BlueTrusty

Sécurisation réseau dans Kubernetes

Episode 5

Stéphane REYTAN
sreytan@bluetrusty.com

BlueTrusty fournit des services de CyberSécurité.

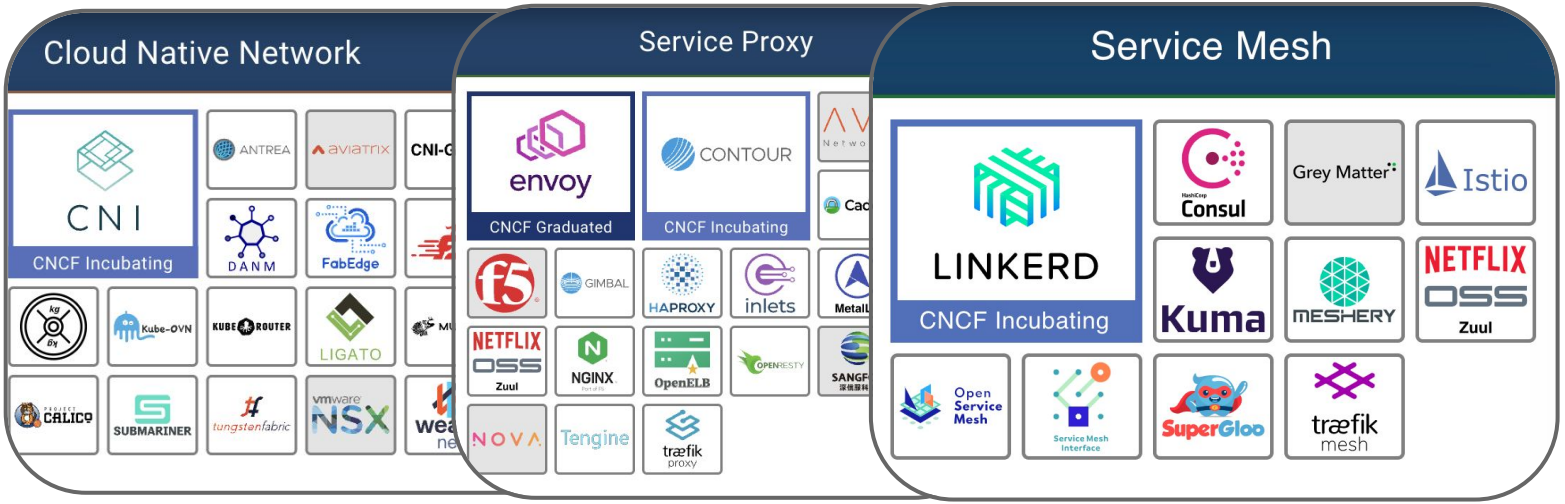
Nous fournissons des services de formation, d'audit et d'assistance sur Kubernetes et les architectures Cloud Native depuis 2017.

BlueTrusty est partenaire revendeur de Calico/Tigera

PROJECT CALICO

TIGERA

L’ambition de cette série de webinars



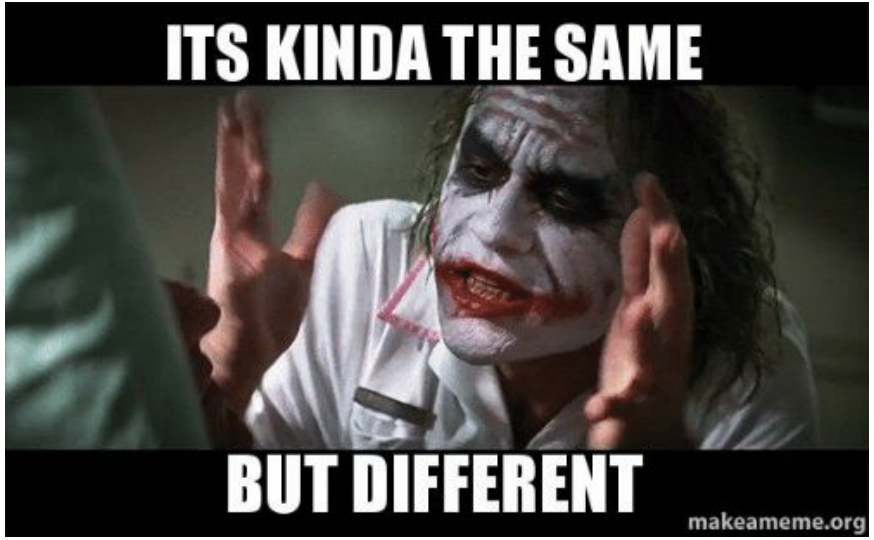
https://landscape.cncf.io/

Avertissement

Je présente les architectures ou implémentations les plus fréquentes.

Kubernetes laisse beaucoup de liberté, je n'ai pas l'ambition d'être exhaustif dans cette présentation.

Kubernetes est encore un projet qui évolue : les fonctionnalités, limitations et patterns d'architecture présentés sont contemporains de ce webinar.



BlueTrusty

Quelles stratégies pour filtrer les flux sortants d'un Cluster Kubernetes ?

