Universidad
Rey Juan Carlos

Escuela Técnica Superior en Ingeniería Informática

Implementación de auto escalado en un producto basado en microservicios

Memoria del Trabajo Fin de grado

Ingeniería Informática

Autor:

Jaime Oñate Rodríguez-Pardo

Tutor: Maria Teresa González De Lena

Mayo 2024

Resumen

En el ámbito profesional de la informática, muchas de las aplicaciones, programas y servicios, tanto internos como en línea están diseñados con una arquitectura modular. Esta arquitectura se caracteriza por la división de los diferentes componentes en módulos separados, cada uno encargado de realizar funciones específicas. Esto aporta ventajas como la facilidad para aportar nuevos componentes o módulos (escalabilidad). Los módulos pueden ser añadidos o actualizados de manera independiente, permitiendo que el sistema crezca y se adapte a nuevas necesidades sin afectar significativamente a los componentes existentes. Esta arquitectura también mejorar el mantenimiento, ya que es posible incorporar nuevas funcionalidades o mejorar las existentes de forma ágil, respondiendo rápidamente a las demandas del mercado o del entorno operativo.

En este contexto, la **escalabilidad automática** emerge como una estrategia adecuada para optimizar el rendimiento de los productos software modulares. La capacidad de adaptarse dinámicamente a cambios en la carga de trabajo permite maximizar la eficiencia operativa y mejorar la experiencia del usuario.

En este trabajo, se ha presentado una solución para abordar este desafío. Se ha implementado un servicio de auto escalado que aprovecha la combinación de las herramientas Prometheus y Grafana para la recolección y visualización de métricas de rendimiento. Además, se ha diseñado un servicio que utiliza estas métricas para lograr la gestión automática del escalado de los componentes necesitados.

Para desplegar estos servicios con sus componentes, se van a emplear ciertas herramientas y tecnologías. En concreto, se ha trabajado con Docker, que es un gestor de contenedores. Se ha empleado esta tecnología de contenedor para encapsular los componentes y modular una aplicación. Además, se ha empleado el concepto de imagen para garantizar consistencia en el rendimiento de los componentes de cada prueba.

Índice general

Índice de tablas

Índice de figuras

Algoritmos

Capítulo 1

Introducción

En este capítulo se hablará de la motivación para desarrollar este trabajo de fin de grado. Después se introducirán las razones y soluciones para desarrollar un servicio de auto escalado para un producto software implementado de manera modular.

1.1. Motivación

En ingeniería informática y en ingeniería de computadores, se estudian materias muy valoradas por las empresas: programación, redes, sistemas operativos, bases de datos, entre otras. Las razones por las que estas materias tienen tanta estima se pueden resumir de la siguiente manera:

- Por un lado, la dificultad y contenido que presentan estas asignaturas supone que los estudiantes que las han superado han adquirido cierto nivel académico que garantiza su calidad profesional.
- Por otro lado, las competencias ganadas suelen servir a las empresas para encontrar una labor útil de estos estudiantes en ellas. Por ejemplo, los estudiantes que aprueban bases de datos tienen un nivel mínimo para trabajar con la información almacenada de una empresa a nivel profesional (Ver referencia [9]).

Una de las asignaturas que más me llamó la atención fue **sistemas distribuidos**. La razón se debe a que pude aprender el desarrollo de un prototipo de un producto comercial. Por ejemplo, una página web con un servicio incorporado. En el doble grado se estudian sistemas distribuidos y desarrollo de aplicaciones distribuidas. En estas asignaturas se aprende el conocimiento necesario para desarrollar una aplicación tanto externa como interna con prestaciones como registro de usuario, venta de un producto, despliegue de programas y búsqueda y almacenamiento de información.

Este conocimiento es esencial para las empresas que desean crecer y suelen reinvertir beneficios en el desarrollo de sus propios servicios. Las empresas necesitan estas aplicaciones tanto para mantenerse y gestionarse como para complementar el producto/servicio que ofrezcan al mercado (Ver referencia [2]).

La habilidad para mantener, arreglar y mejorar estas aplicaciones se vuelve esencial, y una opción atractiva como camino profesional. De hecho, en este trabajo de fin de titulación

quedará reflejada la experiencia profesional acumulada al haber tomado este rol en la empresa. Como se ha señalado, el desarrollo de aplicaciones distribuidas está presente en las empresas que alcanzan cierto tamaño. De la misma manera que las empresas crecen, lo hacen estas aplicaciones, tanto en tamaño como en complejidad. Es por ello por lo que producir estas aplicaciones de manera óptima es esencial.

1.2. Tipos de arquitecturas

Una posible clasificación de aplicaciones es según su tipo de arquitectura. Por un lado, las arquitecturas monolíticas, y, por otro lado, las arquitecturas modulares. Las **arquitecturas monolíticas** se caracterizan porque las funcionalidades y componentes de la aplicación están acoplados y dependen unos de otros. Esto presenta varias ventajas:

- Simplicidad de diseño e implementación, ya que no requiere el diseño y la gestión de interfaces entre módulos.
- Mayor rendimiento, porque no habría recursos adicionales destinados a las comunicaciones y despliegue de módulos, ni cuellos de botella.
- Menos costes, debido a emplear menos recursos.

Sin embargo, las aplicaciones que generalmente se emplean se caracterizan por presentar una **arquitectura modular**, y las ventajas que presentan son las razones por las que se utilizan:

- La principal de todas es la escalabilidad, en las arquitecturas modulares al estar los componentes desacoplados, presentan la capacidad para sustituirse o modificarse sin afectar el resto de la arquitectura o afectándola parcialmente.
- Este desacoplamiento también favorece la flexibilidad, donde gracias a la modularidad, se pueden cambiar los componentes que hagan falta para modificar el producto según las necesidades del usuario final. También favorece la adaptabilidad, necesaria para afrontar cambios en las tecnologías empleadas y para afrontar actualizaciones.
- Aunque estas son las principales razones para usar una arquitectura modular, también facilitan el mantenimiento y la reutilización de código.

Vistos estos tipos de arquitecturas, en los productos de software también podemos distinguir dos tipos de aplicaciones según el usuario final. Las aplicaciones cuyo usuario final las emplea con fines laborales, y las aplicaciones cuyo usuario final es un consumidor. Esta distinción es importante para entender diferentes necesidades de escalado según la aplicación.

Las aplicaciones cuyos usuarios las utilizan para desempeñar su trabajo no suelen requerir tanto escalado automático como otras. Pero el escalado que necesitan es diverso, por ejemplo, en desarrollo e investigación puede ser necesario añadir escalado vertical (que añade CPU o RAM a una arquitectura que no varía) o escalado horizontal donde la arquitectura crecería recibiendo nuevos servidores e instancias de recursos para distribuir

la carga de trabajo. Esta decisión puede ser tomada por el usuario, que por ejemplo elige añadir recursos para un procesamiento. O puede ser automático, y crecer según métricas predefinidas para la autogestión del rendimiento.

Esta última política de escalado es la que se analizará en este trabajo. El escalado automático de aplicaciones se emplea en aplicaciones web cara al público cuyo usuario final es el cliente. Esto se debe a que recibir peticiones cliente de la web pública supone una situación impredecible, como eventos de marketing, lanzamientos de productos, campañas promocionales o eventos en vivo.

1.3. Objetivos

Vamos a definir los objetivos del trabajo y luego presentar en el próximo capítulo el estado de la tecnología respecto a este tema. El objetivo principal del trabajo es presentar una manera realista de escalar automáticamente una aplicación en el presente:

- Será necesario una aplicación web funcional e interactiva con necesidad potencial de escalado.
- Sera necesaria una implementación modular.
- Se requerirán métricas que permitan analizar el rendimiento de los módulos.
- Se precisará un comportamiento de escalado automático según el valor de las métricas procesadas.

1.4. Estructura de la memoria

Este trabajo de fin de grado se estructura en los siguientes capítulos:

1. – **Introducción**: donde se explica la motivación del trabajo, los distintos tipos de arquitecturas, los objetivos y la estructura.
2. – **Estado del arte**: Se exponen los distintos tipos de tecnologías que se pueden utilizar. Se realiza una investigación de las imágenes en Docker
3. – **Análisis de Requisitos**: donde se enumeran los requisitos funcionales y no funcionales y se explica la metodología.
4. – **Diseño e implementación**: Se realiza una exposición de los elementos en los contenedores, se explica el funcionamiento de cada servicio y el flujo de trabajo y diseño del sistema de auto escalado.
5. – **Resultados**: donde se explican las mediciones y se valora el rendimiento y comportamiento final del producto.
6. – **Conclusiones y trabajo futuro**: qué se ha cumplido, aprendido y qué se podría hacer en trabajos posteriores

Capítulo 2

Estado del arte

La modulación de los componentes en una arquitectura software está estandarizado como una manera profesional de implementarlos. Antes de diseñar una arquitectura modulada (apartado 3 requisitos) realizaremos unos estudios de mercado para elegir nuestra tecnología de implementación. Hay diferentes maneras de lograr modularidad de los componentes, la principal es la de microservicios ya que es de las más prominentes en el presente (Ver referencia [5]).

2.1. Microservicios y contenedores

Los **microservicios** son servicios ligeros que se pueden desarrollar, desplegar y escalar de forma independiente, lo que facilita la modularidad y la colaboración entre equipos de desarrollo. Para la comunicación entre los microservicios se implementan interfaces y se utiliza una API accesible al menos en la red en la que estén los componentes.

A pesar de sus ventajas, los microservicios también presentan desafíos, como la complejidad en la gestión del despliegue. Por eso una manera muy extendida de trabajar es empleando contenedores.

Un **contenedor** es un entorno de ejecución que proporciona al servicio aislamiento a nivel de sistema operativo otorgándole su propio sistema de archivos, procesos, espacio de red y recursos (Ver referencia [4]). Este aislamiento también supone una encapsulación de las herramientas, bibliotecas y dependencias que utilicen.

Los contenedores ofrecen una ejecución consistente y tienen algunas similitudes con las máquinas virtuales, pero también importantes diferencias y ventajas.

Ejecución Consistente

- Los contenedores aíslan las aplicaciones y sus dependencias del sistema operativo subyacente, asegurando que funcionen de manera consistente en diferentes entornos, desde el desarrollo hasta la producción.
- Las aplicaciones en contenedores se empaquetan en archivos de imagen que incluyen todo lo necesario para ejecutarse, lo que garantiza que la aplicación se ejecute de la misma manera independientemente del entorno.

Comparación con Máquinas Virtuales

Similitudes

- **Aislamiento**: Tanto los contenedores como las máquinas virtuales proporcionan aislamiento entre aplicaciones y el entorno subyacente.
- **Portabilidad**: Ambas tecnologías permiten mover aplicaciones entre diferentes entornos de manera relativamente sencilla.

Diferencias y Ventajas de los Contenedores

- **Ligereza**: Los contenedores comparten el mismo sistema operativo del host, lo que reduce significativamente el uso de recursos en comparación con las máquinas virtuales que requieren un sistema operativo completo.
- **Velocidad de Inicio**: Los contenedores se inician mucho más rápido que las máquinas virtuales porque no necesitan arrancar un sistema operativo completo.
- **Eficiencia de Recursos**: Debido a que no requieren un sistema operativo completo para cada instancia, los contenedores utilizan menos memoria y CPU, permitiendo ejecutar más contenedores en el mismo hardware que las máquinas virtuales.

Desventajas de los Contenedores

- **Seguridad**: Aunque el aislamiento de los contenedores es bueno, no es tan robusto como el de las máquinas virtuales. Los contenedores comparten el mismo kernel del sistema operativo, lo que puede ser una preocupación de seguridad.
- **Persistencia de Datos**: Los contenedores están diseñados para ser efímeros, lo que puede complicar la persistencia de datos a menos que se utilicen volúmenes o soluciones de almacenamiento específicas.

Los contenedores ofrecen una ejecución consistente y muchas ventajas en términos de eficiencia y portabilidad en comparación con las máquinas virtuales. Sin embargo, también tienen sus propias desventajas y consideraciones, especialmente en términos de seguridad y gestión de datos persistentes.

Para trabajar con contenedores se define la configuración de cada contenedor en un archivo, la ejecución del servicio es consistente independientemente del lugar de ejecución, permitiendo trabajar en el rendimiento de los servicios desde diferentes entornos.

Si en este archivo se define la configuración del contenedor, también existe la definición de los servicios.

El servicio que se ejecuta en un contenedor puede llevar las mismas herramientas, bibliotecas y archivos si se crea una imagen con la configuración de este.

