Continuous Delivery

Vera Juan Ignacio, Demarchi Leandro, Gemio Patricia Soledad, Marcelo Salaberri, Fonseca Villarreal Rodrigo

Universidad Tecnológica Nacional - Facultad Regional Córdoba
Ingeniería en Sistemas de Información - Cátedra de Ingeniería de Software

***Resumen*—El continuous delivery (entrega continua en español), es un proceso recíproco que integra y automatiza el desarrollo, la entrega, el feedback y la gestión de calidad. El resultado es que podemos simplificar las fases de trabajo para que nos lleven menos tiempo y se completen de forma más eficiente.**

En este documento especificaremos los beneficios y desventajas de su implementación, sus fases y herramientas más utilizadas.

***Índice de Términos*—Builds, commit, entrega continua, feedback, pipeline, software, testing.**

I. INTRODUCCIÓN

La entrega continua es una disciplina que lleva la práctica del desarrollo ágil a su conclusión lógica, creando software que siempre está listo para ser liberado. Para lograrlo incorpora prácticas y herramientas de Agile, integración continua y DevOps para transformar la manera en que se entrega software. Al automatizar tareas rutinarias y tardadas como la compilación pruebas e instalación -y ejecutarlas de forma temprana y frecuente- la entrega continua hace que el proceso de software sea más predecible y repetible, mejorando significativamente la calidad y frecuencia de las entregas.

Para llegar a lo que hoy es “Entrega Continua”, primero se implementaron las metodologías ágiles, que enseñaron a agregar valor a los productos una historia a la vez, acelerando los circuitos de retroalimentación y alineando los productos con las necesidades del mercado. La integración continua (CI) que siguió creó una avalancha de lanzamientos más pequeños y más frecuentes.

Luego surgió DevOps, (una combinación de las palabras "Desarrollo" y "Operaciones") como un modelo de operaciones unificadoras que enfatiza la automatización, comunicación, colaboración e integración compartidas. Al estandarizar los procesos, las configuraciones de versiones y aumentar la colaboración.

Continuous Delivery es un concepto bastante innovador de desarrollo de software que cada vez se escucha con más frecuencia. Gracias a esta práctica, las fases de producción que incluyen el desarrollo, el control de calidad y la entrega, no son definitivas, sino que se repiten de forma automatizada una y otra vez durante todo el proceso de desarrollo a través de un pipeline de continuous delivery. La principal ventaja es que, de esta manera, un software puede someterse cada poco tiempo a controles de calidad en cada una de sus fases de desarrollo, permitiendo realizar entregas, aunque el equipo siga trabajando en el desarrollo del producto final. Además, hay un feedback constante que procede del pipeline, lo que nos permite mejorar el software de forma inmediata tras cada modificación introducida en el código fuente.

II. DEFINICIÓN

Continuous Delivery hace referencia a una serie de prácticas utilizadas en el desarrollo de software para ejecutar el desarrollo, la entrega, el feedback y la gestión de calidad de forma simultánea y cada poco tiempo, de acuerdo con un proceso repetitivo. De esta forma, se gana en eficiencia y el cliente puede recibir el producto con mucha más prontitud, incluso cuando todavía no está terminado. La entrega continua genera un feedback constante para el desarrollador procedente de pruebas automatizadas que sirven, en general, para comprobar la estructura después de cada cambio introducido en el código fuente.

III. BENEFICIOS

Antes, el desarrollo de software funcionaba de una manera diferente: el producto final solo se entregaba si todas las funcionalidades estaban totalmente desarrolladas, funcionaban a la perfección y no se detectaban fallos importantes cuando se realizaban las pruebas de calidad. Así que el desarrollador tenía que entregar posteriormente parches o actualizaciones cada cierto tiempo. Gracias al continuous delivery, el cliente recibe el producto en una fase más temprana del desarrollo en la que todavía no ha sido terminado. Esta pre-entrega suele incluir la funcionalidad estructural del software para que el cliente pueda probarla en un entorno real. De esta manera, el

propio cliente (o el tester de software) juega un papel muy importante dentro del proceso de control de calidad.

Gracias al feedback recibido, el desarrollador puede mejorar las funcionalidades del producto durante la fase de desarrollo. Además, recibe información muy valiosa que le aporta una idea clara sobre qué funcionalidad debería desarrollar a continuación.

Como muestra en Fig.1, las tres áreas que incluyen el desarrollo, el control de calidad y la producción no se reemplazan por un único proceso, sino que están constantemente interconectadas. De esta forma, un producto pasa por cada una de las fases individuales repetidamente y recibe mejoras continuas

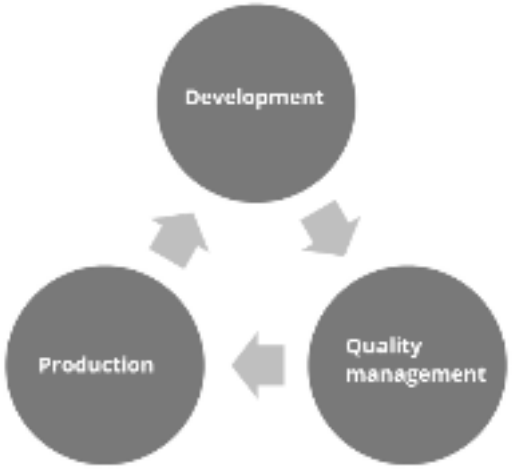

Fig.1. Continuous Delivery automatiza el ciclo de proceso relativo al desarrollo, garantía de calidad y producción.

Las prácticas de la entrega continua nos ayudan a lograr varios beneficios importantes.

A. Lanzamiento de bajo riesgo.

El objetivo principal de la entrega continua es hacer que las implementaciones de software sean eventos indoloros y de bajo riesgo que se puedan realizar en cualquier momento, bajo demanda. Al aplicar patrones como *Blue-green deployments*[1], es relativamente sencillo lograr implementaciones de tiempo de inactividad cero que no sean detectables por los usuarios.

B. Tiempo de comercialización más rápido.

No es raro que la fase de integración y prueba / reparación del ciclo de vida de entrega de software en fases tradicional consuma semanas o incluso meses. Cuando los equipos trabajan juntos para automatizar los procesos de creación y despliegue, aprovisionamiento del entorno y pruebas de regresión, los desarrolladores pueden incorporar pruebas de integración y regresión en su trabajo diario y eliminar por completo estas fases. También evitamos las grandes cantidades de reelaboración que afectan el enfoque por fases.

C. Mayor calidad.

Cuando los desarrolladores tienen herramientas automatizadas que descubren regresiones en cuestión de minutos, los equipos tienen la libertad de concentrar su esfuerzo en la investigación de usuarios y actividades de prueba de nivel superior, como pruebas exploratorias, pruebas de usabilidad y pruebas de rendimiento y seguridad. Al construir un *Deployment Pipeline*, estas actividades se pueden realizar continuamente durante todo el proceso de entrega, asegurando que la calidad se incorpore a los productos y servicios desde el principio.

D. Menores costos.

Cualquier producto o servicio de software exitoso evolucionará significativamente a lo largo de su vida útil. Al invertir en la compilación, prueba, implementación y automatización del entorno, reducimos sustancialmente el costo de realizar y entregar cambios incrementales al software al eliminar muchos de los costos fijos asociados con el proceso de lanzamiento.

E. Mejores productos.

La entrega continua hace que sea económico trabajar en pequeños lotes. Esto significa que podemos obtener comentarios de los usuarios a lo largo del ciclo de vida de entrega en función del software en funcionamiento. Las técnicas como *A/B Testing*[2] nos permiten adoptar un enfoque basado en hipótesis para el desarrollo de productos, mediante el cual podemos probar ideas con los usuarios antes de desarrollar funciones completas. Esto significa que podemos evitar los 2/3 de las características que creamos que ofrecen un valor cero o negativo a nuestros negocios.

F. Equipos más felices.

La investigación revisada por pares ha demostrado que Continuous Delivery hace que las liberaciones sean menos dolorosas y reduce el agotamiento del equipo. Además, cuando lanzamos con más frecuencia, los equipos de entrega de software pueden interactuar más activamente con los usuarios, aprender qué ideas funcionan y cuáles no, y ver de primera mano los resultados del trabajo que han realizado. Al eliminar las actividades dolorosas de bajo valor asociadas con la entrega

de software, podemos concentrarnos en lo que más nos importa: deleitar continuamente a nuestros usuarios.

La entrega continua no es mágica. Se trata de una mejora continua y diaria: la disciplina constante de lograr un mayor rendimiento siguiendo la heurística *"si duele, hágalo con más frecuencia y haga que el dolor avance"*[3].

IV. DESVENTAJAS

Hay una serie de desventajas, especialmente si se está pasando de la entrega tradicional a la entrega continua.

A. Errores en las pruebas automatizadas.

Las pruebas automatizadas tienen que funcionar a la perfección y no presentar errores de código. En caso contrario, pueden ocasionar graves daños al ser ejecutadas.

B. Falta de comunicación.

Requiere una buena y continua comunicación con los clientes y sus sistemas de destino. Requiere que haya una buena coordinación dentro del equipo porque los cambios introducidos en el código debe compilarse de forma eficiente y con una determinada frecuencia.

C. Dificultad para introducir modificaciones.

El cliente espera que haya mejoras y actualizaciones continuamente. Es difícil poder "pausar" el proyecto de desarrollo.

D. Implementación de modificaciones manual

A la hora de implementar innovaciones, mejoras o modificaciones al producto, sigue siendo necesario hacerlo manualmente. Si se quiere automatizar este proceso, es necesario recurrir al continuous deployment.

E. Falta de colaboración del cliente.

El cliente tiene que estar dispuesto a utilizar el software cuando todavía se encuentra en una fase de desarrollo. Además, tiene que poner de su parte y devolvernos su feedback.

F. Costo inicial.

Es necesario contar con un servidor potente y fiable de integración de datos para llevar a cabo las pruebas automatizadas y conseguir una liberación correcta y segura del producto. Las instalaciones iniciales, las configuraciones y los cambios del equipo no están exentos de costos. Estos elementos cuestan tiempo y dinero reales y, al principio, pueden crear interrupciones. El negocio debe invertir el tiempo requerido para inicializar la entrega continua, asegurándose de que las personalizaciones para los objetivos comerciales y las operaciones de infraestructura estén en su lugar y operativas.

G. Consideraciones sobre la cultura organizacional.

Si la empresa está acostumbrada a desarrollar software utilizando metodologías en cascada, en espiral u otras, debe superar la curva de aprendizaje antes de implementar la entrega continua. Desde la capacitación del personal hasta los procesos de ajuste, la organización debe mantener las operaciones existentes durante la transición a la entrega continua. Algunos miembros del equipo acostumbrados a ser más "prácticos" pueden desconfiar del uso intensivo de la automatización que conlleva la entrega continua. La clave es trabajar con expertos para ayudar a garantizar el éxito. Los resultados exitosos desde el principio generarán confianza en el proceso y reducirán el escepticismo.

V. EL DEPLOYMENT PIPELINE

El patrón clave introducido en la entrega continua es el *Deployment Pipeline*.

En el patrón, cada cambio en el control de versiones desencadena un proceso (generalmente en un servidor de integración continua) que crea paquetes desplegables y ejecuta pruebas unitarias automatizadas y otras validaciones, como el análisis de código estático. Este primer paso está optimizado para que se ejecute solo unos minutos. Si el *commit stage* falla, el problema debe solucionarse de inmediato. Cada *commit stage* desencadena el siguiente paso en el *pipeline*, que podría consistir en un conjunto más completo de pruebas automatizadas. Las versiones del software que pasan todas las pruebas automatizadas se pueden implementar a pedido en etapas posteriores, como pruebas exploratorias, pruebas de rendimiento, puesta en escena y producción.

El *Deployment Pipeline* vincula la administración de la configuración, la integración continua y la automatización de pruebas e implementación de una manera holística y poderosa que funciona para mejorar la calidad del software, aumentar la estabilidad y reducir el tiempo y el costo necesarios para realizar cambios incrementales en el software, independientemente del dominio que esté operando.