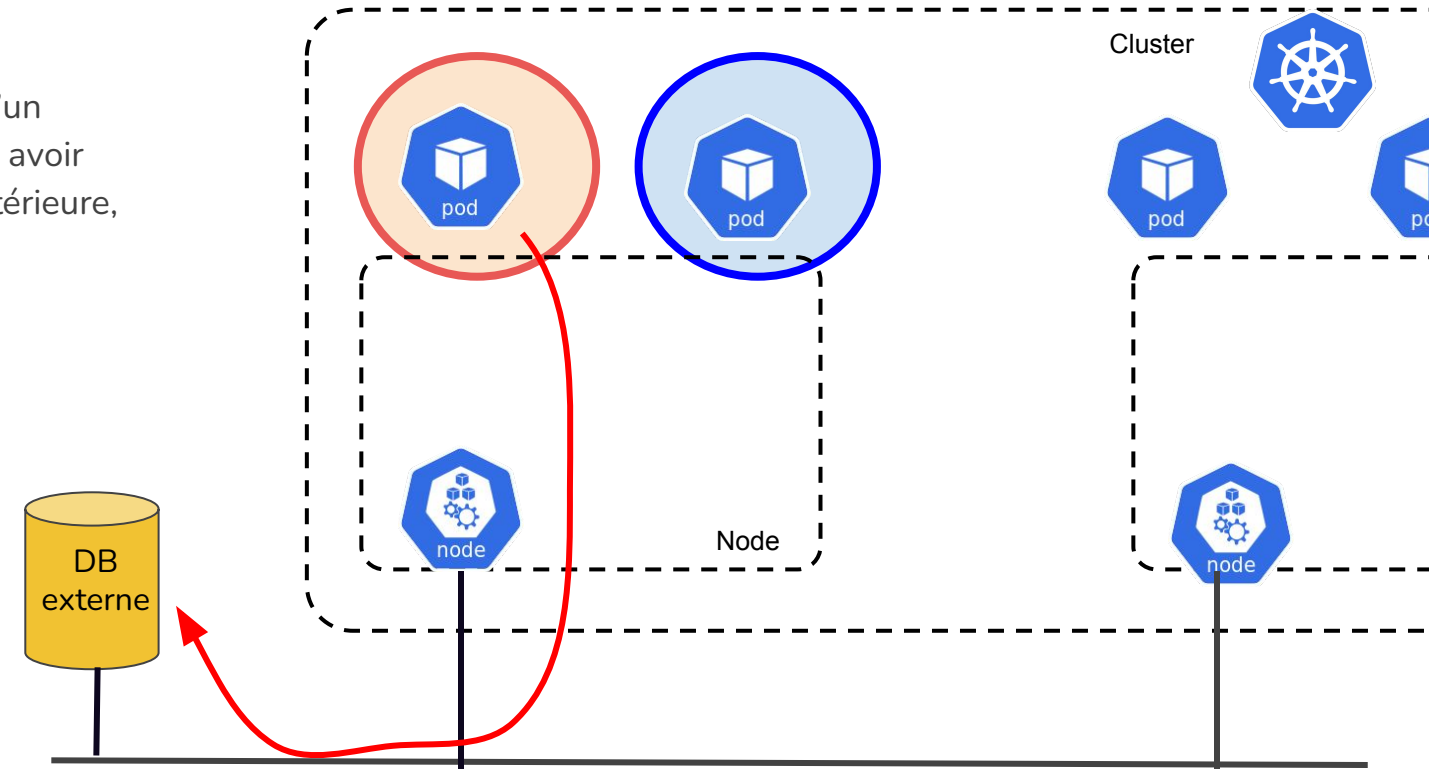
Episode 5

Agenda

- **Le problème avec le filtrage des flux sortants**
- Stratégie #1 : Egress Network Policy
- Stratégie #2 : Placer les Pods sur certains noeuds (`Node Affinity`)
- Stratégie #3 : Utiliser IPAM pour discriminer les Pods
- Stratégie #4 : Utiliser des Egress Gateway

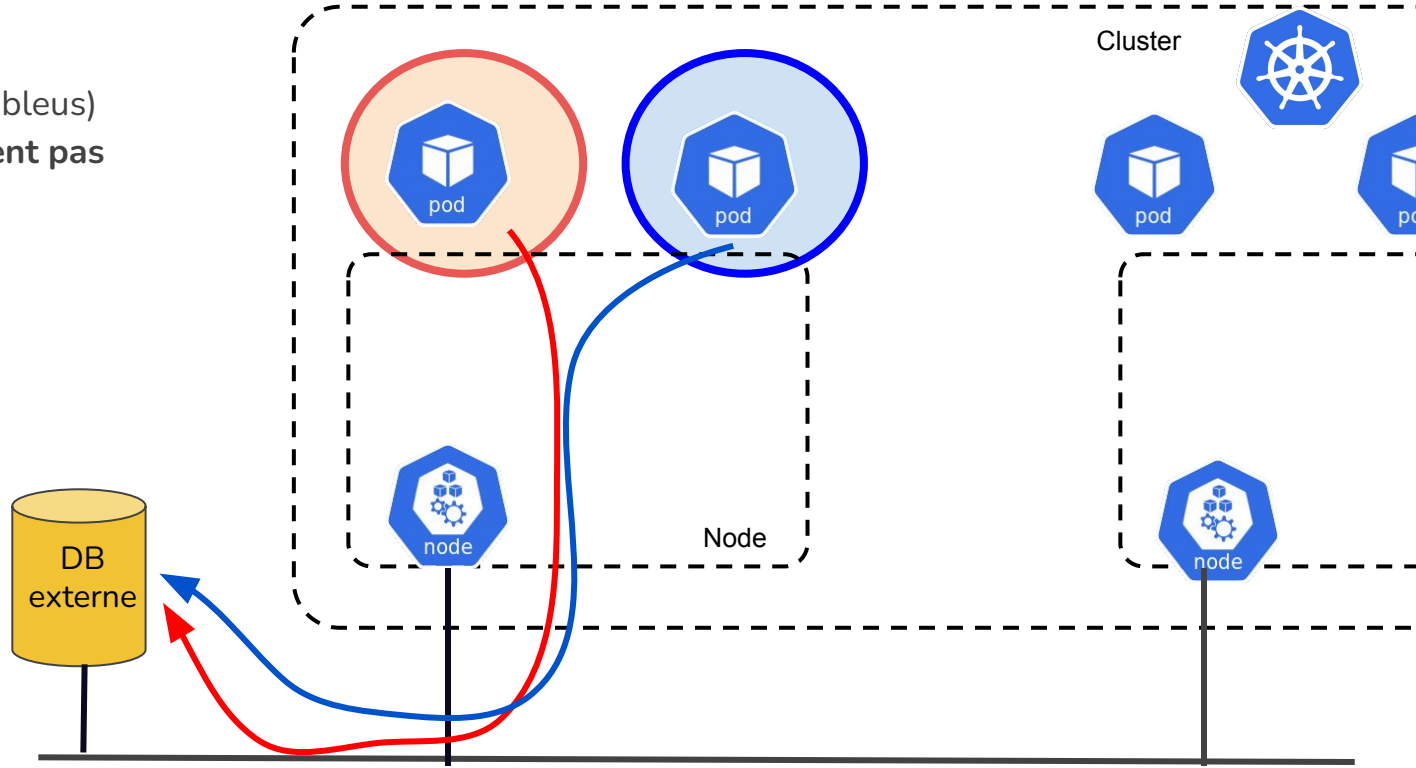
Les flux sortants sont par défaut autorisés…

