



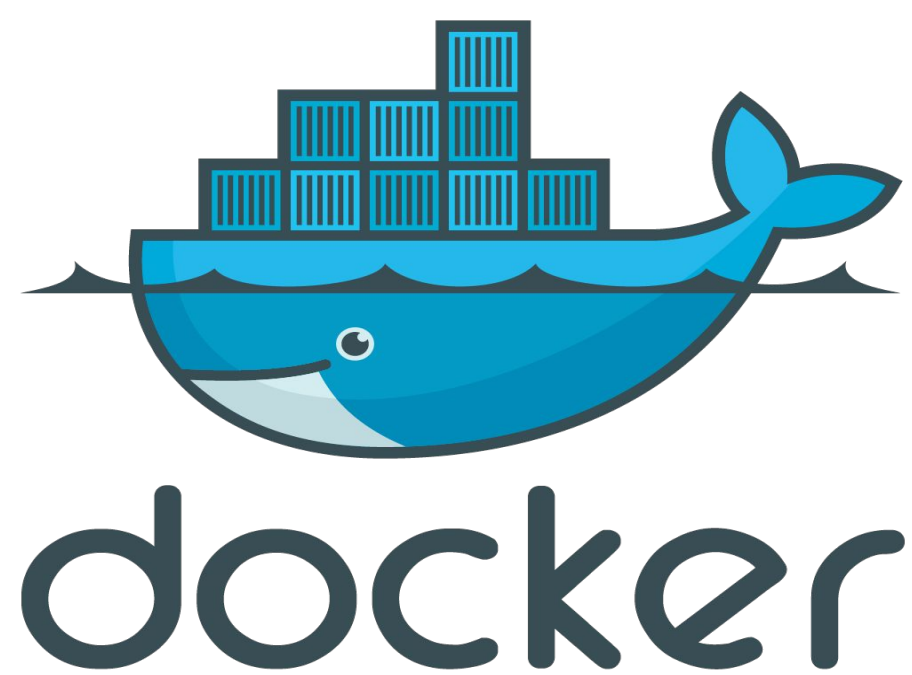
Samuel ANTUNES
Consultant Ingénieur DevSecOps
OCTO Technology
Email : contact@samuelantunes.fr

ICEBREAKER

shorturl.at/HMOP2

1. Les bases de Kubernetes
2. Manipulation simple de Kubernetes
3. Mettre son application dans K8s
4. Le Continuous Delivery avec K8s
5. Conclusion & Take Away

“ Les bases de Kubernetes ”



- D'anciens développeurs de Borg écrivent K8s en Go
- Directement pensé pour utiliser Docker (engine)
- Directement dans l'optique d'en faire un projet OpenSource
- Version 1.0 en Juin 2015

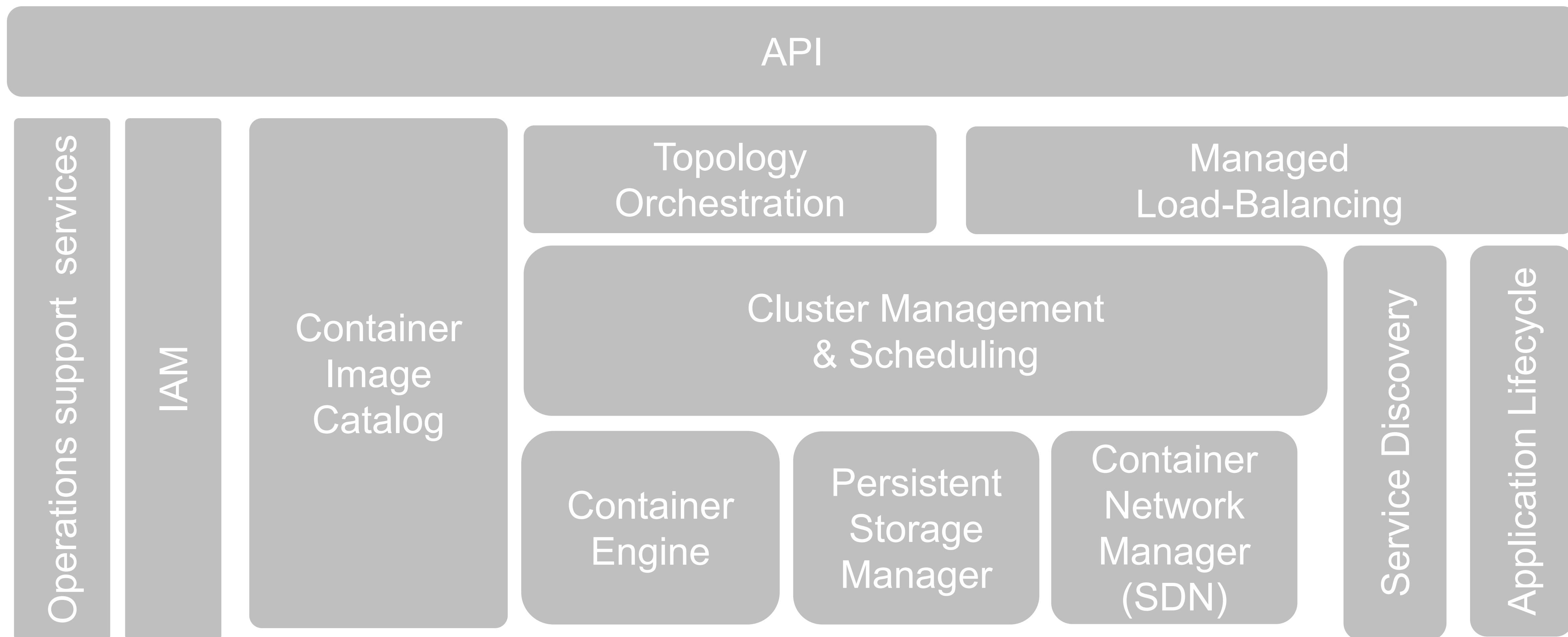


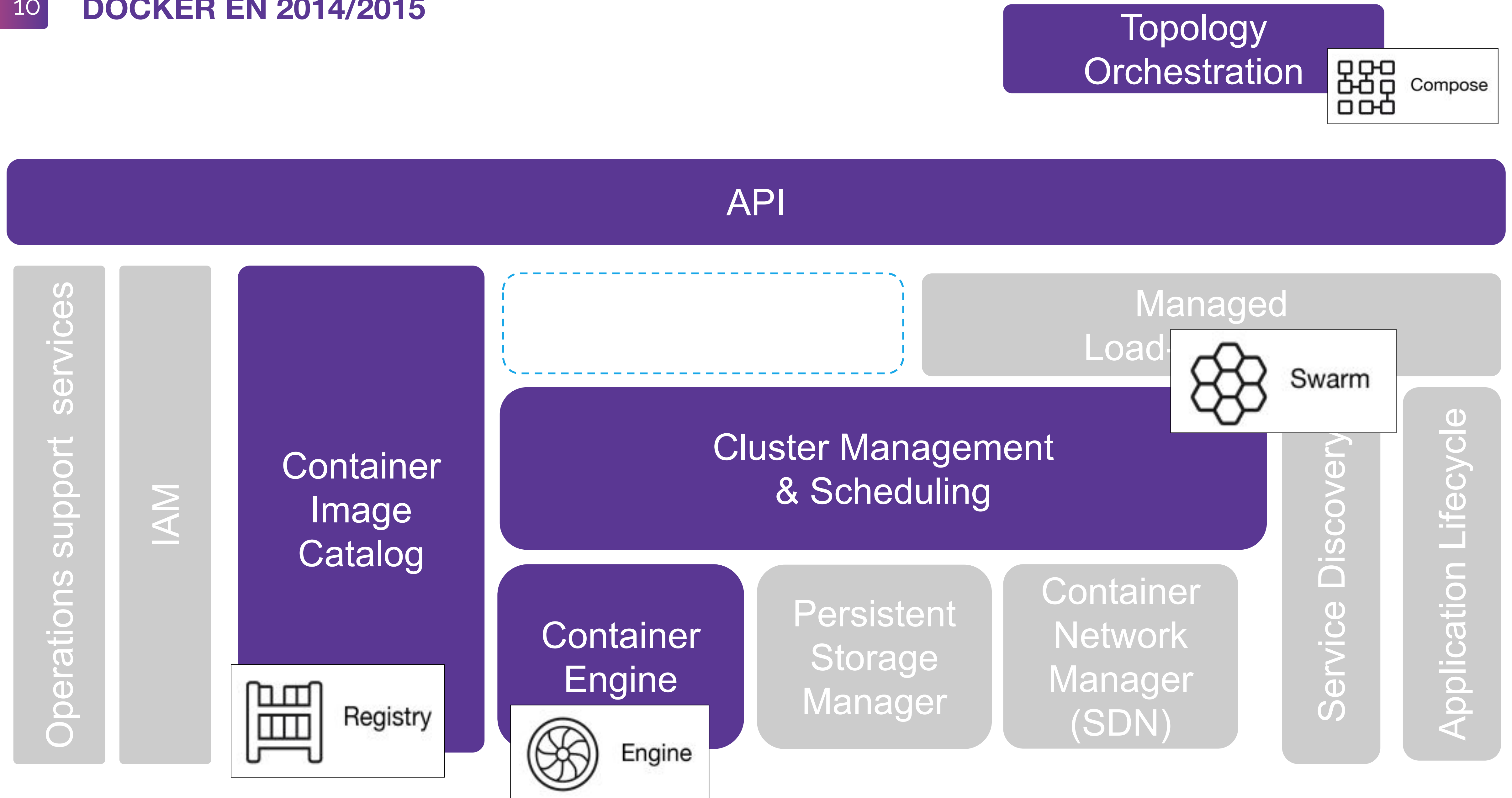
- Définir et déployer des applications multi-conteneurs
- Répartir les conteneurs sur une flotte d'hôtes (nœuds)
- Optimiser et adapter le placement des conteneurs
- Surveiller la santé des conteneurs
- Définir et appliquer des contraintes de niveaux de services
- Gérer la disponibilité et la scalabilité des conteneurs
- Gérer le provisionnement et l'accès au stockage
- Isoler les conteneurs
 - Limitation de ressources
 - Sécurité (vision multi-tenant)

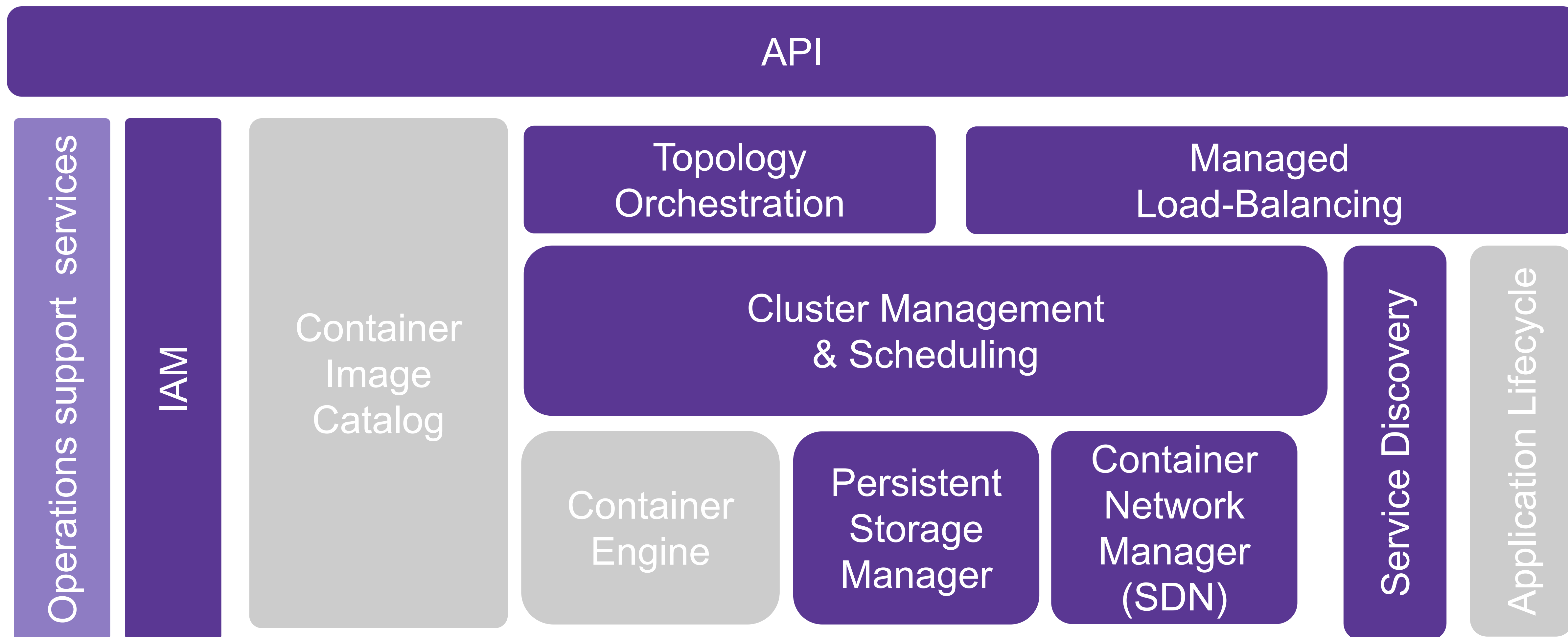


Tout ça de manière dynamique et pour des milliers de conteneurs !

