Graduado en Ingeniería de Informática

Desarrollo y despliegue de infraestructuras en cloud

Development and deployment of a cloud infraestructure

Realizado por
Rubén Ruiz Nieto

Tutorizado por
Eladio Damián Gutiérrez Carrasco

Departamento
Arquitectura de Computadores

MÁLAGA, junio de 2023

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA INFORMÁTICA

Desarrollo y despliegue de infraestructuras en cloud

Development and deployment of a cloud infraestructure

Realizado por
Rubén Ruiz Nieto

Tutorizado por
Eladio Damián Gutiérrez Carrasco

Departamento
Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2023

Fecha defensa: junio de 2023

Abstract

The intention of this study is to comprehend the functioning of the DevOps methodology with containers by employing an application. To achieve this objective, a comprehensive overview of the general concepts related to the employed technologies has been conducted, emphasizing the most relevant aspects of each one. For this purpose, containers have been defined for their usage in development environments, they have been automatically deployed through CI/CD in a container orchestrator, and the application has been monitored. Through this analysis, and based on the carried out implementation, whose manuals are attached to this study, it is concluded that containers, along with orchestrators, are more efficient in many use cases compared to the previously employed technologies for the same purpose.

Keywords: DevOps, Container orchestrator, Container, Continuous integration, Continuous deployment, Monitoring, Observability.

Resumen

La intención del presente trabajo es entender el funcionamiento de la metodología DevOps con contenedores mediante el uso de una aplicación. A tal efecto, y en primer lugar, se ha hecho un repaso por los conceptos generales relativos a las tecnologías usadas, en las que se ha profundizado haciendo hincapié en los aspectos más relevantes de cada una de ellas. Finalmente, para mostrar el funcionamiento de dichas tecnologías, se ha implementado una aplicación. Para ello se han definido los contenedores para su uso en entornos de desarrollo, se han desplegado automáticamente mediante CI/CD en un orquestador de contenedores y se ha monitorizado la aplicación. A través este análisis, y mediante la implementación llevada a cabo, cuyos manuales se encuentran anexados a este trabajo, se concluye que los contenedores junto a los orquestadores son más eficientes en muchos casos de uso que las tecnologías utilizadas anteriormente para el mismo fin.

Palabras clave: DevOps, Orquestador de contenedores, Contenedores, Integración continua, Despliegue continuo, Monitorización, Observabilidad.

Índice

1. Introducción **9**

1.1. Motivación 9

1.2. Objetivos 9

1.3. Estructura del documento 10

2. Estado del Arte **11**

2.1. Virtualización 11

2.2. Contenedores 12

2.3. Escalabilidad 15

2.4. Alta disponibilidad 16

2.5. Máquinas virtuales y contenedores 17

2.6. Orquestadores de contenedores 19

2.7. Observabilidad 20

3. Metodologías de trabajo **23**

3.1. DevOps 23

3.2. Integración Continua y Distribución Continua 24

4. Descripción del entorno tecnológico **27**

4.1. Docker 27

4.1.1. Referencia de Dockerfile 28

4.2. Kubernetes 29

4.2.1. Kubernetes en local 30

4.3. Circle CI 31

4.4. MetalLB 32

4.5. Prometheus y Grafana 32

4.6. Elastic Stack 32

4.7. Istio y Kiali 33

5. Implementación 35
5.1. Estructura de la aplicación 35
5.2. Diseño y construcción de las imágenes de docker de una aplicación 36
5.2.1. Servicio Vote 36
5.2.2. Servicio Worker 37
5.2.3. Servicio Result 38
5.2.4. Despliegue en local con docker-compose 39
5.3. Almacenamiento de las imágenes docker 41
5.4. Diseño y despliegue de una aplicación en kubernetes 42
5.4.1. Servicio Redis 42
5.4.2. Servicio Postgres 43
5.4.3. Servicio Vote 44
5.4.4. Servicio Worker 44
5.4.5. Servicio Result 45
5.4.6. LoadBalancers en local 46
5.5. Diseño de un pipeline en CircleCI 47
5.6. Configuración de mecanismos de observabilidad sobre k0s 50
5.6.1. Logs 51
5.6.2. Métricas 54
5.6.3. Service Mesh 55

6. Conclusiones 61

Apéndice A. Manual de Instalación de docker 67

Apéndice B. Manual de Instalación de k0s 69
B.1. Instalación manual 69
B.2. Instalación con k0sctl 70

Apéndice C. Manual de Instalación de un runner de Circle CI 73

Apéndice D. Despliegue en Kubernetes 75

Apéndice E. Despliegue Elasticsearch y Kibana con docker-compose 81

1
Introducción

1.1. Motivación

El acceso a internet de manera casi universal ha hecho que la demanda de recursos en los servidores cambie constantemente, complicando la planificación de recursos en ello. La proliferación de dispositivos móviles y la tasa de conexiones a internet a través de los mismos, ha jugado un papel fundamental en este hecho [1]. El aumento de servicios que hacen uso intensivo de internet para su funcionamiento, como plataformas de streaming, ha supuesto un aumento del tráfico de red en ellos. Por todo esto, han ido surgiendo mecanismos que permiten el aumento o la disminución de los recursos asociados a un servicio de manera automática.

Debido a las metodologías ágiles de desarrollo [2], los desarrolladores tienen la necesidad de desplegar sus cambios en el código en distintos entornos de manera constante e incremental, dejando desfasadas las grandes actualizaciones que se realizaban en el pasado. Para paliar esta problemática, sin sobrecargar al equipo o equipos de operaciones, se implementan automatismos y baterías de tests que generen confianza en el código desplegado y mecanismos que permitan una rápida actuación en caso de fallo.

Como consecuencia, surgen las metodologías ágiles para planificar despliegues y sus tecnologías asociadas, que intentan solucionar estos problemas permitiendo que los distintos servicios obtengan dinámicamente recursos dependiendo de su uso. Además, facilitan la interoperabilidad de los equipos de desarrollo y operaciones, mejorando su comunicación e integración.

1.2. Objetivos

El objetivo de este trabajo es mostrar las tecnologías utilizadas para realizar despliegues usando metodologías ágiles. Para ello, se explica en que consisten y se analizan las ventajas e inconvenientes en comparación con sistemas antiguos de despliegue.

Para ilustrar lo anterior, se despliega una aplicación de prueba. Con esta aplicación, se emula el proceso desde la fase de desarrollo, hasta la monitorización de la aplicación sobre un cluster similar al que se puede encontrar en entornos cloud.

1.3. Estructura del documento

Con la estructura de este trabajo, se pretende ayudar al lector a entender el proyecto de una manera incremental, desde los conceptos básicos hasta la implementación final.

El documento se estructura en tres partes. Comienza con una explicación de los principales conceptos que se tratan en él, así como antecedentes históricos de los mismos. Acto seguido, se detallan las distintas herramientas y tecnologías, explicando para que sirven y como se usan. Para finalizar, se realiza una implementación de una aplicación usando las tecnologías explicadas anteriormente. En esta descripción de la implementación, se detalla el despliegue en local con fines de desarrollo, el despliegue de la aplicación en un cluster y la monitorización posterior de la misma. Explicando paso a paso cómo replicar la implementación y cómo podría implementarse en otro tipo de entornos.

2
Estado del Arte

Para comprender las ventajas que nos ofrecen las tecnologías y las metodologías de trabajo que abordamos en este documento, necesitamos conocer algunos conceptos referentes a las infraestructuras previamente. Estos conceptos los definiremos en este apartado, además de hacer una pequeña retrospectiva sobre alguna de las tecnologías para comprender cómo han ido evolucionando y que impacto han tenido.

2.1. Virtualización

La virtualización es una tecnología que permite crear múltiples entornos simulados o recursos dedicados desde un solo sistema de hardware físico [3]. Se realiza mediante un software que permite dividir un sistema en entornos separados, distintos y seguros, y repartir los recursos físicos entre los distintos sistemas virtuales llamado hypervisor.

Un hypervisor conecta con el hardware directamente o a través de un sistema operativo, permitiendo asignar parte de los recursos hardware a una o varias máquinas virtuales.[3] Esto permite mantenerlas como sistemas independientes en los que se han simulado mediante software todas las características hardware de una máquina física.

Existen dos tipos de hypervisores: los de tipo 1, también llamados hypervisores bare-metal, se instalan directamente sobre el hardware, por ejemplo ESXi; los de tipo 2 o hypervisores alojados, se instalan sobre un sistema operativo, como VMWare o VirtualBox[3].

A la máquina física en la que se instala el hypervisor, se le llama, generalmente, host o máquina anfitriona, y a las máquinas virtuales creadas a partir del hypervisor, se les suelen llamar guests, o simplemente MV (Máquinas Virtuales)[3].

Una máquina virtual (MV en adelante) es una máquina con el hardware virtualizado que se ejecuta sobre un hardware físico[4]. Esto puede resultar muy pesado, ya que en cada máquina virtual han de ser instalados todos los componentes que necesitaría una máquina física (Poner

ejemplos de los componentes instalados, SO, drivers...). Este exceso de peso genera que la portabilidad de las máquinas, aunque posible, sea lenta, así como su tiempo de despliegue.

La ventaja que proporcionan las MVs sistemas que lo único que comparten es el hardware físico, lo cual proporciona un aislamiento casi completo.

2.2. Contenedores

Las características de la virtualización, generan la necesidad de desarrollar otros sistemas de aislamiento que puedan ejecutar distintos procesos aislados, sobre un mismo hardware, de una manera más ligera y eficiente.

En 1982, Bill Joy, uno de los arquitectos del Unix de BSD y fundador de Sun Microsystems, creó la llamada al sistema chroot() para ayudarse en el desarrollo de Unix. Dado que desarrollaba en la misma computadora donde hacía sus pruebas, quería poder mantener aislado su entorno de desarrollo y construcción del entorno de pruebas [5]. Con chroot() se arrancaba un proceso, pero se le ponía el punto de montaje del directorio root en un punto, previamente seleccionado, del árbol de directorio de linux, consiguiendo así que el proceso se ejecutase junto a otros procesos del SO HOST, pero teniendo acceso solo y exclusivamente al árbol de directorios generado a partir del punto de montaje seleccionado para su directorio root.

En el año 2000 aparecieron las llamadas ”freebsd jails”, estos fueron los primeros contenedores, aunque con muchas limitaciones si los comparamos con los actuales. También aislaban el árbol de directorios, pero también aislaban distintos elementos de la red y los usuarios, teniendo sus propios IP, hostname, set de usuarios e incluso su propio usuario root [6].

Poco después, esta funcionalidad apareció en Solaris, Linux e incluso para Windows. En Linux, esta implementación se llamó ”linux vserverz en Windows ”Parallels Virtuozzo Containers”, ambas eran productos de terceros y las opciones que ofrecía eran muy similares a las de las ”freebd jails”[5].

En 2006 Google desarrolló los Process container. Con ellos podías asilar casi todos los recursos hardware que se asignaban a un comando (CPU, Memoria, Uso del disco, Networking...) del host. Fue implementado con la tecnología cgroups en Linux y con ellos sentó la base de lo que conocemos hoy en día como contenedores.

En 2008 aparecieron los Linux Containers, un motor de contenedores que consiste en una serie de herramientas y librerías que permiten implementar la funcionalidad de los contene-

dores de una forma ligera y a muy bajo nivel [7]. En estos contenedores se puede ejecutar más de un comando y se podían descargar imágenes con librerías y herramientas preinstaladas.

En 2013 docker se convirtió en un estándar para ejecutar código empaquetado, heredaba funcionalidades de sus predecesores, sobre todo de los Linux Containers, aunque estos solo pueden ejecutar un comando por contenedor.

Después de docker surgieron muchos otros sistemas con una arquitectura similar, Por ello, en 2015, tanto docker como otros lideres del sector de los contenedores, decidieron estandarizar el concepto de contenedor. Para ello crearon la Open Container Initiative (OCI en adelante), en la que se definen dos especificaciones de los contenedores: cómo debe ser el motor de contenedores y como han de ser las imágenes de esos contenedores [8].

Los contenedores, tal y como los conocemos hoy en día, son abstracciones lógicas del sistema operativo host que permiten ejecutar de forma aislada un proceso o un conjunto de procesos junto con todas sus dependencias, compartiendo con el host, tan solo, parte del sistema operativo [9].

Este aislamiento los hace ideales para ejecutar múltiples aplicaciones en el mismo servidor, sin que interaccionen entre ellas de una forma no deseada. Además, compartir el sistema operativo con el host, al contrario de lo que sucede con la máquinas virtuales, los hace mucho más ligeros, lo cual tiene un fuerte impacto en los tiempos de creación y despliegue.

La estructura de creación de un contenedor le aporta la capacidad de ser portable, replicable y resiliente, ya que, a partir de un documento llamado Containerfile, en el que se especifica el contenido de los contenedores, se crea una imagen inmutable en la que se ha definido el estado del contenedor y su contenido. Esta imagen será a partir de la cual se instancien todos los contenedores de ese tipo (Figura 1).

Siempre se instanciarán en el mismo estado y se podrá hacer tantas réplicas del mismo como sea necesario en pocos segundos. Por otro lado, este modo de funcionamiento, permite que ante un fallo el contenedor se elimine y se sustituya por uno idéntico al original, solventando de esta manera los errores que haya podido surgir por una modificación o por una corrupción del proceso.

Al tener todas las dependencias empaquetadas junto a la aplicación y ser tan ligeros, estos sistemas, son portables de una forma rápida y cómoda. Esto también nos brinda la facilidad de usar la misma definición de contenedor (o una muy similar) tanto para desarrollar sobre

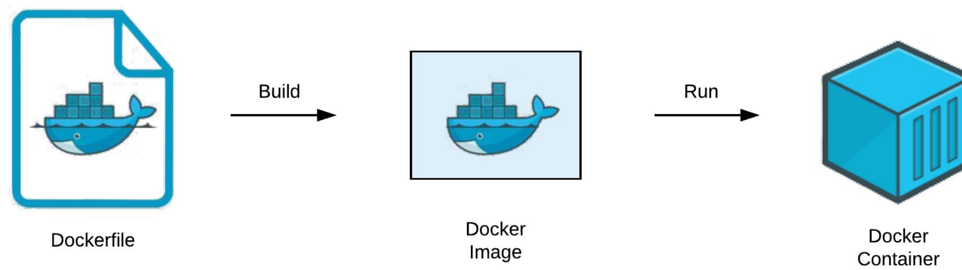

Figura 1: Proceso de construcción de un contenedor

él como para ponerlo en producción, minimizando la problemática que pueda surgir entre los entornos de producción y de desarrollo.

Este modo de funcionamiento también presenta retos a la hora de manejarlos, ya que, al ser volátiles, requieren de mecanismos externos al propio contenedor o estructuras especiales dentro del mismo para persistir los datos, en el caso de ser necesario.

Para conseguir el aislamiento en los contenedores se usan principalmente dos funcionalidades del kernel de linux: los cgroups (control groups) y los namespaces (espacios de nombre).

- **Cgroups** se incluye en el kernel de linux de manera oficial desde la versión 2.6.24. Es una funcionalidad del kernel de linux que permite organizar los procesos en grupos jerárquicos de manera que el uso de varios tipos de recursos puedan ser limitados y monitorizados. La interfaz de cgroups del kernel es proporcionada a través de un pseudo-sistema de ficheros llamado cgroupfs [10].

- **Namespaces** se incluye en el kernel de linux de manera oficial actualmente. Las primeras funcionalidades relacionadas con namespaces empezaron a incluirse en la versión 3.0 del kernel de linux [11]. Sirven para dividir ciertos objetos del kernel de forma que cada proceso vea su propio espacio de recursos. No todos los objetos del kernel pueden dividíos y asignados de manera individual, por lo cual, han de ser compartidos entre todos los procesos independientemente del espacio de nombre en el que se encuentre.

Podemos encontrar principalmente dos tipos de contenedores atendiendo al número de procesos que ejecutan de forma nativa: los de aplicación (como docker) y los sistema (por ejemplo, LXC).

El uso de los contenedores está altamente extendido. Hasta el momento, los más populares han sido los contenedores docker, creando no solo contenedores de procesos, los cuales ya existían, si no también proporcionando una CLI (Comand Line Interface) fácil de usar, mecanismos para el manejo del hardware virtualizado y una documentación pulida y actualizada para facilitar su uso.

2.3. Escalabilidad

La escalabilidad se define como la capacidad de una aplicación o producto para seguir funcionando bien cuando cambia su tamaño o su volumen para ajustarse a las necesidades del usuario [12]. Hay dos formas de escalado: el escalado vertical y el escalado horizontal (Figura 2).