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



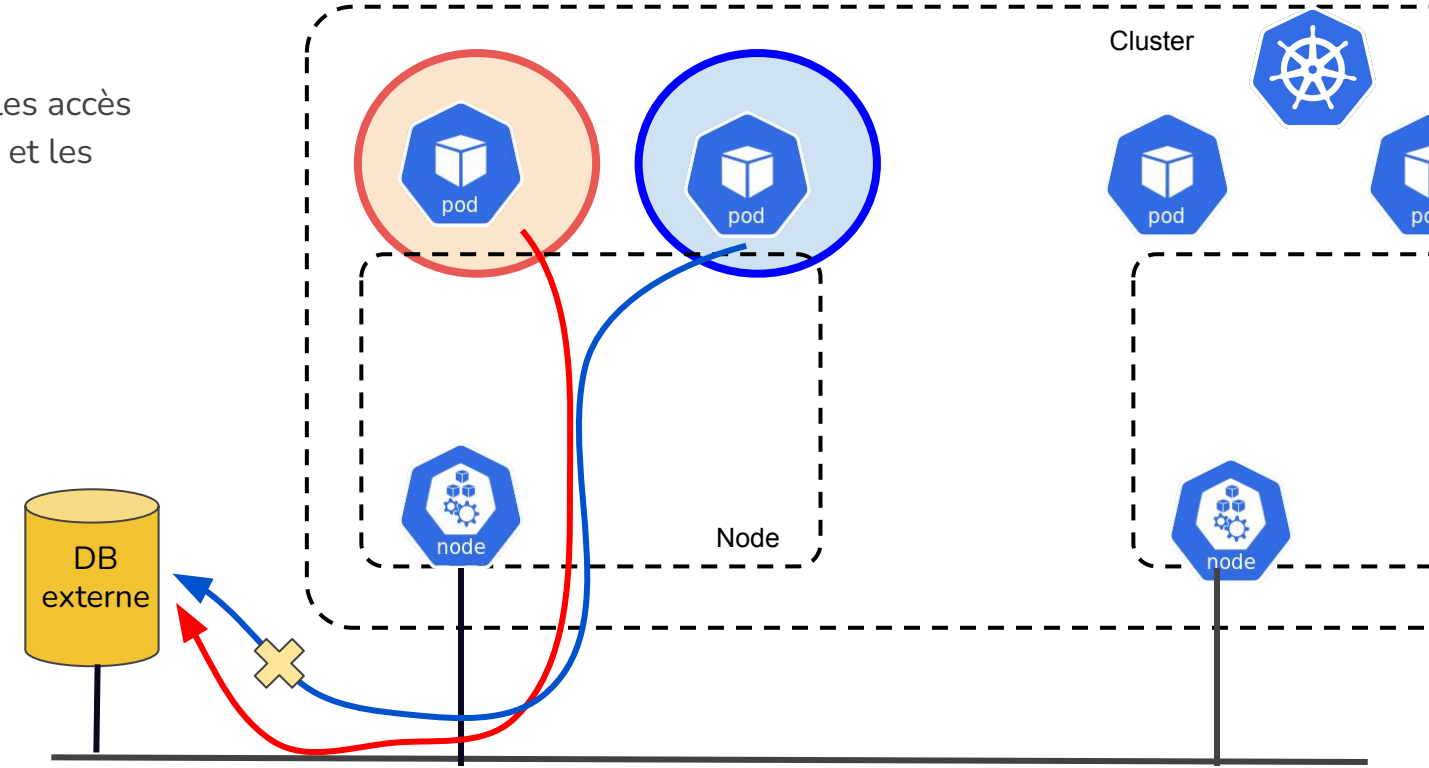
Tous les flux sortants…

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



Notre objectif : un filtrage des flux sortants

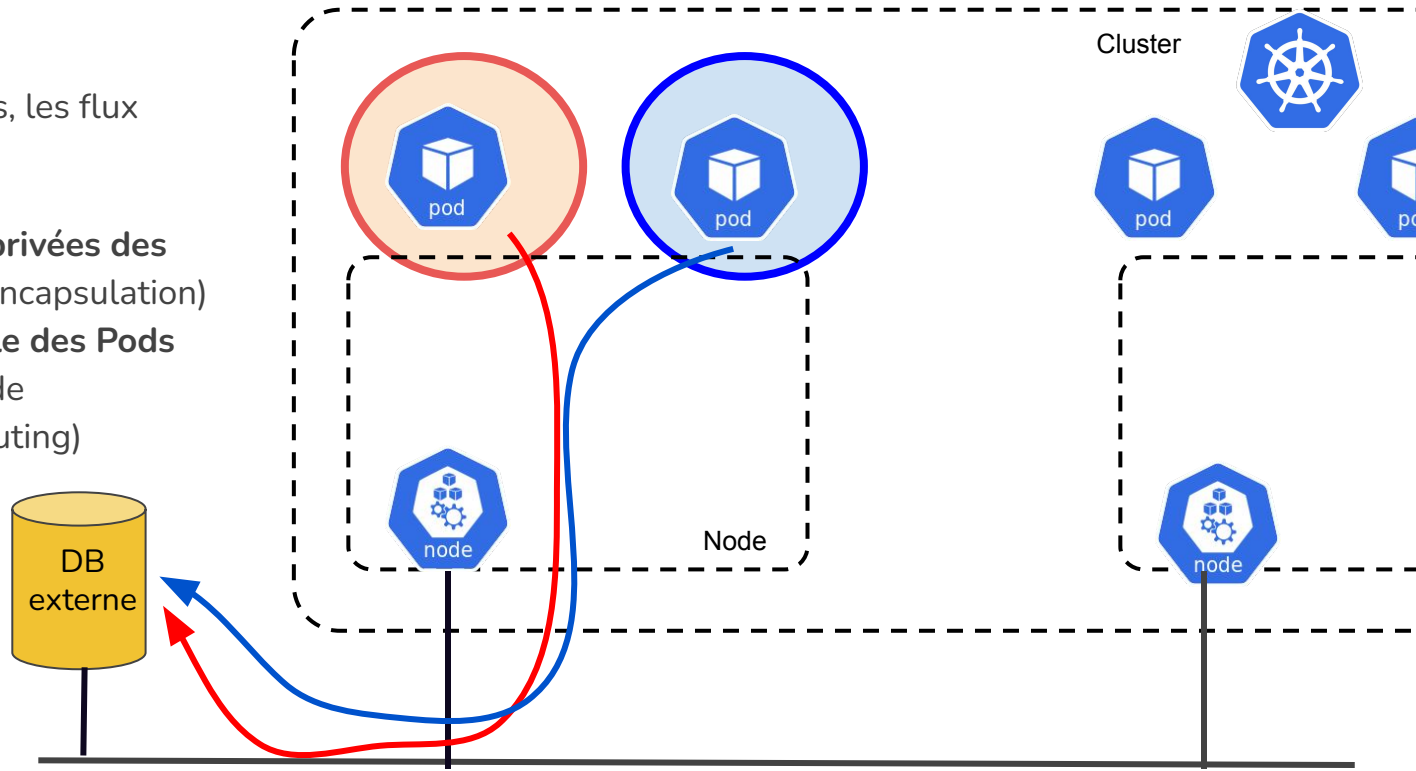
D'une façon ou d'une autre, les accès rouge doivent être **autorisés** et les accès bleus **bloqués**.



