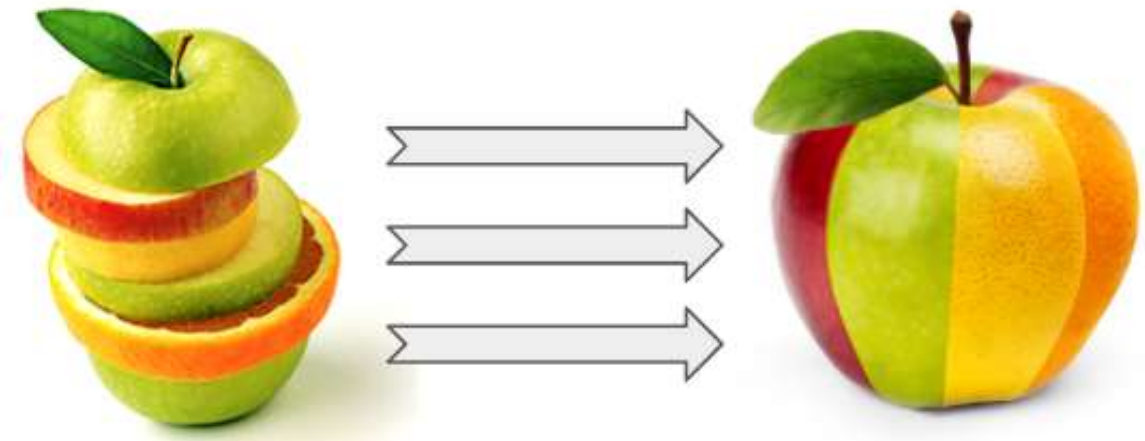
UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

DevOps, Integridad y Agilidad Continúa

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Unidad 3: Integración Continua



Presentación:

En esta unidad se identificarán las bases en las cuales se sostiene la práctica de Integración Continua, observaremos sus componentes y sus procesos para concientizar al alumno que en DevOps y sus prácticas frecuentemente utilizadas no sólo se trata de herramientas, sino que abarcan cambios en los equipos y en la organización. Además, se podrá apreciar la importancia de la automatización de las actividades manuales y repetitivas que realizan los integrantes de los equipos de elaboración de software.

Se logrará discernir gracias a los conceptos de las unidades anteriores la importancia de conocer los conceptos subyacentes y fundamentales para la gestión de un pipeline development.

Objetivos:

Que los participantes:

- Conozcan los requisitos básicos para la implementar la práctica de Integración Continua.
- Conozcan el proceso de la práctica de Integración Continua y sus beneficios.
- Identifiquen la importancia de la gestión de la configuración.
- Identifiquen y analicen los componentes para la automatización.
- Identifiquen los conceptos subyacentes un pipeline development.

Bloques temáticos:

1.¿Qué es y para qué practicar Continuous Integration? 8

2. Automatización (builds, integración de código, test) 14

3.Gestión de configuración 17

4. Pipeline Development 20

Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1. ¿Qué es y para qué practicar Continuous Integration?

En esta unidad daremos una introducción a la práctica de Integración Continua (CI, Continuous Integration). Al igual que en anteriores secciones donde recalcamos sobre el concepto de DevOps, el cual excede la instalación y configuración de herramientas de software, Integración Continua es una práctica y como tal hay ciertos aspectos que hay que tener en cuenta. Las herramientas de software por sí solas no proporcionan el cambio necesario para lograr que la organización adopte DevOps, así como tampoco por realizar la compra de una licencia de software que haga mención a Integración Continua, sin poseer los conceptos intrínsecos se estará destinado a resultados mediocres. Recordemos que DevOps es una filosofía que propone que una mayor colaboración en las áreas de desarrollo y operaciones, para diseñar, construir, desplegar y monitorear software de una manera más rápida, fiable, escalable y robusta.

La práctica de Integración Continua surge de la metodología XP (Extreme Programming), sí desean conocer más acerca de esta metodología ágil pueden obtener información del libro Extreme Programming Explained de Kent Beck el cual publicó su primera edición en 1999. Si bien la práctica de Integración Continua se está fomentando mucho más en los últimos años, no es algo nuevo. El concepto principal de esta práctica es que el código esté siempre en un estado estable para ser pasado a producción y la manera de que esto suceda sin demasiado riesgo y costo, es cuando se integra rápidamente código nuevo, asegurando a través de las pruebas automáticas que lo nuevo integrado no se están introduciendo fallas.

Parece bastante obvio, pero lo que generalmente sucede son dos escenarios, no son los únicos, pero si los que más he visto:

Uno, es que los equipos de desarrollo realizan sus cambios en repositorios de código distintos o bien en ramas de trabajo divididas por equipo (con un fuerte acoplamiento de código) por lo cual, al finalizar el trabajo, por ejemplo, de dos o más equipos se requiere que se integre lo desarrollado en un único repositorio o rama de código (branch) que los unifique. Este escenario es el que se conoce como **fase de integración** y conlleva a múltiples inconvenientes. El más sustancial es que en esta fase, en general, no se conoce cuando finaliza ya que los problemas que podrían existir por la integración son desconocidos hasta el momento de llevarla a cabo y esos inconvenientes la mayoría de las veces son costosos de resolver dado que se necesita retomar trabajo realizado con anterioridad (cambio de foco).

El segundo escenario, es que el trabajo de cada miembro del equipo se divide en ramas

distintas de código (branch por feature) o no integran sus cambios frecuentemente a la rama principal, por lo cual cuando se requiera unificar el trabajo de varias personas suele ser bastante similar a una fase de integración, pero con la posibilidad de volver los cambios hacia atrás, en el caso de que estén en un mismo repositorio.

Para decir que un equipo está realizando Integración Continua, se deben basar en una serie de reglas y acuerdos de equipo. Se requieren de tres requisitos para la realización de esta práctica:

1. **Tener un repositorio de control de versiones** (GIT, Mercurial, SVN, etc.). Con respecto de poseer un repositorio de control de versiones se hizo mención en la unidad anterior por lo que no nos detendremos a profundizar en este tema. Lo que hay que tener en cuenta es que todo proyecto, por más que se crea que es pequeño, debe tener un repositorio de control de versiones y de ser posible versionar todo lo que incluye el proyecto: código productivo y de pruebas, scripts de base de datos, scripts de deploy, documentación, etc.).

2. **Poder generar un build (compilación de la solución del código) de manera automática.** La automatización es fundamental, pero en lo posible debe ser independiente del software que utilizan los programadores para el desarrollo (IDE). Esto no quiere decir que no se utilicen, por lo contrario, úsenlos ya que tienen una versatilidad importante, pero se debe prestar atención a que la ejecución del proceso de Integración Continua sea posible instanciarlo desde múltiples entornos, preferentemente desde la línea de comandos.

3. **Tener un acuerdo con el equipo**. Tal como se mencionó anteriormente Integración Continua es una práctica que requiere de compromiso y disciplina por parte del equipo. En el acuerdo se debe incluir que los miembros del equipo de desarrollo realicen al menos un commit al día y lo publiquen, es decir integrando código continuamente. La prioridad más alta del equipo es que si hay un fallo en la línea principal dejaran lo que están desarrollando para solucionar el inconveniente. Si no existe un compromiso por parte del equipo es poco probable que se obtengan los beneficios de calidad que provee esta práctica.

Hasta ahora, lo que hemos visto son temas de surgimiento de esta práctica, sus

requisitos, lo interesante como también hemos mencionado es que la Integración Continua no se refiere sólo al software propiamente dicho, sino a las actividades que deben desenvolver los integrantes del equipo de desarrollo para que esta práctica sea efectiva. Para ello mencionaremos algunos puntos claves.

- **Commits frecuentes**:

  Integración continua quiere decir integrar muchas veces el código. Si un equipo dice hacer integración continua y hacen commit una vez por semana o cada persona desarrolla en ramas diferentes (Branch por Feature individual) e integran al finalizar una iteración, eso no es CI. Será otra práctica, pero no es integración continua. Mientras más seguido hagamos el commit más continuamente estaremos integrando código, mientras más frecuente sea este cambio por cada desarrollador menor será la porción de código que subirá y de tener algún error menor será el costo ya que hacer un rollback (volver al estado anterior al error) será rápido y menos doloroso ya que serán menos cantidad de líneas. Este concepto nos remite nuevamente a considerar al desarrollo con la noción ya vista de desarrollo orgánico.

- **Tener una batería automatizada de pruebas**:

  Este punto es de mucha importancia. Integrar el código no significa sólo que los cambios de los integrantes del equipo se integren (merge) y compile la aplicación (build), aunque como inicio es un buen comienzo. Lo importante es tener una batería de pruebas que se ejecuten al realizar un cambio, pero antes de que se haga el merge en el repositorio de control de versiones.
  Este conjunto de pruebas actúa como una red de aseguramiento de calidad, ya que un cambio introducido podría provocar la falla de la compilación de la aplicación, pero al verificar que las pruebas fallan, por ejemplo, test unitarios o test de componentes, dicho cambio debe ser rechazado.
  La ventaja que nos dará tener esta batería de pruebas automatizadas es tener feedback instantáneo al realizar un cambio. Si falla una prueba, el desarrollador sabrá exactamente en qué lugar tiene que revisar y qué cambio fue el que produjo la falla.

- **Mantener el proceso de compilación y el conjunto de pruebas en un tiempo corto**:

  Este punto es una recomendación muy útil, ya que hay que tener presente cuánto

demora en ejecutar la compilación y la batería de pruebas en el proceso de integración continua. Lo recomendable es que no demore más de diez minutos, dado que, si el proceso demora mucho tiempo, los desarrolladores tenderán a evitar incorporar cambios frecuentemente o bien se podrían encolar múltiples cambios provocando que el build falle, con lo cual dificultará determinar qué cambios provocaron la falla.
Para reducir los tiempos de compilación y ejecución de pruebas, se suele dividir el proceso de integración continua en distintas etapas, ambientes y ejecuciones paralelas de análisis de código estático.

- **Los desarrolladores deben administrar sus entornos**:

  La aplicación debería poder ejecutarse en los entornos locales de los desarrolladores y estos a su vez deberían poder ejecutar el conjunto de pruebas automatizadas en su propio ambiente. Si el desarrollador no posee una administración completa de su entorno, ante un fallo se podría hacer un cuello de botella al esperar que venga la persona encargada de darle los permisos necesarios para que lo solucione o bien para que tome cartas en el asunto.
  Por otro lado, al ejecutar el proceso de integración continua, el desarrollador debe poder partir de un punto común conocido. Esto quiere decir que cada vez que un desarrollador obtiene el código para trabajar será desde un punto en donde se ejecutaron todas pruebas necesarias y todas pasaron, es decir que la aplicación está en un estado estable. Este tipo de práctica es altamente recomendable ya que tiende a reducir ampliamente los fallos en el servidor centralizado, ahorrando tiempo al equipo entero de desarrollo.

- **No integrar sobre un build que falló**:

  Si ocurre un fallo al momento de publicar un cambio se debe prestar total atención a que se solucione por la persona que subió dicho cambio. Si dicha persona no lo puede solucionar en un lapso corto (tiempo acordado por el equipo) el desarrollador debe hacer un rollback para dejar la aplicación en un estado saludable o bien pedir ayuda al equipo para solucionarlo, pero nunca dejar que los builds fallen y acostumbrarse a que estén así. Debe ser regla mantener en estado saludable el código de la aplicación.

Por último, la recomendación es que, si se está por comenzar un proyecto nuevo, la

incorporación temprana de un ciclo de Integración Continua es lo menos doloroso y costoso. La implementación Integración Continua en una aplicación heredada (legacy) es laboriosa pero no imposible, lo cual igualmente recomiendo su incorporación ya que la organización poseerá más visibilidad sobre los errores (de los cuales podrá obtener métricas), el equipo incrementará su velocidad poco a poco y la calidad tenderá a aumentar gradualmente generando más confianza en el producto. El proceso de integración y despliegue de software no debe ser riesgoso y altamente costoso. Si lo es, la recomendación es que mientras más doloroso sea más veces hay que intentar implementar la práctica para revertir el síntoma.

2. Automatización (builds, integración de código, test)

Imagen obtenida de https://www.hfsresearch.com

El movimiento DevOps es reconocido por la instauración de herramientas de software para la automatización de pasajes de código en ambientes y las pruebas, pero en el comienzo del camino hacia la implementación de las prácticas de Integración y/o Entrega Continua se debe prestar especial atención a lo que se va a automatizar.

Lo primero que sugiero automatizar es el build de la aplicación y el deploy a un ambiente distinto del desarrollo. Debe ser posible de configurar la aplicación para que el servidor de Integración Continua reciba notificaciones de cambios sobre un determinado branch de un repositorio de versionado de código y que el desarrollador al ejecutar un commit realice el merge y el build.
Al generar el build se obtendrán los binarios/ejecutables de la aplicación los cuales se copiarán a otro ambiente distinto del servidor de Integración Continua y lo más parecido a uno de producción en el cual se le aplicará la configuración necesaria para que la aplicación logre ejecutarse. De tener más ambientes para distintos propósitos, la automatización que se escoja debe permitir que al transferir el binario a los distintos ambientes no sea necesario recompilar (**y no se debe hacer**) sino que hay que aplicar distintas configuraciones dependiendo del ambiente en el que se realice el despliegue. Uno de los primeros pasos para enfrentarse a los dolores de los despliegues, es generar una aplicación básica al estilo “hola mundo” y se haga el despliegue al ambiente (esta tarea es conocida como deploy técnico) para comprobar lo más pronto posible las configuraciones que es necesario ajustar. Todas las advertencias y errores que se fueron detectando a lo largo de la instalación y configuración es muy importante que sean documentadas para futuras ocasiones.

Una vez logrado el merge y la compilación automática del código, el paso siguiente es la automatización de las pruebas unitarias y el análisis de código. La configuración de las pruebas unitarias no requiere mucha complejidad, es un paso más al procedimiento de automatización de la compilación y despliegue que se realiza antes de que se realice la integración de código. Recordemos que estas operaciones generalmente se realizan en el servidor de Integración Continua. Luego es conveniente instalar alguna aplicación que pueda leer el código fuente y realice un análisis de “salud” del mismo. Según el grado de

madurez que se tenga con la práctica de Integración Continua en la organización y en el equipo en la cual se esté llevando a cabo se deberán configurar valores que puedan ser alcanzables para métricas de análisis de código como por ejemplo “complejidad ciclomática”, “posibles vulnerabilidades”, y algunas otras muy útiles. Esto quiere decir que el no alcanzar los índices de los parámetros de análisis de código se podría considerar como si hubiese fallado la compilación.

El paso siguiente, es configurar la automatización de las pruebas de aceptación. En esta etapa se ejecutarán las pruebas que fueron definidas tanto para el equipo como para el usuario, las cuales nos aseguran que nuestro sistema continúa funcionando sin importar del constante cambio. La automatización debería abarcar no sólo pruebas funcionales (la que entrega valor al negocio) sino también pruebas no funcionales tales como pruebas de capacidad, pruebas sobre el consumo de servicios, pruebas de seguridad, pruebas de fallas de infraestructura, etc. Comenzar con un conjunto, aunque sea pequeño, de estas pruebas al comienzo del desarrollo del pipeline es de gran beneficio.

Es necesario diseñar una arquitectura de solución que posibilite la automatización del conjunto de datos necesarios para que la aplicación funcione y para que las pruebas puedan ser ejecutadas. La mayoría de los pequeños programas que construyen los desarrolladores para poder probar su desarrollo son candidatos a que sean scripts automatizables y que sean incorporados idealmente al ciclo de Integración Continua. Así mismo, las actualizaciones a la base de datos debieran estar bajo el mismo sistema de control de versiones y que la incorporación de algún cambio se propague con algún script de automatización. En este curso no abordaremos el tema de la gestión de los datos en la práctica de Integración Continua ya que es un tema por demás extenso y a su vez la implementación de esta característica depende de la arquitectura de la aplicación en cuestión. En ocasiones, es posible versionar el cambio que se realiza en la base de datos (por ejemplo, el producto Flyway), pero si esta se encuentra en un producto que ya está en producción es más complejo volver el cambio atrás.

La ventaja de contar con un proceso automatizado es que ante cualquier error será más visible su origen y cualquier procedimiento automático será menos propenso a errores que si fuera manual. Tomemos por ejemplo los escenarios en los que una organización realiza los despliegues a producción de manera manual. Hay un alto riesgo de introducir errores, con lo que es más costoso detectarlo ya que si fuese automático existiría un registro del estado de cada operación posibilitando determinar la causa del inconveniente que se haya producido (menor tiempo de recolección de datos y diagnóstico).

En DevOps se busca eliminar cualquier paso manual que sea repetitivo y/o que posea riesgo para la organización, sobre todo los que son propensos a introducir errores. Tal como se ha mencionado anteriormente, es útil obtener métricas de tareas o pasos del proceso para poder saber lo que hay que optimizar, incluidas las tareas manuales y con ello se tendrá información suficiente para diseñar una estrategia de mejoras.

3.Gestión de configuración

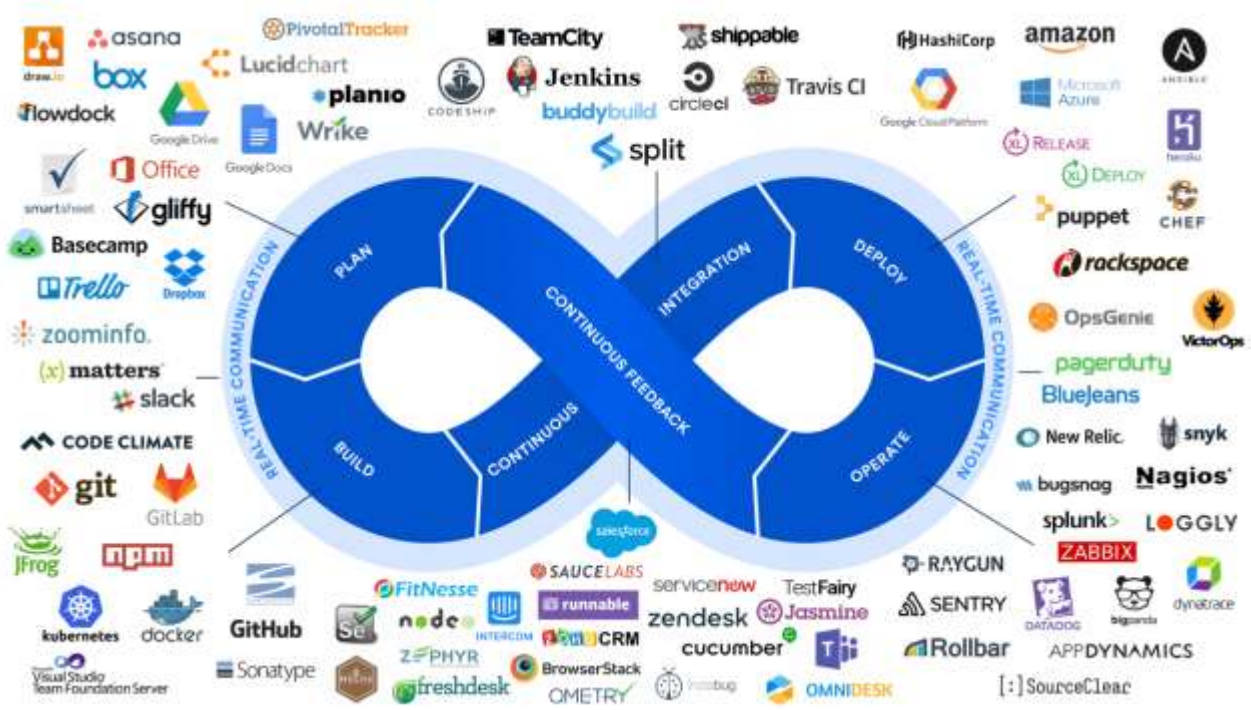
Una vez que el sistema operativo en donde nuestra aplicación habitará ya se encuentra totalmente instalado y configurado con su respectivo aprovisionamiento de software, lo que usualmente sucede es que el mantenimiento de este ambiente ya sea el de desarrollo, el de pruebas o bien el de producción, se realiza manualmente. De esta intervención manual en el mantenimiento, y en una instalación desde el inicio, provienen, en su mayoría, los problemas de ambientes.

Si se introducen cambios no controlados la detección de un error puede tomar horas y requerirá un esfuerzo enorme por parte del personal de infraestructura / operaciones o de los desarrolladores para lograr descubrir su origen y luego corregirlo, y con suerte esa corrección se documentará (muchas veces lo que se plasma en los documentos no fue lo que se hizo) y se trasladará al resto de los ambientes que conforman el ciclo de desarrollo del equipo. Este tipo de escenario es el que más se suele dar en las organizaciones.

La intervención manual para el aprovisionamiento de configuración debe hacerse a través de un proceso automatizado y controlado. De ser posible, debe tratarse de un proceso tal que no se puedan aplicar cambios manuales en los ambientes o al menos en los más críticos. La meta de poseer una gestión de configuración automatizada de nuestros ambientes y de nuestra aplicación mediante un único proceso, es que nos posibilite ante cualquier circunstancia partir de un estado “saludable”, estable y conocido.

Debe ser posible llevar un seguimiento de los cambios que se apliquen en la configuración de nuestra aplicación o en la infraestructura de los ambientes. Cada cambio aplicado debe ser auditado mediante un control automatizado para prevenir errores humanos. Estos cambios pueden variar desde configuraciones en un firewall hasta una pequeña configuración de registro de errores de nuestra aplicación. Además, el cambio aplicado debe estar probado primero en algún ambiente similar al de producción, luego incorporarlo a un sistema de control de versiones, y por último incorporarlo a un proceso automático de gestión de cambios con alguna prueba que verifique que dicho cambio es efectivo. Hacer el despliegue de cambios de configuración e infraestructura de una manera automatizada a través de un sistema de control de versiones es signo de un buen sistema de gestión de cambios de configuración.

4. Pipeline Development



El concepto de pipeline development se refiere a una abstracción del proceso de desarrollo, pruebas e implementación de software desde su versionado de código hasta el usuario que lo utilizará. El pipeline de desarrollo (development) implica que cada cambio introducido desde que llega un requerimiento pasará por distintas etapas, en distintos ambientes de desarrollo, ambientes de prueba, ambientes de producción, distintas personas y hasta distintos equipos o áreas de la organización, con lo cual hay que prestar especial atención sobre este concepto y verlo de una mirada holística, es decir desde el principio a fin para optimizar su totalidad (visión Lean del proceso). Poseer un pipeline de desarrollo automatizado nos aportará una visión de la que tendremos la posibilidad de medir las distintas partes que componen el proceso de desarrollo y nos permitirá obtener feedback mucho más rápido y certero, así mismo al estar automatizado podremos repetir el proceso tantas veces como se quiera aportando más confiabilidad y minimizando el riesgo de implementar en producción.

Esto no es sólo un concepto más para visualizar y medir. Se debe tener la mirada sobre el pipeline de desarrollo y presentarlo como herramienta de posibilidad de mejora continua (por ejemplo, mediante una herramienta Lean como Value Stream Map), para introducir cambios en la organización. En las culturas embebidas con el movimiento de DevOps poseen este tipo de enfoques para aplicar la mejora continua en la entrega de producto al cliente con la mejor calidad y en el menor tiempo posible. Buscan la eficiencia de sus productos a través de la colaboración de todas las partes.

El pipeline de desarrollo se compone de distintas etapas y ambientes. Estos, comúnmente se dividen en cuatro, respetando el siguiente orden: **Commit** (Desarrollo), **Pruebas automáticas de aceptación**, **Pruebas Manuales** (QA) y **Producción**. Es importante entender que la separación en cada etapa cumple distintos objetivos de pruebas con lo que dará distintos resultados, y cada una de las etapas está encadenada a la otra. No se debería poder pasar a **QA** si no se ha pasado primero por commit.

Durante la etapa de **Commit** lo que se establece es la ejecución del acuerdo realizado por el equipo de desarrollo sobre la práctica de Integración Continua. Los desarrolladores deberían hacer una publicación al menos una vez al día. Apenas suceda la publicación el servidor que posee el software, para modelar el proceso de Integración Continua, orquesta una serie de pasos similares a los siguientes: estará escuchando novedades en un determinado branch y determinará si el merge del cambio que introdujo el desarrollador es exitoso, realizará un build de la solución, ejecutará la batería de pruebas unitarias (y de ser posible alguna pequeña prueba que implique un escenario punta-punta) y generará los

binarios para hacer el deploy en el resto de los ambientes. Si alguno de estos pasos falla se debe considerar que el cambio no es válido, por lo que según lo acordado con el equipo se debe dejar lo que se esté haciendo para solucionar el inconveniente y dejar la aplicación en un estado “saludable”. Es muy recomendable que los desarrolladores posean administración del ambiente del servidor de Integración Continua para que puedan solucionar rápidamente cualquier inconveniente que se pueda presentar. No recomiendo que los desarrolladores utilicen un único ambiente para desarrollar, un síntoma de una buena arquitectura es que el desarrollador esté en condiciones de instalar y ejecutar la aplicación en su entorno local (por supuesto que con algunas limitaciones).

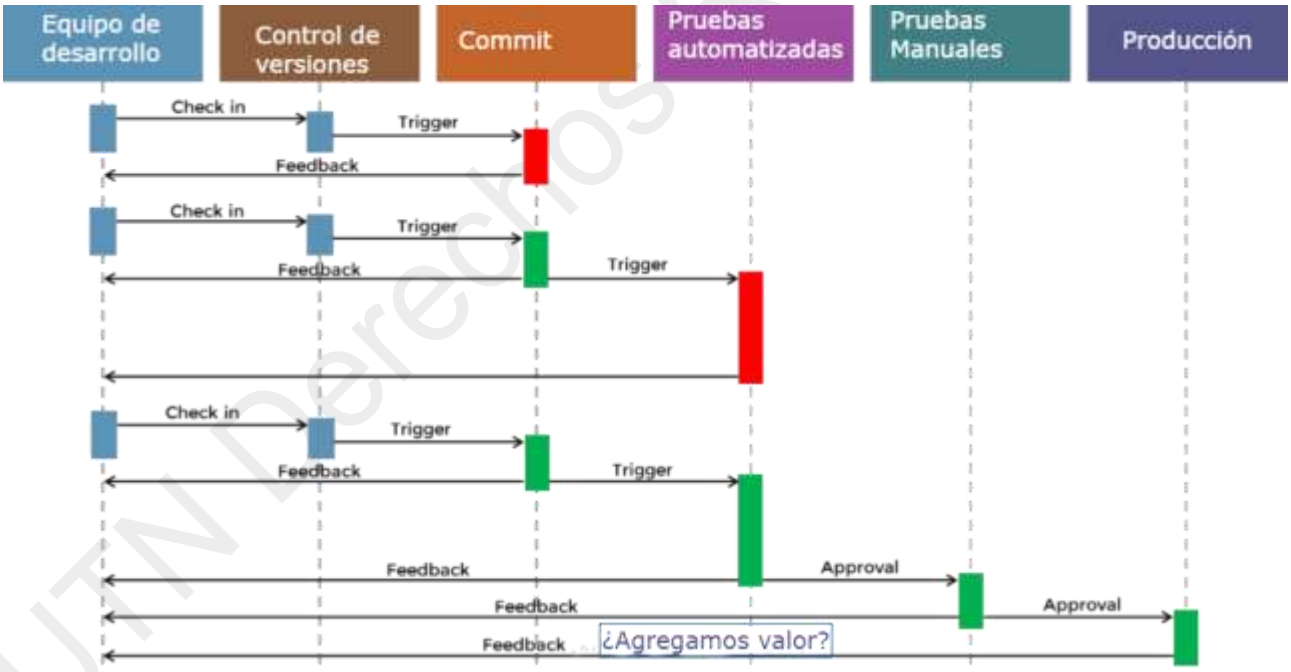
Además, es muy recomendable que al publicar un cambio se ejecute un análisis del código fuente para determinar métricas tales como: repetición de código, posibles vulnerabilidades, complejidad ciclomática o estándares de estilo de escritura. Si el equipo y la organización poseen un buen nivel de madurez se podría considerar que ante un nuevo cambio que se introduce se dispara el umbral de alguno de los parámetros antes descritos se pueda tomar como invalido el cambio. Como se mencionó anteriormente, la recomendación es que la duración máxima de esta etapa no supere los diez minutos.

En la etapa de **Pruebas automatizadas de aceptación** se darán otro tipo de pruebas en un ambiente distinto más similar a uno de producción. Para que un cambio pueda ser promovido hasta aquí tuvo que haber pasado exitosamente por la etapa de **commit** y en general se ejecuta automáticamente, dependiendo de la magnitud del proyecto suele demorar entre treinta minutos y dos horas. Las pruebas que se realizan son funcionales y no funcionales (por ejemplo, comprobación de servicios, de seguridad, etc.). Estas pruebas son todos los test de aceptación de usuario que se confeccionaron para testing automático. Pueden ser por ejemplo pruebas que accedan a archivos, registros de base de datos, que consuman datos de algún servicio, o similares. Esta etapa es utilizada como test de regresión para asegurar que los cambios introducidos no afecten otras partes y que la aplicación continúe funcionando como se espera. Este tipo de pruebas es fundamental ya que se ejecutan regresiones continuamente proveyendo feedback rápido al equipo de elaboración y por sobre todas las cosas no se estará pasando defectos a producción.

En la etapa de **QA** se darán otro tipo de pruebas en un ambiente distinto el cual es mayormente utilizado por los testers, Business Analysts o Dueños de Producto. Para que un cambio pueda ser promovido hasta aquí tuvo que haber pasado exitosamente por la etapa de **Pruebas automatizadas de aceptación**. En un equipo y en una organización con la suficiente madurez enfocada en DevOps, los testers serán quienes busquen qué versión

candidata probar y mediante un proceso automatizado podrán hacer el deploy de la versión deseada a este ambiente de pruebas. Si falla en este momento la aplicación tampoco se encuentra en un estado saludable para ser lanzada a producción por lo que el tester rechazará el cambio y se deberá corregir comenzando desde la etapa inicial. Las pruebas que se realizan en esta etapa son manuales e incluyen pruebas funcionales que tal vez no están del todo automatizadas, pruebas exploratorias y pruebas no funcionales como podrían ser las pruebas de estrés (si el presupuesto lo permite, generalmente se ejecutan en un ambiente distinto), pruebas de capacidad, etc.

La etapa final es **Producción**, aquí la falta de madurez que tiene la organización en el manejo de los deploy es donde generalmente se encuentra el mayor dolor y costo, tanto para la organización como para el cliente. Si se posee un presupuesto lo bastante amplio, lo ideal es tener una copia fiel del ambiente de producción para ejecutar pruebas sobre la topología de red, configuración del firewall, versiones del sistema operativo y aprovisionamiento de software y la validez de los datos que se utilizarán.



Una práctica recomendada para el caso de que se tengan binarios de la aplicación es que estos se compilen por única vez en la etapa de **Commit**, distribuyéndolos a lo largo del resto de los ambientes, incluido **Producción**. Además de hacer el deploy de los mismos binarios, se debería ejecutar el mismo proceso de deploy de un ambiente a otro, también

incluido el ambiente de **Producción** para asegurar que no sólo estaremos probando la aplicación sino este mismo proceso de deploy desde principio a fin, dándonos la confianza suficiente y minimizando los riesgos de salida a producción. Lo que va a variar en el deploy de cada ambiente seguramente sean las configuraciones y los datos que se utilizaran, pero no el ejecutable de la aplicación.

La implementación del pipeline de desarrollo es recomendable que sea definido y construido de manera incremental, no es necesario que esté terminado todo de una vez. Las tareas que se comiencen a automatizar deben registrar y almacenar el tiempo que demoran en ejecutarse, así como también las diferentes etapas que atravesará un requerimiento a lo largo de todo el proceso del pipeline, incluyendo aquellos que aún no tenemos automatizados. Obtener estas métricas nos permitirá tomar decisiones sobre las tareas o etapas en nuestro ciclo de desarrollo para tomar acciones sobre lo que es necesario optimizar. Por esta razón es muy importante que midamos cuánto demora nuestro proceso manual para que luego podamos determinar la eficiencia que tiene, y luego poco a poco se implemente un único proceso y modo de llevar nuestra aplicación de ambiente en ambiente de manera automática.

Bibliografía utilizada y sugerida

- Alex Williams. The Docker & Container Ecosystem. The New Stack.
- Jez Humble, David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. 1era edición. EE:UU: Addison Wesley; 2010
- Paul M. Duvall, Steve Matyas, Andrew Glover. Continuous Integration: Improving Software Quality and Reducing Risk. 1era edición. EE:UU: Addison Wesley Signature Series;2007

Lo que vimos:

En la unidad vista se pudo apreciar las principales características y principios de la práctica de Integración Continua, las cuales consisten en una concientización más amplia que la instalación de un servidor que automatice el pipeline.

Dimos una reseña de la importancia que cobra la automatización de las actividades del pipeline de desarrollo, con sus diferentes etapas, para construir software de calidad, con despliegues rápidos entre ambientes, incluido producción, y de bajo riesgo. Dentro de las actividades que se incluyen en la automatización del pipeline de desarrollo, la gestión de la configuración es un tópico al que hay prestar especial atención para no caer en problemas recurrentes de ambientes por diferencias de configuración al no tener un efectivo control de los cambios.

Lo que viene:

En la siguiente unidad se hará una reseña de las características y ventajas de la práctica de Continuous Delivery. Se abordarán de manera introductoria distintas estrategias para realizar esta práctica y la importancia de tener la infraestructura en un sistema de control de versiones para abordar otra fluidez en las entregas de una organización.