qualoom®

aws

¿Conoces esta filosofía de trabajo?

En esta entrega hablaremos sobre las diferencias entre la metodología DevOps y otras metodologías ágiles, y sobre las actividades principales que se pueden realizar con ella.

1/ Qué es DevOps

El **término DevOps proviene de las palabras *development* (desarrollo) y *operations* (operaciones)**. La cultura DevOps, es una filosofía de trabajo que busca acercar o unificar dentro de una compañía el desarrollo o construcción de software (*Dev*), y la administración de sistemas (*Ops*).

Algunas de las actividades que realizan estos dos equipos de trabajo son:

» **Desarrollo:** Análisis de requisitos del software, diseño arquitectónico, codificación, pruebas, integración, pruebas de aceptación.

» **Operaciones:** Instalación, actualización, migración, monitorización, administración de configuración, soporte.

Ambos equipos utilizan un conjunto de prácticas y herramientas que buscan **la reducción del tiempo entre la introducción de un cambio en el código del sistema y la puesta en producción de ese cambio**, al mismo tiempo que se garantiza la calidad del sistema.

Habitualmente los equipos de desarrollo y operaciones han estado separados en las empresas de creación de software, ya que normalmente desempeñan funciones muy diferentes. En los últimos años se ha visto la necesidad de acercar ambos grupos debido a dos factores:

» La **utilización de metodologías ágiles** para la creación de software. Estas metodologías se basan en iteraciones cortas, donde son liberadas nuevas funcionalidades del software en un plazo reducido de tiempo. Para implementar está nueva forma de trabajar es necesario mejorar la colaboración entre los desarrolladores y el equipo de sistemas, para que el proceso transcurra de la manera más fluida posible.

» La **popularización del uso de plataformas de cloud computing como Amazon Web Services o Microsoft Azure e infraestructuras de contenedores avanzadas con orquestadores como Kubernetes, RedHat OpenShift o Amazon ECS**, que permiten que las operaciones y el desarrollo sean actividades cada vez más próximas.

En este contexto surge DevOps, supervisando todo el proceso de construcción de software y automatizando numerosas tareas relacionadas con la integración, pruebas y despliegue de las aplicaciones.

Con DevOps **se trata de buscar la intersección entre desarrollo y operaciones**, teniendo en cuenta la calidad del producto,. Como vemos en la figura, DevOps no es una metodología, ni un conjunto de actividades sino un enfoque cultural que adoptará cada empresa dependiendo de sus necesidades.

Figura 1. Representación esquemática del concepto DevOps.

Fuente: Bello (2017)

Hay una estrecha relación entre DevOps y las metodologías ágiles. Al aproximar los equipos de desarrollo y operaciones, DevOps mejora el desarrollo ágil. Como se ha indicado anteriormente las metodologías ágiles se basan en actualizaciones frecuentes de software para mejorar las funcionalidades del producto final, esto provoca que sea necesario **entregas continuas de código.**

Para poder realizar entregas continuas, como las pruebas, el despliegue, o el propio desarrollo, **se debe automatizar en la medida de lo posible**, todo el proceso que va desde la codificación hasta el despliegue, pasando por la ejecución de pruebas de validación, integración, seguridad y calidad. Por tanto, las compañías han de adaptarse a este nuevo mecanismo de desarrollo y operación.

Todos estos cambios provocados con la adopción de metodologías ágiles, ha han variado la manera en que las organizaciones desarrollan y entregan el software al cliente.

Como vemos en la figura, **DevOps supone una evolución frente a los principios ágiles**. Mientras que estos destacan la importancia de la colaboración entre desarrolladores y clientes, **DevOps acentúa la colaboración entre desarrolladores y administradores de sistemas**. Este acercamiento entre ambos ámbitos también permite obtener información de los usuarios a partir del uso que estos hacen del producto en funcionamiento, de manera más fácil y asequible.