- Se llama **escalado vertical** a la acción de asignar más recursos hardware dentro de una misma máquina, en el caso de una MV, aumentar la cantidad de recursos virtualizados asignados. Este tipo de escalado está bastante limitado por el hardware de la máquina que se está usando, y puede ser excesivamente costoso o incluso inviable por limitaciones de la tecnología. La ventaja de este tipo de escalado ningún mecanismo adicional para hacerlo, tan solo la compra e instalación de hardware nuevo o la asignación de una mayor cantidad de recursos virtuales.

- En el **escalado horizontal**, también se asignan más recursos hardware, pero en este caso, replicando el sistema en otro equipo, o en el mismo, y establecer mecanismos de coordinación entre las réplicas. Al contrario que el vertical, este requiere mecanismos adicionales para su correcto funcionamiento, pero es mucho más flexible y tiene menos limitaciones. La ligereza y facilidad de réplica de los contenedores, los hacen idóneos para este tipo de escalado facilitando el trabajo en entornos cloud.

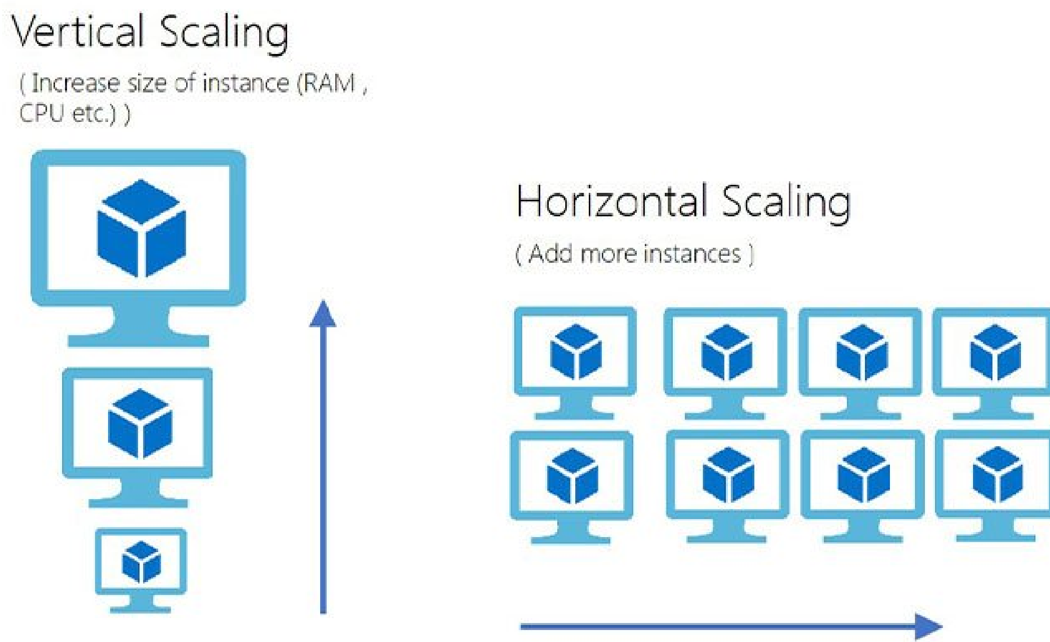

Figura 2: Comparativa entre escalado vertical y escalado horizontal

2.4. Alta disponibilidad

En informática, el término disponibilidad se utiliza para describir el período de tiempo en que un servicio está disponible, así como el tiempo requerido por un sistema para responder a una solicitud realizada por un usuario. La alta disponibilidad es la calidad de un sistema o componente que asegura un alto nivel de rendimiento operativo durante un período de tiempo determinado [13].

La disponibilidad a menudo se expresa como un porcentaje que indica cuánto tiempo de actividad se espera de un sistema o componente en particular en un período de tiempo determinado, donde un valor del 100 % indicaría que el sistema nunca falla [13].

Gracias a las características de los contenedores, los hace ideales para mantener la alta disponibilidad, ya que pueden escalar rápidamente para mantener el tiempo de respuesta y en caso de fallo, se puede sustituir por otro en pocos segundos, obteniendo el servicio en el mismo estado en el que se encontraba en un principio.

En algunos sistemas, para asegurar la disponibilidad incluso durante los tiempos de des-

pliegue, se utiliza el despliegue blue/green (Figura 3). Este tipo de despliegue se basa en tener dos entornos para producción, en uno de ellos se despliega el código nuevo y se prueba, en el otro se sigue usando el código que está actualmente en producción. Una vez el código ha sido probado, se cambia el acceso del balanceador de carga del entorno actual al que contiene el código nuevo. En el próximo despliegue se llevará a cabo el mismo proceso, pero el servidor que alojará el código nuevo, será el que en este ejemplo contenía el código en producción.

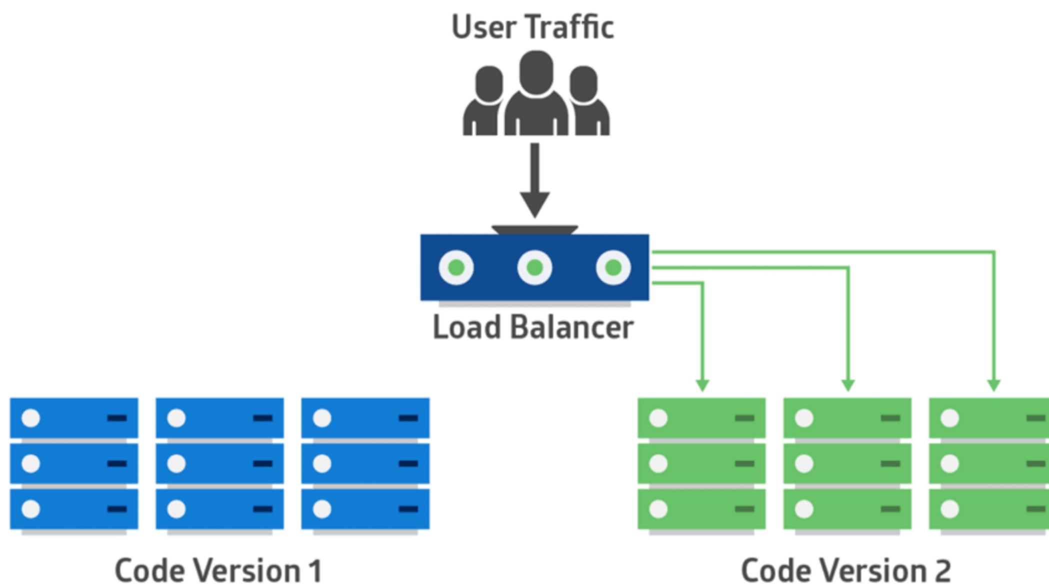

Figura 3: Despliegue blue/green

2.5. Máquinas virtuales y contenedores

El uso de las máquinas virtuales, ha cambiado con el paso de los años. Durante un tiempo fueron el estándar para ejecutar procesos o aplicaciones de forma aislada en el mismo hardware. En la actualidad se usan principalmente para la creación de entornos de pruebas y la de servidores virtuales en un mismo hardware. Actualmente las metodologías de desarrollo tienden a los cambios incrementales en el proyecto para añadir nuevas funcionalidades. Esto ha hecho que uso para ejecutar procesos o aplicaciones ha quedado relegado a un segundo plano, por ser poco ágiles.

El aumento de los cambios en el tráfico de las aplicaciones web, ha hecho que las necesi-

dades de escalabilidad, sean cada vez mayores. Por consiguiente, las necesidades de escalado también han aumentado. Este hecho, hace que los contenedores sean más óptimos para este cometido, ya que para arrancar un contenedor hacen falta apenas unos segundos, mientras, que una MV requiere varios minutos.

MVs	Contenedores
- SO completo, por lo cual pesadas - Gran aislamiento y seguridad - Pensados para conservar el estado - Permite hacer Snapshots	- SO compartido con el host, por lo cual ligeros - Menor grado de aislamiento - Pensados para ser volátiles y stateless

Cuadro 1: Comparativa de las características de las MVs frente a las de los contenedores

Como ilustra la figura 4, en una máquina virtual, se ejecuta todo el sistema operativo, junto a las dependencias de la aplicación y la propia aplicación, todo ello sobre el hypervisor. Por contra, los contenedores no necesitan de un hypervisor, pero si un entorno preparado para ejecutarlos. Este entorno no se puede instalar directamente sobre el hardware, si no que necesita un sistema operativo sobre el cual ser instalado. Los contenedores tampoco requieren un sistema operativo completo instalado en su interior, pero si la aplicación y todas sus dependencias. Estas características lo hacen mucho más ligero, lo cual supone un ahorro en tiempo, un cambio en la metodología de trabajo, y, lo que a las empresas más les resulta más beneficioso, un ahorro económico en la mayoría de los casos por la facilidad de escalado y desescalado.

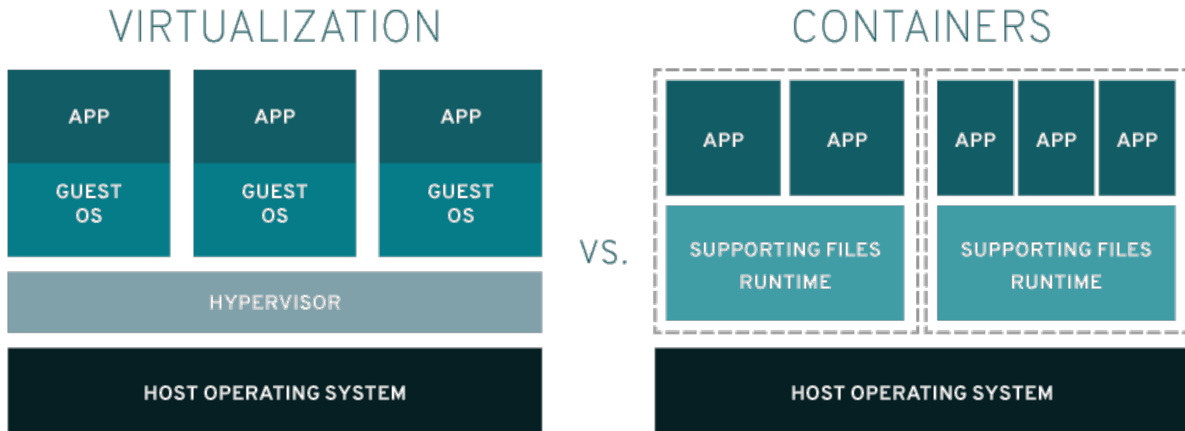

Figura 4: Comparativa de la estructura de un sistema de MVs en contraposición a un sistema de contenedores (RedHat)

2.6. Orquestadores de contenedores

Los contenedores, por sí solos, no tienen la capacidad de coordinarse de forma automatizada. Por tanto, necesitan herramientas externas que permitan la automatización de las operaciones de gestión y coordinación. Estas herramientas se llaman orquestadores de contenedores.

La orquestación de contenedores consiste en la automatización de la mayoría de las operaciones necesarias para ejecutar cargas de trabajo y servicios en contenedores. Estos sistemas permiten gestionar los contenedores del mismo tipo como un grupo, configurarlos, interconectarlos y securizarlos, reduciendo de esta manera los posibles errores humanos [5].

Existen muchos orquestadores de contenedores distintos, aunque la mayoría de ellos, en la actualidad, toman como base o copian la estructura del más popular, kubernetes, a veces abreviado como k8s.

Podemos diferenciar dos grupos en los orquestadores: los que podemos instalar en nuestra propia máquina o conjunto de máquinas y los que son ofrecidos por proveedores de servicios cloud. Estos últimos suelen ser una implementación a medida de kubernetes hecha por el propio proveedor de servicios.

Entre las implementaciones hechas por proveedores de servicios cloud más utilizadas encontramos AWS EKS, ofrecida por Amazon Web Services y GKE, servida por Google Cloud

Platform. Estas implementaciones de kubernetes están muy extendidas, ya que permiten beneficiarse de las ventajas de este orquestador sin tener que lidiar con la complejidad de gestionar el cluster y su seguridad. Además, gracias a la disponibilidad de recursos hardware en un entorno cloud, la escalabilidad horizontal que podemos llevar a cabo es casi ilimitada.

Por otro lado, para entornos en los que tengamos que instalar kubernetes nosotros mismos, existen herramientas que nos alivian esta complejidad. La mayoría de ellas están pensadas para entornos de pruebas y desarrollo, como microk8s o k0s, que describiré brevemente más adelante.

2.7. Observabilidad

En 1960 Rudolf E. Kalman introdujo la definición de lo que él llamó observabilidad, para describir un sistema de control matemático. Según la teoría de control, la observabilidad, está definida como una medida de con que precisión pueden ser inferidos los estados internos de un sistema, conociendo sus salidas externas [14].

Lo que entendemos por observabilidad en los sistemas informáticos, no dista mucho de la definición de Kalman y se suele fundamentar en tres métricas distintas: rendimiento, trazas y logs [15].

- **Métricas**: Las métricas de rendimiento son una representación numérica de datos que pueden ser usados para determinar el comportamiento de un componente o servicio a través del tiempo [16].
- **Trazas**: Representan el recorrido completo que una petición o una acción, hace a través de los distintos módulos de una aplicación [16].
- **Logs**: Son cadenas de texto que contienen los eventos que ocurren en un sistema. Hay dos tipos principales, los logs de sistema, que provienen del sistema operativo, y los logs de aplicación, que proporcionan información sobre qué eventos están ocurriendo en una aplicación [16].

Esta obtención de datos y su posterior análisis, no es compleja en sistemas monolíticos que se ejecutan en un único servidor. Pero en sistemas distribuidos, se convierte en una tarea más ardua.

La aparición de los contenedores ha permitido que las aplicaciones dejen de programarse como sistemas monolíticos y que se divida en módulos independientes. A esta arquitectura se la llama microservicios. De hecho, la tendencia en el ámbito de las aplicaciones es a programarlas como microservicios, ya que, entre otras, permiten el escalado horizontal de los módulos, de forma independiente, atendiendo a sus necesidades. Esto convierte en un reto y a la vez, en una necesidad, la tarea de observar la aplicación, ya que al existir distintos nodos del mismo tipo y distribuidos en distintas máquinas, los puntos de posible fallo se multiplican, a la vez que lo hacen el número de logs y métricas.

Para paliar esta problemática, existen distintos sistemas que permiten recolectar esos logs y métricas en cada uno de los microservicios, concentrarlos, ordenarlos y visualizarlos. Como contrapartida, estos sistemas suelen tener distintas piezas y manejar una gran cantidad de datos, lo cual, hace la infraestructura más pesada. En ocasiones, puede llegar a suponer un sobre coste bastante significativo, sobre todo, en aplicaciones pequeñas.

En los últimos años, con la proliferación de los microservicios y kubernetes, ha surgido un nuevo concepto, que intenta aunar la observación del cluster y la administración del mismo en un único servicio. A esto se le ha llamado malla de servicios o service mesh [17] y se utiliza para controlar el intercambio de datos entre las distintas partes de una aplicación. A diferencia de otros sistemas que también administran esta comunicación, la malla de servicios es una capa visible y específica de la infraestructura integrada a la aplicación, la cual puede registrar si las distintas partes interactúan bien o no, a fin de facilitar la optimización de las comunicaciones y evitar el tiempo de inactividad a medida que crece una aplicación.

3

Metodologías de trabajo

Gracias a las tecnologías descritas en el capítulo anterior, en las empresas se pueden implementar nuevas metodologías de trabajo, que hacen el despliegue de aplicaciones mucho más ágil y disponible para ser dirigido por el equipo de desarrollo. Por tanto, permite ajustar los despliegues a los tiempos de trabajo de dicho equipo y no depender de otro para realizarlos. La metodología que integra las tecnologías vistas anteriormente con el ciclo de desarrollo del producto se llama DevOps.

3.1. DevOps

El concepto DevOps surge de las metodologías agilistas de desarrollo de software. Estas necesitaban una forma de desplegar su código de una forma tan ágil como eran capaces de desarrollarlo [9]. Por ello, DevOps es el acrónimo de Development (Desarrollo) and Operations (Operaciones tecnológicas)

DevOps es un marco de trabajo y una filosofía en constante evolución que promueve un mejor desarrollo de aplicaciones en menos tiempo y la rápida publicación de nuevas o revisadas funciones de software o productos para los clientes (Figura 5) [18]. Con la metodología DevOps se promueve una comunicación continua más fluida, colaborativa y transparente entre equipos de desarrollo de aplicaciones y sus homólogos en operaciones tecnológicas.

