UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS Y COMPUTACIÓ

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Infraestructura elástica en un entorno de desarrollo ágil

TRABAJO FIN DE MASTER

Máster Universitario en Computación Paralela y Distribuida

Autor: Pablo Borja Hernández
Tutor: Francesc D. Muñoz-Escoí

Curso 2018-2019

Resumen

Cada vez se necesitan infraestructuras distribuidas más complejas que en la medida de lo posible, sean capaces de gobernarse a sí mismas.

En este documento veremos cómo desplegar una infraestructura basada en Kubernetes, capaz de regenerarse sola frente a fallos, así como poder actualizarse sin necesidad de tiempos de caída. También se explicará cómo otorgar elasticidad y cómo monitorizar la infraestructura.

Para esto, primero se definirán los conceptos básicos que rodean Kubernetes. A su vez veremos qué tipos de actualización se pueden realizar en un cluster de este tipo.

Dotaremos a las cargas de trabajo desplegadas en el cluster de escalado horizontal. También conseguiremos elasticidad en los nodos pertenecientes al cluster, para adaptarse así a la carga de trabajo.

Se explicará cómo desplegar el cluster en diferentes proveedores cloud.

Para controlar el estado de toda la infraestructura se monitorizará y crearán alarmas que nos avisarán de cuando algo no está funcionando correctamente.

Además se ha desarrollado una pequeña web reactiva basada en *Vue* para controlar el estado y las versiones desplegadas de las cargas de trabajo entre los diferentes entornos típicos de desarrollo, pre-producción y producción.

Todo esto utilizando herramientas OpenSource y la gran mayoría apadrinadas por la Cloud Native Computing Foundation.

Pablo Borja Hernández
pabborh1@upv.es

Palabras clave: Kubernetes,Infraestructura elástica,Automatización,Rancher,Kops,Docker

CAPÍTULO 1
Introducción

1.1 Motivación

Los contenedores han venido para quedarse. Gracias a ellos hemos llegado a un nuevo nivel de abstracción.

A través de los contenedores podemos desarrollar arquitecturas distribuidas, pero ¿Qué tipo de infraestructuras podemos dar a los contenedores?

Las infraestructuras distribuidas son complejas. Debemos facilitar lo máximo posible la mantenibilidad de estas infraestructuras. Para ello debemos diseñarlas de forma que sean lo mas autónomas posibles.

Debemos de otorgar a las infraestructuras cierta inteligencia para que sean capaces de auto-curarse sin que medie ninguna acción externa al sistema. De esta forma conseguiremos dar servicios de forma más robusta.

Además, si la carga de trabajo que soportamos es muy cambiante, debemos ser capaces de generar sistemas elásticos que puedan absorber esa carga variable de la forma más eficiente posible. La capacidad de cómputo no es ilimitada, debemos adaptar nuestros requerimientos al trabajo que vayamos a soportar.

De la misma forma, también es importante conseguir actualizar nuestros servicios sin que este se vea afectado, para así cumplir con SLA cada vez más estrictos, dando mejor servicio. Así podremos hacer entregas muy frecuentes de código, para conseguir entornos donde arreglar fallos y añadir funcionalidades sea sencillo y rápido.

1.2 Objetivos

El Objetivo de este trabajo es hacer una Infraestructura altamente disponible y lo más autónoma posible.

La infraestructura debe auto-curarse sola, es decir, si algún nodo falla o algún servicio deja de responder de la forma esperada, la infraestructura debe sobreponerse y ser capaz de seguir dando servicio de la forma esperada.

También debe soportar cargas de trabajo variables. Para servicios de cara al público, no es la misma carga la recibida a las 10 de la mañana que a las 10 de la noche. Es por ello que debe adaptarse a la carga recibida.

La actualización de servicios no debe afectar en la disponibilidad del mismo. La infraestructura está contextualizada en un entorno de desarrollo ágil, donde la entrega continua de código está presente.

Debemos ser capaces de saber qué está pasando en el cluster. Por lo tanto es necesaria la monitorización del mismo. Esta monitorización debe tener alarmas automatizadas que avisen por el canal deseado cuando ciertas métricas sean anómalas al comportamiento esperado.

De esta forma conseguiremos una infraestructura robusta, capaz de dar servicio y cumplir con un SLA exigente.

1.3 Estructura del documento

1. **Introducción**
   Introducción al trabajo donde se expone la motivación y los objetivos del mismo.

2. **Arquitectura de la aplicación**
   Descripción de una aplicación distribuida, la cual cumple los patrones para ser desplegada en una infraestructura distribuida.

3. **Kubernetes**
   Se intenta dar una idea general del funcionamiento y posibilidades de configuración de la herramienta open-source Kubernetes.

4. **Escalado Horizontal a medida**
   Una aplicación desarrollada la cual puede gestionar el escalado horizontal de un clúster Kubernetes en base a métricas. *custom*.

5. **Desplegar un clúster Kuberentes**
   En este apartado se explican diferentes formas de desplegar un clúster Kubernetes usando diversas herramientas y en distintos proveedores cloud.

6. **Monitorización**
   Cómo y qué herramientas utilizar para monitorizar una infraestructura distribuida.

7. **Integración Continua**
   Integración continua en un contexto de aplicaciones distribuidas.

8. **Pruebas de Carga**
   Pruebas realizadas para comprobar el funcionamiento de la infraestructura.

9. **Conclusión**
   Conclusiones adquiridas después de realizar el trabajo.

CAPÍTULO 2
Arquitectura de la aplicación

2.1 Introducción

En este apartado se va a describir un ejemplo de aplicación distribuida a la cual podremos otorgar una infraestructura elástica en un entorno de desarrollo ágil.

Vamos a suponer que tenemos una arquitectura basada en microservicios. Estos son mayormente *"stateless"*, no obstante, debido a limitaciones en la arquitectura, podríamos enfrentarnos a servicios los cuales retienen estado.

Una posible arquitectura podría ser la siguiente.

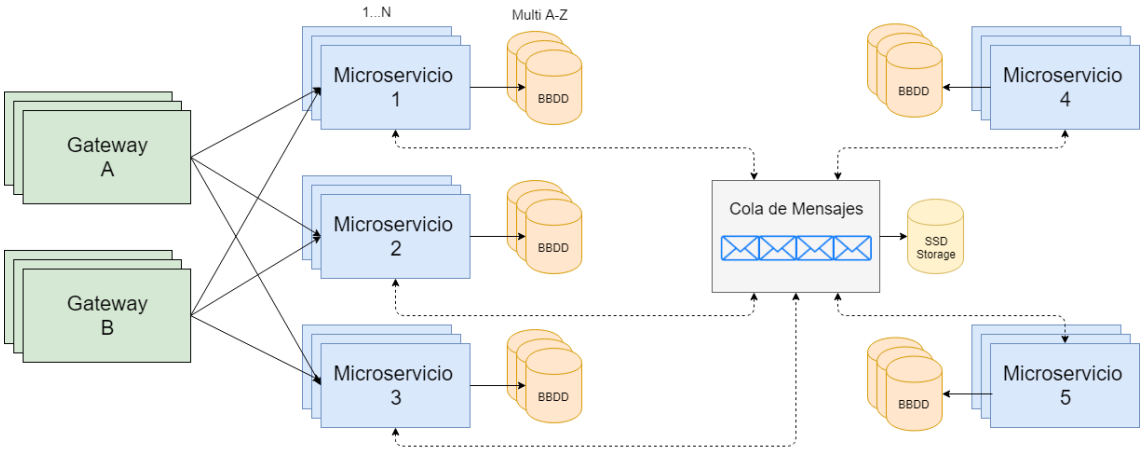

Figura 2.1: Arquitectura de la aplicación

En la figura anterior vemos dos puntos de entrada o *"gateway"*. Estos puntos de entrada realizan peticiones a distintos servicios, a su vez los servicios se comunican entre sí mediante colas de mensajes.

Cada microservicio dispone de una base de datos SQL trabajando en Multi A-Z[17]. Además la cola de mensajes persiste en memoria secundaria.

CAPÍTULO 3
Kubernetes

3.1 Introducción

Kubernetes es un orquestador de contenedores open-source. Con él podemos escalar nuestras aplicaciones y hacer despliegues automatizados. El proyecto Kubernetes fue creado en el entorno de Google y es el sucesor del proyecto Borg. Con el conseguimos un cluster con N nodos capaces de albergar nuestra aplicación distribuida. Para conseguir información más detallada acerca de los proyectos Borg y Kubernetes podemos encontrarla aquí [1][3]

Google liberó Kubernetes en julio de 2015 a la Cloud Native Computing Foundation, contribuyendo así con el open-source. Actualmente va por la versión 1.15.

Esta herramienta de orquestación nos da la posibilidad de tener nuestra infraestructura de aplicación como código. De esta forma podemos llevar nuestra solución software a cualquier plataforma siempre y cuando tengamos un cluster Kubernetes desplegado. La realidad es que todos los proveedores importantes de cómputo en la nube están empezando a ofrecer Kubernetes como servicio, liberando el trabajo que conlleva mantener y desplegar el cluster. Así se consigue una solución software que evita el *"Vendor lock-in"*, pudiendo migrar la solución a cualquier Cloud.

Además, Kubernetes realiza una tarea de monitorización de los contenedores. De esta forma intenta siempre que el número deseado de contenedores esté en funcionamiento, acercándonos a la alta disponibilidad.

3.2 Cargas de Trabajo

Las Cargas de Trabajo (*Workloads*) son la forma de definir lo que se va a ejecutar dentro del cluster Kubernetes.

3.2.1. PODs

Los PODs[8] son la unidad mínima de despliegue en Kubernetes. Normalmente los POD están formados por un contenedor. También se puede definir PODs con varios contenedores, de esta forma obligamos a que estos contenedores se desplieguen siempre juntos en el mismo nodo. Esto puede ser útil, por ejemplo, si los contenedores se comunican a través del sistema de ficheros.

Los POD son mortales, van a ser actualizados, terminados e inicializados, no debemos pensar en ellos como algo permanente e inmutable.

Dentro de la definición de los POD, podemos definir los contenedores mediante multitud de opciones. Las principales son:

- **Nombre del POD**. Nombre del POD para identificarlo dentro del cluster
- **Imagen del contenedor a desplegar**. Mediante la clave *image* definimos el nombre del contenedor que queremos desplegar
- **Variables de entorno**. Indicando la clave 'env:' podemos indicar una lista de nombres de variables y valores que se inyectarán en el contenedor como variables de entorno.
- **Puertos usados por el contenedor**. Se puede incluir una lista de puertos utilizados por el contenedor
- **PullPolicy**. Política de descarga de los contenedores. Por ejemplo podemos indicar que siempre haga un pull de la imagen del contenedor antes de desplegarlo.
- **Reserva y límites de recursos**. Mediante la clave 'resources', podemos definir reservas en los recursos (RAM y CPU) del nodo en el cual será desplegado el nodo. Además también se pueden definir límites.
- **Readiness y Liveness**. Gracias a los ReadinessProbes y LivenessProbes, el cluster Kubernetes puede saber cuando un contenedor está listo para servir tráfico después de iniciarlo y si sigue listo para esta tarea. Estas pruebas pueden ser llamadas REST a cierto endpoint o incluso comandos ejecutados dentro del propio contenedor.

3.2.2. Controladores: Deployment, StatefulSets, DaemonSets y Jobs

Los *Deployment*, *StatefulSets* y *DaemonSets* son controladores de los POD. En ellos se define el POD pero también otros aspectos. Aquí entra en juego la replicación y la actualización de los POD. En ellos también se define cómo elegir los nodos donde van a ser desplegados.

Los POD definidos mediante estos controladores estarán siempre activos hasta que decidamos eliminar el controlador.

Deployment

Los *Deployment*[9] definen intrínsecamente los ReplicaSet, que son los encargados de controlar el número de PODs y la asignación de estos entre los diferentes nodos.

En la definición del *Deployment* principalmente podemos indicar:

- **Nombre del *Deployment***. Nombre del *Deployment* para identificarlo dentro del cluster
- **Etiquetas del *Deployment***. Usando la clave 'labels' podemos definir etiquetas para poder referenciar el *Deployment* en otros componentes del cluster.
- **Número de réplicas deseadas**. El número de réplicas del POD que queremos desplegar
- **Estrategias de actualización**. Podemos elegir entre varias estrategias de actualización del *Deployment*.

- **Historial de versiones**. Si indicamos la clave 'revisionHistoryLimit' en nuestro fichero de definición del *Deployment*, podemos indicar qué número de versiones anteriores se guarda. De esta forma podemos volver a una versión anterior de *Deployment* de forma rápida.

ReplicaSet

El ReplicaSet[10] va ligado al *Deployment*. Este es el encargado de controlar el número de PODs desplegados por el *Deployment* así como de su buen funcionamiento.

Para esta tarea, Kubernetes utiliza los servicios kube-controller y kube-scheduler. De esta forma si un *Deployment* especifica que queremos 5 réplicas de un POD, el ReplicaSet se encargará de que siempre haya 5 réplicas activas y funcionando en el cluster.

Además el ReplicaSet es el encargado de dar soporte a la funcionalidad del historial de versiones mencionado en los *Deployment*, dependiendo de cuántas versiones queramos guardar, podremos volver de forma sencilla a una versión anterior.

Cuando actualizamos un *Deployment* con una versión nueva de una imagen de contenedor, automáticamente se genera un nuevo ReplicaSet encargado de las nuevas réplicas con la nueva imagen. Más adelante se explicará en detalle la forma que tiene Kubernetes de actualizar y controlar las versiones.

StatefulSets

Los *StatefulSets*[11] son similares a los *Deployment*, pero a diferencia de estos, están pensados para PODs que requieren identificadores de red únicos, almacenamiento persistente, despliegues ordenados o actualizaciones en cierto orden. Además los PODs mantienen un identificador único que persiste aunque sean reasignados a otros nodos a diferencia de los POD controlados por los *Deployment*.

Los volúmenes de datos que manejan estos controladores deben ser del tipo PersistentVolume de los cuales se hablará más adelante.

DaemonSets

Los *DaemonSets* son una variante de los *Deployment*. Estos están pensados para que los POD actúen como demonios y sean asignados a todos los nodos del cluster.

No obstante, al igual que con el resto de controllers, podemos decidir los nodos afines, restringiendo así el despliegue en los nodos que deseemos.

Jobs

A diferencia del resto de controladores, que están pensados para una ejecución continua, los *Jobs*[12] se encargan de hacer una sola ejecución.

Los *Jobs* son controladores encargados de que se ejecuten de forma satisfactoria los PODs definidos en ellos. Por ejemplo, si queremos realizar trabajos parametrizados Batch, crearemos una imagen de contenedor preparada para esta ejecución y parametrizaremos las definiciones de los *Jobs* para posteriormente lanzarlas en el cluster.

3.3 Persistencia de estado

Si necesitamos que nuestros servicios persistan el estado, podemos hacerlo mediante volúmenes persistentes. Para hacer uso de ellos debemos definir el Volumen Persistente (*PersistentVolume*) y reclamar dicho volumen mediante los *PersistentVolumeClaim*.

Dependiendo de nuestros requerimientos deberemos elegir qué tipo de volumen configurar.

- **HostPath**. Si queremos un comportamiento similar a los volúmenes en *Docker*, deberemos configurar el volumen como *HostPath*. De esta forma conectaremos una ruta del Nodo donde se vaya a desplegar el *POD*, con una ruta interna al *POD*.

  Sin embargo si necesitamos persistir el estado, este solo se encuentra en el nodo donde se está ejecutando el *POD*. Por lo tanto deberemos tomar medidas adicionales para asegurarnos de que si el *POD* muere, mantenga dicho estado.

  Por ejemplo podemos forzar al *POD* a desplegarse en un nodo en concreto mediante la definición del *POD*. De esta forma el Nodo siempre mantendrá el estado.

- **Volumen externo**. Por otro lado, si estamos usando la infraestructura que nos ofrece un proveedor cloud, como por ejemplo AWS, Azure o GCE podemos utilizar volúmenes externos de dicho proveedor.

  De esta forma tendremos volúmenes externos al cluster, los cuales se conectarán al nodo donde se vaya a desplegar el *POD*. Gracias a este tipo de configuración, podemos persistir el estado independientemente de dónde se despliegue nuestro *POD*.

- **Volumen compartido**. También podemos hacer uso de volúmenes compartidos entre *POD*. Si necesitamos *PODs* que compartan estado entre ellos, podemos hacerlo mediante volúmenes de tipo *HostPath* en caso de que todos se desplieguen en la misma máquina o bien usando volúmenes compartidos como por ejemplo volúmenes NFS.

3.4 Servicios

Ya hemos visto cómo se definen las cargas de trabajo. Pero, ¿cómo podemos comunicarlas entre ellas? y ¿cómo exponemos esas cargas de trabajo al exterior?

Los Servicios son una abstracción de las cargas de trabajo. Si las cargas de trabajo fueran el *BackEnd*, los Servicios (*Services*) serían el *FrontEnd*.

Tanto los *POD* como los Servicios reciben una IP interna en el cluster, sin embargo, los *POD* son mortales. Por lo tanto, si queremos acceder a ellos, la mejor forma es a través de los Servicios.

Por ejemplo, si tenemos tres réplicas de un *POD* ofreciendo una funcionalidad a través del puerto 90, podemos configurar un Servicio que sirva a través del puerto 9000. De esta forma si alguno de los *POD* deja de funcionar, si accedemos a través del Servicio, Kubernetes redirigirá la petición hacia uno de los POD en funcionamiento.

Para vincular los *POD* y los Servicios se utilizan etiquetas en los *POD* y selectores de estas en los Servicios. Por ejemplo si queremos exponer un servicio soportado por un determinado *Deployment*, bastará con etiquetar dicho *Deployment* para luego poder hacer referencia a la etiqueta en la especificación del Servicio. De esta manera podríamos tener un servicio que englobara dos *Deployment* distintos si estos tuvieran la misma etiqueta.

Además de la IP asignada a los Servicios, Kubernetes ofrece resolución de nombres de los Servicios dentro del cluster, de esta forma podremos comunicarnos entre *PODs* vía nombre de los Servicios previamente configurados.

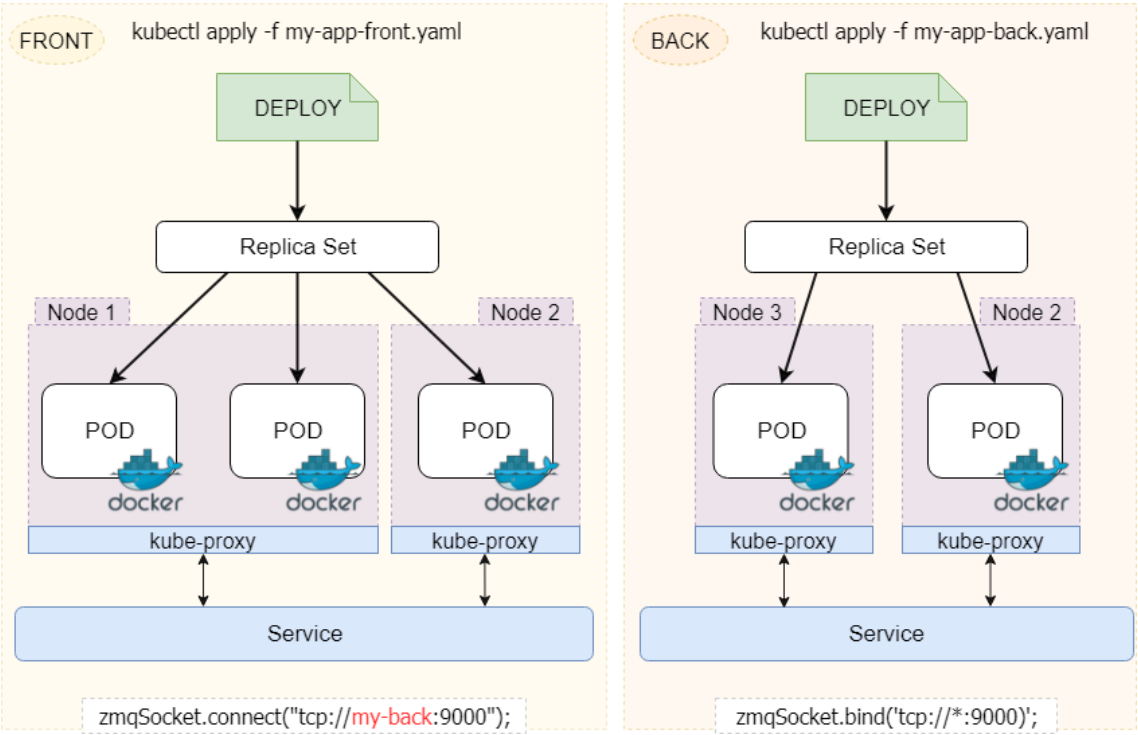

Figura 3.1: Arquitectura de comunicación a través de Servicios

En la figura anterior vemos una distribución con la cual podemos resistir a la caída de cualquiera de los nodos. Esto es así debido a que ningún nodo contiene la totalidad de las réplicas existentes de un POD. De esta forma si algún nodo cae, siempre habrá un POD vivo. Además gracias a los *ReplicaSet* se volverían a crear los PODs pertenecientes al nodo caído.

Gracias a la abstracción de los servicios, Kubernetes, junto a una buena distribución de las cargas de trabajo, nos ofrece tolerancia a caídas de nodos.

Existen principalmente dos tipos de servicios:

- **ClusterIP**. Los servicios de tipo ClusterIP son internos al cluster y solo se puede acceder a ellos de forma interna al cluster.

- **NodePort**. Los servicios NodePort se utilizan para exponer el servicio a través de un puerto hacia fuera del cluster además del direccionado interno. Para esto Kubernetes guarda un rango de puertos. Estos puertos deben ser únicos por Servicio.

  Por ejemplo si un *POD* sirviese a través del puerto 80 y el servicio relacionado con el *POD* sirviese a través del puerto 8080, podríamos definir dicho servicio para que a su vez sirviese hacia el exterior en el puerto 30000.

En este artículo [6] se habla del funcionamiento de los servicios en Kubernetes a un nivel más bajo, explicando también la implicación de kube-proxy, la herramienta interna de Kubernetes que se encarga del direccionamiento. En este otro artículo [7] también se habla acerca de los servicios, cuándo usarlos y el papel que juegan los *Ingress*

Ingress

Si quisiéramos acceder a través de HTTP a nuestros servicios, deberíamos hacerlo a través de los *Ingress*. Este objeto de Kubernetes ofrece acceso a través de HTTP, además, también proporciona terminación SSL y balanceo de carga.

Su forma de funcionar es sencilla, se define la dirección HTTP del host, el servicio a exponer y el puerto por donde queremos que entren las peticiones.

3.5 Actualización de las Cargas de Trabajo.

Si nuestro propósito es el de ofrecer un servicio mediante nuestras cargas de trabajo es posible que queramos actualizarlo a lo largo de su ciclo de vida. Dependiendo de nuestra arquitectura de aplicación deberemos aplicar metodologías más o menos complejas.

Si nuestra aplicación no requiere de trasformaciones de estado, podemos configurar fácilmente varias estrategias de actualización dentro del cluster. Nativamente Kubernetes ofrece la estrategia de *Rolling Update*. Además puede implementarse otras estrategias como *Blue/Green* o *Canary Release* gracias a la implementación de los servicios en Kubernetes.

También existe la posibilidad de eliminar los PODs obsoletos y volver a crearlos con la nueva versión, pero estaremos incurriendo en indisponibilidad. No obstante dependiendo de la naturaleza de la carga de trabajo puede ser una buena opción.

Pero antes de detallar las estrategias de actualización, hay que conocer las *Readiness* y *Liveness Probes* que ofrece Kubernetes.

3.5.1. Readiness Probe y Liveness Probe

Estas dos pruebas son fundamentales a la hora de garantizar la alta disponibilidad del servicio. ¿Cómo sabemos cuando un POD está listo para servir tráfico? ¿Está el POD vivo?

Estas dos preguntas son contestadas por las dos pruebas de *Readiness* y *Liveness*.

Ambas pruebas admiten exactamente los mismos parámetros, pero el *Readiness* se utiliza para saber cuándo un POD se ha iniciado y está listo para servir peticiones. Mientras que el *Liveness* se encarga de consultar la salud del POD, es decir, si sigue listo para servir peticiones.

Para determinar el éxito o fracaso de la prueba, se pueden configurar cuál es el número límite de comprobaciones fallidas, así como la frecuencia de estas. Si alguna de estas dos pruebas falla, el POD será reiniciado.

Esta prueba puede configurarse de dos formas. Como un comando ejecutado en el POD, por ejemplo, buscar cierto fichero creado cuando el POD está completamente arrancado o como una petición HTTP a cierto endpoint. Si el comando devuelve 0 como código de retorno o la petición HTTP un 200 OK se da como satisfactoria la prueba.

3.5.2. Rolling Update

Esta estrategia es la soportada nativamente por Kubernetes y se puede configurar de varias formas. El principal requerimiento para nuestra carga de trabajo de esta estrategia de actualización es permitir la coexistencia de versiones, pues existirán intervalos de

tiempo donde dos versiones distintas sirvan peticiones de forma simultánea a través de los Servicios.

La manera de funcionar del *Rolling Update* es la siguiente.

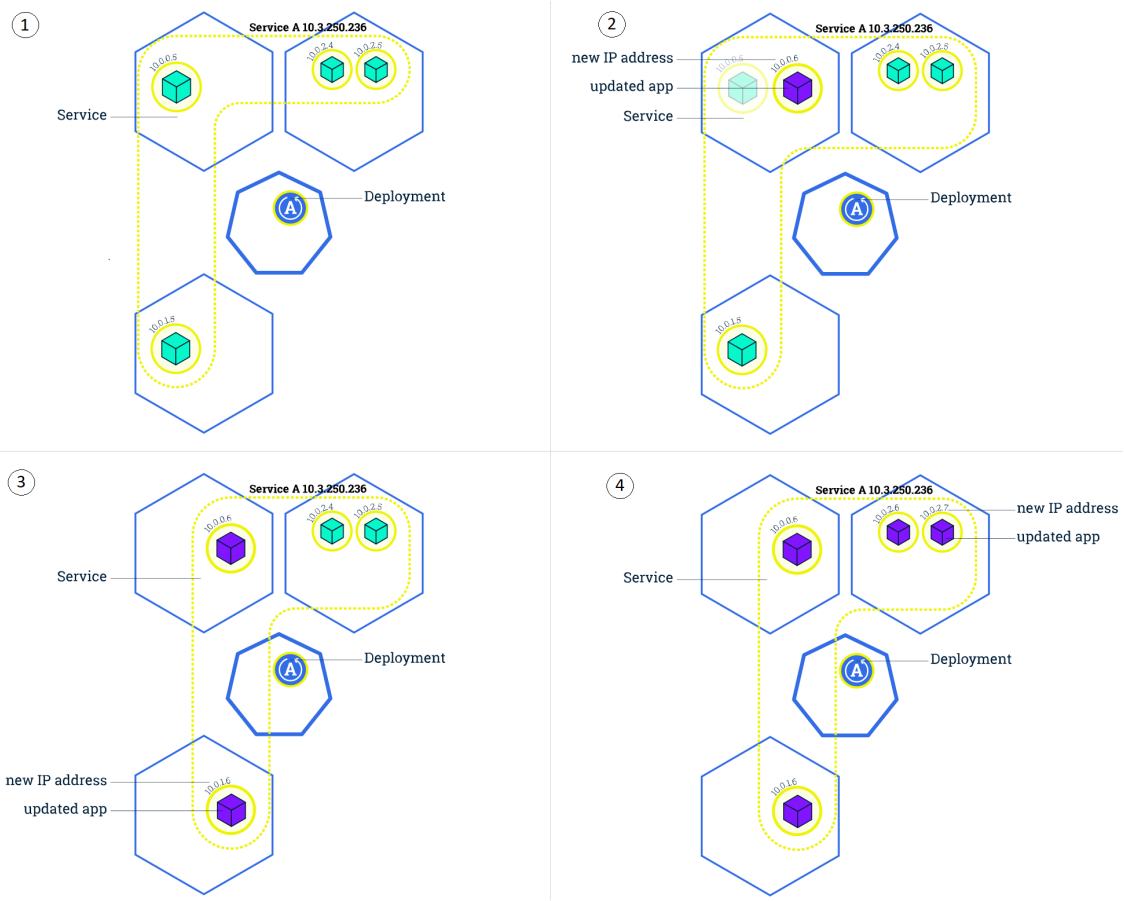

Figura 3.2: Estrategia Rolling Update. Imagen de kubernetes.io

En la figura anterior vemos un *Deployment* el cual es englobado por un Servicio. Las peticiones de nuestros clientes serán recibidas a través de dicho servicio. El *Deployment* con 4 réplicas va a ser actualizado.

En el segundo paso, observamos como un nuevo POD con la nueva versión se ha desplegado. Cuando el cluster a través del *ReadinessProbe* cree que ya puede servir tráfico, se elimina la versión vieja. Quedando las 4 réplicas del *Deployment*.

Una vez ha terminado con la actualización de una de las réplicas, se pasa a actualizar la siguiente. Así hasta completar la actualización en todas ellas.

Como detalle también muestra la política de asignación de IPs internas a los POD. La cual cambia con cada POD generado.

Durante la actualización conviven aplicaciones con versiones diferentes. Kubernetes asume que esta coexistencia es posible.

Los momentos de actualización son críticos a la hora de cumplir determinado SLA. Para ello Kubernetes ofrece la posibilidad de configurar el Rolling Update.

Para poder hacer frente a todas las peticiones recibidas mientras dure el *Rolling Update* podemos indicar cuántas réplicas añadir a un *Deployment* a la hora de actualizarlo.

Además, este parámetro puede ser un número o un porcentaje. Si nuestro cluster tiene cargas de peticiones muy variadas con respecto al tiempo, podemos amoldarnos al número de réplicas actual para incrementarlo a la hora de realizar la actualización. Por ejemplo, añadir un 10 % de réplicas antes de iniciar la actualización.

También se puede especificar el número de PODs no disponibles admitidos durante la actualización. Si configuramos 0 quiere decir que si disponemos de 5 réplicas a la hora de actualizar siempre tendremos 5 réplicas listas para servir peticiones.

3.5.3. Blue / Green

Rolling Update es la forma natural de realizar actualizaciones en Kubernetes, pero no es la única manera. ¿Qué pasa si no soportamos la coexistencia de versiones?

Blue/Green es una estrategia de actualización diseñada para reducir el tiempo de indisponibilidad, además ofrece una fácil vuelta atrás en caso de que algo fuese mal durante la actualización.

Nuestra versión actual (*Blue*) es la que está recibiendo peticiones. En el momento en el que queramos actualizar, desplegaremos la nueva versión (*Green*) pero mientras esta arranca, la versión *Blue* seguirá sirviendo las peticiones. Cuando la versión *Green* esté preparada, se redirigirán todas las peticiones a esta nueva versión.

En Kubernetes se puede realizar gracias a la abstracción de los Servicios y la forma en la que funcionan las etiquetas.

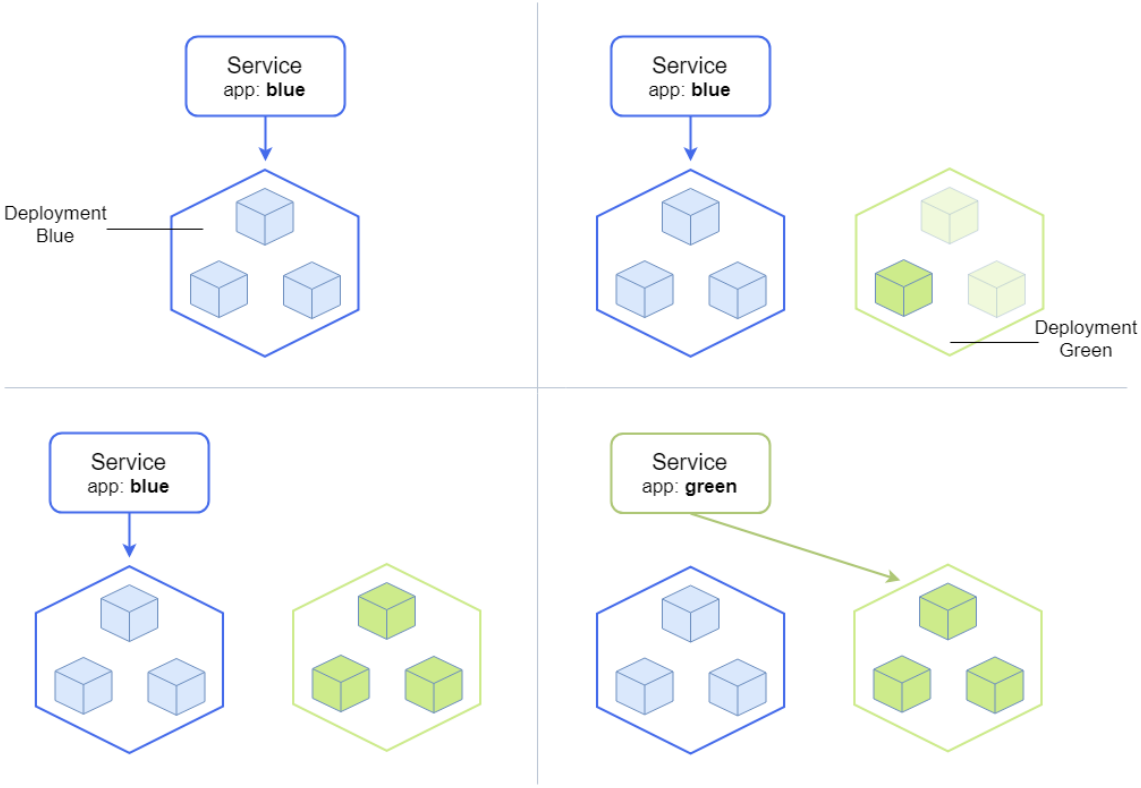

Figura 3.3: Estrategia Blue / Green

Siguiendo la figura anterior disponemos de dos *Deployment*, el primero etiquetado con "app: blue" y el segundo con "app: green". El primero lleva días funcionando y queremos sustituirlo por el segundo.

Desplegamos el segundo y esperamos a que esté listo para servir peticiones. Cuando creamos oportuno actualizamos el Servicio, cambiando el selector de PODs de "app: blue" a "app: green". A partir de ese momento las peticiones recibidas a través del servicio serán redirigidas a la nueva versión.

Si quisiésemos volver a la versión antigua sería tan rápido como volver a cambiar el selector del Servicio.

El gran inconveniente de este tipo de actualización es que se necesitan el doble de recursos ya que tendríamos las dos versiones en ejecución al mismo tiempo. Además necesitamos que la aplicación sea *stateless*, es decir, que no contenga el estado dentro de ella.

3.5.4. Canary Release

Esta estrategia de actualización se basa en redirigir un pequeño porcentaje de tráfico a la versión nueva. De esta forma podemos monitorizar el funcionamiento y en caso de no encontrar errores, actualizar la versión.

Al igual que la anterior estrategia, la aplicación necesita ser *stateless*.

Esta estrategia se puede utilizar como complemento al *Rolling Update*. También es necesaria la coexistencia de versiones, pues dos versiones distintas servirán peticiones al mismo tiempo.

Si algo fuese mal en la versión nueva, la principal ventaja de esta estrategia es que solo afectaría a un número reducido de usuarios.

La desventaja dentro del cluster Kubernetes es que no hay forma sencilla de balancear porcentajes de tráfico y se divide de forma equitativa entre todos los PODs. Por lo tanto si tuviésemos cuatro PODs con la versión actual y uno con la nueva, se redirigiría un 20 % a la nueva versión.

Cuanto mayor número de réplicas normales desplegadas, menos tráfico podríamos enviar a la versión *"Canary"*. Esto es así porque no podemos configurar Kubernetes para que una réplica reciba más o menos tráfico que otra.

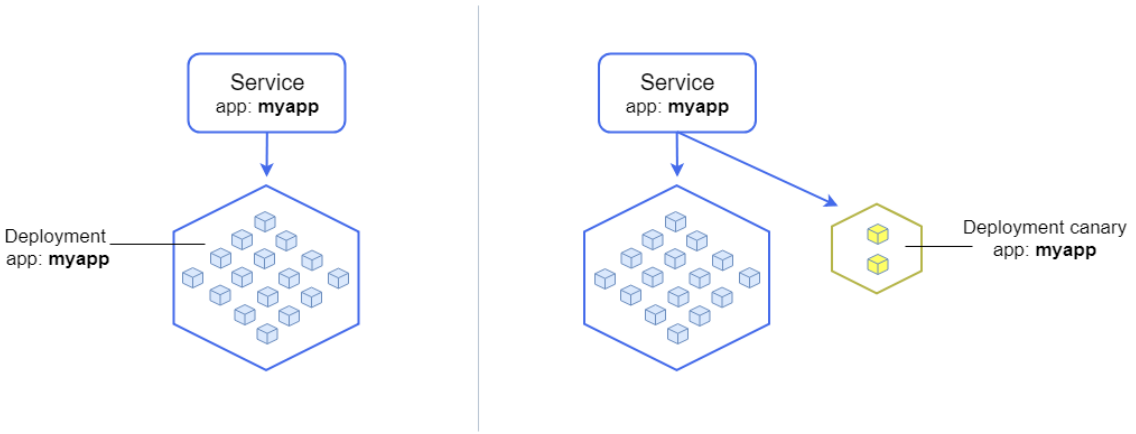

Figura 3.4: Estrategia Canary Release

La forma de implantarlo dentro del cluster Kubernetes es similar al *Blue/Green*, pero a diferencia de este, el *Deployment* nuevo llevaría la misma etiqueta que el viejo y el número de réplicas de este *Deployment* sería un número pequeño.

3.6 Distribución de las cargas de trabajo sobre los nodos

Ya hemos visto las diferentes formas de actualizar un servicio sin incurrir en indisponibilidad, pero, en un sistema distribuido puede ser muy útil la distribución de las diferentes cargas de trabajo. Por ejemplo, si hay cargas de trabajo que realizan muchas comunicaciones entre ellas o queremos distribuirlas a lo largo de diferentes nodos para garantizar la alta disponibilidad pese a caídas de servidores o incluso caídas de regiones enteras.

Por otro lado también es interesante la distribución si tenemos una arquitectura compleja, donde algunas cargas consumen CPU a ráfagas otras tienen un consumo más estable, más alto o más bajo.

Entonces, ¿cómo podemos distribuir nuestros POD a lo largo de todos nuestros nodos?

3.6.1. NodeSelector

Hay diferentes formas de realizar esta tarea en Kubernetes. La más básica es el *nodeSelector*. Su funcionamiento es sencillo. Dentro de Kubernetes podemos poner etiquetas a todos los objetos, como por ejemplo los nodos. De esta forma podemos configurar un *Deployment* seleccionando las etiquetas de los nodos donde queremos desplegar la carga de trabajo a través del *nodeSelector*.

Por ejemplo, tenemos dos tipos de servidores, uno con almacenaje de tipo SSD y otro sin él. Tenemos diferentes cargas de trabajo y una de ellas escribe mucho en disco. Si etiquetamos los servidores con ”disco: ssd” y utilizamos la clave *nodeSelector* con valor ”disco: ssd’ conseguimos que solo se despliegue en nodos con disco SSD.

3.6.2. Afinidad y Anti-afinidad

Si bien es cierto que utilizando *nodeSelector* podemos elegir el nodo donde desplegar la carga de trabajo, es posible que esta funcionalidad se nos quede corta. Imaginemos que tenemos dos cargas de trabajo que se comunican de forma asidua y queremos que se desplieguen juntas o dos cargas de trabajo que las dos utilizan en mayor medida la CPU que el resto de cargas del cluster y no queremos que se desplieguen juntas, pues podrían sobrecargar el nodo y poner en riesgo el resto de cargas.

Para estas tareas, Kubernetes dispone de la Afinidad y la Anti-Afinidad. De esta manera podremos hacer distribuciones complejas de nuestras cargas de trabajo. En estos dos artículos se explica de forma detallada el funcionamiento de la Afinidad en Kubernetes [4][5].

Estas reglas van en base a una topología. Podemos definir varios tipos de topología, por ejemplo que cada nodo de la topología sea una zona de disponibilidad de los servidores. Así podremos hacer distribuciones en base a las zonas de disponibilidad. Para los siguientes ejemplos tomaremos el caso más sencillo en el cual cada nodo de la topología es un nodo del cluster.

Al igual que *nodeSelector*, esta funcionalidad se basa en las etiquetas. Básicamente se trata de elegir la etiqueta a la cual hacer referencia indicando un operador (*In, NotIn, Exists, DoesNotExist*) e indicando si es una regla de afinidad o anti-afinidad. Por ejemplo, si a cada carga de trabajo la etiquetamos con ”nombre: nombre-de-la-carga” podemos hacer una regla de afinidad para una ”carga-1” la cual sea afín a una ”carga-2” utilizando

el operador *In*. De la misma forma podemos separar esas dos cargas utilizando la anti-afinidad o el operador "Not-In".

Esta funcionalidad también se puede utilizar para separar nuestras cargas a lo largo de todos los nodos y que no recaigan todas en el mismo. Por ejemplo si tenemos un cluster de 5 nodos y queremos dar alta disponibilidad de una carga de trabajo replicada, debemos asegurarnos de que las réplicas se repartan a lo largo de los 5 nodos y no se sitúen todas en uno. Para esto, podemos utilizar la anti-afinidad en la carga, referenciándose a sí misma.

Además, estas afinidades pueden ser estrictas o no. Imaginemos que tenemos 5 nodos y 6 réplicas de una carga de trabajo con una regla de anti-afinidad sobre sí misma, si la regla es estricta, solo podrá desplegar 5 réplicas, pues hay una réplica que no cumplirá la regla de anti-afinidad y no podrá ser desplegada. Si la regla no es estricta, habrá un nodo con dos réplicas.

3.7 Escalado Horizontal de los PODs

Si queremos que nuestro sistema sea elástico, necesitamos configurar el escalado horizontal de nuestros PODs.

Cada aplicación se comporta de forma diferente y en base a ella deberemos aplicar unas u otras reglas de escalado.

Por ejemplo tomando la aplicación de ejemplo definida en la sección anterior, ¿qué pasaría si hay un micro-servicio que está recibiendo más carga y necesita más capacidad de cómputo?

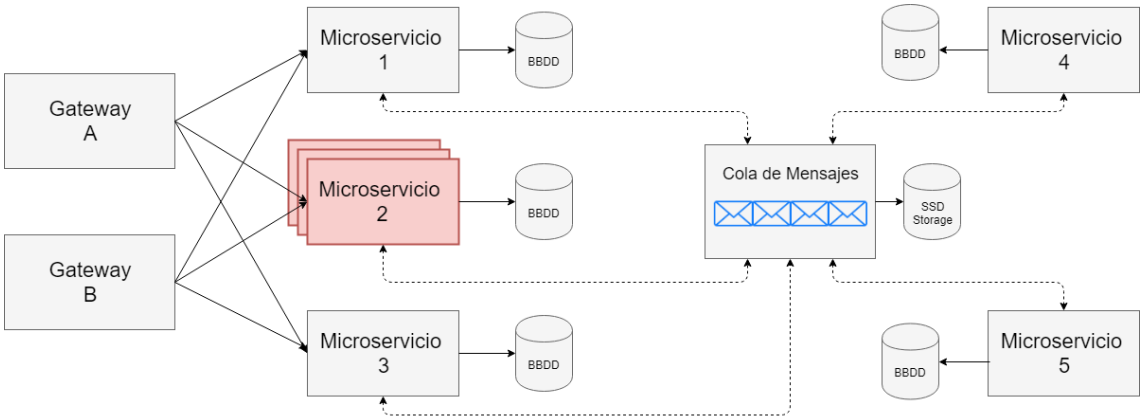

Figura 3.5: Estrategia Rolling Update

Para poder escalar horizontalmente ese micro-servicio que correspondería con un POD, podemos definir en Kubernetes una regla que gestione tanto el escalado como el desescalado.

Kubernetes ofrece de forma nativa la posibilidad de configurar el escalado horizontal de los PODs. El objeto se llama *HorizontalPodAutoescaler* y en él podemos definir el número mínimo y máximo de réplicas así como las reglas de escalado en base a medidas como consumo de CPU o consumo de RAM.

A partir de Kubernetes 1.8 con la entrada de Metrics-Server estas métricas pueden ser más complejas pudiendo ser combinadas, también han añadido nuevas como por ejemplo escalar en base al número de peticiones que se realizan.

Sin embargo si queremos elaborar un sistema de escalado más complejo, se puede escalar de forma manual a través de la línea de comandos. Lo que nos facilitaría realizar scripting que dicte cómo y cuándo queremos escalar.

De esta manera, podríamos realizar modelos de control *'MAPE-K'*, los cuales a través de la monitorización, análisis, planificación, ejecución y el conocimiento adquirido, otorgaríamos una inteligencia adicional al escalado nativo de Kubernetes.

CAPÍTULO 4
Escalado Horizontal a medida

4.1 ¿Por qué autoescalado a medida?

Es cierto que Kubernetes ofrece la posibilidad de definir reglas de escalado horizontal de los POD de forma nativa. Sin demasiado esfuerzo podemos conseguir que nuestro sistema sea escalado en base a algunas métricas soportadas por la plataforma. Pero, ¿es esto suficiente?

Dependiendo de la complejidad de nuestra aplicación y dependiendo de la carga de trabajo que se le aplique, podemos necesitar un sistema de escalado más o menos inteligente. Pudiendo controlar así de una forma más a medida el comportamiento del escalado del sistema.

Con esta motivación he diseñado un sistema de autoescalado preparado para trabajar con la API de Kubernetes y diseñado para que sea sencillo añadir estrategias de escalado y fuentes de datos diversas.

4.2 Arquitectura del escalador

La arquitectura diseñada del escalador es la siguiente y consta de 4 componentes.

- **HorizontalScaler**. Es el encargado de leer la configuración proporcionada a partir de un fichero en formato *yaml* y enviársela a los diferentes controladores de las diferentes estrategias de escalado.

- **ControllerType**. Los ControllerType implementan las diferentes estrategias de escalado. Estos componentes hablan con el cluster de Kubernetes y con los diferentes Datastores de los cuales obtienen la información necesaria para decidir cuál es el estado deseado.

- **DatastoreType**. Los DatastoreType dan acceso a las métricas que utilizan los controladores para calcular el estado deseado.

- **KubeHandler**. Es el encargado de proporcionar una interfaz la cual utilizan los controladores para interactuar con la API Kubernetes.

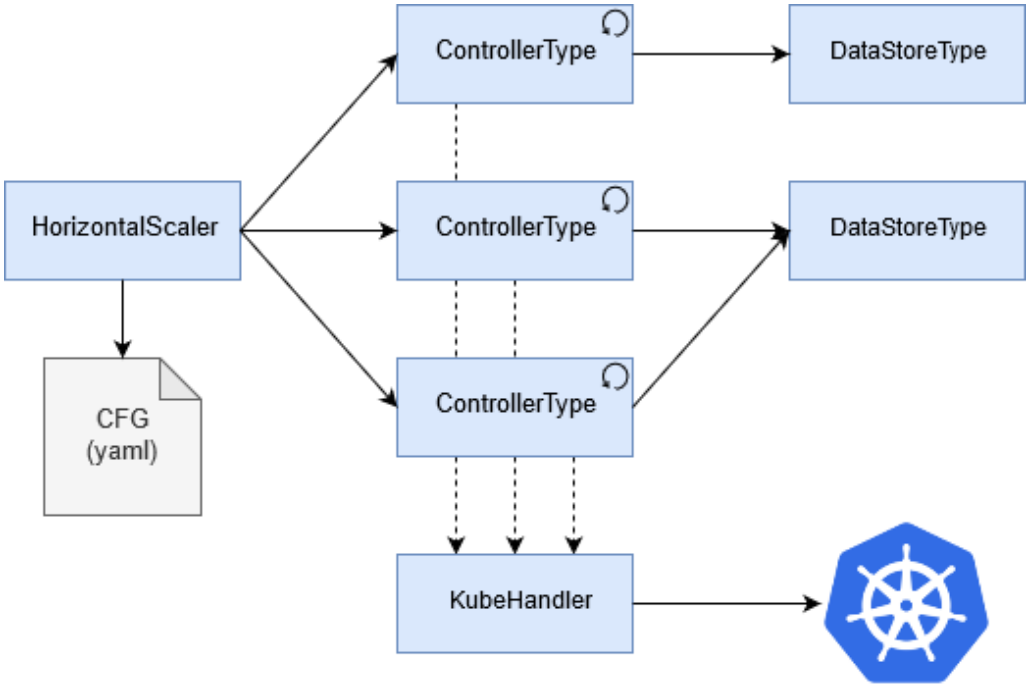

Figura 4.1: Arquitectura del escalador

En la imagen anterior podemos ver de forma visual cómo se relacionan los diferentes componentes. Todos los diferentes componentes están desarrollados en NodeJS.

4.2.1. Comunicación con el cluster

El escalador se comunica con el cluster a través de la biblioteca kubernetes-client[18] para JavaScript.

A través de esta biblioteca podemos interactuar con el clúster, pudiendo realizar la tarea que deseemos. Para realizar la autenticación y configuración podemos utilizar varios mecanismos.

En este caso el escalador se configura mediante el fichero *"kubeconfig"* el cual se proporciona a través de un *secreto*, el cual se inyecta al POD.

Kubernetes-client no es la única biblioteca Existen más bibliotecas para JavaScript, por ejemplo GoDaddy dispone de una propia[19]

4.2.2. Estrategias del escalador

Actualmente solamente hay un tipo de estrategia implementada. En esta se puede configurar los siguientes parámetros:

- **Workload afectado**.La carga de trabajo a la cual deseamos aplicar la regla de escalado.
- **Número mínimo de replicas**. Numero mínimo de PODs que debe tener la carga de trabajo.
- **Número máximo de replicas**. Numero máximo de PODs que debe tener la carga de trabajo.

- **Periodo de comprobación**. Cada cuánto tiempo deseamos que se ejecute el ciclo de comprobación del estado.
- **BoostFactor**. Cuántas réplicas se añaden/eliminan cada vez que se incumplen los límites establecidos. Gracias a este parámetro podemos hacer que escale el sistema de forma más rápida.
- **Ventana de encendido**. El tiempo que necesita nuestra aplicación para encenderse.Este parámetro se usa para no tomar dos medidas de escalado consecutivas sin que la primera no haya tenido efecto primero.
- **Ventana de apagado**. El tiempo que necesita nuestra aplicación para apagarse. De la misma forma que la ventana de encendido, el parámetro se utiliza para no tomar dos medidas de des-escalado consecutivas.
- **Configuración del datastore**.
  - **Tipo**. Actualmente solo se soporta Elasticsearch. Pero podría añadirse cualquiera.
  - **Endpoint**. Dirección donde conectar con el datastore.
  - **Usuario**. Usuario (si es necesario) para establecer la conexión.
  - **Contraseña**. Contraseña (Si es necesaria) para establecer la conexión.
- **Métrica a evaluar**. El nombre de la métrica que usaremos para tomar medidas de escalado.
- **Límite de la métrica donde ejecutar el escalado**. Cuál es el rango de funcionamiento deseado en la métrica elegida.Si está por encima se escalará la carga de trabajo y si está por debajo se tomarán medidas de desescalado.

CAPÍTULO 5
Desplegar un clúster Kubernetes

5.1 Introducción

Kubernetes ofrece muchas funcionalidades útiles para los sistemas distribuidos. Si nuestra arquitectura de aplicación se adecua, nos da replicación, actualizaciones sin tiempo de caída, escalabilidad, secciones aisladas dentro del clúster *(namespaces)*, descubrimiento de servicios, etc.

Sin embargo todas estas funcionalidades hacen que desplegar un clúster de esta envergadura sea complejo. Kubernetes dispone de muchos componentes, si tuviésemos que configurar todos ellos sería una tarea compleja.

Afortunadamente disponemos de varias herramientas que facilitan esta tarea. Dependiendo de la finalidad de nuestro clúster, podemos utilizar unas u otras.

Si quisiéramos utilizar un clúster para desarrollo, donde probar cosas en local, existe una herramienta llamada minikube con la cual podemos desplegar un clúster de forma rápida y sencilla.

Para un clúster que vaya a ser utilizado en producción deberemos realizarlo de manera diferente. Existen varias herramientas que pueden realizar este cometido. Las más destacadas y utilizadas son Kops, Kubespray y Rancher. Con estas herramientas podemos desplegar un clúster Kubernetes donde queramos.

De esta forma podemos conseguir ser multiplataforma. No obstante, si esto no nos preocupa y pensamos utilizar los recursos de los proveedores Cloud, tenemos la posibilidad de desplegar clústeres Kubernetes autogestionados.

Actualmente la mejor opción es Google Cloud Platform. Tener un clúster Kubernetes autogestionado cuesta lo mismo que desplegar el mismo tamaño de instancia en AWS.

Sin la autogestión del clúster. Además con GCP solo se pagan los nodos de cómputo, cosa que si gestionamos nuestro propio clúster, necesitamos nodos que persistan el estado del mismo mediante ETCD. Incluso si nos fijamos en las políticas de uso de CPU de las instancias, podemos observar como AWS ofrece CPU a ráfagas, es decir, compartimos la CPU, mientras que GCP te da toda la CPU de la instancia.

Hay otras plataformas Cloud que ofrecen este servicio. Como Azure y AWS desde hace muy poco, pero actualmente solo se encuentra disponible en Estados Unidos y tiene un coste más elevado que el resto de la competencia.

En este documento vamos a ver cómo desplegar un clúster mediante Rancher en AWS y en Google Cloud Platform a través del sistema autogestionado Google Kubernetes Engine.

5.2 Despliegue de un clúster Kubernetes mediante Rancher en AWS

Rancher es una compañía que trabaja con plataformas de gestión de contenedores. Mediante su plataforma se puede desplegar más clústeres aparte de Kubernetes, como por ejemplo Mesos. Aún así, con la versión 2, ha pivotado enteramente hacia Kubernetes.

Mediante esta herramienta podemos utilizar cualquier servidor de cualquier proveedor cloud, incluso servidores *'bare-metal'*, ofreciendo la posibilidad de conectarse con algunos proveedores cloud como AWS, Azure, Digital Ocean o Packet para agilizar el proceso de aprovisionado de servidores.

A pesar de tener esta opción, vamos a desplegar mediante los llamados por Rancher, servidores *'Custom'*. De esta forma podremos ser compatibles con todas las opciones.

5.2.1. Despliegue del servidor Rancher

Lo primero que debemos hacer es desplegar un servidor Rancher desde donde podremos lanzar la creación de los clúster Kubernetes que deseemos.

Desde AWS crearemos una instancia de tamaño medio y le asignaremos un rol IAM especial, el cual podemos encontrar en la página oficial de Rancher. Este rol sirve para poder acceder a información acerca de las instancias, volúmenes vinculados a estas, registro privado de imágenes Docker en AWS (ECR), entre otras cosas.

Rancher soporta varios sistemas operativos, incluso uno propio. En mi caso, he elegido Ubuntu 16.04 por ser un estándar dentro de la comunidad.

Una vez desplegada la imagen, deberemos instalar Docker. Para ello Rancher también dispone de scripts que automatizan el proceso. Además podemos incluir configuraciones como habilitar la rotación de logs dentro de Docker, para que a la larga no tengamos problemas de espacio debido a esto.

También es interesante utilizar un volumen externo para contener la base de datos de Rancher. De esta forma podemos persistir el estado en él, siendo así tolerantes a posibles fallos.

Todo este proceso se debe automatizar mediante alguna herramienta de tipo Ansible o Chef junto con Cloud Formation o el CLI que ofrece AWS. De esta forma podremos recrear el servidor en caso de necesitarlo.

5.2.2. Creación de la plantilla Kubernetes

Una vez desplegado el servidor de Rancher, podemos acceder a él para configurar la plantilla que se utilizará para crear el clúster Kubernetes.

Si queremos utilizar el clúster en un entorno de producción debemos tomar ciertas precauciones. Aquí podremos definir si queremos aislamiento del clúster, el tipo de autorización que usa el clúster, RBAC es el recomendado, comandos que ejecutará el servicio kubelet, aquí podemos indicar por ejemplo que si la máquina se queda con menos de cierta cantidad de RAM, Kubernetes tome medidas y mate algún POD para liberar espacio, como medida de prevención de última instancia.

También podemos seleccionar qué plugin de red va a utilizar el clúster, así como la versión de Kubernetes. Además existen otros plugins mantenidos por Rancher o incluso por la comunidad que facilitan implantar ciertas herramientas. Como por ejemplo, Elasticsearch, Prometheus, Kafka, ECR Credential Updater, entre otros.

Esta plantilla nos servirá para todos los clúster que se vayan a gestionar desde el servidor Rancher. Una vez creada, podremos proceder a la creación del clúster.

5.2.3. Agregar Nodos al Clúster

Para agregar nodos *'Custom'* al clúster Kubernetes, simplemente necesitamos ejecutar el agente de Rancher con el ID asociado al clúster.

Rancher dispone de 3 *'flags'* principales que podemos indicar al agente para hacer que el servidor se comporte de una manera específica dándoles el valor *true*.

- **Orchestration**. Si configuramos este *flag*, Rancher lo utilizará para desplegar la infraestructura necesaria para la creación del clúster. Como mínimo debemos tener un nodo con este *flag* para poder desplegar el clúster.
- **ETCD**. Con este *flag*, indicamos a Rancher que el nodo que vamos a agregar se va a utilizar para persistir el estado del clúster Kubernetes.
- **Compute**. Si indicamos el *flag compute*, el nodo se encargará de albergar las cargas de trabajo definidas en el clúster

Cada clúster, dependiendo del tamaño y el uso que se le vaya a dar, puede tener configuraciones distintas acerca de cuántos nodos de cada tipo debe tener.

Para un clúster de tamaño mediano-pequeño, lo normal es disponer de tres nodos ETCD que persistan el estado, además estos nodos también pueden hacer la función de orquestación que necesita Rancher.

Por otro lado los nodos que ejecutarán las cargas de trabajo están separados del estado del clúster. Gracias a esto, si alguna carga de trabajo está defectuosa y consigue corromper alguno de los nodos, no afectará al estado del clúster. Siendo así más resistentes a posibles fallos.

Al igual que con el servidor Rancher, disponemos de varios sistemas operativos soportados. En mi caso vuelvo a utilizar Ubuntu por la comodidad que me supone personalmente.

También debemos instalar Docker en la máquina, sumado a las herramientas que queramos disponer en los servidores, por ejemplo, considero útil la herramienta fail2ban, encargada de excluir IPs que intentan acceder por SSH de forma incorrecta a nuestros servidores para evitar posibles ataques de BotNets.

De la misma forma que con el servidor Rancher, debemos de automatizar el proceso con herramientas típicas de DevOps.

5.2.4. Autoescalado

Gracias a los grupos de autoescalado de AWS, podemos además de otorgar elasticidad al clúster, mejorar la robustez.

Disponemos de dos grupos de nodos. Los nodos que persisten el estado del clúster, es decir, nodos *'ETCD'* y nodos que albergan las cargas de trabajo, nodos *'Compute'*.

Por un lado podemos hacer que un clúster sea elástico mediante una herramienta oficial de Kubernetes llamada clúster-autoscaler. Esta herramienta permite vincular un grupo de autoescalado de AWS con el clúster. De estado forma cuando el clúster necesite más nodos para aumentar su carga de trabajo, podrá hacerlo. De la misma forma si considera que se infrautilizan los recursos, podrá eliminar nodos.

De esta forma no dispondremos de una predicción a la hora de escalar, pero dependiendo de la naturaleza de nuestra aplicación puede que esto sea suficiente.

En caso de no serlo, se podría añadir lógica a la monitorización, por ejemplo mediante funciones lambda que actuasen sobre el propio grupo de autoescalado. De esta forma podríamos tener monitorizaciones activas predictivas.

También podemos aumentar la robustez de nuestro clúster creando un grupo de autoescalado para los nodos *'ETCD'*. Si configuramos dicho grupo con un tamaño fijo, no conseguiremos escalabilidad, tampoco la buscábamos, pero sí conseguiremos que siempre haya un número preestablecido de réplicas.

Por ejemplo si disponemos de dos grupos de máquinas de cómputo, una arquitectura válida para ser usada dentro de una infraestructura distribuida podría ser la siguiente.

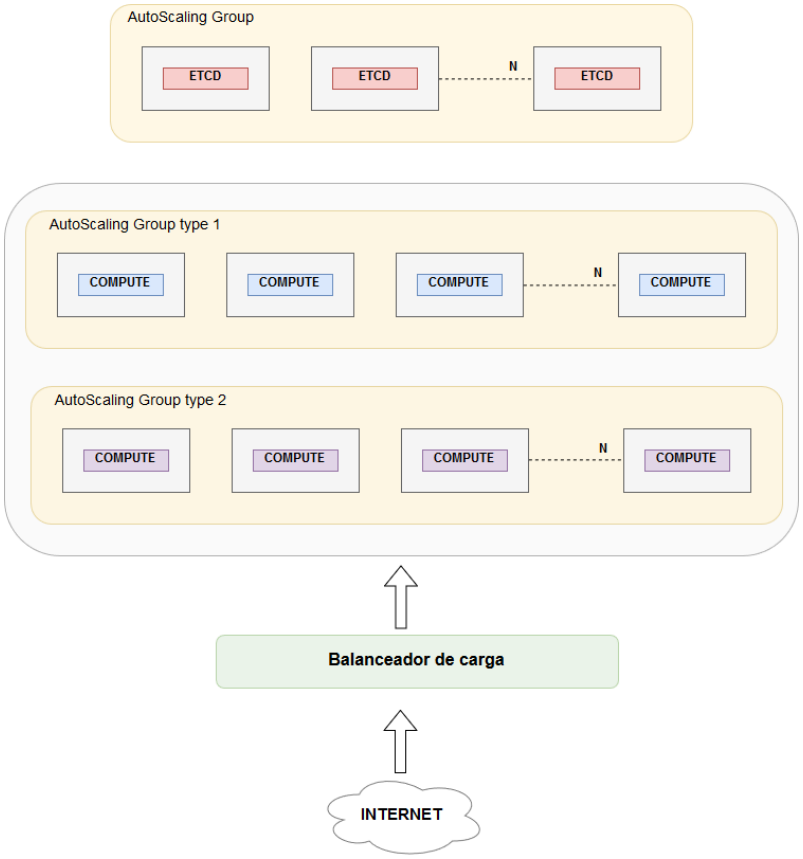

Figura 5.1: Arquitectura distribuida con grupos de autoescalado

En la imagen anterior vemos tres grupos de autoescalado. El primero corresponde a los nodos ETCD que persisten el estado del clúster. Los otros dos son los encargados de albergar las cargas de trabajo de nuestra aplicación.

Estos dos grupos son accedidos desde Internet a través de un balanceador de carga. De esta forma podemos distribuir las peticiones a lo largo de todos nuestros nodos de cómputo.

Gracias a la arquitectura de Kubernetes, podremos acceder a cualquier carga de trabajo que exponga hacia el exterior cualquier tipo de servicio independientemente de si accedemos al nodo donde realmente está desplegada.

5.2.5. Eventos de autoescalado

Cuando escalamos o desescalamos un clúster, es probable que necesitemos modificar de cierta forma estos nodos que se van a unir o a eliminar del resto.

El ejemplo más básico es configurar un nodo que se va a unir con las herramientas necesarias para poder ejercer su cometido. Para este caso podemos configurar las configuraciones de lanzamiento con el parámetro *'user-data'*. Este nos permite pasar un script de inicialización en el cual podemos incluir las instalaciones y configuraciones necesarias para que el nodo se agregue con todo lo necesario a nuestro clúster.

Por ejemplo, en este campo deberemos lanzar la instalación de Docker, configurar las rotaciones de los logs, lanzar el agente de Rancher para que el nodo se agregue al clúster e instalaciones de herramientas que consideremos como fail2ban.

Además, AWS dispone de *'Hooks'*, que son capturadores de eventos, capaces de interactuar con el resto de soluciones de AWS. De esta forma podemos capturar los eventos de *scale in* o *scale out* y lanzar, por ejemplo, funciones lambda.

Una función lambda es un trozo de código que podemos ejecutar en un *runtime* gestionado por Amazon. De esta forma nos olvidamos por completo de la infraestructura necesaria. Además solo pagamos por ella cuando la invocamos.

Rancher no sabe cuándo se ha eliminado un nodo si utilizamos directamente la herramienta de cluster-autoscaler proporcionada por Kubernetes. Simplemente reconoce que un nodo está desconectado, pero sigue guardando registro de él.

Si queremos eliminarlo por completo del sistema, debemos borrarlo. Una manera sencilla de realizar esta tarea es hacer un pequeño script que se ejecute cuando exista un evento de borrado de un nodo en el grupo de autoescalado.

Creando un evento de *scale in* en el grupo de autoescalado, que active una función lambda que recoja los datos de la instancia que va a ser borrada y realice las acciones necesarias contra la API de Rancher, solventamos el problema.

5.2.6. Infraestructura Resultante

Kubernetes ofrece la posibilidad de construir una infraestructura escalable, altamente disponible y resistente a fallos, pero para ello hay que desplegarlo de forma correcta.

Los principales elementos de mi aproximación son los siguientes:

- Grupo de auto-escalado para nodos *ETCD*. Usando los grupos de auto-escalado y los grupos objetivos o "*Target Group*" conseguimos tener un número siempre disponible de nodos *ETCD*, ya que el grupo de auto-escalado se encargará de tener el número definido de nodos encendidos.
- Grupo de autoescalado para nodos de cómputo. De la misma forma que los nodos *ETCD*, los de cómputo están englobados en un grupo de auto-escalado.
- Balanceador de carga en los nodos de cómputo. Un balanceador de carga que engloba los nodos de cómputo. De esta forma el acceso al clúster es escalable. Gracias a Kubernetes y su proxy interno, podemos acceder a cualquier servicio a través de cualquier nodo.

Los Grupos de auto-escalado están vinculados a los *Target Group* que son la definicion de los nodos pertenecientes al grupo de auto-escalado. En la definición de los *Target Group* podemos incluir un script para inicializar el nodo.

Hay algunas limitaciones a la hora de modificar los *Target Group*. Por ejemplo no se puede cambiar el script de inicio, por lo tanto si deseamos tener una infraestructura más flexible, podemos descargarnos desde un repositorio de código el script de inicio y ejecutarlo. De esta forma podremos modificar esta inicialización sin tener que modificar la infraestructura en AWS. Además de tener versionada la configuración de inicio de los servidores.

Para que toda la infraestructura sea replicable y esté definida en código podemos utilizar varias herramientas como Terraform o CloudFormation, entre otras.

Dependiendo de la naturaleza de nuestra infraestructura y el interés de soportar varios proveedores cloud, elegiremos una u otra. Como no uso más de un proveedor cloud y este es *AWS* me decanto por CloudFormation, la cual al ser una herramienta interna de *AWS* ofrece una compatibilidad total con todos los elementos de la infraestructura.

Como se ha explicado en la página 16, podemos diferenciar los nodos que pertenecen al clúster. De esta forma podemos incluir varios tipos de instancia y desplegar las cargas de trabajo que mejor se adapten a su tipo.

En la siguiente figura podemos ver un esquema de la infraestructura.

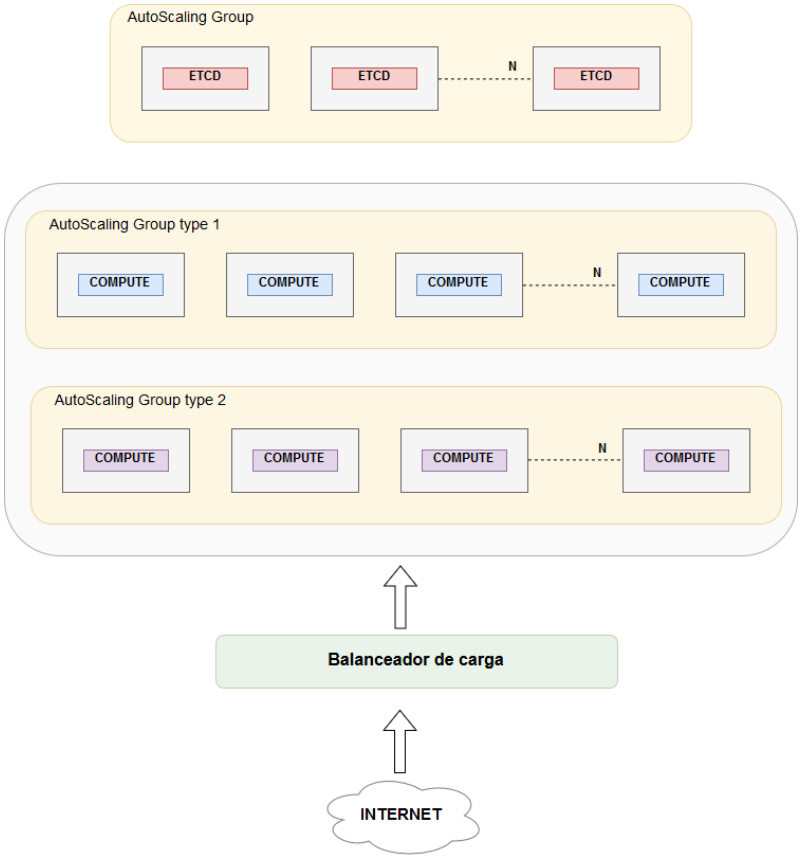

Figura 5.2: Infraestructura resultante

5.3 Despliegue de un clúster Kubernetes mediante Kops en AWS

Si bien es cierto que Rancher nos ofrece una interfaz visual para poder interactuar con el clúster, la cual puede sernos útil dependiendo del escenario y los usuarios que vayan

a usarla. Existen otras herramientas que nos ayudan a la hora de desplegar un clúster Kubernetes.

Kops [13] Kubernetes Operations es la herramienta amparada por el proyecto Kubernetes para realizar los despliegues de clústeres Kubernetes. Al igual que kubectl el CLI diseñado para interactuar con los clúster, kops es también un CLI pensado para crear, actualizar y mantener clústeres Kubernetes

Debido a la naturaleza de la herramienta hace que sea sencillo automatizar el proceso de creación. Consiguiendo así definir el proceso como código.

5.3.1. Definición del clúster

Kops ofrece la posibilidad de crear el clúster mediante un comando "kops create cluster" seguido de las opciones que deseemos configurar para adaptarlo a nuestras necesidades.

Dentro de las opciones podemos configurar la región de AWS donde se desplegará el clúster. El número de nodos Master a desplegar, el número mínimo y máximo de nodos Worker, la AMI elegida para los nodos del clúster, el plug-in de red que usará nuestro clúster, una topología privada o pública, entre otras opciones.

Sin embargo Kops también ofrece una potente funcionalidad la cual permite crear clústeres en base a ficheros yaml de configuración. Además permite usar plantillas de estos ficheros yaml y parametrizarlos en base a otros ficheros de variables.

De esta forma podemos tener una plantilla de clúster y en base a "¿Qué entorno quiero desplegar?" seleccionar un fichero de variables u otro. Siendo ahora el comando *kops create -f path-del-yaml* el comando necesario para crear el clúster.

5.3.2. Arquitectura de la infraestructura Kops en AWS

Lo primero que necesitamos para usar kops, es crear un usuario en AWS con los permisos necesarios. Después necesitaremos un bucket S3 para guardar la configuración del clúster.

Una vez satisfechos los dos primeros requisitos podemos crear el clúster.

Dependiendo de las zonas de disponibilidad elegidas para nuestros nodos, el clúster resultante en AWS tendrá diferentes elementos.

Con Kops podemos crear los LaunchConfiguration y AutoScalingGroups correspondientes, para cada Master-AvailabilityZone y nodos Worker. Además también se crearán los SecurityGroups correspondientes, así como los IAMrole necesarios.

5.3.3. Automatización del despliegue de la infraestructura y el software con Kops, HELM y Docker

Debido a la facilidad de automatizar el despliegue de un clúster Kubernetes mediante Kops, resulta lógico pensar en automatizar el proceso entero de despliegue de una aplicación. Desde el aprovisionado de la infraestructura al lanzamiento de nuestras cargas de trabajo.

Es cierto que solamente con Kops no podemos realizar esta tarea, pero si la acompañamos con otras, podemos conseguir este objetivo.

Una de las diferentes opciones de dar acceso a nuestro clúster es mediante balanceadores de carga. Si otorgamos la inteligencia suficiente a nuestra infraestructura para

realizar esta tarea, ella sola aprovisionará los recursos necesarios. Esto se puede conseguir con la herramienta kube-ingress-aws-controller [14].

De la misma forma podemos configurar de forma automática las entradas DNS de Route53 con herramientas como externalDNS [15]. Esta herramienta al igual que kubeingress-aws-controller se basan en las definiciones de nuestras cargas de trabajo para aprovisionar todo lo necesario en AWS. Por ejemplo si definimos un ingress hacia uno de nuestros servicios, el clúster será lo suficientemente inteligente como para configurar todo lo necesario para dar acceso a ese servicio.

Siguiendo la misma filosofía de añadir inteligencia a la infraestructura para que no necesite de un operario para administrarla, se pueden aprovisionar volúmenes externos mediante el aprovisionador dinámico de volúmenes EBS del cual dispone el propio kuberentes.

Teniendo la parte de la infraestructura automatiza y auto-gestionada, solo falta definir qué cargas de trabajo componen la aplicación para poder desplegarla. Para agrupar esta suma de Deployments, StatefulSets, Servicios, Ingress, etc. Podemos utilizar Helm [16].

Teniendo todo este proceso definido podemos hacer cosas interesantes como medir cuánto tiempo cuesta desplegar todo lo necesario para prestar nuestro servicio. Pudiendo tomar métricas importantes para conseguir un buen Mean Time to Restore (MTTR) y conseguir acercarnos a lo que en el libro **Accelerate** [2] llaman "high performers".

Es posible que este proceso de automatización se tenga que lanzar en contextos con una alta presión para el operario, por lo tanto es bueno simplificar todo lo posible la tarea para que pueda ser lanzada en cualquier situación desde cualquier lugar.

Para ello no es descabellado pensar en Dockerizar este proceso, añadiendole todo el contexto necesario para poder lanzar una ejecución.

De esta forma si configuramos una imagen Docker con todas las dependencias necesarias, como awscli, kubectl, kops, git, jq, etc. Dispondremos de un contenedor Docker el cual desplegará nuestra infraestructura y todas las cargas necesarias que componen nuestra aplicación.

Además junto con el soporte de plantillas de kops y helm, podemos hacer que ese contenedor despliegue clústers de desarrollo o producción indistintamente.

5.4 Desplegar un clúster Kubernetes en Google Kubernetes Engine

Kubernetes se está convirtiendo en un gestor de contenedores muy popular. Tanto es así que los principales proveedores cloud permiten usar clústeres autogestionados por los propios proveedores.

Esto quiere decir que hoy en día no es necesario desplegar un clúster Kubernetes, puedes simplemente configurarlo y utilizarlo.

Google Cloud es hoy en día la mejor opción para usar un clúster autogestionado si miramos los costes económicos.

Aquí no nos tenemos que preocupar en configurar los nodos que se agregan o eliminan del clúster, todo lo gestiona Google. Además a la hora de crear el clúster podemos configurar en qué zonas de disponibilidad lo queremos, así como el tipo de replicación.

Otra de las ventajas es no administrar los nodos *ETCD*, por lo que eliminamos costes de instancias en comparación con gestionarlos nosotros mismos.

Desplegar el clúster es tan sencillo como rellenar una serie de parámetros que van desde las zonas de disponibilidad deseadas hasta el tamaño de disco de los nodos. Así como el tipo de instancia deseado y la versión de Kubernetes deseada.

También podemos elegir si el clúster tendrá autoescalado o no, qué política de autorización se usará y qué política de red.

Una vez completada la configuración GCE creará el clúster. El proceso dura apenas unos minutos y después de esto podremos desplegar nuestras cargas de trabajo en el clúster mediante la consola que proporciona GCE o el CLI propio de Kubernetes.

Gracias a que todas las cargas de trabajo y las arquitecturas de aplicación que se definen en Kubernetes son especificaciones en ficheros, conseguimos que funcionen en cualquier proveedor Cloud y en cualquier clúster Kubernetes. Consiguiendo tener una aplicación multi-proveedor.

CAPÍTULO 6
Monitorización

6.1 Introducción

La monitorización es una parte esencial de la infraestructura. Gracias a ella vamos a percibir fallos. También nos va a dar información para tomar medidas de escalado y nos ayudará a entender qué sucede y cómo se comporta nuestro cluster.

Dependiendo de la aplicación tendremos más o menos métricas de interés. Aquí veremos cómo monitorizar y exponer métricas en un cluster Kubernetes.

El propio cluster Kubernetes expone métricas del cluster. Originalmente la herramienta usada se llamaba Heapster, pero desde la versión 1.8 Kubernetes cuenta con Metrics-Server, un agregador de datos del uso de recursos.

Gracias a Metrics-Server Kubernetes puede dar información del uso de los recursos a través del CLI kubectl. Sin embargo existen otras formas de exponer información interna del cluster si no somos compatibles con Metrics-Server.

Por ejemplo kube-state-metrics expone los datos obtenidos por la API de Kubernetes para que Prometheus u otro recolector de información compatible pueda consumirlos.

6.2 Prometheus, Grafana

Para poder disponer de paneles con la información del cluster, así como de alarmas que nos den información precisa sobre el estado del cluster, necesitamos herramientas externas. La combinación Prometheus-Grafana es la más utilizada en la actualidad.

Estas dos herramientas son OpenSource y disponen de una amplia comunidad que las va mejorando y adaptando a las nuevas necesidades.

6.2.1. Prometheus

Prometheus es una herramienta OpenSource con la cual podemos recolectar métricas y almacenarlas. Dispone de múltiples integraciones y es una de las herramientas más utilizadas por la comunidad para este propósito.

Cuenta con una base de datos integrada y un sistema propio de consultas para extraer la información.

Para recolectar información utiliza un exportador de datos llamado node-exporter. Esta herramienta debe desplegarse dentro del cluster Kubernetes.

Una de las ventajas destacadas de Prometheus es su lenguaje de consultas, el cual es bastante flexible. También dispone de un modelo de pull para los recolectores de métricas.

También dispone de un servicio descubridor para los objetivos, esto facilita mucho la integración con herramientas como Kubernetes, la cual dispone de elementos que se crean y se destruyen como los POD.

Su arquitectura está diseñada para ser altamente escalable. Lo cual es idóneo para entornos donde la probabilidad de escalado es alta o entornos ya escalados que necesiten configuraciones eficientes.

6.2.2. Grafana

Grafana es una herramienta OpenSource que nos da la posibilidad de mostrar gráficas de datos recolectados a partir de Prometheus, ElasticSearch, InfluxDB y OpenTSDB entre otros.

Podemos definir distintos paneles con diferentes métricas. Dentro de cada panel podemos añadir y editar diferentes tipos de gráficos como tablas, mapas de calor o típicos gráficos.

En la siguiente figura podemos apreciar una sección de un panel típico.

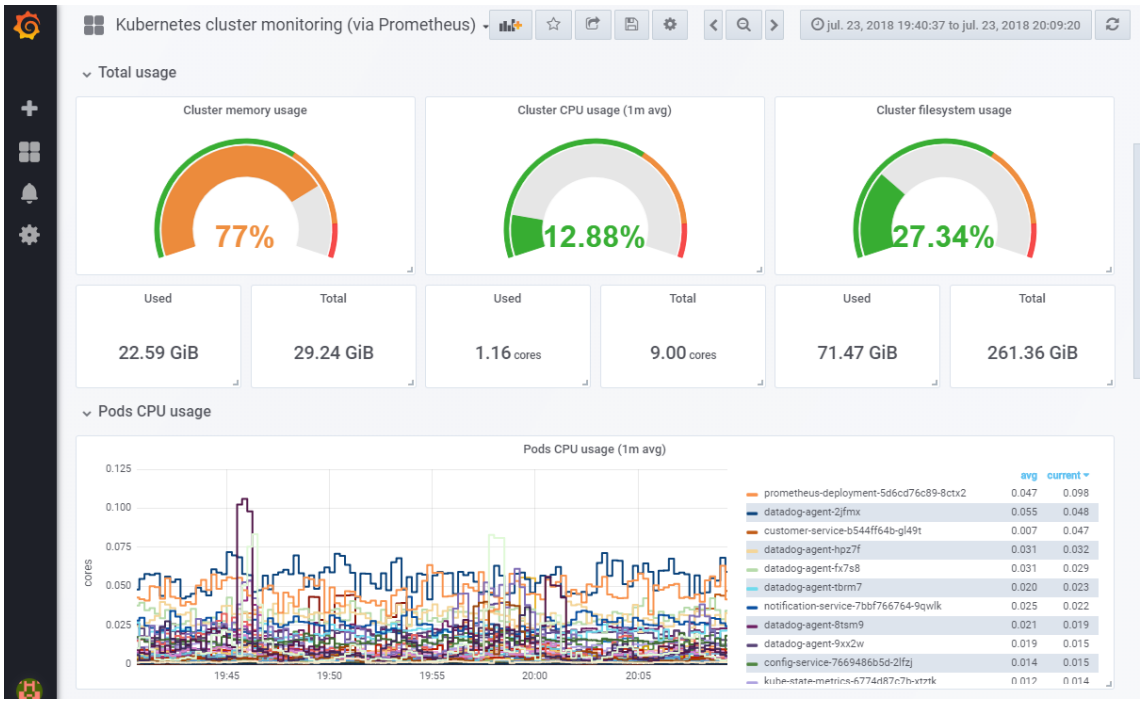

Figura 6.1: Ejemplo de panel en Grafana

En él podemos ver por ejemplo el consumo total de CPU, RAM y capacidad de almacenaje. También podemos ver el consumo de CPU de cada POD. De la misma forma el panel contiene el uso de memoria de los POD así como el uso de red.

Además podemos modificar la forma en la que se muestran los datos, de forma interactiva. Si observamos la figura anterior, si pulsásemos en la leyenda algún POD o varios de ellos, solo mostraría los pulsados. También podemos seleccionar el periodo de tiempo que deseemos sobre el panel para hacer foco en él.

Detrás de Grafana hay una comunidad muy activa que comparte diferentes paneles y configuraciones lo cual permite aprovechar mejor la herramienta.

Cada panel se puede definir, exportar e importar en formato JSON. Esta herramienta está muy bien adaptada a plataformas como Docker, y se pueden configurar y desplegar de forma sencilla. Contiene un fichero de inicialización donde configurar las diferentes opciones. Sin embargo, todas las opciones que existen en ese fichero de inicialización también se pueden configurar a través de variables de entorno.

Toda la configuración se puede trasladar a código, no obstante, si queremos que persista en caso de fallos en los servidores, podemos aprovechar las funcionalidades que ofrece Kubernetes y vincularlo a un volumen externo, de esta forma nunca perderemos el estado de nuestro Grafana.

Alarmas

Las alarmas son un recurso muy útil a la hora de monitorizar un sistema. Si ofrecemos un servicio a terceros y queremos saber la calidad del mismo, debemos darnos cuenta de cuándo no lo estamos haciendo.

Con las alarmas podemos mandar notificaciones a nuestro correo, móvil o chats internos a nuestra organización.

Con Grafana, desde la versión 5, podemos definir alarmas sobre nuestras métricas. Además de integrar estas alarmas con herramientas de mensajería como Slack, un chat muy utilizado en el ámbito empresarial.

Además junto con los avisos, se pueden enviar gráficas de las métricas que están incumpliendo las reglas de las alarmas. Para ello basta con configurar algún almacén de imágenes como pueda ser S3 en AWS o incluso en un registro local.

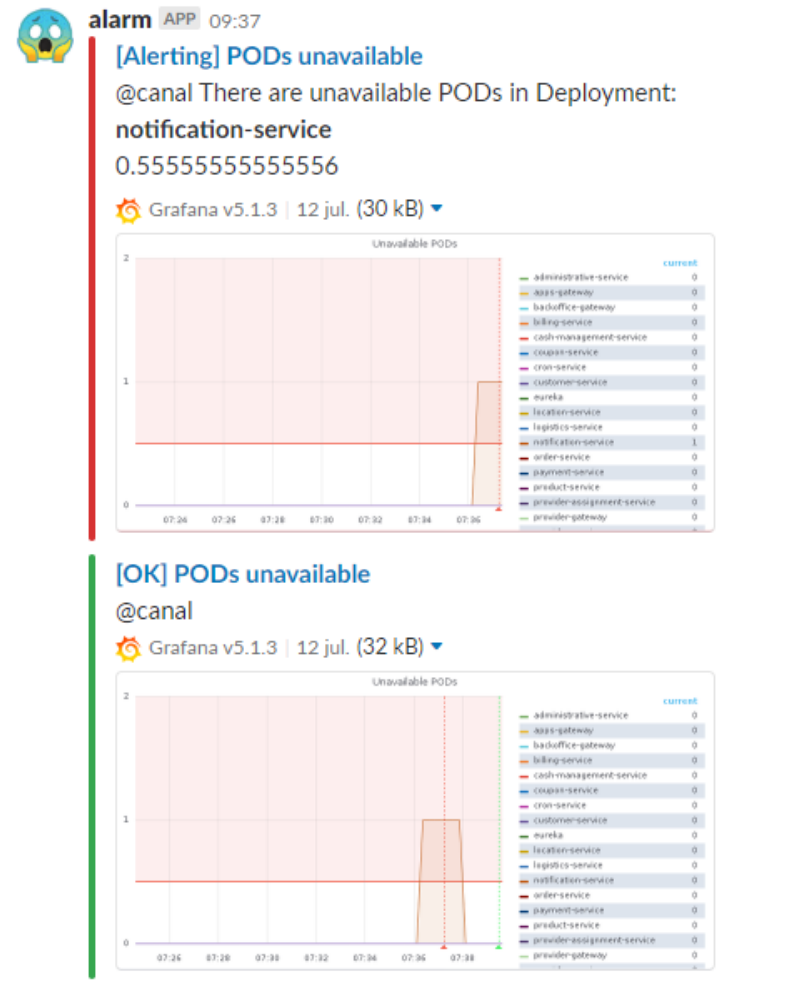

Figura 6.2: Ejemplo de notificación

La figura anterior es un ejemplo de una notificación de alarma en un *Deployment* de PODs no disponibles, en este caso de un *Deployment* llamado notification-service. También podemos ver como el cluster ha podido reiniciar el POD y la métrica vuelve a su estado deseado.

Las alarmas que en mi caso he decidido definir son:

- **Uso CPU por nodo**. El uso de la CPU siempre es importante, una notificación que nos indique que el cluster está cerca del máximo consumo posible de CPU puede ser interesante.
- **Uso de RAM por nodo**. Aunque hay opciones que hacen que los nodos de un cluster Kubernetes no llegue al 100 % de uso de RAM, es interesante controlar cuando algún Nodo está cerca de su límite.
- **Uso del sistema de ficheros por nodo**. El uso del sistema de ficheros es una métrica importante. Si se saturase puede hacer fallar las cargas de trabajo que se ejecutan en el nodo. Con una simple alarma estamos al tanto de esta métrica.
- **PODs no disponibles**. Siempre es interesante conocer cuándo tenemos algún POD no disponible dentro de nuestro cluster y si este ha podido regenerarse.
- **PODs deseados menos disponibles por Deployment**. Esta métrica es más interesante que la anterior, ya que en circunstancias como actualizaciones podemos tener

PODs no disponibles, pero esto no quiere decir que el número de PODs deseado no estén en funcionamiento.

Como hemos visto, las actualizaciones en Kubernetes generan PODs adicionales que arrancan mientras los viejos sirven tráfico. Con esta métrica nos aseguramos de que el número de réplicas esperado estén en funcionamiento.

6.3 Estado de versiones desplegadas en los entornos

En el contexto de integración continua donde estamos, no hay un versionado explícito. Eventualmente se pueden usar tags, pero lo común es vincular la imagen docker al hash de git que ha generado dicha imagen.

Por lo tanto es interesante tener una visión de qué hay desplegado, sobre todo en entornos de desarrollo y preproducción donde se puede desplegar de distintas ramas. Con una visión de qué hay desplegado los desarrolladores van a poder interactuar de mejor manera con los entornos de pruebas.

Para ello he desarrollado una pequeña aplicación web en *Vue*, un framework de JavaScript para hacer interfaces de usuario reactivas.

6.3.1. Qué debería estar desplegado

El estado de la web se construye a partir de qué debería estar desplegado. Para recolectar estos datos se construye un JSON a partir de cada despliegue de micro-servicio realizado en Jenkins.

Los datos guardados de la imagen son: la fecha, el usuario que ha realizado el despliegue y el hash de git que identifica el último *commit* del repositorio a partir del cual se ha generado la imagen.

Esta información se recoge mediante un trabajo definido en Jenkins el cual se llama en los trabajos de despliegue de los micro-servicios.

Este JSON con la información generada en Jenkins, se publica a través de un endpoint, el cual consulta la aplicación web.

Además el trabajo encargado de construir el JSON es capaz de recibir micro-servicios nuevos y generar la estructura necesaria para que la web lo muestre. Así, sin tener que modificar la web, es capaz de monitorizar nuevos micro-servicios sin ningún tipo de cambio.

6.3.2. Qué está realmente desplegado

Para mejorar esta vista de monitorización, he implementado una comprobación la cual consulta al cluster correspondiente para así determinar qué versión está desplegada en el cluster y si tiene todos los POD funcionando.

Para esto se consulta periódicamente al cluster a través de un trabajo programado en Jenkins. Dicho trabajo publica en otro endpoint la información de interés perteneciente a cada cluster.

De esta manera la web puede comprobar no solo qué está desplegado sino si el despliegue funciona correctamente.

También sirve para controlar contenedores que arrancan más lentamente o despliegues con muchos contenedores a actualizar. Ya que desde que el trabajo de Jenkins finali-

za el despliegue hasta que todos los contenedores de la versión correspondiente están en funcionamiento, pueden pasar varios minutos.

De esta forma el desarrollador tiene una visión clara del estado del cluster.

6.3.3. Qué se muestra en la web y de qué manera

La web es una tabla con un buscador que permite filtrar por cualquier campo de los elementos mostrados.

La tabla está dividida en cuatro columnas. Los micro-servicios y los tres entornos: desarrollo, pre-producción y producción.

En cada entorno se puede ver el ID del último *commit* de git el cual ha generado la imagen, el usuario que ha realizado el despliegue y la fecha de este por cada micro-servicio.

Search MicroService

MicroService	DEV	PRE	PRO
provider	643217cb8e1e4d12a061820362ec10f078e26664 ignas 07/30/18-11:29:22	d99f20a3f0971e6265c72296bbcbd21c7bf7e035 ignas 07/30/18-16:41:06	-
administrative-service	f960094d9599cb86f247e892a6d61e125123e753 gauxachs 02/08/18-14:40:23	f960094d9599cb86f247e892a6d61e125123e753 gauxachs 02/08/18-14:43:05	f960094d9599cb86f247e892a6d61e125123e753 gauxachs 02/08/18-14:46:39
coupon-service	7eeaedea35d61a5b6b3dd3ecfccf1bbbab546cb0 hector 02/08/18-07:49:56	7eeaedea35d61a5b6b3dd3ecfccf1bbbab546cb0 hector 02/08/18-07:51:07	7eeaedea35d61a5b6b3dd3ecfccf1bbbab546cb0 hector 02/08/18-08:00:13
backoffice	b6afd19004245d1027c1fd748e545b677283ec5c mario 01/08/18-16:31:39	6a7b94fe180307b184d2d5a1e09d88f42f8eb304 jarodllo 02/08/18-14:28:34	6a7b94fe180307b184d2d5a1e09d88f42f8eb304 jarodllo 02/08/18-14:36:06

Figura 6.3: Web en Vue para monitorizar las versiones desplegadas

Los ID del *commit* de git se utilizan para referenciar a la imagen. Si estos ID coinciden entre entornos, se pintan del mismo color. Este color varía dependiendo de los entornos donde coinciden. No es lo mismo que coincida entre los tres entornos o solo en desarrollo y pre-producción o pre-producción y producción.

No obstante, estos despliegues o estas actualizaciones de los despliegues ya realizados a través de Jenkins no son instantáneos.

Dependiendo del tiempo empleado por los contenedores para llegar a un estado el cual permita recibir carga y del número de réplicas de cada micro-servicio, el tiempo necesitado para completar la actualización varía.

Es por esto que a través de la consulta al endpoint que publica información relativa a los clústeres, se colorea de rojo los micro-servicios que no tienen todos los POD desplegados y funcionando en la versión deseada.

De esta forma podemos saber de forma muy visual el estado de los micro-servicios y sus actualizaciones, así como de las versiones desplegadas de los mismos.

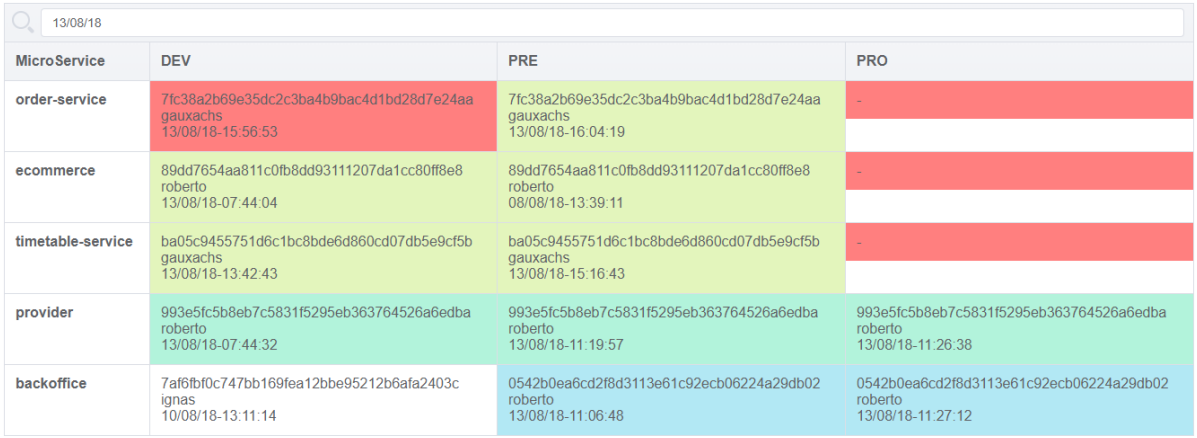

Figura 6.4: Ejemplo de una actualización en proceso.

De la misma forma si algún micro-servicio tiene un problema y no dispone de los POD funcionando, la web mostrará en rojo dicho micro-servicio.

Por otro lado, si todavía no se ha hecho ningún despliegue en el entorno correspondiente, la web muestra '-'.

CAPÍTULO 7
Integración Continua

7.1 Introducción

La integración continua es una metodología de desarrollo software la cual se basa en hacer entregas pequeñas y continuas de desarrollo sobre una solución software.

Esta metodología se centra en el desarollo ágil y pretende reducir al máximo el tiempo transcurrido entre que se descubre un fallo o se propone una mejora en el código y la entrega del mismo a un cliente.

Kubernetes da facilidades a la integración continua, gracias a los *Rolling Updates* podemos hacer pequeñas entregas de código sin repercutir en la disponibilidad del servicio. Además debido al historial de ReplicaSets que guarda podemos volver a un estado anterior en muy poco tiempo.

La integración continua va ligada al automatismo de los procesos, pues así podemos agilizar más el proceso. Hay varias herramientas que nos van a permitir hacerlo. Hoy en día la más conocida y usada es Jenkins.

7.2 Jenkins

Jenkins es un software open source de integración continua. Básicamente Jenkins es un repositorio de trabajos que pueden ser lanzados de forma manual, a través de temporizadores o *'hooks'* externos como un push a un repositorio de código.

Una de las grandes ventajas que ofrece Jenkins es la definición de estos trabajos como un script en groovy. De esta forma podemos tener los trabajos definidos y versionados en un repositorio de código. Además Jenkins puede descargarse directamente del repositorio el script del trabajo que va a realizar.

Este script se encarga de definir etapas del trabajo, las cuales se pueden configurar para ser ejecutadas en paralelo o de forma secuencial.

7.2.1. Integración continua en Jenkins y Kubernetes

A través de Jenkins podemos definir un trabajo el cual esté pendiente de un push a determinada rama de un repositorio git. Este push puede lanzar la ejecución del trabajo.

Supongamos que tenemos los tres típicos entornos de desarrollo, pre-producción y producción.

El trabajo puede descargarse el repositorio donde se ha realizado el cambio y realizar las tareas necesarias para compilarlo, si es necesario. Una vez compilado el código, se pueden pasar test de integración y test unitarios.

Si todo ha ido correctamente, se procede a crear la imagen Docker que contendrá nuestro servicio.

Una vez creada la imagen de forma satisfactoria se procede a entregar el código al entorno de desarrollo. Para ello basta con un simple comando a través de la línea de comandos de Kubernetes.

Cuando la imagen se despliega en el cluster, se etiqueta la imagen como *'dev'*, lo cual indica que está en el entorno de desarrollo. Además esta imagen se etiqueta con el hash de git referente al commit el cual ha disparado la creación de la imagen.

Dependiendo del nivel de automatismo que tengamos en nuestra infraestructura, podemos plantearnos hacer automático el paso a pre-producción, el siguiente nivel de entorno.

Por ejemplo, si tenemos testing automatizado con herramientas como RobotFramework, Selenium o Postman, podemos definir una batería de pruebas que debe de pasar antes de llegar al siguiente entorno. Una vez completadas de forma satisfactoria, se puede desplegar la imagen en el entorno de pre-producción. Otra vez a través de la línea de comandos ofrecida por Kubernetes.

Si no tenemos ese nivel de automatización podemos definir un trabajo en Jenkins que se lance de forma manual, el cual promociona la imagen de desarrollo etiquetándola como pre-producción y desplegándola en el cluster correspondiente.

El último paso al entorno de producción se suele hacer siempre de forma manual, para que intervenga el desarrollador o el validador de esa versión. El procedimiento es el mismo que el descrito con el paso manual entre desarrollo y pre-producción.

Se etiqueta la imagen como *'pro'* y se despliega en el cluster de producción.

Durante todo este proceso siempre hay una posibilidad de rollback, es decir, de vuelta atrás. Kubernetes además ofrece esa posibilidad de forma nativa.

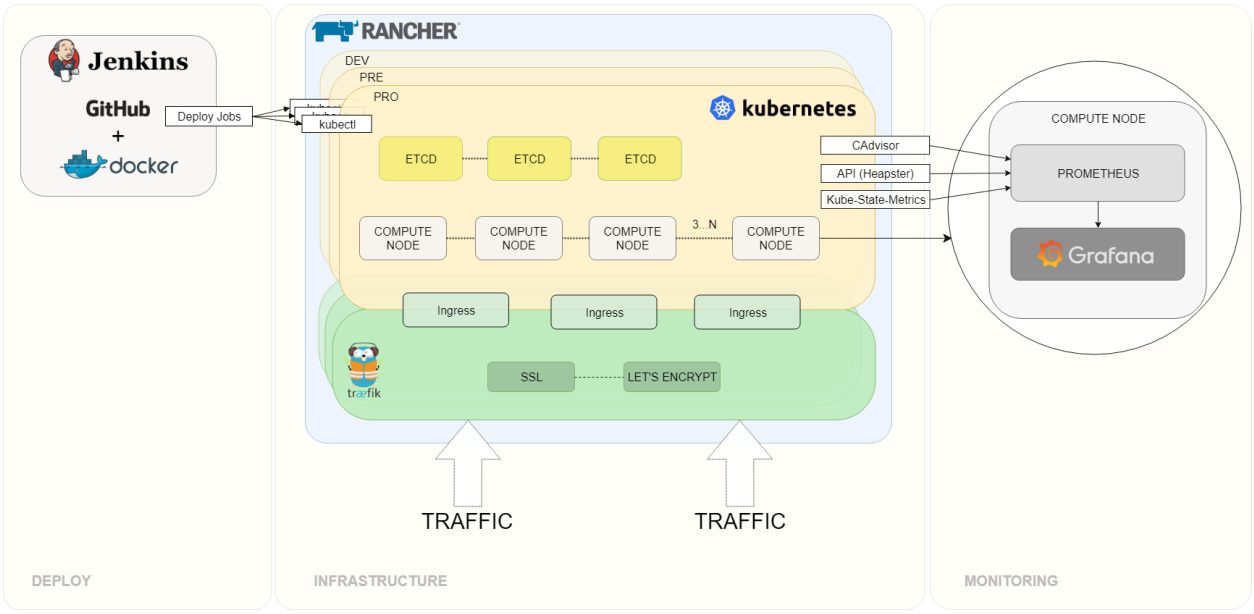

Figura 7.1: Infraestructura, vision global

CAPÍTULO 8
Pruebas de Carga

8.1 Introducción

Las pruebas de carga se han realizado en una aplicación similar a la explicada al principio de la memoria.

El punto de vista de los gráficos que se van a mostrar siempre es el del gateway. La aplicación va a recibir carga a través de la API y esta va a ser redirigida hasta el gateway, el cual tendrá que comunicarse con un micro-servicio que consultará la base de datos y devolverá el resultado.

Para realizar la prueba de carga se ha utilizado la herramienta AB (Apache Bench) con distintos grados de concurrencia, cada grado de concurrencia es una simulación de un cliente emitiendo solicitudes.

En algunas pruebas también se ha utilizado la herramienta AB desde diferentes dispositivos, para conseguir más tráfico.

La medición está realizada por *micrometer*, una herramienta de recolección de datos para aplicaciones *spring boot*.

Gracias a herramientas como *Prometheus* de la cual se habla en el capítulo de monitorización, es sencillo recolectar las métricas que sirve *micrometer* en un entorno distribuido como Kubernetes.

Además mediante Grafana podemos ver estas métricas de forma bastante interactiva.

Dentro de los gráficos observaremos en algunas ocasiones varias llamadas a distintos endpoint. Esto se debe a que durante la prueba la aplicación recibía alguna llamada no generada por AB, no obstante los gráficos están filtrados para ver solo el endpoint de interés "customer-service".

El estudio de rendimiento siguiente no va dirigido a saber hasta donde puede escalar el sistema, sino a cómo se comporta el sistema cuando necesita escalar.

8.2 Plan de pruebas y objetivos

El objetivo principal de una infraestructura escalable y elástica es poder absorber diferentes intensidades en las cargas de trabajo adaptándose a estas.

En estas pruebas se pretende probar el funcionamiento de la infraestructura, pretendiendo conseguir un tiempo de respuesta de los microservicios estable independientemente de la intensidad de trabajo que reciba.

Para poder explorar los comportamientos en los diferentes estados posibles de la infraestructura y debido al gasto que supondría disponer de una gran capacidad de cómputo se han limitado los recursos de la infraestructura.

Además se han configurado los servicios para que solo puedan escalar horizontalmente hasta 5.

Métricas

Las métricas que se han usado para medir el comportamiento de la infraestructura son las siguientes:

- **Peticiones por segundo**. Las peticiones que recibe el microservicio
- **Tiempo medio de respuesta**. Es el tiempo medio que tarda el servicio en responder
- **Tiempo medio de respuesta en percentil**. El percentil 95, 90, 75 y 50 del tiempo medio de respuesta. Usado para detectar anomalías en el comportamiento.

Ejecución de las pruebas

- **Fase inicial**. La fase inicial de las pruebas es obtener métricas que nos sirvan de punto de referencia. Para ello se han lanzado diferentes intensidades de trabajo a los servicios con el escalado horizontal deshabilitado. Obteniendo así el comportamiento de una infraestructura no escalable.
- **Escenarios**. Se plantean varios escenarios, con distintos grados de concurrencia, 5, 10 y 40. Además de 5x40 el cual debido a la limitación de los equipos desde donde se lanzan las pruebas sirve para escalar horizontalmente la carga que se suministra al clúster.

  Estas pruebas están ejecutadas con y sin el escalado horizontal del clúster activado.

8.3 Pruebas de carga sin autoescalado

En esta sección vamos a ver las pruebas de carga realizadas sin activar el autoescalado.

Los gráficos principales que vamos a observar son el número de peticiones por segundo (*Request per second*), tiempo medio de respuesta (*Mean response time*) y los tiempos de respuesta en porcentaje.

8.3.1. Concurrencia 5 sin autoescalado

Con un grado concurrencia igual a 5 en la herramienta ab, se ha conseguido la siguiente carga.

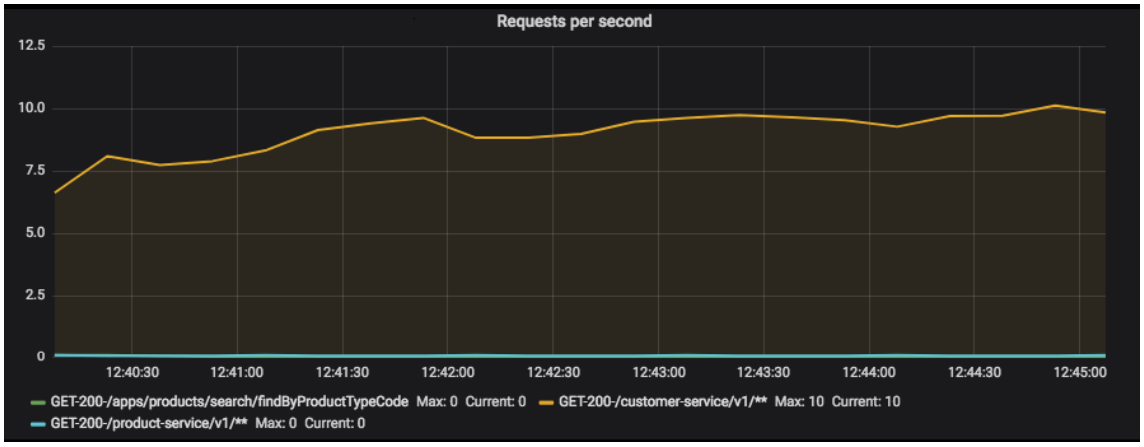

Figura 8.1: Peticiones por segundo, concurrencia 5 sin autoescalado

La resuesta media a la peticion ha sido la siguiente.

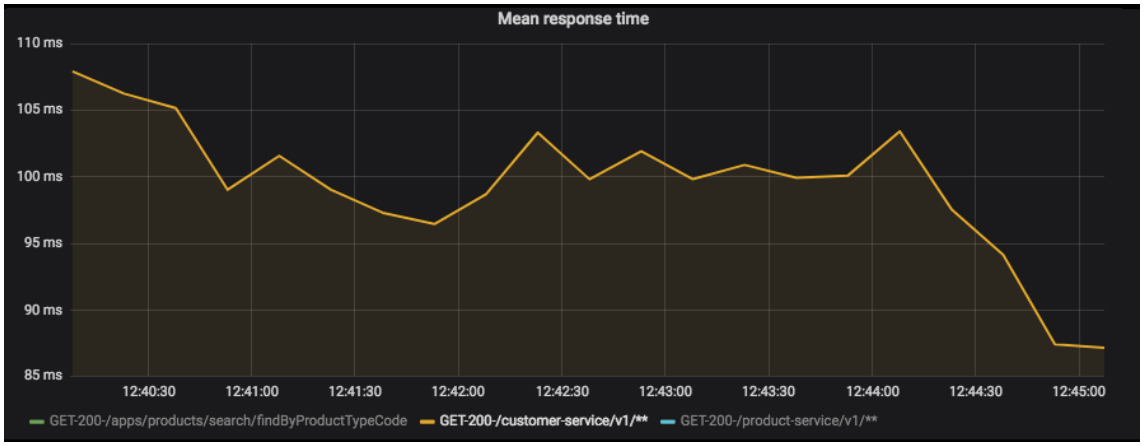

Figura 8.2: Respuesta media, concurrencia 5 sin autoescalado

Y en percentiles.

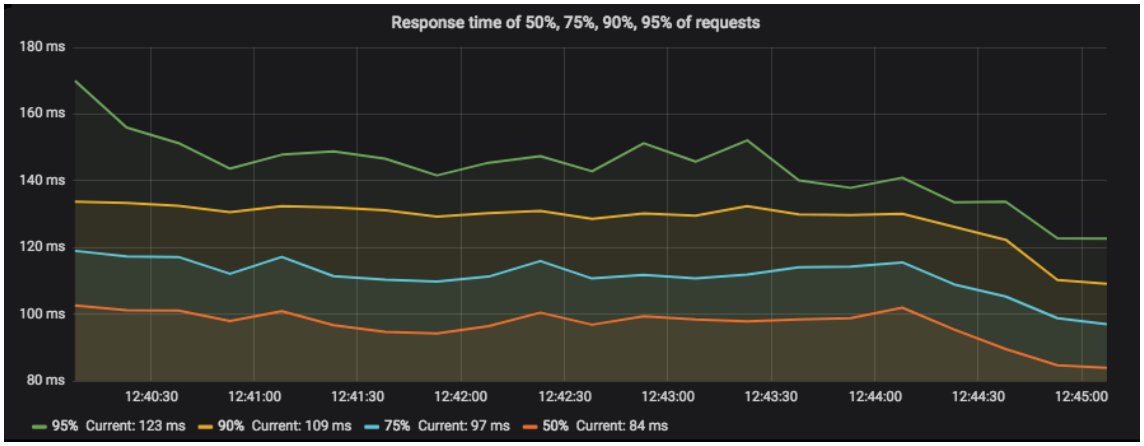

Figura 8.3: Respuesta en percentiles, concurrencia 5 sin autoescalado

Como podemos observar el tiempo de respuesta está entre 110ms y 85ms. Este dato lo podemos tomar como punto de referencia para los siguientes escenarios.

8.3.2. Concurrencia 10 sin autoescalado

Si aumentamos el grado de concurrencia a 10, observamos como el tiempo de respuesta medio aumenta unos 50ms.

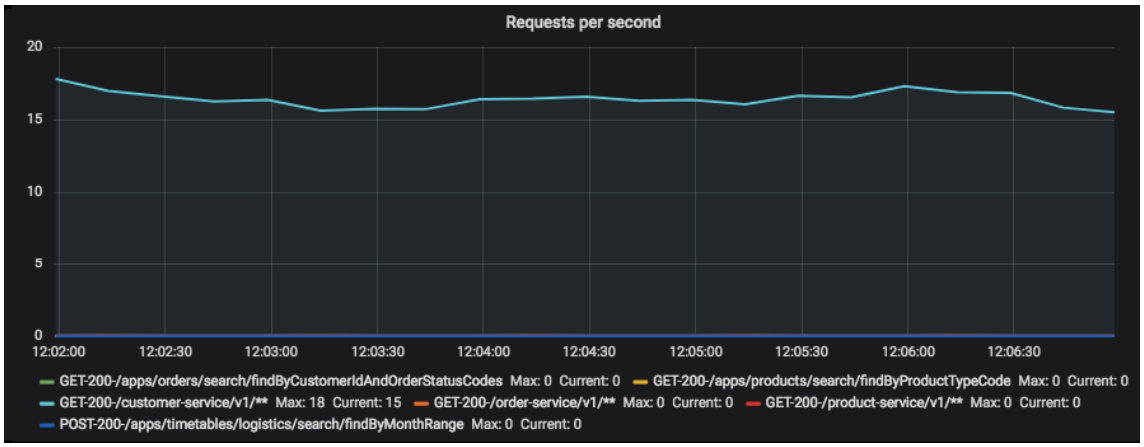

Figura 8.4: Peticiones por segundo, concurrencia 10 sin autoescalado

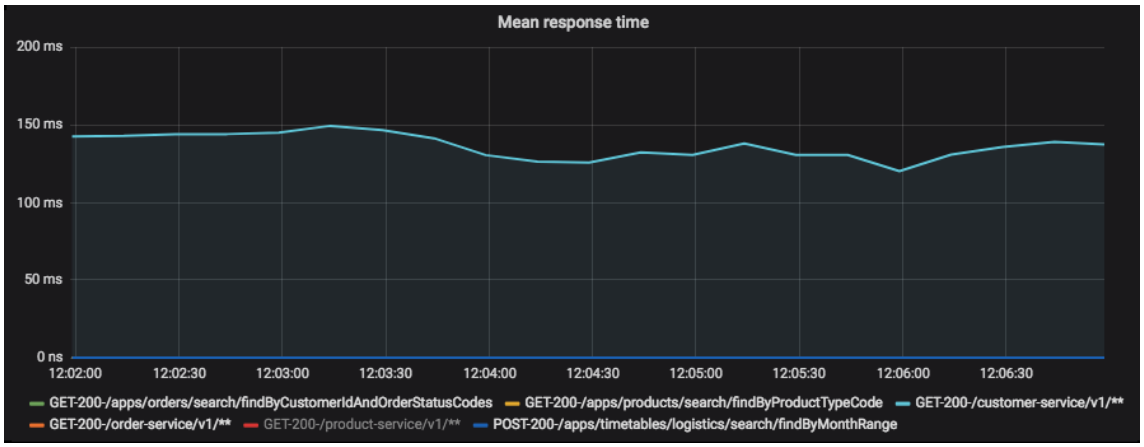

Figura 8.5: Respuesta media, concurrencia 10 sin autoescalado

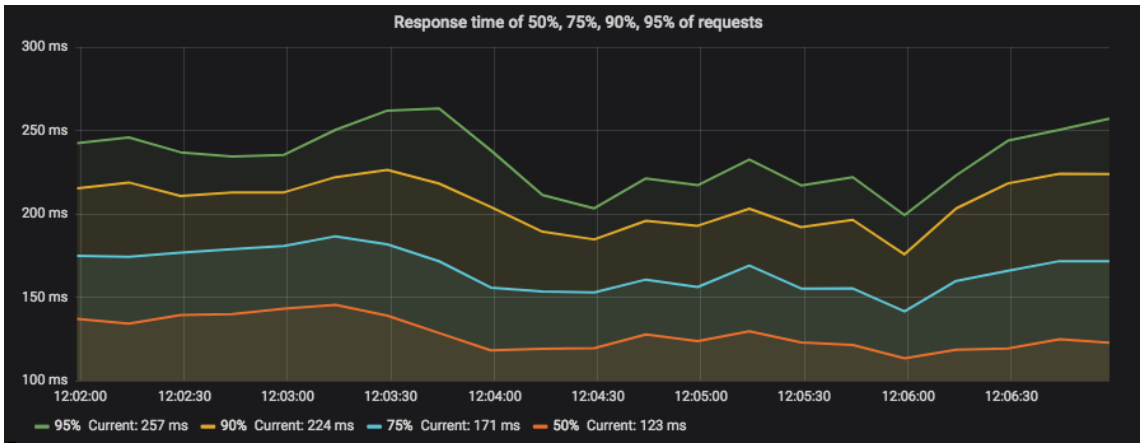

Figura 8.6: Respuesta en percentiles, concurrencia 10 sin autoescalado

8.3.3. Concurrencia 40 sin autoescalado

Al aumentar el grado de concurrencia a 40, se aprecia un incremento considerable en el tiempo de respuesta, en torno a 400ms. La ampliación empieza a notar la carga de trabajo.

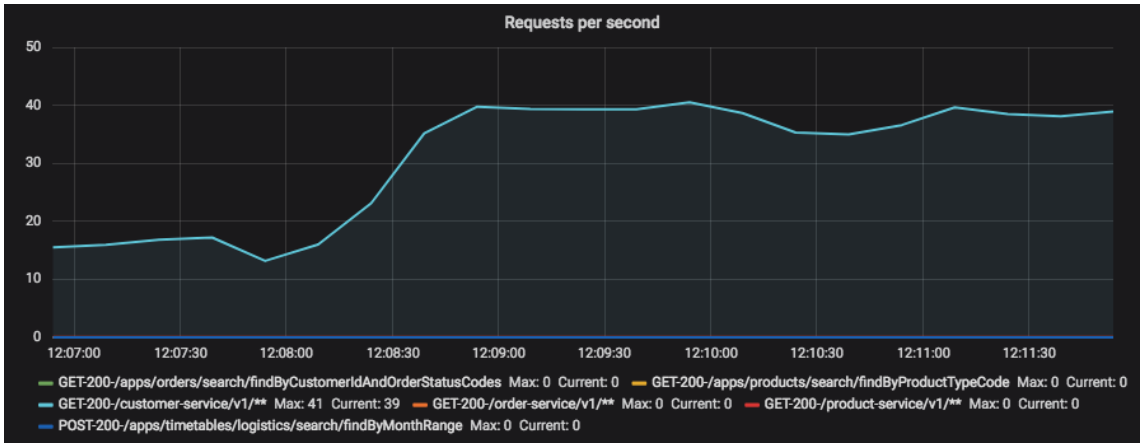

Figura 8.7: Peticiones por segundo, concurrencia 40 sin autoescalado

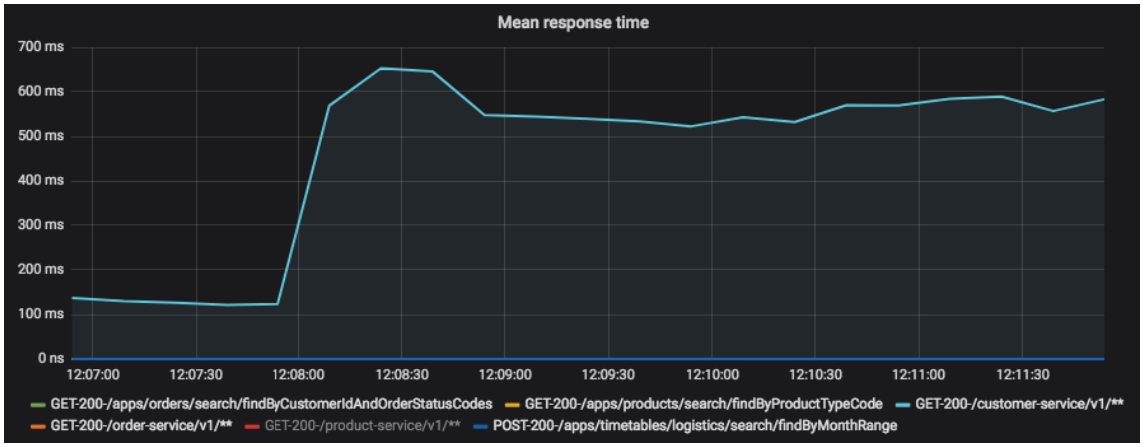

Figura 8.8: Respuesta media, concurrencia 40 sin autoescalado

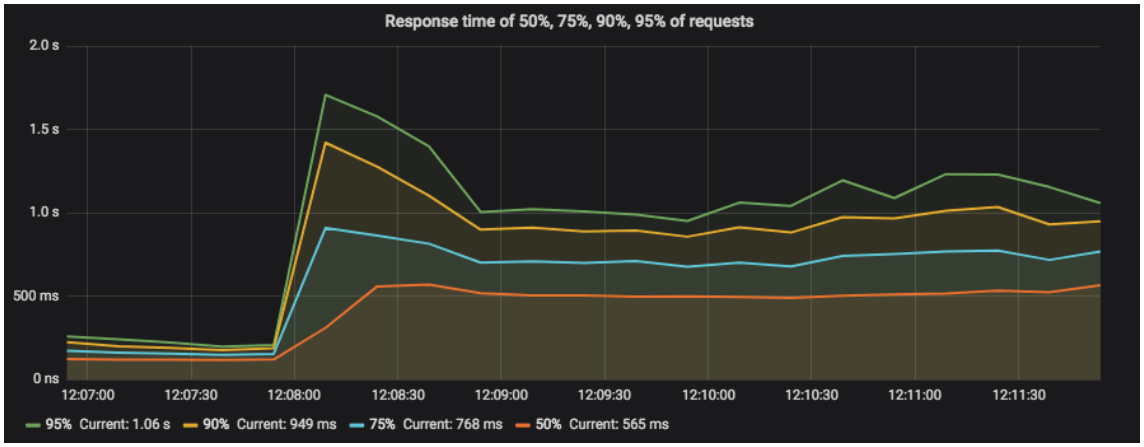

Figura 8.9: Respuesta en percentiles, concurrencia 40 sin autoescalado

8.3.4. Concurrencia 5x40

Si seguimos aumentado el grado de concurrencia en 100 o 200, realmente no obtenemos más peticiones por segundo, limitados por el equipo que estamos usando para generar la carga.

Es por ello que en este escenario se utilizan 5 equipos generando carga en concurrencia 40.

Aquí podemos ver como el servicio deja de responder de forma correcta. El gateway colapsa y el micro-servicio encargado de realizar la consulta tarda mucho más que en los escenarios anteriores, el incremento es del orden de segundos.

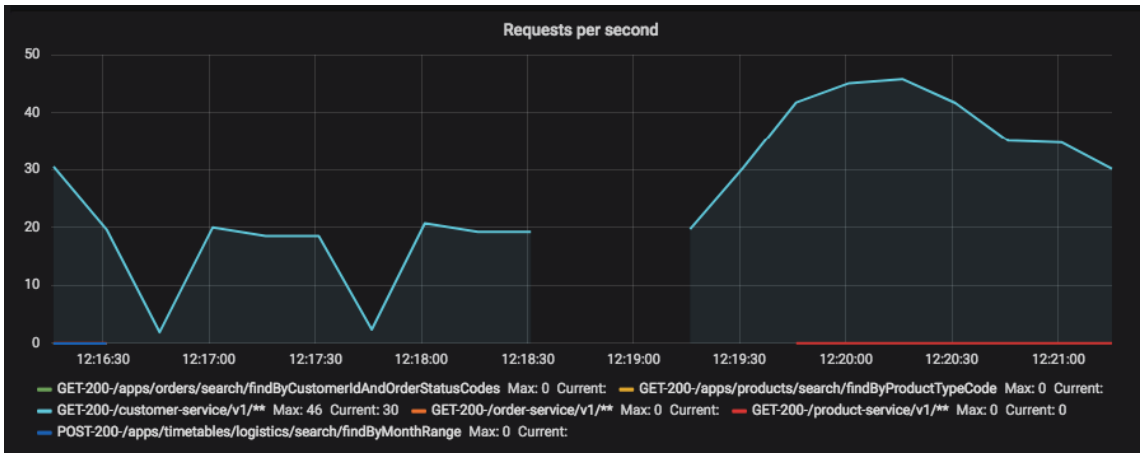

Figura 8.10: Peticiones por segundo, concurrencia 5x40

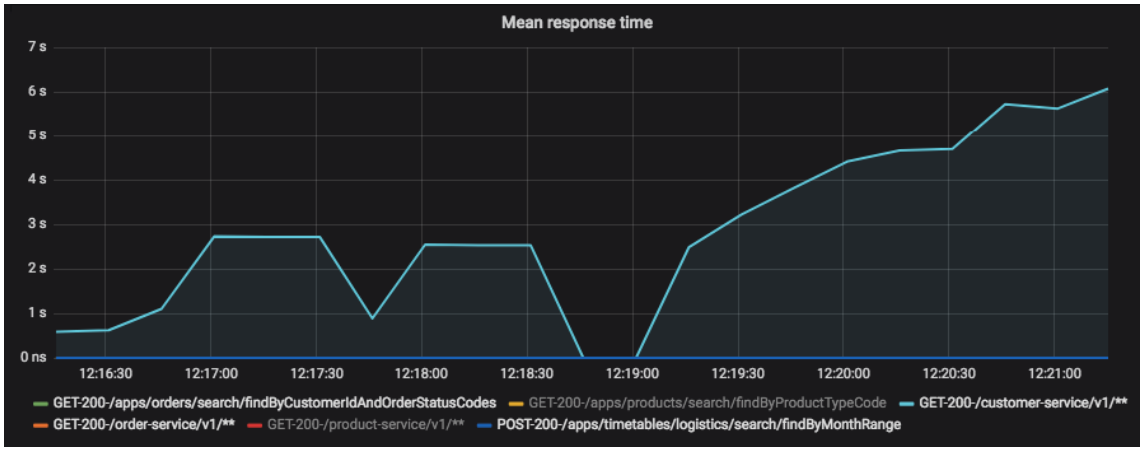

Figura 8.11: Respuesta media, concurrencia 5x40

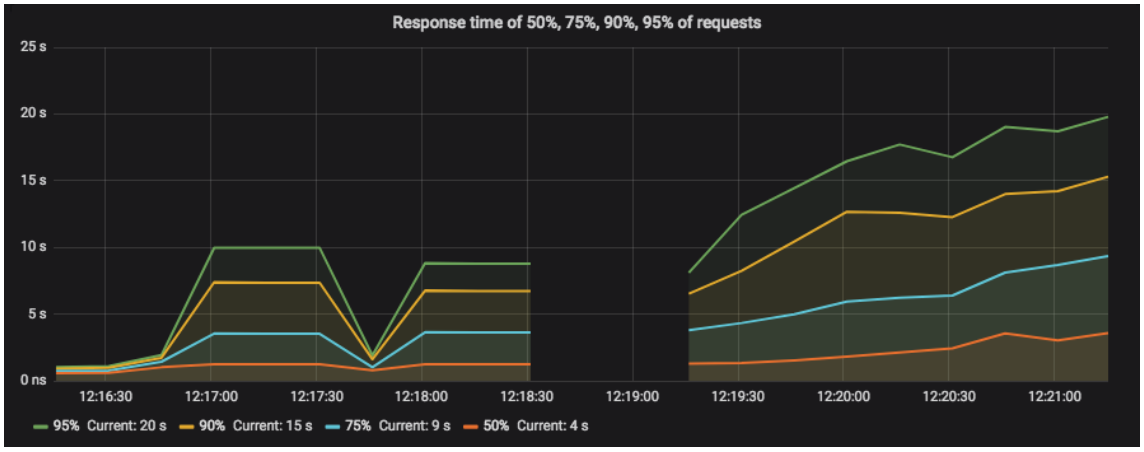

Figura 8.12: Respuesta en percentiles, concurrencia 5x40

A diferencia del resto de escenarios, en este se observan peticiones fallidas. El número de peticiones realizadas en la prueba fue de 51.184, mientras que el número de fallidas fue de 13.760. Por lo tanto un 26,88 % de peticiones fallaron.

8.4 Pruebas de carga con autoescalado

En los siguientes escenarios se habilita el autoescalado.

A diferencia del escenario anterior, se obvia el primer grado de concurrencia, ya que la carga con concurrencia 5 no es suficiente como para escalar el servicio.

8.4.1. Concurrencia 10 con autoescalado

A diferencia del escenario homólogo el cual no disponía de autoescalado, se observa una mejora del tiempo de respuesta en cuanto el sistema escala.

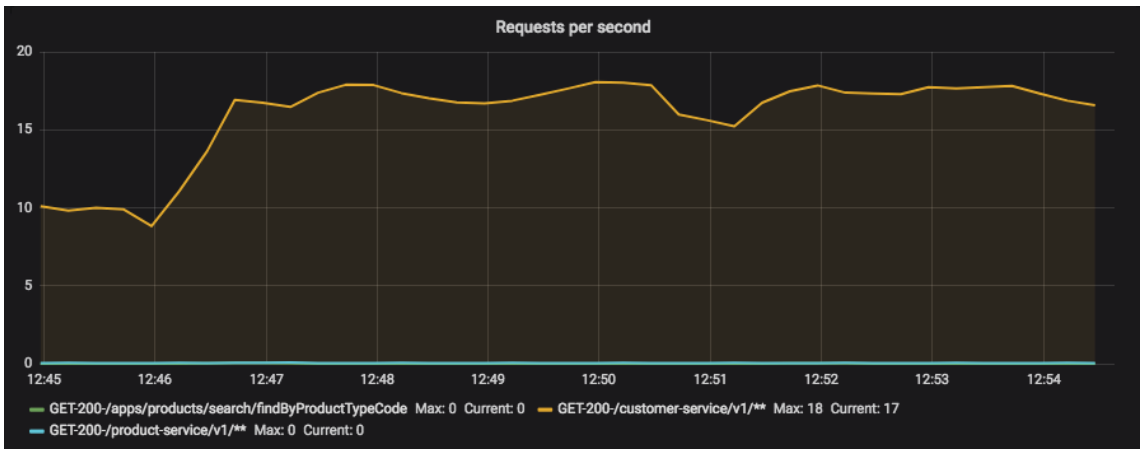

Figura 8.13: Peticiones por segundo, concurrencia 10 con autoescalado

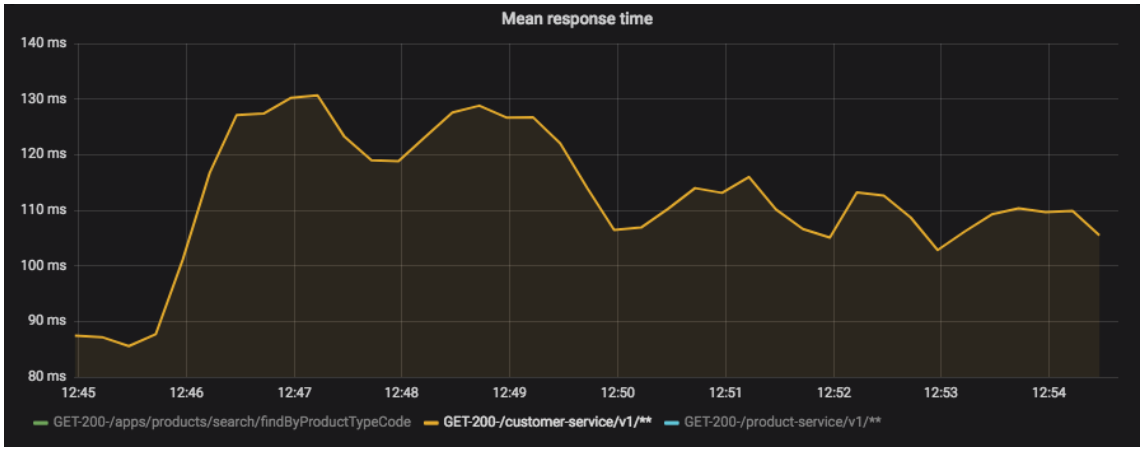

Figura 8.14: Respuesta media, concurrencia 10 con autoescalado

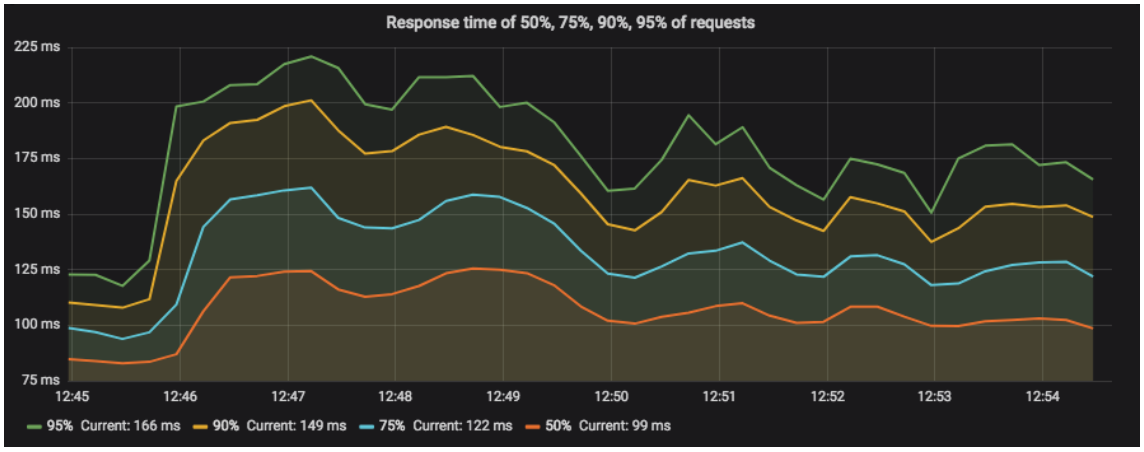

Figura 8.15: Respuesta en percentiles, concurrencia 10 con autoescalado

8.4.2. Concurrencia 40 con autoescalado

Si aumentamos la concurrencia en nuestro generador de carga de nuevo, volvemos a ver la mejora del tiempo de respuesta.

Tanto en este escenario como en el anterior, el micro-servicio que escala es el que realiza la petición a base de datos. El gateway permanece estable, ya que no necesita escalar para absorber el número de peticiones.

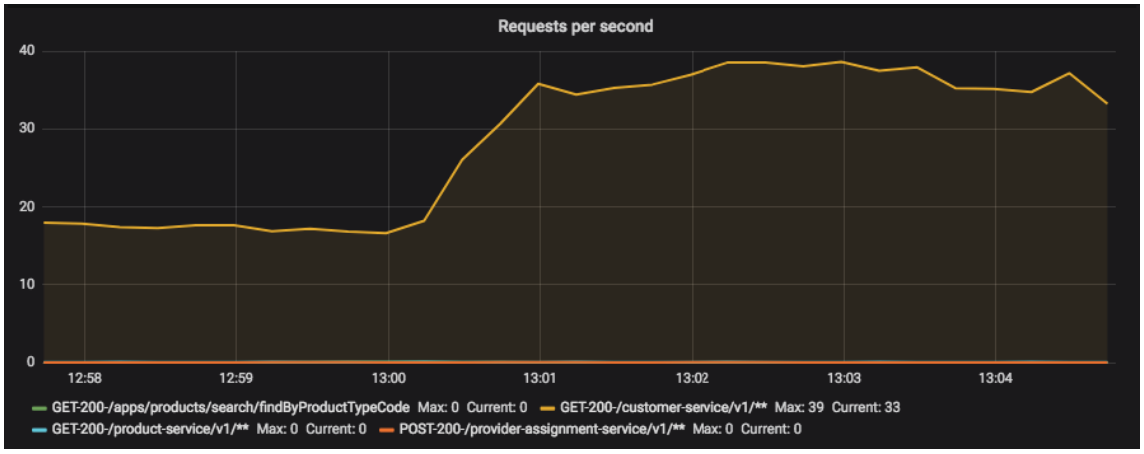

Figura 8.16: Peticiones por segundo, concurrencia 40 con autoescalado

Figura 8.17: Respuesta media, concurrencia 40 con autoescalado

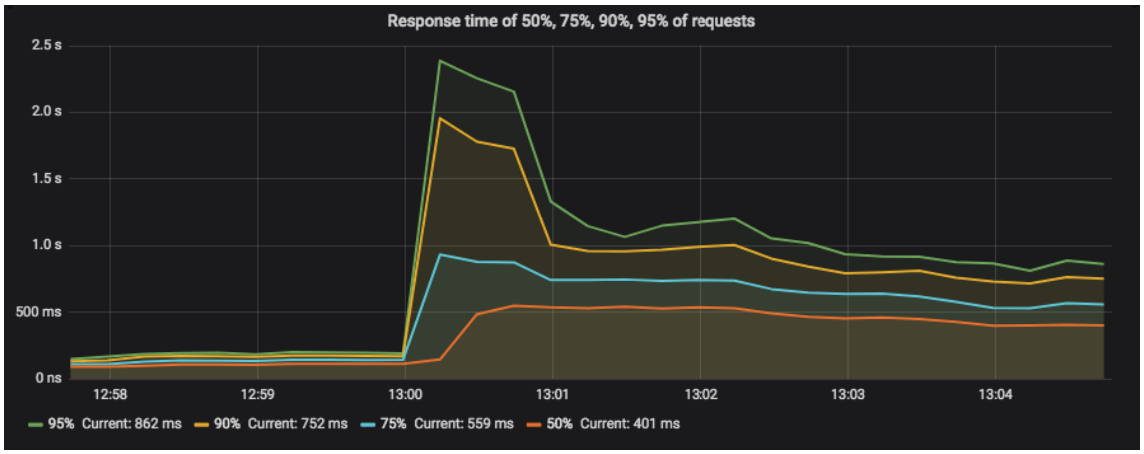

Figura 8.18: Respuesta en percentiles, concurrencia 40 con autoescalado

8.4.3. Concurrencia 5x40 con autoescalado

En este escenario, escaló tanto el gateway como el micro-servicio encargado de realizar las peticiones a base de datos.

El punto de vista es desde uno de los gateway, por eso observamos menos peticiones por segundo en los gráficos siguientes.

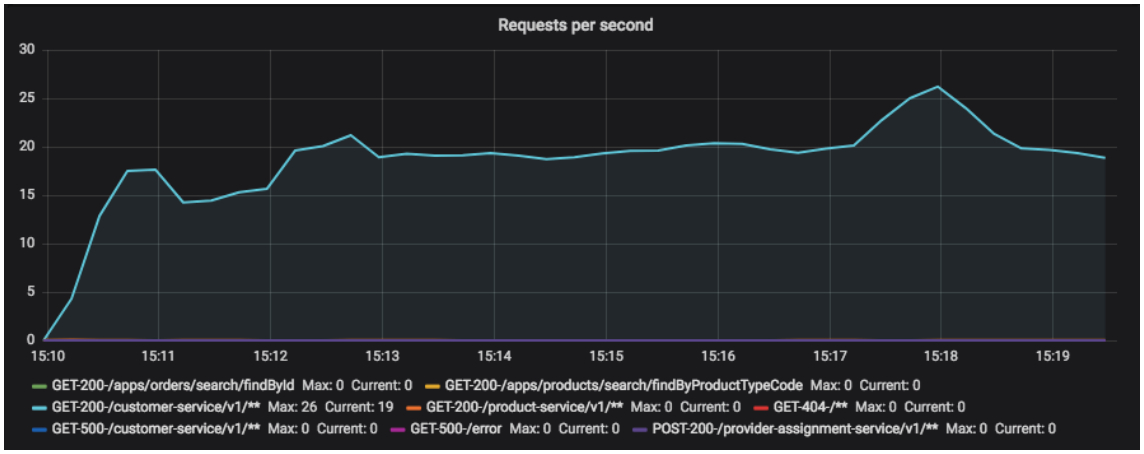

Figura 8.19: Peticiones por segundo, concurrencia 5x40 con autoescalado

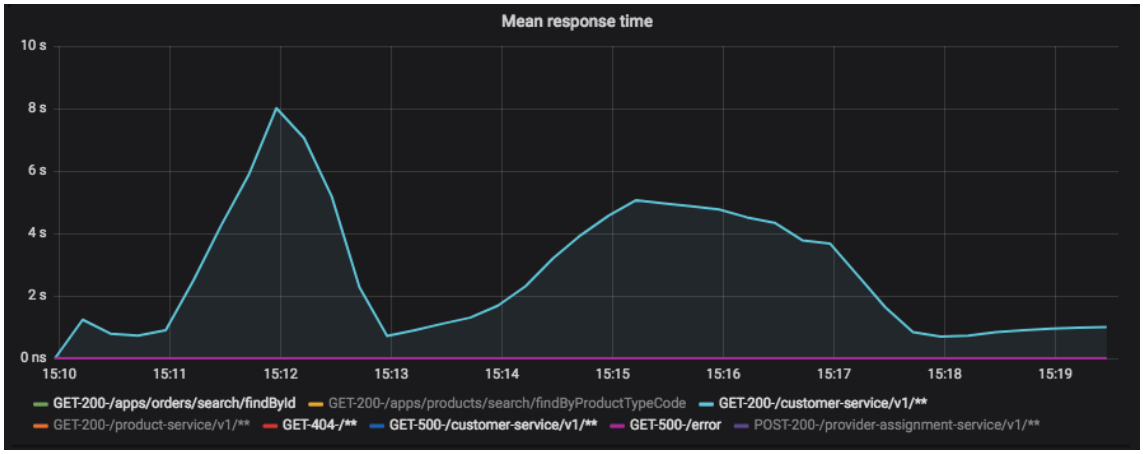

Figura 8.20: Respuesta media, concurrencia 5x40 con autoescalado

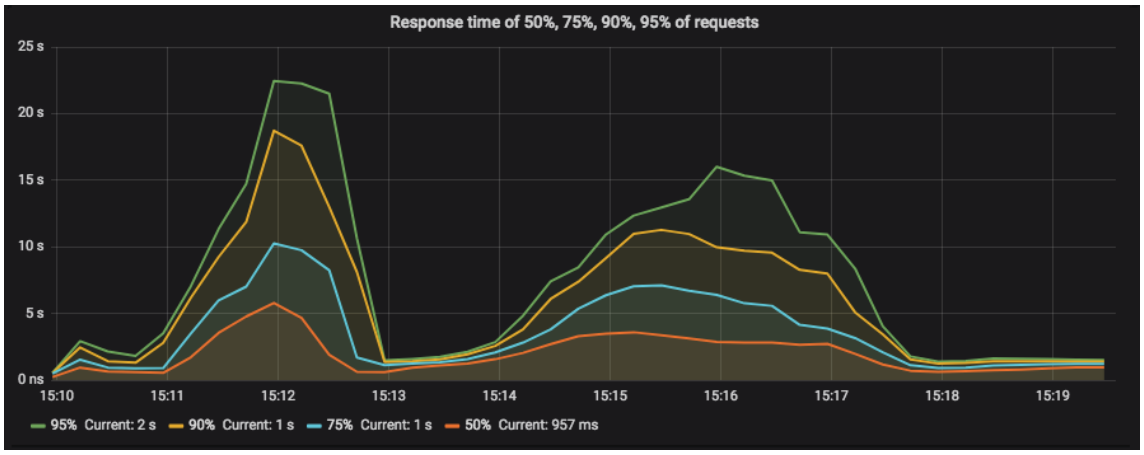

Figura 8.21: Respuesta en percentiles, concurrencia 5x40 con autoescalado

Las diferentes máquinas que generaban la carga se fueron añadiendo progresivamente, por eso vemos dos picos en las gráficas.

Al final el tiempo medio de respuesta se estabilizó en 800ms. Al igual que el escenario sin autoescalado, hubo peticiones fallidas. No obstante el porcentaje fue mucho menor. El número de peticiones realizadas en la prueba fue de 31.979, mientras que el número de fallidas fue de 487 . Por lo tanto un 1,52 % de peticiones fallaron.

El tiempo de respuesta asciende a incluso picos de 8 segundos, esto es debido a que el servicio, cinco veces replicado, no puede asumir la cantidad de peticiones recibidas.

Al tener configurado el sistema para que escale horizontalmente hasta 5, la infraestructura no puede adaptarse más a la carga recibida.

CAPÍTULO 9
Conclusión

La tecnología avanza. Ofrece nuevas capas de abstracción, gracias a esto se pueden generar infraestructuras y arquitecturas de aplicación cada vez más complejas. La batalla por ofrecer un servicio que esté siempre disponible es uno de los focos de la informática moderna.

Gracias a avances como los contenedores y a herramientas como Kubernetes podemos gestionar más fácilmente infraestructuras elásticas capaces de auto regenerarse. Donde realizar actualizaciones de los servicios no requiera un tiempo de indisponibilidad.

Además los proveedores cloud nos dan la posibilidad de repartir nuestros nodos a lo largo del mundo. Dándonos la oportunidad de ser más resistentes a fallos y de ofrecer mejor servicio debido a la cercanía.

La monitorización forma un papel crucial. Si no monitorizamos nuestra infraestructura de forma correcta, no podemos saber qué calidad de servicio estamos dando. Además gracias a las alarmas y la automatización, podemos saber qué está ocurriendo en todo momento y poder tomar medidas correctivas al respecto.

Una infraestructura que se gobierne sola sin necesidad de la intervención humana es una de las claves para diseñar infraestructuras cada vez más complejas y resistentes.

Bibliografía

[1] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, John Wilkes "Large-scale cluster management at Google with Borg" 2015. Disponible en https://ai.google/research/pubs/pub43438.

[2] Nicole Forsgren, Gene Kim, Jez Humble. Accelerate. IT Revolution Press, Marzo 2018 https://www.oreilly.com/library/view/accelerate/9781457191435/

[3] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, John Wilkes "Borg, Omega, and Kubernetes" 2016. Disponible en https://ai.google/research/pubs/pub44843.

[4] Kynan Rilee "Scheduling in Kubernetes, Part 1: Node Affinity" Diciembre de 2017. Disponible en https://medium.com/kokster/scheduling-in-kubernetes-part-1-node-affinity-b77c97556424.

[5] Kynan Rilee "Scheduling in Kubernetes, Part 2: Pod Affinity" Diciembre de 2017. Disponible en https://medium.com/kokster/scheduling-in-kubernetes-part-2-pod-affinity-c2b217312ae1.

[6] Mark Betz "Understanding kubernetes networking: services" Noviembre de 2017. Disponible en https://medium.com/google-cloud/understanding-kubernetes-networking-services-f0cb48e4cc82.

[7] Sandeep Dinesh "Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?" Marzo de 2018. Disponible en https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0.

[8] Documentación Kubernetes "POD", Disponible en https://kubernetes.io/docs/concepts/workloads/pods/pod/.

[9] Documentación Kubernetes "Deployment", Disponible en https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[10] Documentación Kubernetes "ReplicaSet", Disponible en https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

[11] Documentación Kubernetes "StatefuLSet", Disponible en https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.

[12] Documentación Kubernetes "Jobs", Disponible en https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/.

[13] Repositorio oficial de Kops Disponible en https://github.com/kubernetes/kops

[14] Addon para aprovisionar balanceadores de carga en kubernetes. Disponible en https://github.com/kubernetes/kops/tree/master/addons/kube-ingress-aws-controller

[15] Repositorio oficial de la herramienta externalDNS https://github.com/kubernetes-incubator/external-dns

[16] Documentación oficial de Helm Disponible en https://helm.sh/docs/

[17] Amazon "Alta disponibilidad (Multi-AZ)". Disponible en https://docs.aws.amazon.com/es_es/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html.

[18] Cliente kubernetes para javascript https://github.com/kubernetes-client/javascript

[19] Cliente kubernetes para nodejs desarrollado por GoDaddy https://github.com/godaddy/kubernetes-client