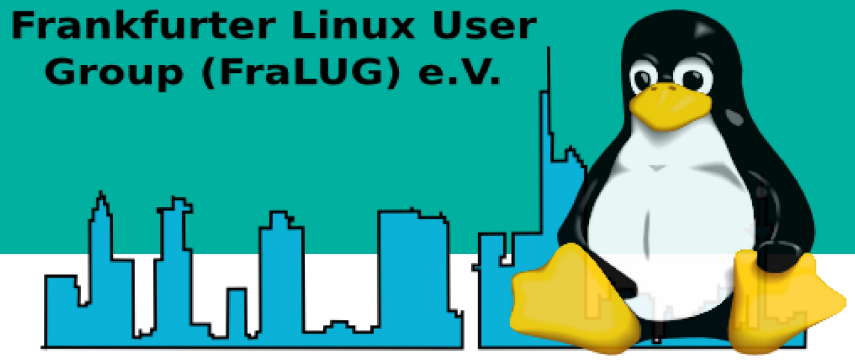
K8s - Service Meshes

Istio und Network Service Mesh



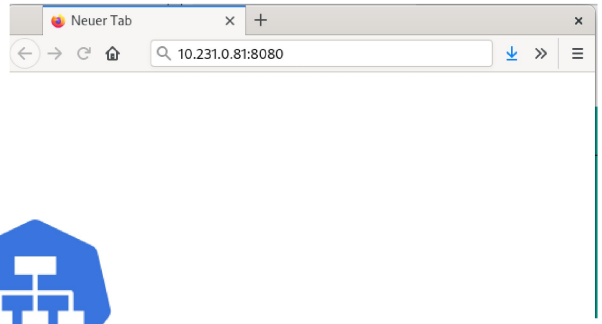
Kubernetes - Networking

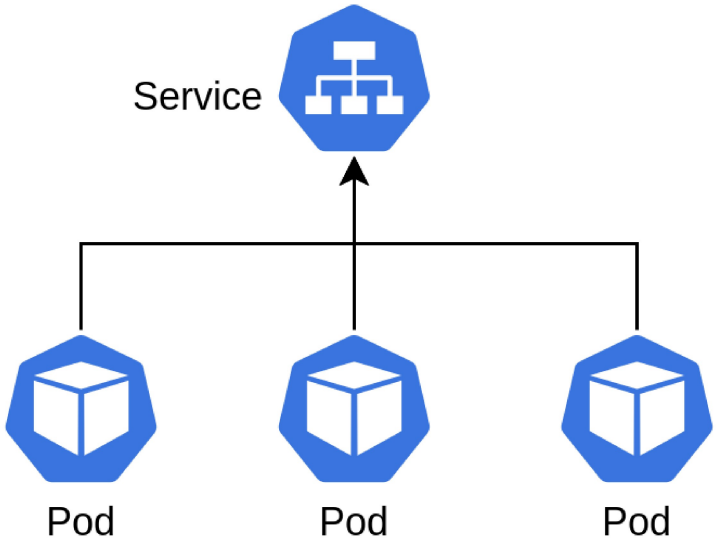
Networking-Herausforderungen von Microservice-Architekturen

- Application Exposure
- Microservices müssen untereinander kommunizieren
- Kommunikation muss sicher sein
- Metrics
- L3 intradomain Connectivity
- L3 interdomain Connectivity

Basic Application Exposure

- Deployments/Stateful Sets sind nur lokal erreichbar
- Services machen Deployments/Stateful Sets via IP/Port verfügbar