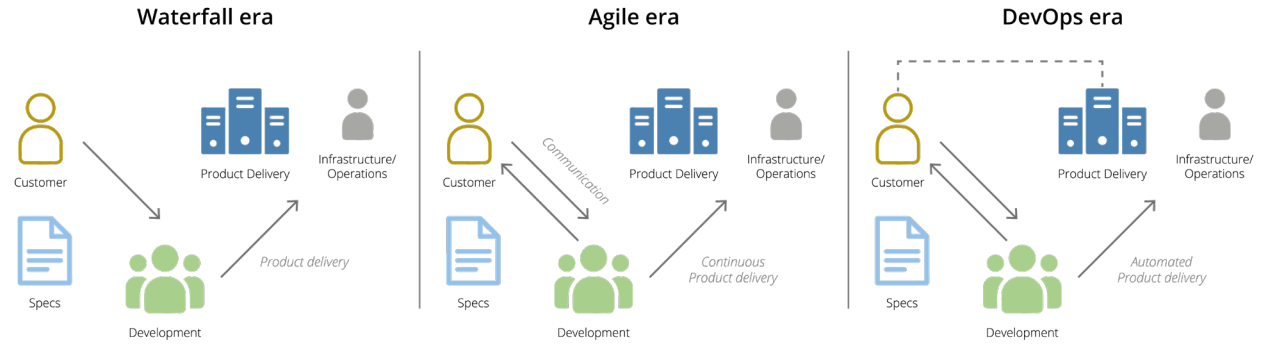

Figura 2. Evolución del modelo «Waterfall» al modelo «DevOps».

Fuente: Lwakatare (2017)

2/ Qué no es DevOps

DevOps **está enfocado a acercar el mundo de los desarrolladores y los administradores de sistemas**, con un conjunto de mejores prácticas para la producción de aplicaciones en la nube, orientadas a la integración continua y la liberación frecuente del código. Pero también es importante saber distinguir lo que no es:

1. **No es un puesto de trabajo, o un rol específico.** En algunas ocasiones podemos observar ofertas de trabajo del tipo "responsable DevOps", pero como hemos podido ver, se trata más bien de una cultura que abarca al conjunto de miembros del equipo, y no a un perfil específico diferenciado del resto.

2. **No es una categoría de herramientas.** DevOps no se ocupa tanto de herramientas específicas, como de la manera de organizar el trabajo, que permite un desarrollo y despliegue continuo del producto software.

3. **No es sinónimo de integración continua.** El hecho de que un proyecto realice integraciones continuas no significa que exista una cultura DevOps implantada en la organización.

4. **No es simplemente un término de moda.** DevOps da una respuesta a la necesidad actual de adaptarse a los cambios del entorno de manera temprana y manteniendo un nivel de calidad.

3/ Actividades principales

DevOps es fundamentalmente **una forma de abordar el proceso de creación de software que intenta unificar las actividades de desarrollo con las de operación y mantenimiento del sistema**, pasando por la involucración del equipo de pruebas, que garantiza la calidad del software entregado. Para que todo esto funcione correctamente es necesario aplicar al menos siete prácticas principales:

3.1/ Gestión de configuración del software

La gestión de configuración del software *—Configuration Management (CM)—* **se encarga del control de versiones del código fuente, de la gestión y el despliegue de esas versiones a producción**. Esta práctica resalta la importancia de mantener un repositorio de código fuente organizado, que permita la **automatización de los despliegues** a partir del mismo, persiguiendo **el objetivo de maximizar la productividad del equipo y minimizar los errores**.

Para controlar las diferentes versiones de código existen una serie de flujos de trabajo, que se apoyan en los sistemas de control de versiones, como Git.

Los siguientes son los modelos más habituales:

» **Centralized workflow (flujo centralizado).** Existe un repositorio de código central en el que existe una rama de desarrollo denominada "master". Cualquier cambio se realiza directamente sobre ella. Todos los desarrolladores mantienen una copia local del repositorio y, cuando mezclan los cambios realizados, deben resolver a menudo posibles conflictos de manera manual.

