BlueTrusty

Filtrage des flux réseaux issus des clusters Kubernetes : pourquoi, comment ?

Stéphane REYTAN
sreytan@bluetrusty.com

BlueTrusty fournit des services de CyberSécurité.

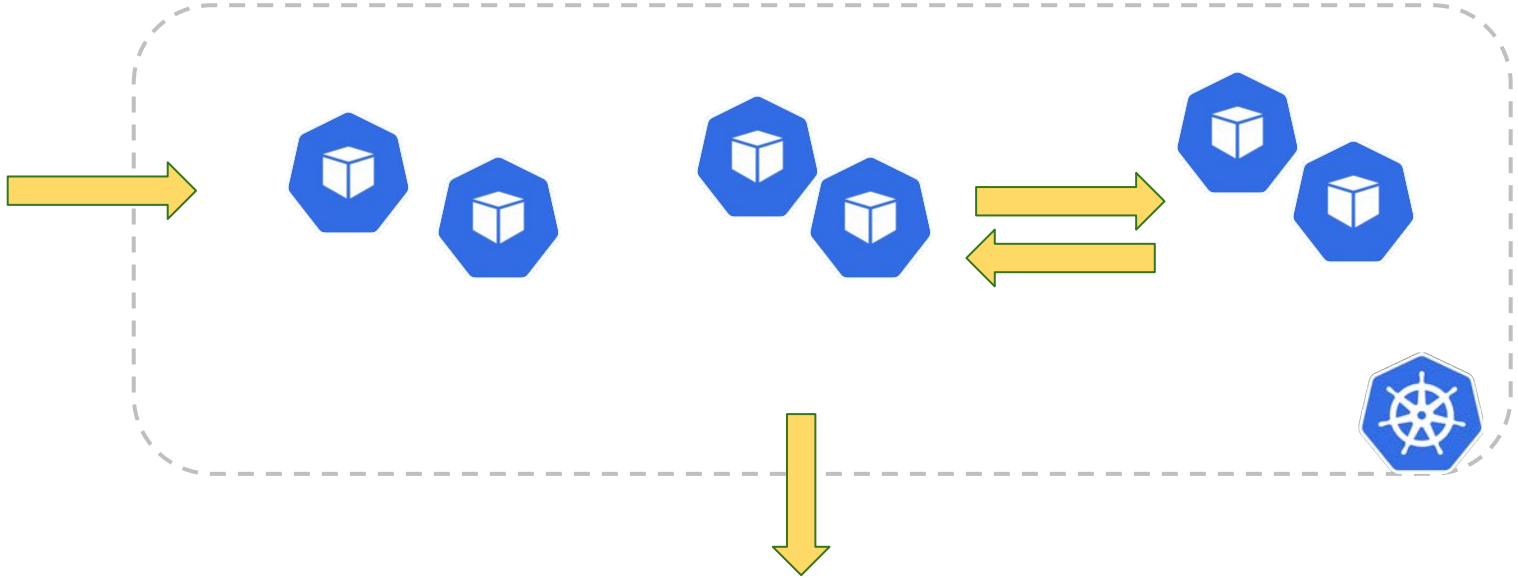
Nous fournissons des services de formation, d'audit et d'assistance sur Kubernetes et les architectures Cloud Native depuis 2017.

BlueTrusty est partenaire revendeur de Calico Cloud/Enterprise (éditeur Tigera).

