Módulo 8
Fundamentos de integración continua

Pruebas y Automatización en Integración Continua

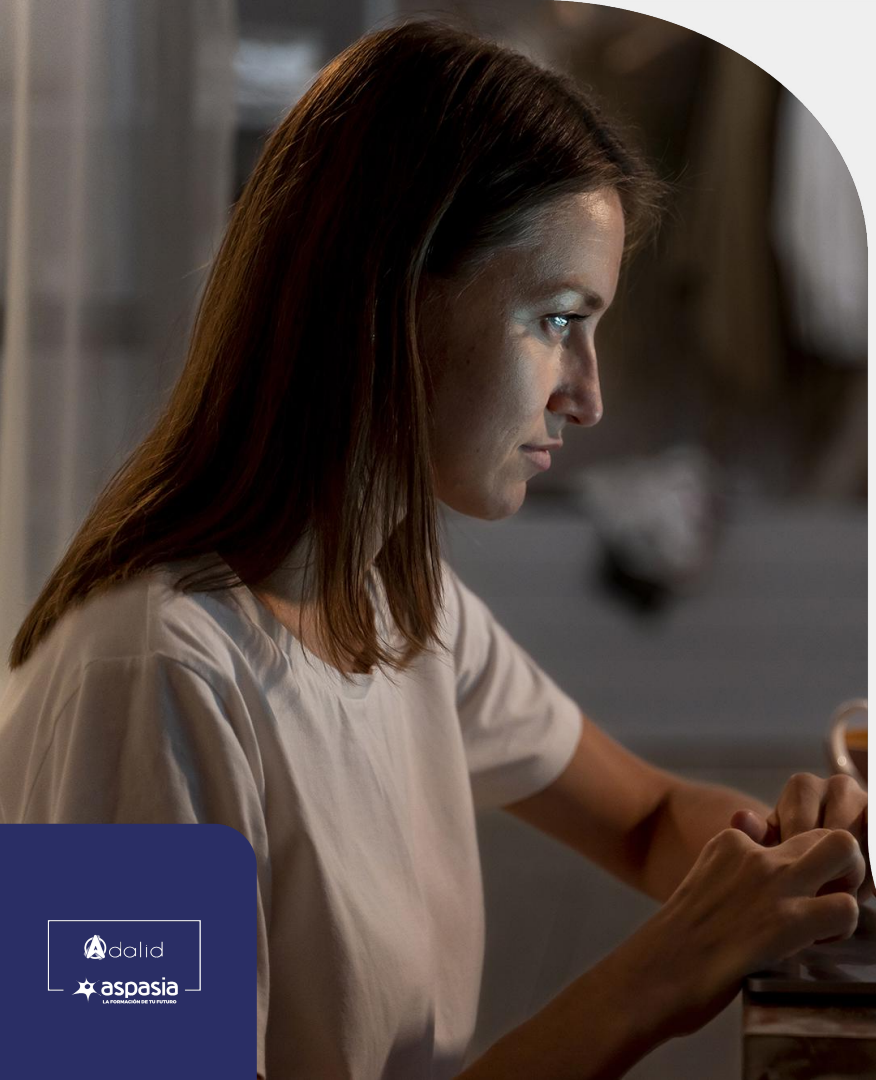
Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

