Universidad Tecnológica Nacional
Facultad Regional Córdoba
Cátedra de Ingeniería de Software

Continuous Delivery

Docente: Judith Meles

Castillo Dahbar, Francisco	70280
Farfán, Fabio Hugo	52224
Martín Carrión, Rodolfo Emiliano	72223
Martínez, Gabriel Gonzalo	69567
Sturtz, Mailen	71846

Curso 4K1

10 de Octubre de 2019

Resumen

Dentro de la filosofía ágil existen muchas metodologías que, siguiendo sus conceptos centrales, buscan nuevas y mejores formas de llevar a cabo el proceso de desarrollo de software. Una de ellas es continuous delivery, que consiste en producir software de tal manera que pueda ser liberado a producción en cualquier momento con el objetivo de maximizar el valor puro del producto. Esta disciplina se ha vuelto popular en el desarrollo de software, ya que permite realizar mejores lanzamientos, más seguros y rápidos, brindando también otros beneficios tanto para el equipo de trabajo como para el cliente, que puede tener el producto en sus manos más rápido y brindar retroalimentación. Todo esto contribuye a que el resultado final sea un software de mayor calidad.

Dentro de esta metodología surge un punto clave, que consiste en la automatización de estos lanzamientos continuos, denominado continuous delivery pipeline, el cual posee algunas etapas para su aplicación. Implementarlo conlleva esfuerzo, pero brinda mayores beneficios y contribuye a incrementar la calidad del software.

Índice

Glosario

- **Ágil (Agile):** Filosofía que se compone por un conjunto de metodologías para el desarrollo de proyectos que precisan de una especial rapidez y flexibilidad en su proceso. En muchas ocasiones son proyectos relacionados con el desarrollo de software o el mundo de internet. Más información en: https://www.iebschool.com/blog/que-es-agile-project-management-ventajas-agile-scrum/

- **API (Interfaz de programación de aplicaciones):** Conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. Más información en:https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces

- **Archivos binarios.** Se las denomina así a las colecciones de código ejecutable como por ejemplo archivos .jar, .NET y .so.

- **Implementación azul-verde.** Este enfoque minimiza el tiempo de inactividad al garantizar que se tenga dos entornos de producción, lo más idénticos posible. Además, brinda una forma rápida de retroceder si algo sale mal. Más información en: https://martinfowler.com/bliki/BlueGreenDeployment.html

- **Lean:** Es una metodología basada en el sistema de producción de Toyota que, mediante la eliminación de desperdicios o actividades que no agregan valor, permite alcanzar resultados inmediatos en la productividad, competitividad y rentabilidad de las empresas sin la necesidad de realizar inversiones en maquinaria, personal o tecnología. Más información en: https://www.eoi.es/blogs/carmenrosabernabe/2012/02/06/filosofia-lean-manufacturing/#targetText=Lean%20Manufacturing%20es%20una%20metodolog%C3%ADa,en%20maquinaria%2C%20personal%20o%20tecnolog%C3%ADa.

- **Middleware:** Es software que se sitúa entre un sistema operativo y las aplicaciones que se ejecutan en él. Básicamente, funciona como una capa de traducción oculta para permitir la comunicación y la administración de datos en aplicaciones distribuidas. Más información en: https://azure.microsoft.com/es-es/overview/what-is-middleware/#targetText=Middleware%20es%20software%20que%20se,de%20datos%20en%20aplicaciones%20distribuidas.

- **Product Owner:** Es la persona responsable de asegurar que el equipo aporte valor al negocio. Representa las partes interesadas internas y externas, por lo que debe comprender y apoyar las necesidades de todos los usuarios en el negocio, así como también las necesidades y el funcionamiento del equipo de trabajo. Más información en: https://www.eude.es/blog/responsabilidades-product-owner/

- **Servidor CI (Servidor de integración continua):** Se encarga de ejecutar un script automáticamente según la programación que se le establezca para, por ejemplo, se realiza un commit/checkin/push al control de código fuente. Más información en: http://blog.koalite.com/2013/05/servidor-de-integracion-continua-una-buena-inversion/

Introducción

Desde sus inicios, la producción de software, como actividad, ha pasado por muchas etapas, evolucionando y mejorando cada vez más. Estos cambios, que afectan metodologías, paradigmas y filosofías, han sido adoptados por el mundo de manera positiva. Sin embargo, la actualidad continúa ofreciendo posibilidades y es oportuno aprovecharlas para no detenerse en el tiempo.

Los enfoques tradicionales brindaron métodos y modelos para estandarizar y ordenar las actividades en la producción de software, permitiendo desarrollar productos de calidad. O al menos, para el momento. A medida que pasa el tiempo siguen surgiendo paradigmas que hacen frente a las debilidades presentes en el desarrollo de software y que plantean diferentes enfoques para esta tarea. Uno de los cambios de enfoque más notorios que se puede observar es el de la filosofía ágil.

La filosofía ágil, desde su aparición, propone un enfoque distinto al tradicional, donde su lema es agilizar la producción de software sin derrochar tiempo en actividades innecesarias. Es importante aclarar que esto no implica que pueda permitirse la pérdida de calidad. Por el contrario, ágil centra su objetividad en maximizar el valor puro del producto.

Dentro de la filosofía, se pueden encontrar e incluir distintas prácticas que exploran y profundizan conceptos de ágil. Una de ellas en la entrega continua (Continuous Delivery), que se analizará para dar un vistazo general de qué es y cómo se integra con el proceso de desarrollo de software, dando como resultado la construcción de software que puede ser liberado a producción en cualquier momento, obteniendo retroalimentación más frecuentemente.

Además, se pondrá en comparación con las prácticas Continuous Deployment y Continuous Integration para diferenciarlas y ver cómo se relacionan entre sí, así como con la metodología Agile.

1 Continuous Delivery

Martín Fowler [1] define Continuous Delivery como una práctica de desarrollo software en la que este se construye de tal manera que puede ser liberado en producción en cualquier momento. Por su parte, Jez Humble [2] la define como la capacidad de obtener cambios de todo tipo, incluidas nuevas funciones, cambios de configuración, correcciones de errores y experimentos; en producción o en manos de los usuarios, de forma segura y rápida de una manera sostenible. Es decir que, al realizar continuous delivery, se busca asegurar que cada cambio realizado esté listo para ser lanzado a producción.

