CORFO
BecasCapital Humano

dalid
aspasia
LA FORMACIÓN DE TU FUTURO

MÓDULO 8

FUNDAMENTOS DE INTREGRACIÓN CONTINÚA

Manual Módulo 8: Fundamentos de Integración Continua

Introducción

En este módulo, explicaremos los fundamentos de la **Integración Continua (CI)**, una práctica esencial en el desarrollo moderno de software. La integración continua permite a los equipos de desarrollo automatizar la compilación, pruebas y despliegue de código, asegurando una entrega rápida y confiable de software de alta calidad. Aprenderás cómo implementar un pipeline de CI efectivo utilizando herramientas y técnicas clave que facilitan la colaboración entre desarrolladores y equipos de operaciones, alineándose con los principios de **DevOps**. Durante el módulo, abordaremos los conceptos básicos de DevOps, el ciclo de vida de la integración continua, y cómo contenedores como **Docker** desempeñan un rol fundamental en este proceso. También se profundizará en las herramientas de prueba automatizadas y los sistemas de control de versiones que soportan un pipeline de CI eficiente. Al finalizar, tendrás el conocimiento necesario para implementar un circuito completo de **Integración Continua** en proyectos de desarrollo, mejorando la calidad del software y acelerando los ciclos de entrega.

1. Fundamentos de DevOps

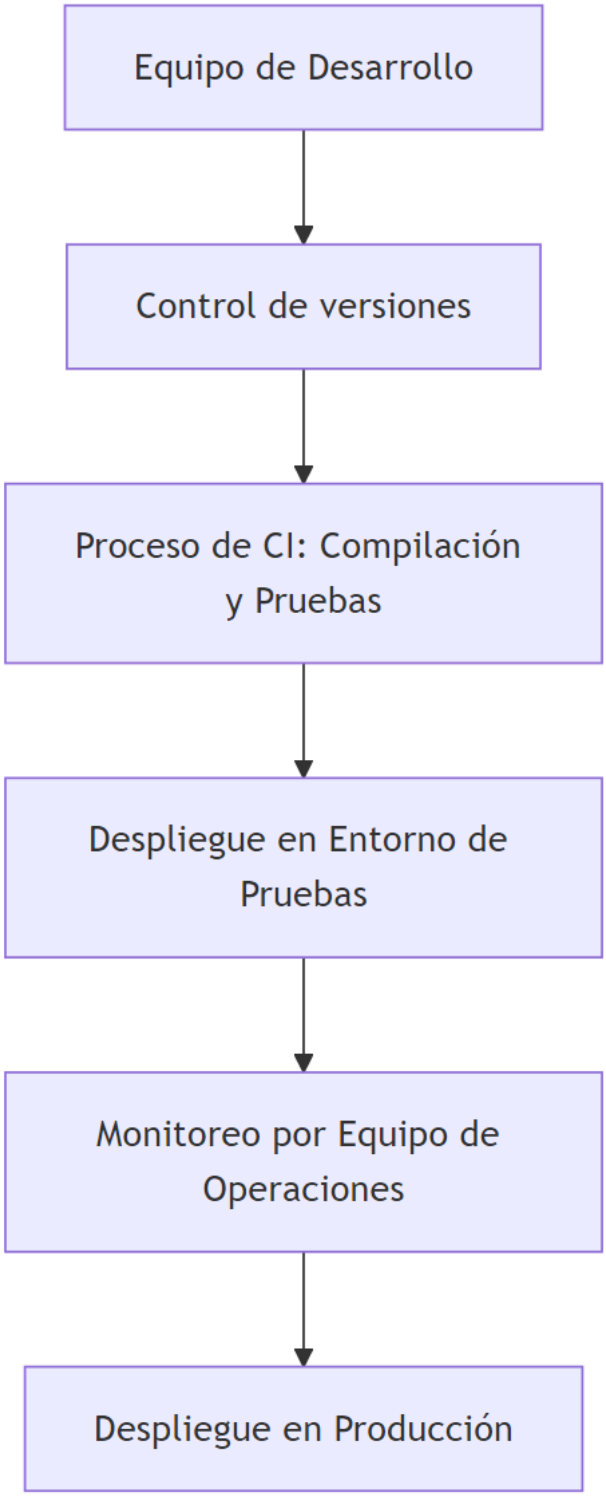
1.1 Qué es DevOps y cuál es su propósito

DevOps es una filosofía de trabajo que combina las prácticas y herramientas de los equipos de **desarrollo (Dev)** y **operaciones (Ops)** para mejorar la colaboración, acelerar la entrega de software y automatizar procesos críticos como la integración y el despliegue de aplicaciones. Su propósito principal es reducir el ciclo de desarrollo y garantizar la estabilidad, escalabilidad y calidad del software a través de una entrega continua.

El modelo DevOps es clave en la **Integración Continua (CI)** y la **Entrega Continua (CD)**, ya que ambos equipos trabajan juntos para implementar cambios de código de manera eficiente, confiable y frecuente. DevOps no es solo una práctica técnica, sino una **cultura de colaboración**.

Ejemplo de flujo en DevOps:

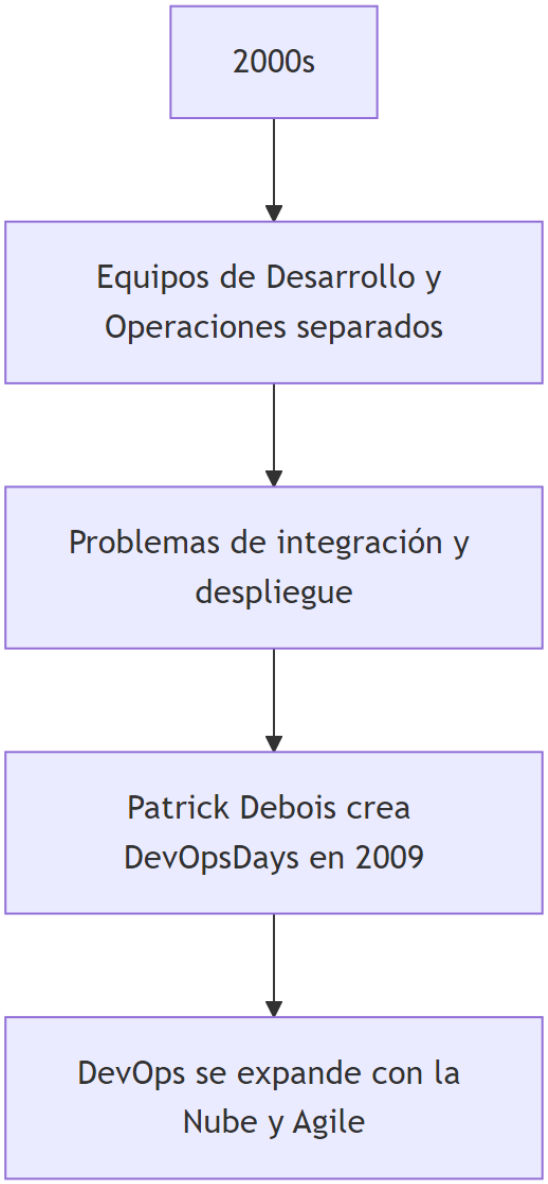
1. El equipo de desarrollo escribe el código y lo almacena en un sistema de control de versiones.
2. Un proceso automatizado de CI compila el código, ejecuta pruebas y despliega el software en un entorno de pruebas.
3. El equipo de operaciones monitorea el entorno y asegura su estabilidad antes de realizar el despliegue en producción.


Equipo de Desarrollo
Control de versiones
Proceso de CI: Compilación y Pruebas
Despliegue en Entorno de Pruebas
Monitoreo por Equipo de Operaciones
Despliegue en Producción

1.2 Origen y evolución de DevOps

El término **DevOps** surgió a finales de la década de 2000, como una respuesta a la creciente brecha entre los equipos de desarrollo y operaciones. Tradicionalmente, estos equipos trabajaban de manera independiente, lo que resultaba en conflictos, errores y retrasos en la entrega del software.

Patrick Debois, uno de los pioneros de DevOps, organizó el primer evento de **DevOpsDays** en 2009, lo que marcó el inicio del movimiento DevOps. Desde entonces, ha evolucionado rápidamente con la adopción masiva de tecnologías como la **nube** y las **metodologías ágiles**, que han permitido a las organizaciones reducir los ciclos de entrega y mejorar la calidad de sus productos.

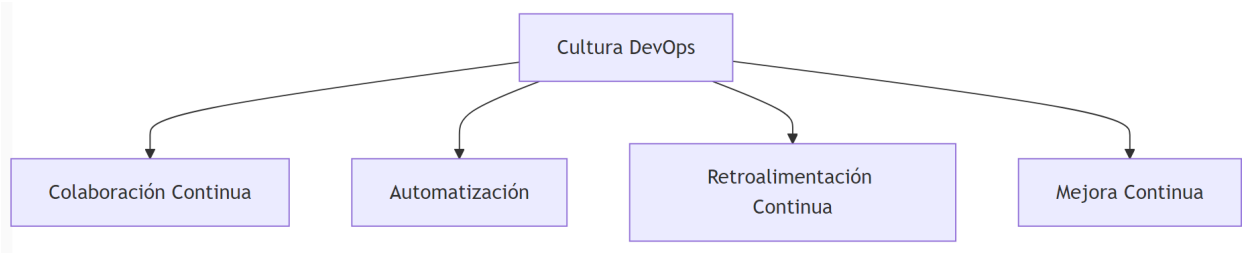


1.3 Cultura y principios DevOps

La **cultura DevOps** está basada en una serie de principios fundamentales que buscan mejorar la colaboración y la entrega de software:

- **Colaboración continua**: Fomenta la eliminación de silos entre los equipos de desarrollo y operaciones.

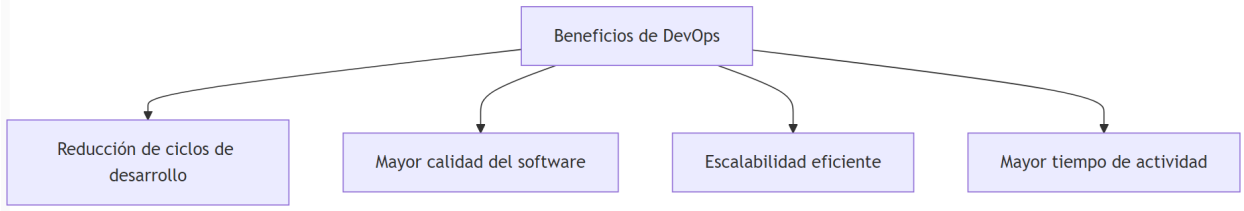
- **Automatización**: Automatiza procesos repetitivos, como la integración, el testing y el despliegue.
- **Retroalimentación continua**: Implementa mecanismos para recibir y actuar sobre la retroalimentación de manera rápida.
- **Mejora continua**: Siempre se busca optimizar procesos y herramientas para aumentar la eficiencia.



1.4 Beneficios de DevOps

Adoptar **DevOps** aporta varios beneficios clave a las organizaciones:

- **Reducción de ciclos de desarrollo**: La automatización y colaboración mejorada permiten un ciclo de desarrollo más corto y eficiente.
- **Mayor calidad del software**: Las pruebas automatizadas y el monitoreo continuo garantizan que los errores se detecten antes de que lleguen a producción.
- **Escalabilidad**: DevOps permite escalar aplicaciones y sistemas de manera eficiente sin comprometer su rendimiento.
- **Mayor tiempo de actividad**: Los sistemas son más estables y tienen menos interrupciones gracias al monitoreo proactivo y a la rápida resolución de problemas.

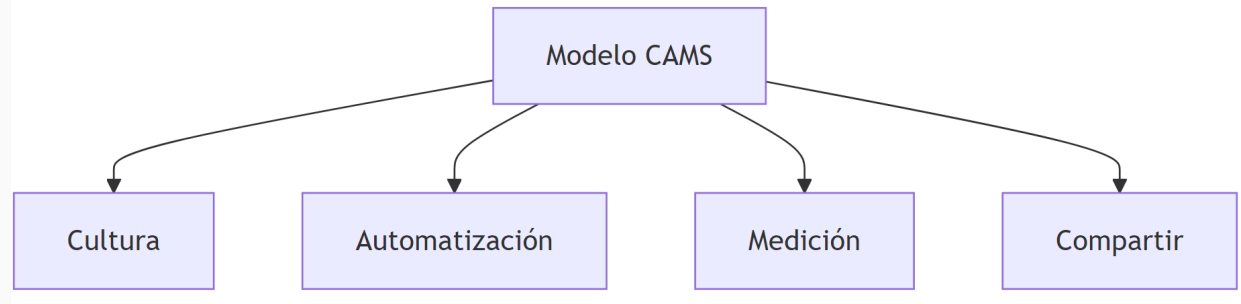


1.5 Modelo CAMS (Culture, Automation, Measurement, Sharing)

El modelo **CAMS** establece los cuatro pilares esenciales de **DevOps**:

1. **Culture (Cultura)**: Fomenta la colaboración entre equipos, eliminando los silos.

