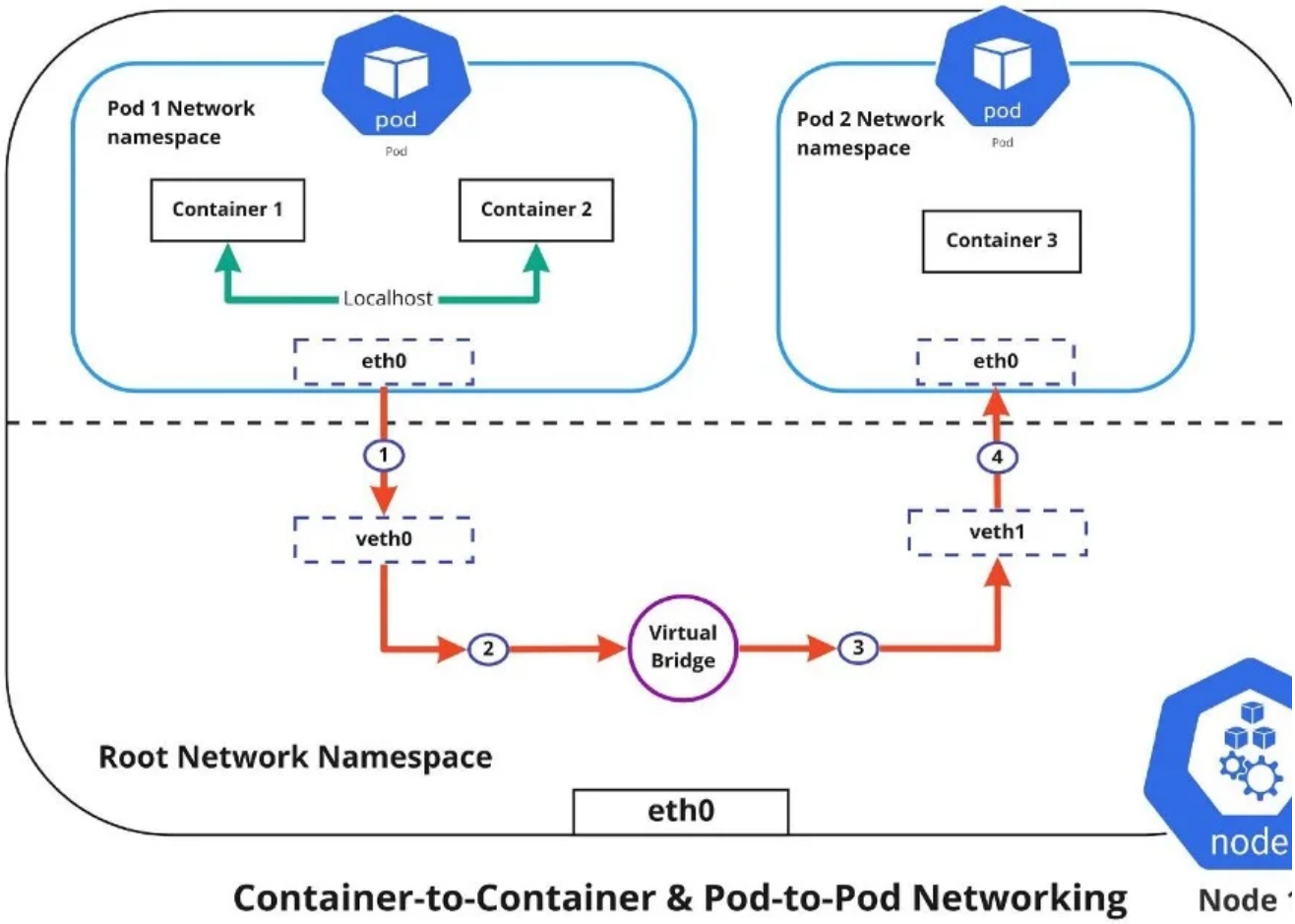
Основные концепции сетевой архитектуры Kubernetes, а также CNI, Service Mesh и т.д.

Базовые сетевые концепции Kubernetes



Вообще по задумке Kubernetes придерживается модели плоской сети, где все поды и узлы могут свободно обмениваться трафиком по IP без NAT. Но к настоящему времени картина

немного усложнилась, чтобы учесть требования безопасности и другие требования — добавились CNI плагины и много всего другого, что мы ниже и обсудим

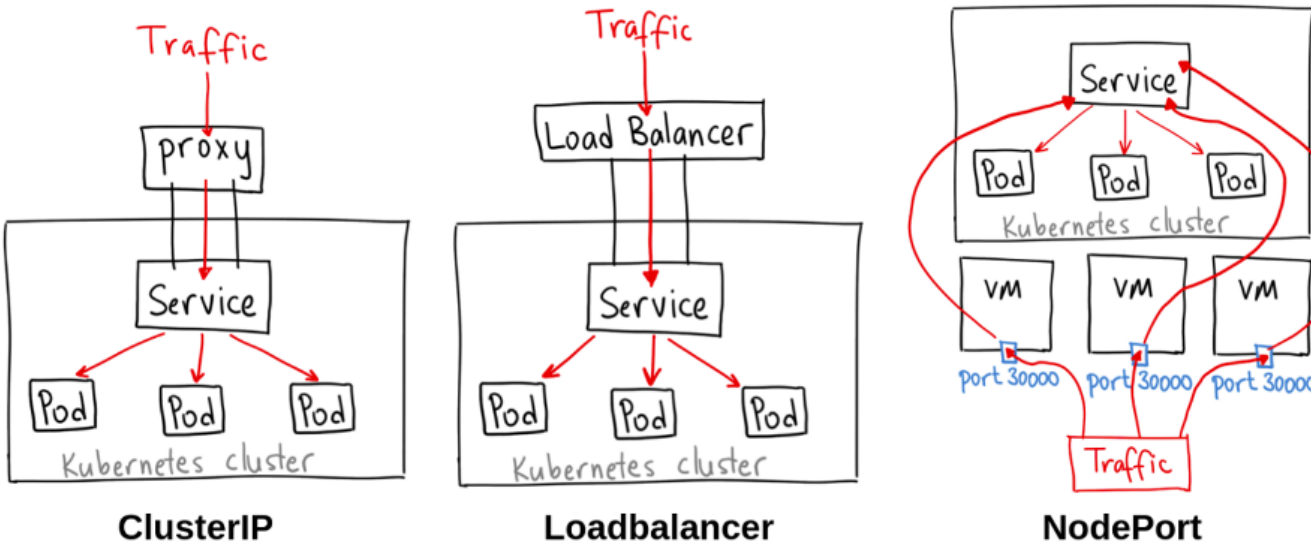
Pod и модель сети. Каждый Pod в Kubernetes получает уникальный IP в пределах кластера. Все поды находятся в одном сетевом неймспейсе, благодаря чему любой Pod может достучаться до любого другого напрямую (если не ограничено политиками). IP, который Pod видит у себя, совпадает с тем, по которому его видят остальные – никаких NAT внутри кластера нет. Это упрощает взаимодействие сервисов: несколько подов могут слушать один и тот же порт (у каждого свой IP), исключая конфликты портов.

CNI – плагин сетевого интерфейса. Kubernetes не реализует сетевые функции сам по себе – эту роль выполняют плагины Container Network Interface (CNI). Именно CNI настраивает сетевые интерфейсы подов, выделяет им IP, настраивает маршрутизацию и т.д. Kubelet на каждом узле вызывает CNI-плагин при запуске/удалении подов, чтобы создать/убрать интерфейсы и маршруты. Существует множество CNI-реализаций (Calico, Flannel, Cilium etc.), которые реализуют базовые требования модели Kubernetes (сквозная IP-связность подов) разными способами — через оверлейные сети, маршрутизацию BGP, eBPF и пр.

