



ADMINISTRACIÓN DE INFRAESTRUCTURAS Y SISTEMAS INFORMÁTICOS (AISI)

Grado en Ingeniería Informática

Grado en Ingeniería Informática

Roberto R. Expósito (roberto.rey.exposito@udc.es)

TEMA 1

Despliegue y administración de infraestructura



Contenidos

- Antecedentes
- DevOps
- IaC: Infrastructure as Code
- Modelos de despliegue



Contenidos

- **Antecedentes**
- DevOps
- IaC: Infrastructure as Code
- Modelos de despliegue

*Hace mucho tiempo, en un CPD muy, muy lejano, un antiguo grupo de seres poderosos conocidos como sysadmins desplegaban y configuraban **infraestructura manualmente**...*

Cada servidor, cada máquina virtual, cada contenedor, cada entrada de la tabla de rutas, cada configuración de la base de datos y cada balanceador de carga se creaba y administraba a mano...

*Era una época **oscura** y temerosa: altos tiempos de parada, configuraciones incorrectas debido a errores accidentales, despliegues en producción lentos y mucho miedo a qué sucedería si los sysadmins cayeran al **lado oscuro**...*

*Pero gracias a la alianza rebelde **DevOps**, tenemos una forma mejor de hacer las cosas: **Infraestructura como código (IaC)**...*



Antecedentes

6

- El despliegue y administración de servidores físicos/virtuales de forma robusta y **reproducible** siempre ha supuesto un desafío importante
 - Históricamente, los *sysadmins* estaban “aislados” de los desarrolladores y usuarios que interactuaban con los sistemas que ellos administraban
 - Departamentos de desarrollo y de sistemas (operaciones) separados/aislados
 - La administración se hacía fundamentalmente “a mano” instalando el *software* necesario, aplicando los cambios en la configuración y desplegando los servicios de forma individual en cada servidor
- Si el personal de sistemas formase parte de los equipos que desarrollan los productos *software*, sólo tendrían que preocuparse de los servidores, de las redes, del almacenamiento...de “su producto”
 - De esta forma podrían dar una mejor solución a los requisitos no funcionales
 - Podrían aceptar cambios “en cualquier momento” para adaptarse a las necesidades reales del producto según se va desarrollando, con un control continuo de la infraestructura necesaria para ello



Antecedentes

7

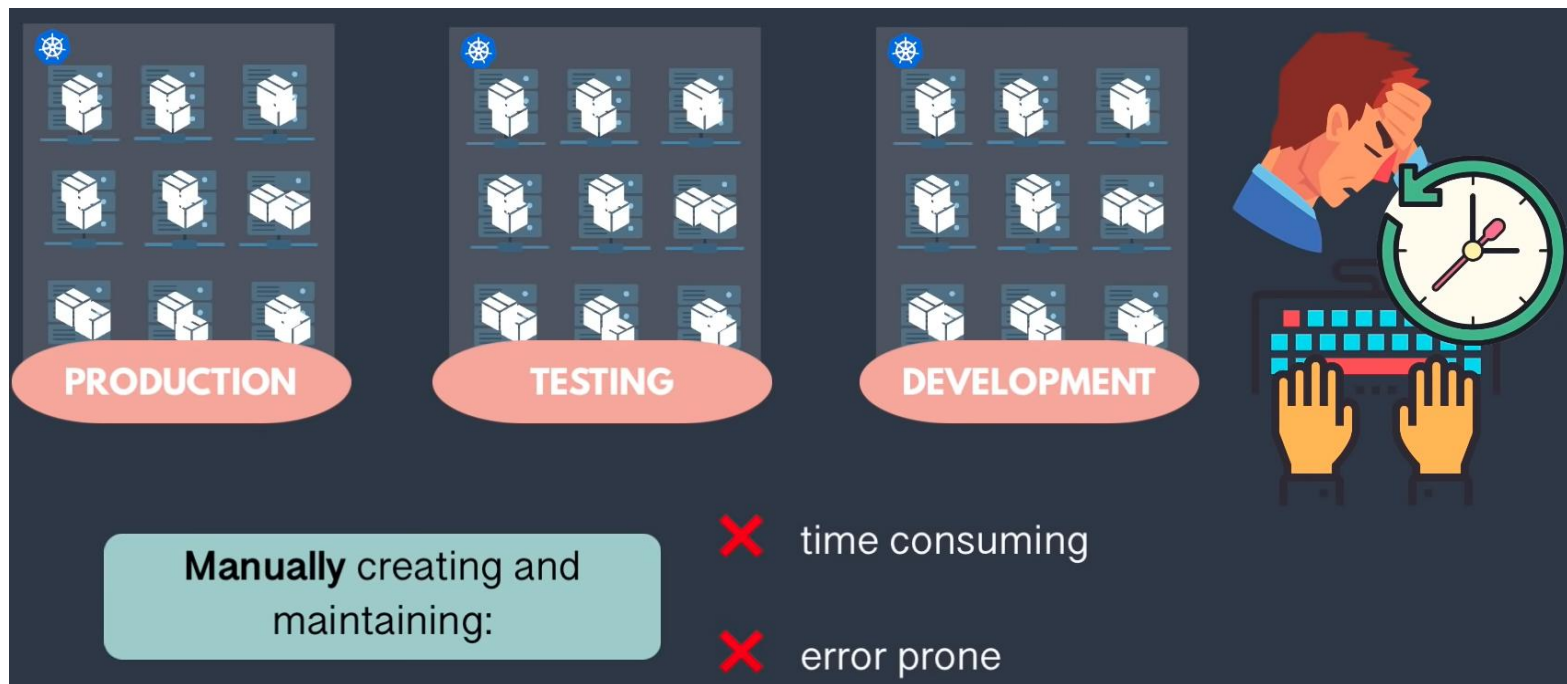
- El despliegue de entornos para desarrollo, pruebas, preproducción, producción...es un proceso costoso que requiere mucho tiempo y esfuerzo
 - Además, es un proceso propenso a un sinfín de problemas y errores derivados de su **operativa** tradicionalmente **manual**
- La creciente complejidad de los sistemas, así como de los entornos y aplicaciones a desplegar, requiere de herramientas de administración que faciliten la labor de los *sysadmins*
 - Además, el auge de la **virtualización** ha aumentado considerablemente el número de servidores a gestionar por cada *sysadmin*
- Es altamente deseable que el despliegue y configuración de un nuevo servidor (físico o virtual) se realice con la mínima intervención humana posible y de forma reproducible
 - Surge la necesidad de herramientas de **automatización** que minimicen las configuraciones manuales por parte de los *sysadmin*