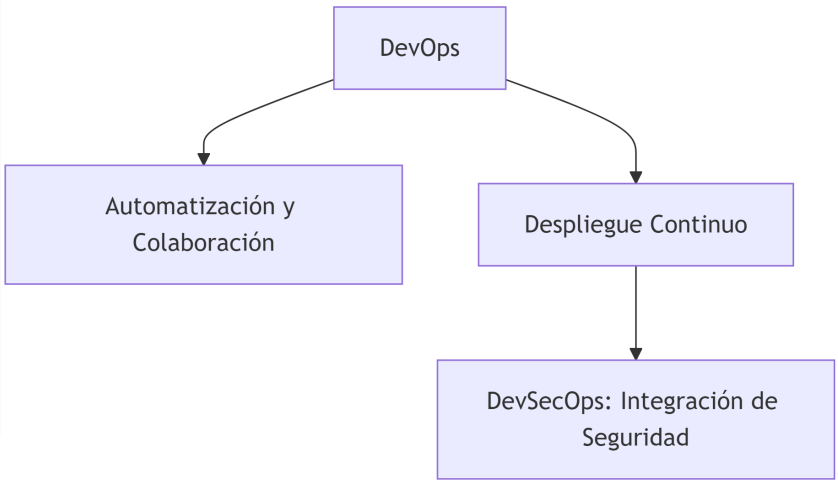
2. **Automation (Automatización)**: Automatiza tareas manuales repetitivas, como las pruebas y el despliegue.
3. **Measurement (Medición)**: Monitorea y mide continuamente el rendimiento del sistema y los equipos.
4. **Sharing (Compartir)**: Fomenta la transparencia y el intercambio de conocimiento entre los miembros del equipo.



1.6 DevOps y DevSecOps

DevSecOps es una evolución de **DevOps** que integra las prácticas de seguridad dentro del ciclo de vida del desarrollo y operaciones. En lugar de agregar la seguridad como un paso final, **DevSecOps** se asegura de que las pruebas de seguridad se realicen continuamente, de manera similar a las pruebas funcionales y de rendimiento.

Esto asegura que los errores de seguridad se detecten lo antes posible, minimizando el impacto y el coste de solucionarlos.

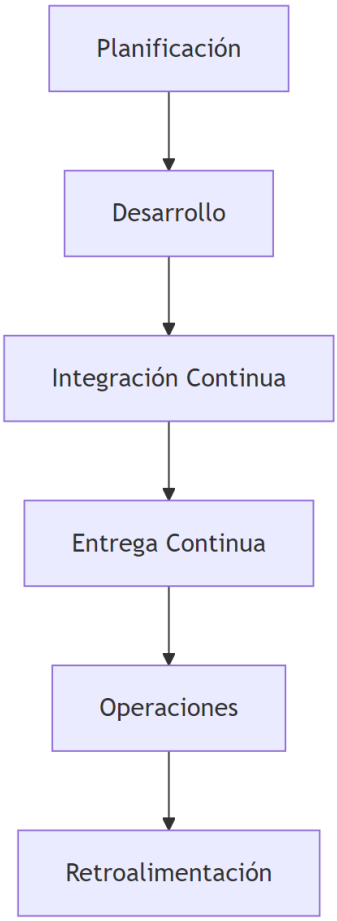


1.7 Ciclo de vida DevOps

El ciclo de vida en **DevOps** es un proceso continuo que abarca las siguientes fases clave:

1. **Planificación**: Definir las características y objetivos del producto.
2. **Desarrollo**: Codificación y control de versiones.
3. **Integración Continua**: Integrar y probar el código de manera continua.
4. **Entrega Continua**: Desplegar el código en entornos de pruebas y producción de manera automática.
5. **Operaciones**: Monitorear el rendimiento de la aplicación y resolver cualquier problema.
6. **Retroalimentación**: Recolectar datos para mejorar el ciclo de desarrollo en futuras iteraciones.

Este ciclo se repite de manera continua, mejorando el proceso con cada iteración.



Con estos principios, el equipo DevOps es capaz de implementar una cultura de mejora continua que aumenta la eficiencia, reduce los tiempos de entrega y asegura la calidad del software.

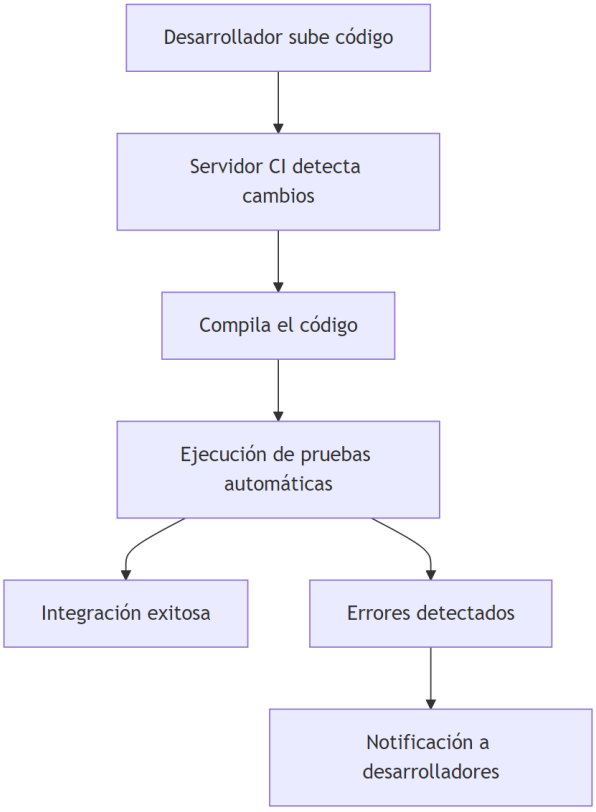
2. Integración Continua / Entrega Continua

2.1 Qué es Integración Continua

La **Integración Continua (CI)** es una práctica de desarrollo en la que los desarrolladores integran su código de manera frecuente en un repositorio compartido, generalmente varias veces al día. Cada integración es verificada automáticamente mediante una **compilación** y pruebas, lo que permite detectar y corregir errores rápidamente. El objetivo principal de CI es reducir los problemas de integración, evitando que los desarrolladores trabajen en aislamiento durante largos periodos y luego enfrenten grandes problemas al fusionar el código.

El flujo de trabajo de CI incluye los siguientes pasos:

1. Los desarrolladores realizan cambios en su código local y lo suben al sistema de control de versiones.
2. Un servidor de CI automáticamente toma los cambios, los compila y ejecuta pruebas automatizadas.
3. Si las pruebas pasan, los cambios se integran al código principal.
4. Si se detectan errores, los desarrolladores son notificados para corregirlos de inmediato.

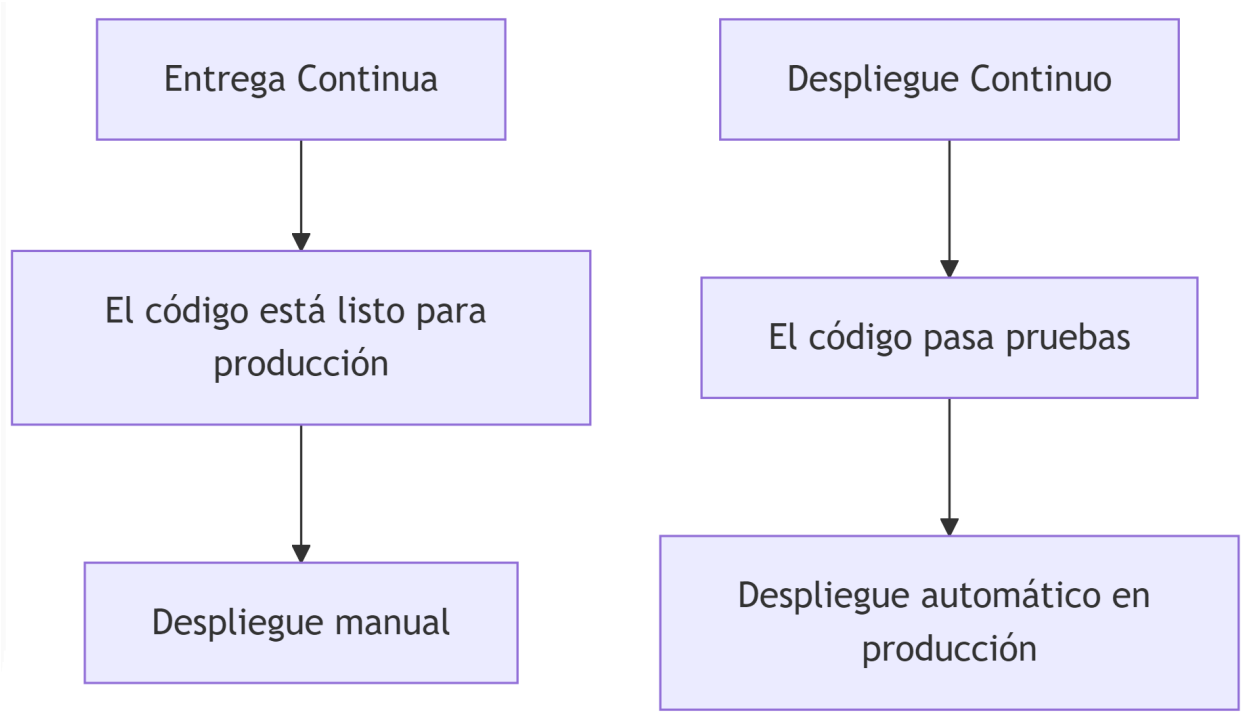


2.2 Despliegue Continuo vs Entrega Continua

En **Despliegue Continuo** y **Entrega Continua**, ambos conceptos están relacionados con la automatización y la entrega de software, pero tienen diferencias importantes:

- **Entrega Continua (Continuous Delivery)**: El objetivo de la entrega continua es tener el código listo para ser desplegado en producción en cualquier momento. Esto significa que el código pasa por todo el ciclo de pruebas y validación, pero la decisión de cuándo desplegarlo en producción es manual.

- **Despliegue Continuo (Continuous Deployment)**: En el despliegue continuo, cualquier cambio que pase todas las fases de pruebas y validación se despliega automáticamente en producción sin intervención manual. Es una extensión de la entrega continua con un paso automatizado adicional: el despliegue.



Ejemplo:

- **Entrega Continua**: Un equipo de desarrollo tiene una nueva versión del producto que ha pasado todas las pruebas, pero el equipo decide cuándo implementarla en producción.
- **Despliegue Continuo**: En cuanto la nueva versión pasa todas las pruebas, se despliega automáticamente en producción sin intervención humana.

2.3 Flujo del proceso de Integración Continua

El flujo de trabajo típico de la **Integración Continua** consiste en varias etapas automatizadas que garantizan que cada cambio de código se integre de forma eficiente y sin errores.

2.3.1 Control de versiones

El proceso de CI comienza con el uso de un **Sistema de Control de Versiones (SCV)**, como **Git**, que permite a los desarrolladores mantener un registro de todos los cambios realizados en el código. Cada vez que un desarrollador realiza un commit o push de su código al repositorio, se dispara el proceso de CI.

Ejemplo de comandos Git:

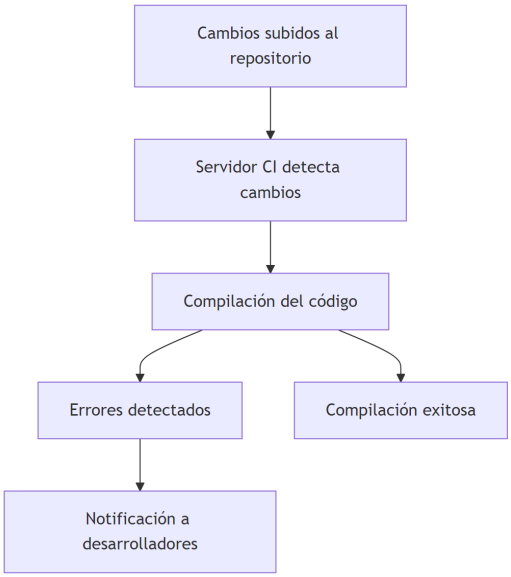
```
# Añadir cambios al repositorio local
git add .

# Realizar un commit de los cambios
git commit -m "Añadir nueva funcionalidad"

# Enviar los cambios al repositorio remoto
git push origin main
```

2.3.2 Compilación

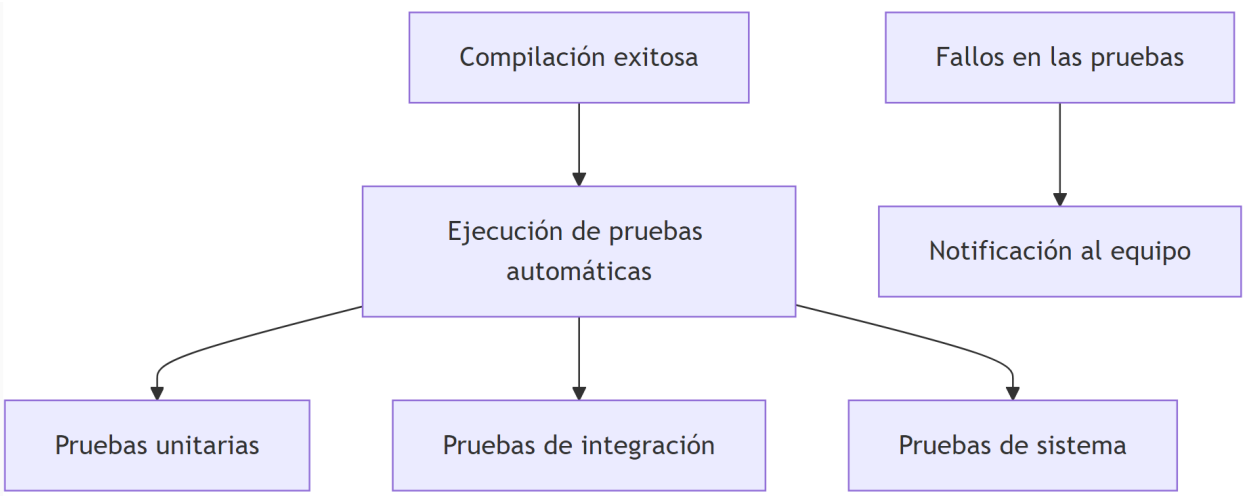
Una vez que los cambios son subidos al repositorio, el **servidor de CI** realiza una **compilación** del código. Este proceso convierte el código fuente en un formato ejecutable, validando que no existan errores de compilación.



