, два планировщика могут разместить под на одном и том же узле. Линеаризуемая модель etcd гарантирует, что такого не произойдёт.

5. Примеры ключей и данных в etcd

Все данные в `etcd` представлены в виде **иерархии ключей** (как файловая система):

```
/registry/pods/default/nginx-pod
/registry/deployments/production/web-deployment
/registry/secrets/default/db-credentials
```

Каждое изменение записывается как новая ревизия, что позволяет API-серверу отслеживать версионность и передавать обновления компонентам через механизмы watch.

6. Резервное копирование и восстановление

Так как `etcd` хранит всё состояние кластера, его **резервное копирование** — критически важная операция. Kubernetes CLI предоставляет встроенные инструменты:

```
ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-snapshot.db \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
```

Эти снимки позволяют полностью восстановить кластер при потере мастер-узлов.

7. Практические соображения

- **Сеть:** задержка между узлами etcd должна быть минимальной — рекомендуется размещать их в одной сети с низким RTT.
- **Хранилище:** используйте SSD-диски и избегайте сетевых томов с высокой латентностью.
- **Мониторинг:** следите за метриками Raft (`leader_changes_seen_total`, `apply_entries_total`, `fsync_duration_seconds`).
- **Изоляция:** не размещайте etcd вместе с рабочими нагрузками — это повышает стабильность.

kube-scheduler: распределение ресурсов и оптимизация размещения

`kube-scheduler` — это компонент контрольной плоскости, отвечающий за **принятие решений о размещении подов** в кластере Kubernetes. Он отслеживает новые поды, которые ещё не назначены на узел, и подбирает для них **оптимальное место** с учётом доступных ресурсов, ограничений и политик. После выбора узла планировщик обновляет объект пода в `etcd`, а `kubelet` целевого узла обнаруживает это изменение и запускает контейнеры.

Основная роль планировщика

Планировщик отвечает не за запуск, а за **распределение** — то есть за выбор, *где* будут запущены контейнеры. Процесс планирования включает три ключевых шага:

- **Filtering (фильтрация)** — исключает узлы, не удовлетворяющие базовым требованиям (ресурсы, taints, политика размещения).
- **Scoring (оценка)** — оценивает оставшиеся узлы по множеству факторов, чтобы выбрать наилучший.
- **Binding (привязка)** — записывает результат в `etcd`, назначая под на выбранный узел.

Развёртывание и высокая доступность

Планировщик разворачивается как **статический под**, определённый в:

```
/etc/kubernetes/manifests/kube-scheduler.yaml
```

Для отказоустойчивости могут работать **несколько экземпляров планировщика**, но активен всегда только один. Это обеспечивается через **leader election** — механизм выбора лидера с помощью объекта аренды в `etcd`.

Как это работает:

- Все экземпляры конкурируют за получение аренды (`Lease`).
- Экземпляр, который первый её получает, становится лидером.
- Остальные находятся в режиме standby.
- Если лидер не обновил аренду (например, при сбое), другой экземпляр автоматически становится новым лидером.

Такой механизм гарантирует, что **планирование выполняется только одним процессом**, предотвращая конфликты и сценарии split-brain.

Плагиновая архитектура

Современный `kube-scheduler` построен на **plugin-based framework**, что позволяет легко расширять и настраивать его поведение. Он разделяет процесс планирования на два этапа — *filtering* и *scoring*, каждый из которых реализуется отдельным набором плагинов.

Примеры встроенных плагинов:

- `NodeResourcesFit` — проверяет, хватает ли узлу CPU, памяти или GPU.
- `NodeAffinity` — учитывает предпочтения по размещению (например, «в зоне eu-central-1a»).
- `PodTopologySpread` — распределяет поды между зонами доступности.
- `InterPodAffinity` — размещает поды рядом с другими (например, рядом с базой данных).

> 💡 *Affinity (аффинность)* — это механизм, определяющий **предпочтения или ограничения размещения** подов на узлах, например «размещай вместе с подом сервиса X» или «избегай зоны Y».

Благодаря модульности можно реализовать собственные плагины — например, учитывать стоимость узлов, энергопотребление или пользовательские теги.