Antecedentes

8

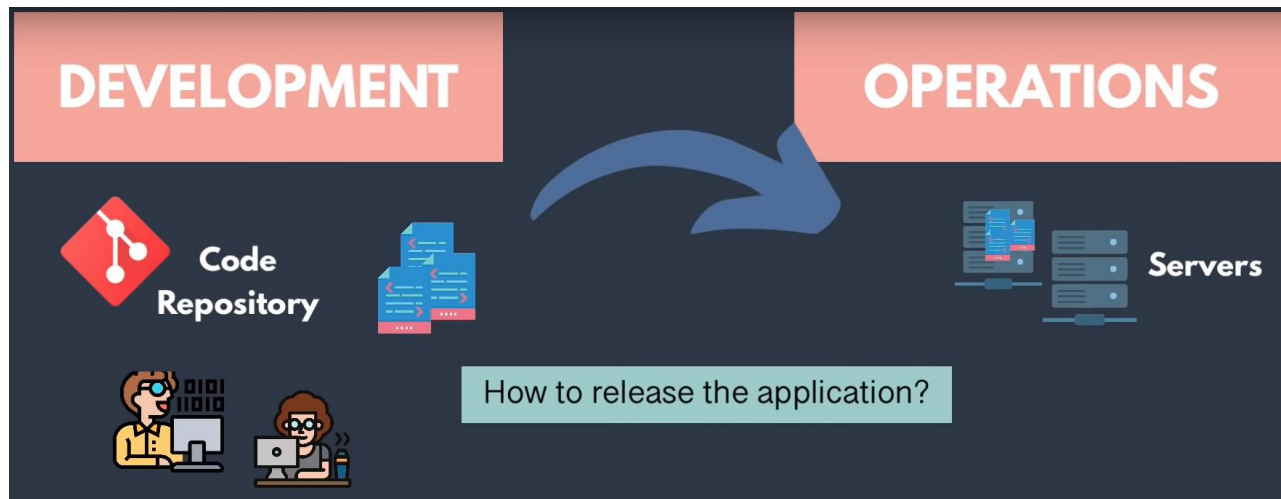
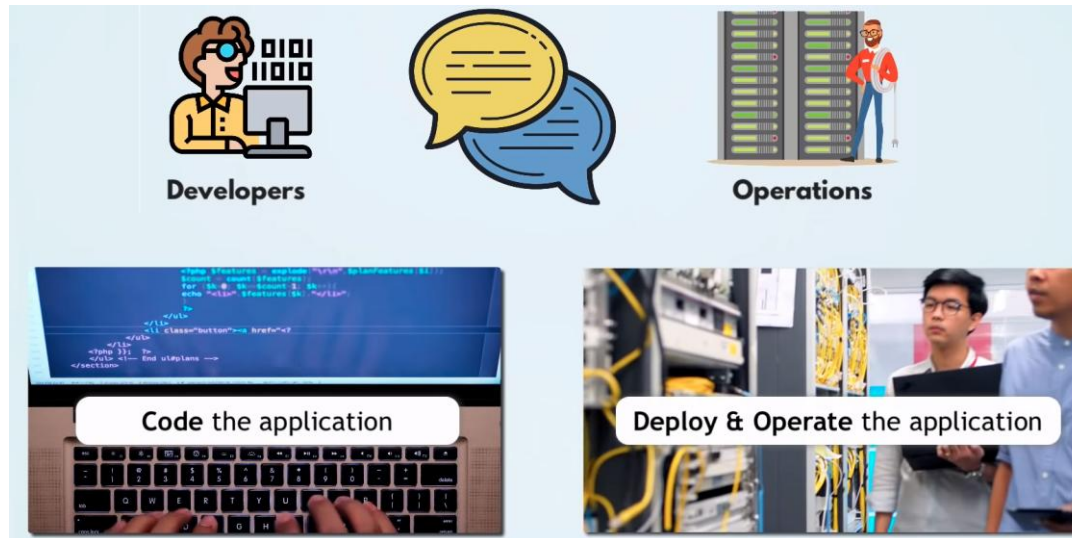
- Principales problemas de la operativa manual
 - Consume mucho tiempo
 - Propenso a errores





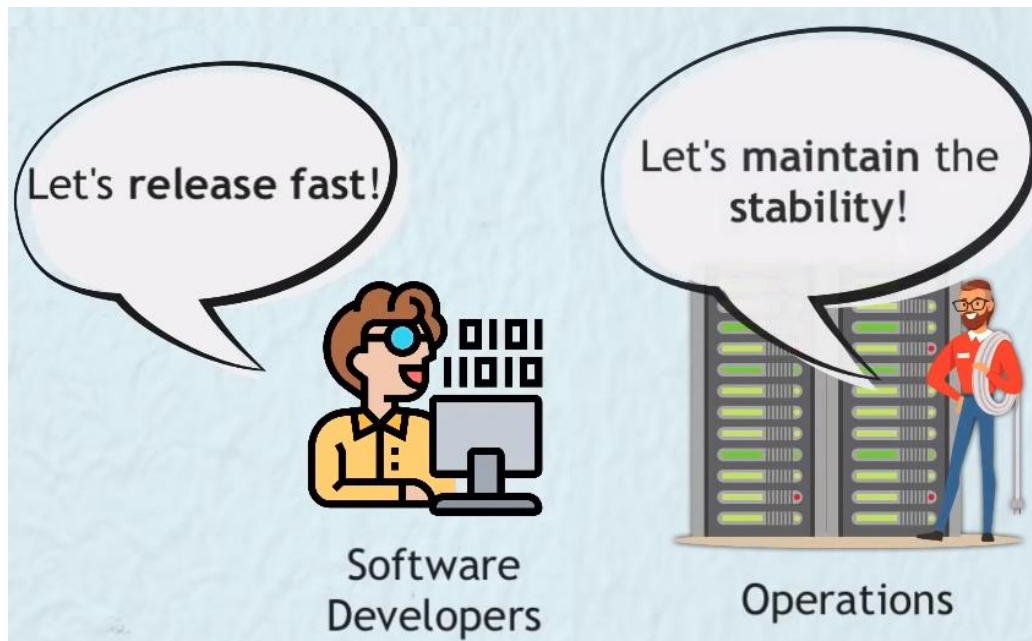
Contenidos

- Antecedentes
- **DevOps**
- IaC: Infrastructure as Code
- Modelos de despliegue





- ¿Qué quiere cada uno?

