UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación

Universitat Politècnica de València

Aplicación Web para Composición de Workflows Serverless

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas prestaciones

Autor: David Soler Marco

Tutor: Germán Moltó Martínez

Director experimental: Sebastián Risco Gallardo

Curso: 2020-2021

Fecha: 09/04/2021

Resumen

Este TFM plantea la creación de una aplicación web que posibilite la composición de workflows dirigidos por eventos para la herramienta SCAR, que permite la ejecución de cadenas de funciones (modalidad FaaS - Functions as a Service) sobre el servicio AWS Lambda y sobre AWS Batch. También será compatible con la herramienta de código abierto OSCAR para el soporte a la computación serverless científica sobre plataformas Cloud on-premises, permitiendo la composición de workflows híbridos entre ambos sistemas.

SCAR utiliza el lenguaje YAML para la definición de dichos workflows por lo que la aplicación web permitirá la definición gráfica de los componentes del workflow para acabar produciendo como resultado dicho fichero YAML, que podrá ser usado como entrada para su ejecución delegando en la herramienta SCAR, aunque también será posible utilizarlo con OSCAR.

La aplicación web se desarrollará mediante tecnologías de web estática para que la aplicación pueda ejecutarse de forma serverless mediante alguna tecnología de hosting como GitHub pages o Amazon S3.

Palabras clave: Computación en la nube, Serverless, SCAR, AWS, React, Flujos de trabajo

Abstract

The purpose of this master's thesis is the development of a web application that allows event-driven workflow composition for the SCAR tool, which allows the execution of functions (Functions as a Service) via AWS Lambda and over AWS Batch. In addition, it will be also compatible with the open source tool OSCAR to support serverless computing for data-processing applications over on-premises Cloud platforms, allowing hybrid workflow composition between both systems.

SCAR uses YAML language files to define workflows so this web application will provide a graphic user interface that will generate these YAML files, which will be used as an input for the execution in SCAR and OSCAR.

This web app will be developed using client-side rendering (CSR) frameworks, as this technology can be deployed as a static web site on a hosting provider like GitHub pages or Amazon S3.

Keywords: Cloud computing, Serverless, SCAR, AWS, React, Workflows

Tabla de contenidos

1. Introducción

Motivación

La computación en la nube ha reducido considerablemente la necesidad de adquirir y gestionar servidores. Si a esto le sumamos los nuevos patrones arquitectónicos, la adopción masiva de máquinas virtuales y contenedores Linux, y las mejoras en computación en la nube como el auto-escalado, obtenemos un cambio de paradigma que nos traslada de sistemas monolíticos a pequeños servicios sin estado.

En el modelo de plataforma como servicio la gestión de la infraestructura de la aplicación se encuentra delegada en el proveedor, lo que evita tener que preocuparse por el aprovisionamiento de recursos necesarios para la ejecución de las aplicaciones. El modelo de función como servicio va todavía más lejos, permitiendo desplegar un fragmento de código para ser ejecutado bajo demanda sin necesidad de tener un servidor contratado, pagando solo por el uso de la función.

Los proveedores de computación en la nube como Amazon Web Services (AWS) [1], Microsoft Azure [2] o Google Cloud [3] han favorecido la migración de aplicaciones con una arquitectura compleja a la nube, consiguiendo las ventajas de un modelo de pago por uso.

En los últimos años han proliferado los servicios del tipo *Function as a Service* (FaaS), proveyendo de un ecosistema que permite la ejecución de funciones bajo demanda, en cualquier momento, y sin que el usuario tenga que configurar los detalles de la infraestructura subyacente. Habitualmente se trata de servicios que no requieren un coste fijo, sino que se tarifican mediante un modelo de pago por uso con una granularidad más fina que la empleada en el modelo Infrastructure as a Service (IaaS). AWS Lambda [4] fue uno de los primeros servicios de este tipo en aparecer, seguido por Google Cloud Functions [5], Microsoft Azure Functions [6], y la plataforma de código abierto Apache OpenWhisk [7].

En definitiva, los proveedores de servicio ya no solo ofrecen un modelo de infraestructura como servicio, donde únicamente se proporcionaba un servidor o una máquina virtual. Ahora, además de estos, ofrecen modelos más completos, como los modelos de plataforma como servicio (PaaS - Platform as a Service) y función como servicio (FaaS- Function as a Service). Estos modelos ofrecen un mayor nivel de abstracción para el desarrollador pero a su vez han generado nuevas necesidades.

El nacimiento de estos servicios ha derivado en la aparición de prácticas que se benefician de esta arquitectura, como es el modelo de la programación dirigida por eventos. Este es el caso de [8] donde se presenta un prototipo de arquitectura de chatbot usando la plataforma OpenWhisk o el uso en [9] para el análisis de datos mediante Spark [10] y OpenWhisk.

La creación de aplicaciones distribuidas basadas en microservicios ha desencadenado en la necesidad de gestionar el ciclo de vida y escalar los contenedores software donde se ejecutan, lo que ha derivado en la aparición de plataformas de orquestación de contenedores como Kubernetes [11], sin embargo, estas plataformas no nos proporcionan todas las características que podemos necesitar. El trabajo de Jonas et al. [12] describe los tipos de aplicaciones susceptibles de ser usadas en este tipo de arquitectura sacando a relucir las limitaciones que actualmente presentan las infraestructuras sin servidor, como pueden ser carencias en el almacenamiento para las operaciones de grano fino, el bajo rendimiento con los patrones de comunicación estándar o la falta de coordinación entre funciones.

Los servicios típicamente ofrecidos por un proveedor de cloud público son:

1. Funciones como servicio (FaaS), ejecutadas en respuesta a eventos.

2. Proveedores/Servicios de almacenamiento (Storage providers), que alojan ficheros provocando eventos.

3. Servicio de trazabilidad (Log service), donde se registran las ejecuciones de las funciones.

4. Servicio de monitorización, que proporciona una visión sobre los recursos consumidos por la función.

SCAR (Serverless Container-aware ARchitectures) [13] es una herramienta de código abierto que permite la ejecución de contenedores Docker [14] en AWS a partir de imágenes Docker que pueden estar almacenadas en Docker Hub. En SCAR, mediante un fichero YAML es posible especificar todo un flujo de trabajo, incluyendo proveedores de datos, entradas, salidas y funciones.

SCAR permite la generación de flujos serverless mediante la combinación de funciones que pueden ser ejecutadas sobre AWS Batch [15] o AWS Lambda, y el proveedor de datos Amazon S3 [16], proporcionando un entorno altamente escalable. La integración con AWS Batch permite superar las limitaciones existentes en AWS Lambda, pues permite ejecutar aplicaciones que requieran más de 15 minutos de tiempo de ejecución, más de 512 MBs de espacio de almacenamiento temporal y el uso de GPUs para acelerar la computación.

Por su parte, OSCAR - Open Source Serverless Computing for Data-Processing Applications [17] es una plataforma de código abierto que soporta el paradigma de ejecución Serverless dirigido por eventos mediante ejecución de funciones como servicio. OSCAR implementa el mismo modelo de computación ofrecido por SCAR [18] pero *on-premises,* es decir, en la infraestructura propia de una organización. Esto permite crear un entorno para la ejecución de aplicaciones dirigido por eventos sobre un cluster Kubernetes elástico. OSCAR permite utilizar el proveedor MinIO [19] como fuente de eventos y almacén de datos y los proveedores de datos Amazon S3 y Onedata [20] para almacenar los resultados de salida

En ambos casos, el flujo es el siguiente. El usuario sube un fichero (o varios) a un sistema de almacenamiento, lo que provoca el disparo de un evento que provoca la ejecución de una función encargada de crear un contenedor a partir de una imagen de contenedor Docker que ejecuta un shell-script definido por el usuario encargado del procesado de dicho fichero. El resultado, en la forma de fichero, se guarda automáticamente en el proveedor de almacenamiento definido. El escalado de la plataforma y la transferencia de los ficheros de entrada y de salida se gestiona de forma automáticamente por la herramienta, ya sea SCAR u OSCAR.

Objetivos

Este proyecto plantea desarrollar una herramienta gráfica para la composición de workflows serverless y la generación automatizada del fichero YAML que lo describe facilitando así la creación de flujos de trabajos sobre SCAR.

Dado que el tamaño de los ficheros para la generación de un flujo de trabajo y su complejidad puede llegar a ser muy elevada, en el presente documento se plantea un proyecto para desarrollar una herramienta visual que permita generar estos ficheros YAML de manera gráfica, permitiendo ver de una forma fácil y clara todos los elementos que operan en la plataforma y el flujo de trabajo.

El usuario podrá, de forma intuitiva, seleccionar los elementos que desea incluir en su flujo de trabajo: proveedores de datos, funciones a ejecutar y entradas y salidas de datos desde/hacia los proveedores de datos.

Para lograr que este servicio sea escalable y pueda ser desplegado de manera que no sea necesario tener el coste fijo asociado a un servidor (serverless), optamos por un lenguaje Frontend que renderice del lado del cliente. Esto permitirá la creación de una web estática, compuesta únicamente por el código (HTML), hojas de estilo (CSS) y scripts (JavaScript) que podrá ser alojada en cualquier servidor de páginas web estáticas como pueda ser GitHub Pages.

El estado de la aplicación se gestionará mediante la importación y exportación de ficheros JSON, careciendo de estado (stateless) y eliminando la dependencia de un servicio para la obtención y almacenamiento de datos.

2. Tecnologías Relacionadas y Estado del Arte

Tecnologías Relacionadas

Proveedores de datos

Amazon S3

Amazon S3 es un servicio de almacenamiento de objetos ofrecido por Amazon Web Services proporcionando una interfaz de servicio web. El tamaño máximo del objeto es de 5TB.

MinIO

MinIO es un servidor de almacenamiento en la nube compatible con Amazon S3. Puede almacenar datos como fotos, vídeos, o incluso imágenes de contenedor. El tamaño máximo del objeto es de 5TB.

Onedata

Onedata es una solución de alto rendimiento para la gestión de datos que ofrece un acceso global unificado entre entornos distribuidos.

Funciones

SCAR

La herramienta SCAR [21] permite la creación de aplicaciones sin gestión explícita de servidores (*serverless*) guiadas por eventos altamente paralelizables, siendo compatible con cualquier aplicación que pueda ejecutarse dentro de un contenedor.

La computación *serverless* en la nube proporciona un nuevo nivel de paralelismo que permite un diseño de aplicaciones guiado por eventos. Este modelo de programación es especialmente útil ante la siguiente problemática:

Un usuario quiere procesar un fichero que se almacenará en un bucket de un proveedor de almacenamiento. La subida del fichero desencadena la ejecución

de una función cuya salida se almacenará en otro bucket del mismo servicio. En la Ilustración 1 podemos ver un ejemplo de este flujo.

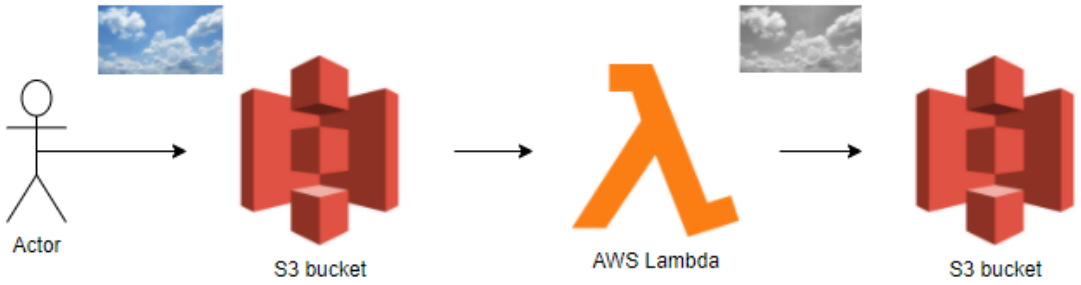

Ilustración 1. Flujo dirigido por eventos

SCAR permite al usuario definir funciones Lambda, donde cada ejecución ejecutará un contenedor a partir de una imagen Docker almacenada en Docker Hub [22].

La Ilustración 2 muestra la arquitectura de SCAR. El usuario selecciona una imagen Docker ubicada en Docker Hub y crea una función lambda con unos requisitos determinados (CPU, RAM, etc.), e indica en los proveedores de almacenamiento que actúan como entrada y salida.

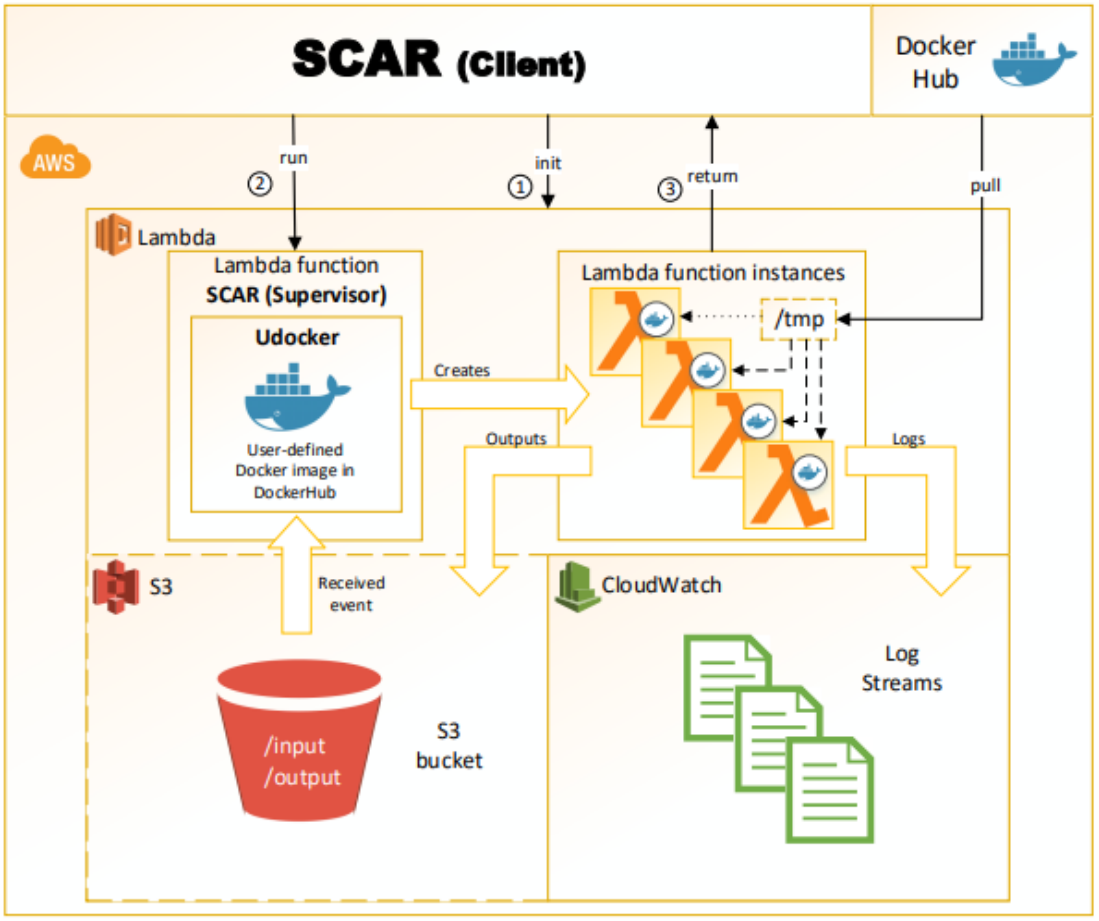

Ilustración 2. Arquitectura SCAR. [23]

OSCAR

La plataforma OSCAR [24] controla el flujo de trabajo de la aplicación mediante *triggers*. La subida de un fichero a un proveedor de almacenamiento desencadena un evento que provocará la ejecución de una función encargada de procesar dicho fichero.

OSCAR se ejecuta en un clúster elástico de Kubernetes desplegado utilizando:

- EC3: Elastic Cloud Computing Cluster [25], una herramienta de código abierto para el despliegue de clústeres de cómputo que puede escalar de forma horizontal en términos de número de nodos.
- IM: Infrastructure Manager [26], una herramienta para el aprovisionamiento de infraestructuras virtuales.

- CLUES: Cluster Energy Saving [27], un administrador de elasticidad que permite configurar el número de nodos en función de la carga de trabajo.

Los siguientes componentes se despliegan dentro del clúster de Kubernetes con el fin de dar soporte a la plataforma OSCAR:

- MinIO, un almacén de objetos de altas prestaciones que provee una API compatible con S3.
- OpenFaaS [28], una plataforma que permite la ejecución de funciones como servicio mediante llamadas HTTP.
- OSCAR manager, la aplicación principal responsable de la gestión de servicios y la integración de los diferentes componentes para dar soporte a la computación sin servidor y dirigida por eventos para el procesado de ficheros.

Como proveedores de datos externos, dentro de OSCAR se pueden usar:

- MinIO
- Amazon S3
- Onedata

La herramienta gráfica desarrollada en el presente trabajo fin de máster se encarga de generar un fichero YAML que describe el flujo de trabajo dirigido por eventos y que podrá ser utilizado tanto para SCAR como para OSCAR. Por comodidad, a lo largo del documento indicaremos que será SCAR la herramienta que principalmente consumirá dichos documentos YAML, pero será posible también utilizar OSCAR, pues ambos soportan el mismo modelo de computación. De hecho, es posible crear workflows híbridos que combinen el uso de funciones de SCAR y de OSCAR.

Workflows

Un workflow es una sucesión de trabajos interrelacionados, donde el comienzo de un nuevo trabajo puede tener una dependencia del anterior.

La mayoría de productos software para la automatización de workflows proporcionan una herramienta gráfica que facilita la creación sin necesidad de codificar. La combinación de *drag-and-drop* y formularios resulta idónea para resolver esta casuística.

ProcessMaker [29], es un claro ejemplo de herramienta de código abierto para la creación de flujos de trabajo de forma gráfica, empleando un interfaz WYSIWYG (what you see is what you get – lo que ves es lo que obtienes).

En el presente proyecto se pretende automatizar la forma en la que el usuario realiza un flujo de trabajo para definir un workflow de funciones para SCAR. Mediante una sencilla interfaz gráfica el usuario definirá el flujo de trabajo, indicando proveedores de almacenamiento de entrada y salida y las funciones a ejecutar.

Los siguientes subapartados identifican y justifican las tecnologías utilizadas para el desarrollo de la aplicación.

Desarrollo Frontend

Nuestro objetivo es crear una web estática, una página web que se entrega al navegador del usuario exactamente como está almacenada. Así, los ficheros serán servidos a través GitHub Pages [30]. GitHub Pages permite servir cualquier página web estática directamente desde un repositorio GitHub [31]. Para ello requerimos de una tecnología que permita el renderizado del lado del cliente (Client Side Rendering – CSR).

Al enfrentarnos al desarrollo de una interfaz gráfica mediante CSR no podemos pasar por alto los siguientes tres frameworks ampliamente utilizados por la mayoría de los proyectos actuales: Angular [32], Vue.js [33] y React [34]. Estas herramientas están disponibles a través del sistema de paquetes npm [35] disponible por defecto en Node.js [36].

Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.

Angular

Angular es un framework para aplicaciones web desarrollado en TypeScript [37], de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página.

Vue.js

Vue.js es un framework de JavaScript de código abierto para la construcción de interfaces de usuario y aplicaciones de una sola página. Es una librería ligera, altamente desacoplada de otras librerías, incluyendo lo esencial para permitir a los desarrolladores crear interfaces de usuario.

React

React es una biblioteca Javascript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de aplicaciones en una sola página. Al igual que Vue.js es una librería ligera para la construcción de interfaces de usuario. Está mantenida por Facebook y la comunidad de software libre. El proyecto cuenta con una comunidad de más de mil desarrolladores.

Cualquiera de las tres opciones podría ser una buena opción para el desarrollo del proyecto, puesto que todas proveen de renderizado del lado del cliente, sin embargo cabe destacar que Vue.js y React utilizan ambas el denominado Document Object Model (DOM) virtual que consigue una mayor tasa de actualización de la vista.

Antes de seleccionar una de estas librerías, vamos a hacer un análisis de las librerías construidas alrededor de estos frameworks que puedan sernos de utilidad para el desarrollo de nuestra herramienta.

Librerías Frontend para la composición de flujos de trabajo

Se ha realizado un estudio de las librerías disponibles de código abierto para la generación de flujos de trabajo con el objetivo de seleccionar la más apropiada. Para ello, se han realizado búsquedas en npm y GitHub utilizado las palabras clave “Javascript Workflow”, “Javascript Flowchart, “React workflow builder”, “Vuejs workflow builder”, “React flowchart builder” y “Vuejs flowchart builder”.

Se han descartado las librerías sin documentación, descontinuadas o que no han tenido desarrollo activo desde hace más de tres años.

A continuación, se listan las librerías que más se ajustan al objetivo del proyecto:

Moqui Workflow Designer

Moqui Workflow designer [38] es una librería de código abierto bajo licencia Creative Commons CC0 1.0, basada en Vue.js, para la creación de flujos de trabajo. Podemos ver un ejemplo de un flujo de trabajo realizado mediante esta herramienta en Ilustración 3.

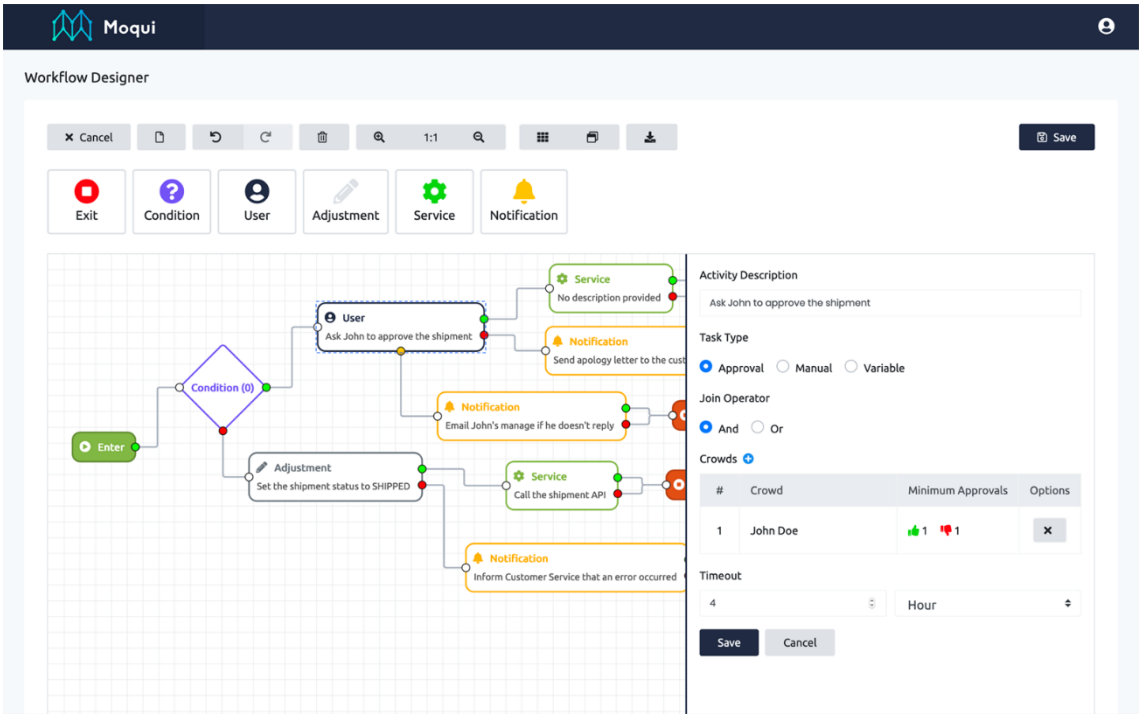

Ilustración 3. Interfaz Moqui Workflow Designer

Ventajas que aporta la utilización de esta librería:

- Gran cantidad de componentes visuales.
- Desarrollada en una librería Frontend actual (Vue.js).

Inconvenientes que involucran la utilización de esta librería:

- Librería no documentada.
- Última actualización el 28 de marzo de 2019.
- Dependencia con REST API Moqui, que imposibilita el despliegue de una web estática.

Vue Simple Flowchart

Vue Simple Flowchart [39] es una librería de código abierto bajo licencia MIT, bajada en Vue.js para la creación de flujogramas. Podemos ver un ejemplo de un flujo de trabajo realizado mediante esta herramienta en la Ilustración 4.

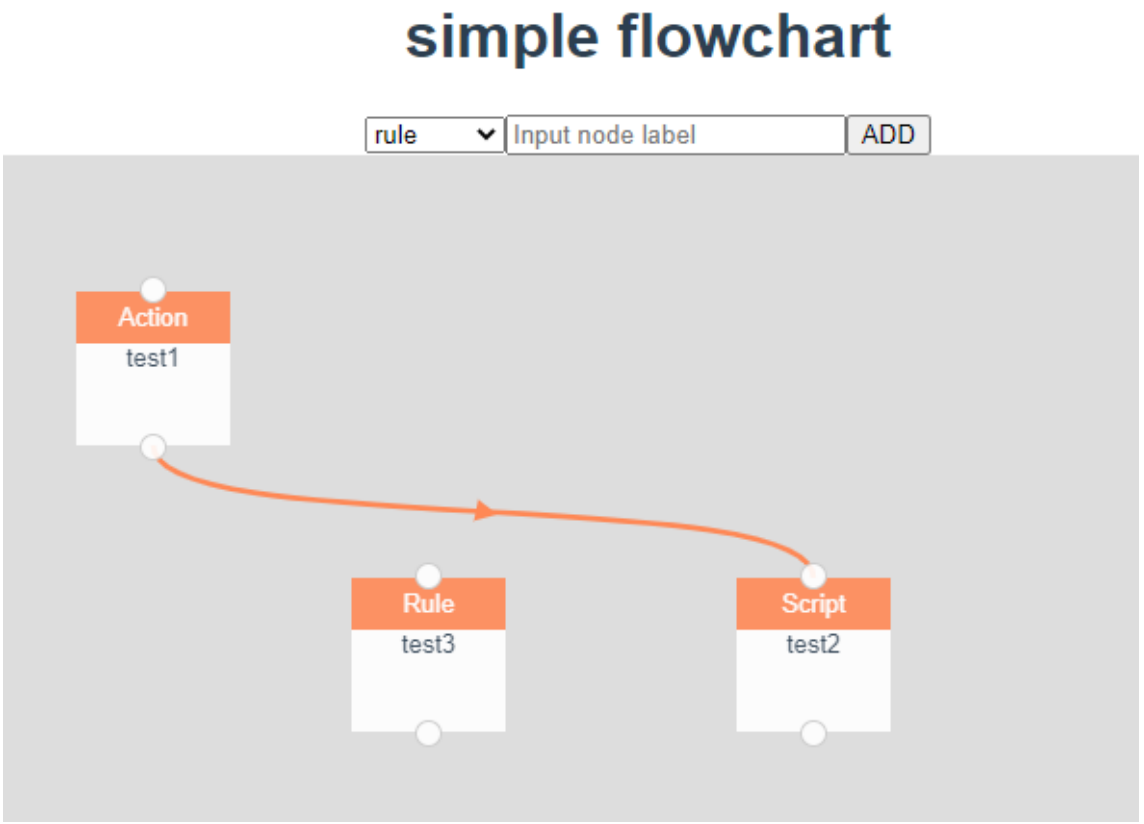

Ilustración 4. Vue Simple Flowchart

Ventajas que aporta la utilización de esta librería:

- Desarrollada en una librería Frontend actual (Vue.js).
- Incluye demostración [40].

Inconvenientes:

- Última actualización el 14 de agosto de 2019.
- Librería no continuada.
- Poca documentación.

Flowchart Vue

Flowchart Vue [41] es una librería de código abierto bajo licencia MIT, bajada en Vue.js para la creación de flujogramas. Podemos ver un ejemplo de un flujo de trabajo realizado mediante esta herramienta en la Ilustración 5.

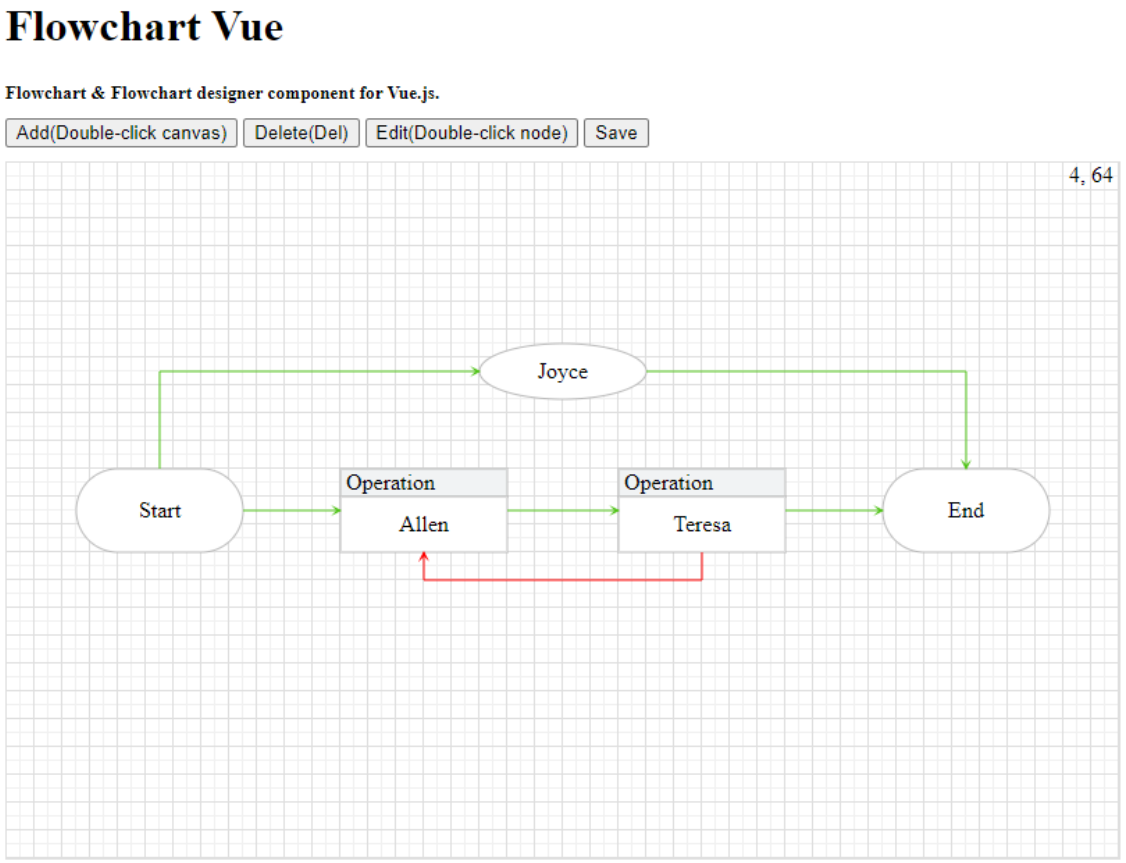

Ilustración 5. Flowchart Vue

Ventajas que aporta la utilización de esta librería:

- Desarrollada en una librería Frontend actual (Vue.js).
- Incluye demostración [42]
- Última actualización reciente (14 de marzo de 2021).

Inconvenientes:

- Desarrollada en JavaScript que a diferencia de TypeScript no tiene los conceptos de tipos e interfaces, lo que lo hace menos robusto.
- No incluye tests.
- Interfaz rudimentaria, requiere de gran inversión en tiempo para la personalización de los componentes.

React Flow Chart

React Flow Chart [43] es una librería de código abierto bajo licencia MIT, basada en React para la creación de flujogramas. Podemos ver un ejemplo de un flujo de trabajo realizado mediante esta herramienta en la Ilustración 6.

Ventajas que aporta la utilización de esta librería:

- Desarrollada en una librería Frontend actual (React).
- Incluye demostración mediante historias de usuario [44].
- Última actualización reciente (28 de junio de 2020).
- Incluye tests del código fuente.
- Incluye tipados (TypeScript)
- Alta capacidad de personalización.
- Sin estado.
- Permite drag-and-drop desde el menú lateral.
- Permite edición del nodo desde el menú lateral.

Inconvenientes:

- Problemas de renderizado al utilizar componentes funcionales [45]

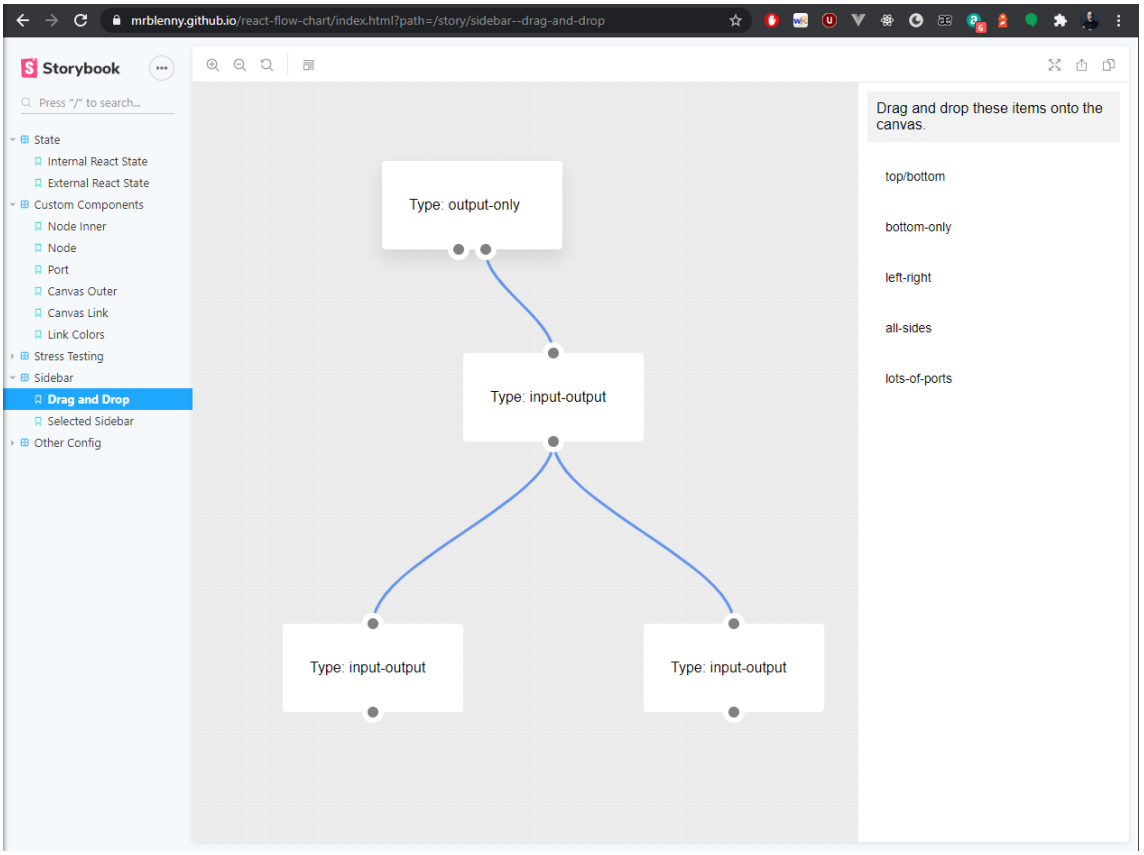

Ilustración 6. Storybook React Flow Chart

Por la alta capacidad de realizar *drag-and-drop* desde un menú lateral y la facilidad de personalización (estilización de los nodos), características que no tiene el resto de librerías, nos hemos decantado por React Flow Chart para la realización de nuestro proyecto.

Estado del arte

Actualmente podemos encontrar herramientas de libre distribución para composición de flujos de función como servicio como Faas-flow [46] una herramienta de composición de funciones para OpenFaaS. Sin embargo estas no ofrecen una interfaz gráfica para su desarrollo.

Sí que podemos encontrar herramientas privativas como AWS Step Functions [47] que permiten la creación visual de flujos de trabajo.

En la Ilustración 7 podemos ver un ejemplo de un flujo de trabajo diseñado en esta herramienta.

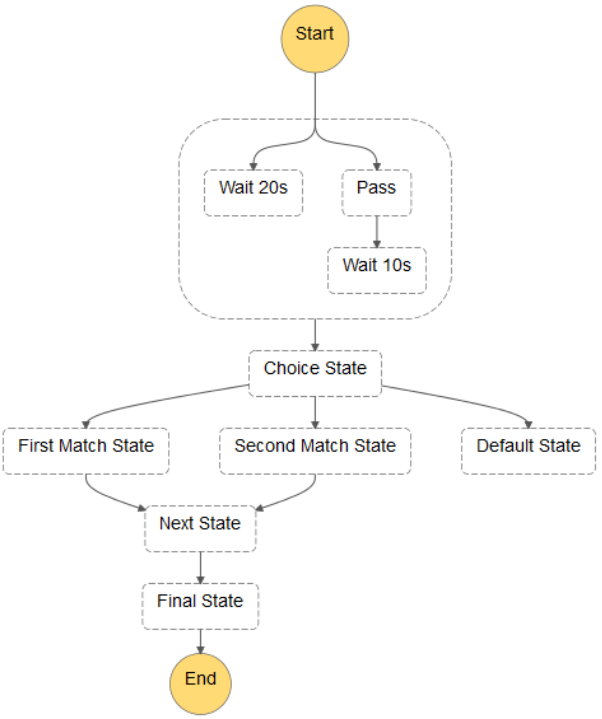

Ilustración 7. Flujo de trabajo con AWS Step Functions

Azure Logic Apps [48], permite la creación de flujos de trabajo, creando una aplicación lógica, agregando un desencadenador y una acción a la aplicación lógica. En la Ilustración 8 podemos ver un ejemplo de una aplicación lógica que

comprueba regularmente una fuente de RSS y envía una notificación por correo electrónico para los nuevos elementos.

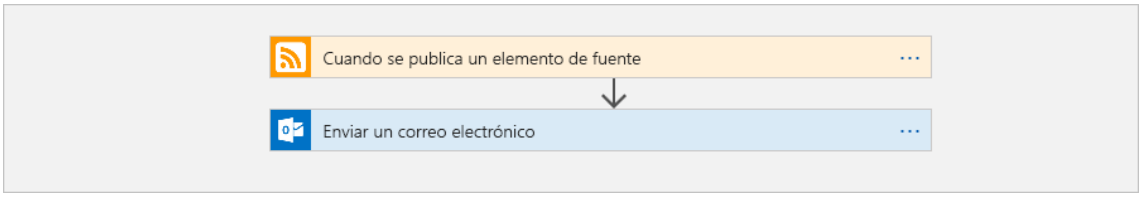

Ilustración 8. Azure Logic App

Son muchas las herramientas que permiten la automatización de tareas mediante flujos de trabajo. Por ejemplo, Slack [49] permite la creación de flujos de trabajo mediante activadores [50], cómo pueden ser la entrada de un nuevo miembro al canal, la reacción de un emoji, la programación por fecha y hora o mediante Webhooks.

Las librerías expuestas anteriormente permiten casos de uso muy delimitados y ligados a sus plataformas. Nuestro objetivo es desarrollar una herramienta de código abierto, extensible y publicable como una web estática para definir flujos de trabajo para el framework SCAR sin depender de ninguna plataforma.

3. Desarrollo

Diseño

Como hemos mencionado anteriormente se pretende facilitar la forma en la que el usuario realiza la definición de workflows serverless con la herramienta SCAR. Mediante una sencilla interfaz gráfica el usuario definirá el flujo de trabajo, indicando proveedores de almacenamiento de entrada y salida, así como las funciones a ejecutar. Para ello, haremos uso de herramientas gráficas actuales con el fin de obtener mejores resultados.

El flujo de trabajo se define mediante un fichero, tal y como se observa en la Ilustración 9, donde podemos diferenciar dos apartados principales: “functions” y “storage_providers”.

En el primer apartado podemos ver las claves “aws” y “oscar”, dentro de los cuales se definen las características de cada función, como son el script a ejecutar, los recursos o la imagen Docker utilizada, y se establecen los proveedores de datos tanto de entrada (input) como de salida (output).

En el segundo apartado podemos encontrar las claves “s3”, “onedata”, y “MinIO” que donde se establecen los distintos proveedores de datos que se utilizarán en el flujo de trabajo, y sus características propias como los tokens de autenticación.

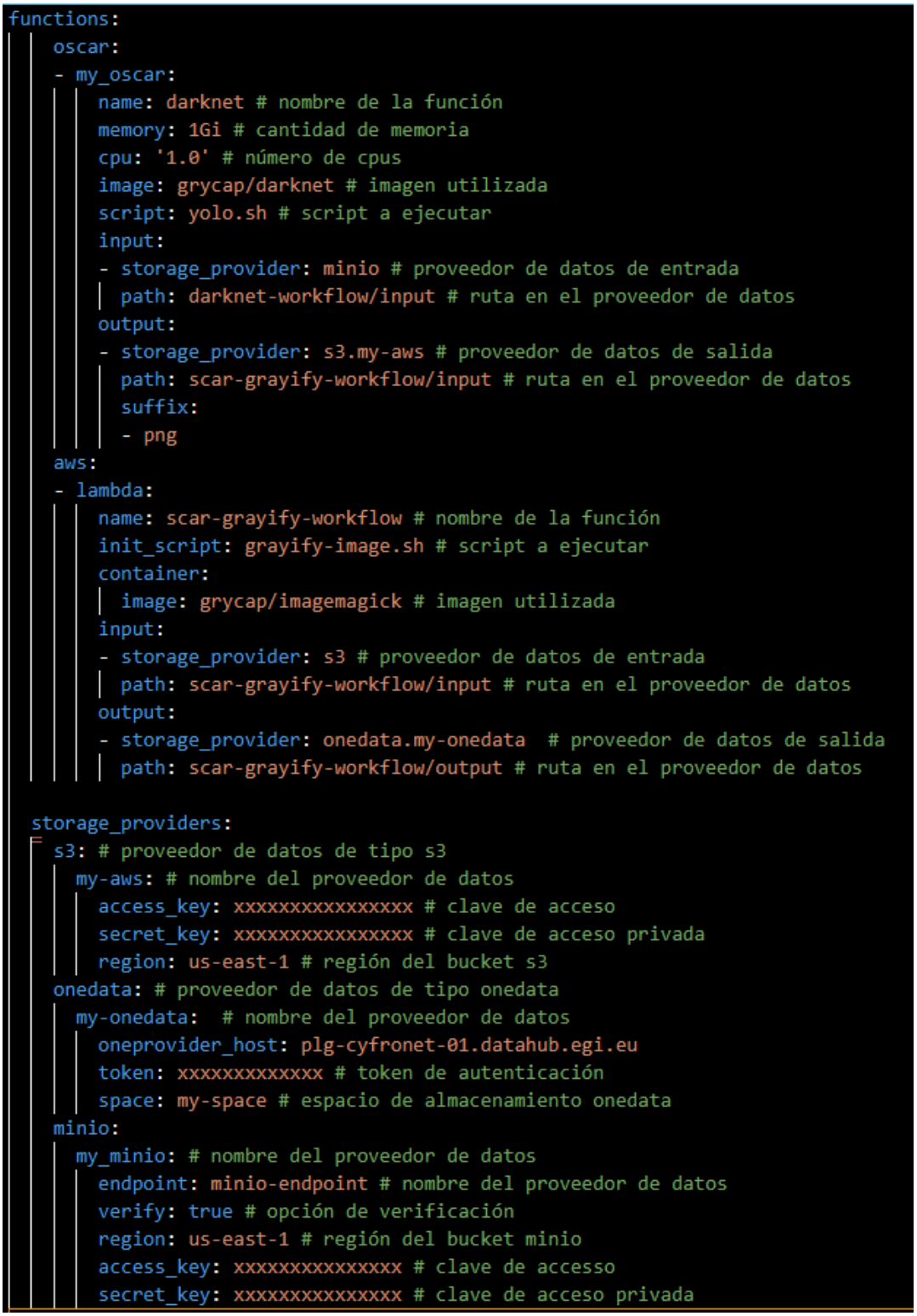

Ilustración 9. Definición del workflow en YAML

El objetivo es definir los elementos de este fichero (proveedores de datos y funciones) en forma de cajas, y la interacción entre estas cajas definirán las entradas y salidas. En Ilustración 10 podemos ver un ejemplo de cómo sería el flujo.

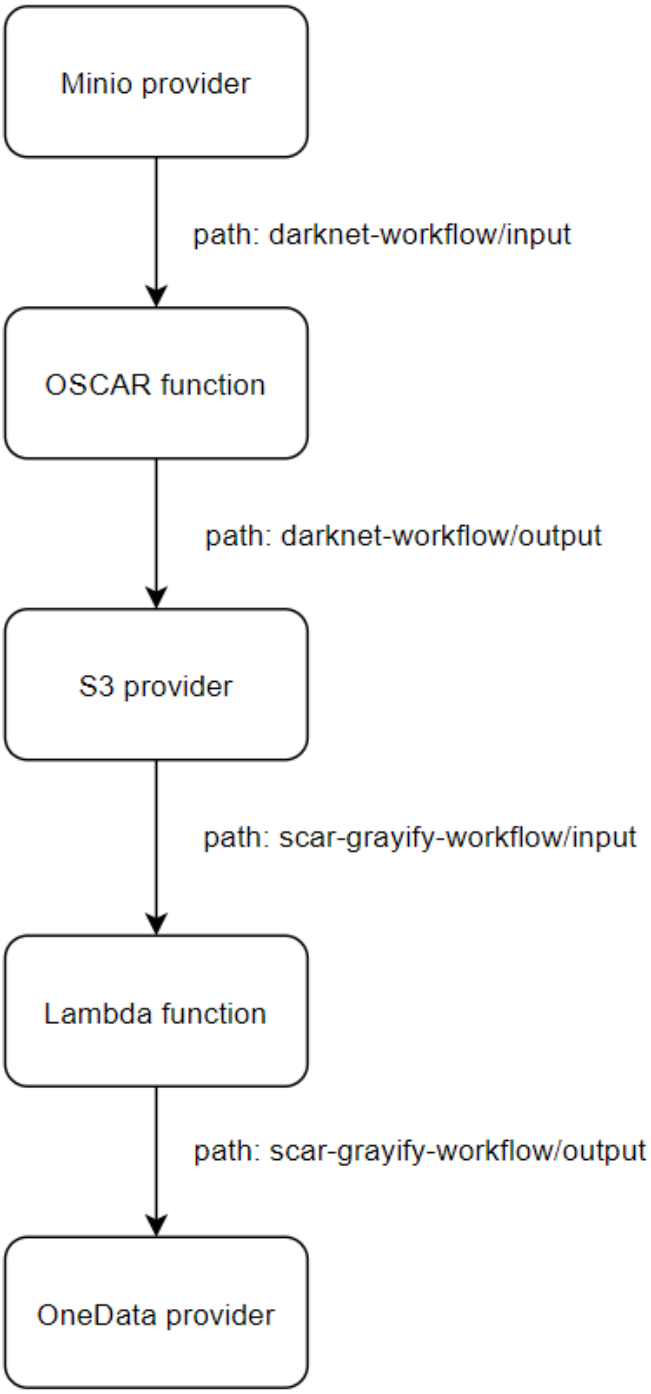

Ilustración 10. Flujo de trabajo simple.

Este flujo de trabajo realiza las siguientes acciones: al subir un fichero a un bucket MinIO en el path seleccionado se desencadena una función OSCAR. La ejecución de la función OSCAR produce un nuevo fichero que se almacena en un bucket S3 en el path seleccionado. Esta subida al bucket S3 desencadena una función Lambda, cuya ejecución generará un fichero que finalmente se almacena en un bucket Onedata.

Librerías seleccionadas

Una vez analizado el estado del arte en las librerías que nos facilitan el desarrollo del proyecto concluimos que la librería de desarrollo Frontend que mejor se adapta a nuestras necesidades es React.

Utilizaremos TypeScript como lenguaje de programación dado que nos provee de un tipado que nos proporcionará una mayor robustez a nuestro código frente a JavaScript.

La composición de flujos se realizará utilizando la librería mencionada en el apartado anterior: React Flow Chart.

El desarrollo de la interfaz se hará apoyándose en la librería Ant Design [51] que provee un conjunto de componentes: botones, formularios, etc., que nos facilitarán la construcción de la interfaz.

La conversión entre el formato de objeto JavaScript a YAML se realizará mediante la librería js-yaml [52], una librería de código abierto específica para la conversión de objetos en JavaScript a YAML con más de 28 millones de descargas semanales, que nos simplifica el proceso exportación a YAML.

Implementación

Un flujo de trabajo se caracteriza por la composición entre proveedores de datos: S3, MinIO y OneData y ejecución de funciones: SCAR (AWS Lambda y AWS Batch) y OSCAR. La relación entre cada proveedor de datos y cada ejecución de función requiere de una entrada o salida. La definición de este flujo de forma visual requiere definir estos elementos. Para ello nos valdremos de un menú para añadir proveedores de datos, mediante un formulario, un espacio de dibujo y una barra lateral que contenga los elementos a añadir al espacio de dibujo mediante Drag-and-drop.

Una vez añadidos los elementos al espacio de dibujo se podrán editar sus propiedades para completar los parámetros necesarios para la ejecución del flujo de trabajo.

El estado actual de aplicación podrá ser guardado y recuperado en cualquier momento.

Definición de proveedores de datos

La definición de los proveedores de datos se realizará mediante un formulario. Contaremos con un menú desplegable “Storage providers” que desplegará las siguientes opciones:

· New S3: Al interaccionar con esta opción se mostrará un formulario con los campos: Name, Region, Access Key y Secret Key. Estos campos indican el nombre, la región y las claves de usuario que deben disponer de privilegios de acceso al servicio S3, y son necesarios para la utilización de este elemento en el flujo de trabajo. En la Ilustración 11 podemos ver este formulario. Al rellenar esta información y pulsar sobre el botón OK se añadirá un elemento proveedor de datos de tipo AWS S3 a la barra lateral, para su posterior uso en la composición de flujos de trabajo. En la Ilustración 12 podemos ver cómo se mostraría un proveedor de datos Amazon S3 nombrado MyS3.

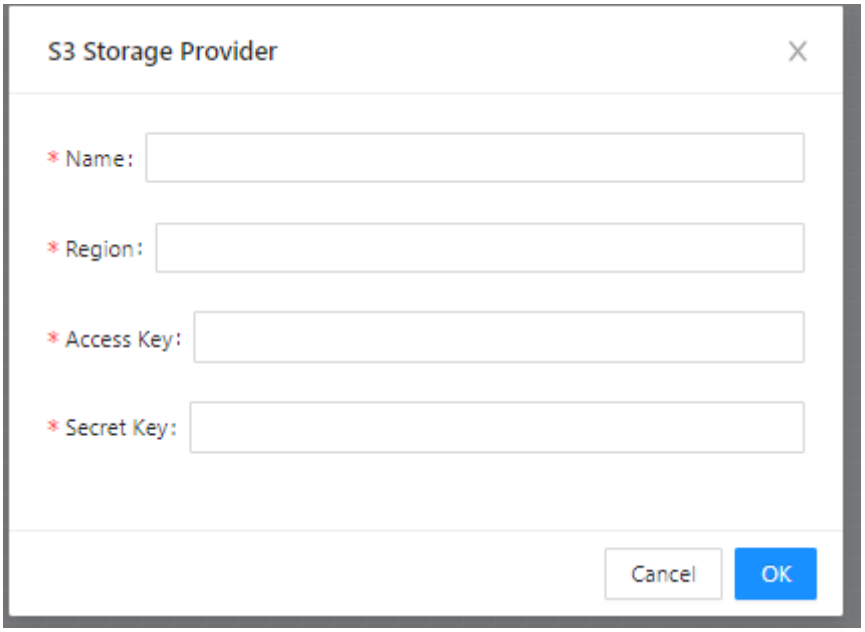

Ilustración 11. Formulario S3

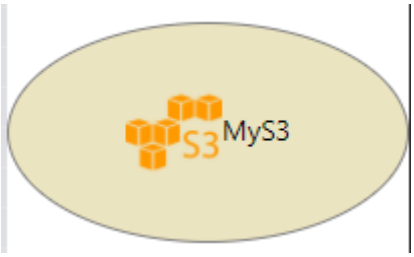

Ilustración 12. Proveedor de datos S3

· New MinIO: Al interaccionar con esta opción se mostrará un formulario con los campos: Name, Endpoint, Verify, Region, Access Key y Secret Key. Estos campos indican el nombre, la región y las claves del servicio MinIO, y son necesarios para la utilización de este elemento en el flujo de trabajo. El *checkbox* Verify permite activar o desactivar la comprobación de certificados SSL. En la Ilustración 13 podemos ver este formulario. Al rellenar esta información y pulsar sobre el botón OK se añadirá un elemento proveedor datos de tipo MinIO a la barra lateral, para su posterior uso en la composición de flujos de trabajo. En la Ilustración 14 podemos ver cómo se mostraría un proveedor de datos MinIO nombrado MyMinIO.

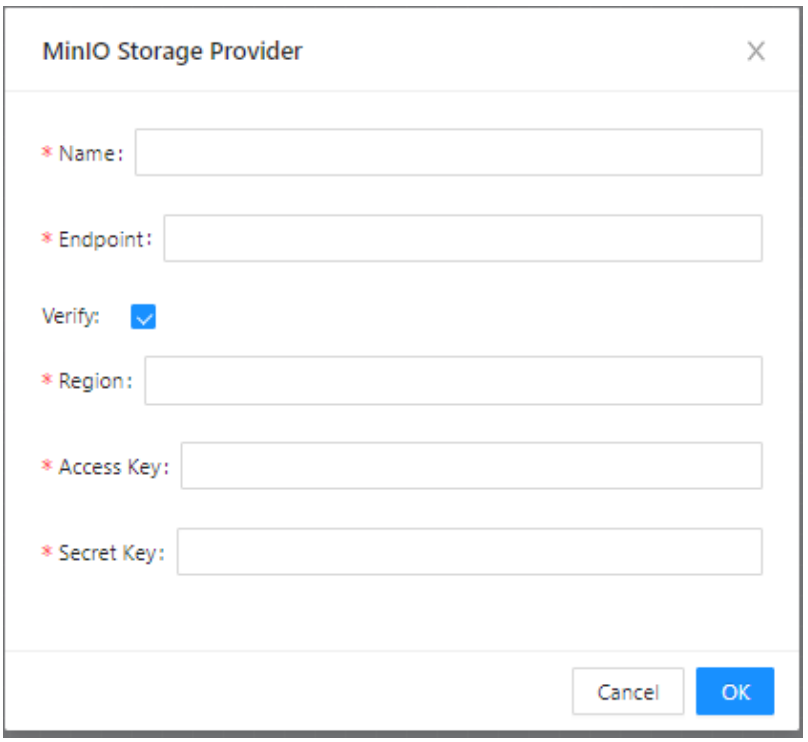

Ilustración 13. Formulario MinIO

Ilustración 14. Proveedor de datos MinIO

· New Onedata: Al interaccionar con esta opción se mostrará un formulario con los campos: Name, One Provider Host, Token y Space. Estos campos indican el nombre, el host, el espacio y la clave del servicio Onedata, y son necesarios para la utilización de este elemento en el flujo de trabajo. En la Ilustración 15 podemos ver este formulario. Al rellenar esta información y pulsar sobre el botón OK se añadirá un elemento proveedor datos de tipo Onedata a la barra lateral, para su posterior uso en la composición de flujos de trabajo. En la Ilustración 16 podemos ver cómo se mostraría un proveedor de datos Onedata nombrado MyOneData.

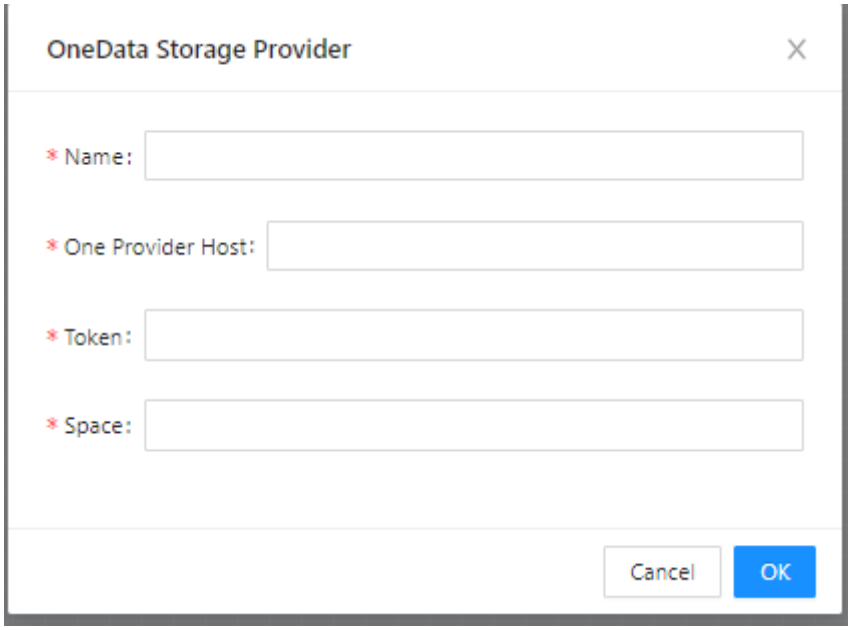

Ilustración 15. Formulario Onedata

Ilustración 16. Proveedor de datos Onedata

Definición de ejecución de funciones

La barra lateral contará con dos elementos para la definición de funciones, identificados por los logos de OSCAR y AWS como podemos ver en las Ilustraciones 17 y 18.

Ilustración 17. Función OSCAR

Ilustración 18. Función AWS

Una vez arrastrado uno de los elementos al espacio de dibujo podemos editarlo mediante un formulario, que aparecerá haciendo doble click sobre el elemento.

En la Ilustración 19 podemos ver cómo se visualiza un elemento de tipo OSCAR Function al arrastrarlo al espacio de dibujo.

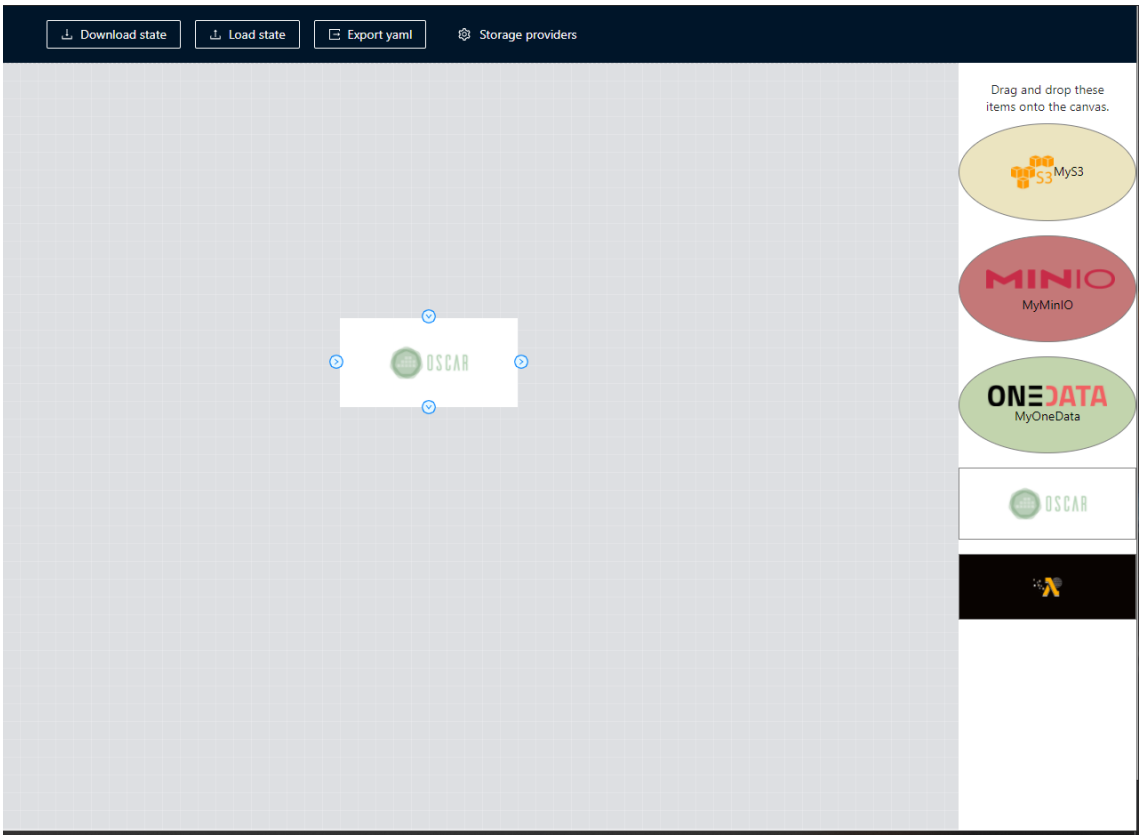

Ilustración 19. Visualización de un elemento OSCAR en el espacio de trabajo

Podemos configurar este elemento rellenando los campos obligatorios Name y Script, necesarios para poder incluir una función de este tipo en el flujo de trabajo, además de otros campos como Memory o CPU que definen los recursos

disponibles para la ejecución de la función. En los apartados Input y Output se definen los prefijos y sufijos que tienen los ficheros de entrada y salida de la función. En la Ilustración 20 podemos ver cómo se visualizará el formulario para realizar esta tarea.

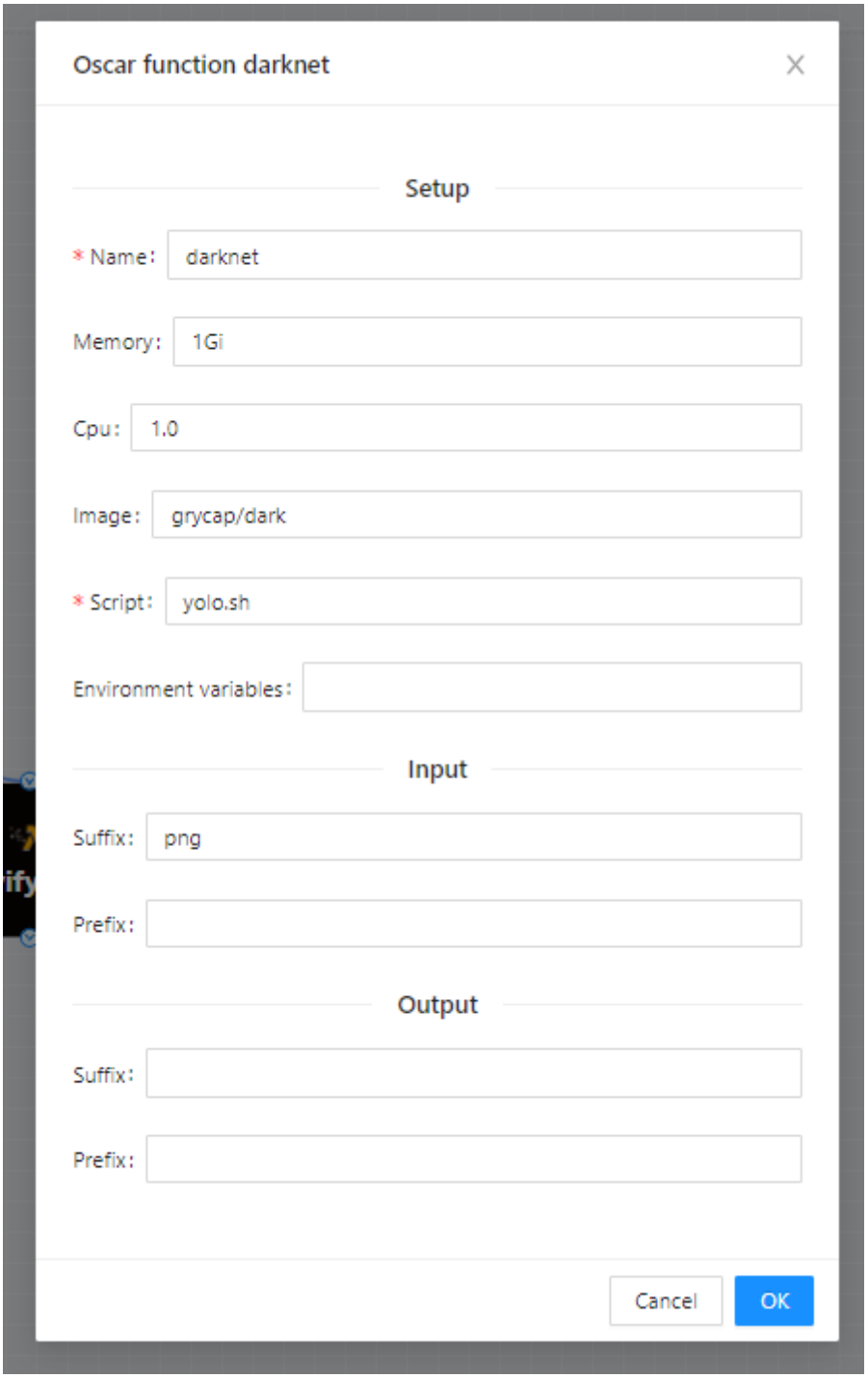

Ilustración 20. Formulario campos OSCAR

Tras editar el nombre de nuestra función OSCAR podremos ver reflejado en cambio en el elemento del espacio de trabajo, lo que nos ayudará a identificar el elemento, pues puede haber varias instancias de tipo OSCAR. En la Ilustración 21 podemos ver cómo se visualizará.

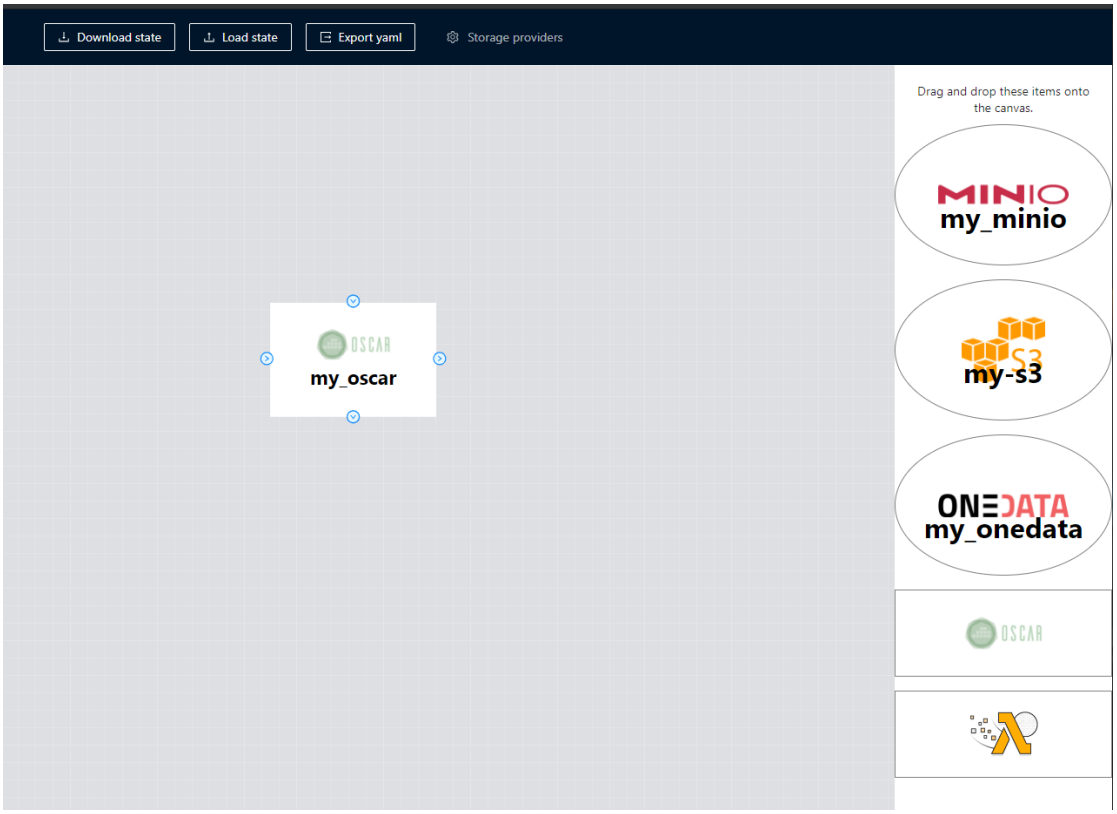

Ilustración 21. Visualización de un elemento OSCAR con nombre en el espacio de trabajo

En la Ilustración 22 podemos ver cómo se visualiza un elemento de tipo AWS al arrastrarlo al espacio de dibujo.

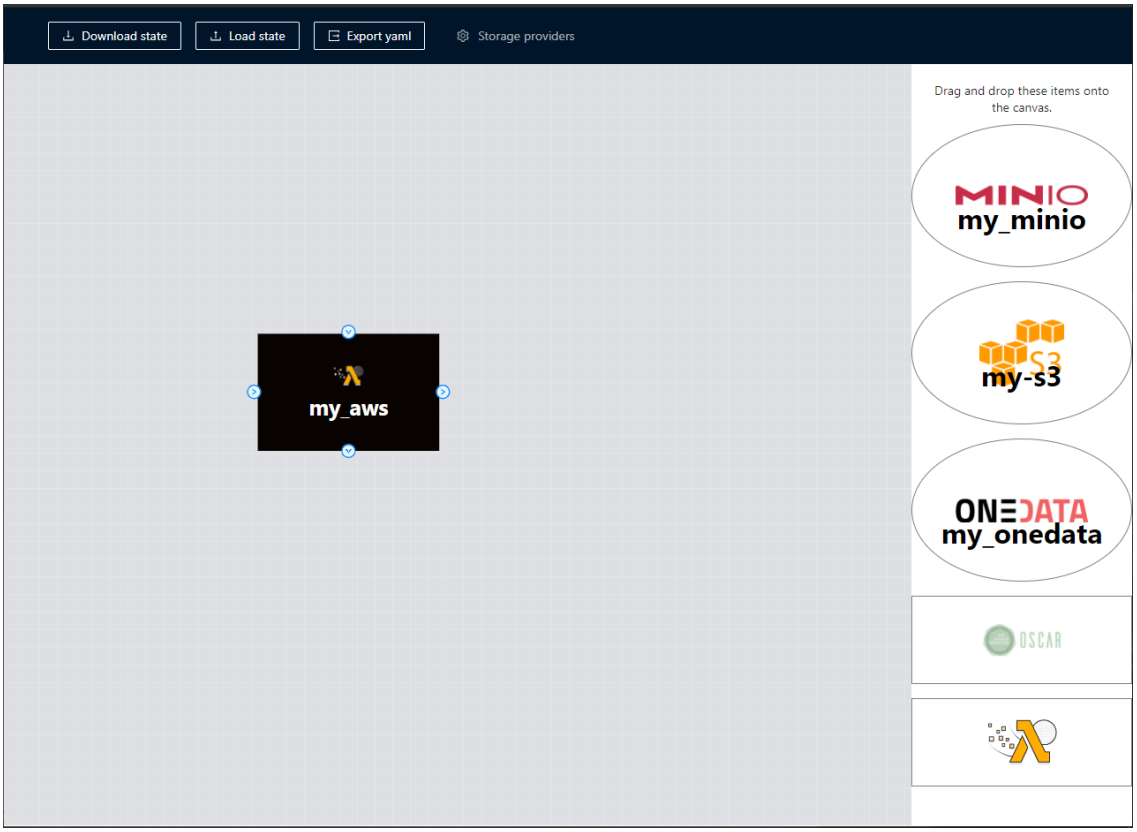

Ilustración 22. Visualización de un elemento AWS en el espacio de trabajo

Este elemento puede ser configurado en dos pestañas: Lambda y Batch.

En la pestaña Lambda se rellenan los campos Name, Region, Memory, Cpu, Image, Init Script, Boto profile, Timeout, Log level, Execution mode y Environment variables. Estos campos indican el nombre, la región, los recursos, la imagen base, el tiempo máximo de ejecución, las variables de entorno (definimos las variables de entorno relacionadas con el supervisor u otros componentes de AWS) y otras configuraciones específicas de Lambda. Además se rellenará la siguiente información relativa al contenedor: Image, Timeout threshold y Environment variables, los cuales indican la imagen del contenedor, el umbral de tiempo de espera y las variables de entorno (definimos las variables de entorno que deseamos tener disponibles dentro del contenedor). En la Ilustración 23 podemos ver cómo se visualizará el formulario para realizar esta tarea.

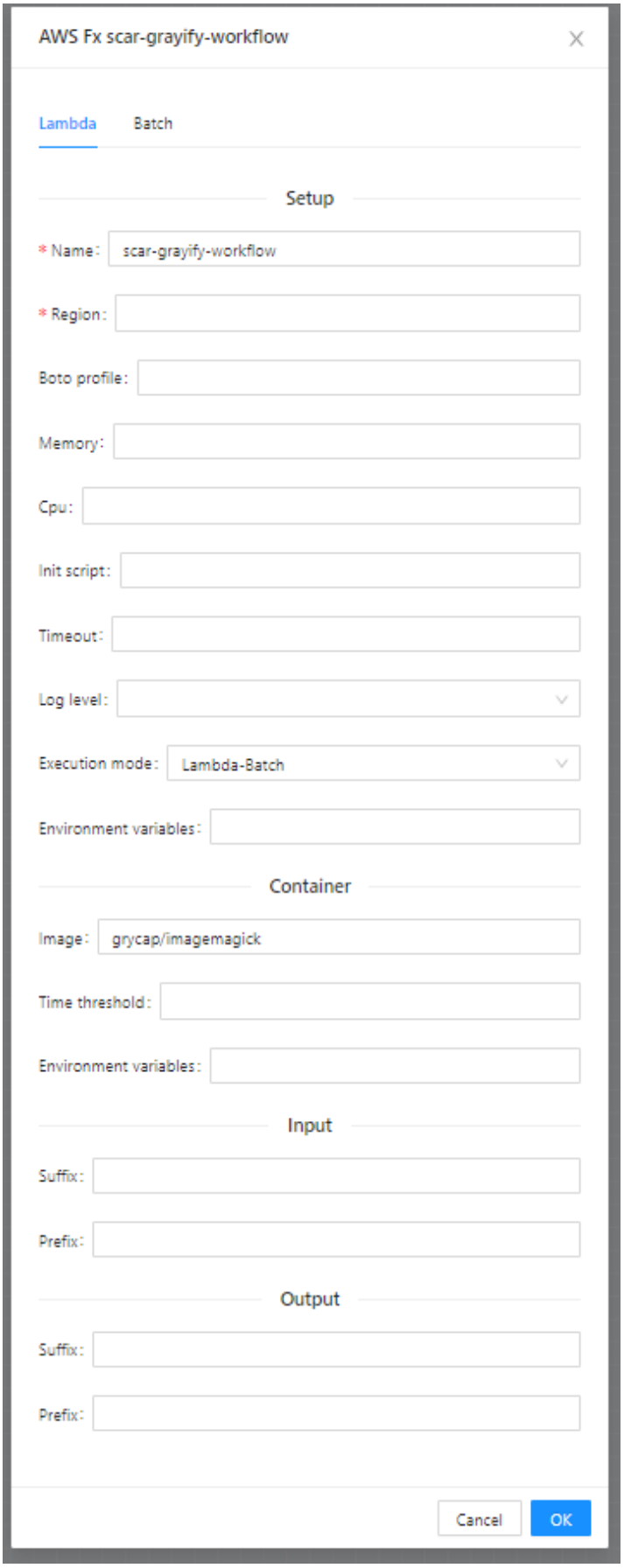

Ilustración 23. Formulario campos AWS Lambda

En la segunda pestaña “Batch” de esta ventana modal se rellenan los campos Region, Boto profile, Memory, vCPU, Enable GPU, Service role y Environment variables. Estos campos indican el nombre, la región, los recursos, las variables de entorno y otras configuraciones específicas de Batch. Además, se rellenan los campos de Compute resources Desired vCpus, Min vCpus, Max vCpus, Instance role, Security group ids, Subnets y Instance types, los cuales establecen los parámetros de elasticidad y grupos de seguridad. Esta pestaña solo ha de rellenarse en el caso que optemos por una ejecución por lotes mediante AWS Batch. En la Ilustración 24 podemos ver cómo se visualizará el formulario para realizar esta tarea.

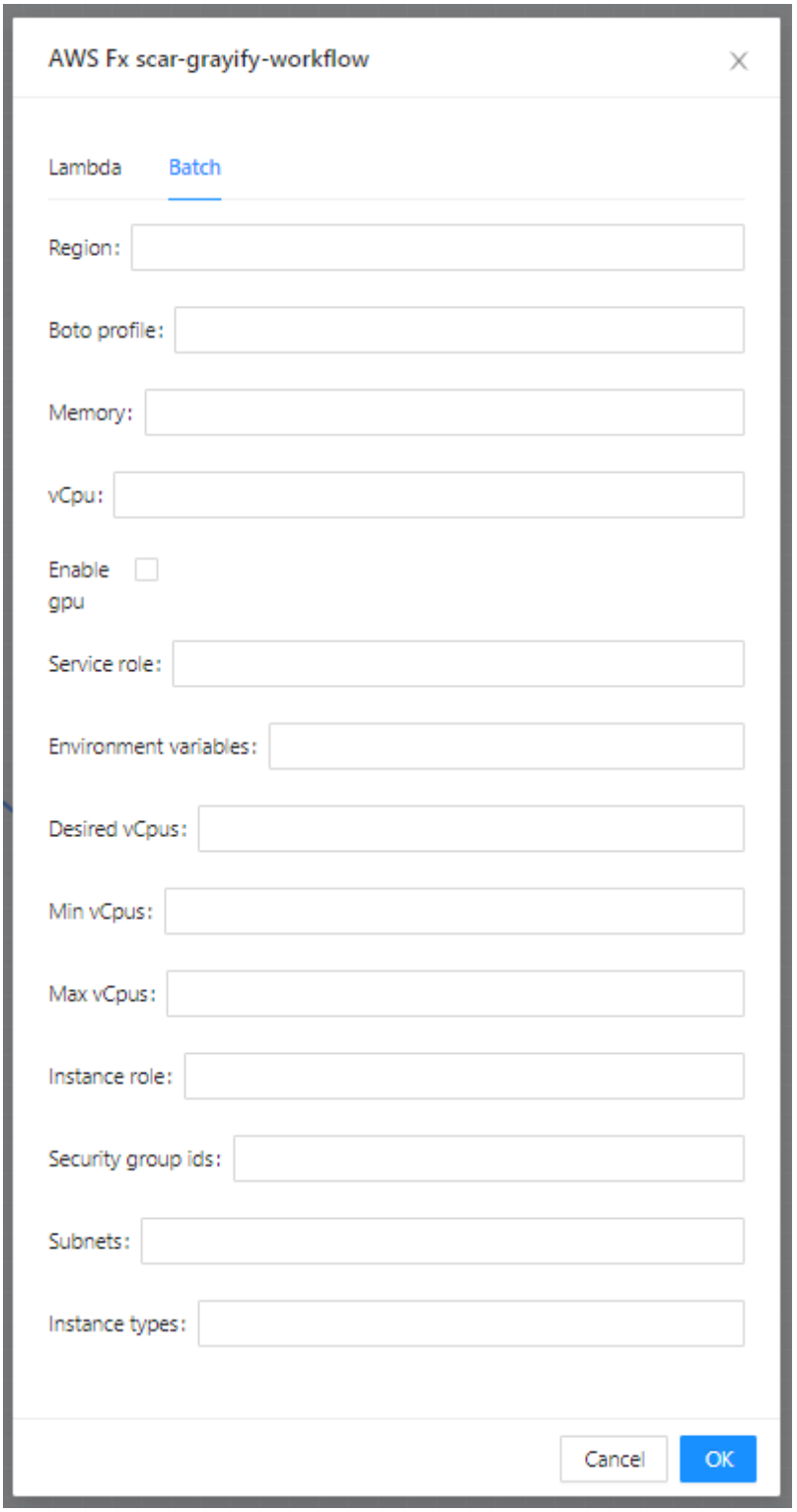

Ilustración 24. Formulario edición de campos AWS Batch

Tras editar el nombre de nuestra función Lambda podremos ver reflejado en cambio en el elemento del espacio de trabajo, lo que nos ayudará a identificar el elemento, pues puede haber varias instancias de tipo Lambda. En la Ilustración 25 podemos ver cómo se visualizará.

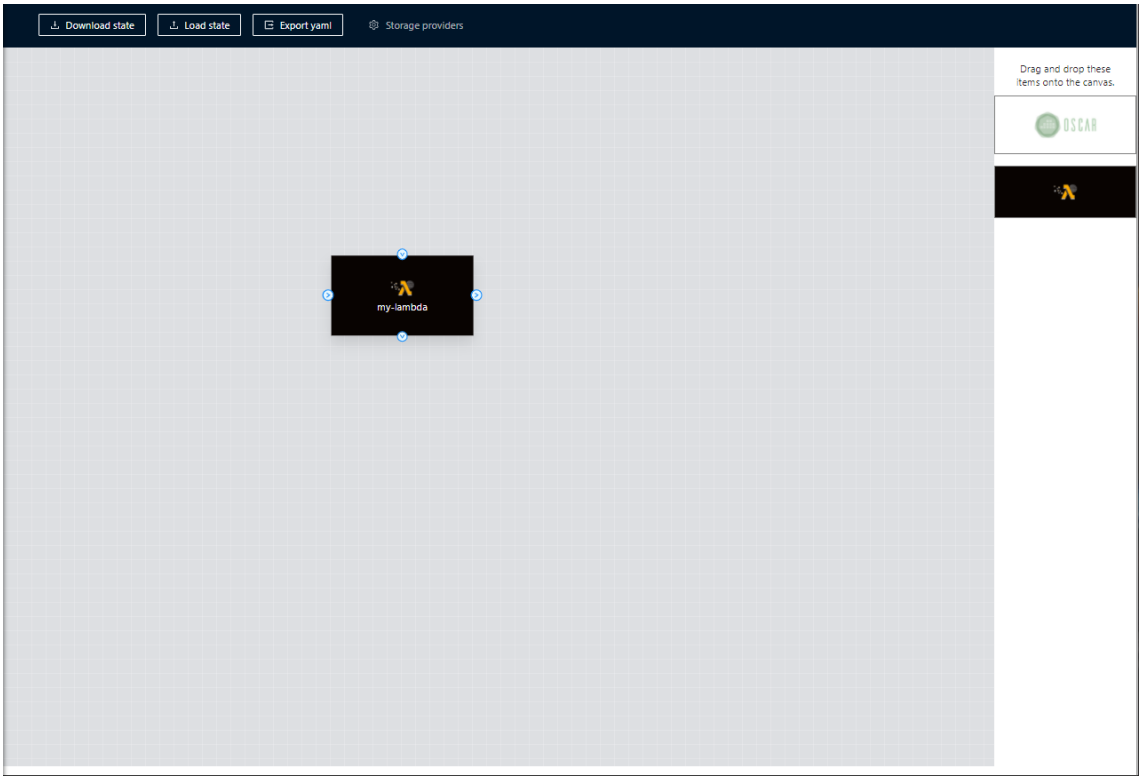

Ilustración 25. Visualización de un elemento AWS Lambda con nombre en el espacio de trabajo.

Una vez definidos los proveedores de datos y funciones a utilizar procederemos a la creación del flujo enlazando los diferentes elementos, a los que llamaremos *buckets* a partir de ahora. Cada función puede tener múltiples buckets de entrada. En la Ilustración 26 podemos ver cómo conectar un proveedor S3 a una función Lambda como entrada y un proveedor OneData como salida de la función.

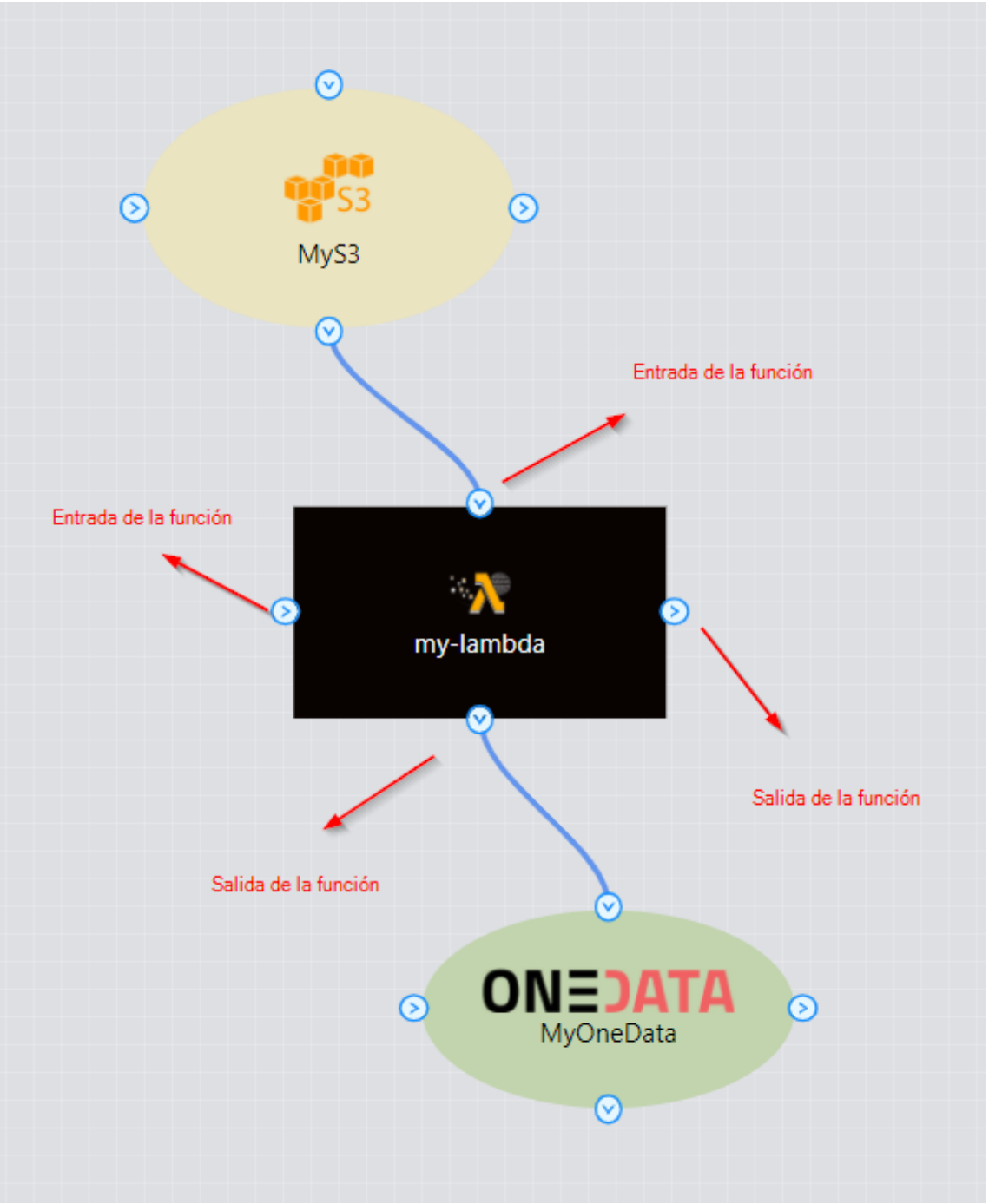

Ilustración 26. Entradas y salidas de una función

4. Casos de uso

En el presente capítulo se define un caso de uso de un *workflow* híbrido, con el fin de mostrar el funcionamiento de la herramienta gráfica desarrollada. En este *workflow* se utilizarán los proveedores de datos MinIO, Onedata y Amazon S3, y funciones de OSCAR y SCAR.

Supongamos que queremos definir un flujo de trabajo donde al subir una imagen jpg en color a un bucket MinIO desencadene la ejecución de una función OSCAR.

La función OSCAR generará una imagen en blanco y negro que se almacenará en un bucket S3. A su vez, al subir el fichero al bucket S3 se desencadenará la ejecución de una función Lambda. La función Lambda rotará la imagen 180º y finalmente se guardará el resultado en un bucket Onedata.

En la Ilustración 27 podemos ver el flujo gráficamente.

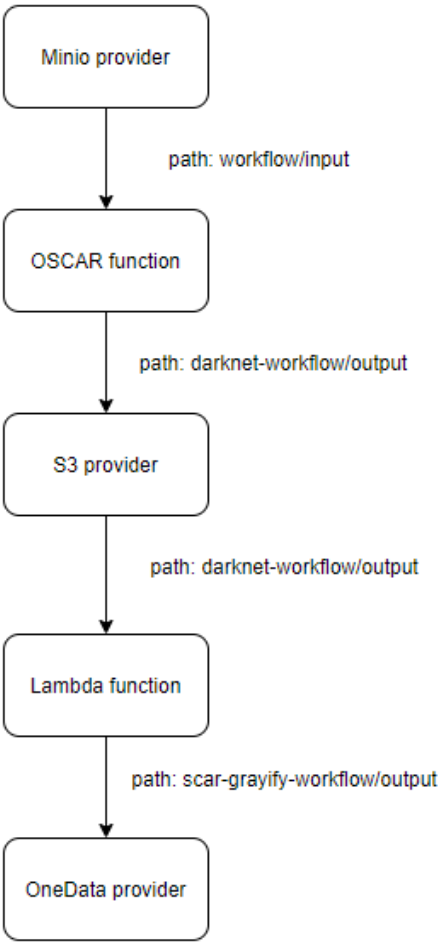

Ilustración 27. Flujo de trabajo

Comenzaremos definiendo un bucket para el proveedor MinIO pasando el cursor sobre Storage providers y haciendo click en "New MinIO", como podemos ver en la Ilustración 28.

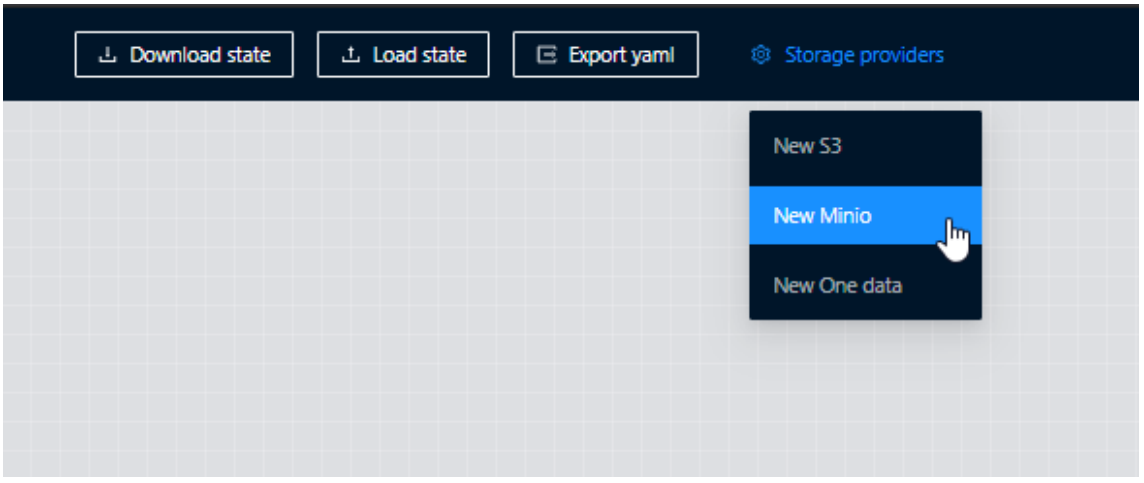

Ilustración 28. Creación bucket MinIO

A continuación, se abrirá una ventana modal con un formulario donde rellenaremos la información relativa al bucket, como podemos ver en la Ilustración 29.

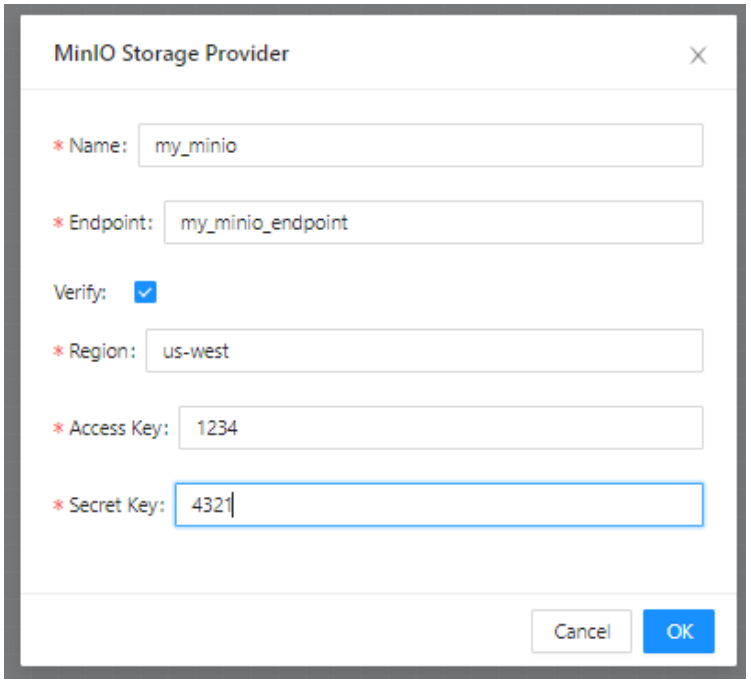

Ilustración 29. Modal formulario de creación bucket MinIO

Una vez introducidos los datos pulsaremos el botón “OK” y aparecerá un nuevo bucket MinIO en la barra lateral derecha de la aplicación como podemos ver en la Ilustración 30.

Ilustración 30. Panel lateral con bucket my_minio de tipo MinIO

El siguiente paso será añadir el nuevo elemento al área de trabajo. Para ello haremos click sobre la elipse my_minio y la arrastraremos hasta el área de trabajo (zona cuadriculada). Una vez arrastrado, nuestra área de trabajo quedará como en la Ilustración 31.

Ilustración 31. Área de trabajo con el elemento my_minio de tipo MinIO

Una vez añadido procederemos a editar su path mediante un formulario que se mostrará haciendo doble click en el elemento my_minio del área de trabajo. En la Ilustración 32 podemos ver el formulario:

Ilustración 32. Formulario edición path my_minio

Podemos editar o eliminar cualquier bucket en el momento que lo deseemos. Al pasar el cursor sobre el elemento en el panel lateral se mostrará un menú con las opciones "Editar", que abrirá el formulario con los datos introducidos anteriormente y "Eliminar" que borrará permanentemente el bucket. En la Ilustración 33 podemos ver el menú mencionado.

Ilustración 33. Menú bucket my_minio

El siguiente paso será añadir y editar una función OSCAR. Para ello procederemos a arrastrar el elemento de tipo OSCAR del panel lateral derecho al área de trabajo Ilustración 34.

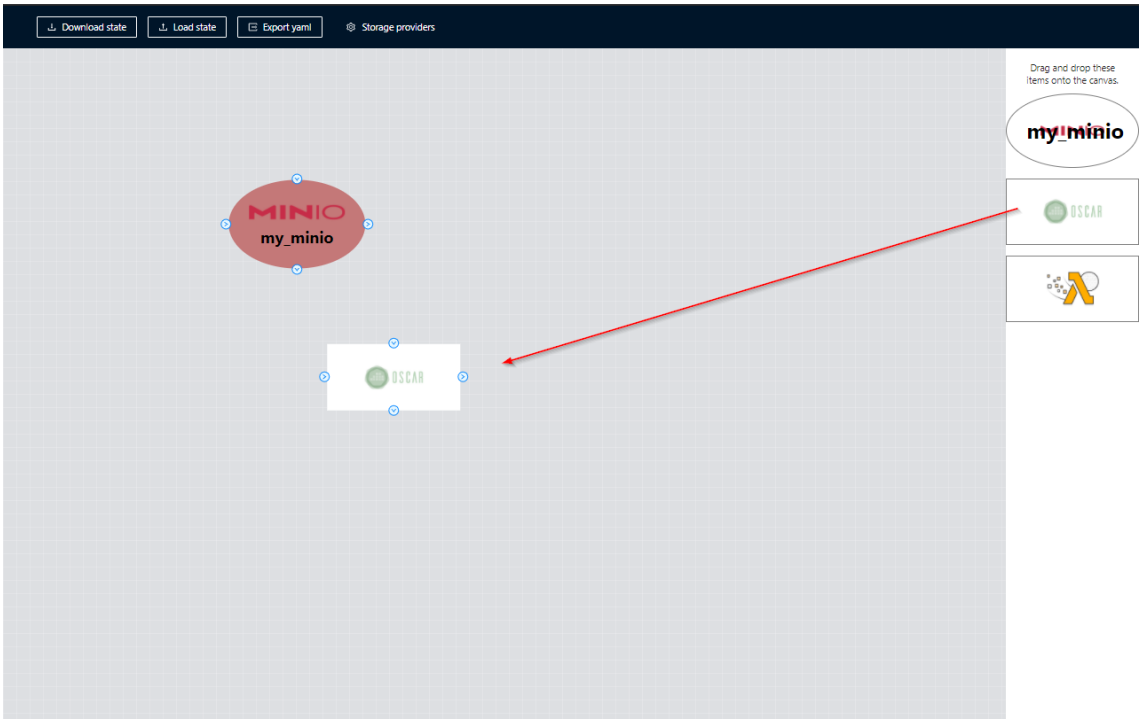

Ilustración 34. Añadiendo una función OSCAR

Una vez añadido editaremos sus valores mediante un formulario en un modal que se abrirá realizando doble click en el elemento OSCAR que se encuentra en el área de trabajo. En la Ilustración 35 podemos ver el resultado. Los campos marcados con un asterisco en rojo son los campos obligatorios. El resto de campos son opcionales.

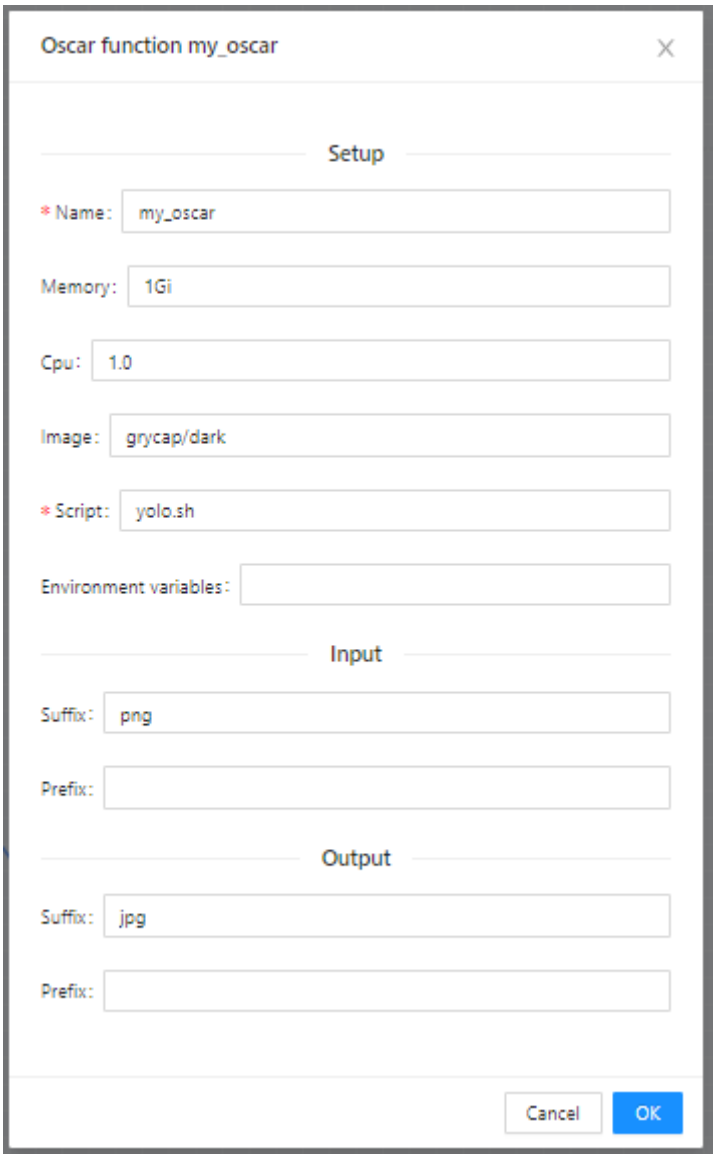

Ilustración 35. Definición función my_oscar

Tras haber añadido estos dos elementos procederemos a conectarlos. Dado que el bucket my_minio es un input a la función my_oscar conectaremos una de las salidas del elemento my_minio con una de las entradas del elemento my_oscar, como podemos ver en la Ilustración 36. Para ello haremos click sobre una de las

salidas del elemento my_minio y arrastraremos hasta la entrada el elemento my_oscar.

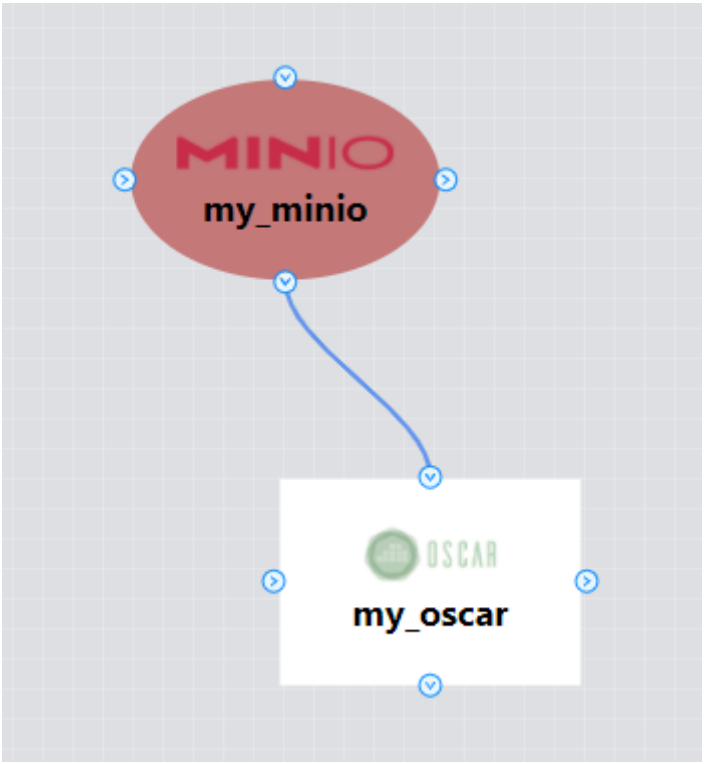

Ilustración 36. Elementos my_minio y my_oscar conectados

El siguiente paso será crear el bucket de tipo S3 para ello iremos al menú Storage provider > New S3 y rellenaremos el formulario como podemos ver en la Ilustración 37.

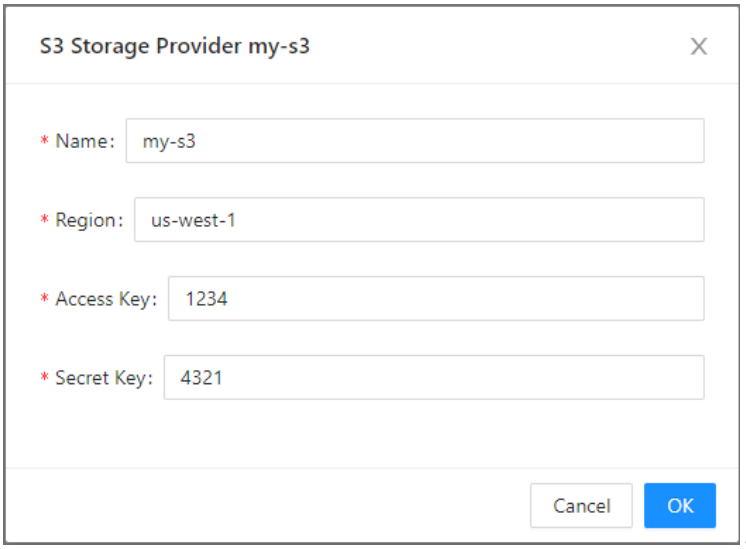

Ilustración 37. Formulario bucket my_s3

Una vez agregado, aparecerá el elemento my_s3 en el panel lateral derecho como podemos ver en la Ilustración 38.

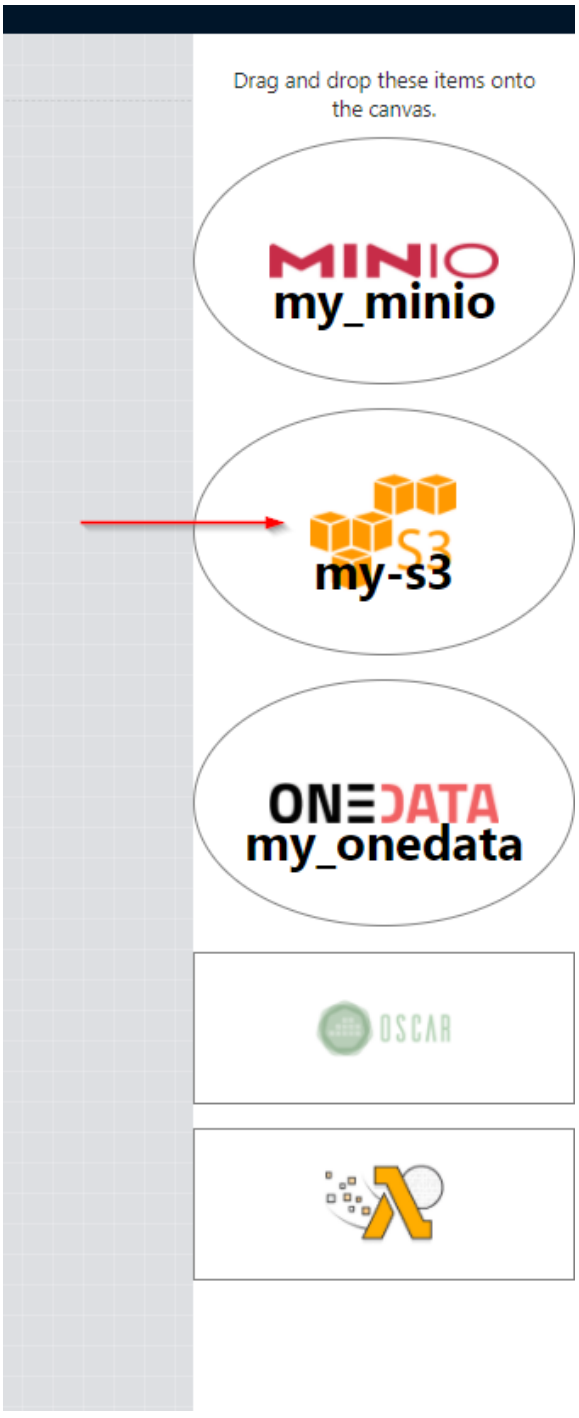

Ilustración 38. Bucket my_s3 de tipo S3

Arrastraremos el elemento al área de trabajo y editaremos el path, como vemos en la Ilustración 39, y, finalmente, procederemos a conectar la salida del elemento my_oscar a la entrada del elemento my_s3, quedando el workflow como podemos ver en la Ilustración 40.

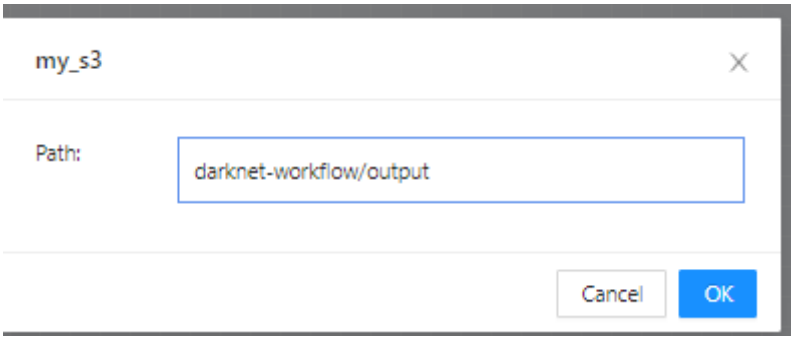

Ilustración 39. Formulario edición path bucket my_s3

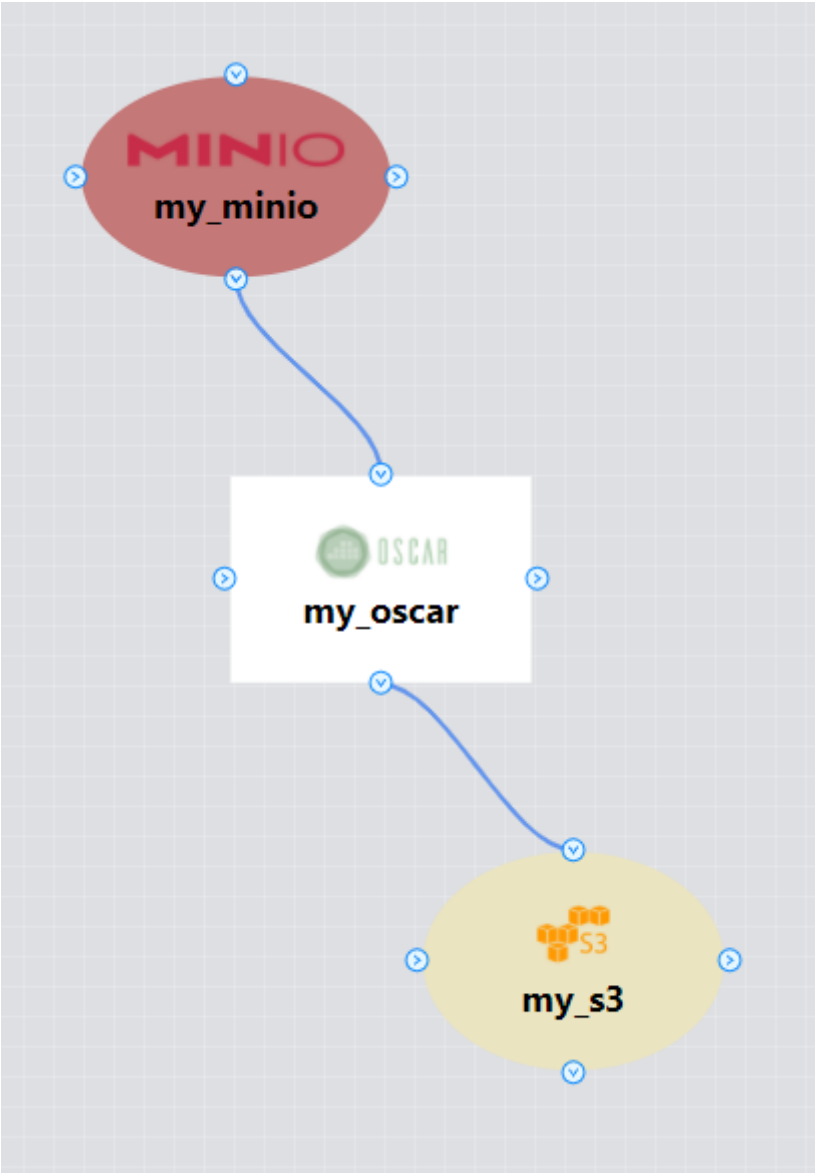

Ilustración 40. Flujo de trabajo my_minio > my_oscar > my_s3

El siguiente paso será añadir nuestra función Lambda. Para ello arrastraremos un elemento de tipo Lambda como podemos ver en la Ilustración 41.

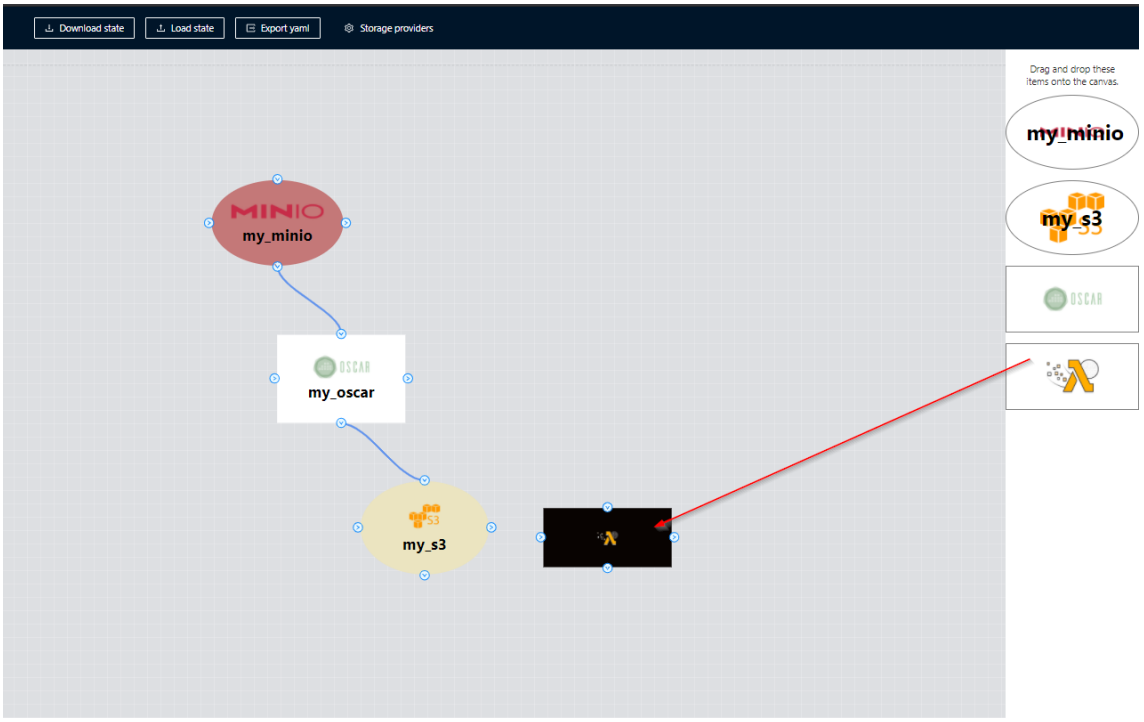

Ilustración 41. Añadir función Lambda

Una vez añadido procederemos a su edición mediante doble click. Rellenaremos el formulario como podemos ver en la Ilustración 42. Los campos marcados con un asterisco en rojo son los campos obligatorios. El resto de campos son opcionales.

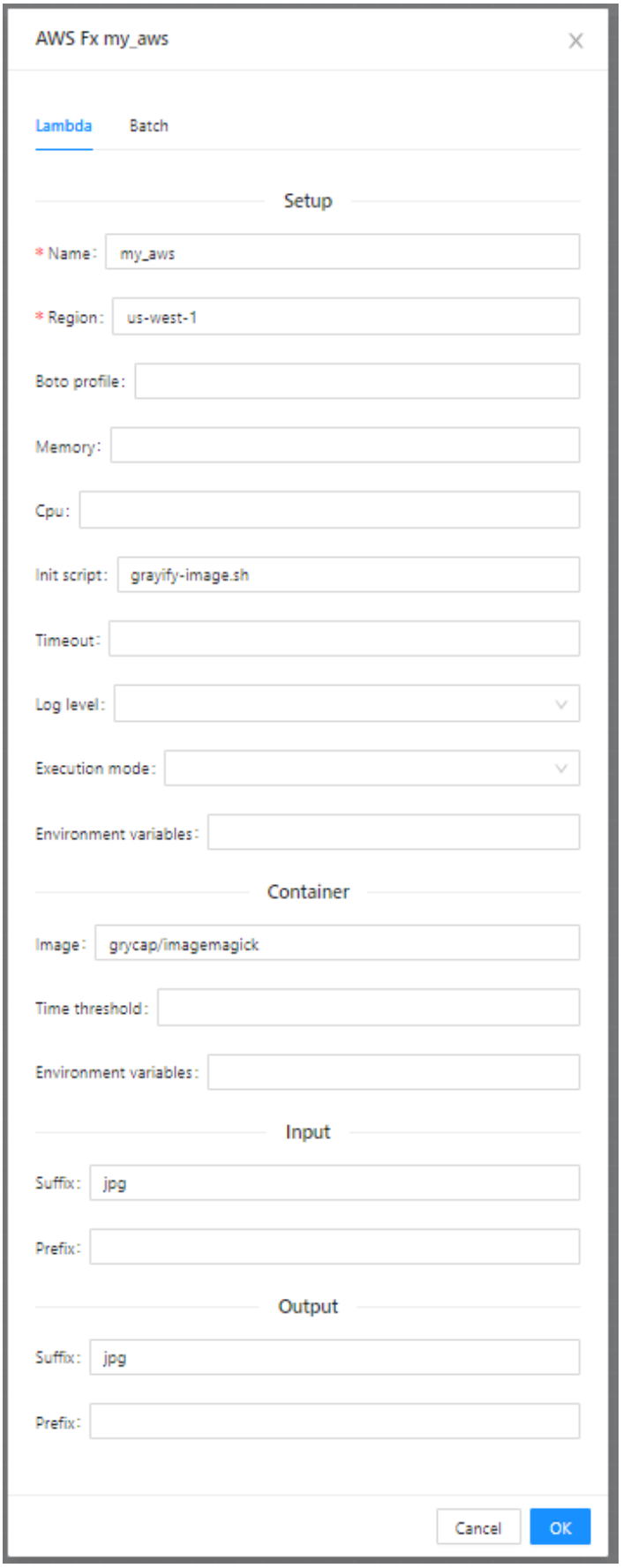

Ilustración 42. Formulario función Lambda

Una vez guardados los cambios conectaremos la salida del bucket S3 a la entrada de la función Lambda, quedando el flujo como podemos ver en la Ilustración 43.

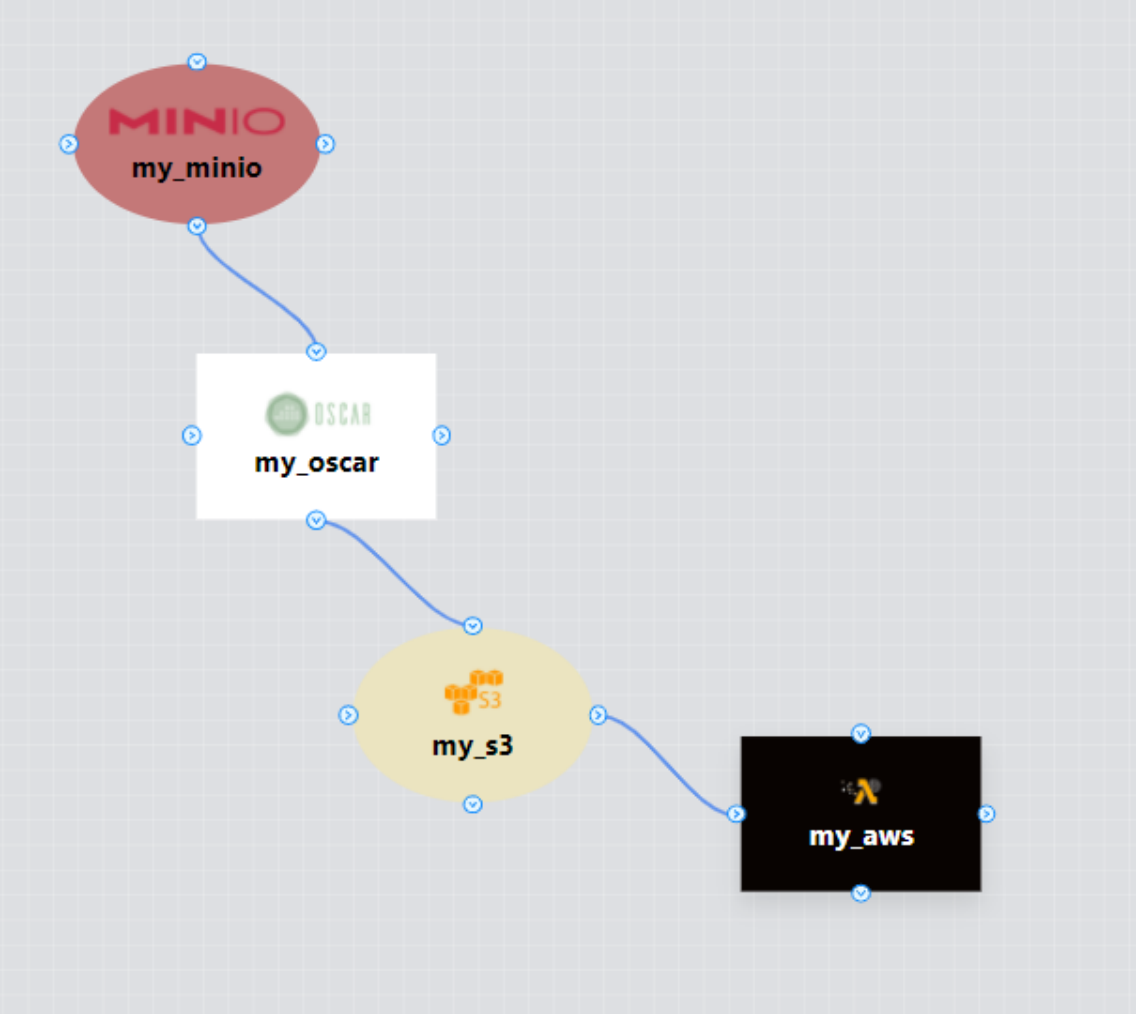

Ilustración 43. Flujo de trabajo my_minio > my_oscar > my_s3 > my_aws

El siguiente paso será añadir el último bucket, esta vez de tipo Onedata. Para ello volveremos al menú Storage providers y seleccionaremos la opción New Onedata. Posteriormente rellenaremos el formulario como podemos ver en la Ilustración 44.

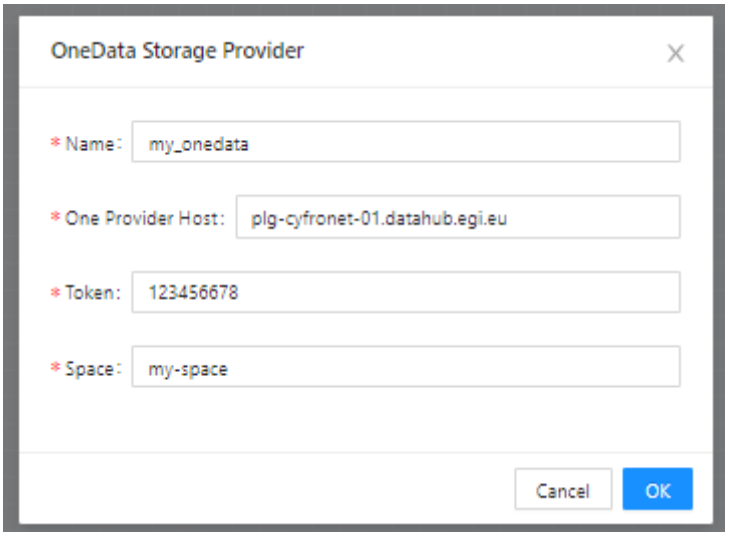

Ilustración 44. Formulario creación bucket my_onedata de tipo Onedata

Una vez agregado, aparecerá el elemento my_s3 en el panel lateral derecho como podemos ver en la Ilustración 45.

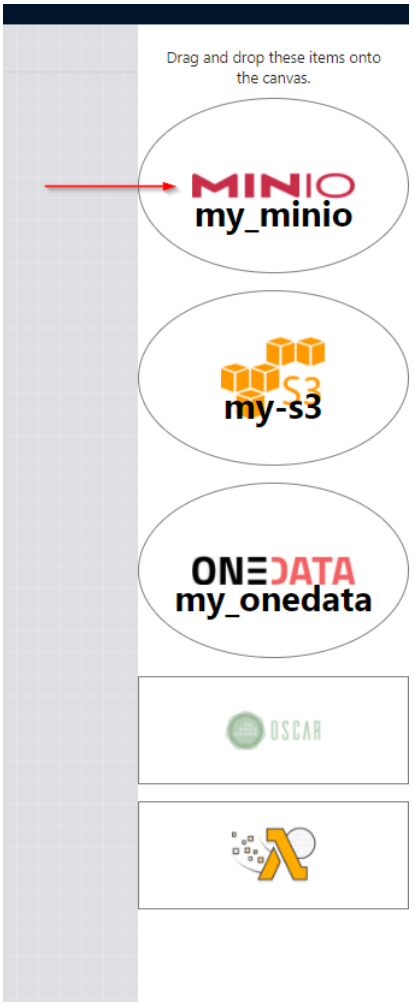

Ilustración 45. Bucket my_onedata de tipo onedata

El siguiente paso será añadir nuestra bucket my_onedata. Para ello lo arrastraremos al área de trabajo como podemos ver en la Ilustración 45.

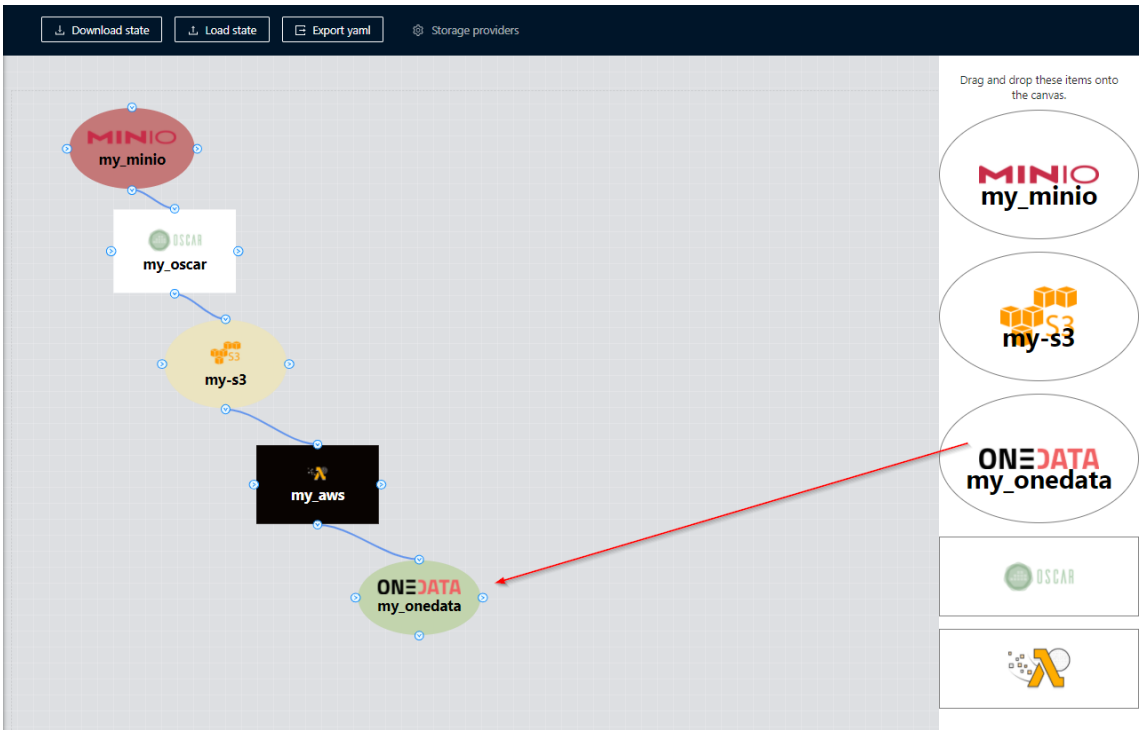

Ilustración 46. Añadir bucket my_onedata

A continuación, editaremos el path haciendo click en el elemento my_onedata del área de trabajo como podemos ver en la Ilustración 47.

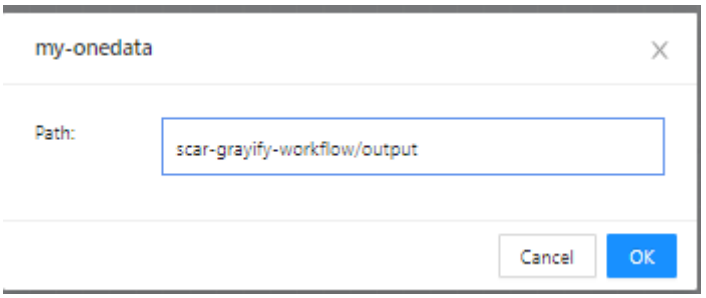

Ilustración 47. Formulario path my_onedata

Finalmente, completamos nuestro flujo de trabajo enlazando la salida de la función my_aws con la entrada del bucket my_onedata como podemos ver en la Ilustración 48.

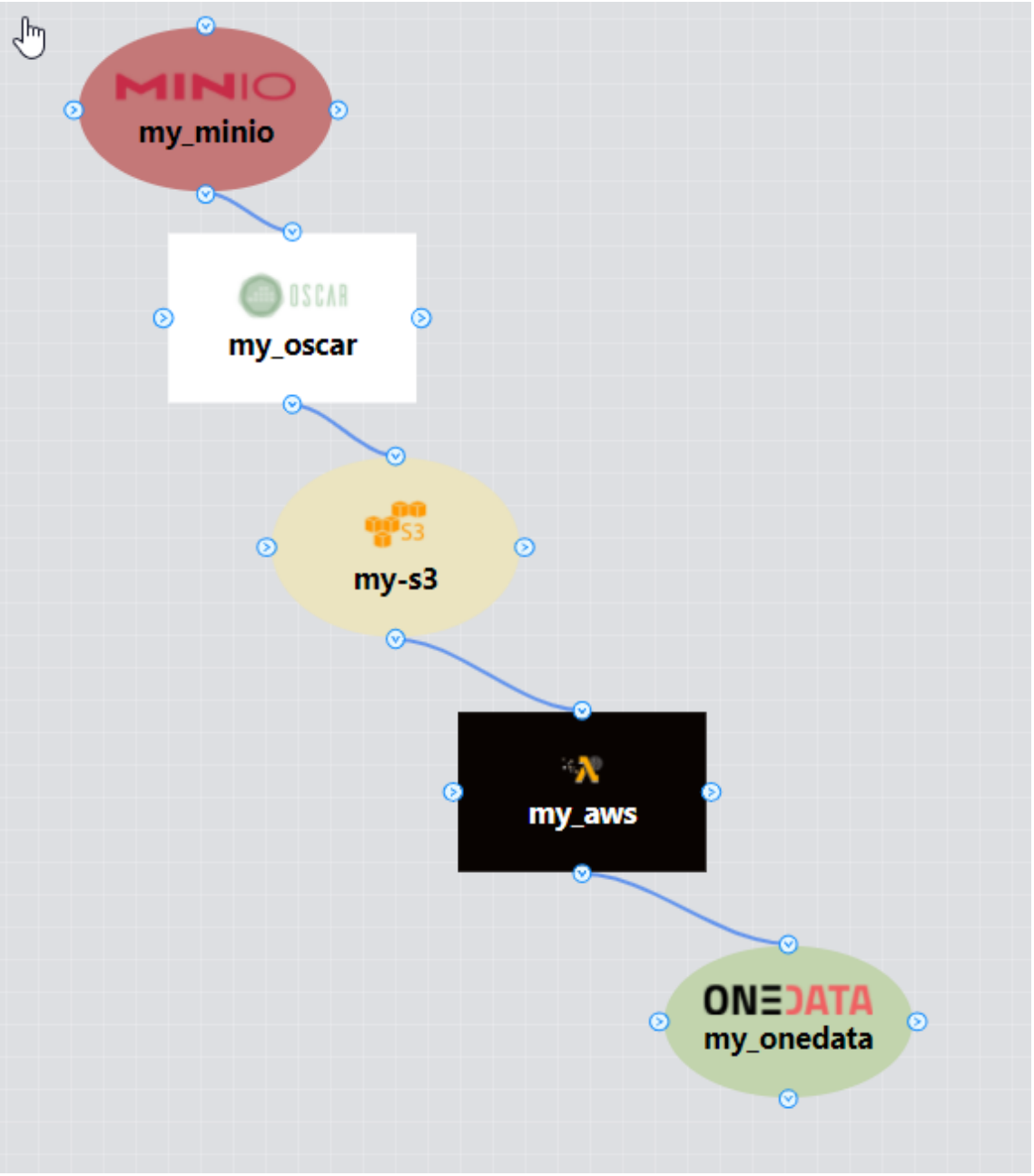

Ilustración 48. Flujo de trabajo my_minio > my_oscar > my_s3 > my_aws > my_onedata

Una vez establecido nuestro flujo de trabajo pulsaremos en botón “Export yaml” y se generará nuestro fichero “workflow.yaml”. Podemos ver el resultado en la Ilustración 49.

```yaml
functions:
  oscar:
    - my_oscar:
        name: my_oscar
        memory: 1Gi
        cpu: '1.0'
        image: grycap/dark
        script: yolo.sh
        input:
          - path: darknet-workflow/input
            storage_provider: minio.my_minio
        output:
          - path: darknet-workflow/output
            storage_provider: s3.my-s3
  aws:
    - lambda:
        name: my_aws
        region: us-west-1
        init_script: grayify-image.sh
        input:
          - suffix:
              - jpg
            path: darknet-workflow/output
            storage_provider: s3.my-s3
        output:
          - suffix:
              - jpg
            path: scar-grayify-workflow/output
            storage_provider: onedata.my_onedata
        container:
          image: grycap/imagemagick
storage_providers:
  s3:
    my-s3:
      region: us-west-1
      access_key: '1234'
      secret_key: '4321'
  onedata:
    my_onedata:
      name: my_onedata
      oneprovider_host: plg-cyfronet-01.datahub.egi.eu
      token: '12345678'
      space: my-space
  minio:
    my_minio:
      endpoint: minio-endpoint
      verify: true
      region: us-east-1
      access_key: '1234'
      secret_key: '4321'
```

Ilustración 49. workflow.yaml

Una vez generado el fichero YAML con el flujo de trabajo utilizaremos este fichero para proceder a la creación de los recursos en el cliente SCAR mediante el comando:

```
scar init -f workflow.yaml
```

5. Integración continua

La integración continua junto con la entrega continua son filosofías de desarrollo actuales que conducen automatizar la entrega de software.

El objetivo de la integración continua es establecer un camino automatizado y consistente para probar, construir y empaquetar aplicaciones. El despliegue continuo empieza justo después. Una vez ha finalizado la integración continua el despliegue continuo se encarga de desplegar la nueva aplicación al entorno que se haya preestablecido.

En la actualidad las grandes plataformas de repositorios git [53] ofrecen estos servicios para sus usuarios. Bitbucket [54] ofrece Bitbucket pipelines [55], Gitlab [56] ofrece Gitlab Continuous Integration (CI) [57] y GitHub ofrece GitHub Actions [58].

El código fuente de este proyecto se pueden encontrar íntegramente en el repositorio de GitHub del Grupo de Grid y Computación de Altas Prestaciones [59]. Dado que estamos utilizando la plataforma GitHub utilizaremos GitHub Actions para la integración continua.

La aplicación web resultante de este proyecto se puede encontrar publicada como una página de GitHub Pages [60].

El despliegue en GitHub Pages se ha establecido mediante integración continua. Cada vez que se añade un cambio sobre la rama main del repositorio se desencadena una GitHub Action, que se encarga de generar la aplicación y distribuirla a GitHub Pages.

La configuración de la GitHub Action se realiza mediante el lenguaje YAML, previa configuración de las claves privadas y públicas en el repositorio de GitHub de la herramienta. En la Ilustración 50 podemos ver el fichero ci.yml utilizado para crear el despliegue continuo a GitHub Pages.

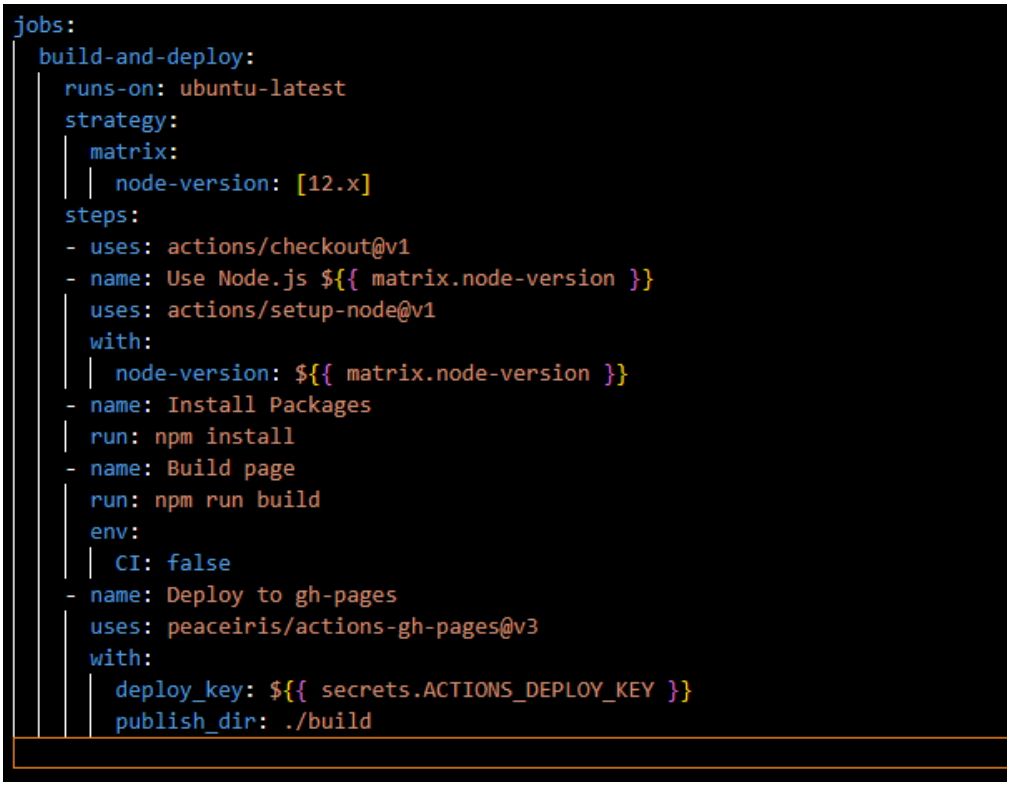

Ilustración 50. Fichero de configuración para la integración continua

En este fichero se establece el entorno y las dependencias necesarias para ejecutar la aplicación. Podemos observar que se parte del sistema operativo Ubuntu [61] (ubuntu-latest) y que se va a utilizar la versión de node 12.x (la última actualización dentro de la versión 12). Además podemos ver que, mediante la acción setup-node [62], se realizan las acciones npm install, para instalar todas las dependencias del proyecto, npm run build, para compilar una versión de producción de la aplicación y finalmente una acción de deployment, que utilizando la carpeta "build" y la acción actions-gh-pages [63] carga el contenido de la web resultante en la rama gh-pages, empleada para publicar la web.

La utilización de integración continua y despliegue continuo en nuestra herramienta permite que todos los cambios agregados a la rama principal del repositorio produzcan una nueva versión que será desplegada automáticamente, en lugar de tener que realizar manualmente esta tarea cada vez que necesitemos desplegar.

6. Conclusiones y Trabajos Futuros

El presente capítulo está dedicado a las conclusiones del proyecto, extraídas a lo largo de su desarrollo.

Se han analizado las tecnologías relacionadas y estado del arte: proveedores de datos Amazon S3, MinIO y Onedata, funciones: SCAR y OSCAR, librerías *frontend* para el diseño de *workflows*, y herramientas para la generación de *workflows*: Faas-flow, Amazon Step Functions y Azure Logic Apps.

Se han cumplido los objetivos planteados para el proyecto:

- Proveer al *framework* SCAR de una herramienta gráfica para la composición de flujos de trabajo, que puede ser utilizada también para el framework de OSCAR
- Permitir la exportación del flujo de trabajo al formato YAML.
- Permitir guardar y restaurar el estado de un *workflow*
- Distribuir la aplicación como una web estática a través de una infraestructura pública.

Por todo ello, se ha logrado crear una aplicación web que permite crear flujos de trabajo de forma visual y publicada de forma totalmente gratuita a través de la plataforma de GitHub Pages. Además, se ha establecido la integración continua para que los nuevos cambios se despliegan automáticamente a esta plataforma.

Cabe destacar que la realización del proyecto ha requerido un análisis de los framework SCAR y OSCAR, dado que el principal objetivo ha sido la integración con estos sistemas mediante la generación del fichero YAML que sirve como entrada a estos sistemas.

Como posibles mejoras y ampliaciones en el futuro del proyecto se propone:

- Proveer a la herramienta de autenticación mediante AWS Cognito
- Permitir importar un flujo de trabajo desde un fichero YAML.
- Permitir el envío de la función a ejecutar y el fichero YAML mediante una llamada POST HTTP a un endpoint del cliente SCAR para la creación de los recursos especificados en el fichero YAML.

Referencias

[1] Amazon. Amazon AWS. https://aws.amazon.com/es/ [Online; accessed 05-Abr-21]

[2] Microsoft. Azure. https://azure.microsoft.com/es-es/ [Online; accessed 05-Abr-21]

[3] Google. Google Cloud. https://cloud.google.com/ [Online; accessed 05-Abr-21]

[4] Amazon. AWS Lambda. https://aws.amazon.com/es/lambda/[Online; accessed 05-Abr-21]

[5] Google. Google Cloud Functions. https://cloud.google.com/functions?hl=es [Online; accessed 05-Abr-21]

[6] Microsoft. Microsoft Azure Functions. https://azure.microsoft.com/es-es/services/functions/ [Online; accessed 05-Abr-21]

[7] Apache. Apache OpenWhisk. https://openwhisk.apache.org/ [Online; accessed 05-Abr-21]

[8] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In Proceedings of the 1st International Workshop on Mashups of Things and APIs, MOTA '16, pages 5:1–5:4, New York, NY, USA, 2016. ACM.

[9] Alex Glikson. TRANSIT: Flexible pipeline for IoT data with Bluemix and OpenWhisk. https://medium.com/openwhisk/transit-flexible-pipeline-for-iot-data-with-bluemix-and-openwhisk-4824cf20f1e0. [Online; accessed 05-Abr-21]

[10] Apache. Apache Spark. http://spark.apache.org/ [Online; accessed 05-Abr-21]

[11] Google. Kubernetes. https://kubernetes.io/es/ [Online; accessed 05-Abr-21]

[12] Eric Jonas et al. Cloud Programming Simplied: A Berkeley View on Serverless Computing. In: (2019). arXiv: 1902.03383. url: http://arxiv.org/abs/1902.03383 (cit. on p. 30).

[13] Pérez-González, AM.; Moltó, G.; Caballer Fernández, M.; Calatrava Arroyo, A. (2018). Serverless computing for container-based architectures. Future Generation Computer Systems. 83:50-59. https://doi.org/10.1016/j.future.2018.01.022

[14] Docker. Docker. https://docker.com [Online; accessed 05-Abr-21]

[17] A. Pérez, S. Risco, D. M. Naranjo, M. Caballer and G. Moltó, "On-Premises Serverless Computing for Event-Driven Data Processing Applications," 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 2019, pp. 414-421, doi: 10.1109/CLOUD.2019.00073.

[15] Amazon. AWS Batch. https://aws.amazon.com/es/batch/ [Online; accessed 05-Abr-21]

[18] GRyCAP. SCARD Docs. https://scar.readthedocs.io/en/latest/prog_model.html [Online; accessed 05-Abr-21]

[16] Amazon. Amazon S3. https://aws.amazon.com/es/s3/ [Online; accessed 05-Abr-21]

[19] MinIO. MinIO. https://min.io/ [Online; accessed 05-Abr-21]

[20] Onedata. Onedata. https://onedata.org/#/home [Online; accessed 05-Abr-21]

[21] GryCAP. SCAR. https://github.com/grycap/scar [Online; accessed 05-Abr-21]

[22] Docker. Docker Hub. https://hub.docker.com/ [Online; accessed 05-Abr-21]

[23] Pérez, A., Moltó, G., Caballer, M., Calatrava, A., 2018. Serverless computing for container-based architectures. Futur. Gener. Comput. Syst. 83, 50–59. https://doi.org/10.1016/j.future.2018.01.022

[24] GryCAP. Repositorio OSCAR https://github.com/grycap/oscar [Online; accessed 05-Abr-21]

[25] GryCAP-I3M-UPV, Universitat Politècnica de València. EC3: Elastic Cloud Computing Cluster https://servproject.i3m.upv.es/ec3/ [Online; accessed 05-Abr-21]

[26] GryCAP-I3M-UPV, Universitat Politècnica de València. IM: Infrastructure Manager https://www.grycap.upv.es/im/index.php [Online; accessed 05-Abr-21]

[27] GryCAP-I3M-UPV, Universitat Politècnica de València. CLUES: cluster energy saving https://www.grycap.upv.es/im/index.php [Online; accessed 05-Abr-21]

[28] OpenFaaS. OpenFaaS. https://www.openfaas.com/ [Online; accessed 05-Abr-21]

[29] ProcessMaker. ProcessMaker. https://www.processmaker.com/es/ [Online; accessed 05-Abr-21]

[30] GitHub. GitHub Pages. https://pages.github.com/ [Online; accessed 05-Abr-21]

[31] GitHub. GitHub. https://github.com/ [Online; accessed 05-Abr-21]

[32] Google. Angular. https://angular.io/ [Online; accessed 05-Abr-21]

[33] Vue.js. https://vuejs.org/ [Online; accessed 05-Abr-21]

[34] Facebook. React.js https://es.reactjs.org/ [Online; accessed 05-Abr-21]

[35] npm. https://www.npmjs.com/ [Online; accessed 05-Abr-21]

[36] Ryan Dahl. Node.js. https://nodejs.org/es/ [Online; accessed 05-Abr-21]

[37] Microsoft. TypeScript. https://www.typescriptlang.org/ [Online; accessed 05-Abr-21]

[38] Netvariant. Moqui Workflow designer. https://github.com/Netvariant/workflow-designer [Online; accessed 05-Abr-21]

[39] Vue Simple Flowchart. https://github.com/Jeffreyrn/vue-simple-flowchart [Online; accessed 05-Abr-21]

[40] Vue Simple Flowchart demo. https://jeffreyrn.github.io/vue-simple-flowchart/demo/ [Online; accessed 05-Abr-21]

[41] Flowchart Vue. https://github.com/joyceworks/flowchart-vue [Online; accessed 05-Abr-21]

[42] Flowchart Vue demo. https://joyceworks.github.io/flowchart-vue/?web=1&wdLOR=c6ADCF58B-BFCB-46FD-9086-877B5DCC8AC3 [Online; accessed 05-Abr-21]

[43] React Flow Chart. https://github.com/MrBlenny/react-flow-chart [Online; accessed 05-Abr-21]

[44] React Flow Chart Storybook. https://mrblenny.github.io/react-flow-chart/?path=/story/state--internal-react-state [Online; accessed 05-Abr-21]

[45] React Flow Chart Functional Component issue. https://github.com/MrBlenny/react-flow-chart/issues/208 [Online; accessed 05-Abr-21]

[46] Faas-flow – Function Composition for OpenFaaS. https://github.com/s8sg/faas-flow [Online; accessed 05-Abr-21]

[47] Amazon. AWS Step Functions. https://aws.amazon.com/es/step-functions/ [Online; accessed 05-Abr-21]

[48] Azure. Azure Logic Apps. https://azure.microsoft.com/es-es/services/logic-apps/ [Online; accessed 05-Abr-21]

[49] Slack. Slack. https://slack.com/intl/es-es/ [Online; accessed 05-Abr-21]

[50] Slack. Slack help center. https://slack.com/intl/es-es/help/articles/360035692513-Gu%C3%ADa-del-creador-de-flujos-de-trabajo [Online; accessed 05-Abr-21]

[51] XTech. Ant Design. https://ant.design/ [Online; accessed 05-Abr-21]

[52] js-yaml. https://www.npmjs.com/package/js-yaml [Online; accessed 05-Abr-21]

[53] git. https://git-scm.com/ [Online; accessed 05-Abr-21]

[54] Atlassian. Bitbucket. https://bitbucket.org/ [Online; accessed 05-Abr-21]

[55] Atlassian. Bitbucket pipelines. https://bitbucket.org/product/es/features/pipelines [Online; accessed 05-Abr-21]

[56] GitLab. GitLab. https://about.gitlab.com/ [Online; accessed 05-Abr-21]

[57] GitLab. Gitlab Continuous integration. https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/ [Online; accessed 05-Abr-21]

[58] GitHub. GitHub Actions. https://github.com/features/actions [Online; accessed 05-Abr-21]

[59] GRyCAP. GitHub fdl-composer. https://github.com/grycap/fdl-composer [Online; accessed 05-Abr-21]

[60] GRyCAP. Fdl-composer https://grycap.github.io/fdl-composer/ [Online; accessed 05-Abr-21]

[61] Canonical Ltd. Ubuntu. https://ubuntu.com/ [Online; accessed 05-Abr-21]

[62] GitHub Action setup-node. https://github.com/actions/setup-node [Online; accessed 05-Abr-21]

[63] GitHub Action actions-gh-pages. https://github.com/peaceiris/actions-gh-pages [Online; accessed 05-Abr-21]

Anexos

I. Glosario

A

Arquitectura sin estado

Mediante una arquitectura sin estado, comúnmente conocida como *stateless*, el software solo depende del almacenamiento en terceros. Todos los datos necesarios se obtienen de un origen que mantiene el estado, comúnmente conocido como *stateful*.

AWS

Amazon Web Services (AWS) es una colección de servicios de computación en la nube (SaaS), ofrecidas por Amazon.

C

Contenedor

Un contenedor se trata de virtualización ligera, independiente del sistema en el que se hospede, que permite la ejecución de aplicaciones en su interior conteniendo todas las dependencias necesarias para la ejecución de la aplicación.

Cloud deployment

Se trata del despliegue en la nube, la aplicación en su totalidad se despliega en una infraestructura gestionada por terceros.

D

DevOps

DevOps (acrónimo de development -desarrollo- y operations -operaciones-) es una metodología que busca lograr la automatización y la supervisión del ciclo de vida del software, desde su implementación a la monitorización de su ejecución.

Drag-and-drop

Un sistema drag-and-drop permite arrastrar y soltar elementos dentro de una interfaz.

Docker

Docker es un proyecto de código abierto que permite el despliegue de aplicaciones dentro de contenedores.

E

Elasticidad

Se define elasticidad como la capacidad de un sistema para adaptarse a la carga de trabajo. Es deseable, en términos económicos, tener un sistema que aumente la capacidad de cómputo cuando hay un pico y no consuma nada cuando no lo haya.

F

Funciones como servicio

Comúnmente conocido como *Serverless services* o *Function as a Service (FaaS)* proveen de un ecosistema que permite la ejecución de funciones bajo demanda, en cualquier momento y sin que el usuario tenga que configurar un entorno, habitualmente se trata de servicios que no requiere un coste fijo, sino que se tarifican mediante un modelo de pago por uso.

H

Hybrid deployment

Esta opción de despliegue combina las opciones On-Premises deployment y Cloud deployment, y se caracteriza por mantener los datos dentro de los límites de la organización.

I

Imágenes de contenedor

Una imagen de contenedor es una foto estática del estado de un contenedor, que puede ser utilizada como base para construcción de instanciación de un contenedor.

Infraestructura como servicio

Una infraestructura como servicio, en inglés Infrastructure as a Service (IaaS), es una forma de computación en la nube que proporciona virtualización de recursos informáticos a través de internet. En el modelo IaaS, el proveedor proporciona los componentes informáticos típicos (servidores, almacenamiento y red).

K

Kubernetes

Sistema de código libre para la orquestación de servicios y el ajuste de escala de estos.

M

Microservicio

Estilo arquitectónico que busca la construcción de aplicaciones ligeras con una única responsabilidad.

O

On-Premises deployment

Se trata de una modalidad de despliegue tradicional, donde la aplicación se aloja en la infraestructura del cliente.

Orquestación

La orquestación mediante herramientas permite habilitar un ambiente para la elasticidad del sistema, el uso óptimo de recursos y la tolerancia a fallo.

P

Plataforma como servicio

Una plataforma como servicio, en inglés Platform as a Service (PaaS), permite al usuario desplegar software, con una mínima configuración, sin necesidad de mantener infraestructura.

T

Tolerancia a fallos

La tolerancia a fallos es una propiedad de un sistema que permite a éste continuar operando en caso de fallo.

S

Software como servicio

Modelo de distribución de software donde, tanto la aplicación como la infraestructura, es controlada por quien oferta el servicio.