Universidad de Cuenca

Facultad de Ingeniería

Maestría en Gestión Estratégica de Tecnologías de la Información

Proyecto de Tesis

ARQUITECTURA TECNOLÓGICA PARA LA ENTREGA CONTINUA DE SOFTWARE CON DESPLIEGUE EN CONTENEDORES

Autor
Ing. Luis Alberto Iñiguez Sánchez
C.I. 0102156429

Director
Ing. Víctor Hugo Saquicela Galarza, PhD.
C.I. 0103599577

Grado académico: Magíster

Cuenca, junio 2017

RESUMEN

La Universidad de Cuenca a través de la Dirección de Tecnologías de la Información y Comunicación ofrece a la Comunidad Universitaria servicios relacionados con las TICs para dar soporte a las actividades administrativas, académicas y de investigación. La Dirección dispone de una unidad de desarrollo de software que se encuentra encargada de dar mantenimiento a los sistemas de información que se encuentran en producción y el desarrollo de nuevos proyectos enfocados a fortalecer la estrategia institucional. La carencia de una arquitectura estándar para el desarrollo de software ha provocado que el despliegue de nuevas versiones de las aplicaciones sea una tarea extensa y complicada, introduciendo con facilidad errores de regresión en el software que se pone en producción dificultando el aseguramiento de calidad. Por tanto, se ha planteado la necesidad de implementar un *pipeline* de entrega continua de software que permita potenciar al equipo de desarrollo para mitigar los riesgos inherentes a la liberación de producto. Para el éxito en la implementación de un *pipeline* de entrega continua se requiere un alto nivel de automatización en el proceso de construcción y verificación del software que permita despliegues automatizados de manera confiable. Las herramientas que lo soportan se encuentran en función de los *stacks* de tecnologías y el ambiente de ejecución de las aplicaciones. Este proyecto de tesis define a partir de los fundamentos de la entrega continua de software y el análisis del trabajo realizado hasta el momento, una arquitectura tecnológica para la implementación de un proceso automatizado de entrega continua acorde a los lineamientos y necesidades de la unidad de desarrollo de software de la Universidad de Cuenca.

Palabras clave: Entrega Continua de Software, Microservicios, Contenedores, Test Driven Development.

ABSTRACT

The University of Cuenca through the Information and Communications Technology Division offers to the University Community services related to ICTs to support administrative, academic and research activities. The division has a software development unit that is in charge of maintaining the information systems, that are in production and the development of new projects focused on strengthening the institutional strategy. The lack of a standard architecture for software development has made the deployment of new versions of applications an extensive and complicated task, easily introducing regression errors in the software that is used in production, that is hindering quality assurance. Therefore, the need to implement a pipeline of continuous software delivery that allows the development team to mitigate the risks that are inherent in the release of the product has been raised. For the success in the implementation of a pipeline, an elevated level of automation is required in the process of construction and verification of the software that allows automated deployments in a reliable manner. The tools that support it are based on technology stacks and the execution environment of the applications. This thesis project is defined from the foundations of the continuous delivery of software and the analysis of the work done so far, a technological architecture for the implementation of an automated process of continuous delivery according to the guidelines and needs of the development unit of software of the University of Cuenca.

Keywords: Continuous Delivery, Microservices, Containers, Test Driven Development.

CONTENIDO

RESUMEN 2

ABSTRACT 3

CLÁUSULA DE LICENCIA Y AUTORIZACIÓN PARA LA PUBLICACIÓN EN EL REPOSITORIO INSTITUCIONAL 9

CLÁUSULA DE PROPIEDAD INTELECTUAL 10

DEDICATORIA 11

AGRADECIMIENTOS 12

1 INTRODUCCIÓN 13

1.1 ESTRUCTURA DEL TRABAJO DE TESIS 15

1.1 OBJETIVOS 15

1.1.1. OBJETIVO GENERAL 15

1.1.2. OBJETIVOS ESPECÍFICOS 15

1.2 ALCANCE 15

2 MARCO TEÓRICO 17

2.1 ANTECEDENTES 17

2.2 PLATAFORMA TECNOLÓGICA PARA IMPLEMENTACIÓN DE APLICACIONES DE LA UNIVERSIDAD DE CUENCA 19

2.2.1 COMPONENTES DE LA PLATAFORMA DE DESARROLLO DE SOFTWARE 19

2.2.2 COMPONENTES DE LA INFRAESTRUCTURA DE DESPLIEGUE 20

2.3 ENTREGA CONTINUA DE SOFTWARE 21

2.3.1 PRINCIPIOS PARA LA IMPLEMENTACIÓN DE ENTREGA CONTINUA 22

2.3.2 PIPELINE DE ENTREGA CONTINUA DE SOFTWARE 23

2.3.2.1 CONTROL DE VERSIONES 25

2.3.2.2 GESTIÓN DE LA CONFIGURACIÓN 26

2.3.2.3 AUTOMATIZACIÓN DE LA CONSTRUCCIÓN 28

2.3.2.4 AUTOMATIZACIÓN DE DESPLIEGUE 29

2.3.2.5 AUTOMATIZACIÓN DE LAS PRUEBAS 29

2.3.2.6 INTEGRACIÓN CONTINUA 34

2.3.3 PIPELINE DE DESPLIEGUE 35
2.3.3.1 ETAPA DE COMMIT 36
2.3.3.2 ETAPA DE PRUEBAS DE ACEPTACIÓN AUTOMATIZADAS 36
2.3.3.3 ETAPA DE PRUEBAS MANUALES 37
2.3.3.4 ETAPA DE LIBERACIÓN 37
2.4 MICROSERVICIOS 38
2.4.1. CARACTERÍSTICAS 38
2.4.2. DESAFÍOS EN LA CONSTRUCCIÓN DE UNA ARQUITECTURA BASADA EN MICROSERVICIOS 40
2.5 VIRTUALIZACIÓN COMO PLATAFORMA 42
2.5.1. MÁQUINAS VIRTUALES 42
2.5.2. CONTENEDORES 43
2.6 DOCKER: INFRAESTRUCTURA BASADA EN CONTENEDORES 45
2.6.1. IMAGEN DOCKER 45
2.6.2. CONTENEDOR 46
2.6.3. ARQUITECTURA 46
3 TRABAJOS RELACIONADOS 48
3.1 PROCESO DE ENTREGA CONTINUA DE SOFTWARE 48
3.2 PIPELINE DE ENTREGA CONTINUA DE SOFTWARE 53
3.3 ESTRATEGIA DE AUTOMATIZACIÓN DE PRUEBAS 57
3.4 ENTREGA CONTINUA Y MICROSERVICIOS 59
3.5 ENTREGA CONTINUA Y CONTENEDORES 60
4 ARQUITECTURA TECNOLÓGICA PLANTEADA 63
4.1 SELECCIÓN DE LOS COMPONENTES DE SOFTWARE 63
4.1.1 REQUERIMIENTOS NO FUNCIONALES 63
4.1.2 ETAPAS DEL PIPELINE DE ENTREGA CONTINUA DE SOFTWARE 64
4.1.2.1 INTEGRACIÓN CONTINUA 65
4.1.2.2 CONTROL DE VERSIONES 66
4.1.2.3 ANÁLISIS Y GESTIÓN DE CÓDIGO 67

4.1.2.4 CONSTRUCCIÓN 67

4.1.2.5 PRUEBAS 68

4.1.2.6 CONFIGURACIÓN Y APROVISIONAMIENTO 69

4.1.2.7 DESPLIEGUE 69

4.2 ARQUITECTURA TECNOLÓGICA RESULTANTE 69

4.2.1. ASEGURAMIENTO DEL PIPELINE 72

4.1 PROTOTIPO DE ENTREGA CONTINUA DE SOFTWARE 74

4.1.1 INSTALACIÓN Y CONFIGURACIÓN DEL SOFTWARE BASE 74

4.1.2 CREACIÓN DE PROYECTO 76

4.1.3 CONFIGURACIÓN DEL PIPELINE 77

4.1.4 VERIFICACIÓN DE REPOSITORIO GIT 78

4.1.5 VERIFICACIÓN DEL PROCESO DE ENTREGA CONTINUA 78

4.1.6 PRUEBA DE DESPLIEGUE EN PRODUCCIÓN 79

5 CONCLUSIONES Y TRABAJOS FUTUROS 80

4.3 CONCLUSIONES 80

4.4 TRABAJOS FUTUROS 81

6 TRABAJOS CITADOS 82

ÍNDICE DE FIGURAS

Figura 2.1 - Stack de tecnologías para desarrollo de software en la Universidad de Cuenca 19
Figura 2.2 Despliegue de arquitectura monolítica (Villamizar, y otros, 2015)........ 20
Figura 2.3 - Despliegue de arquitectura de microservicios (Villamizar, y otros, 2015) 20
Figura 2.4 - Diagrama de secuencia de flujo de cambios a través del pipeline (Humble & Farley, 2010)........ 24
Figura 2.5. Relación entre integración continua, entrega y despliegue (Shahin, Babar, & Zhu, 2017) 24
Figura 2.6- Elementos del núcleo de un Pipeline de entrega continua (Elaboración propia) 25
Figura 2.7 - Desacoplamiento del script de construcción y el IDE (Duvall, Matyas, & Glover, 2007)........ 29
Figura 2.8- Componentes de un sistema de integración continua (Duvall, Matyas, & Glover, 2007)........ 35
Figura 2.9 - Ejemplo de un pipeline de despliegue (Chen, 2015) 37
Figura 2.10- Implementación basada en microservicios independiente del lenguaje (Jaramillo, 2016)........ 40
Figura 2.11 - Virtualización: Hypervisor vs Contenedores (Li, Kihl, Lu, & Andersson, 2017) 44
Figura 2.12 - Arquitectura de una imagen Docker (Pahl, 2015) 45
Figura 2.13 - Arquitectura de Docker (Paraiso, Challita, Al-Dhuraibi, & Merle, 2016) 47
Figura 3.1 - Resumen de desafíos, prácticas y sus relaciones para adopción de CI y CD (Shahin, Babar, & Zhu, 2017) 53
Figura 3.2 - Herramientas usadas para construir un pipeline de despliegue (Shahin, Babar, & Zhu, 2017) 54
Figura 3.3 - Pipeline de entrega continua. Perspectiva de aseguramiento de calidad (Gmeiner, Ramler, & Haslinger, 2015)........ 58
Figura 3.4 - Migración de un pipeline entrega continua de una arquitectura monolítica (a) a microservicios (b) (Balalaie & Heydarnoori,, 2016) 60
Figura 3.5 - Visualización de los momentos de aprovisionamiento de infraestructura en el pipeline de entrega continua de Gmeiner. 62

Figura 4.1 - Etapas y flujo de actividades del pipeline de entrega continua 65
Figura 4.2 - Arquitectura tecnológica para entrega continua de software en la Universidad de Cuenca 70
Figura 4.3 - Proceso de creación de prototipo 74
Figura 4.4 - Consola de administración de fabric8 75
Figura 4.5 - Ingreso al dashboard del equipo de proyecto 76
Figura 4.6 - Creación de aplicación en fabric8 76
Figura 4.7 - Creación de proyecto fabric8 con tecnología Spring Boot 77
Figura 4.8 - Creación de proyecto fabric8, parámetros de proyecto 77
Figura 4.9 - Selección de plantilla de pipeline de entrega continua de software 77
Figura 4.10 - Visualización del repositorio Git 78
Figura 4.11 - Pipeline de entrega continua 78
Figura 4.12 - Despliegue de solución a producción con Fabric8 79

ÍNDICE DE TABLAS

Tabla 2.1. Arquitectura de sistemas de control de versiones según tipo de acceso (Elaboración propia) 26
Tabla 3.1 - Desafíos de la entrega continua de software (Shahin, Babar, & Zhu, 2017) 49
Tabla 3.2 - Prácticas para entrega continua de software (Shahin, Babar, & Zhu, 2017)50
Tabla 3.3 - Lista de factores críticos para el éxito de la entrega continua (Shahin, Babar, & Zhu, 2017) 52
Tabla 3.4 - Herramientas para la entrega continua con software opensource (Gomede & Barros, 2015) 56
Tabla 3.5 - Elementos esenciales para entrega continua y despliegue de software (Pulkkinen, 2013) 57
Tabla 4.1 - Requerimientos no funcionales de la plataforma tecnológica para implementación de aplicaciones de la Universidad de Cuenca 63
Tabla 4.2 - Resumen de componentes de software de pipeline de entrega continua de software 70

CLÁUSULA DE LICENCIA Y AUTORIZACIÓN PARA LA PUBLICACIÓN EN EL REPOSITORIO INSTITUCIONAL

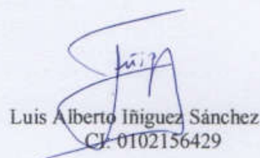
Universidad de Cuenca
Maestría en Gestión Estratégica de Tecnologías de la Información

CLÁUSULA DE LICENCIA Y AUTORIZACIÓN PARA LA PUBLICACIÓN EN EL REPOSITORIO INSTITUCIONAL

Luis Alberto Iñiguez Sánchez, en calidad de autor y titular de los derechos morales y patrimoniales del trabajo de titulación "Arquitectura Tecnológica para la entrega continua de software con despliegue en contenedores" de conformidad con el Art. 114 del CÓDIGO ORGÁNICO DE LA ECONOMÍA SOCIAL DE LOS CONOCIMIENTOS CREATIVIDAD E INNOVACIÓN reconozco a favor de la Universidad de Cuenca una licencia gratuita, intransferible y no exclusiva para el uso no comercial de la obra, con fines estrictamente académicos.

Así mismo, autorizo a la Universidad de Cuenca para que realice la publicación de este trabajo de titulación en el Repositorio Institucional, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior.

Cuenca, 6 de octubre de 2017



Luis Alberto Iñiguez Sánchez
CI: 0102156429

CLÁUSULA DE PROPIEDAD INTELECTUAL

Universidad de Cuenca
Maestría en Gestión Estratégica de Tecnologías de la Información

CLÁUSULA DE LICENCIA Y AUTORIZACIÓN PARA LA PUBLICACIÓN EN EL REPOSITORIO INSTITUCIONAL

Luis Alberto Iñiguez Sánchez, en calidad de autor y titular de los derechos morales y patrimoniales del trabajo de titulación "Arquitectura Tecnológica para la entrega continua de software con despliegue en contenedores" de conformidad con el Art. 114 del CÓDIGO ORGÁNICO DE LA ECONOMÍA SOCIAL DE LOS CONOCIMIENTOS CREATIVIDAD E INNOVACIÓN reconozco a favor de la Universidad de Cuenca una licencia gratuita, intransferible y no exclusiva para el uso no comercial de la obra, con fines estrictamente académicos.

Así mismo, autorizo a la Universidad de Cuenca para que realice la publicación de este trabajo de titulación en el Repositorio Institucional, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior.

Cuenca, 6 de octubre de 2017

Luis Alberto Iñiguez Sánchez
CI: 0102156429

DEDICATORIA

Dedico este trabajo a Dios por darme la posibilidad de mejorar cada día y la voluntad para hacerlo.

… a mi esposa, por todo el amor, la paciencia y el apoyo incondicional para afrontar cada desafío que se presenta en nuestras vidas.

… a mis hijas Bernarda, Rafaela y Martina, por ser la inspiración de cada paso que doy y por sacrificar el tiempo que debí dedicarles pero no pude hacerlo para alcanzar mis objetivos.

… a mis padres, por el ejemplo de vida y por todo lo que han entregado por sus hijos sin esperar nada a cambio.

Luis Alberto

AGRADECIMIENTOS

El presente proyecto se ha realizado con el apoyo incondicional del director de tesis, Ing. Víctor Saquicela, PhD, a quién expreso mis más sinceros agradecimientos por la motivación, guía y visión crítica que me ha permitido llevar a puerto seguro todo el esfuerzo realizado para alcanzar esta meta personal que venía siendo postergada y que ahora es una realidad.

1 INTRODUCCIÓN

El auge de la Computación en la Nube se debe a varios factores que han impulsado su adopción, como las importantes innovaciones de la virtualización y la computación distribuida, los avances en la velocidad y mayor acceso a Internet, y, la necesidad de acceso a la tecnología a costos asequibles.

En los últimos años los entornos de nube han evolucionado gracias a la denominada “tecnología de contenedores” que provee ambientes de componentes ligeros que facilitan la portabilidad de las aplicaciones entre nubes, mejorando significativamente la escalabilidad y rendimiento, ofreciendo un enfoque alternativo a la virtualización de servidores (Kang, Le, & Tao, 2016) .

Gran cantidad de organizaciones están cambiando la manera en la que consumen o proveen sus servicios gracias a la sencillez y a la facilidad de acceso a recursos de computación y servicios de TI que ofrece la computación en la nube dando soporte a su estrategia de negocio a través del desarrollo de capacidades que les permitan reaccionar rápidamente a las presiones del mercado.

Dentro de la Ingeniería de Software se han desarrollado metodologías que han emergido como una respuesta a la necesidad de las organizaciones de poder adaptarse al cambio, conocidas como metodologías ágiles. En su mayoría estas están enfocadas en entregas continuas para generar valor en la organización de manera temprana; sin embargo, la capacidad de reacción de los equipos de desarrollo se encuentra supeditada al nivel de automatización del proceso de construcción y aseguramiento de calidad del software (Vassallo, y otros, 2016).

Si bien las metodologías de desarrollo de software son importantes, se sugiere que si se desea practicar la integración y entrega continua de software, se ponga un énfasis especial en las decisiones de diseño arquitectónico (Kang, Le, & Tao, 2016), pues la arquitectura seleccionada puede afectar de manera determinante los objetivos del proyecto, más aún, cuando se desea obtener los beneficios de la computación en la nube desplegada sobre tecnología basada en contenedores que favorecen la portabilidad, escalabilidad y facilidad de despliegue de las aplicaciones (Balalaie & Heydarnoori,

2016). En este contexto, los microservicios [1] han emergido como un estilo arquitectónico nativo para entornos de nube que busca promover el bajo acoplamiento de las aplicaciones informáticas de gran complejidad mediante su descomposición en uno o más servicios que se constituyen en unidades de software autónomas, reemplazables y actualizables (Bakshi, 2017). Cada servicio puede ser desplegado de manera independiente el uno del otro en plataformas y *stacks* tecnológicos diferentes, característica que hace de los contenedores una tecnología idónea para facilitar la portabilidad en el despliegue desde el desarrollo hasta la puesta en producción del software (Jaramillo, 2016).

La Universidad de Cuenca a través de la Dirección de Tecnologías de la Información y Comunicación ofrece a la Comunidad Universitaria servicios relacionados con las TICs para dar soporte a las actividades administrativas, académicas y de investigación. La Dirección dispone de una unidad de desarrollo de software que se encuentra encargada de dar mantenimiento a los sistemas de información que se encuentran en producción y el desarrollo de nuevos proyectos enfocados a fortalecer la estrategia institucional. La carencia de una arquitectura estándar para el desarrollo de software ha provocado que el despliegue de nuevas versiones de las aplicaciones sea una tarea extensa y complicada, introduciendo con facilidad errores de regresión en el software que se pone en producción dificultando el aseguramiento de calidad.

Entre las acciones de mejora, se ha planteado la necesidad de adoptar prácticas y tecnologías que permitan a la unidad de desarrollo establecer un proceso de integración y entrega continua de software, de manera que se puedan alcanzar niveles adecuados de fiabilidad y menores tiempos en las operaciones de despliegue en los ambientes de producción.

El presente trabajo propone una arquitectura tecnológica para dar soporte a un proceso automatizado para la entrega continua de software, en función de los requerimientos de entorno para el despliegue de aplicaciones basadas en Microservicios con infraestructura de nube desplegada sobre contenedores.

[1] https://martinfowler.com/articles/microservices.html

1.1 ESTRUCTURA DEL TRABAJO DE TESIS

El presente trabajo se encuentra organizado en seis capítulos: en el Capítulo 1 se determina los objetivos y el alcance del proyecto; en el Capítulo 2, se establece el marco teórico de los temas relacionados con la entrega continua de software y elementos colaterales que complementan el contexto del proyecto; en el Capítulo 3, se presenta el trabajo realizado hasta el momento que permitirá conocer el estado actual de los desafíos, prácticas, arquitecturas y herramientas que se relacionan con este proyecto; en el Capítulo 4, se presenta la arquitectura tecnológica seleccionada para el *pipeline* de entrega continua de software y la implementación del prototipo para la verificación de la prueba de concepto, para finalmente en el Capítulo 5, presentar las conclusiones del trabajo realizado y las opciones para el trabajo futuro que permita complementar el planteamiento realizado.

1.1 OBJETIVOS

1.1.1. OBJETIVO GENERAL

Plantear una arquitectura tecnológica que facilite la adopción de un proceso de integración y entrega continua de software, acorde al ecosistema de la Universidad de Cuenca.

1.1.2. OBJETIVOS ESPECÍFICOS

- Evaluar tecnologías que permitan un proceso de integración y entrega continua de software con despliegue en contenedores.
- Establecer el diseño y la arquitectura de componentes de software a partir de los requerimientos funcionales y no funcionales que permitan conseguir un proceso de integración y entrega continua de software.
- Desarrollar una prueba de concepto del ciclo de vida del desarrollo de producto con la arquitectura seleccionada.

1.2 ALCANCE

Para establecer los criterios base en la definición de los componentes de software de la arquitectura tecnológica se identificarán los requerimientos esenciales que permitirían

alcanzar los objetivos de construcción con integración continua, pruebas y automatización en el aprovisionamiento de infraestructura para el despliegue de las aplicaciones. Una vez establecidos los parámetros se analizarán las opciones que existen en el mercado teniendo en cuenta los requerimientos de la Universidad de Cuenca en cuanto a esquemas de licenciamiento, compatibilidad, interoperabilidad, soporte, infraestructura de despliegue (contenedores), políticas de actualización, ambientes de usuario final, etc.

Con las alternativas seleccionadas se establecerán los componentes de alto nivel que participarán en el diseño de la arquitectura tecnológica además de los componentes tecnológicos que manejarán el versionamiento y control de calidad del código fuente, la construcción de binarios, el marco de trabajo para la ejecución de pruebas unitarias, de integración y de interfaz de usuario y el despliegue de las aplicaciones en un entorno de alta disponibilidad.

Finalmente, se desarrollará la prueba de concepto del ciclo de vida del proceso con un mayor detalle en la integración que en el despliegue, pues es el área en la que mayor aporte se requiere.

