THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA

Embedded Linux Conference

# Why is it always DNS?:

# Rethinking & Engineering Node-Level DNS Resolution in Kubernetes

# It's Always DNS

*"It's not DNS.*
*There's no way it's DNS.*

*It was DNS."*

— Every engineer, 3 AM

**In Kubernetes, in-cluster service domains get resolved by CoreDNS daemon service.**

However, every cluster quietly depends on nameserver mentioned in `/etc/resolv.conf` for all non-service name resolution which usually gets redirected to external DNS.

**At scale, this dependency becomes a liability.**

# Meet the Speakers

## Ankur Singh

Senior Site Reliability Engineer

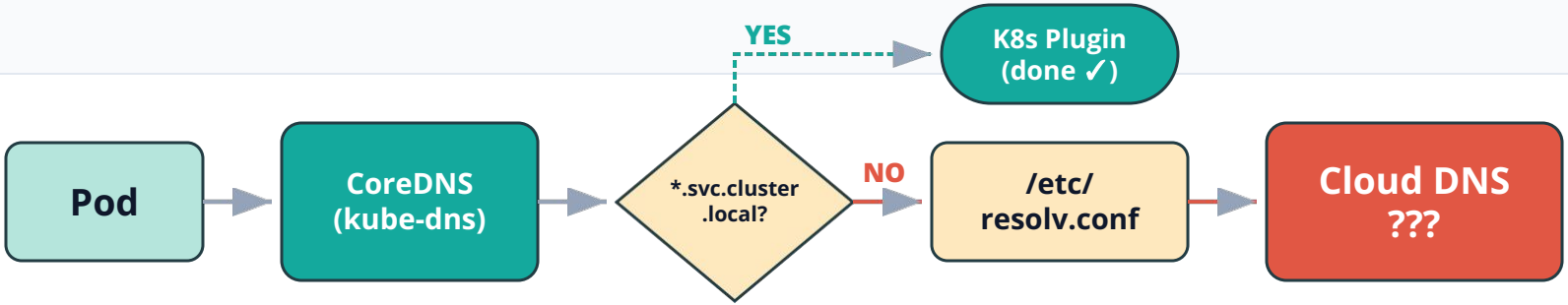
## Shaheen Sayyed

Site Reliability Engineer

# Agenda

| | |
|---|---|
| **The Problem** | How K8s consumes node DNS today |
| **Solution 1: dnsmasq** | Medium-scale, battle-tested DNS |
| ▶ **Live Demo** | *dnsmasq deployment + validation setup* |
| **Solution 2: CoreDNS Static Pods** | Large-scale, k8s-native DNS |
| ▶ **Live Demo** | *Custom CoreDNS + in-built observability* |
| **Choosing Your Approach** | Decision framework & trade-offs |

# How K8s Consumes Node DNS Today




Non-service queries leave the node via a cloud DNS endpoint you don't control

- API server address (api.cluster.example.com)
- Container registries (quay.io, gcr.io, docker.io)
- Webhook endpoints, OIDC providers
- External secrets backends
- Cloud metadata & storage endpoints
- Any non-.svc.cluster.local domain

# What's in /etc/resolv.conf?

| Cloud Provider | nameserver | You configured it? | You can observe it? |
|---|---|---|---|
| Azure | 168.63.129.16 | No | No |
| AWS | 169.254.169.253<br>(or VPC+2 address) | No | No |
| GCP | 169.254.169.254 | No | No |

**Three clouds. Same problem. Zero control.**

Each cloud sets one nameserver line. You didn't choose it.
You can't tune it. You can't see what it's doing. You depend on it completely.

