View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Universidad de Zaragoza

Despliegue, operación y mantenimiento de aplicaciones a escala global

Trabajo fin de grado de Ingeniería Informática

Autor: Víctor Fernández Melic

Director: Enrique Fermín Torres Moreno

EINA / Universidad de Zaragoza

2020

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe entregarse en la Secretaría de la EINA, dentro del plazo de depósito del TFG/TFM para su evaluación).

D./Dª. ,en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de (Grado/Máster) (Título del Trabajo)

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza,

Fdo:

Resumen

Cada vez más, las empresas necesitan dar servicio a sus clientes a escala global (a usuarios en cualquier parte del mundo). Es crucial afrontar correctamente los retos que esto supone para no perder clientes debido a la degradación del servicio (tiempos de acceso largos, denegación de servicio), evitar desperdiciar recursos por sobredimensionar el sistema y mantener bajo control los fallos *hardware* y *software* que puedan ocurrir de manera fortuita o derivados de la intervención humana durante la actualización o el mantenimiento. Estos retos no son solo tecnológicos, sino también metodológicos, pues hay que gestionar el desarrollo, despliegue y mantenimiento de estas aplicaciones a escala global.

En este TFG, se propone aprovechar las posibilidades que brinda el entorno de computación en la nube para solucionar estos problemas. Así pues, se plantean una arquitectura *multi-cluster* basada en infraestructura en la nube, con IP única globa, para reducir la latencia con los usuarios por debajo de 200 ms y aumentar la tolerancia a fallos. Asimismo, se propone el uso de una arquitectura de aplicación basada en microservicios desplegados sobre *kubernetes* para adaptarse al tráfico de cada momento (elasticidad), facilitar la gestión de las actualizaciones, aumentar la resiliencia y permitir una monitorización más exhaustiva.

Pero estos cambios por si solos no garantizan el éxito, sino que hace falta una evolución en el método de trabajo que permita gestionarlos. Por eso se propone el uso de la metodología *DevOps* frente al uso de metodologías tradicionales. El enfoque de la metodología *DevOps* se basa en el uso de herramientas de automatización destinadas a agilizar el proceso de despliegue, mantenimiento y corrección de errores, posibilitando una entrega más frecuente, más segura y con menos esfuerzo a producción.

Así pues, para poner a prueba estas propuestas, se lleva a cabo el despliegue de una arquitectura *multi-cluster* con *kubernetes* sobre la nube de *Google*, haciendo uso de la metodología *DevOps* para facilitar el despliegue de una aplicación de ejemplo y su posterior actualización mediante herramientas de automatización.

Posteriormente, se realizan varias pruebas sobre esta infraestructura para verificar que efectivamente es capaz de resolver los problemas que se planteaban inicialmente y para ello se construye una infraestructura de test utilizando diferentes herramientas de la nube.

Los resultados de estas pruebas confirman que efectivamente, el uso de una arquitectura *multi-cluster* basada en *kubernetes* y desplegada en la nube mediante la ayuda de herramientas *DevOps*, permite conseguir los objetivos de elasticidad, escalabilidad, reducción de la latencia y aumento de la tolerancia a fallos, así como facilitar la operación del sistema.

Tabla de contenidos

Resumen 1

1. Introducción a la problemática de las aplicaciones a escala global 4
2. Estudio de infraestructura y arquitectura para solucionar el problema de las aplicaciones a escala global 6
   2.1 Diagrama de arquitectura 9
3. Metodología *DevOps* frente a metodología tradicional 11
4. Despliegue de una aplicación a escala global 14
   4.1 Aplicación de ejemplo para el despliegue global 14
   4.2 Despliegue de infraestructura 14
   4.3 Federación de *clusters* 15
   4.4 Integración y despliegue continuo 16
5. Verificación y resultados 18
   5.1 Resiliencia, IP única global y respuesta del *cluster* más cercano 18
   5.2 Análisis de latencia: 19
   5.3 Elasticidad y escalado: 20
   5.4 Actualización y *Rollback* con *pipelines* 22
   5.5 Testeo de diferentes versiones en producción 23
6. Conclusiones 24

Anexos 27

Anexo I: Código fuente de la aplicación de ejemplo y del fichero *Dockerfile* 27

Aplicación 27

Dockerfile 27

Anexo II: Scripts de aprovisionamiento de *hardware* 28

Anexo III: Federación de *clusters* de *kubernetes* 30

Anexo IV: Códgio pipelines de ejecución 30

Anexo V: Actualización y *rollback* 33

Referencias: 35

Índice de tablas y figuras

Tabla 1: Requisitos que deben cumplir los servicios proporcionados por la empresa elegida como base para este trabajo. 5
Tabla 2: Tabla comparativa entre infraestructura on premise y cloud. 7
Figura 1: Arquitectura de la solución propuesta para Google Cloud Platform 9
Figura 2: Etapas del proceso de Integración y Despliegue Continuo (CI/CD). 12
Figura 3: Representación del patrón sidecar. 13
Figura 4: Federación de clusters de kubernetes mediante la herramienta kubemci. 16
Figura 5: Estrategia de despliegue mediante un pipeline de CI / CD. 17
Figura 6: Petición curl a IP del servicio desde infraestructura de test. 18
Tabla 2: Latencia media de acceso a la IP global del servicio desde diferentes puntos del planeta. 19
Figura 7: Arquitectura del escenario de test de elasticidad y escalado 20
Figura 8: Métricas de rendimiento de las pruebas de cargas 21
Captura 9: Logs de kubernetes en GCP para la aplicación echo-app 21
Figura 10: Resultado tras el despliegue de la versión "v1" 22
Figura 11: Resultado tras el despliegue de la versión "v2" 22
Figura 12: Resultado tras rollback a la versión "v1" 22
Figura 13: Detalle del servicio de entrada a la aplicación. Reparte la carga a partes iguales entre las dos versiones de la aplicación. 23
Tabla 3: Diagrama de Gantt sobre el reparto temporal de tareas de este TFG. 25
Figura 14: Código Fuente de la aplicación de ejemplo utilizada para el despliegue en este TFG. 27
Figura 15: Código fuente del fichero Dockerfile para construir el contenedor y encapsular la aplicación de ejemplo 28
Figura 16: Código fuente de un script de aprovisionamiento de hardware en GCP. 29
Figura 17: Código fuente de un script para eliminar de hardware en GCP. 30
Figura 18: Código fuente de un pipeline de ejecución para el despliegue del código de la aplicación. 32
Figura 19: Código fuente de un pipeline de ejecución para la actualización del código del entorno. 33
Figura 20: Resultado de ejecución del pipeline para actualizar el código de la aplicación. 33
Figura 21: Resultado de ejecución del pipeline para actualizar el entorno y desplegar la aplicación. 34
Figura 22: Actualización y rollback de una aplicación en producción. 34

1. Introducción a la problemática de las aplicaciones a escala global

En el mundo actual, cada vez más globalizado, es habitual que empresas de cualquier tamaño tengan que dar servicio de manera global, es decir, a usuarios de todo el mundo, ya sea para proveer algún tipo de servicio, ofrecer contenidos multimedia, vender productos a través de Internet o por cualquier otro motivo. [1] Esta situación supone retos como la capacidad de adaptarse al tráfico de cada momento sin desperdiciar recursos (lo que se conoce como elasticidad), garantizar un tiempo de latencia que no haga perder usuarios, ser capaz de responder ante fallos *hardware* y *software*, bien sean fortuitos o errores humanos (resiliencia, monitorización) y planear adecuadamente la arquitectura de sus sistemas y servicios para garantizar una cierta calidad de servicio (*QoS*).

Así pues, una empresa que no tenga en cuenta estos retos corre el riesgo de perder usuarios debido a la falta de recursos para atenderlos (denegación de servicio, tiempos de espera demasiado largos…), o de desperdiciar recursos económicos si sobrestima la demanda real de sus servicios (servidores infrautilizados).

Los retos planteados pueden dividirse en dos tipos:

- Retos tecnológicos: ¿Qué infraestructura y arquitectura de aplicaciones se necesita?
- Retos metodológicos: ¿Cómo llevar a cabo el desarrollo, el despliegue y la actualización de los servicios?

El objetivo de este TFG es seleccionar una infraestructura, una arquitectura y una metodología que sean capaces de dar respuesta a los retos que supone gestionar aplicaciones a escala global. Para ello, define un caso de uso con una serie de requisitos a los que ajustarse (Introducción), reflexiona sobre la infraestructura y arquitectura más adecuadas (capítulo 2), expone las ventajas de la metodología *DevOps* en contraposición a la metodología tradicional de trabajo (capitulo 3) y lleva a cabo una prueba de concepto consistente en el despliegue de una aplicación, ajustándose a la infraestructura y arquitectura seleccionadas y siguiendo la metodología *DevOps (capítulo 4)*. Por último, se realizan una serie de comprobaciones sobre esta aplicación para verificar que efectivamente cumple con los requisitos (capítulo 5) antes de finalizar con las conclusiones.

A continuación, se presentan el caso de uso y la tabla de requisitos.

El escenario seleccionado tiene como actor principal a una empresa de servicios *IT B2B* (*Bussines to Bussines*), es decir, a una empresa que provee a otras de servicios *IT*, bien sea para su uso como clientes finales, o como intermediarios. Esta empresa tiene clientes en Europa y EE. UU. principalmente y quiere darles acceso a sus servicios, que están en constante evolución, sin sufrir caídas por problemas derivados de la cantidad de tráfico o los fallos *hardware o software*, pero sin desperdiciar recursos. Así

mismo, no quiere que la latencia de acceso a sus servicios se vea muy afectada por la localización del usuario.

Las elecciones que se hacen en este TFG vienen marcadas por los requisitos que se detallan en la siguiente tabla.

Requisito	Descripción
R1	El sistema tiene que adaptarse a la fluctuación del tráfico (ser elástico).
R2	El sistema tiene que ser escalable.
R3	La latencia de acceso al sistema no debe empeorar sustancialmente en función de la localización del usuario. Debe ser accesible a nivel global con una calidad de servicio aceptable y de manera transparente para los usuarios. Debería mantenerse en torno a 200ms como máximo.
R4	El sistema tiene que ser tolerante a fallos de la infraestructura. Es decir, resiliente.
R5	Las aplicaciones que corran en el sistema deben poder actualizarse habitualmente con un tiempo de parada de servicio mínimo.
R6	Tienen que poder testearse varias versiones de una aplicación al mismo tiempo.
R7	Tienen que poder desplegarse versiones modificadas de las aplicaciones dependiendo de la región.
R8	Tiene que existir un mecanismo de monitorización de la infraestructura y los servicios.
R9	Deben existir mecanismos para corregir errores en la actualización y despliegue de las aplicaciones.
R10	El presupuesto tiene que mantenerse lo más ajustado posible siempre que se cumplan todos los requisitos anteriores.

Tabla 1: Requisitos que deben cumplir los servicios proporcionados por la empresa elegida como base para este trabajo.

Como se puede observar en la tabla 1, los requisitos planteados para este trabajo incluyen la solución a los problemas de escalabilidad, latencia y resiliencia, así como facilidad en el mantenimiento y corrección de errores. Por último, se hace hincapié en que debe mantenerse el presupuesto tan austero como sea posible siempre que se cumplan los requisitos anteriores.

Conviene mencionar que este TFG comenzó a desarrollarse en el contexto de unas prácticas en una empresa aragonesa del sector tecnológico pero está completamente desvinculado de ella, se realizó de manera individual y sin relación alguna con sus clientes o infraestructura, tan solo con su soporte en el uso de algunas herramientas mencionadas en este TFG. La propuesta y el desarrollo de la memoria se presentaron por la vía del TFG académico, no en empresa, y con un tutor de la universidad. La empresa tenía interés en los resultados del TFG, y por eso se desarrolló algo de documentación y algunas figuras (de elaboración propia) en inglés que se incluyen en este TFG.

2. Estudio de infraestructura y arquitectura para solucionar el problema de las aplicaciones a escala global

Cuando se abordan problemas de alta disponibilidad o tolerancia a fallos, lo primero que hay que tener en cuenta es que debe existir redundancia a varios niveles (aplicación, comunicaciones, *hardware*…). En esencia, es importante que si una máquina, dispositivo o proceso no puede atender una petición, la atienda otro [2].

Se propone el uso de un *cluster* (conjunto de máquinas que trabajan de manera coordinada como si fueran una sola.) para solucionar el primero de estos problemas (redundancia hardware). Pero existen dos tipos de *clusters: on premise* (situado en las dependencias de la empresa que lo utiliza y lo gestiona) o en la nube (alojado en un proveedor *cloud*, al que se accede de manera remota y sigue un modelo de pago por uso). Uno de los requisitos principales consiste en la capacidad de adaptarse al tráfico sin desperdiciar recursos. Por ello, resulta muy conveniente el modelo de pago por uso (alquiler de recursos según la demanda) que ofrece el *cluster* en la nube, frente al modelo *on premise*, en el que para atender una mayor demanda hay que comprar previamente los servidores en base a una estimación. Si la estimación no se ajusta perfectamente a la realidad, se corre el riego de pasarse (y desperdiciar recursos) o quedarse corto (y perder clientes por no ser capaz de atender sus peticiones).

Otro requisito importante es el de mantener una latencia que no empeore sustancialmente en función de la localización. Es decir, hay que acercar (literalmente) los servidores a los clientes, de tal forma que un usuario cualquiera tenga el menor retardo posible. Este requisito nos obliga a redundar el *clúster*, ya que la empresa tiene clientes importantes en Europa y EE. UU. y la elección de usar un solo *cluster* supondría perjudicar a uno u otro mercado.

Esta solución requiere bien poseer infraestructura en varios puntos del planeta, o bien alquilarla. De nuevo el dilema *cluster on premise vs cluster* en la nube.

La infraestructura *on premise* tiene un alto coste inicial debido a la compra del *hardware*, aunque se amortiza con el tiempo y su uso no implica un sobrecoste. Por otro lado, los costes en *cloud* se basan en pagar a medida que es necesario. Este modelo permite escalar simplemente alquilando nuevos servidores de manera inmediata en la zona geográfica donde se necesiten. Con una infraestructura *on premise* el escalado está limitado a la capacidad máxima de los servidores que se han comprado. El mantenimiento *on premise* supone ocuparse de la gestión completa de la infraestructura, mientras que en la nube existen diferentes modelos de servicio:

- Infraestructura como servicio (*IaaS*), se trata del modelo de servicio de la nube en el que el proveedor *cloud* proporciona acceso remoto a los servidores, se encarga de su alojamiento y del mantenimiento del *hardware* y el hipervisor de virtualización, pero son los usuarios los encargados de gestionar el resto de las capas a partir del sistema operativo.

- Plataforma como servicio (*PaaS*), se trata de otro modelo de servicio de la nube. En este caso el proveedor de *cloud* se encarga de la gestión de la infraestructura, el hipervisor, el sistema operativo, el *middleware* y el *runtime*, mientras que el usuario tan solo se preocupa de gestionar datos y aplicaciones.

En la siguiente tabla pueden apreciarse las diferencias entre una infraestructura *on premise* y una en la nube.

	On Premise	Cloud
Coste	Inversión inicial	Pago por uso
Escalabilidad	Limitada al número de servidores físicos disponibles.	Se pueden alquilar nuevos servidores según necesidad.
Distribución geográfica	Limitada a las zonas donde estén los servidores físicos disponibles.	Pueden alquilarse servidores remotos en decenas de sitios
Tolerancia a fallos	Limitada al número de servidores físicos disponibles.	Se pueden adquirir nuevos servidores cuando algunos fallen.
Mantenimiento infraestructura	Gestión completa del mantenimiento de la infraestructura.	*IaaS* o *PaaS*

Tabla 2: Tabla comparativa entre infraestructura on premise y cloud.

Atendiendo al requisito de mantener el presupuesto bajo (dentro de la consecución del resto de requisitos), así como a las necesidades de elasticidad, escalabilidad, latencia y tolerancia a fallos (resiliencia), se propone el uso de múltiples *clusters* (dos en este caso) en la nube.

Para ajustarnos a los requisitos mencionados hasta ahora (elasticidad, escalabilidad, latencia, resiliencia, presupuesto…) desde el punto de vista del *software* no vale con coger cualquier aplicación tradicional, correrla en un *cluster* en la nube y olvidarse. De hecho, esta aproximación vale de poco. Para sacar el máximo partido, normalmente hay que re-arquitecturar la aplicación. Una aplicación tradicional (monolítica) que corra en la nube como un único gran proceso en una máquina virtual es susceptible de fallar y producir una caída en el servicio igual que si funcionase *on premise*.

Lo que se propone en este TFG es el uso de una arquitectura de microservicios. La arquitectura de microservicios, en contraposición a la arquitectura monolítica, se basa en diseñar una aplicación partiéndola en sus diferentes procesos de negocio, de tal manera que cada uno pueda funcionar de manera independiente y se comunique con el resto mediante paso de mensajes. De esta forma, cada parte (microservicio) puede funcionar y escalar por separado [3]. Si el microservicio de "inicio de sesión" recibe 10 veces más tráfico que el de "reseñas", este puede crecer independientemente del resto, permitiendo ahorrar recursos (y por lo tanto dinero).

La manera de trabajar con microservicios viene de la mano del uso de contenedores. Los contenedores son una abstracción para definir la encapsulación de los componentes de una aplicación y sus dependencias. Múltiples contenedores pueden ser ejecutados sobre un mismo sistema operativo, de tal forma que comparten su núcleo. Los contenedores proporcionan aislamiento lógico entre ellos y facilitan la portabilidad de aplicaciones completas con todas sus dependencias.

El uso de contenedores permite contar con unidades independientes que contienen parte de una aplicación. Réplicas de estos contenedores pueden aumentarse o reducirse en función de la carga de trabajo de cada momento, permitiendo lograr los objetivos de elasticidad, escalabilidad, resiliencia y ajustar el presupuesto.

Si bien es cierto que podemos alcanzar estos objetivos, la gestión de aplicaciones contenerizadas, basadas en microservicios puede suponer un gran reto de gestión debido a la complejidad con respecto a una aplicación monolítica. Para solucionar este problema de gestión, se propone el uso de *kubernetes*. Se trata de un planificador (*scheduler)* de contenedores que facilita el despliegue de contenedores sobre un *cluster* de máquinas y se encarga de mantener el estado definido por el usuario en cuanto a número de máquinas y réplicas, pudiendo escalar automáticamente en función de la demanda. *kubernetes* puede correr sobre un *cluster* de máquinas como el que hemos seleccionado al comienzo de este apartado.

Otro de los requisitos planteados en la introducción hace referencia al funcionamiento de manera transparente para los usuarios. Es decir, estos no tienen que darse cuenta de si están trabajando con una única aplicación monolítica corriendo en un servidor, o de si están interactuando con varios *clusters* de servidores redundados que utilizan *kubernetes*. Por ello, hay que conseguir un único acceso global (IP única para todos los usuarios) que redireccione el tráfico al *cluster* más cercano. Esto solo puede lograrse usando las capacidades de la nube pública (que pone a nuestra disposición balanceadores de carga globales) y la federación de los *clusters* para unificar su acceso (de esto se hablará más adelante).

Por lo tanto, resumiendo las ideas expuestas en este capítulo, atendiendo a las necesidades de elasticidad, escalabilidad, resiliencia, latencia, transparencia y ahorro de costes, se ha seleccionado el uso de infraestructura en la nube, compuesta por múltiples clusters (desplegados en diferentes zonas geográficas) federados bajo una misma IP y el uso de una arquitectura de microservicios encapsulados en contenedores corriendo sobre *kubernetes*.

2.1 Diagrama de arquitectura

En base a los requisitos expuestos en la introducción y a las conclusiones extraídas en el capítulo dos, se ha planteado el siguiente diagrama de arquitectura, que se ha implementado posteriormente en un proveedor de nube pública para llevar a cabo las pruebas que permitan evaluar la validez de la solución propuesta.

La empresa donde comenzó a desarrollarse este TFG trabajaba habitualmente con los servicios de *Google Cloud Platform (GCP)*, y contaba con personal más habituado a dicha plataforma. Se hizo un estudio de las características de diferentes proveedores de la nube (*GCP*, *AWS*, *Azure*) y se verificó que los servicios ofrecidos por todas ellas se ajustaban a los requisitos del proyecto, con la excepción de la federación de *clusters* de *kubernetes* en *AWS* y *Azure*, que quedaba en duda por la novedad de esta tecnología y la poca documentación al respecto. Por esta razón, dado que sí existía algo de documentación para *GCP* y que algunos miembros de la empresa contaban con cierta experiencia en esta plataforma, se decidió escogerla para implementar las pruebas de concepto.

El diagrama de arquitectura decidido finalmente es el que se muestra a continuación. Incluye tanto la arquitectura considerada de referencia (del balanceador de carga hacia abajo), como una parte dedicada a realizar las pruebas de verificación (infraestructura de test)

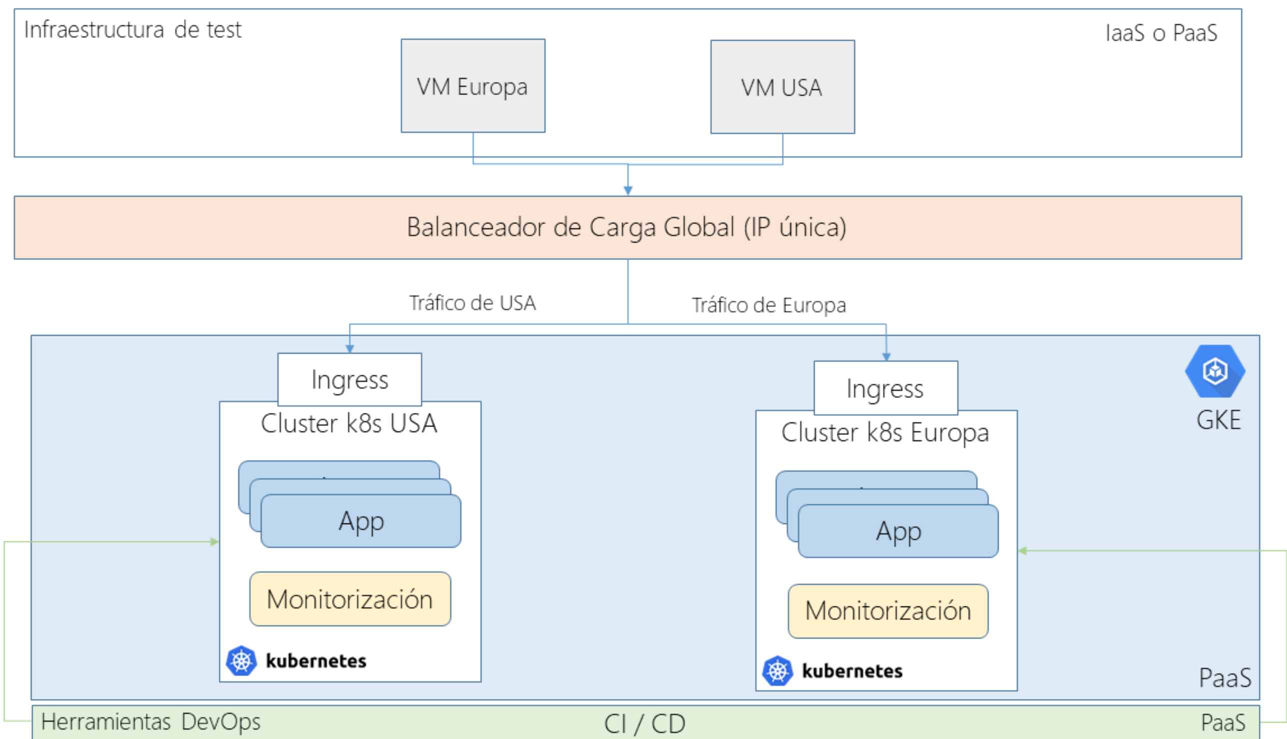

Figura 1: Arquitectura de la solución propuesta para Google Cloud Platform

Como se puede observar en la figura anterior, se han levantado dos *clusters* de *kubernetes* (*k8s*) sobre la plataforma de kubernetes gestionada de *Google Cloud Platform*: *Google Kubernetes Engine*, uno en Europa y otro en USA para realizar las pruebas. Cada uno de ellos contiene una o varias aplicaciones en producción y unas herramientas de monitorización que se explicarán en detalle más adelante. Cada uno de los *clusters* contiene un *Ingress*, que es el objeto de *kubernetes* que se encarga de dirigir el tráfico del exterior a la aplicación correspondiente dentro del *cluster*, discerniendo mediante la ruta del *path* o el puerto de entrada.

Por encima de estos *clusters* se ha colocado un balanceador de carga global con IP única que debe dirigir el tráfico al *cluster* más cercano geográficamente hablando, y que, asimismo, debe ser capaz de redirigir el tráfico a otro *cluster* en caso de fallo de cualquiera de ellos.

Por lo tanto, es fácil darse cuenta de que existen varios niveles de redundancia que permiten aumentar la fiabilidad de la aplicación:

- Redundancia a nivel de *kubernetes*: Cada contenedor desplegado dentro de *kubernetes* puede constar de varias réplicas, pudiéndose aumentar o reducir fácilmente este número en función de la carga. Esto permite optimizar los recursos y responder a picos de tráfico.
- Redundancia a nivel de *nodo*: Cada *cluster* está compuesto por una serie de nodos (máquinas virtuales) que como mínimo han de ser 3. Si un nodo falla, los otros continúan funcionando y reubican las aplicaciones en un nodo que funcione. También se pueden auto escalar los nodos que forman el *cluster* para evitar que *kubernetes* se quede sin espacio en el que ubicar nuevas aplicaciones.
- Redundancia a nivel de *cluster:* Existen *n clusters* (2 en este caso, pero podrían ser más dada la construcción del sistema) federados bajo una misma IP de entrada. Si uno de ellos falla, el otro asume todo el tráfico.

Implementando pues una arquitectura como esta se cubren los requisitos sobre elasticidad, escalabilidad, latencia, resiliencia, transparencia, ahorro de costes y tolerancia a fallos.

3. Metodología *DevOps* frente a metodología tradicional

Sumado a la complejidad de desarrollar, mantener y actualizar aplicaciones en un entorno tan complejo como el que se ha definido en el capítulo 2 (con varios *clusters*, aplicaciones basadas en microservicios, contenedores y *kubernetes*), los requisitos especifican la necesidad de actualizar la aplicación frecuentemente y con el mínimo tiempo de parada, así como ser capaces de testear diferentes versiones en producción o desplegar versiones distintas en cada *cluster*. Para lidiar con este aumento de la complejidad y hacer frente a las actualizaciones y los errores, es necesario definir una metodología de trabajo que facilite las cosas a los equipos de operaciones.

Históricamente, los equipos de trabajo dentro de las empresas tecnológicas se han dividido en dos grupos diferenciados: desarrollo y operaciones. Los primeros se encargaban del desarrollo de las aplicaciones, mientras que los segundos se ocupaban de su puesta en producción y mantenimiento. Este enfoque tradicional ha traído numerosos problemas derivados de la falta de comunicación entre equipos: errores en el despliegue, entregas a producción cada mucho tiempo, falta de verificación del código o diferencias entre entornos de desarrollo y de producción que producen fallos.

Como respuesta a esta problemática, surgió en las empresas tecnológicas la metodología / filosofía *DevOps*, centrada en la colaboración máxima entre equipos y la automatización de los procesos de construcción, despliegue y entrega de *software* a producción [4][5]. Una metodología como esta es perfecta para poder ajustarnos a los requisitos de actualización frecuente, reducción de errores y tiempos mínimos de parada. La automatización del flujo de desarrollo y despliegue permiten eliminar fallos humanos y retardos derivados de la ejecución manual, en la entrega de *software*. Asimismo, el versionado de código y artefactos de *software* (contenedores, bibliotecas, binarios) permiten corregir errores en el despliegue rápidamente (fácil *rollback*).

Se distinguen dos grandes fases en este proceso de automatización del ciclo de vida del software, como se ve en la siguiente figura:

- Integración Continua (*CI*): Hace referencia a las etapas de desarrollo y consolidación (*commit*) del *software* en un sistema de control de versiones, así como a la compilación (*build*), tests, calidad (*QA)*, encapsulación y almacenamiento (*stage)*.

- Despliegue Continuo (*CD*): Hace referencia a la etapa en la que los artefactos construidos son desplegado en producción sin necesidad de intervención humana (*deploy*).

En la siguiente figura se muestra el proceso de integración y despliegue continuo.

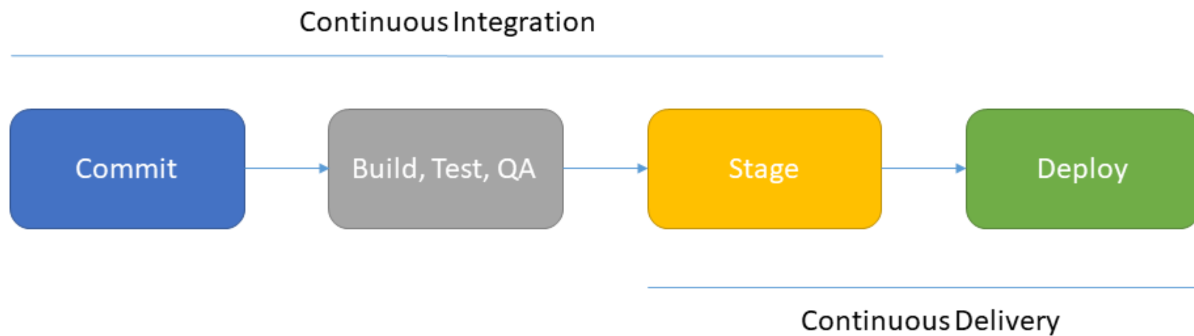

Figura 2: Etapas del proceso de Integración y Despliegue Continuo (CI/CD).

En la figura 2 se puede ver como la integración continua se refiere a todos procesos secuenciales desde que el desarrollador hace *commit* del código en un sistema de control de versiones hasta el almacenamiento de los contenedores en un registro de contenedores (*stage*), mientras que el despliegue continuo (*CD*) hace referencia al proceso de despliegue de los contenedores. Este conjunto de procesos secuenciales son lo que se conoce como línea de procesos de ejecución o *pipeline* de ejecución.

Se hizo un estudio de múltiples herramientas que permiten llevar a cabo las tareas de cada una de las fases de *CI / CD* y se comprobó, que aunque existen diferencias entre ellas (*IaaS*, *PaaS*, declarativas, procedurales, gratuitas, de pago…) la gran mayoría de ellas eran capaces de cumplir con el objetivo de implementar un *pipeline* de ejecución valido para la plataforma propuesta. Por cuestiones de concisión, y dada la amplia variedad de herramientas disponibles, no se considera relevante detallar exhaustivamente cada una de ellas. Sin embargo, se propone utilizar las herramientas de *DevOps* que proporciona *Google Cloud Platform*, ya que se integran perfectamente con el resto de los servicios de la nube de *Google*, que es (como se ha mencionado anteriormente) la que se utiliza en este TFG, y además son servicios gestionados (*PaaS)* y por lo tanto no requieren mantenimiento a nivel de infraestructura.

Para solucionar el problema de los tiempos de parada en las actualizaciones, se puede utilizar la estrategia de *rollingUpdate* de *kubernetes* [6], que consiste en la sustitución progresiva de los *pods* (objeto de *kubernetes* que aloja un contenedor y que es unidad mínima de escalado), de tal forma que un *pod* no es sustituido hasta que no han terminado todas las conexiones que está atendiendo y hasta que no está listo otro *pod* con la nueva versión de la aplicación.

Los requisitos de testeo de versiones diferentes en producción también se pueden solventar con el uso conjunto de *pipelines* de ejecución y el propio *kubernetes*. Para conocer en detalle cómo se ha solucionado este requisito, se puede revisar el apartado 5.5.

Asimismo, el requisito de despliegue de diferentes versiones según la localización se puede solucionar con ramas diferentes del código de la aplicación. Cada una de ellas será desplegada por el *pipeline* de ejecución en el *cluster* que corresponda.

Para garantizar el correcto funcionamiento de un sistema, saber si algo falla, si nos estamos quedando cortos de recursos o el rendimiento de nuestras aplicaciones, es de vital importancia el uso de sistemas de monitorización. Además, se trata de uno de los requisitos de este TFG y otro de los aspectos que marca la metodología *DevOps*.

Como solución al problema de monitorizar un sistema como este, se propone el uso de *Istio*. *Istio* es un *framework* desarrollado especialmente para *kubernetes* que hace de intermediario en la comunicación entre los diferentes microservicios que conforman una aplicación, y por tanto permite recolectar métricas de rendimiento o tracear el recorrido de una petición concreta. Está compuesto por una pila tecnológica que incluye herramientas de recolección de métricas (*Prometheus*), visualización de datos (*Grafana*) y de la comunicación (*Kiali*) y otras funciones como gestión del tráfico. Todas estas herramientas se despliegan en el *cluster* de *kubernetes* como herramientas de soporte, y la contrapartida es que por supuesto, consumen espacio, *cpu*, memoria y añaden una ligera sobrecarga a la comunicación [7].

Istio sigue el patrón de diseño *sidecar* [8], que consiste en añadir un *middleware* (el *sidecar*) a cada microservicio, de tal forma que cada microservicio solo se comunica con su *sidecar*, y toda comunicación entre microservicios o con el exterior se produzca a través de los *sidecars*. En la siguiente figura se puede observar el funcionamiento de manera sencilla.

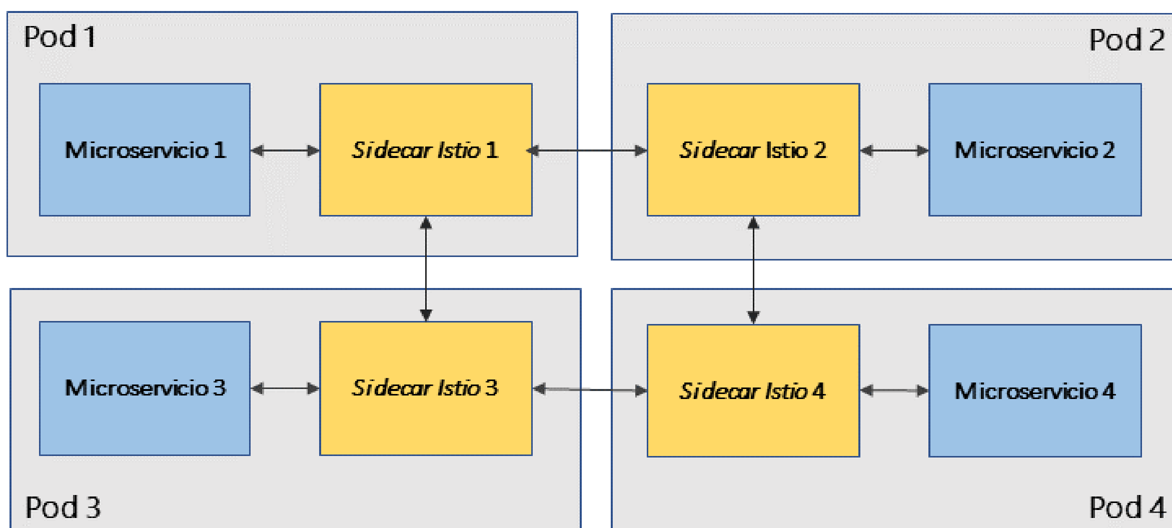

Figura 3: Representación del patrón sidecar.

En esta figura, cada uno de los *pods* contiene un microservicio acompañado de un *sidecar*. Cada microservicio se comunica con los demás a través de dicho *middleware*.

En resumen, para atender a los requisitos de despliegue, tiempo de parada, actualizaciones frecuentes y corrección de errores en el despliegue, se propone el uso de la metodología *DevOps*, que fomenta la colaboración entre los equipos de desarrollo y operaciones, y da respuesta a la complejidad que supone una arquitectura global mediante el uso de herramientas de automatización y monitorización que permiten agilizar y dotar de fiabilidad los despliegues frecuentes a producción.

4. Despliegue de una aplicación a escala global

En este capítulo se describe el proceso que se ha seguido para llevar a cabo el despliegue de una aplicación de ejemplo a escala global en la nube de *Google*, siguiendo la infraestructura, arquitectura y metodología que se ha descrito en los capítulos anteriores.

Primero se va a tratar la aplicación de ejemplo que se ha desarrollado, después se explicará el despliegue de infraestructura, así como la federación de *clusters* bajo una misma IP global y por último los *pipelines* de ejecución que se han desarrollado para llevar a cabo el despliegue.

4.1 Aplicación de ejemplo para el despliegue global

Lo primero que hacía falta desarrollar para construir la infraestructura y arquitectura que se ha propuesto en los capítulos anteriores era una aplicación que sirviera de ejemplo. La mínima aplicación viable para poder verificar los requisitos y que se pudiera encapsular en un contenedor al estilo microservicios.

La aplicación que se ha desarrollado se llama *echo-app* y consiste en un servidor que expone una API que define dos *endpoints* que contestan a peticiones diciendo en qué región se encuentra desplegada la aplicación (*EU*, *USA*). De esta forma, la aplicación sirve (además de para ser desplegada) para verificar si el tráfico realmente se redirige al *cluster* más cercano.

La aplicación está desarrollada en el lenguaje de programación *nodeJs* aunque podría haberse seleccionado cualquier otro. Esta es una ventaja de las arquitecturas de microservicios: debido al bajo acoplamiento entre los mismos, cada microservicio puede ser desarrollado en el lenguaje que decida el desarrollador.

Para utilizar la aplicación dentro de un *pipeline* de ejecución había que encapsularla en un contenedor, y esto se realizó escribiendo un fichero *Dockerfile* (documento declarativo que utiliza *Docker* para construir los contenedores). El código de la aplicación y del *Dockerfile* se encuentran en el anexo I.

4.2 Despliegue de infraestructura

Antes de seleccionar unas características definitivas para la infraestructura que se iba a desplegar, se realizaron varias pruebas para encontrar el óptimo en el compromiso entre rendimiento y coste: número de máquinas, cantidad de *cpu*, memoria, disco, localización, versiones de *kubernetes*, ya que había límites en el uso gratuito de los recursos de la plataforma.

Al final se concretó un número de 4 máquinas virtuales (nodos) para conformar cada uno de los *clusters* de *kubernetes* con 4GB de memoria cada una y una CPU dedicada (no compartida) por máquina. El número mínimo de nodos que acepta *kubernetes* para funcionar en alta disponibilidad es de 3.

Se configuró el autoescalado máximo del *cluster* (a nivel de máquinas virtuales subyacentes) en 5, para evitar sorpresas de escalado masivo derivadas de ataques de denegación de servicio o un uso excesivo.

Por supuesto, esta es una decisión que se realizó para realizar las pruebas en el contexto de este TFG, pero cada empresa debería decidir cual es el número de recursos óptimo para sus necesidades.

Se probaron 2 estrategias de aprovisionamiento del hardware:

- Manual: Para probar y aprender el correcto funcionamiento de las herramientas de *GCP*. Como se ha comentado anteriormente, se utiliza el servicio gestionado de *kubernetes* en *GCP* que tiene la ventaja de proveer *kubernetes* e *Istio* automáticamente al aprovisionar el cluster.

- Automático: Aunque la parte de despliegue de la infraestructura queda fuera del proceso habitual de *CI / CD* (más enfocado en el ciclo de vida del *software*), en este punto se quiso probar (siendo fiel a los principios de *DevOps*) a automatizar también el aprovisionamiento de *hardware*, lo que se conoce como "infraestructura como código" (*IaC*). En el anexo II hay dos pequeños *scripts* que se utilizaron para aprovisionar y eliminar toda la infraestructura cada día durante el desarrollo y las pruebas, y mantener así los costes al mínimo.

Utilizar la versión gestionada de *kubernetes* resultó una experiencia a la vez simplificada pero más confusa de utilizarlo, ya que, si bien el proceso se simplificó, inicialmente la confusión fue mayor, ya que ocurren muchas cosas de manera automática y se puede perder la concepción real de lo que está ocurriendo. Por este motivo, se utilizó cierto tiempo en aprender a gestionar *kubernetes* de manera manual.

4.3 Federación de *clusters*

Federar un conjunto de *clusters* de *kubernetes* baja una única IP global que sea capaz de resolver de manera diferente en función de la localización del usuario, es una idea novedosa que en el momento de hacer estas pruebas (diciembre de 2019) estaba todavía en desarrollo. Es posible que posteriormente hayan surgido alternativas más nuevas [9] y avanzadas, pero en el momento se escogió la herramienta *kubemci*, desarrollada por los creadores de *kubernetes* con vistas a incorporarla como parte de su estándar.

Su funcionamiento es conceptualmente sencillo: agrupar un conjunto de *Ingress* (objeto de *kubernetes* que se encarga de enrutar el tráfico de cada *cluster* a las aplicaciones correspondientes) para que

funcionen como uno solo, y después asociarle un balanceador de carga (con IP pública) de tal forma que las peticiones se repartan entre ambos *clusters* en función de su localización y disponibilidad.

En la siguiente figura queda más claro lo que se acaba de explicar.

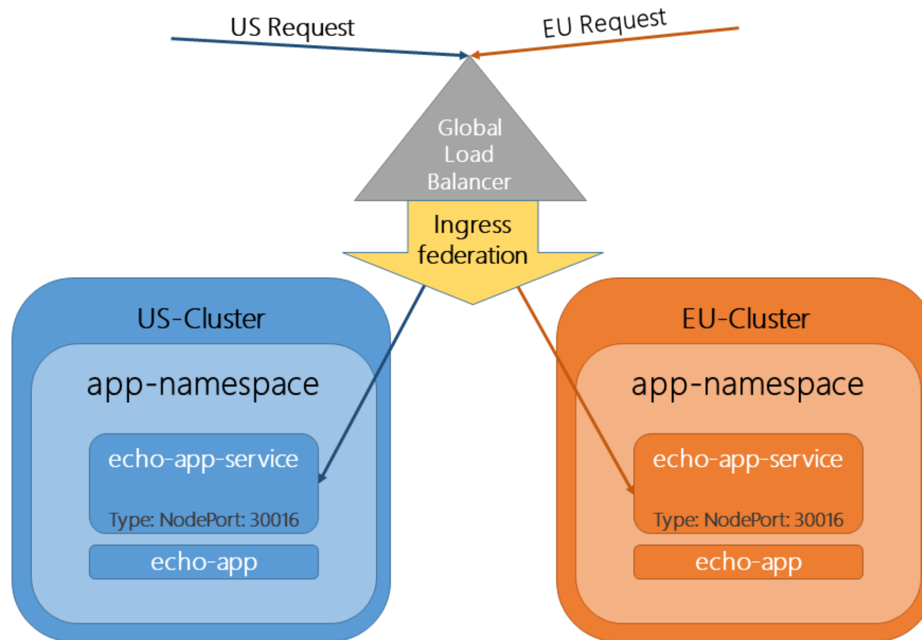

Figura 4: Federación de clusters de kubernetes mediante la herramienta kubemci.

En la figura anterior se puede ver cómo las peticiones iniciadas en Europa (*EU*) y en EE. UU. (*US*) a la IP del balanceador de carga, son enrutadas al *cluster* correspondiente por el *Ingress* federado.

Para más información sobre el procedimiento técnico necesario para federar los *Ingress* de *kubernetes*, se puede acudir al anexo III.

Llegar a esta solución final no resultó especialmente fácil debido a la escasa documentación al respecto, por tratarse de una tecnología bastante nueva. Se investigaron otras opciones como el reparto de carga por resolución de DNS con múltiples registros de tipo A, pero tenían carencias como el retardo en la propagación de los registros o el balanceo de carga no inteligente (*round-robing*), pero finalmente se consiguió llevar a cabo con éxito. Es de esperar que con el avance de esta tecnología, la federación de *clusters* se vuelva más habitual y sencilla.

4.4 Integración y despliegue continuo

Una vez desplegada la infraestructura necesaria, se abordó la implementación de un *pipeline* de CI / CD que permitiera la entrega a producción de manera automática, así como la fácil corrección de errores de despliegue. El código del *pipeline* se encuentra disponible en el anexo IV.

Siguiendo las recomendaciones de la documentación técnica de *Google* [10], se implementó una estrategia de despliegue con versionado por separado del código de la aplicación y el código del despliegue (los ficheros declarativos de *kubernetes*). En la siguiente figura se puede observar dicha estrategia.

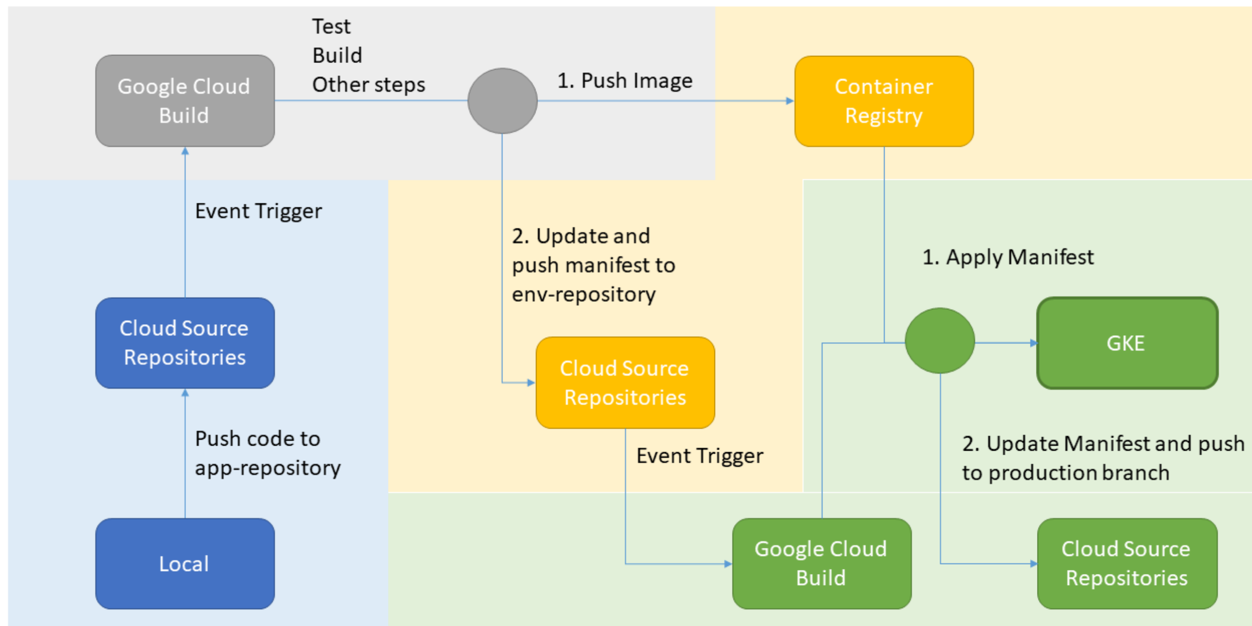

Figura 5: Estrategia de despliegue mediante un pipeline de CI / CD.

La idea del *pipeline* consiste en encadenar una serie de procesos desde el desarrollo en local hasta el despliegue en producción: cuando el desarrollador sube el código de la aplicación al repositorio de *Google*, comienzan los test, la inyección de dependencias y la construcción de un contenedor que encapsule la aplicación (el número de pasos puede variar). Después, el contenedor se almacena en el repositorio de contenedores y se actualizan automáticamente los ficheros de despliegue de *kubernetes*. Estos ficheros se versionan como tentativa de despliegue en otro repositorio dedicado (se separa aplicación de entorno) y posteriormente se aplican los cambios en *kubernetes*, lo que produce la actualización de la aplicación. Si el despliegue no da problemas al aplicarse, los ficheros se versionan como versión en producción.

Como se puede observar, esta estrategia permite mantener un histórico de versiones en producción, lo que permite volver a una versión anterior (*rollback)* rápidamente en caso de necesidad.

5. Verificación y resultados

Una vez que se ha desplegada la infraestructura necesaria, se ha desarrollado una aplicación de ejemplo y se ha puesto en producción sobre dicha infraestructura mediante un *pipeline* de ejecución de *CI / CD*, es el momento de realizar una serie de verificaciones para comprobar que efectivamente se han conseguido alcanzar los objetivos y los requisitos planteados al inicio.

A continuación se hace un repaso de las pruebas que se han llevado a cabo, y para cada una de ellas, se describe el escenario de partida, el objetivo y los resultados.

5.1 Resiliencia, IP única global y respuesta del *cluster* más cercano

El objetivo de esta prueba es verificar que la federación de *clusters* bajo una única IP ha funcionado correctamente y está correctamente configurado, y que efectivamente contesta a las peticiones el *cluster* más cercano disponible.

El escenario de partida de la prueba consiste en dos máquinas virtuales (de tipo *IaaS*) levantadas, una en Europa y otra EE. UU. completamente ajenas a la red del *cluster* (forman parte de la infraestructura de *test* de la arquitectura de referencia que se puede ver en la figura 1 de la página 8).

La prueba consiste en lanzar peticiones de tipo *curl* a la IP pública única de nuestro servicio desde cada una de las máquinas virtuales y verificar que responde el *cluster* más cercano. Después, eliminar uno de los *clusters* y verificar que las peticiones son redirigidas y contestadas por el *cluster* que queda.

Las siguientes imágenes son capturas de pantalla de la ejecución del *curl* desde las máquinas de test en Europa (primera) y EE. UU. (segunda). Puede observarse como la primera responde con un mensaje de saludo desde Europa (*EU*) y la segunda con un mensaje de saludo desde EE. UU. (US).

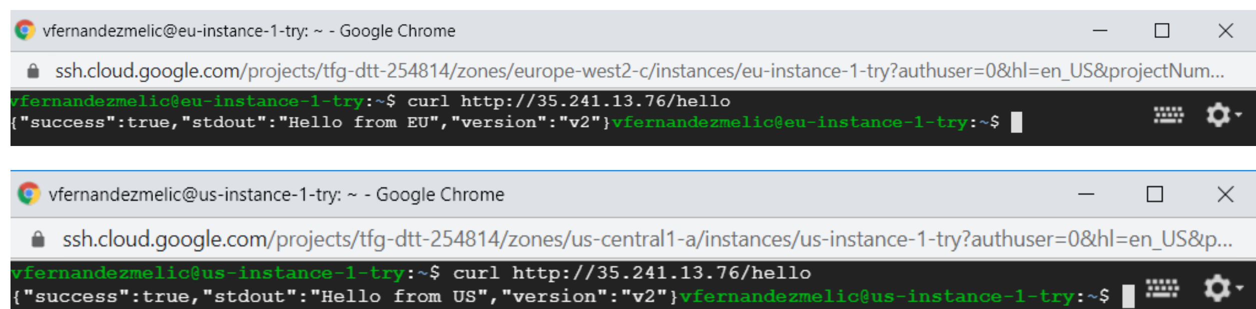

Figura 6: Petición curl a IP del servicio desde infraestructura de test.

Se puede ver cómo desde la instancia *eu-instance-1-try* situada en la región *europe-west2* contesta el *cluster* de Europa (*EU*) y desde la instancia *us-instace-1-try* situada en la región *us-central1-a* contesta el *cluster* de EE. UU. (*US*).

Esta prueba se realizó en repetidas ocasiones y siempre recuperó el mismo resultado. Posteriormente, se apagó el *cluster* de Europa y se certificó que las peticiones pasaban a mandarse al único *cluster* activo en funcionamiento, el de EE. UU.

Se trata de una prueba muy concreta que no requiere de mucha infraestructura extra y que permite verificar con éxito la resiliencia, la IP única global y la respuesta del *cluster* más cercano.

5.2 Análisis de latencia:

La siguiente prueba que se realizó fue el análisis de latencia, con el objetivo de verificar que el tiempo de latencia en el acceso a los servicios era menor de 200ms, y si existe diferencia real entre tener uno o dos *clusters* levantados.

El escenario del test consiste en el uso de la herramienta *Stackdriver* para mandar paquetes *ICMP* de manera continuada a la IP del servicio desde diferentes puntos del planeta, y calcular la media. Primero con un único *cluster* levantado y después con los 2.

Estas pruebas se repitieron varias veces, arrojando resultados muy similares. A continuación se muestra el resultado de una de ellas.

	Un *cluster*: EE.UU.	Dos *clusters*: EE. UU. y Europa
Singapur	371.07 ms	190.62 ms
Bélgica	202.69 ms	8.25 ms
Brasil	288.19 ms	147.08 ms
Iowa	6.51 ms	6.568 ms
Oregón	40.85 ms	41.14
Virginia	30.35 ms	30.15

Tabla 2: Latencia media de acceso a la IP global del servicio desde diferentes puntos del planeta.

Como se puede observar en la tabla anterior, los lugares que están más lejos de EE. UU. reducen notablemente el tiempo de latencia: Singapur, que con un solo cluster levantado quedaba cercano a los 400ms (muy lejos de los 200ms marcados por los requisitos), consigue mejorar hasta los poco más de 190ms. Brasil pasa de 288ms a 147ms, y el caso más notable es el de Bélgica, que pasa de 202ms a 8.25ms. Por otro lado, las latencias de Iowa, Oregón y Virginia se mantienen prácticamente iguales, ya que son ciudades de EE. UU. que van al mismo cluster que antes y por lo tanto sufren el mismo retardo.

Se puede concluir sin duda que el uso de 2 clusters repartidos geograficamente beneficia la latencia media y sirve como medida para intentar mantenerse por debajo de los 200ms.

5.3 Elasticidad y escalado:

Se han establecido como requisitos fundamentales del TFG, la escalabilidad y la elasticidad, y conviene repasar sus diferencias: La escalabilidad de un sistema es la capacidad de crecer para adaptarse al aumento del trabajo, mientras que la elasticidad es la capacidad de adaptarse al trabajo de cada momento, creciendo o disminuyendo, de tal forma que sea capaz de gestionar dicho trabajo pero sin desperdicio de recursos. Elasticidad implica escalabilidad, pero no a la inversa.

Cuantificar la elasticidad de un sistema es complicado, ya que no existen métricas estandarizadas para hacerlo [11]. Por ello, con esta prueba se pretenden dos cosas:

- Observar cómo el sistema consigue responder sin fallos ante un aumento del tráfico entrante (escalabilidad): mediante el número de peticiones exitosas y erróneas.
- Detectar cómo el sistema incrementa los recursos que dedica durante la ejecución de la prueba, y como los reduce una vez que la prueba ha terminado (elasticidad): mediante los logs de *kubernetes*, que indicarán el número de *pods* dedicados a esta tarea en cada momento.

Así pues, se tomó la decisión de aplicar los conocimientos aprendidos hasta el momento y se desplegó un entorno experimental en un nuevo *cluster* de *kubernetes* de test. Se realizó una investigación de las posibilidades existentes y se seleccionó la herramienta *JMeter* para realizar las pruebas de carga (lanzamiento de peticiones) en modo *master-worker*, *InfluxDB* como base de datos orientada a series temporales para almacenar los resultados y *Grafana* para visualizarlos [11]. Esta arquitectura de test (que se muestra en la siguiente figura) es asimismo escalable, ya que permite la ejecución de pruebas de carga intensivas por su funcionamiento en modo *master-worker* y su independencia entre herramientas (microservicios).

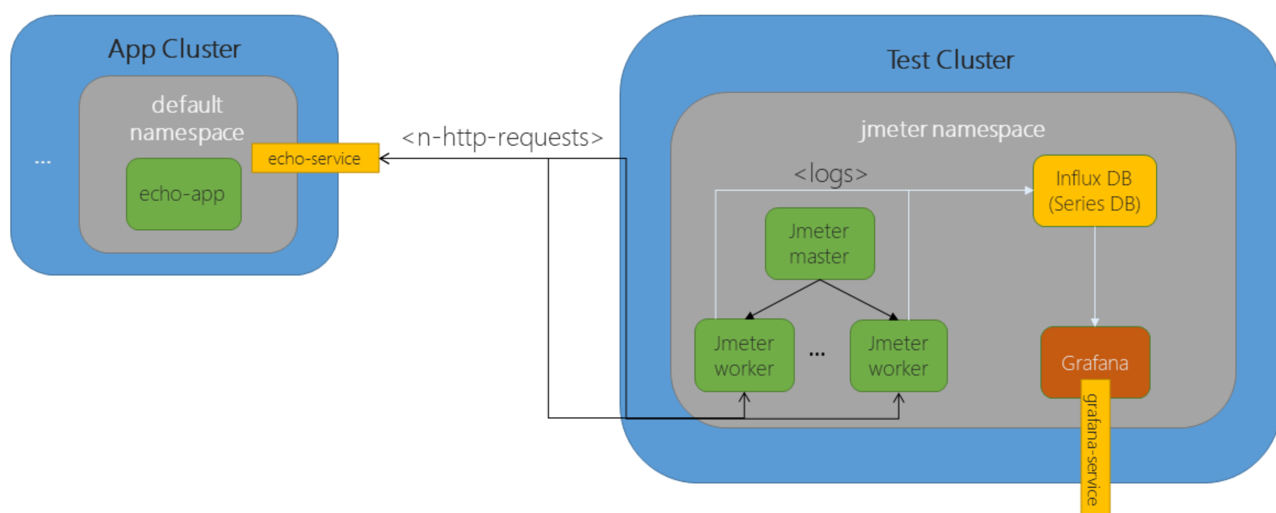

Figura 7: Arquitectura del escenario de test de elasticidad y escalado

Para evaluar los resultados vamos a mirar tanto los paneles de *Grafana* como los *logs* de *kubernetes* en *GCP*.

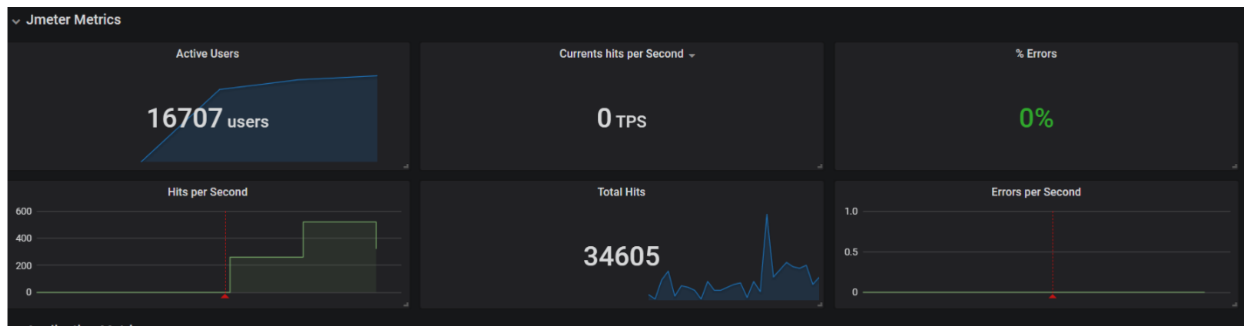

Figura 8: Métricas de rendimiento de las pruebas de cargas

En la captura anterior de las gráficas de *Grafana* se puede observar la evolución del test, cómo va aumentando paulatinamente el número de usuario y de peticiones por segundo hasta llegar a 16707 y 34605 respectivamente en el momento de la captura. Como se puede ver, hay un 0% de errores.

Esto nos demuestra una gran capacidad de absorción del tráfico por parte de nuestro servicio, pero no nos dice nada de la elasticidad de la aplicación por sí solo. Para verificar este punto, hay que revisar los logs de *kubernetes* en *GCP* durante las pruebas de carga. En la siguiente imagen se pueden ver una serie de eventos que indican paso por paso, como *kubernetes* escala el número de réplicas de la aplicación (*pods*) primero de 1 a 2, luego de 2 a 3, y posteriormente de 3 a 2 y de 2 a 1 cuando baja la carga de trabajo.

echo-app

Overview Details Revision history Events YAML

Message	Reason
Scaled up replica set echo-app-5f9c78c5cc to 2	ScalingReplicaSet
Scaled up replica set echo-app-5f9c78c5cc to 3	ScalingReplicaSet
Scaled down replica set echo-app-5f9c78c5cc to 2	ScalingReplicaSet
Scaled down replica set echo-app-5f9c78c5cc to 1	ScalingReplicaSet

Captura 9: Logs de kubernetes en GCP para la aplicación echo-app

Los logs de *kubernetes* de la captura anterior muestran elasticidad: Se pasa de una a tres réplicas cuando aumentan las peticiones por segundo y luego de tres a una de nuevo cuando baja el tráfico.

Se puede concluir pues, tras revisar los resultados de las dos pruebas anteriores y ver como efectivamente el sistema responde aumentando y disminuyendo el número de recursos dedicados en función de la carga de trabajo, que el sistema es escalable y elástico.

5.4 Actualización y *Rollback* con *pipelines*

Se ha comentado muchas veces la importancia de poder actualizar la aplicación y volver atrás (hacer *rollback*) en caso de que algo no funcione como se esperaba. Probar esta característica es tan sencillo como realizar un despliegue de una versión de la aplicación, después un nuevo despliegue con una versión actualizada, y finalmente ejecutar manualmente el *pipeline* de la primera versión para volver atrás.

El escenario de esta prueba debe contar con dos repositorios (uno para el código de la aplicación y otro para el entorno) y un pipeline de desarrollo en *GCP* (tal y como se explica en el capítulo 4). Las dos versiones de la aplicación que se usan tan solo varían en el *id* de versión que muestran, pero esto es más que suficiente para verificar el funcionamiento.

A continuación se muestran una serie de imágenes del resultado de la prueba que ilustran su correcto funcionamiento, pero el proceso completo se muestra en el anexo V por cuestión de concisión.

Primer paso: Despliegue de la versión "v1":

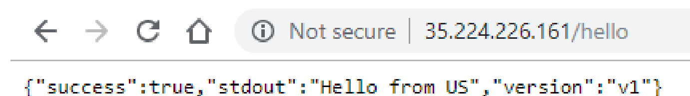

Figura 10: Resultado tras el despliegue de la versión "v1"

Segundo paso: Despliegue de la versión "v2":

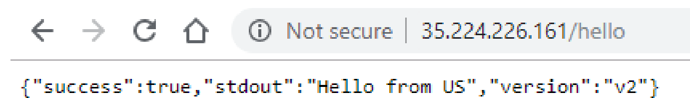

Figura 11: Resultado tras el despliegue de la versión "v2"

Tercer paso: Rollback, vuelta a la versión "v1":

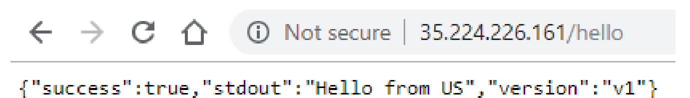

Figura 12: Resultado tras rollback a la versión "v1"

Por lo tanto, mediante esta prueba, se confirma la facilidad para actualizar y hacer *rollback* a versiones anteriores cuando se trabaja con pipelines de integración continua.

5.5 Testeo de diferentes versiones en producción

Por último, para verificar la capacidad de testear versiones diferentes en producción de manera simultánea, de tal forma que un segmento de usuarios utilice una y el resto de los usuarios utilice otra, se han lanzado con *kubernetes* las dos versiones, asociadas al mismo servicio de entrada de tráfico, y en la siguiente captura de pantalla se puede observar el correcto funcionamiento de la prueba.

Labels	app : echo-app
Stackdriver logs	echo-app , echo-app-v2
Type	LoadBalancer
External endpoints	34.77.179.21:80

LoadBalancer

Cluster IP	10.0.8.36
Load balancer IP	34.77.179.21
Load balancer	a87438d24150311eaa54242010a84006

Deployments

Name	Status	Pods
echo-app	OK	1/1
echo-app-v2	OK	1/1

Carga repartida al 50% entre ambas versiones

Serving pods

Name	Status	Restarts	Created on
echo-app-5f9c78c5cc-2gq4v	Running	0	Dec 2, 2019, 1:59:08 PM
echo-app-v2-74f5b65574-5lbh9	Running	0	Dec 2, 2019, 3:19:29 PM

Figura 13: Detalle del servicio de entrada a la aplicación. Reparte la carga a partes iguales entre las dos versiones de la aplicación.

Se trata del detalle del servicio de entrada del tráfico, y se puede observar cómo las peticiones se reparten a partes iguales entre las dos versiones de la aplicación.

De esta forma se comprueba que efectivamente, la arquitectura propuesta es capaz de testear diferentes versiones en producción de manera simultánea.

6. Conclusiones

En este trabajo se ha expuesto la necesidad cada vez más habitual de las empresas de desarrollar, actualizar y mantener aplicaciones para dar servicio a sus clientes a escala global, y se ha visto que esto supone retos que de no resolverse correctamente, pueden suponer pérdidas económicas por el desperdicio de recursos derivado de la sobrestimación de la demanda o la pérdida de clientes por culpa de las altas latencias, la denegación de servicio o los fallos *hardware* y *software*.

Para resolver estos problemas, en base a una serie de requisitos, se ha propuesto una solución no solo tecnológica, sino también metodológica, consistente en aprovechar las características que ofrece la nube así como adoptar la metodología *DevOps* como forma de trabajo sin olvidarnos de mantener un presupuesto ajustado dentro de lo posible.

Así pues, se ha propuesto el uso de una arquitectura de referencia *multi-cluster* con IP única global desplegada en la nube, el uso de *kubernetes* y una arquitectura de microservicios, así como el uso de las herramientas y prácticas de la metodología *DevOps*, que consiste en la automatización del flujo de desarrollo de *software* para agilizar las entregas a producción y evitar errores.

Posteriormente, se ha implementado la solución propuesta en la nube de *Google Cloud Platform* para comprobar que se trata de una solución plausible. Se ha desarrollado, desplegado, actualizado y mantenido una aplicación de ejemplo desplegada sobre la infraestructura de referencia y se ha llevado a cabo la federación de los *clusters* de *kubernetes* para que compartan un única IP asociada a un balanceador de carga global que permite reducir la latencia de acceso desde diferentes partes del mundo. Por último, se han realizado con éxito una serie de pruebas para verificar la viabilidad y el correcto funcionamiento de todo el sistema.

Concretamente, se ha verificado el acceso al sistema mediante IP única global con redirección al *cluster* más cercano disponible en cada momento, mediante el lanzamiento de peticiones desde diferentes máquinas virtuales desplegadas en la infraestructura de test (resiliencia). Se ha logrado reducir las latencias de acceso desde diferentes partes del mundo a menos de 200 ms gracias al despliegue de varios *clusters* (reducción de la latencia). Se han hecho pruebas de cargas para verificar la elasticidad y la escalabilidad mediante la construcción de un entorno de test capaz de generar tráfico hacia el sistema, recoger los resultados mediante una base de datos orientada a series temporales y mostrarlos mediante la herramienta *Grafana* así como con los propios logs de *kubernetes*. También se han realizado pruebas de actualización y *rollback* (corrección de errores), así como de testeo de diferentes versiones en producción mediante la creación y uso de pipelines de ejecución.

Otros requisitos se han solucionado por la propia construcción del sistema: el uso de *kubernetes* e *Istio* permite actualizaciones con tiempos mínimos de parada y monitorización constante del sistema y sus comunicaciones. El uso de pipelines de ejecución y control de versiones permite el despliegue de diferentes versiones en zonas geográficas distintas. El mecanismo de pago por uso de la nube y la

metodología *DevOps* en general, con su filosofía de automatizar procesos y evitar errores, permite mantener los costes ajustados dentro de la consecución del resto de objetivos y requisitos.

Es interesante comentar que existen limitaciones a esta solución, y la principal de ellas es la alta volatilidad de los servicios en la nube, en el sentido de evolución constante y obsolescencia de tecnologías. Si ya se trata de un problema importante en la informática en general, cuando se habla de la nube, el problema se agrava porque muchas veces las soluciones tecnológicas que se ofrecen son específicas de los proveedores de la nube, poco interoperables y pueden atarte a ella (*vendor lock-in*) [13], no siempre tienen garantías de mantenimiento a largo plazo y pueden sufrir cambios no retrocompatibles. Por esta razón, en este TFG se ha procurado elegir tecnología estándar cuando esto era posible, por ejemplo con el uso de *kubernetes*. Todo el trabajo desarrollado en esta dirección es directamente trasladable a cualquier otro *cluster* de *kubernetes* en cualquier proveedor de la nube.

La dedicación a este TFG se ha cuantificado en torno a las 500 horas de trabajo, sin tener en cuenta la escritura de la memoria, todos los conocimientos aprendidos fuera del contexto académico y del TFG, como muchos servicios de la nube, metodologías de desarrollo y conocimiento de la realidad en el mundo laboral. A continuación se muestra un diagrama de Gantt sobre la organización temporal y reparto de las tareas que se ha seguido durante el desarrollo de este TFG desglosado por meses.

Tarea	septiembre-19	octubre-19	noviembre-19	diciembre-19	enero-20	febrero-20	marzo-20	abril-20	mayo-20	junio-20	julio-20	agosto-20
Planteamiento del problema												
Investigación y documentación												
Metodología DevOps												
Kubernetes												
Arquitectura de referencia												
Google Cloud Platform												
Despliegue de la infraestructura												
Infraestructura como código												
Pipelines de ejecución												
Pruebas de resiliencia												
Pruebas de latencia												
Pruebas de elasticidad y escalado												
Actualización y rollback												
Testeo de diferentes versiones												
Escritura de la memoria												
Covid-19 y fin de curso												

Tabla 3: Diagrama de Gantt sobre el reparto temporal de tareas de este TFG.

Así pues, los primeros dos meses y medio se dedicaron principalmente al planteamiento del escenario de partida, a la investigación y documentación sobre las diferentes tecnologías y metodologías (aunque esto se mantuvo constante durante todo el desarrollo), así como a definir una arquitectura de referencia. En total, estas primeras tareas supusieron en torno a un 40% del total de la dedicación. Después se pasó a realizar el despliegue de la infraestructura, que supuso en torno a un 30% de la dedicación y por último se llevaron a cabo las pruebas de verificación, que supusieron el otro 30% de la dedicación. No se está teniendo en cuenta en estos porcentajes la dedicación final a la escritura de la memoria. En la siguiente figura se describe con más detalle el reparto temporal de las tareas.

Así pues, se considera que se han alcanzado todos los objetivos propuestos y de cara a continuar con este TFG, sería interesante profundizar en el tema de la infraestructura como código como parte de la metodología *DevOps* [14], llegando un paso más lejos y evaluando en profundidad herramientas como *Terraform*, que son *multi-cloud* y permiten la definición de recursos *hardware* de manera declarativa y la independencia de plataforma. Asimismo, sería interesante estudiar las nuevas opciones que han ido madurando este tiempo en lo referente al uso de *multi-clusters* y ver si son válidas, como es el caso de algunas características del *framework Istio* [15] que no se han explorado en este trabajo, a pesar de sí haber utilizado otras, como la parte referente a la monitorización.

Anexos

Anexo I: Código fuente de la aplicación de ejemplo y del fichero *Dockerfile*

Aplicación

```
const express = require('express');
const bodyParser = require('body-parser');

let app = express();

const port = process.env.PORT || 8000;

app.use(bodyParser.json());
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-
Type, Accept");
  next();
});

// Rutas: Ambos endpoints retornan el mismo resultado
app.get('/', (req, res) => {
    res.json({ success: true, stdout: "Hello world from " + process.env.ZONE, version: "v2"});
  }
)

app.get('/hello', (req, res) => {
    res.json({ success: true, stdout: "Hello from " + process.env.ZONE, version: "v2"});
  }
)

app.listen(port, () => console.log(`Server is listening on port: ${port}`));
```

Figura 14: Código Fuente de la aplicación de ejemplo utilizada para el despliegue en este TFG.

Dockerfile

```
FROM node:10

# Crear directorio para alojar la aplicación
WORKDIR /usr/src/app

# Copiar ficheros de definición de dependencias
# dentro del contenedor
COPY package*.json ./
```

```
# Instalar dependencias de la aplicación
RUN npm install

# Copiar fuentes de la app dentro del contenedor
COPY . .

EXPOSE 8000

# Primer comando a ejecutar al arrancar el contenedor
CMD [ "node", "server.js" ]
```

Figura 15: Código fuente del fichero Dockerfile para construir el contenedor y encapsular la aplicación de ejemplo

Anexo II: Scripts de aprovisionamiento de *hardware*

A continuación se muestra el código de 2 scripts que se utilizaron para el despliegue y eliminación de infraestructura *hardware* en GCP.

```
# ---

# Env Variables

# Cluster
PROJECT="tfg-dtt-254814"
CLUSTER_NAME="eu-cluster-1"
#CLUSTER_ZONE="us-central1-c"
CLUSTER_ZONE="europe-west1-c"
CLUSTER_REGION="europe-west1"
CLUSTER_VERSION="1.13.11-gke.14"
CLUSTER_NUM_NODES="4"

# Deployment
REPLACE_ENV_ZONE="EU"
REPLACE_PROJECT="tfg-dtt-254814"
REPLACE_NAMESPACE="default"
REPLACE_IMAGE="5f2e38cd1acc7fef0fc9d4d7d0b4c24484c30a78203076417fb380d33d6ffa20"
REPLACE_EXTERNAL_PORT="80"

# ---

# Create Cluster
gcloud beta container --project ${PROJECT} clusters create ${CLUSTER_NAME} --
zone ${CLUSTER_ZONE} --no-enable-basic-auth \
```

```
--cluster-version ${CLUSTER_VERSION} --machine-type "custom-1-5120" --image-
type "COS" --disk-type "pd-standard" --disk-size "100" \
--
scopes "https://www.googleapis.com/auth/devstorage.read_only","https://www.google
apis.com/auth/logging.write","https://www.googleapis.com/auth/monitoring","https:
//www.googleapis.com/auth/servicecontrol","https://www.googleapis.com/auth/servic
e.management.readonly","https://www.googleapis.com/auth/trace.append" \
--num-nodes ${CLUSTER_NUM_NODES} --enable-stackdriver-kubernetes --enable-
stackdriver-kubernetes --enable-ip-alias --
network "projects/${PROJECT}/global/networks/default" \
--
subnetwork "projects/${PROJECT}/regions/${CLUSTER_REGION}/subnetworks/default" --
default-max-pods-per-node "110" \
--addons HorizontalPodAutoscaling,HttpLoadBalancing,Istio --istio-
config auth=MTLS_PERMISSIVE --enable-autoupgrade --enable-autorepair

# ---

gcloud container clusters get-credentials "${CLUSTER_NAME}" --
zone "${CLUSTER_ZONE}" --project ${PROJECT}

# ---

# Generate custom YAML
sed "s/REPLACE_ENV_ZONE/${REPLACE_ENV_ZONE}/g" echo-app.yaml.tpl | \
sed "s/REPLACE_PROJECT/${REPLACE_PROJECT}/g" | \
sed "s/REPLACE_IMAGE/${REPLACE_IMAGE}/g" | \
sed "s/REPLACE_NAMESPACE/${REPLACE_NAMESPACE}/g" | \
sed "s/REPLACE_EXTERNAL_PORT/${REPLACE_EXTERNAL_PORT}/g" > echo-app.yaml

# Deploy echo-app
kubectl apply -f echo-app.yaml

# Delete customized file
rm echo-app.yaml
```

Figura 16: Código fuente de un script de aprovisionamiento de hardware en GCP.

```
# Env Variables

# Cluster
PROJECT="tfg-dtt-254814"
CLUSTER_NAME="us-cluster-1"
CLUSTER_ZONE="us-central1-a"
```

```
# ---

# Delete Cluster
gcloud beta container --project ${PROJECT} clusters delete ${CLUSTER_NAME} --
zone ${CLUSTER_ZONE}
```

Figura 17: Código fuente de un script para eliminar de hardware en GCP.

Anexo III: Federación de *clusters* de *kubernetes*

Para llevar a cabo la federación de los *clusters* de kubernetes, es necesario cumplir con las siguientes restricciones:

- El nombre de los servicios que vayan a ser incluidos en el *ingress* debe tener el mismo nombre en todos los *clusters*.
- El servicio debe estar en el mismo *namespace* en todos los *clusters*.
- El servicio tiene que ser de tipo *NodePort*.
- El servicio tiene que encontrarse en el mismo puerto en todos los *clusters*.

Para cada *cluster*, hay que aplicar los ficheros de configuración de las aplicaciones y los servicios que se quieran desplegar, cumpliendo las restricciones anteriormente descritas.

Una vez hecho esto, hay que reservar una IP estática global en *GCP*.

Por último, hay que crear la configuración de un *ingress* indicando los servicios de *backend* y aplicar la configuración mediante la herramienta *kubecmi*.

Anexo IV: Códgio pipelines de ejecución

A continuación se muestra el código de dos ejemplos de pipelines de ejecución: uno para el pipeline de la aplicación y otro para el del entorno.

```
# [START cbuild-delivery]
steps:

# This step clones the echo-env repository
- name: 'gcr.io/cloud-builders/gcloud'
  id: Clone env repository
  entrypoint: /bin/sh
  args:
  - '-c'
  - |
```

```
    gcloud source repos clone echo-env && \
    cd echo-env && \
    git checkout candidate && \
    git config user.email $(gcloud auth list --filter=status:ACTIVE --
format='value(account)')

# This step builds the container image.
- name: 'gcr.io/cloud-builders/docker'
  id: Build
  args:
  - 'build'
  - '-t'
  - 'gcr.io/$PROJECT_ID/echo-app:$SHORT_SHA'
  - '.'

# This step pushes the image to Container Registry
# The PROJECT_ID and SHORT_SHA variables are automatically
# replaced by Cloud Build.
- name: 'gcr.io/cloud-builders/docker'
  id: Push
  args:
  - 'push'
  - 'gcr.io/$PROJECT_ID/echo-app:$SHORT_SHA'
# [END cloudbuild]

# This step generates the new manifest
- name: 'gcr.io/cloud-builders/gcloud'
  id: Generate manifest
  entrypoint: /bin/sh
  args:
  - '-c'
  - |
     sed "s/GOOGLE_CLOUD_PROJECT/${PROJECT_ID}/g" echo-app.yaml.tmp | \
     sed "s/COMMIT_SHA/${SHORT_SHA}/g" > echo-env/echo-app.yaml

# This step pushes the manifest back to hello-cloudbuild-env
- name: 'gcr.io/cloud-builders/gcloud'
  id: Push manifest
  entrypoint: /bin/sh
  args:
  - '-c'
  - |
    set -x && \
    cd echo-env && \
    cat echo-app.yaml && \
```

```
    git add echo-app.yaml && \
    git commit -m "Deploying image gcr.io/${PROJECT_ID}/echo-app:${SHORT_SHA}
    Built from commit ${COMMIT_SHA} of repository echo-app
    Author: $(git log --format='%an <%ae>' -n 1 HEAD)" && \
    git push origin candidate

# [END cbuild-delivery]
```

Figura 18: Código fuente de un pipeline de ejecución para el despliegue del código de la aplicación.

```
# [START cbuild-delivery]
steps:

# This step deploys the new version of our container image
# in the hello-cloudbuild Kubernetes Engine cluster.
- name: 'gcr.io/cloud-builders/kubectl'
  id: Deploy
  args:
  - 'apply'
  - '-f'
  - 'echo-app.yaml'
  env:
  - 'CLOUDSDK_COMPUTE_ZONE=europe-west1-c'
  - 'CLOUDSDK_CONTAINER_CLUSTER=eu-cluster-1'

# This step copies the applied manifest to the production branch
# The COMMIT_SHA variable is automatically
# replaced by Cloud Build.
- name: 'gcr.io/cloud-builders/git'
  id: Copy to production branch
  entrypoint: /bin/sh
  args:
  - '-c'
  - |
    set -x && \
    # Configure Git to create commits with Cloud Build's service account
    git config user.email $(gcloud auth list --filter=status:ACTIVE --
format='value(account)') && \
    # Switch to the production branch and copy the kubernetes.yaml file from the
candidate branch
    git fetch origin production && git checkout production && \
    git checkout $COMMIT_SHA echo-app.yaml && \
    # Commit the kubernetes.yaml file
```

```
    git commit -m "Manifest from commit $COMMIT_SHA
    $(git log --format=%B -n 1 $COMMIT_SHA)" && \
    # Push the changes back to Cloud Source Repository
    git push origin production

# [END cbuild-delivery]
```

Figura 19: Código fuente de un pipeline de ejecución para la actualización del código del entorno.

Anexo V: Actualización y *rollback*

En la siguiente captura de pantalla puede verse cómo se mostraría en *GCP* el resultado de ejecución de un *pipeline* para la actualización del código de la aplicación. Cada uno de los pasos que se pueden observar en la imagen, corresponde con uno de los pasos definidos en el fichero de despliegue que se ejecuta en *Google Cloud Build*.

Build information

Status	Build successful
Build id	3de0af73-b25e-4a85-bb3e-a786ef493657
Image	–
Trigger	Push to master branch (Push to master branch)
Source	Cloud Source Repository echo-app
Git commit	933b627f6c144eb742ba1f0d26d46f4acb948031
Started	November 20, 2019 at 5:27:04 PM UTC+1
Duration	1 min 45 sec

Build steps expand all

QA Analysis - SonarQube 44 sec

gcr.io/cloud-builders/gcloud — -c "gcloud container clusters get-credentials us-cluster-1 --zone us-central1-a --project tfg-dtt-254814 && \ kubectl port-forward --namespace tools $(kubectl get pod --namespace tools --selector="app.kubernetes.io/component=sonarqube,app.kubernetes.io/name=sonarqube" --output jsonpath='{.items[0].metadata.name}') 8080:9000 & > /dev/null apt update && \ apt install unzip && \ apt install wget && \ wget https://binaries.sonarsource.com/Distribution/sonar-scanner-cli/sonar-scanner-cli-4.2.0.1873-linux.zip && \ unzip sonar-scanner-cli-4.2.0.1873-linux.zip && \ gcloud source repos clone echo-app && \ cd echo-app && \ git config user.email $(gcloud auth list --filter=status:ACTIVE --format='value(account)') && \ ./../sonar-scanner-4.2.0.1873-linux/bin/sonar-scanner -X \ -Dsonar.projectKey=echo-app \ -Dsonar.sources=. \ -Dsonar.host.url=http://127.0.0.1:8080 \ -Dsonar.login=395acbd29ff6dca88de4b6202ca363d3e2e99e01 "

Clone env repository 4 sec

gcr.io/cloud-builders/gcloud — -c "gcloud source repos clone echo-env && \ cd echo-env && \ git checkout candidate && \ git config user.email $(gcloud auth list --filter=status:ACTIVE --format='value(account)') "

Build 28 sec

gcr.io/cloud-builders/docker — build -t gcr.io/tfg-dtt-254814/echo-app:933b627 .

Push 17 sec

gcr.io/cloud-builders/docker — push gcr.io/tfg-dtt-254814/echo-app:933b627

Generate manifest 0.9 sec

gcr.io/cloud-builders/gcloud — -c "sed "s/GOOGLE_CLOUD_PROJECT/tfg-dtt-254814/g" echo-app.yaml.tmp | \ sed "s/COMMIT_SHA/933b627/g" > echo-env/echo-app.yaml "

Push manifest 3 sec

gcr.io/cloud-builders/gcloud — -c "set -x && \ cd echo-env && \ cat echo-app.yaml && \ git add echo-app.yaml && \ git commit -m "Deploying image gcr.io/tfg-dtt-254814/echo-app:933b627 Built from commit 933b627f6c144eb742ba1f0d26d46f4acb948031 of repository echo-app Author: $(git log --format='%an <%ae>' -n 1 HEAD)" && \ git push origin candidate "

Figura 20: Resultado de ejecución del pipeline para actualizar el código de la aplicación.

La ejecución correcta de este *pipeline* dispara la ejecución del siguiente, que actualiza el entorno.

Build information

Status	Build successful
Build id	230886eb-4576-4c2f-82cb-0f1f864bf8e8
Image	–
Trigger	Push to candidate branch (Push to candidate branch)
Source	Cloud Source Repository echo-env
Git commit	155022fad431180409fb6b516527c1c955189948
Started	November 20, 2019 at 5:28:48 PM UTC+1
Duration	23 sec

Build steps expand all

Deploy 8 sec

gcr.io/cloud-builders/kubectl — apply -f echo-app.yaml

Copy to production branch 5 sec

gcr.io/cloud-builders/git — -c "set -x && \ # Configure Git to create commits with Cloud Build's service account git config user.email $(gcloud auth list --filter=status:ACTIVE --format='value(account)') && \ # Switch to the production branch and copy the kubernetes.yaml file from the candidate branch git fetch origin production && git checkout production && \ git checkout 155022fad431180409fb6b516527c1c955189948 echo-app.yaml && \ # Commit the kubernetes.yaml file with a descriptive commit message git commit -m "Manifest from commit 155022fad431180409fb6b516527c1c955189948 $(git log --format=%B -n 1 155022fad431180409fb6b516527c1c955189948)" && \ # Push the changes back to Cloud Source Repository git push origin production "

Figura 21: Resultado de ejecución del pipeline para actualizar el entorno y desplegar la aplicación.

Con esto, la nueva versión de la aplicación estaría desplegada. Para volver a la versión anterior, habría que reejecutar una versión anterior del entorno de manera manual. En la siguiente imagen se puede ver como se despliegan (de abajo hacia arriba) aplicación y entorno de una versión, después aplicación y entorno de una segunda versión, y finalmente se vuelve a una versión anterior del entorno (commit 15022f). Nótese que no es necesario deshacer el commit del código en el control de versiones, sino tan solo desplegar en producción el contenedor que encapsula la versión anterior de la aplicación.

Build	Source		Git commit
afc4eb38-28ab...	Cloud Source Repository echo-env	*Rollback*	155022f
4ba4ece0-20ef...	Cloud Source Repository echo-env	Versión 2	2b9c0de
4586e242-c3f2...	Cloud Source Repository echo-app		adaa0cc
230886eb-4576...	Cloud Source Repository echo-env	Versión 1	155022f
3de0af73-b25e...	Cloud Source Repository echo-app		933b627

Figura 22: Actualización y rollback de una aplicación en producción.

Referencias:

[1] Puerto Becerra, Doria Patricia, La globalización y el crecimiento empresarial a través de estrategias de internacionalización. Pensamiento & Gestión [Internet]. 2010; (28):171-195. Recuperado de: https://www.redalyc.org/articulo.oa?id=64615176009

[2] A. Aviziens, "Fault-Tolerant Systems," in IEEE Transactions on Computers, vol. C-25, no. 12, pp. 1304-1312, Dec. 1976, doi: 10.1109/TC.1976.1674598.

[3] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, 2017, pp. 243-246, doi: 10.1109/ICSAW.2017.11.

[4] C. Ebert, G. Gallardo, J. Hernantes and N. Serrano, "DevOps," in IEEE Software, vol. 33, no. 3, pp. 94-100, May-June 2016, doi: 10.1109/MS.2016.68.

[5] A. Balalaie, A. Heydarnoori and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," in IEEE Software, vol. 33, no. 3, pp. 42-52, May-June 2016, doi: 10.1109/MS.2016.64.

[6] Kubernetes. 2020. Performing A Rolling Update. [online] Disponible en: https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

[7] "Performance and Scalability", Istio, 2020. [Online]. Available: https://istio.io/latest/docs/ops/deployment/performance-and-scalability/.

[8] "Architecture", Istio, 2020. [Online]. Available: https://istio.io/latest/docs/ops/deployment/architecture/.

[9] "Deploying Ingress across clusters | Kubernetes Engine Documentation", Google Cloud, 2020. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/how-to/ingress-for-anthos.

[10] "Guías prácticas | Documentación de Cloud Build | Google Cloud", Google Cloud, 2020. [Online]. Available: https://cloud.google.com/cloud-build/docs/how-to?hl=es.

[11] Nikolas Roman Herbst, Samuel Kounev, Ralf Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not". In 10th International Conference on Autonomic Computing (ICAC 13) 2013 (pp.

23–27). USENIX Association.

[12] "kubernauts/jmeter-kubernetes", GitHub, 2020. [Online]. Available: https://github.com/kubernauts/jmeter-kubernetes.git.

[13] J. Opara-Martins, R. Sahandi and F. Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing," International Conference on Information Society (i-Society 2014), London, 2014, pp. 92-97, doi: 10.1109/i-Society.2014.7009018.

[14] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero and D. A. Tamburri, "DevOps: Introducing Infrastructure-as-Code," 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, 2017, pp. 497-498, doi: 10.1109/ICSE-C.2017.162.

[15] "Replicated control planes", Istio, 2020. [Online]. Available: https://istio.io/latest/docs/setup/install/multicluster/gateways/. [Accessed: 30- Aug- 2020]