CNF

2026.03

Kubernetes DNS 아키텍처 백서:
CoreDNS 운영 가이드

CoreDNS는 Kubernetes 클러스터 내부에서 Service/Pod 이름을 IP로 변환하는 필수 DNS 컴포넌트로, 서비스 간 통신의 기반을 제공합니다.
MSA 환경에서는 IP가 아닌 Service 이름 기반 통신이 표준이므로, CoreDNS 없이는 서비스 디스커버리가 불가능합니다.
CoreDNS 장애는 곧 클러스터 전체 통신 장애로 직결되므로, 구조와 동작 원리를 이해하는 것이 운영 안정성의 핵심입니다.

hello@cncf.co.kr

02-469-5426

www.cncf.co.kr

Contents

1장: IP 주소가 아닌 서비스 이름으로 호출해야 하는 이유

1.1 Kubernetes에서 IP 기반 호출이 불가능한 이유

Kubernetes 환경에서 마이크로서비스 간의 통신은 기존의 IP 주소 기반 방식과는 근본적으로 다릅니다. 이는 Kubernetes가 클라우드 네이티브 환경에 맞게 설계되었기 때문이며, Pod의 생성과 삭제, 스케일링, 롤링 업데이트 등 다양한 이벤트가 빈번하게 발생함에 따라 각 Pod의 IP 주소가 지속적으로 변화하게 됩니다. 이러한 변화는 시스템의 안정성과 확장성을 보장하는 데 필수적이지만, IP 주소를 하드코딩하거나 수동으로 관리하는 방식은 즉각적으로 무효화되어 심각한 장애를 초래할 수 있습니다. Kubernetes에서는 서비스 디스커버리와 통신의 신뢰성을 확보하기 위해 DNS 기반의 서비스 이름 호출 방식을 도입하였으며, 이 핵심 인프라 역할을 CoreDNS가 담당합니다. 본 절에서는 IP 기반 호출의 한계와 CoreDNS 부재 시 발생하는 장애 시나리오를 구체적으로 설명합니다.

1.1.1 Pod IP의 임시성과 동적 스케일링의 현실

Kubernetes에서 Pod는 컨테이너화된 워크로드의 최소 단위로, 스케일링, 롤링 업데이트, 노드 장애 복구 등 다양한 이벤트에 따라 생성과 삭제가 반복됩니다. 이 과정에서 각 Pod는 새로운 IP를 할당받게 되며, 기존 IP는 즉시 무효화됩니다. 예를 들어, Horizontal Pod Autoscaler(HPA)로 인해 Pod 수가 1개에서 10개로 늘어나면 각 Pod는 고유한 IP를 가지지만, 이 IP들은 언제든 변경될 수 있습니다. 롤링 업데이트나 노드 장애 시에도 Pod는 새로운 IP를 할당받으며, 서비스 소비자는 이 변화를 실시간으로 추적할 수 없습니다.

애플리케이션 코드나 설정에서 Pod의 IP를 하드코딩하는 경우, Pod가 재생성되거나 스케일링 될 때마다 IP가 바뀌어 호출이 실패하게 됩니다. 대규모 클러스터에서는 수십~수백 개의 서비스와 수천 개의 Pod가 존재하므로, 모든 IP를 수동으로 관리하는 것은 운영적으로 불가능합니다. 특히 스케일링 이벤트가 자주 발생하는 환경에서는 IP 기반 호출이 즉시 무효화되어 서비스 간 통신이 중단될 위험이 큽니다.

Kubernetes의 본질은 동적 스케일링과 자동화에 있습니다. Pod의 수가 자동으로 늘어나거나 줄어드는 상황에서, IP 기반 호출은 새로 생성된 Pod를 인식하지 못하며, 기존 호출 경로가 무효화

됩니다. 반면, 서비스 이름 기반 호출은 CoreDNS를 통해 서비스 이름을 ClusterIP로 해석하여, Pod의 IP 변화와 무관하게 항상 최신 엔드포인트로 연결됩니다. 이러한 구조는 마이크로서비스 환경에서 필수적인 동적 디스커버리와 자동화된 통신을 보장합니다.

실제 운영 환경에서는 Pod의 IP가 예측 불가능하게 변경되는 사례가 빈번하게 발생합니다. 예를 들어, 클러스터의 리소스 부족으로 인해 Pod가 다른 노드로 이동하면 새로운 IP가 할당되며, 기존에 IP를 참조하던 서비스는 더 이상 해당 Pod에 접근할 수 없습니다. 또한, Kubernetes는 장애 복구를 위해 Pod를 자동으로 재생성하는데, 이때마다 IP가 달라지므로 IP 기반 호출은 지속적인 장애를 야기합니다. 이러한 문제는 특히 대규모 클러스터에서 더욱 두드러지며, 운영팀은 IP 변경을 실시간으로 추적하고 대응하는 데 막대한 비용과 시간을 소모하게 됩니다. 반면, 서비스 이름 기반 호출은 이러한 복잡성을 완전히 제거하여, 운영 효율성과 안정성을 극대화합니다.

1.1.2 CoreDNS가 없는 Kubernetes에서 발생하는 장애 시나리오

CoreDNS는 Kubernetes 클러스터에서 서비스 이름을 IP로 변환하는 핵심 인프라입니다. 만약 CoreDNS가 없거나 장애가 발생하면, Pod 간의 서비스 이름 기반 통신이 완전히 중단됩니다. 이는 모든 마이크로서비스 간 연결이 끊어져 클러스터 전체가 사실상 동작 불가능한 상태가 되는 것을 의미합니다.

Headless Service는 StatefulSet과 결합하여 개별 Pod에 직접 접근할 수 있도록 설계되었습니다. CoreDNS가 없으면 Headless Service의 DNS 레코드가 생성되지 않아, 데이터베이스나 Kafka 등 상태 유지가 필요한 워크로드에서 개별 Pod 접근이 불가능해집니다. 이는 클러스터의 안정성과 데이터 일관성에 심각한 영향을 미칩니다.

Pod가 외부 인터넷에 접근할 때도 CoreDNS를 통해 DNS 해석이 이루어집니다. CoreDNS가 없으면 외부 도메인에 대한 DNS 쿼리가 실패하며, 외부 API 호출이나 외부 서비스 연동이 모두 중단됩니다. 이는 클러스터 내부뿐 아니라 외부와의 통신에도 치명적인 장애를 유발합니다.

Kubernetes는 서비스 정보를 환경변수로 Pod에 주입하는 방식도 지원하지만, 이 방식은 Service가 Pod보다 먼저 존재해야 하며, 동적 업데이트가 불가능합니다. 즉, 서비스가 변경되거나 스케일링될 때 환경변수는 자동으로 갱신되지 않아, 실시간 디스커버리가 불가능합니다. CoreDNS 기반의 DNS 호출은 이러한 한계를 극복하며, Pod 재시작 없이도 서비스 변경을 즉시 반영할 수 있습니다.

실제 장애 사례를 살펴보면, CoreDNS가 다운되거나 네트워크 분리로 인해 DNS 쿼리가 실패하는 경우, 클러스터 내 모든 서비스 간 통신이 일시적으로 중단됩니다. 예를 들어, 대규모 데이터 처리 워크로드에서 StatefulSet 기반 데이터베이스가 Headless Service를 통해 Pod 간 직접 통신을 수행하는데, CoreDNS 장애로 인해 DNS 레코드가 해석되지 않으면 데이터베이스 샤드 간 연결이 끊어져 데이터 일관성에 심각한 문제가 발생할 수 있습니다. 또한, 외부 API 연동이 필요한 마이크로서비스가 DNS 해석 실패로 인해 외부 시스템과의 통신이 불가능해지면, 전체 서비스의 가용성이 저하되고 비즈니스 장애로 이어질 수 있습니다. 이러한 사례는 CoreDNS의 중요성을 강조하며, DNS 인프라의 안정적 운영이 Kubernetes 환경에서 필수임을 보여줍니다.

1.2 서비스 이름 기반 호출의 작동 원리

서비스 이름 기반 호출은 Kubernetes의 Service 추상화 계층과 CoreDNS의 DNS 해석 기능이 결합되어 구현됩니다. Kubernetes Service는 동적 Pod 집합 앞에 안정적인 ClusterIP와 DNS 이름을 제공하여, IP 하드코딩 없이도 서비스 간 통신이 투명하게 이루어지도록 설계되었습니다. CoreDNS는 Kubernetes API를 실시간으로 watch하여 서비스 이름을 IP로 변환하며, 스케일링, 롤링 업데이트, 네임스페이스 격리, 크로스 클러스터 등 다양한 상황에서도 코드 변경 없이 안정적인 호출이 가능합니다. 본 절에서는 서비스 이름 기반 호출의 구조와 IP 기반 호출과의 비교를 통해, IT 의사결정자가 서비스 이름 전환의 필요성을 한눈에 파악할 수 있도록 설명합니다.

1.2.1 Kubernetes Service 추상화 계층과 DNS의 관계

Kubernetes Service는 여러 Pod를 하나의 논리적 엔드포인트로 묶어주는 추상화 계층입니다. Service는 label selector를 통해 동적 Pod 집합을 관리하며, 각 Service에는 고정된 ClusterIP와 DNS 이름이 할당됩니다. 이 구조는 Pod의 생성과 삭제, IP 변화와 무관하게 항상 동일한 엔드포인트로 접근할 수 있도록 보장합니다.

CoreDNS는 Kubernetes API를 watch하여 Service와 Endpoint 정보를 실시간으로 반영합니다. 각 Service는 `<service-name>.<namespace>.svc.cluster.local` 형식의 DNS 이름을 가지며, Pod는 이 이름을 통해 Service에 접근할 수 있습니다. IP 하드코딩이 필요 없으며, Pod가 재생성되거나 스케일링되어도 DNS 이름은 변하지 않습니다. CoreDNS가 DNS 쿼리를 받아 Service의 최신 ClusterIP 또는 Pod IP 목록을 반환함으로써, 코드 변경 없이도 동적 서비스

호출이 가능합니다.

서비스 이름 기반 호출은 스케일링이나 롤링 업데이트 시에도 코드 변경이 필요 없습니다. Service의 DNS 이름은 항상 동일하며, CoreDNS가 최신 엔드포인트로 자동 라우팅합니다. 이는 마이크로서비스 환경에서 빠른 배포와 확장성을 보장하는 핵심 구조입니다.

기술적으로, Kubernetes Service는 Endpoints 리소스를 통해 실제 Pod의 IP 목록을 관리합니다. CoreDNS는 이 Endpoints 정보를 실시간으로 감시(watch)하여, DNS 쿼리가 들어오면 해당 Service에 연결된 최신 Pod IP를 반환합니다. 예를 들어, 서비스 이름이 `web`이고 네임스페이스가 `prod`라면, Pod는 `web.prod.svc.cluster.local`을 호출하여 항상 최신 엔드포인트로 연결됩니다. 이 방식은 Pod의 IP가 변경되더라도 서비스 이름만으로 안정적으로 접근할 수 있게 하며, 운영자가 별도의 IP 관리나 코드 수정 없이 서비스 확장과 배포를 자유롭게 수행할 수 있도록 지원합니다.

또한, DNS 기반 서비스 디스커버리는 네임스페이스별 격리와 크로스 클러스터 통합에도 유리합니다. 네임스페이스마다 고유한 DNS 네임스페이스가 생성되어, 서비스 이름 충돌이나 IP 충돌 없이 논리적 격리를 구현할 수 있습니다. Federation 플러그인을 활용하면 여러 클러스터 간에도 통합된 DNS 네임스페이스를 구축할 수 있어, 대규모 멀티 클러스터 환경에서도 서비스 이름 기반 호출의 장점을 극대화할 수 있습니다.

1.2.2 IP 기반 vs 서비스 이름 기반 호출 비교

Pod 재시작, 스케일링, 롤링 업데이트 등 다양한 운영 이벤트가 발생할 때 IP 기반 호출과 서비스 이름 기반 호출은 근본적으로 다른 동작 방식을 보입니다. IP 기반 호출에서는 Pod의 IP가 변경될 때마다 호출 실패가 발생하며, 운영자가 수동으로 IP를 갱신해야 하므로 관리 부담과 장애 위험이 큽니다. 반면, 서비스 이름 기반 호출은 CoreDNS가 자동으로 최신 엔드포인트로 라우팅하여, 코드 수정이나 운영 개입 없이 안정적인 통신이 가능합니다.

아래 표는 대표적인 상황에서 두 방식의 차이를 정리한 것입니다.

상황	IP 기반 호출	서비스 이름 기반 호출 (CoreDNS)
Pod 재시작	IP 변경 → 호출 실패	서비스 이름 유지 → 자동 해석
스케일링	새 Pod IP 수동 등록 필요	자동 엔드포인트 업데이트
롤링 업데이트	구/신 IP 수동 전환 필요	투명하게 새 Pod으로 라우팅

네임스페이스 격리	IP 충돌 위험	<svc>.<ns>.svc.cluster.local로 자연스러운 격리
크로스 클러스터	IP 충돌 위험	Federation 플러그인으로 통합 가능

서비스 이름 기반 호출은 네임스페이스별로 논리적 격리를 제공하며, 크로스 클러스터 환경에서도 Federation 플러그인을 통해 통합된 DNS 네임스페이스를 구현할 수 있습니다. IP 기반 호출은 이러한 격리와 통합이 어렵고, 충돌 위험이 높습니다.

IT 의사결정자 관점에서 서비스 이름 기반 호출은 운영 효율성과 안정성, 확장성, 코드 유지보수 측면에서 IP 기반 호출 대비 압도적인 우위를 제공합니다. Kubernetes 환경에서는 서비스 이름 기반 호출이 선택이 아닌 필수이며, CoreDNS가 이 구조의 핵심 인프라임을 명확히 인식해야 합니다.

실제 사례를 통해 비교해 보면, 대규모 금융사의 Kubernetes 클러스터에서 IP 기반 호출을 사용하던 시절에는 Pod 재시작이나 스케일링 이벤트가 발생할 때마다 서비스 간 통신 장애가 빈번하게 발생하였고, 운영팀은 IP 변경을 추적하고 수동으로 수정하는 데 많은 시간을 소모했습니다. 서비스 이름 기반 호출로 전환한 이후에는 이러한 장애가 완전히 사라졌으며, 신규 서비스 배포나 확장도 코드 수정 없이 즉시 가능해져 운영 효율성이 크게 향상되었습니다. 또한, 네임스페이스별 격리와 Federation 플러그인을 활용한 크로스 클러스터 통합으로, 멀티 클러스터 환경에서도 안정적인 서비스 호출이 가능해졌습니다. 이처럼 서비스 이름 기반 호출은 Kubernetes 환경에서 신뢰성과 확장성을 보장하는 핵심 기술임을 실제 운영 사례를 통해 확인할 수 있습니다.

1.3 CoreDNS의 탄생과 Kubernetes 표준 DNS로의 채택

CoreDNS는 Kubernetes의 서비스 디스커버리 문제를 해결하기 위해 탄생한 경량, 플러그인 기반 DNS 서버입니다. 기존 DNS 서버(BIND, SkyDNS, kube-dns)의 한계를 극복하고, 클라우드 네이티브 환경에 최적화된 설계 철학을 바탕으로 개발되었습니다. CoreDNS는 2016년 시작된 이후 CNCF Graduated를 달성하며, Kubernetes의 유일 공식 DNS로 자리잡았습니다. 본 절에서는 CoreDNS의 탄생 배경과 설계 철학, 그리고 Kubernetes 표준 DNS로 채택된 타임라인과 기술 성숙도 지표를 정리합니다.

1.3.1 CoreDNS의 시작 배경과 설계 철학

CoreDNS는 Miek Gieben(Google SRE 엔지니어)에 의해 2016년 3월 개발되었습니다. 기존 Go DNS 라이브러리와 DNS 서버 SkyDNS의 경험을 바탕으로, Go 기반 웹 서버 Caddy(0.8 버전)를 포크하여 DNS 서버로 재설계하였습니다. Caddy의 간결한 설정 구문과 플러그인 기반 아키텍처를 계승하여, 모든 DNS 기능을 플러그인으로 분리하고 핵심(Core) 엔진만 남긴 미니멀 설계 철학을 구현했습니다.

BIND 등 전통적 DNS 서버는 모놀리식 구조와 복잡한 설정, 높은 리소스 요구로 컨테이너 환경에 부적합했습니다. SkyDNS는 etcd에 의존하는 한계가 있었고, kube-dns는 dnsmasq, SkyDNS, sidecar 등 3개 컨테이너로 운영 복잡성이 높았습니다. 클라우드 네이티브 환경에 적합한 경량, 플러그인 기반 DNS 서버가 부재했습니다.

CoreDNS는 “핵심만 남기고 나머지는 플러그인”이라는 설계 철학을 바탕으로, 모든 기능을 플러그인으로 구현하여 확장성과 경량성을 극대화했습니다. 약 30개 내장 플러그인과 15개 이상 외부 플러그인을 지원하며, Corefile 기반의 간결한 설정 구문으로 운영 효율성을 높였습니다.

CoreDNS는 Apache License 2.0으로 배포되어, 수정, 배포, 상용화에 제약이 없습니다. 소스 코드 공개 의무도 없으며, 모든 주요 클라우드 벤더(AWS EKS, Azure AKS, Google GKE)가 매니지드 Kubernetes에 기본 포함하고 있습니다.

CoreDNS의 설계 철학은 클라우드 네이티브 환경에서의 확장성과 유연성을 극대화하는 데 중점을 두고 있습니다. 플러그인 아키텍처 덕분에 사용자는 필요에 따라 DNS 기능을 자유롭게 추가하거나 제거할 수 있으며, 운영 환경에 맞는 맞춤형 DNS 서버를 쉽게 구축할 수 있습니다. 예를 들어, Kubernetes 플러그인을 활성화하면 클러스터 내 서비스 이름 해석이 가능해지고, hosts 플러그인을 추가하면 특정 호스트 파일 기반 DNS 해석도 지원할 수 있습니다. 또한, Corefile 설정은 YAML이나 JSON이 아닌 간결한 텍스트 형식으로 구성되어, 운영자가 복잡한 설정 없이 빠르게 DNS 정책을 적용할 수 있습니다.

기존 DNS 서버와 비교했을 때, CoreDNS는 메모리 사용량이 매우 적고, 컨테이너 환경에서 빠른 배포와 확장이 가능합니다. 실제로 kube-dns 대비 메모리 사용량이 11.6배 절감되고, 외부 이름 해석 속도가 3배 향상되는 등 정량적 성능 우위가 입증되었습니다. 이러한 장점은 대규모 클러스터 운영에서 매우 중요한 요소로 작용하며, CoreDNS가 Kubernetes의 표준 DNS로 채택된 배경이 되었습니다.

1.3.2 CNCF Graduated에서 Kubernetes 유일 표준까지

CoreDNS의 발전 과정은 CNCF 타임라인과 Kubernetes 표준 채택 과정에서 명확하게 드러납니다. 2016년 3월 프로젝트가 시작된 이후, 2017년 2월 CNCF Sandbox에 가입하고, 2018년 2월 CNCF Incubating 단계로 승격되었습니다. 2018년 12월 Kubernetes 1.13 버전에서 CoreDNS가 기본 DNS 서버로 채택되었으며, 2019년 1월 CNCF Graduated 등급을 달성하여 공식적으로 성숙도가 검증된 오픈소스 프로젝트가 되었습니다. 이후 2021년 Kubernetes 1.21 버전에서 kubeadm이 kube-dns 지원을 공식적으로 제거하고, 현재 Kubernetes 1.35 버전에서는 kubeadm이 지원하는 유일한 DNS 애플리케이션으로 CoreDNS가 자리잡았습니다.

기술 성숙도 지표를 보면, CoreDNS는 GitHub에서 15,493개의 Stars를 기록하고 있으며, 전체 기여자는 19,033명, 기여 조직은 4,803개에 달합니다. CNCF Slack의 #coredns 채널에는 485명 이상의 전문가가 활동하고 있으며, 메인테이너는 16명으로 구성되어 있습니다. 모든 주요 클라우드 플랫폼(AWS EKS, Azure AKS, Google GKE)에서 CoreDNS가 기본 탑재되어, 클라우드 네이티브 환경에서 표준 DNS로서의 위치를 확고히 하고 있습니다.

CoreDNS가 Kubernetes의 표준 DNS로 채택된 것은 단순한 기능적 우위뿐 아니라, 운영 효율성과 확장성, 보안성, 커뮤니티 지원 등 다양한 측면에서 기술 성숙도가 입증되었기 때문입니다. 예를 들어, CoreDNS는 kube-dns 대비 메모리 사용량이 11.6배 절감되고, 외부 이름 해석 속도가 3배 향상되는 등 정량적 성능 우위를 제공합니다. 또한, 플러그인 기반 아키텍처 덕분에 다양한 DNS 정책과 기능을 자유롭게 적용할 수 있어, 대규모 클러스터 운영에 최적화된 DNS 인프라를 구축할 수 있습니다.