# Five Problems at Scale

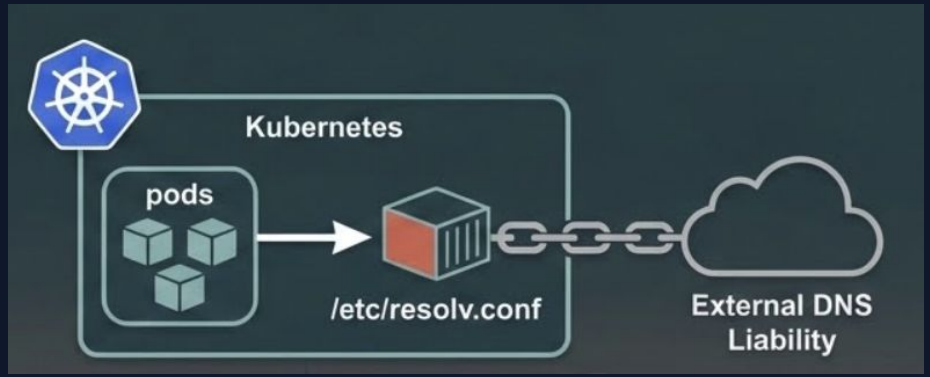
| | | |
|---|---|---|
| 1 | Fragile Forwarding | Single upstream, no fallback, single point of failure |
| 2 | No Caching Control | Cloud DNS TTL behavior is opaque and untunable |
| 3 | Zero Observability | No metrics, no logs, no per-query visibility |
| 4 | Upstream Overload | Hundreds of nodes hitting the same cloud DNS endpoint |
| 5 | Reduced Security | All non-service domain resolutions leave the cluster |

# The Core Challenge

Increasing resilience, observability & security for DNS

## Aim

- Reduce dependency on external DNS for cluster functioning
- Tune & optimise DNS resolution as per requirement.
- Make DNS more observable and easier for troubleshooting.
- Limit domain querying and metadata exposed to external DNS



# Two Solutions Compared

## Dnsmasq service

- Lightweight DNS server
- Battle-tested (~20 years)
- Simple address record config
- Runs as service on each node

**BEST FOR:**
Medium-scale clusters, simple requirements, fast setup

&

## coreDNS static pods

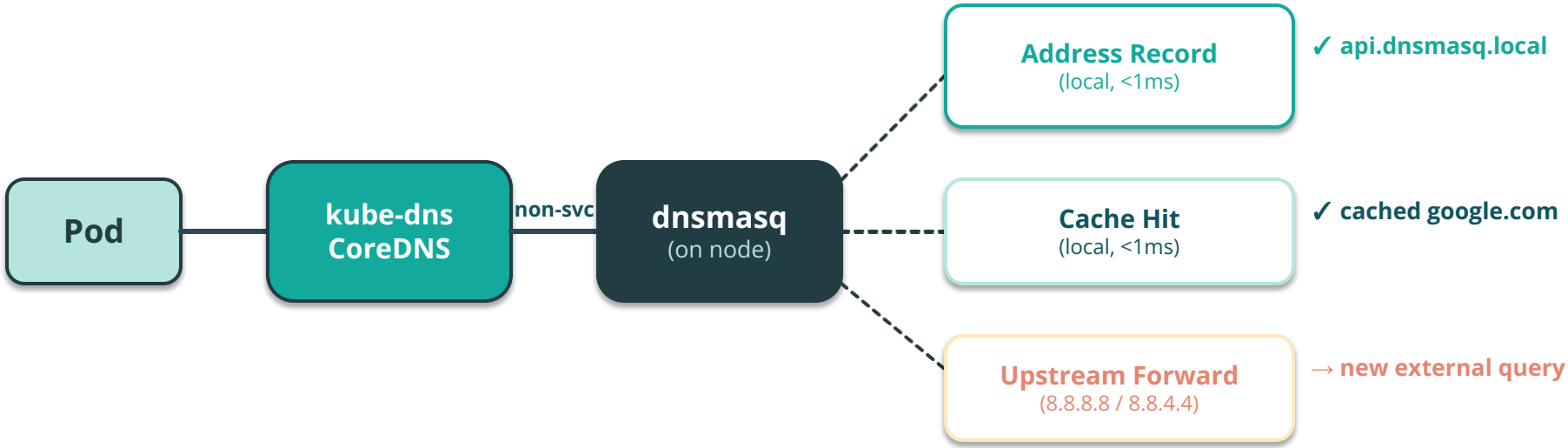
- CNCF graduated project
- Plugin-based architecture
- Per-zone Prometheus metrics
- Kubelet-managed static pods