Según Javier Garzas [4], Continuous Delivery no implica, necesariamente, que se libere cada vez que hay un cambio. Sino que, esto solo ocurre si el responsable de negocio, el Product Owner de turno, decide pasar a producción el software una vez realizado el cambio. Por ende, hay un componente “humano” a la hora de tomar la decisión.

A su vez, Jez Humble [2] definió su objetivo como poner en producción continuamente cualquier sistema, ya sea un sistema distribuido a gran escala, en un entorno de producción complejo, un sistema integrado o una aplicación. Generalizando, asuntos predecibles y rutinarios que se pueden realizar a pedido. Todo esto se logra asegurando que el código esté siempre en un estado desplegable o de producción, incluso frente a equipos de miles de desarrolladores que realizan cambios a diario. De este modo, se eliminan por completo las fases de integración, prueba y endurecimiento que tradicionalmente seguían al "desarrollo completo", así como la congelación de código.

1.1 Principios de Continuous Delivery

Continuando con lo escrito por Jez Humble [2] en su libro Continuous Delivery, se definen los principios de esta práctica:

- **Calidad en la construcción.** Es mucho más barato solucionar problemas y defectos si se encuentran de inmediato, idealmente antes de que se registren en el control de versiones, ejecutando pruebas automatizadas localmente. Encontrar defectos aguas abajo a través de la inspección (como las pruebas manuales) lleva mucho tiempo y requiere un esfuerzo significativo. Luego arreglar el defecto, tratando de recordar lo que se estaba pensando cuando se abordó el problema días o incluso semanas atrás, es complicado.

  Crear y desarrollar bucles de retroalimentación para detectar problemas lo antes posible es un trabajo esencial e interminable en la entrega continua. Si se encuentra un problema en las pruebas exploratorias, no solo se deben solucionar, sino también preguntar: ¿Cómo se podría haber detectado el problema con una prueba de aceptación automatizada? Y cuando una prueba de aceptación falla, se debe preguntar: ¿Se podría haber escrito una prueba unitaria para detectar este problema?

- **Trabajar en lotes pequeños.** En los enfoques tradicionales por etapas para el desarrollo de software, las transferencias de desarrollo a prueba o de prueba a operaciones de TI consisten en lanzamientos completos, es decir, meses de trabajo de equipos compuestos por decenas o cientos de personas. En cambio, en la entrega continua, se toma el enfoque opuesto y se intenta obtener todos los cambios en el control de versiones lo más pronto posible hacia el lanzamiento, obteniendo comentarios exhaustivos lo más rápido posible.

Trabajar en lotes pequeños tiene muchos beneficios. Reduce el tiempo que lleva obtener retroalimentación sobre el trabajo, hace que sea más fácil clasificar y remediar problemas, aumenta la eficiencia y la motivación; y la razón de trabajar en pequeños lotes es que un objetivo clave de la entrega continua es cambiar la economía del proceso de entrega de software para que sea económicamente viable trabajar de esta manera y que se puedan obtener los mayores beneficios de este enfoque.

- **Las computadoras realizan tareas repetitivas, las personas resuelven problemas.** El objetivo es que las computadoras realicen tareas simples y repetitivas, como pruebas de regresión, para que los humanos puedan concentrarse en la resolución de problemas. Así, las computadoras y las personas se complementan entre sí.

  A muchas personas les preocupa que la automatización los deje sin trabajo, pero este no es el objetivo. Más bien, es que las personas se liberen del trabajo repetitivo para concentrarse en actividades de mayor valor. Esto también tiene el beneficio de mejorar la calidad, ya que los humanos son más propensos a errores cuando realizan tareas repetitivas.

- **Persigue implacablemente la mejora continua.** La mejora continua es una idea clave del movimiento Lean. Las mejores organizaciones son aquellas en las que todos tratan la mejora en el trabajo como una parte esencial de su desempeño diario.

- **Todos son responsables.** En las organizaciones de alto rendimiento, nada es "problema de otra persona". Los desarrolladores son responsables de la calidad y la estabilidad del software que crean. Los equipos de operaciones son responsables de ayudar a los desarrolladores a crear calidad. Todos trabajan juntos para lograr los objetivos a nivel organizacional, en lugar de optimizar lo que es mejor para su equipo o departamento.

  Cuando las personas realizan optimizaciones locales que reducen el rendimiento general de la organización, a menudo se debe a problemas sistémicos, como sistemas de gestión deficientes, ciclos presupuestarios anuales, o incentivos que recompensan los comportamientos incorrectos. Un ejemplo es recompensar a los desarrolladores por aumentar su velocidad de escribir código, y recompensar a los evaluadores en función de la cantidad de errores que encuentren.

  La mayoría de las personas quieren hacer lo correcto, pero adaptarán su comportamiento en función de cómo se les recompense. Por lo tanto, es muy importante crear ciclos de retroalimentación rápidos a partir de las cosas que realmente importan, cómo reaccionan los clientes a lo que construimos para ellos y el impacto en nuestra organización.

1.2 Beneficios de Continuous Delivery

Jez Humble [2] explica varios beneficios importantes que trae aparejado la implementación de continuous delivery:

- **Lanzamientos 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 pueden realizar en cualquier momento, bajo demanda. Al aplicar patrones como las implementaciones azul-verde, es relativamente sencillo lograr implementaciones de tiempo de inactividad cero que no sean detectables por los usuarios.

- **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 tradicionales 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 se evita las grandes cantidades de reelaboración que afectan el enfoque por fases.

- **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 niveles superiores, como pruebas exploratorias, pruebas de usabilidad y pruebas de rendimiento y seguridad. Al construir una tubería de implementación, éstas 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.

- **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, se reduce sustancialmente el costo de realizar y entregar cambios incrementales al software al eliminar muchos de los costos fijos asociados con el proceso de lanzamiento.

- **Mejores productos.** La entrega continua hace que sea económico trabajar en pequeños lotes. Esto significa que se puede obtener comentarios de los usuarios a lo largo del ciclo de vida de entrega en función del software en funcionamiento. Varias técnicas permiten adoptar un enfoque basado en hipótesis para el desarrollo de productos, mediante el cual se puede probar ideas con los usuarios antes de desarrollar funciones completas. Esto significa que se puede evitar los 2/3 de las características que se crean que ofrecen un valor cero o negativo al negocio.

