



Despliegue de Aplicaciones: Máquinas virtuales

Evolución y Gestión de la Configuración

Paquetes (deb, msi, rpm...)

VirtualEnv

Contenedores

VM

Permite tener “instalaciones” de módulos y paquetes Python de manera simultanea

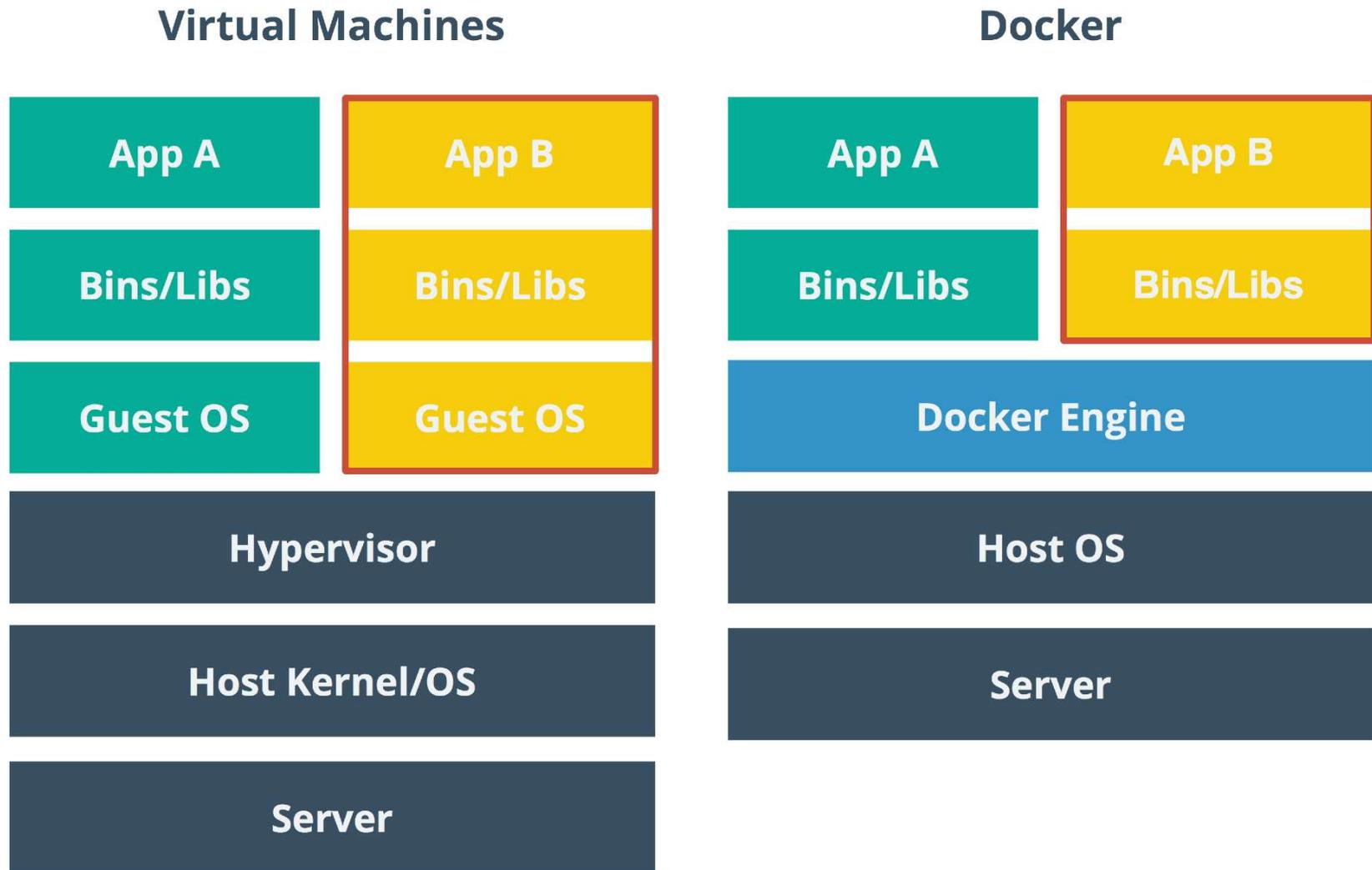
Con Contenedores aislamos dependencias más allá de python