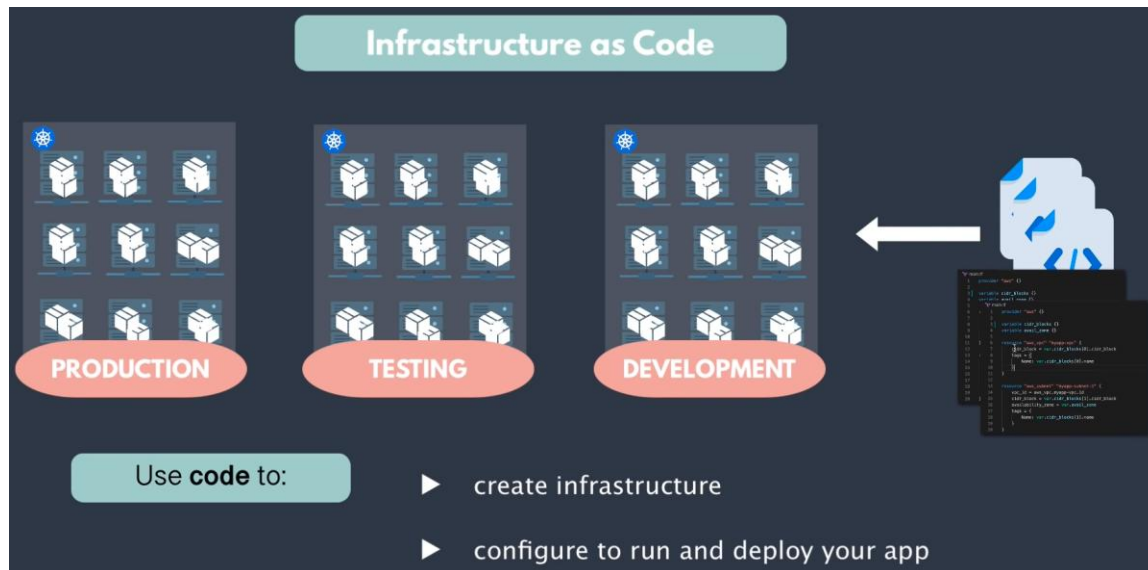
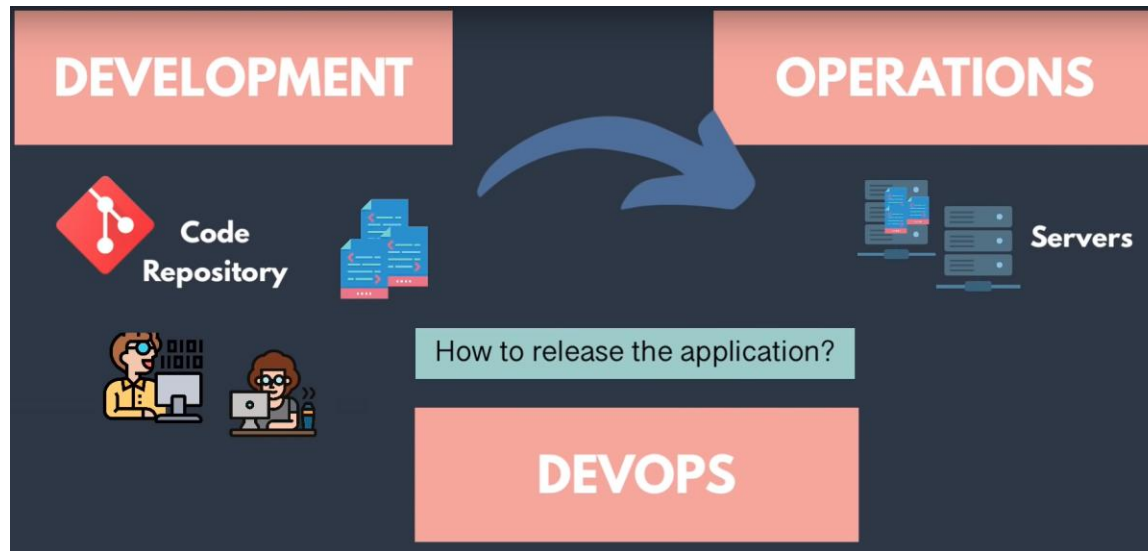




- **DevOps** es un enfoque “cultural”, organizativo y técnico que tiene como objetivo lograr la máxima colaboración e integración entre el **desarrollo de software (Dev)** y la **operación del software (Ops)**
 - “Cultura” de colaboración sin “muros” ni departamentos separados/aislados
- Objetivo principal: desarrollar *software* más rápidamente, con mayor calidad, menor coste y una mayor frecuencia de *releases*
 - Desarrolladores: “desean” enfocarse (solo) en desarrollar y poder desplegar y probar su código lo más rápido posible (“en cuestión de segundos”)
 - Cuando el personal de sistemas (*sysadmins*) trabaja más estrechamente con los desarrolladores se reducen los ciclos de desarrollo de las aplicaciones y se emplea más tiempo en “hacer las cosas” que en “apagar fuegos”
- DevOps orienta tanto el departamento de desarrollo de *software* como el departamento de sistemas (u operaciones) al **producto**
 - Hace años que los equipos de desarrollo se estructuran alrededor de **productos** específicos para satisfacer las necesidades de los clientes
 - Tradicionalmente el departamento de sistemas estaba “verticalizado” (estructura jerárquica) y orientado al **servicio**



- DevOps está fuertemente asociado a los siguientes conceptos:
 - Desarrollo (Dev): a prácticas de ingeniería del *software* basadas en metodologías ágiles (e.g. Scrum, Kanban) y técnicas de integración continua y entrega/despliegue continuo (CI/CD)
 - Operación (Ops): a prácticas de aprovisionamiento de la infraestructura, administración de sistemas y gestión de su configuración basadas en el paradigma **Infrastructure as Code (IaC)**
- En ambos casos, hay dos principios clave en común:
 - **Automatización** de procesos para reducir errores manuales y acelerar las entregas
 - **Monitorización** y registro continuo: se supervisa constantemente el rendimiento y la salud de las aplicaciones para identificar y resolver problemas más rápidamente
- Surge así el requerimiento de nuevos perfiles con habilidades específicas para organizar equipos DevOps
 - Para un desarrollador, pasar a un modelo DevOps resulta “relativamente sencillo” e inmediato (se enfocan en desarrollar)
 - Un *sysadmin* “tradicional” necesita diversas (y nuevas) habilidades
 - Conocimientos amplios sobre Linux y *networking*, conceptos básicos de programación y *scripting*, formación en tecnologías de virtualización y *cloud computing*, experiencia con herramientas IaC...