Многомерная оптимизация

Планировщик Kubernetes решает **многомерную задачу оптимизации**, учитывая одновременно:

- ресурсы (CPU, RAM, GPU, дисковая производительность);
- affinity и anti-affinity правила;
- taints и tolerations;
- приоритеты и политики preemption;
- пользовательские ограничения и QoS-классы.

Такой подход обеспечивает **баланс между производительностью, доступностью и эффективностью использования ресурсов**, что особенно важно в масштабных SaaS-системах.

Двухэтапное планирование

Процесс разделён на два шага:

1. **Scheduling Decision** — выбор узла на основе фильтров и приоритетов.
2. **Binding** — фиксация решения в `etcd` и уведомление `kubelet`.

Это разделение даёт:

- поддержку **preemption** (вытеснение менее приоритетных подов);
- гибкость при **перепланировании**;
- возможность интеграции с внешними системами планирования (например, Volcano, Kueue).

Примеры использования

- **Веб-приложения:** равномерное распределение подов между зонами для отказоустойчивости.
- **ML-задачи:** выбор узлов с GPU и высокой памятью.
- **CI/CD пайплайны:** приоритизация узлов с наименьшей нагрузкой.

> `kube-scheduler` — это мозг Kubernetes, который превращает кластер в *самоорганизующуюся систему*. Он сочетает:
>
> - **распределённую устойчивость** через leader election,
> - **plugin-based архитектуру** для гибкости,
> - **многомерную оптимизацию** для эффективного использования ресурсов.
>
> Благодаря этому Kubernetes способен масштабироваться от десятков до тысяч узлов, поддерживая стабильность и равномерную загрузку инфраструктуры.

Контроллер-менеджер: шаблоны согласования

`kube-controller-manager` — это компонент контрольной плоскости, который **реализует декларативную модель управления состоянием Kubernetes**. Он запускает множество контроллеров, которые непрерывно **согласовывают текущее состояние кластера с желаемым**, как оно описано в `etcd`. Контроллеры следят за объектами (например, Deployment, ReplicaSet, Node, Service) и автоматически выполняют операции, необходимые для поддержания соответствия между желаемым и фактическим состоянием.

Основная роль контроллер-менеджера

Kubernetes работает по принципу *declarative state reconciliation* — вы не говорите системе, *что делать*, а описываете, *чего вы хотите достичь*. Контроллеры превращают это описание в реальность.

Пример:

- Вы создаёте `Deployment`, в котором указано `replicas: 3`.
- Контроллер развертывания проверяет, сколько подов реально запущено.
- Если только 2 — он создаёт третий; если 4 — удаляет один.

Эти процессы работают постоянно, в фоновом режиме, обеспечивая устойчивость и самовосстановление системы.

Развёртывание и высокая доступность

`kube-controller-manager` запускается как **статический под** на каждом мастер-узле:

```
/etc/kubernetes/manifests/kube-controller-manager.yaml
```

В отличие от отдельных контроллеров, он представляет собой **единый бинарный процесс**, внутри которого одновременно работают десятки контроллеров:

- **Deployment Controller** — управляет обновлениями и масштабированием реплик;
- **ReplicaSet Controller** — следит за количеством подов;
- **Node Controller** — отслеживает состояние узлов и инициирует пересоздание подов при сбоях;
- **Service Controller** — управляет балансировщиками нагрузки и обновлением эндпоинтов;
- **Namespace, Endpoint, Job, CronJob, PV Controller** и многие другие.

Как и планировщик, контроллер-менеджер использует механизм **leader election**, чтобы одновременно активен был только один экземпляр. Резервные экземпляры остаются в режиме standby и могут мгновенно взять управление при сбое лидера.

Декларативное управление состоянием

Контроллеры не выполняют команды напрямую — они реализуют **цикл согласования (reconciliation loop)**:

1. **Наблюдение (Watch)** — контроллер получает событие об изменении ресурса из API-сервера.
2. **Сравнение** — сверяет текущее состояние с желаемым, хранящимся в `etcd`.
3. **Действие** — предпринимает шаги для устранения расхождений (создаёт, обновляет, удаляет объекты).

> 💡 Это ключевая особенность Kubernetes: вы задаёте цель, а система *сама* обеспечивает её достижение, даже при частичных сбоях или асинхронных изменениях.