Al construir un *Deployment Pipeline*, se descubrieron valiosas prácticas:

a. *Solo crea paquetes una vez*. Queremos asegurarnos de que lo que estamos implementando es lo mismo que hemos probado en todo el proceso de implementación, por lo que si falla una

implementación, podemos eliminar los paquetes como la fuente de la falla.

b. *Implemente de la misma manera en todos los entorno*s, incluido el desarrollo. De esta manera, probamos el proceso de implementación muchas veces antes de que llegue a producción, y nuevamente, podemos eliminarlo como la fuente de cualquier problema.
c. *Prueba de humo en tus implementaciones*. Tenga disponible un script que valide que todas las

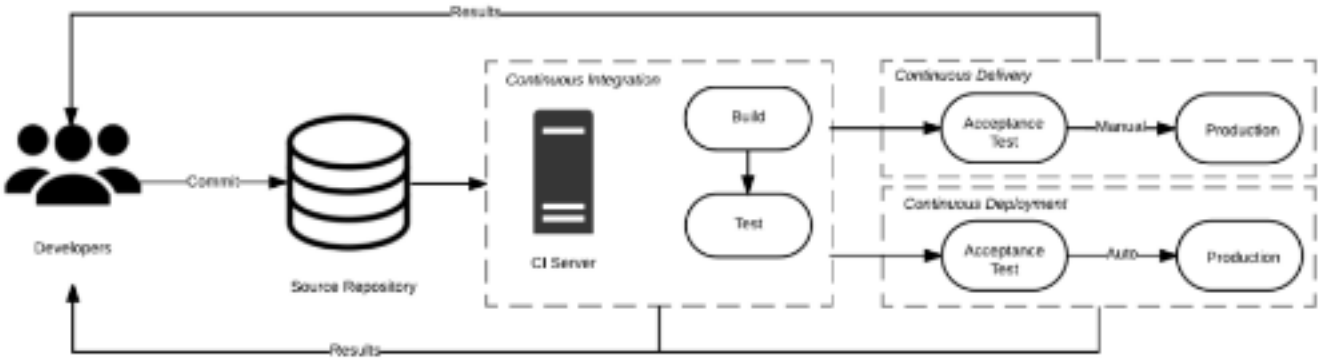

Fig.2.La relación entre integración continua, entrega y despliegue

dependencias de su aplicación están disponibles, en la ubicación donde ha configurado su aplicación. Asegúrese de que su aplicación se esté ejecutando y disponible como parte del proceso de implementación.

d. *Mantenga sus ambientes similares*. Aunque pueden diferir en la configuración del hardware, deben tener la misma versión del sistema operativo y los paquetes de middleware, y deben configurarse de la misma manera. Esto se ha vuelto mucho más fácil de lograr con la virtualización moderna y la tecnología de contenedores.

VI. FASES DEL PIPELINE

Cuando se produce una modificación en el código, se activa el pipeline de continuous delivery y se ejecutan las pruebas.

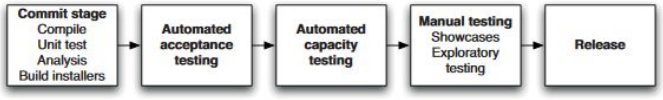

Fig.3. El pipeline de continuous delivery atraviesa varias fases y las ejecuta de principio a fin cada vez que se realizan cambios en el código.

A. Fase de commit (Commit Stage).

En esta primera fase de prueba se ejecutan tests de la versión del software, se desarrollan sus componentes y, cuando es necesario, se compilan. Además, se llevan a cabo las pruebas unitarias pertinentes. Si se superan con éxito todas las pruebas, la fase se da por finalizada. En este momento, se compilan y se almacenan en el repositorio los artefactos binarios de los componentes de software. Por todo lo anterior, se trata de un paquete que afecta decisivamente a la funcionalidad del pipeline porque determina el estado en el que se encuentra el software. Además, este paquete incluye todos los datos que serán instalados en un momento posterior en el sistema de destino. Los resultados de las pruebas en la fase de commit pueden asignarse de esta manera a las modificaciones concretas realizadas sobre el código fuente, que es precisamente una de las ventajas más significativas de la entrega continua.

B. Fase de aceptación (Acceptance Test Stage).

En la segunda fase de prueba se ejecutan los tests de aceptación. Estamos hablando de las pruebas de integración (para comprobar si la interacción entre los componentes funciona) y las pruebas necesarias del sistema (para comprobar si el software funciona del lado del usuario). Además, existen otras pruebas opcionales que forman parte de la fase de aceptación, como pruebas de rendimiento y otros tests que ponen a prueba requisitos no funcionales del software. Durante la fase de aceptación, se vuelve a utilizar la compilación ejecutada en la fase previa que pasa a ser instalada en un entorno de pruebas adecuado.

C. Feedback.

En caso de que se constaten errores o complicaciones durante alguna de las fases comentadas, se creará documentación al respecto y, cuando sea necesario, se deberá enviar feedback al desarrollador. Puede enviarse mediante correo electrónico, programas de mensajería o herramientas especiales (que comentaremos más adelante). Cada vez que se produce una modificación en el código, el pipeline se pone en funcionamiento por lo que los mensajes de error siempre hacen

referencia a la última modificación. Así, el desarrollador puede reaccionar con rapidez y de forma eficiente para arreglar los bugs o el código defectuoso.

D. Pruebas manuales.

Se realizan en función de las necesidades concretas de cada caso. A la hora de realizar estas pruebas, el pipeline vuelve a utilizar la compilación creada en la primera fase y la reinstala en un entorno de pruebas adecuado.

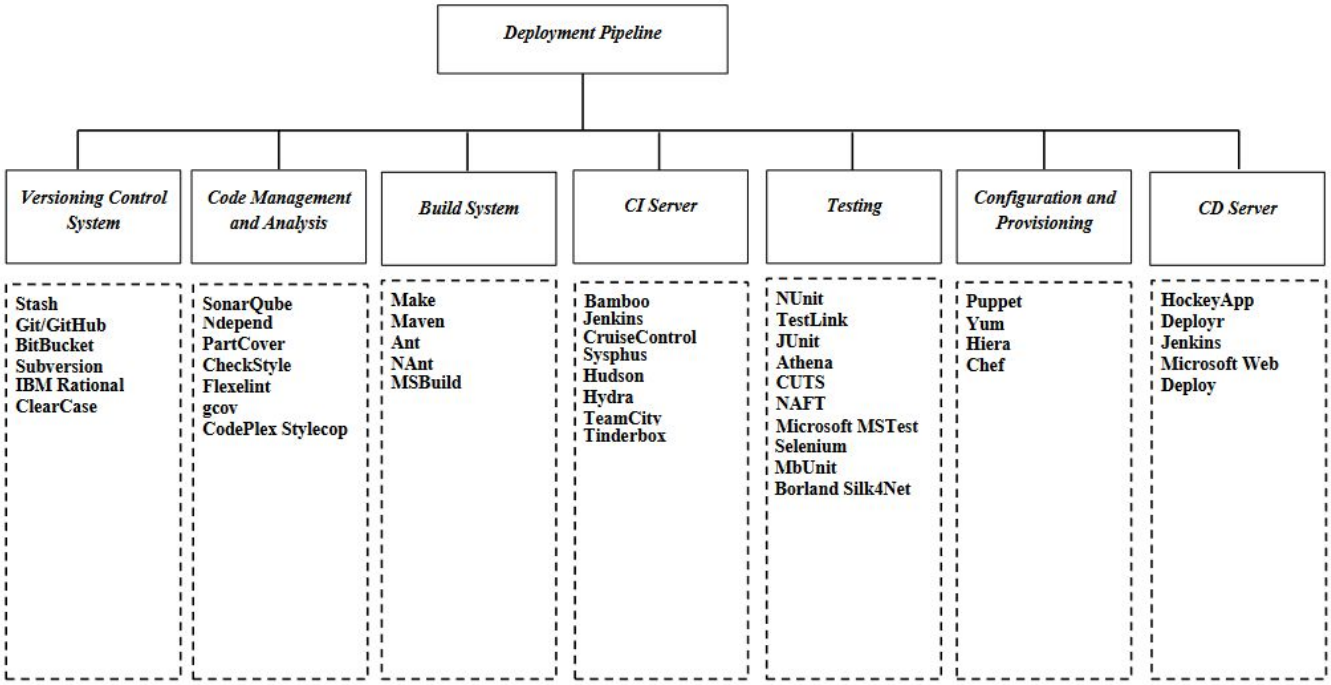

Fig.4.Herramientas utilizadas para formar el Deployment Pipeline.

E. Instalar el paquete.

Si se completan todas las pruebas y el feedback recibido es positivo, ha llegado el momento de instalar el paquete manualmente en el sistema de destino. Normalmente, esto se hace con un solo clic. Es posible automatizar este paso a través del continuous deployment.