2.3.3 Testing

Después de la compilación, se ejecutan **pruebas automáticas** para garantizar que los cambios no introduzcan errores. Las pruebas más comunes incluyen **pruebas unitarias**, **pruebas de integración** y **pruebas de sistema**.

- **Pruebas unitarias**: Verifican que cada unidad de código funcione correctamente.
- **Pruebas de integración**: Aseguran que diferentes componentes del sistema funcionen bien juntos.
- **Pruebas de sistema**: Verifican que el sistema completo se comporte como se espera.



2.4 Sistemas de integración continua

Existen varias herramientas de **Integración Continua (CI)** que permiten automatizar todo el flujo, desde el control de versiones hasta las pruebas y el despliegue. Algunas de las más utilizadas son:

2.4.1 Alternativas

Jenkins

Es una de las herramientas de CI más populares y de código abierto. **Jenkins** permite automatizar todo el proceso de integración y entrega continua. Se puede integrar con una gran variedad de sistemas de control de versiones y proporciona soporte para crear pipelines complejos.

Ejemplo de archivo JenkinsFile:

```
pipeline {
    agent any
    stages {
```

```
        stage('Compilar') {
            steps {
                echo 'Compilando el proyecto...'

                // Comando de compilación
                sh 'make build'
            }
        }

        stage('Pruebas') {
            steps {
                echo 'Ejecutando pruebas...'

                // Comando para ejecutar pruebas
                sh 'make test'
            }
        }
    }
}
```

CircleCI

CircleCI es una plataforma de CI/CD en la nube que permite a los desarrolladores automatizar pruebas y despliegues fácilmente. Su configuración es a través de un archivo `.circleci/config.yml`.

Ejemplo de archivo de configuración CircleCI:

```
version: 2.1

jobs:
  build:
    docker:
      - image: circleci/node:12

    steps:
      - checkout
      - run:
```

```
          name: Instalar dependencias
          command: npm install
      - run:
          name: Ejecutar pruebas
          command: npm test
```

GitLab CI

GitLab CI es una herramienta de CI/CD integrada directamente en GitLab. Permite a los desarrolladores configurar pipelines de CI/CD usando un archivo `.gitlab-ci.yml`.

Ejemplo de archivo GitLab CI:

```
stages:
  - build
  - test
  - deploy

build:
  script:
    - npm install
    - npm run build

test:
  script:
    - npm test

deploy:

  script:
    - npm run deploy
```

Bamboo

Bamboo, de Atlassian, es una herramienta de CI/CD que permite crear pipelines automatizados y gestionar la entrega continua. Se integra fácilmente con otras herramientas de Atlassian como **Jira** y **Bitbucket**.

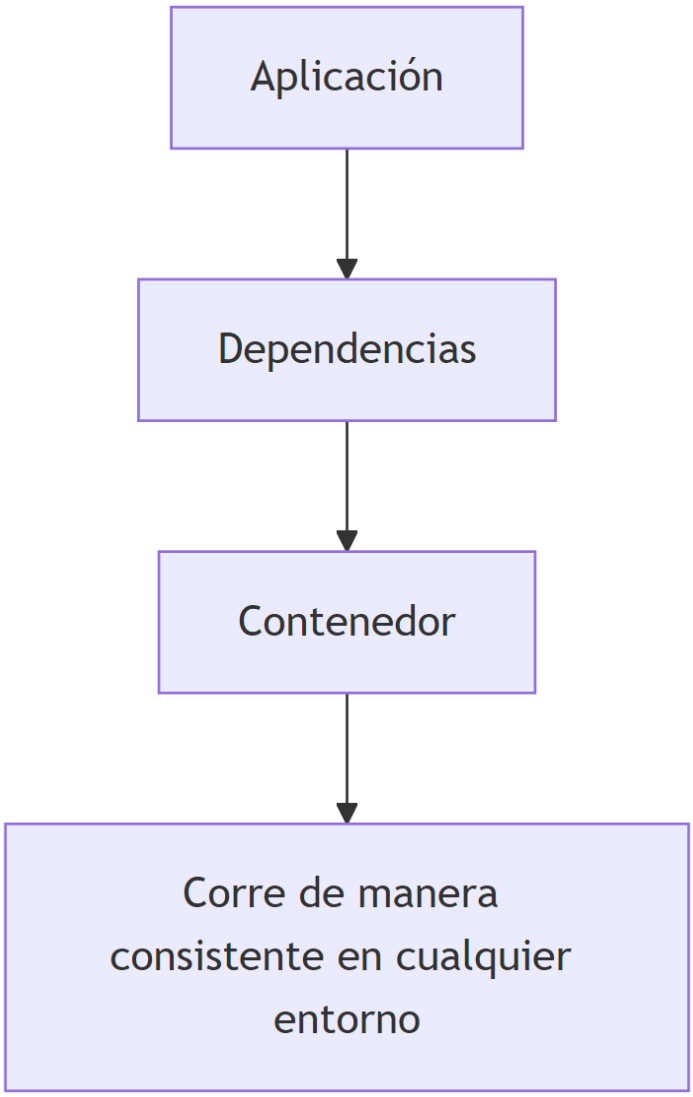
Cada una de estas herramientas ofrece diferentes niveles de integración y personalización, pero todas comparten el objetivo común de automatizar el proceso de integración y despliegue continuo para mejorar la calidad del software y acelerar los ciclos de desarrollo.

3. Contenedores de Aplicaciones

3.1 Qué es un contenedor de aplicaciones

Un **contenedor de aplicaciones** es una unidad estandarizada que empaqueta todo el código de una aplicación y sus dependencias, de forma que se pueda ejecutar de manera consistente en cualquier entorno. A diferencia de los entornos tradicionales, donde las aplicaciones dependen del sistema operativo subyacente, los contenedores incluyen todos los binarios, bibliotecas y configuraciones necesarios para que la aplicación funcione.

Los contenedores son ligeros y rápidos de implementar, ya que comparten el núcleo del sistema operativo subyacente, en lugar de virtualizar todo un sistema operativo como lo hacen las **máquinas virtuales**.

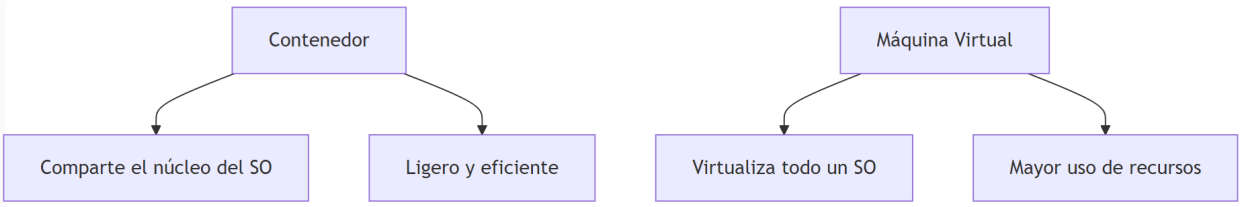


3.2 Diferencias entre un contenedor y una máquina virtual

Aunque los **contenedores** y las **máquinas virtuales (VM)** parecen similares, tienen diferencias clave:

- **Máquinas Virtuales (VMs)**: Virtualizan todo un sistema operativo. Cada VM incluye su propio sistema operativo invitado, lo que requiere más recursos y espacio.
- **Contenedores**: Utilizan el núcleo del sistema operativo del anfitrión y solo contienen los binarios y librerías necesarios para la aplicación. Esto los hace más ligeros y eficientes en términos de rendimiento y uso de recursos.

Característica	Contenedores	Máquinas Virtuales
Aislamiento	Aislamiento a nivel de proceso	Aislamiento a nivel de sistema operativo
Tamaño	Ligero (MBs)	Pesado (GBs)
Velocidad de inicio	Rápida (segundos)	Lenta (minutos)
Uso de recursos	Menos recursos (comparte el kernel)	Más recursos (cada VM tiene su SO)



3.3 El contenedor Docker - Conceptos básicos de Docker

Docker es una plataforma que permite crear, implementar y ejecutar aplicaciones en contenedores. Los contenedores Docker encapsulan el código y las dependencias en un entorno aislado y portátil, lo que garantiza que se ejecuten de manera uniforme en diferentes entornos de desarrollo, prueba y producción.

3.3.1 Images

Una **imagen Docker** es una plantilla de solo lectura que contiene todo lo necesario para ejecutar una aplicación: código, dependencias, configuraciones, etc. Es el blueprint a partir del cual se crean los contenedores.

Ejemplo de comandos Docker para listar imágenes:

```
# Listar imágenes existentes
docker images
```

3.3.2 DockerFile

El **Dockerfile** es un archivo de texto que contiene un conjunto de instrucciones para crear una imagen Docker. Define cómo se construye la imagen, incluyendo el sistema base, dependencias y configuraciones.

Ejemplo de un Dockerfile simple:

```
# Usar una imagen base de Node.js
FROM node:14

# Establecer el directorio de trabajo
WORKDIR /app

# Copiar el código al contenedor
COPY . .

# Instalar las dependencias
RUN npm install

# Exponer el puerto de la aplicación
EXPOSE 3000

# Comando para ejecutar la aplicación
CMD ["npm", "start"]
```

3.3.3 Containers

Un **contenedor** es una instancia en ejecución de una imagen Docker. Cuando ejecutas un contenedor, Docker utiliza la imagen como plantilla y crea un entorno aislado donde se ejecuta la aplicación.

Ejemplo de cómo correr un contenedor:

```
# Correr un contenedor a partir de una imagen
docker run -d -p 3000:3000 nombre_imagen
```

3.3.4 Volumes

Los **volúmenes** en Docker permiten que los contenedores almacenen datos de manera persistente. Mientras que los datos dentro de un contenedor desaparecen cuando el contenedor se detiene, los volúmenes permiten que los datos persistan entre las ejecuciones.

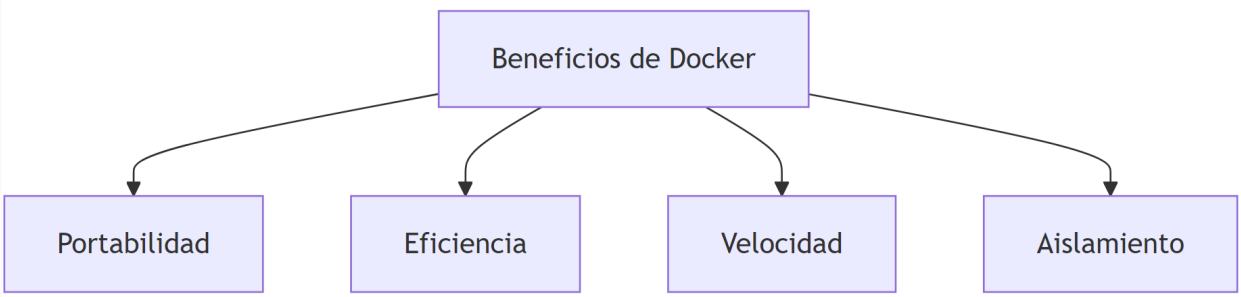
Ejemplo de cómo crear y usar un volumen:

```
# Crear un volumen
docker volume create my_volume

# Ejecutar un contenedor con un volumen
docker run -d -v my_volume:/data nombre_imagen
```

3.4 Beneficios de utilizar un contenedor Docker

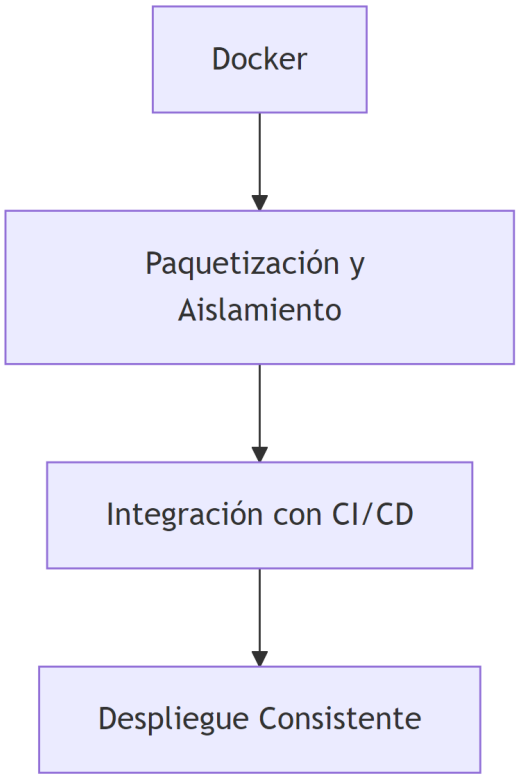
- **Portabilidad**: Los contenedores Docker permiten ejecutar aplicaciones de manera consistente en diferentes entornos, desde la máquina del desarrollador hasta los servidores de producción.
- **Eficiencia**: Los contenedores son ligeros y utilizan menos recursos que las máquinas virtuales, ya que no virtualizan todo el sistema operativo.
- **Velocidad**: Los contenedores se inician en segundos, lo que mejora la eficiencia en los flujos de trabajo de CI/CD.
- **Aislamiento**: Cada contenedor está aislado, lo que permite ejecutar múltiples aplicaciones en la misma máquina sin conflictos entre ellas.