Bajo un modelo de DevOps, los equipos de desarrollo y operaciones ya no están separados. A veces, los dos equipos se fusionan en uno solo, donde trabajan en todo el ciclo de vida de la aplicación, desde el desarrollo y las pruebas, hasta la implementación y las operaciones, y desarrollan una variedad de habilidades no limitadas a una única función [18]. Los equipos utilizan prácticas para automatizar los procesos que anteriormente habían sido manuales y

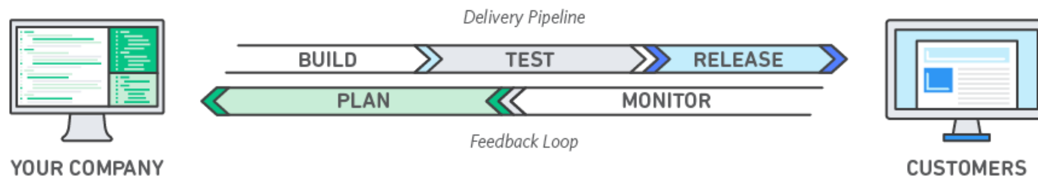

Figura 5: Ciclo de vida de una aplicación usando la metodología DevOps

lentos. Utilizan un stack de tecnología y herramientas que los ayudan a operar y mejorar aplicaciones de forma rápida y confiable. Además, estas herramientas ayudan a los ingenieros a realizar de forma independiente tareas que normalmente hubieran requerido la ayuda de otros equipos (por ejemplo, implementar código o aprovisionar infraestructura), lo que incrementa todavía más la velocidad del equipo [18].

Es un término relativamente moderno que se cree que surgió en 2008 en una convención sobre desarrollo ágil, y que se ha extendido rápidamente. En la actualidad, existen discrepancias sobre qué es DevOps y cómo se implementa en un equipo. Al ser una metodología pensada para ser ágil y flexible, cada equipo la implementa como mejor se adapta a sus necesidades, aunque hay algunos conceptos que son la base de la metodología:

- Automatización de los despliegues.
- Baterías de pruebas automatizadas.
- Monitorización constante de las aplicaciones.
- Comunicación eficiente entre equipos.

Gracias al buen resultado que ha dado esta metodología, otros equipos se han ido sumando a este movimiento. Por ejemplo, DevSecOps, que permite que las prácticas de seguridad se integren en el enfoque de esta forma de trabajo o DataOps, que utiliza esta metodología en el análisis de datos.

3.2. Integración Continua y Distribución Continua

Integración continua y distribución continua (CI/CD en adelante) es un método para distribuir las aplicaciones a los clientes con frecuencia, mediante el uso de la automatización en

las etapas del desarrollo de aplicaciones. Se trata de una solución para los problemas que puede generar la integración del código nuevo. En concreto, el proceso de CI/CD incorpora la automatización y la supervisión permanentes en todo el ciclo de vida de las aplicaciones, desde las etapas de integración y prueba hasta las de distribución e implementación [19].

En estos sistemas se configuran una serie de pasos, que pueden ser tanto secuenciales como paralelos, en los que se ejecutan las tareas previas al despliegue y el despliegue en sí. A estos pasos se le denomina pipeline.

Aunque hemos tomado el significado de CI/CD como integración continua y distribución continua, estas siglas tienen diferentes significados. Mientras que CI siempre se refiere a la integración continua, proceso que implica que se diseñen, prueben y combinen los cambios nuevos en el código de una aplicación con regularidad en un repositorio compartido, CD puede referirse tanto a la distribución como a la implementación. No se suele hacer distinción entre ellos, ya que son conceptos íntimamente relacionados, ambos se refieren a la automatización de las etapas posteriores del proceso, pero a veces se usan por separado para explicar hasta dónde llega la automatización [19]. Por lo general, la CI se refiere a que los cambios que implementa un desarrollador en una aplicación se someten a pruebas automáticas de errores y se cargan en un repositorio para que luego el equipo de operaciones pueda implementarlos en un entorno de producción. Por otro lado, la CD hace referencia al lanzamiento automático de los cambios que implementa el desarrollador desde el repositorio hasta la producción, para ponerlos a disposición de los clientes (Figura 6).

Figura 6: Diferencias entre integración continua, entrega continua y despliegue continuo

Normalmente, un pipeline ejecuta los siguientes pasos (Esto lo puedo poner redactado):

- Detecta un cambio en una rama del repositorio concreta
- Descarga el código de esa rama

- Lo compila en caso de ser necesario
- Ejecuta una batería de tests
- Lo despliega

A esta serie de pasos se le puede añadir también revisiones de código para ver si cumple los estándares definidos, despliegues en entornos de testing para hacer tests de integración... al ser una metodología flexible, cada empresa la implementa dependiendo de sus necesidades.

Dependiendo del sistema de CI/CD que se use y el diseño que hagamos de nuestro proceso de CI/CD, podemos conectar unos pipelines con otros, haciéndolos, de esta manera, modulares y reutilizables.

4

Descripción del entorno tecnológico

En este apartado profundizaremos en cada una de las herramientas que usaremos en la parte técnica de este proyecto. De esta manera, conseguiremos dar una visión más precisa sobre como funcionan las herramientas relacionadas con las tecnologías vistas anteriormente y una mejor comprensión de qué hacen estas herramientas y cómo lo hacen.

4.1. Docker

Docker es un motor de containerización de código abierto, el cual automatiza el empaquetamiento, el envío y el despliegue de cualquier aplicación software, ofrecidos en forma de contenedores ligeros, portables y autosuficientes que se ejecutarán virtualmente en cualquier lugar [9].

Un contenedor docker es un paquete de software que comprende todo lo necesario para ejecutar una aplicación de forma independiente. Puede haber múltiples contenedores docker en una única máquina y estos estarán completamente aislados tanto de otros contenedores como de la máquina host [9].

En el proceso de instalación de docker, es opcional añadir un plugin llamado docker-compose que se utiliza para escribir y guardar la configuración de arranque de uno o varios contenedores para distribuirla y reutilizarla. Esto puede ser muy útil en entornos de desarrollo, ya que nos permite arrancar todo el entorno de forma automática, tan solo lanzando el comando 'docker-compose up' en el directorio en el que esté nuestro fichero de configuración. Esta configuración se escribe en formato *.yaml.*

Docker también provee un lugar centralizado que se usa para mantener las imágenes de docker, tanto en un repositorio público como en uno privado. En este repositorio de imágenes

gratuito, cualquiera puede publicar sus propias imágenes, tanto para uso personal, como para que pueda ser utilizada por la comunidad [9]. Las imágenes más descargadas de este repositorio son las oficiales, publicadas y actualizadas por las empresas o grupos que mantienen las tecnologías, como pueden ser Node.js o PHP.

4.1.1. Referencia de Dockerfile

Ya que Dockerfile es un formato específico para la construcción de contenedores, explicaremos brevemente su sintaxis para que se entienda de manera adecuada. Para empezar a diseñar un Dockerfile, se escoge una imagen base, ya sea una imagen vacía preparada para la creación de contenedores desde cero, una imagen con un sistema operativo completo o una imagen con el entorno de ejecución para un lenguaje en concreto. Por ello, al principio de los Dockerfiles encontramos normalmente la instrucción **FROM** como primera linea:

```
FROM nombre_imagen:version
```

Las instrucciones se van ejecutando de forma secuencial e incremental, por ello, es conveniente que se ordene de manera adecuada el docuemnto. En el resto del documento el formato básico es:

```
#Comentario
INSTRUCCION argumentos
```

Aunque este formato no diferencia entre mayúsculas y minúsculas, por convenio, las instrucciones se ponen en mayúsculas para mejorar la legibilidad. Estas instrucciones son propias de docker. Algunas de las más comunes son:

- **RUN**: se le pasa como argumento un comando del sistema operativo base que se esté usando en la imagen para ser ejecutado. Por ejemplo *RUN echo hello*

- **COPY**: esta instrucción sirve para copiar ficheros o directorios desde el host a la imagen. Tiene dos argumentos, el origen y el destino.

- **ENV**: sirve para crear una variable de entorno dentro de la imagen.

- **WORKDIR**: se selecciona un directorio dentro de la imagen para que sea el directorio principal de la misma.

Al final de la imagen encontramos la instrucción **CMD**, la cual indica que comando se ejecutará cuando el contenedor se levante. La sintaxis de la instrucción CMD tiene dos formas. La primera, en formato ejecutable:

```
CMD ["ejecutable", "arg1", "arg2", ..., "argN"]
```

La segunda, en formato shell:

```
CMD command arg1 arg2 ... argN
```

Aunque casi siempre se pueden usar indistintamente, hay pequeñas variaciones, por ejemplo, en el formato ejecutable no se sustituirán las variables de entorno.

4.2. Kubernetes

Kubernetes es el orquestador de contenedores que goza de mayor popularidad. Es de código abierto y tiene una amplia comunidad contribuyendo a su desarrollo. Además existen de una gran cantidad de implementaciones hechas a medida para usarlo como servicio, la mayoría de ellas ofrecidas por proveedores de servicios cloud.

Los elementos más básicos en kubernetes a nivel de infraestructura son:

- Nodos Máster: se encargan de la gestión de los recursos del cluster de kubernetes.
- Nodos Worker: ejecutan las tareas dentro del cluster. En ellos se pueden levantar varios contenedores a la vez de manera aislada entre ellos.
- Etcd: es una almacén clave-valor que aloja, de manera no centralizada, los datos a los que se necesita acceder en un cluster distribuido.
- Capa de red: conecta los distintos contenedores dentro del cluster mediante una red interna aislada de la red exterior del cluster.

En los nodos máster encontramos:

- El servicio de API, a través del cual podemos gestionar el cluster de manera externa y los contenedores pueden recibir y enviar información al nodo máster.
- El planificador, que se distribuye los distintos contenedores a los nodos worker dependiendo de la carga de trabajo.

- El etcd, que almacena los datos a los que se necesita acceder desde todo el cluster.
- El controller-manager, que supervisa de modo continuo del estado del cluster para llevar a cabo las modificaciones necesarias en el mismo.

Por otro lado, en los nodos worker, encontramos tres elementos:

- El motor de contenedores, mediante el que se ejecuta la aplicación.
- kubelet, con el que es posible vigilar los distintos contenedores en el nodo.
- kube-proxy, que permite abstraer la conexión entre los contenedores en un mismo nodo de la red completa del cluster.

Para desplegar contenedores sobre un cluster de kubernetes, la unidad mínima de organización es el Pod. Un Pod es un grupo de uno o más contenedores que comparten el almacenamiento, la red interna y unas especificaciones de cómo ejecutar los contenedores [20]. Por lo general, se desaconseja generar Pods directamente, es recomendable hacerlo a través de un controlador que le aporte las capacidades de autorecuperación, de replicación y de gestión. El controlador que se suele usar para generar Pods se llama Deployment, quien además genera los elementos de control necesarios para su gestión.

Como los Pods son replicables, podemos tener distintas instancias del mismo en distintos nodos. Es deseable acceder a los Pods de una forma transparente y uniforme. Para esto se utilizan los Services, un elemento que hace las veces de proxy inverso y de balanceador de carga interno en el cluster. Estas características permiten abstraernos de a cual de los Pods desplegados, del mismo tipo, estamos accediendo.

Para gestionar como se interacciona con la aplicación desde fuera del cluster, existe un objeto de kubernetes llamado Ingress, con la capacidad de enviar las peticiones al Service correspondiente mediante reglas de enrutamiento, hacer las veces de balanceador de carga y añadir soporte SSL.

4.2.1. Kubernetes en local

Dado que la arquitectura de kubernetes es muy compleja, instalarla en entornos locales para desarrollo sería muy costoso, tanto en recursos hardware como humanos. Es por esto, que

en entornos de desarrollo, se usan algunas herramientas que construyen un cluster de una manera más sencilla. Estas herramientas suelen tener opciones tanto para desarrollar proyectos en un solo nodo como para desarrollar proyectos que necesiten varios nodos. Algunos ejemplos de estas herramientas son minikube, microk8s y k0s.

- **minikube**: es una herramienta multiplataforma, desarrollada por el equipo de kubernetes, con una gran cantidad de opciones de personalización, lo cual puede impactar negativamente en la facilidad de configuración del cluster.
- **microk8s**: desarrollada por Cannonical, está soportada por toda distribución que permita instalar paquetes de tipo Snap, aunque está pensada especialmente para ser ejecutada en Ubuntu.
- **k0s**: al igual que minikube, se trata de una herramienta multiplataforma. Está diseñada por el equipo de Mirantis su principal pretensión es poder desplegar un cluster de kubernetes con la menor fricción posible por parte del equipo de operaciones. De hecho, el nombre k0s pretende representar k8s con 0 fricción.

Una vez analizadas las ventajas e inconvenientes de estas herramientas, el presente trabajo se ha desarrollado sobre k0s. Ya que proporciona tanto flexibilidad a la hora de desplegar la infraestructura, como facilidad para hacerlo. Además, su uso es muy similar al de kubernetes.

4.3. Circle CI

Circle CI se trata de una plataforma de CI/CD fundada en 2011. Según su página web, destacan por ser de los más rápidos, escalables y seguros del mercado. Su configuración se escribe en formato *.yaml*, muy usado en configuraciones de servicios, lo que lo hace más sencillo de utilizar.

Circle CI permite ejecutar una serie pipelines en su entorno, en nodos llamados runners, de manera gratuita. También permite ejecutar esos pipelines en un entorno propio mediante sus Self-Hosted runner. Esto no solo es útil porque no consume créditos, si no porque habilita hacer despliegues automatizados en una red local o detrás de un firewall sin necesidad de vulnerar la seguridad de la red. Este último es uno de los principales motivos por los que se ha escogido para el trabajo actual, ya que el entorno utilizado está en una red privada.

4.4. MetalLB

MetalLB es una implementación de balanceador de carga para clusters de kubernetes bare-metal usando protocolos de enrutamiento estándares [21]. Es necesario para utilizar balanceadores de carga en este tipo de entornos, ya que kubernetes no lo proporciona por defecto.

En entornos cloud se suelen usar los balanceadores de carga que ofrece el propio cloud de forma independiente a kubernetes. Si no existe ninguna implementación de balanceador de carga cuando se define uno en kubernetes, el balanceador de carga quedará, al implementarlo, se quedará en estado ”pending”hasta que una implementación esté disponible.

4.5. Prometheus y Grafana

Dentro de los sistemas de monitorización de métricas, está Prometheus, un software de código abierto que recoge métricas a través de endpoints HTTP para guardar y procesar cualquier serie temporal puramente numérica [22]. Es decir, podremos almacenar y procesar de forma cronológica cualquier colección de datos numéricos medidos en un momento determinado.

Para visualizar esos datos, son necesarias herramientas que los muestren de forma sencilla y que puedan relacionarlos entre sí. Para tal propósito, surgen distintos paneles de control, como Grafana. Grafana, un software de código abierto y gratuito, capaz de integrarse tanto con Prometheus como con otros sistemas de recolección y procesamiento de métricas [23]. En el panel de administración de Granafa, se puede configurar tanto la forma de visualización de las gráficas, como relacionar distintas métricas entre sí. Estos datos se obtienen mediante querys de consulta que se realizan, en nuestro caso, a Prometheus o a cualquier otro sistema de almacenamiento de series temporales.

4.6. Elastic Stack

La pila de tecnologías Elastic, o Elastic Stack, está formado por distintos componentes, que tienen como cometido desde la recogida de todo tipo de métricas y logs, hasta su visualización en un panel de administración. Esta pila de tecnologías, desarrollada por la compañía Elastic, está compuesta por cuatro componentes fundamentales, aunque es lo suficientemente flexible

para añadir o eliminar algunos de ellos. El stack de consumo y procesamiento de logs está formado por: Filebeat, Logstash, ElasticSearch y Kibana [24].

- **Filebeat** es el recolector de logs del Elastic Stack. Se instala junto a cada uno de los elementos de la aplicación, y se encarga de recolectar y enviar los logs a Logstash. En el caso de estar en un sistema de contenedores docker/OCI, Beats tiene que desplegarse como un contenedor independiente de la aplicación, ya que cada contenedor está pensado para ejecutar un único proceso.
- **Logstash** es el concentrador y procesador de logs del Elastic Stack. En él se define de que forma se curan los datos y se formatean para ser enviados a la pieza que los almacena. No es una pieza necesaria en el stack, pero sí muy útil, sobre todo si vamos a recoger logs con distintos formatos u orígenes.
- **ElasticSearch** es un motor de búsqueda y analítica distribuido, capaz de abordar un número creciente de casos de uso. Como núcleo del Elastic Stack, almacena de forma central los datos para una búsqueda sencilla, rápida y eficiente que escala con facilidad.
- **Kibana** es una interfaz de usuario gratuita y abierta que permite visualizar y ordenar los datos de Elasticsearch y navegar en el Elastic Stack. Para tal propósito, envía querys y genera índices en ElasticSearch que permiten hacer consultas muy eficientes.

4.7. Istio y Kiali

Istio es una plataforma de malla de servicios con tecnología open source que permite controlar el intercambio de datos entre servicios. El diseño de esta plataforma facilita su ejecución en distintos entornos: on-premise, alojados en la nube, en contenedores de kubernetes y en servicios que se ejecutan en máquinas virtuales, entre otros [25].