2 MARCO TEÓRICO

En el contexto de aplicación de esta tesis, existen restricciones en cuanto a la pila de tecnologías, estilo arquitectónico e infraestructura de despliegue de las aplicaciones para los nuevos proyectos de desarrollo de software en la Universidad de Cuenca.

El propósito de este capítulo es presentar los fundamentos que permitirán comprender los conceptos y enfoques presentados en el presente trabajo. Para proporcionar información estructurada, cada sección de este capítulo se enfoca en un campo específico en relación a los elementos requeridos para la definición de una arquitectura tecnológica para la entrega continua de software, de aplicaciones diseñadas con un estilo arquitectónico basado en Microservicios e infraestructura de despliegue sobre contenedores.

2.1 ANTECEDENTES

La disciplina de la Ingeniería de Software se propone formalmente en 1968, año en el que la OTAN [2] organiza la primera conferencia para tratar a lo que entonces se denominó "La crisis del software", causada por "la dificultad de escribir programas libres de defectos, fácilmente comprensibles, y que sean verificables" (Randell & Naur, 1969). Desde entonces los esfuerzos de los investigadores han estado enfocados en conseguir que los proyectos de desarrollo de software sean más predecibles en cuanto a la asertividad en los plazos y presupuestos para la ejecución de los proyectos, y una mejor calidad del software considerando los cambios constantes que deben realizarse a las aplicaciones para adaptarse a las necesidades de los usuarios.

Durante los últimos 50 años se han formulado diversidad de metodologías para gestionar el ciclo de vida del software, entre las más conocidas: el modelo en cascada, el modelo de desarrollo incremental, el modelo de desarrollo evolutivo, el modelo de prototipado de requerimientos, el modelo en espiral, el modelo concurrente, la Ingeniería de Software orientada a la reutilización, etc. (Sommerville, 2002); sin embargo, es a inicios del año 2001 que un creciente interés por nuevos enfoques en el desarrollo de software, llevaron a un grupo de 17 de los más fervientes promotores del movimiento de metodologías ágiles a conformar el grupo *Agile Software Development Aliiance*, quienes

[2] https://www.nato.int/

en su primera reunión emiten un documento simbólico conocido como “El manifiesto ágil” (Fowler & Highsmith, The Agile Manifesto, 2001). El manifiesto planteó ideas que revolucionaron los métodos tradicionales de gestión de proyectos constituyéndose en un importante avance en la compresión de la forma de trabajar de los desarrolladores de software, proponiendo la valoración de: a) “Individuos e interacciones sobre procesos y herramientas”, b) “Software funcionando sobre documentación extensiva”, c) “Colaboración con el cliente sobre negociación contractual”, y d) “Respuesta ante el cambio sobre seguir un plan”. Para marcar diferencia con respecto a un proceso tradicional, el manifiesto caracteriza mediante 12 *principios* las ideas centrales de un proceso ágil. El primero de ellos establece que: “Nuestra prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor” (Beck, Agile Manifest Principles , 2001; Koc & Uz, 2015), enfatizando explícitamente la entrega frecuente de software funcional. Esto se consigue solo si el software es desplegado en un ambiente de producción proveyendo todas las funcionalidades necesarias para el usuario final; y para que esto suceda, a pesar de que los desarrolladores hayan conseguido escribir código que funcione en sus computadores, deberán transcurrir varias etapas dependiendo de los métodos de validación del software antes de lograr que las aplicaciones se encuentren operativas dentro de la organización (Humble & Farley, 2010).

Actualmente, en la Universidad de Cuenca la liberación de producto se realiza creando los entornos de ejecución de las aplicaciones individualmente para los ambientes de pruebas y de producción de manera manual, convirtiendo el despliegue de las aplicaciones en una tarea extensa y complicada propensa a errores de regresión.

La Dirección de Tecnologías de la Información y Comunicación está introduciendo cambios en la arquitectura de las aplicaciones y el *stack* de tecnologías de los nuevos proyectos de desarrollo, oportunidad que se desea aprovechar para implementar un proceso automatizado para la entrega continua de software alineado a una metodología de desarrollo con enfoque ágil.

2.2 Plataforma tecnológica para implementación de aplicaciones de la Universidad de Cuenca

2.2.1 Componentes de la plataforma de desarrollo de software

La Figura 2.1 muestra el *stack* de tecnologías seleccionadas para la plataforma de desarrollo de software de la Universidad de Cuenca, la cual contiene tres componentes: i) servidor web, desde dónde se sirven recursos estáticos a los clientes, en este caso las páginas Html5 [3] en combinación con el framework Bootsrap[4] y los artefactos de Angular JS[5]; ii) en el lado del servidor, Spring Security[6] para el aseguramiento de los servicios, Spring Boot[7] para la gestión de microservicios, gateways y servicios rest, componentes Java [8]que contienen los servicios de la lógica de negocio, Spring Data[9] para la capa de acceso a datos, y; iii) los repositorios de datos, que en este caso se encuentran representados en el gráfico por los motores de base de datos con los que cuenta actualmente la Universidad de Cuenca que son Oracle[10], Postgresql [11]y Mysql[12].

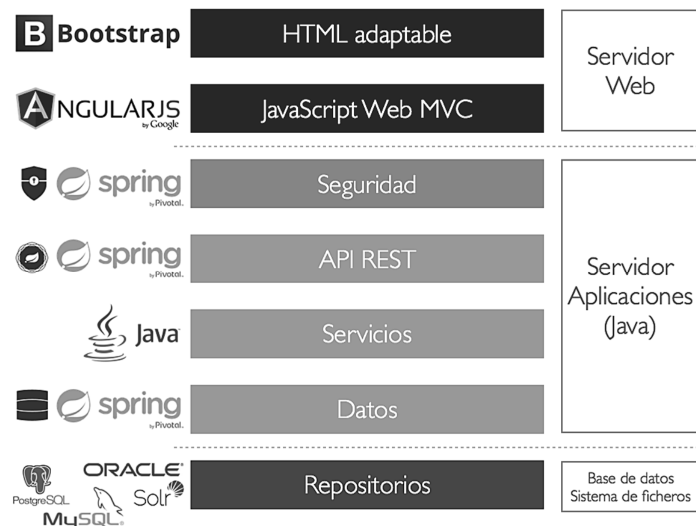


Figura 2.1 - Stack de tecnologías para desarrollo de software en la Universidad de Cuenca

---

[3] https://www.w3.org/TR/html5/
[4] http://getbootstrap.com/
[5] https://angularjs.org/
[6] https://projects.spring.io/spring-security/
[7] https://projects.spring.io/spring-boot/
[8] https://go.java/index.html?intcmp=gojava-banner-java-com
[9] https://projects.spring.io/spring-data/
[10] https://www.oracle.com/database/index.html
[11] https://www.postgresql.org/
[12] https://www.mysql.com/

2.2.2 COMPONENTES DE LA INFRAESTRUCTURA DE DESPLIEGUE

Para facilitar la escalabilidad de las aplicaciones, la Universidad de Cuenca requiere cambiar el despliegue de las aplicaciones de plataformas servidores virtualizados compuestas de máquinas virtuales a un esquema de computación en la nube con contenedores *Docker*[13]. La Figura 2.2 muestra el estado actual de despliegue de las aplicaciones monolíticas, frente a la arquitectura de despliegue esperada basada en contenedores (Figura 2.3).

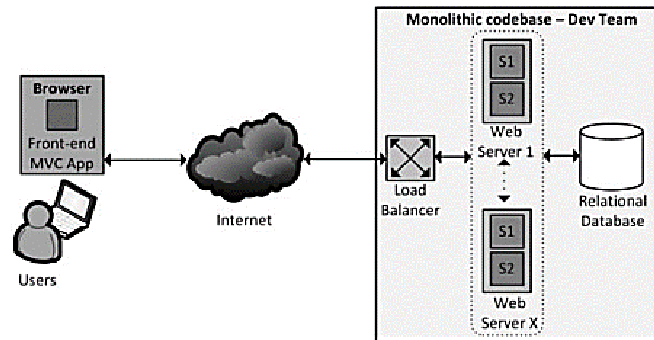


Figura 2.2 Despliegue de arquitectura monolítica (Villamizar, y otros, 2015)

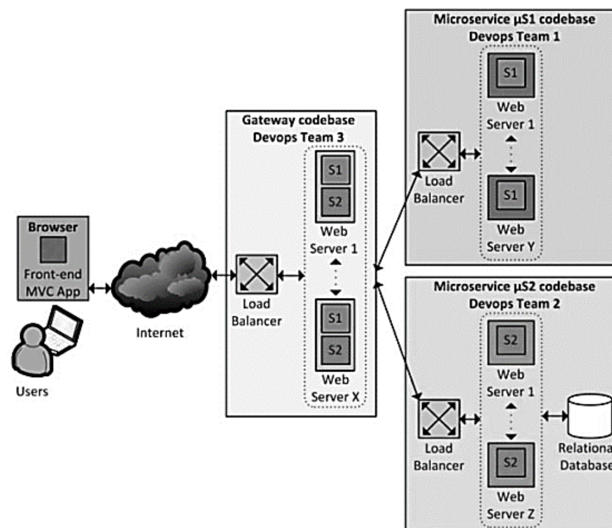


Figura 2.3 - Despliegue de arquitectura de microservicios (Villamizar, y otros, 2015)

[13] https://www.docker.com/

2.3 ENTREGA CONTINUA DE SOFTWARE

Las soluciones empresariales hoy en día están compuestas de distintas fuentes e integran software de diversa base tecnológica. La infraestructura y ambientes de desarrollo, pre producción y producción suelen ser manejados por separado, en donde los motores de persistencia de datos, servidores web, servidores de aplicaciones, etc., tienen sus propios parámetros de configuración pues están destinados a proveer servicios para diferentes propósitos durante la construcción y despliegue de las aplicaciones. En proyectos modernos el software se despliega en múltiples plataformas: aplicaciones de escritorio, aplicaciones móviles (iOS [14]y Android[15]), aplicaciones web, aplicaciones de capa media desplegada en la nube, consumiendose APIs basadas en servicios web o servicios REST[16], integradas a redes sociales y mensajería instantánea (*SMS, Whatsapp*[17], *Snapchat*[18], etc.), todo al mismo tiempo y para una sola solución; como consecuencia, la complejidad de las operaciones a lo largo de todo el ciclo de vida del desarrollo de software es cada vez mayor (Soni , 2015).

Una ejecución manual de los procesos de construcción, despliegue, pruebas y liberación de producto es propensa a errores por la manipulación que se da durante la transformación de los artefactos producidos por el equipo de trabajo; desde la recepción de los cambios realizados en el repositorio de código fuente, hasta pasar las pruebas de aceptación que permiten la liberación del producto. La automatización de lo que se conoce como “*Pipeline* de entrega continua de software” permite encontrar con mayor facilidad y de manera temprana fallos en la programación, introducción de errores de regresión y errores en la integración de los componentes de software generando la retroalimentación que permitirá obtener un alto nivel de confianza y control sobre el proceso (Duvall, Matyas, & Glover, 2007).

En la literatura revisada se encontraron varias definiciones; sin embargo, por su simplicidad, en este trabajo se definirá a la entrega continua de software como “una disciplina de desarrollo de software en donde éste se construye de manera que pueda ser liberado a producción en cualquier momento” (Fowler, 2013). En la práctica, ésta consiste

[14] https://www.apple.com/ios/ios-11/
[15] https://www.android.com/
[16] https://spring.io/understanding/REST
[17] http://whatsapp.com/
[18] https://www.snapchat.com/l/es/

de su implementación técnica denominada "*Pipeline* de entrega continua de software", y el procedimiento que se encuentra instrumentado mediante principios y prácticas.

2.3.1 PRINCIPIOS PARA LA IMPLEMENTACIÓN DE ENTREGA CONTINUA

Humble et al. (2010) sintetizan una serie de principios que se identificaron como patrones durante la ejecución de varios proyectos para que un proceso de entrega continua sea efectivo, estos se resumen a continuación:

a) **Crear un proceso confiable y repetible para liberar software**
Se debe usar el mismo proceso de liberación en todos los entornos. Para liberar una característica o mejora es necesario desplegarla en distintos ambientes durante las pruebas de integración, pruebas de aceptación hasta la liberación a producción, por tanto, es necesario gestionar la configuración de los ambientes en cada etapa del proceso.

b) **Automatizar casi todo**
Automatice las construcciones de software, pruebas, despliegues, cambios de configuración y todo lo demás. Los procesos manuales son intrínsecamente menos repetibles, más propensos a errores y menos eficientes. Una vez que se automatiza un proceso, se necesita menos esfuerzo para ejecutar y monitorear su progreso, asegurando la obtención de resultados consistentes.

c) **Mantenga todo bajo control de versiones**
Código, configuración, scripts, bases de datos, documentación, etc. Tener una fuente confiable, brinda una base estable para construir procesos.

d) **Si le duele, hágalo con más frecuencia y supere el dolor**
Enfrentar las cosas duras primero. Las tareas que consumen mucho tiempo o propensas a errores deben tratarse tan pronto como sea posible. Una vez que se haya solucionado los problemas dolorosos, el resto será más fácil de perfeccionar.

e) **Constrúyalo con calidad**
Se debe crear ciclos de retroalimentación cortos para tratar los errores tan pronto como se crean. Al tener problemas devueltos a los desarrolladores tan pronto como fallan las pruebas posteriores a la construcción, es posible producir código de mayor calidad más rápidamente. Además, se encontrarán menos problemas más adelante en el proceso, en donde es más costoso corregirlos.

f) **Considere una tarea terminada cuando esté liberada a producción**
Una característica o corrección al software se considera realizada solo cuando está en producción. Tener una definición clara de "hecho" desde el principio ayudará a que todos se comuniquen mejor y se den cuenta del valor de cada característica.

g) **Todos son responsables del proceso de entrega**
La responsabilidad incluye a todos los participantes del proceso y debe extenderse todo el camino hasta llegar a producción. El cambio cultural puede ser el más difícil de implementar. Sin embargo, contar con el apoyo de la administración y un líder comprometido sin duda ayudará.

h) **Practique la mejora continua**
De los 8 principios de entrega continua enumerados, este principio es el más importante para la automatización efectiva. Es absolutamente necesario tener una cultura que busque la mejora continua. La automatización representa la máxima expresión de un proceso iterativo de mejora continua que hace que la práctica sea perfecta.
(Humble & Farley, 2010)

2.3.2 PIPELINE DE ENTREGA CONTINUA DE SOFTWARE

Un *pipeline* de entrega continua es un proceso automatizado que se encarga de llevar el software desde el repositorio de código fuente hasta el usuario final. El proceso involucra la construcción de binarios seguido de varias etapas de prueba hasta el despliegue, ejecutado y monitoreado mediante una herramienta informática de integración y liberación continua de software. La Figura 2.4 visualiza a manera de diagrama de secuencia como fluyen los cambios a través del *pipeline* de entrega continua.

Para lograr su propósito se apoya en dos prácticas esenciales: la “integración continua” y la “automatización del despliegue” para obtener una versión del software que se encuentra en un estado listo para producción (Fitzgerald & Stol, 2015). La Figura 2.5 muestra de manera simplificada las relaciones entre actores, componentes y prácticas en un *pipeline* de entrega continua de software.

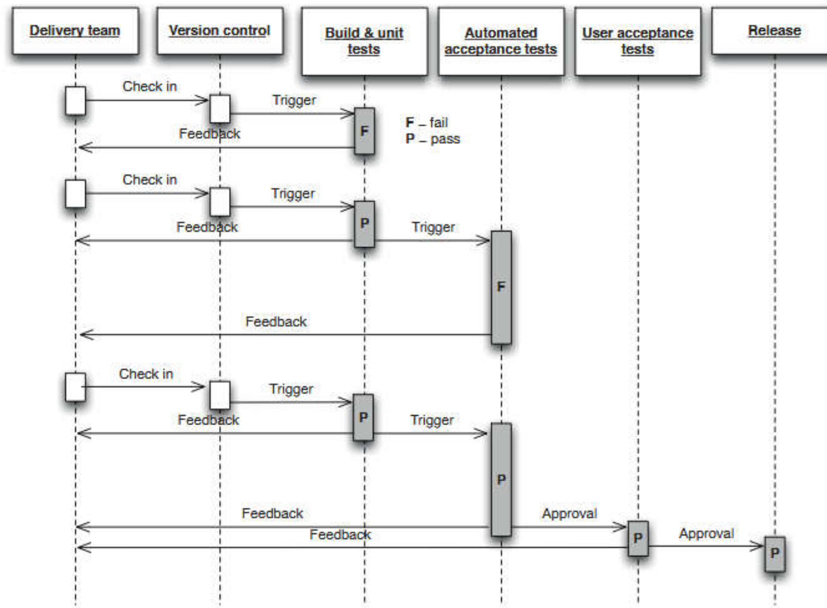


Figura 2.4 - Diagrama de secuencia de flujo de cambios a través del pipeline (Humble & Farley, 2010)

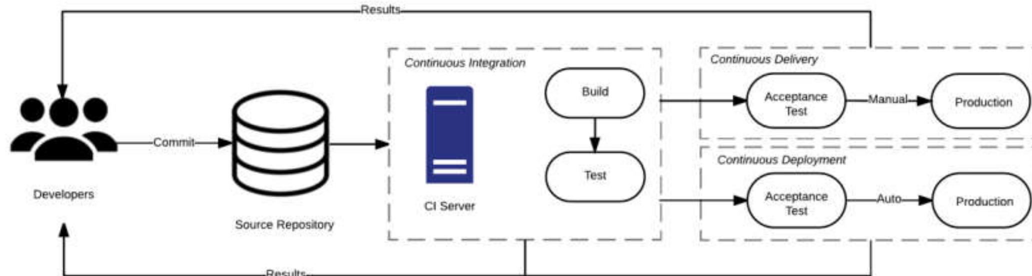


Figura 2.5. Relación entre integración continua, entrega y despliegue (Shahin, Babar, & Zhu, 2017)

En la revisión sistemática de literatura realizada sobre 69 artículos por Shahin et al. (2017) se determina que existe un patrón en el que se define como elementos esenciales para una implementación exitosa de un *pipeline* para la entrega continua de software: i) Control de versiones, ii) servidor de integración continua, iii) herramienta para construcción, iv) herramientas para pruebas, v) herramienta para gestión de configuración y aprovisionamiento, vi) servidor de despliegue. La Figura 2.6 muestra la relación de los elementos esenciales para la entrega continua de software.

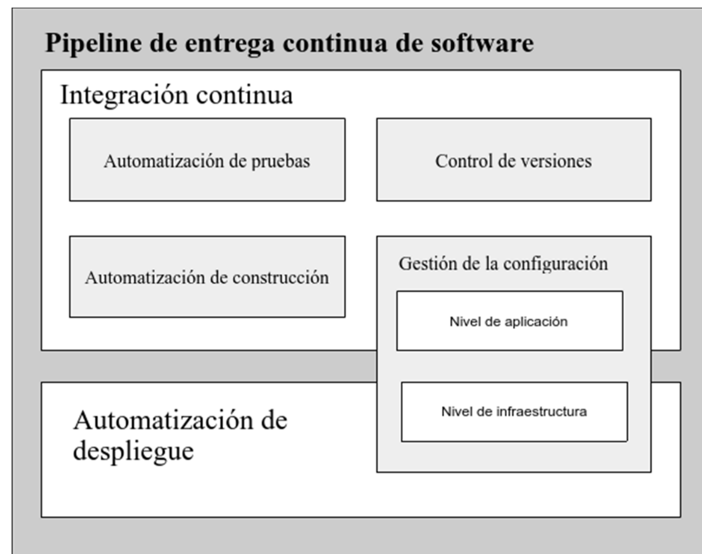


Figura 2.6- Elementos del núcleo de un Pipeline de entrega continua (Elaboración propia)

A continuación se introducen los conceptos más relevantes de cada uno de los elementos que conforman el núcleo de un *pipeline* de entrega continua de software.

2.3.2.1 CONTROL DE VERSIONES

Un equipo de desarrollo de software usualmente trabaja en tareas simultáneas generando múltiples artefactos de Ingeniería de Software. Para conseguir una sola base de código es necesario converger todas las actividades paralelas para poder construir, verificar y entregar un producto. En este escenario, es común que se den conflictos como consecuencia de los cambios aplicados de manera concurrente e incluso se pueden causar problemas graves que no pueden ser resueltos automáticamente, por lo tanto, provocar inconsistencias o la introducción de errores de regresión en el repositorio.

Las actividades de control de versiones se realizan mediante sistemas de control de versiones, su propósito es: i) proveer el acceso a todas las versiones de todos los artefactos permitiendo que los desarrolladores puedan ver los cambios introducidos, en que archivos, cuándo y por quién, y, ii) facilitar la comunicación.

Desde los años 70 los sistemas de control de versiones se han venido creando basados en 4 arquitecturas (Koc & Uz, 2015). La clasificación de los sistemas de control de versiones según su arquitectura se muestra en la Tabla 2.1.

Tabla 2.1. Arquitectura de sistemas de control de versiones según tipo de acceso (Elaboración propia)

Tipo de acceso a repositorio	Descripción
Centralizado - Local no compartido	- Repositorio único en computador del desarrollador. - Acceso a repositorio exclusivamente local. - Todos los cambios deben ser realizados en el mismo equipo. - Es la arquitectura más primitiva y menos útil para trabajar en equipo
Centralizado - Local compartido	- Repositorio único en computador del desarrollador - Acceso a repositorio mediante red de área local. - Acceso a manera de carpeta compartida para múltiples usuarios.
Remoto - Cliente/Servidor	- Repositorio almacenado en un servidor - Acceso remoto a repositorio mediante red.
Punto a punto - Distribuido	- Repositorio almacenado en copia local en computador del desarrollador. - Cada copia local se maneja como una ramificación del código. - Los desarrolladores depositan los cambios en su propia ramificación - De considerarse necesario se pueden depositar cambios en ramificaciones de otros usuarios. - La rama principal (*trunk*) es la que se considera la última versión del código.

Los sistemas de control de versiones modernos están implementados con una arquitectura distribuida, facilitando la adopción de metodologías ágiles en los equipos de desarrollo a través de procesos de integración más rápidos, y mejorando el aislamiento entre los desarrolladores para garantizar la independencia de los cambios (Just, Herzig, Czerwonka, & Murphy, 2016).

2.3.2.2 GESTIÓN DE LA CONFIGURACIÓN

La gestión de la configuración se refiere al proceso por el cual todos los artefactos relevantes para un proyecto, y las relaciones entre ellos, son almacenados, recuperados, identificados de manera única, y modificados (Humble & Farley, 2010).

