Rapport de Projet d'Infrastructure Cloud et Big Data
-
Automation of Spark Deployment with Ansible and Terraform

Charlotte IBAÑEZ et Emile JEANDON

Département Sciences du Numérique - Troisième année
Parcours IBDIOT
2024-2025

Table des matières

Table des figures

1 Introduction

Dans ce projet, nous allons automatiser le déploiement d'une infrastructure Big Data basée sur Apache Spark.
Pour ce faire, nous utiliserons Terraform pour mettre en place l'infrastructure sur au moins 2 hôtes physiques et Ansible pour le déploiement et la configuration de Spark.
De plus, nous utiliserons WordCount comme application de test pour valider le bon fonctionnement de l'installation.

Dans ce rapport, nous détaillerons tout d'abord le journal de bord de notre projet, en expliquant pas à pas comment nous l'avons abordé, les difficultés auxquelles nous nous sommes confrontés et les solutions que nous avons apportées. Nous décrirons ensuite l'architecture finale de notre projet en expliquant le rôle de Terraform, d'Ansible et de Spark, et le rôle des différents fichiers que nous avons. Enfin, nous expliquerons notre organisation pour ce projet, et nous étudierons les pistes d'amélioration de notre travail.

2 Journal de bord du projet

Nous avons suivi différentes étapes pour pouvoir réaliser ce projet. Tout d'abord, nous avons cherché à bien comprendre ce qui nous était demandé, afin d'être sûrs de commencer dans la bonne direction.

Le projet a été assez difficile à démarrer car nous sommes tous les deux sur MacBook, et KVM n'est pas supporté nativement sur macOS. Il nous était également impossible de faire un dualboot car la machine d'Emile n'a pas assez d'espace libre pour cela. Nous sommes donc partis sur l'idée d'utiliser VirtualBox, qui est compatible avec macOS, à la place de KVM et libvirt.

Nous avons décidé de séparer le travail en plusieurs étapes :

- Tout d'abord, générer 2 VMs à l'aide d'un script Terraform, puis faire communiquer ces deux machines virtuelles entre elles ;
- Ensuite, connecter nos 2 machines physiques pour qu'en lançant un script Terraform depuis l'une d'entre elles, 2 VMs se créent sur chaque machine physique ;
- Puis, automatiser le déploiement de Spark sur les VMs avec un playbook Ansible ;
- Et enfin, déployer l'application WordCount dans notre infrastructure.

Nous avons fait plusieurs recherches Internet pour comprendre le fonctionnement de VirtualBox pour faire le projet, et nous avons trouvé plusieurs sites intéressants desquels nous nous sommes inspirés pour commencer le projet.

Nous avons commencé par télécharger Terraform et Ansible. Puis, nous avons chacun configuré un fichier **main.tf**, en nous basant sur le provider Terraform pour VirtualBox (terra-farm) 1 car Terraform ne prend pas en charge VirtualBox par défaut. Ensuite, on télécharge un fichier ISO que l'on va utiliser pour créer les machines virtuelles Ubuntu. Après avoir eu plusieurs types d'erreur différents et après avoir modifié notre **main.tf** 1 a plusieurs reprises, en nous aidant de différents conseils lus sur Internet 2 3 4, nous avons réussi à générer des VMs avec des adresses IP.

```
terraform {
  required_providers {
    virtualbox = {
      source = "terra-farm/virtualbox"
      version = "0.2.2-alpha.1"
    }
  }
}

# There are currently no configuration options for the provider itself.

resource "virtualbox_vm" "node" {
  count     = 2
  name      = "node-${count.index + 1}"
  image     = "https://app.vagrantup.com/ubuntu/boxes/bionic64/versions/20180903.0.0/providers/virtualbox.box"
  cpus      = 2
  memory    = "512 mib"
  #user_data = file("${p")

  network_adapter {
        type = "hostonly"
        host_interface = "en0"
  }

  #network_adapter {
  #  type           = "hostonly"
  #  host_interface = "VirtualBox Host-Only Ethernet Adapter"
  #}

}

output "IPAddr" {
  value = element(virtualbox_vm.node.*.network_adapter.0.ipv4_address, 1)
}

output "IPAddr_2" {
  value = element(virtualbox_vm.node.*.network_adapter.0.ipv4_address, 2)
}
```

FIGURE 1 – Contenu du fichier main.tf avec VirtualBox

Pour initialiser et lancer Terraform, on tape les commandes :

```
terraform init
terraform apply -auto-approve
```

Puis, on vérifie que les VMs ont été créées avec la commande :

```
VBoxManage list vms
```

Lorsque l'on regarde dans le terminal, on voit que les deux machines virtuelles ont bien été créées, avec une adresse IP différente pour chacune mais appartenant au même réseau que ma machine physique 2. Cependant, il y a un problème : il m'est impossible de me connecter aux VMs par SSH. Je modifie donc mon **main.tf** pour injecter ma clé publique dans la VM, et je tape dans le terminal la commande :

```
ssh ubuntu@172.20.10.8
```

