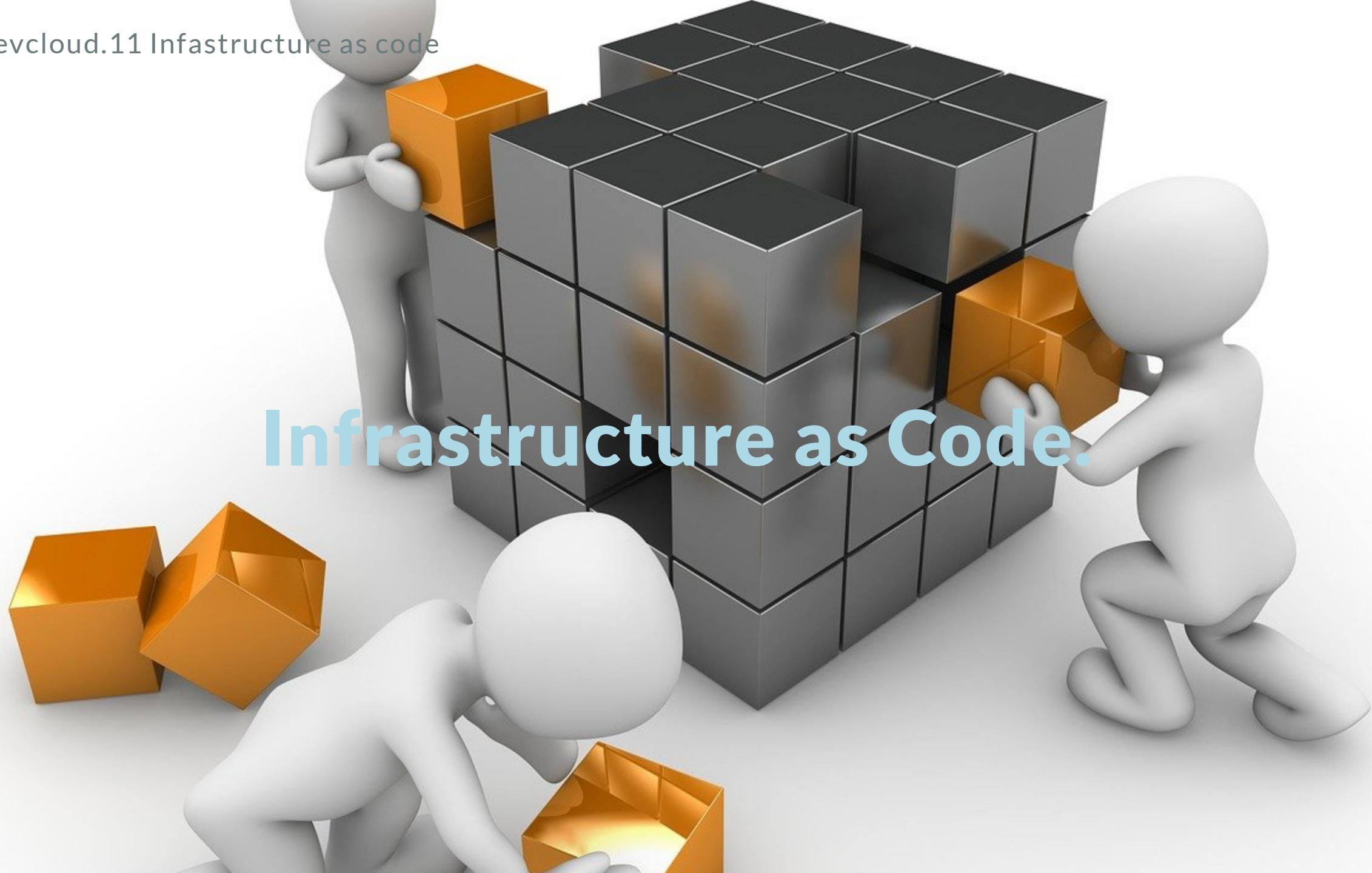


# Infrastructure as Code.



# Définitions

- L'infrastructure as code (**IaC**) est le processus de gestion et d'approvisionnement de ressources informatiques par le biais de fichiers de configuration au lieu d'interactions manuelles avec des systèmes ou des interfaces utilisateur.
- C'est de l'**automatisation** de l'infrastructure.

Quelle est la différence entre l'IaC et la gestion de configuration ?

# Intérêts

L'infrastructure as code permet aux équipes informatiques de:

- Gérer et de provisionner **rapidement** des ressources informatiques.
- Gérer des **versions** et des historiques de configuration.
- D'améliorer la **collaboration** entre les équiptiers.
- D'**automatiser des tests** de configuration ou de participer à la validation du code.
- D'**auditer** les configurations.

# Intérêts

- De **reproduire** rapidement des **environnements** (prod, pré-prod, dev, test, formation ...)
- De **documenter** les configurations.
- D'améliorer la **fiabilité** et la sécurité en évitant les erreurs humaines et en limitant le **drift** (dérive) des configurations.
- C'est un outil de la chaîne CI/CD.

## **L'IaC est en phase avec les technologies du Cloud et la philosophie DevOps.**

- Il faut provisionner les ressources qui correspondent aux besoins instantanés pour ne pas sur-consommer ou ne pas avoir de ressources insuffisantes pour tenir la charge. (trop d'agent dépensé ou perdu)
- L'automatisation est un des piliers du Cloud et du DevOps, l'IaC permet de le faire aussi sur le "build" d'infrastructure.

# Inconvénients

L'IaC peut:

- Rajouter de la **complexité** (outils supplémentaires , résolution de conflits lors des merge de configurations, inflation des infrastructures)
- Augmenter la **dépendance** à un outil et/ou à un "cloud provider" (lock-in).

# Les outils de l'IaC "indépendants":

- **Terraform** (Hashicorp) ou son fork F.O.S.S **OpenTofu** : Déploiement de ressources sur les principaux Clouds et sur des serveurs locaux.
- "**Pulumi**" : Déploiement de ressources sur les principaux Clouds et sur des serveurs locaux.
- "**Heat**" (OpenStack) : Déploiement de ressources sur OpenStack.

...

L'intérêt de ces outils est de limiter sa dépendance à un "cloud provider" et de pouvoir déployer sur plusieurs clouds.

# Les outils de l'IaC liés au "Cloud providers":

- "**CloudFormation**" (AWS) : Déploiement de ressources sur AWS.
- "**Azure Resource Manager**" (Microsoft) : Déploiement de ressources sur Azure.
- "**Google Cloud Deployment Manager**" (Google) : Déploiement de ressources sur Google Cloud.

# Les outils connexes à l'IaC.

- Github et Gitlab et autres outils de gestion de versions des fichiers.
- Nautobot, Netbox et autres sources de vérité.
- Cloud-init qui permet de configurer des VM au démarrage et qui est utilisable par Terraform, Vagrant..

# Gitops

GitOps est un cadre opérationnel qui utilise Git comme **source unique de vérité** pour l'infrastructure informatique. Il s'agit d'une approche de gestion de l'infrastructure qui utilise les pratiques DevOps traditionnelles, telles que l'automatisation, le contrôle des versions et la collaboration, pour fournir une infrastructure cohérente et fiable.

# Les outils HASHICORP liés à l'IaC.

- **Packer** : Création d'images de VM (KVM,"cloud providers", OpenStack, VMware, VirtualBox, Docker, ...)
- **Vagrant** : Packaging de VM pour le poste de travail (KVM, VirtualBox, VMware, Docker ...)
- **Terraform** : Déploiement de ressources sur les principaux Clouds et "on premises".

# Principes communs aux produits Hashicorp.

Chacun des outils Hashicorp a ses spécificités mais ils partagent des principes communs:

- Le langage de configuration est le **HCL** (Hashicorp Configuration Language) qui est un langage déclaratif est utilisé par Terraform et par Packer.
- Les **cibles** de ces outils sont les environnements Cloud ou "on premises". On utilise des plugins qui font le lien avec la cible (cloud, KVM, OpenStack, Docker, K8s...)

# Composants de Packer

- Un "**artifact**" est le résultat d'un build (images de VM, conteneur, ...) à partir d'un template (HCL2 ou JSON).
- Un "**build**" est le processus de création d'un "artifact" ou plusieurs "artifacts" à partir d'une ou plusieurs sources en utilisant des "**builders**" (pour des environnements CLOUD ou "on premise").
- Les **provisionners** sont des scripts ((Power)Shell, Ansible, ...) qui permettent de configurer les VM après leurs démarrages.
- Les **post-processors** permettent de modifier les "artifacts" après le build (compression, upload vers un registry, génération de box (vagrant) upload d'une AMI vers le Cloud) ...)

# Composants d'un fichier HCL pour Packer: les variables

Le fichier HCL contient des variables utilisables dans le script.

```
variable "memory" {  
    type    = string  
    default = "2048"  
}  
variable "cpus" {  
    type    = string  
    default = "1"  
}  
variable "disk_size" {  
    type    = string  
    default = "30000"  
}
```

# Composants d'un fichier HCL: "sources"

La source définit la ressource initiale (ici une iso) et les paramètres du build de l'image à partir de la source comme ici des arguments pour le binaire VboxManage.

```
source "virtualbox-iso" "vbox" {
  vboxmanage      = [ ["modifyvm", "{{ .Name }}", "--memory", "${var.memory}"]...
  ...
}
```

# Composants de Packer: "builders"

C'est le "main" de Packer qui lie les sources et les VM finales. Ils utilisent un "**provisionners**" (scripts, Ansible...) pour configurer les VM.

```
build {  
    sources = [ "source.qemu.qemu", "source.virtualbox-iso.vbox", "source.vmware-iso.vmware" ]  
  
    provisioner "shell" {  
        scripts = [ "scripts/debian/virtualbox-iut-bookworm.sh",  
                   "scripts/debian-12/vmware.sh", "scripts/common/sshd.sh", *  
                   "scripts/common/minimize.sh",  
                   "scripts/debian/cleanup.sh" ]  
    }  
}
```

# Retex sur Packer

- On peut avec Packer générer des images de VM personnalisés. Pour ce faire, il est **dépendant** d'autres outils ou de sources comme des repositories.
- **Plus** le build est **long** et complexe, **plus** le **risque** d'échec au build est important. Il est donc judicieux de faire des builds simples et court surtout dans un contexte de CI/CD et d'utiliser des outils de configurations comme Ansible ou de personnaliser avec Vagrant qui peut être vu comme le deuxième étage de la fusée Hashicorp.

# Retex sur Packer

- Les processus de build sont **très différents** entre une Debian ou une Ubuntu, ou une machine Windows:

Il vous faut donc **apprendre** Packer **mais aussi les spécificités de chaque distribution.** (ubiquity, preseed, kickstart, cloud-init...)

Vagrant est plus facile d'accès.

# Commandes principales de Packer

- *packer validate*: Vérifie la syntaxe du fichier de configuration.
- *packer build* : Construit une image à partir d'un fichier de configuration.
- `_packer plugins install github.com/hashicorp/virtualbox_`: Installe le plugin vbox.
- `PACKER_LOG=1 FOREGROUND=1 packer build --force debian-12-amd64-small.json.pkr.hcl` : construit avec debug une Debian 12.

# Vagrant

- Vagrant est un outil de "packaging" de VM pour le *poste de travail du développeur*.
- Il permet de créer des VM ou des containers à partir de **Boxes** (images pré-configurées d'OS générées avec Vagrant ) ou d'images Docker et de les personnaliser avec des **provisionners** (SHELL, Ansible...).
- Le fichier de configuration est un fichier **Vagrantfile** qui est un fichier **Ruby**.

# Protocoles utilisés par Vagrant

- **SSH** : Pour se connecter aux VM et les configurer.
- **WinRM** : Pour se connecter aux VM Windows et les configurer.