3.5 Rol del contenedor Docker dentro del ciclo DevOps

En el ciclo **DevOps**, Docker juega un rol fundamental al permitir que las aplicaciones se empaqueten junto con todas sus dependencias, lo que garantiza que se ejecuten de manera idéntica en diferentes entornos (desarrollo, pruebas y producción).

Al integrar Docker con herramientas de **CI/CD**, como **Jenkins** o **GitLab CI**, los desarrolladores pueden crear pipelines automatizados que construyen imágenes Docker, ejecutan pruebas y despliegan contenedores automáticamente en servidores de producción o en entornos de prueba.



3.6 Implementación de Docker

3.6.1 Instalación y configuración

Para instalar Docker en un sistema Linux, puedes ejecutar los siguientes comandos:

```
# Actualizar los paquetes existentes
sudo apt update

# Instalar los paquetes necesarios
sudo apt install apt-transport-https ca-certificates curl
software-properties-common

# Añadir la clave GPG de Docker
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add
-

# Añadir el repositorio de Docker
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

# Instalar Docker
sudo apt install docker-ce

# Verificar la instalación de Docker
docker --version
```

3.6.2 Comandos administrativos básicos

Algunos de los comandos Docker más útiles incluyen:

```
# Listar todos los contenedores en ejecución
docker ps

# Detener un contenedor
docker stop <container_id>

# Eliminar un contenedor
docker rm <container_id>

# Ver detalles de una imagen
docker inspect <image_name>
```

3.6.3 Utilización de imágenes

Docker permite utilizar imágenes predefinidas o crear las tuyas propias. Para buscar imágenes en Docker Hub:

```
# Buscar imágenes en Docker Hub
docker search nginx

# Descargar una imagen desde Docker Hub
docker pull nginx
```

3.6.4 Monitoreo de eventos y logs asociados a contenedores

Docker proporciona herramientas para monitorear los eventos y logs generados por los contenedores. Esto es crucial para el monitoreo continuo en un entorno de producción.

```
# Ver los logs de un contenedor
docker logs <container_id>

# Ver eventos de Docker en tiempo real
docker events
```

Con estos conceptos, los contenedores Docker permiten crear aplicaciones portátiles y consistentes, garantizando que el software funcione de manera idéntica en todos los entornos, uniendo perfectamente el desarrollo y las operaciones en un ciclo de **DevOps**.

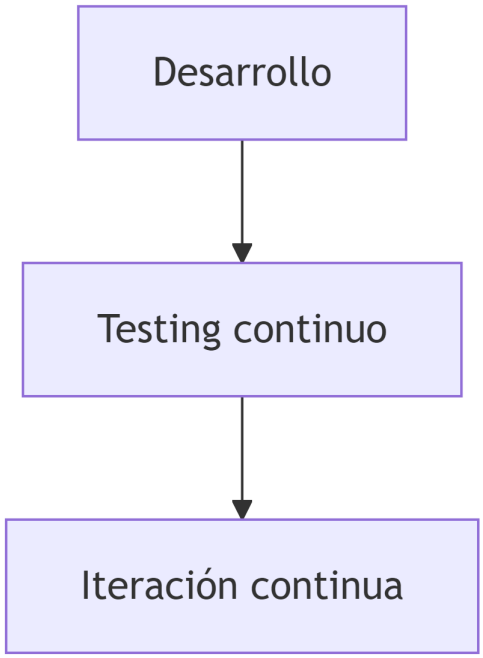
4. Las Pruebas en un Entorno de Integración Continua

4.1 Testing en un entorno ágil

4.1.1 Qué es Agile Testing

Agile Testing es un enfoque de pruebas que sigue los principios de las **metodologías ágiles**, donde el testing se integra desde las primeras etapas del desarrollo y continúa de manera

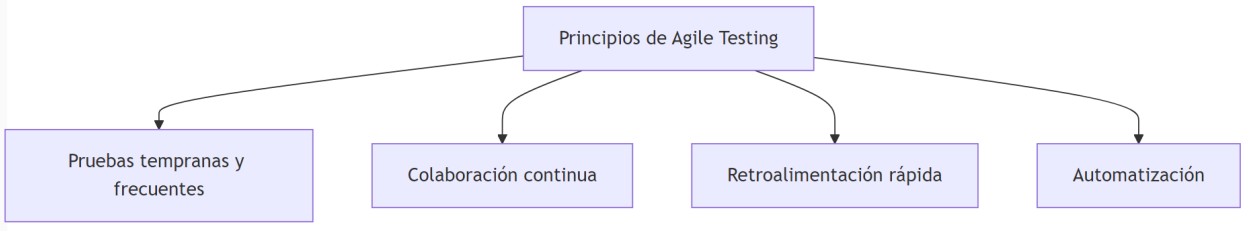
iterativa a lo largo del ciclo de vida del proyecto. A diferencia de los enfoques tradicionales, en Agile Testing las pruebas no son una fase separada que ocurre al final del desarrollo, sino que están integradas de manera continua y colaborativa con los desarrolladores.



4.1.2 Principios de Agile Testing

Los principios clave de **Agile Testing** son:

1. **Pruebas tempranas y frecuentes**: Se realizan pruebas en cada iteración, desde el comienzo del desarrollo, para detectar errores lo antes posible.
2. **Colaboración continua**: Los testers, desarrolladores y stakeholders trabajan de manera conjunta para garantizar la calidad del producto.
3. **Retroalimentación rápida**: Las pruebas ofrecen retroalimentación inmediata sobre los cambios de código.
4. **Automatización**: El uso de pruebas automatizadas es fundamental para mejorar la eficiencia y permitir ciclos de desarrollo más rápidos.



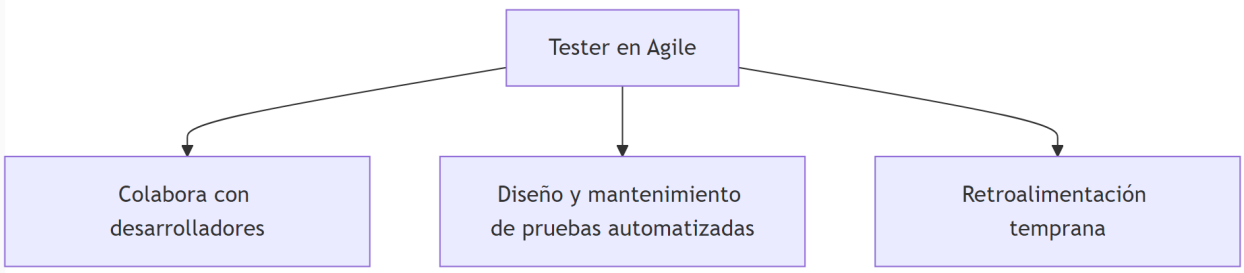
4.1.3 Prácticas relacionadas con Agile Testing

Algunas de las principales prácticas asociadas con **Agile Testing** son:

- **Desarrollo guiado por pruebas (TDD)**: Una metodología en la que los desarrolladores escriben primero las pruebas antes de escribir el código de la funcionalidad.
- **Pruebas automatizadas**: Se usan para ejecutar regresiones frecuentes y asegurar que los cambios no rompan funcionalidades existentes.
- **Pruebas exploratorias**: Permiten a los testers investigar el sistema sin seguir un plan de pruebas estricto.

4.1.4 Rol del tester en un marco ágil

El rol del **tester en Agile** es mucho más activo y colaborativo que en enfoques tradicionales. Los testers trabajan estrechamente con los desarrolladores y el Product Owner desde el principio, participando en las decisiones de diseño, desarrollo y validación. Su trabajo incluye tanto pruebas manuales como la creación y mantenimiento de pruebas automatizadas.

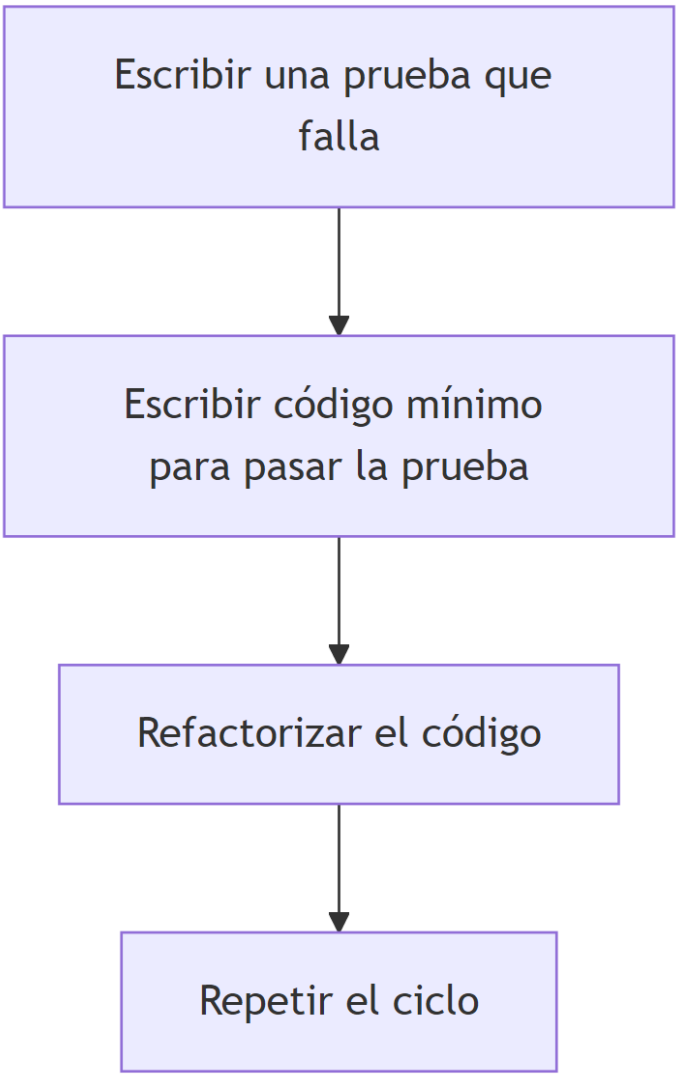


4.1.5 Qué es TDD

Test-Driven Development (TDD), o desarrollo guiado por pruebas, es una práctica de desarrollo donde los desarrolladores escriben primero una prueba que falla y luego implementan el código necesario para que la prueba pase. Este ciclo se repite hasta que se completa la funcionalidad.

El flujo típico de TDD es:

1. **Escribir una prueba** que inicialmente falla (porque el código no existe o no está implementado).
2. **Escribir el código mínimo** necesario para que la prueba pase.
3. **Refactorizar** el código para mejorar su calidad sin romper la prueba.



4.2 Objetivo de las pruebas

El **objetivo principal de las pruebas** en un entorno de integración continua es garantizar la calidad y estabilidad del software mediante la identificación temprana de errores y problemas. Al integrar pruebas automáticas en cada paso del proceso de desarrollo, se asegura que cualquier nuevo cambio en el código no introduzca errores ni afecte negativamente las funcionalidades existentes.

- **Detectar fallos** en el comportamiento del sistema.
- **Asegurar la calidad** antes de integrar el código en la rama principal.
- **Proporcionar retroalimentación rápida** a los desarrolladores.
- **Garantizar la estabilidad del producto**.

4.3 Tipos de prueba

4.3.1 Pruebas unitarias

Las **pruebas unitarias** verifican el comportamiento de unidades individuales de código, como funciones o clases. Son la base de la pirámide de pruebas y ayudan a detectar errores en una fase temprana.

Ejemplo de prueba unitaria en Python

```
def suma(a, b):
    return a + b

def test_suma():
    assert suma(2, 3) == 5
```

4.3.2 Pruebas de integración

Las **pruebas de integración** aseguran que diferentes módulos o componentes de una aplicación funcionen bien juntos. Verifican que las interacciones entre distintas partes del sistema se comporten como se espera.

4.3.3 Pruebas de sistema

Las **pruebas de sistema** validan el comportamiento del sistema completo. Existen varios tipos de pruebas de sistema, entre ellos:

4.3.3.1 Funcionales

Verifican que el sistema cumpla con los requisitos funcionales definidos por los usuarios o stakeholders.

4.3.3.2 Rendimiento

Evalúan cómo se comporta el sistema bajo diferentes cargas de trabajo, asegurando que funcione de manera eficiente bajo estrés.

4.3.4 Pruebas de aceptación

Las **pruebas de aceptación** validan que el software cumpla con los criterios de aceptación definidos por el cliente. Generalmente se realizan antes de que el software sea desplegado en producción.

4.3.5 Pruebas de humo

Las **pruebas de humo** son un conjunto básico de pruebas rápidas que verifican si los aspectos principales del sistema funcionan correctamente después de una nueva compilación.