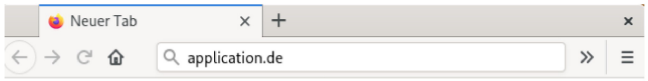


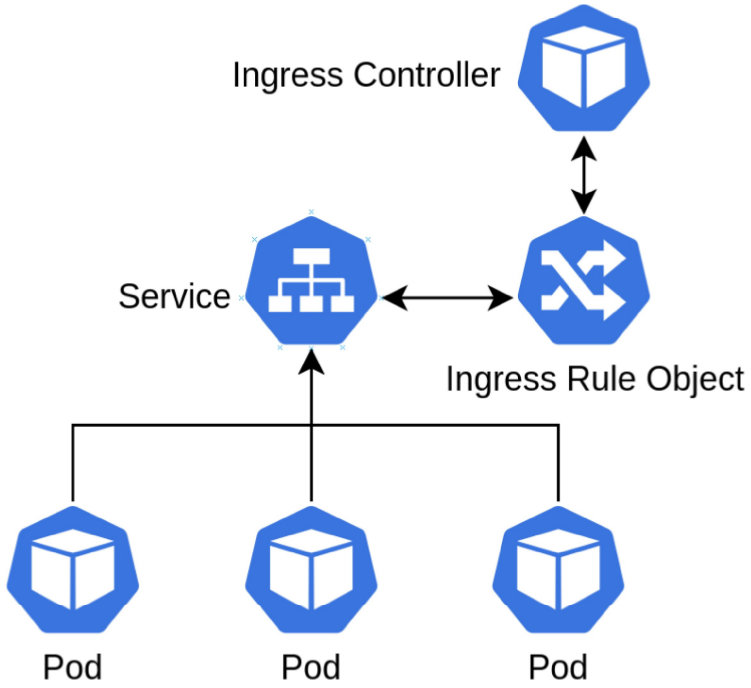


Basic Application Exposure

INGRESS

- INGRESS mappt Hostnames und URLs auf Services

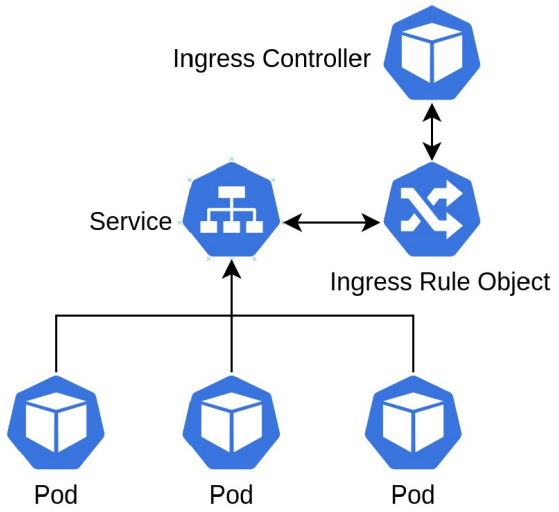




Basic Application Exposure

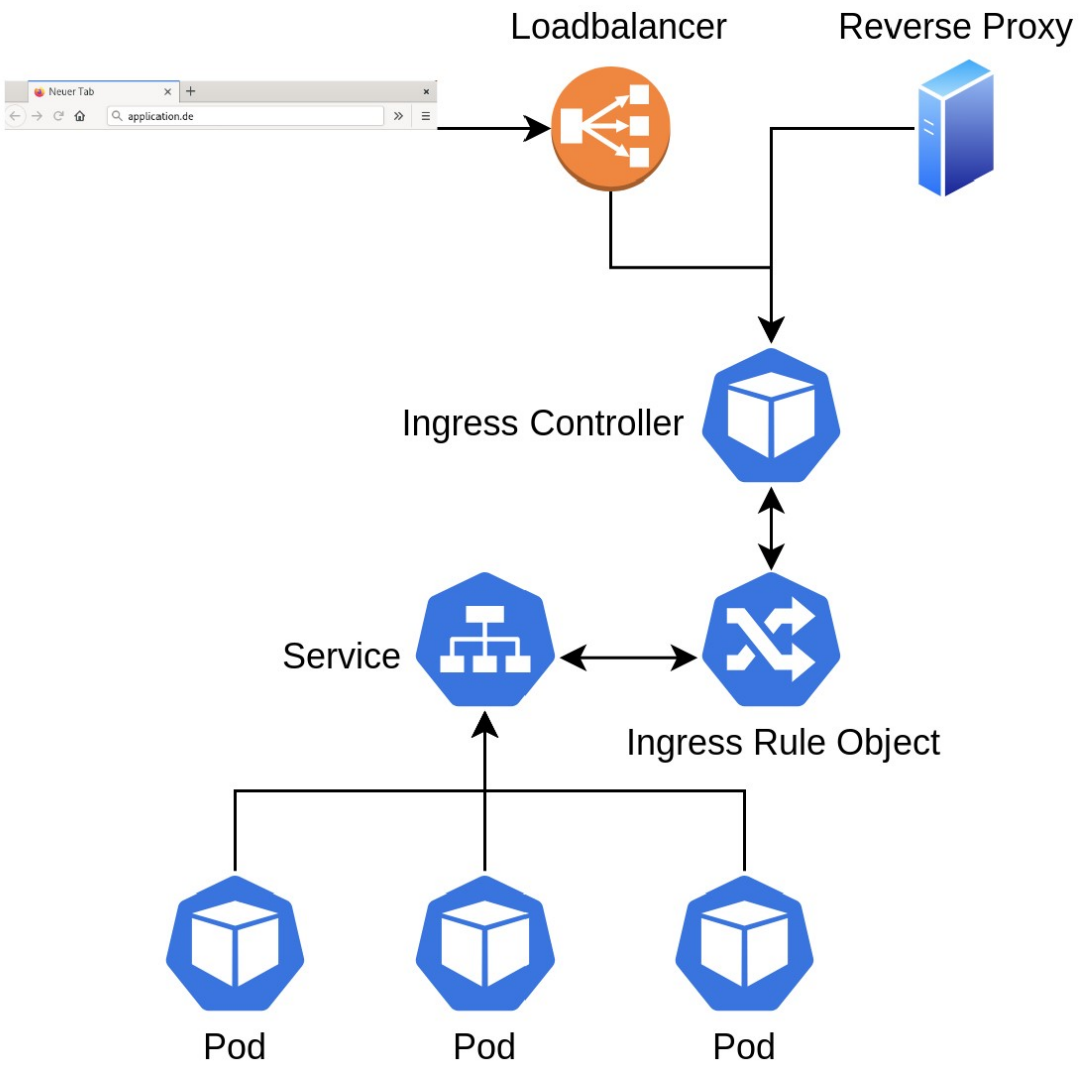
INGRESS

- ```yaml
  apiVersion: networking.k8s.io/v1
  kind: Ingress
  metadata:
    name: application-ingress
  spec:
    rules:
    - host: application.de
      http:
        paths:
          backend:
              serviceName: nginx-service
              servicePort: 8080
  ```


  ```

Basic Application Exposure

INGRESS

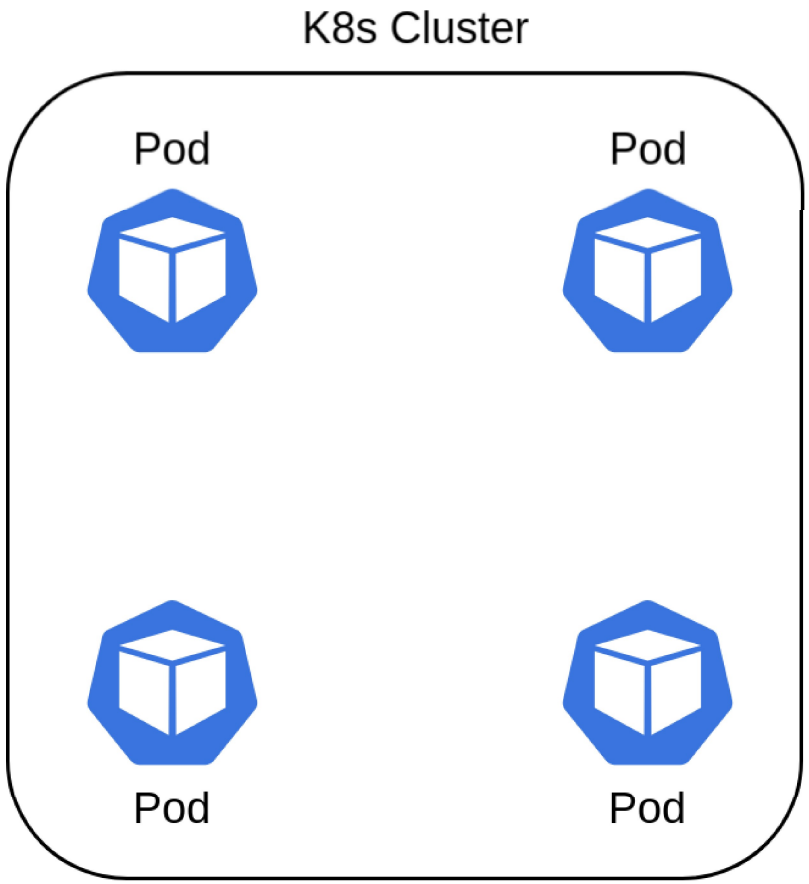


Istio

L4/L7 Application Service Mesh

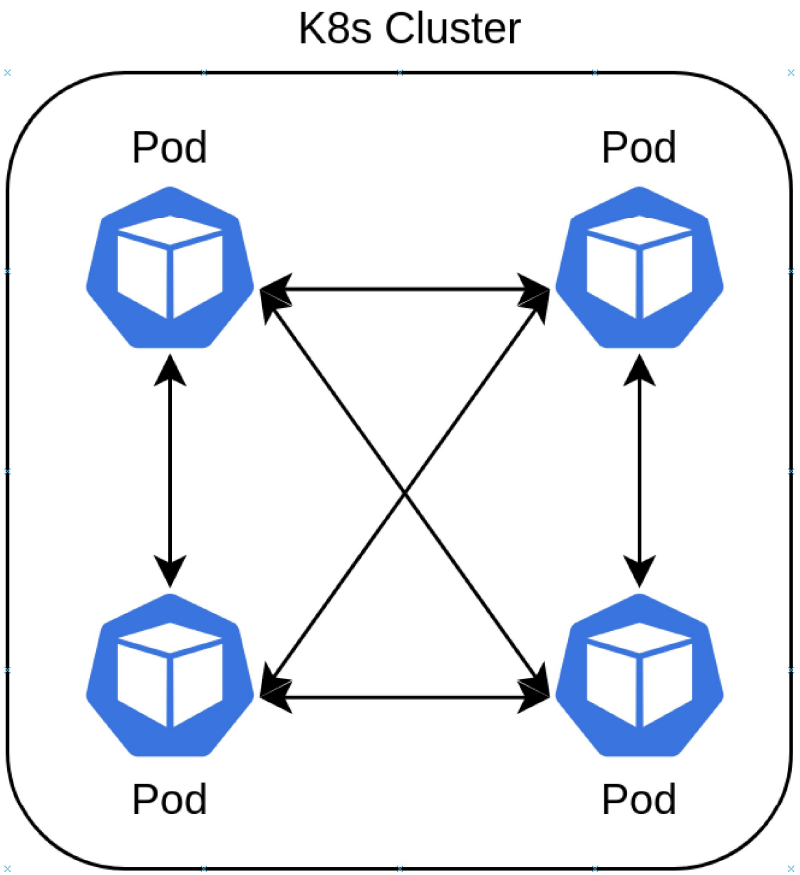
Microservice-Challenges

- Keine Traffic Restriktionen



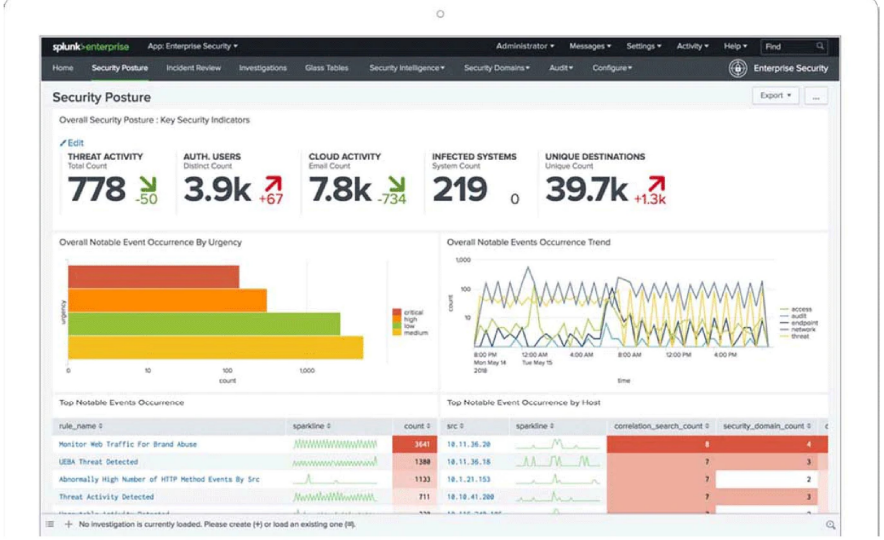
Microservice-Challenges

- Keine Traffic Restriktionen



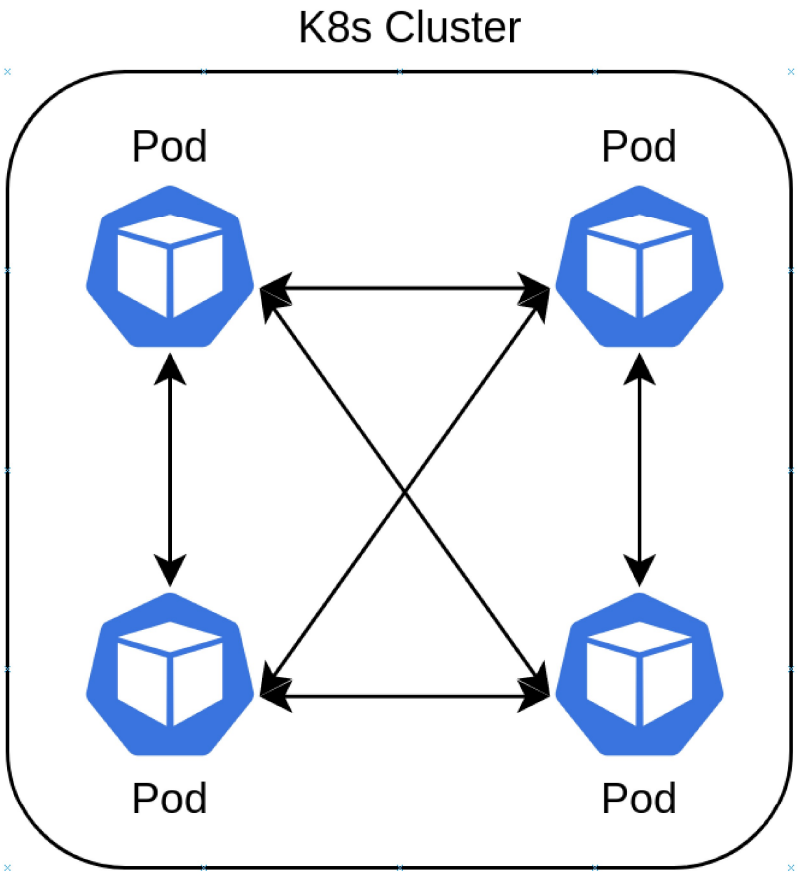
Microservice-Challenges

- Keine Metriken



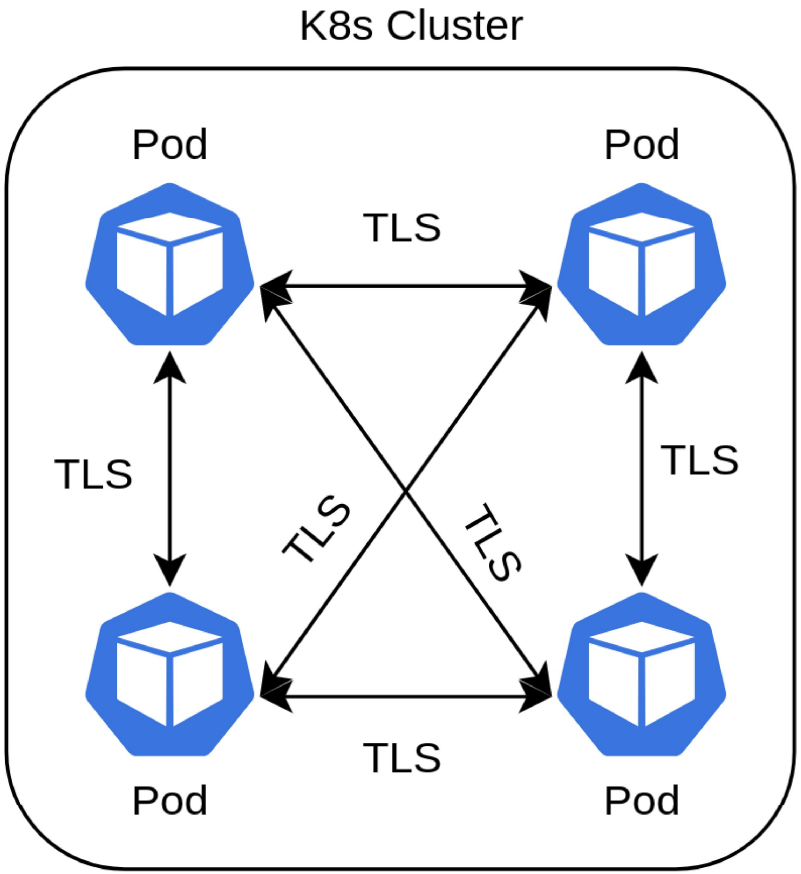
Microservice-Challenges

- Traffic unverschlüsselt



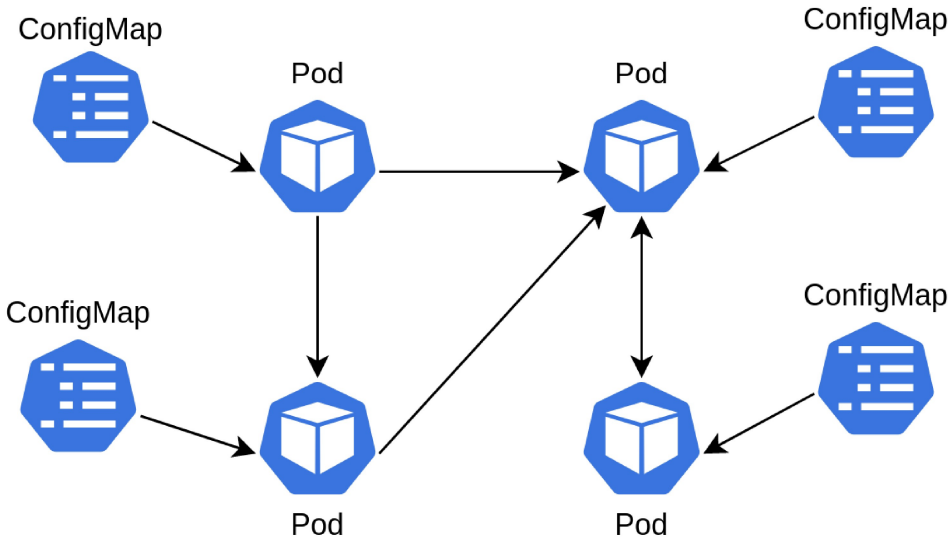
Microservice-Challenges

- Traffic unverschlüsselt



Microservice-Challenges

- komplizierte per pod endpoint Definition



- Service Endpoint Definition komplex
- Retry Logik/Resilienz
- Unsichere Kommunikation
  in einem Standard Kubernetes Cluster kann jeder Service mit jedem kommunizieren
  - zusätzliches Policing erforderlich (FW/Identy)
  - Anzahl der Services erhöht die Komplexität
- Traffic ist in der Regel unverschlüsselt
  - komplexe Encryption Setups auf per-Service/Pod Basis