Permiten aislar todas las dependencias del sistema

Overhead y aislamiento

INTRODUCCIÓN A MAQUINAS VIRTUALES

VM vs contenedores



Contenedores

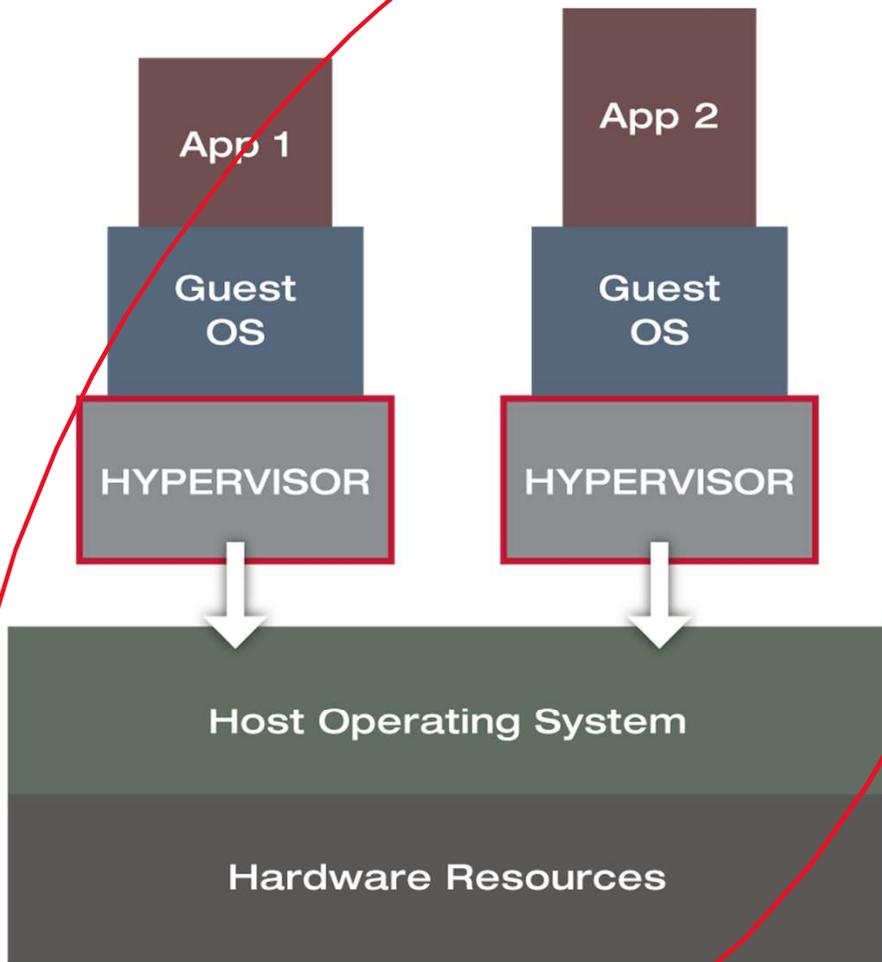


Máquinas virtuales

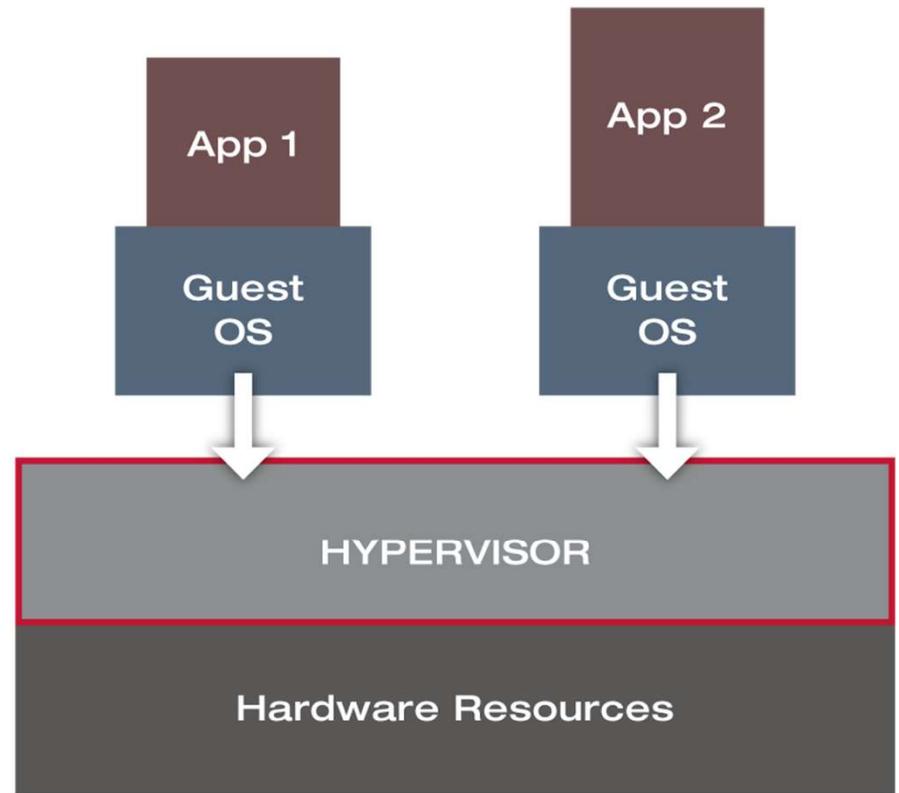
- Aislamiento parcial del sistema operativo host. Menos seguro con respecto a ataques.
- Ejecuta el mismo núcleo que el host. E.g. un host Windows soporta contenedores Windows.
- Actualiza el Dockerfile, genera una nueva imagen, sube de nuevo al host de imágenes