Módulo 8

AE 2.1

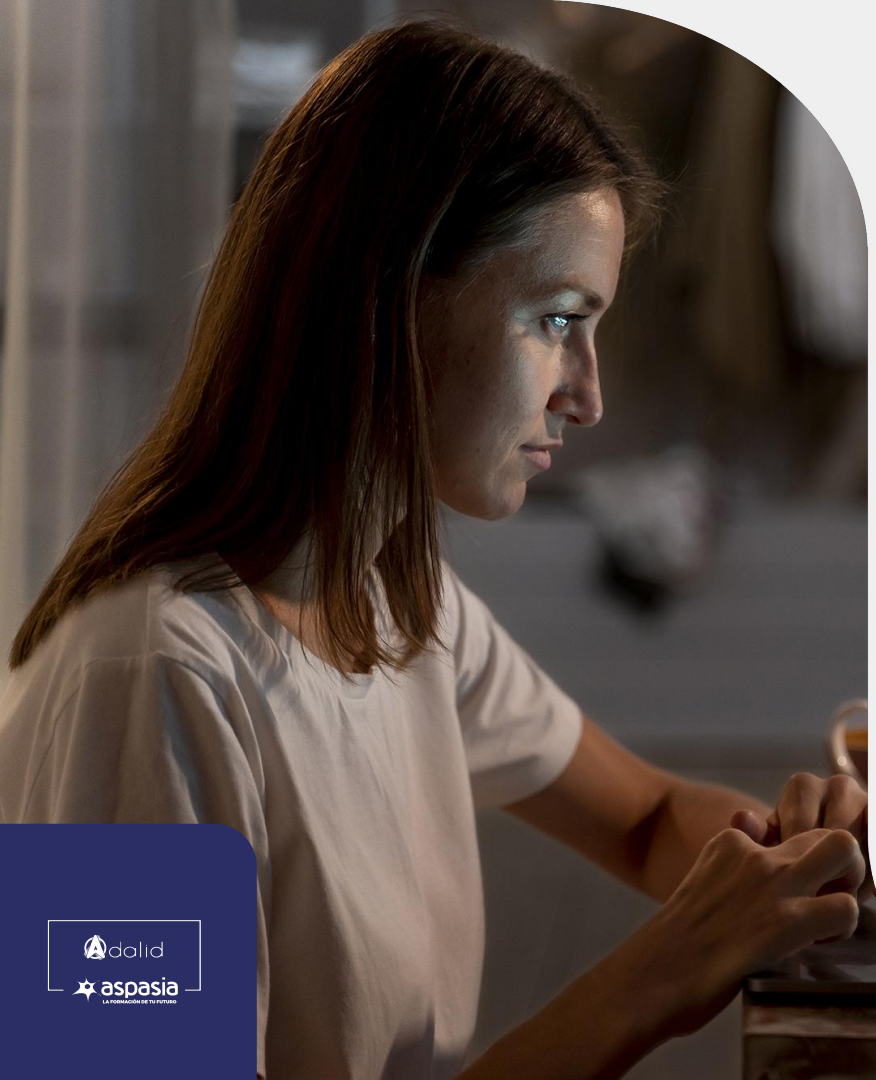
OBJETIVOS

Comprender la importancia del testing y la automatización en CI/CD para garantizar calidad y velocidad.
Aprender a integrar pruebas automatizadas en pipelines de CI/CD con herramientas como Jenkins.
Conocer herramientas de monitoreo y gestión de logs para mejorar la estabilidad en DevOps.



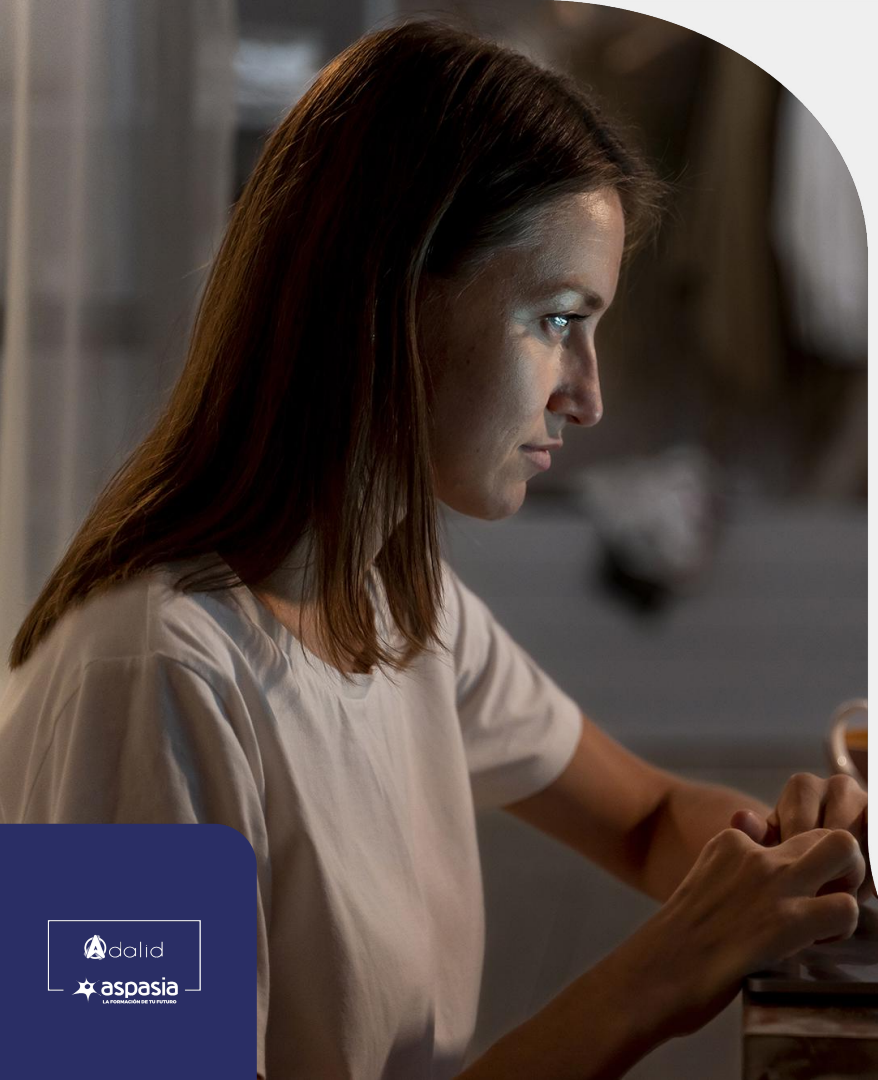
¿QUÉ VAMOS A VER?