Kube-proxy. Для организации доступа к Service Kubernetes использует компонент kube-proxy, запущенный на каждом узле. Он отвечает за виртуальные IP сервисов (ClusterIP) и балансировку трафика от сервисов на поды. Исторически kube-proxy использует правила iptables (или IPVS) в ядре Linux для проксирования: при создании Service и связанных Endpoints kube-proxy добавляет набор правил NAT, чтобы перехватывать пакеты на IP сервиса и перенаправлять их на IP целевые поды. В режиме iptables каждая новая служба добавляет правила (представляете себе количество этих правил в выводе `iptables -L -n -v`)); в режиме IPVS используется внутренний балансировщик ядра, что повышает производительность на больших кластерах (IPVS лучше масштабируется, поддерживает больше алгоритмов балансировки и более эффективен при 1000+ сервисов).

Сетевые сервисы Kubernetes. Давайте немного вспомним основные типы Service:

- **ClusterIP** — виртуальный внутренний адрес для доступа к приложению внутри кластера. Запросы на ClusterIP распределяются между подами (эндпоинтами) этого сервиса посредством kube-proxy (с использованием iptables/IPVS).
- **NodePort** — как следует из названия, сервис получает фиксированный порт на каждом узле, через который доступен извне (подходит для простого проброса порта, но вручную управлять не очень удобно).
- **LoadBalancer** — интеграция с внешним балансировщиком (обычно облачным). Kubernetes сам запрашивает у провайдера внешний IP/балансировщик и привязывает его к сервису.
- **Headless Service** (`ClusterIP=None`) — особый случай, когда DNS-сервис Kubernetes возвращает IP всех подов без виртуального IP, позволяя клиентам реализовать свой алгоритм балансировки или обнаружения.



4 типа Service

Ingress. Ingress-ресурс в Kubernetes представляет **L7-маршрутизацию (HTTP/HTTPS)**, определяя правила, как внешние запросы (по URI, хосту и т.д.) направлять на сервисы внутри кластера. Однако сам по себе объект Ingress — лишь конфигурация (объект в API Kubernetes); для его работы нужен Ingress Controller — отдельный контроллер/прокси, который следит за объектами Ingress и на их основе управляет L7-прокси (например, на базе NGINX, Envoy, Traefik и тд). Kubernetes не поставляется с контроллером Ingress из коробки — админ сам ручками устанавливает подходящий

NetworkPolicy. По умолчанию, сеть Kubernetes полностью открыта (плоская сеть, обсуждали выше): любой Pod может обращаться к любому другому. Но существует способ изменить такое поведение, и NetworkPolicy — это тот самый объект, который позволяет ограничить этот трафик, задав правила на уровне подов и namespace. NetworkPolicy работает примерно так — по умолчанию Pod принимает всё, но если к нему применена хотя бы одна политика, то начинает блокироваться всё, что не разрешено в правилах. Политики могут ограничивать входящий (ingress) и/или исходящий (egress) трафик на основе селекторов подов, namespace и портов. Например, можно разрешить доступ к БД только из подов с меткой `app=backend` того же неймспейса, или вообще изолировать среду, запретив всем подам выход в интернет, кроме специальных. NetworkPolicy — это объект Kubernetes, но обеспечивает его выполнение сам CNI-плагин (поэтому необходим выбор CNI с поддержкой сетевых политик, таких как Calico, Cilium, Weave, Antrea и др.; а вот простые CNI вроде Flannel игнорируют NetworkPolicy). В сложных сценариях безопасности одной NetworkPolicy может быть мало – тогда на помощь приходят сервис-меши (mTLS, политики на основе идентификации сервисов и тд, обсудим это чуть дальше).

DNS. В кластере Kubernetes автоматически развёрнут DNS (CoreDNS) для разрешения имён сервисов и подов. Каждому сервису присваивается DNS-имя вида `<my-svc>.<my-namespace>.svc.cluster.local`. Под капотом Kubernetes отслеживает события создания сервисов и эндпоинтов и генерирует/обновляет DNS-записи. Благодаря этому приложения могут обращаться друг к другу по логическим именам. Также kubelet на узлах прописывает в DNS записи для удобства (например, `<pod-ip-address>.<my-namespace>.pod.cluster.local` для

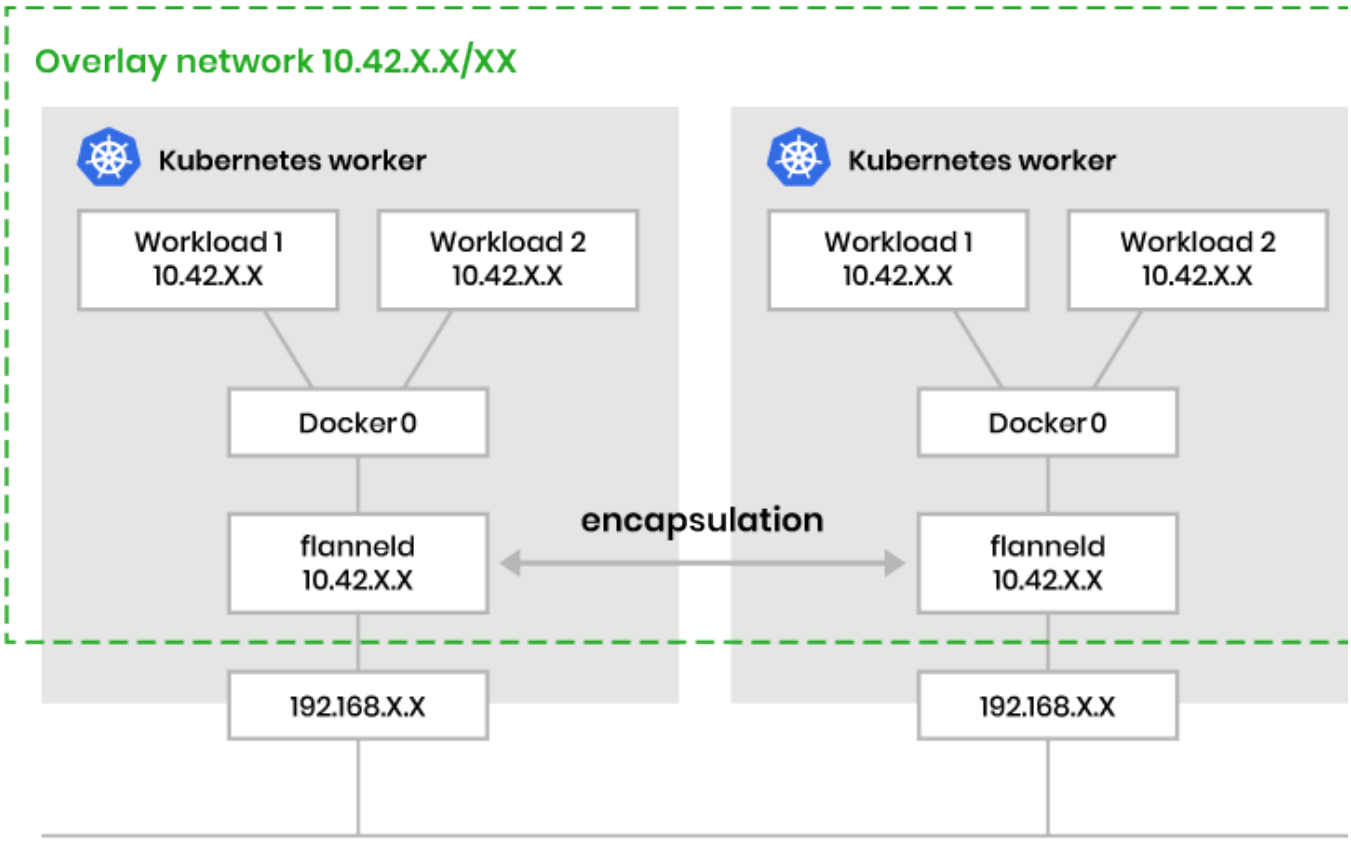
доступа к конкретному Pod по имени).