- **Abstraction** des concepts
 - *Apparition de différents types de ressources de haut niveau*
 - *On ne manipule que très rarement la notion de conteneur directement*
- Approche **déclarative** plutôt que procédurale
 - *On décrit ce que l'on souhaite, pas comment l'obtenir*
 - *Notion de Desired State Configuration*







Comme Docker, K8s respecte ces différents principes d'architecture:

- Scalables horizontalement
- Immuables
- Sans état
- Share nothing
- Création et destruction facile, rapide et à faible coût

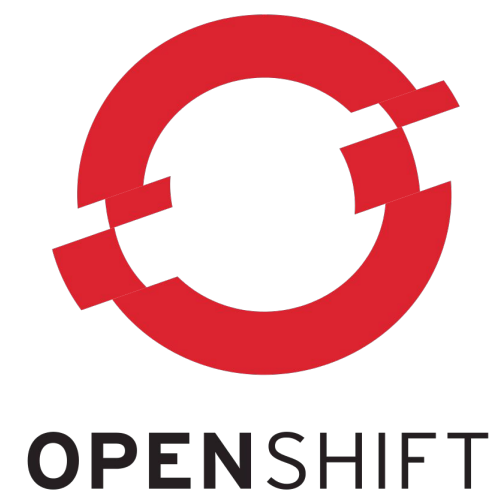


- On leur donne des noms (ex: monchaton)
- Ils sont uniques, on en prend soin
- Quand ils vont mal, on les chouchoute pour qu'ils aillent mieux

- On leur donne des numéros (ex: 3402)
- Elles se ressemblent toutes
- Quand elles sont malades, on les remplace



Amazon EC2
Container Service



Chez soi

- Directement sur des serveurs physiques (bare metal)
- Sur des VMs traditionnelles (VMWare)
- Sur un cloud privé (OpenStack)
- Sur son **laptop** (minikube / Docker Desktop)

Sur le cloud (avec des capacités d'intégration avancées : LB, Volumes...)

- Amazon Web Services (VM ou EKS)
- Google Cloud Platform (VM ou GKE)
- Azure (VM ou AKS)
- Chez beaucoup d'autres fournisseurs de clusters Kubernetes managés

Des outils connexes au projet Kubernetes sont là pour aider les déploiements (kops, kubeadm...)

<https://kubernetes.io/fr/docs/tasks/tools/install-minikube/#installer-minikube>

Pour vérifier la bonne installation des différents outils, il faut que vous lanciez les commandes suivantes et ayez un résultat similaire à l'image ci-dessous :

- `kubectl version --client`
- `minikube version`

```
samuel.antunes@AMAC02ZV0ZQMD6V ➤ ~/perso/formations/Kubernetes ➤ kubectl version --client
Client Version: version.Info{Major:"1", Minor:"16+", GitVersion:"v1.16.6-beta.0", GitCommit:"e7f962ba86f4ce7033828210ca3556393c377bcc", GitTreeState:"clean", BuildDate:"2020-01-15T08:26:26Z", GoVersion:"go1.13.5", Compiler:"gc", Platform:"darwin/amd64"}
samuel.antunes@AMAC02ZV0ZQMD6V ➤ ~/perso/formations/Kubernetes ➤ minikube version
minikube version: v1.13.0
commit: eeb05350f8ba6ff3a12791fcce350c131cb2ff44
samuel.antunes@AMAC02ZV0ZQMD6V ➤ ~/perso/formations/Kubernetes ➤
```

On peut à présent démarrer notre minikube avec la commande :

- minikube start

A savoir que la première fois il est demandé de choisir un driver, référez-vous à ce lien pour choisir un driver en fonction de votre environnement. On utilise généralement "docker" par défaut : `minikube start --driver="docker"`

<https://minikube.sigs.k8s.io/docs/drivers/>

```
samuel.antunes@AMAC02ZV0ZQMD6V ~/perso/formations/Kubernetes$ minikube start
🌟 minikube v1.13.0 on Darwin 10.15.7
🌟 Using the docker driver based on existing profile
🚫 Requested memory allocation (1990MB) is less than the recommended minimum 2000MB. Deployments may fail.
👍 Starting control plane node minikube in cluster minikube
🔄 Restarting existing docker container for "minikube" ...
🔧 Preparing Kubernetes v1.19.0 on Docker 19.03.8 ...
🔍 Verifying Kubernetes components...
🌟 Enabled addons: default-storageclass, storage-provisioner
❗ /usr/local/bin/kubectl is version 1.16.6-beta.0, which may have incompatibilities with Kubernetes 1.19.0.
💡 Want kubectl v1.19.0? Try 'minikube kubectl -- get pods -A'
👉 Done! kubectl is now configured to use "minikube" by default
```

Avec l'API

- Interaction programmatique pour manipuler les ressources
- Approche déclarative de l'État Attendu (Desired State)

Avec un SDK (Golang, Python...) qui s'appuie sur l'API

- Terraform, Ansible

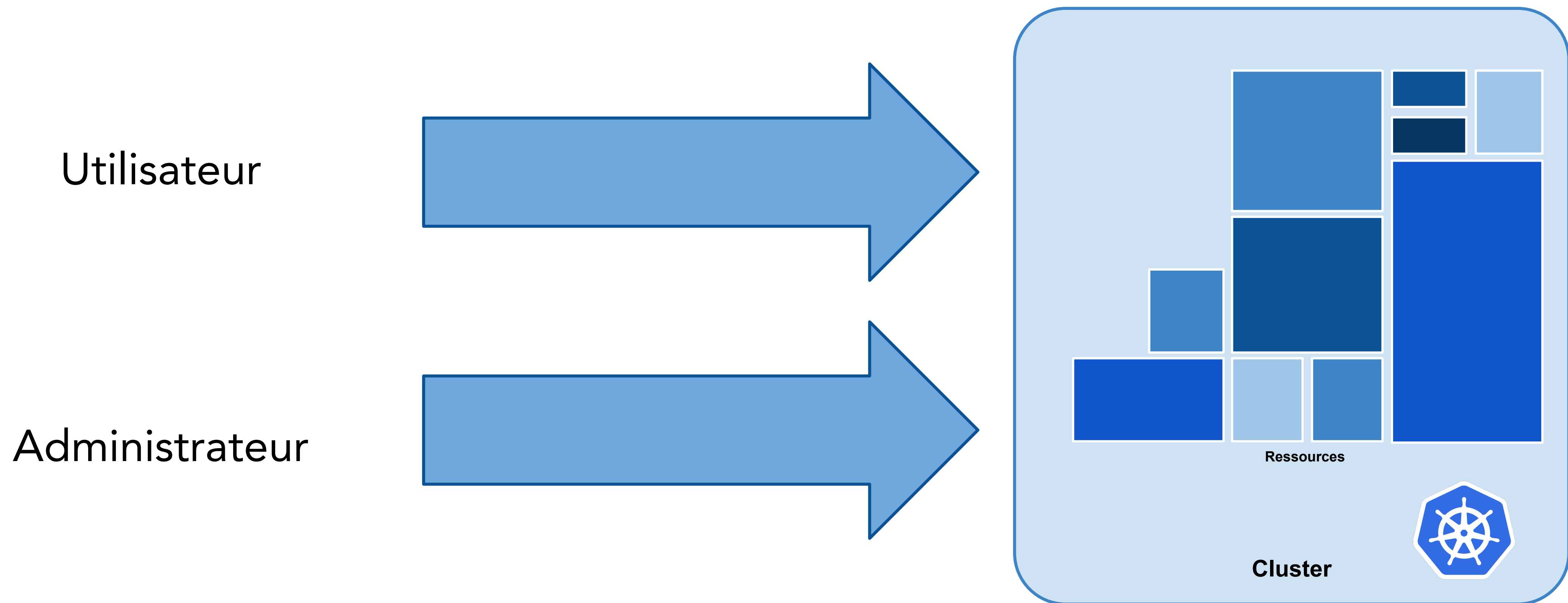
Avec le CLI

- Binaire kubectl

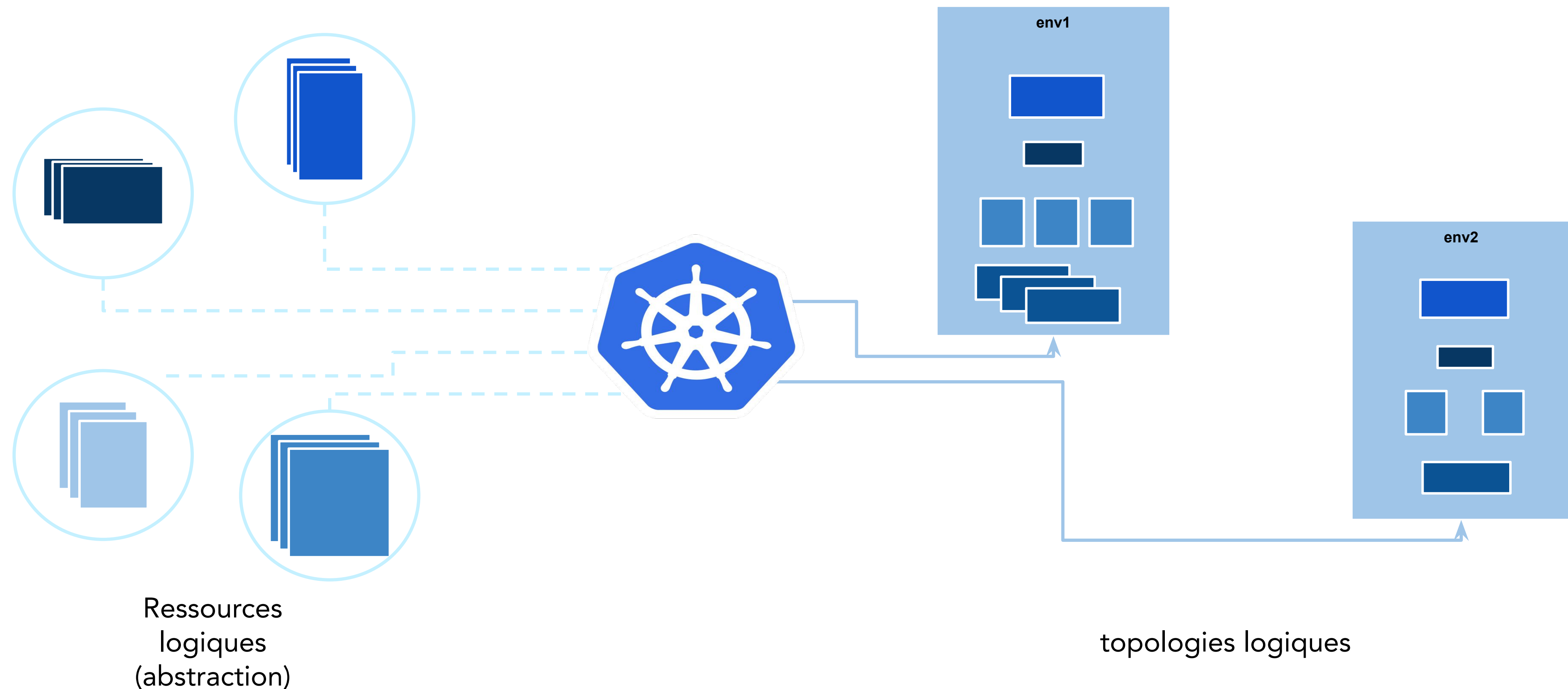
Avec le(s) Dashboard(s)

- Visualisation simplifiée des ressources présentes dans le cluster
- Pas complet, les ressources récentes de Kubernetes n'apparaissent pas en visualisation

Une ressource au sens Kubernetes représente un concept logique manipulé dans Kubernetes. Il en existe plus d'une 20aine et certaines sont réservées aux administrateurs.



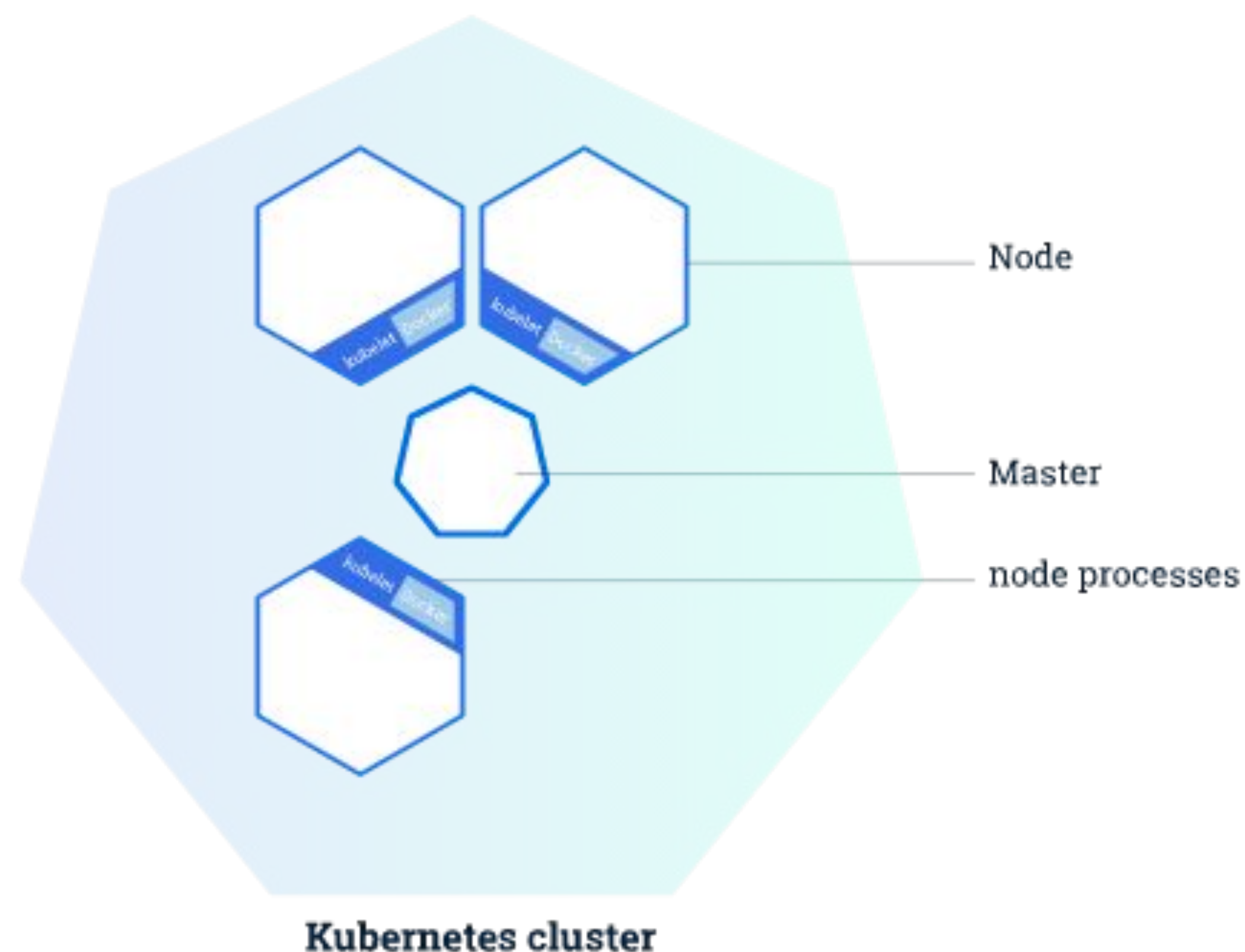
C'est donc avec des concepts logiques que nous allons pouvoir configurer et interagir avec notre cluster Kubernetes (ici minikube).
Concrètement qu'est-ce que ça veut dire ?