Istio instala proxies en cada uno de los servicios por los que pasa todo el tráfico de los Pods asociados, pudiendo así analizarlos y conectarse con el resto de proxies para intercambiar información. Para visualizar este tráfico, se suele usar Kiali, una herramienta que sirve como panel de control y de administración para Istio. A través de ella, se puede obtener toda la información que Istio recoge. También se puede conectar con otros servicios para añadir métricas, trazas, logs... y tener más información centralizada en un único punto.

5

Implementación

Una vez explicadas las distintas tecnologías involucradas en este ciclo de trabajo, procederemos, en este apartado, a implementarlas a través de una aplicación de ejemplo llamada *voting-app*.

Generaremos las imágenes de docker y las desplegaremos en local usando docker-compose. Después, veremos como se configuraría y desplegaría de forma automática en un cluster de kubernetes.

Una vez haya sido desplegada la aplicación, veremos como desplegar mecanismos que nos permitan observarla para detectar posibles fallos o incidencias.

5.1. Estructura de la aplicación

La aplicación que usaremos es una aplicación de ejemplo que sirve para votar entre dos opciones y almacenar los votos en una base de datos. Se puede encontrar el código en `https://github.com/dockersamples/example-voting-app`.

La aplicación consta de 5 servicios distintos. Dos de ellos son frontends, uno para votar y otro para visualizar los resultados, llamados vote y result respectivamente. El primero está escrito en python y el segundo en javascript.

En el backend encontramos tres servicios. Un worker, escrito en .NET, que se encarga de recoger y procesar las peticiones del frontend vote, una base de datos postgres, para almacenar los datos de las votaciones y un redis que funciona como cola para las peticiones de escritura sobre la base de datos.

El servicio vote, muestra un interfaz en la que se vota entre dos opciones. Esta envía el voto a una cola redis, que es consumida por el servicio worker, que procesa la petición y la almacena en la base de datos. Por úlitmo, el servicio result, muestra los resultados de las votaciones (Figura 7).

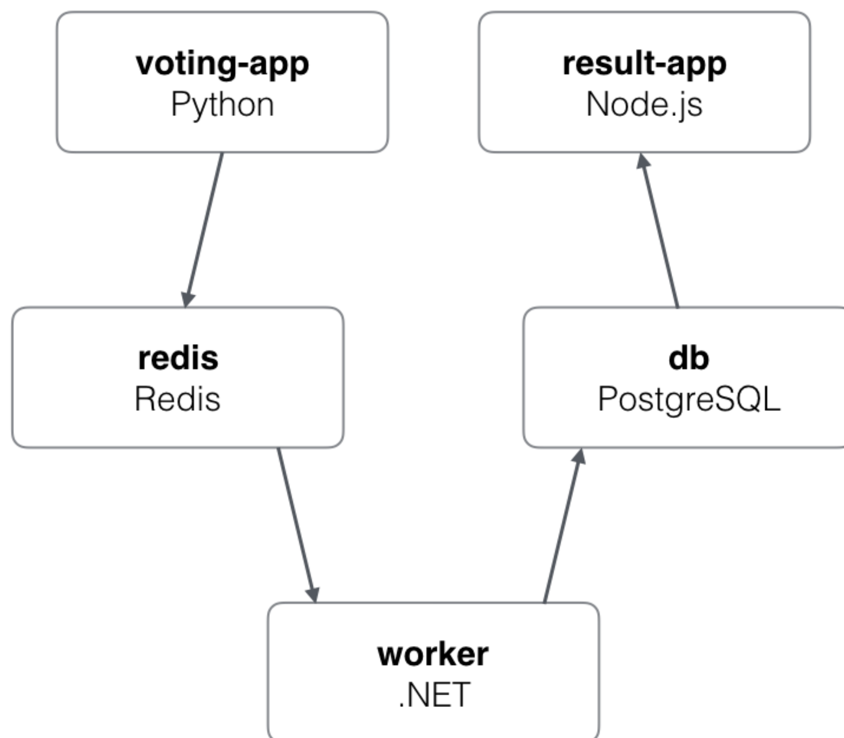

Figura 7: Estructura de la aplicación voting-app

5.2. Diseño y construcción de las imágenes de docker de una aplicación

Para cada uno de los servicios se ha de construir una imagen docker distinta. De esta forma podrán ejecutarse y escalar independientemente. Crearemos las imágenes de los servicios vote, result y worker, ya que tanto redis como postresql tienen sus imágenes subidas a dockerhub de donde las podremos descargar directamente.

5.2.1. Servicio Vote

El servicio vote provee de una interfaz de usuario escrita en python para realizar los votos en la aplicación. Envía los datos a una cola redis para que posteriormente sean consumidas por el worker.

```
# Usa como base la imagen oficial del entorno de ejecución de python versión 3.9-
    slim
FROM python:3.9-slim

# Instala la herramienta curl
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Elige el directorio de la aplicación
WORKDIR /app

# Copia e instala las dependencias contenidas en el archivo requirements.txt
COPY requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt

# Copia el código desde el directorio actual al directorio de la aplicación dentro
    del contenedor
COPY . .

# Expone el puerto 80
EXPOSE 80

# Define el comando que se ejecutará cuando se lance el contenedor
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80", "--log-file", "-", "--access-logfile
    ", "-", "--workers", "4", "--keep-alive", "0"]
```

En este fichero vemos como copia primero el archivo *requirements.txt*, archivo en el que se define las dependencias del proyecto, y luego el código. Esto aporta grandes beneficios por dos factores. El primero, las dependencias cambian mucho menos que el proyecto. El segundo, docker se construye por capas y cada linea en el Dockerfile, representa una capa. Esto permite que docker use las capas que no han cambiado de una caché y solo construya el resto, lo que aumenta la velocidad de construcción notablemente en Dockerfiles construidos teniendo en cuenta esta funcionalidad

5.2.2. Servicio Worker

El servicio worker, escrito en /.NET, consume los votos de la cola Redis para almacenarlos en una base de datos PostgreSQL. Como en este servicio es necesario que el código se compile, se crea una imagen docker llamada multistage. Esta tiene dos fases, la primera, que se nombra con el seudónimo builder, sirve para compilar la aplicación. En la segunda, se copia la aplicación compilada y se ejecuta usando el entorno de ejecución del lenguaje en concreto. Esto permite mantener que el entorno de compilación y de ejecución estén separados, lo cual hace

que la imagen sea ligera, ya que lo único que tiene de la imagen de compilación es el código compilado.

```
# Usa como imagen base el SDK de .NET en la versión 3.1 etiquetandola como builder
FROM mcr.microsoft.com/dotnet/core/sdk:3.1 as builder

# Preapara el entorno de compilación de .NET
WORKDIR /Worker
COPY src/Worker/Worker.csproj .
RUN dotnet restore

# Copia el código desde el directorio actual al directorio de la aplicación dentro
    del contenedor
COPY src/Worker/ .
# Ejecuta la compilación del código en el entorno .NET
RUN dotnet publish -c Release -o /out Worker.csproj

# Usa como imagen el entorno de ejecución de .NET en la versión 3.1
FROM mcr.microsoft.com/dotnet/core/runtime:3.1

# Elige el directorio de la aplicación
WORKDIR /app

# Copia de la imagen etiquetada como "builder" el resultado de la compilación que se
     encuentra en el directorio /out a la imagen actual
COPY --from=builder /out .

# Se indica el comando que se ejecutará al arrancar el contenedor
CMD ["dotnet", "Worker.dll"]
```

5.2.3. Servicio Result

El servicio result es una webapp escrita en javascript que se utiliza para leer los datos de la base de datos y mostrarlos en tiempo real en un frontend.

En este fichero, vemos como al principio añade Tini, un *init* creado para ser ejecutado en contenedores. Este software permite que el comando *node server.js* se pueda ejecutar en el contenedor sin tener el PID 1. De esta forma, el control de los procesos hijos que pueda crear queda a cargo de Tini, en lugar de el comando *node server.js*. Así conseguimos que si el proceso principal muere por la finalización del contenedor, los procesos creados por este, no queden huérfanos, si no bajo el control de Tini.

Después de añadir Tini, copia todos los ficheros que cumplan la expresión regular *package*.json*. Estos contienen información sobre las dependencias usadas en el servicio y las versiones de las mismas. Al ejecutar el comando ”npm ci”se instalan las dependencias contenidas en los ficheros *package.json*. Estas se instalarán en el directorio *node_modules* del proyecto. Una

vez más, vemos como se ha estructurado el Dockerfile para aprovechar la caché de docker con los elementos que menos se modifican.

Por último, copia la aplicación y arranca Tini como *init* pasándole como argumento el comando que será su proceso hijo.

```
# Selecciona como base la imagen de node en su versión 10-slim
FROM node:10-slim

# Instala la herramienta curl
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Añade Tini, un "init" simple para contenedores que permite interpretar las señales
     del sistema correctamente
ENV TINI_VERSION v0.19.0
ADD https://github.com/krallin/tini/releases/download/${TINI_VERSION}/tini /tini
RUN chmod +x /tini

# Elige el directorio de la aplicación
WORKDIR /app

# Copia todos los ficheros que su nombre empiece por package y termine por .json al
    directorio de la aplicación
COPY package*.json ./

# Instala las dependencias necesarias para el proyecto (contenidas en los package*.
    json)
RUN npm ci \
 && npm cache clean --force \
 && mv /app/node_modules /node_modules

# Copia la aplicación
COPY . .

# Selecciona como variable de entorno PORT=80
ENV PORT 80

# Expone el puerto 80
EXPOSE 80

#Indica el comando que será ejecutado cuando se arranque el contenedor
CMD ["/tini", "--", "node", "server.js"]
```

5.2.4. Despliegue en local con docker-compose

Para conseguir desplegar estos servicios en conjunto con las opciones configuradas para que funcionen correctamente, se suele recurrir a docker-compose, ya que nos permite alma-

cenar esa configuración en un fichero. De esta manera podemos distribuir la configuración y reutilizarla una y otra vez con la seguridad de que siempre funcionará igual y con la comodidad de poder desplegarla con un solo comando docker-compose up.

Estos ficheros de configuración normalmente son utilizados en entornos locales de desarrollo. Permite levantar una aproximación de la infraestructura que encontraremos en el entorno de producción.

En este docker-compose encontramos tanto las imágenes que hemos definido para los servicios vote, worker y result, como las imágenes por defecto de los servicios redis y postgres. El nombre de las imágenes sirven para indicar la dirección del repositorio de imágenes en el que se almacenarán.

```
version: "3"

services:
  # Configuración del servicio vote
  vote:
    image: 192.168.1.5:5001/vote   #Indica el nombre de la imagen resultante
    build: ./vote       #Indica donde se encuentra el Dockerfile a construir
    volumes:
     - ./vote:/app      #Define un directorio que se compartirá entre el contenedor
    y el host
    ports:
      - "5002:80"       #Enlaza el puerto 80 del contenedor con el 5002 del host

  # Configuración del servicio redis
  redis:
    image: redis:alpine #Indica la imagen y la versión que usaremos para redis
    ports: ["6379"]     #

  # Configuración del servicio worker
  worker:
    image: 192.168.1.5:5001/worker #Indica el nombre de la imagen resultante
    build: ./worker     #Indica donde se encuentra el Dockerfile a construir

  # Configuración del servicio de base de datos
  db:
    image: postgres:9.4 #Indica la imagen y la versión que usaremos para redis
    environment:        #Define variables de entorno en el contenedor
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "postgres"

  # Configuración del servicio result
  result:
    image: 192.168.1.5:5001/result #Indica el nombre de la imagen resultante
    build: ./result     #Indica donde se encuentra el Dockerfile a construir
    volumes:
      - ./result:/app   #Define un directorio que se compartirá entre el contenedor
    y el host
```

```
    ports:
      - "5001:80"       #Enlaza el puerto 80 del contenedor con el 5001 del host
      - "5858:5858"     #Enlaza el puerto 5858 del contenedor con el 5858 del host
```

5.3. Almacenamiento de las imágenes docker

Para que las imágenes docker se puedan distribuir en distintos entornos y servidores sin importar su localización, se almacenan en un repositorio centralizado llamado registry. Este registry puede ser público, como dockerhub, o privado. En este caso, vamos a construir un repositorio privado desplegando un contenedor docker para ello. De esta manera, podremos subir y desplegar las imágenes docker sin problema. Para levantar el registry, solo tendremos que ejecutar el comando:

```
$ docker run -d -p 5000:5000 --name registry registry:2
```

Para subir las imágenes, hay que nombrarlas con la dirección en la que se encuentra el registry seguido del nombre de la imagen separados por una barra:

```
192.168.1.5:5000/imagename
```

Por último, las imágenes se subirán con el comando *push* del CLI de *docker* y el nombre que le hemos dado a la imagen.

```
$ docker push 192.168.1.5:5000/imagename
```

En el caso de que estas estén nombradas en un docker-compose, podemos subir todas las imágenes a la vez.

```
$ docker-compose push
```

Las imágenes con el mismo nombre se irán sobrescribiendo aprovechando la estructura en capas de docker para subir solamente las capas que se hayan actualizado y las siguientes. De esta forma, evita usar más ancho de banda del necesario y reduce el tiempo de subida de la imagen.

Para este caso en concreto, como estamos trabajando en un entorno local, se debe habilitar la subida insegura de imágenes al registry. Esto solo está indicado para casos de prueba en entornos aislados. Para tal propósito, debemos modificar o crear, si no existe, el archivo *daemon.json* en el directorio */etc/docker* añadiendole la siguiente linea con la IP en la que tenemos desplegado el contenedor del registry:

```
{
    "insecure-registries" : ["192.168.1.5:5000"]
}
```

Y a continuación, reiniciar el demonio de docker.

```
$ systemctl daemon-reload && systemctl restart docker
```

5.4. Diseño y despliegue de una aplicación en kubernetes

En entornos de producción, en los que se necesita que la aplicación pueda escalar adaptándose a la demanda, se suelen usar orquestadores de contenedores como plataforma sobre la que desplegar la aplicación. En este caso, desplegaremos los distintos servicios usando la herramienta de linea de comandos kubectl. Con ella desplegaremos sobre k0s nuestra aplicación.

Para el despliegue, se escribirá un archivo *.yaml* (Apéndice D) en el que irán definidas todas las estructuras que se crearán en el cluster que forma el orquestador de contenedores. Para cada servicio se define un Deployment, que será el encargado de generar los Pods necesarios y gestionar el número de réplicas de cada Pod a través de un ReplicaSet. También se define un Service, que se encargará de generar una capa transparente para poder acceder a los distintos Pods de manera indistinguible.

Para enlazar los Pods con los ReplicaSet y los Services, se utilizan los labels y los selectors de cada una de las partes de los documentos. Es importante que estas sean iguales, ya que si no, el cluster no podrá crear y enlazar los objetos correctamente.

5.4.1. Servicio Redis

El servicio redis tiene definido un Deployment con una sola réplica. El servicio expone el puerto 6379, el puerto por defecto de redis, y se construye con la imagen *redis:alpine* (Ver Apéndice D). Esta imagen, etiquetada como *alpine* se refiere a que es una imagen ligera basada en el sistema operativo con el mismo nombre. Este sistema operativo está pensado específicamente para contenedores y solo tiene lo indispensable para asegurar su funcionamiento. Este servicio también dispondrá de un objeto Service del tipo ClusterIP. Este tipo de Service solo permite que el servicio sea alcanzado desde el interior del cluster ya que este tipo solo permite que el servicio se alcanzado desde el interior del cluster. Para permitir que sea alcanzable, se

enlaza el puerto 6379 de las distintas instancias del Pod con el puerto 6379, en este caso, del Service.

5.4.2. Servicio Postgres

El Desployment del servicio postgres construye una sola réplica del servicio con la imagen *postgres:9.4*. También expone el puerto estándar de postgres, el 5432 y pasa unas variables de entorno al contenedor necesarias para configurar postgres correctamente. Estas variables son *POSTGRES_USER*, que se utiliza para configurar el usuario por defecto de la base de datos, *POSTGRES_PASSWORD* usada para indicar la contraseña del usuario mencionado anteriormente y*PGDATA* que se usa para seleccionar donde se guardan los archivos de configuración y datos necesarios para crear un cluster de bases de datos (Ver Apéndice D).