- **Equipos más felices.** Cuando se realizan lanzamientos 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 de bajo valor asociadas con la entrega de software, se pueden concentrar en lo que más importa: deleitar continuamente a los usuarios.

A su vez, Juan F. Samaniego [5] añade a los ya mencionados, los siguientes beneficios:

- **Aumento de la capacidad de reacción ante errores.** Poder probar el producto de forma continua facilita la identificación de errores y su subsanación.

- **Mayor capacidad de experimentación.** Al agilizar el proceso de lanzamiento, las nuevas funciones pueden probarse, activarse o desactivarse con diferentes audiencias o públicos y con mayor flexibilidad.

1.3 Problemas que afectan a la calidad del software

En esta sección, se presenta una lista de problemas relacionados con la calidad de software, que han sido reportados en revisiones de la literatura, casos de estudio y artículos empíricos, cuyo objetivo era el estudio y la implementación de Continuous Delivery (CD) presentados en el artículo de Maximiliano A. Mascheroni y Emanuel Irrazábal [6].

- **Pruebas que consumen mucho tiempo de ejecución.** Existen muchos tipos de pruebas que pueden implementarse en CD. Sin embargo, ejecutar un gran lote de pruebas es una tarea que lleva mucho tiempo. Además, los cambios son introducidos con más frecuencia, y por eso, es necesario la ejecución de regresiones lo más rápido posible.

- **Pruebas no deterministas (Flaky Tests).** Una prueba no determinista es aquella que podría generar un resultado positivo o negativo por la misma versión del software. Son pruebas que producen los llamados "falsos positivos" y por su inestabilidad son más conocidas como "flaky tests". Una de las características más importantes de CD es la confiabilidad, y pruebas que fallan aleatoriamente no son confiables.

- **Resultados de ejecución de pruebas ambiguos.** En entornos de desarrollo continuo, los desarrolladores deben ser notificados de cualquier defecto introducido, detectado a través de las pruebas. Cuando los resultados de la ejecución no son claros, es decir, no detallan el error, su causa, el lugar donde ocurrió, capturas de pantalla, etc. se tienen resultados ambiguos.

- **Pruebas en Big Data.** Big data es el proceso de utilizar grandes conjuntos de datos que no se pueden procesar utilizando técnicas tradicionales. Realizar verificaciones sobre este conjunto de datos es un nuevo reto que involucra varias técnicas y herramientas que aún están madurando, y por lo tanto es difícil incorporarlas en CD.

- **Pruebas de datos.** Los datos son muy importantes para diferentes tipos de sistemas y los errores en estos son costosos. Si bien las pruebas de software han recibido gran atención en diferentes niveles, las pruebas de datos han sido poco tenidas en cuenta. Es por ello, que no existen soluciones totales para la implementación de pruebas de datos en un entorno de CD.

- **Pruebas en dispositivos móviles.** Las pruebas móviles han traído consigo muchos desafíos. Entre ellos se encuentran: el proceso de pruebas en sí, los niveles y tipos de pruebas a considerar, los diferentes tipos de dispositivos y los costos de las pruebas automáticas. Todos estos factores dificultan la implementación de pruebas en dispositivos móviles en CD.

- **Pruebas de aplicaciones compuestas por servicios en la nube.** Actualmente, existe una gran cantidad de servicios en la nube, y el tamaño de los datos que deben manejar es muy grande. Probar flujos compuestos por estos servicios es muy complejo. Además, es necesario proporcionar garantías de calidad de servicio.

2 Cómo realizar Continuous Delivery

Según el ingeniero en software y experto en metodologías ágiles Martin Fowler [1], la clave para realizar un proceso exitoso de entrega continua es integrar continuamente el software realizado por el equipo de desarrollo, la creación automática de versiones instalables y la ejecución de estas pruebas automatizadas para detectar errores, todo a través de un flujo de tareas automatizado que se conoce como Continuous Delivery Pipeline.

Para lograr esta integración es necesario crear una estrecha relación de trabajo colaborativo entre todos los empleados implicados en la entrega, así como automatizar el mayor número de fases del proceso de entrega, definido a través del Deployment Pipeline.

2.1 Pruebas de software que se pueden realizar

2.1.1 Estrategias

En términos generales, hay dos estrategias que se pueden aplicar en la etapa de automatización de pruebas:

- **Caja Negra.** Es una técnica de prueba que ignora el mecanismo interno del sistema y se enfoca en la salida generada contra cualquier entrada y ejecución del mismo. También se llama prueba funcional y básicamente se usa para validar el software.

- **Caja Blanca.** Es una técnica de prueba que tiene en cuenta el mecanismo interno de un sistema. También se le llama prueba estructural o prueba de caja de vidrio y básicamente se usa para verificar el software.

2.1.2 Niveles de Testing

Según el nivel de granularidad con la cual se van probando diferentes porciones/secciones del producto de software, tenemos:

- **Unitario:** se le llama primer nivel de testing, y lo lleva a cabo el desarrollador a un nivel de componentes muy pequeño. Su objetivo no es encontrar defectos, sino que se busca garantizar que el requerimiento se comprende y la funcionalidad desarrollada hace lo que tiene que hacer.

- **Integración:** su objetivo es encontrar defectos, lo realiza un tester y se prueban componentes de granularidad más baja (es decir, más grandes). Apunta a la integración entre diferentes componentes, sistemas o subsistemas. Se lo conoce también como prueba de interfaces, porque el foco está en probar cómo encajan y funcionan los componentes entre sí.

- **Sistema o versión:** lo realiza el tester probando una versión del producto en condiciones de ser ejecutado para encontrar defectos. Para ello, el entorno de prueba debe corresponder al entorno de producción tanto como sea posible para reducir al mínimo el riesgo de incidentes provocados específicamente por el ambiente y que no se encontraron al realizar las pruebas.

- **Aceptación de usuario:** se entrega el sistema al usuario para que lo pruebe y determine si hace lo que él esperaba o no (validación). El objetivo es generar confianza en el sistema y en las características específicas y no funcionales del mismo, es decir, no encontrar defectos.