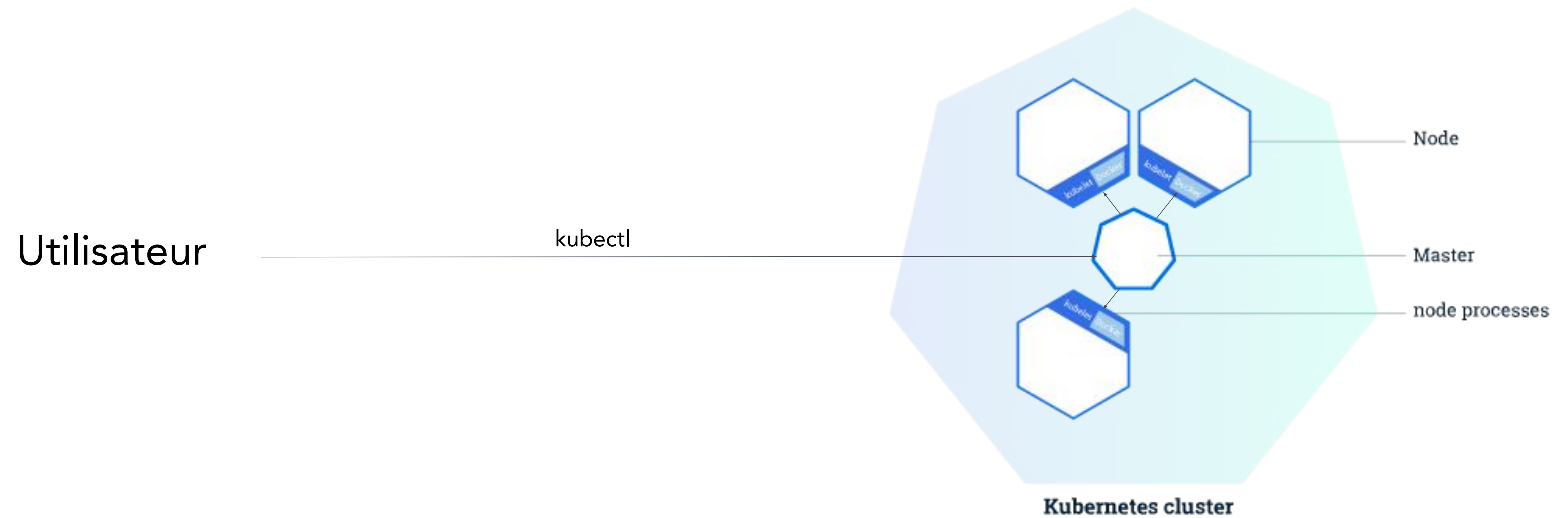


La première ressource que nous allons découvrir est le "node". Il y en a de deux types, un node dit "master" et des nodes "slaves".

Le master contrôle le cluster à travers des agents posés sur les noeuds slaves : le Kubelet.

Un node au sens propre du terme peut être une machine virtuelle, physique ou bien un conteneur docker.





C'est un exécutable binaire écrit en Go qui existe pour la plupart des plateformes classiques (Windows, Linux, MacOSX)

Le client kubectl suit la même numérotation de version que la partie serveur

Pour connaître la dernière version stable :

<https://storage.googleapis.com/kubernetes-release/release/stable.txt>

Lister toutes les ressources d'un type

```
$ kubectl get (type1|type2|type3|...)
```

Lister une ou des ressources spécifique(s)

```
$ kubectl get type1/nom-ressource1 type2/nom-ressource2  
$ kubectl get type3 nom-ressource3
```

Pour avoir plus de détails

```
$ kubectl get type1/nom-ressource -o wide  
$ kubectl get type1/nom-ressource -o (yaml|json)  
$ kubectl describe type8/nom-ressource
```

```
json
yaml
wide
custom-columns=...
custom-columns-file=...
[go-]template=...
[go-]template-file=...
jsonpath=...
jsonpath-file=...
```

```
$ kubectl get nodes/minikube -o=go-template \
  --template="{{.metadata.name}} est sous {{.status.nodeInfo.operatingSystem}}"
minikube est sous Linux
```

```
$ kubectl get nodes \
  -o=custom-columns=NAME:.metadata.name,CPU:.status.allocatable.cpu
NAME      CPU
minikube  2
```

Le mode automatique

```
$ kubectl run [plein d'options]
```

Pour faire le ménage

```
$ kubectl delete (type1|type2|type3|...) NAME  
$ kubectl delete type6/NAME  
$ kubectl delete type8 --all
```

Vous aurez dans ce repo un fichier TP1.md à lire et réaliser :

git clone https://token_kubernetes:eWNyr6S5QGzz8TayhMB4@gitlab.com/santunes-formations/kubernetes.git

Vous l'aurez compris, Kubectl est un outil puissant permettant de faire énormément de choses et nous en avons vu là qu'une infime partie.

PS: Les plus énervés peuvent tout faire avec des curls ou en développant leur propre client K8s...

Un seul fichier de configuration kubeconfig (par défaut .kube/config) permet de décrire plusieurs contextes d'exécution

- Sur quel cluster se connecter
- En tant que quel utilisateur et avec quelle méthode d'authentification
 - Login / mot de passe
 - Token
 - Certificat client + sa clé privée

On peut choisir le contexte (--context)

On peut surcharger le cluster (--cluster)

On peut surcharger l'utilisateur (--user)

Il y a un contexte courant, actif par défaut (current-context, use-context)


```
$ kubectl config current-context  
minikube
```

```
$ kubectl config get-contexts  
NAME  
cluster-dev  
minikube
```

```
$ kubectl config use-context cluster-dev  
Switched to context "cluster-dev".
```

contexte cluster-dev

```
apiVersion: v1
kind: Config
clusters:
- name: minikube
  cluster:
    certificate-authority: [snip]/ca.crt
    server: https://192.168.42.209:8443
- name: cluster-dev
  cluster:
    certificate-authority-data: [snip]
    server: https://35.187.116.50
users:
- name: minikube
  user:
    client-certificate: [snip]/client.crt
    client-key: [snip]/client.key
- name: cluster-dev-spc
  user:
    token: [snip]
contexts:
- name: minikube
  context:
    cluster: minikube
    user: minikube
- name: cluster-dev
  context:
    cluster: cluster-dev
    user: cluster-dev-spc
current-context: minikube
```

contexte minikube

“ Manip simple de
Kubernetes ”

KUBECTL - LES TYPES DE RESSOURCE DANS K8S (spoiler, il y en a beaucoup)

```
$ kubectl api-resources
```

```
You must specify the type of resource to get. Valid resource types include:
```

```
* all
* certificatesigningrequests (aka 'csr')
* clusterrolebindings
* clusterroles
* clusters (valid only for federation apiservers)
* componentstatuses (aka 'cs')
* configmaps (aka 'cm')
* controllerrevisions
* cronjobs
* customresourcedefinition (aka 'crd')
* daemonsets (aka 'ds')
* deployments (aka 'deploy')
* endpoints (aka 'ep')
* events (aka 'ev')
* horizontalpodautoscalers (aka 'hpa')
* ingresses (aka 'ing')
* jobs
* limitranges (aka 'limits')
* namespaces (aka 'ns')
* networkpolicies (aka 'netpol')
* nodes (aka 'no')
* persistentvolumeclaims (aka 'pvc')
* persistentvolumes (aka 'pv')
* poddisruptionbudgets (aka 'pdb')
* podpreset
* pods (aka 'po')
* podsecuritypolicies (aka 'psp')
* podtemplates
* replicaset (aka 'rs')
* replicationcontrollers (aka 'rc')
* resourcequotas (aka 'quota')
* rolebindings
* roles
* secrets
* serviceaccounts (aka 'sa')
* services (aka 'svc')
* statefulsets
* storageclasses
```

```
error: Required resource not specified
Use "kubectl explain <resource>" for a detailed description of that resource (e.g.
```

```
* all
```

vrai nom

alias (pour les paresseux)

```
* namespaces (aka 'ns')
```

```
* nodes (aka 'no')
```

Use "[kubectl explain <resource>](#)" for a detailed description of that resource (e.g. `kubectl explain pods`).

See '[kubectl get -h](#)' for help and examples....

KUBECTL - LES TYPES DE RESSOURCE DANS K8S (spoiler, il y en a beaucoup)

```
$ kubectl api-resources
```

NAME	SHORTNAMES	APIGROUP	NAMESPACED	KIND
configmaps	cm		true	ConfigMap
namespaces	ns		false	Namespace
nodes	no		false	Node
persistentvolumes	pv		false	
PersistentVolume				
Pods	po		true	Pod
...				

alias -> kubectl get po

Indique si la ressource doit être placée dans un namespace

```
$ kubectl explain pod
KIND:      Pod
VERSION:   v1

DESCRIPTION:
  Pod is a collection of containers that can run on a host. This resource is
  created by clients and scheduled onto hosts.

FIELDS:
  apiVersion  <string>
    APIVersion defines the versioned schema of this representation of an
    object. Servers should convert recognized schemas to the latest internal
    value, and may reject unrecognized values. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#resources

  kind <string>
    Kind is a string value representing the REST resource this object
    represents. Servers may infer this from the endpoint the client submits
    requests to. Cannot be updated. In CamelCase. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#types-kinds

  metadata <Object>
    Standard object's metadata. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#metadata

  spec <Object>
    Specification of the desired behavior of the pod. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status

  status  <Object>
    Most recently observed status of the pod. This data may not be up to date.
    Populated by the system. Read-only. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status
```

QUELQUES GENERALITES

Dans Kubernetes, (presque) **toutes les ressources** sont structurées de façon identique

```
apiVersion: VERSION_API_RESSOURCE
kind: TYPE_DE_RESSOURCE
metadata:
  name: NOM_DE_LA_RESSOURCE
  ...
  ...
  ...
spec:
  ...
  ...
  ...
  ...
status:
  ...
  ...
  ...
  ...
  ...
```

Exemple : v1

Exemple : Node

Exemple : minikube

État attendu de la ressource

État actuel de la ressource dans le cluster

```
$ kubectl explain NOM_DE_LA_RESSOURCE --recursive
```

