

Programa DevOps



≡ DevOps e Infraestructura

≡ 1. Introducción a DevSecOps

En este manual daremos los primeros pasos hacia la comprensión de la cultura DevSecOps, comenzaremos entendiendo el rol y los conceptos principales para embarcarnos luego en los conceptos de Cloud Computing y Control de Versiones. Más adelante veremos Conceptos fundamentales de la cultura DevSecOps y las herramientas más utilizadas por el mercado. Para finalizar comprenderemos cómo crear Pipelines y entenderemos los principios de CI/CD y cómo los mismos benefician el ciclo de vida aplicativo mediante la implementación de Automatización y DevOps.

¡Manos a la obra! 

Objetivos de aprendizaje

- Comprender el papel de DevSecOps y Cloud Computing en el ciclo de vida aplicativo.
- Conocer las principales herramientas y conceptos asociados a la práctica.
- Entender los principios de CI/CD y Pipelines y como la empresa puede beneficiarse al implementarlos.

Índice

DevOps e Infraestructura

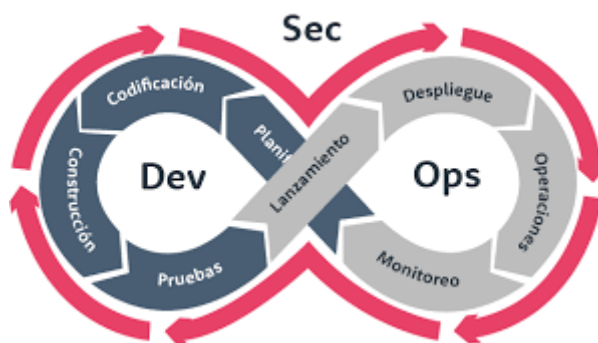
1. Introducción a DevSecOps	2
1.1 Concepto	4
1.2 ¿Por qué DevOps?	5
1.3 Tareas principales del Rol	6
• 1.3.1. Roles y Responsabilidades	6
• 1.3.2. Las 9 principales habilidades de los ingenieros de DevOps	9
• 1.3.4. DevOps es más que solo un rol	12
IconoDescripción generada automáticamente 1.4 Introducción a Cloud – Nube (CloudOps)	12

• 1.4.1. On Premises	13
• 1.4.2. Nube Pública vs Nube Privada	13
• 1.4.3. Infrastructure as a Service (IaaS)	16
• 1.4.4. Platform as a Service (PaaS)	16
• 1.4.5. Software as a Service (SaaS)	16
• 1.4.6. Diagrama de Servicios (OP, IaaS, PaaS, SaaS)	16
• 1.4.7. Ejemplos de Proveedores de Nube (Cloud Providers)	17
1.5 Herramientas de Control de Versiones	18
• 1.5.1. Concepto	18
• 1.5.2. Código Fuente y Pertenencia	18
• 1.5.3. Sistemas de Control de Versiones	19
• 1.5.4. Branches o Ramas	27
1.6 Introducción a Automatización	31
• 1.6.1. Conceptos y Herramientas	31
1.7 Branching Strategies – Estrategias de Ramificación	42
• 1.7.1. Concepto	42
• 1.7.2. Motivo	42
• 1.7.3. GitFlow	43
• 1.7.4. GitHub Flow	44
1.8 Ciclo de Vida Aplicativo	46
• 1.8.1. Ambientes del ciclo de vida aplicativo	46
• 1.8.2. Etapas del Ciclo de Vida Aplicativo	47
1.9 Ciclo de Vida Aplicativo	49
• 1.9.1. Integración Continua (CI)	49
• 1.9.2. Entrega Continua y Despliegue Continuo (CD)	50
• 1.9.3. Entrega continua vs Despliegue continuo	51

• 1.9.4. Pipelines	52
• 1.9.5. Herramientas de CI/CD	54
Resumen	62
Referencias	62

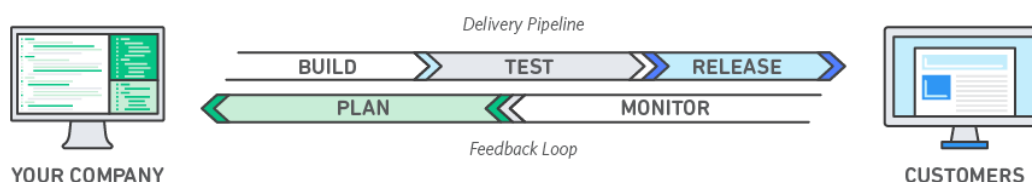
1.1 Concepto

DevOps deriva de la combinación de palabras Developer y Operations. Definiendo una clara combinación entre un rol capaz de resolver los problemas de Código y scripts, conociendo y proporcionando mejoras en la automatización de procesos a través de Código, como así también un rol de Operaciones, donde los despliegues, modificaciones y optimizaciones tanto a infraestructura como herramientas pueden ser ejecutados.



Comúnmente se habla de un Rol DevOps, pero esto es incorrecto, DevOps es una metodología. Que puede implementarse en diferentes roles de la compañía siendo el SRE (Site Reliability Engineer – Ingeniero de Confiabilidad del Sitio) quien se acerca más a su implementación en el uso del día a día.

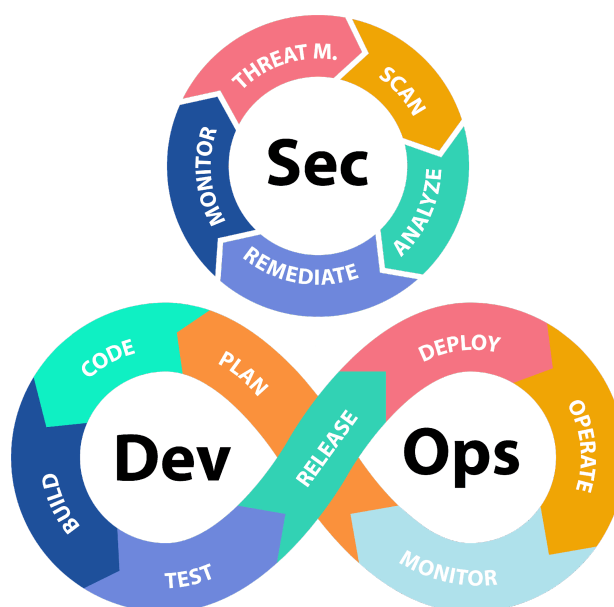
La evolución de DevOps ha sido agregar además los principios de seguridad informática (InfoSec) en esta práctica o metodología, obteniendo así el nombre DevSecOps.



DevSecOps constituye una combinación de filosofías culturales, prácticas y herramientas que incrementan la capacidad de una organización de proporcionar aplicaciones y servicios a gran velocidad: desarrollar y mejorar productos con mayor rapidez que las organizaciones que utilizan procesos tradicionales de desarrollo de software y

administración de la infraestructura. Esta velocidad permite a las organizaciones servir mejor a sus clientes y competir de forma más eficaz en el mercado.

≡ 1.2 ¿Por qué DevOps?



Bajo un modelo de DevOps, los equipos de desarrollo y operaciones ya no están “aislados”. A veces, los dos equipos se fusionan en uno solo, donde los ingenieros trabajan en todo el ciclo de vida de la aplicación, desde el desarrollo y las pruebas hasta la implementación y las operaciones, y desarrollan una variedad de habilidades no limitadas a una única función.

En algunos modelos de DevOps, los equipos de control de calidad y de seguridad también se integran más con el desarrollo y las operaciones e intervienen durante todo el ciclo de vida de la aplicación. Cuando la seguridad es la prioridad de todos los miembros de DevOps, a veces se conoce como DevSecOps, como indiqué anteriormente.

Los equipos utilizan prácticas para automatizar los procesos que anteriormente habían sido manuales y lentos. Utilizan una pila de tecnología y herramientas que los ayudan a operar y mejorar aplicaciones de forma rápida y confiable. Además, estas herramientas ayudan a los ingenieros a realizar de forma independiente tareas que normalmente hubieran requerido la ayuda de otros equipos (por ejemplo, implementar código o aprovisionar infraestructura), lo que incrementa todavía más la velocidad del equipo.

Mediante el uso de Herramientas específicas y lenguajes de programación rápidos o scripts los equipos que implementan DevSecOps son capaces de automatizar procesos y lograr así tiempos de respuestas más altos, estándares de calidad más justos y un bienestar general de satisfacción y logro al equipo, al ser capaces de encontrar defectos y optimizar tiempos en cada ciclo de entrega.

≡ 1.3 Tareas principales del Rol

Un ingeniero de DevOps es un generalista de TI que debe tener un amplio conocimiento tanto del desarrollo como de las operaciones, lo que incluye programación, gestión de infraestructuras, administración de sistemas y cadenas de herramientas de DevOps. Los ingenieros de DevOps también deben tener habilidades interpersonales, ya que trabajan con los distintos grupos aislados de la empresa para crear un entorno más colaborativo. Los ingenieros de DevOps necesitan sólidos conocimientos de la arquitectura común del sistema, el aprovisionamiento y la administración, pero también deben tener experiencia con el conjunto de herramientas y prácticas tradicionales de los desarrolladores, como el control del código fuente, la entrega y recepción de revisiones de código, la escritura de pruebas unitarias y la familiaridad con los principios de la metodología ágil.

● 1.3.1. Roles y Responsabilidades

El papel de un ingeniero de DevOps puede ser distinto de una organización a otra, pero invariablemente implica una combinación de ingeniería de publicación, aprovisionamiento y gestión de infraestructuras, administración de sistemas, seguridad y la promoción de DevOps.



La ingeniería de publicación incluye el trabajo necesario para compilar e implementar código de aplicaciones. Las herramientas y los procesos exactos varían mucho en función de un gran número de variables, como el lenguaje en el que se programa el

código, cuánta canalización se ha automatizado y si la infraestructura de producción es local o en la nube. La ingeniería de publicación puede implicar seleccionar, aprovisionar y mantener herramientas de CI/CD o escribir y mantener scripts de compilación/implementación personalizados.

El aprovisionamiento de infraestructuras y la administración de sistemas incluyen la implementación y el mantenimiento de los servidores, el almacenamiento y los recursos de red necesarios para alojar aplicaciones. Para organizaciones con recursos locales, esto puede incluir la gestión de servidores físicos, dispositivos de almacenamiento, conmutadores y software de virtualización en un Data Center. Si se trata de una organización híbrida o totalmente basada en la nube, esto suele incluir el aprovisionamiento y la gestión de instancias virtuales de los mismos componentes.

La promoción de la cultura DevOps a menudo se infravalora o se pasa por alto por completo, pero podría decirse que es la función más importante de un ingeniero de DevOps. El cambio a una cultura de DevOps puede ser disruptivo y confuso para los miembros del equipo de ingeniería. Como experto en DevOps, corresponde al ingeniero de DevOps promover y enseñar este enfoque en toda la organización.

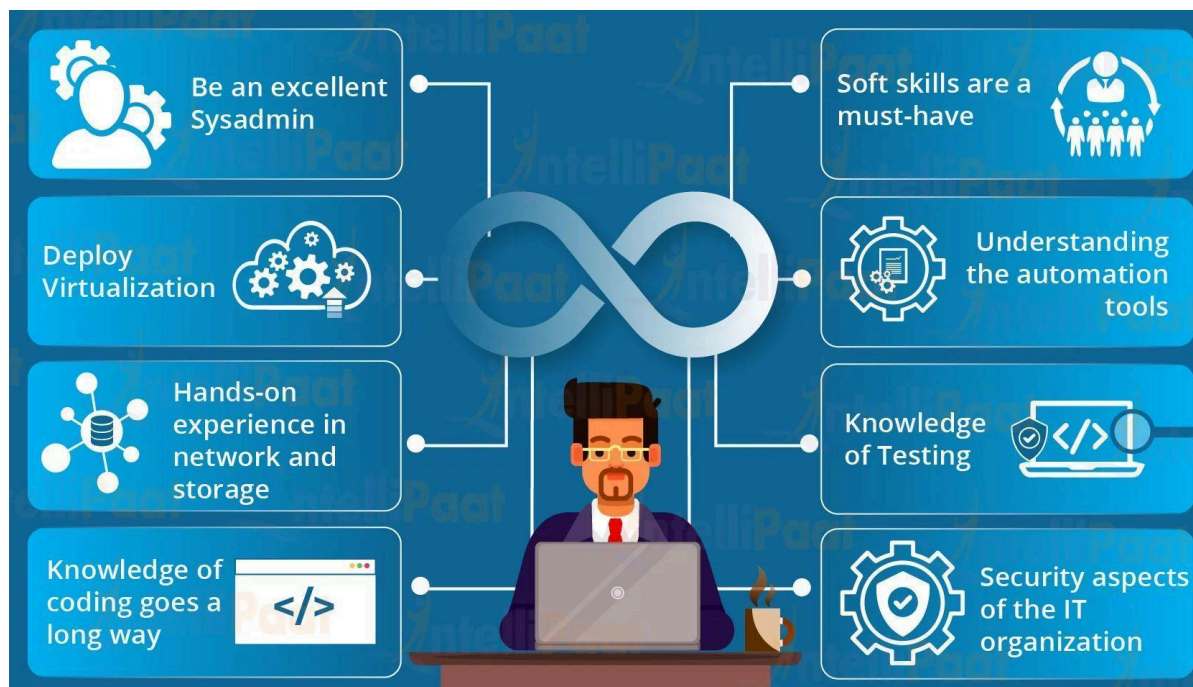
1. **Automatización de la entrega de software:** Automatizar el proceso de desarrollo, pruebas y despliegue de aplicaciones para garantizar una entrega rápida y confiable. A través de herramientas, scripts y Pipelines
2. **Configuración y mantenimiento de herramientas de CI/CD:** Implementar y mantener herramientas de integración continua (CI) y entrega continua (CD), como Jenkins, Bitbucket Pipelines, Travis CI, GitLab CI/CD, Azure DevOps, AWS CodeBuild y AWS CodePipeline o CircleCI.
3. **Gestión de la infraestructura como código (IaC):** Utilizar herramientas como Terraform o Ansible para definir y gestionar la infraestructura de manera programática.
4. **Colaboración estrecha con desarrolladores y operaciones:** Fomentar la colaboración entre equipos de desarrollo y operaciones para eliminar silos y mejorar la eficiencia.
5. **Gestión de contenedores y orquestación:** Utilizar tecnologías como Docker y Kubernetes para gestionar contenedores y orquestar aplicaciones en entornos de producción.
6. **Automatización de pruebas y despliegues:** Crear flujos de trabajo automatizados de pruebas unitarias, de integración, de carga y funcionales, así como implementar despliegues sin intervención manual.

7. **Gestión de configuración:** Gestionar la configuración de aplicaciones y servidores de manera consistente y escalable.
8. **Monitorización y registro:** Configurar herramientas de monitoreo como Prometheus, Grafana, ELK Stack o Splunk para supervisar el rendimiento y la disponibilidad de servicios.
9. **Seguridad de la infraestructura:** Implementar medidas de seguridad en la infraestructura, como cortafuegos, detección de intrusiones y gestión de identidad y acceso.
10. **Gestión de versiones y control de cambios:** Utilizar sistemas de control de versiones como Git para rastrear y gestionar cambios en código, configuración y otros recursos.
11. **Optimización de recursos:** Identificar y eliminar recursos no utilizados o subutilizados para optimizar costos y escalabilidad.
12. **Resolución de problemas y soporte:** Diagnosticar y resolver problemas de despliegue, rendimiento y configuración en entornos de producción. Como soporte Nivel 4 cuando otros equipos no puedan encontrar la solución correcta.
13. **Documentación y buenas prácticas:** Mantener documentación actualizada sobre la configuración, los flujos de trabajo y las prácticas recomendadas de DevOps.
14. **Asegurar la alta disponibilidad y la recuperación ante desastres:** Implementar estrategias para garantizar la disponibilidad de aplicaciones y la recuperación en caso de fallas.
15. **Actualizaciones y parches:** Gestionar y aplicar actualizaciones de seguridad y parches en servidores y aplicaciones de manera oportuna.
16. **Participación en la planificación y revisión de arquitectura:** Colaborar en la definición de la arquitectura de aplicaciones para garantizar su despliegue y escalabilidad eficientes.

● 1.3.2. Las 9 principales habilidades de los ingenieros de DevOps

Las habilidades técnicas que necesita un ingeniero de DevOps variarán en función de la estructura del equipo, las tecnologías y los conjuntos de herramientas que se utilicen. Sin embargo, es esencial contar con sólidas habilidades de comunicación y colaboración.

También es importante que un ingeniero de DevOps conozca bien todos los componentes de una canalización de entrega (Pipelines), así como las ventajas y desventajas de las herramientas y los servicios disponibles.



Comunicación y colaboración ☎

Es importante que un ingeniero de DevOps se comunique y colabore de forma eficaz con los equipos, los gestores y los clientes. Las habilidades interpersonales a menudo se pasan por alto y se infravaloran, pero el éxito de DevOps depende en gran medida de la calidad y la cantidad de feedback en todo el flujo de valor.

Administración de sistemas 🔄

Un ingeniero de DevOps tiene experiencia en la administración de sistemas, como el aprovisionamiento y la gestión de servidores, la implementación de bases de datos, la supervisión de seguridad, la aplicación de parches en sistemas y la gestión de la conectividad de red interna y externa.

Experiencia con herramientas de DevOps 🛠

Dado que el uso de las herramientas adecuadas es esencial para las prácticas de DevOps, el ingeniero de DevOps debe conocer y saber utilizar distintas herramientas.

Estas herramientas abarcan todo el ciclo de vida de DevOps, desde la infraestructura y la creación hasta la supervisión y el funcionamiento de un producto o servicio.

Gestión de la configuración

A menudo se espera que los ingenieros de DevOps tengan experiencia con una o varias herramientas de gestión de la configuración como Chef, Puppet o Ansible. Muchas organizaciones han adoptado estas herramientas o herramientas similares para automatizar las tareas de administración de sistemas, como la implementación de nuevos sistemas o la aplicación de parches de seguridad a los sistemas que ya están en funcionamiento.

Contenedores y orquestación de contenedores

Con la contenerización, una tecnología popularizada por Docker, el código de la aplicación y su entorno de ejecución se agrupan en la misma imagen. Esto hace que las herramientas de gestión de la configuración tradicionales no sean tan necesarias. A su vez, la gestión de contenedores conlleva sus propios desafíos, y tener experiencia con el tipo de herramientas conocido como "orquestadores de contenedores" (por ejemplo, Docker Swarm o Kubernetes) se convierte en una habilidad imprescindible para los ingenieros de DevOps.

Integración e implementación continuas

La integración y la entrega continuas (CI/CD) son prácticas básicas de un enfoque de DevOps para el desarrollo de software, y están habilitadas por una serie de herramientas disponibles. La función principal de cualquier herramienta o conjunto de herramientas de CI/CD es automatizar el proceso de compilación, prueba e implementación de software.

Los ingenieros de DevOps suelen necesitar experiencia en la configuración e implementación de una o más herramientas de CI/CD y, por lo general, deberán trabajar en estrecha colaboración con el resto de la organización de desarrollo para garantizar que estas herramientas se utilizan de forma eficaz.

Arquitectura y aprovisionamiento del sistema

Un ingeniero de DevOps debe tener la capacidad de diseñar, aprovisionar y gestionar ecosistemas informáticos, ya sea de forma local o en la nube. Es importante entender la infraestructura como código (IaC), un proceso de gestión de TI que aplica las prácticas recomendadas desde el desarrollo de software de DevOps hasta la gestión de los recursos de la infraestructura en la nube. Un ingeniero de DevOps debe saber modelar la infraestructura del sistema en la nube con Ansible o Terraform, y cualquiera de sus variantes dependiendo del proveedor de Nube (AWS CloudFormation, Azure ARM, etc.)

Conocimientos de programación y scripts

Muchos administradores de sistemas tradicionales tienen experiencia con scripts de línea de comandos para automatizar tareas repetitivas. Un ingeniero de DevOps debe ir más allá de los scripts de automatización y conocer las prácticas avanzadas de desarrollo de software y cómo implementar prácticas de desarrollo ágil, como las revisiones de código y el control de código fuente.

Capacidades de gestión colaborativa 🙏

La colaboración entre equipos es un componente fundamental para que una estrategia de DevOps sea eficaz, independientemente de la estructura organizativa específica. No importa si el equipo de ingeniería se divide en funciones o incluye equipos independientes para el desarrollo de funciones, el control de calidad, DevOps, etc., el ingeniero de DevOps debe ser orientador y compañero de trabajo de muchas personas diferentes de la organización.

Por ejemplo, una de las mayores ventajas de apostar por DevOps es la capacidad de proporcionar retroalimentación más rápido a los desarrolladores. Los ingenieros de DevOps a menudo tienen que trabajar con el equipo de control de calidad (ya sean evaluadores manuales o desarrolladores encargados de la automatización de pruebas) para mejorar la rapidez, la eficacia y el resultado de las metodologías de pruebas.

Al mismo tiempo, los desarrolladores pueden necesitar el apoyo de los ingenieros de DevOps para mejorar el proceso de compilación e implementación de código de las aplicaciones.

● 1.3.4. DevOps es más que solo un rol

DevOps es una práctica que requiere un cambio cultural, nuevos principios de gestión y el uso de herramientas tecnológicas. El ingeniero de DevOps es el núcleo de la transformación a DevOps y debe tener un amplio conjunto de habilidades para facilitar este cambio. Sin embargo, la mayoría de las organizaciones necesitan algo más que un ingeniero de DevOps, normalmente una combinación de generalistas y especialistas que colaboren estrechamente para implementar el enfoque de DevOps y mejorar el ciclo de vida del desarrollo de software. Un ingeniero de DevOps ayuda a acabar con los grupos aislados para facilitar la colaboración entre los distintos expertos y en todas las cadenas de herramientas para beneficiarse por completo de DevOps.

≡ 1.4 Introducción a Cloud – Nube (CloudOps)



La definición de la nube puede parecer poco clara, pero, básicamente, es un término que se utiliza para describir una red mundial de servidores, cada uno con una función única. La nube no es una entidad física, sino una red enorme de servidores remotos de todo el mundo que están conectados para funcionar como un único ecosistema. Estos servidores están diseñados para almacenar y administrar datos, ejecutar aplicaciones o entregar contenido o servicios, como streaming de vídeos, correo web, software de ofimática o medios sociales. En lugar de acceder a archivos y datos desde un equipo personal o local, accedes a ellos online desde cualquier dispositivo conectado a Internet, es decir, la información está disponible dondequiera que vayas y siempre que la necesites.

Las empresas utilizan cuatro métodos diferentes para implementar recursos en la nube. Hay una nube pública, que comparte recursos y ofrece servicios al público a través de Internet; una nube privada, que no se comparte y ofrece servicios a través de una red interna privada, normalmente hospedada en el entorno local; una nube híbrida, que comparte servicios entre nubes públicas y privadas, según su finalidad; y una nube comunitaria, que comparte recursos solo entre organizaciones, por ejemplo, con instituciones gubernamentales.

● 1.4.1. On Premises

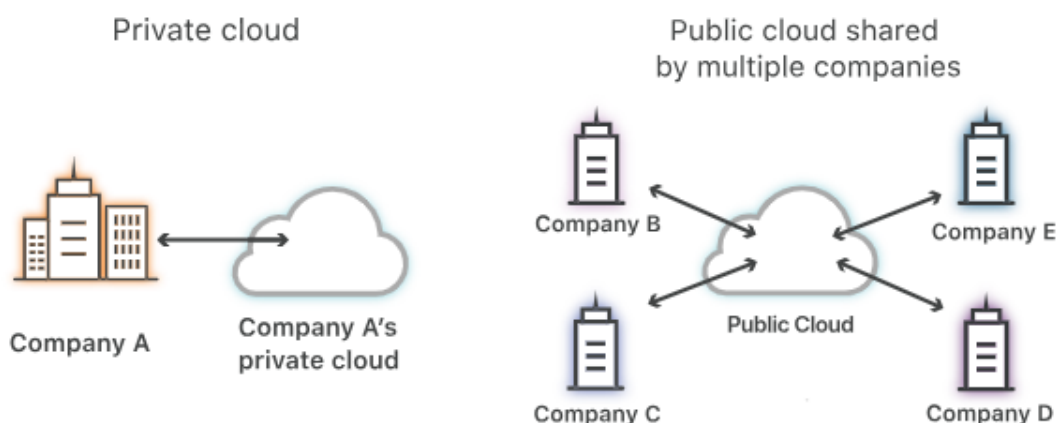
"On premises" (también escrito como "on-premises" o "on-premise") se refiere a la ubicación física de los recursos informáticos y de tecnología de una organización, como servidores, equipos de red y software, que se encuentran y operan dentro de las instalaciones físicas de la propia organización o empresa. En otras palabras, cuando algo está "on premises", significa que está localizado y gestionado en las instalaciones de la organización en lugar de estar en la nube o en un centro de datos externo.

El término es comúnmente utilizado en contraste con la "nube" o "cloud", donde los servicios y recursos se alojan en servidores remotos gestionados por terceros proveedores de servicios en la nube. La elección entre tener una infraestructura "on premises" o utilizar servicios en la nube depende de las necesidades y recursos de la organización, así como de consideraciones como la inversión inicial, la escalabilidad y la seguridad.

● 1.4.2. Nube Pública vs Nube Privada

Una nube privada es un servicio en la nube que no está compartido con ninguna otra organización. El usuario de la nube privada tiene un uso exclusivo de la nube.

En cambio, una nube pública es un servicio en la nube que comparte servicios informáticos entre diferentes clientes, aunque los datos y las aplicaciones de cada cliente permanecen ocultos para los otros clientes de la nube.



Una nube pública es como alquilar un piso, mientras que una nube privada es como alquilar una casa unifamiliar con un tamaño similar. La casa ofrece más privacidad, pero también el alquiler es mayor y no es el uso más eficiente de los recursos. El mantenimiento del piso lo gestiona el supervisor del edificio, mientras que es más difícil

conseguir que un contratista arregle la casa (en ocasiones, el inquilino tendrá que hacerlo por sí mismo).

Hay nubes privadas alojadas, ofrecidas por un proveedor de nube externo, y nubes privadas internas, gestionadas y mantenidas por una organización a nivel interno.

Nube Pública: Pros y Contras

● Pros:

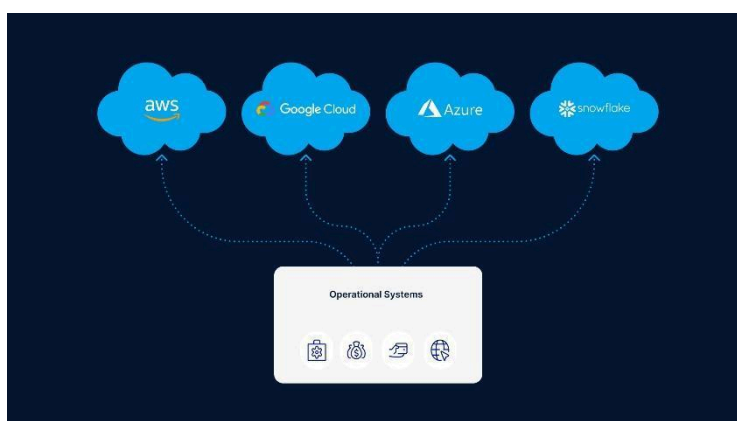
1. **Costo inicial reducido:** No necesitas invertir en infraestructura física costosa, lo que reduce los costos iniciales.
2. **Elasticidad:** Puedes escalar los recursos según tus necesidades, lo que te permite ahorrar dinero cuando no necesitas capacidad adicional.
3. **Facilidad de uso:** Los proveedores de nube pública ofrecen interfaces de usuario intuitivas y herramientas de gestión que facilitan la administración de recursos.
4. **Acceso global:** Puedes acceder a tus datos y aplicaciones desde cualquier lugar con conexión a Internet.
5. **Seguridad mejorada:** Los principales proveedores de nube pública suelen invertir en medidas de seguridad avanzadas, como cifrado y detección de amenazas, para proteger tus datos.
6. **Actualizaciones y mantenimiento automáticos:** Los proveedores se encargan de las actualizaciones de software y el mantenimiento de la infraestructura, liberándote de esa responsabilidad.
7. **Escalabilidad:** Puedes escalar recursos de manera rápida y sencilla para hacer frente a picos de demanda o crecimiento del negocio.

● Contras:

1. **Seguridad y privacidad:** Aunque los proveedores se preocupan por la seguridad, aún existen preocupaciones sobre la privacidad de los datos y posibles violaciones.
2. **Costos a largo plazo:** A largo plazo, los costos de suscripción a servicios de nube pueden superar los gastos iniciales de infraestructura propia.

3. **Dependencia del proveedor:** Estás atado a las políticas y precios del proveedor de nube, lo que puede limitar tu flexibilidad.
4. **Latencia:** La latencia de red puede ser un problema en aplicaciones que requieren tiempos de respuesta muy bajos.
5. **Personalización limitada:** Algunas aplicaciones y cargas de trabajo pueden requerir configuraciones específicas que no son compatibles con la nube pública.
6. **Costos imprevistos:** Los costos pueden aumentar si no se gestiona adecuadamente el uso de los recursos.
7. **Cumplimiento normativo:** Cumplir con regulaciones específicas, como HIPAA o GDPR, puede ser más complicado en la nube pública debido a la compartición de recursos.

MultiCloud y Nube Híbrida



Las implementaciones de multinube y nube híbrida incorporan nubes públicas:

- Multinube (MultiCloud) hace referencia al uso de muchas nubes públicas al mismo tiempo.
- Los despliegues de nube híbrida combinan una o más nubes públicas con una nube privada o con infraestructura en las instalaciones.

• 1.4.3. Infrastructure as a Service (IaaS)

La IaaS incluye los bloques de creación básicos para la TI basada en la nube. Generalmente, provee acceso a características de conexión en red, equipos (virtuales o en hardware exclusivo) y espacio de almacenamiento de datos. La IaaS le ofrece el mayor nivel de flexibilidad y control de administración en relación con sus recursos de TI. Es similar a los recursos de TI que muchos desarrolladores y departamentos de TI ya conocen. En otras palabras, permite que nos desliguemos de la infraestructura tradicional y nos ofrece servicios virtualizados de hardware y que paguemos solo por lo que consumimos, dejando de tener que pensar en mantenimiento, electricidad, gastos edilicios, seguridad física, etc. Pero aun somos responsables de todo lo que esté ocurriendo en nuestros dispositivos.

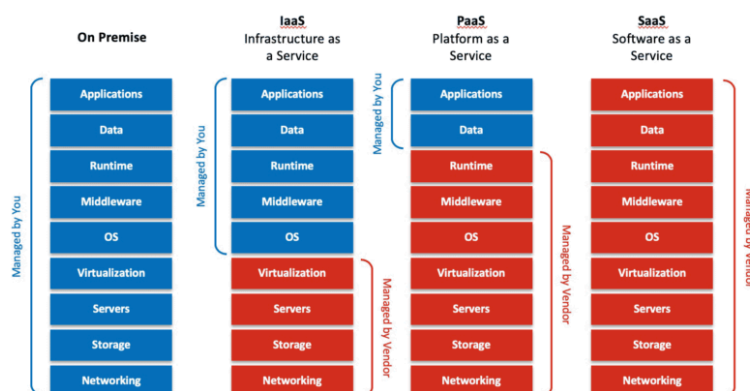
● 1.4.4. Platform as a Service (PaaS)

La PaaS elimina la necesidad de administrar la infraestructura subyacente (normalmente hardware y sistemas operativos) y permite enfocarse en la implementación y administración de aplicaciones. Esto contribuye a mejorar el nivel de eficiencia, ya que no debe preocuparse por el aprovisionamiento de recursos, la planificación de la capacidad, el mantenimiento del software, la implementación de parches ni ninguna de las demás arduas tareas que conlleva la ejecución de su aplicación.

● 1.4.5. Software as a Service (SaaS)

El SaaS proporciona un producto completo que el proveedor del servicio ejecuta y administra. En la mayoría de los casos, quienes hablan de SaaS en realidad se refieren a aplicaciones de usuario final (como el email basado en la Web). Con una solución basada en SaaS, ya no se debe pensar en cómo mantener el servicio ni en cómo administrar la infraestructura subyacente. Solamente debe pensar en cómo utilizar ese software en particular.

● 1.4.6. Diagrama de Servicios (OP, IaaS, PaaS, SaaS)



• 1.4.7. Ejemplos de Proveedores de Nube (Cloud Providers)

Tipo de Nube	Producto	Empresa Asociada
Nube Privada	vSphere	VMware, Inc.
Nube Privada	Azure Stack	Microsoft
Nube Privada	IBM Cloud Private	IBM Corporation
Nube Privada	Prism	Nutanix, Inc.
Nube Pública	AWS	Amazon Web Services, Inc.
Nube Pública	Azure	Microsoft Corporation
Nube Pública	Google Cloud (GCP)	Google LLC
Nube Pública	IBM Cloud	IBM Corporation
Nube Híbrida	EMC VxRail	Dell Technologies
Nube Híbrida	HPE GreenLake	Hewlett Packard Enterprise
Nube Híbrida	OpenShift	Red Hat, Inc.
Nube Híbrida	Oracle Cloud	Oracle Corporation

● 1.5 Herramientas de Control de Versiones

● 1.5.1. Concepto

El control de versiones, también conocido como control de cambios o versionado, es un sistema que permite llevar un registro de los cambios realizados en un conjunto de archivos o en un proyecto de software a lo largo del tiempo. El principal objetivo del control de versiones es proporcionar un historial completo de las modificaciones efectuadas en los archivos, lo que facilita la colaboración en proyectos, la identificación y resolución de problemas, y la gestión de múltiples versiones de un software o conjunto de documentos.

Características:

- Historial de cambios: El sistema registra quién realizó cada cambio, cuándo se hizo y qué se modificó en cada versión.
- Ramificación y fusión: Permite crear ramas de desarrollo paralelas para trabajar en nuevas características o correcciones de errores sin afectar la versión principal. Luego, se pueden fusionar estas ramas de vuelta a la versión principal.
- Reversión: Permite volver atrás en el historial y restaurar una versión anterior de los archivos si es necesario.
- Colaboración: Facilita la colaboración entre miembros del equipo, ya que varios desarrolladores pueden trabajar en el mismo proyecto de manera simultánea sin temor a sobrescribir el trabajo de otros.
- Etiquetado: Se pueden agregar etiquetas o marcas a versiones específicas para indicar hitos importantes, como versiones de lanzamiento.
- Auditoría: Proporciona un registro completo de todas las acciones realizadas en el sistema de control de versiones, lo que facilita la auditoría y la resolución de disputas.

● 1.5.2. Código Fuente y Pertenencia

El Código fuente es propiedad de la empresa, es por esto por lo que intelectualmente es su dueño, con lo cual no deberíamos nunca tomar decisiones a la ligera sobre el mismo,

pero hay un punto que está muy claro. Si tenemos proveedores externos, estos deben guardar el código fuente en un repositorio de la empresa y a su vez su propiedad debería ser de la empresa. Salvo casos especiales contractuales de llave en mano. ¿Por qué? Básicamente si en el futuro se desean corregir errores, modificar el producto o simplemente auditar no se tendrá control sobre el mismo sin necesidad de recurrir al proveedor. Además, contando con repositorio propio de fuente será más fácil llevar a cabo los controles necesarios de auditoría para saber quién cambió qué, y por qué.



Si el desarrollo se efectuó dentro de la empresa, entonces el Proyecto y la empresa son responsables y dueños del código, esto quiere decir que la responsabilidad de mantener el código estable y actualizado, sin errores y performante recae sobre los desarrolladores. Si bien es cierto que el equipo responsable de implementar DevSecOps podría ayudar a esto último mediante la implementación de automatizaciones y herramientas, el proceso de mergear (integrar) código entre ramas y efectuar los correspondientes arreglos al código no recae sobre quien implementa DevSecOps, sino de nuevo, íntegramente sobre el equipo de Desarrollo. Dicho de otro modo, así como QA-Testing no modifica código, DevSecOps tampoco debería, salvo en los casos donde se utilice Infraestructura como código (IaC), pero esta última se mantendría aparte de los repositorios de código fuente aplicativos.

¿Y los scripts Powershell/Bash? Bueno, es el mismo caso que IaC, dado que hablamos de lo mismo en caso de que no sean parte del aplicativo en cuestión, salvo cuando se trate de Scripts de Bases de Datos o Ingestión de Datos. En los casos de Data Ingestión se utiliza el mismo criterio que para el código fuente, en algunas empresas inclusive se tiene un rol especial para la gestión de Data Pipelines o ingestión de datos, no se ocupa ni el equipo de Desarrollo, ni el equipo de QA, ni los equipos que implementen DevSecOps.

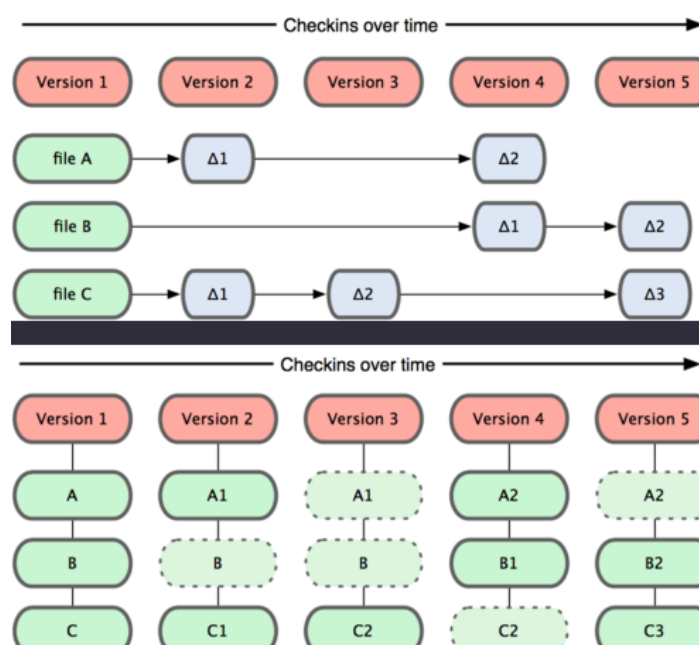
● 1.5.3. Sistemas de Control de Versiones

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración de este. Una versión, revisión o edición de un producto, es el estado en el que se encuentra dicho producto en un momento dado de su desarrollo o modificación. Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas que faciliten esta gestión dando lugar a los llamados sistemas de control de versiones o SVC (del inglés System Version Control).

Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo, para algún cliente específico). Ejemplos de este tipo de herramientas son entre otros: CVS, Subversion, BitBucket, SourceSafe, ClearCase, GitLab, GitHub, herramientas Basadas en GIT, Mercurial.

Git (GitHub, GitLab, BitBucket)

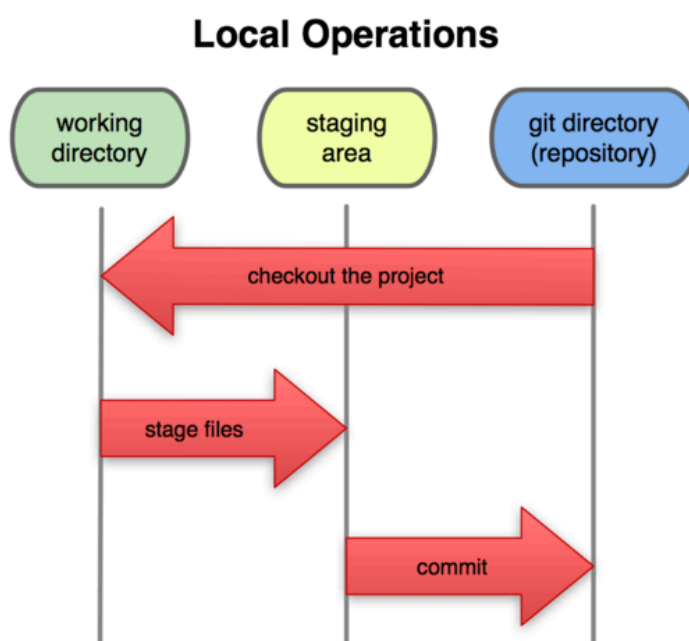
Git es un sistema de control de versiones distribuido que se diferencia del resto en el modo en que modela sus datos. La mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos, mientras que Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos.



Los tres estados

Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo, pero todavía no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).



Directorio de trabajo, área de preparación, y directorio de Git

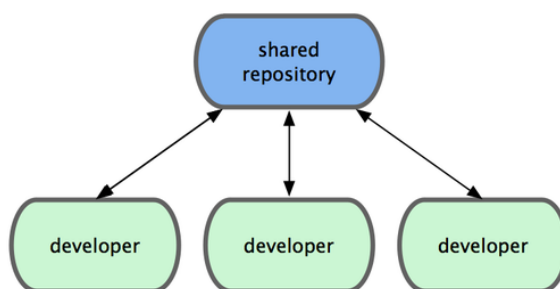
Para que se entienda mejor... el working directory es el directorio donde tengo mi código fuente, ejemplo c:\source

El staging area o Index es el espacio que está siendo seguido por git, esto es, todos los archivos que se encuentren aquí serán monitoreados y sus cambios registrados.

De dichos cambios nosotros podemos deshacernos o enviarlos al repositorio. En este caso con el commit lo enviamos al repositorio local, pero no así al repositorio remoto. O sea, los cambios se guardarán como si se tratara de un Savestate en un video juego, pero no serán enviados al repositorio remoto. Para esto deberemos utilizar un Push para enviar la información al repositorio remoto y limpiar los cambios pendientes de envío en nuestro repositorio local.

Flujo de trabajo centralizado

Existe un único repositorio o punto central que guarda el código y todo el mundo sincroniza su trabajo con él. Si dos desarrolladores clonan desde el punto central, y ambos hacen cambios; tan solo el primero de ellos en enviar sus cambios de vuelta lo podrá hacer limpiamente. El segundo desarrollador deberá fusionar previamente su trabajo con el del primero, antes de enviarlo, para evitar el sobrescribir los cambios del primero



Glosario Git

Repositorio ("repository")

El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios.

Revisión ("revision")

Una revisión es una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (Ej. subversión). Hay otros sistemas que identifican las revisiones mediante un código de detección de modificaciones (Ej. git usa SHA1).

Etiqueta ("tag")

Los tags permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo, se suelen usar tags para identificar el contenido de las versiones publicadas del proyecto.

Rama ("branch")

Un conjunto de archivos puede ser ramificado o bifurcado en un punto en el tiempo de manera que, a partir de ese momento, dos copias de esos archivos se pueden desarrollar a velocidades diferentes o en formas diferentes de forma independiente el uno del otro.

Cambio ("change")

Un cambio (o diff, o delta) representa una modificación específica de un documento bajo el control de versiones. La granularidad de la modificación que es considerada como un cambio varía entre los sistemas de control de versiones.

Desplegar ("checkout")

Es crear una copia de trabajo local desde el repositorio. Un usuario puede especificar una revisión en concreto u obtener la última. El término 'checkout' también se puede utilizar como un sustantivo para describir la copia de trabajo.

Confirmar ("commit")

Confirmar es escribir o mezclar los cambios realizados en la copia de trabajo del repositorio. Los términos 'commit' y 'checkin' también se pueden utilizar como sustantivos para describir la nueva revisión que se crea como resultado de confirmar.

Conflicto ("conflict")

Un conflicto se produce cuando diferentes partes realizan cambios en el mismo documento, y el sistema es incapaz de conciliar los cambios. Un usuario debe resolver el conflicto mediante la integración de los cambios, o mediante la selección de un cambio en favor del otro.

Cabeza ("head")

También a veces se llama tip (punta) y se refiere a la última confirmación, ya sea en el tronco ('trunk') o en una rama ('branch'). El tronco y cada rama tienen su propia cabeza, aunque HEAD se utiliza a veces libremente para referirse al tronco.

Tronco ("trunk")

La única línea de desarrollo que no es una rama (a veces también llamada línea base, línea principal o máster).

Fusionar, integrar, mezclar ("merge")

Una fusión o integración es una operación en la que se aplican dos tipos de cambios en un archivo o conjunto de archivos. Algunos escenarios de ejemplo son los siguientes:
Un usuario, trabajando en un conjunto de archivos, actualiza o sincroniza su copia de trabajo con los cambios realizados y confirmados, por otros usuarios, en el repositorio.

Un usuario intenta confirmar archivos que han sido actualizado por otros usuarios desde el último despliegue ('checkout'), y el software de control de versiones integra automáticamente los archivos (por lo general, después de preguntarle al usuario si se debe proceder con la integración automática, y en algunos casos sólo se hace si la fusión puede ser clara y razonablemente resuelta).

Un conjunto de archivos se bifurca, un problema que existía antes de la ramificación se trabaja en una nueva rama, y la solución se combina luego en la otra rama.

Se crea una rama, el código de los archivos es independiente editado, y la rama actualizada se incorpora más tarde en un único tronco unificado.

Flujo de Trabajo

Tu repositorio local está compuesto por tres "árboles" administrados por git.

El primero es tu Directorio de trabajo que contiene los archivos, el segundo es el Index que actúa como una zona intermedia, y el último es el HEAD que apunta al último commit realizado.



Comenzando con GIT

En este apartado solo indicaremos algunos comandos básicos dado que el Taller de GIT tendrá todos los detalles para poder efectuar lo necesario para utilizar esta herramienta.

Crear un repositorio local nuevo

En un nuevo directorio ejecutar para crear un nuevo repositorio local de git.

```
git init
```

Hacer checkout (traer código) a un repositorio

Crea una copia local del repositorio ejecutando

```
git clone /path/to/repository
```

Si utilizas un servidor remoto, ejecuta

```
git clone username@host:/path/to/repository
```

Add & Commit

Puedes registrar cambios (añadirlos al Index) usando

```
git add <filename>
```

```
git add .
```

(Note el punto luego de “add” en el segundo ejemplo, esto quiere decir Todos los Archivos y equivale al * en línea de comandos) O sea agregara todos los archivos al index.

Este es el primer paso en el flujo de trabajo básico. Cuando se agregan al index quiere decir que a partir de este momento están siendo “seguidos” por git y cualquier cambio a los mismos será registrado.

Para hacer commit a estos cambios o sea para enviarlos al repositorio local usa

```
git commit -m "Commit message"
```

Ahora el archivo está incluido en el HEAD, pero aún no en tu repositorio remoto.

Envío de cambios

Tus cambios están ahora en el HEAD de tu copia local. Para enviar estos cambios a tu repositorio remoto ejecuta

```
git push origin master
```

Reemplaza master por la rama a la que quieres enviar tus cambios.

Si no has **clonado** un repositorio ya existente y quieres **conectar** tu repositorio local a un repositorio remoto, usa

```
git remote add origin <server>
```

Ahora podrás subir tus cambios al repositorio remoto seleccionado.

Git Config

Es una herramienta de Git que permite configurar variables de configuración específicas de Git a nivel local, global o del sistema. Estas configuraciones determinan el comportamiento de Git en diversos aspectos.

1. Configuración de Usuario:

```
git config --global user.name "Tu Nombre" git config --global user.email "tu@email.com"
```

- **Descripción:** Establece el nombre y la dirección de correo electrónico asociados con tus commits. Estas configuraciones suelen ser globales y se aplican a todos los repositorios en tu máquina.

2. Configuración de Editor de Texto:

```
git config --global core.editor "nombre-del-editor"
```

- **Descripción:** Especifica el editor de texto que se utilizará para los mensajes de commit.

3. Configuración de Colores en la Interfaz de Usuario:

```
git config --global color.ui true
```

- **Descripción:** Habilita el color en la interfaz de usuario de Git para facilitar la lectura de la salida en la línea de comandos.

4. Configuración de Alias:

```
git config --global alias.co checkout
```

- **Descripción:** Crea alias para comandos Git. En este ejemplo, **git co** ahora es equivalente a **git checkout**.

5. Configuración de URL Remota:

```
git config --global url."https://".insteadOf git://
```

- **Descripción:** Cambia las URLs de clonación de los repositorios para que utilicen **https://** en lugar de **git://**.

6. Configuración de Ramas Predeterminadas:

```
git config --global init.defaultBranch main
```

- **Descripción:** Establece la rama predeterminada al inicializar un nuevo repositorio.

7. Configuración de Diversos Aspectos de Git:

```
git config --global pull.rebase true
```

- **Descripción:** Permite personalizar varios aspectos de Git, como el comportamiento de **pull** en este caso específico.

8. Configuración de Commit Dedicado (Commit con Mensaje Único):

```
git config --global commit.template ~/.gitmessage
```

- **Descripción:** Establece un archivo de plantilla para los mensajes de commit, promoviendo el uso de mensajes estructurados y descriptivos.

9. Consultar Configuraciones:

```
git config --list
```

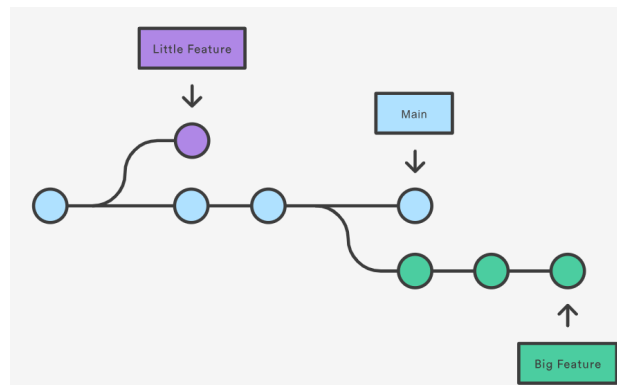
- **Descripción:** Muestra todas las configuraciones actuales a nivel de sistema, global y local.

10. Archivo de Configuración de Git:

- **Ubicación:**
 - A nivel de usuario global: **~/.gitconfig**
 - A nivel de repositorio local: **.git/config** en el directorio del repositorio.

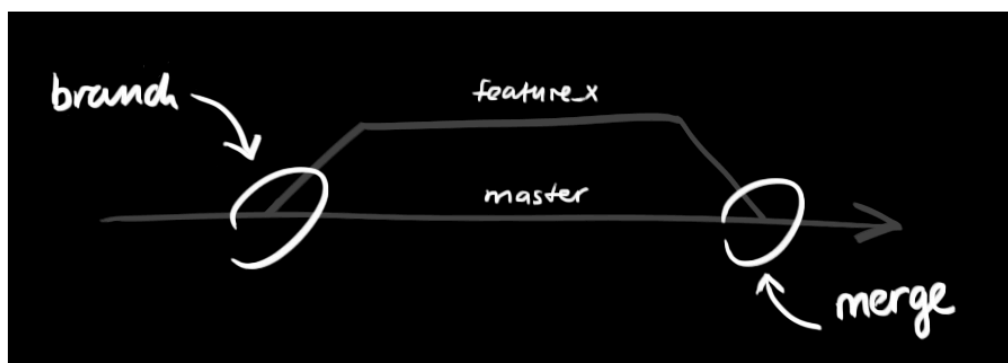
• 1.5.4. Branches o Ramas

Las ramas permiten trabajar fuera de la rama principal (Master – Main) y evitar así tener problemas de conflictos o dañar el software estable.



Cada desarrollador puede tener una o más ramas, y las buenas prácticas indican que cada rama debería tener una o más funcionalidades diferentes. De esta manera si una funcionalidad no se completa a tiempo en un Sprint o tiene errores que no permiten su puesta en producción se puede identificar y desactivar de manera más rápida o sin la necesidad de impactar al resto del desarrollo.

Las ramas son utilizadas para desarrollar funcionalidades aisladas unas de otras. La rama master es la rama "por defecto" cuando creas un repositorio. Crea nuevas ramas durante el desarrollo y fusionalas a la rama principal cuando termines.



La Manera en la que las Ramas deberían ser manejadas se basan en buenas prácticas y en las políticas o estrategias de Branch que se explicaran más abajo, siendo las más conocidas y aceptadas **Gitflow** y **GithubFlow**, debido a su simplicidad y capacidad de manejo de grandes equipos.

Branches Funcionamiento

Una rama representa una línea independiente de desarrollo. Las ramas sirven como una abstracción de los procesos de cambio, preparación y confirmación. Puedes concebirlas como una forma de solicitar un nuevo directorio de trabajo, un nuevo entorno de ensayo o un nuevo historial de proyecto. Las nuevas confirmaciones se registran en el historial de la rama actual, lo que crea una bifurcación en el historial del proyecto.

El comando `git branch` te permite crear, enumerar y eliminar ramas, así como cambiar su nombre. No te permite cambiar entre ramas o volver a unir un historial bifurcado. Por este motivo, `git branch` está estrechamente integrado con los comandos `git checkout` y `git merge`.

Opciones comunes

`git branch`

Enumera todas las ramas de tu repositorio. Es similar a `git branch --list`.

`git branch <branch>`

Crea una nueva rama llamada `<branch>`. Este comando **no** extrae la nueva rama.

`git branch -d <branch>`

Elimina la rama especificada. Esta es una operación segura, ya que Git evita que elimines la rama si tiene cambios que aún no se han fusionado.

`git branch -D <branch>`

Fuerza la eliminación de la rama especificada, incluso si tiene cambios sin fusionar. Este comando lo puedes usar si quieres eliminar de forma permanente todas las confirmaciones asociadas con una línea concreta de desarrollo.

`git branch -m <branch>`

Cambia el nombre de la rama actual a `<branch>`.

`git branch -a`

Enumera todas las ramas remotas.

Creación de ramas

Es importante comprender que las ramas son solo punteros a las confirmaciones. Cuando creas una rama, todo lo que Git tiene que hacer es crear un nuevo puntero, no modifica el repositorio de ninguna otra forma.

Si utilizas:

```
git branch crazy-experiment
```

Ten en cuenta que este comando solo **crea** la nueva rama. Para empezar a añadir confirmaciones utilizar los comandos estándar `git add` y `git commit`. Previamente deberías haber tenido un checkout de algún repositorio remoto.

Eliminación de ramas

Una vez que hayas terminado de trabajar en una rama y la hayas fusionado con el código base principal, puedes eliminar la rama sin perder ninguna historia:

```
git branch -d crazy-experiment
```

No obstante, si la rama no se ha fusionado, el comando anterior mostrará un mensaje de error:

```
error: The branch 'crazy experiment' is not fully merged. If you
are sure you want to drop it, run 'git branch -D
crazy-experiment'.
```

Esto te protege ante la pérdida de acceso a una línea de desarrollo completa. Si realmente quieres eliminar la rama (por ejemplo, si se trata de un experimento fallido), puedes usar el indicador `-D` (en mayúscula):

```
git branch -D crazy-experiment
```

Este comando elimina la rama independientemente de su estado y sin avisos previos, así que úsalo con cuidado.

Los comandos anteriores eliminarán una copia local de la rama. La rama seguirá existiendo en el repositorio remoto. Para eliminar una rama remota, ejecuta estos comandos.

```
git push origin --delete crazy-experiment
```

O también:

```
git push origin :crazy-experiment
```

Enviarán una señal de eliminación al repositorio de origen remoto que desencadena la eliminación de la rama remota `crazy-experiment`.

Ejemplos:

Para crear una nueva rama llamada "feature_x" y cambiarse a ella

```
git checkout -b feature_x
```

volver a la rama principal

```
git checkout master
```

y borrar la rama

```
git branch -d feature_x
```

Una rama nueva no estará disponible para los demás a menos que se suba (push) la rama al repositorio remoto

```
git push origin <branch>
```

≡ 1.6 Introducción a Automatización

● 1.6.1. Conceptos y Herramientas

Scripting y lenguajes de programación

Los lenguajes de scripting son lenguajes de programación que se utilizan principalmente para escribir scripts o programas pequeños que automatizan tareas específicas. Estos lenguajes están diseñados para ser fáciles de aprender y utilizar, y generalmente son interpretados en lugar de compilados, lo que significa que el código se ejecuta línea por línea por un intérprete en lugar de ser convertido previamente a código máquina.

1. **Automatización de tareas:** Los lenguajes de scripting se utilizan para escribir scripts que automatizan tareas repetitivas en el desarrollo y la operación de software, como la implementación de código, la gestión de servidores, la configuración de redes, la gestión de bases de datos, entre otros. Esto ayuda a reducir errores humanos y acelera el flujo de trabajo.
2. **Aprovisionamiento de infraestructura:** Los lenguajes de scripting como Python, Ruby o PowerShell se utilizan para definir y configurar la infraestructura de

manera programática. Herramientas como Terraform o Ansible permiten escribir scripts para aprovisionar y gestionar recursos en la nube y en entornos locales.

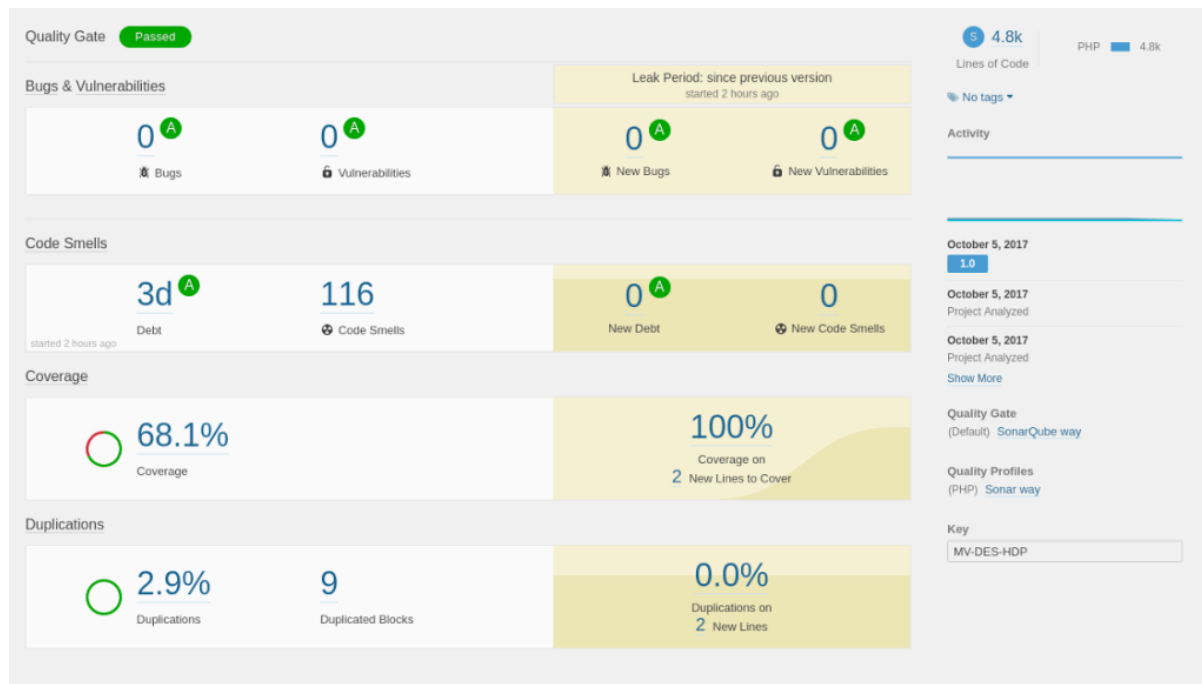
3. **Gestión de configuración:** Los scripts se utilizan para definir y mantener la configuración de servidores y aplicaciones, lo que permite mantener la consistencia y la reproducibilidad en diferentes entornos.
4. **Monitoreo y alertas:** Los lenguajes de scripting se utilizan para crear scripts que recopilan datos de monitoreo, generan alertas en caso de problemas y toman medidas correctivas automáticas cuando sea necesario.

Algunos ejemplos de lenguajes de scripting populares utilizados en DevOps incluyen Python, Bash, Ruby, PowerShell y Groovy, entre otros. La elección del lenguaje depende de los requisitos específicos del proyecto y las preferencias del equipo.

Los tres más populares son sin lugar a duda PowerShell, Bash y Groovy, debido a la cantidad de equipos que corren Windows y Linux por sobre el resto de los sistemas. Groovy es utilizado solo en integraciones con el sistema de CI/CD Jenkins.

SAST (Static Application Security Testing)

El análisis estático del código es el proceso de evaluar el software sin ejecutarlo. La idea de este análisis es que teniendo como entrada nuestro código fuente, podamos obtener información y métricas que nos permita mejorar la base de código detectando errores típicos de programación, bugs, code smells, etc. Esta herramienta nos hará sugerencias sobre qué partes del código son mejorables. Una herramienta muy conocida utilizada para tal fin es SonarQube. La veremos en este curso más adelante.



Unit Testing

Las pruebas unitarias, o "Unit Tests", se enfocan en evaluar individualmente las unidades más pequeñas de código, como funciones, métodos o clases, de manera aislada del resto del sistema. El objetivo principal de los Unit Tests es verificar que estas unidades de código funcionen correctamente de manera independiente antes de ser integradas en el sistema completo.

Características de los Unit Test

♦ Aislamiento:

Los Unit Tests se diseñan para probar una unidad específica de código sin depender de otras partes del sistema. Para lograr esto, es común utilizar técnicas como stubs (simuladores) y mocks (imitaciones) para aislar la unidad en prueba.

♦ Rapidez

Dado que se enfocan en una unidad pequeña de código, los Unit Tests suelen ser rápidos de ejecutar. Esto permite ejecutarlos de manera frecuente durante el proceso de desarrollo sin afectar significativamente la productividad.

♦ Automatización

Los Unit Tests son automatizados, lo que significa que se escriben en código y se ejecutan automáticamente mediante herramientas específicas. Esta automatización facilita la repetición de las pruebas y asegura que siempre se realicen de la misma manera.

♦ Detección temprana de errores

Al evaluar las unidades de código de forma individual, los Unit Tests ayudan a detectar errores y defectos en una etapa temprana del desarrollo. Esto reduce la posibilidad de que problemas se acumulen y se vuelvan más difíciles y costosos de corregir.

♦ Documentación viva

Los Unit Tests también pueden actuar como una forma de documentación, ya que describen cómo se supone que debe funcionar una unidad de código. Los nuevos miembros del equipo pueden entender la funcionalidad al leer los tests asociados.

♦ Facilitan el cambio

Al realizar modificaciones en una unidad de código, los Unit Test actúan como una red de seguridad. Si las pruebas siguen pasando después de los cambios, es más probable que la funcionalidad siga siendo correcta.

Ejemplo en Java utilizando JUnit:

```
import static org.junit.jupiter.api.Assertions.fail;

public class ExampleUnitTest {

    @org.junit.jupiter.api.Test
    public void testSomething() {
        // ...
        if (condicionEsperada) {
            // Si la condición se cumple, la prueba fallará en este punto
            fail("Esta prueba debería fallar aquí");
        }
        // ...
    }
}
```

En general, es preferible utilizar afirmaciones específicas para verificar el comportamiento esperado. El uso de `fail()` se reserva más para situaciones excepcionales en las que deseamos un control más preciso sobre cuándo y por qué falla una prueba.

Para ejecutar una prueba unitaria desde un script simplemente invoque el programa de test Unitario con parámetros. Por ejemplo, en Bash puedes ejecutar una prueba JUnit utilizando el comando `java` junto con el classpath y el nombre de la clase de prueba.

```
java -cp .:lib/junit-<version>.jar:lib/your-project-dependencies.jar org.junit.runner.JUnitCore com.example.TestClass
```

Frameworks para trabajo con Unit Tests

Existen numerosos frameworks para trabajar con pruebas unitarias en varios lenguajes de programación. Tenga en cuenta que la elección del framework correcto debe ser pensada de antemano antes de comenzar a escribir los mismos, dado que si cambiamos de biblioteca en medio de un desarrollo puede que debamos refactorizar o reescribir por completo todos los test.

JUnit (Java)

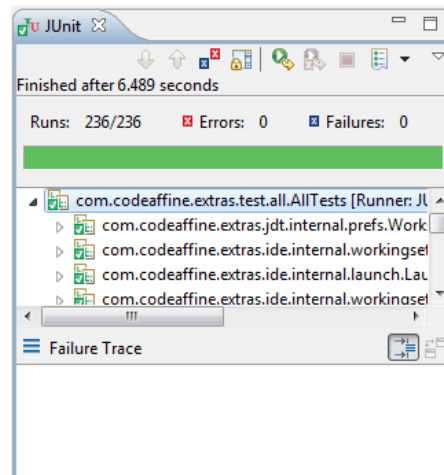
Licencia: Eclipse Public License—v 2.0

Repo URL: <https://github.com/junit-team/junit5>

Una de las bibliotecas más conocidas para pruebas unitarias en Java. Ofrece una amplia gama de funciones para definir, organizar y ejecutar pruebas.

Proporciona ejecutores de pruebas que ejecutan pruebas y aserciones para comprobar los resultados esperados. No es necesario verificar manualmente los resultados de las pruebas, ya que ejecuta las pruebas y verifica los resultados automáticamente.

Muestra los resultados de las pruebas de manera interactiva mediante una barra de progreso. Ofrece varios gráficos que representan el progreso de tus pruebas, mostrando un color verde cuando una prueba se ejecuta sin problemas y un color rojo cuando falla. También proporciona anotaciones para ayudarte a identificar los métodos de prueba.



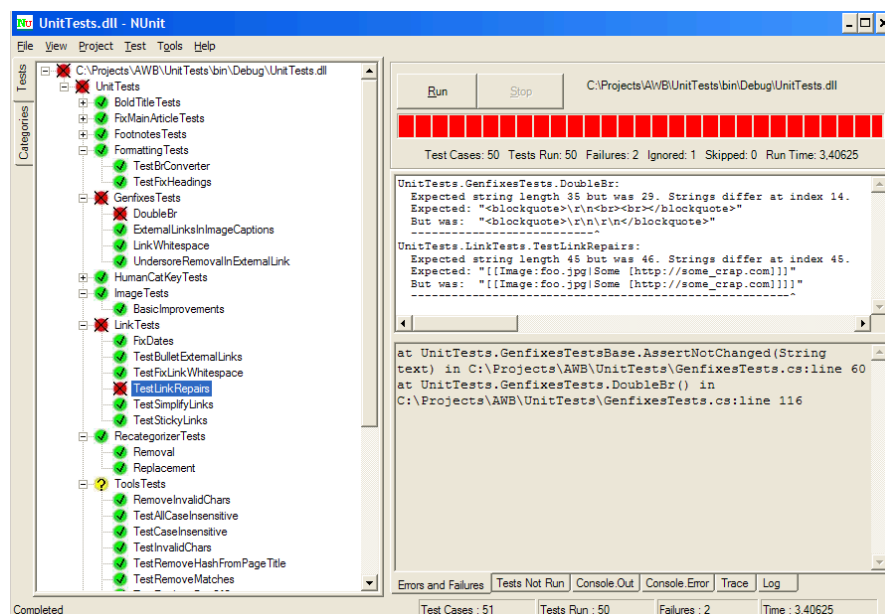
TestNG (Java): Una alternativa a JUnit en Java que ofrece características adicionales como la agrupación de pruebas y la ejecución paralela.

NUnit (C#)

Licencia: MIT License

Repo URL: <https://github.com/nunit/nunit>

Similar a JUnit, pero diseñado para C#. Proporciona un marco de trabajo sólido para escribir pruebas unitarias en aplicaciones .NET.



RSpec (Ruby)

Licencia: Mit License

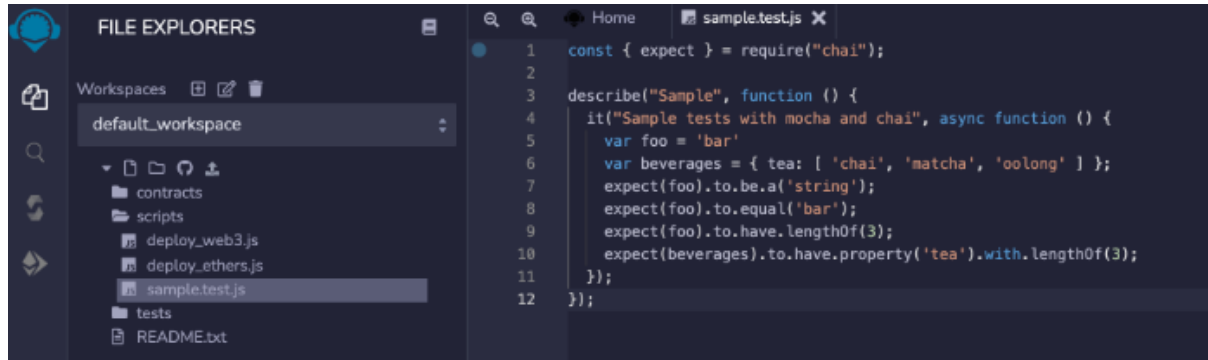
Un framework de pruebas unitarias popular en Ruby. Proporciona una sintaxis expresiva y legible para escribir pruebas.

Mocha (JavaScript)

Licencia: MIT License

Repo URL: <https://github.com/mochajs/mocha>

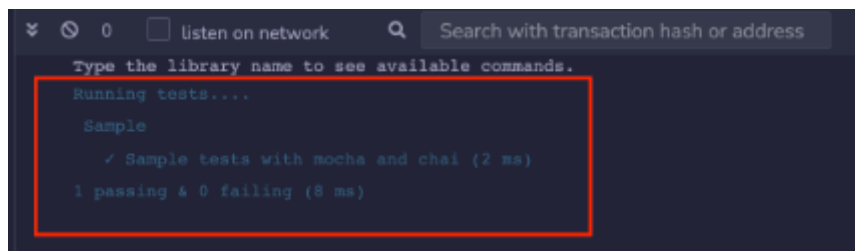
Una biblioteca de pruebas flexible para JavaScript. Puede utilizarse en entornos de navegador y Node.js.



```

1  const { expect } = require("chai");
2
3  describe("Sample", function () {
4    it("Sample tests with mocha and chai", async function () {
5      var foo = 'bar'
6      var beverages = { tea: [ 'chai', 'matcha', 'oolong' ] };
7      expect(foo).to.be.a('string');
8      expect(foo).to.equal('bar');
9      expect(foo).to.have.lengthOf(3);
10     expect(beverages).to.have.property('tea').with.lengthOf(3);
11   });
12 });

```



```

Type the library name to see available commands.
Running tests....
Sample
  ✓ Sample tests with mocha and chai (2 ms)
1 passing & 0 failing (8 ms)

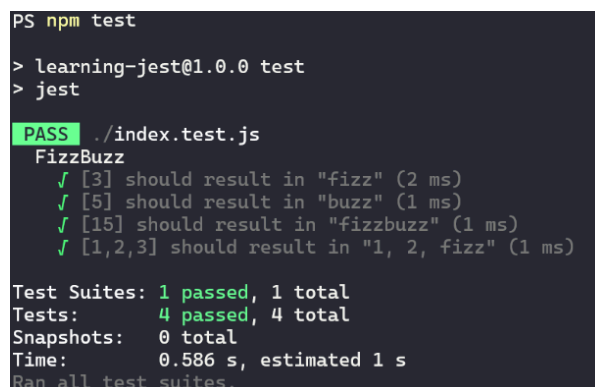
```

Jest (JavaScript)

License: MIT License

Repo URL: <https://github.com/facebook/jest>

Un framework moderno de pruebas para JavaScript, especialmente diseñado para trabajar con proyectos de React y otras bibliotecas populares.



```

PS npm test

> learning-jest@1.0.0 test
> jest

PASS ./index.test.js
  FizzBuzz
    ✓ [3] should result in "fizz" (2 ms)
    ✓ [5] should result in "buzz" (1 ms)
    ✓ [15] should result in "fizzbuzz" (1 ms)
    ✓ [1,2,3] should result in "1, 2, fizz" (1 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:  0 total
Time:        0.586 s, estimated 1 s
Ran all test suites.

```

Jasmine (Javascript)

License: MIT License

Repo URL: <https://github.com/jasmine/jasmine>

Jasmine es un framework que puede ejecutar en plataformas habilitadas para JavaScript sin entrometerse en el IDE o la aplicación. Este marco de código abierto proporciona una sintaxis fácil de leer y una biblioteca que incluye todos los componentes necesarios para probar su código.

Puede usar Jasmine en el navegador y también para Node, Ruby y Python. Es rápido, proporciona una API completa y sencilla, y le permite usar lenguaje natural para describir su prueba y los resultados esperados.

Existen muchísimos frameworks y no tiene sentido nombrarlos a todos, les recomiendo buscar el que mejor funcione para su lenguaje, ojo no siempre es el más popular.

QA Automation (Automatización de Pruebas de Calidad)

La automatización de pruebas de calidad es una práctica en el desarrollo de software que existe para garantizar que una aplicación funcione de manera correcta y cumpla con los requisitos definidos. Cucumber, Selenium y Robot Framework son herramientas populares, cada una con sus propias características y enfoques.



1. Cucumber: Cucumber

- **Enfoque:** Cucumber es una herramienta de automatización de pruebas que sigue el principio de BDD (Behavior-Driven Development). Permite escribir pruebas en un lenguaje natural legible por humanos, lo que facilita la colaboración entre los equipos de desarrollo y QA.
- **Lenguaje de Pruebas:** Las pruebas en Cucumber se escriben en lenguaje Gherkin, que utiliza palabras clave como "Given," "When," y "Then" para describir el comportamiento esperado de la aplicación.
- **Flexibilidad:** Cucumber es compatible con varios lenguajes de programación, incluyendo Java, Ruby y JavaScript, lo que facilita la integración con diferentes tecnologías.



2. Selenium:

- **Enfoque:** Selenium es una suite de herramientas de automatización de pruebas enfocada en la automatización de pruebas de aplicaciones web. Puede interactuar con un navegador web real para realizar acciones como hacer clic en botones, llenar formularios y verificar resultados.
- **Lenguaje de Pruebas:** Selenium no tiene su propio lenguaje de pruebas, por lo que generalmente se usa junto con lenguajes de programación como Java, Python o C# para escribir pruebas de manera programática.
- **Versatilidad:** Selenium es muy versátil y puede utilizarse para probar aplicaciones web en varios navegadores y sistemas operativos.

3. Robot Framework:

- **Enfoque:** Robot Framework es un marco de automatización de pruebas que se destaca por su facilidad de uso y legibilidad. Ofrece una sintaxis simple que facilita la creación de pruebas de aceptación y pruebas de regresión.
- **Lenguaje de Pruebas:** Utiliza una sintaxis de tabla con palabras clave que hacen que las pruebas sean legibles y expresivas. Además, es extensible y admite la creación de bibliotecas personalizadas en Python o Java.
- **Bibliotecas Integradas:** Robot Framework incluye bibliotecas integradas para la automatización de pruebas web, pruebas de API, pruebas de bases de datos y más.

DAST (Dynamic Application Security Testing)

Se traduce como Pruebas Dinámicas de Seguridad de Aplicaciones, es una técnica utilizada en el campo de la seguridad de aplicaciones informáticas. Su principal objetivo es identificar y evaluar las vulnerabilidades de seguridad en una aplicación web o una aplicación móvil en tiempo de ejecución. Los PenTest están abarcados como DAST.

Una herramienta que entra en esta categoría es OWASP Zap, pero dependiendo del departamento de Seguridad informática de su empresa podría utilizar un toolset de herramientas pago diferente. No enumeramos ninguna aquí debido a la cantidad que existen en el mercado.

Características:

- **Escaneo en Tiempo de Ejecución:** DAST realiza pruebas de seguridad durante el tiempo de ejecución de una aplicación. Esto significa que simula ataques reales contra la aplicación en funcionamiento para identificar vulnerabilidades y debilidades en el código y la configuración de seguridad.
- **No se Requiere Acceso al Código Fuente:** Una ventaja importante de DAST es que no requiere acceso al código fuente de la aplicación. Puede analizar aplicaciones de terceros o aplicaciones heredadas sin necesidad de modificar el código.
- **Identificación de Vulnerabilidades Comunes:** DAST se centra en la identificación de vulnerabilidades de seguridad comunes, como inyecciones de SQL, vulnerabilidades de Cross-Site Scripting (XSS), problemas de autenticación y autorización, entre otros.
- **Escenarios de Ataque Simulados:** DAST simula escenarios de ataque comunes, como intentos de inyección de código malicioso, ataques de fuerza bruta y manipulación de datos, para evaluar cómo la aplicación responde a estos ataques y si se pueden explotar vulnerabilidades.
- **Resultados de Pruebas en Tiempo Real:** DAST proporciona resultados en tiempo real mientras se realizan las pruebas, lo que permite a los equipos de seguridad abordar y solucionar problemas de inmediato.
- **Escaneo de Páginas Web y APIs:** DAST puede escanear tanto las páginas web visibles para los usuarios como las APIs que manejan solicitudes y respuestas detrás de escena.
- **Limitaciones:** A pesar de sus ventajas, DAST tiene algunas limitaciones. No es tan efectivo para identificar problemas de seguridad relacionados con la lógica de negocio de la aplicación o problemas en el código fuente que no se ejecutan durante el escaneo. También puede generar falsos positivos en ciertos casos.

(IAC) Infraestructura como Código

La Infraestructura como Código (IaC, por sus siglas en inglés) trata de definir y gestionar la infraestructura de sistemas de manera programática y automatizada, utilizando código en lugar de configuraciones manuales. Tal como lo indica, IaC permite que la infraestructura se defina, configure y mantenga como código.

Características:

- **Definición Declarativa:** En lugar de especificar pasos detallados sobre cómo configurar servidores y recursos de infraestructura, IaC utiliza una definición

declarativa que describe el estado deseado de la infraestructura. Esto permite que las herramientas de IaC se encarguen de las acciones necesarias para alcanzar ese estado.

- **Automatización:** IaC se basa en la automatización para desplegar y gestionar la infraestructura de manera eficiente y sin errores. Las herramientas de IaC pueden realizar tareas como aprovisionar servidores, configurar redes, instalar software y aplicar actualizaciones de manera automática.
- **Reproducibilidad:** Una de las ventajas clave de IaC es la capacidad de reproducir de manera exacta una infraestructura en múltiples entornos, como desarrollo, pruebas y producción. Esto garantiza que la infraestructura sea coherente y que las diferencias entre entornos se minimicen.
- **Control de Versiones:** El código de infraestructura se almacena en sistemas de control de versiones como Git, lo que permite el seguimiento de cambios, la colaboración en equipo y la reversión a versiones anteriores si es necesario.
- **Herramientas de IaC:** Hay varias herramientas populares de IaC disponibles, como Terraform, AWS CloudFormation, Azure Resource Manager, Ansible y Puppet. Estas herramientas proporcionan abstracciones para definir y gestionar la infraestructura a través de archivos de configuración o scripts.
- **Seguridad y Cumplimiento:** IaC facilita la aplicación de políticas de seguridad y cumplimiento de manera consistente en toda la infraestructura. Las configuraciones de seguridad pueden definirse como código y aplicarse automáticamente durante el despliegue.
- **Escalabilidad:** IaC es especialmente útil en entornos de nube, donde la infraestructura puede escalar según las necesidades. Las herramientas de IaC permiten definir reglas de escalabilidad automática.
- **Documentación Viva:** El código de infraestructura sirve como documentación viva de la infraestructura, lo que facilita la comprensión de cómo está configurada y cómo evoluciona con el tiempo.

(PaC) Política como código

Con la infraestructura y su configuración codificadas en la nube, las organizaciones pueden monitorear y garantizar la conformidad de forma dinámica y a escala. La infraestructura descrita por el código se puede supervisar, validar y reconfigurar de forma automática. De este modo, las organizaciones pueden controlar los cambios en los recursos con mayor facilidad y garantizar que se siguen las medidas de seguridad de

forma distribuida (p. ej. seguridad de la información o conformidad con PCI-DSS o HIPAA). Así, los equipos de la organización pueden avanzar a mayor velocidad, ya que los recursos no conformes se identifican de forma automática para su investigación o incluso se los dota de conformidad automáticamente.

≡ 1.7 Branching Strategies – Estrategias de Ramificación

● 1.7.1. Concepto

Una “estrategia de branching” es una serie de reglas que aplica un equipo de desarrollo de software cuando necesita escribir un código para incorporar una nueva funcionalidad o hacer una corrección, fusionarlo y enviarlo al repositorio donde se encuentra alojado el resto del código del software en uso, por ejemplo, en un sistema de control de versiones como Git.

Los desarrolladores de software que trabajan en equipo en la misma base de código deben compartir sus cambios entre sí. Pero *¿cómo pueden hacer esto de manera eficiente mientras evitan inconsistencias en el software?*

El objetivo de cualquier estrategia de branching es resolver precisamente ese problema, permitir que los equipos que trabajan juntos en la misma base de código fuente no afecten su código los unos a los otros.

Una estrategia de branching define como un equipo utiliza los branches para lograr un proceso de desarrollo concurrente, a través de un conjunto de reglas y convenciones que establecen:

- ¿Cuándo un desarrollador debe crear un branch?
- ¿De qué otro branch deben derivarse el nuevo branch?
- ¿Cuándo debe el desarrollador hacer merge?
- ¿Y a qué branch debería hacer el merge?

● 1.7.2. Motivo

Para los equipos que tienen cientos o miles de desarrolladores, si no tienen establecida una estrategia de branching, los procesos de branching y merging del código pueden resultar sumamente difíciles. Las fusiones incorrectas y las integraciones en etapas

tardías pueden consumir mucho tiempo de los desarrolladores, retrasando potencialmente la entrega del software y releases futuros.

Una estrategia de branching garantiza que todos los miembros del equipo sigan el mismo proceso para realizar cambios en código. La estrategia correcta mejora la colaboración, la eficiencia y la precisión en el proceso de entrega de software, mientras que la estrategia incorrecta (o ninguna estrategia) conduce a muchos errores y problemas de integración, que se traducen en horas de trabajo, mucho esfuerzo y sobre todo pérdida de tiempo y dinero.

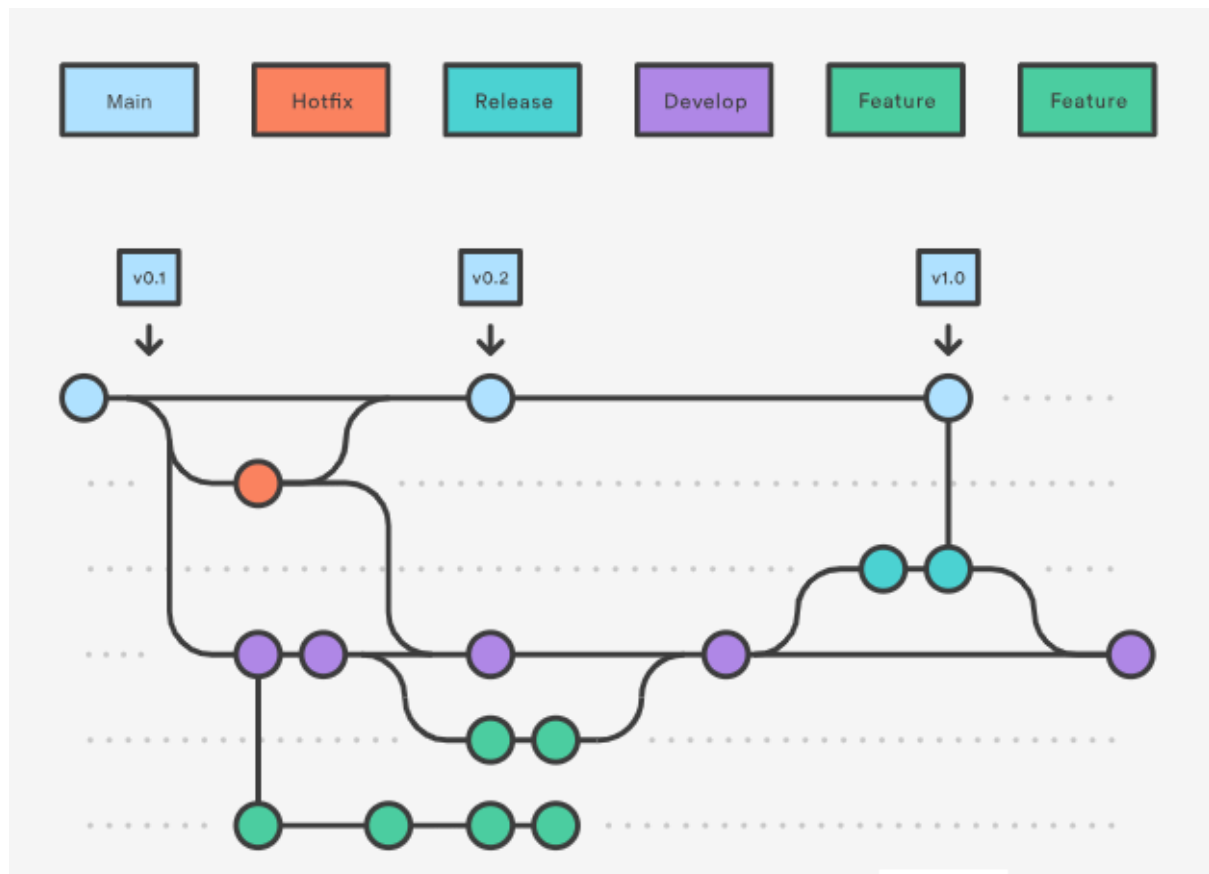
Para reducir estos problemas, una estrategia de branching debe buscar:

- Permitir a los desarrolladores optimizar su productividad.
- Habilitar el desarrollo en paralelo.
- Permitir un conjunto de releases estructurados y planificados.
- Proporcionar una ruta clara para promover los cambios del software al entorno de producción.
- Evolucionar y adaptarse a los cambios que se realizan a diario en el código.
- Admitir múltiples versiones de software.

● 1.7.3. GitFlow

La idea principal detrás de GitFlow es aislar el trabajo en diferentes tipos de branches, lo que le permite adaptarse muy bien al proceso colaborativo que necesita un equipo de desarrollo. GitFlow está basado principalmente en dos branches que tienen una vida infinita:

- **master:** contiene el código de producción.
- **develop:** contiene el código que ha finalizado desarrollo.



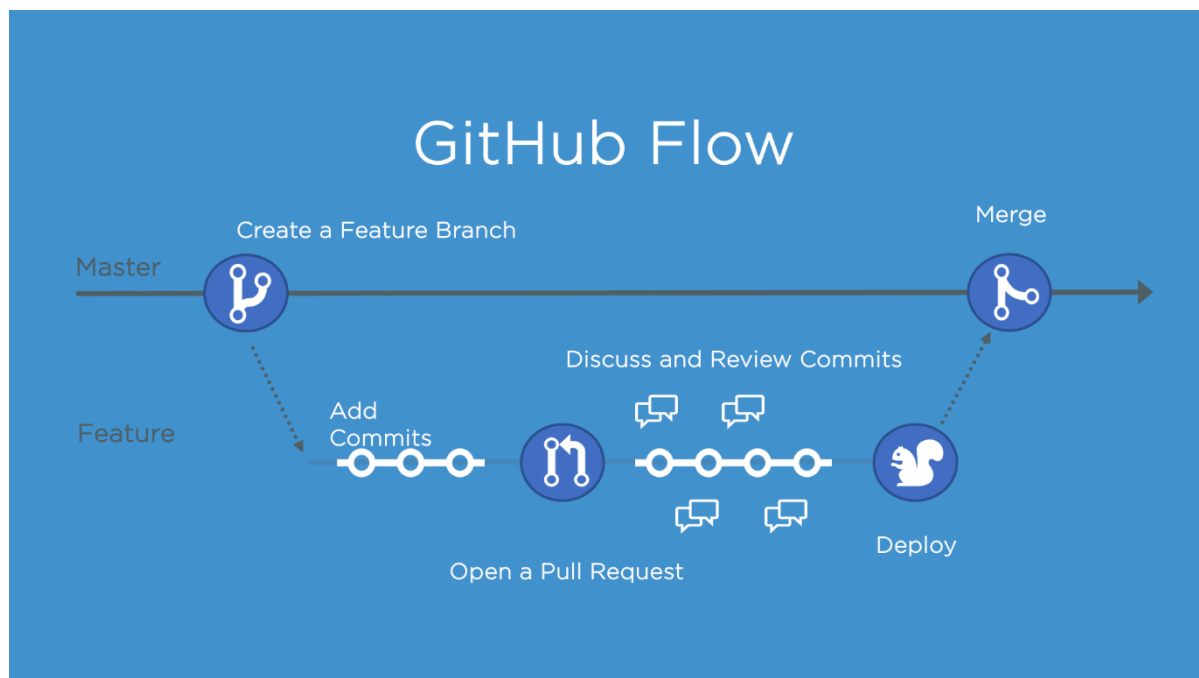
Adicional a estos branches principales, durante el desarrollo se crean otros branches de soporte que tienen una vida finita, es decir, existen mientras exista el desarrollo:

- **feature:** se crea a partir de *develop* cuando una nueva funcionalidad necesita ser desarrollada. Al finalizar el desarrollo se hace merge a *develop* nuevamente.
- **release:** se crea a partir de *develop* para preparar una nueva versión del código que debe ser liberada en producción. Al finalizar el desarrollo se hace merge a *develop* y a *master*.
- **hotfix:** se crea a partir de *master* cuando es necesario corregir un error detectado en producción de manera urgente. Al finalizar el desarrollo se hace merge a *develop* y a *master*.

• 1.7.4. GitHub Flow

Fue creado por GitHub y es conocido en la comunidad de desarrolladores como una alternativa simple y ligera a GitFlow. GitHub Flow se basa en un flujo de trabajo basado en branches que permite a equipos de desarrollo enfocarse principalmente en la entrega

continua. A diferencia de Git Flow, no existen los branches de “releases”, ya que está pensado para que la implementación en producción ocurra con frecuencia, incluso varias veces al día si es posible.



En esta estrategia de branching, en el repositorio tenemos dos tipos de branches:

- **main (o master):** el branch de código principal, es el que contiene el código que está listo para producción.
- **features:** los branches de funcionalidades que permiten el desarrollo en paralelo.

Cuando utilizas GitHub Flow, existen seis principios que debemos seguir para asegurarnos de mantener un buen código listo para producción en todo momento:

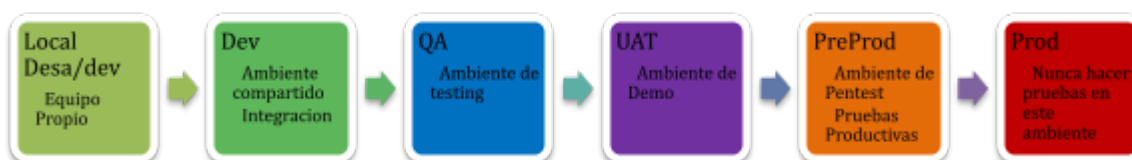
- El código que está en `_main` debe poder implementarse en producción en cualquier momento.
- Cuando se crean nuevos `_feature` branches, se deben crear con nombres descriptivos. Por ejemplo, **feature/add-new-account-type**.
- Primero hacer commit en local, para luego hacer push al repositorio remoto.
- Abrir **pull requests** para solicitar feedback o ayuda, antes de hacer merge en el branch principal.

- Hacer merge del código en el branch `_main`, solo después de haber recibido aprobación del pull request.
- Una vez hecho el merge del código en `_main`, debe implementarse de inmediato, mejor si es en un entorno de producción.

GitHub Flow es ideal cuando necesitas mantener una única versión del código de producción.

≡ 1.8 Ciclo de Vida Aplicativo

• 1.8.1. Ambientes del ciclo de vida aplicativo



Ambiente de Desarrollo (Development)

Este es el entorno donde los desarrolladores escriben y prueban el código de la aplicación. Es un entorno aislado donde se pueden realizar experimentos y pruebas sin afectar la versión en producción. Generalmente se recomienda utilizar un ambiente Local de desarrollo en su propia IDE y luego tener un entorno Desa de integración para Desarrollo, donde si bien uno puede probar sin afectar podría impactar el desarrollo de otro miembro del equipo que se encuentre efectuando pruebas en el mismo ambiente.

Ambiente de Pruebas (QA/Testing)

Una vez que el código se ha desarrollado en el entorno de desarrollo, se pasa al entorno de pruebas. Aquí, los equipos de pruebas realizan pruebas exhaustivas para identificar y corregir errores y garantizar que la aplicación funcione correctamente. Los entornos de prueba suelen estar más cerca de la producción que el entorno de desarrollo.

En este ambiente los equipos de QA (Quality Assurance / Aseguramiento de Calidad) se encargan de efectuar los Tests de regresión y aceptación correspondientes. También podríamos correr aquí los Tests de manera automatizada en el proceso de CI/CD en el papel de DevSecOps.

Ambiente de UAT (User Acceptance Testing)

Antes de implementar la aplicación en el entorno de producción, se realiza una fase final de pruebas en un entorno similar al de producción, pero sin usuarios finales. Esto se hace para asegurarse de que la aplicación esté lista para su lanzamiento y cumpla con las expectativas del usuario. Este es el ambiente que se utiliza comúnmente para Demos. Puede llegar a reemplazar al ambiente de Staging/Pre-Producción dependiendo de las necesidades del proyecto. Nota: Es una mala práctica hacer los Demos en el ambiente de QA o Desarrollo, debido a que impactará de manera negativa tanto a la calidad del software como al desarrollo de este.

Ambiente de Staging/Pre-Prod

En algunos casos es necesario hacer una salida a producción suave, que no impacte demasiado a los usuarios. Para tal fin se podría tener un escenario que esté conectado a los mismos servicios que producción, pero no sea de público conocimiento de los usuarios. De esta forma, se podría hacer una salida más aislada y progresiva, lo que se denomina Friends and Family (Amigos y Familia), y luego de que validamos que los cambios fueron correctos entonces podríamos promocionarlo a producción. También suelen ser utilizados para que los equipos de InfoSec puedan efectuar sus DAST sin afectar el ambiente productivo.

Ambiente de Producción

Este es el entorno en el que la aplicación está disponible para los usuarios finales. Aquí, se ejecuta y presta servicio a los usuarios reales. La estabilidad y el rendimiento son cruciales en este entorno.

● 1.8.2. Etapas del Ciclo de Vida Aplicativo

Etapas de Planificación

En esta etapa, los equipos de desarrollo y operaciones trabajan juntos para definir los objetivos del proyecto, los requisitos y las expectativas. Se crean planes de desarrollo y se establecen métricas para medir el éxito.

Etapas de Compilación / Build

Los desarrolladores escriben y prueban el código de la aplicación. Los ingenieros DevOps colaboran con los desarrolladores para asegurarse de que se sigan las mejores prácticas de automatización y que se integren herramientas de construcción y pruebas automatizadas.

Cada aplicación podría tener una manera diferente de compilarse o integrarse, es por esto por lo que es fundamental estar cerca de los desarrolladores, serán ellos quienes

nos darán los comandos que deberemos utilizar en nuestros pipelines para poder compilar las soluciones/proyectos/aplicaciones.

Etapas de Automated Testing

En esta etapa deberemos trabajar codo a codo con el equipo de QA para lograr ejecutar de manera adecuada los Tests automatizados que tengan generados, evitando así la pérdida de tiempo necesaria para realizar integraciones o pruebas de regresión manuales. Si bien es cierto que no puede automatizarse todo, se pueden lograr buenos objetivos si se trabaja en conjunto y se define que se automatizará y que no.

Recordemos que el equipo encargado de implementar DevSecOps no escribe Tests, al menos no desde la definición de separación de responsabilidades, simplemente automatiza su ejecución.

Etapas de Despliegue / Release

Durante el Release (desde el punto de vista de DevSecOps) se toman o bien los fuentes y se ejecuta una nueva instancia de compilación para obtener un binario, o bien se toman los binarios almacenados durante la etapa de Build y se despliegan a los ambientes requeridos.

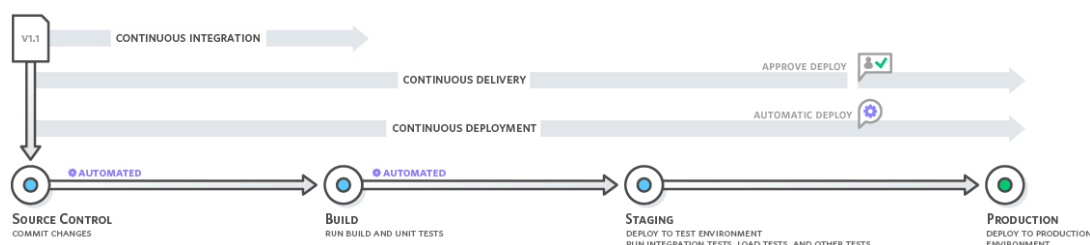
Es importante destacar que si bien se podrían generar las reléase notes de manera manual, también es posible efectuarlo de manera automatizada. Por ejemplo, mediante el uso de los commits de Git en el código fuente. De manera tal que DevSecOps pueda automatizar la generación de estas sin intervención manual, excepto por las aprobaciones de entrega correspondientes.

Etapas de Monitoreo

Las organizaciones monitorean métricas y registros para ver cómo el desempeño de las aplicaciones y la infraestructura afecta a la experiencia que el usuario final tiene de su producto. Cuando recopilan, categorizan y analizan los datos y registros generados por las aplicaciones y la infraestructura, las organizaciones pueden entender cómo los cambios y actualizaciones afectan a los usuarios, esto les aporta información sobre la causa raíz de los problemas o cambios inesperados. El monitoreo activo se vuelve cada vez más importante, ya que los servicios deben estar disponibles las 24 horas del día, los 7 días de la semana, a medida que la frecuencia de actualizaciones de las aplicaciones e infraestructura incrementa. La creación de alertas y el análisis en tiempo real de los datos también ayuda a las organizaciones a monitorear sus servicios de forma proactiva.

1.9 Ciclo de Vida Aplicativo

1.9.1. Integración Continua (CI)



La integración continua es una práctica de desarrollo de software mediante la cual los desarrolladores combinan los cambios en el código en un repositorio central de forma periódica, tras lo cual se ejecutan versiones y pruebas automáticas. La integración continua se refiere en su mayoría a la fase de creación o integración del proceso de publicación de software y conlleva un componente de automatización (p. ej., CI o servicio de versiones) y un componente cultural (p. ej., aprender a integrar con frecuencia). Los objetivos clave de la integración continua consisten en encontrar y arreglar errores con mayor rapidez, mejorar la calidad del software y reducir el tiempo que se tarda en validar y publicar nuevas actualizaciones de software.

¿Por qué es necesaria la integración continua?

Anteriormente, era común que los desarrolladores de un equipo trabajasen aislados durante un largo periodo de tiempo y solo intentasen combinar los cambios en la versión maestra una vez que habían completado el trabajo. Como consecuencia, la combinación de los cambios en el código resultaba difícil y ardua, además de dar lugar a la acumulación de errores durante mucho tiempo que no se corregían. Estos factores hacían que resultase más difícil proporcionar las actualizaciones a los clientes con rapidez.

¿En qué consiste la integración continua?

Con la integración continua, los desarrolladores envían los cambios de forma periódica a un repositorio compartido con un sistema de control de versiones como Git. Antes de cada envío, los desarrolladores pueden elegir ejecutar pruebas de unidad local en el código como medida de verificación adicional antes de la integración. Un servicio de integración continua crea y ejecuta automáticamente pruebas de unidad en los nuevos cambios realizados en el código para identificar inmediatamente cualquier error.

La integración continua se refiere a la fase de creación y pruebas de unidad del proceso de publicación de software. Cada revisión enviada activa automáticamente la creación y las pruebas.

Beneficios de la integración continua

- Mejorar la productividad de desarrollo

La integración continua mejora la productividad del equipo al liberar a los desarrolladores de las tareas manuales y fomentar comportamientos que ayudan a reducir la cantidad de errores y bugs enviados a los clientes.

- Encontrar y arreglar los errores con mayor rapidez

Gracias a la realización de pruebas más frecuentes, el equipo puede descubrir y arreglar los errores antes de que se conviertan en problemas más graves.

- Entregar las actualizaciones con mayor rapidez

La integración continua le permite a su equipo entregar actualizaciones a los clientes con mayor rapidez y frecuencia.

● 1.9.2. Entrega Continua y Despliegue Continuo (CD)

La entrega continua es una práctica de desarrollo de software mediante la cual se preparan automáticamente los cambios en el código y se entregan a la fase de producción. Fundamental para el desarrollo de aplicaciones modernas, la entrega continua amplía la integración continua al implementar todos los cambios en el código en un entorno de pruebas o de producción después de la fase de compilación. Cuando la entrega continua se implementa de manera adecuada, los desarrolladores dispondrán siempre de un artefacto listo para su implementación que se ha sometido a un proceso de pruebas estandarizado.

La entrega continua permite a los desarrolladores automatizar las pruebas más allá de las pruebas de unidades, por lo que pueden verificar actualizaciones en las aplicaciones en varias dimensiones antes de enviarlas a los clientes. Las pruebas pueden incluir pruebas de la UI, de carga, de integración, de fiabilidad de la API, etc. De este modo, los desarrolladores pueden validar las actualizaciones de forma más exhaustiva y descubrir problemas por anticipado. Con la nube, resulta sencillo y rentable automatizar la creación y replicación de varios entornos de pruebas, algo que anteriormente era complicado en las instalaciones.

Beneficios de la entrega continua

- Automatización del proceso de publicación de software

La entrega continua permite al equipo crear, probar y preparar automáticamente los cambios en el código para su envío a producción, con lo que se mejora la eficacia y rapidez de la entrega de software.

- Mejorar la productividad de desarrollo

Estas prácticas mejoran la productividad del equipo al liberar a los desarrolladores de las tareas manuales y fomentar comportamientos que ayudan a reducir la cantidad de errores y bugs enviados a los clientes.

- Permite encontrar y arreglar los errores con mayor rapidez

Su equipo puede descubrir y arreglar los errores antes de que se conviertan en problemas más graves gracias a las pruebas más frecuentes y exhaustivas. La entrega continua le permite realizar tipos de pruebas adicionales en el código con facilidad, ya que todo el proceso se ha automatizado.

- Permite entregar las actualizaciones con mayor rapidez

La entrega continua le permite a su equipo entregar actualizaciones a los clientes con mayor rapidez y frecuencia. Cuando se la entrega continua se implementa de manera adecuada, dispondrá siempre de un artefacto listo para su implementación que se ha sometido a un proceso de pruebas estandarizado.

● 1.9.3. Entrega continua vs Despliegue continuo

Con la entrega continua, todos los cambios en el código se crean, se prueban y se envían a un entorno de almacenamiento o pruebas de no producción. Pueden efectuarse varias pruebas al mismo tiempo antes de la implementación en producción. La diferencia entre la entrega y la implementación continuas es la diferencia de aprobación manual para actualizar la producción. Con la implementación continua, la producción tiene lugar de manera automática, sin aprobación explícita.

La entrega continua automatiza todo el proceso de publicación de software. Cada revisión efectuada activa un proceso automatizado que crea, prueba y almacena la

actualización. La decisión definitiva de implementarla en un entorno de producción en vivo la toma el desarrollador.

● 1.9.4. Pipelines

Un Pipeline de CI / CD es el componente más importante del desarrollo de software automatizado. Si bien el término se ha utilizado para describir muchos aspectos diferentes de la informática, en gran parte de la industria de DevOps, usamos “Pipelines” para ilustrar las amplias aplicaciones de comportamientos y procesos involucrados en la integración continua (CI).

CI es una estrategia de desarrollo de software que aumenta la velocidad de desarrollo al tiempo que garantiza que la calidad del código implementado no se vea comprometida. Mediante el uso de herramientas de CI, los desarrolladores envían código continuamente en pequeños incrementos, a veces varias veces al día, que luego se crea y prueba automáticamente antes de fusionarse con el repositorio compartido. Las canalizaciones (Pipelines) de entrega (Pipelines) de software modernas pueden crear, probar e implementar aplicaciones según las necesidades de su negocio.

Un Pipeline es el conjunto completo de procesos que se ejecutan cuando se activa el trabajo en sus proyectos. Las canalizaciones (Pipelines) abarcan sus flujos de trabajo, que coordinan sus trabajos, y todo esto se define en el archivo de configuración de su proyecto.

La integración continua automatiza la construcción y prueba de su software. La implementación continua es una extensión de esta automatización y permite que su software se implemente después de cada confirmación de código que pasa su conjunto de pruebas. Los equipos de desarrollo más exitosos implementan su software con frecuencia.

Los Pipelines pueden variar según la herramienta que se utilice y también así el lenguaje utilizado para su generación o descripción de código.

El lenguaje comúnmente aceptado es YAML, aunque también se utiliza Groovy que es una versión más simple de JAVA.

Ejemplo de un Pipeline Jenkins:

```
pipeline {
  agent {
    label 'ecs'
  }
  stages {
    stage('Build') {
      steps {
```

```

sh 'git clone --single-branch --branch
simplified-for-ci
https://github.com/tkgregory/spring-boot-api-example.git'
sh 'cd spring-boot-api-example && ./gradlew build
docker --info'
}
}
}
}
}
```

En este ejemplo tendremos un YAML que a través de llaves nos indica que objeto está incluido dentro de cual otro. Esto es Groovy, que trabaja solo en Jenkins, pero la mayoría de los entornos de trabajo para Pipelines escriben código similar.

Como veremos aquí pipeline tiene una llave que incluye un agent (El agente que ejecutará las tareas que necesitamos)

Dentro del agente tendremos etapas (stages) y dentro de los stages tendremos los stage individuales, como pueden ser las etapas de Compilar, testear, desplegar, etc.

Dentro de la etapa build tenemos los pasos (Steps) que deberá ejecutar Jenkins para lograr que nuestro stage se ejecute correctamente.

Es muy común que los steps o pasos individuales involucren llamar scripts individuales o línea de comandos CLI para que efectúe dichos pasos. Por ejemplo, con un SH llamamos a la línea de comandos que ejecute `git clone` para traer el código del repositorio y luego a través de invocación de `gradlew` ejecuta un build de Docker.

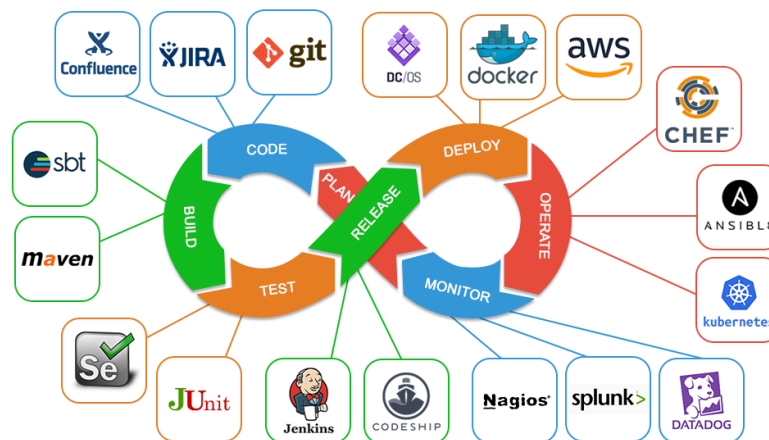
No vamos a explicar todos los comandos que groovy puede presentar debido a que, como indicamos antes, esto puede variar si utiliza CircleCI, AWS CodeBuild o BitBucket. Pero todas las herramientas comparten la premisa de que un Pipeline engloba acciones que pueden ejecutarse para lograr automatizar los procesos.

Veremos ahora el mismo ejemplo, pero con CodeBuild de AWS:

```
version: 0.2
phases:
  build:
    commands:
      - './gradlew build docker --info'
```

Como se puede observar, aquí la estructura es simplificada y no usa llaves. Solo se indican los atributos con indentación y dos puntos. Las palabras clave son phases, donde podríamos tener build, deploy, test, o lo que se nos ocurra. Y luego dentro de “commands” los comandos a ejecutar

• 1.9.5. Herramientas de CI/CD



Jenkins

La interfaz de Jenkins muestra el siguiente estado:

S	W	Name ↓	Last Success	Last Failure
		Test	4.7 sec - #5	1 hr 11 min - #2

Icon: [S](#) [M](#) [L](#)

Build Queue
No builds in the queue.

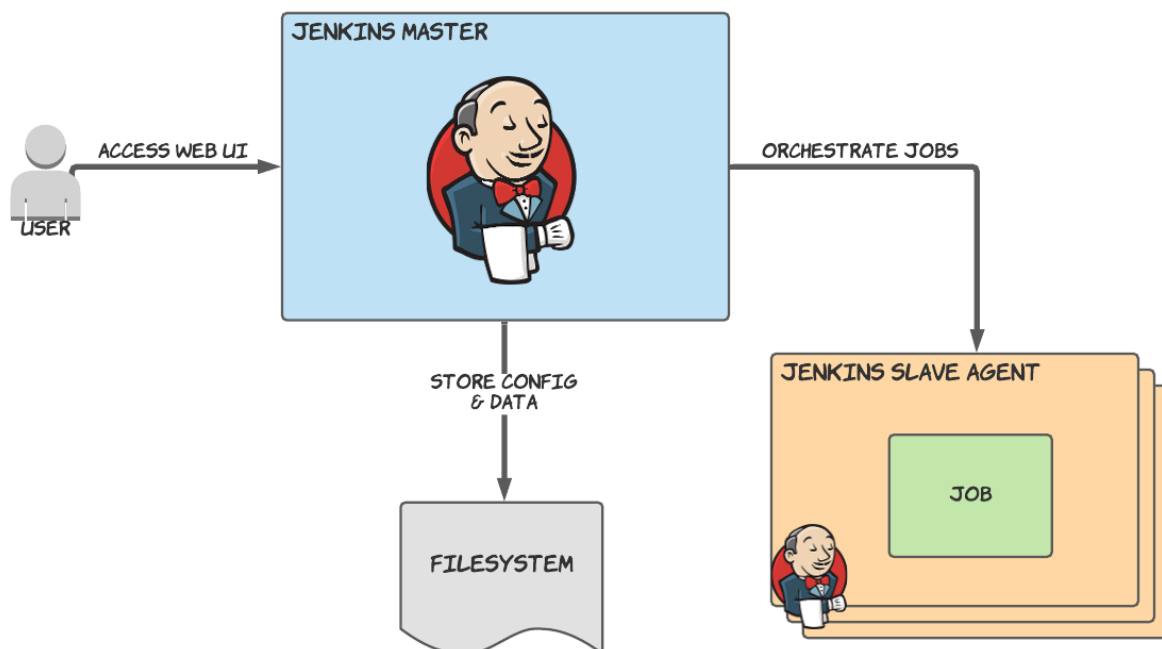
Build Executor Status

- 1 Idle
- 2 Idle

Jenkins es una herramienta basada en Java que permite gestionar nuestros procesos automatizados de Compilación y Despliegue de manera sencilla. Es ampliamente adoptado debido a su facilidad de uso y su simplicidad de ejecución.

Se basa en el uso de máquinas de compilación y despliegue (agentes) que pueden o bien ser instaladas físicamente en un equipo para tal fin o bien escalar de manera dinámica mediante el uso de Agentes efímeros con Docker. (Veremos esto más adelante en nuestro curso)

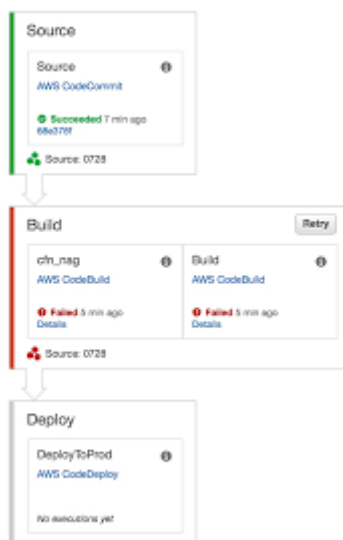
El servicio principal o también denominado Maestro u Orquestador (Master) tendrá una interfaz gráfica que los equipos podrán utilizar para crear sus Pipelines, ejecutar las integraciones o visualizar el estado de estas.



Permite integrar fácilmente otras plataformas o herramientas mediante el uso de Plugins y Webhooks. Estos últimos responden a eventos específicos, como puede ser la subida de código fuente a un repositorio, y en función de esto dispara triggers que pueden ejecutar diferentes tareas.

Veremos más adelante en este curso esta herramienta en detalle.

AWS (CodePipeline y CodeBuild)



AWS CodePipeline y AWS CodeBuild son dos servicios de Amazon Web Services (AWS) que forman parte de la suite de herramientas de desarrollo y entrega continua (CI/CD). Estas herramientas ayudan a los equipos de desarrollo y operaciones a automatizar y orquestar el proceso de desarrollo de software, desde la construcción y prueba hasta la implementación en producción. Aquí hay una descripción más detallada de cada uno de estos servicios:

1. AWS CodePipeline:

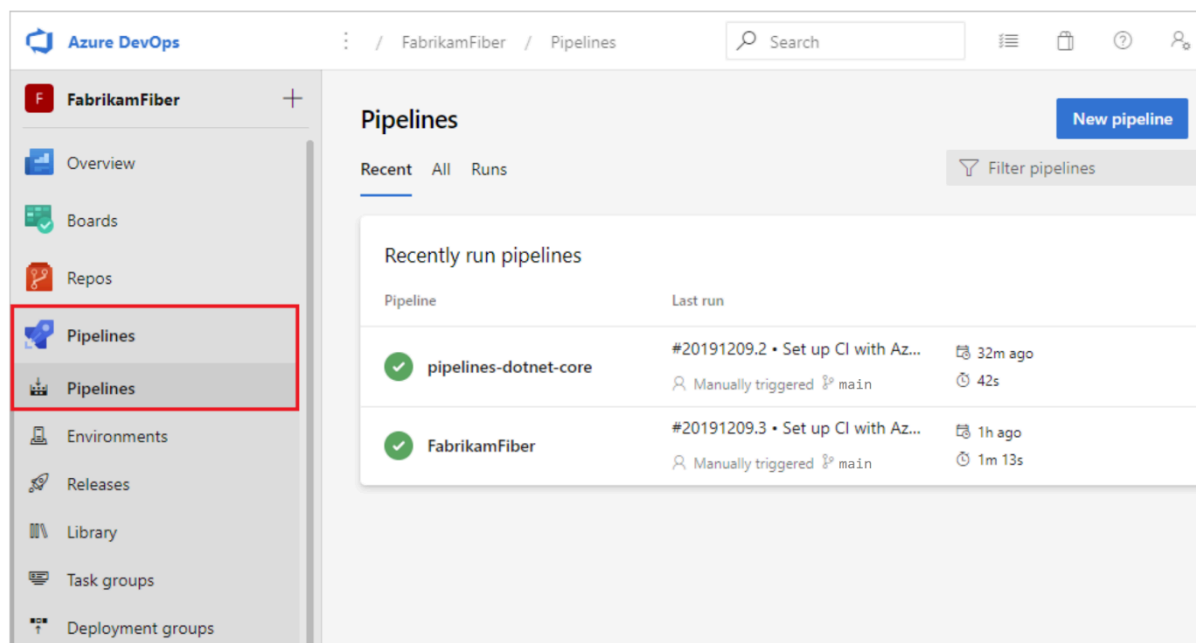
- **Descripción:** AWS CodePipeline es un servicio de orquestación de entrega continua que le permite crear, automatizar y gestionar pipelines (canales) de entrega continua. Estos pipelines representan el flujo de trabajo de desarrollo de software, que incluye la construcción, las pruebas y la implementación.
- **Características principales:**
 - **Automatización:** CodePipeline permite automatizar todo el proceso de entrega continua, desde la obtención del código fuente hasta la implementación en entornos de producción.
 - **Integración con Herramientas:** Se integra fácilmente con otros servicios de AWS, así como con herramientas de terceros, como GitHub, Jenkins, Docker, etc.
 - **Personalización:** Puede personalizar y configurar su pipeline según sus necesidades específicas, definiendo las etapas y las acciones que deben ejecutarse.

- **Notificaciones:** Proporciona notificaciones en tiempo real a través de Amazon SNS y otras integraciones para mantener informados a los equipos sobre el estado de la implementación.

2. AWS CodeBuild:

- **Descripción:** AWS CodeBuild es un servicio de compilación completamente administrado que compila el código fuente, ejecuta pruebas y genera artefactos de compilación, como archivos binarios o paquetes, de manera automatizada. Es una parte fundamental en el proceso de entrega continua.
- **Características principales:**
 - **Compilaciones Personalizadas:** CodeBuild permite definir entornos de compilación personalizados con configuraciones específicas para diferentes proyectos.
 - **Integración:** Se integra estrechamente con CodePipeline y otros servicios de AWS, lo que permite construir automáticamente aplicaciones cada vez que se desencadena una acción en el pipeline.
 - **Soporte Multiplataforma:** Admite múltiples lenguajes de programación y plataformas, lo que facilita la construcción de una variedad de aplicaciones.
 - **Escalabilidad:** CodeBuild es escalable y se adapta a las necesidades de su proyecto, ya sea una pequeña aplicación o un proyecto empresarial de gran envergadura.
 - **Registro y Depuración:** Proporciona registros detallados de compilación para ayudar a identificar problemas y depurar el proceso de construcción.

Azure DevOps

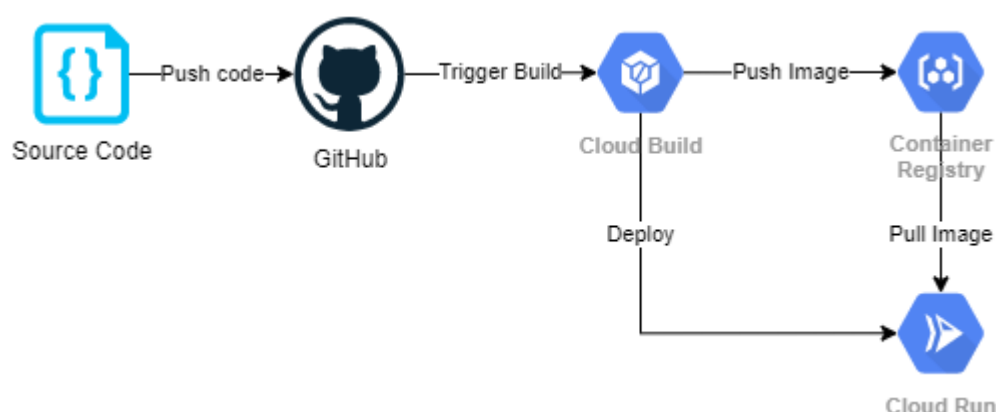


Azure DevOps es un conjunto integral de herramientas y servicios de desarrollo y entrega continua (CI/CD) proporcionado por Microsoft Azure. Está diseñado para ayudar a los equipos de desarrollo de software a planificar, desarrollar, probar, implementar y monitorear aplicaciones de manera eficiente y colaborativa. Azure DevOps es una plataforma basada en la nube que ofrece una variedad de características y servicios esenciales para respaldar el ciclo de vida de desarrollo de aplicaciones. Aquí hay una descripción general de sus componentes clave:

1. **Azure Boards:** Esta herramienta se utiliza para la planificación y gestión de proyectos ágiles. Permite la creación de tableros Kanban y tableros de seguimiento de problemas (backlogs), así como la administración de sprints y la asignación de tareas a los miembros del equipo.
2. **Azure Repos:** Azure Repos es un sistema de control de versiones que admite repositorios Git y Team Foundation Version Control (TFVC). Ofrece características de seguimiento de cambios, revisión de código y colaboración en el desarrollo de software.
3. **Azure Pipelines:** Azure Pipelines es una plataforma de CI/CD que permite la automatización de la construcción, las pruebas y la implementación de aplicaciones. Puede crear pipelines de construcción y despliegue personalizados para aplicaciones en una variedad de lenguajes y plataformas.

4. **Azure Test Plans:** Esta herramienta permite la planificación, el seguimiento y la administración de pruebas en todo el ciclo de vida del proyecto. Puede crear casos de prueba, automatizar pruebas y realizar un seguimiento de los resultados de las pruebas.
5. **Azure Artifacts:** Azure Artifacts proporciona un repositorio de paquetes para administrar dependencias de software. Puede alojar y compartir paquetes de NuGet, npm y Maven, entre otros, para facilitar la gestión de dependencias en sus proyectos.
6. **Azure DevTest Labs:** Este servicio facilita la creación de entornos de desarrollo y pruebas bajo demanda. Puede configurar rápidamente máquinas virtuales y otros recursos para pruebas y desarrollo.
7. **Azure Monitor y Azure Application Insights:** Estos servicios se utilizan para monitorear aplicaciones en producción y recopilar datos sobre su rendimiento y uso. Ayudan a identificar problemas y optimizar el rendimiento de las aplicaciones en tiempo real.
8. **Azure DevOps Server:** Aunque Azure DevOps suele utilizarse como un servicio en la nube, también existe una versión local llamada Azure DevOps Server (anteriormente conocida como Team Foundation Server). Esta opción permite implementar Azure DevOps en su propia infraestructura si se prefiere un enfoque local o se necesita cumplir con requisitos de seguridad específicos.

Google (GCP Cloud Build y Cloud Pipeline)



Google Cloud Platform (GCP) ofrece servicios y herramientas que permiten a los equipos de desarrollo y operaciones crear flujos de trabajo de construcción (build) y entrega continua (CI/CD) para implementar aplicaciones en la nube de Google. Dos de los

servicios clave para lograr esto son Cloud Build y Cloud Build Pipeline. Aquí tienes una descripción general de ambos:

1. Cloud Build:

- **Descripción:** Cloud Build es un servicio de Google Cloud que permite la automatización de la construcción, las pruebas y el empaquetado de aplicaciones. Se integra estrechamente con otros servicios de GCP, como Google Cloud Source Repositories y Google Container Registry, así como con repositorios de código fuente externos como GitHub y Bitbucket.
- **Características:**
 - **Automatización de construcción:** Puedes configurar y personalizar tareas de construcción y pruebas en archivos de configuración (por ejemplo, archivos de configuración de Cloud Build YAML).
 - **Compatibilidad con contenedores:** Cloud Build se utiliza comúnmente para construir imágenes de contenedor de Docker, lo que facilita la implementación de aplicaciones en contenedores en Google Kubernetes Engine (GKE) u otros entornos de contenedor.
 - **Integración con notificaciones:** Puedes configurar notificaciones para recibir alertas sobre el estado de las compilaciones y los resultados de las pruebas.
 - **Escalabilidad y paralelismo:** Cloud Build es escalable y admite ejecuciones en paralelo, lo que lo hace adecuado para proyectos de cualquier tamaño.

2. Cloud Build Pipeline:

- **Descripción:** Cloud Build Pipeline es una función de Cloud Build que te permite crear flujos de trabajo de entrega continua (CI/CD) personalizados y automatizados. Puedes utilizar Cloud Build Pipeline para definir una serie de pasos y tareas que se ejecutan en secuencia para construir, probar y desplegar tu aplicación.
- **Características:**
 - **Automatización de CI/CD:** Puedes crear pipelines que automatizan completamente el proceso de entrega continua, desde la construcción y las pruebas hasta la implementación en los entornos de producción.

- **Personalización:** Tienes flexibilidad para definir y personalizar cada paso del pipeline según las necesidades de tu aplicación.
- **Integración con servicios de GCP:** Puedes integrar fácilmente tu pipeline con otros servicios de Google Cloud, como GKE, Cloud Functions y Firebase, para implementar aplicaciones en la nube de Google.
- **Gestión de versiones y ramas:** Puedes configurar pipelines que se desencadenan automáticamente en respuesta a cambios en repositorios de código fuente y ramas específicas.

≡ Resumen

En este módulo exploramos una variedad de temas relacionados con DevOps e Infraestructura.

Comenzamos definiendo DevOps y exploramos su importancia. Describimos las principales responsabilidades del rol de DevOps, incluyendo las habilidades esenciales requeridas.

Además, abordamos el concepto de la nube, discutiendo las diferencias entre la nube pública y privada, así como los modelos de servicio como IaaS, PaaS y SaaS. También presentamos ejemplos de proveedores de nube.

Otra área clave es el control de versiones, donde definimos conceptos y sistemas relevantes.

Finalmente, exploramos conceptos y herramientas relacionados con el ciclo de vida de las aplicaciones, incluyendo la integración y la entrega continua (CI/CD).

≡ Referencias

Libros

1. "The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win" de Gene Kim, Kevin Behr, y George Spafford.
2. "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" de Jez Humble y David Farley.
3. "Site Reliability Engineering: How Google Runs Production Systems" de Niall Richard Murphy, Betsy Beyer, Chris Jones, y Jennifer Petoff.
4. "The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations" de Gene Kim, Patrick Debois, John Willis, y Jez Humble.
5. "Infrastructure as Code: Managing Servers in the Cloud" de Kief Morris.
6. "DevOps for Dummies" de Emily Freeman, Ian Buchanan, y Jeffrey Scheaffer.

Papers

1. "Continuous Integration" de Martin Fowler (<https://martinfowler.com/articles/continuousIntegration.html>).
2. "The Twelve-Factor App" (<https://12factor.net/>).
3. "Cattle, Not Pets: The Evolution of DevOps" de Randy Bias (<https://www.slideshare.net/randybias/cattle-not-pets-the-evolution-of-devops>).
4. "Infrastructure as Code: Dynamic Systems for the Cloud Age" de Kief Morris (<https://www.oreilly.com/library/view/infrastructure-as-code/9781491924339/>).

5. "Continuous Delivery vs. Continuous Deployment vs. Continuous Integration" de Atlassian (<https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd>).

¡Muchas Gracias!

Nos vemos en el siguiente módulo 💪