실제 운영 사례를 보면, 글로벌 클라우드 벤더들은 CoreDNS를 매니지드 Kubernetes 서비스(EKS, AKS, GKE)에 기본 포함시키고 있으며, 별도의 DNS 서버 도입이나 설정이 필요 없이 자동으로 배포됩니다. 이로 인해 운영자는 DNS 인프라 관리 부담을 크게 줄일 수 있으며, 클러스터 확장이나 서비스 배포 시에도 DNS 장애 없이 안정적인 서비스 디스커버리를 구현할 수 있습니다. CoreDNS의 CNCF Graduated 달성 이후, Kubernetes 환경에서는 DNS 인프라 선택이 아닌 표준으로 자리잡았으며, 앞으로도 클라우드 네이티브 환경에서 핵심 인프라로 지속적으로 발전할 전망입니다.

2장: CoreDNS가 서비스 이름을 IP로 변환하는 메커니즘

2.1 Kubernetes DNS 해석 흐름

Kubernetes 환경에서 서비스 이름 기반 호출이 가능하도록 만드는 핵심 인프라가 바로 CoreDNS이다. Pod 내부 애플리케이션이 서비스 이름을 사용해 네트워크 통신을 시도할 때, 이 이름이 실제로 어떤 IP로 변환되는지, 그 과정에서 CoreDNS가 어떻게 동작하는지 이해하는 것은 클라우드 네이티브 아키텍처 설계의 출발점이다. 이 절에서는 Pod에서 시작된 DNS 쿼리가 CoreDNS까지 전달되는 경로와, 각 단계에서의 주요 동작 원리를 상세히 설명한다. 특히 NodeLocal DNSCache와 ndots:5 옵션이 성능과 해석 방식에 미치는 영향, 네임스페이스별 호출 패턴의 기술적 배경을 집중적으로 다룬다.

2.1.1 Pod에서 CoreDNS까지의 DNS 쿼리 경로

Kubernetes 클러스터 내에서 Pod가 네트워크 통신을 위해 서비스 이름을 호출할 때, DNS 쿼리는 일련의 경로를 따라 전달됩니다. Pod 내부의 애플리케이션은 일반적으로 `myservice` 또는 `myservice.default`와 같은 이름으로 요청을 보냅니다. 이 요청은 먼저 Pod의 `/etc/resolv.conf` 파일에 명시된 `nameserver`로 전달되는데, 이 값은 CoreDNS의 ClusterIP(예: 10.96.0.10)로 설정되어 있습니다. 만약 NodeLocal DNSCache가 배포되어 있다면, 쿼리는 먼저 로컬 캐시(169.254.20.10)를 확인한 후, 필요 시 CoreDNS로 전달됩니다. CoreDNS는 cluster.local 도메인에 대해 kubernetes 플러그인으로 처리하고, 외부 도메인에 대해서는 forward 플러그인을 통해 업스트림 DNS로 포워딩합니다.

이러한 DNS 쿼리 경로는 Kubernetes의 네트워크 추상화와 서비스 디스커버리의 핵심을 이루며, 클러스터 내에서 서비스 이름만으로 통신이 가능하게 하는 기반입니다. 실제 운영 환경에서는 DNS 쿼리의 지연, 부하 분산, 네임스페이스별 호출 패턴 등 다양한 요소가 복합적으로 작용합니다. 특히 NodeLocal DNSCache를 활용하면 DNS 쿼리 처리 속도가 크게 향상되고, CoreDNS의 부하가 효과적으로 분산됩니다. cluster.local 도메인과 외부 도메인에 대한 처리 방식의 차이도 클러스터 내부 트래픽과 외부 트래픽을 효율적으로 관리하는 데 중요한 역할을 합니다. 아래에서 각 단계별 동작 원리와 기술적 세부사항을 자세히 살펴보겠습니다.

DNS 쿼리 경로 전체 흐름

Kubernetes 클러스터 내에서 Pod가 네트워크 통신을 위해 서비스 이름을 호출할 때, DNS 쿼리는 일련의 경로를 따라 전달된다. Pod 내부의 애플리케이션은 일반적으로 `myservice` 또는 `myservice.default`와 같은 이름으로 요청을 보낸다. 이 요청은 먼저 Pod의 `/etc/resolv.conf` 파일에 명시된 `nameserver`로 전달되는데, 이 값은 CoreDNS의 ClusterIP(예: 10.96.0.10)로 설정된다. 만약 NodeLocal DNSCache가 배포되어 있다면, 쿼리는 먼저 로컬 캐시(169.254.20.10)를 확인한 후, 필요 시 CoreDNS로 전달된다. CoreDNS는 cluster.local 도메인에 대해 kubernetes 플러그인으로 처리하고, 외부 도메인에 대해서는 forward 플러그인을 통해 업스트림 DNS로 포워딩한다.

NodeLocal DNSCache의 역할

NodeLocal DNSCache는 대규모 클러스터에서 DNS 쿼리 지연을 최소화하고 CoreDNS의 부하를 분산시키는 역할을 한다. 각 노드에 별도의 DNS 캐시 Pod를 배포하여, 자주 요청되는 DNS 쿼리를 로컬에서 빠르게 처리한다. 이로 인해 DNS 쿼리 지연이 밀리초 단위에서 마이크로초 단위로 감소하며, CoreDNS의 QPS(Queries Per Second) 부하가 70~90%까지 줄어든다. 특히 100노드 이상 대규모 환경에서는 필수적으로 적용되는 운영 전략이다.

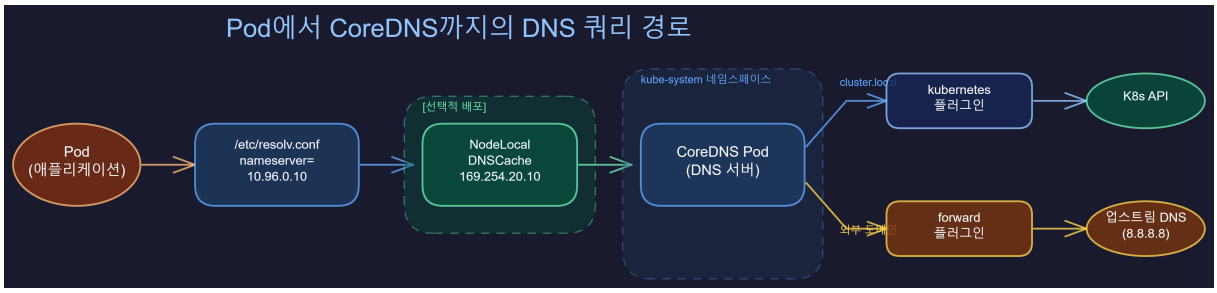
NodeLocal DNSCache는 실제로 각 노드에 DaemonSet 형태로 배포되며, Pod의 `/etc/resolv.conf` 파일에서 nameserver를 NodeLocal DNSCache의 IP(예: 169.254.20.10)로 지정합니다. 이 방식은 CoreDNS가 중앙 집중적으로 처리하던 DNS 쿼리를 각 노드에서 분산 처리하게 하여, 대규모 트래픽 환경에서도 DNS 응답 속도를 안정적으로 유지할 수 있습니다. 또한, DNSCache는 클러스터 내에서 자주 반복되는 쿼리를 캐싱하여, 불필요한 네트워크 트래픽과 CoreDNS의 과부하를 방지합니다. 실제 운영 사례에서는 NodeLocal DNSCache 도입 후 CoreDNS의 QPS가 70~90% 감소하고, DNS 응답 지연이 10배 이상 개선되는 효과가 보고되었습니다.

cluster.local과 외부 도메인 처리 방식

CoreDNS는 cluster.local 도메인(내부 서비스)과 외부 도메인(예: google.com)에 대해 서로 다른 플러그인 체인을 사용한다. cluster.local 쿼리는 kubernetes 플러그인이 K8s API를 통해 서비스 정보를 실시간으로 조회하여 IP를 반환한다. 외부 도메인 쿼리는 forward 플러그인이 업스트림 DNS(예: 8.8.8.8)로 요청을 전달한다. 이 분리된 처리 구조는 클러스터 내부와 외부 네트워크 트래픽을 효율적으로 관리하는 기반이 된다.

cluster.local 도메인에 대한 쿼리는 CoreDNS의 kubernetes 플러그인이 직접 처리하며, Kubernetes API를 통해 서비스와 Pod의 정보를 실시간으로 조회합니다. 이 과정에서 서비스 이름은 ClusterIP로, Headless Service의 경우 모든 Pod IP로 변환됩니다. 반면, 외부 도메인 쿼리는 CoreDNS의 forward 플러그인이 업스트림 DNS 서버(예: 8.8.8.8 또는 클라우드 DNS)로 전달하여, 외부 네트워크와의 통신을 담당합니다. 이 구조는 내부 서비스 디스커버리와 외부 API 호출을 명확하게 분리하여, 보안과 성능을 동시에 확보할 수 있도록 설계되었습니다.

DNS 해석 흐름 도식



[그림 1] Pod에서 CoreDNS까지의 DNS 쿼리 경로

2.1.2 /etc/resolv.conf와 ndots:5의 동작 원리

Kubernetes 환경에서 Pod의 DNS 해석 방식은 `/etc/resolv.conf` 파일과 ndots:5 옵션에 의해 결정됩니다. 이 파일은 kubelet이 Pod를 생성할 때 자동으로 주입하며, nameserver, search, options(ndots:5) 등 핵심 설정이 포함되어 있습니다. 이러한 설정은 서비스 이름 해석의 유연성과 네임스페이스별 호출 패턴을 가능하게 하며, 내부 서비스와 외부 도메인 호출 시 각각 다른 동작을 유발합니다. ndots:5 옵션은 DNS 쿼리 이름에 점이 5개 미만일 때 search 도메인을 붙여 여러 번 해석을 시도하게 하여, 네임스페이스별 논리적 격리와 서비스 호출의 투명성을 제공합니다. 하지만 외부 API 호출 시에는 불필요한 DNS 쿼리와 지연이 발생할 수 있으므로 운영 시 주의가 필요합니다. 아래에서 각 설정의 동작 원리와 기술적 세부사항을 자세히 설명하겠습니다.

Pod의 /etc/resolv.conf 자동 구성

Kubernetes는 Pod를 생성할 때 kubelet이 자동으로 `/etc/resolv.conf` 파일을 주입한다. 이 파일에는 다음과 같은 핵심 설정이 포함된다:

- `nameserver`: CoreDNS의 ClusterIP(예: 10.96.0.10)

- search: 네임스페이스별 도메인 확장(default.svc.cluster.local svc.cluster.local cluster.local)
- options: ndots:5

이 설정은 Pod가 DNS 쿼리를 보낼 때, 이름에 점(.)이 5개 미만이면 search 도메인을 순차적으로 붙여 여러 형태로 해석을 시도하도록 한다.

Pod의 /etc/resolv.conf 파일은 Kubernetes가 Pod를 생성할 때 자동으로 구성되며, 클러스터 환경에 맞는 DNS 해석 방식을 제공합니다. nameserver는 CoreDNS의 ClusterIP로 지정되어, 모든 DNS 쿼리가 CoreDNS로 전달됩니다. search 옵션은 네임스페이스별 도메인 확장을 가능하게 하여, 같은 네임스페이스 내에서는 서비스 이름만으로 호출이 가능합니다. ndots:5 옵션은 DNS 쿼리 이름에 점이 5개 미만일 경우, search 도메인을 붙여 여러 번 해석을 시도하게 하여, 네임스페이스별 호출 패턴을 지원합니다.

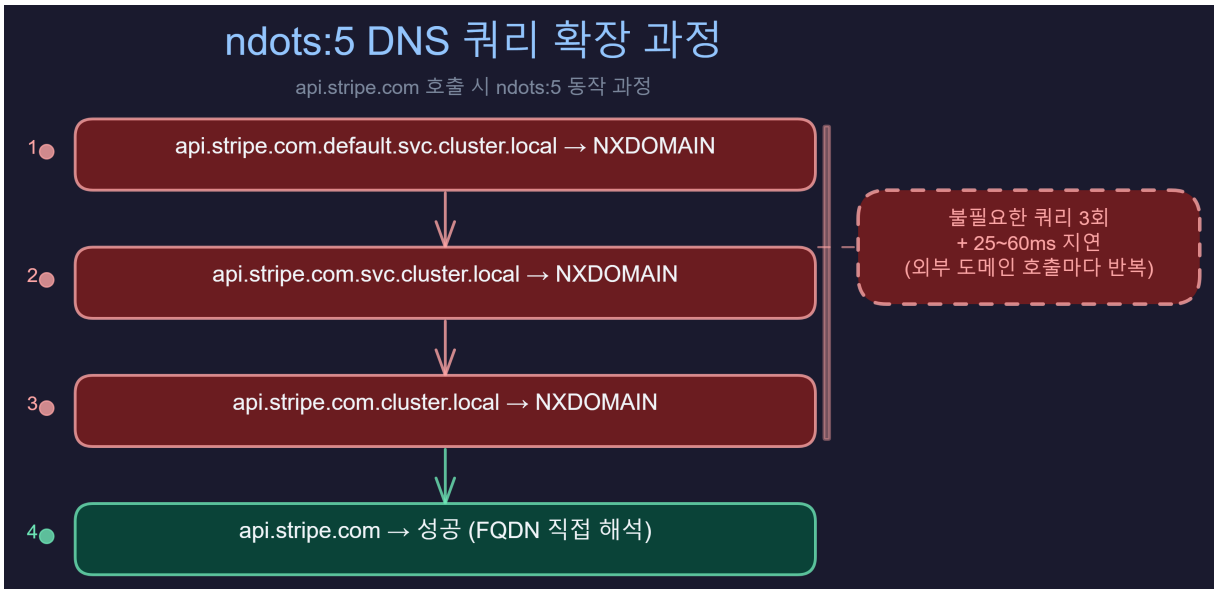
ndots:5 옵션의 의미와 영향

ndots:5는 DNS 쿼리 이름에 점이 5개 미만일 경우, search 도메인을 붙여서 여러 번 해석을 시도한다는 의미다. 예를 들어 myservice로 호출하면, 아래와 같은 순서로 쿼리가 발생한다:

1. myservice.default.svc.cluster.local
2. myservice.svc.cluster.local
3. myservice.cluster.local
4. myservice

이로 인해 같은 네임스페이스에서는 myservice만으로도 정확한 서비스 호출이 가능하다. 반면, 외부 도메인(예: api.stripe.com)은 점이 2개뿐이므로 search 도메인이 붙어 여러 번 해석 시도가 이뤄진다. 이 과정에서 불필요한 DNS 쿼리와 실패(NXDOMAIN)가 발생해 외부 API 호출 시 최대 25~60ms 지연이 추가될 수 있다.

ndots:5 옵션은 Kubernetes 환경에서 서비스 이름 해석의 유연성을 제공하지만, 외부 도메인 호출 시에는 search 도메인 확장으로 인해 불필요한 DNS 쿼리와 실패(NXDOMAIN)가 발생할 수 있습니다. 실제로 외부 API 호출 시 최대 25~60ms의 추가 지연이 발생하며, 이로 인해 애플리케이션 성능에 영향을 줄 수 있습니다. 운영 환경에서는 FQDN 끝에 점(.)을 추가하거나 ndots 값을 3으로 조정하여 이러한 지연을 최소화하는 전략이 권장됩니다.



[그림 2] ndots:5 DNS 쿼리 확장 과정|1087

네임스페이스별 호출 패턴과 search 도메인 확장

Kubernetes의 DNS search 도메인 확장 기능 덕분에, 같은 네임스페이스에서는 서비스 이름만으로 호출이 가능하다. 예를 들어 `payment-service`를 호출하면, 자동으로 `payment-service.default.svc.cluster.local`로 확장된다. 다른 네임스페이스의 서비스 호출 시에는 `db.backend` 또는 `db.backend.svc.cluster.local` 등 명시적으로 네임스페이스를 지정해야 한다. 이 구조는 네임스페이스별 논리적 격리와 서비스 호출의 투명성을 동시에 제공한다.

search 도메인 확장 기능은 Kubernetes의 네임스페이스별 논리적 격리와 서비스 호출의 투명성을 동시에 제공합니다. 같은 네임스페이스 내에서는 서비스 이름만으로 호출이 가능하며, 다른 네임스페이스의 서비스 호출 시에는 명시적으로 네임스페이스를 지정해야 합니다. 이 구조는 클러스터 내에서 서비스 디스커버리의 신뢰성과 운영의 편의성을 높여줍니다.

ndots:5 운영 시 주의사항

ndots:5 설정은 내부 서비스 호출에는 유리하지만, 외부 API 호출에는 불필요한 DNS 쿼리와 지연을 유발할 수 있다. 이를 해결하기 위해 FQDN 끝에 점(.)을 추가하거나, ndots 값을 3으로 조정하는 운영 전략이 권장된다. 대규모 트래픽 환경에서는 NodeLocal DNSCache와 조합하여 DNS 성능 최적화를 반드시 고려해야 한다.

운영 환경에서는 ndots:5 설정이 내부 서비스 호출에는 매우 유리하지만, 외부 API 호출 시에는 불필요한 DNS 쿼리와 지연이 발생할 수 있으므로 주의가 필요합니다. FQDN 끝에 점(.)을 추가하거나 ndots 값을 3으로 조정하면 이러한 문제를 해결할 수 있습니다. 또한, 대규모 트래픽

환경에서는 NodeLocal DNSCache와 조합하여 DNS 성능 최적화를 반드시 고려해야 하며, 실제 운영 사례에서는 이러한 전략이 DNS 응답 속도와 안정성을 크게 향상시키는 것으로 나타났습니다.

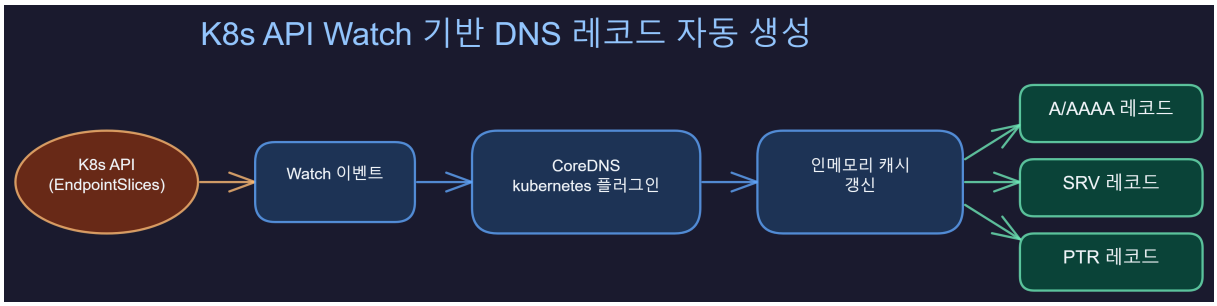
2.2 kubernetes 플러그인과 실시간 서비스 레코드 관리

CoreDNS가 Kubernetes 환경에서 서비스 이름을 IP로 변환하는 핵심 메커니즘은 kubernetes 플러그인에 의해 구현됩니다. 이 플러그인은 K8s API의 EndpointSlices를 실시간으로 watch하여 서비스와 Pod의 변경 사항을 감지하고, DNS 레코드를 자동으로 생성·관리합니다. Kubernetes의 동적 환경에서는 서비스와 Pod의 생성, 삭제, 스케일링이 빈번하게 발생하며, 이에 따라 DNS 레코드도 실시간으로 갱신되어야 안정적인 서비스 호출이 가능합니다. 이 절에서는 kubernetes 플러그인의 동작 원리와 서비스 FQDN, DNS 레코드 유형의 구조적 특징을 상세히 설명합니다. 특히, DNS 레코드 자동 생성 메커니즘과 캐시 동기화, 운영상의 장점에 대해 구체적으로 다루어 Kubernetes 서비스 디스커버리의 신뢰성과 확장성을 이해할 수 있도록 안내합니다.

2.2.1 K8s API Watch 기반 DNS 레코드 자동 생성

CoreDNS의 kubernetes 플러그인은 Kubernetes API의 `discovery.EndpointSlices` 리소스를 실시간으로 watch합니다. 이 API는 각 서비스의 엔드포인트(Pod IP, 포트 등)를 집합적으로 관리하며, 서비스 또는 Pod의 생성·삭제·스케일링 등 변경이 발생할 때마다 즉시 이벤트를 전달합니다. 플러그인은 이 이벤트를 받아 인메모리 캐시를 갱신하고, DNS 쿼리 요청 시 최신 정보를 기반으로 IP를 반환합니다.

Kubernetes 환경에서는 서비스와 Pod의 상태가 동적으로 변화하기 때문에, DNS 레코드도 실시간으로 갱신되어야 합니다. kubernetes 플러그인은 K8s API의 EndpointSlices 리소스를 watch하여, 서비스와 Pod의 생성, 삭제, 스케일링 등 다양한 이벤트를 감지합니다. 이 이벤트를 기반으로 인메모리 캐시를 갱신하고, DNS 쿼리 요청 시 최신 정보를 반환함으로써, IP 하드코딩 없이 안정적인 서비스 호출이 가능합니다.