2.2 Tubería de implementación

Según uno de los capítulos del libro Continuous Delivery de Jez Humble y David Farley [3], se define, en un nivel abstracto, a una tubería de implementación como una manifestación automatizada de su proceso para llevar el software del control de versiones a las manos de sus usuarios. Cada cambio en su software pasa por un proceso complejo en su camino a ser lanzado. Es decir, el patrón clave en la entrega continua es la tubería de implementación, donde cada cambio ejecuta una compilación que crea paquetes que se pueden implementar en cualquier entorno y se ejecutan pruebas unitarias, además de otras tareas como el análisis estático, dando retroalimentación a los desarrolladores en el lapso de unos pocos minutos. Los paquetes que

pasan este conjunto de pruebas tienen pruebas de aceptación automatizadas más completas que se ejecutan en su contra. Una vez que tenemos paquetes que pasan todas las pruebas automatizadas, están disponibles para la implementación de autoservicio a otros entornos para actividades como pruebas exploratorias, pruebas de usabilidad y finalmente lanzamiento. Los productos y servicios complejos pueden tener tuberías de implementación sofisticadas.

En la tubería de implementación, cada cambio es efectivamente un candidato de lanzamiento. El trabajo de la canalización de implementación es detectar problemas conocidos. Si no se puede detectar ningún problema conocido, se debería lanzar cualquier paquete que haya pasado por él. Pero si se descubren defectos más adelante, significa que se necesita mejorar la tubería, quizás agregando o actualizando algunas pruebas.

El objetivo debe ser encontrar los problemas lo antes posible y hacer que el tiempo de espera desde el check-in hasta el lanzamiento sea lo más breve posible. Por lo tanto, se quiere paralelizar las actividades en la tubería de implementación, no tener muchas etapas ejecutándose en serie. También hay un proceso de retroalimentación, si se descubre errores en las pruebas exploratorias, se debería buscar mejorar las pruebas automatizadas. En cambio, si se descubre un defecto en las pruebas de aceptación, se debería buscar mejorar las pruebas unitarias, ya que la mayoría de los defectos deberían descubrirse mediante pruebas unitarias.

2.2.1 Etapas

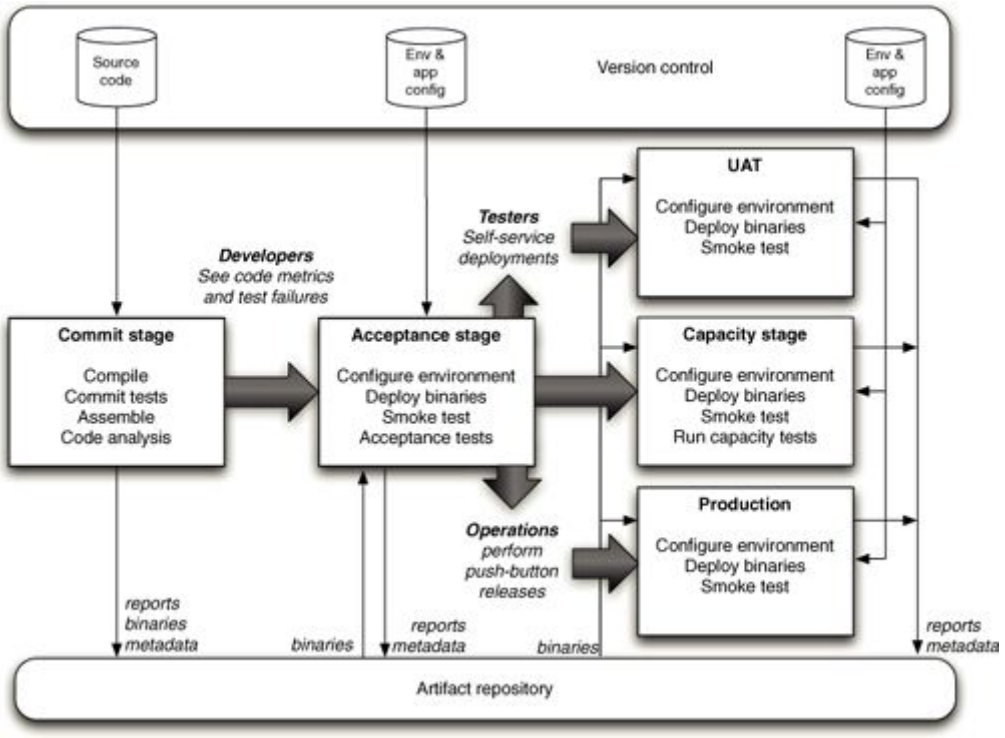
Las siguientes etapas conforman la tubería de implementación que es, fundamentalmente, un proceso automatizado de entrega de software. Esto no implica que no haya interacción humana con el sistema a través de este proceso de liberación, sino que brinda apoyo asegurando que los pasos propensos a errores y complejos sean automatizados, confiables y repetibles en la ejecución.

- La **etapa de confirmación (commit)** afirma que el sistema funciona a nivel técnico. Esto implica que el sistema se compila, pasa un conjunto de pruebas automatizadas (principalmente a nivel de unidad) y ejecuta análisis de código.

- Las **etapas de prueba de aceptación automatizada** afirman que el sistema funciona a nivel funcional y no funcional, es decir, que su "conducta" satisface las necesidades de sus usuarios y las especificaciones del cliente.

- Las **etapas de prueba manual** afirman que el sistema es utilizable y cumple con sus requisitos, detecta cualquier defecto no detectado por las pruebas automáticas y verifica que proporciona valor a sus usuarios. Estas etapas generalmente pueden incluir entornos de pruebas exploratorias, entornos de integración y UAT (pruebas de aceptación del usuario).

- La **etapa de lanzamiento** entrega el sistema a los usuarios, ya sea como software empaquetado o al implementarlo en un entorno de producción o de ensayo (un entorno de ensayo es un entorno de prueba idéntico al entorno de producción).



Tubería de Implementación básica. Libro: Continuous Delivery [3].

2.2.1.1 Etapa de confirmación (commit)

El objetivo de la primera etapa en la tubería es eliminar las construcciones que no son aptas para la producción y señalar al equipo que la aplicación se interrumpe lo más rápido posible. Se busca minimizar el gasto de tiempo y esfuerzo en versiones de la aplicación que no funcionan. Entonces, cuando un desarrollador realiza un cambio en el sistema de control de versiones, se intenta evaluar rápidamente la última versión de la aplicación, y el desarrollador espera los resultados de esa evaluación antes de pasar a la siguiente tarea.

Esta etapa comprende una serie de tareas que se ejecutan como un conjunto de trabajos en una cuadrícula de compilación para que la etapa se complete en un período de tiempo razonable. La etapa de confirmación idealmente debería demorar menos de cinco minutos en ejecutarse, y no más de diez. Además, incluye los siguientes pasos:

- ❏ Compilación del código (si es necesario).
- ❏ Ejecución de un conjunto de pruebas de confirmación.
- ❏ Creación de binarios para usar en etapas posteriores.
- ❏ Realización de un análisis del código para verificar su estado.
- ❏ Preparación de artefactos, como bases de datos de prueba, para su uso en etapas posteriores.

El primer paso es compilar la última versión del código fuente y notificar a los desarrolladores que cometieron cambios desde el último registro exitoso si hay un error en la compilación. Si

este paso falla, podemos fallar la etapa de confirmación inmediatamente y eliminar esta instancia de la tubería de una consideración adicional.

A continuación, se ejecuta un conjunto de pruebas, el cual debe estar optimizado para ejecutarse muy rápidamente. Este conjunto de pruebas también se conoce como “pruebas de etapa de compromiso” en lugar de “pruebas unitarias” porque, aunque la gran mayoría de ellas son pruebas unitarias, es útil incluir una pequeña selección de pruebas de otros tipos en esta etapa para obtener una mayor nivel de confianza de que la aplicación realmente funciona si pasa la etapa de confirmación. Estas son las mismas pruebas que los desarrolladores ejecutan antes de verificar su código (o, si tienen la posibilidad de hacerlo, a través de un compromiso previamente probado en la cuadrícula de compilación).

Se debe diseñar una prueba de confirmación ejecutando todas las pruebas unitarias. En el futuro, a medida que se aprende más sobre qué tipos de fallas son comunes en las ejecuciones de pruebas de aceptación y otras etapas posteriores en la tubería, se deben agregar pruebas específicas al conjunto de pruebas de confirmación para tratar de encontrarlas desde el principio. Esta es una optimización continua del proceso que es importante si se quieren evitar los costos de encontrar y corregir errores en las etapas posteriores de la tubería.

Establecer que el código compila y pasa las pruebas no brinda información sobre las características no funcionales de la aplicación. Probar características no funcionales, como la capacidad, puede ser difícil, pero se pueden ejecutar herramientas de análisis que brinden comentarios sobre características de la base de código como cobertura de prueba, mantenibilidad e infracciones de seguridad. Si el código no cumple con los estándares preestablecidos para estas métricas, la etapa de confirmación debe fallar de la misma manera que lo hace una prueba fallida. Las métricas útiles incluyen:

- ❑ Cobertura de prueba (si las pruebas de confirmación solo cubren un bajo porcentaje de la base de código no son muy útiles).
- ❑ Cantidad de código duplicado.
- ❑ Complejidad ciclomática.
- ❑ Acoplamiento aferente y eferente.
- ❑ Número de advertencias.
- ❑ Estilo de código.

El paso final en la etapa de confirmación es la creación de un ensamblaje desplegable del código listo para su implementación en cualquier entorno posterior. Esto también debe tener éxito para que la etapa de compromiso se considere un éxito en su conjunto.

2.2.1.2 Etapa de prueba de aceptación automatizada

El conjunto completo de pruebas de confirmación es una excelente prueba de fuego para muchas clases de errores, pero hay muchos aspectos que este conjunto no puede capturar. Las pruebas unitarias, que comprenden la gran mayoría de las pruebas de confirmación, están tan acopladas a la API de bajo nivel que a menudo es difícil para los desarrolladores evitar la trampa de probar que la solución funciona de una manera particular, en lugar de afirmar que resuelve un problema particular.

Las pruebas de confirmación que se ejecutan en cada registro brindan comentarios oportunos sobre problemas con la última compilación y sobre errores en la aplicación en pequeña escala. Pero sin ejecutar pruebas de aceptación en un entorno similar a la producción, no se sabe si la aplicación cumple con las especificaciones del cliente, ni si puede implementarse y sobrevivir en el mundo real.

El objetivo de la etapa de prueba de aceptación es afirmar que el sistema cumple con los criterios de aceptación y entrega el valor que el cliente espera. La etapa de prueba de aceptación también sirve como un conjunto de pruebas de regresión, verificando que los nuevos cambios no introduzcan errores en el comportamiento existente. El proceso de creación y mantenimiento de pruebas de aceptación automatizadas, no se lleva a cabo por equipos separados sino por equipos de entrega multifuncionales. Los desarrolladores, evaluadores y clientes trabajan juntos para crear estas pruebas junto con las pruebas unitarias y el código que escriben como parte de su proceso normal de desarrollo.

El equipo de desarrollo debe responder de inmediato a los fallos en las pruebas de aceptación que se producen como parte del proceso normal de desarrollo, y se debe decidir si el fallo de la prueba es el resultado de una regresión que se ha introducido, un cambio intencional en el comportamiento de la aplicación o un problema con la prueba. Luego, se deben tomar las medidas apropiadas para que el conjunto de pruebas de aceptación automatizadas vuelva a "pasar".

2.2.1.3 Etapa de prueba manual

En los procesos iterativos, las pruebas de aceptación siempre son seguidas por algunas pruebas manuales en forma de pruebas exploratorias, pruebas de usabilidad y exhibiciones. Antes de este punto, los desarrolladores pueden haber demostrado características de la aplicación a analistas y probadores, pero ninguno de estos roles habrá perdido el tiempo en una compilación que se sabe que no pasó las pruebas de aceptación automatizadas. El papel de un probador en este proceso no debe ser la prueba de regresión del sistema, sino, garantizar que las pruebas de aceptación validen genuinamente el comportamiento del sistema al demostrar manualmente que se cumplen los criterios de aceptación.