Благодаря этому подходу Kubernetes обладает свойством **самовосстановления** — если узел выходит из строя, контроллер пересоздаёт поды на других узлах без ручного вмешательства.

Шаблон наблюдения (Watch Pattern)

Контроллеры не опрашивают сервер API, а **реагируют на события** через механизм *watch API*. Это означает:

- Минимальная нагрузка на сеть и API-сервер;
- Реакция почти в реальном времени на изменения состояния;
- Возможность обработки большого числа событий асинхронно.

Пример: Когда новый `Pod` создаётся, контроллер ReplicaSet получает уведомление и мгновенно проверяет, нужно ли создать ещё один.

Независимость и слабая связность контроллеров

Каждый контроллер отвечает за **один тип ресурса** и работает автономно. Это обеспечивает устойчивость и масштабируемость:

- Сбой одного контроллера (например, Node Controller) не влияет на работу других.
- Контроллеры не вызывают друг друга напрямую — они **взаимодействуют только через API-сервер и etcd**, что исключает каскадные ошибки.

Такое разделение позволяет обновлять, расширять и отлаживать контроллеры независимо, сохраняя стабильность системы.

> `kube-controller-manager` — это «нервная система» Kubernetes, реализующая его декларативную природу. Он:
>
> - непрерывно **согласовывает** текущее состояние с желаемым;
> - обеспечивает **самовосстановление** при сбоях;
> - использует **watch-паттерн** для реактивной работы;
> - поддерживает **десятки специализированных контроллеров**, работающих независимо.
>
> Благодаря этой архитектуре Kubernetes способен поддерживать стабильность и согласованность даже в динамичных, распределённых кластерах с тысячами ресурсов.

cloud-controller-manager: облачная абстракция и интеграция инфраструктуры

`cloud-controller-manager` отвечает за интеграцию Kubernetes с API конкретного облачного провайдера, обеспечивая единое поведение при работе с ресурсами, зависящими от инфраструктуры. Он выступает в роли **адаптера**, который переводит действия Kubernetes в операции, понятные облаку — будь то AWS, GCP, Azure или OpenStack. Это позволяет остальной части контрольной плоскости работать в **унифицированной модели**, независимо от особенностей конкретной платформы.

Роль и задачи

Главная цель cloud-controller-manager — **отделить логику управления кластером от логики взаимодействия с инфраструктурой облака**. Без него Kubernetes должен был бы напрямую обращаться к API провайдера, что усложнило бы переносимость и сопровождение.

Он выполняет четыре ключевые функции:

- **Node Controller** — управляет жизненным циклом узлов, проверяя их наличие в облаке. Если экземпляр VM был удалён вручную, контроллер удаляет соответствующий объект узла из кластера.
- **Route Controller** — создаёт и обновляет сетевые маршруты между узлами, обеспечивая связность подов в облачных сетях (например, в AWS VPC или Azure VNet).
- **Service Controller** — управляет балансировщиками нагрузки для сервисов типа `LoadBalancer`. При создании сервиса контроллер автоматически выделяет облачный балансировщик (ALB/NLB в AWS, Azure LB, GCP LB), настраивает правила и группы безопасности.
- **Volume Controller** — отвечает за динамическое выделение и подключение постоянных томов (например, EBS, Persistent Disk, Managed Disk).

> 💡 Эти функции могут быть реализованы как отдельные контроллеры или объединены в один процесс, в зависимости от поставщика и конфигурации.

Развёртывание и управление

В отличие от других компонентов контрольной плоскости, `cloud-controller-manager` **не запускается как статический под**, а работает как **Deployment** в пространстве имён `kube-system`. Такое решение обеспечивает:

- гибкое обновление без остановки кластера;
- возможность настройки и замены драйвера для конкретного провайдера;
- контроль версий и управление через стандартные Kubernetes-механизмы.

В управляемых сервисах (например, AWS EKS, GKE, AKS) `cloud-controller-manager` обычно встроен в контрольную плоскость и управляется самим облачным провайдером. В самоуправляемых кластерах он разворачивается вручную или с помощью плагинов Cloud Provider Interface (CPI), предоставляющих конфигурацию и сетевой доступ к облачным API.

