**DevOps**

# (Developers + Operations)

DevOps es una metodología o enfoque que busca **unificar el desarrollo de software (Dev) y las operaciones de TI (Ops)**

En lugar de tener equipos de desarrollo y operaciones trabajando de manera aislada y secuencial, el enfoque DevOps **promueve la colaboración y la comunicación constante entre estos dos grupos**. El **objetivo principal de DevOps es acelerar la entrega de software al mercado y mejorar la calidad del mismo, a través de la automatización de tareas repetitivas**, como la compilación, pruebas, implementación y monitoreo, utilizando herramientas y tecnologías adecuadas. Al adoptar esta metodología, las organizaciones pueden lograr despliegues más rápidos y frecuentes, reducir los riesgos asociados con los cambios y responder de manera más ágil a los requisitos cambiantes del mercado y de los usuarios

**Qué son los Developers y que son las Operations**

Los **Developers (desarrolladores) son los programadores**

**---------------------------------------------------------------------------------------------------------------------------------------------**

Las **Operations (operaciones)** engloban a los **departamentos que mantienen el funcionamiento de de la infraestructura de la IT o TI (Tecnología de la Información)**, siendo estos:

\*Administradores de sistemas: los cuales trabajan con los servidores y aplicaciones

\*Administradores de red: mantienen toda la parte de “networking”, redes, routers, etc

\*Atención al Cliente/usuario: tienen contacto con los usuarios cuando hay problemas para ayudar en la resolución de los mismos

**Objetivos y Ventajas de la Metodología DevOps**

**Desventajas del desarrollo tradicional de software**

En el desarrollo tradicional de software, no es común que ambos equipos (desarrollo y operaciones) trabajen de usualmente de forma coordinada

También se suele compilar (refiriéndonos a una “release” o versión completa de nuestro producto) y desplegar muy poco, de forma que se programa como loco y cuando se está por obtener la versión completa del producto se pasa a compilarlo, y es ahí cuando se topan con todo tipo de errores y problemas de integración

**Ventajas de la metodología DevOps**

\***Mejora la frecuencia y calidad del Deploy (despliegue)**

Para eso utilizamos el “continuous delivery” (entrega continua) y el “continuous deployment” (despliegue continuo), lo que básicamente es **compilar y testear nuestro código continuamente cada vez que se realiza un cambio**

De esta forma **captamos los errores de forma temprana y tenemos siempre listas versiones entregables de nuestro software**

**\*Captura y corrige rápidamente los errores**

La **monitorización, testeo continuo y corrección nos permiten capturar y resolver rápidamente los errores para que estos tengan el menor impacto posible en nuestro producto**

**\*Realizar correcciones en mucho menos tiempo**

Debido a la monitorización y testeo continuo podemos realizar las correcciones en mucho menos tiempo, ya que encontraremos con más facilidad el origen de los fallos

**\*Mejora el “time to market” (tiempo de salida al mercado)**

Esto lo logramos al **detectar los errores de forma rápida y debido a eso trabajar con productos funcionales de forma temprana**

**Flujo de Trabajo DevOps**

Una **problemática bastante común es la falta del testeo previo en entornos que simulen las condiciones de la demanda que tendrá el producto** (por ejemplo testear con datos de menor tamaño, con versiones de bases de datos diferentes, cantidad de usuarios menor, estructura de red diferente, etc)

**Infraestructura**

Para poder **desplegar y testear de manera eficiente** debemos tener una buena infraestructura, replicando los entornos de producción y desarrollo y **probar el producto bajo demanda**

Además la infraestructura debe ser fácil de sustituir

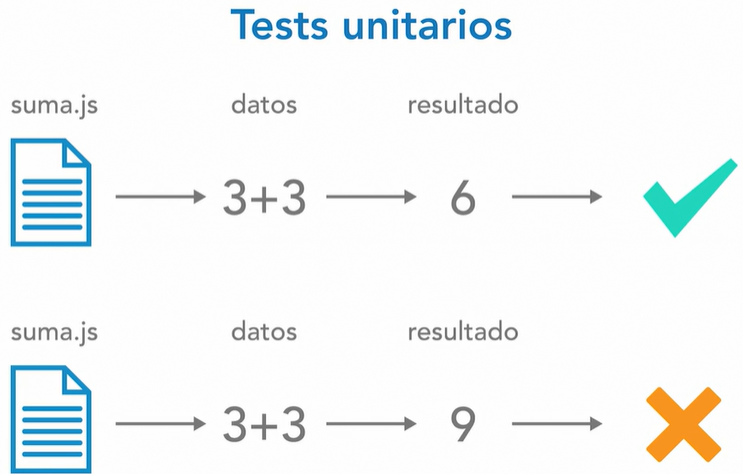
**Tests automáticos**

Para conseguir un producto de calidad **debemos tratar de automatizar los Tests** haciendo que estos sean automáticos, y **que no dependan de las ejecuciones manuales para recibir un feedback continuo sobre los errores que se van encontrando**

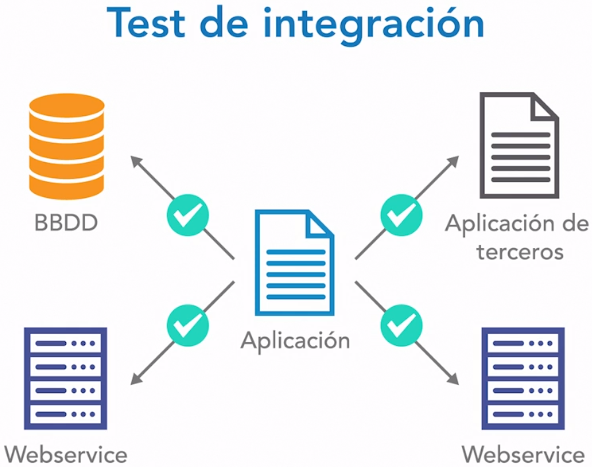
La idea es crear un flujo de despliegue en el que se **ejecuten una serie de Tests automáticos cada vez que un developer realiza un cambio en el código, por ende se crea un flujo de test que se ejecuta automáticamente tras cada “commit”**. Para eso tenemos herramientas como **Bamboo, GitLab o Travis**

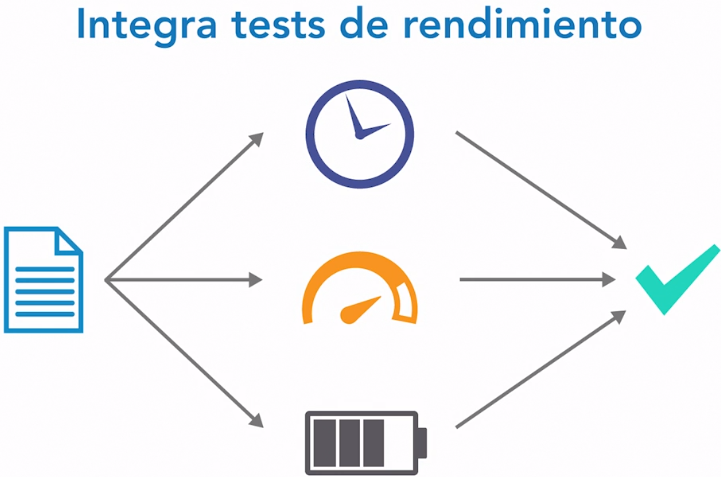
Por ende, se busca **AUTOMATIZAR TODO LO MANUAL POSIBLE**

**Hay 4 tipos de Tests que debe pasar nuestro software:**

**Tests Unitarios**: sirven para comprobar el **funcionamiento de un solo método o función de nuestro código para ver si cuando lo alimentamos con datos nos devuelve el resultado esperado**

**Tests de Aceptación**: se testea la app completa para **verificar en un alto nivel que funciona de forma amigable con el usuario**

**Tests de Integración**: se verifica que la **app se comunica de forma correcta con todos los servicios externos a los que debe conectarse para cumplir sus objetivos** (API, bases de datos, microservicios, apps externas, etc)

**Tests de Rendimiento**: debemos **evitar los problemas de rendimiento que pudieran generar nuestros cambios en el software**

**Es necesario que los Tests corran en paralelo para que así sean lo más efectivos y rápidos posibles, y para eso necesitamos eliminar la mayor cantidad posible de tareas manuales y convertirlas en procesos y Tests automáticos, permitiéndonos así eliminar retrasos y esperas en la cadena de despliegue y construcción**

**Despliegues de bajo riesgo**

**Los Tests automáticos permiten que la programación sea de “bajo riesgo”**, ya que los mismos nos alertarán continuamente de los fallos de nuestro software

**Es importante que los procesos de Testing y Despliegue sean automáticos**, y en este flujo podemos nombrar: Empaquetado del código, creación de contenedores o máquinas virtuales, instalación de paquetes o software para el despliegue y su subida a los servidores, reinicio de servicios de la app, ejecución de Tests directos, etc.

**Feedback entre departamentos**

**Monitorización (Telemetría)**

El monitorizar activamente nuestro servicios nos permite **darnos cuenta rápidamente de las cosas que van mal** y como solucionarlas de forma eficiente

Para eso importante **dirigir todos los datos que estamos monitorizando** (ya sean eventos, métricas,”logs”, etc) **hacia un enrutador centralizado que decida qué hacer con ellos** (almacenarlos a largo plazo, usarlos para alertar, hacer detección de anomalías, etc)

**Aclaraciónes:** Un "**log**" es un **registro o un archivo que registra eventos, acciones o mensajes importantes que ocurren en un sistema**. Son útiles para el seguimiento de problemas, el análisis de fallos, la auditoría de seguridad, el monitoreo de sistemas y la solución de problemas

La **telemetría** en programación es un enfoque utilizado para **recopilar datos y métricas sobre una aplicación o sistema en tiempo real**, con el objetivo de monitorear su rendimiento, detectar y diagnosticar errores, y tomar decisiones basadas en datos para mejorar su funcionamiento

Debemos asegurarnos de que lleguen **logs suficientes y con la información adecuada** para poder generar eventos precisos

Por ende es importante que configuremos la **severidad de los logs** mediante:

\*categorizarlos por nivel (“debug”, “info”, “warning”, “error”, “fatal”, etc)

\*asegurarnos que tengan información concreta de la situación generada

Se deben realizar logs en:

\*eventos de autenticación y autorización

\*accesos al sistema y datos

\*cambios a la app y al sistema

\*cambios a los datos

\*entrada de datos inválidos

\*recursos (memoria, disco, etc)

\*salud y disponibilidad del servicio

\*fallos y errores

\*tiempo de retardo y éxito en las copias de seguridad

**Pasos de la monitorización o Telemetría**

Primero se elige un **framework para la toma de datos** (Stats de Telegraph, Fluende, etc), luego debemos tener una **librería adecuada para extraer telemetría de cualquier punto de la app y mandarlo a nuestro “router”** de telemetría, y por último **permitir el acceso a estos datos a los componentes de Operaciones** (mediante una web o panel de control, capacidad de autoservicio, ver qué efecto tienen los cambios)

**Análisis**

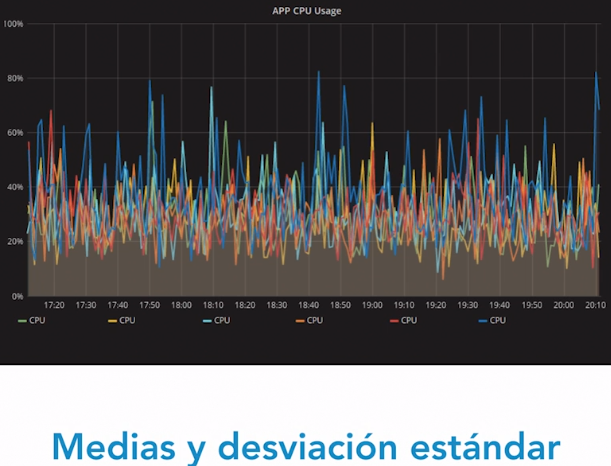
**Es imposible revisar todos los datos que recibimos de la monitorización**, ya que incluso la mayoría de ellos son innecesarios. Por eso **es vital que hagamos un buen análisis de ellos para darles un sentido**

Un recurso muy usual es usar **Medias o Desviaciones Estándar**

**Es importante establecer alertas cada vez que la telemetría excede X valor la media de nuestros datos**

**¿Con qué datos debemos establecer las alertas?**

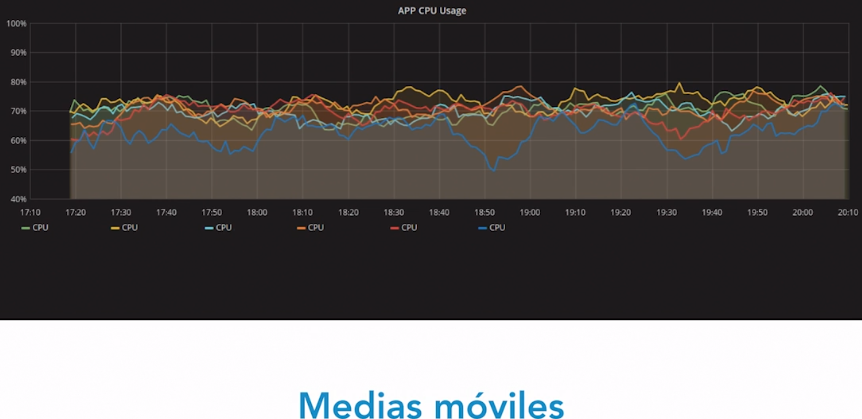
Si ponemos alertas en demasiados valores o las mismas son muy bajas, **corremos el riesgo de que las alertas salten demasiadas veces y de forma incorrecta al personal de Operaciones**

**Problemas que nos encontramos**

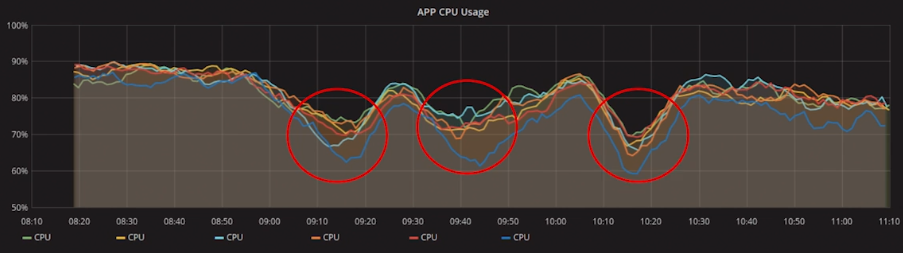
\***Los picos continuos de nuestro tráfico hacen saltar las alertas continuamente**, pero si elevamos el nivel de alerta hasta un punto que esto no pase haría que se pierda todo el sentido y la utilidad de las alertas

**¿Con qué método podemos solucionar este problema?**

Podemos utilizar las **Medias Móviles**

En vez de dibujar una toma de datos instantánea de cada punto de la gráfica, **se juntan los valores de los últimos 10 o 15 minutos para realizar una media, y esto suaviza muchísimo las gráficas y nos ayuda a evitar los picos que hacen saltar falsas alarmas**

**Problema que surge**

**Las medias móviles detectan picos de tráfico continuados de periodos más largos que la media móvil que se está calculando, pero no detecta auténticas anomalías**

Ej:

En este ejemplo vemos que no tenemos picos hacia arriba, sino que vemos 3 anomalías de bajada de tráfico que pueden coincidir con algún tipo de problema o incidencia

**Cómo resolverlo**

Debemos **utilizar un Algoritmo de Detección de Anomalías**, por ejemplo Kolmogorov-Smirnov

Sea cual sea el método de detección de anomalías, debemos centrarnos en una serie de campos en los que trabajar: **dónde monitorizamos, cómo monitorizamos y qué problemas por arriba y por debajo de la gráfica no podemos monitorizar**

**Feedback y Reviews**

El flujo de información entre los Developers y las Operaciones es muy importante

**La revisión entre pares antes del despliegue y la telemetría de los sistemas tras el despliegue son información fundamental para el éxito**

Ej: en GitHub existen los “**pull request**” (**peticiones al administrador del repositorio para unir nuestro código modificado al código principal**)

**En ellos el developer crea una nueva rama o copia del código dedicada exclusivamente a ese “bug” o funcionalidad que se desea probar, aplica en esa rama del código las modificaciones necesarias, y cuando todo está listo abre una “pull request”**. Durante este proceso se pueden recibir comentarios sobre el código desde los demás programadores y, si todo es correcto, aceptar esas modificaciones

Hay una relación directa entre el tamaño del código y el tiempo que se tarda en aprobar el mismo

**Aprendizaje Técnico**

**Cultura de Trabajo**

Es importante inculcar una buena cultura de trabajo en la organización con el objetivo de mejorar el autodiagnóstico, conocer mejor nuestra app y mejorar la capacidad de recuperación y reparación de nuestros sistemas ante cualquier eventualidad

Una de las políticas más importantes es la “**Cultura del Caos**”

**La cultura del caos** **se basa en crear caos en nuestro sistema y hacer fallar nuestra app a propósito para poder ver cuál es la manera que tiene de fallar y cómo podemos hacer que cuando falle lo haga causando el menor impacto posible en el usuario en la vida real, con el objetivo de que el sistema pueda responder de buena forma mientras los Developers resuelven el problema**

Podemos conocer más de estos principios en principlesofchaos.org

Una vez que hayamos creado el caos en nuestro sistema, mejorado la respuesta a estas incidencias y reducido el daño que recibe la app cada vez que se encuentra con algún fallo, tendremos menos alertas en nuestro sistema de telemetría

Pero ahora nos **queda perfeccionar nuestro sistema, y para eso debemos ir reduciendo paulatinamente el margen de tolerancia de nuestro sistema de monitorización para que cada vez nos avise de fallos más pequeños y triviales** y asi tengamos un perfeccionamiento del sistema

**Comunicación y Conocimiento**

Es clave **disponer de canales claros de comunicación y documentación** tanto para los humanos como para los procesos, además de la visibilidad y facilidad de acceso a todos los eventos diarios de la empresa

**Sistemas de Comunicación**

Algunos sistemas muy utilizados por las empresas pueden ser Slack, HipChat, Mattermost, etc.

Además son muy utilizados los **ChatBots informativos**, los cuáles son muy útiles para visibilizar los procesos y actividades comunes dentro de la empresa (advertencias sobre reuniones de tu departamento, reserva de salas, aviso sobre despliegues o compilaciones, etc)

Una evolución de estos Chatbots son los **ChatOps**, los cuales están conectados a los flujos del sistema y nos permiten realizar operaciones directamente desde la ventana del chat (lanzar un despliegue, mantener un contenedor, escalar un grupo de servidores, etc)

**Estandarización de la Documentación**

Es importante reservar tiempo para la creación de la documentación, la creación de plantillas de documentos obligatorios, requisitos de información mínima, documentación a través de los Tests, automatización de la documentación, etc

Además es importante compartir experiencias y conocimientos entre los empleados para así generar un flujo de conocimiento muy efectivo

**Herramientas DevOps**

**Integración y despliegue continuos**

Tenemos **herramientas de integración continua (CI) y despliegue continuo** para utilizar en nuestras apps DevOps, ya sea de forma online o integrada en nuestros servidores

Aquí unos ejemplos:

**\*Jenkins**

Es “**open source**” y contiene **gran cantidad de “pluggins”** o complementos (componente de software que se agrega a una aplicación o sistema existente para ampliar o modificar su funcionalidad y añadir nuevas características o capacidades a un software sin necesidad de modificar su código fuente original) para cosas como: conectar con la nube, autoescalar, etc

**\*Gitlab CI**

Es “**open source**”, posee un **sistema de control de versiones** basado en Git, sistema de gestión de incidencias, **sistema de DevOps para despliegues automáticos**, “workers” locales y en la nube para realizar nuestras compilaciones, Tests para CI integrados dentro del código y versionados

Posee instalación en nube o en local

**Monitorización**

**La monitorización es esencial ya que debe ser capaz de captar información de todos nuestros servicios y mostrarla de una manera útil**

Aquí algunos ejemplos de software de monitorización:

**\*Nagios**

Es una de las más clásicas y antiguas, está orientada a sistemas y tiene gran cantidad de pluggins para modificar y agregar funcionalidades

Además hay otros como StatsD, Telegraf, Graphite, AWS CloudWatch, Google StackDriver, Pingdom, NewRelic, etc

**Alerta**

**Tanta monitorización no nos servirá de mucho si no tenemos un buen sistema de alertas que nos haga saber si hemos perdido el control de algún servicio**

Tenemos softwares de alerta como Kapacitor, OpsGenie, PagerDuty, PushOver, etc.

**Infraestructura como código**

Podemos almacenar toda la configuración de la infraestructura y servidores en forma de código y gestionar los cambios como si fueran versiones de nuestro software

Ejemplos de estos son: Ansible, Chef, Puppet, Salt Stack, Stack HashiCorp,

**Comunicación y ChatOps**

Estas herramientas permiten que los trabajadores se comuniquen eficientemente y que gracias a ChatOps puedan tener feedback desde los procesos y lanzar operaciones desde el chat

Ej: Hubot, Atlassian Stride, Slack, Mattermost, etc