Gracias a la sencillez y simplicidad de utilizar estos archivos de configuración de contenedores y de imágenes, podemos centrarnos en la gestión del producto en términos de rendimiento y almacenamiento.

Pero para trabajar a este nivel aparece la necesidad de crear, buscar y eliminar estos archivos de manera organizada y para ellos se utilizan los gestores de contenedores.

2.2. Gestores de contenedores

Un **gestor de contenedores** es una herramienta que simplifica la gestión y despliegue de imágenes en contenedores. Vamos a enumerar y desarrollar diferentes alternativas disponibles en la actualidad:

1. **Docker**: El gestor de contenedores más utilizado en la actualidad, por su sencillez y facilidad de uso: Se ejecuta en un sistema Linux, pero Windows puede desplegar un subsistema Linux (wsl) para funcionar en él. Además, Docker proporciona una aplicación para escritorio "Docker Desktop" para trabajar a través de una interfaz gráfica e interactiva (Ver referencia [4]).

2. **Podman**: "Podman (el administrador de pods) es una herramienta de código abierto para desarrollar, gestionar y ejecutar los contenedores en los sistemas Linux®". Posee una arquitectura sin Daemon central y opciones de personalización. Se centra en ser compatible con Docker y en proporcionar una interfaz de línea de comandos familiar para los usuarios de Docker (Ver referencia [10]).

3. **LXC (Linux Containers)**: LXC proporciona contenedores que permiten ejecutar aplicaciones en un entorno aislado, pero no son máquinas virtuales completas. "Los contenedores del sistema ofrecen un entorno lo más parecido posible al que se obtendría de una máquina virtual, pero sin la sobrecarga que conlleva ejecutar un kernel independiente y simular todo el hardware". Los contenedores LXC comparten el mismo kernel del sistema operativo host, mientras que las máquinas virtuales utilizan su propio kernel (Ver referencia [6]).

4.**RKT (rocket)**: Es un motor de contenedores que se centra en la simplicidad, la seguridad y la interoperabilidad. Está diseñado para ejecutar contenedores de forma segura y eficiente, y se integra bien con herramientas existentes de contenedores y sistemas de orquestación (Ver referencia [1]).

5. **Containerd**: Containerd es un hilo demonio con la capacidad de gestionar el ciclo de vida completo del contenedor de su sistema host, desde la transferencia y el almacenamiento de imágenes hasta la ejecución y supervisión del contenedor. De hecho, fue desarrollado por la misma empresa que Docker y se utiliza para complementar sus prestaciones (Ver referencia [3]).

6. **CRI-O**: Un gestor de contenedores ligero que está diseñado específicamente para su uso en entornos de contenedores de Kubernetes. También utiliza containerd para la gestión de contenedores a bajo nivel (Ver referencia [8]).

De todas estas tecnologías la más empleada es Docker, por un lado, por las razones listadas anteriormente, pero otra razón es el repositorio de imágenes que posee: Docker Hub (Ver referencia [7]).

2.3. Registro de imágenes: Docker Hub

Docker Hub es un servicio en la nube proporcionado por Docker Inc. que actúa como un registro de imágenes, permitiendo a los usuarios almacenar, distribuir y compartir imágenes de contenedor.

Las **imágenes** son archivos ejecutables que contienen las bibliotecas, las dependencias y los archivos que el contenedor necesita para ejecutarse. Están formadas por capas, donde cada capa agrega dichos componentes.

Docker Hub permite a los usuarios almacenar sus imágenes de manera privada y pública. Permite también buscar y descargar las imágenes públicas de otros usuarios para utilizarlas en sus propios entornos de desarrollo, pruebas o producción. Estos usuarios pueden ser cuentas oficiales de empresas, que ofrecen imágenes de software muy utilizados por otras empresas, por ejemplo, la imagen de Oracle "jdk", que permite ejecutar un archivo compilado de java.

Es común usar imágenes almacenadas en Docker Hub, especialmente las oficiales como Ubuntu, Alpine, PostgreSQL y MySQL. De esta manera se pueden conseguir los servicios de estos productos a un nivel fiable mínimo.

Pero también están disponibles las herramientas más conocidas como Tomcat, Spring Boot, Django, Flask, Maven, Gradle, npm… Estas imágenes facilitan el despliegue y la ejecución de aplicaciones web proporcionando los mismos servicios que proporcionarían en una máquina convencional.

De la misma manera Docker Hub también ofrece imágenes de contenedores de servicio para servicios como Redis, Elasticsearch, Kafka, RabbitMQ, Redis, entre otros. Estos contenedores de servicio son útiles para implementar infraestructuras de microservicios y arquitecturas distribuidas.

Dados estos hechos, en este trabajo se han empleado imágenes de Docker para implementar toda la infraestructura de microservicios.

Docker Hub también ofrece capacidades de automatización para compilar, probar y desplegar imágenes de contenedor. Los usuarios pueden configurar repositorios para que se construyan automáticamente cuando se actualice el código fuente, lo que facilita la integración y la entrega continuas (CI/CD). Esto se complementa con la interfaz interactiva de Docker Desktop para producir un entorno cómodo e intuitivo para los desarrolladores que trabajan en un producto profesional.

Como nuestro objetivo es implementar escalado automático en la infraestructura de microservicios, hemos utilizado imágenes de Docker Hub que proporcionarán lo necesario para implementarlo (Prometheus y Grafana).

Capítulo 3

Análisis

En este capítulo se explicará la metodología a seguir durante el trabajo. Luego, se listarán los requisitos funcionales y no funcionales de la aplicación desarrollada en este trabajo. Se ha empleado una notación estándar IEEE 830 para definir y formalizar los distintos requisitos del sistema.

3.1. Metodología

La metodología de trabajo que se utiliza se fundamenta en una combinación de principios y prácticas de *"DevOps"* y los conceptos de integración continua y despliegue continuo. Esta metodología es esencial para el desarrollo y despliegue eficiente de una aplicación basada en microservicios, asegurando que todos los servicios funcionen de manera independiente pero cohesiva.

- ***DevOps*** es una práctica que combina el desarrollo de software (*Dev*) y las operaciones de tecnología de la información (*Ops*). Su objetivo es acortar el ciclo de vida del desarrollo de sistemas y proporcionar una entrega continua con alta calidad. *DevOps* promueve una cultura de colaboración entre los equipos de desarrollo y operaciones, automatizando procesos para mejorar la eficiencia y la fiabilidad del software.
- **Integración continua (CI)** es una práctica de desarrollo de software donde los desarrolladores integran sus cambios de código en un repositorio compartido con frecuencia, preferiblemente varias veces al día. Cada integración es verificada mediante una compilación automatizada y pruebas, lo que permite detectar y corregir errores de manera rápida y eficiente.
- **Despliegue continuo (CD)**, por otro lado, va un paso más allá de la integración continua. No solo implica la integración frecuente del código, sino también su despliegue automático en entornos de producción o de prueba. Esto asegura que el software está siempre en un estado que puede ser desplegado, permitiendo una entrega rápida y fiable a los usuarios finales.

La metodología específica aplicada a aplicaciones basadas en microservicios implica dividir la aplicación en servicios independientes, cada uno correspondiente a funcionalidades

específicas del producto. Cada microservicio se ejecuta en su propio proceso y se comunica con otros servicios a través de interfaces bien definidas, por ejemplo *API REST*.

Para asegurar una comunicación eficiente entre microservicios, es crucial configurar puertos y URLs (*endpoints*) estables. Esto permite que los microservicios se encuentren y se comuniquen entre sí de manera predecible y segura.

Además, cada microservicio debe mantener su propio repositorio de código separado. Esta separación facilita la independencia en el desarrollo y despliegue, permitiendo que los equipos trabajen de manera aislada en diferentes servicios sin interferir entre sí. Cada microservicio puede ser desarrollado, actualizado y desplegado de manera autónoma, lo que aumenta la agilidad y flexibilidad del sistema en su conjunto.

La integración continua de código asegura que cada cambio se someta a pruebas automáticas y se integre de manera rápida y fiable. El despliegue continuo en entornos de prueba permite que los nuevos componentes y actualizaciones sean evaluados en condiciones controladas antes de ser liberados a producción.

Esta metodología es esencial para garantizar desde el principio que el producto ofrezca prestaciones fiables, mejorando y ampliando conforme se integran nuevos componentes dentro de cada contenedor. Cada imagen de contenedor implementada y cada contenedor desplegado garantizan la consistencia y estabilidad del sistema, independientemente de la fase del ciclo de vida de la aplicación.

En resumen, esta combinación de *DevOps*, integración continua y despliegue continuo, aplicada a una arquitectura de microservicios, permite desarrollar y desplegar aplicaciones robustas y escalables de manera eficiente y ágil.

3.2. Requisitos Funcionales

Los requisitos funcionales describen las funcionalidades que debe tener la aplicación (Ver Tablas 1-5):

Número de requisito	RF-1
Nombre de requisito	Accesibilidad del producto
Descripción	El sistema debe ser un producto software operativo
Motivación	Garantizar que los usuarios finales puedan utilizar el producto
Criterios de aceptación	El producto debe ser accesible a través de las plataformas designadas y estar completamente operativo
Tipo	Requisito

Tabla 1: Accesibilidad del producto

Número de requisito	RF-2
Nombre de requisito	Arquitectura modular y escalable
Descripción	El sistema debe presentar una arquitectura modular y escalable
Motivación	Permitir el crecimiento y la adaptación del sistema a futuras necesidades y cambios
Criterios de aceptación	El sistema debe demostrar modularidad y capacidad de escalabilidad en pruebas de integración y expansión
Tipo	Requisito

Tabla 2: Arquitectura modular y escalable

Número de requisito	RF-3
Nombre de requisito	Recogida y visualización de métricas
Descripción	El sistema debe recoger y mostrar métricas manejables por el desarrollador
Motivación	Proveer información útil para el monitoreo y la optimización del sistema
Criterios de aceptación	Las métricas deben ser visibles y accesibles en un panel de control para desarrolladores
Tipo	Requisito

Tabla 3: Recogida y visualización de métricas

Número de requisito	RF-4
Nombre de requisito	Alertas basadas en métricas
Descripción	El sistema debe mostrar alertas cuando las métricas alcancen ciertos valores
Motivación	Permitir una respuesta rápida ante condiciones anómalas o críticas
Criterios de aceptación	Las alertas deben ser generadas y notificadas correctamente cuando las métricas superen los umbrales predefinidos
Tipo	Requisito

Tabla 4: Alertas basadas en métricas

Número de requisito	RF-5
Nombre de requisito	Reacción a las alertas
Descripción	El sistema debe poder actuar en consecuencia a alertas configuradas, modificando su rendimiento y/o estructura
Motivación	Mantener la estabilidad y el rendimiento óptimo del sistema
Criterios de aceptación	El sistema debe demostrar capacidad de autoajuste en pruebas controladas, modificando parámetros operativos según las alertas recibidas
Tipo	Requisito

Tabla 5: Reacción a las alertas

3.3. Requisitos No Funcionales

Los requisitos no funcionales son aquellos que describen otros aspectos y características de la aplicación (Ver tablas 6-9):

Número de requisito	RNF-1
Nombre de requisito	Prestaciones y características actuales
Descripción	La aplicación debe poseer prestaciones y características presentes en tecnologías actuales
Motivación	Asegurar la relevancia y competitividad del sistema
Criterios de aceptación	La aplicación debe incorporar y demostrar características equivalentes a las tecnologías de vanguardia
Tipo	Requisito

Tabla 6: Prestaciones y características actuales

Número de requisito	RNF-2
Nombre de requisito	Características profesionales
Descripción	Las características deben encontrarse en productos profesionales
Motivación	Garantizar que el sistema cumpla con estándares profesionales de la industria
Criterios de aceptación	La aplicación debe cumplir con criterios de evaluación y comparación con productos profesionales
Tipo	Requisito

Tabla 7: Características profesionales

Número de requisito	RNF-3
Nombre de requisito	Potencial de uso futuro
Descripción	El sistema debe tener el potencial de uso para un continuo desarrollo en el futuro
Motivación	Facilitar la evolución y mejora continua del sistema
Criterios de aceptación	La arquitectura y diseño del sistema deben permitir adiciones y mejoras futuras sin necesidad de reescrituras significativas
Tipo	Requisito

Tabla 8: Potencial de uso futuro

Número de requisito	RNF-4
Nombre de requisito	Bajo coste de desarrollo e implementación
Descripción	El sistema debe tener un bajo coste de desarrollo e implementación
Motivación	Asegurar la viabilidad económica del proyecto
Criterios de aceptación	Los costes estimados y reales de desarrollo e implementación deben estar dentro de los límites presupuestarios establecidos
Tipo	Requisito

Tabla 9: Bajo coste de desarrollo e implementación

3.4. Restricciones

Concretan un aspecto del proyecto discriminando cualquier otro tipo de opción diferente a los requisitos(Ver Tablas 10 y 11):

Número de requisito	R-1
Nombre de requisito	Funcionamiento en entorno Docker
Descripción	El sistema debe funcionar en un entorno Docker
Motivación	Asegurar la portabilidad y la consistencia del entorno de ejecución
Criterios de aceptación	El sistema debe poder ser desplegado y ejecutado correctamente en un contenedor Docker
Tipo	Restricción

Tabla 10: Funcionamiento en entorno Docker

Número de requisito	R-2
Nombre de requisito	Compatibilidad con sistemas operativos actuales
Descripción	El sistema debe poder funcionar en sistemas operativos empleados en el presente con soporte de mantenimiento por parte del desarrollador
Motivación	Garantizar la interoperabilidad y uso inmediato en los entornos actuales
Criterios de aceptación	El sistema debe ser probado y demostrado funcionar en los sistemas operativos especificados
Tipo	Restricción

Tabla 11: Compatibilidad con sistemas operativos actuales

Capítulo 4

Diseño e implementación

Para realizar un producto software con auto escalado integrado hace falta un modelo de negocio que necesite dicha funcionalidad. Por ejemplo, la venta de entradas de un concierto o un servicio de noticias de última hora. En estos casos de uso la cantidad de conexiones soportables debe crecer al mismo tiempo que la demanda, a veces de manera instantánea.