Промежуточные выводы: Kubernetes обеспечивает плоскую сеть для подов, вводит виртуальные сервисы с балансировкой (ClusterIP / NodePort / LoadBalancer) и L7-маршрутизацию через Ingress, а за безопасность отвечает механизм сетевых политик. Всё это реализуется в связке с CNI и kube-proxy, которые настраивают сетевые примитивы ядра Linux (veth-интерфейсы, маршруты, iptables и др.).

Современные CNI-плагины

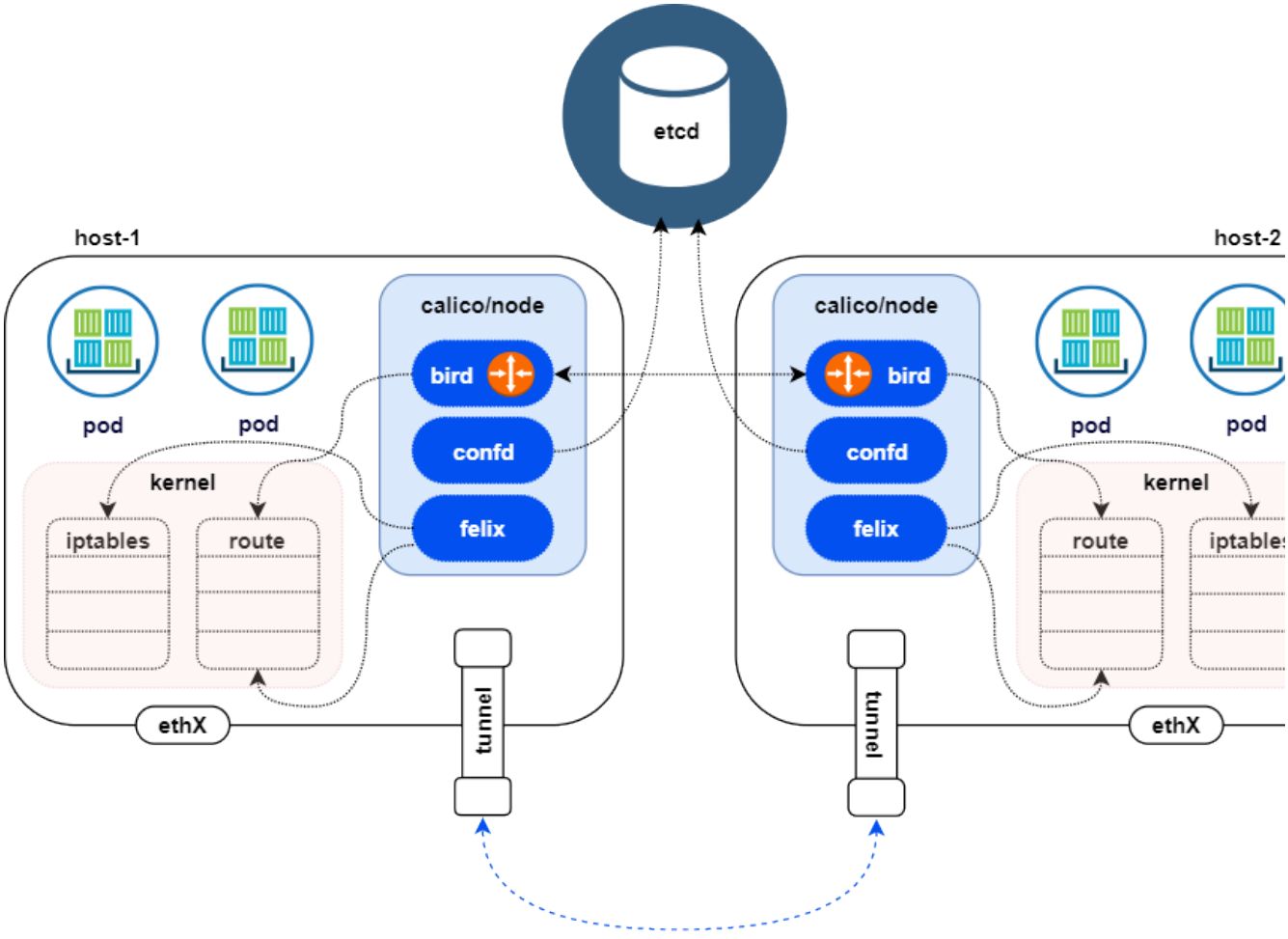
Выбор CNI-плагина определяет, как именно будут соединены поды в кластере: прямая маршрутизация, оверлейные туннели, использование eBPF и т.д. По моим ощущениям и наблюдениям, которые в принципе соответствуют статистике, особо часто используются такие CNI-плагины:

- **Flannel** — (он ещё жив)) простейший L3-оверлей от CoreOS. Каждому узлу выделяется подсеть для подов, трафик между узлами упаковывается во VXLAN (по умолчанию) или UDP туннели. Flannel запускает на каждом узле демона flanneld, который занимается раздачей подсетей и поддерживает маршруты (хранит конфиг в etcd или через API Kubernetes). **Плюсы:** минимальная сложность, один бинарник, работает из коробки для простых сценариев. **Минусы:** нет поддержки NetworkPolicy (только обеспечивает связность); при росте кластера оверлей может стать узким местом, скорость ниже, чем у прямой маршрутизации. Да, Flannel поддерживает IPsec для шифрования межузлового трафика, но это дополнительный оверхед