**BEST FOR:**
Large/complex clusters, deep observability, CNCF ecosystem

# Dnsmasq Integration into k8s

dnsmasq-kubernetes

# dnsmasq Architecture




**Two-layer DNS: kube-dns for services, dnsmasq for everything else.**

*.svc.cluster.local never reaches dnsmasq — kube-dns handles it first

# dnsmasq: What You Get

### Deterministic Caching
Configurable cache size (1000 entries), per-node isolation

### Local Address Records
API server, ingress wildcard — resolved in <1ms, never leave node

### Query Logging
Every query logged: source, type, cached/forwarded/local

### Per-Upstream Forwarding
Forward to 8.8.8.8 + 8.8.4.4, track per-upstream metrics

### Failover Resilience
Upstream DNS down? Cluster-critical domains still resolve.

# dnsmasq Configuration

```
# /etc/dnsmasq.conf
resolv-file=/etc/resolv.conf.upstream
cache-size=1000
dns-forward-max=10000
bind-interfaces
listen-address=0.0.0.0

# Local domain resolution
address=/api.dnsmasq.local/172.18.0.3
address=/api-int.dnsmasq.local/172.18.0.3
address=/.apps.dnsmasq.local/172.18.0.5

log-queries
log-facility=/var/log/dnsmasq.log
```

**Upstream Persistence**
Original cloud DNS preserved via upstream config.

**Efficient Caching**
1000 entries cached per node for high performance.

**Local Resolution**
Critical domains resolved locally; never leave node.

**Full Observability**
Complete query logging for visibility and debugging.

# Live Demo: dnsmasq

## SCALABLE

- Deterministic Caching
- Decoupled Upstream Path
- Predictable Query Volume

## CONSISTENT

- Uniform resolv.conf Structure
- Centralized Configuration Management
- Standardized Observability

*Watch DNS become scalable & consistent*

# CoreDNS Static Pod Integration as a Custom DNS

custom-dns-kubernetes

# CoreDNS Static Pods: What It Is

## Architecture & Management

- **CNCF Graduated:** Same project as kube-dns
- **Kubelet-Managed:** Self-managed static pods
- **Decoupled:** One pod per node — no Deployment or scheduler required

## Unmatched Resiliency

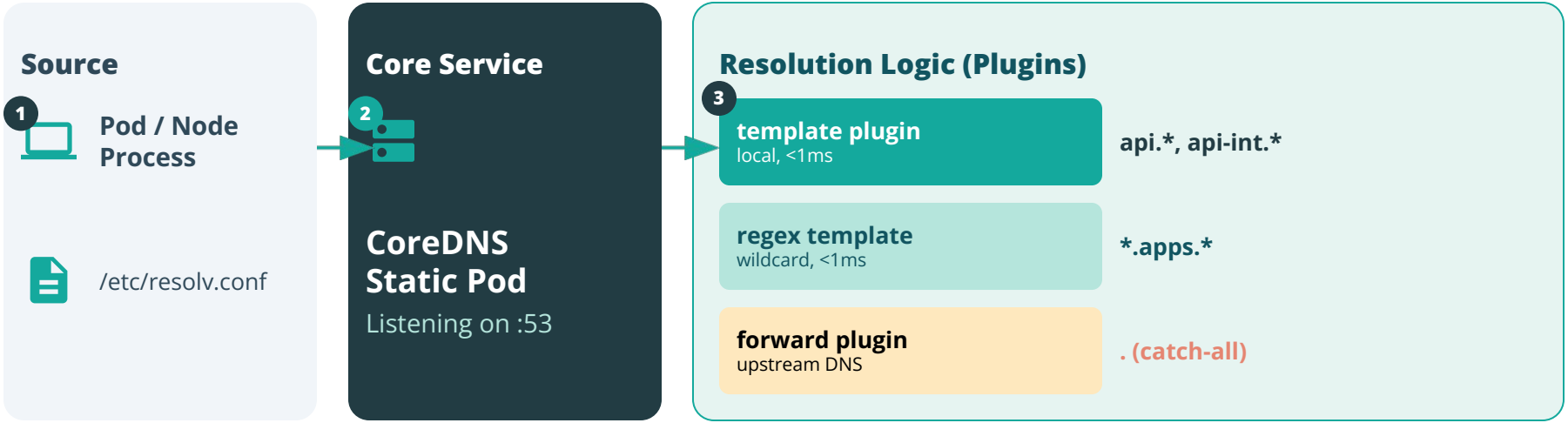
- **Auto-Healing:** kubelet restarts it immediately if it crashes
- **API Independence:** Works even if API server is unreachable
- **Critical Priority:** DNS must work before any other service can

## Extensible Plugin Architecture

Compose exactly the behavior you need in Corefile: **template, forward, cache, prometheus**

Static pods represent the most resilient deployment model in Kubernetes

# CoreDNS Static Pod Architecture



# CoreDNS: What You Get

### Per-Zone Metrics
Separate request/latency counters for each domain zone

### Authoritative Answers
aa flag set — node is the authority, not relaying

### Sub-ms Resolution
Query time: <1ms for cluster domains (no network)

### Regex Wildcards
Any *.apps.domain resolves via pattern matching

### Failover Resilience
Same upstream-independent resolution as dnsmasq

# The Corefile

```
api.custom-dns.local {
prometheus
template IN A api.custom-dns.local {
answer "{{ .Name }} 30 IN A <cp-ip>"
}
}

apps.custom-dns.local {
prometheus
template IN A apps.custom-dns.local {
match .*\.apps\.custom-dns\.local
answer "{{ .Name }} 30 IN A <worker-ip>"
}
}

. { forward . <upstream> ; cache 30 ; prometheus :9153
}
```

**Control Plane Resolution**
- Per-zone Prometheus metrics
- Authoritative A record via template
- Returns the control-plane IP

**Application Wildcards**
- Regex wildcard matching
- Any *.apps.* resolves to worker-ip

**Global Catch-all**
- Forward + Cache + Metrics

**Pro Fact**

This Corefile allows sub-millisecond local resolution for internal domains while preserving native observability.

# CoreDNS: Native Observability

**Built-in Prometheus endpoint at :9153/metrics — no custom exporter needed**

| Metric | Labels | What It Shows |
|---|---|---|
| coredns_dns_requests_total | zone, type | Per-zone request counts |
| coredns_dns_responses_total | rcode | NOERROR / NXDOMAIN / SERVFAIL |
| coredns_dns_request_duration_seconds | zone | Latency histogram (p50/p95/p99) |
| coredns_cache_hits/misses_total | — | Cache effectiveness |

**DNS observability is native. Same Prometheus you already run. Zero custom code.**

# CoreDNS: Alerts & SLOs

| Alert | Severity | Condition | For |
|---|---|---|---|
| CoreDNSDown | critical | Instance unreachable | 1m |
| CoreDNSHighSERVFAILRate | warning | SERVFAIL > 1% | 5m |
| CoreDNSHighNXDOMAINRate | warning | NXDOMAIN > 10% | 10m |
| CoreDNSLatencyP99High | warning | p99 > 100ms | 5m |
| CoreDNSCacheHitRateLow | info | Cache hit rate < 50% | 10m |