- Aislamiento completo del sistema operativo. Más seguro con respecto a ataques a la infraestructura.
- Ejecuta cualquier sistema operativo como invitado.
- Cuando actualizamos, necesitamos descargar e instale las actualizaciones del sistema operativo en cada VM. Instalar una nueva versión del sistema operativo requiere actualizar o, a menudo, sólo crear una VM completamente nueva. Esto puede llevar mucho tiempo, especialmente si tiene muchas máquinas virtuales.
- Compartir archivos mediante protocolos de red

Tipos de hipervisores

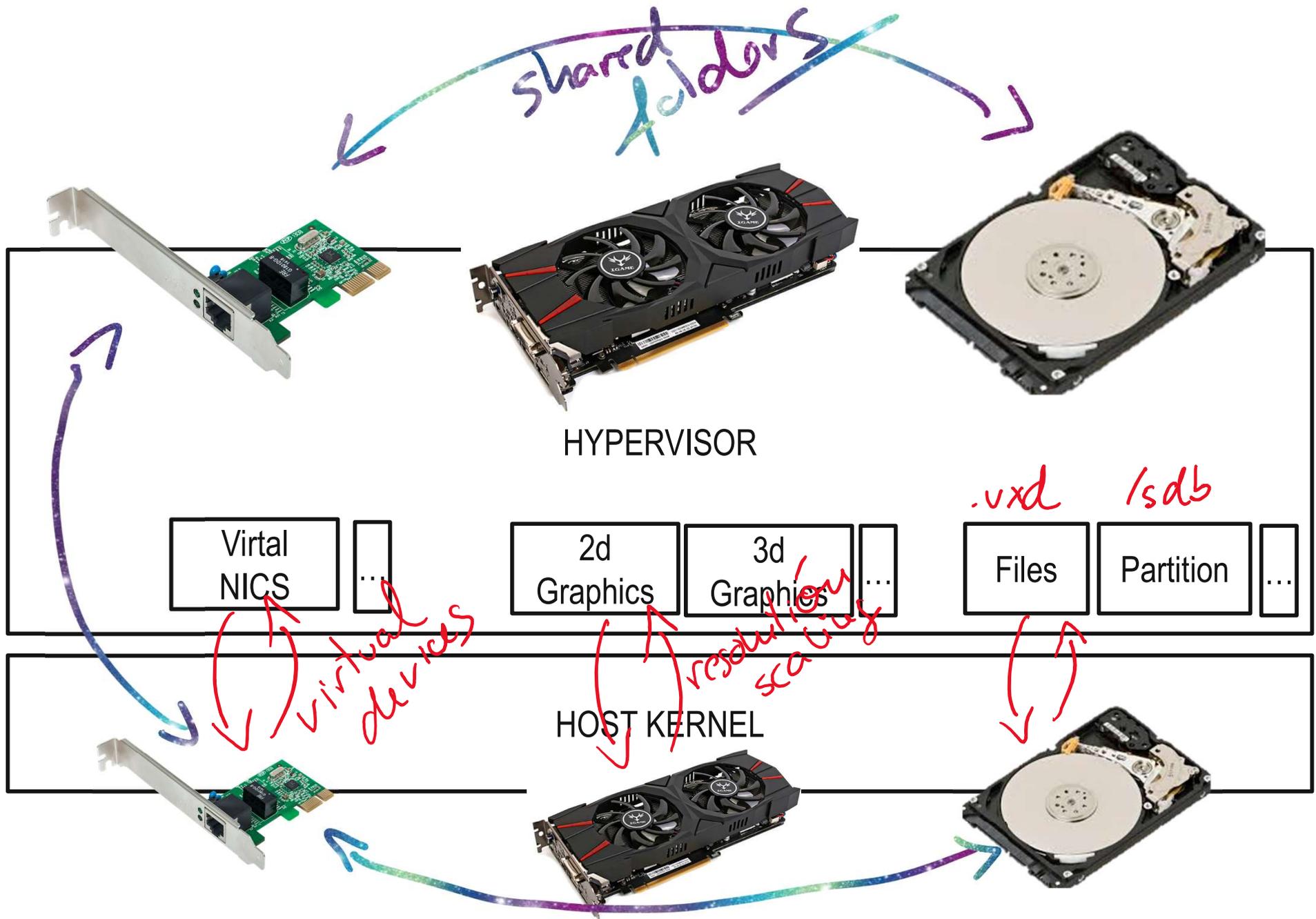


TYPE-2 HYPERVISOR