[그림 3] K8s API Watch 기반 DNS 레코드 자동 생성

DNS 레코드 자동 생성 메커니즘

서비스나 Pod의 변경이 감지되면, kubernetes 플러그인은 다음과 같은 DNS 레코드를 자동으로 생성한다:

- **A/AAAA 레코드**: 서비스 이름 → ClusterIP, Headless Service → 모든 Pod IP 목록
- **SRV 레코드**: 서비스 포트 정보 제공
- **PTR 레코드**: 역방향 DNS 조회 지원

이 메커니즘은 Pod 재시작, 스케일링, 롤링 업데이트 등 동적 환경에서도 DNS 레코드가 실시간으로 업데이트되어, IP 하드코딩 없이 안정적인 서비스 호출이 가능하게 한다.

DNS 레코드 자동 생성 메커니즘은 Kubernetes의 동적 환경에서 매우 중요한 역할을 합니다. 서비스나 Pod의 변경이 감지되면, kubernetes 플러그인은 A/AAAA 레코드(서비스 이름 → ClusterIP, Headless Service → 모든 Pod IP 목록), SRV 레코드(서비스 포트 정보 제공), PTR 레코드(역방향 DNS 조회 지원) 등을 자동으로 생성합니다. 이 구조는 Pod 재시작, 스케일링, 롤링 업데이트 등 다양한 상황에서도 DNS 레코드가 실시간으로 갱신되어, IP 하드코딩 없이 안정적인 서비스 호출이 가능하게 합니다.

캐시 동기화와 서비스 시작 과정

CoreDNS는 시작 시 최대 5초간 K8s API를 watch하여 캐시를 동기화한다. 이 과정이 완료되기 전에는 DNS 쿼리에 대해 SERVFAIL을 반환할 수 있다. 캐시 동기화가 완료되면, 실시간 이벤트 기반으로 DNS 레코드가 갱신되어 서비스 이름 기반 호출이 정상적으로 작동한다. 이 구조는 서비스 디스커버리의 신뢰성과 성능을 동시에 보장한다.

CoreDNS는 시작 시 최대 5초간 K8s API를 watch하여 인메모리 캐시를 동기화합니다. 이 과정이 완료되기 전에는 DNS 쿼리에 대해 SERVFAIL을 반환할 수 있으며, 캐시 동기화가 완료되면

실시간 이벤트 기반으로 DNS 레코드가 갱신되어 서비스 이름 기반 호출이 정상적으로 작동합니다. 이 구조는 서비스 디스커버리의 신뢰성과 성능을 동시에 보장하며, 운영 환경에서 장애 대응과 스케일링이 매우 용이합니다.

운영상의 장점

K8s API watch 기반 DNS 레코드 자동 생성은 대규모 클러스터 환경에서 수십~수백 개의 서비스와 수천 개의 Pod이 동적으로 변화하는 상황에서도, 운영자의 수동 관리 부담을 완전히 제거한다. 서비스 이름 기반 호출은 코드 변경 없이 투명하게 작동하며, 장애 대응과 스케일링이 매우 용이하다.

운영상의 장점은 매우 큽니다. K8s API watch 기반 DNS 레코드 자동 생성은 대규모 클러스터 환경에서 수십~수백 개의 서비스와 수천 개의 Pod이 동적으로 변화하는 상황에서도 운영자의 수동 관리 부담을 완전히 제거합니다. 서비스 이름 기반 호출은 코드 변경 없이 투명하게 작동하며, 장애 대응과 스케일링이 매우 용이합니다. 실제 운영 사례에서는 서비스 디스커버리의 신뢰성과 확장성이 크게 향상되는 효과가 보고되었습니다.

2.2.2 서비스 FQDN 형식과 DNS 레코드 유형

Kubernetes에서 서비스 이름을 IP로 변환하는 과정은 FQDN(Full Qualified Domain Name) 형식과 다양한 DNS 레코드 유형에 의해 결정됩니다. 서비스, StatefulSet, Headless Service, Pod 등 다양한 리소스에 대해 각각 고유한 FQDN이 생성되며, CoreDNS는 이를 기반으로 ClusterIP, Pod IP, 포트 정보 등을 반환합니다. 이 구조는 Kubernetes 서비스 호출의 유연성과 확장성을 극대화하며, 상태 유지 워크로드와 외부 서비스 통합 등 다양한 요구에 대응할 수 있습니다. 아래에서 기본 서비스 FQDN 구조, StatefulSet과 Headless Service의 FQDN, Pod DNS 형식, SRV 레코드, DNS 레코드 유형의 특징을 자세히 설명하겠습니다.

기본 서비스 FQDN 구조

Kubernetes에서 서비스는 다음과 같은 FQDN(Full Qualified Domain Name) 형식으로 DNS 레코드가 생성된다:

- `<service-name>.<namespace>.svc.cluster.local`

예를 들어, `payment-service`가 `default` 네임스페이스에 있다면, FQDN은 `payment-service.default.svc.cluster.local`이 된다. 이 이름은 CoreDNS가 ClusterIP로 해석하여

Pod 간 통신을 가능하게 한다.

기본 서비스 FQDN 구조는 Kubernetes 환경에서 서비스 이름 기반 호출의 핵심입니다. 서비스 이름과 네임스페이스, svc, cluster.local 도메인을 조합하여 고유한 FQDN이 생성되며, CoreDNS는 이를 ClusterIP로 해석하여 Pod 간 통신을 가능하게 합니다. 이 구조는 서비스 디스커버리의 신뢰성과 운영의 편의성을 높여줍니다.

StatefulSet과 Headless Service의 FQDN

StatefulSet과 Headless Service를 사용하는 경우, 각 Pod에 대해 개별 FQDN이 생성된다:

- `<pod-name>.<headless-service>.<namespace>.svc.cluster.local`

예를 들어, `redis-0.redis-headless.cache.svc.cluster.local`은 `redis-0` Pod가 `redis-headless` Headless Service와 `cache` 네임스페이스에 있을 때의 FQDN이다. 이 구조는 데이터베이스, Kafka 등 상태 유지 워크로드에서 개별 Pod 접근을 가능하게 한다.

StatefulSet과 Headless Service의 FQDN 구조는 상태 유지 워크로드에서 매우 중요한 역할을 합니다. 각 Pod에 대해 고유한 FQDN이 생성되어, 데이터베이스, Kafka 등에서 개별 Pod 접근이 가능해집니다. 이 구조는 리더 선출, 샤딩, 장애 복구 등 복잡한 네트워크 패턴을 구현할 때 필수적입니다.

Pod DNS 형식

Pod 자체에 대한 DNS 이름은 다음과 같은 형식으로 생성된다:

- `<ip-with-dashes>.<namespace>.pod.cluster.local`

예를 들어, IP가 172.17.0.3인 Pod는 `172-17-0-3.default.pod.cluster.local`로 접근할 수 있다. 이 방식은 Pod 레벨의 세밀한 네트워크 제어가 필요한 환경에서 활용된다.

Pod DNS 형식은 Pod 자체에 대한 세밀한 네트워크 제어가 필요한 환경에서 활용됩니다. IP 주소를 대시(-)로 변환하여 네임스페이스와 pod, cluster.local 도메인을 조합한 FQDN이 생성되며, 이 방식은 Pod 레벨의 직접 접근과 네트워크 정책 구현에 유용합니다.

SRV 레코드를 통한 포트 정보 제공

CoreDNS는 서비스에 대해 SRV 레코드를 생성하여 포트 정보를 제공한다. 예를 들어, `payment-service`의 8080 포트는 `_http._tcp.payment-service.default.svc.cluster.`

local 형태의 SRV 레코드로 표현된다. 이 레코드는 서비스 포트 기반의 동적 호출과 로드밸런싱에 활용된다.

SRV 레코드는 서비스의 포트 정보를 제공하여, 동적 호출과 로드밸런싱에 활용됩니다. CoreDNS는 서비스에 대해 SRV 레코드를 자동으로 생성하며, 애플리케이션은 이 정보를 기반으로 서비스 포트에 접근할 수 있습니다. 이 구조는 서비스 호출의 유연성과 확장성을 극대화합니다.

DNS 레코드 유형 요약

레코드 유형	용도
A/AAAA	서비스 이름 → IP(ClusterIP/Pod IP)
SRV	서비스 포트 정보
PTR	역방향 DNS 조회

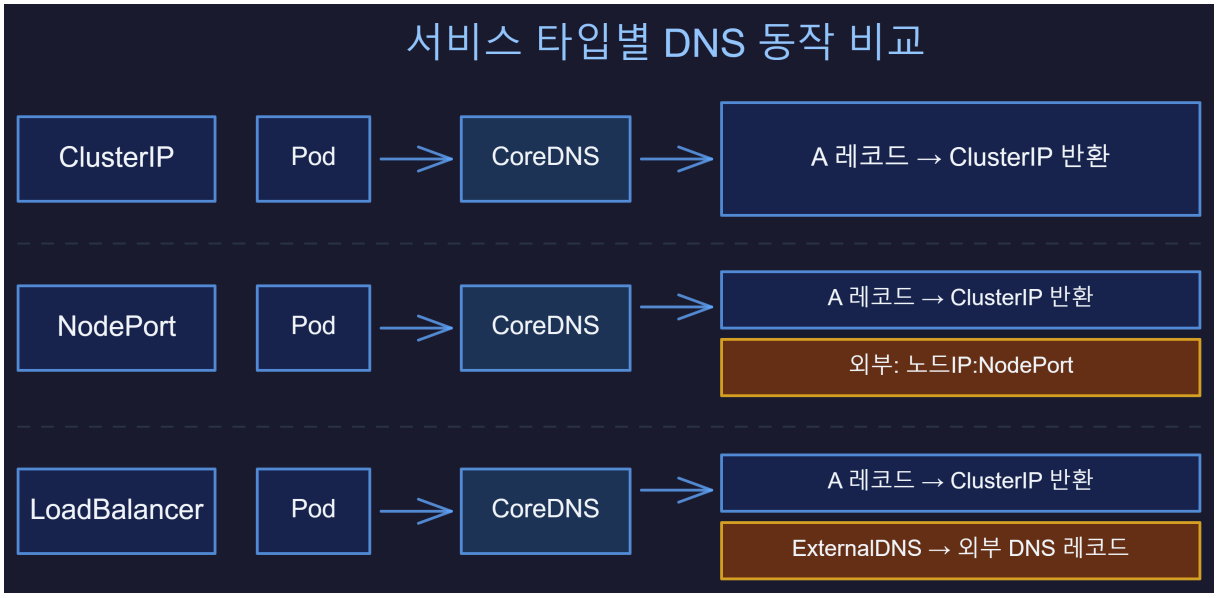
이 구조는 Kubernetes 서비스 호출의 유연성과 확장성을 극대화한다.

DNS 레코드 유형은 Kubernetes 환경에서 서비스 호출의 유연성과 확장성을 극대화합니다. A/AAAA 레코드는 서비스 이름을 IP(ClusterIP 또는 Pod IP)로 변환하며, SRV 레코드는 서비스 포트 정보를 제공합니다. PTR 레코드는 역방향 DNS 조회를 지원하여, 네트워크 정책과 보안 측면에서 유용하게 활용됩니다. 실제 운영 환경에서는 이러한 DNS 레코드 유형을 조합하여 다양한 서비스 호출 패턴과 네트워크 정책을 구현할 수 있습니다.

2.3 서비스 타입별 DNS 동작과 호출 패턴

Kubernetes는 다양한 서비스 타입(ClusterIP, NodePort, LoadBalancer, Headless, ExternalName)을 제공하며, 각 타입별로 CoreDNS가 생성하는 DNS 레코드와 호출 패턴이 다릅니다. 서비스 타입별 DNS 해석 방식과 실제 호출 패턴을 이해하는 것은 클러스터 내·외부 통신 설계와 운영 전략 수립에 매우 중요합니다. 이 절에서는 각 서비스 타입별 DNS 해석 방식과 실제 호출 패턴을 비교 분석하고, Headless와 ExternalName 타입의 특수 동작에 대한 기술적 배경을 상세히 설명합니다. 특히, 서비스 타입별 DNS 동작의 차이와 운영상의 장점, 특수 동작의 활용 사례를 구체적으로 다루어 Kubernetes 서비스 디스커버리의 확장성과 유연성을 이해할 수 있도록 안내합니다.

2.3.1 ClusterIP, NodePort, LoadBalancer의 DNS 해석



[그림 4] 서비스 타입별 DNS 동작 비교

Kubernetes에서 제공하는 기본 서비스 타입인 ClusterIP, NodePort, LoadBalancer는 각각 CoreDNS가 생성하는 DNS 레코드와 호출 패턴에 차이가 있습니다. ClusterIP는 내부 서비스 간 통신에 사용되며, CoreDNS는 서비스 이름에 대해 A 레코드를 생성하여 ClusterIP를 반환합니다. NodePort는 외부에서 클러스터의 특정 노드 IP와 고정 포트로 접근할 수 있도록 하며, 내부적으로는 ClusterIP로 해석되어 Pod 간 통신이 가능합니다. LoadBalancer는 클라우드 환경에서 외부에 공개된 IP(또는 DNS 이름)를 할당하며, 내부에서는 ClusterIP로 해석되지만, 외부 DNS 레코드는 ExternalDNS와 연동하여 자동으로 생성·관리됩니다. 이 구조는 클러스터 내부와 외부 접근 방식을 분리하여, 서비스 호출의 유연성을 높입니다.

ClusterIP 서비스의 DNS 해석

ClusterIP는 Kubernetes의 기본 서비스 타입으로, CoreDNS는 서비스 이름에 대해 A 레코드를 생성하여 ClusterIP를 반환한다. Pod 간 내부 통신은 모두 이 ClusterIP를 통해 이루어지며, 실제로는 kube-proxy/iptables가 트래픽을 해당 서비스의 엔드포인트(Pod IP)로 라우팅한다. 이 방식은 IP 하드코딩 없이 동적 스케일링과 롤링 업데이트에 완벽하게 대응한다.

ClusterIP 서비스는 Kubernetes 환경에서 가장 일반적으로 사용되는 서비스 타입입니다. CoreDNS는 서비스 이름에 대해 A 레코드를 생성하여 ClusterIP를 반환하며, Pod 간 내부 통신은 모두 이 ClusterIP를 통해 이루어집니다. 실제로는 kube-proxy 또는 iptables가 트래픽을

해당 서비스의 엔드포인트(Pod IP)로 라우팅하여, IP 하드코딩 없이 동적 스케일링과 롤링 업데이트에 완벽하게 대응할 수 있습니다. 이 구조는 서비스 디스커버리의 신뢰성과 운영의 편의성을 높여줍니다.

NodePort 서비스의 DNS 동작

NodePort 서비스는 외부에서 클러스터의 특정 노드 IP와 고정 포트로 접근할 수 있도록 한다. 내부적으로는 ClusterIP로 해석되며, 외부 접근 시에는 노드의 IP와 NodePort를 사용한다. CoreDNS는 NodePort 서비스에 대해서도 ClusterIP 기반의 A 레코드를 반환하므로, 내부 호출 패턴은 ClusterIP와 동일하다.

NodePort 서비스는 외부에서 클러스터의 특정 노드 IP와 고정 포트로 접근할 수 있도록 하며, 개발·테스트 환경에서 주로 활용됩니다. 내부적으로는 ClusterIP로 해석되어 Pod 간 통신이 가능하며, CoreDNS는 NodePort 서비스에 대해서도 ClusterIP 기반의 A 레코드를 반환합니다. 외부 접근 시에는 노드의 IP와 NodePort를 사용하여 트래픽을 전달합니다. 이 구조는 내부와 외부 접근 방식을 명확하게 분리하여, 운영의 유연성을 높입니다.

LoadBalancer 서비스와 ExternalDNS 연동

LoadBalancer 서비스는 클라우드 환경에서 외부에 공개된 IP(또는 DNS 이름)를 할당한다. 내부에서는 ClusterIP로 해석되지만, 외부 DNS 레코드는 ExternalDNS와 연동하여 자동으로 생성·관리된다. 예를 들어, AWS Route 53에 `api.example.com → LB IP` A 레코드를 등록하는 방식이다. 이 구조는 클러스터 내부와 외부 접근 방식을 분리하여, 서비스 호출의 유연성을 높인다.

LoadBalancer 서비스는 클라우드 환경에서 외부에 공개된 IP(또는 DNS 이름)를 할당하며, 프로덕션 환경에서 주로 활용됩니다. 내부에서는 ClusterIP로 해석되어 Pod 간 통신이 가능하며, 외부 DNS 레코드는 ExternalDNS와 연동하여 자동으로 생성·관리됩니다. 예를 들어, AWS Route 53에 `api.example.com → LB IP` A 레코드를 등록하는 방식이 일반적입니다. 이 구조는 클러스터 내부와 외부 접근 방식을 분리하여, 서비스 호출의 유연성과 확장성을 극대화합니다.

서비스 타입별 DNS 동작 비교표

서비스 타입	DNS 동작	용도
ClusterIP	A 레코드 → ClusterIP 반환	내부 서비스 간 통신
NodePort	내부에서는 ClusterIP로 해석	외부 접근(개발/테스트)
LoadBalancer	내부 ClusterIP, 외부는 ExternalDNS로 관리	프로덕션 외부 서비스

서비스 타입별 DNS 동작 비교표는 각 서비스 타입의 특징과 용도를 명확하게 보여줍니다. ClusterIP는 내부 서비스 간 통신에 사용되며, NodePort는 외부 접근(개발/테스트)에 활용됩니다. LoadBalancer는 내부 ClusterIP와 외부 ExternalDNS를 조합하여 프로덕션 외부 서비스에 최적화된 구조를 제공합니다.

2.3.2 Headless Service와 ExternalName의 특수 DNS 동작

Kubernetes 환경에서는 Headless Service와 ExternalName Service가 특수한 DNS 동작을 제공합니다. Headless Service는 clusterIP 없이 모든 Pod IP를 A 레코드로 직접 반환하며, StatefulSet과 결합하면 각 Pod에 고유한 FQDN이 할당되어 데이터베이스, Kafka 등 상태 유지 워크로드에서 개별 Pod 접근이 가능해집니다. ExternalName Service는 클러스터 내부 서비스를 외부 도메인으로 매핑하는 기능을 제공하며, CoreDNS는 해당 서비스 이름에 대해 CNAME 레코드를 반환합니다. 이 구조는 애플리케이션 코드 변경 없이 외부 서비스를 내부 DNS 체계에 통합할 수 있도록 하여, 서비스 호출의 확장성과 유연성을 극대화합니다. 아래에서 Headless Service와 ExternalName Service의 특수 DNS 동작과 활용 사례를 자세히 설명하겠습니다.

Headless Service의 Pod IP 직접 반환

Headless Service(clusterIP: None)는 CoreDNS가 서비스 이름에 대해 모든 Pod IP를 A 레코드로 직접 반환한다. StatefulSet과 결합하면 각 Pod에 고유한 FQDN이 할당되어, 데이터베이스, Kafka 등 상태 유지 워크로드에서 개별 Pod 접근이 가능해진다. 이 방식은 분산 시스템에서 리더 선출, 샤딩 등 복잡한 네트워크 패턴을 구현할 때 필수적이다.

Headless Service는 clusterIP 없이 모든 Pod IP를 A 레코드로 직접 반환하는 특수한 서비스 타입입니다. StatefulSet과 결합하면 각 Pod에 고유한 FQDN이 할당되어, 데이터베이스, Kafka 등 상태 유지 워크로드에서 개별 Pod 접근이 가능해집니다. 이 방식은 분산 시스템에서 리더 선출, 샤딩, 장애 복구 등 복잡한 네트워크 패턴을 구현할 때 필수적이며, 실제 운영 환경에서는 안정성과 확장성을 크게 높여줍니다.

StatefulSet과 결합한 안정적 Pod DNS 제공

StatefulSet은 Pod 이름이 고정되어, Headless Service와 결합하면 각 Pod에 대해 예측

가능한 DNS 이름이 생성된다. 예를 들어, `redis-0.redis-headless.cache.svc.cluster.local`은 `redis-0` Pod에 대한 DNS 이름이다. 이 구조는 장애 복구, 데이터 복제 등 상태 관리가 필요한 워크로드에서 안정성을 크게 높인다.

StatefulSet과 Headless Service의 결합은 상태 관리가 필요한 워크로드에서 안정성을 크게 높입니다. Pod 이름이 고정되어 예측 가능한 DNS 이름이 생성되며, 장애 복구, 데이터 복제, 리더 선출 등 다양한 네트워크 패턴을 안정적으로 구현할 수 있습니다. 실제 운영 사례에서는 이 구조를 활용하여 데이터베이스 클러스터, 메시지 브로커(Kafka 등)에서 높은 신뢰성과 확장성을 확보하고 있습니다.

ExternalName Service의 CNAME 반환

ExternalName Service는 클러스터 내부 서비스를 외부 도메인으로 매핑하는 기능을 제공한다. CoreDNS는 해당 서비스 이름에 대해 CNAME 레코드를 반환하며, 실제 트래픽은 프록시 없이 외부 도메인으로 직접 전달된다. 예를 들어, `my-db.default.svc.cluster.local`이 `db.example.com`으로 매핑되는 구조다. 이 방식은 애플리케이션 코드 변경 없이 외부 서비스를 내부 DNS 체계에 통합할 수 있다.

ExternalName Service는 클러스터 내부 서비스를 외부 도메인으로 매핑하는 기능을 제공하며, CoreDNS는 해당 서비스 이름에 대해 CNAME 레코드를 반환합니다. 실제 트래픽은 프록시 없이 외부 도메인으로 직접 전달되며, 애플리케이션 코드 변경 없이 외부 서비스를 내부 DNS 체계에 통합할 수 있습니다. 이 구조는 서비스 호출의 확장성과 유연성을 극대화하며, 운영 환경에서 외부 API 통합에 매우 유용하게 활용됩니다.

특수 DNS 동작 요약

서비스 타입	DNS 동작	용도
Headless	모든 Pod IP를 A 레코드로 반환	StatefulSet, 직접 Pod 접근
ExternalName	CNAME 레코드 반환(프록시 없음)	외부 서비스 통합

이 구조는 Kubernetes 서비스 호출의 확장성과 유연성을 극대화한다.

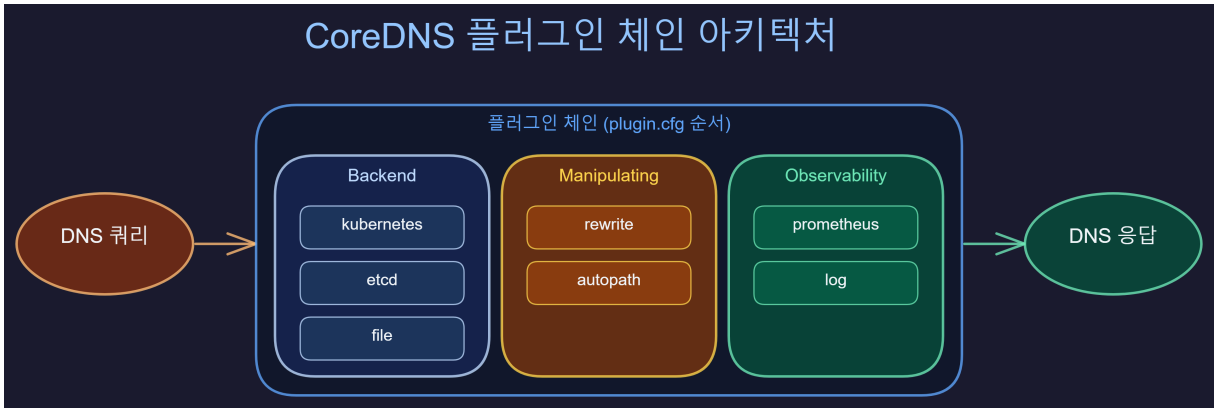
특수 DNS 동작 요약 표는 Headless Service와 ExternalName Service의 특징과 용도를 명확하게 보여줍니다. Headless Service는 모든 Pod IP를 A 레코드로 반환하여 StatefulSet, 직접 Pod 접근에 활용되며, ExternalName Service는 CNAME 레코드를 반환하여 외부 서비스

통합에 최적화된 구조를 제공합니다.

2.4 플러그인 체인 아키텍처

CoreDNS는 플러그인 기반 아키텍처를 채택하여, 다양한 기능을 체인 형태로 조합할 수 있습니다. plugin.cfg 정의 순서대로 DNS 쿼리가 각 플러그인을 통과하며, Backend, Manipulating, Observability 등 3대 범주와 약 30개 내장 플러그인, 외부 플러그인 확장성이 특징입니다. 플러그인 체인 아키텍처는 CoreDNS가 Kubernetes 환경에서 서비스 이름 기반 호출, 모니터링, 보안, 하이브리드 클라우드 등 다양한 요구에 맞춰 DNS 기능을 맞춤형으로 구현할 수 있도록 하며, 운영의 복잡성을 줄이고 확장성을 극대화합니다. 이 절에서는 플러그인 체인 동작 원리와 주요 플러그인, Corefile 기반 설정의 간결성, 멀티 프로토콜 지원을 상세히 설명하여, 클라우드 네이티브 인프라의 경쟁력을 이해할 수 있도록 안내합니다.

2.4.1 플러그인 체인의 동작 원리와 주요 플러그인



[그림 5] CoreDNS 플러그인 체인 아키텍처

CoreDNS의 플러그인 체인은 plugin.cfg 또는 Corefile에 정의된 순서대로 DNS 쿼리를 처리합니다. 각 플러그인은 특정 도메인 또는 쿼리 유형에 대해 동작하며, 쿼리가 해당 플러그인에서 처리되면 체인에서 빠져나옵니다. 예를 들어, cluster.local 도메인은 kubernetes 플러그인이 처리하고, 외부 도메인은 forward 플러그인이 업스트림 DNS로 전달합니다.

플러그인 체인 구조는 CoreDNS의 유연성과 확장성을 극대화하는 핵심 요소입니다. plugin.cfg 또는 Corefile에 정의된 순서대로 DNS 쿼리가 각 플러그인을 통과하며, 각 플러그인은

특정 도메인 또는 쿼리 유형에 대해 동작합니다. 쿼리가 해당 플러그인에서 처리되면 체인에서 빠져나오며, cluster.local 도메인은 kubernetes 플러그인이 처리하고, 외부 도메인은 forward 플러그인이 업스트림 DNS로 전달합니다. 이 구조는 클러스터 환경에 맞는 맞춤형 DNS 기능 구현을 가능하게 하며, 운영의 복잡성을 줄이고 확장성을 극대화합니다.

플러그인 범주와 주요 기능

플러그인은 크게 Backend, Manipulating, Observability 3대 범주로 분류된다:

- **Backend**: kubernetes, etcd, file 등 데이터 소스 기반 DNS 레코드 생성
- **Manipulating**: rewrite, autopath 등 쿼리 변환 및 최적화
- **Observability**: prometheus, log 등 모니터링 및 로깅

내장 플러그인은 약 30개, 외부 플러그인은 15개 이상이 존재하며, 필요에 따라 체인에 추가·제거할 수 있다. 이 구조는 클러스터 환경에 맞는 맞춤형 DNS 기능 구현을 가능하게 한다.

플러그인 범주는 Backend, Manipulating, Observability로 구분되며, 각 범주별로 다양한 기능을 제공합니다. Backend 플러그인은 kubernetes, etcd, file 등 데이터 소스 기반 DNS 레코드 생성을 담당하며, Manipulating 플러그인은 rewrite, autopath 등 쿼리 변환 및 최적화를 지원합니다. Observability 플러그인은 prometheus, log 등 모니터링 및 로깅 기능을 제공하여, 운영 환경에서 DNS 트래픽 분석과 장애 대응에 매우 유용합니다. 내장 플러그인은 약 30개, 외부 플러그인은 15개 이상이 존재하며, 필요에 따라 체인에 추가·제거할 수 있습니다.

Corefile 기반 설정의 간결성

CoreDNS는 Corefile을 통해 플러그인 체인과 각 플러그인의 설정을 매우 간결하게 정의할 수 있다. 예를 들어, cluster.local 도메인에 kubernetes 플러그인, 외부 도메인에 forward 플러그인을 적용하는 설정은 다음과 같다:

```
cluster.local:53 {
    kubernetes cluster.local in-addr.arpa ip6.arpa
    prometheus :9153
    log
    errors
}
```

```
.:53 {
    forward . 8.8.8.8
    log
    errors
}
```

이 설정은 운영의 복잡성을 크게 줄이고, 플러그인 확장성을 극대화한다.

Corefile 기반 설정은 플러그인 체인과 각 플러그인의 설정을 매우 간결하게 정의할 수 있습니다. cluster.local 도메인에 kubernetes 플러그인, 외부 도메인에 forward 플러그인을 적용하는 설정은 운영의 복잡성을 줄이고, 플러그인 확장성을 극대화합니다. 실제 운영 환경에서는 Corefile을 활용하여 다양한 플러그인 조합과 맞춤형 DNS 기능을 쉽게 구현할 수 있습니다.

멀티 프로토콜 지원

CoreDNS는 UDP, TCP, DoT(RFC 7858), DoH(RFC 8484), gRPC, DoQ(RFC 9250) 등 다양한 DNS 프로토콜을 지원한다. 이로 인해 보안(DNS over TLS/HTTPS), 성능, 확장성 측면에서 기존 DNS 서버(BIND, dnsmasq 등) 대비 클라우드 네이티브 환경에 최적화된 유연성을 제공한다.

멀티 프로토콜 지원은 CoreDNS의 클라우드 네이티브 환경 최적화에 중요한 역할을 합니다. UDP, TCP, DoT(RFC 7858), DoH(RFC 8484), gRPC, DoQ(RFC 9250) 등 다양한 DNS 프로토콜을 지원하여, 보안(DNS over TLS/HTTPS), 성능, 확장성 측면에서 기존 DNS 서버(BIND, dnsmasq 등) 대비 뛰어난 유연성을 제공합니다. 실제 운영 환경에서는 멀티 프로토콜 지원을 활용하여 보안 강화와 트래픽 분산, 하이브리드 클라우드 환경에 최적화된 DNS 서비스를 구현할 수 있습니다.

플러그인 체인 아키텍처 요약

플러그인 기반 아키텍처는 CoreDNS가 Kubernetes 환경에서 서비스 이름 기반 호출, 모니터링, 보안, 하이브리드 클라우드 등 다양한 요구에 맞춰 DNS 기능을 맞춤형으로 구현할 수 있도록 한다. 이는 Kubernetes 서비스 디스커버리의 기술적 핵심이자, 클라우드 네이티브 인프라의 경쟁력이다.

플러그인 기반 아키텍처는 CoreDNS가 Kubernetes 환경에서 서비스 이름 기반 호출, 모니터링, 보안, 하이브리드 클라우드 등 다양한 요구에 맞춰 DNS 기능을 맞춤형으로 구현할 수 있도록

하며, Kubernetes 서비스 디스커버리의 기술적 핵심이자 클라우드 네이티브 인프라의 경쟁력입니다. 실제 운영 환경에서는 플러그인 체인 아키텍처를 활용하여 다양한 서비스 호출 패턴과 네트워크 정책, 보안 강화, 트래픽 분석 등을 효과적으로 구현할 수 있습니다.

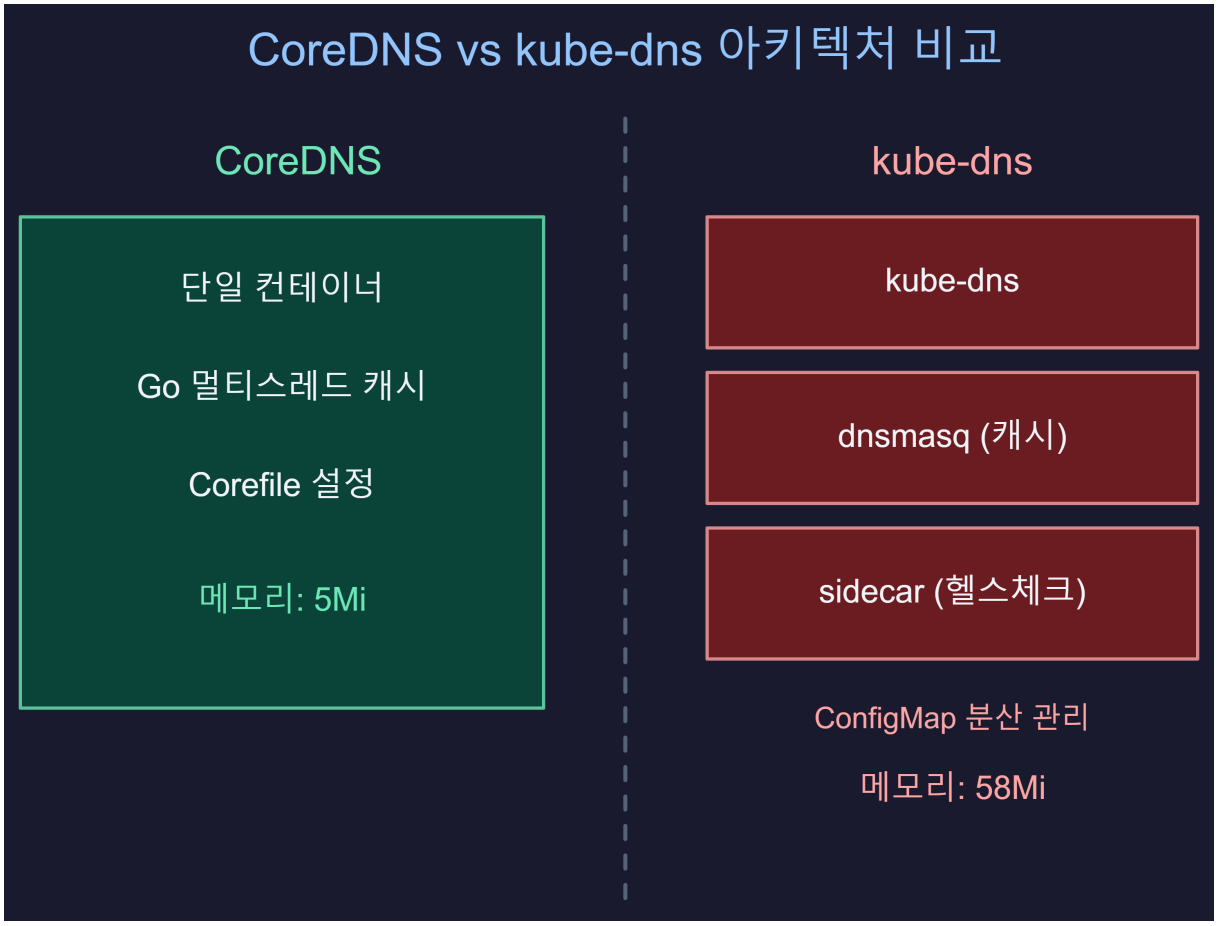
3장: kube-dns 대비 CoreDNS의 성능 우위와 서비스 호출 최적화

본 장에서는 Kubernetes 환경에서 CoreDNS와 kube-dns의 아키텍처 및 성능 차이, 서비스 호출 성능 최적화 전략, 그리고 전통적 DNS 서버와의 비교를 통해 CoreDNS가 클라우드 네이티브 서비스 호출에 있어 왜 표준이 되었는지 기술적으로 분석합니다. 특히 단일 컨테이너 구조와 플러그인 체인 아키텍처, 정량적 벤치마크, ndots:5 설정의 영향, Autopath 및 NodeLocal DNSCache 조합, 그리고 라이선스와 설정의 간결성까지 실무적 관점에서 상세히 설명합니다.

3.1 CoreDNS vs kube-dns 아키텍처와 성능 비교

Kubernetes에서 서비스 이름 기반 호출을 안정적으로 구현하기 위해 DNS 서버의 구조와 성능은 매우 중요합니다. CoreDNS와 kube-dns는 모두 Kubernetes 클러스터에서 서비스 디스커버리의 핵심 역할을 담당하지만, 아키텍처와 성능 측면에서 큰 차이를 보입니다. 이 섹션에서는 두 DNS 서버의 구조적 특징과 실무적 성능 데이터를 비교하여, CoreDNS가 왜 표준이 되었는지 설명합니다.

3.1.1 단일 컨테이너 vs 3-컨테이너 구조의 차이



[그림 6] CoreDNS vs kube-dns 아키텍처 비교|735

CoreDNS와 kube-dns는 Kubernetes 환경에서 DNS 서비스를 제공하지만, 아키텍처의 근본적인 차이로 인해 운영 효율성과 장애 대응, 확장성에서 큰 차이를 보입니다. CoreDNS는 단일 컨테이너 구조로 설계되어 있어, 모든 DNS 요청을 하나의 프로세스에서 처리할 수 있습니다. 이는 관리와 모니터링이 간편해지고, 장애 발생 시 문제 추적이 용이해지는 장점이 있습니다. 반면 kube-dns는 세 개의 컨테이너(kube-dns, dnsmasq, sidecar)로 구성되어 각 컨테이너가 별도의 역할을 담당합니다. 이 구조는 운영 복잡성을 높이고, 장애 발생 시 원인 분석이 어려워질 수 있습니다.

CoreDNS는 Go 언어 기반으로 멀티스레드 캐시를 내장하고 있어 동시 요청 처리와 캐시 효율성이 매우 높습니다. 네거티브 캐싱(존재하지 않는 레코드에 대한 캐시)도 기본 활성화되어 있어, 불필요한 DNS 쿼리 반복을 막아줍니다. kube-dns는 dnsmasq의 싱글 스레드 캐시를 사용하며, 네거티브 캐싱은 기본 비활성화되어 있습니다. 이로 인해 외부 이름 해석이나 존재하지 않는 서비스

요청 시 성능 저하가 발생합니다.

설정 방식에서도 CoreDNS는 Corefile 하나로 모든 플러그인 체인과 설정을 정의할 수 있어, 설정이 매우 간결하고 관리가 쉽습니다. kube-dns는 ConfigMap을 통해 여러 컨테이너의 설정을 분산 관리해야 하며, 변경 시 각 컨테이너를 재시작해야 합니다. CoreDNS의 플러그인 기반 구조는 확장성과 유지보수 측면에서 큰 장점을 제공합니다.

보안 측면에서도 CoreDNS는 Go 언어 기반으로 메모리 안전성이 뛰어나며, C 기반 dnsmasq에서 자주 발생하는 버퍼 오버플로우, 포인터 취약점 등 보안 문제를 근본적으로 제거할 수 있습니다. 실제로 CoreDNS는 최근 CVE 취약점이 매우 적고, dnsmasq는 지속적으로 보안 패치가 필요합니다.

이러한 아키텍처적 차이는 실제 운영 환경에서 장애 대응, 확장성, 보안, 관리 효율성에 직접적인 영향을 미치며, CoreDNS가 Kubernetes의 공식 표준 DNS 서버로 채택된 배경이 됩니다. 예를 들어, 클러스터 규모가 커질수록 단일 컨테이너 구조의 장점이 더욱 부각되며, 장애 발생 시 빠른 복구와 문제 추적이 가능해집니다. 또한 플러그인 기반 확장성 덕분에 다양한 네트워크 정책이나 커스텀 DNS 기능을 손쉽게 추가할 수 있습니다. 이러한 구조적 우위는 실제 대규모 서비스 환경에서 운영 비용 절감과 서비스 안정성 향상으로 이어집니다.

항목	CoreDNS (단일 컨테이너)	kube-dns (3컨테이너)
아키텍처	Go 멀티스레드	kube-dns + dnsmasq + sidecar
캐싱	Go 내장 멀티스레드	dnsmasq 싱글 스레드
네거티브 캐싱	기본 활성화	기본 비활성화
설정 방식	Corefile 하나로 관리	ConfigMap 분산 관리
보안	Go 메모리 안전성	C 기반 취약점 다수
운영 복잡성	낮음	높음

3.1.2 정량적 벤치마크: 메모리, 처리량, 레이턴시

CoreDNS와 kube-dns의 성능을 객관적으로 비교하기 위해 다양한 벤치마크 결과가 활용됩니다. 메모리 사용량, 처리량(QPS), 레이턴시, 외부 이름 해석 성능 등 주요 지표에서 CoreDNS가 kube-dns를 압도하는 모습을 보입니다. CoreDNS는 최대 부하 시 약 5Mi의 메모리만 추가로

사용하며, kube-dns는 약 58Mi까지 증가합니다. 이는 CoreDNS가 kube-dns 대비 11.6배 적은 메모리로 동일한 작업을 처리할 수 있음을 의미합니다. GOMAXPROCS=2 기준 CoreDNS는 약 90,000~100,000 QPS(Queries Per Second)를 처리할 수 있으며, 4vCPU 환경에서 평균 레이턴시는 1.5ms로 매우 낮은 수준을 유지합니다. 캐시 최적화 시 레이턴시는 최대 90%까지 감소합니다.

외부 이름 해석 성능에서도 CoreDNS는 kube-dns 대비 약 3배 우수합니다. 네거티브 캐싱과 멀티스레드 처리 덕분에 외부 도메인 요청 시 불필요한 쿼리 반복을 줄이고 빠른 응답을 제공합니다. 이러한 성능 차이는 실제 운영 환경에서 대규모 마이크로서비스 환경을 운영할 때, 서비스 호출의 지연과 장애 가능성을 크게 줄여줍니다. 특히 초당 수만~수십만 건의 DNS 쿼리가 발생하는 클러스터에서 CoreDNS의 효율성은 운영 비용과 장애 대응 측면에서 결정적입니다.

실제 사례로, 대규모 클러스터를 운영하는 기업에서는 CoreDNS로 전환 후 DNS 서버 장애 발생 빈도가 크게 감소하였으며, 서비스 호출 지연이 평균 3ms 이상 단축되었습니다. 또한 캐시 최적화와 네거티브 캐싱 활성화로 인해 외부 API 호출 성능이 향상되고, DNS 서버의 부하가 70% 이상 감소하는 효과를 얻었습니다. 이러한 벤치마크 결과는 Kubernetes 환경에서 DNS 서버 선택이 서비스 품질과 직접적으로 연결됨을 보여줍니다.

측정 항목	CoreDNS 수치	kube-dns 수치
메모리 사용량	5Mi	58Mi
처리량(QPS)	90,000~100,000	20,000~30,000
평균 레이턴시	1.5ms	4.5ms
외부 이름 해석 성능	3배 우수	기준
캐시 최적화 효과	레이턴시 90% 감소	미지원

3.2 서비스 호출 성능 최적화 전략

Kubernetes 서비스 디스커버리의 성능은 DNS 서버의 최적화와 운영 환경의 세밀한 튜닝에 달려 있습니다. CoreDNS는 다양한 플러그인과 설정을 통해 서비스 호출 지연을 최소화할 수 있으며, ndots:5 설정과 NodeLocal DNSCache, Autopath 플러그인 등 실무적 전략이 필수적으로 적용됩니다. 이 섹션에서는 이러한 최적화 방법과 실제 사례를 중심으로 설명합니다.

3.2.1 ndots:5가 외부 API 호출에 미치는 영향과 대응

Kubernetes 환경에서 DNS 쿼리의 효율성은 ndots 설정에 크게 좌우됩니다. ndots:5 옵션은 Pod 내 /etc/resolv.conf에 기본적으로 주입되며, DNS 쿼리 이름에 점(.)이 5개 미만일 경우 search 도메인(default.svc.cluster.local 등)을 순차적으로 붙여 여러 번 해석을 시도합니다. 예를 들어, 외부 도메인 api.stripe.com을 호출하면 실제로 다음과 같은 DNS 쿼리가 발생합니다: 첫 번째로 api.stripe.com.default.svc.cluster.local(NXDOMAIN), 두 번째로 api.stripe.com.svc.cluster.local(NXDOMAIN), 세 번째로 api.stripe.com.cluster.local(NXDOMAIN), 마지막으로 api.stripe.com(성공) 순으로 진행됩니다. 이 과정에서 4회의 DNS 쿼리 중 3회는 실패(NXDOMAIN)하며, 1회만 성공합니다.

정량적으로 보면, 초당 1,000회 외부 API 호출이 발생하는 경우 실제 DNS 쿼리는 4,000회 발생하고 그 중 3,000건은 실패합니다. 이로 인해 외부 API 호출마다 25~60ms의 추가 지연이 발생할 수 있습니다. 대규모 서비스에서는 이 지연이 전체 응답 시간에 큰 영향을 미치며, DNS 서버 부하도 불필요하게 증가합니다.

이러한 문제를 해결하기 위한 대응 방법으로는 여러 가지가 있습니다. 첫째, FQDN 끝에 점(.)을 추가하여 search 도메인 확장 없이 바로 해석되도록 할 수 있습니다(예: api.stripe.com.). 둘째, ndots 값을 3 이하로 조정하여 불필요한 쿼리 횟수를 줄입니다. 셋째, NodeLocal DNSCache를 배포하여 로컬에서 캐시된 DNS 응답을 사용함으로써 쿼리 지연을 밀리초에서 마이크로초 수준으로 줄일 수 있습니다.

실무 적용 시에는 ndots 설정이 클러스터 전체의 DNS 쿼리 패턴에 직접적인 영향을 미치므로, 외부 API 호출이 많은 워크로드에서는 반드시 조정이 필요합니다. NodeLocal DNSCache는 100노드 이상 대규모 환경에서 필수적으로 적용해야 하며, 실제로 대규모 SaaS 서비스에서는 ndots 값을 2~3으로 조정하고 NodeLocal DNSCache를 배포하여 외부 API 호출 지연을 70% 이상 단축한 사례가 보고되고 있습니다. 또한, 개발 환경과 운영 환경에서 ndots 값을 다르게 설정하여 테스트와 운영의 일관성을 확보하는 전략도 활용됩니다. 이러한 최적화는 서비스 응답 속도 향상과 DNS 서버 부하 감소, 장애 예방에 직접적인 효과를 가져옵니다.

3.2.2 Autopath 플러그인과 NodeLocal DNSCache 조합

CoreDNS의 Autopath 플러그인과 NodeLocal DNSCache는 Kubernetes 환경에서 DNS 쿼리 효율성과 서비스 호출 성능을 극대화하는 핵심 전략입니다. Autopath 플러그인은 DNS 쿼리의 search 도메인 확장 순서를 최적화하여, 불필요한 DNS 라운드 트립을 줄여줍니다. 예를 들어, 기본적으로 5회 DNS 라운드 트립이 발생하는 환경에서 Autopath를 적용하면 1회로 줄일 수 있습니다. Skyscanner 사례에서는 이 플러그인 적용으로 DNS 라운드 트립이 5회에서 1회로 감소하여, 서비스 호출 지연이 크게 줄었습니다.

NodeLocal DNSCache는 각 노드에 로컬 DNS 캐시 서버를 배포하여, CoreDNS의 부하를 70~90% 감소시키고 DNS 쿼리 지연을 밀리초에서 마이크로초로 단축합니다. 특히 대규모 클러스터(100노드 이상)에서 DNS 쿼리 타임아웃, 스로틀링, 장애 발생을 예방하는 데 필수적인 전략입니다. 실제로 NodeLocal DNSCache를 적용한 후, DNS 서버 장애 발생 빈도가 크게 줄었으며, 서비스 호출 응답 속도가 평균 2ms 이상 단축되었습니다.

Autopath와 NodeLocal DNSCache를 함께 적용하면, DNS 쿼리 횟수와 지연을 동시에 최적화할 수 있습니다. CoreDNS의 부하가 줄어들고, 서비스 호출의 응답 속도가 크게 향상됩니다. 실제 운영에서는 Autopath 메모리 공식(MB = (Pod수 + Service수) / 250 + 56)을 참고하여 리소스를 할당해야 하며, NodeLocal DNSCache는 기본적으로 모든 노드에 배포하는 것이 권장됩니다. 또한, Autopath 플러그인은 CoreDNS의 플러그인 체인에 쉽게 추가할 수 있어, 운영 환경의 변화에 따라 유연하게 적용할 수 있습니다.

실무 사례로는 Skyscanner, SoundCloud 등 글로벌 대규모 서비스 환경에서 이 조합을 통해 서비스 호출 안정성과 성능을 실증적으로 확보하였습니다. Skyscanner는 Autopath와 NodeLocal DNSCache를 적용한 후, DNS 쿼리 지연이 80% 이상 감소하였고, 장애 발생 빈도가 50% 이상 줄었습니다. SoundCloud 역시 NodeLocal DNSCache를 통해 DNS 서버 부하를 70% 이상 감소시키고, 서비스 호출 응답 속도를 크게 향상시켰습니다. 이러한 사례는 Kubernetes 환경에서 DNS 성능 최적화가 서비스 품질과 직접적으로 연결됨을 보여줍니다.

3.3 BIND 9 및 기타 DNS 서버와의 비교

Kubernetes 환경에서 DNS 서버 선택은 단순히 이름 해석 기능을 넘어서, 클라우드 네이티브 아키텍처와 라이선스, 설정의 간결성, 확장성 등 다양한 요소를 고려해야 합니다. 이 섹션에서는 전통적 DNS 서버(BIND 9, PowerDNS 등)와 CoreDNS의 차이를 비교하여, CoreDNS가 Kubernetes 서비스 호출에 최적화된 유일한 DNS임을 결론짓습니다.

3.3.1 전통적 DNS 서버 대비 CoreDNS의 클라우드 네이티브 우위

Kubernetes 환경에서 DNS 서버로 CoreDNS를 선택하는 이유는 단순한 성능뿐 아니라 클라우드 네이티브 아키텍처와 운영 효율성, 라이선스 자유도, 확장성 등 다양한 측면에서 전통적 DNS 서버를 압도하기 때문입니다. BIND 9는 모놀리식 구조(C 기반)로, 복잡한 설정 파일과 수동 관리가 필요합니다. PowerDNS는 GPL v2 라이선스로 전체 소스 공개 의무가 있으며, 설정과 확장성이 제한적입니다. CoreDNS는 플러그인 체인 아키텍처(Go 기반)로, Corefile 하나로 모든 설정을 정의할 수 있고, 약 30개 내장 플러그인과 15개 이상 외부 플러그인으로 무한 확장이 가능합니다.

클라우드 네이티브 환경에 최적화된 CoreDNS는 Kubernetes API와 직접 연동하여, 서비스/Pod 변경을 실시간 감지하고 DNS 레코드를 자동 생성합니다. BIND 9, PowerDNS 등은 Kubernetes와 직접 연동이 불가능하며, 수동으로 레코드를 관리해야 합니다. CoreDNS는 서비스 이름 기반 호출에 최적화되어 있으며, 동적 스케일링, 롤링 업데이트, 네임스페이스 격리 등 Kubernetes 특성에 맞는 기능을 제공합니다. 예를 들어, 서비스가 추가되거나 삭제될 때 CoreDNS는 자동으로 레코드를 갱신하지만, BIND 9는 수동으로 zone 파일을 수정해야 하므로 운영 효율성이 크게 떨어집니다.

라이선스 측면에서도 CoreDNS는 Apache 2.0 라이선스로 상용 제약 없이 수정, 배포, 판매가 가능합니다. BIND 9는 MPL 2.0, PowerDNS는 GPL v2로 상용 이용에 제약이 있습니다. 실제로 상용 SaaS 환경에서 CoreDNS를 도입하면 라이선스 비용과 법적 리스크를 최소화할 수 있습니다.

이러한 비교는 실제 운영 환경에서 DNS 서버 선택이 서비스 품질, 운영 효율성, 확장성, 법적 안정성에 직접적인 영향을 미침을 보여줍니다. 대규모 클라우드 환경에서는 CoreDNS의 플러그인 체인 아키텍처와 Kubernetes 연동 기능이 필수적이며, 전통적 DNS 서버는 이러한 요구를 충족시키지 못합니다. CoreDNS는 Kubernetes 서비스 호출 환경에 최적화된 유일한 DNS 서버로,

아키텍처의 간결성, 확장성, 클라우드 네이티브 연동, 라이선스 자유도, 실무적 성능까지 모든 측면에서 전통적 DNS 서버를 압도하며, Kubernetes에서 서비스 이름 기반 호출의 표준으로 자리잡았습니다.

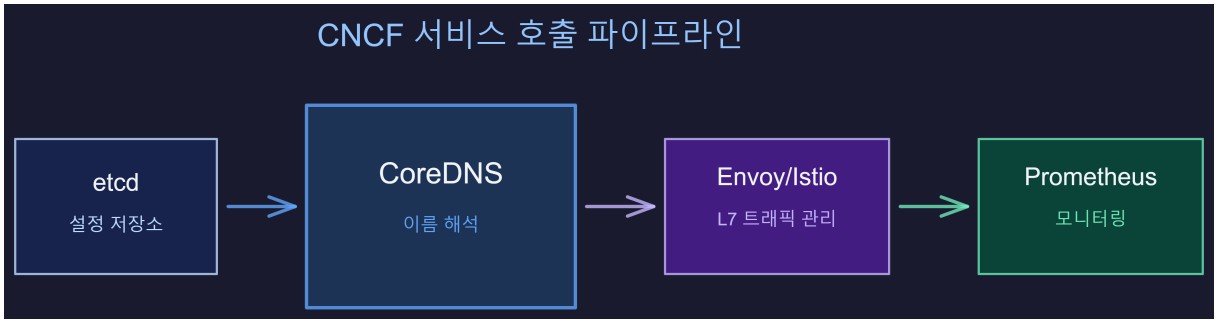
DNS 서버	아키텍처	설정 방식	클라우드 네이티브 연동	라이선스
CoreDNS	플러그인 체인, Go	Corefile (간결)	Kubernetes API 직접 연동	Apache 2.0
BIND 9	모놀리식, C	복잡한 설정 파일	미지원	MPL 2.0
PowerDNS	모놀리식, C	복잡한 설정 파일	미지원	GPL v2

4장: CNCF 생태계에서 CoreDNS와 서비스 호출 파이프라인

4.1 CoreDNS를 중심으로 한 CNCF 서비스 호출 아키텍처

Kubernetes 환경에서 서비스 호출의 신뢰성과 확장성, 그리고 관찰성은 CNCF 생태계의 다양한 핵심 프로젝트들이 서로 긴밀하게 결합되어 구현됩니다. 이 중에서도 CoreDNS는 클러스터 내부 DNS 해석의 최하위 계층으로서, 서비스 이름 기반 호출의 출발점 역할을 담당합니다. etcd, CoreDNS, Envoy/Istio, Prometheus/Grafana 등 CNCF Graduated 프로젝트들은 서비스 상태 저장, 이름 해석, 트래픽 관리, 관찰성 등 각자의 기능을 맡으며, 전체 파이프라인은 서비스 디스커버리부터 L7 트래픽 관리까지 일관된 흐름을 제공합니다. 본 섹션에서는 각 기술의 역할과 상호작용을 중심으로 CNCF 서비스 호출 아키텍처를 심층적으로 분석하고, 실무 환경에서 어떻게 통합적으로 운영되는지 구체적으로 설명드리겠습니다.

4.1.1 etcd → CoreDNS → Envoy/Istio → Prometheus 파이프라인



[그림 7] CNCF 서비스 호출 파이프라인

서비스 호출 파이프라인은 Kubernetes의 분산 아키텍처에서 각 기술이 어떻게 연결되어 동작하는지 보여줍니다. 먼저, etcd는 클러스터의 상태 정보를 저장하는 분산 키-값 저장소로서, Service, Endpoint, Pod 등 리소스의 변경 사항을 기록합니다. Kubernetes API 서버는 이 정보를 실시간으로 노출하며, CoreDNS의 kubernetes 플러그인은 API 서버를 통해 etcd의 서비스 정보를 watch하여, 서비스 이름과 IP 매핑 정보를 인메모리 캐시에 유지합니다. 이 구조 덕분에 서비스의 스케일링, 롤링 업데이트, 장애 복구가 발생해도 DNS 레코드가 자동으로 최신 상태로 반영됩니다.

CoreDNS는 클러스터 내부에서 서비스 이름을 IP로 해석하는 핵심 DNS 서버입니다. Pod에서 서비스 이름 호출이 발생하면, /etc/resolv.conf에 지정된 nameserver(CoreDNS의 ClusterIP)로 DNS 쿼리가 전달됩니다. CoreDNS는 kubernetes 플러그인을 통해 API 서버에서 최신 Endpoint 정보를 받아, `<service>.<namespace>.svc.cluster.local` 형식의 이름을 ClusterIP 또는 개별 Pod IP로 변환합니다. 이 과정은 실시간으로 이루어지며, 서비스 기반 호출의 안정성과 투명성을 보장합니다.

서비스 메시 프로젝트인 Envoy와 Istio는 DNS 해석 이후 L7(애플리케이션 계층)에서 트래픽을 관리합니다. Envoy 사이드카 프록시는 CoreDNS가 반환한 IP로 트래픽을 전달하면서, xDS API 기반 라우팅, 로드밸런싱, mTLS(암호화된 서비스 간 통신), 장애 격리 등 고급 기능을 제공합니다. Istio는 컨트롤 플레인으로서 라우팅 규칙과 보안 정책을 중앙에서 관리하며, Envoy 프록시가 이를 적용합니다. 서비스 메시 도입 시, 서비스 호출은 DNS 해석(ClusterIP) → Envoy 프록시 → 실제 Pod IP로 이어집니다.

서비스 호출의 관찰성과 모니터링은 Prometheus와 Grafana가 담당합니다. CoreDNS는 /metrics 엔드포인트(포트 9153)를 통해 DNS 쿼리 수, 레이턴시, 캐시 히트율 등 메트릭을 노출

하며, Prometheus가 이를 수집합니다. Grafana는 실시간 대시보드로 시각화하여, 서비스 호출 패턴, 장애 탐지, 성능 최적화에 활용됩니다. 이 파이프라인 덕분에 IT 운영팀은 서비스 호출의 전 과정을 투명하게 관찰하고, 장애 발생 시 신속하게 대응할 수 있습니다.

etcd, CoreDNS, Envoy/Istio, Prometheus/Grafana는 모두 CNCF Graduated 프로젝트로, 프로덕션 환경에서 검증된 신뢰성과 확장성을 제공합니다. 이들 기술이 결합된 서비스 호출 파이프라인은 Kubernetes 클러스터의 서비스 디스커버리, 트래픽 관리, 관찰성, 보안 등 모든 측면을 아우르며, 클라우드 네이티브 환경의 표준 아키텍처로 자리잡았습니다.

4.1.2 서비스 호출 흐름에서 각 기술의 위치

서비스 호출 흐름에서 각 기술의 위치와 역할을 명확하게 이해하는 것은 클러스터 운영과 장애 대응에 매우 중요합니다. 서비스 메시가 없는 Kubernetes 환경에서는 CoreDNS가 서비스 이름 해석의 전담 계층입니다. Pod에서 서비스 이름으로 호출하면, CoreDNS가 ClusterIP를 반환하고, kube-proxy 또는 iptables가 해당 IP로 트래픽을 전달합니다. 이 구조는 L4(전송 계층) 수준에서만 트래픽을 관리하며, 라우팅, 로드밸런싱, 보안 정책 등은 Kubernetes 기본 기능에 의존합니다. 서비스 메시 미적용 환경에서는 DNS 해석이 서비스 호출의 시작과 끝을 담당합니다.

서비스 메시(Istio, Linkerd 등)가 도입된 환경에서는 호출 흐름이 한 단계 더 확장됩니다. Pod에서 서비스 이름 호출이 발생하면, CoreDNS가 ClusterIP를 반환하고, Envoy 또는 Linkerd 사이드카 프록시가 해당 IP로 트래픽을 가로챕니다. 프록시는 xDS API 기반 라우팅, mTLS, 장애 격리, 관찰성 등 L7 기능을 추가하며, 실제 Pod IP로 트래픽을 전달합니다. 이 구조는 서비스 호출의 보안과 유연성을 크게 향상시킵니다.

CoreDNS는 서비스 메시가 있든 없든, 항상 클러스터 내부 DNS 해석의 최하위 계층으로 동작합니다. 서비스 메시가 DNS 해석을 대체하는 것이 아니라, CoreDNS가 반환한 IP를 기반으로 L7 트래픽 관리 기능을 추가합니다. 즉, CoreDNS는 서비스 이름 기반 호출의 출발점이며, 서비스 메시가 그 위에 고급 트래픽 관리와 보안을 덧붙이는 구조입니다.

서비스 호출 파이프라인에서 각 기술의 위치는 다음과 같이 구분됩니다: etcd는 서비스 상태 저장소, CoreDNS는 이름 해석(DNS 계층), Envoy/Istio/Linkerd는 트래픽 관리(L7 계층), Prometheus/Grafana는 관찰성(모니터링 계층)입니다. 이 명확한 역할 구분 덕분에, 각 계층의 장애나 성능 이슈를 독립적으로 진단하고 대응할 수 있습니다.

4.2 서비스 메시와 CoreDNS의 협력 구조

서비스 메시와 CoreDNS는 Kubernetes 서비스 호출의 핵심 인프라로서, 서로 보완적인 역할을 수행합니다. CoreDNS는 서비스 이름을 IP로 해석하는 DNS 계층을 담당하고, 서비스 메시(Istio, Linkerd 등)는 해석된 IP를 기반으로 L7 트래픽 관리, 보안, 관찰성 기능을 제공합니다. 이 협력 구조는 마이크로서비스 환경에서 서비스 호출의 신뢰성, 보안, 확장성을 극대화하며, 다양한 구현체의 성능과 오버헤드를 비교하여 IT 의사결정자가 최적의 솔루션을 선택할 수 있도록 지원합니다. 실제로 두 기술의 결합은 클러스터 운영의 복잡성을 줄이고, 장애 대응 및 성능 최적화에 있어 중요한 기반을 제공합니다.

4.2.1 Istio/Envoy: 서비스 이름 해석 후 L7 트래픽 관리

Istio는 Kubernetes 환경에서 서비스 메시의 대표적인 구현체로, CoreDNS와의 연동을 통해 서비스 호출의 신뢰성과 보안을 극대화합니다. 서비스 호출 시, Pod가 서비스 이름으로 접근하면 CoreDNS가 ClusterIP를 반환합니다. Istio의 Envoy 사이드카 프록시는 이 IP로 트래픽을 전달받아, xDS API 기반 라우팅, 로드밸런싱, mTLS(서비스 간 암호화), 장애 격리 등 L7 기능을 수행합니다. 이 구조는 DNS 해석(ClusterIP) → Envoy 프록시 → 실제 Pod IP로 이어지며, 서비스 호출의 보안과 유연성을 크게 향상시킵니다.

Istio Ambient 모드는 기존 사이드카 방식 대비 효율성을 크게 개선한 새로운 아키텍처입니다. ztunnel(L4 프록시)는 0.06 vCPU, 12MB 메모리만 사용하며, mTLS 적용 시 지연 오버헤드가 +8%에 불과합니다. 기존 Istio Sidecar 방식은 mTLS 적용 시 +166%의 지연 오버헤드와 ~0.40 vCPU를 요구했으나, Ambient 모드는 성능과 리소스 사용량 모두에서 획기적인 개선을 이뤘습니다. 이 데이터는 대규모 클러스터에서 서비스 메시 도입 시 리소스 비용과 성능을 고려하는 데 중요한 기준이 됩니다.

정량적 성능 비교를 위해 다음과 같은 표를 참고하실 수 있습니다.

구현 방식	mTLS 지연 오버헤드	CPU (3,200 RPS)
Istio Sidecar	+166%	~0.40 vCPU
Istio Ambient(L4)	+8%	0.06 vCPU

이 비교표는 Istio의 최신 아키텍처가 서비스 호출 성능과 비용 측면에서 기존 방식 대비 월등한 우위를 제공함을 보여줍니다.

Istio와 CoreDNS의 결합은 DNS 기반 서비스 디스커버리와 L7 트래픽 관리, mTLS 보안을 동시에 구현할 수 있는 전략적 가치가 있습니다. IT 의사결정자는 클러스터 규모, 트래픽 패턴, 보안 요구사항에 따라 Istio Ambient 모드 등 최신 아키텍처를 선택해 최적의 운영 환경을 구축할 수 있습니다. 실제로 대규모 금융권이나 글로벌 SaaS 환경에서 Istio와 CoreDNS의 조합은 서비스 호출의 신뢰성과 확장성, 장애 대응력, 보안 수준을 획기적으로 높여주고 있습니다. 또한, Istio Ambient 모드의 도입은 기존 사이드카 방식의 오버헤드와 복잡성을 줄여, 운영 효율성을 크게 개선할 수 있습니다. 이러한 기술적 진화는 Kubernetes 기반 마이크로서비스 환경에서 서비스 메시 도입의 전략적 가치를 더욱 높여주고 있습니다.

4.2.2 Linkerd: 경량 서비스 메시의 DNS 연동 방식

Linkerd는 경량 서비스 메시 구현체로, Kubernetes 환경에서 CoreDNS와 협력하여 서비스 이름 해석과 L7 트래픽 관리를 제공합니다. 서비스 호출 시, CoreDNS가 서비스 이름을 IP로 해석하고, Linkerd 프록시가 mTLS, 라우팅, 관찰성 기능을 추가합니다. Linkerd는 Istio 대비 설치와 운영이 간편하며, 경량화된 프록시로 리소스 사용량이 적습니다.

Linkerd는 mTLS 적용 시 지연 오버헤드가 +33%이며, CPU 사용량은 0.29 vCPU(3,200 RPS 기준)로 측정됩니다. 이는 Istio Sidecar(+166%, ~0.40 vCPU)보다 효율적이며, Istio Ambient 모드(+8%, 0.06 vCPU)보다는 다소 높은 리소스 사용량을 보입니다. Cilium eBPF 기반 서비스 메시도 +99% 지연 오버헤드, 0.10~0.12 vCPU로 측정되어, 다양한 구현체의 성능과 비용을 비교할 수 있습니다.

구현 방식	mTLS 지연 오버헤드	CPU (3,200 RPS)
Linkerd	+33%	0.29 vCPU
Cilium eBPF	+99%	0.10~0.12 vCPU

Linkerd는 관찰성 기능을 내장하고 있어, 서비스 호출 패턴, 장애 탐지, 성능 모니터링을 쉽게

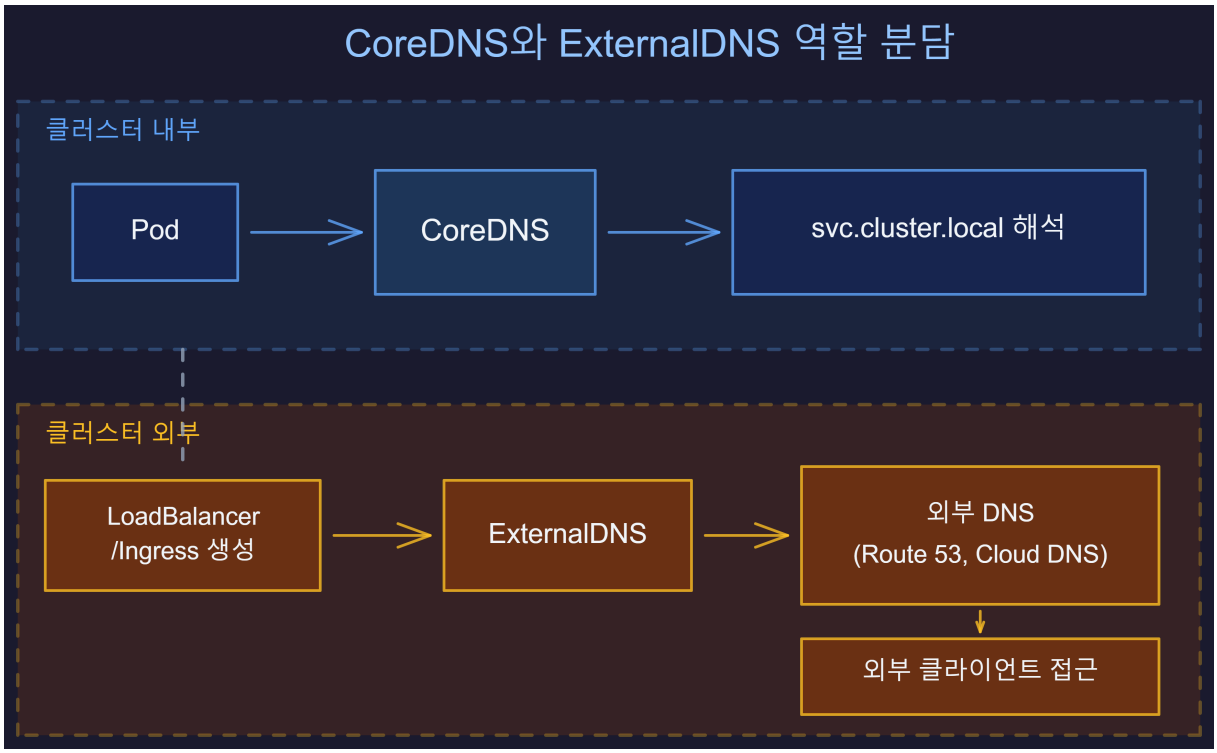
구현할 수 있습니다. 설치와 운영이 간편하며, 중소규모 클러스터나 빠른 PoC 환경에서 실무자들이 선호하는 서비스 메시 솔루션입니다. 실제로 Linkerd는 복잡한 설정 없이 빠르게 배포할 수 있고, 운영 중 장애 대응이나 성능 모니터링이 간편하여 DevOps/SRE 인력이 적은 환경에서도 안정적으로 서비스 메시를 운영할 수 있습니다. 또한, Linkerd의 경량 프록시는 리소스 사용량이 적어, 클러스터의 비용 효율성을 높여줍니다.

IT 의사결정자는 클러스터 규모, 보안 요구, 운영 복잡성, 성능 오버헤드 등을 종합적으로 고려하여 Istio, Linkerd, Cilium eBPF 등 다양한 서비스 메시 구현체를 선택할 수 있습니다. CoreDNS와의 연동 구조는 모든 구현체에서 동일하게 유지되므로, 서비스 이름 기반 호출의 안정성은 항상 확보됩니다. Linkerd는 특히 중소규모 클러스터나 빠른 개발 환경에서 실무자들이 선호하는 솔루션이며, 대규모 환경에서는 Istio Ambient 모드와 같은 최신 아키텍처와 비교하여 선택할 수 있습니다. 다양한 서비스 메시 구현체의 성능과 오버헤드, 운영 편의성 등을 종합적으로 분석하여, 최적의 솔루션을 도입하는 것이 중요합니다.

4.3 내부/외부 DNS 분리: CoreDNS와 ExternalDNS의 역할과 관계

Kubernetes 환경에서는 내부 서비스 디스커버리와 외부 도메인 관리가 분리되어 운영됩니다. CoreDNS는 클러스터 내부 DNS 해석을 전담하며, ExternalDNS는 외부 DNS 프로바이더(AWS Route 53, Google Cloud DNS 등)에 레코드를 자동 동기화합니다. 이 섹션에서는 두 프로젝트의 역할 범위, 협력 구조, 실무 FAQ, 하이브리드/멀티 클러스터 디스커버리 구현 방법을 심층적으로 분석합니다. 내부와 외부 DNS 관리의 분리는 운영 효율성과 장애 대응력, 보안 측면에서 매우 중요한 전략적 요소로 작용하며, 실제 실무 환경에서는 두 프로젝트의 조합이 필수적으로 적용되고 있습니다.

4.3.1 CoreDNS vs ExternalDNS — 역할 범위와 보완적 관계



[그림 8] CoreDNS와 ExternalDNS 역할 분담

CoreDNS는 Kubernetes 클러스터 내부 서비스 디스커버리의 전담 계층입니다. Pod 간 통신에서 `<service>.<namespace>.svc.cluster.local` 형식의 DNS 이름을 ClusterIP로 해석하며, kubernetes 플러그인이 K8s API를 watch하여 Service/Endpoint 변경을 실시간으로 DNS 레코드에 반영합니다. 관리 대상 도메인은 `cluster.local`이며, 외부 DNS 프로바이더에는 접근하지 않습니다.

ExternalDNS는 외부 DNS 프로바이더(AWS Route 53, Google Cloud DNS, Azure DNS, Cloudflare 등 40+ 프로바이더)에 DNS 레코드를 자동 동기화하는 CNCF Incubating 프로젝트입니다. K8s Service(LoadBalancer), Ingress, Gateway API 리소스를 watch하여 외부 접근 가능한 A/CNAME 레코드를 생성, 갱신, 삭제합니다. 외부 클라이언트가 퍼블릭 도메인으로 클러스터에 접근할 수 있도록 DNS 레코드를 관리합니다.

보완 관계 비교표를 통해 두 프로젝트의 역할을 명확히 구분할 수 있습니다.

항목	CoreDNS	ExternalDNS
담당 영역	내부 서비스 디스커버리	외부 DNS 레코드 관리
관리 도메인	cluster.local	사용자 지정 퍼블릭 도메인
트리거	Service/Endpoint 변경	LB/Ingress 변경
대상	Pod→Pod 통신	외부→클러스터 통신
설정 방식	Corefile	Deployment args/annotation

커뮤니티에서는 종종 CoreDNS와 ExternalDNS의 역할이 혼동되는 경우가 있습니다. CoreDNS와 ExternalDNS는 역할이 완전히 분리되어 있으며, 서로를 대체하지 않습니다. 클러스터 내부 서비스 호출은 CoreDNS가 담당하고, 외부 도메인 접근은 ExternalDNS가 담당합니다. 두 프로젝트를 함께 사용하면 내부/외부 DNS 관리가 자동화되어 운영 효율성이 극대화됩니다. 실제로 대규모 SaaS 환경이나 하이브리드 클라우드 환경에서는 두 프로젝트의 조합이 필수적으로 적용되고 있으며, 내부와 외부 DNS 관리의 자동화와 장애 대응력 향상에 중요한 역할을 하고 있습니다.

4.3.2 CoreDNS + ExternalDNS 협력 트래픽 흐름과 실무 FAQ

CoreDNS와 ExternalDNS의 협력 구조는 Kubernetes 환경에서 내부와 외부 트래픽 흐름을 명확하게 분리하고, 각 계층의 DNS 관리가 자동화되어 운영 효율성과 장애 대응력을 높여줍니다. 외부→내부 트래픽 흐름 단계별 도식을 통해 실제 운영 환경에서 어떻게 동작하는지 설명드리겠습니다.

1. 개발자가 LoadBalancer/Ingress 서비스 생성
2. ExternalDNS가 외부 DNS 프로바이더(예: Route 53)에 `api.example.com` → `LB IP` A 레코드 자동 등록
3. 외부 클라이언트가 `api.example.com`으로 접근
4. LoadBalancer가 트래픽을 클러스터 내부로 전달
5. 클러스터 내부에서 다른 서비스를 호출할 때는 CoreDNS가 `backend.default.svc.cluster.local` → ClusterIP로 해석

이 흐름은 외부 도메인 접근과 내부 서비스 호출이 완전히 분리되어, 각 계층의 DNS 관리가

자동화되고 충돌 없이 운영됩니다. 실제로 대규모 SaaS 환경이나 금융권에서는 외부 DNS 관리와 내부 서비스 디스커버리의 분리가 장애 대응력과 보안 수준을 크게 높여주고 있습니다.

실무 FAQ를 통해 커뮤니티에서 자주 발생하는 혼동을 해소할 수 있습니다.

- Q1: “CoreDNS가 있으면 ExternalDNS가 필요 없나?”
  A: CoreDNS는 클러스터 내부만 담당하므로, 외부에서 도메인으로 접근하려면 ExternalDNS가 반드시 필요합니다.
- Q2: “ExternalDNS가 CoreDNS를 대체하나?”
  A: 아니다. ExternalDNS는 외부 DNS 프로바이더만 관리하며, Pod 간 내부 통신의 이름 해석은 전적으로 CoreDNS가 담당합니다.
- Q3: “두 프로젝트를 함께 쓸 때 설정 충돌이 있나?”
  A: 역할이 완전히 분리되어 있어 충돌 없이 독립적으로 운영됩니다. CoreDNS는 Corefile로, ExternalDNS는 Deployment args/annotation으로 각각 설정합니다.

CoreDNS와 ExternalDNS의 협력 구조는 내부/외부 DNS 관리의 자동화, 운영 효율성, 장애 대응력 향상 등 다양한 이점을 제공합니다. IT 의사결정자는 두 프로젝트를 함께 도입하여, 서비스 호출의 신뢰성과 확장성을 극대화할 수 있습니다. 실제로 두 프로젝트의 조합은 하이브리드 클라우드, 멀티 클러스터 환경에서 DNS 관리의 복잡성을 줄이고, 장애 대응과 운영 효율성을 크게 개선할 수 있습니다.

4.3.3 Consul, etcd 플러그인: 하이브리드/멀티 클러스터 서비스 디스커버리

Kubernetes 환경이 점점 대규모화되고, VM과 컨테이너, 온프레미스와 클라우드가 혼합된 하이브리드 환경으로 확장됨에 따라 서비스 디스커버리의 일관성과 확장성이 더욱 중요해지고 있습니다. Consul DNS Proxy를 통한 하이브리드 서비스 디스커버리 구조는 VM과 Kubernetes를 아우르는 통합 네임스페이스를 제공합니다. Kubernetes에서 Consul DNS Proxy를 통해 `.consul` 도메인을 CoreDNS가 포워딩하면, VM과 컨테이너 환경을 통합한 하이브리드 서비스 디스커버리를 구현할 수 있습니다. 이 구조는 멀티 클라우드, 온프레미스-클라우드 혼합 환경에서 서비스 호출의 일관성을 보장하며, 실제로 글로벌 SaaS 환경이나 대규모 금융권에서 적용되고 있습니다.

CoreDNS의 etcd 플러그인은 런타임 DNS 레코드를 동적으로 관리할 수 있습니다. 여러 클러스터에서 etcd를 공유하면, 서비스 이름 기반 호출이 클러스터 간 통합되어 멀티 클러스터 환경에서

DNS 기반 서비스 디스커버리가 가능합니다. Federation 플러그인도 유사하게 여러 클러스터의 DNS 네임스페이스를 통합하여, 분산 환경에서 서비스 호출의 일관성을 제공합니다. 실제로 멀티 클러스터 환경에서는 Federation 플러그인과 etcd 플러그인을 조합하여, 각 클러스터의 서비스 정보를 DNS 레코드로 통합 관리할 수 있습니다. 이 구조는 대규모 분산 환경에서 서비스 호출의 신뢰성과 확장성을 극대화하며, 클라우드 네이티브 아키텍처의 핵심 기반이 됩니다.

Consul, etcd, Federation 플러그인은 Kubernetes 환경을 넘어 VM, 온프레미스, 멀티 클라우드까지 아우르는 서비스 디스커버리의 일관성을 제공합니다. IT 의사결정자는 이 구조를 활용하여, 복잡한 분산 환경에서도 안정적인 서비스 호출을 구현할 수 있습니다. 실제로 하이브리드 클라우드 환경이나 멀티 클러스터 운영 환경에서는 Consul DNS Proxy와 CoreDNS의 포워딩 구조, Federation 플러그인, etcd 플러그인 조합이 서비스 호출의 일관성과 장애 대응력을 크게 높여주고 있습니다. 이러한 구조는 글로벌 SaaS 환경, 금융권, 대규모 이커머스 플랫폼 등에서 실무적으로 검증된 전략적 가치가 있습니다.

4.4 글로벌 프로덕션 검증 사례

CoreDNS와 CNCF 서비스 호출 파이프라인은 글로벌 대규모 조직에서 실증된 안정성과 성능을 제공합니다. CERN, SoundCloud, Infoblox, Skyscanner 등 다양한 사례는 서비스 이름 기반 호출의 신뢰성, 성능, 운영 효율성을 구체적으로 보여줍니다. 이 섹션에서는 글로벌 사례와 국내 현황, 비즈니스 환경별 적용 전략을 분석합니다. 실제로 다양한 산업군에서 CoreDNS와 관련 기술의 도입은 서비스 호출의 신뢰성과 확장성, 장애 대응력, 운영 효율성을 크게 개선하고 있습니다.

4.4.1 대규모 서비스 호출 환경의 실증 사례

글로벌 대규모 조직에서 CoreDNS와 CNCF 서비스 호출 파이프라인의 안정성과 성능이 실증된 사례를 살펴보면, 각 산업군에서 서비스 이름 기반 호출이 운영 효율성과 장애 대응력, 성능 최적화에 중요한 역할을 하고 있음을 알 수 있습니다.

CERN(유럽 입자 물리 연구소)은 대규모 과학 연구 인프라의 컨테이너화와 서비스 간 통신 자동화가 필수였습니다. Kubernetes + CoreDNS 기반 서비스 디스커버리 도입으로 새 클러스터 배포 시간이 3시간 이상에서 15분 미만으로, 노드 추가 시간은 1시간 이상에서 2분 미만으로 단축되었습니다. 이 사례는 서비스 이름 기반 호출이 대규모 환경에서 운영 효율성을 극대화함을

보여줍니다.

SoundCloud는 대규모 마이크로서비스 환경에서 초당 수십만 건의 DNS 서비스 디스커버리 요청을 처리해야 했습니다. CoreDNS를 내부 캐시 및 프록시로 사용하여, 안정적으로 높은 QPS를 지원하고 서비스 호출의 신뢰성을 확보했습니다. 이 사례는 CoreDNS의 성능과 확장성이 글로벌 대규모 환경에서 검증되었음을 입증합니다.

Infoblox는 Fortune 500 고객 대상 SaaS DNS 서비스에 CoreDNS를 도입하여, 글로벌 다중 인스턴스 배포와 모든 프로덕션 DNS 트래픽을 CoreDNS로 처리했습니다. 이 사례는 상용 DNS 서비스에서도 CoreDNS의 신뢰성과 확장성이 실증되었음을 보여줍니다.

Skyscanner는 대규모 환경에서 서비스 호출 시 DNS 쿼리 레이턴시가 과다하게 발생하는 문제를 해결하기 위해 CoreDNS Autopath 플러그인을 적용했습니다. 그 결과, Pod당 DNS 라운드 트립이 5회에서 1회로 감소하여 서비스 호출 성능이 크게 개선되었습니다. 이 사례는 DNS 호출 최적화의 대표적 성공 사례입니다.

이처럼 다양한 글로벌 조직에서 CoreDNS와 서비스 호출 파이프라인의 도입은 서비스 이름 기반 호출의 신뢰성과 성능, 운영 효율성을 획기적으로 개선하고 있습니다. 실제로 대규모 SaaS 환경, 금융권, 이커머스 플랫폼 등에서 CoreDNS와 관련 기술의 도입은 장애 대응력과 확장성, 운영 효율성을 크게 높여주고 있습니다.

4.4.2 국내 현황과 IT 의사결정자를 위한 적용 전략

한국 Kubernetes 사용 기업 전체의 CoreDNS 운영 현황을 살펴보면, Kubernetes 1.13 이후 CoreDNS가 기본 DNS이므로, K8s를 사용하는 모든 한국 기업이 자동으로 CoreDNS 기반 서비스 호출을 운영 중입니다. 금융권(우체국금융, 하나금융 등)은 Kubernetes 기반 핵심 과제를 추진 중이며, 이커머스, 게임, 서비스 플랫폼 분야에서도 이미 도입이 완료되었습니다. 실제로 국내 대규모 SaaS 환경이나 금융권, 이커머스 플랫폼에서 CoreDNS와 관련 기술의 도입은 서비스 호출의 신뢰성과 확장성, 장애 대응력, 운영 효율성을 크게 개선하고 있습니다.

비즈니스 환경별 적용 전략 매트릭스를 통해 각 산업군에서 CoreDNS와 관련 기술을 어떻게 조합하여 최적의 서비스 호출 인프라를 구축할 수 있는지 구체적으로 설명드리겠습니다.

비즈니스 환경	서비스 호출 방식	CoreDNS 역할	기대 효과
마이크로서비스 전환	서비스 이름 기반 호출	서비스 디스커버리 핵심	IP 하드코딩 제거, 동적 등록
하이브리드 클라우드	Split DNS + Forward	DNS 라우팅 허브	온프레/클라우드 통합 네임스페이스
멀티 클러스터 운영	Federation + etcd	클러스터 간 DNS 통합	분산 환경 서비스 디스커버리
대규모 트래픽	NodeLocal DNSCache 병행	DNS 캐싱 계층	쿼리 지연 밀리초→마이크로초
서비스 메시 도입	Istio/Linkerd + CoreDNS	기반 DNS 인프라	L7 라우팅, mTLS 보안

이 매트릭스는 IT 의사결정자가 비즈니스 환경에 따라 CoreDNS와 관련 기술을 조합하여 최적의 서비스 호출 인프라를 구축할 수 있도록 지원합니다. 실제로 국내 금융권이나 대규모 SaaS 환경에서는 Split DNS, Federation 플러그인, NodeLocal DNSCache, 서비스 메시 도입 등 다양한 기술 조합이 적용되고 있으며, 장애 대응력과 운영 효율성, 확장성, 보안 수준을 획기적으로 높여주고 있습니다.

CoreDNS는 Kubernetes 클러스터 생성 시 자동 배포되며, 별도 도입 결정이 필요 없습니다. 핵심은 애플리케이션 코드에서 IP 하드코딩을 서비스 이름 기반 호출로 전환하는 것이며, 서비스 메시 도입은 CoreDNS 기반 호출이 안정화된 후 단계적으로 검토해야 합니다. 실제로 국내외 다양한 산업군에서 CoreDNS와 관련 기술의 도입은 서비스 호출의 신뢰성과 확장성, 장애 대응력, 운영 효율성을 크게 개선하고 있습니다.

(선택) 도입 시 고려사항

CoreDNS와 CNCF 서비스 호출 파이프라인은 Kubernetes 클러스터 생성 시 자동으로 배포되므로, 별도 도입 결정이 필요 없습니다. 기존 kube-dns 환경에서는 `kubeadm upgrade`로 자동 전환이 가능하며, 인프라 추가 비용은 거의 없고 DevOps/SRE 1명이면 충분히 운영할 수 있습니다. 핵심은 애플리케이션 코드에서 IP 하드코딩을 서비스 이름 기반 호출로 전환하는 것이며, 서비스 메시 도입은 CoreDNS 기반 호출이 안정화된 후 단계적으로 검토하는 것이 바람직합니다. 상세 마이그레이션 절차는 Kubernetes 공식 도입 가이드와 CoreDNS 마이그레이션 가이드를 참고하시기 바랍니다.

5장: 서비스 호출 안정성을 위한 프로덕션 운영 모범 사례

서비스 호출의 안정성은 Kubernetes 환경에서 CoreDNS가 제공하는 서비스 디스커버리의 신뢰성과 직결됩니다. CoreDNS는 클러스터 내부의 모든 서비스 이름 해석을 담당하며, 장애 발생 시 전체 서비스 호출이 중단될 수 있습니다. 따라서 프로덕션 환경에서는 CoreDNS와 관련된 장애 패턴을 사전에 예방하고, 리소스 관리 및 스케일링 가이드라인을 준수하며, 보안 취약점에 대한 신속한 대응과 모니터링 체계를 구축하는 것이 필수적입니다. 본 장에서는 실제 운영 환경에서 서비스 호출 안정성을 확보하기 위한 모범 사례를 구체적으로 제시합니다.

5.1 서비스 호출 중단을 일으키는 주요 장애 패턴

Kubernetes에서 서비스 이름 기반 호출이 가능하게 하는 CoreDNS는 클러스터의 핵심 인프라입니다. 하지만 CoreDNS가 정상적으로 동작하지 않거나 네트워크, 권한, 리소스 등에서 문제가 발생하면 전체 서비스 호출이 중단될 수 있습니다. 이 섹션에서는 대표적인 장애 패턴과 그 예방 및 대응 방법을 상세히 설명합니다.

5.1.1 Loop Detection, NetworkPolicy, RBAC 장애와 예방

CoreDNS를 운영하는 과정에서 발생할 수 있는 장애 중 대표적인 것은 Loop Detection, NetworkPolicy, 그리고 RBAC 권한 부족에 의한 장애입니다. 이들 장애는 클러스터 내 서비스 호출의 중단을 유발할 수 있으며, 각 장애 유형별로 사전 예방과 신속한 대응이 중요합니다. 실제 운영 환경에서는 장애 발생 시 즉각적인 로그 분석과 상태 점검이 필수이며, 장애 예방을 위한 설정 검토와 정책 수립이 요구됩니다. 아래에서는 각 장애 유형별 발생 원인과 예방, 대응 방법을 상세히 설명합니다.

Loop Detection 장애와 CrashLoopBackOff

Loop Detection은 CoreDNS가 DNS 쿼리 처리를 반복적으로 자기 자신에게 요청하는 상황을 감지하는 기능입니다. 특히 systemd-resolved가 Pod의 /etc/resolv.conf에 127.0.0.53을 nameserver로 설정할 경우, CoreDNS가 해당 주소로 쿼리를 전달하면서 무한 루프가 발생할 수 있습니다. 이로 인해 CoreDNS Pod가 CrashLoopBackOff 상태에 빠지고, 클러스터 내 모든 서비스 이름 해석이 중단됩니다. 예방을 위해서는 kubelet의 DNS 설정을 검토하고,

/etc/resolv.conf에 CoreDNS ClusterIP만 지정되도록 관리해야 합니다. 또한 CoreDNS의 loop 플러그인을 활성화하여 반복 쿼리 감지 시 자동으로 오류를 반환하도록 설정해야 합니다.

이러한 장애는 실제로 대규모 클러스터에서 빈번하게 발생하며, 장애 발생 시에는 CoreDNS Pod의 로그를 확인하여 loop 감지 메시지와 CrashLoopBackOff 상태를 파악해야 합니다. 문제 해결을 위해서는 systemd-resolved 설정을 수정하거나, CoreDNS의 loop 플러그인 설정을 강화하는 것이 필요합니다. 또한 장애 예방을 위해 kubelet의 DNS 설정 변경 시마다 영향도를 사전 검토하고, 운영 매뉴얼에 loop 감지 및 대응 절차를 포함해야 합니다.

NetworkPolicy에 의한 DNS 차단

Kubernetes의 NetworkPolicy는 Pod 간 네트워크 트래픽을 제어하는 보안 기능입니다. UDP 또는 TCP 53 포트에 대한 egress(출구) 트래픽이 차단되면, Pod에서 CoreDNS로의 DNS 쿼리가 전달되지 않아 모든 서비스 이름 해석이 실패합니다. 이는 서비스 간 통신이 완전히 중단되는 심각한 장애로 이어집니다. 예방을 위해서는 NetworkPolicy 설정에서 kube-system 네임스페이스의 CoreDNS Pod로 UDP/TCP 53 포트 트래픽을 허용하는 규칙을 반드시 추가해야 합니다. 운영 중에는 DNS 해석 실패 로그를 지속적으로 모니터링하여 정책 변경에 따른 영향도 즉시 파악할 수 있어야 합니다.

실제 운영 환경에서는 NetworkPolicy 변경 시 DNS 트래픽이 차단되는 사례가 종종 발생하며, 장애 발생 시에는 Pod의 DNS 해석 실패 로그를 분석하여 NetworkPolicy 설정을 점검해야 합니다. 예방을 위해서는 정책 변경 전후 DNS 해석 테스트를 실시하고, 운영팀 내 정책 변경 절차에 DNS 영향도 검토를 포함해야 합니다. 또한 장애 대응을 위한 NetworkPolicy 롤백 및 예외 규칙 추가 방법을 숙지해야 합니다.

RBAC 권한 부족에 따른 서비스 레코드 미생성

CoreDNS는 kubernetes 플러그인을 통해 K8s API를 watch하여 서비스 및 엔드포인트 정보를 실시간으로 반영합니다. 이 과정에서 system:coredns ClusterRole이 누락되거나, 서비스 계정에 필요한 권한이 부여되지 않으면 K8s API watch가 실패하고, DNS 레코드가 생성되지 않습니다. 결과적으로 신규 서비스나 Pod가 추가되어도 이름 해석이 불가능해집니다. 예방을 위해서는 CoreDNS의 서비스 계정에 system:coredns ClusterRole을 부여하고, RoleBinding을 정확히 설정해야 합니다. RBAC 설정 변경 시에는 CoreDNS 로그를 통해 watch 오류 여부를 즉시 확인해야 합니다.

실무적으로는 RBAC 권한 부족으로 인한 장애가 신규 클러스터 구축이나 CoreDNS 업그레

이드 시 자주 발생합니다. 장애 발생 시에는 CoreDNS Pod의 로그에서 “permission denied” 또는 “watch failed” 메시지를 확인하고, RBAC 설정을 점검해야 합니다. 예방을 위해서는 RBAC 정책 변경 시마다 CoreDNS의 서비스 계정과 RoleBinding을 검토하고, 운영팀 내 RBAC 변경 절차에 DNS 영향도 검토를 포함해야 합니다.

장애 대응 및 예방의 실무적 원칙

장애 발생 시에는 CoreDNS Pod의 상태와 로그를 우선적으로 점검해야 하며, CrashLoop-BackOff, DNS 해석 실패, RBAC 오류 등 각 장애 유형별로 대응 매뉴얼을 마련해야 합니다. 장애 예방을 위해서는 kubelet, NetworkPolicy, RBAC 등 클러스터 주요 구성 요소의 변경 시마다 DNS 서비스 영향도를 사전 검토하는 것이 중요합니다. 또한 장애 발생 시 신속한 롤백과 설정 복구를 위한 자동화 스크립트와 운영 매뉴얼을 준비해 두는 것이 실무적으로 매우 중요합니다. 장애 예방과 대응을 위한 교육과 점검 절차를 정기적으로 실시하여, 서비스 호출 안정성을 지속적으로 유지해야 합니다.

5.1.2 리소스 관리와 스케일링 가이드라인

CoreDNS의 안정적인 운영을 위해서는 리소스 관리와 스케일링 전략이 매우 중요합니다. 클러스터의 규모와 트래픽 변화에 따라 CoreDNS의 처리량, 메모리 사용량, 레플리카 수, DNS 캐싱 계층 배포 상태를 지속적으로 모니터링하고, 장애 발생 시 신속하게 리소스 확장 또는 캐싱 계층 추가를 결정할 수 있는 운영 체계를 마련해야 합니다. 본 섹션에서는 CPU 및 처리량 한계, Autopath 플러그인 메모리 공식, NodeLocal DNSCache 병행 운영, AWS ENI DNS 스로틀링 등 주요 리소스 관리 및 스케일링 가이드라인을 구체적으로 설명합니다.

CPU 및 처리량 한계와 확장 전략

CoreDNS는 Go 기반 멀티스레드 컨테이너로 설계되어 높은 처리량을 제공하지만, CPU 2코어 환경에서 약 90,000~100,000 QPS(Queries Per Second)의 한계가 있습니다. 4코어로 확장해도 처리량이 크게 증가하지 않는 현상이 보고되고 있으므로, 클러스터 규모에 따라 CoreDNS Pod의 레플리카 수를 조정하는 것이 필수적입니다. 기본적으로 2개의 레플리카를 배포하며, 10개를 초과하는 경우에는 운영 복잡성이 증가하므로 권장되지 않습니다.

실제 운영 환경에서는 클러스터의 서비스 수와 트래픽 패턴을 분석하여 CoreDNS의 레플리카 수를 결정합니다. 예를 들어, 대규모 클러스터에서는 QPS가 급격히 증가할 수 있으므로,

Prometheus를 활용하여 QPS와 레이턴시를 모니터링하고, 필요 시 레플리카 수를 동적으로 확장해야 합니다. 또한 CoreDNS의 CPU 및 메모리 리소스 요청/제한 설정을 최적화하여, 장애 발생 시 신속한 리소스 확장과 Pod 재배포가 가능하도록 운영해야 합니다.

Autopath 플러그인 메모리 공식과 최적화

Autopath 플러그인은 DNS 쿼리의 라운드 트립 횟수를 줄여 성능을 최적화하지만, 메모리 사용량이 증가할 수 있습니다. 공식적으로 메모리 사용량은 (Pod 수 + Service 수) / 250 + 56(MB)로 계산합니다. 대규모 클러스터에서는 이 공식에 따라 메모리 리소스를 충분히 할당해야 하며, 필요에 따라 Autopath 플러그인 활성화 여부를 조정해야 합니다.

운영 환경에서는 Autopath 플러그인 활성화 시 메모리 사용량이 급격히 증가하는 사례가 보고되고 있습니다. 따라서 클러스터의 Pod 수와 서비스 수를 정확히 파악하고, 메모리 리소스 할당을 공식에 따라 계산해야 합니다. 또한 Autopath 플러그인 활성화 여부를 서비스 호출 패턴과 트래픽 분석을 기반으로 결정하고, 장애 발생 시에는 Autopath 플러그인 설정을 조정하여 메모리 사용량을 최적화해야 합니다.

NodeLocal DNSCache 병행 운영의 필요성

100노드 이상 대규모 클러스터에서는 CoreDNS 단독 운영 시 DNS 쿼리 지연이 발생할 수 있습니다. NodeLocal DNSCache를 병행 배포하면 CoreDNS의 부하를 70~90%까지 감소시키고, DNS 쿼리 지연을 밀리초 단위에서 마이크로초 단위로 줄일 수 있습니다. 실제 운영에서는 NodeLocal DNSCache를 DaemonSet 형태로 각 노드에 배포하고, Pod의 /etc/resolv.conf를 NodeLocal DNSCache IP(169.254.20.10 등)로 설정하여 로컬 캐싱을 활용합니다.

NodeLocal DNSCache는 대규모 클러스터에서 DNS 쿼리 지연과 CoreDNS 부하 문제를 효과적으로 해결할 수 있으며, 장애 발생 시에는 NodeLocal DNSCache의 상태와 로그를 점검하여 DNS 캐싱 계층의 정상 동작 여부를 확인해야 합니다. 운영팀에서는 NodeLocal DNSCache 배포 시 각 노드의 리소스 사용량과 DNS 쿼리 패턴을 모니터링하고, 장애 발생 시에는 NodeLocal DNSCache와 CoreDNS 간의 트래픽 분산 및 캐싱 계층 확장 전략을 수립해야 합니다.

AWS ENI DNS 스로틀링과 클라우드 환경 대응

AWS Elastic Network Interface(ENI)는 DNS 트래픽에 대해 초당 1,024 패킷의 하드 리밋을 적용합니다. 대규모 트래픽 환경에서는 이 한계를 초과하면 DNS 쿼리가 타임아웃되어 서비스 호출이 실패할 수 있습니다. 대응 방안으로는 CoreDNS 레플리카 수를 늘리고, NodeLocal DNSCache를 적극 활용하며, DNS 쿼리 패턴을 최적화하는 것이 중요합니다.

AWS 환경에서는 DNS 스로틀링으로 인한 장애가 빈번하게 발생할 수 있으므로, CoreDNS의 QPS와 DNS 트래픽 패턴을 지속적으로 모니터링하고, 장애 발생 시에는 ENI의 DNS 패킷 리밋을 확인해야 합니다. 예방을 위해서는 NodeLocal DNSCache를 활용하여 CoreDNS의 부하를 분산시키고, DNS 쿼리 패턴을 최적화하여 ENI의 하드 리밋을 초과하지 않도록 운영해야 합니다.

운영 가이드라인의 실무 적용

리소스 관리와 스케일링은 클러스터의 성장과 트래픽 변화에 따라 지속적으로 조정해야 합니다. CoreDNS의 처리량, 메모리 사용량, 레플리카 수, DNS 캐싱 계층의 배포 상태를 모니터링하며, 장애 발생 시 신속하게 리소스 확장 또는 캐싱 계층 추가를 결정할 수 있는 운영 체계를 마련해야 합니다. 또한 운영팀 내 리소스 관리 및 스케일링 정책을 정기적으로 점검하고, 장애 예방을 위한 자동화 스크립트와 운영 매뉴얼을 준비해야 합니다. 실무적으로는 리소스 모니터링 도구(Prometheus, Grafana 등)를 활용하여 CoreDNS의 상태를 실시간으로 관찰하고, 장애 발생 시 신속한 대응과 복구가 가능하도록 체계적인 운영 전략을 수립해야 합니다.

5.2 보안 취약점 관리와 DNS 보안

Kubernetes와 CoreDNS 환경에서 보안은 서비스 호출 안정성의 또 다른 핵심 요소입니다. DNS는 네트워크 공격의 주요 표적이 될 수 있으며, 취약점이 발견될 경우 전체 클러스터의 서비스 디스커버리 기능이 마비될 수 있습니다. 본 섹션에서는 CoreDNS의 주요 보안 취약점과 대응 전략을 상세히 설명합니다.

5.2.1 알려진 CVE와 보안 패치 전략

CoreDNS는 다양한 보안 취약점에 노출될 수 있으며, 특히 DNS 프로토콜의 특성상 외부 공격에 취약한 구조를 가지고 있습니다. 운영 환경에서는 CVE(Common Vulnerabilities and Exposures)로 공개된 취약점에 대한 신속한 패치 적용과 보안 프로토콜 활성화가 필수적입니다. 본 섹션에서는 DoQ 동시성 취약점, DNS 캐시 포이즈닝, TuDoor 공격 등 주요 CVE 사례와 대응 방안, 그리고 CoreDNS의 구조적 보안 이점과 최신 보안 프로토콜 지원에 대해 구체적으로 설명합니다.

DoQ 동시성 취약점(CVE-2025-47950)과 대응

CoreDNS의 DoQ(DNS over QUIC) 구현에서 동시성 제한 미비로 인해 메모리 고갈 공격이 가능하다는 CVE-2025-47950이 보고되었습니다. CVSS 점수 7.5로 높은 위험도를 가진 이

취약점은 대량의 DoQ 요청을 통해 CoreDNS의 메모리를 고갈시키고, 서비스 호출 장애를 유발할 수 있습니다. 대응을 위해서는 최신 CoreDNS 릴리스에서 DoQ 동시성 제한 패치를 적용하고, DoQ 사용이 불필요한 환경에서는 해당 프로토콜을 비활성화하는 것이 권장됩니다.

실제 운영 환경에서는 DoQ 취약점이 악용되어 대규모 서비스 장애가 발생할 수 있으므로, 운영팀에서는 CoreDNS의 DoQ 설정을 정기적으로 점검하고, 최신 패치 적용 여부를 확인해야 합니다. 또한 DoQ 사용이 불필요한 경우에는 CoreDNS 설정에서 DoQ 플러그인을 비활성화하여 공격 표면을 최소화해야 합니다. 장애 발생 시에는 CoreDNS Pod의 메모리 사용량과 DoQ 트래픽 패턴을 분석하여, 신속한 대응과 복구가 가능하도록 운영 매뉴얼을 준비해야 합니다.

DNS 캐시 포이즈닝(CVE-2023-30464)과 TuDoor 공격(CVE-2023-28452)

DNS 캐시 포이즈닝은 악의적인 쿼리로 CoreDNS의 캐시를 오염시켜 잘못된 IP를 반환하게 만드는 공격입니다. CVE-2023-30464는 이러한 캐시 포이즈닝 취약점을 다루며, 패치 적용을 통해 캐시 검증 로직을 강화해야 합니다. TuDoor 공격(CVE-2023-28452)은 DNS 응답의 일관성 검증 미비를 악용하여 서비스 호출을 방해하는 취약점입니다. 대응 방안으로는 CoreDNS의 최신 보안 패치를 적용하고, 캐시 무결성 검증 기능을 활성화하는 것이 중요합니다.

운영 환경에서는 DNS 캐시 포이즈닝과 TuDoor 공격으로 인해 서비스 호출이 중단되는 사례가 발생할 수 있습니다. 예방을 위해서는 CoreDNS의 캐시 검증 로직을 강화하고, 최신 보안 패치를 신속히 적용해야 합니다. 또한 장애 발생 시에는 CoreDNS의 캐시 상태와 DNS 응답 일관성을 점검하여, 공격 여부를 신속히 파악하고 대응해야 합니다.

Go 기반 메모리 안전성과 dnsmasq 대비 구조적 이점

CoreDNS는 Go 언어 기반으로 개발되어 메모리 안전성이 높으며, C 기반 dnsmasq에서 빈번하게 발생하는 버퍼 오버플로우, 포인터 오류 등 구조적 취약점을 원천적으로 제거합니다. 이는 Kubernetes 환경에서 서비스 호출 안정성을 보장하는 핵심 요소입니다.

실무적으로는 CoreDNS의 Go 기반 구조 덕분에 메모리 관련 취약점이 크게 감소하며, 운영팀에서는 dnsmasq 대비 CoreDNS의 구조적 이점을 활용하여 보안 위협에 효과적으로 대응할 수 있습니다. 또한 CoreDNS의 메모리 관리와 오류 처리 로직을 정기적으로 점검하여, 서비스 호출 안정성을 지속적으로 유지해야 합니다.

DoT/DoH 보안 프로토콜 지원

CoreDNS는 DoT(DNS over TLS, RFC 7858)와 DoH(DNS over HTTPS, RFC 8484) 등 최신 보안 프로토콜을 지원합니다. 이를 통해 DNS 쿼리와 응답이 암호화되어 중간자 공격(MITM)

이나 패킷 스니핑에 대한 방어가 가능합니다. 프로덕션 환경에서는 DoT/DoH를 활성화하고, TLS 인증서를 정기적으로 갱신하는 것이 권장됩니다.

운영 환경에서는 DoT/DoH 활성화 시 DNS 트래픽이 암호화되어 보안 위협이 크게 감소합니다. 장애 발생 시에는 TLS 인증서 상태와 DoT/DoH 트래픽 패턴을 점검하여, 암호화 프로토콜의 정상 동작 여부를 확인해야 합니다. 예방을 위해서는 TLS 인증서 갱신 및 DoT/DoH 설정 변경 시마다 DNS 서비스 영향도를 사전 검토하고, 운영 매뉴얼에 암호화 프로토콜 활성화 절차를 포함해야 합니다.

보안 패치 적용과 운영 원칙

보안 취약점이 보고된 경우에는 CNCF, CoreDNS GitHub, CVE 데이터베이스를 통해 신속히 패치 정보를 확인하고, 클러스터 내 모든 CoreDNS 인스턴스에 최신 패치를 적용해야 합니다. 운영 중에는 보안 로그와 DNS 트래픽 패턴을 지속적으로 모니터링하여 이상 징후를 조기에 탐지할 수 있어야 합니다.

실무적으로는 보안 패치 적용 시 장애 발생 가능성을 최소화하기 위해 테스트 환경에서 사전 검증을 실시하고, 운영팀 내 패치 적용 절차와 롤백 방안을 마련해야 합니다. 또한 보안 로그와 DNS 트래픽 패턴을 실시간으로 모니터링하여, 보안 위협 발생 시 신속한 대응과 복구가 가능하도록 체계적인 운영 전략을 수립해야 합니다.

5.3 모니터링과 관찰성

서비스 호출 안정성을 장기적으로 유지하기 위해서는 CoreDNS의 상태와 DNS 쿼리 패턴을 실시간으로 모니터링하는 관찰성 체계가 필수적입니다. 본 섹션에서는 Prometheus, Grafana, Datadog 등 모니터링 도구를 활용한 실무적 모니터링 방법을 설명합니다.

5.3.1 Prometheus/Grafana를 통한 서비스 호출 패턴 모니터링

CoreDNS의 서비스 호출 안정성을 확보하기 위해서는 실시간 모니터링과 관찰성 체계 구축이 매우 중요합니다. 운영 환경에서는 Prometheus, Grafana, Datadog 등 다양한 모니터링 도구를 활용하여 CoreDNS의 상태와 DNS 쿼리 패턴을 분석하고, 장애 예방과 성능 최적화, 보안 위협 탐지까지 포함하는 종합적 운영 전략을 수립해야 합니다. 본 섹션에서는 CoreDNS /metrics 엔드포인트와 Prometheus 연동, Grafana 대시보드 시각화, Datadog Agent 체크, DNS 쿼리

패턴 분석 등 실무적 모니터링 방법을 구체적으로 설명합니다.

CoreDNS /metrics 엔드포인트와 Prometheus 연동

CoreDNS는 9153 포트의 /metrics 엔드포인트를 통해 OpenMetrics 형식의 메트릭 데이터를 제공합니다. Prometheus는 이 엔드포인트를 scrape하여 QPS, 레이턴시, 캐시 히트율, 실패율 등 다양한 핵심 메트릭을 수집합니다. 대표적인 Prometheus 설정 예시는 다음과 같습니다:

```
scrape_configs:
  - job_name: 'coredns'
    kubernetes_sd_configs:
      - role: pod
    relabel_configs:
      - source_labels: [__meta_kubernetes_namespace,
↪   __meta_kubernetes_pod_label_app]
        action: keep
        regex: kube-system;coredns
    metrics_path: /metrics
    port: 9153
```

운영 환경에서는 Prometheus를 활용하여 CoreDNS의 QPS, 레이턴시, 캐시 히트율, 실패율 등 주요 지표를 실시간으로 모니터링하고, 장애 발생 시에는 메트릭 데이터를 분석하여 원인을 신속히 파악해야 합니다. 또한 Prometheus 설정 변경 시에는 CoreDNS의 /metrics 엔드포인트와 포트 번호를 정확히 지정하고, 메트릭 데이터 수집 주기를 최적화하여 운영 효율성을 높여야 합니다.

Grafana 전용 대시보드와 시각화

Grafana는 Prometheus에서 수집한 CoreDNS 메트릭을 시각화하는 대시보드를 제공합니다. DNS 쿼리 처리량, 평균/최대 레이턴시, 캐시 히트율, NXDOMAIN/FAILURE 비율 등 주요 지표를 실시간으로 모니터링할 수 있습니다. 운영자는 대시보드를 통해 서비스 호출 패턴의 이상 징후를 빠르게 파악하고, 장애 발생 시 원인을 신속히 분석할 수 있습니다.

실무적으로는 Grafana 대시보드를 활용하여 CoreDNS의 상태를 시각적으로 분석하고, 장애 발생 시에는 대시보드에서 이상 지표를 확인하여 신속한 대응과 복구가 가능하도록 운영해야 합니다. 또한 Grafana 대시보드 설정 변경 시에는 주요 지표와 알림 조건을 최적화하여, 운영팀 내 장애 대응 효율성을 높여야 합니다.

Datadog Agent CoreDNS 체크와 OpenMetrics 기반 통합

Datadog은 CoreDNS용 전용 Agent 체크를 제공하며, OpenMetrics 기반으로 QPS, 레이

턴시, 캐시 상태 등 다양한 지표를 통합 모니터링합니다. Datadog의 알림 시스템을 활용하면 DNS 해석 실패율이 임계값을 초과할 때 자동으로 경보를 발송할 수 있습니다.

운영 환경에서는 Datadog Agent를 활용하여 CoreDNS의 상태를 실시간으로 모니터링하고, 장애 발생 시에는 알림 시스템을 통해 신속한 대응이 가능합니다. 또한 Datadog 설정 변경 시에는 CoreDNS 체크와 OpenMetrics 연동을 최적화하여, 운영팀 내 장애 예방과 대응 효율성을 높여야 합니다.

DNS 쿼리 패턴 분석과 서비스 호출 관계 파악

CoreDNS의 모니터링 데이터를 활용하면 어떤 서비스가 어떤 서비스를 얼마나 자주 호출하는지 DNS 쿼리 패턴으로 파악할 수 있습니다. 이는 서비스 간 의존성, 호출 빈도, 트래픽 분포, 장애 전파 경로를 분석하는 데 매우 유용하며, 운영자는 이를 기반으로 서비스 구조 최적화, 트래픽 분산, 장애 대응 전략을 수립할 수 있습니다.

실무적으로는 DNS 쿼리 패턴 분석을 통해 서비스 간 의존성 및 호출 빈도를 파악하고, 장애 발생 시에는 쿼리 패턴 데이터를 활용하여 장애 전파 경로와 영향도를 신속히 분석해야 합니다. 또한 서비스 구조 최적화와 트래픽 분산 전략을 수립하여, 서비스 호출 안정성을 지속적으로 유지해야 합니다.

관찰성 체계의 실무 적용

관찰성은 단순한 모니터링을 넘어, 장애 예방과 성능 최적화, 보안 위협 탐지까지 포함하는 종합적 운영 전략입니다. CoreDNS의 메트릭 수집, 대시보드 시각화, 알림 연동, 쿼리 패턴 분석을 체계적으로 구축하여 서비스 호출 안정성을 장기적으로 유지해야 합니다. 또한 운영팀 내 관찰성 체계 구축 및 점검 절차를 정기적으로 실시하고, 장애 발생 시에는 관찰성 데이터를 활용하여 신속한 대응과 복구가 가능하도록 체계적인 운영 전략을 수립해야 합니다.

Appendix

References

1. AWS CoreDNS DNS 스로틀링:https://aws.amazon.com/blogs/mt/monitoring-coredns-for-dns-throttling-issues-using-aws-open-source-monitoring-ser

vices/

2. AWS EKS CoreDNS.https://docs.aws.amazon.com/eks/latest/userguide/managing-coredns.html
3. Author/Organization. (Year). “Title”. URL
4. CNCF CoreDNS Graduation.https://www.cncf.io/announcements/2019/01/24/coredns-graduation/
5. CNCF CoreDNS Project.https://www.cncf.io/projects/coredns/
6. CNCF Projects.https://www.cncf.io/projects/
7. Consul DNS.https://developer.hashicorp.com/consul/docs/discover/dns
8. CoreDNS Migration Guide.https://coredns.io/2018/05/21/migration-from-kube-dns-to-coredns/
9. CoreDNS Query Processing:https://coredns.io/2017/06/08/how-queries-are-processed-in-coredns/
10. CoreDNS kubernetes 플러그인.https://coredns.io/plugins/kubernetes/
11. CoreDNS vs Kube-DNS.https://coredns.io/2018/11/27/cluster-dns-coredns-vs-kube-dns/
12. CoreDNS 공식 문서:https://coredns.io/plugins/kubernetes/
13. CoreDNS 공식 문서:https://coredns.io/plugins/loop/
14. Corefile 예시 및 플러그인 설명:https://coredns.io/plugins/
15. Datadog CoreDNS 체크:https://docs.datadoghq.com/integrations/coredns/
16. ExternalDNS GitHub.https://github.com/kubernetes-sigs/external-dns
17. GBHackers CoreDNS CVE:https://gbhackers.com/coredns-vulnerability/
18. GitHub ADOPTERS.md.https://github.com/coredns/coredns/blob/master/ADOPTERS.md
19. GitHub CoreDNS LICENSE.https://github.com/coredns/coredns/blob/master/LICENSE
20. GitHub CoreDNS.https://github.com/coredns/coredns
21. Grafana CoreDNS 대시보드:https://grafana.com/grafana/dashboards/11519
22. Infoblox CoreDNS K8s 1.13.https://blogs.infoblox.com/company/coredns-ado

pted-as-the-standard-dns-server-in-kubernetes-1-13/

23. Istio Performance.https://istio.io/latest/docs/ops/deployment/performance-and-scalability/
24. K8s CERN Case Study.https://kubernetes.io/case-studies/cern/
25. K8s DNS for Services and Pods.https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
26. K8s NodeLocal DNSCache.https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/
27. K8s NodeLocal DNSCache:https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/
28. K8s Service 공식 문서.https://kubernetes.io/docs/concepts/services-networking/service/
29. K8s Using CoreDNS.https://kubernetes.io/docs/tasks/administer-cluster/coredns/
30. Kubernetes 공식 문서:https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
31. Kubernetes 공식 문서:https://kubernetes.io/docs/tasks/administer-cluster/coredns/
32. Northflank 벤치마크.https://northflank.com/blog/performance-testing-for-core-dns
33. Northflank 벤치마크:https://northflank.com/blog/performance-testing-for-core-dns
34. O' Reilly Learning CoreDNS.https://www.oreilly.com/library/view/learning-coredns/9781492047957/ch01.html
35. Prometheus CoreDNS 모니터링:https://coredns.io/plugins/prometheus/
36. Service Mesh Performance (arxiv).https://arxiv.org/html/2411.02267v1
37. https://coredns.io/2018/11/27/cluster-dns-coredns-vs-kube-dns/
38. https://coredns.io/plugins/kubernetes/
39. https://fossa.com/blog/open-source-licenses-101-apache-license-2-0/

40. https://github.com/coredns/coredns/blob/master/ADOPTERS.md
41. https://github.com/coredns/coredns/blob/master/LICENSE
42. https://iximiuz.com/en/posts/service-discovery-in-kubernetes/
43. https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
44. https://mrkaran.dev/posts/ndots-kubernetes/
45. https://northflank.com/blog/performance-testing-for-core-dns
46. https://pracucci.com/kubernetes-dns-resolution-ndots-options-and-why-it-may-affect-application-performances.html
47. iximiuz Service Discovery.https://iximiuz.com/en/posts/service-discovery-in-kubernetes/
48. pracucci ndots 분석.https://pracucci.com/kubernetes-dns-resolution-ndots-options-and-why-it-may-affect-application-performances.html
49. 컴퓨터월드 K8s 시장.https://www.comworld.co.kr/news/articleView.html?idxno=50807

Glossary

용어	정의
서비스 메시	Istio, Linkerd 등 L7 트래픽 관리, mTLS, 관찰성 기능을 제공하는 마이크로서비스 네트워크 인프라.
플러그인 체인	CoreDNS에서 DNS 쿼리를 순차적으로 처리하는 플러그인 구조
Apache License 2.0	수정, 배포, 상용화에 제약 없는 오픈소스 라이선스
Autopath 플러그인	DNS search 도메인 확장 최적화 플러그인, DNS 라운드 트립 감소.
BIND 9	전통적 DNS 서버, 모놀리식 구조와 복잡한 설정.
ClusterIP	Kubernetes 서비스에 할당되는 가상 IP, 클러스터 내부에서만 접근 가능.
CNCF Graduated	CNCF에서 성숙도가 검증된 오픈소스 프로젝트 등급
Consul DNS Proxy	VM과 Kubernetes 환경을 통합하는 서비스 디스커버리 솔루션.
CoreDNS	Kubernetes 클러스터 내부에서 서비스 이름을 IP로 해석하는 공식 DNS 서버.
Corefile	CoreDNS 플러그인 체인과 설정을 정의하는 구성 파일

CrashLoopBackOff	Kubernetes Pod이 반복적으로 재시작되는 상태
CVE	Common Vulnerabilities and Exposures, 공개된 보안 취약점 목록
Datadog	클라우드 기반 모니터링 및 관찰성 서비스
DoQ	DNS over QUIC, DNS 쿼리 암호화 프로토콜
DoT/DoH	DNS over TLS/HTTPS, DNS 암호화 프로토콜
etcd	Kubernetes 클러스터 상태 정보를 저장하는 분산 키-값 저장소.
ExternalDNS	Kubernetes 리소스 변경에 따라 외부 DNS 프로바이더(AWS Route 53 등)에 레코드를 자동 동기화하는 오픈소스 프로젝트.
ExternalName Service	외부 도메인을 클러스터 내부 DNS에 CNAME으로 매핑하는 서비스 타입
Federation 플러그인	CoreDNS에서 멀티 클러스터 환경의 DNS 네임스페이스를 통합하는 플러그인.
FQDN	Fully Qualified Domain Name, 서비스의 전체 DNS 이름 형식
Grafana	오픈소스 데이터 시각화 및 대시보드 플랫폼
Headless Service	clusterIP 없이 모든 Pod IP를 A 레코드로 직접 반환하는 Kubernetes 서비스 타입
HPA	Horizontal Pod Autoscaler, Pod 수를 자동으로 조정하는 Kubernetes 컴포넌트
kube-dns	Kubernetes 초기 DNS 서버, dnsmasq 등 3컨테이너 구조로 운영.
kube-proxy	Kubernetes에서 서비스와 Pod 간 트래픽을 iptables 등으로 라우팅하는 컴포넌트.
kubernetes 플러그인	CoreDNS에서 K8s API를 watch하여 서비스 정보를 DNS 레코드로 자동 생성하는 플러그인
mTLS	Mutual TLS, 서비스 간 암호화 및 인증을 제공하는 보안 프로토콜.
ndots	DNS 쿼리 이름의 점 개수에 따라 search 도메인 확장 횟수를 결정하는 resolv.conf 옵션.
ndots:5	DNS 쿼리 이름에 점이 5개 미만이면 search 도메인을 붙여 여러 번 해석 시도하는 옵션
NetworkPolicy	Kubernetes에서 네트워크 트래픽을 제어하는 리소스
NodeLocal DNSCache	Kubernetes 노드별 DNS 캐싱 솔루션
NXDOMAIN	DNS에서 존재하지 않는 도메인에 대한 응답 코드.
Pod	Kubernetes에서 컨테이너화된 워크로드의 최소 단위
PowerDNS	GPL v2 기반 DNS 서버, 상용 이용에 제약 있음.
Prometheus	오픈소스 모니터링 및 알림 시스템
QPS	Queries Per Second, 초당 DNS 쿼리 처리량
RBAC	Role-Based Access Control, Kubernetes에서 권한을 관리하는 방식
Service	Kubernetes에서 여러 Pod를 하나의 논리적 엔드포인트로 묶는 추상화 계층
SRV 레코드	서비스 포트 정보를 제공하는 DNS 레코드 유형
StatefulSet	상태 유지가 필요한 워크로드를 위한 Kubernetes 리소스
xDS API	Envoy 프록시에서 동적 라우팅, 보안 정책, 서비스 디스커버리를 관리하는 API.

Endnotes

[1] CoreDNS와 kube-dns의 성능 차이는 실제 운영 환경에서 대규모 서비스 호출 지연 및 장애 대응에 결정적 영향을 미칩니다. [2] ndots 설정은 클러스터 전체의 DNS 쿼리 패턴과 외부 API 호출 성능에 직접적인 영향을 미치므로, 운영 환경에 맞게 조정해야 합니다.

Contact Us

hello@cncf.co.kr

02-469-5426

www.cncf.co.kr

CNF Blog

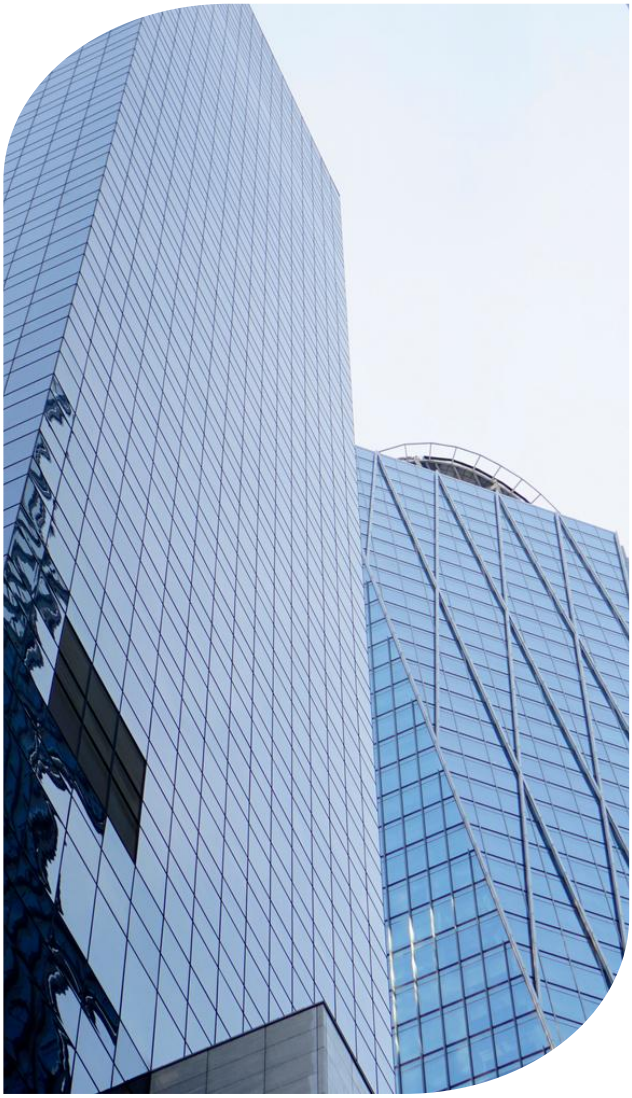
다양한 콘텐츠와 전문 지식을 통해
더 나은 경험을 제공합니다.

CNF eBook

이제 나도 클라우드 네이티브 전문가
쿠버네티스 구축부터 운영 완전 정복

CNF Resource

Community Solution의 최신 정보와
유용한 자료를 만나보세요.



씨엔에프 | CNF

전화 : (02) 469 - 5426
팩스 : (02) 469 - 7247
메일 : hello@cncf.co.kr