Par défaut, les flux sortants sont indiscriminables

Vu des ressources extérieures, les flux proviennent :

- **soit des IP publiques/privées des Nodes** (mode Tunnel/Encapsulation)
- **soit de la plage globale des Pods** `PodSubnet CIDR`, (mode Transparent/Native Routing)



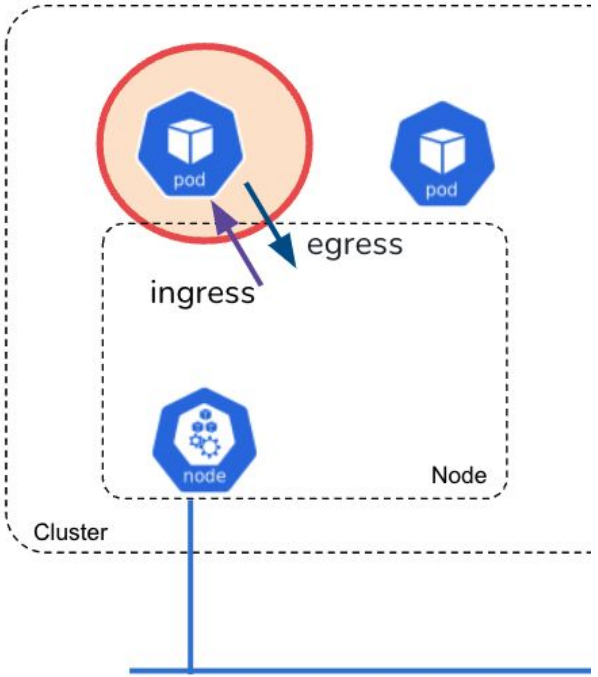
Agenda

- Le problème
- **Stratégie #1 : Egress Network Policy**
- Stratégie #2 : Placer les Pods sur certains noeuds (`Node Affinity`)
- Stratégie #3 : Utiliser IPAM pour discriminer les Pods
- Stratégie #4 : Utiliser des Egress Gateway

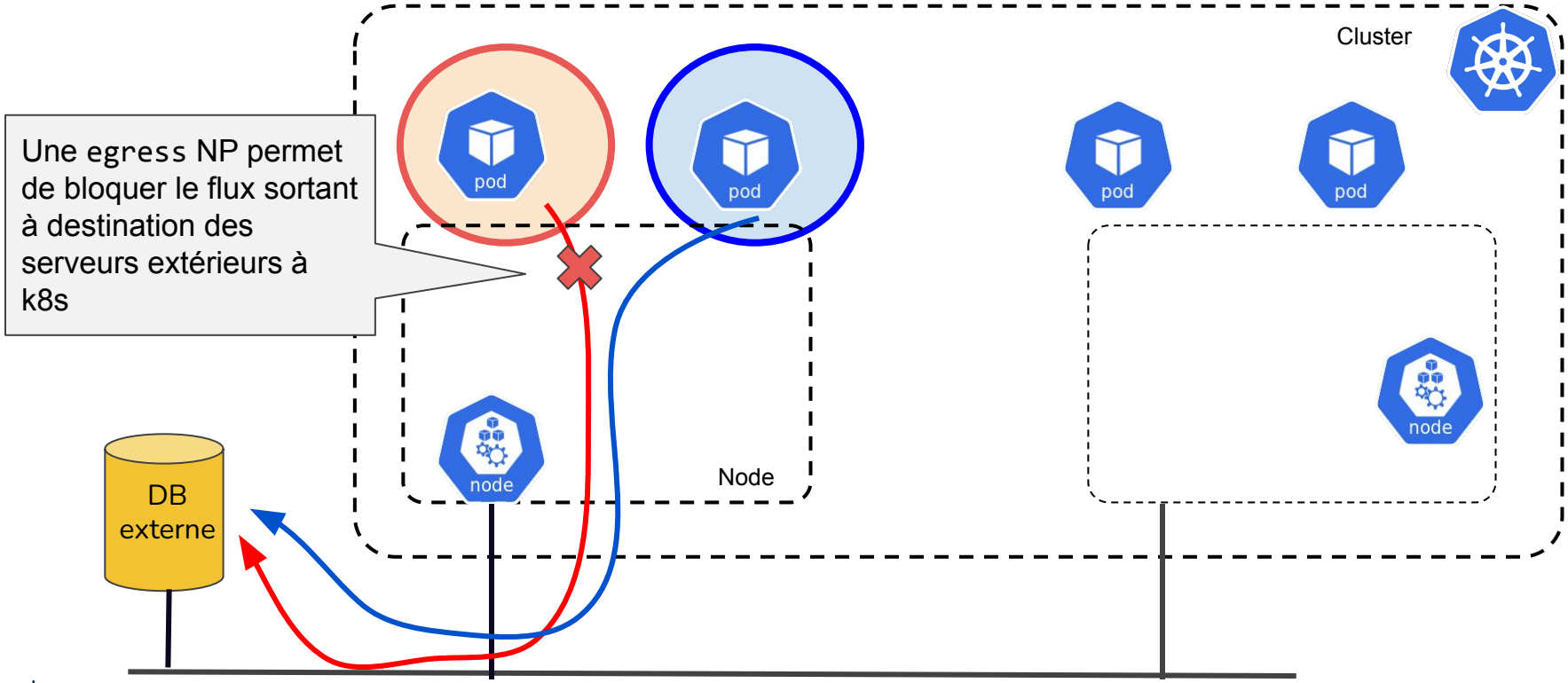
Philosophie

Les Network Policy (NP) sont dites “**application centric**” c’est-à-dire qu’elles s’appliquent directement sur les Pod (et non sur les flux en transit).

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



Cas d'usage des `egress network policy`



Cas d’usage des `egress network policy`

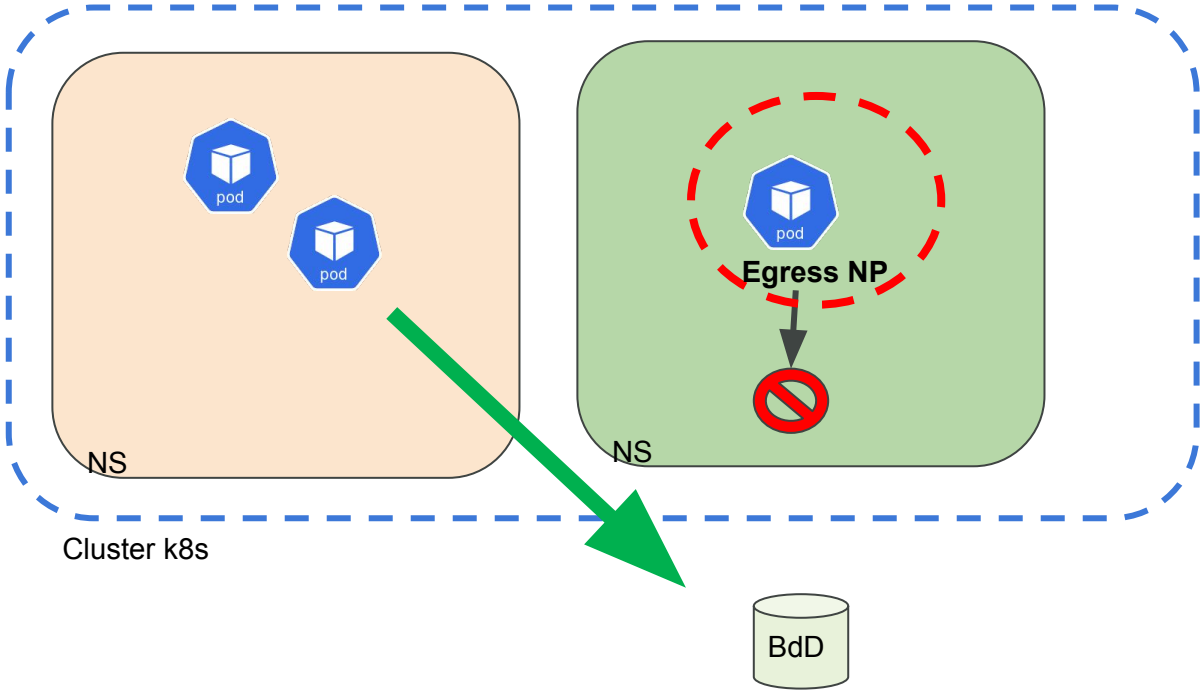
Seuls certains Pods ont le droit d’accéder à une BdD externe.

Il faut donc appliquer une `Network Policy` (NP) `Egress` à tous les `Pods` dans tous les `Namespaces` (NS)

La `NP Egress` standard nécessite de lister tous les usages permis (pas d’action DROP, uniquement du ALLOW implicite) : **tout ce qui n’est pas explicitement permis est bloqué***.

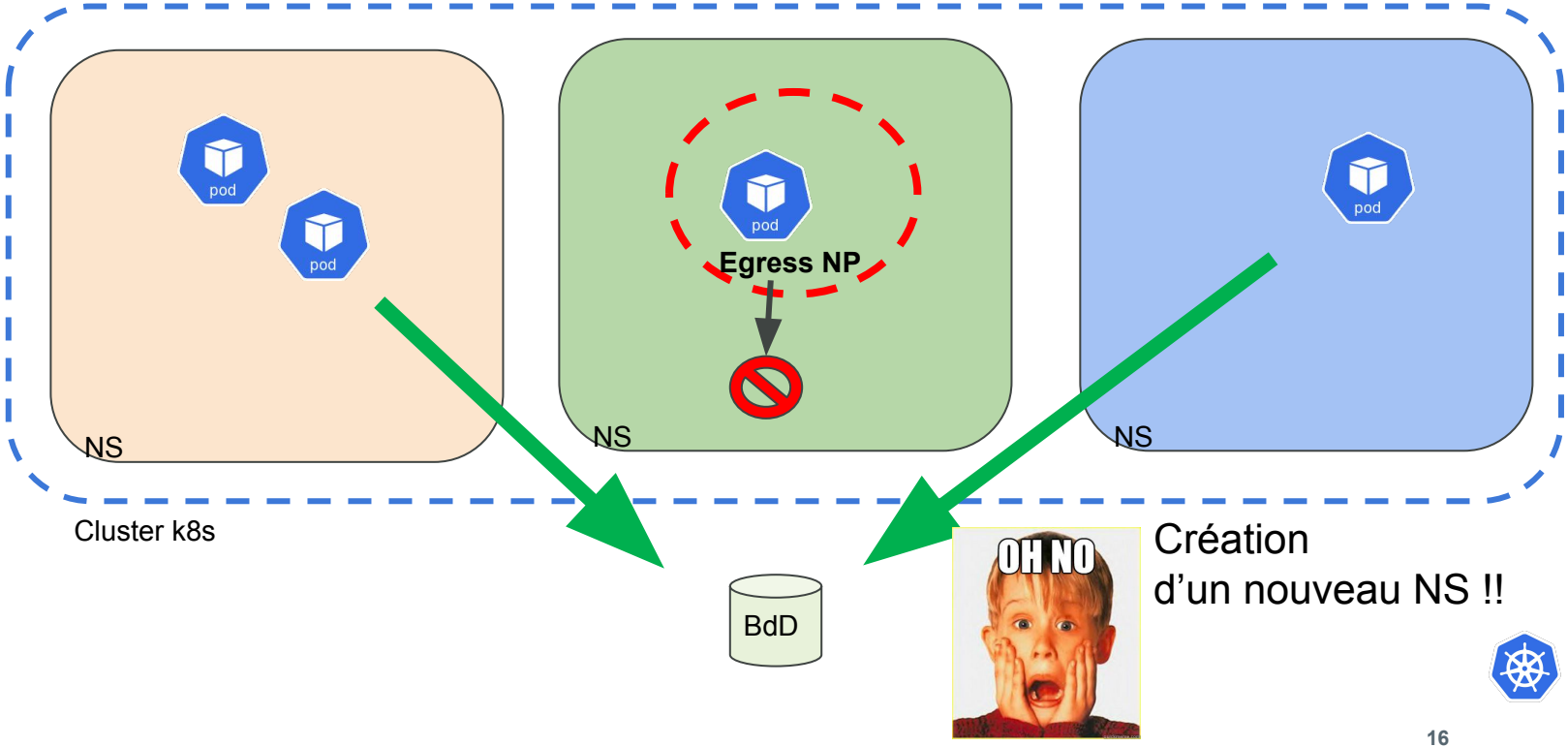
(*) Ceci est le cas, dès qu’une NP s’applique à un Pod. Si un Pod n’est concerné par aucune NP, tout son trafic est autorisé

Cas d'usage des `egress network policy`



Les `Network Policy` standard sont namespacées

Cas d’usage des `egress network policy`



Création
d’un nouveau NS !!

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

As of Kubernetes 1.23, 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).
- 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

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

Solution :enrichissement des Network Policy par un CNI Plugin Project Calico

The main features of Kubernetes network policies are:

- Policies are namespace scoped (i.e. you create them within the context of a specific namespace just like, for example, pods)
- Policies are applied to pods using label selectors
- Policy rules can specify the traffic that is allowed to/from other pods, namespaces, or CIDRs
- Policy rules can specify protocols (TCP, UDP, SCTP), named ports or port numbers

Kubernetes itself does not enforce network policies, and instead delegates their enforcement to network plugins. Most network plugins implement the mainline elements of Kubernetes network policies, though not all implement every feature of the specification. (Calico does implement every feature, and was the original reference implementation of Kubernetes network policies.)

To learn more about Kubernetes network policies, read the Get started with Kubernetes network policy guide.

Calico network policy

In addition to enforcing Kubernetes network policy, Calico supports its own namespaced NetworkPolicy and non-namespaced GlobalNetworkPolicy resources, which provide additional features beyond those supported by Kubernetes network policy. This includes support for:

- policy ordering/priority
- deny and log actions in rules
- more flexible match criteria for applying policies and in policy rules, including matching on Kubernetes ServiceAccounts, and (if using Istio & Envoy) cryptographic identity and layer 5-7 match criteria such as HTTP & gRPC URLs.
- ability to reference non-Kubernetes workloads in polices, including matching on NetworkSets in policy rules

While Kubernetes network policy applies only to pods, Calico network policy can be applied to multiple types of endpoints including pods, VMs, and host interfaces.

https://docs.projectcalico.org/about/about-network-policy

Stratégie #1 : Egress `Global Network Policy`

Seuls certains pods doivent accéder à une BdD externe. Il faut donc appliquer un NP Egress à **tous** les Pods dans **tous** les NS existants.

Plus tard, un nouveau NS est crée.

Les pods de ce nouveau NS ont accès à la BdD

Solution : les `Global Network Policy` (GNP) avec un Deny

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
 name: denyone
spec:
 order: 10
 namespaceSelector:
kubernetes.io/metadata.name not in {'rose'}
 types:
   - Egress
 egress:
   - action: Deny
     protocol: TCP
     destination:
       nets:
       - 10.2.3.4/32
       ports:
       - 80
```

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

Stratégie #1 : Egress `Network Policy`

Wait !!

Je veux que les autres destinations soient accessibles en HTTP(s), mais conserver ma règle en Deny….

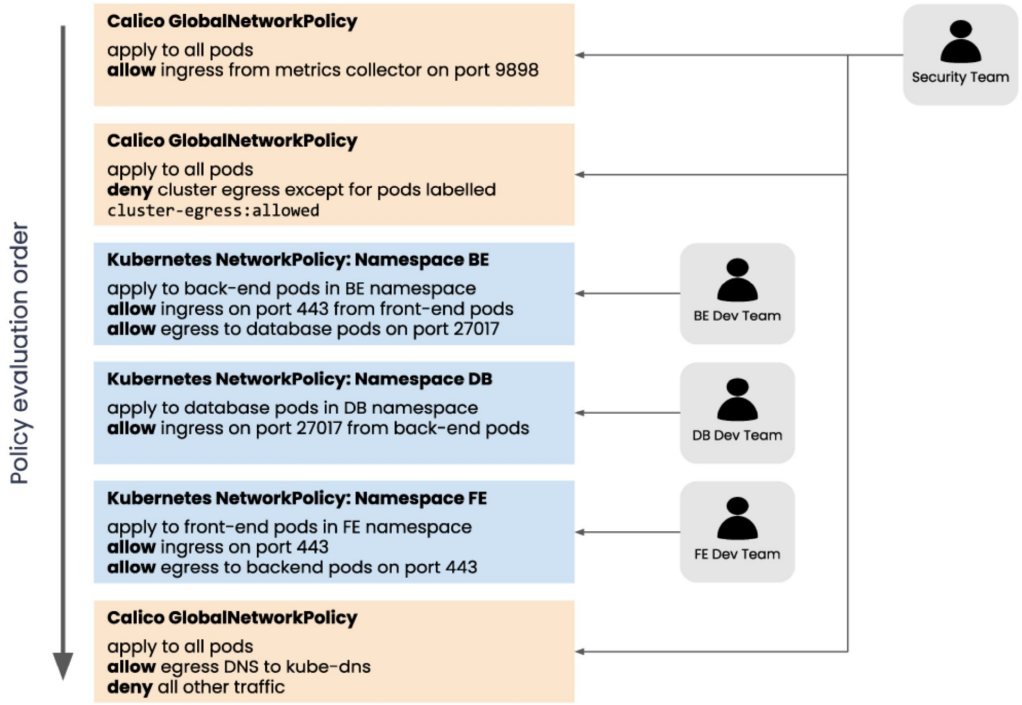
10	Deny HTTP vers 10.2.3.4
20	Allow HTTP + HTTPS
30	Deny All

Solution : Utiliser les “`order`” dans GNP

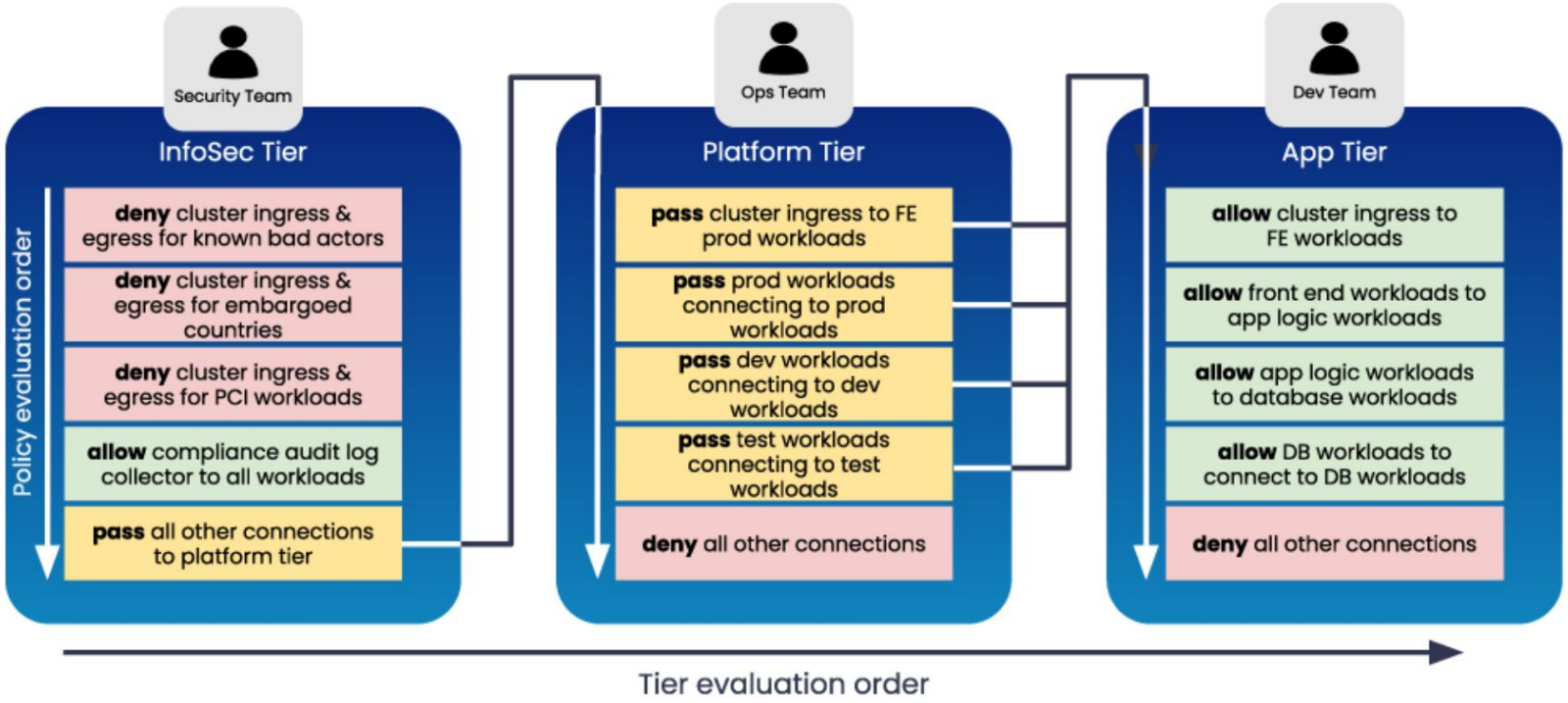
```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
 name: allowhttp
spec:
 order: 20
 types:
   - Egress
 egress:
   - action: Allow
     protocol: TCP
     source: {}
     destination:
       ports:
       - 80
       - 443
```

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

Cas d'usage : Un ordre dans l'évaluation des NP émerge..



Cas d'usage : Chaque équipe peut ajouter ses règles (soumis à RBAC)



NB: L'action PASS est l'équivalent d'un GOTO

Stratégie #1 : Egress Network Policy

✔ Sécurité maximum

✔ Adapté à toutes les topologies

✔ Résout tous les cas notamment les politiques de filtrage complexes intra-cluster

✘ Gestion lourde

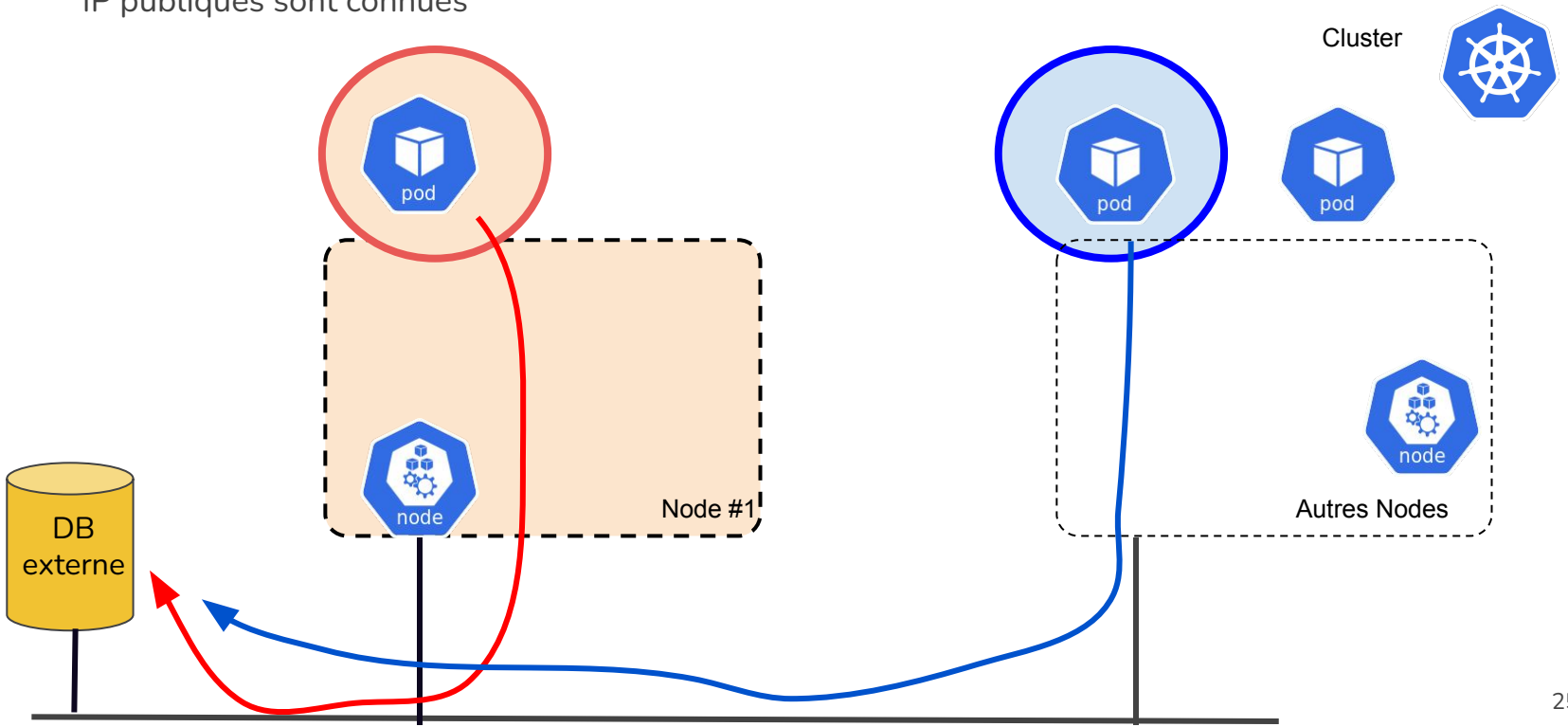
✘ Nécessite de façon réaliste le recours à des extensions non standard du plugin CNI

Agenda

- Le problème
- Stratégie #1 : Egress Network Policy
- **Stratégie #2 : Placer les Pods sur certains noeuds (`Node Affinity`)**
- Stratégie #3 : Utiliser IPAM pour discriminer les Pods
- Stratégie #4 : Utiliser des Egress Gateway

Stratégie #2 : Node Affinity

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



Stratégie #2 : Node Affinity

```
kubectl label node $NODENAME zone="dmz" --overwrite
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
      nodeSelector:
        zone: dmz
```

Stratégie #2 : Node Affinity

✔ Simplicté

✘ moindre optimisation du placement des Pods

✘ applicable uniquement en mode Tunnel/Encapsulation

✘ les Nodes n’ont pas vocation à être éternels

Agenda

- Le problème
- Stratégie #1 : Egress Network Policy
- Stratégie #2 : Placer les Pods sur certains noeuds (`Node Affinity`)
- **Stratégie #3 : Utiliser IPAM pour discriminer les Pods**
- Stratégie #4 : Utiliser des Egress Gateway

Stratégie #3 : IPAM et Pod

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
after-68c99f9f66-8q26s	1/1	Running	0	10s	10.243.0.90	aks-defaultpool-35447537-vmss000000
after-68c99f9f66-dxg8b	1/1	Running	0	10s	10.243.0.89	aks-defaultpool-35447537-vmss000000
after-68c99f9f66-x24nc	1/1	Running	0	10s	10.243.0.9	aks-defaultpool-35447537-vmss000001
before-76c6997d77-jtptf	1/1	Running	0	19s	10.244.72.12	aks-defaultpool-35447537-vmss000001
before-76c6997d77-stnnv	1/1	Running	0	19s	10.244.194.197	aks-defaultpool-35447537-vmss000000
before-76c6997d77-zqgxn	1/1	Running	0	19s	10.244.72.13	aks-defaultpool-35447537-vmss000001
pod-242	1/1	Running	0	4s	10.242.0.8	aks-defaultpool-35447537-vmss000001

Dans un architecture sans SNAT (TransparentMode, 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.

Stratégie #3 : IPAM et Pod

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-242
  annotations:
    cni.projectcalico.org/ipv4pools: '["242-pool"]'
spec:
  containers:
  - name: nginx
    image: nginx
```

```yaml
apiVersion:
projectcalico.org/v3
kind: IPPool
metadata:
  name: 243-pool
spec:
  cidr: 10.243.0.0/24
  blockSize: 29
  ipipMode: Always
  natOutgoing: true
  nodeSelector: !all()
```

Stratégie #3 : IPAM et Pod

✔ Simple et élégant coté k8s

✘ Seul le mode Transparent/Native Routing est pertinent pour cette stratégie ; en mode 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.

Agenda

- Le problème
- Stratégie #1 : Egress Network Policy
- Stratégie #2 : Placer les Pods sur certains noeuds (`Node Affinity`)
- Stratégie #3 : Utiliser IPAM pour discriminer les Pods
- **Stratégie #4 : Utiliser des Egress Gateway**

Stratégie #4 : Egress gateway

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

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

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: egress-ippool-1
spec:
  cidr: 10.10.10.0/31
  blockSize: 31
  nodeSelector: "!all()"
```

On annote le namespace pour que ses Pods utilisent l'Egress Gateway "red"

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

Stratégie #4 : Egress Gateway

✔ Simple et élégant coté k8s

✔ Plus permanent que `NodeAffinity`

✘ Nécessite une topologie réseau supportée (AWS et OnPrem)

✘ Nécessite la version Entreprise de Calico

Conclusion

Les documents et liens seront disponibles sous https://github.com/srnfr/Episode5

En savoir plus sur l'offre Tigera : sécurité des images, IDS, capture dynamique de paquets … ?
👉sreytan@bluetrusty.com

Nous serons bien sûr présents à **Devops D-Day** à Marseille le 1er décembre.

BlueTrusty

Temps pour les questions…