4.4 Automatización de las pruebas

4.4.1 Objetivo de la automatización

El **objetivo principal de la automatización de pruebas** es acelerar el ciclo de desarrollo al reducir la necesidad de pruebas manuales repetitivas. Al automatizar las pruebas, los equipos pueden ejecutar pruebas frecuentes, lo que permite identificar errores rápidamente y mantener la estabilidad del software.

4.4.2 Herramientas para la automatización de las pruebas

Existen varias herramientas populares para la automatización de pruebas, entre las más utilizadas están:

- **Selenium**: Herramienta para automatizar pruebas de aplicaciones web.
- **JUnit**: Para pruebas unitarias en Java.
- **pytest**: Marco para pruebas automatizadas en Python.
- **Jest**: Utilizado para pruebas en aplicaciones basadas en JavaScript.

4.4.3 Incorporación de la automatización de pruebas al pipeline de integración continua

La **automatización de pruebas** debe integrarse en el **pipeline de CI/CD** para garantizar que cada cambio en el código pase por una serie de pruebas antes de ser integrado en la rama principal y desplegado en producción.

Ejemplo de pipeline en Jenkins con pruebas automatizadas:

```
pipeline {
    agent any

    stages {
        stage('Compilar') {
            steps {
                echo 'Compilando...'
                sh 'npm install'
            }
        }
        stage('Pruebas') {
            steps {
                echo 'Ejecutando pruebas...'
                sh 'npm test'
            }
        }
        stage('Despliegue') {
            steps {
                echo 'Desplegando...'
                sh 'npm run deploy'
            }
        }
    }
}
```

Con la integración de pruebas automatizadas en un entorno de integración continua, los equipos pueden garantizar una entrega de software más rápida, segura y eficiente.

5. Implementación de un Pipeline CI

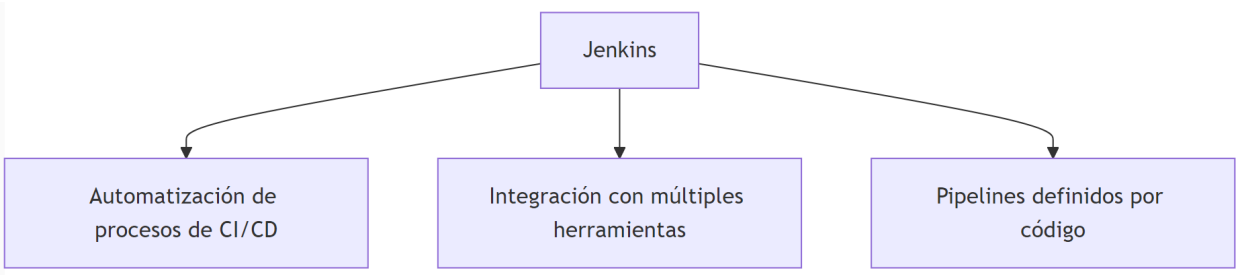
5.1 Qué es Jenkins y para qué sirve

Jenkins es una herramienta de código abierto utilizada para automatizar el proceso de integración continua (CI) y entrega continua (CD). Jenkins facilita la automatización de las tareas relacionadas con la compilación, prueba y despliegue del software, lo que permite que los desarrolladores se centren en la escritura de código en lugar de en las tareas manuales repetitivas.

Jenkins está basado en **pipelines**, que son flujos de trabajo definidos por código y que controlan el ciclo de vida completo del software, desde el desarrollo hasta el despliegue en producción.

Beneficios clave de Jenkins:

- **Automatización de procesos** como la compilación, pruebas y despliegue.
- **Integración con numerosas herramientas** de desarrollo como Git, Docker, AWS, entre otros.
- **Facilidad de configuración de pipelines** mediante archivos de código (JenkinsFile).
- **Monitoreo en tiempo real** del estado de los builds y notificaciones automáticas.

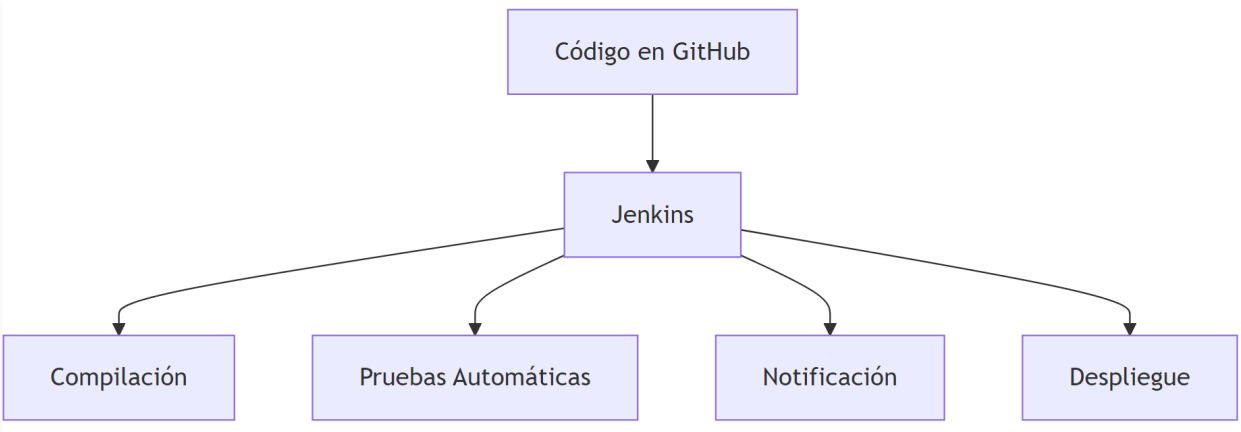


5.2 Rol de Jenkins dentro de la Integración Continua

En el contexto de la **Integración Continua (CI)**, Jenkins actúa como un **servidor de automatización** que coordina los siguientes pasos:

1. **Compilación del código**: Jenkins toma el código desde un repositorio de control de versiones (Git, SVN) y lo compila.
2. **Ejecución de pruebas**: Ejecuta pruebas automáticas para verificar que el código no introduzca errores.

3. **Notificación**: Si el build falla o pasa, Jenkins notifica automáticamente a los desarrolladores mediante correos electrónicos o herramientas de mensajería como Slack.
4. **Despliegue**: Jenkins puede automatizar el despliegue del código en un entorno de pruebas o producción si todas las pruebas son exitosas.



5.3 Instalación y configuración de Jenkins

5.3.1 Instalación de Jenkins

Jenkins puede instalarse fácilmente en sistemas operativos como **Linux**, **Windows**, o **macOS**. Aquí un ejemplo de cómo instalar Jenkins en **Ubuntu**:

```
# Actualizar el sistema
sudo apt update

# Instalar Java (Jenkins requiere Java)
sudo apt install openjdk-11-jdk

# Añadir el repositorio de Jenkins y su clave GPG
wget -q -O - https://pkg.jenkins.io/debian/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'

# Instalar Jenkins
```

```
sudo apt update

sudo apt install jenkins

# Iniciar Jenkins
sudo systemctl start jenkins
```

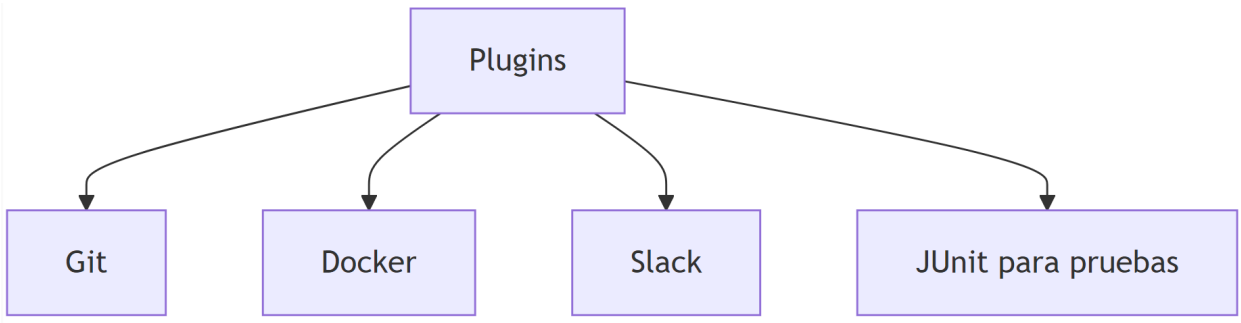
5.3.1 Configuración de plugins, usuarios, roles y permisos

Una vez instalado, Jenkins se configura a través de su interfaz web. Para acceder a Jenkins, abre un navegador y dirígete a: `http://localhost:8080`. El primer paso es configurar **usuarios y roles**:

- **Usuarios**: Jenkins permite gestionar múltiples usuarios, lo que facilita la asignación de permisos y responsabilidades a diferentes miembros del equipo.
- **Roles**: Utiliza roles para definir permisos específicos (como acceso a configuraciones, ejecución de builds, etc.) en función de los usuarios o equipos.

Configuración de plugins: Jenkins se integra con diversas herramientas mediante plugins. Por ejemplo:

- **Git**: Para trabajar con repositorios Git.
- **Docker**: Para ejecutar contenedores Docker.
- **Slack**: Para notificaciones.



5.4 Definición de pipelines de integración con código: JenkinsFile

En Jenkins, los pipelines se definen mediante un archivo llamado **JenkinsFile**, que contiene las instrucciones para realizar las distintas etapas del pipeline, como compilación, pruebas y despliegue.

Ejemplo de un **JenkinsFile** básico para una aplicación Node.js:

```
pipeline {
    agent any

    stages {
        stage('Clonar Repositorio') {
            steps {
                git 'https://github.com/usuario/repositorio.git'
            }
        }
        stage('Instalar Dependencias') {
            steps {
                sh 'npm install'
            }
        }
        stage('Ejecutar Pruebas') {
            steps {
                sh 'npm test'
            }
        }
        stage('Compilar Proyecto') {
            steps {
                sh 'npm run build'
            }
        }
        stage('Desplegar') {
            steps {
                sh 'npm run deploy'
            }
        }
    }
```

```
}
```

Este pipeline realiza las siguientes tareas:

1. Clona el repositorio.
2. Instala las dependencias del proyecto.
3. Ejecuta las pruebas unitarias.
4. Compila el código.
5. Despliega la aplicación en un entorno de pruebas o producción.

5.5 Utilización de Jenkins en el ciclo CI/CD

Jenkins es una pieza clave en la automatización del ciclo CI/CD, permitiendo que los desarrolladores integren y desplieguen cambios de manera continua.

5.5.1 Creación y configuración de un job en Jenkins

Los **jobs** en Jenkins son unidades que definen un conjunto de tareas automatizadas. Para crear un job en Jenkins:

1. Inicia sesión en Jenkins.
2. Haz clic en **"Nuevo Item"**.
3. Selecciona **"Pipeline"** y nombra el job.
4. Configura las etapas del pipeline, como la compilación, ejecución de pruebas, etc.
5. Guarda y ejecuta el job.

5.5.2 Creación y configuración de pipeline de Jenkins mediante código JenkinsFile

Al usar un **JenkinsFile**, defines un pipeline directamente en el código, lo que permite que el pipeline sea versionado junto con el código fuente.

Pasos para configurar un pipeline:

1. Crear un **JenkinsFile** en el repositorio de tu proyecto.
2. En Jenkins, crea un nuevo **job** y selecciona "Pipeline".
3. Configura Jenkins para que lea el **JenkinsFile** desde el repositorio Git.
4. Guarda y ejecuta el job.

5.5.3 Creación y configuración de despliegue sobre el ambiente de pruebas

Jenkins permite configurar pipelines para realizar despliegues automáticos en entornos de pruebas o producción. El despliegue puede implicar tareas como:

- **Construcción de contenedores Docker**.
- **Despliegue en servidores remotos** usando **SSH**.
- **Despliegue en la nube** con servicios como **AWS** o **Azure**.

Ejemplo de un pipeline que usa **Docker** para el despliegue:

```
pipeline {
    agent any

    stages {
        stage('Construir imagen Docker') {
            steps {
                script {
                    docker.build('mi-app:latest')
                }
            }
        }
        stage('Desplegar en Producción') {
            steps {
                script {
                    docker.image('mi-app:latest').run('-d -p 80:80')
                }
            }
        }
    }
}
```

Este pipeline construye una imagen Docker para la aplicación y la despliega en un servidor de producción.

Con Jenkins, es posible definir pipelines complejos que automaticen todo el proceso de CI/CD, desde la compilación hasta el despliegue, lo que garantiza una entrega rápida, segura y eficiente del software.

6. Infraestructura y Operaciones

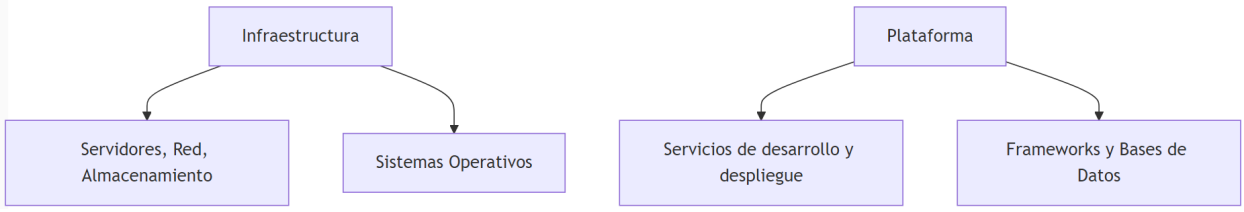
6.1 Qué se entiende por infraestructura y por plataforma

En el contexto de TI y **DevOps**, la **infraestructura** se refiere a los recursos físicos y virtuales necesarios para ejecutar aplicaciones y servicios, como servidores, redes, almacenamiento, y sistemas operativos.

Por otro lado, una **plataforma** es un conjunto de herramientas y servicios que permiten el desarrollo, implementación y administración de aplicaciones. Las plataformas pueden incluir bases de datos, servicios de almacenamiento, middleware y frameworks.

Ejemplo:

- **Infraestructura**: Un conjunto de servidores, switches de red y discos duros.
- **Plataforma**: Un servicio en la nube como **AWS** que ofrece infraestructura bajo demanda junto con servicios como bases de datos y almacenamiento.



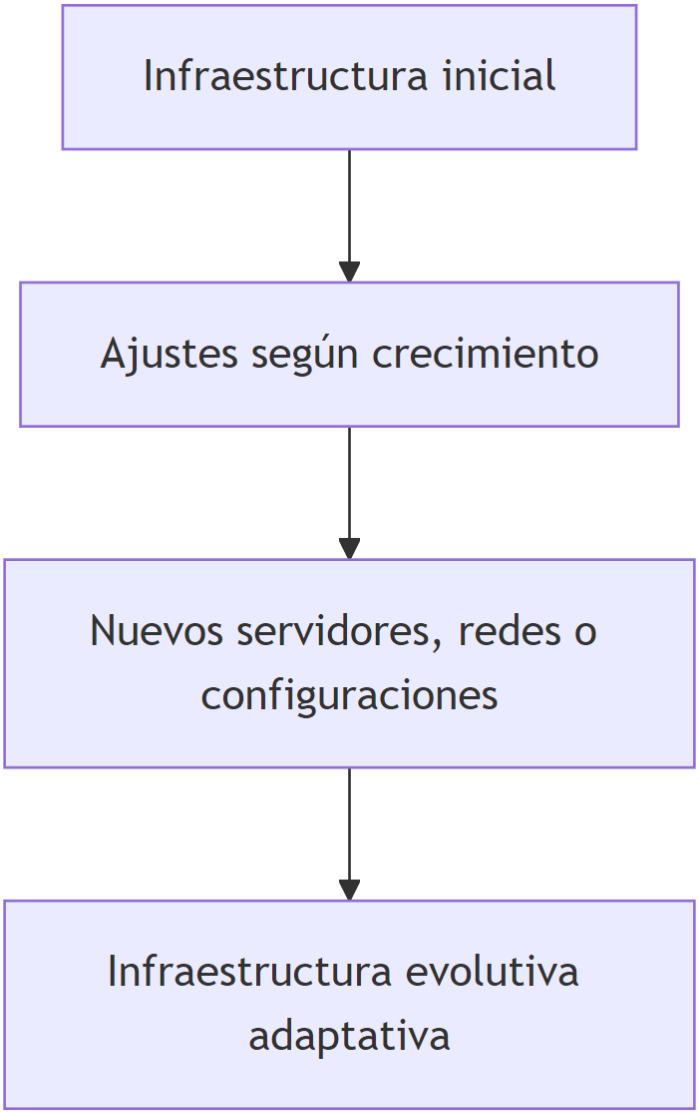
6.2 Infraestructura evolutiva

La **infraestructura evolutiva** es un enfoque en el cual la infraestructura crece y cambia a lo largo del tiempo para adaptarse a las necesidades del negocio y las aplicaciones. En lugar de crear toda la infraestructura desde el inicio de un proyecto, se ajusta y mejora continuamente para soportar nuevas funcionalidades y requisitos.

La infraestructura evolutiva es especialmente relevante en el ciclo **DevOps**, donde los cambios constantes en el desarrollo y despliegue requieren una infraestructura adaptable.

Ejemplo:

- Escalar un clúster de servidores para manejar una carga de trabajo creciente en una aplicación de comercio electrónico.

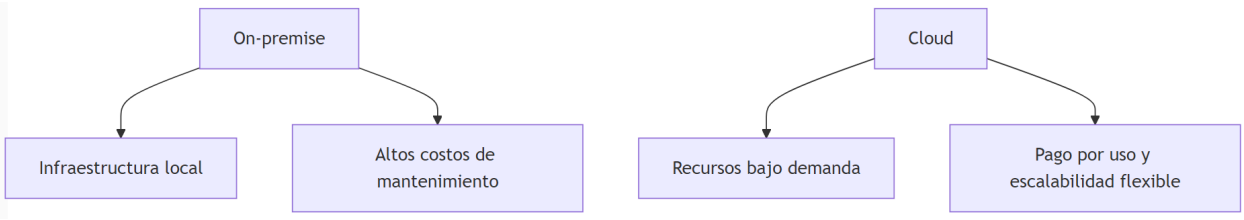


6.3 Diferencias entre on-premise y cloud

- **On-premise**: Los recursos y servidores están ubicados físicamente en las instalaciones de la organización. Esto implica que la empresa gestiona completamente la infraestructura, incluyendo el mantenimiento y seguridad.

- **Cloud**: La infraestructura se aloja en la nube y es proporcionada por terceros (AWS, Azure, Google Cloud). Esto permite un acceso bajo demanda a recursos escalables, lo que reduce los costos iniciales de infraestructura y el mantenimiento.

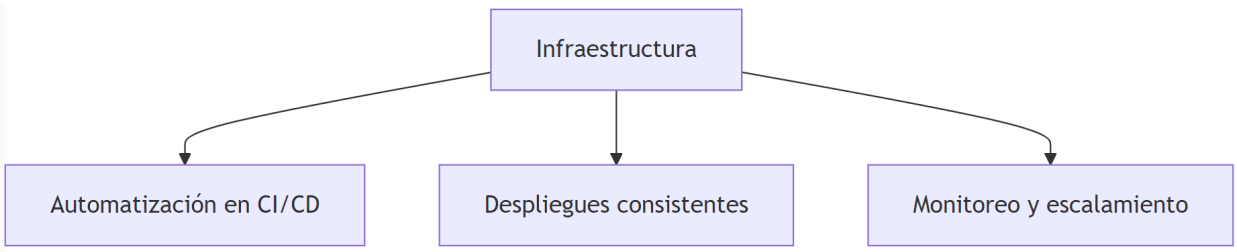
Característica	On-premise	Cloud
Ubicación	En las instalaciones del cliente	Proveedor externo (remoto)
Costos	Altos costos iniciales y mantenimiento	Pago por uso (escalable)
Escalabilidad	Limitada a la infraestructura física	Escalabilidad bajo demanda



6.4 Importancia de la infraestructura en el ciclo DevOps

La infraestructura es crítica en el ciclo **DevOps**, ya que permite la ejecución eficiente de aplicaciones en entornos consistentes. Una infraestructura bien diseñada garantiza que las aplicaciones puedan ser desplegadas, monitoreadas y escaladas automáticamente, lo que es esencial para un flujo de trabajo continuo de desarrollo y operaciones.

Una infraestructura sólida y automatizada facilita la implementación de **CI/CD** (Integración Continua/Entrega Continua) al garantizar que las pruebas y los despliegues se realicen en entornos consistentes.



6.5 Infraestructura como código

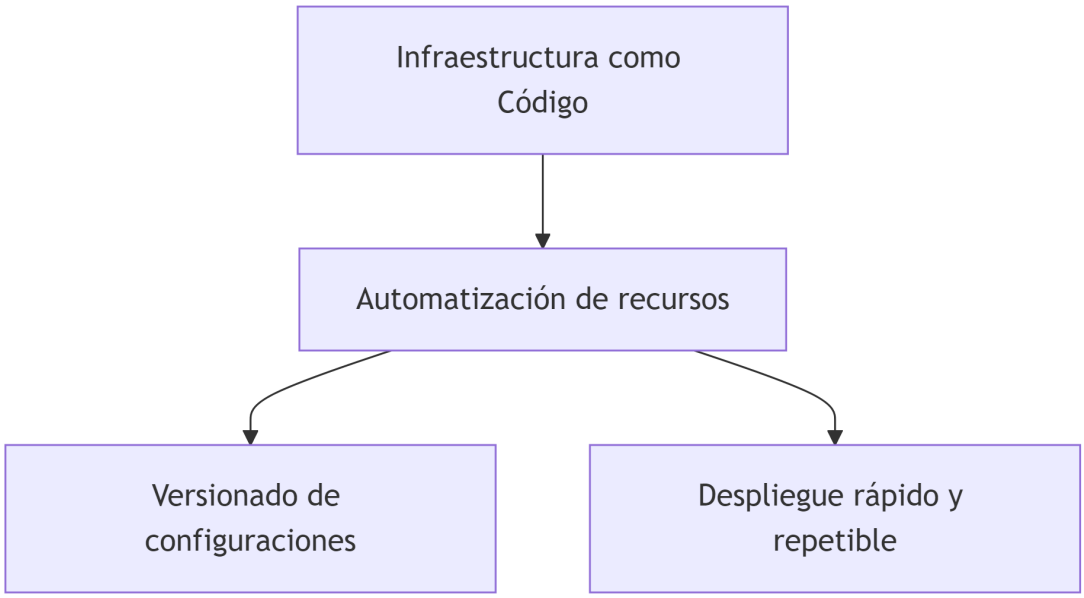
Infraestructura como Código (IaC) es la práctica de gestionar y aprovisionar la infraestructura mediante código en lugar de configuraciones manuales. Utilizando herramientas como **Terraform** o **Ansible**, se puede definir la infraestructura (servidores, redes, almacenamiento) en archivos de configuración que son versionados y aplicados automáticamente.

Ejemplo:

Un archivo de configuración de Terraform que define una máquina virtual en **AWS**:

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

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



6.6 El modelo de infraestructura cloud

El **modelo de infraestructura cloud** permite que las organizaciones accedan a recursos computacionales bajo demanda, gestionados por un proveedor externo. Los recursos pueden ser escalados según la demanda y se paga solo por lo que se utiliza, reduciendo los costos iniciales y mejorando la flexibilidad.

Las nubes públicas como **AWS**, **Azure** o **Google Cloud** ofrecen modelos que permiten a las organizaciones enfocarse más en su software y menos en la administración de la infraestructura física.

6.7 Modelos de servicio en la nube (IaaS, PaaS, SaaS)

Existen tres modelos principales de servicios en la nube:

1. **IaaS (Infrastructure as a Service)**: Ofrece recursos de infraestructura como servidores, redes y almacenamiento. El cliente gestiona el sistema operativo y las aplicaciones. Ejemplo: **AWS EC2**.
2. **PaaS (Platform as a Service)**: Proporciona una plataforma que incluye un entorno para desarrollar, ejecutar y gestionar aplicaciones sin tener que preocuparse por la infraestructura subyacente. Ejemplo: **Heroku**.

3. **SaaS (Software as a Service)**: El cliente utiliza aplicaciones a través de la web que están completamente gestionadas por el proveedor. Ejemplo: **Google Workspace**, **Salesforce**.

6.8 Operaciones y escalamiento

6.8.1 Problemas de la operación

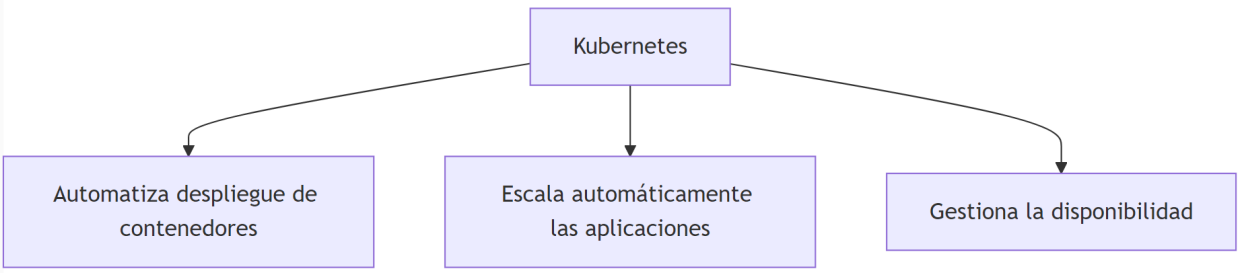
En las operaciones diarias, los problemas comunes incluyen la falta de escalabilidad, configuraciones inconsistentes entre entornos y dificultades para gestionar grandes volúmenes de tráfico o datos. DevOps busca resolver estos problemas a través de la automatización y la infraestructura como código.

6.8.2 Orquestadores de contenedores

Los **orquestadores de contenedores** permiten gestionar de manera eficiente grandes cantidades de contenedores en producción, asegurando que se escalen y distribuyan de forma automática.

6.8.2 Qué es Kubernetes

Kubernetes es una plataforma de orquestación de contenedores de código abierto que automatiza la implementación, escalado y gestión de aplicaciones en contenedores. Facilita la distribución automática de cargas de trabajo entre contenedores y asegura que las aplicaciones se mantengan disponibles y escalables.



6.9 Monitoreo

6.9.1 Qué es el monitoreo continuo

El **monitoreo continuo** es el proceso de supervisar constantemente el rendimiento y el estado de la infraestructura y las aplicaciones en producción. Esto permite detectar problemas antes de que afecten a los usuarios finales y garantiza que los sistemas funcionen de manera eficiente.

