Actividad 2

Explicar qué es IaC y por qué es un cambio de paradigma frente a la configuración manual.

IaC es una forma de gestionar y configurar la infraestructura de manera automática usando scripts, una configuración declarativa

Es un cambio de paradigma respecto a la configuración manual ya que permite reproducir de forma consistente en distintos entornos, además de reducir los errores manuales

Nombrar algunas herramientas de escritura de IaC

```
variable "nombre_de_la_aplicacion" {
  type        = string
  description = "Nombre descriptivo para la aplicación"
  default     = "mi_aplicacion"
}

variable "instance_type" {
  type        = string
  description = "Tipo de instancia"
  default     = "t2.micro"
}

resource "aws_instance" "app_server" {
  ami           = "ami-12345678"
  instance_type = var.instance_type
  tags = {
    Name = var.nombre_de_la_aplicacion
  }
}
```

Terraform: describe el estado final deseado de la infraestructura de forma. HCL es el lenguaje propio de Terraform.

Ansible:

- Gestión de la configuración de sistemas, la implementación de software, y la gestión de infraestructuras en la nube
- Permite automatizar tareas de almacenamiento, servidores y redes

Modularización

La infraestructura se descompone en módulos o componentes reutilizables, de manera que se facilita el mantenimiento y la reutilización

Por ejemplo, en Terraform, puedes crear un módulo para gestionar redes (VPC), otro para bases de datos (RDS) y otro para servidores (EC2).

DEPENDENCIAS

IaC libera a los equipos de la dependencia en configuraciones manuales

Tarea teórica

```
modulo/
├──main.tf          Lógica de la Infraestructura - Recursos Principales
├──variables.tf     Entrada, declaración de variables
└──outputs.tf       Salida
```

```
root/
├── modulo/
│   ├── network/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   ├── database/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── application/
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
```

Contenerización

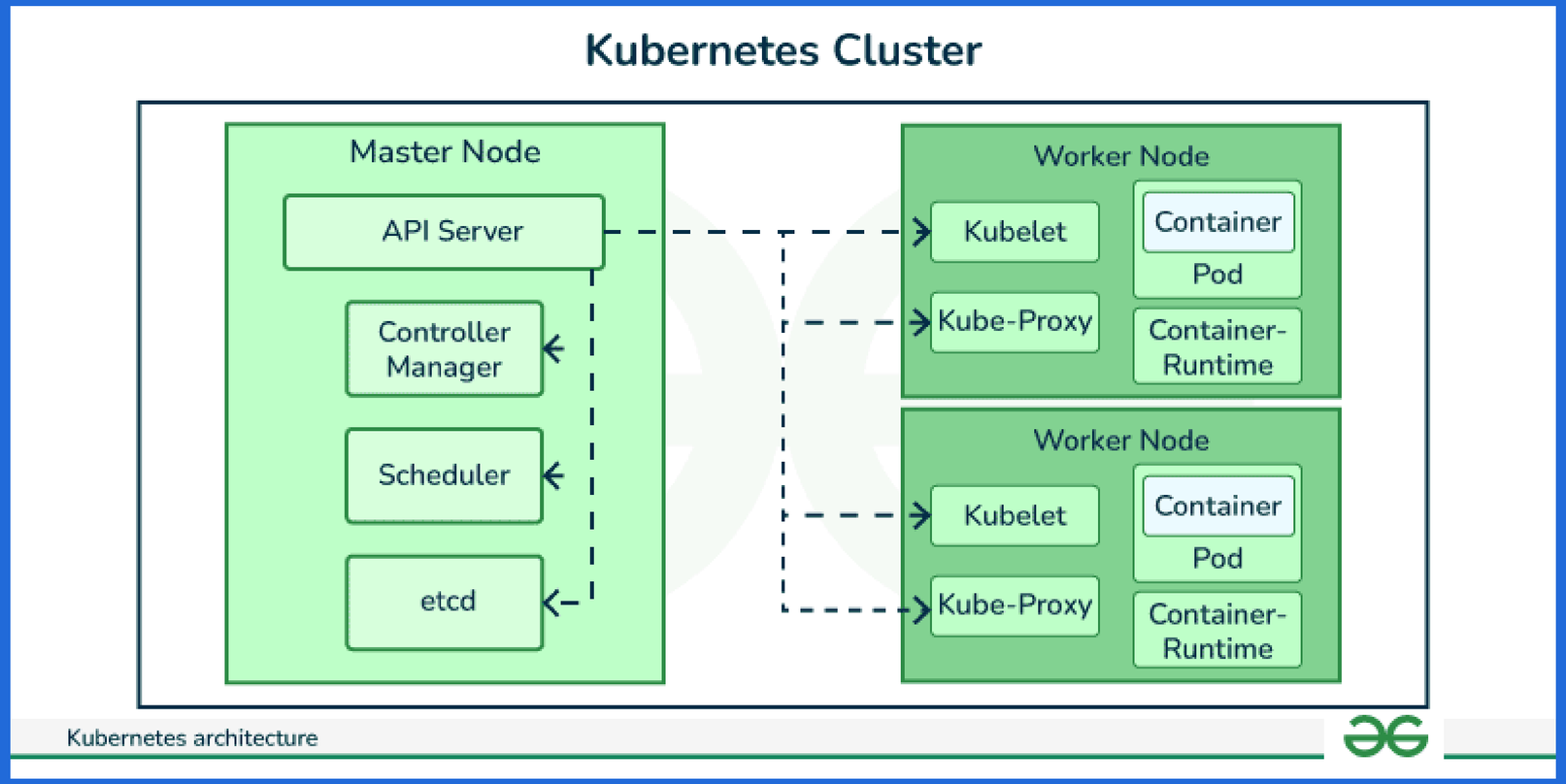
Son entornos exactos e iguales independientes del sistema operativo