- Traffic sollte zwischen workloads vermittelt werden
- Sammeln von Metriken kompliziert
- Traffic Splitting

- Multivendor Application Service Mesh
  Unterstützung für Kubernetes, Cisco CCP,
  VMWare NSX (und mehr)

- Istio erweitert die Kubernetes API
  - benutzt YAML Files
    deklarativer Approach ohne Kenntnis von
    spezifischen DSLs
  - integiert in kubectl als CRD
    (custom resource definition)
  - agiert als Helfer

- ControlPlane - Istiod
  - Traffic Management API
  - Certificate Authority
  - Service Discovery
  - speichert Performance Metriken

- Envoy Proxy
  - Loadbalancing
  - SSL/TLS Termination
  - Service Discovery
  - Resilience/health checking
  - traffic splitting(staged Rollouts)
  - MetrikCollector
  - Support für TCP/http/gRPC/Websockets
  - Service Routing

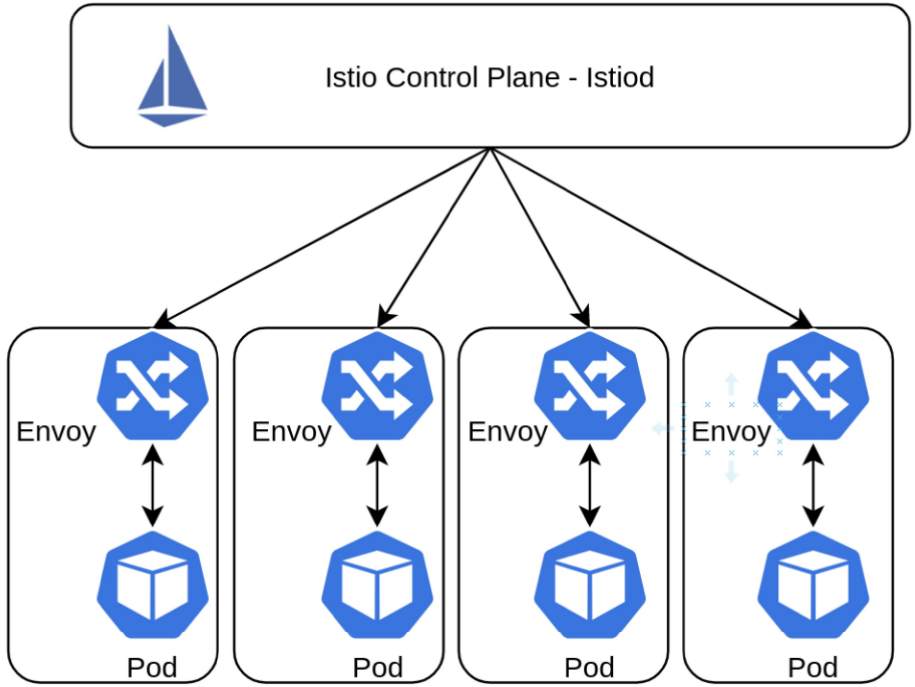
- Istio Ingress/Egress Gateway
  - verwaltet ein-/ausgehenden Traffic zum/vom Service Mesh (edge device)
  - Ingress Loadbalancing
  - CRSF Protection/CORS Support

Wie funktionierts?

- Istiod importiert die Kubernetes Service Registry
- Virtual Service Creation
  - definiert wie Traffic zu einem Service geroutet wird
- Destination Rule Creation
  - definiert was mit diesem Traffic geschieht
- Rules & Services werden zu den Envoy Proxies deployed

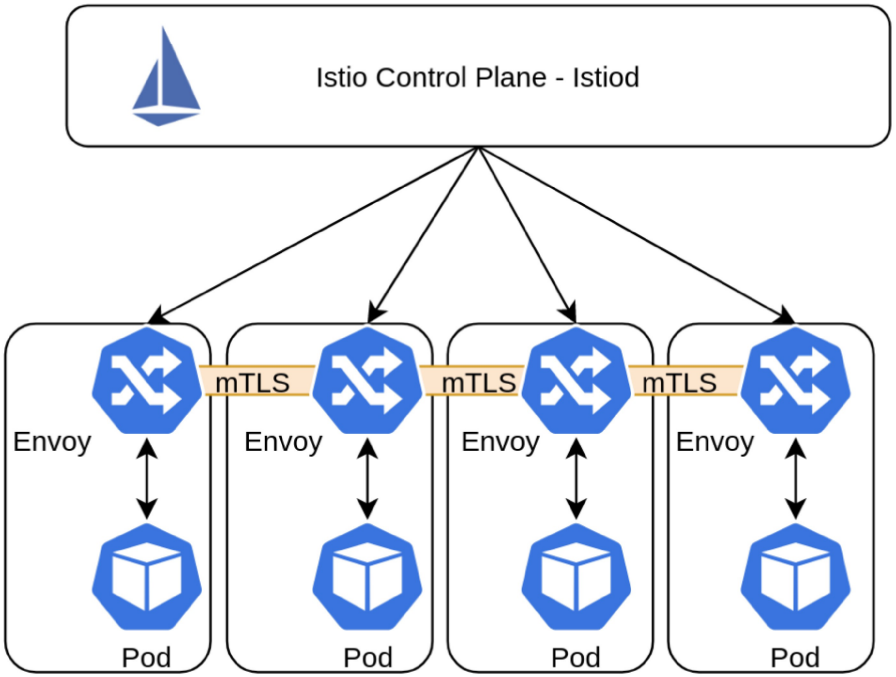
Istio Architecture

- Istio injiziert einen Envoy Proxy in jeden pod - diese envoy proxies bilden die Dataplane



Istio Architecture

- traffic zwischen den Services wird verschlüsselt über die Envoy Proxies geroutet



- Virtual Service Definition