Al ser una base de datos, es deseable que los datos persistan y sean accesibles por todas las instancias de la base de datos. Por ello, se genera un objeto PersistentVolume en el que se define el tamaño, dónde estará guardado y los tipos de acceso, entre otros, de la estructura en la que se almacenarán los datos.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Para asociar este PersistentVolume con los Pods que genere el Deployment, hace falta generar una petición PersistentVolumeClaim. En esta petición se definirá cual es el tamaño deseado para el volumen y los permisos que se requieren. De esta manera se asignan los volumenes a los Pods de forma dinámica, esperando a que el volumen esté disponible para asociarlo.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pv-claim
spec:
  storageClassName: manual
```

```
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Para que el PersistentVolumeClaim sea llamado por el Deployment, hay que definir en este último un punto de montaje del volumen y que ese almacenamiento provenga del resultado de la petición.

```
...
        volumeMounts:
        - name: db-data
          mountPath: /var/lib/postgresql/data
      volumes:
      - name: db-data
        persistentVolumeClaim:
          claimName: postgres-pv-claim
```

Con el objetivo de que la base de datos sea alcanzable por otros elementos del cluster se genera un objeto Service de tipo ClusterIP para que solo sea accesible desde el interior del propio cluster (Ver Apéndice D). Esta exposición se realiza enlazando el puerto 5432 de las distintas instancias del servicio con el puerto 5432 del propio Service.

5.4.3. Servicio Vote

El Deployment del servicio Vote construye dos réplicas a partir de la imagen docker que se ha creado para este anteriormente (Figura 8). La descarga del registry y la despliega exponiendo el puerto 80.

El objeto Service es de tipo LoadBalancer, lo cual permite exponer el servicio para que sea alcanzable desde fuera del cluster. Para tal propósito, enlaza el puerto 80 de las distintas instancias, ya que es el puerto por defecto para peticiones HTTP, con el puerto 80 del objeto Service. De esta forma, las peticiones hechas al servicio a través del puerto 80 serán enviadas a alguna de las instancias a través de su puerto 80.

5.4.4. Servicio Worker

Para levantar el servicio Worker, usaremos la imagen docker que ya se ha construido y subido al registry, como podemos ver en la definición de su objeto Deployment (ver Apéndice D). De este servicio tan solo se levantará una réplica y no se expondrá ningún puerto, ya

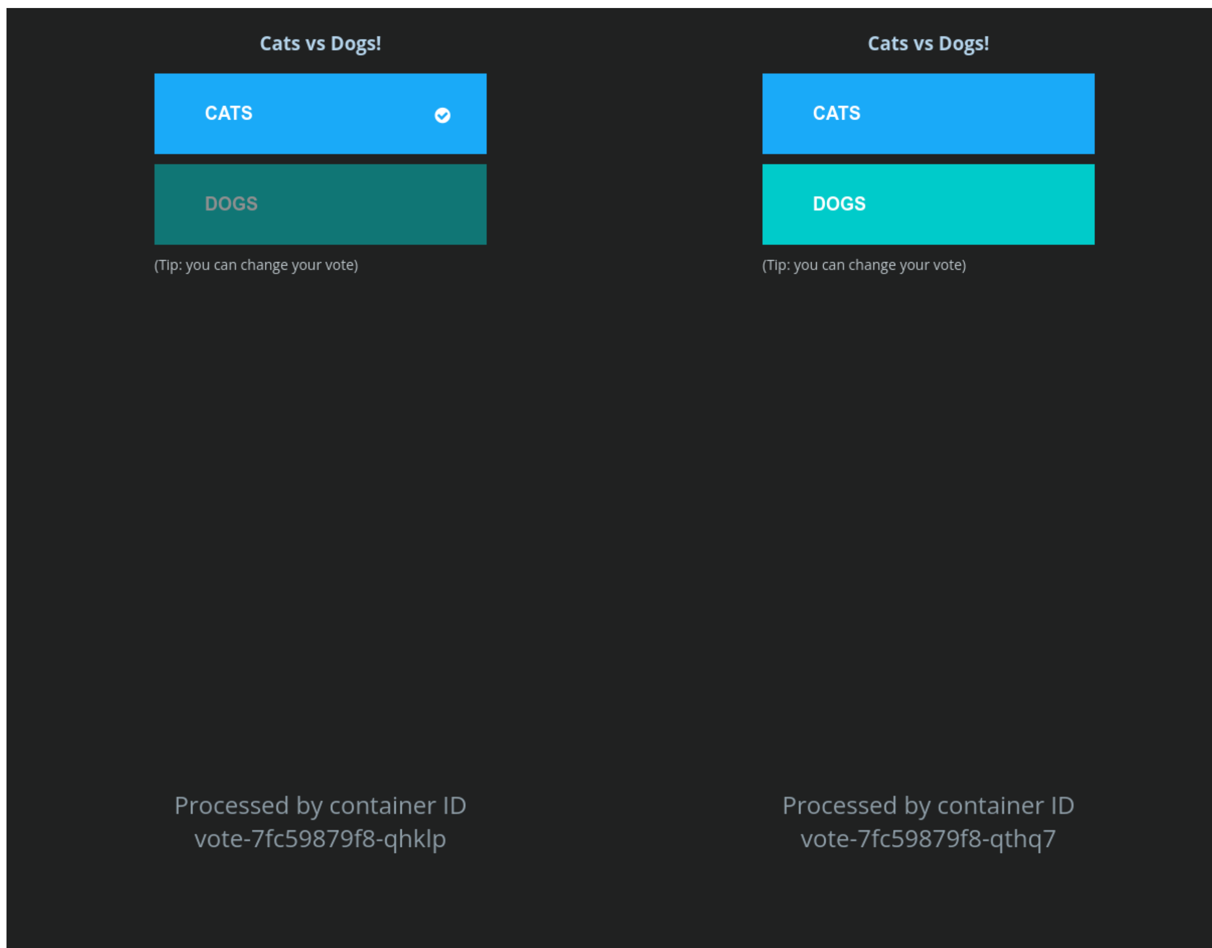

Figura 8: Servicio vote con distintos IDs

que será él el que hará las peticiones al redis para consumir objetos de la cola y las enviará a la base de datos, por lo cual, no espera recibir peticiones. Por tal motivo, tampoco necesita un Service, cada una de las réplicas trabaja de forma independiente.

5.4.5. Servicio Result

Para el servicio Result, se ha definido un Deployment que, con la imagen previamente creada para este servicio, levanta una réplica del mismo. Al ser un servicio al que se va a acceder a través de web, expone el puerto 80 para poder hacerle peticiones.

Por su parte, el objeto Service es de tipo LoadBalancer para permitir que se realicen peticiones desde el exterior del cluster. Este enlaza el puerto 80 de las instancias del servicio con el puerto 80 del objeto Service y así, poder alcanzar los distintos Pods del servicio a través de

ese puerto.

5.4.6. LoadBalancers en local

En un entorno cloud, al desplegar un Service del tipo LoadBalancer el propio entorno proporciona un Balanceador de Carga a través del cual se puede acceder. Sin embargo, en entornos bare metal hay que instalar en kubernetes un balanceador de carga adecuado (Figura 9).

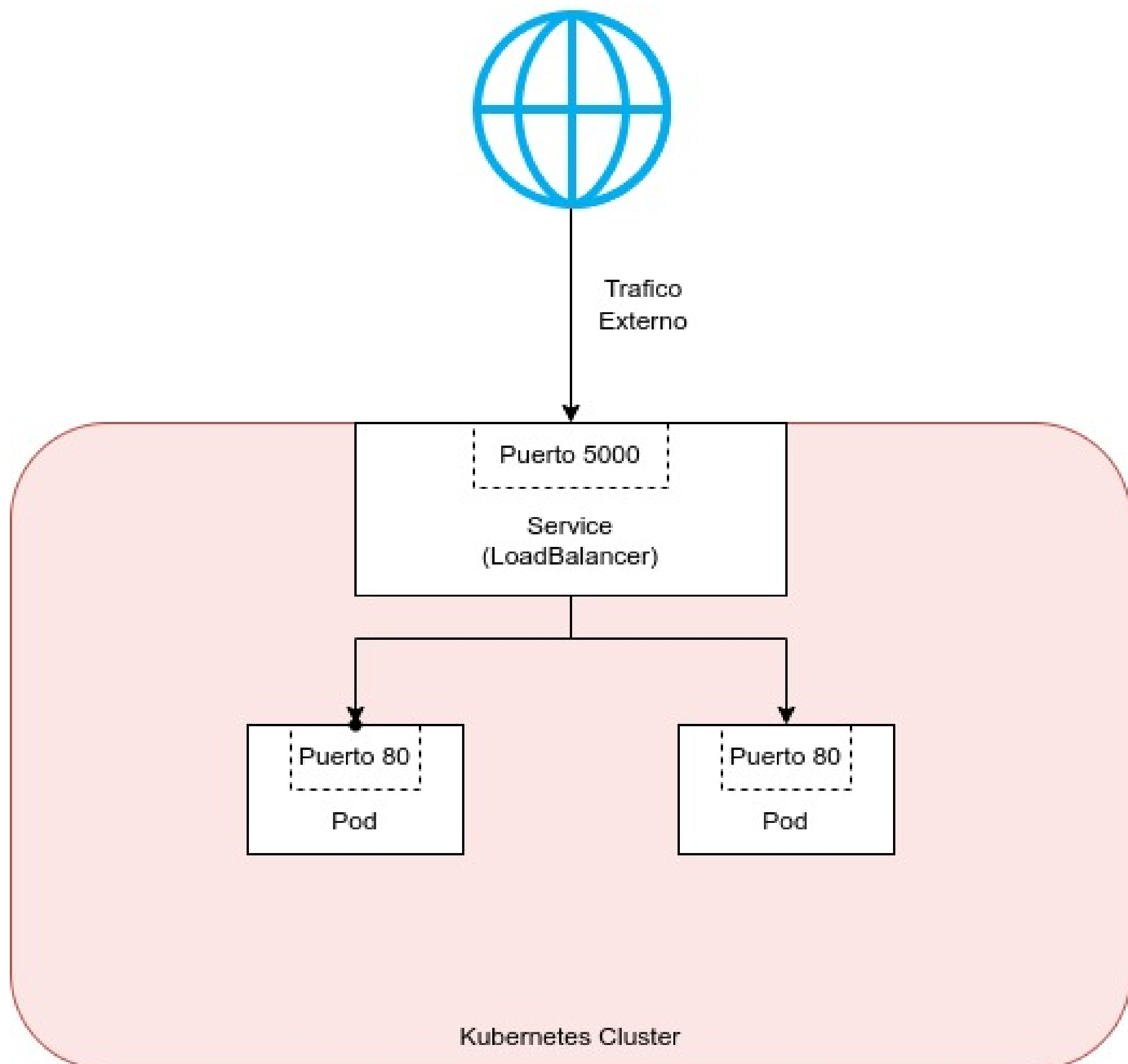

Figura 9: Diagrama de funcionamiento de un Balanceador de carga en kubernetes

En este caso, se ha usado MetalLB por ser un balanceador de carga preparado para tal fin

y con mucho uso. Se ha instalado con Helm, el instalador de paquetes para kubernetes más usado y que ya viene integrado en K0s. Para ello, hay que acceder a la configuración del cluster mediante el comando

```
$ sudo ./k0sctl config edit
```

Una vez abierto ese archivo de configuración, para instalar MetalLB, hay que añadir las siguientes lineas que indican el repositorio y el paquete que se va a instalar tal y como se muestra a continuación.

```
helm:
      repositories:
      - name: metallb
        url: https://metallb.github.io/metallb
      charts:
      - name: metallb
        chartname: metallb/metallb
        namespace: metallb
```

Una vez instalado, habrá que configurar cual será el conjunto de IPs que podrán usar los balanceadores de carga mediante un objeto definido en la propia API de MetalLB llamado IPAddressPool y configurar la publicación de servicios en estas IPs a través de la capa de red de kubernetes, llamada layer2. Para tal fin, se aplicará al cluster la siguiente configuración con el comando kubectl apply usando el siguiente archivo

```
---
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: first-pool
  namespace: metallb
spec:
  addresses:
  - 192.168.1.100-192.168.1.150
---
#L2Advertisement, describing the intention to advertise an IP coming from an
    IPAddressPool via L2
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: example
  namespace: metallb
```

5.5. Diseño de un pipeline en CircleCI

Una vez se ha definido como se construirá la aplicación en el cluster de kubernetes, lo óptimo es automatizar el despliegue, dejando así el control sobre los mismos al propio equipo

de desarrollo.

Para tal finalidad, definiremos en cada rama del repositorio de Git, los pasos para testear, construir y desplegar los servicios en el entorno adecuado.

En este caso, como repositorio, usaremos GitHub y solo trabajaremos con una rama llamada circleci-project-setup. Al subir el código a esta rama, y solo a esta rama, automáticamente comenzará un proceso automático dividido en tres partes conocidas como jobs.

El primero de ellos se ha nombrado como *build-push-images*. Construirá las imágenes docker y las subirá a un registry. El segundo, llamado *install-kubectl* instalará kubectl, la herramienta para gestionar kubernetes que nos permitirá desplegar los contenedores en el cluster. Y por último, el tercer job, llamado deploy, desplegará los contenedores en el cluster de kubernetes tal y como los hemos definido en los archivos de deployment.

De esta forma conseguimos que el despliegue sea desatendido por el equipo de operaciones y que solo actúen en caso de error. Para tal propósito, se suelen configurar alertas por correo electrónico o servicios de mensajería instantánea internos que avisan al equipo de operaciones en caso de fallo.

En este caso, ejecutaremos este pipeline en un runner alojado en un servidor local y llamado *kirtashkill/test*. El motivo principal para hacerlo de esta forma es que toda la infraestructura está montada en local, por lo cual, usando esta funcionalidad, no tendremos que dar acceso a CircleCI desde fuera.

```
# Use the latest 2.1 version of CircleCI pipeline process engine.
# See: https://circleci.com/docs/2.0/configuration-reference
version: 2.1

# Se definen distintos jobs que luego son invocados más adelante, en este caso, los
   jobs son build-push-images, install-kubectl y deploy
jobs:
    # Este job construye las imágenes de docker y las sube a un registry de docker
  build-push-images:
    # Define en que recurso, en este caso en el Self-Hosted Runner, se ejecutará el
    job
    resource_class: kirtashkill/test
    # Escoge que imagen se usará para ejecutar el runner
    docker:
      - image: cimg/base:edge
    # Describe los pasos del job
    steps:
      - checkout
      - run:
          name: "build-images"
          command: "docker-compose build && \
```

```
                    docker-compose push"


  # Este job instala kubectl, la herramienta para gestionar kubernetes, en el runner
  install-kubectl:
    # Define en que recurso, en este caso en el Self-Hosted Runner, se ejecutará el
    job
    resource_class: kirtashkill/test
    # Escoge que imagen se usará para ejecutar el runner
    docker:
      - image: cimg/base:edge
    # Describe los pasos del job
    steps:
      - checkout
      - run:
          name: "Install kubectl"
          command: "sudo curl -LO \"https://dl.k8s.io/release/$(curl -L -s https://
    dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl\" && \
                    sudo curl -LO \"https://dl.k8s.io/$(curl -L -s https://dl.k8s.io
    /release/stable.txt)/bin/linux/amd64/kubectl.sha256\" && \
                    echo \"$(cat kubectl.sha256)  kubectl\" | sha256sum --check && \
                    sudo chmod +x kubectl && sudo mkdir -p ~/.local/bin && sudo mv
    ./kubectl ~/.local/bin/kubectl && \
                    kubectl version --client"


  # Este job despliega la aplicación en kubernetes
  deploy:
    # Define en que recurso, en este caso en el Self-Hosted Runner, se ejecutará el
    job
    resource_class: kirtashkill/test
    # Escoge que imagen se usará para ejecutar el runner
    docker:
      - image: cimg/base:edge
    # Describe los pasos del job
    steps:
      - checkout
      - run:
          name: "Deploy in k8s"
          command: "kubectl --kubeconfig ./kubeconfig apply -f kube-deployment.yaml"

# Invoke jobs via workflows
# See: https://circleci.com/docs/2.0/configuration-reference/#workflows
workflows:
  say-hello-workflow:
    jobs:
      - build-push-images:
          filters:
            branches:
              only: circleci-project-setup
      - install-kubectl:
          filters:
            branches:
              only: circleci-project-setup
```

```
    - deploy:
        filters:
          branches:
            only: circleci-project-setup
        requires:
          - install-kubectl
          - build-push-images
```

Una vez ejecutado el pipeline obtendremos una salida de éxito o de fallo en el pipelines completo y en cada uno de los jobs. Lo cual nos indicará, que partes del proceso se han completado satisfactoriamente y cuales no. Esto nos permitirá ejecutar de nuevo solo las partes del pipeline que hayan fallado (Figura 10), en caso de ser necesario.