permet d'obtenir l'ensemble des champs possibles pour la ressource donnée


```
$ kubectl explain pod
KIND:      Pod
VERSION:   v1

DESCRIPTION:
  Pod is a collection of containers that can run on a host. This resource is
  created by clients and scheduled onto hosts.

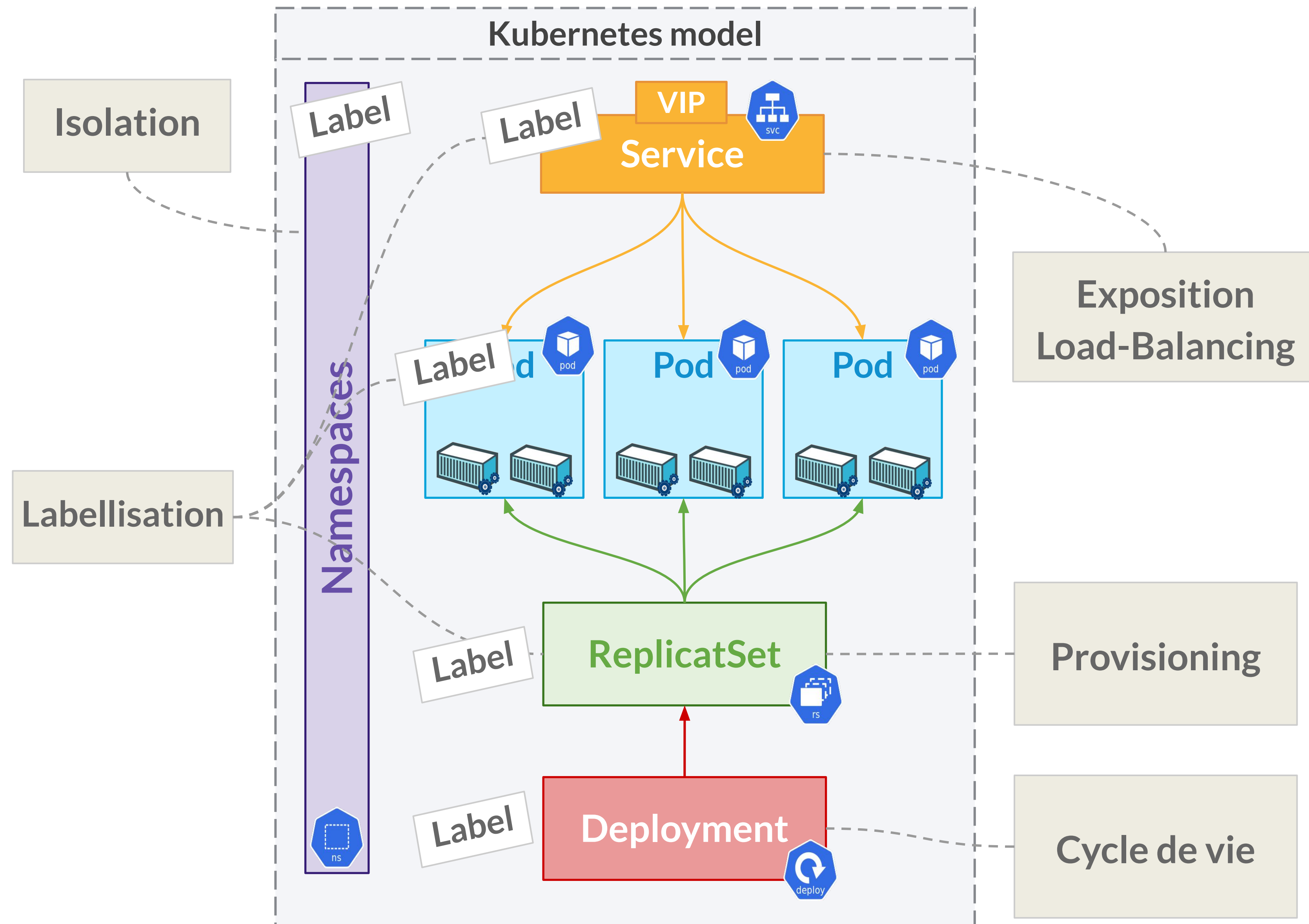
FIELDS:
  apiVersion  <string>
    APIVersion defines the versioned schema of this representation of an
    object. Servers should convert recognized schemas to the latest internal
    value, and may reject unrecognized values. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#resources

  kind <string>
    Kind is a string value representing the REST resource this object
    represents. Servers may infer this from the endpoint the client submits
    requests to. Cannot be updated. In CamelCase. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#types-kinds

  metadata <Object>
    Standard object's metadata. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#metadata

  spec <Object>
    Specification of the desired behavior of the pod. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status

  status  <Object>
    Most recently observed status of the pod. This data may not be up to date.
    Populated by the system. Read-only. More info:
    https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status
```



- ▷ Dans Kubernetes, **toutes les ressources créées sont labellisés et labellisables**
- ▷ Permet d'**associer** de manière souple et libre les **ressources Kubernetes** à des concepts
 - > Localisation (dc1, dc2, eu-west...)
 - > Environnements logiques (dev, qualif, prod...)
 - > Produits / projets (app1, app2...)
 - > Caractéristiques techniques (hdd, ssd...)
 - > Architecture (front, back...)
 - > Organisations (team1, team2...)
- ▷ Les labels sont **requêtables** au travers d'une syntaxe, appelé un ***selector***
- ▷ Les labels et les selectors sont énormément **utilisés** et **nécessaires** au fonctionnement de nombreuses fonctions (exemple : le load-balancing)

- ▷ Opérateur de type égalité

```
'disklabel!=ssd'
```

- ▷ Opérateur ensembliste

```
'region in (usa, europe)'
```

- ▷ Présence / absence d'un label (*Attention au ! et au SHELL, protection par quote*)

```
'my_label'  
'!is_production_ready'
```

- ▷ multi critères, séparés par une virgule (&& => tous doivent matcher)

```
'is_backend, dc in (dc1,dc2),disk_type=ssd'
```

- ▷ Il est possible de poser un label à la **création** des ressources, mais aussi de modifier les labels en **cours de vie**
- ▷ L'option `-l` de `kubectl get` permet d'appliquer un sélecteur sur les ressources pour les filtrer

```
$ kubectl label no/node1 region=eu-west-1 zone=eu-west-1a
node "node1" labeled

$ kubectl get no -l region=eu-west-2
No resources found.

$ kubectl get no -l region=eu-west-1
NAME      STATUS    ROLES    AGE      VERSION
node1     Ready     <none>   44m      v1.8.0

$ kubectl get no -l region=eu-west-1 \
  -o=custom-columns=NAME:.metadata.name,ZONE:.metadata.labels.zone
NAME      ZONE
node1     eu-west-1a
```

- ▷ L'écrasement d'un label doit être confirmé

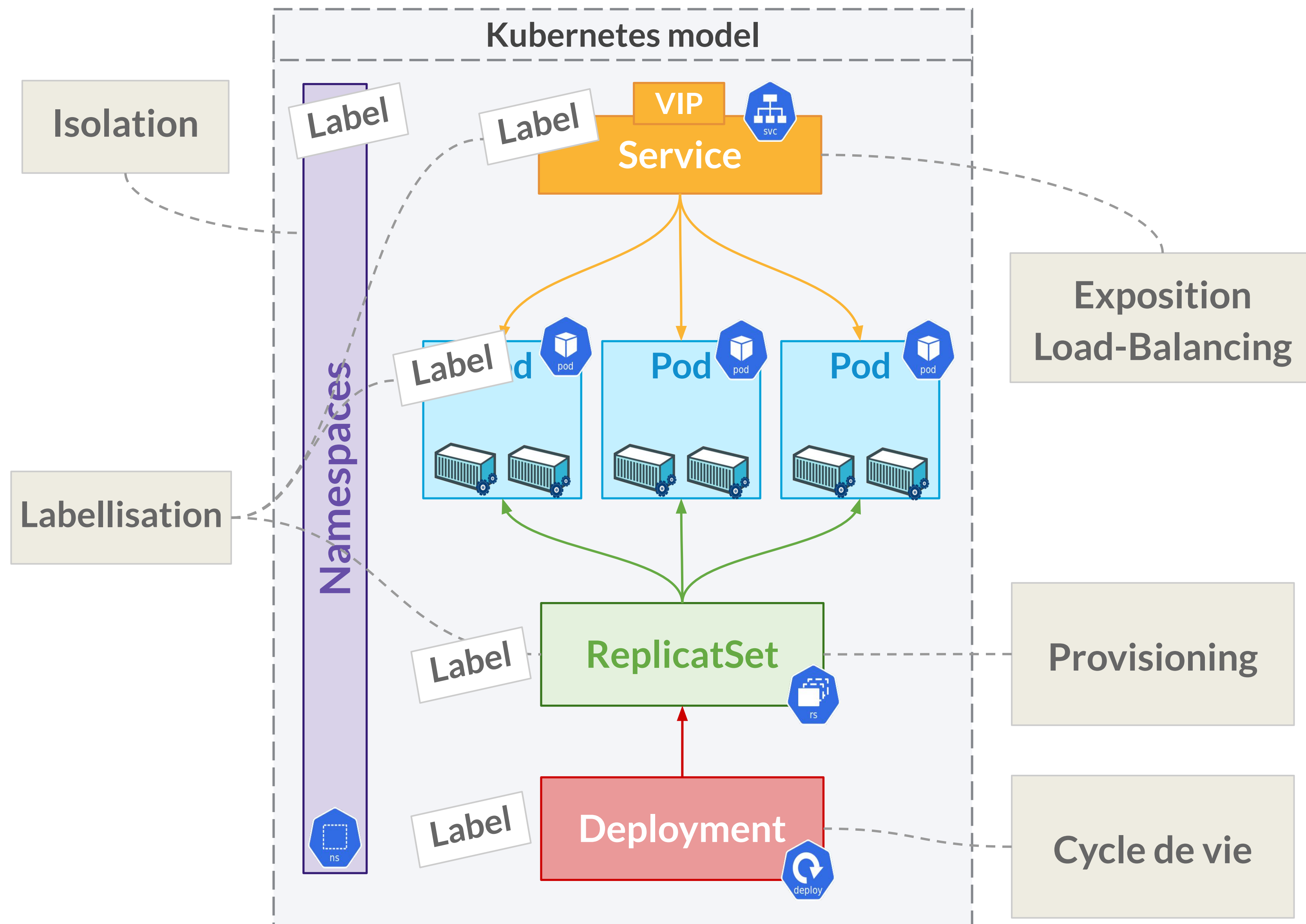
```
$  
error: 'truc' already has a value (machin), and --overwrite is false  
  
$ kubectl label no/minikube --overwrite truc=bidule  
node "minikube" labeled
```

- ▷ La suppression d'un label se fait en ajoutant un - à la fin du nom du label

```
$ kubectl label no/minikube truc-  
node "minikube" labeled  
  
$ kubectl get no/minikube -o template \  
--template='{{ .metadata.labels.truc }}'  
<no value>
```

- ▷ Le concept d'**annotations** est également présent sur tous les objets
- ▷ **Comme les labels**, elles permettent d'ajouter des informations descriptives aux ressources
- ▷ En général, on les utilise pour
 - > Activer des **fonctions expérimentales**
 - > Préciser des **comportements spécifiques** du cluster
 - > Tracer l'**historique** de certains changements sur les objets
- ▷ À la différence des labels, elles ne peuvent pas être utilisées pour filtrer les objets

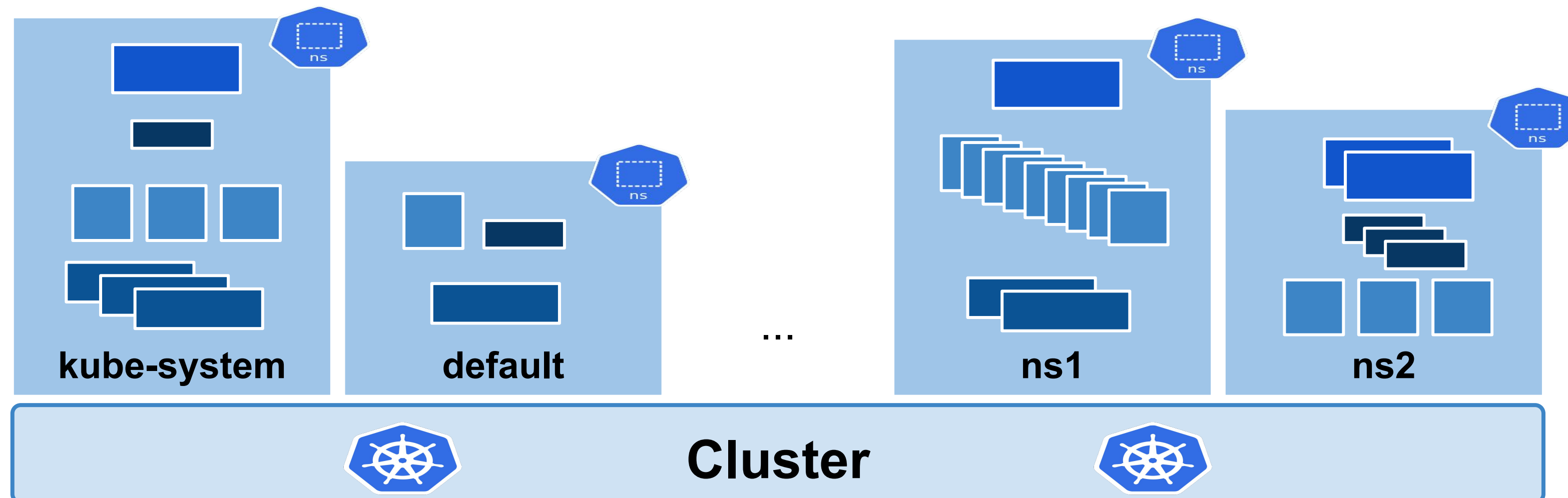
*Nous aurons l'occasion d'en reparler car le rôle des labels est **majeur** dans K8s...*