```yaml
- apiVersion: networking.istio.io/v1
  kind: VirtualService
  metadata:
  name: vservice
  spec:
    hosts:
    - db.cluster.local
    tcp:
    - match:
      - port: 3300
      route:
      - destination:
          host: db.cluster.local
          port:
            number: 3309

- apiVersion: networking.istio.io/v1
  kind: VirtualService
  metadata:
    name: webservice
  spec:
  hosts:
  - web.cluster.local
  http:
  - name: webservice-route
    match:
    - uri:
        prefix: "/"
    route:
    - destination:
        host: web.cluster.local
```

- Destination Rule Definition

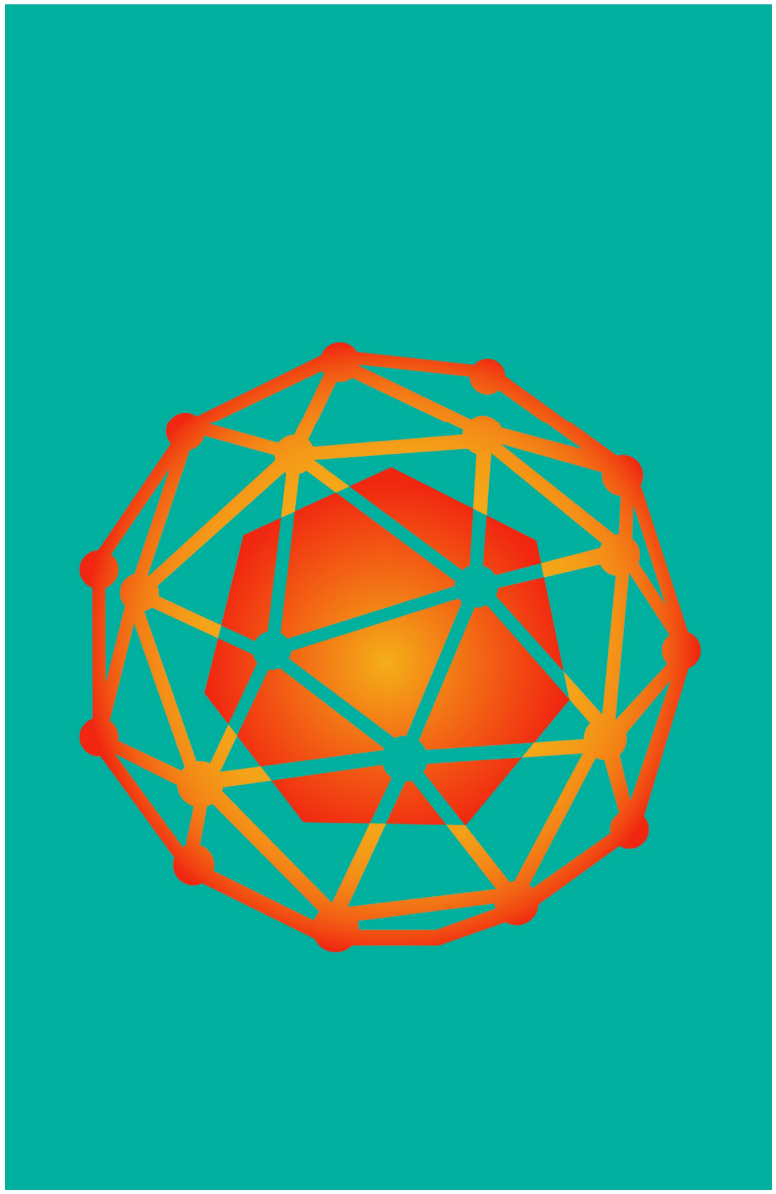
```yaml
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: webservice-destination-rule
spec:
  host: web.cluster.local
  trafficPolicy:
    loadBalancer:
      simple: LEAST_CONN
```

- Ingress Gateway Definition

```yaml
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: webgw
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

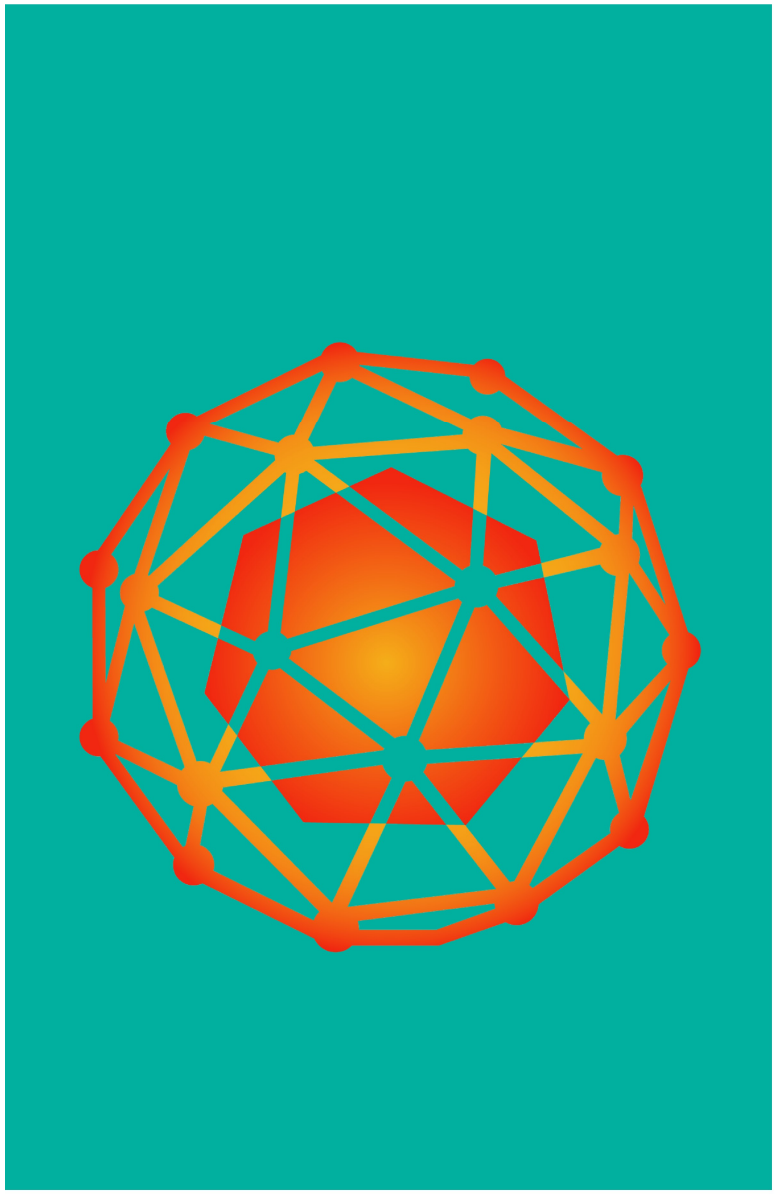
- Ingress Gateway Einbinden

```yaml
- apiVersion: networking.istio.io/v1
  kind: VirtualService
  metadata:
    name: webservice
  spec:
    hosts:
    - "*"
    gateways:
    - webgw
    http:
    - name: webservice-route
      match:
      - uri:
        prefix: "/"
      route:
      - destination:
          host: web.cluster.local
          port:
            number: 80
```