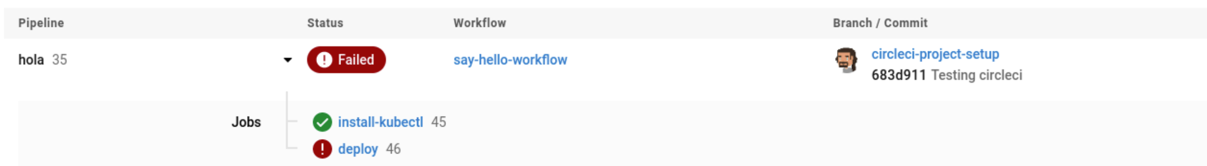

Figura 10: Salida de un pipeline con fallos en uno de los jobs

También obtenemos como resultado la salida de texto de los distintos comandos que hemos ejecutado (Figura 11). Esto nos dará información que nos permitirá saber, en caso de fallo, que ha funcionado mal y como podemos solucionarlo (Figura 12).

Install kubectl 9s

```
#!/bin/bash -eo pipefail
sudo curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl" && sudo curl -LO "https://dl.k8s.io/$(curl -L -s https://dl.k8s

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   138  100   138    0     0    669      0 --:--:-- --:--:-- --:--:--   669
100 45.7M  100 45.7M    0     0  8164k      0  0:00:05  0:00:05 --:--:-- 9242k
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   138  100   138    0     0    641      0 --:--:-- --:--:-- --:--:--   641
100    64  100    64    0     0    188      0 --:--:-- --:--:-- --:--:--   571
kubectl: OK
WARNING: This version information is deprecated and will be replaced with the output from kubectl version --short.  Use --output=yaml|json to get the full version.
Client Version: version.Info{Major:"1", Minor:"24", GitVersion:"v1.24.3", GitCommit:"aef86a93758dc3cb2c658dd9657ab4ad4afc21cb", GitTreeState:"clean", BuildDate:"2022-07-13T14:30:4
Kustomize Version: v4.5.4
CircleCI received exit code 0
```

Figura 11: Salida sin fallos de un job de CircleCI

5.6. Configuración de mecanismos de observabilidad sobre k0s

Cuando ya se han desplegado los servicios en el cluster, es deseable poder observar su comportamiento y detectar rápidamente si hay fallos en estos. Para ello se utilizan distintos

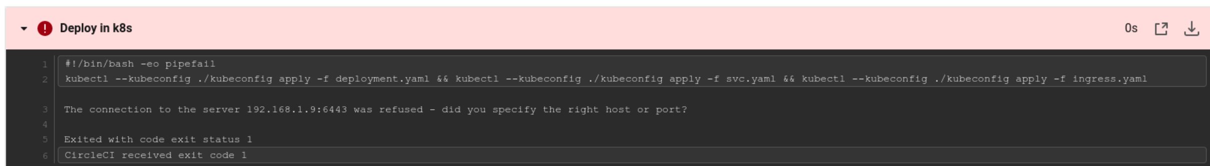

Figura 12: Salida con fallos de un job de CircleCI

mecanismos y herramientas que permitirán esa vigilancia de una manera cómoda y sin tener que acceder al propio cluster. Para tal finalidad, se recogerán logs, métricas y trazas de los distintos servicios y del cluster y se mostrarán en distintos paneles gráficos. De esta manera podrán ser consultados tanto por cualquiera aunque no tenga conocimientos sobre la obtención de esos datos.

Para el despliegue de estas herramientas se usará de Helm, un gestor de paquetes para kubernetes [26]. Este ofrece paquetes configurados en sus repositorios, llamados charts, para desplegarlos en clusters reduciendo la problemática de la configuración para la mayoría de los casos y permitiendo hacer una configuración a medida en aquellos en los que sea necesaria.

5.6.1. Logs

Los logs, o registros, muestran información en texto ordenada cronológicamente sobre los distintos acontecimientos que ocurren en el servicio. Estos serán recogidos por Filebeat.

Filebeat es un recolector de logs que instalaremos en nuestro cluster de kubernetes. Este nos permitirá enviarlos a una base de datos optimizada para indexación y búsqueda llamada Elasticsearch. Una vez en esta base de datos, podremos visualizarlos en un panel, filtrarlos y agruparlos por tipos y fecha. El panel que se suele utilizar para mostrar esos logs se llama Kibana y está preparado para la integración con Elasticsearch y Filebeat, lo que facilita mucho su uso.

En este caso en concreto, solo se ha instalado en el cluster Filebeat, ya que Elasticsearch y Kibana son elementos más pesados y no es necesario que estén instalados dentro del cluster, ya que FIlebeat es el único de los tres elementos que recoge los datos. Por ello han sido levantados fuera del cluster mediante contenedores docker.

Para instalar Filebeat hay que añadir el repositorio en el que se encuentra con el comando

```
$ helm repo add elastic https://helm.elastic.co
```

Una vez añadido el repositorio, hay que instalar Filebeat. Filebeat viene con una configuración por defecto que hay que personalizar para añadir la dirección en la que se encuentra Elasticsearch para enviarle los logs recolectados. Estos cambios los realizaremos modificando el archivo values.yaml que podemos encontrar en el repositorio de github asociado al chart de Helm de Filebeat (https://github.com/elastic/helm-charts/tree/main/filebeat). Este cambio en concreto, se realizará mediante la adición de una variable de entorno de la siguiente forma:

```
    - name: "ELASTICSEARCH_HOSTS"
      value: "192.168.1.15"
```

Además de esto, también hay que definir una objeto Secret a través de un fichero *secret.yaml* en el que se encuentran el usuario y la contraseña con la que se accederá a Elasticsearch convertidos a base64. Este archivo tiene la siguiente forma:

```
apiVersion: v1
kind: Secret
metadata:
  name: elasticsearch-master-credentials
data:
  username: dXNlcg==
  password: Y2hhbmdlbWU=
```

Y se añadirá al cluster con el comando

```
$ kubectl apply -f secret.yaml
```

Una vez preparado el archivo *values.yaml* y añadidos el usuario y la contraseña de Elasticsearch al cluster de kubernetes, se procede a la instalación de la herramienta mediante el comando

```
$ helm install filebeat elastic/filebeat -f values.yaml
```

Usando la opción f"para sobrescribir la configuración por defecto de Filebeat.

Una vez instalado Filebeat, habrá que levantar Elasticsearch y Kibana de manera externa al cluster. En este caso se han levantado con un archivo *docker-compose.yaml* (Apéndice E) en el que se ha definido como desplegar los servicios.

Elastisearch enlaza los puertos 9200 y 9300 del contenedor con los mismos del host, pasa a una configuración en la que se define el número de nodos para usar, los plugins instalados, el nombre del cluster de nodos, aunque en este caso sea solo uno, y la dirección del host, en este caso 0.0.0.0 para usar la dirección local. También se enlaza un volumen en el que se almacenarán los datos persistentes de Elasticsearch para que no exista perdida de datos en caso de reinicio del contenedor.

Kibana, por su parte, enlaza el puerto 5601 al mismo puerto del host y pasa un archivo de configuración a través de un volumen en el que se da nombre al servidor de Kibana, se define su dirección, nuevamente 0.0.0.0 para usar la dirección loca, y se define, la dirección, el usuario y la contraseña que se usarán para acceder a Elasticsearch.

Una vez desplegados los tres servicios, habrá que definir varios usuarios en Elasticsearch para garantizar el acceso tanto de Filebeat como de Kibana. Existen varias formas de crear usuarios en Elasticsearch, en este caso se ha usado la herramienta de linea de comandos .elasticsearch-usersïncluida en el contenedor de Elasticsearch. Se definirán varios usuarios dependiendo del uso. Para Filebeat se definirá uno con el rol "beats_system", para Kibana uno con el rol "kibana_systemz para acceder al panel de administración uno con el rol "kibana_admin". El comando que se utiliza para definirlos es

```
$ elasticsearch-users useradd username -p password -r [roles]
```

Por último, hay que acceder al panel de administración de Kibana con el usuario y contraseña definidos para ello y añadir a la visualización la expresión regular "filebeat-*"que encaja con los índices con el que Filebeat envía los logs por defecto. Después de este proceso, ya se estarán recibiendo y visualizando los logs del cluster en el panel de administración de Kibana. En él se pueden filtrar, agrupar y ver de una forma más clara los logs (Figura 13).

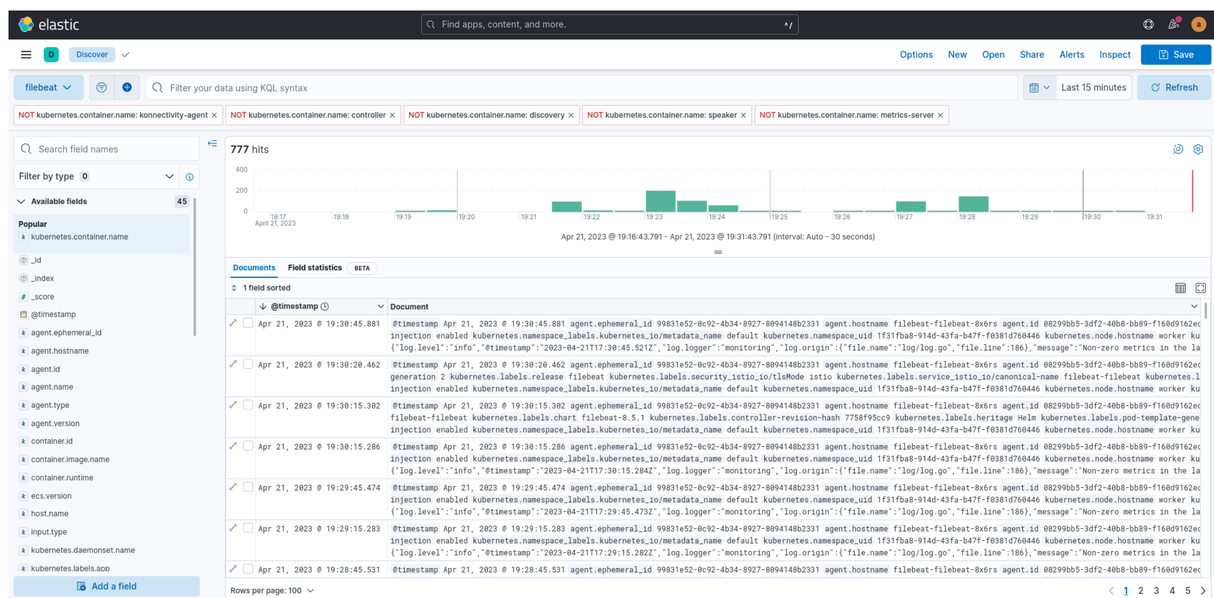

Figura 13: Muestra del panel con los logs

5.6.2. Métricas

Las métricas muestran la información relativa los recursos del cluster. Mediante ellas podemos monitorear la cantidad de CPU usada, la cantidad de RAM, el tráfico de red…

En este caso, las métricas serán recogidas por node-exporter, un servicio que permite recolectar las métricas y que sean accesibles desde el exterior. El resto del sistema de métricas estará montado fuera del cluster para reducir la carga en el mismo.

Esta parte externa, estará formada por una base de datos optimizada para series temporales llamada Prometheus y un panel de visualización llamado Grafana, los que nos ayudarán a visualizar los datos exportados por node-exporter.

Para instalar node-exporter con Helm, hay que añadir el repositorio *pometheus-comunity* mediante el comando

```
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-
    charts
```

e instalar el chart correspondiente con el comando

```
$ helm install prometheus-node-exporter prometheus-community/prometheus-node-
    exporter
```

De esta forma expondrá la información sobre las métricas a través del puerto 9100 del nodo del cluster en el que está instalado.

Para desplegar la parte del sistema externa al cluster, usaremos docker-compose y la definición de los servicios ser hará en un archivo *docker-compose.yaml* (Apéndice F).

La definición de Prometheus incluye un volumen anónimo en el que guardar los archivos persistentes, un volumen enlazado con el host para pasar la configuración, en la que se define la dirección de node-exporter, y un enlace entre los puertos 9090 del contenedor y el mismo del host.

Por su parte, Grafana define un volumen anónimo para permitir persistencia en los datos, la contraseña para entrar la primera vez al panel de administración y un enlace entre los puertos 3000 del contenedor y del host. Este enlace se usará para acceder al panel de administración a través del puerto 3000. La primera vez que entremos al panel de administración de Grafana, podremos usar la contraseña que hayamos definido en la definición del servicio de docker-compose, pero nos invitará a cambiarla después del primer login.

Una vez dentro, tendremos que definir una fuente de datos, en este caso, la base de datos

Prometheus que hemos levantado anteriormente. Para ello se añadirá una nueva fuente de datos en la configuración (Figura 14).

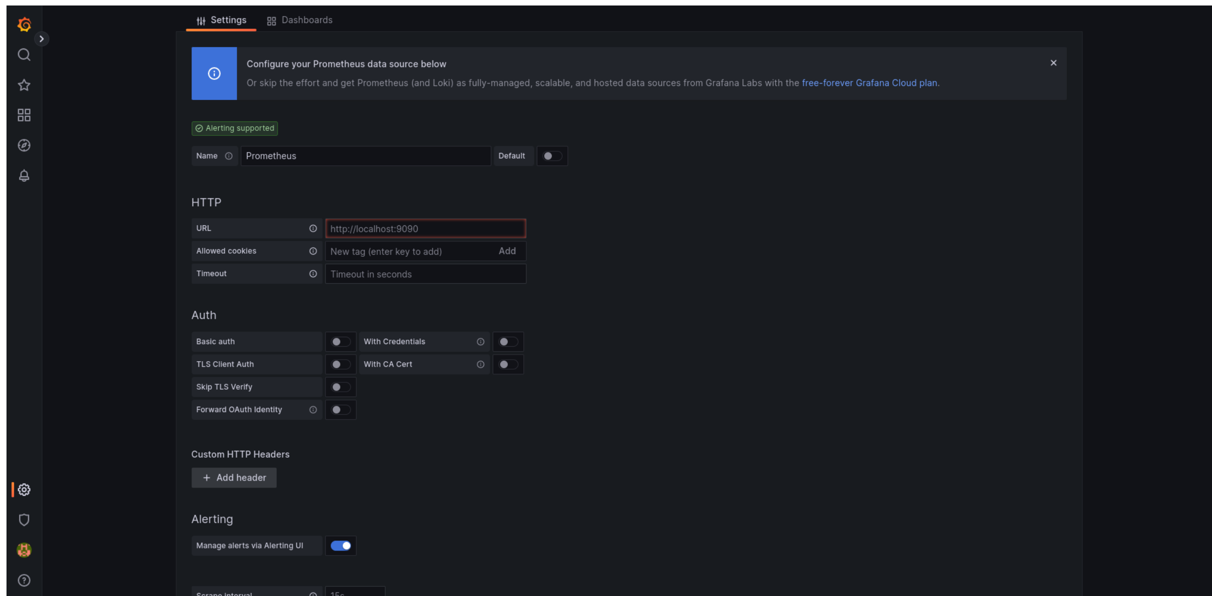

Figura 14: Addición de fuentes de datos a Grafana

Una vez configurada la fuente de datos, podemos añadir un panel de visualización para los datos recolectados. Estos paneles de visualización se pueden definir por uno mismo diseñando querys para obtener los datos de Prometheus o descargar uno de los miles ya definidos por la comunidad. En la mayoría de los casos, podemos encontrar un panel de visualización adecuado para lo que necesitamos e importarlo en Grafana (Figura 15).

Para este caso concreto, se ha importado un panel preparado para representar todos los datos recogidos por node-exporter (https://grafana.com/grafana/dashboards/1860-node-exporter-full/).

5.6.3. Service Mesh

Service Mesh, o malla de servicios, es una práctica de arquitectura para administrar y visualizar conjuntos de múltiples microservicios basados en contenedores.

Un service mesh proporciona servicios de vigilancia, escalabilidad y alta disponibilidad, a través de una API de servicios, en lugar de obligar su implementación en cada microservicio [27].

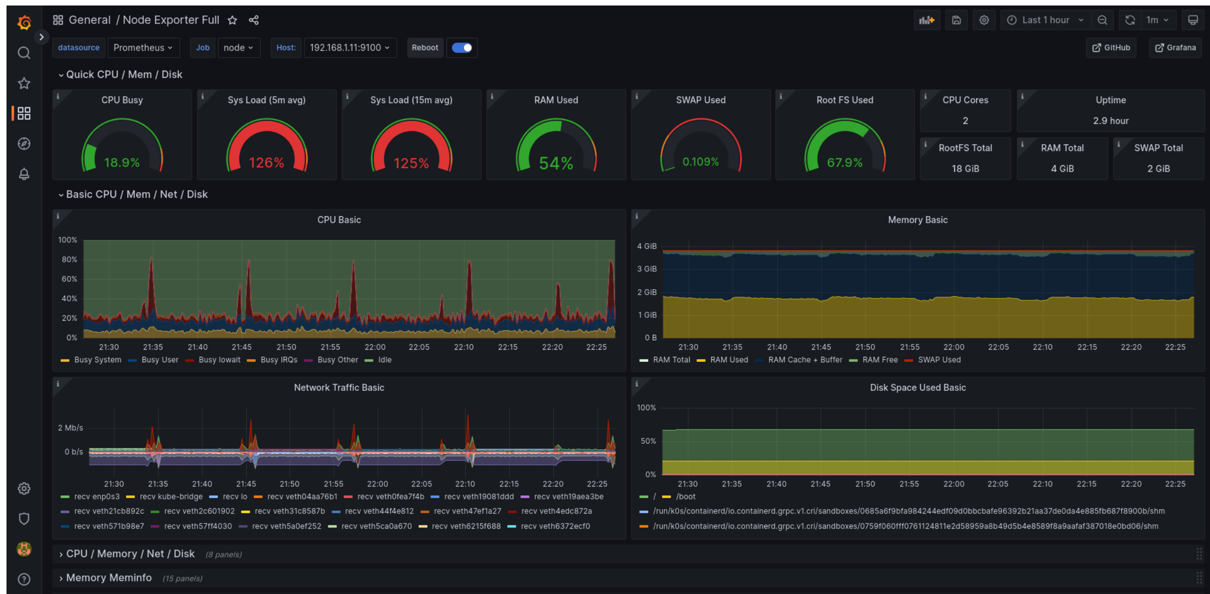

Figura 15: Grafana Dashboard Panel

La service mesh de este cluster ha sido construida con Istio. Istio, construye la malla de servicios inyectando en cada Pod del cluster un contenedor como proxy que se encarga de filtrar todo el tráfico del Pod y monitorear los recursos del Pod. Este patrón de construcción se llama sidecar (Figura 16). A parte de Istio, para que la funcionalidad de la service mesh sea completa y los datos recogidos por la misma sean mostrados y procesados, hay que instalar como add-on los servicios Prometheus y Jaeger. El primero permite almacenar series temporales, como hemos visto anteriormente, para poder monitorear la salud de Istio y de las aplicaciones en la malla de servicios [28]. Jaeger, por su parte, recoge y analiza las peticiones y las respuestas de cada uno de los Pods para obtener información sobre el tráfico de la aplicación [29]. Para visualizar los datos recogidos por Istio, se usa el panel de visualización Kiali. Todos los elementos estarán instalados en el cluster para mejorar la integración entre ambos servicios.

Para instalar tanto Istio como Kiali, se ha usado Helm. Para Istio, hay que añadir el repositorio en el que están alojados los charts del mismo mediante el comando

```
$ helm repo add istio https://istio-release.storage.googleapis.com/charts
```

y crear un namespace para este sistema llamado ïstio-systemçon el comando

```
$ kubectl create namespace istio-system
```

Una vez realizados estos pasos, se puede proceder a instalar los dos elementos principales

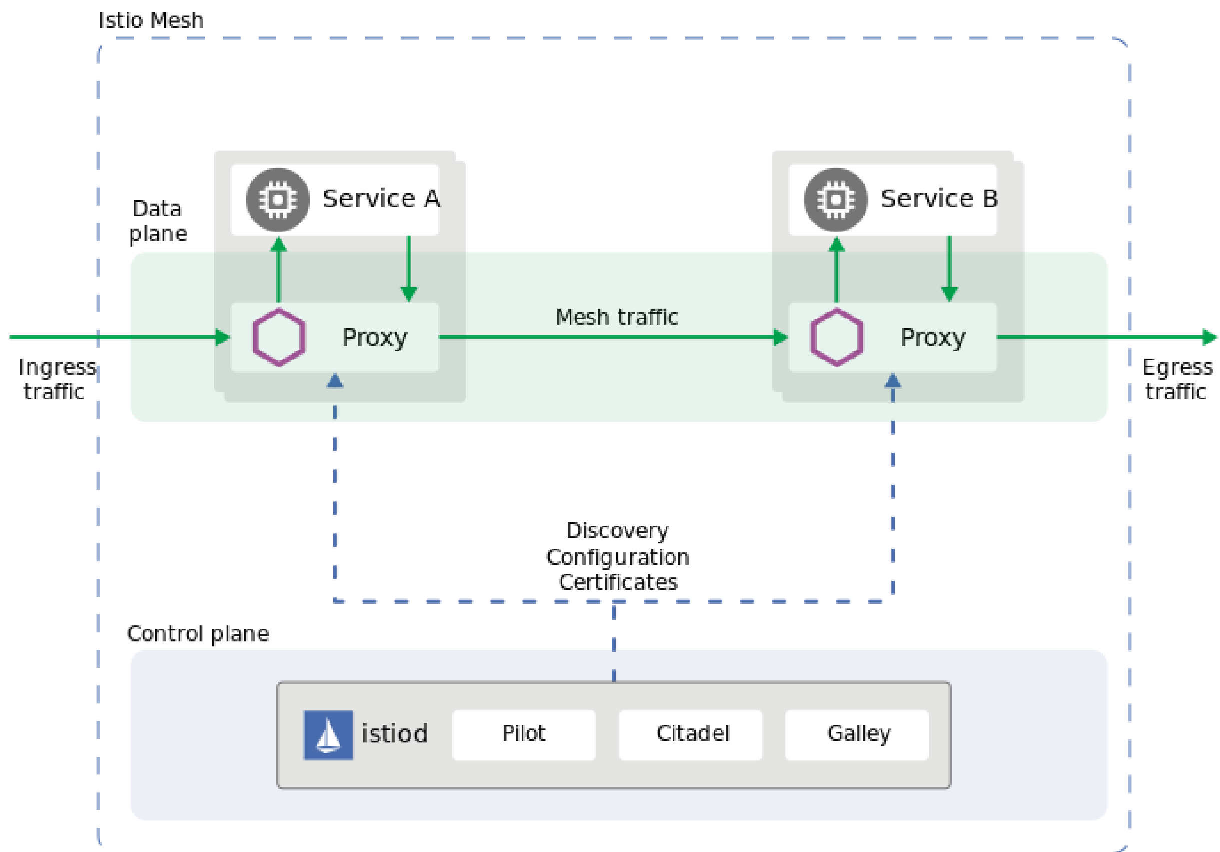

Figura 16: Istio Mesh

de Istio, Istio-base e Istiod.

Istio-base contiene las definiciones de recursos personalizadas (CRDs) necesarias para desplegar los proxies en los distintos Pods. Para instalarlo, hay que ejecutar el comando ”helm install istio-base istio/base -n istio-system̈. Por su parte, istiod proporciona los mecanismos necesarios para descubrir donde se sitúan los distintos servicios desplegados y no perder la referencia a estos en el caso de que cambien. Para instalarlo, hay que ejecutar el comando

```
$ helm install istiod istio/istiod -n istio-system
```

Cuando hayan sido instalados estos elementos de Istio, hay que indicar en que Pods se desean desplegar los sidecars. Para ello, elegiremos el namespace o namespaces en los que estén desplegados los distintos servicios que queremos incluir en la service mesh, en este caso el namespace se llama default. Se añadirá a todos los Pods de este namespace un label llamado istio-injection para indicar a Istio que se debe inyectar los sidecars en esos Pods. Para ello, se ejecuta el comando ”kubectl label namespace default istio-injection=enabledz, a posteriori, se

reiniciarán los Pods para que se puedan recrear con el sidecar inyectado.

Para poder sacar el máximo partido a Istio, se instalarán como add-on los servicios Prometheus y Jaeger con los comandos

```
$ kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.17/
    samples/addons/prometheus.yaml
```

y

```
$ kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.17/
    samples/addons/jaeger.yaml
```

respectivamente. Al tener que integrarse con Istio, no se distribuyen para tal finalidad a través de Helm, si no que son los propios desarrolladores de Istio los que proveen a los usuarios de un archivo para poder desplegar cada uno de las herramientas con todas sus dependencias preconfiguradas.

Para poder visualizar los datos recogidos por Istio hay que instalar Kiali, el panel de visualización asociado a Istio. La instalación se hará a través de Helm, añadiendo el repositorio de la aplicación con el comando

```
$ helm repo add kiali https://kiali.org/helm-charts
```

e instalándolo mediante el comando:

```
helm install \
    --set cr.create=true \
    --set cr.namespace=istio-system \
    --namespace kiali-operator \
    --create-namespace \
    kiali-operator \
    kiali/kiali-operator
```

Una vez instalados todos los elementos que componen la service mesh, habrá que acceder a Kiali. Para ello se hará un reenvío de las peticiones hechas al puerto 20001 de nuestro host al puerto 20001 del servicio de Kiali. Para tal finalidad usaremos el comando

```
$ kubectl port-forward deployment/kiali 20001:20001 -n istio-system
```

pudiendo de esta manera acceder a Kiali a través de la dirección ”localhost:20001”del host en el que está alojado Kiali.

Al acceder a Kiali por primera vez, nos pedirá una autenticación. Esta autenticación es un token provisto por Istio mediante el comando ”kubectl -n istio-system create token kiali-service-account”. Habrá que copiar ese token y pegarlo en la web de visualización de Kiali. Una vez hecho esto, ya se podrá consultar el estado de la service mesh de forma correcta (Figura 17).

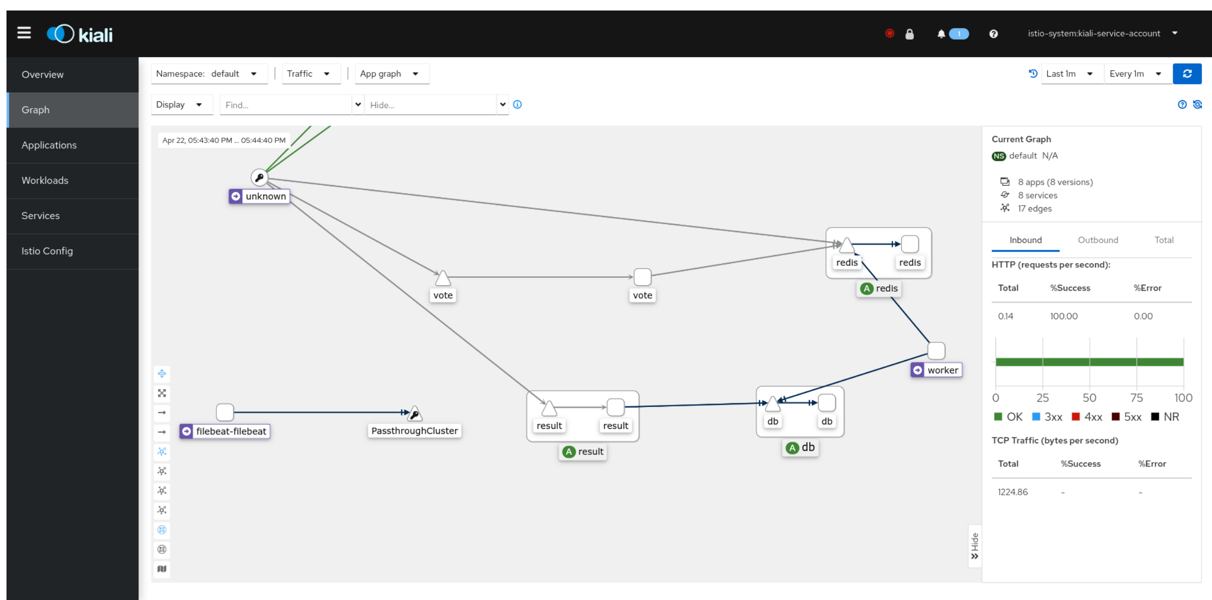

Figura 17: Kiali Graph

6
Conclusiones

En este trabajo, se ha podido ver una muestra de la potencia de diversas tecnologías y metodologías que se usan actualmente para desplegar código. Los contenedores son una buena herramienta tanto para evitar errores en los despliegues propiciados por la incompatibilidad de entornos, como para realizar escalado y desescalado rápidamente. Aunque en ocasiones, los contenedores puedan añadir cierta complejidad al desarrollo, los beneficios anteriormente comentados, han llevado a que su uso sea masivo. Por otra parte, los contenedores suelen hacer más sencillas las configuraciones del CI/CD, ya que el propio contenedor se ocupa de gran parte de la ejecución del código.

Durante la realización de este trabajo, se han ido probando las herramientas en las arquitecturas x86_64 y ARM64. En la mayoría de los casos, la arquitectura del procesador es transparente para las herramientas. Sin embargo, k0s no permite instalar el nodo controller en ARM64, como explican en su documentación en github [30].

La complejidad de la instalación, gestión y mantenimiento del cluster pone en valor la facilidad de uso de las plataformas cloud. Estas nos brindan una variedad de herramientas muy completas que ahorran tiempo y dificultades, lo que explica que su uso sea tan popular en empresas. Por otro lado, el hecho de que sea un sistema tan complejo, lo hace innecesario para aplicaciones muy sencillas. En las plataformas cloud podemos encontrar soluciones que se adapten mejor a las necesidades del producto sin un exceso de carga como el que requiere kubernetes y su monitorización.

Como lineas futuras para este proyecto, se podría configurar el escalado automático en kubernetes, llamado Horizontal Pod Autoscaler(HPA) y crear scripts con Ansible o Terraform para aprovisionar máquinas y aumentar de forma automatizada la cantidad de nodos en el cluster.

Referencias

[1] *El móvil, líder en el consumo de internet.* Think with Google. URL: https://www.thinkwithgoogle.com/intl/es-es/insights/tendencias-de-consumo/el-m%5C%C3%5C%B3vil-l%5C%C3%5C%ADder-en-el-consumo-de-internet/.

[2] *¿Qué es la metodología ágil?* URL: https://www.redhat.com/es/devops/what-is-agile-methodology (visitado 23-05-2023).

[3] *Virtualización.* URL: https://www.redhat.com/es/topics/virtualization.

[4] *Qué es una máquina virtual y cómo funciona | Microsoft Azure.* URL: https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-a-virtual-machine.

[5] *Contenedores, sistemas de contenerización y orquestadores | Stackscale.* Section: Sistemas. 9 de dic. de 2021. URL: https://www.stackscale.com/es/blog/contenerizacion-orquestacion-contenedor/.

[6] *Chapter 16. Jails.* FreeBSD Documentation Portal. URL: https://docs.freebsd.org/en/books/handbook/jails/.

[7] *Linux Containers.* URL: https://linuxcontainers.org/.

[8] *Open Container Initiative - Open Container Initiative.* URL: https://opencontainers.org/.

[9] Joakim Verona. *Practical Devops.* Packt, feb. de 2016. 240 págs. ISBN: 978-1-78588-287-6.

[10] *cgroups(7) - Linux manual page.* URL: https://man7.org/linux/man-pages/man7/cgroups.7.html.

[11] *namespaces(7) - Linux manual page.* URL: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[12] *What Is Scalability? - Definition from SearchDataCenter.com.* Data Center. URL: https://www.techtarget.com/searchdatacenter/definition/scalability.

[13] *Alta disponibilidad - ¿Qué es y cómo funciona?* URL: https://ciberseguridad.com/guias/alta-disponibilidad/#%5C%C2%5C%BFQue%5C_es%5C_la%5C_alta%5C_disponibilidad.

[14] *Observability*. Observability. URL: https://www.observability.com.

[15] Roshni Y. *What is Observability in Control System? Kalman's Test and Condition for observability in s-plane*. Electronics Coach. 4 de mayo de 2020. URL: https://electronicscoach.com/observability.html.

[16] Samuel. *The 3 Pillars of System Observability: Logs, Metrics, and Tracing*. IOD - The Content Engineers. 15 de dic. de 2020. URL: https://iamondemand.com/blog/the-3-pillars-of-system-observability-logs-metrics-and-tracing/.

[17] *¿Qué es la malla de servicios o service mesh?* URL: https://www.redhat.com/es/topics/microservices/what-is-a-service-mesh.

[18] *¿Qué es DevOps y para qué sirve? | NetApp*. URL: https://www.netapp.com/es/devops-solutions/what-is-devops/.

[19] *La integración y la distribución continuas (CI/CD)*. URL: https://www.redhat.com/es/topics/devops/what-is-ci-cd.

[20] *Pods*. Kubernetes. Section: docs. URL: https://kubernetes.io/es/docs/concepts/workloads/pods/pod/.

[21] *MetalLB, bare metal load-balancer for Kubernetes*. URL: https://metallb.universe.tf/.

[22] *Sensu | An Introduction to Prometheus Monitoring (2021)*. URL: https://sensu.io/blog/introduction-to-prometheus-monitoring.

[23] *Grafana OSS | Metrics, logs, traces, and more*. Grafana Labs. URL: https://grafana.com/oss/grafana/.

[24] *El ELK Stack: de los creadores de Elasticsearch*. URL: https://www.elastic.co/es/what-is/elk-stack.

[25] *¿Qué es Istio?* URL: https://www.redhat.com/es/topics/microservices/what-is-istio.