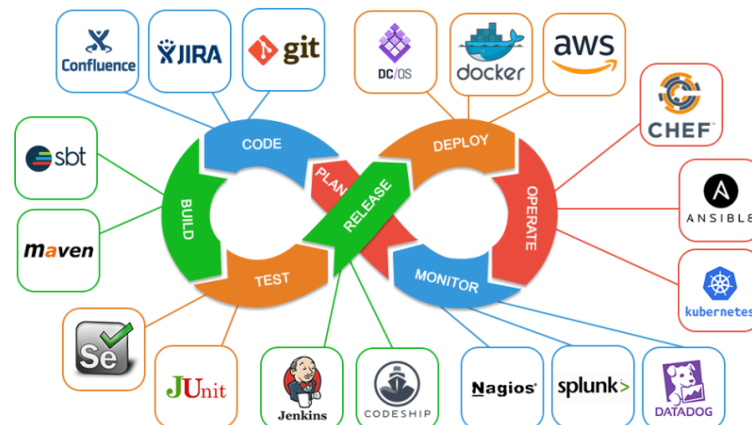


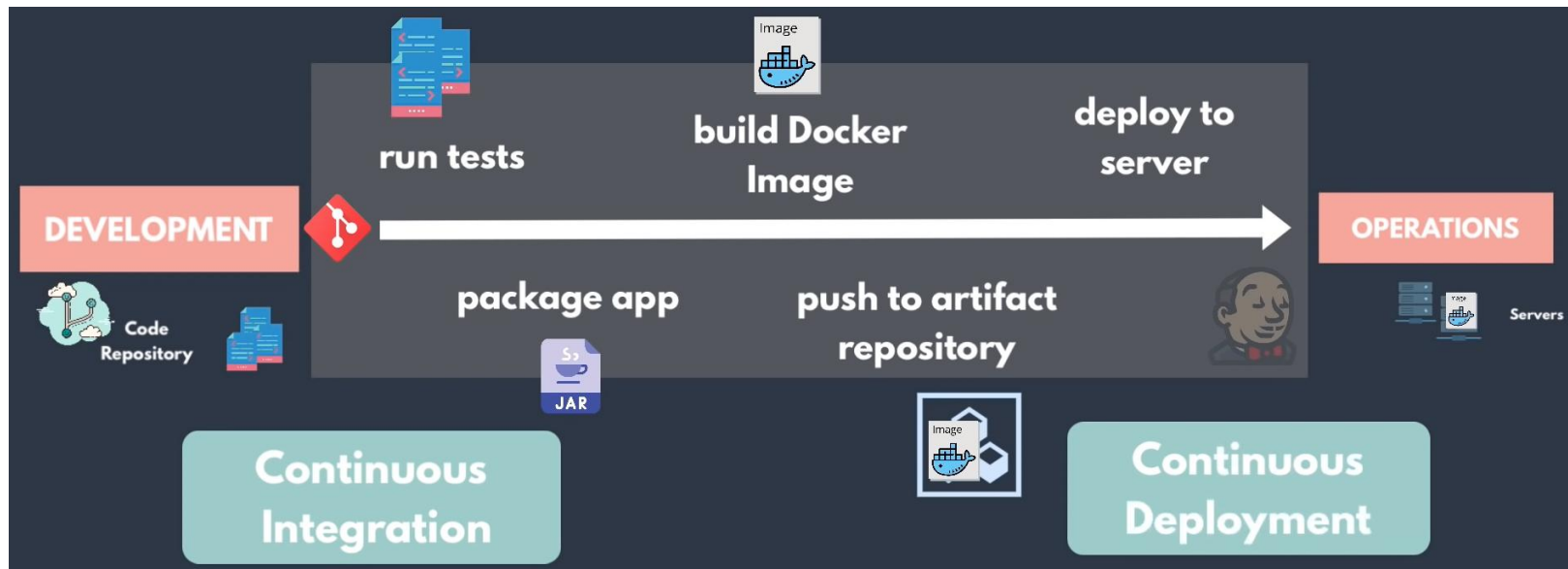
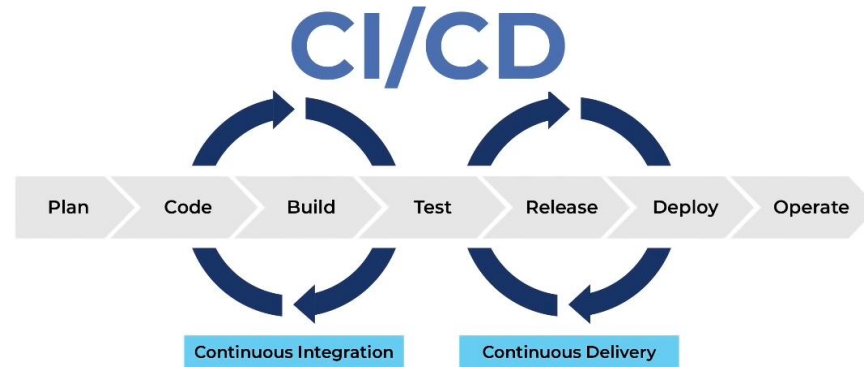


Herramientas DevOps

15

- Existen multitud de herramientas **de apoyo al desarrollo** y de **gestión de la infraestructura** enfocadas a simplificar y **automatizar todos** los procesos en entornos DevOps
 - CI/CD: Jenkins, CircleCI, Travis CI, ArgoCD, GitLab CI/CD...
 - Gestión de la infraestructura: Terraform, **Vagrant**, **Ansible**, Puppet, Chef, SaltStack, Pulumi, OpenTofu, CloudFormation...
 - Virtualización: **Docker**, **Kubernetes**, **Vagrant**, **Packer**, Openshift...
 - Monitorización y registro: Prometheus, Grafana, Datadog, Icinga, Splunk...
 - Plataformas cloud: AWS, Azure, GCP, IBM Cloud, Kamatera...
 - Otras: Slack, Gradle, JIRA, GitHub, Bitbucket, Maven...