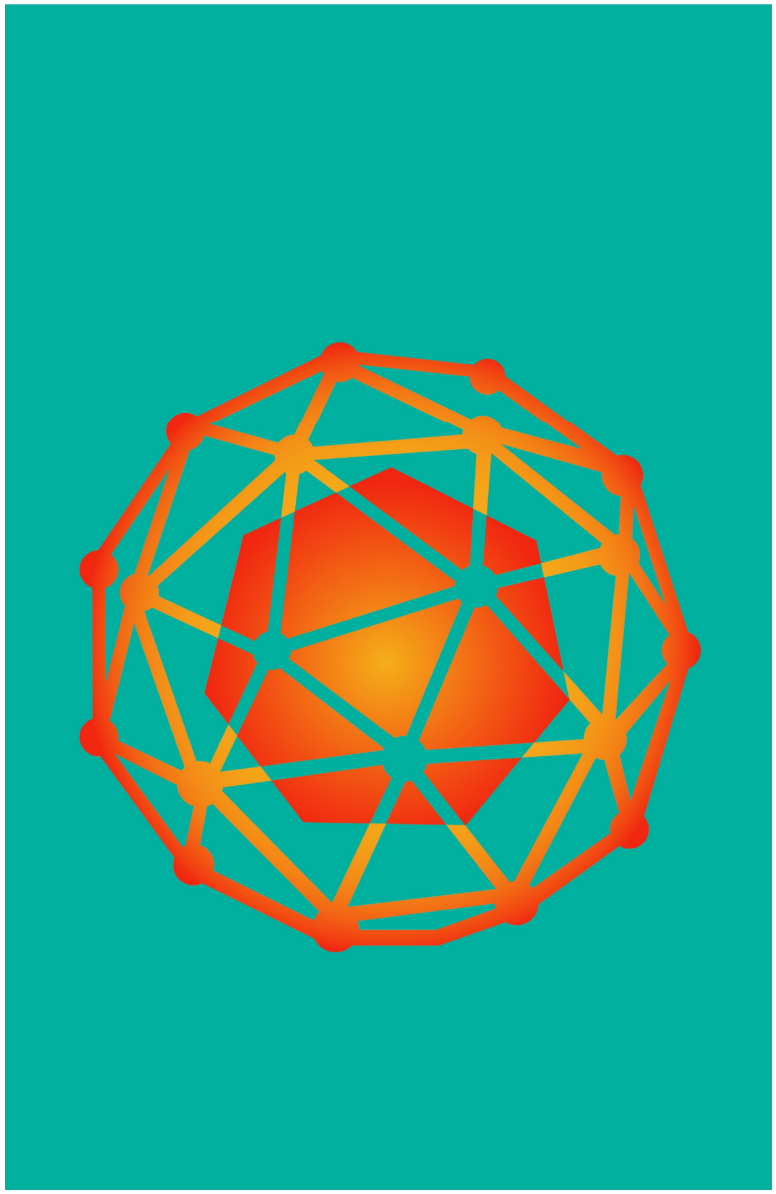
Network Service Mesh

L2/L3 interdomain connectivity



Network Service Mesh

- Konnektivität zwischen verschiedenen Workloads bei
  - onPremise Datacentern
  - verschiedenen Cloudanbietern
  - anderen Clustern
- ermöglicht L2/L3 Connectivity
- Cloud Native Approach
  - Entwickler statt Operator fokussiert
  - deklarative an Stelle von imperativer Konfiguration
- Einbindung via Custom Resource Definition (CRD)

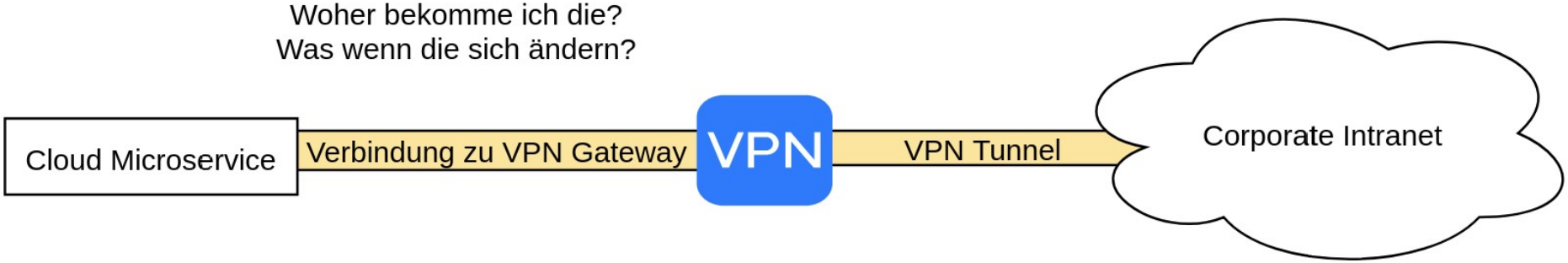


Ein Beispiel...

Sarahs Pod

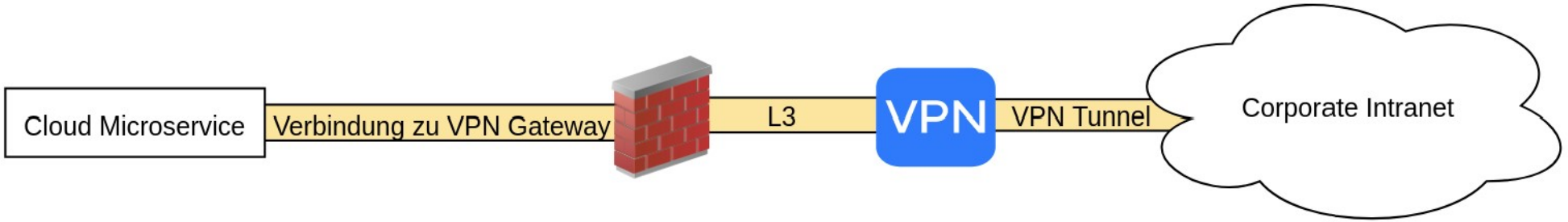
Definitionshölle für Entwickler

Welches Subnet brauch ich eigentlich?
Wie groß muss das sein?
Muss ich auch den VPN Pod konfigurieren?
Muss ich Routen für Corporate IPs in meinem Pod definieren?
Woher bekomme ich die?
Was wenn die sich ändern?



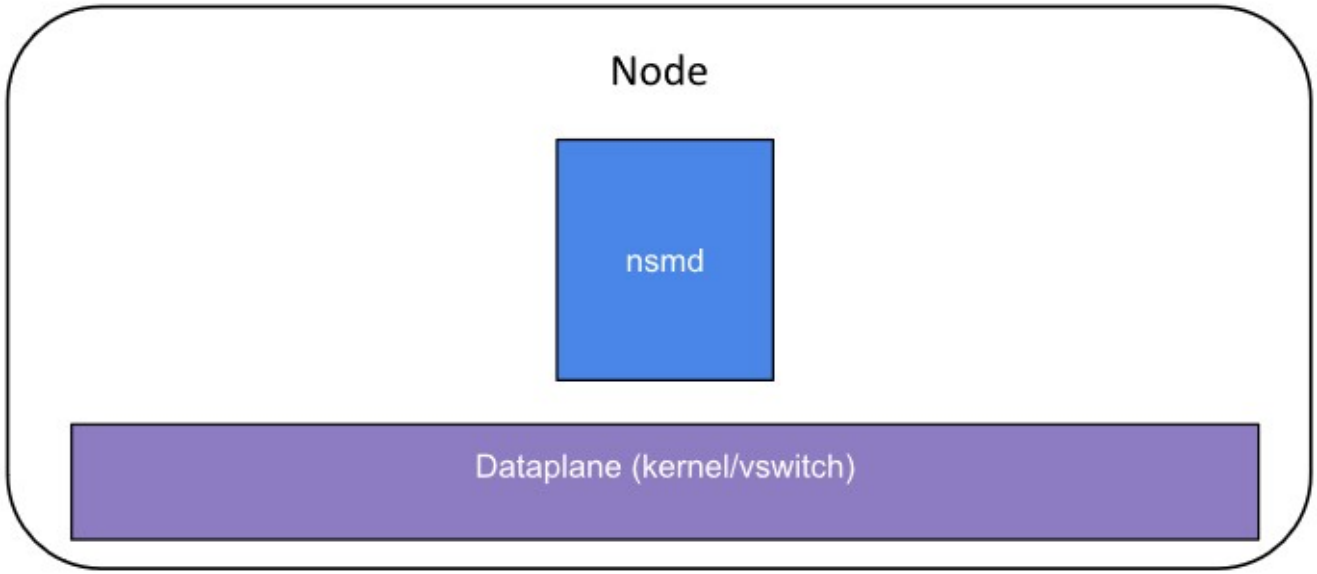
Definitionshölle für Entwickler die Zweite

Operations sagt ohne Firewall geht garnichts...
Muss ich die jetzt konfigurieren?
Was muss ich konfigurieren?
Warum beschäftige ich mich jetzt mit Netzwerk Setups?
Was wenn sich was ändert?



