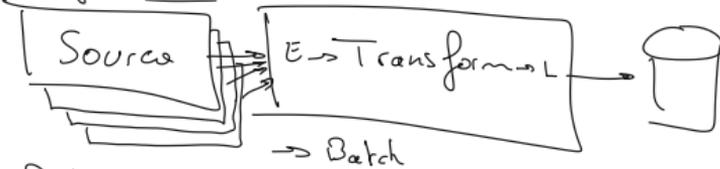


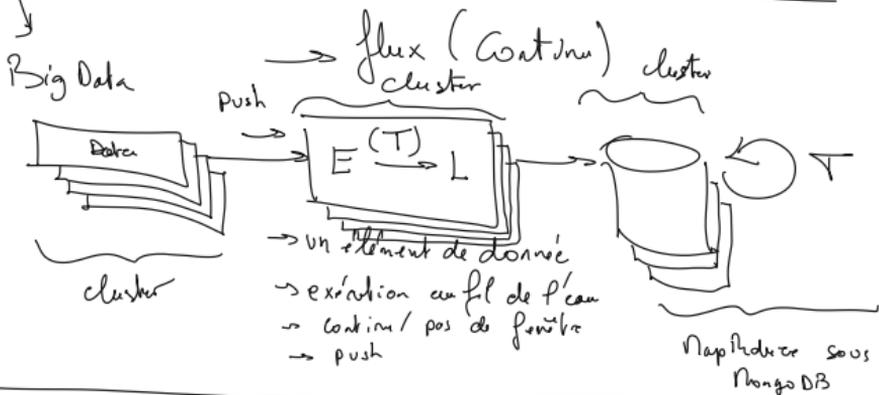
Bonjour tout le monde

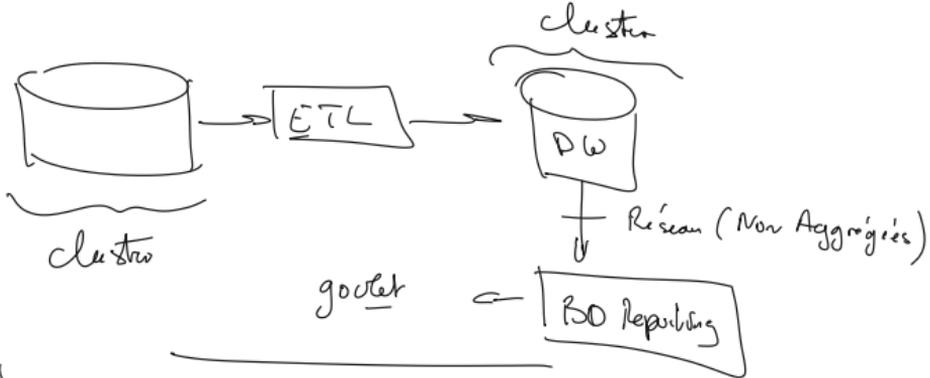
Démarrage du cluster : source crc.sh

Refaire régulièrement

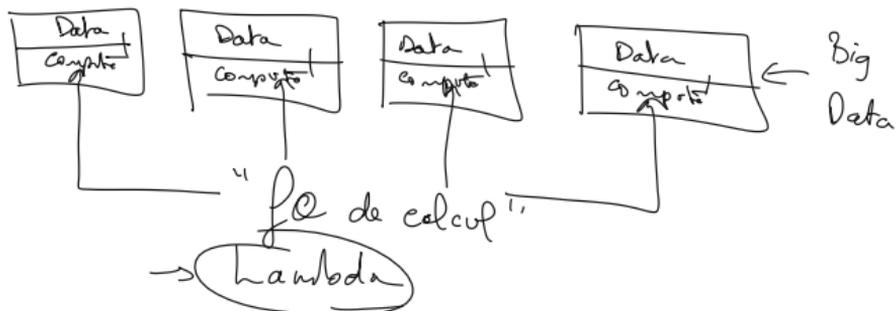


- Data
- voit un ensemble de données (Requête)
 - exécution régulière / planifiée
 - Durée d'exéc (fenêtre)
 - Push / Pull





"Big Compute"

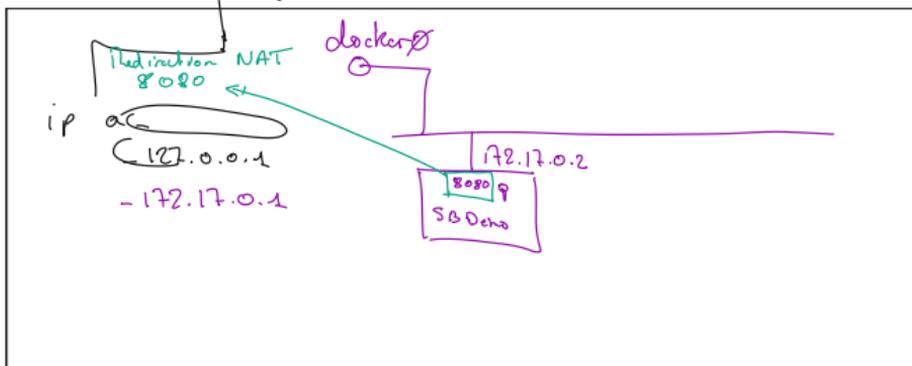


Solutions "HPC" → Très cher, spécifique, etc...