6.9.2 Importancia del monitoreo continuo durante el ciclo DevOps

El monitoreo continuo es esencial en el ciclo **DevOps**, ya que proporciona retroalimentación inmediata sobre el rendimiento y estado del sistema después de cada despliegue. Esto asegura que los cambios en el código no introduzcan errores o reduzcan la eficiencia del sistema.

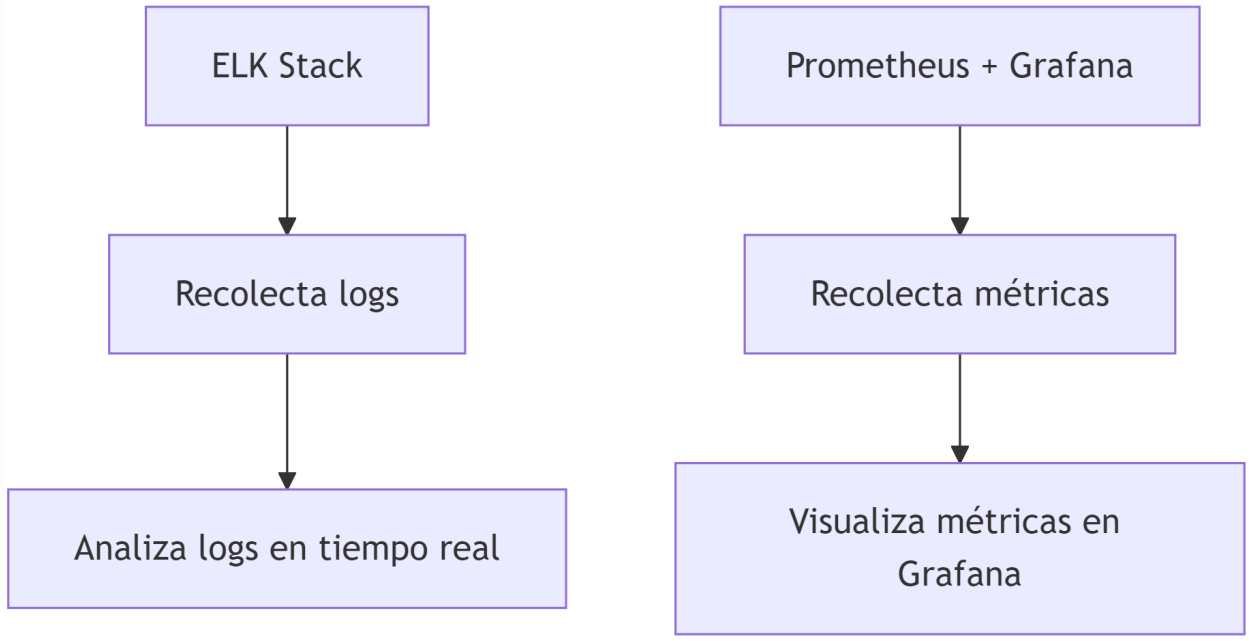
6.9.3 Importancia del manejo de logs y eventos

Los **logs** y **eventos** son registros clave que permiten diagnosticar problemas en los sistemas y aplicaciones. La gestión adecuada de logs ayuda a identificar patrones de errores y a corregir problemas recurrentes.

6.9.4 Stacks de monitoreo (ELK, GFG)

Existen varias soluciones de monitoreo continuo. Entre las más comunes están:

- **ELK Stack (Elasticsearch, Logstash, Kibana)**: Un conjunto de herramientas que permite la recolección, análisis y visualización de logs en tiempo real.
- **Prometheus + Grafana (GFG)**: Prometheus es una herramienta de monitoreo que recolecta métricas de sistemas, mientras que Grafana visualiza esas métricas en tiempo real.



La infraestructura y operaciones juegan un papel fundamental en el ciclo de **DevOps**, asegurando que las aplicaciones sean desplegadas de manera eficiente, escalable y monitoreada continuamente.

7. Sistemas de Control de Versiones (SCV)

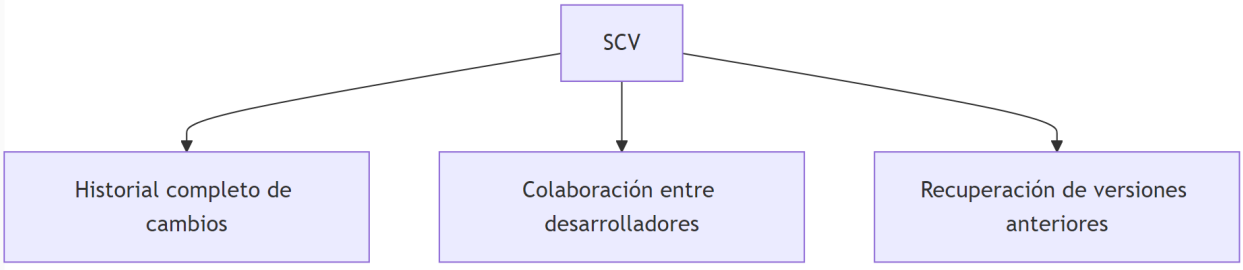
7.1 Qué es un SCV

Un **Sistema de Control de Versiones (SCV)** es una herramienta que permite gestionar y realizar un seguimiento de los cambios en el código fuente a lo largo del tiempo. Un SCV guarda un historial completo de todas las modificaciones realizadas en los archivos, lo que facilita la colaboración entre varios desarrolladores y permite recuperar versiones anteriores del código si es necesario.

Ejemplo:

Un equipo de desarrollo que trabaja en un proyecto puede utilizar un SCV para mantener un registro detallado de los cambios en el código, permitiendo que cada miembro del equipo

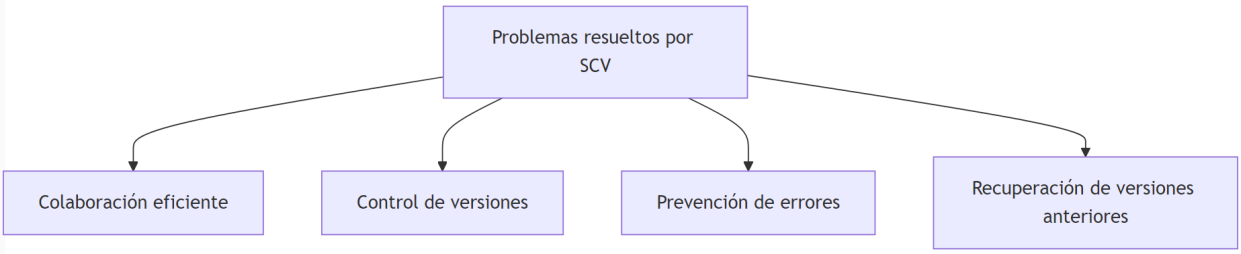
pueda trabajar en paralelo sin conflictos y pueda volver a versiones anteriores si se introduce un error.



7.2 Problema que resuelve un SCV

Un SCV resuelve varios problemas en el desarrollo de software:

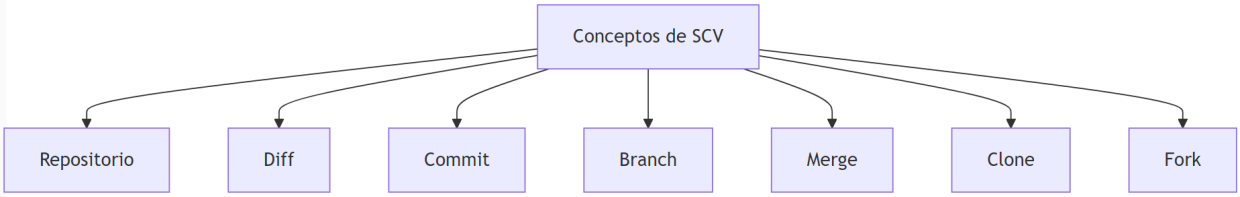
1. **Colaboración eficiente**: Permite que múltiples desarrolladores trabajen simultáneamente en un proyecto sin sobrescribir el trabajo de otros.
2. **Control de versiones**: Facilita el seguimiento de los cambios en el código, proporcionando un historial de modificaciones que puede ser revertido en cualquier momento.
3. **Prevención de errores**: Permite comparar cambios antes de fusionarlos con el código principal, reduciendo el riesgo de introducir errores.
4. **Recuperación de versiones anteriores**: Los desarrolladores pueden regresar a una versión anterior del código si algo falla en la versión actual.



7.3 Principales conceptos de un SCV (Repositorio, Diff, Commit, Branch, Merge, Clone, Fork)

- **Repositorio**: El lugar donde se almacena todo el historial del proyecto, incluidos los archivos y los cambios.
- **Diff**: La comparación de cambios entre dos versiones de un archivo.

- **Commit**: Un conjunto de cambios que se guarda en el historial del repositorio.
- **Branch**: Una rama separada del código principal que permite trabajar en paralelo sin afectar la versión principal.
- **Merge**: El proceso de combinar los cambios de diferentes ramas en una sola.
- **Clone**: Crear una copia local de un repositorio remoto.
- **Fork**: Crear una copia de un repositorio para hacer cambios sin afectar el repositorio original.



Ejemplo:

```
# Crear un commit en Git
git add archivo.txt
git commit -m "Añadir nueva funcionalidad"
```

7.4 Tipos de SCV y alternativas

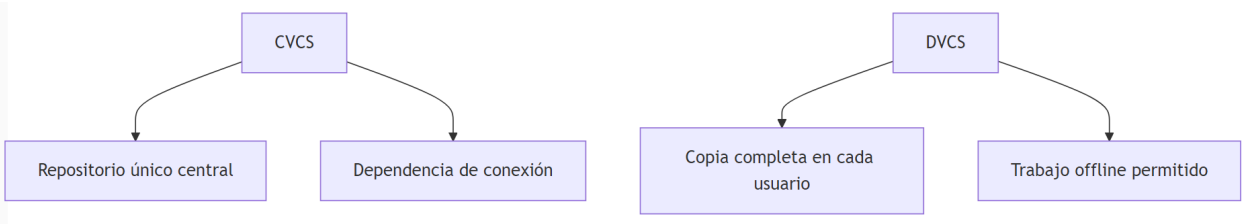
7.4.1 Centralizados (SVN, CVS)

En los **Sistemas de Control de Versiones Centralizados (CVCS)**, como **SVN** o **CVS**, existe un único repositorio centralizado donde se almacenan los archivos del proyecto. Los desarrolladores deben conectarse a este servidor para obtener las versiones más recientes del código y realizar cambios.

7.4.2 Distribuidos (Git, Mercurial)

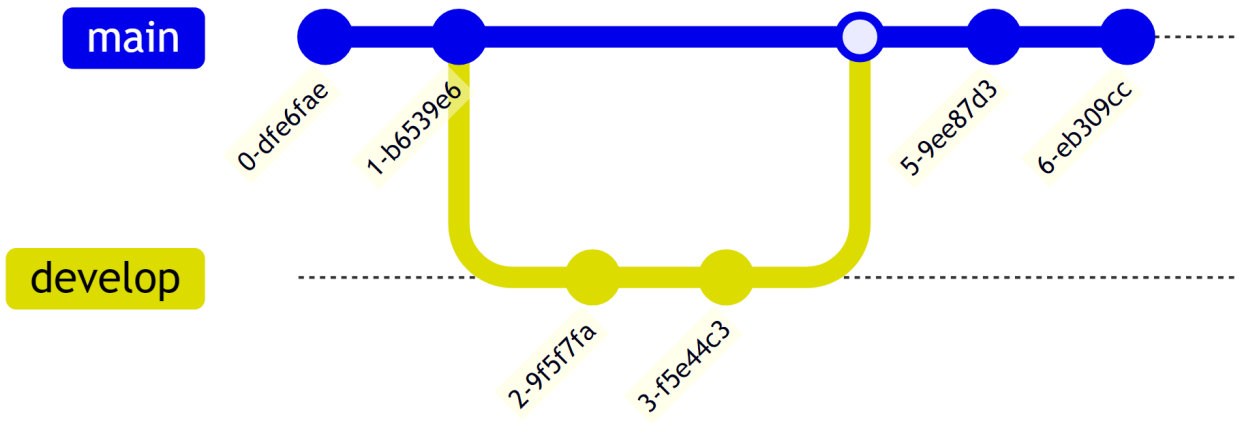
En los **Sistemas de Control de Versiones Distribuidos (DVCS)**, como **Git** o **Mercurial**, cada desarrollador tiene una copia completa del repositorio, incluyendo todo el historial de cambios. Esto permite trabajar de manera más eficiente sin depender de una conexión constante al servidor central.

Característica	Centralizados (SVN, CVS)	Distribuidos (Git, Mercurial)
Repositorio	Un único repositorio central	Cada usuario tiene una copia completa
Conectividad	Se necesita conexión al servidor	Se puede trabajar offline
Colaboración	Menos flexible	Alta flexibilidad y colaboración



7.5 Git como sistema de control de versiones

Git es el sistema de control de versiones distribuido más popular, creado por Linus Torvalds. Git permite la gestión eficiente de proyectos grandes y pequeños, proporcionando herramientas para colaboración, revisión y control de versiones de forma descentralizada.



7.5.1 Instalación, configuración y comandos básicos

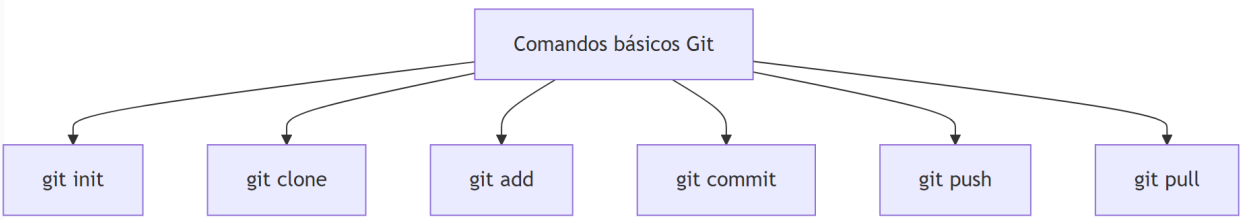
Para instalar y configurar Git en un sistema Linux:

```
# Instalar Git en Ubuntu
sudo apt update
sudo apt install git

# Configurar el nombre de usuario y correo
git config --global user.name "Tu Nombre"
git config --global user.email "tuemail@ejemplo.com"
```

Comandos básicos de Git:

- **git init**: Inicializa un nuevo repositorio.
- **git clone**: Clona un repositorio remoto.
- **git add**: Añade archivos al área de preparación para un commit.
- **git commit**: Realiza un commit de los cambios.
- **git push**: Envía los cambios al repositorio remoto.
- **git pull**: Obtiene los cambios del repositorio remoto.



7.5.2 Utilización de Git en repositorios locales

Git permite trabajar en repositorios locales antes de subir los cambios a un repositorio remoto. Esto permite a los desarrolladores realizar commits locales, probar el código y luego enviarlo al servidor cuando esté listo.

```
# Crear un nuevo repositorio local
git init

# Añadir un archivo al área de preparación
git add archivo.txt

# Realizar un commit
```

```
git commit -m "Primer commit"
```

7.5.3 Commits y restauración de archivos

Los **commits** son puntos en el historial del proyecto que permiten realizar cambios, y estos pueden ser revertidos si se produce un error.

```
# Revertir un archivo a una versión anterior
git checkout -- archivo.txt
```

7.5.4 Ignorando archivos

Git permite ignorar archivos específicos utilizando el archivo `.gitignore`. Esto es útil para evitar que archivos no deseados (como dependencias o archivos temporales) se agreguen al repositorio.

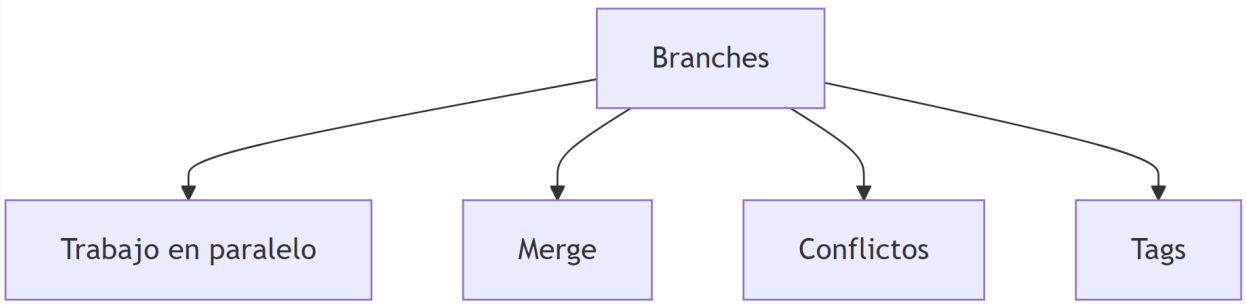
Ejemplo de `.gitignore`:

```
# Ignorar archivos temporales
*.tmp

# Ignorar directorio de dependencias
node_modules/
```

7.5.5 Ramas, uniones, conflictos y tags

- **Ramas (Branches)**: Permiten a los desarrolladores trabajar en paralelo en diferentes características o correcciones sin afectar el código principal.
- **Uniones (Merge)**: Proceso de combinar los cambios de diferentes ramas en una sola.
- **Conflictos**: Ocurren cuando dos ramas modifican la misma parte del código y deben resolverse manualmente.
- **Tags**: Se utilizan para marcar versiones específicas del código, como lanzamientos o hitos.



7.5.6 Stash y rebase

- **Stash**: Permite guardar cambios sin realizar un commit, lo que es útil cuando necesitas cambiar de rama pero aún tienes trabajo sin finalizar.

```
# Guardar cambios en stash
git stash

# Recuperar cambios del stash
git stash apply
```

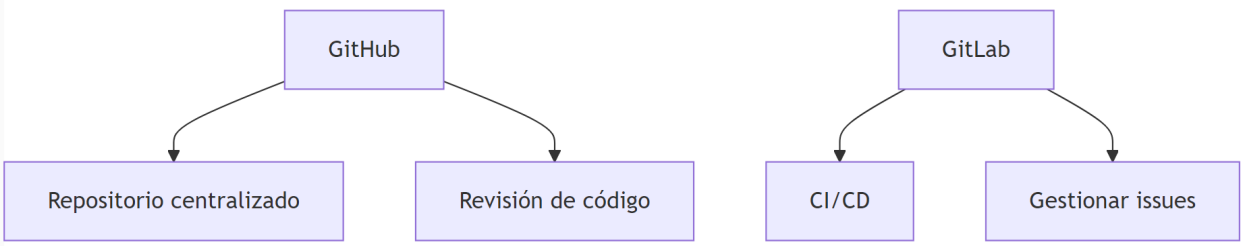
- **Rebase**: Reorganiza commits para crear una historia de commits más limpia y lineal. Es útil para mantener el historial del proyecto más claro.

```
# Rebase de una rama
git rebase main
```

7.6 Centralización de repositorios

7.6.1 Servicios de centralización de repositorios (GitLab, GitHub, Bitbucket)

GitHub, **GitLab**, y **Bitbucket** son plataformas que permiten centralizar repositorios Git, facilitando la colaboración, gestión de proyectos y la integración continua. Estas plataformas ofrecen herramientas para la revisión de código, seguimiento de problemas y despliegue automático.



7.6.2 Repositorios remotos, push y pull

- **Push**: Envía cambios locales al repositorio remoto.
- **Pull**: Obtiene los cambios del repositorio remoto.

```
# Enviar cambios locales a un repositorio remoto
git push origin main

# Obtener cambios desde el repositorio remoto
git pull origin main
```

7.6.3 Fetch vs Pull

- **git fetch**: Obtiene los cambios del repositorio remoto, pero no los aplica a la rama actual.
- **git pull**: Obtiene y aplica los cambios del repositorio remoto en la rama actual.

7.6.4 Clone y Fork de un repositorio

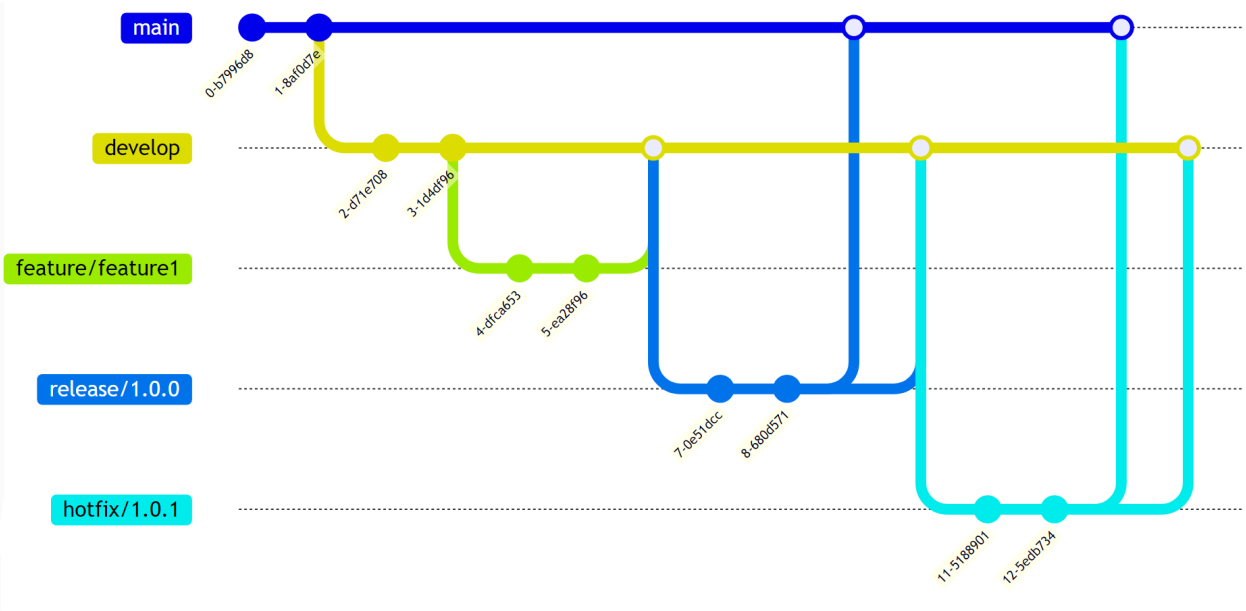
- **Clone**: Copia completa de un repositorio, permitiendo al desarrollador trabajar en una versión local del proyecto.

```
# Clonar un repositorio
git clone https://github.com/usuario/proyecto.git
```

- **Fork**: Copia de un repositorio en tu cuenta para hacer modificaciones sin afectar el repositorio original. Es común en GitHub para contribuir a proyectos de código abierto.

7.6.5 Flujos de trabajo

Existen varios **flujos de trabajo** en Git, como **Git Flow** y **GitHub Flow**, que definen la manera en que los equipos deben organizar su trabajo con ramas y commits.



Git y otros sistemas de control de versiones permiten una colaboración eficiente en proyectos de software, garantizando que los desarrolladores puedan trabajar juntos sin conflictos y con un control completo del historial del proyecto.

Material de Referencia

Libros:

- **"Version Control with Git"** de Jon Loeliger y Matthew McCullough
  Este libro ofrece una guía detallada para usar Git, desde los conceptos básicos hasta temas avanzados como la fusión de ramas, rebase y resolución de conflictos. Ideal para desarrolladores que buscan aprender Git a profundidad y aplicarlo en sus proyectos.

- **"Pro Git"** de Scott Chacon y Ben Straub
  Uno de los libros más completos sobre Git, cubriendo desde los conceptos fundamentales hasta los flujos de trabajo avanzados como GitFlow. Disponible gratuitamente en línea, es una excelente referencia para desarrolladores de todos los niveles.
  Pro Git - Libro Gratis

- **"Git Pocket Guide"** de Richard E. Silverman
  Un recurso compacto que cubre todos los comandos y funcionalidades clave de Git. Es ideal para desarrolladores que buscan una referencia rápida para usar Git de manera eficiente.

- **"The DevOps Handbook"** de Gene Kim, Patrick Debois, John Willis, y Jez Humble
  Aunque no está exclusivamente centrado en Git, este libro proporciona una perspectiva integral sobre cómo Git y otros sistemas de control de versiones encajan en el ciclo DevOps y la integración continua.

Enlaces a Recursos Online:

- **Git - Sitio Oficial**: El sitio oficial de Git, que incluye la documentación completa, descargas y recursos para aprender sobre cómo usar Git y sus diferentes funcionalidades.

- **Atlassian Git Tutorials**: Atlassian, creadores de herramientas como Bitbucket, ofrece una serie de tutoriales que cubren todos los aspectos de Git, desde comandos básicos hasta flujos de trabajo avanzados y estrategias para la gestión de ramas.

- **Pro Git - Libro en Línea**: Versión en línea del popular libro "Pro Git", disponible de manera gratuita. Es una referencia extensa que cubre tanto los fundamentos como temas más avanzados, como la manipulación de ramas y la gestión de grandes proyectos.

- **GitLab CI/CD Documentation**: Una guía completa de **GitLab** que explica cómo integrar Git con CI/CD (Integración Continua/Despliegue Continuo), proporcionando ejemplos prácticos y tutoriales sobre cómo configurar pipelines automatizados.

Videos Recomendados:

- **Git and GitHub Crash Course For Beginners (Traversy Media)**: Un curso de introducción sobre Git y GitHub que cubre los conceptos básicos y el uso de comandos esenciales para gestionar repositorios de código. Es una excelente opción para quienes están comenzando a usar Git.

- **Git for Professionals Tutorial - FreeCodeCamp**: Este tutorial cubre no sólo los comandos básicos de Git, sino también temas avanzados como el rebase, stash y el manejo de conflictos, lo que lo hace ideal para quienes buscan profundizar en Git.

Otros Recursos Útiles:

- **Git Flow Cheatsheet**. Una hoja de referencia rápida que detalla los comandos esenciales para trabajar con **Git Flow**, un flujo de trabajo ampliamente utilizado en proyectos que requieren múltiples ramas y lanzamientos.

- **Oh My Git!**: Un juego interactivo para aprender Git de manera divertida. Ayuda a entender los conceptos básicos y cómo funcionan los comandos en un entorno visual.

- **GitHub Actions Documentation**: GitHub Actions es una herramienta poderosa para automatizar flujos de trabajo de CI/CD. Esta documentación enseña cómo integrarla con Git para automatizar pruebas y despliegues.

CORFO
BecasCapital Humano

adalid
aspasia
LA FORMACIÓN DE TU FUTURO

MÓDULO 8

FUNDAMENTOS DE INTREGRACIÓN CONTINÚA