	Docker	Máquina virtual
Tamaño	Imágenes de Docker son pequeñas, de unos kilobytes	Instancias de VM pueden ser de gigabytes o terabytes
Recursos	Utiliza menos recursos que una VM	Consume más recursos que Docker
Portabilidad	Más portable que las VM	Permite ejecutar múltiples sistemas operativos en un solo servidor físico
Aislamiento	Aislados de procesos, pero las fallas pueden afectar a todo el sistema	Proporcionan un alto nivel de aislamiento entre sistemas vecinos

> Creating Dockerfiles

```
# Derive image from another image with FROM{image}
FROM ubuntu:jammy-20230301

# Set a build and runtime environment variable with ENV {name}={value}
ENV TZ="America/New_York"

# Set a build-time variable with ARG {name}={default_value}
ARG VERBOSE=1

# Set the working directory with WORKDIR {path}
WORKDIR /home

# Switch to the user with USER {username}
USER doctordocker

# Copy a local file into the image with COPY {existing_path} {image_path}
COPY ./settings/config.yml ./settings/config.yml

# Run a shell command during the build step with RUN {command}
# \ lets commands continue across multiple lines
# && means run this command only if the preceding command succeeded
RUN apt-get update \
  && install -y libxml2-dev

# Run a shell command on launch with CMD ["{executable}", "{param1}"]
# Each Dockerfile should only have 1 CMD statement
CMD ["python", "-i"] # Start Python interactively
```