Пример: На AWS EKS при создании `Service` типа `LoadBalancer` контроллер автоматически вызывает AWS API для создания Application Load Balancer (ALB), настраивает Security Groups, добавляет Target Groups и следит за их состоянием. Когда сервис удаляется, контроллер корректно очищает все связанные облачные ресурсы.

Механизм адаптации (Cloud Provider Interface)

`cloud-controller-manager` использует **интерфейс Cloud Provider Interface (CPI)** — это набор контрактов, которые реализуются для каждого облака. Каждый драйвер CPI содержит модули для управления:

- экземплярами узлов (Instances);
- маршрутами (Routes);
- балансировщиками (LoadBalancers);
- томами хранения (Volumes).

Такой подход обеспечивает **расширяемость и переносимость**: новый облачный провайдер может быть интегрирован в Kubernetes без изменения ядра, просто реализуя стандартный интерфейс.

Конечная согласованность с внешними системами

В отличие от строго согласованного состояния внутри Kubernetes (через `etcd`), взаимодействие с внешними API облака подчиняется **модели eventual consistency**. Например, создание балансировщика или выделение тома может занять десятки секунд, а API может возвращать временные ошибки.

`cloud-controller-manager` решает эту проблему через постоянное **согласование состояния**:

- если ресурс в облаке отсутствует, он будет создан;
- если конфигурация изменилась вручную, контроллер её скорректирует;
- если облако возвращает ошибку, контроллер повторит операцию с экспоненциальной задержкой.

Это гарантирует, что **состояние кластера Kubernetes и облачной инфраструктуры остаются синхронизированными**.

> `cloud-controller-manager` — это «переводчик» между Kubernetes и облачным провайдером. Он обеспечивает:
>
> - **автоматическое управление инфраструктурой** (узлы, маршруты, тома, балансировщики);
> - **унифицированную модель ресурсов**, независимую от платформы;
> - **постоянное согласование состояния** между Kubernetes и облаком;
> - **гибкость развёртывания** через интерфейс Cloud Provider Interface.
>
> Благодаря этому Kubernetes может одинаково эффективно работать как в AWS, GCP, Azure, так и в частных облаках — сохраняя единый подход к управлению инфраструктурой.

Архитектура плоскости данных (Data Plane)

Плоскость данных (Data Plane) отвечает за фактическое выполнение рабочих нагрузок Kubernetes — контейнеров и подов, распределённых по рабочим узлам (worker nodes).
В то время как **контрольная плоскость (Control Plane)** принимает решения (где, когда и как запускать), плоскость данных **реализует эти решения**, обеспечивая выполнение, сетевое взаимодействие, хранение и мониторинг.

Это разделение ролей — **основополагающий принцип проектирования распределённых систем**, позволяющий системе масштабироваться, быть устойчивой к сбоям и управляемой централизованно.

Рабочие узлы: инфраструктура выполнения приложений

Рабочие узлы (Worker Nodes) — это основа вычислительной мощности Kubernetes. Именно на них запускаются контейнеры и поды, реализующие бизнес-логику приложений. Если контрольная плоскость принимает решения, то рабочие узлы — **выполняют их**, превращая декларации в реальную работу.

Роль и назначение

Рабочие узлы образуют **уровень выполнения (Execution Layer)** в архитектуре Kubernetes. Каждый узел является полноценным сервером (виртуальным или физическим), который предоставляет ресурсы:

- **ЦП и память** — для выполнения контейнеров;
- **хранилище** — для локальных или подключаемых томов;
- **сеть** — для коммуникации между подами и сервисами.

Мастер-узлы управляют кластером, а **рабочие узлы — выполняют приложения**, изолируя задачи, балансируя нагрузку и обеспечивая доступность.