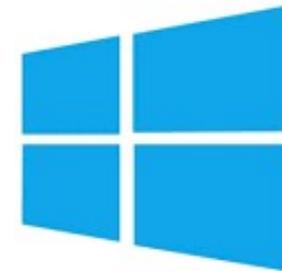


TYPE-1 HYPERVISOR

Se necesitan implementar drivers virtuales para todos los dispositivos



Modelos comerciales de hypervisores tipo 2



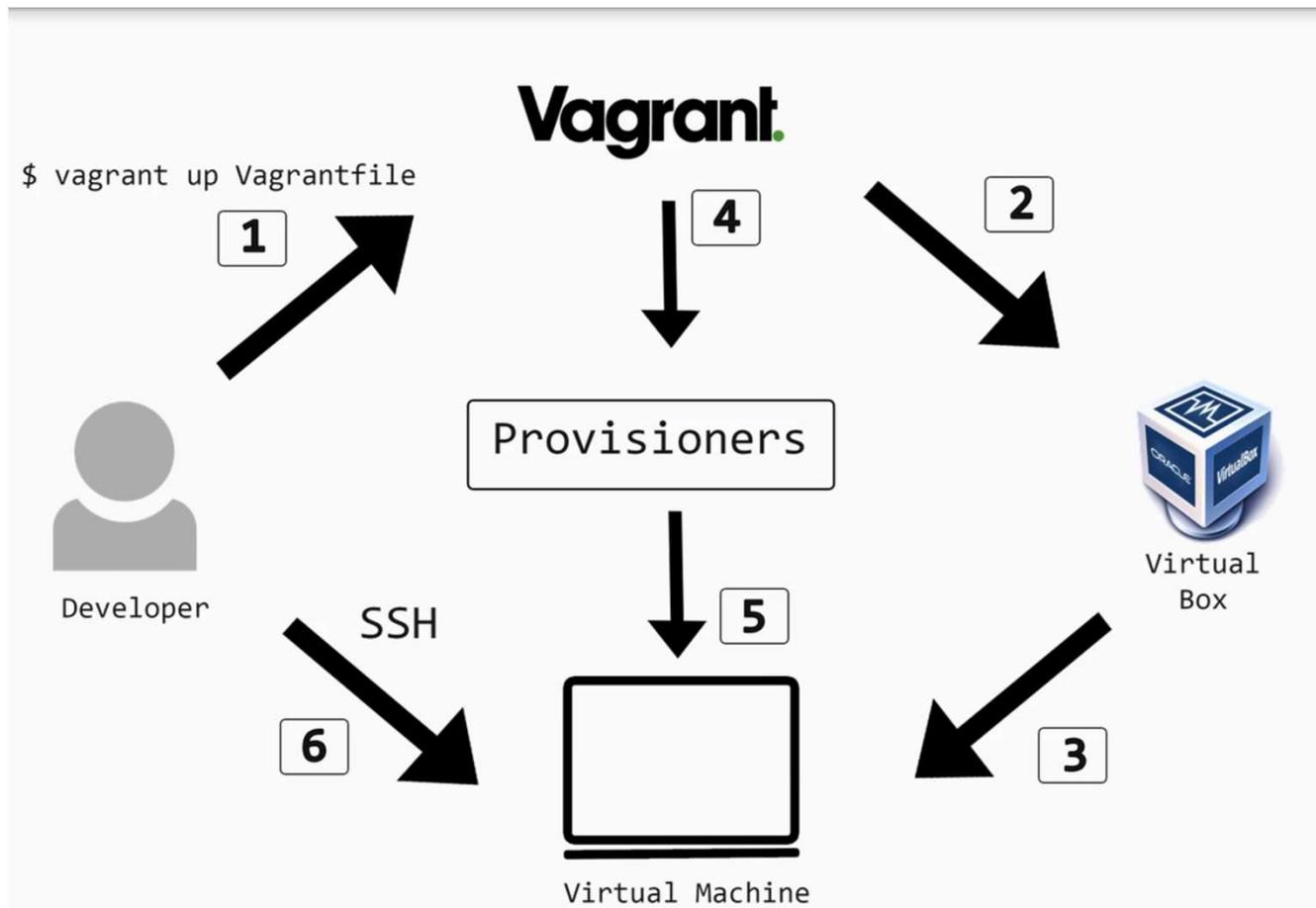
Microsoft
Hyper-V

vmware®

GESTORES DE VM (soportando distintos hipervisores)



VAGRANT WORKFLOW



- Backend of Vagrant
- VirtualBox
- VMware
- Hyper-V
- vCloud
- AWS

Hay dos etapas principales

- Primera etapa. Creación de la vm en el hipervisor
 - Configuración de red
 - Discos duros
 - Drivers gráficos
- Segunda etapa. Aprovisionamiento del software
 - Script sh de instalación
 - Ansible

PRIMEROS PASOS

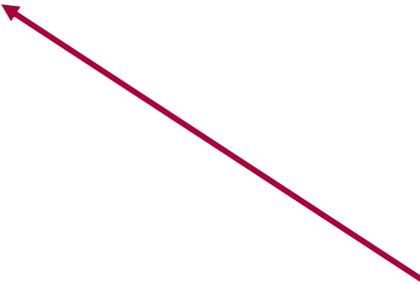
Nuestro “hello world” con Vagrant

```
> vagrant init ubuntu/trusty32
```

Lanzar una imagen



Nombre de la imagen



Otro ejemplo

> vagrant up



Enciende la máquina virtual

Otro más

- > Vagrant ssh
- > Vagrant ssh -c "cat /etc/sources.list"

Otro más

```
> vagrant init obihann/nginx \  
    --box-version 0.0.1  
    vagrant up
```

config.vm.network "forwarded_port", guest: 80, host: 8080

Creamos un .html de ejemplo

Esta es una lista de comandos básicos:

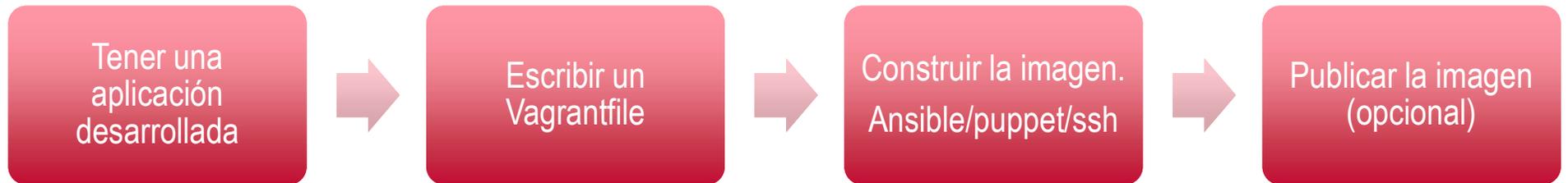
- Adding a vagrant box:
 - Syntax: `vagrant box add`
 - Example: `vagrant box add ubuntu/trusty32`
- Listing and removing vagrant boxes:
 - `vagrant box list`
 - `vagrant box remove`
- Creating a VM environment:
 - Syntax: `vagrant init`
 - Example: `vagrant init ubuntu/trusty32`
- Starting a VM environment:
 - `vagrant up ubuntu/trusty32`
 - `vagrant up`
- Connecting:
 - `vagrant ssh ubuntu/trusty32`
 - `vagrant ssh`
- Stopping, restarting, and destroying
 - `vagrant halt`
 - `vagrant reload`
 - `vagrant destroy`

VIRTUALIZANDO APLICACIONES

Imágenes en vagrant

- Una imagen un fichero de disco más un fichero de configuración
- Se parte de una imagen base y luego se construyen imágenes personalizadas encima
- Un Vagrant file define las opciones de arranque de la máquina
- Vagrant no se encarga del aprovisionamiento (instalación de apps y dependencias)

Pasos para VMizar una aplicación



Nuestra aplicación: Un “Hello world” hecho en python con el framework Flask

```
# Importamos el modulo de flask para poder usar ese framkework
from flask import Flask

# Constructor de Flask
app = Flask(__name__)

# En flask tenemos distintas rutas para distintas funciones
@app.route('/')

# '/' está asociada a la función hello_world().
def hello_world():
    return 'Hello World'

# '/hello/name' está asociada a la función hello_name().
@app.route('/hello/<name>')
def hello_name(name):
    return 'Hello %s!' % name

# Función principal
if __name__ == '__main__':
    app.run()
```

El Vagrantfile

```
Vagrant.configure("2") do |config|  
  config.vm.box = "ubuntu/bionic64"  
  config.vm.network "forwarded_port", guest: 80, host: 8080  
  config.vm.provision "shell", path: 'provision.sh'  
end
```

El aprovisionamiento

```
sudo apt update  
sudo apt upgrade -y  
sudo apt install -y  
git python3 python3-pip screen  
git clone https://github.com/EGCETSII/1920-Practica-1.git  
cd 1920-Practica-1 pip3 install -r requirements.txt  
screen -m -d python3 holamundo.py
```

**EJECUTANDO DECIDE EN VAGRANT CON
ANSIBLE**

DECIDE ON VAGRANT

- What we do need to run decide?
 - Python
 - Webserver
 - Postgres
- How to provision this?
 - Ssh?
 - Sudo apt install python3-pip postgresql ...
 - Pip install ...
 - Etc etc
 - ¿But wat about if we do run this on alpine instaed of Ubuntu? ¿And if we move to Debian?
 - Ansible to the resque

Ansible



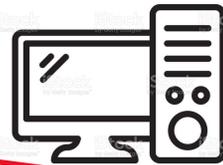
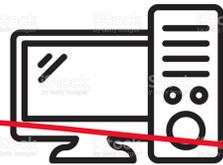
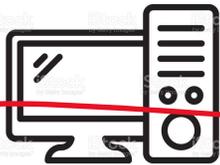
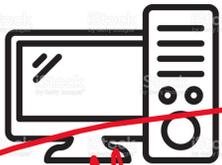
[Ansible](#) is quite often called “a loop for ssh”. It is a bit an oversimplification, however – yes it allows you to loop over your multiple hosts (physical or virtual) and apply changes.

Ansible working



ANSIBLE

Playbook.yml = recetas



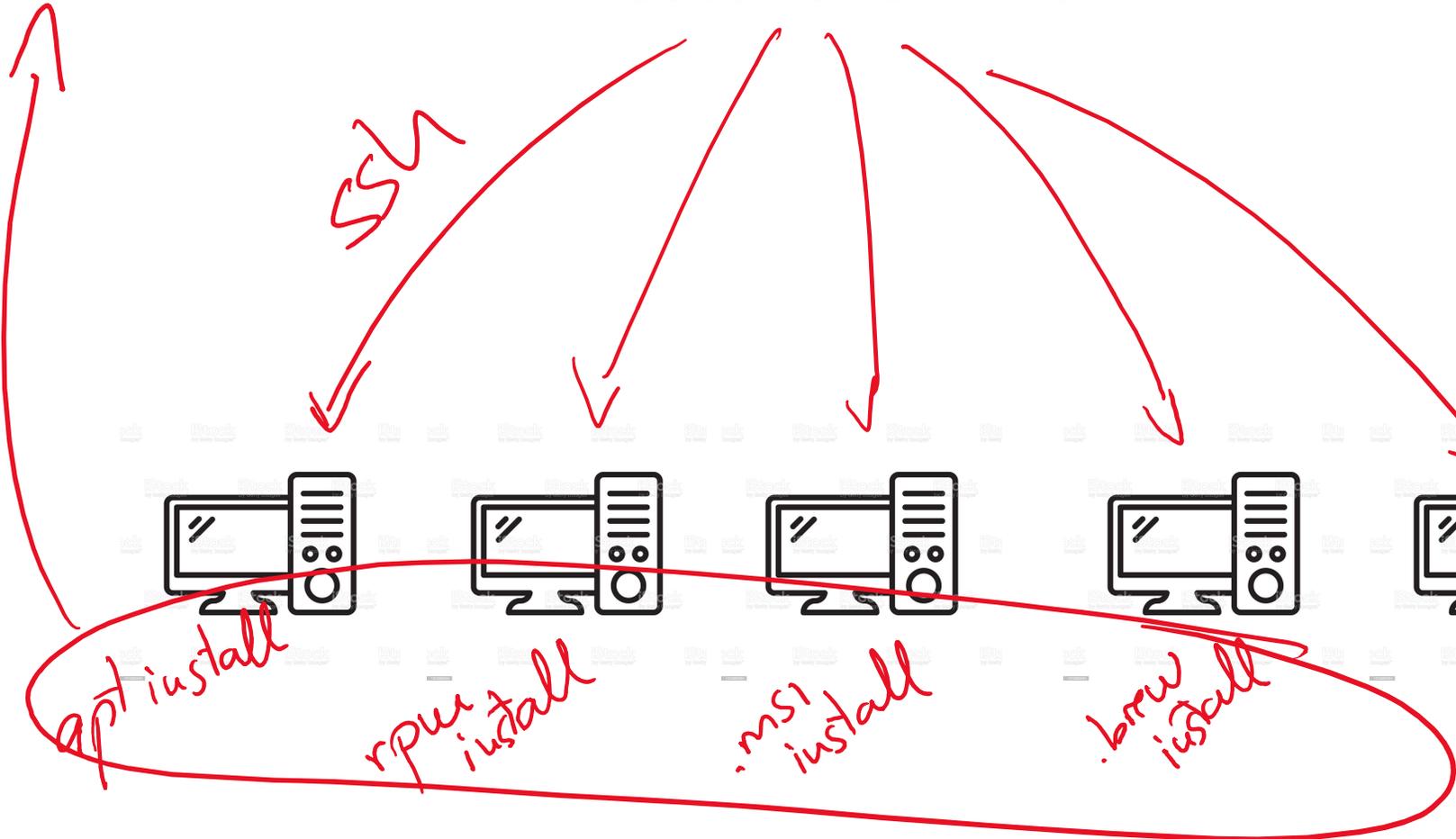
SSH

apt install

rpm install

msi install

brew install



The vagrantfile

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/bionic64"
  config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.provider "virtualbox" do |v|
    v.memory = 512
    v.cpus = 1
  end
end
```

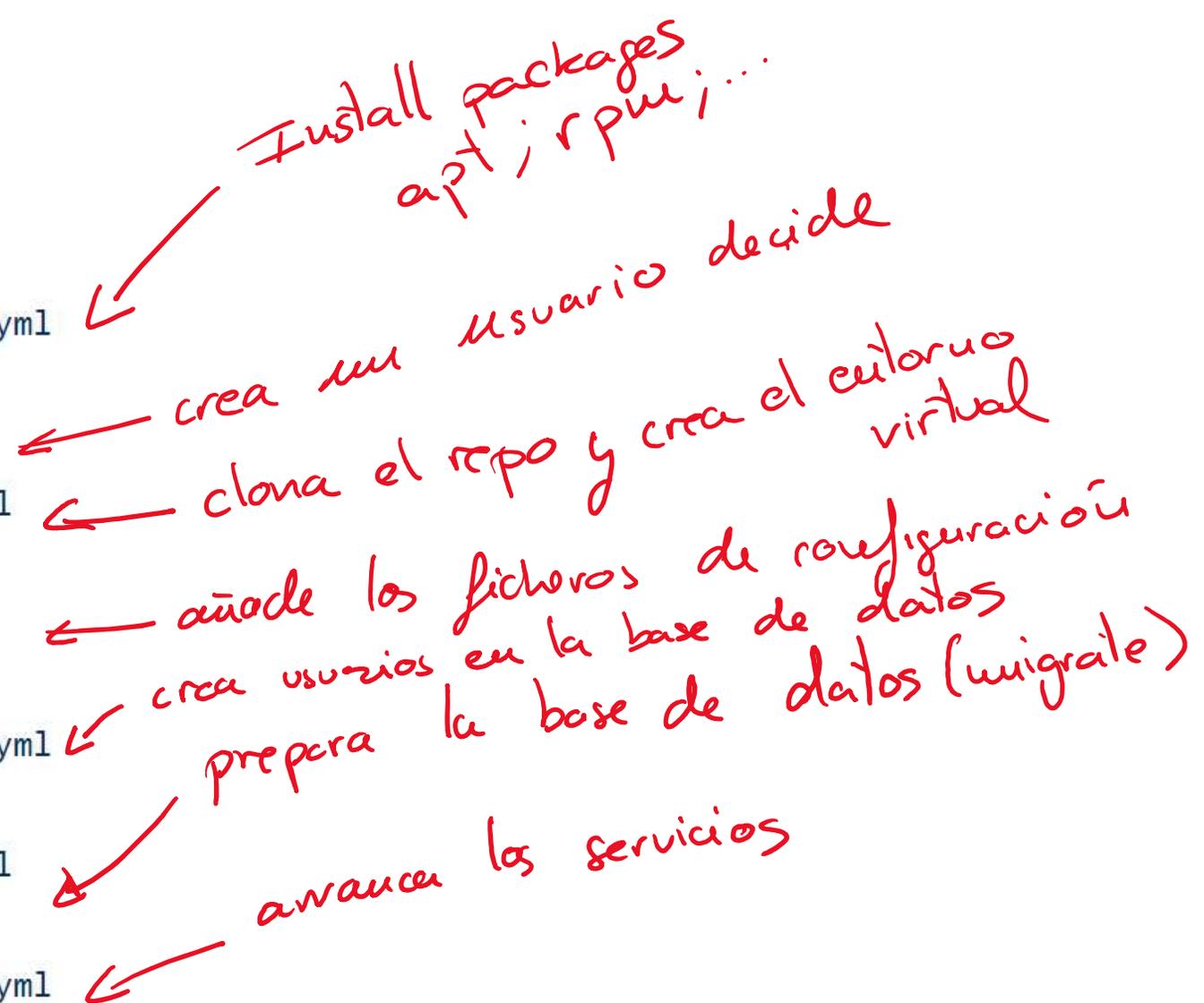
```
config.vm.provision "ansible" do |ansible|
  ansible.compatibility_mode = '2.0'
  ansible.playbook = "playbook.yml"
  ansible.extra_vars = { ansible_python_interpreter:"/usr/bin/python3" }
end
```