# Se servir d'une box Vagrant existante

```
vagrant init bento/rockylinux-8.8  
vagrant up
```

- Configurez un Vagrantfile pour utiliser la box.

```
Vagrant.configure("2") do |config|  
  config.vm.box = "bento/rockylinux-8.8"  
  config.vm.hostname = "vagrantbox"  
end
```

```
vagrant validate  
vagrant up  
vagrant status  
vagrant ssh
```

# Un exemple de vagrantfile Windows.

Ce fichier contient la configuration de la VM et les scripts de provisionning.

```
Vagrant.configure("2") do |config|
  config.vm.box = "jborean93/WindowsServer2019"
  config.vm.hostname = "win2019"
end
```

# Vagrantfile: exemples de provisionner SHELL

```
config.vm.provision "shell", inline: <<-SHELL  
  apt update  
  apt -y dist-upgrade  
  apt -y install screen htop nmap
```

```
config.vm.provision "provision-script", type: "shell", path: "provision.ps1"
```

# Vagrantfile: exemple de private network

```
config.vm.network "private_network",  
  ip: "192.168.123.111", libvirt__network_name: 'vagrant-libvirt'
```

# Principes génériques de Terraform

Le fonctionnement de Terraform est basé sur deux concepts clés :

- La **configuration** est un fichier de code qui décrit votre infrastructure. Il contient des définitions de ressources, telles que des serveurs, des bases de données, des réseaux, etc.
- L'**état** est une représentation de l'infrastructure réelle. Il est utilisé par Terraform pour comparer la configuration à l'état actuel et déterminer les modifications à apporter.

Avec Terraform comme avec d'autres outils (K8s), on déclare une cible et l'outil se charge de l'atteindre :**nul besoin de déclarer les étapes intermédiaires.**

# Les éléments de la configuration

## Terraform

- Les **ressources** sont les éléments de base de la configuration Terraform. Ce sont les composants de l'infrastructure que vous souhaitez gérer. Les ressources les plus courantes sont les serveurs, les instances, les réseaux, les sous-réseaux, les groupes de sécurité, les applications...

# Les "providers"

- Les **providers** permettent à Terraform d'interagir avec des API. Chaque fournisseur implémente des ressources et des données. Il faut indiquer à Terraform quel fournisseur utiliser pour chaque ressource.

```
terraform {  
  required_providers {  
    libvirt = {  
      source = "dmacvicar/libvirt"  
    }  
  }  
}
```

# Fichiers Terraform

**main.tf** est le fichier principal de configuration de Terraform. Il contient généralement:

- La liste des modules à importer et nécessaires pour le projet. Ils peuvent aussi être mis dans **provider.tf** .
- Les variables utilisables dans la définition des ressources. Elles peuvent être aussi mises dans **variables.tf** pour leurs déclarations et **provider.tfvars** pour leurs valeurs .
- Les définitions des ressources à déployer sont dans **main.tf** ou dans des fichiers **.tf** dans des sous-répertoires.

# Fichiers Terraform (suite)

- **terraform.tfstate** contient l'état de l'infrastructure déployée. Il est créé par Terraform et ne doit pas être modifié manuellement. Il contient l'état actuel de votre infrastructure et est utilisé pour comparer l'état actuel avec la configuration Terraform cible. En production, il est recommandé de stocker l'état dans un emplacement distant, tel qu'un compartiment S3.

# Terraform workspace

Un "wokspace" l'équivalent des branches de Git. Il permet de gérer plusieurs environnements (dev, prod, test, formation, ...) avec une seule configuration Terraform. Il est possible de créer des workspaces pour chaque environnement.

# Exemple de main.tf

# Pulumi

**C'est un concurrent de Terraform qui a pour particularité de pouvoir utiliser des langages de programmation pour décrire l'infrastructure. Il est possible d'utiliser des langages comme Python, Go, Yaml, Java, .net, typescript, javascript...) pour décrire l'infrastructure. Il est possible de déployer sur les principaux Clouds et "on premises"**

# Concepts clefs de Pulumi: Stack

Chaque programme pulumi est déployé dans un "stack". C'est un environnement (dev, prod, test, formation, ...) qui contient les ressources déployées. Il est possible de déployer plusieurs "stacks" dans un même projet. (c'est l'équivalent des workspaces de Terraform).