Основные функции рабочего узла

1. **Размещение приложений** Каждый под, описанный в манифесте или созданный через контроллеры (Deployment, StatefulSet и т.д.), в конечном итоге выполняется на одном из рабочих узлов. Планировщик (`kube-scheduler`) выбирает подходящий узел, а локальные компоненты (`kubelet`, `container runtime`) запускают контейнеры.
2. **Распределение ресурсов** Kubernetes отслеживает использование ресурсов каждого узла и обеспечивает, чтобы поды не превышали заданные лимиты. Это позволяет поддерживать стабильность и предотвращать конкуренцию за ресурсы между приложениями.
3. **Гибкость масштабирования** Новые узлы можно добавлять или удалять динамически — это обеспечивает горизонтальное масштабирование. В облачных средах (AWS, GCP, Azure) можно использовать **Node Auto-Scaler**, который автоматически изменяет количество узлов в пуле в зависимости от нагрузки.
4. **Гетерогенность узлов** Не все узлы одинаковы:
   - **Compute-optimized** — для высоконагруженных вычислений;
   - **Memory-optimized** — для in-memory баз данных и кэширования;
   - **GPU-узлы** — для задач машинного обучения и рендеринга;
   - **Storage-optimized** — для систем, требующих высокой пропускной способности ввода-вывода. Планировщик Kubernetes учитывает эти различия при выборе узлов для подов.

Основные компоненты рабочего узла

Каждый узел включает три ключевых компонента:

- **kubelet** — агент, который получает задания от API-сервера, запускает поды и следит за их состоянием.
- **kube-proxy** — управляет сетевыми правилами, обеспечивая маршрутизацию трафика между подами и сервисами.
- **Container Runtime** — низкоуровневый движок, который запускает контейнеры (например, `containerd`, `CRI-O`, `Docker`).

Вместе они образуют **самодостаточную исполнительную среду**, способную поддерживать жизненный цикл контейнеров даже при временной потере связи с контрольной плоскостью.

Автоматическое масштабирование и отказоустойчивость

В облачных средах Kubernetes использует **автоматическое масштабирование узлов (Cluster Autoscaler)**:

- когда поды не могут быть размещены из-за нехватки ресурсов — создаются новые узлы;
- когда узлы простаивают — они автоматически удаляются для экономии затрат.

Кроме того, при сбое узла его поды автоматически **перезапускаются на других узлах**, что обеспечивает высокую доступность приложений.

> Рабочие узлы — это основа вычислительной мощности Kubernetes. Каждый из них:
>
> - запускает поды и контейнеры,
> - управляет локальными ресурсами,
> - обеспечивает сетевое взаимодействие и хранилище,
> - участвует в автоматическом масштабировании и балансировке.
>
> В совокупности узлы образуют **динамическую вычислительную платформу**, которая автоматически растёт, восстанавливается и адаптируется к нагрузке — основа эластичности Kubernetes.

Kubelet: архитектура агента узла

Kubelet — это ключевой агент Kubernetes, который работает на каждом рабочем узле и отвечает за выполнение контейнеров, поддержание их состояния и синхронизацию с контрольной плоскостью. Он выступает как **локальный исполнитель желаемого состояния**, гарантируя, что всё, что описано в спецификациях подов, действительно запущено и работает на узле.

Роль и назначение

Kubelet — это связующее звено между **контрольной плоскостью** и **плоскостью данных**. Он не принимает решения о планировании — этим занимается `kube-scheduler` — но реализует их, обеспечивая выполнение задач, назначенных конкретному узлу.

Основные функции:

- получает из API-сервера декларативные спецификации подов (манифесты);
- запускает контейнеры с помощью среды выполнения (`containerd`, `CRI-O`, `Docker` и др.);
- следит за состоянием подов, контейнеров и узла;
- сообщает API-серверу о метриках, событиях и текущем состоянии.

Таким образом, kubelet превращает «обещания» контрольной плоскости в **фактическое состояние** инфраструктуры.

Развёртывание и жизненный цикл

Kubelet работает как **системная служба** (демон `systemd`), запускаемая непосредственно в операционной системе узла — **не как контейнер и не как под**. Такое размещение гарантирует, что kubelet сможет:

- взаимодействовать с контейнерным runtime через сокет (`/var/run/containerd/containerd.sock` и т.п.);
- управлять файловой системой хоста;
- настраивать сетевые интерфейсы и подключать тома.

Обычно установка kubelet выполняется:

- вручную при конфигурации узлов (`kubeadm`, `kops`, `eksctl` и др.);
- или автоматически при инициализации узлов в управляемых сервисах (EKS, GKE, AKS).

После запуска kubelet:

- регистрируется в API-сервере;
- публикует состояние узла (capacity, allocatable, taints, conditions);
- начинает синхронизировать состояние подов.

