UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Grid y Computación de Altas Prestaciones GRyCAP

etsinf Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Plataforma Serverless Híbrida de Procesado de Datos

Trabajo Fin de Máster

Máster Universitario en Gestión de la Información

Autor: Sebastián Risco Gallardo

Tutor: Germán Moltó Martínez

Curso 2018-2019

Agradecimientos

Me gustaría mostrar mi más sincero agradecimiento a Germán Moltó por confiar en mí desde el principio y por toda su ayuda durante la realización de este trabajo. Gracias por darme la gran oportunidad de trabajar haciendo lo que me gusta y permitirme continuar los estudios en este campo que tanto me apasiona.

Gracias tambien a mis compañeros Alfonso Pérez y Miguel Caballer por todo el tiempo que han dedicado a resolver mis dudas. Y gracias al resto de compañeros del grupo de Grid y Computación de Altas Prestaciones por acogerme como uno más y ayudarme siempre que lo he necesitado.

Y como no podía ser de otra forma, gracias a mis familiares y amigos por apoyarme siempre, y en especial a mi padre, porque sin él nada de esto habría sido posible.

Resumen

El auge de la virtualización en los últimos años ha permitido que diversas compañías ofrezcan sus recursos informáticos a través de Internet mediante un modelo de pago por uso, conocido como computación en la nube. La gran aceptación de este modelo, unida a los recientes avances en las tecnologías de contenedores de *software*, han propiciado la aparición del paradigma Serverless. Este paradigma permite ejecutar funciones sin que los usuarios tengan que preocuparse por la gestión y escalado de la infraestructura subyacente. Sin embargo, las plataformas Serverless de los principales proveedores de computación en la nube tienen serias limitaciones que impiden su uso para procesar aplicaciones generales.

Con el fin de evitar esas limitaciones, en este trabajo se ha colaborado con el grupo de Grid y Computación de Altas Prestaciones de la Universitat Politècnica de València en el desarrollo de una plataforma Serverless. Esta plataforma puede ser desplegada sobre cualquier infraestructura de cómputo en la nube, ya sea pública o privada. Para ello se han integrado diversas herramientas de código abierto junto a nuevos componentes desarrollados por el grupo. Complementada con un modelo de programación que permite invocar funciones ante la subida de ficheros a un sistema de almacenamiento externo, la plataforma desarrollada es capaz de procesar datos en entornos híbridos, es decir, donde el cómputo se realiza en una infraestructura y la persistencia de datos en otra. Finalmente, se han implementado dos casos de uso basados en código y datos abiertos para demostrar el funcionamiento y la potencia de la plataforma.

Palabras clave: *Serverless*, Funciones como servicio, procesamiento de ficheros, computación en la nube

Abstract

The rise of virtualization in recent years has allowed companies to offer their computing resources over the Internet through a pay-per-use model known as cloud computing. The great acceptance of this model, together with recent advances in software container technologies, have led to the emergence of the Serverless paradigm. This paradigm allows functions to be executed without users having to worry about the management and scaling of the underlying infrastructure. However, the Serverless platforms of the main cloud computing providers have serious limitations that restrict their use to process general applications.

In order to overcome these limitations, in this work there has been collaboration with the Grid and High Performance Computing group of the Universitat Politècnica de València in the development of a Serverless platform. This platform can be deployed on any computing infrastructure in the cloud, either public or private. For this purpose, several open source tools have been integrated together with new components developed by the group. Complemented with a programming model that allows the invocation of functions when files are uploaded to an external storage system, the developed platform allows the processing of data in hybrid environments, that is, where computation is carried out in one infrastructure and data persistence in another. Finally, two use cases based on open source and open data have been implemented in order to demonstrate the operation and performance of the platform.

Key words: Serverless, Functions as a Service, file processing, cloud computing

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

En este capítulo inicial de la memoria se pretende introducir el Trabajo de Fin de Máster en Gestión de la Información realizado. Para ello se dedica la primera sección a explicar el contexto. A continuación, se exponen los principales motivos para la realización del proyecto, tratando de justificar la realización del mismo. Posteriormente, se definen los objetivos a conseguir y, para terminar, se detalla la estructura seguida en el resto del documento.

1.1 Contexto

Gracias a los avances y la popularización de la virtualización[1] en los últimos años, múltiples compañías comenzaron a ofrecer diferentes recursos informáticos a través de Internet mediante un modelo de pago por uso. Así surgió lo que se conoce como Cloud Computing [1], o computación en la nube en castellano.

Inicialmente, estas compañías, nombradas de ahora en adelante como proveedores de Cloud público, se centraron en proporcionar su infraestructura como servicio (IaaS por sus siglas en inglés). El modelo IaaS permite que cualquier usuario pueda tener acceso a máquinas virtuales, almacenamiento y redes a través de Internet de forma instantánea. Así se puede evitar la inversión en la compra y mantenimiento de equipos informáticos cuando no se precisa o no se considera oportuno. Además, no es necesario establecer un contrato fijo sobre los recursos que se desean alquilar, al contrario de lo que ocurre con los clásicos proveedores de *hosting*.

Conforme fueron consolidándose, los proveedores de Cloud público fueron añadiendo servicios a su catálogo, especialmente enfocados a desarrolladores, bajo el modelo PaaS (plataforma como servicio). Este modelo añade un nivel de abstracción sobre IaaS, siendo los proveedores los encargados de gestionar no solo los equipos informáticos y las máquinas virtuales, sino también los sistemas operativos y los entornos que se ejecutan sobre las mismas. Gracias a esto los desarrolladores pueden crear aplicaciones sobre la plataforma utilizando las herramientas *software* suministradas por el proveedor sin tener que preocuparse por los sistemas subyacentes.

Por otra parte, enfocándose principalmente a usuarios finales surge el modelo SaaS (*software* como servicio), que consiste en ofrecer aplicaciones a través de Internet. Algunos ejemplos de aplicaciones bajo este modelo son: Google Docs[2], para la creación y

[1] https://www.redhat.com/es/topics/virtualization/what-is-virtualization

distribución de documentos ofimáticos; Dropbox [3], para almacenamiento de ficheros; Spotify [4], para reproducción de música bajo demanda y Netflix [5], para la visualización de series y películas bajo demanda.

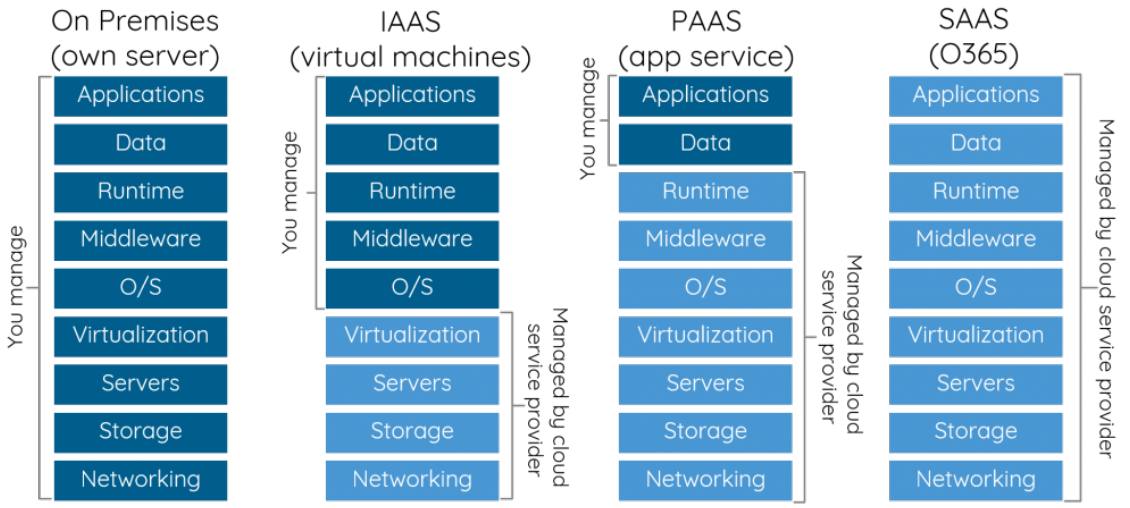

Figura 1.1: Diferencias entre los modelos de servicios Cloud

Fuente: [6]

La gran aceptación del Cloud Computing y la popularidad de las tecnologías de contenedores de *software*[2] propició que, el 13 de noviembre de 2014, el proveedor de Cloud público Amazon Web Services presentara su servicio AWS Lambda [7]. AWS Lambda [8] define un nuevo modelo en la computación en la nube, denominado FaaS (funciones como servicio) [9], englobado típicamente dentro de la categoría Serverless Computing, que implica el uso de servicios gestionados por el proveedor donde el usuario no necesita interactuar y escalar la infraestructura subyacente, pues esta tarea se lleva a cabo por parte del proveedor.

El modelo de servicio FaaS consiste en la definición de funciones en un determinado lenguaje de programación sobre un proveedor Cloud. Las funciones se ejecutan sobre contenedores que son creados para responder a peticiones, normalmente generadas por la interacción de los usuarios, y eliminados automáticamente cuando dejan de utilizarse.

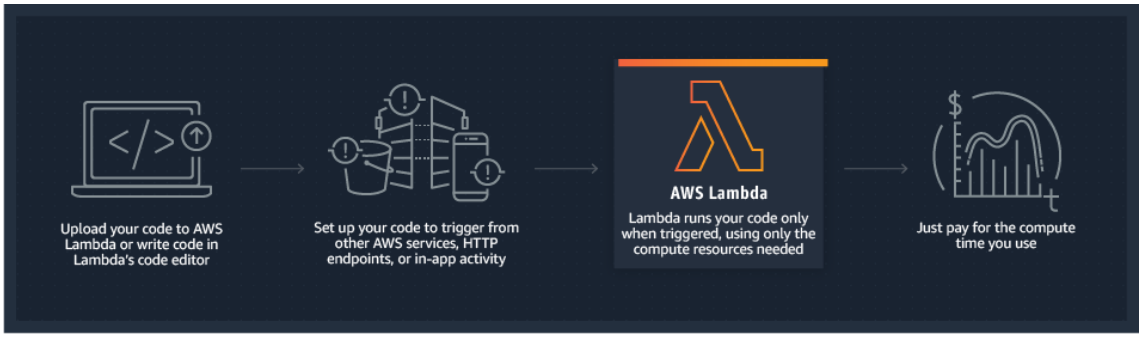

Figura 1.2: Esquema de funcionamiento de AWS Lambda

Fuente: [8]

[2] https://www.redhat.com/es/topics/containers

Gracias a esto no es necesario disponer de un servicio constantemente activo como sucede con los modelos mencionados anteriormente. Además, el proveedor asegura escalabilidad de forma transparente a los usuarios y solo se cobra por consumo real, es decir, por el tiempo de ejecución de la función en base a los recursos asignados a la misma.

1.2 Motivación

La aparición del Serverless Computing supone una revolución en el desarrollo de servicios web. Los desarrolladores ya no se tienen que preocupar por el aprovisionamiento y escalado de la infraestructura, por lo que se pueden centrar en la lógica de sus aplicaciones.

Sin embargo, las plataformas de funciones como servicio de los principales proveedores de Cloud público tienen algunas limitaciones que impiden que sus beneficios puedan aprovecharse para ejecutar aplicaciones generales. Estas limitaciones suelen estar definidas por los lenguajes de programación soportados por la plataforma, así como por el tiempo máximo de ejecución y memoria asignable a las funciones.

Con el fin de ejecutar aplicaciones genéricas sobre la plataforma FaaS AWS Lambda, el grupo de Grid y Computación de Altas Prestaciones (GRyCAP)[3] de la Universitat Politècnica de València desarrolló SCAR [10]. SCAR es una herramienta que permite ejecutar contenedores Docker [11] dentro de funciones para así permitir la ejecución de aplicaciones generales sobre entornos de ejecución predefinidos por el usuario, no únicamente funciones escritas en los lenguajes de programación soportados por el proveedor. Además, la herramienta define un modelo de programación que consiste en la ejecución automática de funciones ante la carga de ficheros en el sistema de almacenamiento Amazon S3 [12]. De esta forma, se pueden explotar las capacidades de AWS Lambda para procesar ficheros utilizando aplicaciones generales.

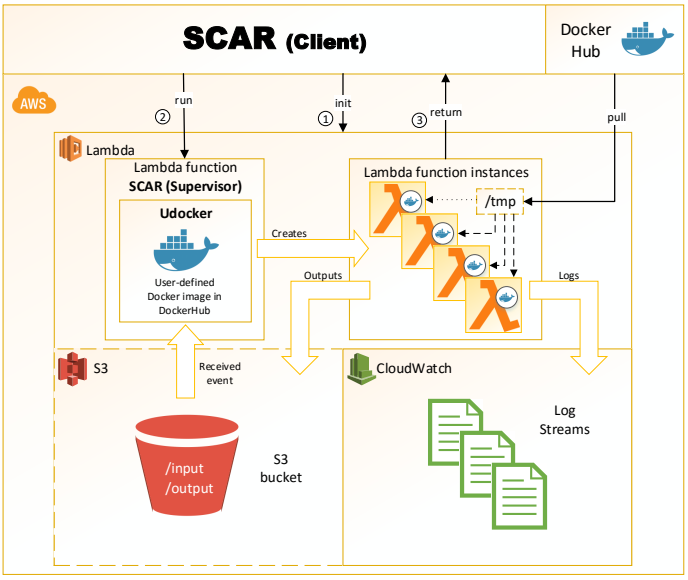

Figura 1.3: Arquitectura general de SCAR

Fuente: [13]

[3] https://www.grycap.upv.es

No obstante, las limitaciones de tiempo máximo de ejecución (15 minutos), así como la cantidad máxima de memoria asignable a las funciones (3008 MB) siguen siendo inevitables. Además, las imágenes de los contenedores soportados por SCAR no pueden superar los 512MB de tamaño, ya descomprimidas, debido a la capacidad de almacenamiento proporcionada por AWS Lambda. Otro punto a mejorar sería el soporte a sistemas de almacenamiento externos. Esto se debe principalmente a la necesidad de mantener la información en entornos protegidos, siendo comunes los datos sensibles en ámbitos como el científico o médico.

Por otra parte, las ventajas proporcionadas por el paradigma Serverless han propiciado la aparición de múltiples marcos de trabajo para definir funciones sobre plataformas de orquestación de contenedores como Kubernetes [14]. Estas plataformas, al poder ejecutarse sobre cualquier tipo de infraestructura, ya sea pública o privada, no tienen las restricciones nombradas anteriormente.

Así pues, es importante disponer de una plataforma capaz de desplegarse sobre diversas infraestructuras sin las limitaciones identificadas en AWS Lambda. Todas estas circunstancias motivan la realización de este trabajo, que se plantea como una colaboración con el grupo de Grid y Computación de Altas Prestaciones de la Universitat Politècnica de València.

1.3 Objetivos

El principal objetivo identificado en este trabajo de fin de máster consiste en la integración de diferentes componentes *software* para definir una plataforma Serverless para el procesamiento de ficheros. Esta plataforma debe explotar el modelo de procesamiento de ficheros definido en SCAR y, además, se debe poder ejecutar sobre cualquier infraestructura, permitiendo así que cualquier organización pueda beneficiarse del paradigma de ejecución de funciones dirigido por eventos. De esta manera, el procesamiento de los ficheros se desencadenará automáticamente ante la subida de los ficheros a un sistema de almacenamiento. Para ello se utilizarán herramientas de código abierto que suplan a las proporcionadas típicamente por un proveedor de Cloud público, aparte de las necesarias como base para la creación de la propia plataforma.

Una vez lograda la interconexión de todos los componentes se recogerán las configuraciones necesarias para automatizar todo el proceso de despliegue. Para esta tarea será necesaria alguna herramienta de aprovisionamiento de infraestructuras, que se encargará de replicar el despliegue y la configuración de los componentes independientemente del sistema operativo y el proveedor subyacente.

Otro objetivo importante es la capacidad de conexión con un sistema de almacenamiento de ficheros externo, soportando así flujos de trabajo híbridos para procesar ficheros. De esta forma se podrá desacoplar la capacidad de almacenamiento de la de cómputo para mantener datos sensibles en entornos más seguros.

Por último, se deben elaborar casos de uso que exploten la capacidad de la plataforma con el fin de demostrar su operatividad y analizar su funcionamiento.

1.4 Planificación temporal

En la Figura 1.4 mostrada a continuación se puede observar el diagrama de Gantt con la planificación temporal seguida durante el desarrollo de este trabajo.

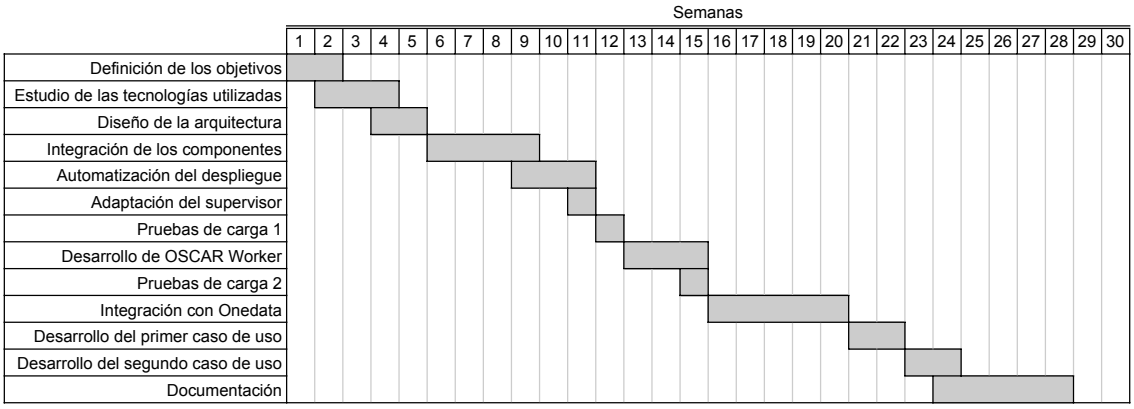

Figura 1.4: Planificación temporal del trabajo

1.5 Estructura de la memoria

Esta memoria está estructurada de la siguiente forma:

- En el Capítulo 2 se introducen los diferentes aspectos tecnológicos relacionados con el proyecto. Además, se exponen las herramientas y componentes *software* escogidos para la creación de la plataforma, tratando de justificar su selección.
- El Capítulo 3 detalla la arquitectura de la plataforma, profundizando en los pasos realizados para lograr la integración de todos los componentes. También se muestran algunos problemas encontrados, explicando las soluciones escogidas y su implementación.
- En el Capítulo 4 se presentan dos casos de uso de la plataforma desarrollada, mostrando los aspectos más relevantes de su implementación y los resultados obtenidos. El primero consiste en una aplicación de reconocimiento de plantas en imágenes, basado en técnicas de aprendizaje profundo. Por otra parte, el segundo caso de uso trata de demostrar las ventajas de la plataforma para extraer información a partir de datos abiertos. Para ello se conectarán dos funciones a través de un sistema de almacenamiento externo, creando así un flujo de trabajo híbrido.
- Finalmente, el Capítulo 5 recoge las conclusiones del trabajo, así como las posibles futuras ampliaciones y las contribuciones científicas relacionadas con el mismo.

CAPÍTULO 2
Conceptos iniciales y tecnologías

En este capítulo de la memoria se definen los conceptos y tecnologías que sirven como base para el desarrollo del proyecto.

Las primeras cinco secciones describen diferentes conceptos tecnológicos y algunas herramientas *software* que los aplican, todos ellos fundamentales para la plataforma desarrollada. Por otra parte, la última sección trata otras tecnologías y herramientas utilizadas que no se pueden clasificar en ninguno de los apartados anteriores.

2.1 Infraestructura como servicio

El concepto de infraestructura como servicio (IaaS) consiste en la capacidad de ofrecer sistemas informáticos, basados en máquinas virtuales, bajo demanda.

En la práctica, los proveedores de servicios Cloud utilizan una serie de aplicaciones que sirven como interfaces entre los usuarios y sus sistemas físicos. Estas aplicaciones permiten que diferentes usuarios puedan solicitar recursos informáticos a través de Internet cuando lo consideren oportuno sin tener que preocuparse por los componentes *hardware* subyacentes. De esta forma se abstrae un grado de complejidad en la gestión de los sistemas, siendo el proveedor Cloud el encargado de mantener dichos componentes. Sin embargo, los usuarios deben encargarse de administrar los sistemas operativos de las máquinas virtuales utilizadas, así como las aplicaciones instaladas sobre los mismos. Esto significa que siguen siendo necesarias ciertas tareas de mantenimiento, como actualizaciones de *software* o copias de seguridad, para evitar problemas.

IaaS es considerado como el servicio de más bajo nivel (menor abstracción) de los ofrecidos en la nube. Una de sus características principales es el modelo de pago por uso, normalmente tarificado en base a los minutos que las máquinas virtuales están encendidas y las características de las mismas (número de CPUs, memoria, etc).

Es importante mencionar que este concepto no está obligatoriamente relacionado con el contrato de servicios a un proveedor público. Cualquier compañía o entidad que disponga de una infraestructura propia puede beneficiarse de las ventajas que proporciona utilizando alguna herramienta como OpenStack [15] u OpenNebula [16], mencionadas en los siguientes apartados.

2.1.1. Amazon EC2

Amazon EC2 (Elastic Compute Cloud) [17] es el servicio de computación bajo demanda (IaaS) ofrecido por el proveedor de Cloud público Amazon Web Services. Proporciona la capacidad de desplegar instancias (máquinas virtuales) con una amplia variedad de configuraciones, denominadas *tipos de instancias* [18], que pueden clasificarse en cinco grupos en función de sus características y su coste: uso general, optimizadas para informática, optimizadas para memoria, informática acelerada y optimizadas para almacenamiento.

A continuación se resumen algunas de las características más relevantes del servicio:

- Posibilidad de iniciar instancias en distintas ubicaciones. Actualmente AWS cuenta con 66 zonas de disponibilidad ubicadas en 21 regiones geográficas del mundo.
- Reglas de escalado automático. Se pueden definir reglas para ajustar el tamaño de grupos de instancias según las condiciones deseadas.
- Posibilidad de pausar y reanudar instancias. Mientras estén pausadas solo se cobrará por el almacenamiento.
- Posibilidad de alquilar instancias dedicadas, es decir, máquinas sin virtualizar con acceso directo al *hardware*.
- Direcciones IP elásticas. Una cuenta de AWS puede disponer de una o múltiples direcciones IP estáticas de este tipo. Pueden ser asociadas a instancias y reasignarse a otras según las necesidades del usuario.

Para indicar el sistema operativo se debe seleccionar una AMI (Amazon Machine Image). Una AMI es una imagen de un sistema operativo preconfigurado, de forma que los usuarios no tienen que esperar a que se instale el sistema o determinado software, simplemente escogen la que mejor se ajuste a sus necesidades. Es importante mencionar que aunque existe una gran cantidad de AMIs gratuitas en la plataforma de AWS, algunas son de pago. La tarificación se realiza en función al tipo de instancia, la AMI escogida y el tiempo que la instancia permanezca encendida, entre otros factores.

Amazon EC2 se utiliza en este proyecto durante las pruebas de despliegue de la plataforma desarrollada sobre AWS, para demostrar que puede desplegarse sobre múltiples plataformas Cloud.

2.1.2. OpenNebula

OpenNebula [16] es una herramienta de código abierto para administrar infraestructuras de centros de datos. Su desarrollo comenzó en el año 2005 en un grupo de investigación de la Universidad Complutense de Madrid. Actualmente es utilizada por una gran cantidad de entidades académicas y científicas.

Su principal objetivo es ofrecer plataformas Cloud de infraestructura como servicio sobre recursos *hardware* privados. Para ello cuenta, entre otras, con las siguientes características:

- Interfaces para consumidores Cloud.

- Gestión de máquinas virtuales y contenedores.
- Gestión de redes virtuales.
- Interfaces para administradores y usuarios avanzados.
- Tenencia múltiple y seguridad.
- Provisión bajo demanda de centros de datos virtuales

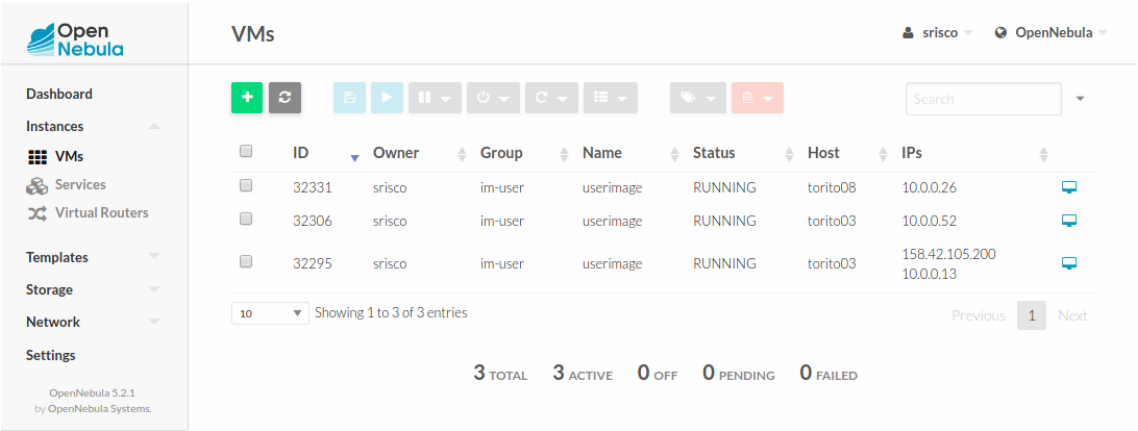

Figura 2.1: Interfaz web de OpenNebula

OpenNebula se utiliza en este trabajo como uno de los gestores de Cloud privado donde se despliega la plataforma desarrollada.

2.1.3. OpenStack

OpenStack [15] es un proyecto de código abierto creado a raíz de una colaboración entre la NASA [19] y el proveedor Cloud Rackspace [20] en el año 2010.

Al igual que OpenNebula, OpenStack ofrece una serie de herramientas para proporcionar plataformas Cloud de infraestructura como servicio. Sin embargo, OpenStack está diseñado de forma modular, lo que significa que los administradores pueden instalar únicamente los módulos que consideren necesarios en función de los servicios que quieran que ofrezca su infraestructura. Algunos de sus componentes más relevantes son:

- **Nova**. Es el componente principal de OpenStack, encargado de gestionar las máquinas virtuales.
- **Glance**. Encargado de la gestión de las imágenes de sistema a utilizar en las instancias.
- **Neutron**. Es el sistema de gestión de redes y direcciones IP de OpenStack.
- **Horizon**. Es la interfaz web de OpenStack. Ofrece a los usuarios y administradores una manera sencilla para interactuar con la infraestructura.
- **Keystone**. Es el servicio encargado de gestionar los usuarios y sus permisos.

- **Cinder**. Es el sistema de almacenamiento de bloques de OpenStack que permite crear y asignar volúmenes lógicos a las instancias.

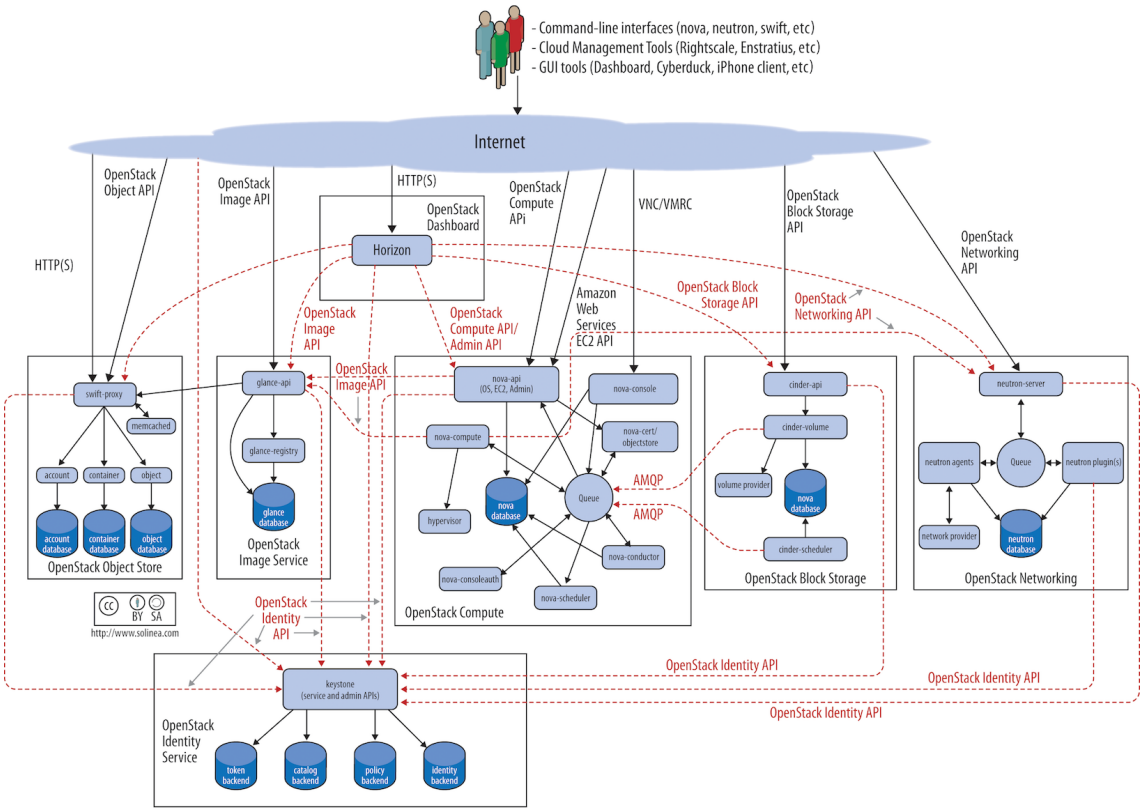

Figura 2.2: Arquitectura general de OpenStack

Fuente: [21]

Al igual que en el caso anterior, OpenStack se utiliza en este trabajo como uno de los gestores de Cloud privado donde se despliega la plataforma desarrollada.

2.2 Contenedores de *software*

Gracias a diferentes características de los sistemas operativos GNU/Linux [22] para aislar y gestionar los recursos de determinados grupos de procesos, aparece el concepto de contenedores de *software*.

La principal idea de este concepto es la encapsulación de aplicaciones junto a todas sus dependencias en contenedores, término que se utiliza por la similitud con los contenedores de mercancías, que pueden ser transportados por trenes, camiones o barcos independientemente de su contenido. De igual manera, estas aplicaciones encapsuladas funcionarán en cualquier sistema sin necesidad de modificaciones ni instalación de dependencias.

Al igual que ocurre con las máquinas virtuales, existen capturas del estado de los contenedores, conocidas como imágenes. Así pues, el término *contenedores* queda reservado para las instancias en ejecución de una imagen. Estas imágenes pueden distribuirse fácilmente a través de *registros*, que no son más que almacenes de imágenes privados o públicos.

Este sistema de empaquetado y distribución de aplicaciones *software* supuso toda una revolución y tuvo una gran acogida en su llegada. Se trata de una solución mucho más ligera y rápida que la virtualización, normalmente utilizada también para este propósito. El auge de esta tecnología, en especial para la distribución y despliegue de servicios, propició la aparición de *sistemas orquestadores de contenedores*. Estos sistemas suelen instalarse sobre un conjunto de máquinas, que pueden ser físicas o virtuales, conformando clústeres. Su finalidad no es solo la puesta en marcha de contenedores, sino la gestión de lo mismos, ocupándose de tareas como la configuración automática, balanceo de carga, escalado y tolerancia a fallos, entre otras.

2.2.1. Docker

Docker [11] es un proyecto de código abierto que permite la creación y puesta en marcha de contenedores de *software*. Normalmente es utilizado en sistemas operativos GNU/Linux, sin embargo, existen versiones para otros sistemas como Windows y macOS. La aplicación general, conocida como Docker Engine, está compuesta por los siguientes componentes:

- **Servidor**. Es la pieza central de la aplicación. Se trata de un proceso que, estando activo, se encarga de la creación de contenedores sobre el sistema operativo de la máquina.
- **API REST**[1]. Define una interfaz para que el cliente u otros programas puedan comunicarse con el servidor.
- **Ciente**. Herramienta de linea de comandos *docker*, que se comunica con el servidor a través de la API REST. A través del cliente se pueden gestionar imágenes y contenedores, así como redes y volúmenes de datos asignados a los mismos.

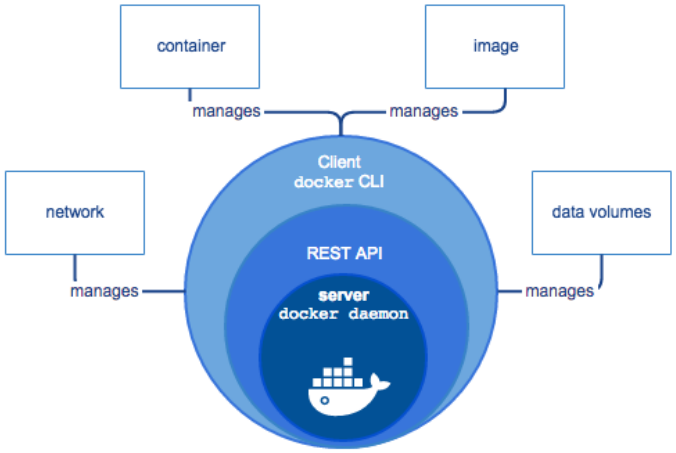

Figura 2.3: Arquitectura general de Docker

Fuente: [23]

[1] https://bbvaopen4u.com/es/actualidad/api-rest-que-es-y-cuales-son-sus-ventajas-en-el-desarrollo-de-proyectos

La creación de imágenes de contenedores se hace a través de la definición de ficheros denominados *Dockerfile*. Estos ficheros especifican las características del contenedor a construir, como la imagen base a utilizar (registrada en Docker Hub), los archivos locales que se quieren incluir o los comandos a ejecutar para instalar dependencias.

En este trabajo se utiliza Docker como herramienta para encapsular la aplicación del usuario encargada de procesar los ficheros de datos sobre la plataforma desarrollada.

2.2.2. Docker Registry

Docker Registry [24] es la solución del proyecto Docker para la creación de almacenes de imágenes, conocidos como *registros*.

La herramienta proporciona un servicio mediante el cual usuarios u organizaciones pueden montar sus propios registros. Estos almacenes de imágenes pueden ser públicos o privados, lo que resulta de gran utilidad cuando se utilizan contenedores que incluyen aplicaciones con licencias restrictivas o datos sensibles. Para ello, la herramienta cuenta con un sistema de control de usuarios.

En este trabajo se ha utilizado Docker Registry para proporcionar un registro privado en el que almacenar las imágenes de las funciones dentro de la plataforma desarrollada.

2.2.3. Docker Hub

Docker Hub [25] es el mayor registro de contenedores público existente en la actualidad. A diferencia de Docker Registry, Docker Hub permite que los usuarios se den de alta en su plataforma de forma gratuita para así poder subir imágenes de contenedores a través de Internet. Además, todas las imágenes almacenadas en Docker Hub pueden ser descargadas sin necesidad de poseer una cuenta. Es el registro utilizado por defecto por el cliente oficial de Docker.

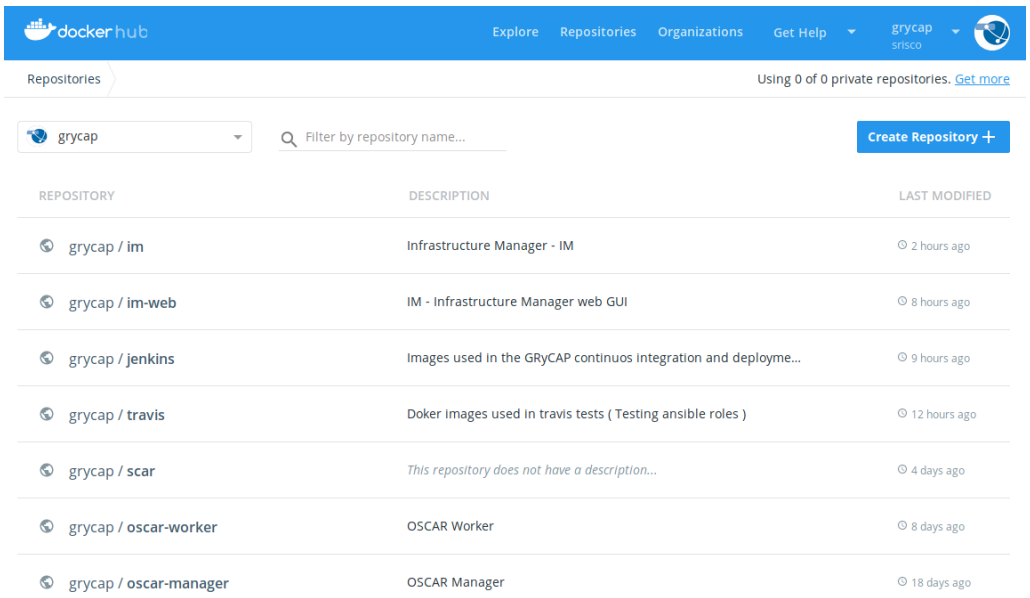

Figura 2.4: Perfil del grupo de Grid y Computación de Altas Prestaciones en Docker Hub

Docker Hub se utiliza en este trabajo para almacenar en un registro público las imágenes que contienen las aplicaciones desarrolladas así como para utilizar imágenes de contenedores de sistemas operativos base a partir del cual se general las imágenes específicas con las aplicaciones de usuario.

2.2.4. Kubernetes

Kubernetes [14] es un sistema de orquestación de contenedores mantenido por la Cloud Native Computing Foundation [26]. En la actualidad es uno de los proyectos de *software* libre con mayor número de contribuciones.

Este sistema, al igual que la mayoría de orquestadores de contenedores de *software*, funciona sobre un conjunto de ordenadores, denominados *nodos*. Estas agrupaciones definen lo que se conoce como un clúster. Los clústeres de Kubernetes siempre deben tener, al menos, un nodo maestro. El nodo maestro incluye una serie de componentes, entre los que destacan: la API del servidor, que define una interfaz para gestionar la comunicación con el exterior; el planificador, que se encarga de asignar los contenedores en el resto de nodos (nombrados *workers* o *working nodes*); y la base de datos *etcd* [27], donde se almacena la configuración del clúster.

Por otra parte, los *working nodes* deben contar con: un componente *software*, llamado *kubelet*, que es el agente encargado de recibir las ordenes del nodo maestro; un *proxy*, componente que permite la abstracción de los servicios de red de los contenedores; y el propio entorno de ejecución de contenedores, que normalmente suele ser el sistema Docker.

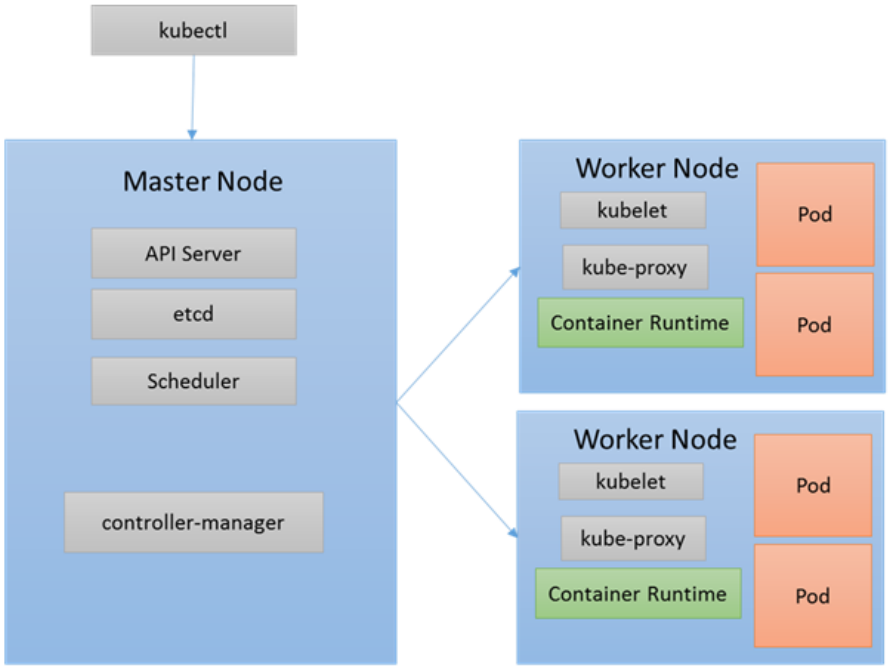

Figura 2.5: Arquitectura simplificada de Kubernetes

Fuente: [28]

El objetivo de Kubernetes es abstraer la configuración de despliegue de aplicaciones basadas en contenedores, para ello define una serie de primitivas que facilitan estas tareas, entre las que destacan:

- **Pods**. Son las unidades básicas de computación en Kubernetes. Están compuestos por uno o más contenedores, en los que se pueden definir volúmenes que pueden ser directorios locales o en red. Es importante mencionar que cada *pod* tiene asignada una dirección IP.
- **Deployments**. Se trata de una primitiva para definir despliegues de aplicaciones, añadiendo un nivel de abstracción a de los *pods*. Los *deployments* se encargan de gestionar el ciclo de vida de una aplicación basada en contenedores, asegurando que siempre haya la cantidad de *pods* especificada en ejecución. Además, permiten definir reglas para el escalado de la aplicación en cuestión.
- **Services**. Es la primitiva que se encarga de asignar puntos de entrada a un conjunto de *pods*, normalmente desplegados a través de un *deployment*. En los *services* se pueden especificar ciertas características de descubrimiento como el puerto asignado, el nombre DNS o una dirección IP estable.
- **Jobs**. Se trata de una primitiva básica similar a los *pods*. Sin embargo, los *pods* suelen componerse por contenedores que albergan servicios cuya duración es indefinida, como por ejemplo un servidor web; mientras que los *jobs* se componen por contenedores que realizan trabajos específicos que terminan tras su ejecución.

En este trabajo se ha utilizado Kubernetes como sistema base de la plataforma donde se despliegan todos los componentes utilizados, así como los desarrollos propios.

2.2.5. Kaniko

Kaniko [29] es una herramienta que permite construir imágenes de contenedores de *software* desde un contenedor o un clúster Kubernetes a partir de un fichero Dockerfile.

Su principal ventaja es que no requiere de permisos de administración en el sistema donde se va a crear la imagen, al contrario que ocurre con el cliente de Docker u otras herramientas similares. Además, permite subir automáticamente la imagen generada a un registro de contenedores, ya sea público o privado.

En este trabajo se ha utilizado Kaniko para crear las imágenes de las funciones utilizadas en la plataforma a través de *jobs* de Kubernetes.

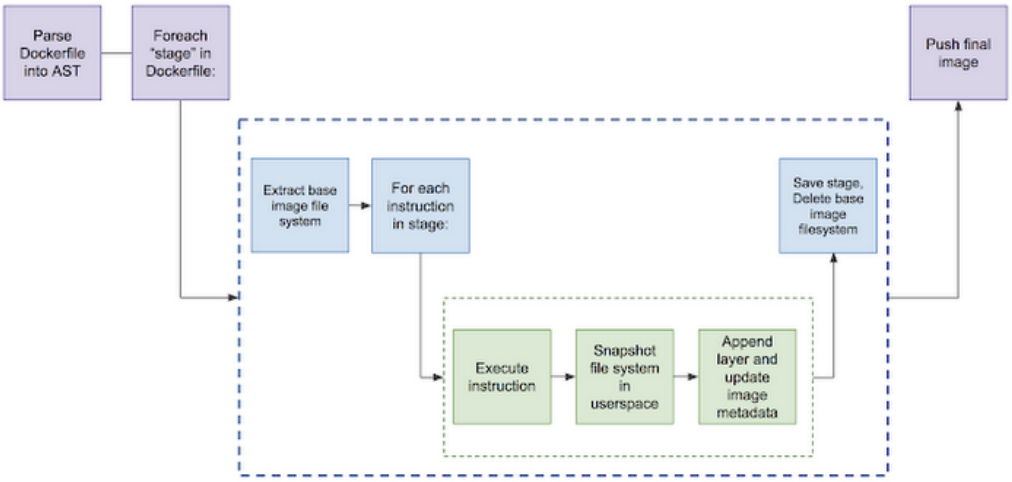

Figura 2.6: Arquitectura general de Kaniko

Fuente: [30]

2.3 Serverless Computing

Serverless Computing es el paradigma sobre el cual se basa este trabajo. Como se ha mencionado en el Capítulo 1, este paradigma trata de abstraer de la gestión de servidores (aprovisionamiento, configuración, escalado, etc) para que los usuarios, en este caso desarrolladores, puedan enfocarse en la lógica de sus aplicaciones.

Este paradigma se relaciona normalmente con el modelo FaaS, que consiste en la definición de funciones en un determinado lenguaje de programación que son ejecutadas sobre un proveedor Cloud. Para gestionar este tipo de servicios se utilizan tecnologías de contenedores *software*, que son los entornos sobre los cuales se ejecutan las funciones, si bien su gestión suele quedar oculta a los desarrolladores.

2.3.1. AWS Lambda

AWS Lambda [8] es el servicio Serverless, basado en el modelo de funciones como servicio, del proveedor Cloud público AWS.

Este servicio permite la definición de funciones en determinados lenguajes de programación y se encarga del escalado automático de los recursos dependiendo de la cantidad de invocaciones recibidas. Las invocaciones a las funciones se pueden realizar a través de otros servicios del proveedor, siendo el más común API Gateway [31], que permite definir puntos de acceso HTTP y así poder crear aplicaciones web sin preocuparse por la infraestructura subyacente. Además, el servicio se tarifica por tiempo real de ejecución de las funciones, por lo que no supone ningún gasto mientras que no se utiliza a pesar de mantenerse disponible.

Sin embargo, AWS Lambda tiene una serie de limitaciones que, tal y como se expuso en el Capítulo 1, motivaron el desarrollo de la herramienta SCAR y, a continuación, el trabajo llevado a cabo en esta memoria.

2.3.2. SCAR

Como se ha mencionado en el capítulo de introducción, SCAR [13] [10], acrónimo de *Serverless Container-aware ARchitectures*, es la herramienta desarrollada por el grupo de Grid y Computación de Altas Prestaciones de la Universitat Politècnica de València para ejecutar aplicaciones basadas en contenedores Docker sobre AWS Lambda.

Gracias a SCAR se puede eludir la limitación de los lenguajes soportados en el servicio y se pueden ejecutar aplicaciones generales, marcos de trabajo que requieran librerías específicas o funciones programadas en otros lenguajes de programación. Sin embargo, existen restricciones en el entorno de ejecución de AWS Lambda que son inevitables, como el tamaño máximo de memoria RAM, el tiempo máximo de ejecución y la cantidad de almacenamiento en disco disponible para las funciones. Estas restricciones siguen suponiendo serias limitaciones cuando se pretenden ejecutar programas complejos como, por ejemplo, aplicaciones científicas que pueden depender de librerías externas o pueden haber sido programadas con lenguajes de programación interpretados no soportados por AWS Lambda.

Por otra parte, otra de las características principales de SCAR es su modelo de programación, que define un sistema de procesamiento de ficheros altamente escalable basado en eventos [32]. Este modelo permite usar la plataforma para procesar archivos cargados en sistema de almacenamiento propio del proveedor, Amazon S3 [12], sin que los usuarios tengan que realizar ninguna acción adicional. Simplemente se suben ficheros a una carpeta y se descargan los resultados de otra.

Como características adicionales, esta herramienta también permite generar puntos de entrada a las funciones a través del protocolo HTTP utilizando el servicio Amazon API Gateway [31]; y almacena y proporciona fácil acceso a *logs* gracias al servicio Amazon CloudWatch [33].

2.3.3. OpenFaaS

OpenFaaS [34] es una herramienta de código abierto que permite desplegar un conjunto de servicios para soportar un esquema de funciones como servicio (FaaS) basado en contenedores Docker y ejecutándose sobre un clúster Kubernetes, además de Docker Swarm [35].

La arquitectura general de OpenFaaS se muestra en la Figura 2.7 y consta de un API Gateway encargado de recibir las peticiones de invocación a funciones, que se redirigen al servicio de Kubernetes correspondiente. Estas invocaciones son eventos que al llegar al contenedor pasan al componente denominado *Watchdog*, que se encarga de ejecutar y monitorizar la función.

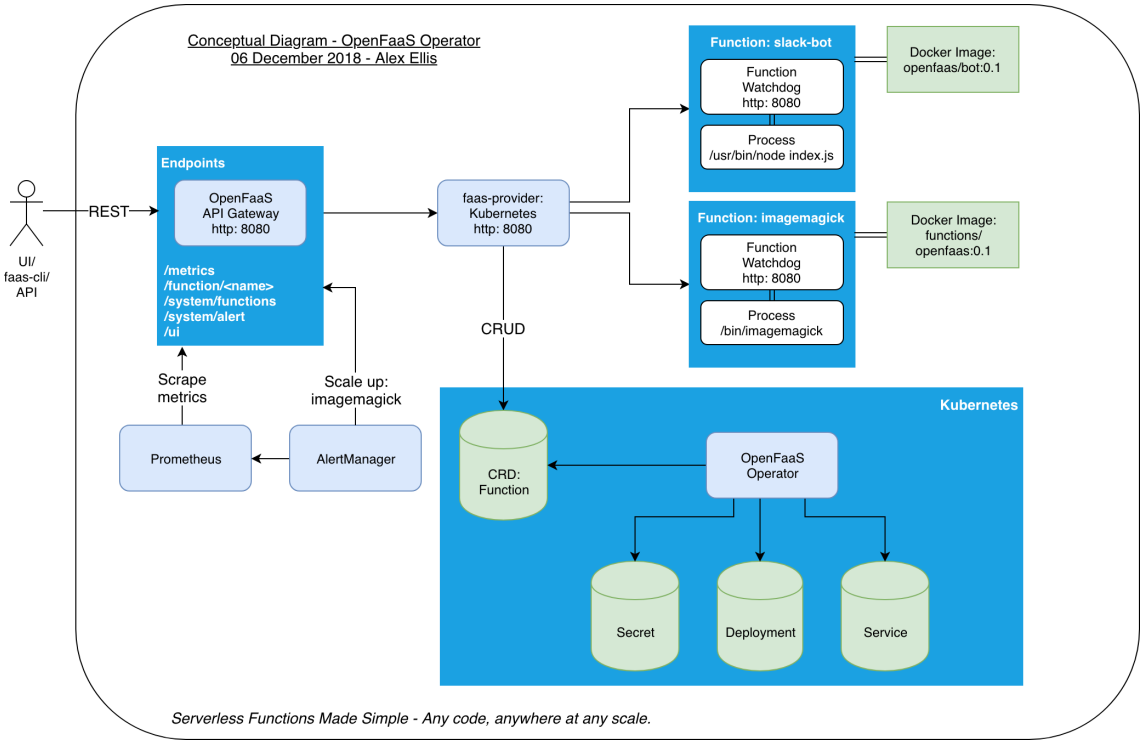

Figura 2.7: Arquitectura general de OpenFaaS

Fuente: [36]

En el trabajo se utiliza OpenFaaS como plataforma base para la definición y ejecución de funciones, sobre la que se soportan el resto de funcionalidades necesarias para la consecución de los objetivos definidos, como la integración con sistemas de almacenamiento externos.

2.4 Sistemas de almacenamiento de datos

Uno de los servicios más importantes de la nube, aparte de la computación, es el almacenamiento de datos. Los sistemas que proporcionan estos servicios no solo ofrecen la capacidad de almacenar información en centros de datos remotos, también cuentan con mecanismos para garantizar altos niveles de disponibilidad, confidencialidad e integridad. Para ello, los proveedores realizan replicas de los datos en diferentes localizaciones, utilizan sistemas de seguridad para evitar el acceso de usuarios no deseados y cuentan con infraestructuras de red capaces de transmitirlos a altas velocidades.

2.4.1. Amazon S3

Amazon S3 [12] es un sistema de almacenamiento de objetos altamente escalable y que ofrece replicación automática de los ficheros que allí se almacenan. Ofrece diferentes clases de almacenamiento en función de la durabilidad y la disponibilidad de acceso de los datos, y que involucran diferentes opciones de coste.

Amazon S3 ofrece un API de para gestionar la subida, descarga y gestión de los datos y posibilita el acceso mediante protocolos estándar como HTTP y BitTorrent.

2.4.2. MinIO

MinIO [37] es una herramienta de código abierto que implementa el API de Amazon S3 para ofrecer una solución abierta de almacenamiento de datos, soportando esquemas de replicación de datos en modo distribuido. De esta manera, permite utilizar las mismas herramientas cliente que se utilizan para interactuar con Amazon S3, pero utilizando Minio como sistema de almacenamiento.

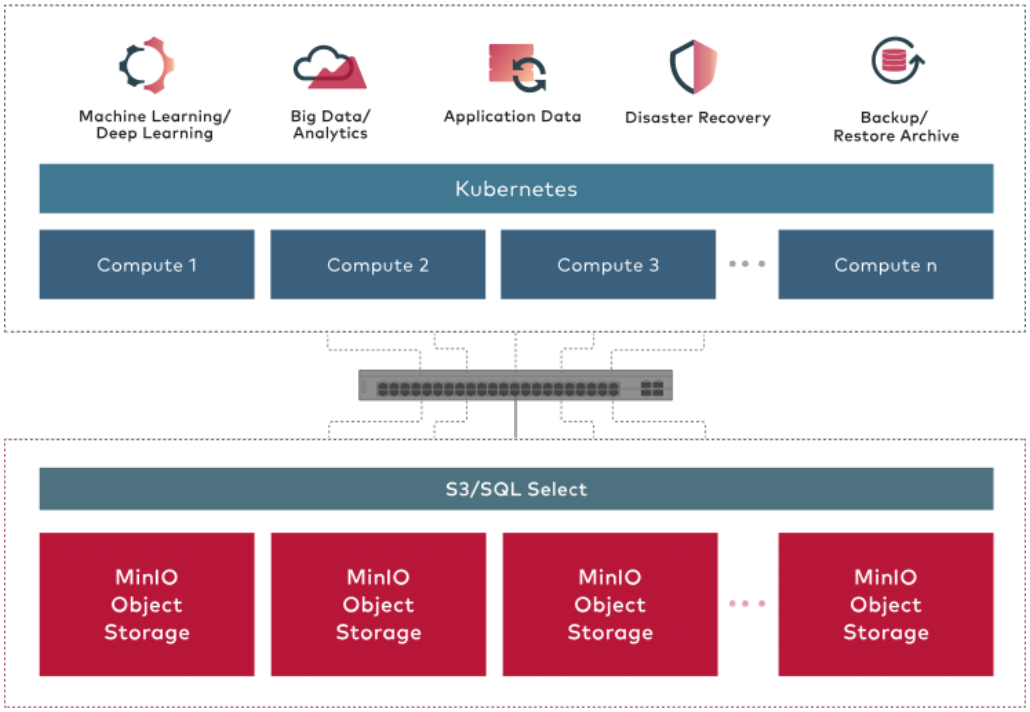

Figura 2.8: Arquitectura general de MinIO

Fuente: [38]

En este trabajo MinIO se utiliza como una de las opciones de almacenamiento de ficheros que permiten disparar los eventos necesarios para el posterior procesado de los mismos en la plataforma de funciones como servicio.

2.4.3. Onedata

Onedata [39] es un sistema de almacenamiento global orientado a la comunidad científica. Se caracteriza por ofrecer acceso unificado a datos en entornos distribuidos globalmente, permitir la colaboración entre usuarios y facilitar el acceso a los datos almacenados. Los conceptos fundamentales en los que se basa Onedata son:

- **Espacios**. Son volúmenes virtuales distribuidos donde los usuarios pueden organizar sus datos.
- **Proveedores**. Entidades que dan soporte a los espacios de usuario con recursos de almacenamiento reales expuestos a través de los servicios de Oneprovider [40].
- **Zonas**. Federaciones de proveedores, que permiten la creación de comunidades cerradas o interconectadas, gestionadas por los servicios de Onezone [41].

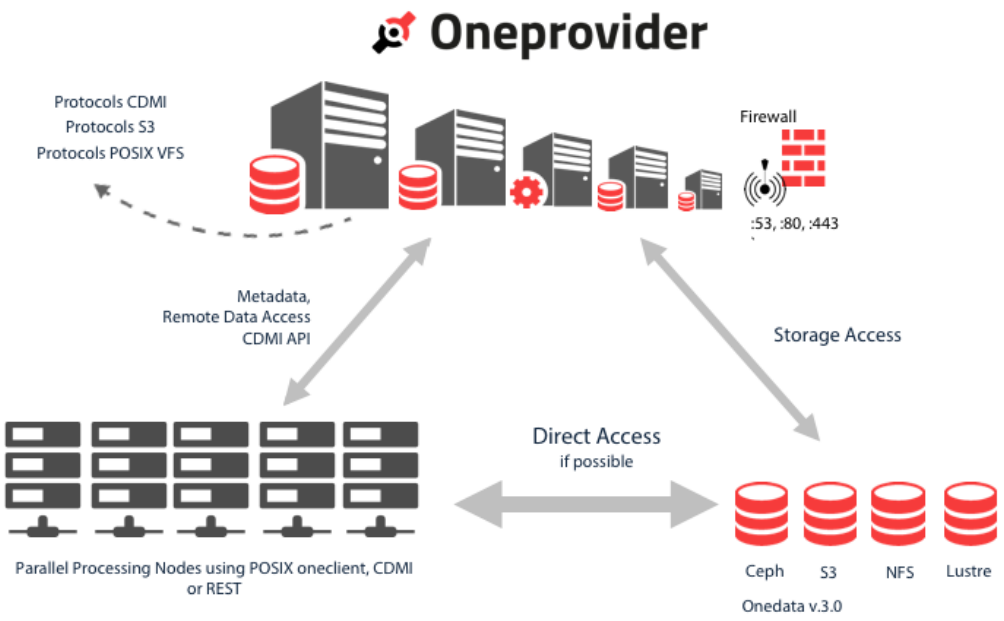

Figura 2.9: Arquitectura general de Oneprovider

Fuente: [40]

En este trabajo se utiliza Onedata como una opción de sistema de almacenamiento externo tanto para subir los ficheros que dispararán el procesado automático de los mismos como para almacenar los resultados.

2.5 Herramientas para aprovisionamiento de infraestructuras

Debido a la creciente complejidad de las arquitecturas necesarias para desplegar aplicaciones y servicios, así como la necesidad de replicarlas fácilmente, surgen las herramientas para aprovisionamiento de infraestructuras.

2.5.1. Ansible

Ansible [42] es una herramienta publicada como *software* libre para configurar y administrar sistemas de forma automática. A diferencia de otras herramientas similares, Ansible no requiere instalar ninguna aplicación en las máquinas a configurar, sino que se conecta a ellas a través del protocolo SSH[2].

La definición de configuraciones específicas en Ansible se hace mediante *roles*. Los roles se componen por un conjunto de ficheros en formato YAML[3] en los cuales se indican los comandos a ejecutar, siguiendo un orden, para configurar el sistema en cuestión. Además, esta herramienta cuenta con el servicio Ansible Galaxy [43], que es una plataforma pública en la que se pueden registrar los roles creados para que otros usuarios puedan utilizarlos. La principal ventaja de estos roles de Ansible es que posibilitan la definición

[2] https://www.hostinger.es/tutoriales/que-es-ssh
[3] https://yaml.org/

de recetas de instalación de software que funcionen dependiendo del sistema operativo en el que se ejecuten, facilitando así el despliegue multi-plataforma de las aplicaciones.

En este trabajo se ha utilizado Ansible para definir roles de instalación y configuración de los componentes de la plataforma desarrollada.

2.5.2. CLUES

CLUES [44] es un gestor de elasticidad para infraestructuras de cómputo de tipo clúster que posibilita el despliegue automático de máquinas virtuales en función de diferentes políticas de escalado configurables por el usuario. Está integrado con diferentes sistemas de colas como PBS/Torque[4] o SLURM[5], así como con plataformas de orquestación de contenedores como Apache Mesos[6] o Kubernetes.

En el contexto de este trabajo, CLUES se utiliza como gestor de elasticidad de los clústeres Kubernetes de manera que sea posible de escalar automáticamente sobre una plataforma Cloud, mediante el despliegue y terminación automática de máquinas virtuales en función de los trabajos (*jobs*) en ejecución dentro del clúster.

2.5.3. IM

Infrastructure Manager [45] [46] es una herramienta de código abierto que permite la definición y despliegue de infraestructuras virtuales complejas sobre múltiples proveedores Cloud, ya sean públicos como Amazon Web Services, Google Cloud Platform[7] o Microsoft Azure[8] o bien privados como OpenNebula u OpenStack, entre otros. Soporta un lenguaje de definición de aplicaciones llamado RADL (*Resource Application & Description Language*) [47] así como el estándar para descripción de arquitecturas de aplicaciones para la nube OASIS TOSCA (*Topology and Orchestration Specification for Cloud Applications*) [48].

La Figura 2.10 resume la arquitectura del IM, que ofrece tanto una interfaz web como una interfaz de línea de comandos. La parte servidora se encarga de gestionar el despliegue y la configuración automatizada mediante roles de Ansible. Es posible encontrar una explicación más detallada del uso del IM para el despliegue de clústeres en el trabajo de Caballer, Donvito, Moltó y col. [49].

[4] https://www.adaptivecomputing.com/products/torque/
[5] https://slurm.schedmd.com/
[6] http://mesos.apache.org/
[7] https://cloud.google.com/
[8] https://azure.microsoft.com/es-es/

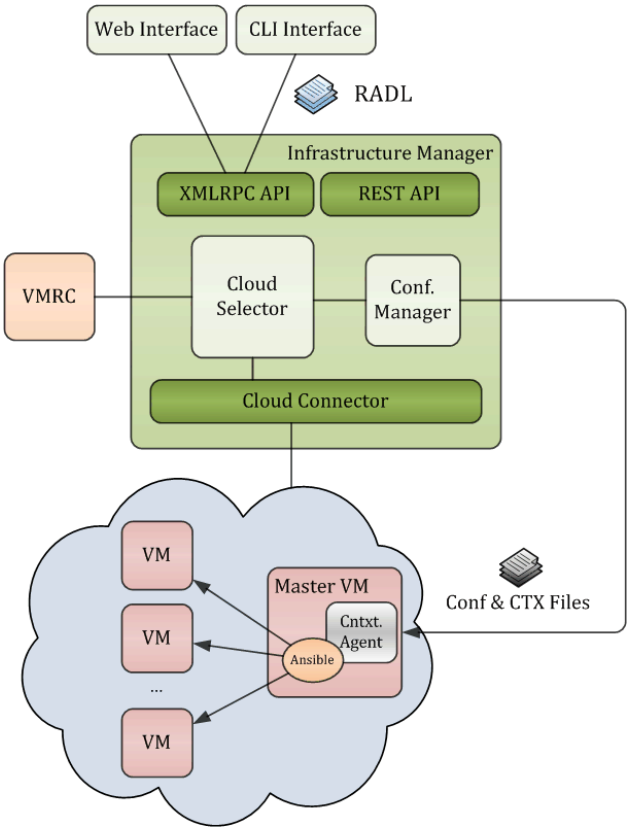

Figura 2.10: Arquitectura general de Infrastructure Manager

Fuente: [50]

En este trabajo se ha utilizado el IM como herramienta para el aprovisionamiento de máquinas virtuales y la configuración automatizada de aplicaciones en base a roles de Ansible. Esto posibilita desplegar la plataforma desarrollada en múltiples proveedores Cloud.

2.5.4. EC3

EC3 [51] [52] es una herramienta de código abierto que permite el despliegue de clústeres elásticos virtuales en la nube. Tal y como se muestra en la Figura 2.11, EC3 combina las herramientas anteriormente descritas, IM y CLUES, para posibilitar el despliegue de clústeres virtuales elásticos sobre múltiples proveedores Cloud.

EC3 despliega únicamente el nodo principal del clúster y lo configura con CLUES como gestor de elasticidad. Los usuarios pueden lanzar trabajos al sistema de colas y CLUES se encarga de aprovisionar nuevas máquinas virtuales bajo demanda, en función de los trabajos en cola, para adecuar el tamaño del cluster a las necesidades de cómputo.

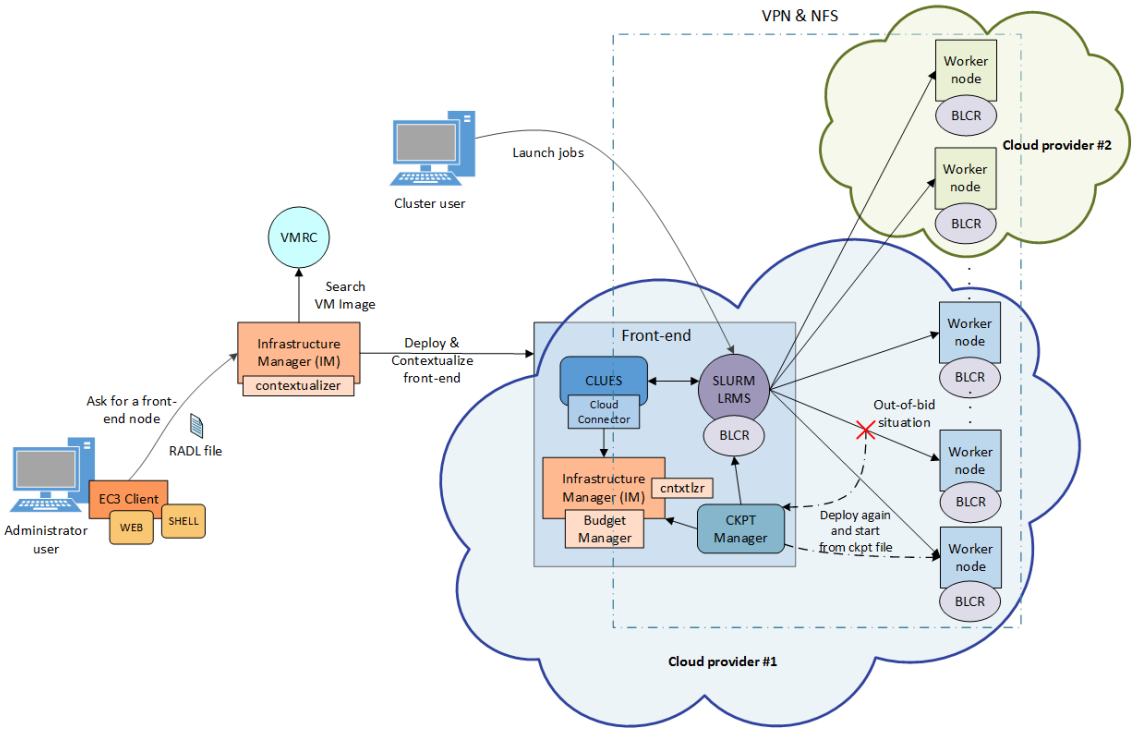

Figura 2.11: Arquitectura general de EC3

Fuente: [53]

En este trabajo se utiliza EC3 para desplegar, a través del IM, un clúster Kubernetes elástico que aloja todos los servicios de la plataforma desarrollada.

2.6 Otras tecnologías y herramientas

En esta sección se describen tecnologías y herramientas que han sido fundamentales para la realización de este trabajo.

2.6.1. Python

Python [54] es uno de los lenguajes de programación más utilizado en la actualidad. Es de alto nivel, orientado a objetos, interpretado y multiplataforma. Se caracteriza por la sencillez de su sintaxis, lo que hace que el código sea muy legible. Además, dada su popularidad cuenta con una gran cantidad de librerías para realizar todo tipo de funciones de forma sencilla.

En el proyecto se ha utilizado este lenguaje para desarrollar todos los componentes que han sido necesarios, así como para las funciones utilizadas en los casos de uso.

2.6.2. NATS

Nats [55] es un sistema de mensajería de código abierto simple, seguro y de alto rendimiento para aplicaciones en la nube, dispositivos IoT y arquitecturas de microservicios.

Es importante en este trabajo debido a que es la herramienta utilizada por OpenFaaS para gestionar los eventos de invocaciones asíncronas.

2.6.3. GitHub

Github [56] es una plataforma de desarrollo colaborativo a través de Internet que permite crear repositorios de *software* utilizando el sistema de control de versiones Git [57]. Los repositorios cuentan con una serie de funcionalidades adicionales que facilitan la colaboración, documentación y mantenimiento de los proyectos alojados, como por ejemplo el apartado *Issues* donde los usuarios pueden publicar los problemas que tengan usando el *software*.

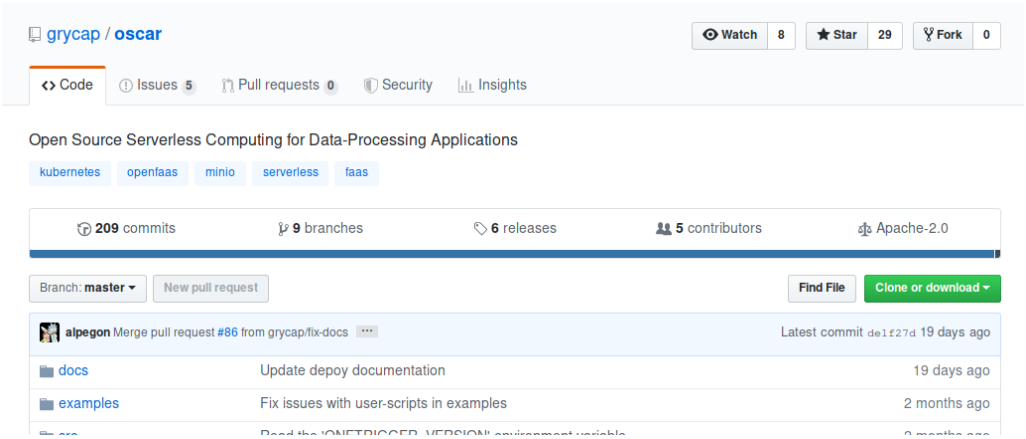

Figura 2.12: Repositorio del proyecto en GitHub

Es la plataforma utilizada para almacenar y gestionar todo el código escrito en el proyecto, además de para documentar el funcionamiento y los casos de uso de la plataforma. En particular, el principal repositorio de código donde se han realizado las contribuciones a la plataforma está disponible en [58].

CAPÍTULO 3
Desarrollo de la plataforma

En este capítulo de la memoria se detalla el proceso de desarrollo de la plataforma. En primer lugar se proporciona una visión general de su arquitectura. A continuación, se explican los pasos realizados para lograr la interconexión y automatización del despliegue de todos sus componentes. Para terminar, las tres últimas secciones profundizan en la definición de los componentes cuyo desarrollo ha sido necesario para que la plataforma funcione según los objetivos establecidos.

El nombre escogido para publicar el proyecto ha sido OSCAR [58], acrónimo de *On-premises Serverless Container-aware ARchitectures*, considerándose una versión de SCAR capaz de desplegarse sobre cualquier infraestructura.

3.1 Arquitectura

La plataforma OSCAR se ha desarrollado integrando una serie de herramientas de código abierto con la finalidad de soportar el modelo de programación para procesar ficheros basado en eventos. Estas herramientas son las detalladas a continuación:

- **Minio**. Es el sistema de almacenamiento de datos interno de la plataforma. Se caracteriza por tener una interfaz compatible con S3 y ser capaz de enviar eventos cuando se cargan o modifican ficheros.
- **OpenFaaS**. Es la plataforma de funciones como servicio utilizada por la plataforma para definir las funciones y el punto de acceso a las mismas.
- **NATS**. Se trata de un sistema de mensajería utilizado internamente por OpenFaaS para gestionar las invocaciones asíncronas a funciones. Su estudio ha sido necesario para solucionar el problema detallado en la Sección 3.4.
- **Docker Registry**. Es el registro de imágenes de contenedores *software* utilizado de forma interna en la plataforma para almacenar las imágenes de las funciones definidas.
- **Kaniko**. Es la herramienta mediante la cual se construyen las imágenes de las funciones en la plataforma. Su finalidad de añadir los binarios necesarios a las imágenes proporcionadas por los usuarios para asegurar el correcto funcionamiento de las funciones.

La Figura 3.1 resume la arquitectura principal del sistema. El usuario define una función indicando el sistema de almacenamiento de ficheros a utilizar (por defecto es MinIO) así como la aplicación a ejecutar, invocada mediante un *shell-script* definido por el usuario que se ejecuta sobre un contenedor a partir de una imagen también definida por el usuario y típicamente almacenada en Docker Hub. A partir de ese momento el usuario puede subir ficheros al sistema de almacenamiento. Esto provocará la creación de eventos que serán gestionados como *jobs* de Kubernetes para descargar el fichero, procesarlo y subir el resultado al sistema de almacenamiento. En caso de que sea necesario, CLUES solicitará al IM el despliegue de nuevos nodos adicionales sobre la plataforma Cloud donde se haya desplegado el cluster de Kubernetes, para poder aumentar la capacidad de procesamiento en paralelo de los ficheros.

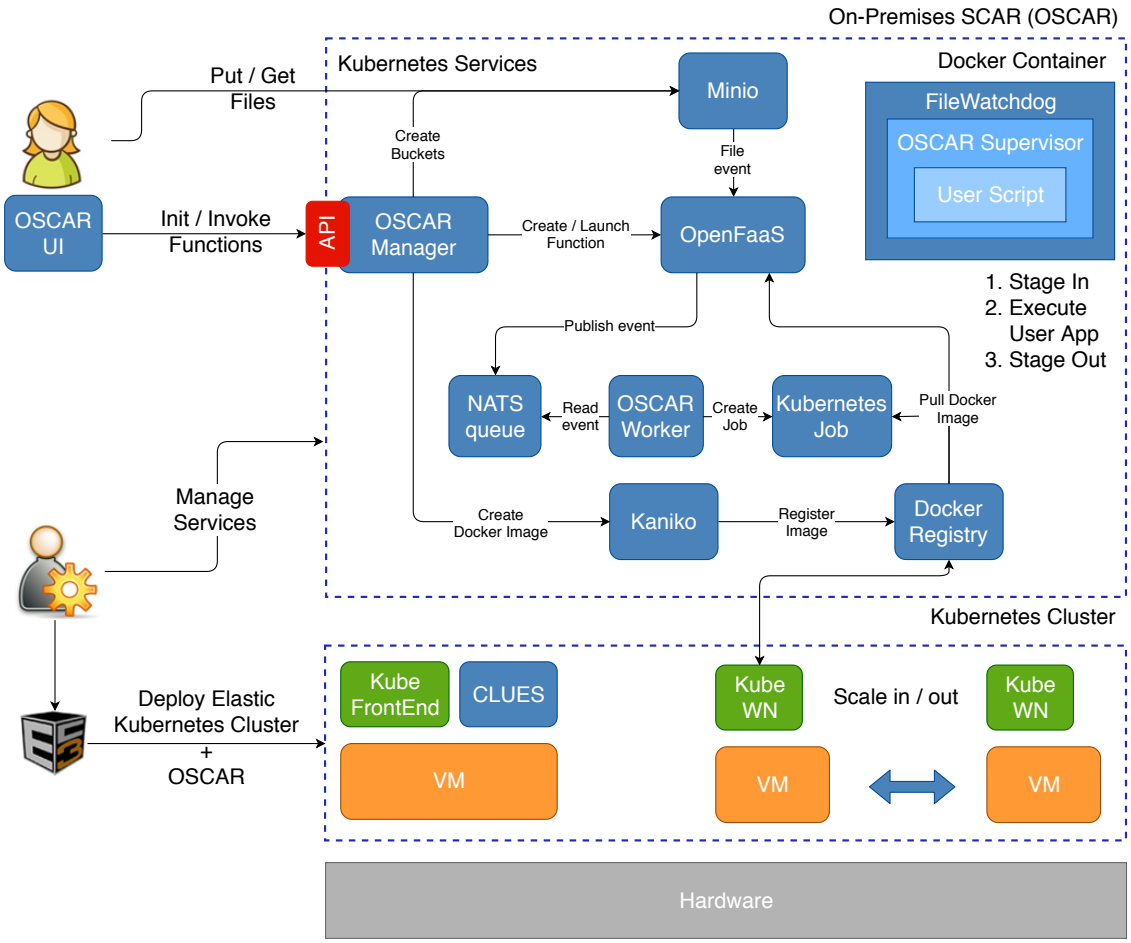

Figura 3.1: Arquitectura general de OSCAR

Fuente: [59]

Ha sido necesario implementar algunos componentes para que el comportamiento de la plataforma se corresponda con el establecido en los objetivos. Es importante mencionar que todos no han sido desarrollados como parte de este trabajo de fin de máster. A continuación se describen estos componentes y se indica el nivel de colaboración en los mismos:

- **Supervisor**. Es el componente encargado gestionar la entrada y salida de datos en las funciones para que estas puedan soportar el modelo de programación para procesar

de ficheros establecido. Su desarrollo, descrito en la Sección 3.3, ha formado parte de este trabajo.

- **OSCAR Manager**. Es un servicio desarrollado con el fin de gestionar las funciones y los recursos de almacenamiento de las mismas. Su principal objetivo es proporcionar una interfaz (API) mediante la cual se pueden crear nuevas funciones, siendo OSCAR Manager el encargado de construir la imagen de contenedor necesaria, así como los *buckets* o carpetas de entrada y salida en los sistemas de almacenamiento especificados. Su desarrollo no ha formado parte de este trabajo; sin embargo, el autor ha colaborado en el mismo.

- **OSCAR-UI**[1]. Se trata de una interfaz gráfica web que se comunica con la API de OSCAR Manager para facilitar el uso de esta herramienta y permitir que usuarios poco experimentados puedan utilizar la plataforma. Su desarrollo no ha formado parte de este trabajo.

- **OSCAR Worker**. Este componente se encarga de convertir las invocaciones asíncronas a funciones en *jobs* de Kubernetes, permitiendo así el escalado automático y asegurando que cada ejecución cuenta con los recursos definidos. Su desarrollo, descrito en la Sección 3.4, ha formado parte de este trabajo.

- **OneTrigger**. Es una herramienta desarrollada con la finalidad de generar eventos de forma automática cuando se cargan archivos en una carpeta o espacio de Onedata. Su implementación ha sido necesaria para poder cumplir con el objetivo establecido que consiste en la conexión de la plataforma con un sistema de almacenamiento externo. Su desarrollo, descrito en la Sección 3.5, ha formado parte de este trabajo.

Todos los componentes de la plataforma, tanto las herramientas como los desarrollos propios, se despliegan sobre un clúster Kubernetes elástico que puede ser aprovisionado sobre diferentes plataformas IaaS gracias a EC3.

3.2 Interconexión de los componentes y automatización del despliegue

Como se ha mencionado en la sección anterior, todas las herramientas de la plataforma han sido desplegadas sobre un clúster elástico de Kubernetes. El proceso para hacerlo no es muy complicado, ya que las propias herramientas proporcionan archivos con la definición de todas las directivas necesarias, mayoritariamente *deployments* y *services*.

Sin embargo, con el fin de soportar el modelo de programación de la plataforma, han sido necesarias algunas tareas de configuración para interconectar estos componentes. A continuación se exponen los detalles de configuración para realizar la interconexión y para su posterior despliegue automático:

- **OpenFaaS**. Para su despliegue únicamente ha sido necesario modificar el archivo de la directiva *service*, ya que por defecto el API Gateway no es accesible desde Internet.

[1]Publicado en: https://github.com/grycap/oscar-ui

La automatización del despliegue se ha llevado a cabo mediante la creación de un rol de Ansible llamado *ansible-role-kubefaas*[2]. En este rol, además, se ha incluido una opción para desplegar el componente OSCAR Worker.

- **MinIO**. Al tratarse de un sistema de almacenamiento de datos, MinIO requiere la creación de otra directiva de Kubernetes: Volúmenes Persistentes. En estos volúmenes se almacenarán los datos de los usuarios y, para poder crearlos en la plataforma, es necesario configurar una carpeta compartida mediante el protocolo NFS[3] en el nodo maestro del clúster. Por otra parte, para que los eventos generados al cargar ficheros sobre determinados *buckets* lleguen a la plataforma OpenFaaS se debe configurar el servidor indicando la ruta de acceso a las funciones.

  La automatización del despliegue se ha llevado a cabo mediante la creación de un rol de Ansible llamado *ansible-role-kubeminio*[4]. En este rol se ha añadido la opción de configurar el servidor para indicar diferentes rutas a las que se podrán enviar eventos.

- **Docker Registry**. El despliegue del registro de imágenes privado requiere varias configuraciones. En primer lugar, es necesario crear claves SSL[5] para poder utilizar un protocolo de acceso seguro. En segundo lugar, es necesario definir una dirección IP fija en la definición del *service*. Para terminar, la dirección IP fija se añade, junto con un nombre, al archivo de resolución de nombres de todos los nodos del clúster. De esta forma podrán resolverlo.

  La automatización del despliegue se ha llevado a cabo mediante la creación de un rol de Ansible llamado *ansible-role-kuberegistry*[6].

- **Kaniko**. Esta herramienta no necesita ser desplegada sobre la plataforma ya que no se trata de un servicio que permanezca activo constantemente. Como solo se utiliza para construir imágenes de contenedores cuando se crean nuevas funciones, se ha estudiado su uso para poder aplicarla a la directiva *jobs* de Kubernetes. Gracias a este estudio se ha podido automatizar el proceso de construcción de imágenes desde el componente OSCAR Manager.

3.2.1. Definición de la receta de despliegue

El despliegue de la plataforma completa se ha definido a través de una receta en el formato RADL utilizado por la herramienta EC3.

A partir de los roles de Ansible creados se ha procedido a la definición de la receta de despliegue de un cluster Kubernetes suplementada con los nuevos roles. Únicamente ha sido necesaria una ligera modificación para automatizar la creación de carpetas compartidas con la finalidad de soportar los Volúmenes Persistentes necesarios para el despliegue de MinIO.

[2]Publicado en: https://github.com/grycap/ansible-role-kubefaas
[3]https://web.mit.edu/rhel-doc/4/RH-DOCS/rhel-rg-es-4/ch-nfs.html
[4]Publicado en: https://github.com/grycap/ansible-role-kubeminio
[5]https://www.digicert.com/es/ssl.htm
[6]Publicado en: https://github.com/grycap/ansible-role-kuberegistry

En el Apéndice A se incluye la receta definida, mediante la cual se puede desplegar la plataforma desarrollada en cualquier proveedor Cloud, público o privado, soportado por EC3.

3.3 Gestión de Entrada/Salida

Para soportar el modelo de programación de procesamiento de ficheros basado en eventos es necesario disponer de un componente que gestione la entrada y salida de los datos. Este componente, al igual que en la herramienta SCAR, se denomina *supervisor*.

Al haberse implementado utilizando el lenguaje de programación Python, se han podido reutilizar ciertas funcionalidades del supervisor de SCAR. La más importante ha sido la conexión con el sistema de almacenamiento que, debido a interfaz compatible con S3 de MinIO, ha requerido mínimas modificaciones.

Sin embargo, al existir diferencias importantes entre el entorno de la plataforma desarrollada y el de AWS Lambda, se han tenido que realizar algunos cambios.

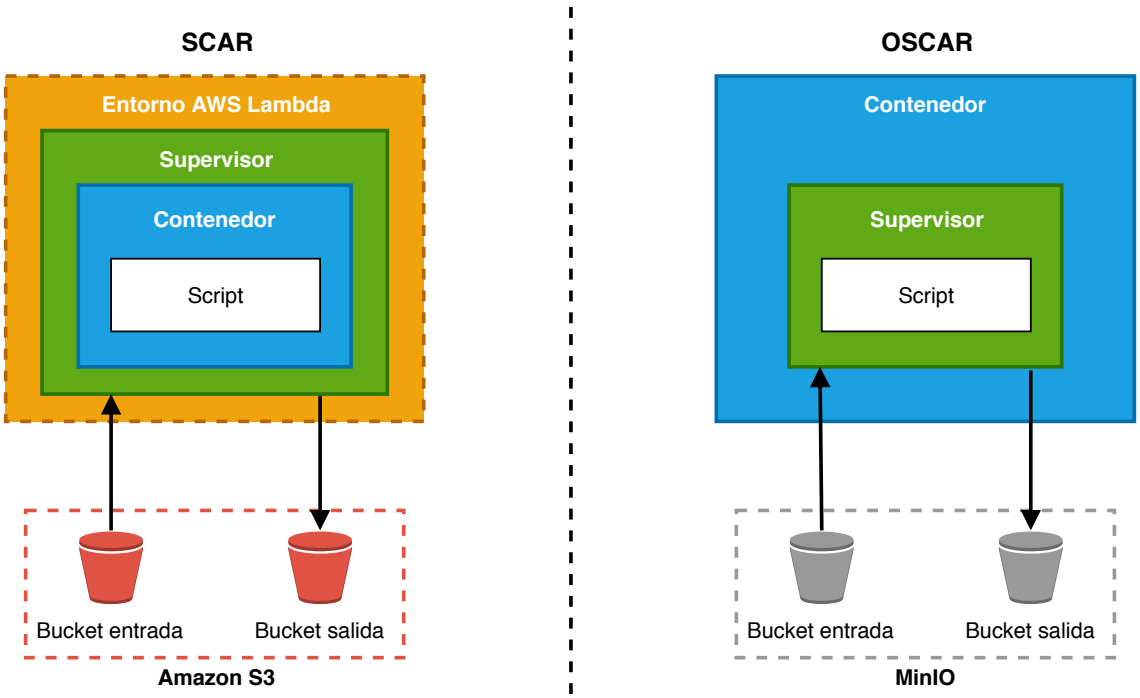

Figura 3.2: Diferencias entre los supervisores de SCAR y OSCAR

En primer lugar, una de las características más importantes del supervisor de SCAR consiste en la capacidad de arrancar contenedores Docker sobre el entorno restringido de AWS Lambda (que también es un contenedor). En el caso de la plataforma OSCAR esta funcionalidad deja de tener sentido. Esto se debe a que en la misma, gracias a OpenFaaS y al propio Kubernetes, se pueden desplegar contenedores sin ningún tipo de restricción.

En segundo lugar, en el entorno de AWS Lambda el supervisor se ejecuta como una función en el lenguaje de programación Python soportado por la plataforma. No obstante, en OSCAR no se debe obviar que el contenedor proporcionado por el usuario tenga la capacidad de ejecutar código Python. Por lo tanto, el nuevo supervisor se ha

empaquetado, junto a todas sus dependencias, como un archivo binario capaz de ejecutarse sobre cualquier contenedor.

A modo de resumen, este nuevo componente se encargará de realizar las siguientes tareas:

- Recibir el evento con el nombre y la ruta en el sistema de almacenamiento del fichero a procesar.
- Descargar el archivo desde el *bucket* de entrada y asignar su ruta a la variable de entorno *INPUT_FILE_PATH*.
- Crear una carpeta temporal en el contenedor para almacenar los ficheros resultantes de la ejecución y asignar su ruta a la variable de entorno *TMP_OUTPUT_DIR*.
- Ejecutar el *script* proporcionado por el usuario con los comandos para procesar el fichero. Para indicar la ruta del fichero de entrada, así como la ruta del directorio de salida, se deben utilizar las variables de entorno definidas para ello.
- Cargar el fichero o los ficheros resultantes al *bucket* de salida del sistema de almacenamiento.

3.4 Desarrollo de OSCAR Worker

En esta sección se detalla la problemática surgida tras las primeras pruebas de carga realizadas sobre la plataforma y la solución desarrollada, a la que se ha nombrado OSCAR Worker[7].

3.4.1. Motivación

Debido a las características propias de los sistemas orquestadores de contenedores y sus diferencias con las plataformas Serverless de proveedores Cloud públicos como AWS, los *frameworks* de funciones como servicios como OpenFaaS gestionan los recursos de forma diferente. En AWS Lambda cada invocación a una función reserva la cantidad de recursos establecida por el usuario. No obstante, OpenFaaS crea *deployments* en Kubernetes por cada función, que escalan en número de *pods* dependiendo de la cantidad de invocaciones recibidas en un periodo de tiempo. Esto significa que los recursos (CPU y memoria RAM) establecidos por los usuarios se gestionan a nivel de *pod*, que puede recibir múltiples peticiones de forma simultanea. Generalmente, este modelo de gestión de recursos cubre las necesidades de los usuarios, ya que suelen utilizar este tipo de plataformas para desarrollar microservicios que no requieren mucha capacidad de cómputo.

Tras las primeras pruebas de carga, que consistían en el procesamiento de múltiples ficheros de forma simultanea, se pudo observar que la plataforma no se comportaba correctamente. El problema estaba causado por la gestión de recursos descrita anteriormente, incompatible con los objetivos propuestos.

Si se quiere explotar una plataforma FaaS para realizar procesamientos de mayor duración y que consuman más recursos es necesario asegurar que cada invocación esté

[7]Publicado en: https://github.com/grycap/oscar-worker

aislada en un contenedor (o *pod*). Para ello, OpenFaaS dispone de un método de invocación a funciones asíncrono, que además evita que el cliente que realiza la petición quede en espera hasta que esta se procese.

OpenFaaS utiliza el sistema de mensajería NATS para almacenar las invocaciones asíncronas a funciones en una cola. Las invocaciones de este tipo se van procesando de una en una gracias a un componente interno llamado *nats-queue-worker*. Este componente se encarga de obtener los eventos de la cola y enviarlos hacia el servicio de la función. Una vez procesada la invocación, se elimina el evento de la cola y se procesa el siguiente. En la Figura 3.3 se muestra la secuencia seguida para procesar una invocación asíncrona.

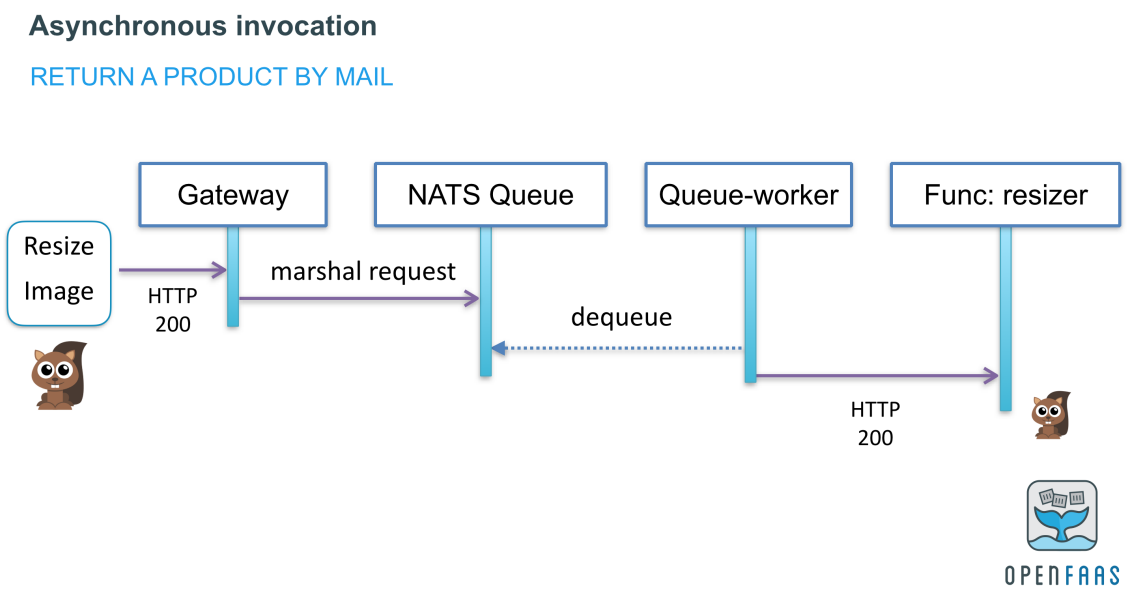

Figura 3.3: Diagrama de secuencia de invocaciones asíncronas en OpenFaaS

Fuente: [60]

El problema de esta solución es que la plataforma no explota todos los recursos disponibles en el clúster, ya que las invocaciones a una misma función nunca se procesan en paralelo. Además, no se aprovechan las capacidades de escalado automático proporcionado por CLUES, ya que de esta forma no se aumenta el número de *pods* asignados a la función.

Debido a estos motivos se ha considerado necesario implementar un nuevo componente que asegure que cada invocación se procese en un entorno aislado.

3.4.2. Implementación

La solución planteada consiste en la creación de un componente que sustituya al *nats-queue-worker* de OpenFaaS y se encargue de transformar las invocaciones a las funciones en *jobs*. Estas directivas se ajustan completamente con los objetivos de la plataforma. Cada *job* cuenta con un entorno aislado que solicita los recursos necesarios al planificador de Kubernetes, lo que permite que el sistema CLUES gestione de forma adecuada la elasticidad del clúster.

Para desarrollar el componente, llamado OSCAR Worker, se ha utilizado el lenguaje de programación Python. En primer lugar, ha sido necesario un estudio del funcionamiento

del componente interno de OpenFaaS *nats-queue-worker*. Tras lograr entender el formato en el que se almacenan los eventos en la cola NATS y el modo de conexión con la misma, se ha realizado la implementación.

Entrando en detalle, la herramienta está compuesta por dos módulos que realizan dos tareas específicas. El primero es el encargado de suscribirse a la cola donde se guardan los eventos generados por invocaciones asíncronas y de reaccionar cuando se recibe alguno. El segundo es un cliente simple de Kubernetes que se encarga de registrar los nuevos *jobs* en el sistema. En la Figura 3.4 se muestra un resumen del funcionamiento de OSCAR Worker.

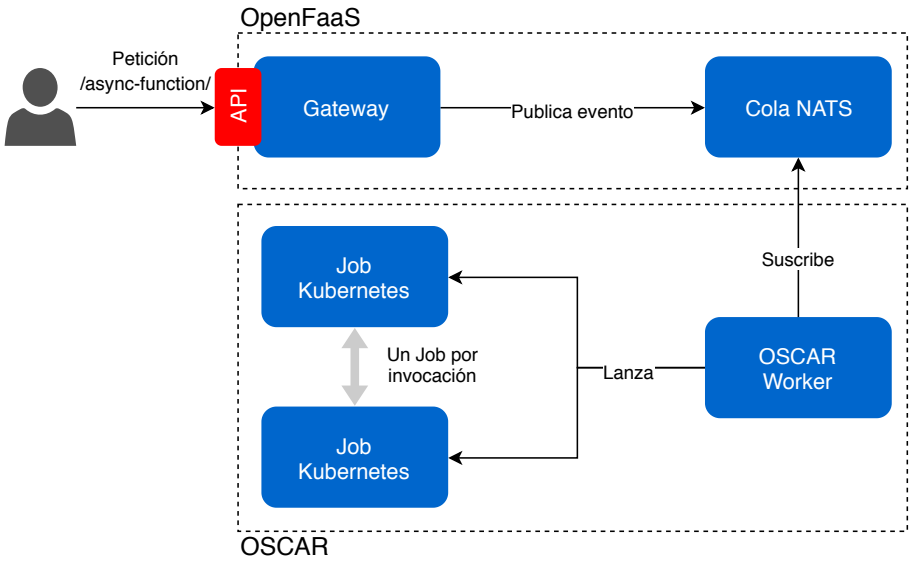

Figura 3.4: Resumen del funcionamiento de OSCAR Worker

Mediante esta aproximación se han aprovechado las características de sistema de colas ofrecido por Kubernetes para poder ejecutar múltiples *jobs* encargados del procesamiento en paralelo de los ficheros.

3.5 Integración con Onedata

En esta sección se exponen los detalles de integración de la plataforma con el sistema de almacenamiento Onedata [39]. Para lograr el objetivo planteado ha sido necesario el desarrollo de un nuevo componente y una pequeña adaptación del supervisor.

3.5.1. Motivación

Tal y como se indica en los objetivos del trabajo, una de las características deseadas para la plataforma es la capacidad de procesar archivos en entornos híbridos. Para ello es necesario integrarla con un sistema de almacenamiento de datos externo. El sistema escogido ha sido Onedata debido a su enfoque hacia usuarios científicos y a su utilización en otros proyectos del grupo de Grid y Computación de Altas Prestaciones.

El soporte a un espacio para poder realizar pruebas ha sido proporcionado por EGI DataHub [61]. EGI DataHub es una implementación de Onedata sobre la infraestructura EGI (European Grid Infrastructure) [62], con la que también colabora el grupo.

Tras un estudio del funcionamiento del sistema Onedata (en concreto del servidor Oneprovider) se han identificado los métodos de su API necesarios para la carga y descarga de ficheros. Sin embargo, Onedata carece de un sistema para generar eventos ante la subida de archivos al espacio del usuario, por lo que resulta necesario desarrollar alguna solución que se encargue de ello. Por otra parte, su API también permite consultar el listado de archivos dentro de un determinado espacio o una carpeta en su interior.

3.5.2. Implementación de OneTrigger

La solución planteada para la generación automática de eventos cuando se interactúa con en el sistema de almacenamiento Onedata se ha nombrado OneTrigger[8]. OneTrigger ha sido desarrollada como una aplicación de línea de comandos para facilitar su comprobación. Sin embargo, tambien se ha pensado en la integración con Kubernetes, por lo que se han añadido métodos para poder configurarla a través de variables de entorno y se ha subido como imagen de contenedor en Docker Hub.

Su funcionamiento consiste en la comprobación periódica de los archivos almacenados en un espacio o una carpeta de Onedata. La aplicación guarda un listado de los ficheros y, cuando comprueba que existe uno nuevo, genera el evento y lo envía a un punto de entrada HTTP (conocido como *Webhook*) especificado en su configuración. Además, la aplicación se ha implementado de forma que gestione automáticamente errores de conexión y tenga en cuenta los archivos que se eliminen del sistema. En la Figura 3.5 se resume su funcionamiento.

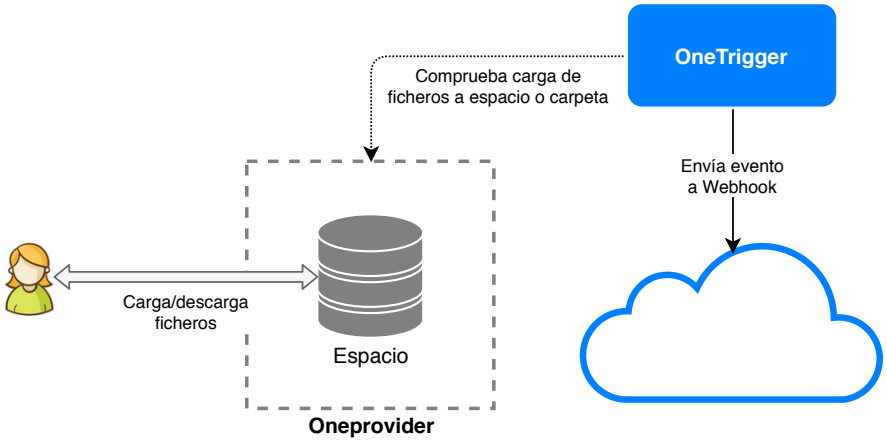

Figura 3.5: Resumen del funcionamiento de OneTrigger

OSCAR Manager ha sido modificado para crear automáticamente un *deployment* de OneTrigger por cada función que se despliegue en la plataforma con soporte a Onedata. Por otra parte, se ha definido un formato estándar para los eventos generados. Así el supervisor podrá diferenciar fácilmente el sistema proveedor de datos que lo ha generado y utilizar los métodos necesarios para la carga y descarga. En la Figura 3.6 se puede observar el formato establecido para los eventos.

[8]Publicado en: https://github.com/grycap/onetrigger

```
{
    "Key": "/my-onedata-space/files/file.txt",
    "Records": [
        {
            "objectKey": "file.txt",
            "objectId": "0000034500046EE9C67756964233837",
            "eventTime": "2019-02-07T09:51:04.347823",
            "eventSource": "OneTrigger"
        }
    ]
}
```

Figura 3.6: Ejemplo del formato de un evento de OneTrigger

3.5.3. Adaptación del supervisor

La adaptación del supervisor con Onedata se ha realizado añadiendo métodos para la carga y descarga de ficheros a un determinado espacio. Estos métodos se han implementado utilizando la API de Oneprovider, que sigue la especificación CDMI (*Cloud Data Management Interface*) [63]. Esta especificación define un estándar para gestionar archivos en la nube, por lo que el procedimiento de adaptación ha sido sencillo.

Para que el supervisor diferencie entre invocaciones de MinIO y de OneTrigger, simplemente se ha añadido una comprobación de la variable "eventSource" de los eventos.

En la Figura 3.7 se muestra un resumen de todos los componentes que se encargan de integrar Onedata con la plataforma OSCAR.

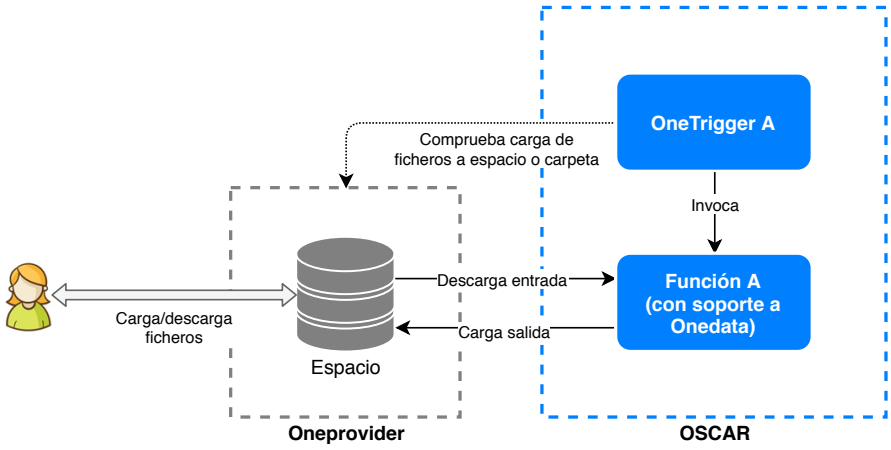

Figura 3.7: Resumen de la integración de OSCAR con Onedata

Esta solución permite utilizar EGI DataHub como una solución de persistencia de ficheros a largo plazo. Por otra parte, se puede utilizar una plataforma Cloud federada europea como EGI Federated Cloud para desplegar de forma dinámica un clúster Kubernetes con OSCAR y así poder realizar el procesado de nuevos ficheros dirigido por eventos sobre esta infraestructura.

CAPÍTULO 4
Casos de uso

En este capítulo se presentan dos casos de uso para demostrar la utilidad del proyecto. Ambos han sido desarrollados a partir de aplicaciones o datos abiertos, tratando de aportar valor y generando información gracias a la capacidad de procesamiento de ficheros de la plataforma Serverless creada.

Estos casos de uso han sido probados sobre la infraestructura física del clúster "Ramses", perteneciente al grupo de Grid y Computación de Altas Prestaciones en el Instituto de Instrumentación para Imagen Molecular de la Universitat Politècnica de València. El clúster, gestionado por OpenNebula, dispone de una red de área de almacenamiento Dell Equallogic PS4210 con 16TB disponibles. La infraestructura física está constituida por dos tipos de nodos. El primero tiene 240GB de disco de estado sólido, 64GB de memoria RAM, dos procesadores Intel(R) Xeon(R) CPU E5-2683 v3 2.00GHz con 14 núcleos cada uno, dos adaptadores de red Ethernet de 1GB y otro de 10GB. El segundo tipo de nodo tiene 250GB de disco de estado sólido, 128GB de memoria RAM, dos procesadores Intel(R) Xeon(R) CPU E5-2660 v4 2.00GHz con 14 núcleos cada uno y tres adaptadores de red Ethernet, dos de 1GB y un tercero de 10GB.

En todos los casos de uso se despliega un cluster Kubernetes formado por máquinas virtuales ejecutándose sobre la plataforma Cloud privada anteriormente definida.

4.1 Clasificación de plantas

El primer caso de uso está basado en el estudio de Ignacio Heredia [64], parte del proyecto europeo DEEP Hybrid DataCloud[1], cuyo código se encuentra disponible de forma pública[2]. El estudio describe el proceso realizado para entrenar una red neuronal convolucional[3] con la finalidad de obtener un clasificador de miles de variedades de plantas a partir de imágenes.

Como se ha mencionado, la aplicación resultante del estudio se puede encontrar en un repositorio de *software* público. Esta aplicación ha sido desarrollada como una herramienta web en la que los usuarios deben subir la fotografía de una planta para, posteriormente, poder visualizar el resultado de la clasificación en el navegador.

[1] https://deep-hybrid-datacloud.eu

[2] https://github.com/deephdc/plant-classification-theano

[3] https://www.ibm.com/developerworks/ssa/library/cc-convolutional-neural-network-vision-recognition/index.html

A pesar de no estar desarrollada como una aplicación de procesamiento de ficheros directamente compatible con la plataforma, se ha considerado que, tras una ligera adaptación, podría tratarse de un buen ejemplo de utilización de la misma.

4.1.1. Adaptación del código

Para reutilizar la aplicación ha sido necesario un proceso de adaptación. Este proceso se ha realizado en tres pasos:

- Definición de la aplicación simplificada como una herramienta para procesar ficheros desde la línea de comandos. Tras estudiar el código de la aplicación se ha decidido crear un pequeño programa en el lenguaje de programación Python. Este programa recibe dos argumentos desde la linea de comandos: el fichero de entrada y el fichero de salida. Como la aplicación original está escrita en el mismo lenguaje, se han podido reutilizar los métodos encargados de cargar el modelo entrenado y clasificar la imagen. El proceso de mostrar el resultado y guardarlo como una nueva imagen ha sido implementado sin mucha dificultad gracias al conocimiento previo y la experiencia en el uso de Python.
- Creación de una imagen de contenedor de *software*. Para ello se ha definido un archivo Dockerfile encargado de incluir la aplicación original, así como todas las librerías necesarias para el correcto funcionamiento de la misma. En la imagen, además, se ha añadido el programa creado en el punto anterior. Una vez construida la imagen del contenedor se ha subido al registro público Docker Hub.
- Definición de un *script* con los comandos necesarios para procesar ficheros desde la plataforma. En este *script* se deben utilizar las variables de entorno definidas por el supervisor para indicar las rutas de las carpetas de entrada y salida.

```
#!/bin/bash

echo "SCRIPT: Invoked classify_image.py. File available in $INPUT_FILE_PATH"
FILE_NAME=`basename "$INPUT_FILE_PATH"`
OUTPUT_FILE="$TMP_OUTPUT_DIR/$FILE_NAME"
python2 /opt/plant-classification-theano/classify_image.py "$INPUT_FILE_PATH" -o "$OUTPUT_FILE"
```

Figura 4.1: *Script* con los comandos necesarios para clasificar una imagen

4.1.2. Despliegue

El despliegue de la función se ha realizado a través de la interfaz web OSCAR-UI. Para ello, el primer paso ha sido acceder desde un navegador web a la dirección IP del nodo maestro del clúster Kubernetes. A continuación, desde la ventana de creación de funciones, se debe indicar el nombre de la función, la imágen del contenedor de *software* a utilizar y el *script* creado anteriormente con los comandos encargados de procesar ficheros.

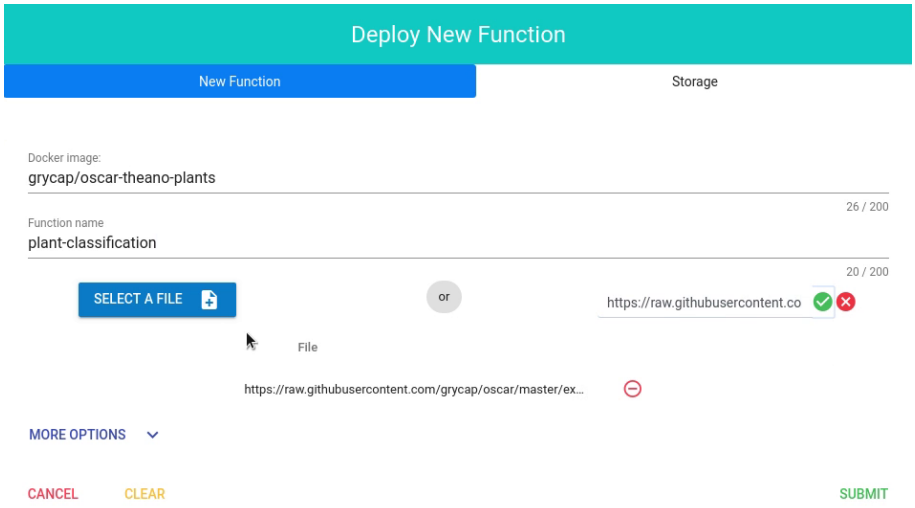

Figura 4.2: Despliegue de la función *plant-classification* desde la interfaz web de OSCAR

A continuación, es recomendable establecer los recursos asociados a la función. Para ello simplemente se debe hacer clic en el menú desplegable "MORE OPTIONS" e indicar la cantidad de CPU y memoria.

Figura 4.3: Opciones avanzadas de una función en la interfaz web de OSCAR

Finalmente, se pulsa sobre el botón "SUBMIT" para desplegar la función, que aparecerá en la pantalla principal de la interfaz web tras unos segundos.

4.1.3. Resultados

Para comprobar el correcto funcionamiento de la función desplegada y el comportamiento de la plataforma, especialmente en términos de paralelismo y elasticidad, se ha decidido realizar una prueba de carga. Esta prueba consiste en el procesamiento de una gran cantidad de imágenes al mismo tiempo. Para ello, simplemente se deben subir las imágenes al *bucket* de entrada creado automáticamente en MinIO, accesible desde su propia interfaz web o desde OSCAR-UI.

La plataforma creará un trabajo por cada imagen subida y lo registrará en el planificador del orquestador Kubernetes. Estos trabajos se procesan de forma simultanea en función de los recursos especificados en la función y los disponibles en el clúster.

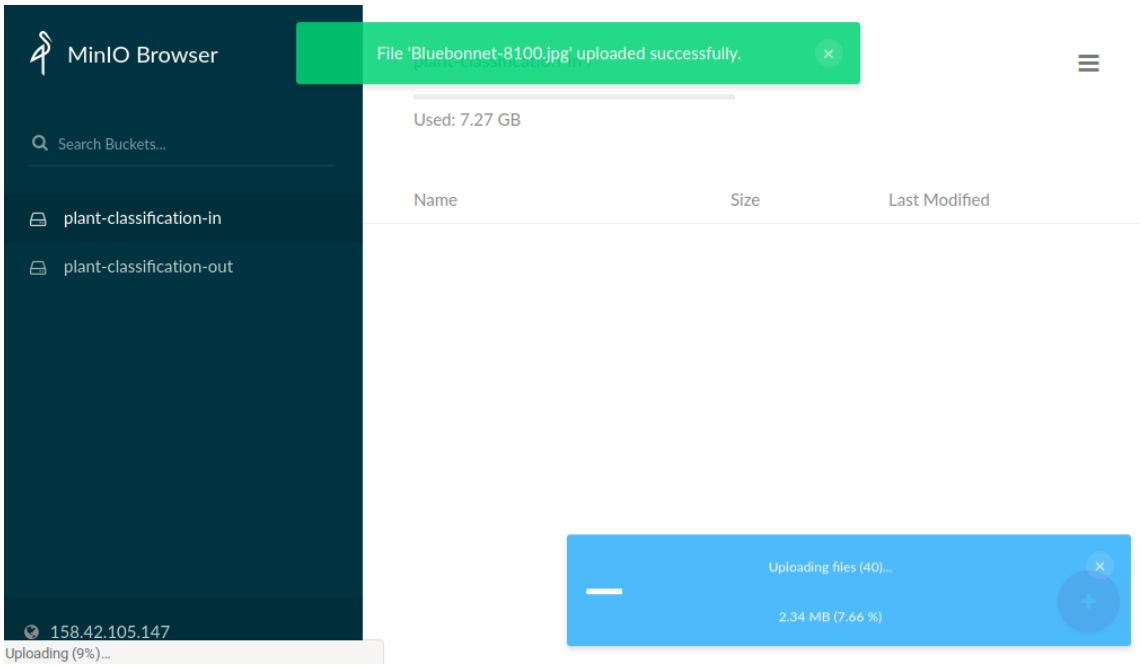

Figura 4.4: Carga de imagenes en el *bucket* de MinIO *plant-classification-in*

Además, gracias al uso de CLUES, la plataforma crecerá o decrecerá en número de nodos en función de la cantidad de trabajos definidos. En la Figura 4.5 se puede observar como cuatro nodos se encuentran en estado “powon”, es decir, en proceso de encendido. Una vez arrancados y configurados serán capaces de procesar trabajos, aumentando así el nivel de paralelismo de la plataforma.

```
ubuntu@kubeserver:~$ clues status
node                    state    enabled   time stable   (cpu,mem) used   (cpu,mem) total
-------------------------------------------------------------------------------------------
wn1.localdomain          used    enabled     20h24'19"    3.3,3810525184      4,4038340608
wn2.localdomain         powon    enabled     00h00'52"      0,0              4,4294967296
wn3.localdomain         powon    enabled     00h00'12"      0,0              4,4294967296
wn4.localdomain         powon    enabled     00h00'11"      0,0              4,4294967296
wn5.localdomain         powon    enabled     00h00'10"      0,0              4,4294967296
```

Figura 4.5: Consulta del estado de los nodos del clúster desde la herramienta CLUES

Cuando todos los trabajos han sido procesados, los nodos que ya no son necesarios se apagan de forma automática. Esto permite disminuir drásticamente el coste de la infraes-

tructura cuando no se esté utilizando. Todo este proceso se realiza de forma totalmente transparente al usuario, tal y como dicta el paradigma Serverless.

Finalmente, para obtener los resultados de la clasificación hay que acceder al *bucket* de salida desde la interfaz web de MinIO o la de OSCAR. Desde ambas interfaces se podrán descargar las imágenes resultantes, que serán similares a la mostrada en la figura 4.6. La imagen queda etiquetada con aquellos tipos de plantas que más se asemejan, de acuerdo al modelo entrenado, a la planta de la fotografía.

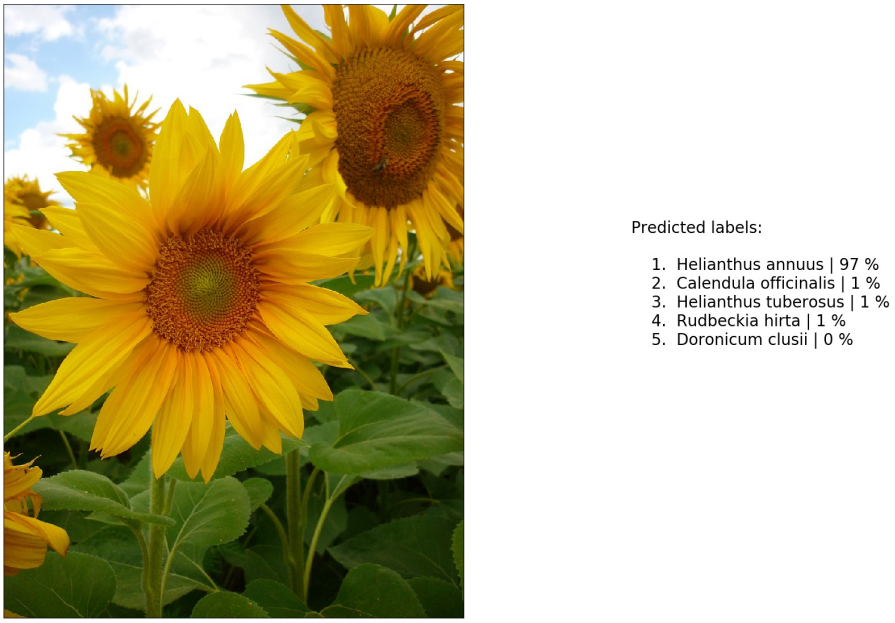

Figura 4.6: Resultado de la clasificación automática de una planta

Es importante destacar que el uso de la plataforma OSCAR ofrece al usuario una plataforma de cómputo simplificada en la que este únicamente debe subir los ficheros al sistema de almacenamiento para desencadenar el procesado de los mismos sobre una infraestructura de cómputo distribuida. Esta capacidad de simplificar el acceso a potencia computacional es la principal característica de la plataforma desarrollada.

4.2 Información diaria sobre contaminación atmosférica

El segundo caso de uso ha sido desarrollado desde cero con el propósito de demostrar la capacidad de la plataforma para generar valor. El principal objetivo es mostrar que el modelo de programación para procesar ficheros de la misma puede ser explotado para producir información a partir de fuentes de datos abiertos.

La fuente de datos escogida ha sido la página web del proyecto del Índice de la Calidad del Aire Mundial [65]. Este proyecto recoge periódicamente información sobre la contaminación atmosférica de más de 10.000 estaciones repartidas por todo el mundo. Desde su página web se pueden visualizar los datos actualizados de todas las estaciones sobre un mapa global. No obstante, este proyecto también proporciona herramientas para acceder a sus datos de forma sencilla y así poder desarrollar aplicaciones que los utilicen.

Así pues, el planteamiento del caso de uso consiste en la generación periódica de informes gráficos que muestren los valores de contaminación de una estación concreta. La idea es que los usuarios puedan tener un informe diario sobre la calidad del aire para, por ejemplo, publicarlo en su sitio web o simplemente consultarlo y así mantenerse informados. Todo ello de forma automática sin que tengan que realizar ninguna acción adicional.

Para ello se ha creado una función simple en OpenFaaS que obtiene los datos actualizados de una estación y los carga como un fichero dentro de la carpeta o *bucket* de entrada de la función principal, encargada de generar los informes. La invocación periódica de la función de descarga se programará con alguna herramienta adicional. Además, para demostrar el soporte a flujos de trabajo híbridos, se utilizará Onedata como sistema de almacenamiento de datos externo. En la Figura 4.7 puede observarse un esquema del funcionamiento general del caso de uso.

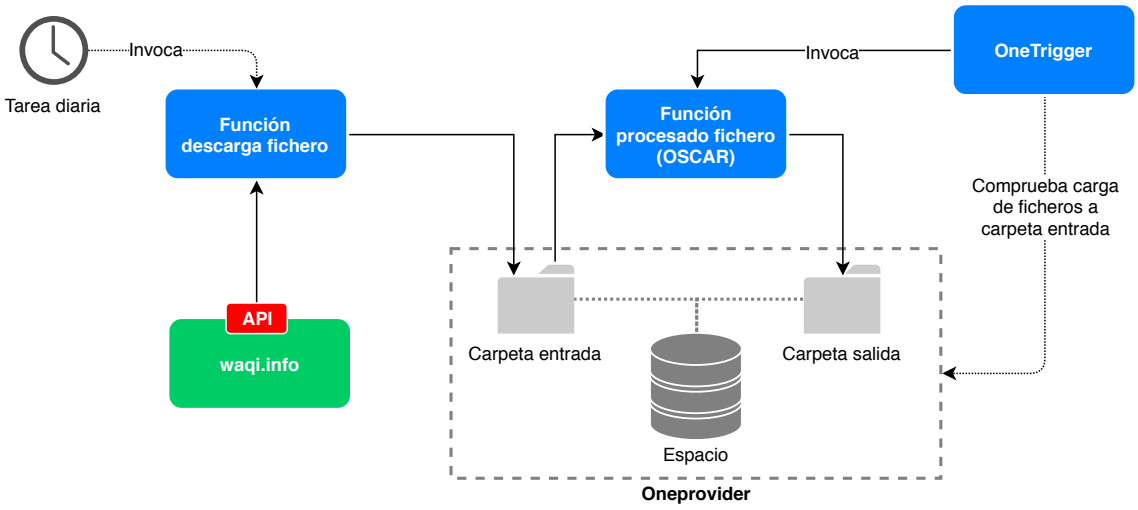

Figura 4.7: Resumen del funcionamiento del segundo caso de uso

4.2.1. Desarrollo de la función de descarga

Debido a que esta función no requiere el modelo de programación para procesar ficheros definido en las funciones de OSCAR, se ha decidido implementarla directamente sobre OpenFaaS. El nombre escogido para la misma ha sido *waqi2onedata*[4].

Para obtener los datos actualizados de una estación se ha utilizado la API pública del proyecto del Índice de la Calidad del Aire Mundial [66]. Esta API proporciona una interfaz para obtener datos en formato JSON[5] a través de peticiones web. La función simplemente realiza una petición con el identificador de la estación deseada. Es importante mencionar que, a pesar de tratarse de datos abiertos, para poder utilizar la API del proyecto del Índice de la Calidad del Aire Mundial es necesario solicitar un *token* de acceso de forma gratuita.

El acceso al espacio de Onedata para almacenar el fichero obtenido se ha implementado reutilizando algunos métodos definidos en el supervisor de OSCAR. En el momento del despliegue el usuario deberá indicar la información de acceso a su espacio, así como el nombre de la carpeta donde se cargarán los archivos.

[4]Publicado en: https://github.com/srisco/waqi2onedata
[5]https://www.json.org/json-es.html

4.2.2. Temporización diaria

Para programar una invocación diaria a la función *waqi2onedata* se ha utilizado el gestor del sistema y servicios *systemd* [67], presente en la mayoría de sistemas operativos GNU/Linux actuales. En concreto se ha usado la funcionalidad *timers* [68], que permite programar la ejecución de cualquier servicio definido por el usuario.

El proceso es sencillo: primero se crea un servicio con el comando a ejecutar, en este caso una invocación vía HTTP a la función usando la herramienta *curl*[6]; y luego se define el temporizador, en el que se especifica cada cuanto tiempo se va a ejecutar el servicio. Finalmente se activa el *timer* tal y como se muestra en la Figura 4.8. Este proceso se ha realizado en el nodo maestro del clúster Kubernetes de la plataforma.

```
ubuntu@kubeserver:~$ systemctl status aqi-upv.timer --user
● aqi-upv.timer - Ejecuta el servicio aqi-upv diariamente
   Loaded: loaded (/home/ubuntu/.config/systemd/user/aqi-upv.timer; enabled; vendor preset: enabled)
   Active: active (waiting) since vie 2019-07-05 12:57:11 CEST; 13s ago

jul 05 12:57:11 kubeserver.localdomain systemd[8936]: Started Ejecuta el servicio aqi-upv diariamente.
```

Figura 4.8: Estado del temporizador *aqi-upv.timer*

4.2.3. Desarrollo de la función de procesado

Al tratarse de una función que explota el modelo de procesamiento de ficheros de OSCAR, el desarrollo de esta pequeña aplicación, llamada *waqi-station-report*[7], se ha complementado con la creación de una imagen de contenedor de *software* y un *script* con los comandos necesarios para procesar los datos. El procedimiento seguido ha sido el siguiente:

- Desarrollo de la aplicación. Utilizando el lenguaje de programación Python, se ha implementado un pequeño programa que lee y valida el archivo JSON que contiene los datos de la estación. Una vez validado, haciendo uso del módulo *matplotlib* [69] para generación de gráficos, crea un informe gráfico con la información de la contaminación atmosférica y lo guarda como una imagen. Este informe trata de mostrar los datos de una forma más visual y, además, añade una breve descripción sobre el nivel de calidad del aire para que los usuarios puedan tomar medidas si lo consideran necesario. En la Figura 4.13 se puede observar el formato de estos informes gráficos.
- Creación de la imagen del contenedor con la aplicación y todas las librerías necesarias a partir de un archivo Dockerfile. Posteriormente la imagen construida se ha publicado en Docker Hub.
- Definición del *script* igual que en el caso de uso anterior (Figura 4.1), pero indicando el comando de la aplicación *waqi-station-report*.

[6] https://curl.haxx.se/
[7] Publicado en: https://github.com/srisco/waqi-station-report

4.2.4. Despliegue

En este caso de uso el proceso se puede dividir claramente en dos partes: el despliegue de la primera función a través del cliente de línea de comandos de OpenFaaS y el despliegue de la segunda función a través de la interfaz web de OSCAR.

Para desplegar la primera función se debe crear un fichero YAML con la información requerida por OpenFaaS[8] para la definición de funciones. En este fichero se debe indicar el nombre que se le va a asignar a la función, así como el código de la misma o, si ya ha sido creada, la imagen del contenedor de *software* que lo incluye. Un punto muy importante para que funcione correctamente es añadir las variables de entorno con la información de acceso al espacio de Onedata, así como el token necesario para utilizar la API del proyecto del Índice de la Calidad del Aire Mundial. Finalmente, cuando el archivo está definido, la función se puede desplegar ejecutando un comando, tal y como se muestra en la Figura 4.9.

```
ubuntu@kubeserver:~$ sudo faas deploy -f waqi2onedata-function.yaml
Deploying: waqi2onedata.
WARNING! Communication is not secure, please consider using HTTPS. Letsencrypt.org offers free SSL/TLS certificates.

Deployed. 202 Accepted.
URL: http://127.0.0.1:31112/function/waqi2onedata
```

Figura 4.9: Despliegue de la función *waqi2onedata* desde el cliente de OpenFaaS

El proceso de despliegue de la función encargada de procesar los datos es similar al del caso de uso anterior. Se debe acceder a la interfaz web de OSCAR a través de un navegador y, en la ventana de creación de funciones, se debe indicar el nombre, la imagen del contenedor y el *script*.

La única diferencia respecto al primer caso de uso consiste en añadir soporte al sistema de almacenamiento externo Onedata. Para ello se deben introducir los datos de acceso al mismo en el menú accesible a través de la pestaña "Storage" de la interfaz como se muestra en la Figura 4.10. Finalmente se crea la función pulsando sobre el botón "SUBMIT".

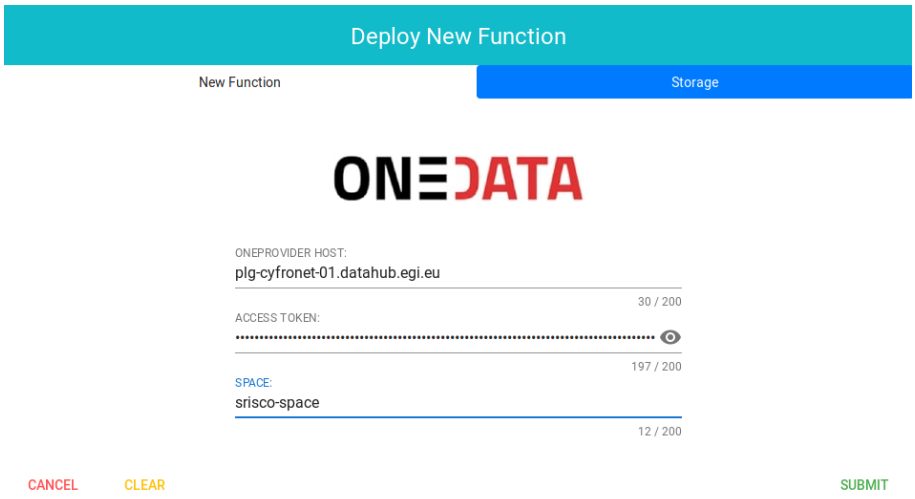

Figura 4.10: Configuración de acceso a Onedata desde la interfaz web de OSCAR

[8] https://docs.openfaas.com/reference/yaml/

4.2.5. Resultados

Para comprobar que el caso de uso funciona correctamente se debe acceder al espacio de Onedata. Este proceso puede hacerse utilizando el cliente Oneclient o a través de la interfaz web accesible desde la dirección del Oneprovider que soporte al espacio. Para no complicar esta sección se ha optado por la interfaz web, en la cual se deben introducir las credenciales de usuario.

Dentro del espacio, se puede observar como se han creado las carpetas *process-waqi-in* y *process-waqi-out*. En la primera se cargan los archivos JSON con los datos de la estación, lo que desencadena la invocación a la función de procesado.

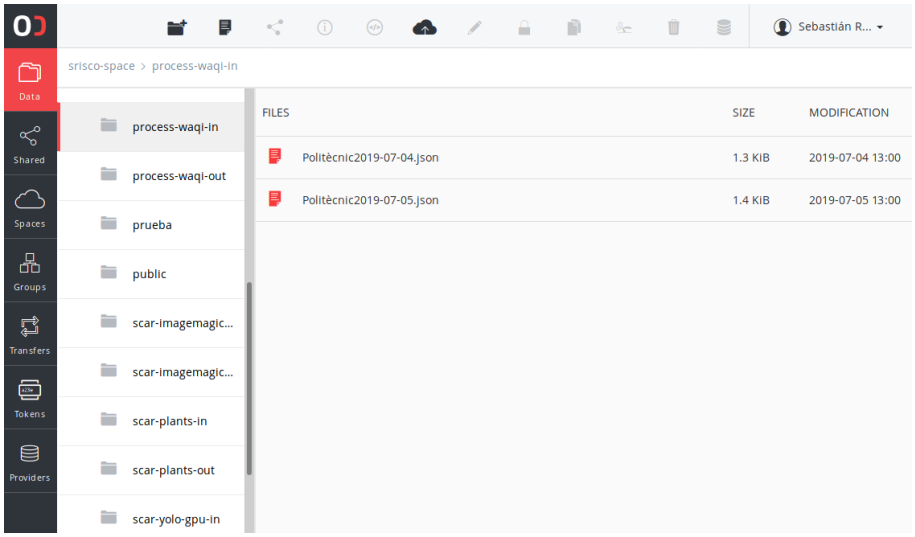

Figura 4.11: Carpeta de entrada *process-waqi-in* en la interfaz web de Onedata

Cuando la segunda función termina, sube automáticamente el resultado a la carpeta de salida.

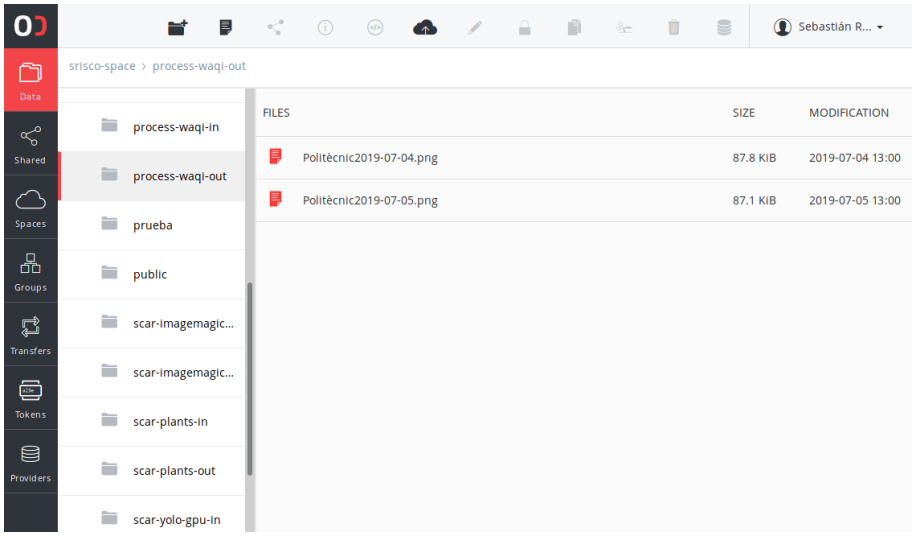

Figura 4.12: Carpeta de salida *process-waqi-out* en la interfaz web de Onedata

Desde la interfaz web de Onedata se pueden descargar los informes gráficos, que aparecerán cada día en la carpeta de salida, en este caso *process-waqi-out*.

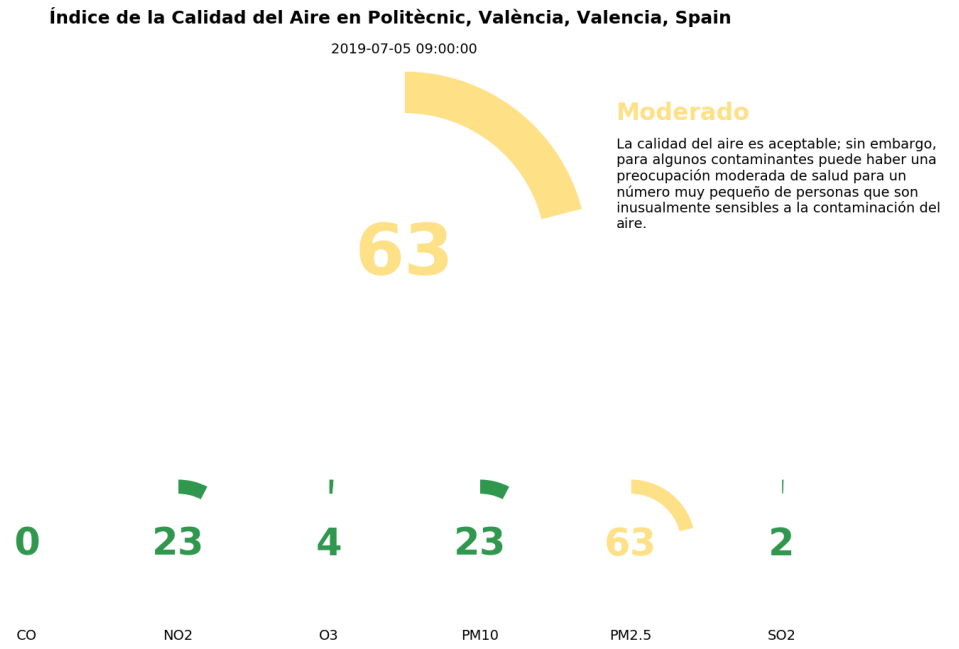

Figura 4.13: Informe gráfico del Índice de la Calidad del Aire

De nuevo, la versatilidad de la plataforma desarrollada facilita el uso de computación dirigida por eventos para aplicaciones generales. En este caso de uso, se posibilita la obtención de informes gráficos a partir de datos abiertos que se publican de forma periódica, lo que puede servir para mantener actualizada una web con diferentes gráficos generados de forma periódica y automática a partir de fuentes de datos abiertos.

CAPÍTULO 5
Conclusiones

En este trabajo de fin de máster se ha colaborado con el grupo de Grid y Computación de Altas Prestaciones de la Universitat Politècnica de València para desarrollar una plataforma Serverless híbrida de procesado de datos.

En primer lugar se ha realizado un estudio, pasando por un proceso de aprendizaje, de las herramientas necesarias para conformar la base de la plataforma. Seguidamente se han integrado los componentes específicos para disponer de un sistema de almacenamiento interno, una plataforma de funciones como servicio, un registro de contenedores de *software* y una herramienta para construirlos. Una vez lograda la integración se ha trabajado en la creación de roles con una aplicación de automatización que, en conjunto con la herramienta EC3, ha permitido definir una receta de despliegue para replicar la configuración de la plataforma sobre cualquier infraestructura. Esta receta ha sido probada satisfactoriamente, por lo que se podría afirmar que uno de los objetivos más importantes del trabajo ha sido alcanzado.

Además, durante el proceso de integración se detectó que el comportamiento de la plataforma no era el esperado en cuanto a términos de elasticidad y paralelismo, lo que propició el desarrollo de un nuevo componente. Este componente, explotando las características del orquestador de contenedores Kubernetes, asegura el máximo aprovechamiento de los recursos disponibles en la infraestructura.

Por otra parte, se ha hecho un análisis de la solución de almacenamiento y gestión de datos Onedata con el fin de soportar a la misma como sistema de almacenamiento externo. Este objetivo se ha logrado gracias al desarrollo de una herramienta, llamada OneTrigger, capaz de generar eventos ante la subida de ficheros. Complementada con la adaptación del supervisor, componente encargado de la carga y descarga de los archivos a procesar, ha servido para demostrar el correcto funcionamiento de la plataforma en entornos híbridos, logrando así otro de los objetivos principales.

Con la plataforma ya definida y cumplidos los objetivos principales, se han implementado dos casos de uso. Estos casos de uso han explotado aplicaciones y datos abiertos para generar información y, además, demostrar el correcto funcionamiento de la plataforma.

Otro punto a destacar ha sido la colaboración con otros investigadores del grupo en la definición de dos nuevos componentes: un gestor de funciones, encargado de la creación, modificación y eliminación de las mismas, así como de la creación de *buckets* y carpetas en los sistemas de almacenamiento soportados; y una interfaz web capaz de comunicarse con el gestor. Estos componentes permiten que la plataforma sea fácil de utilizar por usuarios con poca experiencia.

Desde el punto de vista personal, la realización de este trabajo ha supuesto una enriquecedora experiencia para el autor. Se ha podido trabajar junto a un equipo de grandes investigadores, se han utilizando herramientas innovadoras y se han afianzado muchos conceptos aprendidos en el máster. Además, se ha logrado desarrollar un producto capaz de resolver problemas reales que ha sido publicado como *software* libre para que otras comunidades puedan beneficiarse de él. Todos estos motivos, así como un gran interés en este apasionante campo, han impulsado al autor a continuar investigando y formándose a través del programa de doctorado en informática.

En conclusión, se han alcanzado todos los objetivos establecidos de forma satisfactoria, solucionando los problemas encontrados a través de un proceso de análisis, diseño e implementación. La plataforma ha sido publicada de forma abierta y su desarrollo continúa a través de mejoras y nuevas funcionalidades.

5.1 Trabajo futuro

Para continuar mejorando la plataforma y así satisfacer las necesidades de un mayor número de usuarios, se han identificado las siguientes ampliaciones:

- Desacoplamiento del supervisor de la plataforma. Puesto que este elemento guarda muchas similitudes con el utilizado en la herramienta SCAR, tambien desarrollada por el grupo de Grid y Computación de Altas Prestaciones, se ha planteado el desarrollo de un único supervisor con el fin de reducir las tareas de mantenimiento. Esta mejora se encuentra en proceso de desarrollo en el momento de publicación del trabajo.
- Soporte a otros sistemas de almacenamiento externos, como Ceph [70] o dCache [71].
- Creación de una interfaz de línea de comandos para usuarios avanzados o integración con el cliente de SCAR.

5.2 Contribuciones científicas

El proyecto relacionado con este Trabajo Fin de Máster ha originado las siguientes contribuciones científicas:

- Alfonso Pérez, Sebastián Risco, Germán Moltó, Miguel Caballer, Amanda Calatrava. *On-premises Serverless Container-aware ARchitectures (OSCAR)*. Presentación en el congreso *Ibergrid 2018*, 11-12 de octubre del 2018 en Lisboa, Portugal.
- Alfonso Pérez, Sebastián Risco, Miguel Caballer, Diana M. Naranjo, Germán Moltó. *OSCAR: Open Source Serverless Computing for Data-Processing Applications*. Póster en el congreso *EOSC-hub Week 2019*, 10-12 de abril del 2019 en Praga, República Checa.

- Alfonso Pérez, Sebastián Risco, Germán Moltó, Miguel Caballer, Diana María Naranjo. *OSCAR: Open Source Serverless Computing for Data-Processing Applications*. Presentación en el congreso *EGI Conference 2019*, 6-8 de mayo del 2019 en Amsterdam, Paises Bajos.
- Alfonso Pérez, Sebastián Risco, Diana María Naranjo, Miguel Caballer, Germán Moltó. *Serverless Computing for Event-Driven Data Processing Applications*. Artículo publicado en el congreso *2019 IEEE International Conference on Cloud Computing (CLOUD 2019)*, 8-13 de julio en Milán, Italia. [59].

El autor participó en la presentación realizada en la EGI Conference 2019 con el objetivo de presentar este desarrollo a la EGI Foundation, ya que ha sido financiado parcialmente por esta entidad y se espera que, en los próximos meses, se incorpore como un servicio en pruebas dentro del catálogo de servicios de EGI.

Bibliografía

[1] Rajkumar Buyya, James Broberg y Andrzej Goscinski, eds. *Cloud Computing: Principles and Paradigms*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 28 de feb. de 2011. ISBN: 978-0-470-94010-5 978-0-470-88799-8. DOI: 10.1002/9780470940105. URL: http://doi.wiley.com/10.1002/9780470940105 (visitado 10-07-2019).

[2] Google. *Documentos de Google: Crea y Edita Documentos Online de Forma Gratuita.* URL: https://www.google.es/intl/es/docs/about/ (visitado 04-03-2019).

[3] Dropbox. *Dropbox.* URL: https://www.dropbox.com/es_ES/ (visitado 04-03-2019).

[4] Spotify. *Música para todos.* URL: https://www.spotify.com/es/ (visitado 04-03-2019).

[5] Netflix. *Ver Series En Línea, Ver Películas En Línea*. URL: https://www.netflix.com/es/ (visitado 04-03-2019).

[6] ENSI-MARIA. *IaaS, PaaS, SaaS – What Do They Mean? | CloudOnMove*. URL: http://cloudonmove.com/iaas-paas-saas-what-do-they-mean/ (visitado 04-03-2019).

[7] AWS. *Introducing AWS Lambda*. URL: https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/ (visitado 24-02-2019).

[8] AWS. *Lambda - Gestión de recursos informáticos*. URL: https://aws.amazon.com/es/lambda/ (visitado 23-02-2019).

[9] Erwin van Eyk, Alexandru Iosup, Simon Seif y col. "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures". En: *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*. The 2nd International Workshop. Las Vegas, Nevada: ACM Press, 2017, págs. 1-4. ISBN: 978-1-4503-5434-9. DOI: 10.1145/3154847.3154848. URL: http://dl.acm.org/citation.cfm?doid=3154847.3154848 (visitado 10-07-2019).

[10] GRyCAP. *Serverless Container-Aware ARchitectures (e.g. Docker in AWS Lambda)*. GRyCAP, 16 de feb. de 2019. URL: https://github.com/grycap/scar (visitado 17-02-2019).

[11] Docker. *Enterprise Container Platform*. URL: https://www.docker.com/index.html (visitado 24-02-2019).

[12] AWS. *Almacenamiento de datos seguro en la nube (S3)*. URL: https://aws.amazon.com/es/s3/ (visitado 29-06-2019).

[13] Alfonso Pérez, Germán Moltó, Miguel Caballer y col. "Serverless Computing for Container-Based Architectures". En: *Future Generation Computer Systems* 83 (jun. de 2018), págs. 50-59. ISSN: 0167739X. DOI: 10.1016/j.future.2018.01.022. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X17316485 (visitado 17-02-2019).

[14] Kubernetes. *Production-Grade Container Orchestration*. URL: https://kubernetes.io/ (visitado 28-02-2019).

[15] OpenStack. *Build the Future of Open Infrastructure*. URL: https://www.openstack.org/ (visitado 22-06-2019).

[16] OpenNebula. *Home*. URL: https://opennebula.org/ (visitado 23-02-2019).

[17] AWS. *Elastic compute cloud (EC2) de capacidad modificable en la nube*. URL: https://aws.amazon.com/es/ec2/ (visitado 29-06-2019).

[18] AWS. *Tipos de instancias de Amazon EC2*. URL: https://aws.amazon.com/es/ec2/instance-types/ (visitado 03-07-2019).

[19] NASA. *National Aeronautics and Space Administration*. URL: http://www.nasa.gov/index.html (visitado 09-07-2019).

[20] Rackspace. *Managed Dedicated & Cloud Computing Services*. URL: https://www.rackspace.com/es (visitado 09-07-2019).

[21] OpenStack. *OpenStack Docs: Design*. URL: https://docs.openstack.org/arch-design/design.html (visitado 10-07-2019).

[22] Free Software Fundation. *gnu.org*. URL: https://www.gnu.org/home.es.html (visitado 09-06-2019).

[23] Docker. *Docker Overview*. 2 de jul. de 2019. URL: https://docs.docker.com/engine/docker-overview/ (visitado 09-07-2019).

[24] Docker. *Docker Registry*. 22 de feb. de 2019. URL: https://docs.docker.com/registry/ (visitado 24-02-2019).

[25] Docker. *Docker Hub*. URL: https://hub.docker.com/ (visitado 24-02-2019).

[26] CNCF. *Cloud Native Computing Foundation*. URL: https://www.cncf.io/ (visitado 10-07-2019).

[27] CNCF. *Etcd: A Distributed, Reliable Key-Value Store for the Most Critical Data of a Distributed System*. URL: https://etcd.io/ (visitado 10-07-2019).

[28] PRADEEP KUMAR. KUMARI SINGH MADHURI. *CONTAINERIZATION IN OPENSTACK*. OCLC: 1015847246. Place of publication not identified: PACKT Publishing Limited, 2018. ISBN: 978-1-78839-438-3.

[29] GoogleContainerTools. *Build Container Images In Kubernetes*. GoogleContainerTools, 29 de jun. de 2019. URL: https://github.com/GoogleContainerTools/kaniko (visitado 29-06-2019).

[30] Google Cloud. *Introducing Kaniko: Build Container Images in Kubernetes and Google Container Builder without Privileges*. URL: https://cloud.google.com/blog/products/gcp/introducing-kaniko-build-container-images-in-kubernetes-and-google-container-builder-even-without-root-access/ (visitado 11-07-2019).

[31] AWS. *Amazon API Gateway*. URL: https://aws.amazon.com/es/api-gateway/ (visitado 09-07-2019).

[32] GRyCAP. *Event-Driven File-Processing Programming Model — Scar Documentation*. URL: https://scar.readthedocs.io/en/latest/prog_model.html (visitado 11-07-2019).

[33] AWS. *Amazon CloudWatch: Monitoreo de infraestructuras y aplicaciones*. URL: https://aws.amazon.com/es/cloudwatch/ (visitado 09-07-2019).

[34] OpenFaaS. *Serverless Functions Made Simple*. URL: https://www.openfaas.com/ (visitado 12-02-2019).

[35] Docker. *Swarm Mode Overview*. 2 de jul. de 2019. URL: https://docs.docker.com/engine/swarm/ (visitado 10-07-2019).

[36] OpenFaaS. *OpenFaaS API Gateway / Portal*. URL: https://docs.openfaas.com/architecture/gateway/#openfaas-api-gateway-portal (visitado 10-07-2019).

[37] MinIO Inc. *Object Storage for AI*. URL: https://min.io (visitado 29-06-2019).

[38] MinIO Inc. *MinIO Object Storage*. URL: https://min.io (visitado 10-07-2019).

[39] Onedata. *Global Data Access Solution for Science*. URL: https://onedata.org/#/home (visitado 29-06-2019).

[40] Onedata. *Oneprovider Overview*. URL: https://onedata.org/#/home/documentation/doc/administering_onedata/provider_overview.html (visitado 10-07-2019).

[41] Onedata. *Onezone Overview*. URL: https://onedata.org/#/home/documentation/doc/administering_onedata/onezone_overview.html (visitado 10-07-2019).

[42] Ansible, Red Hat. *Ansible Is Simple IT Automation*. URL: https://www.ansible.com (visitado 28-02-2019).

[43] Ansible, Red Hat. *Ansible Galaxy*. URL: https://galaxy.ansible.com/ (visitado 11-07-2019).

[44] GRyCAP. *CLUES - Cluster Energy Saving (for Hpc and Cloud Computing)*. URL: https://www.grycap.upv.es/clues/es/index.php (visitado 13-02-2019).

[45] GRyCAP. *IM - Infrastructure Manager*. URL: https://www.grycap.upv.es/im/index.php (visitado 13-02-2019).

[46] Miguel Caballer, Ignacio Blanquer, Germán Moltó y col. "Dynamic Management of Virtual Infrastructures". En: *Journal of Grid Computing* 13.1 (mar. de 2015), págs. 53-70. ISSN: 1570-7873, 1572-9184. DOI: 10.1007/s10723-014-9296-5. URL: http://link.springer.com/10.1007/s10723-014-9296-5 (visitado 23-02-2019).

[47] GRyCAP. *Resource and Application Description Language (RADL) — IM Documentation 1.0 Documentation*. URL: https://imdocs.readthedocs.io/en/latest/radl.html (visitado 11-07-2019).

[48] Palma Derek, Spatzier Thomas y Rutkowski Matt. *TOSCA Simple Profile in YAML Version 1.1*. 2016. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/TOSCA-Simple-Profile-YAML-v1.1.html.

[49] M Caballer, G Donvito, G Moltó y col. "TOSCA-Based Orchestration of Complex Clusters at the IaaS Level". En: *Journal of Physics: Conference Series* 898 (oct. de 2017), pág. 082036. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/898/8/082036. URL: http://stacks.iop.org/1742-6596/898/i=8/a=082036?key=crossref.af71f04f17660fdd1e050f7c1e00b643 (visitado 10-07-2019).

[50] GRyCAP. *IM - Infrastructure Manager. Overview*. URL: https://www.grycap.upv.es/im/overview.php (visitado 23-02-2019).

[51] GRyCAP. *EC3 - Elastic Cloud Computing Cluster*. URL: https://servproject.i3m.upv.es/ec3/ (visitado 13-02-2019).

[52] Miguel Caballer, Carlos de Alfonso, Fernando Alvarruiz y col. "EC3: Elastic Cloud Computing Cluster". En: *Journal of Computer and System Sciences* 79.8 (dic. de 2013), págs. 1341-1351. ISSN: 00220000. DOI: 10.1016/j.jcss.2013.06.005. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022000013001141 (visitado 28-06-2019).

[53] GRyCAP. *Architecture — EC3 Documentation*. URL: https://ec3.readthedocs.io/en/latest/arch.html#general-architecture (visitado 23-02-2019).

[54] Python Software Foundation. *Welcome to Python.Org*. URL: https://www.python.org/ (visitado 08-07-2019).

[55] CNCF. *NATS - Open Source Messaging System | Secure, Native Cloud Application Development*. URL: https://nats.io/ (visitado 13-02-2019).

[56] GitHub. *Build Software Better, Together*. URL: https://github.com (visitado 29-06-2019).

[57] Git. *Git*. URL: https://git-scm.com/ (visitado 09-07-2019).

[58] GRyCAP. *Open Source Serverless Computing for Data-Processing Applications*. GRyCAP, 10 de jul. de 2019. URL: https://github.com/grycap/oscar (visitado 11-07-2019).

[59] Alfonso Pérez, Sebastián Risco, Diana María Naranjo y col. "Serverless Computing for Event-Driven Data Processing Applications". En: *2019 IEEE International Conference on Cloud Computing (CLOUD 2019)*. 2019.

[60] OpenFaaS. *NATS Streaming Integration for OpenFaaS*. OpenFaaS, 22 de feb. de 2019. URL: https://github.com/openfaas/nats-queue-worker (visitado 24-02-2019).

[61] Matthew Viljoen, Łukasz Dutka, Bartosz Kryza y col. "Towards European Open Science Commons: The EGI Open Data Platform and the EGI DataHub". En: *Procedia Computer Science* 97 (2016), págs. 148-152. ISSN: 18770509. DOI: 10.1016/j.procs.2016.08.294. URL: https://linkinghub.elsevier.com/retrieve/pii/S187705091632110X (visitado 29-06-2019).

[62] EGI. *EGI Advanced Computing Services for Research*. URL: https://www.egi.eu/ (visitado 29-06-2019).

[63] SNIA. *Cloud Data Management Interface (CDMI)*. URL: https://www.snia.org/cdmi (visitado 04-03-2019).

[64] Ignacio Heredia. “Large-Scale Plant Classification with Deep Neural Networks”. En: *Proceedings of the Computing Frontiers Conference on ZZZ - CF’17*. The Computing Frontiers Conference. Siena, Italy: ACM Press, 2017, págs. 259-262. ISBN: 978-1-4503-4487-6. DOI: 10.1145/3075564.3075590. URL: http://dl.acm.org/citation.cfm?doid=3075564.3075590 (visitado 13-02-2019).

[65] The World Air Quality Index project. *World’s Air Pollution: Real-Time Air Quality Index*. URL: https://waqi.info/ (visitado 03-07-2019).

[66] World Air Quality Index project. *Real-Time Air Quality Data Feed - JSON API*. URL: https://aqicn.org/json-api/doc/ (visitado 06-07-2019).

[67] freedesktop.org. *Systemd*. URL: https://www.freedesktop.org/wiki/Software/systemd/ (visitado 08-07-2019).

[68] ArchWiki. *Systemd/Timers*. URL: https://wiki.archlinux.org/index.php/Systemd/Timers (visitado 08-07-2019).

[69] The Matplotlib development team. *Matplotlib: Python Plotting*. URL: https://matplotlib.org/ (visitado 08-07-2019).

[70] Ceph. *Homepage*. URL: https://ceph.com/ (visitado 07-07-2019).

[71] dCache. *Main Page*. URL: https://www.dcache.org/ (visitado 07-07-2019).

APÉNDICE A
Receta de despliegue para EC3

Disponible en el repositorio principal del proyecto [58]. Versión actualizada el 14-05-2019.

```
1
2  description kubernetes (
3      kind = 'main' and
4      short = 'Install and configure a cluster using the grycap.
           kubernetes ansible role and install all needed services
           to run OSCAR.' and
5      content = 'The template installs the grycap.kubernetes
           ansible role. Initially the template creates as many
           working node hostnames as the sum of the values of
           feature "ec3_max_instances_max" in every system.
6
7  Webpage: https://kubernetes.io/'
8  )
9
10 network public (
11   # kubernetes ports
12   outbound = 'yes' and
13   outports contains '443/tcp,22/tcp,6443/tcp,31112/tcp,32112/tcp
         ,31852/tcp,8800/tcp'
14 )
15
16 network private ()
17
18 system front (
19   cpu.count>=2 and
20   memory.size>=4096m and
21   net_interface.0.connection = 'private' and
22   net_interface.0.dns_name = 'kubeserver' and
23   net_interface.1.connection = 'public' and
24   net_interface.1.dns_name = 'kubeserverpublic' and
25   queue_system = 'kubernetes' and
26   ec3_templates contains 'kubernetes_oscar' and
27   disk.0.applications contains (name = 'ansible.modules.grycap.
         kubernetes') and
```

```
28   disk.0.applications contains (name = 'ansible.modules.grycap.
       nfs') and
29   disk.0.applications contains (name = 'ansible.modules.grycap.
       kubefaas') and
30   disk.0.applications contains (name = 'ansible.modules.grycap.
       kubeminio') and
31   disk.0.applications contains (name = 'ansible.modules.grycap.
       kuberegistry') and
32   disk.0.applications contains (name = 'ansible.modules.grycap.
       kubeoscar') and
33   disk.0.applications contains (name = 'ansible.modules.grycap.
       clues') and
34   disk.0.applications contains (name = 'ansible.modules.grycap.
       im') and
35   disk.1.type='standard' and
36   disk.1.size=20GB and
37   disk.1.device='vdf' and
38   disk.1.fstype='ext4' and
39   disk.1.mount_path='/pv/minio' and
40   disk.2.type='standard' and
41   disk.2.size=20GB and
42   disk.2.device='vdg' and
43   disk.2.fstype='ext4' and
44   disk.2.mount_path='/pv/registry'
45 )
46
47 configure front (
48 @begin
49 ---
50   - vars:
51       AUTH:
52         ec3_xpath: /system/front/auth
53       SYSTEMS:
54         ec3_jpath: /system/*
55       NNODES: '{{ SYSTEMS | selectattr("ec3_max_instances_max",
           "defined") | sum(attribute="ec3_max_instances_max") }}'
56
57     pre_tasks:
58     - name: Create dir for kaniko builds
59       file: path=/pv/kaniko-builds state=directory mode=755
60     - name: Create auth file dir
61       file: path=/etc/kubernetes/pki state=directory mode=755
           recurse=yes
62     - name: Create auth data file with an admin user
63       copy: content='{{ lookup('password', '/var/tmp/
           dashboard_token chars=ascii_lowercase,digits length
           =16') }},kubeuser,100,"users,system:masters"' dest=/etc
           /kubernetes/pki/auth mode=600
64     - name: Generate minio secret key
65       set_fact:
66         minio_secret: "{{ lookup('password', '/var/tmp/
             minio_secret_key chars=ascii_letters,digits') }}"
```

```
67
68      roles:
69      - role: 'grycap.nfs'
70        nfs_mode: 'front'
71        nfs_exports:
72        - {path: "/pv/minio", export: "*.localdomain(rw,async,
             no_root_squash,no_subtree_check,insecure)"}
73        - {path: "/pv/registry", export: "*.localdomain(rw,async,
             no_root_squash,no_subtree_check,insecure)"}
74        - {path: "/pv/kaniko-builds", export: "*.localdomain(rw,
             async,no_root_squash,no_subtree_check,insecure)"}
75
76      - role: 'grycap.kubernetes'
77        kube_server: 'kubeserver'
78        kube_apiserver_options:
79        - {option: "--insecure-port", value: "8080"}
80        - {option: "--token-auth-file", value: "/etc/kubernetes/
             pki/auth"}
81        - {option: "--service-node-port-range", value: "80-32767"}
82        kube_deploy_dashboard: true
83        kube_install_metrics: true
84        kube_persistent_volumes:
85        - {namespace : "minio", name : "pvnfsminio", label : "
             minio", capacity_storage : "20Gi", nfs_path : "/pv/
             minio"}
86        - {namespace : "docker-registry", name : "pvnfsregistry",
             label : "registry", capacity_storage : "20Gi", nfs_path
              : "/pv/registry"}
87        - {namespace : "oscar", name : "pvnfskanikobuilds", label
             : "oscar-manager", capacity_storage : "2Gi", nfs_path :
              "/pv/kaniko-builds"}
88        kube_version: 1.13.6
89
90      - role: 'grycap.kubefaas'
91        faas_framework: 'openfaas'
92        master_deploy: true
93        faas_chart_version: 3.3.0
94        cli_version: 0.8.11
95        oscar_worker_version: 1.2.0
96
97      - role: 'grycap.kubeminio'
98        enable_notifications: true
99        webhook_endpoints: [{ id: "1", endpoint: "http://oscar-
             manager.oscar:8080/events"}]
100       minio_secretkey: '{{ minio_secret }}'
101       master_deploy: true
102
103     - role: 'grycap.kuberegistry'
104       public_access: false
105       type_of_node: "front"
106       svc_name: "registry.docker-registry"
107       delete_enabled: true
```

```
108       master_deploy: true
109
110     - role: 'grycap.kubeoscar'
111       minio_pass: '{{ minio_secret }}'
112       vue_app_backend_host: '{{ hostvars[groups["front"][0]]["
             IM_NODE_PUBLIC_IP"] }}:{{ nginx_https_nodeport }}'
113       master_deploy: true
114       oscar_version: 1.1.1
115       oscar_ui_version: 1.0.0
116       supervisor_version: 1.0.5
117       onetrigger_version: 1.0.3
118
119     - role: 'grycap.im'
120
121     - role: 'grycap.clues'
122       auth: '{{AUTH}}'
123       clues_queue_system: kubernetes
124       max_number_of_nodes: '{{ NNODES }}'
125       vnode_prefix: 'wn'
126       clues_config_options:
127           - { section: 'scheduling', option: 'IDLE_TIME', value:
                 '300' }
128           - { section: 'scheduling', option: '
                RECONSIDER_JOB_TIME', value: '60' }
129           - { section: 'monitoring', option: 'MAX_WAIT_POWERON',
                 value: '3000' }
130           - { section: 'monitoring', option: 'MAX_WAIT_POWEROFF
                ', value: '600' }
131           - { section: 'monitoring', option: 'PERIOD_LIFECYCLE',
                 value: '10' }
132           - { section: 'monitoring', option: '
                PERIOD_MONITORING_NODES', value: '2' }
133           - { section: 'client', option: '
                CLUES_REQUEST_WAIT_TIMEOUT', value: '3000' }
134           # These options enable to have always one slot free
135           - { section: 'scheduling', option: 'SCHEDULER_CLASSES
                ', value: 'clueslib.schedulers.
                CLUES_Scheduler_PowOn_Requests, clueslib.schedulers
                .CLUES_Scheduler_Reconsider_Jobs, clueslib.
                schedulers.CLUES_Scheduler_PowOff_IDLE, clueslib.
                schedulers.CLUES_Scheduler_PowOn_Free' }
136           - { section: 'scheduling', option: 'EXTRA_SLOTS_FREE',
                 value: '1' }
137
138 @end
139 )
140
141 system wn (
142   cpu.count>=2 and
143   memory.size>=4096m and
144   ec3_node_type = 'wn' and
145   net_interface.0.connection='private'
```

```
146 )
147
148 configure wn (
149 @begin
150 ---
151   - roles:
152     - role: 'grycap.nfs'
153       nfs_mode: 'wn'
154       nfs_client_imports:
155       - {local: "/pv/minio", remote: "/pv/minio", server_host: "
              kubeserver.localdomain"}
156       - {local: "/pv/registry", remote: "/pv/registry",
              server_host: "kubeserver.localdomain"}
157       - {local: "/pv/kaniko-builds", remote: "/pv/kaniko-builds
              ", server_host: "kubeserver.localdomain"}
158
159     - role: 'grycap.kubernetes'
160       kube_type_of_node: 'wn'
161       kube_server: 'kubeserver'
162       kube_version: 1.13.6
163
164     - role: 'grycap.kuberegistry'
165       public_access: false
166       type_of_node: "wn"
167       svc_name: "registry.docker-registry"
168 @end
169 )
170
171 include kube_misc (
172   template = 'openports'
173 )
174
175 deploy front 1
```