Autre VM

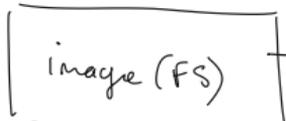
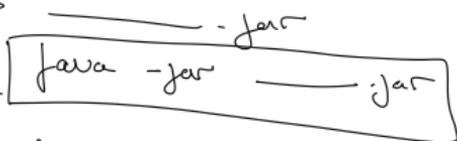
[databricks community edition](https://community.databricks.com/)

VM



♀ Dev → Code Source App. (Springboot)

→ Packager →

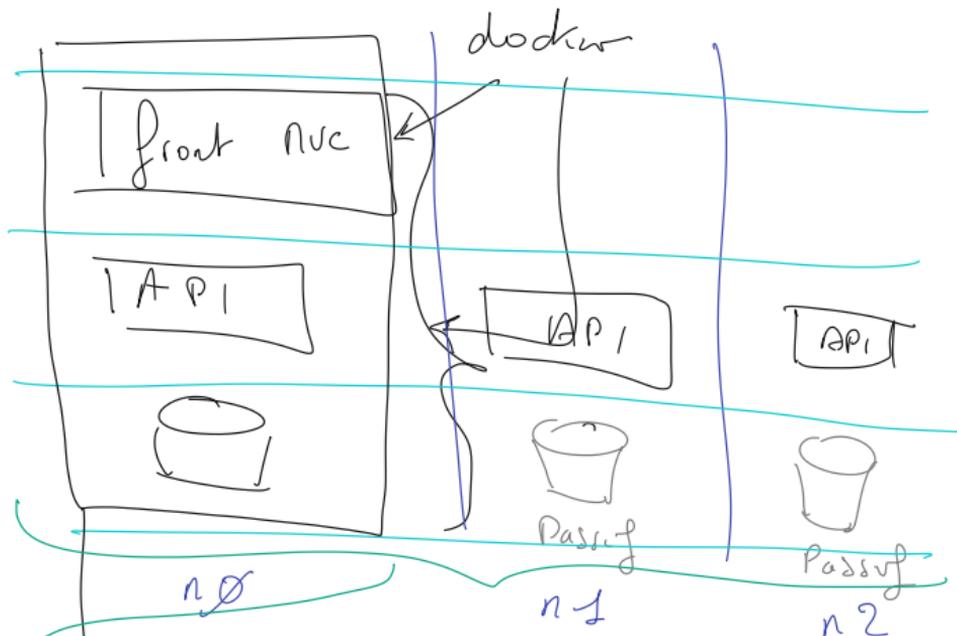


différentiel par rapport à l'image

Conteneurs

→ mode en lecture seule

linux distrib = Nojan + Filesystem.



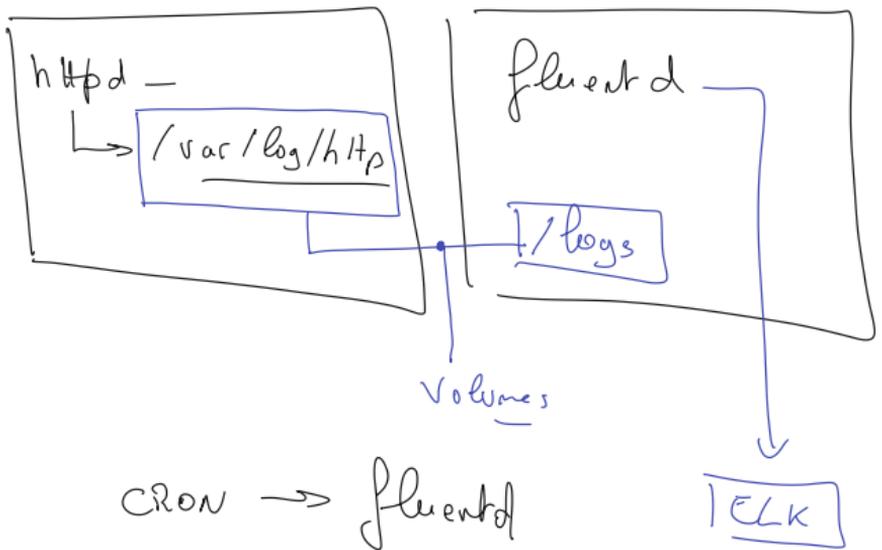
→ docker - Compose → ~~pas de NAT~~

① - docker SWARN

④ - OpenShift

② - (docker EE)

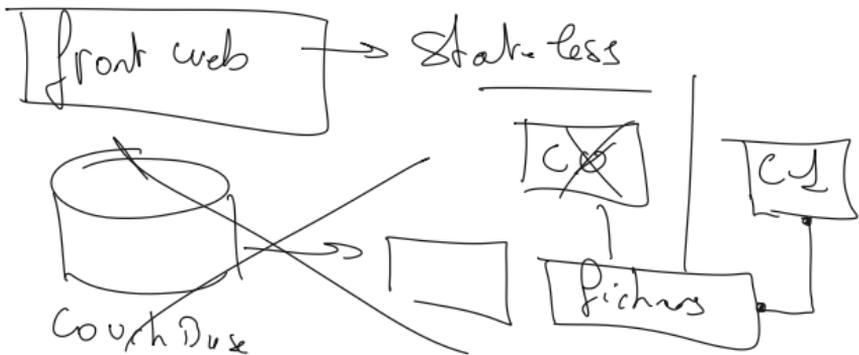
③ - Kubernetes - K8s



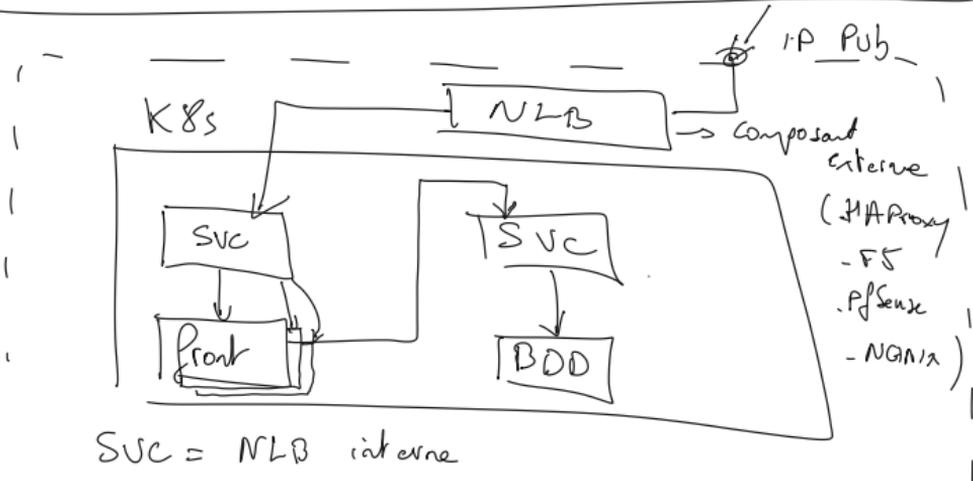
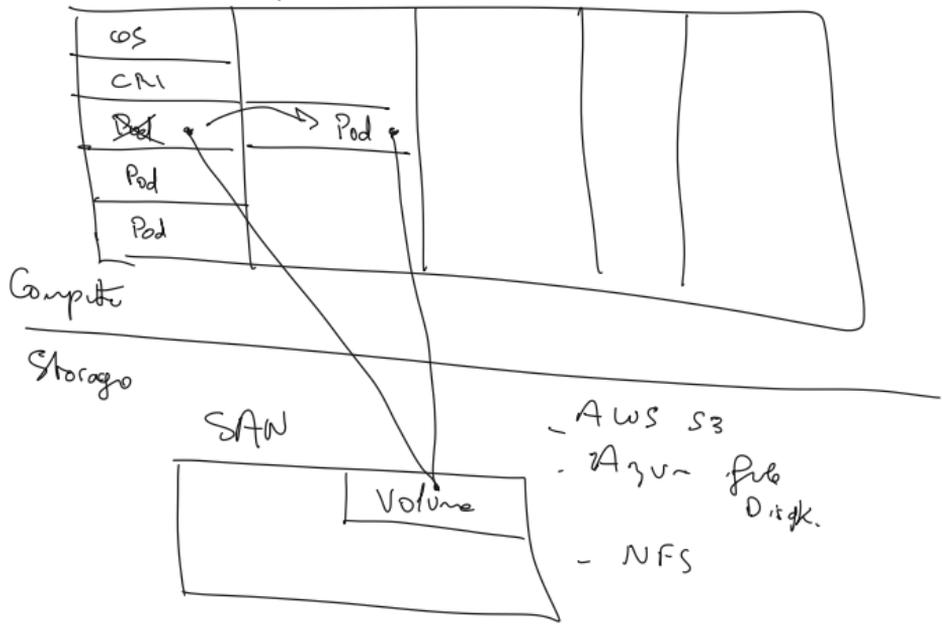
CRON \rightarrow fluentd

- Durée de vie d'une VM =
 \approx quelques années

- Durée de vie d'un conteneur
 \approx quelques heures



cluster OpenShift (RHCoreOS) (RHEL)



- Microsoft

- Oracle

.....

Solutions

① Ajout de column store

② Ajout de MVCC

Master 0

Produits

| <u>id</u> | Nom | Pcia | Timestamp | Async |
|-----------|-----|------|-----------|-------|
| 1 | A | | | |
| | | 12 | 10:07 | |
| 1 | A | 15 | 10:08 | |
| 1 | A | 20 | 10:09 | |
| 1 | A | 17 | 10:10 | |
| | | | 10:09 | |

12 → 15 → 20

Master 1

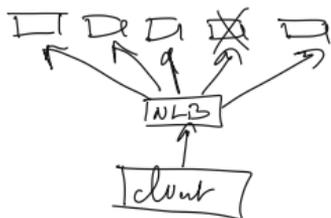
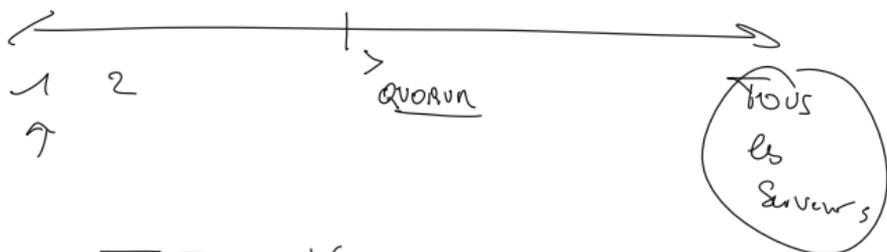
Produits

| <u>id</u> | Nom | Pria | Time Stamp |
|-----------|-----|------|------------|
| 1 | A | 12 | 10:07 |
| 1 | A | 15 | 10:08 |
| 1 | A | 17 | 10:10 |
| 1 | A | 20 | 10:09 |

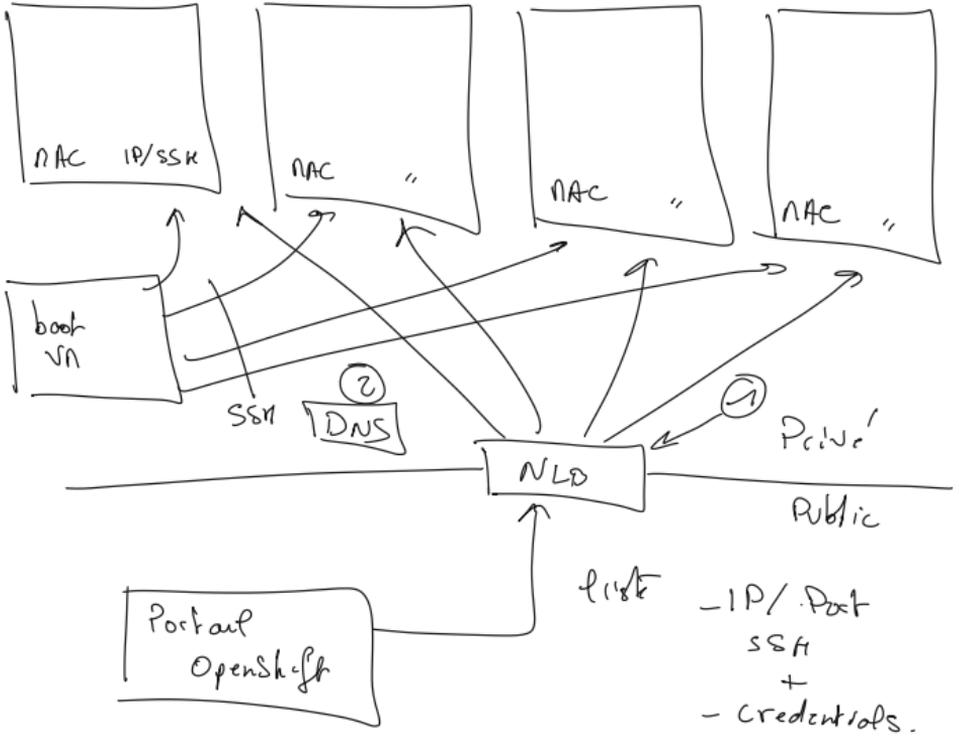
15 → 17

→ MVCC select pr from Pdt where id=1
(implwork = dernier timestamp)

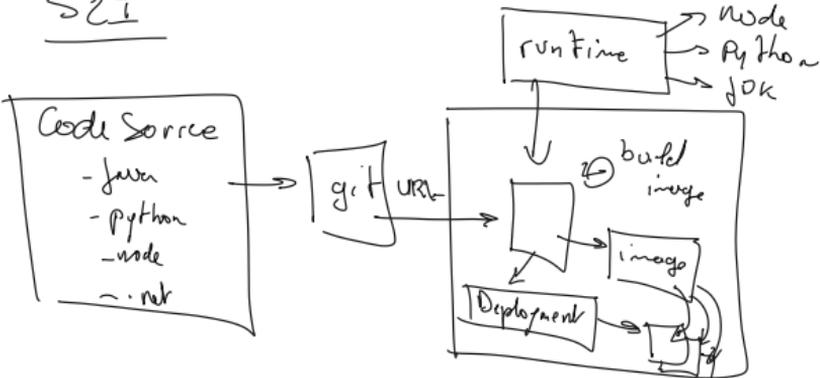
write'



RHCoreOS / RHEL



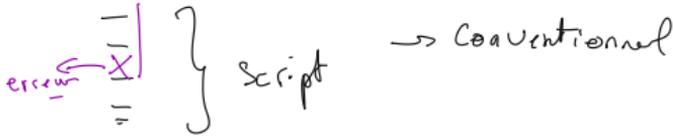
S2I



Tous les tutos en ligne OpenShift :

<https://developers.redhat.com/learn/openshift>

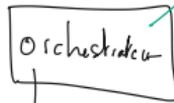
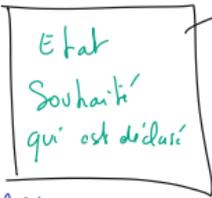
"imperative" → NON RECOMMANDÉ
→ ligne de commande "étape par étape"



→ modifier

"Déclarative"

flux
"déclaratif"
↓
format



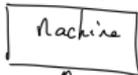
Etat actuel de la cible?

YAML

utilisateur

INI → 2 Section
(CSV) → 1 niveau
def-valeur
XNL → Complexité, bord

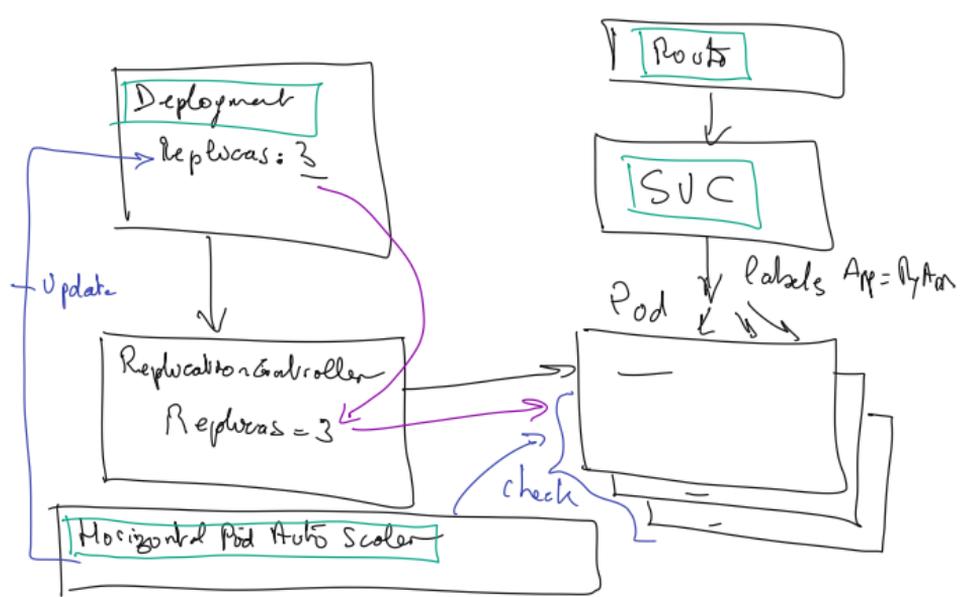
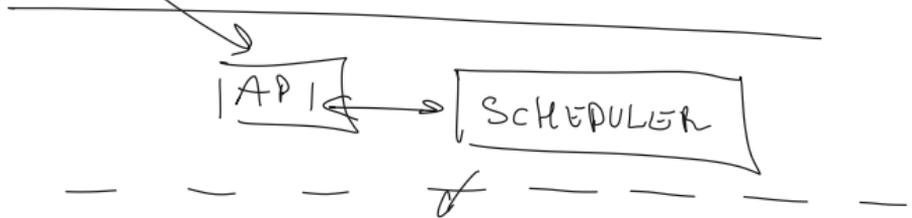
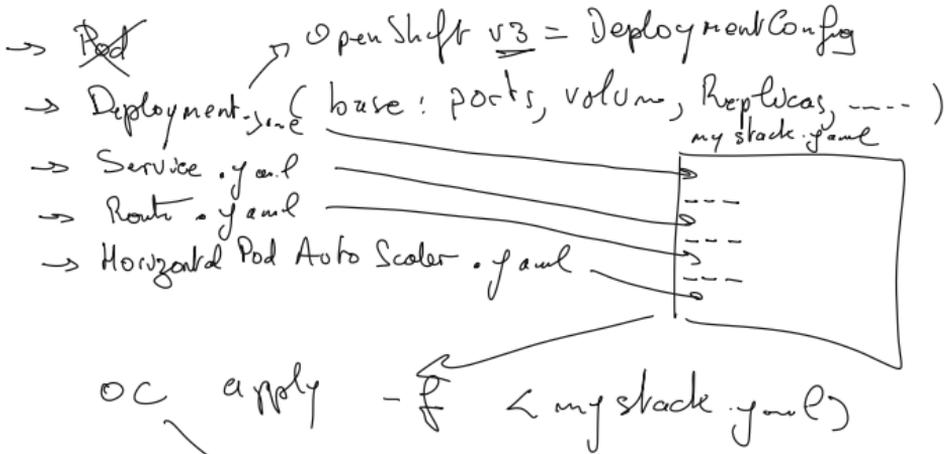
→ YAML

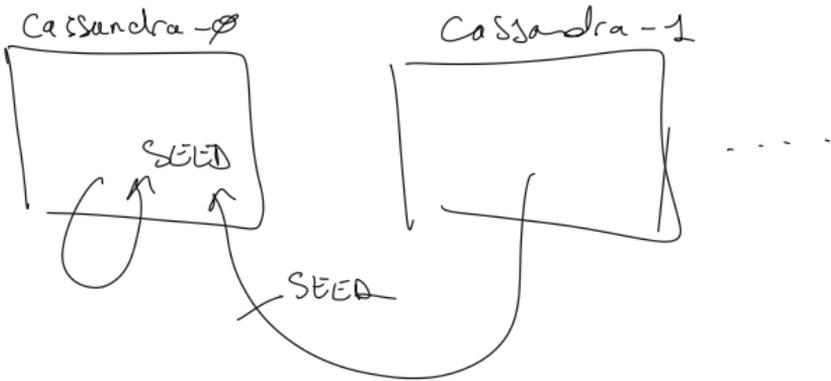


-XNL
-JSON



exercice → Dep en élastique un Service
Spring boot Demo





Dockerfile
FROM

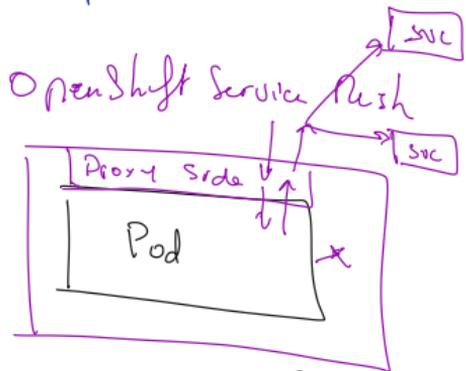
image de base

- ↳ SCRATCH → -sh pas de paquets
- BUSYBOX → -fs
- ALPINE → apk
- Centos/ubuntu

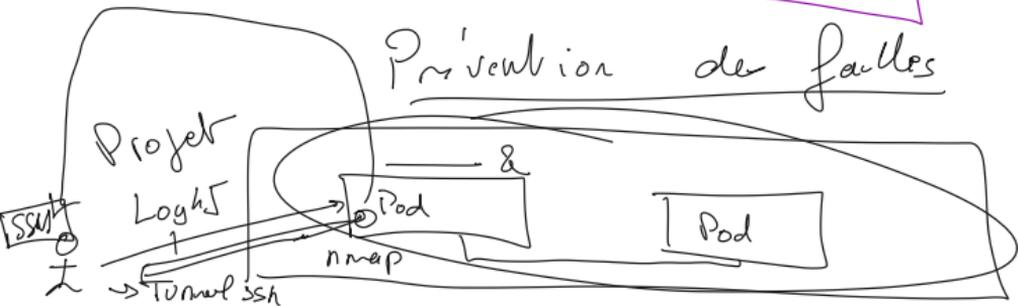
Redis

In/Egress

Book



Prévention de failles



(AAAA.nn

22.04)

- Sémantique -

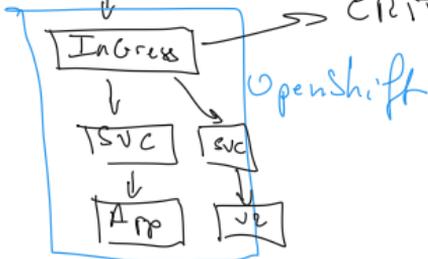
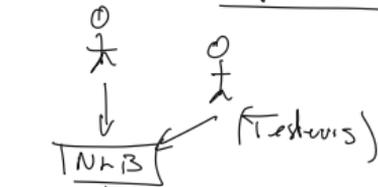
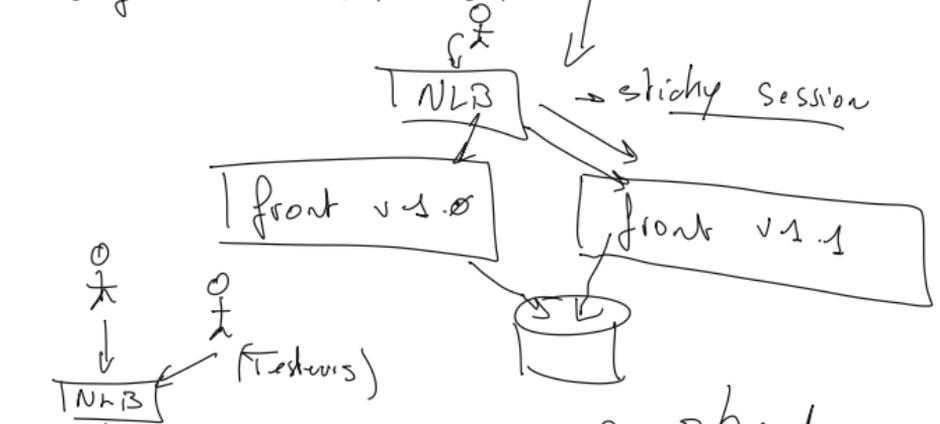
API. mm. Re. b n R

ex:

- Maître → ajout fe/breaking change "autre produit"
- mineur → ajout fe/optim/correctifs compét. ascendant
- Révision → correction de bug
- build number → inc auto sur chaque build

+ fe côte à côte possible

PAS de fonctionnement côte à côte



CRITERES = ?

- header
 - URL (Route)
 - IP Source
 - ... ?
- Groupes AD? (a vérif. par s' possible)

Routage en fonction de la source :

apiVersion: networking.k8s.io/v1

kind: **Ingress**

metadata:

name: ingress-wildcard-host

spec:

rules:

- host: "**foo.bar.com**"

http:

paths:

- pathType: Prefix

path: **"/bar"**

backend:

service:

name: **service1**

port:

number: 80

- host: **"*.foo.com"**

http:

paths:

- pathType: Prefix

path: **"/foo"**

backend:

service:

name: **service2**

port:

number: 80

Types d'ingress K8s:

<https://kubernetes.io/docs/concepts/services-networking/ingress/#types-of-ingress>

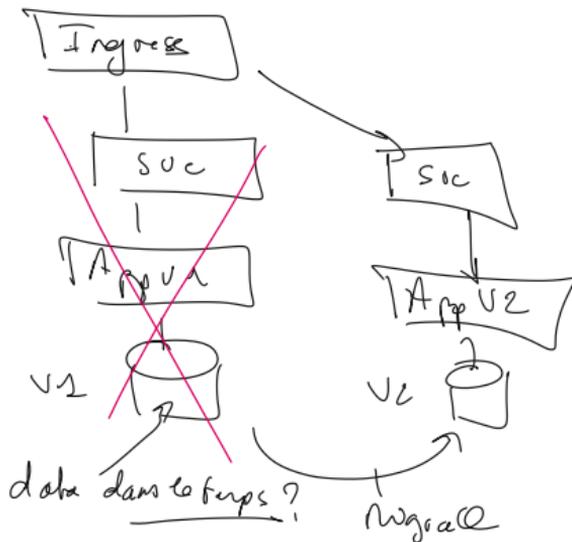
Ingress OpenShift :

<https://docs.openshift.com/container-platform/4.9/networking/ingress-operator.html>

Attention à l'impact de s'appuyer sur OpenShift Service Mesh

Spécifier les règles de trafic sortant (équivalent firewall) :

<https://kubernetes.io/docs/concepts/services-networking/network-policies/#targeting-a-range-of-ports>

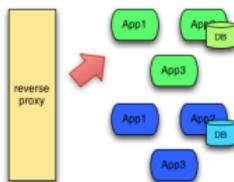


Référence vers les stratégies de déploiement :

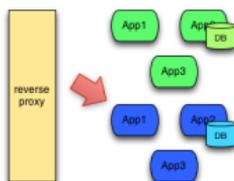
https://docs.openshift.com/container-platform/3.11/dev_guide/deployments/deployment_strategies.html#rolling-strategy

Blue Green :

Simply, you have two identical environments (infrastructure) with the "green" environment hosting the current production apps (app1 version1, app2 version1, app3 version1 for example):



Now, when you're ready to make a change to app2 for example and upgrade it to v2, you'd do so in the "blue environment". In that environment you deploy the new version of the app, run smoke tests, and any other tests (including those to exercise/prime the OS, cache, CPU, etc). When things look good, you change the loadbalancer/reverse proxy/router to point to the blue environment:

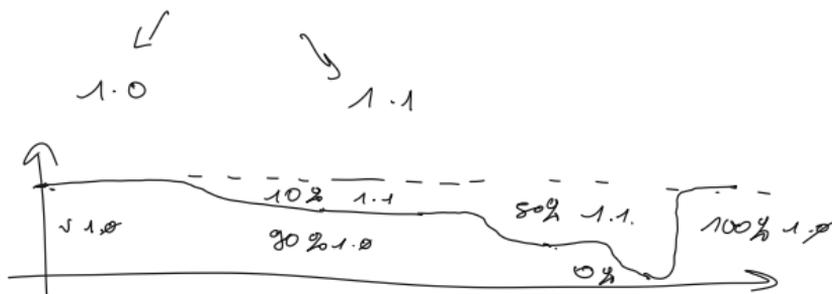


You monitor for any failures or exceptions because of the release. If everything looks good, you can eventually shut down the green environment and use it to stage any new releases. If not, you can quickly rollback to the green environment by pointing the loadbalancer back.

Sounds good in theory. But there are things to watch out for.

- **Long running transactions in the green environment. When you switch over to blue, you have to gracefully handle those outstanding transactions as well as the new ones. This also can become troublesome if your DB backends cannot handle this (see below)**
- **Enterprise deployments are not typically amenable to "microservice" style deployments - that is, you may have a hybrid of microservice style apps, and some traditional, difficult-to-change-apps working together. Coordinating between the two for a blue-green deployment can still lead to downtime**
- **Database migrations can get really tricky and would have to be migrated/rolledback alongside the app deployments. There are good tools and techniques for doing this, but in an environment with traditional RDBMS, NoSQL, and file-system backed DBs, these things really need to be thought through ahead of time; blindly saying you're doing Blue Green deployments doesn't help anything - actually could hurt.**
- **You need to have the infrastructure to do this**
- **If you try to do this on non-isolated infrastructure (VMs, Docker, etc), you run the risk of destroying your blue AND green environments**

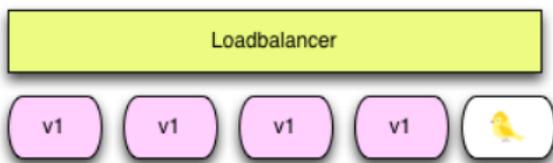
A/B Testing



Canary releases

Lastly, Canary releases are a way of sending out a new version of your app into production that plays the role of a “canary” to get an idea of how it will perform (integrate with other apps, CPU, memory, disk usage, etc). It’s another release strategy that can mitigate the fact that regardless of the immense level of testing you do in lower environments you will still have some bugs in production. Canary releases let you test the waters before pulling the trigger on a full release.

The faster feedback you get, the faster you can fail the deployment, or proceed cautiously. For some of the same reasons as the blue-green deployments, be careful of things above to watch out for; ie, database changes can still trip you up.