Pero la escalabilidad no solo incluye esas situaciones, también existen servicios que las integran por ejemplo para la venta de productos que requieren reserva y confirmación(ver Figura 3).

4.1. Estructura e implementación de auto escalado

Lo primero ha sido encontrar un sistema funcional de monitorización que permitiera implementar la lógica de decisión de escalar el servicio. A partir de su análisis se ha procedido a integrar la base del servicio (ver Figura 1):

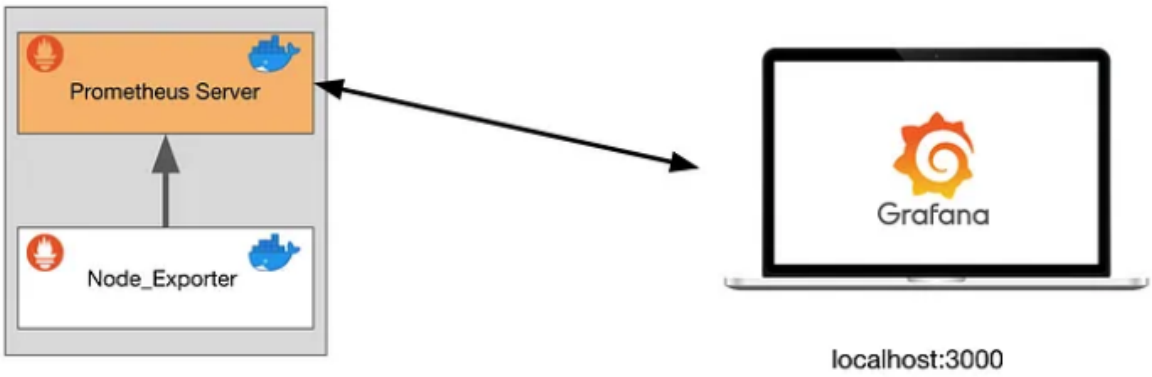

Figura 1: Base de la monitorización inicial, esquema de la primera aproximación del estudio de la prestación del autoescalado

Para integrar servicios de monitorización de rendimiento y métricas en un producto, utilizando herramientas como cAdvisor, Node Exporter, Prometheus y Grafana, es importante entender cómo cada una de estas herramientas encaja en el ecosistema de monitorización. Estas son las opciones elegidas para la implementación del servicio:

1. **cAdvisor (Container Advisor):** es una herramienta de monitorización que proporciona información sobre el rendimiento de los contenedores en un host. Es capaz de recolectar datos de uso de recursos (CPU, memoria, disco, red) de cada contenedor, lo que es esencial para comprender cómo se están utilizando los recursos a nivel de contenedor.

    - Instalación y configuración: cAdvisor se ejecuta como un contenedor en cada host donde hay contenedores que necesitan ser monitoreados.
    - Recolección de datos: cAdvisor recoge datos en tiempo real sobre el uso de recursos de los contenedores.

2. **Node Exporter:** recopila métricas del sistema operativo y de hardware de un nodo, como el uso de CPU, memoria, almacenamiento y red, y las exporta en un formato compatible con herramientas de monitoreo como Prometheus. Es de código abierto.

    - Instalación y configuración: Node Exporter se ejecuta como un binario o contenedor en cada host que necesita ser monitoreado.
    - Recolección de datos: Node Exporter expone estas métricas a través de un endpoint HTTP que puede ser alcanzado por Prometheus.

3. **Prometheus** es un sistema de monitoreo y alerta de código abierto diseñado especialmente para entornos de contenedores y microservicios. Permite recopilar métricas de sistemas y servicios, almacenarlas en una base de datos de series temporales y consultarlas. Además, Prometheus ofrece capacidades de alerta para notificar sobre eventos importantes según ”flags”definidas por el usuario. Es una herramienta popular en entornos de infraestructura moderna debido a su escalabilidad, facilidad de uso y su integración con otras herramientas como Grafana para visualización de datos.

    - Instalación y configuración: Prometheus se configura con un archivo de configuración que define los endpoints de donde recoger métricas (por ejemplo, cAdvisor y Node Exporter).
    - Recolección de datos: Prometheus ”scrapea”(recoge) periódicamente las métricas de los endpoints configurados.
    - Almacenamiento y consulta: Prometheus almacena las métricas en una base de datos de series temporales y proporciona un lenguaje de consulta llamado PromQL para extraer y analizar los datos.

4. **Grafana** es una plataforma de visualización de datos de código abierto que se utiliza ampliamente en sistemas de monitorización y análisis. Permite crear paneles interactivos y gráficas para visualizar métricas y registros de sistemas, aplicaciones, bases de datos y otros recursos. Las gráficas y los paneles de las gráficas son altamente personalizables y su integración con Prometheus hacen que se utilice bastante en productos de software de arquitectura de contenedores y en la nube.

    - Instalación y configuración: Grafana se puede ejecutar como un contenedor, binario o servicio en cualquier máquina.
    - Configuración de datos: En Grafana, se configura una fuente de datos que apunte a la instancia de Prometheus.

- Visualización: Grafana permite crear dashboards personalizados para visualizar las métricas recogidas por Prometheus de manera gráfica e interactiva.

Integración de los Componentes

1. Despliegue de cAdvisor y Node Exporter:
   - Se ejecuta cAdvisor en cada contenedor para recolectar métricas de cada contenedor.
   - Se ejecuta Node Exporter en cada nodo para recolectar métricas a nivel de host.
2. Configuración de Prometheus:
   - Configura Prometheus para scrapear los endpoints de cAdvisor y Node Exporter. Esto se hace editando el archivo prometheus.yml para incluir los targets (endpoints) correspondientes.
3. Despliegue de Prometheus:
   - Ejecuta Prometheus como un servicio o contenedor en la infraestructura.
4. Configuración de Grafana:
   - Despliega Grafana como un servicio o contenedor.
   - Añade Prometheus como una fuente de datos en Grafana.
   - Crea dashboards y paneles en Grafana utilizando las métricas disponibles en Prometheus.
5. Visualización y Monitoreo:
   - Utiliza los dashboards de Grafana para visualizar en tiempo real el rendimiento de tus contenedores y hosts.
   - Configura alertas en Prometheus y Grafana para recibir notificaciones cuando las métricas superen ciertos umbrales.

4.2. Caso de uso, una tienda en línea

En este caso se muestra una tienda en línea que ofrece un servicio de venta de productos.(ver Figura 2). Estos productos se pueden reservar y elegir sus características. Esta prestación del servicio requiere que los componentes puedan ser escalados mientras el cliente determina las características del producto. El producto se bloquea temporalmente para el resto de los usuarios, por si finalmente lo reserva(ver Figura 4).

El **flujo de ejecución** del caso de uso es el siguiente: EL usuario puede navegar por la página web y ver los productos disponibles. Los productos se pueden buscar y se puede acceder a la información de cada uno. Sin embargo, también hay un servicio de reserva,

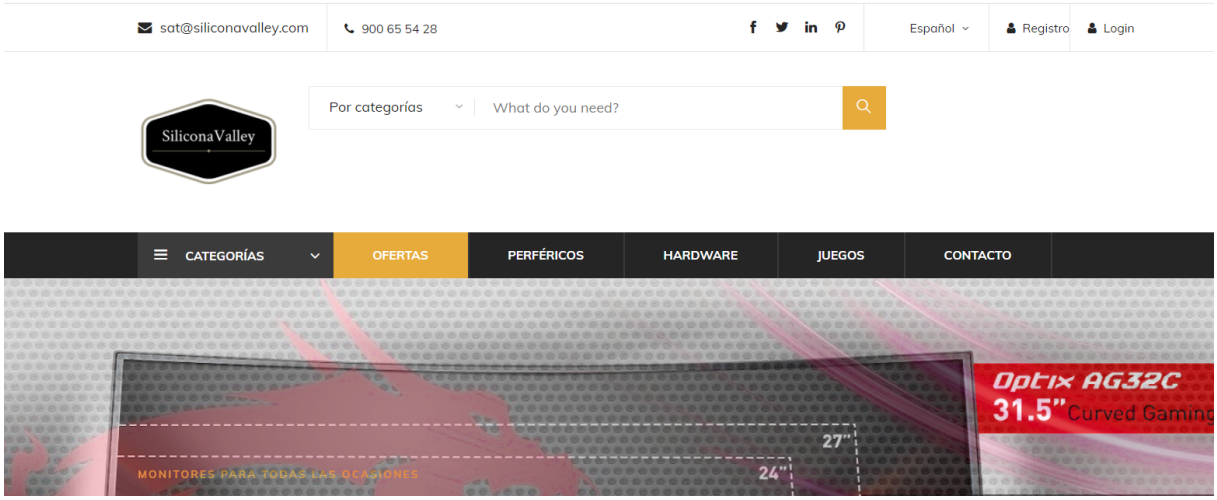

Figura 2: Página principal de la tienda en línea, la aplicación que se presenta como caso real

pero está destinada a los usuarios que están registrados. De esta manera, si el usuario desea reservar un producto, se registra y se autentifica. Una vez identificado, la pestaña de información cambia y aparece disponible un botón de reserva. En el momento en el que el usuario pulse el botón, se inicia una servicio en el que el producto queda bloqueado. El usuario tiene un tiempo para formalizar su reserva eligiendo las características determinadas del producto, y es durante ese tiempo que los recursos de la web varían, y necesitan ser escalados.

Figura 3: Caso de uso: Compra en línea de un usuario registrado en el sistema con opción a reservar un producto.

Figura 4: Caso de uso que requiere escalado: el sistema reserva un producto que todavía no se ha confirmado mientras se eligen sus características.

Para conseguir una infraestructura auto escalable primero hace falta una estructura modulada que pueda ser escalable. La modulación de una arquitectura de software implica organizar el sistema en **módulos independientes** y cohesivos que realizan funciones específicas.

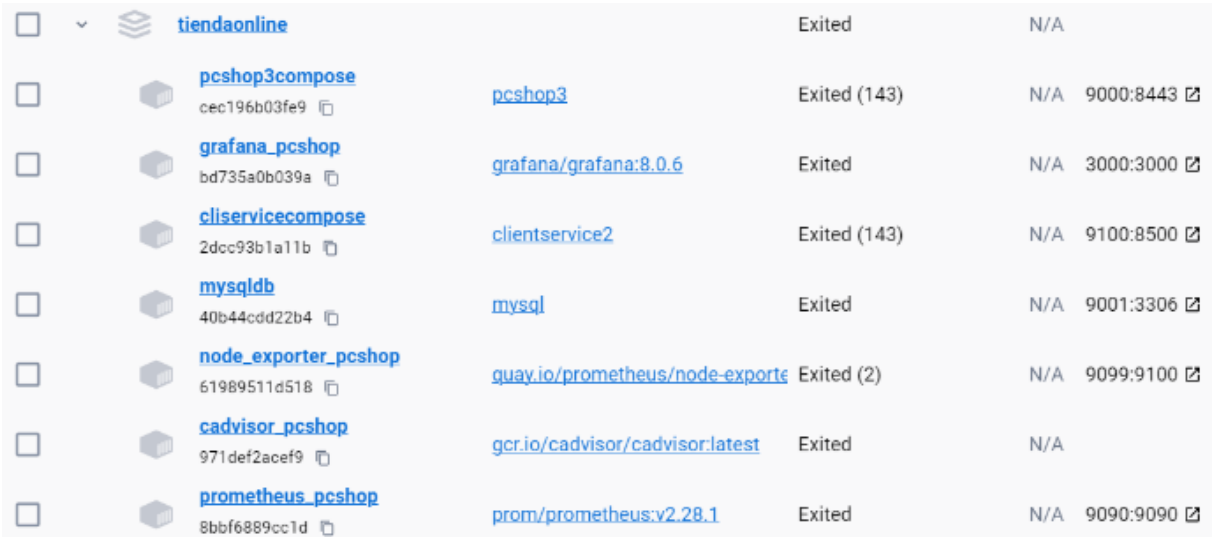

Figura 5: Aplicación completa desplegada en el gestor de contenedores y visualizado gráficamente por Docker Desktop.

Identificamos diferentes funcionalidades y con ello diferentes componentes. Utilizamos Docker para implementarlos y Docker Desktop para trabajar con ello, además establecemos una relación directa entre un componente y una imagen de Docker. Docker Desktop es la herramienta de Docker con una interfaz gráfica figura (Ver figura 5).

Implementar el sistema de esta manera define las dependencias entre los módulos de manera que cada módulo tenga acceso solo a la funcionalidad que necesita y no a más. Esto reduce el acoplamiento entre los módulos y facilita la modificación de partes del sistema sin afectar a otras. Es necesario para conectar los módulos lo justo y necesario ya que el afecta al despliegue y rendimiento de manera directa.

Siendo así la relación, tenemos un producto que estará formado por microservicios en ejecución que estarán definidos de manera consistente en imágenes Docker y ejecutadas y conectadas entre ellas en un entorno Docker. También estarán configuradas según las necesidades de las herramientas empleadas para el desarrollo de cada componente.

4.3. Composición de la aplicación

En este trabajo se han empleado imágenes de Docker para implementar toda una infraestructura de microservicios.

4.3.1. Componentes principales

La primera imagen que integramos es la de la **base de datos**. Para implementarla utilizamos una imagen de Docker Hub que proporciona un servicio de base de datos relacional fiable (MySQL).

La segunda imagen que ejecutamos en un contenedor es la de la **tienda online**, realizada durante la carrera y extendida para este trabajo, demuestra que los contenidos aprendidos durante el curso son útiles para las competencias requeridas por las empresas para sus

Algoritmo 1 Contenido de un DockerFile, archivo que declara qué software instalar y ejecutar en la aplicación. Cada instrucción en un Dockerfile crea una capa en la imagen de Docker.

```
# Usa una imagen base que incluya Java

FROM openjdk

# Establece el directorio de trabajo

WORKDIR /app

# Copia el archivo JAR al contenedor

COPY siliconaValley-0.0.1-SNAPSHOT.jar /app

# Comando para ejecutar la aplicación

CMD ["java", "-jar", "siliconaValley-0.0.1-SNAPSHOT.jar"]
```

productos. Para crearla utilizamos un archivo llamado DockerFile, que es el archivo donde se declaran las instrucciones de construcción de imagen Docker.

Una **imagen de Docker** es un archivo comprimido que contiene todos los componentes necesarios para ejecutar una aplicación en un contenedor Docker. Para emplear estos componentes se ejecutan las líneas definidas en el archivo DockerFile para producir una imagen compuesta por capas que contienen los componentes. Cada línea del DockerFile aplica una capa utilizando ciertos comandos. En nuestro caso utilizaremos el siguiente contenido (Ver código : 1):

Como se puede observar, utiliza una imagen ya creada del Docker Hub que proporciona openjdk para poder ejecutar el archivo .jar que hemos desarrollado personalmente. Este hecho es importante para entender la lógica y potencia de trabajo con Docker.

El DockerFile de la tienda es sencillo en este caso, ya que solamente utiliza dos cláusulas principales. FROM: que especifica la imagen base que se utilizará como punto de partida. Por ejemplo, se podría usar una imagen de Ubuntu, CentOS, Alpine Linux CMD: Se ejecutará cuando se inicie un contenedor basado en la imagen, de la misma manera que ejecutándolo en una terminal de un sistema.

4.3.2. Componentes secundarios

Incluir componentes como cAdvisor, Node Exporter, Prometheus y Grafana en una aplicación basada en microservicios que requiere escalado es fundamental, especialmente para garantizar un rendimiento óptimo y detectar y solucionar problemas de gestión de los recursos:

- **Monitoreo Proactivo**: Con un sistema de monitoreo robusto en su lugar, se pueden identificar patrones de uso y cargas de trabajo que pueden requerir escalado.

Esto permite planificar y ejecutar el escalado de manera proactiva, asegurando que los servicios continúen funcionando sin interrupciones.

- **Gestión Eficiente de Recursos**: Al tener visibilidad sobre cómo se utilizan los recursos, se puede optimizar la asignación de los mismos. Esto es particularmente importante en un entorno de microservicios, donde cada servicio puede tener diferentes requerimientos de recursos.

Por otro lado, y habiendo acabado con las imágenes de Docker Hub, tenemos nuestro **servicio interno** dedicado al tratamiento de las operaciones de compra de un cliente. De este servicio también se generará una imagen a través de un DockerFile y será escalado automáticamente:

- Elaborado en Java con el framework de Spring Boot.
- API pública con métodos HTTP.
- Implementa métodos asíncronos para la efectiva prestación del caso de uso de la reserva del producto, donde bloquea el producto reservado en la base de datos mientras el usuario elige entre las opciones ofrecidas de reserva.

Por último, codificamos un **servicio de auto escalado** que recibe la alerta de Grafana, y se comunica con la API de Docker para escalar la aplicación:

- Elaborado en Java con el framework de Spring Boot.
- API pública con método HTTP POST para replicar el servicio.
- El servicio se sirve de la biblioteca de Docker en Java para interactuar con el hilo demonio de Docker, primero obtiene los datos del servicio que existe en Docker, luego crea una nueva configuración para un nuevo contenedor, y por último ordena a Docker a desplegar el servicio en un contenedor con la nueva configuración a través de la petición POST de la API (Ver código : 6).

4.4. Configuración de la aplicación

Una vez elegidos los componentes de la aplicación, hay que elegir por donde empezar la configuración de los componentes y del sistema que forman. Tenemos las tecnologías elegidas y la herramienta para desplegarlas, pero los componentes todavía no están conectados ni relacionados.

4.4.1. Configuración funcional

Para empezar el despliegue hay que establecer un orden de prioridades a la hora de ejecutar los servicios. De hecho hay componentes que necesitan que otros estén disponibles para desplegarse de manera correcta.

Lo primero para diseñar la aplicación es configurar el servicio de MySQL en un contenedor(Ver código : 2). De esta manera, podremos crear la base de datos de la aplicación, permitiendo que se generen las tablas necesarias para otros servicios y que estén disponibles para el resto de los componentes(Ver código : 3).

Algoritmo 2 Ejecución de la base de datos, con red y puerto.

```
docker run -d -p 9001:3306 --network red1 --name mysqldb -e
    MYSQL_ROOT_PASSWORD=password mysql
```

Accedemos a ella de esta manera para incluir datos válidos que utilizaremos en los casos de prueba.

Algoritmo 3 Configuración de MySQL, creación de la base de datos y ejemplo de creación de un producto.

```
docker exec -it mysqldb bash
mysql -u root -p
create database test;
use test;

INSERT INTO `test`.`producto` (`id_producto`, `categoria`, `
    descripcion`, `nombre`, `precio`, `url_imagen`) VALUES ('7',
     'Portatil', 'Intel Core i7, 16GB RAM, Nvidia GTX1660Ti 6GB
    ', 'MSI GP65 Leopard 9SD-013ES', '999','./img/61fw695XDJL._
    SL1024_.jpg');
```

Lo segundo es iniciar la tienda en linea (Ver código : 4), ya que la base de datos está creada y puede conectarse a ella para asegurar la persistencia:

Algoritmo 4 Ejecución de la tienda en línea, con parámetros de entrada de red, puerto y usuario en la base de datos. Desplegamos la imagen de la tienda e un contenedor.

```
docker run --name pcshop2 -p 9000:8443 --net red1 -e MYSQL_HOST
    =mysqldb -e MYSQL_PORT=3306 pcshop2
```

Pero lanzar una arquitectura formada por contenedores se vuelve tedioso, lioso y arriesgado ejecutarlos de uno en uno de esta manera. Se soluciona con el archivo de **Docker-compose**, que es el siguiente paso a la hora de diseñar una arquitectura por contenedores que escale.

Cuantos más contenedores tenemos o más flexibles queremos que sean, configurarlos de manera manual se vuelve más complicado por cada nuevo contenedor. En este archivo Docker-compose se enlazan los contenedores y se relacionan de manera sencilla (Ver Figuras 6 y 7):

Para diseñar este archivo hace falta bastante conocimiento sobre qué se está configurando, hay que conocer los **tipos de redes** que Docker ofrece (bridge, none, host). Conocer las

políticas de reinicio correctas y las dependencias entre los contenedores en caso de que alguno necesite que otro ya esté en ejecución.

- Las redes **bridge** permiten conectar los contenedores entre sí a través de la red a la que pertenecen, a través de esta red se pueden comunicar de manera completa copn el resto de contenedores de la red, y con la máquina anfitriona.

- Las redes **none** permiten ejecutar un contenedor aislado y hermético para asegurar que no establece ningún tipo de conexión.

- Las redes **host** permiten al Docker Host establecer una conexión con el contenedor que las configura.

En cuanto a las redes, trabajamos con las redes bridge ya que nos permiten conectar los contenedores y que puedan comunicarse. Para ello establecemos los puertos de entrada y salida respecto a cada uno, por ejemplo, la relación 9000:8080 establece que el puerto 9000 de la máquina que ejecuta Docker y un contenedor el contenedor configurado se mapea con el puerto 8080 del entorno formado dentro del contenedor.

```
volumes:
  grafana-data:
  prometheus-data:

networks:
    red1:
        external: true
services:
  cliservicecompose:
    image: clientservice2
    container_name: cliservicecompose
    ports:
    - "9100:8500"
    networks:
    - red1
  pcshop3compose:
    image: pcshop3
    container_name: pcshop3compose
    ports:
    - "9000:8443"
    networks:
    - red1
    environment:
      MYSQL_HOST: mysqldb
      MYSQL_PORT: 3306
    depends_on:
      - mysqldb
  mysqldb:
    image: mysql
    container_name: mysqldb
    ports:
    - "9001:3306"
    environment:
      MYSQL_DATABASE: test
      MYSQL_ROOT_PASSWORD: password
    networks:
    - red1

#pcshop3 hay que relanzarlo, a la primera no sale
#politica restart

  grafana_pcshop:
    image: grafana/grafana:8.0.6
    container_name: grafana_pcshop
    restart: unless-stopped
    volumes:
    - grafana-data:/var/lib/grafana
    ports:
    - 3000:3000
```

Figura 6: Despliegue declarativo de cada componentes y su configuración de la aplicación completa con Docker Compose - Primera parte

```
node_exporter_pcshop:
  image: quay.io/prometheus/node-exporter:latest
  container_name: node_exporter_pcshop
  restart: unless-stopped
  ports:
  - 9099:9100
  ###### linux
  # command:
  # - '--path.rootfs=/host'
  # pid: host
  # volumes:
  # - '/:/host:ro,rslave'
  ###### windows
  volumes:
    - /proc:/host/proc:ro
    - /sys:/host/sys:ro
  command:
    - '--path.procfs=/host/proc'
    - '--path.sysfs=/host/sys'
    - --collector.filesystem.ignored-mount-points
    - "^/(sys|proc|dev|host|etc|rootfs/var/lib/docker/containers
  networks:
  - red1

cadvisor_pcshop:
  image: gcr.io/cadvisor/cadvisor:latest
  container_name: cadvisor_pcshop
  restart: unless-stopped
  expose:
  - 8080
  volumes:
  - /:/rootfs:ro
  - /var/run:/var/run:rw
  - /sys:/sys:ro
  - /var/lib/docker/:/var/lib/docker:ro
  networks:
```

Figura 7: Despliegue declarativo de cada componentes y su configuración de la aplicación completa con Docker Compose - Segunda parte

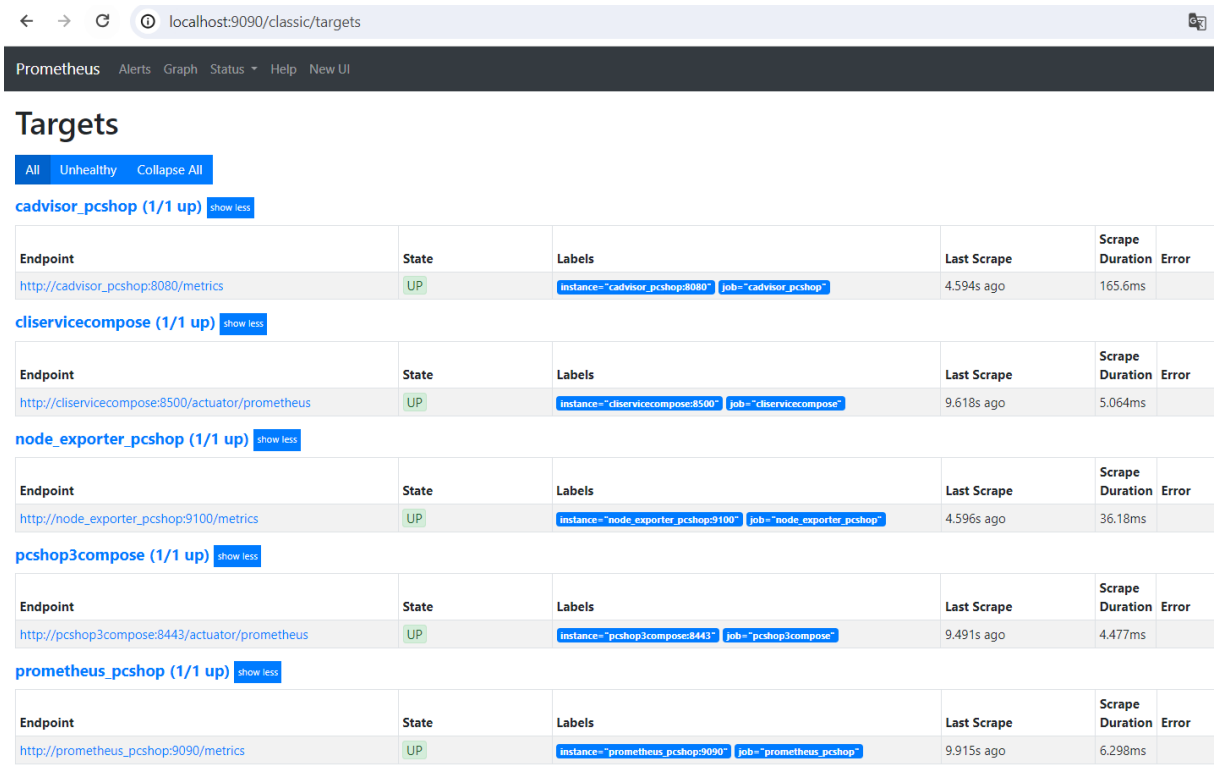

Figura 8: Configuración de Prometheus de las fuentes de métricas del sistema

Prometheus se configura en un archivo de configuración, aquí logramos exportar las métricas de los servicios de la tienda online. Están disponibles para Prometheus ya que Cadvisor y Node Exporter las pueden exportar a través de la declaración de permisos en los archivos de configuración del producto (*pom.xml, application.properties*)(Ver código : 5). En el archivo se definen los *scrape configs*, que son los nombres de los trabajos de recolección. Se pueden configurar para establecer los intervalos de recolección, el tiempo máximo dedicado a cada intervalo, y varias configuraciones mas.

Algoritmo 5 ARCHIVO PROMETHEUS.YML. Configura la red de trabajo de Prometheus, informándole los servicios de los que sacar métricas y estableciendo la conexión.

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
    - targets: ['prometheus:9090']

  - job_name: 'cadvisor'
    static_configs:
    - targets: ['cadvisor:8080']

  - job_name: 'node_exporter'
    static_configs:
    - targets: ['node_exporter:9100']