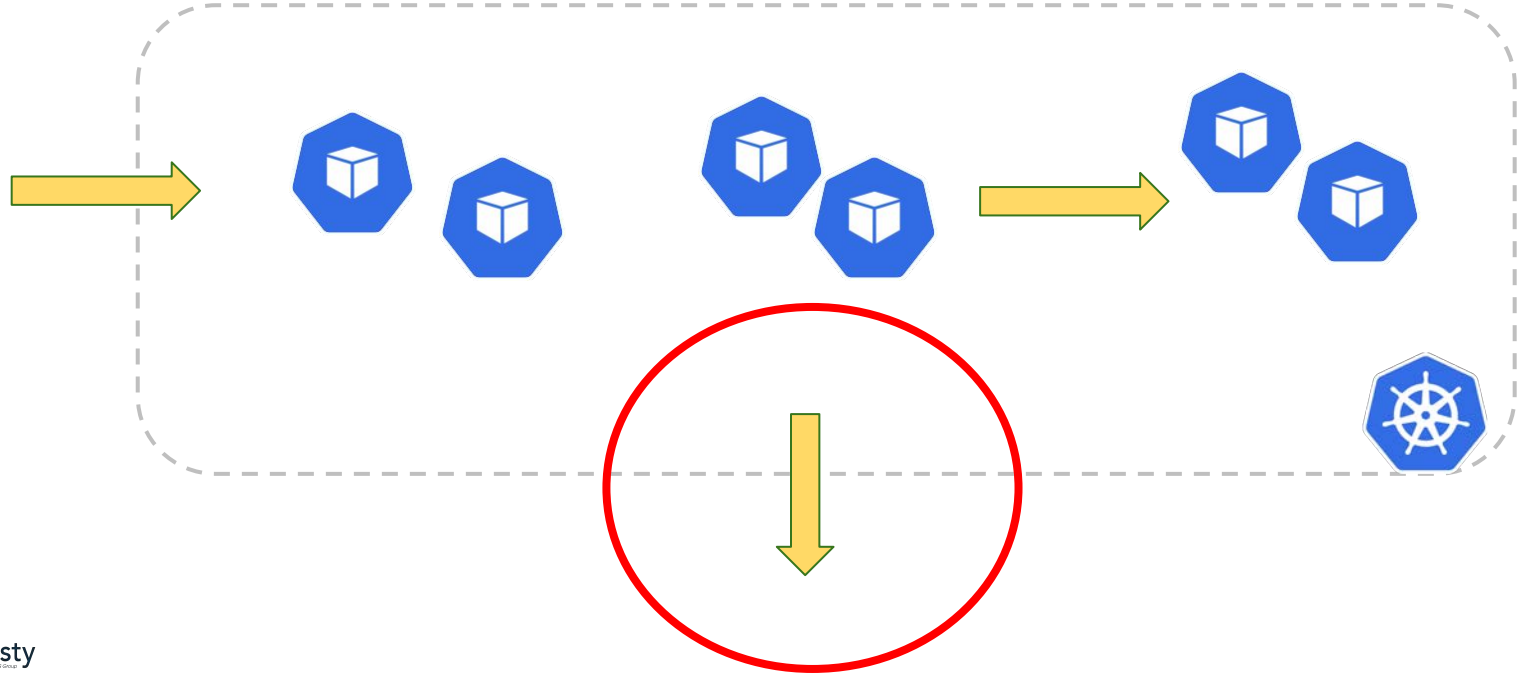
PROJECT CALICO

TIGERA

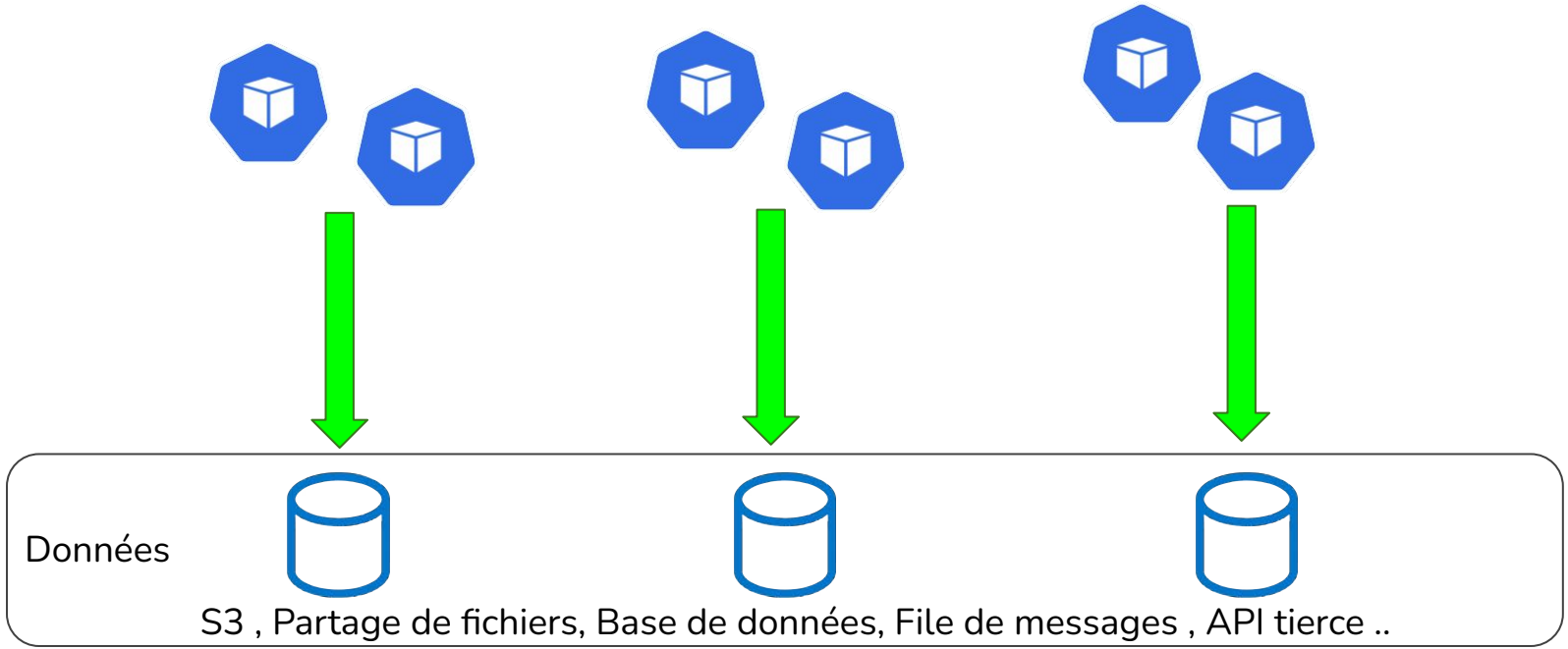
Il existe de nombreuses opportunités de filtrage des flux réseaux concernant les charges de travail dans un cluster k8s



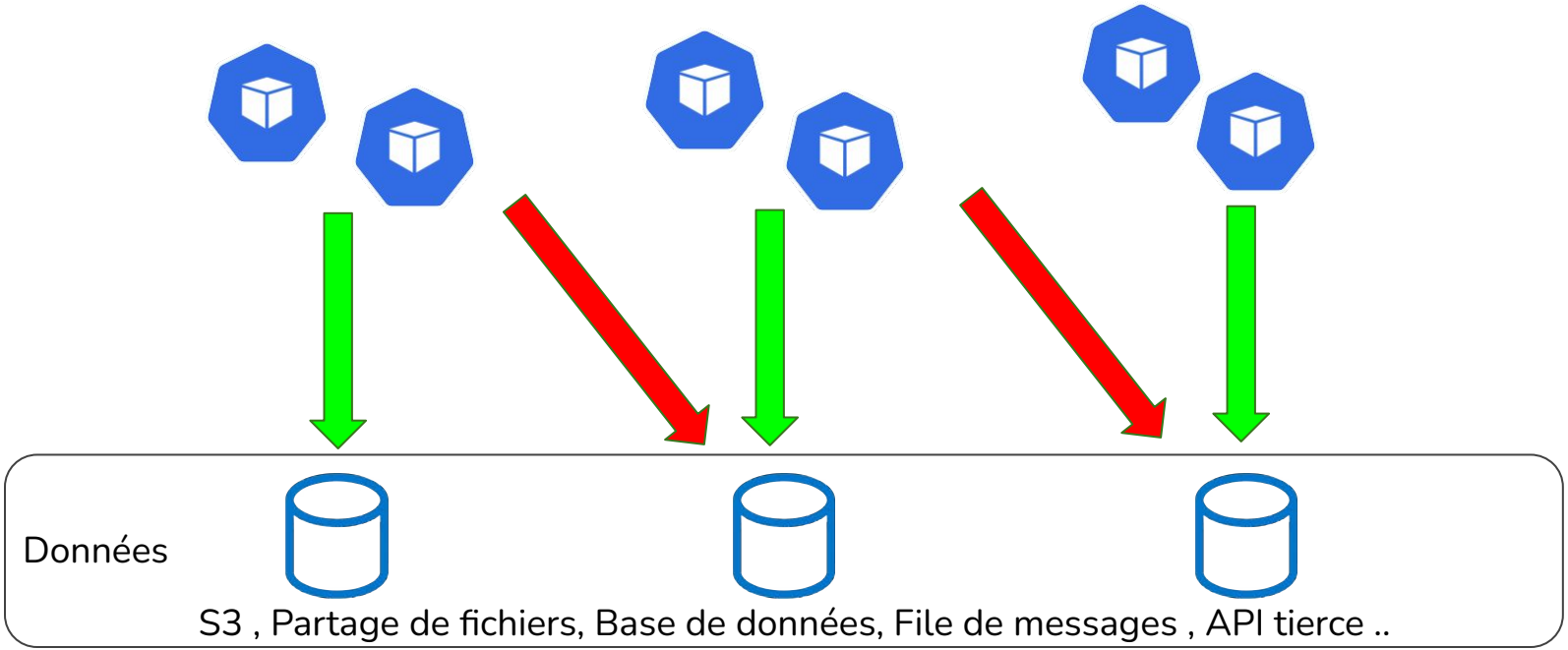
Le sujet du jour : filtrage des flux sortants



Les charges de travail et le stockage des données sont découplés



Les charges de travail et le stockage des données sont découplés



Le cloisonnement réseau : un passage obligé

Network separation and hardening

Cluster networking is a central concept of Kubernetes. Communication among containers, Pods, services, and external services must be taken into consideration. By default, Kubernetes resources are not isolated and do not prevent lateral movement or escalation if a cluster is compromised. Resource separation and encryption can be an effective way to limit a cyber actor's movement and escalation within a cluster.

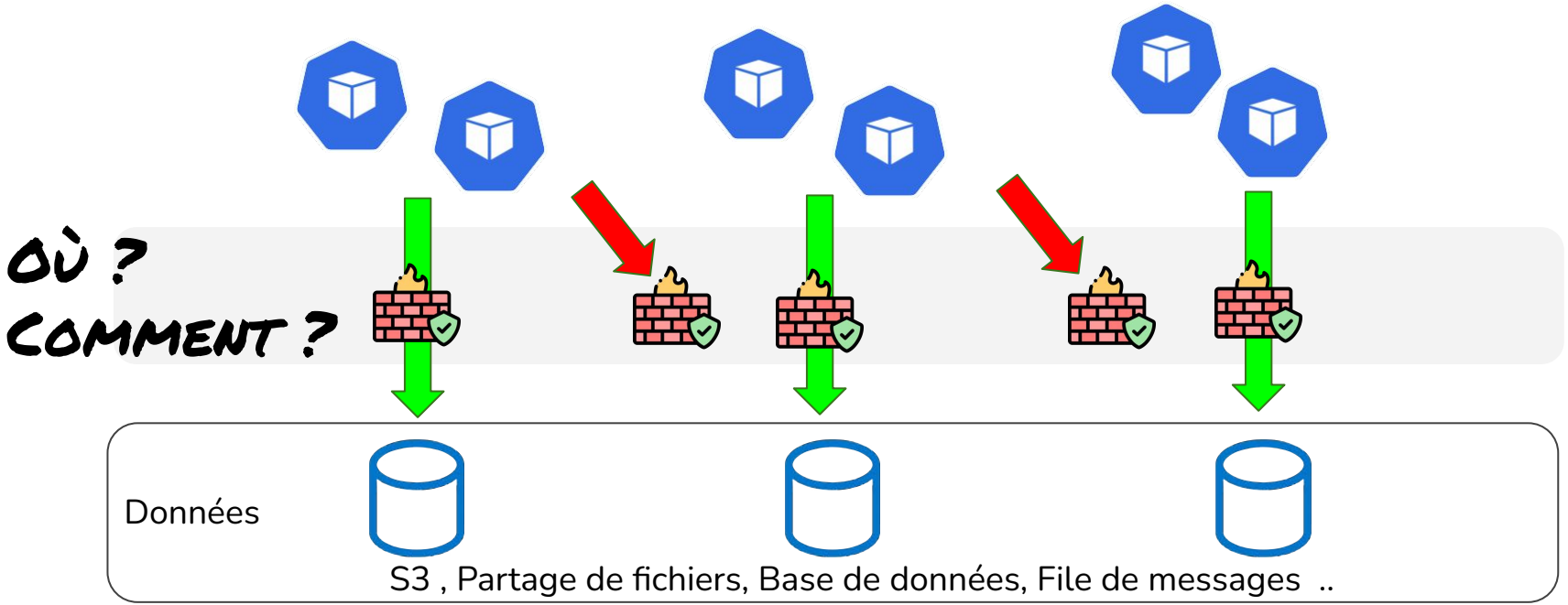
Key points

- Use network policies and firewalls to separate and isolate resources.
- Secure the control plane.
- Encrypt traffic and sensitive data (such as Secrets) at rest.