Orquestación

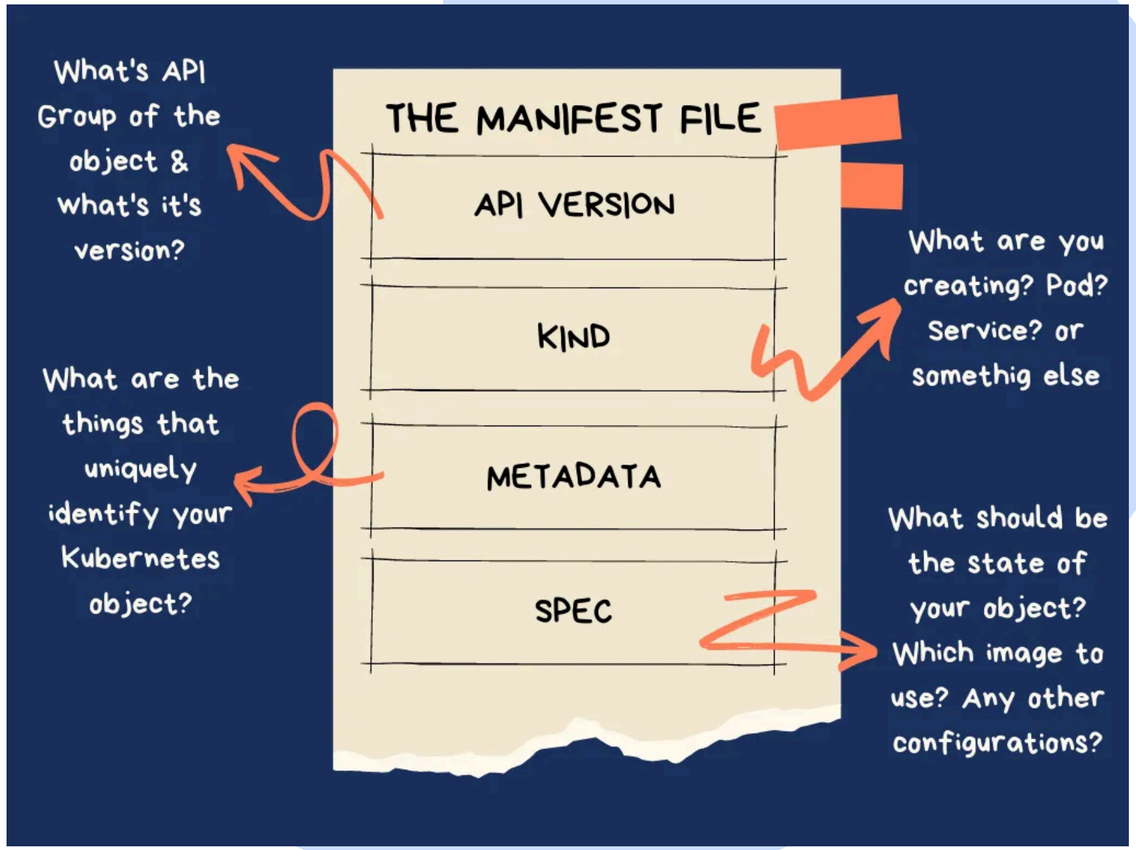


Manifiestos en YAML

Son archivos YAML o JSON que describen los objetos de un clúster de Kubernetes

YAML ✔

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: spring-pod
spec:
    containers:
    - image: amro/springapp:example
      name: spring-app
      ports:
      - containerPort: 80
        protocol: TCP
```



Tarea teórica

1. Crear una aplicación
2. Dockerizar la aplicación
3. Push the docker image a un contenedor
4. Kubernetes deployment
5. CI/CD pipeline

```
# Use the official Python image from Docker Hub
FROM python:3.9-slim

# Set working directory
WORKDIR /app

# Copy requirements and install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy the source code into the container
COPY . .

# Expose the application's port
EXPOSE 5000

# Run the application
CMD ["python", "app.py"]
```

```
# Build the Docker image
docker build -t flask-app .

# Run the container
docker run -p 5000:5000 flask-app
```

Tarea teórica

1. Crear una aplicación
2. Dockerizar la aplicación
3. Push the docker image a un contenedor
4. Kubernetes deployment
5. CI/CD pipeline

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: flask-app
  template:
    metadata:
      labels:
        app: flask-app
    spec:
      containers:
      - name: flask-app
        image: your-dockerhub-username/flask-app:latest
        ports:
        - containerPort: 5000
```

Tarea teórica

1. Crear una aplicación
2. Dockerizar la aplicación
3. Push the docker image a un contenedor
4. Kubernetes deployment
5. CI/CD pipeline

```yaml
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1

      - name: Log in to Docker Hub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and push Docker image
        uses: docker/build-push-action@v2
        with:
          push: true
          tags: your-dockerhub-username/flask-app:latest

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to Kubernetes
        env:
          KUBECONFIG: ${{ secrets.KUBECONFIG }}
        run: |
          kubectl apply -f deployment.yaml
          kubectl apply -f service.yaml
```

Tarea teórica

1. Escalado automático
2. Orquestación de contenedores
3. Orquestación de almacenamiento
4. Autocuración
5. Equilibrio de carga
6. Actualización y reversiones

Observabilidad

Capacidad de monitorear, entender y analizar el comportamiento de las aplicaciones, infraestructura y servicios.

Métricas: son cifras cuantificables que proporcionan información sobre el estado y desempeño de un sistema.

Logs: proporcionan detalles sobre los eventos específicos que ocurren en el sistema, como las transacciones o acciones ejecutadas

Trazas: son registros detallados de la ejecución de una transacción o solicitud a través de los distintos componentes del sistema

Ventajas

Detección temprana de errores

Optimización de recursos

Insights de negocio

Herramientas

Se encarga de recopilar y almacenar métricas de rendimiento

Permite visualizar esos datos de manera intuitiva y atractiva.

Tarea teórica

Adding the Helm Repository:
add the Prometheus community Helm repository

Instalar Prometeus y Grafana (custom-values.yaml)

install kube-prometheus-stack (custom-values.yaml)

Verifying the Installation:
(kubectl get services)

```yaml
# custom-values.yaml
prometheus:
  service:
    type: NodePort
grafana:
  service:
    type: NodePort
```

Accessing Prometheus and Grafana:
(kubectl get nodes -o wide)

```
$ kubectl get nodes -o wide
NAME                   STATUS   ROLES    AGE   VERSION   INTERNAL-IP   EXTERNAL-IP
pool-t5ss0fagn-jeb47   Ready    <none>   57d   v1.29.1   10.124.0.2    146.190.55.2
```

Integración