Proyecto Fin de Carrera

Grado en Ingeniería de las Tecnologías de la Telecomunicación

Diseño y validación de una arquitectura escalable en Kubernetes para el despliegue de un SaaS

Autor: Ángel Rodríguez Bohórquez

Tutor: Rafael María Estepa Alonso

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Proyecto Fin de Carrera
Grado en Ingeniería de las Tecnologías de la Telecomunicación

Diseño y validación de una arquitectura escalable en Kubernetes para el despliegue de un SaaS

Autor:
Ángel Rodríguez Bohórquez

Tutor:
Rafael María Estepa Alonso
Profesor titular

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2022

Proyecto Fin de Carrera: Diseño y validación de una arquitectura escalable en Kubernetes para el despliegue de un SaaS

Autor: Ángel Rodríguez Bohórquez

Tutor: Rafael María Estepa Alonso

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2022

El Secretario del Tribunal

A mi familia

A mis profesores

A mis amigos

Agradecimientos

Para comenzar, me gustaría agradecer a mi familia por acompañarme durante todo el transcurso del grado. Sin lugar a duda son los que han vivido mi día a día en la Universidad, y los que han visto mi crecimiento desde que me matriculé el primer año del grado hasta hoy. Su apoyo no ha sido únicamente emocional, sino en todo lo que ha estado en sus manos, con tal de que yo pudiese seguir adelante en mi etapa como estudiante.

También quiero agradecer especialmente a Damián, ahijado de mi madre y doctor en Ingeniería Industrial, el tiempo que dedicó a escucharme, conocer cuáles eran mis inquietudes y aconsejarme estudiar este grado a diferencia de los otros que se ofertan en esta escuela. Fue sin duda un consejo que me ayudó muchísimo en la etapa de decisión sobre lo que quería ser en el futuro.

En tercer lugar, agradecer a los que han sido mis compañeros de viaje durante estos últimos años. Con ellos he compartido momentos que no olvidaré nunca. Nos hemos reído, hemos trabajado para conseguir nuestras metas, nos hemos apoyado cuando más lo necesitábamos y hemos forjado una gran amistad entre todos nosotros. Espero que, aunque nuestros caminos no permanezcan unidos, siempre recibáis todos los frutos que hemos cultivado juntos en el transcurso del grado.

Agradecer también a mis amigos de Espartinas, que me acompañaron desde primara hasta la etapa previa a la entrada a la Universidad, el apoyo y los ratos en los hemos compartido las experiencias que vivíamos cada uno en su día a día. No es fácil acostumbrarse a dejar de ver a la gente que te acompañaba todos los días durante la educación primaria y secundaria.

Ángel Rodríguez Bohórquez

Sevilla, 2022

Resumen

La finalidad de este trabajo es diseñar una arquitectura de componentes escalable que interactúe con su entorno, creciendo o decreciendo según la demanda de tráfico que se le exija. Esta arquitectura va a ser diseñada exclusivamente para un sistema de gestión de agrupaciones musicales basado en microservicios. Para simplificar las tareas relacionadas con el diseño, se elaborará primero una prueba de concepto de la arquitectura que no se escale y más tarde una segunda versión escalable, sobre la que se realizarán una serie de pruebas para validarla.

Existen tres beneficios principales de esta solución: en primer lugar, la disminución de personal necesario encargado de las labores de operación del sistema, en segundo lugar, la optimización de costo que se emplea para mantener el sistema disponible para los clientes que quieran utilizarlo y, por último, la disminución de los tiempos de espera en el diálogo establecido entre el cliente y el servidor.

Una buena comparación del funcionamiento de una arquitectura escalable es la analogía del piloto automático de un coche. Tras fijar un parámetro de velocidad de crucero, el sistema realiza las debidas acciones para que la velocidad se mantenga al valor que el operador ha establecido. En este caso, nuestra variable es la utilización de recursos de procesador y memoria volátil que utiliza el sistema de gestión, y el piloto automático es el encargado de otorgar más recursos al sistema para que el uso medio del procesador y la memoria permanezcan dentro de los umbrales establecidos, mejorando el tiempo de respuesta del sistema.

Además de los beneficios mencionados anteriormente, el sistema poseerá una capacidad de respuesta rápida frente a grandes picos de demanda, es decir, cuando un número elevado de usuarios se conecten simultáneamente a este, manteniendo los niveles de calidad de servicio. Esta capacidad permitirá mantener los niveles de calidad de servicio ofrecidos a los clientes.

La tecnología que se va a utilizar para resolver el problema será Kubernetes, un orquestador de contenedores “Open Source” desarrollado por Google en 2014 y que pertenece ahora a la “Cloud Native Computing Foundation” (CNCF).

Abstract

The aim of this project is to design a scalable architecture based on components that interact with their environment, changing its size depending on the amount of the incoming traffic. The architecture is going to be designed only for an Orchestra Manager system that has been developed with microservices. Stress tests are going to be carried out according to Quality of Service's levels defined, to ensure the architecture validity of the whole system.

There are three main benefits of this approach: first, reduction of the staff required to monitor and operate the system, second, decrease of costs of the system maintenance to keep it always available to clients, and last, to shorten waiting times after requests sent from clients.

To understand the logic behind this architecture, you may compare it with a car autopilot. After the operator sets the speed limit, the system makes the required actions to keep the car speed to the established value. In this case, our system variables are processor and memory resources that the system takes advantage of, and the autopilot is the responsible to provide more resources to the system, so that its measurements will remain between the threshold levels set.

In addition to the benefits already mentioned, the response ability of the system will be faster to traffic peaks, keeping the Quality of Service's levels defined when a big number of users connect to the system at the same time.

The technology to implement the approach will be Kubernetes, an open-source container orchestrator developed by Google in 2014 that is part of the Cloud Native Computing Foundation (CNCF).

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN Y OBJETIVOS

Este trabajo consiste en el diseño y validación de una arquitectura escalable que mejore la respuesta que tendría un sistema tradicional monolítico ante un determinado evento de saturación, donde se produzcan una gran cantidad de solicitudes en un mismo instante de tiempo, llevando a la aplicación a un estado crítico en el que empeorará la calidad de servicio ofrecida a los usuarios.

La arquitectura va a ser aplicada específicamente a un sistema de gestión ofrecido como un SaaS[1], el cual es accedido a través de un navegador web. Este sistema de gestión ha sido diseñado como una aplicación monolítica, por lo que será necesario realizar una adaptación a el formato de aplicaciones basadas en microservicios.

1.1. Contexto y motivación

Hoy en día, con el aumento de la tendencia a desarrollar aplicaciones basadas en microservicios y los avances que se han realizado en la infraestructura sobre la que se despliegan, resulta algo no condescendiente invertir una gran parte del tiempo en el encapsulamiento y no en el diseño de arquitecturas que permitan optimizar los recursos de las máquinas donde acaban desplegándose estos microservicios.

Uno de los principales problemas que se encuentran la mayoría de las aplicaciones monolíticas es la gestión de los recursos del sistema ante un evento de saturación de peticiones. Una aplicación desplegada en una única máquina plantea un escenario complejo con el paso del tiempo, por ejemplo, cuando en algún instante los recursos de esta máquina lleguen al límite y el tiempo de respuesta se vea incrementado en consecuencia.

Es trivial que el aumento del tiempo de respuesta va a influir directamente en la calidad de servicio que se les ofrece a los usuarios finales, llegando al punto en el que los servicios y funcionalidades que ofrezca la aplicación serán inservibles. Por ello, resulta necesario adaptar y encapsular este tipo de aplicaciones en primer lugar a la metodología de los microservicios y posteriormente a una arquitectura escalable.

La arquitectura escalable por diseñar actuará como un lazo de control que determina el crecimiento o decrecimiento de los recursos asignados a la aplicación, en función del uso de los recursos individuales de cada proceso que la componen. De esta manera, se conseguirá mejorar la respuesta ante un evento de sobrecarga y se optimizarán los costes de despliegue de la aplicación al adaptar la cantidad de recursos asignados a la demanda de peticiones de cada instante.

En este trabajo no se realizará el diseño y validación de una aplicación arbitraria, para este caso se ha escogido un sistema de gestión que ha sido diseñado y desplegado como una aplicación monolítica hasta ahora en una máquina con unos recursos limitados. Y que, por su naturaleza de aplicación monolítica, presenta el problema de saturación del que se ha hablado anteriormente.

1.1.1 Sobre el sistema de gestión

Este apartado tiene como objetivo explicar y poner en contexto al lector que es el sistema de gestión y cuál es su finalidad como aplicación SaaS.

En primer lugar, el nombre de sistema de gestión se le otorga al software ofrecido como servicio sobre el que se va a realizar el estudio del diseño y validación de la arquitectura. Como se mencionó anteriormente este trabajo no tiene como objetivo explicar detalladamente el diseño e implementación del propio sistema.

La finalidad que se pretende alcanzar es servir como apoyo a agrupaciones musicales en sus tareas diarias. Para lograrlo, el sistema ofrece un conjunto de módulos: un tablón de anuncios, calendario, almacenamiento de imágenes y obras, incluyendo algunas funcionalidades adicionales específicas para el ámbito musical.

[1] Software as a Service es un modelo de negocio en el que el cliente final no abona por el software, simplemente lo utiliza como un servicio.

Dentro del sistema de gestión (ofrecido como un SaaS), se pueden distinguir dos tipos de usuarios finales: los gestores de la agrupación, que son los que contratan el servicio y se encargan de administrar el sistema en completo; y los componentes de esta, que realizan el uso principal de cada uno de los módulos mencionados anteriormente.

El beneficio, a pesar de que el uso principal del sistema se encuentre en los componentes, se produce en los gestores, ya que, gracias al sistema logran emplear un menor tiempo en sus tareas administrativas dentro de la agrupación, entre las que se incluyen realizar avisos o convocatorias de conciertos, actualizar el repertorio añadiendo nuevas obras, etc. Los componentes a cambio obtienen como beneficio un canal de comunicación único por donde se recibe toda la información relacionada con la agrupación a la que pertenecen, evitando así el uso de otras soluciones que no terminan englobando todas las necesidades de la agrupación.

Este sistema de gestión recibe el nombre de Orchestra Manager y se encuentra operativo desde el año 2018 en 4 agrupaciones musicales diferentes, donde se han realizado las pruebas del sistema completo en producción corrigiendo fallos existentes y añadiendo las nuevas funcionalidades que han surgido como propuestas por parte de todas las agrupaciones.

1.2. Objetivos

El principal objetivo de este trabajo es lograr controlar, optimizar el uso de los recursos y el tiempo de respuesta de la aplicación web a partir del diseño y validación de una arquitectura escalable que modifique su tamaño automáticamente en función de la saturación de peticiones de cada instante.

De esta manera, contemplando la posibilidad de controlar la cantidad de recursos que consume la aplicación, se podrá implementar una lógica que en función de la saturación que este tolerando la aplicación determine si es necesario incrementar los recursos o disminuirlos.

A pesar de que un evento de saturación en la aplicación implique un gran problema en cuanto al uso de recursos y el tiempo de respuesta, existen algunos problemas adicionales que surgen en consecuencia a que la aplicación no disponga de todos los recursos que necesita.

La experiencia del usuario y las condiciones de carrera representan dos grandes problemas que se generan a raíz y que no se consideran la mayoría de las veces cuando se dimensionan los recursos asignados a una aplicación monolítica.

En primer lugar, ante un escenario de saturación el flujo de actividad de un usuario se vuelve incómodo al tener que esperar un excesivo tiempo entre cada cambio que se produzca en la información que se representa. Algo que termina desembocando en que el propio usuario evite utilizar la aplicación de forma natural debido a estas situaciones puntuales.

De la misma manera, ante un escenario de saturación pueden existir solicitudes que se descarten por parte de cualquiera de los nodos intermedios que procesan el tráfico hacia la aplicación, causando que los cambios realizados por un usuario no queden reflejados en la aplicación. Por lo tanto, con el objetivo de mantener la integridad de toda la información es necesario evitar este tipo de casuísticas.

Hasta ahora, se ha hablado a grandes rasgos de la solución necesaria para controlar los eventos de saturación que puede sufrir una aplicación web. En todos los casos se ha especificado que la fuente de los eventos de saturación proviene del despliegue realizado en una máquina con unos recursos limitados. Y, se ha detallado que la solución trivial que existe es aumentar los recursos que se asignan a la aplicación web para que pueda afrontar el evento de saturación.

Dada la arquitectura y la naturaleza de una máquina, resulta imposible aumentar la cantidad de recursos de manera eficaz y lo más importante, sin interrumpir el servicio que se está ofreciendo a todos los usuarios.

De todos modos, a pesar de que el principal problema parezca como aumentar los recursos de una máquina dinámicamente, existe una limitación más restrictiva: cuando una aplicación se despliega en una máquina, está no puede sobrepasar los recursos límite que se han especificado en el momento de despliegue. Por lo que, aunque se consiguiese aumentar la cantidad de recursos dinámicamente sería necesario reiniciar la ejecución de la aplicación, interrumpiendo de nuevo el servicio ofrecido a los usuarios.

Por lo tanto, está claro que la solución no va a ir estrictamente orientada a aumentar los recursos de los que dispone la máquina donde se encuentre desplegada la aplicación, sino que:

- En primer lugar, se van a definir los recursos asignados al despliegue de la aplicación, de manera que se garantice un correcto funcionamiento sin influir en el tiempo de respuesta.
- En segundo lugar, cuando la aplicación empiece a consumir un porcentaje de los recursos asignados se instanciará otra aplicación idéntica, que asuma las peticiones que no puede recibir la primera aplicación.

Este concepto que se ha descrito previamente se conoce como la escalabilidad o flexibilidad de una arquitectura y se consigue agrupando un conjunto de máquinas coordinadas entre sí para compartir recursos. De esta manera, se podrán aumentar los recursos de los que dispone la arquitectura escalable, añadiendo más máquinas al conjunto y se irá replicando la aplicación a través de todas las máquinas.

Sin embargo, aunque replicar la aplicación completa dentro de este conjunto resulte una buena aproximación, no es la más correcta ya que dentro de la aplicación existen varios procesos independientes que tienen un patrón de consumo diferente de los recursos.

Si dividimos la aplicación en un conjunto de procesos que se ejecutan en paralelo, habrá procesos que debido a su diseño o arquitectura podrán afrontar perfectamente un evento de saturación, y otros procesos que supondrán un cuello de botella que ralentizará el tiempo de respuesta en conjunto. Por lo tanto, será identificar los procesos que componen la aplicación y realizar un análisis de flexibilidad para cada uno de ellos.

Realizar un análisis de la flexibilidad de la aplicación consiste en estudiar cada uno de los procesos por separado identificando su comportamiento ante un evento de saturación en un momento determinado. Para que, estos procesos puedan ser replicados paralelamente con el objetivo de afrontar el evento de sobrecarga cumpliendo con los límites de calidad de servicio ofrecidos y replicar únicamente los procesos necesarios.

Una vez descrita la solución principal del problema de saturación, se detallará el proceso a realizar dividido en varias fases:

- Estado de la tecnología actual, donde se detallará el conjunto de tecnologías utilizadas para el diseño actual de la aplicación sobre la que se va a realizar el trabajo y una introducción a las tecnologías para implementar la arquitectura escalable.
- Recogida de requisitos de la aplicación, es decir, del sistema de gestión descrito anteriormente. En esta fase se detallará como se encuentra implementada la aplicación actualmente y los problemas a resolver dentro de la arquitectura escalable, logrando que todas las funcionalidades del sistema de gestión funcionen correctamente. Como resultado se obtendrá el diseño de la arquitectura escalable para sistema de gestión.
- Despliegue de la arquitectura escalable en la nube. En esta fase se explicará el proceso necesario para ejecutar la arquitectura y poder realizar la validación de esta, incluyendo desde el aprovisionamiento del clúster hasta el despliegue de cada uno de los componentes de la arquitectura.
- Validación de la arquitectura escalable mediante pruebas de rendimiento. En esta fase se describirán el tipo de pruebas a realizar y los resultados que se esperan para poder completar la validación de la arquitectura escalable.

1.3. Beneficios adicionales

Al terminar todas las fases y con la correcta validación de la arquitectura se logrará resolver el problema planteado inicialmente, y en consecuencia los siguientes beneficios:

- Esta arquitectura otorgará consistencia y portabilidad al entorno de desarrollo en todas las variantes de escenarios que se puedan contemplar (entorno de pruebas, producción, etc.), ofreciendo un aumento de la eficiencia y facilidad en comparación con otras metodologías como el uso de máquinas para el despliegue de aplicaciones monolíticas. La misma arquitectura podrá ser desplegada en varios entornos sin realizar ninguna modificación significativa.

- La implementación en la nube ofrecerá una gestión centralizada de la aplicación y todos sus procesos, es decir, incrementando el nivel de abstracción del operador al proceso de ejecución de una aplicación sobre una plataforma con recursos limitados.
- Al utilizar la encapsulación de los procesos del sistema en microservicios se mejorará el proceso de desarrollo continuo de la propia aplicación, ya que cada uno de los procesos se encontrará encapsulado en un entorno específico.
- Aumentará el porcentaje de tiempo de disponibilidad para la aplicación completa, ya que al encontrarse replicada en más de una máquina estará siempre disponible, y disminuirán los tiempos de actualización de un componente específico de la arquitectura, debido a que puede ser realizado independientemente en el proceso específico y secuencialmente entre todas las réplicas que se encuentren en ejecución, dejando siempre operativa la aplicación.

2 REVISIÓN DE LA TECNOLOGÍA APLICADA

En este capítulo se van a detallar todas las tecnologías que se han empleado para el diseño del sistema de gestión. A pesar de que el diseño e implementación se encuentra fuera de los objetivos de este trabajo, se considera crucial para realizar una correcta recogida de requisitos.

Una vez se detalle cómo se encuentra implementado el sistema, se pasará a explicar la metodología empleada para dividir los procesos del sistema en microservicios y el proceso de iteración necesario. Para concluir se incluirán las tecnologías disponibles para elaborar el diseño de la arquitectura escalable.

2.1. Tecnologías aplicadas al sistema de gestión monolítico

Con el objetivo de que resulte más sencillo la interpretación del sistema completo, se van a describir todos los procesos que existen dentro del sistema, es decir, las tareas que se desarrollan en paralelo con el objetivo de ofrecer los diferentes servicios y funcionalidades que se encuentran implementadas.

De manera análoga, existirá un apartado dentro de este capítulo donde se incluirá una simplificación del modelo de datos que representa toda la información que se manipula dentro del sistema de gestión, presentado a través de diferentes requisitos de información.

2.1.1 Diseño del sistema de gestión

Este sistema se ofrece a los usuarios de cada agrupación musical a través de un único punto de acceso, es decir, se utiliza un único despliegue para todas las agrupaciones que deseen utilizarlo. De esta manera, el servicio que se ofrece será similar a un servicio de correo corporativo, donde cada usuario selecciona primero su organización e introduce sus credenciales.

Lógicamente, como todas las agrupaciones compartirán los recursos de los que disponga el sistema, será necesario identificar bien la agrupación a la que pertenece cada usuario para mostrarle los requisitos de información que le correspondan con el objetivo de preservar los datos de manera aislada.

2.1.2 Procesos que componen el sistema

Los procesos que forman parte del sistema de gestión son los siguientes:

- Interfaz de usuario, escrita en *React JS* [1] y servida a través de un servidor HTTP estático (*Nginx*).
- API Rest[2], escrita en *Node JS* [2] y servida a través del módulo HTTP de JavaScript conocido como *Express*.
- Base de datos no relacional, a través de la implementación de *MongoDB* [3].
- Base de datos en memoria, a través de la implementación de *Redis* [4].
- Almacenamiento de archivos en red, a través del protocolo NFS[3].

Todo este conjunto de procesos entabla una serie de relaciones y dependencias entre ellos para lograr ofrecer los servicios que dan soporte al funcionamiento del sistema de gestión. Cada una de las relaciones y/o dependencias se establecen por pares, donde se emplea un protocolo de comunicación y una codificación acorde con el tipo de datos que se intercambian.

[2] Un API Rest es una interfaz de programación que se ofrece a través del protocolo HTTP que permite representar las diferentes transiciones del estado de cada uno de los recursos que son ofrecidos a través de esta.

[3] El protocolo NFS (Network File System) permite compartir un sistema de archivos a través de equipos alcanzables por dirección de red. Cada cliente tiene la posibilidad de acceder en modo lectura y/o escritura al volumen que es compartido por el servidor.

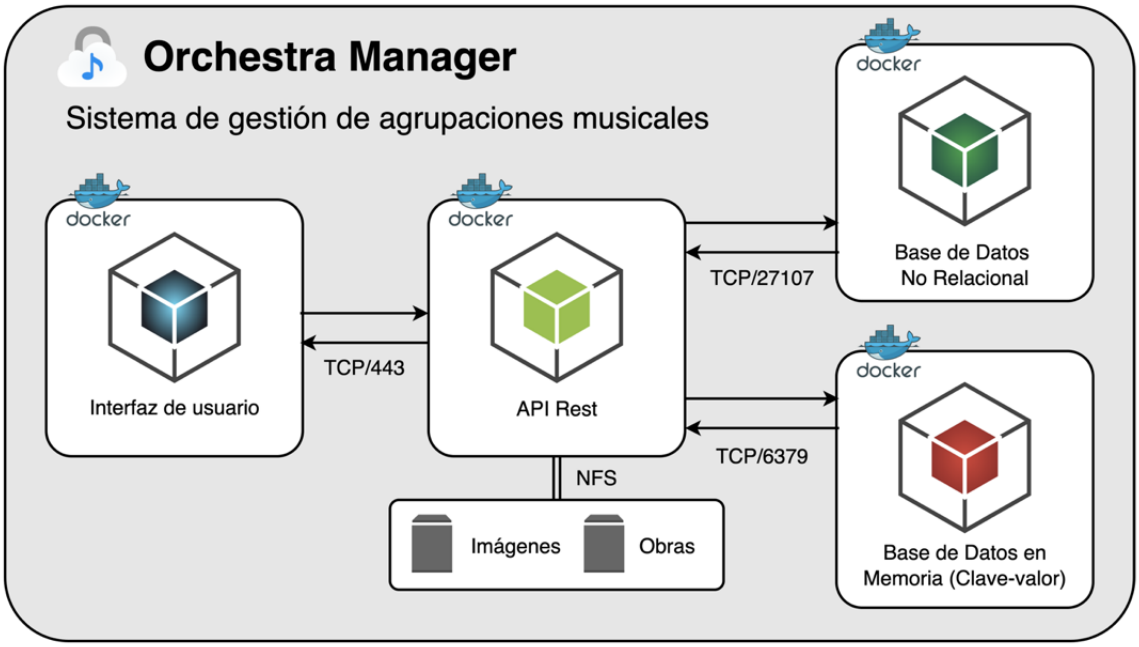

Figura 2-1: Esquema de procesos del sistema.

En el esquema anterior se ha representado a cada uno de los procesos encapsulado dentro de un contenedor, debido a que forma el punto de partida en el proceso de elaboración del diseño y validación de la arquitectura escalable. Las relaciones que se establecen entre cada par de procesos se realizan con las siguientes codificaciones:

- Entre la interfaz de usuario y el API Rest se utiliza el estándar de representación para las transiciones de estado de los recursos (REST), donde cada requisito de información representa un recurso sobre el que se pueden realizar consultas o modificaciones.
- Entre el API Rest y la BBDD no relacional se utiliza un lenguaje propietario de *MongoDB* conocido como *MQL*[4].
- Entre el API Rest y la BBDD en memoria se utiliza un lenguaje propietario de *Redis* que es implementado por las librerías disponibles en cada lenguaje de programación.
- Entre el API Rest y el servidor NFS se establece una comunicación sobre el protocolo estandarizado NFS (RFC1813 [5]).

2.1.3 Técnicas, metodologías y estándares utilizados

Al fin y al cabo, dentro de los procesos que se han definido anteriormente se pueden distinguir dos tipos: aquellos que han sido desarrollados para el sistema de gestión, la interfaz de usuario y el API Rest; y aquellos que se han utilizado como complementos a los anteriores, para resolver problemas ya conocidos.

Los procesos que han sido desarrollados para el sistema de gestión y que implementan la mayoría de las funcionalidades que se ofrecen tienen un contenido extenso, el cual no es objetivo de este trabajo. Sin embargo, para conocer que técnicas y estándares se han seguido en su diseño se va a incluir una pequeña lista explicando a muy alto nivel los recursos empleados.

Los procesos correspondientes a la interfaz de usuario y API Rest implementan:

- La tecnología de comunicación y acceso a recursos a través de una API mediante el protocolo HTTP que sigue el estándar REST. Utilizando como lenguaje de codificación JavaScript con el Framework de NodeJS.

[4] MQL: Mongo Query Language, en español lenguaje de consultas para Mongo.

- El estándar de claves JWT[5] que permite gestionar todo el flujo de autenticación. Para controlar las claves generadas, estas se persisten temporalmente en un servidor de bases de datos "Redis" (almacenamiento de datos en memoria).
- El Framework de React, codificado en JavaScript, que se utiliza para diseñar el aspecto e implementar la funcionalidad del sistema. Servido como una interfaz web de usuario.
- La herramienta de compilación y simplificación para archivos codificados en JavaScript "Webpack". De esta manera, los archivos que contienen el código escrito en JavaScript (utilizando las Frameworks de React y NodeJS) se agrupan y disminuyen su tamaño, logrando así agilizar su interpretación y ejecución.
- Un algoritmo de compresión de imágenes utilizando el índice de similitud estructural [6], implementado por una librería de terceros. Este algoritmo permite generar una previsualización de una imagen, de tal manera que se almacenen la previsualización (con un tamaño muy reducido) y el archivo original.
- Un algoritmo para extraer el nombre de una obra en función de todos los archivos adjuntados. Se utiliza un algoritmo propio complementado por el algoritmo LCS [7], que permite encontrar la cadena de caracteres en común más larga dentro dada una lista con varias muestras.

