FACULTAD DE INGENIERÍA Y CIENCIAS AGROPECUARIAS

INFRAESTRUCTURA COMO CÓDIGO: PRÁCTICAS DE LA INGENIERÍA DE SOFTWARE APLICADAS A LA GESTIÓN DE SERVIDORES VIRTUALIZADOS EN LA NUBE

Trabajo de Titulación presentado en conformidad con los requisitos establecidos para optar por el título de Ingeniera en Electrónica y Redes de Información

Profesor Guía

MBA Christian Aníbal Bastidas Romero

Autora

Luisa María Emme Bedoya

Año

2017

DECLARACIÓN DEL PROFESOR GUÍA

“Declaro haber dirigido este trabajo a través de reuniones periódicas con el estudiante, orientando sus conocimientos y competencias para un eficiente desarrollo del tema escogido y dando cumplimiento a todas las disposiciones vigentes que regulan los Trabajos de Titulación”.

Christian Aníbal Bastidas Romero
Máster en Dirección de Empresas
1710528546

DECLARACIÓN DEL PROFESOR CORRECTOR

“Declaro haber revisado este trabajo, dando cumplimiento a todas las disposiciones vigentes que regulan los Trabajos de Titulación”.

Santiago Ramiro Villarreal Narvaez
Máster en Ciencias
1713980074

DECLARACIÓN DE AUTORÍA DEL ESTUDIANTE

“Declaro que este trabajo es original, de mi autoría, que se han citado las fuentes correspondientes y que en su ejecución se respetaron las disposiciones legales que protegen los derechos de autor vigentes”.

Luisa María Emme Bedoya

1715789226

AGRADECIMIENTOS

Agradezco profundamente el apoyo, comprensión y las incontables enseñanzas de mi familia, quienes infundieron esa perseverancia clave para completar este capítulo de mi vida. En especial a mis padres; día a día luchando a mi lado y aportando todo cuanto podían. Son mi ejemplo de amor, constancia y superación.

A Edison, gracias por tu paciencia, apoyo y amor.

A mi hermana Natalia, aunque la vida haya decidido llevarte temprano, sé que caminas de mi lado y hoy celebro contigo.

DEDICATORIA

A mi sobrino Camilo Andrés. La felicidad es algo que debemos perseguir. Quiero verte grande, seguro, humilde, apasionado, pero sobre todo, feliz. Todo lo que te propongas conseguir, lo podrás tener. Nunca desfallezcas, siempre estaré a tu lado.

RESUMEN

En la actualidad, más equipos de TI están construyendo y gestionando su infraestructura descrita como código a través de herramientas automatizadas que permiten modelar sus elementos en código fuente de software. Esto, sumado al surgimiento de otras tecnologías como la virtualización y la nube, hace que los equipos se enfrenten tanto a un portafolio de servidores en constante crecimiento como a una naciente fuente de código cuya gestión y mantenimiento plantea retos adicionales.

La gestión de la infraestructura como código es muy diferente a la gestión de la infraestructura tradicional. Los enfoques de gestión del cambio tradicionales no logran hacer frente al ritmo del cambio que ofrece la automatización y la nube. El concepto *infraestructura como código* nace ante la necesidad de gestionar el cambio de manera más eficiente en un panorama de sistemas creados por la nube y las herramientas de automatización en constante expansión.

El presente proyecto de titulación, realiza una introducción a los principios y las prácticas de las metodologías ágiles en la Ingeniería de Software que se adaptan al concepto de infraestructura como código para ayudar a solventar dicha necesidad.

Finalmente, el caso de estudio muestra que existen empresas en el mercado local cuya experiencia en el uso de metodologías ágiles genera interés en la aplicación de éstas a la gestión de su infraestructura, motivando y facilitando su adopción.

ABSTRACT

Nowadays, more IT teams are building and managing their infrastructure described as code through automated tools that allow them to model their elements as software source code. This, along with the rise of other technologies such as virtualization and the cloud, make it so that teams must deal with an ever-increasing server portfolio as well as a growing source code whose management and maintenance outlines additional challenges.

The management of infrastructure as code is far more different than the management of traditional infrastructure. The foci of traditional change management cannot keep up with the change of pace that automation and the cloud offer. The concept of infrastructure as code is born out of the necessity of managing the changes in a more efficient way given an overview of systems created on the cloud and the constant expansion of automation tools.

This degree study realizes an introduction to the principles and practices of agile methodologies in Software Engineering that adapt to the concept of Infrastructure as Code to help solvent said necessity.

Finally, the study case shows that there exist companies in the local market whose experience in the use of agile methodologies generate interest in their application to the management of their infrastructure, motivating and facilitating their adoption.

ÍNDICE

1. Capítulo I. Introducción 1
1.1. Antecedentes 1
1.2. Alcance 2
1.3. Justificación 3
1.4. Objetivos 4
1.4.1. Objetivo general 4
1.4.2. Objetivos específicos 4
2. Capítulo II. Desafíos y prácticas comunes en la gestión de la infraestructura dinámica 5
2.1. Proliferación de servidores 5
2.2. Desviación de la configuración 6
2.3. Servidores “copo de nieve” 7
2.4. Miedo a la automatización 7
2.5. Infraestructura frágil 8
2.6. Erosión 9
3. Capítulo III. Prácticas de la Ingeniería de Software aplicadas a la gestión de servidores 9
3.1. Introducción 9
3.2. Calidad del sistema 10
3.2.1. Desarrollo guiado por pruebas 11
3.2.2. Refactorización 12
3.2.3. Programación en pares 13
3.2.4. Código limpio 14
3.3. SCV para la gestión de la infraestructura 14
3.4. Integración continua 15
3.5. Entrega continua 17

3.6. Gestión de grandes cambios en la infraestructura 18
4. Capítulo IV. Infraestructura como código 20
4.1. Principios 20
4.1.1. Infraestructura reproducible 20
4.1.2. Infraestructura descartable 21
4.1.3. Consistencia 22
4.1.4. Procesos repetibles 23
4.1.5. Diseño evolutivo 24
4.2. Prácticas 25
4.2.1. Archivos de definición de la infraestructura 25
4.2.2. Sistemas y procesos auto documentados 25
4.2.3. Versionamiento 26
4.2.4. Sistemas y procesos probados de manera continua 27
4.2.5. Cambios pequeños 28
4.2.6. Continuidad del servicio 28
5. Capítulo V. Consideraciones de implementación desde el punto de vista organizacional 29
5.1. Acordar sobre los resultados esperados 29
5.2. Escoger métricas que ayuden al equipo 30
5.3. Utilizar una herramienta como Kanban para aumentar la visibilidad del trabajo 31
5.4. Organizar a los equipos para empoderar a los usuarios 32
5.5. Gobernabilidad a través de la gestión del cambio continuo 34
6. Capítulo VI. Caso de estudio: empresa Denarius 35
6.1. Antecedentes 35
6.2. Análisis del caso 36
6.3. Próximos pasos 42
7. Conclusiones y recomendaciones 44

7.1. Conclusiones 44
7.2. Recomendaciones 45

REFERENCIAS 46

ANEXOS 49

ÍNDICE DE FIGURAS

Figura 1. Espiral del miedo a la automatización ……..…… 8
Figura 2. Proceso de Integración continua ……..…… 17
Figura 3. Cómo construir un Mínimo Producto Viable ……... 19
Figura 4. Ejemplo de un Tablero Kanban ……..... 31
Figura 5. Flujo de valor con traspasos entre equipos ……..... 33
Figura 6. Diagrama de la plataforma de Denarius con Chef, Git y Azure …….. 37
Figura 7. Captura de pantalla de la herramienta Visual Studio Code en Denarius ……..…… 38
Figura 8. Repositorio Git privado para el código de la infraestructura en Visual Studio Online ……..…… 38
Figura 9. Componentes de Chef ……..……..…… 39
Figura 10. Organizaciones creadas en el servidor Chef …… 40
Figura 11. Cookbooks (políticas) registradas desde la máquina de desarrollo en el servidor Chef ……..……...…….. 40
Figura 12. Visualización del estado de arranque de los nodos clientes en el servidor Chef ……..…… 41
Figura 13. Ejemplo de algunos reportes disponibles sobre el estado de los nodos clientes en el servidor Chef ……..…… 41
Figura 14. Instalación del cliente Chef en un servidor de Denarius ……..……. 42

ÍNDICE DE TABLAS

Tabla 1. Técnicas para el despliegue de sistemas con funcionalidades incompletas 19
Tabla 2. Relación funcionalidad-valor del SCV en la Infraestructura como código 26
Tabla 3. Resumen de métricas comunes utilizadas por equipos de infraestructura 30
Tabla 4. Problemas comunes en la división de equipos por funciones 32
Tabla 5. Elementos clave de un proceso de gestión del cambio efectivo 34

1. Capítulo I. Introducción

1.1. Antecedentes

La evolución de las prácticas en la Ingeniería de Software, gracias al surgimiento de las metodologías ágiles en la década de los 70, está enfocada en abordar los problemas frecuentes en el desarrollo de soluciones como principales causas del fracaso de los proyectos y desencadenadores de grandes impactos humanos y económicos (Beck, 2001). Según los resultados del reporte CHAOS en 2015 (Standish Group, 2015), en el cual se estudiaron 50,000 proyectos de software (desde pequeñas mejoras hasta implementaciones de sistemas masivos con re-ingeniería) alrededor del mundo, se evidencia que, en 2015, sólo el 29% de los proyectos llevados a cabo resultaron exitosos mientras que el 71% restante se dividió entre proyectos tanto desafiantes como fallidos. El éxito estuvo definido en base a la Resolución Moderna que evalúa si el proyecto terminó a tiempo, dentro del presupuesto planificado y con un resultado satisfactorio.

Con la asimilación de los métodos ágiles de desarrollo de software en los últimos años, fue posible comparar estos resultados entre los proyectos tradicionales en cascada y los proyectos ágiles. El 39% de los proyectos (grandes, medianos y pequeños) llevados a cabo utilizando metodologías ágiles resultaron exitosos en contraste con el 11% de proyectos gestionados con metodologías tradicionales en cascada (Standish Group, 2015). Estas cifras demuestran el éxito alcanzado a partir de la adopción de prácticas ágiles en los equipos de desarrollo, entre ellas el desarrollo guiado por pruebas, la integración continua y el despliegue continuo (Morris, 2016b).

El problema básico en la Ingeniería de Software es el riesgo. Algunos ejemplos de riesgo pueden ser: ampliaciones en el plazo de entrega, cancelación del proyecto, el sistema no puede ser mantenido a largo plazo, alta tasa de defectos, malinterpretación de las necesidades del negocio, cambios en el negocio, características del sistema inutilizadas, rotación del personal de TI, etc. Las metodologías ágiles minimizan el riesgo mediante la reducción de los ciclos de entrega del producto, el reconocimiento de que la incertidumbre es inherente al

desarrollo de software, el enfoque en un mínimo producto viable que potencia el valor del software entregado y reduce los puntos de fallo en la entrega, la creación y el mantenimiento de una suite de pruebas automáticas y comprensivas que garantizan la calidad de cada entrega, el trabajo colaborativo y cercano con el cliente como parte integral del equipo, el desarrollo de funcionalidades que entregan el máximo valor en cada iteración de trabajo, entre otros (Beck, 2001, págs. 3-4). Estas y otro tipo de prácticas aplicadas a la Ingeniería de Software han tenido éxito al abordar y atacar las principales causas de fracaso en los proyectos mediante la implementación de distintas técnicas y herramientas que han probado ser de gran ayuda durante el ciclo de vida de un proyecto.

Entre los principios relacionados al desarrollo de software se encuentran la reproducibilidad de los sistemas, la consistencia, repetibilidad, diseño evolutivo y emergente, la auto evaluación y auto documentación, los cambios pequeños y el versionamiento de los sistemas. Estos principios son aplicables al software debido a su naturaleza maleable. La evolución y el crecimiento de una solución de software puede ser gradual, paulatina y basada en la retroalimentación inmediata de la interacción del usuario final con el sistema en producción (Ford, 2011).

Hoy en día, el surgimiento de las herramientas de automatización de la infraestructura, la virtualización y la nube permiten la definición de los elementos de infraestructura en forma de software. Es posible adaptar los principios y prácticas que han dado grandes resultados en la Ingeniería de Software y aplicarlos en la gestión de servidores virtualizados en la nube (Morris, 2016b).

1.2. Alcance

El presente proyecto de investigación tiene como alcance la descripción de las prácticas de desarrollo de software y su adaptación a la infraestructura como código, enfocándose en la gestión de servidores virtualizados en la nube.

Como parte del proyecto se incluyen algunos de los desafíos que enfrenta la gestión de la infraestructura dinámica, abarcando: alteraciones en la configuración, servidores frágiles, infraestructura inestable, miedo a la automatización y erosión. Además, se explican las prácticas comunes en los equipos de TI, desencadenadoras de estos desafíos y problemas, tales como: tareas rutinarias y scripts manuales, aplicación de cambios directamente en sistemas importantes, automatización que corre infrecuentemente y desviación de la automatización.

El núcleo de la investigación desarrolla los principios de la infraestructura como código, más específicamente: infraestructura reproducible (reproducibility), consistencia, acciones repetibles (repeatability), infraestructura descartable (disposability), diseño evolutivo, continuidad del servicio, sistemas auto probados, sistemas auto documentados, cambios pequeños y versionamiento. Se incluye la introducción a las prácticas de la Ingeniería de Software que se aplican a la infraestructura, a saber: calidad del sistema, integración continua, sistemas de control de versiones, calidad del código, gestión de grandes cambios. Para finalizar, se detalla la forma en la que dichas prácticas se adaptan a la infraestructura como código para solucionar los problemas y sobrellevar los desafíos presentados al inicio.

1.3. Justificación

Mientras que las prácticas en el desarrollo del software han evolucionado y cambiado ampliamente durante los últimos años, las prácticas de operación en TI no han avanzado al mismo ritmo. Muchos equipos de TI todavía se basan en una combinación de configuraciones manuales, scripts personalizados, imágenes maestras o herramientas obsoletas para administrar la infraestructura. El resultado, son despliegues de ambientes cada vez más lentos y con demasiados errores (Morris, 2014).

La virtualización y la nube han forzado la necesidad de algún tipo de automatización. La capacidad de levantar nuevas máquinas virtuales en minutos desencadenó la automatización en el proceso de configuración de los servidores

al mismo tiempo que la necesidad de mantener al día un número de servidores en constante crecimiento y cambio evitando la desviación de configuraciones, dio lugar a nuevas herramientas. CFengine, Puppet y Chef establecieron una nueva categoría de herramientas para la automatización de la infraestructura, rápidamente adoptada por organizaciones que estaban aprovechando al máximo las ventajas de la IaaS en la nube desde su surgimiento (Morris, 2014).

La diferencia entre la automatización de la infraestructura y la infraestructura como código es que, la primera posibilita llevar a cabo acciones de manera repetida a través de un largo número de nodos mientras que, la segunda utiliza técnicas, prácticas y herramientas del desarrollo de software para asegurar que esas acciones son probadas a fondo antes de ser aplicadas a los sistemas de alta criticidad para el negocio (Morris, 2014).

La esencia de la Infraestructura como Código es tratar la configuración de sistemas de la misma manera como es tratado el código fuente del software. Los sistemas de gestión de código fuente, el desarrollo guiado por pruebas, la integración continua, la refactorización y otras prácticas de XP son especialmente útiles para asegurar que los cambios en la infraestructura son probados a fondo, repetibles y transparentes (Morris, 2014).

1.4. Objetivos

1.4.1. Objetivo general

Describir y documentar las prácticas del desarrollo de software adaptadas a la gestión de la infraestructura para que distintos equipos de TI puedan adoptarlas minimizando los desafíos a los que se enfrentan en sus tareas cotidianas.

1.4.2. Objetivos específicos

- Listar los desafíos comunes que enfrentan los equipos de TI en la gestión de la infraestructura dinámica.
- Examinar las prácticas en los equipos de TI que desencadenan dichos desafíos, así como las consecuencias de cada una.

- Resumir las prácticas de la Ingeniería de Software aplicadas a la gestión de servidores virtualizados en la nube.
- Explicar los principios de la Infraestructura como Código.
- Presentar un caso de estudio de la aplicación de dichas prácticas.

2. Capítulo II. Desafíos y prácticas comunes en la gestión de la infraestructura dinámica

2.1. Proliferación de servidores

Con la aparición de la virtualización y la nube, la creación de nuevos servidores a partir de un conjunto de recursos y el aprovisionamiento de los mismos se ha convertido en una tarea fácil y rápida. Hoy en día, es posible crear y aprovisionar máquinas en cuestión de minutos (Morris, 2016b, pág. Kindle Location 274). Herramientas como Vagrant, Chef, Ansible, Puppet, entre otras, y proveedores en la nube como Amazon EC2 potencian el crecimiento de servidores e infraestructura virtuales.

Este beneficio puede tornarse contraproducente si el número de servidores y sistemas llega a ser mayor que la capacidad que tienen los equipos de operaciones para su gestión (Morris, 2016b, pág. Kindle Location 276).

Las actualizaciones de software y la detección de problemas sobre una infraestructura en crecimiento traen consigo varios desafíos. Con el escalamiento en el número de servidores gestionados, las actualizaciones de software de manera consistente sobre toda la infraestructura se tornan difíciles y engorrosas. Si los equipos de operaciones de TI no pueden mantener al día las actualizaciones sobre todos sus sistemas, estos pueden quedar vulnerables a ataques comunes.

Así mismo, cuando un problema es detectado y la cantidad de tiempo necesaria para el despliegue de la solución sobre todos los sistemas supera el tiempo disponible del equipo, aparecen diferencias en versiones y configuraciones entre servidores del mismo tipo (desviación de la configuración). Esto lleva a tener

software y scripts que funcionan sobre ciertos sistemas, pero no funcionan sobre otros.

2.2. Desviación de la configuración

La desviación de la configuración es la causa directa de la inconsistencia entre servidores y hace referencia a las diferencias que pueden aparecer en el tiempo entre servidores aun cuando éstos hayan sido creados y configurados inicialmente de manera consistente (Morris, 2016b, págs. Kindle Locations 281-282).

Algunos ejemplos de las diferencias que aparecen en el tiempo entre servidores:

- Alguien hace un arreglo sobre uno de los servidores Oracle. Este servidor ahora es diferente de los demás servidores Oracle en la red (Morris, 2016b, págs. Kindle Locations 282-283).
- La nueva versión de una aplicación necesita una nueva versión de Java. El equipo de operaciones no tiene tiempo para probar todas las otras aplicaciones que se basan en Java con la nueva versión. Los servidores que contienen a las otras aplicaciones permanecen con la versión antigua de Java.
- Tres personas distintas configuran una aplicación en tres servidores. Cada persona configura la aplicación de manera diferente, por tanto, los servidores ahora difieren entre sí en su configuración.
- Un servidor JBoss recibe más tráfico que los otros. Una persona del equipo, lo adapta para que pueda soportar el exceso de tráfico y ahora su configuración es diferente a los demás servidores JBoss en la red (Morris, 2016b, págs. Kindle Locations 285-286).

Tal como lo dice Morris (2016b, pág. Kindle Location 286), *“ser diferente no es algo malo”.* El problema con la desviación de la configuración se presenta cuando las diferencias no pueden ser capturadas y manejadas por el equipo, de manera que se vuelve difícil y en ocasiones hasta imposible reconstruir servidores y servicios.

2.3. Servidores “copo de nieve”

Se conoce como servidor “copo de nieve” a todo aquel servidor que es único y distinto de los demás servidores en la red. Este tipo de servidores son especiales en maneras que no pueden ser reproducidas.

Existen muchas organizaciones que poseen este servidor que “*no puede ser tocado”*. Nadie está seguro de las funciones que cumple, pero se encuentra allí por alguna razón, de manera que ninguna persona se atreve a tocarlo, mucho menos a desconectarlo.

Citando nuevamente a Morris (2016b, pág. Kindle Location 286), *“ser diferente no es algo malo”*. El problema con el servidor copo de nieve nace cuando el equipo responsable por él, no entiende cómo ni por qué el servidor es diferente y por tanto es incapaz de reproducirlo.

2.4. Miedo a la automatización

En la actualidad, existe una serie de herramientas enfocadas en la automatización de la infraestructura entre las que se incluyen: Chef, Puppet, y Ansible. Estas herramientas buscan ahorrar tiempo a los administradores de TI haciéndose cargo de tareas rutinarias como la configuración, ajustes y gestión de la infraestructura de manera automatizada. Sin embargo, el uso incorrecto de dichas herramientas puede llevar a los equipos a desconfiar de ellas entrando en la espiral del miedo a la automatización (Figura 1) como lo denomina Morris (2016b, pág. Kindle Location 332).

Las herramientas de automatización, entre ellas, las mencionadas anteriormente, describen la infraestructura en forma de código. La habilidad de describir hardware en forma de software debe venir acompañada de una correcta gestión del código generado. La inclusión de sistemas de control de versiones, pruebas unitarias, pruebas de menor granularidad y flujos de proceso automatizados también conocidos como pipelines, son claves para garantizar la confiabilidad, la calidad y la gestión de cambios en el tiempo del código de la infraestructura de manera automatizada.

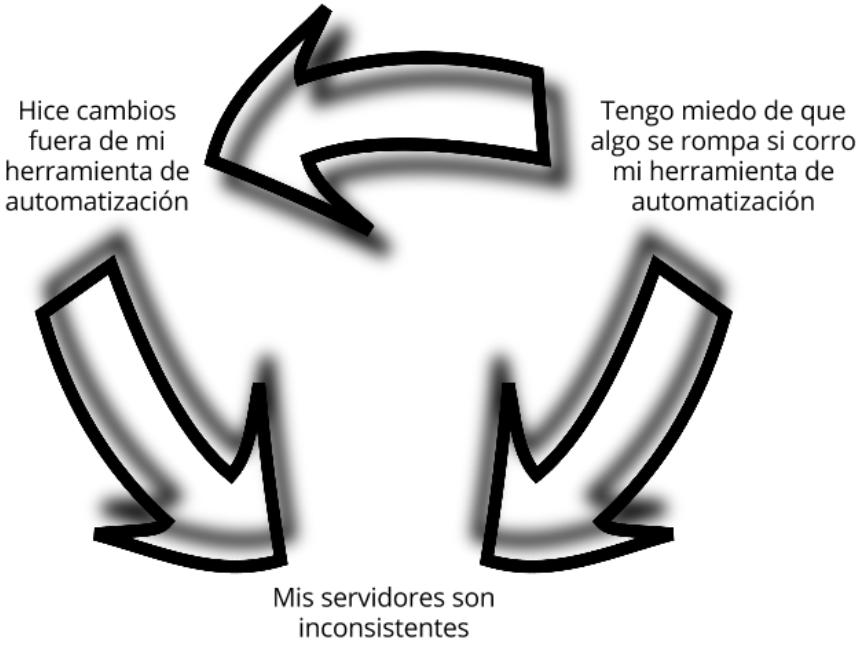

Figura 1. Espiral del miedo a la automatización.
Adaptado de Morris (2016b, pág. Kindle Location 330).

Cuando existe una ausencia o bajo uso de prácticas y técnicas que garanticen el funcionamiento del código generado desde los ambientes de prueba hasta los ambientes productivos, los equipos de desarrollo evitan la automatización por la falta de confianza que tienen en sus herramientas y procesos.

Una de las sugerencias de Morris (2016b, pág. Kindle Location 315) para romper la espiral en los equipos que sufren de este llamado “miedo a la automatización” es empezar con un grupo pequeño de servidores, cambiar las definiciones de configuración en la herramienta de automatización, programarlas para que corran desatendidas y repetir el proceso con otro grupo de servidores, hasta que toda la infraestructura esté constantemente actualizada.

2.5. Infraestructura frágil

Una infraestructura frágil es aquella poco resiliente a fallos, por tanto, es fácil de romper y difícil de reparar. Puede pensarse como el escalamiento del servidor copo de nieve a un portafolio completo de servidores y sistemas.

El trabajo de los equipos que gestionan este tipo de infraestructura es puramente reactivo. La solución proactiva conlleva migrar uno a uno todos los sistemas y servidores hacia una infraestructura confiable que pueda ser reproducida.

2.6. Erosión

Los problemas en un sistema en funcionamiento aparecerán con el tiempo ya sea porque la propia infraestructura sufre deterioro con los años, o porque se van descubriendo fallas y vulnerabilidades en el software a través de la retroalimentación que genera la interacción del sistema con el usuario final. A esta idea se la conoce como erosión.

Algunos ejemplos de erosión incluyen:

- Deterioro del disco duro.
- Disco duro lleno de archivos de registro.
- Uno o más procesos en el sistema se detienen o se quedan suspendidos.
- Deterioro del hardware: memoria, tarjetas de red, etc.

3. Capítulo III. Prácticas de la ingeniería de software aplicadas a la gestión de servidores

3.1. Introducción

La infraestructura como código se basa en la idea de que los sistemas y dispositivos sobre los que se ejecuta cualquier software pueden ser tratados como software. Esto permite que las prácticas, que han probado ser exitosas en el desarrollo de software, sean aplicadas y utilizadas para la gestión de la infraestructura.

La construcción de la calidad en el sistema es el núcleo de las prácticas de software que se detallan en este capítulo. La calidad ágil, que enfoca la calidad como una parte integral en la forma de planificar, diseñar, implementar y entregar sistemas, se fundamenta en los Principios del Software Ágil que pueden ser aplicados tanto al código en el desarrollo de aplicaciones como al código que

describe la infraestructura. Entre los Principios del Software Ágil, que Ward Cunningham (2001), uno de los autores del Manifiesto Ágil, describe en su sitio web y que se aplican a la infraestructura, se encuentran los siguientes:

- Empezar a entregar temprano código útil y que funciona.
- Continuar con la entrega de manera incremental, en iteraciones pequeñas y útiles.
- Construir solamente lo necesario en ese momento.
- Construir cada incremento tan simple como sea posible.
- Asegurarse de que cada cambio esté bien diseñado e implementado.
- Obtener retroalimentación de cada cambio tan temprano como sea posible.
- Se espera que los requerimientos cambien conforme el equipo y los usuarios aprendan del sistema.
- Asumir que todo lo que se entrega tendrá que cambiar a medida que el sistema evoluciona.

3.2. Calidad del sistema

De manera habitual, la calidad es vista como un tema de exactitud funcional, es decir, si el sistema funciona bien, su calidad es alta. Sin embargo, la calidad en los sistemas es, principalmente, un agente que habilita el cambio (Morris, 2016b).

En sistemas con baja calidad un cambio simple puede tomar mucho tiempo y, además, causar más problemas de los esperados. Un sistema con alta calidad tiene, generalmente, un claro impacto del cambio. Las herramientas de automatización y los conjuntos de prueba agregan visibilidad rápida y temprana de cualquier problema. Prácticas como la Integración Continua y la Entrega Continua se enfocan en la reducción del tiempo que existe al introducir un cambio y ser notificado de los problemas introducidos con dicho cambio en ambientes pre productivos, de manera que sea más fácil para el equipo encontrar la causa de los problemas minimizando el riesgo en sistemas de alta criticidad. La necesidad de documentación técnica se reduce gracias a la facilidad de

entendimiento del sistema a través de sus pruebas y su propio código, simplificando así la inducción de nuevos miembros en el equipo.

Tal y como Morris (2016b, págs. Kindle Locations 3676-3677) señala “definir un sistema en términos de infraestructura como código y construirlo con herramientas no mejora su calidad”. Un conjunto de archivos de definición y herramientas inconsistentes y con poco mantenimiento sumados a una serie de cambios manuales ad hoc y casos especiales da como resultado una infraestructura frágil en la cual, correr la herramienta incorrecta, puede llegar a causar daños catastróficos (Morris, 2016b, págs. Kindle Locations 3678-3679).

La idea de la infraestructura como código es mover el enfoque de la calidad a los sistemas de definición y a las herramientas. Prácticas como programación en pares, TDD, código limpio y refactorización son claves para tener las virtudes de un código de calidad: fácil de entender, fácil de cambiar y con retroalimentación temprana de los problemas. A mayor calidad en las definiciones y herramientas utilizadas para construir y cambiar la infraestructura, mayor calidad, confiabilidad y estabilidad tendrá la infraestructura generada (Morris, 2016b, págs. Kindle Locations 3681-3682).

3.2.1. Desarrollo guiado por pruebas (TDD)

“Producimos código bien diseñado, probado y factorizado en pasos pequeños y verificables” (Shore & Warden, 2008).

El Desarrollo guiado por pruebas, también conocido como TDD por sus siglas en inglés, es una técnica para la construcción de software que guía el desarrollo a través de pruebas. Representa un ciclo con tres pasos principales:

- Escribir una prueba para la siguiente funcionalidad que se desea añadir.
- Escribir el mínimo código necesario para que la prueba pase.
- Aplicar refactorización para restructurar tanto el código de las pruebas como el código de producción.

La técnica del TDD fue introducida a finales de 1990 por Kent Beck, autor de la metodología ágil XP y, desde entonces, ha probado tener un gran impacto positivo sobre la calidad del código y la detección temprana de defectos.

Usado de forma correcta, TDD incluso ayuda en la mejora del diseño del software, sirve como documentación viva del código y de las interfaces públicas en una aplicación y previene futuros errores (Shore & Warden, 2008).

3.2.2. Refactorización

“Cada día, nuestro código es ligeramente mejor de lo que fue el día anterior” (Shore & Warden, 2008).

La refactorización es una técnica disciplinada que busca restructurar una parte de código existente, alterando su estructura interna sin cambiar su comportamiento externo (Fowler, 2015).

Gracias a la refactorización, es posible transformar un mal código (código “sucio”, que no cumple con buenos principios de diseño, etc.) en código bien estructurado y diseñado sin introducir errores en el funcionamiento del software.

Esta técnica se vale de “malos olores en el código” (code smells) para la identificación de posibles puntos en el código en donde la necesidad de refactorización se hace evidente.

Además de ser una poderosa herramienta para trabajar sobre código existente, la refactorización también es un mecanismo que habilita el Diseño Emergente, concepto introducido por Neil Ford en su libro Agile Engineering Practices. Antes de implementar una nueva característica en el sistema, debe hacerse la siguiente pregunta: ¿se puede cambiar el diseño actual del sistema para facilitar la introducción de la nueva característica?. Después de implementar la característica es necesario cuestionarse si es factible hacer el sistema aún más sencillo (Beck, 2001).

3.2.3. Programación en pares

“Todo el código de producción es escrito por parejas de desarrolladores que trabajan juntos en la misma máquina” (Martin, 2003, pág. 13).

La programación en pares es una forma de trabajo mediante la cual se busca incrementar la calidad del código y reducir significativamente los defectos. La dinámica se basa en una pareja de desarrolladores que siguen las siguientes reglas:

- Un miembro de cada pareja toma el control sobre la computadora y escribe el código necesario (o una prueba unitaria en el caso en el que se combine la programación en parejas con el desarrollo guiado por pruebas).
- El otro miembro, mira el código que está siendo escrito, corrigiendo posibles errores y sugiriendo mejoras.
- Los roles rotan varias veces cada hora.

Existen distintas técnicas que pueden ser utilizadas para la programación en parejas, entre ellas se encuentran:

- Ping-Pong: Define una rotación basada en la escritura de pruebas y la escritura del código de producción. Un miembro de la pareja escribe una prueba que falla. El otro miembro escribe el código mínimo necesario para hacer pasar dicha prueba, refactoriza y escribe una nueva prueba que falla, regresando el teclado al primer desarrollador. La actividad se repite varias veces hasta completar la funcionalidad deseada.
- Conductor-Navegador: Un miembro de la pareja empieza como Conductor tomando el control del teclado. El otro miembro actúa como Navegador, guiando al conductor sobre la solución y removiendo obstáculos. Los roles cambian cuando así la pareja lo decida, o cuando el conductor haya llegado a un obstáculo significativo que le impida seguir desarrollando.

Existen estudios que demuestran que la programación en parejas no reduce la eficiencia en el desarrollo; sin embargo, reduce significativamente la tasa de defectos en el código (Martin, 2003, pág. 13).

3.2.4. Código limpio

La idea detrás del código limpio es aprender a diferenciar entre un buen código y un mal código y aprovechar las ventajas que un buen código puede brindar al equipo. Es importante saber identificar esos “malos olores en el código” que guiarán una refactorización posterior. En su libro, Robert C. Martin (Clean Code, 2011) vincula fuertemente el profesionalismo del desarrollador/desarrolladora con la generación de código limpio. De allí la importancia de listar este tema como uno de los agentes en la construcción de la calidad en los sistemas.

Un mal código puede inutilizar un sistema por completo. Como lo dice Robert C. Martin (Clean Code, 2011), un código que no sigue principios básicos de limpieza en sí mismo, puede ralentizar el trabajo de todo aquel que requiera trabajar sobre él. Y aún más, un mal código introduce defectos y genera miedo al cambio ya que, un cambio en una parte del código puede romper otras partes del sistema que no se tenían previstas.

3.3. SCV para la gestión de la infraestructura

Un sistema de control de versiones (SCV) es aquel capaz de registrar cambios realizados en el tiempo a uno o varios archivos permitiendo regresar a cualquier versión específica de los mismos en cualquier momento (Somasundaram, 2013). Algunos SCV proveen, entre otras cosas, lo siguiente:

- Trazabilidad de los cambios, permitiendo conocer el histórico de modificaciones sobre un archivo o un conjunto de archivos. Entre otras cosas es posible conocer quién realizó los cambios y cuándo fueron efectuados.
- Reversión de cambios, facilitando el restablecimiento de archivos a cualquier versión previa gracias a la capacidad de estos sistemas de rastrear y mantener cambios en el tiempo.

- Acciones gatilladas de manera automatizada. De esta forma, es posible, por ejemplo, gatillar la ejecución de distintos conjuntos de prueba cada vez que se detecta un cambio en uno o más archivos (Morris, 2016b, pág. Kindle Location 3692).

Todo lo necesario para construir y reconstruir los elementos de la infraestructura debe ser gestionado y registrado con un sistema de control de versiones. Algunos ejemplos de lo que puede ser manejado por un SCV incluyen: definiciones de configuración, archivos y plantillas de configuración, código de pruebas, definiciones de las tareas de CI y CD, scripts utilitarios, código fuente de utilitarios y aplicaciones, documentación, etc. (Morris, 2016b, pág. Kindle Location 3692)

Sin embargo, existen ciertos elementos que no deberían ser gestionados por un SCV por distintas razones, ya sea porque existen repositorios específicos en los que deberían ser almacenados o porque vulneran principios de seguridad. Entre ellos se incluyen: artefactos de software, información y datos gestionados por aplicaciones, archivos de registro, contraseñas y otros secretos (Morris, 2016b, pág. Kindle Location 3700).

3.4. Integración continua

Integración continua o CI por sus siglas en inglés, es una práctica que se enfoca en la integración y la validación de código a nivel de pruebas unitarias y pruebas de integración (Fowler, 2006). Para alcanzar su objetivo, CI incluye lo siguiente:

- Los desarrolladores registran su código en el sistema de gestión de código fuente al menos una vez al día.
- El código es compilado y construido por el servidor de construcciones de código fuente automáticamente en cada registro.
- Las pruebas unitarias se ejecutan automáticamente por cada compilación y construcción exitosa.
- La construcción del código es rápida, por tanto, el equipo de desarrollo obtiene retroalimentación rápida.

- Las pruebas son ejecutadas en una versión reducida del entorno de producción.
- Los artefactos o entregables se almacenan en un repositorio de artefactos con versionamiento.
- Los artefactos o entregables se despliegan automáticamente en un ambiente de pruebas después de cada compilación, construcción y ejecución exitosa de las pruebas unitarias (Versteijnen, 2015).

CI es una práctica que aporta múltiples beneficios al equipo de desarrollo, entre los que se encuentran: la integración de código se realiza de manera constante y frecuente durante el proceso de desarrollo en lugar de ser un proceso aislado posterior, reducción del riesgo, habilita despliegues más frecuentes y la retroalimentación sobre los cambios realizados es rápida y, por tanto, la resolución de problemas se simplifica (Morris, 2016b, pág. Kindle Location 3719).

Para alcanzar un nivel maduro y efectivo de integración continua, además de las ya mencionadas anteriormente, es necesario tener en cuenta las siguientes consideraciones:

- Cualquier fallo en la construcción del código o en la ejecución de las pruebas genera una construcción rota o un flujo de trabajo en rojo.
- Nadie en el equipo puede registrar cambios en el sistema de gestión de código fuente si la construcción está rota.
- El arreglo del flujo de trabajo en rojo es priorizado por la o las personas que introdujeron los errores en el mismo, antes tomar una nueva tarea.
- El equipo es notificado cada vez que la construcción se rompe a través de alertas generadas por herramientas como monitores o radiadores de construcción.
- Una prueba que falla debe ser arreglada y nunca ignorada (Morris, 2016b, págs. Kindle Locations 3748-3766).

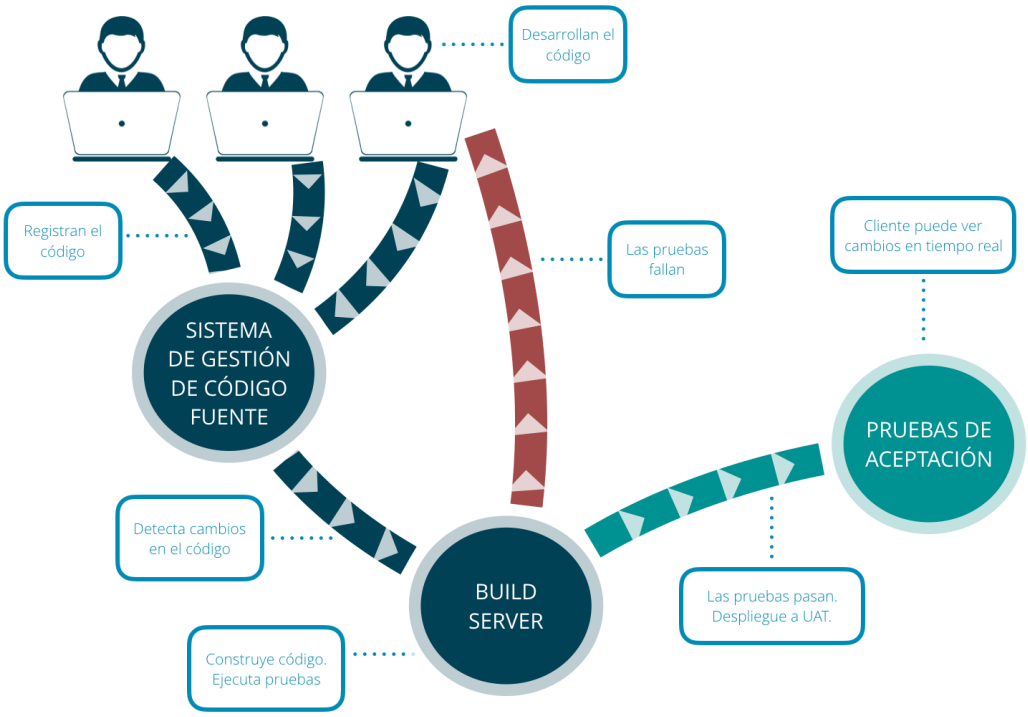

Figura 2. Proceso de Integración continua.
Adaptado de 1minus1 Limited (1minus1, s.f.).

La práctica de integración continua en infraestructura se traduce en probar continuamente los cambios generados sobre archivos de definición, scripts y otras herramientas y configuraciones utilizadas para la construcción y reconstrucción de los elementos de la infraestructura. Todos estos archivos son gestionados por un SCV evitando ramificaciones, pruebas ignoradas y registros sobre una construcción rota (Morris, 2016b, pág. Kindle Location 3779).

3.5. Entrega continua

La entrega continua (CD) es el siguiente paso después de la integración continua (Versteijnen, 2015). Se basa en extender el alcance de la integración continua a todo el sistema, certificando cada cambio a través de pruebas automatizadas sobre distintos ambientes y, asegurando que todos los componentes desplegables, sistemas e infraestructura son continuamente validados para garantizar que están listos y pueden ser aplicados en producción (Morris, 2016b, pág. Kindle Location 3843).

La implementación de CD es realizada a través de flujos de despliegue (deployment pipelines) para el software y flujos de cambios (change pipelines) para la infraestructura. Ambos manifiestan de manera automatizada el proceso de despliegue: construcción de código, despliegue en distintos ambientes pre productivos y ejecución de pruebas (Morris, 2016b, pág. Kindle Location 3824).

Es importante notar que la entrega continua no es lo mismo que el despliegue continuo. La primera, se enfoca en asegurar que cada cambio está listo para ir a producción, delegando al negocio la decisión de cuándo aplicarlo y automatizando el proceso de despliegue una vez que la decisión está tomada. La segunda, se enfoca en aplicar en producción de manera automática e inmediata cada cambio registrado, después de haber superado exitosamente la ejecución del conjunto de pruebas automatizadas en los distintos ambientes pre productivos (Morris, 2016b, pág. Kindle Location 3846).

3.6. Gestión de grandes cambios en la infraestructura

Las prácticas de ingeniería mencionadas hasta este momento se basan en el principio del desarrollo de pequeños cambios a la vez. Sin embargo, esto no es posible cuando se quiere entregar cambios grandes y potencialmente disruptivos. Un ejemplo de esto sería el reemplazo de un Servicio de Directorio. Podría llegar a tomar semanas o incluso meses tener el nuevo servicio funcionando y completamente probado y, además, el cambio del antiguo servicio por el nuevo podría llegar a causar serios problemas (Morris, 2016b, págs. Kindle Locations 3886-3889).

La clave en la entrega de trabajo complejo de una manera ágil está en romper las funcionalidades complejas en partes más pequeñas (Morris, 2016b, págs. Kindle Locations 3890-3891) que cumplan con el concepto de MVP.

Lo más importante es asegurar que cualquier cambio que tome tiempo en ser implementado, esté siendo continuamente probado durante el desarrollo. Para esto, existen distintas estrategias que permiten construir incrementalmente grandes cambios en sistemas que ya se encuentran en producción. La Tabla 1

resume algunas de las técnicas que existen para desplegar sistemas en producción con funcionalidades incompletas y que a su vez habilitan la validación continua de cada cambio durante su desarrollo (Morris, 2016b, págs. Kindle Locations 3908-3909).

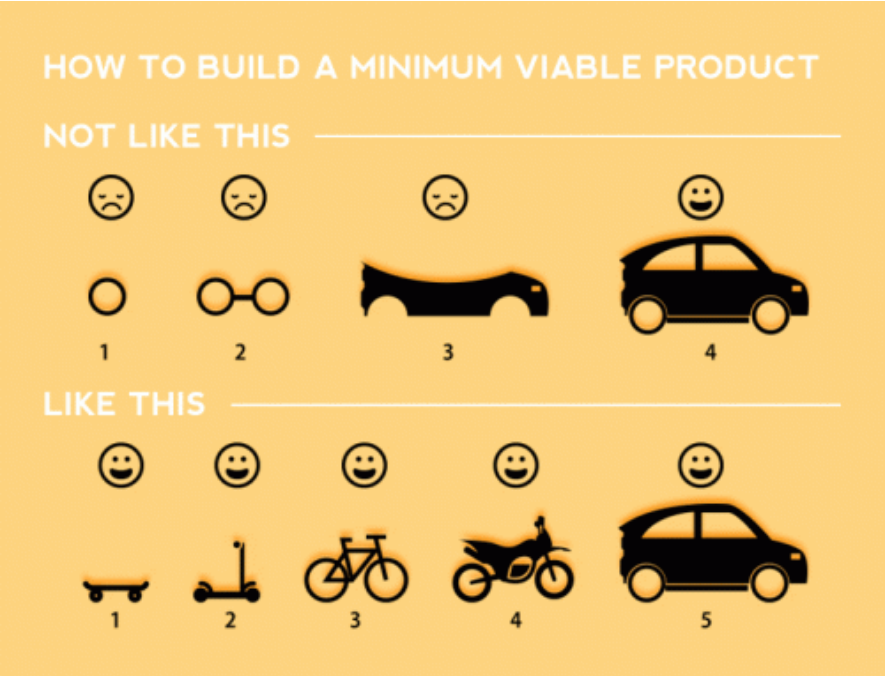

Figura 3. Cómo construir un Mínimo Producto Viable.
Tomado de Fast Monkeys (Fast Monkeys, 2014).

Tabla 1.
Técnicas para el despliegue de sistemas con funcionalidades incompletas.

Técnica	***Descripción***
Ocultamiento de funcionalidades	Se trata de desplegar el sistema a producción con la funcionalidad incompleta oculta para usuarios o sistemas
Interruptor de funcionalidades	Se implementa una configuración que permita encender/apagar la funcionalidad en producción
Ramificación por abstracción	A través del interruptor de funcionalidades, se despliega el antiguo y el nuevo componente a producción

> configurando el encendido/apagado de la nueva funcionalidad por ambiente.

Adaptado de Morris (2016b, págs. Kindle Locations 3903-3908).

4. Capítulo IV. Infraestructura como código

4.1. Principios

A continuación, se enuncian los principios de la Infraestructura como código, los mismos que buscan ayudar a los equipos de operaciones a superar los retos en la gestión de la infraestructura dinámica mencionados en el Capítulo II del presente documento.

4.1.1. Infraestructura reproducible

“Debe ser posible reproducir con el mínimo esfuerzo y de forma confiable cualquier elemento de la infraestructura” (Morris, 2016b, págs. Kindle Locations 348-349)

El principio de infraestructura reproducible permite:

- Reducir gran parte del riesgo técnico y de negocio que se produce al hacer cambios sobre la infraestructura y del propio miedo al cambio. Esto se logra a través de la captura de todas aquellas decisiones relacionadas al cambio: qué software y qué versiones instalar, cómo escoger un nombre de equipo, entre otras, en scripts y herramientas de automatización que aprovisionan dicha infraestructura, eliminando así el factor humano y con él, la probabilidad de error.
- Aumentar la velocidad y la confianza en la gestión de fallos de TI, también conocida como Recuperación de Desastres. Si un servidor se cae o deja de funcionar, construir y levantar uno exactamente igual debe ser una tarea rápida y sencilla.
- Aprovisionar nuevos servicios y ambientes con el mínimo esfuerzo. Esto es factible debido a que las definiciones de todos los elementos de la

infraestructura se encuentran gestionados por herramientas de automatización de la infraestructura, lo que hace posible instanciar y aprovisionar nuevos elementos en muy poco tiempo, elimina el factor del esfuerzo manual y habilita el escalamiento de la infraestructura en momentos críticos para el negocio (Morris, 2016b, págs. Kindle Locations 348-354).

4.1.2. Infraestructura descartable

“Ningún elemento de la infraestructura es permanente, por tanto, los sistemas deben ser diseñados asumiendo que la infraestructura estará en constante cambio. El software debe seguir funcionando aun cuando los servidores desaparezcan, aparezcan o sean redimensionados” (Morris, 2016b, págs. Kindle Locations 357-358).

La confiabilidad es el término clave de este principio. Tal como Sam Johnston (Simplifying cloud: Reliability, 2012) menciona en su artículo, los sistemas tradicionales en la Era del Hierro consistían en software relativamente poco confiable funcionando sobre hardware relativamente confiable. El software poco confiable no está diseñado para soportar fallos en la infraestructura, por tanto, la continuidad del servicio se basaba en el incremento de confiabilidad a nivel del hardware, generalmente a través del uso excesivo de redundancia.

La virtualización y la nube (y con ella la infraestructura dinámica) trajeron consigo la habilidad de crear, destruir, reemplazar, redimensionar y reubicar cualquier recurso y elemento de la infraestructura (Morris, 2016b, págs. Kindle Locations 356-357). En la actualidad se vive la Era de la Nube, en la cual, el software debe ser diseñado para soportar fallas, asumiendo que los componentes de hardware de los que depende son susceptibles a ellas. La continuidad del servicio hoy se describe como software que funciona de manera confiable sobre hardware poco confiable (Johnston, 2012). Un sistema diseñado e implementado bajo la premisa de que los servidores y otros elementos de la infraestructura están siendo rutinariamente añadidos y removidos, maneja el cambio y la evolución de

la infraestructura sin problema (Morris, 2016b, págs. Kindle Locations 5603-5605).

4.1.3. Consistencia

“Dos elementos de la infraestructura que proveen el mismo servicio, por ejemplo, dos servidores de aplicación en un clúster, deben ser idénticos. El software del sistema y su configuración debe ser la misma a excepción de aquellos bits de configuración que los diferencian, como por ejemplo, sus direcciones IP” (Morris, 2016b, págs. Kindle Locations 379-381).

El permitir que las inconsistencias aparezcan sobre la infraestructura, reduce la confianza en la automatización. Por ejemplo, no es posible confiar que una acción entregue los mismos resultados después de ser ejecutada sobre tres servidores de archivos que tienen particiones de disco diferentes. La acción no es repetible y, por tanto, amigable con la automatización (Morris, 2016b, págs. Kindle Locations 382-389).

El principio de consistencia busca, entre otras cosas, eliminar los problemas que surgen a partir de la desviación de la configuración (Morris, 2016b, pág. Kindle Location 388), reto al que se enfrentan con frecuencia los equipos de TI y que ha sido descrito en el Capítulo II del presente documento. Al aplicar este principio, los equipos de TI son capaces de crear y recrear infraestructura consistente ya que:

- Los cambios que aparecen en el tiempo sobre los elementos de infraestructura son capturados en archivos de definición y gestionados por herramientas de automatización.
- El equipo conoce cómo y por qué un elemento de la infraestructura es diferente de otro facilitando así, su gestión y reproducción.

Es importante tomar en consideración que el cambio, al ser un factor constante, puede aparecer antes o después de la creación y aprovisionamiento de la infraestructura. En ambos casos, la consistencia debe ser manejada y

mantenida. La siguiente lista, describe las características que debe tener un proceso de cambio efectivo sobre servidores.

- El proceso automatizado es la manera más fácil para efectuar cambios.
- Los cambios se implementan en todos los servidores existentes relevantes.
- No está permitido que los servidores destinados a ser similares, caigan en inconsistencia.
- Los cambios se aplican sobre un proceso desatendido.
- El esfuerzo no cambia al aplicar cualquier cambio o serie de cambios, sin importar cuántos elementos de la infraestructura estén involucrados.
- Cualquier error introducido con un cambio se hace visible rápidamente (Morris, 2016b, pág. Kindle Location 2878).

4.1.4. Procesos repetibles

“Construyendo sobre la base del principio de infraestructura reproducible, cualquier acción llevada a cabo sobre los elementos de la misma debe poder ser repetible” (Morris, 2016b, págs. Kindle Locations 392-393).

A pesar de que este, es un beneficio obvio al utilizar scripts y herramientas de gestión de la configuración para la definición de la infraestructura y la gestión de cambios, puede llegar a ser difícil para los equipos de TI acostumbrarse a implementar dichas prácticas en el día a día. Por ejemplo, existen tareas que pueden parecer más fáciles si se realizan directamente sobre la infraestructura en lugar de crear y probar un script automatizado que lleve la tarea a cabo. Sin embargo, la percepción del beneficio obtenido al realizar el proceso directo, como la velocidad, por ejemplo, trae consecuencias a largo plazo, tanto en el momento en el que es necesario repetir dicho proceso como cuando se desea garantizar la consistencia de los servidores. Las tareas llevadas a cabo manualmente sobre la infraestructura disminuyen, nuevamente, la confianza sobre la automatización (Morris, 2016b, págs. Kindle Locations 394-399).

Los equipos de infraestructura más efectivos son aquellos que poseen una sólida cultura de scripting (Morris, 2016b, págs. Kindle Locations 399-400) ya que comprenden los beneficios de ella:

- La infraestructura descrita en archivos de definición y gestionada por herramientas de automatización, es declarativa, reproducible, descartable y consistente.
- Los cambios y tareas manuales llevan al equipo a enfrentar los mismos retos que la infraestructura como código busca superar.

4.1.5. Diseño evolutivo

“El software y la infraestructura deben estar diseñados tan sencillamente como sea posible para satisfacer los requisitos actuales y evolucionar a medida que aparezcan nuevos requisitos. La gestión del cambio debe ser capaz de entregar los cambios y mejoras de forma rápida y segura” (Morris, 2016b, págs. Kindle Locations 409-410).

Hoy en día, las organizaciones se han vuelto extremadamente dependientes de la tecnología para sus operaciones cotidianas (Thejendra B.S, 2014). Dado que las capacidades de negocio están siempre en constante evolución, la tecnología debe ser capaz de adaptarse al cambio en lugar de prevenirlo.

El enfoque de la planificación extensa y el diseño a detalle que toma muchos requerimientos y situaciones en cuenta desde un principio genera, sin duda, sistemas excesivamente complejos. Esta complejidad representa un problema desde dos puntos de vista:

- Debido a que no es posible predecir con absoluta certeza cómo será utilizado un sistema en la práctica y cómo sus requerimientos variarán con el paso del tiempo, las organizaciones terminan teniendo sistemas, procesos e infraestructura subutilizados.
- La complejidad dificulta el cambio y las mejoras. Un sistema complejo es poco flexible y su escalamiento de vuelve difícil y costoso (Morris, 2016b, págs. Kindle Locations 406-408).

Mediante el uso continuo y frecuente de prácticas que garantizan y construyen la calidad en el sistema y, la delegación de la gestión de configuraciones en las herramientas de automatización, los equipos de TI serán capaces de aumentar el entendimiento sobre el sistema, así como también la confianza para realizar cambios sobre él (Morris, 2016b, págs. Kindle Locations 410-412). De esta manera, la necesidad de diseñar sistemas e infraestructura compleja desde un inicio se elimina, dando paso a la evolución de la tecnología que acompaña la evolución misma del negocio.

4.2. Prácticas

En esta sección, se resumen las prácticas generales de infraestructura como código. Todas las prácticas aquí mencionadas soportan los principios descritos en la sección previa.

4.2.1. Archivos de definición de la infraestructura

Una práctica clave de la infraestructura como código es el uso de archivos de definición de configuración. Estos archivos detallan las características de los elementos de la infraestructura (servidores, partes de un servidor, configuración de la red, etc.) y cómo deben ser configurados utilizando, generalmente, lenguajes de dominio específico (DSL) como Chef, Puppet y Ansible, o formatos más conocidos como JSON, YAML o XML (Morris, 2016b, págs. Kindle Locations 420-421).

Un archivo de definición puede ser tratado de igual manera como es tratado el código fuente del software, haciendo posible aprovechar el ecosistema de herramientas disponibles para su desarrollo. Este tipo de archivos se utilizan como entradas para la herramienta que hace el trabajo de aprovisionamiento y/o configuración de instancias de cada uno de los elementos de la infraestructura (Morris, 2016b, págs. Kindle Locations 418-420).

4.2.2. Sistemas y procesos auto documentados

En las áreas de TI la documentación es utilizada, entre otras cosas, como:

- Un mecanismo de comunicación entre equipos.
- Una herramienta para asegurar el cumplimiento de estándares, consistencia e incluso acuerdos legales.
- Una manera de describir y detallar procesos.

Sin embargo, de manera frecuente, se convierte en un punto de información poco relevante, útil o precisa debido al trabajo adicional que requiere su actualización y mantenimiento (Morris, 2016b, págs. Kindle Locations 437-442).

Con la infraestructura como código se busca generar documentación que pueda ser actualizada de manera automática cada vez que se introduzca un cambio en el código, sistemas o procesos a los que hace referencia (Morris, 2016b, págs. Kindle Locations 442-443). A esto, se le conoce también como documentación viva y su objetivo es ayudar a minimizar la información obsoleta. Los conjuntos de prueba y las buenas prácticas de código limpio son una excelente herramienta de comunicación de los sistemas y, un claro ejemplo de documentación viva.

4.2.3. Versionamiento

El SCV, mencionado en el Capítulo III del presente documento, es una parte esencial de la infraestructura como código ya que impulsa los cambios en la infraestructura a través de los cambios registrados en el código que administra (Morris, 2016b, págs. Kindle Locations 453-454).

La Tabla 2 muestra la relación entre las funcionalidades del SCV y el valor que aporta cada una de ellas en la infraestructura como código.

Tabla 2.
Relación funcionalidad-valor del SCV en la Infraestructura como código.

Funcionalidad	***Valor***
Trazabilidad	Facilita la depuración de errores

Reversión de cambios	Provee una acción rápida cuando uno o más cambios rompen el funcionamiento del sistema
Correlación	Es útil para el rastreo y resolución de problemas complejos
Visibilidad	Ayuda en la identificación de los problemas de manera rápida
Accionamiento	Es clave para la habilitación de los flujos de tareas de CI y CD

Adaptado de Morris (2016b, págs. Kindle Locations 455-466).

4.2.4. Sistemas y procesos probados de manera continua

Con el uso de herramientas de automatización de la infraestructura, una gran ventaja y una potente herramienta para los equipos de TI, es la habilidad de configurar máquinas rápida, automática y simultáneamente. Sin embargo, es importante tener en cuenta que, de la mano de aquellas ventajas viene la posibilidad de dañar varias máquinas con la misma velocidad y facilidad. Esta posibilidad es la principal desencadenante del miedo a la automatización por parte de los equipos (Morris, 2016b, págs. Kindle Locations 4613-4623). Un error simple puede causar un daño masivo en los sistemas de alta criticidad para el negocio y generar un impacto económico sustancial.

Las pruebas automatizadas efectivas habilitan el proceso de evaluación continua de los sistemas y son una práctica básica de los equipos de desarrollo de alto rendimiento. Estas pruebas proveen una retroalimentación activa del efecto de cada cambio sobre los sistemas lo que, a su vez, genera confianza en el equipo para efectuar cambios de manera más rápida y frecuente. La confianza de hacer cambios sin generar impactos negativos es clave en la eliminación del miedo a la automatización y, por consiguiente, un factor de suma importancia en la infraestructura como código (Morris, 2016b, págs. Kindle Locations 475-478).

4.2.5. Cambios pequeños

Esta, al igual que las prácticas mencionadas anteriormente, busca fomentar la confianza en la introducción de cambios mediante la simplificación en la identificación y resolución de problemas.

Toda la idea detrás de la Infraestructura como código se basa, principalmente, en el planteamiento de que un cambio pequeño es mejor que uno o varios cambios grandes (Morris, 2016b, pág. Kindle Location 486), siendo este, uno más de los aprendizajes adoptados de las metodologías ágiles para el desarrollo de software.

Las razones que apoyan dicho planteamiento incluyen:

- Es más fácil y toma menos trabajo probar un cambio pequeño y asegurar su solidez.
- Es más sencillo detectar la causa de un problema en un cambio pequeño.
- Es más rápido revertir un cambio pequeño cuando surge un problema que revertir una serie de cambios que componen una funcionalidad más grande.
- Se reduce el acoplamiento entre cambios importantes. Si se presenta un problema y el cambio no puede ser aplicado, otros cambios no se ven afectados ni retrasados.
- La motivación del equipo aumenta cuando es capaz de ver los resultados de su trabajo de manera más frecuente (Morris, 2016b, págs. Kindle Locations 487-490).

4.2.6. Continuidad del servicio

En la sección 4.1.2. Infraestructura descartable, se ha hecho mención de la continuidad del servicio y su importancia para el negocio.

Es necesario recalcar la relevancia en la infraestructura dinámica de mantener datos intactos y servicios siempre disponibles sin importar lo que esté sucediendo con la infraestructura. Si un servidor desaparece, deben existir

servidores disponibles para suplir la necesidad de manera inmediata mientras nuevos servidores se preparan rápidamente para empezar el ciclo nuevamente (Morris, 2016b, págs. Kindle Locations 495-497).

La gestión de los datos es un caso particularmente especial y sale del alcance de este trabajo. Sin embargo, existen algunas técnicas para mantener la consistencia y continuidad de los datos que, aunque ya son ampliamente conocidas por administradores de TI, vale la pena listar:

- Replicación de datos con redundancia.
- Regeneración de datos.
- Delegación de la persistencia de los datos.
- Copias de seguridad en almacenamiento persistente (Morris, 2016b, pág. Kindle Location 5786).

5. Capítulo V. Consideraciones de implementación desde el punto de vista organizacional

El presente capítulo recopila algunas recomendaciones basadas en patrones visibles en organizaciones que hacen un uso particularmente efectivo de los principios y prácticas de la infraestructura como código y las tecnologías basadas en la nube.

5.1. Acordar sobre los resultados esperados

Antes de considerar iniciativas como mover la infraestructura desde un centro de datos local hacia la nube o dar un paso adelante hacia la automatización de la misma, es importante que los equipos de TI comprendan y lleguen a acuerdos sobre los resultados que esperan lograr. El éxito de dichas iniciativas depende directamente de la participación de los usuarios y las partes interesadas dentro y fuera de la organización en el establecimiento de los objetivos de los servicios de TI, incluida la infraestructura (Morris, 2016b, págs. Kindle Locations 6271-6274).

Algunas actividades que han demostrado ser de gran ayuda para discutir dificultades actuales y necesidades de la organización con el fin de construir de un entendimiento común sobre los problemas que la organización necesita resolver, definir potenciales enfoques para su solución y encontrar formas de medir el progreso y realizar mejoras en el tiempo incluyen talleres como Futurespectives, Discurso de ascensor, Caja de producto y Personas (Morris, 2016b, págs. Kindle Locations 6275-6277).

5.2. Escoger métricas que ayuden al equipo

Las métricas que el equipo de TI escoja para medir su desempeño, efectividad, entre otros factores, deben proveer valor en base a los objetivos y el entorno del mismo. Estas métricas son indispensables para el proceso de mejora continua y requieren de una evaluación periódica para determinar si continuan aportando el valor que deberían (Morris, 2016b, págs. Kindle Locations 6288-6291). La Tabla 3 resume algunas de las métricas más comunes utilizadas por equipos de infraestructura.

Tabla 3.
Resumen de métricas comunes utilizadas por equipos de infraestructura.

Métrica	***Descripción***
Tiempo del ciclo	Tiempo que toma en satisfacer una necesidad desde su identificación. Esta es una medida de la eficiencia y velocidad de la gestión del cambio.
Tiempo medio de recuperación (MTTR)	Tiempo que toma en resolver un problema de disponibilidad (que incluye un desempeño o funcionalidad degradados críticamente) desde su identificación. Esta es una medida de la eficiencia y velocidad de la resolución de problemas.

Tiempo medio entre fallos (MTBF)	Tiempo entre problemas críticos de disponibilidad. Esta es una medida de la estabilidad del sistema y la calidad del proceso de gestión del cambio.
Disponibilidad	Porcentaje del tiempo en el que el sistema se encuentra disponible, excluyendo normalmente el tiempo de indisponibilidad causado por mantenimientos planificados. Esta métrica es utilizada a menudo como SLA en contratos de servicio.
Disponibilidad real	Porcentaje del tiempo en el que el sistema se encuentra disponible incluyendo mantenimientos planificados.

Adaptado de Morris (2016b, págs. Kindle Locations 6295-6307).

5.3. Utilizar una herramienta como Kanban para aumentar la visibilidad del trabajo

El tablero Kanban es una herramienta poderosa a la hora de visibilizar la cadena de valor en el trabajo de un equipo e identificar posibles cuellos de botella en los procesos, cambiando notablemente la dinámica y la efectividad del equipo. La Figura 4 muestra un ejemplo de un tablero Kanban.

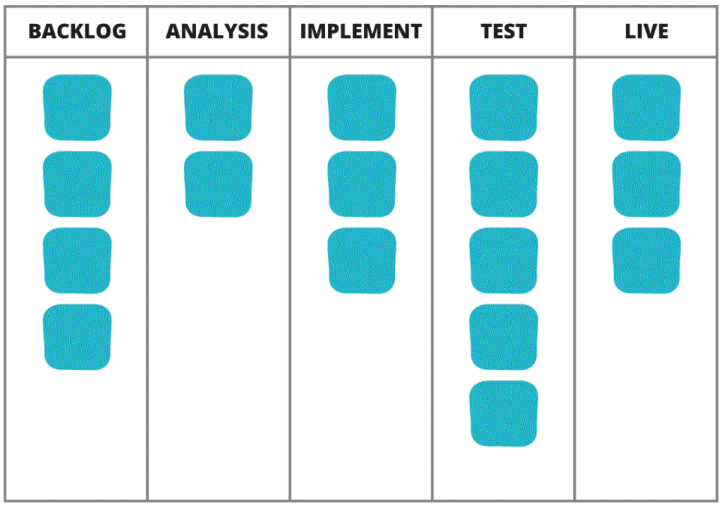

Figura 4. Ejemplo de un Tablero Kanban.
Tomado de Morris (2016b, pág. Kindle Location 6361).

5.4. Organizar a los equipos para empoderar a los usuarios

La forma más poderosa de administrar una infraestructura a gran escala para múltiples usuarios es delegar el control y la responsabilidad a esos usuarios. Desafortunadamente, es muy común en las organizaciones la división de equipos que sigue el modelo por separación de funciones. Así las responsabilidades de diseño, implementación y soporte de servicios quedan distribuidas entre múltiples equipos, obteniendo algunos beneficios como la delimitación clara y bien definida de trabajo y la creación de equipos expertos en ciertas funcionalidades. Sin embargo, este tipo de división crea algunas dificultades. La Tabla 4 recopila algunos de los problemas más comunes vistos en equipos divididos por funciones.

Tabla 4.
Problemas comunes en la división de equipos por funciones.

Problema	***Descripción***
Fragmentación del diseño	El diseño y las especificaciones son creadas de manera centralizada y se reparten entre los distintos equipos para su implementación. Esto reduce la visibilidad del panorama completo de diseño en los equipos, resultando en integraciones difíciles y problemáticas que aumentan la probabilidad de la aparición de trabajo no planificado a último minuto.
Rigidez en la planificación	Cuando la implementación está dividida entre múltiples equipos entonces los equipos se vuelven dependientes entre sí. La planificación de proyectos y recursos crean calendarios rígidos con cero tolerancia a fallos, mejoras o evolución del diseño.

Tiempos de ciclo largos	El tiempo de ciclo es proporcional al número de equipos involucrados en la construcción de una pieza de trabajo. Cada nuevo equipo agrega una sobrecarga de tiempo adicional como se puede ver en la Figura 5.

Adaptado de Morris (2016b, págs. Kindle Locations 6399-6419).

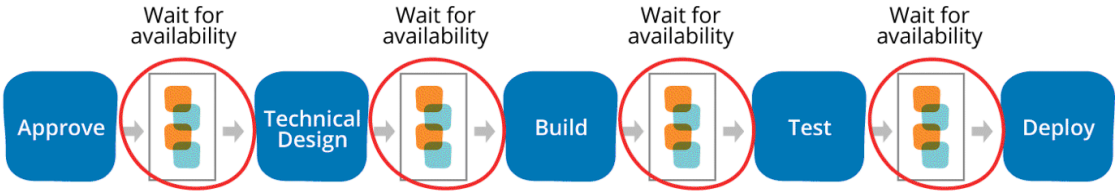

Figura 5. Flujo de valor con traspasos entre equipos.
Tomado de Morris (2016b, pág. Kindle Location 6420).

Los usuarios de la infraestructura pueden ser empoderados a través del modelo de autoservicio. Con este modelo, los equipos proveedores deben asegurar que los equipos usuarios de la infraestructura tienen las herramientas necesarias para definir, aprovisionar y configurar infraestructura para su propio uso. Los equipos proveedores a menudo asesoran, asisten y ofrecen revisiones de implementación a los equipos usuarios. A través de este modelo se logra optimizar el flujo de valor ya que un solo equipo es capaz de realizar el diseño, la construcción, las pruebas y el despliegue de su aplicación minimizando o eliminando la necesidad de traspaso entre equipos (Morris, 2016b, págs. Kindle Locations 6433-6437).

Una estrategia que agrega valor a la hora de empoderar a los equipos consiste en eliminar el concepto de equipos de desarrollo y equipos de soporte y reemplazarlo por el concepto “quien lo construye, lo despliega” popularizado por Amazon y Netflix. Esto se logra a través de la configuración de equipos en funcionalidades cruzadas. De esta manera, cada equipo se conforma de personas capacitadas y responsables por construir y ejecutar un aspecto del sistema conjuntamente (Morris, 2016b, págs. Kindle Locations 6454-6465).

5.5. Gobernabilidad de TI a través de la gestión del cambio continuo

El empoderamiento de los equipos con funciones cruzadas puede traer conflictos en los enfoques tradicionales de gobernabilidad, principalmente porque los procesos con tecnologías antiguas de infraestructura dependen de revisiones manuales e inspecciones profundas cada vez que se efectúa un cambio. Sin embargo, las prácticas y principios de infraestructura como código habilitan una gestión del cambio de manera continua que toma las ventajas de los procesos automatizados para asegurar que cada cambio es validado de manera temprana y cualquier problema es identificado y corregido inmediatamente. Este enfoque no necesita de intervención manual dado que construye las validaciones de conformidad y auditoría en la herramientas del flujo de trabajo (Morris, 2016b, págs. Kindle Locations 6482-6487). Algunos elementos clave de un proceso de gestión del cambio efectivo se resumen en la Tabla 5.

Tabla 5.
Elementos clave de un proceso de gestión del cambio efectivo.

Elemento	***Descripción***
Proveer bloques de construcción sólidos	Los equipos de infraestructura deben ser capaces de proveer a los equipos de aplicaciones bloques de construcción sólidos y flexibles que incluyan todas las validaciones de conformidad y seguridad necesarias.
Demostrar la disponibilidad operativa a través del flujo de tareas	Los equipos de infraestructura deben trabajar de la mano con los equipos de aplicaciones para garantizar que el flujo de tareas del equipo valida los requisitos operacionales en cada cambio.
Compartir la responsabilidad por la calidad operacional	La seguridad, la conformidad con las normas, la calidad operacional, el desempeño y demás requerimientos no funcionales, deben ser vistos

	como una responsabilidad compartida de todos los equipos en la organización. Es así que, el rol de los equipos de operaciones en este esquema cambia, para ser quienes ayuden a potenciar el empoderamiento de estos requerimientos en los equipos de aplicaciones, de manera que ellos sean capaces de probar y garantizar su cumplimiento.
Revisar y auditar los procesos automatizados	Los equipos de infraestructura deben revisar periódicamente los flujos de tareas y la automatización de sus pruebas a través de registros y reportes de ejecución de manera que puedan evaluar la calidad del proceso en sí mismo.

Adaptado de Morris (2016b, págs. Kindle Locations 6488-6529).

6. Capítulo VI. Caso de estudio: empresa Denarius

6.1. Antecedentes

Denarius es una sociedad anónima que surge con el enfoque principal de prestar servicios de procesamiento y aprovisionamiento tecnológico a entidades financieras del Ecuador en modalidad de software como servicios.

El producto principal que Denarius ofrece es el Core Financiero que incluye módulos de gestión de clientes, cuentas de ahorros y corrientes, depósitos a plazo fijo, crédito, cartera, garantías y contabilidad. Sobre el Core Financiero se proveen servicios agregados como facturación electrónica y gestión de pagos a proveedores.

La dinámica del negocio financiero y las fuertes regulaciones del sector, demandan de los proveedores de servicios tecnológicos un alto nivel de experiencia, compromiso, orientación a la calidad y excelencia técnica para poder proveer a las instituciones de una alta disponibilidad, interoperabilidad,

flexibilidad y rendimiento de las aplicaciones y cumplir con las regulaciones relacionadas con la continuidad del negocio.

Para hacer frente a este reto, Denarius basa su estrategia en el seguimiento y aplicación de los principios y valores del manifiesto ágil. Esta aplicación se concreta en dos frentes: el desarrollo ágil y la cultura DevOps. Para el desarrollo ágil Denarius utiliza las metodologías Scrum y Kanban, mientras que la infraestructura como código y la entrega continua es la forma más efectiva de implementar una estrategia de DevOps.

En el proceso de mejora continua iniciado por Denarius para implementar su estrategia de DevOps, fue indispensable migrar su infraestructura desde un centro de datos local costoso y sin la flexibilidad requerida, hacia la plataforma en la nube Azure de Microsoft que ofrece las herramientas y los servicios requeridos para implementar el concepto de infraestructura como código.

6.2. Análisis del caso

Tal como se describe en los antecedentes, la empresa Denarius se encuentra en el proceso de migración de su infraestructura hacia la plataforma en la nube Azure de Microsoft. Este proceso requiere la adopción de una herramienta de automatización de la infraestructura que le permita, entre otras cosas, la descripción de los elementos de su infraestructura en forma de código, el despliegue automatizado de dichos elementos y la gestión de sus cambios.

En la fase inicial del estudio se identificó que la empresa Denarius hacía uso del lenguaje PowerShell para el aprovisionamiento de los servidores en su centro de datos local. Sin embargo, con el fin de aprovechar al máximo las ventajas de la automatización de la infraestructura y, debido a la alta compatibilidad con scripts de PowerShell y con la plataforma Azure, se decidió empezar a utilizar Chef como plataforma de automatización y lenguaje descriptivo para sus nuevos servidores en la nube.

Por otro lado, con el fin de administrar y gestionar cambios sobre el código fuente de la infraestructura, se escogió Git como el sistema de control de versiones por

defecto. Git se encuentra incluido en el entorno integrado de desarrollo Visual Studio Code de Microsoft con el cual se desarrolla el código de la infraestructura de Denarius permitiendo la creación de repositorios locales con control de versiones de manera fácil y rápida.

En la Figura 6 se encuentra el diagrama de la nueva plataforma con Chef, Git y Azure como principales componentes.

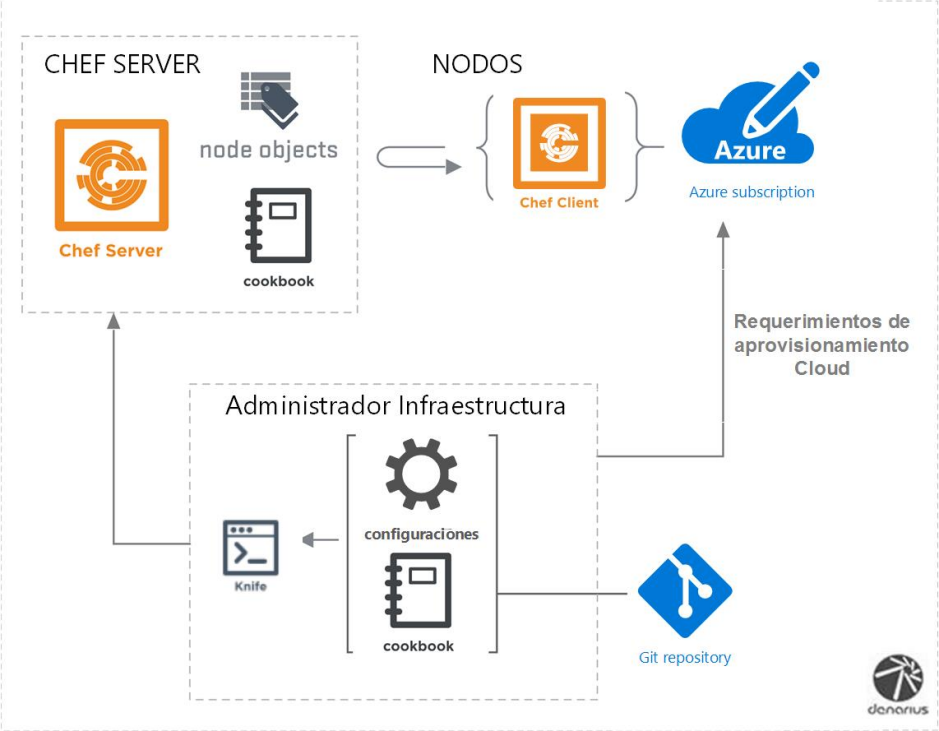

Figura 6. Diagrama de la plataforma de Denarius con Chef, Git y Azure

La herramienta Visual Studio Code es, en esencia, un editor de código fuente que provee extensiones para personalizar la herramienta según las necesidades del desarrollador. El código de la infraestructura de Denarius, desarrollado en dicho entorno y gestionado por Git, es registrado varias veces al día en un repositorio distribuido privado localizado en la plataforma Visual Studio Online. De esta manera, es posible implementar a futuro un proceso de integración continua sobre la infraestructura como código de la empresa. En la Figura 7 se encuentra una captura de pantalla del uso de dicha herramienta en Denarius.

La plataforma Visual Studio Online provee herramientas de desarrollo de software para equipos, entre ellas, repositorios Git privados, repositorios privados con SCV centralizados, integración continua, gestión de despliegues y herramientas ágiles como paneles Kanban o tableros Scrum. En el proceso de migración hacia Azure, Denarius ha aprovechado, inicialmente, dicha plataforma para implementar un repositorio Git privado que permite la integración del código de manera frecuente tal como se muestra en la Figura 8.

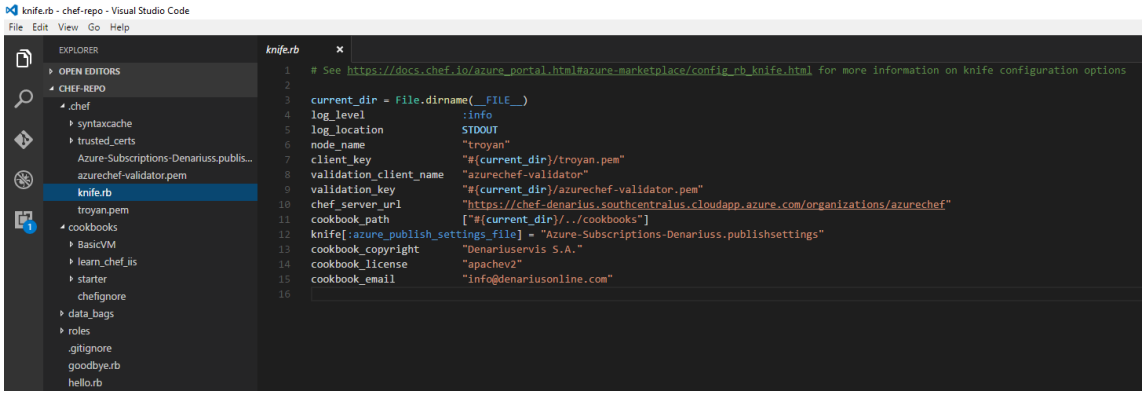

Figura 7. Captura de pantalla de la herramienta Visual Studio Code en Denarius

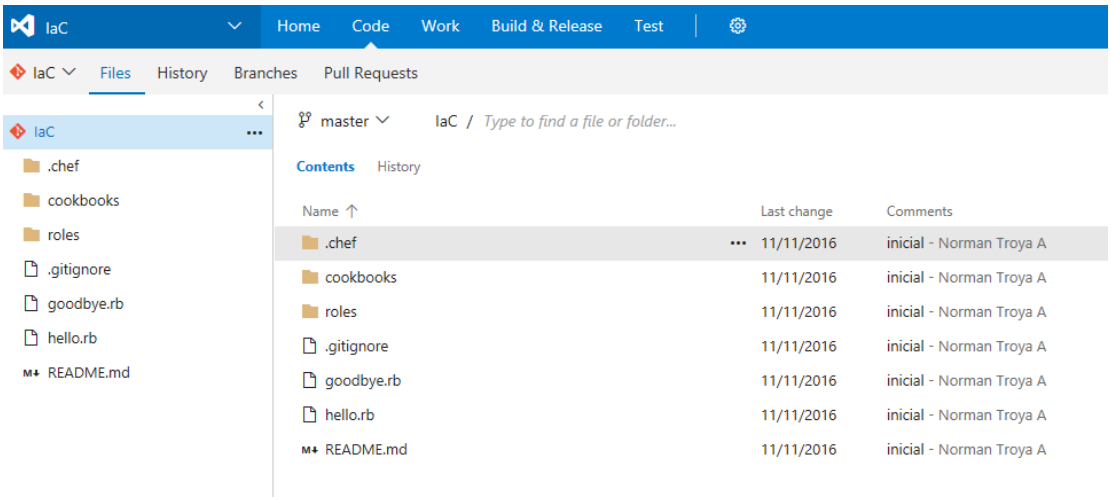

Figura 8. Repositorio Git privado para el código de la infraestructura en Visual Studio Online

Chef está conformado por distintos componentes que proporcionan todo lo necesario para administrar la infraestructura en forma de código. El primer componente de Chef es el equipo para desarrollo o Chef DK el cual provee un

conjunto de herramientas para desarrollar y probar el código de automatización de la infraestructura en ambientes locales de desarrollo antes de desplegar cambios en ambientes productivos. El segundo componente se denomina servidor Chef y actúa como un repositorio central de cookbooks e información general sobre cada nodo que él gestiona en producción. El tercer, y último componente, es el cliente Chef, el mismo que representa a cualquier nodo en una máquina física o virtual de la red que es gestionado por el servidor Chef. El cliente Chef se ejecuta en cada uno de los nodos y se comunica de manera segura con el servidor Chef para obtener las instrucciones de configuración más actualizadas (Chef Software Inc., 2016). La Figura 9 muestra dichos componentes a más detalle.

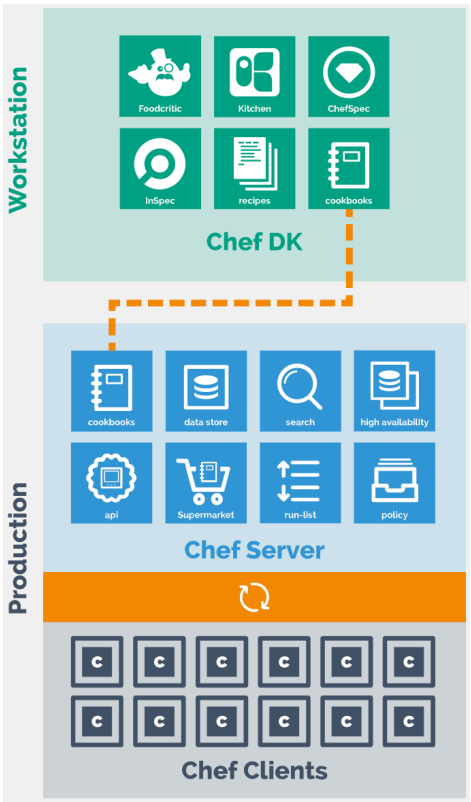

Figura 9. Componentes de Chef.
Tomado de Chef.io (Chef Software Inc., 2016).

En Denarius se configuró un servidor Chef en la plataforma Azure, el mismo que almacena los cookbooks que describen los elementos de la infraestructura en la empresa y gestiona el despliegue y los cambios de los servidores en la nube de

manera automatizada. Las Figuras 10, 11, 12, 13 y 14 muestran detalles del servidor chef y un ejemplo de un cliente chef funcionando sobre Azure en Denarius.

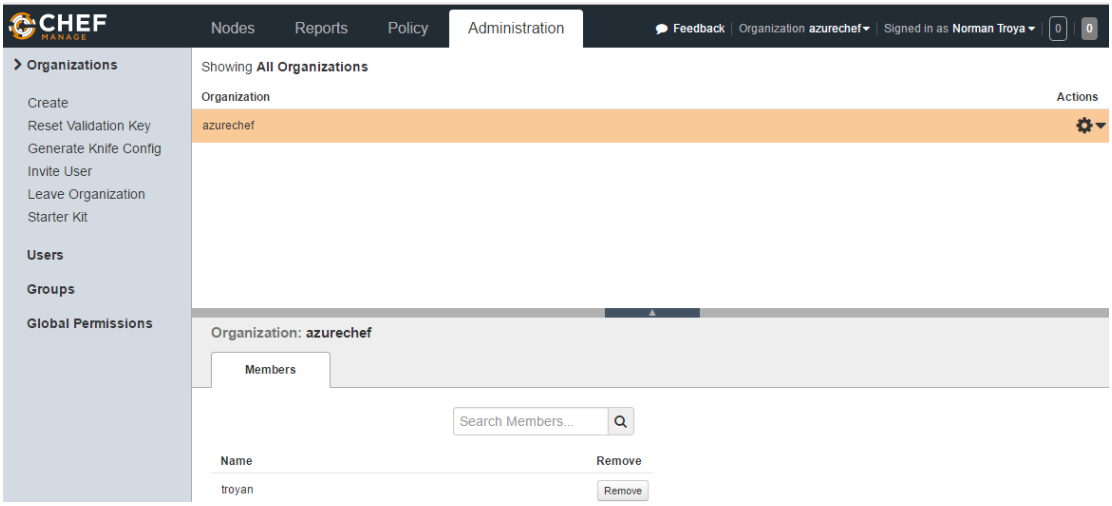

Figura 10. Organizaciones creadas en el servidor Chef.

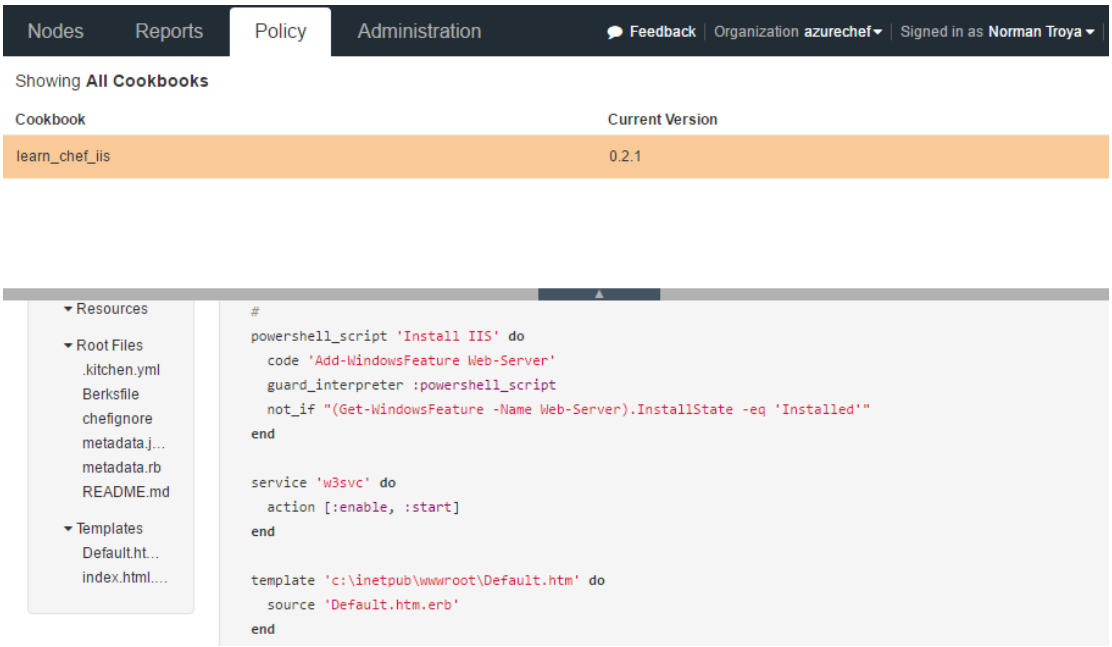

Figura 11. Cookbooks (políticas) registradas desde la máquina de desarrollo en el servidor Chef.

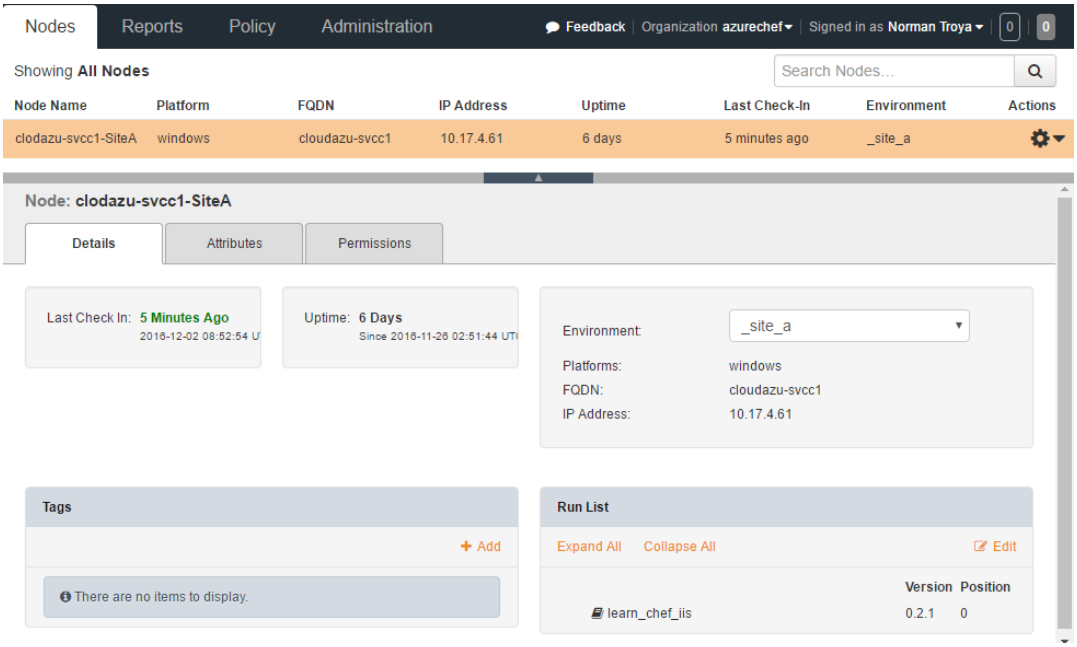

Figura 12. Visualización del estado de arranque de los nodos clientes en el servidor Chef.

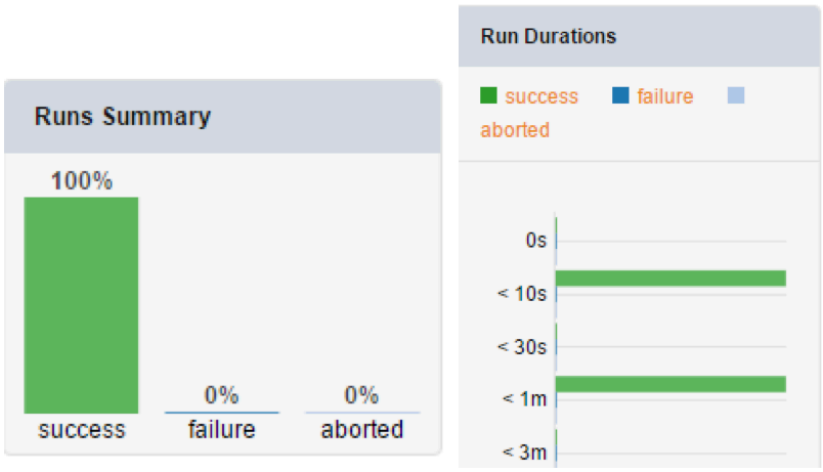

Figura 13. Ejemplo de algunos reportes disponibles sobre el estado de los nodos clientes en el servidor Chef.

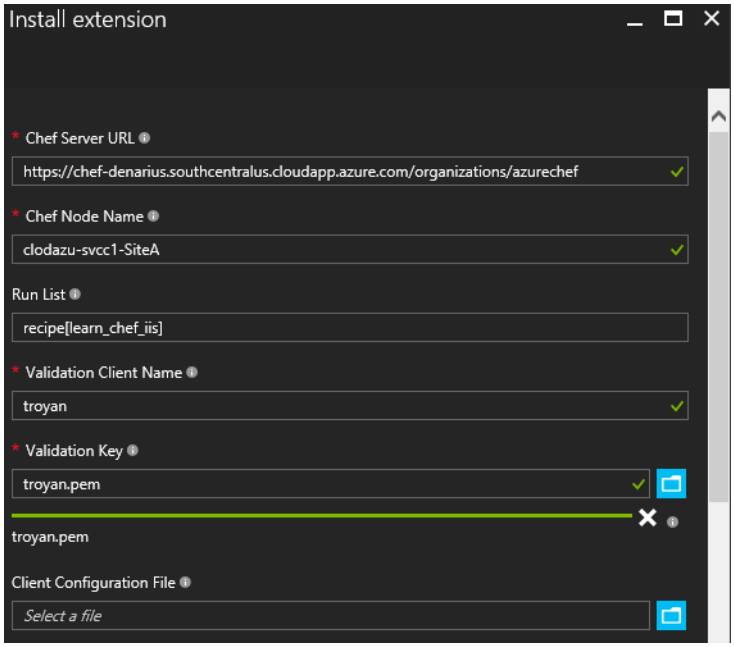

Figura 14. Instalación del cliente Chef en un servidor de Denarius.

6.3. Próximos pasos

Durante el seguimiento y análisis del caso de estudio se logró plantear una base de herramientas y configuraciones necesarias para la futura implementación de las prácticas de infraestructura como código. El avance ha sido relacionado a la preparación de las herramientas de automatización de la infraestructura necesarias para empezar a describir la infraestructura en términos de software.

Los siguientes pasos a realizarse en Denarius se basan, directamente, en la adopción de prácticas que permitan construir la calidad del código de la infraestructura, entre ellas, la utilización de una herramienta como ChefSpec para implementar pruebas unitarias utilizando TDD, la generación de pruebas de aprovisionamiento o integración con una herramienta como Test Kitchen y, de manera opcional, la creación de pruebas de auditoría sobre la infraestructura como una herramienta como InSpec.

Una vez implementado lo anterior, es posible continuar con la adopción de prácticas de integración continua. Para esto se deberá trabajar en la configuración y puesta en funcionamiento de flujos de trabajo que aseguren la ejecución continua de los conjuntos de prueba con cada registro de cambios

sobre el código. Por último, la empresa podrá decidir si desea implementar flujos de entrega continua, lo cual dependerá de la frecuencia de los cambios que sufra la infraestructura en el tiempo.

Es importante notar que la empresa Denarius ya posee experiencia con las prácticas ágiles relacionadas a lo antes mencionado lo que facilita la adopción de las mismas en la gestión del código de la infraestructura.

7. Conclusiones y recomendaciones

7.1. Conclusiones

La tecnología avanza a un ritmo acelerado. Cada avance tecnológico trae consigo potenciales beneficios, pero a su vez, plantea nuevos retos que deben ser superados por profesionales preparados para adoptar y abrazar el cambio en lugar de evitarlo.

La nube, con la introducción del concepto de hardware dinámico ha forzado a los equipos de operaciones de TI no solo a cambiar sus prácticas de gestión, sino también a sumergirse en un entorno creciente de nuevas herramientas y tecnologías que disponen grandes retos de aprendizaje e implementación.

La posibilidad de describir infraestructura virtual y física en forma de código ha disminuido la brecha que separaba a los equipos de desarrollo de aplicaciones de los equipos de operaciones de TI. Con esto, las capacidades en desarrollo de código se vuelven necesarias para los profesionales que buscan enfocarse en la gestión de cualquier tipo de infraestructura.

La automatización de la infraestructura no es suficiente para la gestión de servidores, elementos de red, etc., en empresas de gran escala que han decidido mover su Core tecnológico a la nube. En el peor escenario, las herramientas asociadas introducen más problemas que los que solucionan. Es necesario implementar prácticas y principios que han probado ser de utilidad en la Ingeniería de Software y modificar las prácticas de gestión en los equipos para alcanzar resultados exitosos.

Para finalizar, en el Ecuador, existen ya una serie de empresas que desarrollan y proveen software con metodologías ágiles. La adopción de los principios y prácticas de la infraestructura como código que se relacionan a dichas metodologías es más sencillo en empresas como estas, ya que el impacto del cambio en su recurso humano es menor.

7.2. Recomendaciones

Es recomendable asegurar y promover el entendimiento de los términos Infraestructura como código y Automatización de la infraestructura en las empresas donde se deseen implementar. Es común que ambos conceptos se confundan dificultando así, la comunicación entre las distintas personas involucradas en el proceso de adopción y generando malos entendidos.

Por otro lado, si se desea migrar infraestructura y servicios a la nube, es recomendable describir desde un inicio dichos elementos en código capturando las configuraciones de cada uno en archivos de definición. Esto permitirá obtener una infraestructura robusta, reproducible y descartable con alta confiabilidad.

Por último, para equipos de operaciones de TI que deseen implementar Infraestructura como código y que no hayan tenido exposición previa a las metodologías ágiles y sus prácticas, se recomienda empezar generando una cultura sólida con algunas de ellas, incrementando paulatinamente su adopción de manera que se evidencie el valor de cada una en todas las personas del equipo. Una vez generada una costumbre de buenas prácticas en comunicación, scripting y gestión de cambios, el salto a la infraestructura como código será menos doloroso y tendrá una mayor probabilidad de éxito.

REFERENCIAS

1minus1 Limited. (s.f.). *1minus1*. Recuperado el 13 de Noviembre de 2016, de 1minus1.com: https://1minus1.com/develop

Beck, K. (2001). *Extreme Programming Explained: Embrace change.* Indianapolis, Indiana, United States of America: Addison Wesley, Inc.

Caroli, P. (2016). *To the point: A recipe for creating lean products.* Leanpub.

CCM BenchMark Group. (2017). Gobernabilidad de sistemas de información (Control de TI). Recuperado el 1 de Febrero de 2017, de CCM.net: http://es.ccm.net/contents/641-gobernabilidad-de-sistemas-de-informacion-control-de-ti

Chef Software Inc. (2016). *Chef.* Recuperado el 6 de Diciembre de 2016, de Chef.io: https://www.chef.io/chef/

Cunningham, W. (2001). *Agile Manifesto*. Recuperado el 23 de Octubre de 2016, de Agile Manifesto Web Site: agilemanifesto.org

Fast Monkeys. (2014). *Your ultimate guide to Minimum Viable Product (+great examples).* Recuperado el 26 de Noviembre de 2016, de FastMonkeys.com: https://blog.fastmonkeys.com/2014/06/18/minimum-viable-product-your-ultimate-guide-to-mvp-great-examples/

Ford, N. (2011). *Agile Engineering Practices.* O'Reilly Media Inc.

Fowler, M. (2006). *Continuous Integration.* Recuperado el 13 de Noviembre de 2016, de MartinFowler.com: http://martinfowler.com/articles/continuousIntegration.html

Fowler, M. (2015). *Refactoring: Improving the design of existing code.* (J. C. Shanklin, Ed.) Westford, Massachusetts, United States of America:

Addison Wesley Longman, Inc. Recuperado el 28 de Octubre de 2016, de Refactoring.com: http://refactoring.com/

Fowler, M., & Parsons, R. (2011). *Domain Specific Languages.* Boston, Masachussets, United States of America: Addison-Wesley.

Git. (s.f.). *Git --fast-version-control .* Recuperado el 6 de Diciembre de 2016, de Git.com: https://git-scm.com/

Heavy Water Operations, LLC (OR). (2013). *KitchenCI.* Recuperado el 6 de Diciembre de 2016, de Kitchen.ci: http://kitchen.ci/

Johnston, S. (2012). *Simplifying cloud: Reliability.* Recuperado el 28 de Noviembre de 2016, de Sam Johnston Scribes: https://samj.net/2012/03/08/simplifying-cloud-reliability/

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices.* Upper Saddle River, New Jersey, United States of America: Prentice Hall.

Martin, R. C. (2011). *Clean Code.* Boston, Massachusetts, United States of America: Prentice Hall.

Microsoft. (2016). *PowerShell.* Recuperado el 6 de Diciembre de 2016, de Microsoft.com: https://msdn.microsoft.com/en-us/powershell/mt173057.aspx

Morris, K. (2014). *Infrastructure as Code: Automation Is Not Enough.* Recuperado el 15 de Agosto de 2016, de Kief.com: http://kief.com/infrastructure-as-code-versus-automation.html

Morris, K. (2016a). *Infrastructure as Code: From the Iron Age to the Cloud Age.* (T. Inc., Ed.) Recuperado el 28 de Noviembre de 2016, de Thoughtworks.com: https://www.thoughtworks.com/insights/blog/infrastructure-code-iron-age-cloud-age

Morris, K. (2016b). *Infrastructure as Code: Managing Servers in the Cloud.* (B. Anderson, Ed.) Sebastopol, California, United States of America: O'Reilly Media, Inc.

Shore, J., & Warden, S. (2008). *The Art of Agile Development.* (M. TRESELER, Ed.) O'Reilly Media.

Somasundaram, R. (2013). *Git: Version Control for Everyone Beginner's Guide.* Birmingham, United Kindom: Packt Publishing.

Standish Group. (2015). *CHAOS Report 2015.* New York: Standish Group Inc.

Thejendra B.S. (2014). *Practical IT Service Management: A concise guide for busy executives.* Ely, Cambridgeshire, United Kingdom: IT Governance Publishing.

Versteijnen, P. (2015). *Differences between Continuous Integration, -Delivery and -Deployment explained.* Recuperado el 13 de Noviembre de 2016, de Prowareness: http://www.scrum.nl/prowareness/website/scrumblog.nsf/dx/differences-between-continuous-integration-delivery-and-deployment-explained

ANEXOS

ANEXO 1: Lista de Abreviaturas

IaaS	Infrastructure as a Service (Infraestructura como servicio)
XP	Extreme Programming (Programación extrema)
SCV	Sistema de control de versiones
TDD	Test Driven Development (Desarrollo guiado por pruebas)
CI	Continuous Integration (Integración continua)
CD	Continuous Delivery (Entrega continua)
MVP	Minimum Viable Product (Mínimo producto viable)
TI	Tecnologías de la Información
DSL	Domain Specific Language (Lenguaje de dominio específico)
SLA	Service Level Agreement (Acuerdo de nivel de servicio)

ANEXO 2: Diccionario

ChefSpec: Framework que permite probar recursos y recetas de Chef sobre un cliente Chef simulado. Esta herramienta es ampliamente utilizada para la generación de pruebas unitarias sobre el código de Chef (Chef Software Inc., 2016).

Continuidad del servicio: El objetivo de la continuidad del servicio es que los servicios estén permanentemente a disposición de los usuarios sin interrupción. El tiempo en el cual un servicio no se encuentra disponible a los usuarios, puede generar pérdidas significativas al negocio perjudicando, incluso, su reputación (Thejendra B.S, 2014).

Cookbook: Un cookbook de Chef es la unidad fundamental de distribución de configuraciones y políticas. Contiene código que describe el estado deseado de la infraestructura (Chef Software Inc., 2016).

Discurso de ascensor: Término utilizado para referirse al discurso de presentación de un proyecto ante clientes potenciales. Consiste en un juego de palabras con el que se pretende destacar sobre todo la importancia de la brevedad del mensaje.

Era del hierro: La Era del Hierro hace referencia a los tiempos pasados de la infraestructura en donde el crecimiento de la misma estaba condicionado por el ciclo de compra del hardware (Morris, 2016a).

Erosión: Concepto introducido por Ingenieros de Heroku.

Flujo de cambios: Término acuñado por Kief Morris (2016b) para diferenciar entre el flujo de despliegue de una aplicación de software y el flujo de despliegue de cambios en el código de infraestructura.

Futurespectives: Actividades que ayudan a los equipos a prepararse para el futuro. La actividad Catapulta es un ejemplo de futurespective que ayuda al equipo en la planificación y preparación para enfrentar un desafío próximo.

Git: Sistema de control de versiones distribuido, gratuito y de código abierto diseñado para la gestión de proyectos grandes o pequeños con velocidad y eficiencia (Git).

Gobernabilidad de TI: Gobernabilidad de TI se refiere a la administración y regulación de los sistemas de información que establece una compañía para el logro de sus objetivos. Por lo tanto, la gobernabilidad de TI forma parte integral del control corporativo (CCM BenchMark Group, 2017).

InSpec: Framework de pruebas para infraestructura de código abierto que permite asegurar el cumplimiento de los requisitos de seguridad y otras políticas (Chef Software Inc., 2016).

Lenguajes de dominio específico: Los DSL son lenguajes pequeños, enfocados en un aspecto particular de un sistema de software. No se puede construir un programa completo con un DSL, pero a menudo se utilizan varios DSL en un sistema escrito, principalmente, en un lenguaje de propósito general (Fowler & Parsons, 2011).

Mínimo producto viable: Ayuda a verificar si la dirección tomada es la correcta. Es el conjunto de funcionalidades inicial necesario para la validación de hipótesis o capacidades del sistema y que evoluciona a través de la validación de dichas capacidades (Caroli, 2016).

Personas: Actividad que permite perfilar usuarios del producto/servicio que se desea construir con el fin de entender cómo y por qué usarían dicho producto o servicio.

PowerShell: Plataforma de automatización y lenguaje de scripting para Windows y Windows Server que simplifica la gestión de estos sistemas mediante el aprovechamiento del Framework .Net (Microsoft, 2016).

Principios del software ágil: Inspirados en el Manifiesto Ágil, particularmente en los Doce Principios del Software Ágil.

Caja de producto: Marco de colaboración que permite aprovechar las experiencias colectivas de consumo de los clientes al pedirles que diseñen una caja de producto que representa el producto que quieren comprar. Ayuda en el aprendizaje de lo que los clientes piensan que son las características más importantes y emocionantes de un producto o servicio dado.

Ramificación: Una ramificación en un SCV es una función utilizada para iniciar una copia independiente y similar del espacio de trabajo actual o rama principal (Somasundaram, 2013).

Recuperación de desastres: Recuperación de Desastres (DR, por sus siglas en inglés), es la preparación y ejecución metódica de todos los pasos que se necesitarán para recuperarse de un desastre, generalmente uno causado por la tecnología (Thejendra B.S, 2014).

Test Kitchen: Herramienta instrumental de pruebas utilizada para ejecutar el código de configuración de la infraestructura en una o varias plataformas de manera aislada (Heavy Water Operations, LLC (OR), 2013).