CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Diseño e implementación de un sistema de Continuous Delivery en Google Cloud Platform

Autor: Francisco Javier Roger Bauzá

Director: Tomás San Feliu Gilabert

MADRID, JUNIO DE 2019

Resumen

La integración, entrega y despliegue continuos son conceptos cada vez más comunes en el ámbito del desarrollo del software, en un contexto en el cual las compañías centran sus esfuerzos en alcanzar la máxima calidad en el producto empleando técnicas que favorecen la colaboración, comunicación e integración entre los componentes del equipo de desarrollo.

Sin embargo, Duvall et. al. con *Integración continua* (2007) y Humble et. al. con *Continuous Delivery* (2010) son los únicos autores que han abordado este tema. Existe más información disponible de conferencias y estudios científicos, pero se encuentra muy dispersa y difícil de recopilar.

Por ello, muchas empresas tienen problemas a la hora de aplicar estas prácticas. A pesar de ser conscientes de los beneficios que conlleva su utilización, la poca flexibilidad y la complejidad de sus entornos conlleva frecuentemente a no saber cómo adaptar estas pautas a sus propios proyectos.

Este Trabajo de Fin de Grado describe la implementación de un sistema de Entrega Continua en la empresa española, *OrbitalAds*. Dicho proyecto se desarrolla con el objetivo de adaptar todos los servicios alojados en Google Cloud Platform a la práctica de Entrega Continua, estudiando las herramientas disponibles y la forma óptima de diseñar el sistema.

Palabras clave: Entrega Continua, Integración Continua, Google Cloud Platform, Docker, Contenedores

Abstract

Continuous integration, delivery and deployment are increasingly common concepts in software development, in a context in which companies focus their efforts on achieving the highest product quality using techniques that encourage collaboration, communication and integration between the components of the development team.

However, Duvall et. al. with *Continuous Integration* (2007) and Humble et. al. with *Continuous Delivery* (2010) are the only authors who have addressed this issue. There is more information available from scientific conferences and studies, but it is scattered and difficult to compile.

As a result, many companies have problems applying these practices. Despite being aware of the benefits of their use, the lack of flexibility and complexity of their environments often leads to not knowing how to adapt these guidelines to their own projects.

This End of Degree Project describes the implementation of a Continuous Delivery system in the Spanish company, *OrbitalAds*. This project is developed with the aim of adapting all services hosted on Google Cloud Platform to the practice of Continuous Delivery, studying the tools available and the best way to design the system.

Keywords: Continuous Delivery, Continuous Integration, Google Cloud Platform, Docker, Containers

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.
- John F. Woods -

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

GLOSARIO

DockerHub DockerHub es un repositorio público en la nube, similar a Github, para distribuir los contenidos. Está mantenido por Docker y hay multitud de imágenes, de carácter gratuito, que se pueden descargar y utilizar.

Keyword Palabra o grupo de palabras que describen un producto o servicio para ayudar a determinar cuándo y dónde puede aparecer un anuncio.

OrbitalAds Empresa española que aplica tecnologías de Inteligencia Artificial y Procesamiento del Lenguaje Natural al targeting semántico en Google Ads Search.

Pipeline Término inglés que puede traducirse como "tubería". Este tipo de flujo de trabajo implica que la salida de una fase es la entrada de otra. Así, las diversas fases se encadenan a modo de tubería.

Release Una release es la distribución de la versión final de un producto de software. Generalmente, una release constituye la versión inicial del producto o una actualización de la misma.

LISTA DE ACRÓNIMOS

CI/CD Integración Continua/Entrega Continua.

GCP Google Cloud Platform.

IA Inteligencia Artificial.

NLP Procesamiento de Lenguaje Natural.

OKR Objectives and Key Results.

SEM Search engine marketing.

VCS Sistema de Control de Versiones.

XP Extreme Programming.

1. INTRODUCCIÓN

Combinar el trabajo de un equipo de desarrollo no es una operación fácil. Los sistemas de software pueden ser grandes y complejos, y un cambio aparentemente simple e independiente del resto de funcionalidades en una parte de dicho sistema puede provocar consecuencias imprevistas que comprometen la integridad del mismo.

Como consecuencia de esto, algunos equipos de desarrollo hacen que cada programador trabaje de manera aislada, en su propia rama, manteniendo una base de código estable o rama principal para evitar conflictos en la integración de código. Sin embargo, con el paso del tiempo las bifurcaciones con las que trabaja cada uno acaban divergiendo, provocando que la unificación de éstas en la base principal pueda llegar a ser problemática, requiriendo una cantidad significativa de trabajo adicional para resolver dichas divergencias.

Este proceso puede ser caro e impredecible. En equipos de desarrollo de multitud de personas, la integración de numerosas ramas puede requerir varias rondas de pruebas y validación para garantizar que el sistema funcionará como se desea. El problema se incrementa exponencialmente a medida que los equipos crecen, o si la rama en cuestión no ha sido integrada en un largo período de tiempo.

Para asegurar que la base principal se encuentra en todo momento perfectamente funcional, y para evitar que las ramas de los desarrolladores no difieran considerablemente unas de otras, se plantea la solución de Integración Continua.

En este proyecto, se propone adoptar el enfoque de Entrega Continua, un planteamiento que incorpora los conceptos de Integración Continua, pero los amplía con el objetivo de automatizar, en la medida de lo posible, las tareas a realizar para llevar el código nuevo a un entorno de producción.

1.1. Área del trabajo

El sistema de Entrega continua se va a implementar en *OrbitalAds*, una empresa emergente española fundada en 2016 con el objetivo de desarrollar una plataforma tecnológica capaz de optimizar de forma automática y en tiempo real cualquier campaña de *Search engine marketing (SEM)*.

OrbitalAds aplica tecnologías de *Inteligencia Artificial (IA)* y *Procesamiento de Lenguaje Natural (NLP)* al targeting semántico, una técnica de búsqueda de datos en la cual el objetivo no es sólo encontrar palabras clave o *keywords*, sino determinar la intención y el significado contextual de las palabras que una persona está utilizando para la búsqueda, para así optimizar el rendimiento de las cuentas de *SEM* de sus clientes, haciendo foco en incrementar el volumen de búsquedas y maximizar los resultados de la inversión.

Figura 1: Logo de OrbitalAds

1.2. Motivación

El propósito de *OrbitalAds* al plantear la implantación de este sistema es mejorar la experiencia de los clientes con el producto y, en definitiva, la calidad de los servicios ofrecidos, rediseñando el proceso de entrega no solo a nivel de flujo de trabajo, sino a nivel de cultura de empresa, alineando *la tecnología, los procesos y las personas* de la organización para que la implementación e integración de nuevo código se convierta en una actividad optimizada y automatizada.

Concretamente, los aspectos que se desean mejorar son:

- **La tecnología:** La Entrega Continua se centra en la automatización. Puede ser la fase que más trabajo conlleva pero también la más fácil de lograr.

- **Los procesos:** Adaptarse a las herramientas más modernas reporta ventajas para las etapas de desarrollo, pero requiere revisar el flujo actual de trabajo y evaluar las partes que se deben mejorar, mantener o descartar para conseguir que la Entrega Continua sea efectiva.
- **Las personas:** La gestión del cambio en el equipo de desarrollo para poder afrontar la resistencia a la transformación de tareas cotidianas, con el objetivo de incrementar la productividad y reducir los riesgos y el tiempo invertido en ellas.

1.3. Definición del problema

La Entrega Continua es la capacidad de conseguir que cambios de cualquier tipo, como nuevas características, cambios de configuración, correcciones de errores o experimentos, lleguen a los clientes, de una forma rápida y segura. [2]

Sin embargo, existen ciertos antipatrones que hacen que esta rapidez y seguridad se vean mermadas. Los procesos manuales se pueden considerar uno de estos antipatrones; las aplicaciones modernas pueden ser complejas y difíciles de desplegar, teniendo que gestionar múltiples tecnologías o decenas de dependencias, en su gran mayoría, de terceros. La complejidad se traduce en errores potenciales, acentuados especialmente por la propensión a fallos humanos en una tarea manual.

Además, en *OrbitalAds* la seguridad en la integración de nuevos cambios se ve especialmente reducida por la desconfianza que generan los nuevos cambios en el código. Esto se debe, concretamente, a la falta de pruebas que validen la funcionalidad de las nuevas modificaciones.

Estos problemas se pueden solventar con una estrategia definida de pruebas para el sistema, que reporte inmediatamente cualquier fallo para que pueda ser arreglado lo antes posible. Sí, además, se libera al programador de las tareas más repetitivas, se podrá reducir el riesgo de los procesos manuales sin sacrificar la estabilidad y la confiabilidad del sistema.

La automatización tomará un papel fundamental para entender los beneficios de la adopción de la Entrega Continua, de forma que la mayor parte del tiempo se pueda invertir en el desarrollo del producto, reduciendo el riesgo y los costes, pero aumentando la calidad y la productividad.

1.4. Objetivos

El objetivo principal del trabajo consiste en analizar e implementar la práctica de Integración y Entrega Continua en los procesos de desarrollo de *OrbitalAds*, facilitando el trabajo del equipo y la gestión del proyecto.

Los objetivos específicos del proyecto vienen descritos a continuación:

1.4.1. Adaptación a metodologías ágiles de desarrollo de software

OrbitalAds, como empresa emergente que está desarrollando un producto desde cero, ve especialmente útil las metodologías ágiles orientadas a un sistema cuyos requisitos cambian rápidamente durante el proceso de implementación. Es por ello que en *OrbitalAds* se aplica al ciclo de desarrollo una de las metodologías ágiles más populares [1], Scrum, combinada con el uso de *Objectives and Key Results (OKR)* [14].

1.4.2. Análisis del estado del arte y herramientas actuales

OrbitalAds, consciente de los beneficios de la implementación de un sistema de Entrega Continua, requiere analizar las herramientas disponibles actualmente con el objetivo de poder adoptar la tecnología que mejor se adapte a los procesos de la organización, manteniendo la integración en Google Cloud Platform.

1.4.3. Analizar y diseñar el pipeline de Integración y Entrega Continua

El propósito de construir un *pipeline* de Integración y Entrega continuas es hacer visible cada parte del proceso de construcción, implementación, pruebas y despliegue de software a todos los integrantes del equipo de desarrollo.

1.4.4. Automatización de tareas

Con el fin de garantizar resultados de calidad y un funcionamiento correcto del producto, se planteará una continua supervisión gracias a la automatización del proceso de validación del sistema y todos los procedimientos manuales, evitando posibles fallos humanos y facilitando la evaluación de código de forma que se haga posible detectar errores lo antes posible.

1.4.5. Proposición de posibles mejoras

Se llevará a cabo un análisis del proyecto realizado, evaluando líneas futuras de trabajo que puedan complementar el proceso de desarrollo planteado en este documento.

1.5. Aproximación a la solución

Para poder eliminar los procesos manuales de los ciclos de desarrollo y de implementación con herramientas automatizadas de compilación, pruebas y despliegue de servicios en la nube como Google Compute Engine, Google Cloud Functions o Google Firebase, se estudiarán las herramientas de Entrega Conctinua que cuentan con integración en Google Cloud Platform, seleccionando la que mejor se ajuste a las necesidades de *OrbitalAds*.

1.6. Estructura del trabajo

- **Introducción:** En este capítulo se trata de dar una visión general sobre el tema que se va a tratar, planteando tanto el problema a resolver como los objetivos del proyecto.
- **Estado del arte:** En este capítulo se realiza una revisión y estudio del estado de la cuestión a tratar, analizando los conceptos principales del tema y las herramientas que serán utilizadas.
- **Planteamiento del problema:** Antes de entrar en el desarrollo de la solución, se presenta en este capítulo la descripción de los diferentes pasos y fases que componen la solución, así como las limitaciones y restricciones que puedan afectar a la misma.
- **Resolución:** En este capítulo se explica la solución planteada, cómo se ha desarrollado el proyecto con el fin de cumplir los objetivos propuestos inicialmente.
- **Conclusiones:** En este capítulo se exponen las conclusiones obtenidas tras finalizar el desarrollo del proyecto. También se incluyen propuestas de trabajo futuro y próximos pasos a seguir para continuar y mejorar la solución abordada.

Durante el desarrollo de este proyecto se hacen uso de anglicismos y acrónimos técnicos. Éstos aparecerán escritos en cursiva y explicados en el Glosario y Lista de Acrónimos.

2. ESTADO DEL ARTE

Desde hace más de diez años, las filosofías ágiles están revolucionando el desarrollo de software. Estas metodologías surgieron como respuesta a las limitaciones de las metodologías "pesadas", que habían sido concebidas originalmente como una adaptación a los procesos usados en los sectores de la fabricación y la construcción [3]. Es decir, se plantearon para entornos intensivos en el uso de materiales y maquinaria específica y donde los costes de hacer cualquier cambio cuando el proyecto estaba en un punto avanzado son inasequibles, exigiendo una exhaustiva definición inicial de requisitos y un diseño que no está abierto a cambios.

Sin embargo, el desarrollo de software, por su propia naturaleza, no se ajustaba correctamente a dichas condiciones: La complejidad de los problemas a resolver, la falta de concreción de los requisitos de un producto, o el continuo cambio tecnológico, entre otros, hacían muy complicado que aspectos como la solución planteada, los requisitos, el diseño o la arquitectura no cambiasen durante el proceso de desarrollo, problema que las metodologías tradicionales no podían encajar.

Por otra parte, los proyectos de software poseen algunas características particulares, como la modularidad de los componentes de desarrollo, la capacidad de crear prototipos y los bajos costes de realizar cambios, en comparación con otros tipos de proyectos, abriendo la posibilidad de idear nuevas metodologías, como *Extreme Programming (XP)*.

2.1. eXtreme Programming

XP es una metodología ágil para el desarrollo de software muy útil en proyectos cuyos requisitos cambian constantemente. Fue formulada por Kent Beck en 1999 [4], exponiendo que los cambios de requisitos durante el desarrollo eran un aspecto natural, inevitable e incluso deseable, siendo la capacidad de adaptarse a estos cambios en cualquier punto del ciclo de vida del proyecto una aproximación más realista que intentar definir todos los requisitos en la fase inicial y tener que invertir después esfuerzo en controlar los cambios en la especificación.

XP se puede definir como un conjunto de pasos de diversas prácticas, acopladas de forma que cada uno de los pasos del ciclo de vida sean flexibles, buscando la simplicidad y la retroalimentación continua.

Una de estas prácticas es la integración frecuente del código por parte de los desarrolladores, lo que dio lugar al concepto de Integración Continua.

2.2. Integración Continua

La Integración Continua es una práctica de desarrollo que forma parte del proceso de desarrollo de *XP*, muy popular gracias al concepto de *"integration hell"*. Este término se puede definir como la dificultad de combinar el trabajo de varias personas que han estado desarrollando nuevas características o realizando cambios en la misma base de código. Si los cambios son significativos, la unión del trabajo de todos ellos puede llegar a ser una operación realmente complicada.

El proceso de integración de software no es un problema nuevo. Aunque para proyectos pequeños de una sola persona y que no tenga muchas dependencias de terceros este inconveniente puede no ser tan importante, a medida que la complejidad aumenta, incluso solo añadiendo una persona más, hay una mayor necesidad de integrar y asegurar que todos los componentes de software funcionan como era esperado. Esperar a realizar la integración al final del proyecto puede conducir a todo tipo de problemas, retrasando entregas y aumentando los costes.

