

# Actividad 2: Del código a la producción:

**Infraestructura,  
contenedores,  
despliegue y  
observabilidad**

# Integrantes

01

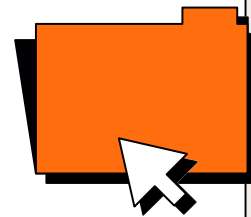
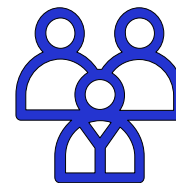
Hinojosa Zamora, Frank Oliver

02

Serrano Arostegui, Edy Saul

03

Choquechambi Quispe, Germain Ronald



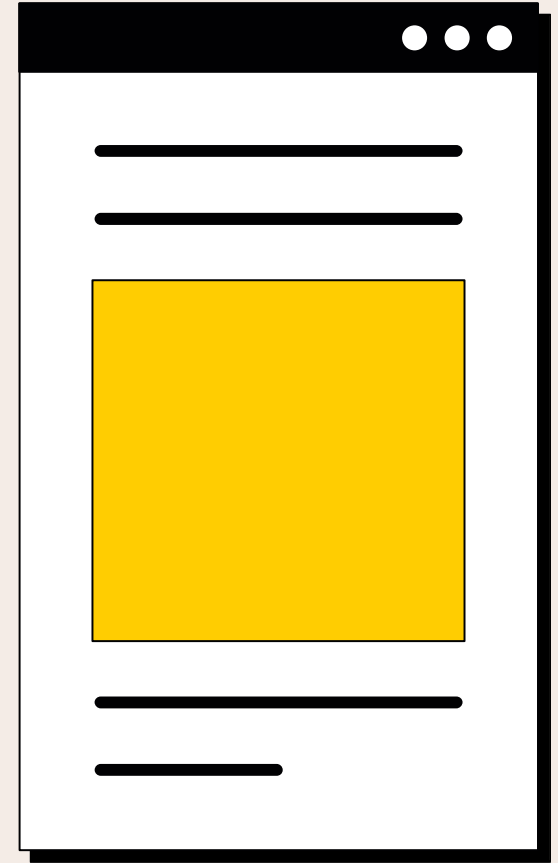


# Infraestructura como Código

A

# 1. Introducción a IaC

Infraestructura como Código (IaC) es una forma de gestionar y configurar servidores, redes y otros recursos de TI mediante código en lugar de hacerlo manualmente. En lugar de entrar a cada servidor y configurarlo a mano.



# Herramientas



Terraform



Ansible



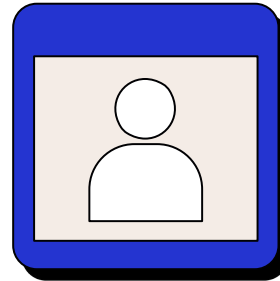
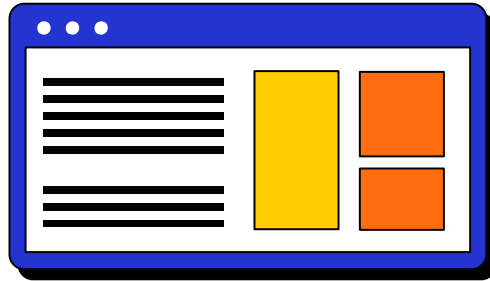
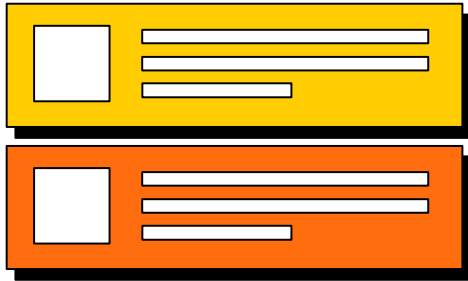
Pulumi



AWS

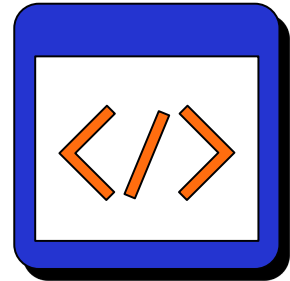
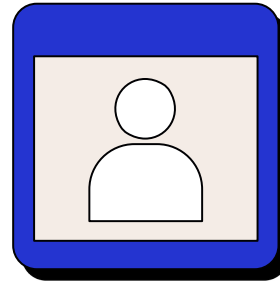
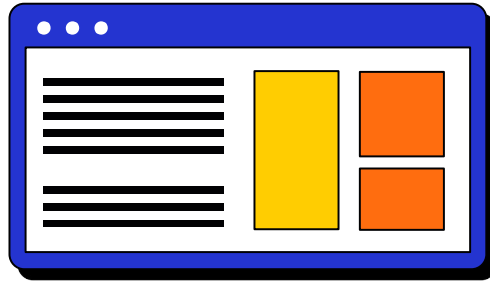
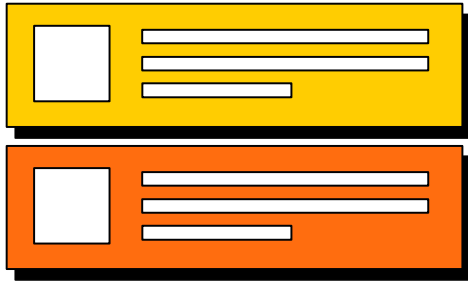
# Patrones para módulos

- **Red (network):** Define las VPCs, subredes, reglas de firewall.
- **Base de datos (database):** Configura los servidores de bases de datos.
- **Aplicación (application):** Define los servidores donde se ejecutará la aplicación.



# Patrones para dependencias

- **Inputs (entradas):** Variables que un módulo recibe.
- **Outputs (salidas):** Datos generados por un módulo que otros pueden usar.



# Tarea Teorica: Terraform



Terraform es una herramienta de Infraestructura como Código (IaC) de código abierto creada por HashiCorp. Permite definir y gestionar infraestructura en la nube de manera declarativa usando archivos de configuración escritos en HCL (HashiCorp Configuration Language).



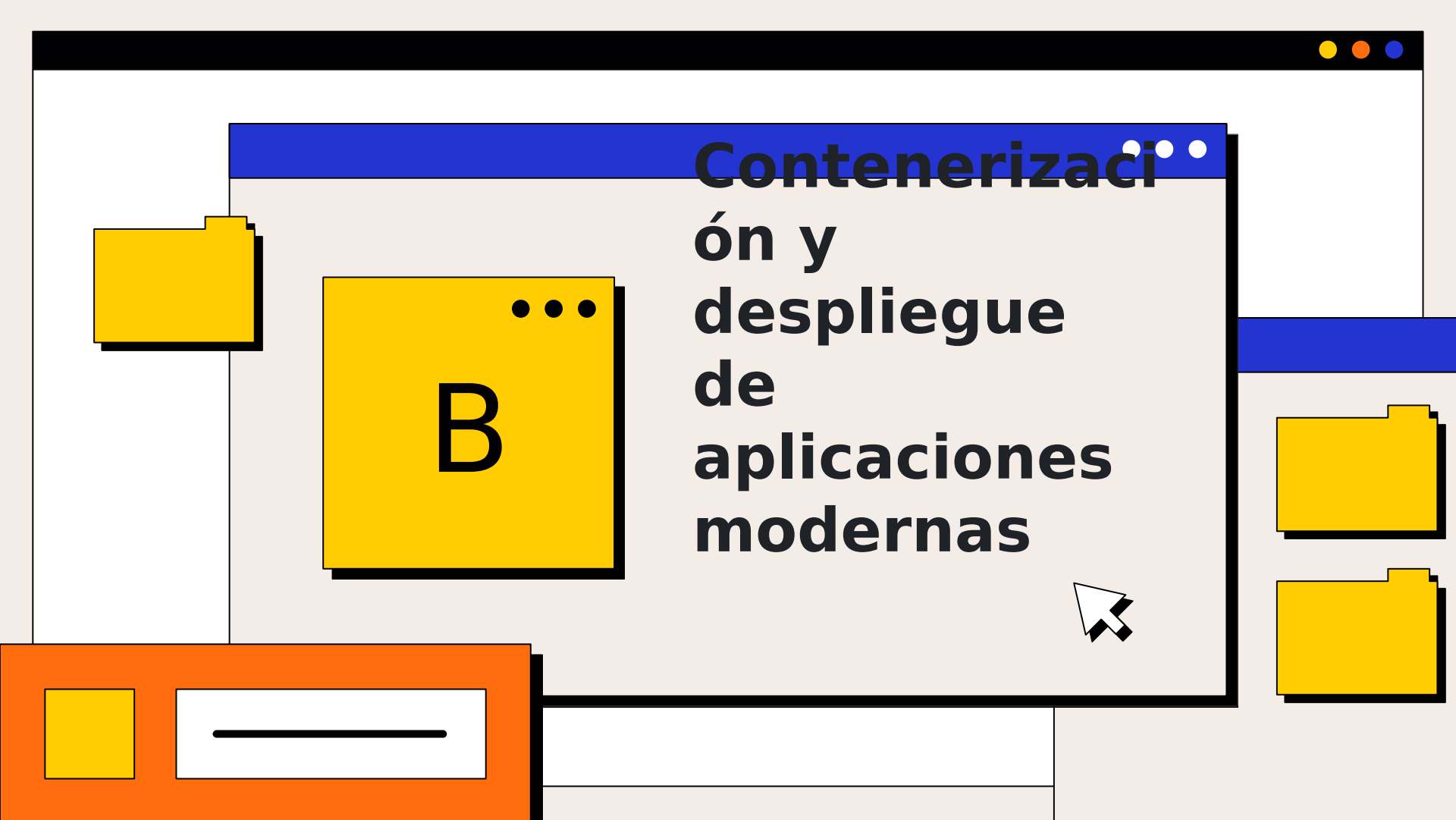
Terraform organiza la infraestructura en módulos reutilizables, lo que facilita la administración y escalabilidad.



# Network, database y application. Justificar la jerarquía elegida.

- Separar network, database y application facilita la reutilización y el mantenimiento.
- Permite gestionar configuraciones específicas para dev y prod sin modificar los módulos base.



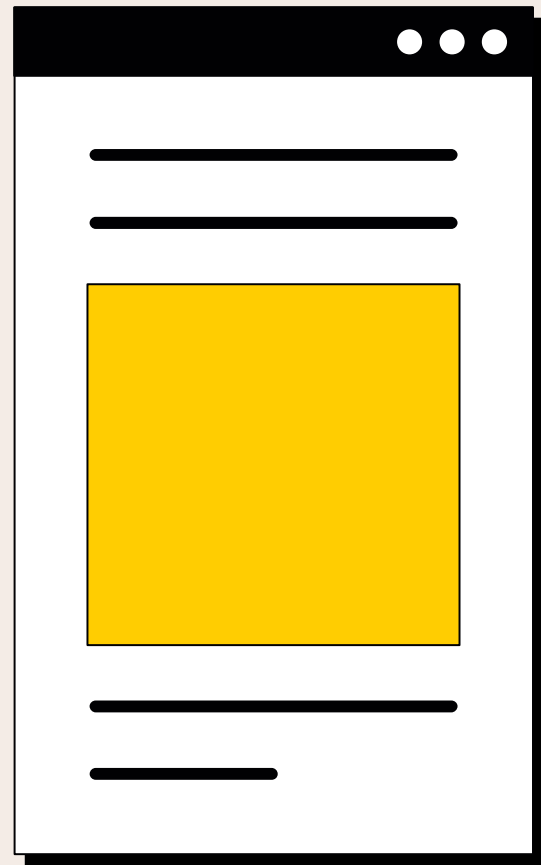


# Contenerización y despliegue de aplicaciones modernas

# ¿Qué son los contenedores?


Los contenedores son entornos ligeros y aislados que incluyen el código, librerías y dependencias necesarias para ejecutar una aplicación.

- VMs: Requieren un sistema operativo completo y son más pesadas.
- Contenedores: Comparten el kernel del sistema y son más eficientes.





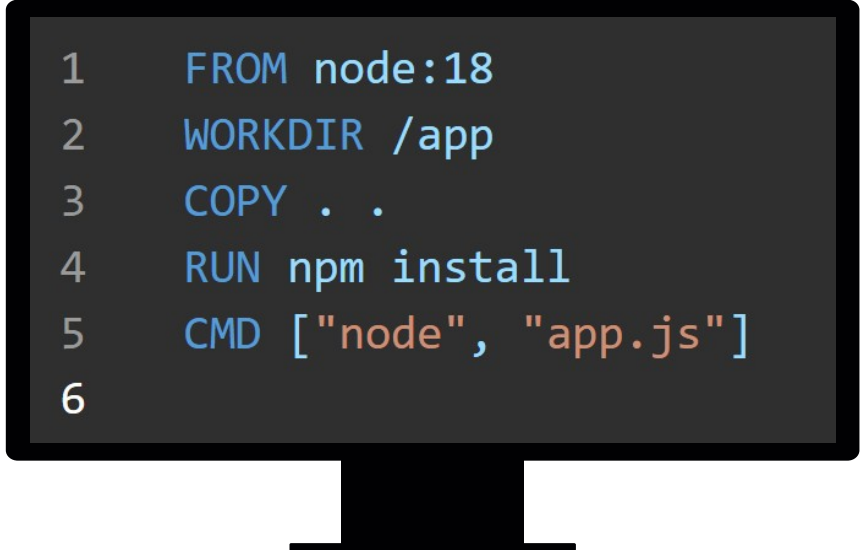
# Imagen vs Contenedor

- **Imagen:** Plantilla con el entorno de ejecución.
  - **Contenedor:** Instancia en ejecución de una imagen, su uso en desarrollo se pueden reconstruir imágenes con cambios, en producción se usan versiones estables.
- 

# Dockerfile: Estructura básica

Creamos una imagen de Docker:

1. Imagen base
2. Definimos el directorio de trabajo
3. Copiamos los archivos al contenedor
4. Ejecutamos comandos
5. Definimos el comando de inicio



```
1 FROM node:18
2 WORKDIR /app
3 COPY . .
4 RUN npm install
5 CMD ["node", "app.js"]
6
```



# Orquestación con Kubernetes

Permite administrar y escalar contenedores automáticamente.

<b>Pod</b>	Unidad mínima que contiene uno o más contenedores.
<b>Services</b>	Expone aplicaciones dentro o fuera del clúster.
<b>Deployment</b>	Gestiona el ciclo de vida de los pods.
<b>ReplicaSet</b>	Mantiene la cantidad deseada de pods.
<b>Namespaces</b>	Separan recursos dentro de un clúster
<b>ConfigMaps &amp; Secrets</b>	Permiten gestionar configuración y credenciales de manera segura.
<b>Ingress</b>	Controla el acceso HTTP y HTTPS hacia los servicios dentro del clúster

```
1  apiVersion: apps/v1
2  kind: Deployment
3  ∨ metadata:
4    name: mi-app
5  ∨ spec:
6    replicas: 3
7  ∨ selector:
8  ∨   matchLabels:
9    app: mi-app
10 ∨ template:
11 ∨   metadata:
12 ∨     labels:
13     app: mi-app
14 ∨   spec:
15 ∨     containers:
16 ∨       - name: mi-app
17         image: mi-app:latest
18 ∨       ports:
19         - containerPort: 80
20 |
```

# Manifiestos en YAML:

1. Versión de la API de Kubernetes.
2. Tipo de recurso.
4. Nombre del recurso.
6. Número de réplicas del pod.

# Estrategias de despliegue en Kubernetes

Rolling  
Updates

Actualiza

Canary  
Releases

Despliega una  
versión nueva solo a  
un porcentaje de  
usuarios

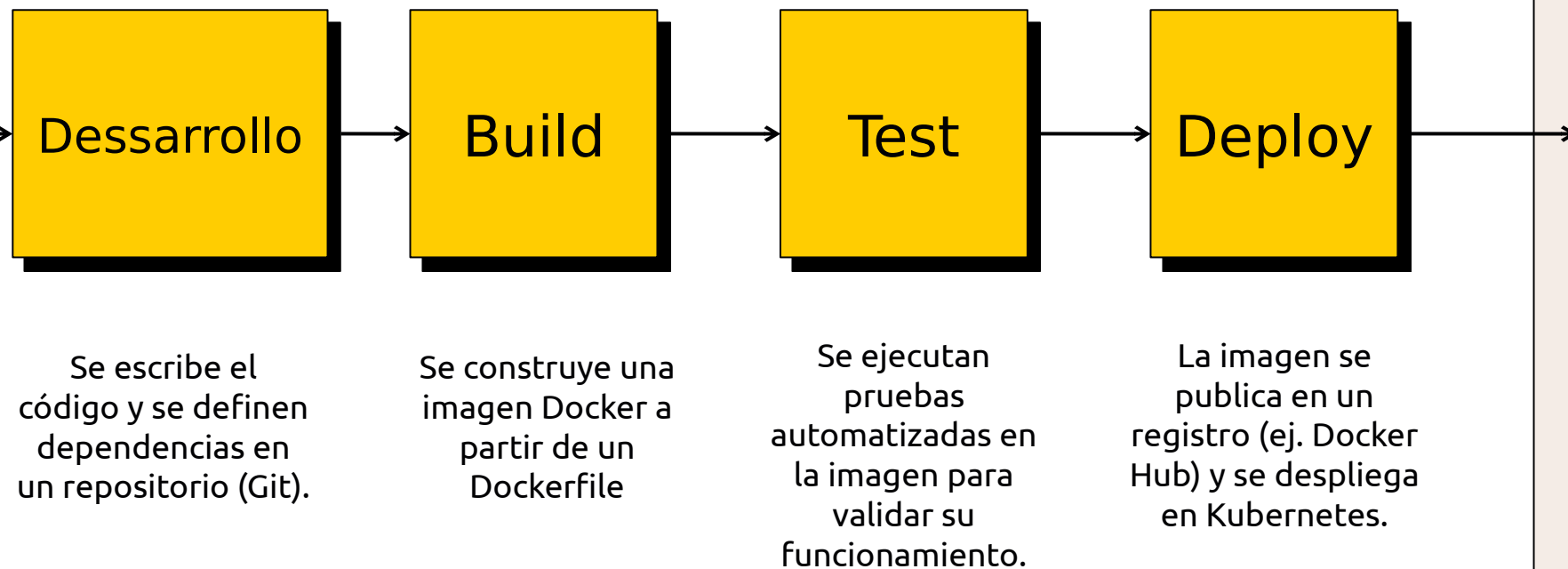
MBlue-Green  
Deployments

Mantiene dos  
versiones y cambia  
el tráfico entre ellas

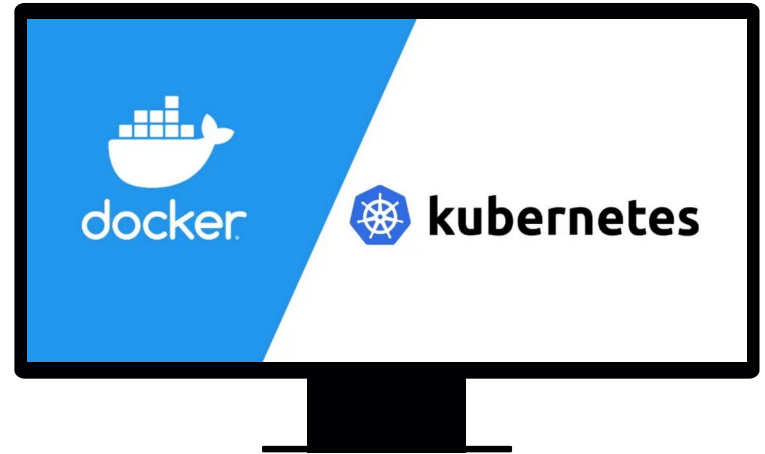


# Desplegando código:

## Ciclo de vida del despliegue:

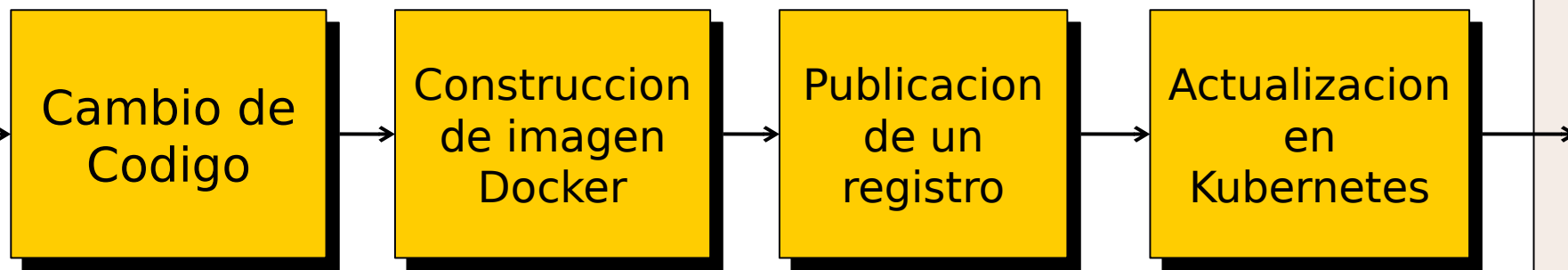


# Implementación de Docker y Kubernetes en pipelines



# Tarea Teórica:

## Flujo simple de despliegue

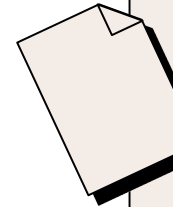
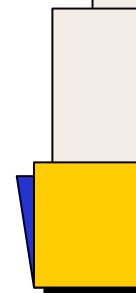





# Tarea Teorica:



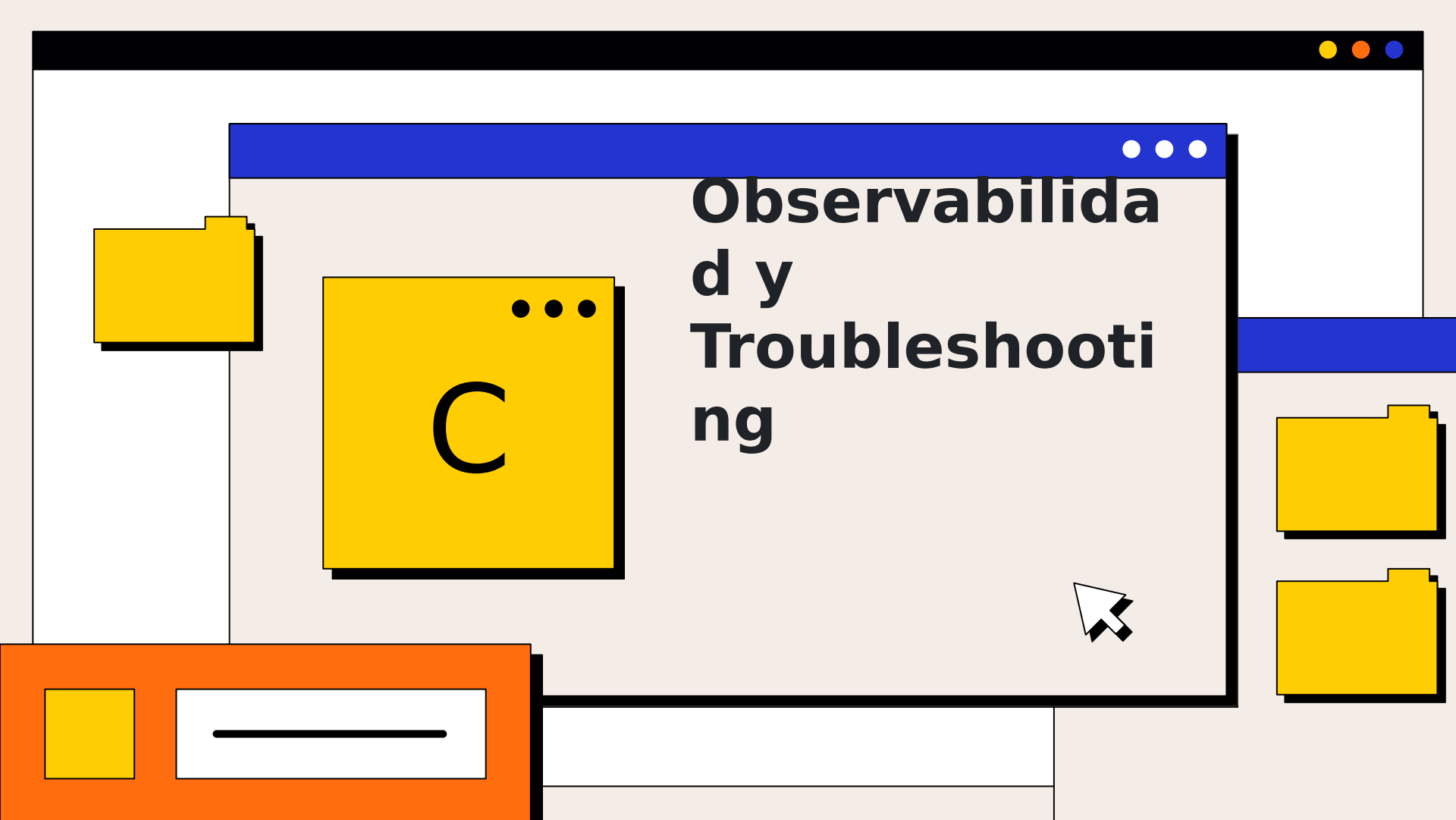
## Ventajas de usar Kubernetes para escalar una aplicación en un evento de alto tráfico.

- **Escalabilidad automática:** Kubernetes puede aumentar el número de réplicas de pods según la demanda con el Horizontal Pod Autoscaler (HPA).
  - **Balanceo de carga:** Distribuye el tráfico entre múltiples pods mediante Services, evitando sobrecargas en un solo nodo.
  - **Resiliencia y autorecuperación:** Si un pod falla, Kubernetes lo reemplaza automáticamente para mantener la disponibilidad del servicio.
- 
- 
- 



# Estrategias de troubleshooting

<b>Triage de problemas</b>	Se usan dashboards y alertas para identificar incidentes críticos.
<b>Diagnóstico</b>	Se revisan logs, métricas y trazas en tiempo real para encontrar la causa raíz.
<b>Resolución</b>	Se implementan soluciones como rollback a una versión estable, escalado de recursos o ajustes en la infraestructura.

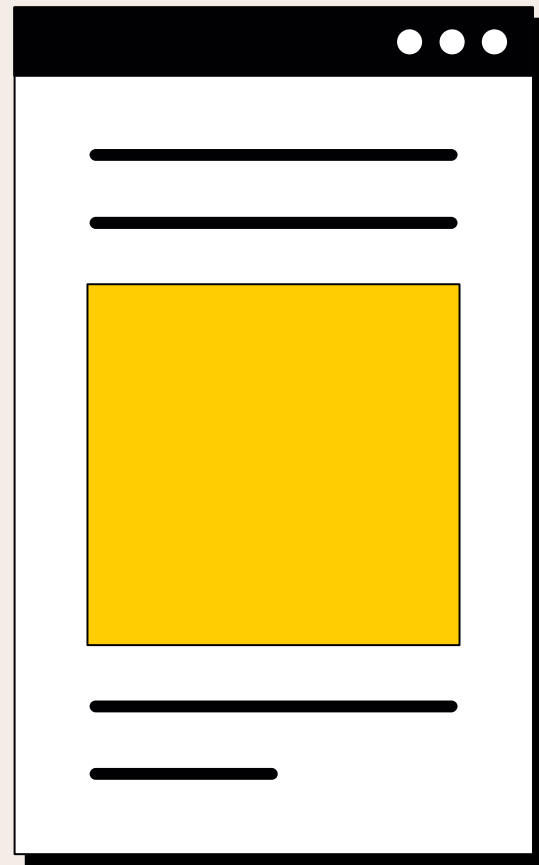


# Observabilidad y Troubleshooting

C

# Introducción de observabilidad

La observabilidad es la capacidad de un sistema para proporcionar información sobre su estado interno a través de logs, métricas y trazas, permitiendo identificar y solucionar problemas más allá del monitoreo tradicional.





## Diferencias con monitoreo

Mientras el monitoreo solo alerta sobre fallos, la observabilidad permite entender por qué ocurren, facilitando el diagnóstico y la optimización del sistema.



## Ventajas:

- Detección temprana de errores: Permite reaccionar rápidamente a fallos.
- Optimización de los recursos: Identifica cuellos de botella y mejora el rendimiento.
- Ayuda a entender patrones de uso y comportamiento del usuario.



# Herramientas



Prometheus



Grafana

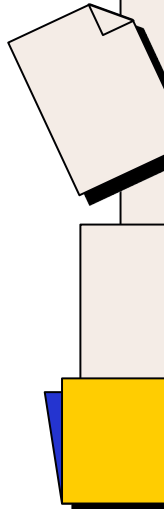



ELK Stack



# Tarea Teorica:

## Integración de Prometheus y Grafana con Kubernetes

- 
1. **Prometheus:** Se integra con Kubernetes mediante service discovery, detectando automáticamente los pods y servicios etiquetados para monitoreo.
  2. **Grafana:** Es una herramienta de visualización que se conecta a Prometheus como fuente de datos. Permite crear dashboards interactivos y gráficos personalizados para observar las métricas recolectadas de Kubernetes.
- 


# Tarea Teorica: Metricas y Alertas minimas

## Métricas clave

- Latencia de peticiones HTTP (`http_request_duration_seconds`) → Para medir tiempos de respuesta.
- Uso de CPU y memoria (`container_cpu_usage_seconds_total`, `container_memory_usage_bytes`) → Para detectar sobrecargas.
- Tasa de errores HTTP (`http_requests_total{status=~"5.."}`) → Para identificar fallos en el servicio.

## Alertas

- Alta latencia (**>500ms en 5 min**) → "La aplicación responde lentamente."
- Uso de CPU **> 80% durante 10 min** → "Posible sobrecarga del servidor."
- Errores HTTP **5XX > 5% de las peticiones en 5 min** → "Alta tasa de errores detectada."

The background is a stylized desktop environment. It features several yellow folders: one on the left, one on the right, and two stacked on the right. There are also several windows: a large central window with a blue title bar and three white dots, a smaller yellow window with a black 'D' and three dots, and an orange dock at the bottom left with a yellow square and a white rectangle containing a horizontal line. A white mouse cursor is pointing at the bottom right of the central window.

# CI/CD (Integración continua / Despliegue continuo)



# Conceptos:

<b>Integración continua (CI):</b>	Es el proceso de automatizar la construcción y pruebas de una aplicación cada vez que se realiza un commit.
<b>Despliegue continuo (CD):</b>	Automatiza la entrega de nuevas versiones de la aplicación a entornos de prueba o producción.
<b>Herramientas:</b>	Jenkins, GitLab CI, GitHub Actions, entre otras, permiten configurar pipelines de CI/CD de manera eficiente.

# Diseñando un pipeline:

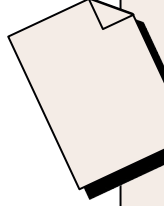
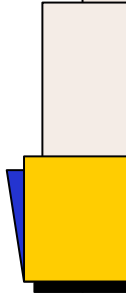

- **Fases:** Un pipeline típico incluye las fases de construcción (build), pruebas (test), análisis de calidad (escaneos de seguridad) y despliegue (deploy).
- **Jobs y Stages:** Los pipelines se dividen en jobs (tareas específicas como compilación o pruebas) agrupados en fases (stages).



# Tarea Teorica:



## Relevancia de las pruebas automáticas en el pipeline:

- ❖ **Pruebas unitarias:** Verifican el correcto funcionamiento de componentes individuales (unidades) del código.
  - ❖ **Pruebas de integración:** Evalúan cómo interactúan diferentes módulos o servicios entre sí.
  - ❖ **Pruebas de seguridad:** Detectan vulnerabilidades y posibles brechas de seguridad en el código o en las dependencias utilizadas.
- 
- 
- 



Thank you!