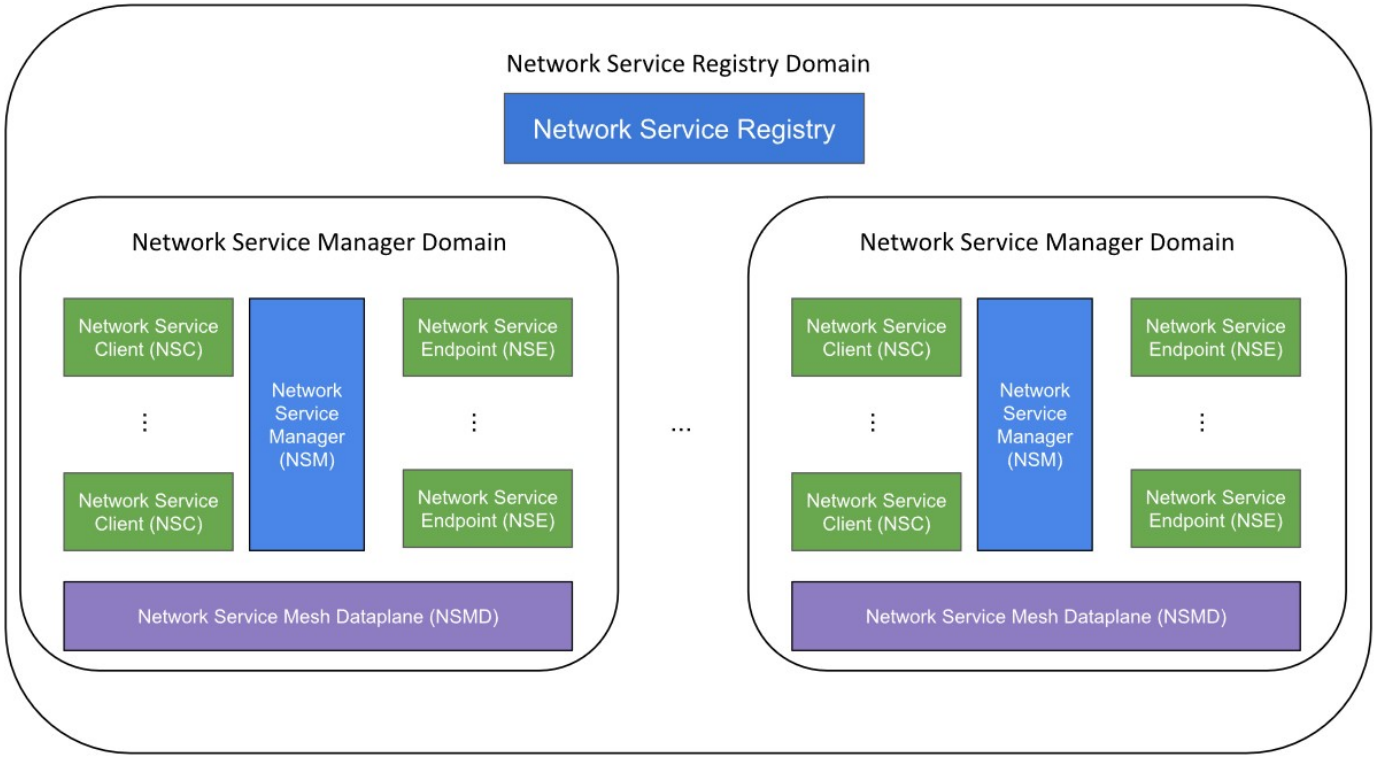
Architecture

- jeder Worker Node hat zusätzlich einen nsmd service und bildet eine Network Service Manager Domain

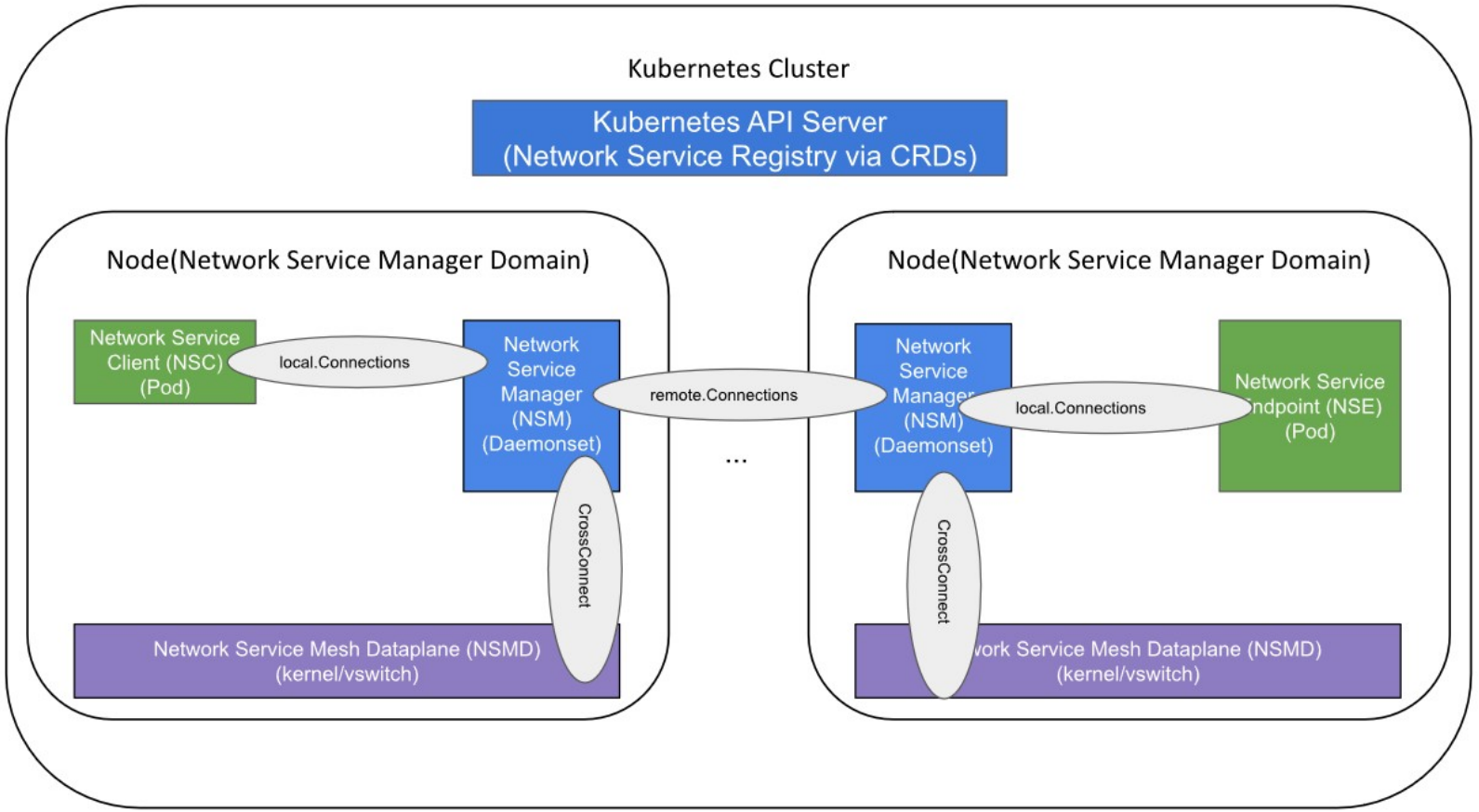


Architecture

- eine NSM Domain kann Clients und Endpoints enthalten
  - Clients stellen Applikationspods dar(Sarahs Pod zum Bsp.)
  - jedem Client Pod wird ein NSM initContainer hinzugefügt
  - Endpoints sind pods die externe Connectivity herstellen