En su artículo *Continuous Integration* [5], Martin Fowler describe este concepto como "... una práctica de desarrollo de software en la cuál los miembros del equipo integran su trabajo de forma frecuente. Cada integración desencadena una ejecución de pruebas que permiten detectar errores lo mas rápido posible, reducir significativamente los problemas de integración y desarrollar software de manera más cohesiva."

Para llevar a cabo este objetivo, Fowler define una serie de ejercicios que se deben seguir para asegurar la eficiencia de un sistema de Integración Continua:

- Mantener un único repositorio de código fuente
- Automatizar la compilación del proyecto
- Automatizar la validación del proyecto
- La integración a la rama principal dispara la compilación
- Los errores deben arreglarse inmediatamente
- Mantener la compilación lo más rápida posible
- Realizar pruebas en una réplica del entorno de producción

2.2.1. Mantener un único repositorio de código fuente

Esta práctica aboga por el uso de un *Sistema de Control de Versiones (VCS)* para el código fuente del proyecto, una herramienta ideada para gestionar los cambios en el código fuente de los proyectos de software, de modo que se puedan recuperar versiones específicas más adelante, en caso de ser necesario.

Estas herramientas permiten la creación de ramas o bifurcaciones de la base de código, pero Fowler destaca que su uso debe ser minimizado. En vez de tener que gestionar distintas versiones de código simultáneamente, todo debe estar integrado en la rama principal.

2.2.2. Automatizar la compilación del proyecto

El proceso de compilación no siempre es un proceso trivial. Puede implicar el acceso a base de datos, mover archivos de lugar, etc. Sin embargo, es una tarea que puede ser automatizada, evitando así la ejecución de comandos extensos o la navegación entre distintas ventanas para preparar la ejecución de este paso.

2.2.3. Automatizar la validación del proyecto

Es posible que la compilación finalice con éxito, pero este hecho no garantiza que su funcionalidad sea la correcta. Actualmente, muchos de los errores pueden ser capturados en el momento de escribir el código, pero muchos otros pueden pasar inadvertidos.

Una buena forma de encontrar estos problemas rápida y eficientemente es incluir una fase automática de validación durante el proceso de compilación. Este proceso no es completamente fiable [6], pero la gran mayoría de fallos pueden ser detectados. Las pruebas en Integración Continua representan uno de los pilares fundamentales para garantizar el éxito de esta práctica [5]. Además, es esencial porque:

- Permite la detección temprana de fallos; hay menos tiempo para resolver un defecto, pero estos se encuentran rápidamente ya que su introducción puede haber sido reciente.
- Mejora la calidad del software, ya que las pruebas disminuyen la probabilidad de fallo del software.
- Facilita la integración; las pruebas deben pasar para que el código se integre, evitando así la introducción de nuevos defectos.

2.2.4. La integración a la rama principal dispara la compilación

Cuando los miembros del equipo de desarrollo integran sus cambios con la rama principal, tenemos la oportunidad de tener versiones compiladas del código que han sido probadas y validadas. Para ello, existen los gestores de Integración Continua, softwares que permiten automatizar la compilación de un proyecto para que se ejecuten cada vez que se integra el código en la base principal.

2.2.5. Los errores deben arreglarse inmediatamente

Una parte clave de la Integración Continua es que, cuando la compilación falle, se debe revisar y arreglar inmediatamente, ya que esta práctica se basa en el hecho de trabajar siempre sobre una base funcional y estable de código.

Frecuentemente, la manera más rápida de arreglar el fallo es revertir el último cambio realizado. A menos que la causa del fallo sea trivial y se detecte en el momento, lo más apropiado es revertir los cambios y tratar de arreglarlos antes de realizar de nuevo la integración.

Kent Beck añade que "nadie tiene una tarea de mayor prioridad que arreglar una compilación errónea". Esto no significa que todos los miembros del equipo tengan que dejar de hacer lo que están haciendo para arreglar la construcción, normalmente sólo se necesita que la persona que ha causado el fallo revise sus cambios para que todo vuelva a funcionar. Significa que los errores en la compilación son una tarea urgente y prioritaria.

2.2.6. Mantener la compilación lo más rápida posible

La idea detrás de este principio es que el equipo de desarrollo requiere una retroalimentación rápida: ¿Cuál es el impacto de los cambios que han hecho?, ¿Han introducido algún fallo? El problema es que para saber con seguridad si algo está fallando, es necesario ejecutar una batería completa de pruebas en entornos similares a los de producción. Cuantas más pruebas realice y cuanta más parecido sea el entorno de pruebas al de producción, más tiempo tardará en ejecutarse la compilación y más trabajo manual se necesitará para ejecutar la compilación en ese entorno.

Como muestra el gráfico, la duración y la complejidad de la compilación aumentan exponencialmente cuantas más pruebas se desean realizar. Esto limita la capacidad de obtener retroalimentación de alta calidad con rapidez, tal y como requiere la práctica descrita.

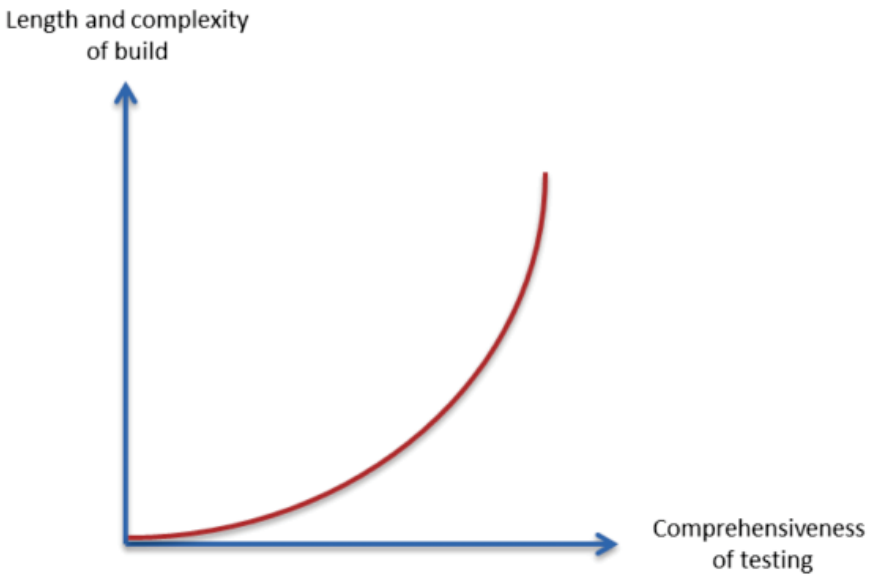

Figura 2: Comparativa de complejidad y duración vs exhaustividad de las pruebas [7]

2.2.7. Realizar pruebas en una réplica del entorno de producción

Un entorno de pruebas que difiere del de producción puede conducir a un software que se ha probado con éxito en la fase de desarrollo, pero que falla en producción. Sin embargo, construir una réplica de un entorno de producción tiene, en ocasiones, un coste prohibitivo. Por ello, un entorno de preproducción debería construirse de manera que sea una versión escalable del entorno de producción real, de forma que se reduzcan los costes pero se simule con gran exactitud el entorno final.

Si el grado de complejidad de producción es muy alto, es posible que el entorno sea difícil de replicar, pero se pueden tomar medidas para que la réplica sea lo más parecida posible. Por ejemplo, usando servicios virtualizados para obtener acceso a dependencias externas al control del desarrollador.

2.3. Entrega Continua

Jez Humble, autor del libro *continuous delivery: reliable software releases through build, test, and deployment automation*, define la Entrega Continua como "la habilidad de llevar cambios nuevos de cualquier tipo (características, configuraciones, correcciones, y experimentos) a producción o a las manos del usuario, sin peligro y rápidamente" [2].

La Entrega Continua es una extensión natural de la Integración Continua para asegurar que se pueden desplegar nuevos cambios en el entorno de producción de forma rápida y segura. Para ello, además de haber automatizado el proceso de pruebas y validación, hace falta automatizar también el ciclo de lanzamiento de una nueva versión.

La Entrega Continua permite plantear despliegues a producción en el momento que mejor se adapte a las necesidades de la empresa, pero siempre desde la premisa de que el código se encuentra en un estado "desplegable", independientemente de los cambios que se hayan hecho, evitando así problemas de integración y congelación de código.

Esta práctica puede ser extendida, a su vez, por el Despliegue Continuo. Con este ejercicio, cada cambio que pasa por todas las etapas del proceso anterior es liberado a los clientes. No hay intervención humana, y tan sólo una prueba fallida impedirá que se produzca un nuevo cambio en producción. Sin embargo, un lanzamiento puede tener un componente estratégico, o depender de otros servicios que deban actualizarse a la vez, por lo que esta práctica no tiene por qué ser la adecuada para todo tipo de organizaciones, como en el caso de este proyecto.

En pocas palabras, la Integración Continua es parte tanto de la Entrega Continua como del Despliegue Continuo. Y el Despliegue Continuo es como la Entrega Continua, con la excepción de que los despliegues a producción ocurren automáticamente.

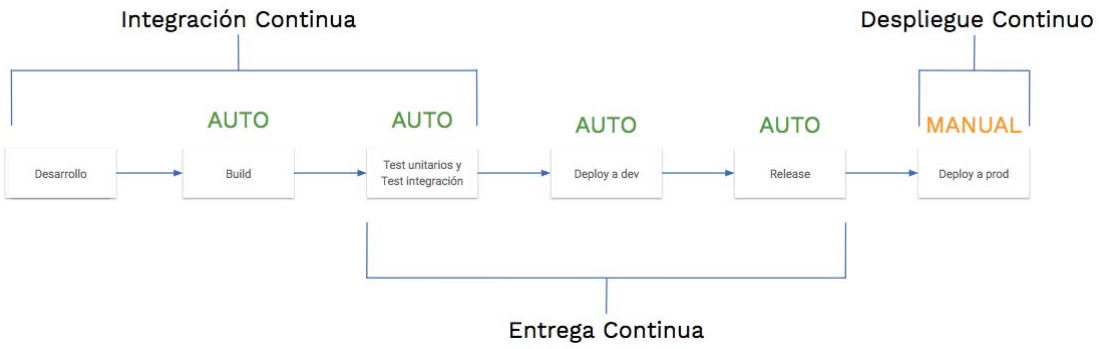

Figura 3: Comparativa de CI/CD

2.3.1. Pipeline de despliegue

El *pipeline* de despliegue es la columna vertebral de la Entrega Continua. También se conoce como *pipeline* de construcción o construcción escenificada, haciendo uso de todos los principios previamente discutidos, orquestándolos para funcionar en conjunto. Su propósito es automatizar grandes franjas del ciclo de vida del desarrollo de producto para la puesta en producción; desde la compilación, pasando por las pruebas, hasta el despliegue y el lanzamiento.Un ejemplo de *pipeline* se puede ver en la figura 4.

En el proceso de despliegue, cada cambio en el código debe ser un candidato a una nueva versión. El trabajo del *pipeline* de despliegue es la detección de problemas conocidos. Son conocidos porque se han tenido en cuenta a la hora de desarrollar las pruebas del sistema. Que la ejecución del *pipeline* no detecte ningún fallo otorga la confianza necesaria para lanzar la nueva versión con la seguridad de que todo funcionará como esperado. Si no fuese así, o si se detectasen errores más tarde, significa que el *pipeline* debe ser mejorado, posiblemente añadiendo nuevos casos de prueba o actualizando las ya existentes.

El objetivo es encontrar problemas lo antes posible, y que el tiempo desde que el desarrollador integra su código en la base principal hasta que el cambio llega a producción sea lo más corto posible. Por lo tanto, la paralelización de tareas tomará un papel importante para reducir tiempos y que la retroalimentación sea rápida. Asimismo, si se descubre algún defecto durante pruebas exploratorias manuales, significa que las pruebas automatizadas se deben mejorar.

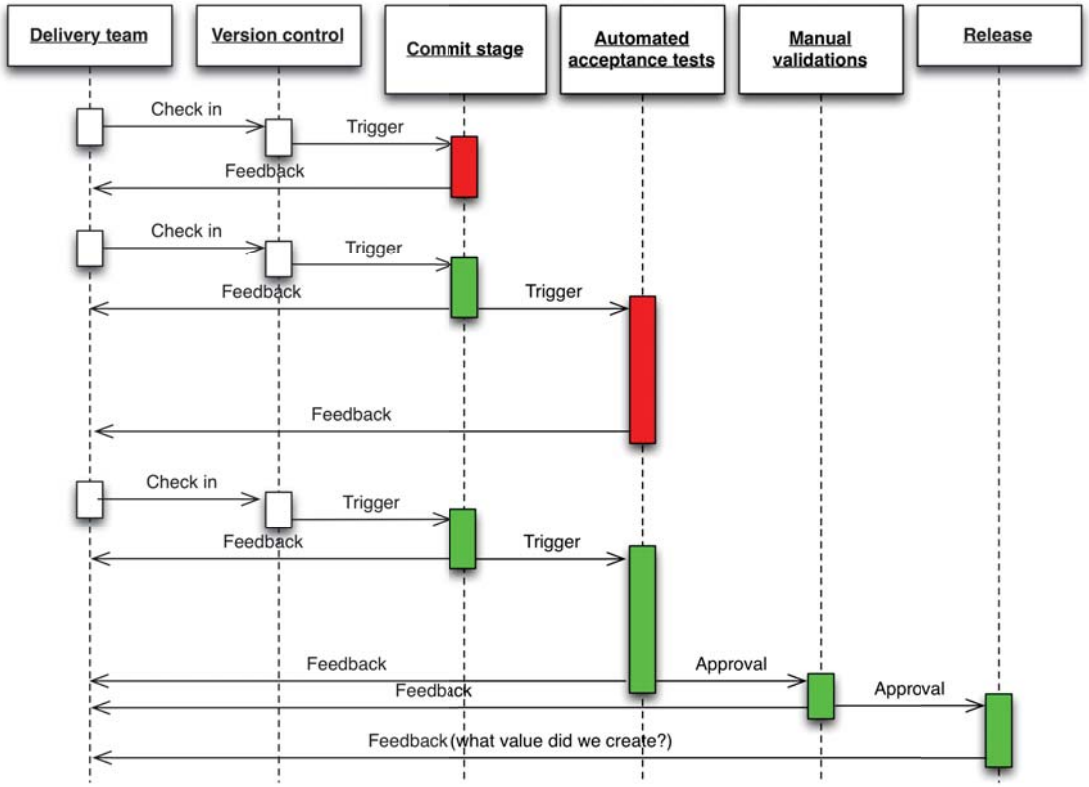

Figura 4: Etapas de un *pipeline* de despliegue
[2]

2.4. Google Cloud Platform

GCP es una colección de recursos informáticos de Google, que se encuentra a disposición del público a través de servicios como oferta pública de Cloud Computing.

Los recursos de *GCP* consisten en infraestructura de hardware físico (ordenadores, unidades de disco duro, unidades de estado sólido y conectividad y redes) contenidos en los centros de procesamiento de datos de Google distribuidos a nivel global, en los que cualquiera de los componentes se diseña de forma personalizada usando patrones similares a los disponibles en el Open Compute Project [10].