Otros autores definen la entrega continua de software en el contexto de la Gestión de la Configuración como la automatización de los procesos de instalación y configuración de diferentes ambientes de ejecución e infraestructura (Pulkkinen, 2013). Humble y Farley (2010) agregan a este concepto otros elementos como configuraciones de aplicación y en general todos los artefactos requeridos para el proyecto.

La estrategia de gestión de configuración determina como se van a gestionar los cambios que se dan dentro de un proyecto. Para automatizar adecuadamente los ambientes de ejecución de las aplicaciones y el aprovisionamiento de la infraestructura, es necesario separar la gestión de la configuración en dos niveles (Benson, Prevost, & Rad , 2016):

Nivel de aplicación: incluye el almacenamiento de todas las configuraciones relacionadas con las aplicaciones que son necesarias para su construcción, instalación y ejecución.

Nivel de infraestructura: incluye el almacenamiento de todas las configuraciones necesarias para establecer una infraestructura sobre la cual las aplicaciones van a ejecutarse, gestionándose información como configuraciones de red, sistemas operativos, etc.

Las acciones que permiten automatizar la parametrización de los entornos de ejecución y el aprovisionamiento de infraestructura generalmente se expresan mediante *scripts* declarativos (Meyer, Healy, Lynn, & Morrison, 2013). La motivación para una apropiada gestión de la configuración diferenciando aplicación de infraestructura, es mantener los scripts de los distintos ambientes y configuraciones consistentes, con un adecuado seguimiento y control de cambios a lo largo de las etapas de desarrollo, pruebas unitarias, pruebas de integración, pruebas de aceptación y despliegue en producción.

Omisiones en la gestión de la configuración pueden introducir problemas en el software que son difíciles de reproducir y depurar. Incluso cuando un software falla en producción, el mismo software podría trabajar en un entorno de pruebas a causa de ambientes de ejecución inconsistentes (Gmeiner, Ramler, & Haslinger, 2015).

El control de versiones es el núcleo de las actividades de gestión de la configuración. Además del código fuente, como mínimo, se deben gestionar todos los artefactos

necesarios para la construcción de los archivos binarios de las aplicaciones desde cero y la preparación de sus entornos de ejecución. Los artefactos que se encuentran bajo el control de versiones deben ser accesibles y modificables por todos los miembros del equipo.

2.3.2.3 AUTOMATIZACIÓN DE LA CONSTRUCCIÓN

Para convertir los archivos fuente de una aplicación en componentes de software listos para ser ejecutados se requiere a menudo compilar, estructurar y empaquetar archivos y carpetas, verificar dependencias de otras librerías, etc. El proceso de automatización de estas tareas se lo denomina "automatización de la construcción", siendo su objetivo, reducir errores permitiendo construir una aplicación o un componente de software de manera consistente y repetible, facilitando la liberación de cambios del producto mitigando riesgos en la entrega del software. En la práctica, esto significa que se debería poder construir una aplicación ejecutando un solo comando (Fowler, 2006).

Si bien los entornos de desarrollo integrado (IDE)[19] incluyen herramientas para la automatización de la construcción del software, en muchas ocasiones, éstas son utilizadas exclusivamente en construcciones para pruebas preliminares durante el desarrollo de software. Los entornos automatizados para construcción de software deben ser independientes del IDE de manera que puedan ser compartidos y manejados bajo control de versiones convirtiéndose en artefactos reutilizables por otros desarrolladores y la herramienta de integración continua de software (Duvall, Matyas, & Glover, 2007). La construcción del software se define de manera declarativa mediante scripts y en muchas ocasiones contemplan como una etapa más dentro de su proceso la ejecución de pruebas unitarias. Estos han permanecido vigentes por décadas, siendo el más antiguo de ellos el comando '*make*[20]' utilizado ampliamente en sistemas UNIX[21].

La Figura 2.7 muestra el desacoplamiento de los componentes de software y la reutilización del script de construcción.

[19] https://www.veracode.com/security/integrated-development-environments
[20] http://archive.oreilly.com/pub/a/linux/excerpts/9780596100292/gnu-make-utility.html
[21] https://hipertextual.com/archivo/2014/05/que-es-unix/

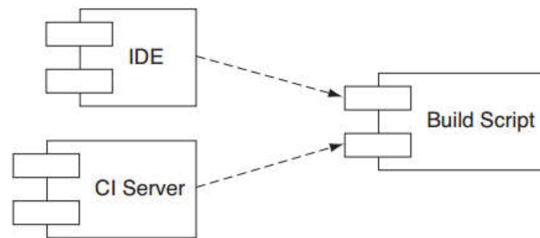


Figura 2.7 - Desacoplamiento del script de construcción y el IDE (Duvall, Matyas, & Glover, 2007)

La creación de un script por separado es importante porque cada programador puede utilizar un IDE diferente y aunque se utilice el mismo, se podría aplicar configuraciones diferentes. Por otra parte, con el mismo *script*, el servidor de integración continua puede ejecutar una construcción automatizada sin intervención humana.

2.3.2.4 AUTOMATIZACIÓN DE DESPLIEGUE

El término "*deployment scripting*" es utilizado para describir la secuencia de comandos que implementan el proceso de automatización del despliegue de una aplicación (Duvall, Matyas, & Glover, 2007). Las secuencias de comandos de despliegue deben abarcar tanto el proceso de actualización de una aplicación como su instalación desde cero (Humble & Farley, 2010). La misma secuencia de comandos de despliegue se debe usar para desplegar la aplicación en diferentes entornos, lo que significa que las configuraciones específicas para cada entorno deben recuperarse de archivos de configuración. Esto crea la conexión entre la automatización de despliegue, la gestión de la configuración y el control de versiones (Virmani, 2015). Otro factor importante es que se debe tener la capacidad de revertir los cambios implementados, en caso de que se produzcan errores. El proceso de reversión se puede lograr manteniendo la versión del software implementada previamente o volviéndola a implementar desde cero. Se puede entonces definir a la automatización del despliegue como el proceso de configuración automática de los ambientes y la infraestructura en la que se ejecuta el software, así como el despliegue de la aplicación en ese entorno.

2.3.2.5 AUTOMATIZACIÓN DE LAS PRUEBAS

Para contextualizar la automatización de las pruebas, es necesario en primera instancia establecer que son las "pruebas de software". Las pruebas de software se

definen como “el proceso de ejecución de un programa con la intención de encontrar errores" (Myers, Sandler, & Badgett, 2011). En esta definición se destaca como el objetivo final de las pruebas encontrar errores en el software, mas no demostrar que no existen. En la actividad de desarrollo de software los errores de las aplicaciones deben detectarse y repararse lo antes posible para mejorar la calidad del software y reducir el costo de las pruebas, considerando que muchas organizaciones gastan hasta el 40% de sus recursos en pruebas en su intento por conseguir una cobertura del 100%, tarea que es prácticamente imposible (Elberzhager, Rosbach, Münch, & Eschbach, 2012).

La automatización consiste en la integración de herramientas en el ambiente de pruebas de manera que su ejecución, seguimiento y comparación de resultados se realice con la menor intervención humana posible (Koirala & Sheikh, 2009). La automatización permite la acumulación de casos de prueba a lo largo del ciclo de vida de la aplicación de manera que siempre se puede probar corrección de errores, características existentes y características nuevas del software (Hayes, 2004).

Los elementos que se deben contemplar en el proceso son: la generación de pruebas, configuración del sistema, simuladores, registro de actividades y análisis de cobertura. Esto muestra que la automatización de pruebas es un tema amplio y que, dependiendo del entorno, puede tener diferentes significados de lo que realmente está cubierto.

La estrategia seleccionada tiene un rol preponderante para alcanzar los objetivos en la automatización de las pruebas, siendo muy común que ésta no sea implementada correctamente (Berner, Weber, & Keller, 2005). La estrategia establece qué probar y qué tan extensas deben ser, permitiendo al equipo de pruebas enfocarse en partes críticas del software en las cuales los errores son más propensos a presentarse, teniendo en mente que no todas las pruebas deben ser automatizadas; las pruebas que requieren un trabajo manual más intensivo y que son ejecutadas a menudo son usualmente buenas candidatas a ser automatizadas.

Según Berner et al. (2005), las pruebas se repiten con mucha más frecuencia de la esperada, lo que debe tenerse en cuenta al evaluar si la automatización de pruebas debe aplicarse en un proyecto. Los autores indican que si un caso de prueba debe ejecutarse más de 10 veces, su automatización ya debería ser considerada. También argumentan que se debe tener en cuenta que las pruebas deben implementarse en un nivel correcto. Por

ejemplo, la lógica del programa interno debe probarse a nivel de prueba unitaria en lugar de a través de la interfaz del usuario. Probar las cosas correctas en el nivel correcto facilita la depuración de un caso de prueba fallido y también hace que la creación de las pruebas sea más sencillo.

Beneficios de la automatización de las pruebas

La automatización de las pruebas tiene múltiples beneficios comparado con las pruebas manuales. Entre los principales beneficios podemos mencionar los siguientes:

- Las pruebas automatizadas incrementan la calidad del software habilitando una mejor cobertura de las pruebas (Karhu, Repo, Taipale, & Smolander, 2009).
- La cobertura de las pruebas miden que tan bien se ha probado el código (Marick, 1997).
- La automatización de las pruebas reduce el tiempo de prueba, lo que significa que el tiempo necesario para ejecutar una cierta cantidad de pruebas disminuye (Karhu, Repo, Taipale, & Smolander, 2009).
- Los períodos de pruebas son más cortos y las pruebas se pueden ejecutar con mayor frecuencia (Berner, Weber, & Keller, 2005).
- Cuando la eficacia de las pruebas aumenta, se pueden implementar pruebas más exhaustivas e integrales (Graham & Fewster, 1999).
- Cuando tareas manuales que son repetitivas se automatizan, el equipo de pruebas puede enfocarse en escenarios más complejos y en la construcción de mejores pruebas (Fecko & Lott, 2002).
- La reutilización de una prueba permite distribuir el costo de su implementación entre las veces que se ejecuta la prueba (Graham & Fewster, 1999). A largo plazo, hará que la creación de pruebas automatizadas sea económicamente competitiva en comparación con las pruebas manuales (Karhu, Repo, Taipale, & Smolander, 2009).
- Al contar con una sólida automatización de pruebas, la confianza hacia la calidad del software aumenta. Los cambios y las versiones se pueden realizar con mayor confianza; sin embargo, tener pruebas de baja calidad puede llevar a una falsa sensación de seguridad (Graham & Fewster, 1999).

- Las fallas son más fáciles de reproducir cuando la ejecución de la prueba es automática (Dustin, Garrett, & Gauf, 2009).
- Las pruebas automatizadas permiten ser ejecutadas en diferentes configuraciones de hardware (Graham & Fewster, 1999).
- La automatización de pruebas también permite nuevos tipos de pruebas que no pueden realizarse manualmente, como las pruebas de rendimiento (Graham & Fewster, 1999).

Desafíos de la automatización de las pruebas

En la revisión de la literatura se evidencian retos que se deben afrontar para una implementación exitosa de un proceso de automatización de las pruebas. A continuación se cita las más relevantes:

- Programadores y miembros del equipo de proyecto deben salir de su zona de confort para adoptar las prácticas requeridas para la automatización de pruebas.
- En el corto plazo la automatización de pruebas representa un costo mayor en relación a las pruebas manuales (Karhu, Repo, Taipale, & Smolander, 2009).
- La implementación de casos de prueba automatizados requiere inicialmente más esfuerzo que la documentación manual de casos de prueba (Dustin, Garrett, & Gauf, 2009).
- La automatización de pruebas también requiere un conjunto específico de habilidades y, por lo tanto, los evaluadores necesitan capacitación antes de poder tener éxito en el proceso (Rafi, Kiran, Petersen, & Mäntylä, 2012).
- Se pueden generar falsas expectativas sobre la automatización de pruebas pensando que se reduciría automáticamente el esfuerzo requerido para la ejecución de las pruebas (Myers, Sandler, & Badgett, 2011).
- Si no se asigna el tiempo y los recursos suficientes para la implementación de la automatización de pruebas en un proyecto, las personas la pasarán por alto para buscar conseguir que las funcionalidades del software se realicen dentro del presupuesto asignado (Karhu, Repo, Taipale, & Smolander, 2009) (Vassallo et al., 2016).
- Según Graham y Fewster (1999) las organizaciones generalmente asumen que la automatización de pruebas se la considera a nivel de proyecto. Este es un

problema, sobre todo en grandes organizaciones que no tienen estándares sobre cómo debe implementarse la automatización de pruebas. Esto puede llevar a implementaciones diferentes generando costos iniciales más altos para todos los proyectos, dificultando la rotación del personal de pruebas en diferentes proyectos.

- La estrategia de pruebas es una parte central de la automatización de pruebas y de las pruebas en general (Dustin, Garrett, & Gauf, 2009). Si no existe una estrategia de pruebas sólida, la automatización de las pruebas no será suficiente. Graham y Fewster (1999) describen este problema estableciendo que "automatizar el caos solo produce un caos más rápido".
- Los requisitos de software deben estar de acuerdo con su intención real; si los requerimientos están mal establecidos, las aplicaciones serán programadas en función de estos, por tanto, las pruebas no podrán detectar errores (Haugset & Hanssen, 2008). Las pruebas deben estar actualizadas para reflejar los cambiantes requerimientos de las organizaciones, caso contrario, no aseguran de manera confiable la calidad del software (Graham & Fewster, 1999).
- Se debe tomar en cuenta que una prueba automatizada no puede encontrar defectos complejos; una persona experimentada en pruebas es capaz de hacerlo (Berner, Weber, & Keller, 2005). El propósito de las pruebas automatizadas es asegurar que la funcionalidad existente trabaje después de que se ha introducido nuevo código en el software o si el software es ejecutado en un nuevo ambiente (Graham & Fewster, 1999).
- Según Berner et al. (2005), la mayoría de los defectos encontrados en las pruebas automáticas son detectados durante la creación de los casos de prueba. Validan su argumento en base a los hallazgos de Kaner (1997), quien afirma que del 60% al 80% de los errores se encuentran durante la fase de desarrollo de los casos de prueba automatizados.
- No todo se puede probar fácilmente de forma automática debido a problemas técnicos. Si el software es complejo (Karhu, Repo, Taipale, & Smolander, 2009), tiene dependencias con productos de terceros (Graham & Fewster, 1999) o, en general, no se implementa con capacidad de prueba en mente, las pruebas automatizadas pueden ser difíciles de lograr. Por este motivo, la automatización

de pruebas no es algo que solo afecte el trabajo de los evaluadores; los desarrolladores también deben crear software comprobable.

2.3.2.6 INTEGRACIÓN CONTINUA

El concepto de integración continua fue originalmente presentado por Beck (2000) en su libro "*Extreme programming*" explicado como parte de las prácticas primarias de la programación extrema. La integración continua es hoy ampliamente utilizada, pero la forma en que se la aplica en los proyectos varía mucho (Ståhl & Bosch, 2014). La integración continua se considera como uno de los elementos clave del desarrollo ágil (Stolberg, 2009) y uno de los componentes básicos de la entrega continua (Humble & Farley, 2010).

La integración continua es una práctica de desarrollo de software en la cual los desarrolladores integran sus cambios de código frecuentemente con otros. Esto se logra al realizar cambios en la misma línea principal de un sistema de control de versiones al menos diariamente (Fowler, 2006) (Ståhl & Bosch, 2014). Un criterio de integración continua es que cada vez que se depositan cambios en el sistema de control de versiones, la aplicación se genera automáticamente y se prueba (Humble & Farley, 2010).

En conclusión, se puede decir que el objetivo de la integración continua es mantener el software cohesivo a lo largo de su fase de desarrollo para que la integración no sea un proceso lento y propenso a errores al final del proyecto; más bien, debería hacerse continuamente, desprendiéndose de ahí el nombre de integración continua. Si los desarrolladores no integran su código de manera frecuente, será difícil predecir cuánto tiempo llevará realmente la integración (Fowler, 2006). Además, si las diferentes partes del software no se han integrado, no se puede asegurar si el software realmente va a funcionar (Humble & Farley, 2010).

Un beneficio de la integración continua es que los desarrolladores obtienen una respuesta rápida en caso de que los cambios publicados en el repositorio de control de versiones hayan introducido errores. Si el proceso de construcción de software falla, los desarrolladores tienen que enfocarse en solucionar los problemas detectados para mantener el software en un estado consistente (Humble & Farley, 2010), de allí que una

adecuada implementación de las pruebas automatizadas se convierte en un factor determinante para una integración continua exitosa.

En la Figura 2.8 se muestra la implementación técnica de la integración continua, consta de dos componentes: flujos de trabajo automatizados y un sistema de retroalimentación. Los flujos de trabajo automatizados permiten la ejecución de acciones definidas por el usuario, que generalmente obtienen los cambios de código más recientes del sistema de control de versiones, construyen el software y ejecutan sus pruebas automatizadas, luego se notifica a los desarrolladores mediante el sistema de retroalimentación si la construcción fue exitosa o no (Duvall, Matyas, & Glover, 2007).

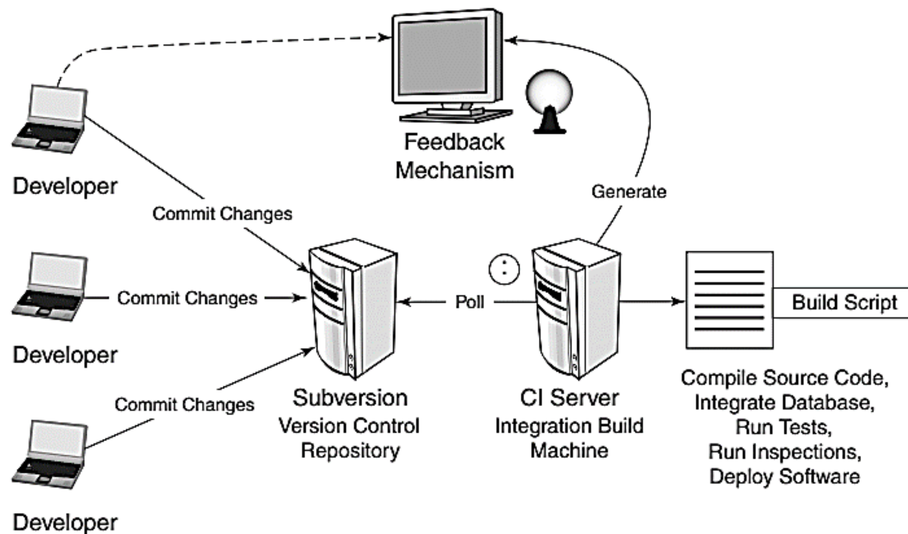


Figura 2.8- Componentes de un sistema de integración continua (Duvall, Matyas, & Glover, 2007)

2.3.3 PIPELINE DE DESPLIEGUE

Un *pipeline* de despliegue, también llamado pipeline de entrega continua (Chen, 2015), es un complemento lógico de la integración continua y un aspecto central de la entrega continua. La diferencia entre la integración continua y la entrega continua es que mientras la integración continua incluye integrar, construir y probar el código dentro de un entorno de desarrollo, la entrega continua se centra en garantizar que el software esté constantemente en un estado de entrega consistente (Fowler, 2013). Esto significa que la

calidad del software debe asegurarse también en un entorno similar al de producción. En la entrega continua el proceso de implementación es automático y solo debe activarse manualmente, mientras que la integración continua no tiene en cuenta el proceso de implementación.

El *pipeline* de despliegue describe las diferentes etapas del proceso de entrega de un proyecto, cada etapa aumenta la confianza en la construcción ejecutando más pruebas de alto nivel en un entorno similar al de producción (Fowler, 2013). La idea es proporcionar retroalimentación rápida a los desarrolladores en etapas tempranas y ejecutar pruebas más intensivas en cuanto a tiempo y recursos, solo si la construcción del software ha superado con éxito las etapas anteriores (Chen, 2015). Esta compensación entre confianza y costo-tiempo se visualiza en la Figura 2.9. El proceso de construcción pasa a la siguiente etapa solo si supera satisfactoriamente la etapa anterior.

De acuerdo con Humble y Farley (2010), no es posible estandarizar los pipelines de despliegue y sus etapas de una manera que sea aplicable a todos los proyectos. Argumentan que, aunque pueden existir etapas de prueba adicionales, todos los proyectos que implementan la entrega continua tienen las siguientes etapas en común:

2.3.3.1 Etapa de commit

En esta etapa el software se construye y se realiza una verificación técnica de calidad del software, proporcionando retroalimentación inicial a los desarrolladores sobre los cambios que se han realizado sobre el código (Chen, 2015).

La verificación técnica en este contexto significa que se realiza una recopilación de pruebas automatizadas de nivel inferior (como pruebas unitarias) y análisis estático de código (Zampetti, Scalabrino, Oliveto, Canfora, & Di Penta, 2017).

2.3.3.2 Etapa de pruebas de aceptación automatizadas

En esta etapa el software se prueba automáticamente según sus requisitos funcionales y no funcionales (Chen, 2015) (Humble & Farley, 2010). A menudo, el entorno de producción es significativamente diferente de los entornos de desarrollo y prueba (Humble & Farley, 2010); por lo tanto, las pruebas de aceptación deben ejecutarse en un

entorno similar al ambiente de producción, que se instala y configura automáticamente (Chen, 2015).

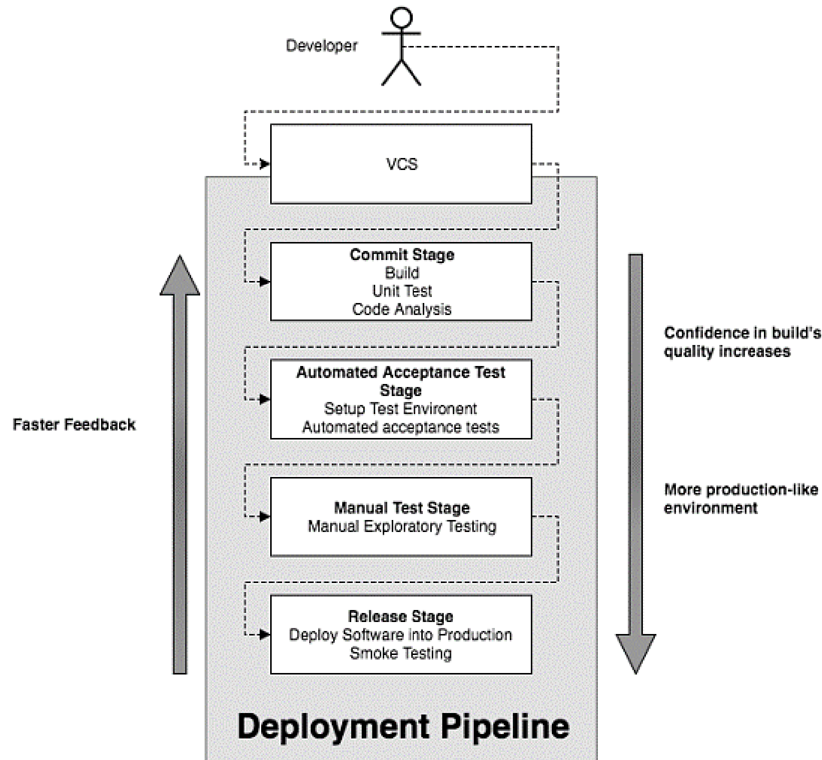


Figura 2.9 - Ejemplo de un pipeline de despliegue (Chen, 2015)

2.3.3.3 ETAPA DE PRUEBAS MANUALES

El software se prueba manualmente para detectar errores que no han podido ser detectados durante la etapa de prueba automática (Chen, 2015).

2.3.3.4 ETAPA DE LIBERACIÓN

En esta etapa el software se lanza a los usuarios finales o se implementa en un entorno de pre-producción. En la entrega continua, la decisión de implementar es "un proceso manual", es decir, la implementación debe activarse manualmente, pero la implementación en sí misma debe realizarse automáticamente (Fowler, 2013). En el despliegue continuo, esta etapa va un paso más allá: todos los cambios se implementan sin una aprobación manual (Humble, 2010).

Chen (2015) afirma que la etapa de prueba manual no siempre es necesaria, lo que también es respaldado por (Hayes, 2004). La automatización de pruebas juega un papel crucial en el *pipeline* de despliegue para asegurar la calidad del producto en cada construcción, permitiendo una respuesta rápida en caso de que se rompa la funcionalidad existente.

2.4 MICROSERVICIOS

En la literatura revisada se identificaron distintas definiciones del término "microservicio"; sin embargo, Di Francesco et al. (2017) concluyen que la mayoría de los artículos académicos tienden a inclinarse hacia la definición de microservicio proporcionada por Fowler y Lewis (2014): "el estilo arquitectónico de microservicios es un enfoque para desarrollar una aplicación como un conjunto de pequeños servicios, cada uno ejecutando su propio proceso y comunicándose mediante mecanismos livianos, a menudo una API sobre el protocolo HTTP [22]".

Jaramillo (2016) desde con un enfoque minimalista establece que los microservicios son "pequeños servicios autónomos que trabajan en conjunto para cumplir un requisito empresarial". O'Connor et al. (2017) amplian este concepto describiendo características no funcionales de los microservicios indicando que deben tener la capacidad de ser desplegados, escalados y probados de manera independiente.

Jaramillo (2016) presenta una clasificación de conceptos y características básicas del estilo arquitectónico basado en microservicios que a continuación se complementan con definiciones de varios autores.

2.4.1. CARACTERÍSTICAS

a) Pequeños y específicos

El término "pequeño" se utiliza en el mundo de los microservicios para enfatizar una de sus características más relevantes: cada servicio es refinado y de alta cohesión para enfocarse en cumplir con una responsabilidad específica (Richter, Konrad, Utecht, & Polze, 2017), representa una parte de la funcionalidad del sistema.

[22] https://tools.ietf.org/html/

El tamaño de un microservicio depende del dominio que cubre. Desde la perspectiva del desarrollo, cada servicio debe tratarse como una aplicación independiente con su propio repositorio de código fuente y *pipeline* de entrega continua (Yu, Silveira, & Sundaram, 2016) (Jaramillo, 2016).

b) Bajo acoplamiento

El bajo acoplamiento es una característica esencial de los microservicios. Cada microservicio debe ser desplegado según sea necesario sin necesidad de coordinar con otros servicios (Yu, Silveira, & Sundaram, 2016). Si se tienen dos servicios y siempre se están liberando conjuntamente, podría ser una señal de que deben ser uno solo y que hay trabajo por hacer en la descomposición de los servicios actuales. El bajo acoplamiento permite despliegues rápidos y más frecuentes mejorando la capacidad de respuesta de la aplicación frente a los requerimientos de los usuarios (Jaramillo, 2016).

c) Lenguaje neutral

Los microservicios deben construirse utilizando la tecnología con la que los desarrolladores se sienten más cómodos. Los equipos de desarrollo no deben ser dictados por ningún lenguaje de programación que se esté utilizando, lo que significa que la arquitectura de microservicios aprovecha la libertad al usar tecnologías que tienen más sentido para la tarea y las personas que la realizan (Richter, Konrad, Utecht, & Polze, 2017). Esto hace que sea más fácil aprovechar al máximo las tecnologías y habilidades óptimas que tienen los equipos (Jaramillo, 2016).

Debido a que los microservicios son neutrales en cuanto al lenguaje de programación, la comunicación entre ellos también se realiza a través de una interfaz de programación de aplicaciones (API) independiente de la plataforma, generalmente basada en el protocolo HTTP, como los servicios REST (Villamizar et al. ,2015). La Figura 2.10 muestra un ejemplo describiendo la variedad de idiomas y las tecnologías que se podrían usar para construir un sistema típico de compras en línea con un enfoque de microservicios, desarrollado utilizando diferentes lenguajes de programación y *frameworks* como Java, PHP[23] o

[23] https://php.net

Node.js[24]. Cada servicio incluso tiene su propio tipo de almacenamiento de datos en el que el servicio de catálogo utiliza Cloudant[25], el servicio de pedido usa una base de datos SQL, etc.

d) Contexto delimitado

Un contexto delimitado encapsula detalles de un solo dominio, como modelo de datos, modelo de dominio, etc (Fowler, 2014), también define los puntos de integración con otros servicios. En la arquitectura de microservicios, es fundamental tener un contexto consolidado bien definido (Richter, Konrad, Utecht, & Polze, 2017), esto significa que mientras más límites estén explícitamente definidos entre los dominios, más se puede razonar sobre el diseño y dimensionamiento de los servicios de manera eficiente.

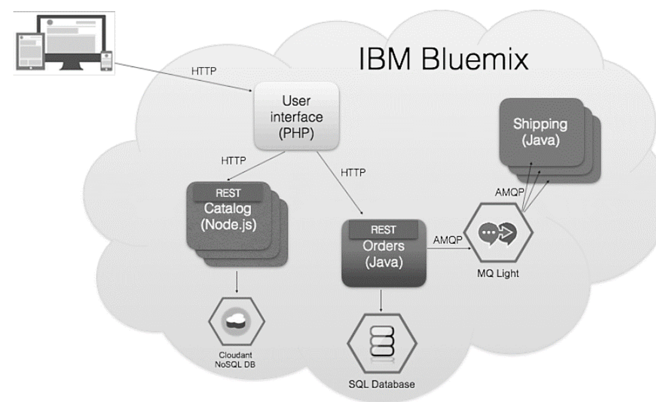


Figura 2.10- Implementación basada en microservicios independiente del lenguaje (Jaramillo, 2016)

2.4.2. DESAFÍOS EN LA CONSTRUCCIÓN DE UNA ARQUITECTURA BASADA EN MICROSERVICIOS

El estilo arquitectónico basado en microservicios provee una larga lista de ventajas; sin embargo, existen varios desafíos que deben abordarse antes de poder producir beneficios:

a) Aislamiento de fallos

Durante el ciclo de vida de un producto de software los fallos en algún momento se presentarán, es solo una cuestión de tiempo y cuando suceden en servicios

[24] https://nodejs.org/es/
[25] https://cloudant.com/

importantes, es mejor que fallen rápidamente. Las fallas rápidas conducen a una mejor comprensión y resolución de problemas. Es necesario poder dividir las cosas en pedazos pequeños para facilitar que las pruebas automatizadas puedan ayudar a encontrar errores de una manera temprana a través de las herramientas establecidas para la entrega continua (Jaramillo, 2016). Cambios pequeños permiten que el desarrollador pueda cambiar una cosa a la vez y si se rompe, se sabe exactamente qué es lo que se rompió (Balalaie & Heydarnoori, 2016).

b) Observabilidad

La construcción de una arquitectura basada en microservicios necesita una forma de visualizar el estado de salud de todos los servicios del sistema para localizar y responder rápidamente ante cualquier problema (Richter, Konrad, Utecht, & Polze, 2017), esto también incluye un mecanismo integral de seguimiento para registrar y almacenar los eventos que permitan analizar las situaciones que se presentan durante la operación de los servicios (Yu, Silveira, & Sundaram, 2016).

c) Requerimientos de Automatización

La automatización debe convertirse en una cultura dentro de la organización, siendo su institucionalización uno de los desafíos más importantes (Jaramillo, 2016). Aplicaciones de mediano tamaño podrían contener una cantidad importante de servicios, y no se podrán manejar sin una forma de automatizar las tareas (Duvall, Matyas, & Glover, 2007).

d) Alta independencia

Uno de los principios más relevantes de un sistema basado en microservicios es que los microservicios se encuentren altamente desacoplados (Salah, Zemerly, Yeun, Al-Qutayri, & Al-Hammadi, 2016), esto significa que es fundamental mantener la independencia entre los servicios para que cada uno de ellos pueda desarrollarse e implementarse de manera independiente sin afectar el uno al otro.

e) Pruebas

En un esquema en el que es necesario una mayor especialización de los servicios para fomentar un bajo acoplamiento, la proliferación de los componentes puede aumentar las posibilidades de que ocurran fallos (Jaramillo, 2016). En consecuencia, no es fácil asegurarse que las pruebas sean lo suficientemente exhaustivas como para cubrir todos los aspectos. Las pruebas automáticas han avanzado significativamente a medida que se siguen introduciendo nuevas

herramientas y técnicas (Berner, Weber, & Keller, 2005). Sin embargo, persisten desafíos en cuanto a la manera de probar eficaz y eficientemente los aspectos funcionales y no funcionales del sistema, especialmente en un modelo distribuido. Con más componentes independientes y patrones de colaboración entre ellos, la arquitectura de microservicios agrega un nuevo nivel de complejidad a las pruebas (Daya, Van Duy , Eati, & Ferreira, 2015).

f) **Escalabilidad**

Una de las motivaciones importantes de la implementación de una arquitectura basada en microservicios es resolver el problema de la escalabilidad; sin embargo, la escalabilidad como tal, es un desafío al que se debe prestar especial atención (Jaramillo, 2016). En algún momento habrá una explosión en el número de microservicios que se crearán en el sistema, sin mencionar las versiones que podrían existir de cada uno de ellos, el número de conexiones entre ellos crecerá considerablemente, aumentando la complejidad de un sistema basado en microservicios. Se requerirán soluciones especiales para el descubrimiento de servicios para saber qué servicio se está ejecutando, un mecanismo de enrutamiento para enrutar el tráfico a través de las API de los servicios, una mejor administración de la configuración para realizar la configuración de forma dinámica y aplicar cambios al sistema, y así sucesivamente (Do, Do, Tran, Farkas, & Rotter, 2017).

2.5 VIRTUALIZACIÓN COMO PLATAFORMA

La virtualización de recursos es uno de los conceptos clave de la computación en la nube y se refiere a la creación de una versión virtual (en lugar de la versión real) de algo, incluyendo pero no limitado a una plataforma de hardware, un sistema operativo, un dispositivo de almacenamiento o recursos de una red informática. La virtualización hace uso de una capa intermedia de software sobre un sistema para proporcionar abstracción de múltiples recursos virtuales. Actualmente se pueden considerar dos grandes grupos de modelos de virtualización: las máquinas virtuales y los contenedores.

2.5.1. MÁQUINAS VIRTUALES

Los recursos virtuales son componentes de software conocidos como Máquinas Virtuales (VM), que podrían ser descritos como contextos de ejecución aislados. Entre

las técnicas de virtualización que existen en el mercado, una de las más populares es la virtualización basada en *hypervisor*, que requiere de un *Virtual Machine Monitor* (VMM) ejecutándose sobre un sistema operativo anfitrión para proporcionar una abstracción completa que permite ejecutar múltiples sistemas operativos en un solo anfitrión real (Singh & Singh, 2016). Ejemplos de virtualización mediante *hypervisor* incluyen a herramientas como Xen[26], KVM[27], Virtual Box[28] y WM-Ware[29].

2.5.2. CONTENEDORES

La técnica de virtualización basada en contenedores (o virtualización de sistema operativo) permite la ejecución de varias máquinas virtuales diferentes sobre el kernel del sistema operativo anfitrión (Celesti, Mulfari, Fazio, Villari, & Puliafito, 2016).

La Figura 2.11 muestra la diferencia clave entre las tecnologías de virtualización. Mientras que la virtualización basada en *hypervisor* proporciona abstracción para sistemas operativos invitados completos (uno por VM), la virtualización basada en contenedores funciona a nivel del sistema operativo proporcionando abstracciones directamente para los procesos de invitado (Li, Kihl, Lu, & Andersson, 2017). En esencia, las soluciones de *hypervisor* funcionan como abstracción a nivel de hardware mientras que la virtualización basada en contenedores opera a nivel de llamada del sistema.

Los contenedores comparten un solo *kernel* del sistema operativo; entonces, la virtualización basada en contenedores se supone que tiene un aislamiento más débil comparado con la virtualización basada en *hypervisor* (Celesti, Mulfari, Fazio, Villari, & Puliafito, 2016). Desde el punto de vista de los usuarios, cada contenedor se ve y ejecuta exactamente como un sistema operativo independiente. Como consecuencia, se pueden desplegar un mayor número de contenedores que máquinas virtuales en un mismo anfitrión físico.

[26] https://www.xenproject.org
[27] https://www.linux-kvm.org
[28] https://www.virtualbox.org
[29] https://www.vmware.com

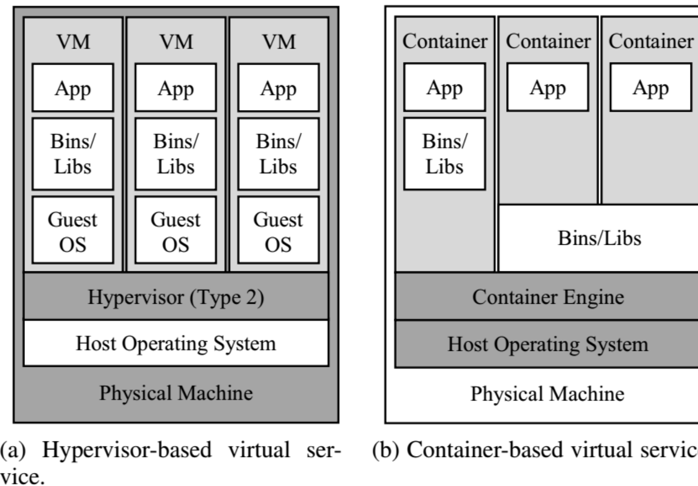


Figura 2.11 - Virtualización: Hypervisor vs Contenedores (Li, Kihl, Lu, & Andersson, 2017)

En términos de escalabilidad y rendimiento los contenedores son mucho mejores que las máquinas virtuales, ya que ejecutan varios sistemas operativos de forma concurrente acarreando una importante sobrecarga sobre el uso y disponibilidad de los recursos (Li, Kihl, Lu, & Andersson, 2017). Entre sus principales beneficios están:

a) **Bajo consumo de recursos**

Los contenedores comparten recursos con el sistema operativo anfitrión haciéndolos más eficientes, por tanto, las acciones de arranque y parada en un contenedor toman segundos mientras que en una máquina virtual las mismas operaciones podrían tardar varios minutos. Además, las aplicaciones que corren en un contenedor tienen menor sobrecarga comparado con aquellas que corren nativamente en el sistema operativo anfitrión (Singh & Singh, 2016) (Paraiso, Challita, Al-Dhuraibi, & Merle, 2016).

b) **Portabilidad**

La portabilidad de los contenedores tiene el potencial de eliminar todo tipo de errores causados por los cambios en los ambientes de ejecución y la dependencia de los proveedores (Singh & Singh, 2016) (Paraiso, Challita, Al-Dhuraibi, & Merle, 2016).

c) **Ligero**

Los contenedores por naturaleza son livianos permitiendo a los desarrolladores ejecutar docenas de contenedores al mismo tiempo, haciendo posible emular

entornos listos para producción. Los ingenieros de operación pueden correr más contenedores en un solo anfitrión que usando máquinas virtuales (Singh & Singh, 2016) (Paraiso, Challita, Al-Dhuraibi, & Merle, 2016).

2.6 DOCKER: INFRAESTRUCTURA BASADA EN CONTENEDORES

Docker es una herramienta que hace que las tecnologías *Linux Containers* (LXC) y *Kernel Based Virtual Machine (KVM)* sean más fáciles de usar. Docker permite desarrollar, desplegar y ejecutar aplicaciones empaquetadas en contenedores (Singh & Singh, 2016), siendo una alternativa de menor costo comparado con máquinas virtuales basadas en *hypervisor*. Los elementos esenciales de la tecnología son las imágenes de Docker y los contenedores.

2.6.1. IMAGEN DOCKER

Una imagen Docker es un binario que incluye todos los elementos requeridos para correr un contenedor Docker. Se puede concebir como un conjunto de capas superpuestas de un sistema de archivos o una instantánea del sistema de archivos en un momento específico en el tiempo que es inmutable y no tiene estado. La Figura 2.12 muestra un contenedor Docker que está proveyendo un entorno de desarrollo web básico, basado en imágenes apiladas de un sistema operativo Ubuntu[30], imagen Emacs [31](editor de texto popular entre desarrolladores) e imagen de un servidor web Apache[32]. Se identifica mediante un nombre y una etiqueta que expresa la versión concreta de la imagen.

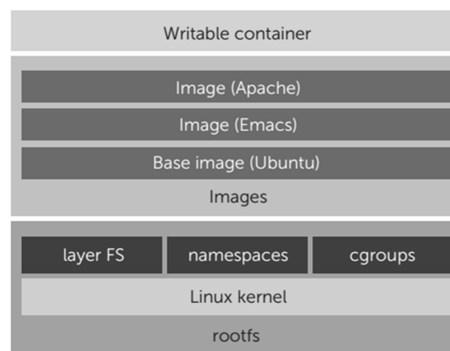


Figura 2.12 - Arquitectura de una imagen Docker (Pahl, 2015)

[30] https://www.ubuntu.com
[31] https://www.gnu.org/s/emacs
[32] https://httpd.apache.org

2.6.2. CONTENEDOR

Un contenedor es una instancia de una imagen en tiempo de ejecución; por tanto, tiene estado. Un contenedor una vez creado (instanciado) a partir de una imagen se puede encender o apagar a voluntad. Un contenedor consta de tres componentes: i) una imagen de Docker, ii) un entorno de ejecución, y, iii) un conjunto de instrucciones.

2.6.3. ARQUITECTURA

La Figura 2.13 muestra la arquitectura de Docker, cuyos componentes se detallan a continuación:

2.6.3.1. DOCKER HOST

La máquina física o virtual en la que se despliegan el servicio de Docker y los contenedores se la denomina "Docker Host". El servicio de Docker es responsable de crear, correr y monitorear contenedores, así como la construcción y el almacenamiento de imágenes. La ejecución del servicio de Docker es normalmente manejado por el sistema operativo del sistema anfitrión (Richter, Konrad, Utecht, & Polze, 2017).

2.6.3.2. DOCKER CLIENT

El cliente se comunica con el servicio de Docker mediante *sockets* a través de una API con interfaz RESTful. El objetivo de este componente es recibir comandos de los usuarios para controlar el "host", crear imágenes, publicar, ejecutar y administrar contenedores que corresponden a las instancias de sus imágenes (Kumar & Kurhekar, 2016). El cliente mantiene una comunicación bidireccional a través de HTTP facilitando las conexiones remotas al servicio de Docker. "Docker client" y "Docker host" conforman lo que se conoce como "Docker engine" (Jaramillo, 2016).

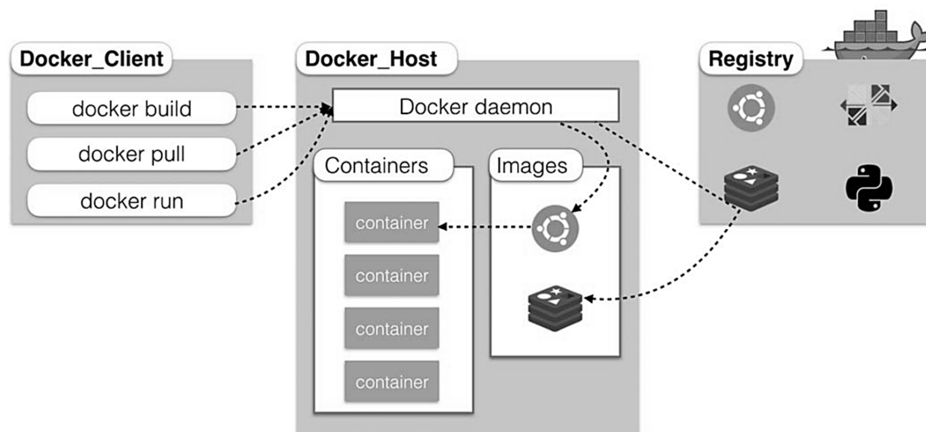


Figura 2.13 - Arquitectura de Docker (Paraiso, Challita, Al-Dhuraibi, & Merle, 2016)

2.6.3.3. DOCKER REGISTRY

El componente “Docker registry” es el repositorio que tiene la responsabilidad de almacenar y distribuir imágenes. “Docker Hub” es el repositorio por omisión que aloja miles de imágenes públicas. Una imagen de Docker puede contener solo componentes básicos de un sistema operativo o instalaciones completas de aplicaciones y *stacks* de tecnologías pre configuradas listas para ser utilizadas (Jaramillo, 2016). Para crear una imagen la opción más conveniente es escribir un archivo de secuencia de comandos (script) denominado *Dockerfile* que describe declarativamente las características del contenedor. Muchas organizaciones mantienen su propio repositorio que puede ser usado para almacenar imágenes privadas.

3 TRABAJOS RELACIONADOS

En el Capítulo 2 se describe el contexto y la arquitectura tecnológica seleccionada por la Universidad de Cuenca para futuros proyectos de desarrollo de software. En la arquitectura se evidencia un alineamiento por el uso de tecnologías que han venido teniendo éxito en la adopción de la computación en la nube.

En este capítulo se aborda la investigación realizada por diferentes autores sobre la implementación de un *pipeline* de entrega continua de software enfatizando el análisis de los trabajos relacionados con arquitectura de aplicaciones basadas en un estilo orientado a microservicios, tecnologías habilitantes para el aprovisionamiento y despliegue de aplicaciones virtualizadas sobre contenedores y el tratamiento de la infraestructura como código para la gestión de recursos en entornos de nube.

3.1 PROCESO DE ENTREGA CONTINUA DE SOFTWARE

En la investigación realizada por (Shahin, Babar, & Zhu, 2017) en la que se aborda el estado del arte de la entrega continua de software, los autores realizan una revisión sistemática de literatura en la que luego de aplicar los criterios de inclusión y exclusión, de una base inicial de 449 artículos se seleccionan 69 para el análisis. Durante la revisión se determinó la existencia de 5 investigaciones previas entre los años 2012 y 2015 que realizan un estudio del estado del arte en áreas de conocimiento relacionadas a la integración continua, liberación rápida y entrega continua de software, incluyendo 46, 43, 24, 50 y 30 artículos respectivamente.

Según los autores, existen elementos de índole procedimental, organizacional y operacional que facilitan la implementación de prácticas de entrega continua de la siguiente manera:

1) Reduciendo el tiempo de construcción y pruebas en la entrega continua durante la etapa de integración continua.
2) Aumentando la visibilidad y conciencia de los resultados de la etapa de integración continua.
3) Dando soporte (semi) automatizado de pruebas continuas.
4) Detectando violaciones, defectos y fallas durante la integración continua.
5) Abordando los problemas de seguridad y escalabilidad en el proceso de implementación.

6) Mejorando la confiabilidad y fiabilidad del proceso de implementación.

En función de los hallazgos, Shahin et al. (2017) realizan un compendio que resume los desafíos a enfrentar (Tabla 3.1), prácticas (Tabla 3.2), herramientas y factores de éxito (Tabla 3.3). En la Tabla 3.1 se mapean en los puntos clave las referencias a artículos académicos recopilados para la elaboración de este proyecto de tesis que comprueban que los criterios de los autores se mantienen vigentes. Adicionalmente, la Figura 3.1 muestra cómo las prácticas se relacionan con los desafíos para una implementación exitosa de un proceso de entrega continua de software.

Tabla 3.1 - Desafíos de la entrega continua de software (Shahin, Babar, & Zhu, 2017)

		Desafíos	Puntos Clave y artículos incluidos
Desafíos comunes para la adopción de integración y entrega continua de software	Conocimiento del equipo y comunicación	Falta de conocimiento y transparencia	- Falta de conocimiento y transparencia del proceso de entrega continua (Pulkkinen, 2016) - Falta de entendimiento acerca del estado del proyecto incrementa el número de conflictos al unir el código de la aplicación (Pulkkinen, 2016)
		Coordinación y colaboración	- Para practicar integración continua y/o entrega continua se requiere más que coordinación efectiva y comunicación entre los miembros del equipo (Chen, 2015).
	Falta de inversión	Costo	- Inversión o actualización de infraestructura y recursos. - Entrenamiento y dirección.
		Falta de experiencia y habilidades	- Integración continua y entrega continua demandan nuevas habilidades técnicas (Pulkkinen, 2016). - Se requiere programadores altamente calificados.
		Mayor presión y carga de trabajo para los miembros del equipo	- Mayor estrés para desarrolladores y equipo de operaciones (Pulkkinen, 2016) (Vassallo, y otros, 2016). - Mayor responsabilidad de los desarrolladores (Gmeiner, Ramler, & Haslinger, 2015).
		Carencia de tecnologías y herramientas adecuadas	- Carencia de herramientas maduras para la automatización de las pruebas y revisión de código en la integración continua (Mårtensson, Ståhl, & Bosch, 2017) - Cambios frecuentes en las herramientas (Chen, 2015). - Problemas de seguridad y confianza en herramientas de construcción y despliegue (Mårtensson, Ståhl, & Bosch, 2017). - Las herramientas no se ajustan para todas las organizaciones.
	Resistencia al cambio	Resistencia general al cambio	- Cambiar hábitos de los miembros del equipo. - Proceso que consume mucho tiempo cambiar la mentalidad del equipo (Chen, 2015).
		Escepticismo y desconfianza en prácticas continuas	- Falta de confianza en los beneficios de la integración y entrega continua (Pulkkinen, 2016).

		Desafíos	Puntos Clave y artículos incluidos
	Procesos organizacionales, estructura y políticas	Dificultad para cambiar políticas y cultura organizacional	- Carencia de un modelo de negocio ágil y adecuado (Pulkkinen, 2016) - Cambio en la gestión de largo plazo por corto plazo
		Organización distribuida	- Modelo de equipo distribuido (Pulkkinen, 2016) - Percepciones incoherentes entre los miembros del equipo
Desafíos adopción integración continua	Pruebas	Carencia de una estrategia de pruebas apropiada	- Falta de automatización completa de las pruebas (Amrit & Meijberg, 2017) - Falta de desarrollo orientado a las pruebas (Vassallo, y otros, 2016) (Ambler, 2007) (Amrit & Meijberg, 2017) (Gmeiner, Ramler, & Haslinger, 2015)
		Pobre calidad de las pruebas	- Pruebas inestables (Akerele, Ramachandran, & Dixon, 2013) - Baja cobertura de pruebas (Pulkkinen, 2016) - Baja calidad de datos de prueba (Pulkkinen, 2016) - Pruebas de larga duración (Pulkkinen, 2016) - Dependencias en las pruebas
	Mezcla de conflictos	Conflictos en integración de código	- Componentes de terceros - Incompatibilidad entre los componentes dependientes - Falta de entendimiento de los cambios en componentes (Pulkkinen, 2016)
Desafíos en adopción de entrega continua	Carencia arquitectura adecuada	Dependencias en diseño y código	- Arquitecturas altamente acopladas (Pulkkinen, 2016) (Sultanía, 2015) - Dificultad para encontrar requerimientos autónomos para integraciones frecuentes
		Cambios en esquema de bases de datos	- Cambios frecuentes en el esquema de base de datos (Pulkkinen, 2016) (Ambler, 2007)
	Dependencias en el equipo	Dependencias en el equipo	- Dependencia cruzada de equipos - Efectos colaterales de cambios en múltiples equipos

Tabla 3.2 - Prácticas para entrega continua de software (Shahin, Babar, & Zhu, 2017)

		Prácticas	Puntos Clave y artículos incluidos
Prácticas comunes de implementación de integración y entrega continua	Conocimiento y comunicación del equipo	Mejorar el conocimiento y comunicación del equipo de trabajo	- Listar las características cambiadas en el registro de cambios. - Etiquetar las últimas versiones y características. - Informar a los miembros del equipo acerca de las ramificaciones del código que se encuentran obsoletas. - Mejorar y compartir el conocimiento entre el personal técnico y de gestión en distintos niveles.
	Inversión	Planificación y documentación	- Definir un camino planificado para la adopción de prácticas de integración y entrega continua (Mårtensson, Ståhl, & Bosch, 2017). - Documentar la integración, pruebas y otras actividades en el proceso de integración. - Definir el modelo de flujo de integración
		Incentivar la mentalidad del equipo	- Organizar eventos sobre prácticas continuas para difundir la mentalidad y formar a los miembros del equipo. - Otorgar libertad a los desarrolladores. - Empoderar la cultura (Mårtensson, Ståhl, & Bosch, 2017)

		Prácticas	Puntos Clave y artículos incluidos
		Mejorar la experiencia y habilidades del equipo	- Entrenamiento formal y dirección a los miembros del equipo.
	Clarificar estructuras de trabajo	Definir nuevos roles y equipos	- Establecer y desarrollar un equipo dedicado a mantener el pipeline de despliegue. - Pilotear el equipo
		Adoptar nuevas reglas y políticas	- Todos los desarrolladores deben estar alerta cuando se libera el software
Prácticas para integración continua	Mejorar actividades de prueba	Mejorar las actividades de prueba	- Practicar desarrollo orientado a pruebas (Vassallo, y otros, 2016) - Practicar planificación de pruebas (Pulkkinen, 2016) - Practicar pruebas cruzadas en el equipo de desarrollo - Diseñar pruebas desacopladas para separar pruebas unitarias de pruebas funcionales y de pruebas de aceptación
	Estrategias de ramificación	Establecer estrategias para ramificaciones de código	- Usar integración local o repositorio local - Usar ramificaciones de código de tiempo de vida corto (Pulkkinen, 2016) - Practicar el uso del repositorio - No manejar muchas ramificaciones de código
	Descomponer el desarrollo en unidades más pequeñas	Descomponer el desarrollo en unidades más pequeñas	- Eliminar código no utilizado - Descomponer funcionalidades y cambios grandes en más pequeños y seguros (Mårtensson, Ståhl, & Bosch, 2017) (Pulkkinen, 2016) - Formar equipos de trabajo más pequeños e independientes.
Prácticas para entrega continua	Arquitectura flexible y modular	Establecer arquitectura flexible y modular	- Diseñar los sistemas manteniendo en mente las metas de despliegue de las aplicaciones (Mårtensson, Ståhl, & Bosch, 2017) (Bellomo, Ernst, Nord, & Kazman, 2014). - Definir claramente las interfaces de los componentes.
	Comprometer al personal	Comprometer al personal en el proceso de despliegue	- Desarrolladores y personal de pruebas deben tomar una mayor responsabilidad de su código (Mårtensson, Ståhl, & Bosch, 2017) (Gmeiner, Ramler, & Haslinger, 2015) - Contar con desarrolladores bajo llamada
Prácticas para despliegue continuo	Liberar por partes	Liberar por partes	- Liberación inicial - Esconder o deshabilitar funcionalidades nuevas o problemáticas a los usuarios - Liberar software a un grupo pequeño de usuarios - Retroceder a una versión estable (Pulkkinen, 2016)
	Involucrar al cliente	Involucrar al cliente	- Cliente con liderazgo - Cliente con capacidad de monitoreo - Involucrar al cliente en la fase de prueba - Reuniones de seguimiento.

En el estudio los investigadores determinan que no existe una relación uno a uno entre los desafíos identificados, las prácticas propuestas, enfoques y herramientas

asociadas. Inclusive en algunos casos, no pudieron identificar ninguna práctica ni enfoque que establezca una relación unaria o bidireccional entre ellas, por tanto decidieron definir un conjunto de factores críticos que podrían afectar a una exitosa entrega continua. La Tabla 3.3 muestra la lista de 7 factores críticos: "Pruebas" (27 artículos, 39.1%) es el factor más mencionado para el éxito de las prácticas continuas, seguido de "conocimiento y transparencia del equipo" (24 artículos, 34.7%), "principios de buen diseño" (21 artículos, 30.4%) y “cliente” (17 documentos, 24.6%).

Estos resultados indican que las pruebas juegan un papel preponderante en el establecimiento exitoso de prácticas continuas en una determinada organización.

Tabla 3.3 - Lista de factores críticos para el éxito de la entrega continua (Shahin, Babar, & Zhu, 2017)

ID	Factor	N° artículos	%
F1	Pruebas (esfuerzo y tiempo)	27	39.1
F2	Conocimiento y transparencia del equipo	24	34.7
F3	Principios de buen diseño	21	30.4
F4	Cliente	17	24.6
F5	Equipo altamente calificado y motivado	15	21.7
F6	Dominio de la aplicación	14	20.2
F7	Infraestructura apropiada	14	20.2

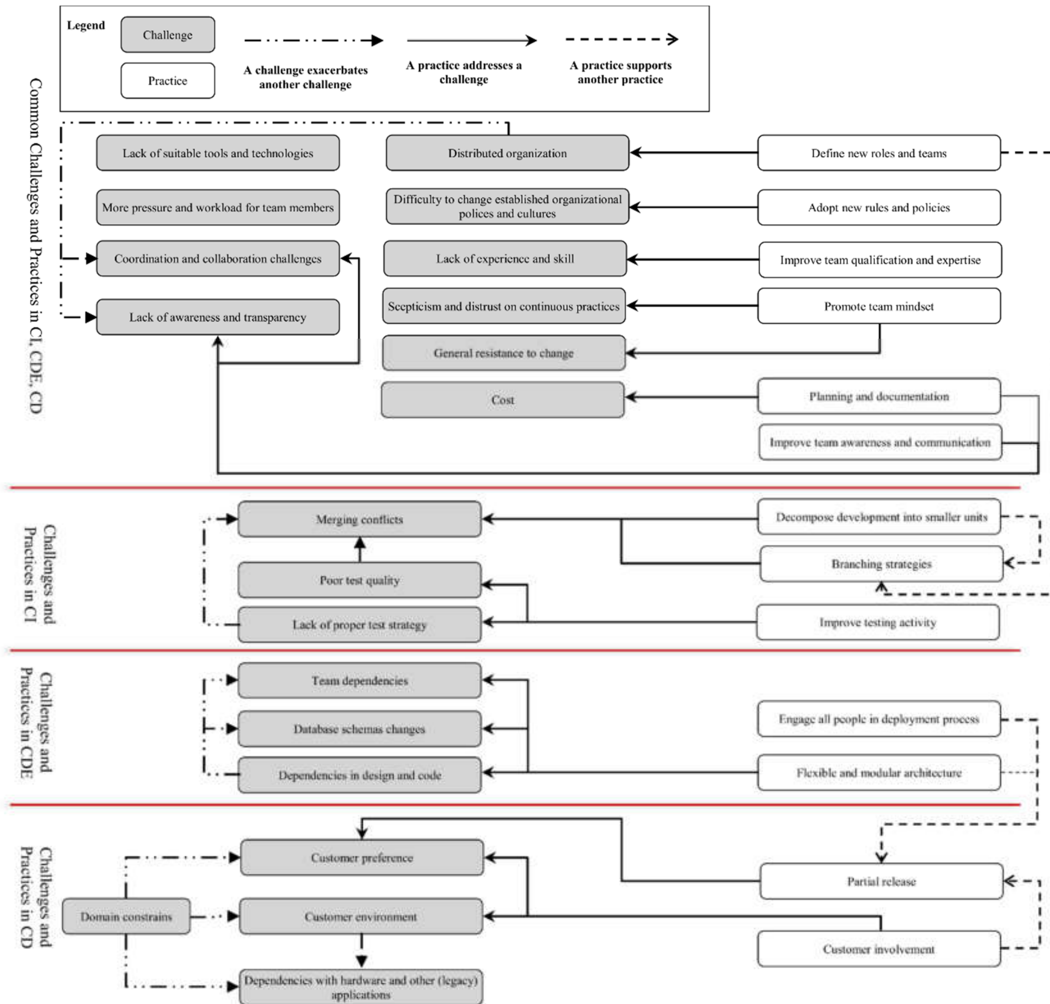


Figura 3.1 - Resumen de desafíos, prácticas y sus relaciones para adopción de CI y CD (Shahin, Babar, & Zhu, 2017)

3.2 PIPELINE DE ENTREGA CONTINUA DE SOFTWARE

En esta sección se analiza el trabajo realizado por varios autores relacionado con la entrega continua desde el punto de vista de las herramientas de software y como éstas están siendo utilizadas para la construcción de un *pipeline* de entrega continua de software.

En el análisis del estado del arte realizado por Shahin et al. (2017) la revisión sistemática de literatura reporta una cadena de herramientas para la implementación de pipelines de entrega continua de software. De acuerdo a los datos obtenidos, los autores determinan que en la actualidad no existe un estándar en cuanto a las etapas que deben

incluirse en un *pipeline* pues la determinación de éstas, depende de muchos factores: *stack* de tecnología, estrategia de automatización de pruebas, arquitectura de las aplicaciones, objetivos de despliegue, madurez de las prácticas de Ingeniería de Software de la organización, etc. Sin embargo, los autores identifican 7 etapas que se nombran en los artículos que se incluyeron en la investigación:

a) Control de versiones
b) Análisis y gestión de código
c) Construcción
d) Integración continua
e) Pruebas
f) Configuración y aprovisionamiento
g) Entrega continua

La Figura 3.2 muestra las etapas y el software que se utiliza en cada una de ellas.

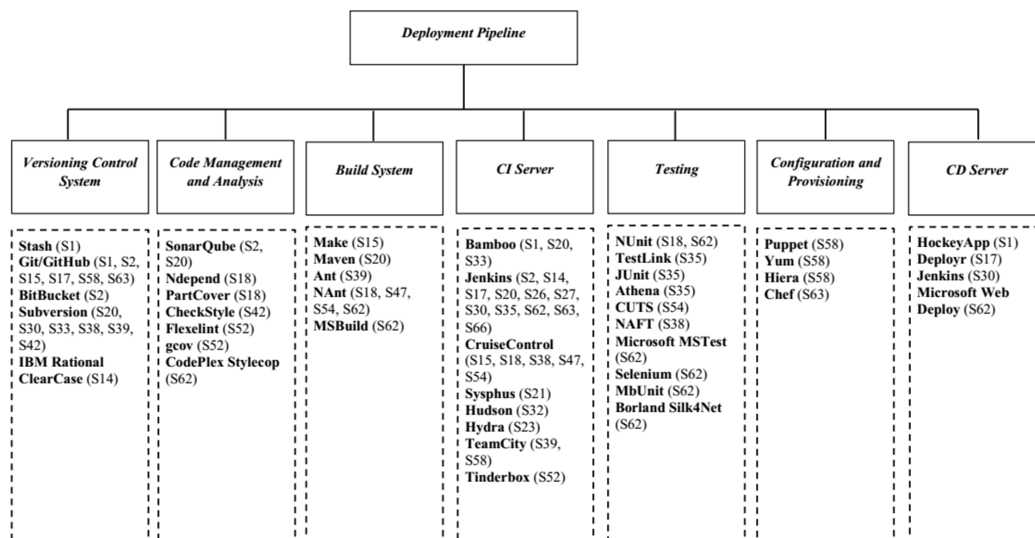


Figura 3.2 - Herramientas usadas para construir un pipeline de despliegue (Shahin, Babar, & Zhu, 2017)

En la primera etapa, los desarrolladores depositan código constantemente en el repositorio de código fuente (sistema de control de versiones). Las herramientas reportadas como las más utilizadas son *Git* [33] y *Subversion*[34]. En la segunda etapa, se

[33] https://git-scm.com
[34] https://subversion.apache.org

reporta el uso de herramientas de administración y análisis de código para: encontrar fallos en el software sin necesidad de ejecutarlo, recopilación de métricas tales como la cobertura de las pruebas y violaciones de estándares de codificación; el software más utilizado: *SonarQube*[35]. En la tercera etapa, el sistema de construcción de software accede al sistema de control de versiones para determinar si hay cambios; de ser necesario, se ejecuta la herramienta para compilación y empaquetamiento, siendo *Maven* [36] el producto más utilizado. En la cuarta etapa, el servidor de integración continua puede realizar tareas de construcción de software y/o ejecutar pruebas unitarias. La herramienta más utilizada: *Jenkins*[37]. En la quinta etapa, se ejecutan las pruebas de integración, pruebas de aceptación, pruebas de carga, con diferentes tecnologías, dependiendo del tipo. En la sexta se realiza la configuración y el aprovisionamiento de la infraestructura para el despliegue de las aplicaciones en distintos ambientes, no existe un indicador que identifique la preferencia por alguna de las tecnologías encontradas. Finalmente, la última etapa se encarga de realizar las operaciones de despliegue en los ambientes de producción. El software utilizado generalmente es el mismo servidor de integración continua, en este caso, *Jenkins*.

Gomede y Barros (2015) en el estudio que realizan sobre las prácticas y herramientas utilizadas para la implementación de la entrega continua, desarrollan una prueba de concepto en la que definen un grupo de herramientas que las clasifican por práctica dentro del *pipeline* de entrega de software: i) gestión de configuración (Git, Gerrit[38], Nexus[39], Flywaydb [40]y Vagrant[41]), ii) integración continua (Jenkins y Maven) , iii) aseguramiento de calidad (SonarQube, JMeter[42]), iv) gestión del ciclo de vida de la aplicación (Redmine[43]); y, v) gestión de infraestructura (Splunk [44]y OpenLDAP[45]). Los autores aclaran que para la selección se consideraron productos exclusivamente *open source*. La Tabla 3.4 muestra el stack de tecnologías, productos y su propósito dentro del *pipeline*.

[35] https://www.sonarqube.org
[36] https://maven.apache.org
[37] https://jenkins.io
[38] https://www.gerritcodereview.com
[39] www.sonatype.org/nexus
[40] https://flywaydb.org/
[41] https://www.vagrantup.com
[42] http://jmeter.apache.org
[43] www.redmine.org
[44] https://www.splunk.com
[45] https://www.openldap.org

Tabla 3.4 - Herramientas para la entrega continua con software opensource (Gomede & Barros, 2015)

Technology Stack	Description
Integrated Development Environment	**Eclipse:** Development environment used by developers to create an application
Source Code repositories	**SVN, Git, StarTeam:** Centralized location to keep updated source code based on authentication
Build Tools	**Ant, Maven:** Build automation tools to automate the process of compilation of source code
Continuous Integration Server	**Jenkins:** Verify integrated code with automated build and notify status of build execution to stakeholders
Automated / Testing Continuous Testing	**Junit:** Automated test cases: Unit test execution on integrated source code
Static Code Analysis Tool	**Sonar, FindBug:** Manage code quality: Comments, Coding rules, Potential bugs, Complexity, Unit Test, Duplications, Architecture & Design
Binary Repository	**Artifactory:** Versioning of Application Binaries into repository to manage, host and control the flow of binary artifacts
Configuration Management Provisioning	**VMware vSphere, VMware vCloud Director, Amazon Web Services, Microsoft Azure:** Infrastructure Provisioning in Cloud Environment
Configuration Management – Preparing Runtime Environment	**Chef:** Application Runtime Environment Creation by creating various cookbooks to configure runtime environment.
Deployment Tool / Script	**Deployment Plugins or Shell scripts:** To deploy final artifact to deployment server in different environment
Monitoring / Notifications	**Jenkins Plugins:** To monitor various environments as well as complete deployment pipeline process to get notification on any failures

Pulkkinen (2013) en su investigación establece un grupo de herramientas que las califica como “escenciales para entrega y despliegue de software”. La Tabla 3.5 muestra la clasificación que se realiza por tipo de herramienta incluyendo: i) sistemas de control de versiones, ii) servidores de integración continua, iii) software para gestión de configuración, iv) software para automatización de pruebas, v) software para gestión del cambio en bases de datos relacionales, y, vi) herramientas para gestión de dependencias y construcción automática de software.

Tabla 3.5 - Elementos esenciales para entrega continua y despliegue de software (Pulkkinen, 2013)

Type of tool or software	Explanation	Examples
Version Control System (VCS)	To manage and revision source code changes. Essential to establish an automated build pipeline.	Git, Mercurial, SVN, CVS
Continuous Integration Server	To automatically run integration tests when code changes are introduced via VCS. Orchestrate automated testing and deployment.	Jenkins, Hudson, Atlassian Bamboo, CruiseControl, Teamcity
Software Configuration Management	To automatically setup and configure environment and infrastructure for example testing and production purposes. Also provides support for scalability.	Puppet, Chef, Salt
Automated Test Suites	To run automated tests for unit, integration, acceptance testing phases	Junit, JMeter, Cucumber, Selenium, FitNesse
Database Change Management	To automatically apply database changes related to current build of software. Also for tracking changes and to apply rollbacks.	DbDeploy, Liquibase, FlyWay, ActiveRecord
Build Tool and Dependency Management System	To automatically build the software and manage dependencies to other libraries	Apache Ant+Ivy, Rake, Apache Maven, Gradle

3.3 ESTRATEGIA DE AUTOMATIZACIÓN DE PRUEBAS

Según Elberzhager et al. (2012), los proyectos de software dedican hasta un 50% de su presupuesto a las pruebas, por tanto, es necesario reducir costos y tiempos para su implementación. En este contexto, la automatización es ampliamente usada para reducir el esfuerzo en la aplicación de pruebas (Elberzhager, Rosbach, Münch, & Eschbach, 2012).

Según Shahin et al. (2017) la automatización de pruebas se constituye en el factor de éxito de mayor valoración para una implementación de un proceso de entrega continua, aseveración que es corroborada por (Amrit & Meijberg, 2017) (Humble, 2010) (Gmeiner, Ramler, & Haslinger, 2015).

Varios autores consideran además que la selección de una estrategia adecuada es fundamental para la automatización de las pruebas (Vassallo et al.,2016) (Gmeiner, Ramler, & Haslinger, 2015), pues alcanzar una cobertura del 100% es prácticamente imposible (Karhu, Repo, Taipale, & Smolander, 2009).

Durante el desarrollo de software las actividades de aseguramiento de calidad incluyen la práctica de: pruebas unitarias, pruebas de integración, pruebas de rendimiento y pruebas de aceptación, en cualquier combinación de ellas (Gmeiner, Ramler, & Haslinger, 2015).

Existe un criterio generalizado sobre los beneficios de la aplicación de pruebas unitarias a lo largo de toda la arquitectura del software, por lo que se sugiere de manera casi obligatoria su implementación (Amrit & Meijberg, 2017) (Elberzhager, Rosbach,

Münch, & Eschbach, 2012). Gmeiner et al. (2015) sugieren que en el largo plazo la falta de pruebas unitarias desemboca en un mal diseño del software y dificulta la refactorización.

Según Gmeiner et al. (2015) las pruebas de aceptación deben combinarse con pruebas manuales, pues con frecuencia se encuentran omisiones en los casos de prueba que se detectan durante la ejecución manual de las pruebas, o simplemente, no se pueden reproducir de manera automatizada.

La Figura 3.3 muestra un ejemplo de pipeline de entrega continua de un sistema desde el punto de vista del aseguramiento de calidad. Se describe al *pipeline* en función de etapas de pruebas. El proceso inicia con la etapa de *commit* en donde la entrada son las fuentes extraídas del repositorio de código, se realiza el análisis estático de código, se construye el software, se ejecutan pruebas unitarias y pruebas de integración. En caso de no encontrarse fallos o defectos en el software, en la siguiente etapa se construyen los entornos de prueba y se despliegan las aplicaciones para la ejecución de las "pruebas automatizadas de aceptación". En la tercera etapa, "Pruebas de aceptación de usuario" interviene el equipo de aseguramiento de calidad en donde se ejecutan pruebas manuales y automatizadas en un entorno similar al de producción. En la última etapa se realizan pruebas de capacidad y carga antes de la liberación de las aplicaciones. En cualquier momento en que se evidencie una situación que rompe el flujo de la entrega continua, las incidencias son comunicadas para la retroalimentación del proceso.

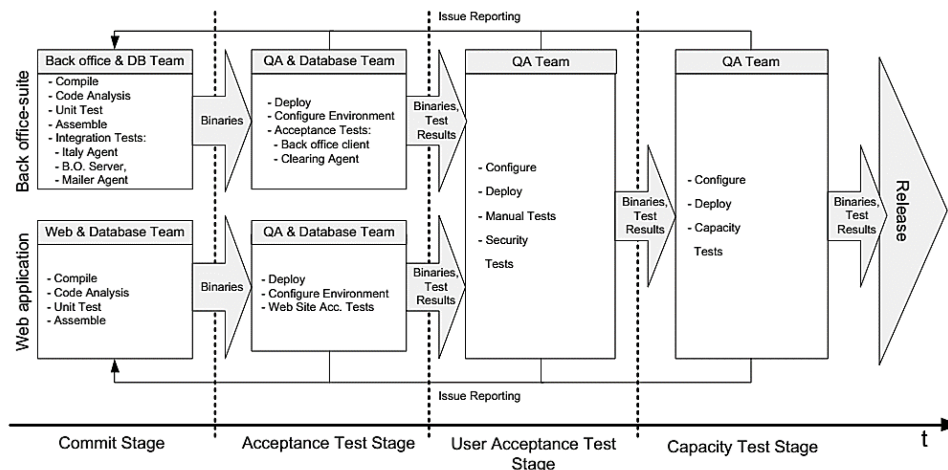


Figura 3.3 - Pipeline de entrega continua. Perspectiva de aseguramiento de calidad (Gmeiner, Ramler, & Haslinger, 2015)

Existe una tendencia marcada en el uso de *Test Driven Development* (TDD) como práctica habilitante para la automatización de las pruebas (Amrit & Meijberg, 2017) (Gmeiner, Ramler, & Haslinger, 2015) (Humble, 2010) (Vassallo, y otros, 2016). TDD permite encontrar una mayor cantidad de fallos de forma temprana y menores tiempos en la resolución de defectos (Amrit & Meijberg, 2017). La literatura indica que si bien no es un estándar, la adopción de TDD permite incrementar sustancialmente la cobertura de las pruebas (Amrit & Meijberg, 2017).

Se evita utilizar TDD para las pruebas de interfaz de usuario por las dificultades que presenta en el desarrollo de las pruebas que requieren herramientas para automatizar la captura de la interacción con los usuarios, convirtiéndose en una barrera para la metodología porque no permite una respuesta ágil a los cambios (Vassallo et al., 2016).

3.4 Entrega continua y microservicios

En el Capítulo 3 se realizó una introducción al concepto de microservicios. En general se concibe a los microservicios como un estilo arquitectónico nativo para la computación en la nube que tiene como objetivo realizar sistemas de software como un paquete de pequeños servicios. A nivel arquitectónico se busca un bajo acoplamiento de sus componentes debiendo ser desplegados de manera independiente. La entrega continua de software es una disciplina esencial para la adopción de los microservicios, pues a medida que el número de unidades desplegables aumenta se vuelve indispensable la automatización del proceso de despliegue para mantener el ciclo de vida de cada servicio independiente de los otros (Balalaie & Heydarnoori, 2016) (Haselböck, Weinreich, & Buchgeher, 2017).

La Figura 3.4 muestra: a) la instancia de un *pipeline* de entrega continua de software de una aplicación monolítica que se despliega sobre contenedores *Docker*, y, b) el mismo *pipeline* visualizado como instancias para el despliegue de *n* microservicios. En el gráfico además se incluyen los componentes: *GitLab*[46] como repositorio de código fuente, *Jenkins* como agente para la construcción y ejecución de pruebas automatizadas del software, *Artifactory*[47] como repositorio de gestión de binarios, *Docker* para la

[46] https://gitlab.com
[47] https://www.jfrog.com/artifactory/

construcción, almacenamiento de imágenes de los entornos de ejecución y *Kubernetes*[48] para la gestión, aprovisionamiento y despliegue de los contenedores.

Según Villamizar et al. (2015), el beneficio de poder desplegar (actualizar) los microservicios en cualquier momento, puede afectar a servicios dependientes que requieren trabajar con una versión específica de los mismos. El autor advierte que la falta de criterio para manejar el versionamiento de los servicios, podría generar inconsistencias importantes después del despliegue en los entornos de producción. Para evitar esta situación, Haselböck et al. (2017) sugieren realizar las pruebas de integración de manera individual al final del *pipeline* de entrega continua.

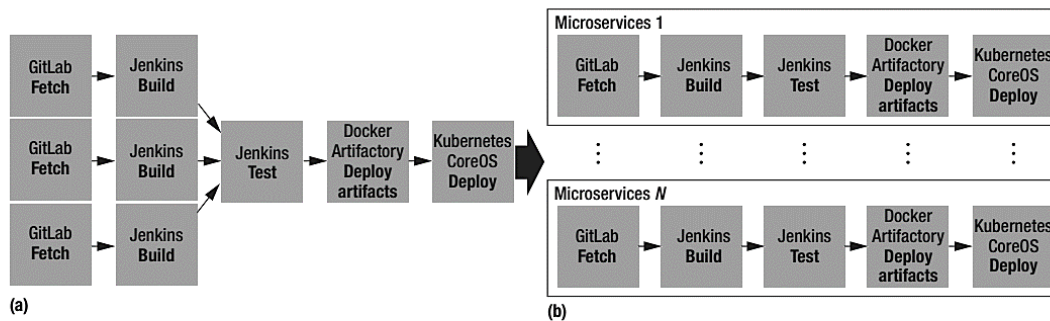


Figura 3.4 - Migración de un pipeline entrega continua de una arquitectura monolítica (a) a microservicios (b) (Balalaie & Heydarnoori,, 2016)

3.5 ENTREGA CONTINUA Y CONTENEDORES

En la Sección 2.5 se realizó una introducción a la virtualización como plataforma abordándose brevemente los conceptos principales sobre los contenedores. En la Sección 2.6 se describe la tecnología de contenedores *Docker,* su arquitectura y componentes.

Docker es una plataforma diseñada para facilitar la entrega de aplicaciones mediante el uso de una plataforma liviana de virtualización de contenedores, que provee un conjunto de herramientas y flujos de trabajo que ayudan a los desarrolladores a implementar y administrar las aplicaciones (Jaramillo, 2016).

Según Singh y Singh (2016) los contenedores *Docker* encajan de una manera natural como una tecnología apropiada para la arquitectura de microservicios, ya que cada uno

[48] https://kubernetes.io/

de ellos se puede utilizar como una unidad de despliegue para contener un servicio de manera granular sin interferir con otros microservicios. Los contenedores proveen ambientes adecuados para el despliegue de servicios en términos de velocidad, aislamiento y facilidad de despliegue de nuevas versiones (Jaramillo, 2016).

Docker provee la capacidad de expresar la creación y lanzamiento de los contenedores de forma declarativa como secuencias de comandos que pueden ser almacenados en un archivo de texto conocido dentro de la tecnología como *Dockerfile*.

Según Punjabi y Bajaj (2016) los archivos *Dockerfile* permiten tratar a la infraestructura como código produciendo los siguientes beneficios:

a) El código puede ser probado a fondo para reproducir infraestructura consistente a gran escala.
b) La infraestructura puede ser aprovisionada y configurada bajo demanda
c) El código de la infraestructura puede ser versionado en el sistema de control de versiones.
d) Los desarrolladores pueden acceder a un entorno simulado de producción incrementando la fiabilidad y las capacidades de prueba.
e) Permite la recuperación proactiva ante fallos mediante la supervisión continua de los entornos de ejecución para la identificación de problemas y la ejecución automática de scripts para restauración de los servicios a un estado consistente.

Como beneficio adicional Virmani (2015) destaca que el proceso de despliegue gana consistencia y repetibilidad permitiendo que pueda ser incluido como un elemento más del *pipeline* de entrega continua de software.

En la propuesta realizada por Soni (2015), el aprovisionamiento de infraestructura puede darse durante la ejecución de las pruebas de integración, pruebas de aceptación, pruebas manuales y en el despliegue en producción. La Figura 3.5 muestra una adecuación del *pipeline* de (Gmeiner, Ramler, & Haslinger, 2015) que visualiza los momentos en los que podría realizarse el aprovisionamiento y/o reutilización de las imágenes de *Docker*.

Según Cito et al. (2017) si bien en el *pipeline* de entrega continua se puede construir imágenes de *Docker* de acuerdo a las especificaciones de un archivo *Dockerfile* tomado

del repositorio de control de versiones, el proceso de construcción de las imágenes puede tomar un tiempo considerable si se requiere descargar recursos de internet, convirtiéndose en una situación problemática que entorpece el concepto de la liberación ágil de software. Para mitigar este problema Cito et al. (2017) sugieren la separación en capas del contenedor para descargar exclusivamente las imágenes que no puedan ser obtenidas del *Docker registry* que actúa como caché local.

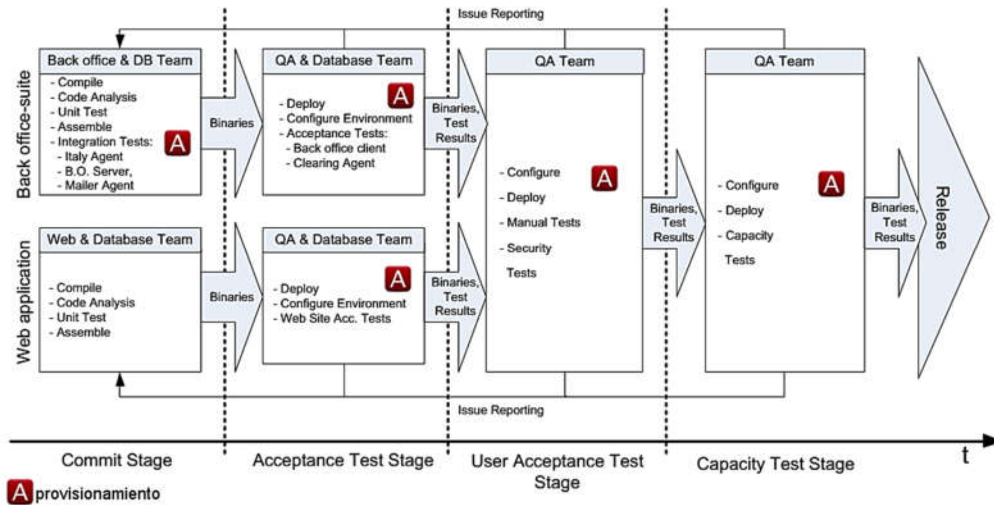


Figura 3.5 - Visualización de los momentos de aprovisionamiento de infraestructura en el pipeline de entrega continua de Gmeiner.

4 ARQUITECTURA TECNOLÓGICA PLANTEADA

En este capítulo se presenta el proceso de definición de componentes y la arquitectura tecnológica resultante para la implementación de un *pipeline* de entrega continua de software de acuerdo a los requerimientos de la plataforma tecnológica para implementación de aplicaciones de la Universidad de Cuenca.

4.1 SELECCIÓN DE LOS COMPONENTES DE SOFTWARE

Según Duvall et al. (2007), elegir el software adecuado para la automatización del proceso es una cuestión de encontrar la mejor opción para un entorno y proceso de desarrollo en particular. La mejor herramienta es la que más trabajo ahorra al equipo de desarrollo y le sirve por más tiempo. A juicio de los autores las conversaciones de comparación de herramientas a menudo pueden trascender lo práctico y escalar a lo que parece un debate religioso. Además indican, que se debe tomar en cuenta que la elección de herramientas no tiene por qué ser un compromiso de por vida; si llega a serlo, es un indicador de que la herramienta funciona dentro del ecosistema.

La selección de los componentes de software de la arquitectura tecnológica para el *pipeline* de entrega continua de software se realizará considerando: i) reporte de uso en el análisis del estado del arte, ii) *stack* de tecnologías para desarrollo de aplicaciones planteado por la Universidad de Cuenca, iii) requerimientos no funcionales, y, iii) características que permitan la conformidad con el proceso de entrega continua plantado.

4.1.1 REQUERIMIENTOS NO FUNCIONALES

La Tabla 4.1 muestra los requerimientos no funcionales que se encuentran en conformidad con la plataforma tecnológica para implementación de aplicaciones de la Universidad de Cuenca. Estos han sido especificados usando la clasificación de requerimientos no funcionales de Sommerville (2002).

Tabla 4.1 - Requerimientos no funcionales de la plataforma tecnológica para implementación de aplicaciones de la Universidad de Cuenca

Número	Requerimiento	Descripción
RNF1	Reusabilidad / Escalabilidad / Facilidad de instalación	Los servicios de negocio deben ser implementados aplicando microservicios como estilo arquitectónico

RNF2	Estándares organizacionales	Los componentes de servidor deben ser programados en el lenguaje Java 8 o superior.
RNF3	Estándares organizacionales	Se debe utilizar el framework Spring Boot para el desarrollo y empaquetamiento de microservicios.
RNF4	Disponibilidad / Portabilidad / Escalabilidad / Facilidad de instalación	El despliegue de los servicios debe gestionarse a través de Kubernetes.
RNF5	Interoperabilidad	Los servicios de negocio deben implementar interfaces de comunicación tipo REST
RNF6	Disponibilidad / Portabilidad / Escalabilidad / Facilidad de instalación	Las aplicaciones deben estar desplegadas en entornos de alta disponibilidad usando tecnología de contenedores
RNF7	Estándares organizacionales / Usabilidad / Portabilidad	Las aplicaciones deben estar programadas en capas implementando un patrón arquitectónico MVC usando el framework AngularJS 2.
RNF8	Estándares organizacionales / Usabilidad / Portabilidad	Las interfaces de usuario deben estar desarrolladas con HTML5
RNF9	Actores externos / Regulaciones legales	El Código Ingenios publicado en el Registro Oficial en diciembre de 2016, mediante la décima transitoria se establece que las instituciones públicas tienen un plazo de hasta cinco años para migrar de manera obligatoria a software libre.

4.1.2 ETAPAS DEL PIPELINE DE ENTREGA CONTINUA DE SOFTWARE

En la Sección 3.2, durante el análisis de los trabajos relacionados, se estableció que en una de las revisiones sistemáticas de literatura más recientes (Shahin, Babar, & Zhu, 2017) se determinó que en la actualidad en los procesos de entrega continua se están construyendo *pipelines* en una combinación de siete etapas, algunas de ellas implementadas a discreción en función de los requisitos y objetivos de aseguramiento de calidad de cada proyecto.

El presente trabajo establece un diseño de una arquitectura tecnológica que contempla en el contexto más amplio las siete etapas, dejando a criterio de los líderes de proyecto las decisiones sobre el *pipeline* de entrega continua en cuanto a la inclusión o exclusión de una determinada etapa en función del contexto propio de cada proyecto.

La Figura 4.1 presenta una visión inicial del flujo y la secuencia de las actividades dentro del pipeline de entrega continua de software planteado.

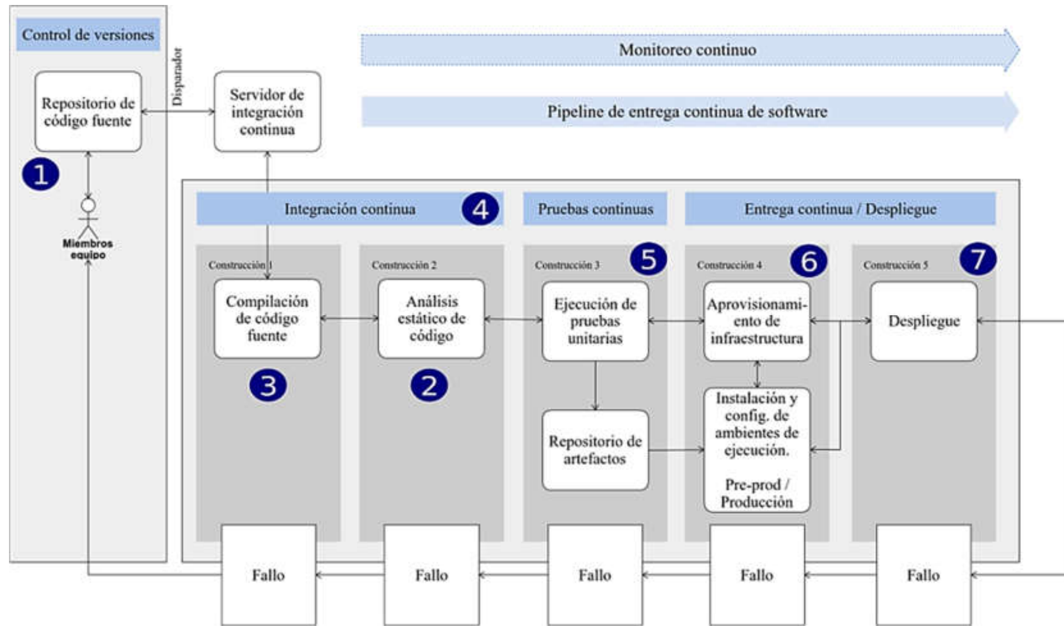


Figura 4.1 - Etapas y flujo de actividades del pipeline de entrega continua

Id	Etapa
1	Control de versiones
2	Análisis y gestión de código
3	Construcción
4	Integración continua
5	Pruebas
6	Configuración y aprovisionamiento
7	Entrega continua

A continuación, se definen los componentes de la arquitectura, considerando que el requerimiento no funcional RNF9, restringe de manera determinante los productos candidatos para la selección del software a utilizar en el flujo del *pipeline* de entrega continua exclusivamente a software libre.

4.1.2.1 INTEGRACIÓN CONTINUA

El servidor de integración es el núcleo del proceso de entrega continua de software, razón por la que es el primer componente que se debe seleccionar. La decisión que se

tome con respecto a este afectará de manera determinante al grupo de alternativas que podrían ser consideradas para ser utilizadas en las demás etapas del proceso. Afortunadamente la abundancia de las evidencias permite salvar la necesidad de realizar un estudio profundo con respecto a este componente. *Jenkins* es el software que mayores beneficios ofrece en el mercado por sus características en cuanto a evolución, popularidad, tamaño de comunidad, soporte comercial, mecanismos de integración con otras herramientas, *plugins*, lenguajes de desarrollo soportados, modalidades de despliegue, etc. En distintos estudios se resalta su utilización y resultados (Shahin, Babar, & Zhu, 2017) (Soni , 2015) (Rai, Madhurima, Dhir, Madhulika, & Garg, 2015) (Pulkkinen, 2013). El índice BOSS (The BOSS Index: Tracking the Explosive Growth of Open-Source Software, 2017) lo ubica en la posición 14 por sobre otros proyectos *open source* emblemáticos como la base de datos relacional Postgresql ubicada en la posición 16.

Las funciones que Jenkins cumplirá en el *pipeline* como servidor de integración continua serán:

a) Comunicar a los componentes con el sistema de control de versiones.
b) Ejecutar procesos de construcción y despliegue automático cuando se registren cambios en el repositorio de código fuente.
c) Mantener la configuración del *pipeline*.
d) Solicitar la ejecución de análisis estático de código.
e) Orquestar el registro y distribución de binarios en el servidor de artefactos.
f) Recuperar registros y visualizar resultados de procesos.
g) Solicitar el aprovisionamiento de infraestructura cuando el *pipeline* lo requiera.
h) Construir los ambientes de ejecución y despliegue en producción.

En resumen, es el ente orquestador de todas las operaciones dentro del pipeline de entrega continua de software.

4.1.2.2 CONTROL DE VERSIONES

El estado del arte y los trabajos relacionados abordados en el presente estudio, indican que Git es el software que se ha convertido en el estándar de facto como sistema de control de versiones en los proyectos de software, que además posee soporte nativo en el servidor

de integración continua Jenkins. El índice de software libre BOSS (2017) posiciona a Git en la ubicación número 2 del ranking después del proyecto Linux.

En la arquitectura tecnológica para el *pipeline* que se plantea en el presente proyecto, se utiliza Git para las actividades de control de versiones de los artefactos que requieran gestión de configuración durante el proceso de entrega continua de software.

4.1.2.3 ANÁLISIS Y GESTIÓN DE CÓDIGO

En el mundo del software libre existen varias opciones para lo que se conoce de manera técnica como herramientas para análisis estático de código. El requerimiento funcional RNF2 establece que las aplicaciones deberán ser desarrolladas con el lenguaje de programación Java, por tanto, las alternativas a considerar son las siguientes: i) FindBugs, ii) PMD, y, iii) SonarQube. Este último no se registra en las primeras 40 posiciones del índice BOSS (2017); sin embargo, tanto el estado del arte como los artículos académicos revisados en esta investigación, reportan un mayor uso de SonarQube en esta etapa. Este producto ofrece una serie de mecanismos de integración con otros componentes de software, específicamente con Jenkins, beneficios que finalmente nos hacen decantar por esta solución.

4.1.2.4 CONSTRUCCIÓN

La construcción de software es la única etapa que es obligatoria para lenguajes de programación que requieren compilación para su ejecución. Las opciones que existen de software libre para la construcción con soporte para Java son: i) Apache Ant, ii) Apache Maven, y, iii) Gradle. Este último ha venido ganando una importante participación de mercado gracias a que Google la estableció como herramienta de construcción de software por omisión para las aplicaciones basadas en el sistema operativo Android. Por otra parte, Ant ha perdido vigencia en los últimos años por la enorme aceptación de Apache Maven gracias a dos características fundamentales: su esquema de gestión de dependencias y el amplio ecosistema de *plugins* que permiten realizar desde tareas triviales como el empaquetamiento de binarios, hasta operaciones complejas como la construcción de imágenes Docker, por ejemplo. En el índice BOSS (2017) se encuentra en la posición 30. Maven es la herramienta que utilizaremos para la etapa de construcción en el *pipeline* de entrega continua de software.

4.1.2.5 PRUEBAS

La automatización de las pruebas revierte una mayor complejidad en la definición de las herramientas a utilizar dentro del proceso. Es necesario hacer un análisis de toda la arquitectura de la aplicación para identificar en cada capa la herramienta adecuada. Para el caso de la Universidad de Cuenca de acuerdo a la Figura 2.1, se definen las herramientas con se indica a continuación:

Capa de interfaz de usuario

La interfaz de usuario debe estar implementada con el framework de Google para desarrollo de aplicaciones web Angular JS (RNF7). Entre los productos para automatización de pruebas se encontró: i) Jasmine [49]+ Karma[50], una combinación de herramientas que permiten implementar prácticas de TDD a manera de pruebas unitarias. ii) Protractor[51], es un framework para pruebas en la modalidad E2E (de inicio a fin). Este tipo de pruebas son difíciles de mantener, pues cualquier cambio en la interfaz de usuario requiere la actualización de las pruebas. Se ejecuta sobre un ambiente virtualizado "*X Windows*" para verificar las interfaces de usuario y se fundamenta en Jasmine para su ejecución. Para esta capa se ha seleccionado Protractor como framework de pruebas para la interfaz de usuario en gran medida, por el *plugin* disponible para el servidor de integración continua Jenkins.

Microservicios y capa de acceso a datos.

Las capas de microservicios y acceso a datos estarán implementadas con Spring Boot (RNF3). Spring provee su propio framework para automatización de pruebas que a su vez está implementado sobre el framework JUnit. Spring es un framework que trabaja fuera de los estándares de java, por tanto, no existe una alternativa viable para la automatización de las pruebas de esta capa.

[49] https://jasmine.github.io
[50] https://karma-runner.github.io
[51] www.protractortest.org

4.1.2.6 Configuración y aprovisionamiento

Según Benson et al. (2016), existen 3 herramientas que pueden utilizarse para el aprovisionamiento de infraestructura: i) Ansible[52], ii) Chef[53], y, iii) Puppet[54]. La ventaja de Ansible frente a Chef y Puppet es que no requiere instalar ningún componente en el servidor destino, el trabajo lo hace mediante una sesión ssh. En contraposición, la arquitectura cliente servidor de Chef y Puppet ofrecen mejores tiempos de respuesta durante la ejecución del aprovisionamiento de recursos.

En el escenario de la Universidad de Cuenca, la infraestructura por aprovisionar estará basada en contenedores Docker; por tanto, el conjunto de herramientas que provee Kubernetes para la gestión y ejecución de contenedores, son suficientes para cumplir con el objetivo. Esta herramienta es soportada por Google y se encuentra en la posición 33 del índice BOSS (2017). En el *pipeline* se utilizará el *plugin* Kubernetes para Jenkins que facilita la inclusión de los procesos de aprovisionamiento como parte del *pipeline* propuesto.

4.1.2.7 Despliegue

El despliegue de las aplicaciones dependerá del entorno del proyecto, generalmente Kubernetes tiene las capacidades suficientes para hacerlo; sin embargo, se hace uso de OpenShift que es una plataforma de aplicaciones que integra las funcionalidades de Docker y Kubernetes. Independientemente de la arquitectura de las aplicaciones, la herramienta permite diseñar, desarrollar e implementar de forma fácil y rápida un *pipeline* en casi cualquier infraestructura ya sea ésta una nube pública o privada. Puede ejecutarse dentro del servicio de computación en la nube provista por Redhat, en una instalación Openshift Onpremise y para desarrollo, en su versión Minishift. Esta plataforma se encuentra ubicada en el índice BOSS (2017) sobre Kubernetes en la posición 32.

4.2 Arquitectura tecnológica resultante

La Figura 4.2 complementa la definición de las etapas del *pipeline* y el flujo de actividades para el proceso de entrega continua (Figura 4.1) con los componentes de

[52] https://www.ansible.com
[53] https://www.chef.io
[54] https://puppet.com

software seleccionados a lo largo de este capítulo para conformar la arquitectura tecnológica resultante. Adicionalmente, en el diagrama se especifican las actividades y componentes de software mapeados a los objetivos de despliegue de (Bellomo, Ernst, Nord, & Kazman, 2014):

G1) Habilitar la construcción y entrega continua de software.

G2) Habilitar la automatización de pruebas.

G3) Habilitar un despliegue rápido y operaciones robustas.

G4) Habilitar entornos sincronizados y flexibles.

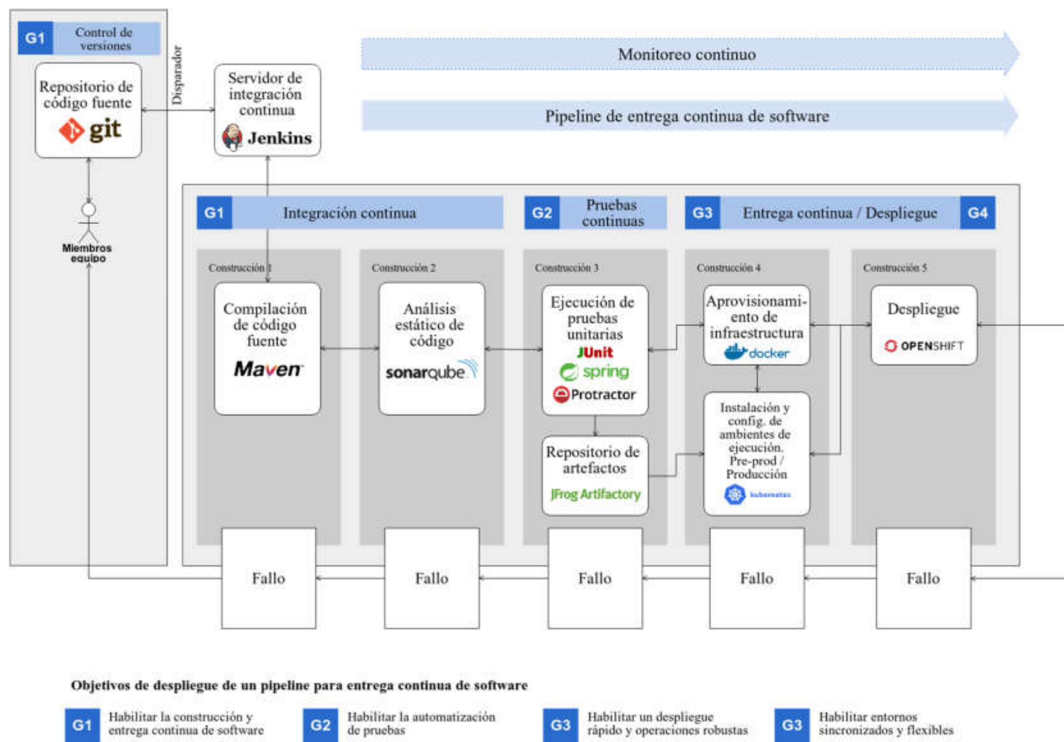


Figura 4.2 - Arquitectura tecnológica para entrega continua de software en la Universidad de Cuenca

Tabla 4.2 - Resumen de componentes de software de pipeline de entrega continua de software

Elemento	Recurso / Tarea	***Stack* de tecnologías**	Descripción
Control de versiones	Repositorio de código fuente	GIT	Software *open source* para gestión de código fuente (SCM)
Integración continua	Servidor de integración continua	Jenkins	Software *open source* para integración continua de software

Elemento	Recurso / Tarea	***Stack* de tecnologías**	**Descripción**
Integración continua	Compilación de código fuente	Maven	Software *open source* para construcción de software
Integración continua	Análisis estático de código	Sonar	Software *open source* para analizar la calidad del código, comentarios, reglas de codificación, potenciales defectos de software, código duplicado, etc
Pruebas continuas	Ejecución de pruebas unitarias	JUnit	Software *open source* para automatización de pruebas unitarias
Pruebas continuas	Ejecución de pruebas unitarias	JUnit [55]/ SpringRunner / SpringBootTest	Software *open source* para automatización de pruebas unitarias de microservicios implementados con SpringBoot
Pruebas continuas	Ejecución de pruebas unitarias	JUnit / SpringRunner / SpringBootTest	Software *open source* para automatización de pruebas unitarias de microservicios implementados con SpringBoot
Pruebas continuas	Ejecución de pruebas unitarias	Protractor	Software *open source*. Framework para pruebas de interfaz de usuario basada en web
Pruebas continuas	Ejecución de pruebas unitarias	Mock Service Layer	Software *open source* para realizar pruebas de interfaz de usuario simulando la respuesta del servidor. Aliviana el consumo de recursos focalizando las pruebas exclusivamente en la funcionalidad de la interfaz de usuario.
Integración continua / Pruebas continuas	Repositorio de binarios	Artifactory	Software *open source* para versionamiento de binarios de aplicación. Maneja y controla el flujo de artefactos binarios
Entrega continua / Despliegue	Aprovisionamiento de infraestructura	Kubernetes	Software *open source* para el despliegue, escalabilidad y operaciones de contenedores

[55] http://junit.org/junit5/

Elemento	Recurso / Tarea	***Stack* de tecnologías**	Descripción
Entrega continua / Despliegue	Instalación y configuración de ambientes de ejecución	Maven / Docker plugin	Software *open source*. Plugin de Maven que permite la construcción parametrizable de imágenes Docker para entornos de ejecución
Entrega continua / Despliegue	Despliegue	Kubernetes	Software *open source* para el despliegue, escalabilidad y operaciones de contenedores

4.2.1. ASEGURAMIENTO DEL PIPELINE

El *pipeline* de entrega continua de software está conformado por un conjunto de componentes de que trabajan como un sistema interactuando entre sí para conseguir que el software se entregue de manera predecible y confiable. Este sistema debe ser asegurado al igual que las aplicaciones y los entornos de ejecución que se entregan como producto del proceso. Es necesario entonces, que toda la cadena de herramientas de entrega continua y los entornos de construcción y pruebas generen confianza en la integridad de la entrega y la cadena de custodia, no solo por motivos de cumplimiento y seguridad, sino también para garantizar que los cambios se realicen de forma segura, repetible y rastreable.

La cadena de herramientas de entrega continua como tal podría constituirse en un objetivo de ataque peligroso, debido a que proporciona una ruta clara para realizar cambios al software y empujarlos automáticamente a producción. Si es vulnerable, los atacantes tienen un camino fácil para acceder a los entornos de desarrollo, pruebas y producción. Se podrían exponer datos o propiedad intelectual, inyectar malware en cualquier parte del entorno, introducir ataques de denegación de servicio, o paralizar la capacidad de la organización para responder a un ataque al bloquear el *pipeline* en sí mismo.

También se debe proteger el *pipeline* de los ataques internos asegurando que todos los cambios sean completamente transparentes y rastreables de un extremo a otro, evitando que una persona que posea información interna de la organización actúe de

forma maliciosa realizando cambios, saltándose verificaciones o validaciones, sin ser detectada.

Para asegurar el *pipeline* se debe considerar la planificación y ejecución de las siguientes actividades:

a) Asegurar y auditar el acceso al repositorio de código fuente Git y el repositorio de binarios Artifactory.
b) Implementar el control de acceso a todos los componentes de la cadena de integración continua. No permitir acceso anónimo o compartido a los repositorios, servidor de entrega continua o a cualquier otra herramienta.
c) Asegurar el servidor de integración y entrega continua (Jenkins), y los sistemas de compilación (Maven) y despliegue (Kubernetes). Las herramientas como Jenkins están diseñadas para la comodidad del desarrollador y no son seguras por defecto.
d) Verificar que las herramientas de la cadena de entrega continua (y los complementos necesarios) se mantengan actualizados con sus últimas versiones y probarlas con frecuencia.
e) Verificar que las claves, credenciales y otros secretos estén protegidos. Se debe eliminar toda referencia de secuencias de comandos (*scripts*), código fuente y archivos de texto sin formato. Usar un administrador de secretos seguro y auditado como KeyWhiz[56].
f) Introducir en el proceso de compilación y construcción los pasos necesarios para firmar binarios y otros artefactos para evitar manipulaciones.
g) Revisar periódicamente los registros para asegurarse de que estén completos y para que se pueda rastrear un cambio de principio a fin. Asegurarse de que los registros sean inmutables, y que no se puedan borrar ni falsificar.
h) Verificar que todos los componentes y sistemas sean monitoreados como parte del entorno de producción.

[56] https://square.github.io/keywhiz/

4.1 PROTOTIPO DE ENTREGA CONTINUA DE SOFTWARE

Por la extensión del proceso de creación del prototipo, en esta sección se presenta de manera resumida los componentes elementales que permitieron validar la arquitectura tecnológica propuesta a través del prototipo en el cual se configuró todos los componentes para la implementación del *pipeline* de entrega continua de software. La Figura 4.3 esquematiza los pasos que se realizaron durante el proceso.

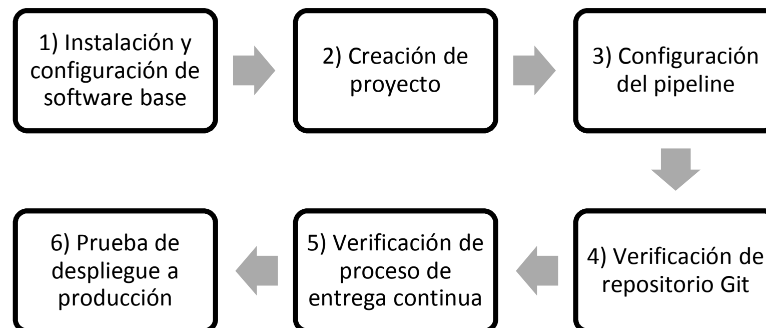


Figura 4.3 - Proceso de creación de prototipo

4.1.1 INSTALACIÓN Y CONFIGURACIÓN DEL SOFTWARE BASE

La creación del *pipeline* de entrega continua requiere la instalación de todo el software base: i) sistema de control de versiones (Git); ii) servidor de integración continua (Jenkins); iii) plataforma para virtualización de contenedores (Docker); y, iv) los plugins de Jenkins que permiten realizar la construcción (Maven), análisis estático de código (SonarQube), ejecución de pruebas unitarias (JUnit + SpringBootTest + Protractor), aprovisionamiento de infraestructura (Kubernetes).

Durante la investigación se encontró que el software *open source* Fabric8, automatiza todo el proceso de creación, ejecución, puesta en marcha y monitoreo del pipeline de entrega continua, por lo que se decidió trabajar con esta herramienta para la construcción del prototipo que permita validar la prueba de concepto.

A continuación se presentan los comandos para la instalación de Fabric8 en un sistema Linux:

```
# -- Descarga e instalación de fabric8
/dev/fabric8$ curl -sS https://get.fabric8.io/download.txt | bash

#-- Registro de fabric8 en el path del Sistema operative
/dev/fabric8$ echo 'export PATH=$PATH:~/.fabric8/bin' >> ~/.bashrc
/dev/fabric8$ source ~/.bashrc

# -- Instalación del runtime para contenedores KVM
/dev/fabric8$ sudo apt install libvirt-bin qemu-kvm
/dev/fabric8$ sudo usermod -a -G libvirtd lainiguez
/dev/fabric8$ newgrp libvirtd

# -- Instalación de Docker
sudo curl -L https://github.com/dhiltgen/docker-machine-kvm/releases/download/v0.7.0/docker-machine-driver-kvm -o /usr/local/bin/docker-machine-driver-kvm
sudo chmod +x /usr/local/bin/docker-machine-driver-kvm
/dev/fabric8$ eval $(gofabric8 docker-env)

# -- Inicio de fabric8
/dev/fabric8$ gofabric8 start --vm-driver=virtualbox

# -- Iniciar consola, servicio Git, servicio Jenkins
/dev/fabric8$ gofabric8 console
/dev/fabric8$ gofabric8 service gogs
/dev/fabric8$ gofabric8 service jenkins
```

Una vez terminada la instalación, se puede acceder a un navegador de internet para visualizar la consola de administración de la herramienta, como se muestra en la Figura 4.4.

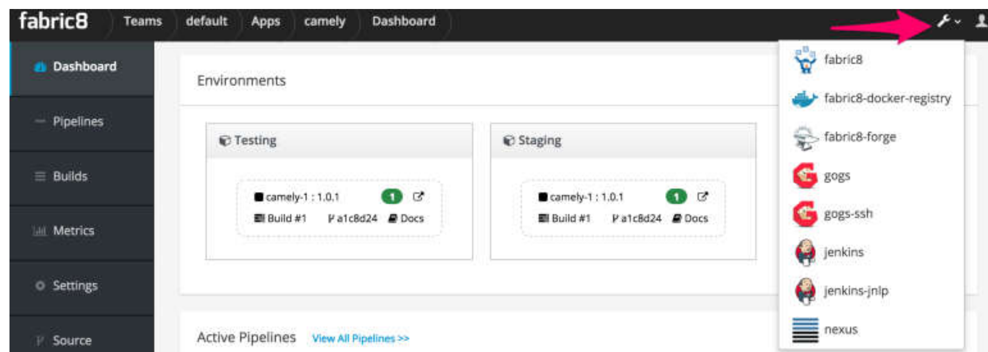


Figura 4.4 - Consola de administración de fabric8

4.1.2 CREACIÓN DE PROYECTO

Para crear un proyecto, se ingresa a la consola de Fabric8, se selecciona la opción "Teams" como se muestra en la Figura 4.5

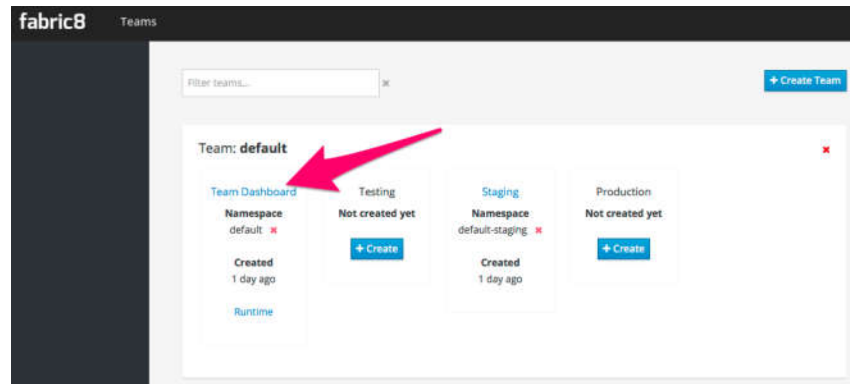


Figura 4.5 - Ingreso al dashboard del equipo de proyecto

Luego se selecciona la opción crear aplicación como se muestra en la Figura 4.6

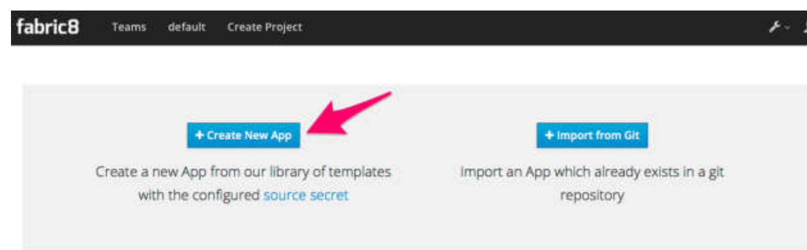


Figura 4.6 - Creación de aplicación en fabric8

A continuación se define el tipo de proyecto, que para este caso es Spring Boot como se muestra en la Figura 4.7 y Figura 4.8

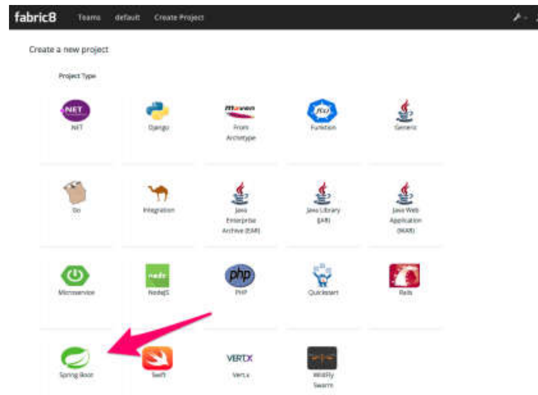


Figura 4.7 - Creación de proyecto fabric8 con tecnología Spring Boot

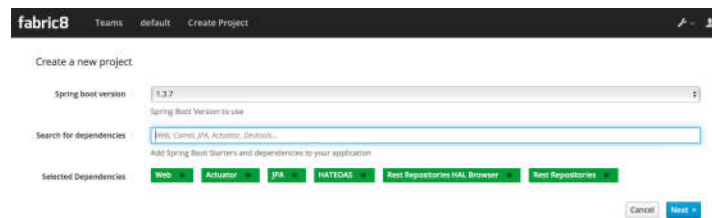


Figura 4.8 - Creación de proyecto fabric8, parámetros de proyecto

4.1.3 CONFIGURACIÓN DEL PIPELINE

Una vez creado el proyecto se procede a definir el *pipeline*, para lo cual Fabric8 ofrece plantillas que permiten establecer los pasos a realizar dentro del *pipeline* y las tecnologías a utilizar en cada uno de ellos para la ejecución de actividades (se excluye los detalles de parametrización de cada componente), como se muestra en la Figura 4.9

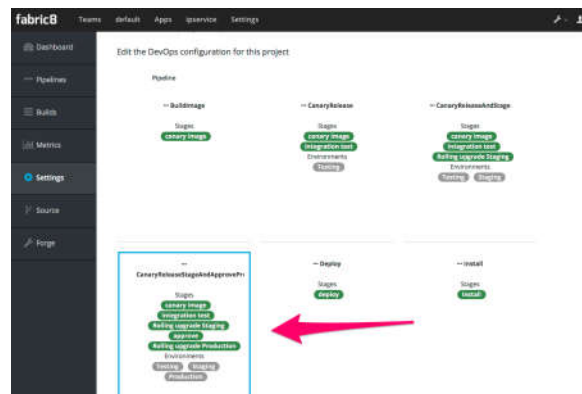


Figura 4.9 - Selección de plantilla de pipeline de entrega continua de software

4.1.4 VERIFICACIÓN DE REPOSITORIO GIT

Una vez creado el proyecto y definido el *pipeline* de entrega continua de software, se debe verificar el funcionamiento del repositorio Git como se muestra en la Figura 4.10

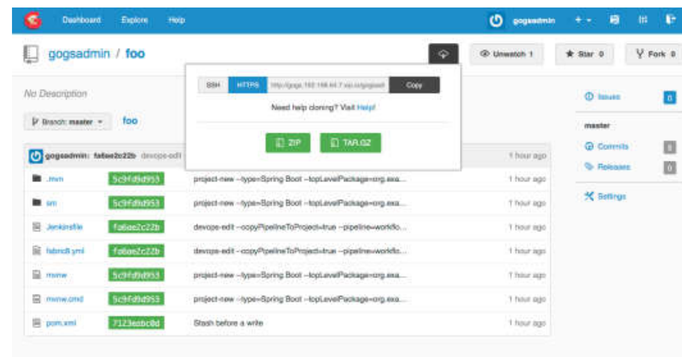


Figura 4.10 - Visualización del repositorio Git

4.1.5 VERIFICACIÓN DEL PROCESO DE ENTREGA CONTINUA

Una vez verificado el repositorio, se regresa al *dashboard* para visualizar el *pipeline* recientemente creado (Figura 4.11).

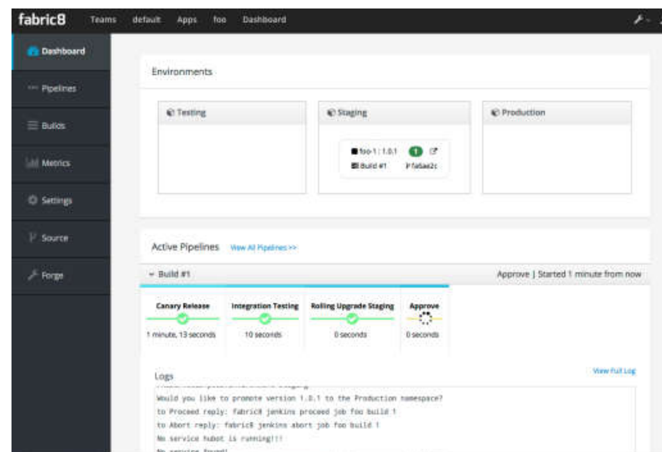


Figura 4.11 - Pipeline de entrega continua

4.1.6 PRUEBA DE DESPLIEGUE EN PRODUCCIÓN

Luego de haber ejecutado el pipeline las veces que fuesen necesarias, y si todas las etapas se han ejecutado satisfactoriamente (estado identificado por el color verde) se puede desplegar en producción la aplicación haciendo click en el botón "*Procced*" (Figura 4.12) encargándose de crear las imágenes de Docker para trasladar los contenedores con la solución a producción de manera automática.

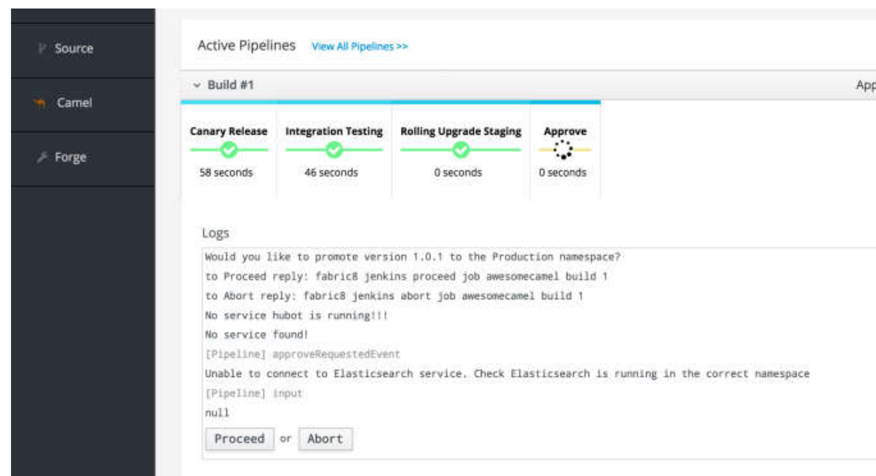


Figura 4.12 - Despliegue de solución a producción con Fabric8

5 CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se presentan las conclusiones y resultados obtenidos durante el desarrollo del trabajo de tesis. Adicionalmente se define el trabajo complementario para mejorar el *pipeline* de entrega continua de software.

4.3 CONCLUSIONES

- La selección de tecnologías y la automatización del proceso de entrega continua de software no es una tarea trivial. A la hora de definir la arquitectura de software y el *stack* de tecnologías que darán soporte a las aplicaciones se deben mantener presentes los objetivos de despliegue para que el diseño del *pipeline* produzca los resultados esperados.
- Se debe implementar dentro la organización prácticas como el *Test Driven Development* para la obtención de mejores resultados del proceso de automatización con un índice apropiado de cobertura de las pruebas.
- El equipo de desarrollo debe estar preparado para afrontar nuevos retos, pues estos exigen un mayor nivel de madurez en los procesos de Ingeniería de Software implementados en la organización.
- Docker no debe verse como la herramienta mágica que salva a las personas de los flujos de trabajo obsoletos, sino como una herramienta más que puede facilitar los procesos hacia un nuevo pensamiento ágil. A pesar de que los contenedores ayudaron al proceso de convertir la infraestructura en código, no fue una herramienta indispensable.
- Con la implementación de un *pipeline* de entrega continua de software los desarrolladores obtienen varios beneficios: i) disponibilidad inmediata de entornos similares a los de producción; ii) automatización inmediata de tareas repetitivas; iii) menor tiempo de aprobación para la conformidad de las tareas realizadas; iv) retroalimentación más rápida de los usuarios para mejorar el software.

Como resultado final se obtuvo un modelo de entrega continua basado en tecnología que provee un soporte completo al stack tecnológico definido por la unidad de desarrollo de software de la Dirección de Tecnologías de la Información de la

Universidad de Cuenca con herramientas específicas para la ejecución de pruebas de cada uno de los componentes de la arquitectura de la solución, implementándose una prueba de concepto para validar la arquitectura tecnológica planteada.

4.4 TRABAJOS FUTUROS

- En el trabajo realizado se determinó que si bien se implementó un proceso de entrega continua, no se definió el proceso de automatización del registro y el descubrimiento de microservicios, operaciones que permiten determinar las direcciones (*end points*) que varían dinámicamente durante las operaciones de despliegue y escalamiento de los servicios, haciendo posible que los consumidores puedan conectarse a ellos. Para complementar este trabajo se podría buscar alternativas que permitan el uso de herramientas especializadas como ZooKeeper y Kubernetes, para cubrir la brecha detectada.

- Adicionalmente, es necesario que esta propuesta se aplique en la unidad de desarrollo de software de la Dirección de Tecnologías de la Información de la Universidad de Cuenca, de manera que la arquitectura tecnológica planteada pueda empezar a generar la retroalimentación necesaria que permita su evolución para adaptarse en el corto plazo a los procesos de Ingeniería de Software que se busca institucionalizar en la organización.

6 TRABAJOS CITADOS

Beck, K. (2000). *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional.

Myers, G., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing.* Wiley Publishing ©2011.

Akerele, O., Ramachandran, M., & Dixon, M. (2013). System Dynamics Modeling of Agile Continuous Delivery Process. *2013 Agile Conference* (págs. 60 - 63). Nashville, USA : IEEE.

Ambler, S. (2007). Test-Driven Development of Relational Databases. *IEEE Software, 24(3)*, 37 - 43.

Amrit, C., & Meijberg, Y. (2017). Effectiveness of Test Driven Development and Continuous Integration – A Case Study. *IT Professional (99)*, 1-1.

Bakshi, K. (2017). Microservices-based software architecture and approaches. *IEEE Aerospace Conference.* Big Sky: IEEE computer society.

Balalaie, A., & Heydarnoori, A. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software 33(3)*, 42-52.

Balalaie, A., & Heydarnoori,, A. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3), 42-52.

Bartusevics, A., & Novickis, L. (2015). Model-Based Approach for Implementation of Software Configuration Management Process. *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)* . Angers, France: IEEE.

Beck, K. (February de 2001). *Agile Manifest Principles* . Obtenido de http://agilemanifesto.org/iso/es/principles.html

Bellomo, S., Ernst, N., Nord, R., & Kazman, R. (2014). Toward Design Decisions to Enable Deployability. *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (págs. 702 - 707). Atlanta, USA : IEEE.

Benson, J., Prevost, J., & Rad , P. (2016). Survey of Automated Software Deployment for Computational and Engineering Research. *Systems Conference (SysCon), 2016 Annual IEEE.* Orlando, USA: IEEE.

Berner, S., Weber, R., & Keller, R. (2005). Observations and Lessons Learned from Automated Testing. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* (págs. 571-579). Saint Louis, USA: IEEE.

Celesti, A., Mulfari, D., Fazio, M., Villari, M., & Puliafito, A. (2016). Exploring Container Virtualization in IoT Clouds. *2016 IEEE International Conference on Smart Computing (SMARTCOMP)* (págs. 1-6). St. Louis, USA : IEEE.

Chen, L. (2015). Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*, 50-54.

Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S., & Gall, H. C. (2017). An Empirical Analysis of the Docker Container Ecosystem on GitHub. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (págs. 323 - 333). Buenos Aires, Argentina : IEEE.

Daya, S., Van Duy , N., Eati, K., & Ferreira, C. (2015). *Microservices from Theory to Practice.* IBM.

Di Francesco, P., Lago, P., & Malavolta, I. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. *IEEE International Conference on Software Architecture* (págs. 21-30). Gothenburg, Sweden: IEEE.

Do, N. H., Do, T. V., Tran, X. T., Farkas, L., & Rotter, C. (2017). A scalable routing mechanism for stateful microservices. *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)* , 72-78.

Dustin, E., Garrett, T., & Gauf, B. (2009). *mplementing automated software testing : how to save time and lower costs while raising quality.* Addison-Wesley.

Duvall, P., Matyas, S., & Glover, A. (2007). *Continuous Integration - Improving Software Quality and Reducing Risk.* Addison-Wesley Professional.

Elberzhager, F., Rosbach, A., Münch, J., & Eschbach, R. (2012). Reducing test effort: A systematic mapping study on existing approaches. *Elsevier*, 1093-1106.

Elberzhager, F., Rosbach, A., Münch, J., & Eschbach, R. (2012). Reducing test effort: A systematic mapping study on existing approaches . *Information and Software Technology (54)*, 1092-1106.

Fecko, M., & Lott, C. (2002). Lessons learned from automating tests for an operations support system. *Software: Practice and Experience 32(15)*, (págs. 1485-1506).

Fitzgerald, B., & Stol, K.-J. (2015). Continuous software engineering: A roadmap and agenda. *The Journal of Systems and Software*.

Fowler, M. (1 de Mayo de 2006). *Continuous Integration.* Obtenido de https://martinfowler.com/articles/continuousIntegration.html

Fowler, M. (Mayo de 2013). *Continuous Delivery.* Obtenido de http://martinfowler.com/bliki/ContinuousDelivery.html

Fowler, M. (30 de Mayo de 2013). *Deploymentpipeline.* Obtenido de http://martinfowler.com/bliki/DeploymentPipeline.html

Fowler, M. (15 de Enero de 2014). *BoundedContext.* Obtenido de https://martinfowler.com/bliki/BoundedContext.html

Fowler, M., & Highsmith, J. (August de 2001). *The Agile Manifesto.* Obtenido de Software Development Magazine: http://www.sdmagazine.com/documents/s=844/sdm0108a/0108a.htm

Fowler, M., & Lewis, J. (25 de Marzo de 2014). *Microservices a definition of this new architectural term.* Obtenido de https://martinfowler.com/articles/microservices.html

Gmeiner, J., Ramler, R., & Haslinger, J. (2015). Automated Testing in the Continuous Delivery Pipeline: A Case Study of an Online Company . *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops.* Graz, Austria: IEEE.

Gomede, & Barros, R. (2015). A Practical Approach to Software Continuous Delivery. *ICONS 2015 : The Tenth International Conference on Systems* (págs. 98-101). SEKE.

Graham, D., & Fewster, M. (1999). *Software Test Automation: Effective use of test execution tools.* ACM Press/Addison-Wesley.

Haselböck, S., Weinreich, R., & Buchgeher, G. (2017). Decision Models for Microservices: Design Areas, Stakeholders, Use Cases, and Requirements. *Lecture Notes in Computer Science, 10475* (págs. 155-170). Cham: Springer.

Haugset, B., & Hanssen, G. (2008). Automated Acceptance Testing: A Literature Review and an Industrial Case Study. *Agile, 2008. AGILE '08.* . Toronto, Canada: IEEE.

Hayes, L. (2004). *The Automated Testing Handbook.* Software Testing Inst; 2nd edition.

Humble, J. (13 de 08 de 2010). *Continuous Delivery vs Continuous Deployment.* Obtenido de https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/

Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison Wesley.

Jaramillo, D. (2016). Leveraging microservices architecture by using Docker technology. *SoutheastCon 2016.* Norfolk: IEEE.

Just, S., Herzig, K., Czerwonka, J., & Murphy, B. (2016). Switching to Git: the Good, the Bad, and the Ugly. *IEEE 27th International Symposium on Software Reliability Engineering* (págs. 400-411). Ottawa,Canada : IEEE.

Kaner, C. (1997). Improving the Maintainability of Automated Test Suites. *Software QA, 4(4).*

Kang, H., Le, M., & Tao, S. (2016). Container and Microservice Driven Design for Cloud Infrastructure DevOps. *IEEE International Conference on Cloud Engineering.* Berlín: IEEE computer society.

Karhu, K., Repo, T., Taipale, O., & Smolander, K. (2009). Empirical Observations on Software Testing Automation. *International Conference on Software Testing Verification and Validation* (págs. 201-209). Denver, USA: IEEE.

Koc, A., & Uz, A. (2015). *A Survey of Version Control Systems.* Obtenido de https://www.researchgate.net/publication/266866968_A_Survey_of_Version_Control_Systems

Koirala, S., & Sheikh, S. (2009). *Software Testing.* Jones & Bartlett Learning.

Kumar, K., & Kurhekar, M. (2016). Economically Efficient Virtualization Over Cloud Using Docker Containers. *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)* (págs. 95 - 100). Bangalore, India : IEEE.

Li, Z., Kihl, M., Lu, Q., & Andersson, J. A. (2017). Performance Overhead Comparison between Hypervisor and Container based Virtualization. *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)* (págs. 955 - 962). Taipei, Taiwan: IEEE.

Marick, B. (1997). *How to misuse code coverage*. Obtenido de Testing Foundations: http://www.exampler.com/testing-com/writings/coverage.pdf

Mårtensson, T., Ståhl, D., & Bosch, J. (2017). Continuous Integration Impediments in Large-Scale Industry Projects . *2017 IEEE International Conference on Software Architecture (ICSA)* (págs. 169 - 178). Gothenburg, Sweden: IEEE.

Meyer, S., Healy, P., Lynn, T., & Morrison, J. (2013). Quality Assurance for Open Source Software Configuration Management. *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on.* Timisoara, Romania : IEEE.

O'Connor, R., Elger, P., & Clarke, P. (2017). Continuous software engineering - A microservices architecture perspective. *Software: Evolution and process*, e1866.

Pahl, C. (2015). Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 24 - 31.

Paraiso, F., Challita, S., Al-Dhuraibi, Y., & Merle, P. (2016). Model-Driven Management of Docker Containers. *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)* (págs. 718 - 725). San Francisco, USA : IEEE.

Pulkkinen, V. (2013). Continuous Deployment of Software. *Cloud-Based Software Engineering - Proceedings of the seminar N°. 58312107*, (págs. 46-52). Helsinki, Finlandia.

Pulkkinen, V. (2016). Synthesizing Perceived Challenges in Continuous Delivery - A Systematic Literature Review. *Master Science Tesis*, 46 - 52. Helsinki, Finlandia.

Punjabi, R., & Bajaj, R. (2016). User Stories to User Reality: A DevOps Approach for the Cloud. *IEEE International Conference On Recent Trends In Electronics Information Communication Technology* (págs. 658 - 662). Bangalore, India: IEEE.

Rafi, D., Kiran, K., Petersen, K., & Mäntylä, M. (2012). Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey. *International Workshop on Automation of Software Test* (págs. 36-42). Zurich, Switzerland: IEEE.

Rai, P., Madhurima, Dhir, S., Madhulika, & Garg, A. (2015). *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)* (págs. 201-205). New Delhi, India: IEEE.

Randell, B., & Naur, P. (1969). Ingeniería de software: Reporte de una conferencia patrocinada por el comité científico de la OTAN. Garmisch, Alemania, 7 a 11 de octubre de 1968.

Richter, D., Konrad, M., Utecht, K., & Polze, A. (2017). Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice. *IEEE*

International Conference on Software Quality, Reliability and Security (Companion Volume) (págs. 130-137). Prague, Czech Republic : IEEE.

Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2016). The Evolution ofDistributed Systems Towards Microservices Architecture. *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, 318-325.

Shahin, M., Babar, A., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Software*.

Singh, S., & Singh, N. (2016). Containers & Docker: Emerging Roles & Future of Cloud Technology. *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology* (págs. 804 - 807). Bangalore, India: IEEE.

Sommerville, I. (2002). *Ingeniería de Software.* México: Pearson.

Soni , M. (2015). End to End Automation On Cloud with Build Pipeline: The case for DevOps in Insurance industry. *IEEE International Conference on Cloud Computing in Emerging Markets* (págs. 85-89). Gandhinagar, India: IEEE Computer Society.

Ståhl, D., & Bosch, J. (2014). Modeling continuous integration practice differences in industry software development. *The Journal of Systems and Software*, 48-59.

Stolberg, S. (2009). Enabling Agile Testing through Continuous Integration. *Agile Conference, AGILE '09.* Chicago, USA: IEEE.

Sultanía, A. (2015). Developing Software Product and Test Automation Software Using Agile Methodology. *Proceedings of the 2015 Third International Conference on Computer, Communication, Control and Information Technology (C3IT)* (págs. 1 - 4). Hooghly, India: IEEE.

The BOSS Index: Tracking the Explosive Growth of Open-Source Software. (07 de 04 de 2017). Obtenido de https://www.battery.com/powered/boss-index-tracking-explosive-growth-open-source-software/

Vassallo, C., Zampetti, F., Romano, D., Beller, M., Panichella, A., Di Penta, M., & Zaidman, A. (2016). Continuous Delivery Practices in a Large Financial Organization. *IEEE Software Maintenance and Evolution (ICSME).* Raleigh: IEEE computer society.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., & Casallas, R. (2015). Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. *2015 10th Computing Colombian Conference (10CCC)* (págs. 583-590). Bogota, Colombia : IEEE.

Virmani, M. (2015). Understanding DevOps & Bridging the gap from Continuous Integration to Continuous Delivery. *Fifth international conference on Innovative Computing Technology* (págs. 78-82). Pontevedra, Spain: IEEE.

Yu, Y., Silveira, H., & Sundaram, M. (2016). A Microservice Based Reference Architecture Model in the Context of Enterrise Architecture. *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)* (págs. 1856-1860). Xi'an, China: IEEE.

Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., & Di Penta, M. (2017). How Open Source Projects use Static Code Analysis Tools in Continuous Integration Pipelines. *14th International Conference on Mining Software Repositories (MSR)* (págs. 334-344). Buenos Aires, Argentina : IEEE.