Contenidos

- Antecedentes
- DevOps
- **IaC: Infrastructure as Code**
- Modelos de despliegue



laC: Infraestructura como código

18

- En vez de gestionar infraestructura mediante “*clicks*” usando herramientas GUI o mediante la ejecución de comandos/*scripts* por CLI, la idea de laC es **escribir código fuente** que permita **definir, aprovisionar y/o gestionar** infraestructura de una forma **automatizada**
 - Un objetivo de laC es que los *sysadmins* no tengan que acceder (frecuentemente) a los servidores hacer configuraciones de forma manual
- laC pretende proporcionar dinámicamente sistemas y servidores que son **predecibles** y cuyas configuraciones sean **inmutables**
 - La configuración específica de un servidor (su “**estado**”) debería poder predecirse de forma exacta (o casi) por el código fuente que la define
 - Código que podemos tener (idealmente) versionado en sistemas CVS
- Existen en la actualidad múltiples herramientas laC de código abierto y propietarias con diferentes características y funcionalidades
 - Chef, Ansible, Puppet, SaltStack, Terraform, CloudFormation, OpenTofu, CFEngine, Packer, Pulumi, Vagrant, OpenStack Heat, Foreman...
 - Es habitual (y necesario a veces) combinar varias de ellas para conseguir el objetivo de desplegar infraestructura inmutable en entornos DevOps



Ventajas de IaC

19

- IaC busca automatizar por completo el proceso de aprovisionamiento y despliegue de sistemas e infraestructuras
 - Despliegues más rápidos y fiables
- Permite representar el **estado** de la infraestructura en ficheros de código fuente que “cualquiera” puede leer e interpretar
 - El estado actual de un servidor ya no reside “en la mente” de un *sysadmin*
- Los ficheros fuente que representan la infraestructura pueden (y deben) ser **versionados** de igual forma que el código de una aplicación
 - Esto permite disponer de un histórico de cambios que puede ser usado para *debugging* ante posibles problemas
- Cada cambio que se haga en la infraestructura puede ser **validado** mediante revisiones de código y *testing* automatizados
 - Integración continua y *testing* automatizado de la infraestructura
- Permite la **reutilización** de código de forma robusta y fiable que además puede ser documentado y compartido



- A grandes rasgos, podemos clasificar las herramientas IaC existentes en cuatro conjuntos no necesariamente disjuntos
 - Herramientas para el **aprovisionamiento de la infraestructura**
 - *Infrastructure/server provisioning tools*
 - Herramientas para la **gestión de la configuración**
 - *Configuration Management (CM) tools*
 - Herramientas para la **gestión de imágenes/plantillas**
 - *Server templating tools*
 - Herramientas para la **orquestración de la infraestructura**
 - *Orchestration tools*



- Herramientas para el **aprovisionamiento de la infraestructura**
 - Funcionalidad principal: aprovisionar (instanciar) los recursos computacionales necesarios (p.e. servidores físicos/virtuales) sobre los cuales se desplegará el *software* de base (e.g. SO) y las aplicaciones/servicios
 - Incluye todas las acciones necesarias para preparar una instancia en ejecución de un servidor físico desde el *hardware* “en bruto” (*bare metal*) hasta un sistema completamente funcional con un SO instalado
 - Los recursos pueden ser instanciados sobre *hardware on-premises* o sobre recursos en la nube proporcionados por proveedores *cloud* públicos (e.g. AWS, Azure) o privados (OpenStack, OpenNebula)
 - Algunas herramientas de este tipo también permiten aprovisionar no solo los servidores, sino cualquier otro recurso de infraestructura (e.g. *networking*, balanceadores de carga, *firewalls*)
 - También pueden proporcionar funcionalidad básica para gestionar la configuración (i.e. el “estado”) de los recursos aprovisionados
 - Ejemplos:
 - Terraform, Vagrant, CloudFormation, Pulumi, OpenStack Heat, OpenTofu, Foreman, Cobbler, Spacewalk, Razor, OpenQRM, Crossplane



- Herramientas para la **gestión de la configuración**
 - Funcionalidad principal: gestionar la configuración *software* (i.e. el “estado”) de servidores físicos/virtuales
 - Incluye todas las acciones necesarias para configurar el *software* de los servidores:
 - Tareas de administración del SO (gestión de usuarios, configuración de la red, configuración del almacenamiento, securización/*hardening*...)
 - Instalación y configuración de los servicios y las aplicaciones
 - Generalmente, estas herramientas asumen que los recursos de infraestructura que deben gestionar han sido aprovisionados previamente
 - Por ejemplo, usando alguna herramienta de la categoría previa
 - También pueden proporcionar algún grado de soporte básico para el aprovisionamiento de la infraestructura
 - Ejemplos:
 - Ansible, Chef, Puppet, SaltStack, CFEngine, Powershell DSC



- Herramientas para la **gestión de imágenes/plantillas**
 - Permiten crear **imágenes máquina** de forma automatizada con todo el *software* necesario y su configuración asociada
 - Una imagen máquina es unidad estática que contiene un SO y *software* pre-instalado que se utiliza como plantilla para instanciar rápidamente entornos virtuales de ejecución (p.e. VM, contenedores) en local y/o en la nube
 - De esta forma se optimiza (y minimiza) el tiempo necesario para el aprovisionamiento de la infraestructura y/o la gestión de su configuración
 - Usar imágenes personalizadas proporciona una mayor velocidad de despliegue y mejora la reproducibilidad
 - En cambio, construir las imágenes implica un mayor esfuerzo inicial y puede ralentizar las actualizaciones del *software* y/o la configuración
 - Ejemplos:
 - Vagrant, Packer, Docker, Netflix Animator



- Herramientas para la **orquestación de la infraestructura**
 - Funcionalidad principal: despliegue, ejecución y administración automatizada de aplicaciones en VM/contenedores en entornos distribuidos
 - Típicamente en infraestructuras de tipo clúster o plataformas *cloud*
 - Suelen proporcionar varias de las siguientes características
 - Auto escalado en respuesta a la carga de trabajo (*auto scaling*)
 - Horizontal: aumentar/reducir el número de VM/contenedores
 - Vertical: aumentar/reducir los recursos (CPU, memoria) de una VM/contenedor
 - Balanceo de la carga entre las VM/contenedores (*load balancing*)
 - Descubrimiento de servicios (*service discovery*)
 - Monitorización y registro (*logging*) de las VM/contenedores, aplicaciones...
 - Técnicas de auto remediación (*auto healing/self healing*)
 - Soporte para despliegues y actualizaciones de las aplicaciones sin interrupción de servicio (*zero-downtime and blue/green deployments*)
 - Ejemplos:
 - Kubernetes, Docker Swarm, OpenShift/OKD, Nomad, Amazon EKS, Azure AKS, Mesos, Marathon