Este hardware se facilita a los clientes en forma de recursos virtualizados, por ejemplo, en máquinas virtuales (VM), como alternativa a la construcción y mantenimiento de su

Figura 5: Logo de *GCP*

propia infraestructura física.

Como oferta pública en la nube, los productos de software y hardware se proporcionan como servicios integrados que permiten el acceso a recursos subyacentes. *GCP* ofrece más de 50 servicios, incluyendo Infraestructura como Servicio (IaaS), Plataforma como Servicio (PaaS) y Software como Servicio (SaaS) en las categorías de computación, almacenamiento y bases de datos, redes, Big Data, seguridad, Machine Learning...

Estos servicios pueden utilizarse de forma independiente o combinándolos para que desarrolladores y profesionales de TI construyan su propia infraestructura personalizada basada en la nube.

GCP se aloja en la misma infraestructura subyacente que Google utiliza internamente para sus propios productos de usuario final, incluidos Google Search y YouTube.

2.5. Integración y Entrega Continua en Google Cloud Platform

Para poder llevar a cabo una entrega continua de actualizaciones de producto a los clientes, es necesario un proceso que compile, valide y actualice el software de forma automática. Los cambios en el código deben fluir a través de un *pipeline* que incluya la creación del artefacto a desplegar, pruebas unitarias y funcionales y la implementación del mismo en producción.

GCP ofrece una serie de servicios integrados con la plataforma, desarrollados para el uso de contenedores (sección 2.6)[11]:

2.5.1. Google Cloud Build

Google Cloud Build es un servicio que ejecuta la compilación de software en la propia infraestructura de Google Cloud Platform. Además, permite importar código fuente desde distintos *VCS*, como Google Cloud Storage, Cloud source Repositories, GitHub o Bitbucket, compilando los artefactos especificados, en este caso, contenedores.

Figura 6: Logo de Google Cloud Build

Google Cloud Build ejecuta la compilación como una serie de pasos de construcción, donde cada paso se ejecuta en un contenedor de Docker. Cada paso puede ejecutar cualquier cosa que se pueda hacer en un contenedor, independientemente del entorno. Para establecer los pasos del *pipeline* de construcción, se pueden utilizar pasos predefinidos por Google Cloud Build o personalizarlo para adaptar los pasos necesarios.

2.5.2. CircleCI

CircleCI es un servicio de Integración Continua basado en la nube. Soporta múltiples lenguajes de programación, entre ellos Python, y ha sido diseñado para reducir al máximo la sobrecarga producida durante el proceso de pruebas y validación.

Por el momento, solo admite GitHub y BitBucket como repositorios de código fuente. Tiene una versión limitada gratuita y una versión de pago, y facilita el flujo de trabajo con un *pipeline* definido.

Figura 7: Logo de CircleCI

2.5.3. CodeShip

CodeShip es un servicio de Entrega Continua alojado en la nube que se centra en la velocidad, fiabilidad y simplicidad. Con CodeShip se puede configurar un *pipeline* para construir una aplicación desde GitHub o BitBucket a la plataforma que se desee.

El servicio ofrece una gran variedad de potentes opciones de configuración y soporta múltiples lenguajes, así como plataformas de despliegue (incluida Google Cloud Platform).

Figura 8: Logo de CodeShip

Trabajar con un servicio como CodeShip permite minimizar las consecuencias de errores, acelerar su detección y hacer que el equipo de desarrollo se sienta más cómodo y competente a la hora de introducir actualizaciones incrementales regulares a su plataforma, con la esperanza de mejorar su capacidad de respuesta a los clientes y al mercado.

2.5.4. Codefresh

Codefresh es una plataforma de Entrega Continua creada en colaboración con Docker. El servicio ayuda a los equipos de desarrollo a mejorar la calidad y aumentar la velocidad de lanzamiento del producto, facilitando la retroalimentación rápida durante la fase de validación.

Figura 9: Logo de Codefresh

Codefresh permite a los desarrolladores compartir cualquier cambio en el código con el resto del equipo, incluyendo así al principio del ciclo de vida la retroalimentación sobre posibles fallos. Esta plataforma es la primera en adaptar Integración y Entrega Continua a los contenedores de forma nativa, permitiendo paralelizar las pruebas unitarias y las de integración.

2.5.5. Jenkins

Jenkins es un servicio de automatización, open-source, escrito en Java. Este proyecto está basado en otro anterior, Hudson, llegando a ser considerado como un simple cambio de nombre de éste. Jenkins se ha consolidado como una de las herramientas más usadas en la actualidad para realizar tareas de Integración Continua.

Figura 10: Logo de Jenkins

Jenkins, como servidor de Integración Continua desarrollado principalmente por Kohsuke Kawaguchi, permite planificar y realizar multitud de tareas, simplificando los procesos involucrados en el ciclo de vida de un proyecto. Admite diversas herramientas de control de versiones.

2.5.6. Spinnaker

Spinnaker es una plataforma para Entrega Continua de código abierto y multi-nube. Fue creada por Netflix, y ha sido probada por cientos de equipos a lo largo de millones de despliegues. Combina un potente y flexible sistema de gestión de *pipelines* con integraciones con los principales proveedores de cloud computing, incluido Google Cloud Platform.

Figura 11: Logo de Spinnaker

El punto más importante para entender la funcionalidad de Spinnaker es que necesita un servidor de Integración Continua para funcionar. Spinnaker solo gestiona la parte del despliegue a la nube durante el proceso de Entrega Continua, no realiza compilaciones ni ejecuta pruebas, porque no es un *task runner* genérico.

El uso de Spinnaker puede ser complementario a las herramientas mencionadas anteriormente, como a Jankins o Cloud Build, siendo especialmente interesante para proyectos multi-nube.

Nombre	Servicio	Preparado para contenedores	Integración con GCP	Precio	Compilación paralela
Google Cloud Build	CI/CD	Sí	Sí	120 mins gratis/día, 0.0034$/min	Hasta 10
CircleCI	CI	Sí	Sí	1000 mins gratis/mes, 50$ comp. paralelo	No
CodeShip	CD	Solo versión Pro	Sí	100 comp. gratis/mes, 49$ plan Basic	No (Basic)
Codefresh	CI/CD	Sí	Sí	220 comp. gratis/mes	2
Jenkins	CI/CD	Sí	Sí	Solo costes de la VM	Sí
Spinnaker	CD	Sí	Sí	121.40 $/mes	Sí

Tabla 1: Comparativa de servicios de *CI/CD* con integración en GCP

2.6. Docker y contenedores

Docker es un proyecto de código libre que proporciona muchas ventajas a los profesionales del desarrollo web y de aplicaciones, o a los administradores de sistemas, por la facilidad que supone el trabajar con el concepto de contenedores. La ventaja principal es que con Docker es posible encapsular todo el entorno de trabajo de forma que los desarrolladores puedan trabajar en su máquina (local) con la seguridad y garantía de que, al llegar el momento de llevar el código a producción, éste se ejecutará con la misma configuración sobre la que se han hecho todas las pruebas [8].

Estos contenedores de Docker podríamos definirlos como máquinas virtuales ligeras, menos exigentes con el hardware de los equipos donde se ejecutarán. Las características principales de estos contenedores son:

- **Portabilidad:** El contenedor Docker se podrá desplegar en cualquier otro sistema que soporte esta tecnología, evitando tener que instalar en este nuevo entorno todas aquellas aplicaciones que utilice el sistema.
- **Ligereza:** Al no virtualizar un sistema completo, sino solo lo necesario, el consumo de recursos se minimiza.
- **Autosuficiencia:** Docker se encarga de todo, por lo que los contenedores tan solo deben tener lo necesario para que la aplicación funcione, por ejemplo, aquellas librerías, archivos y configuraciones necesarias para poder realizar su función [9].

Docker también cuenta con una serie de repositorios, como *DockerHub*, similares a los de Linux, donde los usuarios publican sus propios contenedores de manera que los usuarios que los necesiten los puedan descargar rápidamente desde allí.

2.6.1. Docker Compose

Docker Compose es una herramienta que permite simplificar el uso de Docker, generando scripts que facilitan el diseño y la construcción de servicios.

Docker utiliza la tecnología de contenedores para desplegar imágenes. Estas imágenes suelen contener un servicio concreto, por ejemplo un sistema de bases de datos, un servidor web, un compilador específico etc. El gran problema que presenta este sistema es que, normalmente, vamos a necesitar múltiples de estos servicios para poder hacer funcionar nuestro software [12].

2.7. Gitflow

Gitflow es una extensión de trabajo con la que se consigue gestionar de manera eficiente las ramas de los repositorios de Git [13]. Con ella podemos clasificar las ramas en principales y de soporte. En cuanto a las ramas principales, encontramos:

- ***Master***: La rama *master* es la rama principal del repositorio, creándose por defecto cuando se inicia un repositorio de Git. En esta bifurcación solo estará la última versión del código, preparado para ser desplegado en producción.
- ***Develop***: La rama *develop* es la rama principal de desarrollo, en la cual se irán añadiendo cambios y nuevas funcionalidades, a la espera de ser añadidos en producción.

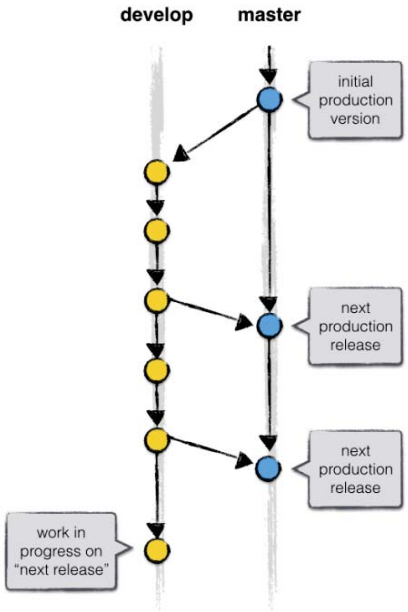

Figura 12: Diagrama de las ramas principales durante el desarrollo.

En las ramas principales no se debe modificar código en ningún caso. Para entender por qué no se debe, se puede utilizar el siguiente escenario: En un proyecto de software, hay un grupo de personas que trabajan en colaboración. El encargado del proyecto asigna a uno de ellos el desarrollo de una nueva característica y éste le dedica varias jornadas de trabajo. Antes de terminar los cambios, finalmente se decide que esa nueva característica no va a ser necesaria. Si se hubiese trabajado directamente en una rama principal, probablemente revertir esos cambios generaría muchos conflictos. En cambio, si se hubiese utilizado una rama aislada, bastaría simplemente con descartarla o bien mantenerla por si en un futuro próximo se decidiese retomar dicho desarrollo, sin afectar en ninguno de los casos a las ramas principales. Es por eso que existen las ramas de soporte, entre las cuales podemos encontrar:

- ***Feature***: Una *Feature* no es más que una rama de Git con una copia exacta del contenido de *develop*. Su propósito es desarrollar nuevas funcionalidades para el proyecto, en un ambiente controlado, de tal forma que, una vez se hayan terminado los cambios, se añadan a la rama de *develop*. Así se puede evitar que se generen conflictos

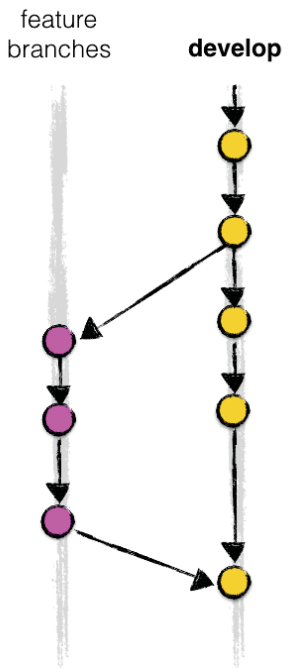

Figura 13: Diagrama de flujo de trabajo de nuevas características.

- ***Release***: Una *release* es una copia exacta del contenido de *develop*, que está lista (a falta de las últimas pruebas manuales), para ser subida a producción. Es importante destacar que en esta tipo de ramas no se deben hacer grandes cambios, solo gestionar posibles cambios como fallos menores o actualizaciones del número de versión. Al cerrar esta rama, se integra tanto con la rama de *master* como con *develop*, para que todas las ramas principales queden sincronizadas.

- ***Hotfix***: Un hotfix es una rama con una copia exacta del contenido de *master*. Este tipo de bifurcaciones se utilizan para corregir errores inesperados en producción, cuando se necesite solucionarlos rápidamente y desplegar una nueva versión.

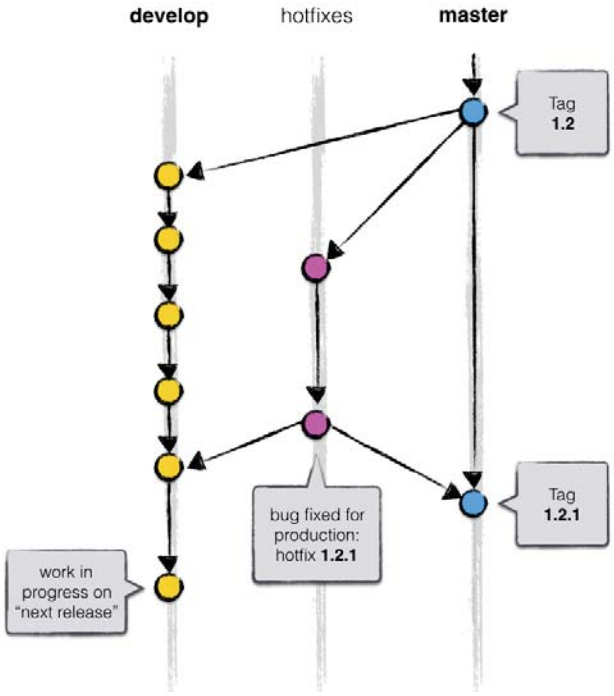

Figura 14: Diagrama de flujo de trabajo de Hotfixes.