2.1.4 Servicios y funcionalidades ofrecidas

Los servicios que se ofrecen a los usuarios finales tienen una relación directa con el funcionamiento en conjunto de todos los procesos que conforman el sistema. Los servicios se engloban por módulos, dependiendo de la finalidad, los procesos que utilicen y el tipo de acceso que se asigna a cada tipo de usuario.

A continuación, se detallará cada uno de los módulos en los que se engloban los servicios ofrecidos y una breve descripción de la finalidad:

- Módulo de anuncios: se presenta como un panel donde los gestores pueden crear, modificar o eliminar anuncios en los que se suelen incluir avisos de última hora o información relevantes con la agrupación musical. Los componentes sólo pueden acceder al tablón de anuncios y reciben notificaciones por correo si procede.

- Módulo de eventos: se visualiza como un calendario donde los gestores pueden crear, modificar o eliminar eventos en los que se convoca a la agrupación musical a un ensayo, concierto o cualquier tipo citación. De manera análoga a los anuncios los componentes sólo pueden visualizar eventos. Adicionalmente el día del evento los gestores pueden anotar la asistencia de cada uno de los componentes que han acudido con fines propios[6].

- Módulo de galería: se muestra como un listado de álbumes asociados a un evento concreto que ha acontecido. Los gestores deciden generar una carpeta en función de la naturaleza del evento, y toda la agrupación puede incluir imágenes dentro de la carpeta, creando así un lugar en el que se pueda dar un vistazo a todos los eventos que han transcurrido y recordarlos. Todas las imágenes que se incluyen en cada álbum son procesadas y comprimidas para poder ofrecer una buena previsualización sin perder la calidad de la imagen original cuando el usuario decida descargarla.

- Módulo de obras: se ofrece como un listado de obras, con nombre y compositor, donde los gestores suben las partes[7] de cada uno de los instrumentos y mediante un algoritmo propio a cada componente se le muestran únicamente las partes de su instrumento. Esta funcionalidad resulta clave cuando la obra se ha adquirido con unos derechos limitados y no se pretende que cada componente tenga acceso a todas las partes de la obra, sino que únicamente pueda acceder a la de su instrumento. Adicionalmente, para cada obra se permite adjuntar archivos de sonido, con el objetivo de que los componentes dispongan de una grabación para estudiar sus partes.

[5] JWT, Json Web Tokens, estándar para la generación, encriptación y verificación de claves utilizadas para la autenticación.
[6] En las agrupaciones donde se abona una cuantía económica a los componentes por cada actuación resulta imprescindible tener un archivo con la asistencia que se ha producido en cada evento, con el objetivo de facilitar el proceso de abonar la cuantía a cada componente.
[7] Dentro de la obra, la cual tiene varias voces simultáneas (polifonía), cada parte representa la voz que interpreta cada componente de la agrupación musical.

De los módulos descritos anteriormente, el uso del sistema de archivos *NFS* es únicamente demandado por los módulos de galería y obras, precisamente porque tienen el requisito de almacenar imágenes, partes (archivos PDF) y audios.

2.1.5 Modelo de datos del sistema de gestión

Para cada uno de los módulos que se han visto anteriormente se definen uno o más de un requisito de información, dependiendo de la naturaleza del módulo y del conjunto de servicios que este ofrezca. Este requisito representa al recurso que se ofrece a través del API Rest a la interfaz de usuario. Cada requisito se presentará en una tabla a continuación:

Tabla 2-1: Requisito de información de anuncio.

Nombre	Anuncio
Descripción	Representa un aviso que los gestores de la agrupación musical quieren trasladar a cada uno de los componentes de esta.
Datos específicos	• Título • Contenido • Fecha de creación • Autor

Tabla 2-2: Requisito de información de evento.

Nombre	Evento
Descripción	Representa una ensayo o actuación que se planifica en la agrupación con una fecha, hora y lugar determinados. Este evento tendrá asociado de manera indirecta una galería de fotos y un control de asistencia, siempre en función de si el gestor decide crearlos.
Datos específicos	• Identificador del evento • Tipo • Descripción • Fecha de citación • Fecha del evento • Ubicación • Repertorio por interpretar • Gestor que lo convoca

Tabla 2-3: Requisito de información de un control de asistencia.

Nombre	Control de asistencia
Descripción	Representa un control que es realizado por cualquier gestor de la agrupación con la finalidad de anotar las ausencias de los componentes que no hayan asistido al evento programado.
Datos específicos	• Identificador del evento asociado • Fecha de última actualización • Listado de ausencias

Tabla 2-4: Requisito de información de un álbum.

Nombre	Álbum
Descripción	Representa un conjunto de imágenes que se han tomado durante el desarrollo de un evento. Cada componente puede adjuntar todas las fotos que haya realizado, una vez que el gestor habilite la subida de archivos para un evento determinado.
Datos específicos	• Identificador del evento asociado • Gestor que habilita la galería • Conjunto de imágenes almacenados en un sistema de archivos basado en *NFS*.

Tabla 2-5: Requisito de información de una obra.

Nombre	Obra
Descripción	Representa un conjunto partituras interpretables para los instrumentos de la agrupación. En una obra se distinguirán archivos en dos formatos diferentes: audio MP3 y documento PDF.
Datos específicos	• Identificador de la obra • Nombre de la obra • Compositor de la obra • Conjunto de partes y audios almacenados en un sistema de archivos basado en *NFS*.

2.1.6 Paradigma de comunicación entre cliente y servidor

Para contextualizar y realizar la explicación del sistema se han descrito todos los procesos que lo engloban, junto a las tareas desarrolladas por cada uno de ellos y sus relaciones con el resto. En cambio, dentro de este escenario se ha omitido al cliente que accede al sistema y consume los servicios. El principal motivo se basa en que para comprender el funcionamiento del sistema y describir los procesos es necesario situarse a un nivel bajo de abstracción, algo no compatible con el nivel de abstracción necesario para incluir al cliente y usuario final.

Por lo tanto, se necesita simplificar todo el sistema a un único componente que llamaremos servidor, al que el cliente accederá a través del protocolo HTTP a través de un navegador web. Dentro la comunicación se define al cliente como todo el código que se ejecuta en el navegador web: incluyendo vistas (generalmente código HTML y CSS), lógica necesaria para la interacción del usuario (JavaScript) y todas las validaciones necesarias.

En figura mostrada a continuación se describe una comunicación cualquiera representando los extremos de cliente y servidor, donde el usuario inicia sesión en el sistema de gestión y accede al módulo de calendario.

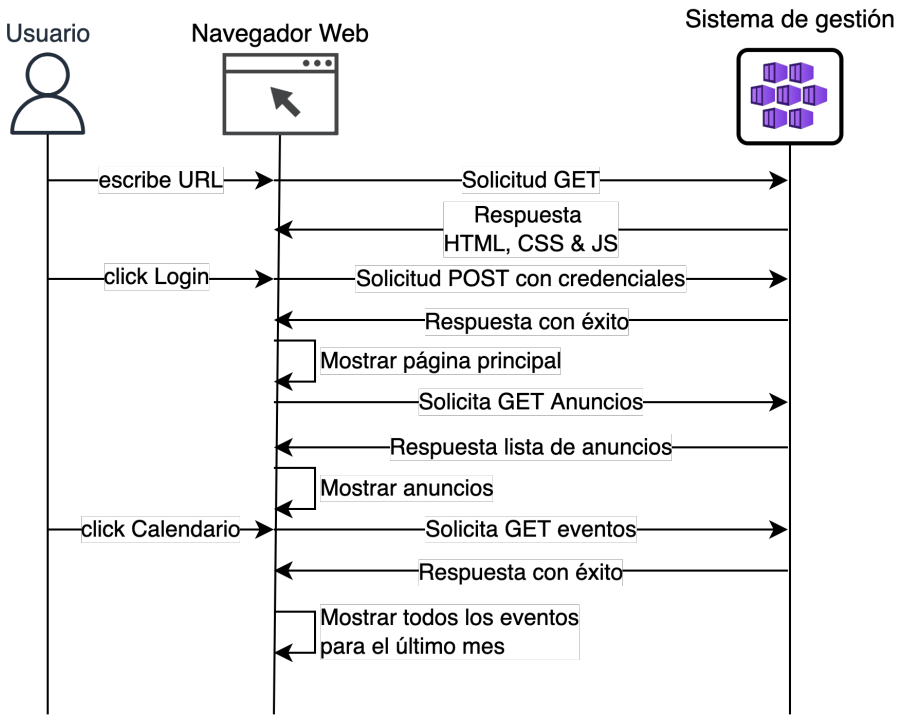

Figura 2-2: Diagrama flujo entre cliente-servidor.

Con este conjunto de tareas asignadas al cliente, se pretende que el servidor se dedique de manera única y exclusiva a la gestión de peticiones orientadas a los servicios a los que el cliente tiene acceso. Por lo tanto, el servidor solo recibirá y enviará peticiones que contengan:

- Objetos codificados en JSON.
- Archivos en formato PDF, MP3/WAW y PNG/JPEG (Debido a que existen servicios ofrecidos dentro del sistema que se desempeñan a través de la lectura y/o escritura de este tipo de archivos).

Este modelo, conocido como Frontend – Backend (cliente y servidor respectivamente), establece su diálogo a través del protocolo HTTP, utilizando para ello el estándar REST mediante el envío de información de manera segura TLS[8] y con credenciales JWT en los recursos que sea necesario proteger el acceso.

[8] Transport Layer Security representa un protocolo criptográfico que se encarga de asegurar las comunicaciones en una red cualquiera.

2.2. Tecnologías para la encapsulación de los procesos

Una vez identificados los principales procesos del sistema de gestión el siguiente paso es realizar la encapsulación de los procesos en microservicios. Para ello, se detalla la metodología empleada con ese fin.

Al principio es normal pensar que cuanto más aumentemos el número de microservicios del sistema, más correcta será la solución, pero debido a la naturaleza de cada sistema no siempre el punto de partida del proceso de encapsulación admite ese nivel de división. Por lo tanto, para comenzar no resulta un buen objetivo establecer un número mínimo de microservicios en los que granular el sistema., lo más importante es iniciar el proceso permitiendo futuras iteraciones sobre él.

Puede que la primera vez solo se logre identificar el 20% procesos en los que se puede granular el sistema, sin embargo, es un punto de partida adecuado teniendo en cuenta que no será necesario rediseñar la arquitectura al incluir un nuevo microservicio encapsulado.

Cada pieza en la que dividamos el sistema representará un microservicio dentro de la arquitectura, por lo tanto, en este caso nuestra aproximación inicial será encapsular cada proceso definido en el apartado anterior por separado.

2.2.1. Definición de un microservicio

Un microservicio es un concepto que se sustenta en un estilo de arquitectura diferente al enfoque tradicional (monolítico). A grandes rasgos se comporta como una caja negra, pero no con una entrada y una salida única, si no con diferentes flujos de comunicación establecidos con un usuario final u otros microservicios que formen parte de la arquitectura.

El objetivo principal es granular los diferentes procesos que comprendan a una aplicación, para hacerlos independientes y simplificar su complejidad. Toda esta tarea hace más sencillo el propio desarrollo de cada microservicio, sin embargo, complica las tareas involucradas con la coordinación y gestión de estos.

2.2.2. Metodologías de encapsulación

La mejor analogía del proceso de encapsulación es compararlo entre realizar el reparto de una cantidad de trabajo que asume una máquina muy potente y dividir la cantidad de trabajo según la naturaleza y características para repartirlo en máquinas de menor potencia, pero que en conjunto consiguen una mayor eficiencia.

Para lograr una encapsulación adecuada se debe pensar en el principal objetivo que se quiere lograr a través de esta: dividir las tareas u obligaciones desarrolladas por los procesos que existan en el sistema. Por lo tanto, una vez identificados los diferentes procesos que compongan el sistema, es necesario analizar el entorno que requiere cada uno.

Mediante entorno se pretende englobar el conjunto de librerías, tecnologías o lenguajes necesarios para el funcionamiento correcto del proceso. Este entorno debe contener estrictamente lo necesario e indispensable, para optimizar su tamaño.

Una vez se tenga el entorno correctamente identificado y recreado será necesario analizar las dependencias que tenga este proceso con el resto, con el objetivo de dar una solución escalable a esos vínculos externos.

Normalmente, un ejemplo de dependencia podría ser una conexión con una base de datos, en donde se tendría que especificar la dirección o nombre para que el microservicio pueda alcanzarlo. Una aproximación válida podría ser definir determinadas variables o argumentos de ejecución del proceso para que este sea totalmente transparente al entorno exterior en el que se encuentre.

Los requisitos más fundamentales para tener en cuenta al encapsular un microservicio son:

- Mantener transparente las variables necesarias de configuración.
- Proporcionar una gestión eficiente de los recursos internos al proceso.
- Gestionar el ciclo de vida del proceso deteniendo la ejecución ante posibles fallos.

Una vez se tenga al proceso debidamente encapsulado es necesario someterlo a un conjunto de pruebas, de

forma que el comportamiento que desarrolle sea el correspondiente a su diseño y conjunto de funcionalidades.

Una de las pruebas más recomendables consta en lanzar el proceso simulando su entorno exterior (en función de sus dependencias) y modificar algunos de los parámetros del entorno exterior (simulando determinadas situaciones) para verificar que el proceso cumple los requisitos anteriores.

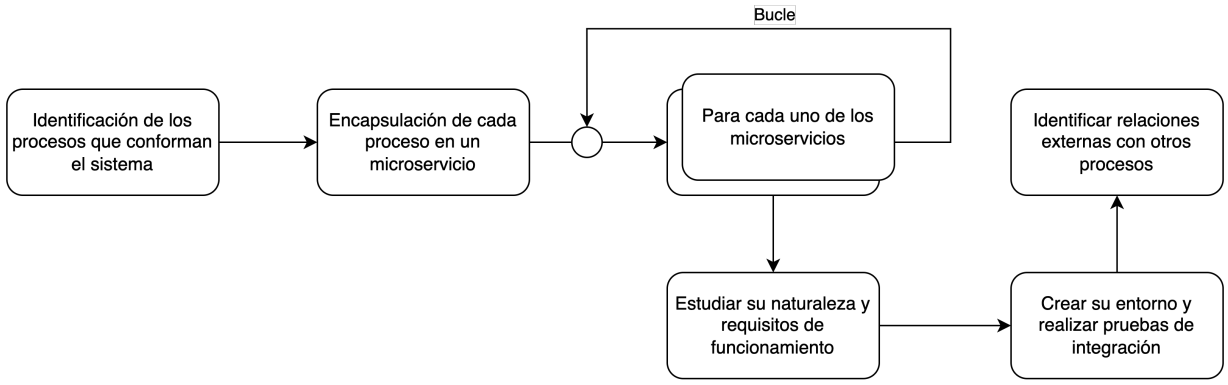

Figura 2-3: Diagrama del proceso de encapsulación.

Adicionalmente, la encapsulación también otorga beneficios como la compatibilidad entre los diferentes procesos que existan dentro de una aplicación.

Por ejemplo, en el desarrollo continuo de algunas aplicaciones monolíticas hay ciertos momentos donde por motivos de vulnerabilidad se ha tenido que actualizar la versión de una librería A que por cuestiones ajenas a la aplicación resulta incompatible con otra librería B. En estos casos, este problema incrementa en gran cantidad el esfuerzo empleado en el desarrollo debido a que la decisión entre preservar la integridad o realizar la actualización que sea necesaria se vuelve más compleja.

2.2.3. Caso real: escenario y conclusiones

A continuación, se va a exponer un pequeño ejemplo sobre los beneficios que aporta la encapsulación del sistema monolítico a varios microservicios. Se describe una aplicación web que utiliza un algoritmo de transformación de archivos de formato CSV a JSON, los usuarios suben el archivo en formato CSV y se les proporciona un link para descargarlo en formato JSON. La aplicación está codificada en Ruby utilizando una clase que se encarga de realizar la conversión cuando se realiza una petición a la propia aplicación.

Se presenta un problema, la conversión que se realiza entre ambos formatos resulta tediosa de codificar en Ruby y se quiere implementar la conversión inversa, algo no muy sencillo dada la limitación de implementaciones en el leguaje escogido. La primera opción es implementar la conversión inversa en Ruby, pero resulta que en el leguaje de Python la conversión se pueden realizar ambas conversiones utilizando la librería llamada Pandas.

Esto lleva a pensar en dividir el sistema en dos microservicios, el primero que realice las tareas de servir la aplicación web y proporcionar el formulario para los usuarios; y el segundo que una vez un usuario solicite la conversión de un archivo la realice y genere la salida accesible para que el primero pueda proporcionar el enlace al resultado.

Hasta ahora todo parece fácil, sin embargo, la gran dificultad llega en el momento en el que se plantea el formato de comunicación entre estos dos microservicios que hemos decidido encapsular.

Como se ha especificado anteriormente granular el sistema no es complicado, sobre todo en este ejemplo donde se utilizaría un lenguaje para la tarea donde este es más apropiado. El problema llega al plantear cómo se van a comunicar ambos microservicios con el fin de colaborar entre ambos y lograr el mismo resultado que ofrecería una aplicación monolítica.

A pesar de que resulte algo complejo no es para nada imposible, al contrario, la implementación de ese formato de comunicación puede realizarse a través de la escritura o lectura de un fichero accesible por ambos, una base de datos, un sistema externo de intercambio de mensajes con colas, etc.

2.3. Tecnologías para la orquestación de microservicios

Tenidos en cuenta los aspectos tecnológicos del sistema de gestión monolítico y de la encapsulación de procesos del sistema a microservicios, en este apartado se van a aclarar las diferentes soluciones existentes para resolver el problema de la orquestación de los microservicios. En los apartados previos se ha deducido que la solución ante un evento de saturación es desplegar cada uno de los procesos de manera replicada dentro de un conjunto de máquinas coordinadas entre sí.

Claramente, esto no es una tarea sencilla, principalmente es necesario controlar cada una de las instancias desde un núcleo que se encargue monitorizar la saturación instantánea y responder aumentando o decrementando el número de instancias de cada proceso. Además, es necesario dar un soporte a todas las relaciones que existan entre los procesos del sistema, teniendo en cuenta que no siempre se encontrarán en la misma máquina.

Este gran problema dispone de una solución existente, conocida como orquestación de contenedores, entendiendo a los contenedores como los microservicios donde se encapsulan cada uno de los procesos del sistema. Al barrer el mercado buscando las diferentes implementaciones disponibles se encuentra una de ellas que posee una ocupación superior al resto, conocida como *Kubernetes* [8].

Por lo tanto, se va a elegir esta implementación para orquestar los diferentes microservicios que resulten del proceso de encapsulación de los procesos del sistema de gestión. Para aclarar algunos conceptos clave, se van a explicar más detalladamente todos los conceptos involucrados dentro de la orquestación de contenedores.

2.3.1. Concepto clave: el orquestador de contenedores

Un orquestador de contenedores representa el conjunto de microservicios que operan conjuntamente para automatizar el despliegue de contenedores, escalarlo y gestionarlo de una manera eficiente, es decir, abstrayendo al operador a un nivel de objetos. El operador del sistema lanzará el manifiesto[9] de una serie de objetos y el orquestador comprobará su estado actual, y si es necesario aplicará algún cambio para lograr el nuevo estado que ha sido lanzado por el operador.

Como es lógico, el orquestador necesitará vigilar en todo momento todos sus objetos que controle para que permanezcan en el estado deseado, tomando las acciones necesarias en el caso de que cualquiera de ellos dejase de funcionar, sufriese algún problema o recibiese una modificación a raíz de un manifiesto enviado por un operador.

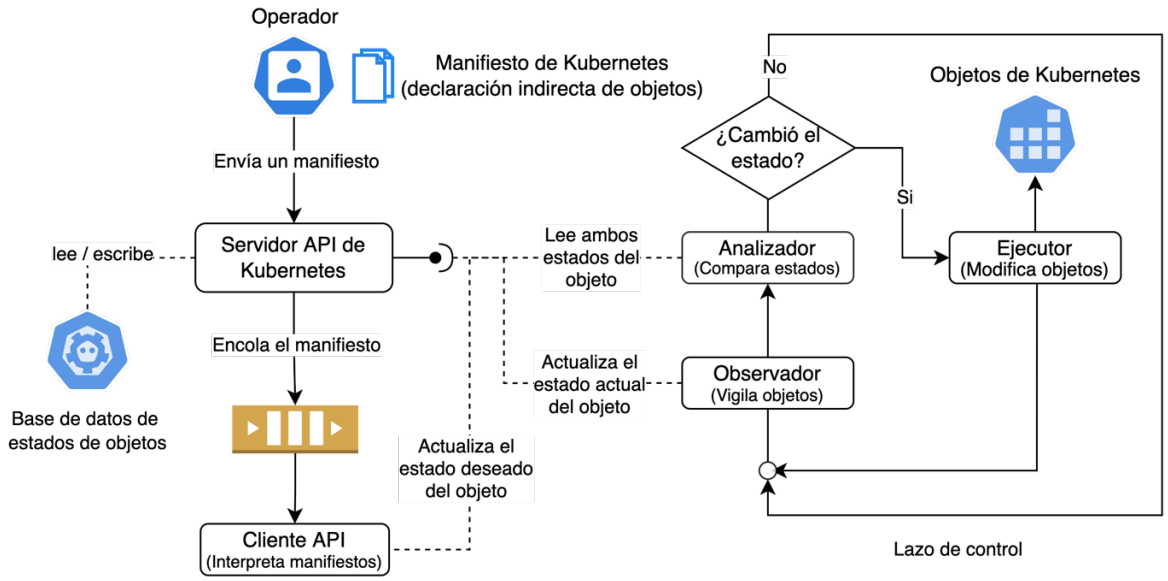

Figura 2-4: Diagrama del proceso de control de un orquestador de contenedores.

[9] Un manifiesto representa el conjunto de directrices y ordenes que el operador encomienda al orquestador para lograr el estado deseado del clúster.

A la infraestructura hardware o conjunto de máquinas (como se ha visto anteriormente) sobre la que un orquestador de contenedores realiza acciones CRUD[10] (como crear, modificar y eliminar) para determinados objetos se conoce como *clúster* o agrupación de nodos.

Los objetos sobre los que se realizan las acciones CRUD se definen como una abstracción de los procesos que se pueden lanzar dentro del clúster, entendiendo un proceso como la tarea relacionada con la ejecución de un algoritmo de IA o el despliegue de un servicio basado en UDP.

Para lograr un control exhaustivo de todos los objetos que se encuentren activos, existen un conjunto de microservicios (conocidos en Kubernetes como plano de control) que se encargan de vigilar estos objetos en cada uno de los nodos y realizar las acciones necesarias para mantener el estado deseado de cada objeto.

Cada clúster se mide por el conjunto de recursos del que dispone, que resultan la suma del número procesadores, la cantidad de memoria volátil y la determinada capacidad de almacenamiento de cada uno de los nodos por separado.

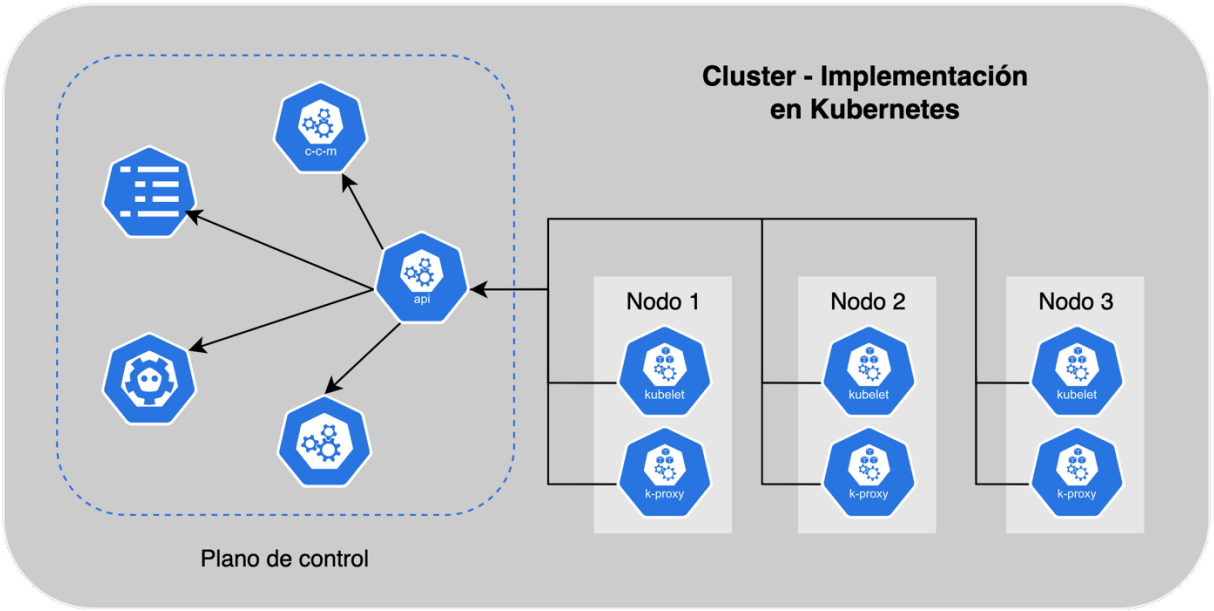

Figura 2-5: Diagrama de componentes de un clúster.

2.3.2. Consideraciones adicionales: tamaño adecuado del clúster

En determinadas ocasiones se puede pensar que la gestión de un clúster formado por un nodo muy potente es más sencilla, que se puede llegar a obtener mejores rendimientos en comparación, pero esa idea resulta todo lo contrario a la verdadera realidad.

La estrategia de diseñar las aplicaciones utilizando microservicios resulta beneficioso en términos de encapsulación, y distribución de tareas entre estos, por ejemplo, si nuestra aplicación se compone de tres procesos principales que se ejecutan paralelamente, se aislará a cada uno de ellos en un microservicio.

Con esto, se consigue que mejoren las actividades relacionadas con desarrollo, la gestión de fallos y rendimiento de cada uno de ellos. Por lo tanto, si a este enfoque le sumamos la capacidad de desplegar cada uno de los microservicios de manera independiente podremos obtener beneficios como:

- Aumentar el número de instancias del proceso que resulta actuar como cuello de botella en nuestra aplicación de manera independiente. Otorgándole a cada microservicio los recursos que necesita.
- Controlar de manera independiente el estado de cada microservicio y poder realizar actualizaciones de manera parcial, evitando para el funcionamiento de la aplicación completa.

[10] Acrónimo "Create, Read, Update, Delete", representa operaciones de creación, lectura, actualización y eliminación de objetos.

- Analizar detalladamente el comportamiento de cada microservicio, con el objetivo de mejorar la implementación y su rendimiento.

Debido a ello, con la misma idea de facilitar la gestión del *clúster*, el objetivo clave es disponer del mayor número de nodos posibles para que cada uno asuma una carga de trabajo independiente y ante un fallo o actualización, se pueda realizar las acciones necesarias en el nodo afectado sin necesidad alterar al resto de nodos. Si sólo disponemos de un nodo, ante un fallo o una actualización será necesario detener la ejecución completa de nuestra aplicación.

2.3.3. Problemas frecuentes: gestión de almacenamiento

Como se ha visto anteriormente, todo son ventajas cuando se dispone de un *clúster* con muchos nodos. En cambio, existe un problema a resolver y es cómo gestionar el almacenamiento a través de todos los nodos. Este problema solo aparece cuando el objeto que se despliega se encuentra instanciado por todo el *clúster* y necesita obligatoriamente que cada una de las instancias compartan el mismo sistema de archivos.

Un claro ejemplo podría ser un servidor SFTP que se encuentra desplegado y replicado en 30 instancias (por cuestiones de alta demanda), al que se accede por un punto de entrada que va redirigiendo cada cliente a un servidor diferente (con el objetivo de distribuir la carga), y que a cada uno de los clientes se espera que le ofrezca el mismo directorio. Lo más lógico es que si un cliente sube un archivo X al directorio, otro cliente que se conecte 10 segundos más tarde pueda acceder a ese mismo archivo y descargárselo.

Aquí llega el problema, ¿cómo se consigue que todos los servidores tengan el mismo directorio accesible?, partiendo de que cada servidor se encuentra encapsulado en un microservicio que esta desplegado en un nodo diferente del *clúster*.

La solución no es sencilla, ni tampoco única, pero una buena aproximación consiste en interconectar cada uno de los nodos a un servidor NFS que exporte el directorio a través de la red que interconecta los todos los nodos del *clúster*. Con esto, se consigue que todos los nodos puedan acceder al mismo directorio utilizando un sistema de archivos distribuido en la red.

2.3.4. Soluciones dentro de un orquestador para escalar contenedores

Dentro del conjunto de objetos pertenecientes a Kubernetes existen dos tipos de escalado: vertical y horizontal. Ambos son dos soluciones al evento de sobrecarga que dispara la demanda de recursos de los objetos que estén instanciados en el clúster, pero tienen una principal diferencia en torno al producto que dimensionan:

- El escalado horizontal consiste en aumentar el número de Pods dentro de un Replica Set. Justo la primera aproximación que hemos planteado anteriormente.
- El escalado vertical consiste en aumentar el número de Nodos dentro del clúster. Es un servicio que se ofrece por cada plataforma de Kubernetes.

Aunque parezca que son soluciones únicas, son totalmente compatibles entre ellas y se complementan. No obstante, dado el alcance de este trabajo se va a utilizar únicamente el escalado horizontal mediante el objeto Horizontal Pod Autoescaler.

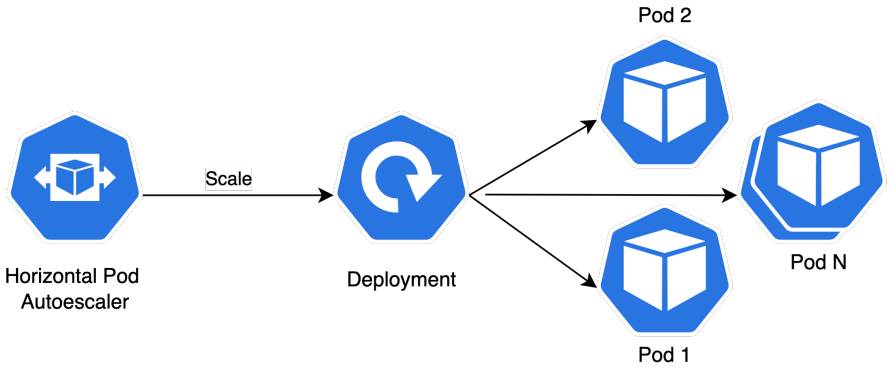

Figura 2-6: Diagrama de arquitectura de un Horizontal Pod Autoescaler.

Ecuación 1: Algoritmo que implementa el Horizontal Pod Autoescaler

$$R_{deseadas} \approx R_{actuales} \cdot \frac{V_{actual}}{V_{deseado}}$$

Donde la "R" representa el valor de las réplicas o instancias, y la "V" el valor que tiene la variable de control elegida. Este objeto que se crea dentro del clúster lleva en su configuración los siguientes parámetros:

- El identificador del *Deployment* que se pretende escalar, que, dada la naturaleza y la herencia de objetos en *Kubernetes* apunta a un objeto de tipo *Replica Set.*
- El número mínimo y máximo de replicas que el *Replica Set* va a manejar.
- Las variables de control y los umbrales necesarios para que el lazo de control tome la decisión de incrementar o decrementar el número de instancias.

3 DISEÑO E IMPLEMENTACIÓN DE LA ARQUITECTURA ESCALABLE

Dentro de este capítulo se van a detallar las etapas que se van a seguir para crear la arquitectura escalable y realizar el despliegue con el objetivo de validarla. El primer paso comprenderá la toma de requisitos del sistema de gestión, con el objetivo de poner en conjunto todas las consideraciones necesarias para encapsular el conjunto de procesos en microservicios.

Antes de pasar a la elaboración del diseño de la arquitectura escalable será necesario aplicar la metodología de encapsulación de procesos, un requisito obligatorio antes de definir los componentes que formen la arquitectura escalable ya que se parte de que el sistema de gestión es una aplicación de carácter monolítico.

A continuación, se realizará una prueba de conceptos no escalable, ya que el paso más complejo de esta etapa se concentrará en la creación de todos los componentes necesarios para la arquitectura escalable. El único motivo por el que se decide omitir el factor de escalabilidad este paso es para simplificar esta etapa.

Cuando se logre la prueba de conceptos para la arquitectura no escalable, se realizará el estudio de los recursos de cada uno de los microservicios para dimensionar los valores de decisión y umbrales necesarios para que el orquestador tome decisiones sobre el número de replicas necesarias.

Por último, una vez se obtengan todos los manifiestos de la arquitectura escalable se pasará al despliegue de esta en un proveedor de servicios *cloud*, con el objetivo de preparar el entorno de pruebas para validar la arquitectura diseñada.

Para facilitar la comprensión de este capítulo se ha incluido un anexo donde se explican detalladamente el significado y el uso de todos los objetos de *Kubernetes*.

3.1. Requisitos del sistema de gestión

Según lo descrito en apartado de revisión de tecnología, el sistema de gestión tiene un requisito fundamental: el almacenamiento de los archivos PDF, imágenes y audios. Este requisito se cumple para el proceso que se desarrolla las tareas de API Rest, el cuál cuando un usuario quiera añadir una imagen a un álbum o descargar una partitura deberá resolver la solicitud mediante el sistema de archivos que tiene disponible.

Cuando encapsulemos y repliquemos el proceso de la API Rest será necesaria dar una solución a este problema: cómo gestionar de manera transparente para el proceso la gestión del almacenamiento para todas las instancias que se encuentren en ejecución.

Al tener separados los dos procesos al que el usuario accede: interfaz de usuario y API Rest, dado que sólo se puede proporcionar un único a la arquitectura desplegada, será necesario implementar un *proxy reverso*[11] que utilice alguna técnica para distinguir las solicitudes que van dirigidas a cada uno de los procesos. De esta manera, mediante un único acceso este componente que actúe como *proxy reverso* pueda dirigir cada petición a el componente que se encarga de servirla dentro de la arquitectura.

Otro de los requisitos fundamentales es la gestión centralizada de claves JWT para la autenticación de los usuarios dentro de los diferentes módulos del sistema. Para ello será necesario que la BBDD en memoria se encuentre sincronizada en el caso de ser replicada.

[11] Un proxy reverso es un servidor que se sitúa en el medio del enlace entre el cliente y una o más aplicaciones web. El cliente realiza una solicitud web, y, en vez de ir directamente al servidor, va al proxy reverso que redirige la petición en función de los parámetros de configuración establecidos a la aplicación web correspondiente. Este tipo de proxy enmascara las diferentes aplicaciones a través de un único punto de acceso.

Por último, para conservar la integridad del sistema, se tendrán que mantener las relaciones de dependencia entre todos los procesos que lo componen, como se han descrito en la revisión de la tecnología.

3.2. Desarrollo de la encapsulación de los procesos del sistema

En este apartado se va a razonar el procedimiento de encapsulación realizado para cada uno de los procesos en los que se ha dividido el sistema. El sistema está compuesto por tres principales procesos, una interfaz de usuario, un servidor web HTTP que actúa como API Rest y un sistema de bases de datos. Por lo tanto, se da lugar a la encapsulación de cada proceso en un único microservicio, exceptuando el sistema de base de datos, donde cada base de datos es encapsulada como un microservicio independiente, al existir ninguna relación de dependencia entre ambas.

Una vez identificados los microservicios que se van a definir dentro de la arquitectura procedemos a detallar los requisitos de entorno de cada uno de estos, es decir, el conjunto de librerías, configuraciones y lenguajes que serán necesarios para que cada microservicio se comporte de manera esperada.

Para ello, definimos cada microservicio junto a su requisito de entorno:

- Interfaz de usuario: aplicación creada utilizando ReactJS servida a través de un servidor HTTP estático. Se realizará la compilación de todo el código fuente para generar los ficheros correspondientes (HTML[12], CSS[13] y JS[14]) que servirá un servidor HTTP con la implementación de Nginx. Este servidor se configurará para que sirva el mismo contenido independientemente de la ruta a la que acceda el cliente.
- API Rest: servidor HTTP creado utilizando NodeJS que conforma la interfaz REST de aplicación a la que accede el navegador para obtener los diferentes recursos disponibles. Se realizará la compilación del código fuente para generar el fichero de entrada que ejecutará el microservicio, abriendo un *socket* para recibir las conexiones entrantes.
- Base de Datos No Relacional: implementación de MongoDB configurado con un modelo de datos diseñado para el sistema de gestión de agrupaciones musicales. Como la implementación se encuentra ya encapsulada por el proveedor no será necesario realizar el proceso de nuevo. Únicamente se modificará el proceso encapsulado para incluir un subproceso (en segundo plano) que se encargue de realizar una copia de seguridad cada 24 horas.
- Base de Datos en Memoria: implementación de Redis configurado para almacenar las claves de acceso JWT (tokens) otorgados a los diferentes usuarios en el proceso de autenticación. De manera análoga, como la implementación se encuentra ya encapsulada por el proveedor no se realizará el proceso de nuevo.

3.2.1. Relaciones de dependencia entre los procesos

De acuerdo con el procedimiento, el siguiente paso es identificar todas las relaciones externas de cada uno de los microservicios. Para ello, de manera análoga al diagrama se irá iterando por cada uno de los microservicios.

En primer lugar, la interfaz de usuario es el punto de acceso por donde el cliente entra al sistema de gestión. Desde esa interfaz, el cliente se comunica con el API Rest para obtener y/o modificar los diferentes recursos disponibles. Por cada solicitud que realiza el cliente al servidor API Rest, este último, realiza las transacciones necesarias para modificar las entradas en la base de datos y/o sistema de archivos.

Por lo tanto, el entorno desde donde se sirve el microservicio que contiene la interfaz solo es accedido por el usuario la primera vez que se descarga la aplicación. Una vez el navegador posee los archivos HTML, JS y CSS necesarios, se encarga de ir actualizando los componentes de la interfaz en función de la interacción del usuario y las peticiones-respuestas que se realicen al API Rest sin solicitar ningún tipo de información al microservicio que contiene la interfaz.

[12] HTML "Hyper Text Markup Language", utilizado para representar contenido web en cualquier navegador.
[13] CSS "Cascade Style Sheets", hojas de estilo para personalizar el contenido web mostrado en el navegador.
[14] JS "JavaScript", lenguaje de programación que se ejecuta en el navegador, capturando la interacción del usuario.

Una de las principales ventajas de esta arquitectura es que se logran dividir las dos grandes responsabilidades del servidor en dos procesos independientes y encapsulados. Por una parte, el envío de HTML, CSS y JS (Microservicio interfaz – Servidor HTTP estático), y el envío de datos en JSON (Microservicio API Rest – Servidor HTTP dinámico).

De esta manera, el servidor deja de asumir la tarea de calcular y enviar las actualizaciones en HTML al navegador web, siendo este el encargado de evolucionar el estado visual de la interfaz en función de las respuestas y datos que le envíe el servidor.

En conclusión, las relaciones existentes entre los microservicios y con los usuarios finales son (entre paréntesis el protocolo utilizado y el formato o codificación de información que se intercambia):

- Flujo entre **navegador web** y microservicio de **Interfaz de usuario** (HTTP – HTML, CSS, JS).
- Flujo entre **navegador web** y microservicio de **API Rest** (HTTP – JSON, JPEG, MP3, PDF).
- Flujo entre **API Rest** y microservicio de **BBDD no-SQL** (TCP - MQL).
- Flujo entre **API Rest** y microservicio de **BBDD clave-valor** (TCP, Redis).

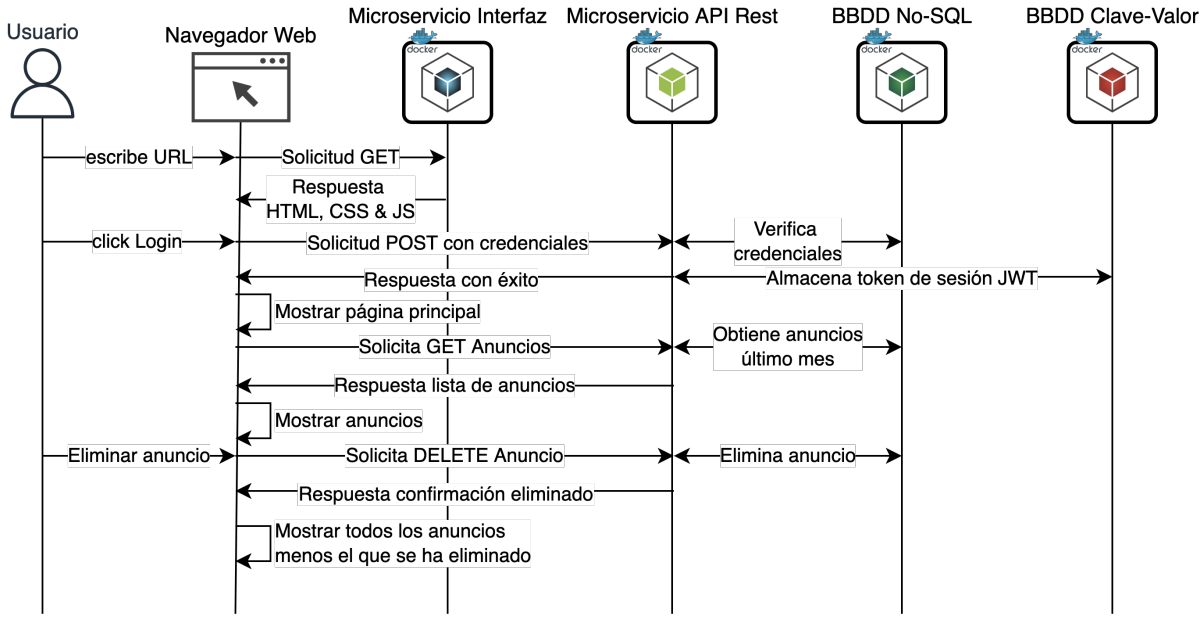

Figura 3-1: Diagrama de secuencia entre los diferentes microservicios.

3.2.2. Solución de encapsulación escogida: contenedores implementados en *Docker*

Al escoger la solución de *Kubernetes* como orquestador de contenedores se determina indirectamente la implementación a utilizar para la encapsulación, debido a que el orquestador de *Kubernetes* es capaz de controlar únicamente contenedores que se implementen mediante la tecnología de *Docker* [9].

La implementación de *Docker* se basa en la encapsulación de únicamente una unidad software o proceso junto a todas sus dependencias con el objetivo de que esta se ejecute de manera rápida y eficaz entre diferentes escenarios.

El único requisito es que el núcleo de *Docker*, conocido como *Docker Engine*, esté ejecutándose en el escenario donde se tenga como objetivo desplegar el software encapsulado.

En nuestro caso, no se utilizará directamente el cliente proporcionado por *Docker* para controlar los diferentes contenedores que se desplieguen, sino que, el plano de control de *Kubernetes* se encargará de manipular los diferentes contenedores en función del estado deseado del operador que envíe los manifiestos al *clúster* que implemente la tecnología de *Kubernetes*. Todo esto fruto de la abstracción que se consigue utilizando un orquestador de contenedores.

Una vez realizado el proceso de encapsulación de acuerdo con el apartado anterior, la implementación de

contenedores en *Docker* exige recoger el conjunto de dependencias y el punto de entrada al software encapsulado a través de un archivo de configuración conocido como *Dockerfile*.

En el sistema de gestión existen 4 procesos principales, de los cuáles 2 suponen un cuello de botella para el sistema completo: el proceso que sirve la interfaz y el proceso que proporciona un API Rest. El principal motivo es la propia naturaleza de los procesos, el uso del protocolo HTTP y que realizan cálculos para transformar los datos que se obtienen de las bases de datos y presentárselos al cliente final.

Los clientes que consumen el sistema no solo realizan una petición, si no que durante todo el uso que hagan del sistema van a realizar múltiples peticiones según los servicios que estén consumiendo. Por ello, el volumen de peticiones siempre va a ser de varios ordenes de magnitud mayor que el número de clientes, contando con el retraso de conexión que experimenten individualmente.

En cambio, los procesos que contienen las bases de datos no relacional y de clave valor utilizan protocolos de comunicación propietarios (diseñados para tolerar una gran cantidad de peticiones) y no realizan apenas cálculos de transformación, almacenan y sirven datos de manera transparente.

3.3. Prueba de concepto no escalable

En este apartado se elaborará una primera aproximación de la arquitectura escalable para el sistema de gestión. Puede resultar algo innecesario, pero dada la complejidad de describir todos los componentes de la arquitectura, se infiere más sencillo el ignorar el objetivo de escalar el sistema, es decir, forzando a que sólo se despliegue una instancia de cada microservicio.

Cómo dentro de *Kubernetes* el nivel de abstracción se encuentra a nivel de *Pod*, es decir, un objeto que contiene uno o más contenedores, se han definido tres Pods principales: “Backend”, “Frontend” y “MongoDB”. Dentro de cada uno de los Pods se encuentran los siguientes microservicios (procesos encapsulados):

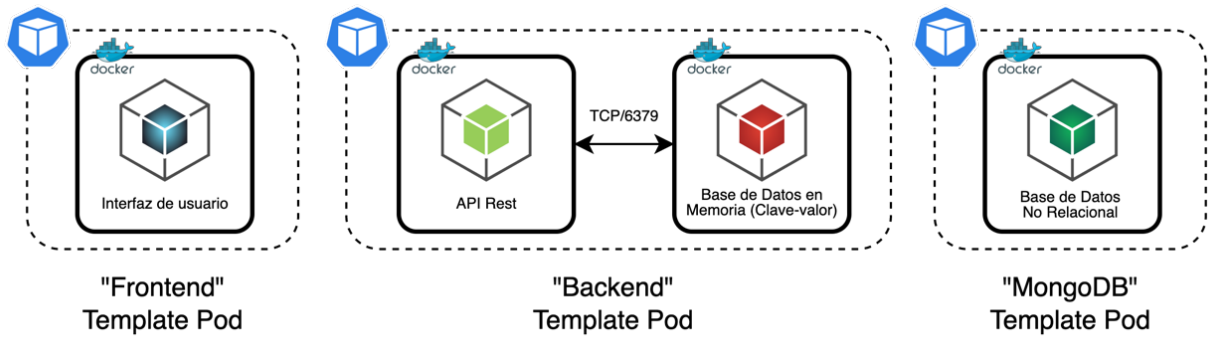

Figura 3-2: *Pods* resultantes de los procesos encapsulados.

La decisión de encapsular los microservicios del API Rest y la Base de Datos en Memoria en el mismo *Pod* viene de la función que desempeña la base de datos en memoria dentro del sistema de gestión.

Los datos que se van a persistir serán las claves JWT que se hayan otorgado en cada una de las sesiones que se creen por cada acceso de un usuario.

Cómo es lógico, si el Pod va a estar instanciado múltiples veces y las claves JWT tienen un tiempo de vida determinado no interesa tener centralizadas todas las claves, simplemente que cada API Rest persista las claves que ha otorgado, con la idea de verificarlas en cada una de las solicitudes.

Una vez disponemos de todos los procesos encargados de formar el sistema de gestión se podría concluir que podemos ofrecer los servicios que se vieron en el apartado, pero en contra del pensamiento inductivo todavía faltan algunos problemas por resolver.

Hasta ahora no se ha tenido en cuenta la solución que se le va a dar al problema de almacenamiento ya que los *Pods* “Backend” y “MongoDB” necesitan consumir volúmenes persistentes.

3.3.1. El problema de los volúmenes persistentes

Dentro de *Kubernetes* existen muchas maneras de alcanzar el objetivo de disponer de un volumen persistente de tipo RWX[15], el cual permitirá que nuestros *Pods* puedan ser replicables y todos puedan tener el mismo volumen montado con permisos de lectura y escritura.

Muchas de las plataformas que ofrecen el servicio de *Kubernetes* tienen la posibilidad de crear este tipo de volúmenes, pero se pretende desarrollar una arquitectura que no dependa estrictamente de la plataforma o clúster que se esté utilizando.

Por lo tanto, se va a optar por utilizar un proveedor de volúmenes definido dentro del clúster. Este proveedor es sustentado por un servidor NFS (similar a la implementación del sistema de gestión monolítico), de tal manera que el servidor montará un volumen persistente de tipo RWO[16], una opción que si está disponible en todas las plataformas que ofrecen *Kubernetes*.

Una vez se instancie el servidor NFS, a través de un *Stateful Set*, se creará una *Storage Class* por defecto que será consumida por todos los *PVC* que se creen dentro del clúster. La arquitectura del proveedor de volúmenes será creada en un Namespace diferente.

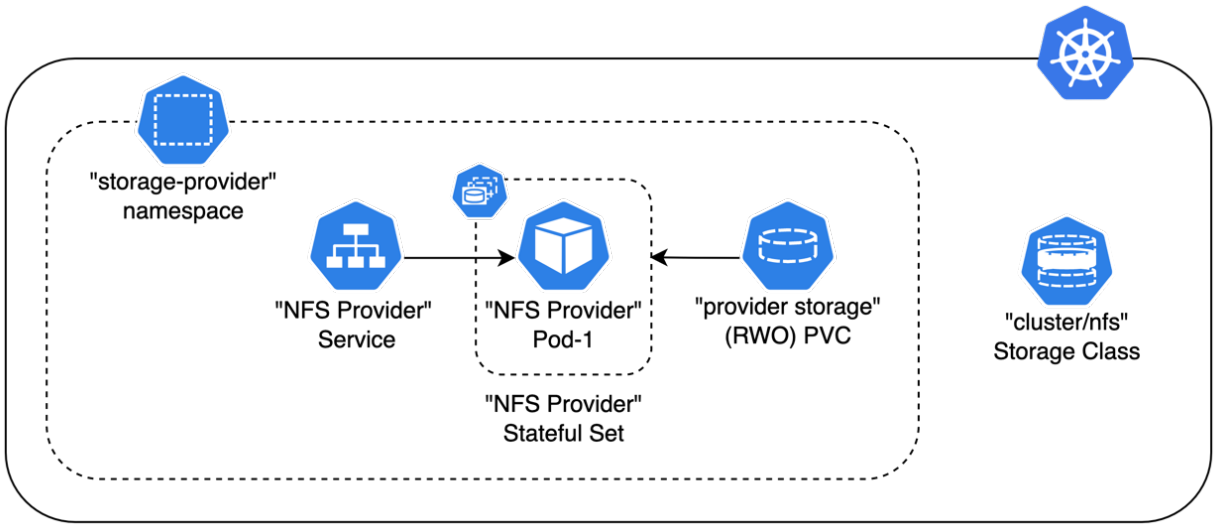

Figura 3-3: Conjunto de objetos para el aprovisionamiento de volúmenes persistentes RWX.

3.3.2. Obtención de certificados *ACME* y balanceo de carga para múltiples instancias

En este apartado se resolverán los problemas de aprovisionamiento de certificados, con el objetivo de ofrecer una conexión segura (encriptada) a los usuarios finales, y se definirá el punto de entrada desde el exterior a nuestra arquitectura implementada en *Kubernetes*.

Uno de los problemas no tan aparente y que se convierte en un problema real a la hora de llevar la arquitectura a un entorno de producción resulta ser el aprovisionamiento de certificados para utilizar el protocolo HTTPS/TLS y la resolución de nombres para acceder al sistema mediante un dominio específico (DNS).

Para lograr este objetivo de forma transparente y de paso configurar el *Ingress* (punto de acceso a la arquitectura implementada en *Kubernetes*) de nuestro sistema utilizaremos la solución que ofrece *Traefik* [10], es decir, un servidor HTTP codificado en Golang que se adhiere al clúster de Kubernetes creando algunos *CRDs*.

Con ello logramos enlazar los servicios correspondientes a los dos procesos que son accedidos por HTTP "Backend" y "Frontend" con el exterior.

[15] RWX es una tipología de los volúmenes persistentes en Kubernetes. Básicamente, un volumen de este tipo puede ser anexionado a múltiples *Pods* con permisos de lectura y escritura.
[16] RWO es una tipología de los volúmenes persistentes en Kubernetes. Básicamente, un volumen de este tipo puede ser anexionado a un único *Pod* con permisos de lectura y escritura.