Modelo imperativo vs declarativo

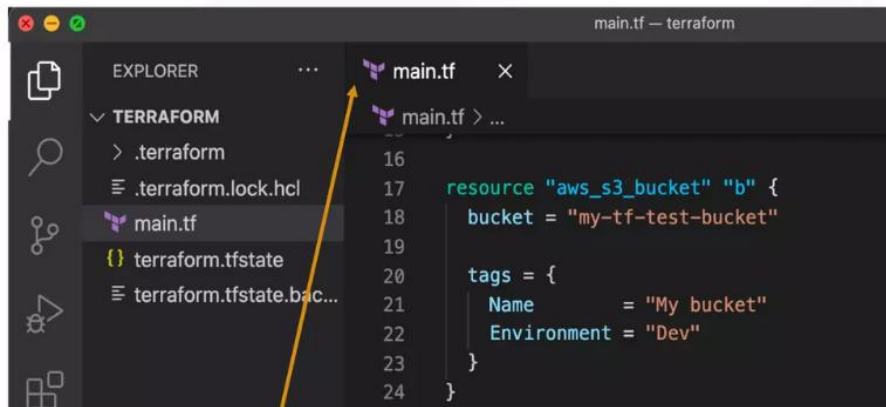
25

- El “lenguaje” en el que las herramientas IaC permiten definir la infraestructura y su configuración puede seguir una aproximación imperativa (procedural), declarativa o incluso ambas
- **Imperativo:** el código **especifica cómo alcanzar el estado final** de la infraestructura y/o la configuración que se desea de la misma
 - Acciones de ejemplo: instalar el paquete X, iniciar el servicio Y, ejecutar 2 VM
 - El estado actual de la infraestructura en un momento dado no puede extraerse exclusivamente desde el código fuente que la define
 - Puede ser necesario lógica adicional que tenga en cuenta el estado actual y los cambios realizados en el pasado
 - Ejemplos: Chef, AWS CLI, Vagrant, Ansible
- **Declarativo:** el código **especifica el estado final deseado** y la herramienta decide la lógica necesaria para alcanzar dicho estado
 - Acciones de ejemplo: el paquete X debe estar instalado, el servicio Y debe ser iniciado, 2 VM deben estar en ejecución
 - El código fuente representa el último estado de la infraestructura (i.e. su estado actual), lo que favorece su **idempotencia**
 - Ejemplo: Terraform, OpenTofu, Puppet, Packer, Ansible



Ejemplo: Terraform vs Pulumi

AWS Bucket Setup with Terraform

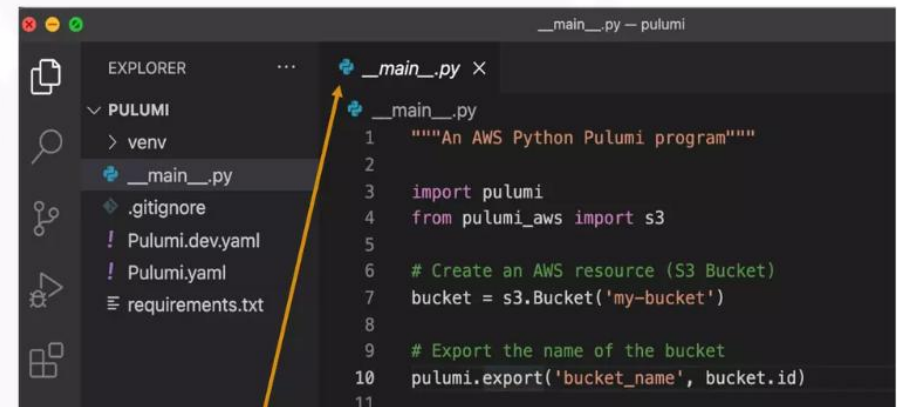


The screenshot shows a VS Code editor with a file explorer on the left. The file explorer shows a project named 'TERRAFORM' with files: '.terraform', '.terraform.lock.hcl', 'main.tf', 'terraform.tfstate', and 'terraform.tfstate.bac...'. The 'main.tf' file is selected and its content is displayed in the editor. The code is written in HashiCorp's Configuration Language (HCL) and defines an AWS S3 bucket resource named 'b' with the name 'my-tf-test-bucket' and tags for 'Name' and 'Environment'.

```
16
17 resource "aws_s3_bucket" "b" {
18     bucket = "my-tf-test-bucket"
19
20     tags = {
21         Name      = "My bucket"
22         Environment = "Dev"
23     }
24 }
```

HashiCorp's Language with .tf extension

AWS Bucket Setup with Pulumi



The screenshot shows a VS Code editor with a file explorer on the left. The file explorer shows a project named 'PULUMI' with files: 'venv', '__main__.py', '.gitignore', 'Pulumi.dev.yaml', 'Pulumi.yaml', and 'requirements.txt'. The '__main__.py' file is selected and its content is displayed in the editor. The code is written in Python and uses the Pulumi SDK to create an AWS S3 bucket resource named 'bucket' with the name 'my-bucket' and export its ID.

```
1 """An AWS Python Pulumi program"""
2
3 import pulumi
4 from pulumi_aws import s3
5
6 # Create an AWS resource (S3 Bucket)
7 bucket = s3.Bucket('my-bucket')
8
9 # Export the name of the bucket
10 pulumi.export('bucket_name', bucket.id)
11
```

Regular Python Programming Language
with .py extension



Master vs Masterless

27

- Algunas herramientas IaC requieren ejecutar un proceso **master** que almacena de forma centralizada el estado de la infraestructura
 - Cada vez que se quiere actualizar la infraestructura se debe usar un cliente que envía los comandos necesarios al *master*
 - El *master* propaga los cambios a los servidores gestionados (**modelo push**) o bien estos consultan al *master* cada cierto tiempo (**modelo pull**)
 - Desventaja: la presencia de un *master* requiere infraestructura extra para su ejecución que debe ser mantenida, actualizada y securizada
 - Puede implicar el uso de varios servidores adicionales para proporcionar alta disponibilidad
 - Ejemplos: Chef, Puppet, SaltStack
 - Muchas herramientas diseñadas en un principio para el modelo *master* han evolucionado hacia un modelo *masterless*
- Otras herramientas fueron diseñadas desde cero para un modelo *masterless* y no requieren desplegar procesos/infraestructura adicional
 - Ejemplos: Terraform, Ansible