  - job_name: 'app_example'
    static_configs:
    - targets: ['app_example:8080']
    #metrics_path: '/metrics'
    #metrics_path: '/prometheus'
```

4.4.2. Configuración y visualización de métricas en Grafana

Finalmente entramos en la configuración del sistema de monitorización, aquí tenemos un servicio por pantalla que mostrará gráficas definidas por nosotros en base a las métricas que hemos configurado para enviar de nuestro sistema (Ver figura 10). Grafana es una plataforma de visualización de datos que permite crear, explorar y compartir gráficos interactivos y paneles a partir de diferentes fuentes de datos. Estas gráficas se caracterizan por las siguientes cualidades:

1. **Interactividad**:
   - **Zoom**: Permiten acercar y alejar en los datos y desplazarse lateralmente para explorar diferentes partes del conjunto de datos.
   - **Herramientas de selección**: Facilitan la selección de rangos de tiempo específicos o subconjuntos de datos.
2. **Personalización**:
   - **Temas**: Opciones de temas claros y oscuros.
   - **Estilo de gráficos**: Permite ajustar colores, leyendas, títulos, ejes, y otras configuraciones visuales.
   - **Anotaciones**: Añadir notas y eventos específicos en los gráficos para resaltar puntos de interés o eventos importantes.
3. **Tipos de Gráficos**:

- **Gráficos de líneas**: Ideal para datos de series temporales.
- **Gráficos de barras**: Útil para comparar diferentes categorías de datos.
- **Gráficos circulares y de anillos**: Para mostrar proporciones y porcentajes.
- **Mapas de calor**: Para visualizar la densidad de datos.
- **Diagramas de dispersión, métricas únicas, tablas y otros**.

4. **Paneles (Dashboards)**:

- **Compilación de gráficos**: Agrupa múltiples gráficos en un solo panel para una vista consolidada.
- **Filtros y variables**: Uso de variables dinámicas y filtros para modificar la visualización sin necesidad de cambiar la configuración del gráfico.
- **Consultas múltiples**: Integración de múltiples consultas y fuentes de datos en un solo gráfico.

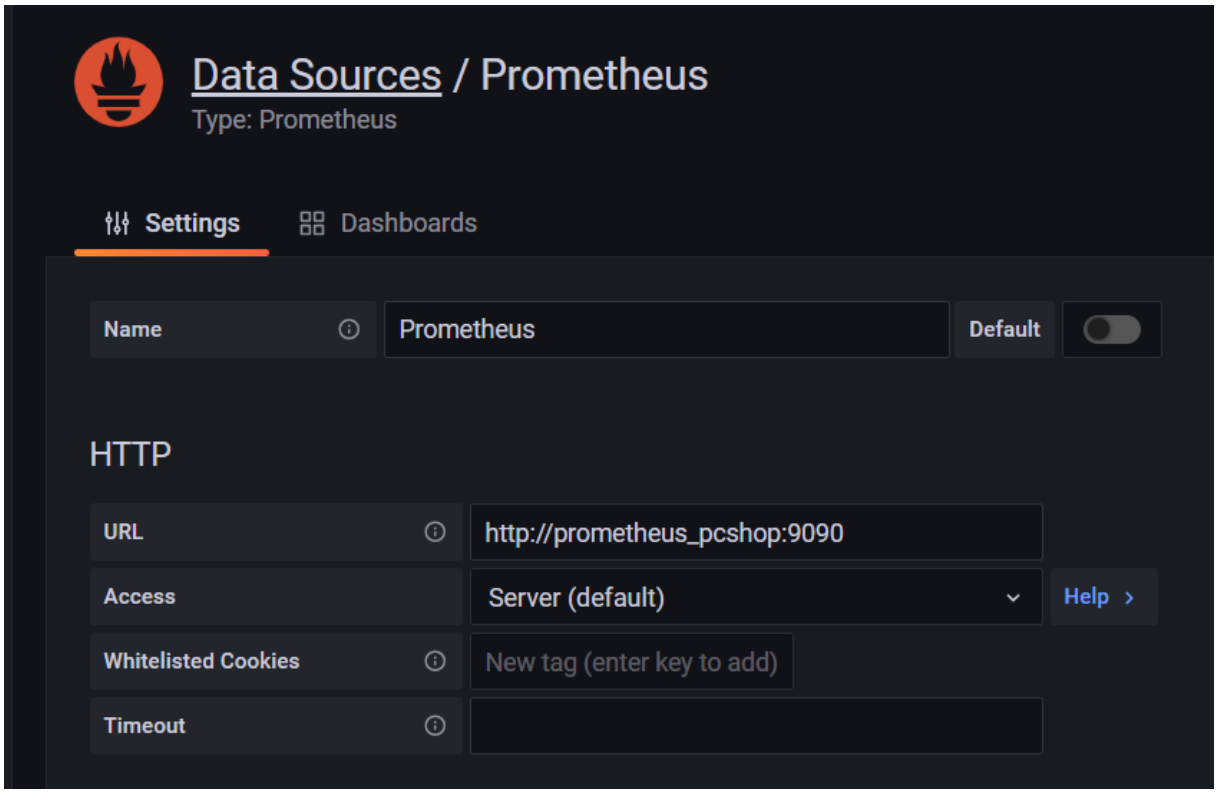

Figura 9: Configuración de la fuente de métricas de Grafana

Lo primero necesario es configurar la recepción de datos, declarando la dirección correcta y probando (Ver figura 9).

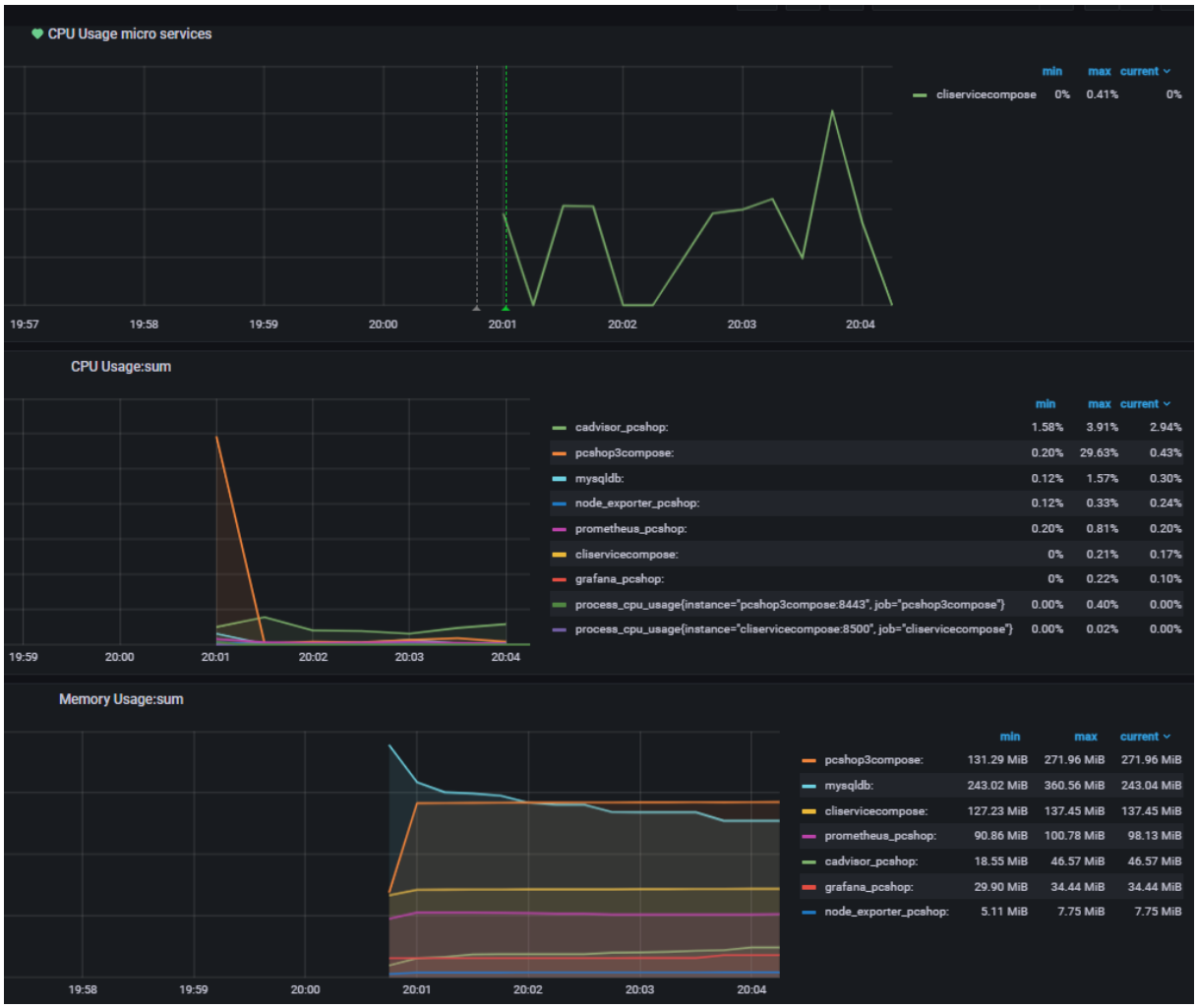

Figura 10: Tablón de gráficas en Grafana como interfaz interactiva

Para recrear estas gráficas tenemos el buscador de métricas que Grafana recibe a través de Prometheus, el cual las consigue gracias a Cadvisor y a Node Exporter. La cantidad de métricas es bastante grande. Dependiendo de la métrica escogida, aparecen en el paso dos las etiquetas posibles. También se permiten la combinación de valores (Ver figura 11).

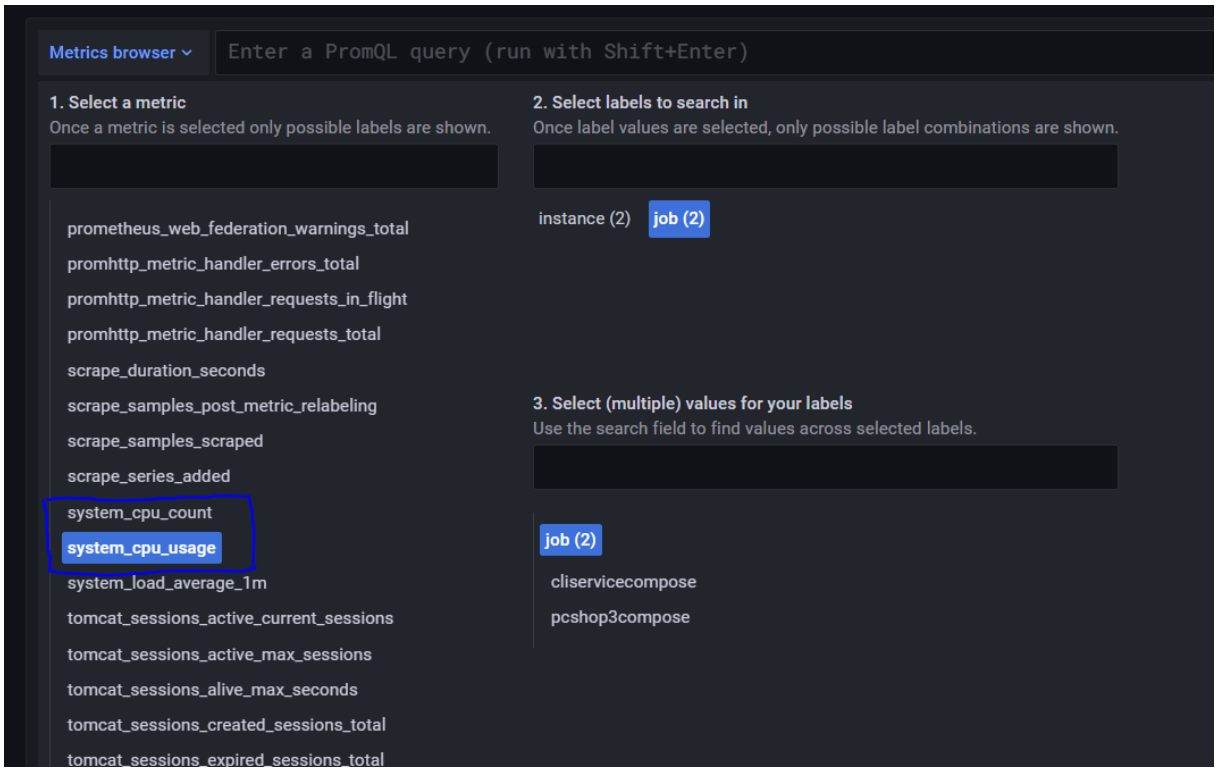

Figura 11: Elección de las métricas recibidas a través de la fuente de datos configurada (Prometheus)

Estas métricas se utilizan en la consulta que mostrará la gráfica definitiva. Cuando se selecciona una métrica en el "Metric Browser", Grafana genera una consulta para recuperar los datos correspondientes de la fuente de datos configurada.

En concreto, nuestra consulta (Ver figura 12) la podemos desglosar de la siguiente manera:

- **Función irate:**

$$\text{irate}(\text{container\_cpu\_user\_seconds\_total}\{name = \text{"\$}name\text{"}\}[5m])$$

  - irate: Función en PromQL que calcula la tasa de incremento instantánea por segundo de una métrica de tipo contador durante un rango de tiempo especificado.
  - container_cpu_user_seconds_total: Métrica que representa el número total de segundos de CPU en modo de usuario consumidos por los contenedores.
  - name= "$name: Filtro que aplica una expresión regular para el valor de la etiqueta name. $name es una variable que Grafana sustituirá con el valor seleccionado en el panel.
  - [5m]: Indica que irate calculará la tasa de incremento para los últimos 5 minutos.

- **Multiplicación por 100:**

$$\text{irate}(\text{container\_cpu\_user\_seconds\_total}\{name = \text{"\$}name\text{"}\}[5m]) \times 100$$

  Multiplica el resultado de irate por 100 para convertir la tasa de segundos por segundo en un porcentaje del uso de CPU.

- **Función sum without:**
  - sum: Función de agregación que suma los valores de la serie de tiempo.
  - without: Indica que la suma debe realizarse excluyendo las etiquetas especificadas (dc, from, id y cualquier otra etiqueta que se expanda de $sum_without:csv). Este es un patrón para realizar la agregación sin agrupar por estas etiquetas.

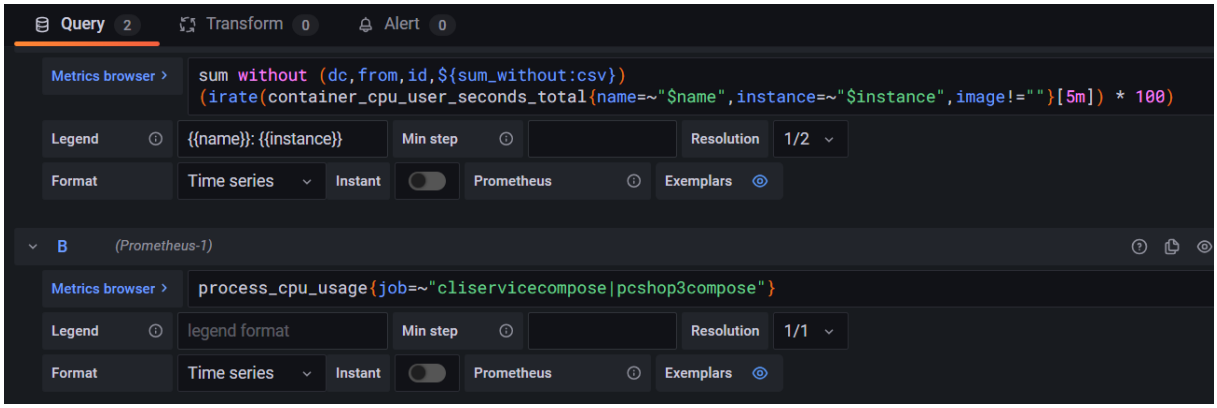