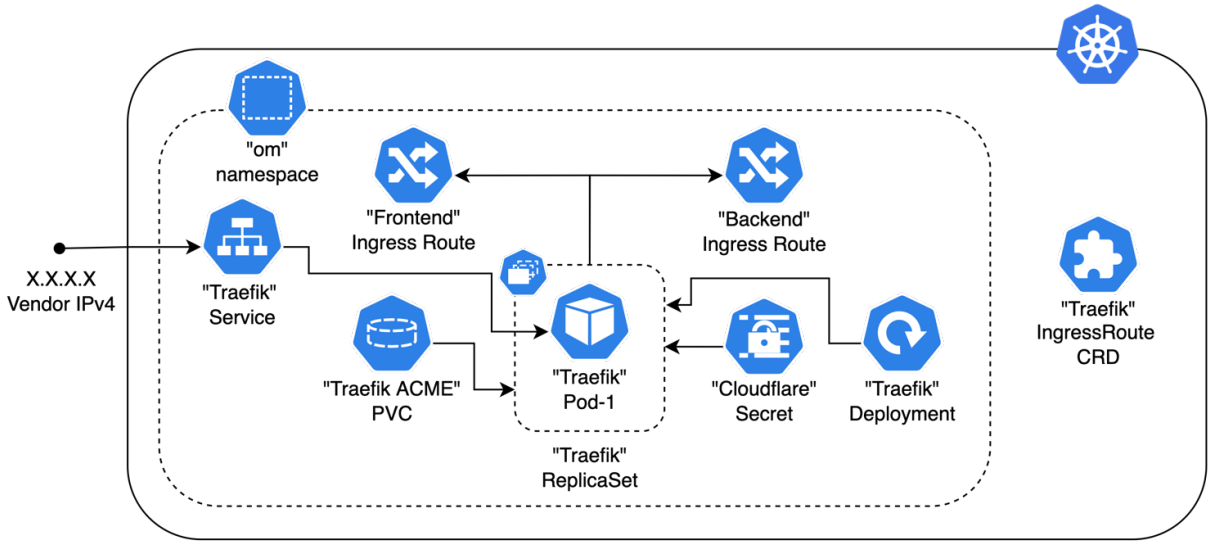

Figura 3-4: Conjunto de objetos para el despliegue de *Traefik*.

Como el punto de entrada va a ser el *proxy reverso* de *Traefik*, en las definiciones de los *Ingress Route* especificaremos la cabecera de tipo Host que permitirá diferenciar al *proxy* que peticiones van al servicio "Backend" o al servicio "Frontend". Mediante esta configuración también se puede vincular el aprovisionamiento de certificados *ACME* utilizando *Let's Encrypt* para que se verifiquen los dominios automáticamente y se generen los certificados en un volumen que consume *Traefik*.

Lo único que necesita *Traefik* para completar el proceso de aprovisionamiento y realizar la verificación de manera transparente son las credenciales del proveedor de DNS. Por lo tanto, la única tarea restante será configurar en nuestro proveedor DNS la dirección IPv4 a la que apuntan los dominios de cada uno de los dos servicios, es decir, la dirección externa que el proveedor de *Kubernetes* asigna al despliegue de *Traefik* al iniciarse (dada la configuración del servicio de *Traefik* como *Load Balancer*).

3.3.3. Persistencia de las sesiones establecidas a través del balanceador de carga

Otro de los problemas que presenta al flexibilizarse es que el conjunto de *Pods* "Backend" utilizan un sistema de claves JWT para mantener la autenticación a los diferentes usuarios que lo consuman.

El problema se basa en que, todas las claves que otorga el proceso API Rest se persisten en la Base de Datos en memoria junto a una fecha de expiración.

Si el balanceador de carga que implementa *Traefik* redirige las peticiones del servicio "Backend" de manera arbitraria a cada uno de los *Pods* que se encuentren instanciados, se producirán ocasiones en las que el usuario se haya autenticado, pero al estar accediendo a un Pod diferente en cada petición, el proceso API Rest que se encuentre encapsulado dentro no sea capaz de verificarlo. Básicamente, porque la base de datos en memoria que se encuentra en ese *Pod* no tiene esa clave registrada.

Para resolver este problema se ha configurado el *Ingress Route* correspondiente al conjunto de *Pods* "Backend" para que genere una *Cookie* (Sticky Mode) en el cliente HTTP donde quede almacenado el *Pod* que se ha encargado de gestionar la última petición. De tal manera que la primera vez la asignación será aleatoria, pero para las peticiones consecuentes el balanceador de carga distinguirá a que *Pod* es necesario reenviar la petición entrante de ese cliente y no será necesario sincronizar todas las bases de datos en memoria que se encuentren en cada *Pod* "Backend".

3.3.4. Detalles sobre las soluciones adaptadas: *Traefik* y el aprovisionamiento de PVC con *NFS*

Tanto el despliegue de *Traefik* como el del proveedor de almacenamiento basado en un servidor NFS se introducen en el clúster utilizando el gestor de paquetes *Helm*, con el objetivo de mantener transparente sus requisitos de despliegue y almacenar sólo la configuración necesaria.

3.3.5. Arquitectura no escalable del sistema de gestión

Para terminar, una vez que todos los problemas previstos se encuentran resueltos, queda diseñar la propia arquitectura del sistema. Al comienzo de este apartado se ha introducido los *Pods* que diseñados para contener todos los procesos que componen al sistema de gestión: “Backend”, “Frontend” y “MongoDB”.

Una vez que se encuentran definidos los *Pods* y se conocen las dependencias que existen entre ellos resulta sencillo trazar el diseño de la arquitectura restante.

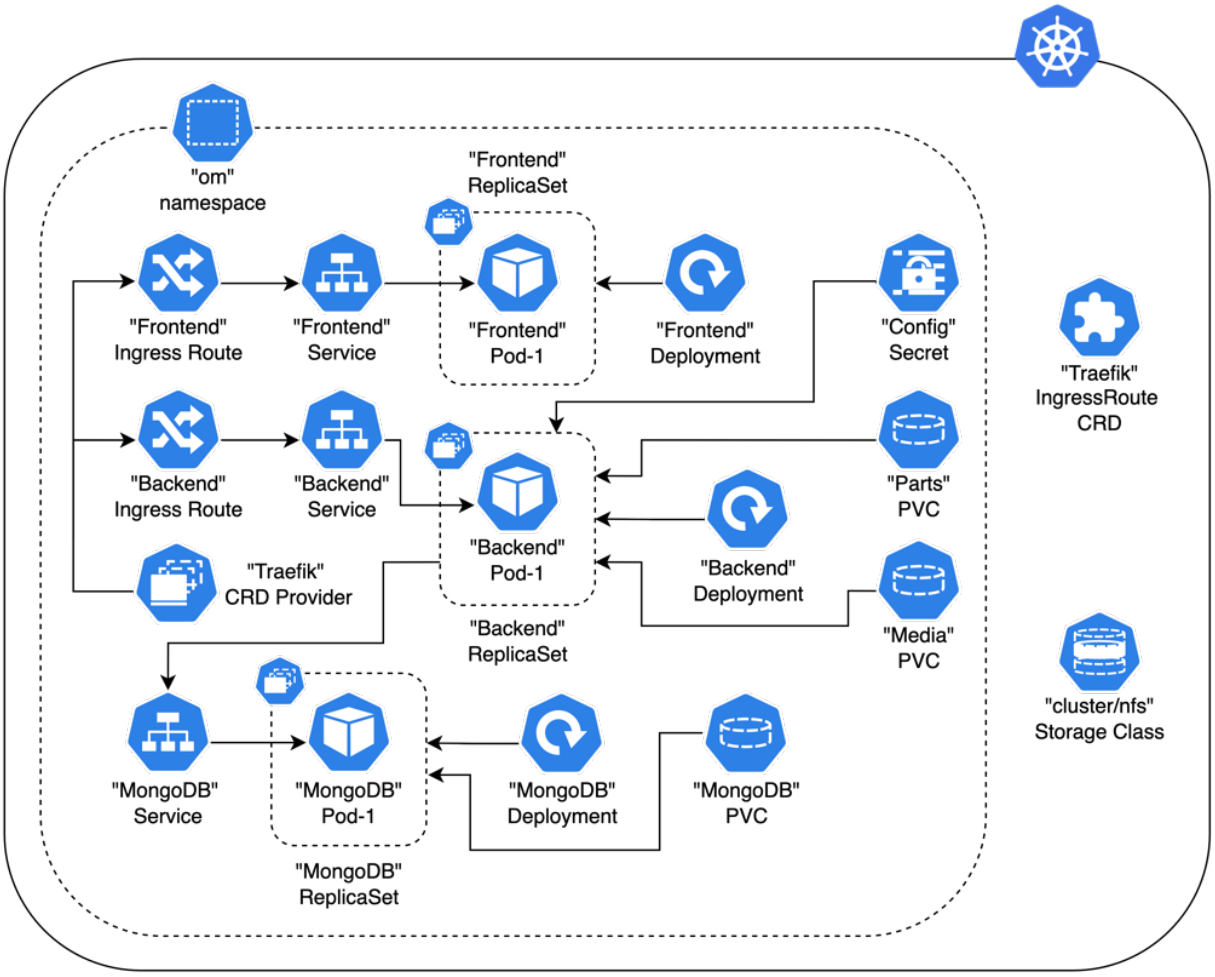

Figura 3-5: Diagrama de la arquitectura no escalable para el sistema de gestión.

Para modificar el número de instancias de los *Deployments* replicables: “Backend” y “Frontend”, simplemente será necesario ejecutar un comando a través el cliente de *Kubernetes* (kubectl). Cada vez que modifiquemos este parámetro el plano de control creará los *Pods* necesarios para alcanzar el valor impuesto.

Este comportamiento parece ser flexible ya que se cumple con la base de aumentar o disminuir el número de instancias de cada proceso de manera transparente, pero no resulta un control práctico en absoluto.

De esta manera, cuando se desee modificar el número de instancias será necesario modificar manualmente ese parámetro. Es una arquitectura escalable, pero no ideal para operar, ya que no se puede responder de manera eficaz ante un evento de sobrecarga.

Con esta prueba de concepto funcional, el siguiente paso será incluir los objetos necesarios dentro de la arquitectura implementada en *Kubernetes* para proporcionar el escalado de servicios de manera transparente y automática.

3.4. Diseño de la arquitectura escalable

En la prueba de concepto donde se elaboró la arquitectura no escalable se han logrado uno de los objetivos principales: flexibilizar la arquitectura del sistema, utilizando una arquitectura diseñada en *Kubernetes*. El único problema que se presenta ahora es plantear cómo controlar el número de instancias de cada uno de los *Pods* de los *Replica Sets* definidos para: "Backend" y "Frontend".

La primer que se viene a la cabeza es plantear un lazo cerrado de control, ya que tenemos una variable llamada número de instancias del *Pod* X. Pero puede resultar algo complejo implementar un lazo de control sobre todo que opere a través del cliente de Kubernetes (kubectl) ya que no resultaría eficiente.

El lazo de control que vaya operando cada uno de los *Replica Sets* debe estar implementado en el plano de control del clúster.

Por lo tanto, se va a utilizar el objeto *Horizontal Pod Autoescaler* dentro de la arquitectura implementada en *Kubernetes*. Como se ha especificado en el apartado de revisión de la tecnología para el orquestador de contenedores, será necesario configurar y parametrizar este elemento para que se comporte de manera adecuada.

Dentro del clúster existe una herramienta de estadísticas que se encarga de recolectar el estado de los dos objetos principales que hacen uso de los recursos como procesamiento y memoria volátil: los *Pods* y los *Nodos*. Que, de manera consecuente son los objetivos de los dos tipos diferentes de escalado que ofrece *Kubernetes*: escalado horizontal y vertical, para *Pods* y *Nodos* respectivamente.

Como el escalado que se va a aplicar es horizontal, se centrará la decisión de las variables en las estadísticas que se recolectan de cada uno de los *Pods*. Estas variables disponibles, las cuales se encuentran disponibles a través de la herramienta de recolección, son el uso de la capacidad de procesamiento en (unidades) y la cantidad de memoria volátil en uso (Megabytes).

En primer lugar, será necesario determinar qué variable de control se va a elegir junto a los umbrales de decisión. Esta variable será la que determine al plano de control de *Kubernetes* la decisión de aumentar o disminuir el número de instancias del *Pod* X.

Aunque es cierto que se podrían elegir otras variables de control, el trabajo de recolección, almacenamiento y servicio de los valores de estas a cada lazo de control resulta una tarea bastante compleja y no eficiente como la propia herramienta que viene integrada con el plano de control de *Kubernetes*.

3.4.1. Definición de las variables de control para el *HPA*

Para que el lazo de control implementado en el plano de control de *Kubernetes* pueda operar correctamente es necesario establecer solicitudes y límites de uso de capacidad de computación y memoria para cada uno de los contenedores que se incluyan en un *Pod*.

La solicitud comprende lo que el plano de control reserva de su capacidad total para ese *Pod* y el límite el valor a partir del cual el servicio se vuelve inestable y empeora su rendimiento.

Tabla 3-1: Reserva y limitación de los recursos para los *Pods*

Nombre	*Pod* "Frontend"	*Pod "Backend"*	
Recurso "CPU"	Reservado: 100m Límite: 100m	Reservado: 100m Límite: 100m	Reservado: 100m Límite: 300m
Recurso "Memory"	Reservado: 100MiB Límite: 100MiB	Reservado: 100MiB Límite: 100MiB	Reservado: 200MiB Límite: 400MiB
Proceso	**Interfaz de Usuario (1)**	**BBDD en memoria (2)**	**API Rest (3)**

En el caso de los procesos 1 y 2 se han utilizado los valores de solicitud de reserva y límites recomendados por el propio desarrollador del software que implementan (marcados color azul claro). En cambio, para el proceso API Rest al procesar más carga de trabajo se le han asignado unos valores más elevados.

3.4.2. Asignación de los umbrales de decisión para el *HPA*

Una vez elegidas las variables de control, sólo queda decidir cuáles van a constituir los umbrales de decisión del lazo de control. Una vez se determinen los umbrales el plano control se encargará de calcular la diferencia entre el porcentaje actual y el deseado para incrementar o decrementar el número de instancias del *Pod* X.

Para cada uno de los lazos en los que se pretende escalar de manera automática las instancias se han definido los siguientes umbrales en función del porcentaje de uso de los recursos asignados:

Tabla 3-2: Umbrales de decisión para los *Pods*.

Nombre	*Pod* "Frontend"	*Pod "Backend"*
Recurso "CPU"	50%	50%
Recurso "Memory"	50%	70%
Proceso	**Interfaz de Usuario**	**BBDD en memoria y API Rest**

A pesar de que los valores de reserva y límite se establecen por cada uno de los procesos/microservicios que se encuentren dentro del Pod, la recolección de valores se realiza por cada uno de los Pods, computando el valor más crítico de todos los procesos en cada recolección de datos.

Para el conjunto de procesos que pertenece al Pod "Backend" se ha utilizado un valor más elevado del recurso de memoria volátil debido a que debido a su naturaleza y tipo de transacciones que realiza necesita disponer de un umbral superior de ese recurso.

De forma resumida, registrar un valor de uso de hasta un 70% en el Pod "Backend" no resulta un evento crítico que implique aumentar el número de instancias del *Replica Set* asociado.

3.5. Despliegue en Cloud

En este apartado se van a incluir el proceso final para desplegar la arquitectura elaborada en una infraestructura ofrecida por un proveedor de *Kubernetes*. Para este caso se ha utilizado el proveedor *Google Cloud*, que ofrece la posibilidad de crear un *clúster* de hasta 4 nodos con el conjunto de recursos mostrado a continuación:

Tabla 3-3: Descripción del clúster aprovisionado para el proceso de validación.

Tipo de nodo	**Número total de nodos**	**vCPU total**	**Memoria volátil total**
E2-Standard-2	4	8	32GiB

Para el aprovisionamiento del *clúster* se ha utilizado el cliente llamado *Google Cloud Console*, que se instala mediante la terminal de comandos de cualquier sistema operativo.

Cuando se realice la instalación, será necesario autenticar este cliente con el proveedor, de manera que se vincule la cuenta que se vaya a utilizar con el cliente.

Una vez se disponga del clúster accesible mediante el cliente de *Kubernetes* (kubectl), se irán lanzando los manifiestos correspondientes a cada uno de los objetos que conforman la arquitectura escalable del sistema de gestión. Que, dada la amplitud de esta, se va creando secuencialmente mediante los siguientes pasos:

- Instalación mediante *Helm* del proveedor de almacenamiento interno.
- Manifiesto para el aprovisionamiento de todos los *PVC*. Incluye todos los volúmenes persistentes definidos en la arquitectura.
- Manifiesto para el despliegue de la BBDD no-SQL (*MongoDB*) e inicialización de la base de datos.
- Manifiesto para las credenciales necesarias para verificar los dominios (*ACME*) obtenidas del proveedor DNS utilizado (*Cloudflare*).
- Instalación mediante *Helm* del *proxy reverso* (*Traefik*)
- Manifiesto conjunto para los despliegues: "Backend" y "Frontend". Incluye los objetos: *Service*, *Deployment*, *Horizontal Pod Autoescaler* e *Ingress Route*.

Para agilizar el despliegue de los servicios que no han sido desarrollados y se utilizan como solución externa para sistema de gestión se ha utilizado el gestor de paquetes *Helm* [11].

Este gestor permite desplegar conjuntos de objetos de *Kubernetes* sin necesidad de almacenar todos los manifiestos, simplemente guardando los parámetros.

Mediante esta metodología se evita almacenar manifiestos innecesarios y se utilizan los definidos por los creadores de ambos servicios: "Traefik" y "NFS Provisioner" (proveedor de volúmenes interno al *clúster*).

3.6. Revisión de los manifiestos correspondientes a la arquitectura escalable

En el proceso de diseño de la arquitectura se han ido incluyendo los diferentes esquemas que conforman todas las partes de la arquitectura escalable para el sistema de gestión. Sin embargo, no se han incluido los manifiestos de cada uno de los componentes que forman parte de la arquitectura para simplificar.

El contenido de todos los manifiestos y configuraciones realizadas para la elaboración y posterior despliegue de la interfaz se encuentran accesibles a través de un repositorio público en [12].

4 PRUEBAS DE RENDIMIENTO

En este capítulo se va a recoger el banco de pruebas que se ha realizado a la arquitectura escalable diseñada en el capítulo anterior, objetivo principal de este trabajo. Como es lógico, a pesar de que la arquitectura se haya diseñado siguiendo los estándares tecnológicos de *Kubernetes*, resulta indispensable validar el comportamiento de esta.

Por lo tanto, se han diseñado una serie de pruebas con ese mismo fin. Considerando cómo objetivo fundamental disminuir el tiempo de respuesta de una petición realizada desde un cliente externo hacia el sistema de gestión. Ya que se considera el valor más adecuado para medir la calidad de servicio global de un sistema ante un evento de sobrecarga.

En primer lugar, se detallará el diseño de las pruebas realizadas y los diferentes escenarios que se han escogido. Una vez comprendida la naturaleza de cada prueba y cómo se ha desarrollado se mostrarán los resultados obtenidos para cada escenario.

Para encomendar la validación del sistema de gestión se utilizará la herramienta de línea de comandos *Apache Bench* [13] que permite realizar pruebas de carga sencillas utilizando el protocolo HTTP. Esta herramienta permitirá sacar conclusiones y valores sobre el comportamiento que tiene la arquitectura escalable del sistema.

4.1. Diseño de las pruebas

Como es deductivo, los procesos del sistema elegidos para realizar las pruebas son los que forman parte de los *Pods* "Backend" y "Frontend", al fin y al cabo, los que son servidos mediante el protocolo HTTP. El resto de los procesos se involucrarán en la prueba a través de estos dos, ya que poseen relaciones de dependencia.

El banco de pruebas se va a lanzar desde fuera del *clúster* que se ha desplegado en *Google Cloud*, con el objetivo de considerar dentro de cada prueba el retraso producido entre cliente y servidor en la conexión establecida. A continuación, se detallarán los tres escenarios escogidos para el desarrollo de las pruebas.

4.1.1 Escenario I: conjunto de procesos en el *Pod* "Frontend"

En segundo lugar, al proceso encapsulado dentro del *Pod* "Frontend" (Interfaz de Usuario) se le va a realizar un único tipo de prueba, ya que a través de la conexión con el servidor solo se sirve la propia Interfaz de Usuario. Una vez el cliente (navegador) completa la descarga de la Interfaz no se establece ninguna nueva conexión, salvo que el usuario refresque la página.

Tabla 4-1: Descripción del escenario *P-FRONT*.

Prefijo del escenario	[P-FRONT]
Descripción	Pruebas de descarga de la interfaz de usuario del sistema de gestión. Para este escenario la secuencia de transacciones del *Pod* serán las descritas a continuación.
Transacciones internas	1. Recibir operación sobre un recurso 2. Obtener clave JWT proporcionada en la petición 3. Comprobar la integridad de la clave 4. Consultar si la clave existe dentro la BBDD en memoria

	5. Realizar la operación sobre el recurso en la BBDD no relacional si la verificación de la autenticación es correcta 6. Devolver el resultado de la operación realizada

4.1.2 Escenario II y III: conjunto de procesos en el *Pod* "Backend"

En primer lugar, a los procesos encapsulados dentro del *Pod* "Backend" (API Rest y BBDD en memoria) se les va a realizar dos tipos de prueba. El objetivo de las pruebas consiste en contemplar todas las combinaciones posibles de transacciones dentro de un proceso. Cómo los únicos tipos de flujo que soporta el proceso *API Rest* son la operación a un recurso con o sin restricción de acceso, se plantean dos escenarios diferentes:

Tabla 4-2: Descripción del escenario *P-BACK-NO-AUTH*.

Prefijo del escenario	[P-BACK-NO-AUTH]
Descripción	Pruebas a un recurso ofrecido que no requiere autenticación. En este caso la secuencia de transacciones dentro del *Pod* serán las descritas a continuación.
Transacciones internas	1. Recibir operación sobre un recurso 2. Realizar la operación sobre el recurso en la BBDD no relacional 3. Devolver el resultado de la operación realizada

Tabla 4-3: Descripción del escenario *P-BACK-AUTH*.

Prefijo del escenario	[P-BACK-AUTH]
Descripción	Pruebas a varios recursos ofrecidos que requieren autenticación. Para este escenario la secuencia de transacciones del *Pod* serán las descritas a continuación.
Transacciones internas	1. Recibir operación sobre un recurso 2. Obtener clave JWT proporcionada en la petición 3. Comprobar la integridad de la clave 4. Consultar si la clave existe dentro la BBDD en memoria 5. Realizar la operación sobre el recurso en la BBDD no relacional si la verificación de la autenticación es correcta 6. Devolver el resultado de la operación realizada

4.1.3 Descripción de las pruebas

Para realizar las pruebas se ha escogido un servidor virtual privado ofrecido por un proveedor de servicios en la nube, que cuenta con un alto ancho de banda y que será accedido mediante el protocolo de conexión remota SSH.

En la configuración que ofrece la herramienta *Apache Bench* se incluyen ciertos parámetros configurables que se irán modificando secuencialmente. Cada tipo de prueba se realizará con 3 configuraciones diferentes, donde se modificará el número de peticiones simultáneas (50, 100, 200 o 10, 20, 40) y se mantendrá la duración del experimento a 240 segundos.

Con el fin de evitar que el balanceador de carga implementado por *Traefik* redirija las peticiones al mismo *Pod*, funcionalidad implementada para que cada cliente tenga asignado siempre el mismo "Backend" y las bases de datos en memoria no tengan que estar sincronizadas, se va a configurar la herramienta Apache Bench para que no almacene *Cookies* y el manifiesto del *Pod* para que se cree por defecto con esa clave JWT en su memoria. De esta manera todos los Pods "Backend" que se creen reconocerán esa clave como válida y procesarán la petición.

Para realizar las pruebas se han configurado a cada *Ingress Route* ("Backend" y "Frontend") con un subdominio diferente, necesario para que el cliente desde donde se está ejecutando la herramienta *Apache Bench* pueda alcanzar a cada uno de los *Services* que *Traefik* expone como balanceador de carga.

Los subdominios escogidos son los siguientes (creados exclusivamente para el desarrollo de las pruebas de rendimiento):

- *Ingress Route* "Frontend" – Proceso Interfaz de Usuario:

https://front.tfg.orchestramanager.pw

- *Ingress Route* "Backend" – Proceso API Rest y BBDD en memoria:

https://back.tfg.orchestramanager.pw

4.1.4 Parametrización de cada escenario

Para cada uno de los escenarios que se han detallado anteriormente se van a parametrizar algunos valores, con el objetivo de generar varios tipos de prueba para cada escenario planteado. A continuación, se mostrará cada uno de los escenarios junto al rango de parámetros asignado para cada uno de estos.

Para comenzar, cómo el *Pod* "Frontend" siempre devuelve el mismo contenido, se ha escogido la *URI* raíz. La configuración de los parámetros será la siguiente:

Tabla 4-4: Parametrización del escenario I.

Identificador de la prueba	P-FRONT-50	P-FRONT-100	P-FRONT-200
Valor de concurrencia	50	100	200
Duración	240 segundos		
URI de acceso	https://front.tfg.orchestramanager.pw		

Continuando con el segundo escenario donde se accede a un recurso privado, se ha escogido la URI de un recurso que devuelve la lista de orquestas disponibles para iniciar sesión. La configuración de los parámetros será la siguiente:

Tabla 4-5: Parametrización del escenario II.

Identificador de la prueba	P-BACK-NO-AUTH-10	P-BACK-NO-AUTH-20	P-BACK-NO-AUTH-50
Valor de concurrencia	10	20	50

Duración	240 segundos
URI de acceso	https://back.tfg.orchestramanager.pw/api/v1/orchestras

Por último, para el tercer escenario se ha escogido la URI de un recurso que devuelve la información de un usuario en concreto "test" (datos personales) en formato *JSON*. La petición incluirá ya una clave JWT válida de sesión para evitar el paso previo de aprovisionamiento de esta.

Tabla 4-6: Parametrización del escenario III.

Identificador de la prueba	P-BACK-AUTH-10	P-BACK-AUTH-20	P-BACK-AUTH-50
Valor de concurrencia	10	20	50
Duración	240 segundos		
URI de acceso	https://back.tfg.orchestramanager.pw/api/v1/user/test/info		
Cabeceras	Authorization: Bearer "jwt_token"		

Como se puede observar, cada uno de los escenarios tiene una parametrización diferente entorno al valor de concurrencia. El motivo principal resulta en el tipo de tráfico que asume cada uno de los *Pods* sobre los que se elaboran los escenarios.

Mientras que el *Pod* "Frontend" es solo accedido para descargar la interfaz de usuario, algo que influye escasamente en el impacto de recursos, el *Pod* "Backend" dialoga con ambas bases de datos incluyendo el impacto que tiene el proceso de transformación de datos que se sirven a través de él.

4.1.5 Recolección de resultados

Para mostrar los resultados de cada una de las pruebas se realizarán varios tipos de gráficas para que sea fácil comparar los diferentes escenarios. Para ello, se utilizará la opción de la herramienta *Apache Bench* para exportar los resultados de los tiempos de respuesta a un archivo de salida.

Este archivo atravesará un proceso de transformación, mediante un "script" en *Python* para generar finalmente los archivos de *GNU Plot* utilizados para crear las diferentes gráficas.

De manera adicional, para visualizar la respuesta instantánea del clúster, se ha elaborado un "script" en *Bash* para obtener el valor de las réplicas actuales y objetivo durante el desarrollo de la prueba.

4.2. Resultados

Con el objetivo de analizar de la manera más realista cada uno de los resultados, se van a algunas gráficas, además de realizar una comparativa con la prueba de concepto de la arquitectura no escalable. El objetivo consiste en analizar los beneficios que se obtienen con una arquitectura escalable de manera automática en contra de una arquitectura que no escala, es decir, las instancias de cada uno de los procesos se mantienen constantes durante toda la prueba.

Por lo tanto, para cada prueba existirán dos entornos diferentes: la arquitectura escalable con una instancia de cada tipo de *Pod* (no flexible o estática) y la arquitectura escalable operada automáticamente. Los resultados que se obtengan de cada una de las pruebas serán representados con diferentes formatos incluyendo:

- Gráfica de dispersión con todas las peticiones realizadas, el instante (desde el comienzo de la prueba) y el tiempo de retardo que han experimentado.
- Gráfica monótona de tiempo de respuesta, se ordenan todas las peticiones realizadas de menor a mayor tiempo de respuesta y se presentan con una línea continua.
- Gráfica con la evolución del número de instancias del *Replica Set* perteneciente al Pod correspondiente.

4.2.1. Resultados de las pruebas sobre el primer escenario

En primer lugar, se comparan los resultados para el mismo nivel de concurrencia (100) y diferente entorno:

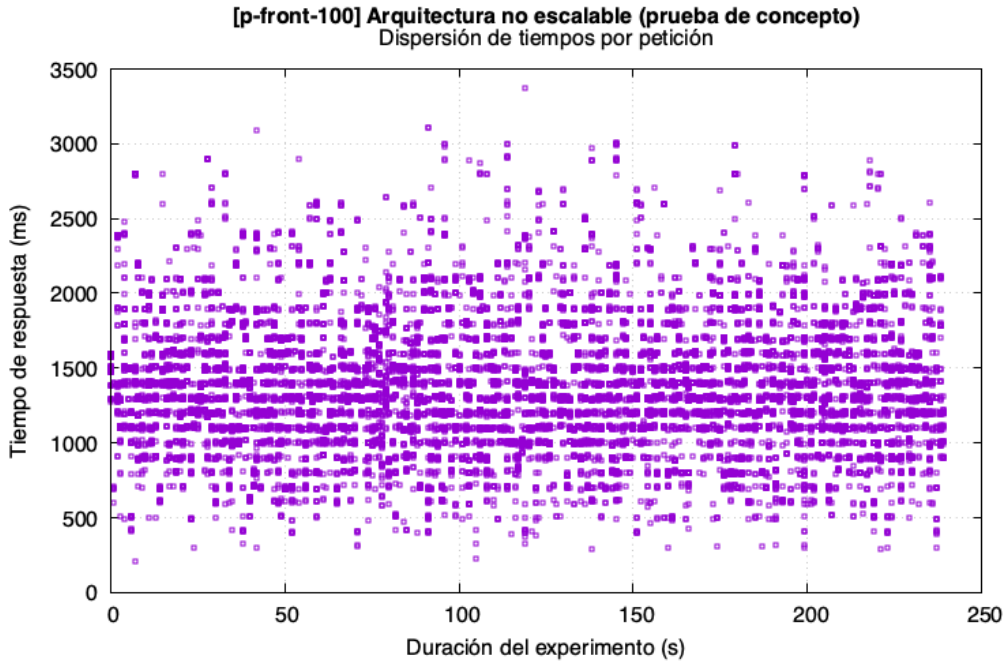

Figura 4-1: Dispersión de tiempos *P-FRONT-100* (Arq. no escalable).

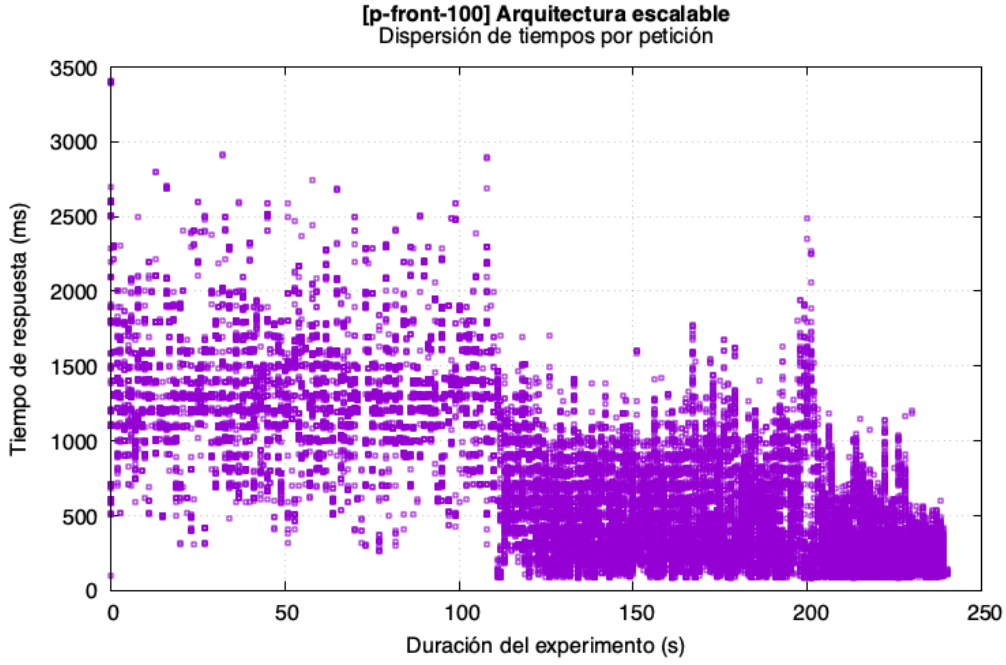

Figura 4-2: Dispersión de tiempos *P-FRONT-100* (Arq. escalable).

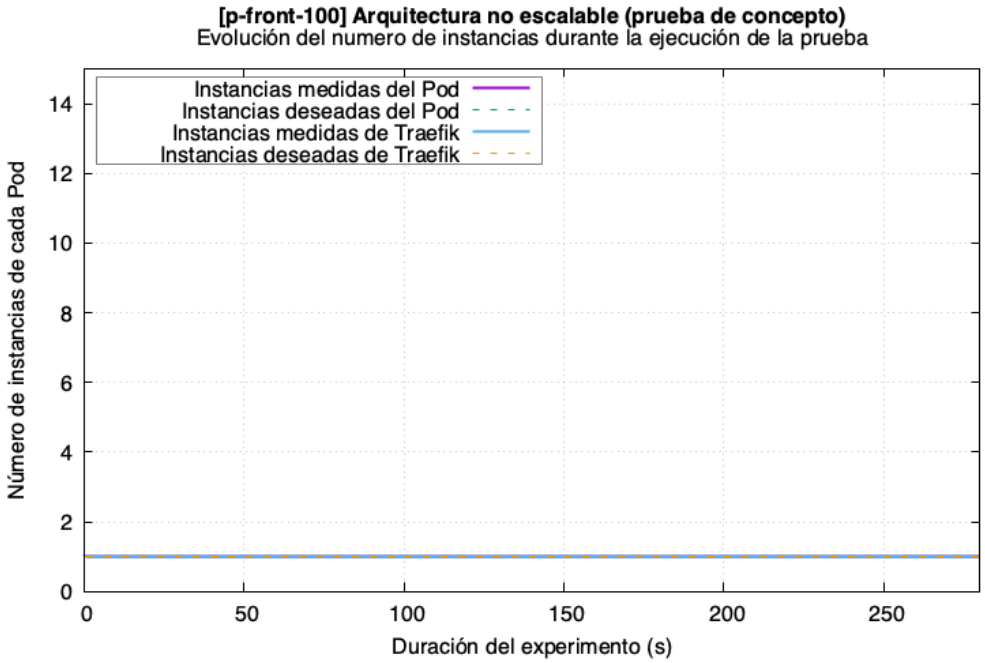

Figura 4-3: Evolución de instancias *P-FRONT-100* (Arq. no escalable).

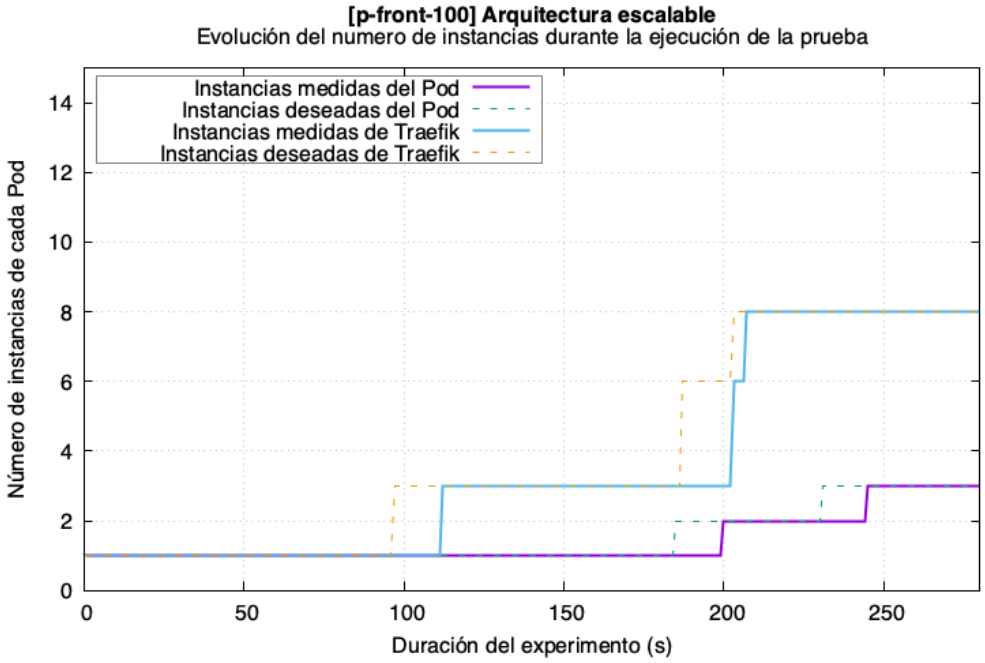

Figura 4-4: Evolución de instancias *P-FRONT-100* (Arq. escalable).

En las dos figuras anteriores se puede apreciar el comportamiento de la arquitectura escalable, que durante el transcurso de la prueba va incrementando su tamaño. En esta figura también se han incluido el número de réplicas del *proxy reverso* (Traefik), que de manera similar al *Pod* sobre el que se realiza la prueba, experimenta una evolución a lo largo de la prueba.

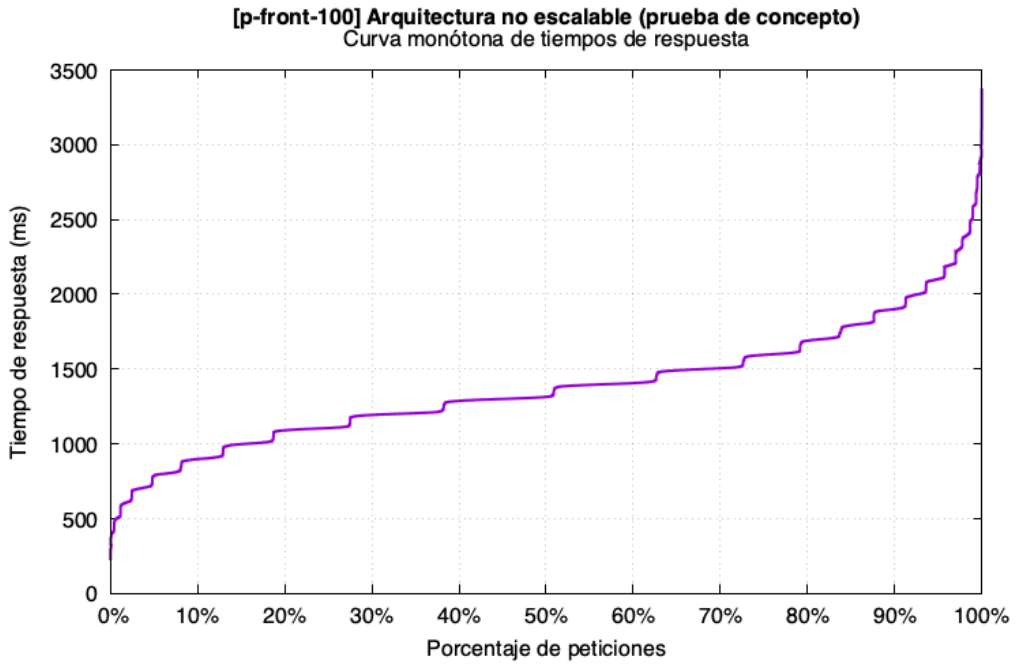

Figura 4-5: Curva monótona de tiempos de respuesta *P-FRONT-100* (Arq. no escalable).

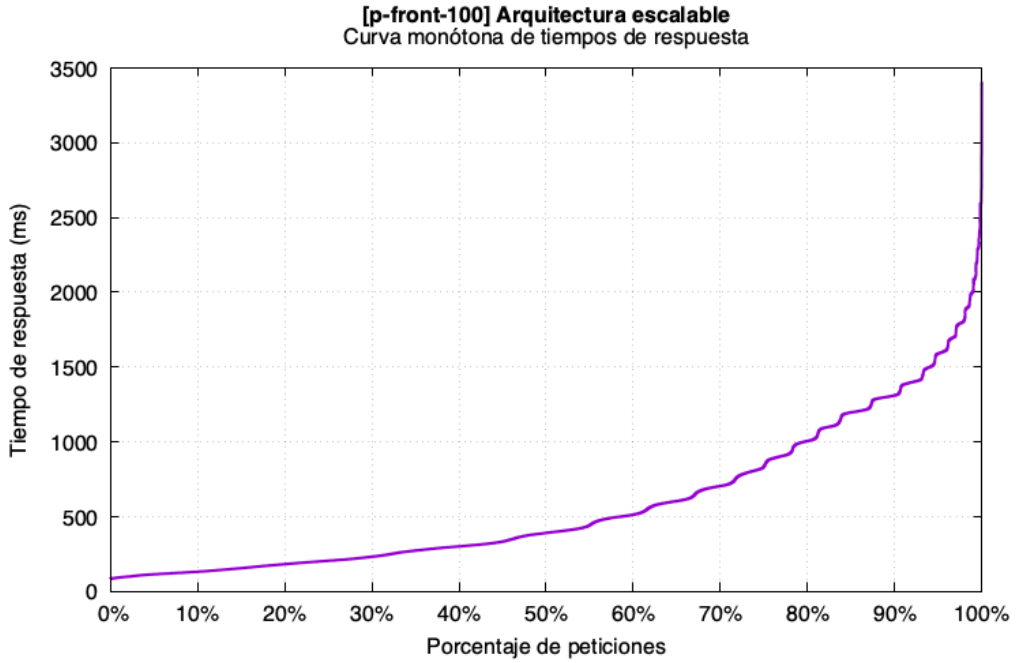

Figura 4-6: Curva monótona de tiempos de respuesta *P-FRONT-100* (Arq. escalable).

En estos resultados se puede visualizar la diferencia en cuanto a tiempos de respuesta cuando entra en juego el lazo de control que va escalando al sistema de gestión.

A continuación, se muestran las parametrizaciones restantes sólo para la arquitectura escalable, con valores de concurrencia de 50 y 200 respectivamente.

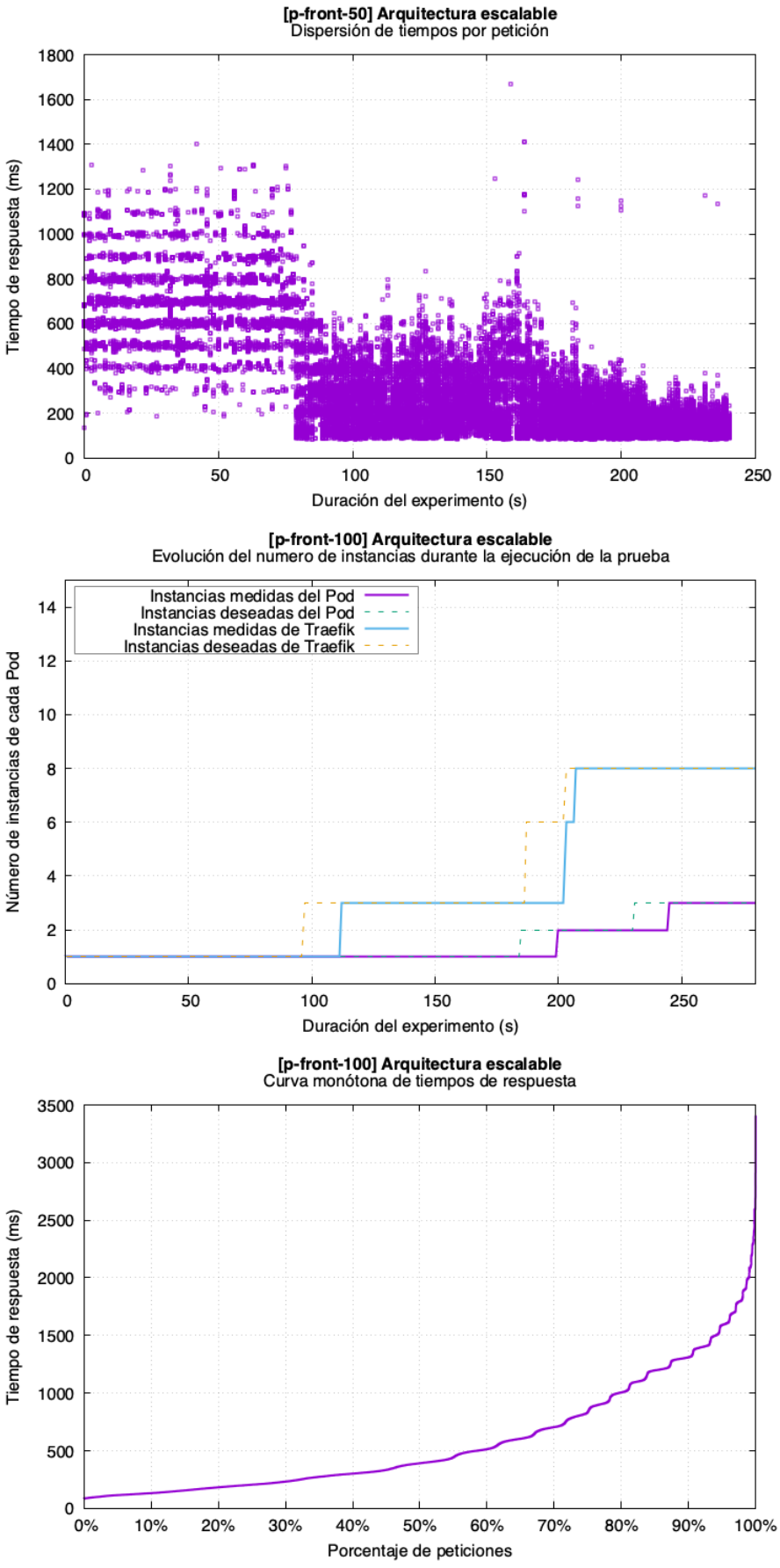

Figura 4-7: Resultados para la prueba *P-FRONT-50* (Arq. escalable).

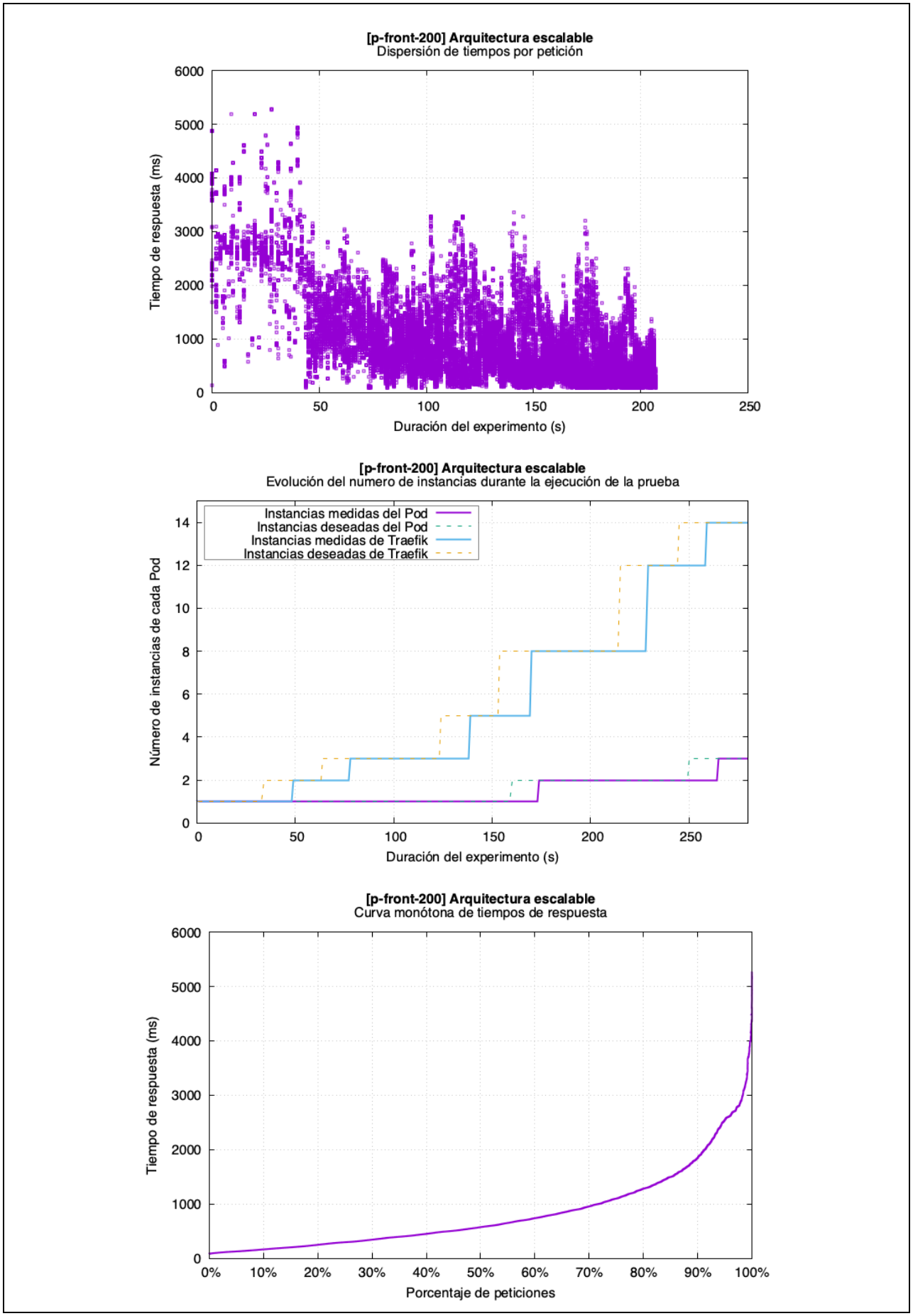

Figura 4-8: Resultados para la prueba *P-FRONT-200* (Arq. escalable).

4.2.2. Resultados de las pruebas sobre el segundo escenario

En primer lugar, se comparan los resultados para el mismo nivel de concurrencia (20) y diferente entorno:

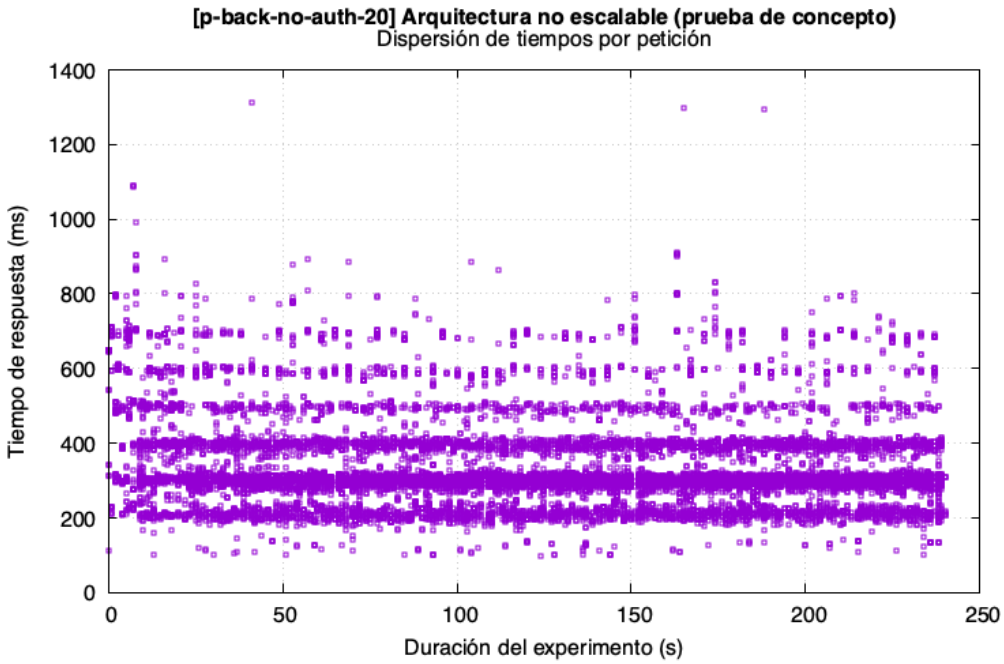

Figura 4-9: Dispersión de tiempos *P-BACK-NO-AUTH-20* (Arq. no escalable).

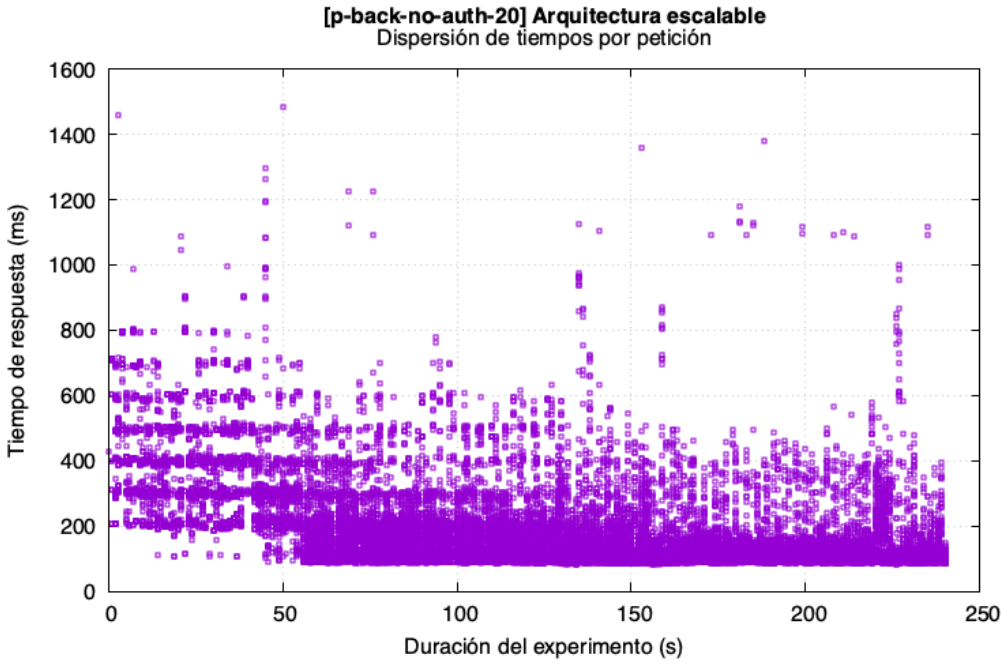

Figura 4-10: Dispersión de tiempos *P-BACK-NO-AUTH-20* (Arq. escalable).

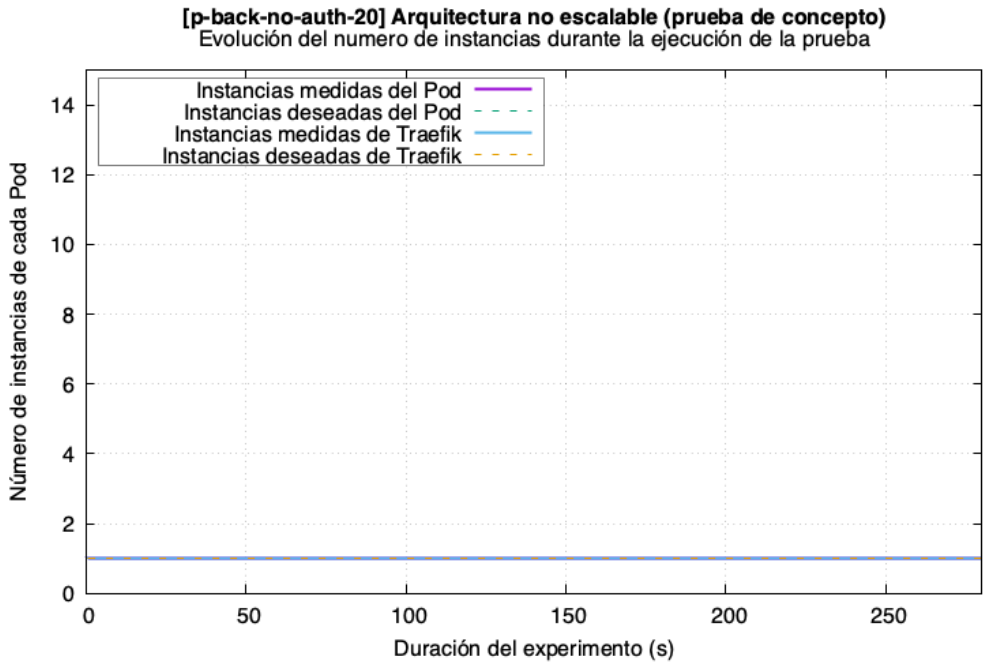

Figura 4-11: Evolución de instancias *P-BACK-NO-AUTH-20* (Arq. no escalable).

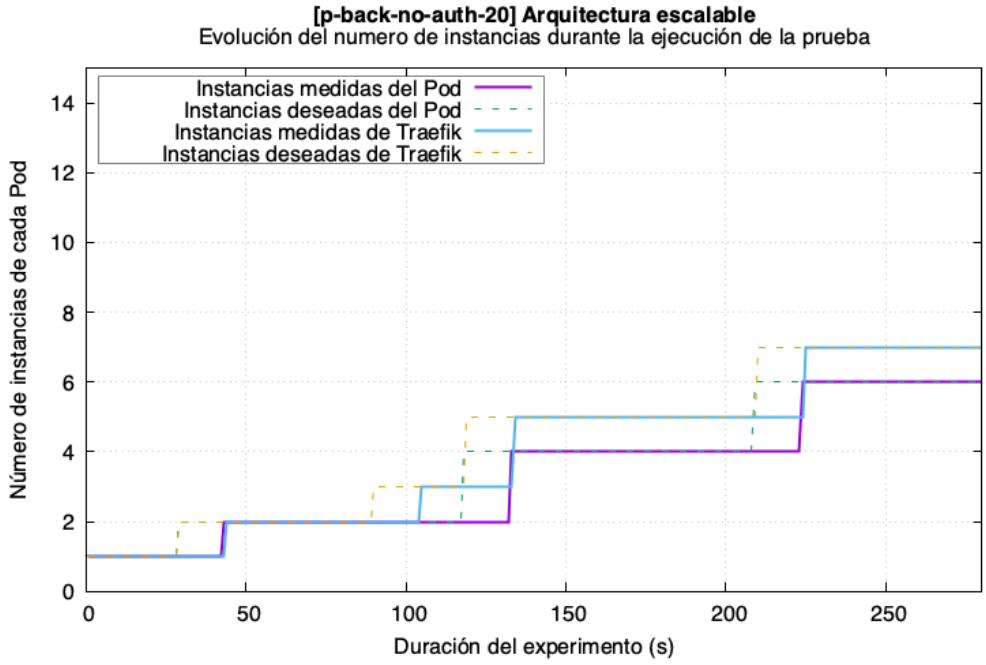

Figura 4-12: Evolución de instancias *P-BACK-NO-AUTH-20* (Arq. escalable).

En las dos figuras anteriores se puede apreciar el comportamiento de la arquitectura escalable, que durante el transcurso de la prueba va incrementando su tamaño. En esta figura también se han incluido el número de réplicas del *proxy reverso* (Traefik), que de manera similar al *Pod* sobre el que se realiza la prueba, experimenta una evolución a lo largo de la prueba.

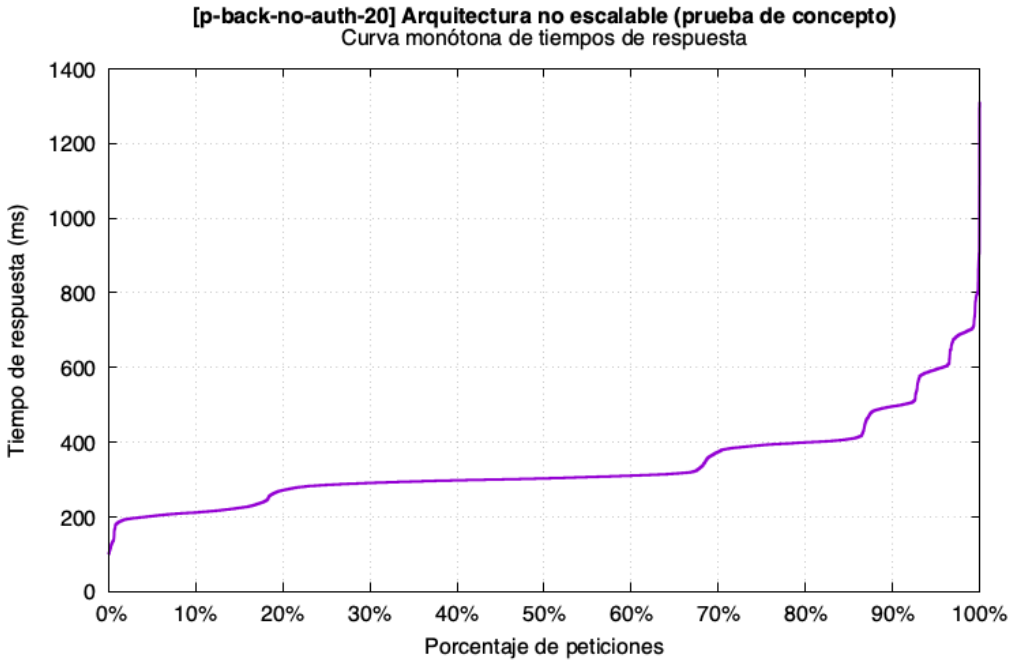

Figura 4-13: Curva monótona de tiempos de respuesta *P-BACK-NO-AUTH-20* (Arq. no escalable).

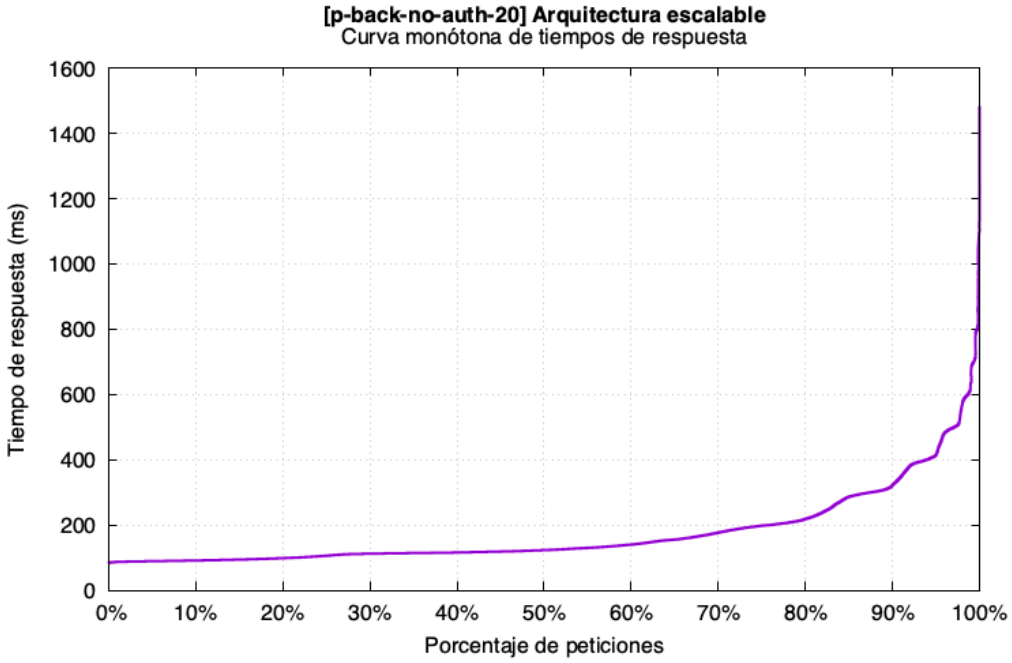

Figura 4-14: Curva monótona de tiempos de respuesta *P-BACK-NO-AUTH-20* (Arq. escalable).

En estos resultados se puede visualizar la diferencia en cuanto a tiempos de respuesta cuando entra en juego el lazo de control que va escalando al sistema de gestión.

A continuación, se muestran las parametrizaciones restantes sólo para la arquitectura escalable, con valores de concurrencia de 10 y 40 respectivamente.

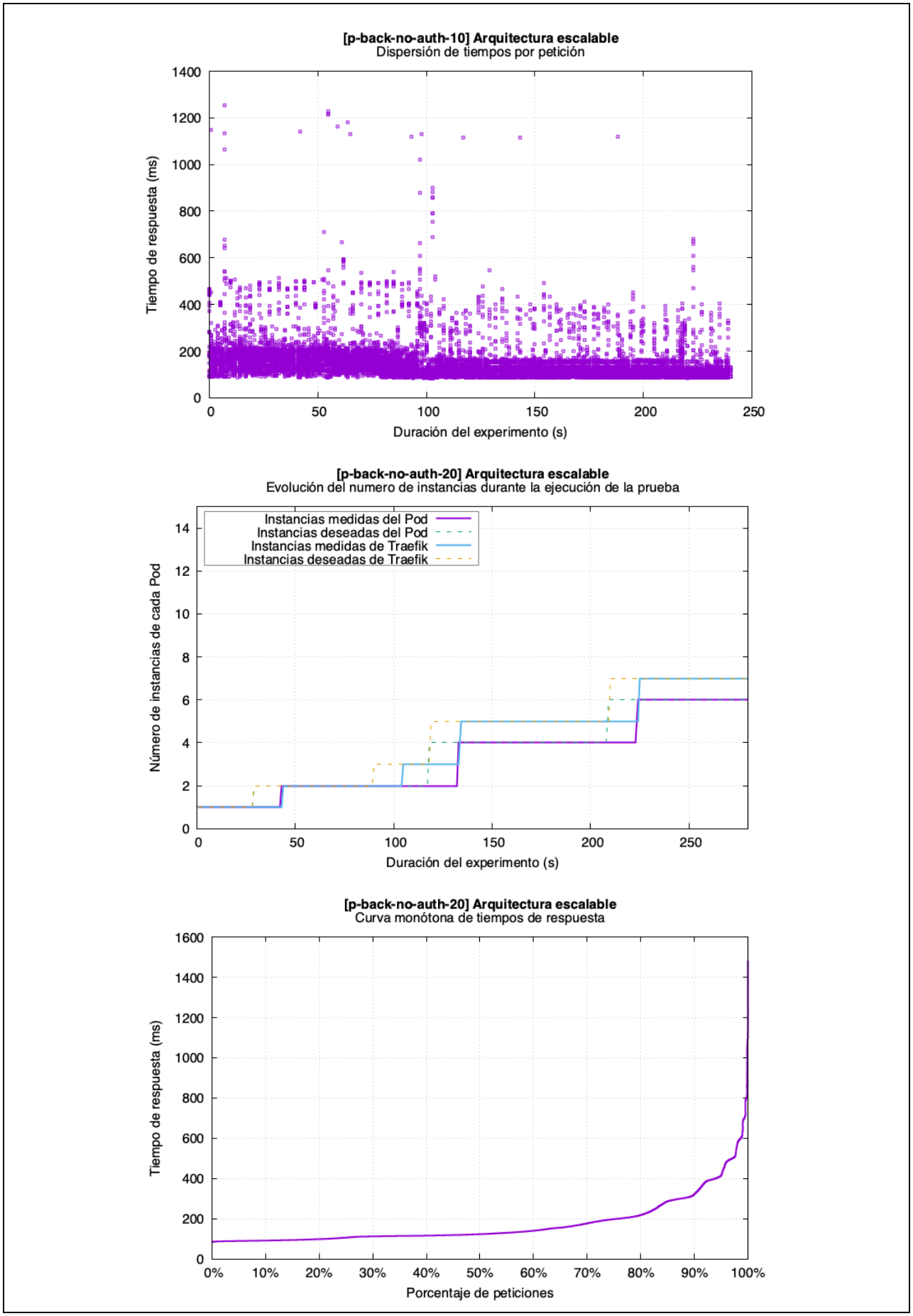

Figura 4-15: Resultados para la prueba *P-BACK-NO-AUTH-10* (Arq. escalable).

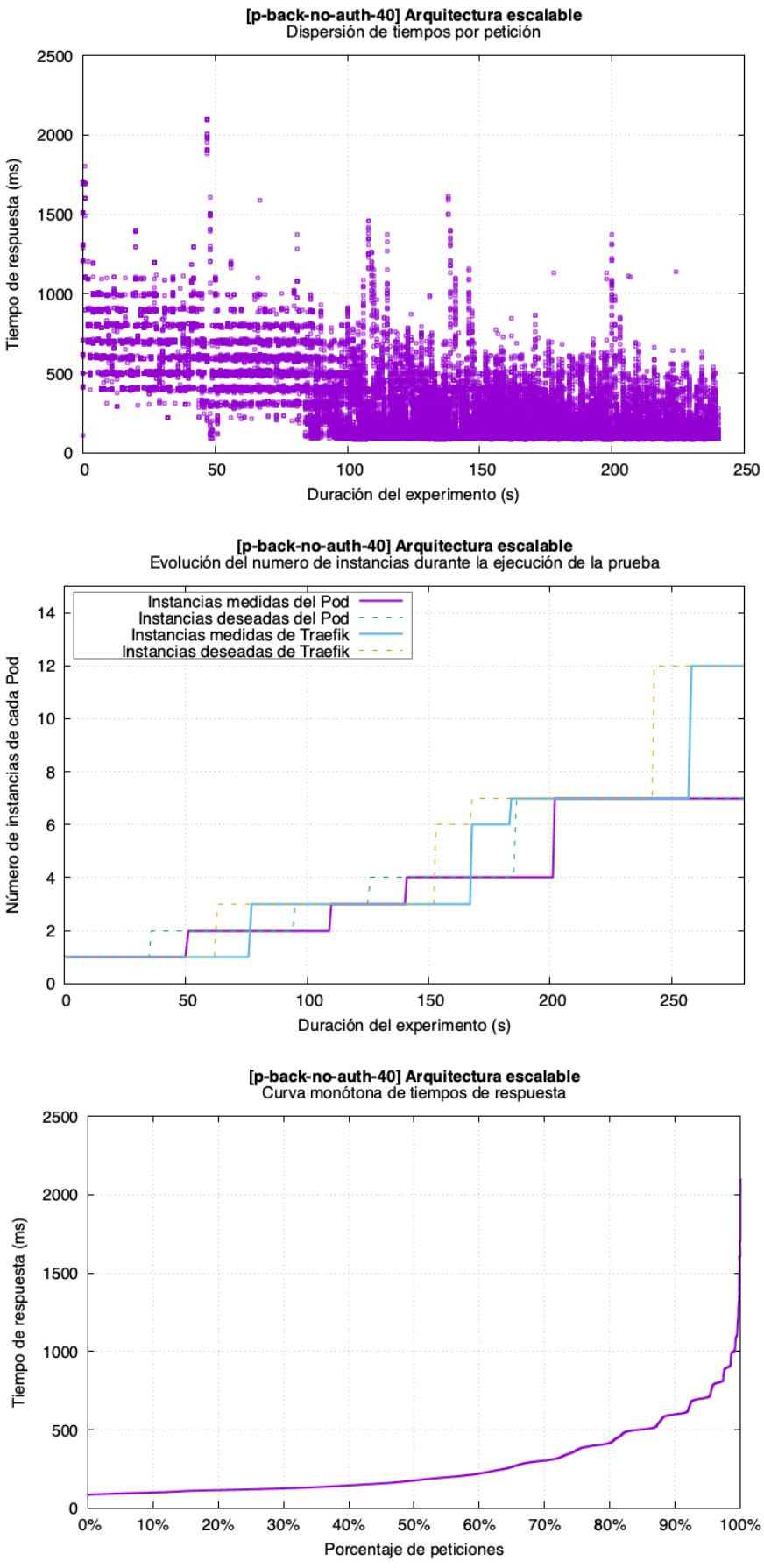

Figura 4-16: Resultados para la prueba *P-BACK-NO-AUTH-40* (Arq. escalable).

4.2.3. Resultados de las pruebas sobre el tercer escenario

En primer lugar, se comparan los resultados para el mismo nivel de concurrencia (20) y diferente entorno:

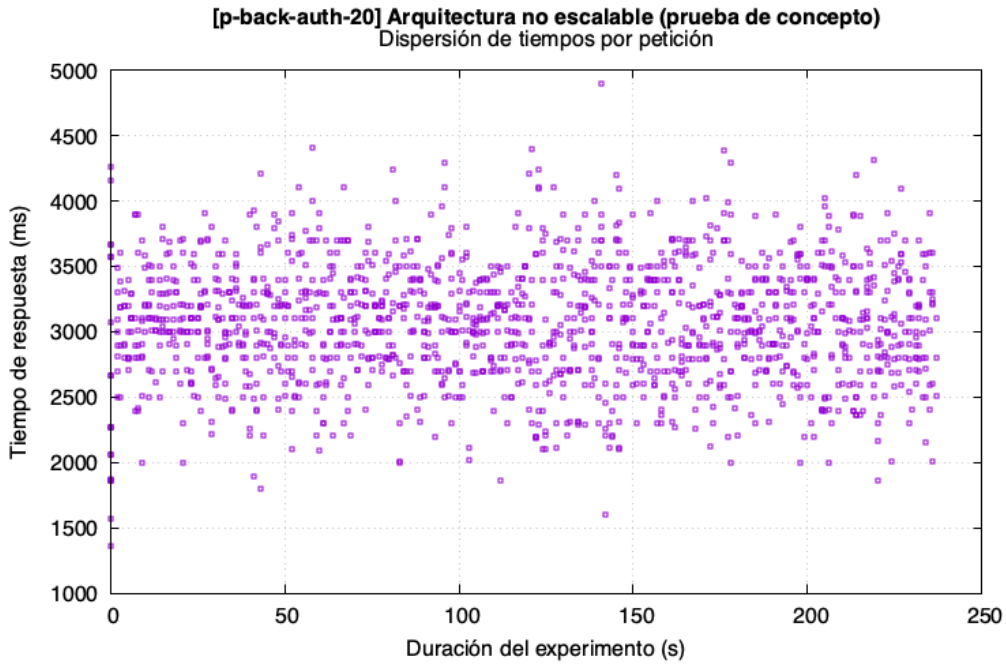

Figura 4-17: Dispersión de tiempos *P-BACK-AUTH-20)* (Arq. no escalable).

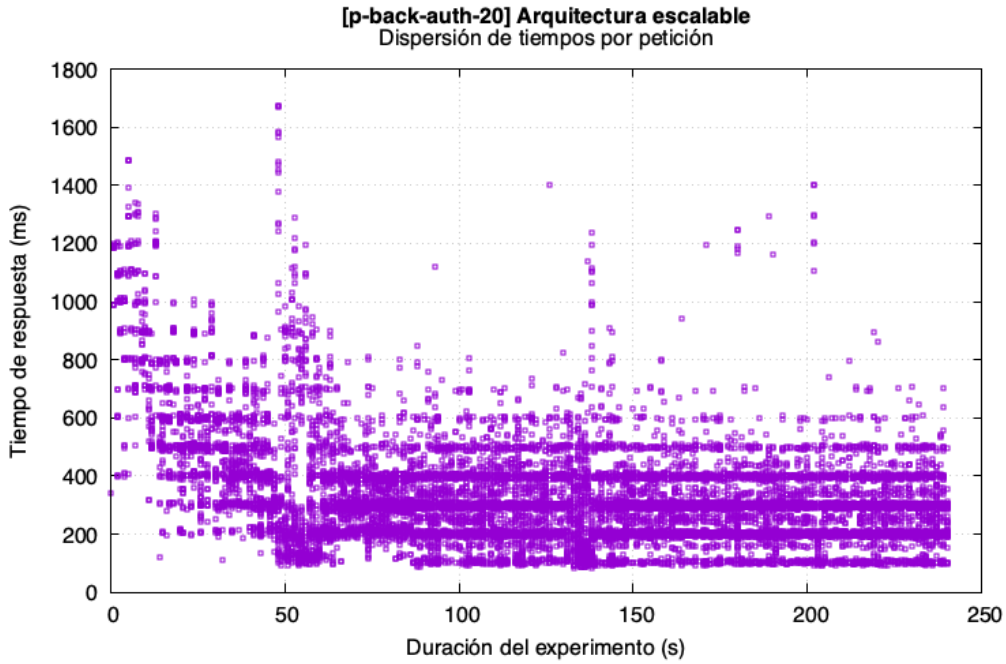

Figura 4-18: Dispersión de tiempos *P-BACK-AUTH-20)* (Arq. escalable).

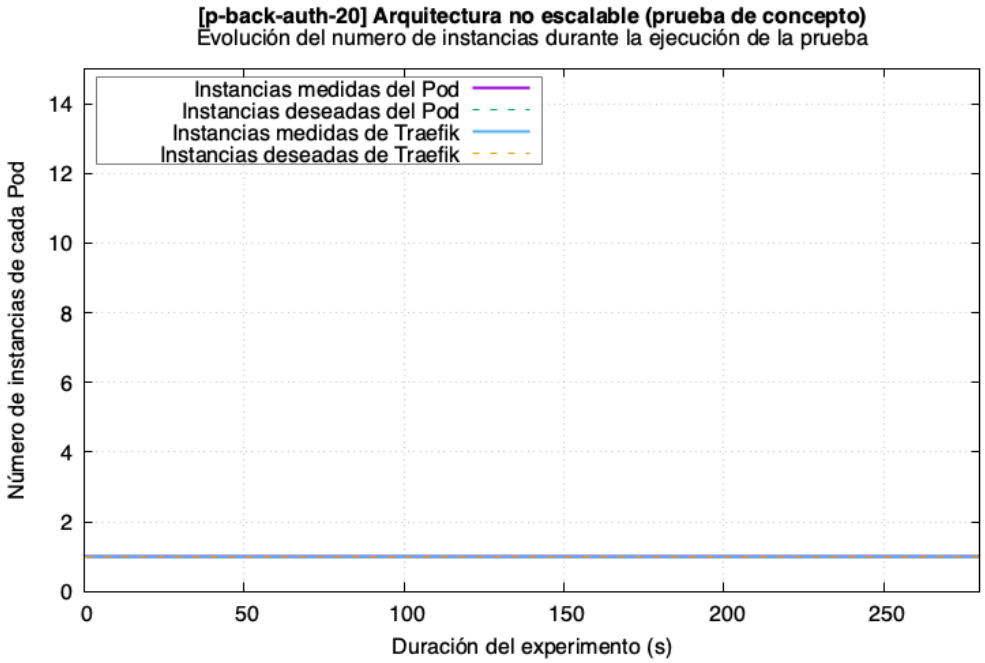

Figura 4-19: Evolución de instancias *P-BACK-AUTH-20* (Arq. no escalable).

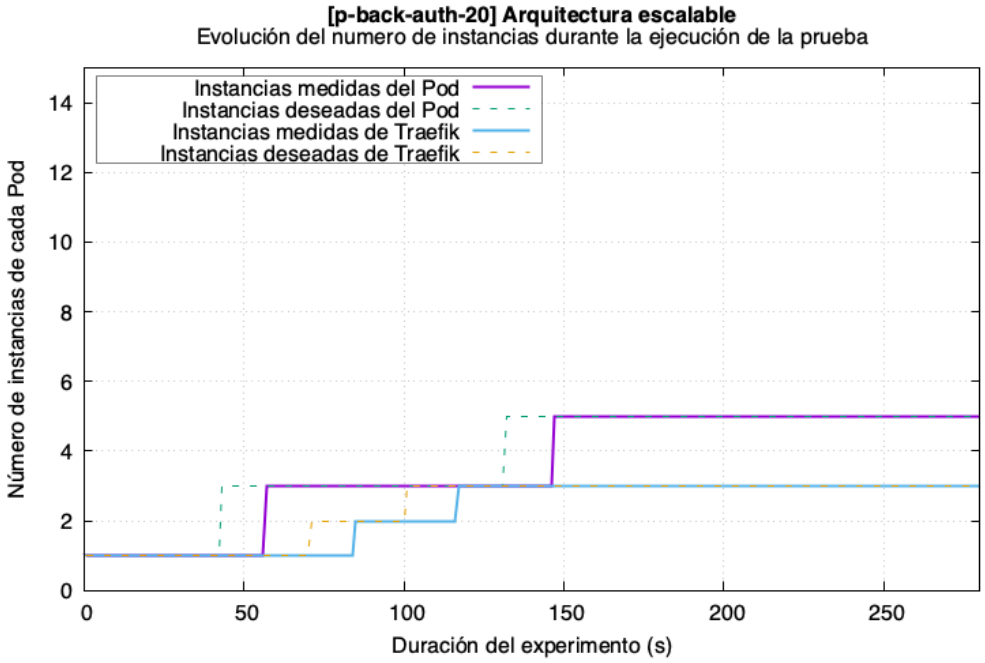

Figura 4-20: Evolución de instancias *P-BACK-AUTH-20* (Arq. escalable).

En las dos figuras anteriores se puede apreciar el comportamiento de la arquitectura escalable, que durante el transcurso de la prueba va incrementando su tamaño. En esta figura también se han incluido el número de réplicas del *proxy reverso* (Traefik), que de manera similar al *Pod* sobre el que se realiza la prueba, experimenta una evolución a lo largo de la prueba.

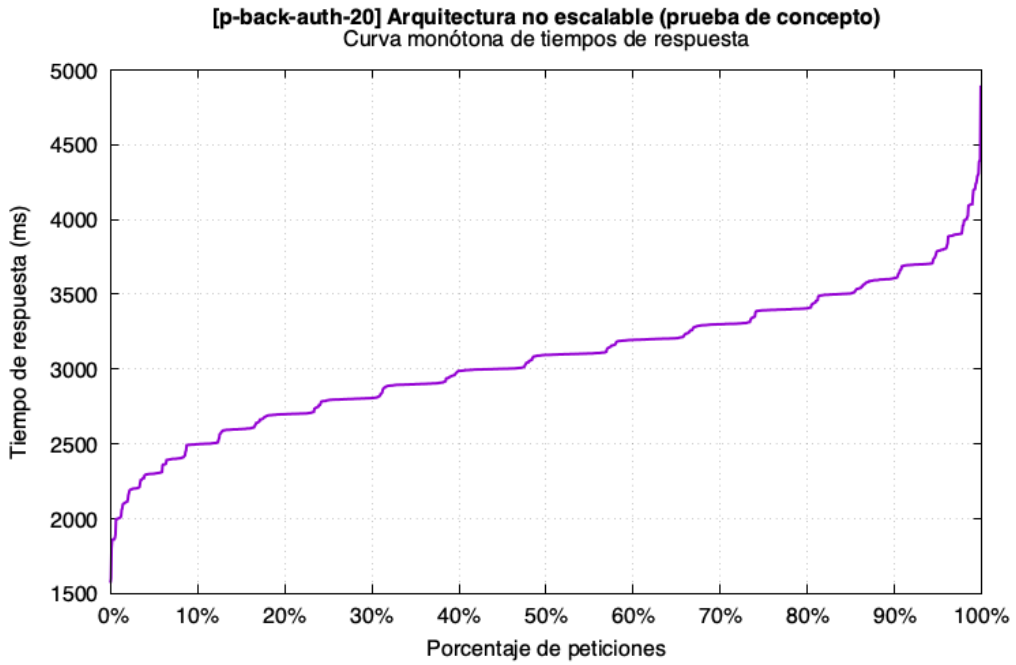

Figura 4-21: Curva monótona de tiempos de respuesta *P-BACK-AUTH-20* (Arq. no escalable).

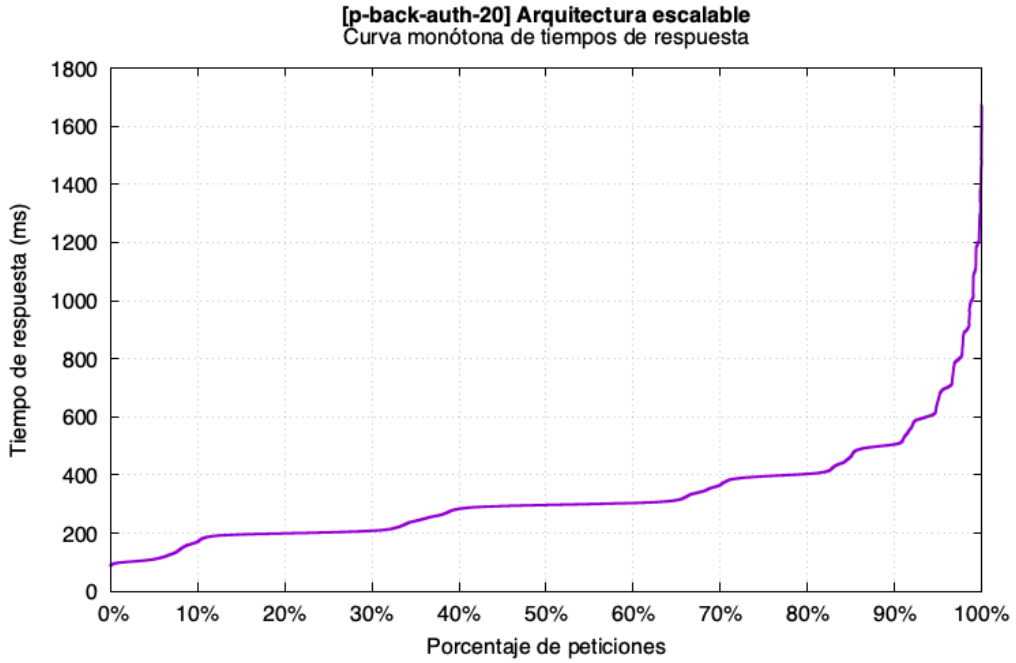

Figura 4-22: Curva monótona de tiempos de respuesta *P-BACK-AUTH-20* (Arq. escalable).

En estos resultados se puede visualizar la diferencia en cuanto a tiempos de respuesta cuando entra en juego el lazo de control que va escalando al sistema de gestión.

A continuación, se muestran las parametrizaciones restantes sólo para la arquitectura escalable, con valores de concurrencia de 10 y 40 respectivamente.

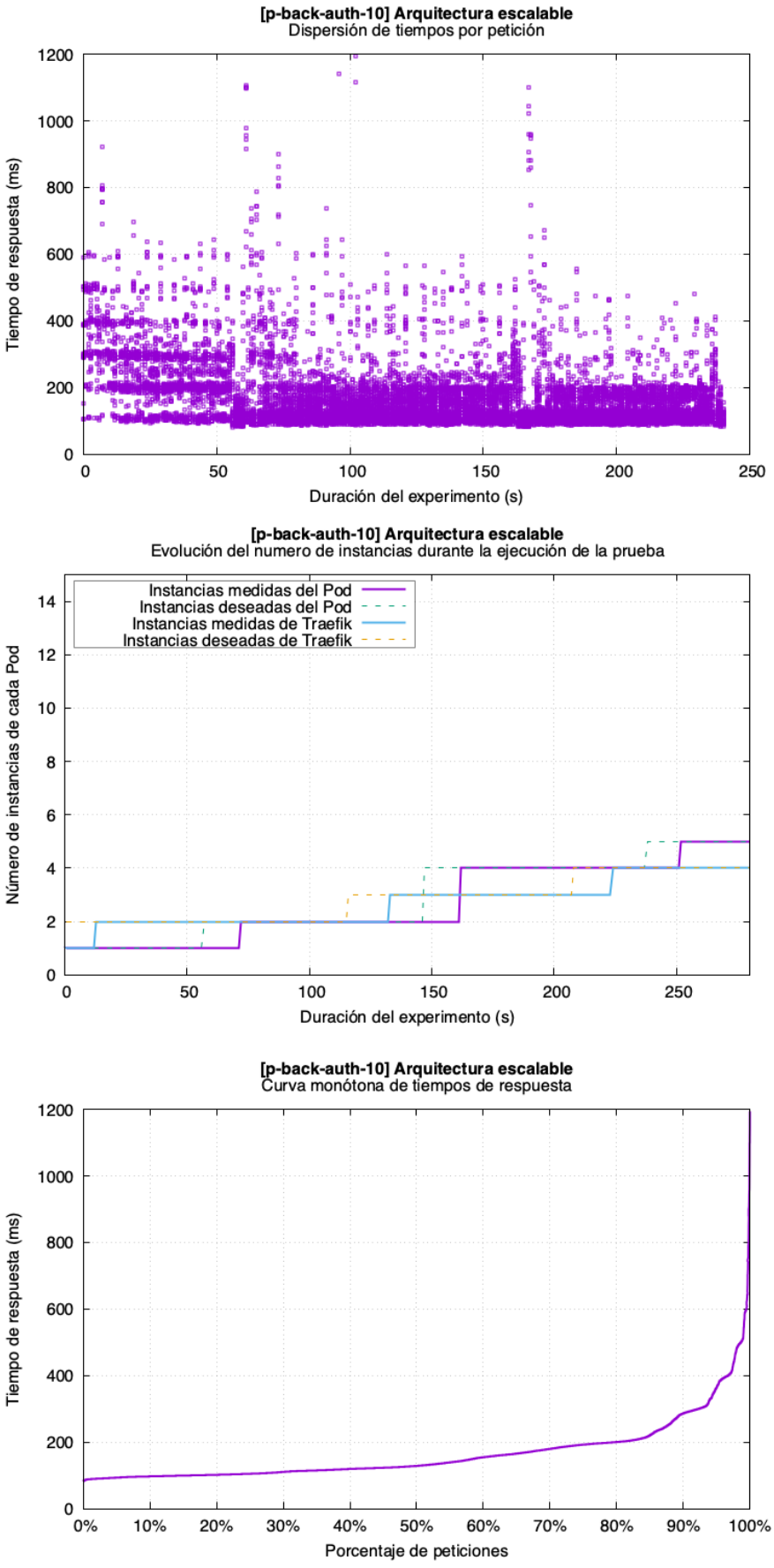

Figura 4-23: Resultados para la prueba *P-BACK-AUTH-10* (Arq. escalable).

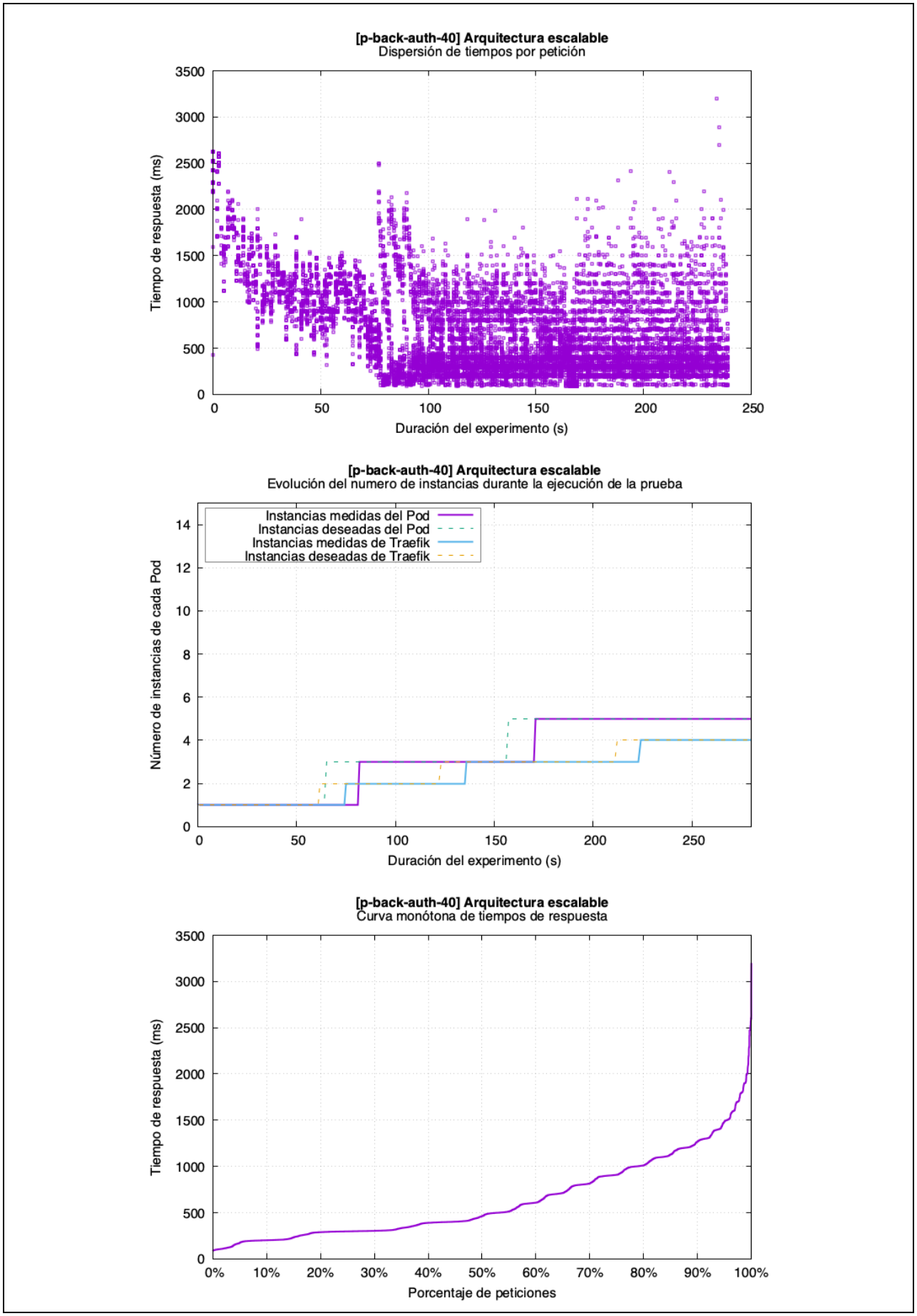

Figura 4-24: Resultados para la prueba *P-BACK-AUTH-40* (Arq. escalable).

4.3. Discusión

Tras la exposición de todas las gráficas, donde se han comparado diferentes entornos: escalable y no escalable, se puede concluir que la arquitectura es válida y se comporta de manera esperada. El tiempo de actuación no es inmediato, pero se encuentra dentro del margen previsto y supera claramente a la opción de despliegue para aplicaciones monolíticas visto en los primeros capítulos de este trabajo.

Es cierto que cuanto más se incremente el parámetro de concurrencia más incrementa el tiempo de respuesta, pero siempre de una manera controlada. Con la ayuda de las curvas monótonas también se comprueba que para todos los casos en los que entra en juego la arquitectura escalable el tiempo de respuesta del 90% de las peticiones no supera los 1,5 segundos.

En las comparativas que se realizan al comienzo de cada apartado, donde se comparan los dos diferentes entornos, se puede observar claramente como en cuanto el *clúster* comienza a crecer los tiempos de respuesta disminuyen y se concentran en torno a un valor medio, decrementando la varianza de todas las muestras de tiempo de respuesta.

Por lo tanto, se concluye que la transición que se ha realizado del sistema de gestión para desplegarlo en una tecnología como *Kubernetes* cumple con las expectativas iniciales que se habían planteado.

5 MEJORAS ADICIONALES A LA SOLUCIÓN PROPUESTA

En este capítulo se van a incluir ciertos servicios adicionales que no son obligatorios para el objetivo de este trabajo, pero si constituyen un requisito para el despliegue de un sistema real, que perdure en el tiempo. Dentro de este marco de mejoras existe un amplio abanico, sin embargo, en este capítulo sólo se van a ver tres opciones de las más conocidas.

5.1 Vigilancia del rendimiento del *clúster*

Pese a tener configurados los *Horizontal Pod Autoescaler* en cada uno de los *Deployments* críticos del sistema (los que se escalan de manera automática), resulta útil tener una herramienta de control en la que se pueda observar con detalle la evolución de todas las variables que se miden del *clúster*.

Dentro del repositorio de *Helm* existe un paquete llamado "*Kube Prometheus Stack*". Este paquete incluye un conjunto de herramientas formado por un recolector propio de métricas conocido como *Prometheus*, y una interfaz de visualización llamada *Grafana*. En esta interfaz se pueden construir gráficos y paneles de control con las diferentes variables de todo el clúster: estadísticas por *Nodo*, *Pod*, *Deployment*, etc.

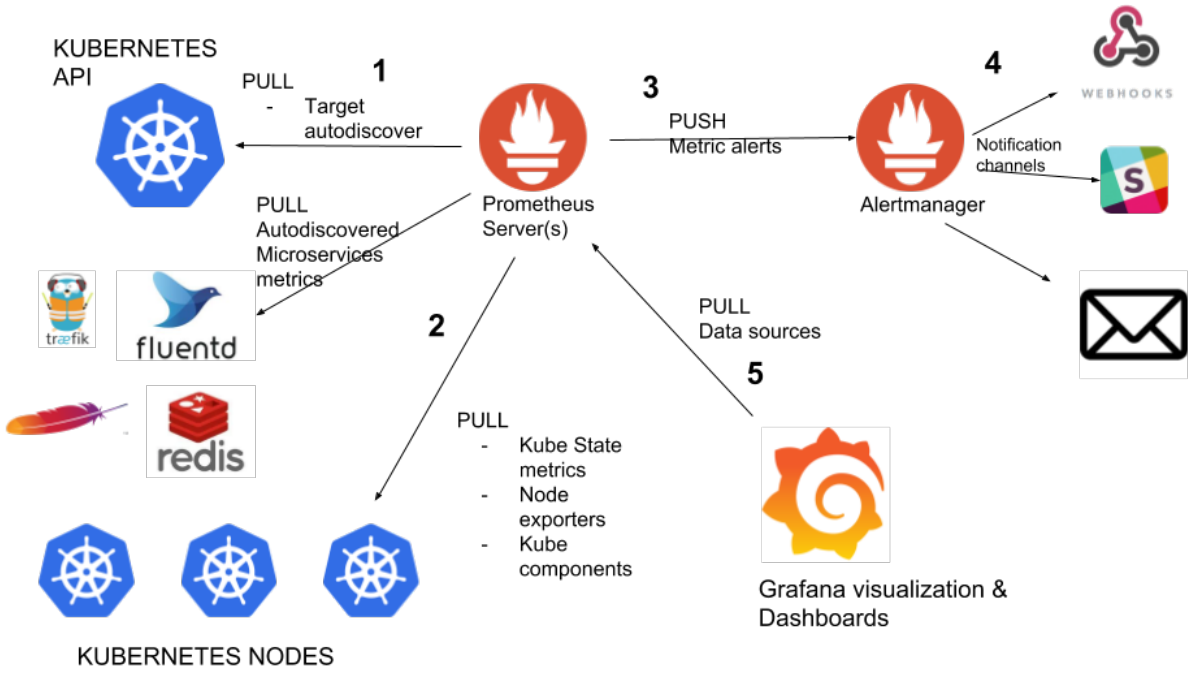

Figura 5-1: Esquema de la arquitectura para la vigilancia del *clúster*.

Adicionalmente también se puede configurar al *proxy reverso* (Traefik), el cual genera una gran cantidad de valores relacionados con el tiempo de respuesta, número de peticiones, que resulta útil y fácil de integrar con el paquete de herramientas mencionado anteriormente. Para poder completar las herramientas de vigilancia y que no solo contemplen aspectos entorno a los recursos hardware del *clúster*.

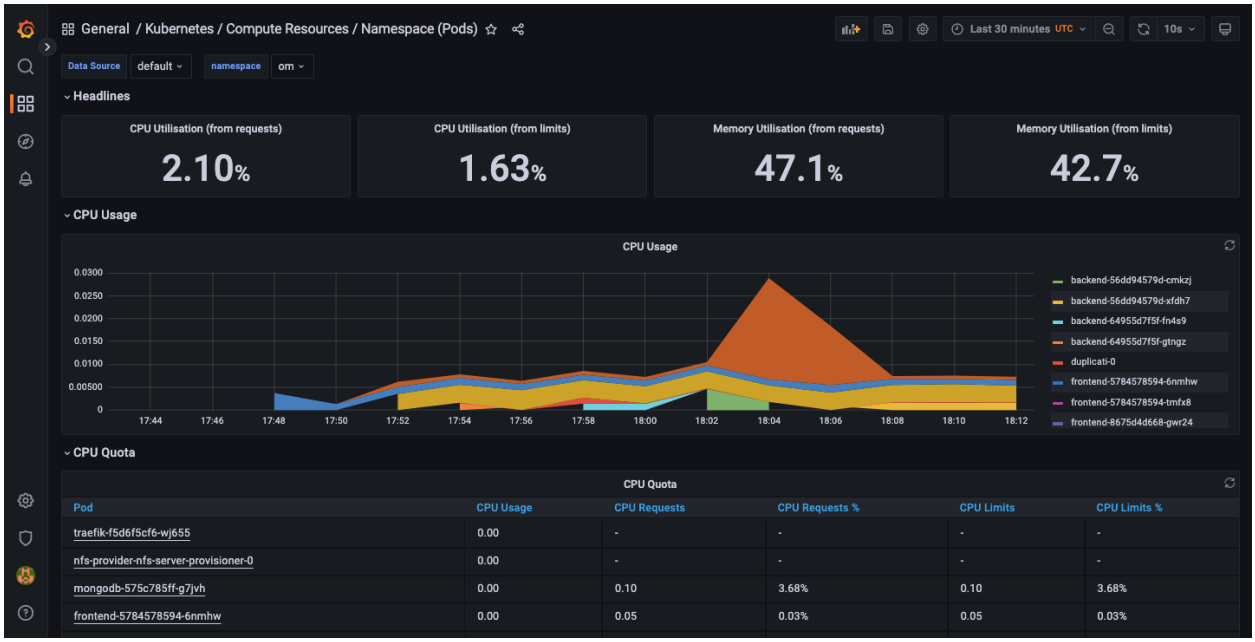

Figura 5-2: Captura de pantalla de la herramienta *Grafana* (*Pods* en un *Namespace*).

Figura 5-3: Captura de pantalla de la herramienta *Grafana* (*Recursos del clúster*).

5.2 Gestión de copias de seguridad de los archivos almacenados

Es cierto que dentro del entorno de *Kubernetes* existen soluciones de almacenamiento como los *Persistent Volumes*, que representan un sistema de archivos independiente que se puede montar en cualquiera de los *Pods* que se instancien. Estas soluciones permiten desvincular el ciclo de vida de los volúmenes del ciclo de vida de los *Pods*, que son elementos reemplazables ante fallos.

A pesar de desvincular ambos ciclos de vida, los datos siguen estando comprometidos dentro de la plataforma de *Kubernetes* o clúster que se haya elegido, por lo tanto, resulta imprescindible disponer de un sistema de gestión de copias de seguridad que almacene los datos en un lugar externo al clúster y replicado si es posible.

Para lograr ese objetivo introducimos un software de gestión de copias de seguridad conocido como *Duplicati*, que se encargará de realizar copias diarias de todos los volúmenes persistentes definidos en un proveedor externo y ofrecerá la posibilidad de restaurar un volumen persistente a partir de una copia de seguridad de un día en concreto. La adaptación del despliegue para *Kubernetes* sería la siguiente:

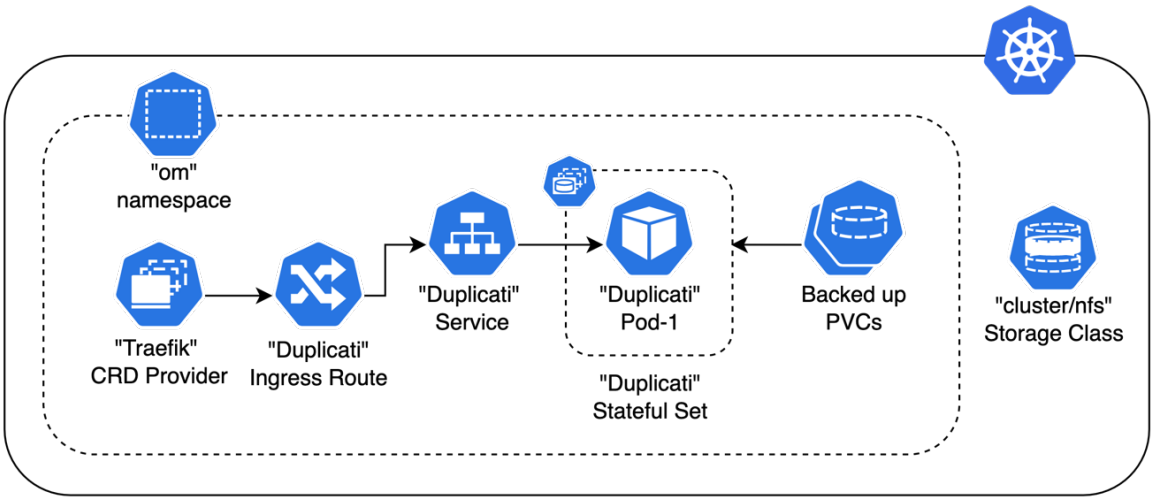

Figura 5-4: Diagrama de arquitectura para el software *Duplicati*.

Además, las copias de seguridad se almacenan encriptadas y comprimidas en un formato propio, optimizando el espacio y la integridad de estas. Para gestionar la configuración de "Duplicati" será necesario añadir un *Ingress Route* para que se realice la provisión de certificados y se pueda acceder mediante dominio utilizando HTTPS.

A continuación, se muestra una captura de la interfaz de esta herramienta. En este despliegue se han configurado la realización automática de tres copias de seguridad diarias, que se almacenan en el sistema externo y gratuito de *Google Drive*.

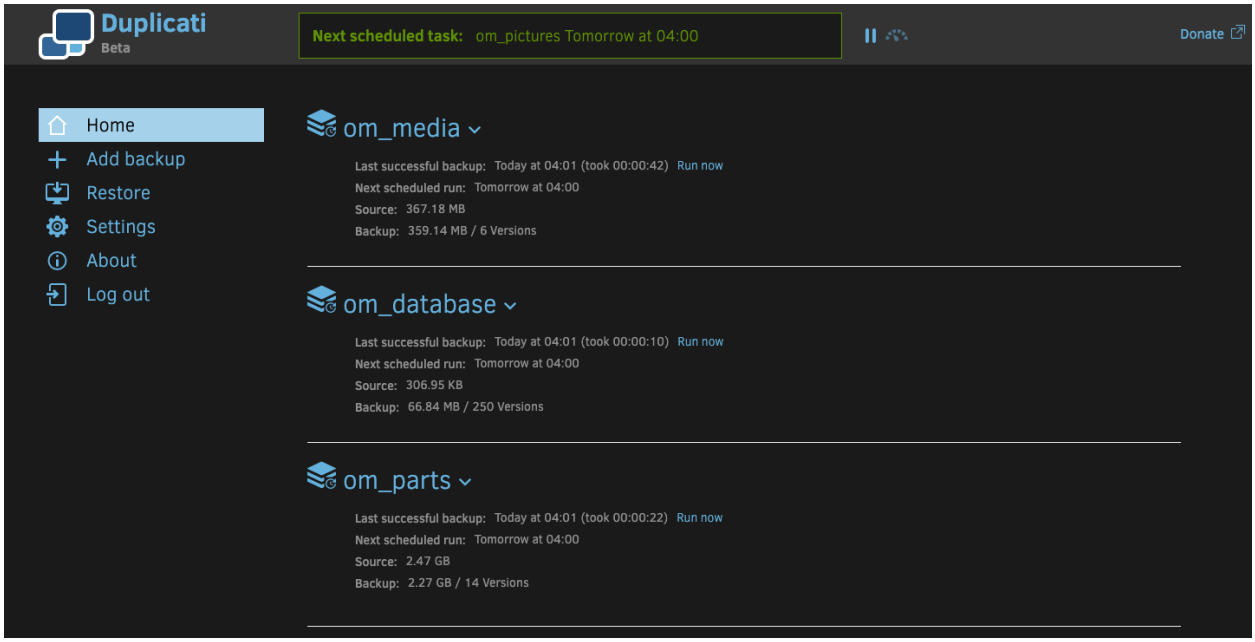

Figura 5-5: Captura de pantalla de la herramienta *Duplicati*.

5.3 Uso de metodologías para desarrollo continuo: GitOps

Como mejora adicional para el flujo del sistema de gestión resulta notable la introducción de técnicas como GitOps en el clúster de *Kubernetes*. En este caso la implementación que se ha elegido se conoce como *ArgoCD*, un software de código abierto que se implementa dentro de *Kubernetes* a través de un *CRD*.

Esta herramienta permite realizar agrupaciones de diferentes objetos de *Kubernetes* (*Deployments*, *Services*, *Persistent Volumes*, etc.) en otros objetos llamados *Applications*. Cada *Application* tiene definido una configuración diferente: repositorios de *Helm* con una configuración propia, archivos *Kustomize*, o simplemente manifiestos en *YAML*.

Los archivos de *Kustomize* y *YAML* se incluyen vinculando la carpeta contenedora de estos a través de un repositorio de Git, permitiendo generar eventos de actualización cuando se produzca algún cambio en los archivos del repositorio.

Al poder declarar estas configuraciones como un objeto *Applications* resulta bastante sencillo hacer la puesta en marcha de la herramienta al completo, ya que estos objetos pueden incluirse en un fichero de manifiesto (*YAML*).

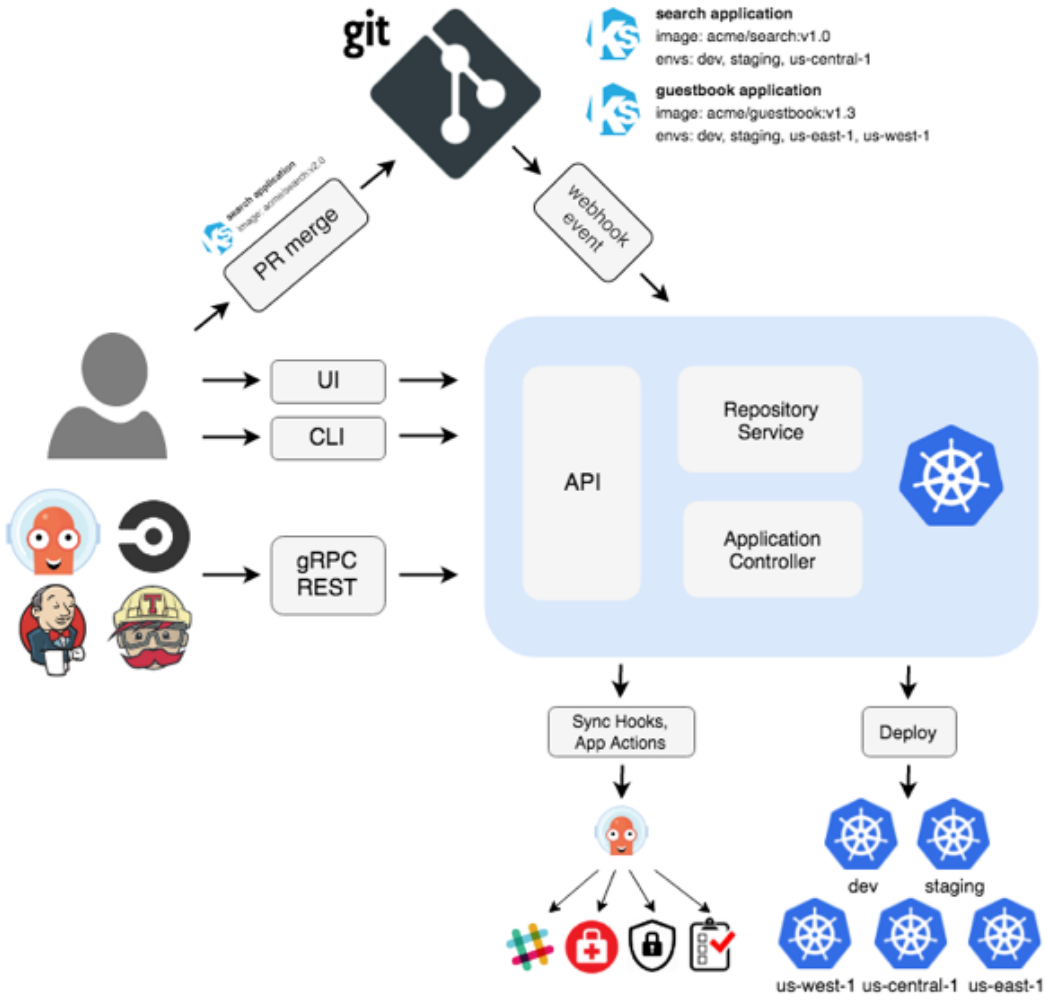

Figura 5-6: Arquitectura de la herramienta *ArgoCD*.

Para operar *ArgoCD* existen varias alternativas, como la interfaz web o el cliente que se instala en la consola de comandos. Por simplicidad, las labores de configuración se van a realizar utilizando el cliente en la consola de comandos, aunque las imágenes que se muestren a continuación sean de la interfaz web que ofrece.

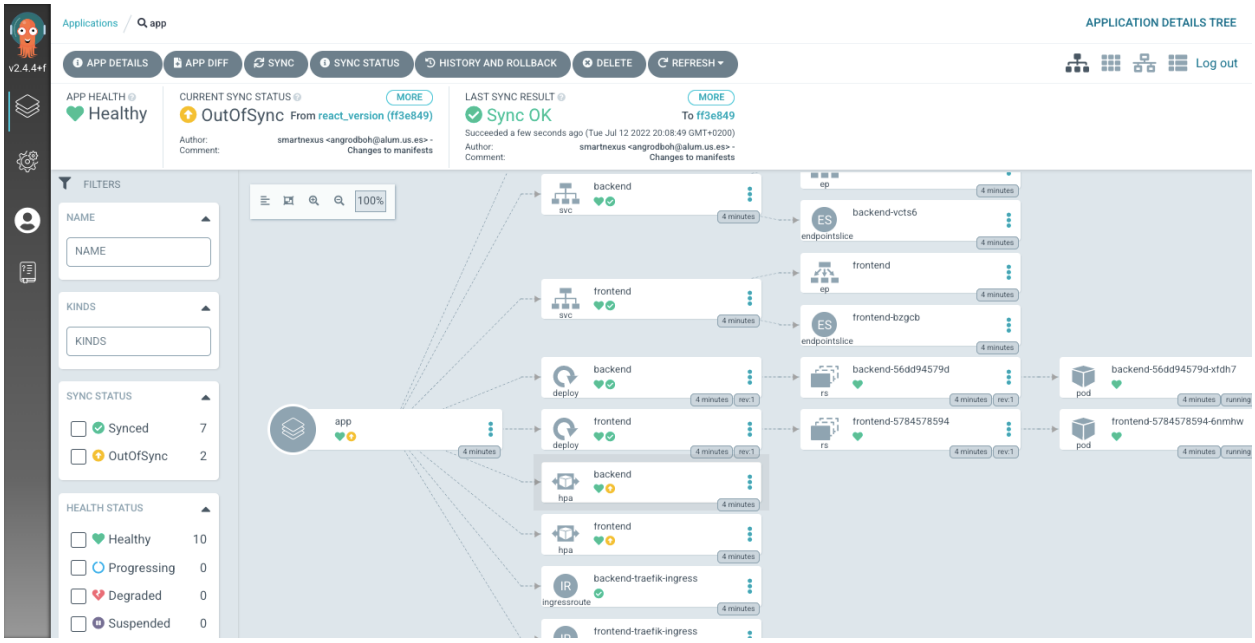

Figura 5-7: Captura de pantalla de la herramienta *ArgoCD* (despliegue sistema de gestión).

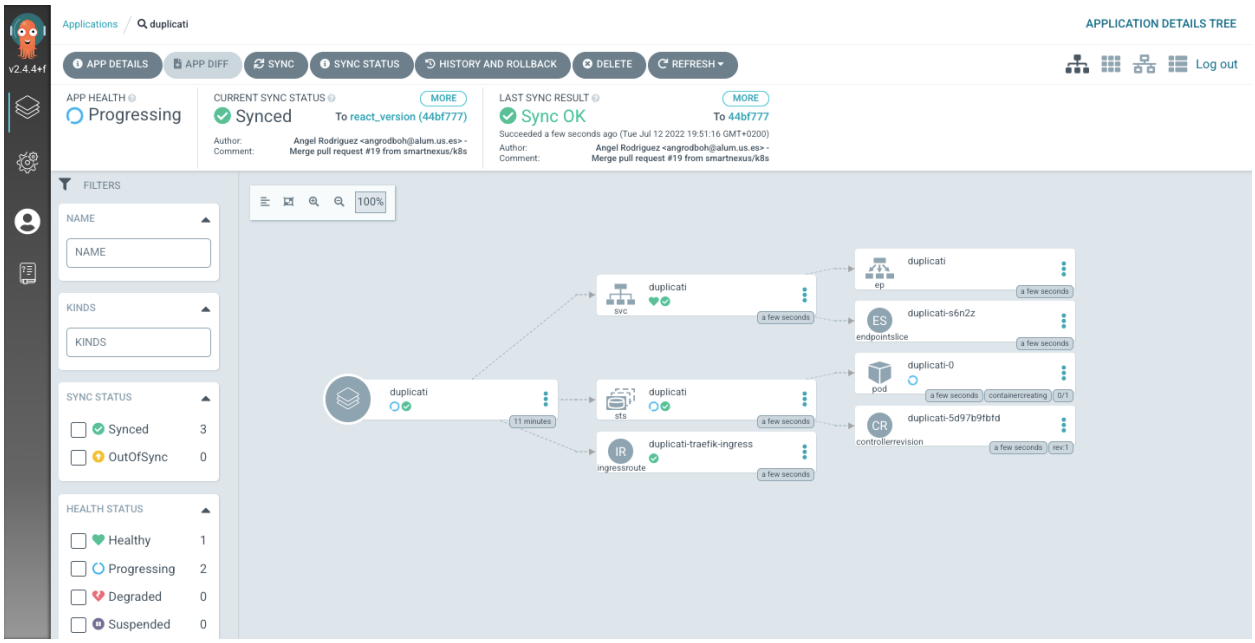

Figura 5-8: Captura de pantalla de la herramienta *ArgoCD* (despliegue Duplicati).

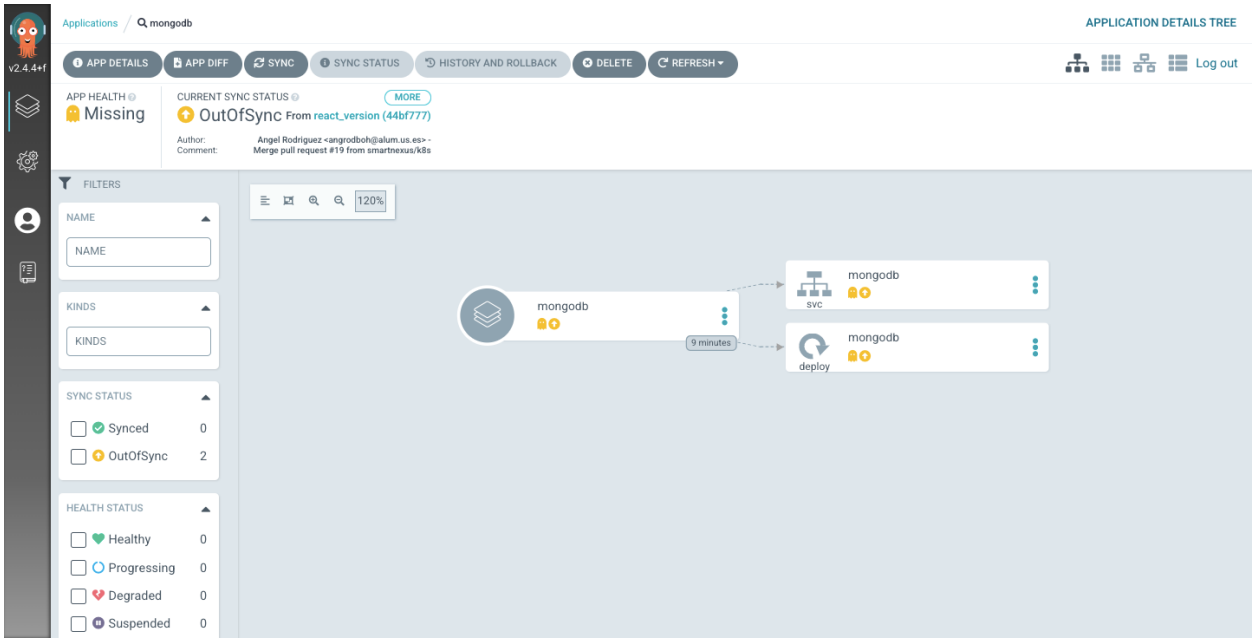

Figura 5-9: Captura de pantalla de la herramienta *ArgoCD* (despliegue MongoDB).

6 CONCLUSIONES Y LÍNEAS DE AVANCE

6.1 Conclusiones

En este trabajo se ha observado el impacto que tiene la solución de una arquitectura escalable ante un problema causado por eventos de saturación. Es verdad que la solución obtenida no es fácil, ni tampoco definitiva, será necesario adaptar la arquitectura en función de la evolución que sufra el sistema de gestión.

A pesar de ello, las modificaciones que sean necesarias realizar serán sencillas en comparación con todo el proceso que se ha seguido en el trabajo. Una vez el sistema se ha encapsulado y se ha elaborado la arquitectura escalable con todos los microservicios sólo serán necesarias modificaciones como incluir un nuevo microservicio o actualizar el comportamiento de cualquiera del resto.

Además de conseguir beneficios en la operación del sistema, también se ha reducido bastante el coste de todo el sistema se está abonando el precio de cada uno de los recursos que se están utilizando. Con esto no conseguimos únicamente poder destinar este nuevo beneficio a otros objetivos, sino que reducimos el consumo eléctrico y la huella de carbono en el planeta.

Anteriormente, cuando todos los despliegues se realizaban con máquinas muy potentes, las cuales pasaban más del 70% de su tiempo de operación con menos de la mitad de los recursos en uso, se estaba demandando unos recursos eléctricos mayores que si disponemos de un número variable de máquinas a utilizar y se reparten equitativamente entre todas las entidades o personas que necesiten utilizarlas.

Una vez conocida la solución a este problema, la clave no está en cambiar instantáneamente todas las aplicaciones existentes en el mundo que no sigan este patrón, eso también sería un error. El proceso debe ser algo gradual, donde gracias a la divulgación y existencia de estas tecnologías como *Kubernetes* cada entidad o persona pueda dar comienzo a su transición interna.

6.2 Líneas de avance

A pesar de que la solución obtenida a través de la arquitectura escalable resulta ser válida para el problema inicial que se planteó, existen ciertas partes de la arquitectura que no se encuentran preparadas para sobrevivir a un crecimiento de las funcionalidades del sistema de gestión.

En primer lugar, la solución de almacenamiento que se ha conseguido no es del todo escalable, ya que por limitaciones de *Kubernetes* es necesario especificar el tamaño de los *PVC* en el momento de su creación. De esta manera, si en algún momento se llegase al máximo establecido en la creación del sistema, sería necesario hacer una migración eliminando y volviendo a crear los nuevos *PVC* asignándoles esta vez un tamaño mayor.

Esta limitación se encuentra en todos los *PVCs* que se creen en *Kubernetes*, aunque existen ciertos avances en los que se asegura que en futuras versiones se permitirá la creación de *PVCs* con tamaño ampliable. Al no disponer con seguridad de esta funcionalidad futura, será necesario buscar una alternativa.

La principal alternativa es realizar una modificación de uno de los procesos del sistema de gestión, con el objetivo de utilizar un servicio de almacenamiento distribuido que implemente S3 [14]. Esta tecnología no se considera estándar, ya que pertenece a *Amazon Web Services*, pero se encuentra bastante extendida por todo el mercado de los proveedores de almacenamiento distribuido en *cloud*, lo que la convierte en una solución óptima.

Otro de los avances más prioritarios es realizar una división más minuciosa de los microservicios que componen el sistema de gestión. En la arquitectura que se ha diseñado existen microservicios que realizan ciertas tareas que se podrían encapsular en otros microservicios, como es por ejemplo el algoritmo de compresión de imágenes, que se ejecuta dentro del proceso API Rest.

REFERENCIAS

[1] Facebook, «Introducción a las interfaces de usuario con React,» [En línea]. Available: https://reactjs.org/.

[2] OpenJS, «Sobre NodeJS,» [En línea]. Available: https://nodejs.org/en/about/.

[3] MongoDB, «What is MongoDB?,» [En línea]. Available: https://www.mongodb.com/docs/manual/.

[4] Redis, «Redis (almacenamiento de sesiones),» [En línea]. Available: https://redis.io/docs/about/.

[5] IETF, «NFS Version 3 Protocol Specification,» [En línea]. Available: https://datatracker.ietf.org/doc/html/rfc1813.

[6] Mathworks, «Structural similarity (SSIM) index for measuring image quality,» [En línea]. Available: https://www.mathworks.com/help/images/ref/ssim.html.

[7] Oracle, «Longest Common Substring,» [En línea]. Available: https://www.oracle.com/webfolder/technetwork/data-quality/edqhelp/Content/processor_library/matching/comparisons/longest_common_substring_percentage.htm.

[8] Kubernetes, «Introducción a Kubernetes y sus casos de uso,» [En línea]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[9] Docker, «Use containers to Build, Share and Run your applications,» [En línea]. Available: https://www.docker.com/resources/what-container/.

[10] Traefik-Labs, «Traefik & Kubernetes,» [En línea]. Available: https://doc.traefik.io/traefik/routing/providers/kubernetes-crd/.

[11] Helm, «What is Helm?,» [En línea]. Available: https://helm.sh/.

[12] Á. Rodríguez, «Repositorio con manifiestos y configuración de la arquitectura escalable,» [En línea]. Available: https://github.com/smartnexus/tfg.

[13] Apache, «ab - Apache HTTP server benchmarking tool,» [En línea]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html.

[14] AWS, «Amazon S3 REST API Introduction,» [En línea]. Available: https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html.

ANEXO A: LENGUAJE DE KUBERNETES

En este anexo se pretende explicar los diferentes objetos que existen en *Kubernetes* con la idea de la comprensión de los diferentes términos utilizados a lo largo de este documento sea más sencilla.

Dentro de *Kubernetes* existen múltiples tipos de objetos, de los cuales algunos forman parte de la arquitectura y otros permiten el funcionamiento interno del clúster o definen parámetros de configuración. Los objetos utilizados para elaborar la arquitectura son los siguientes:

- *Pod:* es un objeto que instancia uno o más contenedores dentro de él. Estos contenedores se encuentran alcanzables a través de una red aislada e interna al *Pod* y se configuran independientemente a través de una plantilla.
- *Replica Set:* representa a un objeto que instancia un *Pod* replicado un número determinado de veces, definido a través de un parámetro. Se suele emplear cuando se desea desplegar un software que va a instanciarse múltiples veces.
- *Stateful Set:* es un objeto que instancia un único *Pod*, el cual tiene un identificador numérico, a diferencia de los otros *Pods*. Se suele utilizar para desplegar cualquier software no replicable, es decir que actúe como un único proceso dentro del clúster.
- *Deployment:* es un objeto que instancia un *Replica Set*. La configuración disponible es mínima, se especifican los contenedores que formarán parte del *Pod* que genere el *Replica Set* y el número de instancias.
- *Storage Class:* es un objeto de define el aprovisionamiento de volúmenes. Cada uno de estos lleva configurado toda la lógica de almacenamiento de la información que se persiste. Estos objetos permiten abstraer la implementación de cómo se almacenan los archivos, permitiendo así al operador elegir entre las diferentes soluciones disponibles.
- *Persistent Volume Claim (PVC):* representa una solicitud de almacenamiento por parte de un operador del *clúster*. En cada uno de los objetos se especifica la clase elegida (implementación) y el tamaño que se desea que tenga el sistema de archivos. Una vez exista este *PVC*, cualquiera de los *Pods* podrá montarlo siguiendo los permisos de lectura y escritura configuradas en el volumen.
- *Namespace:* representa un ámbito aislado dentro del *clúster*. Todos los objetos que se definen pertenecen a uno diferente, y siguiendo con los principios de encapsulación, un *Service* que se encuentre en el *Namespace A* no puede alcanzar a otro que se encuentre en un *Namespace B*.
- *Service:* representa un punto de acceso, denominado por dirección IPv4 y un puerto que apunta a un conjunto de *Pods* utilizando una etiqueta. El manifiesto de cada *Pod* se configura para que incluya esa etiqueta y al *Service* se le asigna un selector para que rediriga el tráfico entrante de manera distribuida a cada *Pod*.
- *Custom Resource Definition (CRD):* representa un objeto personalizado de *Kubernetes*. Se define como un manifiesto y permite incorporar una lógica de actuación o la creación de otros objetos de *Kubernetes*.
- *Secret:* es un objeto que almacena texto plano dentro del clúster. Normalmente suele almacenar contenido encriptado y puede ser configuración de algún *Pod*, claves de acceso a un servicio externo, etc…
- *Ingress Route:* es un objeto definido por *Traefik* que permite configurar el *proxy reverso* para que funcione dentro de *Kubernetes*. Mediante este objeto se evita tener que almacenar la configuración de *Traefik* en un archivo externo y se almacena dentro del conjunto de objetos del clúster.