Network Policies Checklist

- ✓ Use a CNI plugin that supports NetworkPolicy API
- ✓ Create policies that select Pods using podSelector and/or the namespaceSelector
- ✓ Use a default policy to deny all ingress and egress traffic. Ensures unselected Pods are isolated to all namespaces except kube-system

src Kubernetes Hardening Guidance


OÙ ?
COMMENT ?
Données
S3 , Partage de fichiers, Base de données, File de messages ..

Les possibilités d'implémentation

- Filter nativement au moyen d' `Egress Network Policy`

- Filtrer en aval par un firewall réseau "classique"
  - Utiliser l'`IPAM *` pour distinguer les Pods source suivant leur adresses IP
  - Utiliser la `Node Affinity` pour distinguer les Pods suivant leur Node d'hébergement
  - Utiliser les `Egress Gateway` pour distinguer les Pods

(*) IPAM : IP Address Management

Vanilla Kubernetes

- **Network Policy**

Calico Open Source

- eBPF Dataplane
- Windows Dataplane
- Data-in-transit encryption
- Global Network Policy
- **Advanced IPAM**
- Adv Selector for NetPol

Calico Cloud & Enterprise (i.e Tigera)

- GUI
- Staging and recommandation NetPol
- **Egress Gateway**
- IDS, IPS , WAF
- Logs, SIEM
- Threat Protection
- 24x7 Support
- Image Assurance

and more...

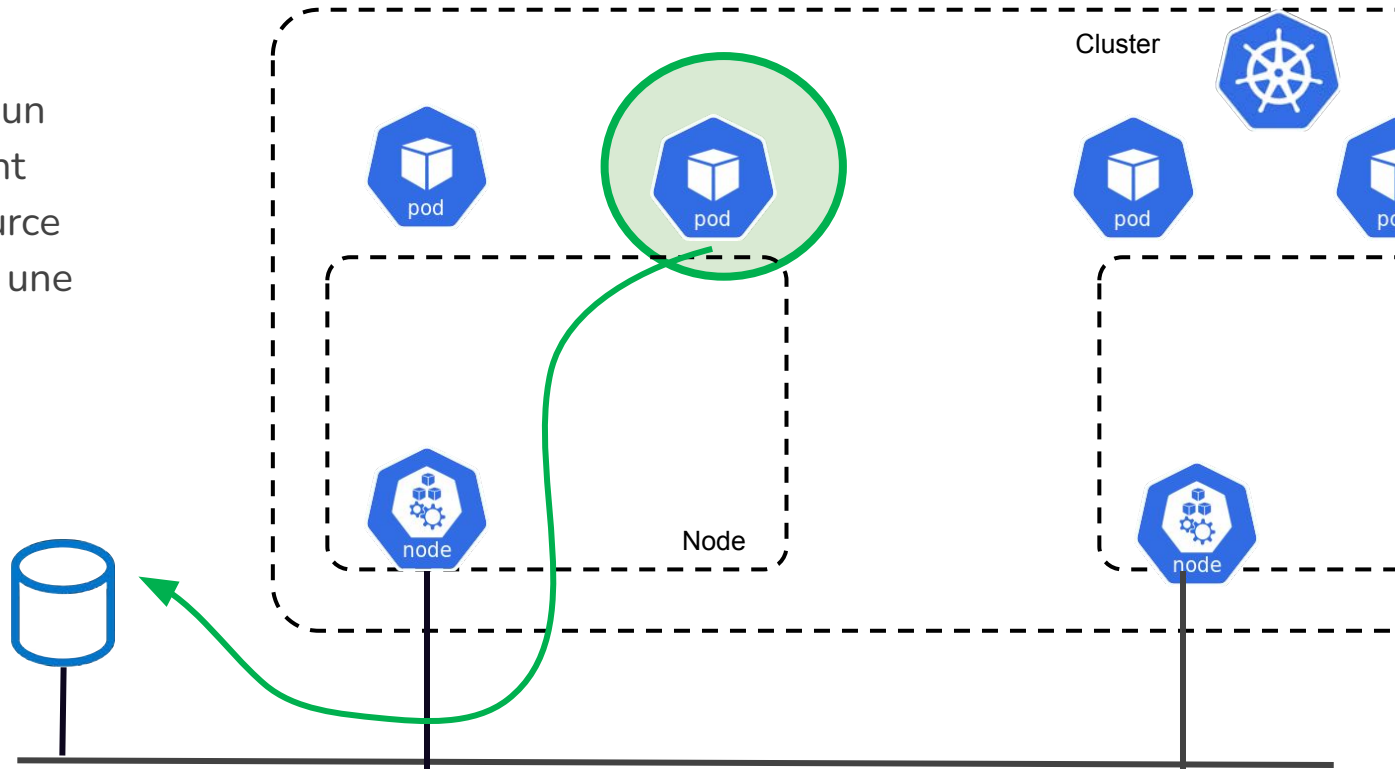
See https://www.tigera.io/tigera-products/compare-products/

BlueTrusty

Stratégie #1 :
filtrer avec les Egress Network Policies

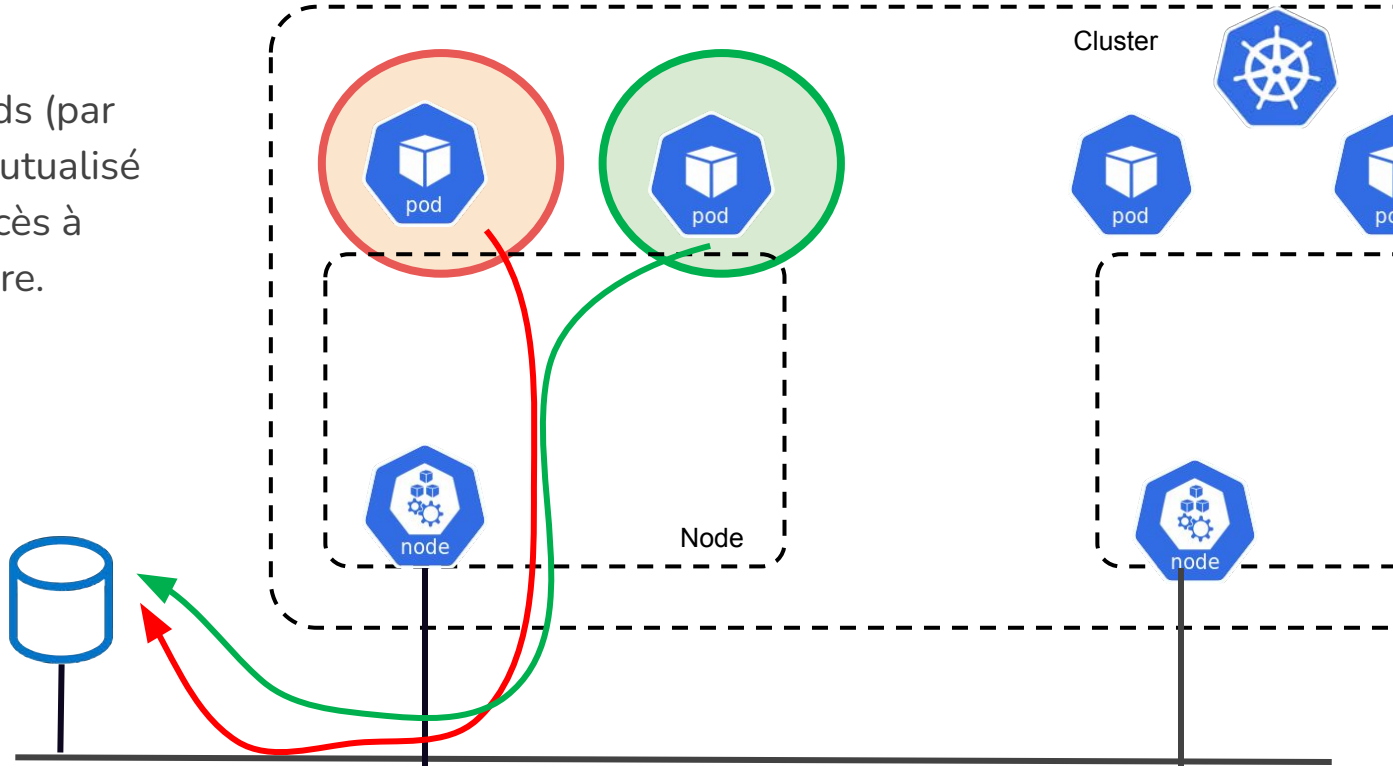
Les flux sortants sont par défaut autorisés…

Certains pods (**verts**) d'un cluster mutualisé doivent avoir accès à une ressource extérieure, par exemple une base de données.