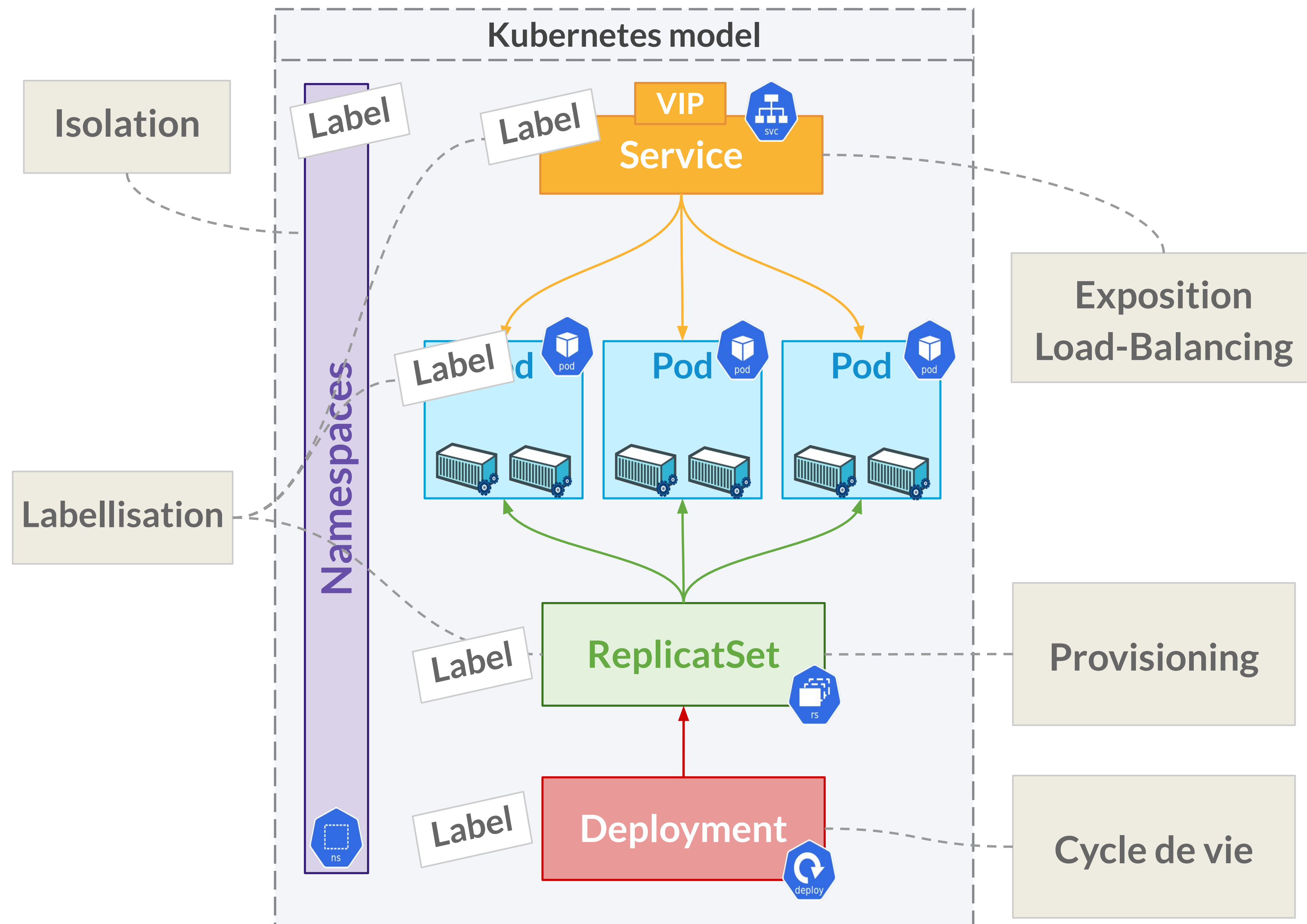


- ▷ *Définition* : l'unité de regroupement des ressources pour représenter des équipes / des projets, des environnements...
- ▷ C'est sur les *namespaces* que se positionnent les **limitations de ressources et quotas**
- ▷ Des objets de même nom dans deux *namespaces* différents ne sont **pas en conflit** et ne se voient pas directement
- ▷ **Les namespaces ne peuvent pas s'imbriquer**
- ▷ Les **contextes** kubeconfig permettent de fixer le namespace à utiliser **par défaut**
- ▷ Des **Règles** peuvent s'appliquer par namespace pour en **restreindre les accès**

LES NAMESPACES (ns)

- ▷ Même si ce n'est pas explicitement décrit, (presque) **toutes les ressources** sont dans **un** (et un seul) **namespace**
 - > Les **nodes** sont une des **exceptions**
 - > Supprimer un **ns** supprime les **ressources** qu'il contient
 - > Une ressource **ne peut pas être déplacée** d'un ns à un autre
- ▷ Dans un cluster Kubernetes, il existe généralement au moins trois namespaces
 - > **kube-public**
 - > **kube-system**
 - > **default**





- ▷ *Définition* : un ensemble de conteneurs ayant **un lien logique et une colocalisation**
- ▷ **Une abstraction supplémentaire** au-dessus des conteneurs
- ▷ **Objet éphémère**, se construit et se détruit à un coût négligeable
- ▷ **Les conteneurs d'un même Pod partagent des composants** comme le réseau (ex: même adresse IP, même boucle locale)
- ▷ La plupart du temps en pratique: **1 pod = 1 conteneur**
- ▷ En pratique K8s ajoute un conteneur **technique** dans chaque pod. Ce conteneur **technique** porte l'IP. Cette complexité est masquée à l'utilisateur.



Le **pod** est un objet très **technique** qui n'est que très rarement directement manipulé. D'**autres concepts** de plus haut niveau sont là pour le faire à notre place...



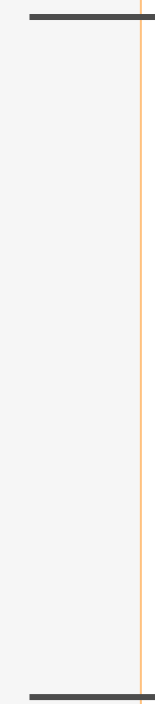
Le **pod** servira des applications très souvent **stateless**, et **sans notion de dépendances** (vis à vis de l'OS sous jacent). Ce que l'on retrouve dans les **2eme** et **6eme principes du twelve-factor app**

Exemple de pod co-localisant une application et un cache mémoire

C'est un exemple uniquement. Faire ça est une mauvaise idée...

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  containers:
  - image: node/app1:v1.0
    name: app1
  - image: memcached
    name: memcached
```

Liste des
conteneurs
du Pod

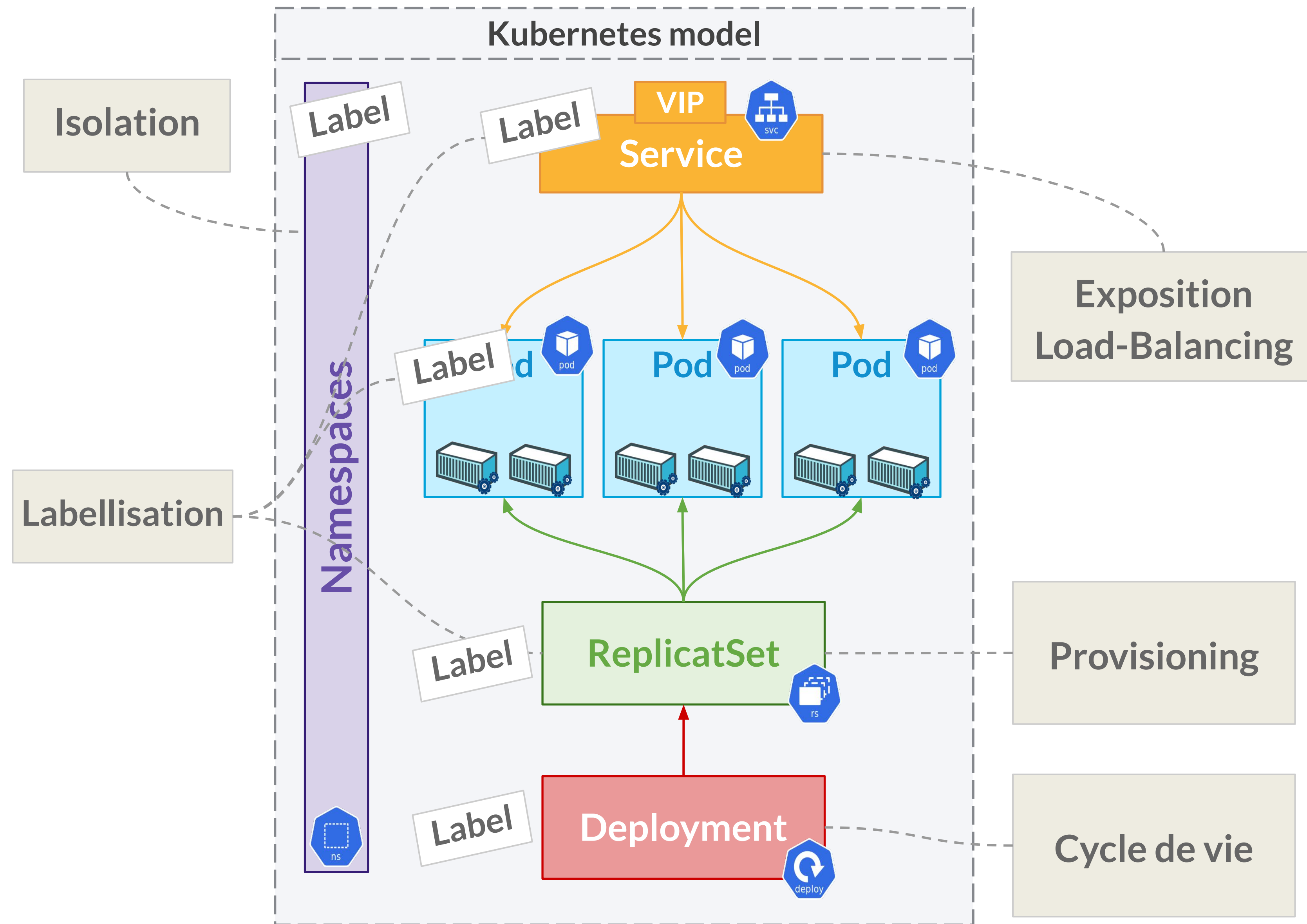



```
apiVersion: v1
kind: Pod
metadata:
  ...
spec:
  containers:
  - image: app:v1
    name: app
    env:
    - name: BIDULE
      value: machine
      ...
    ...
  restartPolicy: Always
  nodeSelector:
    disk: ssd
```

Injecter des variables
d'environnement dans mon
conteneur

- Always
- OnFailure
- Never

Règle de placement sur les
nœuds



LES SERVICES (svc)

- ▷ *Définition* : une interface **nommée** permettant d'**accéder à un groupe de pods**
- ▷ Le service sert à
 - > **Nommer un groupe de conteneurs**
 - > **Agir en tant que Load-Balancer** devant des Pods
- ▷ Il est **accessible depuis tous les pods du même namespace** par son nom **DNS court** (nginx-svc)

Exemple d'un service nginx

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    run: nginx
```

Port exposé
par le *service*

Choix des *pods*
du *service*

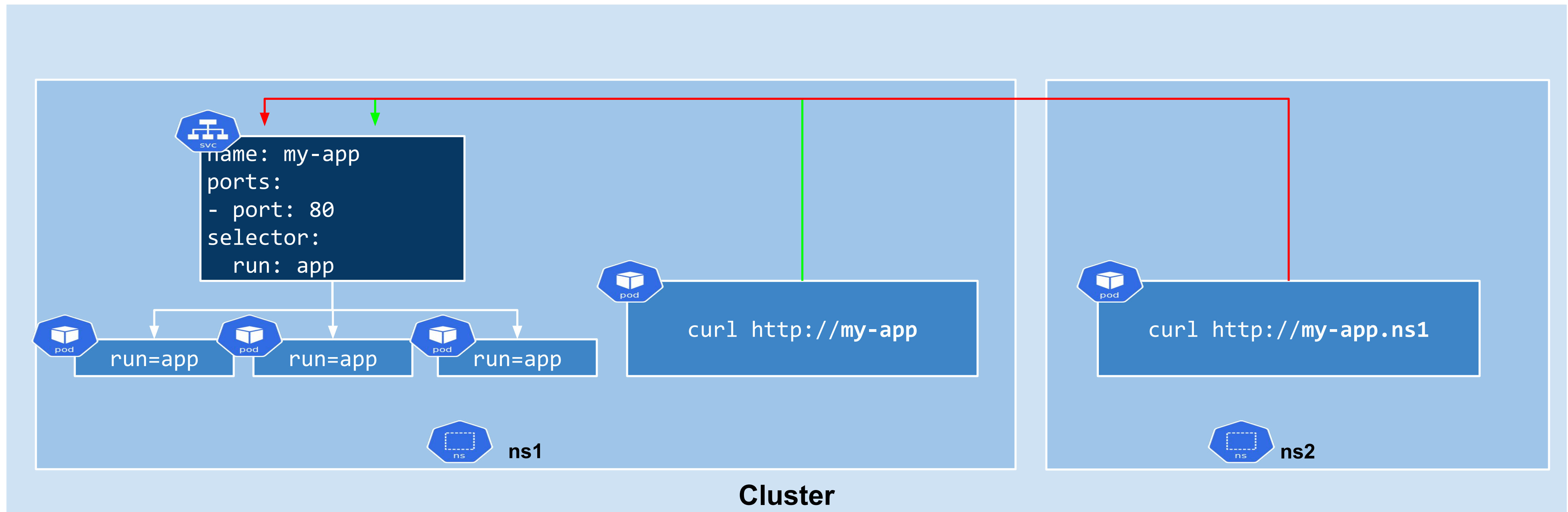


Il s'agit du 7eme des **twelve-factor app** :
Associations de ports

LES SERVICES ET LE NOMMAGE

- ▶ Normalement, un pod ne parle **jamais à un autre pod directement**, il passe par un service qui l'« **expose** »
- ▶ Les autres namespaces peuvent résoudre les services des autres ns avec `${svc}.${ns}`

service
pod
namespace
cluster

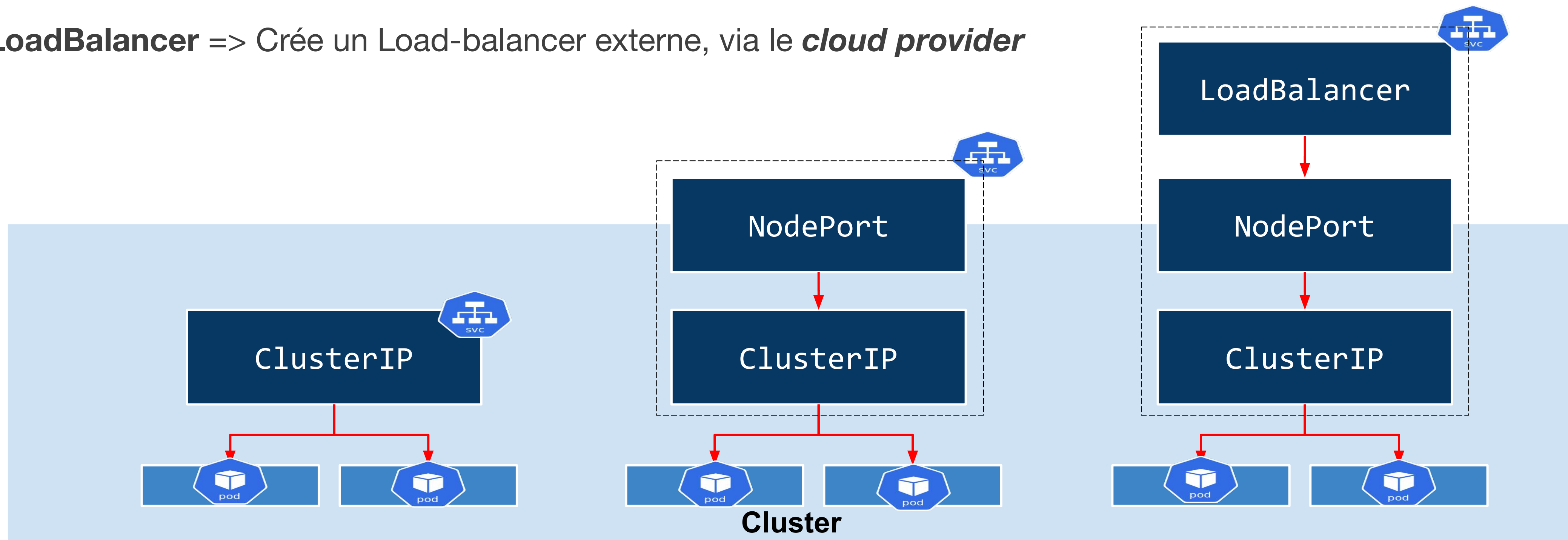


- ▶ **ClusterIP (défaut)** : allocation d'une adresse Interne au Cluster, uniquement accessible par d'autres pods
- ▶ **NodePort** : allocation d'un port spécifique (par défaut 30000-32767) sur tous les Nodes => permet l'accès par des composants externes au cluster
- ▶ **LoadBalancer** => Crée un Load-balancer externe, via le *cloud provider*

service

pod

cluster

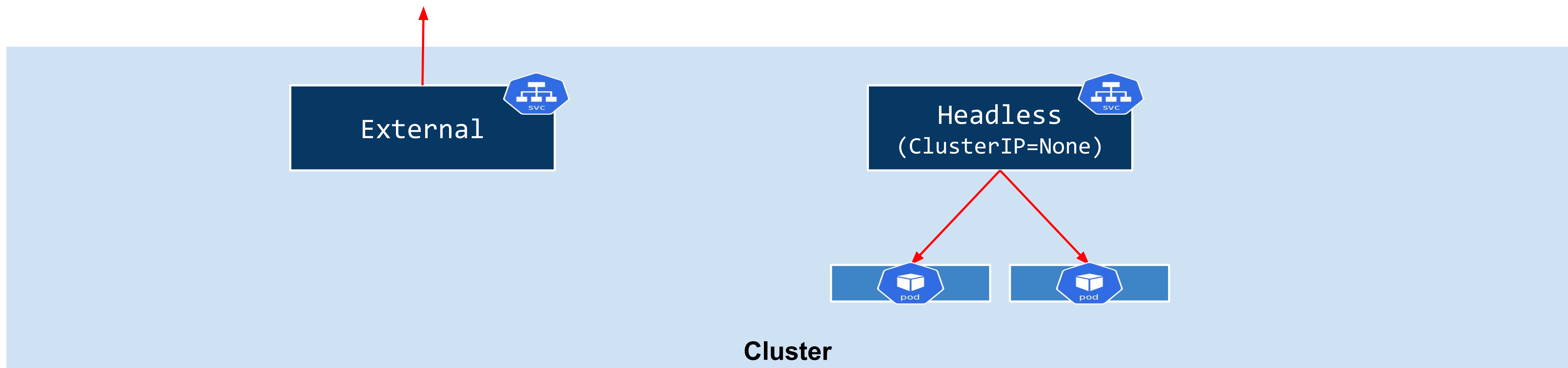


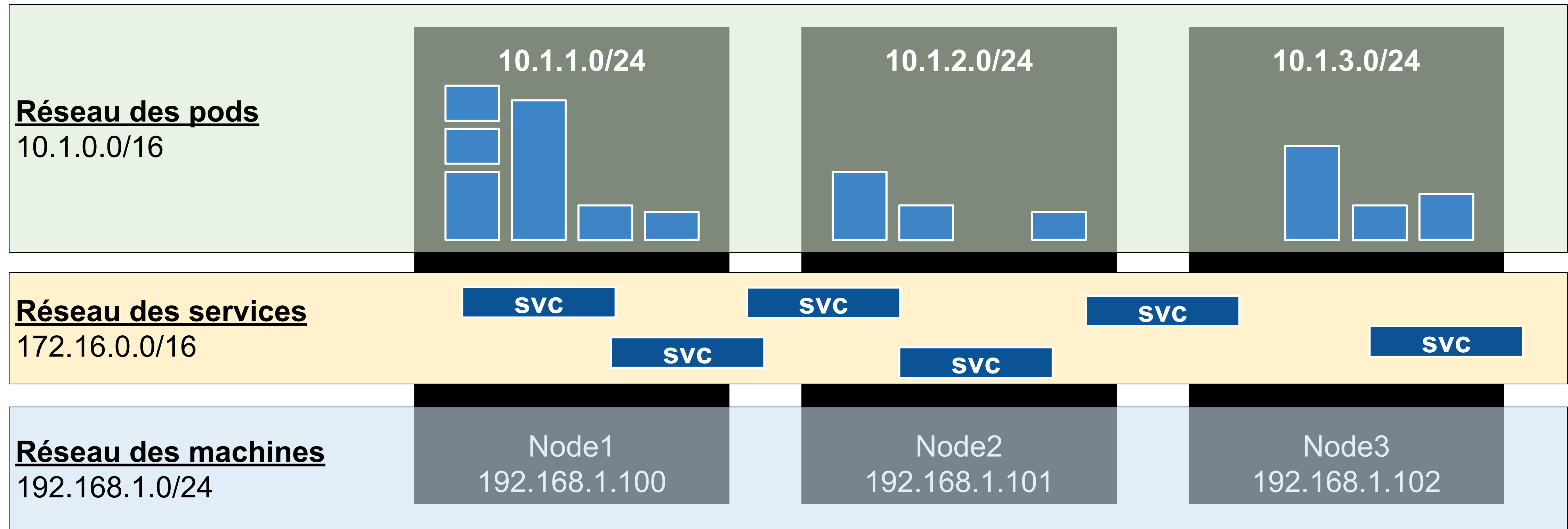
- ▷ **External** : pointeur DNS (statique, manuel) vers un service externe
- ▷ **ClusterIP (Headless)** : pas d'allocation d'une adresse Interne, simple liste ou *round-robin* DNS vers les pods

service

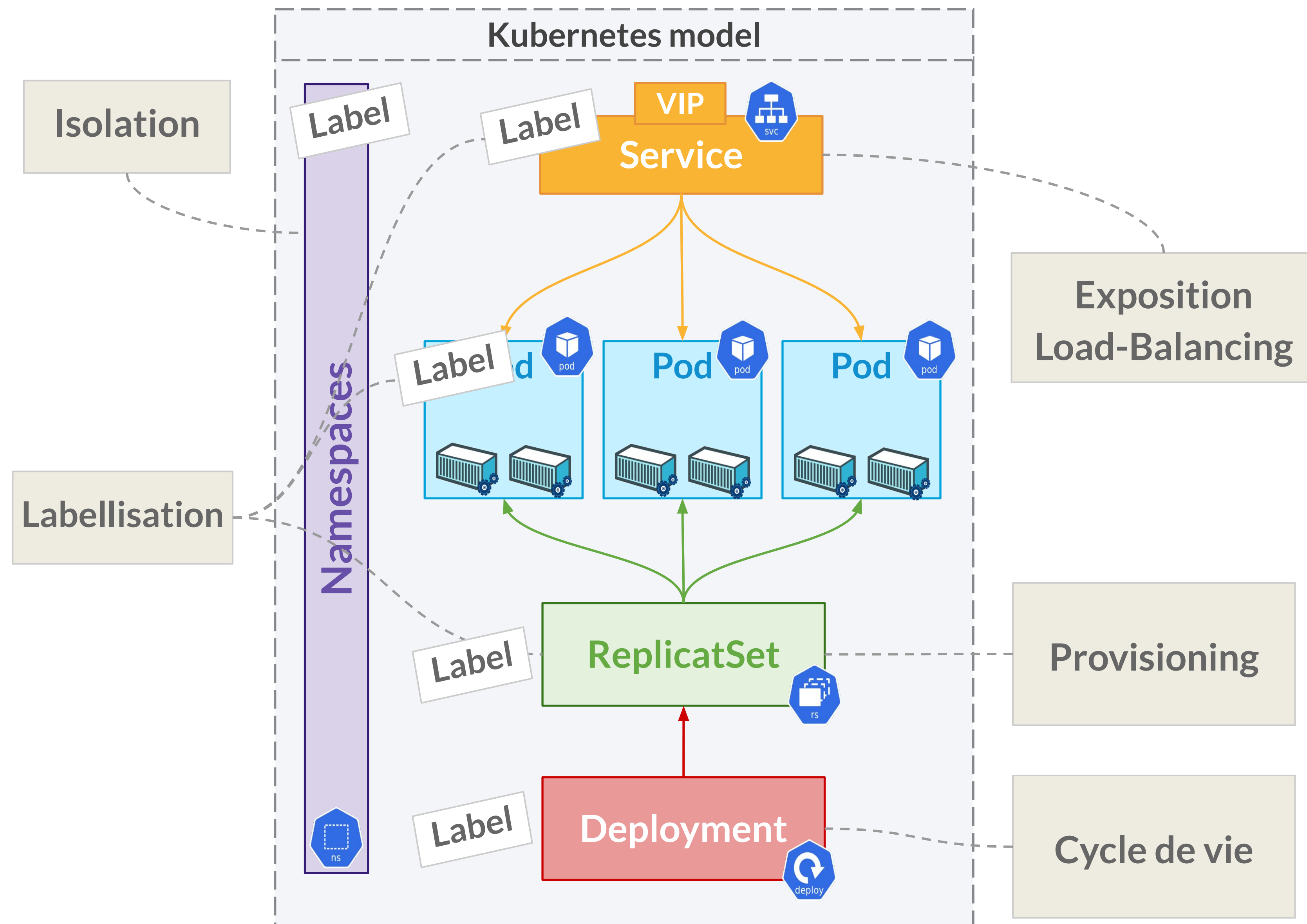
pod

cluster





Le réseau des machines est le seul à être obligatoirement accessible de l'extérieur d'un cluster. Les réseaux des pods et des services ne le sont généralement pas.



- ▷ Ils garantissent la (re)création des pods en encapsulant la définition d'un ou de plusieurs pod(s)
 - > Notion de **template** de pods à instancier
- ▷ Ils s'assurent du respect du « **taux de réplication** » **attendu** en re-crétant ou supprimant des pods au besoin

```
$ kubectl scale rs/rs1 --replicas=5  
replicaset "rs1" scaled
```



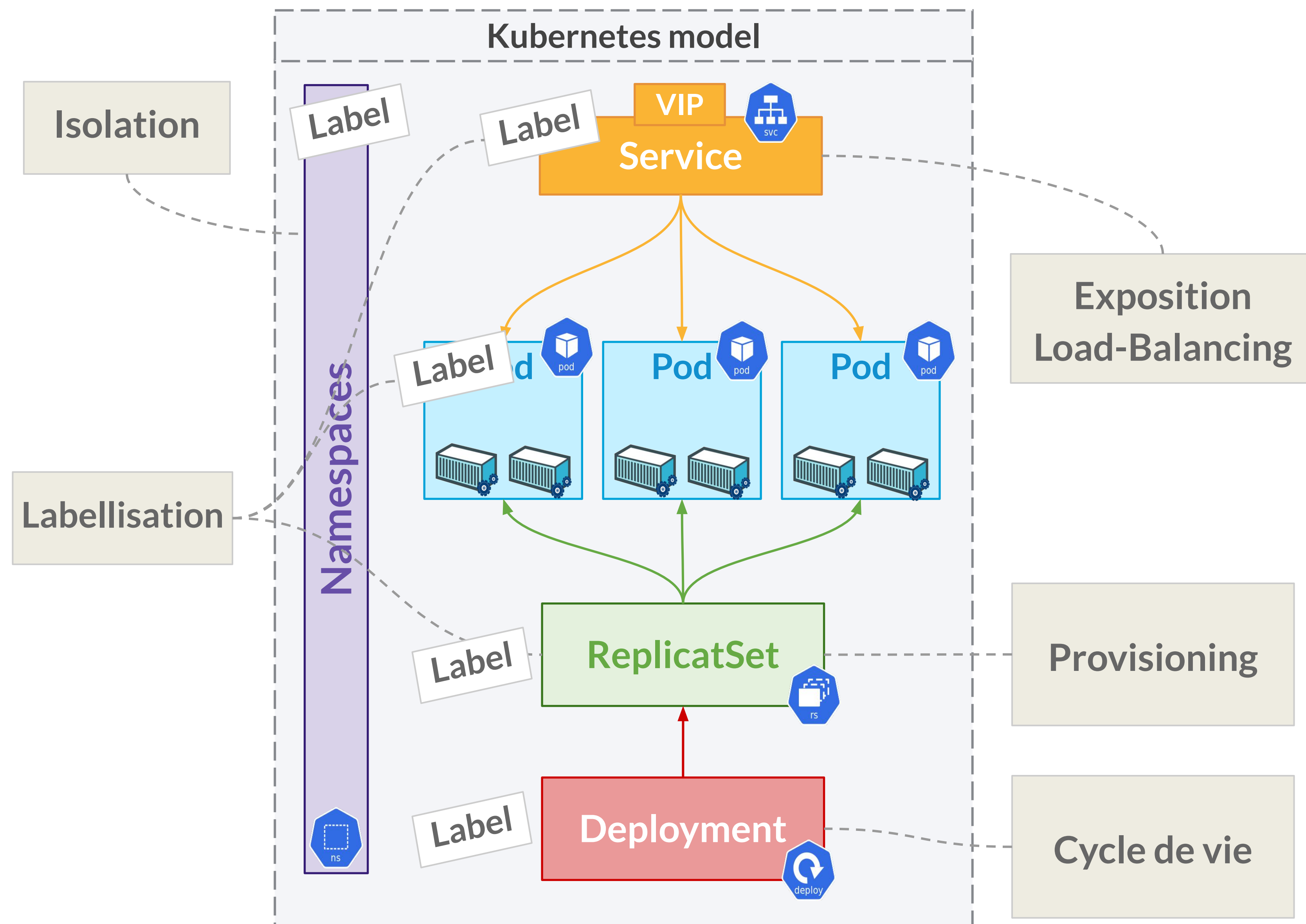
La gestion de **création** ou **recréation** d'un certain nombre de **réplicas** suit les principes édictés dans le **8eme** et **6eme twelve-factor app: Concurrency**.

Spécification des pods à
créer

*(voir exemple pods
pour format)*

```
apiVersion: v1
kind: ReplicaSet
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: nginx
  template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - image: nginx:1-10
        name: nginx
        ports:
        - containerPort: 80
          protocol: TCP
```

Nombre cible
et identification
des pods




La manipulation directe des **ReplicaSets**, même si elle est possible, est déconseillée, au profit d'un objet qui l'encapsule : le **Deployment**

Le **Deployment** est la gestion du **cycle de vie** et du **versioning** d'un ReplicaSet

Les **Deployments** peuvent adopter plusieurs stratégies pour les montées de version :

- ▷ **Recreate**
- ▷ **Rolling updates**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: truc
spec:
  replicas: 2
  selector:
    matchLabels:
      run: truc
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 1
      type: RollingUpdate
  template:
    metadata:
      labels:
        run: truc
    spec:
      containers:
        - image: my_image:v1
          name: truc
```



Légende

svc

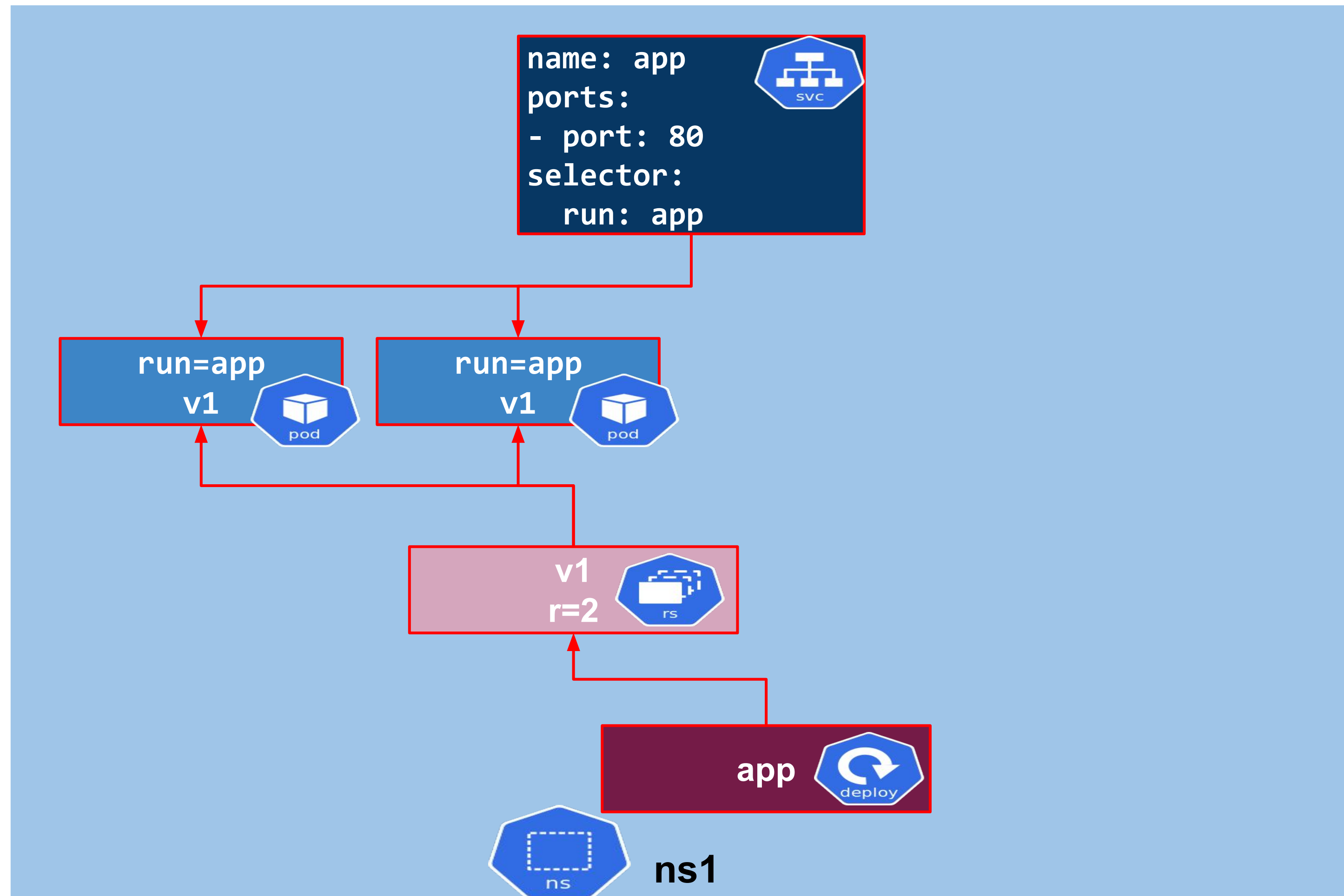
po

ns

rs

deploy

```
$ kubectl run  
--image=img:v1 \  
-r=2 \  
--expose \  
--port=80 \  
app
```



Légende

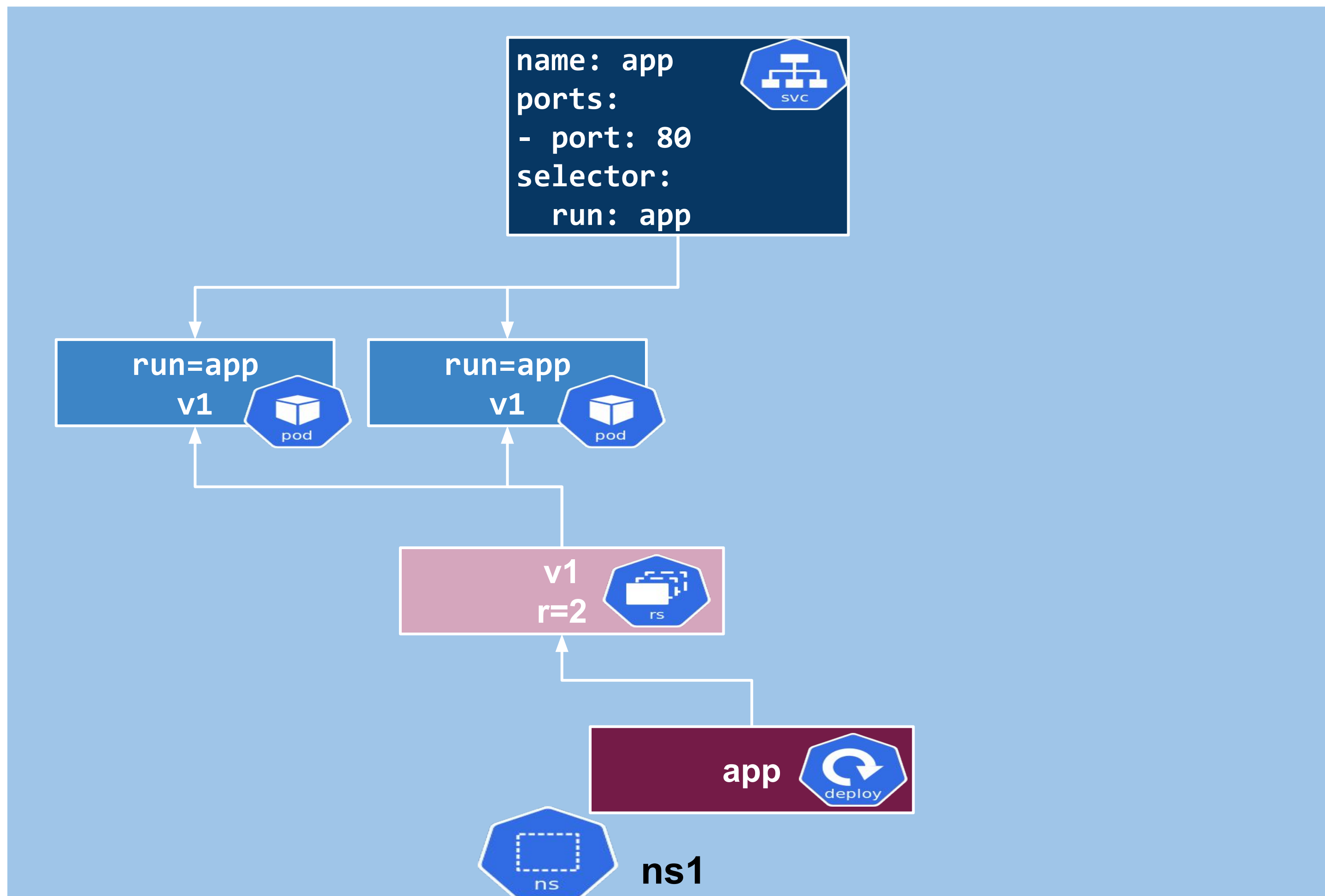
svc

po

ns

rs

deploy



Légende

svc

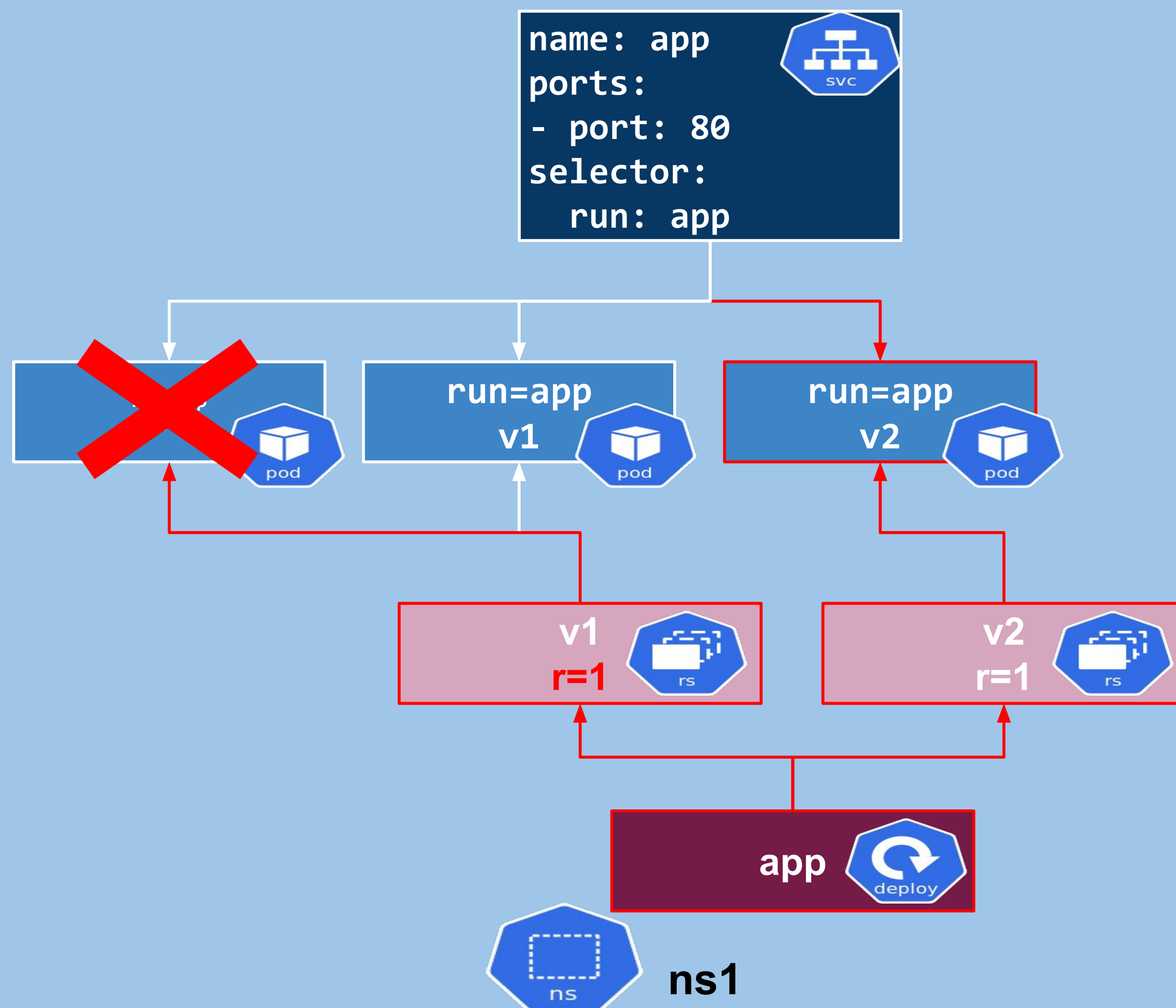
po

ns

rs

deploy

```
$ kubectl set image \
  deploy/app \
  app=img:v2
```



Légende

svc

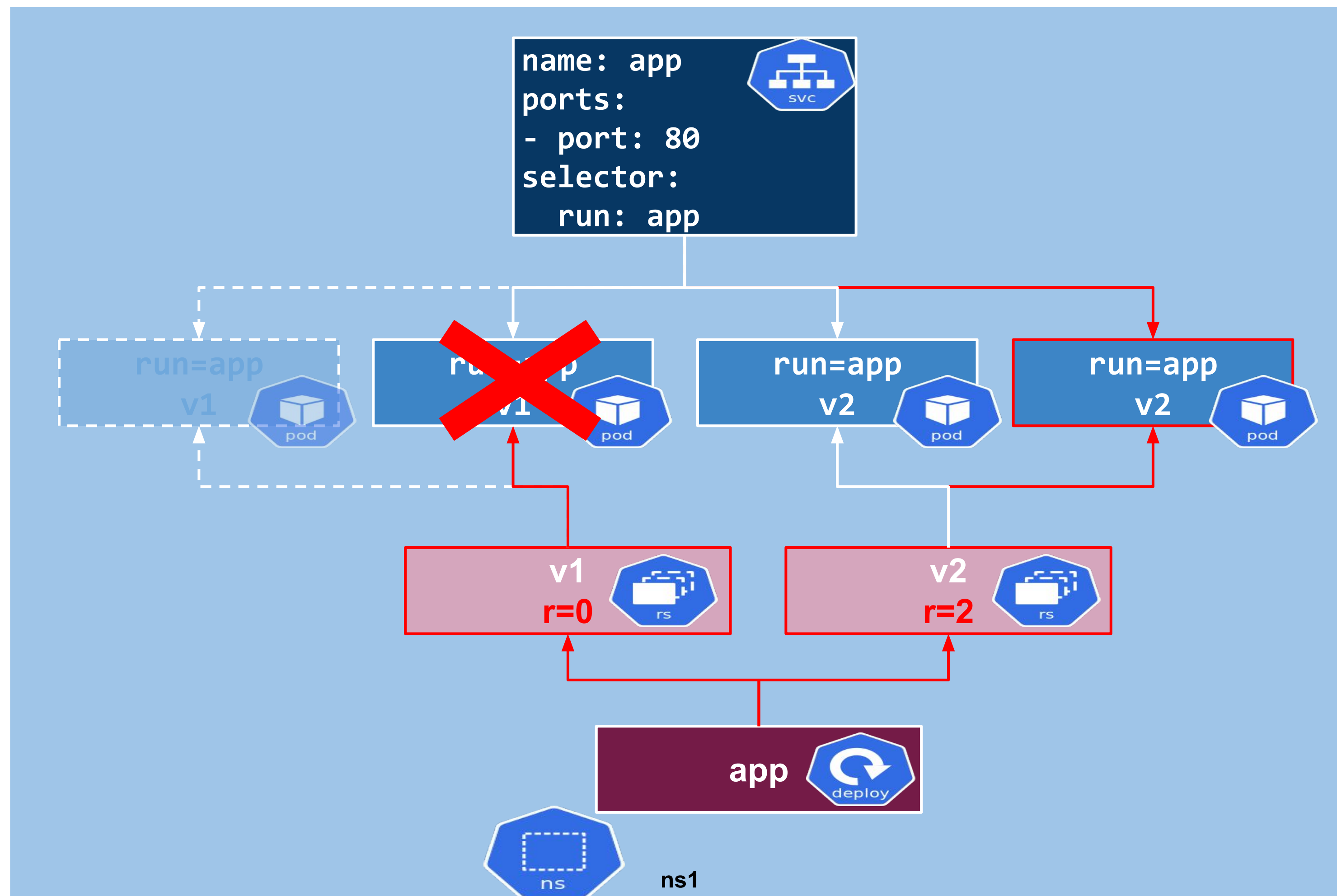
po

ns

rs

deploy

Note : le **svc** pointe sur des **pods** qui ont été provisionnés par **deux rs**. C'est là que la magie des labels / sélecteurs opère...



Légende

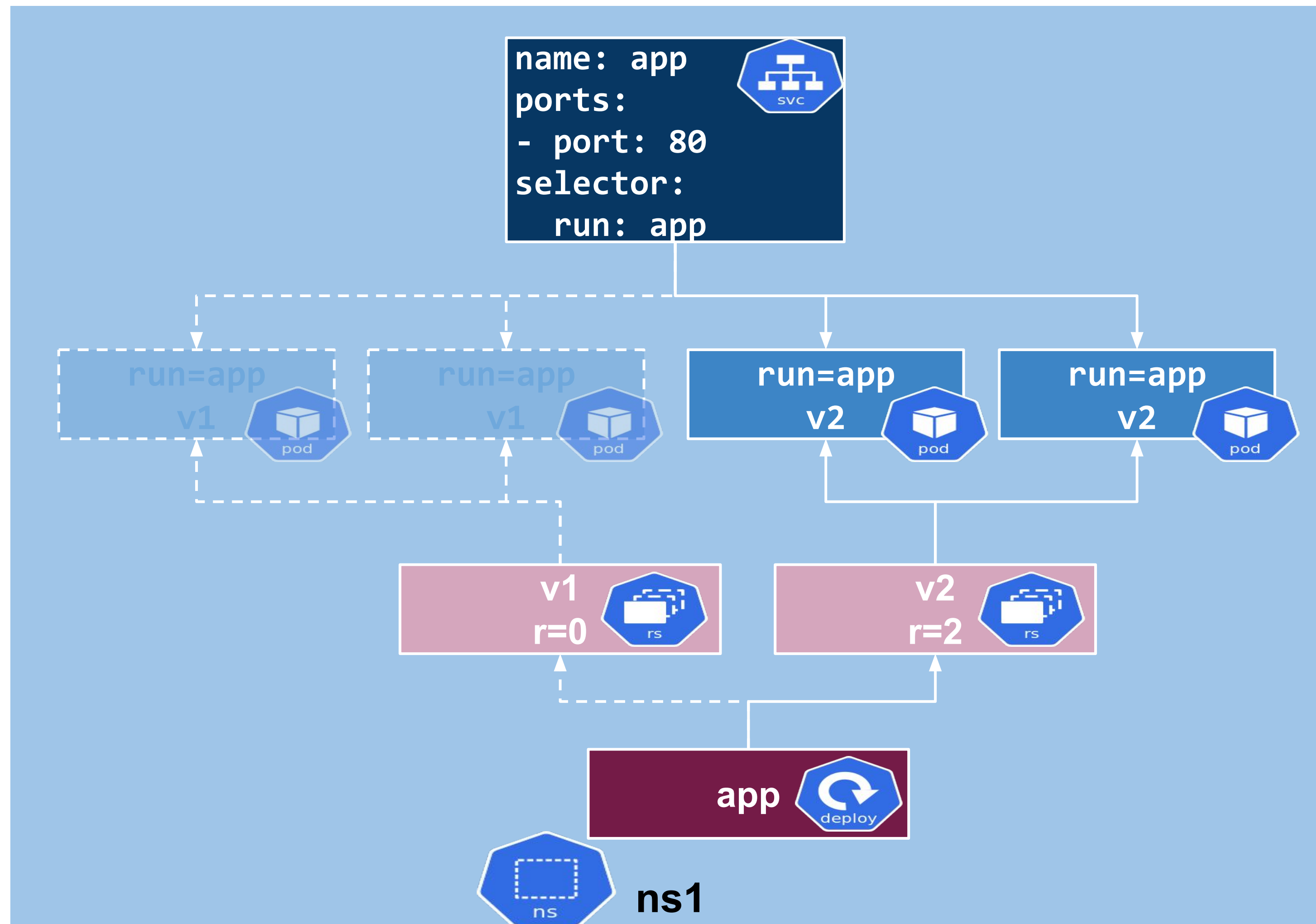
svc

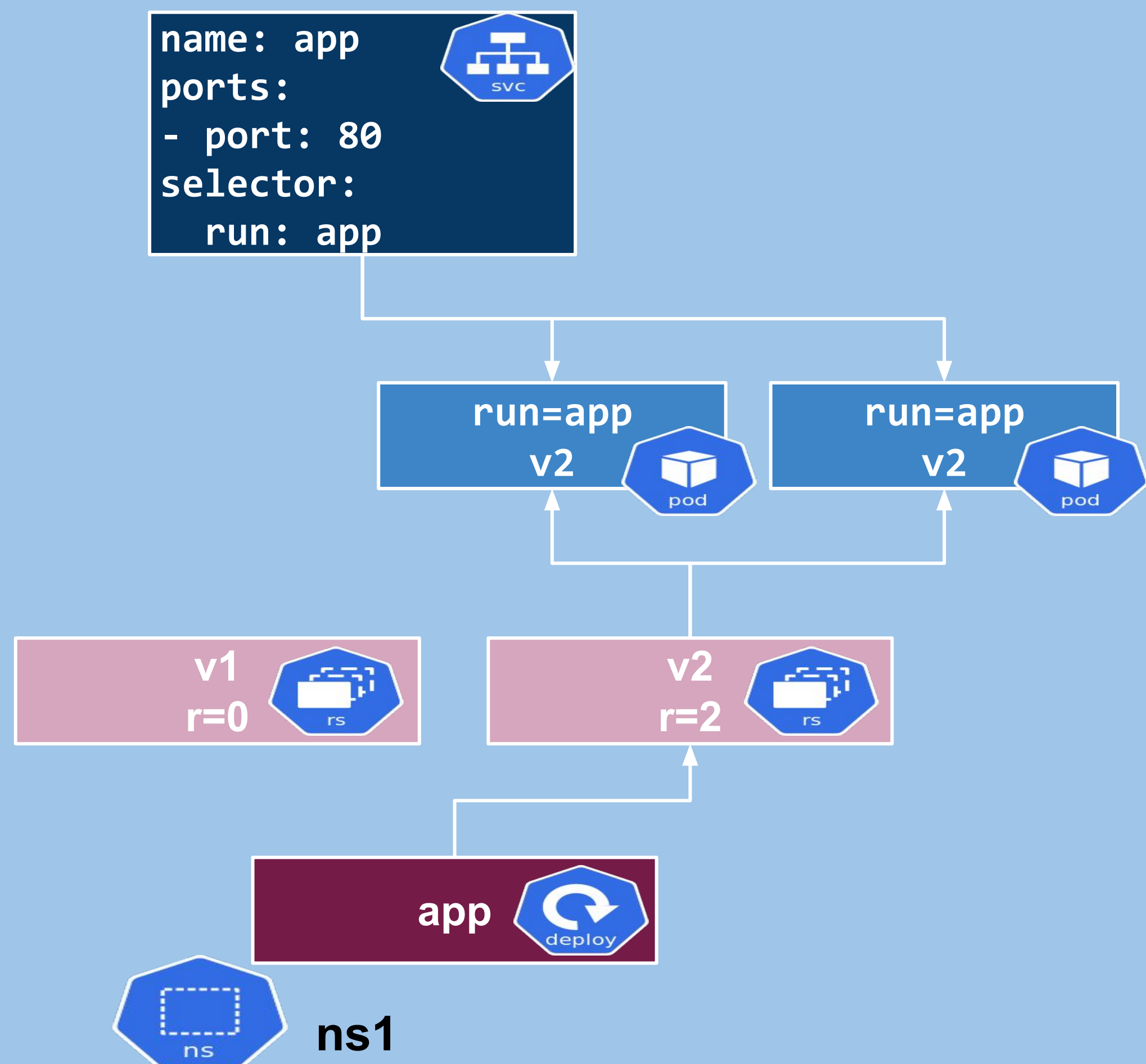
po

ns

rs

deploy





Légende

svc

po

ns

rs

deploy

Note : le rs de la v1 reste présent, il permet d'effectuer rapidement un rollback.

Voir la propriété
revisionHistoryLimit

- ▷ Voir l'historique des versions d'un Deployment

```
$ kubectl rollout history deploy/app
deployments "app"
REVISION  CHANGE-CAUSE
1          kubectl run app --image=nginx:1.12 --replicas=2 --expose=true --port=80 --record=true
2          kubectl set image deploy/app app=nginx:1.13 --record=true
```

- ▷ Faire un rollback sur une version précédente

```
$ kubectl rollout undo deploy/app --to-revision=1
```

- ▷ Mettre en pause / reprendre un changement de version d'un ReplicaSet

```
$ kubectl rollout (pause|resume) deploy/app
```

Pour l'instant, nous avons utilisé kubectl principalement sous sa forme qui masque la structure des ressources

- ▷ `kubectl run`
- ▷ `kubectl set`
- ▷ `kubectl scale`

Mais il est très souvent (tout le temps en fait) nécessaire d'être beaucoup **plus fin** dans la définition ou la **gestion des ressources**

Nous allons voir comment manipuler les ressources sous forme de **fichiers (JSON, YAML)**

- ▷ Tricher pour avoir un squelette de fichier à adapter

```
$ kubectl run ... --dry-run -o yaml > ressource.yaml
```

- ▷ Création d'objet à partir d'un fichier (ou d'un répertoire)

```
$ kubectl create -f ressource.(yaml|json)
```

- ▷ En mode « idempotence »

```
$ kubectl apply -f ressource.(yaml|json)
```

- ▷ En mode interactif (lance vim) => jamais en prod !!

```
$ kubectl edit type/ma_ressource
```

- ▷ En mode différentiel

```
$ kubectl patch -f ressource.(yaml|json)
```

- ▷ Suppression d'un objet à partir d'un fichier (ou d'un répertoire)

```
$ kubectl delete -f ressource.(yaml|json)
```

“ Mettre son App
dans K8s ”

Vous aurez dans ce repo un fichier TP2.md à lire et réaliser :

git clone https://token_kubernetes:vv87xgaLU7a8BQLKeP2e@gitlab.com/santunes-formations/kubernetes.git