Архитектура на основе pull

Kubelet реализует **pull-модель синхронизации** — он *запрашивает* желаемое состояние у API-сервера, а не *получает команды* от него напрямую. Это фундаментальное свойство архитектуры Kubernetes.

Преимущества pull-подхода:

- устойчивость к временным сетевым сбоям (узел продолжает работать автономно);
- упрощённая безопасность (инициатор соединения — узел, а не сервер);
- изоляция и ограничение влияния сбоев контрольной плоскости.

Таким образом, даже если API-сервер временно недоступен, kubelet продолжает обслуживать уже запущенные контейнеры.

Управление локальным состоянием

Kubelet поддерживает **локальный кеш состояния**, который описывает текущие поды, контейнеры, точки монтирования и сетевые настройки. Этот кеш используется для сравнения с **желаемым состоянием**, полученным из API-сервера. Если обнаруживается несоответствие, kubelet выполняет корректирующие действия:

- создаёт отсутствующие контейнеры;
- перезапускает завершившиеся;
- удаляет лишние;
- восстанавливает монтирование или сетевую конфигурацию.

Такой механизм обеспечивает **декларативное самовосстановление** на уровне узла.

Взаимодействие с другими компонентами

- **API-сервер:** источник желаемого состояния и приёмник метрик узла.
- **Container Runtime:** интерфейс CRI, через который kubelet создаёт и управляет контейнерами.
- **CNI-плагины:** настройка сетевых интерфейсов подов.
- **CSI-плагины:** управление подключением и отсоединением томов.
- **kube-proxy и метрики:** обмен сетевой и диагностической информацией.

> `kubelet` — это автономный агент исполнения Kubernetes. Он обеспечивает:
>
> - получение и реализацию желаемого состояния узла;
> - взаимодействие с контейнерным runtime, сетью и хранилищем;
> - постоянную синхронизацию с API-сервером по pull-модели;
> - самовосстановление и автономность при сбоях.
>
> Именно благодаря kubelet Kubernetes способен **самостоятельно поддерживать согласованность**, даже в условиях сетевых разделений и отказов контрольной плоскости.

Kube-proxy: реализация сервисной сети

kube-proxy — это сетевой агент, работающий на каждом узле Kubernetes и отвечающий за реализацию **сетевой абстракции**. Он управляет маршрутизацией сетевого трафика между подами и сервисами, обеспечивая прозрачный доступ к приложениям, независимо от того, где они физически запущены в кластере.

Роль и назначение

Сервисы Kubernetes предоставляют **виртуальные IP-адреса (ClusterIP)**, которые абстрагируют набор подов, выполняющих одну и ту же функцию. kube-proxy обеспечивает, чтобы весь трафик, направленный на этот виртуальный IP, автоматически перенаправлялся к **актуальным конечным точкам (Endpoints)**, представляющим живые поды.

Таким образом, kube-proxy выступает в роли:

- **распределённого балансировщика нагрузки**,
- **динамического маршрутизатора**,
- **интерфейса между приложениями и сервисами Kubernetes**.

Развёртывание и режимы работы

kube-proxy обычно запускается как **DaemonSet** в пространстве имён `kube-system`, обеспечивая один экземпляр на каждый узел. Это гарантирует, что каждый узел способен локально обрабатывать сетевой трафик без необходимости обращаться к центральной точке.

Режимы развертывания:

- **DaemonSet (по умолчанию)** — стандартная модель в современных кластерах;
- **Static Pod или systemd-служба** — встречается в кластерах, созданных вручную (`kubeadm`, `kops` и др.);
- **Замещение через CNI** — в кластерах с плагинами вроде *Cilium* или *Calico*, kube-proxy может быть полностью заменён встроенной реализацией балансировки на основе **eBPF**.

Поскольку kube-proxy управляет системными правилами маршрутизации, он требует **повышенных прав** для изменения `iptables`, `ipvs` или загрузки программ eBPF в ядро.

DaemonSet: кластерное развертывание агентов

`DaemonSet` — это тип контроллера Kubernetes, который гарантирует, что **экземпляр пода** запущен на **каждом узле** кластера. Он идеально подходит для системных агентов, таких как kube-proxy, логгеры (Fluent Bit, Vector), агенты мониторинга (Datadog, Prometheus Node Exporter) и сетевые плагины.

Ключевые особенности DaemonSet:

- Автоматически развертывает поды на всех узлах (в том числе новых).
- Обеспечивает единообразное покрытие инфраструктуры.
- Удаляет поды при удалении узлов.
- Может ограничиваться определённой группой узлов через селекторы (`nodeSelector`, `affinity`).

Обнаружение сервисов

В Kubernetes сервисы предоставляют **стабильные конечные точки** для доступа к подам, даже если их IP-адреса меняются. Эта абстракция позволяет приложениям общаться друг с другом по предсказуемым DNS-именам (например, `my-service.default.svc.cluster.local`), в то время как kube-proxy под капотом перенастраивает маршруты в зависимости от текущего состояния подов.

Когда создаётся сервис, API-сервер назначает ему **ClusterIP** и список связанных **Endpoint-объектов**. kube-proxy следит за этими объектами через API и обновляет сетевые правила на узле при каждом изменении — добавлении, удалении или замене подов.

Балансировка нагрузки

Одной из ключевых функций kube-proxy является **распределение трафика** между всеми живыми подами, связанными с сервисом. Поддерживаются несколько моделей балансировки:

- **iptables** — трафик перенаправляется через цепочки NAT с случайным выбором конечной точки;
- **IPVS (IP Virtual Server)** — более производительный вариант с поддержкой алгоритмов (RR, LC, WRR и др.);
- **eBPF** — высокоэффективная, современная альтернатива, позволяющая обрабатывать маршруты в ядре Linux с минимальной задержкой.

Благодаря распределённой архитектуре каждый узел обрабатывает маршруты локально, устраняя единые точки отказа.

Виртуализация сети

kube-proxy создаёт **виртуальные сетевые интерфейсы и наложения**, позволяя приложениям взаимодействовать друг с другом как будто они находятся в одной локальной сети, даже если поды размещены на разных узлах.

В режиме `iptables` или `IPVS` он конфигурирует правила NAT для перенаправления трафика, а в eBPF-реализациях — загружает программы прямо в ядро, что снижает накладные расходы и повышает производительность.

> kube-proxy обеспечивает «магическую» связность сервисов Kubernetes, скрывая сложность распределённой сети за простыми виртуальными IP-адресами. Он:
>
> - отслеживает состояние сервисов и их конечных точек через API;
> - управляет локальными таблицами маршрутизации (`iptables`, `IPVS`, eBPF`);
> - балансирует трафик между подами;
> - устраняет единые точки отказа, обрабатывая трафик локально на каждом узле.
>
> В современных реализациях (например, с **Cilium** или **Calico eBPF**) kube-proxy становится частью ядра сетевой логики, обеспечивая производительность и безопасность на уровне ядра Linux.

Среда выполнения контейнеров: уровень выполнения

Среда выполнения контейнеров (Container Runtime) — это фундаментальный слой Kubernetes, который отвечает за фактическое **запускание, остановку и управление контейнерами** на каждом рабочем узле. Если kubelet — это «мозг» узла, то контейнерный runtime — его «руки», выполняющие работу.

Роль и назначение

Контейнерный runtime управляет всем, что связано с жизненным циклом контейнера:

- извлекает образы из реестров (Docker Hub, Amazon ECR, GitHub Container Registry и др.);
- создаёт контейнеры и управляет их процессами;
- обеспечивает сетевую изоляцию и доступ к томам;
- отслеживает состояние контейнеров и сообщает kubelet о результатах выполнения.

Это ключевой компонент **плоскости данных**, который превращает декларации Kubernetes (манифесты подов) в реальные, изолированные процессы.

Развёртывание

Среда выполнения контейнеров устанавливается **непосредственно на каждом узле** как системная служба (systemd). Она запускается при инициализации узла и остаётся активной на протяжении всего жизненного цикла кластера.

Наиболее распространённые реализации:

- **containerd** — базовый runtime, изначально созданный в составе Docker, теперь используется как стандарт в Kubernetes;
- **CRI-O** — лёгкий и минималистичный runtime, ориентированный исключительно на Kubernetes;
- **Docker (dockershim)** — ранее широко использовался, но официально **удалён** из Kubernetes начиная с версии 1.24.