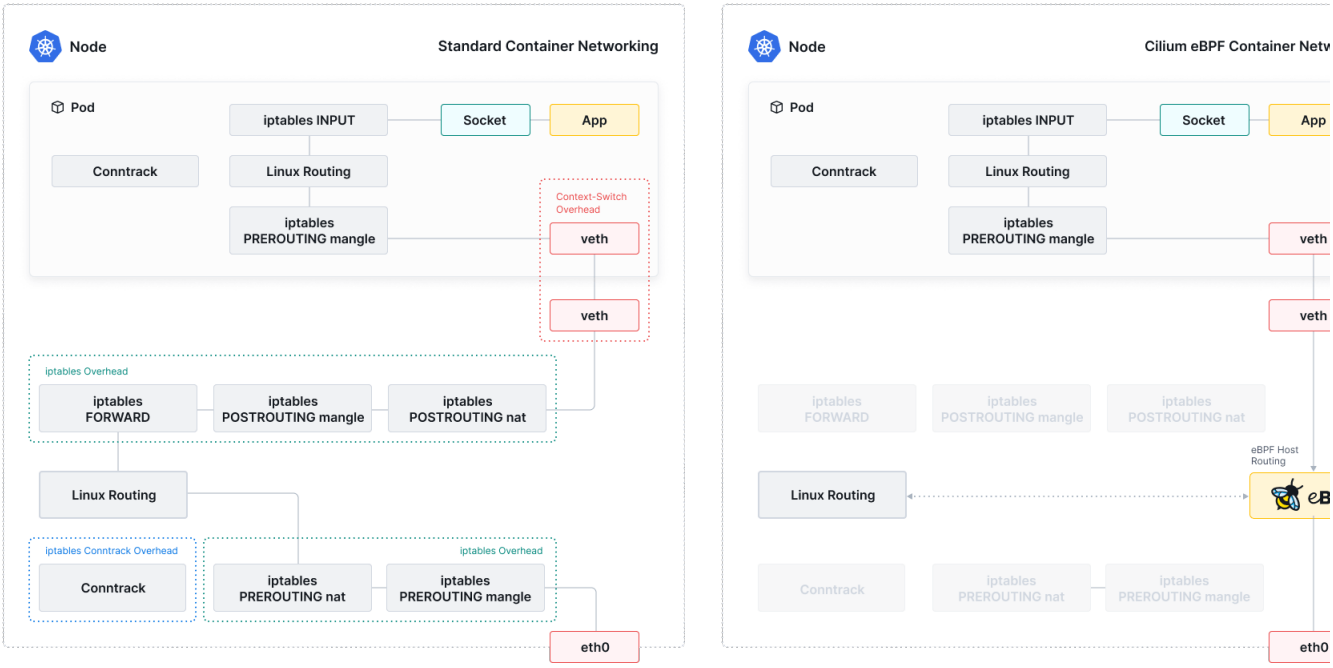


Оверлейная сеть, создаваемая Flannel

- **Calico** — CNI-плагин от Tigera, обеспечивающий как сеть, так и безопасность. Calico по умолчанию делает маршрутизацию L3 через BGP: на каждом узле агент анонсирует в кластерную сеть маршруты к своим подам (либо через полноценный BGP между узлами, либо, в облаке, через BGP-сессии с сетевыми устройствами провайдера или маршрутизатором). При прямой маршрутизации пакеты идут без дополнительных заголовков, что даёт высокую производительность. Если BGP не подходит (например, overlapping IP), Calico может работать и в режиме оверлея (IP-in-IP или VXLAN туннели). Сильная сторона Calico — сетевые политики: он реализует Kubernetes NetworkPolicy (и собственные расширения глобальных политик), позволяя тонко контролировать, кто с кем общается. Кроме того, Calico умеет шифровать трафик между нодами с помощью WireGuard, поддерживает протоколы вроде SCTP, интегрируется с сервис-мешами для сквозных политик безопасности. **Плюсы:** высокая производительность (нет оверлейной обёртки на каждом пакете), продвинутая безопасность, зрелость (используется многими компаниями в продакшене; кстати при установке Kubespray тоже предлагает использовать Calico). **Минусы:** относительно сложнее в настройке, чем Flannel (нужно понимание BGP/IP-инфраструктуры); отсутствует поддержка multicast



- **Cilium** — CNI нового поколения (как все его гордо именуют), использующий eBPF для обработки трафика прямо в ядре. Cilium устанавливает на узлах cilium-agent, который загружает eBPF-программы для реализации сетевых функций. Cilium поддерживает два режима: оверлей (VXLAN) или безоверлейную маршрутизацию (в том числе через BGP, если подключить дополнительно, либо прямой routing при сетевой связанности узлов). У Cilium в фокусе не только L3/L4, но и L7 (application-aware): можно писать NetworkPolicy с фильтрацией по HTTP-запросам, использовать identity-based policy, наблюдать за трафиком с помощью встроенных инструментов (Hubble) — всё благодаря возможностям eBPF перехватывать и анализировать пакеты на лету. **Плюсы:** высокая производительность, особенно на больших масштабах (eBPF обходится без тысяч iptables-правил); богатая функциональность (от NetworkPolicy до сервис-меша, обсудим ниже); поддержка multi-cluster (ClusterMesh) для объединения нескольких кластеров на базе Cilium; активное развитие со стороны сообщества и компаний (Isovalent, Google и др.). **Минусы:** требует современного ядра Linux и определённых навыков для отладки; некоторые возможности (например, BGP для внешней маршрутизации) могут требовать дополнительной интеграции или сочетания с другим CNI; настройка multi-cluster с Cilium сложнее, чем использование специализированных инструментов.



- **Weave Net** — CNI от WeaveWorks, реализующий mesh-оверлей. Каждый узел устанавливает соединения со всеми другими — получается «полносвязная» сеть. Weave Net использует собственный протокол на уровне ядра (fast datapath) для ускорения доставки пакетов без лишних переключений контекстов. Если такой путь недоступен, он прибегает к fallback-механизму (sleeve) для пересылки трафика. WeaveNet включает свой DNS-сервер (WeaveDNS) для разрешения имён между контейнерами. Поддержка NetworkPolicy реализована через отдельный контейнер weave-npc, который внедряет iptables-правила по политикам K8s. Да, и ещё Weave может шифровать трафик (IPsec) для безопасности. **Достоинства:** очень простая установка, автоматическое построение mesh-сети, работает даже в сложных сетях

(поддерживает overlapping networks, т.к. полностью виртуализует сеть поверх существующей). Есть коммерческая поддержка. **Недостатки:** из-за использования на уровне ядра Linux – только для Linux-узлов; доп. шифрование и оверлей могут снижать скорость (fast datapath нивелирует часть накладных расходов, но всё же прямой routing быстрее)

- **Antrea** — проект VMware, Kubernetes-нативный CNI на базе Open vSwitch (OVS). Каждому узлу ставится OVS-коммутатор, а Antrea-контроллер управляет потоком правил в этих коммутаторах. Таким образом, Antrea обеспечивает единый стек сети и поддерживает Kubernetes NetworkPolicy на уровне потоковых правил OVS. Antrea легко интегрируется с экосистемой VMware (NSX) и может расширяться для особых нужд (например, аппаратное ускорение с SmartNIC). **Плюсы:** единый подход к разным инфраструктурам (OVS абстрагирует различия облаков/металла); гибкие сетевые политики (поддерживает стандартные NP и дополнительные CRD от Antrea, например Antrea NetworkPolicy для межкластерных или L7-фильтров); высокий performance на больших policy-списках (OVS масштабируется лучше iptables по числу правил). **Минусы:** добавляет ещё один слой (OVS); чуть менее популярен, поэтому сообщество меньше, чем у Calico/Cilium.
- **Multus** — особый CNI-плагин, точнее **мета-плагин**, позволяющий подключать к поду несколько сетевых интерфейсов. По умолчанию у пода одна сеть, но Multus даёт возможность добавлять дополнительные сети (например, вторую сеть для хранения, выделенную SR-IOV интерфейс для высокоскоростного трафика и т.п.). Multus не заменяет основной CNI, а работает поверх него: он вызывает основной CNI для основного интерфейса и дополнительные CNI для дополнительных. Multus часто используется в NFV (виртуализация сетевых функций), Telco-приложениях, а также совместно с Cilium или Calico для выделения сервис-сети и data-plane сети.