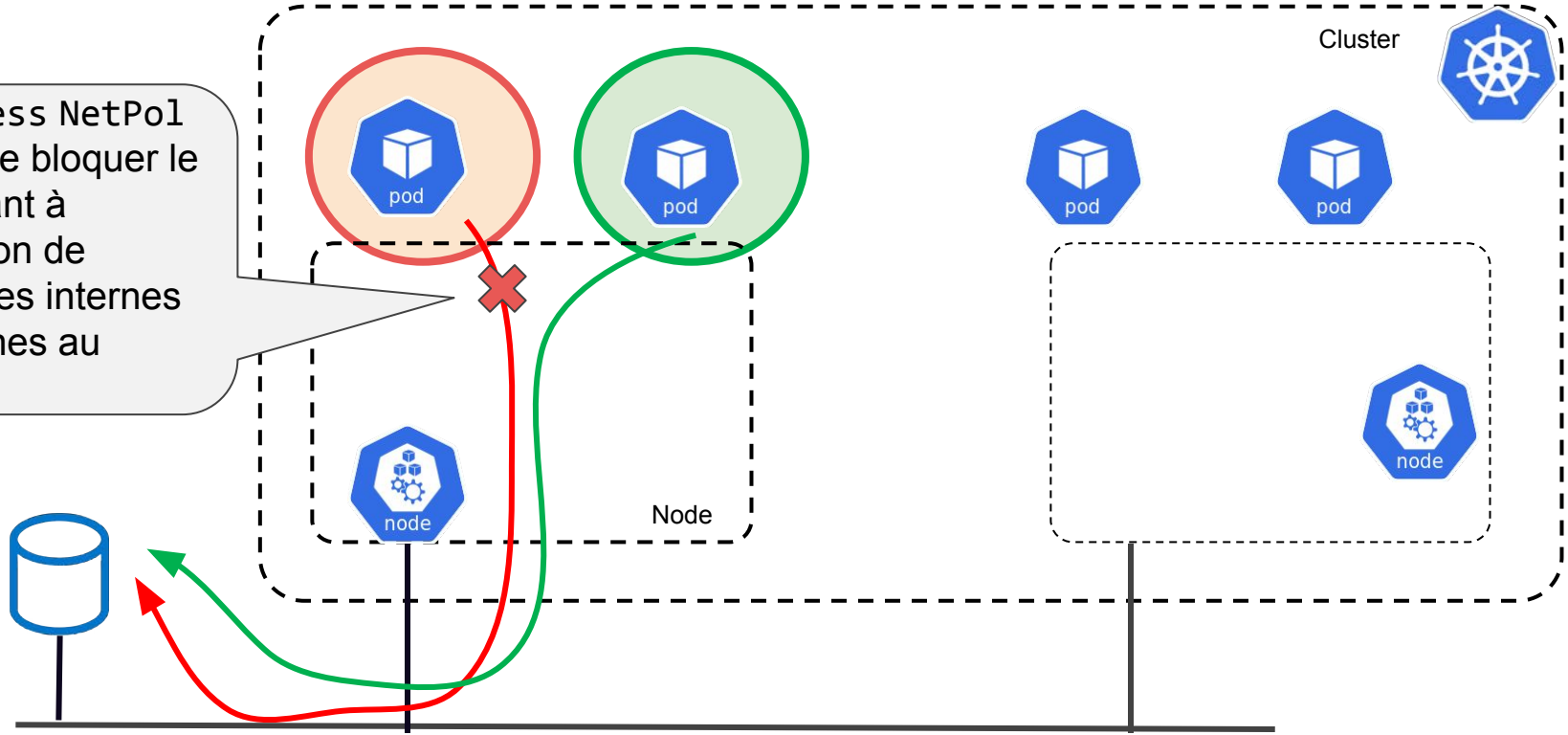
Même ceux qui devraient être filtrés…

Pourtant les autres pods (par ex **rouges**) du cluster mutualisé **ne doivent pas** avoir accès à cette ressource extérieure.



Utilisons une "egress Network Policy"

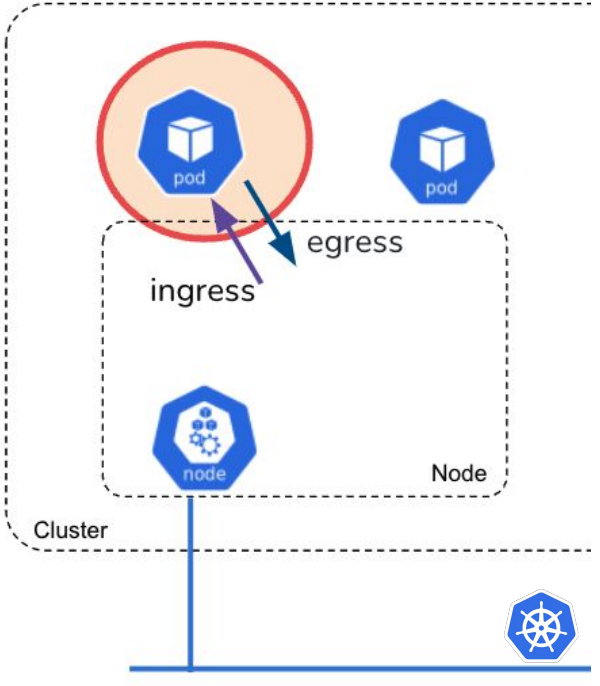
une `egress NetPol` permet de bloquer le flux sortant à destination de ressources internes ou externes au Cluster



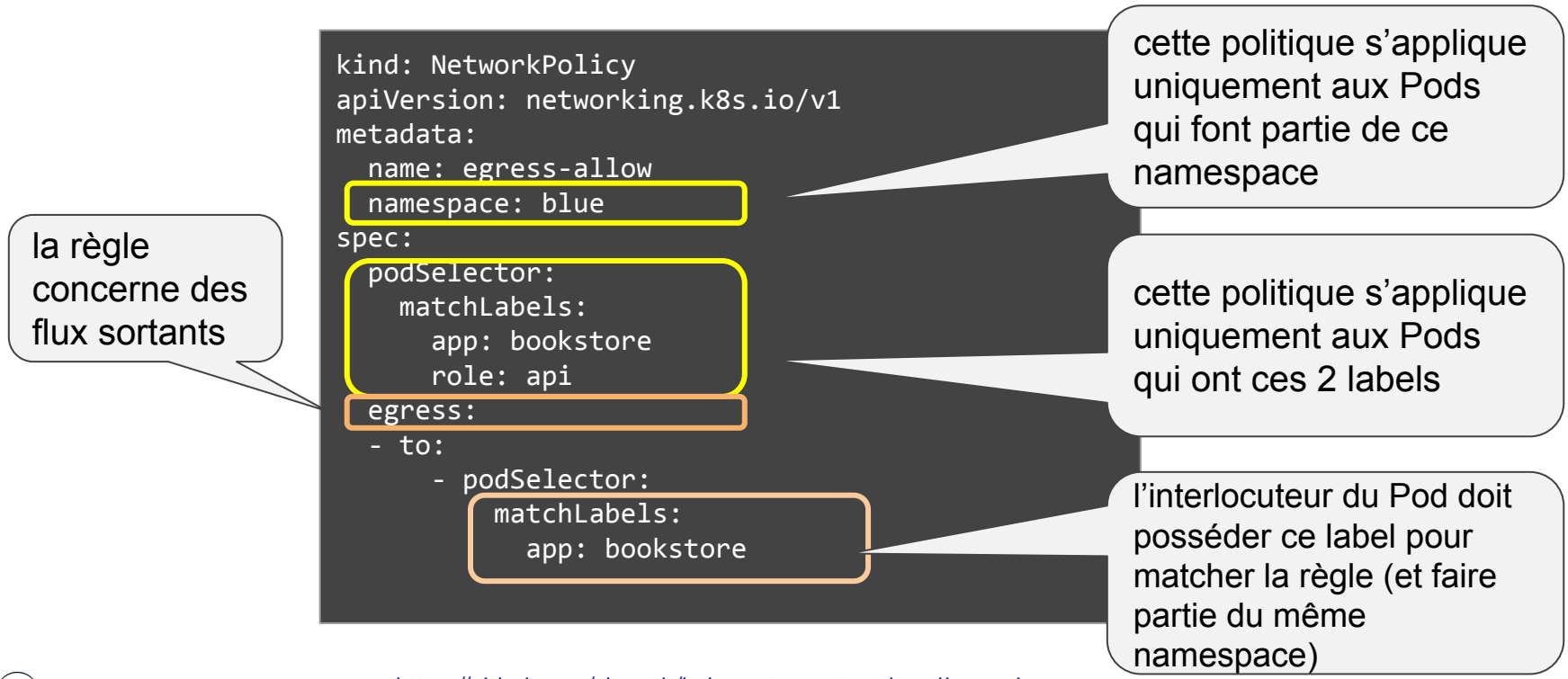
Les Network Policy fonctionnent par liste d’autorisation