Un ejemplo de flujo de trabajo completo es el siguiente:

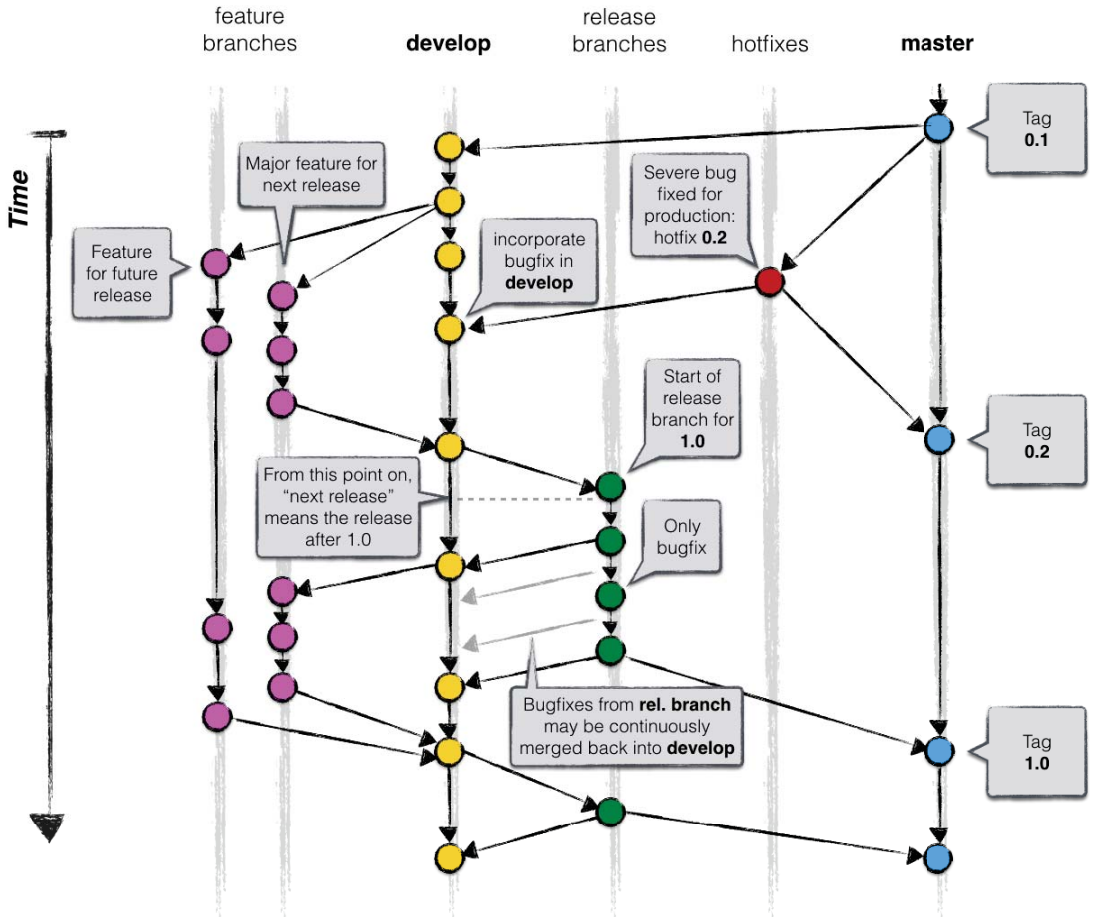

Figura 15: Diagrama de flujo de trabajo completo con Gitflow.

3. PLANTEAMIENTO DEL PROBLEMA

La implementación de Entrega Continua no es algo que se pueda lograr de la noche a la mañana. Antes de entrar en el desarrollo de la solución, hay que estudiar y plantear una aproximación de los requisitos, para poder llegar así a la solución óptima.

Uno de los mayores errores que cometen las empresas que quieren adoptar Entrega Continua es la prisa por llegar al resultado final. Las presiones del mercado pueden ser intensas, pero se recomienda encarecidamente trabajar y validar cada parte individualmente antes de continuar con la siguiente.

Los pasos que se van a plantear en este proyecto son los siguientes:

- Adoptar un control de versiones
- Definición de release
- Análisis del código
- Automatización de la compilación
- Automatización de las pruebas
- Automatización del despliegue

3.1. Adoptar un control de versiones

El sistema de control de versiones es la herramienta fundamental que permite que el equipo trabaje de forma sincronizada, organizada y sencilla. Permite a cada programador trabajar de forma individual y poder actualizar sus cambios sin necesidad de depender del resto de desarrolladores.

Un *VCS* es una herramienta capaz de registrar los cambios que se realizan en los repositorios de código, guardando el histórico de versiones que se han ido realizando y pudiendo regresar a cualquiera de ellas en cualquier momento.

Los *VCS* se pueden clasificar según la arquitectura utilizada para almacenar el código fuente:

- ***VCS* distribuidos:** Cada usuario tiene su propio repositorio de forma local. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Frecuentemente, se usa un repositorio como base, sirviendo éste como base y como punto de sincronización de los distintos repositorios locales, al estar disponible continuamente.

  Los *VCS* distribuidos no necesitan estar siempre conectados para realizar operaciones, lo que produce una mayor autonomía y rapidez. Además, al hacer uso de múltiples copias del repositorio remoto de forma local, el código del repositorio está muy replicada y es sencillo recuperar información en caso de pérdida de datos.

  Los ejemplos más utilizados de *VCS* distribuidos son Git, Mercurial y Google Cloud Source Repositories.

- ***VCS* centralizados:** Existe un único repositorio de todo el código fuente, del cual es responsable un usuario o un conjunto de ellos. Esto facilita las tareas administrativas a cambio de reducir la flexibilidad, pues todas las decisiones importantes, como la creación de una nueva rama, requieren la aprobación del encargado.

  La ventaja de los *VCS* centralizados respecto a los distribuidos es que en los sistemas distribuidos hay menos control a la hora de trabajar en equipo, pues no existe una versión centralizada de los cambios que se están haciendo y puede haber problemas a la hora de combinar el trabajo. Algunos ejemplos son CVS, SubVersion o Team Foundation Server.

3.2. Definición de release

Un principio básico de Entrega Continua es tener el código en un estado en el que en cualquier momento se pueda hacer una *release*. Antes de asumir las tareas de automatización de tareas, lo primero que se debe determinar es qué constituye una nueva *release*. Por ejemplo, se puede decidir que cada cambio en el código resulte en una *release* potencial, o crear una rama específica para cada versión.

Para crear una nueva *release*, el primer paso es conocer las dependencias con los componentes existentes y los requisitos que se habían planteado, para poder plantear así las pruebas que el código debe superar para garantizar su funcionalidad. Una vez conocidos los requisitos y las dependencias, se puede comenzar a planificar el proceso de automatización de la Entrega.

3.3. Análisis del código

OrbitalAds cuenta con varios tipos de servicios, que se validan y despliegan de diferentes formas. Entre ellos, podemos encontrar:

- Google Cloud Functions
- Servicios de Google Compute Engine
- Servicios de Google Firebase

Se deberá analizar cada tipo de servicio individualmente para poder determinar los pasos a seguir durante la compilación y despliegue de los mismos. Además, habrá que tener en cuenta que las Cloud Functions de *OrbitalAds* utilizan el entorno de Python 3.7, que se encuentra en estado beta, y los servicios de Firebase se están desarrollando desde cero paralelamente a este proyecto, por lo que podrá haber funcionalidades que no estén disponibles o aún no se hayan implementado.

3.4. Automatización de la compilación

Para automatizar la compilación, se debe conectar la herramienta de Integración Continua que se haya elegido con el repositorio de código fuente.

En el caso de Google CloudBuild, las compilaciones automáticas se hacen a través de *build triggers*, es decir, hay que definir qué eventos van a disparar el proceso de compi-

lación automática. Con la función Activadores de compilación se pueden compilar contenedores de forma automática en función del código fuente o los cambios en las etiquetas de un repositorio y se puede, además, definir diferentes pasos previos a la compilación, que serán muy útiles para los siguientes pasos de automatización.

La creación de activadores no se puede automatizar, por lo que será un proceso manual para cada servicio.

Para compilar el código fuente de un repositorio, CloudBuild clona la rama especificada en el Activador del repositorio, por lo que no comprueba ninguna otra bifurcación o historial. Existen distintos tipos de activadores:

- **Activadores de rama:** Se lanzarán cuando exista algún cambio en la rama especificada.
- **Activadores de etiqueta:** Se lanzarán cuando exista algún cambio con una determinada etiqueta.

Tanto los activadores de rama como los de etiqueta admiten también expresiones regulares, pudiendo flexibilizar así el nombrado de ramas y etiquetas. También se pueden especificar en qué archivos buscar cambios, y en cuáles los cambios no desencadenarán la compilación.

Para ejecutar compilaciones de Docker con activadores de CloudBuild, se puede especificar tanto un Dockerfile, es decir, un documento que contiene todas las instrucciones y comandos necesarios para que Docker genere la imagen del contenedor, o un fichero de configuración de la compilación, que incluye los pasos necesarios para realizar la compilación, extendiendo a los especificados en el Dockerfile.

Este archivo se modela utilizando la sintaxis de YAML, y se leerá cada vez que se lance una compilación. Los pasos de la compilación especifican una acción que CloudBuild deberá ejecutar. Para cada paso, CloudBuild ejecuta un contenedor de Docker como instancia del comando `docker run`. Estos pasos se pueden considerar análogos a los comandos que se pueden encontrar en cualquier script, proporcionando la flexibilidad de

ejecutar instrucciones arbitrarias en cada compilación. Los pasos especificados tienen la siguiente forma:

```
steps:
- name: 'gcr.io/cloud-builders/git'
  args: ['clone',
  'https://github.com/GoogleCloudPlatform/cloud-builders']
  env: ['PROJECT_ROOT=hello']
- name: 'gcr.io/cloud-builders/docker'
  args: ['build', '-t', 'gcr.io/my-project-id/myimage',
  '.']
```

El campo `name` de un paso de compilación especifica un *Cloud Builder*, que son los contenedores que utiliza CloudBuild para ejecutar los distintos comandos. En este ejemplo se utiliza el *Cloud Builder* de Git y Docker.

El campo `args` es una lista de los argumentos que se pasarán al comando ejecutado por el *Cloud Builder*, permitiendo invocar cualquier comando soportado por la herramienta. Si el *Cloud Builder* tiene un punto de entrada o *entrypoint*, la lista se utilizará como argumentos de ese entrypoint. Si no está definido un entrypoint específico, el primer elemento de la lista de argumentos será utilizado como entrypoint y el resto como argumentos.

Con el campo `env` se puede especificar una lista de variables de entorno que se usarán durante la ejecución de ese paso, con la forma Clave=Valor.

3.5. Automatización de las pruebas

Para poder garantizar la calidad de nuestro software debemos asegurar que podemos obtener información rápida sobre el impacto de los cambios en el código. Tradicionalmente, la validación se hacía a través de inspecciones manuales de código, es decir, las pruebas se hacían siguiendo la documentación que describía los pasos necesarios para

probar las diversas funciones del sistema, con el fin de demostrar la correcta funcionalidad del sistema. Sin embargo, esta prueba tiene diversos inconvenientes:

- Las pruebas de regresión manuales consumían mucho tiempo, creando un cuello de botella a la hora de desplegar los cambios con más frecuencia, y la retroalimentación para los desarrolladores podía tardar semanas.
- Las pruebas e inspecciones manuales no son confiables, ya que los procesos manuales son muy susceptibles a errores humanos al ser tareas repetitivas.
- Además, es muy difícil predecir el impacto de los cambios en el software tan solo con inspecciones manuales, y conlleva invertir un esfuerzo considerable en actualizar la documentación de las pruebas para poder mantenerla al día.

En *OrbitalAds*, la validación de los cambios en el código tiene dos fases; la revisión por parejas y la ejecución manual de pruebas del código. La primera consiste en una revisión del código por otro miembro del equipo distinto al autor original de los cambios, cuyo objetivo principal es la búsqueda de defectos y la propuesta y evaluación de soluciones alternativas o de mejoras en el diseño y los algoritmos utilizados. A pesar de no ser un método confiable, facilita la difusión de conocimiento a través del equipo. Esta fase no se puede automatizar.

La segunda fase consiste en una serie de pruebas unitarias y de integración desarrolladas con pyTest y unittest, marcos de trabajo que facilitan la implementación de pruebas pequeñas, pero escalables a la hora de soportar funcionalidades complejas. Esta fase también es manual, y normalmente solo se ejecutan cuando el encargado de realizar cambios en el código termina el trabajo, pero no cada vez que se realiza una compilación. Esta tarea se puede automatizar con CloudBuild, añadiendo el paso correspondiente, pero habrá que estudiar cómo gestionar las dependencias de cada servicio. Al implementar la automatización de pruebas, se debe emitir un informe de las pruebas realizadas y los resultados, para que puedan ser revisados automáticamente por CloudBuild y poder así registrar dónde ha ocurrido el fallo, en caso de haberlo.

3.6. Automatización del despliegue

Una vez validados y compilados, los servicios tienen que ser desplegados. El primer paso será desplegarlos en un entorno de pruebas que debe emular al entorno de producción, con el objetivo de evitar errores cuando ya se haya desplegado. Lo más complicado de emular los servicios en un entorno de pre-producción es emular las dependencias de bases de datos y de terceros, problema que habrá que abordar para poder completar la fase de validación.

Asimismo, cada tipo de servicio se despliega de forma distinta:

- Las Cloud Functions se desplegarán en el entorno de ejecución de Python3.7, basado en Ubuntu 18.04, que se encuentra en fase beta.
- Los servicios de Compute Engine se desplegarán como Plantillas de instancia y Grupos de instancia, una arquitectura novedosa recién implementada en *OrbitalAds*, de la cual, de momento, se desconocen sus peculiaridades de cara a su mantenimiento, teniendo que definir una política de versionado para los nuevos despliegues.
- Los servicios de Google Firebase también se incluyen en el grupo de servicios implementados con la nueva arquitectura. Este proyecto no incluirá pruebas que validen su funcionamiento, debido a la proximidad de la fecha de entrega. Por lo tanto, no se incluirán en este proyecto.

  Para su despliegue, los servicios de Firebase necesitan un token de autenticación, el cual se evitará su distribución en claro cifrándolo con Google Key Management Service.

4. RESOLUCIÓN

Uno de los prerrequisitos de este proyecto era realizarlo en Google Cloud Platform (ver 1.4.2). En la sección 2.5 se estudiaron las herramientas con integración en esta plataforma, seleccionando para el desarrollo Google CloudBuild.

Entre las ventajas que reporta el uso de Google CloudBuild, podemos encontrar:

- **Compatibilidad nativa con contenedores de Docker**: Google CloudBuild permite definir los pasos del *pipeline* de integración utilizando imágenes de Docker, ya sean imágenes públicas o privadas, pudiendo aprovechar pequeñas imágenes construidas específicamente para cada paso.
- **Despliegues más rápidos**: Google CloudBuild, al utilizar la red propia de Google, puede descargar y subir imágenes de Docker más rápidamente.
- **Pruebas locales**: Google CloudBuild proporciona herramientas para realizar pruebas locales, distribuidas en el SDK de Google Cloud, a través del comando *cloud-build-local*. Por tanto, permite crear y depurar en la propia máquina local de forma gratuita.
- **Concurrencia**: Dentro de los gestores de Integración y Entrega continuas estudiados, muchos no soportaban realizar compilaciones concurrentes, o este servicio era muy limitado y de pago. Google CloudBuild soporta hasta 10 compilaciones concurrentes, de forma que el precio mantiene el mismo enfoque que si fueran 10 compilaciones secuenciales. Esto permite aprovechar al máximo el plan gratuito del servicio.
- **Precio**: Google CloudBuild ofrece 120 minutos de compilación gratuitos al día, indefinidamente. En el caso de sobrepasar ese límite, el precio es de 0,0034 dólares por minuto. Esto se traduce en 3600 minutos de compilación gratuitos al mes y un precio muy reducido. Suponiendo que la compilación de los servicios de *OrbitalAds* tardaran, de media, 10 minutos, se podrían hacer 1667 compilaciones al mes o 50 compilaciones al día para llegar al precio base que costaría, por ejemplo, CircleCI.