Figura 12: Configuración de la gráfica que ilustrará el uso de CPU del servicio.

4.4.3. Configuración de la alerta

Ahora configuramos una alerta que monitorizará esta gráfica, se configurará en base al uso de CPU del servicio de reserva del caso de uso (Ver figura 13). Para configurarlo hay que conocer que las alertas en Grafana se caracterizan por las siguientes cualidades:

1. **Configuración de Alertas**:
   - **Umbrales personalizados**: Se pueden definir umbrales específicos para diferentes métricas y condiciones.
   - **Condiciones complejas**: Soporta condiciones lógicas avanzadas (AND, OR) y combinaciones de múltiples métricas.
   - **Frecuencia de evaluación**: Configura la frecuencia con la que se evaluarán las condiciones de alerta. En nuestro caso de uso son 15 segundos.
2. **Notificaciones**:
   - **Canales de notificación**: Envío de notificaciones a través de varios canales como correo electrónico, Slack, PagerDuty, Microsoft Teams, y más.
   - **Plantillas de mensajes**: Personalización de mensajes de notificación utilizando plantillas y variables.
   - **Escalamiento**: Configuración de reglas de escalamiento para notificar a diferentes equipos o personas según la gravedad de la alerta.
3. **Gestión de Alertas**:

- **Silenciar alertas**: Posibilidad de silenciar alertas temporalmente durante periodos de mantenimiento o para evitar alertas repetitivas.
- **Historial de alertas**: Registro histórico de todas las alertas generadas, incluyendo su estado y acciones tomadas.
- **Alertas manuales**: Capacidad para crear alertas manuales directamente desde la interfaz de usuario.

4. **Integración con Paneles (Dashboards)**:
   - **Visualización de alertas**: Mostrar el estado de las alertas directamente en los paneles de Grafana.
   - **Indicadores visuales**: Uso de indicadores visuales (colores, iconos) para resaltar el estado de las alertas en los gráficos.
   - **Paneles dedicados a alertas**: Crear paneles específicos para el monitoreo y gestión de alertas.

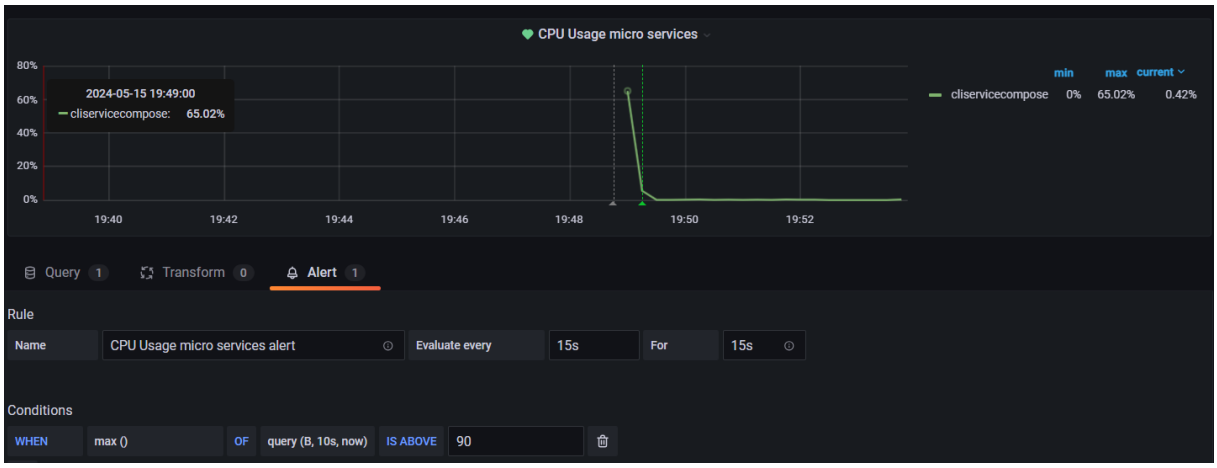

Figura 13: Configuración de la alerta de uso de CPU máximo

Se configura el canal de la alerta por el que se va a enviar (*endpoint*) (Ver figura 14). En ella se pueden configurar diferentes tipos de petición:

La configuración **Type** (Tipo) especifica el tipo de canal de notificación que se desea utilizar para enviar las alertas. Grafana admite varios tipos de canales de notificación, lo que permite elegir el método más adecuado para tu caso de uso. Algunos de los tipos de canales de notificación más comunes incluyen:

- **Email**: Utiliza direcciones de correo electrónico para enviar notificaciones.
- **Slack**: Envía notificaciones a un canal de Slack (sistema de comunicación de organizaciones) utilizando un webhook.
- **Webhook**: Utiliza una URL de webhook genérica para enviar notificaciones a otros sistemas.
- **Opsgenie**: Integra con Opsgenie (servicio de alerta y gestión de incidentes) para la gestión de alertas.

- **VictorOps**: Utiliza VictorOps (plataforma de gestión) para la gestión de incidentes.

- **Sensu**: Integra con Sensu (plataforma de monitorización) para la monitorización de infraestructura.

Para los tipos de canal de notificación que utilizan **Webhook**, es necesario proporcionar una URL específica a la cual Grafana enviará las notificaciones. Esta URL es un punto de integración donde las alertas generadas por Grafana serán enviadas.

Esta es una URL genérica de webhook donde Grafana enviará las notificaciones. Esta URL debe ser proporcionada por el sistema con el que se desea integrar. En este caso conocemos la dirección del servicio de escalado.

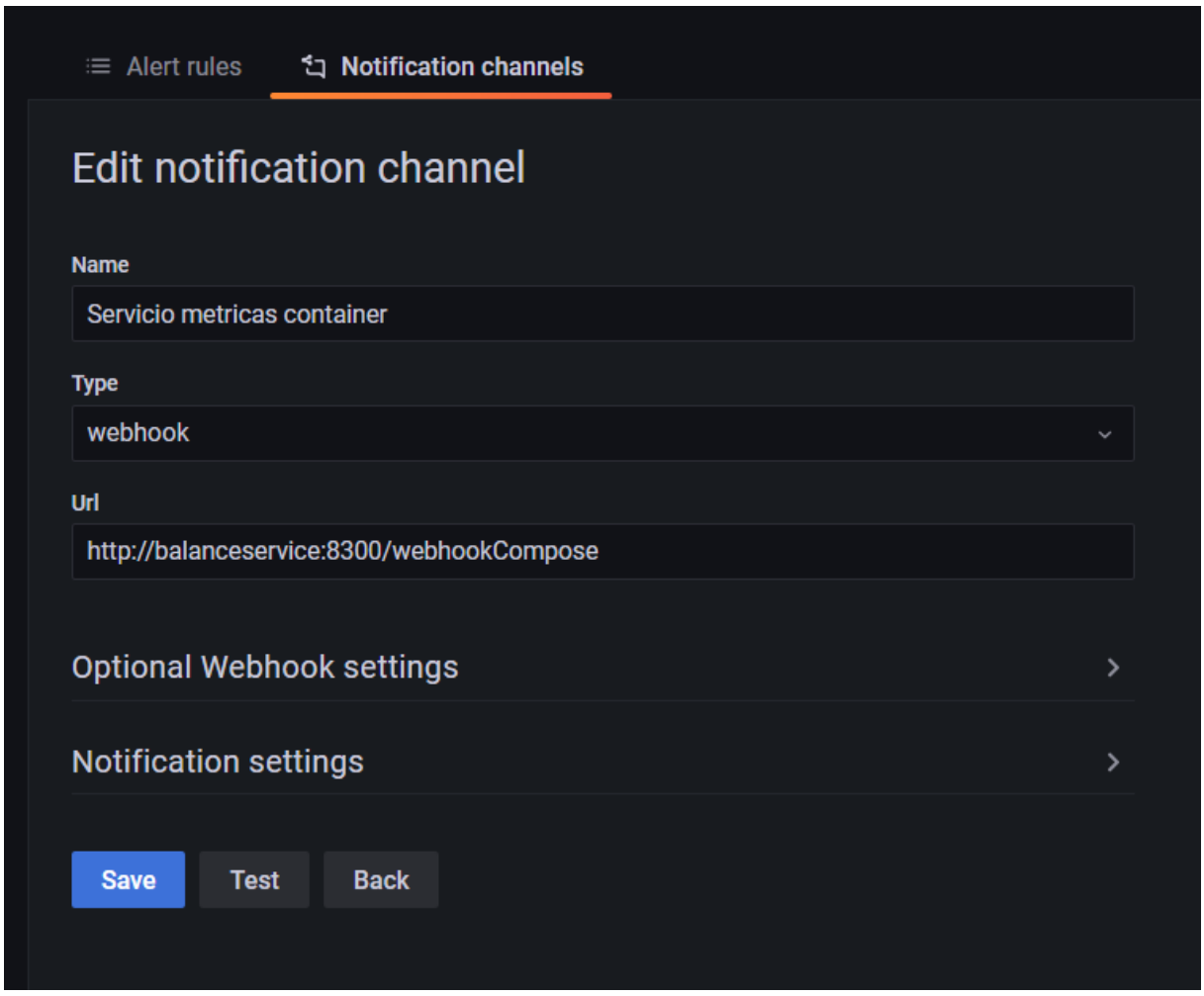

Figura 14: Configuración del canal de la alerta

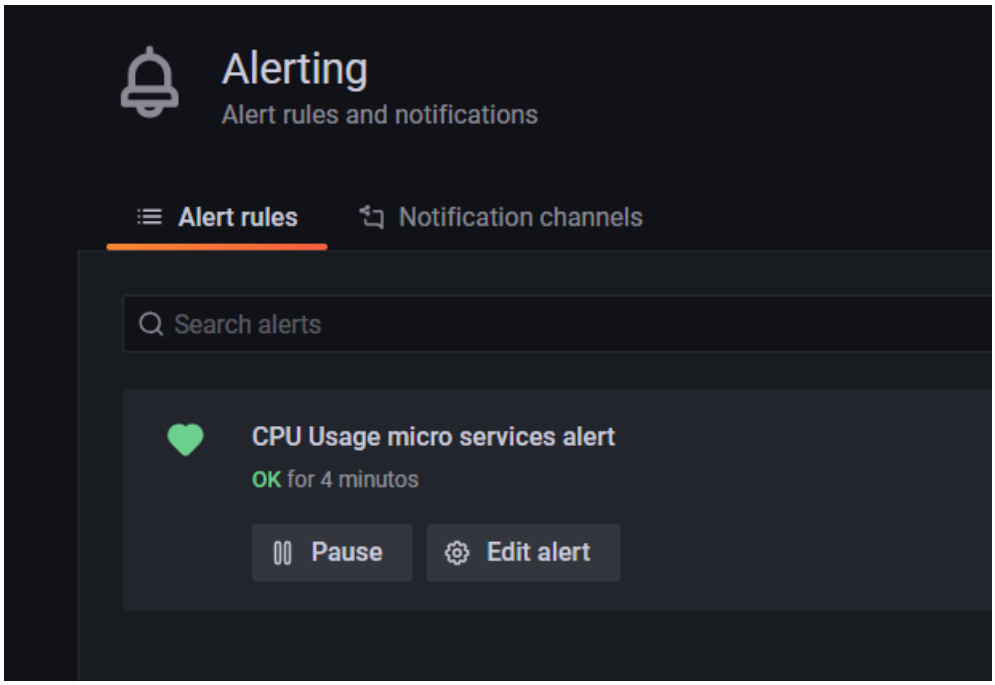

Figura 15: Alerta funcional asociada a la gráfica de uso de CPU del servicio

Ahora en el momento en que el contenedor supere el 90 % de uso de CPU, se activará la alerta y se notificará a Docker (Ver figura 15).

4.4.4. Configuración del auto escalado

Por último, queda lanzar el servicio del host de auto escalado, es un servicio que está escuchando por el puerto una petición webhook.

El servicio instancia una clase en java que detecta la configuración de Docker local y crea un contenedor que ejecute una imagen del mismo(Ver código : 6).

Algoritmo 6 Código central del servicio. Analiza el servicio a replicar y cómo está configurado, para desplegar un contenedor que ejecute el servicio cuando la alerta envíe la orden.

```
DockerClientConfig config = DefaultDockerClientConfig.
    createDefaultConfigBuilder()
.withDockerHost(ip).build();

// Obtiene el servicio
Service service = dockerClient.inspectServiceCmd(serviceName).
    exec();

// Obtiene la configuración del servicio
ServiceSpec serviceSpec = service.getSpec();
```

Cuando se envía la alerta(Ver código : 7), se crea dicho contenedor en nuestro entorno,

con su misma red y con la información de los puertos. Ejecutando el servicio interno que queremos escalar.

Algoritmo 7 Código de la orden de escalado. Despliega un contenedor que ejecuta el servicio a escalar.

```
dockerClient.updateServiceCmd(service.getId(), serviceSpec).
    withVersion(service.getVersion().getIndex()).exec();
```

Capítulo 5

Resultados

Se prueba y se analiza el **flujo de ejecución del auto escalado**. En concreto, realizamos la ejecución del caso de uso de reserva de un producto. Que deberá auto escalarse ya que la reserva consumirá demasiados recursos de CPU.

En cuanto se reserva un producto, el uso del CPU aumenta ya que ejecuta unos métodos de reserva en concreto (Ver figura 17). Esto se muestran en las gráficas de Grafana (Ver figura 18). y provoca que envíen una alerta por el canal configurado (Ver figura 16). El servicio lanzando en el *host* está escuchando por un puerto que recibe esa alerta y ejecuta el servicio de escalado que accede a la API del demonio de Docker para ejecutar un contenedor configurado de manera dinámica.

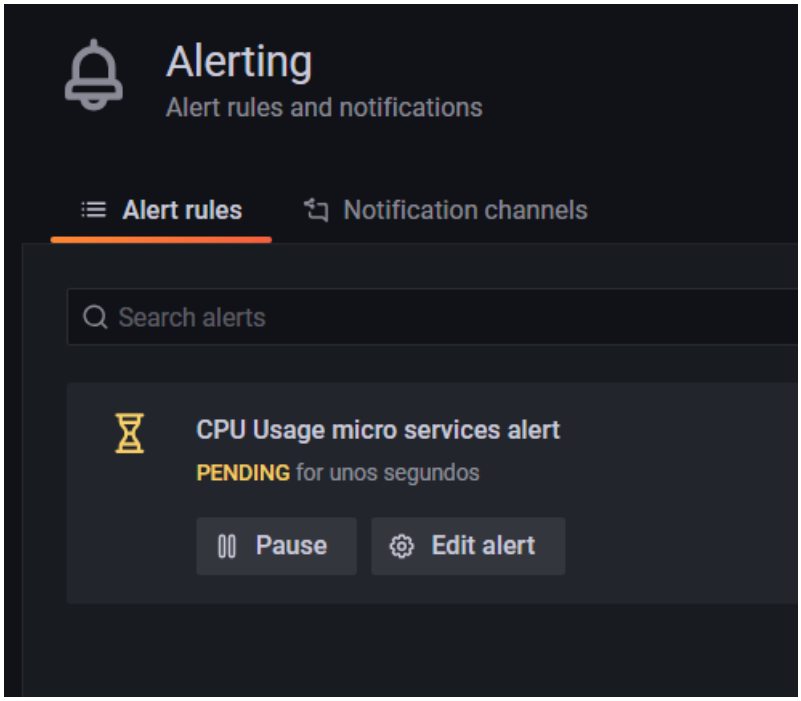

Figura 16: Alerta activada por el caso de uso empleado en las pruebas

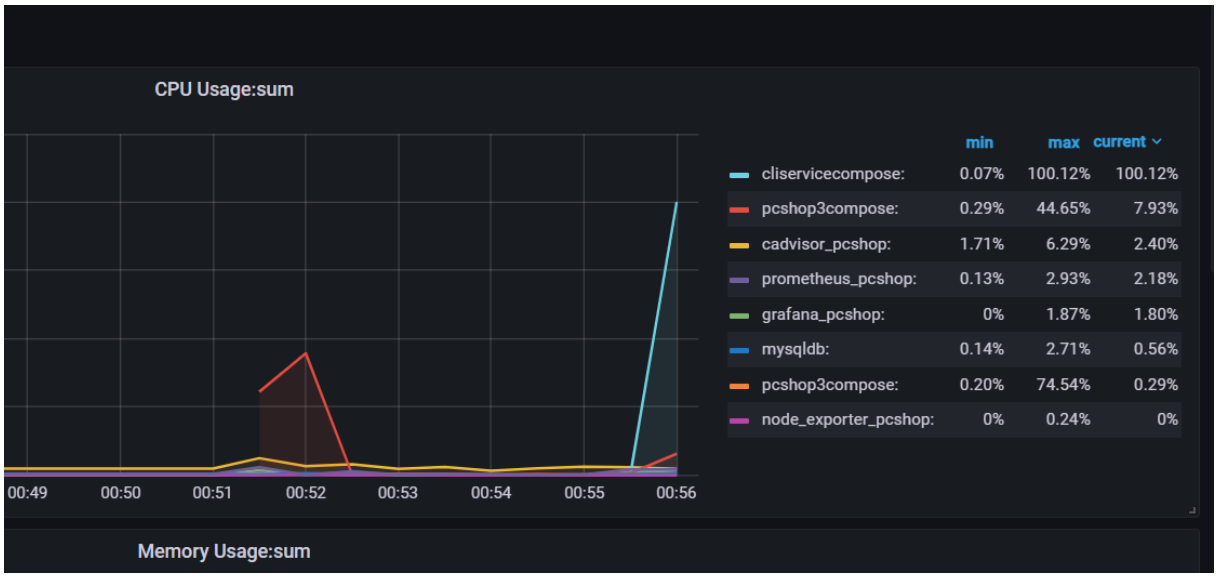

Figura 17: Gráfica de uso de CPU del servicio

La monitorización es constante e inmediata, y podemos observar el uso de CPU tanto en gráfica visual como con porcentajes.

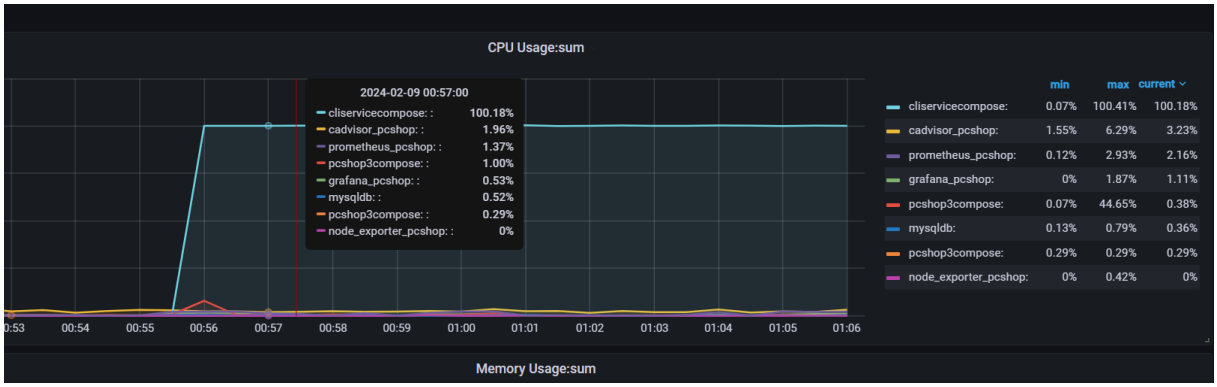

Figura 18: La gráfica de CPU activa la alerta

En el **comportamiento final** observamos un servicio replicado del servicio interno que permite al producto ofrecer las mismas prestaciones aun cuando el servicio se ha quedado ocupado. Además, ya que tiene una configuración de reinicio determinada, Docker se encarga de mantener y reiniciar el contenedor (Ver figura 19). El ciclo de vida del contenedor desplegado por el auto escalado depende de la configuración añadida en la petición del servicio, por lo tanto el mantenimiento del auto escalado se vuelve manual.

☐	cliserviceswarm.1.lazyfr6esmt7z2 9f3c50fa12f3	clientservice2:latest	Exited (143)	0%		6 minutes ago
☐	cliserviceswarm.1.21tm2cc7nsh5l 17978ab48b8b	clientservice2:latest	Running	0%		0 seconds ago
☐	tiendaonline		Running (7/7	101.79%		6 minutes ago
☐	pcshop3compose cec196b03fe9	pcshop3	Running	0.19%	9000:8443	25 minutes ago
☐	grafana_pcshop bd735a0b039a	grafana/grafana:8.0.6	Running	0.02%	3000:3000	25 minutes ago
☐	cliservicecompose 2dcc93b1a11b	clientservice2	Running	100.56%	9100:8500	6 minutes ago
☐	mysqldb 40b44cdd22b4	mysql	Running	0.29%	9001:3306	26 minutes ago
☐	node_exporter_pcshop 61989511d518	quay.io/prometheus/node-exporte	Running	0%	9099:9100	25 minutes ago

Figura 19: Producto con el servicio escalado de manera automática

Capítulo 6

Conclusiones

En este capítulo presentamos las conclusiones alcanzadas al realizar el Trabajo Fin de Grado, así como unas reflexiones personales y posibles trabajos futuros.

6.1. Objetivos logrados

En este trabajo se ha logrado implementar una aplicación basada en microservicios ejecutados en Docker. La aplicación contiene un componente de auto escalado configurado según unas métricas obtenidas por un microservicio integrado.

- Se empleó una aplicación de **tienda online interactiva y funcional** que simula el caso de uso de reservar un producto.
- Se emplearon **microservicios tipo imagen de Docke**r desplegados en contenedores Docker.
- Se utilizaron estos microservicios para **monitorizar** y comunicar el rendimiento del servicio interno que implementa el caso de uso.
- Se codificó y se desplegó un servicio que escucha las alertas del microservicio de monitorización y **escala de manera automática** el servicio con dicha necesidad.

6.2. Reflexiones

Lo más complicado es aprender sobre la integración de componentes en el producto funcional. La elaboración de las conexiones ha supuesto la mayoría de los estancamientos durante el desarrollo.

También lo ha sido la configuración de red del producto, la configuración de puertos internos de cada Docker y el mapeo de estos con el puerto de la máquina "host". Por ejemplo, conectar el servicio de reserva de producto. Este envía las métricas cuando realiza su trabajo, permite que estas salgan del puerto del contenedor, se procesen en contenedores de métricas y alertas, y lleguen al servicio de escalado, que envía la petición al demonio de Docker y realiza el escalado.

Durante el desarrollo de prototipos se ha trabajado con algunos de los servicios primero ejecutados en la máquina, sin contenedor. Luego tras comprobar su funcionamiento se han integrado en el Docker-compose, obligando a realizar un nuevo mapeo de direcciones.

Además, se ha investigado y experimentado con la tecnología de Docker Swarm y DinD . Se ha probado un sistema que acceda a la API del demonio de Docker para que cree contenedores de manera dinámica, volviéndose un producto bastante más escalable y avanzado tecnológicamente.

6.3. Trabajos futuros

La producción de una arquitectura escalable lleva a rediseñar el producto de manera que se ejecute y administre en remoto, esto es, diseñándolo para su ejecución en la nube donde la escalabilidad de manera dinámica la gestione una empresa que ofrezca dichos servicios (IaaS).

Si el producto avanza tomará un camino donde la tecnología de Docker queda limitada, siendo necesaria entonces la aplicación de otras tecnologías como por ejemplo Kubernetes.

Kubernetes ejecutado en AWS o Azure suponen el presente de los productos de software de bastantes empresas. Estos productos poseen una escalabilidad delegada a la nube, y de manera dinámica y sencilla la nube administra las necesidades de rendimiento de manera óptima.

Apéndice A

Figuras complementarias

Se exponen figuras que explican pasos realizados de manera complementaria:

Para realizar escalado automático se ha tenido que probar el sistema de monitorización previamente con una aplicación de prueba (ver Figura 20).

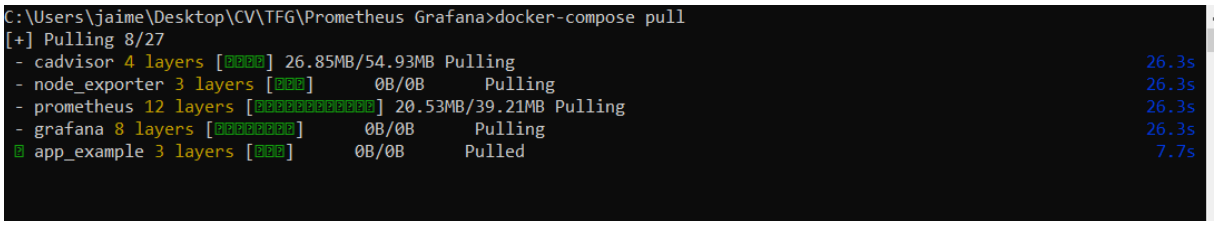

Figura 20: Prueba con el sistema de monitorización

Pero la configuración de la aplicación resultó ser mas complicada, dando lugar a varios errores de conexión consecutivos. En el sistema hubo que configurar tanto prometheus como el resto de servicios, teniendo en cuneta que se ejecutan en contenedores y que por lo tanto cambia la dirección IP (ver Figura 21).

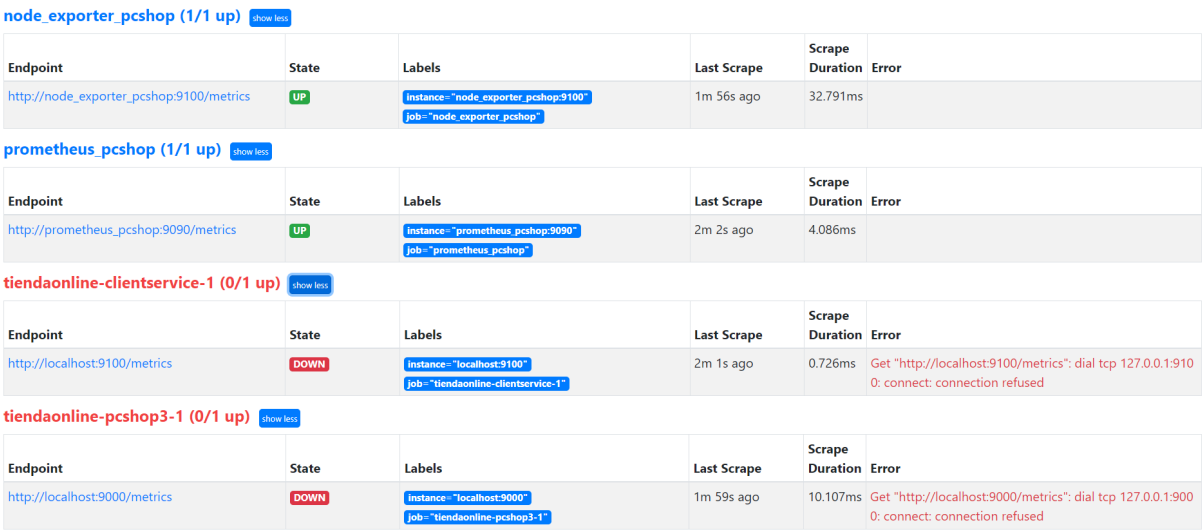

Figura 21: Error en la conexión con Prometheus

Por último, se exponen los comandos donde se ha trabajado con el control de versiones de imágenes de Docker Hub.

```
clietnservice en publico
docker tag clientservice:latest jaimedeonate/public:clientservicetag
docker push jaimedeonate/public:clientservicetag

clientservice en privado
docker tag clientservice:latest jaimedeonate/development:clientservicetag
docker push jaimedeonate/development:clientservicetag

bd en privado
docker tag mysql:latest jaimedeonate/development:mysqltag
docker push jaimedeonate/development:mysqltag

pcshop3 en privado
docker tag pcshop3:latest jaimedeonate/development:pcshop3tag
docker push jaimedeonate/development:pcshop3tag
```

Figura 22: Control de versiones en Docker Hub de las imágenes recurrentemente modificadas

Bibliografía

[1] J. Boulle, A. Crequy y R. Gaubert. *CoreOS in Action*. Manning Publications, 2017. ISBN: 978-1617296272 (vid. pág. 7).

[2] G. Coulouris y col. *Distributed Systems: Concepts and Design*. 5.ª ed. capítulos 1, 2, 3 y 14. Addison-Wesley, 2011. ISBN: 978-0132143011 (vid. pág. 1).

[3] P. Estes, M. Crosby y S. Day. *Containerd in Action*. O'Reilly Media, 2022. ISBN: 978-1492056713 (vid. pág. 7).

[4] K. Matthias y S. P. Kane. *Docker: Up & Running: Shipping Reliable Containers in Production*. 2.ª ed. O'Reilly Media, 2018. ISBN: 978-1492036739 (vid. págs. 5, 7).

[5] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. 2.ª ed. O'Reilly Media, 2021. ISBN: 978-1492034025 (vid. pág. 5).

[6] J. Petazzoni, P. Reznik y J. Duncan. *Containerization with LXC*. O'Reilly Media, 2015. ISBN: 978-1491917572 (vid. pág. 7).

[7] N. Poulton. *Docker Deep Dive*. 4.ª ed. Independently Published, 2020. ISBN: 978-1521822801 (vid. pág. 7).

[8] J. Strong y V. Lancey. *Kubernetes Native: Using CRI-O and Podman for OpenShift and Kubernetes*. O'Reilly Media, 2021. ISBN: 978-1492081654 (vid. pág. 7).

[9] U.S. Bureau of Labor Statistics. *Computer and Information Technology Occupations: Occupational Outlook Handbook*. https://www.bls.gov/ooh/computer-and-information-technology/home.htm. [Consulta: 27 de marzo de 2023] (vid. pág. 1).

[10] D. Walsh y S. McCarty. *Podman in Action*. Manning Publications, 2022. ISBN: 978-1617298306 (vid. pág. 7).