VII. HERRAMIENTAS MÁS CONOCIDAS

Existen varios programas que nos ayudarán a comenzar a trabajar incorporando la entrega continua. A continuación cuatro de los más conocidos.

A. Jenkins.

Jenkins es una aplicación web que facilita la integración continua de los componentes del software. Jenkins está escrito en Java, se ejecuta en un contenedor EJB y dispone de varias herramientas de build (Apache Ant, Maven/Gradle, CVS, Subversion, Git, etc.) y, por supuesto, de los procesos de pruebas automáticas tan importantes en el caso del continuous delivery (Junit, Emma). Existen plugins opcionales que sirven para garantizar la compatibilidad con otros compiladores. Gracias a su interfaz basada en REST, otros programas pueden acceder a Jenkins. Jenkins es un programa gratuito de código abierto. Está recomendado especialmente para principiantes porque su interfaz y las funcionalidades que ofrece son muy intuitivas y fáciles de usar.

B. CircleCI.

CircleCI también es una aplicación web para continuous integration, delivery and deployment. CircleCI funciona preferiblemente con GitHub, GitHub Enterprise y Bitbucket. Además, la plataforma ofrece muchas funcionalidades prácticas como la gestión de pedidos, gestión de recursos, docker support, soporte de todos los lenguajes de programación conocidos, almacenamiento seguro en caché, análisis de datos con estadísticas y completos conceptos de seguridad. CircleCI recibió el premio “Leader in Continuous Integration” de Forrester en 2017.

C. Microsoft Team Foundation Server.

Microsoft Team Foundation Server (TFS) es una herramienta colaborativa para proyectos de software que tienen que planificarse, desarrollarse y finalmente ser gestionados de

manera conjunta. TFS es el sucesor no oficial de Microsoft's Visual SourceSafe. Para permitir el trabajo colaborativo en los proyectos de software, TFS soporta varios procesos de desarrollo, incluido el CMMS, el desarrollo de software ágil y Scrum. Además, TFS está vinculado e integra programas de Office muy conocidos como Word y Excel para que no tengas que salir de TFS y abrir otro programa.

Existen varias características que te permiten crear un pipeline para continuous integration, delivery and deployment. Lo que hace TFS básicamente es dividir el proceso completo en varios apartados: control de versiones, build, reports y administración de usuario.

D. Codeship.

Codeship es una plataforma SaaS de integración continua (y de entrega) que se adapta a las necesidades particulares de cada usuario. Codeship soporta GitHub, Bitbucket y GitLab. Las funcionalidades disponibles dependen del plan contratado.

VIII. PRINCIPIOS

Hay cinco principios en el corazón de entrega continua:

- Calidad de construcción.
- Trabajar en lotes pequeños.
- Las computadoras realizan tareas repetitivas, las personas resuelven problemas.
- Sin descanso persigue la mejora continua.
- Todos son responsables.

Es fácil empantanarse en los detalles de la implementación de la entrega continua —herramientas, arquitectura, prácticas, política— si se encuentra perdido, intente revisar estos principios y es posible que le ayude a re-enfocarse en lo que es importante.

IX. TESTING

Hay muchos tipos diferentes de testing que se pueden usar para asegurar de que los cambios en el código funcionen como se espera. Sin embargo, no todas las pruebas son iguales, y a continuación se presentan las principales técnicas de testing que difieren entre sí.

En un nivel alto, necesitamos hacer la distinción entre pruebas manuales y automatizadas. Las pruebas manuales se realizan en persona, haciendo clic en la aplicación o interactuando con el software y las API con las herramientas adecuadas. Esto es muy costoso, ya que requiere que alguien configure un entorno y ejecute las pruebas por sí mismo, y puede ser propenso a errores humanos, ya que el probador puede hacer errores tipográficos u omitir pasos en el script de prueba.

Las pruebas automatizadas, por otro lado, son realizadas por una máquina que ejecuta un script de prueba que se ha escrito de antemano. Estas pruebas pueden variar mucho en complejidad, desde verificar un solo método en una clase hasta asegurarse de que realizar una secuencia de acciones complejas en la interfaz de usuario conduzca a los mismos resultados. Es mucho más robusto y confiable que las pruebas manuales, pero la calidad de sus pruebas automáticas depende de qué tan bien se hayan escrito sus guiones de prueba.

Las pruebas automatizadas son un componente clave de la integración continua y la entrega continua y es una excelente manera de escalar su proceso de control de calidad a medida que agrega nuevas funciones a su aplicación. Pero todavía hay valor en hacer algunas pruebas manuales con lo que se llama pruebas exploratorias.

X. DIFERENTES TIPOS DE TESTING

A. Unit tests

Las pruebas unitarias son de muy bajo nivel, cercanas a la fuente de su aplicación. Consisten en probar métodos y funciones individuales de las clases, componentes o módulos utilizados por su software. En general, las pruebas unitarias son bastante baratas de automatizar y un servidor de integración continua puede ejecutarlas muy rápidamente.

B. Integration tests

Las pruebas de integración verifican que los diferentes módulos o servicios utilizados por su aplicación funcionen bien juntos. Por ejemplo, puede probar la interacción con la base de datos o asegurarse de que los microservicios funcionan juntos como se espera.

Este tipo de pruebas son más costosas de ejecutar, ya que requieren que varias partes de la aplicación estén en funcionamiento.

C. Functional tests

Las pruebas funcionales se centran en los requisitos comerciales de una aplicación. Solo verifican el resultado de una acción y no verifican los estados intermedios del sistema cuando realizan esa acción.

A veces hay una confusión entre las pruebas de integración y las pruebas funcionales, ya que ambas requieren múltiples componentes para interactuar entre sí. La diferencia es que una prueba de integración simplemente puede verificar que puede consultar la base de datos mientras que una prueba funcional esperaría obtener un valor específico de la base de datos según lo definido por los requisitos del producto.

D. End-to-end tests

Las pruebas de extremo a extremo replican el comportamiento de un usuario con el software en un entorno de aplicación completo. Verifica que varios flujos de usuarios funcionan como se espera y pueden ser tan simples como cargar una página web o iniciar sesión o escenarios mucho más complejos que verifican notificaciones por correo electrónico, pagos en línea, etc.

Las pruebas de extremo a extremo son muy útiles, pero son caras de realizar y pueden ser difíciles de mantener cuando están automatizadas. Se recomienda realizar algunas pruebas clave de principio a fin y confiar más en los tipos de pruebas de nivel inferior (pruebas de unidad e integración) para poder identificar rápidamente los cambios importantes.

E. Acceptance testing

Las pruebas de aceptación son pruebas formales ejecutadas para verificar si un sistema satisface sus requisitos comerciales. Requieren que toda la aplicación esté en funcionamiento y se centren en replicar los comportamientos de los usuarios. Pero también pueden ir más allá y medir el rendimiento del sistema y rechazar los cambios si no se cumplen ciertos objetivos.

F. Performance testing

Las pruebas de rendimiento verifican los comportamientos del sistema cuando está bajo una carga significativa. Estas pruebas no son funcionales y pueden tener varias formas para comprender la confiabilidad, estabilidad y disponibilidad de la plataforma. Por ejemplo, puede estar observando los tiempos de respuesta al ejecutar una gran cantidad de solicitudes, o ver cómo se comporta el sistema con una cantidad significativa de datos.

Las pruebas de rendimiento son, por naturaleza, bastante costosas de implementar y ejecutar, pero pueden ayudarlo a comprender si nuevos cambios van a degradar su sistema.

F. Smoke testing

Las pruebas de humo son pruebas básicas que verifican la funcionalidad básica de la aplicación. Están destinados a ser rápidos de ejecutar, y su objetivo es brindarle la seguridad de que las principales características de su sistema funcionan como se espera.

Las pruebas de humo pueden ser útiles justo después de que se realice una nueva compilación para decidir si puede ejecutar pruebas más costosas o justo después de una implementación para asegurarse de que su aplicación se ejecute correctamente en el entorno recién implementado.

XI. MÉTRICAS

Para la entrega continua, la métrica global más importante es la de los tiempos de ciclos, que es el tiempo entre que se decide qué característica implementar, y cuando dicha característica se halla lanzado para los usuarios. Esta métrica es difícil de medir ya que abarca desde el análisis, a través del desarrollo, hasta su despliegue, pero nos dice más acerca del proceso que cualquier otra métrica.

Una implementación adecuada de la deployment pipeline es hacer a esta simple, para poder calcular los tiempos de ciclo de una parte por todo el flujo desde que se revisa, hasta que se entrega o implementa dicha parte. Una vez que se conoce el tiempo de ciclo de la aplicación, se puede trabajar para reducirla, para esto se puede utilizar la teoría de restricciones, aplicando el siguiente proceso:

1. Identificar las restricciones que limiten el sistema.
2. Explotar la restricción. Esto quiere decir maximizar el rendimiento de esa parte del proceso.
3. Subordinar todos los otros procesos a la restricción. Esto implica que los otros recursos no trabajará al 100%.
4. Elevar la restricción. Si el tiempo de ciclo es muy largo, se necesita incrementar los recursos disponibles, contratando nuevos testers o tal vez invirtiendo más en testing automatizado
5. Revisar y repetir. Encontrar la siguiente restricción en el sistema y volver al paso 1.

Mientras que el tiempo de ciclo es la métrica más importante de la entrega continua, existen otros diagnósticos que nos pueden advertir de problemas, estos diagnósticos pueden ser:

- Cobertura de prueba automatizada.
- Propiedades del código base como la cantidad de duplicación, problemas de estilo, entre otros.
- Número de defectos.
- Velocidad, el ritmo con el que el equipo entrega codigo funcionando, listo para usar.
- Número de commits en el control de versión por dia.
- Número de builds por dia
- Número de fallos de build por dia
- Duración de la build, incluidas las pruebas automáticas.

XII. CONCLUSIONES

Poner el software en producción es lento y arriesgado. La optimización de este proceso tiene el potencial de hacer que el desarrollo de software en general sea más efectivo y eficiente. Por lo tanto, la entrega continua podría ser una de las mejores opciones para mejorar los proyectos de software.

Continuous delivery es una buena práctica que no es fácil de lograr o implementar en el equipo, pero es el camino adecuado para alejarse de los dolores de cabeza que pueden ser realizar deploy a producción, también nos ayuda a reducir los errores al momento de realizar nuestra actualización y cualquier tipo de problemas que puedan ser visibles para los usuarios.

La entrega continua tiene como objetivo procesos regulares y reproducibles para entregar software, al igual que lo hace la integración continua para integrar todos los cambios. Si bien la entrega continua parece una excelente opción para disminuir el tiempo de comercialización, en realidad tiene mucho más que ofrecer: es un enfoque para minimizar el riesgo en un proyecto de desarrollo de software porque garantiza que el software se pueda implementar y ejecutar en producción. Por lo tanto, cualquier proyecto puede obtener alguna ventaja, incluso si no se encuentra en un mercado muy competitivo donde el tiempo de comercialización no es tan importante después de todo.

REFERENCIAS

[1] Blue-green deployment. *Martin Fowler.* https://martinfowler.com/bliki/BlueGreenDeployment.html

[2] Online Controlled Experiments: Introduction, Insights, Scaling, and Humbling Statistics. *Conference by Ronny Kohavi.* https://www.infoq.com/presentations/controlled-experiments/

[3] Frequency Reduces Difficulty. *Martin Fowler.* https://martinfowler.com/bliki/FrequencyReducesDifficulty.html

ANEXO

[1] IEEE conferences templates. IEEE. https://www.ieee.org/conferences/publishing/templates.html

BIBLIOGRAFÍA

[1] Continuous Delivery – Pipelines en el desarrollo de software. Artículo del 27/05/2019 - *Digital Guide powered by IONOS.* https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/continuous-delivery/

[2] What is Continuous Delivery?. https://continuousdelivery.com/

[3] M. Shahin, M. Ali Babar, and L. Zhu, Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices", IEEE Access, 2017.

[4] Rossel Sander, Continuous Integration, Delivery and Deployment, Editorial Packt, 2017.

[5] What is a continuous delivery pipeline? https://www.atlassian.com/continuous-delivery/pipeline

[6] Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. *Jez Humble, David Farley.*

[7] The different types of software testing. *Sten Pittet.* https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

[8] Entrega Continua: ¿Cómo llegamos aquí? https://sg.com.mx/revista/54/entrega-continua