Después de eso, los evaluadores se centran en el tipo de pruebas en las que los seres humanos sobresalen, pero las pruebas automatizadas son deficientes. Se realizan pruebas exploratorias, pruebas de usuario de la usabilidad de la aplicación, se comprueba el aspecto en varias plataformas y se realizan pruebas patológicas en el peor de los casos. Las pruebas de aceptación automatizadas liberan tiempo para que los evaluadores puedan concentrarse en estas actividades de alto valor, en lugar de ser máquinas de ejecución de script de prueba humano.

2.2.1.4 Etapa de lanzamiento

Preparando para liberar. Existe un riesgo comercial asociado con cada lanzamiento de un sistema de producción. En el mejor de los casos, si hay un problema grave en el momento del lanzamiento, puede retrasar la introducción de nuevas capacidades valiosas. En el peor de los casos, si no existe un plan de retroceso razonable, puede dejar el negocio sin recursos de misión crítica porque tuvieron que retirarse como parte del lanzamiento del nuevo sistema.

La mitigación de estos problemas es muy simple cuando se ve el paso de lanzamiento como un resultado natural de nuestro proceso de implementación. Fundamentalmente, se quiere:

- ❏ Tener un plan de lanzamiento creado y mantenido por todos los involucrados en la entrega del software, incluidos los desarrolladores y probadores, así como el personal de operaciones, infraestructura y soporte.
- ❏ Minimizar el efecto de las personas que cometen errores al automatizar la mayor parte del proceso posible, comenzando por las etapas más propensas a errores.
- ❏ Ensayar el procedimiento a menudo en entornos de producción, para que pueda depurar el proceso y la tecnología que lo respalda.
- ❏ Tener la capacidad de retroceder un lanzamiento si las cosas no salen según lo planeado.
- ❏ Tener una estrategia para migrar datos de configuración y producción como parte de los procesos de actualización y reversión.

El objetivo es lograr un proceso de lanzamiento completamente automatizado.

Implementación y lanzamiento automáticos. Cuanto menos control se tenga sobre el entorno en el que se ejecuta el código, más posibilidades hay de comportamientos inesperados. Por lo tanto, cada vez que se lanza un sistema de software, se quiere tener el control de cada bit que se implementa, y hay dos factores que pueden funcionar en contra de este ideal.

El primer factor es que para muchas aplicaciones, simplemente no se tiene control total del entorno operativo del software que se crea. Esto es especialmente cierto para los productos y aplicaciones que instalan los usuarios, como juegos o aplicaciones de oficina. Este problema generalmente se mitiga seleccionando una muestra representativa de entornos objetivo y ejecutando su conjunto de pruebas de aceptación automatizadas en cada uno de estos entornos de muestra en paralelo. Luego se pueden extraer los datos producidos para determinar qué pruebas fallan en qué plataformas.

El segundo factor (o restricción) es que el costo de establecer ese grado de control generalmente se supone que supera los beneficios. Sin embargo, generalmente lo contrario es cierto: la mayoría de los problemas con los entornos de producción son causados por un control insuficiente. Los entornos de producción deben estar completamente bloqueados, los cambios en ellos solo deben realizarse a través de procesos automatizados. Eso incluye no solo la implementación de la aplicación, sino también cambios en su configuración, pila de software, topología de red y estado. Solo de esta manera es posible auditarlos de manera confiable, diagnosticar problemas y repararlos en un tiempo predecible. A medida que aumenta la complejidad del sistema, también lo hace el número de diferentes tipos de servidores, y cuanto mayor sea el nivel de rendimiento requerido, más vital se vuelve este nivel de control.

El proceso para administrar el entorno de producción debe usarse para los otros entornos de prueba, como la preparación, la integración, etc. De esta forma, se puede utilizar el sistema automatizado de gestión de cambios para crear una configuración perfectamente ajustada en sus entornos de prueba manual. Estos pueden ajustarse a la perfección, utilizando los comentarios de las pruebas de capacidad para evaluar los cambios de configuración que realice. Cuando se está satisfecho con el resultado, se puede replicar en cada servidor que se necesite esa configuración, incluida la producción, de una manera predecible y confiable. Todos los aspectos del entorno deben gestionarse de esta manera, incluido el middleware (bases de datos, servidores web, intermediarios de mensajes y servidores de aplicaciones). Cada uno se puede ajustar con la configuración óptima agregada a su línea de base de configuración.

Los costos de automatizar la provisión y el mantenimiento de entornos se pueden reducir significativamente mediante el aprovisionamiento y la administración automatizados de entornos, las buenas prácticas de administración de la configuración y (si corresponde) la virtualización.

Una vez que la configuración del entorno se gestiona correctamente, la aplicación se puede implementar. Los detalles de esto varían ampliamente dependiendo de las tecnologías empleadas en el sistema, pero los pasos son siempre muy similares.

Una razón importante para la reducción del riesgo es el grado en que el proceso de liberación se ensaya, prueba y perfecciona. Dado que utiliza el mismo proceso para implementar el sistema en cada uno de sus entornos y lanzarlo, el proceso de implementación se prueba con mucha frecuencia, tal vez muchas veces al día. Después de haber desplegado un sistema complejo sin problemas, ya no lo considera un gran evento. El objetivo es llegar a esa etapa lo más rápido posible. Si se quiere tener plena confianza en el proceso de lanzamiento y la tecnología, se debe usar y demostrar que es bueno de forma regular, al igual que cualquier otro aspecto del sistema. Debería ser posible implementar un solo cambio en la producción a través de la tubería de implementación con el mínimo tiempo y ceremonia posibles.

Retroceder cambios. Hay dos razones por las cuales los días de lanzamiento son tradicionalmente temidos. El primero es el miedo a presentar un problema porque alguien podría cometer un error difícil de detectar al seguir los pasos manuales de una versión de software, o porque hay un error en las instrucciones. El segundo temor es que, si falla la versión, ya sea por un problema en el proceso de liberación o por un defecto en la nueva versión del software, está comprometido. En cualquier caso, la única esperanza es que sea lo suficientemente inteligente como para resolver el problema muy rápidamente.

El primer problema se mitiga ensayando el lanzamiento muchas veces al día, demostrando que el sistema de implementación automatizado funciona. El segundo temor se mitiga al proporcionar una estrategia de retroceso. En el peor de los casos, puede volver a donde estaba antes de comenzar el lanzamiento, lo que le permite tomar un tiempo para evaluar el problema y encontrar una solución sensata.