- Pruebas y Automatización en Integración Continua
  - Principios de Agile Testing y el rol del tester en entornos ágiles.
  - Tipos de pruebas en CI/CD: unitarias, integración, sistema y aceptación.
  - Automatización de pruebas: herramientas y su integración en el pipeline.
  - Test-Driven Development (TDD) y su aplicación en CI/CD.



¿QUÉ VAMOS A VER?

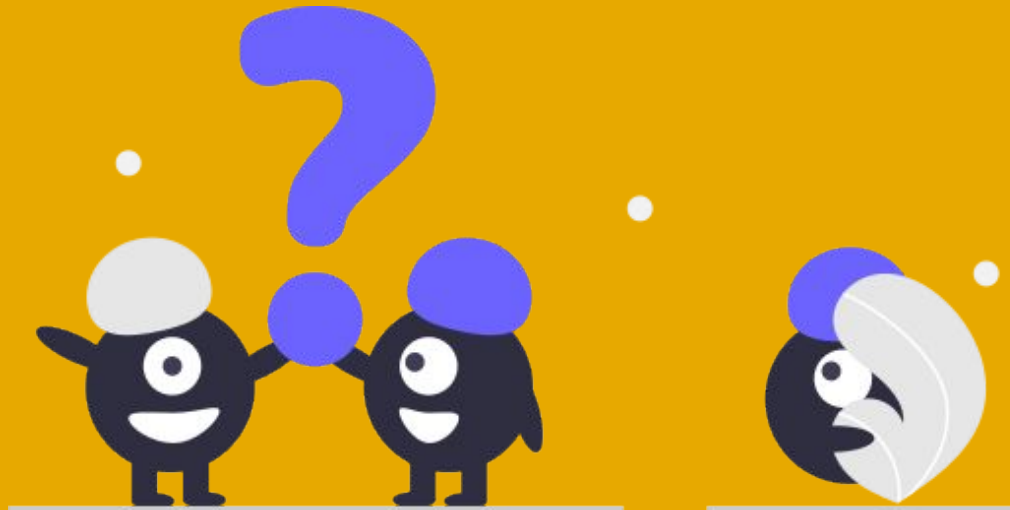
- Implementación de Pipelines de CI/CD
  - Jenkins: Instalación, configuración y creación de pipelines con JenkinsFile.
  - Automatización de despliegues en entornos de prueba y producción.
- Infraestructura y Operaciones en DevOps
  - Infraestructura como código (IaC) y modelos de servicio en la nube (IaaS, PaaS, SaaS).
  - Kubernetes y orquestación de contenedores.

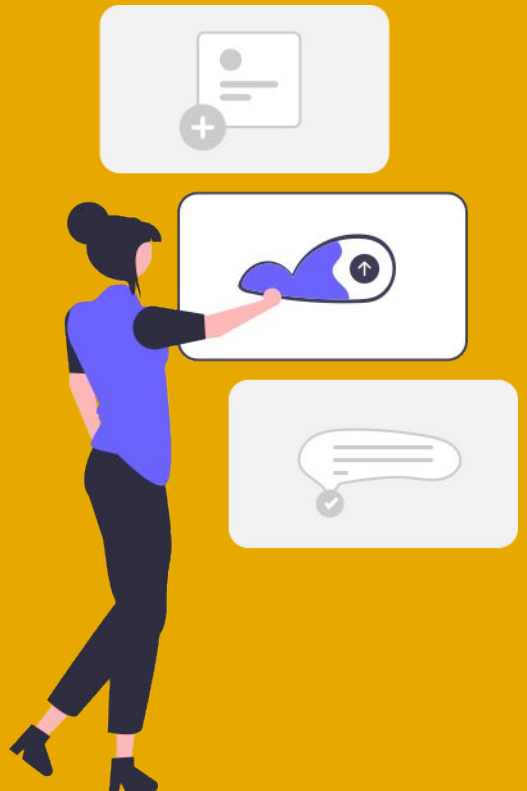


¿QUÉ VAMOS A VER?