Agent vs Agentless

28

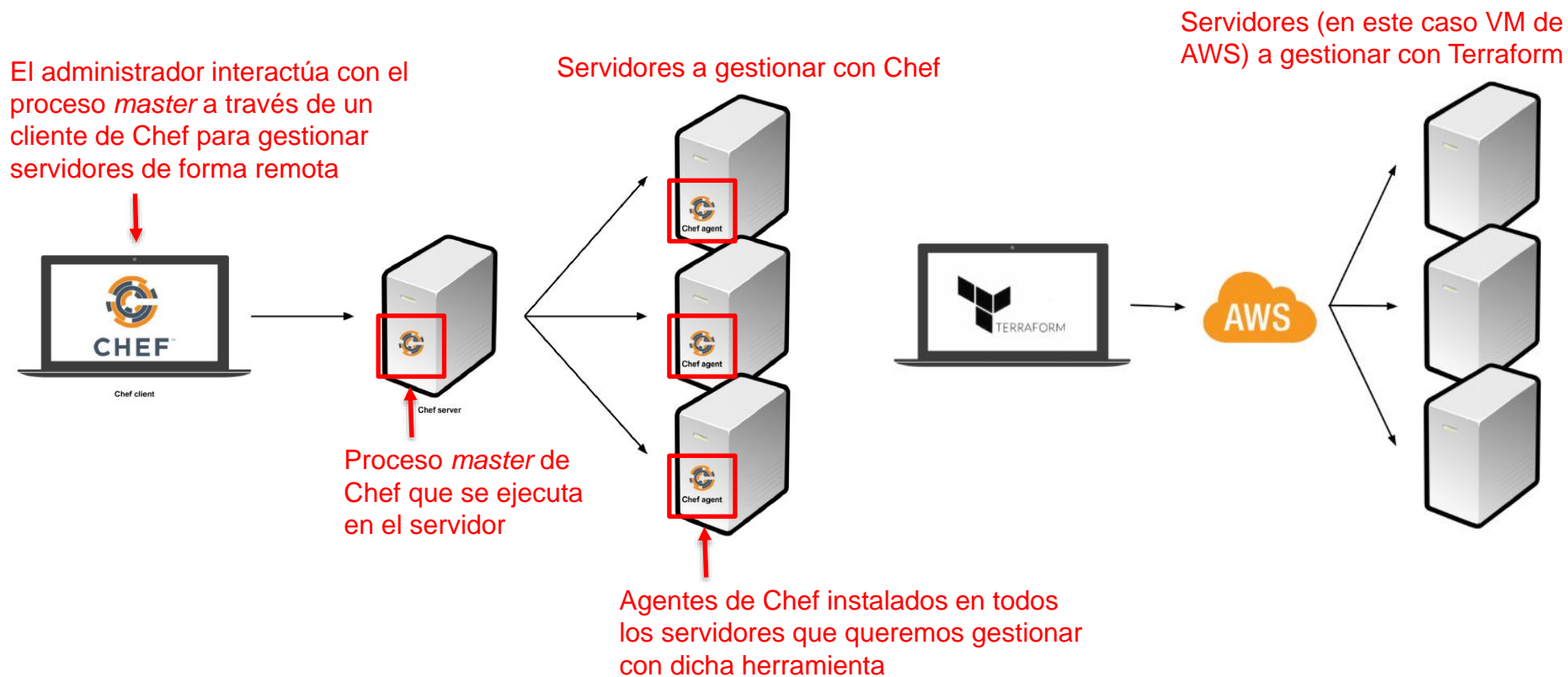
- Algunas herramientas IaC necesitan la instalación de un **software específico** (“agente”) en todos los servidores que se quieren gestionar
 - El agente se ejecuta en *background* y es el responsable de realizar la configuración obtenida (*pull*) o recibida (*push*) desde el *master*
 - Desventaja: el uso de agentes requiere tareas extra de mantenimiento, actualización y securización de los mismos
 - La instalación del agente en los servidores a gestionar debe realizarse como paso previo, por ejemplo, durante la fase de aprovisionamiento
 - Ejemplos: Chef, Puppet, SaltStack
 - De nuevo estas herramientas han ido evolucionando y suelen ofrecer algún tipo de soporte para el modo *agentless*
- Otras herramientas se diseñaron desde cero para un modo *agentless* y no requieren instalar ningún **software** específico adicional
 - Para ser más precisos, requieren algún **software** no específico proporcionado como parte intrínseca de la infraestructura (en el caso de proveedores *cloud*) o que se encuentran típicamente disponibles en los servidores (e.g. SSH)
 - Ejemplos: Terraform, Ansible



Ejemplo: Chef vs Terraform

29

- Terraform y Ansible siguen un modelo *masterless* y *agentless* en contraposición a una herramienta como Chef





Contenidos

- Antecedentes
- DevOps
- IaC: Infrastructure as Code
- **Modelos de despliegue**



Modelos de despliegue

31

- Es conveniente (y habitual) combinar varias herramientas IaC con el objetivo de conseguir que el proceso de despliegue de la infraestructura y su configuración esté **definida completamente en código**
 - Además de perseguir el objetivo de obtener una infraestructura **inmutable**
- Los modelos o aproximaciones de despliegue más comunes son:
 - *Provisioning + Configuration Management*
 - *Provisioning + Server templating*
 - *Provisioning + Server templating + Orchestration*



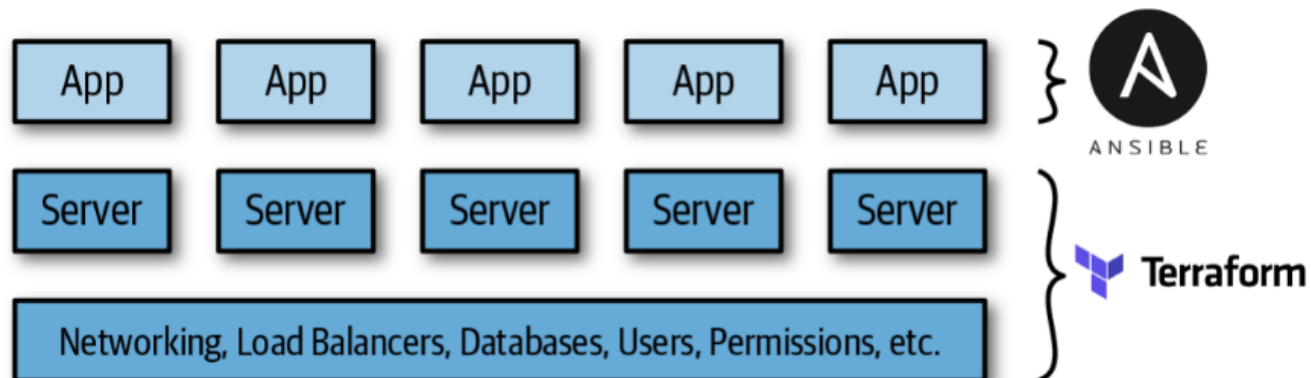
Modelos de despliegue

32

- **Provisioning + Configuration Management: Terraform + Ansible**

- Terraform permite desplegar toda la infraestructura necesaria (e.g. en la nube): desde la topología de red (e.g. subredes), balanceadores de carga, bases de datos, hasta los entornos virtuales (VM/contenedores)
- Con Ansible se realizaría el despliegue de las aplicaciones y servicios en los recursos aprovisionados con Terraform así como su configuración
- Ventajas: no es necesario desplegar infraestructura extra ya que ambas herramientas no requieren agentes específicos ni ejecutar procesos *master*
- Desventajas: Ansible puede producir servidores mutables en el tiempo dependiendo de la idempotencia de los comandos/playbooks utilizados