- Les Network Policy sont dites “**application centric**” c'est-à-dire qu’elles s’appliquent directement sur les Pods (et non sur les flux en transit).
- En standard, les règles n’ont pas d’ordre car elles ne spécifient **que les flux à autoriser**.
- Une règle finale de ***Deny* implicite** est appliquée pour les Pods dès le moment qu’une Network Policy leur est appliquée.

Cette approche est parfaitement adaptée à la **microsegmentation**, au **Zero Trust Network**.



Analyse d'une première egress NetworkPolicy



src : https://github.com/ahmetb/kubernetes-network-policy-recipes

Les NetPol sont implémentées en iptables/netfilter (ou eBPF)

```
node# iptables -n -L cali-po-_abDLit6fTnFw6lig4Kd
Chain cali-po-_abDLit6fTnFw6lig4Kd (1 references)
target     prot opt source               destination
MARK       all  --  0.0.0.0/0            0.0.0.0/0            /* cali:Ze5J_Y3hmkX4M0_7 */ /* Policy
default/knp.default.allow-except-websrv egress */ MARK xset 0x80000/0x180000
MARK       all  --  0.0.0.0/0            172.18.0.1           /* cali:FDO4Igs10Wy0OLCV */ MARK and 0xfff7ffff
MARK       tcp  --  0.0.0.0/0            0.0.0.0/0            /* cali:YHDVp5ifh1SZWvBi */ multiport dports 80
mark match 0x80000/0x80000 MARK or 0x10000
NFLOG      all  --  0.0.0.0/0            0.0.0.0/0            /* cali:iCdg_FGT7IuUTN7j */ mark match
0x10000/0x10000 nflog-prefix  "APE0|default/knp.default.allow-except-websrv" nflog-group 2 nflog-size 80
RETURN     all  --  0.0.0.0/0            0.0.0.0/0            /* cali:Gqsec9OCBYQzMdcK */ mark match
0x10000/0x10000
```

Sous Calico, des chaînes iptables sont créés : `calico-fw` (from workload) , `-tw` (to workload) , `-po` (policy)

La syntaxe est *pointue*

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: egress-allow
  namespace: default
spec:
  podSelector: {}
  egress:
  - to:
      - namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: default
        podSelector:
          matchLabels:
            app: bookstore
```

Cette politique s'applique à tous (liste vide) les pods du namespace `default`

Les pods ayant le label `app=bookstore` et contenus dans un Namespace qui a le label `metadata.name: default`

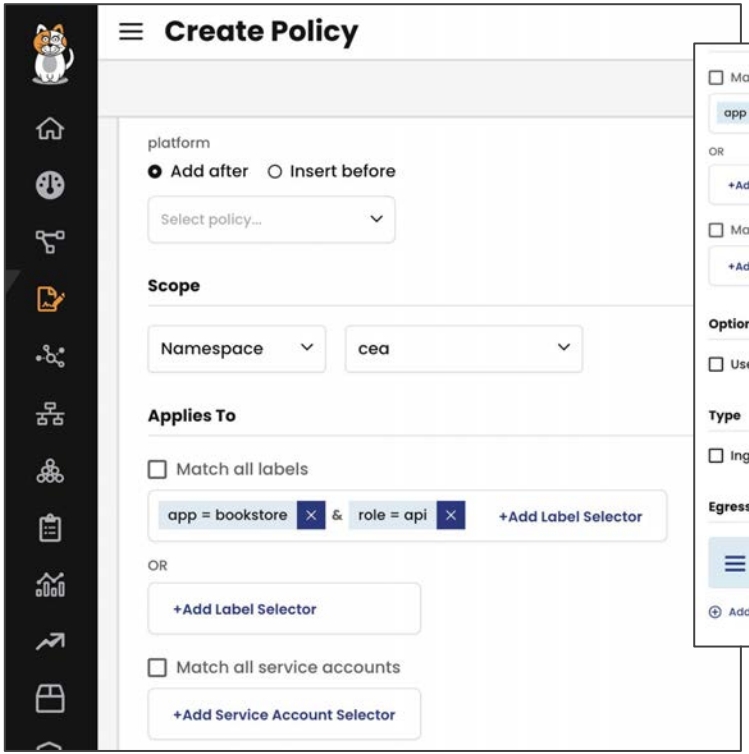
```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: egress-allow
  namespace: default
spec:
  podSelector: {}
   egress:
  - to:
      - namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: default
      - podSelector:
          matchLabels:
            app: bookstore
```

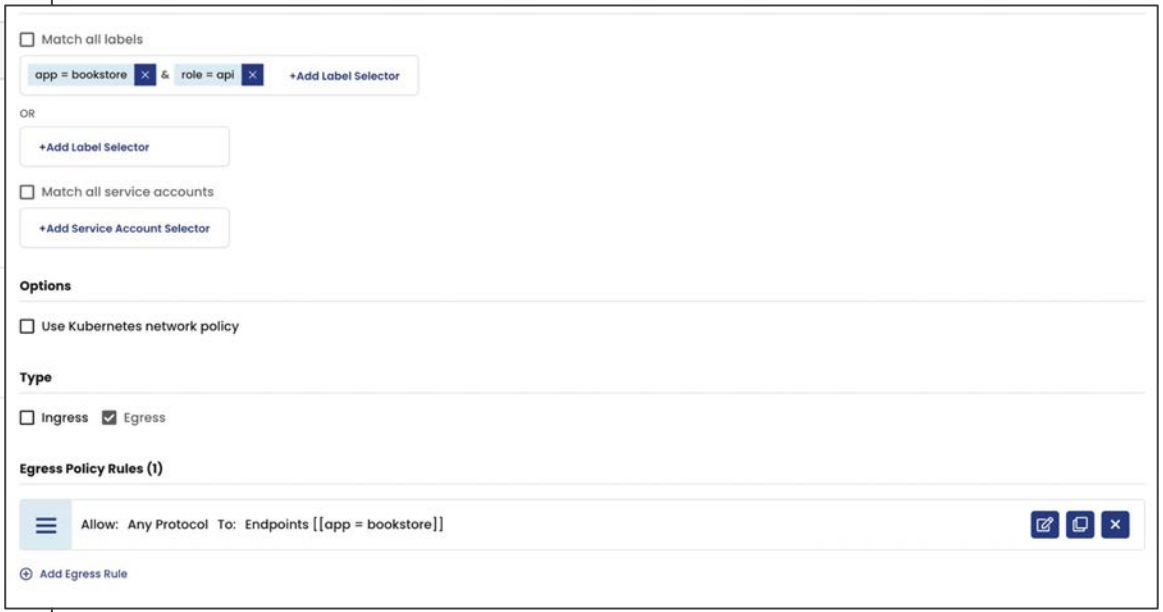
Les pods ayant le label `app=bookstore` ou contenus dans un Namespace qui a le label `metadata.name: default`

BlueTrusty

src = https://kubernetes.io/docs/concepts/services-networking/network-policies/

Le GUI de Calico facilite la vie





L’écriture est parfois alambiquée

Seuls certains pods doivent accéder à un serveur externe.

Il faut donc appliquer une Network Policy Egress qui permet l’accès depuis certains Pods.

Par défaut, le reste des flux est interdit….

la sortie en HTTP est autorisée vers 172.18.0.1 uniquement pour les Pods qui ont le label `type=test`

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-websrv
spec:
  podSelector:
    matchLabels:
      type: test
  policyTypes:
    - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 172.18.0.1/32
    ports:
      - protocol: TCP
        port: 80
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-except-websrv
spec:
  podSelector: {}
  policyTypes:
    - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0
        except:
          - 172.18.0.1/32
    ports:
    - protocol: TCP
      port: 80
```

la sortie en HTTP est autorisée pour tous, sauf vers 172.18.0.1

Les NetPol standards sont “namespacées”

Seuls certains pods doivent accéder à un serveur externe.
Il faut donc appliquer une NetPol Egress qui permet l’accès depuis certains Pods.
Par défaut, le reste des flux est interdit….

…. enfin, tant que cette Network Policy leur sera appliquée !

….ce qui ne sera pas le cas des Pods logés dans un namespace créé postérieurement.

What you can't do with network policies (at least, not yet)

src: https://kubernetes.io/docs/concepts/services-networking/network-policies/

As of Kubernetes 1.29 the following functionality does not exist in the NetworkPolicy API, but you might be able to implement workarounds using Operating System components (such as SELinux, OpenVSwitch, IPTables, and so on) or Layer 7 technologies (Ingress controllers, Service Mesh implementations) or admission controllers. In case you are new to network security in Kubernetes, its worth noting that the following User Stories cannot (yet) be implemented using the NetworkPolicy API.

- Forcing internal cluster traffic to go through a common gateway (this might be best served with a service mesh or other proxy). OR EGRESS GATEWAY
- Anything TLS related (use a service mesh or ingress controller for this).
- Node specific policies (you can use CIDR notation for these, but you cannot target nodes by their Kubernetes identities specifically).
- Targeting of services by name (you can, however, target pods or namespaces by their labels, which is often a viable workaround).
- Creation or management of "Policy requests" that are fulfilled by a third party.
- Default policies which are applied to all namespaces or pods (there are some third party Kubernetes distributions and projects which can do this).
- Advanced policy querying and reachability tooling.
- The ability to log network security events (for example connections that are blocked or accepted).
- The ability to explicitly deny policies (currently the model for NetworkPolicies are deny by default, with only the ability to add allow rules).
- The ability to prevent loopback or incoming host traffic (Pods cannot currently block localhost access, nor do they have the ability to block access from their resident node).

Perte de l'identité lors de interconnexion avec le Legacy

Gouvernance

Politique de filtrage

Observabilité

Des Network Policy “étendues” sont nécessaires

Par exemple, avec le plugin CNI

ProjectCalico, les `Network Policy`

apportent -entre autres -

des **actions distinctes** (Allow, Deny, Log) ,

une **sémantique plus riche**

et peuvent s’appliquer à tout le Cluster

(**Global** Network Policy)

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: gnp-allow-from-test
spec:
  namespaceSelector: projectcalico.org/name not in
{"kube-system", "calico-system", "tigera-system"}
  types:
  - Egress
  egress:
  - action: Allow
    protocol: TCP
    source:
      selector: type == "test" &&
projectcalico.org/namespace == "default"
    destination:
      nets:
        - 172.18.0.1/32
      ports:
        - 80
```

https://projectcalico.docs.tigera.io/reference/resources/globalnetworkpolicy#entityrule

Dernier obstacle : numérotation des règles

Attends !!

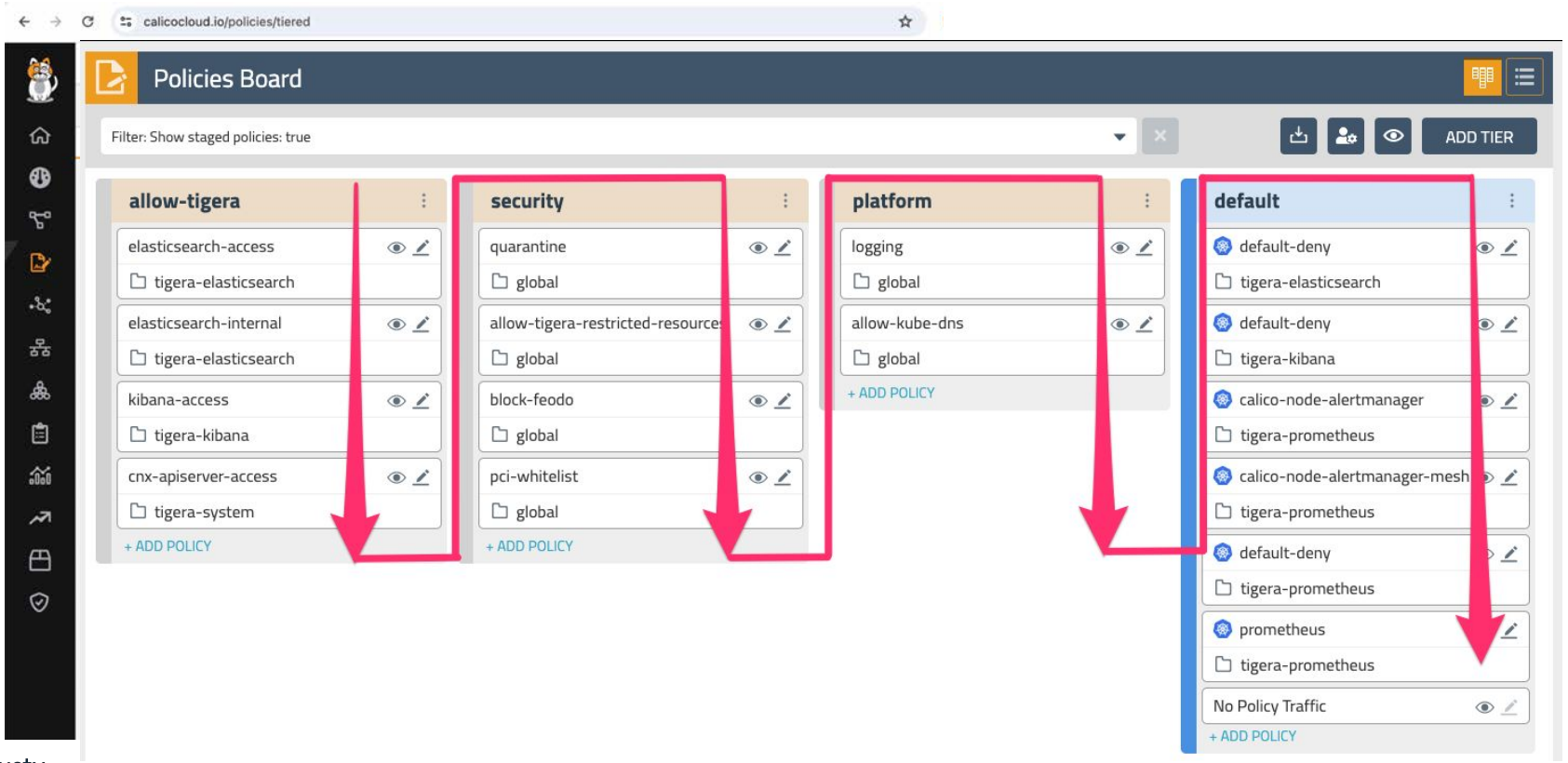
Je veux également que les Pods puissent résoudre les noms de domaine, mais conserver ma règle en Deny All....

10	Deny HTTP vers 172.18.0.1
20	Allow HTTP + HTTPS vers 172.16.0.0/16
30	Allow UDP+TCP/53 vers serveurs DNS
40	Deny All

```yaml
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
 name: allowhttp
spec:
 order: 20
 types:
   - Egress
 egress:
   - action: Allow
     protocol: TCP
     source: {}
     destination:
       nets:
         - 172.16.0.0/16
       ports:
         - 80
         - 443
```

https://projectcalico.docs.tigera.io/reference/resources/globalnetworkpolicy#entityrule

Le GUI de Calico facilite la vie

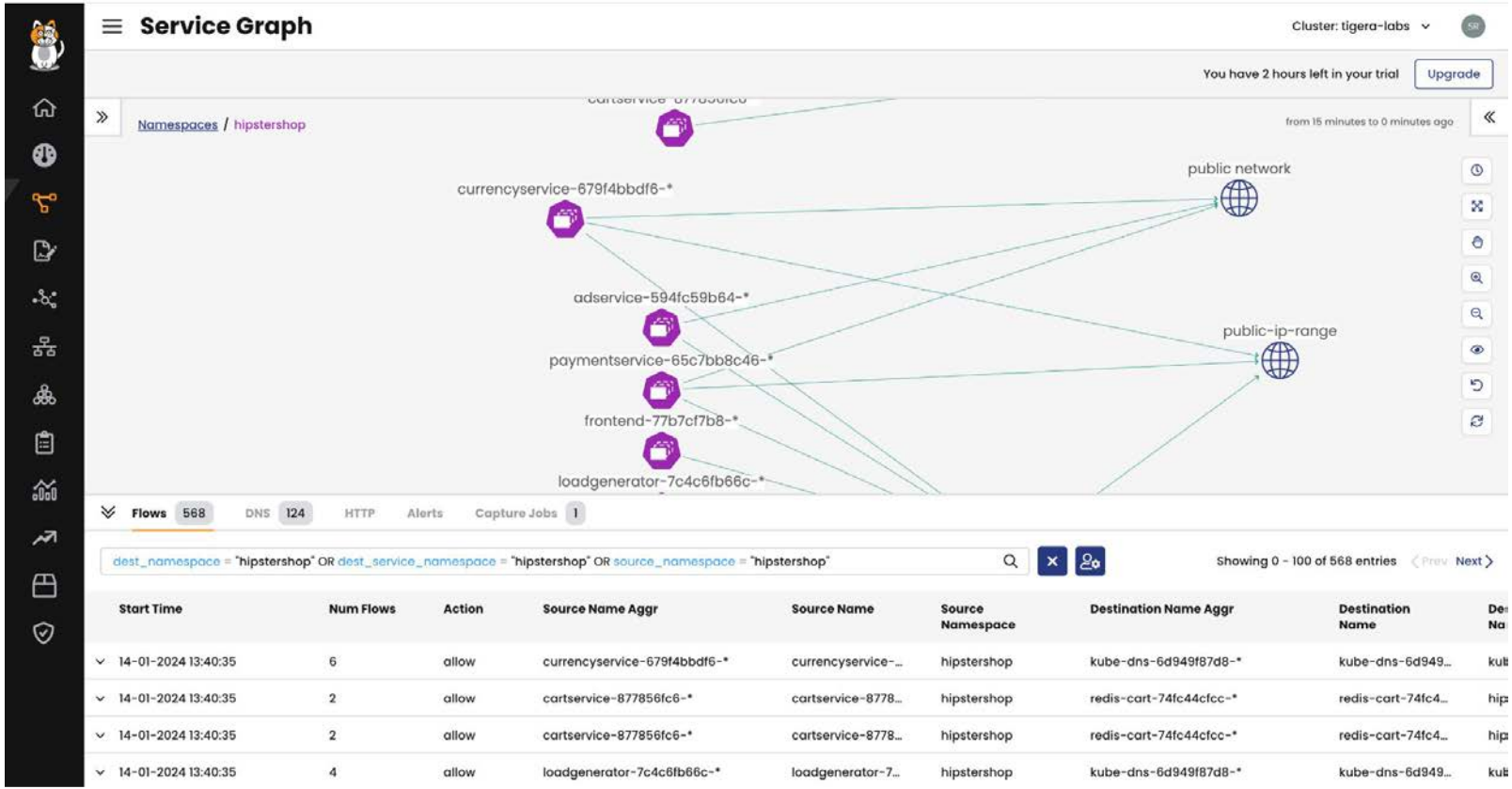


https://www.eksworkshop.com/beginner/120_network-policies/calico-enterprise/lab-3/

Stratégie #1 : Utilisons les Network Policy

✔ Natif et standard à K8s	✘ Sémantique limitée, ce qui nuit à la maintenabilité
✔ Conforme à la culture GitOps	✘ Nécessite une bonne compréhension des mécanismes k8s et réseau, pas évident pour les équipes “Legacy” ..
✔ Basé sur une liste d’autorisation	

Le GUI de Calico facilite la vie



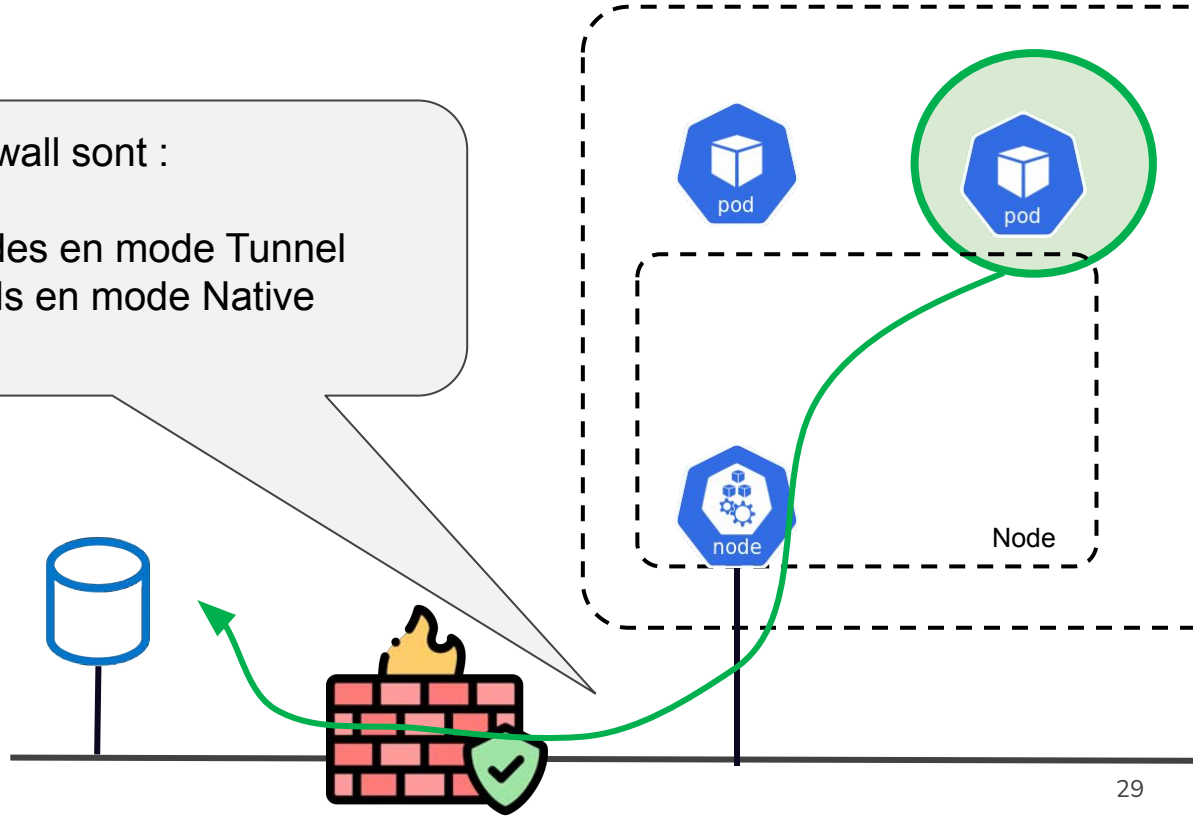
BlueTrusty

Stratégie #2 :
filtrer en aval avec un Firewall

Problème : les IPs sont souvent imprévisibles

Les IP vues du Firewall sont :

- celles des Nodes en mode Tunnel
- celles des Pods en mode Native Routing



Tactique #1 : Plaçons les Pods à autoriser sur des Nodes spécifiques

L'idée est de placer les Pods qui auront des autorisations de flux sur **certains** Nodes dont les IP "publiques" sont connues

```
kubectl label node worker zone=dmz –overwrite
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      type: test
  template:
    metadata:
      labels:
        type: test
    spec:
      containers:
      - name: nginx
        image: nginx
      nodeSelector:
        zone: dmz
```

Tactique #1 : Plaçons les Pods à autoriser sur des Nodes spécifiques

✔ Simplicité	✘ moindre optimisation du placement des Pods
	✘ applicable uniquement en mode Tunnel/Encapsulation
	✘ **Problématique de Haute Disponibilité** les Nodes n'ont pas vocation à être éternels

Tactique #2 : via IPAM, affectons des IP réservées aux Pods à autoriser

On peut attribuer des IPs aux Pods selon leurs labels ou leur NameSpace d'appartenance.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-ipam-pool
  annotations:
    cni.projectcalico.org/ipv4pools: '["248-100-pool"]'
spec:
  containers:
  - name: nginx
    image: nginx
```

```yaml
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: 248-100-pool
spec:
  cidr: 10.248.100.0/24
  blockSize: 29
  ipipMode: Always
  natOutgoing: false
  nodeSelector: !all()
```

Tactique #2 : via IPAM, affectons des IP réservées aux Pods à autoriser

On peut attribuer des IPs aux Pods selon leurs labels ou leur Namespace d'appartenance.Les plages (CIDR) sont subdivisées en Blocs attribués aux Nodes.