Google CloudBuild también cuenta con algunas desventajas que habrá que tener en cuenta durante el desarrollo del proyecto. Entre ellas:

- **Configuración de máquina limitada**: La máquina base donde se ejecutarán las compilaciones es del tipo *n1-standard-1*, una máquina con 1 núcleo de CPU y 3,75GB de RAM. Esta máquina se puede ampliar a 8 o 32 núcleos de CPU, pero sería interesante tener opciones intermedias que se ajusten más a las necesidades de *OrbitalAds*.
- **Falta de integraciones**: Añadir notificaciones, por ejemplo, a Slack, se basa en el desarrollo de una Cloud Function privada, en lugar de la integración *en un click* que ofrecen otros servicios.
- **UI muy pobre**: Las compilaciones se pueden etiquetar para poder posteriormente filtrar según el tipo que se desee, pero utiliza un lenguaje para las consultas propio que no está documentado. Además, no se puede ver el progreso individual de los pasos de un *pipeline* hasta que no haya finalizado la ejecución, ya sea con éxito o con errores.
- **Pasos nativos muy limitados**: Google CloudBuild proporciona imágenes para crear pasos con las herramientas más comunes. La lista completa es esta:


- `bazel` : runs the bazel tool
- `curl` : runs the curl tool
- `docker` : runs the docker tool
- `dotnet` : run the dotnet tool
- `gcloud` : runs the gcloud tool
- `git` : runs the git tool
- `go` : runs the go tool
- `gradle` : runs the gradle tool
- `gsutil` : runs the gsutil tool
- `kubectl` : runs the kubectl tool
- `mvn` : runs the maven tool
- `npm` : runs the npm tool
- `wget` : runs the wget tool
- `yarn` : runs the yarn tool

Figura 16: Herramientas nativas de Google CloudBuild.

Sin embargo, no incluye herramientas para servicios de su propia plataforma, como puede ser Google Firebase, o compatibilidad nativa con Docker compose.

4.1. Análisis del pipeline de integración actual

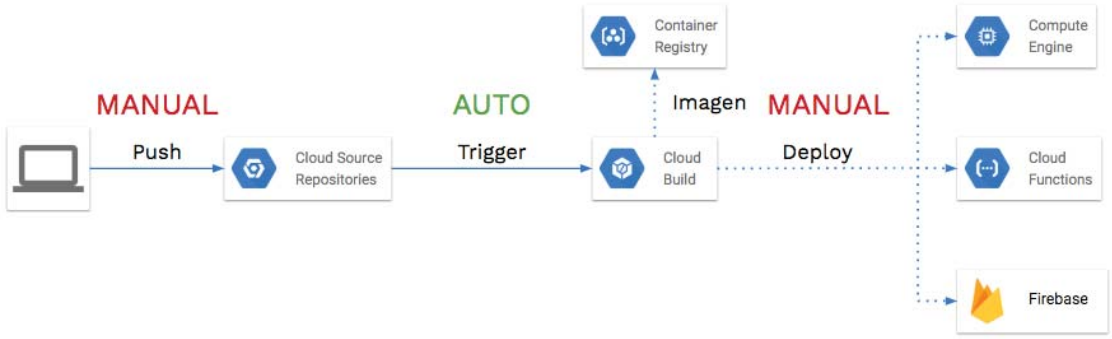

Figura 17: Diagrama del *pipeline* actual de *OrbitalAds*

El *pipeline* actual de *OrbitalAds* consiste en una serie de pasos manuales en los que únicamente está automatizado la compilación de la imagen de Docker. Cuando se termina una característica nueva o se hace algún cambio en el código fuente, el programador integra sus cambios en la rama *develop* del repositorio, lo que desencadena la compilación automática del proyecto y la subida de la imagen del contenedor a Google Container Registry. En este paso no se ejecuta ningún tipo de prueba, por lo que no existe ninguna fase de validación automática de los cambios que se han introducido. Al finalizar la compilación, el desarrollador puede, manualmente, desplegar el servicio en Google Cloud Platform, en el entorno que corresponda.

Comprobando los pasos que se plantearon durante la definición del problema, expuestos en la sección 3, podemos analizar el estado del pipeline actual:

Paso	Completado	Justificación
Adoptar un control de versiones	✓	Git + Gitflow en Google Cloud Source Repositories
Definición de release	✗	No existe definición clara de release.
Análisis de código	✗	Todos los servicios pasan el mismo proceso manual.
Automatización de la compilación	✓	El contenedor se compila automáticamente, pero no hace uso de caché.
Automatización de las pruebas	✗	Pruebas manuales.
Automatización del despliegue	✗	Despliegues manuales.

Tabla 2: Análisis del estado del Pipeline actual

4.2. Definición de release

Tras el análisis del estado actual del pipeline, el primer paso a realizar es formalizar la definición de *release*. Normalmente, dentro del contexto de la arquitectura de software hay varios atributos que se deben tener en cuenta, como la disponibilidad, la seguridad, el rendimiento, o la facilidad de uso, entre otros. La Entrega Continua introduce dos nuevas propiedades: Capacidad de prueba y de despliegue.

En sistemas con buena capacidad de prueba, el software se diseña de forma que la mayoría de defectos puedan ser detectados, en principio, en las primeras fases del ciclo de vida mediante la ejecución de pruebas automatizadas, sin depender de entornos complejos para llevar a cabo la validación del producto.

En sistemas con buena capacidad de despliegue, los cambios en un servicio en particular se pueden realizar de manera independiente y de forma totalmente automatizada, sin la necesidad de niveles significativos de orquestación. Este tipo de sistemas se pueden actualizar de forma que su tiempo de inactividad sea prácticamente nulo.

Desde la premisa de diseño de arquitectura centrada en la capacidad de prueba y despliegue, podemos definir una *release* a partir de cualquier cambio que se realice en el código fuente, asegurando que los servicios son componentes bien encapsulados. En cada *release* será posible probar y desplegar un servicio por sí mismo, reemplazando cualquier dependencia con un *clon* para pruebas adecuado, para así obtener un alto nivel de confianza de que el componente funciona correctamente.

4.3. Google Cloud Functions

Google Cloud Functions es una plataforma de ejecución sin servidor controlada por eventos. Se utiliza como solución de procesamiento ligero para que los desarrolladores puedan crear funciones individuales y de un sólo propósito que respondan a eventos de Cloud, sin la necesidad de administrar un entorno de servidor o de ejecución.

Existen distintos tipos de Cloud Functions, dependiendo del tipo de evento que desencadene su ejecución. Por ejemplo:

- Por peticiones HTTP.
- Por eventos generados en algún cambio en Google Cloud Storage.
- Por mensaje de Google Pub/Sub.

En *OrbitalAds* las Cloud Functions se desencadenan, en su gran mayoría, por peticiones HTTP, y algunas por mensajes de Pub/Sub. Por tanto, en este proyecto nos centraremos en la Entrega Continua de este tipo de funciones.

4.3.1. Permisos y IAM

El primer paso para poder automatizar las pruebas y el despliegue de Cloud Functions es garantizar los permisos necesarios para ello a una cuenta de servicio. Una cuenta de servicio o *service account* es una identidad que las aplicaciones pueden utilizar para ejecutar solicitudes a cualquier API de Google.

Si una cuenta de servicio tiene los permisos IAM (Identity and Access Management) necesarios, puede crear y administrar instancias de servicios y otros recursos. Por ejemplo, en un servicio que lee y escribe archivos de Google Cloud Storage, la cuenta de servicio asociada primero debe autenticarse en la API de Google Cloud Storage para poder llevar a cabo su función.

Para poder desplegar Google Cloud Functions, habrá que otorgar a la cuenta de servicio de CloudBuild los permisos de Desarrollador de Cloud Functions. En la consola de Google Cloud Platform podemos encontrar el menú de IAM y administración:

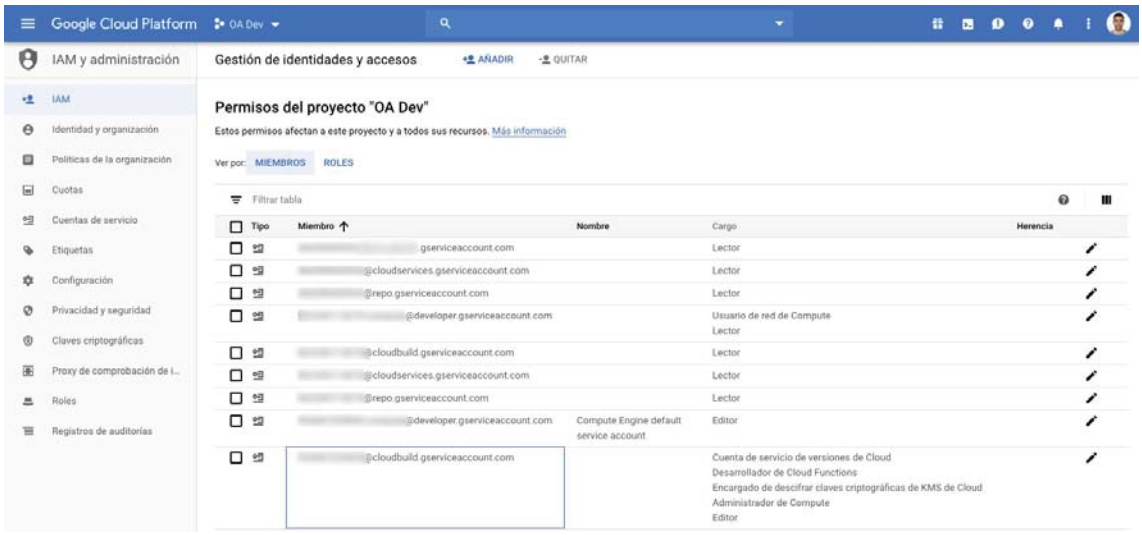

Figura 18: Panel de Permisos de Google Cloud.

4.3.2. Gestión de dependencias

La gestión de dependencias de las Cloud Functions se hace a través de Pipenv, una herramienta oficial recomendada por PyPA (Python Packaging Authority) muy útil para administrar dependencias en entornos virtuales.

Pipenv proporciona un método sencillo de configurar un entorno de trabajo, unificando las herramientas pip y virtualenv. Normalmente, para gestionar las dependencias de un proyecto de Python se utiliza pip, creando un archivo `requirements.txt` en el cual se definen las distintas dependencias del proyecto, especificando la versión necesaria. En el caso de Pipenv, se utilizan dos archivos, `Pipfile` y `Pipfile.lock`, para poder separar las declaraciones abstractas de dependencias de la última combinación de versiones ya validada. Un ejempo de `Pipfile` se puede encontrar en el Anexo A.

4.3.3. Activadores

Previamente a definir los pasos que va a realizar el *pipeline* de despliegue, hay que crear el activador que desencadenará la ejecución de estos pasos. En la sección de Herramientas de Google Cloud Platform encontramos el menú de CloudBuild, y ahí podremos definir los activadores:

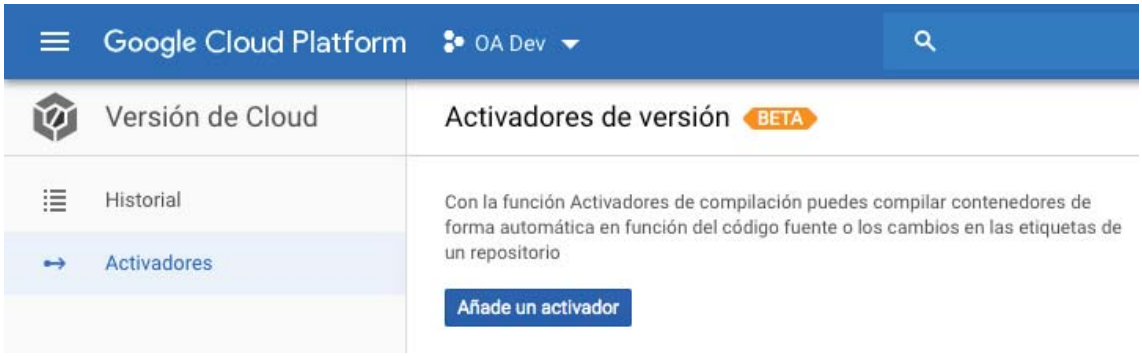

Figura 19: Menú de creación de activadores de CloudBuild.

El siguiente paso es seleccionar el origen del código fuente, es decir, el repositorio donde estará el código del proyecto. En el caso de *OrbitalAds*, el origen es Google Cloud source Repositories.

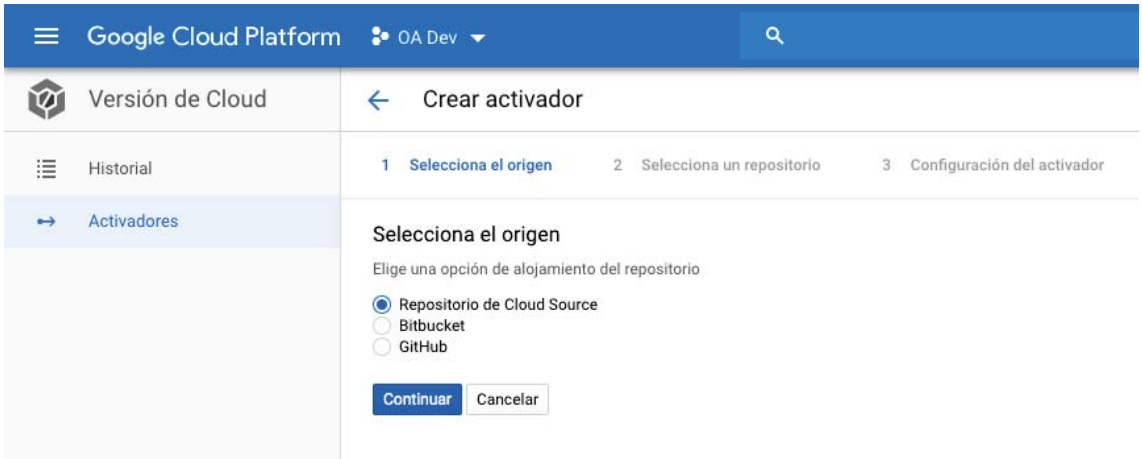

Figura 20: Panel de selección de Origen de CloudBuild.

Por último, hay que especificar el tipo de activador que va a ser (ver 3.4), la expresión regular que determinará como filtrar los eventos que ocurran en el repositorio según el nombre de la rama o etiqueta y la ruta al script de configuración de `cloudbuild.yaml`.

Para que la adopción de Entrega Continua fuese más sencilla para el equipo de desarrollo, se definió un único *pipeline* que fuese válido para los entornos tanto de desarrollo como de producción. Esto es posible gracias a las variables de sustitución, que se declaran en `cloudbuild.yaml` y se definen en el momento de crear el activador.

Las variables de sustitución deben comenzar por un guión bajo (_), y solo pueden utilizarse letras mayúsculas y números, para evitar conflictos con las sustituciones nativas. En el caso de las cloud functions, las variables que se han definido son las siguientes:

- **_CF_NAME**: Nombre de la Cloud Function.
- **_ENTRY_POINT**: Entry point de la Cloud Function.
- **_ENV_FILE**: Nombre del archivo donde se definen las variables de entorno necesarias durante la ejecución.
- **_GCP_PROJECT_ID**: nombre del proyecto donde se desplegará la Cloud function.