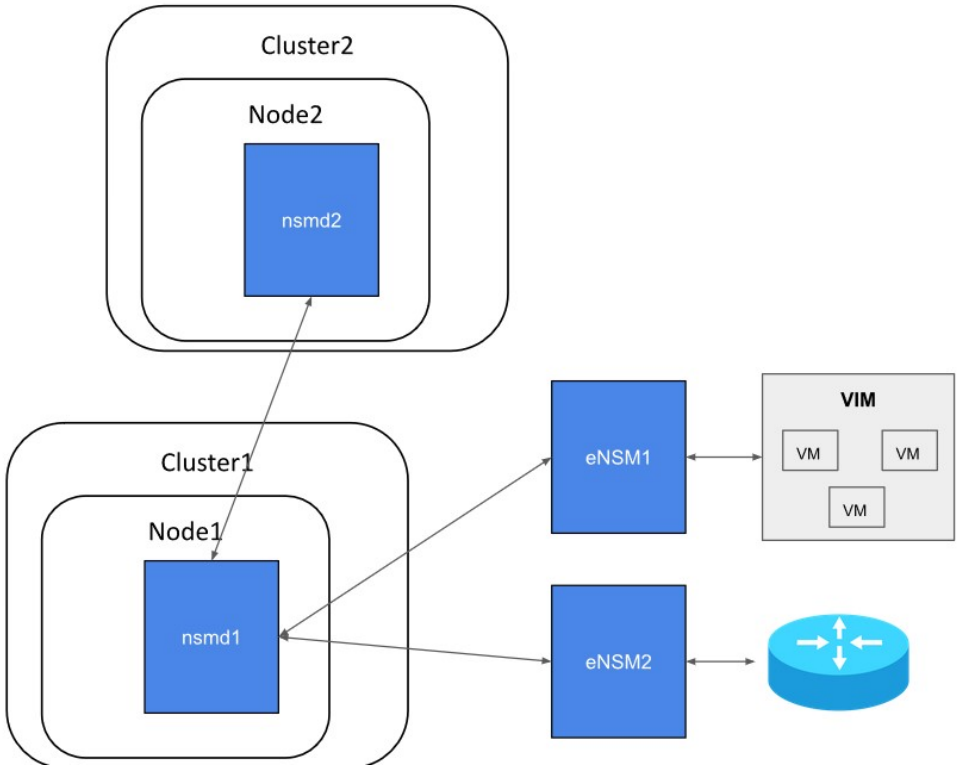


Architecture



Architecture

eNSMs bilden die Schnittstelle zu externen Devices



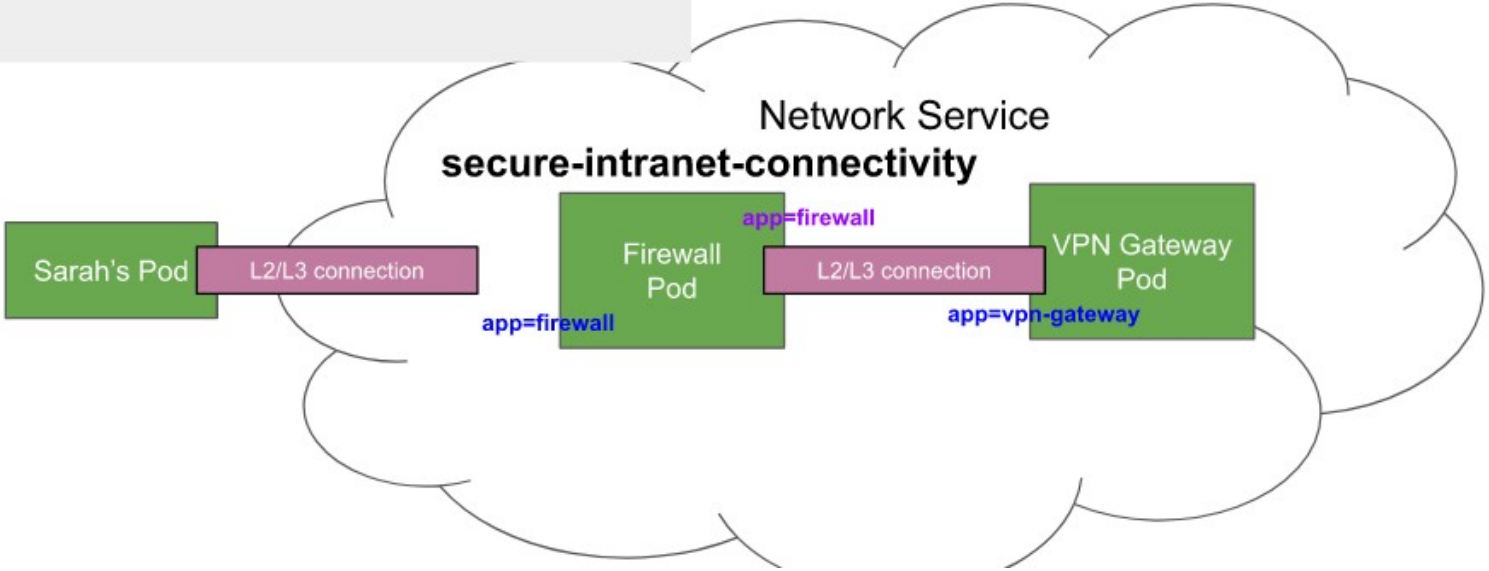
Architecture - eNSM Connectivity

- Externe NSMs werden anhand von DNS Records identifiziert
  - service records (SRV)
- Authentification/Authorization erfolgt via SPIFFE/SPIRE (https://spiffe.io/)

```
# _dienst._proto.name.    TTL   Klasse SRV Priorität Gewicht Port Ziel.
_sip._tcp.example.com.    86400 IN     SRV 10        0       5060 sip.example.com.
_nsm._tcp.example.com     86400 IN     SRV 10        0       8000 nsm.example.com.
```

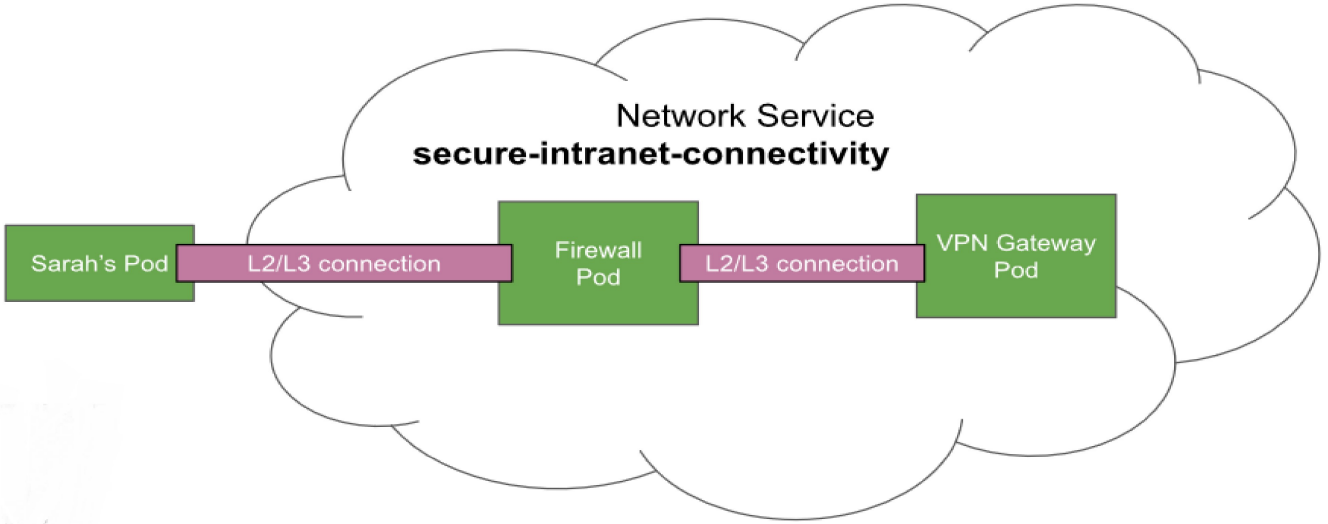
Konfiguration - der deklarative Approach
Source und Destination Selektoren

```
kind: NetworkService
apiVersion: V1
metadata:
       name: secure-intranet-connectivity
spec:
  payload: IP
  matches:
    - match:
      sourceSelector:
        app:firewall
      route:
        - destination:
          destinationSelector:
            App:vpn-gateway
    - match:
      route:
        - destination:
          destinationSelector:
            app:firewall
```



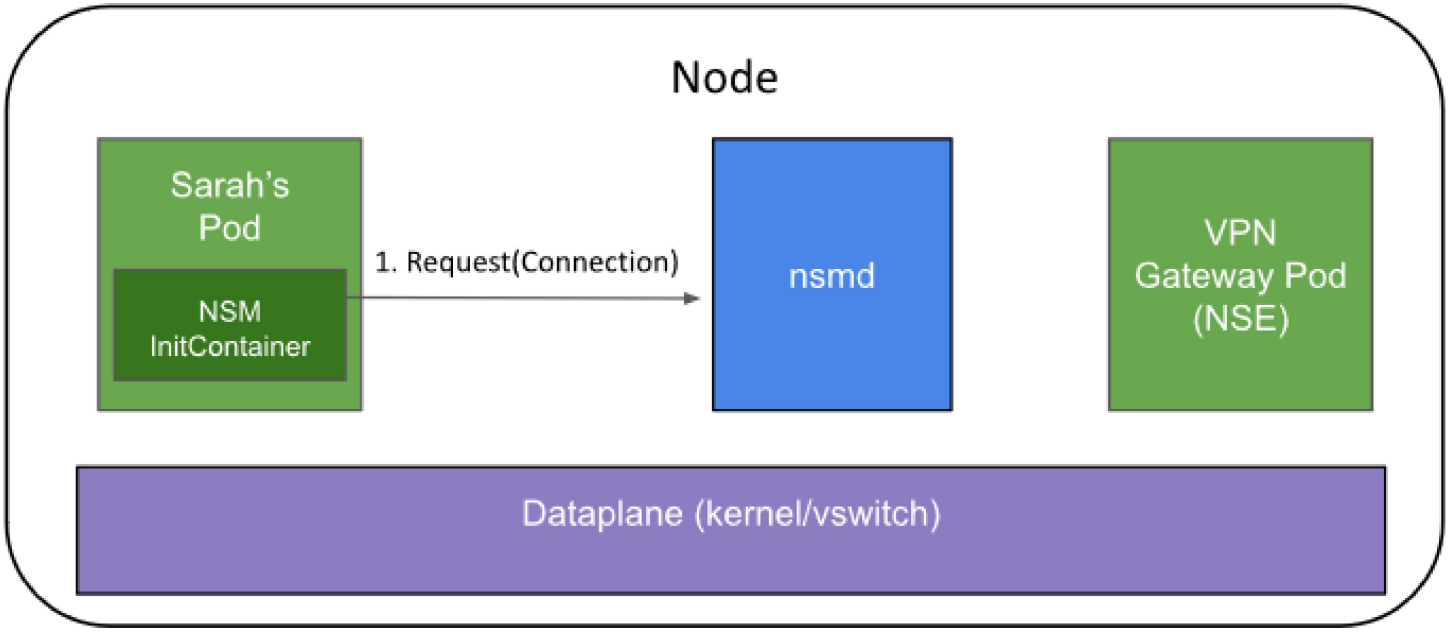
Serviceeinbindung - Annotationlabel

```yaml
apiVersion: v1
kind: Deployment
metadata:
  name: saras-app
  annotations:
    ns.networkservicemesh.io: secure-intranet-connectivity
spec:
  containers:
  - name: myapp
    image: myapp:1.0.0
......
```