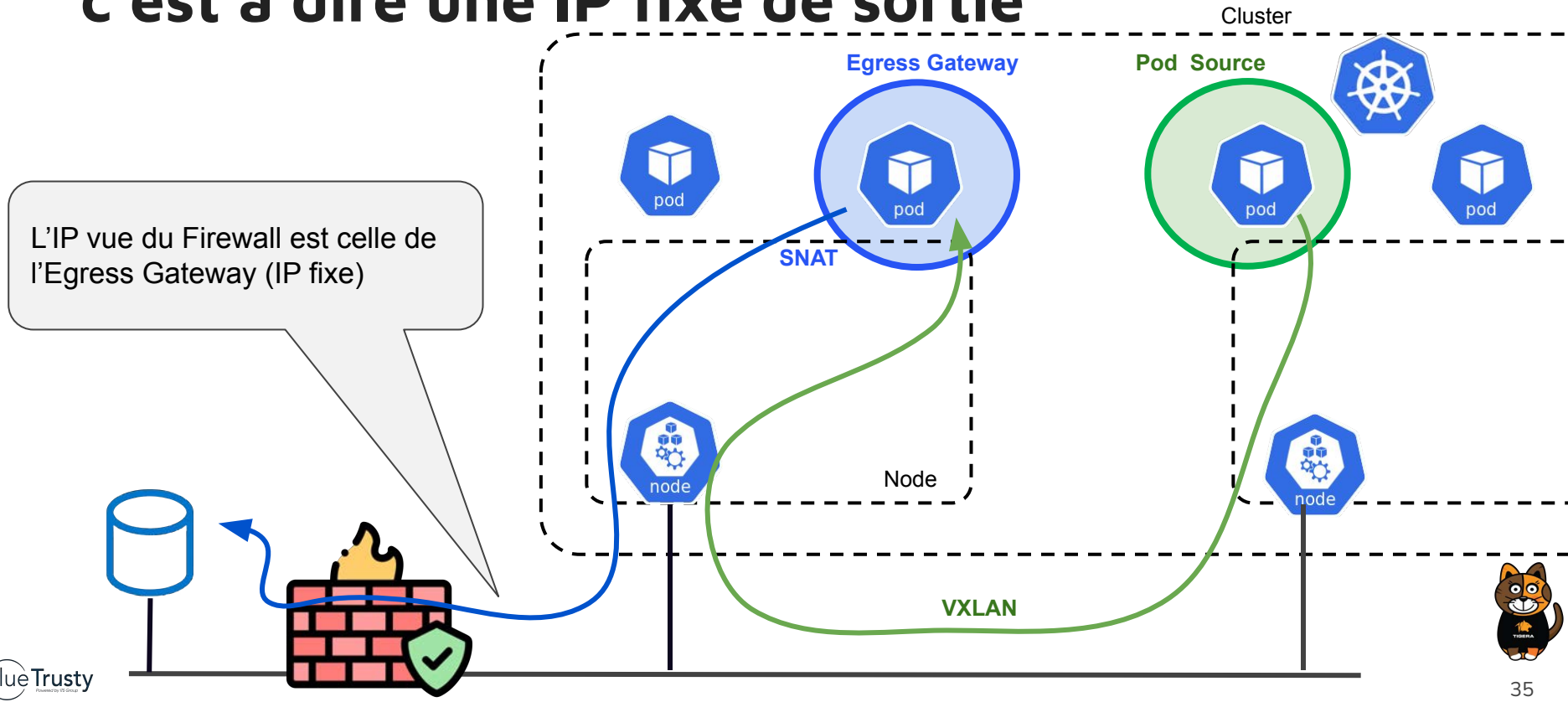
```
NAME                              READY   STATUS    RESTARTS   AGE   IP              NODE            NOM
pod-ipam-pool                     1/1     Running   0          72s   10.248.100.204  basic-worker    <no
pod-std-pool-6799fc88d8-dh2gv     1/1     Running   0          72s   10.244.0.195    basic-worker    <no
pod-std-pool-6799fc88d8-f4dgc     1/1     Running   0          72s   10.244.0.196    basic-worker    <no
pod-std-pool-6799fc88d8-f7pmd     1/1     Running   0          72s   10.244.0.134    basic-worker2   <no
```

Dans un architecture sans SNAT (aka Transparent Mode ou Native Routing, c-a-d pas d'encapsulation) on peut différencier les Pods par leur adresse IP au niveau de firewalls legacy sur le chemin vers les ressources externes.

Tactique #2 : via IPAM, affectons des IP réservées aux Pods à autoriser

✔ Simple et élégant coté k8s	✘ Seul le mode Transparent/Native Routing est pertinent pour cette stratégie ; en mode Tunnel/Encapsulation, les Pods sont de toute façon SNATés
	✘ En environnement Cloud managé (ex Azure AKS) , il faut souvent débrancher l'IPAM intégré et gérer le routage avec le reste du réseau.
	✘ La difficulté est d'annoncer les Pods CIDR au monde extérieur, cela se fait classiquement en BGP (supporté dans ProjectCalico)

Tactique #3 : utiliser une Egress Gateway, c'est à dire une IP fixe de sortie



Tactique #3 : utiliser une Egress Gateway, c'est à dire une IP fixe de sortie

On crée le Pool des IP Publiques à utiliser et on crée un *deployment* de plusieurs Pods d'image `egress-gateway`

```yaml
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: egress-ippool-1
spec:
  cidr: 10.246.254.0/31
  blockSize: 31
  nodeSelector: "!all()"
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
[..]
spec:
  replicas: 1
  selector:
    matchLabels:
      egress-code: egw
  template:
    metadata:
      annotations:
        cni.projectcalico.org/ipv4pools: "["egress-ippool-1"]"
      labels:
        egress-code: egw
```

On annote le namespace (ou le déployment) pour que ses Pods utilisent l'Egress Gateway

```
kubectl annotate ns <namespace> egress.projectcalico.org/selector="egress-code == egw"
```

Les mécanismes sous-jacents sont ceux de Linux

```
node# ip rule
0:    from all lookup local
100:  from 10.244.0.76 fwmark 0x80000/0x80000 lookup 250
32766:     from all lookup main
32767:     from all lookup default

node# ip route list table 250
default via 10.246.254.0 dev egress.calico onlink

node# ip route list table main
default via 172.18.0.1 dev eth0
blackhole 10.244.0.64/26 proto 80
10.244.0.65 dev cali1b6d4f12a42 scope link
...
10.244.0.192/26 via 10.244.0.192 dev vxlan.calico onlink
172.18.0.0/16 dev eth0 proto kernel scope link src 172.18.0.4

node# ip -d link show egress.calico
72: egress.calico: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 [..] mode group default
    vxlan id 4097 local 172.18.0.4 dev eth0 srcport 0 0 dstport 4790 nolearning ttl auto
```

Tables de routage multiples, vxlan...

On peut concevoir des stratégies complexes

```
apiVersion: projectcalico.org/v3
kind: EgressGatewayPolicy
metadata:
  name: "egw-policy1"
spec:
  rules:
  - destination:
      cidr: 10.0.0.0/8
    description: "Local: no gateway"
  - destination:
      cidr: 11.0.0.0/8
    description: "Gateway to on prem"
    gateway:
      namespaceSelector: "projectcalico.org/name == 'default'"
      selector: "egress-code == 'blue'"
  - description: "Gateway to internet"
    gateway:
      namespaceSelector: "projectcalico.org/name == 'default'"
      selector: "egress-code == 'red'"
    gatewayPreference: PreferNodeLocal
```

Tactique #3 : utiliser une Egress Gateway, c'est à dire une IP fixe de sortie

✔ Simple et élégant coté k8s	✘ Nécessite une topologie réseau supportée (AWS, Azure , OnPrem avec BGP)
✔ Méthode plus fiable que `NodeAffinity`	✘ Nécessite la version Entreprise de Calico
✔ Utilise les mécanismes de k8s pour la haute disponibilité puisque les Egress Gateways sont des Pods	
✔ Répartition de charge possible sur plusieurs Egress Gateways	

Conclusion

Filtrez les flux sortants !

Utilisez les fonctionnalités avancées de votre plugin CNI pour écrire des politiques de filtrage efficaces et maintenables (les GUI aident !)

Utilisez les fonctionnalités IPAM ou Egress Gateway de votre plugin CNI pour rendre l'IP des Pods distinguables

BlueTrusty

Temps pour les questions…