**SLOs: Availability ≥ 99.9% (~43min downtime/month) · Latency p99 ≤ 100ms (99% of requests)**

# Live Demo: CoreDNS Static Pods

***Per-zone metrics, regex wildcard, and more***

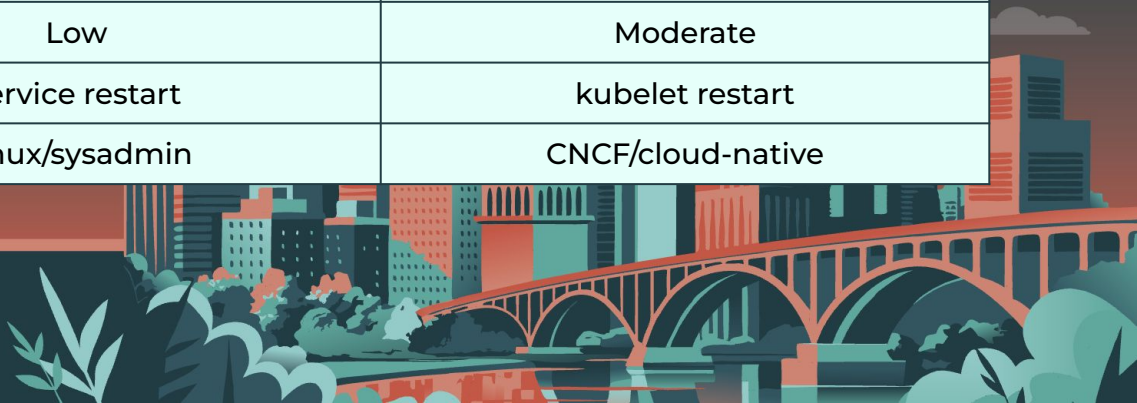
THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA
Embedded Linux Conference

# Choosing Your Approach

***There's no wrong answer — only trade-offs***

# Side-by-Side Comparison

| Aspect | dnsmasq | CoreDNS Static Pods |
|---|---|---|
| Deployment | Service on node | Static pod (kubelet) |
| Configuration | /etc/dnsmasq.conf | Corefile (server blocks) |
| Metrics | Custom exporter | Native Prometheus |
| Cache | Built-in, configurable | Plugin, 30s TTL |
| Wildcards | Simple prefix (*.apps.) | Regex matching |
| Zone Metrics | No | Yes (per-zone) |
| Latency Histograms | No | Yes (p50/p95/p99) |
| Complexity | Low | Moderate |
| Resilience | service restart | kubelet restart |
| Ecosystem | Linux/sysadmin | CNCF/cloud-native |



# Choosing Your Approach

## Rethinking the Engineering Trade-offs

**dnsmasq**
(Operational Simplicity)

- <50 Nodes
- Aggregate QPS
- Simple wildcards

**CoreDNS Static Pods**
(Native Observability)

- >50 Nodes
- Per-Zone Metrics
- Regex wildcards
- SLO tracking

Analyze scale, required visibility, and engineering requirements
to find the fit for your operational constraints.

# Closing

*DNS doesn't have to be a mystery*

# Key Takeaways

### 01. Cloud Dependency
Every K8s cluster has an invisible dependency on cloud DNS. If that DNS goes down, the cluster goes down.

### 02. Local Resolution
Self-hosted node-local DNS eliminates that dependency. Cluster-critical domains resolve locally. Always.

### 03. Flexible Tooling
Proven approaches: dnsmasq (simple) or CoreDNS (deep). Choose based on scale and observability needs.

### 04. Full Observability
DNS becomes observable via per-node metrics and SLO tracking. No more black box.

**05. Additive Strategy:** It's not a replacement. Kubernetes service DNS continues to work exactly as before.

**Stop hoping DNS works. Start knowing it does.**

# The DNS Layer Cake

**Layer 3 — Upstream DNS (external)**

google.com, container registries, webhooks → forwarded to 8.8.8.8 / cloud DNS

**Layer 2 — Custom DNS (infrastructure)** ⭐ **NEW**

api.cluster.local, *.apps.cluster.local → resolved locally by dnsmasq or CoreDNS

**Never leaves the node. Survives upstream failure.**

**Layer 1 — kube-dns (services)** **UNCHANGED**

*.svc.cluster.local, *.pod.cluster.local → standard Kubernetes service discovery

**Additive, not a replacement. Each layer handles its own domains.**

# QnA