La figura muestra un repositorio con una única rama, que va acumulando todos los cambios realizados.

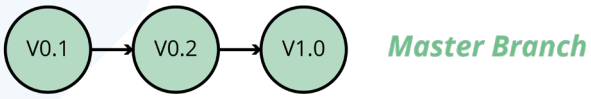

Figura 3. Gestión de versiones del código fuente con flujo de trabajo centralizado.

Fuente: Ott et al. (2017, p. 14)

» **Feature workflow (flujo por características).** En este modelo se crean ramas independientes y dedicadas para el desarrollo de las nuevas características que se van incluyendo en el producto. Esto permite que los desarrolladores trabajen de manera independiente sin interferir con el desarrollo del resto, y sin la necesidad de realizar los cambios de manera frecuente. Este modelo sirve como base más sólida para otras prácticas, como el despliegue continuo.

La idea se representa en la figura 4 donde, como vemos, además de la rama "master" se crean ramas específicas por cada una de las nuevas características. Todas las ramas de tipo "feature" se generan a partir de «master», y cuando la funcionalidad está terminada, se vuelcan los cambios realizados nuevamente sobre master.

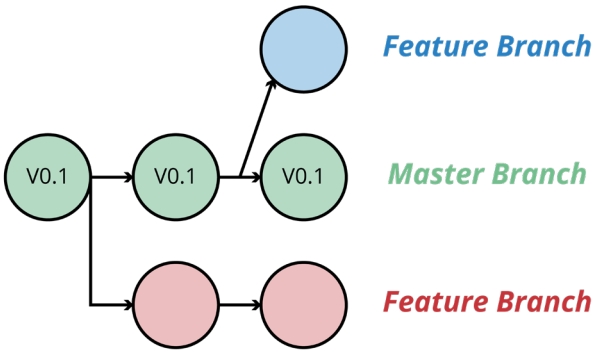

Figura 4. Gestión de versiones del código fuente con flujo de trabajo basado en características (features).

Fuente: Ott et al. (2017, p. 15)

» **Gitflow workflow (flujo Gitflow).** Es un modelo más avanzado que los anteriores, empleado por equipos experimentados. Combina el modelo centralizado y el de características, junto con otra serie de ramas específicas. En este flujo encontramos las ramas "master" y "develop" como principales, mientras que las ramas de tipo "feature", "release" y "hotfix" sirven de soporte al proceso de desarrollo, y se crean en función de las necesidades. Estas últimas tienen un tiempo de vida limitado, y se eliminan cuando dejen de cumplir su función.

- **Master branch (rama master):** esta rama principal contiene la historia de versiones oficiales del software. El código se mezcla en esta rama solo cuando se va a liberar una nueva release, y cada una de ellas aparece marcada en el repositorio con una etiqueta. Además, en ella solo se mezcla código procedente de una rama "release" (que se crean específicamente cuando se va a comenzar el lanzamiento de la nueva release), o de una rama "hotfix" (que se crea de manera temporal para hacer revisiones menores de una release).

- **Develop branch (rama de desarrollo):** es la rama en la que se produce de manera general el proceso de desarrollo, y donde se van integrando las diferentes características desarrolladas en las ramas temporales de tipo "feature". Desde esta rama se realiza el despliegue de integración continua en los servidores de desarrollo.

- **Feature branch (rama de características):** cada desarrollador crea una de estas ramas para implementar una nueva característica. A diferencia del modelo de flujo por características, las ramas de características se crean siempre a partir de la rama "develop", y cuando se termina la codificación se mezcla el código nuevamente con la rama de desarrollo. Como vemos, "develop" es la rama principal de trabajo y de ahí su nombre.