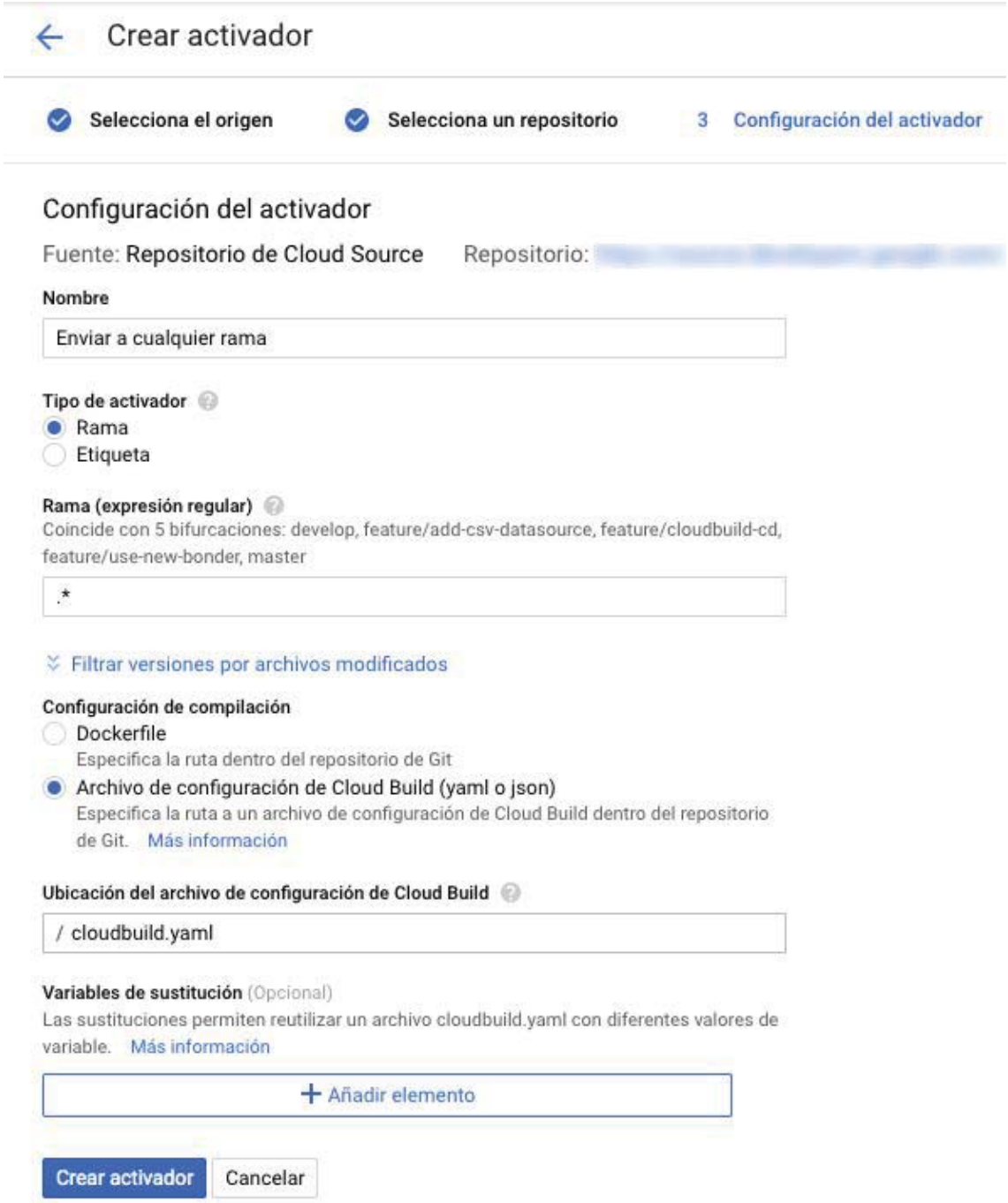

Figura 21: Panel de creación de activadores de CloudBuild.

4.3.4. Automatización con Google CloudBuild

Para automatizar tanto la fase de pruebas y validación como la de despliegue (las Cloud Functions no necesitan compilarse) se desarrolló el archivo `cloudbuild.yaml`, donde se especifican los pasos a seguir del *pipeline* de despliegue. El contenido del archivo `cloudbuild.yaml` es el siguiente:

```
steps:

- name: 'python:3.7'
  args: ['/bin/bash', './cloudbuild.sh']
  id:   'unittest'
  env:  []

- name: 'gcr.io/cloud-builders/gcloud'
  args: ['functions', 'deploy', '${_CF_NAME}',
         '--entry-point', '${_ENTRY_POINT}',
         '--runtime', 'python37',
         '--env-vars-file', '${_ENV_FILE}',
         '--trigger-http',
         '--project', '${_GCP_PROJECT_ID}',
         '--region', 'europe-west1']
  id: 'deploy'
```

El primer paso del *pipeline* ejecuta las pruebas unitarias de la Cloud Function, lanzando un script llamado `cloudbuild.sh` en el entorno de ejecución de Python3.7. Se eligió este entorno de ejecución para poder evitar la instalación de Python, escogiendo un entorno en el que viniera preinstalado. El script realiza los siguientes comandos:

```
#!/bin/bash

pip install --upgrade pip
```

```
4 pip install pipenv
5 pipenv install
6 pipenv run python -m unittest
```

Para poder gestionar las dependencias, instala el paquete Pipenv a través de Pip, para después instalar las dependencias especificadas en los archivos `Pipfile` y `Pipfile.lock`. Finalmente, ejecuta las pruebas unitarias en el entorno virtual recién creado, a través del paquete de Python llamado `unittest`.

4.3.5. Despliegue en entornos de desarrollo y producción

El último paso del *pipeline* es el despliegue de la Cloud Function al entorno que corresponda, según las variables de sustitución que se hayan definido en el activador. Para este paso se utiliza el entorno de ejecución de `gcloud`, para poder utilizar el comando de despliegue del SDK de Google Cloud Platform.

A modo de resumen, la tabla 3 muestra la configuración que habrá que definir en los distintos activadores según el entorno, y la tabla 4 los distintos valores de las variables de sustitución.

	Desarrollo	**Producción**
Nombre	CD develop	CD prod
Tipo	Rama	Etiqueta
Rama/Etiqueta	develop	v[0-9]{4}-[0-9]{2}-[a-z]
Configuración	Cloudbuild.yaml	Cloudbuild.yaml

Tabla 3: Resumen de configuración de Activadores según el entorno.

Variable	**Desarrollo**	**Producción**
_CF_NAME	nombre	nombre
_ENTRY_POINT	entrypoint	entrypoint
_ENV_FILE	env_devel.yaml	env.yaml
_GCP_PROJECT_ID	oa-devel	oa-prod

Tabla 4: Resumen de variables de sustitución según el entorno.

4.4. Google Compute Engine

Google Compute Engine ofrece máquinas virtuales que se ejecutan en los centros de datos de Google, conectadas a nivel mundial a través de una red de fibra.

Las herramientas disponibles en Google Compute Engine permite escalar desde instancias individuales hasta un entorno de *Cloud Computing* global con balanceo automático de carga, iniciando máquinas virtuales rápidamente con un rendimiento uniforme.

La arquitectura de *OrbitalAds* de Compute Engine es mucho más compleja que con las Google Cloud Functions. Para crear las máquinas virtuales se utilizan plantillas de instancia, un recurso de Google Cloud Platform que se utiliza para definir el tipo de máquina, el contenedor de Docker, etc., que utilizará la máquina virtual. Las plantillas de instancia están diseñadas para crear máquinas virtuales con idéntica configuración. Hay que tener en cuenta que, una vez creada una plantilla de instancia, no se podrá modificar, por lo que habrá que definir una política de versionado para administrar las versiones que vayan surgiendo a lo largo del desarrollo. Para crear colecciones de máquinas virtuales con la misma configuración, se utilizan los Grupos de instancia, permitiendo así gestionar el escalado automático del número de instancias.

La arquitectura de los servicios de *OrbitalAds* implementados en Google Compute Engine funciona de la siguiente manera: Cuando se deba levantar una instancia, se recibirá un mensaje en una cola de Google Pub/Sub determinada. Al detectarse ese mensaje por la plataforma de monitorización de Stackdriver, se levantará la máquina virtual, recibirá y leerá el mensaje, lo procesará y al acabar la ejecución se apagará la instancia. Se levantarán en un Grupo de Instancias tantas instancias como mensajes lleguen a la cola, sin sobrepasar el límite establecido en la política de autoescalado.

4.4.1. Activadores

La creación de los activadores se realizará de la misma forma que en la sección 4.3.3, salvo que no serán necesarias las variables de sustitución, porque en este caso no se podrá

estandarizar el proceso para todos los servicios de Google Compute Engine, debido a las peculiaridades tanto de pruebas como de despliegue de cada uno de ellos.

El *pipeline* de despliegue completo se puede encontrar en el anexo B.

4.4.2. Política de versionado

Una vez creada, una Plantilla de instancia no se puede modificar. Esto implica que no se puede cambiar la imagen del contenedor de Docker que usa la plantilla para que utilice la última versión disponible, teniendo que crear una plantilla de instancia nueva (con distinto nombre), o bien borrar la plantilla antigua y crearla de nuevo. La solución en *OrbitalAds* fue utilizar estos dos escenarios.

En el caso del entorno de desarrollo, la última imagen disponible siempre va a tener la etiqueta `develop`. Por tanto, no se podrá diferenciar una versión de otra y cada vez que se haga un nuevo despliegue se borrará la plantilla de instancia y se creará otra vez, sin necesidad de actualizar el Grupo de instancias porque la plantilla es, teóricamente, la misma.

Sin embargo, para el entorno de producción sí es interesante poder mantener distintas versiones, no solo para tener un histórico de versiones, sino para poder también volver rápidamente a una versión anterior en caso de necesidad, o por si hubiera incompatibilidad entre versiones. Debido a esto, la política que se va a seguir para los despliegues a producción (*releases*) es mantener una Plantilla de instancia, nombrada con el formato `<nombre de servicio>-<versión>` y actualizar cada vez el Grupo de instancias para que utilice la última versíon. En el caso de tener que revertir a una versión anterior, bastaría con volver actualizar el Grupo de instancias, operación muy sencilla si fuera necesaria (1-click rollback).

4.4.3. Automatización de la compilación

Como se pudo comprobar durante el análisis del *pipeline* actual (ver sección 4.1), la compilación de los proyectos ya era un proceso automatizado, pero se podía mejorar.

En *OrbitalAds* se utilizaban activadores de rama de CloudBuild para lanzar un proceso automático básico de compilación de imágenes de Docker. Este proceso es largo, teniendo que gestionar la compilación y las descarga de dependencias cada vez que se realizaba algún cambio en el código fuente.

Con el objetivo de acelerar las compilaciones, se analizaron las formas de aumentar la velocidad de compilación de una imagen de Docker. El primer paso fue especificar la imagen antigua del servicio que se estaba compilando para utilizarla como caché en compilaciones posteriores. Sin embargo, este cambio no funcionaba tan bien como se esperaba, teniendo todavía que esperar a la descarga de dependencias cada vez que había modificaciones en el proyecto.

Para entender por qué ocurría esto hubo que analizar cómo funcionan las imágenes de Docker: Cada imagen de Docker está compuesta de capas apiladas, siendo cada capa cada una de las operaciones que se realizan para montar la imagen. Con el uso de una imagen anterior como caché, se vuelven a compilar todas las capas desde la primera capa modificada hasta el final de la compilación. Un ejemplo de `Dockerfile` de un servicio de *OrbitalAds* es el siguiente:

```
1 FROM gcr.io/oa-devel/baseproject:master
2 LABEL maintainer='OrbitalAds "dev@orbitalads.io"'
3
4 EXPOSE 8080
5
6 RUN mkdir -p /usr/src/project
7 ENV PYTHONPATH /usr/src/project
8
9 WORKDIR /usr/src/project/
```

```
10 COPY . /usr/src/project/
11
12 RUN apt-get update && apt-get -y upgrade
13 RUN pipenv install --system --deploy
14
15 ENTRYPOINT python3 main.py
```

En la línea 10 se copiaba todo el proyecto al contenedor de Docker, para posteriormente en la línea 12 y 13 instalar las dependencias del proyecto. Esto implicaba que si se realizaba algún cambio, aunque no tuviera nada que ver con las dependencias, hiciera que se descartasen las capas posteriores a la copia del proyecto al contenedor. Para optimizar el uso de la imagen antigua como caché, se reestructuró el `Dockerfile` de forma que, en primer lugar, solo se copiaran los archivos donde se definían las dependencias, para después proceder a instalarlas. Una vez gestionadas las dependencias, se copia el resto del proyecto al contenedor. El `Dockerfile` queda de la siguiente manera:

```
1  FROM gcr.io/oa-devel/baseproject:master
2  LABEL maintainer='OrbitalAds "dev@orbitalads.io"'
3
4  EXPOSE 8080
5
6  RUN mkdir -p /usr/src/project
7  ENV PYTHONPATH /usr/src/project
8
9  WORKDIR /usr/src/project/
10 COPY  /usr/src/project/
11
12 RUN apt-get update && apt-get -y upgrade
13 RUN pipenv install --system --deploy
14
15 COPY . /usr/src/project/
16
17 ENTRYPOINT python3 main.py
```

Con el `Dockerfile` reestructurado, en la línea 10 solo se copian los archivos de dependencias `Pipfile` y `Pipfile.lock`, que varían menos frecuentemente, copiándose el resto del proyecto en la línea 15, aprovechando así todas las ventajas del uso de caché en la compilación de imágenes de Docker y acelerando drásticamente los tiempos de compilación.

Para automatizar este proceso en CloudBuild, se definieron los siguientes pasos, utilizando el compilador nativo de Docker de CloudBuild para ejecutarlos:

```
1  # Paso 0: Descargar la imagen anterior para poder
   utilizarla como caché de la nueva.
2  - name: 'gcr.io/cloud-builders/docker'
3    entrypoint: 'bash'
4    args:
5    - '-c'
6    - |
7      docker pull gcr.io/oa-devel/project:develop || exit 0
8    id: 'pull-cache-image'
9
10 # Paso 1: Build de la imagen de docker
11 - name: 'gcr.io/cloud-builders/docker'
12   args: ['build', '-t',
     'gcr.io/oa-devel/project:cloudbuild',
13           '--cache-from', 'gcr.io/oa-devel/project:develop',
             '.']
14   id: 'build'
```

4.4.4. Automatización de las pruebas

Para realizar las pruebas tanto unitarias como de integración en los servicios de Google Compute Engine, en *OrbitalAds* se utiliza Docker-compose para levantar emuladores de los servicios de los que depende el servicio que se quiere validar. Por ejemplo, se uti-

lizan emuladores para Google Pub/Sub, Google Datastore, bases de datos PostgreSQL, entre otros.

Este proceso es manual, de forma que en el momento de realizar las pruebas cada desarrollador levanta los servicios en su máquina local y ejecuta después las pruebas.

Para automatizar este proceso en CloudBuild se utilizó la imágen pública de Docker-compose, estudiando la forma de que los logs que recibiera el programador como retroalimentación de la fase de validación fuesen únicamente relevantes al servicio que se estaba probando y no de los emuladores de los servicios dependientes.

Con este objetivo, en vez de ejecutar todo el proceso con `docker-compose up`, se especificaba en el comando el servicio concreto que se quería analizar con `docker-compose up project`. El problema que surgía es que, en el caso de fallar las pruebas, el *pipeline* de despliegue no detectaba el error y continuaba con el despliegue. Este error se pudo solventar especificando en el comando de qué contenedor usar el código de salida: `docker-compose up -exit-code-from project project`.

Los pasos del *pipeline* correspondientes a la fase de pruebas son los siguientes:

```
# Paso 2: Creación de la network de docker para que las dependencias
# de las pruebas puedan comunicarse entre sí
- name: 'gcr.io/cloud-builders/docker'
  args: ['network', 'create', 'oa-net']
  id: 'docker-network'

# Paso 3: Tests. Se utiliza el exitcode de ese contenedor para
# que en caso de fallo no continúe el despliegue.
- name: 'docker/compose:1.23.2'
  args: ['-f', 'docker/docker-compose-cloudbuild.yaml',
         'up', '--exit-code-from', 'project', 'project']
  id: 'run-tests'
```

4.4.5. Automatización del despliegue

Para la automatización del despliegue, se adaptó la política de versionado estudiada en la sección 4.4.2 al *pipeline* de los servicios de Google Compute Engine.

A partir de la gestión de ramas para repositorios colaborativos con Gitflow (ver 2.7), se plantearon dos escenarios:

- Integración de una nueva característica en la rama `develop` del proyecto, tras finalizar una rama del tipo `feature/`.
- Release del proyecto a producción, tras iniciar una rama del tipo `release/` e integrarla con la rama principal `master`.

En el caso del primer escenario, la integración de nuevo código desencadena el despliegue al entorno de desarrollo, tras haber superado las fases de compilación y validación, de forma que se elimina la Plantilla de instancia de desarrollo existente para recrearla después, actualizada con la última versión de código. El grupo de instancias no hace falta modificarlo, pues la Plantilla de instancias que utiliza se llama igual que la anterior. Sin embargo, para garantizar la resiliencia del *pipeline*, se han añadido dos pasos adicionales para borrar y para crear el Grupo de instancia, en el caso en el que no exista o haya sido modificado. Esto será muy útil para servicios nuevos, estando la fase de despliegue totalmente automatizada desde la primera integración. Los pasos de la fase de despliegue al entorno de desarrollo en CloudBuild son los siguientes:

```
1 # Paso 4: Eliminación del instance template antiguo en
  oa-devel
2 - name: 'gcr.io/cloud-builders/gcloud'
3   entrypoint: 'bash'
4   args:
5   - '-c'
6   - |
7     gcloud compute instance-groups managed delete project \
```

```
8       && gcloud compute instance-templates delete project \
9       || exit 0
10    id: 'delete-dev-old-instance-template&group'
11
12  # Paso 5: Creación del instance template en oa-devel
13  - name: 'gcr.io/cloud-builders/gcloud'
14    entrypoint: 'bash'
15    args:
16    - '-c'
17    - |
18      gcloud compute instance-templates create-with-container
        project || exit 0
19    id: 'create-dev-instance-template'
20
21  # Paso 6: Creación del instance group en oa-devel
22  - name: 'gcr.io/cloud-builders/gcloud'
23    entrypoint: 'bash'
24    args:
25    - '-c'
26    - |
27      (gcloud compute instance-groups managed create project
        \
28      --zone europe-west1-d --template project --size 0
        --project oa-devel && \
29      gcloud beta compute instance-groups managed
        set-autoscaling project || exit 0
30    id: 'create-dev-instance-group'
```

De cara al segundo escenario, también se sigue la política de versionado. Cuando se realice una release, el sistema la detectará a través de su etiqueta con el número de versión utilizando la expresión regular `v[0-9]{4}-[0-9]{2}-[a-z]`, dado que el versionado sigue el formato `v<año>-<semana>-<versión>`. En el caso de las releases, también se realiza el despliegue en el entorno de desarrollo, además del despliegue a producción. Para el despliegue a producción, primero se creará la Plantilla de instancia

correspondiente a la nueva versión, para después actualizar el Grupo de instancias correspondiente. De nuevo, se añaden los pasos de creación del Grupo de Instancias por si no existiera. Los pasos de CloudBuild correspondientes al despliegue a producción son los siguientes:

```
# Paso 7: Creación del instance template en oa-prod. Solo para releases.
- name: 'gcr.io/cloud-builders/gcloud'
  entrypoint: 'bash'
  args:
  - '-c'
  - |
    echo "¡Solo para releases!"
    [[ "$TAG_NAME" =~ v[0-9]{4}-[0-9]{2}-[a-z] ]] && gcloud compute instance-templates create-with-container project-$TAG_NAME || echo "No es una release"
  id: 'create-prod-instance-template'

# Paso 8: Creación del instance group en oa-prod. Solo para releases.
- name: 'gcr.io/cloud-builders/gcloud'
  entrypoint: 'bash'
  args:
  - '-c'
  - |
    echo "¡Solo para releases!"
    ([[ "$TAG_NAME" =~ v[0-9]{4}-[0-9]{2}-[a-z] ]] && (gcloud compute instance-groups managed create project && \
    gcloud beta compute instance-groups managed set-autoscaling project \ || exit 0)) || echo "No es una release"
  id: 'create-prod-instance-group'

```

```
# Paso 9: Actualización del instance template que utiliza el instance group
# de oa-prod. Solo para releases.
- name: 'gcr.io/cloud-builders/gcloud'
  entrypoint: 'bash'
  args:
  - '-c'
  - |
    echo "¡Solo para releases!"
    ([[ "$TAG_NAME" =~ v[0-9]{4}-[0-9]{2}-[a-z] ]] &&
    (gcloud compute instance-groups managed
    set-instance-template project \
    --template project-$TAG_NAME || exit 0)) || echo "No es una release"
  id: 'update-prod-instance-group'
```

Por último, la imagen que ha sido probada, compilada y desplegada debe ser subida al registro de imágenes para que pueda ser utilizada por el servicio. La imagen de docker durante la ejecución del *pipeline* ha sido etiquetada como `CloudBuild`, para determinar que se trata de una imagen en pruebas. Si la ejecución ha terminado con éxito, la imagen se reetiqueta a `develop`, para usarse en el entorno de desarrollo, y a la versión que corresponda si se trata de una *release*.

```
# Paso 10: Cambio de etiqueta de la imagen a la etiqueta que se subirá al Container Registry
- name: 'gcr.io/cloud-builders/docker'
  entrypoint: 'bash'
  args:
  - '-c'
  - |
    docker tag gcr.io/oa-devel/project:cloudbuild
    gcr.io/oa-devel/project:develop \
    && [[ "$TAG_NAME" =~ v[0-9]{4}-[0-9]{2}-[a-z] ]] &&
    docker tag gcr.io/oa-devel/project:cloudbuild \
```

```
9       gcr.io/oa-devel/project:$TAG_NAME || exit 0
10    id: 'tag-image'
11
12  # Paso 11: Se borra la tag de la imagen que se usaba en los
    tests
13  - name: 'gcr.io/cloud-builders/docker'
14    args: ['rmi', 'gcr.io/oa-devel/project:cloudbuild']
15    id: 'remove-test-image'
16
17  # Paso 12: Push de la(s) imagen(es) al Container Registry
18  images: ['gcr.io/oa-devel/project']
```

Los comandos de despliegue han sido resumidos en estos fragmentos de código porque se alejan del alcance de este proyecto. El pipeline completo se puede encontrar en el anexo B.

4.5. Google Firebase

Google Firebase es la plataforma de desarrollo de aplicaciones web y aplicaciones móviles de Google Cloud Platform. Incluye un conjunto de herramientas para crear y sincronizar proyectos fácilmente, utilizando la infraestructura de Google y teniendo la posibilidad de escalar automáticamente.

4.5.1. Automatización de las pruebas

El proyecto de Google Firebase de *OrbitalAds* está desarrollado en JavaScript. Para poder ejecutar las pruebas hay que instalar, en primer lugar, todas las dependencias del proyecto. Para realizar todos estos pasos se ha utilizado el compilador nativo de `npm` de CloudBuild. Los pasos correspondientes a la automatización de las pruebas son las siguientes:

```
steps:

- name: 'gcr.io/cloud-builders/npm'
  args: [ 'install' ]
  id: 'install-dependencies'

- name: 'gcr.io/cloud-builders/npm'
  args: [ 'test:unit' ]
  id: 'run-unittests'
```

4.5.2. Automatización de la compilación

En CloudBuild no existe un compilador nativo para Google Firebase. Por lo tanto, el primer paso para automatizar la compilación será generar un compilador propio. Para ello, se necesita crear una imagen de Docker que contenga las herramientas de línea de comandos de Google Firebase. El `Dockerfile` quedaría de la siguiente forma:

```
# use latest Node LTS (Boron)
FROM node:boron
# install Firebase CLI
RUN npm install -g firebase-tools

ENTRYPOINT ["/usr/local/bin/firebase"]
```

Para poder utilizar esta imagen, hay que compilarla y subirla al registro de imágenes privado en Google Container Registry. Este paso se puede realizar con CloudBuild, definiendo el paso correspondiente:

```
steps:
- name: 'gcr.io/cloud-builders/docker'
  args: [ 'build', '-t', 'gcr.io/[PROJECT_ID]/firebase',
  '.' ]
```

```
images:
- 'gcr.io/[PROJECT ID]/firebase'
```

Tras esto, se podrá utilizar esta imagen como compilador en los pasos de CloudBuild. El siguiente paso es analizar cómo se realizan los despliegues a través de la linea de comandos a Firebase.

Para poder desplegar a Firebase a través de una cuenta de servicio, se necesita generar un token de autenticación a través del comando `firebase login:ci`. Bastaría con añadir el comando de despliegue incluyendo el token generado para poder automatizar la operación de despliegue.

```
steps:
  - name: 'gcr.io/cloud-builders/npm:node-6.14.4'
    args: [ 'install' ]
  - name: 'gcr.io/[PROJECT_ID]/firebase'
    args: [ 'deploy', '--project', [PROJECT_ID], '--token',
    '[TOKEN]']
```

Sin embargo, añadir un token privado en claro y distribuirlo, tanto en el repositorio como en las máquinas de los desarrolladores, no es una buena práctica. Para solucionar esto se utilizará Google Key Management Service.

4.5.3. Google Key Management Service

Google Key Management Service es un servicio de gestión de claves de Google Cloud Platform. Permite administrar claves criptográficas, integrándolo, además con los permisos de Google Cloud IAM para poder gestionar las claves concretas y supervisar cómo se usan.

Con este servicio se puede cifrar el token de autenticación de Firebase, para que no sea distribuido en claro. En CloudBuild, utilizaremos el token cifrado de forma que sea la

cuenta de servicio la encargada de descifrarlo. Para ello se utilizará el campo `Secrets`, con el cual se puede emparejar el valor de una variable con una clave secreta de Google Key Management Service. El paso de CloudBuild con el token cifrado quedaría así:

```
- name: 'gcr.io/oa-devel/firebase'
  args: [ 'deploy', '--project', 'oa-front' ]
  secretEnv: ['FIREBASE_TOKEN']

secrets:
- kmsKeyName: 'projects/oa-devel/locations/global/keyRings/
keyring/cryptoKeys/firebase'
  secretEnv:
    FIREBASE_TOKEN: 'CiQAQX/cBY+bKpbQaaDbaNSPRaKX3f...'
```

Los activadores de este tipo de servicios se crean de la misma forma que para las Google Cloud Functions y los servicios de Google Compute Engine. Un ejemplo de *pipeline* completo de Google Firebase se puede encontrar en el anexo C.

5. CONCLUSIONES

La adopción del sistema de Entrega Continua ha reportado numerosos beneficios a *OrbitalAds*. Tras la finalización del proyecto, es interesante volver a evaluar el estado del *pipeline* de integración, como se hizo en la sección 4.1.

Paso	Completado	Justificación
Adoptar un control de versiones	✓	Git + Gitflow en Google Cloud Source Repositories
Definición de release	✓	Características nuevas en el código.
Análisis de código	✓	Se ha diseñado un pipeline específico para GCF, GCE y GFB.
Automatización de la compilación	✓	El contenedor se compila automáticamente, haciendo uso de caché.
Automatización de las pruebas	✓	Implementada una fase automática de pruebas unitarias y de integración.
Automatización del despliegue	✓	Despliegues automáticos para cada tipo de servicio.

Tabla 5: Análisis del estado del Pipeline tras la finalización del proyecto

En primer lugar, la integración frecuente de cambios más pequeños en el código ha llevado a reducir la propensión a errores en los servicios y los riesgos de despliegues erróneos. La incorporación de una fase de pruebas continua en el ciclo de desarrollo ha servido para garantizar el correcto funcionamiento de los servicios, pudiendo así aumentar la confianza y satisfacción del equipo de desarrollo.

Gracias a la rápida retroalimentación de las ejecuciones del *pipeline*, se ha podido mejorar el tiempo de resolución de errores, aumentando la calidad del producto en general. La retroalimentación da visibilidad a todo el equipo de desarrollo del estado del servicio, así como trazabilidad de errores para poder identificar la causa raíz y encontrar la solución adecuada rápidamente.

También, hay que destacar la mejora en los tiempos de comercialización o *time to market*. En un mercado en constante cambio, la mejora del tiempo de respuesta a las nuevas necesidades de los clientes o de convertir una nueva idea o concepto en una realidad es notablemente más rápido. En *OrbitalAds* se hacían, de media, de tres a cuatro despliegues por semana. Desde la implementación de la Entrega Continua, esta métrica ha aumentado considerablemente hasta los cinco despliegues diarios.

Además, al estar el proceso de despliegue totalmente automatizado, el tiempo ne-

cesario para realizar este proceso se ha reducido considerablemente. Con el método de despliegue anterior, donde desde las pruebas hasta el propio despliegue era un proceso manual, se tardaba una media de 44 minutos en realizar un despliegue. Con la automatización con CloudBuild, este tiempo se ha reducido a apenas 4 minutos. Un proceso once veces más rápido que se traduce en ahorro de costes y en una mejoría en la velocidad de desarrollo.

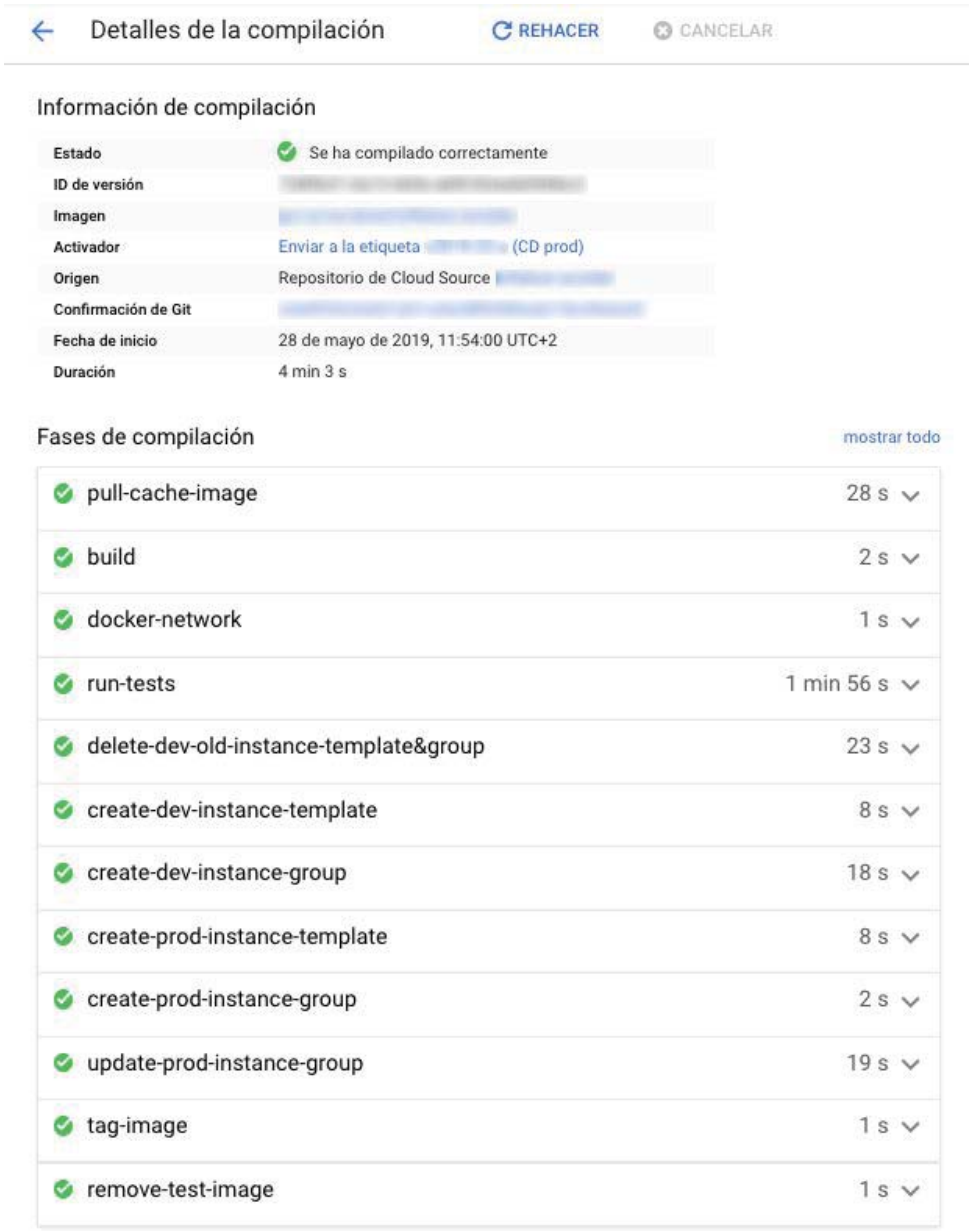

Figura 22: Resultados de la ejecución del pipeline de despliegue.

5.1. Líneas futuras

Durante el desarrollo del proyecto se plantearon una serie de características que podrían mejorar el sistema y la experiencia de usuario.

- **Notificaciones de Slack:** Para poder garantizar la retroalimentación inmediata al desarrollador, convendría enviar notificaciones a un canal de Slack con los resultados de las ejecuciones del *pipeline* de integración. En *OrbitalAds* se utiliza Slack como principal medio de comunicación interno, por lo que todos los miembros del equipo de desarrollo están habituados a su uso y funcionamiento.
- **SonarQube:** Con el fin de mejorar la calidad de código y reducir la propensión de errores, sería interesante implementar una fase automatizada para medir el nivel de calidad del código. SonarQube es una plataforma de código abierto para el análisis de calidad, utilizando métricas que ayudan a disminuir la cantidad de errores.
- **Mejora de imágenes de Docker:** Adicionalmente al uso de caché para mejorar la velocidad de compilación, se podría mejorar más utilizando imágenes base etiquetadas como *alpine*. Este tipo de imágenes hacen uso de distribuciones minimalistas de Linux, reduciendo el espacio que ocupan y, sobre todo, el tiempo que se tarda en descargarlas. Sin embargo, este cambio no es trivial. Hay que estudiar las dependencias de cada servicio para comprobar que no se pierde ninguna herramienta necesaria para la correcta ejecución de los mismos.
- **Análisis del sistema en servicios de *Machine Learning*:** Para poder utilizar el sistema planteado en servicios de *Machine Learning*, se deben analizar los procesos y necesidades concretas de Entrega Continua en este tipo de entornos, evaluando si se puede utilizar el mismo ciclo de desarrollo que en proyectos que no hagan uso de esas tecnologías y viendo en qué parte del *pipeline* encajaría la fase de entrenamiento de modelos.

5.2. Valoración personal

Implementar un sistema de Entrega Continua puede ser difícil, pero no deja de ser una tarea desafiante. Frecuentemente, adaptar la cultura organizacional a un nuevo proceso suele ser el principal obstáculo a la hora de adoptar esta práctica, pero en *OrbitalAds* no ha sido el caso.

El reto que ha habido que afrontar, y que ha llevado a realizar este proyecto, ha sido superar una serie de dificultades técnicas, como la gestión de la configuración, la integración continua o automatizar las pruebas y despliegues.

Entre las aportaciones que este trabajo me ha proporcionado, puedo destacar el haber podido comprobar la importancia que tiene la automatización de procesos en las metodologías ágiles. Otro reto al que este trabajo se ha tenido que enfrentar es lo incipiente del mercado de las herramientas en este ámbito, donde la plataforma escogida, Google CloudBuild, aporta un indiscutible liderazgo y potencial para las empresas, aunque está dando sus primeros pasos y debe reforzar sus recursos internos mientras gana en madurez de producto, documentación y soporte.

ANEXO A PIPFILE

Ejemplo de `Pipfile`:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[packages]
requests = "*"

[dev-packages]
pytest = "*"
```

ANEXO B PIPELINE COMPLETO DE GCE

Ejemplo de Pipeline para servicios de Google Compute Engine:

```
steps:

# Paso 0: Descargar la imagen anterior para poder utilizarla como caché de la nueva.
# Reduce mucho el tiempo del build
- name: 'gcr.io/cloud-builders/docker'
  entrypoint: 'bash'
  args:
  - '-c'
  - |
    docker pull gcr.io/oa-devel/nombre:develop || exit 0
  id: 'pull-cache-image'

# Paso 1: Build de la imagen de docker
- name: 'gcr.io/cloud-builders/docker'
  args: ['build', '-t', 'gcr.io/oa-devel/nombre:cloudbuild',
         '--cache-from', 'gcr.io/oa-devel/nombre:develop', '.']
  id: 'build'

# Paso 2: Creación de la network de docker para que las dependencias
# de los tests puedan comunicarse entre sí
- name: 'gcr.io/cloud-builders/docker'
  args: ['network', 'create', 'nombrenetwork']
  id: 'docker-network'

```

```
25 # Paso 3: Tests. Se utiliza el exitcode de ese contenedor
   para
26 # que en caso de fallo no continúe el build y despliegue.
27 - name: 'docker/compose:1.23.2'
28   args: ['-f', 'docker/docker-compose-cloudbuild.yaml',
29          'up', '--exit-code-from', 'nombre', 'nombre']
30   id: 'run-tests'
31
32 - name: 'gcr.io/cloud-builders/gcloud'
33   entrypoint: 'bash'
34   args:
35   - '-c'
36   - |
37     gcloud compute instance-groups managed delete nombre \
38     --project oa-devel --zone europe-west1-d --quiet \
39     && gcloud compute instance-templates delete nombre \
40     --project oa-devel --quiet|| exit 0
41   id: 'delete-dev-old-instance-template&group'
42
43 # Paso 4: Creación del instance template en oa-devel
44 - name: 'gcr.io/cloud-builders/gcloud'
45   entrypoint: 'bash'
46   args:
47   - '-c'
48   - |
49     gcloud compute instance-templates create-with-container
     nombre \
50     --machine-type n1-standard-1 --container-image
     gcr.io/oa-devel/nombre:develop \
51     --container-restart-policy never --service-account
     {service account} \
52     --scopes=cloud-platform \
53     --container-env-file env_devel.env --project oa-devel
     || exit 0
```

```
54    id: 'create-dev-instance-template'
55
56  # Paso 5: Creación del instance group en oa-devel
57  - name: 'gcr.io/cloud-builders/gcloud'
58    entrypoint: 'bash'
59    args:
60    - '-c'
61    - |
62      (gcloud compute instance-groups managed create nombre \
63      --zone europe-west1-d --template nombre --size 0
      --project oa-devel && \
64      gcloud beta compute instance-groups managed
      set-autoscaling nombre \
65      --max-num-replicas 10 --min-num-replicas 0 \
66      --update-stackdriver-metric
      pubsub.googleapis.com/subscription/num_undelivered_messages
      \
67      --stackdriver-metric-single-instance-assignment 1 \
68      --stackdriver-metric-filter 'resource.type =
      pubsub_subscription AND resource.label.subscription_id
      = "nombre"' \
69      --cool-down-period 60 --zone europe-west1-d --project
      oa-devel) || exit 0
70    id: 'create-dev-instance-group'
71
72  # Paso 6: Creación del instance template en oa-prod. Solo
  para releases.
73  - name: 'gcr.io/cloud-builders/gcloud'
74    entrypoint: 'bash'
75    args:
76    - '-c'
77    - |
78      echo "¡Solo para releases!"
```

```
79      [[ "$TAG_NAME" =~ v[0-9]{4}-[0-9]{2}-[a-z] ]] && gcloud
        compute instance-templates create-with-container
        nombre-$TAG_NAME \
80      --machine-type n1-standard-1 --container-image
        gcr.io/oa-devel/nombre:$TAG_NAME \
81      --container-restart-policy never --service-account
        {service account} \
82      --scopes=cloud-platform \
83      --container-env-file env_prod.env --project oa-prod ||
        echo "No es una release"
84    id: 'create-prod-instance-template'
85
86  # Paso 7: Creación del instance group en oa-prod. Solo para
    releases.
87  - name: 'gcr.io/cloud-builders/gcloud'
88    entrypoint: 'bash'
89    args:
90    - '-c'
91    - |
92      echo "¡Solo para releases!"
93      ([[ "$TAG_NAME" =~ v[0-9]{4}-[0-9]{2}-[a-z] ]] &&
        (gcloud compute instance-groups managed create nombre \
94      --zone europe-west1-d --template nombre-$TAG_NAME
        --size 0 --project oa-prod && \
95      gcloud beta compute instance-groups managed
        set-autoscaling nombre \
96      --max-num-replicas 10 --min-num-replicas 0 \
97      --update-stackdriver-metric
        pubsub.googleapis.com/subscription/num_undelivered_messages
        \
98      --stackdriver-metric-single-instance-assignment 1 \
99      --stackdriver-metric-filter 'resource.type =
        pubsub_subscription AND resource.label.subscription_id
        = "nombre"' \
```

```
100       --cool-down-period 60 --zone europe-west1-d --project
          oa-prod || exit 0)) || echo "No es una release"
101     id: 'create-prod-instance-group'
102
103   # Paso 8: Actualización del instance template que utiliza
      el instance group
104   # de oa-prod. Solo para releases.
105   - name: 'gcr.io/cloud-builders/gcloud'
106     entrypoint: 'bash'
107     args:
108     - '-c'
109     - |
110       echo "¡Solo para releases!"
111       ([[ "$TAG_NAME" =~ v[0-9]{4}-[0-9]{2}-[a-z] ]] &&
          (gcloud compute instance-groups managed
          set-instance-template nombre \
112       --template nombre-$TAG_NAME --zone europe-west1-d
          --project=oa-prod || exit 0)) || echo "No es una
          release"
113     id: 'update-prod-instance-group'
114
115   # Paso 9: Cambio de etiqueta de la imagen a la etiqueta que
      se subirá al Container Registry
116   - name: 'gcr.io/cloud-builders/docker'
117     entrypoint: 'bash'
118     args:
119     - '-c'
120     - |
121       docker tag gcr.io/oa-devel/nombre:cloudbuild
          gcr.io/oa-devel/nombre:develop \
122       && [[ "$TAG_NAME" =~ v[0-9]{4}-[0-9]{2}-[a-z] ]] &&
          docker tag gcr.io/oa-devel/nombre:cloudbuild \
123       gcr.io/oa-devel/nombre:$TAG_NAME || exit 0
124     id: 'tag-image'
```

```
125
126 # Paso 10: Se borra la tag de la imagen que se usaba en los
    tests
127 - name: 'gcr.io/cloud-builders/docker'
128   args: ['rmi', 'gcr.io/oa-devel/nombre:cloudbuild']
129   id: 'remove-test-image'
130
131 # Paso 11: Push de la(s) imagen(es) al Container Registry
132 images: ['gcr.io/oa-devel/nombre']
```

ANEXO C PIPELINE COMPLETO DE GFB

Ejemplo de Pipeline para servicios de Google Firebase:

```
steps:

- name: 'gcr.io/cloud-builders/npm'
  args: [ 'install' ]
  id: 'install-dependencies'

- name: 'gcr.io/cloud-builders/npm'
  args: [ 'test:unit' ]
  id: 'run-unittests'

- name: 'gcr.io/cloud-builders/npm'
  args: [ 'run', 'build:pro' ]
  id: 'compile-binaries'

- name: 'gcr.io/cloud-builders/npm'
  args: [ 'run', 'build:storybook' ]
  id: 'compile-binaries-storybook'

- name: 'gcr.io/oa-devel/firebase'
  args: [ 'deploy', '--project', 'oa-front' ]
  secretEnv: ['FIREBASE_TOKEN']
  id: 'deploy'

secrets:
- kmsKeyName: 'projects/oa-devel/locations/global/keyRings/
keyring/cryptoKeys/firebase'
  secretEnv:
    FIREBASE_TOKEN: 'CiQAQX/cBY+bKpbQaaDbaNSPRaKX3...'
```

REFERENCIAS

[1] L. Lindstrom and R. Jeffrie, *Extreme Programming and Agile software development methodologies*. Information systems management, 2004, vol. 21, no 3, p. 41-52.

[2] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2011.

[3] A. Matarranz (2011, feb 13) *Los orígenes de Agile* [Online]. Available: https://conversisconsulting.com/2011/02/13/los-origenes-de-agile-1/

[4] J. Izquierdo (2014, sep 04) *¿Qué es el XP Programming?* [Online]. Available: https://www.iebschool.com/blog/que-es-el-xp-programming-agile-scrum/

[5] M. Fowler (2006, may 01) *Continuous Integration* [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[6] S. Yamada et Al. *Software reliability growth models with testing-effort*. IEEE Transactions on Reliability, 1986, vol. 35, no 1, p. 19-23.

[7] ElectricCloud *Continuous Integration Best Practices: Vision and Reality* [Online]. Available: https://electric-cloud.com/plugins/continuous-integration/

[8] O. Villacampa (2017, jul 11) *Qué es Docker y para qué sirve* [Online]. Available: https://www.ondho.com/que-es-docker-para-que-sirve/

[9] I. Villafuente (2016, jul 04) *Docker, qué es y cómo funciona la contenerización* [Online]. Available: https://www.stackfire.com/docker-que-es-y-como-funciona-la-contenerizacion/

[10] R. Meier (2017, feb 10) *What is Google's Cloud Platform?* [Online]. Available: https://medium.com/@retomeier/what-is-googles-cloud-platform-d92a9c9e5e89

[11] Google Cloud Documentation (2018, nov 29) *Continuous delivery tool integrations* [Online]. Available: https://cloud.google.com/container-registry/docs/continuous-delivery

[12] J. Durán (2016, oct 06) *Primeros pasos con Docker Compose* [Online]. Available: https://www.somosbinarios.es/primeros-pasos-con-docker-compose/

[13] A. Antunes (2016, Nov 22) *GitFlow mejora la gestión de tu repositorio Git* [Online]. Available: http://bemobile.es/blog/2016/11/gitflow-mejora-la-gestion-de-tu-repositorio-git/

[14] P. R. Niven and B. Lamorte, *Objectives and key results: Driving focus, alignment, and engagement with OKRs*. John Wiley & Sons, 2016.

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Mon Jun 03 16:11:57 CEST 2019
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)