En general, la mejor estrategia de retroceso es mantener disponible la versión anterior de su aplicación mientras se lanza la nueva versión, y durante algún tiempo después. La siguiente mejor opción es volver a implementar la buena versión anterior de su aplicación desde cero.

En ningún caso se debe tener un proceso de retroceso diferente al que se tiene para implementar, o realizar implementaciones o retrocesos incrementales. Estos procesos rara vez se probarán y, por lo tanto, no serán confiables. Siempre se debe retroceder manteniendo implementada una versión anterior de la aplicación o volviendo a implementar por completo una versión anterior conocida.

Construyendo sobre el éxito. Cuando haya un candidato de lanzamiento disponible para su implementación en producción, se deben conocer con certeza que las siguientes afirmaciones al respecto son ciertas:

- ❑ El código puede compilar.
- ❑ El código hace lo que nuestros desarrolladores piensan que debería porque pasó sus pruebas unitarias.

- ❑ El sistema hace lo que nuestros analistas o usuarios piensan que debería hacer porque pasó todas las pruebas de aceptación.
- ❑ La configuración de la infraestructura y los entornos de línea de base se gestionan adecuadamente, porque la aplicación se ha probado en un análogo de producción.
- ❑ El código tiene todos los componentes correctos en su lugar porque era desplegable.
- ❑ El sistema de implementación funciona porque, como mínimo, se habrá utilizado en este candidato de versión al menos una vez en un entorno de desarrollo, una vez en la etapa de prueba de aceptación y una vez en un entorno de prueba antes de que el candidato podría haber sido promovido a esta etapa .
- ❑ El sistema de control de versiones contiene todo lo que necesitamos implementar, sin la necesidad de intervención manual, porque ya hemos implementado el sistema varias veces.

3 Continuous Delivery vs Agile

Juan F. Samaniego [5], en su artículo, hace una pequeña distinción mencionando que la metodología y filosofía agile son anteriores a la entrega continua. Además, que han trascendido del entorno de startups y desarrollo de software para empezar a implementarse en todo tipo de negocios, donde ambos tienen más puntos en común que diferencias. No en vano, la entrega continua es una conclusión de la integración continua, una práctica de desarrollo de software que forma parte del entorno agile. Incluso, en una definición amplia de agile como filosofía de trabajo, la entrega continua podría llegar a considerarse parte de esta.

La única gran diferencia es que agile divide el tiempo de desarrollo de un producto en intervalo periódico, al final de los cuales el producto tiene que ser utilizable. Así, permite adaptarse fácilmente a las demandas del mercado. El paso de desarrollo a producción llega por una decisión humana y puede incluso hacerse de forma manual.

Sin embargo, la entrega continua busca automatizar todo el proceso, así el producto no es utilizable y estable al final de cada intervalo, sino en todo momento. Dependerá solo del cliente o del departamento de negocio de la organización decidir cuándo se pasa a producción.

4 Diferencias entre Continuous Integration, Continuous Delivery y Continuous Deployment

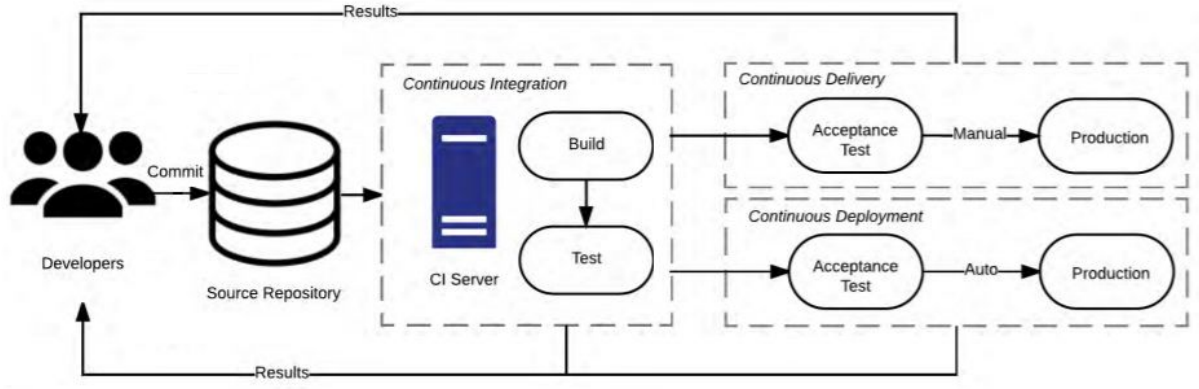
Según el paper publicado en la IEEEAccess de M. Shahin [7], podemos definir cada práctica de la siguiente manera:

Continuous Integration (Integración continua). La integración continua (CI) es una práctica de desarrollo ampliamente establecida en la industria del desarrollo de software, en la que los miembros de un equipo integran y fusionan el trabajo de desarrollo (por ejemplo, código) con frecuencia, por ejemplo, varias veces al día. CI permite a las compañías de software tener un ciclo de lanzamiento más corto y frecuente, mejorar la calidad del software y aumentar la productividad de sus equipos. Esta práctica incluye la creación y prueba de software automatizado.

Continuous Delivery (Entrega continua). La entrega continua (CDE) tiene como objetivo garantizar que una aplicación esté siempre en estado "listo para producción" después de pasar con éxito las pruebas automatizadas y los controles de calidad.

CDE emplea un conjunto de prácticas, por ejemplo, CI y automatización de implementación para entregar software automáticamente a un entorno productivo. Esta práctica ofrece varios beneficios, como un menor riesgo de implementación, menores costos y obtener feedback de los usuarios más rápido. Por lo tanto, tener una práctica de entrega continua requiere poseer una práctica de integración continua anterior.

Continuous Deployment (Despliegue continuo). La práctica de Despliegue continuo (CD) despliega la aplicación de forma automática y continua en entornos de producción o de clientes. Existe un fuerte debate en los círculos académicos e industriales sobre la definición y distinción entre despliegue continuo y entrega continua. Lo que diferencia el despliegue continuo de la entrega continua es un entorno de producción (es decir, clientes reales): el objetivo de la práctica de despliegue continuo es implementar de manera automática y constante cada cambio en el entorno de producción. Es importante tener en cuenta que la práctica de CD implica práctica de CDE, pero lo contrario no es cierto. Si bien la implementación final en CDE es un paso manual, no debe haber pasos manuales en CD, en los que tan pronto como los desarrolladores confirman un cambio, el cambio se implemente en producción. La práctica de CDE es un enfoque basado en extracción para el cual una empresa decide qué y cuándo implementar; La práctica de CD es un enfoque basado en el impulso. Y, si bien, la práctica de CDE se puede aplicar a todo tipo de sistemas y organizaciones, la práctica de CDE solo puede ser adecuada para ciertos tipos de organizaciones o sistemas.



Continuous integration, delivery and deployment. Paper de M. Shahin [7].

4.1 Cómo se relacionan las prácticas entre sí

En su libro, Sander Rossel [8] realiza un análisis, sobre cómo se relacionan las prácticas Continuous Integration, Delivery y Deployment entre sí, de la siguiente manera:

La integración continua trata de validar el software tan pronto como se registra en el repositorio fuente, intentando garantizar que el mismo funciona y va a continuar funcionando al agregar nuevo código. La entrega continua tiene éxito con la integración continua y hace que el software esté a solo un clic del despliegue. Luego de la entrega continua, viene el despliegue continuo y automatiza todo el proceso de puesta en producción de software para los clientes (o sus servidores).

Si la integración continua, la entrega y la implementación se pudieran resumir con una palabra, sería automatización. Las tres prácticas tratan sobre la automatización del proceso de prueba y despliegue, minimizando (o eliminando por completo) la necesidad de intervención humana, al igual que el riesgo de errores. Ésto hace más fácil la construcción y el despliegue del software hasta el punto en que cada desarrollador del equipo pueda liberarlo cuando sea necesario.

El problema con la integración, la entrega y el despliegue continuo es que no son nada fáciles de configurar y conllevan mucho tiempo, especialmente cuando nunca se los ha utilizado o se desean integrar a un proyecto existente. Sin embargo, cuando se hace correctamente, se amortizará al reducir los errores, lo que facilitará la reparación de los mismos cuando se los encuentre, produciendo un software de mayor calidad (que debería conducir a clientes más satisfechos).

Conclusión

La velocidad con la que avanza el mundo moderno crece cada vez más. Estos cambios, que abarcan ámbitos tecnológicos, sociales, empresariales, entre otros, de alguna manera nos obligan (y permiten) a desarrollar nuevas técnicas de trabajo que sean acordes a las nuevas necesidades del mercado. También es necesario aclarar que no siempre aparecen técnicas para "solucionar" problemas existentes, sino que en ocasiones, surgen ideas innovadoras que luego de ser implementadas ofrecen un gran ramo de posibilidades. Haciendo foco en esta idea podemos entender el mundo de la filosofía ágil.

Continuous Delivery ha llegado al mundo ofreciendo la posibilidad de eliminar todo trabajo humano que pueda automatizarse, para así centrar a los trabajadores en tareas de mayor valor, logrando exprimir el mayor recurso que tienen las organizaciones: las personas.

Las personas, como recurso de una organización, son el motor de ideas innovadoras que a corto y largo plazo permiten el real crecimiento como empresa. No es en vano que la filosofía ágil busque optimizar el trabajo en grupo, sino que es absolutamente necesario. Un grupo de trabajadores motivados es capaz de superarse a sí mismo continuamente, creando productos de mayor calidad en todo sentido. Para implementar Continuous Delivery (y cualquier metodología ágil) es necesario contar con dicha motivación. A su vez, la motivación no solo debe estar presente del lado de la empresa, sino también en los stakeholders. La disciplina de entrega continua aporta en gran nivel a la satisfacción del cliente, ya que le permite realizar un seguimiento tangible en cualquier momento.

Un problema notorio de Continuous Delivery es que no se adapta tan velozmente a las nuevas tecnologías, dejando sin posibilidad de participación a un gran número de situaciones. Como por ejemplo, la integración de dispositivos móviles al proyecto, las Flaky Tests, trabajos con Big data, etc. Pero esto, si bien es cierto, no afecta a la totalidad de las organizaciones; y no presenta una mayor preocupación a futuro, ya que Continuous Delivery es una disciplina en continuo progreso, sujeto a mejoras y escalable.

Referencias

Modelo de referencia: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=552620

[1] Martín Fowler, ingeniero de software británico, autor y orador internacional sobre desarrollo de software, especializado en análisis y diseño orientado a objetos, UML, patrones de diseño, y metodologías de desarrollo ágil, incluyendo programación extrema. https://martinfowler.com/bliki/ContinuousDelivery.html

Del libro "Continuous Delivery: Reliable software releases through build, test and deployment automation", escrito por Jez Humble y David Farley, se han tomado las siguientes referencias realizadas por:

[2] Jez Humble, donde realiza una reseña de su libro: https://continuousdelivery.com/

[3] Jez Humble y David Farley, donde se realizó un artículo sobre tubería de implementación en base a un capítulo del libro mencionado anteriormente: http://www.informit.com/articles/article.aspx?p=1621865&seqNum=2

[4] Javier Garzas, mentor Ágil, Ágil Coach, experto en gestión de proyectos y equipos, autor de un artículo sobre Continuous Delivery, Deployment and Integration: https://www.javiergarzas.com/2016/01/devops-continuous-delivery-continuous-deployment-integracion-continua-aclarando-terminos.html

[5] Juan F. Samaniego, que escribió un artículo sobre Continuous Delivery en https://hablemosdeempresas.com/empresa/entrega-continua-agile/

[6] Maximiliano A. Mascheroni y Emanuel Irrazábal, artículo "Problemas que afectan a la Calidad de Software en Entrega Continua y Pruebas Continuas": http://sedici.unlp.edu.ar/bitstream/handle/10915/73270/Documento_completo.pdf-PDFA.pdf?sequence=1&isAllowed=y

[7] Mojtaba Shahin, Muhammad Ali Babar y Liming Zhu, autores del paper "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices", aceptado y publicado por la IEEEAccess.

[8] Sander Rossel, desarrollador en la nube (Senior), revisor técnico y autor del libro "Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment".