Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso - Chile

Diseño e Implementación de un Sistema de Integración y Entrega Continua con Jenkins

Francisco Javier Arcis Seguel

Memoria para optar al título de Ingeniero Civil Informático

Profesor Guía: José Luis Martí Lara

Marzo 2017

DEDICATORIA

A mi familia por ser la fuente de inspiración para que esto sucediera...

A mi pareja por su amor y por estar siempre apoyándome.

A todos mis amigos en especial a aquellos que me presionaron para que llevara esto adelante.

RESUMEN

Resumen—El presente documento busca demostrar cómo aplicar e implementar un sistema de Integración y entrega continua en el área TI de una empresa, con el fin de apoyar la metodología ágil que dicha área utiliza. Los resultados serán llevados a cabo con la herramienta Jenkins, demostrando las ventajas de un proceso automatizado y dinámico posible de construir en esta plataforma y que apoya directamente los objetivos de una metodología ágil. Los efectos se analizarán comparando, a grandes rasgos, cómo se mejoraron los procesos frente a una metodología convencional como la cascada.

Palabras Claves—Metodología de desarrollo Ágil, Integración Continua, Entrega Continua, Jenkins.

ABSTRACT

Abstract— This document seeks to demonstrate how to apply and implement a system of Integration and Continuous Delivery in the IT area of a company, in order to support the agile methodologies that area uses. The results will be carried out with the Jenkins tool, mainly seeking to demonstrate the advantages of an automated and dynamic process possible to build in this platform and that directly supports the objectives of an agile methodology. The results will be analyzed by comparing, broadly, how the processes were improved against a conventional methodology such as the cascade.

Keywords—Agile Software Development, Continuous Integration, Continuous Delivery, Jenkins.

1 DEFINICIÓN DEL PROBLEMA

La unidad bajo la cual se fundamentó este estudio, es el área de TI de una gran empresa perteneciente al rubro de la aeronáutica y transportes, cuenta con varios equipos de desarrollo a cargo de distintos servicios de la empresa. El flujo de trabajo se basaba en un Modelo Cascada (MC), se ocupaba un lenguaje no muy robusto y los tiempos que tardaban en subir a Producción variaban entre una a dos semanas.

La gerencia propone como objetivo transformarse en una de las empresas más importantes del rubro. Se analiza la situación actual y dentro de las medidas adoptadas se planea una renovación de los servicios TI, al considerarse un recurso crítico para alcanzar el objetivo propuesto. Bajo este escenario es que se adopta estratégicamente una renovación del área tanto en sus metodologías como en sus herramientas.

1.1 Contexto

Actualmente, es común encontrarse con empresas u organizaciones dedicadas al desarrollo de software que arrastran prácticas y metodologías de trabajo orientadas a un mundo donde el cambio de reglas o requerimientos no es una opción. En esta línea es ampliamente conocido el Modelo Cascada, el cual se basa en una serie de etapas consecutivas, donde cada una de éstas no puede empezar sin antes haber finalizado la anterior. Pero el cambio y lo inesperado es parte del desarrollo de software, aunque no es lo ideal, y esto ha llevado a crear interpretaciones propias del Modelo Cascada que han evolucionado sobre el camino y que buscan, al fin y al cabo, otorgar un poco más de flexibilidad al proceso de desarrollo.

La situación inicial del área de TI de la organización en análisis, es bien representaba en la descripción anterior. Básicamente, trabajan con un Modelo Cascada y un solo lenguaje de programación que resultaba ser poco robusto. Esto los enfrentaba a una serie de obstáculos en el desarrollo, como:

- Poca capacidad de flexibilidad ante el cambio de requerimientos: si no existía claridad de lo que se quiere desde un principio, se hacía muy difícil y costoso tomar requerimientos de último minuto.

- No existía la opción de volver atrás: era muy importante que cada fase terminara de manera perfecta y sin errores, pues de lo contrario estos harían eco en las etapas posteriores, lo cual complicaba las subidas a Producción.
- *Feedback* tardío: hasta que se completaba la última etapa el cliente no podía tener en sus manos el software; por lo tanto cualquier *feedback* del resultado obtenido era tardío y dejaba a los equipos con dos opciones, se generaba un pendiente de desarrollo con una fecha incierta o el equipo lo tomaba como parte del soporte del producto.

En cuanto a lo que el área de Operaciones prácticamente carecían de procesos automatizados. Las subidas a Producción eran una tarea que involucraba a todos los equipos del área. Antes de subir había que agendar una reunión entre todos los integrantes del área TI, en la práctica esta reunión ocurría una vez por mes marcando una subida por mes a Producción (para los equipos que tuvieran su código listo para la reunión mensual sino había que esperar hasta el mes siguiente) con el objetivo de discutir qué códigos se tocaron y cuáles eran los posibles conflictos que podían generar estos *merges,* respecto al trabajo que estuvieran haciendo otros equipos. En fin había que coordinar entre todos miembros del área las subidas a producción lo cual llevaba mucho trabajo. El proceso manual de *deploy* en producción también era muy delicado, raramente no ocurrían errores. Todos estos factores hacían del proceso de *deploy* a producción, en definitiva, una tarea muy lenta y delicada. En resumen cada equipo de desarrollo se enfrentaba a los siguientes obstáculos:

- No existía uniformidad en los *testing* que se aplicaban a los códigos: si bien todos los equipos tenían una etapa de QA en la cual se hacían *tests* manuales y se revisaban los codigos, solo había uno que otro equipo que aplicaba *tests* unitarios; esto hacía muy probable encontrarse con errores no previstos en Producción.
- Al enfrentarse con procesos delicados y largos en la subida a Producción, un error en el código que no se haya encontrado en la fase de QA traía mucha frustración y molestias a todos los equipos de desarrollo.
- Existía a lo más una subida a Producción al mes: al ser un procedimiento tan complicado y que exigía mucho tiempo, fuera del trabajo que conllevaba para toda el área de TI, era imposible pensar en hacer entregas y subidas a Producción en tasas más altas.

1.2 Objetivo General.

Llevar a cabo una Integración Continua (IC), implica la automatización de una serie de tareas que prueban el código y los artefactos generados de distintas formas, con la idea de mantener un código integrado continuamente con la rama principal del repositorio del proyecto, de manera rápida y sin mayores complicaciones.

Con la implementación de un proceso de Integración Continua en el área de TI de esta organización, **se pretende mejorar el proceso de desarrollo actual, tanto en acortar los tiempos comprendidos en las subidas a producción como disminuir la cantidad de errores que se producen**.

1.3 Objetivos específicos.

La solución propuesta en este trabajo se debe sustentar en los siguientes objetivos específicos que encaminarán los esfuerzos:

- Ser capaz de capturar y resolver errores en el desarrollo de forma rápida.
- Reducir los problemas de integración con los proveedores de software rápidamente.
- Reducir los tiempos de integración.
- Hacer entregas de forma confiada y segura.
- Gastar menos tiempo arreglando errores y más creando nuevas features.

2 MARCO CONCEPTUAL

En el presente capítulo se definirán los conceptos principales envueltos en la solución que analiza esta memoria. Desde un repaso a cómo se define la metodología que ocupaba el área de esta compañía anteriormente, hasta las definiciones de los conceptos que definen un marco de entrega e IC.

2.1 Metodologías de Desarrollo de Software

2.1.1 ¿Qué es el Modelo Cascada?

El MC, es el enfoque metodológico que ordena rigurosamente las fases del proceso para el desarrollo de software, de tal forma que el inicio de cada una de estas debe esperar a la finalización de la etapa anterior. Al término de cada etapa, el modelo está diseñado para llevar a cabo una revisión final, que se encarga de determinar si el proyecto está listo para avanzar a la siguiente fase (ver Figura 1). Este modelo fue el primero en originarse y es la base de todos los demás modelos de ciclo de vida. La versión original fue propuesta por Winston W. Royce en 1970 y posteriormente revisada por Barry Boehm en 1980 e Ian Sommerville en 1985 [modCasc].

Si bien ha sido ampliamente criticado desde el ámbito académico y la industria, sigue siendo el paradigma más seguido al día de hoy.

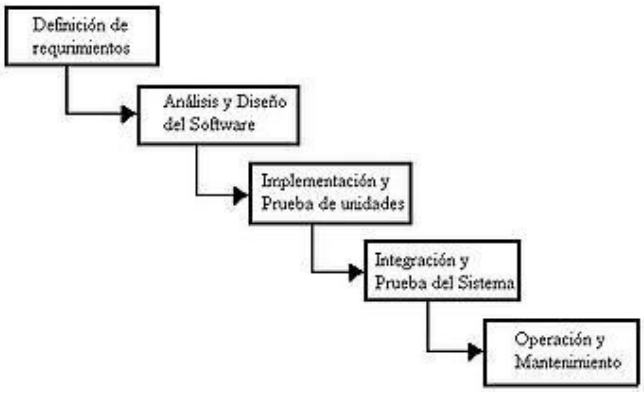


***Figura 1**: Fases del Modelo Cascada*

Fuente: [Topal13]

2.1.2 ¿Qué son las Metodologías Ágiles?

Las Metodologías Ágiles (MA) son una alternativa a la gestión tradicional de proyectos, nacidos en el contexto del desarrollo de software, pero que hoy en día se pueden aplicar a cualquier tipo de proyecto (incluidos los que no se refieren al software). Estas metodologías han ayudado a muchos equipos para hacer frente a los caprichos dentro de un proyecto a través de los ciclos de entrega incrementales e iterativos, convirtiéndose en una alternativa a los métodos tradicionales [metAgil].

El objetivo de las MA es promover un proceso de gestión de proyectos que fomenta la inspección frecuente y la adaptación. Es una filosofía que refuerza un mayor trabajo en equipo, la auto-organización, la comunicación frecuente, orientación al cliente y la entrega de valor. Básicamente, la MA son un conjunto de prácticas eficaces que están diseñada para permitir la entrega rápida de un producto de alta calidad, con un enfoque de negocio que se alinea el desarrollo del proyecto con las necesidades del cliente y los objetivos de la empresa (ver Figura 2).

***Figura 2**: Fases de la Metodología Ágil.*

Fuente: [Samarco]

2.1.3 ¿Qué es la Entrega Continua?

Entrega Continua es la entrega constante de cambios que deja a la aplicación en un estado listo para pasar a Producción o al ambiente que se desee, de manera rápida y sencilla. Esta "entrega" es un proceso automático que se inicia manualmente cuando el desarrollador quiere enviar una nueva versión de la aplicación. A continuación, se ejecuta una serie de pasos que buscan revisar y testear el código con el fin de asegurar una correcta versión del software y quedar listo para su paso al ambiente deseado [entrCont].

2.1.4 ¿Qué es Integración Continua?

La IC es una práctica del desarrollo de software que requiere integrar los desarrollos dentro de un repositorio compartido, a intervalos regulares. Permite de manera automática construir paquetes de código y ejecutar las pruebas necesarias para cumplir con los criterios de calidad establecidos, además de comprobar que el nuevo código funciona perfectamente con el resto de las piezas del sistema [Devops13].

La IC se ha vuelto muy importante en la comunidad de desarrollo de software, probablemente debido al enorme impacto de las metodologías ágiles. En los equipos que han adoptado este tipo de métodos, la IC es uno de los pilares de la agilidad, asegurando que cada proyecto del sistema en su totalidad funcione de forma cohesiva, incluso para equipos grandes y diversos en el uso de sus tecnologías.

Pero ¿por qué IC? ¿Qué beneficios puede traer?. Básicamente, la gran ventaja de la IC es el *feedback* instantáneo. Esto funciona de la siguiente manera: en cada *commit* en el repositorio, se realizan automáticamente una serie de pruebas. Dentro de la fase de pruebas si alguno de los *tests* resulta fallido, el equipo toma conocimiento al instante (por ejemplo, a través de un proceso automatizado que avisa a cierta casilla el *test* fallido y las causas de éste). Posteriormente, corresponde realizar una corrección de el o los problemas lo más rápido posible, lo cual es esencial para no introducir errores al crear nuevas características, refactorización, etc. La IC es más una forma de llevar la seguridad en relación con el cambio: se pueden realizar cambios sin miedo.

Por otra parte, como producto de este proceso se obtiene una versión que teóricamente está lista para entrar en Producción, lo que puede implicar la realización de tareas que no se harían si sólo se estuviera probando manualmente. El proyecto también puede ser

desplegado de forma automática en un servidor de Desarrollo / Certificación, y con esto cada *commit* que se ejecuta sobre el proyecto refleja al instante el cambio.

En conclusión, se puede describir como IC, al proceso automático de construcción de software que corre pruebas de forma automática, para detectar fallos en cada pieza (ver Figura 3).

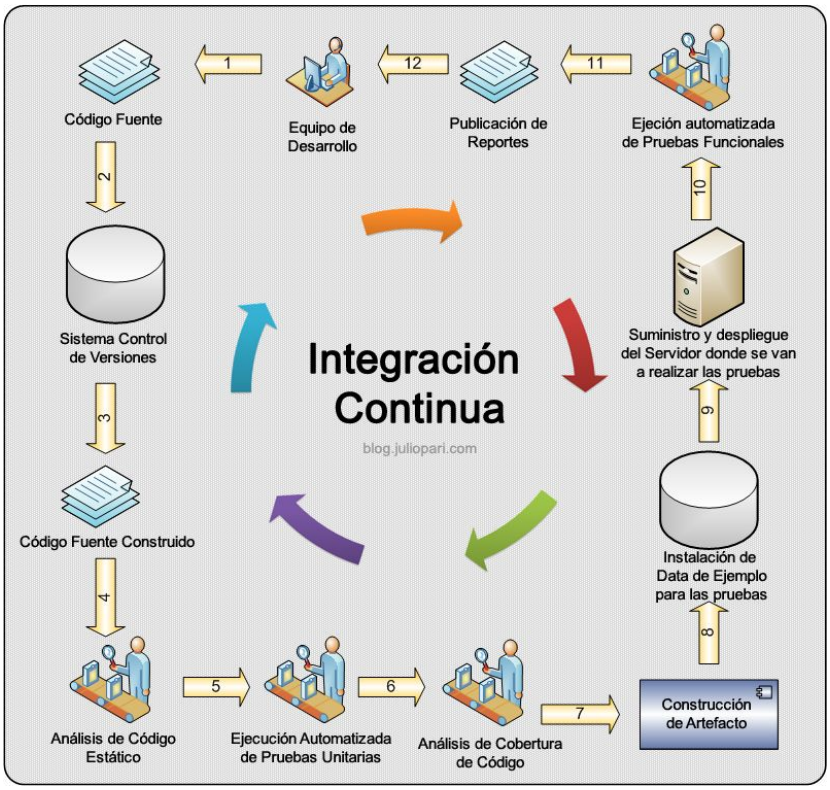


***Figura 3**: Flujo de la Integracion Continua.*

Fuente: [Pari13]

2.2 Software para Integración Continua.

Para generar un flujo de IC es necesario ayudarse de un framework que maneje y establezca las reglas necesarias para que dicho proceso cobre vida. Hoy en día se cuentan con varios *frameworks* que cumplen dichos requerimientos. A continuación se definen tres de los más importantes.

2.2.1 Jenkins.

Jenkins es un software de IC de código libre y escrito en Java. Utilizado por grandes y pequeñas empresas, permite a los desarrolladores organizar *builds*, *test*, incluso instalar sus aplicaciones, así con el tiempo se está herramienta se torna fundamental para el desarrollo. Conforme el número de *jobs* y *builds* aumenta, también lo hace la complejidad de la administración de este software.

El flujo en Jenkins es generalmente planteado como se muestra en la Figura 4.

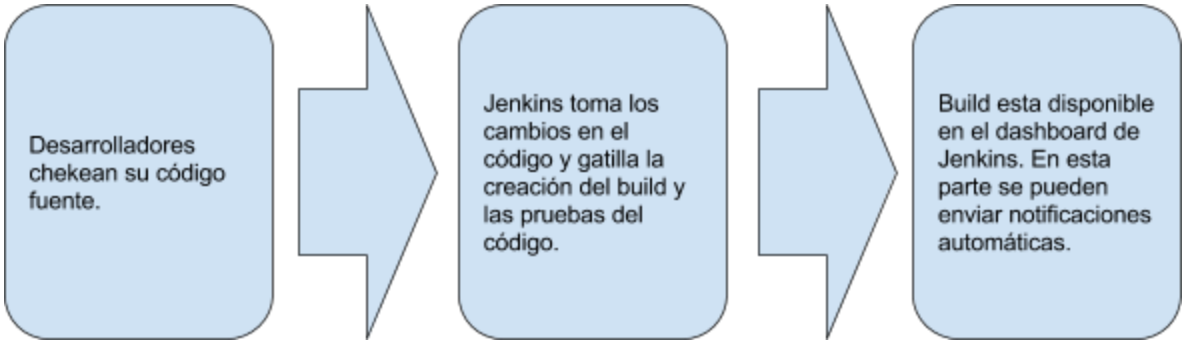


***Figura 4**: Flujo de trabajo en Jenkins.*

Fuente: Elaboración Propia.

2.2.2 Buildbot.

En su núcleo, Buildbot es un sistema de planificación de trabajos: pone en cola los trabajos, ejecuta los trabajos cuando los recursos necesarios están disponibles e informa de los resultados [BuildWiki].

En su instalación Buildbot tiene uno o más maestros y una colección de trabajadores. Los maestros monitorean los repositorios de código fuente para los cambios, coordinan las actividades de los trabajadores y reportan los resultados a los usuarios y desarrolladores, comos se muestra en la Figura 5.

Se puede configurar Buildbot proporcionando una secuencia de comandos en Python al maestro. Este *script* puede ser muy simple, lo que permite la generación dinámica de la configuración, componentes personalizados y cualquier otra cosa que pueda idear.

El *framework* en sí se implementa en Twisted Python, y es compatible con todos los principales sistemas operativos.

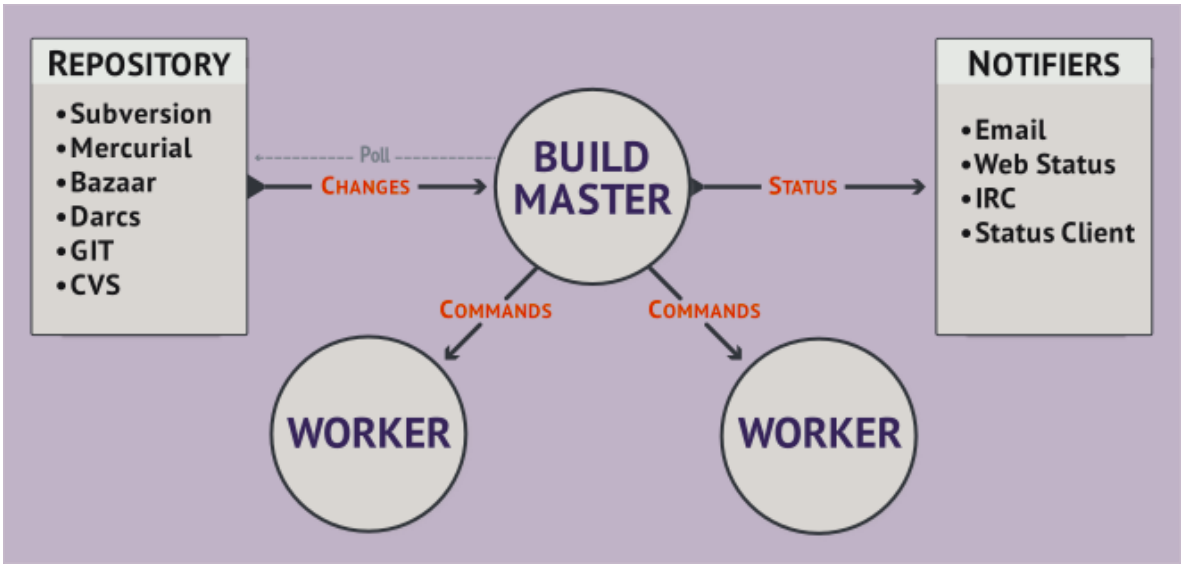


***Figura 5**: Flujo de trabajo en BuildBot.*

Fuente: [BuildBot]

2.2.3 Travis CI.

Travis CI es un servicio de IC alojado, distribuido utilizado para construir y probar proyectos de software alojados en GitHub. Los proyectos de código abierto pueden ser probados sin cargo a través de travis-ci.org. Los proyectos privados pueden ser probados en travis-ci.com sobre una base de honorarios. TravisPro proporciona implementaciones personalizadas de una versión propietaria en el propio hardware del cliente [TravisWiki].

Aunque la fuente es un software técnicamente libre y disponible poco a poco en GitHub bajo licencias permisivas, la compañía señala que es poco probable que los usuarios ocasionales puedan integrarla con éxito en sus propias plataformas.

3 SOLUCIÓN PROPUESTA

Pasar desde una metodología convencional de trabajo, como el desarrollo en MC a una MA, supone una serie de modificaciones en la forma de trabajar en el equipo, además de algunos cambios en la infraestructura, debido a la inclusión de nuevas tecnologías que se puedan involucrar con el fin de generar un entorno seguro.

En este capítulo se presenta la solución final adoptada por la compañía para la implantación de un modelo de integración y entrega continua.

Primero se procederá a analizar qué transformaciones se debieron llevar a cabo en la forma de trabajo de los equipos de la compañia. Tambien se verá qué cambios se debieron tener en cuenta a nivel de infraestructura, y finalmente se presentarán las pautas que adaptaron el proceso antiguo de desarrollo a Jenkins.

Cabe destacar que con el fin de impulsar un cambio más controlado, la compañía decide crear un equipo de trabajo encargado de evangelizar al resto de los equipos de desarrollo, en lo que a IC se refiere. Este equipo es quien decide adoptar Jenkins basado principalmente en su naturaleza *open source*, también por la madurez de la herramienta con una gran comunidad de usuarios en la web y por estar hecho en Java, lo cual se alinea con el lenguaje escogido a nivel gerencial para renovar el área de TI de la compañía.

Así en un principio se presenta una compañía con un proceso de desarrollo maduro, pero poco eficiente. Con el fin de llevar un proceso de experimentación de los cambios que implican la adopción una metodología ágil, se escogen tres equipos con proyectos de distinta índole, que varían en el grado de cohesión al núcleo del negocio de la compañía, así como también en la magnitud del proyecto. Estos equipos llevan a cabo un proceso de adopción de cuyo éxito depende que se difunda este proceso al resto del área de TI.

3.1 Adoptar prácticas que intervienen en proceso de Integración Continua.

Para que los beneficios de la IC se cosechen es necesario adoptar el núcleo de sus prácticas. Tales prácticas pueden afectar la forma habitual de trabajo, exigiendo un esfuerzo de todo el equipo. Como resultado se obtiene un entorno más controlado y seguro para el desarrollo. Estas prácticas transforman la IC en un proceso natural de

implementación de código y reduce el riesgo de errores y problemas, así como también facilita el proceso de corrección.

El equipo evangelizador tuvo la tarea de crear un listado de prácticas que asegurarán la adopción una MA de manera clara para los equipos que estaban en fase de experimentación. Esta metodología también implicó proveer herramientas necesarias para llevar a cabo este proceso. Como resultado se enfocaron los esfuerzos en 2 objetivos principales: Practicar las metodologías necesarias que apoyen la IC, adoptar las herramientas necesarias para apoyar este proceso y generar la infraestructura para soportar este cambio. Estos objetivos generaron un listado de 4 tareas fundamentales:

- Crear un repositorio de código.
- Crear un sistema de construcción automatizado
- Hacer *commits* diarios
- Hacer TDD

A continuación analizaran cada una de las tarea mencionadas anteriormente.

3.1.1 Crear un repositorio de código

Para la compañía en cuestión este paso ya estaba cumplido antes de empezar el proceso de adopción. En concreto se contaba con un repositorio GIT que los equipos de software ocupaban diariamente para respaldar sus proyectos, así este *ítem* no implicó nada nuevo para los equipos y su adopción fue inmediata.

3.1.2 Crear un sistema de construcción automatizado

Sistemas de gestión de proyectos para Java se encontraron varios, pero nuevamente la opción elegida se basó en una herramienta que tuviese algún tiempo en el mercado y con una comunidad activa. Esta herramienta fue Maven [maven]. Los equipos se tuvieron que familiarizar a una nueva herramienta que facilitó el desarrollo.

3.1.3 Hacer *commits* diarios

El principal objetivo de la IC es la incorporación de los cambios de la forma más frecuente como sea posible, para identificar y resolver problemas rápidamente. Mantenerse sin *commits* de código por un largo tiempo disminuye los beneficios de la IC e incluso crea algunos inconvenientes como los conflictos durante la actualización con el repositorio de código actual. En este caso, las tareas simples terminan convirtiéndose en difíciles, ya que el impacto de los cambios es mayor. En algunos casos, el desarrollador puede olvidar o dejar deliberadamente de actualizar el código con el repositorio y esto, en última instancia, retrasa la aparición de conflictos. Cuando esto ocurre, la situación puede ser aún más complicada, lo que dificulta el proceso de IC. Idealmente los cambios deben actualizarse en el repositorio local al menos cada día, lo que permite a todo el equipo conocer el estado del proyecto y, si es necesario, la reutilización del código implementado. Aumentar la frecuencia de los *commits*, incluso haciéndolos a menudo, no significa que los conflictos sigan ocurriendo, pero como las partes del código son más pequeñas, la resolución de ellos se hace más simple.

3.1.4 Hacer TDD.

TDD es una técnica de desarrollo orientada a las pruebas, que se fortaleció con los marcos ágiles. Se basa en tres aspectos básicos llamadas "las tres leyes de TDD" [3leyes]:

- No se permite escribir algún código de Producción, a menos que sea para hacer pasar un *test* unitario fallido.
- No se permite escribir más que un *test* unitario que sea lo suficiente para que éste falle; los errores de compilación se consideran fallos.
- No se permite escribir algún código de Producción más que el que sea suficiente para hacer pasar un *test* unitario.

Además de estos tres leyes existe una etapa para la refactorización del código, en donde es importante tener cuidado con que las pruebas unitarias sigan funcionando. La principal ventaja de TDD es que asegura que para cada función se desarrolló una serie de pruebas para garantizar su funcionamiento. A través de estas pruebas, el servidor de IC comprueba automáticamente si los nuevos elementos insertados no comprometen el funcionamiento

esperado de otros módulos de la aplicación. Hay una serie de *frameworks* disponibles para hacer *test* unitario, entre los que se pueden destacar:

- JUnit [junit], NUnit [nunit] y TestNG [testng]: para el desarrollo de las pruebas unitarias.
- PowerMock [pmock], EasyMock [emock] y Mockito [mockito]: las que se utilizan junto con los *tests* unitarios, Estas herramientas se utilizan para simular las referencias a las dependencias externas que se realizan mediante la creación mocks de objetos.

En este punto la idea no fue dar una pauta de con qué herramientas decir al equipo que haga TDD, sino que ellos tuviesen la libertad de elegirlas en base a experiencia o simplemente por decisión de ellos. El objetivo era llevar un buen proceso de TDD respetando la dinámica que implica.

La curva de aprendizaje fue un poco más lenta, debido a que ninguno de los equipos estaba familiarizado con las pruebas unitarias.

Fuera de estas prácticas que tienen que llevar a cabo cada equipo, se necesitan de otros objetivos que diseñan el trabajo de la IC con Jenkins y que se mencionan en las siguientes secciones.

3.2 Generación de *pipeline* de trabajo

Es muy importante mantener un código confiable al final de cada integración, esto puede suponer entonces incluir pruebas de distintos tipos, aparte de las unitarias, y en distintos entornos en los que existe el proyecto. Por ejemplo, en el entorno de QA se podría asegurar con pruebas funcionales, con las cuales comprobar el comportamiento de las características de una manera integrada. Un buen ejemplo de un marco de este tipo es Selenium [selenium], que pone a prueba la funcionalidad a través de una interfaz de usuario, simulando el uso real de la aplicación. Esto permite que la prueba evalúe el resultado de una operación mediante la comprobación de la integración de todos componentes que son parte de su implementación. Obviamente no se puede probar todos los aspectos de un software, pero con una buena gama en las pruebas pueden minimizar los problemas, facilitar correcciones y minimizar los costos de las pruebas de calidad.

El objetivo en esta etapa es, entonces, analizar cómo es el proceso de desarrollo y definir qué criterios son los mínimos necesarios para que un código pase de un entorno a otro. Además precisar qué se hace en cada entorno. Esto es la base para definir qué pruebas se ejecutarán en cada ambiente, a fin de asegurar un producto final confiable. Se podrían establecer una variedad de pruebas para tal fin, como pruebas de sistema, de carga, funcionales, de estrés, de aceptación, entre otras.

Durante el trabajo los equipos generaron distintos *sets* de pruebas, básicamente se estructuraron en tres ambientes (Figura 6), cada uno con un *set* de pruebas que se listan a continuación:

- Servidor de Desarrollo
    - *Test* unitarios
    - *Test* de Integración
    - *Test* Funcionales
    - Análisis con SonarQube
    - *Deploy* en Desarrollo
- Servidor de Certificación
    - Deploy en Certificación
- Servidor de Producción
    - Generacion de Codigo en Jira
    - Deploy en Producción

***Figura 6**: Ambientes de Trabajo definidos.*

Fuente: Elaboración Propia.

Este *set* de pruebas define la estructura del flujo de ejecución de trabajos, también conocido como **el *pipeline* en Jenkins,** que se debe ejecutar para hacer desarrollos que garanticen el correcto funcionamiento en todos sus ambientes.

Cabe señalar que el ambiente de Desarrollo tiene como objetivo permitir levantar un entorno de pruebas, con el cual se tenga un *input* de las funcionalidades desarrolladas. El

ambiente de Certificación es un espejo del ambiente de Producción pero con menos recursos, la idea acá es tener un entorno de pruebas lo más parecido a Producción, con menos recursos y con las versiones más recientes de los desarrollos. Además, mencionar la función del servidor SonarQube [sqube] y Jira [jira]; el primero es un servidor que se dedica al análisis del código que se sube en el repositorio, básicamente vela por el cumplimiento de buenas normas de programación. Y el segundo es un *framework* de administración del flujo de trabajo en Kanban de los equipos, así solo aquellos desarrollos que tengan un correcto estado en Jira podrán pasar a Producción.

3.3 Diseño de infraestructura necesaria para el proceso de Integración Continua.

Ya con un una idea clara de cuáles serán las pruebas a ejecutar para asegurar el éxito en el *deploy* de cada etapa, hay que traducir este diseño en una estructura que soporte, los ambientes y herramientas definidas en el proceso anterior.

Entonces el flujo de actividades que lleva a cabo mediante Jenkins (ver Figura 7) comienza con éste vigilando el repositorio del control de versiones (GIT). Ante un cambio descargará el código, lo compilará y realizará las comprobaciones pertinentes que se hayan establecido, ahí entra en acción una interacción con Jira para verificar el estado en que se encuentra ese desarrollo. Luego se desplegará el código a distintos entornos. En este caso ejecutará el *set* de *test* definidos en el ambiente de Desarrollo para después hacer un despliegue en este ambiente y así lo mismo en Certificación y Producción. Durante el proceso Jenkins puede ser configurado para dar un *feedback* a los desarrolladores, con el fin de saber si el código que acaban de subir está bien, o hay algún fallo que hay que solucionar.

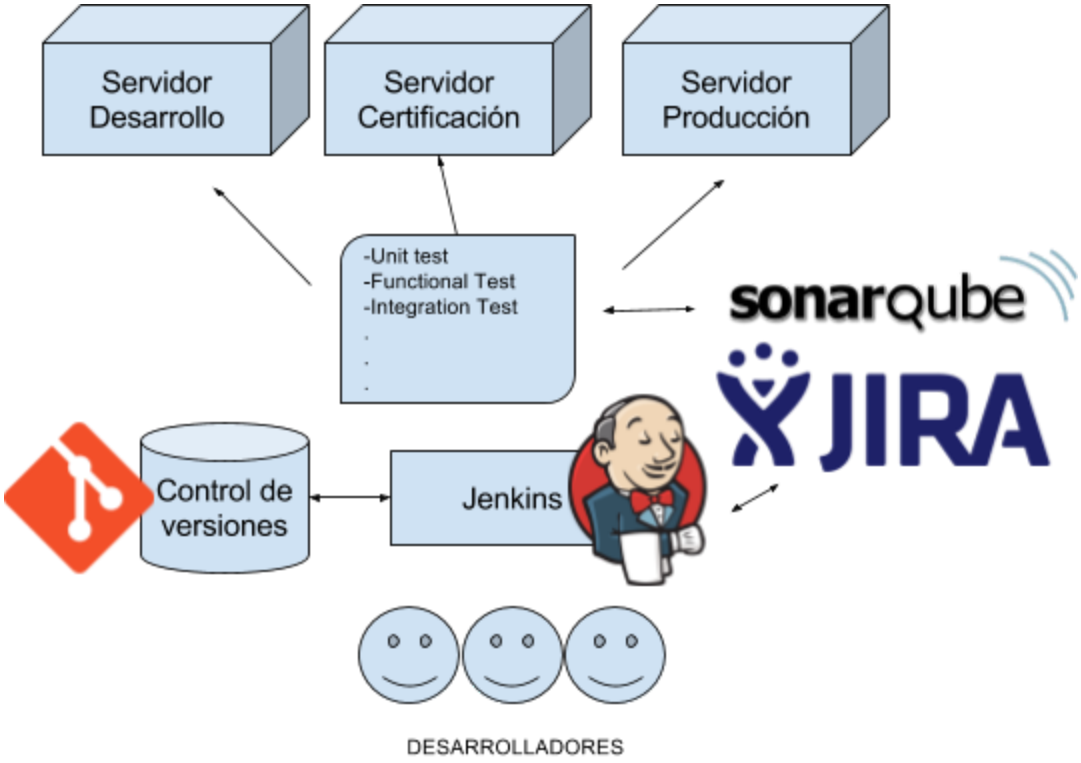


Figura 7:Elementos que actúan para soportar el despliegue de cada ambiente en el flujo de actividades de Jenkins.

Fuente: Elaboración Propia

.

3.4 Jenkins y la creación de Jobs automáticos.

En esta sección se verá un detalle un poco más técnico de cómo se configuraron los jobs en Jenkins y algunos detalles del trabajo con esta herramienta que ayudaron a su uso.

3.4.1 Creación de un Job.

La unidad fundamental del trabajo en Jenkins es crear un *Job,* el cual se gatillará automáticamente o después de la ejecución exitosa de otro. Para los proyectos con los cuales se experimentaron, cada *Job* estaba acompañado de un *script bash* alojado en el mismo proyecto, el cual era ejecutado desde éste. Parte de la estructura de un Job se puede observar en las Figuras 7 y 8.

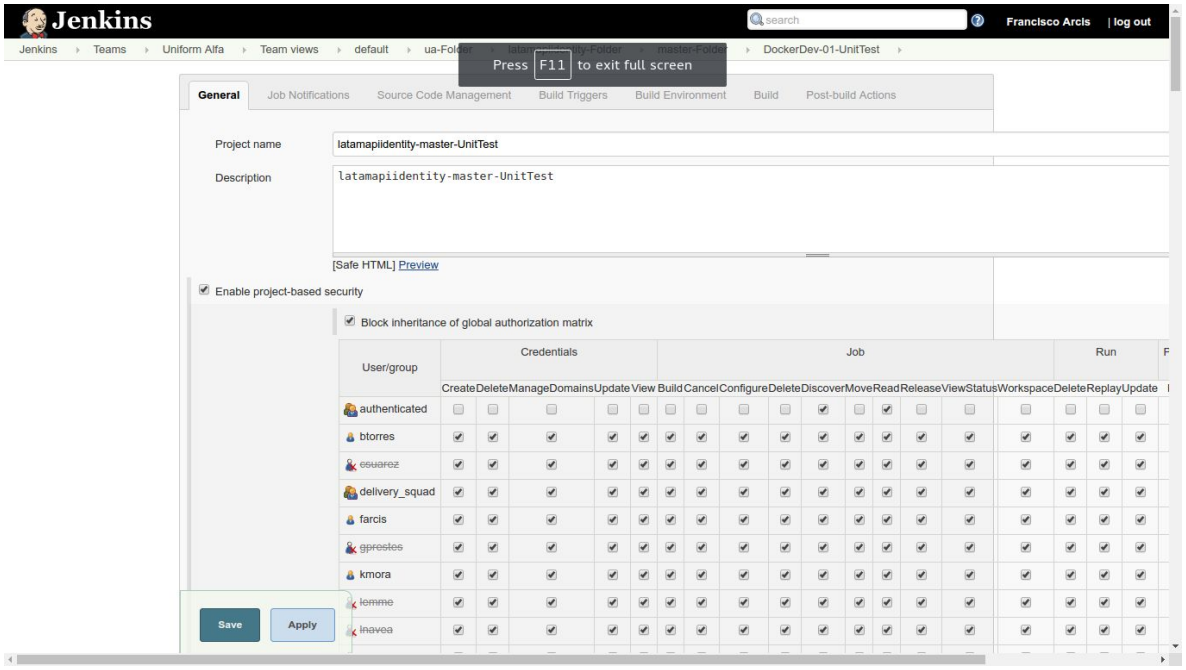


***Figura 7**:Estructura de un Job de Jenkins (Parte I).*

Fuente: Elaboración Propia

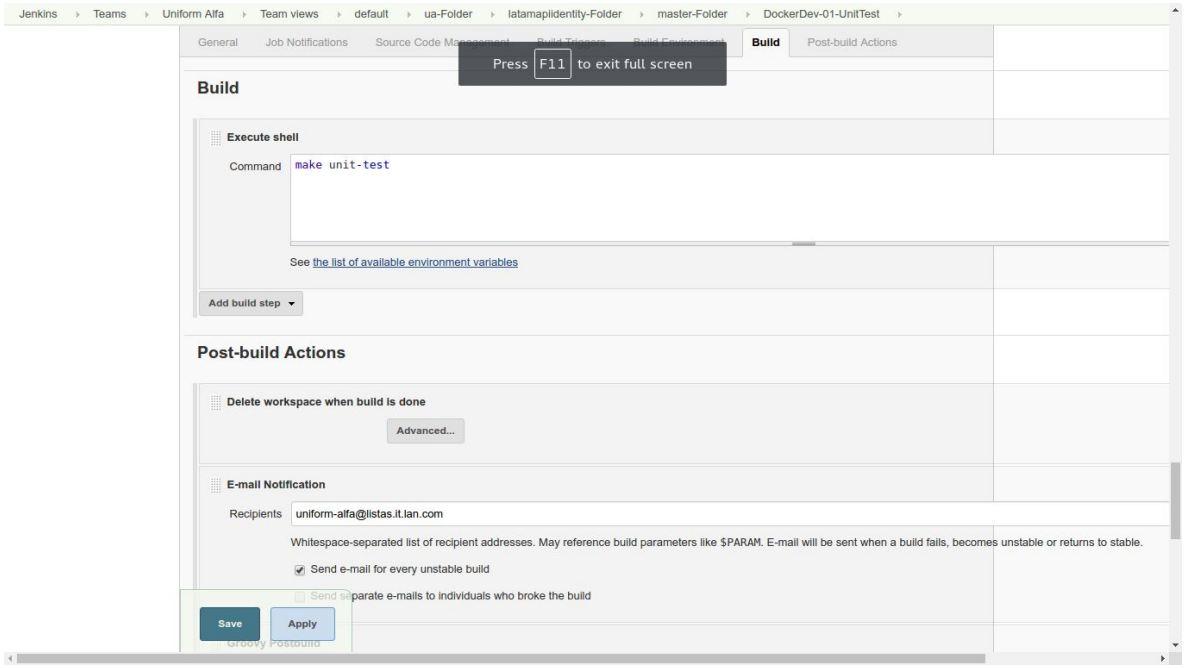
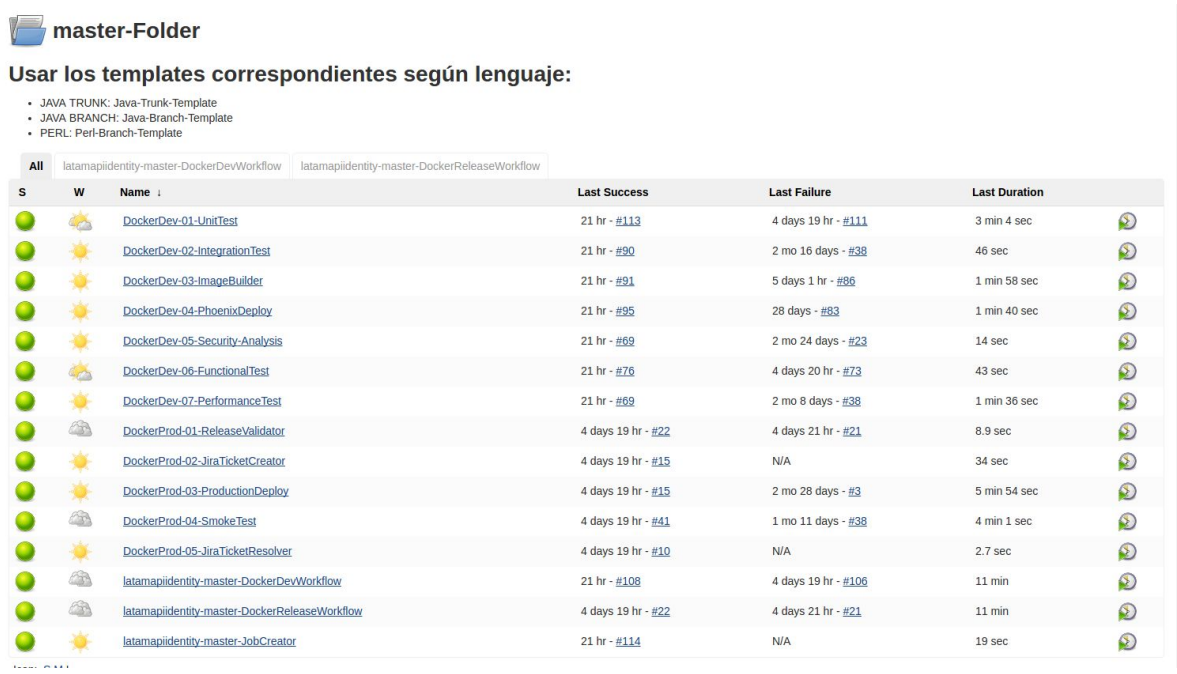


***Figura 8**:Estructura de un Job de Jenkins (Parte II).*

Fuente: Elaboración Propia

3.4.2 El *pipeline* en Jenkins.

Una vez que se conoce el proceso de desarrollo y se adapta una infraestructura para cumplir con tal proceso, se desarrolla el *pipeline* de trabajo, configurando cada Job requerido con su bash correspondiente. El producto de esto lo podemos observar en la Figura 8:



S	W	Name ↓	Last Success	Last Failure	Last Duration
		DockerDev-01-UnitTest	21 hr - #113	4 days 19 hr - #111	3 min 4 sec
		DockerDev-02-IntegrationTest	21 hr - #90	2 mo 16 days - #38	46 sec
		DockerDev-03-ImageBuilder	21 hr - #91	5 days 1 hr - #86	1 min 58 sec
		DockerDev-04-PhoenixDeploy	21 hr - #95	28 days - #83	1 min 40 sec
		DockerDev-05-Security-Analysis	21 hr - #69	2 mo 24 days - #23	14 sec
		DockerDev-06-FunctionalTest	21 hr - #76	4 days 20 hr - #73	43 sec
		DockerDev-07-PerformanceTest	21 hr - #69	2 mo 8 days - #38	1 min 36 sec
		DockerProd-01-ReleaseValidator	4 days 19 hr - #22	4 days 21 hr - #21	8.9 sec
		DockerProd-02-JiraTicketCreator	4 days 19 hr - #15	N/A	34 sec
		DockerProd-03-ProductionDeploy	4 days 19 hr - #15	2 mo 28 days - #3	5 min 54 sec
		DockerProd-04-SmokeTest	4 days 19 hr - #41	1 mo 11 days - #38	4 min 1 sec
		DockerProd-05-JiraTicketResolver	4 days 19 hr - #10	N/A	2.7 sec
		latamapiidentity-master-DockerDevWorkflow	21 hr - #108	4 days 19 hr - #106	11 min
		latamapiidentity-master-DockerReleaseWorkflow	4 days 19 hr - #22	4 days 21 hr - #21	11 min
		latamapiidentity-master-JobCreator	21 hr - #114	N/A	19 sec

***Figura 9**: Ejemplo de un Pipeline modelo de Jenkins.*

Fuente: Jenkins

Cada *commit* genera el despliegue consecutivo de los *Jobs*. Si uno de ellos no logra acabar exitosamente se marcará en rojo en el panel general que se muestra en la Figura 10. A partir de esto este *framework* puede avisar a través del correo, que existe un *job* caído en el *pipeline*,para que el equipo se active y solucione el error a la brevedad.

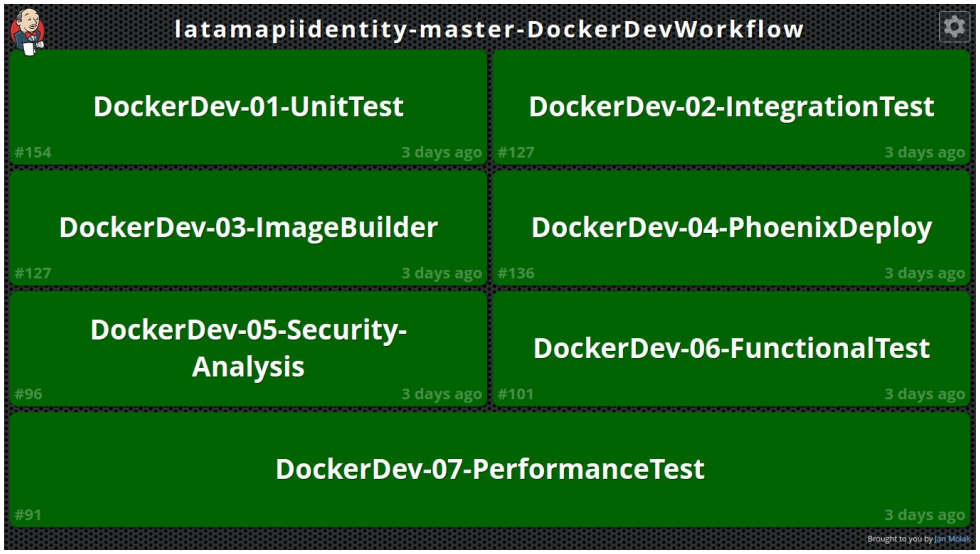


***Figura 10**: Panel General de Jenkins.*

Fuente: Jenkins

4 VALIDACIÓN DE LA SOLUCIÓN PROPUESTA

En este capítulo se discutirán los resultados obtenidos de la solución propuesta con este trabajo, analizando la situación previa y contrastándola con la situación generada con la aplicación de la solución, así como también destacando puntos a favor y dificultades asociadas a esta propuesta.

Cabe recordar que, como se mencionó anteriormente, los tres equipos que experimentaron con la adopción de esta metodología (de casi una veintena de equipos que hay en la organización en estudio), difieren en el grado de cohesión al núcleo del negocio, así como también en la magnitud del proyecto a cargo. La descripción de cada uno de ellos se puede ver en la tabla 1. La experimentación se lleva a cabo con equipos de perfiles distintos a fin de obtener un *input*. En el caso de éxito, de qué equipos deberían continuar con la adopción de esta metodología.

Equipo A	Proyecto pequeño y relativamente nuevo, corresponde a una característica nueva que no es clave para el funcionamiento del negocio.
Equipo B	Proyecto mediano perteneciente al núcleo del negocio.
Equipo C	Proyecto, grande y cuyo ámbito está definido sobre el núcleo de negocio de la compañía.

***Tabla 1**: Definición de Equipos Participantes de la prueba de Jenkins en la organización en estudio.*

Los resultados que se muestran a continuación corresponden a un proceso de 6 meses de adopción

I. Lead Time:

Probablemente el *Lead Time* es el factor que sufre un mayor cambio a favor con la en la siguiente propuesta. Con el MC, demoraban entre uno a dos meses en poder dejar en Producción sus cambios, esto se ve contrastado con equipos en IC a que bajaron esa cifra a 3 semanas y así siguieron bajando para terminar en procesos de 3 a 2 días en algunos casos, en la medida que fueron madurando sus procesos. La Tabla 2 muestra el cambio observado en los equipos.

Equipo	Situación con MC	Situación con MA / IC
Equipo A	1 mes	de 3 días a una semana
Equipo B	1 mes	al menos una semana
Equipo C	1 a 2 meses	menor a dos semanas

***Tabla 2**: Contraste de Lead Time de Equipos.*

II. Coste Inicial:

Cabe destacar que generar un sistema de IC puede necesitar un tiempo inicial para generar un *set* de pruebas que cumpla con lo necesario para obtener un producto confiable.

Esta medida se relaciona en cómo vayan los equipos madurando en sus procesos durante este periodo de experimentación, bajando sus tiempos y volviéndose más efectivos. Debido a su naturaleza más compleja, el Equipo C, fue el que más demoró en madurar y generar un *pipeline* genérico que asegurara sus ambientes, tardando 3 meses en tener una adopción aceptable, con tiempos menores a los que estaban acostumbrados en durante su trabajo con MC, como muestra la Tabla 3. En contraste, los dos equipos restantes a partir del mes y medio ya se les consideraba adoptados al cambio de paradigma.

Equipo	Tiempo de adopción
Equipo A	1 mes y medio
Equipo B	1 mes y medio
Equipo C	3 meses

***Tabla 3**: Tiempos de adopción de un pipeline maduro de los Equipos.*

III. Coste de hardware para configuración de metodología:

IC implica un conjunto de máquinas que albergarán las diversas herramientas para llevar a cabo el proceso: repositorio, automatización, entornos, etc., que si no estaban desde antes implica un costo adicional para generarlos. En este caso específico se tuvo que invertir para la instauración del servidor mismo de Jenkins, además se creó un ambiente nuevo de certificación para hacer pruebas de estrés.

Este factor representa una desventaja en el sentido monetario pues hubo que generar una inversión para costear la infraestructura y herramientas (licencias) que

soportaron un proceso de IC. A medida que la compañía fue incluyendo a otros equipos, hubo mayor gasto de inversión.

IV. Tiempos en subir a Producción:

Acá la diferencia es notoria. En el pasado un paso a Producción suponía un dolor de cabeza, implicaba mucho esfuerzo y tiempo, además de uno que otro *rollback* después de notar que en Producción el reciente cambio había generado errores en otras secciones del código que no se previeron, lo cual sumaba más tiempo al proceso de subida a Producción.

Lo que antes suponía un proceso muy delicado que podía demorar incluso una semana en el caso de encontrar errores en Producción, pasó a demorarse de un día a solo un proceso de horas (ver Tabla 4).

Equipo	Situación con MC	Situación con MA / IC
Equipo A	al menos 1 semana	1 día
Equipo B	al menos 1 semana	1 día
Equipo C	3 dias	2 horas

***Tabla 4**: Contraste en demora de subidas a Producción de Equipos.*

V. Tiempo de reacción frente a dependencias:

La mejora acá se produce por la práctica de TDD junto con las pruebas unitarias y de integración. Estas pruebas generaron en los desarrolladores confianza en el código que se está subiendo y cada vez se hizo menos frecuente encontrarse con errores por alcance. Una vez que cada equipo modeló los *pipelines* que aseguraban cada uno de los ambientes, rara vez se encontraron con errores inesperados en Producción.

5 CONCLUSIONES

Si bien la IC agrega mucho valor al proyecto, al darle mayor estabilidad, disminuir la tasa de fallas en Producción, disminuir los tiempos de integración y permitir hacer entregas más frecuentes, estos cambios no se concretarán si el equipo no se suma a la cultura ágil en la cual se cimenta esta metodología. En otras palabras, sino existe un compromiso de los desarrolladores en realizar un buen *set* de *tests* unitarios en sus desarrollos, perderíamos la confianza en que estos *tests* comprueben que los nuevos cambios no rompen nada de lo existente y por lo tanto esta metodología se volvería sólo una pérdida de tiempo, un obstáculo para desarrollar y subir a Producción.

En cuanto a lo costoso que puede ser en un principio, la ventaja directa acá es que una vez que un equipo la adopta y pasa por esta fase en su comienzo, el siguiente equipo se enfrentará a una carga más liviana y prácticamente tiene que aplicar lo hecho por el equipo anterior. Acá es donde el equipo de la compañía encargado de dar soporte a la IC, tuvo una tarea importante generando la documentación necesaria para evangelizar a los nuevos equipos que adoptaron esta metodología.

Dado que el proceso es similar entre los equipos, se ha propuesto generar un *pipeline* de trabajo genérico en Jenkins que abarque la automatización de los principales tipos de *test* que tienen las aplicaciones de la compañía; esto contribuye en la adopción más rápida y clara de esta metodología. Paralelamente se está trabajando en generar *jobs* que automaticen todos los tipos de tareas que se enfrentan los desarrolladores en esta compañía

Junto lo anterior un proceso de mejora continua fue invadiendo a los equipos, mejorando continuamente sus *pipelines*, con el uso de herramientas que ayudaron a tal objetivo.

6 REFERENCIAS BIBLIOGRÁFICAS

[modCasc] Isa Arteta , “Modelo Cascada”, http://modelo-cascada.blogspot.cl/, revisado 20-03-2017

[Topal13] Yazzdleidy Topal, “MODELO LINEAL SECUENCIAL (CASCADA)",”http://primermodelo.blogspot.com/", revisado el 29-09-2016.

[metAgil] Markos Goikolea, “¿Qué es Agile Project Management?” http://comunidad.iebschool.com/iebs/agile-scrum/que-es-agile-project-management-ventajas-de-ser-el-mas-rapido-y-agil/, revisado 20-03-2017

[Samarco] Samarco web engineering, “Desarrollo web",http://www.samarcoweb.com/es/servicio/desarrollo-web", revisado el 29-09-2016.

[entrCont] Javier Garzas, “¿Tardaríais mucho en pasar a producción un cambio en sólo una línea de código? Aprende entrega continua”, http://www.javiergarzas.com/2012/11/entrega-continua-continuous-delivery.html, revisado 23-03-2017

[Devops13] DevOps, “Integracion Continua",”http://www.devops-es.com/2013/09/23/integracion-continua/", revisado el 29-09-2016.

[Pari13] Julio Pari, “Integración Continua en Proyectos Ágiles de Software",”http://blog.juliopari.com/integracion-continua-en-proyectos-agiles-de-software/", revisado el 11-11-2016.

[BuildWiki] Wikipedia, “BuilBot",”https://en.wikipedia.org/wiki/Buildbot", revisado el 01-02-2017.

[BuildBot] BuildBot, “BuildBot Official Web",”http://buildbot.net/", revisado el 01-02-2017.

[TravisWiki] Wikipedia, “Travis",”https://en.wikipedia.org/wiki/Travis_CI", revisado el 01-02-2017.

[maven] Maven, “https://maven.apache.org/”, revisado 27-03-2017

[3leyes] Willy Mejia, “Las Tres Leyes de TDD”, “https://willyxoft.wordpress.com/2009/09/11/3-leyes-tdd/”, revisado

[junit] JUnit, “http://junit.org/junit4/” , revisado 25-03-2017

[nunit] Nunit ,“https://www.nunit.org/”, revisado 23-03-2017

[testng] TestNG “http://testng.org/doc/”, revisado 23-03-2017

[pmock] Pmock, “http://pmock.sourceforge.net/”, revisado 23-03-2017

[emock] Emock, “https://www.npmjs.com/package/emock”, revisado 23-03-2017

[mockito] Mockito, “http://site.mockito.org/”, revisado 23-03-2017

[selenium] Selenium, “http://www.seleniumhq.org/”, revisado

[sqube] SonarQube, “https://www.sonarqube.org/”, revisado

[jira] Jira, “https://es.atlassian.com/software/jira”, revisado