- Monitoreo y Gestión de Logs
  - Monitoreo continuo en DevOps: importancia y herramientas (ELK, GFG).
  - Gestión de logs y manejo de incidentes en entornos CI/CD.

¿Conoces el término CI?

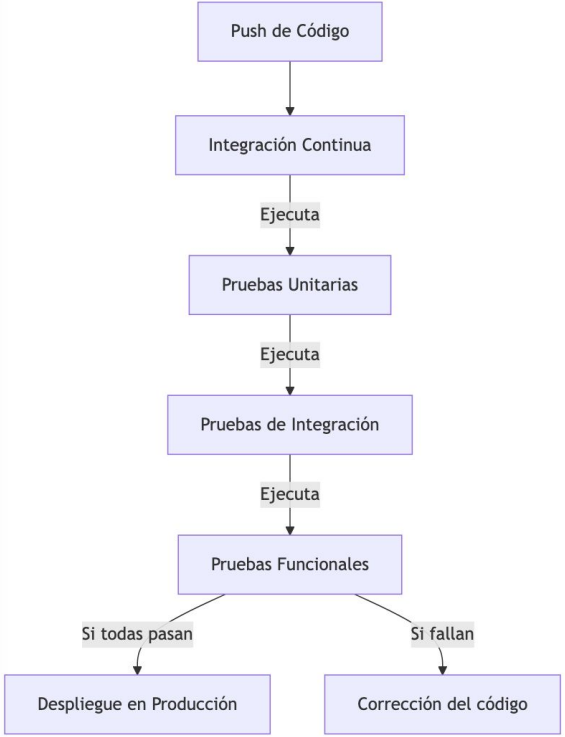




Pruebas en un Entorno de Integración Continua

¿Por qué son importantes las pruebas en CI/CD?

Las pruebas en un entorno de Integración Continua (CI/CD) aseguran que el código nuevo **no rompa la funcionalidad existente** y que el software esté **siempre en condiciones de ser desplegado.**



¿Por qué son importantes las pruebas en CI/CD?

Beneficios de las pruebas en CI/CD:

- Detección temprana de errores.
- Reducción del tiempo de entrega.
- Mayor calidad del software.
- Minimización de riesgos en producción.

Testing en un entorno ágil

Agile Testing: Concepto, Principios y Prácticas

- Las pruebas son continuas en el ciclo de desarrollo.
- El equipo completo es responsable de la calidad.
- Se busca retroalimentación rápida para corregir errores a tiempo.
- Se aplican pruebas automatizadas y manuales en cada iteración.

Testing en un entorno ágil

El rol del tester en entornos ágiles:

- Colabora con desarrolladores en la definición de pruebas.
- Escribe casos de prueba antes de la implementación.
- Automatiza pruebas para acelerar validaciones.
- Reporta y da seguimiento a errores.

Test-Driven Development (TDD)

El **Desarrollo Guiado por Pruebas (TDD)** es una metodología donde **se escriben pruebas antes de desarrollar el código.**

Ciclo de TDD:

1. Escribir una prueba fallida.
2. Implementar la funcionalidad mínima para pasar la prueba.
3. Refactorizar el código y volver a ejecutar la prueba.

Ejemplo de TDD en JavaScript con Jest:

```
// Prueba antes de implementar (Falla inicialmente)
test("sumar 2 + 3 debería dar 5", () => {
  expect(sumar(2, 3)).toBe(5);
});

// Implementación mínima para pasar la prueba
function sumar(a, b) {
  return a + b;
}
```

Tipos de Pruebas en CI/CD

Tipos de pruebas:

Prueba	Objetivo	Ejemplo
Pruebas Unitarias	Verificar el comportamiento de funciones individuales.	Validar que `sumar(2,3)` retorne `5`.
Pruebas de Integración	Comprobar que los módulos funcionan juntos.	Validar que el backend se conecte correctamente con la base de datos.
Pruebas de Sistema	Evaluar el comportamiento de la app en su totalidad.	Verificar que un usuario pueda registrarse y hacer login.
Pruebas de Aceptación	Validar que el sistema cumpla los requisitos del cliente.	Revisar que una tienda online permita realizar compras.
Pruebas de Humo	Confirmar que la aplicación es estable tras cambios.	Verificar que el login y los formularios principales funcionan.