Каждая реализация должна поддерживать **Container Runtime Interface (CRI)** — стандартный интерфейс взаимодействия с kubelet. Kubelet обращается к runtime через **Unix-сокет**, например:

```
/run/containerd/containerd.sock
/var/run/crio/crio.sock
```

Runtime должен обладать привилегиями для:

- управления процессами контейнеров;
- настройки сетевых пространств имен;
- работы с файловыми системами и томами;
- сборки и удаления неиспользуемых образов.

Абстракция и модульность: Container Runtime Interface (CRI)

CRI — это **универсальный интерфейс взаимодействия между kubelet и средой выполнения контейнеров**, который обеспечивает совместимость и заменяемость реализаций. Он позволяет Kubernetes оставаться независимым от конкретной технологии контейнеров.

Благодаря CRI kubelet «не знает», работает ли он с Docker, containerd или CRI-O — все они предоставляют одинаковое поведение. Это делает стек Kubernetes **расширяемым и устойчивым к технологическим изменениям**.

> Пример: Если разработчики хотят использовать другой runtime, например **gVisor** для повышенной безопасности или **Kata Containers** для лёгкой виртуализации, им достаточно реализовать поддержку CRI, и Kubernetes сможет взаимодействовать с ним без модификации ядра системы.

Изоляция и безопасность

Контейнерные runtime обеспечивают изоляцию ресурсов с помощью механизмов ядра Linux:

- **Namespaces** — предоставляют изоляцию для процессов, сети, монтированных томов и идентификаторов пользователей, создавая иллюзию «собственной системы» для контейнера.
- **cgroups (control groups)** — контролируют и ограничивают использование ресурсов: CPU, памяти, ввода-вывода.
- **seccomp, AppArmor, SELinux** — обеспечивают дополнительную безопасность, ограничивая системные вызовы контейнеров.

Эти технологии создают основу **многопользовательской модели Kubernetes**, в которой разные приложения и команды могут безопасно разделять одну инфраструктуру.

Управление образами

Runtime также отвечает за управление жизненным циклом образов контейнеров:

- загрузка образов из реестров (с учётом аутентификации и политики безопасности);
- кэширование локальных копий для ускорения запуска;
- автоматическая очистка неиспользуемых образов и слоёв (garbage collection).

Эта функциональность тесно интегрирована с kubelet и механизмами безопасности (например, ImagePullSecrets).

> Среда выполнения контейнеров — это **исполнительный слой Kubernetes**, обеспечивающий запуск и управление контейнерами. Она:
>
> - реализует спецификацию CRI для совместимости с kubelet;
> - управляет жизненным циклом контейнеров и образов;
> - изолирует процессы с помощью механизмов ядра Linux (namespaces, cgroups);
> - поддерживает безопасность и многопользовательские сценарии.
>
> На практике большинство современных кластеров используют **containerd** как стандартный runtime, обеспечивая стабильность, простоту и совместимость с будущими версиями Kubernetes.

Заключение

Kubernetes является примером сложной архитектуры распределенных систем, демонстрирующей реализацию систем, которые обеспечивают баланс между согласованностью и доступностью, масштабируемостью и надежностью, а также простотой эксплуатации и широкими функциональными возможностями. Выполнение операций `kubectl apply` запускает скоординированные распределенные алгоритмы, протоколы консенсуса и архитектуры, управляемые событиями. Понимание этих архитектурных шаблонов позволяет развить как экспертизу в области Kubernetes, так и более широкие навыки проектирования распределенных систем.

Kubernetes реализует основные шаблоны распределенных систем, включая циклы согласования, гибридные модели согласованности, выбор лидера и плавное снижение производительности. Эти паттерны встречаются во всех современных технологических стеках: системах баз данных, архитектурах микросервисов, брокерах сообщений и сетях доставки контента.

Kubernetes служит как производственной платформой, так и примером распределенной системы, демонстрируя, что сложные распределенные системы могут быть успешно реализованы с помощью принципиального архитектурного проектирования, четких уровней абстракции и системных инженерных подходов. Задача разработки распределенных систем заключается не в том, чтобы избежать сложности, а в том, чтобы управлять ею с помощью проверенных архитектурных паттернов и принципов проектирования.