- **Release branch (rama de lanzamiento):** estas ramas se crean cuando se decide lanzar una nueva release, y actúa como paso intermedio entre la rama "develop" y la rama "master". Una vez creada una rama "release" a partir de "develop", solo se admite aplicar sobre ella correcciones de errores. Cuando se decide que el código en la rama "release" está terminado, se mezcla con "master" y se define una etiqueta que marca la nueva versión de la aplicación que será liberada.

- **Hotfix branch (rama de revisiones o ajustes):** se trata de ramas temporales, y son las únicas que se crean directamente a partir de "master". Se utilizan para aplicar correcciones que deben ser pasados rápidamente a producción, arreglando errores encontradas en el código de la versión actual. Dan lugar a revisiones de la versión que se encuentra en producción. Cuando se arregla el problema sobre la rama "hotfix", el código se mezcla tanto con "master" como con "develop", para que los desarrolladores continúen trabajando sobre la versión más actualizada del código que se encuentra en producción.

Todas las ramas del modelo Gitflow y sus interacciones aparecen representadas en la siguiente figura.

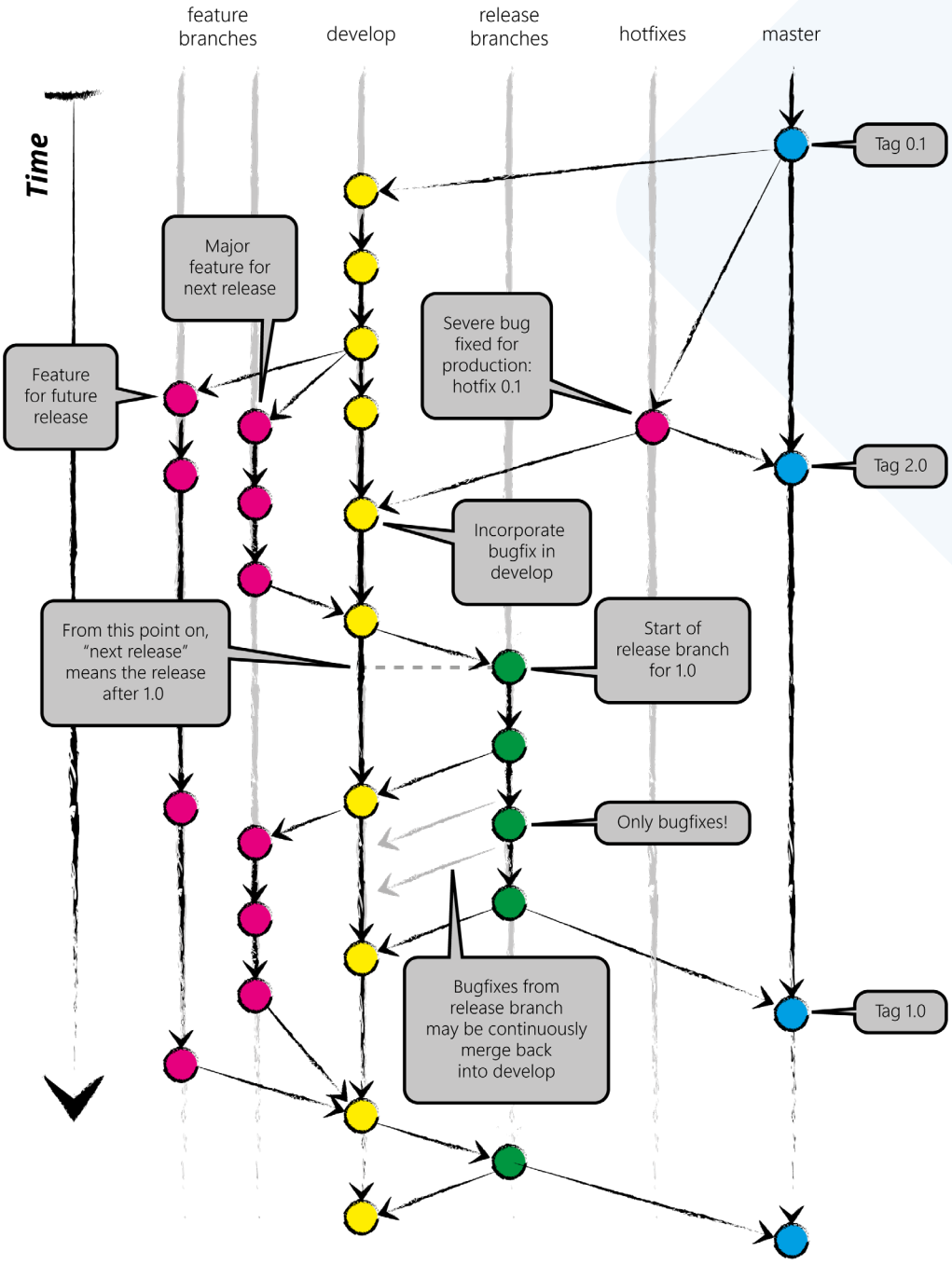

Figura 5. Gestión de versiones del código fuente con el modelo Gitflow.

Fuente: Driessen (2010)

3.2. Integración continua

La integración continua *—Continuous Integration (CI)—* es el proceso que **permite la integración del nuevo código generado de manera individual por los desarrolladores en un repositorio común**.

Lo ideal es realizar esta integración con la mayor frecuencia posible. Esta actividad sirve como base para otras, como la entrega y el despliegue continuos, que veremos más adelante.

La integración continua es un prerrequisito para poder conseguir el despliegue continuo, pero debe trabajar en combinación con otras prácticas, como las pruebas automatizadas, para garantizar que el proceso en su conjunto se realiza con las adecuadas garantías de calidad.

3.3. Pruebas automatizadas

Otra de las prácticas DevOps es la automatización de las pruebas *—Automated Testing (AT)—*. Se trata de **adelantar las pruebas todo lo posible dentro de cada ciclo de desarrollo**, para detectar posibles errores de manera temprana.

A continuación, se indican algunas de las pruebas que se realizan dependiendo del entorno en el que estemos trabajando:

» **Entorno local.** En este entorno podemos encontrar una máquina local o virtual perteneciente al programador. Es responsabilidad de cada desarrollador corregir los errores que aquí se produzcan. Aquí encontramos las siguientes pruebas básicas:

- **Lint y análisis estático.**
- **Pruebas unitarias.**
- **Pruebas de integración simulada.**

» **Entorno de integración continua.** Cuando el código ha sido verificado y aprobado de manera local por el programador, se integra en el repositorio común. Este repositorio se almacena en una máquina diferente a los equipos de desarrollo, y allí se repiten las pruebas unitarias y las pruebas de integración simulada. De esta manera se garantiza que el código en su conjunto, tras las modificaciones, siga funcionando correctamente.

» **Entorno de desarrollo.** En este caso se trata de asegurar que el código está preparado para pasar al entorno de pruebas. En el caso de que falle alguna prueba, el programador correspondientedeberá resolver el error. En este punto se repiten las pruebas unitarias y las pruebas de integración simulada, y se agregan las siguientes:

- **Pruebas de interfaz de usuario (UI).**
- **Pruebas de accesibilidad.**

» **Entorno de pruebas.** Una vez que se toma la decisión de negocio de pasar el producto a producción, comienzan las pruebas que permiten garantizar que el sistema en su conjunto funcionará correctamente en el entorno de producción. Aquí se integran todos los componentes, y se repiten las pruebas unitarias. Las pruebas de integración permiten validar que el comportamiento global es el correcto. Además, se incluyen las siguientes pruebas:

- **Código de infraestructura.**
- **Pruebas de seguridad.**
- **Pruebas de conformidad con estándares.**
- **Pruebas de resiliencia.**
- **Pruebas de rendimiento.**

» **Entorno de producción.** En esta etapa hay una serie de pruebas recomendadas, que están muy relacionadas con la práctica de monitorización continua que veremos más adelante.

- **Pruebas de posproducción.**
- **Pruebas de resiliencia y rendimiento.**
- **Pruebas sintéticas.**

3.4. Infraestructura como código

La infraestructura como código *—Infrastructure as Code (IaC)—* **trabaja con los recursos físicos de infraestructura**. Se trata de aprovisionar y configurar recursos de la plataforma a través de las API.

La implementación de esta práctica puede adoptar dos enfoques:

» **Modelo declarativo:** El código describe cuál es la configuración requerida de la infraestructura, y no los pasos necesarios para llegar a ella. Herramientas como "Puppet" permiten describir el estado deseado del servidor, sin entrar en detalles sobre los pasos necesarios de configuración para obtenerlo.

La ventaja de este modelo es una descripción más clara y estructurada de la configuración final, a costa de una menor flexibilidad en el control de esta configuración.

» **Modelo imperativo:** El código indica de manera explícita los pasos necesarios que deben ser ejecutados para conseguir una determinada configuración del servidor.

La ventaja de este modelo es un control total sobre las acciones de configuración, a costa de una mayor complejidad de los scripts que deben generarse.

3.5. Entregas continuas

En la entrega continua cualquier cambio realizado puede ser apto para subirlo a producción tan pronto como las pruebas automatizadas lo validen. Ello implica la toma de dos decisiones:

1. Una **decisión técnica**, que aprueba el paso al entorno de integración.

2. Una **decisión de negocio**, que acepta los cambios realizados y permite proseguir a los entornos de producción.

DevOps elimina esta última barrera que limita los tiempos de entrega de nuevas versiones y da el poder al Product Owner para decidir en qué momento pasar a producción una nueva historia de usuario, aunque el resto de las historias de la iteración no estén completas.

Para que las entregas continuas sean posibles, el equipo ha debido alcanzar una gran madurez sólo con la aplicación de las cuatro prácticas anteriores (CM, CI, AT, e IaC).

3.6. Despliegue continuo

En el despliegue continuo —Continuous Deployment (CD)— **los cambios se suben a producción en cuando se hayan verificado y validado las pruebas del código**, en vez de ser una decisión procedente de negocio como en las entregas continuas.

El objetivo principal de cualquier empresa eficaz es el despliegue continuo.

3.7. Monitorización continua

En la monitorización continua —*Continuous Monitoring (CM)*— **el equipo de desarrollo participa en el proceso de monitorización de la aplicación en producción**, labor que tradicionalmente realizaban los administradores de sistemas, con el objetivo de reducir el tiempo entre la posible detección de un problema y su corrección.

Las labores de monitorización se definen desde el comienzo del proyecto e implica al equipo de desarrollo.

Podemos identificar varios tipos de monitorización:

» **Monitorización de la infraestructura:** Se trata de obtener datos sobre el uso de los recursos de computación, almacenamiento y redes, midiendo el nivel de utilización y detectando posibles problemas. Ejemplo de ello son Prometheus, Collectd y AWS CloudWatch.

» **Monitorización del rendimiento de aplicaciones:** Consiste en detectar posibles cuellos de botella en el funcionamiento de nuestros servicios y aplicaciones. Como ejemplos tenemos AppDynamics, New Relic, Microsoft Application Insights o Amazon X-Ray.

» **Monitorización de logs:** Se recopila información histórica sobre los servicios ofrecidos para poder analizarla. Splunk, Fluentd y Elastic Stack, son ejemplos de este ámbito.

» **Monitorización de seguridad:** Durante el desarrollo del proyecto se fomenta la transparencia y el flujo de información, de manera que todos los miembros del equipo tengan acceso permanente a las características del producto y del proceso.

Si te interesa descubre más sobre nuestros servicios DevOps

Descubrir más

O ponte en contacto con nosotros directamente

Contactar