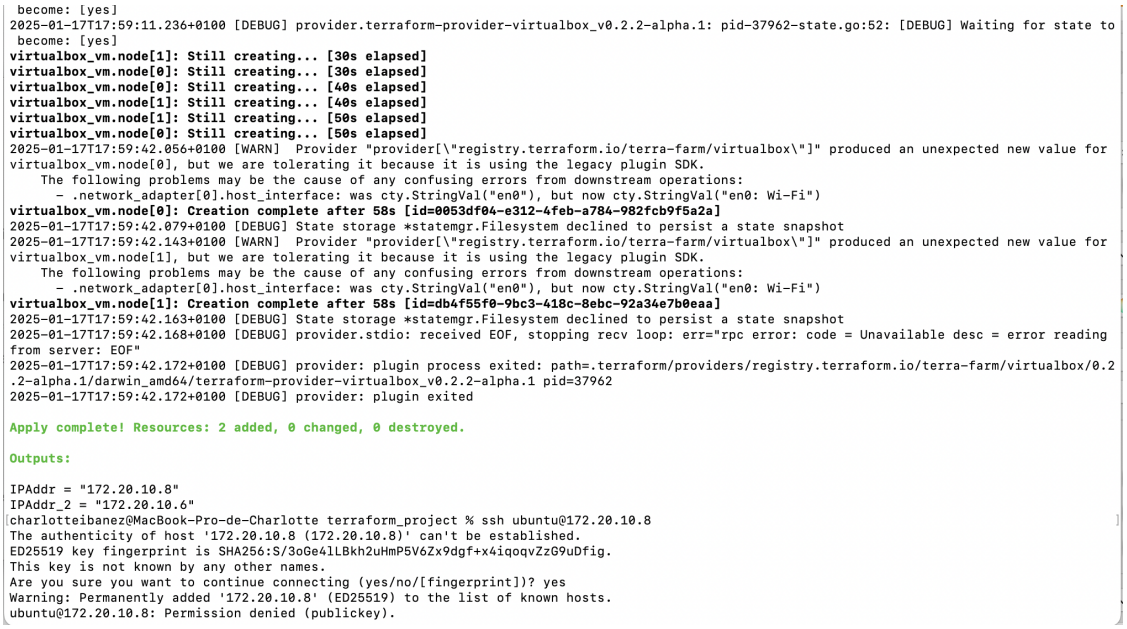

FIGURE 2 – VMs créées avec une adresse IP chacune, avec VirtualBox

Comme j'ai plusieurs nouvelles erreurs, je décide de recommencer à partir du fichier **main.tf** qui fonctionnait. Je supprime les anciens fichiers Terraform et je le relance, mais on se retrouve maintenant avec plusieurs types d'erreur que nous n'avions pas auparavant 3.

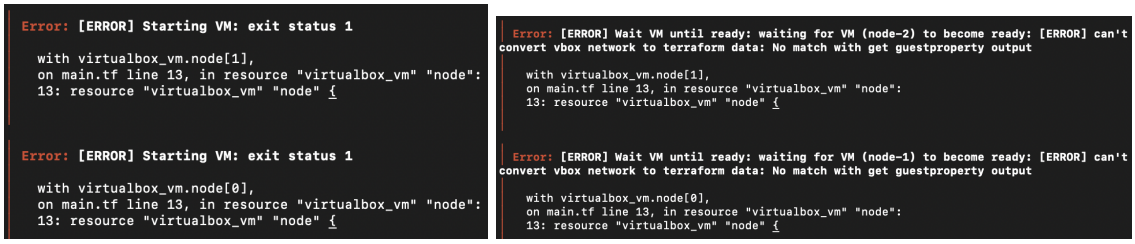

FIGURE 3 – Différents types d'erreur rencontrés avec VirtualBox

Après plusieurs jours de debug et après avoir regardé sur tous les forums possibles 5 6 7 8 9, nous n'avons pas pu faire refonctionner les VMs, même en supprimant tout et en recommençant avec uniquement le code du fichier **main.tf** qui avait fonctionné auparavant. Pourtant, lorsqu'on lance nos VMs, elles sont bien créées comme l'atteste la Figure 4. Sur Internet, nous avons trouvé que l'erreur de guestproperty est connue avec le provider terra-farm/virtualbox, qui est en version alpha 10. Nous avons donc décidé de recommencer le projet d'une autre manière car nous ne savions plus quoi faire pour avancer.

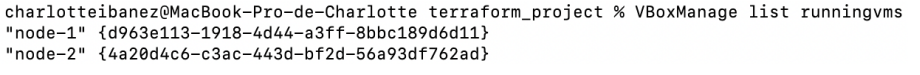

FIGURE 4 – VMs running

Ainsi, nous avons tenté de continuer le projet de deux manières différentes, chacun de notre côté, pour maximiser les chances de trouver une alternative à KVM qui fonctionne. Emile a donc décidé de faire le projet avec Docker, et Charlotte a tenté de faire le projet de manière entièrement virtuelle, en créant deux VMs avec VMware qui feront office de "machines physiques" et à partir

desquelles seront générées les VMs via Terraform. Puisque la version Docker est la version finale de notre projet, nous nous pencherons seulement sur VMware dans cette partie-là.

Avant même de tenter de faire le projet avec VMware, j'ai essayé d'utiliser des machines virtuelles avec VirtualBox mais VirtualBox ne fonctionne pas pour ce projet car avec les dernières mises à jour de macOS, Apple ne rend pas accessible l'hyperviseur créé et je ne peux donc pas faire de virtualisation imbriquée, même en ayant activé le paramètre Enable Nested VT-x/AMD-V.

Avec VMware, l'objectif est de faire de la nested virtualization qui permet d'exécuter un hyperviseur (comme KVM) à l'intérieur d'une VM. Tout d'abord, il faut créer les deux machines virtuelles, et on utilise pour cela un disque Ubuntu. Il faut ensuite correctement configurer la VM et télécharger à nouveau les logiciels requis pour faire le projet, à savoir KVM, libvirt, Terraform et Ansible.

La gestion des VMs sous VMware a été un peu compliquée car il m'était impossible de faire fonctionner le copier/coller, alors qu'il était pourtant autorisé dans les paramètres. Il fallait donc tout écrire, modifier et tester à la main ce qui a été assez long à faire. Après pas mal de temps de recherche (11 12 13 14 15), nous avons enfin réussi à faire fonctionner Terraform sur l'une des VMs avec le code suivant 5 :

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

provider "libvirt" {
  uri = "qemu:///system"
}

resource "libvirt_volume" "ubuntu_disk1" {
  name   = "ubuntu_disk1"
  pool   = "default"
  source = "ubuntu-22.04-server-cloudimg-amd64.img"
  format = "qcow2"
}

resource "libvirt_volume" "ubuntu_disk2" {
  name   = "ubuntu_disk2"
  pool   = "default"
  source = "ubuntu-22.04-server-cloudimg-amd64.img"
  format = "qcow2"
}

resource "libvirt_network" "custom_network" {
  name = "custom_network"
  mode = "nat"
  domain = "local"
  addresses = ["192.168.100.0/24"]
}
```

```
resource "libvirt_domain" "vm1" {
  name = "vm1"
  memory = 2048
  vcpu = 2

  disk {
    volume_id = libvirt_volume.ubuntu_disk1.id
  }
  network_interface {
    network_id = libvirt_network.custom_network.id
    hostname = "vm1"
    addresses = ["192.168.100.101"]
  }
  console {
    type = "pty"
    target_port = "0"
  }
#output "ips" {
  #value = libvirt_domain.domain-ubuntu.*.network_interface.0.addresses
}

resource "libvirt_domain" "vm2" {
  name = "vm2"
  memory = 2048
  vcpu = 2

  disk {
    volume_id = libvirt_volume.ubuntu_disk2.id
  }
  network_interface {
    network_id = libvirt_network.custom_network.id
    hostname = "vm2"
    addresses = ["192.168.100.102"]
  }
  console {
    type = "pty"
    target_port = "0"
  }
}
```

FIGURE 5 – main.tf fonctionnel avec KVM

Dans un premier temps, on déclare les configurations spécifiques à Terraform. Terraform nécessite un provider pour pouvoir interagir avec VMware, et *required_providers* donne le provider nécessaire pour le déploiement, libvirt, avec pour source "dmacvicar/libvirt" 16.
Ensuite, dans le bloc *resource "libvirt_volume"*, on crée deux disques virtuels (ubuntu_disk1 et ubuntu_disk2) pour les VMs que l'on va générer. On précise donc leur nom, leur pool de stockage, le fichier image du disque source, et le format du disque.
Le bloc *resource "libvirt_network"* sert à créer un réseau virtuel NAT pour les futures VMs. On lui attribue donc un nom, un mode (ici NAT), un domaine, et une plage d'adresses IP pour le réseau (198.168.100.0/24).

Enfin, le bloc *resource "libvirt_ domain"* nous permet de créer les machines virtuelles. Ici, on en crée deux pour tester leur déploiement et leur connectivité. On indique donc le nom de la VM, la mémoire qu'on lui alloue, le nombre de vCPU et le volume disk associé à la VM. Ensuite, on définit l'interface réseau et on précise l'adresse IP statique associée à la VM (192.168.100.101 pour la première et 192.168.100.102 pour la seconde).

A la fin du fichier **main.tf**, on ajoute des output pour vérifier que l'adresse IP attribuée à chaque machine, ainsi que son nom, s'affichent bien 6.

```
output "vm1_ip" {
   value = libvirt_domain.vm1.network_interface.0.addresses[0]
}

output "vm1_name" {
   value = libvirt_domain.vm1.name
}


output "vm2_ip" {
   value = libvirt_domain.vm2.network_interface.0.addresses[0]
}

output "vm2_name" {
   value = libvirt_domain.vm2.name
}
```

FIGURE 6 – Blocs output du main.tf avec KVM

Maintenant que le fichier est correctement configuré, on lance Terraform :

```
terraform init
terraform apply -auto-approve
```

Comme on peut le voir sur la Figure 7, nos VMs sont bien créées et elles ont chacune une adresse IP unique.

```
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

vm1_ip = "192.168.100.101"
vm1_name = "vm1"
vm2_ip = "192.168.100.102"
vm2_name = "vm2"
root@ubuntu:/home/ubuntu# virsh list --all
 Id   Name   State
----------------------
 3    vm2    running
 4    vm1    running
```

FIGURE 7 – VMs running avec KVM

Cependant, il y a un problème lorsque je tape la commande :

```
virsh domifaddr vm1
```

Rien ne s'affiche, comme s'il y avait un problème avec la configuration de l'interface réseau. Autre problème, je ne peux même pas accéder aux VMs, même avec la commande :

```
virsh console vm1
```

Après plusieurs recherches pour comprendre nos différentes erreurs (17 18 19 20) et après en avoir discuté avec l'un de nos camarades qui a rencontré le même problème, il s'avère qu'il y a un problème dans un fichier de configuration grub. Il ne nous est pas possible de modifier ce fichier de configuration, et nous ne pourrons donc jamais régler les problèmes auxquels nous faisons face. Cela ne sert donc à rien de continuer à faire le projet entièrement en virtuel, car cela ne fonctionne pas et est bien trop dur à gérer.

3 Architecture du projet

3.1 Introduction

Puisque nous avions testé toutes les options qui s'offraient à nous, et comme nous ne pouvions pas installer de logiciels sur les PC de l'ENSEEIHT, avons décidé d'utiliser Docker pour réaliser le projet. Nous avons choisi Docker pour éviter les problèmes que nous avions avec les autres providers, mais finalement nous avons fait face à d'autres problèmes que nous n'aurions pas eu si nous avions été sous Linux, car ce sont des problèmes inhérents à l'architecture de Docker sur Mac.

De prime abord, l'utilisation de Docker pourrait sembler être une solution un peu trop facile, mais nous avons tenu à faire le projet comme nous l'aurions fait avec KVM, sans profiter des outils Docker mais en faisant tout à la main. Ainsi, au lieu d'utiliser une image préconçue adaptée au déploiement de Spark ou bien au lieu de passer par un dockerfile, nous avons choisi une image Ubuntu vierge pour déployer nos containers. Ils sont donc traités exactement comme le seraient des VMs, en utilisant ensuite des playbooks Ansible pour les setups (mettre en place les configurations ssh, récupérer les dépendances nécessaires, récupérer le code du WordCount, etc.). Dans cette partie, nous présenterons notre code en détails, en expliquant les difficultés auxquelles nous avons fait face.

3.2 Première solution trouvée

Nous sommes tout d'abord partis sur une première solution dans laquelle on attribue des adresses IP à chaque container. Cette solution n'a pas pu aboutir à cause de l'infrastructure forcée par Docker sur Mac, qui utilise une VM pour créer les containers, ce qui nous empêche de faire un routage correctement. Nous allons tout de même présenter le travail accompli dans cette partie, notamment les fichiers **main.tf** et **inventory.ini** qui diffèrent de ceux utilisés dans notre projet final.

Intéressons nous au fichier **main.tf** du dossier Terraform 8. Tout d'abord, on déclare le provider utilisé, "kreuzwerker/docker" 21. Dans un second temps, on va créer l'image Docker avec "rastasheep/ubuntu-sshd" qui fournit un userspace Ubuntu complet avec un accès SSH avec root, comme sur une VM classique.

```
# Créer un réseau Docker partagé
resource "docker_network" "shared_network1" {
  name = "shared_network1"
  driver = "bridge"
  ipam_config {
    subnet = "172.16.0.0/16"
  }
  count = var.host == "localhost" ? 1 : 0
}

resource "docker_network" "shared_network2" {
  name = "shared_network2"
  driver = "bridge"
  ipam_config {
    subnet = "172.15.0.0/16"
  }
  count = var.host == "remote_host" ? 1 : 0
}

# Déployer les conteneurs pour localhost
resource "docker_container" "slave-1" {
  image = docker_image.ubuntu.image_id
  name  = "slave-1"
  restart="unless-stopped"
  ports {
    internal = 22
    external = 8081
  }
  networks_advanced {
    name = docker_network.shared_network2[0].name
    ipv4_address="172.15.0.11"
  }
  count = var.host == "remote_host" ? 1 : 0
}
```

FIGURE 8 – Docker : fichier main.tf de la solution 1

Ensuite, on va créer deux réseaux bridge : **shared_network1** qui a pour plage d'adresses IP **172.16.0.0/16** et qui est attribué à la machine d'Emile, et **shared_network2** qui a pour plage d'adresses IP **172.15.0.0/16** et qui est attribué à la machine de Charlotte. Nous avons choisi ces sous-réseaux pour éviter les collisions avec les plages Docker standard (172.17.0.0/16).
Puis, on déploie 4 containers : **master** et **slave-3** sur **shared_network1** (la machine d'Emile), et **slave-1** et **slave-2** sur **shared_network2** (la machine de Charlotte). Pour simuler un réseau physique, on attribue des adresses IP statiques aux containers que l'on déploie. Par exemple, le container **slave-1** a pour adresse IP **172.15.0.11**. On lui attribue également plusieurs ports :

- Port 22 (internal) : Il s'agit du port SSH standard à l'intérieur du conteneur. Le service SSH écoute ici grâce à l'image ubuntu-sshd.
- Port 8081 (external) : Docker mappe ce port de la machine physique de Charlotte vers le port 22 du conteneur.

L'attribution des ports joue un rôle essentiel pour la communication avec les containers Docker, et notamment pour la connexion via SSH.

Lorsque l'on regarde le container **master** 9 déployé sur la machine physique d'Emile, on remarque qu'il y a un double mapping de ports. En effet, il y a également le port 54310 qui est utilisé pour Spark (comme nous l'avons fait en TP).

```
resource "docker_container" "master" {
  image = docker_image.ubuntu.image_id
  name  = "master"
  restart="unless-stopped"
  ports {
    internal = 22
    external = 8084
  }
  ports{
    internal = 54310
    external = 54310
  }
  networks_advanced {
    name = docker_network.shared_network1[0].name
    ipv4_address="172.16.0.10"
  }
  count = var.host == "localhost" ? 1 : 0
}
```

FIGURE 9 – Docker : fichier main.tf de la solution 1 - Container master

Regardons ensuite le fichier **inventory.ini** qui définit l'inventaire Ansible 10. Précisons tout de même que pour mener à bien ce projet, nos deux machines physiques étaient connectées sur le même réseau, via un partage de connexion.
Le groupe **[remote_host]** cible la machine physique de Charlotte (adresse IP 172.20.10.7) qui héberge une partie des containers. Les groupes **[master_container]** et **[slave_containers]** définissent les containers Docker avec leurs adresses IP statiques (172.16.0.10 pour le master, 172.15.0.11, 172.15.0.12 et 172.16.0.13 pour les slaves), répartis sur nos deux sous-réseaux distincts (**172.16.0.0/16** et **172.15.0.0/16**).

Cependant, cette configuration ne peut pas fonctionner. En effet, les adresses IP des containers ne sont pas routables depuis un hôte macOS, car elles sont encapsulées dans la VM HyperKit de Docker. Les groupes **master_container** et **slave_containers** sont donc injoignables via Ansible.

La tentative de configuration SSH échoue donc, car les containers des deux sous-réseaux ne peuvent pas communiquer entre eux, et les IPs que l'on a déclaré ne correspondent pas à la topologie imposée par Docker sur macOS.

```
[localhost]
localhost ansible_connection=local

[remote_host]
172.20.10.7 ansible_user=charlotteibanez

[master_container]
master ansible_host=172.16.0.10 ansible_user=root ansible_python_interpreter=/usr/bin/python3

[slave_containers]
slave-3 ansible_host=172.16.0.13 ansible_user=root ansible_python_interpreter=/usr/bin/python3
slave-1 ansible_host=172.15.0.11 ansible_user=root ansible_python_interpreter=/usr/bin/python3
slave-2 ansible_host=172.15.0.12 ansible_user=root ansible_python_interpreter=/usr/bin/python3

[containers:children]
master_container
slave_containers
```

FIGURE 10 – Docker : fichier inventory.ini de la solution 1

Ainsi, notre première solution, s'est heurtée à une contradiction entre l'abstraction Docker et les exigences réseau du projet. En effet, Docker Desktop (sur macOS) exécute tous les containers dans une seule VM HyperKit Linux, ce qui annule notre tentative de séparation physique. Ainsi, les réseaux 172.16.0.0/16 (machine physique d'Emile) et 172.15.0.0/16 (machine physique de Charlotte) cohabitent en réalité dans la même VM.

3.3 Seconde solution

Nous allons maintenant présenter une autre solution que nous avons mis en place, dans laquelle le routage se fait par le forwarding de ports entre les machines physiques et les containers.

Avant de mettre en place cette solution, nous avons tenter de créer un réseau overlay de type Swarm/Weave, mais cela ne fonctionne pas sur environnement macOS 22.

Ensuite, nous avons tenté d'utiliser une solution dédié à macOS permettant l'accès au réseau virtuel de containers depuis l'hote physique et facilitant le routage. Le problème est que l'utilisation de routes particulières entraine des conflits d'adresses entre 2 Macs 23.

Regardons donc ce que nous avons fait dans cette seconde solution. Dans le fichier **main.tf** 11, à la différence de ce que nous avions fait dans la solution précédente et qui ne fonctionnait pas, on a maintenant une infrastructure homogène sur nos deux machines physiques (localhost pour la machine d'Emile, et remote_host pour la machine de Charlotte), en utilisant un réseau unique (**shared_network**) pour tous nos containers. L'image "rastasheep/ubuntu-sshd" est utilisée comme base pour tous les containers que l'on va créer.

La logique de déploiement est conditionnée par la variable host. Sur le localhost, Terraform va créer le container **master** (port SSH 22 mappé sur 8084, et port Spark 54310 exposé) et **slave-3** (SSH mappé sur 8083). Sur le remote_host, il déploie **slave-1** (SSH sur 8081) et **slave-2** (SSH sur 8082). Contrairement à la version précédente, tous les containers sont attachés au même réseau bridge, et on attribue automatiquement les IPs via DHCP.

```
# Créer un réseau Docker partagé
resource "docker_network" "shared_network" {
  name = "shared_network"
  driver = "bridge"
}

# Déployer les conteneurs pour localhost
resource "docker_container" "slave-1" {
  image = docker_image.ubuntu.image_id
  name  = "slave-1"
  restart="unless-stopped"
  ports {
    internal = 22
    external = 8081
  }
  networks_advanced {
    name = docker_network.shared_network.name
  }


  count = var.host == "remote_host" ? 1 : 0
}

resource "docker_container" "slave-2" {
  image = docker_image.ubuntu.image_id
  name  = "slave-2"
  restart="unless-stopped"
  ports {
    internal = 22
    external = 8082
  }
  networks_advanced {
    name = docker_network.shared_network.name
  }
  count = var.host == "remote_host" ? 1 : 0
}
```

FIGURE 11 – Docker : fichier main.tf de la solution 2

Le fichier **inventory.ini** est lui aussi différent dans cette seconde solution 12.
Désormais, le groupe **[remote_host]** cible une machine distante via son IP (ip_de_lhote_distant) et un utilisateur SSH (username_de_lhote_distant), contrairement à avant où l'adresse IP et l'utilisateur SSH étaient directement donnés. Les groupes **[master_container]** et **[slave_containers]** utilisent désormais les adresses IP des machines physiques, combinées aux ports forwardés pour accéder aux containers :

- **master** est joint via ip_local :8084 (port SSH mappé du container)
- les **slaves** utilisent ip_local :8083 (slave-3) ou ip_remote :8081-8082 (slave-1 et slave-2).

```
[localhost]
localhost ansible_connection=local

[remote_host]
ip_de_lhote_distant ansible_user=username_de_lhote_distant

[master_container]
master ansible_host=ip_local ansible_port=8084 ansible_user=root ansible_python_interpreter=/usr/bin/python3

[slave_containers]
slave-3 ansible_host=ip_local ansible_port=8083 ansible_user=root ansible_python_interpreter=/usr/bin/python3
slave-1 ansible_host=ip_remote ansible_port=8081 ansible_user=root ansible_python_interpreter=/usr/bin/python3
slave-2 ansible_host=ip_remote ansible_port=8082 ansible_user=root ansible_python_interpreter=/usr/bin/python3

[containers:children]
master_container
slave_containers
```

FIGURE 12 – Docker : fichier inventory.ini de la solution 2

Maintenant, notre nouvelle architecture repose sur une abstraction en deux couches. On a d'un côté la couche physique (Terraform) qui déploie des containers sur deux machines distinctes et expose les ports SSH des containers sur des ports uniques de chaque machine physique. De l'autre côté, on a la couche logique (Ansible) dans laquelle on traite les containers comme des entités réseau indépendantes, accessibles via les IP/ports des hôtes, et on masque la complexité Docker que l'on a pu voir précédemment en utilisant le SSH sur les ports forwardés. Cette approche contourne les limitations de Docker sur macOS : au lieu de tenter de router entre sous-réseaux Docker, on utilise les hôtes physiques comme relais SSH. Le réseau shared_network donc la communication interne entre containers d'un même hôte, qui est essentielle pour Spark.

Regardons comment fonctionnent les différents fichiers Ansible que nous avons créés. Le fichier **deploy.yml** est un playbook qui automatise le déploiement de notre infrastructure Terraform sur nos deux machines physiques (localhost et remote_host). Pour chaque hôte, il exécute une séquence identique mais paramétrée différemment :

Tout d'abord, on prépare l'environnement, puis on injecte la variable *host="localhost"* ou *host="remote_host"* dans terraform.tfvars, ce qui va déterminer les containers à créer (master/slave-3 sur la machine physique locale (Emile), slave-1/slave-2 sur la machine physique distante (Charlotte)). Ensuite, on initialise le projet et on déploie déploie les containers avec les commandes :

```
terraform init
terraform apply -auto-approve
```

La particularité de ce fichier est qu'il fait un traitement parallèle des deux machines physiques. En effet, Ansible orchestre en même temps le déploiement local (sur la machine d'Emile) et distant (sur la machine de Charlotte).

Regardons ensuite le fichier **ssh.yml**, qui est un playbook qui permet au master d'accéder en SSH aux slaves en distribuant sa clé publique 24.

Tout d'abord, sur le localhost, on génère une clé RSA dans le container master. On récupère la clé publique générée et on l'ajoute aux **authorized_keys** du master et du slave-3, ce qui permet de faire un SSH sans mot de passe entre eux. Ensuite, on sauvegarde la clé publique dans /tmp/master_id_rsa.pub sur l'hôte local pour la partager avec l'hôte distant.
Maintenant, sur le remote_host, on copie la clé publique du master depuis l'hôte local vers l'hôte distant, et on injecte cette clé dans les **authorized_keys** des containers slave-1 et slave-2, ce qui nous permet d'autoriser le master à s'y connecter en SSH.
Ainsi, une seule clé est générée (dans le master) et est propagée à tous les slaves.

Le fichier **ssh2.yml** permet à la machine qui déploie les playbooks d'accéder en SSH aux containers en distribuant sa clé publique, ce qui est nécessaire à l'éxecution de playbooks directement sur les containers.

Le playbook lit la clé publique existante de l'utilisateur courant (/.ssh/id_rsa.pub) pour l'injecter dans les containers locaux (master et slave-3), ce qui permet à l'hôte physique de se connecter en SSH directement aux containers via la commande :

```
ssh root@localhost -p <port>
ssh root@localhost -p 8084  (pour accèder au master)
```

Ensuite, on sauvegarde la clé dans /tmp/localhost_id_rsa.pub pour la transmettre à l'hôte distant. La clé publique est copiée depuis la machine physique vers l'hôte distant (/tmp/localhost_id_rsa.pub), puis elle est injectée dans les containers distants (slave-1 et slave-2). Ainsi, on peut exécuter des commandes SSH directement dans les containers depuis notre terminal.

Le playbook précédent (**ssh.yml**) établissait une confiance inter-containers (entre le master et les slaves), et maintenant ce playbook permet d'établir une confiance entre la machine physique et les containers.

Penchons-nous maintenant sur le playbook **ssh3.yml** qui finalise la mise en place d'Ansible en mettant à jour Python dans les containers, afin de pouvoir déployer des playbooks directement sur ces derniers.
Tout d'abord, on désactive la vérification des clés d'hôte SSH (StrictHostKeyChecking=no) pour nos containers. Ensuite, on met à jour les paquets et on installe Python 3.8 dans les containers locaux (master et slave-3). La commande update-alternatives force l'utilisation de Python 3.8 comme version par défaut, car elle est essentielle pour Ansible qui dépend de Python 3.
On réplique l'installation de Python 3.8 sur les containers distants (slave-1 et slave-2), et on copie le fichier /.ssh/config de la machine locale vers /root/.ssh/config du container master. Cela permet de définir des alias SSH ou des paramètres réseau pour faciliter les connexions entre les noeuds du cluster.

Pour terminer, nous avons le fichier **setup.yml**, un playbook qui vise à mettre en place l'environnement pour le déploiement du WordCount. Il automatise l'installation d'un stack Big Data (JDK 8, Hadoop 2.7.1, Spark 2.4.3) sur nos containers Ubuntu.

3.4 Déploiement du projet

Lorsqu'on lance nos playbooks, cela fonctionne bien car toutes les étapes que nous avons détaillées précédemment se font correctement 13.

```
(base) emilejeandon@Air-de-Emile ansible % ansible-playbook -i inventory.ini deploy.yml
[WARNING]: Found both group and host with same name: localhost

PLAY [Déployer des conteneurs Docker avec Terraform sur macOS] *****************

TASK [Gathering Facts] *********************************************************
[WARNING]: Platform darwin on host localhost is using the discovered Python
interpreter at /usr/local/bin/python3.13, but future installation of another
Python interpreter could change the meaning of that path. See
https://docs.ansible.com/ansible-
core/2.18/reference_appendices/interpreter_discovery.html for more information.
ok: [localhost]

TASK [Vérifier si Terraform est installé] **************************************

changed: [localhost]

TASK [Supprimer le dossier terraform] ******************************************
changed: [localhost]

TASK [Copier les fichiers Terraform] *******************************************
changed: [localhost]

TASK [Supprimer les fichiers .*] ***********************************************
changed: [localhost]

TASK [variable host] ***********************************************************
changed: [localhost]

TASK [Initialiser Terraform] ***************************************************
changed: [localhost]

TASK [Appliquer la configuration Terraform] ************************************
```

FIGURE 13 – Résultats dans le terminal du déploiement de notre projet

Cela peut se verifier en tapant la commande :

```
docker ps
```

En effet, lorsque l'on tape cette commande dans le terminal de la machine physique distante, on remarque que slave-1 et slave-2 sont bien actifs, et que le forwarding de ports associé est correct 14.

```
charlotteibanez@MacBook-Pro-de-Charlotte ~ % docker ps
CONTAINER ID   IMAGE          COMMAND               CREATED          STATUS          PORTS                    NAMES
8bb16eaffa15   49533628fb37   "/usr/sbin/sshd -D"   21 seconds ago   Up 19 seconds   0.0.0.0:8081->22/tcp   slave-1
772e051cd290   49533628fb37   "/usr/sbin/sshd -D"   21 seconds ago   Up 19 seconds   0.0.0.0:8082->22/tcp   slave-2
```

FIGURE 14 – Docker ps sur la machine physique distante

Ainsi, comme nous avons pu vous le montrer lors de notre présentation, nous parvenons bien à créer toute l'architecture de ce projet, avec deux machines physiques qui déploient chacune deux machines virtuelles, sur lesquelles on installe tout l'environnement nécessaire au déploiement du WordCount. On a une automatisation complète et une orchestration multi-cloud simulée avec Terraform (localhost + remote_host), ainsi qu'une configuration SSH et un déploiement logiciel unifié via Ansible.

En revanche, il nous est impossible de réussir à déployer le WordCount sur tous les containers à cause de l'architecture même de Docker. Grâce à nos modifications entre la première solution et la solution finale, nous avons pu améliorer la partie réseau en créant un réseau unique. Cependant, le problème fondamental reste le même : Docker sur macOS ne permet pas de simuler de vrais hôtes physiques avec un réseau routable. Spark nécessite une connectivité IP directe entre tous les noeuds, ce qui est incompatible avec l'isolation des réseaux Docker et la couche NAT d'HyperKit.

Même avec un réseau unique, Docker Desktop sur macOS exécute tous les containers dans une VM HyperKit isolée et utilise un NAT masqué pour connecter les containers à l'hôte physique. Par

conséquent, les containers sont joignables entre eux (via leurs IPs Docker), mais les ports exposés ne sont accessibles que depuis l'hôte macOS, pas entre containers. Dans la VM HyperKit, aucun mapping de port n'existe entre containers.

Par exemple, pour Spark, le master doit être joignable via son adresse IP Docker (ex : 172.18.0.2 :54310), mais les slaves étant dans le même réseau, ils ne peuvent pas utiliser le NAT de l'hôte.

Ainsi, nous avons pu résoudre le problème initial d'isolation entre sous-réseaux, mais le coeur du problème reste l'implémentation réseau de Docker sur macOS. Notre échec sur WordCount est donc une conséquence logique des contraintes macOS/Docker.

4 Conclusion

En conclusion, ce projet d'automatisation d'un cluster Spark avec Terraform et Ansible a été une aventure riche en défis et en apprentissages. Malgré notre motivation et les nombreuses heures passées à debugger, nous n'avons pas réussi à faire fonctionner l'application WordCount.

Dans un premier temps, nos tentatives avec VirtualBox et KVM ont été frustrantes car nous y avons passé beaucoup de temps, alors qu'il n'était en fait pas possible de réaliser le projet de cette façon.

Lorsque nous avons basculé sur Docker, nous pensions que créer des containers VM-like suffirait, mais nous avons découvert une incompatibilité fondamentale entre l'isolation réseau imposée par la VM HyperKit et les exigences de communication directe d'Apache Spark. Malgré nos tentatives de contournement (routage via des ports forwardés et configuration SSH hybride), il n'était pas possible d'établir une véritable communication directe entre noeud, et nous n'avons donc pas pu déployer le WordCount.

Références

[1] https ://github.com/terra-farm/terraform-provider-virtualbox
[2] https ://www.roksblog.de/terraform-virtualbox-provider-terrafarm/
[3] https ://blog.opennix.ru/posts/use-terraform-with-virtualbox/
[4] https ://kb.cloudtada.com/devops/English/day59/
[5] https ://github.com/terra-farm/terraform-provider-virtualbox/issues/138
[6] https ://github.com/terra-farm/terraform-provider-virtualbox/issues/112
[7] https ://github.com/terra-farm/terraform-provider-virtualbox/issues/34
[8] https ://www.virtualbox.org/manual/UserManual.html
[9] https ://docs.oracle.com/en/virtualization/virtualbox/6.0/user/vboxmanage-guestproperty.html

[10] https ://github.com/terra-farm/terraform-provider-virtualbox/issues/149
[11] https ://blog.stephane-robert.info/docs/infra-as-code/provisionnement/terraform/premiere-infra/

[12] https ://danstechjourney.com/posts/kvm-terraform/
[13] https ://computingforgeeks.com/how-to-provision-vms-on-kvm-with-terraform/
[14] https ://github.com/a-mt/terraform-kvm
[15] https ://emrah-t.medium.com/provisioning-automation-with-terraform-on-libvirt-and-kvm-ba6fa177ffea

[16] https ://github.com/dmacvicar/terraform-provider-libvirt
[17] https ://github.com/dmacvicar/terraform-provider-libvirt/issues/978
[18] https ://stackoverflow.com/questions/70230848/kvm-virsh-the-storage-pool-is-empty
[19] https ://github.com/simon3z/virt-deploy/issues/8issuecomment-73111541
[20] https ://linustechtips.com/topic/1427164-no-input-using-virt-manager-running-macos-simple-kvm/

[21] https ://github.com/kreuzwerker/terraform-provider-docker
[22] https ://github.com/moby/swarmkit/issues/1146issuecomment-231412874
[23] https ://github.com/chipmk/docker-mac-net-connect/blob/main/README.mddocker-mac-net-connect

[24] https ://phoenixnap.com/kb/ssh-permission-denied-publickey