```
end
```

The playbook

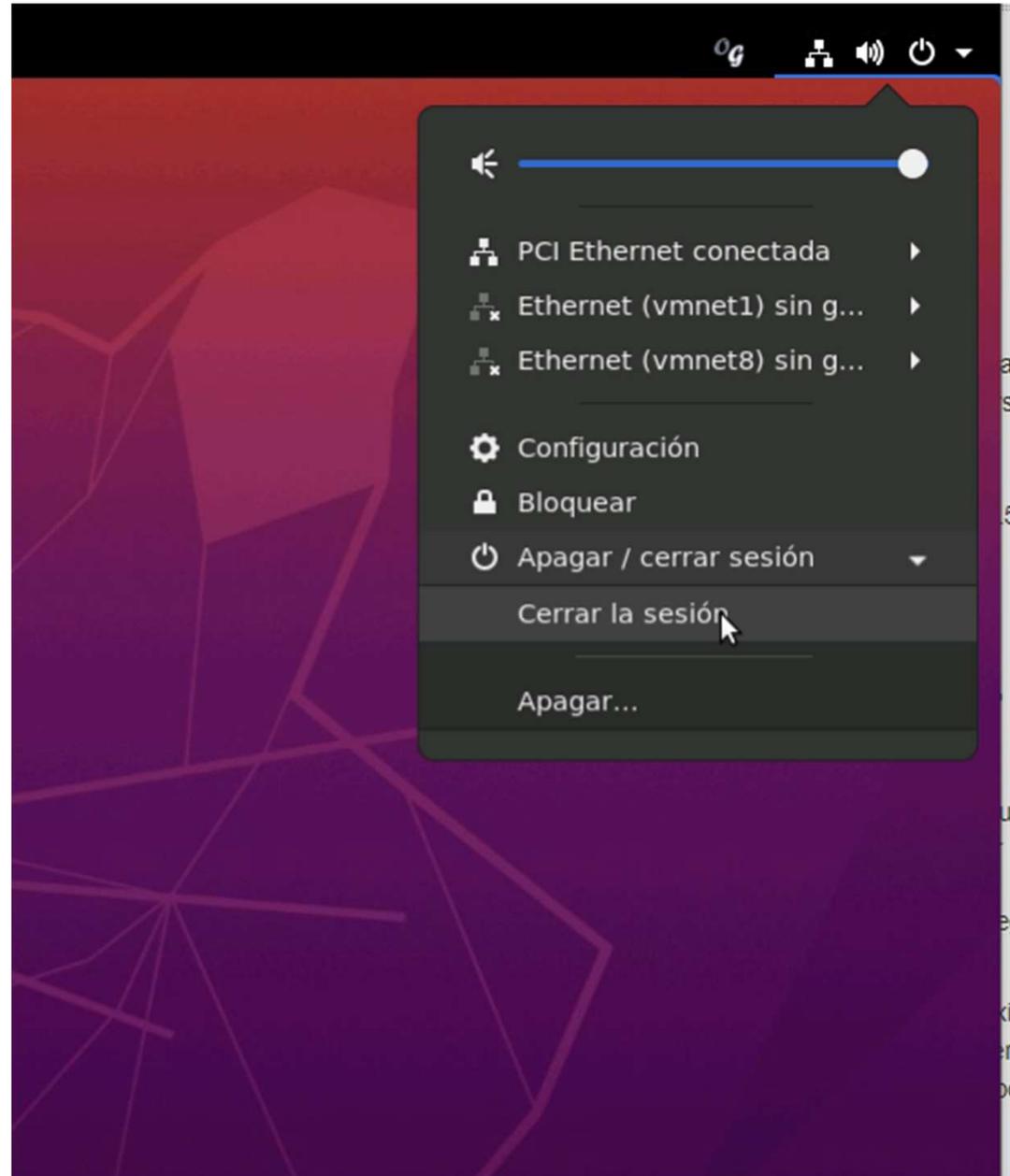
```
---
- hosts: all

tasks:
  - include: packages.yml
    tags: ["packages"]
  - include: user.yml
  - include: python.yml
    tags: ["app"]
  - include: files.yml
    tags: ["files"]
  - include: database.yml
    tags: ["database"]
  - include: django.yml
    tags: ["django"]
  - include: services.yml
    tags: ["services"]
```



Máquinas virtuales del CdC de la ETSII

- Hay que cerrarlas con “cerrar la sesión”
- **No cerrar la pestaña**



CONCLUSIONES

¿Para qué me sirve una VM como desarrollador?

- Entornos de desarrollo:
 - Compartibles
 - Seguros
 - Limpios
 - Extensibles
- Asegura el mismo entorno en:
 - Todos los desarrolladores
 - Pruebas
 - Producción
- Facilita gestionar varias versiones de una misma aplicación
- Ahorra costes en el despliegue

¿Para qué me sirve como administrador?

- Despliegue independiente de la tecnología (Java, PHP, NodeJS...)
- Elimina inconsistencias entre entornos de desarrollo, prueba y producción
- Permite desplegar de forma similar en:
 - El portátil del desarrollador
 - En máquinas virtuales en un data center
 - En servidores cloud (AWS, Azure, DigitalOcean...)
 - En una mezcla de ellos
- Es más caro que las máquinas virtuales