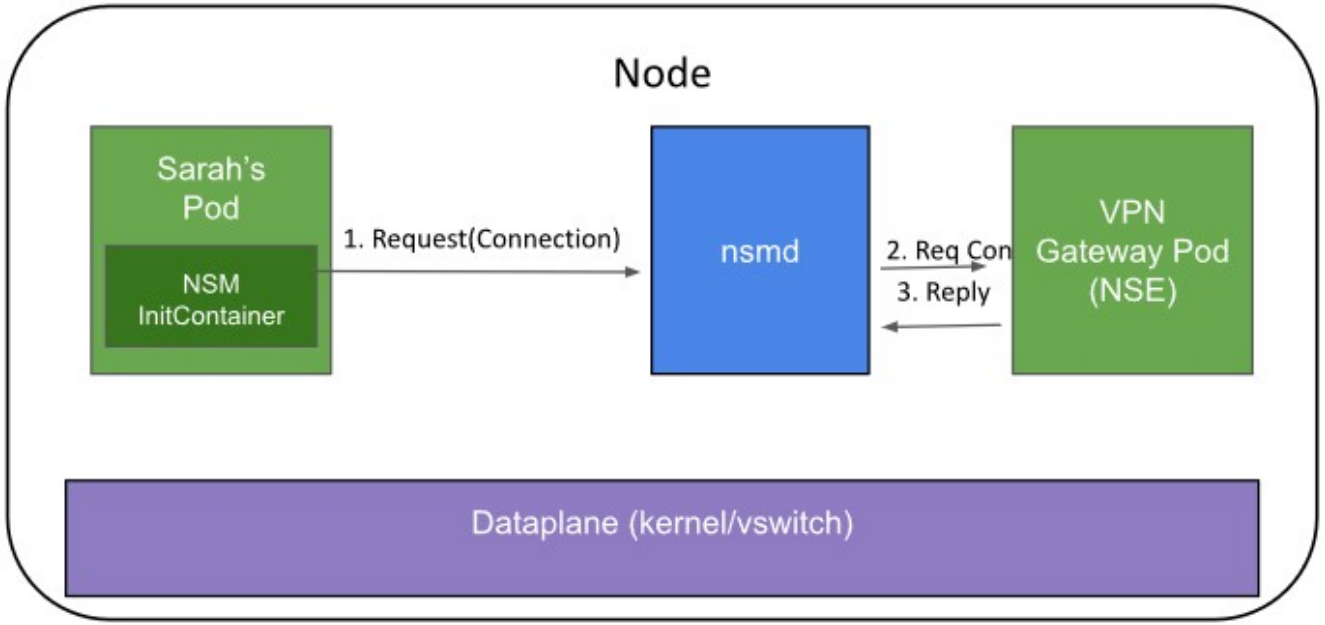
Verbindungsaufbau

- wenn der Application Pod startet wird via NSM InitContainer ein Request and den nsmd geschickt



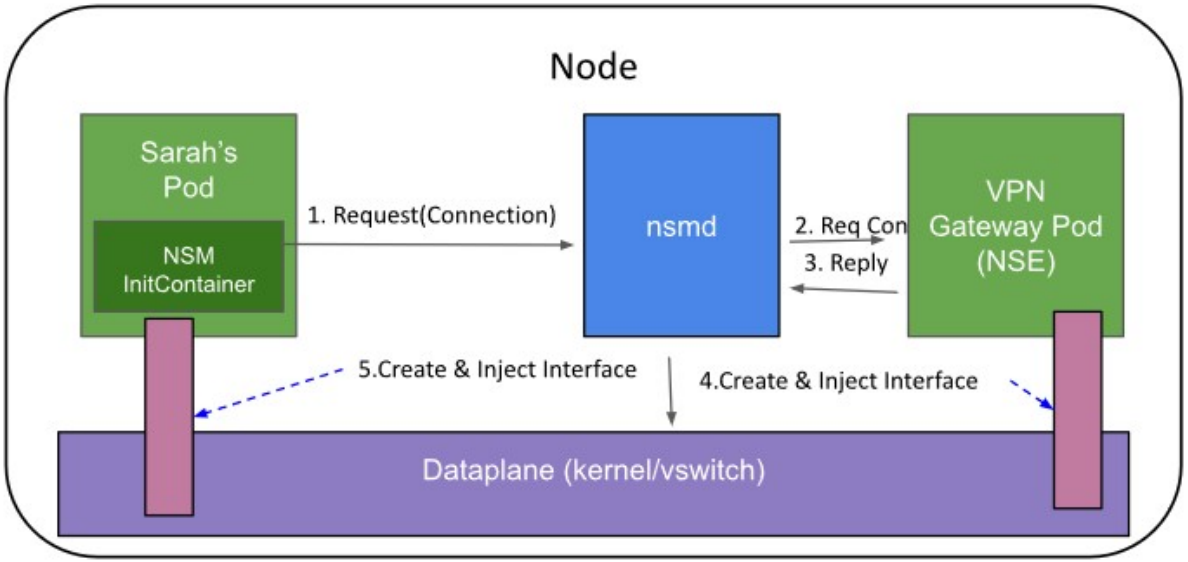
Verbindungsaufbau

- der nsmd kümmert sich um die Einbindung und Konfiguration des Endpoints



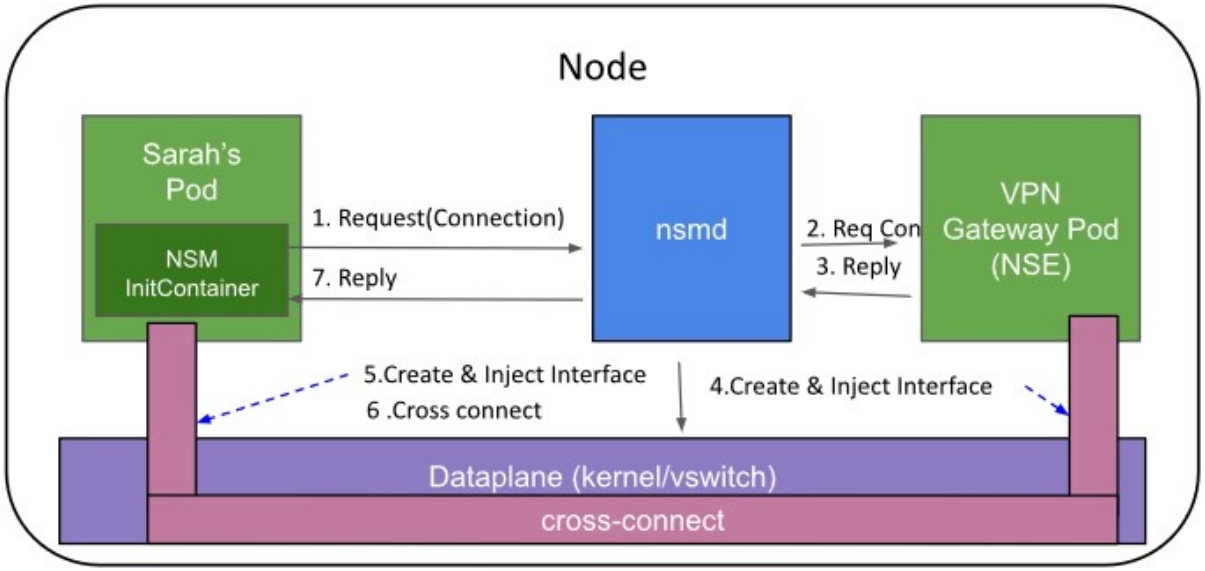
Verbindungsaufbau

- anschliessend wird die Konnektivität zwischen NSE und NSC hergestellt



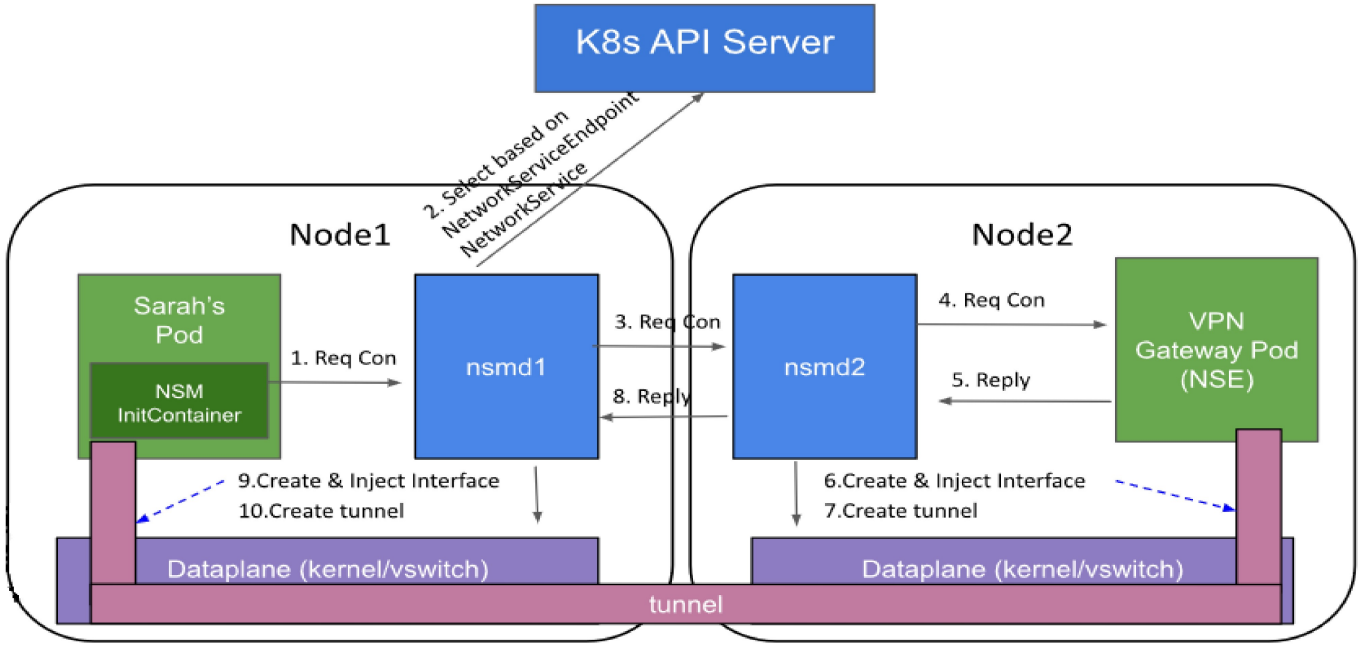
Verbindungsaufbau

- abschliessend wird der Client über den erfolgreichen Verbindungsaufbau informiert



Verbindungsaufbau

- NSC und NSE müssen sich nicht auf dem selben Client befinden



Ende