agentless
masterless

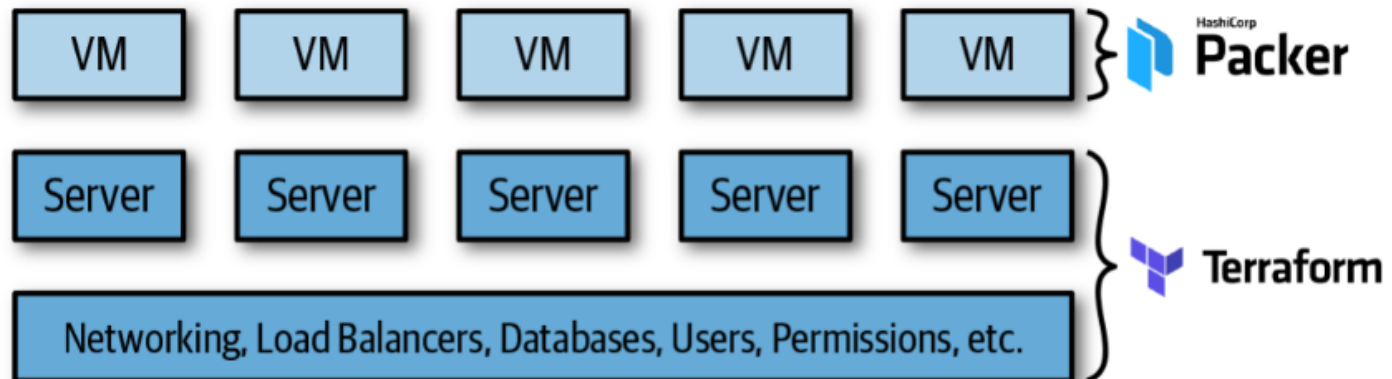




Modelos de despliegue

33

- **Provisioning + Server templating: Terraform + Packer**
 - Packer permite empaquetar las aplicaciones y su configuración asociada en imágenes máquina que pueden ser usadas para instanciar VM
 - Terraform permite desplegar la infraestructura necesaria (e.g. VM) usando como plantillas las imágenes creadas con Packer
 - Ventajas: no es necesario desplegar infraestructura extra igual que la aproximación anterior y además se consigue la inmutabilidad
 - Desventajas: las VM pueden ser costosas de construir y desplegar ya que deben contener todo lo necesario (SO + aplicaciones + dependencias)

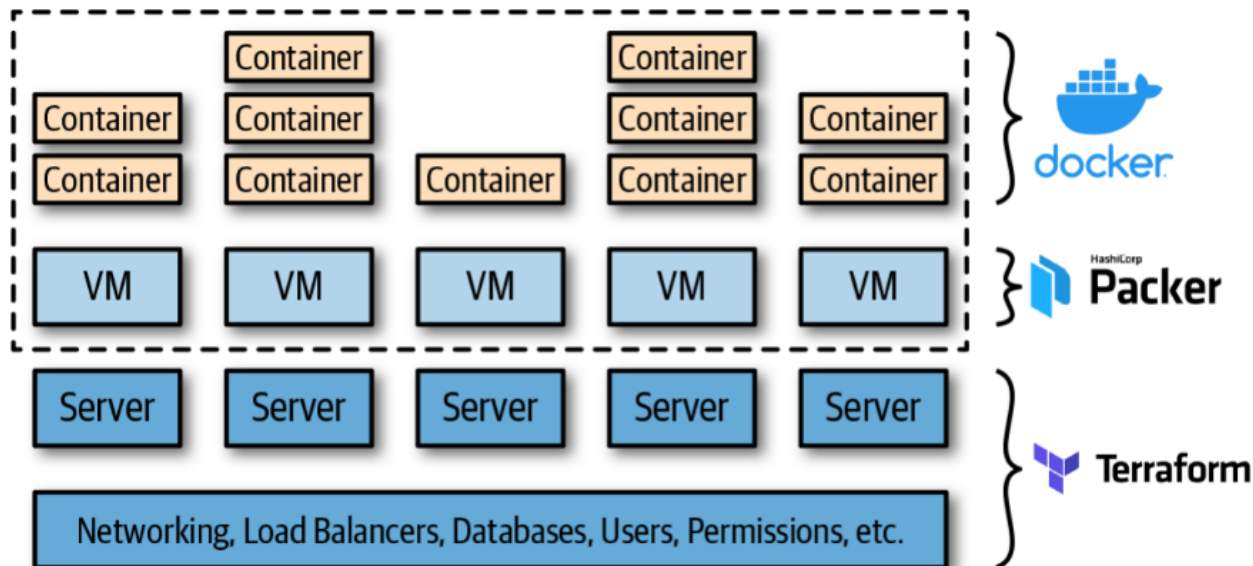




Modelos de despliegue

34

- **Provisioning + Server templating + Orchestration: Terraform + Packer + Docker + Kubernetes/Swarm**
 - Con Packer se crea una imagen máquina que contenga únicamente un motor de ejecución de contenedores (e.g. Docker Engine) y un orquestador (e.g. Kubernetes, Docker Swarm, Amazon EKS)
 - Con Terraform se despliegan las VM usando las plantillas de Packer que una vez iniciadas formarán el clúster de contenedores orquestado mediante Kubernetes o similar





Modelos de despliegue

35

- *Provisioning + Server templating + Orchestration*
 - Ventajas
 - No es necesario desplegar infraestructura extra
 - Se consigue una infraestructura inmutable
 - Las VM ahora son más ligeras al incluir menos *software*
 - Las imágenes de los contenedores (e.g. Docker) se construyen muy rápidamente y se pueden probar en local con facilidad
 - Se obtienen capacidades de auto escalado, balanceo de la carga, monitorización...dependiendo del orquestador utilizado
 - Desventajas
 - Se incrementa la complejidad de la administración debido a tener que aprender a usar, desplegar, configurar y *debuggear* varias “capas” de abstracción extra por la combinación de múltiples herramientas