Herramientas de Automatización de Pruebas

Herramientas Populares:

Tipo de Prueba	Herramienta
Unitarias	Jest, Mocha, JUnit
Integración	Postman, Newman, Cypress
Sistema	Selenium, Playwright
Carga y rendimiento	JMeter, Gatling

Integración de Pruebas en un Pipeline de CI/CD

Ejemplo de pipeline con pruebas en GitHub Actions:

Archivo: .github/workflows/ci.yml

```yaml
name: CI/CD Pipeline

on: [push]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Clonar el código
        uses: actions/checkout@v2
      - name: Instalar dependencias
        run: npm install
      - name: Ejecutar pruebas unitarias
        run: npm test
```

El pipeline ejecuta las pruebas en cada push al repositorio.

Qué es Jenkins y su función en CI/CD

Jenkins es una herramienta de **Integración y Entrega Continua (CI/CD)** que **automatiza la construcción, pruebas y despliegue de aplicaciones.**

Beneficios de Jenkins:

- Código probado automáticamente.
- Integración con herramientas como GitHub y Docker.
- Fácil personalización con plugins.

Instalación y Configuración de Jenkins

En Linux:

```
sudo apt update
sudo apt install openjdk-11-jdk

wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

sudo apt update
sudo apt install jenkins
```

Instalación y Configuración de Jenkins

Acceder a Jenkins:

1. Abrir http://localhost:8080.
2. Iniciar sesión con **admin**.
3. Configurar plugins y usuarios.

Jenkins permite automatizar todo el pipeline de CI/CD.

Creación de un Pipeline con Jenkinsfile

Ejemplo de Jenkinsfile para CI/CD:

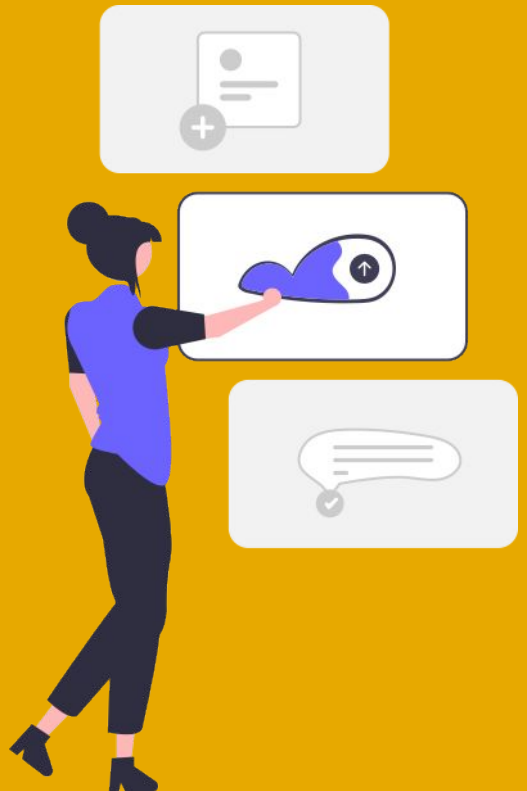
Este Jenkinsfile ejecuta pruebas y despliega la app automáticamente.

```
pipeline {
    agent any
    stages {
        stage('Clonar Repositorio') {
            steps {
                git 'https://github.com/mi-repo.git'
            }
        }

        stage('Instalar Dependencias') {
            steps {
                sh 'npm install'
            }
        }

        stage('Ejecutar Pruebas') {
            steps {
                sh 'npm test'
            }
        }

        stage('Despliegue en Testing') {
            steps {
                sh 'echo "Desplegando en entorno de pruebas..."'
            }
        }
    }
}
```



Infraestructura en DevOps

adalid
aspasia

Conceptos de Infraestructura y Plataforma

En **DevOps**, la infraestructura no es solo servidores y redes, sino un conjunto de recursos gestionados como código para escalar, automatizar y optimizar el desarrollo.

Elementos clave de una infraestructura en DevOps:

- Servidores y máquinas virtuales (on-premise o cloud).
- Contenedores y microservicios.
- Almacenamiento y bases de datos.
- Networking y balanceo de carga.

Infraestructura Evolutiva en DevOps

Ejemplo de Evolución de Infraestructura:

Antes de DevOps	Con DevOps
Servidores físicos	Infraestructura en la nube
Configuración manual	Automatización con scripts
Despliegues lentos	CI/CD con infraestructura dinámica

Infraestructura Evolutiva en DevOps

Comparación de Infraestructura:

Factor	On-Premise	Cloud
Gestión	Propia del equipo	Tercerizada (AWS, Azure, GCP)
Escalabilidad	Limitada	Automática
Costo	Alto (hardware, mantenimiento)	Pago por uso
Tiempo de despliegue	Lento	Rápido

¿Qué es IaC y por qué es importante?

La **Infraestructura como Código (IaC)** permite definir infraestructura en **archivos de configuración**, facilitando la automatización y escalabilidad.

Beneficios de IaC:

- Despliegues más rápidos y repetibles.
- Evita configuraciones manuales.
- Escalabilidad dinámica.
- Mejora la colaboración entre equipos.

¿Qué es IaC y por qué es importante?

Ejemplo de Código Terraform para Crear un Servidor en AWS:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

IaC permite definir y versionar la infraestructura como si fuera código.

Modelos de Servicio en la Nube (IaaS, PaaS, SaaS)

Tipos de Servicios Cloud:

Modelo	Descripción	Ejemplo
IaaS	Infraestructura como servicio (Servidores, redes)	AWS EC2, Google Compute Engine
PaaS	Plataforma como servicio (Bases de datos, herramientas)	Heroku, Google App Engine
SaaS	Software como servicio (Aplicaciones listas para usar)	Gmail, Slack, Dropbox

Orquestadores de Contenedores y Kubernetes

¿Qué es un Orquestador de Contenedores?

- Herramienta que gestiona múltiples contenedores en producción.
- Kubernetes es el orquestador más usado para escalar aplicaciones.

```yaml
apiVersion: apps/v1
kind: Deployment

metadata:
  name: mi-app

spec:
  replicas: 3
  selector:
    matchLabels:
      app: mi-app
  template:
    metadata:
      labels:
        app: mi-app
    spec:
      containers:
      - name: mi-app
        image: mi-app:v1
        ports:
        - containerPort: 80
```

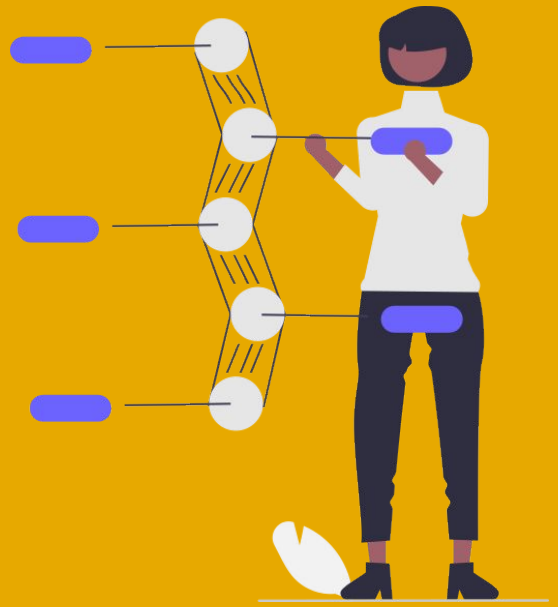
Monitoreo Continuo en DevOps

¿Por qué es importante el monitoreo?

- Detectar fallos antes de que afecten a los usuarios.
- Medir el rendimiento de la aplicación.
- Prevenir problemas con alertas tempranas.

Stack	Descripción
ELK (Elasticsearch, Logstash, Kibana)	Análisis y visualización de logs.
Grafana + Prometheus	Métricas en tiempo real para sistemas y aplicaciones.

Resumen de lo aprendido 🧑‍🎓



Adalid aspasia

Resumen de lo aprendido

- Las pruebas automatizadas en CI/CD garantizan calidad en cada entrega y optimizan el desarrollo ágil.
- Jenkins permite crear pipelines de integración y despliegue continuo con automatización eficiente.
- Infraestructura como código y orquestadores como Kubernetes facilitan escalabilidad y flexibilidad.
- El monitoreo continuo con stacks como ELK y GFG mejora la observabilidad y respuesta operativa.

GRACIAS POR TU ATENCIÓN

Nos vemos en la próxima clase

Chile
Adalid
aspasia
LA FORMACIÓN DE TU FUTURO