Sources SpringBootDemo :

https://nboost.visualstudio.com/DefaultCollection/SpringBootDemo/_git/SpringBootDemo (git clone)

Helu

Kind : Deployment

Kind : svc

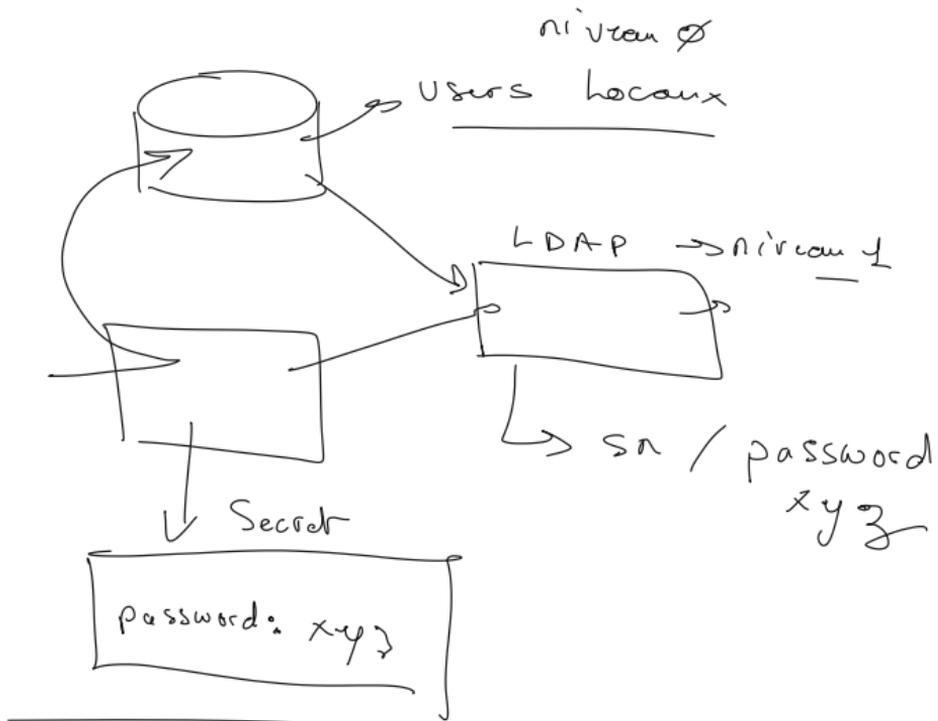
Kind : Ingress

Kind : Deploy
.....

Kind : svc
.....

Kind : Ingress
etc.....

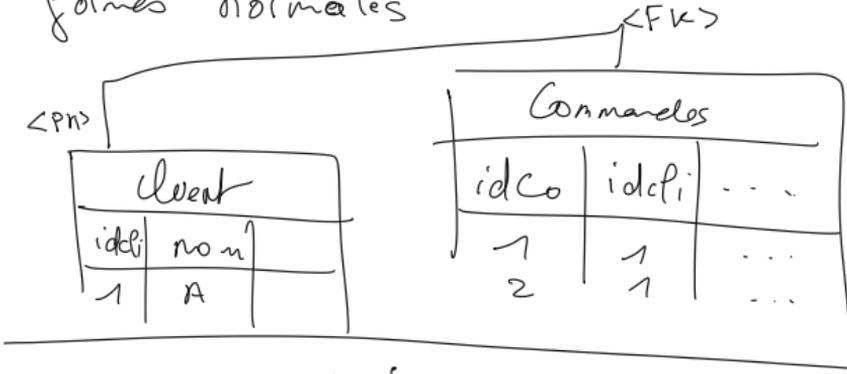
3 Turrets (yaml)



- login / md5 → texte, non modifiable, couple
- clef → texte, copie/collé, non modifiable
- Secret → binaire/base64, flux, clef renforcée
- Certificat →
 - flux norme X.509
 - - secret (auth. TLS)
 - - clef Publique/privée

SQL

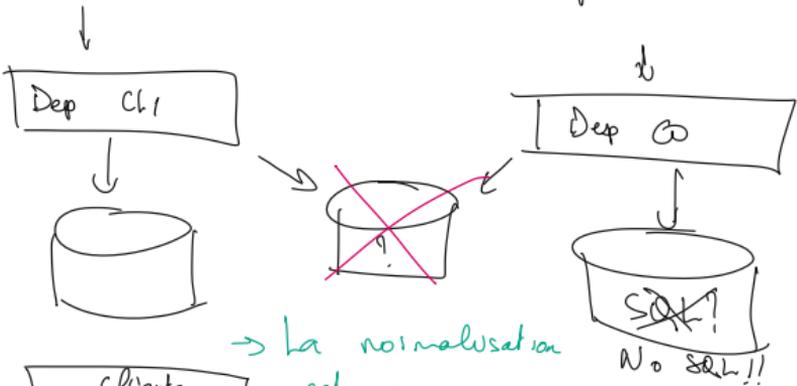
formes normales



Micro Services

api client
/api/clients

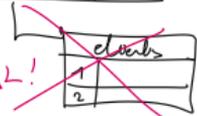
api commandes
/api/commandes



→ la normalisation est un ANTI PATTERN en Micro Services!!

| idco | date Cdo | Nom Client | Adresse |
|------|----------|------------|------------|
| 1 | | A | 10 Rue ... |
| 2 | | A | 10 Rue ... |

Pas en NoSQL!



Service Mesh :

<https://www.youtube.com/watch?v=Uo8LEcUMVxg>