Основные из обсуждаемых выше тезисов можно свести в такую табличку:

CNI-плагин	**Подход к сети**	**Особенности**
Calico	L3 маршрутизация (BGP) или оверлей (IPIP/VXLAN)	Масштабируемый CNI-плагин с хорошей производительностью; обеспечивает гибкую сетевую связность; использует BGP
Flannel	L3 оверлей (VXLAN по умолчанию)	Легковесный CNI-плагин; создаёт оверлейную сеть (с использованием VXLAN или UDP) для подключения подов; вопреки расхожему мнению Flannel ещё жив и вполне себе используется
Cilium	eBPF в ядре (без IPTables), VXLAN оверлей или прямой L3 (BGP/маршруты)	Использует eBPF для обеспечения высокопроизводительного, масштабируемого и безопасного сетевого взаимодействия с детальной политикой безопасности и с мониторингом

Weave Net	L2/L3 mesh-оверлей	Создаёт виртуальную, децентрализованную сеть с автоматическим обменом информацией о маршрутах, инкапсуляцией пакетов и поддержкой обнаружения сервисов
Antrea	L2 оверлей на OVS	Kubernetes-нативный OVS-подход: единый стек на разных площадках (облако/он-прем). Глубокая интеграция с экосистемой VMware (NSX). Выбор для гибридных облаков и enterprise-сред, требующих расширенных политик и мониторинга.
Multus	Meta-CNI (комбинация сетей)	Не обеспечивает сеть сам по себе, а позволяет поду иметь несколько интерфейсов (каждый через свой CNI). Используется для разделения сетевого трафика по различным сетям/владельцам

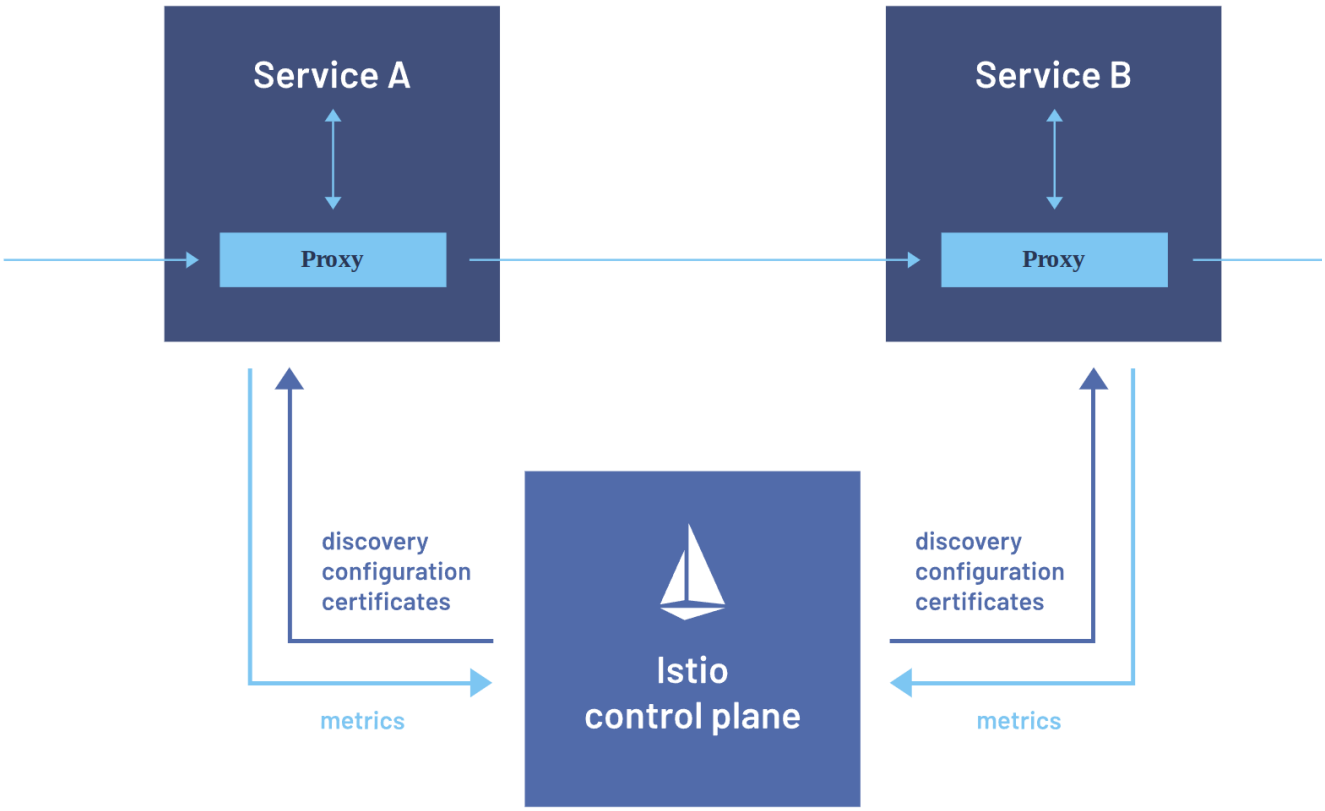
Service Mesh и его влияние на сеть

Service Mesh — надстройка, решающая задачи коммуникации между сервисами на уровне приложений (L7): автоматическая трассировка, балансировка запросов, шифрование между сервисами (mTLS), авторизация и т.д. Популярные реализации — Istio, Linkerd, Cilium Service Mesh (и другие, например Consul Connect).

Классически Service Mesh использует паттерн sidecar. Напомню суть этого паттерна: в под к контейнеру с приложением добавляется вспомогательный контейнер-sidecar, например Envoy. Этот прокси перехватывает весь входящий и исходящий трафик приложения (как правило, через iptables-редирект на уровне пода) и выполняет необходимые функции: шифрует соединения (mTLS), собирает метрики, маршрутизирует вызовы по правилам (например, 10% трафика на новую версию для Canary-релиза) и т.п. Со стороны сети кажется, что под A общается напрямую с другим подом B, но на самом деле трафик идёт так: приложение A ⟶ локальный sidecar ⟶ sidecar получателя ⟶ приложение B.

Istio — наиболее функциональный (на данный момент) и распространённый Service Mesh. Использует прокси Envoy в качестве сайдкара и также использует Istiod, который позволяет настраивать прокси Envoy. Istio предлагает богатые возможности управления трафиком (VirtualService/DestinationRule для реализации Canary-релизов, mirror-трафика и пр.), сквозное шифрование mTLS (PeerAuthentication, AuthorizationPolicy), интеграцию с телеметрией (Prometheus, Jaeger), fault injection, и многое другое. В чём подвох? Плата за гибкость — сложность и накладные расходы. Каждый Pod с сайдкаром потребляет дополнительные CPU/RAM и добавляет ~1–2 перехода по сети (через прокси). В больших масштабах (сотни микросервисов) это приводит к множеству прокси, что Келси Хайтауэр метко назвал "service mess". В 2023 году Istio представил режим Ambient Mesh (без

сайдкаров, с даемонов ztunnel), но он ещё развивается.



Linkerd — лёгкий сервис-меш, изначально от Kubernetes community (CNCF). В Linkerd 2 используется собственный ультра-легковесный прокси (написан на Rust кстати) как сайдкар. Функциональность у Linkerd чуть меньше, чем у Istio (ранее не было полного аналогичного L7-маршрутизатора — только простые методы балансировки и мониторинга), но он проще в установке и потребляет меньше ресурсов. Linkerd делает упор на простоту и высокую надёжность базовых функций: mTLS по умолчанию, ретраи/таймауты, метрики Golden Metrics (latency, success rate) в каждый прокси. Для многих случаев этого достаточно, и Linkerd завоевал популярность как более простой в использовании меш.

Cilium Service Mesh — относительно новый подход, объединяющий L3/CNI и L7/меш в единый слой. Cilium, как описано выше, может на уровне ядра перехватывать трафик. За счёт eBPF Cilium реализует часть функций меша без сайдкаров: шифрование mTLS на уровне узла (ключи загружаются в ядро, трафик между подами шифруется прозрачно), L7-политику (фильтрация HTTP), базовую балансировку. Для продвинутых сценариев (подмена URL, сложный роутинг gRPC и т.д.) Cilium интегрируется с Envoy, но Envoy может работать не как сайдкар, а как демонсет на узле. В таком режиме на узле запущен один Envoy, который обслуживает несколько подов, вызываясь eBPF-программами только при необходимости L7-преобразований. Это устраняет двойное прохождение через прокси (в обычном service mesh – исходящий прокси у клиента + входящий у сервиса), что снижает задержки и накладные расходы. Если же нужен полный функционал Istio, Cilium умеет работать и вместе с Istio (используя знание о sidecar Envoy, оптимизируя путь до него). Таким образом, Cilium Service Mesh предлагает выбор: с сайдкаром или без. Основной упор Cilium Service Mesh — снижение

оверхеда и усложнений, связанных с сайдкар-мешами, с помощью переноса части операций в ядро (eBPF).

Влияние service mesh на сетевую архитектуру: с точки зрения CNI и сети L3, service mesh — дополнительный уровень. Он не заменяет базовую сеть (поды всё так же имеют IP и обмениваются пакетами), но добавляет прокси-прослойку. Для того, кто будет администрировать, это означает:

- Например, нужно учитывать, что Istio по умолчанию переподнимает весь входящий трафик на порту 15006 Envoy. Из-за этого традиционные сетевые политики, основанные на IP/порт, могут начать работать иначе.
- Также усложняется отладка: `tcpdump` внутри пода покажет трафик до/после прокси, а не напрямую между приложениями. При проблемах нужно проверять как сеть L3 (CNI), так и конфигурацию виртуальных сервисов, сертификаты mTLS и т.д.
- Появляются дополнительные сервисы: у Istio, например, появляется IngressGateway (специальный Envoy на вход в mesh) — это ещё один компонент, за которым нужно следить (мониторинг, масштабирование и тд).
- Ресурсы: тысячи сайдкаров могут потреблять существенный объём CPU/RAM. В таком случае бюджеты ресурсов нужно планировать с учётом ~20-30% оверхеда на service mesh.

Несмотря на эти мрачные стороны service mesh, они уже стали стандартным компонентом облачных платформ, так что нужно ориентироваться в том, как они работают. Пара юзкейсов — например, нужно знать, как настроить сетевые политики Calico, чтобы они учитывали трафик Istio, или как включить режим перехвата всего egress-трафика через mesh gateway и убедиться, что Cloud NAT/Firewall пропускает только разрешённые порты. Также нужно понимать модель безопасности: при mTLS сервисы аутентифицируются по X.509 сертификатам, а network policy на уровне IP может видеть только зашифрованный поток. Поэтому часто сеть L3 используется для высокоуровневой изоляции (скажем, на уровне неймспейсов), а внутри неймспейсов доверие и разграничение реализуется с помощью mTLS и авторизации запросов.

TLDR: Service Mesh добавляет уровень L7 поверх сети Kubernetes. Ключевые решения — Istio (богатый функционал, но тяжеловесный), Linkerd (проще и легче) и новые подходы типа Cilium Service Mesh (стремление к бесшовной интеграции с CNI через eBPF). Админу Kubernetes важно понимать накладные расходы сайдкаров и уметь отлаживать проблемы, возникающие в распределённых микросервисах.

Балансировка нагрузки и контроллеры Ingress

Для реализации к сервисам доступа извне используются 2 основных механизма: сервисы типа LoadBalancer (обычно опираются на внешние средства) и Ingress-контроллеры (реализуют обратный прокси внутри кластера).

LoadBalancer. В облаках (AWS, GCP, Azure, ~~YandexCloud~~) Kubernetes может автоматически создавать внешний балансировщик для сервиса с типом LoadBalancer. Например, в AWS это

классический ELB/ALB, в GCP — Cloud Load Balancer. Кластер запрашивает у API облака ресурс балансировщика, настраивает его на нужный порт, и привязывает к IP-адресам узлов (для NodePort) или прямо к подам (в некоторых облаках). Это самый простой способ предоставить сервису связь с внешним миром, однако он зависит от облака и несёт определённые задержки на сознание и настройку облачного LB. В он-прем же такого механизма нет, поэтому используются программные балансировщики типа MetalLB — демон, который притворяется контроллером балансировщиков: в режиме L2 он отвечает ARP на виртуальные IP сервисов, а в режиме BGP анонсирует IP сервисов в вашу физическую сеть.

Ingress-контроллер. Ingress позволяет описать L7-правила (маршруты HTTP) и виртуальные хосты. Но Ingress как мы помним это всего лишь объект в API Kubernetes, поэтому ему для работы необходим контроллер — обычно это специально настроенный веб-прокси. Из того, что я видел: NGINX Ingress, HAProxy Ingress, Traefik, ну и решения на базе Envoy (проекты Contour, Ambassador или сам Istio ingressgateway). Все они выполняют схожую функцию – принимают HTTP/HTTPS снаружи и проксируют его к нужному сервису в кластере по правилам. При этом есть некоторые нюансы:

- **Nginx Ingress** — де-факто стандарт, много лет проект kubernetes/ingress-nginx служит надёжной рабочей лошадкой. Nginx известен своей стабильностью и богатством возможностей (TLS, перенаправления, сжатие, переписывание URL и т.п.). Контроллер Nginx имеет большую комьюнити-поддержку и хорошо подходит для продакшена. Из минусов разве что — конфигурация через аннотации не всегда прозрачна, а некоторые продвинутые вещи требуют знание синтаксиса самого Nginx.
- **Traefik** — современный прокси, написанный на Go, популярный в клауд-нейтив кругах. Traefik легко интегрируется с динамическими средами: автоматически обнаруживает новые сервисы, умеет запрашивать и обновлять сертификаты Lets Encrypt, имеет удобную дашборду. Его часто выбирают, если важны динамическое обновление конфигурации и автоматизация SSL — Traefik может без перезапуска применять новые маршруты и сертификаты. Также Traefik поддерживает много протоколов (WebSocket, HTTP/2, gRPC) и дополнительные фишки (middleware для авторизации, rate limiting). Минусы: на очень высоких нагрузках уступает Nginx/HAProxy по производительности; ну и его конфигурация менее знакома среднестатистическому админу.
- **HAProxy Ingress** — контроллер на базе HAProxy, который, как мы знаем, один из самых быстрых балансировщиков L7. HAProxy-ингресс может обрабатывать в 2 раза больше запросов в секунду, чем Envoy или Nginx, и при этом потребляет меньше CPU. Он отлично справляется с большим числом соединений и малым временем отклика. Контроллер HAProxy предлагает много настроек, включая TCP-проксирование (для нестандартных протоколов). Его стоит рассматривать, если требуется максимальная производительность и низкие задержки. Из минусов — сообщество меньше, чем у Nginx, и чуть сложнее найти примеры конфигов.
- **Envoy (API Gateway)** — Envoy применяется не только в mesh, но и в качестве standalone ingress/API-шлюза. Проекты Contour и Ambassador — популярные контроллеры, использующие Envoy. Envoy славится поддержкой продвинутых L7-фишек: динамическое обновление конфигурации через gRPC (xDS API), возможности

телеметрии. Envoy спроектирован как Cloud Native с самого начала и имеет отличную поддержку HTTP/2, gRPC. Если требуются расширяемость и интеграция с service mesh, Envoy-контроллер – хороший выбор. Если говорить о проектах Contour и Ambassador, то первый фокусируется на простоте: использует CRD HTTPProxy (альтернатива Ingress, упрощающая сложные конфиги), а Ambassador делает упор на интеграцию API Gateway (авторизация, и тд).

Тут можно упомянуть, что в Kubernetes развивается новый стандартизованный API для сетевых шлюзов — Gateway API. Он более гибкий, чем старый Ingress (поддерживает разграничение ролей, разных протоколов: не только HTTP, и тд). В целом выглядит интересно и многообещающе

Немного подрезюмируем информацию о ingress-контроллерах:

Контроллер	Особенности	Когда применять
Nginx Ingress	Проверенная временем надёжность; много функций и настроек (переписывание URL, терминация SSL/TLS, аутентификация); большое комьюнити.	Универсальное решение для большинства кластеров, стабильность, предсказуемость и широкая поддержка (много примеров).
Traefik	Динамическая конфигурация (автообнаружение сервисов); встроенная поддержка ACME (Lets Encrypt) для SSL; веб-интерфейс для мониторинга.	Идеален для сред с часто меняющимися сервисами, где важна автоматизация. Настраивается довольно быстро
HAProxy Ingress	Высокая производительность и низкие задержки; минимальное потребление CPU при больших нагрузках; сильная сторона – TCP/UDP балансировка (L4), а также L7.	Когда трафик очень интенсивный или критична скорость отклика (финтех, рекламные системы). Либо если требуется единый контроллер для HTTP и нестандартных TCP-сервисов.
Envoy-based (Contour/Ambassador)	Расширяемость через фильтры Envoy; готовность к сложным сценариям (Auth, rate-limit, gRPC proxy и др.); плавная интеграция с сервис-мешем (тот же Envoy позволяет разделять конфиг).	Выбор для API Gateway сценариев — когда ingress должен выполнять продвинутые функции API-шлюза, или в средах, где уже используется Istio и нужна консистентность стека.

На самом деле, рассматривая сеть в Kubernetes и говоря про CNI и service mesh, для полноты картины можно ещё рассмотреть вопрос взаимодействия между несколькими кластерами (неплохое решение предлагает Cilium cluster mesh, до 8 кластеров), а также вопрос взаимодействия k8s с сетью облачных провайдеров, но видимо это будет в другой статье)

А пока предлагаю обсудить решение самых частых проблем с сетью в кластере, которые могут возникнуть

Траблшутинг сети в Kubernetes

Базовые шаги диагностики в целом очевидны:

1. **Понять, в чём проблема** — локализовать, что не работает: под не может достучаться до другого пода (проблема внутрикластерной связи), или сервис не отвечает (проблема с kube-proxy или эндпойнтами), или может входящий трафик не доходит (проблема с Ingress или с LoadBalancer)
2. **Базовая проверка:** сперва стоит убедится, что под запущен и имеет IP — `kubectl get pod -o wide`, что на ноде создан veth-интерфейс — `ifconfig` или `ip addr` (интерфейс назван в духе `caliXXXX` для Calico или `vethXXX`). Если Pod вообще не получил IP — проблема на уровне CNI (плагин работает не корректно), смотрим логи CNI (например, `kubectl logs -n kube-system pod/calico-node-...`).
3. **Проверка доступности DNS:** часто приложения ломаются из-за невозможности разрешить имя сервиса. Можно проверить CoreDNS: `kubectl get pods -n kube-system -l k8s-app=kube-dns` — поды должны быть Running. Можно зайти в любой Pod и попробовать `nslookup myservice.myns.svc.cluster.local` или просто `nslookup kubernetes.default`. Если не отвечает — смотрим логи CoreDNS через `kubectl logs`. Распространённая причина — проблемы с iptables-правилами для kube-dns сервисов или сбой CoreDNS плагина.

Полезные для траблшутинга инструменты, команды и т.д:

- Запуск временных контейнеров для диагностики. Например, если нужно проверить подключение по сети, можно запустить под `nicolaka/netshoot`:

  ```
  kubectl run -it --rm debug --image=nicolaka/netshoot -- /bin/bash
  ```

  Этот образ `netshoot` содержит много сетевых утилит (ping, dig, iperf, и тд). Внутри него можно попытаться достучаться до нужного адреса/сервиса (ping, curl). Если под уже существует, но без нужных утилит (часто образы минимальны), можно использовать временный контейнер, подключаемый к работающему поду для отладки. Что-то в духе:

  ```
  kubectl debug -it mypod --image=busybox --target=app-container -- /bin/sh
  ```

  Это добавит в под `mypod` новый контейнер из образа busybox и сразу откроет шелл. Довольно удобно, поскольку не нужно давать доступ по SSH на ноду — отладка происходит на уровне пода.
- **tcpdump и анализ трафика:** Можно запустить `tcpdump` внутри временного контейнера, или даже установить его прямо в под. Полученный трафик можно вывести и проанализировать с помощью Wireshark:

  ```
  kubectl exec mypod -c app -- tcpdump -i any -w - | wireshark -k -i -
  ```

  Это позволит увидеть, какие пакеты приходят (и приходят ли) на интерфейс контейнера. Конечно, если под не получает трафик из-за некорректных сетевых политик ноды, возможно, придётся делать tcpdump на ноде. Обычно доступ по SSH на ноды ограничен (в managed кластерах), но если возможно — `tcpdump -i`

`<интерфейс пода> host <PodIP>` на узле покажет, проходят ли пакеты.

- **iptables / IPVS диагностика:** На узле, где обнаружен проблемный под или сервис, можно проверить цепочки iptables:

  ```
  iptables -L -t nat | grep <ServiceIP>
  ```

  Должна быть запись в KUBE-SERVICES, которая DNAT ServiceIP -> PodIP (Endpoint). Если её нет — kube-proxy не прописал правило, значит либо kube-proxy не работает, либо Service/Endpoint неправильно законофигурированы. В режиме IPVS:

  ```
  ipvsadm -Ln
  ```

  покажет виртуальные сервисы и реальные сервера. Опять же, смотрим есть ли наш ClusterIP.
  Частый кейс: сервис есть, а поды к нему не подключаются. Проверяем, есть ли эндпойнт: `kubectl get endpoints mysvc` — если пусто, значит сервис не нашёл Pod (метки не совпали, или Pod not Ready). Если endpoints есть, но iptables не содержит — проблема в kube-proxy (смотрим `kubectl logs -n kube-system kube-proxy-...`).
- **NetworkPolicy отладка:** Очевидно, если создана политика и что-то не работает, лучше сначала временно удалить эту политику и проверить. У Calico кстати есть утилита calicoctl, с помощью которой можно смотреть какие именно правила применены на узлах (calico выводит iptables rules). Но обычно достаточно внимательно проверить селекторы и namespace политики.
- **eBPF-трассировка:** В современных ядрах можно использовать bpftrace, либо специализированные утилиты (например, `cilium monitor` если используется Cilium). Это более продвинутый шаг — позволяет увидеть на уровне ядра, проходит ли пакет или дропается. Например, cilium monitor покажет, что пакет от Pod A к Pod B дропнут из-за политики, или не прошёл. Если не используется Cilium — можно задействовать BCC скрипты, например, `dropsnoop` (показывает причины дропа пакетов)
- **Базовые возможности kubectl для просмотра событий и тд:** `kubectl describe pod/svc` — часто содержит важные сведения (под не смог прикрепиться к сети, или ошибки EndpointSlice); `kubectl get events --all-namespaces | grep ...` – быстрый способ увидеть, не было ли аномалий

Частые проблемы с сетью в Kubernetes и их решения:

- **DNS не работает:** скажем, поды не резолвят сервисы или внешние адреса. Решение: убедиться, что CoreDNS запущен. Проверить, не перегружен ли (метрика latency). Посмотреть ConfigMap coredns — возможно, неверно настроен forward. Если DNS-пакеты не доходят — проверить iptables: kube-proxy должен перехватывать UDP 53 на kube-dns IP. Можно выполнить `nslookup kubernetes.default <DNSServiceIP>` из пода — если нет ответа, проверять логи coredns. В ряде случаев помогает старый добрый рестарт CoreDNS
- **Ingress не доступен снаружи**: например, развернули Nginx Ingress, а по внешнему IP достучаться до него не получается. Здесь несколько возможных причин:

    - (а) забыли аннотацию класса ingress — и Kubernetes мог не привязать Ingress к контроллеру. Например, для Nginx нужно `kubernetes.io/ingress.class: "nginx"` (для старых версий) или `ingressClassName: nginx`.
    - (б) В Service контроллера нет внешнего IP – в облаке проверить статус LoadBalancer (иногда он Pending из-за квот или subnet issues).
    - (в) Firewall: на облаке может блокировать порт (в GKE приватном нужно создать правилко firewall для входящих на NodePort). Решение: проверить `kubectl describe ingress` на события – часто там пишется, если Ingress не привязался. Также `kubectl get svc -n ingress-nginx` – виден ли External-IP. Если нет – смотреть события на сервисе.
- **Поды на разных узлах не могут взаимодействовать (Flannel):** когда-то была проблема с Flannel — после перезапуска нод поды в Flannel-сети не могли коммуницировать, пока не перезапустишь flanneld. Это известный баг, который исправлялся обновлением Flannel. С остальным CNI я с таким не сталкивался, но общий подход такой: если рабочая сеть сломалась после обновления — можно первым делом проверить changelog плагина, возможно это баг.
- **Calico NetworkPolicy ведёт себя странно:** иногда после удаления/изменения NetworkPolicy могут остаться правила iptables, блокирующие трафик. Диагностировать такую проблему можно с calicoctl diags, также можно перезапустить calico-agent на нодах, чтобы он почистил правила. Calico в новых версиях стал лучше, вроде таких проблем больше не было.
- **Node NotReady, CNI plugin not initialized:** Когда нода регистрируется, kubelet ждёт, что CNI настроит loopback интерфейс и т.д. Если CNI подвис, узел показывает NotReady с ошибкой `CNI plugin not initialized`. Частая причина — не совпадает Pod CIDR: например, kubeadm-кластер развернули с `PodCIDR=10.244.0.0/16`, а поставили Calico с значением по умолчанию `192.168.0.0/16`. Решение: убедиться, что подсеть пода в параметрах API-сервера и в ConfigMap CNI совпадают. Для Flannel это флаг `--pod-network-cidr` в команде `kubeadm init` и `net-conf.json` в DaemonSet flannel. Для Calico — ConfigMap с блоком IPPool.

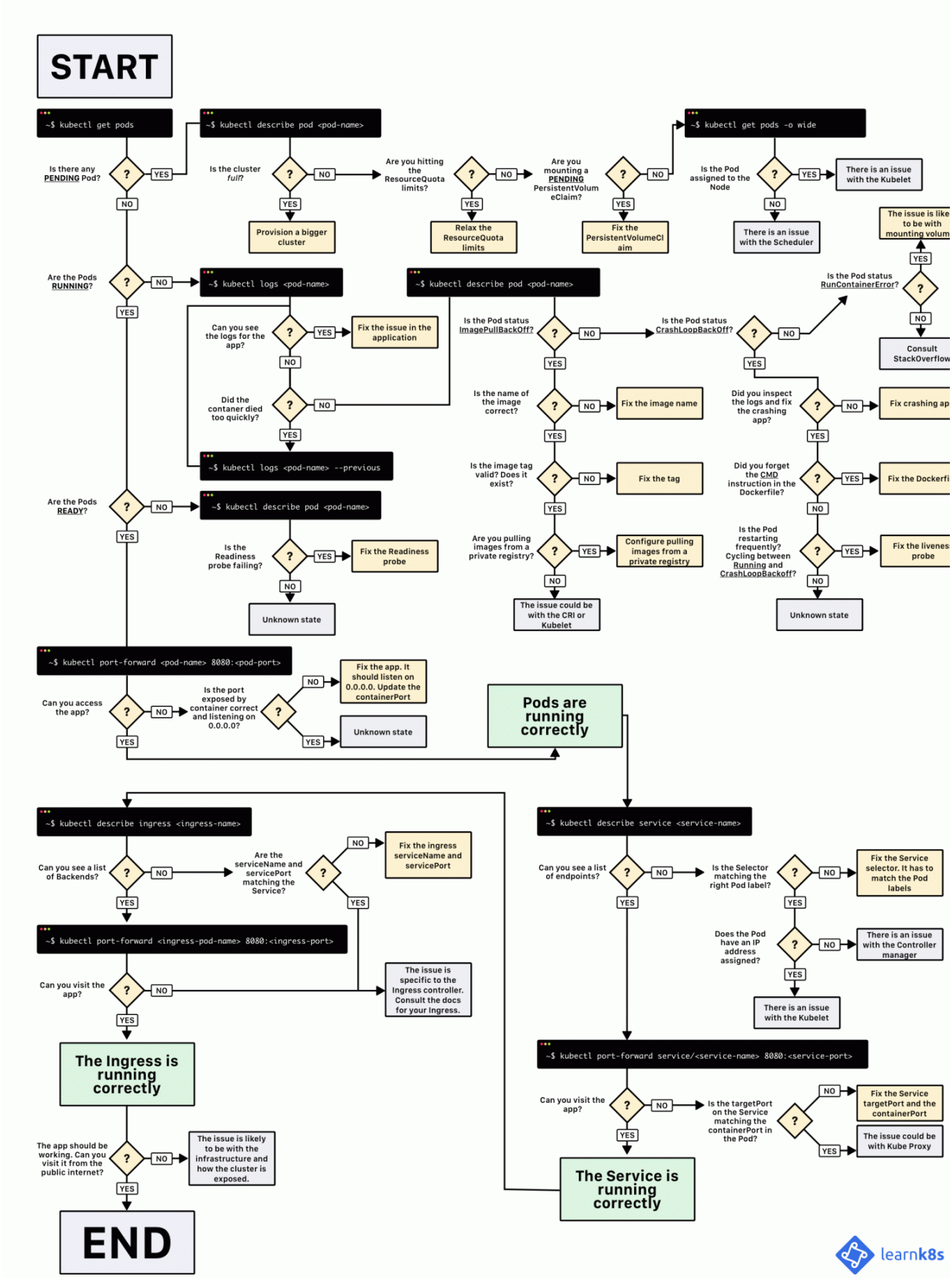
Отладка сети — это зачастую процесс исключения. Поэтому keep calm и методично проверяем: DNS? policy? iptables? CNI? Каждая из этих подсистем может стать причиной проблем с сетью. Из своего опыта могу сказать, что большинство "сетевых" проблем в Kubernetes — это или неправильно настроенные политики/Ingress, или баг/ограничение CNI-плагина, либо ошибка в конфигурации. Хорошая новость: сейчас сетевой стек Kubernetes достаточно стабильный и прозрачный. Инструменты вроде Network Policy Editor, Hubble UI (для Cilium) или встроенных kubernetes/debug команд облегчают жизнь, пользуйтесь)

Что ещё почитать по теме:

- A Guide to the Kubernetes Networking Model — не самая свежая статья, но подробно объясняет работу сети в Kubernetes, и глобально почти ничего не поменялось
- Визуальное руководство по диагностике неисправностей в Kubernetes — схема траблшутинга Kubernetes, прикладываю это изображение ниже (ссылка на оригинал

)

- Simplifying multi-clusters in Kubernetes — незатронутая в рамках этой статьи тема межкластерного взаимодействия
- A visual guide to Kubernetes networking fundamentals — на пальцах объясняется работа сети в Kubernetes, некоторые картинки я позаимствовал отсюда
- Ну и официальная документация, конечно, и для Service Mesh: Cilium Service Mesh, Istio service mesh, и для всего остального



Revision #2
Created 2025-11-26 17:36:57 UTC by Эд Кукса

Updated 2025-11-27 18:33:29 UTC by Эд Кукса