[26] *Helm*. URL: https://helm.sh/.

[27] *Service Mesh - Arquitectura de Microservicios*. Aplyca Tecnología SAS. 4 de oct. de 2022. URL: https://www.aplyca.com/blog/service-mesh-arquitectura-de-microservicios.

[28] 5 Minute Read. *Prometheus*. Istio. URL: https://istio.io/latest/docs/ops/integrations/prometheus/.

[29] *Jaeger*. Istio. URL: https://istio.io/latest/docs/ops/integrations/jaeger/.

[30] *k0s - The Zero Friction Kubernetes by Team Lens*. original-date: 2020-06-10T10:44:09Z. URL: https://github.com/k0sproject/k0s/blob/8f351b3887bf80480c041121d5d1353f94dda4df/docs/troubleshooting.md.

Apéndice A
Manual de Instalación de docker

Para instalar el motor de contenedores docker en un sistema basado en debian con apt instalado, tanto en arquitectura x86_64 como en ARM64, usaremos la guia oficial de instalación. En ella se detalla en unos pocos pasos el proceso de instalación y la comprobación de que la herramienta, ha sido correctamente instalada.

Primero, actualizaremos actualiza la lista de paquetes disponibles y sus versiones con el comando:

```
$ sudo apt update
```

Después, instalaremos las dependencias necesarias, en caso de no tenerlas instaladas previamente:

```
$ sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

A continuación, crearemos un directorio para la clave GPG de docker para comprobar la integridad del software antes de instalarlo y la descargaremos:

```
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /
   etc/apt/keyrings/docker.gpg
```

Una vez hayamos realizado estos pasos, podremos añadir el repositorio de docker a nuestra lista de repositorios con el siguiente comando:

```
$ echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
   https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/
   null
```

Una vez tenemos el sistema preparado, comenzaremos con la instalación del motor de docker. Para ello, debemos actualizar la lista de paquetes de nuevo e instalar los paquetes necesarios para la correcta ejecución de contenedores docker. Junto con estos paquetes, vamos a instalar también el plugin de docker-compose.

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

Finalmente, para comprobar que nuestra instalación se ha realizado de la forma adecuada, ejecutaremos un comando simple para levantar un contenedor.

```
$ sudo docker run hello-world
```

Si este último comando nos devuelve la siguiente salida, tendremos la seguridad de que el motor de contenedores docker ha sido correctamente instalado:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Apéndice B
Manual de Instalación de k0s

Existen diferentes formas de instalación de K0s. Se puede instalar en tan solo un nodo, útil para hacer pruebas, o en varios nodos. Dentro de la instalación en varios nodos, podemos encontrar dos formas, podemos instalarlo de forma manual, configurando cada uno de los nodos y controllers independientemente o con la herramienta que proveen para desplegar el cluster llamada *k0sctl*. Procederemos a explicar la instalación de forma manual en varios nodos y con la herramienta *k0sctl*.

B.1. Instalación manual

Para iniciar la instalación manual, debemos descargar y ejecutar el script de instalación de k0s que ofrecen en su web *https://get.k0s.sh*:

```
$ curl -sSLf https://get.k0s.sh | sudo sh
```

Una vez instalado, generaremos el archivo de configuración en el que vendrá definida la configuración del nodo controller. Esta es personalizable dependiendo de las necesidades de uso.

```
$ mkdir -p /etc/k0s
$ k0s config create > /etc/k0s/k0s.yaml
```

Con esta configuración creada, podemos instalar un nodo controller y arrancarlo para posteriormente ir añadiendo nodos worker. De esta forma, podemos arrancar tantos nodos controller como queramos:

```
$ sudo k0s install controller -c /etc/k0s/k0s.yaml
$ sudo k0s start
```

Para añadir nodos de tipo worker al cluster necesitamos generar un token que contiene la información necesaria para que haya una conexión con confianza mutua entre el worker y el/los controller y almacenar ese token en un archivo. Para ello ejecutaremos:

```
$ k0s token create --role=worker > token-file
```

Por último, en la máquina en la que queramos ejecutar un nodo de tipo worker tenemos que descargar e instalar el script de instalación de k0s, instalar un nodo worker y arrancarlo.

```
$ curl -sSLf https://get.k0s.sh | sudo sh
$ sudo k0s install worker --token-file /path/to/token/file
$ sudo k0s start
```

Para gestionar el cluster, podemos usar, desde el nodo controller, la herramienta para gestionar clusters de kubernetes *kubectl*, que viene incluida en el binario de *k0s*, de la siguiente forma

```
$ sudo k0s kubectl [command] [flags]
```

B.2. Instalación con k0sctl

K0sctl es una herramienta escrita en Golang que automatiza el proceso de instalación y administración de un cluster k0s.

Como está escrita en Golang, primero debemos descargar e instalar el entorno del lenguaje para ejecutarlo. Por último comprobaremos que podemos invocar el comando *go* y la versión que hemos instalado.

```
$ wget https://go.dev/dl/go1.19.linux-amd64.tar.gz
$ tar -xf go1.19.linux-amd64.tar.gz
$ sudo mv go/ /usr/local/
$ export PATH=$PATH:/usr/local/go/bin
$ go version
```

Una vez instalado el entorno de Golang, solo tenemos ejecutar el comando para instalar k0sctl y tendremos la herramienta instalada.

```
$ go install github.com/k0sproject/k0sctl@latest
```

Habiendo instalado ya la herramienta, podemos ejecutar el comando para generar la configuración inicial del cluster.

```
$ k0sctl init > k0sctl.yaml
```

Al abrir el archivo de configuración *k0sctl.yaml* encontraremos algo así:

```
apiVersion: k0sctl.k0sproject.io/v1beta1
kind: Cluster
metadata:
  name: k0s-cluster
spec:
  hosts:
  - ssh:
      address: 10.0.0.1
```

```
      user: root
      port: 22
      keyPath: ~/.ssh/id_rsa
    role: controller
  - ssh:
      address: 10.0.0.2
      user: root
      port: 22
      keyPath: ~/.ssh/id_rsa
    role: worker
  k0s:
    version: 1.24.3+k0s.0
    dynamicConfig: false
```

Para desplegar los nodos del cluster necesitaremos sustituir dirección IP de las máquinas en las que queramos desplegarlos. Para averiguarla podemos usar el comando:

```
$ ifconfig
```

También tendremos que dejar acceder a la herramienta mediante SSH como usuario *root* autenticado mediante una clave privada. Para ello, tendremos que configurar en las máquinas en las que vamos a desplegar los nodos las reglas de acceso a través de SSH. Para ello modificaremos el fichero */etc/ssh/sshd_config* modificando las siguientes opciones para que queden así:

```
PermitRootLogin yes
AuthorizedKeysFile      .ssh/authorized_keys
```

Tendremos que generar para el usuario *root*, un par clave pública clave privada que almacenaremos y añadiremos la clave pública al fichero *.ssh/authorized_keys*

```
$ ssh-keygen -b 2048 -t rsa
$ cat id_rsa.pub >> ~/.ssh/authorized_keys
```

Por último, copiaremos la clave privada a la máquina en la que vamos a ejecutar *k0sctl*. En nuestro caso, vamos a utilizar para identificar es máquina la IP 192.168.1.5 con el usuario ruben

```
$ ssh-copy-id ruben@192.168.1.5
```

Una vez configurada la conexión SSH y modificado el archivo de configuración para cambiar las IPs de los nodos por las que vamos a usar, podemos ejecutar el comando para empezar a desplegar el cluster con la configuración que tenemos en el fichero *k0sctl.yaml*

```
$ k0sctl apply --config k0sctl.yaml
```

Cuando este proceso haya terminado, necesitamos obtener la configuración necesaria para conectarse al cluster.

```
$ k0sctl kubeconfig > kubeconfig
```

Antes de poder conectarnos al cluster, necesitamos instalar la herramienta para gestionar clusters de kubernetes, *kubectl*. Para ello instalaremos las dependencias necesarias y descargaremos la clave pública GPG para comprobar la integridad del software antes de instalarlo, añadiremos el repositorio de kubectl y finalmente lo instalaremos.

```
$ sudo apt-get update
$ sudo apt-get install -y ca-certificates curl apt-transport-https
$ sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://
    packages.cloud.google.com/apt/doc/apt-key.gpg
$ echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https://
    apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/
    kubernetes.list
$ sudo apt-get update
$ sudo apt-get install -y kubectl
```

Y finalmente, ya podremos conectarnos al cluster pasándole como argumento del flag –*kubeconfig* el archivo kubeconfig que hemos obtenido en un paso anterior.

```
$ kubectl [commands] --kubeconfig kubeconfig [flags]
```

Apéndice C
Manual de Instalación de un runner de Circle CI

Para usar Circle CI, lo primero que tenemos que hacer es crear una cuenta en su plataforma. Una vez creada la cuenta, tenemos que conectarla con nuestra cuenta de un repositorio distribuido de git, del cual va a obtener el código para ejecutar los *pipelines* que configuremos.

Para no vulnerar la seguridad de la red en la que se ejecuta el *pipeline*, nuestro sistema sigue el modo de funcionamiento que vemos en el diagrama de la siguiente figura.

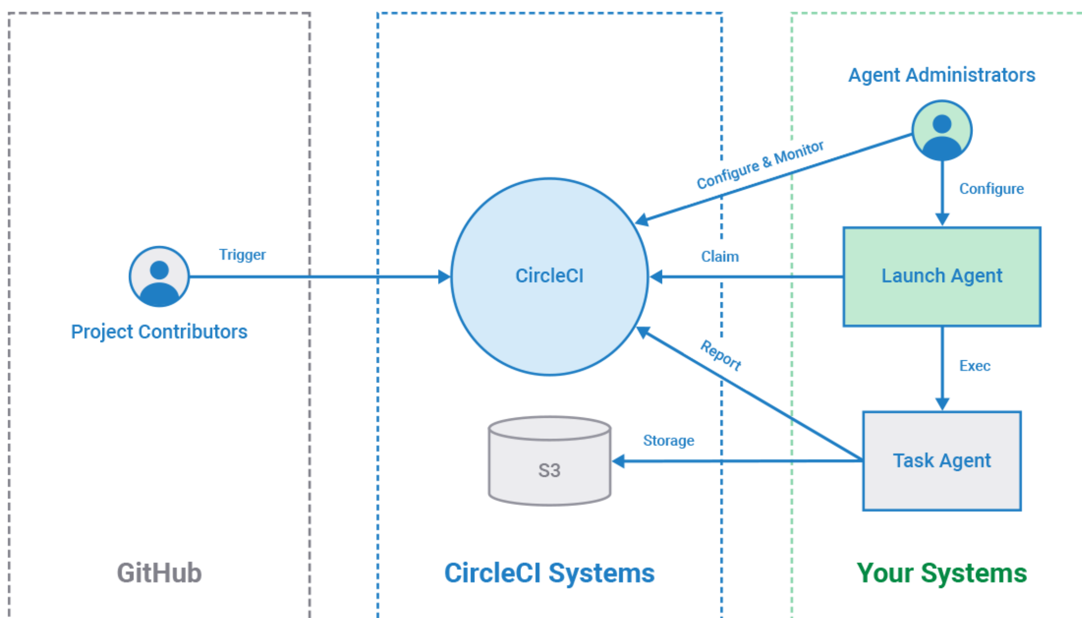

Figura 18: Diagrama de funcionamiento de un Self-hosted runner

Un *Launch Agent* comprueba si hay alguna tarea que tenga que ser ejecutada, si es así, la recibe como respuesta de CircleCI y la ejecuta en un *Task Agent* que después de ejecutar la tarea, envía un informe de la ejecución a CircleCI para que podamos verla en el panel de administración del mismo.

Una vez hemos realizado este registro y conexión iniciales, tenemos que acceder al panel de administración de Circle CI, e ir a la sección *Self-Hosted Runners* que se encuentra en la columna de la izquierda y crear una nueva clase de recurso, así es como se denomina a los grupos de *Self-Hosted Runners*, a la que llamaremos *test.* Al crear esta clase de recurso, sólo tenemos que nombrarlo. Una vez le hemos dado nombre al recurso, procedemos a ejecutar runners asociados a esa clase de recurso.

En nuestro caso, crearemos el *runner* usando un contenedor docker, por un lado para mostrar la versatilidad de los contenedores docker y por otro, por la sencillez a la hora de desplegarlo en distintos entornos y arquitecturas. Para tal propósito, crearemos un Dockerfile, en el que usamos como imagen base la imagen oficial de un runner de CircleCI, actualizaremos la lista de paquetes e instalaremos python3.

```
FROM circleci/runner:launch-agent
RUN sudo apt-get update; \
sudo apt-get install --no-install-recommends -y python3
```

A este archivo lo llamaremos *Dockerfile.runner.extended.* A partir de este archivo, construiremos la imágen Docker.

```
$ sudo docker build --tag runner-image:latest --file ./Dockerfile.runner.extended .
```

Por último, para ejecutar el contenedor, tendremos que pasar como argumento una variable de entorno con el token de autenticación proporcionado por CircleCI al crear la clase de recurso.

```
$ sudo CIRCLECI_API_TOKEN=[AUTH_TOKEN] docker run --env CIRCLECI_API_TOKEN --name
   runner runner-image:latest
```

Apéndice D
Despliegue en Kubernetes

```yaml
# redis
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis
  name: redis
spec:
  clusterIP: None
  ports:
  - name: redis-service
    port: 6379
    targetPort: 6379
  selector:
    app: redis
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
  labels:
    app: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:alpine
        ports:
        - containerPort: 6379
          name: redis

# db
---
```

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: db
  name: db
spec:
  clusterIP: None
  ports:
  - name: db
    port: 5432
    targetPort: 5432
  selector:
    app: db
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db
  labels:
    app: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
      - name: db
        image: postgres:9.4
        env:
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata
        - name: POSTGRES_USER
          value: postgres
        - name: POSTGRES_PASSWORD
          value: postgres
        ports:
        - containerPort: 5432
          name: db
        volumeMounts:
        - name: db-data
          mountPath: /var/lib/postgresql/data
      volumes:
      - name: db-data
        persistentVolumeClaim:
          claimName: postgres-pv-claim

---
apiVersion: v1
```

```yaml
kind: PersistentVolumeClaim
metadata:
  name: postgres-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
# result
---
apiVersion: v1
kind: Service
metadata:
  name: result
  labels:
    app: result
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: result

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: result
  labels:
    app: result
spec:
  replicas: 1
  selector:
    matchLabels:
      app: result
```

```yaml
  template:
    metadata:
      labels:
        app: result
    spec:
      containers:
      - name: result
        image: dockersamples/examplevotingapp_result:before
        ports:
        - containerPort: 80
          name: result

# vote
---
apiVersion: v1
kind: Service
metadata:
  name: vote
  labels:
    apps: vote
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 80
      name: vote-service
  selector:
    app: vote
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vote
  labels:
    app: vote
spec:
  replicas: 2
  selector:
    matchLabels:
      app: vote
  template:
    metadata:
      labels:
        app: vote
    spec:
      containers:
      - name: vote
        image: dockersamples/examplevotingapp_vote:before
        ports:
        - containerPort: 80
          name: vote

# worker
---
```

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    apps: worker
  name: worker
spec:
  clusterIP: None
  selector:
    app: worker
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: worker
  name: worker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: worker
  template:
    metadata:
      labels:
        app: worker
    spec:
      containers:
      - image: dockersamples/examplevotingapp_worker
        name: worker
```

Apéndice E
Despliegue Elasticsearch y Kibana con docker-compose

```
version: '3.2'

services:
  elasticsearch:
    image: elasticsearch:8.6.2
    volumes:
      - ./elasticsearch/config/elasticsearch.yml:/usr/share/elasticsearch/config/
    elasticsearch.yml:ro
      - ./persist:/usr/share/elasticsearch/data
    ports:
      - "9200:9200"
      - "9300:9300"
    environment:
      ES_JAVA_OPTS: "-Xmx256m -Xms256m"
      ELASTIC_PASSWORD: changeme
    networks:
      - elk

  kibana:
    image: kibana:8.6.2
    volumes:
      - ./kibana/config/kibana.yml:/usr/share/kibana/config/kibana.yml:ro
    ports:
      - "5601:5601"
    networks:
      - elk
    depends_on:
      - elasticsearch


networks:
  elk:
    driver: bridge
```

Apéndice F
Despliegue Prometheus y Grafana con docker-compose

```yaml
version: '3.2'

volumes:
    prometheus_data: {}
    grafana_data: {}

services:

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus:/etc/prometheus
      - prometheus_data:/prometheus
    command:
      - "--config.file=/etc/prometheus/prometheus.yaml"
    ports:
      - 9090:9090

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    volumes:
      - grafana_data:/var/lib/grafana
    ports:
      - 3000:3000
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
```

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga